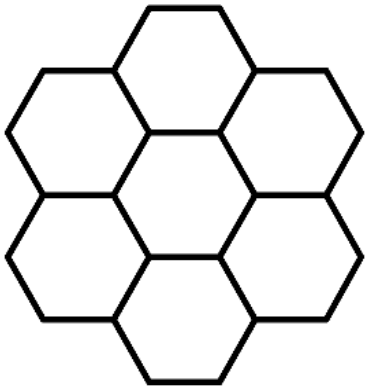

HIVE

32 Bit – 8 Thread – 4 Register/Stack Hybrid – Barrel Pipelined
Verilog 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 good 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 a single instance of code, subroutines, or 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.

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 overhead and interrupts consume a single branch cycle, and their calculations can be performed directly and immediately with 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 with extended width arithmetic results.
- 16 bit addresses, instructions, and memory data accesses.
- 8 equal bandwidth threads.
- 4 independent, isolated, general purpose LIFO data stacks per thread with parameterized depth and fault protections.
- 8 fully independent 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.
- Internal register set with highly configurable base register module that may be easily modified / expanded to enhance I/O, debug, etc.
- 32 bit GPIO with atomic full-width read.
- Written in 100% highly portable verilog (no vendor specific or proprietary language constructs).
- Intelligently partitioned into easy to understand and verify verilog modules.
- May be programmed via a verilog 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 only ~1800 logic elements, or ~34% 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 always on the lookout for simple processor cores because projects often underutilize the hardware due to low data rates (think of a UART, or a sampled audio stream). If latency isn't a big issue, then why not multiplex the 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 / 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 with conservative top-entry-only access and pop bit override) is something new. And the way extended arithmetic results are dealt with uniformly in Hive may possibly be 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 really difficult 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 huge advantage of being a very mature target for compilers.

Many registers are generally available because the register space grows exponentially with register address width. But register opcode indexes can 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 moot, or if the register contents are first saved to memory and later restored, which is something else to keep track of.

Almost 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 and 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 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, or at least wasted program space) or the hardware must somehow store and retrieve a sub index into the return word (more state).

Stack machines are, perhaps somewhat inadvertently, portrayed 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 one side of the case.

Something 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, it isn't always emphasized that conditional branches generally consume the branch test value(s) and the branch address regardless of whether the branch is taken or not. Auto consumption is a big 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.

BACKGROUND: LIFOS

Since the discussion is about stack machines, it helps to fully understand stacks themselves, which are based on the LIFO (**L**ast **I**n **F**irst **O**ut) construct.

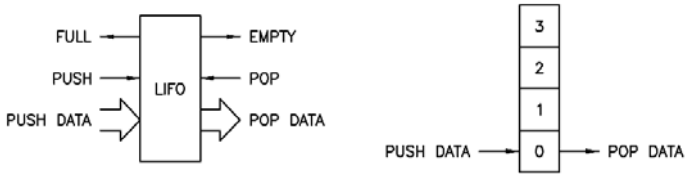

Figure 1. LIFO symbols.

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 it may drop the input data on the floor and may corrupt the LIFO pointer. Pop when empty is an error because it gives false read data and also may corrupt the LIFO pointer.

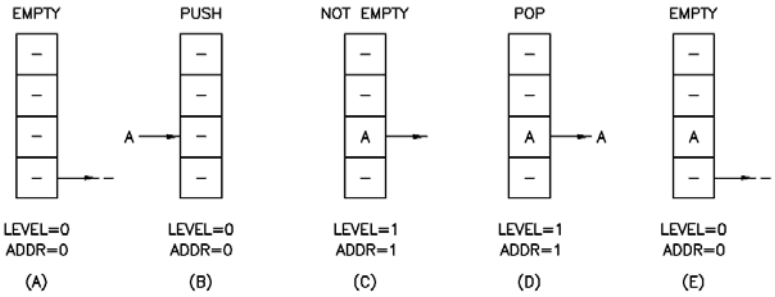

Figure 2. LIFO stack operations – push then pop from empty state.

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.

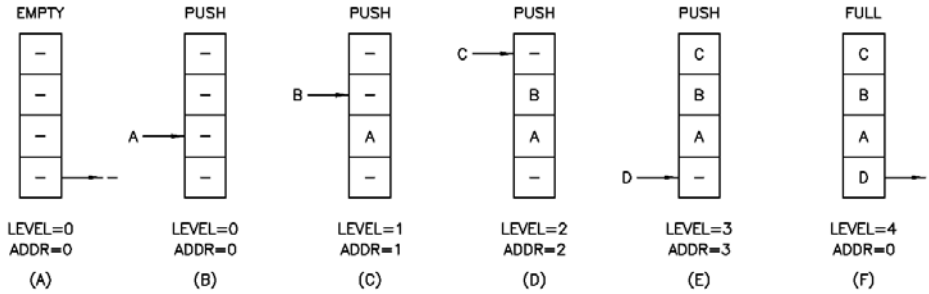

Figure 3. LIFO stack operations – push from empty state to full state.

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 (one MSB less than) 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).

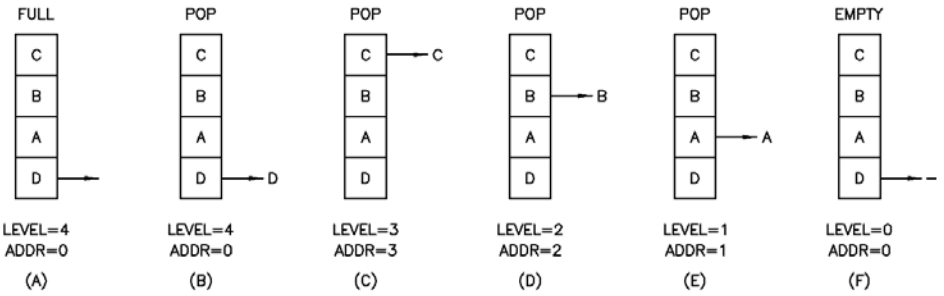

Figure 4. LIFO stack operations – pop from full state to empty state.

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.

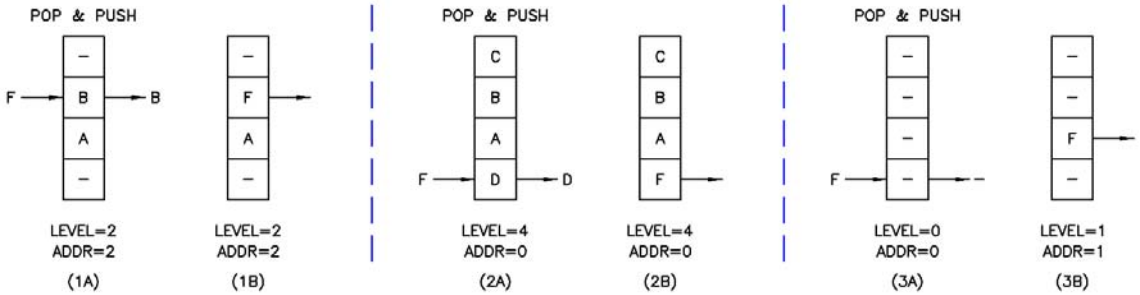

Figure 5. LIFO stack operations – three pop & push scenarios.

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 stack 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 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, but beyond that is there some kind of 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 could easily scale as required by other aspects of the design. Could the stacks indeed be addressed 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 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 both scenarios. 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 suddenly 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: not B => A.
- For two input ALU operations the primary source is A, the secondary source is B, and the result destination is A. For example: A - B => A.
- For single input conditional branch statements A is tested against zero, and the absolute address is B, or the address increment is B or supplied as immediate data. For example: (A>0) ? B => PC.
- For two input conditional branch statements, A is tested against B, and the address increment is supplied as immediate data. For example: (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: mem(B+I) => A.
- 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: A => mem(B+I).
- 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 PC) is automatically pushed to stack S0 (this is the only "special" stack, and this is the only way in which it is "special").
- For operations that exist in both immediate and non-immediate form, B data is simply ignored for the immediate form.
- For operations that use the '1' input on A, the A data is simply ignored.

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

How many stacks are needed? I picked 4. This gives 3 bits for each operand (pop bit, two stack index bits) for a total of 6 bits of opcode consumed. If the opcode is 16 bits total, this leaves 10 bits, or 1024 possible opcodes. 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 4 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).

DATA WIDTH

Hive data is 32 bits wide. Byte data is too narrow for most 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. Obviously data width directly dictates the top speed vs. pipelining depth because wider data requires more deeply cascaded combinatorial logic to perform adds, multiplies, etc.

ACCESS WIDTH

Hive access width, and by that I mean main (data and program) memory read / write – which includes instruction fetch and in-line literal access – is 16 bits. One generally desires everything to be the same width, but an intentionally compacted opcode can easily wind up narrower than the ALU data path. And much can be done with 16 bit data which can directly lead to code compaction.

ADDRESS WIDTH

Hive addresses are limited to 16 bits, giving an address space of 65k 16 bit words.

One can agonize over the small address space and the narrow access, but the target functionality here is a small processor in an FPGA, so large address space isn't necessary, and beyond a certain depth memories are slower. (At any rate, I believe only trivial edits would be required to provide Hive with a full 32 bit address.)

ARITHMETIC RESULTS WIDTH

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, saving, and restoring of them. The results of full width multiplies are usually sent to special concatenated register pairs. These practices 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 the instruction set. 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 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. 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

Hive arithmetic is signed by default. 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 signed half-width read and in-line literal data is more useful because it can influence 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.

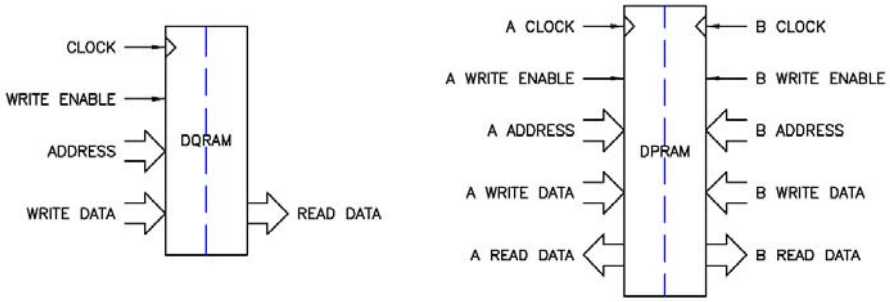

Figure 9. Block RAM: simple (DQ) on left, true dual port (DP) on right.

