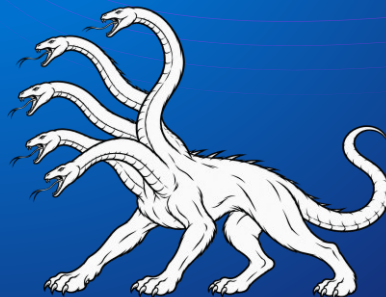


Hydra 4 architecture overview

Dave Gillespie, daveg@synaptics.com

Last updated: October 29, 2026
(Originally written in January 2022)



 synaptics®

WORK IN PROGRESS!

This is a preview of an *internal document* that we use *within Synaptics* to describe our Hydra vector arithmetic processor

We are preparing a more public-friendly version, without the references to past versions, specific chip projects, and other stuff that is only of interest within Synaptics

The cleaned-up document, and the *Hydra User's Guide* itself, as well as the analogous materials for our Chimera processor, will shortly be available here:

- github.com/synaptics-hydra
- github.com/synaptics-chimera

Until then, you are welcome to preview here the “raw” version ...

Thank you!

– Dave Gillespie, daveg@synaptics.com

Hydra 4

Architecture overview

- Introduction
 - *What, why, where*
- Using Hydra
 - *Examples, registers, instructions, types, modes, ...*
- Hardware and implementation
 - *Block diagrams, theory of operation, interesting details*

Hydra 4

Architecture overview

- **Introduction**
 - **What is it**
 - Why we need it
 - Where it came from
- Using Hydra
- Hardware and implementation

Hydra 4 at a glance

The Hydra arithmetic coprocessor adds powerful vector arithmetic capability to a small embedded MCU (such as SiFive RISC-V E, ARM Cortex-M, or Synaptics Chimera)

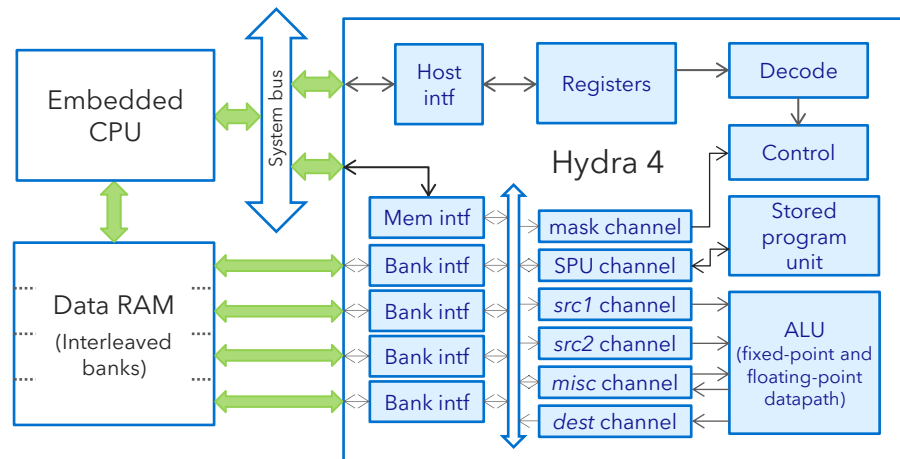
- It adapts “big CPU” vector capabilities to the scale, power, cost, and size of embedded systems

Synaptics products use Hydra for many things, such as:

- Accelerating touch algorithms
- Signal processing
- Fingerprint image processing + matching

Hydra 4 significantly accelerates many operations including:

- Arithmetic on vectors, matrices, and images
- Matrix multiply
- Linear algebra
- FFTs
- 1D and 2D convolutions and median filters
- Neural networks
- Encryption and hashing
- Pseudo-random numbers
- Trig functions
- Bit operations
- Packing data into few bits
- And much more



Hydra

Hydra is an arithmetic accelerator for small Synaptics chips

It works alongside a chip's CPU such as Chimera, RISC-V, ARM, or Xtensa

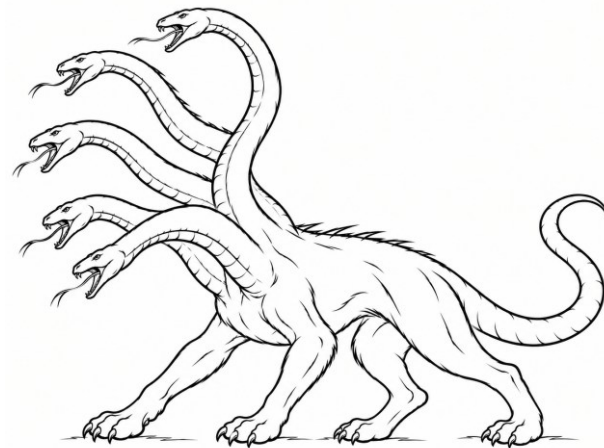
Hydra does bulk calculations on vectors/arrays/matrices/images of data

Hydra version 1, 2, or 3 is found in many "high-performance" Synaptics touch chips

- DIG2401 and DIG2403-DIG2408 (touch controllers)
- TD7800, TD7850, TD7875 (automotive TDDI)
- B1422 "Prometheus" and B1442 "Triton" (fingerprint with match-in-sensor)

Hydra 4, the latest version, appears in our newest chips

- First up: The SB7900 "SmartBridge" automotive chip and the DIG2409 "Byrd" and DIG2411 "Denmarke" touch controllers



"Hydra" of Greek mythology

- The host CPU is an "E76" RISC-V core from SiFive
- The E76 connects to the chip's data RAM as tightly-coupled ("DTIM") memory
- Hydra connects to the CPU as a register-mapped peripheral via the AHB bus
- Hydra also connects directly to the same data RAM
- Optionally, Hydra can drive the data bus to access other parts of the chip (omitted on Byrd A0)



Vector processing on a small budget

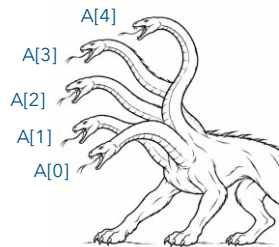
Hydra is a **vector processor**: It operates on whole arrays (“vectors”) of data at a time

One instruction for a vector processor is like a complete loop in a program

The single instruction “VADD A, B, C” might do all this:

```
for (i = 0; i < N; i++) { A[i] = B[i] + C[i]; }
```

B[0]	B[1]	B[2]	B[3]	B[4]	B[5]	B[6]	B[7]	...	B[N-1]
+	+	+	+	+	+	+	+		+
C[0]	C[1]	C[2]	C[3]	C[4]	C[5]	C[6]	C[7]	...	C[N-1]
↓	↓	↓	↓	↓	↓	↓	↓		↓
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	...	A[N-1]



A big vector processor (e.g., x86 SSE / AVX, ARM Neon, or RISC-V “V”) will have wide registers and multiple arithmetic units to do all these additions at once, in parallel

Down in our price range, a vector processor must work through the vector iteratively

But it can work much more efficiently than ordinary CPU firmware can do

DIG2409 “Byrd”

Byrd has dramatically higher computation needs (“MIPS”) than previous touch chips

- Foldable phones $\Rightarrow$ Touch images twice as large
- Gaming phones $\Rightarrow$ Crazy high frame rates!
- AFE improvements $\Rightarrow$ FFTs per-frame or per-burst

Byrd replaces our Chimera CPU with a powerful RISC-V CPU: SiFive’s E76

- An 8-stage pipeline that can clock at 400 MHz (in TSMC’s 40nm Embedded Flash process)
- Full dual-issue superscalar operation, optimized for computational performance
- It’s a beast – at least 3x bigger than either Chimera or Hydra, but we needed the MIPS!

Even the E76 was barely enough to meet Byrd’s anticipated MIPS needs ...

- So, once again, we augment the main CPU with Hydra
- A perfect time to refresh and modernize Hydra $\Rightarrow$ “Hydra 4.0”

SB7900 “SmartBridge”

The SB7900 chip, for large automotive touchscreens, also has Hydra 4

- SB7900 shares a lot of Byrd’s architecture, again using E76 as its main CPU

Hydra 4 was included in SB7900 for more speculative reasons

- The auto industry cares much more about safety and robustness
- It is beginning to get very serious about cybersecurity, too
- Hydra 4’s AES, public-key, and SHA2 cryptographic functions should prove invaluable for these

In fact, the SB7900 chip taped out before Byrd

- Byrd is too expensive for any but the most demanding applications
- Byrd was shelved for a period, but it eventually taped out in 2024
- In any event, most modern Synaptics touch chips have shipped with Hydra, and our future ones will ship with Hydra 4

Hydra 4

A modern vector coprocessor that is ***built for small chips like ours***

A ***32-bit-oriented*** architecture that fits well with ARM, RISC-V, and modern Chimera

A ***clean*** design that is easy to use

A ***fresh*** implementation that is easy to maintain

Ground rules:

- Stay true to Hydra's and Chimera's "nimble" roots: Small, austere, efficient, flexible, agile
- Hydra is not trying to be as fast as RISC-V's big "V" extension or ARM's Neon
 - But it is significantly faster than doing the same operations in microcontroller firmware
 - And Hydra 4 is a lot faster than Hydra 3
- Keep it small and cheap: Roughly comparable in size to Hydra 3
- It is okay to break compatibility with Hydra 1/2/3, but make it easy to port existing firmware
- Fulfill longstanding feature requests from Hydra's past users
- Look toward new application areas such as ultra-low-power small neural networks

First implementation: Hydra 4.0

Hydra 4.0 is a ground-up rewrite of Hydra's Verilog RTL and test suite

- The latest version is now Hydra 4.2.1

The *Hydra 4 User's Guide* defines quite a lot of functionality

- (Available soon on github.com/synaptics-hydra)
- We implemented most of this spec – as much as was possible – in time for SmartBridge tapeout
- Starting with the things needed for today's algorithms and directly anticipated needs
- Beyond that, we will continue adding features and improvements as Hydra 4.x evolves

Hydra 4 aims to be “only fractionally larger” than Hydra 3

- Byrd and Keys synthesis results show Hydra 4.0 is 10-15% larger than fully configured Hydra 3.3

Hydra 4 is able to clock as fast as Byrd's SiFive E76 host CPU (~400 MHz in 40nm EF)

- Hydra shouldn't be what limits our clock speed
- Hence, Hydra 4 is much more extensively pipelined than Hydra 3

Major improvements vs. Hydra 3.3

Much faster (higher max MHz, and also does much more *per* MHz)

- Hydra 4 is twice as fast (per MHz) for floating-point and 32-bit integer computations
- Multiple 32-bit-wide channels to RAM
- (Potentially) less instruction start/stop overhead

Much easier to use: No hairy macros or “Hydra assembler” required

Easy to integrate in hardware, with a clean, maintainable, 32-bit implementation

Much more flexible programming model:

- Instructions can mix any combination of integer, fixed, float, and packed data types and any combination of addressing modes
- FFTs, median filters, and sorting are now affordable standard features

Many new capabilities:

- | | | |
|----------------------------|-------------------------|----------------------------|
| • Complex numbers | • Half-precision floats | • Sparse vector masks |
| • Pseudorandom numbers | • Packed ints & floats | • Generalized convolutions |
| • Neural networks | • Pattern generation | • Generalized matrices |
| • Cryptographic algorithms | • Simplified SPU | • Multi-threaded usage |

Detailed improvements: General architecture

	Hydra 3.3	Hydra 4.2
Memory interface	Direct-style only	Choice of direct, AHB, or mix
Width of memory interfaces	16 bits	32 bits
Max memory data bandwidth	64 bits/cycle	128 bits/cycle
Memory address range	Up to 2^{19} (512K) bytes	Up to 2^{32} (4G) bytes
Host CPU interface	Chimera-style only	Chimera, AHB, APB, or direct
Width of host CPU interface	16 bits	32 bits
Width of arithmetic datapath	16 bits	32 bits
Hardware integration	Tricky timing and interfacing	Much easier
Compatibility	Mostly backward compatible through 10 years to Hydra 1.0 (in both the RTL and the user-visible architecture)	A fresh, clean new design inside and out (A "Hydra 3 compatibility" mode assists firmware porting)

Detailed improvements: Cost and performance

	Hydra 3.3	Hydra 4.2
Size of Hydra core	Approx. same as Chimera 4; Approx. 1/2 of ARM Cortex-M4; Approx. 1/3 of SiFive E76	Similar; fractionally larger
Maximum clock rate (40nm EF)	< 300 MHz?	At least 400 MHz
Speed of 16-bit integer math	1 result per cycle	At least 1 result per cycle
Speed of 32-bit integer math	1 result per 2 cycles	1 result per cycle (2x faster)
Speed of floating-point math	1 result per 2 cycles	1 result per cycle (2x faster)
Time for a typical float vector operation of size N	$2N$ cycles + small overhead	N cycles + small overhead
Float multiply-add pipeline	4 stages	6 stages
Idle power (when not in use or between vector operations)	Nearly zero (leakage only)	Same
Power while operating	(Small; we never measured it)	Probably similar

Detailed improvements: Programming model

	Hydra 3.3	Hydra 4.2
Overall usability for firmware	Hard to use	Much easier to use
"HOCA_xxx" macro method	Ugly; used in most firmware	Only in compatibility mode
"Hydra assembler" method	Less ugly; in newer firmware	Only in compatibility mode
Easy function-style notation	No	Yes
Instruction opcodes	16 bits wide	32 bits wide
Instruction operands	Ad-hoc	Flexible vector-register model
Instruction buffering	1 level + 1 partial level	2 levels
CPU interrupts	6 interrupt flags $\Rightarrow$ 1 CPU intr	7 interrupt flags $\Rightarrow$ 1 CPU intr
Automatic handshaking to simplify register access by FW	No	Yes
Use by multi threads/tasks/cores	No	Yes
Error reporting	Basic	Detailed diagnostic info
Performance counter	No	Yes
Stored program unit	Hard to use	Flexible and easy to use

Detailed improvements: Arithmetic

	Hydra 3.3	Hydra 4.2
Fully IEEE 754-compliant floating-point arithmetic	Not quite	Yes
Support for NaN values	No	Yes
Support for denorm values	No	Yes
Fused multiply-adds	No	Yes
Correctly rounded division	No	Yes
Support for half-precision	No	Yes
Fixed-point arithmetic	16-bit only	8-, 16-, and 32-bit
Mixed-format arithmetic	No	Yes
Packed K-bit floats	No	Yes (using Posits)
Square roots	Yes (not IEEE-rounded)	Same
Support for sin/cos/log/exp/...	No	Yes
Support for frexp/ldexp/...	No	Yes

Detailed improvements: Memory addressing

	Hydra 3.3	Hydra 4.2
Linear/stride addressing	Yes	Yes
Scatter/gather indexing	Yes	Yes
Simple matrix addressing	Limited to 256 columns; tricky and confusing to use	Unlimited and easy to use
Transposed matrices	Limited	Flexible and general
General matrix row/col strides	No	Yes
Repeated-row/col matrices	No	Yes
Nested repeating patterns	No	Yes
Circular addressing	Very limited	Flexible and general
Bit-level data packing	Limited	Flexible and general
Pattern-mode source data	No	Yes
Memory scalar operands	No	Yes
Vector mask	Yes	More flexible, easier to use

Detailed improvements: Special operations

	Hydra 3.3	Hydra 4.2
Fast Fourier Transforms (FFT)	Simple/basic	Faster and more flexible
Convolutions (1D and 2D)	Confusing and hard to use	Easier and more flexible
Matrix multiply	Yes	Easier and more flexible
Median filters (1D and 2D)	Optional	Yes
Basic min/max operations	Yes	Yes
Find local minima/maxima (accelerates IFP segmentation)	No	Yes
Row/column-wise min/max	No	Yes
Complex number arithmetic	No	Yes
Classify special float values	No	Yes
Integer $\log_2$ and variable shifts	No	Yes
Byte swap and bit reverse	No	Yes
Sorting/selection	Limited	Faster and more flexible
Block move or fill	Copy 32 bits per cycle	Copy 64 bits per cycle

Detailed improvements: Neural networks/ML/AI

	Hydra 3.3	Hydra 4.2
Byte data types	No	Yes
BFLOAT16 data type	No	Yes
Posit data type	No	Yes
Byte-granular convolutions	No	Yes
SIMD operations on byte data	No	Yes (Hydra 4.2+)
Packed convolution weights	No	Yes
Packed sparse weights	No	Yes (limited)
Min/max convolutions	No	Yes
Simple minpool/maxpool	No	Yes
General pooling convolutions	No	(Planned; Hydra 4.2 omits it)
ReLU or ReLU6 in 1 instruction	Yes	Yes
Reasonably fast exp(x)	No	Yes
Other ML-inspired features	No	Ongoing development...

Detailed improvements: Cryptographic features

	Hydra 3.3	Hydra 4.2
AES standard encrypt/decrypt	No	Yes (~10 cycles per byte)
AES/GCM carry-less multiply	No	Yes (~8 N^2 cycles for CLMUL)
XXTEA encrypt/decrypt	No	Yes (~10 cycles per byte)
SHA-256/-224 standard hash	No	Yes (~5 cycles per byte)
CRC32/CRC16 hash	No	Yes (2 cycles per byte)
Pseudorandom numbers	No	Yes ($\frac{1}{2}$ cycle per byte)
RSA/DSA public-key arithmetic	Limited	Yes (N^2 cycles to multiply two $32N$ -bit-wide integers)

Note: In areas such as Neural Networks and Cryptography, remember that Hydra is not trying to be as powerful as a big, expensive, dedicated NPU, GPU, or Crypto IP core

- It aims to deliver significantly better performance than pure firmware can do ...
- ... at modest, incremental hardware cost: Cheap enough to include it always “just in case”
- If we *know* we need (say) a big NPU, and our market justifies its cost, fine: We can get one that meets the need
- Hydra includes these features for a different reason: *Cheaply* looking ahead to meet *speculative* future needs
- This is important since our chips often live on for many years, eventually to be used in many products

“Hydra Junior”

Hydra 4 can be built in a way that significantly reduces its area and cost, in return for a decrease in functionality

- ``define HYDRA_JUNIOR`

The Hydra Junior configuration removes features that are rarely used or not required for our basic touch firmware functionality:

- Slower FFTs, median filters, sorting, neural networks, and special math functions
- No cryptographic encryption or hashing; no random number generation
- No packed data, half-precision floats, or Posit data types
- No Stored Program Unit, multi-threaded operation, or Hydra 3 Compatibility Mode
- ... and various other cost savings

Hydra 4 Junior is over 40% smaller than the full configuration of Hydra 4

- Full Hydra 4.2 is ~108K gates, and Hydra 4.2 Junior is ~61K gates, in TSMC 40nm EF
- (These are base synthesis numbers when *not* pushed for the fastest clock rate)
- We first used Hydra 4.1 Junior in the cost-reduced DIG2411 “Denmarke” chip

“Hydra Junior”

Instruction	Kept?
VABS	Yes
VADD	Yes
VADD3	—
VAES	—
VAMAX	Yes
VAMAXI	Yes
VAMAXV	Yes
VAMIN	Yes
VAMINI	Yes
VAMINV	Yes
VAND	Yes
VANDV	—
VANDN	Yes
VASUB	—
VAVG	—
VBOOL	—
VBREV	—
VCALL	—
VCBIN	—
VCLASS	—
VCLMUL	—
VCMAC	—
VCMP	Yes
VCMPM	Yes

Instruction	Kept?
VCMPSEL	—
VCOUNT	—
VCRC	—
VCSIGN	Yes
VDIV	—
VDMAC	Yes
VFIND	—
VFINDT	—
VHAM	Yes
VLOG2	—
VMAC	Yes
VMADD	Yes
VMAX	Yes
VMAXI	Yes
VMAXV	Yes
VMED	—
VMEDR	—
VMIN	Yes
VMINI	Yes
VMINV	Yes
VMINMAX	—
VMOV	Yes
VMOV2	—
VMOV4	—

Instruction	Kept?
VMOVC	Yes
VMOVC2	—
VMOVG	—
VMOVL	Yes
VMOVN	Yes
VMOVS	—
VMOVX	Yes
VMOVX2	—
VMSORT	—
VMSUB	Yes
VMSUBN	Yes
VMSUBR	Yes
VMUL	Yes
VMULH	—
VMULN	Yes
VNAND	Yes
VNEG	Yes
VNEXT	—
VNOP	Yes
VNOR	Yes
VNOT	Yes
VOMAC	Yes
VOR	Yes
VORV	—

Instruction	Kept?
VORN	Yes
VPOP	Yes
VPSORT	—
VRAND	—
VRCPE	Yes
VRCPE2	—
VRSQRTE	Yes
VRSQRTE2	—
VSEL	Yes
VSELQ	—
VSHA2H	—
VSHA2M	—
VSHL	Yes
VSHR	Yes
VSTART	—
VSUB	Yes
VSUBH	—
VSUM	Yes
VXCH	—
VXMADD	—
VXNOR	Yes
VXOR	Yes
VXORV	—
VXXTEA	—

Addressing mode	Kept?
hydra_vector()	Yes
hydra_scalar_int()	Yes
hydra_scalar_float()	Yes
hydra_mscalar()	Yes
hydra_rscalar()	—
hydra_strided()	Yes
hydra_ostride()	Yes
hydra_circular()	Yes
hydra_[t]matrix()	Yes
hydra_rrmatrix()	Yes
hydra_rcmatrix()	Yes
hydra_[t]gmatrix()	Yes
hydra_packed()	—
hydra_bfly()	—
hydra_cosine()	—
hydra_sine()	—
hydra_complex()	—
hydra_neg_vector()	—
hydra_abs_vector()	—
hydra_half_vector()	—
hydra_double_vector()	—
hydra_bitrev()	—
hydra_nested()	—
hydra_offset_circ()	—
hydra_null_vector()	—
hydra_null_scalar()	—
hydra_pattern(), etc.	Yes
Indexed addressing	—
Singleton scalar	Yes

Operand or modifier	Kept?
hydra_set_v1()	Yes
hydra_set_vma()	Yes
hydra_set_vpc()	—
HYDRA_V0, _V1, _V2, _V3	Yes
HYDRA_V4, _V5, _V6, _V7	—
HYDRA_ZERO, _ONE	Yes
HYDRA_IGN, _FGN	Yes
Scalar accumulation mode	Yes
Vector cumulative mode	Yes
HYDRA_CONVO	Yes
HYDRA_MM	Yes
HYDRA_IF	Yes
HYDRA_ELSE	Yes
hydra_mask_if()	Yes
hydra_mask_else()	Yes
HYDRA_FV	—
HYDRA_FM	Limited
hydra_mask_gen()	Yes
hydra_mask_zero()	Yes
hydra_mask_nzero()	Yes
hydra_mask_other()	—
hydra_mask2_any()	Limited
HYDRA_XM	Limited
HYDRA_NOSE	Yes
HYDRA_RN, _RZ, _RD, etc.	Yes
HYDRA_J	Ignored
HYDRA_NJ	Ignored

Hydra 4

Architecture overview

- **Introduction**
 - What is it
 - **Why we need it**
 - Where it came from
- Using Hydra
- Hardware and implementation

Refreshing Hydra 3

Hydra versions 1.0–3.3 have served us well in a variety of touch, display, and fingerprint chips at Synaptics

- Hydra is present in many of our most successful chips, but not in some of the highest-volume, lowest-cost chips
- This makes it hard to work out the total number of Hydras produced so far ...
- ... But most likely, well over **a billion chips** have shipped to date with some version of Hydra

Still, Hydra 3 was showing its age

Hydra 3.3 is still largely backward compatible to the original ten-year-old Hydra 1.0

- An architecture based on guesswork from days when our firmware was much less mature
- It was built around the 16-bit-centric architecture of older Chimera CPUs, which puts it out of step with modern CPUs including our own Chimera 4.2

⇒ ***Time for a thorough refresh***

Why refresh Hydra? (1)

Hydra is **valuable**: For example, Prometheus (and Triton) fingerprint matching

- Its image processing module runs **6x** faster with Hydra 3 than it does in pure firmware on our Prometheus chip's Tensilica Xtensa CPU
- ... even though Prometheus has a maximally-configured >200K-gate 32-bit Xtensa at 192 MHz

⇒ *Match-in-sensor would not have been feasible without Hydra or something like it*

Why refresh Hydra? (2)

Hydra is **valuable**: For example, Prometheus (and Triton) fingerprint matching

Hydra is **strategically valuable**: For example, noise countermeasures in touch firmware

- With Hydra 2, we solved OLED display noise “in firmware” instead of with a one-year chip spin
 - (Yes, I know, oversimplifying...)
- OLED display noise was not known yet, and our algorithms for it were not invented yet, back when we designed our DIG240x series chips with Hydra
- We had originally developed Hydra for something else
- We carried it forward into those chips “just in case” new tough problems might need more computational power
- That bet paid off when OLEDs came along

⇒ *A chip conceived and built with only enough strength for our pre-OLED firmware would never have had the horsepower to handle OLED displays*

Why refresh Hydra? (3)

Hydra is **valuable**: For example, Prometheus (and Triton) fingerprint matching

Hydra is **strategically valuable**: For example, OLED touch firmware

Hydra is **still strategically valuable**:

- Sure, future chips can (*and should!*) solve things like OLED noise in a more fundamental way
- Maybe they will have faster CPUs or better AFEs
- But without Hydra or something like it, we would have nothing left “in our back pocket” for the *next* OLED-like challenge

⇒ *And that is Synaptics' bread and butter!*

- Customers look to us because our technology has the strength and ingrained flexibility ...
- ... to “fix in firmware” with speed and agility ...
- ... even the toughest problems

Why refresh Hydra? (4)

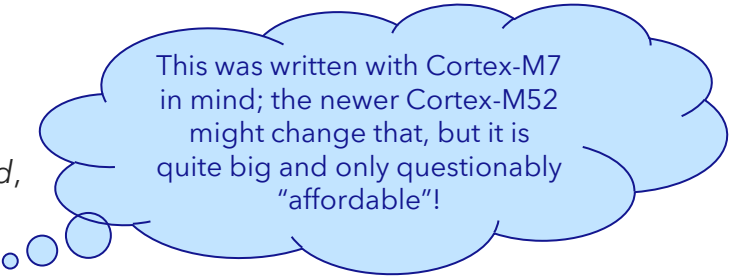
Hydra is **valuable**: For example, Prometheus (and Triton) fingerprint matching

Hydra is **strategically valuable**: For example, OLED touch firmware

Hydra is **still strategically valuable**:

Hydra is **necessary**

- Even our *existing* firmware is hard to run without it
- SiFive's E76 is maxed out by Byrd's needs
 - ... even though it is much larger than Chimera + Hydra combined
 - Even if we *can* clock a CPU that fast, it is not clear we always *should*, from a power/noise/cost perspective
- Also, no affordable ARM processor can run as fast as E76



This was written with Cortex-M7 in mind; the newer Cortex-M52 might change that, but it is quite big and only questionably "affordable"!

⇒ *We don't know how to build an ARM-based touch controller that can run our full firmware unless we pair the ARM with Hydra or something like it*

Why refresh Hydra? (5)

That's all great, but Hydra 3 itself was getting old ...

