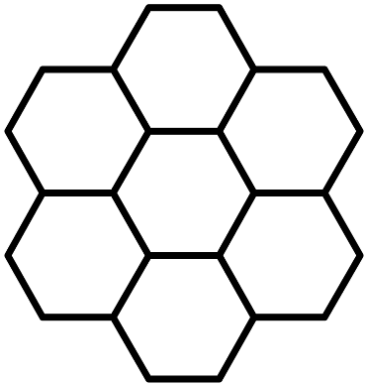

HIVE

32 Bit – 8 Thread – 8 Register/Stack Hybrid – 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, high code density, and excellent ALU utilization – but with very basic control structures and minimal internal state, so it is simple enough for a human to easily grasp and program at the lowest level without any special tools. It fits in the smallest of current FPGAs with sufficient resources left over for peripherals (as well as other unrelated logic) and operates at or near the top speed of the device DSP hardware.

Hive isn't an acronym, 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 and 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, copying to thwart auto-consumption, etc.) normally associated with conventional stack machines, and also minimizes the saving and restoring of register contents normally associated with conventional register machines.

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 that 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 extra overhead, interrupts consume a single branch 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, so if you don't find the architecture of Hive itself to your liking, you may possibly find something else of use in it. Enjoy!

"'Tis the gift to be simple, 'tis the gift to be free…"

From "Simple Gifts" by Elder Joseph

HIVE FEATURE LIST

- A simple, compact, relatively stateless, high speed, barrel pipelined, multi-threaded design based on novel RASH (**R**egister **A**nd **S**tack **H**ybrid) technology.
- 2 operand machine with operand select and stack control fields in the opcode.
- 32 bit data path / ALU with extended width arithmetic results.
- 16 bit compact opcode. 16 & 32 bit memory data access width, both aligned and unaligned.
- Variable width address (set via a build time parameter) – up to 32 bits of directly addressable space.
- 8 equal bandwidth threads.
- 8 independent, isolated, general purpose LIFO data stacks per thread with parameterized depth and fault protections.
- 8 fully independent internal and external interrupts with no hierarchical limitations (one per thread).
- 8 stage pipeline with no stalls or hazards.
- All instructions execute in a single pipeline cycle, including 32 x 32 = 64 bit signed / unsigned multiply (lower 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 code factoring and thread intercommunication.
- Separate (i.e. non-memory mapped) 32 bit wide I/O space.
- 32 bit internal register set in 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.
- 32 bits of GPIO.
- Double buffered UART with BAUD generator and several parameterized options.
- 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 FPGA (Cyclone 3, speed grade 8 – the target device for initial development) while consuming ~2600 logic elements, or ~50% of an EP3C5E144C8.
- 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'm generally 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 really 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 lots of 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 is not unique; I believe it was implemented as far back as 1964 on the CDC 6000 series peripheral barrel processors. Hive shift distances are treated as signed, which works out rather nicely, but the ancient PDP 10 does 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 anther, 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. But 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. And a general purpose register may only be used if the programmer is absolutely 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've 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. Due to the stacked registers, interrupts, subroutines, and other forms of code factoring can be quite efficient; all that is required is that they clean up after themselves. I've 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 end 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 can't 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 doesn't 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 much 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.

So my own personal conclusion is this: a single data stack is a good fit for data input and intermediate results manipulation on a hand calculator. But even with the inclusion of a second general purpose data stack or dedicated return stack, it's not such a great paradigm on which to base processors and programming languages.

BACKGROUND: LIFOS

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

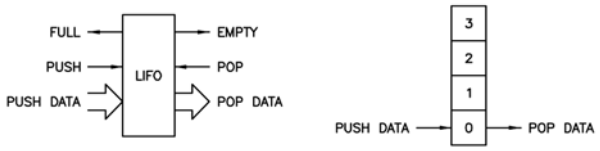

Figure 1. LIFO symbols.

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 only concerned if it is empty or not. Push when full is an error because (depending on stack protection logic) it 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 may give false read data and also may corrupt the LIFO pointer.

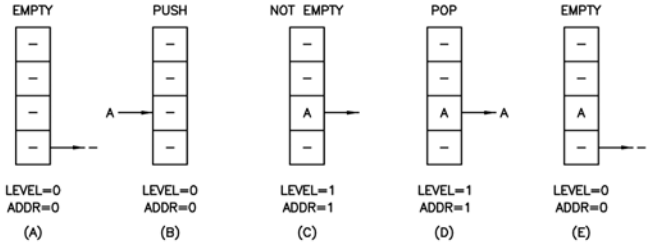

Figure 2. LIFO stack operations – push then pop from empty state.

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.

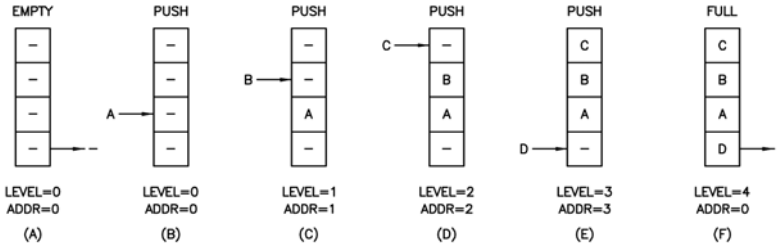

Figure 3. LIFO stack operations – push from empty state to full state.

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 modulo (i.e. the MSB is removed from) the level value. For this 4-deep LIFO there are actually 5 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 the LSBs of this counter to the LIFO memory address input (i.e. the stack pointer).

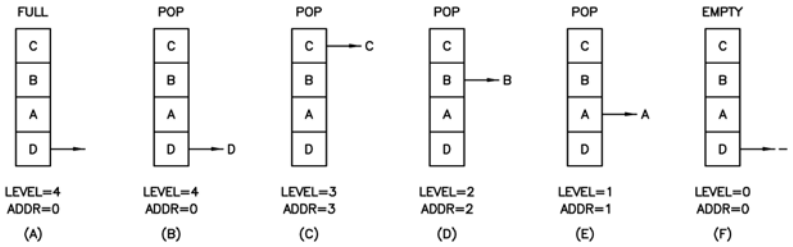

Figure 4. LIFO stack operations – pop from full state to empty state.

The next figure 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 it is flagged as invalid by the empty indicator so the pop side knows not to use it.

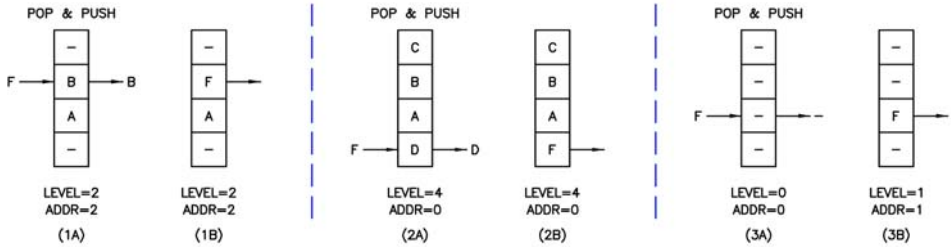

Figure 5. LIFO stack operations – three pop & push scenarios.

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 result onto the stack. This is a pop & push (as opposed to a push & pop, which is 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 pointers are 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? It may seem that the answer to this is always "yes" but consider the following scenario. Say a stack is almost full and a data value is pushed to it, making it full. Then an address is pushed to the stack and the thread attempts to branch to this address. If the pointer and stack memory are overflow protected then the address was dropped on the floor and the thread instead branches to the location given by the previously pushed *data* – off into the weeds it goes with one stack that is essentially stuck and can't accept new data (unless it is a pop & push, or unless a pop is otherwise performed first). The thread could be returned to sanity with an external clear (perhaps issued by another thread on supervisory duty) but the stuck stack means the thread itself has limited ability 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 thread clear? Or perhaps it was about to check itself for stack errors and if it found one would have cleared itself? At least it isn't immediately derailed and off corrupting the contents of main memory.

In contrast, I believe pop (underflow) protection is generally good because it prevents the stack from rolling under and thereby offering up completely unrelated, non-local data and addresses to the thread.

Contemplating how to deal with these "what if" conditions that shouldn't 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. And 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 help keep inefficient and confusing stack thrash to a minimum. 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 value and stack pointer remain unchanged. Whenever anything is written to a stack the value already there is pushed in to make room for the new value. Each stack index is provided with an associated pop bit that influences 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 / register behaviors. For example, 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 don't have conditional single operations because operands are always consumed – the programmer wouldn't be able to tell what state the stack is in after a conditional two operand operation, leading 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 don't change during a single conditional operation.

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 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 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 (the non-incremented PC) is automatically pushed to stack S0 (this is the only “special” stack, and this is the only way in which it is “special”).

So 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 8. This gives a convenient hex nibble of 4 bits for each operand (one pop bit, three stack index bits) for a total of 8 bits of opcode consumed. How deep should the stacks be? I’ve read 32 entries are generally deep enough for single stack machines to not require auto spill-to-memory mechanisms and the like. Since we have 8 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 fairly generous practical lower limit (32 entries per stack per thread in our target device, set via a build-time parameter).

ALU DATA WIDTH

Hive data is 32 bits wide. Byte data is too narrow for many applications, and 16 bit data doesn’t have sufficient resolution to directly perform the internal computations required for audio DSP. 64 bit data is overkill for most applications that would be running on a small FPGA processor. Non-power-of-2 widths can be excluded for efficiency reasons, 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 16 bits wide. 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 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 often 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 (extended) half of the result via instructions. The obvious downside here is that obtaining the full width result takes 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 shouldn't 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 result of signed and unsigned subtraction always forms 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 corner results.

SIGNED VS. UNSIGNED ARITHMETIC

Although addresses are generally thought of as unsigned, 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 largely one of 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 as output need to be differentiated with respect to sign.

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.

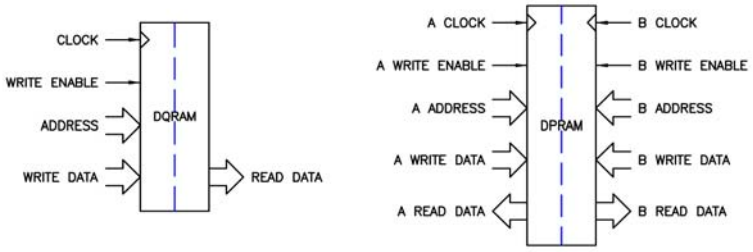

Figure 9. Block RAM: simple (DQ) on left, true dual port (DP) on right.

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, not so much data ports.

Block RAM resources have configurable variable widths, from some maximum down to a single bit. For widths of 8 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 believe 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 / control information.

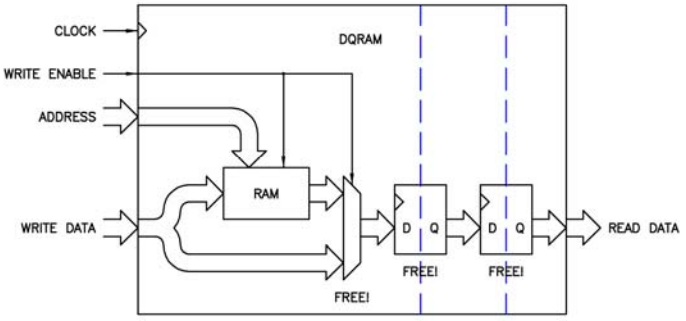

Figure 10. Block RAM internal resources.

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 as asynchronous. This RAM entity is supplemented with "read through" 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've dubbed this *WAR – Write After Read*). With the multiplexer, a read-during-write conveys the data being written to the read data port (*RAW – Read After Write*). Note that these modes only apply to a *given* port of a DP RAM, read/write behavior between ports is never write-through. The register following this optional multiplexer is *always* present. 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 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 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.