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.

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 width of the ALU or instructions, as this precludes the efficient use of conventional 2^n width memory to store internal data / control information.

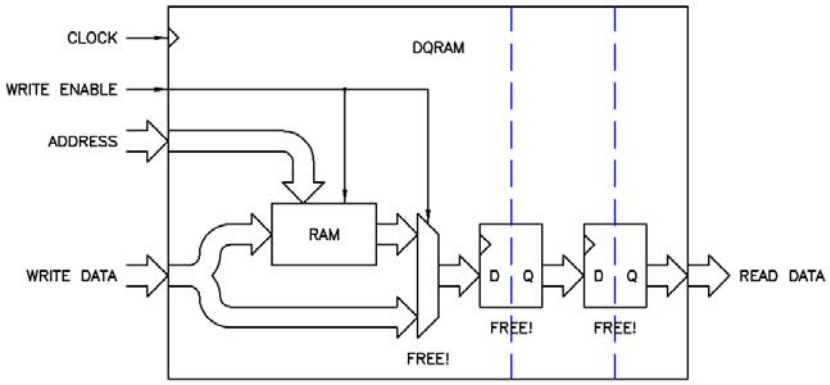

Figure 10. Block RAM internal resources.

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 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. With the multiplexer, a read-during-write conveys the data being written to the read data port. The flip flop following this optional multiplexer is *always* present. Following this is yet another flip flop; 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 is often capable of near fabric flip flop performance, while the read side is rather slow if the additional output register isn't used. So a certain amount of combinatorial logic can be placed directly in front of the write side, and 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.

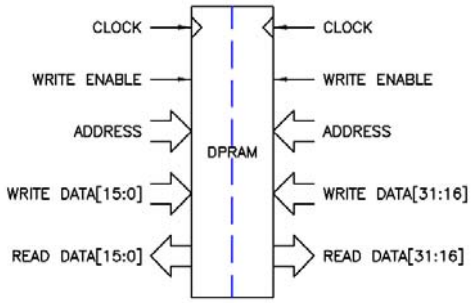

Figure 11. True dual port block RAM utilized as DQ RAM.

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.

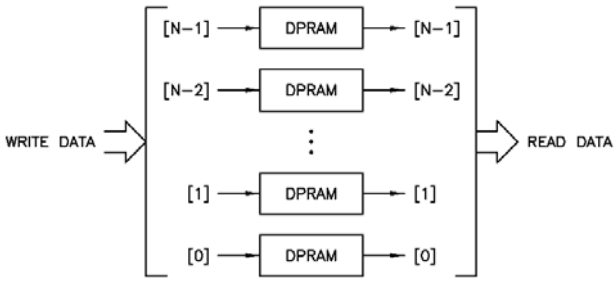

Figure 12. Block RAMs combined via bit-slice.

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 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.

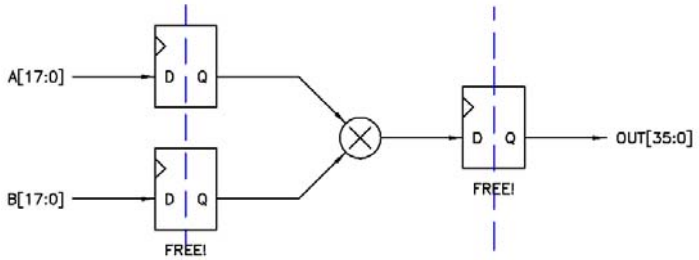

Figure 13. Signed multiplier hardware typically found in an FPGA.

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 do 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).

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. But this leads one inexorably to the issue of 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 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 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 is *the* component that takes the longest to fully develop.

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.

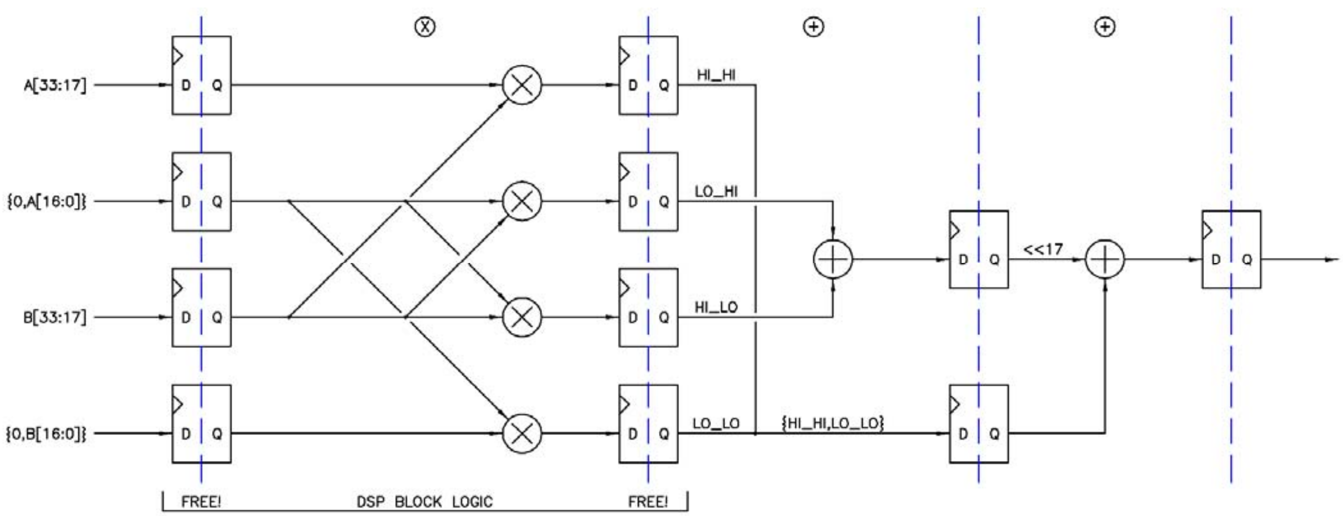

Figure 15. Three stage 33 x 33 = 66 bit signed pipelined multiplication.

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 considerable amount of FPGA fabric logic to shift a number to the right and left some arbitrary distance and the result isn't very speedy. Having a multiplier just sitting there doing nothing useful during the shift is a missed opportunity.

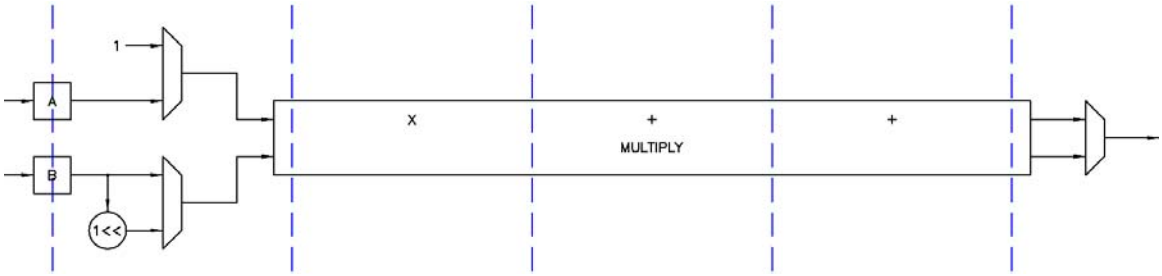

Figure 16. The Multiply and Shift unit.

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 ones 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 resulting 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.) 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 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.

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 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, Out=0100`
OR	A | B	`A=1100, B=0101, Out=1101`
XOR	A ^ B	`A=1100, B=0101, Out=1001`
NOT	~B	`A=xxxx, B=0101, Out=1010`
AND_B	&B	`A=xxxx, B=0101, Out=0000` `A=xxxx, B=1111, Out=1111`
OR_B	|B	`A=xxxx, B=0101, Out=1111` `A=xxxx, B=0000, Out=0000`
XOR_B	^B	`A=xxxx, B=0101, Out=0000` `A=xxxx, B=0111, Out=1111`

Figure 19. Logical functions as implemented (examples here limited to 4 bits).

Note that "_B" 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.

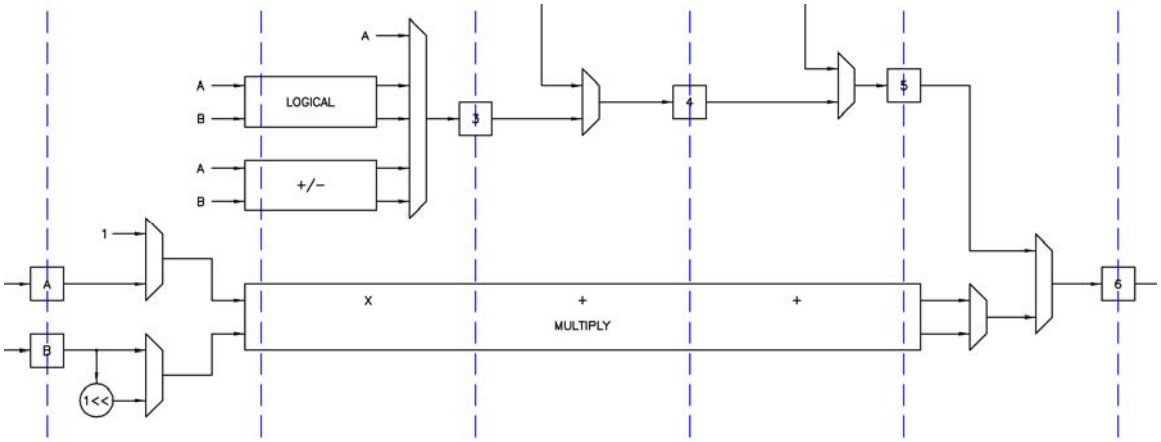

Figure 20. The Arithmetic and Logic Unit (ALU).

