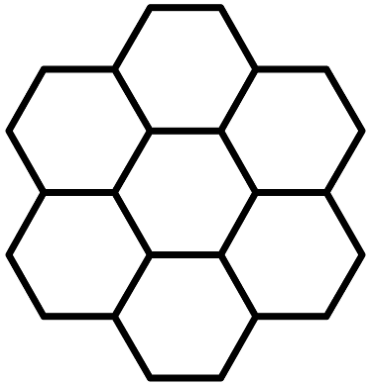


Hive

Register & Stack Hybrid ▪ 32-Bit ▪ Multi-Threaded ▪ Barrel Pipelined

SystemVerilog Soft Processor Core

CHANGE LOG

Date	Hive version	Notes
2022-10-24	13.12	Some general corrections / edits.
2022-10-21	13.12	Updated everything, removed most appendices, document source is now in *.odt format.
2015-09-10	08.06	Updated sim stuff to reflect pseudo code and additional help screen.
2015-09-03	08.06	Finished sim appendix, added sim tour, INV_F subroutine, many small edits. Switch from SV initial type boot code to MIF based boot code as generated by the C++ simulator.
2015-04-14	07.01	Added appendix on simulator. Expanded register set and vector support sections. New endianness screed.
2015-02-24	06.03	Revamped vector control and thread initialization. Register set additions / changes due to above. WAR & RAW sense reversed for clarity.
2014-12-25	06.02	New opcode op_cls_8, some style-driven opcode renaming (particularly the immediate field), other minor edits.
2014-07-15	06.01	Updates to reflect 32 bit memory access, new / changed opcodes, and transition to SystemVerilog code base.
2014-06-07	05.03	Text for enhanced interrupt support, register set changes. Additional UART discussion.
2014-04-21	04.06	More / rearranged text for the instructions / opcodes section, also more text re. register access.
2014-02-15	04.06	Minor opcode renaming, fixed a few typos in the document.
2014-01-05	04.05	Major edits to reflect 8 stacks and somewhat different opcodes / resized immediates, UART, etc. Updated and expanded the coding examples.
2013-07-07	01.10	Edits to reflect reshuffled opcodes. Fixed immediate add range on page 23. Added “barrel” processor classification and PDP 10 signed shift reference. Other sporadic minor edits.
2013-06-19	01.09	First public release.

CONTENTS

INTRODUCTION

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

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

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

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

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

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

HIVE FEATURES LIST

- A simple, compact, relatively stateless, high speed, barrel pipelined, multi-threaded, little endian design based on RASH™ (**R**egister **A**nd **S**tack **H**ybrid) technology.
- 2 operand machine with operand select and stack pointer control fields in the opcode.
- 32-bit data path with extended width mixed arithmetic results. All instructions execute in a single thread cycle, including 33 x 33 = 65 bit signed / unsigned multiply.
- 8-stage pipeline, which naturally stores the state of 8 strictly equal bandwidth threads, with no stalls or hazards possible.
- 8 independent general purpose LIFO data stacks per thread with parameterized depth and fault protections.
- 8 fully independent internal / external interrupts with no hierarchical limitations (one per thread).
- Variable width opcodes – 1 to 6 bytes – provide natural code space compression.
- All stack values have an individual set of “traveling” arithmetic result flags which are set automatically, allowing for instant response to context switches / interrupts.
- 32-bit internal register set in separate I/O space with highly configurable base register module that may be easily modified / expanded to provide coprocessor interfacing, enhanced I/O, detailed debug, etc.
- Common data & instruction memory space (Von Neumann architecture) enables dynamic code / data partitioning, combined code and data constructs, code copy & move, etc. All threads share the entire common data / code space, which facilitates global data / code factoring and thread intercommunication.
- Double buffered UART with BAUD generator and several parameterized options including FIFO buffering.
- 32 bit general purpose I/O port.
- Written in 100% highly portable SystemVerilog (other than clock PLLs, no vendor specific or proprietary language constructs) and partitioned into easy to understand and verify modules.
- Achieves aggregate throughput of ~180 MIPS in a bargain basement Altera EP4CE6 (Cyclone 4, speed grade 8, the target device for development) while consuming ~3200 logic elements.
- Free to use, modify, distribute, etc. (but only for the greater good, please see the copyright).

MOTIVATION

As a (mostly) digital designer who works primarily in FPGAs, I've been on the lookout for a simple processor core because projects often underutilize the hardware when operating at low data rates (e.g. a UART, or a sampled audio stream). If latency isn't a big issue, then why not multiplex the high-speed hardware with a processor construct? But the core needs to be simple, not consume too much in the way of logic (LUTs, block RAMs, multipliers), have compact op codes (internal block RAM isn't limitless nor inexpensive), keep the ALU sufficiently busy, and be easy to program.

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 often legally chained to that vendor's silicon and tool set. There are many free cores available on the web, but one may end up getting exactly what one paid for.

The Hive core is my offering for this problem area. The essentially free and naturally emergent multi-threading / rigid scheduling mechanism in Hive isn't unique; I believe it was implemented as far back as 1964 on the CDC 6000 series peripheral processors. Hive bit shift distances are treated as signed which works out rather nicely, but the ancient PDP 10 did this as well. The notion of multiple stacks isn't original, nor is the explicit control over the processor stack pointer. And the register/stack hybrid as implemented and described here (indexed stacks, top-entry-only conservative access with pop bit override) was described in Appendix E of Mark Shannon's 2006 MSc Thesis. The way extended arithmetic results are dealt with uniformly in Hive may possibly be somewhat novel, but who knows? Processors have been around long enough that most of the good ideas have been mined out and put to the test in one form or another, which makes it difficult / unlikely to bring something fundamentally new or innovative to the table.

REGISTER MACHINES VS. STACK MACHINES

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

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

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

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

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

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

REGISTER / STACK HYBRID

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

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

pop bit	read / write	Stack	Behavior
0	read	no change	register type read
1	read	pop	stack type read
0	write	push	stack type write
1	write	pop & push	register type write

Figure 1. Hybrid register / stack behavior.

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

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

Figure 2. One and two operand hybrid register / stack behavior.

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

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

BASIC DESIGN DECISIONS

Operands
How many operands should be in the opcode? I picked 2 to keep the opcode small, so Hive is a 2 operand machine. The “free” move inherent in a 3 operand machine is quite powerful. But a 2 operand hybrid register / stack machine can come close to the same efficiency, and without all the opcode bloat. Here are the rules:

- For single input ALU operations, the data source is B and the result destination is A. For example: A := not(B).
- For two input ALU operations, the primary data source is A, the secondary source is B, and the result destination is A. For example: A := A – B; or equivalently: A -= B.
- For single input conditional branch statements, A is tested against zero, and the address is an immediate offset to PC. For example: PC := (A>0) ? PC += IM.
- For two input conditional branch statements, A is tested against B, and the address is an immediate offset to PC. For example: PC := (A!=B) ? PC += IM.
- For memory reads, the read data destination is A, and the address is either B, or an immediate offset to B or PC. For example: A := mem[B+IM].
- For memory writes, the write data source is A, and the address is either B, or an immediate offset to B or PC. For example: mem[PC+IM] := A.
- For internal register set access, the absolute address is immediate and the data source or destination is A. Read example: A := reg[IM]. Write example: reg[IM] := A.
- For goto and subroutines, the absolute address is B and the subroutine return address (the PC) is pushed to A.
- When an interrupt is taken the return address (i.e. the non-incremented PC) is automatically pushed to stack 7 – making it a bit of an odd man out, but it is the only “special” stack, and this is the only way in which it is “special”.

Therefore A is the primary data source and destination for two operand operations, is the primary data conditionally 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 conditionally tested against, and provides the address or address base.

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

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

Instruction Width(s)
Conveniently, a byte is sufficient to contain the opcode of all Hive instructions. A few instructions don’t need to reference the stacks, and of this group, those with no immediates can be a single byte wide. The vast majority of instructions do reference the stacks, and are therefore 2 or more bytes in length, depending on the immediate field width. Immediate data is 0, 1, 2, or 3 bytes in width; and 1, 2, and 4 byte wide literals are supported via the main memory read port. So Hive instructions are as compact as possible and therefore efficient, utilizing between 1 and 6 bytes.

Main Memory Data Access Width
Hive's variable instruction width byte granularity obviously necessitates byte addressing of main memory (though all memory ports are 32 bits wide).

Main Memory PC & Address Width
PC and address width are parameterized and so are set at build-time. PC width may be set to coincide with address width, or wider if so desired, up to and including the ALU data width. Address width directly sets the depth of the main memory instantiation and BRAM resource usage (note that deeper settings may negatively impact the top speed of the core).

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

A simple and uniform method of handling wide arithmetic results is to process them as double width regardless of operation (add, subtract, multiply) and select either the lower (i.e. normal) half of the result or the upper (i.e. extended) half of the result via instructions. The obvious downside here is that obtaining the full width result takes at least two cycles even when the operation is actually performed in one. For the full result, it may seem wasteful to perform the same internal calculation both times, but one probably should not think of this as major effort for the ALU or as a huge opportunity lost; in the vast majority of cases only the lower or extended arithmetic result is required.

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

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

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

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

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

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

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

Figure 3. 4 bit input / 8 bit result corner cases.

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

Mixed Sign Arithmetic
DSP scenarios quite often require the multiplication of an unsigned filter coefficient with a signed sample, producing a signed result. Filter coefficients are almost always less than one so the extended result is desired. Hive supports mixed signed and unsigned arithmetic via various opcodes. Even if the result would be the same regardless of sign, the signedness of the both inputs is required to properly set the “traveling” arithmetic flags, which can then be used to limit or saturate the result.

Arithmetic Flags
Most processors have a flag register, the bits of which signify under/overflow, carry in/out, etc. from the last arithmetic operation. Whether or not flags are set or used for a particular operation is then determined by various opcodes, causing increased decode complexity and bloat. I think that placing the flags in a global register is a mistake because it increases state, which forces one to back it up and restore it when an interrupt is taken, or severely limits what can actions can safely transpire in an interrupt. The band-aid many processors provide here is hardware that automates the backup and restore, along with some portion of the registers (stack frame), to and from memory. Hive uses stacks, therefore there are no registers requiring backup and restore with interrupts. Furthermore, the flag bits “travel around” with the results on the stacks using the extra BRAM 4 bit overhead, so they don’t require backup / restore either.

Endianness
Hive is little endian (as the processor gods intended; big endian is provably asinine).