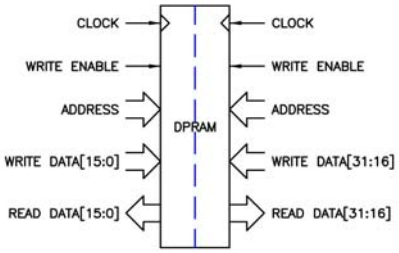

Figure 11. True dual port block RAM utilized as DQ RAM.

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) we can make them 32 bits wide using a single device.

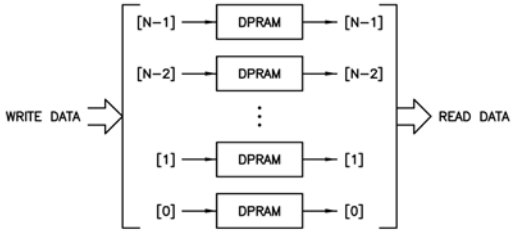

Figure 12. Block RAMs combined via bit-slice.

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. So it behooves us to understand the dedicated multiply hardware.

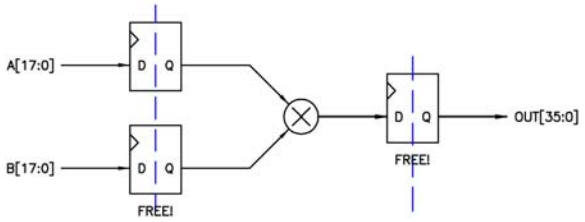

Figure 13. Signed multiplier hardware typically found in an FPGA.

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. 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). Though of course this 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 warnings.

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. Probably 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 is not 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 data (particularly vectors) that cross domain boundaries.

ALU DESIGN

Building an ALU for all but the most trivial of processors is more involved than “compute all results and pick the one you want” (though in the future we may see sufficient unification of DSP blocks across devices and manufacturers and new HDL constructs that allow for more naïve ALU instantiations). Arithmetic and logical calculations aside, the wide output multiplexer itself can be a speed bottleneck. The design of the ALU drives much of the rest of the processor design, particularly one that is pipelined, so it’s not surprising if it takes a fair amount of effort to fully develop and implement it.

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 fairly unsuitable absolute 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’s 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 FPGA hardware multiplier.

Just as multiplication is performed using pencil and paper, addition and concatenation enable the utilization of several hardware multipliers in parallel, thus increasing the input and output widths. Xilinx and Altera both have nice application notes describing how to do this. Consider the following base 10 example:

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

Figure 14. 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 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 4 half width multiplications must be performed, but the three additions have been reduced to two, narrowed, simplified, and therefore likely sped up.

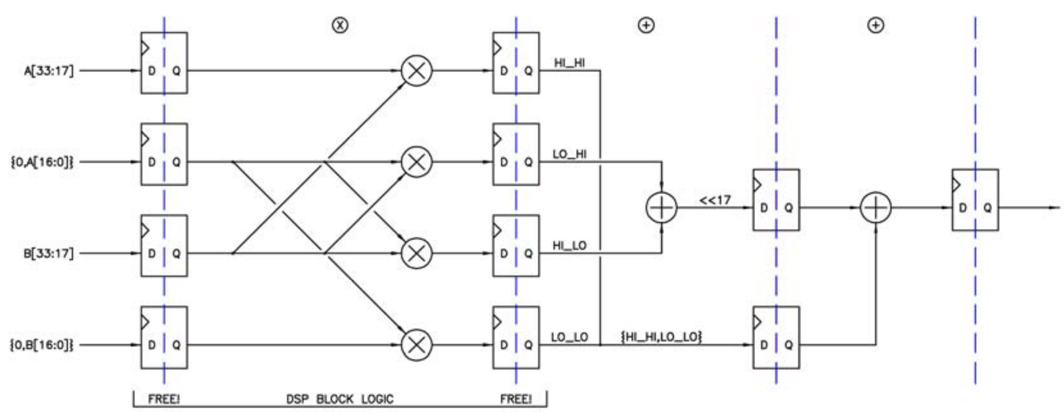

Figure 15. Three stage 33 x 33 = 66 bit signed pipelined multiplication.

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 at least somewhat faster in order to have a chance of hitting 200 MIPS 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 didn't 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.

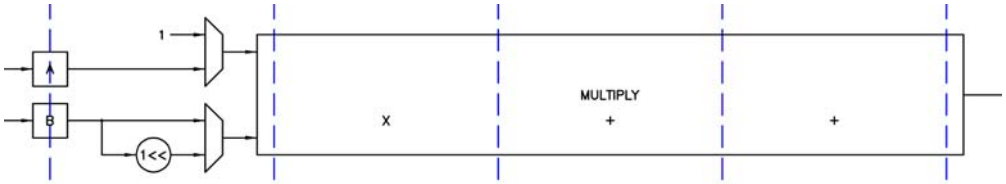

Figure 16. The Multiply and Shift unit.

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 17. 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.) So 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 2 described above. I chose to replace immediate unsigned shift left, non-negative input shift value, with power of 2, which makes it something of an odd man out in terms of operations but hopefully not too confusing. 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 18. 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 with 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 19. Logical functions as implemented (examples here limited to 4 bits).

Note that "BR" stands for "bit reduction" though it is also a mnemonically convenient reminder that B is the input to these single operand functions. 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 full 32 bits, move / copy lower 16 bits both sign and zero extended, 32 bit end-over-end flip, sign bit inversion, and leading zero count.

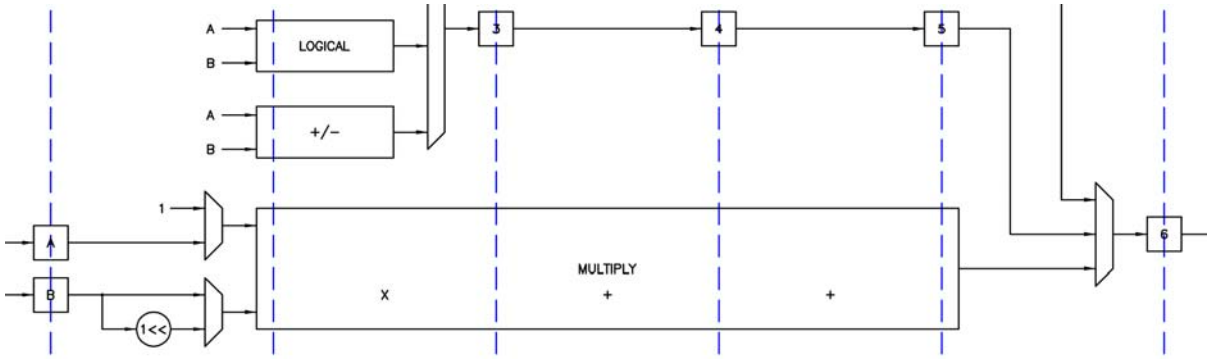

Figure 20. The Arithmetic and Logic Unit (ALU).

The figure above shows the complete ALU. The blue dashed lines represent register boundaries of a pipeline. Data enters from the left and proceeds through the pipe, with the result emerging on the right 6 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 reading and subroutine / interrupt return address use. Read data from the local register set is also multiplexed in between stages 2 and 3. 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 what the control inputs should be and how they 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.

PIPELINED CORE

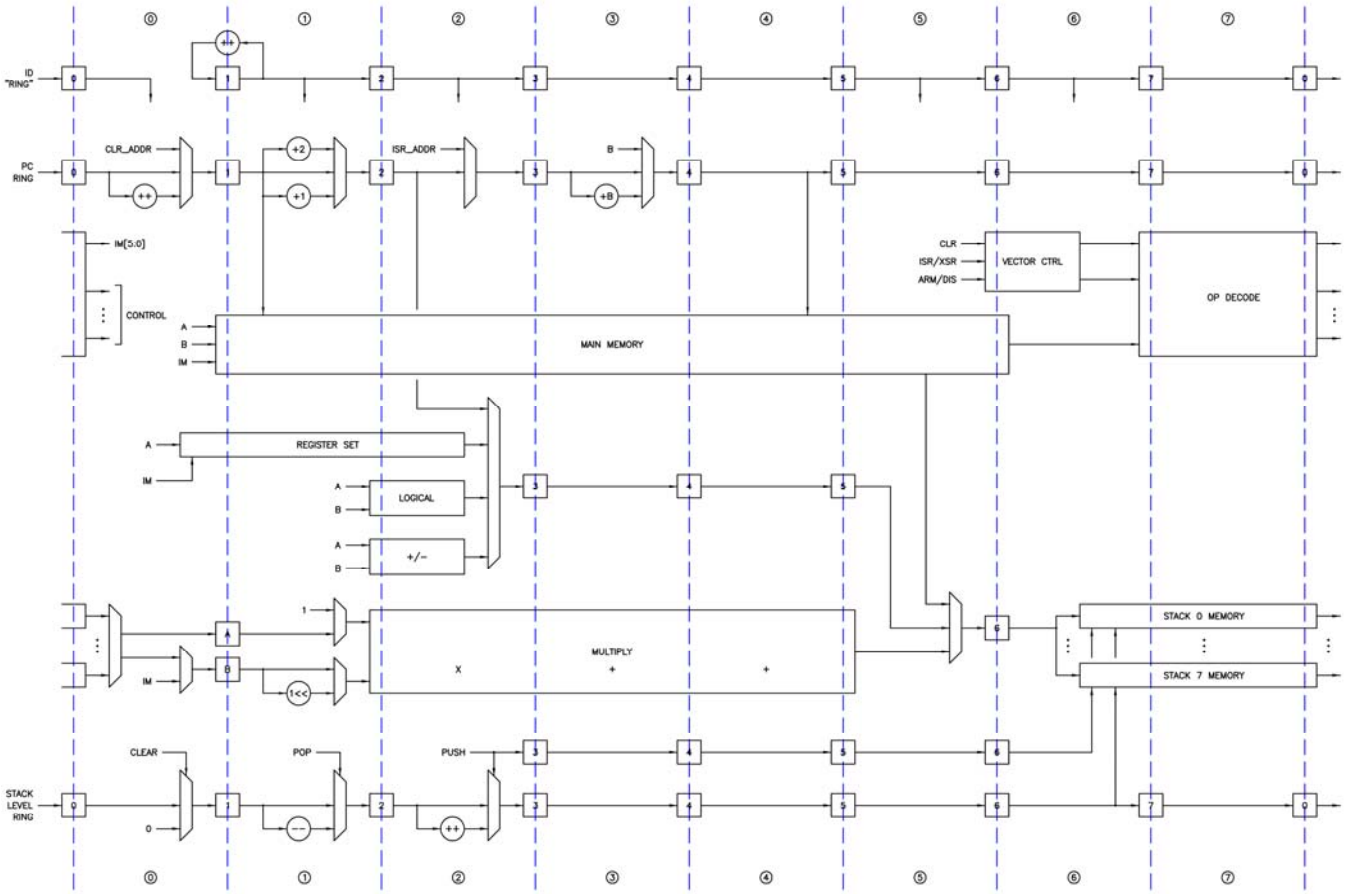

Figure 21. Hive core – view from 100 feet up.

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 data path operations, rather than control path operations. I chose this convention because the lion's share of the core logic – ALU & LIFOs – is contained in the data path.