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 3 and 4 for reading and subroutine / interrupt return address use. Read and literal data from main memory and the local register set 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

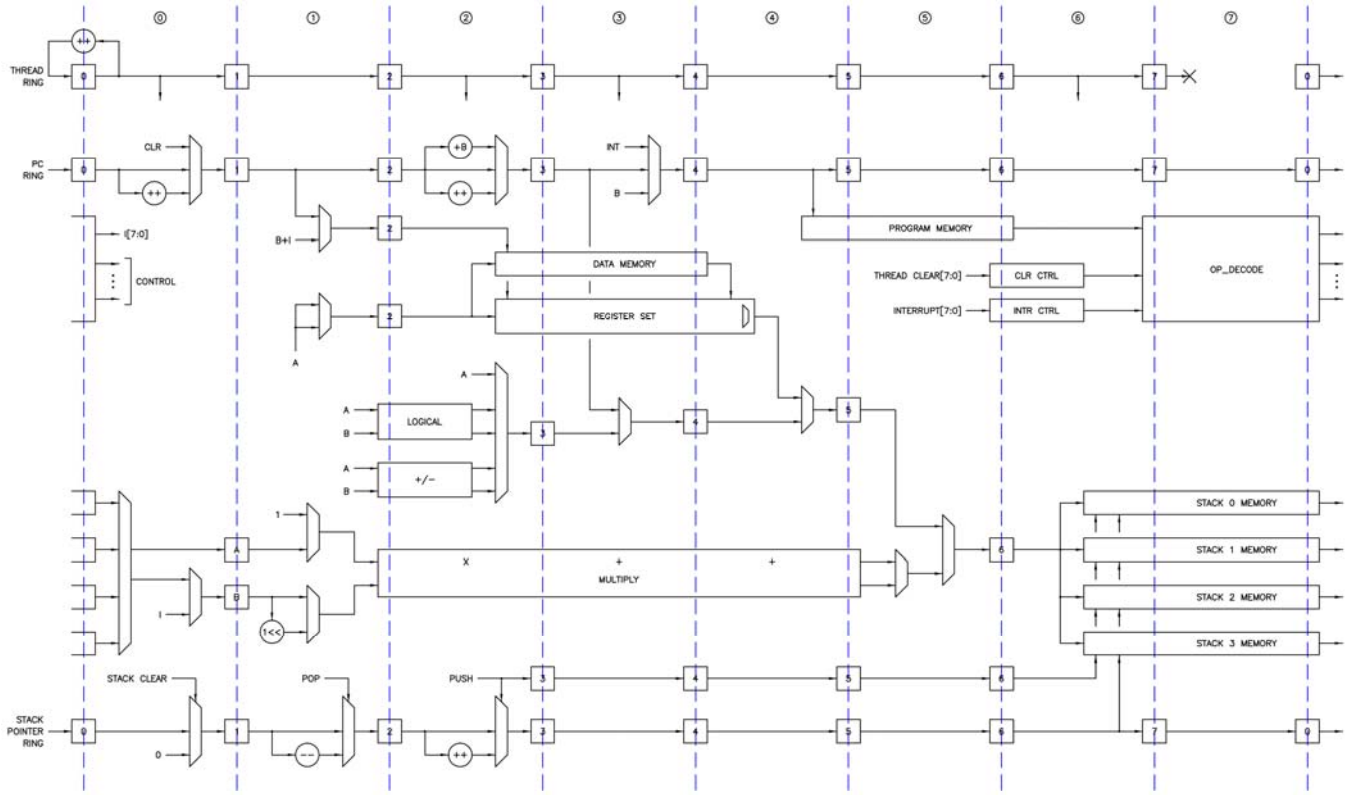

Figure 21. Hive core – view from 100 feet up.

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 (data and address) space which facilitates intercommunication between them as well as code compaction / factoring (the sharing of common data and subroutines). 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 but 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.

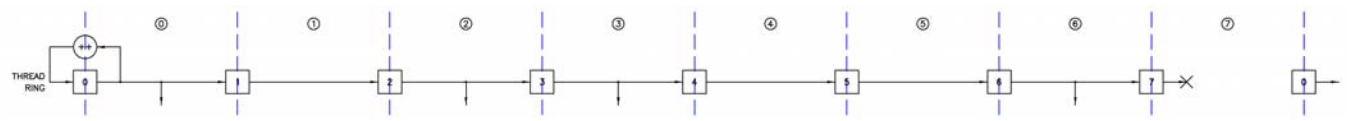

Figure 22. The Thread ID "Ring".

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.

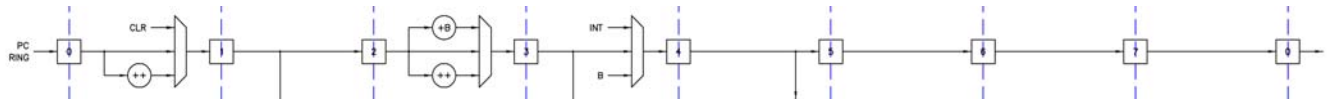

Figure 23. The Program Counter Ring.

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. In stage 2 the PC is incremented by 1 if retrieving an in-literal (to get the next instruction), incremented by B (or an immediate value) if taking a relative jump, or left alone. In stage 3 the PC is sent to the data path for reading, or as a return address if taking a subroutine or interrupt. Also in stage 3 the PC is replaced with the thread interrupt address if taking an interrupt, or 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.

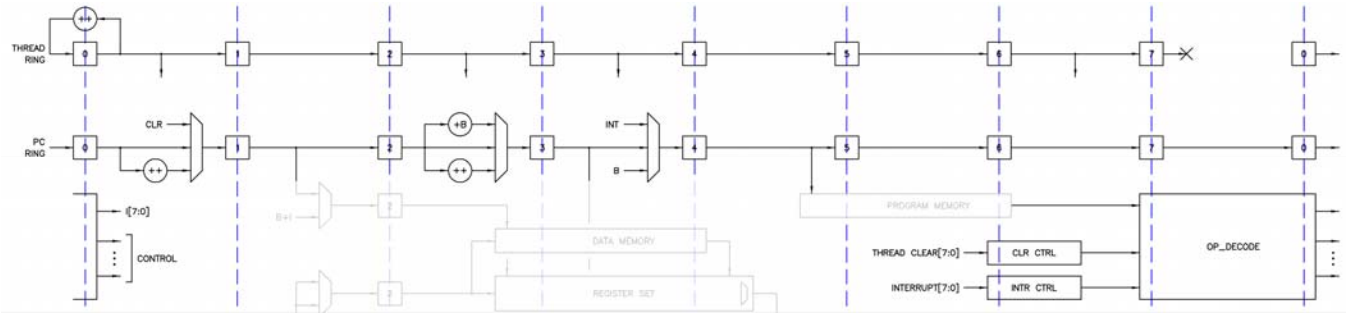

Figure 24. The Control Ring.

The thread ring and PC ring, together with the opcode decoding unit and the clear and interrupt event controllers, 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 clear and interrupt event controllers use 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 interrupt 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.

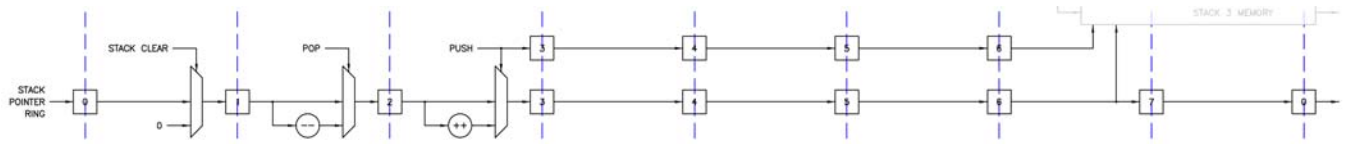

Figure 25. The Stack Pointer Ring.

Shown above is the stack pointer ring. In stage 0 the stack pointers are cleared if the thread is being cleared or if a stack clear instruction is decoded. In stage 1 valid pop events decrement the relevant stack pointer(s). In stage 2 valid push events increment the relevant stack pointer. Not shown in stages 1 and 2 is logic that measures fullness and prevents push when full / pop when empty from corrupting the stack pointers (if so configured at build time). These error events are reported to the local register set for debugging purposes. Separating the 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. In stage 6 the stack pointers 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 makes 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.

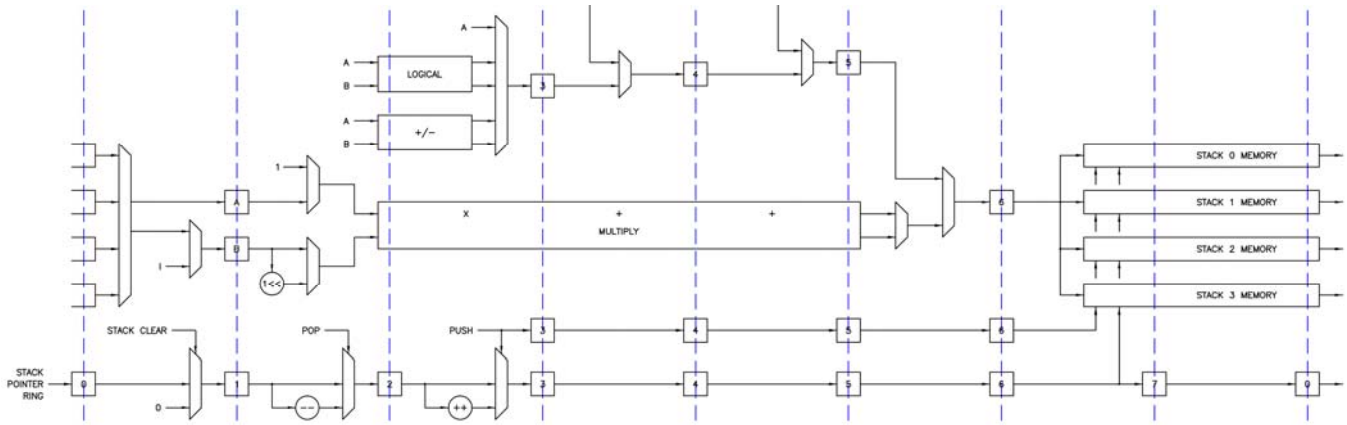

Figure 26. The Data Ring.

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, local register set, and register set shim (not shown) make up the Hive core. The shim is a simple data & address arbiter, which allows the register set to share the same bus as the main memory data port. The shim also arbitrates between in-line literal and data accesses.

INSTRUCTIONS / OPCODES

With the basic hardware structure in place we can now decide on the instructions and opcode formats. In actuality the design process isn't nearly this cut and dried, the inclusion and format of certain instructions will ripple back into the hardware structure and vice versa leading to all sorts of exciting, fun-filled churn.

Selecting a balanced set of instructions and determining how to best fit them into the opcode space can be a formidable challenge, particularly for a processor where the opcode is intentionally compact. From the previous discussion, we know there are 2 stack indexes of 2 bits each, with 1 pop bit for each index. This consumes 6 bits of opcode space, leaving 10 bits remaining. Some designs utilize the room freed up when fewer operands are required for a particular operation, and I decided not to go that route in order to maximize the opportunities for concurrent stack cleanup (popping of unneeded data / addresses).