ALU DESIGN

Design decisions regarding ALU construction drive much of the remaining processor design. Building an ALU for anything but the most trivial of processors is more involved than “compute all results and pick the one you want” though there will necessarily be a lot of that going on. The wide output multiplexer produced by this naive approach can be a bottleneck, so intermediate multiplexing and registering must be employed judiciously in order to keep both the required logic to a minimum and the speed to a maximum. Furthermore, some of the logic can be used to calculate more than one type of final result, though probably less than one might initially imagine. Textbooks sometimes emphasize economic / outdated measures that don’t find their way into actual hardware.

Multiplication

Let’s start with the elephant in the room – the multiply unit. If we want to do audio DSP, we need 16 x 16 = 32 bits signed as a rather unsuitable bare minimum. We might try to 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 completeness and simplicity, let’s set the goal as full 32 x 32 = 64 bits signed and unsigned which makes everything symmetric. The use of a signed base entity requires 33 x 33 = 65 to accommodate unsigned and mixed operations, which conveniently is slightly less than twice the width of a single 18 x 18 = 36 FPGA hardware multiplier. The presence of fast multiplier hardware in the ALU is what largely separates “real” processors from the trivial, as it enables all sorts of vitally useful things such as Newton-Raphson inverse; polynomial-based square root, sine, exp, log, etc.; and digital filters.

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

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

Figure 4. Multiplication example.

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

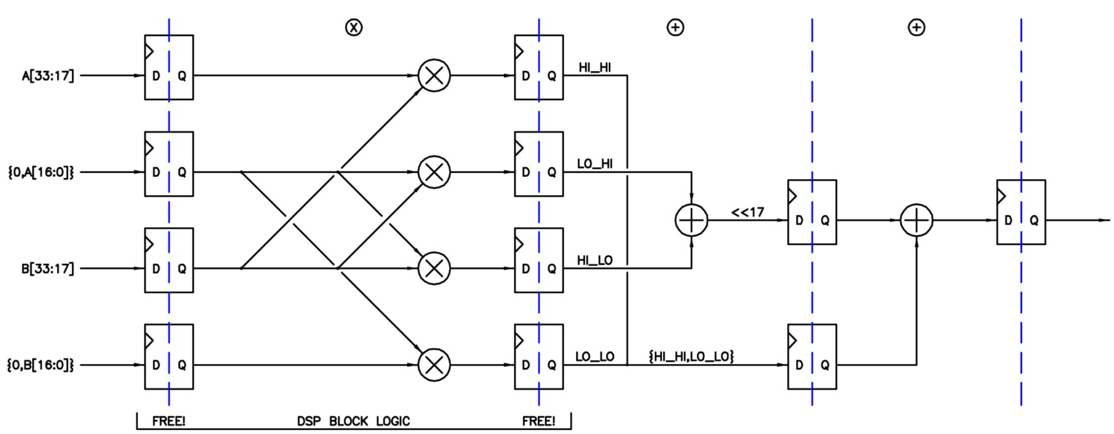


Figure 5. Three stage 33 x 33 = 66 bit signed pipelined multiplication.

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 device the multiply is restricted to 200 MHz, which means we should endeavor to make all of the other logic somewhat faster in order to have a chance of hitting 200 MIPS aggregate with the final design. The dedicated I/O registering in the multiplier hardware should certainly be used, with inter-stage registering to isolate the addition hardware, giving three stages and four clocks of latency.

Shifting

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

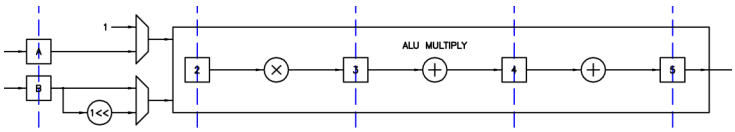


Figure 6. The Multiply and Shift unit.

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

Shift	MSB	LSBs	LSBs	1<<LSBs	Input	Output: upper, lower	upper | lower
+7	0	111	7	10000000	10110111	01011011,**10000000**	11011011
+6	0	110	6	01000000	10110111	00101101,**11000000**	11101101
+5	0	101	5	00100000	10110111	00010110,**11100000**	11110110
+4	0	100	4	00010000	10110111	00001011,**01110000**	01111011
+3	0	011	3	00001000	10110111	00000101,**10111000**	10111101
+2	0	010	2	00000100	10110111	00000010,**11011100**	11011110
+1	0	001	1	00000010	10110111	00000001,**01101110**	01101111
0	0	000	0	00000001	10110111	00000000,**10110111**	10110111
-1	1	111	7	10000000	10110111	**01011011,**10000000	11011011
-2	1	110	6	01000000	10110111	**00101101,**11000000	11101101
-3	1	101	5	00100000	10110111	**00010110,**11100000	11110110
-4	1	100	4	00010000	10110111	**00001011,**01110000	01111011
-5	1	011	3	00001000	10110111	**00000101,**10111000	10111101
-6	1	010	2	00000100	10110111	**00000010,**11011100	11011110
-7	1	001	1	00000010	10110111	**00000001,**01101110	01101111
-8	1	000	0	00000001	10110111	**00000000,**10110111	10110111

Figure 7. 8 bit example of left shift, unsigned right shift, and rotation using a full multiplier.

Though we are thinking of the shift distance input as signed, the shifted one must be presented to the multiplier as unsigned for the 1 << 31 case to work correctly. Then presenting the input data to be shifted as unsigned or signed will conveniently produce unsigned ("logical" or zero extended) and signed ("arithmetic" or sign extended) right shifts. (Note that independent control over the input signedness is required for this to work, global signedness is not sufficient, which restricts the construction of signed shift from a series of more basic instructions.) With these additions we have left shift covered, which is sign agnostic, as well as unsigned and signed right shift.

Rotation
By combining via adding – or simply ORing – the lower and extended results of shifting, we conveniently get a bi-directional rotate. In this case, both inputs to the multiplier are treated as unsigned.

Shifting Limits
How best to handle shifting beyond the limits of +/-31? One could simply do modulo 32 of the shift distance here, which would be correct for large rotations, but would generally produce nonsense results for large shifts. Large unsigned right shifts should clearly be zero because zeros propagate from the MSb position, but should large signed right shifts be zero too, or should they consist of all sign bits? And what about large left shifts? After some programming experience with this situation, it seems safest and most useful to have all out-of-bounds shifts result in zero. This makes logical sense from a left shift perspective, because zeros propagate from the LSb position. And it makes arithmetic sense to have negative input large signed right shifts go to zero when the negative result becomes too small to represent. Large shift distances could of course be detected in software rather than hardware, but it's fairly trivial to add logic to detect shift over/under-range and multiplex in zero as the result.

Other Uses
Can more be done with this construct? A multiplexer on port A with a fixed input value of one can be used for a couple of things. One use is copying the B input shifted-one result to the output of the multiplier, which is useful for generating powers of 2 for bit setting & masking, etc. (though please note that there are no Hive opcodes expressly dedicated to bit manipulation). Another use is even simpler – multiplication by one replicates the B input to the output of the multiplier, which provides a free "copy B" route through the ALU (unused in Hive).

Addition and Subtraction
Next we need to consider addition and subtraction. Signed and unsigned can be handled with the same method employed in the multiplier, i.e. by making the inputs one MSB wider and sign or zero extending them depending on whether that input value is to be considered signed or not. As with multiplication, overflow / carry out is extended into the double width data space and selected via instructions. Note that the lower word result is sign agnostic, so only the extended result will vary based on input signed / unsigned status, though the signed nature of the inputs is relevant if the result is to be limited or saturated.

Addition and multiplication are commutative, but subtraction isn't, which is a bit of a problem sometimes for a two operand machine. Reverse subtraction is therefore provided to cover this scenario, though it doesn't get used a ton, and could probably be removed from the design without significant negative impact.

Logical & Miscellaneous Functions
Several one and two operand functions are implemented:

Operation	Assembly	Description	Uses
CPY	A := B	Copy / move	General
NSB	A := nsb(B)	Negate sign bit	Signed <=> unsigned, +/- 1/2
LIM	A := lim(B)	Limit to unsigned range	General
SAT	A := sat(B)	Saturate to signed range	Filter node clamping
FLP	A := flp(B)	Flip bits end for end	General
SWP	A := swp(B)	Swap bytes end for end	Little <=> big endian
NOT	A := ~B	Bitwise negate	General
BRX	A := ^B	XOR bit reduction (0 or -1)	LFSR feedback
SGN	A := sgn(B)	Sign (-1 if B<0, else 1)	Absolute value, sign preserve
LZC	A := lzc(B)	Leading zero count	Floats, scaling, log & exp algorithms
PCR	A := PC	Read the program counter	Tricky stuff

Figure 8. One operand functions.

Operation	Assembly	Description	Uses
AND	A &= B	Bitwise AND	Bit masking
ORR	A |= B	Bitwise OR	General
XOR	A ^= B	Bitwise XOR	CRC generation

Figure 9. Two operand functions.

The LZC operation is extremely useful for floating point, scaling, normalizing, log2 / exp2 calculations, etc. – it's hard to believe some processors omit it. AND and OR bit reductions are not included here because they are essentially just comparisons to -1 and zero, respectively.

The Completed ALU

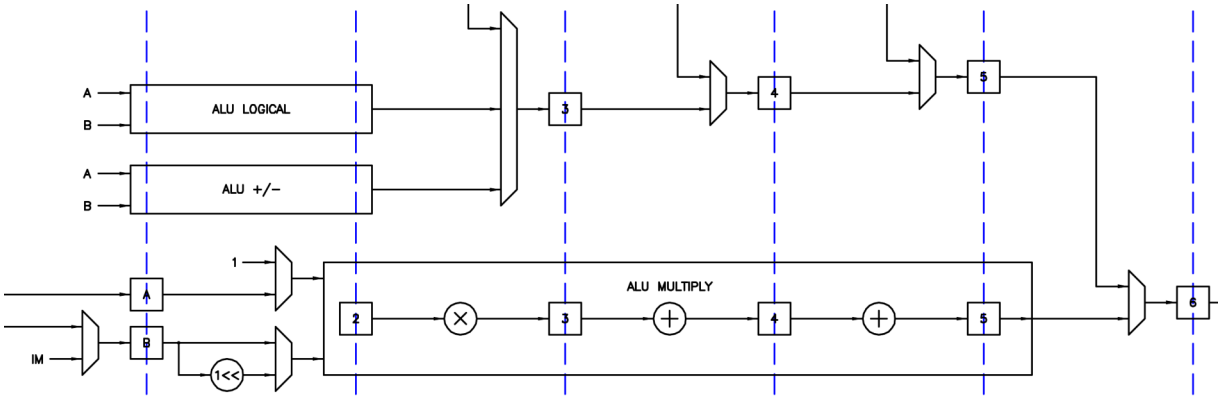


Figure 10. The Arithmetic and Logic Unit (ALU).

