HIVE

Register & Stack Hybrid ▪ 32-Bit ▪ Multi-Threaded ▪ Barrel Pipelined
SystemVerilog Soft Processor Core

INTRODUCTION

Hive is a general-purpose soft processor core intended for instantiation in an FPGA when CPU functionality is desired but when an ARM or similar would be overkill. The Hive core is complex enough to be useful, with a wide data path, a relatively full set of instructions, and high code density and ALU utilization – but with very basic control structures and minimal internal state, so it is simple enough for humans to easily grasp and program at the lowest level without any special tools. It fits in current low end FPGAs with sufficient resources left over for peripherals and other unrelated logic, and operates at or near the top speed of the device DSP hardware.

Hive isn't an acronym, instead the name is meant to suggest the swarm of activity in an insect hive: many threads sharing the same program and data space, individually beavering away on separate tasks, and cooperating together to accomplish larger goals. Because of the shared memory space, thread intercommunication is facilitated, and threads can all share single instances of code, subroutines, and data sets which enables code compaction via global factoring.

The novel hybrid stack / register construct employed reduces the need for a plethora of registers which allows for small operand indexes in the opcode. This construct, coupled with explicit stack pointer control in the form of a pop bit for each stack index, minimizes the confusing and inefficient stack gymnastics (swap, pick, roll, copy to thwart auto-consumption, etc.) normally associated with conventional stack machines, and minimizes the saving and restoring of register contents at context switch points.

Hive employs a naturally emergent form of multi-threaded scheduling which eliminates all pipeline hazards and provides the programmer with as many equal bandwidth threads – each with its own independent interrupt – as pipeline stages. Processors which employ this form of pipelining are classified as *barrel processors*.

Hive is a largely stateless design (no pipeline bubbles, no registered ALU flags that may or may not be automatically updated, no reserved data registers, no pending operations, no branch prediction, etc.) so subroutines require no overhead, interrupts consume a single vector cycle, and their calculations can be performed directly and immediately with almost complete disregard for what may be transpiring in other contexts.

This paper presents the design of Hive along with some general background. Even if you don't find the architecture of this core to your liking, you may possibly find something else of use.

HIVE FEATURES LIST

- A simple, compact, relatively stateless, high speed, barrel pipelined, multi-threaded, little endian design based on novel RASH^{TM} (**R**egister **A**nd **S**tack **H**ybrid) technology.
- 8 independent, isolated, general purpose LIFO data stacks per thread with parameterized depth and fault protections.
- 8 strictly equal bandwidth threads.
- 8-stage pipeline with no stalls or hazards.
- 32-bit data path / ALU with extended width arithmetic results.
- 2 operand machine with operand select and stack pointer control fields in the opcode.
- 16-bit compact opcode. 16 & 32 bit memory data access widths, both aligned and unaligned.
- 8 fully independent internal / external interrupts with no hierarchical limitations (one per thread).
- 32-bit internal register set in separate I/O space with highly configurable base register module that may be easily modified / expanded to provide coprocessor interfacing, enhanced I/O, detailed debug, etc.
- All instructions execute in a single thread cycle, including 32 x 32 = 64 bit signed / unsigned multiply (normal or extended).
- Common data & instruction memory space (Von Neumann architecture) enables dynamic code / data partitioning, combined code and data constructs, code copy & move, etc.
- All threads share the entire common data / code space, which facilitates global data / code factoring and thread intercommunication.
- Variable width address (set via a build time parameter) – up to 32 bits of directly addressable space.
- Double buffered UART with BAUD generator and several parameterized options.
- 32 bits of general purpose I/O.
- Written in 100% highly portable SystemVerilog (no vendor specific or proprietary language constructs) and partitioned into easy to understand and verify modules.
- May be programmed via a SystemVerilog initial text file, no complex tool chain is necessary.
- Achieves aggregate throughput of ~200 MIPS in a bargain basement Altera EP3C5E144C8 (Cyclone 3, speed grade 8, the target device for initial development) while consuming ~2500 logic elements, or <50% of the FPGA fabric.
- Free to use, modify, distribute, etc. (but only for the greater good, please see the copyright).

CONTENTS

MOTIVATION

As a (mostly) digital designer who works primarily in FPGAs, I am often on the lookout for simple processor cores because projects often underutilize the hardware due to low data rates (e.g. a UART, or a sampled audio stream). If latency isn't a big issue, then why not multiplex the high-speed hardware with a processor construct? But the core needs to be simple, not consume too much in the way of logic (LUTs, block RAMs, multipliers), have compact op codes (internal block RAM isn't limitless nor inexpensive), keep the ALU sufficiently busy, and be easy to program at the machine code level without the need for a compiler or even an assembler.

FPGA vendors have off-the-shelf designs that are quite polished and bug-free, but they, and therefore the larger design and the designer, are generally legally chained to that vendor's silicon and tool set. There are many free cores available on the web, but one may end up getting exactly what one paid for.

The Hive core is my offering for this problem area. The essentially free and naturally emergent multi-threading / rigid scheduling mechanism in Hive isn't unique; I believe it was implemented as far back as 1964 on the CDC 6000 series peripheral processors. Hive bit shift distances are treated as signed which works out rather nicely, but the ancient PDP 10 did this as well. The notion of multiple stacks isn't original, nor is the explicit control over the processor stack pointer, but I believe the register/stack hybrid as implemented and described here (indexed stacks, top-entry-only conservative access with pop bit override) is something relatively new. And the way extended arithmetic results are dealt with uniformly in Hive may possibly be somewhat novel as well. But who knows? Processors have been around long enough that most of the good ideas have been mined out and put to the test in one form or another, which makes it difficult / unlikely to bring something fundamentally new or innovative to the table.

REGISTER MACHINES VS. STACK MACHINES

Most modern processors are register based, and so have some form of register set tightly bound to the ALU – a tiny fast triple port memory in a sense. This conveniently continues the memory hierarchy of faster and smaller the closer to the core, and has the advantage of being a mature target for compilers.

Many registers are generally available because the register space grows exponentially with register address width. However, register opcode indexes still consume significant opcode space, particularly in a 3 operand machine, and register count is a limited resource that doesn't scale with the rest of the design. Registers are often reserved for special purposes, and some may be invisible to non-supervisory code. It would seem the more registers available, particularly of the "special" variety, the more the programmer has to juggle in his/her head. In addition, a general-purpose register may only be used if the programmer is certain that any data there is globally moot, or if the register contents are first saved to memory and later restored, which is something else to keep track of.

Since my first exposure to data stacks via an HP calculator (won in a high school engineering contest), I have been fascinated with stack languages and stack machines. With no explicit operands, a data stack, a return stack, and almost no internal state, a stack machine can have incredibly compact op codes - often 5 bits will do. Interrupts, subroutines, and other forms of code factoring can be quite efficient due to the stacked registers; all that is required is that they clean up after themselves. I have studied many of these, and have coded a few of my own and had them running on an FPGA demo board. They are surprisingly easy to implement but surprisingly cumbersome to program - one has to stick loop indices, conditional test values, and branch addresses under the operands on the stack or in memory somewhere, so there are a lot of operations and much real time wasted on stack manipulation which can get very confusing very quickly. Laborious hand optimization of stack code leads to "write only" procedural programs that are difficult to decipher later, and with catastrophic stack faults all too likely. The tiny opcode widths produce a natural instruction caching mechanism, but having multiple opcodes per word is awkward when they aren't powers of 2 wide, a nuisance when one must manually change the code by hand (one usually ends up inserting no-ops to pad out the space), and interrupts / subroutines must either return to a word boundary (more no-ops / wasted program space) or the hardware must somehow store and retrieve a sub index into the return word (more state).

Stack machines are often portrayed (perhaps inadvertently) as a panacea for computing ills, but with little in the way of formal analysis to back up these assertions. They are something very different and on the fringe and as such don't get addressed by the mainstream, so there aren't many technical comparisons (speed, code density, etc.) to more conventional architectures – or detractors for that matter, so the stack machine noob encounters a situation rather like serving on a jury and hearing only the defendant's side of the case. My conclusion is their biggest strength – implicit operands – is also their biggest weakness. One has to follow the intricate stack manipulations closely and with a very clear idea of what the programmer originally had in mind in order to make any sense of the code. One cannot rely on, say, the loop index residing and staying put in register 4 and the like. There are of course stack machines out there that have register sets tacked on, but this tends to complicate the hardware and bloat out the opcodes, which does not strike me as a very elegant solution.

Another thing that isn't discussed much regarding stack machines is that auto consumption of *all* input values is generally necessary. While it is obvious that ALU operations pop the input operand(s) and push the result, what isn't emphasized is that conditional branches generally consume the branch test value(s) and the branch address or address offset regardless of whether the branch is taken or not. Auto consumption is an issue because it leads to copying or restoring of values to be used both now and later, and it also means most instructions cannot be made individually conditional (ala the ARM, or via a skip instruction) because the stack pointer(s) will likely be different depending on whether the instruction was executed or not, something the programmer can't generally track.

Others may reasonably disagree, but my own conclusion is this: a stack is a good fit for data input and intermediate results manipulation on a scientific calculator. However, even with the inclusion of a second dedicated return stack, pure stack machines are not such a great paradigm on which to base processor hardware or programming languages. The indexed register set is simply too powerful and useful a concept to leave by the wayside.

BACKGROUND : LIFOS

Since this paper is about a hybrid stack machine, it helps to understand stacks themselves, which are LIFO (**L**ast **I**n **F**irst **O**ut) constructs.

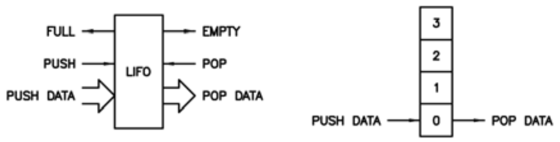


Figure 1. LIFO symbols.

The figure above shows two LIFO symbols, the one on the left is I/O centric, the one on the right more of a schematic memory view. Unlike FIFOs, which need separate read and write side pointers, LIFOs only require a single pointer, which may implemented in such a way as to conveniently reflect the fullness of the LIFO. The push side is only concerned with whether the LIFO is full or not, the pop side is only concerned with whether it is empty or not. Push when full is an error because (depending on the stack protection logic) this action either drops the input data on the floor or corrupts data in the LIFO along with the LIFO pointer. Pop when empty is an error because it gives false read data and (depending on the stack protection logic) may corrupt the LIFO pointer.

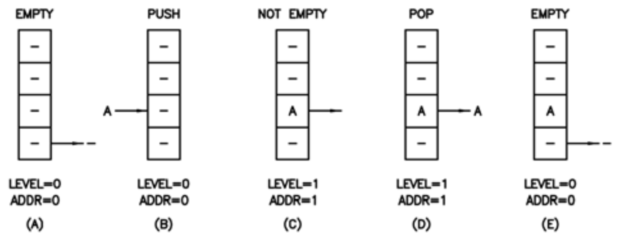


Figure 2. LIFO stack operations – push then pop from empty state.

The figure above shows LIFO operation from empty, to not empty, to empty again. Note that the first write to memory is address 1 rather than address 0, which may seem a bit counter-intuitive. This convention allows the level and pointer values to be the same.

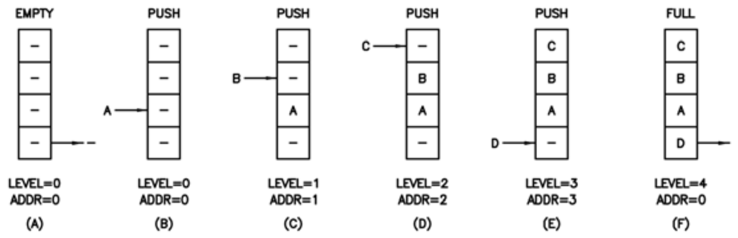


Figure 3. LIFO stack operations – push from empty state to full state.

The next figure shows LIFO operation from empty to full. Note that the last write to memory is at address 0, which may also seem a bit counter-intuitive. It helps here to think of the address as a modulo (i.e. the MSB is removed from) of the level value. For this 4-deep LIFO there are actually five distinct states corresponding to levels 0 through 4. Indeed, when fully utilizing the LIFO memory space there will always $2^n + 1$ levels, and it is easiest and most straightforward to handle them with an extra MSB in the level counter, and present all but the MSB of this counter to the LIFO memory address input (i.e. the stack pointer). This arrangement gives us $2^n - 1$ unused states, but they are fairly easy to decode: full is indicated by a set MSB, empty indicated by all bits zero.

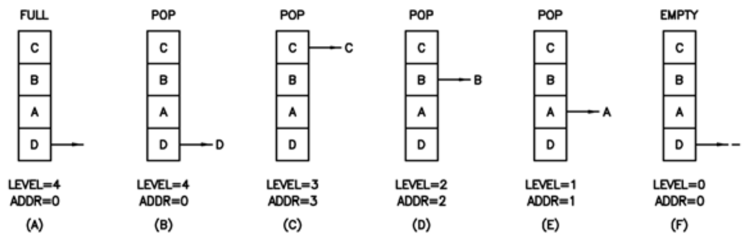


Figure 4. LIFO stack operations – pop from full state to empty state.

The figure above shows the previously filled LIFO operation from full to empty. At the end (in this case) the value D at memory location 0 is presented as output, but the pop side should be keeping track and so know not to use it.

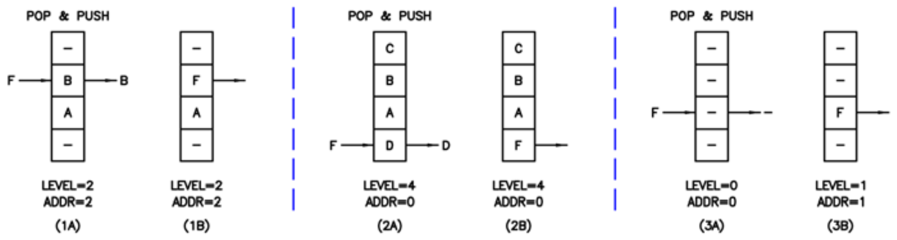


Figure 5. LIFO stack operations – three pop & push scenarios.

What happens if we pop and push at the same time? For a canonical stack machine, we need to read the pop side value, pop it off the stack, and then push the new value onto the stack. This is a pop & push (as opposed to a push & pop, which would be nonsensical for this application). At the above left we see a pop & push in action, the value B at address location 2 is overwritten with the value F, and there is no net pointer change. In the center we see a pop & push when full, which is not an error because pop, which decrements the pointer, can be thought of as preceding push, which increments the pointer. Finally, on the right we see a pop & push when empty. This is obviously a pop error because the read data is invalid – but it is a pop error only! If the pointer is internally protected from corruption then the correct net result is a push.

Stack Protection

Is it always best to protect the stack against the corruption of the pointer or memory contents? I believe that the answer to this is always "yes" – when pointer corruption occurs fullness tracking is lost and so the associated error reporting is confounded. Pop (underflow) protection is clearly advantageous because it prevents the stack from rolling under and thereby offering up completely unrelated, non-local data and addresses to the thread.

On the other hand, push (overflow) protection creates a stuck stack which limits the ability of a thread to fix its own problems. Would it be better to *not* protect against push errors, and just let them corrupt the first stack entries so the thread could continue? Granted this kicks the problem down the road, but perhaps the thread wasn't going to use the earlier entries on the stack anyway and was about to issue a routine stack or thread clear? Perhaps it was about to check itself for stack errors and if it found one would have cleared itself? At least its not possibly derailed, off corrupting the contents of main memory.

Contemplating how to deal with these "what if" conditions that should not happen (but likely will, at least during SW development) can drive you a little crazy. In any case, pop and push protections are individually configurable build options in Hive so you can set them however you like. Regardless of the protection settings, stack errors are always reported to the local register set.

REGISTER / STACK HYBRID

Many register-based machines have a return stack, and many stack machines have a one or more registers stuck somewhere, but beyond this, could there be a more harmonious middle ground between stack based and register based machines? If a register based machine were designed with a LIFO stack under each register, then perhaps the programmer could accomplish the same goals with fewer indexed register locations, meaning the register index could be made narrower, giving a more compact and efficient opcode. Multiple stacks would be more convenient than a single stack for complex algorithms, and would minimize inefficient and confusing stack thrash. Unlike register count, LIFO depth can easily scale as required by other aspects of the design. Could the stacks indeed be indexed as register operands? If so, how might multiple stacks be implemented and how would the stack push/pop mechanism behave?

I recently encountered the J1 stack based processor (http://www.excamera.com/sphinx/fpga-j1.html) which is quite intriguing in that it has a two bit wide signed stack pointer increment field in the opcode. This idea inspired me to investigate explicit rather than implicit stack control. I decided that an array of simple stacks, where only the top stack values are presented to the ALU (as opposed to the top and second values as in a conventional stack machine) would suffice. The stacks could then be indexed normally as register locations, with the usual one or two sources and one destination. I then came up with a simple, inherently conservative stack mechanism: whenever anything is read from a stack, the stack value and stack pointer remain unchanged. Whenever anything is written to a stack, the top stack value is pushed in to make room for the new value. Each stack index is provided with an associated pop bit to alter this default conservative behavior:

pop bit	read / write	Stack	Behavior
0	read	no change	Register type read.
1	read	pop	Stack type read.
0	write	push	Stack type write.
1	write	pop & push	Register type write.

Figure 6. Hybrid register / stack behavior.

This arrangement accommodates the full range of stack and register behaviors. To illustrate this, say the operand source of an ALU single operand operation is stack index B and the result destination is stack index A:

Case	B pop	A pop	B stack	A stack	Behavior
0	0	0	no change	push	Register type read, stack type write.
1	0	1	no change	pop & push	Register type read & write.
2	1	0	pop	push	Stack type read & write.
3	1	1	pop	pop & push	Stack type read, register type write.

Figure 7. One and two operand hybrid register / stack behavior.

Cases 1 and 2 respectively give the normal pure register and pure stack behaviors, while cases 0 and 3 give useful variations. What about the two input operand case? Say the primary input operand is stack index A, the secondary input operand is stack index B, with the result going to stack index A (e.g. a two operand opcode architecture). It turns out that the same table above works for this scenario as well. How do we handle the case where both of the sources and the destination point to the same stack? The solution is to simply OR the two pop bits together. Remember that there is no access to the value below the top LIFO entry as in most stack machines, so when index A = index B for a two operand instruction such as multiply, the result will be A^2 pushed to A. And in this case, if both of the A and B pop bits are set this won't cause a double pop because the pop bits are simply ORed, causing a single pop of A (a pop & push, actually).

Now that we have simpler stacks and more control over them, the conditional execution of single operations is a viable option. Conventional stack machines generally do not have conditional single operations because operands are always consumed – the programmer would not be able to tell how many items were left on the stack after a conditional two operand operation, which would lead directly to stack faults. With no auto-consumption of the input, and by setting the pop bit of the register being conditionally written to, we can ensure the stack pointers do not change during a single conditional operation.

BASIC DESIGN DECISIONS

Operands
How many operands should be in the opcode? I picked 2 to keep the opcode small, so Hive is a 2 operand machine. Here are the rules:

- For single input ALU operations, the source is B and the result destination is A. For example: A=not(B).
- For two input ALU operations, the primary source is A, the secondary source is B, and the result destination is A. For example: A=A-B.
- For single input conditional branch statements, A is tested against zero. The signed address increment is B or is supplied as immediate data. For example: PC=(A>0)?PC+B:PC.
- For two input conditional branch statements, A is tested against B. The signed address increment is supplied as immediate data. For example: PC=(A!=B)?PC+I:PC.
- For memory reads, the base address is B, the read value is written to A, and there is an immediate 4 bit positive address offset. For example: A=mem(B+I).
- For memory writes, the base address is B, the write value is read from A, and there is an immediate 4 bit positive address offset. For example: mem(B+I)=A.
- For internal register set access, the absolute address is I and the data is read from or written to A. Read example: A=reg(I). Write example: reg(I)=A.
- For subroutines, the subroutine absolute address is B and the return address (the PC) is pushed to A.
- When an interrupt is taken the return address (i.e. the non-incremented PC) is automatically pushed to stack 0 – making it a bit of an odd man out, but it is the only "special" stack, and this is the only way in which it is "special".

Therefore, A is the primary data source and destination for two operand operations, is the primary data tested, receives subroutine return addresses, and is the only thing that can be written to. B is the primary data source for one operand operations, the secondary data source for two operand operations, is the secondary data that A is tested against, and always provides the address or address offset.

Stacks & Stack Depth
How many stacks are needed? I picked eight. This gives a convenient hex nibble field of 4 bits for each operand (one pop bit and three stack index bits) for a total of 8 bits of opcode consumed. How deep should the stacks be? I have read that 32 entries are deep enough for single stack machines to not require auto spill-to-memory mechanisms and the like. Since we have eight stacks, and since coding for this core is likely to be done by hand, we could doubtless get by with less depth. In any case, the use of FPGA block RAM for the stacks sets a generous practical lower limit (32 entries per stack per thread in our target device, set via a build-time parameter).

ALU Data Width
Non-power-of-2 widths can be excluded for efficiency reasons, byte data has too little resolution for most cases, 16-bit data can store audio PCM and Unicode text efficiently but it doesn't have sufficient resolution to directly perform the internal computations required for audio DSP, and 64-bit data is overkill for most applications that would be running on a small FPGA processor. Which leaves us with 32 bits. Data width directly dictates the top speed vs. pipelining depth because wider data requires more deeply cascaded combinatorial logic to perform adds, multiplies, etc.

Opcode Width
Hive opcodes are a compact and efficient 16 bits wide, which means the PC is an index into a 16-bit wide memory. With careful planning and some field reuse there is sufficient room for operand indices and small immediates.

Main Memory Data Width
Hive memory access width, and by that I mean main memory read / write data and in-line literals, can be either 16 or 32 bits, depending on the operation. Both aligned and non-aligned 32-bit accesses are supported.

Main Memory PC & Address Width
PC and address width are parameterized and so are set at build-time. PC width may be set to coincide with address width, or wider if so desired, up to and including the ALU data width. Address width directly sets the

depth of the main memory instantiation and BRAM resource usage (note that deeper settings may negatively impact the top speed of the core).

Arithmetic Results Width
Some ALU arithmetic operations invariably produce wider results than the input operands. Traditional processors stick the extended results of add and subtract (carry, overflow, sign, etc.) in dedicated bit flag registers, and then have rules and special instructions that govern the updating, clearing, saving, and restoring of them. The results of full width multiplies are usually sent to special concatenated register pairs. These practices may be efficient, but they introduce complexity and internal state.

A simple and uniform method of handling wide arithmetic results is to treat them as double width regardless of operation (add, subtract, multiply) and select either the lower (i.e. normal) half of the result or the upper (i.e. extended) half of the result via instructions. The obvious downside here is that obtaining the full width result takes at least two cycles even when the operation is actually performed in one. For the full result, it may seem wasteful to perform the same internal calculation both times, but one probably should not think of this as major effort for the ALU or as a huge opportunity lost. All processors have to perform a full width subtraction in order to generate the arithmetic comparison flags between two numbers. By examining the extended result of add / subtract first one can know beforehand if the result will overflow and perhaps not perform it (e.g. restoring division), and often only the lower or extended arithmetic result is required.

Interestingly, the extended results of signed and unsigned subtraction and signed addition always form a convenient all ones or all zeros flag (easily negated with a NOT instruction). The extended result of unsigned addition is a bit more complex. Here are some 4 bit corner case examples to get a flavor of how this works:

```
+ unsigned  15  + 15  =  30  = 0001,1110   max
             0  +  0  =   0  = 0000,0000   min

 + signed    7  +  7  =  14  = 0000,1110   max
            -8  + -8  = -16  = 1111,0000   min

- unsigned  15  -  0  =  15  = 0000,1111   max
             0  - 15  = -15  = 1111,0001   min

 - signed    7  - -8  =  15  = 0000,1111   max
            -8  -  7  = -15  = 1111,0001   min

* unsigned  15  x 15  = 225  = 1110,0001   max
             0  x  0  =   0  = 0000,0000   min

 * signed   -8  x -8  =  64  = 0100,0000   max
             7  x -8  = -54  = 1100,1000   min
```

Figure 8. 4 bit input / 8 bit result corner cases.

Signed vs. Unsigned Arithmetic
Addresses are generally thought of as unsigned, but unsigned subtraction will produce negative numbers whether one likes it or not. The programmer obviously needs the resources to handle both, so the impact of signed vs. unsigned arithmetic is in deciding how to handle both, determining what will be considered the default behavior, and instruction naming conventions. Signed multiply is more basic due to sign/zero extension needs (hence Altera's FPGA multiply hardware primitives being signed). I feel that a signed half-width memory read can sometimes be more useful than unsigned because it influences the MSBs above. Given the way that Hive deals with extended results, lower arithmetic operations are sign neutral (i.e. give the same results regardless of signed / unsigned operation) so only the right shift operations and the arithmetic operations that produce extended results need to be differentiated with respect to sign.

Endianness
Hive can't be big endian because that would require byte addressing. That and it's asinine (see appendix).

BACKGROUND : FPGA RESOURCES

The available physical resources and their detailed behavior, limitations, and timing characteristics in the target FPGA will strongly influence the top speed, size, and other important bulk metrics of any soft processor. One may as well exploit these resources up front rather than be stymied by them later.

Block RAM (BRAM)

The primary FPGA component the soft processor designer needs to understand is block RAM.

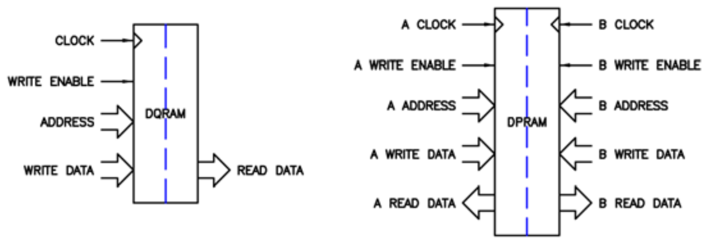


Figure 9. Block RAM: simple (DQ) on left, true dual port (DP) on right.

The figure above shows two common forms of block RAM: a simple dual port (DQ) on the left and a true dual port (DP) on the right. Because it uses a single address, the DQ variant is a good fit for the LIFO stacks. The DP variant is useful for main memory as it gives two independent accesses, which enables a data read / write along with instruction fetch per cycle (thus sidestepping the “Von Neumann bottleneck”). Main memory access is a huge driver in any processor design, and often the limitation encountered is insufficient address ports rather than data ports.

Block RAM resources have configurable variable widths, from some maximum down to a single bit. For widths of eight and above an additional bit per byte (8+1, 16+2, 32+4) is provided for out-of-band signaling, individual byte enables, CRC, error correction, and other common uses. I feel that it is a mistake to employ these extra bits in order to increase the data width of the ALU or instructions, as this precludes the efficient use of conventional 2^n width memory to store internal data / program information.

The two ports of a dual port block RAMs have independently variable aspect ratios. The only sane address mapping that can accommodate this arrangement is little endian.

Width	Index / Address																															
1	31	30	29	28	27	26	25	24	23	22	21	10	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
2	15		14		13		12		11		10		9		8		7		6		5		4		3		2		1		0	
4	7				6				5				4				3				2				1				0			
8	3								2								1								0							
16	1																0															
32	0																															

Figure 10. FPGA Block RAM port aspect ratio address mapping.

As shown above, the address of a bit in the FPGA block RAM is simply the bit address. The address of a two bit entity is the address of its LSb divided by 2. The address of a four bit entity is the address of its LSb divided by 4, and so on. So the address of 2^n bit wide data is the bit address with n address bits lopped off the right end.

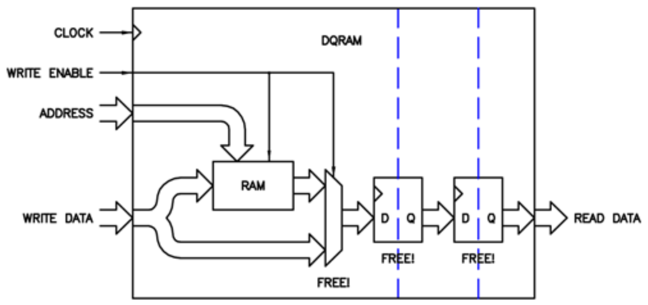


Figure 11. Block RAM internal resources.

What resources are available within block RAMs? The figure above shows a schematic view of the “inside” of a typical DQ RAM, though it applies to each side of a DP RAM as well. Even though FPGA block RAMs are *always* fully synchronous, it is sometimes helpful to think of the base RAM entity inside of the block quasi asynchronous. This RAM entity is supplemented with bypass logic in the form of a multiplexer, which enables two types of configurable (at build time) read-during-write behavior. Without the multiplexer, a read-during-write delivers the old memory data to the read data port (I have dubbed this *RAW – Read Ahead of Write* or *Read And then Write*) also known as *read-first* mode. With the multiplexer, a read-during-write conveys the data being written to the read data port (*WAR – Write Ahead of Read* or *Write And then Read*) also known as *write-through* mode. Note that these modes only apply to a *given* port of a DP RAM, read/write behavior between ports is never write-through (RAW, not WAR). The register following this optional multiplexer is *always* present making the output synchronous. Following this is yet another register; it is optional and generally part of the block RAM circuitry because it can dramatically speed up read clocking at the expense of one additional clock of latency.

In terms of read-during-write behavior, Hive needs write-through mode (WAR) for the LIFO stacks to function correctly. This mode is unimportant for the main memory however because we will never be simultaneously reading from and writing to the data port, and the instruction fetch port is read-only. In terms of speed, the write side can often tolerate a bit of combinatorial logic in front, while the read side is fairly slow if the additional output register isn’t used. So if our architecture can tolerate the latency of the additional read side output register we should certainly use it because it speeds things up and is essentially free.

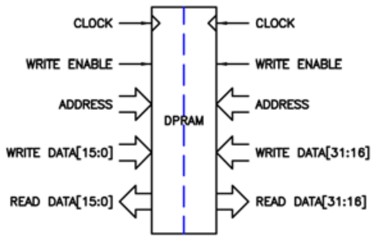


Figure 12. True dual port block RAM utilized as DQ RAM.

There is a way to convert DP RAM to DQ RAM, and this is shown in the figure above. Feeding the same clock, address, and write enable to both sides, along with splits / concatenations of the read / write data, accomplishes this simple transformation. In fact the tool will do this automatically when necessary. For our target Cyclone 3 device, DP data ports are limited to a maximum of 16 (+2) bits wide, and DQ data ports to a maximum of 32 (+4) bits wide – and the 1:2 ratio of these width limits makes sense given the above transformation. Since our LIFO stacks can employ DQ RAM (due to the single pointer / single address port) we can make them 32 bits wide using a single device.

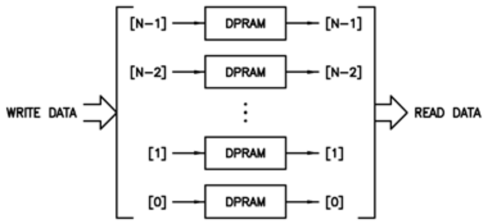


Figure 13. Block RAMs combined via bit-slice.

We may need our main memory to be considerably larger than a single 9k bit block RAM found in our target device. The tool will automatically combine multiple block RAM devices together, and often with no speed decrease – how does it accomplish this? The trick to making the largest and fastest block RAM amalgam is to configure the block RAMs to be one bit wide and maximum depth, 8k in this case, and then simply split / concatenate the write / read data by bit slicing the blocks together. Going above this size requires write enable steering and output multiplexing, which will also be inserted automatically by the tool when needed, but this extra logic tends to slow things down, particularly on the read side (though pipelining this logic could certainly get it back up to speed).

DSP Hardware
Since even quite low-end FPGAs these days have fairly fast hardware multipliers in some form of a DSP block, we should undertake any new designs with the knowledge and trust that they will be there. There is little point in leaving multiply operations out of our instruction set, and no point in trying to outsmart the FPGA manufacturers by constructing what would inevitably be slower and larger multipliers out of shifters, adders, etc. – both of which would needlessly strand this valuable resource. Therefore, it behooves us to understand the dedicated multiply hardware.

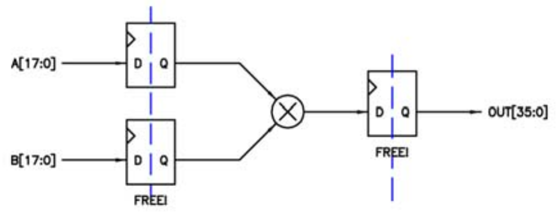


Figure 14. Signed multiplier hardware typically found in an FPGA.

Basic hardware multiplier width is 18 bits, which follows the convention of block RAM widths (2^n + 1 extra bit per byte). Being a full multiplier, the result is obviously double this, or 36 bits wide. As with add hardware, leaving some of the MSBs or LSBs unused will allow the remaining utilized multiply hardware to run faster due to fewer carry propagations, etc.

Altera multiplier blocks are signed by default, which makes sense because this convention simplifies sign extension of the inputs. In order to make a signed multiplier perform unsigned math, all that is necessary is to construct it one MSB wider at the inputs and force those MSBs to zero (zero extension). Conveniently, this same construct can be used to do signed multiplication simply by driving these MSBs with the signs of the inputs (sign extension). This of course requires an extra bit and therefore negatively impacts top speed slightly. The extra output MSBs generated with this scheme are unused (left unconnected) which may generate tool linter warnings if not manually limited in the code.

The multiplier hardware can be used in a purely combinatorial sense, but registering will speed it up considerably so manufacturers provide "free" internal registers at the inputs and outputs that are not part of the general FPGA fabric. As in the case of block RAM output registers, if our architecture can tolerate the latency of the additional multiplier I/O registering we would be crazy not to use it. This leads one almost inexorably to ALU pipelining.

Digital Clock Managers (DCMs)
Virtually all FPGAs have some kind of DCM in the form of one or more PLLs (**P**hase **L**ocked **L**oops), and/or DLLs (**D**elay **L**ocked **L**oops) which may be used for a variety of purposes. A DCM can move the clock edge around to change external setup / hold / data out timing, trade internal cycle time margins for tighter external I/O timing, condition the input clock duty cycle, multiply and divide the input clock, generate multiple clocks with phase offsets, etc. The main use for a DCM in a processor core is to manipulate the input clock frequency (multiply / divide) so that the clock feeding the core is at or a bit below the top theoretical speed of the core in order to get the best performance from it.

Note that there is some lower frequency limit below which a DCM will not be able to lock to or otherwise process the input clock, and this figure is given in the AC specifications datasheet for the FPGA. Also note that running the core at high frequencies will increase dynamic power consumption, and may make other logic which isn't in the core but supplied by the core clock more difficult to construct due to the tighter timing constraints. It is entirely possible to have multiple clock domains inside the FPGA, but then one must take special care to condition and properly constrain the timing of data (particularly vectors) that cross clock domain boundaries.

ALU DESIGN

The ALU drives much of the rest of the processor design. Building an ALU for anything but the most trivial of processors is more involved than “compute all results and pick the one you want” though there will probably be a lot of that going on. The wide output multiplexer produced by this naïve approach can be a bottleneck, so intermediate multiplexing and registering must be employed judiciously in order to keep both the required logic to a minimum and the top speed up. Furthermore, some of the logic can be used to calculate more than one type of final result.