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. Clearly this also forms a natural and simple scheduling mechanism, with packets of data and associated control information spinning around a global ring like horses on a carousel, all independent of one another, isolated by and stored within the pipe interstage registering, passed from stage to stage in a circular bucket brigade fashion. 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, 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's look at the individual rings in a bit more detail.

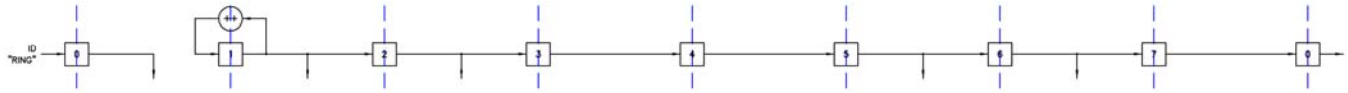

Figure 22. The Time ID "Ring".

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 asynchronous reset with the values they would normally have if previously fed by the counter, and thread ID 0 is the first to emerge from a global reset / clear, followed by 1, 2, etc. Note that this isn't a true scheduler, just a round-robin doling out of identifiers, and that any scheme which produces a continuously repeating fixed pattern where each and every ID is generated once and only once every 8 clocks would suffice. ID here is actually the 3 lowest bits of the 32 bit "Time" counter.

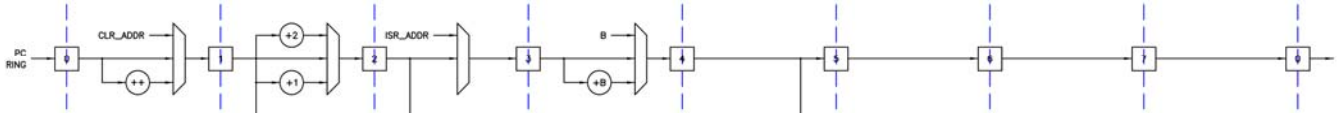

Figure 23. The Program Counter Ring.

Above is the program counter ring. At stage 0 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 1 the PC is used as the address for the main memory data port if retrieving in-line literal data, and the PC is incremented by 1 or 2 if retrieving an in-line 16 or 32 bit literal (to get the next instruction). In stage 2 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 3 the PC is incremented by B (or an immediate value) if taking a relative jump, or replaced by B if performing an absolute jump or subroutine. In stage 4 the PC is used as the address for the main memory instruction port to fetch the next instruction.

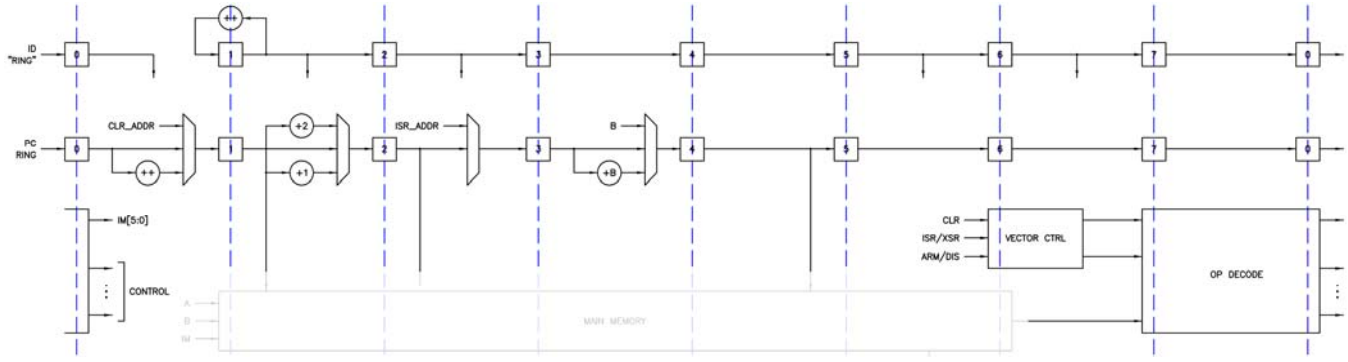

Figure 24. The Control Ring.

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 as a consequence the instruction fetch must happen fairly early in the pipeline, which means conditional testing has to take place even earlier. The vector controller uses the thread ID to correctly inject thread clear and interrupt events into the control ring structure (and to simultaneously 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 won't disturb the other normally functioning threads. The abundance of independent interrupts means that hierarchical interrupt logic / code won't likely be necessary for most applications.

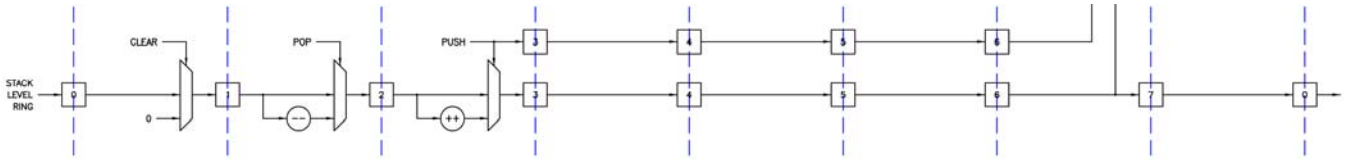

Figure 25. The Stack Level Ring.

Shown above is the stack level ring. In stage 0 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 2 valid push events increment the relevant stack level. Not shown in stages 1 and 2 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 6. Also in stage 6 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 address, 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.

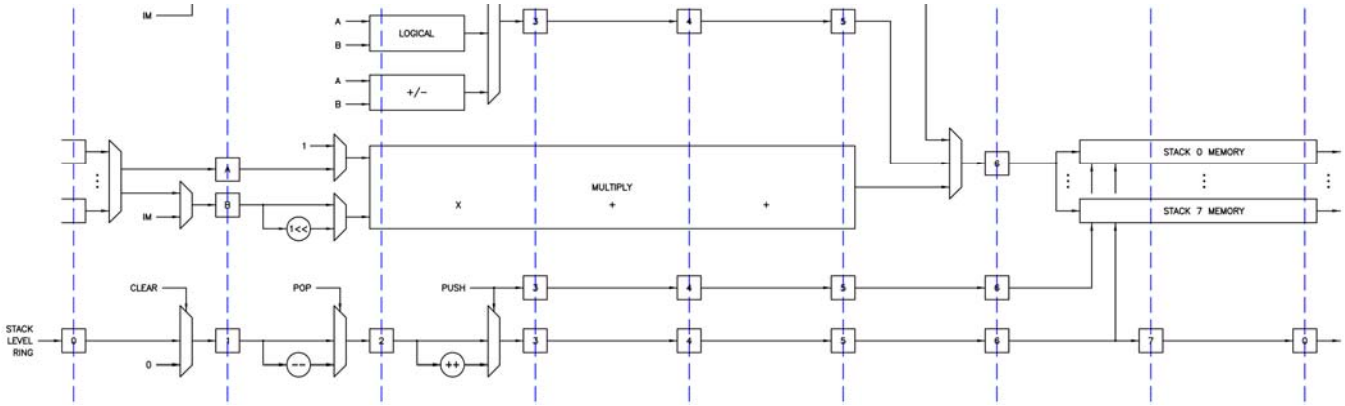

Figure 26. The Data Ring.

As shown above, the stack output multiplexer, ALU, LIFO memories, and stack pointer ring constitute the data ring.

The control ring, data ring, main memory, and local register set make up the Hive core.

INSTRUCTIONS / OPCODES

With the basic hardware structure in place we can now decide on the basic 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 at least minimally accommodate your own early coding examples, but how do you guarantee efficient coverage for all future code you and others may write without resorting to a "kitchen sink" design? Studying the instruction sets of similar processors is useful 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. It's difficult to qualify – much less quantify – this process.

Determining how to best fit the instructions into the opcode space can be a challenge, and you will likely experience design push-back 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 2 stack indexes of 3 bits each, with 1 pop bit for each index. This consumes 8 bits of opcode space, leaving 8 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 (and addresses?)

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 (the value immediately follows the literal 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. There is a full width literal instruction, as well as signed and unsigned literal low instructions.

At the excellent suggestion of one Hive reviewer, I experimented with this in-line mechanism as a source of both absolute addresses and 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 16 or 32 bits. I very reluctantly abandoned this 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 the address offsets 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, and I decided to implement immediate 6 and 4 bit signed instructions, as well as immediate 6 and 4 bit unsigned instructions. The 6 bit unsigned immediate is used exclusively as the register space address, and the 4 bit unsigned immediate is used exclusively as a memory access address offset.

Immediate instruction types and opcode space consumption:

- Two 6 bit unsigned immediate address instructions: register access – 2048 codes.
- Four 4 bit unsigned immediate address instructions: memory access – 16384 codes.
- Six 4 bit signed immediate address instructions: conditional (A?B) jump – 24576 codes.
- Four 6 bit signed immediate address instructions: conditional (A?0) jump – 4096 codes.
- Four 6 bit signed immediate data instructions: signed data, signed add, signed shift left, power / unsigned right shift – 4096 codes.

Immediate Jumps
The longer a loop is, the less we tend to be concerned with loop overhead. But 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 will likely consume a huge portion of the total opcode space.

In the end I decided to implement six 4 bit immediate signed distance [+7/-8] conditional (A?B) jump instructions, and four 6 bit immediate signed distance [+31/-32] conditional (A?0) jump instructions. Jumps are relative to the PC and jump the signed immediate distance if the test is true.

1	C		N	IM[3:0]				PB	B			PA	A		
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

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

Figure 27. Immediate address instruction formats (top to bottom): (A?B) conditional jump; (A?0) conditional jump.

All relative branching is relative to the *next* instruction address, and not *this* instruction address, which seems like 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 or not 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've found other combinations of less than, equal to, and greater than testing to be less useful, and so are not directly supported in Hive. Odd / even testing could be included but the need for it seems less pressing, I ran out of immediate room, and it can be performed when necessary via a flip or shift and sign test. (A?B) testing seems to happen more rarely than (A?0) testing, though it unfortunately consumes more Hive opcode space overall due to the necessary inclusion of the second operand selector field. Less than, and not less than, comparisons have both signed and unsigned variants. Note that there is an unconditional non-immediate jump, but there is no unconditional immediate jump – for short unconditional immediate jumps you can use the (A==B) immediate jump with both A and B fields pointing to the same stack (even an empty stack will work here), or use one of the zero test immediate jumps with A pointing to a stack that has a known quantity.

Some conditional sign conventions / observations:

- The conditional comparisons L (A<Z), and NL (A!<Z) of A to zero necessarily treat A as signed.
- All conditional comparisons of A and B that are signed | unsigned treat *both* A and B as signed | unsigned.
- The equivalency 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 read which is sign extended. 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 off of a base address, and it is up to the programmer to manage this field correctly for 32 bit accesses (note that they need not be aligned to even base addresses).

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

Figure 28. 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 a lot of cycles on average, so they should be made as efficient as possible.

Immediate Register Access
There are 2 instructions for register set access: full 32 bit read and write. An unsigned 6 bit immediate field provides for access of up to 64 registers.

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

Figure 29. Immediate address instruction format: register access.

Immediate Data
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 6 signed bits of data.

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

Figure 30. Immediate data instructions format: data, add, and shift / power.

Immediate Shifts / Power of 2
Shifts distances / directions known at "compile time" can be 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 flag, arbitrary width sign / zero extension, isolation of contiguous bit fields, 2^n integer modulo, etc.).

The immediate unsigned shift performs a power of 2 (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 and decrements [+31/-32].

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

- **JMP** (jump) is relative to the PC and is either conditional or unconditional. It jumps a signed distance given by B if either the test (A?0) is true, or unconditionally.
- **GTO** (go to) is absolute and unconditional. It loads the PC with the value given by B.
- **RTN** (return) is a GTO that re-enables the ISR state machine.
- **GSB** (go to subroutine) is absolute and unconditional. It loads the PC with the value given by B and stores the return address to A.