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 need not be fixed, and I decided to implement immediate 8 and 6 bit signed data instructions, as well as immediate 5 bit signed and 4 bit unsigned address offset instructions, all with the immediate LSB positioned at bit 6 for consistent alignment. A 6 bit wide immediate value when coupled to the shift instruction is useful because it allows for full 32 bit left or right shifting in a single cycle, and one or more shifts can perform many chores that would otherwise require dedicated instructions and hardware (full width MSB flag, arbitrary width sign / zero extension, isolation of arbitrary contiguous bit fields, etc.). The 4 bit immediate is used exclusively as a read / write address offset, so I felt it best be unsigned.

Immediate instruction types and opcode space consumption:

- Four 4 bit immediate address instructions: read and write, both with extended mode – 64 codes.
- One 5 bit immediate address instruction: conditional jump – 416 codes.
- One 8 bit immediate data instruction: byte (signed) – 256 codes.
- Three 6 bit immediate data instructions: shift left (signed), shift left unsigned, add (signed) – 192 codes.

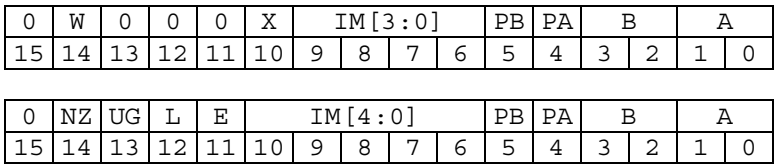

Figure 27. Immediate address instruction formats (top to bottom): immediate read & write; immediate conditional jump.

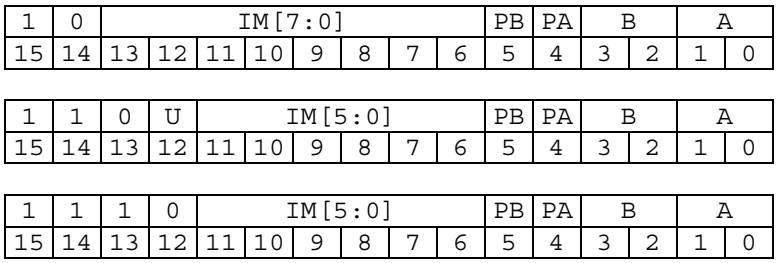

Figure 28. Immediate data instruction formats (top to bottom): immediate literal data; immediate shift; immediate add.

1	1	1	1	0	0	OP	G	L	E	PB	PA	B		A	
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

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

Figure 29. Other instruction formats (top to bottom): conditional branch; single.

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 immediate byte instruction is one way to insert data from the instruction stream, another is via three literal instructions that employ an in-line mechanism (the value immediately follows the literal instruction in program space). The byte instruction obviously uses 16 bits and one cycle to push 8 signed bits of data. The in-line literal instructions use 32 bits (the 16 bit literal instruction followed by 16 bits of data – the data is used "literally" rather than decoded) but still only one cycle to push 16 bits of data. There are signed and unsigned literal low instructions, and an extended instruction that preserves the lower 16 bit value to fill 32 data bits in two cycles.

There are 4 instructions for data space access: two for read and two for write. All have an unsigned 4 bit immediate field in order to provide a group of 16 convenient memory slots off of a base address. The low read is signed, and there is no corresponding unsigned read instruction. The extended read (like the extended in-line literal) preserves the lower 16 bit value to load all 32 bits in two cycles. Write and write extended instructions are provided to store 32 data bits in two cycles. 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.

Add, subtract, multiply, shift, and all of the logical operations have been described previously. An immediate signed add is provided for small quick increments and decrements (+31/-32). There are both immediate and non-immediate forms of the shifts because one often needs to shift by a variable, rather than by a constant.

Branches
There are three types of branches – jump, go to, and subroutine:

- **JMP** (jump) is relative to the PC and is either conditional or unconditional, and either immediate or not. The conditional form jumps a signed distance given by B (or I for the immediate version) if the test (A?0) is true. There are also conditional (A?B) signed and unsigned tests for the immediate JMP. The immediate JMP is deemed (perhaps rashly) so valuable that it consumes almost half of the opcode space.
- **GTO** (go to) is absolute, and is either conditional or unconditional. The conditional form loads the PC with the value given by B if the conditional test (A?0) is true.
- **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.

Relative branches are 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.

Note that there is no explicit *return* operation, an unconditional GTO is used to return from subroutines and interrupts. The return address can be simultaneously popped at this point as well.

Conditional Field

G	L	E	Condition	(A?0)	(A?B)
0	0	0	Never (useless)	0	0
0	0	1	**E**qual	A==0	A==B
0	1	0	**L**ess	A<0	A<B
0	1	1	**L**ess OR **E**qual	A<=0	A<=B
1	0	0	**G**reater	A>0	A>B
1	0	1	**G**reater OR **E**qual	A>=0	A>=B
1	1	0	**G**reater OR **L**ess (not equal)	A<>0	A<>B
1	1	1	Always (unconditional)	1	1

Figure 30. The conditional binary field.

As seen in the above table, the conditional bits form a convenient binary field when placed adjacent to one another and functionally ORed together. GLE=111 is always execute, which is the unconditional case. GLE=000 is wasted space, particularly so for conditional instructions that have immediate fields.

Some conditional sign conventions / observations:

- All conditional comparisons of A to zero treat A as signed.
- All conditional comparisons of A and B that aren't explicitly unsigned treat both A and B as signed.
- All conditional comparisons of A and B that are unsigned treat both A and B as unsigned.
- The equivalency comparisons E and GL (A==0), (A==B), (A<>0), and (A<>B) are obviously sign neutral.

Other Instructions
There are several stack and miscellaneous instructions:

- **PC** pushes the current PC (pointing to the next instruction) to A.
- **CPY** copies the contents of B to A (use with pop bits to form a move, etc.).
- **POP** is a NOP that enables pops (the usual A and/or B) for stack cleanup.
- **NOP** is a do nothing instruction, all functionality including pops is disabled.
- **CLS** clears the stack pointers of this thread.

One of course endeavors to maintain a strict global stack inventory, but it's often difficult to keep track of garbage on the stacks, which can easily lead to catastrophic stack faults. Clearing the stacks now and then can be a good thing. Note the almost complete absence of the usual swap, roll, pick, etc. stack operations normally associated with pure stack machines.

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 verilog reserved words. After this is the two or three letter operation, followed often (but not necessarily) by a second underscore and one or more option letters. Obviously not all operations can use all of the options.

Op_*	Function
byt	**BYT**e data : I=>A
rd	**R**ea**D** : mem[B+I]=>A
wr	**Wr**ite : A=>mem[B+I]
jmp	**J**u**MP** : PC+(B or I)=>PC
gto	**G**o **TO** : B=>PC
gsb	**G**o **S**u**B**routine : B=>PC, PC=>A
lit	In-line **LIT**eral data : mem[PC]=>A
pc	**PC** as data : PC=>A
cpy	**C**o**PY** : B=>A
and	Logical **AND** : A&B=>A
or	Logical **OR** : A|B=>A
xor	Logical **XOR** : A^B=>A
not	Logical **NOT** : ~A=>A
add	Arithmetic **ADD**ition : A+B=>A
sub	Arithmetic **SUB**traction : A-B=>A
mul	Arithmetic **MUL**tiplication : A*B=>A
shl	**Sh**ift **L**eft : A<<(B or I)=>A
cls	**Cl**ear **S**tacks (for this thread only)
pop	NOP with **POP**s enabled
nop	**N**o **Op**eration (no pops either)

Figure 31. Instruction operations.

Op_*_?	Function
i	**I**mmediate
u	**U**nsigned (default is signed)
x	e**X**tended
g	**G**reater than (A>B)?
l	**L**ess than (A<B)?
e	**E**qual to (A==B)?
z	**Z**ero
b	**B**it reduction

Figure 32. Operation options.

The rules for these options are:

- No underscore between options.
- The immediate option comes first.
- The unsigned option comes next. Example: op_shl_iu (shift left immediate unsigned).
- The conditional options g, l, e, and z come next, and in that order. Example: op_jmp_igez (jump immediate greater than or equal to zero).
- The extended option goes last. Example: op_mul_ux (multiply unsigned extended).
- Some operations exist only in an immediate form (byt, rd, wr) and the rule here is to always include the immediate option regardless. Example: op_byt_i (byte immediate).

The conditional field bits (G, L, E) are used with the modifier Z (zero) if the default comparison to B is not desired. Some examples: GE is true if (A>=B); LZ is true if (A<0); GLZ is true if (A<>0) or equivalently (A!=0). GLE and GLEZ are not actually referred to this way because they are equivalent to *always execute*, which is simply the absence of the conditional options in the opcode name.

If a desired (A?B) comparison doesn't exist, pick the logical opposite and switch the A and B stack targets in the instruction. For example, say you want {op_jmp_iule, -8, _, P, s0, s1} but you look at the opcode list and no dice. But the equivalent {op_jmp_iuge, -8, P, _, s1, s0} does exist and is an exact functional replacement.

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, which is much easier to say than to do. I use a spreadsheet to keep track of them, with a column for each control signal that is an output of the opcode decoder. This helps to reveal similar decoding patterns that may then be grouped together or otherwise 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
32	**op_rd_i**	0	0	0	0	0	X	IM[3:0]				PB	PA	B		A	
224	**op_jmp_iglez**			G	L	E	IM[4:0]										
32	**op_wr_i**		1	0	0	0	X	IM[3:0]									
192	**op_jmp_iugle**			UG	L	E	IM[4:0]										
32	**- unused -**			1	1	1	.	.	.	.	.						
256	**op_byt_i**	1	0	IM[7:0]													
128	**op_shl_i**		1	0	U	IM[5:0]											
64	**op_add_i**			1	0	IM[5:0]											
8	**op_jmp_glez**				1	0	0	0	G	L	E						
8	**op_gto_glez**							1									
1	**op_add**					0	1	0	0	X	U						
1	**op_sub**					0	1	0	1								
1	**op_mul**					0	1	1	0								
.	**. . .**					.		.	.	.	.						
1	**op_pop**					1	1	1	1	1	0						
1	**op_nop**					1	1	1	1	1	1						

Figure 33. Opcode encoding.