Our modern chips are **32-bit-oriented**

- Synaptics' Chimera and Hydra began as small 16-bit processors
- But Chimera 4.2 now supports 32-bit data, 32-bit addressing, and 32-bit interfaces to RAM (used today in DIG2405 / DIG2407 / DIG2408 "Vulcan" and "Keys")
- Some customers want a 32-bit commercial CPU (e.g., one customer specified ARM Cortex-M4)
- Byrd and SmartBridge use a 32-bit RISC-V CPU, and future chips may use it too

Hydra 3 still has a 16-bit data path with 16-bit addressing and 16-bit RAM interfaces

⇒ *This is needlessly old-fashioned!*

- Hydra 3 reads and writes floating-point numbers in two 16-bit halves, sequentially
 - Its float operations run at half speed, even though our memories now support 32-bit-wide access
- Hydra 3's 16-bit architecture has a hard time with > 64KW of RAM
- It is awkward to use when connected to modern 32-bit processors
- And it's awkward to connect! It uses Chimera-specific buses; no AHB or APB

Why refresh Hydra? (6)

Hydra 3 is **too hard to use**

Recent versions have made improvements, but it is still clumsy

- It is hard to construct the numeric “opcode”
- Parameters of the operation mingle in tricky ways
- Some parameters must be specified in multiple places
- Lots of sticky modes to watch out for; lots of restrictions to navigate
- Not a very “natural” interface to the firmware

One Hydra 3 VSUB instruction
(in real IFP firmware) ↓

- Meanwhile, we should be making Hydra **easier** and more accessible:
Hydra’s best early power users are no longer here

```
void subtractFitDeltaCol(int16 *col, uint16 colLen, int16 *fit, uint16 stride, uint16 *flags)
{
    HYDRA.MCFG.v = 0;
    HYDRA.CFG.HYDRA_FLAG_EN = (uint16)0; //to be removed if not necessary. See also ADVDIG-237 and ADVDIG-250.
    HYDRA.VFA.v = (uint16)flags; //unfortunately this is not optional at this stage
    HYDRA.VL.v = (uint16)colLen; // total number of elements in result
    HYDRA.S0.v = (uint16)stride; // src1/src2/result address stride
    HYDRA.INST_SRC2.v = (uint16)*fit;
    HYDRA.INST_SRC1.v = (uint16)col;
    HYDRA.INST_DEST.v = (uint16)col; //col
    HYDRA.INST_CODE.v = HOCA_11_15_SUB |
        HOCA_10_SRC2_SCALAR | HOCA_9_SRC2_ADDRSLR_IMM | HOCA_7_8_SRC2SCLR_IMM |
        HOCA_6_SRC1_ADDRVEC_DIR | HOCA_4_5_SRC1VEC_STRIDE_S00 |
        HOCA_3_DST_ADDRVEC_DIR | HOCA_1_2_DSTVEC_STRIDE_S00 |
        HOCA_0_VECLEN_VXX0;
    if_hydra_inst_fifo_notempty_chimera_goto_sleep();
}
```

Why refresh Hydra? (7)

Hydra 3 is **hard to maintain**

- The RTL was originally written by Kam Li in 2012 w/architectural guidance from Mark Pearce and me
- Hydra has evolved dramatically since then, but Hydra 3.3 is still built upon that old RTL
- Mark left Synaptics in 2014, and Kam left in 2017
- I took over Hydra, finished the work for Prometheus, and maintained and developed it since then
- I cultivated Cody Czesler in Rochester to co-own Hydra; we lost him when RDC closed in 2018
- Today, I handle all of Hydra's architecture, design, documentation, testing, and some tools by myself
- This is not "scalable"!
- So it would be great if Hydra's internals were easy to teach and to work with; sadly ...

Hydra 3 is **hard to work with**

- Its Verilog RTL is quirky and obscure "spaghetti code": People find it impenetrable
- I took the time to learn it, but even I have a hard time implementing some feature requests
- It was getting hard to push Hydra 3 any further

Why not just buy it?

Vector processors are an old idea – but getting popular again

However, even today's "small" commercial vector processors are *really big*

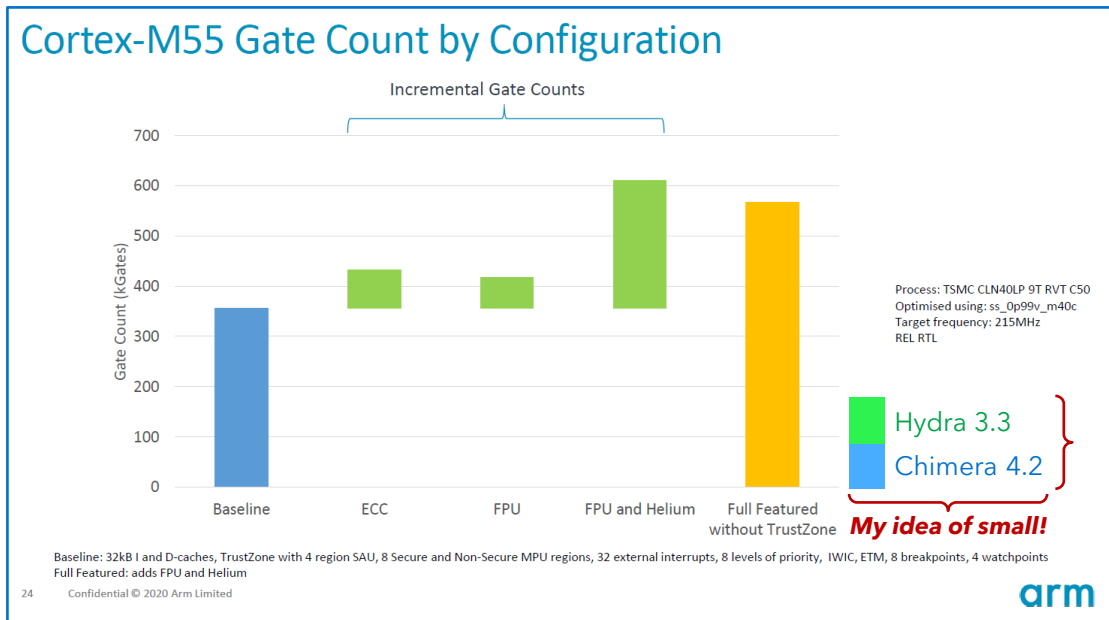
- ARM recently released Cortex-M55 with "Helium / MVE" M-profile Vector Extension
- Yay! But ...

Hydra 3 is less than 100K gates

- Similar in size to Chimera
- *Together* they are only as big as ARM M4 or SiFive RISC-V E34

SiFive's E76 is quite a bit larger than ARM M4, and Hydra 4 is somewhat larger than Hydra 3

- But the principle is the same: Our chips are small; industry standard vector CPUs are not!



(To be fair, ARM released a smaller Cortex-M52 since I wrote this in 2022)

Why not just buy it in RISC-V?

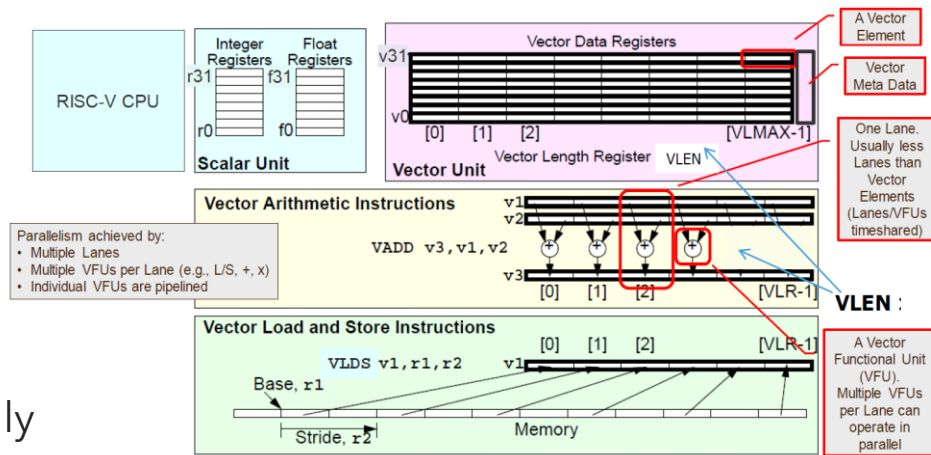
RISC-V offers a “V” vector extension

- An elegant state-of-the-art vector architecture
- Scalable in principle from “large” to “small” implementations
- But even its smallest configuration is still pretty big: 32 × 128-bit vector registers, etc.

Vendors like SiFive and Andes offer “V” mainly in their largest CPUs (> 1 million gates)

- With 512-bit-wide registers that can hold 16 floats at once
- And enough cache bandwidth to load or store 512 bits on every cycle
- And 16 floating-point multipliers and adders in a row, all operating in parallel
- Cool stuff, but way out of our league!

So, in today’s world, for our tiny chips, it still makes sense to build and develop Hydra



From a webinar on andestech.com

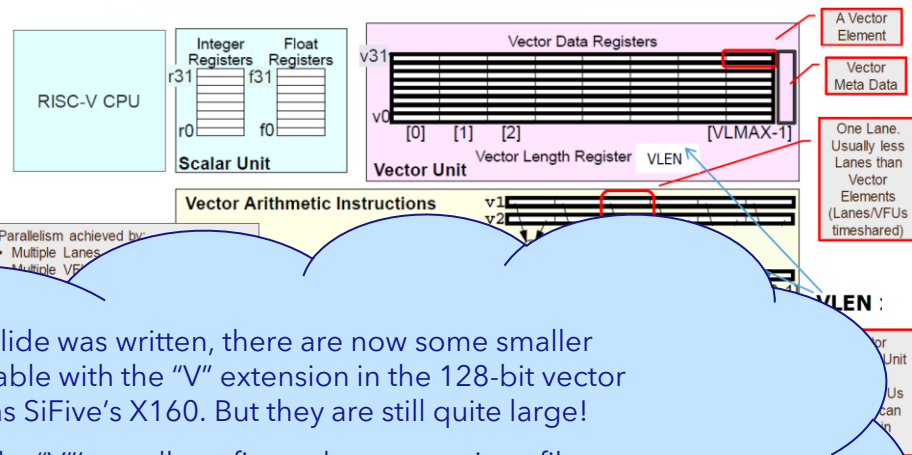
Why not just buy it in RISC-V?

RISC-V offers a "V" vector extension

- An elegant state-of-the-art vector architecture
- Scalable in principle from "large" to "small" implementations
- But even its smallest configuration is still pretty big: 32 × 128-bit vector registers

Vendors like SiFive and Andes offer it in their largest CPUs (> 1 m

- With 512-bit-wide registers the
- And enough cache bandwidth
- And 16 floating-point multipliers
- Cool stuff, but way out of our league

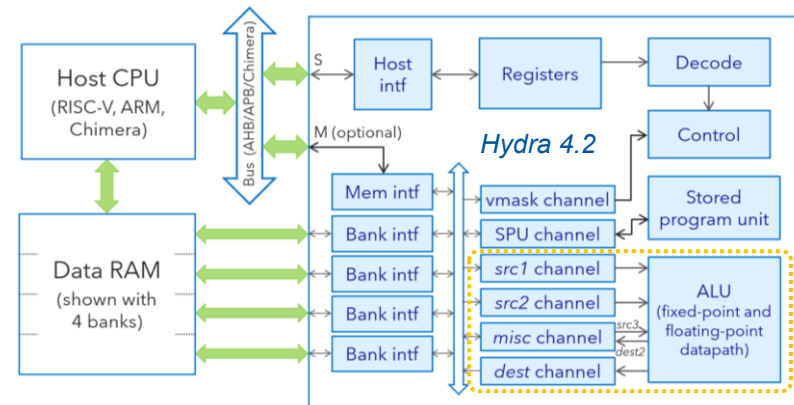
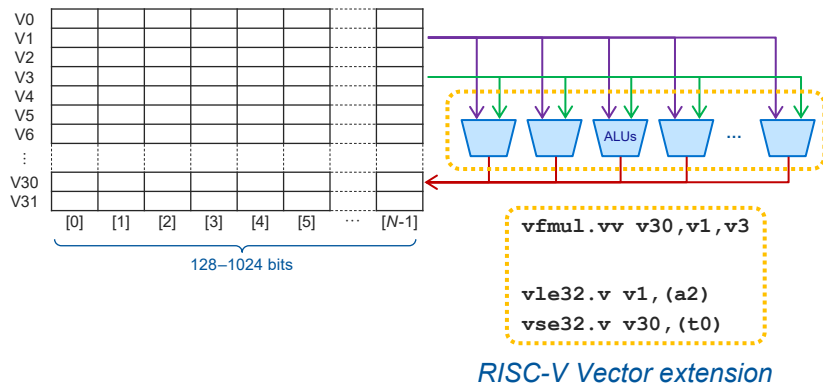


Since this slide was written, there are now some smaller CPUs available with the "V" extension in the 128-bit vector size, such as SiFive's X160. But they are still quite large!

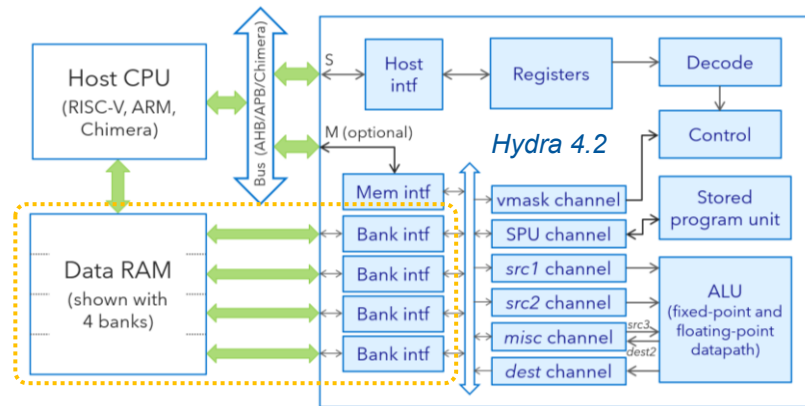
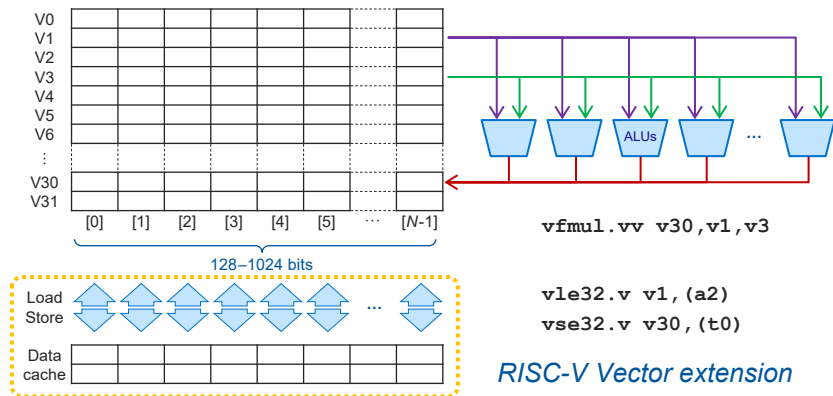
For example, "V"'s small-configured vector register file alone contains $32 \times 128 = 4096$ flip-flops, which is more flip-flops than in the entire Hydra 4.2 non-Junior core including all the registers, FIFOs, pipelines, arithmetic functions, address generators, controllers, etc., all together

So, in today's world, for our tiny chips, it still makes sense to build and develop Hydra

Big: RISC-V "V", ARM Neon / SVE / MVE	Small: Hydra coprocessor
Many wide vector registers inside the CPU	Vector data resides in the main data RAM
Many ALUs / FPUs operating in parallel	Stream the data through one pipelined ALU / FPU
Vector Load + Vector Arithmetic + Vector Store are separate instructions	Multiple DMA channels load and store data in parallel with the ALU's operation
To keep the ALUs busy, combine all these loads, arithmetic, and stores in superscalar fashion	The ALU is 100% utilized during large vector operations: No need for a big superscalar CPU
Without superscalar, a RAM-to-RAM vector multiply would need 4 cycles [load + load + multiply + store] to do (e.g.) 4 floating-point multiplications	Same net speed: One float multiplication per cycle For neural nets: Up to 4 SIMD byte MACs per cycle ⇒ 3.2 GOPS at Byrd's 400 MHz



Big: RISC-V "V", ARM Neon / SVE / MVE	Small: Hydra coprocessor
Vectors need a lot of RAM bandwidth	Vectors <i>still</i> need a lot of RAM bandwidth!
RAM bandwidth comes from wide cache access: Read or write a whole cache line at a time	RAM bandwidth comes from multiple DMA channels connected to multiple interleaved banks of RAM
Big chips need a data cache anyway ...	Small chips prefer to use tightly-coupled data RAM ...
... so wide cache access is a natural fit	... and several small RAMs are faster than one big RAM, so we get our multiple-banked RAM for free
Scalable to almost arbitrarily <i>huge</i> systems	Optimized specifically for <i>small "edge" systems</i>



Hydra 4

Architecture overview

- **Introduction**
 - What is it
 - Why we need it
 - **Where it came from**
- Using Hydra
- Hardware and implementation

Hydra's motivation and design

Why is a vector processor a good choice for us?

The next few slides tell the story behind Hydra's development

- They are adapted from my earlier presentation, *Hydra: A Supercomputer on Our Scale*, which described Hydra 3
- Before we get into the details of Hydra 4, let's see what motivated Hydra's general architecture and approach
- The story goes back about 10 years, with echoes going much further back ...

Scalable computation

In 2012, large touch screens on laptops looked to be the Next Big Thing for Synaptics

They inspired us back then to develop Hydra version 1

Even though LTS did not become the big market we hoped ...

... Hydra has proven effective, since then, for other compute-intensive tasks in our products

- In touch algorithms like OLED noise removal
- In fingerprint algorithms like Quantum Matcher
- And now, as smartphone screens and automotive screens get ever larger
- Recently: Hydra accelerates neural networks for state-of-the-art Accidental Contact Mitigation in our laptop touchpads and moisture immunity in our touchscreens for phones



Scalable computation

What was so special about large touch screens?

Computationally, a large touch screen is like a small touch screen ... but large!

- In every algorithm that scales with some " N ", N is larger
- N might be the number of row or column channels, or the number of touch pixels, max number of fingers, ...

Also, our products were using more image processing and other advanced algorithms that crunch data in bulk

If N is 3x larger, you could just buy a 3x-faster CPU

- They exist, but they scale quickly in size and complexity
- More than 3x as big, expensive, and power-hungry
- For example, Byrd's E76 is ~3x bigger than Chimera, but less than twice as fast (running real firmware)

⇒ Is there a smarter answer than brute force?



Scaling with N : Vector processors

A vector processor is a computer that works on numbers N -at-a-time

On a traditional general-purpose processor, each instruction does one “thing” at a time:

ADD a, b, c // Set $a \leftarrow b + c$ (using a hypothetical example processor...)

Such processors are called *scalar processors*

- (Borrowing the mathematical term for “a single number” instead of a vector or matrix)

Some processors can do several scalar operations at once, as if one larger operation:

ADD a, b, c // Set $a \leftarrow b + c$
LOAD $i, 0$ // Set $i \leftarrow 0$ } Both together in one clock cycle

This is still *scalar*, not *vector*, because it is programmed in terms of scalar operations

- Instead, it is called *superscalar*
- See our Chimera for a simple example, and SiFive’s E76 for a larger-scale example
- See big x86 and ARM Cortex-A processors for really ambitious examples

Scaling with N : Vector processors

When a scalar processor works on a vector, doing the same thing to a whole array of N numbers, it must use a loop of scalar instructions:



```

LOAD  i, 0                // Start with  $i \leftarrow 0$     (again using a hypothetical processor!)
again: LOAD  B[i]          // Load the next two input vector elements ...
      ADD   C[i]          // ... and compute their sum
      STORE A[i]          // Set  $A[i] \leftarrow B[i] + C[i]$ 
      ADD   i, 1          // Increase  $i \leftarrow i + 1$ 
      BLT   i, N, again   // Branch if  $i < N$  back to do it again
  
```

A vector processor expresses the whole thing as one instruction:

```

VADD VL(N) A, B, C    // Set  $A[i] \leftarrow B[i] + C[i]$  for all  $i = 0 \dots N-1$ 
  
```

It may be a *slow* instruction, internally taking many steps to work through a big vector ...
... but it is still much better than a loop of $5N$ instructions!

Mixing vector and scalar processing

Vector processors are efficient when your program is composed all of vector operations such as “add vector B to vector C ”

Real programs are not like that:

- Our touch and fingerprint firmware sometimes operates on N channels or N pixels in bulk
- ... but other times, it does lots of individual “this-and-that” operations on scalar data

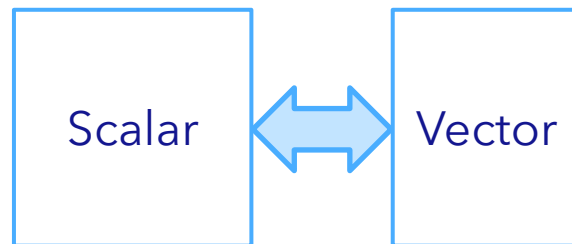
So, in reality, we can't *replace* our general-purpose scalar processor with a vector processor

Instead: Combine the two!

Mixing vector and scalar processing

Two ways to combine vector + scalar:

- Make a processor that can do both scalar instructions and vector instructions
- Or augment a general-purpose processor with a vector “co-processor” to help with heavy math



For the T1326 “Oahu” touch controller (circa 2012), we decided to add a vector coprocessor to sit alongside our traditional Chimera processor

This became Hydra 1.0

- Implemented by Kam Li; architected by Mark Pearce and Dave G.

A legendary example

What kind of vector coprocessor should we use?

- Off-the-shelf vector processors were far too “big” for touch controllers (still true today, c. 2022!)
- So, in true Synaptics style, we built our own vector coprocessor to suit our needs

What kinds of operations should it support?

Can we just recast our firmware’s algorithms in the form of specialized hardware?

- No: Our firmware evolves way too fast for that
- And the firmware varies too much, adapting to the needs of each customer and project

Instead, we wanted a reasonably general-purpose, flexible vector architecture

- What does *that* look like?

For a model, we looked to the most famous vector computer of all ...



The Cray-1 supercomputer

Seymour Cray's heritage

The Cray-1's designer, Seymour Cray, first made his name with the CDC-6600 supercomputer

When it came out in 1964, the CDC-6600 was **3x** faster than its fastest competitor

It remained the world's fastest computer until 1969

[Wikipedia]



Cray's innovations

How did Cray achieve such a huge jump in performance?

⇒ Design innovation!

He rethought everything about computer architecture and design

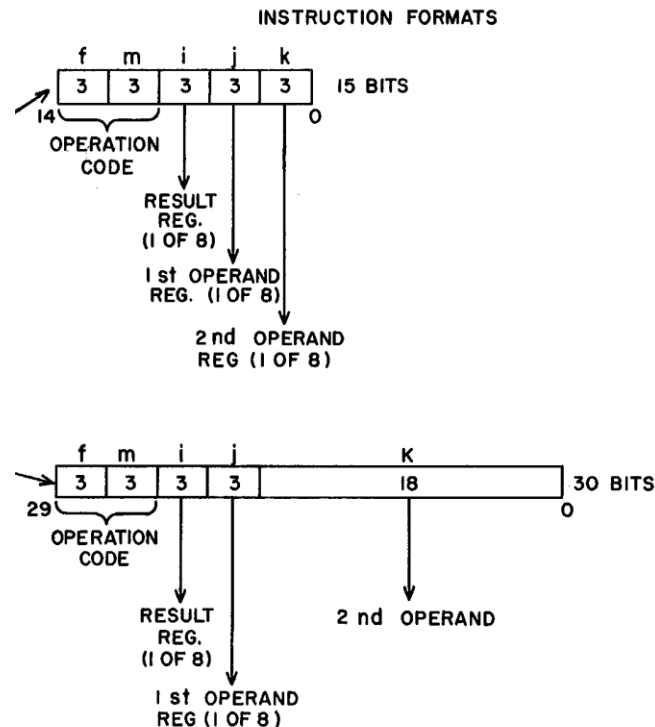
For example:

The CDC-6600 pioneered the “load/store” organization that paved the way for all modern RISC processor architectures

This idea is still widely used today

In fact, our Chimera is a rare bird for *not* using it

- Why? See my talk, “What Chimera is All About”



CDC-6600 reference manual

Cray's innovations

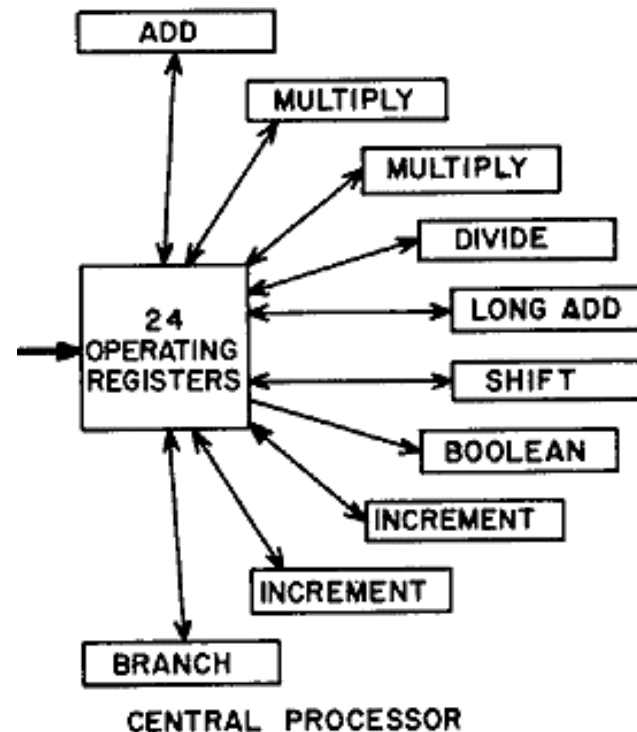
For another example:

The CDC-6600 could keep multiple instructions running simultaneously, one in each functional unit