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

A somewhat thorny issue with ALU design is working out how the control lines should be decoded. 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. Also, separate decoders are use for the control and data paths.

RESET & VECTORING

Resets
There are several ways to reset or clear the Hive core. From most to least significant:

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

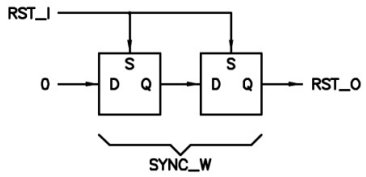


Figure 11. Reset bridge.

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

Vectoring
Two types of vectoring, or breaking out of current execution, are supported in Hive. The first is via thread clearing, which is described in the reset section above. The second is via an interrupt service routine (ISR), where the PC is pushed to stack 7 and the PC is loaded with the ISR address for that thread.

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

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

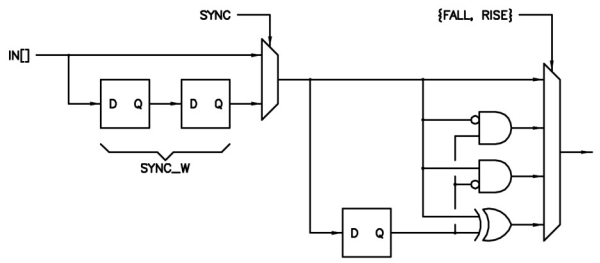


Figure 12. XSR input conditioning logic.

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

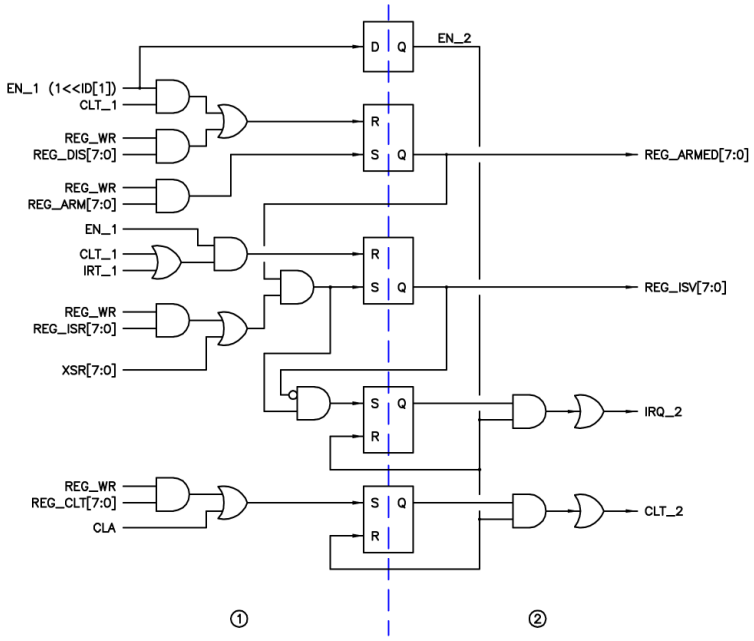


Figure 13. Vector control logic.

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

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

REGISTER SET

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

RBUS

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

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

Base Register Component

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

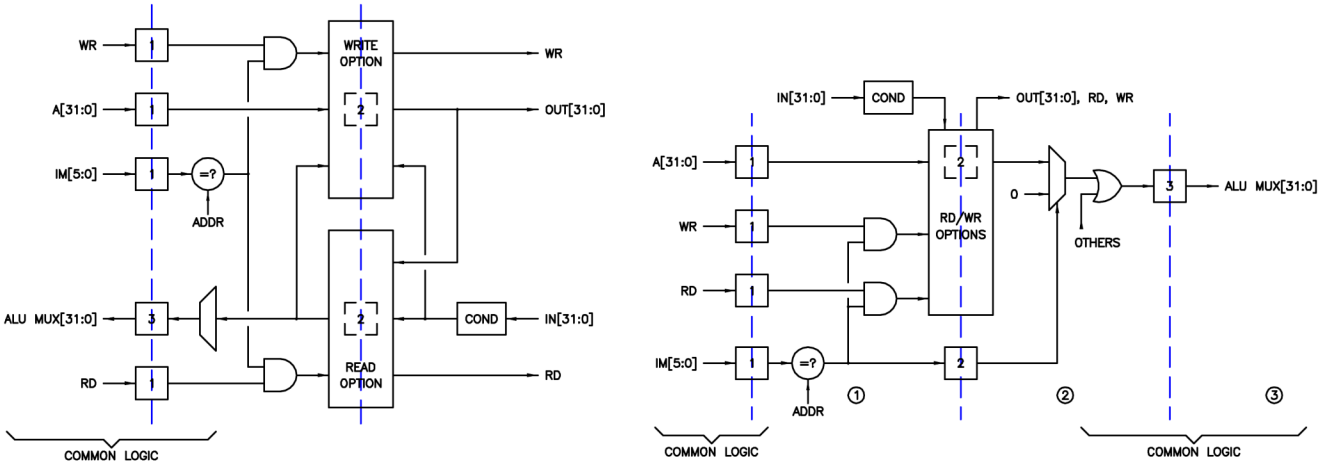


Figure 14. Configurable base register component (two views).

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

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

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

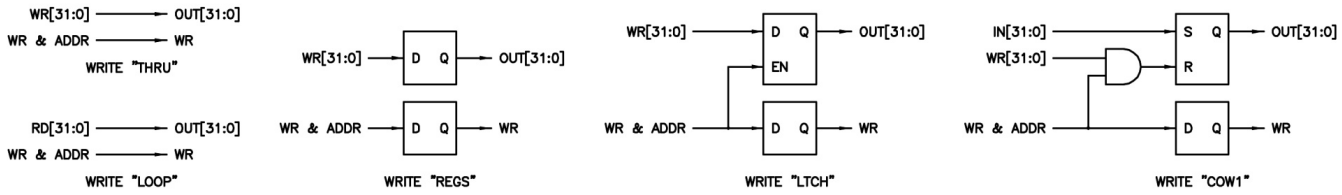


Figure 15. Write modes.

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

Figure 16. Write mode descriptions.

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

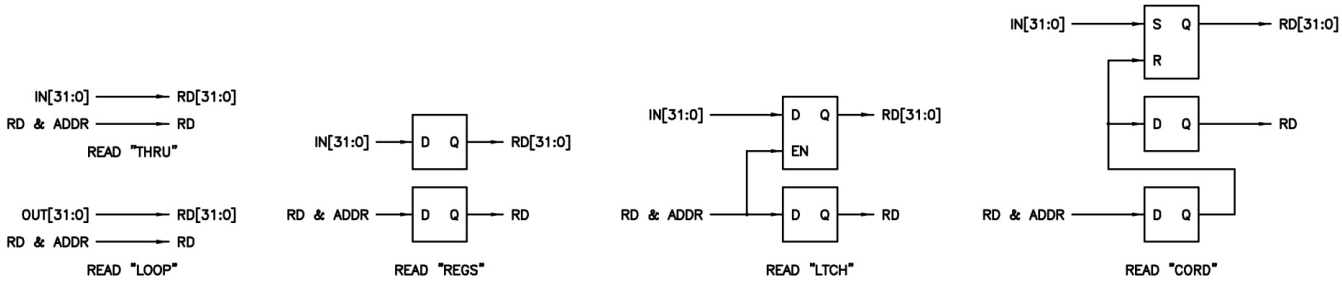


Figure 17. Read modes.

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

Figure 18. Read mode descriptions.

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

Input Conditioning
Via masks, register input data can be optionally re-synchronized and/or made edge sensitive via the circuitry shown below:

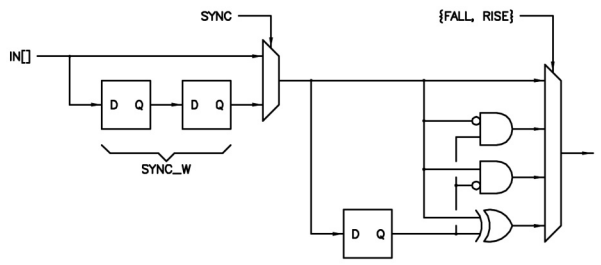


Figure 19. Input conditioning logic.

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

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

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

Figure 20. Some typical examples.

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

Hive Register Set

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

```
--------------------------------------------------------------------------------

MINIMUM:
- 0x00 : VECT
- 0x01 : TIME
- 0x02 : ERROR
- 0x03 : GPIO
EXTENDED:
- 0x04 : UART_TX
- 0x05 : UART_RX
- 0x06 : SPI
- 0x07 : -EMPTY-

================================================================================
- 0x00 : VECT
--------------------------------------------------------------------------------

   bits  name                description
   ----  ----                -----------
[31:24]  clt_req[7:0]        write 1 request thread clear, read ver[15:8]
[23:16]  isr_req[7:0]        write 1 request thread ISR, read ver[7:0]
[15:08]  xsr_dis[7:0]        write 1 thread XSR disarm, read service status
[07:00]  xsr_arm[7:0]        write 1 thread XSR arm, read arm status

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

================================================================================
- 0x01 : TIME
--------------------------------------------------------------------------------

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

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

================================================================================
- 0x02 : ERROR
--------------------------------------------------------------------------------

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

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

```
================================================================================
- 0x03 : GPIO
--------------------------------------------------------------------------------

   bits  name                description
   ----  ----                -----------
[31:00]  gpio[31:0]          I/O data

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

================================================================================
- 0x04 : UART_TX
--------------------------------------------------------------------------------

   bits  name                description
   ----  ----                -----------
   [31]  tx_full             1=buffer is full; 0=buffer can take data
[30:08]  -                   0
[07:00]  tx_data[7:0]        write TX UART data buffer

Notes:
- Writes to this register push data to the TX UART data buffer.
- Full bit is self clearing.

================================================================================
- 0x05 : UART_RX
--------------------------------------------------------------------------------

   bits  name                description
   ----  ----                -----------
   [31]  rx_empty            1=buffer is empty; 0=buffer has data
[30:08]  -                   0
[07:00]  data[7:0]           read RX UART data buffer

Notes:
- Reads from this register pop data from the RX UART data buffer.
- Empty bit is self clearing.