Multiplication

Let’s start with the elephant in the room – the multiply unit. If we want to do audio DSP, we need 16 x 16 = 32 bits signed as a rather unsuitable bare minimum. We could probably get by with 16 x 32 = 48 bits signed, with 16-bit samples, 32-bit filter coefficients, and a 48-bit result. For the sake of symmetry and simplicity, let us set the goal as full 32 x 32 = 64 bits signed and unsigned. The use of a signed base entity requires 33 x 33 = 66 to accommodate unsigned, which conveniently is slightly less than twice the width of a single 18 x 18 = 36 FPGA hardware multiplier.

Just as multiplication is performed by hand using pencil and paper, addition and concatenation enable the utilization of several hardware multipliers in parallel, thus increasing the input and output widths. Consider the following base 10 example:

```
       98
 *     67
 --------
       56            56
 +    630                              63_
 +    480   =>              =>     +   48_   =>      111_
 +   5400         + 54__                            + 5456
 --------         ------           --------        ------
     6566           5456               111_          6566
```

Figure 15. Multiplication example.

On the left 98 and 67 are multiplied together in the usual manner, 7x8, 7x90, 60x8, and 60x90. All of the results of multiplication are added together to get the final answer, which requires three additions – or does it? Looking more closely, 5400 and 56 can be simply concatenated, which eliminates one addition. 630 and 480 will always have zero as their least significant digits, so this addition is simplified to adding 63 and 48 giving 111. The result 1110 will also always have a zero as the least significant digit, so adding it to 5456 simplifies to adding 545 and 111 and concatenating the 6 to the least significant digit location. So four half-width multiplications must be performed, but the three additions have been reduced to two, narrowed, simplified, and therefore likely sped up.

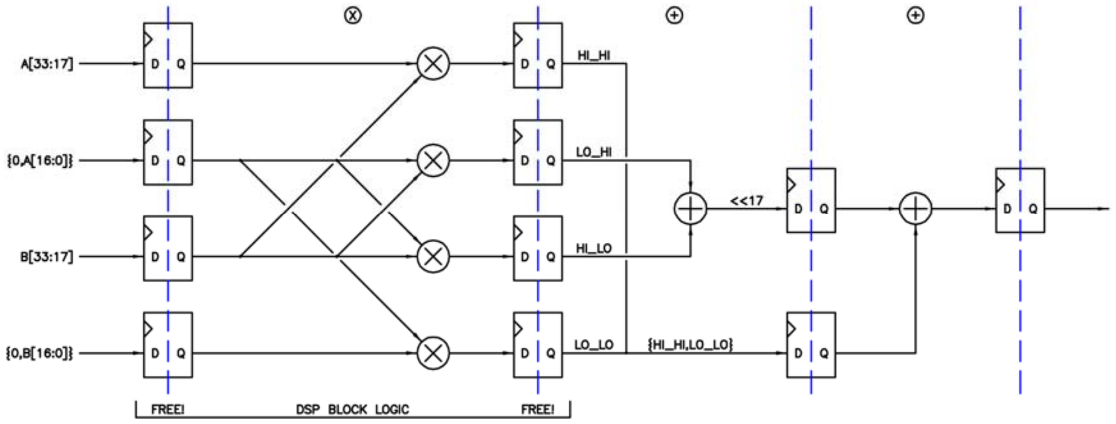


Figure 16. Three stage 33 x 33 = 66 bit signed pipelined multiplication.

The figure above shows these same methods implemented in binary 2s complement logic. The inputs are split in half, with the lower parts zero extended to make them unsigned (interpreting their MSBs as signs would give incorrect results). In the first stage the cross multiplications are performed, in the second stage the outer concatenation and inner add are performed, and in the third stage the final add / concatenation is carried out (the 17 LSBs of the add are automatically implemented by the compiler as a concatenation).

In terms of speed, the 18 bit multiplies in the first stage will likely be the slowest logic in the entire design, though the 47 bit add in the third stage may be close or possibly slightly worse. In the target EP3C5E144C8 device the multiply is restricted to 200 MHz, which means we should endeavor to make all of the other logic somewhat faster in order to have a chance of hitting 200 MIPS aggregate with the final design. The dedicated I/O registering in the multiplier hardware should certainly be used, with interstage registering to isolate the addition hardware, giving three stages and four clocks of latency.

Shifting

One thing that really nagged me about my earlier designs was that their rudimentary ALUs did not exploit the overlapping properties of shift and multiply. It takes a fair amount of FPGA fabric logic to shift a number to the right and left some arbitrary distance and the result isn't super speedy. Having a multiplier just sitting there doing nothing useful during the shift is a missed opportunity.

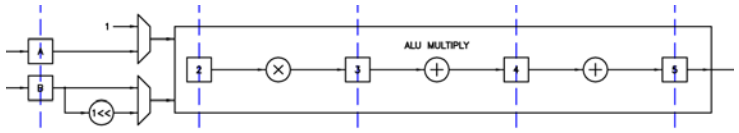


Figure 17. The Multiply and Shift unit.

When a number is multiplied by a power of 2, say 2^5, it is shifted to the left 5 bit positions. So if a full multiplier is already present, the positioning of a simple one-hot shifter at the front (1 << n) can eliminate the left shift hardware. Can a right shift be accomplished with the same hardware? Yes, the trick is to consider the shift distance input as signed, with positive inputs causing shifts to the left and negative inputs shifts to the right. The shift distance MSB (the sign bit) is stripped off and used to select the upper (or extended) multiplication result when set (negative), and the lower result when zero (non-negative). The remaining shift distance LSBs are treated as unsigned and simply routed to the (1 << n) unit at the input as before. Here is an 8-bit example that may help clarify things:

Shift {s,n}	MSB (s)	LSBs (n)	LSBs (n)	B input (1<<n)	A input	X output
+7	0	111	7	10000000	10110111	01011011,**10000000**
+6	0	110	6	01000000	10110111	00101101,**11000000**
+5	0	101	5	00100000	10110111	00010110,**11100000**
+4	0	100	4	00010000	10110111	00001011,**01110000**
+3	0	011	3	00001000	10110111	00000101,**10111000**
+2	0	010	2	00000100	10110111	00000010,**11011100**
+1	0	001	1	00000010	10110111	00000001,**01101110**
0	0	000	0	00000001	10110111	00000000,**10110111**
-1	**1**	111	7	10000000	10110111	**01011011**,10000000
-2	**1**	110	6	01000000	10110111	**00101101**,11000000
-3	**1**	101	5	00100000	10110111	**00010110**,11100000
-4	**1**	100	4	00010000	10110111	**00001011**,01110000
-5	**1**	011	3	00001000	10110111	**00000101**,10111000
-6	**1**	010	2	00000100	10110111	**00000010**,11011100
-7	**1**	001	1	00000010	10110111	**00000001**,01101110
-8	**1**	000	0	00000001	10110111	**00000000**,10110111

Figure 18. 8 bit example of left and unsigned right shifting using a full multiplier.

Though we are thinking of the shift distance input as signed, the shifted one must be presented to the multiplier as unsigned for the 100…000 case to work correctly. Then presenting the input data to be shifted as unsigned or signed will conveniently produce unsigned ("logical" or zero extended) and signed ("arithmetic" or sign extended) right shifts. (Note that independent control over the input signedness is required for this to work, global signedness is not sufficient, which restricts the construction of signed shift from a series of more basic instructions.) With this we have left shift covered, which is sign neutral, as well as unsigned and signed right shift.

Other Uses
Can more be done with this construct? A multiplexer on port A with a fixed input value of one can be used for a couple of things. The first is copying the B input shifted one result to the output of the multiplier, which is useful for generating powers of 2, bit setting & masking, etc. The second is even simpler – multiplication by one replicates the B input to the output of the multiplier, which provides us with a free and convenient "copy B" route through the ALU (though this copy feature is currently unused in Hive).

Note that signed and unsigned left shift are identical (zero padding from the right). With a bit of logic governing the input multiplexers, one of these redundant modes may be replaced with the power of two described above. I chose to replace immediate unsigned shift left operation, non-negative input shift value, with the power of 2 operation, which makes it something of an odd man out in terms of operations, but hopefully isn't too confusing. Immediate signed shift left works as expected. These are summarized below:

Shift Value	Instruction	Operation	Example
-	Shift left, signed	Shift right, signed	`B=-3, A=`**`10110111`**`, Out=111`**`10110`**
+,0	Shift left, signed	Shift left, signed	`B=+3, A=`**`10110111`**`, Out=`**`10111`**`000`
-	Shift left, unsigned	Shift right, unsigned	`B=-3, A=`**`10110111`**`, Out=000`**`10110`**
+,0	Shift left, unsigned	Power of 2	`B=+3, A=`**`xxxxxxxx`**`, Out=0000`**`1`**`000`

Figure 19. Immediate shifting and power of 2 functions as implemented.

Addition and Subtraction
Next we need to consider addition and subtraction. Signed and unsigned can be handled with the same method employed in the multiplier, i.e. by making the inputs one MSB wider and sign or zero extending them depending on whether that input value is to be considered signed or not. As with multiplication, overflow / carry out is extended into the double width data space and selected via instructions. Note that the lower word result is sign neutral, so only the extended result will vary based on input signed / unsigned status. The add / subtract unit is also used to compare (A<B) and (A<0) for conditional branching.

Logical Functions
For logical functions, the usual suspects are implemented:

Operation	Description	Examples
AND	A & B	`A=1100, B=0101 : A=0100`
ORR	A | B	`A=1100, B=0101 : A=1101`
XOR	A ^ B	`A=1100, B=0101 : A=1001`
NOT	~B	`A=xxxx, B=0101 : A=1010`

BRA	&B	`A=xxxx, B=0101 : A=0000`
		`A=xxxx, B=1111 : A=1111`

BRO	|B	`A=xxxx, B=0101 : A=1111`
		`A=xxxx, B=0000 : A=0000`

BRX	^B	`A=xxxx, B=0101 : A=0000`
		`A=xxxx, B=0111 : A=1111`

Figure 20. Logical functions as implemented (examples here limited to 4 bits).

Note that "BR" stands for "bit reduction". The logical unit is also used to compare (A!=B) and (A!=0) for conditional branching.

Miscellaneous Functions
Functions also performed by the logic unit are move / copy, move / copy lower 16 bits both sign and zero extended, 32-bit end-over-end flip, sign bit inversion, and leading zero count.

The Completed ALU

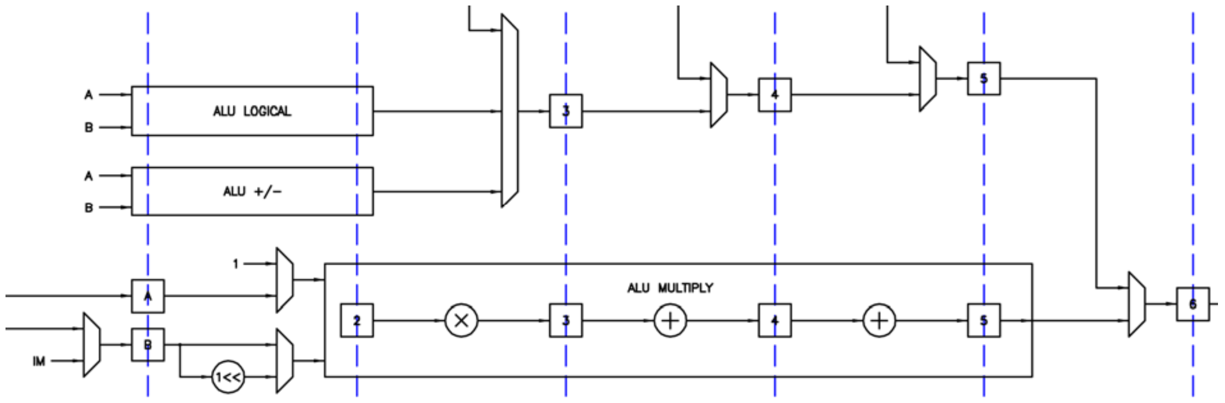


Figure 21. The Arithmetic and Logic Unit (ALU).

The figure above shows the full ALU. The dotted lines and numbered boxes represent interstage registering. Data enters from the left and proceeds through the pipe, with the result emerging on the right six clocks later. Inputs are multiplexed in, and the desired results multiplexed out. The PC (**P**rogram **C**ounter) is multiplexed in between stages 2 and 3 for subroutine / interrupt return address use and for simple reading. Read data from the local register set is multiplexed in between stages 3 and 4. Read and literal data from main memory is multiplexed in between stages 4 and 5. This pipeline structure provides natural intermediate value storage, so the ALU can be presented with new input data on every clock without worry that the new data will be somehow mixed in or confused with previous or later data. Pipeline interstage registering speeds things up and is an otherwise largely stranded FPGA resource, so it might as well be used (my earlier processor designs only employed a few percent of the fabric registers, and not surprisingly were relatively slow).

A somewhat thorny issue with ALU design is working out how the control inputs should be implemented. So as not to slow things down with elaborate encoding and decoding, I decided to encode them one-hot, but with a precedence that is not actually relied upon in practice. The control signals are also pipelined, so the data and the desired operation on it may be conveniently presented together on the left. The multiply and shift unit is complex enough to have its own controls internally pipelined.

MAIN MEMORY

Hive data and program memory space are shared. If we desire full 32-bit single cycle access at the data port then the underlying physical memory must be 32 bits wide. If this is the case then the 16 bit opcode port must strip off the PC LSB, use the truncated result as the address, and use the pipelined LSB as the high or low 16 bit selector - simple enough so far.

However, if we do not want to restrict 32-bit data access alignment to even base addresses, then additional steering logic is necessary at the input and output of the data port. A strong driver here is full 32-bit literal values in-line with the code – it would be nice to be able to place them anywhere in 16-bit width code space with no even/odd address restrictions. For 32-bit values at even addresses the access is straightforward. However, for 32-bit values at odd addresses, we need to swap the 16-bit values both going into and coming out of the data port, and swap the 16-bit write enables as well. Most importantly, we must also increment the lower 16-bit address by one to get the next higher value. FPGA BRAM ports only have a single address, so the need for differing address values necessitates the use of separate ports – the high and low 16 bit memories must be separate physical BRAM entities. This opens a can of worms for defining initial memory contents (boot code) via the SystemVerilog “initial” construct. To skirt this issue I defined a dual memory within a single SystemVerilog file, and then used a temporary continuous / contiguous “ram” array to initialize the high and low real ram arrays via odd / even indexing.

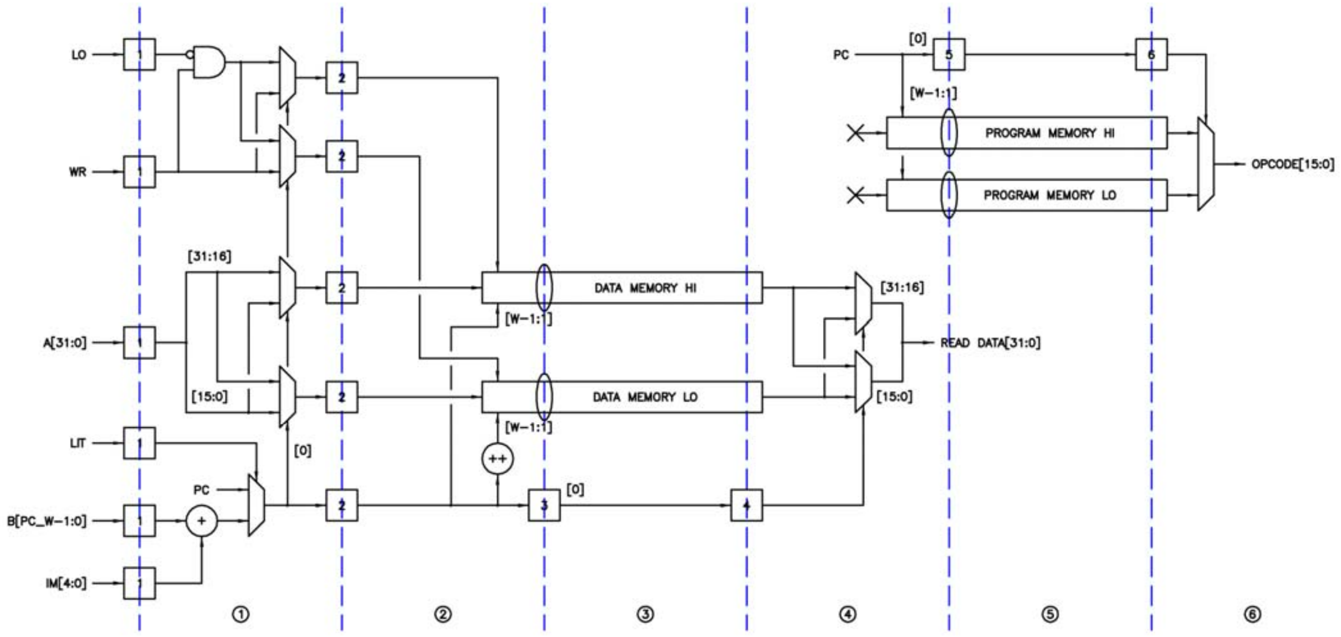


Figure 22. Hive main memory.

The main memory module for Hive is shown in the above figure. The ovals represent the points at which the internal memory arrays are physically read from and written to. Not shown is logic in the op decoder that multiplies the IM value by two when 32-bit read/write access is desired. In the first stage, the PC is selected as the address for literals, otherwise B+IM is selected as the address for data access. The LSB is used to swap 16 bit input data chunks and write enables. In the next stage, the address with its LSB stripped off is used as the upper BRAM address, the incremented address with its LSB stripped off is used as the lower BRAM address and the write data and enables are applied. Note that always incrementing but stripping off the LSB gives an incremented address for odd input addresses but not for even addresses, so multiplexing or switching is unnecessary here. In the final stage the pipelined address LSB is used to swap the read 16-bit data chunks, with the result sent to the ALU multiplexer - where it is used full width, or the lower 16 bits are zero or sign extended.

The program space port is just as describe previously, with the PC LSB stripped off in the first stage and the truncated result applied to both high and low memory address ports. The pipelined LSB is used to select the output in the last stage, the result of which is fed to the opcode decoding unit.

RESET & VECTORING

Resets

There are several ways to reset or clear the Hive core. Let us go through them from most to least significant:

- The first and most comprehensive way to reset the core is via FPGA (re) configuration, which places all of the registers in a known state, and is the only way to unambiguously load or restore the boot code in the main memory block RAMs.
- The second is via asynchronous reset of the fabric registers. This obviously loads the registers with initial values but does not change the contents of block RAM. Asynchronous reset is a top-level pin (rst_i).
- The third is via synchronous core clear, which clears all relevant state such as stack pointers and interrupt history, and vectors all threads back to their reset addresses. Core clear is a top-level pin (cla_i or “clear all”).
- The fourth is by writing ones to all of the vector register thread clear bits. In this manner, threads can clear themselves and all other threads (as well as any combination) as well. The effect is identical to a core clear above (for the threads being cleared).

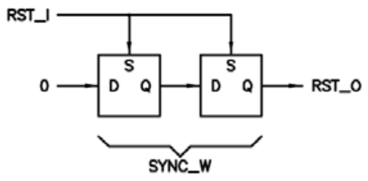


Figure 23. Reset bridge.

The asynchronous reset pin is conditioned and resynchronized to the core clock via a reset bridge, shown above. This well-known construct provides asynchronous set and synchronous release of reset, which is necessary to eliminate race conditions and to prevent block RAM contents from being corrupted. Register depth (to eliminate various metastability issues throughout the design) is set by the global parameter SYNC_W.

Vectoring

Two types of vectoring, or breaking out of current execution, are supported in Hive. The first is via thread clearing, which is described in the reset section above. The second is via an interrupt service routine (ISR), where the PC is pushed to stack zero and the PC is loaded with the ISR address for that thread.

A thread may be interrupted to run a service routine internally via a register-based mechanism similar to that of thread clearing described above (ISR) and via an external interrupt request input (XSR). The thread must be armed to handle I/XSRs before it will respond to them. While servicing an interrupt the thread automatically disables I/XSR response so that subsequent I/XSRs are ignored until the current I/XSR is completed. The operation *op_irt* simultaneously returns the thread to the point of execution before it was interrupted, and re-enables the thread for interrupt operation. This automatic disable/enable action prevents stack overflow / underflow errors in the event of noise on the interrupt input pin or any other series of too closely spaced requests. Any interrupts requested during I/XSR execution are lost, so if your algorithm can’t afford to miss any interrupts you need to either modify this construct or add extra hardware to count / time stamp interrupts. Clearing a thread automatically disarms its I/XSR.

Arming and disarming the I/XSR for a thread is performed by writing a one to the associated arm or disarm bit in the register set. These bits behave like radio buttons, where the last one “pressed” or set is the one that is active, and in the case of contention disarming takes precedence over arming. Writing zeros to these bit fields has no effect, which makes the mechanism safe for multiple access and control by all threads. Reading these bit fields will reveal the current armed/disarmed state for all threads.

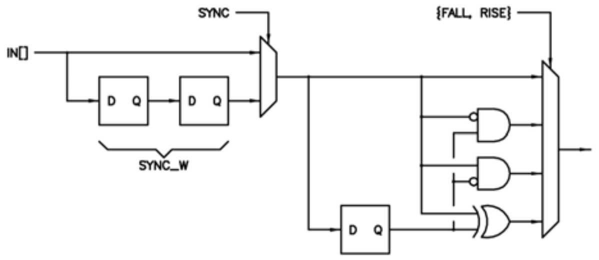


Figure 24. XSR input conditioning logic.

At build time, the user can chose per-thread XSR input conditioning options, the logic for which is shown above. The inputs may be resynchronized or not, after that rising edges and / or falling edges may be detected. This same optional input conditioning logic is used for the register set inputs.

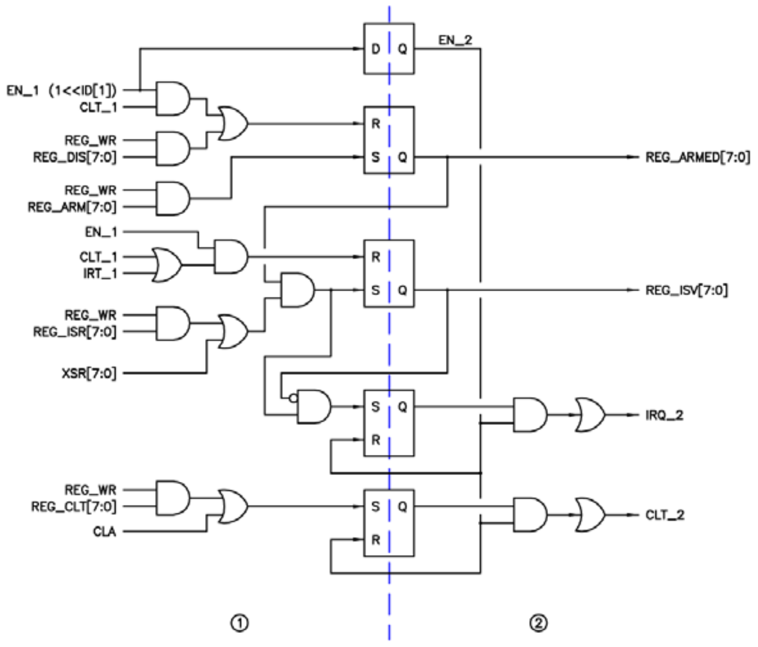


Figure 25. Vector control logic.

Each thread has single layer of vector logic as shown in the above figure. Think of the top most set/reset input of a given flop as having priority over the bottom input in the case of both being asserted. These are clocked flops with the clock inputs omitted for clarity. The top flop delays the enable by one clock. The second flop down is set when a write to the arm register bit is performed, and reset via a write to the disarm register bit or via a thread clear. The flop output is sent to the register set to indicate arm/disarm status for the thread. The third flop down is set when the thread is armed and an internal register-based interrupt or external interrupt is initiated, and reset via an interrupt return (an op_irt instruction decoded by the opcode decoder) or via a thread clear. The flop output is sent to the register set to indicate whether the thread is currently servicing an interrupt. The fourth flop down is set when an interrupt is requested and the thread is not currently servicing an interrupt; the flop output is held (queued up) until the ISR signal is sent to the decoder unit, after which it is retired. The bottom flop employs identical logic to issue thread clears initiated via the register set or via an external synchronous core clear.

Note that there is no distinction made between internally and externally initiated interrupts, which obviates the need for prioritization between them. Because of this, it is recommended that only one or the other be used (per thread).

REGISTER SET

Any processor core will need a local, or internal register set to manage things like the reading and retirement of basic operational errors, enabling and disabling of interrupts, general purpose I/O communications, reading of timers, care and feeding of UARTs and watchdog sanity timers, etc.

RBUS

RBUS is the internal expansion bus that connects all of the registers together and provides communication to / from the core. Multiple registers are assembled into a register set by using a big OR gate to combine their read data vectors. Access to the register set is via two *op_reg* instructions and the internal ALU multiplexer.

The base registers that form the complete register set need not all be instantiated in a single component. They can hang off the RBUS anywhere in the design, thus enhancing modularity and reducing I/O count.

Base Register Component

Register set implementation can be a dull, repetitive, bug prone, and difficult to verify exercise. To automate this to some degree and to reduce the chance of errors creeping in, at the foundation of the Hive register set is a configurable multi-function single base register component with many parameter-based options.

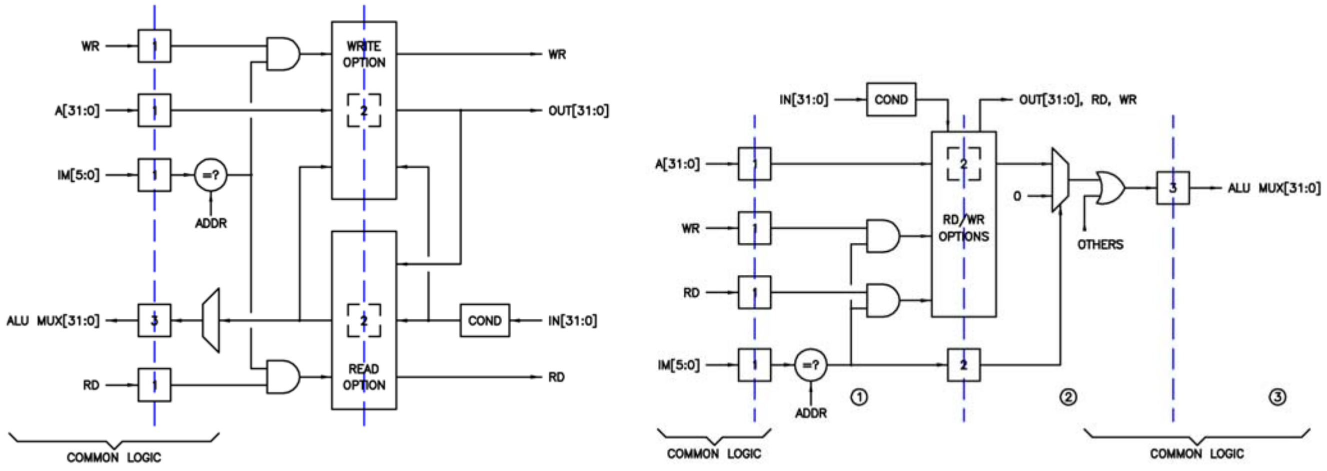


Figure 26. Configurable base register component (two views).

Above are two different schematic views of the interface to the configurable single base register component. The view on the left is interface-centric, with the RBUS interface on left and the per-register logic interface on the right. The view on the right is pipeline-centric; with the RBUS write interface on the left, RBUS read interface on the right, and per-register logic interface on the top. With the interface-centric view the read and write options are more distinct, but the pipeline timing is less obvious. Note that, depending on configured functionality, the pipeline-centric view per-register inputs and outputs are not necessarily shown as residing in the correct pipeline stages. The goal here is to make the selected options plus the external logic “look” or behave like a single layer of stage 2 pipeline registering as this simplifies the consequences of thread interaction via the register set.

Placement of each individual base register within the register set address space is governed by a bus address input parameter to each base register component. Via masks, any number and combination of read / write bits can be “live” (provided with functional logic) with any read / write registers initialized to a known value at reset.

Write Modes
The five possible write side configuration logic modes are shown below and summarized in the following table:

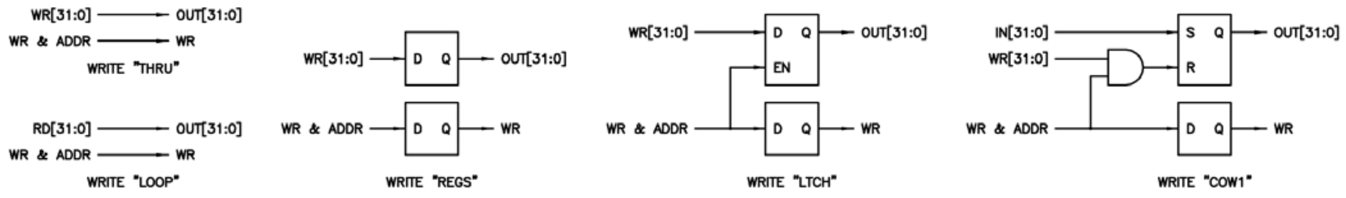


Figure 27. Write modes.

WR Mode	Notes
"THRU"	Write bits directly drive output bits (no latch).
"LOOP"	Read bits looped back to drive output bits (no latch).
"REGS"	Write bits continuously registered (*not* latched), register drives output bits.
"LTCH"	Write bits latched @ write, latch drives output bits.
"COW1"	Output bits set @ input 1, cleared @ write 1 (set has priority).

Figure 28. Write mode descriptions.

Read Modes
The five possible read side configuration logic modes are shown below and summarized in the following table:

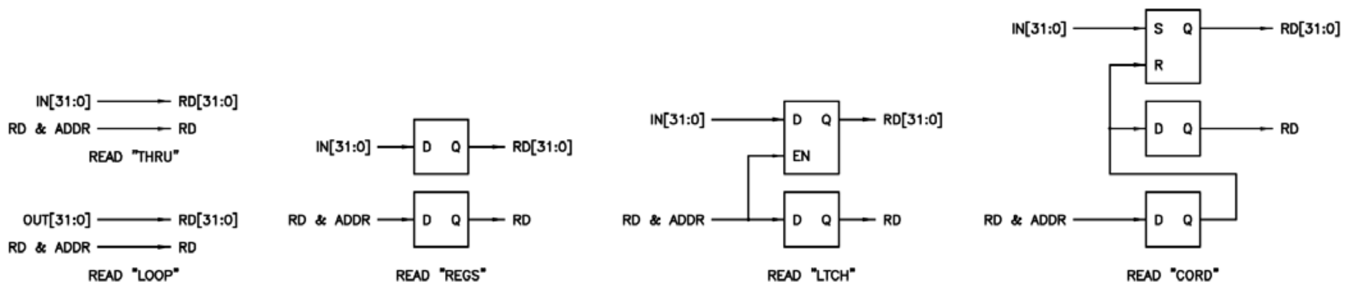


Figure 29. Read modes.

RD Mode	Notes
"THRU"	Input bits directly drive read bits (no latch).
"LOOP"	Output bits looped back to drive read bits (no latch).
"REGS"	Input bits continuously registered (*not* latched), register drives read bits.
"LTCH"	Input bits latched @ read, latch drives read bits (weird!).
"CORD"	Read bits set @ input 1, cleared @ read (set has priority).

Figure 30. Read mode descriptions.

Note that all of the constructs supply read and write event information to the per-register logic interface, and these strobes are timed to indicate the point at which data changes at that interface.

Input Conditioning

Via masks, register input data can be optionally resynchronized and/or made edge sensitive via the circuitry shown below:

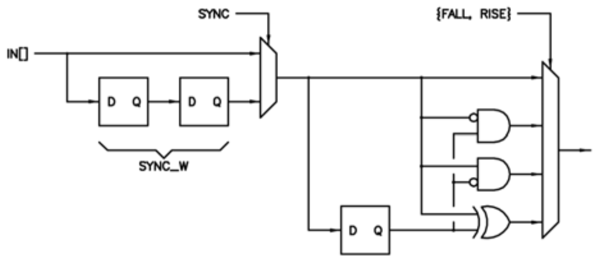


Figure 31. Input conditioning logic.

Also, please note that the "COW1" and "CORD" options were carefully constructed in such a way as to not miss any input events happening at the time of clearing; this was accomplished by giving the set inputs priority over the clear inputs. Missed inputs could lead to the underreporting of events, software lock-up, and other bad things.

Examples

The following table lists the read and write modes for some typical cases:

WR Mode	RD Mode	Notes
"LTCH"	"LOOP"	Your basic read / write register.
"COW1"	"LOOP"	Clear On Write 1 register.
"LOOP"	"CORD"	Clear On ReaD register (and provides feedback to external logic).
"THRU"	"THRU"	Pass-through register to / from external logic.

Figure 32. Some typical examples.

Most common register types can be formed via various combinations of the modes, most others can be implemented by adding a bit of circuitry to this base construct. Mixed mode bits in a single register are not directly supported but two or more base registers could be concatenated if this is desired.

Hive Register Set

The Hive internal register set includes the following basic functionality (note that the first four registers reside in the core and the remaining reside at the top level):

```
Decode:
- 0x00 : Core version register - ver_reg
- 0x01 : Time register - time_reg
- 0x02 : Vector register - vect_reg
- 0x03 : Error register - error_reg
- 0x04 : UART register - uart_reg
- 0x05 : I/O register - io_reg
- 0x06 - 0x3F : UNUSED


================================================================================
- 0x00 : Core version register - ver_reg
--------------------------------------------------------------------------------

  bit  name                 description
-----  ----                 -----------
31-00  ver[31:0]            version info

Notes:
- Read-only.
- Nibbles S/B BCD (0-9; no A-F) to be easily human readable,
  and to eliminate confusion between decimal and hex here.

================================================================================
- 0x01 : Time register - time_reg
--------------------------------------------------------------------------------

  bit  name                 description
-----  ----                 -----------
31-00  time[31:0]           time

Notes:
- Read-only.
- Up-count @ core clock rising edges.
- Threads can read this for relative time.
- Threads can read this & mask off time[2:0] for thread ID.

================================================================================
- 0x02 : Vector register - vect_reg
--------------------------------------------------------------------------------

  bit  name                 description
-----  ----                 -----------
07-00  isr_dis[7:0]         1=thread interrupt disarm
15-08  isr_arm[7:0]         1=thread interrupt arm
23-16  isr_req/isr_act[7:0] write 1 request thread interrupt, read ISR status
31-24  clt_req[7:0]         write 1 request thread clear

Notes:
- Per thread internal interrupt (maskable).
- Per thread clear (non-maskable).
- Per thread arm & disarm masking of interrupts.
- Set on write one radio buttons for ISR arm / disarm.
- Clear takes precedence over ISR:
  - e.g. write 0xFFFFFF00 clears all threads.
- Disarm takes precedence over ISR arm:
  - e.g. write 0x0000FFFF disarms all ISRs for all threads.
- Thread must be armed before ISR can be issued.
- Thread ISRs disarmed @ associated thread clear and async reset.
- ISR ignored during interrupt servicing until op_irt encountered.

================================================================================
================================================================================
- 0x03 : Error register - error_reg
--------------------------------------------------------------------------------

  bit  name                 description
-----  ----                 -----------
07-00  pop_er[7:0]          1=lifo pop when empty; 0=OK
15-08  push_er[7:0]         1=lifo push when full; 0=OK
23-16  op_er[7:0]           1=opcode error; 0=OK
31-24  -                    0

Notes:
- Clear on write one.
- Per thread error reporting.
- All bits cleared @ async reset.

================================================================================
- 0x04 : UART register - uart_reg
--------------------------------------------------------------------------------

  bit  name                 description
-----  ----                 -----------
07-00  uart_data[7:0]       read RX UART data, write TX UART data
   08  rx_rdy               1=RX UART ready (has new data); 0=not ready
   09  tx_rdy               1=TX UART ready (for new data); 0=not ready
31-10  -                    0

Notes:
- Reads from this register pop data from the RX UART.
- To avoid RX data loss, read soon after RX UART might be ready.
- Writes to this register push data to the TX UART.
- To avoid TX data loss, restrict writes to when TX UART is ready.
- UART ready bits will self clear after associated register operation.

================================================================================
- 0x05 : I/O register - io_reg
--------------------------------------------------------------------------------

  bit  name                 description
-----  ----                 -----------
31-00  io[31:0]             I/O data

Notes:
- Separate read / write of I/O data.

================================================================================
- 0x06 - 0x3F : UNUSED
================================================================================
```

UART

For communication with the outside world, Hive has a double buffered UART with DDS (Direct Digital Synthesis) BAUD generator. Parity, flow control, and break detection are not supported. The UART is accessed via the register set.