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

Figure 31. (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 much deliberation I decided against the inclusion of conditional GTO and GSB instructions as being too confusing and not having sufficient need / utility.

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 2 here as exists in the immediate form) so that there could not be unexpected behavior, and so sign testing of the shift variable would not be necessary. A separate variable power function generates powers of 2 and is sign agnostic regarding the shifted 1 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.

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

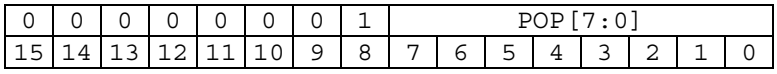

Figure 32. Pop instruction format.

Other Instructions
There are several stack and miscellaneous instructions:

- **PCP** 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 all operations do not support all options.

op_*	Function
nop	**N**o **OP**eration (no pops either)
pop	**POP** : one hot bit per stack
pcp	**P**rogram **C**ounter **P**lus : 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
isg	**I**nvert **S**i**G**n : A:={~B[31], B[30:0]}
not	Logical **NOT** : A:=~B
and	Logical **AND** : A:=A&B
orr	Logical **ORR** : A:=A|B
xor	Logical **XOR** : A:=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:=A+B
sub	Arithmetic **SUB**traction : A:=A-B
mul	Arithmetic **MUL**tiplication : A:=A*B
shl	**Sh**ift **L**eft : A:=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
rtn	**R**e**T**ur**N** : 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
dat	**DAT**a : A:=I
pus	**P**ower | **U**nsigned **S**hift right

Figure 33. Instruction operations.

Op_*_?	Function
i	**I**mmediate
x	e**X**tended
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 34. Operation options.

Rules for these options are:

- No underscore between options.
- The option **i** if present comes first.
- The options **x**, **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**, **u**, or **h** if present go last.
- Some operations exist only in an immediate form (dat, reg, mem) and the rule here is to always include the immediate **i** option regardless.

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 helped to reveal similar decoding patterns which were then 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				P7	P6	P5	P4	P3	P2	P1	P0
256	**pcp**	0x0				0x3				PB	B			PA	A		
768	**lit**	0x0				0	1	L	U	PB	B			PA	A		
2048	**reg_i**	0x0				1	W	IM[5:0]						PA	A		
768	**cpy**	0x1				0	0	L	U	PB	B			PA	A		
256	**isg**	0x1				0x1				PB	B			PA	A		
256	**not**	0x1				0x4				PB	B			PA	A		
256	**and**	0x1				0x5				PB	B			PA	A		
256	**orr**	0x1				0x6				PB	B			PA	A		
256	**xor**	0x1				0x7				PB	B			PA	A		
256	**bra**	0x1				0x8				PB	B			PA	A		
256	**bro**	0x1				0x9				PB	B			PA	A		
256	**brx**	0x1				0xA				PB	B			PA	A		
256	**flp**	0x1				0xC				PB	B			PA	A		
256	**lzc**	0x1				0xD				PB	B			PA	A		
768	**add**	0x2				0	0	X	U	PB	B			PA	A		
768	**sub**	0x2				0	1	X	U	PB	B			PA	A		
768	**mul**	0x2				1	0	X	U	PB	B			PA	A		
512	**shl**	0x2				1	1	0	U	PB	B			PA	A		
256	**pow**	0x2				0xE				PB	B			PA	A		
1024	**jmp (A?0)**	0x3				0	0	L	N	PB	B			PA	A		
256	**jmp**	0x3				0xC				PB	B			PA	A		
256	**gto**	0x3				0xD				PB	B			PA	A		
256	**rtn**	0x3				0xE				PB	B			PA	A		
256	**gsb**	0x3				0xF				PB	B			PA	A		
16384	**mem_i**	0	1	W	L	IM[3:0]				PB	B			PA	A		
24576	**jmp_i (A?B)**	1	C		N	IM[3:0]				PB	B			PA	A		
4096	**jmp_i (A?0)**	0xE				L	N	IM[5:0]						PA	A		
1024	**dat_i**	0xF				0	0	IM[5:0]						PA	A		
1024	**add_i**	0xF				0	1	IM[5:0]						PA	A		
1024	**shl_is**	0xF				1	0	IM[5:0]						PA	A		
1024	**pus_i**	0xF				1	1	IM[5:0]						PA	A		

Figure 35. 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 catch-all, the logical group second, arithmetic third, and branching fourth. The immediates follow, with memory access first, (A?B) jumps next, (A?0) jumps following, and immediate data, add, and shifts 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, and this probably is as it should be for a processor with compact opcodes. It can be a long road leading up to the final selection of operations and their encoding, with much inserting and deleting of operations, resizing of immediate fields, and reshuffling of the encoding space, and I'm not sure there is any way to abbreviate this activity and still really do it justice.

MAIN MEMORY

Hive data and program memory space are shared. If we desire full 32 bit single cycle access to 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 don't 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 memory. 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 based code space with no restrictions. For 32 bit values at even base addresses the access is straightforward. But for 32 bit values at odd base addresses we need to swap the 16 bit values both going into and coming out of the data memory port, and swap the 16 bit write enables as well. But 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 differing address values necessitates the use of separate ports – the high and low 16 bit memories must be physically separate 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 index.

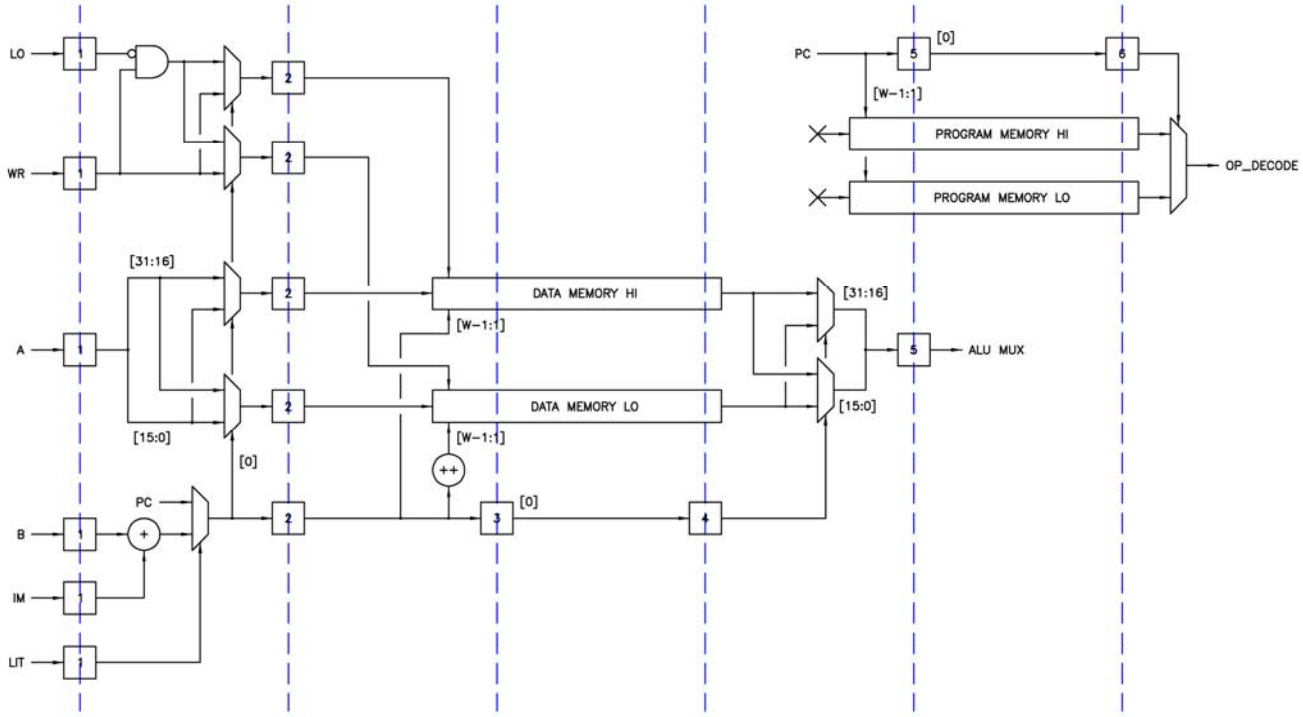

Figure 36. Hive main memory.

The main memory module for Hive is shown in the above figure. In stage 1 the PC is selected as the address for literals, and B+IM is selected otherwise for data access. The LSB is stripped off to swap input 16 bit data chunks and write enables. In stage 2 the address with its LSB stripped off is use as the upper BRAM address, the incremented address with its LSB stripped off is use 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 output address for odd input address but not for even addresses, so no multiplexing or switching is necessary here. In stage 4 the pipelined address LSB is used to swap the read 16 bit data chunks, with the result registered in stage 5 and 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 stage 4 and the truncated result applied to both high and low memory address ports. The pipelined LSB is used to select the output in stage 6, the result of which is fed to the opcode decoding unit.

INTERNAL REGISTER SET

Any processor core will need a local, or internal register set to manage things like the reporting of basic operational errors, enabling and disabling of interrupts, general purpose I/O communications, timers, UARTs, watchdog sanity timers, shoot yourself in the head resets, etc. But register set implementation can be a dull, repetitive, and bug prone exercise. To automate this to some degree and to reduce the chance of errors, at the foundation of the Hive register set is a configurable multi-function single base register component with many parameter-based options.

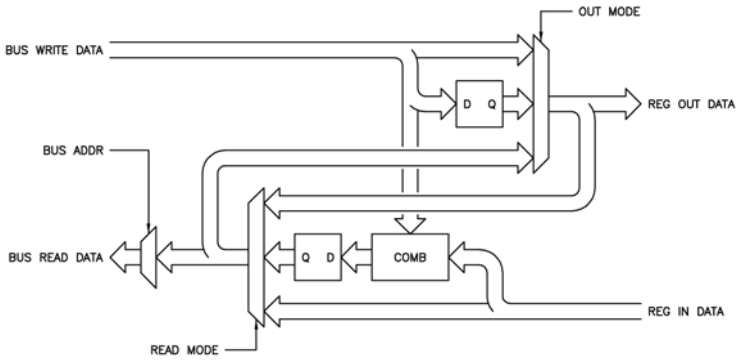

Figure 37. Configurable base register component.

The figure above shows a schematic view of the single base register component. On the left is the common processor expansion bus, on the right are the individual inputs and outputs of the single register. Not shown is logic that detects the address match, nor the read / write / in / out signal conditioning. The two large multiplexers set the read and output modes via parameters at build time. Any number and combination of bits can be "live" (provided with functional logic) and initialized to a known value at reset. Register input data can be optionally resynchronized and/or made edge sensitive. 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 aren't directly supported.

Parameter	Output Mode
"ZERO"	zero output
"THRU"	no latch, direct connect
"REGS"	outputs registered (*not* latched)
"LTCH"	output data latched
"LOOP"	no output latch, output selected read data

Figure 38. Output mode options.

Parameter	Read Mode
"THRU"	no latch, direct connect
"CORD"	set on input one, clear on read
"COW1"	set on input one, clear on write one
"DFFE"	D type flip flop with enable
"LOOP"	no read latch, read selected out data

Figure 39. Read mode options.

Multiple base register components are assembled into a register set by using a big OR gate to combine their bus side data read port bits. Placement of each individual base register within the register set address space is governed by an input parameter to each base register component. Access to the register set is via special REG instructions and the internal ALU mux.