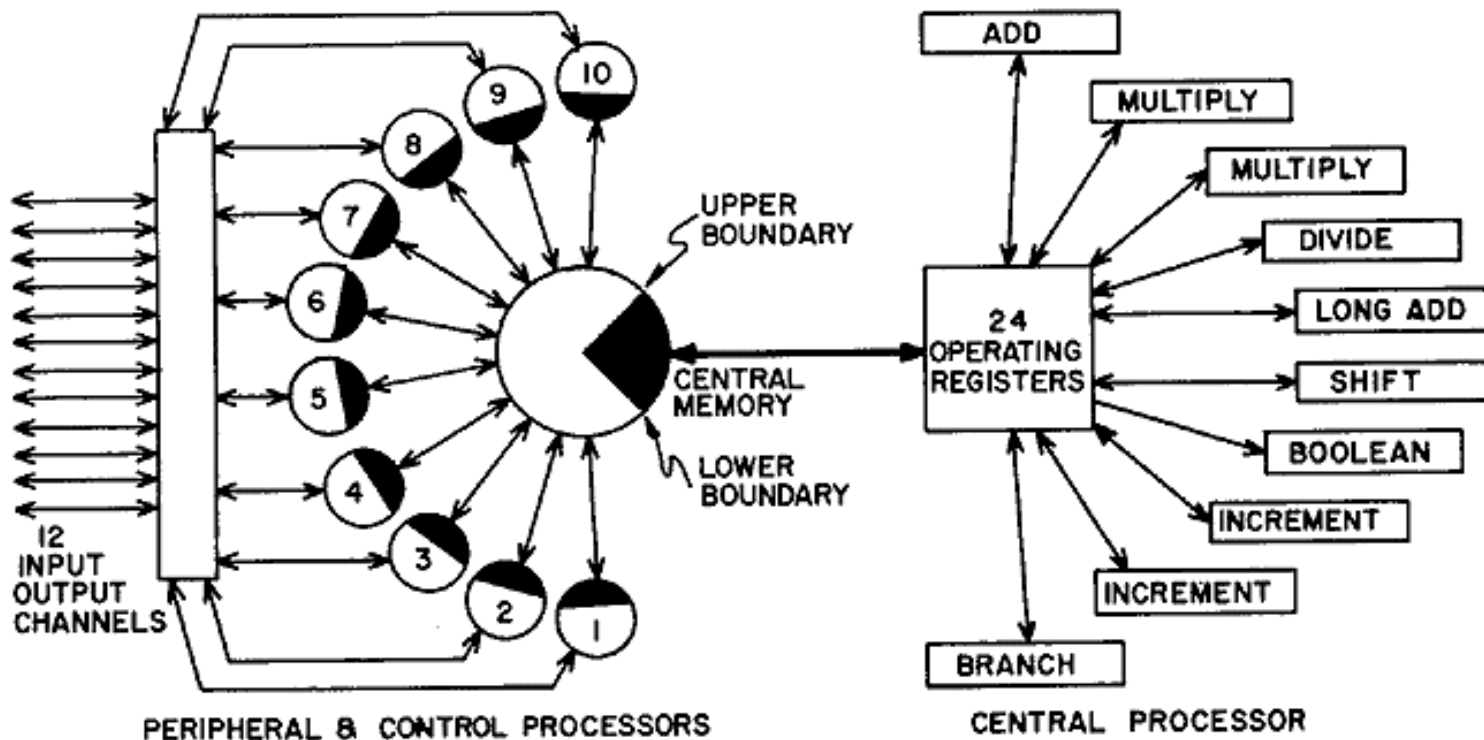
⇒ Superscalar execution

This idea, too, was pioneered by Cray and is still widely used today

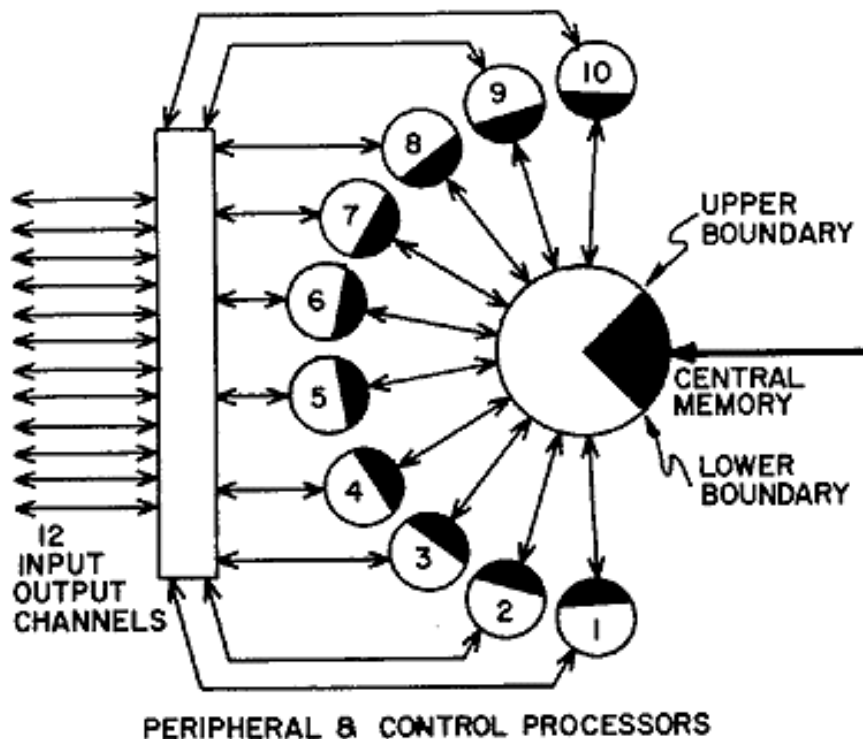
(... including, in a simple form, in our own Chimera processor!)



Cray's innovations: One more example



Cray's innovations: One more example



To let the fast main processor focus on the main task ...

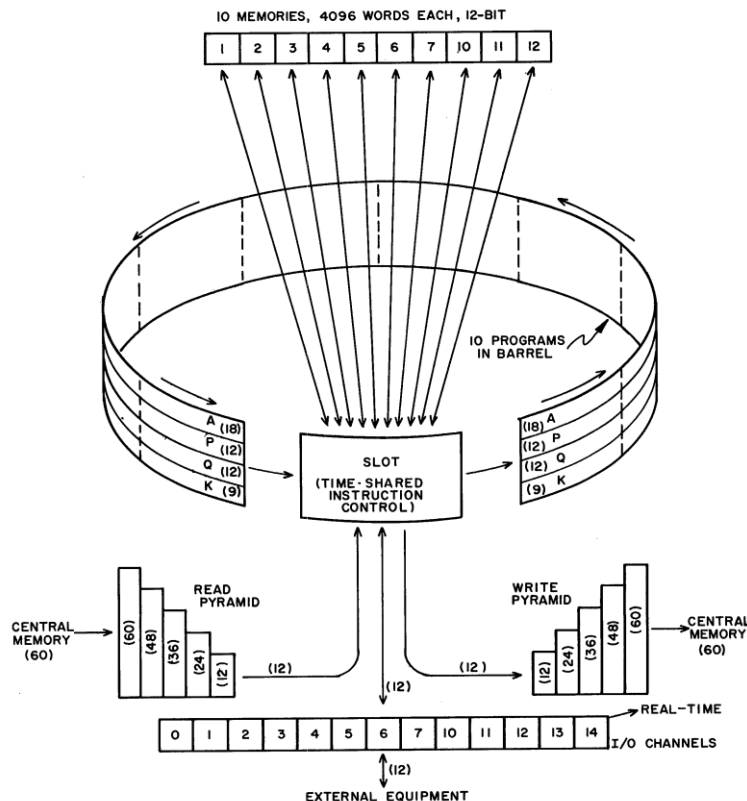
The CDC-6600 had a separate *peripheral processor* to manage real-time hardware functions

Each of 10 peripherals gets its own "virtual" control processor, dedicated to the peripheral

A single hardware processor simulates 10 independent processors by time-slicing

It performs one instruction at a time from each virtual processor in round-robin fashion

Cray's innovations: One more example



In other words: The CDC-6600 had hardware multithreading!

Interleaved hardware threads are not so common in today's computers

But if you work at Synaptics, the idea might sound familiar!

Chimera uses threads for the same reason: Efficiently dividing its time among many real-time tasks

Seymour Cray's innovations have quietly guided us all along

Another Cray hallmark

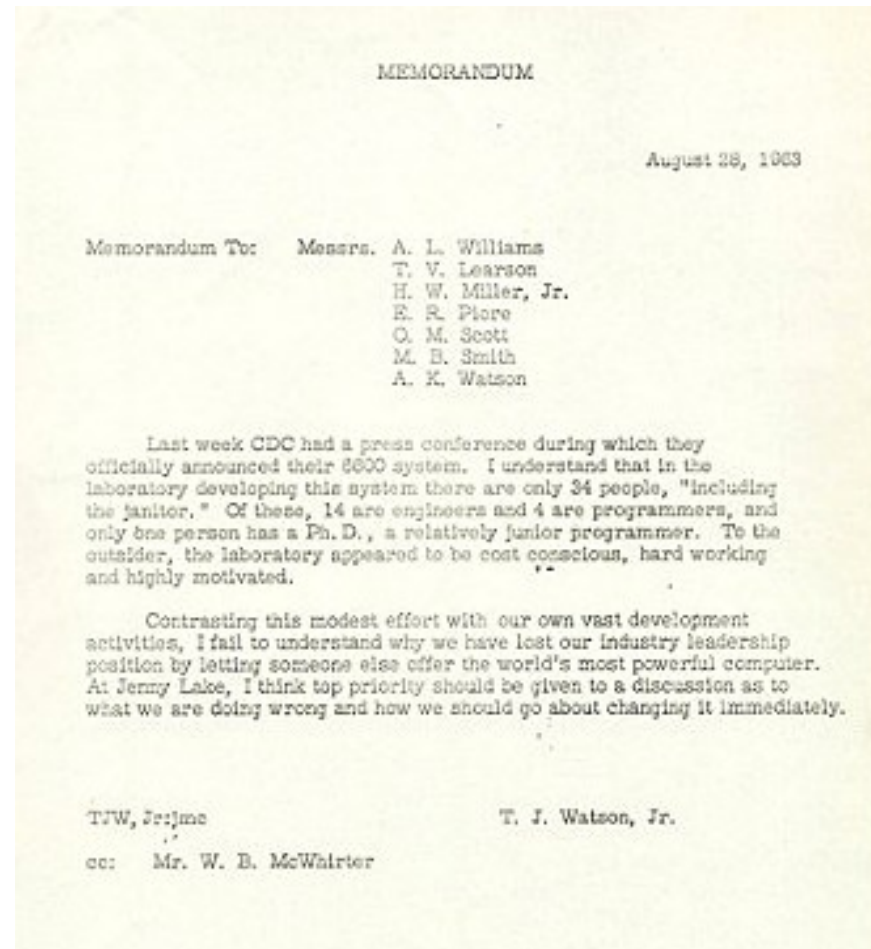
Cray designed his machines by hand,
with a small, focused team

The head of IBM asked how Cray could so
badly beat IBM's "vast development activities"
with "only 34 people, including the janitor"

Cray: "It seems like Mr. Watson has
answered his own question"

We follow Cray's lead in this, too

- Chimera's hardware team has varied in size,
sometimes as "vast" as 4 or 5 people,
sometimes just me alone
- Hydra 1.0 was architected by Mark Pierce,
Kam Li, and me; and Kam (and later I)
did the whole implementation solo
- A small team can (and must!) **focus** to succeed



Another Cray hallmark

Cray designed his machines by hand,
with a small, focused team

The head of IBM asked how Cray could so
badly beat IBM's "vast development activities"
with "only 34 people, including the janitor"

Cray: "It seems like Mr. Watson has
answered his own question"

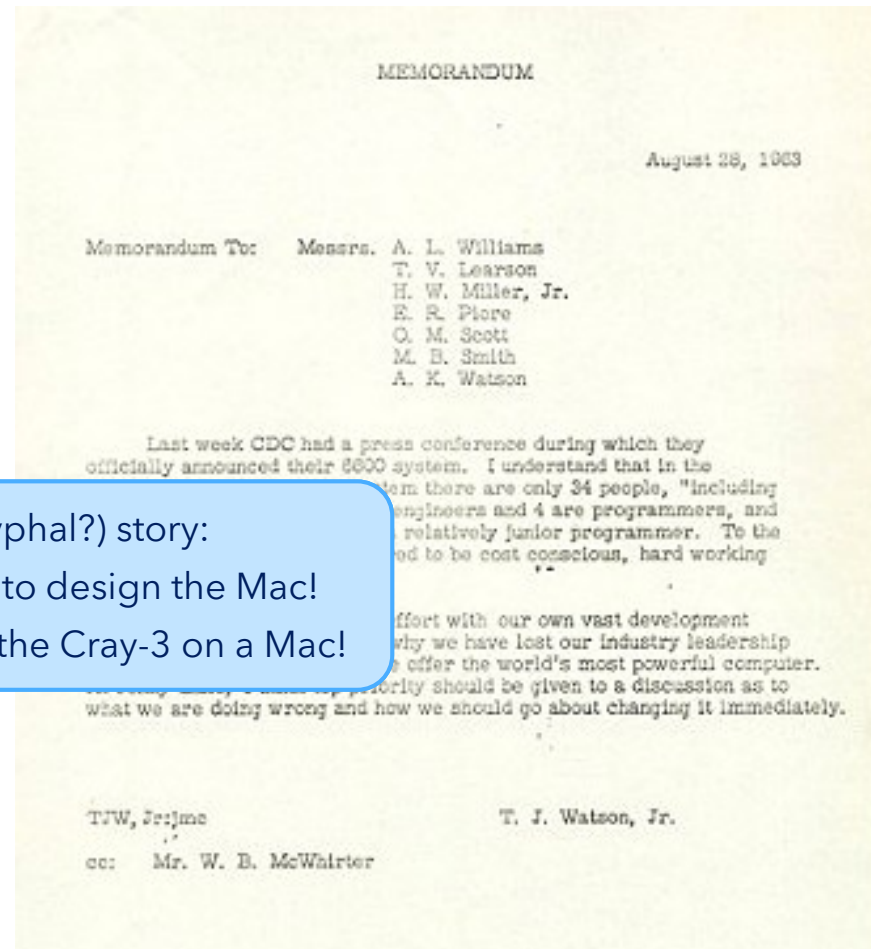
We follow Cray

- Chimera's had sometimes as
- Hydra 1.0 was architected by Mark Pierce, Kam Li, and me; and Kam (and later I) did the whole implementation solo
- A small team can (and must!) **focus** to succeed

A related, wonderful (maybe apocryphal?) story:

Jobs to Cray: We used your Cray-1 to design the Mac!

Cray's reply: Funny, I'm designing the Cray-3 on a Mac!



The Cray-1

For his next computer, Cray looked for the next huge leap in performance
He observed that scientific computations often involve arithmetic on large vectors and matrices

Why not build a computer that's actually *meant* for that kind of computation?

```
for (i = 0; i < N; i++)
```

```
    A[i] = B[i] + C[i];
```

VADD A, B, C

B[0]	B[1]	B[2]	B[3]	B[4]	B[5]	B[6]	B[7]	...	B[N-1]
+	+	+	+	+	+	+	+		+
C[0]	C[1]	C[2]	C[3]	C[4]	C[5]	C[6]	C[7]	...	C[N-1]
↓	↓	↓	↓	↓	↓	↓	↓		↓
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	...	A[N-1]

The Cray-1

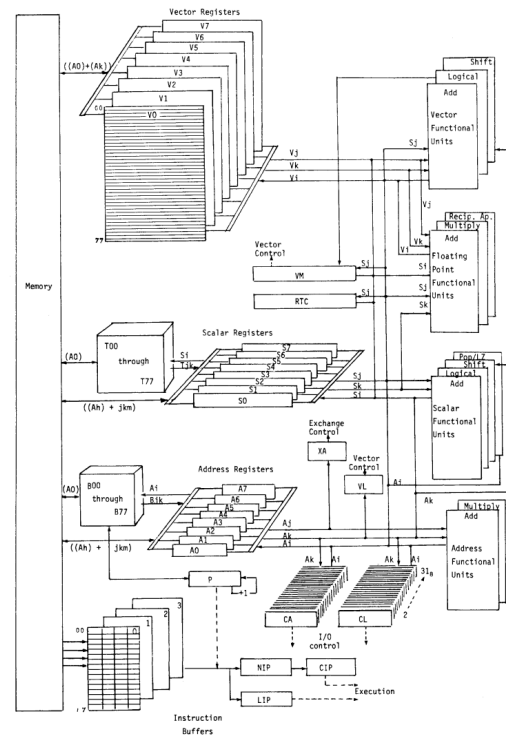
Cray-1, introduced in 1976, once again **held the speed record for years**

As usual, it was packed with innovations at every level

Not just the vector instruction set

But even the physical design

- Its distinctive round shape was not just for show: It optimized routing of wires
- Wires were all cut to the same length so that the speed-of-light delays through them were balanced!



Cray-1 reference manual

The Cray-1 is now history

Literally: You can find it and other Cray machines at the Computer History Museum in Mountain View, California

Cray-1's are still used today ...

... but only as stylish furniture

- Actually those are power / cooling units, but anyway ...

Vector machines in general fell out of fashion

- Most computer architectures have "SIMD" instructions instead, which are like vector instructions but with small fixed $N = 2$ or 4
- The open-source RISC-V processor architecture has an [excellent vector extension](#) that is gaining traction (and whose designers also cite Cray as inspiration)

Cray's vector approach lives on today in Hydra



(Google images)

Ancient history?

Here we are, a quarter or so into the 21st century

Why care about antique machines like the CDC-6600 or the Cray-1?

- Apple's M1 Ultra chip (c. 2022) has [114 billion](#) transistors – wow!
- So why are our chips stuck in the unfashionable 1970's?
- Why study such dinosaurs when they are so totally outclassed by modern technology?

The touch controllers we build today can afford a few million transistors at most

- The M1 Ultra's innovations, impressive as they are ...
- ... and fun as they are to learn about ...
- ... *are irrelevant to what we do here*
- Many orders of magnitude bigger than we could dream of in our tiny mixed-signal embedded-Flash high-voltage silicon



Ancient history

The largest discrete touch chip we have ever built, DIG2409 “Byrd”, has roughly ½ million gates of digital logic

- How many gates of logic did the Cray-1 have?
⇒ *About the same!*
- The Cray-1 may have filled a room, but in the most important way, it was “about as big” as Byrd

- 24 chassis
- 1662 modules (16 banks); 113 module types
- Each module contains up to 288 IC packages per module
- Three basic chip types
 - 5/4 NAND gates
 - Memory chips
 - Register chips

Cray-1 reference manual

This was an era when the best computer designers in the world ...

- ... with enormous resources on tap, a “blank check” from their customers ...
- ... overcoming the severe technological limitations of their day ...
- ... were building top-performing computers **on our current budget** of complexity

For us, the Cray-1, not the M1, is the *model of greatness* that makes the most sense to study and aspire to

Vector processors

A vector processor offers single instructions for various common operations on vectors

VADD	A, B, C	// Add two vectors:	$A[i] \leftarrow B[i] + C[i]$ (for all $0 \leq i < N$)
VMADD	A, B, C, D	// Multiply-add:	$A[i] \leftarrow B[i] + (C[i] * D[i])$
VMAC	sum, B, C	// Multiply-accumulate:	$sum \leftarrow (B[0] * C[0]) + (B[1] * C[1]) + \dots$
VMOVG	A, B, C	// Move-and-gather:	$A[i] \leftarrow B[C[i]]$

... and so on

If $N = 4$ (say), we could just build 4 arithmetic units that work in parallel, all at once

- This is the approach taken by, e.g., x86 SSE/AVX; ARM Cortex-M4 SIMD; Cortex-A with Neon

But in general, we can't afford to do an N -element vector operation all in one clock cycle

- Not when N gets large
- Especially not when N is large *and variable*!

Instead, Hydra works sequentially, but very efficiently, using a deep pipeline

Pipelined processing

Computers go fast by breaking a complex operation into several small steps

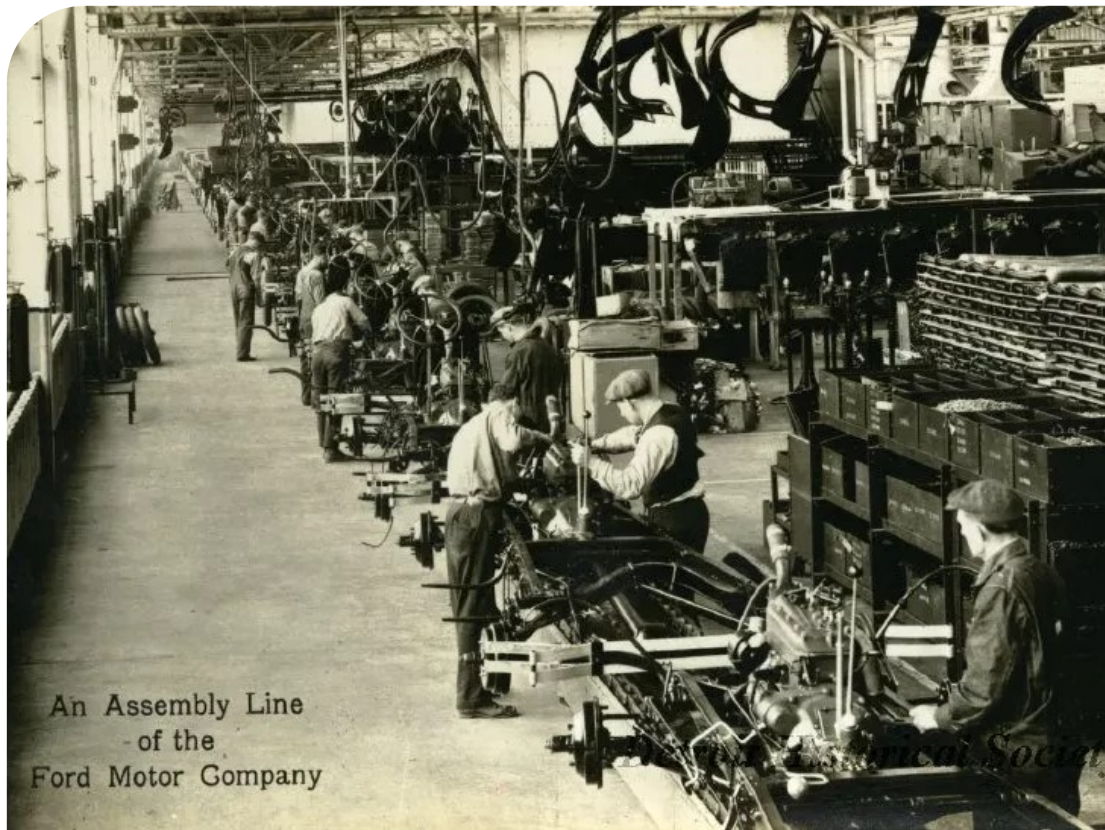
Each step does a self-contained piece of the job

These self-contained steps can be overlapped in assembly-line fashion

A “pipeline” of steps

This is much more efficient than finishing each whole operation before starting the next

Detroit Historical Society



Deep pipelines

Long computer pipelines, with many steps, get complicated very quickly!

There are countless potential interactions among all the instructions in the various stages

This makes them either ...

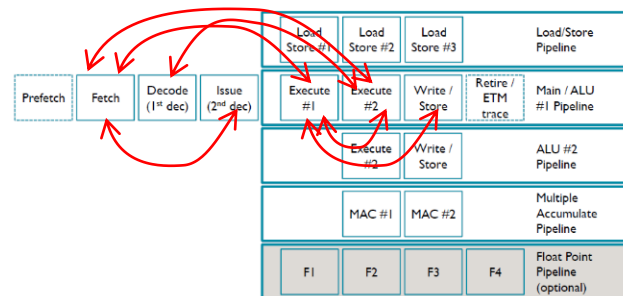
- ... inefficient, as the pipeline often stalls to resolve the interactions ...
- ... or very hard to program, due to the instruction rearrangement needed to avoid stalling ...
- ... or painfully big and complicated, due to the extensive hazard detection, data forwarding, speculative execution, dynamic instruction reordering, and branch prediction needed to eliminate stalls at the hardware level

“Billion transistor” CPUs can efficiently use deep pipelines

- The Intel chip in your laptop might have a 20-stage pipeline!

But general-purpose CPUs in our price range cannot

- This is why low-cost microcontrollers have short pipelines
- And, in turn, it's why small micros can't clock nearly as fast as big application processors



ARM Cortex-M7
8-stage pipeline

Pipelined vector processors

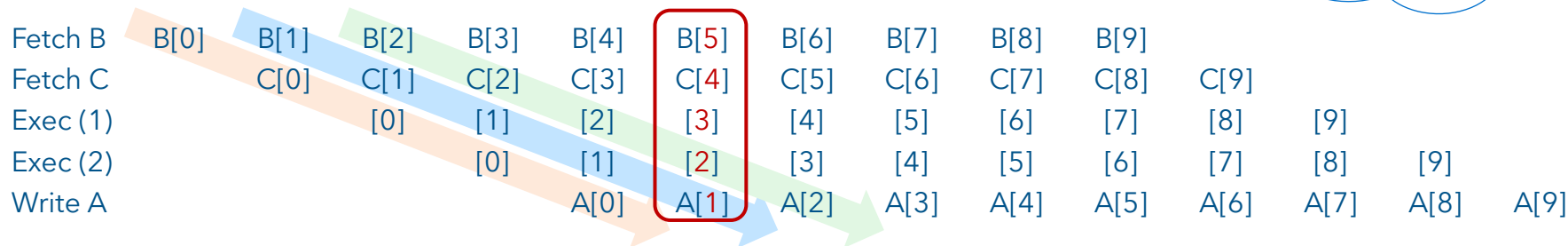
But for a vector processor, deep pipelines are [relatively] easy!

Consecutive operations are predictable

The stages can't possibly interact: Each stage is working on a different vector element

B[0]	B[1]	B[2]	B[3]	B[4]	B[5]	B[6]	B[7]	B[8]	B[9]
+	+	+	+	+	+	+	+	+	+
C[0]	C[1]	C[2]	C[3]	C[4]	C[5]	C[6]	C[7]	C[8]	C[9]
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]

(Hydra's actual pipeline is deeper than shown here)



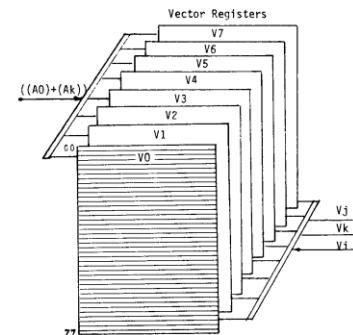
Getting the data in and out

Vector processors need lots of bandwidth to vector memory

During this vector operation, every cycle reads two numbers and writes a third number

How is this possible?

- One answer: Keep "A", "B", and "C" in separate memories
- The Cray-1 did this by holding vector data in eight "vector registers" built as 8 independent 64-entry RAMs (today, RISC-V "V" works similarly)
- Traditional DSPs do this by forcing people to manually allocate different vectors in different sections of main RAM ("X memory" vs. "Y memory"); then the DSP can access X RAM and Y RAM at once via two parallel ports



Fetch B	B[0]	B[1]	B[2]	B[3]	B[4]	B[5]	B[6]	B[7]	B[8]	B[9]				
Fetch C		C[0]	C[1]	C[2]	C[3]	C[4]	C[5]	C[6]	C[7]	C[8]	C[9]			
Exec (1)			[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]		
Exec (2)				[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	
Write A					A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]

Banked data RAM

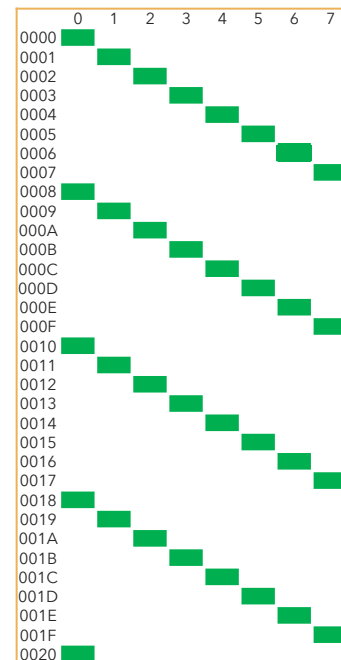
Another answer: Suppose we structure RAM as (say) 8 interleaved banks
 The data word at memory word address X is stored in bank $(X \% 8)$
 at offset $(X / 8)$ within the bank

Notice that "fetch B", "fetch C", and "write A" always use different banks!