Conventions

Internally the serial data is non-inverted, and the quiescent level is high. An external inverting and level shifting serial buffer should be fitted if RS232 electrical levels are desired, but no additions or changes are necessary for intercommunication via TTL levels. The serial bits are little endian and in this order: one start bit (low), eight data bits with LSB first, MSB last, one or more stop bits (high). Common BAUD rates are 2400 and 3600, and 2^n multiples of these: 2400, 4800, 9600, 19200, 38400, 76800, 153600; 3600, 7200, 14400, 28800, 57600, 115200.

Transmit Side

The TX UART state machine is shown below:

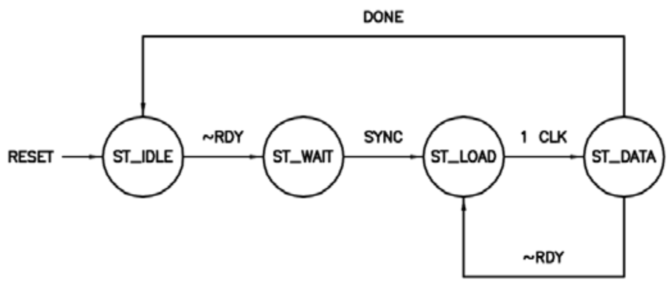


Figure 33. TX UART state machine.

When parallel data is written to the UART register the ready bit goes low. The machine transitions from the idle state to the wait state, and once synchronization to the BAUD generator is achieved it transitions to the load state, where the parallel data is taken, ready is returned high to signal that new parallel data may be written to the register, and the machine then transitions to the data state. Here the data is sent out over the serial line as described above. Once this is done, the machine goes idle if there is no new parallel data, or goes to the load state if there is new parallel data to transmit. In the latter case, since the machine is already synchronized with the BAUD generator, there is no need to resynchronize it. Note that new parallel data can be written as soon as the current parallel data it is taken at the load state, making this a “double buffered” action.

Receive Side

The RX UART state machine is shown below:

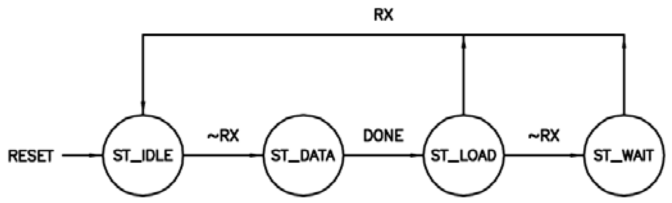


Figure 34. RX UART state machine.

When a low is seen on the serial line (i.e. the start bit) the machine transitions from the idle state to the data state, where the serial data is sampled mid bit and stored in a parallel form. After 10 bits are stored (start, data, stop) the machine transitions to the load state and the parallel data is presented to the register set. At this point, the line is sampled for the current level. If the level is high, the machine transitions to the idle state to wait for new data. If the level is low this is an error, and the machine transitions to the wait state and waits for the error to clear, after which it goes idle. The number of stop bits greater than one is irrelevant to the RX side. This is a “double buffered” action because old parallel data is presented until new data is completely received, after which the old data is overwritten with the new data.

BAUD Generator

The BAUD generator consists of a modulo phase accumulator. This construct is deceptively simple though quite powerful and broadly applicable.

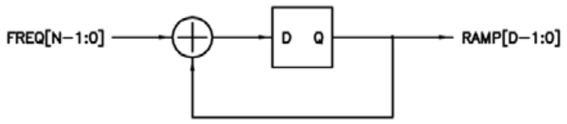


Figure 35. UART BAUD generator.

Successive additions of the unsigned input value cause the accumulated value to steadily increase until rollover, where it naturally restarts from the modulo remainder. The frequency of the rollover rate is therefore directly proportional on the input value, and inversely proportional to the value of 2 raised to the power of the accumulator width D (i.e. the number of unique values a binary number of width D can represent). In mathematical terms:

$$\text{clk\_o} = \text{clk\_i} * \text{FREQ} / 2^D$$

Where clk_i is the system clock frequency and clk_o is the accumulator rollover rate. The input width N governs the output frequency resolution, and I chose 8 bits here to give a maximum possible error of $1/(2^8)$ or 0.39% for a full-scale input. For input values smaller than full-scale, the maximum possible error is governed by the input value itself, which is automatically calculated by the code (at build time) to be in the range [0.5:1.0) of 2^N in order to minimize this error. The qualifier “maximum possible” is used here because, depending on the system clock frequency, the desired output frequency, and D, the specific error may be anywhere between zero and the maximum. Zero error obviously occurs for the cases when clk_o / clk_i = FREQ / 2^D, where all of the numbers are integers.

The MSB of the accumulator is employed as the UART BAUD clock input, with a roughly square wave duty cycle. It isn’t actually used as a clock per se (which would be bad form), but rather it is examined for level changes by the UART logic which runs off of the system clock. When used as a square wave clock source, the phase accumulator construct is known for generating spurious spectral components. There are various techniques that can be applied in order to reduce these, such as setting the FREQ LSB permanently to a '1' and/or dithering the accumulated value with noise, but for UART use I believe spectral purity concerns are moot.

Options

The UART has several build-time parameters, which include BAUD rate, parallel data width, number of stop bits for the TX side, and oversampling rate (generator BAUD rate / line BAUD rate). Errors reported include bad start and stop bits on the RX side, and bad data buffering at the parallel RX interface (data loss due to neglect). There is also a diagnostic serial loopback. Note that the errors and loopback are not currently connected to the UART register.

By default, the UART is configured for 8n1 @ 115.2k. All internal parameters are automatically calculated at build time given the core clock speed, desired baud rate, oversampling ratio, data width, stop bits, etc. BAUD rate is currently fixed, though this could be easily altered in order to accommodate other rates, common (RS232) or uncommon (e.g. MIDI).

If so desired, one could add RX side logic that times the error wait state in order to detect break events. Breaks are traditionally used to interrupt or reset the processor, or to initiate other high level system events.

PIPELINED CORE

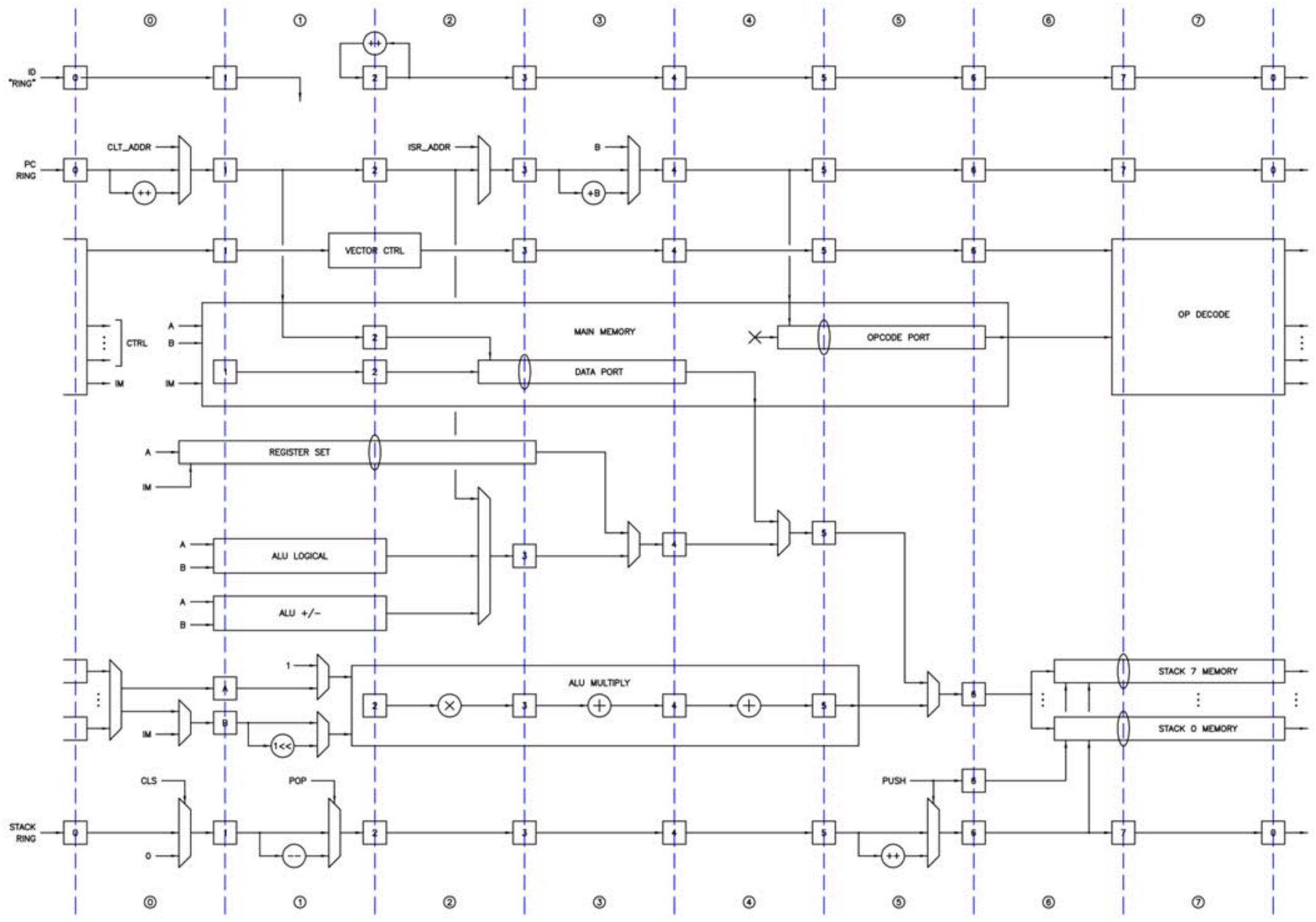


Figure 36. Hive core.

Shown above is the full Hive core. The dotted lines and numbered boxes represent interstage registering. I'll refer to the logic *following* a line of registers with the same numbering as the registers to the left, e.g. stage 3 logic is located between the "3" and "4" register lines. Pipeline stage numbering is relative to the results of opcode decoding being presented to the core logic.

It is vitally important to note that the left and right edges of the figure are connected, which converts the horizontal paths into loops, and so the core may be thought of as one large ring structure. As with the ALU, the pipeline interstage registering provides natural storage for intermediate results. With the pipelines configured as rings, values such as the PC and the LIFO pointers are not only buffered but actually *stored* in the interstage registering (much of which would be required anyway were we to implement multi-threading *sans* pipelining). Clearly, this also forms a natural and simple scheduling mechanism, with packets of data and associated control information spinning around a global ring, passed from stage to stage in a circular bucket brigade fashion, all independent of one another, isolated by and stored within the pipe interstage registering. Let's call these packets "threads" – each stage of the core pipeline can receive and temporarily store, process, and pass on data and control information for a single thread, and there are 8 stages, so we have 8 threads. (Given extra buffering, one could have more threads than pipeline stages with this scheme, but not vice-versa.)

The core may then be thought of as eight processors running at 1/8 the clock speed, sharing a memory (code and data) space which facilitates intercommunication between them as well as code compaction / factoring (the sharing of common constants, subroutines, code, and data). The ring structure of the core forms a "barrel" type scheduler for the threads. Each thread is unique, has as much real time as the next, and gets equal access to the core resources in a strictly offset / overlapped / non-interfering manner. It is up to the programmer to keep the threads busy doing something, though of course unused threads could simply loop, perhaps waiting for an interrupt or a semaphore in memory to change (i.e. “camping on a bit”).

Let us look at the individual rings in a bit more detail.

Time ID Ring

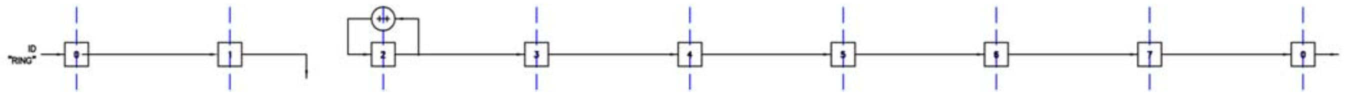


Figure 37. The Time ID “Ring”.

Threads needs an identification number to correctly time the injection of thread clear and interrupt events into the ring, for stack error reporting, and to generate thread clear and interrupt addresses. (All threads *could* vector to the same clear and interrupt address, but that would require overhead for the thread when emerging from start up or when servicing an interrupt: read the thread ID from the local register set, use it to lookup or offset an address, jump there, etc.). A simple up counter at the beginning of the ring generates the thread ID. A true ring structure sans counter could be used here, but that would rely on everything going well from hard reset to infinite time (never do this if you can avoid it) so we break the ring and use a counter and pipe construct instead because it is inherently self-correcting. The interstage registers emerge from reset with the values they would normally have if previously fed by the counter, and thread ID zero is the first to emerge from a global reset, followed by one, two, etc. Note that this isn’t a true scheduler, just a round robin doling out of identifiers, and any scheme that produces a continuously repeating fixed pattern where each ID is generated once every eight clocks would suffice. ID here is actually just the three lowest bits of the 32-bit “Time” counter.

Program Counter Ring

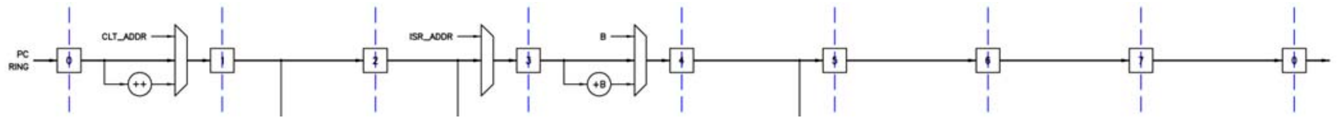


Figure 38. The Program Counter Ring.

Above is the program counter ring. At stage zero the PC is replaced by the thread clear address is if the thread is being cleared, left alone if the thread is taking an interrupt, or incremented to get the next instruction (or in-line literal). In stage one the PC is used as the address for the main memory data port if retrieving in-line literal data. In stage two, the PC is sent to the data path for reading, or as a return address if taking a subroutine or interrupt, and is replaced with the thread interrupt address if taking an interrupt. In stage three, the PC is incremented by B (or an immediate value) if taking a relative branch, or replaced by B if performing an absolute branch or subroutine. In stage four, the PC is used as the address for the main memory instruction port to fetch the next instruction.

Control Ring

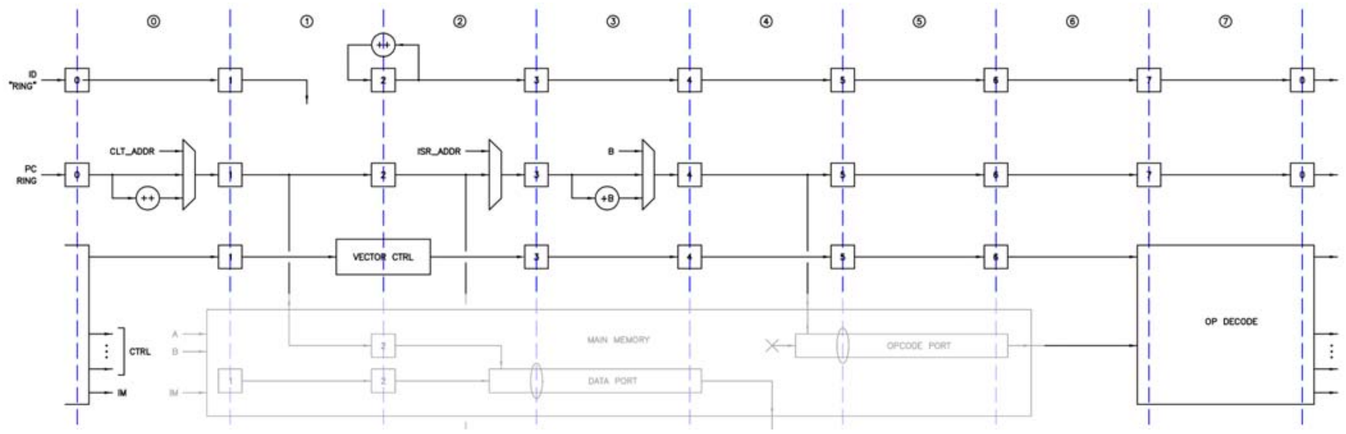


Figure 39. The Control Ring.

The thread ID ring and PC ring, together with the opcode decoding unit and the vector controller, form the control ring. Opcode decoding takes place in several stages in order to speed it up, and consequently the instruction fetch must happen early in the pipeline, which means conditional testing has to take place even earlier. The vector controller uses the thread ID to inject thread clear and interrupt events into the control ring structure (and to retire these events once injected). These events are handed off to the opcode decoder where they are prioritized and decoded. Note that each thread has its own separate clear and interrupt. The clearing or interruption of one or more threads will not disturb the other normally functioning threads. The abundance of independent interrupts means that hierarchical interrupt logic / code will not be necessary for most applications.

Stacks Ring

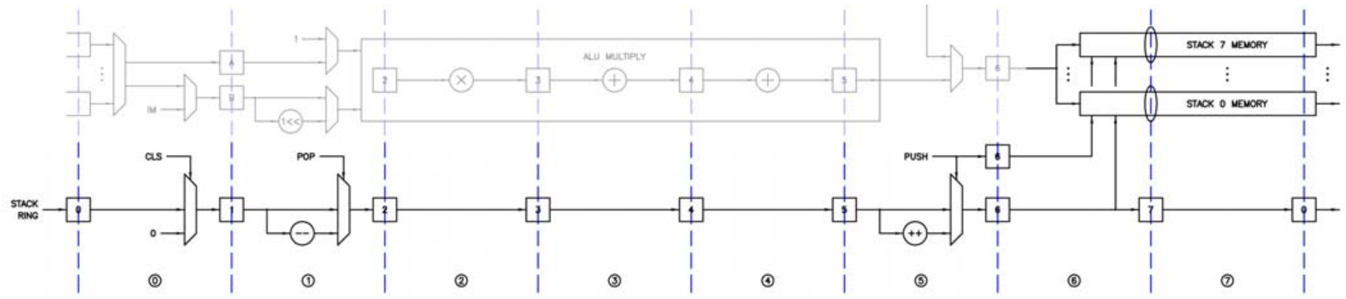


Figure 40. The Stacks Ring.

Shown above is the stacks ring. In stage zero the stack levels are cleared if the thread is being cleared. In stage 1 valid pop events decrement the relevant stack level(s). In stage five valid push events increment the relevant stack level. Not shown in stages one and five is logic that measures fullness and prevents push when full / pop when empty from corrupting the stack levels (if so configured at build time). These error events are reported to the local register set for debugging purposes. Separating the clear, pop, and push logic in this manner actually simplifies combined pop & push actions, as well as error tracking and reporting. Valid pushes also generate write enables for the LIFO memories, which are pipelined and applied in stage six. Also in stage six the stack levels are stripped of their MSB to form pointers, and are concatenated with the thread ID to form the LIFO memory write / read addresses, and the ALU result is written to one of the stack memories. This pointer / thread ID concatenation scheme gives each thread its own private set of stacks in shared block RAM, and renders stack corruption from one thread to another impossible. Stack to stack corruption within a thread is also impossible due to the physically separate block RAMs employed for each stack.

Data Ring

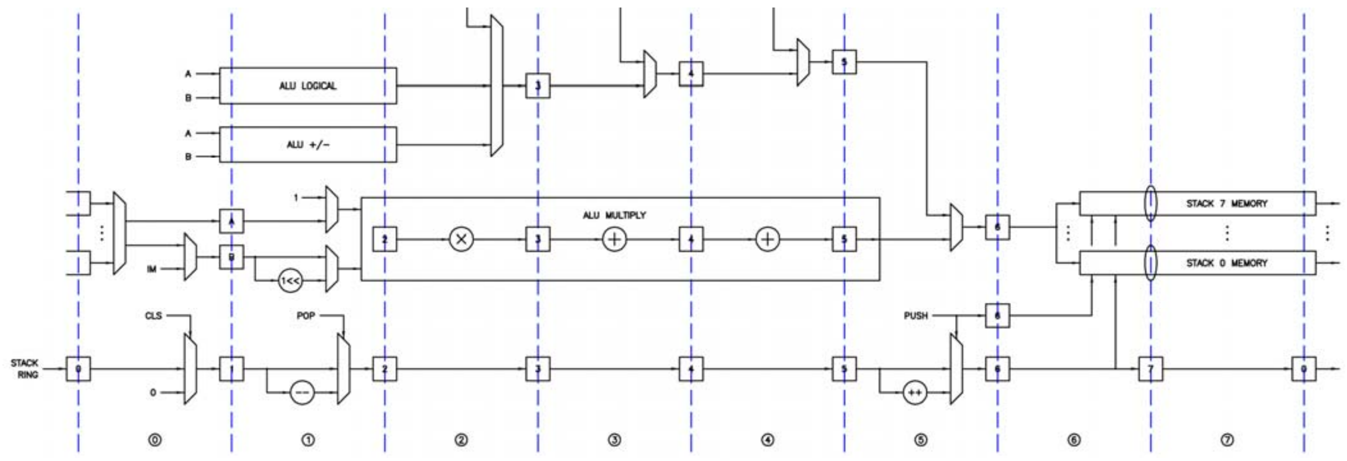


Figure 41. The Data Ring.

As shown above, the stack output multiplexer, ALU, and stacks ring constitute the data ring.

Finally, the control ring, data ring, main memory, and local register set make up the Hive core. Note that the time counter in the ID ring and the vector unit in the controller ring both line up with the register set component read/write oval. This is not a coincidence, this placement facilitates the pipelining of the register set logic and so keeps thread interaction via the register set from “time traveling” in the overall pipeline.

INSTRUCTIONS / OPCODES

With the hardware structure in place, we can now decide on the operations and their encoding. In actuality, the design process isn't this cut and dried, and the inclusion and format of certain instructions will obviously ripple back into the hardware structure.

Beyond Turing completeness, selecting a sufficiently self-contained and balanced set of "basic building block" instructions for general purpose use is something of a conundrum – you want to accommodate your own early coding examples, but how do you guarantee efficient coverage for future code you and others may write without resorting to a "kitchen sink" design? Studying the instruction sets of similar processors is a useful activity here, as is coding up often-used simple functions like division and others, which are not supported directly by the ALU. If the coding process feels particularly laborious, or the resulting code strikes you as unusually awkward or cryptic, then you likely have more work to do.

Determining how to best fit the instructions into the opcode space can be a challenge and you will likely experience design pushback due to opcode space limitations – if you don't you probably left out something important or aren't otherwise utilizing the space efficiently. From the previous discussion, we know there are at most two stack indexes of three bits each, with one pop bit for each index. This consumes eight bits of opcode space, leaving eight bits remaining. Most processors utilize the operand select field room freed up when fewer operands are required for a particular operation, and Hive does this as well.

In-Line Data

The bandwidth consumed by immediate / literal data is quite important; some processor designs devote (literally!) half of the opcode space to a single immediate data operation. With Hive, the way to insert larger literal data values from the instruction stream is via an in-line mechanism, where the data value immediately follows the *op_lit* instruction in program space. The in-line literal instructions use 32 or 48 bits: the 16-bit literal instruction followed by 16 or 32 bits of data (the data is used "literally" rather than decoded) but just one cycle to push 16 or 32 bits of data. So there is a full width literal instruction, as well as signed and unsigned literal low instructions. One bad thing about the in-line literal mechanism is that you can't be 100% sure what is code and what is data without starting at the very beginning of execution and decoding everything up to the point you are interested in. Luckily this is exactly what the processor does anyway, but it can create issues when displaying disassembled memory dumps and the like.

In-Line Addresses?

At the excellent suggestion of one Hive reviewer, I experimented with this in-line mechanism as a source of absolute addresses and address offsets. Doing so conveniently obviates the need for an immediate data field in the branch instructions, which frees up the second stack index and gives generic (A?0) and (A?B) conditional and unconditional relative and absolute jumps (and subroutines if desired) of 32 bits. I very reluctantly abandoned this tantalizing avenue because of the timing pinch point it created between conditional evaluation, address / offset selection, next address calculation, fetch, and decoding, which unacceptably slowed down the core logic. It also introduced a bit of confusion as to what exactly the address offset was relative to - the jump instruction or the following in-line value?

Immediates

Instructions that contain an immediate data or address offset field can be quite effective, though they quickly gobble up opcode space so they need to be firmly in the frequent use category to earn their keep. The immediate field width and position within the opcode need not be fixed. I decided to implement immediate 8-bit and 6-bit signed instructions, as well as immediate 8-bit, 6-bit, and 4-bit unsigned instructions. The 8-bit unsigned immediate is used for operations that require the selection of one or more of the stacks, the 6-bit unsigned immediate as the register space address, and the 4-bit unsigned immediate as a memory access address offset.

Immediate instruction types and opcode space consumption:

- Two 8-bit unsigned immediate instructions: stack operations – 512 codes.
- Two 6-bit signed immediate data instructions: signed shift left, power of 2 / unsigned right shift – 2048 codes.
- Two 6-bit unsigned immediate address instructions: register access – 2048 codes.
- Seven 8-bit signed immediate address instructions: unconditional jump, data, add, conditional (A?0) jump – 24832 codes.
- Four 4-bit unsigned immediate address instructions: memory access – 16384 codes.

These are shown below in tabular form:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OPCODE								**B**				**A**			
OPCODE								**IM[8]**							
OPCODE						**IM[6]**						**A**			
OPCODE				**IM[8]**								**A**			
OPCODE				**IM[4]**				**B**				**A**			

Figure 42. Instruction formats (top to bottom): Non-immediate instruction, 8-bit signed / unsigned; 6-bit signed / unsigned; 8-bit signed; 4-bit unsigned.

Immediate Jumps
The longer a loop is, the less we tend to be concerned with loop overhead. However, immediate branching is vital to the production of fast, compact, iterated code. Even if we constrain the maximum immediate jump distance to be quite small, the plethora of fundamental and therefore essential conditional tests means the immediate branch instructions can easily consume a huge portion of the total opcode space.

In the end I decided to implement eight skip [PC+1] conditional (A?B) jump instructions, eight skip 2 [PC+2] conditional (A?B) jump instructions, four 8 bit immediate signed distance [PC+127/PC-128] conditional (A?0) jump instructions, and one 8 bit immediate signed distance [PC+127/PC-128] unconditional jump instruction. Jumps are relative to the PC and jump the signed immediate distance if the test is true.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x0				**0x4**				**IM[8]**							

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	**1**	**L**	**N**	**IM[8]**								**PA**	**A**		

Figure 43. Immediate address instruction formats (top to bottom): unconditional jump; (A?0) conditional jump.

All relative branching is relative to the *next* instruction address, and not *this* instruction address, which is the most natural convention: a relative jump of 0 does nothing, a relative jump of +1 skips over the next instruction, and a relative jump of -1 is an infinite loop.

The only thing conditional about a conditional instruction is whether the branch is taken. Pops are *always* performed if pop bits are set in the conditional instruction.

Other than equality, the most useful conditional tests tend to split the numerical space under test in half via *sign* (less than zero, not less than zero), and *subtraction sign* (less than, not less than). I have found other combinations of less than, equal to, and greater than testing to be less useful, and so are not directly supported in Hive. (A?B) testing seems to happen more rarely than (A?0) testing, so these comparisons are relegated to the skip groups; swapping the positions of A and B in the tests provides more in the way of useful combinations; the comparisons less than, and not less than, have both signed and unsigned variants. Odd / even testing is included but the need for it seems less pressing so it is placed in the skip groups.

Some conditional sign conventions / observations:

- The comparisons L (A<0), and NL (A>=0) of A to zero necessarily treat A as signed.
- All comparisons of A and B that are signed | unsigned treat *both* A and B as signed | unsigned.
- The equality comparisons Z (A=0), NZ (A!=0), E (A=B), and NE (A!=B) are obviously sign neutral.

Immediate Memory Access
There are 4 instructions for memory access: two for read and two for write. There is a full 32-bit read, and a low 16-bit sign extended read. Similarly, there is a full 32-bit write and low 16-bit write. An unsigned 4-bit immediate field provides a group of 16 convenient memory slots from a base address. This offset is automatically resized

according to access width (*2 for full access), and full access can be across non-even address boundaries (where the base address pointed to by B is odd).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	**1**	**L**	**W**	**IM[4]**				**PB**	**B**			**PA**	**A**		

Figure 44. Immediate address instruction format: memory access.

Data and code space are shared, which enables the programmer to freely allocate and partition it, and enables the copying in of new code via this data read / write mechanism. These instructions use up a lot of the opcode space, but memory operations consume many cycles on average, so they should be made as efficient as possible.

Immediate Register Access
There are two instructions for register set access: full 32-bit read and write. An unsigned 6-bit immediate field provides access to as many as 64 registers.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x9				**1**	**W**	**IM[6]**						**PA**	**A**		

Figure 45. Immediate address instruction format: register access.

Immediate Shifts / Power of two
Shifts distances / directions known at "compile time" are most efficiently coded as 6-bit wide signed immediates. Immediate shift instructions are highly useful because they allow for full 32-bit left or right shifting in a single cycle, and one or two shifts can perform many chores that would otherwise require dedicated instructions and hardware (full width MSB / sign flag, arbitrary width sign / zero extension, isolation of contiguous bit fields, 2^n integer modulo, etc.).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x9				**0**	**U**	**IM[6]**						**PA**	**A**		

Figure 46. Immediate shift instruction format.

The immediate unsigned shift performs a power of two (one hot bit) function when the shift distance is non negative, and unsigned right shift when the shift distance is negative. This dual functionality replaces the redundant left shift (signed and unsigned left shift otherwise produce identical results) with a useful secondary operation.

Immediate Add
An immediate signed add is provided for small quick increments / decrements [+127/-128]. Conveniently for memorization purposes, the most significant nibble is 0xA.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0xA				**IM[8]**								**PA**	**A**		

Figure 47. Immediate add instruction format.

Immediate Byte
With Hive, the immediate signed data instruction is the way to insert small data values from the instruction stream. The immediate data instruction uses 16 bits and one cycle to push eight signed bits of data to the stack selected by A. Conveniently for memorization purposes, the most significant nibble is 0xB.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0xB				**IM[8]**								**PA**	**A**		

Figure 48. Immediate data instruction format: byte.

Branching
There are four types of non-immediate branches – jump, go to, I/XSR return, and subroutine:

- **JMP** (jump) is relative to the current PC and is either conditional or unconditional. It jumps a signed distance given by B either unconditionally or if the test (A?0) is true.
- **GTO** (go to) is absolute and is either conditional or unconditional. It loads the PC with the value given by B either unconditionally or if the test (A?0) is true.
- **IRT** (interrupt return) is an unconditional GTO that re-enables the I/XSR state logic.
- **GSB** (go to subroutine) is absolute and unconditional. It loads the PC with the value given by B and stores the return address (the current PC) to A.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x3				**0**	**0**	**C**	**N**	**PB**	**B**			**PA**	**A**		

Figure 49. (A?0) conditional jump instruction format.

Note that there is no explicit *return from subroutine* operation – a GTO is used here. The return address can be simultaneously popped at this point as well for cleanup.

After some deliberation, I deemed conditional IRT and GSB instructions as too confusing and not possessing sufficient need / utility, so they are not implemented (though they easily could be).

Shifts / Powers of 2
Variable shifts in both signed and unsigned variants are provided. I felt it was important to keep the variable shifts unmixed in functionality (i.e. no power of two here as exists in the immediate form) so that there could not be unexpected behavior, and so sign testing of the shift variable might not be necessary. A separate variable power function generates powers of two and is sign agnostic regarding the shifted one distance input value.

Arithmetic & Logical
Add, subtract, multiply, shift, and all of the logical operations have been described previously. Functions also performed by the logic unit are move / copy, 32-bit end-over-end flip, sign bit inversion, and leading zero count.

Stack Pop
By repurposing the entire operands area of the opcode, the stack pop instruction is able to pop all, none, or any combination of stacks at once.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x0				**0x1**				**IM[8]**							

Figure 50. Stack pop instruction format.

Stack Clear
By repurposing the entire operands area of the opcode, the stack clear instruction is able to clear all, none, or any combination of stacks at once.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x0				**0x2**				**IM[8]**							

Figure 51. Stack clear instruction format.

Other Instructions
There are several stack and miscellaneous instructions:

- **PGC** pushes the current program counter (pointing to the next instruction) to A.
- **NOP** is a do nothing instruction; all functionality including pops is disabled.

Naming Conventions
Consistency is important with instruction naming conventions so that one can easily remember them or, failing that, quickly construct them knowing some basic rules. The letters "op_" precede all Hive instructions, and this is mainly to avoid conflict with SystemVerilog reserved words. After this is the three-letter operation, usually followed (but not necessarily) by a second underscore and one or more option letters. Obviously not all operations support all options.

op_*	Function
nop	**N**o **OP**eration // no pops either
pop	**POP** // one-hot bit per stack
cls	**CL**ear **S**tacks // one-hot bit per stack
pgc	**P**ro**G**ram **C**ounter // A := PC
lit	In-line **LIT**eral data // A := MEM[PC]
reg	**REG**ister access // A := REG[I]; REG[I] := A
cpy	**C**o**PY** // A := B
nsb	**N**ot **S**ign **B**it // A := {~B[31], B[30:0]}
not	Bitwise logical **NOT** // A := ~B
and	Bitwise logical **AND** // A &= B
orr	Bitwise logical **OR(R)** // A |= B
xor	Bitwise logical **XOR** // A ^= B
bra	**B**it **R**eduction **A**nd // A := &B
bro	**B**it **R**eduction **O**r // A := |B
brx	**B**it **R**eduction **X**or // A := ^B
flp	**FL**i**P** // A := B[0:31]
lzc	**L**eading **Z**ero **C**ount // A := LZC(B)
add	Arithmetic **ADD**ition // A += B
sub	Arithmetic **SUB**traction // A -= B
mul	Arithmetic **MUL**tiplication // A *= B
shl	**Sh**ift **L**eft // A <<= (B or I)
pow	**POW**er // A := 1<<B
jmp	**J**u**MP** // PC := PC+(B or I)
gto	**G**o **TO** // PC := B
irt	**I**nterrupt **R**e**T**urn // PC := B (re-enable ISR)
gsb	**G**o **S**u**B**routine // PC := B, A := PC
mem	**MEM**ory access // A := MEM[B+I]; MEM[B+I] := A
byt	**BYT**e data // A := I
shp	**SH**ift (unsigned) **|** **P**ower of 2

Figure 52. Instruction operations.