Unlike their ASIC brethren, one nice thing that soft processor cores have going for them is they don't need huge gobs of configuration registers. Want a timer? Write one in SystemVerilog, connect it to an interrupt, and it's game over. Want 8 UARTs? No problem – you don't have to put them in until you really need them and you can take them out later if / when you don't.

The Hive internal register set includes the following basic functionality:

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


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

  bit  name                 description
-----  ----                 -----------
31-16  -                    0
15-00  ver[15:0]            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 : Error register - error_reg
--------------------------------------------------------------------------------

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

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

================================================================================
- 0x02 : Time / ID register - time_id_reg
--------------------------------------------------------------------------------

  bit  name                 description
-----  ----                 -----------
31-00  time_id[31:0]        time / thread ID

Notes:
- Read-only.
- Up-count @ core clock rising edges.
- Threads can read this for relative time and to discover their
  thread ID (3 LSBs).

================================================================================
```

```
================================================================================
- 0x03 : Vector register - vector_reg
--------------------------------------------------------------------------------

  bit  name                 description
-----  ----                 -----------
31-24  clr[7:0]             write 1 to clear thread
23-16  isr[7:0]             write 1 to interrupt thread
15-08  isr_arm[7:0]         1=thread interrupt arm
07-00  isr_dis[7:0]         1=thread interrupt disarm

Notes:
- Per thread clear (non-maskable).
- Per thread internal interrupt (ISR) (maskable).
- Per thread disarm / arm of internal / external ISRs.
- Set on write one radio buttons for ISR disarm / arm.
- Clear takes precedence over ISR.
  - e.g. write 0xFFFFFFFF clears all threads.
- Disarm takes precedence over arm.
  - e.g. write 0x0000FFFF disarms all threads.
- ISRs disarmed @ clear and async reset (0x000000FF).
- Next ISR automatically disarmed until op_rtn encountered.
- ISR auto disarm gets reset with register disarm then arm.

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

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

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
================================================================================
```

VECTOR SUPPORT

Two types of vectoring, or breaking out of current execution, are supported in Hive. The first is thread clearing, where the program counter (PC) is loaded with the initial base address for the thread, and the stack pointers are cleared. The second is interrupt service routine (ISR) support, where the PC is pushed to stack 0 and the PC is loaded with the ISR address for that thread.

Thread Clear

A thread clear is issued via the VECTOR register by writing a one to the corresponding thread bit. In this manner threads can clear themselves, and can clear any combination of other or all threads as well. All relevant state such as stack pointers and interrupt service request arming is automatically cleared for the thread being cleared. This same mechanism is used at asynchronous reset of the core to initialize all threads.

Thread Interruption

A thread may be interrupted to run a service routine internally via a register-based mechanism similar to the thread clearing described above, and also via an external request. The thread must be armed to handle ISRs before it will respond to them. While servicing an interrupt the thread is automatically disarmed so that subsequent ISRs are ignored until completion of the current ISR. The operation *op_rtn* simultaneously returns the thread to the point of execution before it was interrupted and rearms the thread for interrupt operation if it was armed in the first place. This automatic arm/disarm action prevents stack overflow in the event of noise or a series of closely spaced interrupts. Any interrupts requested during ISR execution are lost, so if your algorithm can't afford to miss any interrupts you will need to modify this construct or add extra hardware to count / time stamp interrupts. Clearing a thread automatically disarms its ISR.

Arming and disarming the ISR for a thread is performed by writing a one to the associated arm or disarm bit. These bits behave like radio buttons, where the last one "pressed" 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 it safe for multiple access and control by all threads. Reading these bit fields will reveal the current armed/disarmed state for all threads.

At build time the user can chose per-thread external ISR input conditioning options. Rising edges and / or falling edged may be detected. If no edges are desired the input is disabled and the conditioning logic removed, though ISRs may still be issued internally via the register set.

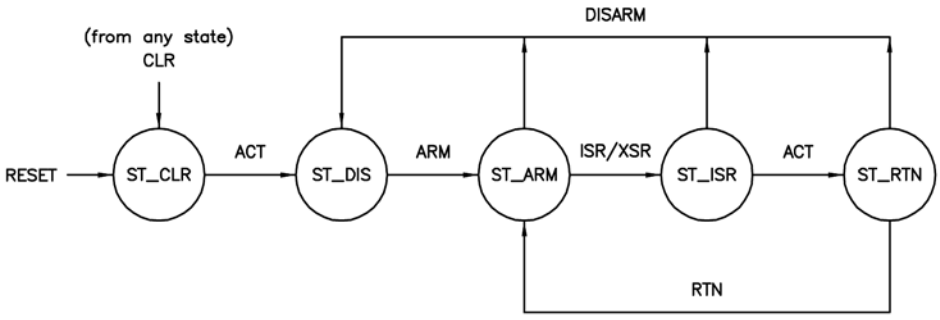

Figure 40. Vector state machine.

Each thread has a vector state machine as shown in the above figure. At asynchronous reset, or when a one is written to the clear bit in the register set, the clear state is entered. The machine stays in this state until the clear is issued to the opcode decoder, whereupon the machine moves to the disarmed state. When a one is written to the arm bit in the register set, the machine moves to the armed state. When an internal or external interrupt is requested the machine moves to the ISR state until the interrupt is issued to the op code decoder, whereupon the machine moves to the return state. When an op_rtn instruction is decoded by the opcode decoder the machine moves back to the armed state. Note that writing a one to the disarm bit will return the machine to the disarmed state, thus making it "forget" if it was currently waiting for an op_rtn instruction to be executed. So disarming and then rearming the ISR, or issuing an op_rtn are two ways to return the state machine to the armed state where it can respond to a new ISR.

UART FUNCTIONALITY

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

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 levels are desired. The serial bits are in this order: one start bit (low), eight data bits with LSB first, MSB last, one or more stop bits (high). 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. Several errors are reported including 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. (But the errors and loopback aren't currently brought to the register set.)

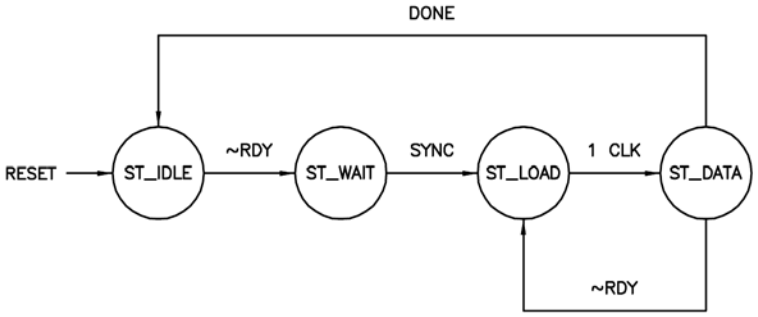

Figure 41. TX UART state machine.

The TX UART state machine is shown above. 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 from the BAUD generator is achieved it transitions to the load state, where the parallel data is taken, ready is returned high to signal 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 the machine is already synchronized with the BAUD generator, so 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.

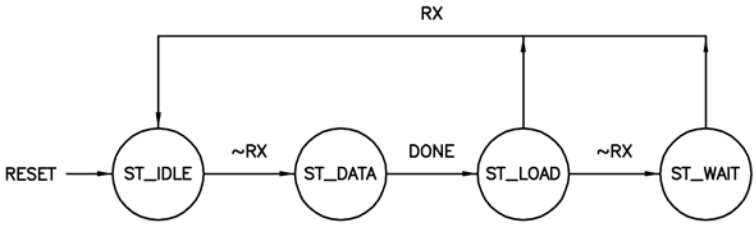

Figure 42. RX UART state machine.

The RX UART state machine is shown above. 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.

By default the UART is configured to be the standard 8n1, with a fixed 115200 baud rate, with the design parameters automatically calculated at build time given the core clock speed and desired baud rate.

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 architectures themselves should be guided by principle as well. Simplicity allows one to more easily 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 which are as self-contained as possible, making verification a straightforward and relatively painless task.

It goes without saying that 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 can enable the processor to essentially verify itself. Stack functioning and error reporting may be fully tested for all threads. Jump distances and all associated conditionals may be confirmed. Each opcode should be tested to make sure it is decoded and functioning correctly – distinctive signatures may be used here rather than exhaustive testing. This is also a good way to get some early experience hand coding the processor (the point at which I've became largely disillusioned with my past designs) which often leads to changes in the op codes or other parts of the fundamental design.

Finally, several simple algorithms should be coded up, first in a spreadsheet and then in the simulated core, with the results compared. When working on this phase of the design I find that I have 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. So I coded up a fully parameterized SystemVerilog module and speed / functionally tested it. But only after I recognized the general use of this function (it has floating point normalization uses as well) did I include it in the logical unit of the Hive ALU.

SPEED

Job #2 when building a processor is getting the top speed as high as possible. To this end most Hive modules have configurable registering on 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 then 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. Just as C compilers can often beat the best human hand coders, there are various FPGA synthesis options that will likely produce a faster top speed, 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 some time in your life 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 sometimes 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.

PROGRAMMING EXAMPLES

The SystemVerilog hardware description language has an “initial” construct that can be used along with other SystemVerilog syntax features to write fairly legible boot code, comments and all. Hive boot code text resides in a text file (boot_code.h) that gets inserted into the main memory module with an include statement. Let’s take a look at some sample boot code:

```
`include "boot_code_defs.h"
```

This include pulls 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;
initial begin
```

The above declares an integer we’ll use to keep from having to name each and every address, and marks the beginning of the initialization code.

A Simple Example

```
// clr space //
i='h0;   ram[i] = { `lit_lu,           `__, `s7 };  // s1='h0040
i=i+1;   ram[i] =                     16'h0040  ;  //
i=i+1;   ram[i] = { `gsb,              `P7, `s3 };  // gsb, pop s7
i=i+1;   ram[i] = { `jmp_ie,    -4'd1, `s0, `s0 };  // loop forever
// all others : loop forever
i='h04;  ram[i] = { `jmp_ie,    -4'd1, `s0, `s0 };  // loop forever
i='h08;  ram[i] = { `jmp_ie,    -4'd1, `s0, `s0 };  // loop forever
i='h0c;  ram[i] = { `jmp_ie,    -4'd1, `s0, `s0 };  // loop forever
i='h10;  ram[i] = { `jmp_ie,    -4'd1, `s0, `s0 };  // loop forever
i='h14;  ram[i] = { `jmp_ie,    -4'd1, `s0, `s0 };  // loop forever
i='h18;  ram[i] = { `jmp_ie,    -4'd1, `s0, `s0 };  // loop forever
i='h1c;  ram[i] = { `jmp_ie,    -4'd1, `s0, `s0 };  // loop forever

// intr space //

// code & data space //

// sub : read core version & write to GPIO, return to (s3)
i='h40;  ram[i] = { `reg_ir,        `VER_A, `s0 };  // s0=reg(VER_A)
i=i+1;   ram[i] = { `reg_iw,         `IO_A, `P0 };  // reg(IO_A)=s0, pop s0
i=i+1;   ram[i] = { `gto,              `P3, `__ };  // return, pop s3
```

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.