================================================================================
- 0x06 : SPI
--------------------------------------------------------------------------------

   bits  name                description
   ----  ----                -----------
   [31]  busy                read busy
[30:09]  -                   0
   [08]  csn                 write chip select, active low
[07:00]  spi_data[7:0]       read & write SPI data

Notes:
- Initiate SPI bus cycle by writing data with csn bit low.
- Continue SPI bus cycle by writing data with csn bit low.
- Terminate SPI bus cycle by writing csn bit high.
- Read & Write new data when not busy.

================================================================================
- 0x07 : -EMPTY-
================================================================================
```

UART

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

Conventions

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

Transmit Side

When parallel data is written to the UART register the ready bit goes low. The machine transitions from the idle state to the load state, where the parallel data is taken, ready is returned high to signal that new parallel data may be written to the register, and the machine then transitions to the data state. Here the data is sent out over the serial line as described above. Once this is done, the machine goes idle. Note that new parallel data can be written to the register as soon as the current parallel data it is taken from it at the load state, making this a “double buffered” action.

Receive Side

When a low is seen on the serial line (i.e. the start bit) the machine transitions from the idle state to the data state, where the serial data is sampled mid bit and stored in a parallel form. After 10 bits are stored (start, data, stop) the machine transitions to the load state and the parallel data is presented to the register set. At this point, the machine goes idle and the line is sampled for the current level. If the level is low this is an error. The write ready is also sampled at this point, and an error is flagged if the data wasn’t taken. The number of stop bits greater than one is irrelevant to the RX side. This is a “double buffered” action because old parallel data is presented in the register until new data is completely received, after which the old data is overwritten with the new data.

BAUD Generator

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

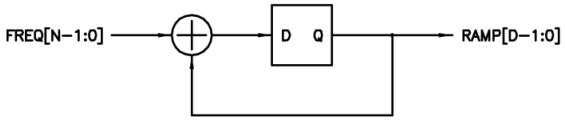


Figure 21. UART BAUD generator.

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

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

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

The MSB of the accumulator is employed as the UART BAUD clock input, with a roughly square wave duty cycle. It isn’t actually used as a clock per se (which would be bad form), but rather it is examined for level changes by the UART logic which runs off of the system clock. When used as a square wave clock source, the phase accumulator construct is known for generating spurious spectral components. There are various techniques that

can be applied in order to reduce these, such as setting the FREQ LSb permanently to a '1' and/or dithering the accumulated value with noise, but for UART use spectral purity concerns are moot.

Separate baud rate generators are used for the RX and TX sides to simplify synchronization via DDFS enable / reset.

Options

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

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

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

PIPELINED CORE

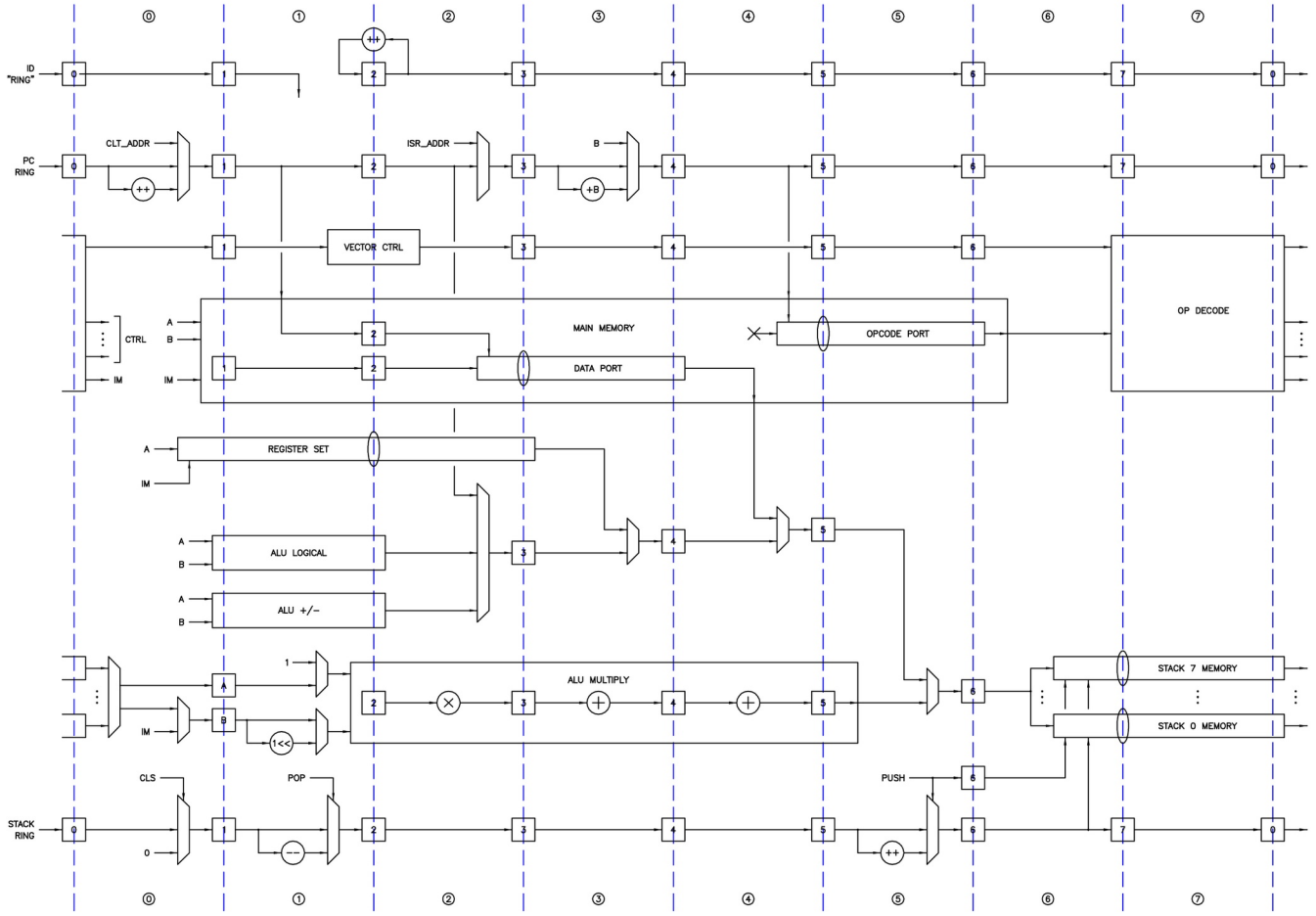


Figure 22. Hive core.

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

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

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

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

Time ID Ring

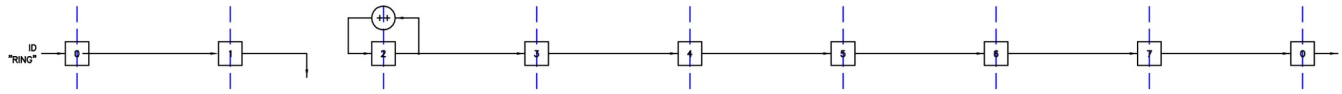


Figure 23. The Time ID "Ring".

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

Program Counter Ring

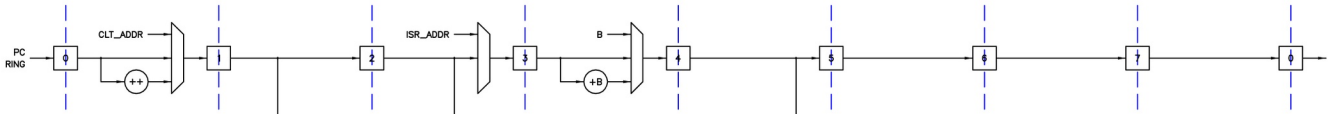


Figure 24. The Program Counter Ring.

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

Control Ring

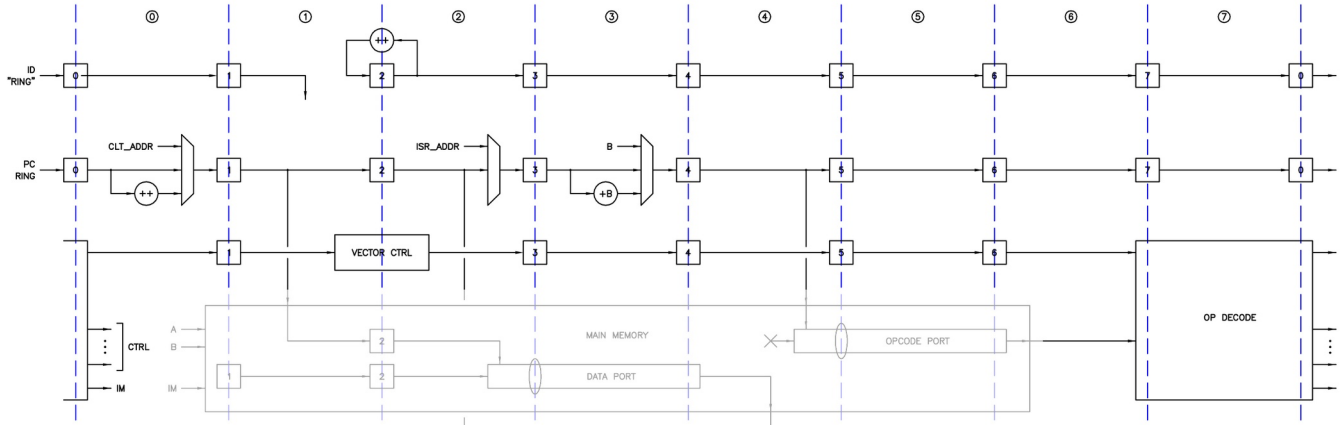


Figure 25. The Control Ring.

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

Stacks Ring

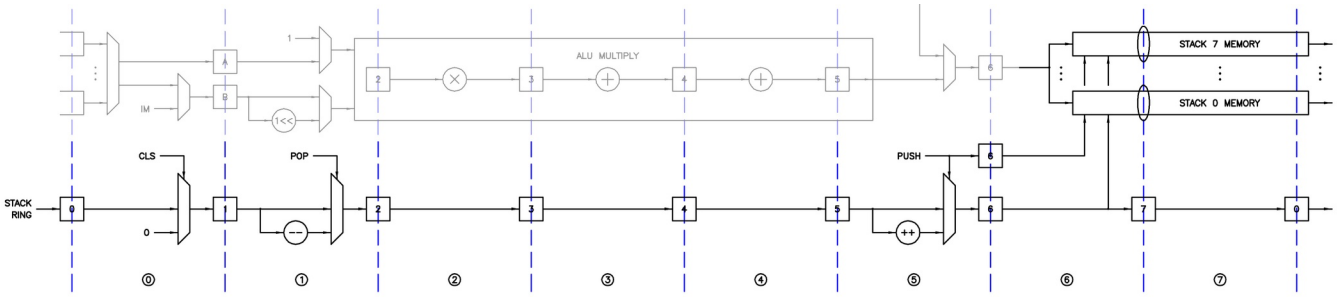


Figure 26. The Stacks Ring.

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

Data Ring

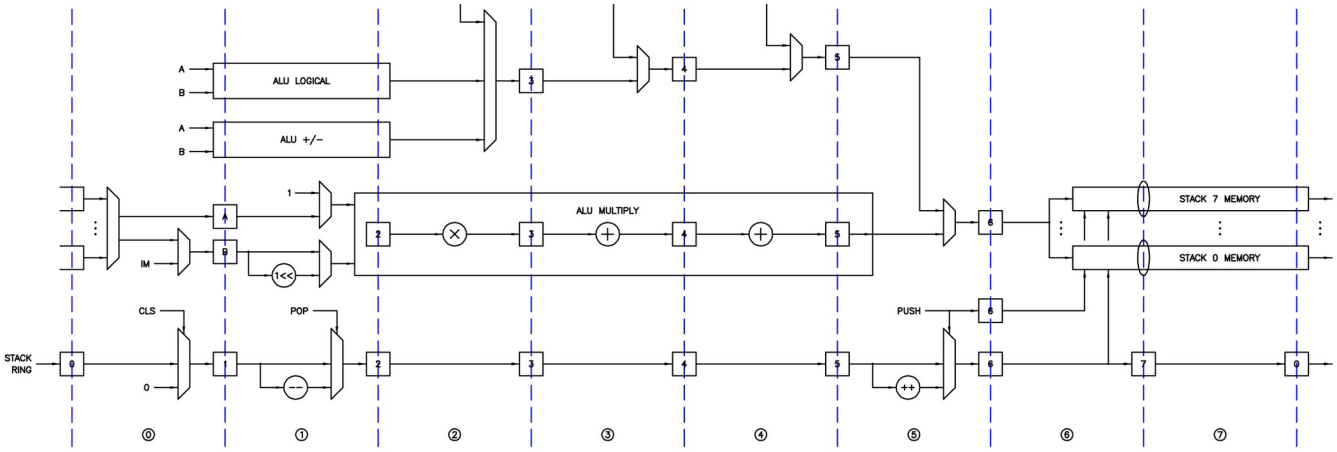


Figure 27. The Data Ring.

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

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

Main Memory

Hive data and program memory space are shared. The opcodes are variable width so the memory must be byte addressable. To ease SV synthesis limitations, main memory is constructed of four byte wide units, with automatic address offsets and byte barrel shifters at the input and output ports.

INSTRUCTIONS / OPCODES

With the hardware structure in place, we can now decide on the final operations and their encoding. In actuality, the design process isn't exactly cut and dried, and the inclusion and format of certain instructions will obviously ripple back into the hardware structure. It doesn't matter terribly much if opcodes aren't all that human readable – It certainly doesn't hurt if things work out that way in the end, but it probably shouldn't be a primary goal.

Beyond Turing completeness, selecting a sufficiently self-contained and balanced set of "basic building blocks" instructions for general purpose use is a hazy puzzle – you want to accommodate your own early coding examples, but how do you ensure efficient coverage for future code you and others may write without an over-reliance on anecdata, or resorting to a "kitchen sink" design? Studying the instruction sets of similar processors is a useful activity here, as is coding up often-used simple functions like reciprocal & division, log2 & exp2, square root, sine & cosine, floating point basic arithmetic & higher functions, and other operations not directly supported by the ALU hardware. If the coding process feels particularly laborious, or the resulting code strikes you as unusually awkward or cryptic, then you likely have more work to do. This is the "art" portion of processor design, often driven by seat-of-the-pants statistics.

For convenience, some "pseudo" instructions can be created via assembly. For example, a right shift can be implemented by negating the signed left shift distance. An absolute goto can be formed by a relative jump and some PC calculations. A greater than or equal conditional is the same as the not less than comparison. A greater than conditional can use less than with a swap of the operands. So it would be a waste to implement these kinds of overlapping functionality in hardware.