Op_*_?	Function
#	Immediate field bit width
r	**R**ead
w	**W**rite
n	**N**ot
e	**E**qual
l	**L**ess / **L**ow
z	**Z**ero
s	**S**igned
u	**U**nsigned

Figure 53. Operation options.

Rules for these options are:

- Single underscore between instruction and options.

- The option **#** if present comes first. Some operations exist only in an immediate form (dat, reg, mem) and the rule here is to only include the immediate **#** option for disambiguation (with shp_6u the only exception).
- The options **r**, or **w** if present come next.
- The conditional options **n**, **e**, **l**, and **z** if present come next, and in that order.
- The options **s** or **u** if present go last.
- The **s** and **u** options for arithmetic operations imply extended results.
- The **s** and **u** options for copy, literal, and memory access operations imply lower results.

Encoding

When assigning the actual numerical values to the instructions – the operational encoding or opcodes – it is important to make the decoding as straightforward and orthogonal as possible. I initially used a spreadsheet to keep track of them, with a column for each output control signal. This helps to reveal similar decoding patterns, which allows the opcodes to be grouped together / advantageously arranged for ease of interpretation by the decoding logic.

Codes	Instruction	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
256	**nop**	0x0				0x0				-				-			
256	**pop**	0x0				0x1				IM[8]							
256	**cls**	0x0				0x2				IM[8]							
256	**jmp_8**	0x0				0x4				IM[8]							
256	**pgc**	0x0				0x8				PB	B			PA	A		
768	**lit**	0x0				1	1	L	U	PB	B			PA	A		
2048	**skp (A?B)**	0x1				0	U	L	N	PB	B			PA	A		
2048	**sk2 (A?B)**	0x1				1	U	L	N	PB	B			PA	A		
1024	**jmp (A?0)**	0x2				0	0	L	N	PB	B			PA	A		
1024	**gto (A?0)**	0x2				0	1	L	N	PB	B			PA	A		
256	**jmp**	0x2				0xC				PB	B			PA	A		
256	**gto**	0x2				0xD				PB	B			PA	A		
256	**irt**	0x2				0xE				PB	B			PA	A		
256	**gsb**	0x2				0xF				PB	B			PA	A		
768	**cpy**	0x3				0	0	L	U	PB	B			PA	A		
256	**bnh**	0x3				0x1				PB	B			PA	A		
256	**not**	0x3				0x4				PB	B			PA	A		
256	**and**	0x3				0x5				PB	B			PA	A		
256	**orr**	0x3				0x6				PB	B			PA	A		
256	**xor**	0x3				0x7				PB	B			PA	A		
256	**bra**	0x3				0x8				PB	B			PA	A		
256	**bro**	0x3				0x9				PB	B			PA	A		
256	**brx**	0x3				0xA				PB	B			PA	A		
256	**flp**	0x3				0xC				PB	B			PA	A		
256	**lzc**	0x3				0xD				PB	B			PA	A		
16384	**mem**	0	1	L	W	IM[4]				PB	B			PA	A		
768	**add**	0x8				0	0	X	U	PB	B			PA	A		
768	**sub**	0x8				0	1	X	U	PB	B			PA	A		
768	**mul**	0x8				1	0	X	U	PB	B			PA	A		
512	**shl**	0x8				1	1	0	U	PB	B			PA	A		
256	**pow**	0x8				0xE				PB	B			PA	A		
1024	**shl_6s**	0x9				0	0	IM[6]						PA	A		
1024	**shp_6u**	0x9				0	1	IM[6]						PA	A		
2048	**reg**	0x9				1	W	IM[6]						PA	A		
4096	**add_8**	0xA				IM[8]								PA	A		
4096	**byt**	0xB				IM[8]								PA	A		
16384	**jmp_8 (A?0)**	1	1	L	N	IM[8]								PA	A		

Figure 54. Opcode encoding.

As seen in the table above, the single instructions are arranged by functionality in four groups of 16, with the first group something of a catchall, branching second, logical third, memory access fourth, and arithmetic fifth. The immediates follow, with immediate shifts, add, and data first, and (A?0) jumps bringing up the rear. Most instructions are conveniently segmented into hex fields, which facilitates human reading / interpretation.

There are some opcode slots open for future expansion, but the opcode space is otherwise largely consumed by instructions with immediate fields.

IMPLEMENTATION

Coding

Designing for ease of comprehension starts at the naming level, and unless one has clinical levels of OCD (likely an asset in this business) I honestly don't believe there is any such thing as spending too much time coming up with terse, yet sufficiently descriptive, names for the various signals, parameters, and modules. Likewise, often too little attention is paid to partitions, modularity, and local / global hierarchy. Design for ease of verification starts at the module / component level, and one should always be looking for ways to make the partitions as useful and as meaningful as possible, rearranging things when it serves these purposes. Base module composition should be a happy medium of non-trivial yet non-overwhelming amounts of common logic that verifies easily and that tends to minimize the interface (a good indication of commonality). As one moves up the hierarchy, the aggregating modules should consist increasingly of instantiated sub modules, and decreasingly of additional miscellaneous logic (ideally none at the very top).

With the exception of the UARTs, there is a general dearth of state machines in the Hive code, which is often an indication of poor coding style. Earlier versions of the vector controller employed a state machine for each thread, but I removed these in order to have more direct control over the simple binary states that are largely orthogonal, and to make the logic a single register layer deep to better fit the register set pipelining. It's been my experience that state machines are not always the best choice when it comes to processing pipelined data streams – and I suppose processors themselves can be seen as vastly overgrown state machines.

In terms of language logical constructs, I find that looping through vector bits usually translates surprisingly efficiently to hardware, so I don't go out of my way to avoid this foreign seeming serial coding style. Synthesis endeavors to eliminate duplicates to save logic and is quite good at this, so I feel free to replicate registers wherever it makes the modules easier to partition (e.g. the post stack selection multiplexer registers are replicated at the input of each ALU sub module). On the flip side, the fitter inserts duplicate registering as necessary to meet timing (if instructed to do so via the fitter settings – usually this is the default) so there is no need to do this manually. I attempted to generate the pipelined multiplier via the Altera HDL wizard instead of the literal translation of pen and paper multiplication, but only powers of two widths were supported (this design needs an extra MSB to handle both signed and unsigned inputs). Even if the desired width construct were supported by the wizard, the resulting code would be less portable, so I decided not to go the wizard route. More general tool-based auto-pipeline inferencing does not strike me as ready for prime time with this kind of project.

Any largish project will make one more familiar with the abilities and limitations of the languages and tools employed. Hive started out in Verilog 2001, and while working on version 5 of the core I discovered that System Verilog was actually a straightforward update to Verilog, and not some radically different verification-centric language as the confusing name change (among other things) led me to believe. (In particular, I encourage you to read the papers by Stuart Sutherland on both Verilog and SV synthesis). The most welcome change in SV is the replacement of the cumbersome and confusing *wire* and *reg* basic types with the generic *logic* type, which establishes some sanity and makes optional registering of signals and ports simpler. I found the added SV package support quite useful for holding magic numbers such as global parameters and derivations of them, as well as custom enumerated logic types, which helps greatly with things like opcode encoding and decoding. (The use of don't cares in enumerated types is particularly powerful when coupled with casex statement decode.) SV multi-dimensional I/O comes in quite handy with the PC and ID ring ports, and SV default (*) port connect cuts down hugely on bug-prone inter-module wiring, while providing useful port linting checks. Altera's Quartus tool supports most of the SV enhancements to Verilog; Xilinx ISE surprisingly has no SV support, though Xilinx claims Vivado does.

Verification

Job #1 when building a processor is obviously wringing out all the bugs. Processors that have caches, pipeline hazards and stalls, and lots of internal state are notoriously difficult to verify (and therefore fundamentally trust – Pentium division bug anyone?). Much of engineering is the exercise of complexity management, and processor architecture should be guided by this principle as well. Simplicity allows one to juggle the processor model in one's head, but it can also greatly ease the verification problem. Hive has relatively simple control structures, minimal internal state, and the entire design is partitioned into hierarchical right-sized modules that are as self-contained as possible, making verification a straightforward and relatively painless task.

All basic blocks should be fully tested before being assembled together. With Hive, most module port widths and associated internal logic are parameterized so, for example, full verification of the ALU may be accomplished by shrinking the data port widths to a trivial size and manually examining the results of all possible inputs. The multiplier may be verified separately at full width by comparing its results to a second naively instantiated multiplier, both supplied identically with corner cases and random input (the inclusion of this test hardware is a parameterized option for the multiplier base module). The intermediate control and data ring constructs allow for the testing of lower level aggregate functionality.

Once basic functionality is up (thread clearing, immediate data, jumps) specially tailored boot code enables the processor to verify itself. Stack functioning and error reporting should be fully tested for all threads. Jump distances and all associated conditionals should be confirmed. Each opcode should be tested to make sure it is being decoded and functioning correctly – distinctive signatures may be used here rather than exhaustive testing. This is also a good way to get early experience hand coding the processor (the point at which I have became largely disillusioned with my past designs) which may lead to changes in the op codes and other parts of the fundamental design.

Finally, several simple algorithms should be coded up, first in a spreadsheet and then in the simulated core boot code, with the results compared. When working on this phase of the design I find that I had to fight a strong inclination to tailor the instruction set to the algorithm *du jour* and keep my eye on the big picture. For instance, after developing the log2 algorithm, a leading zero count instruction (lzc) seemed like it would be a valuable addition. I coded up a fully parameterized SystemVerilog module and speed / functionally tested it, but only after I recognized the general value of this function (it has many normalization uses) did I include it in the logical unit of the Hive ALU.

Speed
Another major goal when building a processor is getting the top speed as high as possible. To this end, many Hive modules have configurable registering on their inputs and outputs, which can effectively isolate timing to the fabric rather than the FPGA I/O pins when doing individual module speed trial builds. The component *pipe.v* can go from a single wire to any desired width and registering depth, and is used throughout the design for general registering and pipelining. (A downside to this approach is that useful internal signal names get reduced to vector indexes, which can make them difficult to differentiate in simulation.)

It is important during early testing to identify the slowest low-level hardware path. This is the lower speed target for the remaining circuitry, which should be written / implemented at least 10% or so faster so as to have a bit of a cushion when it all comes together. The more margin the better because modules have a tendency to slow down considerably when spattered willy-nilly onto the fabric with all of the other logic. There are various FPGA synthesis options that will likely produce a faster top speed (e.g. higher fitter, placement, and router effort levels) and an automated seed hunt with multiple options (e.g. Altera's "Design Space Explorer") will usually produce a faster point in the design space if you've got the time to spare. This is worth doing if only to know the top speed easily attainable.

Watch the fitter resource allocation like a hawk, particularly for any extra block RAM creeping into your design. It seems the synthesis / fitter likes to replace pipe stages with block RAM, which can really slow things down.

Use a DCM to get your board clock up to the maximum speed of the core if you want performance, or keep the clock speed lower to conserve power.

APPENDIX A : HIVE SIMULATION

Motivation

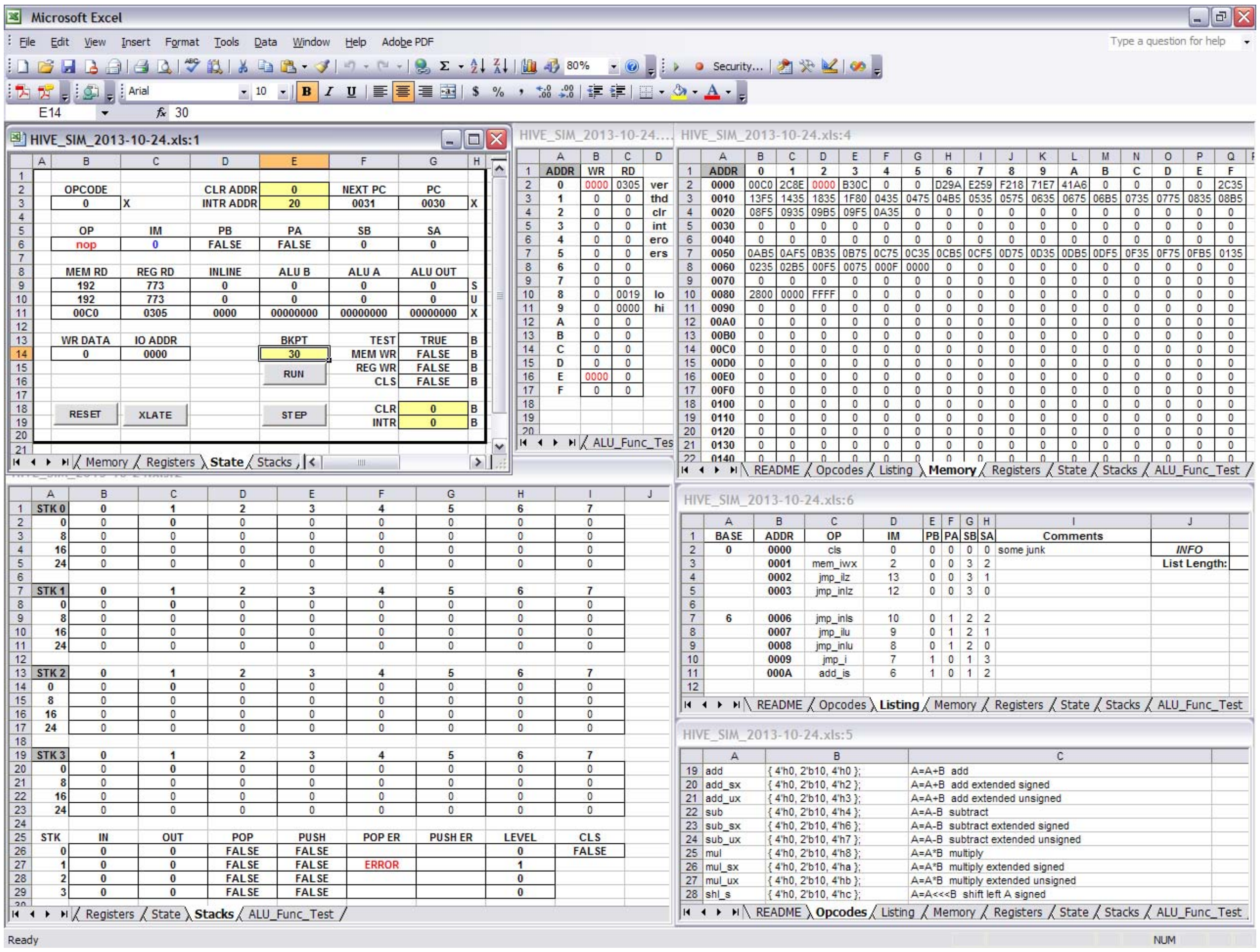
The Hive core can of course be simulated in the native SystemVerilog – the ability to do so is indispensable when exercising the various core components in isolation, as well as the core as a whole with small snippets of boot code. However one is more often interested in the simulation of software running on the core, as opposed to verification of the core logic, but SystemVerilog simulation at this higher level is cumbersome and slow. Software entry and development can be streamlined as well through the use of an external tool.

Early Attempts

I've spent a frightening amount of time working on various simulators for Hive. Besides the obvious utility of simulation itself, writing them gives me new perspectives on the design. While combing back through the source I've uncovered several bugs, and my attention brought to other details that, while not incorrect per se, merited a re-write.

Microsoft Excel

Much state in a processor is tabular in nature: memory, stacks, registers, etc. This led me to imagine that a simulator in a table-based calculator such as Excel might be viable. And I was already quite familiar with Excel from the various electrical, analog, and digital simulations I've used it for. But the ponderous nature of Visual Basic, and the multifarious ways VB vs. Excel have for doing the same thing made for rather tough sledding. I first wrote custom functions to handle the modulo ALU calculations which Excel hated and/or gave incorrect results for, and built off of these functions to decode the opcode fields, maintain the stack pointers, etc. The result is shown below in a screen grab:



Microsoft Excel
File Edit View Insert Format Tools Data Window Help Adobe PDF
Type a question for help
80%
Security...
Arial
10
E14
30
RUN
RESET
XLATE
STEP
Memory Registers State Stacks
ALU_Func_Tes
README Opcodes Listing Memory Registers State Stacks ALU_Func_Test
Registers State Stacks ALU_Func_Test
Ready
NUM

HIVE_SIM_2013-10-24.xls:1

OPCODE		CLR ADDR	0	NEXT PC	PC	
0	X	INTR ADDR	20	0031	0030	X
OP	IM	PB	PA	SB	SA	
nop	0	FALSE	FALSE	0	0	
MEM RD	REG RD	INLINE	ALU B	ALU A	ALU OUT	
192	773	0	0	0	0	S
192	773	0	0	0	0	U
00C0	0305	0000	00000000	00000000	00000000	X
WR DATA	IO ADDR		BKPT	TEST	TRUE	B
0	0000		30	MEM WR	FALSE	B
				REG WR	FALSE	B
				CLS	FALSE	B
				CLR	0	B
				INTR	0	B

HIVE_SIM_2013-10-24....

ADDR	WR	RD	
0	0000	0305	ver
1	0	0	thd
2	0	0	clr
3	0	0	int
4	0	0	ero
5	0	0	ers
6	0	0	
7	0	0	
8	0	0019	lo
9	0	0000	hi
A	0	0	
B	0	0	
C	0	0	
D	0	0	
E	0000	0	
F	0	0	

HIVE_SIM_2013-10-24.xls:4

ADDR	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0000	00C0	2C8E	0000	B30C	0	0	D29A	E259	F218	71E7	41A6	0	0	0	0	2C35
0010	13F5	1435	1835	1F80	0435	0475	04B5	0535	0575	0635	0675	06B5	0735	0775	0835	08B5
0020	08F5	0935	09B5	09F5	0A35	0	0	0	0	0	0	0	0	0	0	0
0030	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0040	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0050	0AB5	0AF5	0B35	0B75	0C75	0C35	0CB5	0CF5	0D75	0D35	0DB5	0DF5	0F35	0F75	0FB5	0135
0060	0235	02B5	00F5	0075	000F	0000	0	0	0	0	0	0	0	0	0	0
0070	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0080	2800	0000	FFFF	0	0	0	0	0	0	0	0	0	0	0	0	0
0090	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
00A0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
00B0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
00C0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
00D0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
00E0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
00F0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0100	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0110	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0120	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0130	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0140	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

STK 0	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0
16	0	0	0	0	0	0	0	0
24	0	0	0	0	0	0	0	0
STK 1	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0
16	0	0	0	0	0	0	0	0
24	0	0	0	0	0	0	0	0
STK 2	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0
16	0	0	0	0	0	0	0	0
24	0	0	0	0	0	0	0	0
STK 3	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0
16	0	0	0	0	0	0	0	0
24	0	0	0	0	0	0	0	0
STK	IN	OUT	POP	PUSH	POP ER	PUSH ER	LEVEL	CLS
0	0	0	FALSE	FALSE			0	FALSE
1	0	0	FALSE	FALSE	ERROR		1	
2	0	0	FALSE	FALSE			0	
3	0	0	FALSE	FALSE			0	

HIVE_SIM_2013-10-24.xls:6

BASE	ADDR	OP	IM	PB	PA	SB	SA	Comments	
0	0000	cls	0	0	0	0	0	some junk	*INFO*
	0001	mem_iwx	2	0	0	3	2		List Length:
	0002	jmp_ilz	13	0	0	3	1		
	0003	jmp_inlz	12	0	0	3	0		
6	0006	jmp_inls	10	0	1	2	2		
	0007	jmp_ilu	9	0	1	2	1		
	0008	jmp_inlu	8	0	1	2	0		
	0009	jmp_i	7	1	0	1	3		
	000A	add_is	6	1	0	1	2		

HIVE_SIM_2013-10-24.xls:5

add	{ 4'h0, 2'b10, 4'h0 };	A=A+B add
add_sx	{ 4'h0, 2'b10, 4'h2 };	A=A+B add extended signed
add_ux	{ 4'h0, 2'b10, 4'h3 };	A=A+B add extended unsigned
sub	{ 4'h0, 2'b10, 4'h4 };	A=A-B subtract
sub_sx	{ 4'h0, 2'b10, 4'h6 };	A=A-B subtract extended signed
sub_ux	{ 4'h0, 2'b10, 4'h7 };	A=A-B subtract extended unsigned
mul	{ 4'h0, 2'b10, 4'h8 };	A=A*B multiply
mul_sx	{ 4'h0, 2'b10, 4'ha };	A=A*B multiply extended signed
mul_ux	{ 4'h0, 2'b10, 4'hb };	A=A*B multiply extended unsigned
shl_s	{ 4'h0, 2'b10, 4'hc };	A=A<<<B shift left A signed

Figure 55. Excel simulation of a single Hive thread.

Given limitless time and patience, one could probably do just about anything in Excel including the simulation of any processor in existence. The real questions are:

1. Does the result have an intuitive and useful interface?
2. Does it run fast enough?
3. Is it maintainable?

From this exercise, the answers I received to these questions were:

1. Sort of. If the separate worksheet windows would open in the same place every time, with the same window size and zoom level, the whole thing would work a lot better. I also discovered that having the same worksheet open in two separate windows would cause any buttons on that worksheet to freeze up, which was often confusing and a minor pain to track down.
2. Marginally. Even though I liberally peppered the VB code with functions to keep the screen from updating when not necessary, it still ran pretty slow. Faster than a SystemVerilog simulation, but that's not saying a lot.
3. No. This is the crux of what made me abandon the Excel / VB approach. The result was too much of a mish-mash of high/low level functions, code, and widgets to really be able to maintain it as the SV code changed.

The above was my experience simulating a single thread, not the full core, in Excel. I was looking at a ton of extra work in expanding the sim to cover the entire core, with diminishing returns in terms of usability and maintainability. Somewhere after version 5 of the core SV I reluctantly abandoned the Excel simulator. Even though it came in handy by uncovering a bug in one of the test algorithms, it always felt kind of shaky and never really struck me as a coherent *thing*.

Python

After some research regarding basic GUI app construction, I turned to this fairly newish language, mainly because of the built-in TKInter support and not the language so much. I got approximately halfway through describing Hive in Python before I abandoned this approach as well. The forced indenting was a minor turn-off, particularly when coupled with the limited line length, but the weak typing and auto-sizing variables were an obvious poor fit for the simulation of hardware which is inherently modulo. The lack of a compiler was the final nail in the coffin. I was fearful that, after (yet again) putting a lot of effort into writing a sim, the result wouldn't run fast enough, and forcing people to install Python on their machines just to run my sim seemed awkward and demanding. Python is natively coded in C/C++, and that is the language I should have turned to in the first place.

C++ Console App (third time's the charm)
Programmers who hang out in web forums dispensing advice will adamantly wave you off if you even think of writing a console app. There are few things that are more portable though. And during my research I discovered commands exist which can be used in the Windows console to size the console itself, move the cursor around, justify the cursor fields, and the set text and background colors. These may be used in lieu of the ANSI escape sequences which are supported in the consoles of most other, saner, operating systems. Call me crazy, but armed with this I set out to describe Hive as a C/C++ console app.

It took me about four months to write the core logic in a modular object oriented way. In retrospect I probably should have just copied the SV as closely as possible, which would have been quicker to do and may have produced a slightly higher fidelity model. But the OO approach is more powerful and flexible, and it forced me to carefully reevaluate the points where the cores interact (memory, and the register set in particular). I spent another three months working on the GUI side of things. I wanted a tool that would not just simulate code, but one that would facilitate interactive code development, and I feel that I've achieved that goal. As I was nearing the end of the simulator development I took the opportunity to completely re-write the verification code for both it and the SV core. This proved to be an excellent exercise that gave me invaluable experience interacting with the tool while working the quirks out of the interface.

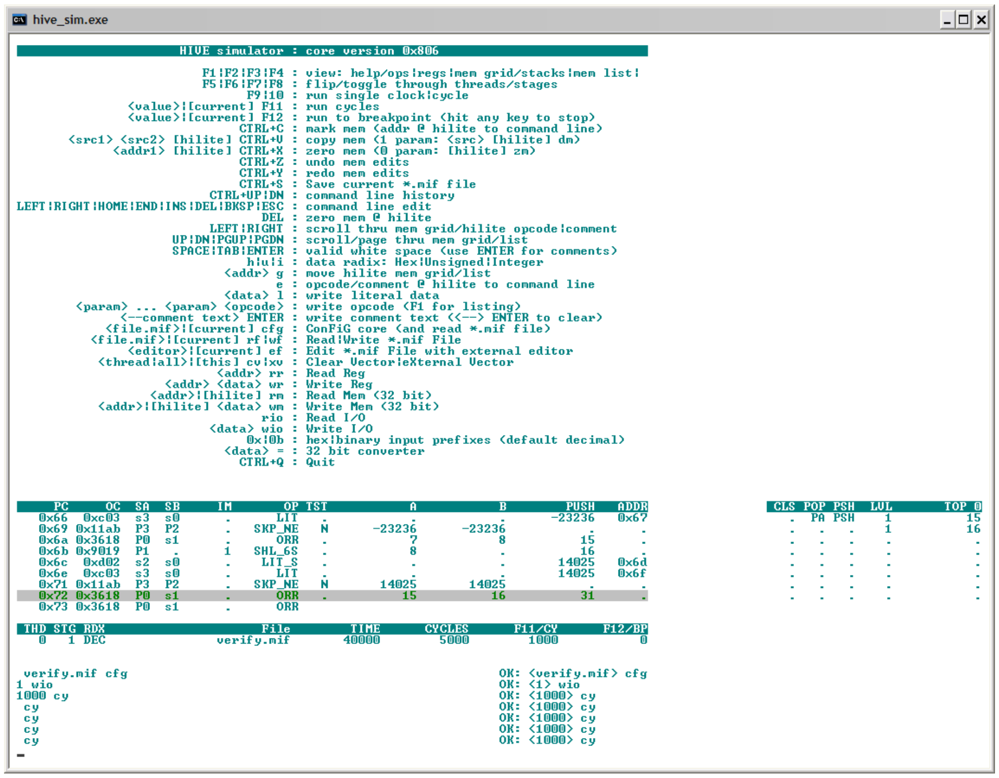
Code Container
For file I/O I discovered that the MIF (*.mif) file format employed by the Quartus tool (to encode memory contents) entirely sufficient for my needs as it is fairly simple to parse and generate, and it supports both block and end-of-line comments. There are provisions in the simulator for entering and editing end-of-line comments, but any start of file block comments must be edited externally - this is because I wanted the simulator to be very memory slot oriented. Start of file comments are always preserved however, as this is a handy place to include background information. The MIF standard allows for compression of repeated lines, and this is done by the simulator only when the lines are contiguous comment-less NOPs. The simulator kicks also kicks out two comment-less "even & odd" files for SV physical dual memory configuration whenever the main file is saved. Writing the MIF parser was fairly eye-opening experience, as it brought into sharp focus the rules associated with comments, white space, numbers, keywords, and general tokenization of input streams. I was able to parlay much of this when it came to implementing the command line interpreter. I found myself re-writing many of the built-in C++ commands for conversion between chars, ints, and strings, as they either didn't behave the way I wanted, or worse caused the program to crash when given unexpected input. (IMO, this type of function should always have an associated function to test for input fitness pre-conversion, which lets everyone know what's going on, and prevents bad data from contaminating / crashing things.)

Command Line Interpreter
My familiarity with Autocad made me desire a robust command line interface. My familiarity with RPN made me desire a postfix command format, wherein the parser holds off until an action keyword is encountered, all parameters are listed before the function, and the function is executed immediately when typed and followed by white space. Going this route forced me to give up doskey support (which relies heavily on the enter key for parsing), so I had to replicate many of the familiar doskey functions from scratch.

Key Commands
I feel that it's important to keep interfaces as familiar as possible, so I endeavored to use as many of the keyboard shortcuts that PC users have become accustomed to, such as doskey line editing and the various CTRL combinations for cut, copy, paste, undo, etc. as possible. For doskey-like command recall I employed the slightly unusual CTRL+up/dn combination in order to leave the arrow keys free for screen navigation. CTRL+C is probably the most unusual in that it places the highlight address on the command line, rather than buffering anything to the clipboard. CTRL+V must be used in conjunction with CTRL+C to copy blocks of code, and it performs a copy rather than an insert (a real insert is generally undesirable as it would move other code that could be relying on absolute addressing). The order of the copy is automatically calculated so as not to destroy any lines in the process of copying them, and doing things this way eliminates the need for a temporary buffer. CTRL+X similarly relies on CTRL+C to mark memory blocks for deletion (implemented as NOP & comment clear). CTRL+Z and CTRL+Y call the undo / redo mechanism, a brute force circular buffer of complete memory images (including comments) which employs a new slot whenever the user (not core software execution) modifies memory / comments. CTRL+S saves the current working file.

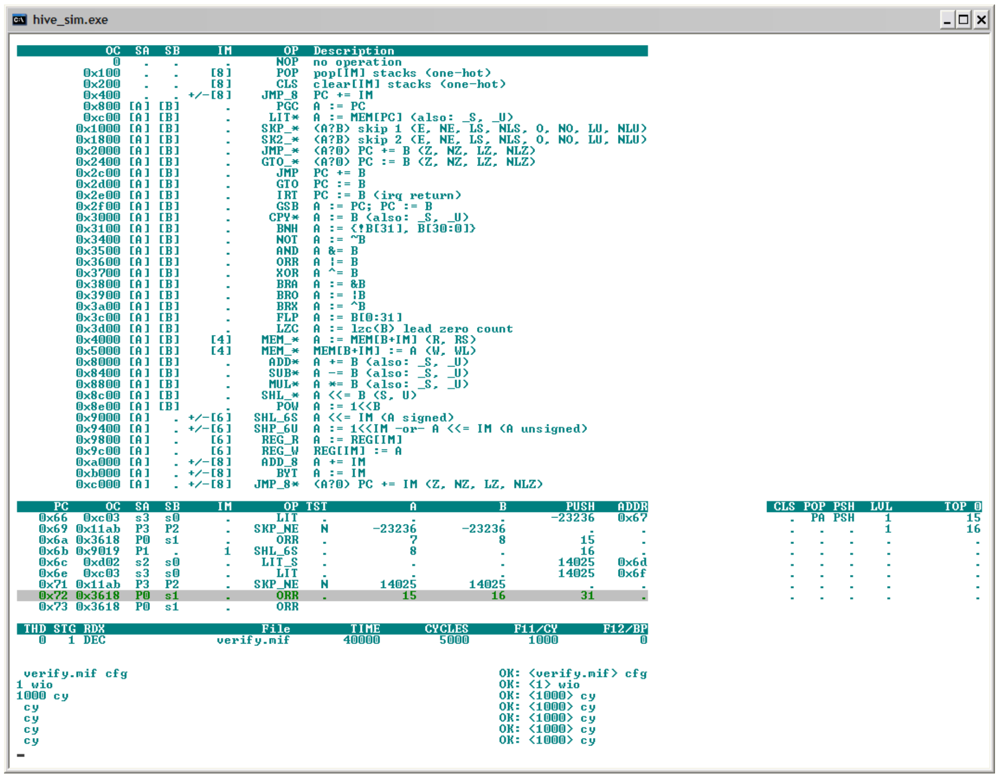
F1 – F4: Views

The function key group F1-F4 calls up a series of views into what is transpiring in the simulated core. F1 calls up a series of three help screens, one for general keys and commands, a second for opcode keywords and formats, and a third for register set descriptions. The selection among the three is via successive pressing of F1, and the last one viewed is "sticky" upon return from other non-help views.



```
                        HIVE simulator : core version 0x806

                        F1|F2|F3|F4 : view: help/ops|regs|mem grid/stacks|mem list|
                        F5|F6|F7|F8 : flip/toggle through threads/stages
                              F9|10 : run single clock|cycle
            <value>|[current] F11 : run cycles
            <value>|[current] F12 : run to breakpoint (hit any key to stop)
                             CTRL+C : mark mem (addr @ hilite to command line)
     <src1> <src2> [hilite] CTRL+V : copy mem (1 param: <src> [hilite] dm)
           <addr1> [hilite] CTRL+X : zero mem (0 param: [hilite] zm)
                             CTRL+Z : undo mem edits
                             CTRL+Y : redo mem edits
                             CTRL+S : Save current *.mif file
                         CTRL+UP|DN : command line history
LEFT|RIGHT|HOME|END|INS|DEL|BKSP|ESC : command line edit
                                DEL : zero mem @ hilite
                         LEFT|RIGHT : scroll thru mem grid/hilite opcode|comment
                     UP|DN|PGUP|PGDN : scroll/page thru mem grid/list
                     SPACE|TAB|ENTER : valid white space (use ENTER for comments)
                              h|u|i : data radix: Hex|Unsigned|Integer
                           <addr> g : move hilite mem grid/list
                                  e : opcode/comment @ hilite to command line
                           <data> l : write literal data
         <param> ... <param> <opcode> : write opcode (F1 for listing)
               <--comment text> ENTER : write comment text (<--> ENTER to clear)
             <file.mif>|[current] cfg : ConFiG core (and read *.mif file)
           <file.mif>|[current] rf|wf : Read|Write *.mif File
               <editor>|[current] ef : Edit *.mif File with external editor
          <thread|all>|[this] cv|xv : Clear Vector|eXternal Vector
                          <addr> rr : Read Reg
                   <addr> <data> wr : Write Reg
                <addr>|[hilite] rm : Read Mem (32 bit)
        <addr>|[hilite] <data> wm : Write Mem (32 bit)
                                rio : Read I/O
                         <data> wio : Write I/O
                              0x|0b : hex|binary input prefixes (default decimal)
                           <data> = : 32 bit converter
                             CTRL+Q : Quit


      PC      OC  SA  SB     IM      OP TST        A          B      PUSH  ADDR          CLS POP PSH  LVL       TOP 0
    0x66  0xc03  s3  s0      .      LIT  .          .          .   -23236  0x67            .  PA PSH   1         15
    0x69 0x11ab  P3  P2      .   SKP_NE  N     -23236     -23236        .     .            .   .   .   1         16
    0x6a 0x3618  P0  s1      .      ORR  .          7          8       15     .            .   .   .   .          .
    0x6b 0x9019  P1   .      1   SHL_6S  .          8          .       16     .            .   .   .   .          .
    0x6c  0xd02  s2  s0      .    LIT_S  .          .          .    14025  0x6d            .   .   .   .          .
    0x6e  0xc03  s3  s0      .      LIT  .          .          .    14025  0x6f            .   .   .   .          .
    0x71 0x11ab  P3  P2      .   SKP_NE  N      14025      14025        .     .            .   .   .   .          .
    0x72 0x3618  P0  s1      .      ORR  .         15         16       31     .            .   .   .   .          .
    0x73 0x3618  P0  s1      .      ORR

 THD STG RDX                File     TIME     CYCLES     F11/CY     F12/BP
   0   1 DEC          verify.mif    40000       5000       1000          0


 verify.mif cfg                                                OK: <verify.mif> cfg
1 wio                                                          OK: <1> wio
1000 cy                                                        OK: <1000> cy
 cy                                                            OK: <1000> cy
 cy                                                            OK: <1000> cy
 cy                                                            OK: <1000> cy
 cy                                                            OK: <1000> cy
_
```