The vector engine can operate at full speed on vectors directly in RAM

- If the vectors are aligned in memory in a favorable way
- See later for how Hydra deals well with unfavorably/randomly aligned data

Some early vector machines used interleaved RAM (CDC STAR; Cray-1),
 and we use this approach in Hydra

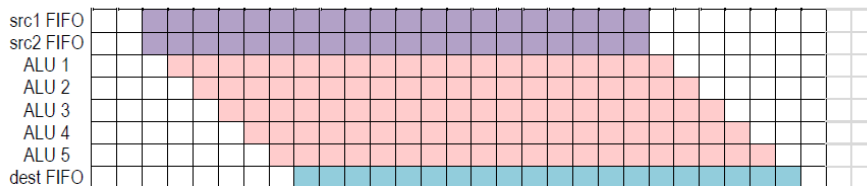
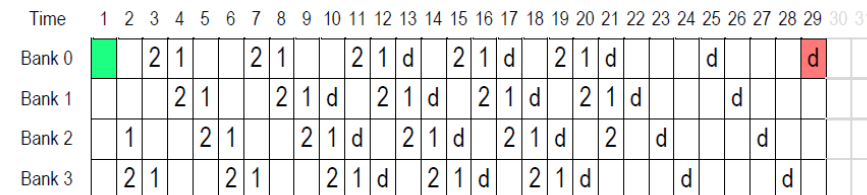


Fetch B	B[0]	B[1]	B[2]	B[3]	B[4]	B[5]	B[6]	B[7]	B[8]	B[9]					
Fetch C		C[0]	C[1]	C[2]	C[3]	C[4]	C[5]	C[6]	C[7]	C[8]	C[9]				
Exec (1)			[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]			
Exec (2)				[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]		
Write A					A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]	

Hydra's pipeline

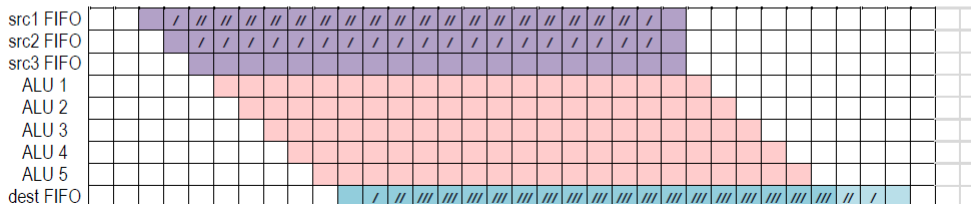
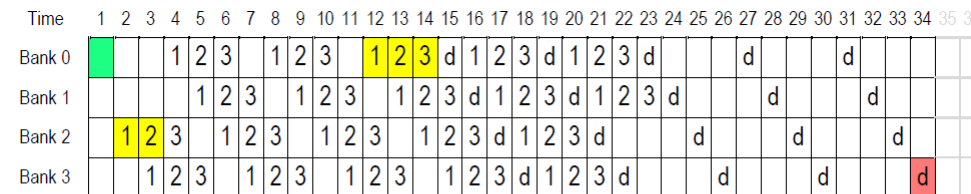
Here are some actual timing diagrams from the *Hydra 4 User's Guide* (based on chip simulation traces)

First, a VADD operation on a chip with Hydra 4.2 and four banks of RAM $\Rightarrow$



This VMADD (fused multiply-add) accesses four vectors at once in four banks $\Rightarrow$

- Here, Hydra's channel FIFO buffers absorb some stalls that occur due to the vectors' alignment in RAM (see the "Hardware" section later)



The Hydra 4 vector coprocessor

The rest of this presentation introduces Hydra 4, Synaptics' latest vector coprocessor

- First silicon in the SB7900A0 "SmartBridge" chip in early 2023
- Also appearing in DIG2409 "Byrd" in 2024 and DIG2411 "Denmarke" in 2025
- Hydra 4.2 is due to appear first in DIG2409B in 2026
- Probably next in the B1458 "Atlas" match-in-sensor fingerprint chip

Leveraging the success of Hydra 3 while improving every detail

Drawing upon the rich heritage of vector processors established by Cray



Iolaos and Heracles battling the Hydra
Etruscan vase based on a Greek painting,
circa 520 BC
(Legion of Honor Museum, San Francisco)



Heracles preparing to attack the Hydra
Greek jar made in Athens, circa 480 BC
(British Museum, London)

Hydra 4

Architecture overview

- Introduction
- **Using Hydra**
 - Examples
 - Hydra registers
 - Vector instructions
 - Data types
 - Addressing modes
 - Stored program unit
 - Documentation and tools
 - Hydra 4.2 functionality
 - Performance
- Hardware and implementation

Hydra 4

Architecture overview

- Introduction
- **Using Hydra**
 - **Examples**
 - Hydra registers
 - Vector instructions
 - Data types
 - Addressing modes
 - Stored program unit
 - Documentation and tools
 - Hydra 4.2 functionality
 - Performance
- Hardware and implementation

Using Hydra 4: A simple example

Hydra 4's header file defines functions and macros for invoking Hydra in a C program:

```
#include "hydra4_defs.h"

long results[100], inputs[500];
short data[100];

hydra_vadd(hydra_set_vl(100),
           hydra_vector(results, HYDRA_INT32),
           hydra_vector(data, HYDRA_INT16),
           hydra_strided(inputs, HYDRA_INT32, 5));
```

Equivalent to...

```
for (i = 0; i < 100; i++)
    results[i] = data[i] + inputs[5*i];
```

Here, the `hydra_vadd()` function tells Hydra to begin a VADD operation with the indicated vector operands and vector length

- Hydra reads 100 short integers from the elements of the array `data[]` ...
- ... and it reads 100 long integers from every 5th element of the array `inputs[]`; ...
- ... it adds them together to get 100 sums, ...
- ... and then it stores these sums, as long integers, in `results[]`

Sequential instructions

The result of a Hydra vector operation can feed directly into the next operation:

```
#include "hydra4_defs.h"

long results[100], inputs[500];
short data[100], scales[100];

hydra_vadd(hydra_set_vl(100),
           hydra_vector(results, HYDRA_INT32),
           hydra_vector(data, HYDRA_INT16),
           hydra_strided(inputs, HYDRA_INT32, 5));

hydra_vmul(0,      // No need to set VL again if it is unchanged.
           hydra_vector(results, HYDRA_INT32),
           _,       // Shorthand for "same vector as the result vector."
           hydra_vector(scales, HYDRA_INT16));
```

Equivalent to...

```
for (i = 0; i < 100; i++)
    results[i] = data[i] + inputs[5*i];
for (i = 0; i < 100; i++)
    results[i] *= scales[i];
```

... which computes this...

```
for (i = 0; i < 100; i++)
    results[i] = ((data[i] + inputs[5*i]) *
                  scales[i]);
```

These operations automatically overlap inside Hydra as much as possible

- The `hydra_vadd()` function starts Hydra working on the VADD and then returns immediately
- The `hydra_vmul()` then sets up the VMUL operation while the VADD crunches away inside Hydra
- The VMUL waits in Hydra's instruction buffer, then immediately runs as soon as VADD completes

Reading Hydra's results in firmware

When firmware wishes to look *directly* at Hydra's results, it must wait for Hydra to finish:

```
#include "hydra4_defs.h"

long results[100], inputs[500];
short data[100], scales[100];

hydra_vadd(hydra_set_vl(100),
           hydra_vector(results, HYDRA_INT32),
           hydra_vector(data, HYDRA_INT16),
           hydra_strided(inputs, HYDRA_INT32, 5));

hydra_vmul(0,      // No need to set VL again if it is unchanged.
           hydra_vector(results, HYDRA_INT32),
           _,       // Shorthand for "same vector as the result vector."
           hydra_vector(scales, HYDRA_INT16));

hydra_wait();
some_stuff(results[0] + results[99]);
```

Equivalent to...

```
for (i = 0; i < 100; i++)
    results[i] = data[i] + inputs[5*i];
for (i = 0; i < 100; i++)
    results[i] *= scales[i];
some_stuff(results[0] + results[99]);
```

Simulating with HydraSim will catch the error if you forget to call `hydra_wait()`

There is a convenient notation to run a Hydra operation and wait for it to complete:

```
hydra_vmul_w(...);    ... is the same as ...    hydra_vmul(...); hydra_wait();
```

Combining data types

Operations like `hydra_vadd()` can work with any combination of data types:

```
#include "hydra4_defs.h"

float results[100];           // Single-precision floating-point numbers.
long data[100];               // Signed 32-bit fixed-point numbers with 22 integer bits and 10 fraction bits.
unsigned short inputs[500];   // Unsigned 16-bit integers.

hydra_vadd(hydra_set_vl(100),
           hydra_vector(results, HYDRA_FLOAT),
           hydra_vector(data, HYDRA_FIX32(10)),
           hydra_strided(inputs, HYDRA_UINT16, 5));
```

Hydra performs the necessary type conversions on its inputs and outputs as it goes:

- For each index i from 0 up to the vector length ...
- Hydra reads the i 'th set of input values from RAM (here, they are `data[i]` and `inputs[i]`);
- It interprets each input value as an exact mathematical number, according to its data type;
- It internally calculates the exact mathematical result;
- Then it rounds, saturates, and converts the exact result to match the output data type;
- And finally, it stores the i 'th result back to RAM (here, the result goes in `results[i]`)

Combining addressing modes

Also, operations like `hydra_vadd()` work with any combination of addressing modes:

```
#include "hydra4_defs.h"

float results[5][20];           // Single-precision floating-point numbers.
long data[20][5];              // Signed 32-bit fixed-point numbers with 22 integer bits and 10 fraction bits.
unsigned short inputs[20];     // Unsigned 16-bit integers.

hydra_vadd(hydra_set_vl(5*20),
           hydra_matrix(results, HYDRA_FLOAT, 20),           // A 5x20 matrix.
           hydra_tmatrix(data, HYDRA_FIX32(10), 5),         // A 5x20 transposed view of a 20x5 matrix in RAM.
           hydra_rrmatrix(inputs, HYDRA_UINT16, 20));       // A 20-element row repeated to form a 5x20 matrix.
```

Each mode defines how each index $i = 0, 1, \dots, VL-1$ maps to a memory address:

- For simple linear vectors: $base_address + (i * element_size)$
- For strided vectors: $base_address + (i * stride * element_size)$
- For a repeated-row matrix: $base_address + ((i \% width) * element_size)$
- For a general matrix: $base_address + (((i / width) * row_stride) + ((i \% width) * column_stride)) * element_size)$
- ... and so on, with many supported addressing modes

Scalar values

Sometimes, an input to a vector operation will be the same for every vector element

- These are called “scalar values”
- Hydra 4 can take a scalar value directly, or it can draw a scalar value from a RAM location:

```
float my_data[100];
float my_factor;

hydra_vmul(hydra_set_vl(100),                                // my_data[i] *= 3.1415 for all i.
           hydra_vector(my_data, HYDRA_FLOAT), _,
           hydra_scalar_float(3.1415));

hydra_vmul(hydra_set_vl(100),                                // my_data[i] *= my_factor for all i.
           hydra_vector(my_data, HYDRA_FLOAT), _,
           hydra_mscalar(&my_factor, HYDRA_FLOAT));           // "mscalar" = "memory scalar".
```

Hydra can also generate various numeric patterns as inputs:

```
hydra_vmul(hydra_set_vl(100),                                // my_data[i] *= (10+2*i) for all i.
           hydra_vector(my_data, HYDRA_FLOAT), _,
           hydra_pattern(10, 2));
```

Vector masks

Operations can apply conditionally to selected vector elements

- First, an operation such as vector comparison creates a *vector mask* with one bit per element
- Subsequent operations can be made to apply only if the mask bit is 1, or only if the mask bit is 0

```
float my_data[100];
uint32_t my_mask[HYDRA_VM_SIZE(100)]; // == [4], since 4*32 ≥ 100 bits.

hydra_vcmp_gt(hydra_set_vl(100), // my_mask[bit i] = (my_data[i] > 3.1415) ? 1 : 0.
              hydra_mask_gen(my_mask),
              hydra_vector(my_data, HYDRA_FLOAT),
              hydra_scalar_float(3.1415));

hydra_vmul(hydra_mask_if(my_mask), // if (my_mask[bit i] == 1) my_data[i] *= 0.5.
           hydra_vector(my_data, HYDRA_FLOAT), _,
           hydra_scalar_float(0.5));

hydra_vmov(hydra_mask_else(my_mask), // if (my_mask[bit i] == 0) my_data[i] = -1.
           hydra_vector(my_data, HYDRA_FLOAT),
           hydra_scalar_float(-1.0));
```

How Hydra functions interface to the hardware

The functions `hydra_vadd()`, `hydra_vector()`, `hydra_strided()`, etc., ...

```
#include "hydra4_defs.h"

long results[100], inputs[500];
short data[100];

hydra_vadd(hydra_set_vl(100),
           hydra_vector(results, HYDRA_INT32),
           hydra_vector(data, HYDRA_INT16),
           hydra_strided(inputs, HYDRA_INT32, 5));
```

Equivalent to...

```
for (i = 0; i < 100; i++)
    results[i] = data[i] + inputs[5*i];
```

... expand to a sequence of lower-level operations:

```
hydra_set_vl(100);

hydra_set_vector(HYDRA_V0, results, HYDRA_INT32);
hydra_set_vector(HYDRA_V1, data, HYDRA_INT16);
hydra_set_strided(HYDRA_V2, inputs, HYDRA_INT32, 5);

hydra_vadd(0, HYDRA_V0, HYDRA_V1, HYDRA_V2);
```

... which, in turn, translate to accessing various Hydra hardware registers:

```
HYDRA_P->VL = 100;
HYDRA_P->V0CFG = 0x00000020;
HYDRA_P->V0ADDR = (int) results;
HYDRA_P->V1CFG = 0x00000010;
HYDRA_P->V1ADDR = (int) data;
HYDRA_P->V2CFG = 0x00005220;
HYDRA_P->V2ADDR = (int) inputs;
while (!(HYDRA_P->IRQ & 2)) { }
HYDRA_P->OPCODE = 0x00006210;
```

Vector registers

Like many vector machines, Hydra 4 organizes vector data as a set of *vector registers*

- Large vector processors such as Cray, ARM Neon, and RISC-V “V” have a large set of vector registers that directly store the vector data inside the CPU
- To save cost, Hydra instead stores the *contents* of each vector in the chip’s main data RAM
- Hydra’s vector registers V0-V7 are just *descriptors* that indicate the data type, addressing mode (the RAM access pattern), and base address of the data in RAM
- Functions such as `hydra_set_strided()` simply initialize a V_n register with that information:

```
hydra_set_vl(100);
hydra_set_vector(HYDRA_V0, results, HYDRA_INT32);
hydra_set_vector(HYDRA_V1, data, HYDRA_INT16);
hydra_set_strided(HYDRA_V2, inputs, HYDRA_INT32, 5);
hydra_vadd(0, HYDRA_V0, HYDRA_V1, HYDRA_V2);
```

- Internally, this corresponds to writing to a pair of Hydra registers called $V_n\text{CFG}$ and $V_n\text{ADDR}$ (for some $n = 0 \dots 7$)

```
HYDRA_P->VL = 100;
HYDRA_P->V0CFG = 0x00000020;
HYDRA_P->V0ADDR = (int) results;
HYDRA_P->V1CFG = 0x00000010;
HYDRA_P->V1ADDR = (int) data;
HYDRA_P->V2CFG = 0x00005220;
HYDRA_P->V2ADDR = (int) inputs;
while (!(HYDRA_P->IRQ & 2)) { }
HYDRA_P->OPCODE = 0x00006210;
```

Choosing the vector registers

Handy functions such as `hydra_strided()` automatically select the next free `Vn`:

```
#include "hydra4_defs.h"

long results[100], inputs[500];
short data[100];

hydra_vadd(hydra_set_vl(100),
           hydra_vector(results, HYDRA_INT32),
           hydra_vector(data, HYDRA_INT16),
           hydra_strided(inputs, HYDRA_INT32, 5));
```

```
...
hydra_set_strided(HYDRA_V2, inputs, HYDRA_INT32, 5);
hydra_vadd(0, HYDRA_V0, HYDRA_V1, HYDRA_V2);
```

- The function then invokes `hydra_set_strided()` to initialize the selected vector `Vn`
- It returns a constant such as `HYDRA_V2`, suitable for passing to the function `hydra_vadd()`
- `hydra_vadd()` formats the vector numbers into the appropriate bits of the Hydra instruction opcode, combined with bits that select the instruction itself and some optional “modifier” bits
- It posts this opcode to the `OPCODE` register to start Hydra

```
HYDRA_P->VL = 100;
HYDRA_P->V0CFG = 0x00000020;
HYDRA_P->V0ADDR = (int) results;
HYDRA_P->V1CFG = 0x00000010;
HYDRA_P->V1ADDR = (int) data;
HYDRA_P->V2CFG = 0x00005220;
HYDRA_P->V2ADDR = (int) inputs;
while (!(HYDRA_P->IRQ & 2)) { }
HYDRA_P->OPCODE = 0x00006210;
```

Using interrupts

Hydra can be set to send an interrupt signal to the host CPU on various conditions:

- Interrupt when Hydra is READY to receive another instruction
- Interrupt when Hydra is IDLE: All posted instructions are done
 - Waiting for an IDLE interrupt is like calling `hydra_wait()`
- Interrupt when an ERROR occurs, etc....
- Hydra's IEN register controls which events, if any, will signal an interrupt

On traditional RISC CPUs, interrupts have tens or hundreds of cycles of overhead

- It is not practical to use interrupts to wait for Hydra in normal circumstances
- Therefore, the standard Hydra functions do not use interrupts by default when building for RISC-V or ARM
- Instead, they use spin loops to wait for READY or IDLE ⇒
- But interrupts are available for very long waits, e.g., waiting for a big matrix-multiply or SPU program to finish

`hydra_vadd_w(...)`

```
HYDRA_P->VL = 100;
HYDRA_P->V0CFG = 0x00000020;
HYDRA_P->V0ADDR = (int) results;
HYDRA_P->V1CFG = 0x00000010;
HYDRA_P->V1ADDR = (int) data;
HYDRA_P->V2CFG = 0x00005220;
HYDRA_P->V2ADDR = (int) inputs;
while (!(HYDRA_P->IRQ & 2)) { }
HYDRA_P->OPCODE = 0x00006210;
while (!(HYDRA_P->IRQ & 1)) { }
```

Using interrupts on Chimera

Chimera's interrupts are very fast and lightweight

- When building for Chimera, the same Hydra functions generate code that puts the thread to sleep, waiting for an interrupt
- These functions wait for the READY interrupt before posting an instruction to Hydra
 - In the common case that Hydra was READY already, the interrupt is already asserted
 - Chimera does not sleep, and no time is wasted
- ... and they wait for the IDLE interrupt to implement `hydra_wait()`, `hydra_vadd_w()`, etc.
- Other firmware threads can run during that time
 - Or the CPU can sleep, conserving power
- Here, the BSET- or BCLR-to-IEN instruction takes just one Chimera clock cycle $\Rightarrow$
- ... and the SLEEP instruction also takes one cycle

`hydra_vadd_w(...)`

```
HYDRA_P->VL = 100;
HYDRA_P->V0CFG = 0x00000020;
HYDRA_P->V0ADDR = (int) results;
HYDRA_P->V1CFG = 0x00000010;
HYDRA_P->V1ADDR = (int) data;
HYDRA_P->V2CFG = 0x00005220;
HYDRA_P->V2ADDR = (int) inputs;
→ HYDRA_P->IEN |= 2; SLEEP();
HYDRA_P->OPCODE = 0x00006210;
→ HYDRA_P->IEN &= ~2; SLEEP();
```

Eliminating setup overhead

Doesn't it take time to write to all those Hydra hardware registers for every instruction?

- Actually, no!
- During a sequence of Hydra instructions, the next instruction is always pending in the instruction buffer
- The time spent writing to registers, etc., completely overlaps the vector operations running inside Hydra
- Hydra stays busy 100% of the time (if N is large)

```

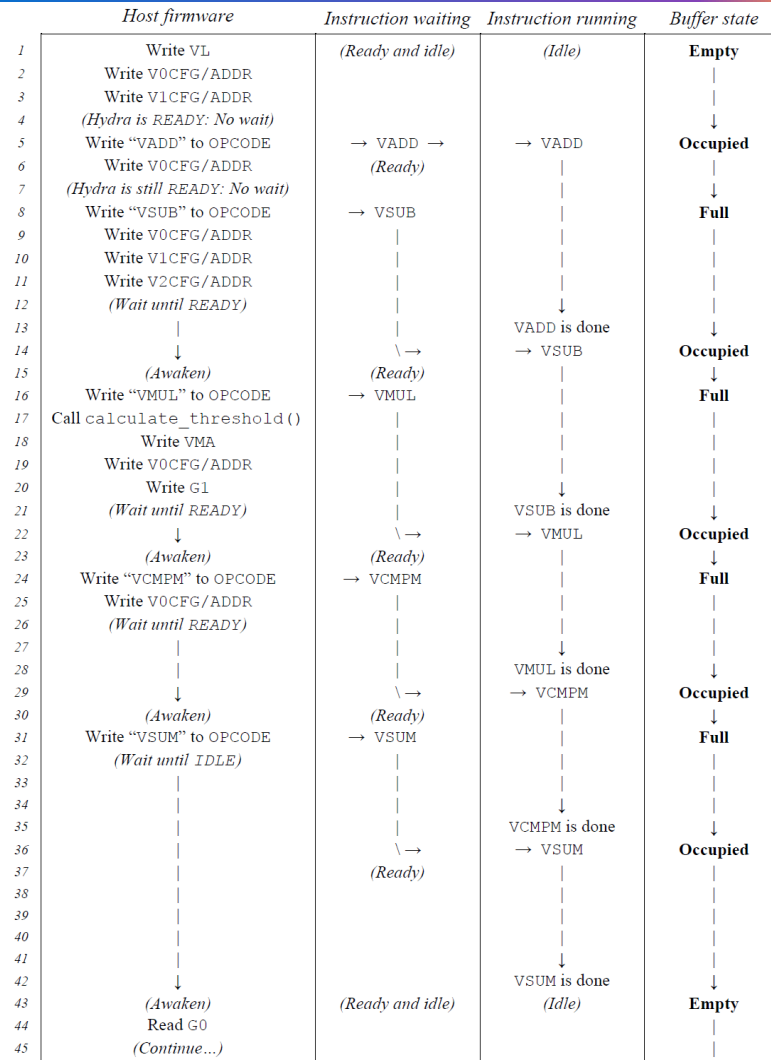
hydra_vadd(hydra_set_vl(N),
           hydra_vector(my_vec, HYDRA_FLOAT), __,
           hydra_vector(other_vec, HYDRA_FLOAT));

hydra_vsub(0, hydra_vector(my_vec, HYDRA_FLOAT), __,
           HYDRA_ONE);

hydra_vmul(0, hydra_vector(other_vec, HYDRA_FLOAT),
           hydra_vector(my_vec, HYDRA_FLOAT),
           hydra_vector(my_vec, HYDRA_FLOAT));

extern float calculate_threshold(); // Some calculation; details are not relevant here.
float thresh = calculate_threshold();
hydra_vcmpm_lt(0, mask, hydra_vector(other_vec, HYDRA_FLOAT),
              hydra_scalar_float(thresh));

hydra_vsum_w(HYDRA_IF, HYDRA_FG0, hydra_vector(other_vec, HYDRA_FLOAT));
float my_sum = hydra_get_g0_float();
    
```



Instruction buffer

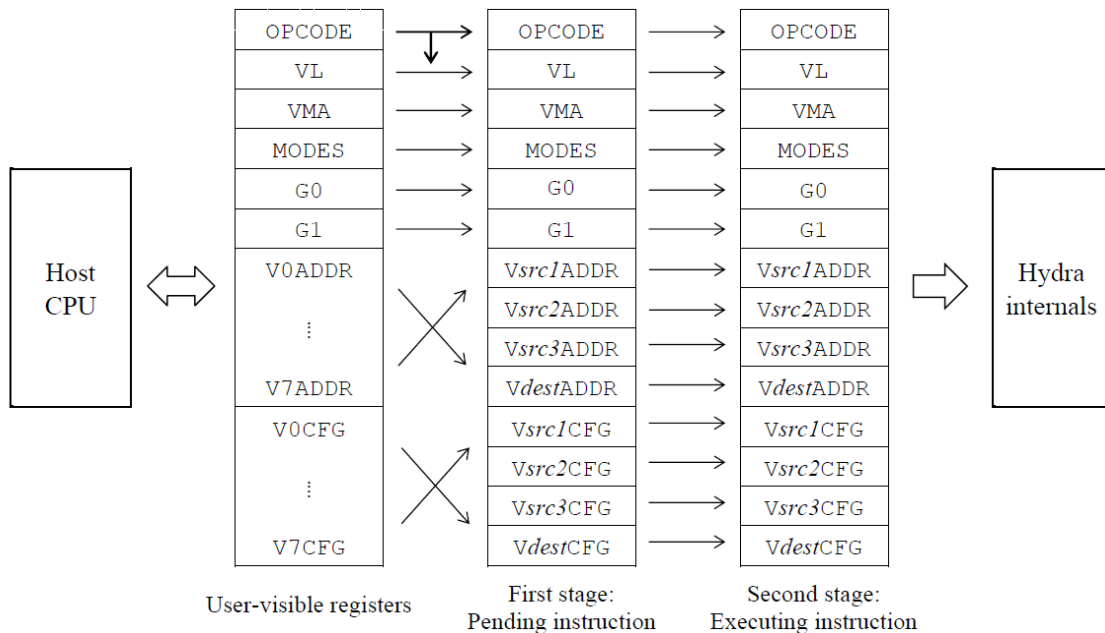
Hydra's two-stage instruction buffer can hold one executing vector instruction ...

... and one pending instruction ...