Fixed vs. Variable Width Instructions

Determining how to best fit the instructions into the opcode space can be a challenge too. The two basic choices here are fixed vs. variable width instructions.

Fixed width brings welcome regularity to the fetching of code, but it is overly confining and inefficient. For anything other than maybe a very simple single data stack machine, a one byte-wide instruction is too narrow to be practically useful. A two byte-wide instruction has just barely enough room, but you'll tear your hair out trying to shoehorn everything in, compromising too much functionality in the process. A four byte-wide instruction works and has been indeed been done repeatedly, but for many instructions it's overkill – do you really want to throw away 4 bytes on something like NOP? Go down that path and you'll be searching in vain for ways to keep trivial instructions somehow as significant as the considerable memory space they occupy.

A variable width opcode introduces some complexity, but solves everything. It provides natural code compaction, and it gives the designer plenty of breathing room, as well as and a very clean and modular working space. There are downsides, such as you can't be 100% sure how to interpret a random snippet of code without starting at the very beginning of execution and decoding everything up to the point you are interested in. Luckily this is exactly what the processor does anyway, but it can create issues when displaying disassembled memory dumps and the like. But the upsides far outweigh the downsides.

Instruction Fields

For ease of decoding, the various opcode fields should stay in one place as much as possible and not be used to specify excessive alternate functionality. Fairly far into the project it became clear that the opcode itself should be a single byte, and the obvious best place for it the byte 0 position.

There are two stack indexes of three bits each, with one pop bit for each index, which neatly consumes one byte of opcode space. Most processors utilize the operand select field room freed up when few or no operands are required for a particular operation, and Hive does this as well. So operand select & pop are generally in the byte 1 position, though this is instead an immediate field for a few instructions.

In-Line Immediate & Literal Data

The bandwidth consumed by immediate / literal data is quite important; some processor designs devote literally (!) half of the opcode space to a single immediate data operation. With Hive, immediate data is read via the main memory opcode port – it is limited to 3 bytes maximum because the opcode consumes 1 byte. Literal data is read via the main memory data port, so it can be up to 4 bytes wide. There is a full width literal instruction, as well as signed and unsigned literal half (two byte) and byte instructions. The main difference between immediate and literal data is the point at which they are injected into the rings; immediates are available quite early so there is time to manipulate and mux them in, whereas literals are available only later, and so have more limited uses.

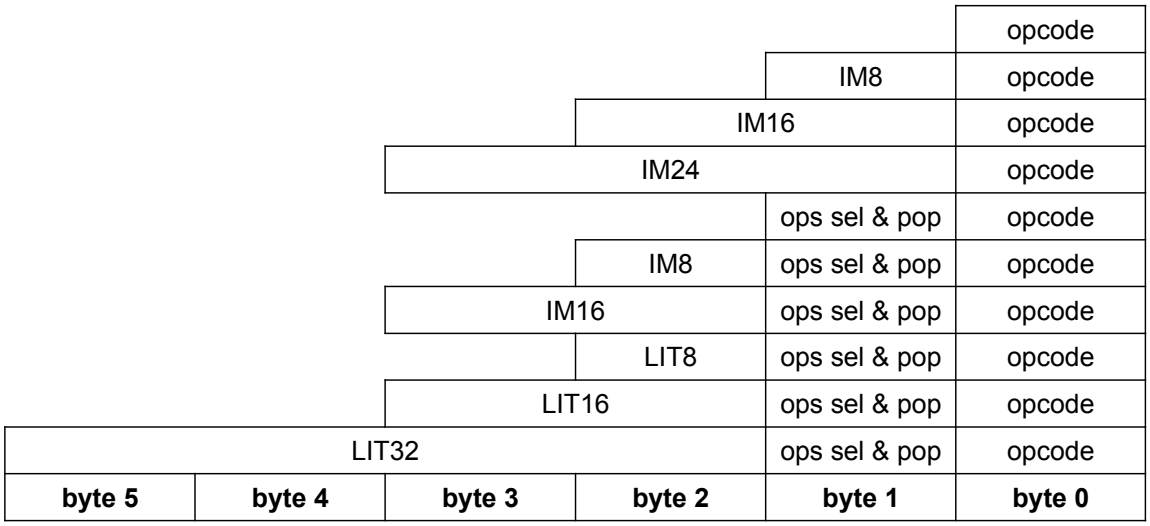


Figure 28. General instruction formats.

In-Line Addresses?
At the excellent suggestion of one Hive reviewer, I experimented with the literal mechanism as a source of absolute addresses and address offsets. I very reluctantly abandoned this tantalizing avenue because of the timing pinch point it created between conditional evaluation, address / offset selection, next address calculation, fetch, and decoding, which unacceptably slowed down the core logic.

Branching
There are four types of non-immediate branches:

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

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

Immediate Branching
There is only one type of immediate branch:

- **JMP** (jump) immediate is relative to the current PC and can be unconditional or conditional. It jumps a signed immediate distance either unconditionally, or if the test (A?0), (A?odd), or (A?B) is true. The minimum immediate jump distance is one byte. Unconditional immediate distance can be up to 3 bytes, for conditional 2 bytes is the maximum.

The ephemeral nature of immediate jump offsets is a good match for the jumps themselves.

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

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

Some conditional sign conventions / observations:

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

Memory Access
There are quite a few instructions for memory access:

- Write 4 byte (full width), 2 byte (half), and 1 byte width data.
- Read 4 byte (full width), 2 byte (half), and 1 byte width data. The 2 and 1 wide reads support both unsigned (zero extended) and signed (signed extended) modes.

The address can be B, or a one or two byte immediate unsigned offset to B or PC.

Register Set Access
There are four instructions for register set access: full data width read and write using either B or a 1 byte wide unsigned immediate as the register address.

Immediate Shifts
Immediate shift instructions are highly useful because one or two shifts / rotations can perform many chores that would otherwise require dedicated instructions and hardware (full width MSb / sign flag; arbitrary width sign / zero extension; isolation of contiguous bit fields; 2^n integer modulo, etc.). But full shifting left and right only requires 6 bits, so there is no need to implement anything beyond 1 byte wide immediate versions of these functions.

Immediate Arithmetic & Logical Functions
There are 1 and 2 byte wide immediate versions of add, subtract, reverse subtract, and multiplication, in all of their signed and/or unsigned inputs, and normal or extended output variants. Similarly, there are 1 and 2 byte wide immediate versions of the two operand functions and, orr, and xor. There is no practical use case for immediate versions of the one operand functions.

Immediate Stack Pop & Clear
These are two byte instructions that use one immediate byte in a one-hot fashion to pop / clear all, none, or any combination of stacks at once. I've never used the clear instruction, but the pop instruction is very useful for mass cleanup.

No Operation & Halt
These are one byte instructions that do nothing / halt the thread. Halted threads can be "resurrected" via an interrupt, so this is a great way to "park" a thread once it's done with all of its real-time tasks.

Encoding
When assigning the actual numerical values to the instructions – the operational encoding or opcodes – it's important to make the decoding as straightforward and orthogonal as possible. It might help to use a spreadsheet to keep track of them and to try out different arrangements and groupings.

Opcode	Operations	IM / Lit	AB field	Width
0x0 to 0x1	no-operation, halt	-	no	1
0x04 to 0x05	stacks pop, stacks clear	1	no	2