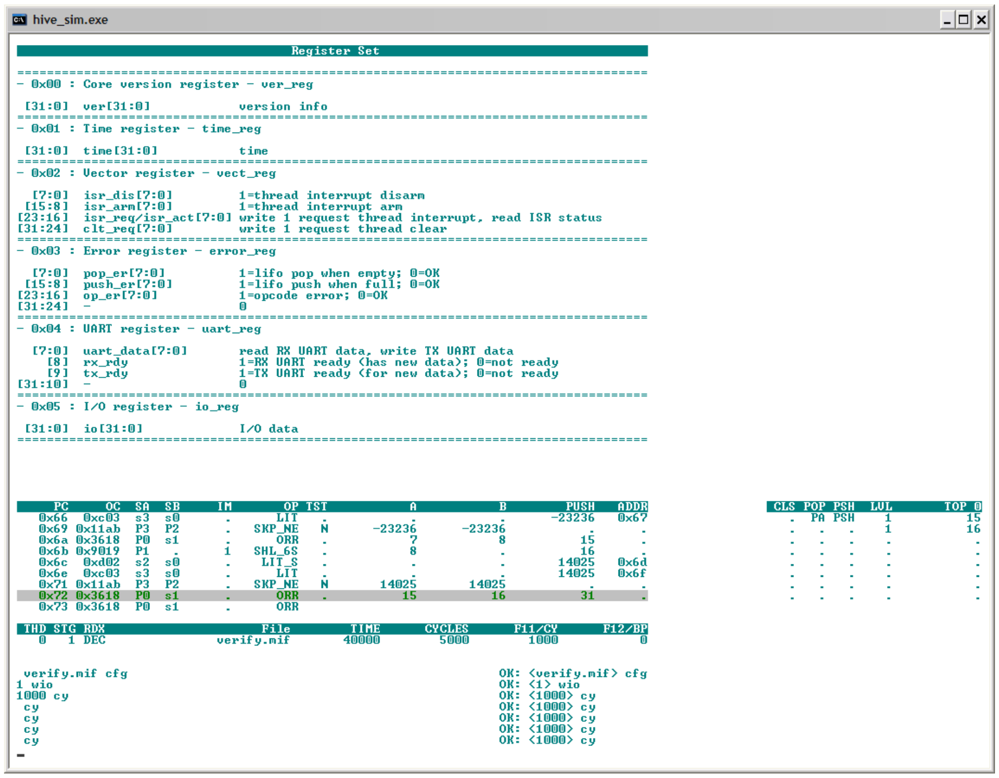
Figure 56. F1 general help.



```
          OC  SA  SB       IM      OP  Description
           0   .   .        .     NOP  no operation
       0x100   .   .      [8]     POP  pop[IM] stacks (one-hot)
       0x200   .   .      [8]     CLS  clear[IM] stacks (one-hot)
       0x400   .   .  +/-[8]    JMP_8  PC += IM
       0x800 [A] [B]        .     PGC  A := PC
       0xc00 [A] [B]        .    LIT*  A := MEM[PC] (also: _S, _U)
      0x1000 [A] [B]        .   SKP_*  (A?B) skip 1 (E, NE, LS, NLS, O, NO, LU, NLU)
      0x1800 [A] [B]        .   SK2_*  (A?B) skip 2 (E, NE, LS, NLS, O, NO, LU, NLU)
      0x2000 [A] [B]        .   JMP_*  (A?0) PC += B (Z, NZ, LZ, NLZ)
      0x2400 [A] [B]        .   GTO_*  (A?0) PC := B (Z, NZ, LZ, NLZ)
      0x2c00 [A] [B]        .     JMP  PC += B
      0x2d00 [A] [B]        .     GTO  PC := B
      0x2e00 [A] [B]        .     IRT  PC := B (irq return)
      0x2f00 [A] [B]        .     GSB  A := PC; PC := B
      0x3000 [A] [B]        .    CPY*  A := B (also: _S, _U)
      0x3100 [A] [B]        .     BNH  A := {!B[31], B[30:0]}
      0x3400 [A] [B]        .     NOT  A := ~B
      0x3500 [A] [B]        .     AND  A &= B
      0x3600 [A] [B]        .     ORR  A |= B
      0x3700 [A] [B]        .     XOR  A ^= B
      0x3800 [A] [B]        .     BRA  A := &B
      0x3900 [A] [B]        .     BRO  A := |B
      0x3a00 [A] [B]        .     BRX  A := ^B
      0x3c00 [A] [B]        .     FLP  A := B[0:31]
      0x3d00 [A] [B]        .     LZC  A := lzc(B) lead zero count
      0x4000 [A] [B]      [4]   MEM_*  A := MEM[B+IM] (R, RS)
      0x5000 [A] [B]      [4]   MEM_*  MEM[B+IM] := A (W, WL)
      0x8000 [A] [B]        .    ADD*  A += B (also: _S, _U)
      0x8400 [A] [B]        .    SUB*  A -= B (also: _S, _U)
      0x8800 [A] [B]        .    MUL*  A *= B (also: _S, _U)
      0x8c00 [A] [B]        .   SHL_*  A <<= B (S, U)
      0x8e00 [A] [B]        .     POW  A := 1<<B
      0x9000 [A]   .  +/-[6]   SHL_6S  A <<= IM (A signed)
      0x9400 [A]   .  +/-[6]   SHP_6U  A := 1<<IM -or- A <<= IM (A unsigned)
      0x9800 [A]   .      [6]   REG_R  A := REG[IM]
      0x9c00 [A]   .      [6]   REG_W  REG[IM] := A
      0xa000 [A]   .  +/-[8]    ADD_8  A += IM
      0xb000 [A]   .  +/-[8]      BYT  A := IM
      0xc000 [A]   .  +/-[8]   JMP_8*  (A?0) PC += IM (Z, NZ, LZ, NLZ)
```

```
      PC      OC  SA  SB     IM      OP TST        A          B      PUSH  ADDR          CLS POP PSH  LVL       TOP 0
    0x66  0xc03  s3  s0      .      LIT  .          .          .   -23236  0x67            .  PA PSH   1         15
    0x69 0x11ab  P3  P2      .   SKP_NE  N     -23236     -23236        .     .            .   .   .   1         16
    0x6a 0x3618  P0  s1      .      ORR  .          7          8       15     .            .   .   .   .          .
    0x6b 0x9019  P1   .      1   SHL_6S  .          8          .       16     .            .   .   .   .          .
    0x6c  0xd02  s2  s0      .    LIT_S  .          .          .    14025  0x6d            .   .   .   .          .
    0x6e  0xc03  s3  s0      .      LIT  .          .          .    14025  0x6f            .   .   .   .          .
    0x71 0x11ab  P3  P2      .   SKP_NE  N      14025      14025        .     .            .   .   .   .          .
    0x72 0x3618  P0  s1      .      ORR  .         15         16       31     .            .   .   .   .          .
    0x73 0x3618  P0  s1      .      ORR

 THD STG RDX                File     TIME     CYCLES     F11/CY     F12/BP
   0   1 DEC          verify.mif    40000       5000       1000          0


 verify.mif cfg                                                OK: <verify.mif> cfg
1 wio                                                          OK: <1> wio
1000 cy                                                        OK: <1000> cy
 cy                                                            OK: <1000> cy
 cy                                                            OK: <1000> cy
 cy                                                            OK: <1000> cy
 cy                                                            OK: <1000> cy
_
```

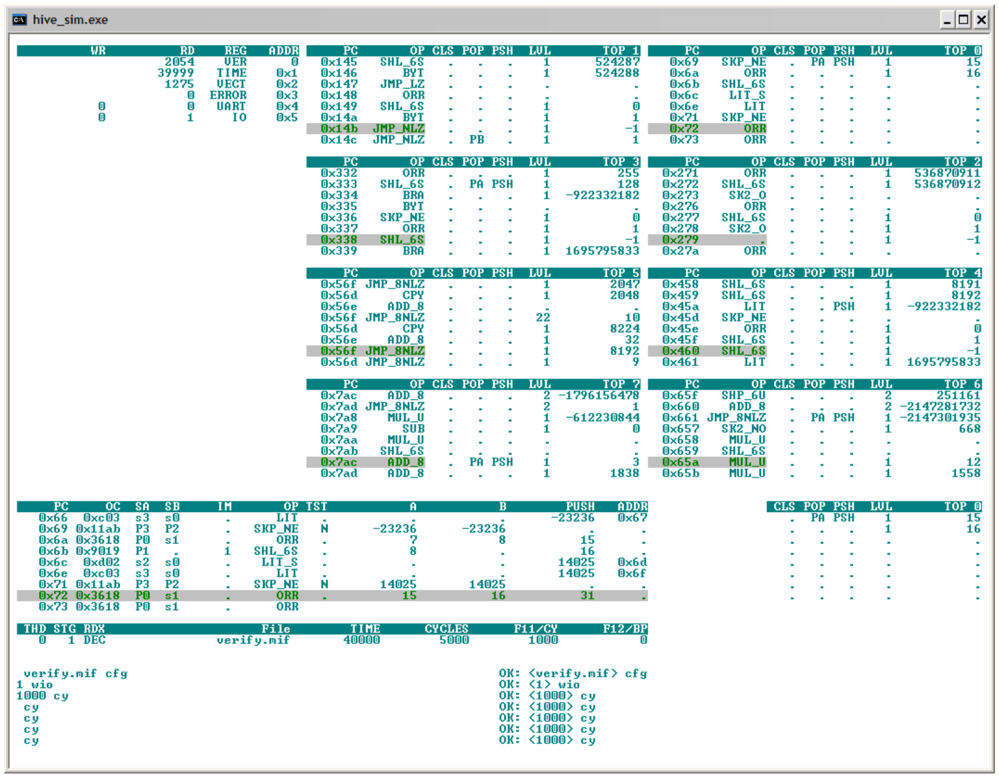
Figure 57. F1 opcode help.



```
hive_sim.exe
                                    Register Set
=======================================================================================
- 0x00 : Core version register - ver_reg

 [31:0]  ver[31:0]              version info
=======================================================================================
- 0x01 : Time register - time_reg

 [31:0]  time[31:0]             time
=======================================================================================
- 0x02 : Vector register - vect_reg

  [7:0]  isr_dis[7:0]           1=thread interrupt disarm
 [15:8]  isr_arm[7:0]           1=thread interrupt arm
[23:16]  isr_req/isr_act[7:0] write 1 request thread interrupt, read ISR status
[31:24]  clt_req[7:0]           write 1 request thread clear
=======================================================================================
- 0x03 : Error register - error_reg

  [7:0]  pop_er[7:0]            1=lifo pop when empty; 0=OK
 [15:8]  push_er[7:0]           1=lifo push when full; 0=OK
[23:16]  op_er[7:0]             1=opcode error; 0=OK
[31:24]  -                      0
=======================================================================================
- 0x04 : UART register - uart_reg

  [7:0]  uart_data[7:0]         read RX UART data, write TX UART data
    [8]  rx_rdy                 1=RX UART ready (has new data); 0=not ready
    [9]  tx_rdy                 1=TX UART ready (for new data); 0=not ready
[31:10]  -                      0
=======================================================================================
- 0x05 : I/O register - io_reg

 [31:0]  io[31:0]               I/O data
=======================================================================================



      PC     OC  SA  SB      IM      OP TST          A           B     PUSH  ADDR             CLS POP PSH  LVL      TOP 0
    0x66  0xc03  s3  s0       .     LIT   .          .           .   -23236  0x67               .  PA PSH    1         15
    0x69 0x11ab  P3  P2       .  SKP_NE   N     -23236      -23236        .     .               .   .   .    1         16
    0x6a 0x3618  P0  s1       .     ORR   .          7           8       15     .               .   .   .    .          .
    0x6b 0x9019  P1   .       1  SHL_6S   .          8           .       16     .               .   .   .    .          .
    0x6c  0xd02  s2  s0       .   LIT_S   .          .           .    14025  0x6d               .   .   .    .          .
    0x6e  0xc03  s3  s0       .     LIT   .          .           .    14025  0x6f               .   .   .    .          .
    0x71 0x11ab  P3  P2       .  SKP_NE   N      14025       14025        .     .               .   .   .    .          .
    0x72 0x3618  P0  s1       .     ORR   .         15          16       31     .               .   .   .    .          .
    0x73 0x3618  P0  s1       .     ORR                                                         .   .   .    .          .

 THD STG RDX                 File      TIME      CYCLES     F11/CY     F12/BP
   0   1 DEC           verify.mif     40000        5000       1000          0


 verify.mif cfg                                                  OK: <verify.mif> cfg
1 wio                                                            OK: <1> wio
1000 cy                                                          OK: <1000> cy
 cy                                                              OK: <1000> cy
 cy                                                              OK: <1000> cy
 cy                                                              OK: <1000> cy
 cy                                                              OK: <1000> cy
_
```

Figure 58. F1 register set help.

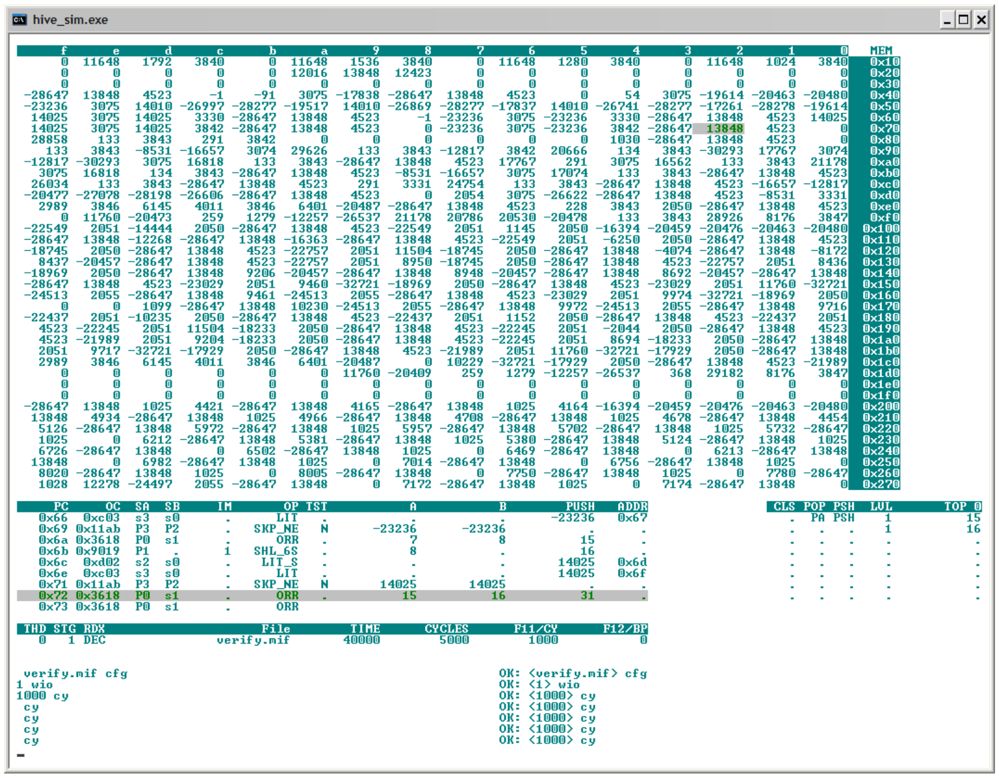
F2 calls up a single view of the register set, as well as “mini-state” views of each thread which shows PC and operation history, as well as stack state.



```
hive_sim.exe
          WR          RD    REG  ADDR      PC      OP CLS POP PSH  LVL      TOP 1      PC      OP CLS POP PSH  LVL      TOP 0
                    2054    VER     0   0x145  SHL_6S   .   .   .    1     524287    0x69  SKP_NE   .  PA PSH    1         15
                   39999   TIME   0x1   0x146     BYT   .   .   .    1     524288    0x6a     ORR   .   .   .    1         16
                    1275   VECT   0x2   0x147  JMP_LZ   .   .   .    .          .    0x6b  SHL_6S   .   .   .    .          .
                       0  ERROR   0x3   0x148     ORR   .   .   .    .          .    0x6c   LIT_S   .   .   .    .          .
           0           0   UART   0x4   0x149  SHL_6S   .   .   .    1          0    0x6e     LIT   .   .   .    .          .
           0           1     IO   0x5   0x14a     BYT   .   .   .    1          1    0x71  SKP_NE   .   .   .    .          .
                                        0x14b JMP_NLZ   .   .   .    1         -1    0x72     ORR   .   .   .    .          .
                                        0x14c JMP_NLZ   .  PB   .    1          1    0x73     ORR   .   .   .    .          .

                                           PC      OP CLS POP PSH  LVL      TOP 3      PC      OP CLS POP PSH  LVL      TOP 2
                                        0x332     ORR   .   .   .    1        255   0x271     ORR   .   .   .    1  536870911
                                        0x333  SHL_6S   .  PA PSH    1        128   0x272  SHL_6S   .   .   .    1  536870912
                                        0x334     BRA   .   .   .    1 -922332182   0x273   SK2_O   .   .   .    .          .
                                        0x335     BYT   .   .   .    .          .   0x276     ORR   .   .   .    .          .
                                        0x336  SKP_NE   .   .   .    1          0   0x277  SHL_6S   .   .   .    1          0
                                        0x337     ORR   .   .   .    1          1   0x278   SK2_O   .   .   .    1          1
                                        0x338  SHL_6S   .   .   .    1         -1   0x279       .   .   .   .    1         -1
                                        0x339     BRA   .   .   .    1 1695795833   0x27a     ORR   .   .   .    .          .

                                           PC      OP CLS POP PSH  LVL      TOP 5      PC      OP CLS POP PSH  LVL      TOP 4
                                        0x56f JMP_8NLZ  .   .   .    1       2047   0x458  SHL_6S   .   .   .    1       8191
                                        0x56d     CPY   .   .   .    1       2048   0x459  SHL_6S   .   .   .    1       8192
                                        0x56e   ADD_8   .   .   .    .          .   0x45a     LIT   .   .  PSH   1 -922332182
                                        0x56f JMP_8NLZ  .   .   .   22         10   0x45d  SKP_NE   .   .   .    .          .
                                        0x56d     CPY   .   .   .    1       8224   0x45e     ORR   .   .   .    1          0
                                        0x56e   ADD_8   .   .   .    1         32   0x45f  SHL_6S   .   .   .    1          1
                                        0x56f JMP_8NLZ  .   .   .    1       8192   0x460  SHL_6S   .   .   .    1         -1
                                        0x56d JMP_8NLZ  .   .   .    1          9   0x461     LIT   .   .   .    1 1695795833

                                           PC      OP CLS POP PSH  LVL      TOP 7      PC      OP CLS POP PSH  LVL      TOP 6
                                        0x7ac   ADD_8   .   .   .    2 -1796156478  0x65f  SHP_6U   .   .   .    2     251161
                                        0x7ad JMP_8NLZ  .   .   .    2          1   0x660   ADD_8   .   .   .    2 -2147281732
                                        0x7a8   MUL_U   .   .   .    1 -612230844   0x661 JMP_8NLZ  .  PA PSH    1 -2147301935
                                        0x7a9     SUB   .   .   .    1          0   0x657  SK2_NO   .   .   .    1        668
                                        0x7aa   MUL_U   .   .   .    .          .   0x658   MUL_U   .   .   .    .          .
                                        0x7ab  SHL_6S   .   .   .    .          .   0x659  SHL_6S   .   .   .    .          .
                                        0x7ac   ADD_8   .  PA PSH    1          3   0x65a   MUL_U   .   .   .    1         12
                                        0x7ad   ADD_8   .   .   .    1       1838   0x65b   MUL_U   .   .   .    1       1558

      PC     OC  SA  SB      IM      OP TST          A           B     PUSH  ADDR             CLS POP PSH  LVL      TOP 0
    0x66  0xc03  s3  s0       .     LIT   .          .           .   -23236  0x67               .  PA PSH    1         15
    0x69 0x11ab  P3  P2       .  SKP_NE   N     -23236      -23236        .     .               .   .   .    1         16
    0x6a 0x3618  P0  s1       .     ORR   .          7           8       15     .               .   .   .    .          .
    0x6b 0x9019  P1   .       1  SHL_6S   .          8           .       16     .               .   .   .    .          .
    0x6c  0xd02  s2  s0       .   LIT_S   .          .           .    14025  0x6d               .   .   .    .          .
    0x6e  0xc03  s3  s0       .     LIT   .          .           .    14025  0x6f               .   .   .    .          .
    0x71 0x11ab  P3  P2       .  SKP_NE   N      14025       14025        .     .               .   .   .    .          .
    0x72 0x3618  P0  s1       .     ORR   .         15          16       31     .               .   .   .    .          .
    0x73 0x3618  P0  s1       .     ORR                                                         .   .   .    .          .

 THD STG RDX                 File      TIME      CYCLES     F11/CY     F12/BP
   0   1 DEC           verify.mif     40000        5000       1000          0


 verify.mif cfg                                                  OK: <verify.mif> cfg
1 wio                                                            OK: <1> wio
1000 cy                                                          OK: <1000> cy
 cy                                                              OK: <1000> cy
 cy                                                              OK: <1000> cy
 cy                                                              OK: <1000> cy
 cy                                                              OK: <1000> cy
```

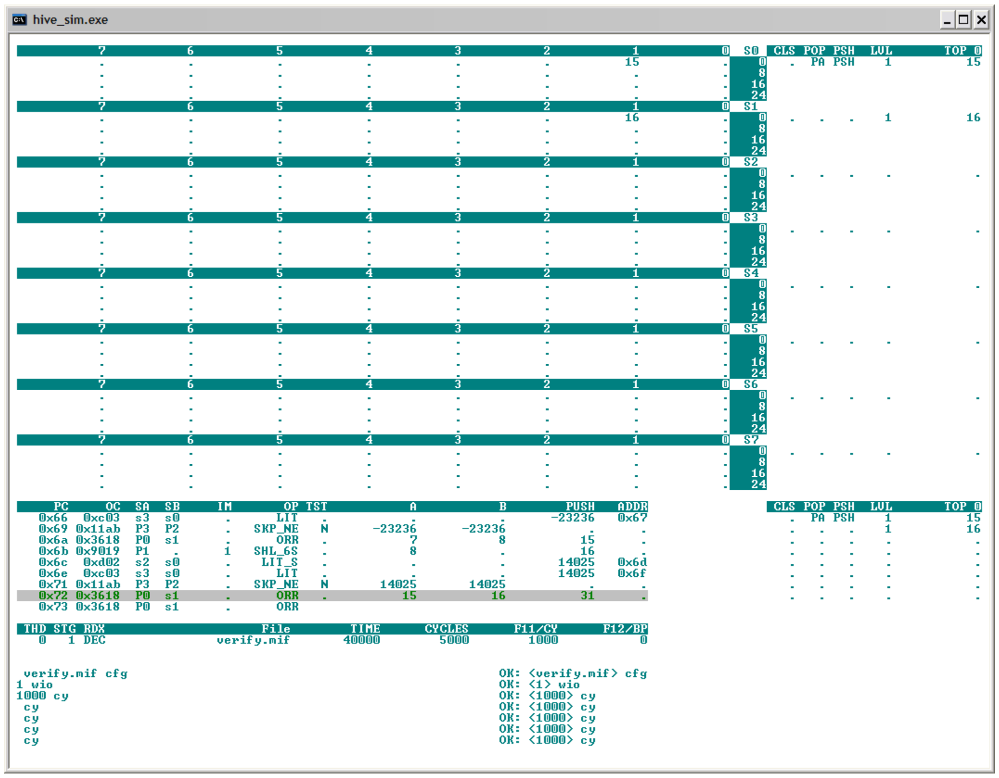
Figure 59. F2 register set & “mini-state”.

F3 calls up two memory view screens, one a grid view of the main memory, the other a grid view of all stack memories for the given thread. The selection among the two is via successive pressing of F3, and the last one viewed is “sticky” upon return from other non-memory views.



```
hive_sim.exe
     f      e      d      c      b      a      9      8      7      6      5      4      3      2      1      0   MEM
     0  11648   1792   3840      0  11648   1536   3840      0  11648   1280   3840      0  11648   1024   3840  0x10
     0      0      0      0      0  12016  13848  12423      0      0      0      0      0      0      0      0  0x20
     0      0      0      0      0      0      0      0      0      0      0      0      0      0      0      0  0x30
-28647  13848   4523     -1    -91   3075 -17838 -28647  13848   4523      0     54   3075 -19614 -20463 -20480  0x40
-23236   3075  14010 -26997 -28277 -19517  14010 -26869 -28277 -17837  14010 -26741 -28277 -17261 -28278 -19614  0x50
 14025   3075  14025   3330 -28647  13848   4523     -1 -23236   3075 -23236   3330 -28647  13848   4523  14025  0x60
 14025   3075  14025   3842 -28647  13848   4523      0 -23236   3075 -23236   3842 -28647  13848   4523      0  0x70
 28858    133   3843    291   3842      0      0      0      0      0      0   1030 -28647  13848   4523      0  0x80
   133   3843  -8531 -16657   3074  29626    133   3843 -12817   3842  20666    134   3843 -30293  17767   3074  0x90
-12817 -30293   3075  16818    133   3843 -28647  13848   4523  17767    291   3075  16562    133   3843  21178  0xa0
  3075  16818    134   3843 -28647  13848   4523  -8531 -16657   3075  17074    133   3843 -28647  13848   4523  0xb0
 26034    133   3843 -28647  13848   4523    291   3331  24754    133   3843 -28647  13848   4523 -16657 -12817  0xc0
-20477 -27078 -28198 -26606 -28647  13848   4523      0   2054   3075 -26622 -28647  13848   4523  -8531   3331  0xd0
  2989   3846   6145   4011   3846   6401 -20487 -28647  13848   4523    228   3843   2050 -28647  13848   4523  0xe0
     0  11760 -20473    259   1279 -12257 -26537  21178  20786  20530 -20478    133   3843  28926   8176   3847  0xf0
-22549   2051 -14444   2050 -28647  13848   4523 -22549   2051   1145   2050 -16394 -20459 -20476 -20463 -20480  0x100
-28647  13848 -12268 -28647  13848 -16363 -28647  13848   4523 -22549   2051  -6250   2050 -28647  13848   4523  0x110
-18745   2050 -28647  13848   4523 -22757   2051  11504 -18745   2050 -28647  13848  -4074 -28647  13848  -8172  0x120
  8437 -20457 -28647  13848   4523 -22757   2051   8950 -18745   2050 -28647  13848   4523 -22757   2051   8436  0x130
-18969   2050 -28647  13848   9206 -20457 -28647  13848   8948 -20457 -28647  13848   8692 -20457 -28647  13848  0x140
-28647  13848   4523 -23029   2051   9460 -32721 -18969   2050 -28647  13848   4523 -23029   2051  11760 -32721  0x150
-24513   2055 -28647  13848   9461 -24513   2055 -28647  13848   4523 -23029   2051   9974 -32721 -18969   2050  0x160
     0      0   1099 -28647  13848  10230 -24513   2055 -28647  13848   9972 -24513   2055 -28647  13848   9716  0x170
-22437   2051 -10235   2050 -28647  13848   4523 -22437   2051   1152   2050 -28647  13848   4523 -22437   2051  0x180
  4523 -22245   2051  11504 -18233   2050 -28647  13848   4523 -22245   2051  -2044   2050 -28647  13848   4523  0x190
  4523 -21989   2051   9204 -18233   2050 -28647  13848   4523 -22245   2051   8694 -18233   2050 -28647  13848  0x1a0
  2051   9717 -32721 -17929   2050 -28647  13848   4523 -21989   2051  11760 -32721 -17929   2050 -28647  13848  0x1b0
  2989   3846   6145   4011   3846   6401 -20487      0  10229 -32721 -17929   2050 -28647  13848   4523 -21989  0x1c0
     0      0      0      0      0      0  11760 -20409    259   1279 -12257 -26537    368  29182   8176   3847  0x1d0
     0      0      0      0      0      0      0      0      0      0      0      0      0      0      0      0  0x1e0
     0      0      0      0      0      0      0      0      0      0      0      0      0      0      0      0  0x1f0
-28647  13848   1025   4421 -28647  13848   4165 -28647  13848   1025   4164 -16394 -20459 -20476 -20463 -20480  0x200
 13848   4934 -28647  13848   1025   4966 -28647  13848   4708 -28647  13848   1025   4678 -28647  13848   4454  0x210
  5126 -28647  13848   5972 -28647  13848   1025   5957 -28647  13848   5702 -28647  13848   1025   5732 -28647  0x220
  1025      0   6212 -28647  13848   5381 -28647  13848   1025   5380 -28647  13848   5124 -28647  13848   1025  0x230
  6726 -28647  13848      0   6502 -28647  13848   1025      0   6469 -28647  13848      0   6213 -28647  13848  0x240
 13848      0   6982 -28647  13848   1025      0   7014 -28647  13848      0   6756 -28647  13848   1025      0  0x250
  8020 -28647  13848   1025      0   8005 -28647  13848      0   7750 -28647  13848   1025      0   7780 -28647  0x260
  1028  12278 -24497   2055 -28647  13848      0   7172 -28647  13848   1025      0   7174 -28647  13848      0  0x270

    PC     OC  SA  SB     IM     OP TST       A         B     PUSH  ADDR           CLS POP PSH  LVL      TOP 0
  0x66  0xc03  s3  s0      .    LIT   .       .         .   -23236  0x67             .  PA PSH    1         15
  0x69 0x11ab  P3  P2      . SKP_NE   N  -23236    -23236        .     .             .   .   .    1         16
  0x6a 0x3618  P0  s1      .    ORR   .       7         8       15     .             .   .   .    .          .
  0x6b 0x9019  P1   .      1 SHL_6S   .       8         .       16     .             .   .   .    .          .
  0x6c  0xd02  s2  s0      .  LIT_S   .       .         .    14025  0x6d             .   .   .    .          .
  0x6e  0xc03  s3  s0      .    LIT   .       .         .    14025  0x6f             .   .   .    .          .
  0x71 0x11ab  P3  P2      . SKP_NE   N   14025     14025        .     .             .   .   .    .          .
  0x72 0x3618  P0  s1      .    ORR   .      15        16       31     .             .   .   .    .          .
  0x73 0x3618  P0  s1      .    ORR

THD STG RDX              File     TIME    CYCLES    F11/CY    F12/BP
  0   1 DEC        verify.mif    40000      5000      1000         0


 verify.mif cfg                                                  OK: <verify.mif> cfg
1 wio                                                            OK: <1> wio
1000 cy                                                          OK: <1000> cy
 cy                                                              OK: <1000> cy
 cy                                                              OK: <1000> cy
 cy                                                              OK: <1000> cy
 cy                                                              OK: <1000> cy
_
```

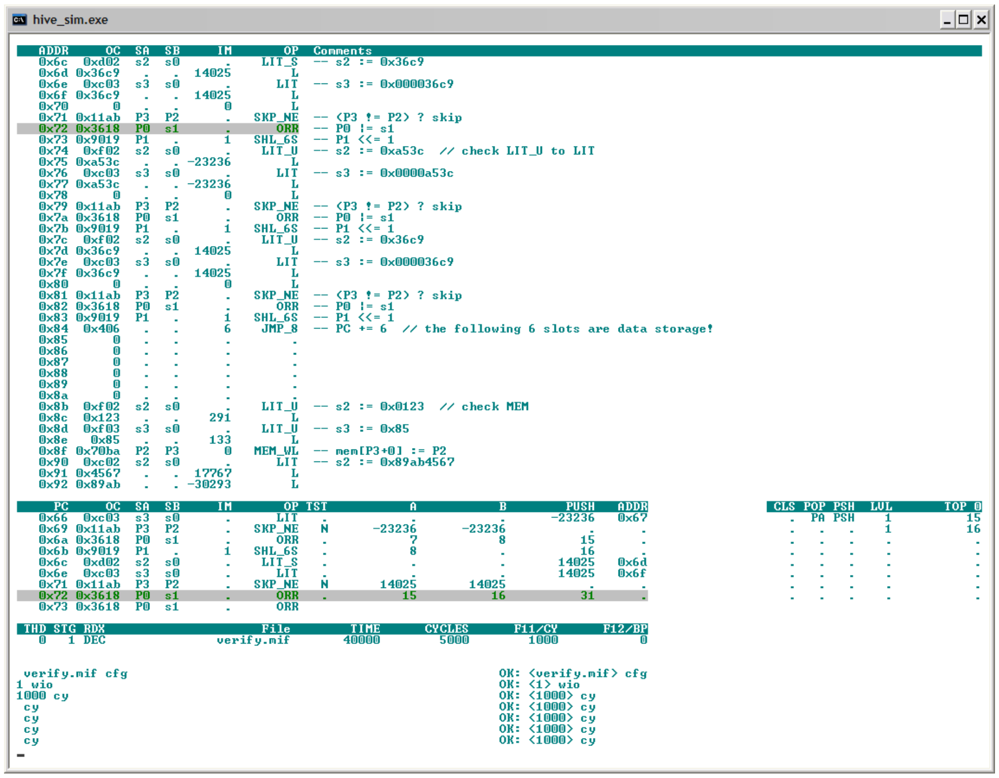
Figure 60. F3 main memory grid.



```
hive_sim.exe
         7          6          5          4          3          2          1          0   S0   CLS POP PSH  LVL      TOP 0
         .          .          .          .          .          .         15          .    0    .  PA PSH    1         15
         .          .          .          .          .          .          .          .    8
         .          .          .          .          .          .          .          .   16
         .          .          .          .          .          .          .          .   24
         7          6          5          4          3          2          1          0   S1
         .          .          .          .          .          .         16          .    0    .   .   .    1         16
         .          .          .          .          .          .          .          .    8
         .          .          .          .          .          .          .          .   16
         .          .          .          .          .          .          .          .   24
         7          6          5          4          3          2          1          0   S2
         .          .          .          .          .          .          .          .    0    .   .   .    .          .
         .          .          .          .          .          .          .          .    8
         .          .          .          .          .          .          .          .   16
         .          .          .          .          .          .          .          .   24
         7          6          5          4          3          2          1          0   S3
         .          .          .          .          .          .          .          .    0    .   .   .    .          .
         .          .          .          .          .          .          .          .    8
         .          .          .          .          .          .          .          .   16
         .          .          .          .          .          .          .          .   24
         7          6          5          4          3          2          1          0   S4
         .          .          .          .          .          .          .          .    0    .   .   .    .          .
         .          .          .          .          .          .          .          .    8
         .          .          .          .          .          .          .          .   16
         .          .          .          .          .          .          .          .   24
         7          6          5          4          3          2          1          0   S5
         .          .          .          .          .          .          .          .    0    .   .   .    .          .
         .          .          .          .          .          .          .          .    8
         .          .          .          .          .          .          .          .   16
         .          .          .          .          .          .          .          .   24
         7          6          5          4          3          2          1          0   S6
         .          .          .          .          .          .          .          .    0    .   .   .    .          .
         .          .          .          .          .          .          .          .    8
         .          .          .          .          .          .          .          .   16
         .          .          .          .          .          .          .          .   24
         7          6          5          4          3          2          1          0   S7
         .          .          .          .          .          .          .          .    0    .   .   .    .          .
         .          .          .          .          .          .          .          .    8
         .          .          .          .          .          .          .          .   16
         .          .          .          .          .          .          .          .   24

    PC     OC  SA  SB     IM     OP TST       A         B     PUSH  ADDR           CLS POP PSH  LVL      TOP 0
  0x66  0xc03  s3  s0      .    LIT   .       .         .   -23236  0x67             .  PA PSH    1         15
  0x69 0x11ab  P3  P2      . SKP_NE   N  -23236    -23236        .     .             .   .   .    1         16
  0x6a 0x3618  P0  s1      .    ORR   .       7         8       15     .             .   .   .    .          .
  0x6b 0x9019  P1   .      1 SHL_6S   .       8         .       16     .             .   .   .    .          .
  0x6c  0xd02  s2  s0      .  LIT_S   .       .         .    14025  0x6d             .   .   .    .          .
  0x6e  0xc03  s3  s0      .    LIT   .       .         .    14025  0x6f             .   .   .    .          .
  0x71 0x11ab  P3  P2      . SKP_NE   N   14025     14025        .     .             .   .   .    .          .
  0x72 0x3618  P0  s1      .    ORR   .      15        16       31     .             .   .   .    .          .
  0x73 0x3618  P0  s1      .    ORR

THD STG RDX              File     TIME    CYCLES    F11/CY    F12/BP
  0   1 DEC        verify.mif    40000      5000      1000         0


 verify.mif cfg                                                  OK: <verify.mif> cfg
1 wio                                                            OK: <1> wio
1000 cy                                                          OK: <1000> cy
 cy                                                              OK: <1000> cy
 cy                                                              OK: <1000> cy
 cy                                                              OK: <1000> cy
 cy                                                              OK: <1000> cy
```

Figure 61. F3 stacks memory grid.