... while the host sets up the next instruction by accessing Hydra's registers

Instructions advance automatically through the instruction buffer's stages

- Entering as they are posted to the OPCODE register
- Leaving as they complete their execution in Hydra's vector pipeline



Example: Square root using Newton's method

Let's look at a real-world example: Square roots

- Hydra does not have an all-in-one “square root” operation
- Instead, it provides an instruction that estimates $1/\text{sqrt}(x)$ to just a few bits of precision

```
hydra_vrsqrte(hydra_set_vl(N),
              hydra_vector(results, HYDRA_FLOAT), // Output vector.
              hydra_vector(data, HYDRA_FLOAT));    // Input vector.
```

Three iterations of Newton's method then refine this estimate to full single precision

- If x_k is an estimate of $1/\text{sqrt}(y)$ with relative error e_k ...
- Then a better estimate is $x_{k+1} = (1/2) * (3 * x_k - y * x_k^3)$
- It converges quadratically: $e_{k+1} \approx e_k^2$ (see *Hydra User's Guide* section 4.3.7 for details)
- `hydra_vrsqrte()` estimates x_0 with $|e_0| < 2^{-3} \Rightarrow \Rightarrow \Rightarrow x_3$ is accurate with $|e_3| < (((2^{-3})^2)^2)^2 = 2^{-24}$
- Each Newton iteration becomes a `hydra_vmulh()` “multiply and halve”, a `hydra_vmsub()` “multiply and subtract”, and a `hydra_vmul()` “multiply”
- One final `hydra_vmul()` gives $\text{sqrt}(y) = y * (1/\text{sqrt}(y))$

Example: Square root using Newton's method

```
float inputs[N], results[N], temp[N];
hydra_vrsqrte(hydra_set_vl(N),
              hydra_vector(results, HYDRA_FLOAT),
              hydra_vector(inputs, HYDRA_FLOAT));
for (int iter = 0; iter < 3; iter++) {
    // Iterate: results = results * (1.5 - (inputs * (results^2) * 0.5)).
    hydra_vmulh(0,    // temp = inputs * results * 0.5.
               hydra_vector(temp, HYDRA_FLOAT),
               hydra_vector(inputs, HYDRA_FLOAT),
               hydra_vector(results, HYDRA_FLOAT));
    hydra_vmsub(0,    // temp = 1.5 - (temp * results).
               hydra_vector(temp, HYDRA_FLOAT),
               hydra_scalar_float(1.5),
               _',
               hydra_vector(results, HYDRA_FLOAT));
    hydra_vmul(0,    // results *= temp.
               hydra_vector(results, HYDRA_FLOAT),
               _',
               hydra_vector(temp, HYDRA_FLOAT));
}
hydra_vmul_w(0,    // Now sqrt(inputs) = inputs * (1/sqrt(inputs)).
             hydra_vector(results, HYDRA_FLOAT),
             _',
             hydra_vector(inputs, HYDRA_FLOAT));
```

Each N -element vector operation takes N cycles

- At 400 MHz (in Byrd)
- Plus a few cycles of instruction startup overhead to fill/empty Hydra's data pipeline
- This startup overhead becomes negligible as N grows large

Total time for this algorithm is $11*N$ cycles (plus startup overhead)

- Division using Newton's method takes $6*N$ cycles

Example: Square root with vector registers

Recall that macros like `hydra_vector()` set up a “vector register” inside Hydra

- These macros normally choose the next available vector register (among V0-V7) automatically
- An alternative form, `hydra_set_vector()`, lets you choose the vector register yourself

We can reduce overhead by putting our working vectors directly in chosen V_n registers:

```
float inputs[N], results[N], temp[N];

hydra_set_vl(N);
hydra_set_vector(HYDRA_V0, inputs, HYDRA_FLOAT);
hydra_set_vector(HYDRA_V1, results, HYDRA_FLOAT);
hydra_set_vector(HYDRA_V2, temp, HYDRA_FLOAT);

hydra_vrsqrte(0, HYDRA_V1, HYDRA_V0);

for (int iter = 0; iter < 3; iter++) {
    hydra_vmulh(0, HYDRA_V2, HYDRA_V0, HYDRA_V1);
    hydra_vmsub(0, HYDRA_V2, hydra_scalar_float(1.5), __, HYDRA_V1);
    hydra_vmul(0, HYDRA_V1, __, HYDRA_V2);
}

hydra_vmul_w(0, HYDRA_V1, __, HYDRA_V0);
```

Example: Square root with the SPU

Hydra's Stored Program Unit can run a complete algorithm on its own inside Hydra

- Guided by a stored program made of vector operations and other MCU-like instructions:

```
uint32_t sqrt_prog[100];                                // An SPU program is just an array in RAM.
hydra_spu_begin(sqrt_prog, 100) {                        // Record the following sequence of Hydra operations in sqrt_prog[:
    hydra_vrsqrte(0, HYDRA_V1, HYDRA_V0);               // In this context, each Hydra instr is recorded instead of running immediately.
    for (int iter = 0; iter < 3; iter++) {
        hydra_vmulh(0, HYDRA_V2, HYDRA_V0, HYDRA_V1);
        hydra_vmsub(0, HYDRA_V2, hydra_scalar_float(1.5), _, HYDRA_V1);
        hydra_vmul(0, HYDRA_V1, _, HYDRA_V2);
    }
    hydra_vmul(0, HYDRA_V1, _, HYDRA_V0);
    hydra_spu_end();
}
```

Notice that the source code between the `hydra_spu_begin()` and the `hydra_spu_end()` is exactly the same as the non-SPU version of the code!

- To run the program:

```
float inputs[N], results[N], temp[N];

hydra_set_vl(N);
hydra_set_vector(HYDRA_V0, inputs, HYDRA_FLOAT);
hydra_set_vector(HYDRA_V1, results, HYDRA_FLOAT);
hydra_set_vector(HYDRA_V2, temp, HYDRA_FLOAT);

hydra_vstart_w(hydra_set_vpc(sqrt_prog));
```

Example: Floating-point FFT

Hydra 3 performed Fast Fourier Transforms as a single, all-in-one operation

- Very convenient, but the hardware was so expensive that we had to omit it in many chips!
- Also, inefficient because the full hardware FFT didn't fit well with Hydra's pipeline

Hydra 4 instead provides "building blocks" that allow an efficient pipelined FFT to be built as a firmware routine or SPU program that operates Hydra

- The algorithm, as expressed in complex numbers in the *Hydra 4 User's Guide*:

```

for all  $0 \leq j < N$ :
     $X[j] \leftarrow x[\text{bitrev}_{LN}(j)]$            // bitrev() is bit reversal (see section 3.3.14);  $LN = \log_2(N)$ .
for each  $M = 2, 4, 8, \dots, N$            // Successive powers of two.
    for all  $0 \leq k < N/M$ :
        for all  $0 \leq j < M/2$ :
             $w \leftarrow \exp(2 \pi i j / M)$     // Complex exponential  $\exp(i z) = \cos(z) + i \sin(z)$ .
             $t \leftarrow w * X[k*M + j + M/2]$     // Complex multiplication.
             $X[k*M + j + M/2] \leftarrow X[k*M + j] - t$     // Complex subtraction.
             $X[k*M + j] \leftarrow X[k*M + j] + t$     // Complex addition.
    
```

Example: Floating-point FFT (2)

The first step uses bit-reversed addressing to rearrange the inputs

- It also applies the arithmetic for $M = 2$ as a special case

```
float real[N], imag[N], temp[N], cosines[N/4];
int t = HYDRA_FLOAT;

#define REALE hydra_strided(real, t, 2)
#define REALO hydra_strided(real+1, t, 2)
#define IMAGE hydra_strided(imag, t, 2)
#define IMAGO hydra_strided(imag+1, t, 2)
#define TEMPE hydra_strided(temp, t, 2)
#define TEMPO hydra_strided(temp+1, t, 2)

hydra_set_vl(N);
hydra_vmov(0, hydra_vector(temp, t), hydra_bitrev(real, t, HYDRA_BITREV_STRIDE(N/2)));
hydra_set_vl(N/2);
hydra_vadd(0, REALE, TEMPE, TEMPO);
hydra_vsub(0, REALO, TEMPE, TEMPO);
hydra_set_vl(N);
hydra_vmov(0, hydra_vector(temp, t), hydra_bitrev(imag, t, HYDRA_BITREV_STRIDE(N/2)));
hydra_set_vl(N/2);
hydra_vadd(0, IMAGE, TEMPE, TEMPO);
hydra_vsub(0, IMAGO, TEMPE, TEMPO);
```

```
for all  $0 \leq j < N$ :
     $X[j] \leftarrow x[\text{bitrev}_{LN}(j)]$ 

for each  $M = 2, 4, 8, \dots, N$ 
    for all  $0 \leq k < N/M$ :
        for all  $0 \leq j < M/2$ :
             $w \leftarrow \exp(2 \pi i j / M)$ 
             $t \leftarrow w * X[k*M + j + M/2]$ 
             $X[k*M + j + M/2] \leftarrow X[k*M + j] - t$ 
             $X[k*M + j] \leftarrow X[k*M + j] + t$ 
```

Example: Floating-point FFT (3)

Then several steps use a “butterfly” mode to combine both inner loops of the FFT into one loop across $N/2$ elements

- And “cosine table” and “sine table” modes to expand a one-quadrant RAM table of cosines into full $\sin(x)$ and $\cos(x)$ functions
- Total execution time is about $4 \cdot N \cdot \log_2(N)$ cycles (not counting instr startup overhead)

```

for all  $0 \leq j < N$ :
     $X[j] \leftarrow x[\text{bitrev}_{LN}(j)]$ 

for each  $M = 2, 4, 8, \dots, N$ 
    for all  $0 \leq k < N/M$ :
        for all  $0 \leq j < M/2$ :
             $w \leftarrow \exp(2 \pi i j / M)$ 
             $t \leftarrow w * X[k * M + j + M/2]$ 
             $X[k * M + j + M/2] \leftarrow X[k * M + j] - t$ 
             $X[k * M + j] \leftarrow X[k * M + j] + t$ 

```

```

for (int M = 4; M <= N; M *= 2) {
    uint32_t bfly_args = HYDRA_BFLY_M(M);
    uint32_t sincos_args = HYDRA_SINCOS_NM(MAX_N, M) + HYDRA_SINCOS_HALF;
    hydra_set_vl(N/2);
    hydra_vmul(0, TEMPE, hydra_bfly(real+(M/2), t, bfly_args), hydra_cosine(cosines, t, sincos_args));
    hydra_vmul(0, TEMPO, hydra_bfly(imag+(M/2), t, bfly_args), hydra_cosine(cosines, t, sincos_args));
    hydra_vmadd(0, TEMPE, _, hydra_bfly(imag+(M/2), t, bfly_args), hydra_sine(cosines, t, sincos_args));
    hydra_vmsub(0, TEMPO, _, hydra_bfly(real+(M/2), t, bfly_args), hydra_sine(cosines, t, sincos_args));
    hydra_vsub(0, hydra_bfly(real+(M/2), t, bfly_args), hydra_bfly(real, t, bfly_args), TEMPE);
    hydra_vsub(0, hydra_bfly(imag+(M/2), t, bfly_args), hydra_bfly(imag, t, bfly_args), TEMPO);
    hydra_vadd(0, hydra_bfly(real, t, bfly_args), _, TEMPE);
    hydra_vadd(0, hydra_bfly(imag, t, bfly_args), _, TEMPO);
}

```

Example: Floating-point FFT (4)

Hydra has a “complex number addressing mode” to facilitate operations on complex numbers

- It optionally negates and/or swaps alternating vector elements, causing a single VADD or VMADD to perform 90° complex rotations, etc.

This, plus a few more refinements, gives a concise FFT algorithm that uses just $4 \cdot \log_2(N)$ Hydra operations each of length N

- See `hydra4_test_fft2.c` in the Hydra 4 test suite for other variants such as an SPU-based version

```
void hydra_fft(int N, cplx_float_t *data)
{
    hydra_set_vl(N);

    extern float cosines[MAX_N/4];    // First-quadrant table of cosines.

    float temp[N];    // or temp[MAX_N] if variable length arrays cannot be used.
    int t = HYDRA_FLOAT;
    int temp_vec = hydra_set_vector(HYDRA_V7, temp, t);

    uint32_t bitrev_args = HYDRA_BITREV_STRIDE(N);
    hydra_vmov(0, temp_vec, hydra_bitrev(&data[0].re, t, bitrev_args));
    hydra_vadd(0, hydra_strided(&data[0].re, t, 2),
               hydra_complex(temp, t, HYDRA_COMPLEX_NODD),
               hydra_complex(temp, t, HYDRA_COMPLEX_SWAP));
    hydra_vmov(0, temp_vec, hydra_bitrev(&data[0].im, t, bitrev_args));
    hydra_vadd(0, hydra_strided(&data[0].im, t, 2),
               hydra_complex(temp, t, HYDRA_COMPLEX_NODD),
               hydra_complex(temp, t, HYDRA_COMPLEX_SWAP));

    for (int M = 4; M <= N; M *= 2) {
        uint32_t bfly_args = HYDRA_BFLY_M(M*2);
        uint32_t sincos_args = HYDRA_SINCOS_NM(MAX_N, M) +
                               HYDRA_SINCOS_HALF + HYDRA_SINCOS_DIV;

        hydra_vmul(0, temp_vec, hydra_bfly(&data[M/2].re, t, bfly_args),
                  hydra_cosine(cosines, t, sincos_args));
        hydra_vmadd(0, hydra_complex(temp, t, HYDRA_COMPLEX_SWAP), _,
                   hydra_bfly(&data[M/2].re, t, bfly_args),
                   hydra_sine(cosines, t, sincos_args));
        hydra_vsub(0, hydra_bfly(&data[M/2].re, t, bfly_args),
                  hydra_bfly(&data[0].re, t, bfly_args), temp_vec);
        hydra_vadd(0, hydra_bfly(&data[0].re, t, bfly_args), _, temp_vec);
    }

    hydra_wait();
}
```

Hydra 4

Architecture overview

- Introduction
- **Using Hydra**
 - Examples
 - **Hydra registers**
 - Vector instructions
 - Data types
 - Addressing modes
 - Stored program unit
 - Documentation and tools
 - Hydra 4.2 functionality
 - Performance
- Hardware and implementation

Hydra 4's registers

Hydra has 14 host-accessible control registers

Also 16 host-accessible $VnCFG$ / $VnADDR$ registers that define eight vectors $V0 \dots V7$

All these registers are architecturally 32 bits wide

- (But some registers implement fewer bits)
- When Chimera is the host CPU, it can do 16-bit MOVs or 32-bit LMOVs to/from these registers
- Non-Chimera host CPUs, which use the AHB or APB bus or the Direct host interface to access Hydra, can do 8-, 16-, or 32-bit reads or writes (usually 32 bits)

```
HYDRA_P->VL = 100;  
HYDRA_P->V0CFG = 0x00000020;  
HYDRA_P->V0ADDR = (int) results;  
HYDRA_P->V1CFG = 0x00000010;  
HYDRA_P->V1ADDR = (int) data;  
HYDRA_P->V2CFG = 0x00005220;  
HYDRA_P->V2ADDR = (int) inputs;  
while (!(HYDRA_P->IRQ & 2)) { }  
HYDRA_P->OPCODE = 0x00006210;
```

Address registers

Hydra works in terms of physical RAM addresses with byte granularity

- These are ordinary pointers for RISC-V / ARM hosts, or “long pointers” for Chimera 4.2 hosts

hydra4_config.vh defines each chip’s RAM address width, ``HYDRA_WIDTH_W = "W"`

- Hydra 4 can be built with any W from 17 to 32
- Example: $W = 17$ for Chimera’s basic 64KW RAM space or Byrd A0’s 128KB DTIM RAM
- Example: $W = 18$ on Byrd B0, allowing Hydra to address 128KB DTIM + 56KB SYS_MEM RAM
- Example: $W = 32$ if Hydra wants to see the chip’s full AHB address space

Registers that hold addresses are W bits wide

- They ignore the data written to their high $(32 - W)$ bits
- For example, if $W = 20$ and host firmware has a pointer into data RAM “0x60012345” ...
- ... the host can write 0x60012345 directly to V0ADDR, which will receive the RAM-relative 20-bit-wide address 0x12345

In Chimera systems, a compatibility feature also allows the host to use traditional Chimera 16-bit word-granular pointers (not applicable to SmartBridge or Byrd)

Hydra 4 registers (shown for $W=20$)

Name	Offset	Bit 31	23	15	7	0
OPCODE	+0x00	Write an opcode here to post a vector instruction				
VL	+0x04	0x000		Vector length		
VMA	+0x08	0x000		Vector mask address		
VPC	+0x0C	0x000		SPU program counter		
G0	+0x10	General register G0				
G1	+0x14	General register G1				
IRQ	+0x20	0x00	Error reason	Int ack	Int flags	
IEN	+0x24	0x000000				Int enable
MODES	+0x28	PRIV_ONLY	DIS/EN	0x00	Rounding	
PCOUNT	+0x2C	Performance counter				Event
STATUS	+0x30	0x0000			Inverted flags	Status flags
COMPAT_OPCODE	+0x34	0x0000			Write Hydra 3 opcode	
VMCNT	+0x38	0x000		Vector mask count		
PROPS	+0x3C	Version 4.2.1 = "0x0421"			Properties	
VnCFG	+0x40 + 8 <i>n</i>	ARG			MODE	TYPE
VnADDR	+0x44 + 8 <i>n</i>	0x000		Vector base address		

Using Hydra in multiple threads, tasks, or cores

Most Synaptics firmware is written in terms of multiple “tasks”

- These could be Chimera threads, RTOS tasks, handlers, modules, or CPU cores all running at once
- What if several of these tasks want to use Hydra at the same time?
- One answer is to arbitrate access in the RTOS: In Hydra 3, this was the only answer

Hydra 4 supports multiple *register contexts*

- Multiple views of the register set just shown
- Eventually, there will be an option for multiple “full” register sets: Expensive!
- Full contexts would allow very fluid multi-tasked use of Hydra, but they are *not yet implemented* in Hydra 4.0–4.2 for Byrd and SmartBridge

By default (and implemented now), each additional register context is *partial*

- Most registers are shared, but each context has its own OPCODE, IRQ.BOUNCE, and IRQ.DONE
- This is enough to allow tasks to safely share Hydra on the granularity of SPU routines
- Alternatively, in Chimera systems, Hydra can integrate with Chimera’s hardware mutexes

Hydra 4

Architecture overview

- Introduction
- **Using Hydra**
 - Examples
 - Hydra registers
 - **Vector instructions**
 - Data types
 - Addressing modes
 - Stored program unit
 - Documentation and tools
 - Hydra 4.2 functionality
 - Performance
- Hardware and implementation

Vector operations

Hydra 4 has a large repertoire of vector operations

- The hardware sees them as vector *instructions* for a vector processor
- To the C programmer, each vector operation looks like a function call

```
hydra_vrsqrte(hydra_set_vl(N),           // Instruction modifier(s).  
              hydra_vector(results, HYDRA_FLOAT), // Output vector.  
              hydra_vector(data, HYDRA_FLOAT));  // Input vector.
```

Almost all Hydra 4 operations take N “hclk” cycles plus a few cycles of startup overhead

- Where “ N ” is the vector length “VL”
- For matrices, N is the total number of matrix elements ($N = VL = R * C$)
- Convolutions and matrix multiplications take $N * M$ cycles (startup overhead is paid just once)
- SIMD operation, when applicable, can reduce the number of cycles to $\frac{1}{2}$ or $\frac{1}{4}$ normal (see later)
- “hclk cycles” = At the full HSO speed, e.g., ~400 MHz on Byrd, ~350 MHz on SB7900
- (On Chimera-based chips, Hydra 4, like Hydra 3, will run at twice Chimera’s pclk rate)

Vector operations: Basic arithmetic

Operation	... performs
hydra_ vadd (<i>m, dest, src1, src2</i>)	$dest[i] \leftarrow src1[i] + src2[i]$
hydra_ vsub (<i>m, dest, src1, src2</i>)	$dest[i] \leftarrow src1[i] - src2[i]$
hydra_ vsubn (<i>m, dest, src1, src2</i>)	$dest[i] \leftarrow -src1[i] - src2[i]$
hydra_ vmul (<i>m, dest, src1, src2</i>)	$dest[i] \leftarrow src1[i] * src2[i]$
hydra_ vmuln (<i>m, dest, src1, src2</i>)	$dest[i] \leftarrow -src1[i] * src2[i]$
hydra_ vneg (<i>m, dest, src</i>)	$dest[i] \leftarrow -src[i]$

"*m*" = instruction setup or modifier bits (0 if no setup/modifiers needed)

The operands *dest*, *src1*, *src2*, and *src3* may have any combination of data types and addressing modes

Vector operations: Fused multiply-add

Operation	... performs
hydra_ vmadd (<i>m, dest, src1, src2, src3</i>)	$dest[i] \leftarrow src1[i] + (src2[i] * src3[i])$
hydra_ vmsub (<i>m, dest, src1, src2, src3</i>)	$dest[i] \leftarrow src1[i] - (src2[i] * src3[i])$
hydra_ vmsubr (<i>m, dest, src1, src2, src3</i>)	$dest[i] \leftarrow -src1[i] + (src2[i] * src3[i])$
hydra_ vmsubn (<i>m, dest, src1, src2, src3</i>)	$dest[i] \leftarrow -src1[i] - (src2[i] * src3[i])$

These do the full computation, $a + b*c$, without rounding the intermediate product

- The “fused multiply-add” operation has many useful properties: See the *Hydra 4 User’s Guide*

Hydra’s VMADD is just as fast as a VMUL or a VADD alone

- The Multiply and the Add happen simultaneously, in different stages of Hydra’s pipeline

In the C language, the standard function **fmaf**(*b, c, a*) does a fused multiply-add

Vector operations: Multiply-accumulate

Operation	... performs
hydra_ vsum (<i>m, dest, src</i>)	$dest \leftarrow \sum_i src[i]$
hydra_ vmac (<i>m, dest, src1, src2</i>)	$dest \leftarrow \sum_i (src1[i] * src2[i])$ "Fast mult-accum"
hydra_ vdmac (<i>m, dest, src1, src2</i>)	$dest \leftarrow \sum_i (src1[i] * src2[i])$ "Deterministic"
hydra_ vomac (<i>m, dest, src1, src2</i>)	$dest \leftarrow \sum_i (src1[i] * src2[i])$ "Fully ordered"
hydra_ vcmac (<i>m, dest, src1, src2</i>)	$dest[i] \leftarrow \sum_{j \leq i} (src1[j] * src2[j])$ "Cumulative"
hydra_ vconvo (<i>m, dest, src1, src2</i>)	Use VMAC as the kernel in a convolution
hydra_ vmm (<i>m, dest, src1, src2</i>)	Use VMAC as the kernel in a matrix-multiply

Hydra offers several flavors of this operation

- VMAC, VOMAC, and VDMAC give three different useful trade-offs of speed and precision vs. ease of matching the result (with bit-for-bit the same rounding) in non-Hydra firmware code
- See the detailed description of Hydra's ALU later in this presentation

Vector operations: Division and square root

Operation	... performs
<code>hydra_vrcpe(m, dest, src)</code>	$dest[i] \leftarrow 1 / src[i]$ "Estimate accurate to 6 bits"
<code>hydra_vrcpe2(m, dest, src1, src2)</code>	$dest[i] \leftarrow 1 / src1[i]$ with special treatment of extreme values for $src2[i] / src1[i]$
<code>hydra_vrsqrte(m, dest, src)</code>	$dest[i] \leftarrow 1 / \sqrt{src[i]}$ "Estimate to 3 bits"
<code>hydra_vrsqrte2(m, dest, src)</code>	$dest[i] \leftarrow 1 / \sqrt{src[i]}$ with special treatment of extreme values for $\sqrt{src[i]}$
<code>hydra_vdiv(m, dest, src1, src2)</code>	$dest[i] \leftarrow src1[i] / src2[i]$ (Red = not yet impl'd)

VDIV is not implemented in Hydra 4.0–4.2, but simple routines based on VRCPE can do:

- Accurate floating-point division in 6 cycles per quotient (*much* faster than VDIV would be!)
- IEEE compliant exactly rounded floating-point division in 10 cycles per quotient
- Exact integer division in 11 cycles per quotient (or 7 cycles for short integers)

Vector operations: Other arithmetic

Operation	... performs
<code>hydra_vadd3(m, dest, src1, src2, src3)</code>	$dest[i] \leftarrow src1[i] + src2[i] + src3[i]$ (integers)
<code>hydra_vavg(m, dest, src1, src2)</code>	$dest[i] \leftarrow (src1[i] + src2[i]) / 2$
<code>hydra_vsubh(m, dest, src1, src2)</code>	$dest[i] \leftarrow (src1[i] - src2[i]) / 2$
<code>hydra_vmulh(m, dest, src1, src2)</code>	$dest[i] \leftarrow (src1[i] * src2[i]) / 2$
<code>hydra_vabs(m, dest, src)</code>	$dest[i] \leftarrow abs(src[i])$
<code>hydra_vasub(m, dest, src1, src2)</code>	$dest[i] \leftarrow abs(src1[i] - src2[i])$
<code>hydra_vcsign(m, dest, src1, src2)</code>	$dest[i] \leftarrow abs(src1[i]) * (src2[i] < 0 ? -1 : 1)$
<code>hydra_vshl(m, dest, src1, src2)</code>	$dest[i] \leftarrow src1[i] * (2^{src2[i]})$ (integer <code>src2</code>)
<code>hydra_vshr(m, dest, src1, src2)</code>	$dest[i] \leftarrow src1[i] * (2^{-src2[i]})$ (integer <code>src2</code>)
<code>hydra_vlog2(m, dest, src1, src2)</code>	$dest[i] \leftarrow floor(log_2(src1[i])) + src2[i]$

Vector operations: Other arithmetic

Operation	... performs
<code>hydra_vadd3(m, dest, src1, src2, src3)</code>	$dest[i] \leftarrow src1[i] + src2[i] + src3[i]$ (integers)
<code>hydra_vavg(m, dest, src1, src2)</code>	$dest[i] \leftarrow (src1[i] + src2[i]) / 2$
<code>hydra_vsubh(m, dest, src1, src2)</code>	$dest[i] \leftarrow (src1[i] - src2[i]) / 2$
<code>hydra_vsubf(m, dest, src1, src2)</code>	$dest[i] \leftarrow (src1[i] - src2[i]) / 2$
<code>hydra_vavgf(m, dest, src1, src2)</code>	$dest[i] \leftarrow (src1[i] + src2[i]) / 2$
<code>hydra_vlgn(m, dest, src1, src2)</code>	$dest[i] \leftarrow \text{floor}(\log_2(src1[i])) * (src2[i] < 0 ? -1 : 1)$
<code>hydra_vshl(m, dest, src1, src2)</code>	$dest[i] \leftarrow src1[i] * (2^{src2[i]})$ (integer <code>src2</code>)
<code>hydra_vshr(m, dest, src1, src2)</code>	$dest[i] \leftarrow src1[i] * (2^{-src2[i]})$ (integer <code>src2</code>)
<code>hydra_vlog2(m, dest, src1, src2)</code>	$dest[i] \leftarrow \text{floor}(\log_2(src1[i])) + src2[i]$

When operating on integers, VSHL and VSHR are bitwise shifts, and VLOG2 finds the leftmost '1' bit

When operating on floats, VSHL and VSHR scale the exponent and VLOG2 extracts the exponent

Vector operations: Data motion

Operation	... performs
hydra_ vmov (<i>m, dest, src</i>)	$dest[i] \leftarrow src[i]$
hydra_ vmovn (<i>m, dest, src</i>)	$dest[i] \leftarrow src[i]$ "Move with no side effects"
hydra_ vmovlr (<i>m, dest, src</i>)	$dest[i] \leftarrow src[i]$ "Move in left-to-right order"
hydra_ vmov2 (<i>m, dest1, dest2, src1, src2</i>)	$dest1[i] \leftarrow src1[i]$ and $dest2[i] \leftarrow src2[i]$
hydra_ vmov4 (<i>m, dest1, dest2, dest3, dest4</i>)	$dest1[i], dest2[i], dest3[i], dest4[i] \leftarrow G0$
hydra_ vxch (<i>m, dest1, dest2</i>)	$dest1[i] \leftarrow dest2[i]$ and $dest2[i] \leftarrow dest1[i]$
hydra_ vsel (<i>m, dest, src1, src2</i>)	$dest[i] \leftarrow mask[i] ? src2[i] : src1[i]$
hydra_ vselq (<i>m, dest, src1, src2, src3</i>)	$dest[i] \leftarrow$ Sine/Cosine argument reduction