As seen in the table above, the immediate reads are positioned at the base of the opcode space, consuming 32 codes. These, with the immediate writes, are shoehorned into the otherwise unused GLE = 000 (i.e. never) conditional area of the immediate conditional jump instruction that consumes 416 codes or nearly half of the entire space – I obviously feel that conditional short relative jumps are really important! Next is an unassigned group of 32 codes in the GLE = 111 (i.e. always) conditional area. Next is the immediate byte instruction which not surprisingly consumes 256 codes or one quarter of the entire space. Next are two immediate shifts and an immediate add which consume 64 codes each for a total of 192 codes. Following these are 8 conditional jump instructions and 8 conditional go to instructions, with the GLE field necessarily shifted down. There are 48 remaining code slots used for singles. The singles are arranged by functionality in groups of 16, with arithmetic first, logical second, and the third and final group something of a catch-all.

Only 27 of the 48 single instruction codes are actually assigned, leaving 21 remaining. Opcode space utilization is therefore:

(1024 – 32 – 21)/1024 = 96.7%

So 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. A compact opcode can be a rather harsh mistress due to the cramped working environment, hence the tucking away of the reads and writes in Hive's otherwise unused immediate conditional jump space. As you might imagine, small changes in grouping can sometimes dramatically alter the top speed of, and logic required for, the opcode decoder function.

The use of a multi-file text search and replace tool such as TextCrawler (Digital Volcano) during development can be quite useful for making global changes to opcode names, etc. reaching all the way into the bootcode text as well.

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, the foundation of the register set is a configurable multi-function single base register component with many parameter-based options.

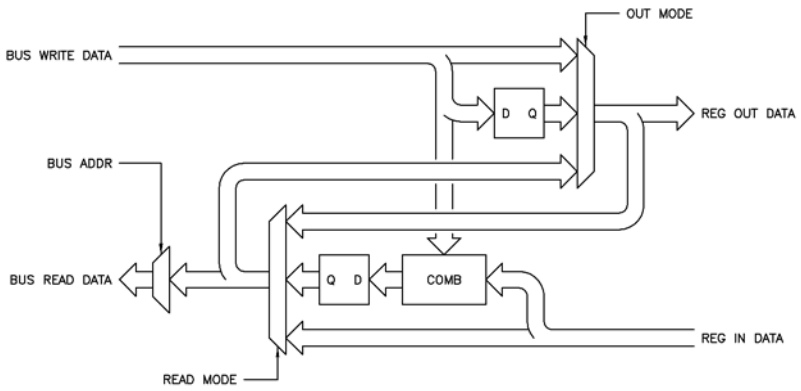

Figure 34. Configurable base register component.

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
"LTCH"	write latch
"READ"	no output latch, output selected read data

Figure 35. 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
"OUT"	no read latch, read selected out data

Figure 36. 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. Placement of the register set within the main memory address space is arbitrated by a shim component that differentiates processor read and write accesses.

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 verilog, 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 when you don't.

The Hive internal register set is located at the top of memory space (base 0xFFF0) and includes the following basic functionality:

```
Decode:
- 0x0 : Core version register - ver_reg
- 0x1 : Thread ID register - thrd_id_reg
- 0x2 : Clear register - clr_reg
- 0x3 : Interrupt enable register - intr_en_reg
- 0x4 : Opcode error register - op_er_reg
- 0x5 : Stack error register - stk_er_reg
- 0x6 - 0x7 : UNUSED
- 0x8 : I/O low register - io_lo_reg
- 0x9 : I/O high register - io_hi_reg
- 0xA - 0xF : UNUSED


================================================================================
- 0x0 : Core version register - ver_reg
--------------------------------------------------------------------------------

  bit  name                 description
-----  ----                 -----------
  7-0  ver_min[7:0]         minor version info
 15-8  ver_maj[7:0]         major 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.
- Major version changes when op_code binary decode changes (incompatibilty).

================================================================================
- 0x1 : Thread ID register - thrd_id_reg
--------------------------------------------------------------------------------

  bit  name                 description
-----  ----                 -----------
  2-0  thrd_id[2:0]         thread ID
 15-3  -                    0000000000000

Notes:
- Read-only.
- Threads can read this to discover their thread ID.

================================================================================
- 0x2 : Clear register - clr_reg
--------------------------------------------------------------------------------

  bit  name                 description
-----  ----                 -----------
  7-0  clr[7:0]             0=>1 clear thread; 1=>0 no effect;
 15-8  -                    00000000

Notes:
- Read / write.
- Per thread clearing.
- All bits cleared on async reset.

================================================================================
- 0x3 : Interrupt enable register - intr_en_reg
--------------------------------------------------------------------------------

  bit  name                 description
-----  ----                 -----------
  7-0  intr_en[7:0]         1=thread interrupt enable; 0=disable
 15-8  -                    00000000

Notes:
- Read / write.
- Per thread enabling of interrupts.
- All bits cleared on async reset.
```

```
================================================================================
- 0x4 : Opcode error register - op_er_reg
--------------------------------------------------------------------------------

  bit  name                 description
-----  ----                 -----------
  7-0  op_er[7:0]           1=opcode error; 0=OK
 15-8  -                    00000000

Notes:
- Clear on write one.
- Per thread opcode error reporting.

================================================================================
- 0x5 : Stack error register - stk_er_reg
--------------------------------------------------------------------------------

  bit  name                 description
-----  ----                 -----------
  7-0  pop_er[7:0]          1=lifo pop when empty; 0=OK
 15-8  push_er[7:0]         1=lifo push when full; 0=OK

Notes:
- Clear on write one.
- Per thread LIFO stack error reporting.

================================================================================
- 0x6 - 0x7 : UNUSED
================================================================================
- 0x8 : I/O low register - io_lo_reg
--------------------------------------------------------------------------------

  bit  name                 description
-----  ----                 -----------
 15-0  io_lo[15:0]          I/O data

Notes:
- Separate read / write.
- Reads of io_lo_reg freeze data in io_hi_reg, so read io_lo_reg first then
  read io_hi_reg for contiguous wide (32 bit) data reads.
- Writes function normally.

================================================================================
- 0x9 : I/O high register - io_hi_reg
--------------------------------------------------------------------------------

  bit  name                 description
-----  ----                 -----------
 15-0  io_hi[15:0]          I/O data

Notes:
- Separate read / write.
- Reads of io_lo_reg freeze data in io_hi_reg, so read io_lo_reg first then
  read io_hi_reg for contiguous wide (32 bit) data reads.
- Writes function normally.

================================================================================
- 0xA - 0xF : UNUSED
================================================================================
```

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 governed by this conduct as well. Simplicity allows one to more easily juggle the processor model in one's head, but can also greatly ease the verification problem. Hive has relatively simple control structures, minimal internal state, and the entire design is partitioned into right sized modules that 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 was accomplished by shrinking the data port widths to a trivial size, which allowed for manual examination of the results of all possible inputs. The multiplier was verified separately at full width by comparing its results to a second naively instantiated multiplier, both supplied 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 allowed for the testing of lower level aggregate functionality.

Once basic functionality was up (thread clearing, immediate byte, jumps) boot code enabled the processor to essentially verify itself. Stack functioning and error reporting were fully tested in all threads. Jump distances and all associated conditionals were confirmed. Each opcode was tested to make sure it was decoded and functioning correctly – distinctive signatures were 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 can often lead to changes in the op codes or other parts of the fundamental design.

Finally, several simple algorithms were 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 coding the log2 algorithm, a leading zero count instruction (lzc, clz) seemed like it would be a valuable inclusion (it has floating point normalization uses as well). So I coded up a fully parameterized verilog module and speed / functionally tested it, but I've successfully resisted the urge to actually stick it in the logical unit of the Hive core (so far). Ditto for ANDN (A & ~B) AKA a "bit squashing" instruction (it doesn't seem worth the extra hardware).

SPEED

Job #2 when building a processor is getting the most speed out of it 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 *vector_sr.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 (particularly when targeting x86 and other insanely complex processors) 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 fitter likes to replace pipe stages with block RAM, which can really slow things down.

Use a DCM to get your board clock up to the maximum speed of the core.

PROGRAMMING

The verilog hardware description language has an “initial” construct that can be used along with other verilog syntax features to write simple, fairly readable 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 "op_encode.h"
`include "register_set_addr.h"
```

These two includes pull in our opcode encoding and internal address register locations so we can refer to them by name rather than by their rather cryptic numerical encoding. The register set is located at the top of the address space, with a base address of 0xFFF0, and the addresses given in the file are offsets to this base address.

```
`define s0              2'd0
`define s1              2'd1
`define s2              2'd2
`define s3              2'd3
`define _               1'b0
`define P               1'b1
```

The above defines make stack indexes and popping a bit clearer.

```
`define op_rd_i         op_rd_i[9:4]
`define op_rd_ix        op_rd_ix[9:4]
//
`define op_jmp_iez      op_jmp_iez[9:5]
`define op_jmp_ilz      op_jmp_ilz[9:5]
`define op_jmp_ilez     op_jmp_ilez[9:5]
`define op_jmp_igz      op_jmp_igz[9:5]
`define op_jmp_igez     op_jmp_igez[9:5]
`define op_jmp_iglz     op_jmp_iglz[9:5]
`define op_jmp_i        op_jmp_i[9:5]
//
`define op_wr_i         op_wr_i[9:4]
`define op_wr_ix        op_wr_ix[9:4]
//
`define op_jmp_ie       op_jmp_ie[9:5]
`define op_jmp_il       op_jmp_il[9:5]
`define op_jmp_ile      op_jmp_ile[9:5]
`define op_jmp_iug      op_jmp_iug[9:5]
`define op_jmp_iuge     op_jmp_iuge[9:5]
`define op_jmp_igl      op_jmp_igl[9:5]
//
`define op_byt_i        op_byt_i[9:8]
//
`define op_shl_i        op_shl_i[9:6]
`define op_shl_iu       op_shl_iu[9:6]
`define op_add_i        op_add_i[9:6]
```

The above defines keep us from having to size the immediate instructions when we call them. If not properly sized we will get concatenation width warnings from the tool.