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:

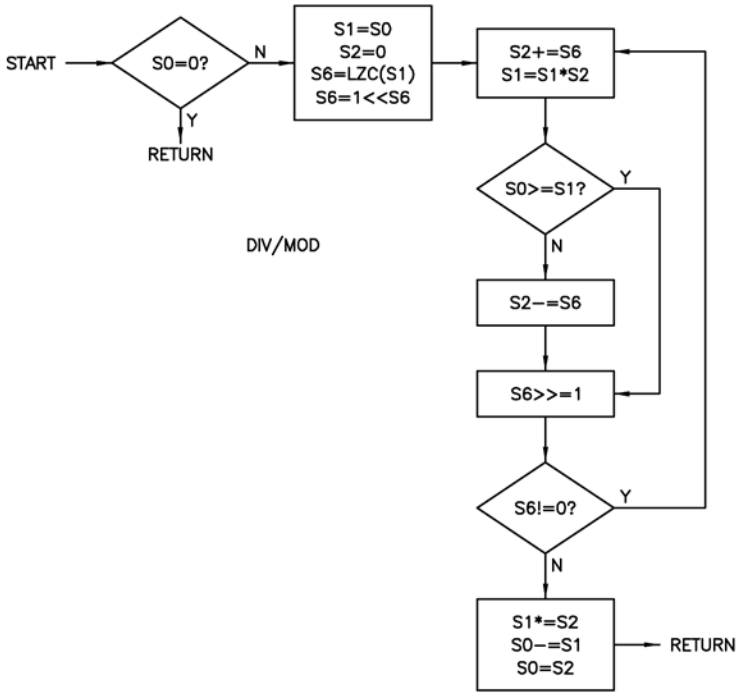

Figure 43. Binary search division algorithm flow chart.

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 trial number Q, multiply it by the denominator D, and compare it to the numerator N.

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.

```
// sub : unsigned divide & modulo remainder, return to (s7)
//
// algorithm: binary search
//
// s0 : N, D(top)/N(under) input, Q(top)/R(under) output
// s1 : D
// s2 : Q
// s3 :
// s4 :
// s5 :
// s6 : one-hot (& loop test)
// s7 : sub return address
//
// (D=0)? is an error, return
i='h90;  ram[i] = { `jmp_inz,         6'd1, `s0 };  // (s0!=0) ? skip return
i=i+1;   ram[i] = { `gto,              `P7, `__ };  // return to (s7), pop s7
// loop setup
i=i+1;   ram[i] = { `cpy,              `P0, `s1 };  // s0=>s1 (s1=D, s0=N)
i=i+1;   ram[i] = { `dat_is,          6'd0, `s2 };  // s2=0 (s2=init Q)
i=i+1;   ram[i] = { `lzc,              `s1, `s6 };  // s6=lzc(s1)
i=i+1;   ram[i] = { `pow,              `s6, `P6 };  // s6=1<<s6, pop s6 (s6=init OH)
// loop start
i=i+1;   ram[i] = { `add,              `s6, `P2 };  // s2+=s6 (s2=new trial Q)
i=i+1;   ram[i] = { `mul,              `s2, `s1 };  // s1=s1*s2 (s1=D*Q)
// jump start
i=i+1;   ram[i] = { `jmp_inlu,   4'd1, `P1, `s0 };  // (s0>=s1) ? skip restore, pop s1 (N>=D*Q)
i=i+1;   ram[i] = { `sub,              `s6, `P2 };  // s2-=s6 (s2=restored Q)
// jump end
i=i+1;   ram[i] = { `pus_i,          -6'd1, `P6 };  // s6>>=1 (new OH)
i=i+1;   ram[i] = { `jmp_inz,        -6'd6, `s6 };  // (s6!=0) ? do again
// loop end
// calc remainder, move Q
i=i+1;   ram[i] = { `mul,              `s2, `P1 };  // s1*=s2 (s1=D*Q)
i=i+1;   ram[i] = { `sub,              `P1, `P0 };  // s0-=s1, pop both (s0=N-D*Q=R)
i=i+1;   ram[i] = { `cpy,              `P2, `s0 };  // s0=s2, pop s2 (s0=Q)
// return
i=i+1;   ram[i] = { `gto,              `P7, `P6 };  // return to (s7), pop s7 & s6
// end sub
```

The subroutine code is above. 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 isn't 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:

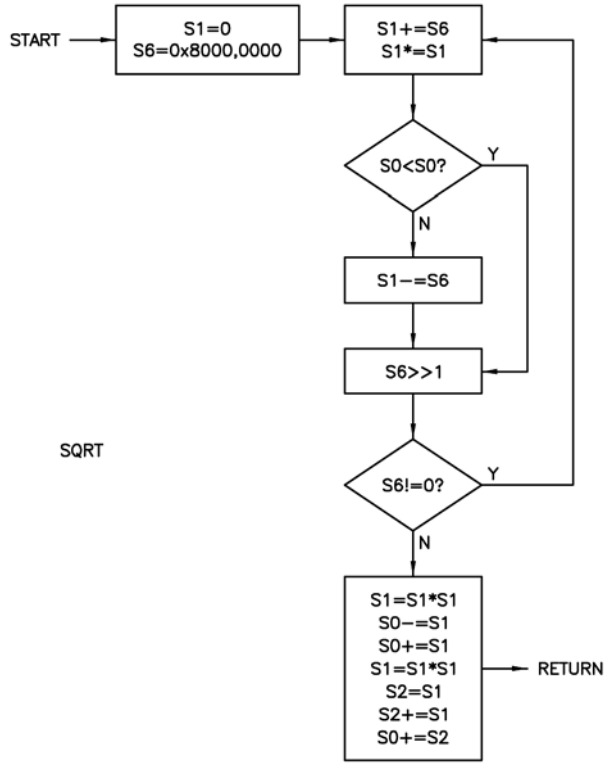

Figure 44. Binary search square root algorithm flow chart.

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 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 or not 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.

```
    // sub : s0=sqrt(s0), return to (s7)
    //
    // input is unsigned 32 integer
    // output is unsigned 16.16 integer
    //
    // algorithm: binary search
    // iterate 32 times
    //
    // s0 : input (x), output (Q)
    // s1 : running q
    // s2 : temp: q^2 + q
    // s3 :
    // s4 :
    // s5 :
    // s6 : one-hot (& loop test)
    // s7 : sub return address
    //
    // loop setup
    i='h90;  ram[i] = { `dat_is,          6'd0, `s1 };  // s1=0 (init Q)
    i=i+1;   ram[i] = { `pus_i,          6'd31, `s6 };  // s6 MSB=1 (init OH)
    // loop start
    i=i+1;   ram[i] = { `add,              `s6, `P1 };  // s1+=s6
    i=i+1;   ram[i] = { `mul_xu,           `s1, `s1 };  // s1=s1*s1 (square, integer portion)
    // jump start
    i=i+1;   ram[i] = { `jmp_ilu,    4'd1, `s0, `P1 };  // (s1<s0) ? jump 1  pop s1 (skip restore)
    i=i+1;   ram[i] = { `sub,              `s6, `P1 };  // s1-=s6 (restore)
    // jump end
    i=i+1;   ram[i] = { `pus_i,          -6'd1, `P6 };  // s6>>=1 (new OH)
    i=i+1;   ram[i] = { `jmp_inz,        -6'd6, `s6 };  // (s6!=0) ? do again
    // loop end
    i=i+1;   ram[i] = { `mul_xu,           `s1, `s1 };  // s1=s1*s1 (square, integer portion)
    i=i+1;   ram[i] = { `sub,              `P1, `P0 };  // s0-=s1, pop s1 : x -= q^2
    i=i+1;   ram[i] = { `add,              `s1, `P0 };  // s0+=s1 : Q = q + x - q^2
    i=i+1;   ram[i] = { `mul,              `s1, `s1 };  // s1=s1*s1 (square, decimal portion) : (q>>n)^2
    i=i+1;   ram[i] = { `cpy,              `P1, `s2 };  // s2=s1, move
    i=i+1;   ram[i] = { `add_xu,           `P1, `P2 };  // s2+=s1 (carry out, integer portion), pop s1 :
int[q>>n + (q>>n)^2]
    i=i+1;   ram[i] = { `sub,              `P2, `P0 };  // s0+=s2, pop s2 : Q = q + (x - q^2) - int[q>>n
+ (q>>n)^2]
    i=i+1;   ram[i] = { `gto,              `P7, `P6 };  // return, pop s7 & s6
    // end sub
```

The subroutine code is above. 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 $\log2(x^2) = 2*\log_2(x)$, and is implemented by a looped squaring processes.

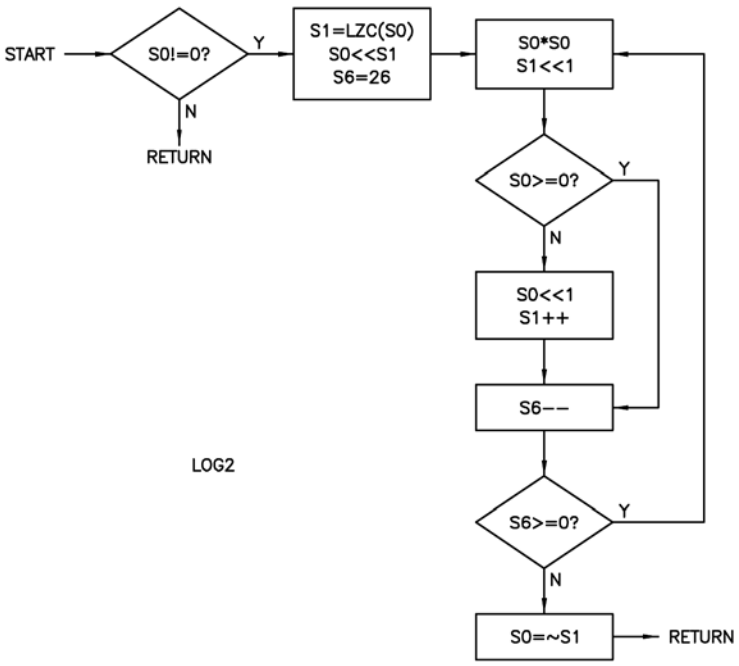

Figure 45. Log base 2 algorithm flow chart.

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 until the MSB is equal to 1 (which tests as negative). 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 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, it's likely that are – there may be a simpler, more elegant approach to be found with a bit more thought.)

```
// sub : s0=log2(s0), return to (s7)
//
// input is in[31:0] an unsigned 32 bit integer
// output is c[31:27].m[26:0] unsigned fixed decimal
//
// s0 : input, normalize, square, output
// s1 : lzc, result
// s2 :
// s3 :
// s4 :
// s5 :
// s6 : loop index
// s7 : sub return address
//
// (input=0)? is an error, return
i='h90;  ram[i] = { `jmp_inz,         6'd1, `s0 };  // (s0!=0) ? skip return
i=i+1;   ram[i] = { `gto,              `P7, `__ };  // return to (s7), pop s7
// normalize
i=i+1;   ram[i] = { `lzc,              `s0, `s1 };  // s1=lzc(s0)
i=i+1;   ram[i] = { `shl_s,            `s1, `P0 };  // s0<<=s1 (normalize)
// loop setup
i=i+1;   ram[i] = { `dat_is,         6'd26, `s6 };  // s6=26 (loop index)
// loop start
i=i+1;   ram[i] = { `mul_xu,           `s0, `P0 };  // s0*=s0
i=i+1;   ram[i] = { `shl_is,          6'd1, `P1 };  // s1<<=1
// jump start
i=i+1;   ram[i] = { `jmp_ilz,         6'd2, `s0 };  // (s0[31]==1) ? jump
i=i+1;   ram[i] = { `shl_is,          6'd1, `P0 };  // s0<<=1
i=i+1;   ram[i] = { `add_is,          6'd1, `P1 };  // s1++
// jump end
i=i+1;   ram[i] = { `add_is,         -6'd1, `P6 };  // s6--
i=i+1;   ram[i] = { `jmp_inlz,       -6'd7, `s6 };  // (s6>=0) ? do again
// loop end
// cleanup, return
i=i+1;   ram[i] = { `not,              `P1, `P0 };  // s0=~s1, pop both
i=i+1;   ram[i] = { `gto,              `P7, `P6 };  // return, pop s7 & s6
// end sub
```

The subroutine code is above. 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 isn't zero and the return is skipped, it takes 4 cycles to test the input, normalize it, and setup 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.