Vector operations: Gather/compact

Operation	... performs
<code>hydra_vmovg(m, dest, k, src1, src2)</code>	$dest[i] \leftarrow src1[src2[i]*k]$ "Gather under index"
<code>hydra_vmovs(m, dest, k, src1, src2)</code>	$dest[src2[i]*k] \leftarrow src1[i]$ "Scatter under index"
<code>hydra_vmovc(m, dest, src, limit)</code>	$dest[j++] \leftarrow src[i]$ "Compact under mask"
<code>hydra_vmovc2(m, dest1, dest2, src, limit)</code>	$dest1[j++] \leftarrow src[i]$ else $dest2[k++] \leftarrow src[i]$
<code>hydra_vmovx(m, dest, src)</code>	$dest[i] \leftarrow src[j++]$ "Expand under mask"
<code>hydra_vmovx2(m, dest, src1, src2)</code>	$dest[i] \leftarrow src1[j++]$ else $dest[i] \leftarrow src2[k++]$

Vector operations: Bitwise operations

Operation	... performs
hydra_ vand (<i>m, dest, src1, src2</i>)	$dest[i] \leftarrow src1[i] \& src2[i]$
hydra_ vandn (<i>m, dest, src1, src2</i>)	$dest[i] \leftarrow src1[i] \& \sim src2[i]$
hydra_ vnand (<i>m, dest, src1, src2</i>)	$dest[i] \leftarrow \sim(src1[i] \& src2[i])$
hydra_ vor (<i>m, dest, src1, src2</i>)	$dest[i] \leftarrow src1[i] \mid src2[i]$
hydra_ vorn (<i>m, dest, src1, src2</i>)	$dest[i] \leftarrow src1[i] \mid \sim src2[i]$
hydra_ vnor (<i>m, dest, src1, src2</i>)	$dest[i] \leftarrow \sim(src1[i] \mid src2[i])$
hydra_ vxor (<i>m, dest, src1, src2</i>)	$dest[i] \leftarrow src1[i] \wedge src2[i]$
hydra_ vxnor (<i>m, dest, src1, src2</i>)	$dest[i] \leftarrow \sim(src1[i] \wedge src2[i])$
hydra_ vnot (<i>m, dest, src</i>)	$dest[i] \leftarrow \sim src[i]$

Vector operations: More bitwise operations

Operation	... performs
<code>hydra_vbool(m, dest, src1, src2, src3)</code>	$dest[i] \leftarrow f_{G0}(src1[i], src2[i], src3[i], mask[i])$ bitwise
<code>hydra_vandv(m, dest, src)</code>	$dest \leftarrow \&_i(src[i])$ "Bitwise AND across N words"
<code>hydra_vorv(m, dest, src)</code>	$dest \leftarrow _i(src[i])$
<code>hydra_vxorv(m, dest, src)</code>	$dest \leftarrow ^_i(src[i])$
<code>hydra_vbrev(m, dest, src1, src2)</code>	$dest[i] \leftarrow \text{bit reverse/rearrange}_{src2[i]}(src1[i])$
<code>hydra_vpop(m, dest, src)</code>	$dest \leftarrow \sum_i \text{popcount}(src[i])$ "Count '1' bits"
<code>hydra_vham(m, dest, src1, src2)</code>	$dest \leftarrow \sum_i \text{popcount}(src1[i] \wedge src2[i])$
<code>hydra_vclass(m, dest, src1, src2)</code>	$dest[i] \leftarrow \text{classify_number}(src1[i] \& src2[i])$

Vector operations: Comparison

Operation	... performs
hydra_ vcmp _cond(<i>m</i> , <i>dest</i> , <i>src1</i> , <i>src2</i>)	$dest[i] \leftarrow src1[i] \Leftrightarrow src2[i] ? 1 : 0$
hydra_ vcmpm _cond(<i>m</i> , <i>mask</i> , <i>src1</i> , <i>src2</i>)	$mask[i] \leftarrow src1[i] \Leftrightarrow src2[i] ? 1 : 0$
hydra_ vcmpsel _cond(<i>m</i> , <i>dest</i> , <i>src1</i> , <i>src2</i>)	$dest[i] \leftarrow src1[i] \Leftrightarrow G0 ? G1 : src2[i]$
hydra_ vnex t_cond(<i>m</i> , <i>dest</i> , <i>src1</i> , <i>src2</i>)	$dest \leftarrow \text{first } i \text{ for which } src1[i] \Leftrightarrow src2[i]$
hydra_ vfind _cond(<i>m</i> , <i>dest</i> , <i>src1</i> , <i>src2</i>)	$dest[j++] \leftarrow i \text{ if } src1[i] \Leftrightarrow src2[i]$
hydra_ vfindt _cond(<i>m</i> , <i>dest</i> , <i>src1</i> , <i>src2</i>)	$dest[j++] \leftarrow i \text{ if } src1[i] \Leftrightarrow src2[i]$ and $!(src1[i-1] \Leftrightarrow src2[i-1])$
Hydra_ vcount _cond(<i>m</i> , <i>dest</i> , <i>src1</i> , <i>src2</i>)	$dest \leftarrow \text{count of matches } src1[i] \Leftrightarrow src2[i]$

“cond” may be: eq, ne, lt, le, gt, ge
 ltl, ltlt, ltlt, ltlt, ltlt, ltlt, ltlt, ltlt
 aeq, ane, alt, ale, agt, age, a[n]ltlt, a[n]ltlt
 class, nclass, ord, unord

(Basic numeric comparisons)
 ($0 < src1[i] < src2[i]$, etc.)
 (Compare by absolute value)
 (Number classification)

Vector operations: Min and Max

Operation	... performs
<code>hydra_vmin(m, dest, src1, src2)</code>	$dest[i] \leftarrow \min(src1[i], src2[i])$
<code>hydra_vamin(m, dest, src1, src2)</code>	$dest[i] \leftarrow \min(src1[i], src2[i])$ "Abs compare"
<code>hydra_v[a]max(m, dest, src1, src2)</code>	$dest[i] \leftarrow \max(src1[i], src2[i])$
<code>hydra_vminmax(m, dest1, dest2, src1, src2)</code>	$dest1[i] \leftarrow \min(src1[i], src2[i])$ and $dest2[i] \leftarrow \max(src1[i], src2[i])$ "Sort a pair"
<code>hydra_v[a]minv(m, dest, src)</code>	$dest \leftarrow \min_i (src[i])$ "Min value across vector"
<code>hydra_v[a]maxv(m, dest, src)</code>	$dest \leftarrow \max_i (src[i])$
<code>hydra_v[a]mini(m, dest, src)</code>	$dest \leftarrow \arg \min_i (src[i])$ "Index of min"
<code>hydra_v[a]maxi(m, dest, src)</code>	$dest \leftarrow \arg \max_i (src[i])$

Vector operations: Sorting and median filters

Operation	... performs
<code>hydra_vmsort(m, dest, src1, src2)</code>	$dest[i] \leftarrow \text{smaller of } src1[j++], src2[k++]$
<code>hydra_vpsort(m, dest1, dest2)</code>	swap as needed to establish $dest1[i] \leq dest2[i]$
<code>hydra_vmed(m, dest, src, src_k)</code>	$dest \leftarrow \text{median}_i(src[i], k)$ "k'th order statistic"
<code>hydra_vmedr(m, dest, src, src_k)</code>	$dest \leftarrow \text{median}_i(src[i], N-k)$
<code>hydra_vcbn(m, dest, src1, src2)</code>	$dest[i] \leftarrow \text{classify_in_bins}(src1[], src2[i])$

VMSORT (in Hydra 4.2 and later) implements a Merge Sort in about $N * \log_2(N)$ cycles

- Also, VCMP + VMOVC2 implement the partitioning step of a Quicksort or Quickselect

VMED is not yet implemented in Hydra 4.0–4.2 for Byrd and SmartBridge

- Hydra's VPSORT and VMSORT instructions are enough to construct a median filter that is much faster than any firmware implementation (17–24 cycles per output element, for a 3×3 filter)
- ... though not as fast as VMED will be (9 cycles per output element)

Vector operations: Cryptographic

Operation	... performs
hydra_ vcrc (<i>m, dest, src1, src2</i>)	$dest \leftarrow CRC_i(src1[i])$ "Cyclic redundancy check"
hydra_ vrand (<i>m, dest, src1, src2</i>)	$dest[i] \leftarrow CRC_{j \leq i}(src1[j])$ "Random numbers"
hydra_ vxmadd (<i>m, dest, src1, src2, src3</i>)	"Extended-precision integer multiply-add"
hydra_ vclmul (<i>m, dest, src1, src2</i>)	"Extended-precision carry-less multiplication"
hydra_ vaes (<i>m, dest, src1, src2</i>)	"AES-128/-192/-256 encryption / decryption"
hydra_ vxxttea (<i>m, dest, src1, src2</i>)	"XXTEA 128-bit encryption / decryption"
hydra_ vsha2m (<i>m, dest, src1, src2</i>)	"SHA-256/-224 message schedule phase"
hydra_ vsha2h (<i>m, dest, src1, src2</i>)	"SHA-256/-224 hash calculation phase"

- VCRC computes a CRC32 (or narrower) hash with any CRC polynomial
- VRAND uses the same CRC machinery, as a 32-bit LFSR, to generate pseudo-random numbers
- VXMADD accelerates the main calculation in public-key DSA/RSA cryptographic systems
- VAES, VCLMUL, and VSHA2M/H implement widely used NIST standards for encryption and hashing

Other operations: Controlling Hydra

Operation	... performs
hydra_ vstart (<i>m</i>)	Start the stored program unit (SPU) at the program address given by VPC
hydra_ vcall (<i>m</i> , <i>n</i>)	Start SPU at subroutine # <i>n</i> in program library
hydra_ vnop (<i>m</i>)	No operation
hydra_wait()	Wait for Hydra to be Idle (all instrs are done)
hydra_wready()	Wait for Hydra to be Ready (to post an instr)
HYDRA_RESET()	Reset Hydra to idle state

Instruction timing

Most Hydra 4 instructions run at a speed of one iteration per hclk cycle

- If $VL = N \Rightarrow (N + T_{SU})$ hclk cycles: N plus a few (" T_{SU} ") cycles of instruction startup overhead
- Same for all data types, all addressing modes, etc.

A few instructions take more time for complex calculations:

- VOMAC: $4 * N$ hclks (for special uses only; normal VMAC runs at full speed)
- Convolutions and matrix multiplications: $N * M$ cycles
- VDIV: **TBD** (not yet implemented in Hydra 4.0–4.2)
- VCRC: 2 hclks per input byte
- VRAND: 1–8 hclks per generated 32-bit result (depending on desired quality of randomness)
- VCLMUL: 8 hclks per 32-bit result
- VAES: 10.25 hclks per input byte (AES-128)
- VXXTEA: 9.75 hclks per input byte
- VSHA2: 4.875 hclks per input byte

Instruction startup overhead

How many cycles of overhead " T_{SU} " are there?

- It depends on factors including the instruction, addressing modes, and memory alignment, and on how Hydra is configured in a chip
- For a simple arithmetic instruction (e.g., VADD, VMUL, or VMADD) with simple linear vectors:
 - In Hydra 4.0 on Byrd A and Smartbridge: $T_{SU} = 14$ cycles
 - In Hydra 4.2 on Byrd B and future chips: $T_{SU} = 9$ or 10 cycles
- Therefore, T_{SU} has a small effect if N is very large, but overhead can dominate as N gets small
 $\Rightarrow$ Hydra is meant to be used with reasonably large vectors, e.g., at least a dozen elements
- A future version of Hydra 4 will include "instruction joining" to help reduce the impact of T_{SU} (see later)

Where does the overhead come from?

- Hydra has a deep pipeline
- This allows it to clock very fast, but it creates a long latency from input to output

Vector processing pipeline

Hydra's net throughput is one vector element per cycle, but each vector element passes through many stages of processing from input to output:

1. Generate address and bank arbitration for *src* channel
2. Send read request to RAM
3. Receive read data from RAM
4. Unpack data in ALU according to its type
5. ALU multiplier first stage
6. ALU multiplier second stage
7. ALU adder
8. ALU normalization
9. ALU round, saturate, and format result
10. Pack result for *dest* channel
11. Bank arbitration for *dest* channel
12. Send write request to RAM
13. Write occurs in RAM

In the middle of a long vector operation, all 13 of these stages are busy during every cycle, with 13 successive vector elements "in flight"

Although this may seem like many stages, it is comparable to the E76 CPU's 8-stage pipeline, accounting for the fact that Hydra can load, do arithmetic, and store all at once



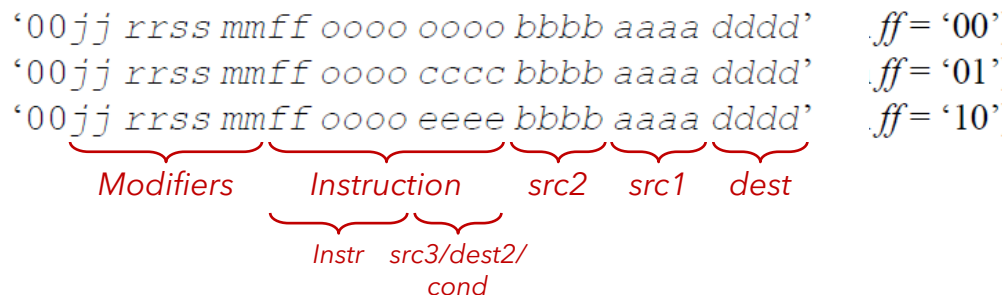
Vector operations: Instruction modifiers “*m*”

Operation	... performs
hydra_set_vl(<i>n</i>)	$VL \leftarrow n$
hydra_set_vma(<i>mask</i>)	$VMA \leftarrow address_of(mask)$
hydra_mask_if(<i>mask</i>)	$VMA \leftarrow address_of(mask)$, enable masking
hydra_mask_else(<i>mask</i>)	$VMA \leftarrow address_of(mask)$, enable complemented masking
hydra_mask_xm(<i>mask</i>)	$VMA \leftarrow address_of(mask)$, enable “extra mask”
hydra_rz()	Force round-toward-zero mode for this instr
hydra_ru()	Force round-upward (“ceiling”) for this instr
hydra_rd()	Force round-downward (“floor”) for this instr
hydra_rn(), _rnu(), _rna(), _ra()	Force other rounding modes for this instr
hydra_convo(), hydra_mm()	Convolution or matrix-multiply mode applied to any scalar-producing instruction

Opcode format

Every Hydra 4 instruction is defined by a 30-bit opcode

- Firmware writes this opcode to Hydra's OPCODE register to start a vector operation



Hydra 4's wide opcode allows a clean format with three or four operand fields and room for many instructions

All the operands *dest*, *src1*, *src2*, and *src3/dest2* support the same choices: Any vector *Vn*, or any scalar in a *Gn* register

- "Scalar" ⇔ Same value for all "i"

Plus modifier bits that control vector masking, flags, rounding, and overlapped execution

<i>aaaa, bbbb, dddd, or eeee</i>	<i>Symbol</i>	<i>Meaning</i>
'0mm'	HYDRA_Vn	Vector or memory-scalar source or destination in <i>Vn</i> (V0–V7) (as configured by <i>VnCFG</i> and addressed by <i>VnADDR</i>).
'100n'	HYDRA_FGn	Scalar single-precision floating-point source or destination in <i>Gn</i> (G0–G1).
'1010'	(none)	<i>Reserved</i> (for possible G2).
'1011'	HYDRA_ONE	Scalar constant 1 for <i>src1</i> , <i>src2</i> , or <i>src3</i> operand; <i>reserved</i> and invalid in <i>dest</i> or <i>dest2</i> operand.
'110n'	HYDRA_IGN	Scalar 32-bit signed saturating integer source or destination in <i>Gn</i> (G0–G1).
'1110'	(none)	<i>Reserved</i> (for possible G2).
'1111'	HYDRA_ZERO	Scalar constant zero for <i>src1</i> , <i>src2</i> , or <i>src3</i> operand; a null vector or scalar in <i>dest</i> or <i>dest2</i> operand.

Vector instructions

There is plenty of room still free in the space of $256 + 40 = 296$ possible instruction opcodes

- We can make future improvements without cramming new instructions into odd corners of a cramped encoding (as happened in Hydra 3)

<i>ff0000...</i>	<i>...000</i>	<i>111</i>	<i>110</i>	<i>101</i>	<i>100</i>	<i>011</i>	<i>010</i>	<i>001</i>	<i>000</i>
<i>00 0000 0</i>		VSUB	VADD	—	VMOV	—	—	VNOP	<i>Special SPU</i>
<i>00 0000 1</i>		—	—	—	—	—	VMAC	VMULN	VMUL
<i>00 0001 0</i>		—	—	—	VMOVS	VSEL	VMOVL	VMOVN	VMOVG
<i>00 0001 1</i>		VMOVS8	VMOVS4	VMOVS2	VMOVS1	VMOVG8	VMOVG4	VMOVG2	VMOVG1
<i>00 0010 0</i>		—	—	—	—	—	—	—	—
<i>00 0010 1</i>		—	—	—	—	VMSORT	VMOVX2	VMOVX	VMOV
<i>00 0011 0</i>		VRSQRTE2	VRCPE2	VRSQRTE	VRCPE	VASUB	VMULH	VSUBH	VAVG
<i>00 0011 1</i>		VCSIGN	VCLASS	VDIV	—	VBREV	VLOG2	VSHR	VSHL
<i>00 0100 0</i>		VXNOR	VNOR	VORN	VOR	VXOR	VNAND	VANDN	VAND
<i>00 0100 1</i>		—	VCBIN	VMEDR	VMED	VAMAX	VAMIN	VMAX	VMIN
<i>00 0101 0</i>		VAMAXV	VAMINV	VMAXV	VMINV	VAMAXI	VAMINI	VMAXI	VMINI
<i>00 0101 1</i>		—	—	VCRC	VHAM	—	VXORV	VORV	VANDV
<i>00 0110 0</i>		VAES8	VAES4	VRAND	VCLMUL	VSHA2H	VSHA2M	VXXTEA	VAES
<i>00 0110 1</i>		—	—	—	—	—	VCMAC	VDMAC	VOMAC

<i>ff0000...</i>	<i>...eeee</i>	<i>1111</i>	<i>1110</i>	<i>1101</i>	<i>1100</i>	<i>1011</i>	<i>1010</i>	<i>1001</i>	<i>1000</i>	<i>0111</i>	<i>0110</i>	<i>0101</i>	<i>0100</i>	<i>0011</i>	<i>0010</i>	<i>0001</i>	<i>0000</i>
<i>10 0000</i>																	VMADD
<i>10 0001</i>																	VMSUB
<i>10 0010</i>																	VMSUBR
<i>10 0011</i>																	VMSUBN
<i>10 0100</i>																	VMOV2
<i>10 0101</i>																	VMOV2C
<i>10 0110</i>																	VBOOL
<i>10 0111</i>																	VSELQ
<i>10 1000</i>																	VXMADD
<i>10 1001</i>																	VMINMAX
<i>10 1010</i>																	VADD3
<i>10 1011</i>																	VMOV4
<i>10 1100</i>																	—
<i>10 1101</i>																	—

<i>ff0000...</i>	<i>...cccc</i>	<i>1111</i>	<i>1110</i>	<i>1101</i>	<i>1100</i>	<i>1011</i>	<i>1010</i>	<i>1001</i>	<i>1000</i>	<i>0111</i>	<i>0110</i>	<i>0101</i>	<i>0100</i>	<i>0011</i>	<i>0010</i>	<i>0001</i>	<i>0000</i>
<i>01 000 0</i>																	VCOMP
<i>01 000 1</i>																	
<i>01 001 0</i>																	VNEXT
<i>01 001 1</i>																	
<i>01 010 0</i>																	VFIND
<i>01 010 1</i>																	
<i>01 011 0</i>																	VFINDT
<i>01 011 1</i>																	
<i>01 100 0</i>																	VCMPESEL
<i>01 100 1</i>																	
<i>01 101 0</i>																	VCOUNT
<i>01 101 1</i>																	
<i>01 110 0</i>																	—
<i>01 110 1</i>																	

Vector mask

Any instruction can generate a vector mask from any selected status flag

- Vector compare (VCMP) is just an instruction that generates 0-or-1 bits that reflect various comparisons

Any instruction can be conditional on the true or complemented mask

- Masking VCMP itself forms an AND or OR combination of conditions

The vector mask also serves as “carry” or “borrow” bits for arithmetic

- But because Hydra 4 can work directly with 32-bit integers, we will rarely need it

VM_EN and FLAG_EN are no longer “sticky” modes, as they were in Hydra 3

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
NNEGATIVE	NZERO	NINFINITE	NDIVZERO	NINEXACT	NINVALID	NUNDERFLOW	NOVERFLOW	NEGATIVE	ZERO	INFINITE	DIVZERO	INEXACT	INVALID	UNDERFLOW	OVERFLOW
R/W 0	R/W 0	R/W 0	R/W 0	R/W 0	R/W 0	R/W 0	R/W 0	R/W 0	R/W 0	R/W 0	R/W 0	R/W 0	R/W 0	R/W 0	R/W 0

STATUS

<i>rrss</i>	<i>Symbol</i>	<i>Meaning</i>
'0000'	(none)	No status vector and no forced rounding. The destination is an ordinary result vector or scalar, and the status flags are only summarized in STATUS. Arithmetic rounding uses the current rounding mode in MODES.ROUND.
'0001'	(none)	Reserved.
'0010'	HYDRA_FV	Generate a status flags vector. The destination vector is filled with per-element status words instead of result data, and the result data is discarded. Arithmetic rounding uses the current rounding mode in MODES.ROUND.
'0011'	HYDRA_FM	Generate a status flag mask. The instruction writes a new vector mask, drawing from each element a status flag bit selected by the dddd field. The result data is discarded. Arithmetic rounding uses the current rounding mode in MODES.ROUND.

<i>mm</i>	<i>Symbol</i>	<i>Meaning</i>
'00'	(none)	Unmasked. All vector elements are active, and the instruction does not read the vector mask.
'01'	HYDRA_CIN	Carry-in. The instruction reads the vector mask but uses it in special ways depending on the instruction.
'10'	HYDRA_IF	Masked. The instruction reads the vector mask, one bit per vector element, and only elements corresponding to '1' mask bits are active.
'11'	HYDRA_ELSE	Masked complemented. The instruction reads the vector mask, and only elements corresponding to '0' mask bits are active.

Indexed addressing and sparse masks

Indexed addressing uses one vector as an index into another: $vec1[vec2[i]]$

- Hydra 3 allowed all three operands at once ($dest, src1, src2$) to use indexed addressing
- Seems cool... But it is very expensive in hardware, totally unlike any other vector architectures, and never used in real Synaptics firmware!

Hydra 4 replaces this with two specialized indexing instructions

- VMOVG (move and gather) and VMOVS (move and scatter)
- Just one indexing path in hardware that uses the $src2$ RAM channel for indexes: *Much* simpler!

Only one use was ever suggested for Hydra 3's "indexed-everything" mode:

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13
$vm[i]$	1	0	0	0	1	1	0	0	1	0	0	0	1	0

- Use indexing to operate on sparse data, e.g., selected by a rarely-true condition
- On Hydra 3, this would be more efficient than vector masks if the mask has only sparse '1's
- Hydra 4's solution: *Handle sparse vector masks more efficiently*
- This was the right solution all along! Sparse vector masks are one of those things that people requested for years... but I could never implement because Hydra 3's logic was *just too obscure*

Other things done differently from Hydra 3

No all-in-one “VFFT” instruction for Fast Fourier Transforms

- Instead, Hydra 4 has a set of FFT addressing modes that allow firmware to implement the FFT algorithm efficiently (see the example earlier)
- This fits better into Hydra’s pipeline structure $\Rightarrow$ Significantly *faster* than Hydra 3’s VFFT
 - About $4 N \log_2(N)$ hclk cycles + $O(\log_2(N))$ overhead for a floating-point FFT
- Each mode can be calculated cheaply $\Rightarrow$ FFTs are a standard capability, not a high-cost option
- Much more flexible $\Rightarrow$ We can finally provide non-power-of-two strides in FFTs

Hydra 3’s VGEN instruction becomes “pattern mode” in Hydra 4

- The address pattern of any addressing mode can be used as a sequence of integers
- Basically, the addresses bypass the RAM and are used directly as data
- Different addressing modes can generate: Consecutive integers, arithmetic sequences, scalar constants, sawtooth waves, triangle waves, stairsteps, the bit-reversal function, and more
- Any instruction can use a pattern as any source operand
- Example: Tabulating a function using its power series

Other things done differently (2)

Convolutions are generalized into a flexible “convolution mode”

- *All* accumulation-style instructions can work in convolution mode (both 1D and 2D)
- Convolver VMAC creates an FIR filter
- Convolver VMAC with packed data gives a fast + compact neural network!
- Convolver VMED creates a median filter
- Convolver VMAXV or VMINV creates a max or min filter or a neural net “maxpool” layer
- Convolver VHAM optimizes our “Hamming distance” fingerprint matching loop, etc...

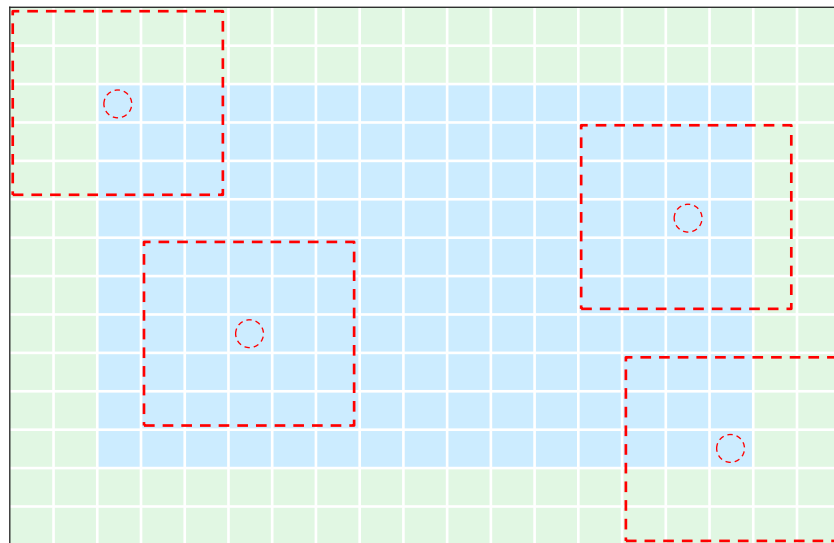
Matrix multiplication also generalizes to work with any two-input accumulation

- Applied to VMAC, this gives a standard matrix multiply
- Applied to VHAM, it performs Hydra 3’s VMHAM operation
- Applied to VNEXT, it granularizes vector values using a table of thresholds
- Etc...