```
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] = { op_lit_u,                `_, `_, `s0, `s1 };  // lit => s1
i=i+1;   ram[i] = { 16'h0040                                   };  // sub addr
i=i+1;   ram[i] = {  op_gsb,                 `P, `_, `s1, `s0 };  // sub (return => s0), pop s1
i=i+1;   ram[i] = { `op_jmp_i,       -5'h1, `_, `_, `s0, `s0 };  // loop forever
//
i='h4;   ram[i] = { `op_jmp_i,       -5'h1, `_, `_, `s0, `s0 };  // loop forever
i=i+4;   ram[i] = { `op_jmp_i,       -5'h1, `_, `_, `s0, `s0 };  // loop forever
i=i+4;   ram[i] = { `op_jmp_i,       -5'h1, `_, `_, `s0, `s0 };  // loop forever
i=i+4;   ram[i] = { `op_jmp_i,       -5'h1, `_, `_, `s0, `s0 };  // loop forever
i=i+4;   ram[i] = { `op_jmp_i,       -5'h1, `_, `_, `s0, `s0 };  // loop forever
i=i+4;   ram[i] = { `op_jmp_i,       -5'h1, `_, `_, `s0, `s0 };  // loop forever
i=i+4;   ram[i] = { `op_jmp_i,       -5'h1, `_, `_, `s0, `s0 };  // loop forever

// intr space //

// code & data space //

// sub : output core version
i='h40;  ram[i] = { op_lit_u,                `_, `_, `s0, `s1 };  // lit => s1
i=i+1;   ram[i] = REG_BASE_ADDR                                 ;  // reg base addr
i=i+1;   ram[i] = { `op_rd_i,     VER_ADDR, `_, `_, `s1, `s0 };  // [s1+im] => s0
i=i+1;   ram[i] = { `op_wr_i,   IO_LO_ADDR, `P, `P, `s1, `s0 };  // s0 => [s1+im], pop both
i=i+1;   ram[i] = {  op_gto,                 `P, `_, `s0, `s0 };  // return, pop s0 (return addr)
// end sub
```

Finally some code!

The first line is located at address 0, which is where thread 0 vectors to when cleared. The instruction puts an unsigned literal in S1, the value of which is the address of a subroutine. The second line is the unsigned in-line literal value, 0x40. The third line calls the subroutine and pushes the return address to S0, and it simultaneously pops the subroutine address in S1 (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 loads the processor register set base address to S1, reads the core version, then writes the core version to the lower I/O port, pops both address and data simultaneously with the write (stack cleanup), then issues a gto S0 and pops S0 (stack cleanup), which is the way subroutines and interrupts are returned from in Hive.

Restoring Division Subroutine Example
A somewhat meatier example is division. The so called “restoring” division algorithm is shown below:

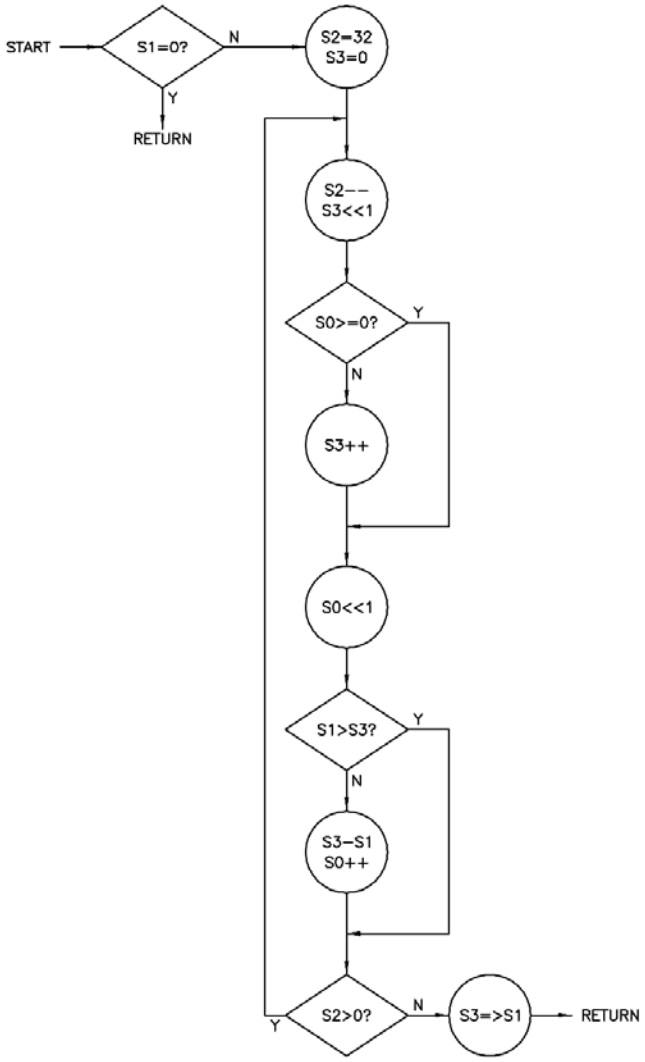

Figure 37. Restoring division algorithm flow chart.

This is much like pencil and paper long division, where the largest multiple of the denominator that will produce a non-negative result when subtracted from the shifted numerator is used. In binary the only multiples are 0 and 1 so the process is simplified considerably. A negative result can be tested for in advance, thus skipping both the subtraction and the restore in that case. The MSB end of the numerator currently under consideration is left shifted into S3, rather than the denominator to the right, and the dividend register does double duty as the currently uninvolved portion of the numerator (the MSB end) as well as the dividend which is being built up bit by bit during the calculations (the LSB end). Hive stacks for this subroutine are assigned as follows:

- S0 : Numerator A input / dividend (A/B) output
- S1 : Denominator B input / mod (A%B) output
- S2 : loop index
- S3 : subroutine return address / mod (A%B) during calculations

```
// sub : divide s0/s1 => result in s0, remainder in s1, return to (s3)
//
// s0 : A input, A/B output
// s1 : B input, A%B output
// s2 : loop index
// s3 : subroutine return address, A%B
//
// 0 input s1 is an error, return
i='h70;  ram[i] = { `op_jmp_iglz,     5'd1, `_, `_, `s0, `s1 };  // (s1!==0) ? skip return
i=i+1;   ram[i] = {  op_gto,                `P, `_, `s3, `s0 };  // return to (s3), pop s3
// loop setup
i=i+1;   ram[i] = { `op_byt_i,       8'd32, `_, `_, `s0, `s2 };  // 32=>s2
i=i+1;   ram[i] = { `op_byt_i,        8'd0, `_, `_, `s0, `s3 };  // 0=>s3
// divide loop
i=i+1;   ram[i] = { `op_add_i,       -6'd1, `_, `P, `s0, `s2 };  // s2--=>s2, pop s2
i=i+1;   ram[i] = { `op_shl_i,        6'd1, `_, `P, `s0, `s3 };  // s3<<1=>s3, pop s3
i=i+1;   ram[i] = { `op_jmp_igez,     5'd1, `_, `_, `s0, `s0 };  // (s0[31]==0) ? skip
i=i+1;   ram[i] = { `op_add_i,        6'd1, `_, `P, `s0, `s3 };  // s3++=>s3, pop s3
i=i+1;   ram[i] = { `op_shl_i,        6'd1, `_, `P, `s0, `s0 };  // s0<<1=>s0, pop s0
i=i+1;   ram[i] = { `op_jmp_iug,      5'd2, `_, `_, `s3, `s1 };  // (s1>s3) ? jump unsigned
i=i+1;   ram[i] = {  op_sub,                `_, `P, `s1, `s3 };  // s3-s1=>s3, pop s3
i=i+1;   ram[i] = { `op_add_i,        6'd1, `_, `P, `s0, `s0 };  // s0++=>s0, pop s0
i=i+1;   ram[i] = { `op_jmp_igz,     -5'd9, `_, `_, `s0, `s2 };  // (s2>0) ? do again
// s3=>s1; cleanup, return
i=i+1;   ram[i] = {  op_cpy,                `P, `P, `s3, `s1 };  // s3=>s1, pop both
i=i+1;   ram[i] = {  op_gto,                `P, `P, `s3, `s2 };  // return to (s3), pop s3 & s2
// end sub
```

The division subroutine code is above. The return is not skipped if the denominator value is zero, which is mathematically undefined (+/- infinity). Note that a conditional gto can't be used here because the S3 pop would always happen regardless of the outcome of the test, leaving us with no return address should the test result be false. The loop index is pushed to S2, S3 is initialized to zero, and the loop begins. The left shifting of S0 and S3 is performed in a staggered manner so that no intermediate results need to be stored. A shift and a conditional immediate add are used to left shift either a 0 or 1 into the remainder LSB. A conditional immediate jump bypasses the subtraction and 1 injection into the LSB of the dividend. After the loop has completed the remainder is copied to S1, with both popped to form a move. The return address in S3 and the loop index in S2 are both popped at return to clean up.

In terms of real time, assuming the denominator isn't zero and the return is skipped, it takes 3 cycles to test the input and setup the loop, 6 cycles per loop best case and 9 cycles worst case, with two cycles after the loop. For 32 loops this gives:

3 + 6*32 + 2 = 197 best case
3 + 9*32 + 2 = 293 worst case

For a 200 MHz clock and 8 clocks per cycle, this is 7.88 us best case, 11.72 us worst case.

Log_2 Subroutine Example

A somewhat similar example to division is the calculation of the 32 bit base 2 logarithm of an unsigned 32 bit input number. The algorithm shown here exploits the fact that $\log2(x^2) = 2*\log2(x)$, and is implemented via an unrolled binary search normalization process followed by a looped squaring processes.

The first section normalizes the input by shifting it to the left until the MSB is equal to 1 (which tests as negative). The number of shifts necessary to accomplish this is 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.

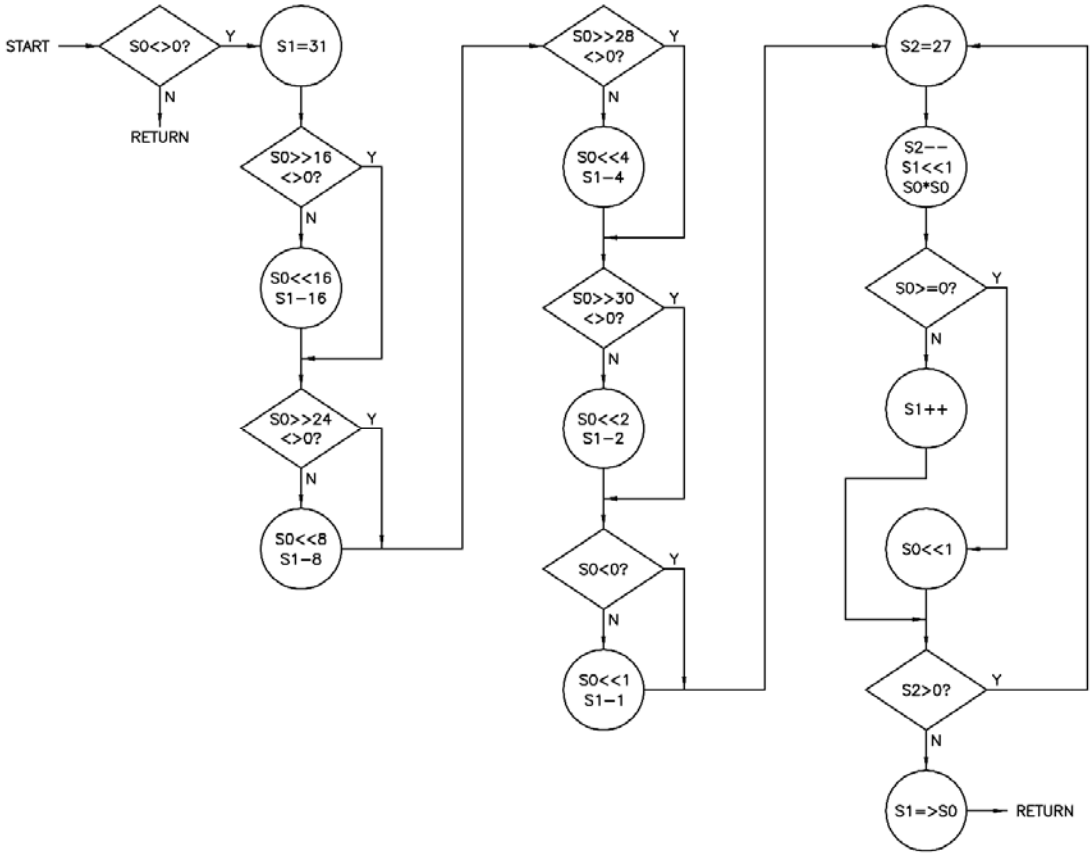

Figure 38. Log base 2 algorithm flow chart.

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 normalization / characteristic binary search. Finally we have the square / mantissa loop, with the loop exit test at the end. Hive stacks are assigned as follows:

- S0 : input / normalized input / squaring / output
- S1 : characteristic.mantissa
- S2 : mantissa loop index
- S3 : subroutine return address