F4 calls up a main memory disassembly list view with comments for a given thread. This is where the bulk of user interaction takes place.



```
hive_sim.exe

  ADDR     OC  SA  SB      IM      OP  Comments
  0x6c  0xd02  s2  s0       .   LIT_S  -- s2 := 0x36c9
  0x6d 0x36c9   .   .   14025       L
  0x6e  0xc03  s3  s0       .     LIT  -- s3 := 0x000036c9
  0x6f 0x36c9   .   .   14025       L
  0x70      0   .   .       0       L
  0x71 0x11ab  P3  P2       .  SKP_NE  -- (P3 != P2) ? skip
  0x72 0x3618  P0  s1       .     ORR  -- P0 |= s1
  0x73 0x9019  P1   .       1  SHL_6S  -- P1 <<= 1
  0x74  0xf02  s2  s0       .   LIT_U  -- s2 := 0xa53c  // check LIT_U to LIT
  0x75 0xa53c   .   .  -23236       L
  0x76  0xc03  s3  s0       .     LIT  -- s3 := 0x0000a53c
  0x77 0xa53c   .   .  -23236       L
  0x78      0   .   .       0       L
  0x79 0x11ab  P3  P2       .  SKP_NE  -- (P3 != P2) ? skip
  0x7a 0x3618  P0  s1       .     ORR  -- P0 |= s1
  0x7b 0x9019  P1   .       1  SHL_6S  -- P1 <<= 1
  0x7c  0xf02  s2  s0       .   LIT_U  -- s2 := 0x36c9
  0x7d 0x36c9   .   .   14025       L
  0x7e  0xc03  s3  s0       .     LIT  -- s3 := 0x000036c9
  0x7f 0x36c9   .   .   14025       L
  0x80      0   .   .       0       L
  0x81 0x11ab  P3  P2       .  SKP_NE  -- (P3 != P2) ? skip
  0x82 0x3618  P0  s1       .     ORR  -- P0 |= s1
  0x83 0x9019  P1   .       1  SHL_6S  -- P1 <<= 1
  0x84  0x406   .   .       6   JMP_8  -- PC += 6  // the following 6 slots are data storage!
  0x85      0   .   .       .       .
  0x86      0   .   .       .       .
  0x87      0   .   .       .       .
  0x88      0   .   .       .       .
  0x89      0   .   .       .       .
  0x8a      0   .   .       .       .
  0x8b  0xf02  s2  s0       .   LIT_U  -- s2 := 0x0123  // check MEM
  0x8c  0x123   .   .     291       L
  0x8d  0xf03  s3  s0       .   LIT_U  -- s3 := 0x85
  0x8e   0x85   .   .     133       L
  0x8f 0x70ba  P2  P3       0  MEM_WL  -- mem[P3+0] := P2
  0x90  0xc02  s2  s0       .     LIT  -- s2 := 0x89ab4567
  0x91 0x4567   .   .   17767       L
  0x92 0x89ab   .   .  -30293       L

    PC     OC  SA  SB      IM      OP TST        A        B     PUSH  ADDR          CLS POP PSH LVL      TOP 0
  0x66  0xc03  s3  s0       .     LIT   .        .        .   -23236  0x67           .   PA PSH   1         15
  0x69 0x11ab  P3  P2       .  SKP_NE   N   -23236   -23236        .     .           .   .   .    1         16
  0x6a 0x3618  P0  s1       .     ORR   .        7        8       15     .           .   .   .    .          .
  0x6b 0x9019  P1   .       1  SHL_6S   .        8        .       16     .           .   .   .    .          .
  0x6c  0xd02  s2  s0       .   LIT_S   .        .        .    14025  0x6d           .   .   .    .          .
  0x6e  0xc03  s3  s0       .     LIT   .        .        .    14025  0x6f           .   .   .    .          .
  0x71 0x11ab  P3  P2       .  SKP_NE   N    14025    14025        .     .           .   .   .    .          .
  0x72 0x3618  P0  s1       .     ORR   .       15       16       31     .           .   .   .    .          .
  0x73 0x3618  P0  s1       .     ORR

 THD STG RDX                   File     TIME   CYCLES   F11/CY  F12/BP
   0   1 DEC             verify.mif    40000     5000     1000       0


  verify.mif cfg                                                 OK: <verify.mif> cfg
 1 wio                                                           OK: <1> wio
 1000 cy                                                         OK: <1000> cy
  cy                                                             OK: <1000> cy
  cy                                                             OK: <1000> cy
  cy                                                             OK: <1000> cy
  cy                                                             OK: <1000> cy
 _
```

Figure 62. F4 main memory disassembly and comments listing.

F5 – F8: Thread Focus

This group controls the thread focus for the various views. F5 sets the current actively displayed thread (the thread for which information pertaining to it is displayed below the F1-F4 views, and for which F11 & F12 commands apply) to thread 0 or thread 1 and toggles between them in a “sticky” manner. F6 thru F8 are the same but for threads 2/3, 4/5, and 6/7 respectively.

F9 – F12: Code Execution Control

This group controls the running of the core or code execution. F9 executes a single core clock. F10 executes a single core cycle (8 core clocks). F11 executes multiple core cycles, using either the last cycle count value or a new value. F12 executes to given breakpoint for the actively displayed thread, again using either the last value or a new value.

Fixed Display Area

The area below the function key selected screens always displays the recent execution history and current stack status of the thread selected via F5-F8, as well as the recent command history and the command line itself at the very bottom. The command line history shows input on the left and response on the right.

Highlight Bar

There is a highlight in the main memory views that follows program execution for the current thread. This highlight can also be manually moved about via the arrow keys and the page up/down keys. The highlight is the command line focus for address marking and editing, which conveniently enables these features without having to rely on a separate edit view or mode.

APPENDIX B : THE TEN MINUTE SIM TOUR

The following will hopefully be a fairly painless intro to the Hive console app simulator. I assume you've read the appendix that came before this one as it provides some useful background.

Startup – Files & Fonts

The simulator is a single self-contained executable file named "hive_sim.exe". Stick it anywhere on a Windows machine (developed on XP, tested a bit on Win7/64) and run it. By default the sim attempts to open "verify.mif" which should be in the same directory (if indeed you want that file opened). If you want to open some other file it can be specified on the command line when invoking the simulator, or you can wait until the simulator starts up and proceed to load whatever file you want at that point. Failure to open and parse "verify.mif" shouldn't cause any trouble. For the purposes of this tutorial I'll assume "verify.mif" has been successfully loaded.

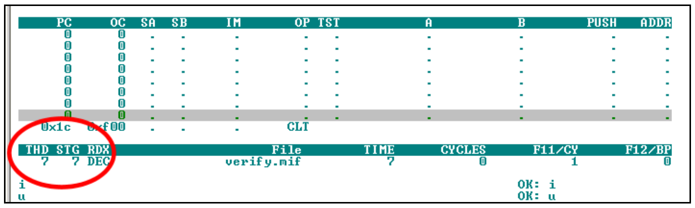
If you get a message at startup that suggests you to adjust the console font, you need to make the font smaller in order for the console to fit on your desktop. Otherwise you will get the message every time you start and the screen will look all jumbled up. An 8x12 raster font seems to work well on my PC with HD monitor. Change the default font by right clicking on the top bar of the console window to bring up the "Properties" menu. Both XP and Win7 seems to remember the font defaults associated with various programs so you only have to do this once. Also, make sure the "Screen Buffer Size" width and height (132 x 66) are the same as those of the "Window Size" so that scrolling doesn't take place.

Views

- Press the F1 key repeatedly to cycle through the various help screens.
- Press F2 to see the register set / mini-state view. Use the arrow keys to select thread 7.
- Press F3 repeatedly to toggle between the main memory and stack memory grid views.
- Press F4 to see the main memory listing view. Stay on this view for now.

Thread / Stage Selection

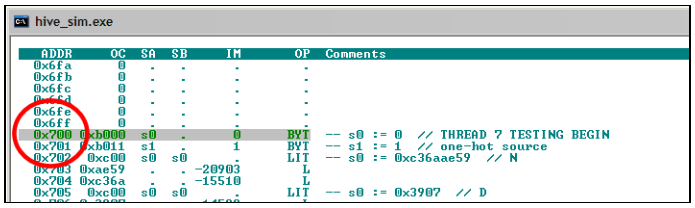
- We want to see thread 7 execute a particular subroutine, so if thread 7 isn't selected press F8 until it is.
- Press F9 to clock the core until it is at stage 7. We do this to be past the fetch point in the pipeline, so the future execution history display will be accurate. If you go too far just keep pressing F9 until stage 7 comes around again.



```
     PC       OC  SA  SB     IM       OP  TST          A            B        PUSH  ADDR
      0        0   .   .      .        .   .           .            .           .     .
      0        0   .   .      .        .   .           .            .           .     .
      0        0   .   .      .        .   .           .            .           .     .
      0        0   .   .      .        .   .           .            .           .     .
      0        0   .   .      .        .   .           .            .           .     .
      0        0   .   .      .        .   .           .            .           .     .
      0        0   .   .      .        .   .           .            .           .     .
      0        0   .   .      .        .   .           .            .           .     .
   0x1c    0xf00   .   .      .      CLT

 THD STG RDX                       File       TIME      CYCLES      F11/CY      F12/BP
   7   7 DEC                 verify.mif          7           0           1           0

i                                                                  OK: i
u                                                                  OK: u
```

Single Stepping

- Press F10 to single step until thread 7 gets to address 0x700.
- Continue pressing F10 and note that the thread jumps to the DIV/MOD subroutine at address 0x760.



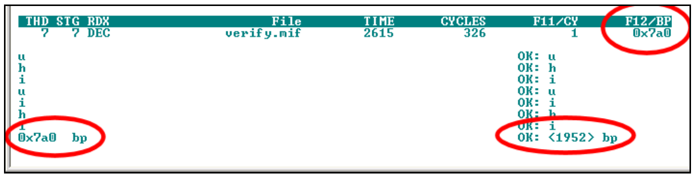
```
hive_sim.exe

   ADDR       OC  SA  SB     IM       OP  Comments
  0x6fa        0   .   .      .        .
  0x6fb        0   .   .      .        .
  0x6fc        0   .   .      .        .
  0x6fd        0   .   .      .        .
  0x6fe        0   .   .      .        .
  0x6ff        0   .   .      .        .
  0x700  0xb000   s0   .      0      BYT  -- s0 := 0  // THREAD 7 TESTING BEGIN
  0x701  0xb011   s1   .      1      BYT  -- s1 := 1  // one-hot source
  0x702   0xc00   s0  s0      .      LIT  -- s0 := 0xc36aae59  // N
  0x703  0xae59    .   . -20903        L
  0x704  0xc36a    .   . -15510        L
  0x705   0xc00   s0  s0      .      LIT  -- s0 := 0x3907  // D
```

Running To A Breakpoint

We want to keep going past the DIV/MOD subroutine in order to see the INV_F subroutine in action. Single stepping would take many tedious cycles to get through it all so we run to a breakpoint.

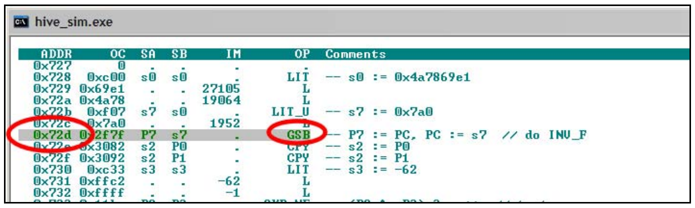
- Type the following command (without quotes): "0x7a0"

- Press F12. Note the addition of “ bp ” to the command address, the confirmation of your command on the right, and the updating of the default F12 value.



```
THD STG RDX                        File       TIME      CYCLES       F11/CY      F12/BP
  7   7 DEC                  verify.mif       2615         326            1       0x7a0

u                                                                    OK: u
h                                                                    OK: h
i                                                                    OK: i
u                                                                    OK: u
i                                                                    OK: i
h                                                                    OK: h
i                                                                    OK: i
0x7a0  bp                                                            OK: <1952> bp
```

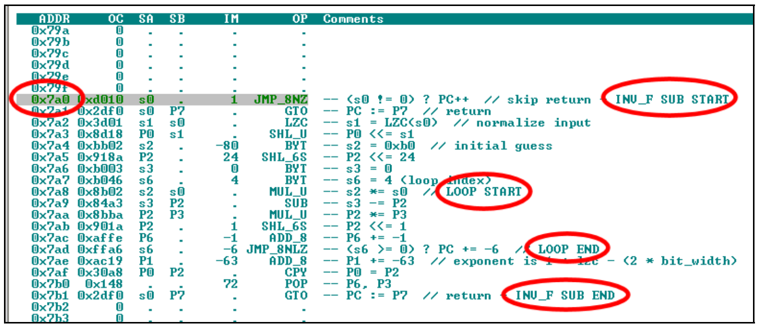
The sim should now be at the address immediately before the execution of the opcode at address 0x7a0, which is a GSB or subroutine instruction located at address 0x72d.



hive_sim.exe

```
   ADDR      OC   SA  SB      IM       OP  Comments
  0x727       0    .   .       .        .
  0x728   0xc00   s0  s0       .      LIT  -- s0 := 0x4a7869e1
  0x729  0x69e1    .   .   27105        L
  0x72a  0x4a78    .   .   19064        L
  0x72b   0xf07   s7  s0       .    LIT_U  -- s7 := 0x7a0
  0x72c   0x7a0    .   .    1952        L
  0x72d  0x2f7f   P7  s7       .      GSB  -- P7 := PC, PC := s7  // do INV_F
  0x72e  0x3082   s2  P0       .      CPY  -- s2 := P0
  0x72f  0x3092   s2  P1       .      CPY  -- s2 := P1
  0x730   0xc33   s3  s3       .      LIT  -- s3 := -62
  0x731  0xffc2    .   .     -62        L
  0x732  0xffff    .   .      -1        L
```

- Press F10 to single step. This should immediately take you to address 0x7a0 which is the start of the INV_F subroutine (floating point inverse using Newton’s Method).
- Press F10 some more and note the looping behavior within the subroutine. The loop should execute a total of 5 times. Stop pressing F10 when the sim hits the end of the subroutine at address 0x7b1.

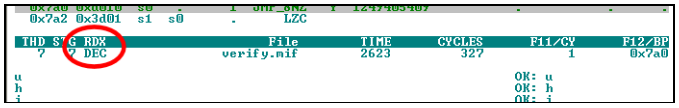


```
   ADDR      OC   SA  SB      IM       OP  Comments
  0x79a       0    .   .       .        .
  0x79b       0    .   .       .        .
  0x79c       0    .   .       .        .
  0x79d       0    .   .       .        .
  0x79e       0    .   .       .        .
  0x79f       0    .   .       .        .
  0x7a0  0xd010   s0   .       1  JMP_8NZ  -- (s0 != 0) ? PC++  // skip return  INV_F SUB START
  0x7a1  0x2df0   s0  P7       .      GTO  -- PC := P7  // return
  0x7a2  0x3d01   s1  s0       .      LZC  -- s1 = LZC(s0)  // normalize input
  0x7a3  0x8d18   P0  s1       .    SHL_U  -- P0 <<= s1
  0x7a4  0xbb02   s2   .     -80      BYT  -- s2 = 0xb0  // initial guess
  0x7a5  0x918a   P2   .      24   SHL_6S  -- P2 <<= 24
  0x7a6  0xb003   s3   .       0      BYT  -- s3 = 0
  0x7a7  0xb046   s6   .       4      BYT  -- s6 = 4 (loop index)
  0x7a8  0x8b02   s2  s0       .    MUL_U  -- s2 *= s0  // LOOP START
  0x7a9  0x84a3   s3  P2       .      SUB  -- s3 -= P2
  0x7aa  0x8bba   P2  P3       .    MUL_U  -- P2 *= P3
  0x7ab  0x901a   P2   .       1   SHL_6S  -- P2 <<= 1
  0x7ac  0xaffe   P6   .      -1    ADD_8  -- P6 += -1
  0x7ad  0xffa6   s6   .      -6 JMP_8NLZ  -- (s6 >= 0) ? PC += -6  // LOOP END
  0x7ae  0xac19   P1   .     -63    ADD_8  -- P1 += -63  // exponent is 1 + lzc - (2 * bit_width)
  0x7af  0x30a8   P0  P2       .      CPY  -- P0 = P2
  0x7b0   0x148    .   .      72      POP  -- P6, P3
  0x7b1  0x2df0   s0  P7       .      GTO  -- PC := P7  // return  INV_F SUB END
  0x7b2       0    .   .       .        .
  0x7b3       0    .   .       .        .
```

Data Display Radix

Simulation addresses are always displayed in hexadecimal, but we can conveniently view data in three different radixes.

- Type the following command (without quotes; and with a space, enter, or tab at the end): “h”



```
  0x7a0  0xd010   s0   .       1  JMP_8NZ  Y  1247405407          .          .      .
  0x7a2  0x3d01   s1  s0       .      LZC

THD STG RDX                        File       TIME      CYCLES       F11/CY      F12/BP
  7   7 DEC                  verify.mif       2623         327            1       0x7a0

u                                                                    OK: u
h                                                                    OK: h
i                                                                    OK: i
```

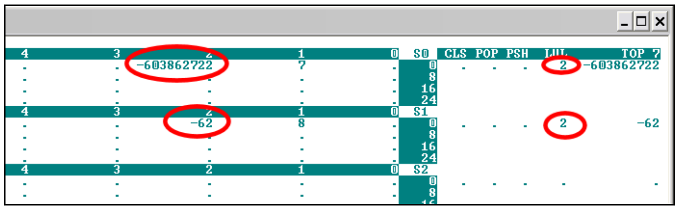
Note the change in displayed data radix to hexadecimal (i.e. base 16).

- Type the command “u“ for unsigned decimal (i.e. base 10) display of data.
- Type the command “i“ for signed decimal display of data.

Stacks View

Let’s examine the results of running the INV_F subroutine.

- Press F3 until you see the stacks memory grid view.



```
4          3          2          1          0  S0  CLS POP PSH  LUL       TOP 7
.          .  -603862722         7          .   0    .   .   .    2  -603862722
.          .          .          .          .   8
.          .          .          .          .  16
.          .          .          .          .  24
4          3          2          1          0  S1
.          .        -62          8          .   0    .   .   .    2         -62
.          .          .          .          .   8
.          .          .          .          .  16
.          .          .          .          .  24
4          3          2          1          0  S2
.          .          .          .          .   0    .   .   .    .           .
.          .          .          .          .   8
```

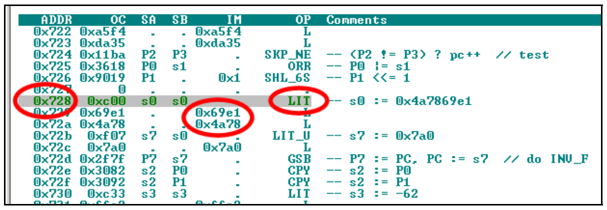
Stack 0 is at the top, stack 1 below it, etc. A summary of the stack states is on the right, details of the contents on the left. Keep in mind that the first item on an empty stack is at location 1 (not 0) and the last item on a full stack is at location 0 (not 31). Above we see that stack 0 has two items on it, the value 7 is the running verification test result for this thread, and the value -603862772 is the subroutine output. Stack 1 also has two items on it, the value 8 is the running verification test value for this thread, and the value -62 is subroutine output. -62 is the signed power of 2 denormalization value, and -603862772 is actually an unsigned value. What is it?

- Type the command “u“ for unsigned decimal display of data.

Now we see that -603862772 in unsigned decimal radix is actually 3691104574. Does any of this make sense? Well, the input value to the subroutine can be found in the calling code at address 0x728.

- Press F4 to return to the main memory listing view.
- Press F10 until subroutine return, which should be address 0x72e.
- Use the arrow keys to move the highlight to address 0x728.
- Type the command “h“ for hexadecimal decimal display of data.

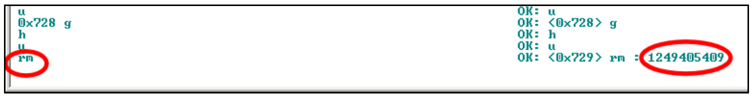
At address 0x728 we see the in-line literal value 0x4a7869e1 being pushed to stack 0.



```
  ADDR      OC   SA   SB       IM       OP  Comments
 0x722  0xa5f4    .    .   0xa5f4        L
 0x723  0xda35    .    .   0xda35        L
 0x724  0x11ba   P2   P3        .   SKP_NE  -- (P2 != P3) ? pc++  // test
 0x725  0x3618   P0   s1        .      ORR  -- P0 |= s1
 0x726  0x9019   P1    .      0x1   SHL_6S  -- P1 <<= 1
 0x727       0    .    .        .        .
 0x728   0xc00   s0   s0               LIT  -- s0 := 0x4a7869e1
 0x729  0x69e1    .        0x69e1        L
 0x72a  0x4a78    .    .   0x4a78        L
 0x72b   0xf07   s7   s0        .    LIT_U  -- s7 := 0x7a0
 0x72c   0x7a0    .    .    0x7a0        L
 0x72d  0x2f7f   P7   s7        .      GSB  -- P7 := PC, PC := s7  // do INV_F
 0x72e  0x3082   s2   P0        .      CPY  -- s2 := P0
 0x72f  0x3092   s2   P1        .      CPY  -- s2 := P1
 0x730   0xc33   s3   s3        .      LIT  -- s3 := -62
```

What is this as an unsigned decimal?

- Press the down arrow once to move the highlight to address 0x729.
- Type the command “u“ for unsigned decimal display of data.
- Type the command “rm” to do a 32 bit memory read at the highlight address.



```
u                                   OK: u
0x728 g                             OK: <0x728> g
h                                   OK: h
u                                   OK: u
rm                                  OK: <0x729> rm :1249405409
```

Here we see the decimal value of the input to the subroutine is 1249405409. Getting out the calculator:

$$1/1249405409 = 8.00380719338 * 10^{-10}$$

The result found in the stacks view was:

$$3691104574 * 2^{-62} = 8.003807196 * 10^{-10}$$

If you do these calculations as integers in a spread sheet you’ll find that the result here is actually only one count off from the rounded ideal. Newton’s method converges quite quickly and is the method employed in AMD processors to do division and such.

Thread Clearing, Editing Data & Comments, Save File

Let’s change the input value of the INV_F subroutine and re-run it.

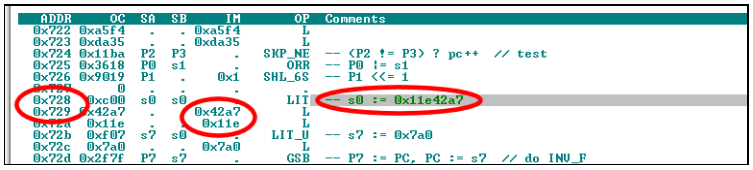
- Type the command “cv” to clear the current thread (7).
- Press F10 several times until you see execution jump back to the starting address (0x1c for thread 7).

Let’s browse our way to the literal address via the main memory grid view where we can cover ground more quickly than in the main memory listing view.

- Press F3 until you see the main memory grid view.
- Press the Page Down key 7 times, which should bring the highlight to the vicinity of address 0x700.
- Use the arrow keys to move the highlight to address 0x729.
- Press F4 to return to the main memory listing view. The highlight should be at address 0x729 here as well.

Let’s pick the new input number to be 18760359 (0x11e42a7). Consulting my spreadsheet, the inverse of this should be 3840949634 * 2^{-56}.

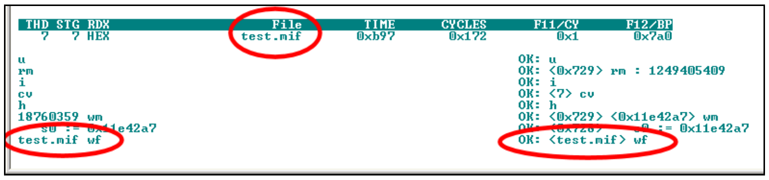
- Type the command “18760359 wm” to do a 32 bit memory write.
- Press the up arrow to move the highlight back to address 0x728.
- Press the right arrow to move the highlight to the comment field.
- Type the command “e” to put the existing comment on the command line.
- Edit the comment to reflect the new number (0x11e42a7) and press ENTER to finish.



```
 ADDR      OC   SA  SB      IM        OP  Comments
 0x722 0xa5f4    .   .  0xa5f4         L
 0x723 0xda35    .   .  0xda35         L
 0x724 0x11ba   P2  P3       .    SKP_NE  -- (P2 != P3) ? pc++   // test
 0x725 0x3618   P0  s1       .       ORR  -- P0 != s1
 0x726 0x9019   P1   .     0x1    SHL_6S  -- P1 <<= 1
 0x727       0   .   .       .         .
 0x728  0xc00   s0  s0       .       LIT  -- s0 := 0x11e42a7
 0x729  0x42a7   .   .  0x42a7         L
 0x72a   0x11e   .   .   0x11e         L
 0x72b   0xf07  s7  s0       .     LIT_U  -- s7 := 0x7a0
 0x72c   0x7a0   .   .   0x7a0         L
 0x72d 0x2f7f   P7  s7       .       GSB  -- P7 := PC, PC := s7  // do INV_F
```

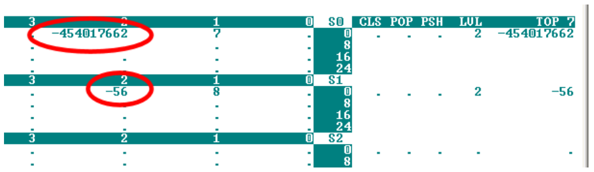
Editing works identically for both opcodes and comments, it only depends on whether the highlight is on the left or on the right when you type the “e” command. ENTER is only required at the end of editing (or entering) a comment, but is also useful when editing an opcode as it returns the cursor to the end of the line before adding white space.

- Save your work to a new file by typing the command “test.mif wf”



```
 THD STG RDX                    File       TIME     CYCLES      F11/CY      F12/BP
   7   7 HEX                 test.mif     0xb97      0x172         0x1       0x7a0

 u                                                         OK: u
 rm                                                        OK: <0x729> rm : 1249405409
 i                                                         OK: i
 cv                                                        OK: <7> cv
 h                                                         OK: h
 18760359 wm                                               OK: <0x729> <0x11e42a7> wm
   s0 := 0x11e42a7                                         OK: <0x728>   s0 := 0x11e42a7
 test.mif wf                                               OK: <test.mif> wf
```

- Press F12 to run to breakpoint 0x7a0.
- Press F10 once to actually execute to 0x7a0, which is the beginning of the INV_F subroutine.
- Type the command “0x7b1” F12 to set a new breakpoint address and run to it.
- Press F10 once to actually execute to 0x7b1.
- Press F3 until you see the stacks memory grid view.



```
   3          2          1          0   S0  CLS POP PSH LVL        TOP 7
   .  -454017662          7          .    0   .   .   .    2  -454017662
   .          .          .          .    8
   .          .          .          .   16
   .          .          .          .   24
   3          2          1          0   S1
   .        -56          8          .    0   .   .   .    2         -56
   .          .          .          .    8
   .          .          .          .   16
   .          .          .          .   24
   3          2          1          0   S2
   .          .          .          .    0   .   .   .    .           .
   .          .          .          .    8
```

Using the radix commands we see -56 as the exponent, and the value 3840949634 (-454017662) which is exactly what we expected.

Copy & Delete
Let’s move the INV_F subroutine to a new location and change the calling link to match.

- Navigate to address 0x7a0 either manually or by typing “0x7a0 g”.
- Press F4 to see the main memory listing view.
- Make sure the highlight is on address 0x7a0 (the first line of the subroutine) and press CTRL+C.
- Using the arrow keys, move the highlight down to address 0x7b1 (the last line of the subroutine) and again press CTRL+C.

Note that CTRL+C puts the highlight address on the command line, which we don’t want to disturb (typing would disturb it, pressing ESC would clear it). Now we select a destination address, say 0x7c0:

- Using the arrow keys, move the highlight down to address 0x7c0 and press CTRL+V.

A copy of the subroutine at 0x7a0 through 0x7b1 should now exist also 0x7c0. We need to delete the original to complete the move:

- Move the highlight back to address 0x7a0 and press CTRL+C.
- Move the highlight down to address 0x7b1 and press CTRL+X.

That should have deleted the original. Now edit the calling code to have the correct address:

- Page up / arrow up to address 0x72c.
- Ensure the highlight is on the opcode (left arrow) and type the command “e”.
- Edit the literal value to “0x7c0 L” and press ENTER.
- Move the highlight to the comment at address 0x72c (up and right arrow) and type the command “e”.
- Edit the comment to reflect the new address and press the ENTER key to finish.
- CTRL+S to save the file.

Check for proper functioning now that the subroutine has been moved:

- Type the command “cv” to clear the current thread (7).
- Press F10 several times until you see execution jump back to the starting address (0x1c for thread 7).
- Type the command “0x7d1” F12 to run to breakpoint 0x7d1.
- Press F10 once to actually execute to 0x7d1.
- Press F3 to examine the stacks memory grid view. You should again see the values 3840949634 and -56 at the tops of stacks 0 and 1 respectively.

Note that pressing the DELETE key (with a blank command line) when viewing the memory grid or listing will delete the opcode and comment at the highlight address and jump to the next line. You don’t want to delete large blocks of code this way because each deletion event consumes a slot in the undo system, and you can quickly run out of slots when performing single line changes to memory.

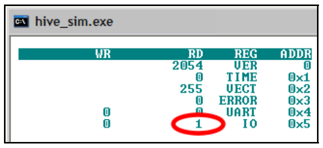
Core Reconfiguration
We can restore the core to the “power up” reconfig state:

- Type the command “cfg”.

This reloads the *.mif file and pretty much resets everything, including the stack memories, thread focus, breakpoint address, register set data, etc.

Speed Testing
How fast is the C++ console sim compared to the real hardware FPGA instantiation? The verification program will run forever with all threads doing real work if the IO port reads non-zero.

- Load the verification MIF file and configure the core by typing the command “verify.mif cfg”.
- Make the IO input non-zero by typing the command “1 wio”.
- Press F2 to confirm the write to the IO input (or type “5 rr” to read the IO register).



```
              WR              RD     REG   ADDR
                            2054     VER      0
                               0    TIME    0x1
                             255    VECT    0x2
                               0   ERROR    0x3
               0               0    UART    0x4
               0               1      IO    0x5
```

- Type the command “1000000” then press the F11 key. Time how long it takes for the command line to become active again.

On my current PC (AMD Athlon II X2 250, 3 GHz, 2.75 GB) this takes about 10 seconds. Since F11 performs cycles, the actual clocks are 8 times this or 8 million. $8 * 10^6 / 10$ sec = 800 kHz. The FPGA version of Hive can do ~200 MHz, so the software simulator is running at about 0.4% of the hardware speed.

This sound kind of shabby but really isn’t, particularly when compared to Quartus functional simulation, where a 210,000 ns @ 50 ns / clock sim of the SV core running the verification program takes around 90 seconds. 210,000 ns / 50 ns = 4200 clocks; 4200 / 90 sec = 46 Hz! 800 kHz / 46 Hz = 17,391 times slower than the C++ sim. It seems you spend your precious life writing efficient sims, or spend it waiting on profoundly less efficient sims to *finish already!*

APPENDIX C : PROGRAMMING EXAMPLES

The SystemVerilog hardware description language has an “initial” construct that can be used along with other SV syntax features to write legible boot code, comments and all. Hive boot code text used to reside in a text file (boot_code.sv) that got inserted into the main memory module with an include statement. (It now resides in a *.mif file.) Let us take a look at some sample SV initial type boot code:

```
   import hive_params::*;
   import hive_defines::*;
```

These includes pull in our opcode encoding and internal address register locations so we can refer to them by name rather than by their rather cryptic numerical encoding.

```
   integer i, j;
   logic               [DATA_W-1:0]        ram[0:MEM_DEPTH-1];  // temp memory

initial begin

   // zero out memory arrays
   ram = '{ MEM_DEPTH{'0} };  // temp memory
   ram0 = '{ DEPTH{'0} };
   ram1 = '{ DEPTH{'0} };
```

The above marks the beginning of the initialization code, declares the integers we will use to keep from having to name every address, instantiates a temporary RAM array, and zeros it and the real RAM arrays out.