0x08 to 0x0f	jump immediate (IM8, IM16, IM24)	1 to 3	no	2 to 4
0x20 to 0x2f	arithmetic extended AB	-	yes	2
0x30 to 0x3f	logical & misc AB	-	yes	2
0x40 to 0x4f	conditional branching B	-	yes	2
0x50 to 0x57	read / write memory B	-	yes	2
0x58 to 0x5f	read literal	1 to 4	yes	3 to 6
0x60 to 0x6f	arithmetic AB	-	yes	2
0x70 to 0x7f	shift, rotate, power, read / write regs, bitwise	-	yes	2
0x80 to 0x8f	jump conditional immediate (IM8)	1	yes	3
0x90 to 0x97	read / write memory B + immediate (IM8)	1	yes	3
0x98 to 0x9f	read / write memory PC + immediate (IM8)	1	yes	3
0xa0 to 0xaf	arithmetic immediate (IM8)	1	yes	3
0xb0 to 0xbf	shift, rotate, power, read / write regs, bitwise (IM8)	1	yes	3
0xc0 to 0xcf	jump conditional immediate (IM16)	2	yes	4
0xd0 to 0xd7	read / write memory B + immediate (IM16)	2	yes	4
0xd8 to 0xdf	read / write memory PC + immediate (IM16)	2	yes	4
0xe0 to 0xef	arithmetic immediate (IM6)	2	yes	4
0xfc to 0xfe	bitwise immediate (IM16)	2	yes	4

Figure 29. Opcode encoding.

As seen in the table above, the instructions are arranged by functionality into roughly three groups, with the first group something of a catch-all, 1 byte immediate second, and 2 byte immediate third. There are some opcode slots open for future expansion, but the opcode space is otherwise largely consumed.

IMPLEMENTATION

Coding

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

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

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

Any largish project will make one more familiar with the abilities and limitations of the languages and tools employed. Hive started out in Verilog 2001, and while working on version 5 of the core I discovered that System Verilog was actually a straightforward update to Verilog, and not some radically different verification-centric language as the confusing name change (among other things) led me to believe. (In particular, I encourage you to read the papers by Stuart Sutherland on both Verilog and SV synthesis). The most welcome change in SV is the replacement of the cumbersome and confusing *wire* and *reg* basic types with the generic *logic* type, which establishes some sanity and makes optional registering of signals and ports simpler. I found the added SV package support quite useful for holding magic numbers such as global parameters and derivations of them, as well as custom enumerated logic types, which helps greatly with things like opcode encoding and decoding. (The use of don't cares in enumerated types is particularly powerful when coupled with casex statement decode.) SV multi-dimensional I/O comes in quite handy with the PC and ID ring ports, and SV default (*) port connect cuts down hugely on bug-prone inter-module wiring, while providing useful port linting checks. Altera's Quartus tool supports most of the important SV enhancements to Verilog, and Xilinx claims Vivado does too.

Verification

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

All basic blocks should be fully tested before being assembled together. With Hive, most module port widths and associated internal logic are parameterized so, for example, full verification of the ALU may be accomplished by shrinking the data port widths to a trivial size and manually examining the results of all possible inputs. The multiplier may be verified separately at full width by comparing its results to a second naively instantiated

multiplier, both supplied identically with corner cases and random input (the inclusion of this test hardware is a parameterized option for the multiplier base module). The intermediate control and data ring constructs allow for the testing of lower level aggregate functionality.

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

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

Speed

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

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

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

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

SIMULATION AND ASSEMBLY

I wrote an assembler and interactive simulator for Hive in C++ and am considering rewriting in the Go language. Writing a simulator gives you a different perspective of the design, which aids verification. Writing an assembler relieves a ton of drudgery associated with coding, and gives you massive insight into programming languages in general, particularly if you implement automatic address label calculations and scoping.

I would also strongly encourage you to write integer and floating point libraries for your processor, where all things math will be revealed (hint: everything is a region normalized / change of variable / polynomial – except inverse, which is best done with Newton-Raphson). This is an absolutely fantastic exercise, both for expanding your brain, and for putting your processor architecture and instruction set to the test!

SUNDRIES

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

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

About Me

I am primarily a digital designer with over 25 years of experience targeting programmable logic devices (i.e. FPGAs and CPLDs made by Altera, Xilinx, and Lattice) with side experience in the mixed signal and analog fields. I have designed many types of digital structures including general purpose processors, DPLLs, TSIs, UARTs, I2C & SPI masters / slaves / arbiters, UTOPIA interfaces, generic FIFOs as well as Ethernet packet and ATM cell FIFOs. I strive for high levels of portability, readability, and craftsmanship in my code, and very much enjoy the documentation phase.

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

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

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

The last decade or so has been spent researching, experimenting, and prototyping in the digital / analog Theremin field – one reason for which Hive was developed. Hive is the beating heart of my D-Lev Theremin project!

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).

Copyright

RASH™ – Register And Stack Hybrid

APPENDIX A : LIFOS

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

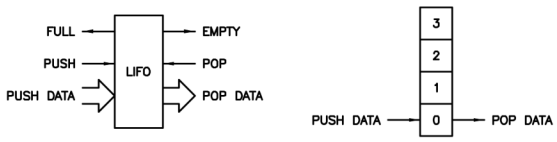


Figure 30. LIFO symbols.

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

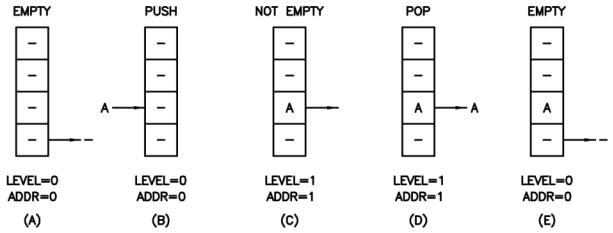


Figure 31. LIFO stack operations – push then pop from empty state.

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.

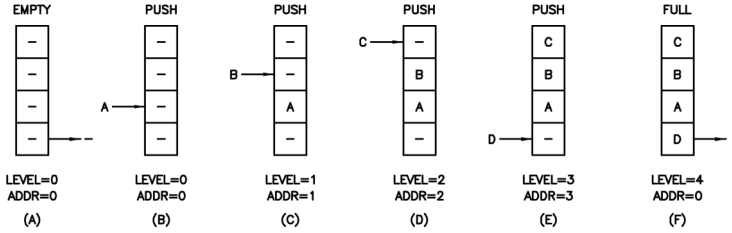


Figure 32. LIFO stack operations – push from empty state to full state.

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

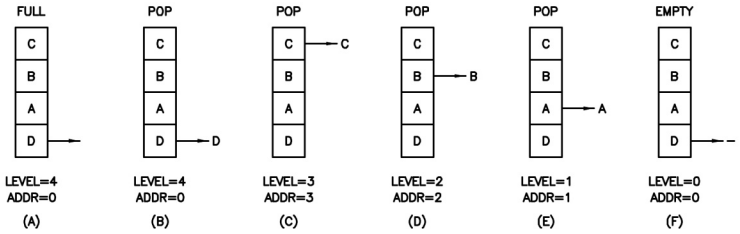


Figure 33. LIFO stack operations – pop from full state to empty state.

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

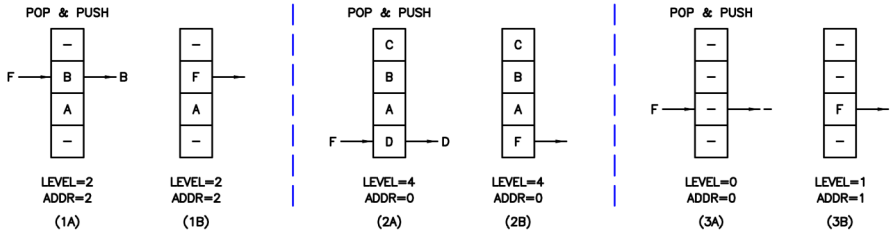


Figure 34. LIFO stack operations – three pop & push scenarios.

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