```
// sub : log2(s0)=>s0, return to (s3)
//
// s0 : input, normalize, square, output
// s1 : characteristic (5 MSBs of output) and mantissa (27 LSBs of output)
// s2 : square loop index
// s3 : subroutine return address
//
// input 0 is an error, return
i='h60;  ram[i] = { `op_jmp_iglz,     5'd1, `_, `_, `s0, `s0 };  // (s0!==0) ? skip return
i=i+1;   ram[i] = {  op_gto,                `P, `_, `s3, `s0 };  // return to (s3), pop s3
// normalize binary search
i=i+1;   ram[i] = { `op_byt_i,       8'd31, `_, `_, `s0, `s1 };  // 31=>s1, characteristic
//
i=i+1;   ram[i] = { `op_shl_iu,     -6'd16, `_, `_, `s0, `s0 };  // s0>>16=>s0
i=i+1;   ram[i] = { `op_jmp_iglz,     5'd2, `_, `P, `s0, `s0 };  // (s0<>0) ? jump, pop s0
i=i+1;   ram[i] = { `op_shl_i,       6'd16, `_, `P, `s0, `s0 };  // s0<<16=>s0, pop s0
i=i+1;   ram[i] = { `op_add_i,      -6'd16, `_, `P, `s0, `s1 };  // s1-16=>s1, pop s1
//
i=i+1;   ram[i] = { `op_shl_iu,     -6'd24, `_, `_, `s0, `s0 };  // s0>>24=>s0
i=i+1;   ram[i] = { `op_jmp_iglz,     5'd2, `_, `P, `s0, `s0 };  // (s0<>0) ? jump, pop s0
i=i+1;   ram[i] = { `op_shl_i,        6'd8, `_, `P, `s0, `s0 };  // s0<<8=>s0, pop s0
i=i+1;   ram[i] = { `op_add_i,       -6'd8, `_, `P, `s0, `s1 };  // s1-8=>s1, pop s1
//
i=i+1;   ram[i] = { `op_shl_iu,     -6'd28, `_, `_, `s0, `s0 };  // s0>>28=>s0
i=i+1;   ram[i] = { `op_jmp_iglz,     5'd2, `_, `P, `s0, `s0 };  // (s0<>0) ? jump, pop s0
i=i+1;   ram[i] = { `op_shl_i,        6'd4, `_, `P, `s0, `s0 };  // s0<<4=>s0, pop s0
i=i+1;   ram[i] = { `op_add_i,       -6'd4, `_, `P, `s0, `s1 };  // s1-4=>s1, pop s1
//
i=i+1;   ram[i] = { `op_shl_iu,     -6'd30, `_, `_, `s0, `s0 };  // s0>>30=>s0
i=i+1;   ram[i] = { `op_jmp_iglz,     5'd2, `_, `P, `s0, `s0 };  // (s0<>0) ? jump, pop s0
i=i+1;   ram[i] = { `op_shl_i,        6'd2, `_, `P, `s0, `s0 };  // s0<<2=>s0, pop s0
i=i+1;   ram[i] = { `op_add_i,       -6'd2, `_, `P, `s0, `s1 };  // s1-2=>s1, pop s1
//
i=i+1;   ram[i] = { `op_jmp_ilz,      5'd2, `_, `_, `s0, `s0 };  // (s0<0) ? jump
i=i+1;   ram[i] = { `op_shl_i,        6'd1, `_, `P, `s0, `s0 };  // s0<<1=>s0, pop s0
i=i+1;   ram[i] = { `op_add_i,       -6'd1, `_, `P, `s0, `s1 };  // s1-1=>s1, pop s1
// loop setup
i=i+1;   ram[i] = { `op_byt_i,       8'd27, `_, `_, `s0, `s2 };  // 27=>s2
// square loop
i=i+1;   ram[i] = { `op_add_i,       -6'd1, `_, `P, `s0, `s2 };  // s2--=>s2, pop s2
i=i+1;   ram[i] = { `op_shl_i,        6'd1, `_, `P, `s0, `s1 };  // s1<<1=>s1, pop s1
i=i+1;   ram[i] = {  op_mul_ux,             `_, `P, `s0, `s0 };  // s0*s0=>s0, pop s0
i=i+1;   ram[i] = { `op_jmp_igez,     5'd2, `_, `_, `s0, `s0 };  // (s0[31]==0) ? jump
i=i+1;   ram[i] = { `op_add_i,        6'd1, `_, `P, `s0, `s1 };  // s1++=>s1, pop s1
i=i+1;   ram[i] = { `op_jmp_i,        5'd1, `_, `_, `s0, `s0 };  // skip
i=i+1;   ram[i] = { `op_shl_i,        6'd1, `_, `P, `s0, `s0 };  // s0<<1=>s0, pop s0
i=i+1;   ram[i] = { `op_jmp_igz,     -5'd8, `_, `_, `s3, `s2 };  // (s2>0) ? do again
// s1=>s0; cleanup, return
i=i+1;   ram[i] = {  op_cpy,                `P, `P, `s1, `s0 };  // s1=>s0, pop both
i=i+1;   ram[i] = {  op_gto,                `P, `P, `s3, `s2 };  // return, pop s3 & s2
// end sub
```

The subroutine code is above. The return is not skipped if the input value is zero. The normalization section shifts the input value to the left until the MSB is 1, subtracting the number of shifts necessary to do this from 31 which is 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 copied to S0 with both popped to form a move. The return address and S2 (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 skipped, it takes 1 cycle to test the input and enter the normalization section, which takes between 11 and 21 cycles to perform. After that there is a 1 cycle setup for the second loop. The two skip conditions oppositely test the same value, so one is always taken when the other one isn't, giving 7 cycles per loop with 27 loops, followed by 2 cycles of cleanup and return.

The best case is when the input value has an MSB of 1, which jumps through the normalization in 11 cycles. The worst case is an input value of 1 which takes 21 cycles to normalize. This gives:

1 + 11 + 1 + 7*27 + 2 = 204 best case

1 + 21 + 1 + 7*27 + 2 = 214 worst case

For a 200 MHz clock and 8 clocks per cycle, this is 8.16 us best case, 8.56 us worst case.

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 verilog boot text (or similar assembly) into uploadable binary data.

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 carefully 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 and two clearing mechanisms are provided).
- 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 nearing 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 nearing the end of support), and converted to PDF with Adobe Acrobat 8 Professional (which is free-ish due to license server retirement).
- The Hive opcode spreadsheet was prepared in MS Excel 2003.
- There are inexpensive FPGA demo boards readily available on eBay, Cyclone 2 is the minimum I would recommend. A very nice Cyclone 4 board can be had for $27 USD or thereabouts. Comparable Xilinx Spartan offerings may require a few code changes here and there (I haven't run the code through ISE yet) and will likely run slower – Altera apparently uses faster, and consequently leakier and more power hungry, transistors.

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)	Notes
2013-07-07	Edits to reflect code v1.10 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	First public release.

COPYRIGHT

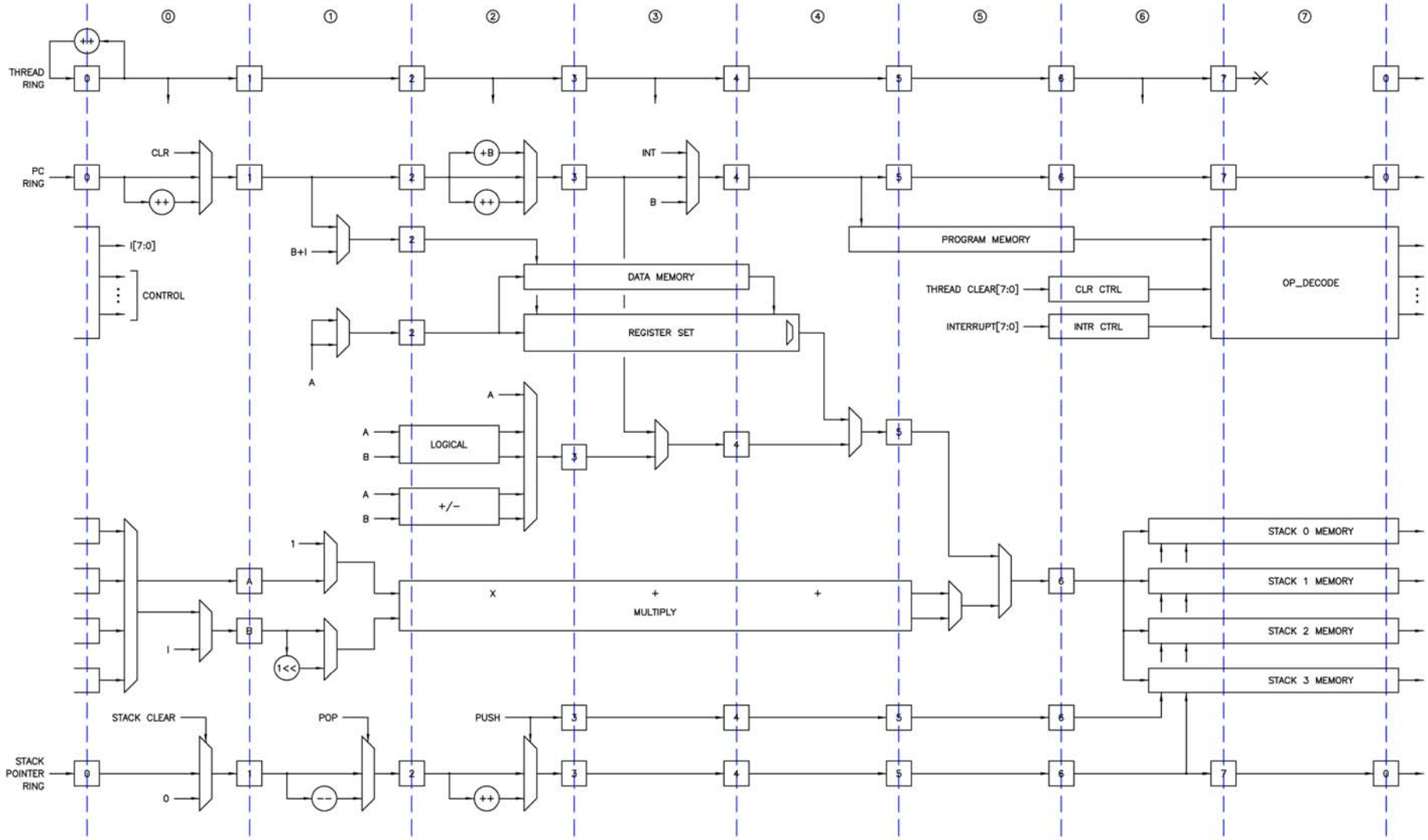

Figure 39. Hive core (embiggened).

OPCODE	DECODE	NOTES	I	push	wr	rd	clr	lit	jmp	gto	tst_ab	tst_gt	tst_lt	tst_eq	imda	imad	rtn	dm	cpy	shl	mul	sub	add	lg	ext	sgn