A Simple SV Initial Example

```
   ///////////////
   // clt space //
   ///////////////
   // thread 0
   i='h0;   ram[i] = { `lit_u ,          `__, `s7 };  // s7 := 0x0040
   i=i+1;   ram[i] =                    16'h0040  ;  //
   i=i+1;   ram[i] = { `gsb,             `P7, `s3 };  // s3 := PC; PC := P7
   i=i+1;   ram[i] = { `jmp_8,        -4'd1, `s0 };  // loop forever
   // others loop forever
   i='h04;  ram[i] = { `jmp_8,        -4'd1, `s0 };  // loop forever
   i='h08;  ram[i] = { `jmp_8,        -4'd1, `s0 };  // loop forever
   i='h0c;  ram[i] = { `jmp_8,        -4'd1, `s0 };  // loop forever
   i='h10;  ram[i] = { `jmp_8,        -4'd1, `s0 };  // loop forever
   i='h14;  ram[i] = { `jmp_8,        -4'd1, `s0 };  // loop forever
   i='h18;  ram[i] = { `jmp_8,        -4'd1, `s0 };  // loop forever
   i='h1c;  ram[i] = { `jmp_8,        -4'd1, `s0 };  // loop forever

   // code & data space //

   // sub : read core version & write to GPIO, return to (s3)
   i='h40;  ram[i] = { `reg_r,     `VER_ADDR, `s0 };  // s0 := reg[VER]
   i=i+1;   ram[i] = { `reg_w ,  `GPIO_ADDR, `P0 };  // reg[GPIO] : = P0
   i=i+1;   ram[i] = { `gto,             `P3, `__ };  // PC := P3
```

The first line is located at address 0, which is where thread 0 vectors to when cleared. The instruction puts an unsigned literal in S7, the value of which is the address of a subroutine. The second line is the unsigned in-line literal value, 0x0040. The third line calls the subroutine and pushes the return address to S3, and it simultaneously pops the subroutine address in S7 (stack cleanup). The fourth line is an immediate jump -1, which is an infinite loop, and it get executed upon subroutine return.

The next seven lines are for threads 1 through 7, which are instructed to twiddle their thumbs by looping infinitely. Note that the clear addresses are spaced 4 apart (both this distance and the base address are configurable at build time for the clear and interrupt vector groups). The interrupt instruction address space is blank because the interrupts won’t be enabled nor used for this program.

The subroutine code at address 0x40 reads the core version and then writes the core version to the I/O port, pops the data simultaneously with the write (stack cleanup), then issues a gto S3 and pops S3 (stack cleanup), which is the way subroutines are returned from in Hive.

Binary Search Division Subroutine Example
A somewhat meatier example is division. The binary search division algorithm is shown below:

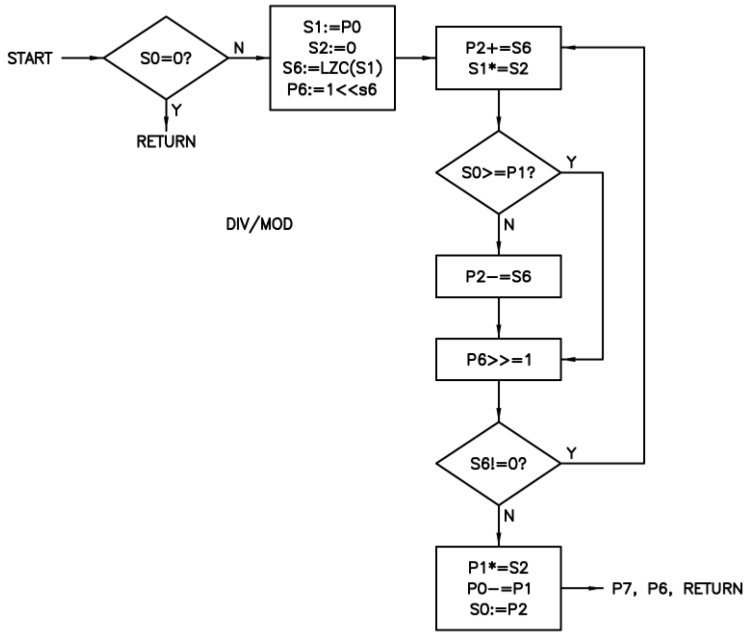


Figure 63. Binary search division algorithm flow chart.

This algorithm divides two unsigned inputs using a binary search. The idea is to light up one-hot bits going from MSB to LSB in a number OH, add this to Q, multiply by the denominator D, and compare this trial number to the numerator N. If the numerator is greater than or equal to the trial number then Q retains the one-hot bit.

Setting the one-hot (OH) start value to LZC(D) prevents internal overflow at D*(Q+OH) and speeds up the average case by reducing the number of loops.

The shifted one-hot value one used in the loop can also be used as the loop counter: when the one is completely shifted out (the vector == 0) exit the loop. This saves one step in the loop and one storage register.

```
 ADDR     OC  SA  SB      IM       OP  Comments
0x760 0xd010  s0   .       1  JMP_8NZ  -- (s0 != 0) ? PC++  // DIV/MOD SUB START
0x761 0x2df0  s0  P7       .      GTO  -- PC := P7  // return if denom zero
0x762 0x3081  s1  P0       .      CPY  -- s1 := P0  // s1=D, s0=N
0x763 0xb002  s2   .       0      BYT  -- s2 := 0  // s2=Q
0x764 0x3d16  s6  s1       .      LZC  -- s6 := LZC(s1)
0x765 0x8e6e  P6  s6       .      POW  -- P6 := 1 << s6  // s6=OH
0x766 0x806a  P2  s6       .      ADD  -- P2 += s6  // trial Q - LOOP START
0x767 0x8821  s1  s2       .      MUL  -- s1 *= s2  // s1=D*Q
0x768 0x1790  s0  P1       .  SKP_NLU  -- (s0 >= P1) ? PC++  // skip restore
0x769 0x846a  P2  s6       .      SUB  -- P2 -= s6  // s2=restored Q
0x76a 0x97fe  P6   .      -1   SHP_6U  -- P6 >>= 1  // new OH
0x76b 0xdfa6  s6   .      -6  JMP_8NZ  -- ( s6 != 0) ? PC += -6 // LOOP END
0x76c 0x8829  P1  s2       .      MUL  -- P1 *= s2  // s1=D*Q
0x76d 0x8498  P0  P1       .      SUB  -- P0 -= P1  // s0=N-D*Q=R
0x76e 0x30a0  s0  P2       .      CPY  -- s0 := P2  // s0=Q
0x76f 0x2dfe  P6  P7       .      GTO  -- PC := P7  // return, P6 - DIV/MOD SUB END
```

The subroutine code is above as it is found in the verification code, disassembled and displayed by the simulator. The return is not skipped if the denominator value is zero, which is mathematically undefined (+/- infinity). The denominator is pushed to S1, S2 is initialized to zero, the LZC of the denominator is converted to a one-hot value and is pushed to S6, and the loop begins.

The one-hot value is added to the quotient, which is then multiplied by the denominator and compared to the numerator. If greater, the one-hot value is subtracted from the quotient (restoring it). The one-hot value is shifted right once and the loop is performed again until the one-hot one is shifted out of the LSB position, leaving all zeros and the loop is exited.

Finally, the remainder is calculated and pushed onto S0, the quotient is pushed on top of it, and some cleanup is performed at subroutine return.

In terms of real time, assuming the denominator is not zero and the return is skipped, it takes 5 cycles to test the input and setup the loop, 5 cycles per loop best case and 6 cycles worst case, with 4 cycles after the loop. For 32 worst-case iterations with all being 6 worst case loops this gives:

5 + 6*32 + 4 = 201 cycles worst case

For a 200 MHz clock and 8 clocks per cycle, this is 8.04 us worst-case.

Binary Search SQRT Subroutine Example
A somewhat similar example to division is the calculation of the square root shown below. The binary search algorithm in a processor that has single cycle unsigned multiply is likely faster than other methods due to fewer necessary loop instructions:

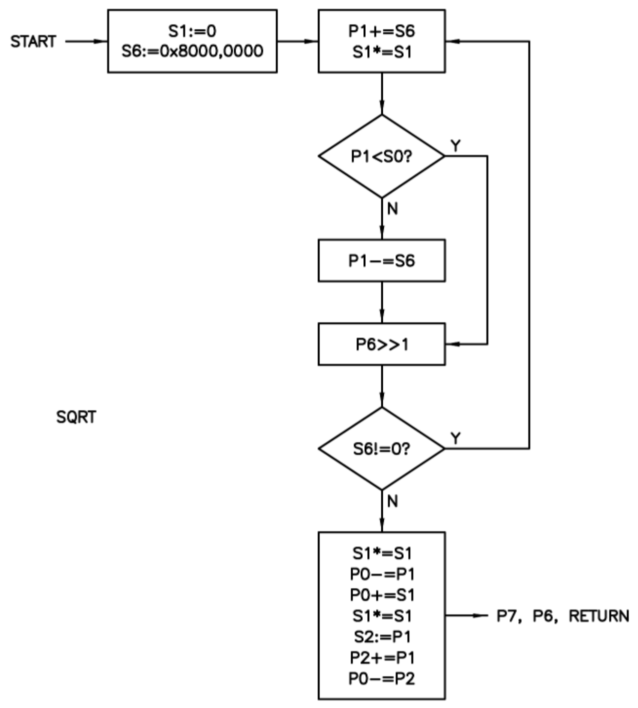


Figure 64. Binary search square root algorithm flow chart.

The idea is to light up the bits going from MSB to LSB in a trial number, square it, and compare it to the input. If the input x is an 8 bit unsigned integer (i.e. N=8) then the output needs at most 4 bits for the integer portion, and carrying out the looping 4 more times gives a result with 4 decimal places. So N bits in gives n.n bits out where n = N/2.

Note that picking the upper half of the square result will truncate the lower half, or the decimal portion. The negative influence of this can be largely ameliorated by the use of '<' rather than '<=' for comparison to the input, and this gives an underestimated result which is too small by at most 1 LSB. Computing the squared result and the square of the incremented result, and picking the one closest to the input corrects this at the end and obviously adds complexity, but provides a significant benefit by making the loop smaller.

Since the loop underestimates q, we know the q is either OK or needs at most 1 LSB of value added to it. The generic expression to evaluate this is:

abs{x - q^2} < abs{x - (q+LSB)^2} ?

If the result is true we pick q as the output, if false we pick q+LSB (= q++).

Since LSB > 0, then q < q+LSB, and q^2 < (q+LSB)^2. So we don't need to use absolute value functions to evaluate the expressions. Also, q^2 will always be in the range (x-2:x] and (q+LSB)^2 will always be in the range [x:x+2).

What we really want to know is the influence of adding LSB/2 to the input. We can square the result and compare it to the input to know whether to increment q.

(q+LSB/2)^2 >= x ?

Expanding the above gives:

q^2 + 2qLSB/2 + (LSB/2)^2 >= x ?
q^2 + qLSB + (LSB/2)^2 >= x ?

Since (LSB/2)^2 is very tiny we can safely discard it which gives:

q^2 + q*LSB >= x ?

The term q*LSB is a simple right shift of n; even more simply it is the uncorrected value of q as it sits in the processor register! This gives:

q^2 + q>>n >= x ?

Which can be rearranged as:

q>>n >= x - q^2 ?

If true we pick q as the result, otherwise we pick q+LSB (= q++).

We can actually eliminate the above conditional:

Q = q + x - integer[q^2] - int[q>>n + decimal[q^2]]

Note that the shifted one-hot one used in the loop can also be used as the loop counter: when the one is completely shifted out (the vector = 0) exit the loop. This saves one storage register, but more importantly eliminates one step in the loop.

```
 ADDR     OC  SA  SB      IM       OP  Comments
0x780 0xb001  s1   .       0      BYT  -- s1 := 0  // Q - SQRT SUB START
0x781 0x95f6  s6   .      31   SHP_6U  -- s6 := 1 << 31  // OH
0x782 0x8069  P1  s6       .      ADD  -- P1 += s6  // new Q - LOOP START
0x783 0x8b11  s1  s1       .    MUL_U  -- s1 *= s1  // square, int portion
0x784 0x1609  P1  s0       .   SKP_LU  -- (P1 < s0) ? PC++  // skip restore
0x785 0x8469  P1  s6       .      SUB  -- P1 -= s6  // restore
0x786 0x97fe  P6   .      -1   SHP_6U  -- P6 <<= -1  // new OH
0x787 0xdfa6  s6   .      -6  JMP_8NZ  -- (s6 != 0) ? PC += -6  // LOOP END
0x788 0x8b11  s1  s1       .    MUL_U  -- s1 *= s1  // square, int portion
0x789 0x8498  P0  P1       .      SUB  -- P0 -= P1  // x-=q^2
0x78a 0x8018  P0  s1       .      ADD  -- P0 += s1  // Q=q+x-q^2
0x78b 0x8811  s1  s1       .      MUL  -- s1 *= s1  // square, dec portion : (q>>n)^2
0x78c 0x3092  s2  P1       .      CPY  -- s2 := P1  // move
0x78d 0x839a  P2  P1       .    ADD_U  -- P2 += P1  // carry out, int
0x78e 0x84a8  P0  P2       .      SUB  -- P0 -= P2
0x78f 0x2dfe  P6  P7       .      GTO  -- PC := P7; P6  // return - SQRT SUB END
```

The subroutine code is above as it is found in the verification code, disassembled and displayed by the simulator. S2 is initialized to zero, the initial one-hot value and is pushed to S6, and the loop begins.

The one-hot value is added to q, which is then squared and compared to the input value. If greater than or equal, the one-hot value is subtracted from q (restoring it). The one-hot value is shifted right once and the loop is performed again until the one-hot one is shifted out of the LSB position, leaving all zeros and the loop is exited.

Finally, q is corrected and pushed to S0, and some cleanup is performed at subroutine return.

In terms of real time, it takes 2 cycles to setup the loop, 5 cycles per loop best case and 6 cycles worst-case, with 8 cycles after the loop. If all 32 iterations are 6 worst case loops this gives:

2 + 6*32 + 8 = 202 cycles worst case

For a 200 MHz clock and 8 clocks per cycle, this is 8.08 us worst-case.

Log$_2$ Subroutine Example
Presented here is the calculation of the 32-bit base 2 logarithm of an unsigned 32-bit input number. The algorithm shown exploits the fact that $\log_2(x^2) = 2*\log_2(x)$, and is implemented by a looped squaring processes.

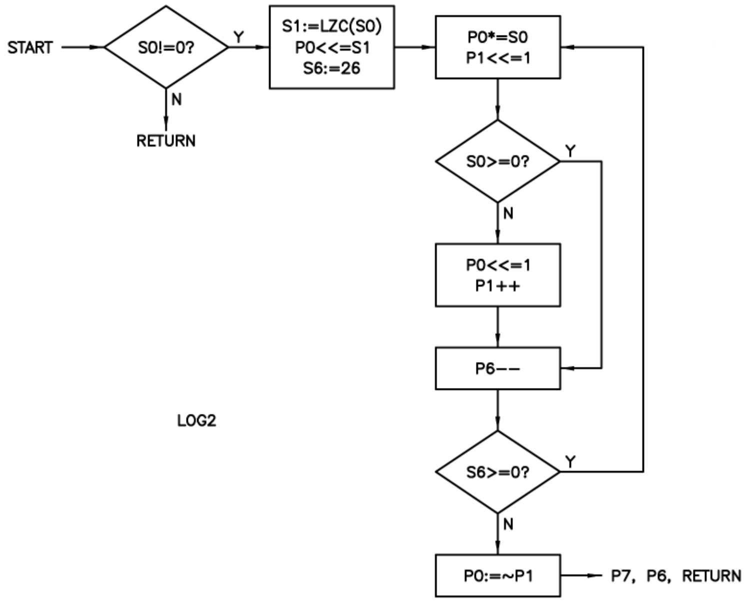


Figure 65. Log base 2 algorithm flow chart.

This algorithm is shown above as a flow chart. The initial test makes sure the input is non-zero because the log of zero is undefined. Next is the input normalization. Finally, we have the square / mantissa loop, with the loop exit test and result negation at the end.

The first section normalizes the input by shifting it to the left so that the MSB is equal to 1. This is accomplished efficiently with the LZC instruction followed by a shift. This number is also subtracted from 31 to form the log characteristic, which is the 5-bit number to the left of the decimal place in the result.

After normalization, the normalized input is squared and the resulting MSB examined. If it is equal to 1 then a 1 is left shifted into the characteristic. If it is equal to 0 then both the characteristic and the squared input are left shifted once (which should make the characteristic LSB = 0 and the squared input MSB = 1). This loop is executed 32 – 5 = 27 times to find all bits to the right of the decimal place in the result, AKA the log mantissa.

It is actually possible to skip the input normalization subtraction and use the same jump test within the loop for both operations that are conditionally jumped over, then simply negate the result at the end. This somewhat tricky but more efficient algorithm is the one implemented. (Whenever it feels like you are fighting the binary, there is likely a simpler, more elegant approach to be found by giving a bit more thought.)

```
 ADDR      OC  SA  SB      IM        OP  Comments
0x640 0xd010  s0   .       1  JMP_8NZ  -- (s0 != 0) ? PC++  // LOG2 SUB START
0x641 0x2df0  s0  P7       .      GTO  -- PC := P7  // return if input zero
0x642 0x3d01  s1  s0       .      LZC  -- s1 := LZC(s0)
0x643 0x8c18  P0  s1       .    SHL_S  -- P0 <<= s1  // normalize
0x644 0xb1a6  s6   .      26      BYT  -- s6 := 26  // loop index
0x645 0x8b08  P0  s0       .    MUL_U  -- P0 *= s0  // square - LOOP START
0x646 0x9019  P1   .       1   SHL_6S  -- P1 <<= 1
0x647 0xe020  s0   .       2  JMP_8LZ  -- (s0 < 0) ? PC += 2  // test MSB
0x648 0x9018  P0   .       1   SHL_6S  -- P0 <<= 1
0x649 0xa019  P1   .       1    ADD_8  -- P1++
0x64a 0xaffe  P6   .      -1    ADD_8  -- P6--
0x64b 0xff96  s6   .      -7 JMP_8NLZ  -- (s6 >= 0) ? PC -= 7  // LOOP END
0x64c 0x3498  P0  P1       .      NOT  -- P0 := ~P1
0x64d 0x2dfe  P6  P7       .      GTO  -- PC := P7, pop P6  // LOG2 SUB END
```

The subroutine code is above as it is found in the verification code, disassembled and displayed by the simulator. The return is not skipped if the input value is zero. Input normalization shifts the input value to the left until the MSB is 1, the number of shifts necessary to do this gives the inverse of the characteristic.

The square loop uses a shift and a conditional immediate add to left shift either a 0 or 1 into the mantissa LSB. Unsigned extended multiplication is the operation used for squaring. After the loop has completed, the result in S1 is negated and copied to S0 with both popped to form a move. The return address and loop index are both popped at return to complete the cleanup.

In terms of real time, assuming the input is not zero and the return is skipped, it takes 4 cycles to test the input, normalize it, and set up the loop, 5 cycles per loop best-case and 7 cycles worst-case, with 2 cycles after the loop. For 26 iterations with all being 7 worst-case loops this gives:

4 + 7*26 + 2 = 188 cycles worst case

For a 200 MHz clock and 8 clocks per cycle, this is 7.52 us worst-case.

Exp_2 Subroutine Example

Presented is the calculation of the 32-bit base 2 exponentiation of an unsigned 32-bit fixed decimal input.

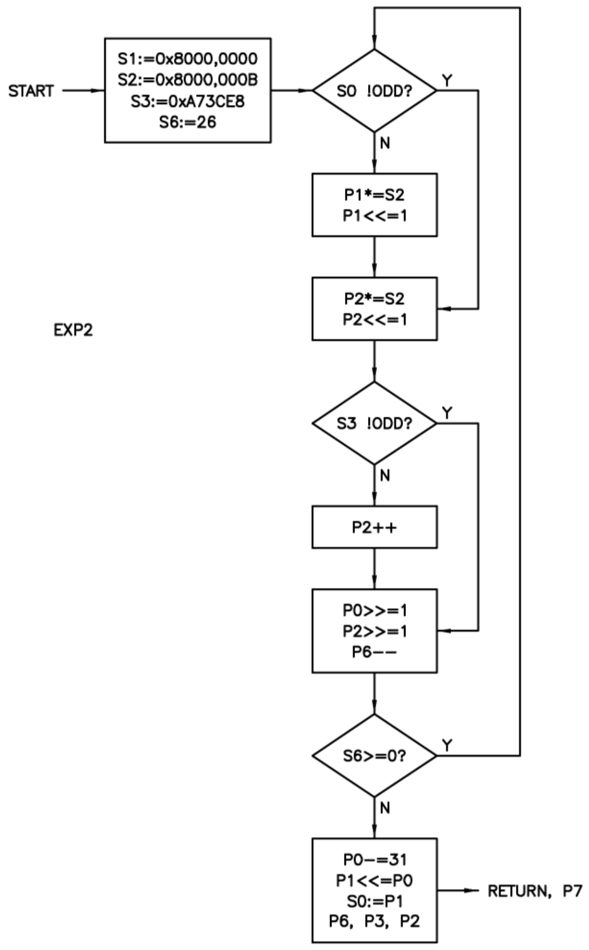


Figure 66. Exponential of 2 algorithm flow chart.

The idea is that exponentiation (2^n) where n is a real positive fixed point number is straightforward because binary numbers are themselves constructed of powers of 2.

For example:

5.5 = 101.1 = (1*2^2)+(0*2^1)+(1*2^0)+(1*2^-1).

So

2^5.5 = 2^[(1*2^2)+(0*2^1)+(1*2^0)+(1*2^-1)] = (2^(2^2))*(2^(2^0))*(2^(2^-1)) = 16*2*1.414... = 45.254...

So we need to find and selectively multiply together successive square roots of 2 (difficult), or start at some 2^(2^-m) root of 2 and square our way up (much easier but prone to error due to repeated squaring of the limited resolution base number).

For a 32-bit input, log_2 gives 5 bits of characteristic and 32 - 5 = 27 bits of mantissa. If we use this fixed decimal form to "undo" the log_2 with 2^n, then the input n to the 2^n function is:

[31:27] = 2^4, ..., 2^0
[26:0] = 2^-1, ..., 2^-27

Therefore, the first root we need is 2^(2^-m) = 2^(2^-27) = 1.00000000516... which is approximated by the 32 bit number 0x8000000B. Squaring this and picking the upper 32 bits requires a left shift of one to re-align the result. Shifting truncates the LSB, and squaring doubles the initial error, so the results tend to be underestimated and deviate more from the ideal with each iteration. We can add a small (0 or 1 LSB) "fudge" factor after each multiply & shift to make up for this, and dramatically increase the overall precision. The fudge factor bits were arrived at via trial and error in the spreadsheet version, with an eye towards minimizing both local and final error. Without the fudge factor, full scale output is off by as much as -0.56%. With the fudge factor, full-scale output is good to approximately 8 significant (base 10) digits.

After we hit the square root of 2 (i.e. 2^-1) the remaining operations are simple right shifts which can be performed as a single bulk right shift (thus likely tossing away many hard-won calculated digits).

```
 ADDR     OC  SA  SB      IM       OP  Comments
0x650 0x95f1  s1   .      31   SHP_6U  -- s1 := 0x8000,0000 (start value = 1)  // EXP2 SUB START
0x651 0x3012  s2  s1       .      CPY  -- s2 := 0x8000,000b  // start root = 2^2^-27
0x652 0xa0ba  P2   .      11    ADD_8
0x653  0xc03  s3  s0       .      LIT  -- s3 := 0xa73ce8  // fudge factor
0x654 0x3ce8   .   .  15592         L
0x655   0xa7   .   .    167         L
0x656 0xb1a6  s6   .      26      BYT  -- s6 := 26  // loop idx
0x657 0x1d00  s0  s0       .   SK2_NO  -- (s0 != odd) ? PC += 2  // skip LOOP START
0x658 0x8b29  P1  s2       .    MUL_U  -- P1 *= s2 (uns)
0x659 0x9019  P1   .       1   SHL_6S  -- P1 <<= 1  // so msb=1
0x65a 0x8b2a  P2  s2       .    MUL_U  -- P2 *= s2  // square to get next root
0x65b 0x901a  P2   .       1   SHL_6S  -- P2 <<= 1  // so msb=1 & lsb=0
0x65c 0x1533  s3  s3       .   SKP_NO  -- (s3 != odd) ? PC++  // skip if fudge bit=0
0x65d 0xa01a  P2   .       1    ADD_8  -- P2++  // set LSB of running root
0x65e 0x97f8  P0   .      -1   SHP_6U  -- P0 >>= 1  // get next intput bit
0x65f 0x97fb  P3   .      -1   SHP_6U  -- P3 >>= 1  // get next fudge bit
0x660 0xaffe  P6   .      -1    ADD_8  -- P6--  // dec loop idx
0x661 0xff56  s6   .     -11 JMP_8NLZ  -- (s6 >= 0) ? PC += -11  // LOOP END
0x662 0xae18  P0   .     -31    ADD_8  -- P0 += -31
0x663 0x8d89  P1  P0       .    SHL_U  -- P1 <<= P0
0x664 0x3090  s0  P1       .      CPY  -- s0 := P1
0x665  0x14c   .   .      76      POP  -- P6, P3, P2
0x666 0x2df0  s0  P7       .      GTO  -- return to P7  // EXP2 SUB END
```

The subroutine code is above as it is found in the verification code, disassembled and displayed by the simulator. The running multiply is initialized to '1' and pushed to S1. The initial running root of 2 is pushed to S2. The fudge factor is pushed to S3. The loop index is pushed to S6.

There are two conditional jumps in the loop. The first tests the input LSB and skips the running multiplication of the current root of two and shift if the bit is zero. After this, the next root is found via squaring and shifting, and the second test applies the fudge bit. Then the input and fudge vectors are shifted right once to expose the next bits to testing, the loop index is decremented, and the loop repeats until the loop index goes negative.

When the loop is exited, -31 is added to the remaining input bits, and the running multiply is shifted left by this amount. The result is pushed to S0, some cleanup occurs, and the subroutine returns.

In terms of real time, it takes 5 cycles to set up the loop, 8 cycles per loop best-case and 11 cycles worst-case, with 5 cycles after the loop. For 26 iterations with all being 11 worst-case loops this gives:

5 + 11*26 + 5 = 296 cycles worst-case

For a 200 MHz clock and 8 clocks per cycle, this is 11.84 us worst-case.

Inverse Subroutine Example
Presented is the calculation of the inverse of an unsigned 32-bit integer input via Newton's Method.

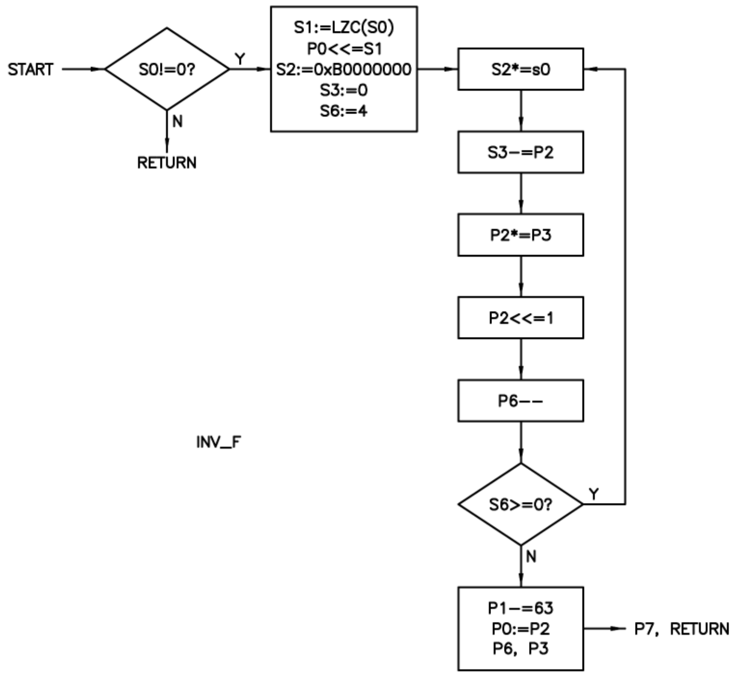


Figure 67. Inverse algorithm flow chart.

Newton's method is iterative:

x(n+1) = x - f(x) / f'(x)

For input D the reciprocal function is:

x = 1/D or D = x^-1

to find the root of this:

f(x) = D - x^-1
f'(x) = x^-2
x(n+1) = f(x)/f'(x)

So the iteration is:

x(n+1) = x * (2 - D * x)

In general the reciprocal is kind of a problem. Integer division doesn't give useful output when the numerator is 1, though we could scale the numerator up. We know that precision will be lost if the input is larger than ~1/2 of the total bits wide, so it would be nice to scale things in order to maximize the precision of the result. To do this we first scale the input so as to have a one in the most significant bit position, which gives a normalized number in the range [0.5:1). Rearranging Newton's method a bit:

x(n+1) = 2 * x * (1 - D * x/2)

Dividing both sides by 2:

x(n+1)/2 = 2 * x/2 * (1 - D * x/2)

The equation is now in terms of x/2 rather than x, which allows intermediate values to fit in a fixed register width.

If D is normalized to the interval [0.5:1) then the reciprocal x is in the interval [2:1) and x/2 is in the interval [1:0.5). Since D * x/2 is around 1/2, one minus this is around 1/2, so the result of the difference also conveniently occupies the same width register space. But the inclusion of 1 in the range of x/2 is potentially problematic as it can cause overflow. It appears that the choice of initial guess and limiting the iterations to only those necessary can prevent this. Overflow is most likely with powers of 2 input where D is smallest, which is easy to test for.

Input normalization also allows the use of a constant initial guess. Using 0.6875 (0x.B or 0b.1011) doesn't seem to require more than 5 iterations with 32 bit operations. The result appears to be off by no more than +2/-2 counts from the ideal rounded result.

The exponent returned is the value required to denormalize the number to a non-float, which is:

EXP = 1 + LZC(D) - 2 * (I/O bit width)

The offset of positive one is required because we are returning x/2 rather than x.

If the result will be multiplied by another number, normalize the number, do the multiplication, then denormalize the result to maintain resolution.

```
 ADDR     OC  SA  SB     IM       OP  Comments
0x7a0 0xd010  s0   .      1  JMP_8NZ  -- (s0 != 0) ? PC++  // skip return - INV_F SUB START
0x7a1 0x2df0  s0  P7      .      GTO  -- PC := P7  // return
0x7a2 0x3d01  s1  s0      .      LZC  -- s1 = LZC(s0)  // normalize input
0x7a3 0x8d18  P0  s1      .    SHL_U  -- P0 <<= s1
0x7a4 0xbb02  s2   .    -80      BYT  -- s2 = 0xb0  // initial guess
0x7a5 0x918a  P2   .     24   SHL_6S  -- P2 <<= 24
0x7a6 0xb003  s3   .      0      BYT  -- s3 = 0
0x7a7 0xb046  s6   .      4      BYT  -- s6 = 4 (loop index)
0x7a8 0x8b02  s2  s0      .    MUL_U  -- s2 *= s0  // LOOP START
0x7a9 0x84a3  s3  P2      .      SUB  -- s3 -= P2
0x7aa 0x8bba  P2  P3      .    MUL_U  -- P2 *= P3
0x7ab 0x901a  P2   .      1   SHL_6S  -- P2 <<= 1
0x7ac 0xaffe  P6   .     -1    ADD_8  -- P6 += -1
0x7ad 0xffa6  s6   .     -6 JMP_8NLZ  -- (s6 >= 0) ? PC += -6  // LOOP END
0x7ae 0xac19  P1   .    -63    ADD_8  -- P1 += -63  // exponent is 1 + lzc - (2 * bit_width)
0x7af 0x30a8  P0  P2      .      CPY  -- P0 = P2
0x7b0  0x148   .   .     72      POP  -- P6, P3
0x7b1 0x2df0  s0  P7      .      GTO  -- PC := P7  // return - INV_F SUB END
```

The subroutine code is above as it is found in the verification code, disassembled and displayed by the simulator. The initial test makes sure the input is non-zero because the inverse of zero is undefined. Next is the input normalization, and the loading of the initial guess and loop index. Next is the iterative implementation of Newton's method as described above. Finally we have the exponent calculation and some cleanup at the end.

The first section normalizes the input by shifting it to the left so that the MSB is equal to 1. This is accomplished efficiently with the LZC instruction followed by a shift. This number is also added to -63 to form the denormalization exponent.

In terms of real time, it takes 7 cycles to set up the loop, 6 cycles per loop, with 4 cycles after the loop. For 5 iterations this gives:

7 + 6*5 + 4 = 41 cycles

For a 200 MHz clock and 8 clocks per cycle, this is 1.64 us - significantly faster than the binary search division method.

“Bouncing Ball” LED PWM Display Example

An interesting digital concept that seems rather unlikely to work in actual practice is the cross-coupled integrator sine / cosine oscillator. The output of an accumulator is fed to the input of a second accumulator; the output of this second accumulator is arithmetically negated and fed back to the input of the first. By identically scaling the accumulator inputs by a number generally much less than one (call it “alpha”) the frequency of the oscillations may be controlled or set. The first accumulator generates cosine, the second sine. It may be best seen as a ringing state variable band pass filter with infinite Q. Sine and cosine amplitude is set by placing an initial value in one accumulator and zeroing the other one out, and then letting it “ring” for infinity. Sine and cosine frequency ~= cycle frequency * alpha / (2 * Pi).

Mathematically this construct works because the integral of cosine = sine, and the integral of sine = -cosine. Numerically this construct will only work if there is a single register delay in the loop, and provided this requirement is met then truncation errors due to integrator input scaling rather mysteriously do not build up or otherwise become a long-term problem.

If we feed a sine wave to an absolute value circuit (invert the entire value if the sign bit is negative) and find the power of two of this value, we can make a one-hot “bouncing ball” type LED display. To add smoothness we can interpret the absolute sine wave value as a fixed decimal, using the integer portion to nominally select the LED, and using the decimal portion to perform PWM (pulse width modulation). First order PWM is most easily accomplished by accumulating the PWM value and looking for accumulator overflow (smaller inputs cause infrequent overflows, and larger inputs cause frequent overflows). When there is overflow, we select the next higher LED. Rather than a flow diagram, a block diagram of this is shown below:

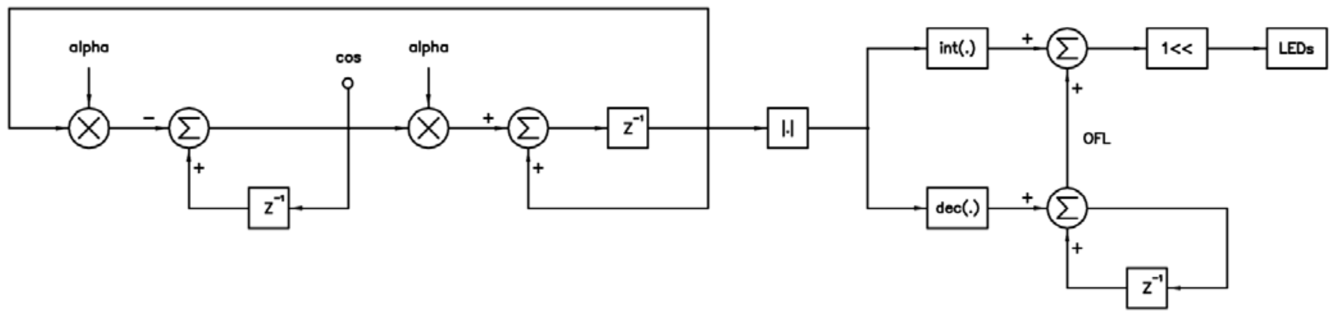


Figure 68. Bouncing ball block diagram.