Padded convolutions

Currently, Hydra 3 and 4.0–4.2 do not support automatic “padding” in convolutions

- For example, convolving a 5x5 kernel  over a 14x19 image  gives a 10x15 result  with just the interior pixels  where the kernel entirely overlaps the input image
- Often, we have a 10x15 input image that we want to filter into another 10x15 image ...
- ... where the border pixels  are padded, e.g., with implicit zeroes or duplicate values
- Currently, users must implement such padding “by hand” in firmware
 - Compute the edge pixels in a firmware loop, or using Hydra 1D ops, or create an explicitly padded image
- The *Hydra 4 User’s Guide* defines a feature to do padding in hardware, but it is not yet implemented in Hydra 4.0–4.2
 - There just wasn’t time for Byrd / SmartBridge



Hydra 4

Architecture overview

- Introduction
- **Using Hydra**
 - Examples
 - Hydra registers
 - Vector instructions
 - **Data types**
 - Addressing modes
 - Stored program unit
 - Documentation and tools
 - Hydra 4.2 functionality
 - Performance
- Hardware and implementation

Data types

Each source and destination vector has a data type

Most Hydra 4 instructions accept any mix of types in their inputs and result

Some things that were special instructions in Hydra 3 are now just cases of VMOV

- For example, VFFLOAT “convert integers to floats” and VFROUND “round floats to integers” now can be done simply as VMOV’s with different *src* and *dest* types

Many things that were modes in Hydra 3 (SIGNED, SHIFT, SATURATE, INT32) are now just part of the vector type

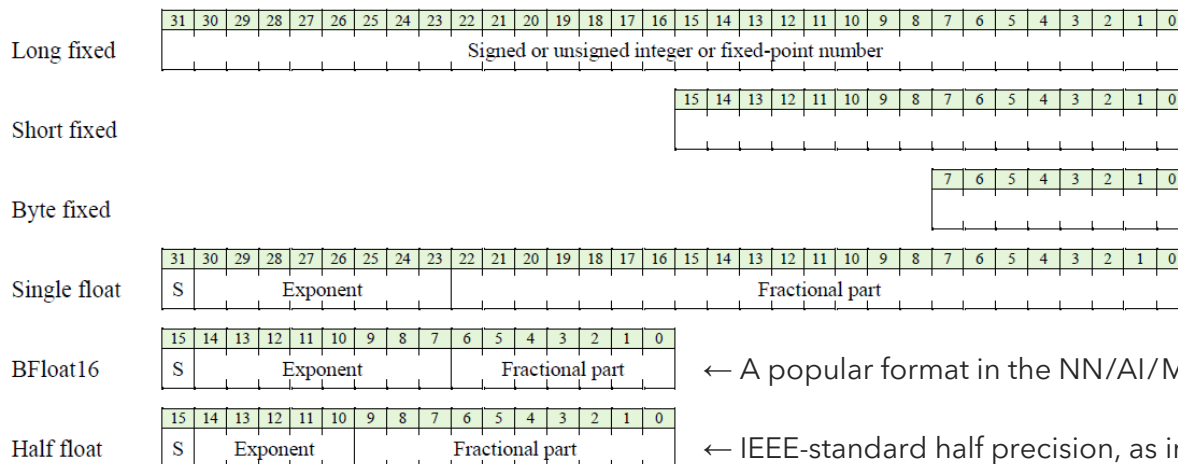
The only real “mode” now is MODES.ROUND, which firmware will hardly ever need to change

Numbers

Hydra 4 supports a variety of data types:

- Fully IEEE-compatible single-precision floats
- Two flavors of 16-bit short floats
- Signed and unsigned 8-, 16-, and 32-bit integers and fixed-point numbers
- Packed K -bit integers, fixed, or “Posit” floats for any K from 1 to 32

E	F	Value “ x ”	Description
$1 \leq E \leq 254$	Any	$(-1)^S * 2^{(E-150)} * (2^{23} + F)$ $= (-1)^S * 2^{(E-127)} * (1 + f)$	Normalized non-zero finite number from $\pm 2^{-126}$ to $\pm(2-2^{-23}) * 2^{127}$.
$E = 0$	$F \neq 0$	$(-1)^S * 2^{-149} * F$ $= (-1)^S * 2^{-126} * f$	Denormal non-zero finite number from $\pm 2^{-149}$ to $\pm(1-2^{-23}) * 2^{-126}$.
$E = 0$	$F = 0$	$(-1)^S * 0.0$	Signed zero, ± 0 .
$E = 255$	$F = 0$	$(-1)^S * \infty$	Signed infinity, $\pm \infty$.
$E = 255$	$F \neq 0$	NaN	“Not a Number.”



← A popular format in the NN/AI/ML community

← IEEE-standard half precision, as in RISC-V and ARM

Data types: Signed integers

Symbol	Data values
HYDRA_INT32	32-bit signed integers
HYDRA_INT16	16-bit signed integers
HYDRA_INT8	8-bit signed integers
HYDRA_INT	Signed integers of this machine's "int" type
HYDRA_LONG	Signed integers of this machine's "long" type
HYDRA_SHORT	Signed integers of this machine's "short" type
HYDRA_CHAR	Signed integers of this machine's "char" type

These are *saturating* types: An out-of-range result will saturate to the most positive or most negative representable number

Data types: Unsigned integers

Symbol	Data values
HYDRA_UINT32	32-bit unsigned integers
HYDRA_UINT16	16-bit unsigned integers
HYDRA_UINT8	8-bit unsigned integers
HYDRA_UINT	Unsigned integers of machine's "int" type
HYDRA_ULONG	Unsigned integers of machine's "long" type
HYDRA_USHORT	Unsigned integers of machine's "short" type
HYDRA_UCHAR	Unsigned integers of machine's "char" type

These, too, are *saturating* types: An out-of-range result will saturate to the most positive representable number (or to 0 if negative)

Data types: Truncating unsigned integers

Symbol	Data values
HYDRA_UTRUNC32	32-bit truncating unsigned integers
HYDRA_UTRUNC16	16-bit truncating unsigned integers
HYDRA_UTRUNC8	8-bit truncating unsigned integers
HYDRA_LWORD	Raw long-words of bits (same as UTRUNC32)
HYDRA_SWORD	Raw short-words of bits
HYDRA_BYTE	Raw bytes of bits

These truncating types take the low K bits of a result and discard any higher bits

- In other words: Overflows act the same as they do in C-language integer arithmetic
- In all of Hydra's other types, out-of-range numbers instead saturate to fit in the K -bit range
- This distinction matters only in *dest* operands: In *src* operands, UTRUNC32 == UINT32, etc.

Data types: Truncating signed integers

Symbol	Data values
HYDRA_TRUNC32	32-bit truncating signed integers
HYDRA_TRUNC16	16-bit truncating signed integers
HYDRA_TRUNC8	8-bit truncating signed integers

Hydra 4.0–4.1 did not include signed counterparts of the `HYDRA_UTRUNCnn` types

- The result is the same if you write to a signed or unsigned truncating *dest*

But there are some secondary benefits to including signed-TRUNC types in the system

- E.g., the setting of the Overflow status flag depends on the signed-ness
- And it is convenient to be able to use the same truncating-typed vector as both *src* and *dest*
- Therefore, Hydra 4.2 introduces “signed” versions of the truncating integer types
- When building for Hydra 4.0–4.1, “`HYDRA_TRUNCnn`” is treated as a synonym for “`_UTRUNCnn`”

Data types: Fixed-point numbers

Symbol	Data values
HYDRA_FIX32(M)	32-bit signed fixed-point with $M = 0 \dots 32$ fraction bits (and $32 - M$ integer+sign bits)
HYDRA_FIX16(M)	16-bit signed fixed-point with $M = 0 \dots 16$
HYDRA_FIX8(M)	8-bit signed fixed-point with $M = 0 \dots 8$
HYDRA_UFIX32(M)	32-bit unsigned fixed-point with $M = 0 \dots 32$
HYDRA_UFIX16(M)	16-bit unsigned fixed-point with $M = 0 \dots 16$
HYDRA_UFIX8(M)	8-bit unsigned fixed-point with $M = 0 \dots 8$

Fixed-point numbers are like saturating integers scaled by 2^{-M}

- An algorithm can choose M based on the data's known range and accuracy
- M effectively puts the "decimal point" at some position within the digits
- Known as "*fixed point*" (vs. floating point) because M is constant across all the vector elements

Data types: Floating-point numbers

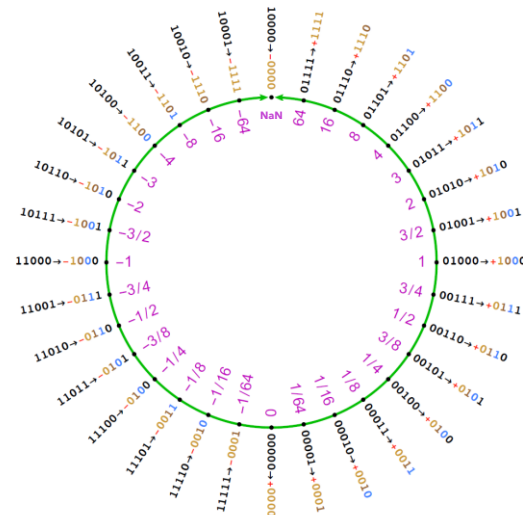
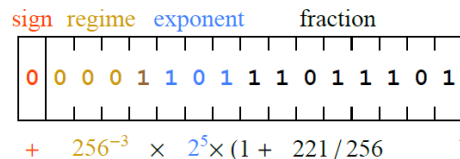
Symbol	Data values
HYDRA_FLOAT32	32-bit (fully IEEE-compatible single precision) floating-point numbers
HYDRA_FLOAT16	16-bit (IEEE half-precision) floats
HYDRA_BFLOAT16	16-bit "BFLOAT"-style floats
HYDRA_FLOAT	Synonym for HYDRA_FLOAT32
HYDRA_HFLOAT	Synonym for HYDRA_FLOAT16
HYDRA_BFLOAT	Synonym for HYDRA_BFLOAT16

Data types: Posits

Symbol	Data values
HYDRA_POSIT32(M)	32-bit Posits with $M = 0 \dots 3$ exponent bits
HYDRA_POSIT16(M)	16-bit (or smaller) Posits
HYDRA_POSIT8(M)	8-bit (or smaller) Posits

Hydra 4 also supports an alternative floating-point format called "Posits"

- https://posithub.org/docs/posit_standard-2.pdf
- <https://posithub.org/docs/Posits4.pdf>
- When used with Hydra's packed addressing mode, Posits can pack floating-point numbers into any number (1-32) of bits with a good and flexible trade between precision and dynamic range



SIMD operation

In computer architecture, **SIMD** (Single Instruction, Multiple Data parallelism) means splitting the normal data path (e.g., 32 bits wide for Hydra 4) into multiple lanes to process several small data items at once (e.g., 4 bytes or 2 short words per cycle)

Hydra 4.2.0 (non-“Junior”) includes a limited SIMD capability:

- VMAC / VSUM / VMM will operate on 8-bit integers at 4x speed under suitable conditions
- ... i.e., when the vectors or matrix rows are a multiple of 4 bytes long, 4-byte-aligned in RAM, not vector-masked, etc.

With SIMD-mode VMM, Hydra 4.2 at 400 MHz (e.g., as found on the DIG2409B0 chip) can evaluate fully-connected neural networks at a speed of over 3 GOPS (3 billion add-or-multiply operations per second)

- This is more than 10x faster than firmware on a RISC-V E76 can achieve
- More than 20x faster than Chimera firmware

Future versions of Hydra will add SIMD treatment for other instructions and data types

Hydra 4

Architecture overview

- Introduction
- **Using Hydra**
 - Examples
 - Hydra registers
 - Vector instructions
 - Data types
 - **Addressing modes**
 - Stored program unit
 - Documentation and tools
 - Hydra 4.2 functionality
 - Performance
- Hardware and implementation

Addressing modes

Each source or destination vector has an addressing mode

The addressing mode defines where each i 'th vector element appears in RAM relative to the vector's base address

Most Hydra 4 instructions accept any mix of modes in their inputs and result

Some things that were special instructions in Hydra 3 (e.g., VPACK, VMT) can be done as simply VMOV's with different *src* and *dest* modes

Addressing modes: Basic vectors and scalars

Notation	Addressed as
hydra_ vector (<i>base, type</i>)	Linear vector in consecutive RAM locations
hydra_ strided (<i>base, type, stride</i>)	Strided vector: $base + i * stride$
hydra_ ostride (<i>base, type, offset, stride</i>)	Strided with offset: $base + offset + i * stride$
hydra_ mscalar (<i>base, type</i>)	Scalar number replicated across the vector (the number comes from RAM)
hydra_ scalar_int (<i>x</i>)	Scalar INT32 number <i>x</i> encoded directly (passed using Hydra's G0 or G1 register) (The constants $x = 0$ and $x = 1$ can be encoded without using any G_n register)
hydra_ scalar_float (<i>x</i>)	Scalar FLOAT32 number <i>x</i> encoded directly (passed using Hydra's G0 or G1 register)
hydra_ rscalar (<i>reg, type</i>) hydra_ rscalar_int/float (<i>reg, type, x</i>)	Scalar number of any type stored in any Hydra register

Addressing modes: Special modes

Notation	Addressed as
hydra_ neg_vector (<i>base, type</i>)	A negated view of a linear source vector: The values are negated (as if by VNEG) as they come from RAM into Hydra's arithmetic unit
hydra_ abs_vector (<i>base, type</i>)	A linear source vector whose values are absolute-valued (as if by VABS) as they come
hydra_ half_vector (<i>base, type</i>)	A multiplied-by- $\frac{1}{2}$ view of a vector in RAM
hydra_ double_vector (<i>base, type</i>)	A multiplied-by-2 view of a vector in RAM
hydra_ circular (<i>base, type, mod</i>)	Circular addressing: $base + (i \% mod)$
hydra_ offset_circ (<i>base, type, mod, i0</i>)	Offset-circular addressing: $base + (i + i_0) \% mod$
hydra_ nested (<i>base, type, args</i>)	A double- or triple-nested repeating pattern of memory accesses, like a combination of circular and repeated-column modes

Addressing modes: Matrix modes

Notation	Addressed as
hydra_ matrix (<i>base, type, C</i>)	An $R \times C$ matrix of width C , height $R = VL/C$, row stride = C , and column stride = 1
hydra_ tmatrix (<i>base, type, R</i>)	An $R \times C$ matrix which is the transposed view of a $C \times R$ matrix in the RAM whose width is R , $C = VL/R$, row stride = R , column stride = 1
hydra_ gmatrix (<i>base, type, C, rstride, cstride</i>)	A matrix with general R, C , row stride, and column stride (the strides are passed in Hydra's G0 or G1 register)
hydra_ rrmatrix (<i>base, type, C</i>)	An $R \times C$ "repeated row" matrix where C = width, $R = VL/C$, row stride = 0, column stride = 1 (actually same as "circular addressing")
hydra_ rcmatrix (<i>base, type, C</i>)	An $R \times C$ "repeated column" matrix where C = width, $R = VL/C$, row stride = 1, column stride = 0

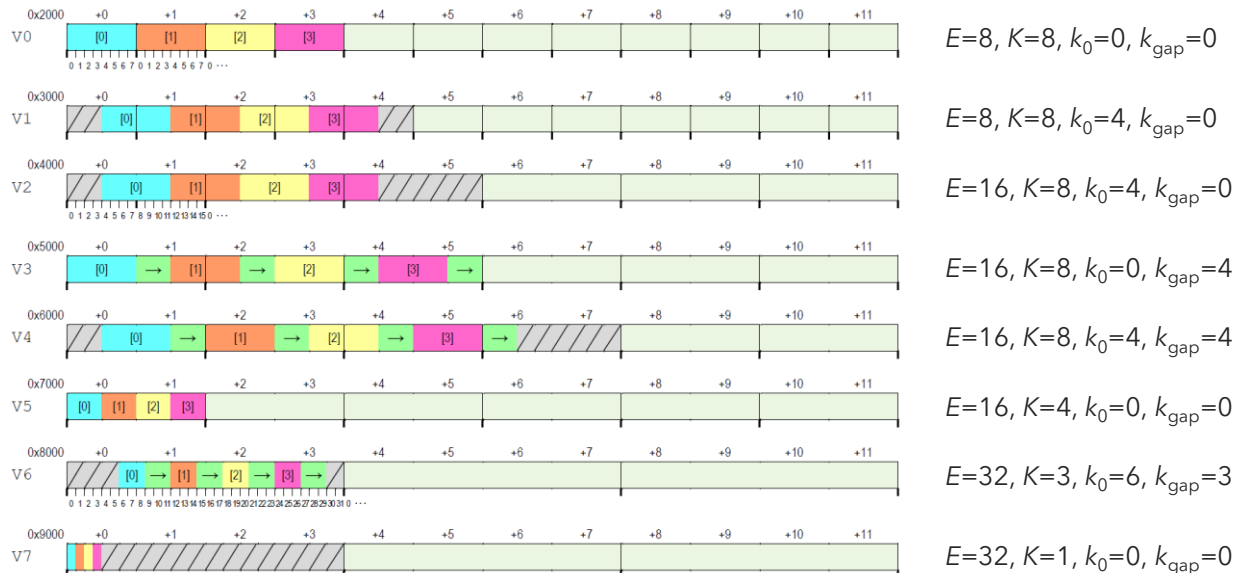
Addressing modes: FFT-related modes

Notation	Addressed as
hydra_ bfly (<i>base, type, args</i>)	FFT butterfly addressing
hydra_ bitrev (<i>base, type, args</i>)	FFT bit-reversed addressing
hydra_ cosine (<i>base, type, args</i>)	Cosine function from cosine table
hydra_ sine (<i>base, type, args</i>)	Sine function from the same cosine table
hydra_ complex (<i>base, type, args</i>)	Complex numbers rotated on the complex plane by a multiple of 90° and/or conjugated, where <i>modes</i> encodes any combination of swapping even-with-odd elements, negating even elements, negating odd elements, and/or absolute-valuing, halving, or doubling all elements

"args" is a bit mask of mode-related options such as HYDRA_BITREV_STRIDE(*stride*) or HYDRA_COMPLEX_SWAP + HYDRA_COMPLEX_ABS

Addressing modes: Packed mode

Notation	Addressed as
<code>hydra_packed(base, type, K_args)</code>	K-bit signed or unsigned integers, fixed-point numbers, or Posit-style floats packed into long words, short words, or bytes, with initial bit offset k_0 and with k_{gap} bits between items



Addressing modes: Other modes

Notation	Addressed as
hydra_ mask_gen (<i>mask</i>)	In the <i>dest</i> operand position, generate a vector mask showing which results are non-zero (the results themselves are discarded)
hydra_ mask_flag (<i>mask</i>)	Generate a vector mask recording a given status flag for each vector element
hydra_ mask2_gen (<i>base, type, mask, args</i>) hydra_ mask2_flag (<i>base, type, mask, args</i>)	Save the numeric results in a linear vector, and also generate a vector mask from a status flag
hydra_ null_vector (<i>type</i>) hydra_ null_scalar (<i>type</i>)	Null data source or sink of the indicated type; delivers 0's as <i>src</i> ; discards data as <i>dest</i>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
NNEGATIVE	NZERO	NINFINITE	NDIVZERO	NINEXACT	NINVALID	NUNDERFLOW	NOVERFLOW	NEGATIVE	ZERO	INFINITE	DIVZERO	INEXACT	INVALID	UNDERFLOW	OVERFLOW

Addressing modes: Pattern mode

Notation	Generated as
<code>hydra_pattern(a, b)</code>	The integer arithmetic sequence $a + (b*i)$
<code>hydra_rrpattern(a, mod)</code>	The integer sequence $a + (i \% mod)$
<code>hydra_rcpattern(a, div)</code>	The integer sequence $a + \text{floor}(i / div)$
<code>hydra_gpattern(a, C, rstride, cstride)</code>	An integer sequence based on the general matrix addressing pattern
<code>hydra_patmode(mode, esize, base, args...)</code> (encoded as any mode with <code>type = PATTERN</code>)	Integer sequence according to the mode

Hydra's pattern mode generates a pattern of integers based on the sequence of addresses produced by any chosen addressing mode

- It bypasses the RAM and uses the address sequence *itself* as the source data
- It can generate: Arithmetic sequences, sawtooths, stairsteps, triangle waves, bit-reversal, ...

Hydra 4

Architecture overview

- Introduction
- **Using Hydra**
 - Examples
 - Hydra registers
 - Vector instructions
 - Data types
 - Addressing modes
 - **Stored program unit**
 - Documentation and tools
 - Hydra 4.0 functionality
 - Performance
- Hardware and implementation

Stored program unit

Hydra 3 introduced an SPU for autonomous operation with no firmware handholding

This especially helps non-Chimera CPUs with their clumsy interrupts, etc.

But Hydra 3's SPU was so "strange" and intricate that people didn't use it!

Hydra 4 has a redesigned SPU

A simple SPU program is just a list of Hydra vector instructions ending with a VSTOP (opcode 0x0)

<i>Instruction</i>	<i>Opcode</i>	<i>Description</i>
<i>Vector ops</i>	<code>'00jj rrss mmff 0000 0000 bbbb aaaa dddd'</code>	Wait until Hydra is READY , then write the indicated 30-bit opcode to the OPCODE register. The SPU can invoke any Hydra opcode from sections 4.1–4.3. Once the opcode has been posted to the instruction buffer, the SPU proceeds immediately to the next SPU instruction while (or even before) Hydra begins processing the posted vector instruction.
VSTOP	<code>'0000 0000 0000 0000 0000 0000 0000 0000'</code>	Wait until Hydra is IDLE and then end (or pause) the SPU program. As viewed by firmware, the IDLE and READY flags will stay low for as long as the SPU program runs, then both flags will go high to indicate that the program has ended.



Stored program unit

Hydra 3 introduced an SPU for autonomous operation with no firmware handholding

This especially helps non-Chimera CPUs with their clumsy interrupts, etc.

But Hydra 3's SPU was so "strange" and intricate that people didn't use it

Hydra 4 has a redesigned SPU

A simple SPU program is just a list of Hydra vector instructions ending with a VSTOP (opcode 0x0)

More complex SPU programs can do lots of things ...

Instruction	Opcode	Description
Vector ops	'00jj rrss mmff 0000 0000 bbbb aaaa dddd'	Wait until Hydra is READY, then write the indicated 30-bit opcode to the OPCODE register. The SPU can invoke any Hydra opcode from sections 4.1–4.3. Once the opcode has been posted to the instruction buffer, the SPU proceeds immediately to the next SPU instruction while (or even before) Hydra begins processing the posted vector instruction.
VSTOP	'0000 0000 0000 0000 0000 0000 0000 0000'	Wait until Hydra is IDLE and then end (or pause) the SPU program. As viewed by firmware, the IDLE and READY flags will stay low for as long as the SPU program runs, then both flags will go high to indicate that the program has ended.
VSTOPR	'0000 0000 0000 0000 0000 0000 0000 0001'	Like VSTOP, but waits only until Hydra is READY before stopping. Firmware can tell when the SPU program has ended by watching for READY to go high. When this instruction is used, there may still be one Hydra vector instruction in progress in its pipeline when the SPU program ends.
VWAIT	'0000 0000 0000 0000 0000 0000 0000 0010'	Wait until Hydra is IDLE before proceeding in the SPU program.
VWAITR	'0000 0000 0000 0000 0000 0000 0000 0011'	Wait until Hydra is READY before proceeding in the SPU program.
VSTART	'0000 0000 0000 0000 0000 0000 0000 1000'	Start an SPU program. The VSTART opcode is written to the OPCODE register and processed through the main instruction buffer; this causes the SPU to begin processing SPU instructions at the address indicated by the VPC register, and the instruction buffer remains full as viewed by host firmware. Once started, the SPU continues operating until it encounters a VSTOP, a VSTOPR, or an error. If this instruction appears in an SPU program, it is quietly ignored.
VCALL	'0000 0000 0000 0000 0000 1nnn nnnn nnnn'	When written to the OPCODE register, this instruction starts an SPU program at address V7ADDR + (4 * n), as if by writing this start address into VPC and issuing VSTART. It stores -1 in V7CFG, which will cause a subsequent VRET to stop the SPU.

Stored program unit (continued...)

The SPU can do more than simply issue a fixed sequence of vector opcodes

The SPU has its own instruction set for setting up vector operands and other register values

The SPU can operate on any Hydra registers while the rest of Hydra is busy with a vector operation (and the rest of the host CPU system may be busy doing unrelated work)

Instruction	Opcode	Description
VLOADV	<code>'01tt tnnn aaaa aaaa aaaa aaaa aaaa aaaa'</code>	Load the $VnADDR$ register with the unsigned 24-bit constant A , and load $VnCFG$ with one of eight common type-and-mode combinations T : $T = '000'$: Memory scalar of a single-precision float. $T = '001'$: Memory scalar of a signed long integer. $T = '010'$: Linear vector of single-precision floats. $T = '011'$: Linear vector of signed long integers. $T = '100'$: Linear vector of signed short integers. $T = '101'$: Linear vector of signed byte integers. $T = '110'$: Linear vector of truncating unsigned long integers. $T = '111'$: Linear vector of unsigned byte integers.
VMOVRI	<code>'100r rrrr aaaa aaaa aaaa aaaa aaaa aaaa'</code>	Load Hydra register R with the sign-extended 24-bit constant A .
VLOADVH	<code>'111a annn 00aa aaaa gggg mmmmm tttt tttt'</code>	Load the $VnADDR$ register with $(A \ll 24)$, and load $VnCFG$ with zero-extended arg G , mode M , and type T . VLOADVH is typically paired with VADDRI to generate a full 32-bit address in $VnADDR$. With VADDR2IH, it can also generate any 32-bit $VnCFG$ value. (But note that on chips where addresses are small, $W \leq 24$, it is simpler to use two VMOVRI instructions to load $VnADDR$ and $VnCFG$.)
VADDR2I	<code>'111r rrrr 010d dddd aaaa aaaa aaaa aaaa'</code>	Set Hydra register D to the contents of Hydra register R plus the signed 16-bit constant A . With $A = 0$, this is a register-to-register move. (Hydra headers and tools accept VMOVRI as a synonym for VADDR2I with $A = 0$.)
VADDR2IH	<code>'111r rrrr 011d dddd aaaa aaaa aaaa aaaa'</code>	Set Hydra register D to Hydra register R plus the constant $(A \ll 16)$. Together with VADDRI, this allows any 32-bit constant to be loaded or added into R . If $R = 0$, this loads a left-shifted constant into D ; this allows a single SPU instruction to generate many common floating-point constants in a Gn register.

Stored program unit (continued...)