				0	0	0	0	0	0	0	0	-	-	-	0	0	0	0	0	0	0	0	0	0	0	1
op_rd_i	= { 2'b00, 4'b0000, 4'bxxxx };	// mem(B+I[3:0]) => A read immediate w/ offset (signed)	[0:15]	1		1												1								
op_rd_ix	= { 2'b00, 4'b0001, 4'bxxxx };	// {mem(B+I[3:0]), A[lo]} => A read immediate extended w/ offset	[0:15]	1		1												1							1	
op_jmp_iez	= { 2'b00, `iez, 5'bxxxxx };	// (A?0) PC+I[4:0] => PC jump relative immediate conditional	[-16:15]						1					1		1										
op_jmp_ilz	= { 2'b00, `ilz, 5'bxxxxx };		[-16:15]						1				1			1										
op_jmp_ilez	= { 2'b00, `ilez, 5'bxxxxx };		[-16:15]						1				1	1		1										
op_jmp_igz	= { 2'b00, `igz, 5'bxxxxx };		[-16:15]						1			1				1										
op_jmp_igez	= { 2'b00, `igze, 5'bxxxxx };		[-16:15]						1			1		1		1										
op_jmp_iglz	= { 2'b00, `iglz, 5'bxxxxx };		[-16:15]						1			1	1			1										
op_jmp_i	= { 2'b00, `igle, 5'bxxxxx };		[-16:15]						1			1	1	1		1										
op_wr_i	= { 2'b01, 4'b0000, 4'bxxxx };	// A[lo] => mem(B+I[3:0]) write immediate w/ offset	[0:15]		1																					
op_wr_ix	= { 2'b01, 4'b0001, 4'bxxxx };	// A[hi] => mem(B+I[3:0]) write immediate extended w/ offset	[0:15]		1																				1	
op_jmp_ie	= { 2'b01, `ie, 5'bxxxxx };	// (A?B) PC+I[4:0] => PC jump relative immediate conditional	[-16:15]						1		1			1		1										
op_jmp_il	= { 2'b01, `il, 5'bxxxxx };		[-16:15]						1		1		1			1										
op_jmp_ile	= { 2'b01, `ile, 5'bxxxxx };		[-16:15]						1		1		1	1		1										
op_jmp_iug	= { 2'b01, `iug, 5'bxxxxx };		[-16:15]						1		1	1				1										0
op_jmp_iuge	= { 2'b01, `iuge, 5'bxxxxx };		[-16:15]						1		1	1		1		1										0
op_jmp_igl	= { 2'b01, `igl, 5'bxxxxx };		[-16:15]						1		1	1	1			1										0
op_byt_i	= { 2'b10, 8'bxxxxxxxx };	// I[7:0] => A byte immediate (signed)	[-128:127]	1											1				1							
op_shl_i	= { 4'hc, 6'bxxxxxx };	// A<<I[5:0] => A shift left A (signed) immediate	[-32:31]	1											1					1						
op_shl_iu	= { 4'hd, 6'bxxxxxx };	// 1<<I[5:0] | A<<I[5:0] => A shift left immediate unsigned	[-32:31]	1											1					1						0
op_add_i	= { 4'he, 6'bxxxxxx };	// A+I[5:0] => A add immediate (I signed)	[-32:31]	1											1								1			
op_jmp_ez	= { 4'hf, 3'b000, `ez };	// (A?0) PC+B[lo] => PC jump relative conditional							1					1												
op_jmp_lz	= { 4'hf, 3'b000, `lz };								1				1													
op_jmp_lez	= { 4'hf, 3'b000, `lez };								1				1	1												
op_jmp_gz	= { 4'hf, 3'b000, `gz };								1			1														
op_jmp_gez	= { 4'hf, 3'b000, `gez };								1			1		1												
op_jmp_glz	= { 4'hf, 3'b000, `glz };								1			1	1													
op_jmp	= { 4'hf, 3'b000, `glez };								1			1	1	1												
op_gto_ez	= { 4'hf, 3'b001, `ez };	// (A?0) B[lo] => PC jump absolute conditional								1				1												
op_gto_lz	= { 4'hf, 3'b001, `lz };									1			1													
op_gto_lez	= { 4'hf, 3'b001, `lez };									1			1	1												
op_gto_gz	= { 4'hf, 3'b001, `gz };									1		1														
op_gto_gez	= { 4'hf, 3'b001, `gez };									1		1		1												
op_gto_glz	= { 4'hf, 3'b001, `glz };									1		1	1													
op_gto	= { 4'hf, 3'b001, `glez };									1		1	1	1												
op_add	= { 4'hf, 2'b01, 4'h0 };	// A+B => A add		1																			1			
op_add_x	= { 4'hf, 2'b01, 4'h2 };	// A+B => A add extended (signed)		1																			1		1	
op_add_ux	= { 4'hf, 2'b01, 4'h3 };	// A+B => A add extended unsigned		1																			1		1	0
op_sub	= { 4'hf, 2'b01, 4'h4 };	// A-B => A subtract		1																		1				
op_sub_x	= { 4'hf, 2'b01, 4'h6 };	// A-B => A subtract extended (signed)		1																		1			1	
op_sub_ux	= { 4'hf, 2'b01, 4'h7 };	// A-B => A subtract extended unsigned		1																		1			1	0
op_mul	= { 4'hf, 2'b01, 4'h8 };	// A*B => A multiply		1																	1					
op_mul_x	= { 4'hf, 2'b01, 4'ha };	// A*B => A multiply extended (signed)		1																	1				1	
op_mul_ux	= { 4'hf, 2'b01, 4'hb };	// A*B => A multiply extended unsigned		1																	1				1	0
op_shl	= { 4'hf, 2'b01, 4'hc };	// A<<B => A shift left A (signed)		1																1						
op_shl_u	= { 4'hf, 2'b01, 4'hd };	// 1<<B | A<<B => A 2^B | shift left A unsigned		1																1						0
op_and	= { 4'hf, 2'b10, 4'h0 };	// A&B => A logical AND		1																				0		
op_or	= { 4'hf, 2'b10, 4'h1 };	// A|B => A logical OR		1																				1		
op_xor	= { 4'hf, 2'b10, 4'h2 };	// A^B => A logical XOR		1																				2		
op_not	= { 4'hf, 2'b10, 4'h3 };	// ~B => A logical NOT		1																				3		
op_and_b	= { 4'hf, 2'b10, 4'h4 };	// &B => A logical AND bit reduction		1																				0	1	
op_or_b	= { 4'hf, 2'b10, 4'h5 };	// |B => A logical OR bit reduction		1																				1	1	
op_xor_b	= { 4'hf, 2'b10, 4'h6 };	// ^B => A logical XOR bit reduction		1																				2	1	
op_lit	= { 4'hf, 2'b11, 4'h0 };	// mem(PC) => A literal low (signed)		1				1										1								
op_lit_u	= { 4'hf, 2'b11, 4'h1 };	// mem(PC) => A literal low unsigned		1				1										1								0
op_lit_x	= { 4'hf, 2'b11, 4'h2 };	// {mem(PC),A[lo]} => A literal extended		1				1										1							1	
op_cpy	= { 4'hf, 2'b11, 4'h4 };	// B => A copy		1															1							
op_pc	= { 4'hf, 2'b11, 4'h8 };	// PC => A read PC (unsigned)		1													1									
op_gsb	= { 4'hf, 2'b11, 4'h9 };	// B[lo] => PC, PC => A subroutine call		1						1		1	1	1			1									
op_cls	= { 4'hf, 2'b11, 4'hc };	// clear stacks					1																			
op_pop	= { 4'hf, 2'b11, 4'he };	// do nothing (but allow pops)																								
op_nop	= { 4'hf, 2'b11, 4'hf };	// do nothing (no pops either)																								

Figure 40. Hive opcode encoding and decoding (ensmallened to the point of illegibility).