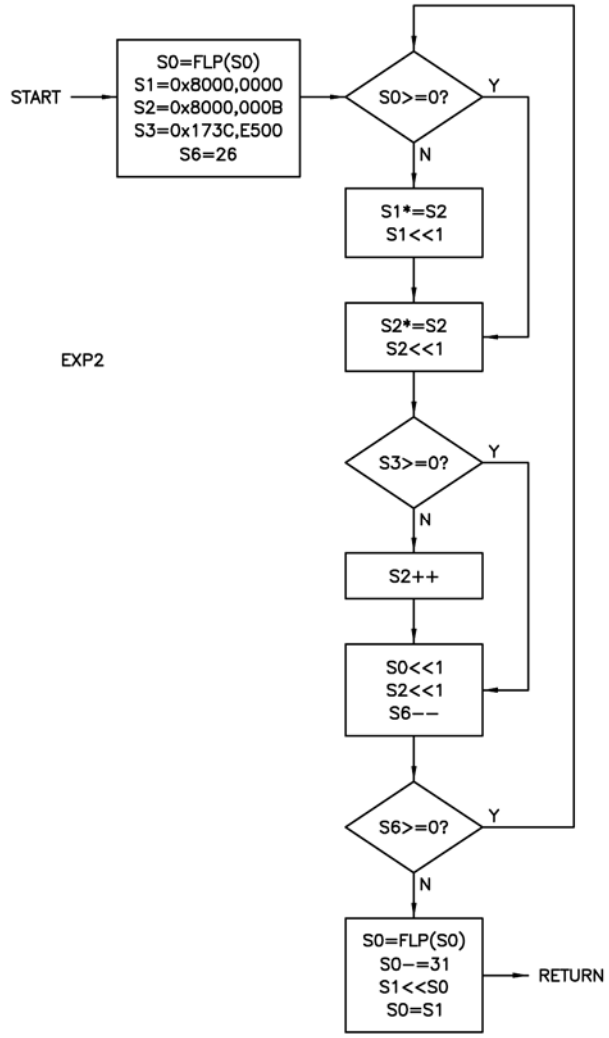

Figure 46. Exponential of 2 algorithm flow chart.

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, log2 gives 5 bits of characteristic and 32 - 5 = 27 bits of mantissa. If we use this fixed decimal form to "undo" the log2 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 via trial and error with an eye towards minimizing both local and final error. Without the fudge factor full scale output is off by approx. -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).

```
// sub : s0=exp2(s0), return to (s7)
//
// input is c[31:27].m[26:0] unsigned fixed decimal
// output is out[31:0] an unsigned 32 bit integer
//
// s0 : input, output
// s1 : running multiply
// s2 : running root
// s3 : fudge factor
// s4 :
// s5 :
// s6 : loop index
// s7 : sub return addr
//
// setup
i='h90;  ram[i] = { `flp,             `s0, `P0 };  // flp(s0) (to examine lsbs via msb)
i=i+1;   ram[i] = { `pus_i,         6'd31, `s1 };  // s1=0x8000,0000 (starting value = 1)
i=i+1;   ram[i] = { `cpy,             `s1, `s2 };  // s2=0x8000,000b (starting root = 2^2^-27)
i=i+1;   ram[i] = { `add_is,         6'hb, `P2 };  //
i=i+1;   ram[i] = { `lit,             `__, `s3 };  // s3=0x173c,e500 (fudge factor bits)
i=i+1;   ram[i] =                    16'he500  ;  //
i=i+1;   ram[i] =                    16'h173c  ;  //
i=i+1;   ram[i] = { `dat_is,        6'd26, `s6 };  // s6=26 (loop index)
// loop start
// jump 0 start
i=i+1;   ram[i] = { `jmp_inlz,       6'd2, `s0 };  // (s0[31]==0) ? jump +2 (skip running mult)
i=i+1;   ram[i] = { `mul_xu,          `s2, `P1 };  // s1*=s2
i=i+1;   ram[i] = { `shl_is,         6'd1, `P1 };  // s1<<=1 (so msb=1)
// jump 0 end
i=i+1;   ram[i] = { `mul_xu,          `s2, `P2 };  // s2*=s2 (square to get next root)
i=i+1;   ram[i] = { `shl_is,         6'd1, `P2 };  // s2<<=1 (so msb=1 & lsb=0)
// jump 1 start
i=i+1;   ram[i] = { `jmp_inlz,       6'd1, `s3 };  // (s3[31]==0) ? jump +1 (no fudge bit)
i=i+1;   ram[i] = { `add_is,         6'd1, `P2 };  // s2++ (set lsb of running root)
// jump 1 end
i=i+1;   ram[i] = { `shl_is,         6'd1, `P0 };  // s0<<=1 (get next input bit)
i=i+1;   ram[i] = { `shl_is,         6'd1, `P3 };  // s3<<=1 (get next fudge bit)
i=i+1;   ram[i] = { `add_is,        -6'd1, `P6 };  // s6-- (loop index--)
i=i+1;   ram[i] = { `jmp_inlz,     -6'd11, `s6 };  // (s6>=0) ? jump -11 (loop again)
// loop end
// final shift
i=i+1;   ram[i] = { `flp,             `s0, `P0 };  // flp(s0) (flip remaining bits)
i=i+1;   ram[i] = { `add_is,       -6'd31, `P0 };  // s0-=31
i=i+1;   ram[i] = { `shl_u,           `P0, `P1 };  // s1<<=s0, pop s0
// cleanup, return
i=i+1;   ram[i] = { `cpy,             `P1, `s0 };  // s0=s1, pop s1 (move)
i=i+1;   ram[i] = { `pop,          8'b01001100 };  // pop s2, s3, s6
i=i+1;   ram[i] = { `gto,             `P7, `__ };  // return, pop s7
// end sub
```

The subroutine code is above. The input is flipped to test the LSBs sequentially via the MSB (the “sign” bit). 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 left once to expose the next bits, the loop index is decremented, and the loop repeats until the loop index goes negative.

When the loop is exited, the remaining input bits are flipped back, -31 is added to them, 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 7 cycles to setup the loop, 8 cycles per loop best case and 11 cycles worst case, with 6 cycles after the loop. For 26 iterations with all being 11 worst case loops this gives:

7 + 11*26 + 6 = 299 cycles worst case

For a 200 MHz clock and 8 clocks per cycle, this is 11.96 us worst case.

“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 inverted 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 ~= total cycle frequency * alpha / Pi.

Mathematically this construct works because the integral of cosine = sine, and the integral of sine = -cosine. Numerically this construct works only if there is a single register delay in the loop, and given this condition truncation errors due to integrator input scaling rather mysteriously don’t 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 2 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, use the integer portion to nominally select the LED, and use 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 more 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:

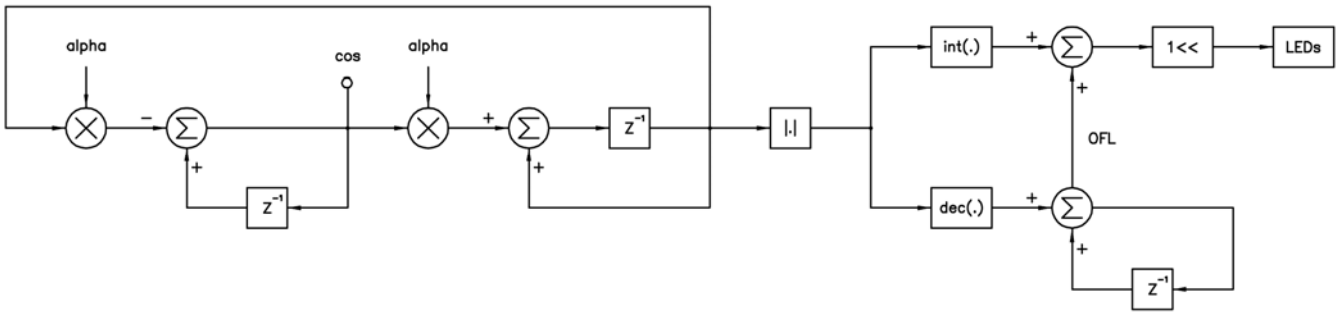

Figure 47. Bouncing ball block diagram.

In terms of 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’m using has 4 LEDs, which correspond to the values 0, 1, 2, and 3, so if we set the most significant hex digit to be our integer portion, then the initialization value should be 0x30000000.

```
// "bouncing ball" 4 LED display w/ PWM
//
// s0 : sin
// s1 : cos
// s2 : alpha (attenuation factor = speed)
// s3 : rectified sin, val, one-hot(val)
// s4 :
// s5 : pwm counter
// s6 :
// s7 :

i='h100; ram[i] = { `dat_is,          6'd0, `s0 };  // s0=0  (sin init)
i=i+1;   ram[i] = { `lit,              `__, `s1 };  // s1=0x3000,0000 (cos init)
i=i+1;   ram[i] =                      16'h0000  ;  //
i=i+1;   ram[i] =                      16'h3000  ;  //
i=i+1;   ram[i] = { `lit_u,            `__, `s2 };  // s2=0x3000 (alpha)
i=i+1;   ram[i] =                      16'h3000  ;  //
i=i+1;   ram[i] = { `dat_is,          6'd0, `s5 };  // s5=0  (pwm init)
// loop start
// sin & cos
i=i+1;   ram[i] = { `mul_xs,           `s2, `s0 };  // s0=s0*s2 (sin*alpha)
i=i+1;   ram[i] = { `sub,              `P0, `P1 };  // s1-=s0 (cos-=sin*alpha)
i=i+1;   ram[i] = { `mul_xs,           `s2, `s1 };  // s1=s1*s2 (cos*alpha)
i=i+1;   ram[i] = { `add,              `P1, `P0 };  // s0+=s1 (sin+=cos*alpha)
// |sin|
i=i+1;   ram[i] = { `cpy,              `s0, `s3 };  // s3=s0
i=i+1;   ram[i] = { `jmp_inlz,        6'd1, `s3 };  // (s3!<0) ? jmp +1
i=i+1;   ram[i] = { `not,              `s3, `P3 };  // s3~=s3
// decimal( |sin| ) + pwm to update, + pwm to get ofl
i=i+1;   ram[i] = { `shl_is,          6'd4, `s3 };  // s3=s3<<4
i=i+1;   ram[i] = { `add,              `s3, `P5 };  // s5+=s3 (update pwm count)
i=i+1;   ram[i] = { `add_xu,           `P3, `s5 };  // s5=s5+s3, pop s3 (get pwm ofl)
// one-hot( int( |sin| ) + pwm ofl )
i=i+1;   ram[i] = { `shl_is,        -6'd28, `P3 };  // s3>>=28
i=i+1;   ram[i] = { `add,              `P5, `P3 };  // s3+=s5, pop s5 (add pwm ofl)
i=i+1;   ram[i] = { `pow,              `s3, `P3 };  // s3=1<<s3, pop s3 (one-hot)
// output
i=i+1;   ram[i] = { `not,              `s3, `P3 };  // s3~=s3 for dark spot
i=i+1;   ram[i] = { `reg_iw,           `IO, `P3 };  // reg(IO)=s3, pop s3
i=i+1;   ram[i] = { `jmp_inz,       -6'd16, `s2 };  // loop forever
// loop end
```

Code is above. 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 3. 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. Also, 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.