Stack Protection

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

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

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

APPENDIX B : 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.

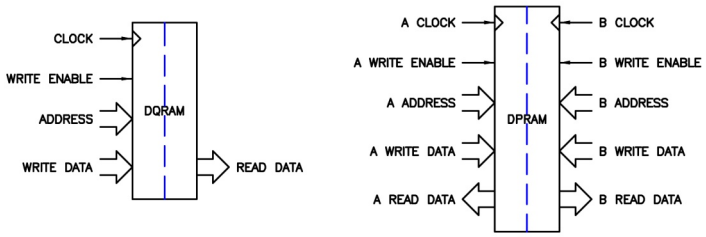


Figure 35. Block RAM: simple (DQ) on left, true dual port (DP) on right.

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

Block RAM resources have configurable variable widths, from some maximum down to a single bit. For widths of eight and above an additional bit per byte (8+1, 16+2, 32+4) is provided for out-of-band signaling, individual byte enables, CRC, error correction, and other common uses. Hive uses these extra bits to hold arithmetic results flags for each stack entry.

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

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

Figure 36. FPGA Block RAM port aspect ratio address mapping.

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

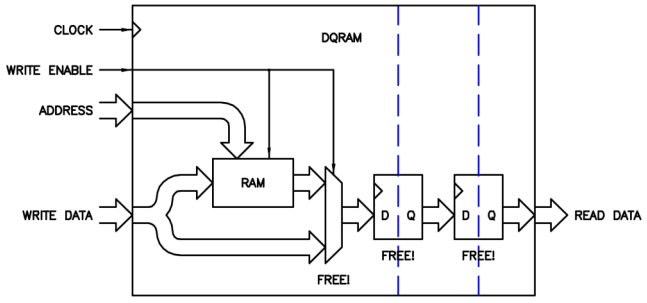


Figure 37. Block RAM internal resources.

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

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

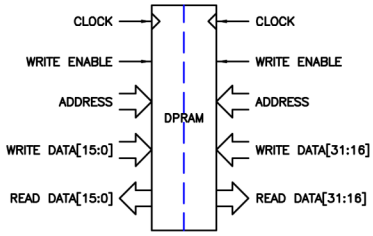


*Figure **38**. True dual port block RAM utilized as DQ RAM.*

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 device, DP data ports are limited to a maximum of 16 (+2) bits wide, and DQ data ports to a maximum of 32 (+4) bits wide – and the 1:2 ratio of these width limits makes sense given the above transformation. Since our LIFO stacks can employ DQ RAM (due to the single pointer / single address port) we can make them 32 bits wide using a single device.

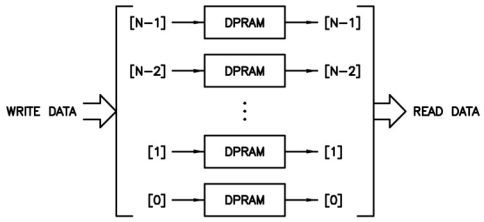


Figure 39. Block RAMs combined via bit-slice.

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

DSP Hardware

Since even quite low-end FPGAs these days have fairly fast hardware multipliers in some form of a DSP block, we should undertake any new designs with the knowledge and trust that they will be there. There is little point in

leaving multiply operations out of our instruction set, and no point in trying to outsmart the FPGA manufacturers by constructing what would inevitably be slower and larger multipliers out of shifters, adders, etc. – both of which would needlessly strand this valuable resource. Therefore, it behooves us to understand the dedicated multiply hardware.

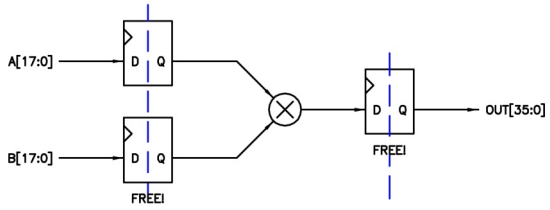


Figure 40. Signed multiplier hardware typically found in an FPGA.

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

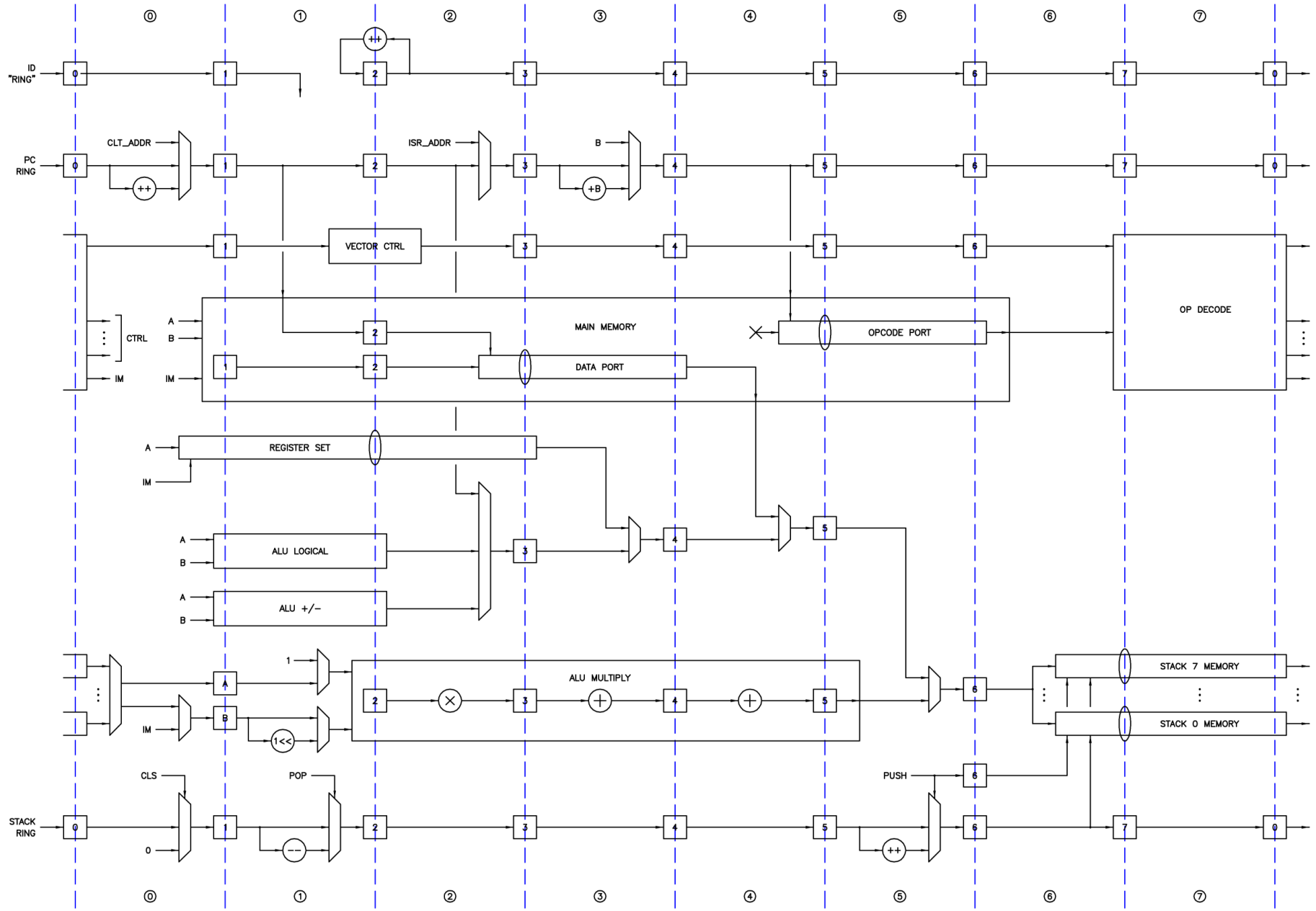
Altera multiplier blocks are signed by default, which makes sense because this convention simplifies sign extension of the inputs. In order to make a signed multiplier perform unsigned math, all that is necessary is to construct it one MSB wider at the inputs and force those MSBs to zero (zero extension).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

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

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

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

APPENDIX C : HIVE CORE



APPENDIX D : INSTRUCTION SET

opcode	operation	assembly	Notes	IM / Lit	AB field	bytes
0x00	nop	nop	no operation (pc++)		no	1
0x01	hlt	hlt	halt (pc := pc)		no	1
0x04	pop	pop[7:0]	pop stacks, one bit per stack	1	no	2
0x05	cls	cls[7:0]	clear stacks, one bit per stack	1	no	2
0x08	jmp_8	pc += #	jump immediate	1	no	2
0x09	jmp_16	pc += #	jump immediate	2	no	3
0x0a	jmp_24	pc += #	jump immediate	3	no	4
0x20	add_xu	A +xu= B	add extended unsigned			2
0x21	add_xus	A +xus= B	add extended unsigned signed			2
0x22	add_xsu	A +xsu= B	add extended signed unsigned			2
0x23	add_xs	A +s= B	add extended signed			2
0x24	sub_xu	A -xu= B	subtract extended unsigned			2
0x25	sub_xus	A -xus= B	subtract extended unsigned signed			2
0x26	sub_xsu	A -xsu= B	subtract extended signed unsigned			2
0x27	sub_xs	A -xs= B	subtract extended signed			2
0x28	sbr_xu	A :xu= B - A	subtract reverse extended unsigned			2
0x29	sbr_xus	A :xus= B - A	subtract reverse extended unsigned signed			2
0x2a	sbr_xsu	A :xsu= B - A	subtract reverse extended signed unsigned			2
0x2b	sbr_xs	A :xs= B - A	subtract reverse extended signed			2
0x2c	mul_xu	A *xu= B	multiply extended unsigned			2
0x2d	mul_xus	A *xus= B	multiply extended unsigned signed			2
0x2e	mul_xsu	A *xsu= B	multiply extended signed unsigned			2
0x2f	mul_xs	A *xs= B	multiply extended signed			2
0x30	cpy	A := B	copy			2
0x31	nsb	A := nsb(B)	negate sign bit (A := B[~31,30:0])			2
0x32	lim	A := lim(B)	limit (unsigned)			2
0x33	sat	A := sat(B)	saturate (signed)			2

opcode	operation	assembly	Notes	IM / Lit	AB field	bytes
0x34	flp	A := flp(B)	flip bits end for end (A := B[0:31])			2
0x35	swp	A := swp(B)	swap bytes (A := B[7:0,15:8,23:16,31:24])			2
0x36	not	A := ~B	bitwise negate			2
0x38	brx	A := ^B	xor bit reduction			2
0x39	sgn	A := sgn(B)	sign ((B < 0) ? -1 : 1)			2
0x3a	lzc	A := lzc(B)	leading zero count			2
0x40	jmp_z	(A == 0) ? pc += B	jump conditional			2
0x41	jmp_nz	(A != 0) ? pc += B	jump conditional			2
0x42	jmp_lz	(A < 0) ? pc += B	jump conditional			2
0x43	jmp_nlz	(A !< 0) ? pc += B	jump conditional			2
0x48	jsb	A := pc += B	jump subroutine			2
0x49	jmp	pc += B	jump			2
0x4a	gsb	A := pc := B	goto subroutine			2
0x4b	gto	pc := B	goto			2
0x4c	irt	pc :i= B	interrupt return			2
0x4e	pcr	A := pc	read pc			2
0x50	mem_r	A := mem[B]	read memory			2
0x51	mem_w	mem[B] := A	write memory			2
0x52	mem_wh	mem[B] :h= A[15:0]	write memory half			2
0x53	mem_wb	mem[B] :b= A[7:0]	write memory byte			2
0x54	mem_rhs	A :hs= mem[B]	read memory half signed			2
0x55	mem_rhu	A :hu= mem[B]	read memory half unsigned			2
0x56	mem_rbs	A :bs= mem[B]	read memory byte signed			2
0x57	mem_rbu	A :bu= mem[B]	read memory byte unsigned			2
0x58	lit	A := #	literal	4		6
0x5c	lit_hs	A :hs= #	literal half signed	2		4
0x5d	lit_hu	A :hu= #	literal half unsigned	2		4
0x5e	lit_bs	A :bs= #	literal byte unsigned	1		3
0x5f	lit_bu	A :bu= #	literal byte signed	1		3

opcode	operation	assembly	Notes	IM / Lit	AB field	bytes
0x60	add_u	A +u= B	add unsigned			2
0x61	add_us	A +us= B	add unsigned signed			2
0x62	add_su	A +su= B	add signed unsigned			2
0x63	add_s	A +s= B	add signed			2
0x64	sub_u	A -u= B	subtract unsigned			2
0x65	sub_us	A -us= B	subtract unsigned signed			2
0x66	sub_su	A -su= B	subtract signed unsigned			2
0x67	sub_s	A -s= B	subtract signed			2
0x68	sbr_u	A :u= B - A	subtract reverse unsigned			2
0x69	sbr_us	A :us= B - A	subtract reverse unsigned signed			2
0x6a	sbr_su	A :su= B - A	subtract reverse signed unsigned			2
0x6b	sbr_s	A :s= B - A	subtract reverse signed			2
0x6c	mul_u	A *u= B	multiply unsigned			2
0x6d	mul_us	A *us= B	multiply unsigned signed			2
0x6e	mul_su	A *su= B	multiply signed unsigned			2
0x6f	mul_s	A *s= B	multiply signed			2