To complete the design we need to establish the frequency and amplitude of the sine wave, as well as fix the decimal place. The core clock is 160 MHz and dividing this by 8 clocks per cycle gives 20 MHz. Our loop takes 14 cycles to execute once, so to get a roughly 1 Hz bounce rate we need an alpha of 14 * 2 * pi / 20 MHz ~= 0.0000044 which is roughly 0x00003000. The Cyclone demo board I am using has four LEDs, which correspond to the values zero, one, two, and three, so if we set the most significant hex digit to be our integer portion, then the initialization value should be 0x30000000.

```
ADDR     OC  SA  SB     IM       OP  Comments
0x40 0xb000  s0   .      0      BYT  -- s0 := 0  // sin init - THREAD 0 BEGIN
0x41  0xc01  s1  s0      .      LIT  -- s1 := 0x3000,0000  // cos init
0x42      0   .   .      0        L
0x43 0x3000   .   .  12288        L
0x44  0xf02  s2  s0      .    LIT_U  -- s2 := 0x3000  // alpha
0x45 0x3000   .   .  12288        L
0x46 0xb005  s5   .      0      BYT  -- s5 := 0  // pwm init
0x47 0x8a20  s0  s2      .    MUL_S  -- s0 *= s2  // sin * alpha  - LOOP START
0x48 0x8489  P1  P0      .      SUB  -- P1 -= P0  // cos -= sin * alpha
0x49 0x8a21  s1  s2      .    MUL_S  -- s1 *= s2  // cos * alpha
0x4a 0x8098  P0  P1      .      ADD  -- P0 += P1  // sin += cos * alpha
0x4b 0x3003  s3  s0      .      CPY  -- s3 := s0
0x4c 0xf013  s3   .      1 JMP_8NLZ  -- (s3 !< 0) ? PC++
0x4d 0x343b  P3  s3      .      NOT  -- ~P3
0x4e 0x9043  s3   .      4   SHL_6S  -- s3 <<= 4
0x4f 0x803d  P5  s3      .      ADD  -- P5 += s3  // update pwm count
0x50 0x83b5  s5  P3      .    ADD_U  -- s5 += P3  // get pwm ofl
0x51 0x924b  P3   .    -28   SHL_6S  -- P3 <<= -28
0x52 0x80db  P3  P5      .      ADD  -- P3 += P5  // add pwm ofl
0x53 0x8e3b  P3  s3      .      POW  -- P3 := 1 << s3  // one-hot
0x54 0x9c5b  P3   .      5    REG_W  -- reg[5] := P3  // led reg
0x55  0x4f1   .   .    -15    JMP_8  -- PC -= 15  // loop forever - LOOP END
```

The subroutine code is above, disassembled and displayed by the simulator. The initial values are loaded and the loop is entered. Cosine is updated first, then sine. The absolute value of sine is found and pushed to stack three. This value is left shifted 4 places to obtain the decimal portion, which is added to the PWM counter to update it, after this it is added again but only to check for overflow. The absolute sine value is right shifted 28 places to obtain the integer portion, the overflow is added to it, the result of this converted to a power of 2, which is output to the LEDs.

There is a bit of cheating going on here. We really should check for PWM overflow *before* we update its value, but for slowly changing inputs the order isn't too important. Note also that the total loop time varies by one cycle depending on the sign of the sine value (due to the negation jump) but this isn't noticeable even if you know about it.

APPENDIX D : ASSEMBLY NOT REQUIRED?

While writing the opcodes as a sort of pseudo code in my notes and code comments, I became enamored with the idea of creating a higher level language for Hive. But now that I've written and used the C++ interactive simulator I'm not so sure that is entirely desirable.

A few people have contacted me since publishing earlier versions of this paper and the core SV code asking if an assembler or compiler were in the works, expressing their need for such tools before they could seriously consider using Hive in their projects. I do understand these sentiments, though I wonder how much of the perceived need is driven by the (IMO needless) complexity of even the most modern of RISC processors at the lowest level. The minute you have things like shared internal busses and caches the user can't really rely on strict timing or even the ordering of events handled and generated by the processor. Lots of registers, flags, modes, etc. all interacting geometrically must cause most people to throw up their hands even at the assembly level. It is likely the norm at this point that engineers who require a certain level of processing power opt for a full blown high level language on top of a full blown operating system because they can't realistically deal with the full blown processor sitting on the board.

It's likely I'm too wet behind the ears at this point to speak authoritatively, but it seems the more I interact with Hive the less I feel the need for even a low level assembly language between me and it. This reaction has surprised me, as I was almost certain there was another level of things to implement on top of the interactive simulator (processors need more support than kids). At this point anyway I don't want dumb software managing where I put constants, variables, and subroutines in memory. I really like seeing where the code is physically located and where there are blank places in the linear address space. If I want legible pseudo code I might write an auto-comment mechanism to generate it from opcode disassembly, rather than the other way around. Micro management of the memory is somewhat painful I suppose, but for the kinds of things I want to use Hive for it seems appropriate, and there doesn't seem to be any easy substitute (outside of a much more powerful processor with gobs of memory to support the layers upon layers of unknown fluff other coders tend to bring to things).

APPENDIX E : ENDIANNESS

Introduction

Endianness has to do with how data bytes, and sometimes the bits within those bytes, are arranged in memory.

LE (little endian) aligns all of the natural binary indices at zero (the little end) and employs one of these indices for the address, but LE is not restricted to the byte index for the address – any larger index (16-bit, 32-bit, etc.), or smaller index all the way down to serial (1-bit) will work equally well. Indeed, variable aspect ratio memories such as those found in FPGAs are LE for this reason.

BE (big endian) elevates the importance of the byte above all other stored data widths. It does this by forcing byte addressing on the architecture, and then *reversing the byte order within multi-byte values stored in memory*. The MSB (most significant byte) – the big end – is therefore stored at the smallest value byte address location for the wider type, and the LSB (least significant byte) is stored at the largest value byte address location for the wider type, with any intermediate bytes following the gradient. Often the bits within the wider stored type are labeled counter to their natural binary index, and sometimes the bits with the bytes themselves are actually reversed. Thus the address is necessarily byte-based but actually runs counter to the byte and higher indices within a wider stored type.

BE can lead to grammatical confusion because it renders common phraseology like “the upper byte” ambiguous. It means what we think it means when referring to the PC or to multi-byte registers inside the processor, but it means the opposite when referring to a multi-byte value in memory, where the byte with the highest address and the byte with the most numeric significance are (at least) two different things.

There is much tut-tutting that endianness is academic, that both little and big have their merits, just pick one but don’t become an advocate or you’ll start a holy war. Along with this we get endian diagrams glossing over the details and thereby portraying the choice of endian as much more arbitrary than it actually is, and a too clever by half tie-in to Swiftian satire pushing the BE agenda. I imagine this notion of “distinction without a difference” is held and promulgated mainly by software types who only have to deal with endianness at the highest levels, where the fits it gives them are mistakenly chalked up to the mixing of endianness, rather than to the continued existence of rival and conflicting systems of numerical representation in memory (BE) that should have been weeded out back when other bad ideas such as one’s complement were culled from the herd.

Simplistic Diagrams and Byte Access

The simplistic diagram below portrays a processor connected to byte addressable memory. The processor almost certainly has a PC (program counter) that is more than 8 bits wide, and its ALU can most likely generate 16-bit or wider internal arithmetic results – how should these wider values be stored in byte addressable memory? The diagram shows 32-bit LE memory storage on the left and BE storage on the right. When drawn this way, in so called “Chinese” vertical fashion with byte addressable memory, the differences appear superficial.

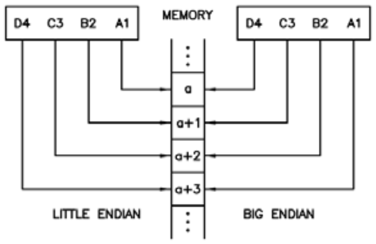


Figure 69. Common little / big endian diagram.

Even here there is an obvious best choice: For calculation purposes (e.g. carry propagation) the ALU is generally is more “interested” in obtaining the little end of a wider stored value first, so the little end should be encountered first in the natural progression of things (i.e. as the address increases). This way, should it have the opportunity, the ALU can be working on the calculation in parallel with the retrieval of the big end of the stored value.

Background : Natural Indices

We name the digit positions of the decimal system ones, tens, hundreds, etc. We do so because they are weightings, or multipliers for the digits in their positions. Weightings are found by raising the base, ten in this case, to the power of the distance from the ones position. For example:

$163_{10} = 1(10^{\mathbf{2}}) + 6(10^{\mathbf{1}}) +3(10^{\mathbf{0}})$.

This conveniently works for the fractional side of things if we consider distances in the opposite direction from the ones position as negative, e.g.:

$3.14_{10} = 3(10^{\mathbf{0}}) + 1(10^{\mathbf{-1}}) +4(10^{\mathbf{-2}})$.

This scheme works for any number base, e.g. binary:

$10100011_2 = 1(2^{\mathbf{7}}) + 0(2^{\mathbf{6}}) + 1(2^{\mathbf{5}}) + 0(2^{\mathbf{4}}) + 0(2^{\mathbf{3}}) + 0(2^{\mathbf{2}}) + 1(2^{\mathbf{1}}) + 1(2^{\mathbf{0}}) = 128_{10} + 32_{10} + 2_{10} + 1_{10} = 163_{10}$.

Indeed, this is how we convert numbers from one base to another. So the distance from the ones position forms a natural index into the digits of any base, and this index is mathematically meaningful so therefore quite useful beyond mere indexing.

As is the custom for Arabic numbering, we list the positions, and therefore the positional weightings ($\text{Base}^{\text{Position}}$), as increasing from right to left. Regardless of whatever direction or order the majority of us have decided is the “correct” way to read, write, vocalize, etc. numbers, these are entirely arbitrary conventions that have nothing to do with endianness. This is because integers are, in a very basic sense, (and I’m not just talking about byte order here) LE. They are anchored by their little ends and increase in the direction of their big ends, and we align their little ends before we perform arithmetic on them. I’m using integers here because I believe that is what processors operate on most, even if the higher level representation may be fixed point, float, character, etc.

31	30	29	28	27	26	25	24	23	22	21	10	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	**Binary Index**
7				6				5				4				3				2				1				0				**Hex Index**
3								2								1								0								**Byte Index**
1																0																**16-bit Index**

3	2	1	0	3	2	1	0	3	2	1	0	3	2	1	0	3	2	1	0	3	2	1	0	3	2	1	0	3	2	1	0	**Binary / Hex**
1				0				1				0				1				0				1				0				**Hex / Byte**
1								0								1								0								**Byte /16-Bit**
1																0																**16-bit / 32-bit**

Figure 70. Natural 2^n Indices: Absolute (top), Relative (bottom).

As can be seen above, the natural indices of powers of two representations are particularly interesting as they form a nested hierarchy. For example, a memory can be seen as a linear sequence of bits, which can be divided up into a sequence of nibbles, nibbles into bytes, byes into 16-bit values, and so on. Aligning their little ends allows us to employ any absolute index as our memory address. This is LE and there really isn’t too much more to say about it. It’s as simple as anything ever gets and that means something.

Internal vs. External Storage

Since everything in the processor world is binary, let’s take a closer look at the natural 2^n indices of an integer stored both internally in a general purpose register and externally in main memory.

31	30	29	28	27	26	25	24	23	22	21	10	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	**Binary Index**
7				6				5				4				3				2				1				0				**Hex Index**
3								2								1								0								**Byte Index**
1																0																**16-bit Index**

1	1	0	1	0	1	0	0	1	1	0	0	0	1	1	1	1	0	1	1	0	0	1	0	1	0	1	0	0	0	0	1	**Binary**
0xD				0x4				0xC				0x3				0xB				0x2				0xA				0x1				**Hex**
0xD4								0xC3								0XB2								0xA1								**Byte**
0xD4C3																0xB2A1																**16-bit**
0xD4C3B2A1																																**32-bit**

Figure 71. LE / BE register storage of the value 0xD4C3B2A1.

The above shows how any 32-bit processor, LE or BE, would store value 0xD4C3B2A1 in one of its internal general purpose registers. Absolute positional 2^n indexing is shown at the top, and values in various 2^n representations of the value itself at the bottom. Note that, because position indices are mathematically tied to weighting, all values and indices, regardless of width, grow in the same direction. How are 32-bit values stored in the memories of LE and BE machines?

31	30	29	28	27	26	25	24	23	22	21	10	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	**Binary Addr**
7				6				5				4				3				2				1				0				**Hex Addr**
3								2								1								0								**Byte Addr**
1																0																**16-bit Addr**

31	30	29	28	27	26	25	24	23	22	21	10	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	**Binary Index**
7				6				5				4				3				2				1				0				**Hex Index**
3								2								1								0								**Byte Index**
1																0																**16-bit Index**

1	1	0	1	0	1	0	0	1	1	0	0	0	1	1	1	1	0	1	1	0	0	1	0	1	0	1	0	0	0	0	1	**Binary**
0xD				0x4				0xC				0x3				0xB				0x2				0xA				0x1				**Hex**
0xD4								0xC3								0xB2								0xA1								**Byte**
0xD4C3																0xB2A1																**16-bit**
0xD4C3B2A1																																**32-bit**

Figure 72. LE memory storage of the value 0xD4C3B2A1 at address 0.

Above we see the way a LE processor stores the same value 0xD4C3B2A1 at address 0. The only thing different from the internal register storage representation here is the addition of an addresses table at the top. Note that the address is identical to a natural index. We can pick any 2^n natural index we like for the address and the memory contents will look exactly the same as the internal register contents.

3								2								1								0								**Byte Addr**

31	30	29	28	27	26	25	24	23	22	21	10	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	**Binary Index**
7				6				5				4				3				2				1				0				**Hex Index**
0								**1**								**2**								**3**								**Byte Index**
0																**1**																**16-bit Index**

1	0	1	0	0	0	0	1	1	0	1	1	0	0	1	0	1	1	0	0	0	0	1	1	1	1	0	1	0	1	0	0	**Binary**
0xA				0x1				0xB				0x2				0xC				0x3				0xD				0x4				**Hex**
0xA1								0xB2								0XC3								0xD4								**Byte**
0xA1B2																0xC3D4																**16-bit**
0xA1B2C3D4																																**32-bit**

Figure 73. BE memory storage of the value 0xD4C3B2A1 at byte address 0.

Above we see the way a BE processor stores the same value 0xD4C3B2A1 at byte address 0. Here we have removed the other possible types of 2^n addressing and left byte addressing, as byte addressing dominates in any BE discussion. Note that sub-byte absolute (and relative as well) indices increase in the direction of the address, but the byte and 16-bit indices run counter to the address. The byte positions within the 32-bit value have been jumbled, which jumbles the higher 16-bit and 32-bit representations

BE proponents may ask us at this point to literally turn the tables, claiming that the above is somehow unfair or unrepresentative because memory dumps are byte oriented affairs listed from left to right. This is quite a reach, and allows them to artificially reframe the terms of the debate by cosmetically reversing the very real and confusing byte reversal that BE imposes on the lowest level. Of them I would ask: why not order the memory presentation in order to maximize dump readability, rather than order of the actual memory itself? This is how it can be done with LE: provide two separate listings, one right-to-left (with increasing address) for hex numbers, and another left-to-right for ASCII. There is usually a separate listing for ASCII anyway as hex and text are rendered differently.

As a further twist, the bits within BE bytes may also be reversed. Which shouldn't be confused with something else BE proponents tend to do, and that is: re-labeling the bit index to start with 0 as signifying the MSb (most significant bit). It's pretty wild encountering this physically on, say, the PowerPC peripheral bus pins during a schematic review, where everything you naturally assume is suddenly wrong. There must be mountains of failed prototypes resting quietly in landfills due to wacky bus bit enumeration alone.

Pros and Cons

It should be clear by now that LE is the natural order of things, which is the biggest pro you can give anything, and has no substantive cons that I'm aware of. Let's go through the supposed BE pros that I've encountered:

1. The big end of a number comes first when reading and writing them in most languages.
2. The sign of a big endian number is always in the LSB position so it is trivial to locate and test. Also, the BE byte arrangement enables more efficient binary to decimal conversion algorithms.
3. BE is needed in our system for backward compatibility.
4. BE is just as consistent as LE so it's a wash.

To which I reply:

1. Integers are little endian, and readability direction is a poor reason to jumble byte storage in memory.
2. These arguments seem to be advocating software end runs around the hardware load / store mechanism. Retrieving only a single byte of a multi-byte data type in memory strikes me as bad form, and particularly so if it requires the software to be endian aware. Memory operations of modern processors are almost always wider than one byte, so there is little or no efficiency to be gained by rearranging bytes in memory for testing or conversion purposes.
3. It's been my experience that backward compatibility should be resisted with every fiber of one's being.
4. No, there is simply no configuration of BE that is as simple and uniform as LE endian when it come to the overriding issues of addressing and natural indices.

Endianness In Serial Protocols

Processors usually employ dedicated serial communication hardware in order to serialize / deserialize parallel data, provide parallel data demarcation, and in general unburden the processor. So the endianness of serial communication seems fairly moot when it comes to picking LE or BE for the *processor* architecture.

Though the most straightforward protocols are those that impose the least "need to know" burden on the protocol ends. Serial LE wins here because when sending / receiving the first bit of a message, how much more "first" can it be than the least significant bit of the first nibble of the first byte of the first 16-bit word etc.? With BE all bets are off, though you will definitely receive the MSb of something, somehow, first. Perhaps a minor point, but LE bit streams are much easier to perform carry propagation serial math on at the hardware level as well.

APPENDIX F : SUNDRIES

Bzzz!

I would be remiss if I did not point out the less positive aspects of Hive that I am aware of:

- In its present state, Hive cannot execute instructions directly from external memory, and I think this is probably its biggest weakness. The whole point of a processor is to leverage fast computation over large cheap storage, and FPGA RAM is neither large nor cheap. (That said, why do modern hard processors not have integrated DRAM or similar in place of their huge on-chip caches? Hans Moravec notes that the ratio of memory size to processor speed has remained a fairly constant one megabyte per MIPS throughout the history of general computing, which is a strong argument for fixed size, tightly integrated, on-chip RAM.)
- The instruction set of Hive was hatched more intuitively than scientifically by a person who does not exactly have loads of practical experience programming assembly. I have put more time into selecting operations, sizing immediate fields, and shuffling things around in the opcode encoding space than I have spent actually programming Hive.
- Common data & instruction memory space (Von Neumann architecture) enables many good things, but it generally hinders code from executing directly from ROM, and it also makes it that much likelier for wild data writes to clobber code. A single thread caught out in the weeds means you should probably clear all threads and refresh the memory space. (I should point out that the "Von Neumann bottleneck" isn't an issue for Hive because it uses dual port BRAM for main memory.)
- With any stack machine, stack fullness is something the programmer must track in order to avoid stack faults, and Hive has more stacks than usual to keep track of (though to be fair they are used in a simpler manner).
- Strict equal bandwidth multi-threading forces the programmer to implement some kind of load sharing arrangement for algorithms that require more real-time / less latency than a single thread can provide.
- Real-time response to an interrupt can be somewhat long and variable (though, depending on the application, this could perhaps be compensated for with additional interrupt time stamp & register set logic).
- FPGA logic will likely never be as fast, power efficient, inexpensive, etc. as an ASIC, so *any* soft processor core is in some sense a solution in search of a problem.

Notes

- As a test, I edited the v6.09 core to have a 16-bit wide ALU. This resulted in 4 BRAMs for the 8 stacks / thread, 1747 LEs, and a top speed of 215MHz. Stated relatively, this resulted in 50% stack BRAM count, ~70% LE count, and 107% top speed when compared to the full 32-bit core. These figures don't strike me as significant enough to warrant further pursuit of a 16-bit variant of Hive.
- Hive was developed (including simulation / verification) with Altera Quartus II 9.1sp2 Web Edition (unfortunately the last edition with integrated simulator) running on WinXP Pro (sadly past the end of support).
- TextCrawler was used extensively to perform multi-file text search and replace (freeware from Digital Volcano).
- Pictures were drawn in AutoCAD 2006 (I've found it ironically difficult to export good-looking image files from AutoCAD) plotted to Adobe Generic PS printer (free from Adobe, good luck finding a suitable *.inf file) and rasterized with Paint Shop Pro X (pretty much broken in Win7/64).
- The Hive document was written in MS Word 2003 (also past end of support), and converted to PDF with Adobe Acrobat 8 Professional (free-ish due to license server retirement).
- There are inexpensive FPGA demo boards readily available on eBay. A very nice Cyclone 4 board can be had for $27 USD or thereabouts. Comparable Xilinx Spartan offerings will require boot code initialization changes, and will likely be more expensive and run slower (Altera apparently uses faster transistors).
- If one has a UART attached to the FPGA inputs (there are quite inexpensive USB to TTL level cables out there that can simultaneously power the FPGA demo board), the boot code may consist of a simple boot loader capable of uploading, storing, and loading executable code. The boot code itself would not need to be touched much after that.
- Simple scripting could be used to convert SystemVerilog boot text (or similar assembly) into uploadable binary data. A ~$1 SPI FLASH or EEPROM device tacked onto a few spare FPGA pins could hold gobs of uploaded programming goodness.

About Me

I am primarily a digital designer with 15 years of experience targeting programmable logic devices (i.e. FPGAs and CPLDs made by Altera, Xilinx, and Lattice) with side experience in the mixed signal and analog fields. I have designed many types of digital structures including general purpose processors, DPLLs, TSIs, UARTs, I2C & SPI masters / slaves / arbiters, UTOPIA interfaces, generic FIFOs as well as Ethernet packet and ATM cell FIFOs. I strive for high levels of portability, readability, and craftsmanship in my code, and very much enjoy the documentation phase.

For ten years I was a Member of Technical Staff for a large telecom. Prior to that I held the positions of Assistant Mechanical Engineer, CNC Programmer, CNC Machinist, QA Inspector, and Vacuum Former Machine Operator.

I received the degrees of BSEE and MSEE from The University of Virginia in 1996 and 1998, respectively. I've published one journal paper, and hold one patent.

I list the above merely to give the curious some sense of my background. Literally anyone can do digital design and I highly encourage all to do so – I find it to be an incredibly creative, as well as an incredibly useful, activity.

I've spent the last couple of years researching, experimenting, and prototyping in the digital / analog Theremin field – one reason for which Hive was developed. My hope is to ultimately produce one or more novel electronic musical instruments with an emphasis on playability and other ergonomic factors.

Comments?

Found a bug in Hive (ha ha)? If you have questions, comments, criticisms, improvements, etc. regarding Hive I'd love to hear them! Contact me at: tammie_eric@verizon.net (note the '_' underscore).

RASH^{TM}

***R**egister **A**nd **S**tack **H**ybrid*

DOCUMENT CHANGE CONTROL

(YYYY-MM-DD)	Hive version	Notes
2015-09-03	08.06	Finished sim appendix, added sim tour, INV_F subroutine, many small edits. Switch from SV initial type boot code to MIF based boot code as generated by the C++ simulator.
2015-04-14	07.01	Added appendix on simulator. Expanded register set and vector support sections. New endianness screed.
2015-02-24	06.03	Revamped vector control and thread initialization. Register set additions / changes due to above. WAR & RAW sense reversed for clarity.
2014-12-25	06.02	New opcode op_cls_8, some style-driven opcode renaming (particularly the immediate field), other minor edits.
2014-07-15	06.01	Updates to reflect 32 bit memory access, new / changed opcodes, and transition to SystemVerilog code base.
2014-06-07	05.03	Text for enhanced interrupt support, register set changes. Additional UART discussion.
2014-04-21	04.06	More / rearranged text for the instructions / opcodes section, also more text re. register access.
2014-02-15	04.06	Minor opcode renaming, fixed a few typos in the document.
2014-01-05	04.05	Major edits to reflect 8 stacks and somewhat different opcodes / resized immediates, UART, etc. Updated and expanded the coding examples.
2013-07-07	01.10	Edits to reflect reshuffled opcodes. Fixed immediate add range on page 23. Added “barrel” processor classification and PDP 10 signed shift reference. Other sporadic minor edits.
2013-06-19	01.09	First public release.

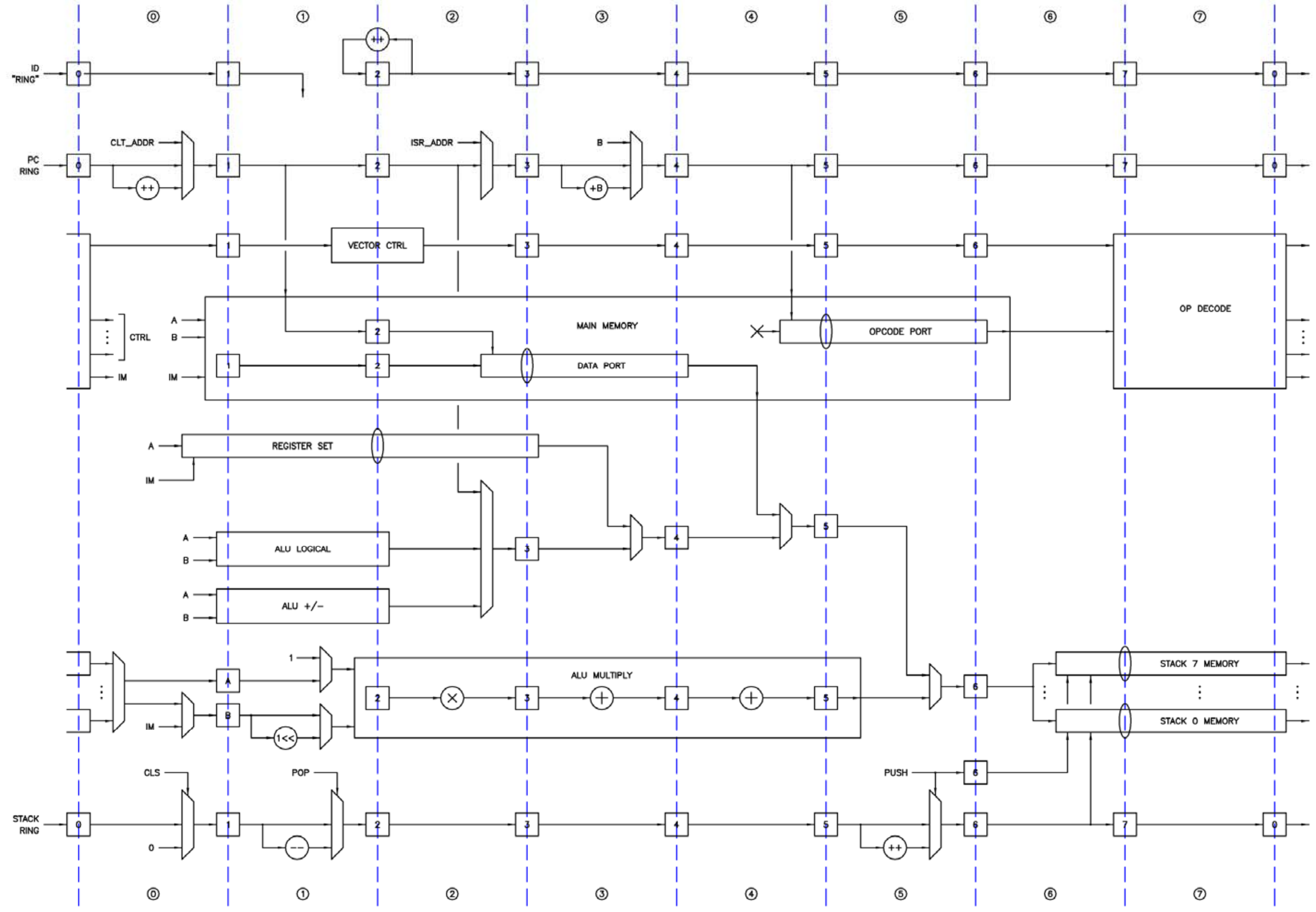


Figure 74. Hive core.

```
// misc
op_nop          = {`nop,              },    // do nothing (no pops either)
op_pop          = {`pop,          `i8x},    // pop[7:0] none/one/some/all stacks
op_cls          = {`cls,          `i8x},    // clear[7:0] none/one/some/all stacks
op_jmp_8        = {`jmp_8,        `i8x},    // PC += I[7:0]  jump immediate unconditional
//
op_pgc          = {`pgc,      `bx,  `ax},    // A := PC  read PC++ (unsigned)
op_lit          = {`lit,      `bx,  `ax},    // A := mem(PC)  literal data
op_lit_s        = {`lit_s,    `bx,  `ax},    // A := mem_lo(PC)  literal data low signed
op_lit_u        = {`lit_u,    `bx,  `ax},    // A := mem_lo(PC)  literal data low unsigned
// conditional (A?B) skips
op_skp_e        = {`skp_e,    `bx,  `ax},    // (A?B) ? PC++  skip 1 conditional
op_skp_ne       = {`skp_ne,   `bx,  `ax},
op_skp_ls       = {`skp_ls,   `bx,  `ax},
op_skp_nls      = {`skp_nls,  `bx,  `ax},
op_skp_o        = {`skp_o,    `bx,  `ax},
op_skp_no       = {`skp_no,   `bx,  `ax},
op_skp_lu       = {`skp_lu,   `bx,  `ax},
op_skp_nlu      = {`skp_nlu,  `bx,  `ax},
//
op_sk2_e        = {`sk2_e,    `bx,  `ax},    // (A?B) ? PC += 2  skip 2 conditional
op_sk2_ne       = {`sk2_ne,   `bx,  `ax},
op_sk2_ls       = {`sk2_ls,   `bx,  `ax},
op_sk2_nls      = {`sk2_nls,  `bx,  `ax},
op_sk2_o        = {`sk2_o,    `bx,  `ax},
op_sk2_no       = {`sk2_no,   `bx,  `ax},
op_sk2_lu       = {`sk2_lu,   `bx,  `ax},
op_sk2_nlu      = {`sk2_nlu,  `bx,  `ax},
// branching
op_jmp_z        = {`jmp_z,    `bx,  `ax},    // (A?0) ? PC += B  jump conditional
op_jmp_nz       = {`jmp_nz,   `bx,  `ax},
op_jmp_lz       = {`jmp_lz,   `bx,  `ax},
op_jmp_nlz      = {`jmp_nlz,  `bx,  `ax},
op_gto_z        = {`gto_z,    `bx,  `ax},    // (A?0) ? PC := B  go to conditional
op_gto_nz       = {`gto_nz,   `bx,  `ax},
op_gto_lz       = {`gto_lz,   `bx,  `ax},
op_gto_nlz      = {`gto_nlz,  `bx,  `ax},
op_jmp          = {`jmp,      `bx,  `ax},    // PC += B  jump unconditional
op_gto          = {`gto,      `bx,  `ax},    // PC := B  go to unconditional
op_irt          = {`irt,      `bx,  `ax},    // PC := B  IRQ return (go to unconditional)
op_gsb          = {`gsb,      `bx,  `ax},    // PC := B, A := PC  subroutine call unconditional
// logical & other
op_cpy          = {`cpy,      `bx,  `ax},    // A := B  copy
op_bnh          = {`bnh,      `bx,  `ax},    // A := {~B[31], B[30:0]}  bit not high
op_cpy_s        = {`cpy_s,    `bx,  `ax},    // A := B[15:0]  low sign extend
op_cpy_u        = {`cpy_u,    `bx,  `ax},    // A := B[15:0]  low zero extend
op_not          = {`not,      `bx,  `ax},    // A := ~B  logical NOT
op_and          = {`and,      `bx,  `ax},    // A &= B  logical AND
op_orr          = {`orr,      `bx,  `ax},    // A |= B  logical OR
op_xor          = {`xor,      `bx,  `ax},    // A ^= B  logical XOR
op_bra          = {`bra,      `bx,  `ax},    // A := &B  logical AND bit reduction
op_bro          = {`bro,      `bx,  `ax},    // A := |B  logical OR bit reduction
op_brx          = {`brx,      `bx,  `ax},    // A := ^B  logical XOR bit reduction
op_flp          = {`flp,      `bx,  `ax},    // A := FLP(B)  flip bits end for end
op_lzc          = {`lzc,      `bx,  `ax},    // A := LZC(B)  leading zero count
// immediate memory access
op_mem_r        = {`mem_r,   `i4x, `bx, `ax},// A := MEM(B+2*I[3:0])  memory read
op_mem_w        = {`mem_w,   `i4x, `bx, `ax},// MEM(B+2*I[3:0]) := A  memory write
op_mem_rs       = {`mem_rs,  `i4x, `bx, `ax},// A[15:0] := MEM(B+I[3:0])  memory read low signed
op_mem_wl       = {`mem_wl,  `i4x, `bx, `ax},// MEM(B+I[3:0]) := A[15:0]  memory write low
// arithmetic
op_add          = {`add,      `bx,  `ax},    // A += B  add
op_add_s        = {`add_s,    `bx,  `ax},    // A += B  add extended signed
op_add_u        = {`add_u,    `bx,  `ax},    // A += B  add extended unsigned
op_sub          = {`sub,      `bx,  `ax},    // A -= B  subtract
op_sub_s        = {`sub_s,    `bx,  `ax},    // A -= B  subtract extended signed
op_sub_u        = {`sub_u,    `bx,  `ax},    // A -= B  subtract extended unsigned
op_mul          = {`mul,      `bx,  `ax},    // A *= B  multiply
op_mul_s        = {`mul_s,    `bx,  `ax},    // A *= B  multiply extended signed
op_mul_u        = {`mul_u,    `bx,  `ax},    // A *= B  multiply extended unsigned
op_shl_s        = {`shl_s,    `bx,  `ax},    // A <<<= B  shift left signed
op_shl_u        = {`shl_u,    `bx,  `ax},    // A <<= B  shift left unsigned
op_pow          = {`pow,      `bx,  `ax},    // A := 1<<B  power of 2
// immediate shifts
op_shl_6s       = {`shl_6s,   `i6x, `ax},    // A <<<= I[5:0]  shift left signed
op_shp_6u       = {`shp_6u,   `i6x, `ax},    // A := 1<<I[5:0]  power of 2;  A <<= I[5:0]  shift unsigned
// immediate register access
op_reg_r        = {`reg_r,    `i6x, `ax},    // A := REG(I[5:0])  register immediate read
op_reg_w        = {`reg_w,    `i6x, `ax},    // REG(I[5:0]) := A  register immediate write
// immediate add
op_add_8        = {`add_8,    `i8x, `ax},    // A += I[7:0]  add immediate signed
// immediate data
op_byt          = {`byt,      `i8x, `ax},    // A := I[7:0]  byte immediate signed
// immediate conditional (A?0) jumps
op_jmp_8z       = {`jmp_8z,   `i8x, `ax},    // (A?0) ? PC += I[7:0]  jump immediate conditional
op_jmp_8nz      = {`jmp_8nz,  `i8x, `ax},
op_jmp_8lz      = {`jmp_8lz,  `i8x, `ax},
op_jmp_8nlz     = {`jmp_8nlz, `i8x, `ax}
```

Figure 75. Hive opcodes.