Generic Boot Code

If one has a UART level shifter attached to one of the FPGA inputs (or the USB equivalent), the boot code may consist of a simple boot loader capable of uploading and storing executable code, and the boot code itself wouldn't 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 device tacked onto a few spare FPGA pins could hold gobs of uploaded programming goodness.

BZZZ!

I would be remiss if I didn't point out the less positive aspects of Hive that I'm aware of:

- The instruction set of Hive was hatched more intuitively than scientifically by a person who doesn't exactly have loads of practical experience programming assembly (that probably just scared off most readers). I've put more time into selecting operations, sizing immediate fields, and shuffling things around in the opcode encoding space than I've spent actually programming Hive (at this point).
- Common data & instruction memory space (Von Neumann architecture) enables many good things, but it generally prevents 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 them all. (I should point out that the "Von Neumann bottleneck" is not 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.

ETC.

- 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 (it is ironically nearly impossible 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 sadly past the end of support), and converted to PDF with Adobe Acrobat 8 Professional (which is 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 run slower – Altera apparently uses faster, and consequently leakier and more power hungry, transistors (it's a two-edged sword).
- I wrote a MS Excel VBA based simulator that allows the user to easily enter and exercise code (but only for a single thread).

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).

DOCUMENT CHANGE CONTROL

(YYYY-MM-DD)	Hive version	Notes
2014-07-15	06.01	Updates to reflect 32 bit memory access, new / changed opcodes, 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.

COPYRIGHT

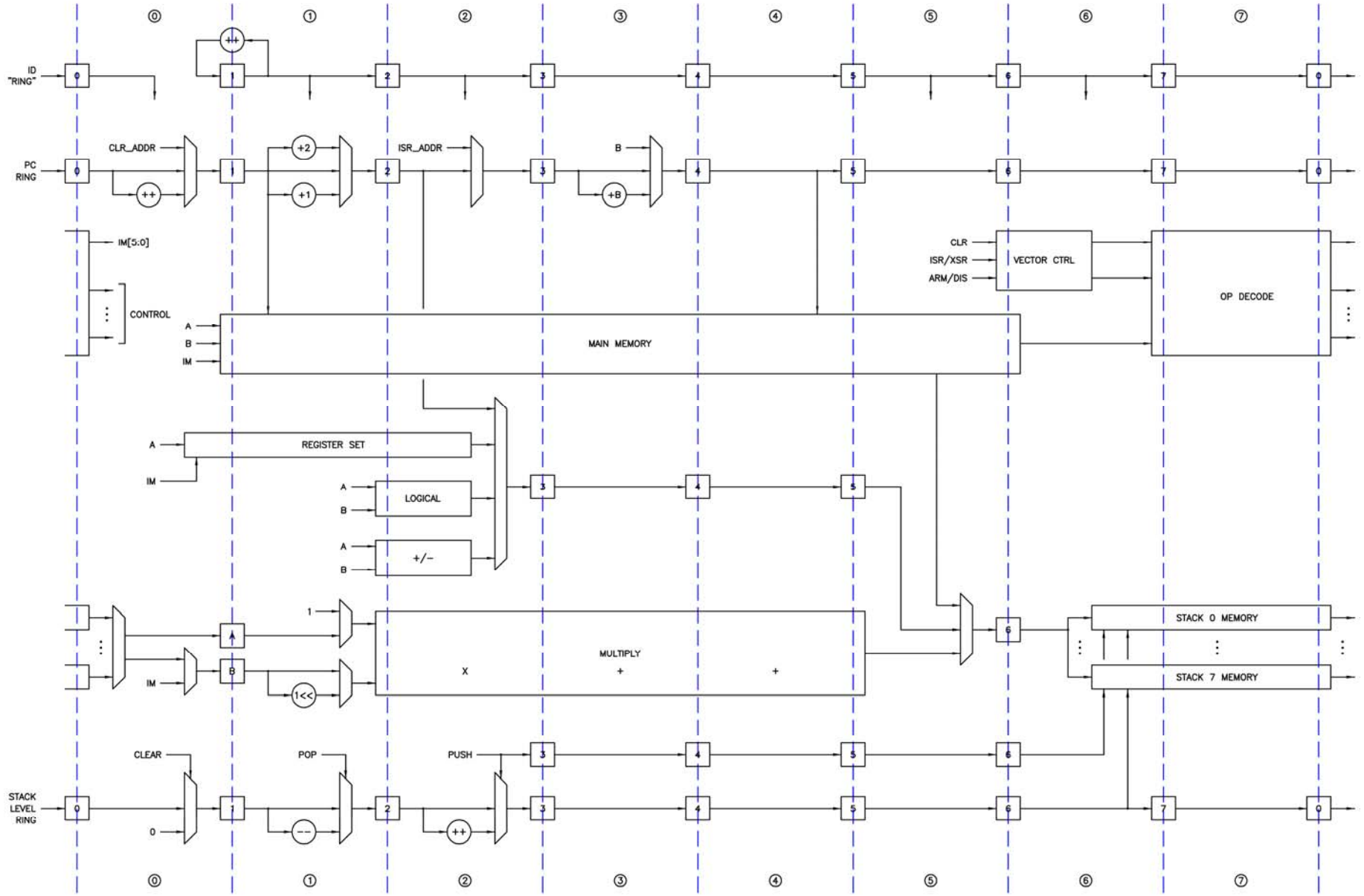

Figure 48. Hive core.

```
// misc - 16 x 16 x 16 = 4096 codes
op_nop       = { `nop,    `bx,   `ax },// do nothing (no pops either)
op_pop       = { `pop,    `popx      },// pop[7:0] none/one/some/all stacks
op_pcp       = { `pcp,    `bx,   `ax },// A:=PC  read PC++ (unsigned)
op_lit       = { `lit,    `bx,   `ax },// A:=mem(PC)  literal data
op_lit_ls    = { `lit_ls, `bx,   `ax },// A:=$signed(mem(PC)[lo])  literal data low signed
op_lit_lu    = { `lit_lu, `bx,   `ax },// A:=mem(PC)[lo]  literal data low unsigned
op_reg_ir    = { `reg_ir, `im6x, `ax },// A:=reg(I)  register immediate read
op_reg_iw    = { `reg_iw, `im6x, `ax },// reg(I):=A  register immediate write
// logical & other - 16 x 16 x 16 = 4096 codes
op_cpy       = { `cpy,    `bx,   `ax },// A:=B  copy
op_isg       = { `isg,    `bx,   `ax },// A[MSB]:=~B[MSB]  invert sign
op_cpy_ls    = { `cpy_ls, `bx,   `ax },// A:=$signed(B[lo])  low sign extend
op_cpy_lu    = { `cpy_lu, `bx,   `ax },// A:=B[lo]  low zero extend
op_not       = { `not,    `bx,   `ax },// A:=~B  logical NOT
op_and       = { `and,    `bx,   `ax },// A:=A&B  logical AND
op_orr       = { `orr,    `bx,   `ax },// A:=A|B  logical OR
op_xor       = { `xor,    `bx,   `ax },// A:=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:=flip(B)  flip bits end for end
op_lzc       = { `lzc,    `bx,   `ax },// A:=lzc(B)  leading zero count
// arithmetic - 16 x 16 x 16 = 4096 codes
op_add       = { `add,    `bx,   `ax },// A:=A+B  add
op_add_xs    = { `add_xs, `bx,   `ax },// A:=A+B  add extended signed
op_add_xu    = { `add_xu, `bx,   `ax },// A:=A+B  add extended unsigned
op_sub       = { `sub,    `bx,   `ax },// A:=A-B  subtract
op_sub_xs    = { `sub_xs, `bx,   `ax },// A:=A-B  subtract extended signed
op_sub_xu    = { `sub_xu, `bx,   `ax },// A:=A-B  subtract extended unsigned
op_mul       = { `mul,    `bx,   `ax },// A:=A*B  multiply
op_mul_xs    = { `mul_xs, `bx,   `ax },// A:=A*B  multiply extended signed
op_mul_xu    = { `mul_xu, `bx,   `ax },// A:=A*B  multiply extended unsigned
op_shl_s     = { `shl_s,  `bx,   `ax },// A:=A<<<B  shift left A signed
op_shl_u     = { `shl_u,  `bx,   `ax },// A:=A<<B  shift left A unsigned
op_pow       = { `pow,    `bx,   `ax },// A:=1<<B  power of 2
// branching - 16 x 16 x 16 = 4096 codes
op_jmp_z     = { `jmp_z,  `bx,   `ax },// PC:=(A?0)?PC+B  jump zero conditional
op_jmp_nz    = { `jmp_nz, `bx,   `ax },
op_jmp_lz    = { `jmp_lz, `bx,   `ax },
op_jmp_nlz   = { `jmp_nlz,`bx,   `ax },
op_jmp       = { `jmp,    `bx,   `ax },// PC:=PC+B  jump unconditional
op_gto       = { `gto,    `bx,   `ax },// PC:=B  go to unconditional
op_rtn       = { `rtn,    `bx,   `ax },// PC:=B  ISR return (go to unconditional)
op_gsb       = { `gsb,    `bx,   `ax },// PC:=B, A=PC  subroutine call unconditional
// immediate memory access - 4 x 16 x 16 x 16 = 16384 codes
op_mem_ir    = { `mem_ir,   `im4x, `bx, `ax },  // A:=mem(B+I)  memory read
op_mem_irls  = { `mem_irls, `im4x, `bx, `ax },  // A:=$signed(mem(B+I)[lo])  memory read low signed
op_mem_iw    = { `mem_iw,   `im4x, `bx, `ax },  // mem(B+I):=A  memory write
op_mem_iwl   = { `mem_iwl,  `im4x, `bx, `ax },  // mem(B+I):=A[lo]  memory write low
// immediate conditional (A?B) jumps - 6 x 16 x 16 x 16 = 24576 codes
op_jmp_ie    = { `jmp_ie,   `im4x, `bx, `ax },  // PC:=(A?B)?PC+I  jump immediate conditional
op_jmp_ine   = { `jmp_ine,  `im4x, `bx, `ax },
op_jmp_ils   = { `jmp_ils,  `im4x, `bx, `ax },
op_jmp_inls  = { `jmp_inls, `im4x, `bx, `ax },
op_jmp_ilu   = { `jmp_ilu,  `im4x, `bx, `ax },
op_jmp_inlu  = { `jmp_inlu, `im4x, `bx, `ax },
// immediate conditional (A?0) jumps - 4 x 64 x 16 = 4096 codes
op_jmp_iz    = { `jmp_iz,   `im6x, `ax },   // PC:=(A?0)?PC+I  jump immediate conditional
op_jmp_inz   = { `jmp_inz,  `im6x, `ax },
op_jmp_ilz   = { `jmp_ilz,  `im6x, `ax },
op_jmp_inlz  = { `jmp_inlz, `im6x, `ax },
// immediate data - 1 x 64 x 16 = 1024 codes
op_dat_is    = { `dat_is,   `im6x, `ax },   // A:=I  data immediate signed
// immediate add - 1 x 64 x 16 = 1024 codes
op_add_is    = { `add_is,   `im6x, `ax },   // A:=A+I  add immediate signed
// immediate shifts - 2 x 64 x 16 = 2048 codes
op_shl_is    = { `shl_is,   `im6x, `ax },   // A:=A<<<I  shift left A signed
op_pus_i     = { `pus_i,    `im6x, `ax }    // A:=1<<I  power of 2; A=A<<I  shift A unsigned
```

Figure 49. Hive opcodes.