0x70	shl_u	A << B	shift left unsigned			2
0x71	shl_s	A <<< B	shift left signed			2
0x72	rol	A <<r B	rotate left			2
0x73	pow	A := 1 << B	power			2
0x78	reg_r	A := reg[B]	read register			2
0x79	reg_w	reg[B] := A	write register			2
0x7c	and	A &= B	bitwise and			2
0x7d	orr	A |= B	bitwise or			2
0x7e	xor	A ^= B	bitwise xor			2
0x80	jmp_8z	(A == 0) ? pc += #	jump conditional immediate	1		3
0x81	jmp_8nz	(A != 0) ? pc += #	jump conditional immediate	1		3
0x82	jmp_8lz	(A < 0) ? pc += #	jump conditional immediate	1		3
0x83	jmp_8nlz	(A !< 0) ? pc += #	jump conditional immediate	1		3
0x84	jmp_8o	(A == odd) ? pc += #	jump conditional immediate	1		3

opcode	operation	assembly	Notes	IM / Lit	AB field	bytes
0x85	jmp_8no	(A != odd) ? pc += #	jump conditional immediate	1		3
0x86	jmp_8e	(A == B) ? pc += #	jump conditional immediate	1		3
0x87	jmp_8ne	(A != B) ? pc += #	jump conditional immediate	1		3
0x88	jmp_8ls	(A <s B) ? pc += #	jump conditional immediate	1		3
0x89	jmp_8nls	(A !<s B) ? pc += #	jump conditional immediate	1		3
0x8a	jmp_8lu	(A <u B) ? pc += #	jump conditional immediate	1		3
0x8b	jmp_8nlu	(A !<u B) ? pc += #	jump conditional immediate	1		3
0x8c	jsb_8	pc += #	jump unconditional immediate	1		3
0x90	mem_8r	A := mem[B+#]	read memory immediate	1		3
0x91	mem_8w	mem[B+#] := A	write memory immediate	1		3
0x92	mem_8wh	mem[B+#] :h= A[15:0]	write memory half immediate	1		3
0x93	mem_8wb	mem[B+#] :b= A[7:0]	write memory byte immediate	1		3
0x94	mem_8rhs	A :hs= mem[B+#]	read memory half signed immediate	1		3
0x95	mem_8rhu	A :hu= mem[B+#]	read memory half unsigned immediate	1		3
0x96	mem_8rbs	A :bs= mem[B+#]	read memory byte signed immediate	1		3
0x97	mem_8rbu	A :bu= mem[B+#]	read memory byte unsigned immediate	1		3
0x98	mem_i8r	A := mem[pc+#]	read memory immediate relative	1		3
0x99	mem_i8w	mem[pc+#] := A	write memory immediate relative	1		3
0x9a	mem_i8wh	mem[pc+#] :h= A[15:0]	write memory half immediate relative	1		3
0x9b	mem_i8wb	mem[pc+#] :b= A[7:0]	write memory byte immediate relative	1		3
0x9c	mem_i8rhs	A :hs= mem[pc+#]	read memory half signed immediate relative	1		3
0x9d	mem_i8rhu	A :hu= mem[pc+#]	read memory half unsigned immediate relative	1		3
0x9e	mem_i8rbs	A :bs= mem[pc+#]	read memory byte signed immediate relative	1		3
0x9f	mem_i8rbu	A :bu= mem[pc+#]	read memory byte unsigned immediate relative	1		3
0xa0	add_8u	A +u= #	add unsigned immediate	1		3
0xa1	add_8us	A +us= #	add unsigned signed immediate	1		3
0xa2	add_8su	A +su= #	add signed unsigned immediate	1		3
0xa3	add_8s	A +s= #	add signed immediate	1		3
0xa4	sub_8u	A -u= #	subtract unsigned immediate	1		3
0xa5	sub_8us	A -us= #	subtract unsigned signed immediate	1		3

opcode	operation	assembly	Notes	IM / Lit	AB field	bytes
0xa6	sub_8su	A -su= #	subtract signed unsigned immediate	1		3
0xa7	sub_8s	A -s= #	subtract signed immediate	1		3
0xa8	sbr_8u	A :u= # - A	subtract reverse unsigned immediate	1		3
0xa9	sbr_8us	A :us= # - A	subtract reverse unsigned signed immediate	1		3
0xaa	sbr_8su	A :su= # - A	subtract reverse signed unsigned immediate	1		3
0xab	sbr_8s	A :s= # - A	subtract reverse signed immediate	1		3
0xac	mul_8u	A *u= #	multiply unsigned immediate	1		3
0xad	mul_8us	A *us= #	multiply unsigned signed immediate	1		3
0xae	mul_8su	A *su= #	multiply signed unsigned immediate	1		3
0xaf	mul_8s	A *s= #	multiply signed immediate	1		3
0xb0	shl_8u	A << #	shift left unsigned immediate	1		3
0xb1	shl_8s	A <<< #	shift left signed immediate	1		3
0xb2	rol_8	A <<r #	rotate left immediate	1		3
0xb3	pow_8	A := 1 << #	power immediate	1		3
0xb8	reg_8r	A := reg[#]	read register immediate	1		3
0xb9	reg_8w	reg[#] := A	write register immediate	1		3
0xbc	and_8	A &= #	bitwise and immediate	1		3
0xbd	orr_8	A |= #	bitwise or immediate	1		3
0xbe	xor_8	A ^= #	bitwise xor immediate	1		3
0xc0	jmp_16z	(A == 0) ? pc += #	jump conditional immediate	2		4
0xc1	jmp_16nz	(A != 0) ? pc += #	jump conditional immediate	2		4
0xc2	jmp_16lz	(A < 0) ? pc += #	jump conditional immediate	2		4
0xc3	jmp_16nlz	(A !< 0) ? pc += #	jump conditional immediate	2		4
0xc4	jmp_16o	(A == odd) ? pc += #	jump conditional immediate	2		4
0xc5	jmp_16no	(A != odd) ? pc += #	jump conditional immediate	2		4
0xc6	jmp_16e	(A == B) ? pc += #	jump conditional immediate	2		4
0xc7	jmp_16ne	(A != B) ? pc += #	jump conditional immediate	2		4
0xc8	jmp_16ls	(A <s B) ? pc += #	jump conditional immediate	2		4
0xc9	jmp_16nls	(A !<s B) ? pc += #	jump conditional immediate	2		4
0xca	jmp_16lu	(A <u B) ? pc += #	jump conditional immediate	2		4

opcode	operation	assembly	Notes	IM / Lit	AB field	bytes
0xcb	jmp_16nlu	(A !<u B) ? pc += #	jump conditional immediate	2		4
0xcc	jsb_16	pc += #	jump unconditional immediate	2		4
0xd0	mem_16r	A := mem[B+#]	read memory immediate	2		4
0xd1	mem_16w	mem[B+#] := A	write memory immediate	2		4
0xd2	mem_16wh	mem[B+#] :h= A[15:0]	write memory half immediate	2		4
0xd3	mem_16wb	mem[B+#] :b= A[7:0]	write memory byte immediate	2		4
0xd4	mem_16rhs	A :hs= mem[B+#]	read memory half signed immediate	2		4
0xd5	mem_16rhu	A :hu= mem[B+#]	read memory half unsigned immediate	2		4
0xd6	mem_16rbs	A :bs= mem[B+#]	read memory byte signed immediate	2		4
0xd7	mem_16rbu	A :bu= mem[B+#]	read memory byte unsigned immediate	2		4
0xd8	mem_i16r	A := mem[pc+#]	read memory immediate relative	2		4
0xd9	mem_i16w	mem[pc+#] := A	write memory immediate relative	2		4
0xda	mem_i16wh	mem[pc+#] :h= A[15:0]	write memory half immediate relative	2		4
0xdb	mem_i16wb	mem[pc+#] :b= A[7:0]	write memory byte immediate relative	2		4
0xdc	mem_i16rhs	A :hs= mem[pc+#]	read memory half signed immediate relative	2		4
0xdd	mem_i16rhu	A :hu= mem[pc+#]	read memory half unsigned immediate relative	2		4
0xde	mem_i16rbs	A :bs= mem[pc+#]	read memory byte signed immediate relative	2		4
0xdf	mem_i16rbu	A :bu= mem[pc+#]	read memory byte unsigned immediate relative	2		4
0xe0	add_16u	A +u= #	add unsigned immediate	2		4
0xe1	add_16us	A +us= #	add unsigned signed immediate	2		4
0xe2	add_16su	A +su= #	add signed unsigned immediate	2		4
0xe3	add_16s	A +s= #	add signed immediate	2		4
0xe4	sub_16u	A -u= #	subtract unsigned immediate	2		4
0xe5	sub_16us	A -us= #	subtract unsigned signed immediate	2		4
0xe6	sub_16su	A -su= #	subtract signed unsigned immediate	2		4
0xe7	sub_16s	A -s= #	subtract signed immediate	2		4
0xe8	sbr_16u	A :u= # - A	subtract reverse unsigned immediate	2		4
0xe9	sbr_16us	A :us= # - A	subtract reverse unsigned signed immediate	2		4
0xea	sbr_16su	A :su= # - A	subtract reverse signed unsigned immediate	2		4
0xeb	sbr_16s	A :s= # - A	subtract reverse signed immediate	2		4

opcode	operation	assembly	Notes	IM / Lit	AB field	bytes
0xec	mul_16u	A *u= #	multiply unsigned immediate	2		4
0xed	mul_16us	A *us= #	multiply unsigned signed immediate	2		4
0xee	mul_16su	A *su= #	multiply signed unsigned immediate	2		4
0xef	mul_16s	A *s= #	multiply signed immediate	2		4
0xfc	and_16	A &= #	bitwise and immediate	2		4
0xfd	orr_16	A |= #	bitwise or immediate	2		4
0xfe	xor_16	A ^= #	bitwise xor immediate	2		4