More generally, SPU opcodes are instructions like those of a simple microcontroller

SPU programs can have their own conditional tests, loops, subroutines, variables, and so on

If the algorithm does not use all eight vector registers V0–V7, the spare VnCFG's and VnADDR's can serve as extra working registers for the SPU program

Instruction	Opcode	Description
VADDRI	'101r rrrr aaaa aaaa aaaa aaaa aaaa aaaa'	Increment Hydra register <i>R</i> by the signed 24-bit constant <i>A</i> .
VCMPRI	'110r rrrr aaaa aaaa aaaa aaaa aaaa aaaa'	Compare Hydra register <i>R</i> (as a signed 32-bit integer) with the signed 24-bit constant <i>A</i> , setting an internal SPU_STATUS register with several bits indicating comparison conditions (see below and also section 4.3.1). For example, this sets bit 2 of SPU_STATUS to '1' if $R \leq A$, analogous to VCMPI with cccc = 2.
VMOVRM	'111r rrrr 100d dddd aaaa aaaa aaaa aaaa'	Read a 32-bit word from the RAM location at address $(R + A)$ and store the word in Hydra register <i>D</i> . <i>A</i> is a signed 16-bit offset in units of bytes. The low two bits of $(R + A)$ are ignored, effectively rounding the byte address down to a multiple of 4. With $R = 0$, this interprets <i>A</i> as an absolute RAM address. With $R = VPC$, this accesses a storage location adjacent to the SPU program itself. Together with VADDR2IH, this instruction can generate any 32-bit RAM address.
VMOVMR	'111r rrrr 101d dddd aaaa aaaa aaaa aaaa'	Write the contents of Hydra register <i>D</i> to the 32-bit RAM location at address $(R + A)$. See the notes for VMOVRM.
VBRA	'111r rrrr 110c cccc aaaa aaaa aaaa aaaa'	Jump by adding the signed 16-bit constant $(A + 4)$ to VPC if the condition is true, else proceed by incrementing VPC by 4 as usual. The offset <i>A</i> must be a multiple of 4 bytes; $A = 0$ will jump to the immediately next instruction. The instruction reads Hydra register <i>R</i> and tests a condition given by <i>C</i> . Register <i>R</i> is often STATUS or SPU_STATUS but it may be any register. Conditions include: If $C = '0cccc'$, true if bit <i>C</i> of <i>R</i> is set; if $C = '10pnc'$, true if <i>z</i> is set and $R = 0$ (as a signed integer) or if <i>p</i> is set and $R > 0$ or if <i>n</i> is set and $R < 0$; conditions $C = '11xxx'$ are reserved. If $C = '10111'$, the branch is unconditional because the condition is always true. Note that any SPU instruction serves as a branch if it writes to VPC: VADDRI to VPC is a long-range unconditional relative branch; VMOVRI to VPC is an unconditional absolute branch; VMOVRM to VPC is an indirect jump to a program address stored in RAM.

Stored program unit (continued...)

But note that the SPU is *not meant to be a fast microprocessor!*

- The SPU uses a very simple sequencer that takes several cycles for most SPU instructions
- For example, the SPU has just one port for reading or writing Hydra registers, so VADDR3 takes at least three cycles to read registers "R" and "S" and to write to register "D"
- The SPU's job is to be *cheap and simple*, while just fast enough to set up Hydra vector operations efficiently

Instruction	Opcode	Description
VADDR3	'111r rrrr 111d dddd xxxx ssss 00kk kkkk'	Set Hydra register <i>D</i> to Hydra register <i>R</i> , shifted left by <i>k</i> bits or shifted right (signed) by $-k$ bits, plus Hydra register <i>S</i> . All shift amounts $-32 \leq k \leq 31$ are allowed. With $S = 0$ this is a pure shift; with $k = 0$, this is a pure addition; with both, this is a move. Currently bits <i>xxx</i> must be '000'. If a future version of Hydra outgrows the space of 32 registers, these bits will extend <i>R</i> , <i>D</i> , and <i>S</i> by one bit each so that instructions in this format can reach up to 64 registers.
VSUBR3	'111r rrrr 111d dddd xxxx ssss 01kk kkkk'	Set Hydra register <i>D</i> to Hydra register <i>S</i> minus (shifted) Hydra register <i>R</i> . If $S = 0$ and $k = 0$, this is negation.
VCMPR2	'111r rrrr 111d 0000 xxxx ssss 10kk kkkk'	Compare Hydra register <i>S</i> with (shifted) Hydra register <i>R</i> , storing status in the SPU_STATUS register (e.g., set bit 2 to '1' if $S \leq$ shifted <i>R</i>). This opcode is invalid / reserved if bits 20:16 are not all '0'.
VMOVR3	'111r rrrr 111d dddd xxxx ssss 1100 0000'	Exchange Hydra registers: $D \leftarrow R$ and $R \leftarrow S$ for any three registers. If <i>D</i> and <i>S</i> are the same register, this is a simple exchange $D \leftrightarrow R$. This is a three-cycle instruction that first reads and saves <i>R</i> , then reads <i>S</i> and writes <i>R</i> . This opcode is invalid / reserved if <i>D</i> and <i>R</i> are the same register or if <i>D</i> selects the VPC register. (However, <i>R</i> and <i>S</i> may be VPC; $R = \text{VPC}$ forms a useful "call subroutine" instruction.)
VANDR3	'111r rrrr 111d dddd xxxx ssss 1100 0010'	Set Hydra register <i>D</i> to Hydra register <i>S</i> bitwise ANDed with Hydra register <i>R</i> .
VANDNR3	'111r rrrr 111d dddd xxxx ssss 1100 0011'	Set Hydra register <i>D</i> to Hydra register <i>S</i> bitwise ANDed with the complement of Hydra register <i>R</i> .

Hydra 4

Architecture overview

- Introduction
- **Using Hydra**
 - Examples
 - Hydra registers
 - Vector instructions
 - Data types
 - Addressing modes
 - Stored program unit
 - **Documentation and tools**
 - Hydra 4.2 functionality
 - Performance
- Hardware and implementation

Hydra 4 User's Guide

The *Hydra 4 User's Guide* describes how to use Hydra 4

- How to program it
- Detailed descriptions of every instruction, addressing mode, register, and feature
- Algorithms such as `sqrt()`, `exp()`, FFTs, merge sorts, median filters, neural nets, etc.
- How to integrate it into a chip's hardware

A public version will be available soon on github.com/synaptics-hydra

Synaptics Confidential
Internal Use Only
Do Not Distribute

Hydra 4 User's Guide

For the Hydra 4.0.0–4.2.0 coprocessor

PN: 511-000xxx-01 Rev 0.25 (January 5, 2026)

Synaptics Confidential — Internal Use Only

Hydra 4 tools

No special tools are required to program Hydra 4

- The hydra4_defs.h header file defines macros such as hydra_vadd_w() for invoking Hydra
- It is self-contained: No run-time library is needed; no special compiler or RTOS support
- It works with the host CPU's standard C or C++ compiler and debugger

This function is complete;
nothing else needed ⇒

```
#include "hydra4_defs.h"
float scalarProduct(int len, float *veca, float *vecb)
{
    float result;

    HYDRA_INITIALIZE();    // Normally this is called just once at startup.
    // Set result = sum(veca[i] * vecb[i] for i=0...len-1).
    hydra_vmac_w(hydra_set_vl(len),
                 hydra_mscalar(&result, HYDRA_FLOAT),
                 hydra_vector(veca, HYDRA_FLOAT),
                 hydra_vector(vecb, HYDRA_FLOAT));

    return result;
}
```

A true Hydra “compiler”?

Wouldn't it be great if we could simply write,

```
float scalarProduct(int len, float *veca, float *vecb)
{
    float sum = 0.0;
    for (int i = 0; i < len; i++)
        sum += (veca[i] * vecb[i]);
    return sum;
}
```

... and this would compile to a Hydra operation, if the target chip includes Hydra?

Vectorizing compilers do exist

- The Cray-1 had a vectorizing FORTRAN compiler
- OpenCL is a good modern example, or automatic vectorization of ordinary C loops

Could we port OpenCL to Hydra? Yes, if we had an army of compiler people ...

- In other words, not for us, not right now anyway
- (Actually, for a RISC-V compiler it would not be *that* hard: Modern GCC can support vectorization)

Porting legacy Hydra 3 firmware to Hydra 4

Synaptics touch and fingerprint firmware includes lots of code written for Hydra 3

- Most older code uses “HOCA_xxx” constants to painfully construct the Hydra 3 opcode
- Some newer code uses “Hydra assembler” macros to express the Hydra 3 opcode symbolically

Hydra 4's *compatibility mode* makes it easy to port either style of Hydra 3 firmware to run on Hydra 4

```
float scalarProduct(int len, float *veca, float *vecb)
{
    if (numElements == 0) return 0;
    if_hydra_inst_fifo_full_chimera_goto_sleep();
    HYDRA.HYDRA_VL.HYDRA_VL = (uint16)len;
    HYDRA.HYDRA_S0.HYDRA_S0 = (uint16)2;
    HYDRA.HYDRA_INST_SRC2.HYDRA_ISRC2 = (uint16)vecb;
    HYDRA.HYDRA_INST_SRC1.HYDRA_ISRC1 = (uint16)veca;
    HYDRA.HYDRA_INST_CODE.HYDRA_ICODE = HOCA_11_15_FMACMM |
        HOCA_10_SRC2_VECTOR |
        HOCA_9_SRC2_ADDRVEC_DIR | HOCA_7_8_SRC2VEC_STRIDE_S00 |
        HOCA_6_SRC1_ADDRVEC_DIR | HOCA_4_5_SRC1VEC_STRIDE_S00 |
        HOCA_1_3_MAC_REG_ADDR |
        HOCA_0_VECLEN_VXX0;
    if_hydra_inst_fifo_notempty_chimera_goto_sleep();
    return *(volatile float*)&(HYDRA.HYDRA_G0.v);
}
```

```
float scalarProduct(int len, float *veca, float *vecb)
{
    HYDRA_WW_OP( VFMAC G10, VL(len) V(veca), V(vecb) );
    return HYDRA_GET_FLOAT_G10();
}
```



Functions like these, originally written for Hydra 3, can compile and run as-is in Hydra 4's compatibility mode

Compatibility mode

If you write a Hydra 3-style 16-bit opcode to Hydra 4's COMPAT_OPCODE register ...

- Hydra's instruction decoder will [attempt to] translate it into a modern Hydra 4 operation
- ... using a defined mapping from Hydra 3 registers onto Hydra 4's register set

Simply use the `hydra4_compat.h` header file in place of Hydra 3's `hydra_base.h` or `hydra_registers_0x0310.h`

- It defines the same Hydra 3 symbols, fields, types, and names as those old headers, all redefined appropriately for Hydra 4's compatibility mode
- Most legacy source code that uses the Hydra 3 headers can build unchanged with `hydra4_compat.h`
- If desired, a single firmware source routine can build for either Hydra 3 or Hydra 4 depending on the target chip

<i>Hydra 3 register</i>	<i>... maps to</i>	<i>Notes</i>
INST_CODE	COMPAT_OPCODE	
INST_DEST	V3ADDR	(See notes below for treatment of addresses.)
INST_SRC1	V4ADDR	
INST_SRC2	V5ADDR	
INST_SRC2X	V6ADDR	
S0	V3CFG bits 15:0	
S1	V4CFG bits 15:0	
S2	V5CFG bits 15:0	
CFG	V6CFG bits 15:0	
MCFG	V6CFG bits 31:16	

<i>Hydra 3 instruction</i>	<i>... maps to</i>	<i>Notes</i>
V*MUL	VMUL	Supported.
V*Ncond	(VNEXT)	Compatibility mode does not treat this.
VNOP	VNOP	Supported.
VOR	VOR	Supported.
VPACK	VMOV	Supported.
V*Pcond	See section 4.3.7	Compatibility mode does not treat this.
VRESET	VNOP	Ignored (but does not cause a reset).
V*RSUM	(VCMAC)	Compatibility mode does not treat this.
V*Scond	VCMP	Supported, except V*SBT.

HydraSim simulator

HydraSim is a software simulator that models all versions of Hydra

- It now covers Hydra 4.0-4.2 as well as the older versions
- HydraSim gives bit-accurate answers for all Hydra instructions, plus timing estimates
- Developed by Adam Greenblatt (thanks, Adam!)

Chimera's ChimSim simulator has HydraSim built-in

- ChimSim invokes it automatically when Chimera firmware uses Hydra

HydraSim is a library that can integrate into other simulators, as needed

- If we use a RISC-V instruction-level simulator to develop firmware for chips like Byrd ...
- ... we can modify that simulator to call HydraSim, just as ChimSim does

HydraSim can be integrated into any C/C++ program

Hosted simulation with HydraSim

HydraSim also allows “hosted” development and testing of firmware that uses Hydra:

- Compile the firmware as a native x86 program under Windows or Linux
- Use `-DHYDRA_HOSTED_SIM` on the compilation command line
- Link with `hydra4_hosted_hydrasim.c` and `hydrasim.a`

Firmware algorithms can now be tested and debugged with all the tools and facilities of a large computer:

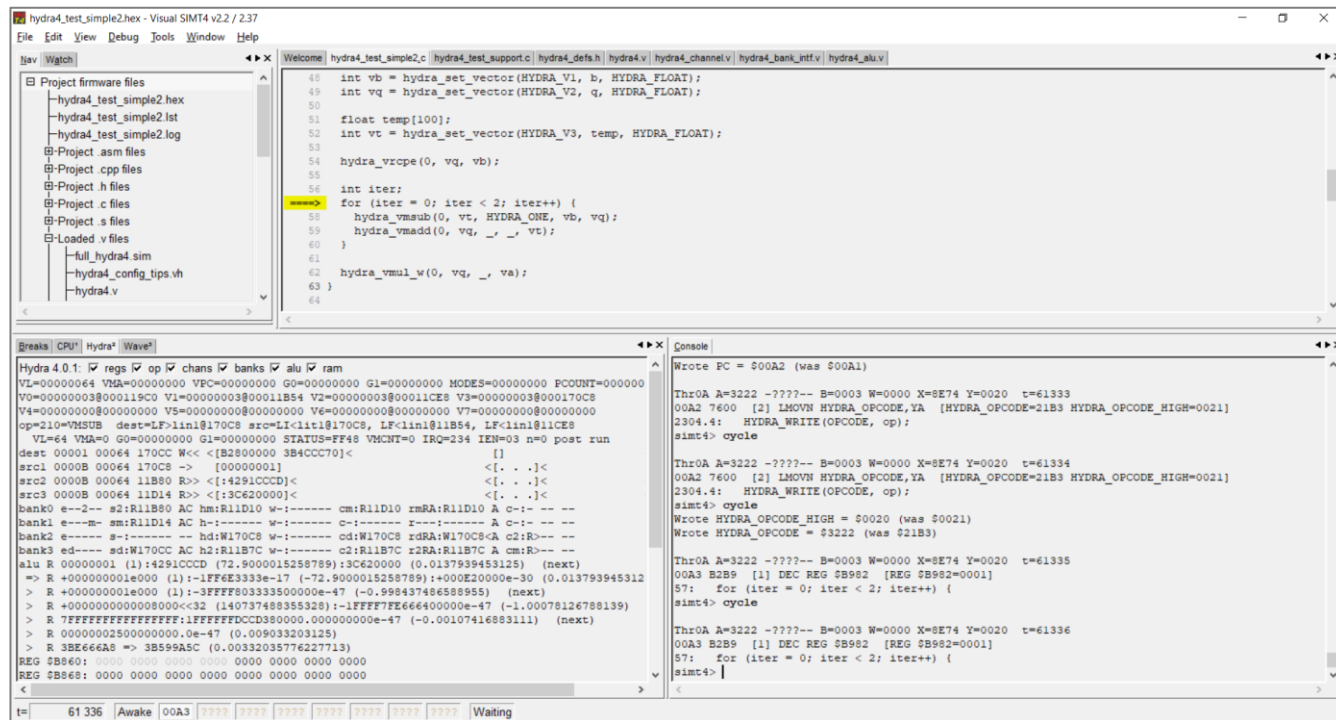
- Debug with GDB or Eclipse
- Use features that would be impractical in real firmware:
double-precision floats, `printf()`, file I/O, huge data arrays, etc.

In hosted mode, every Hydra macro such as `hydra_vadd_w(...)`, `hydra_vector(...)`, etc., is automatically directed to the HydraSim simulator in lieu of using real Hydra hardware

SIMT4 simulator

SIMT4 is a detailed simulator for Chimera chips

- It has a cycle-accurate simulation model of the Chimera CPU, and it models a chip's peripherals, including Hydra, with the peripheral's exact Verilog source code
- Its VSIMT4 graphical interface can debug integrated systems of firmware + hardware
- When firmware uses Hydra 4, the Hydra Viewer shows Hydra's instruction buffer, arithmetic pipeline, and RAM channels and bank interfaces in detail



HydraMath library

HydraMath is a relatively simple software library that exactly emulates the numeric results of most Hydra instructions

- ... but not all the addressing modes, registers, and other Hydra features
- HydraMath also can convert numbers to or from any of Hydra's supported data formats
- It is lightweight and easier to use than HydraSim in scripting environments, etc.
- HydraSim internally uses HydraMath to model Hydra's arithmetic precisely

```
#include "hydra4_math.h"
hydramath_uint32_t result[N], in1[N], in2[N];
for (int i = 0; i < N; i++) {
    int status = hydramath_emul(NULL,           // Optional "state" pointer to allow reentrant use.
                                0x00007000,     // VSUB opcode.
                                &result[i], 0x20, // Destination pointer and dest CFG.
                                in1[i], 0x20,    // src1 data and src1 CFG.
                                in2[i], 0x20,    // src2 data and src2 CFG.
                                0, 0,            // src3 is not used in VSUB.
                                0, 0,            // G0 and G1 if the instr needs them.
                                0);              // Various optional flag bits.

    if (status >= 0) {
        //Now result[i] contains the i'th result of VSUB, and status is the 8 status bits.
    } else {
        //An error occurred such as an invalid opcode or invalid CFG word.
    }
}
```

Hydra 4

Architecture overview

- Introduction
- **Using Hydra**
 - Examples
 - Hydra registers
 - Vector instructions
 - Data types
 - Addressing modes
 - Stored program unit
 - Documentation and tools
 - **Hydra 4.2 functionality**
 - Performance
- Hardware and implementation

Implemented so far...

The full *Hydra 4 User's Guide* describes lots of functionality

There wasn't time to implement all of it for Byrd's and SmartBridge's Hydra 4.0

- But it certainly includes all functionality we currently use in our touch firmware with Hydra 3
- And it includes FFT support (needed for Byrd)
- And cryptographic support (may be needed for SmartBridge or Byrd)

In fact, most features in the *Hydra 4 User's Guide* have been implemented

Not yet implemented, omitted from Byrd

The following features from the full *User's Guide* didn't make it into SmartBridge / Byrd, but they can be added for future chips (especially if people ask for them):

- General division (VDIV)
- Fast median filters (VMED, VMEDR)
- Optimization of sparse vector masks
- Instruction "joining" optimization
- Faster "SIMD" treatment of 8- and 16-bit integer types (limited support appears in Hydra 4.2)
- Automatic edge "padding" in convolutions
- Potential additional features for neural nets, cryptography, special functions, etc., beyond those described here

Hydra 4

Architecture overview

- Introduction
- **Using Hydra**
 - Examples
 - Hydra registers
 - Vector instructions
 - Data types
 - Addressing modes
 - Stored program unit
 - Documentation and tools
 - Hydra 4.2 functionality
 - **Performance**
- Hardware and implementation

Performance

Various projects have achieved dramatic speed-ups by applying Hydra 3 to our touch and fingerprint algorithms

- ... even though Hydra's original inspiration, "large PC touch screens," never took off!

Some examples:

- Gian Mario Manca & Vladan Petrovic dramatically sped up OLED noise removal for touch screens in challenging noise environments
- The quantum fingerprint matcher's image processing module runs 6x faster using Hydra 3 than as pure firmware on Prometheus' Xtensa processor (thanks again to Vladan)
- Eric Ding's Hydra 3 implementation of image column-wise standard deviations runs 10x faster than as pure Chimera firmware
- Karthik's AI-based ACM algorithm runs 3.5x faster with Hydra 3 than as Chimera firmware
- Convolutional neural networks (CNNs) run over 10x faster with Hydra 3
- (And remember that Hydra 4 is 2x faster than Hydra 3 for many operations)

Performance

Hydra 4 significantly outperforms SiFive's E76 RISC-V processor ...

- ... even though the E76 is a full dual-issue superscalar, deeply pipelined processor
- ... whose design was specifically tuned for high computational performance

Examples:

- The Byrd team reports that FFTs are ~4x faster on Hydra 4 than in E76 firmware
- Hydra 4 performs floating-point matrix multiplication 2x faster than Elkin Garcia's firmware code, which was aggressively optimized to best fit the E76's pipeline
 - (Even though the E76, and the RISC architecture itself, has been optimized throughout its history with matrix multiplication as a key standard benchmark)
- Hydra 4.0 can accelerate byte-weighted neural network evaluation by nearly ~4x;
Hydra 4.2 can accelerate the same by over 10x
- Hydra 4's VSHA instructions are ~2x faster than the theoretically optimal assembly language E76 implementation of SHA256 (and faster still compared to practical firmware-friendly E76 code)

Hydra is as fast as a supercomputer ...

... if you don't mind comparing to supercomputers that museums now use as furniture

Hydra 3 on a 116 MHz Stockholm or Rio chip can do **232 million** sustained floating-point operations per second (1 add + 1 multiply on every cycle)

- Or 464 million 16-bit fixed-point ops

From a [Cray-1](#) brochure at the Computer History Museum in Mountain View, California ⇒

Hydra 4 at Byrd's 400 MHz can sustain **800 million** flops

- About as fast as Cray's next machine, the quad-processor Cray X-MP

The possibilities for optimization are further enhanced by SCILIB, a comprehensive library of commonly used mathematical routines. The library currently includes fast Fourier transforms, matrix and linear algebra packages including the widely used LINPACK and EISPACK collections, searches and sorts, and other FORTRAN callable subroutines. Many of these hand-coded routines use the pipeline/chaining properties of the CRAY-1 hardware. These routines are fast—often executing at over 140 million floating-point operations per second—and provide easy access to the full vector power of the CRAY-1.



Is that progress?

If the Cray-1 back in 1976 was roughly as fast as Hydra today, have we really made progress?

Hydra fits in a small portion of a small chip in a corner of your laptop or phone

Less than 0.1 mm² in Byrd's process

It draws a few milliwatts at most

Compared to the Cray ...

The primary power system consists of a 150 KW motor generator, motor-generator control cabinet, and power distribution cabinet. The motor generator supplies 208 V, 400 cycle, three-phase power to the power distribution cabinet, which the power distribution cabinet supplies via a variac to each power supply. The power distribution cabinet also contains voltage and temperature monitoring equipment to detect power and cooling malfunctions.

Two 20-ton compressors are located external to the computer room to complete the cooling system.

- Dimensions
 - Base - 103½ inches diameter by 19 inches high
 - Columns - 56½ inches diameter by 77 inches high including height of base
- 24 chassis
- 1662 modules (16 banks); 113 module types
- Each module contains up to 288 IC packages per module
- Power consumption approximately 115 kw input for maximum memory size
- Freon cooled with Freon/water heat exchange
- Three memory options
- Weight 10,500 lbs (maximum memory size)

Cray-1 reference manual

Hardware details

The remainder of this presentation goes into the hardware and implementation details of each major block within Hydra

It will not be of interest to every reader

But if it interests you, by all means, keep reading!

...

Hydra 4

Architecture overview

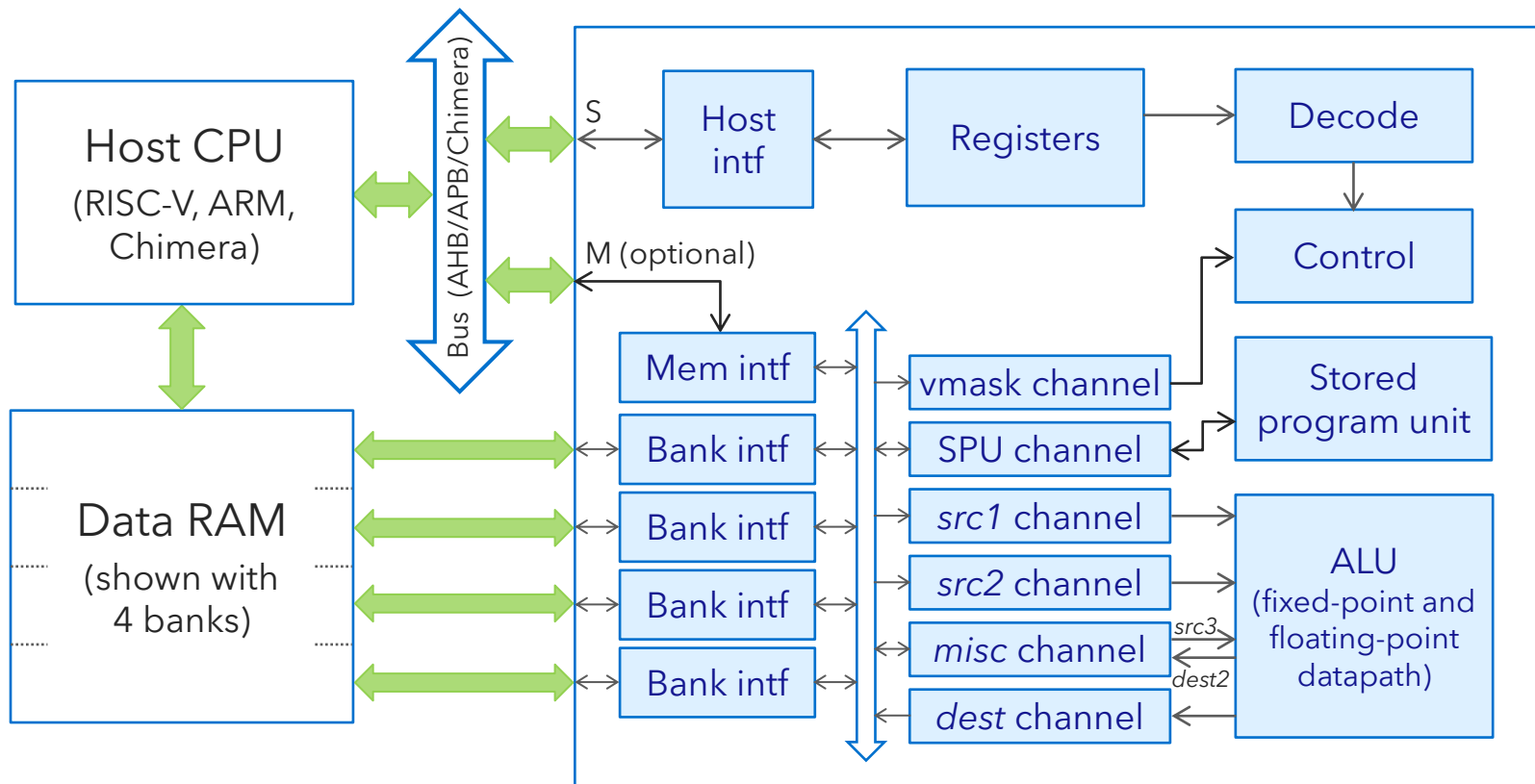
- Introduction
- Using Hydra
- **Hardware and implementation**
 - Block diagram
 - Host interface
 - Decode and control
 - Memory interfaces
 - DMA channels
 - Arithmetic unit (ALU)
 - Stored program unit (SPU)

Hydra 4

Architecture overview

- Introduction
- Using Hydra
- **Hardware and implementation**
 - **Block diagram**
 - Host interface
 - Decode and control
 - Memory interfaces
 - DMA channels
 - Arithmetic unit (ALU)
 - Stored program unit (SPU)

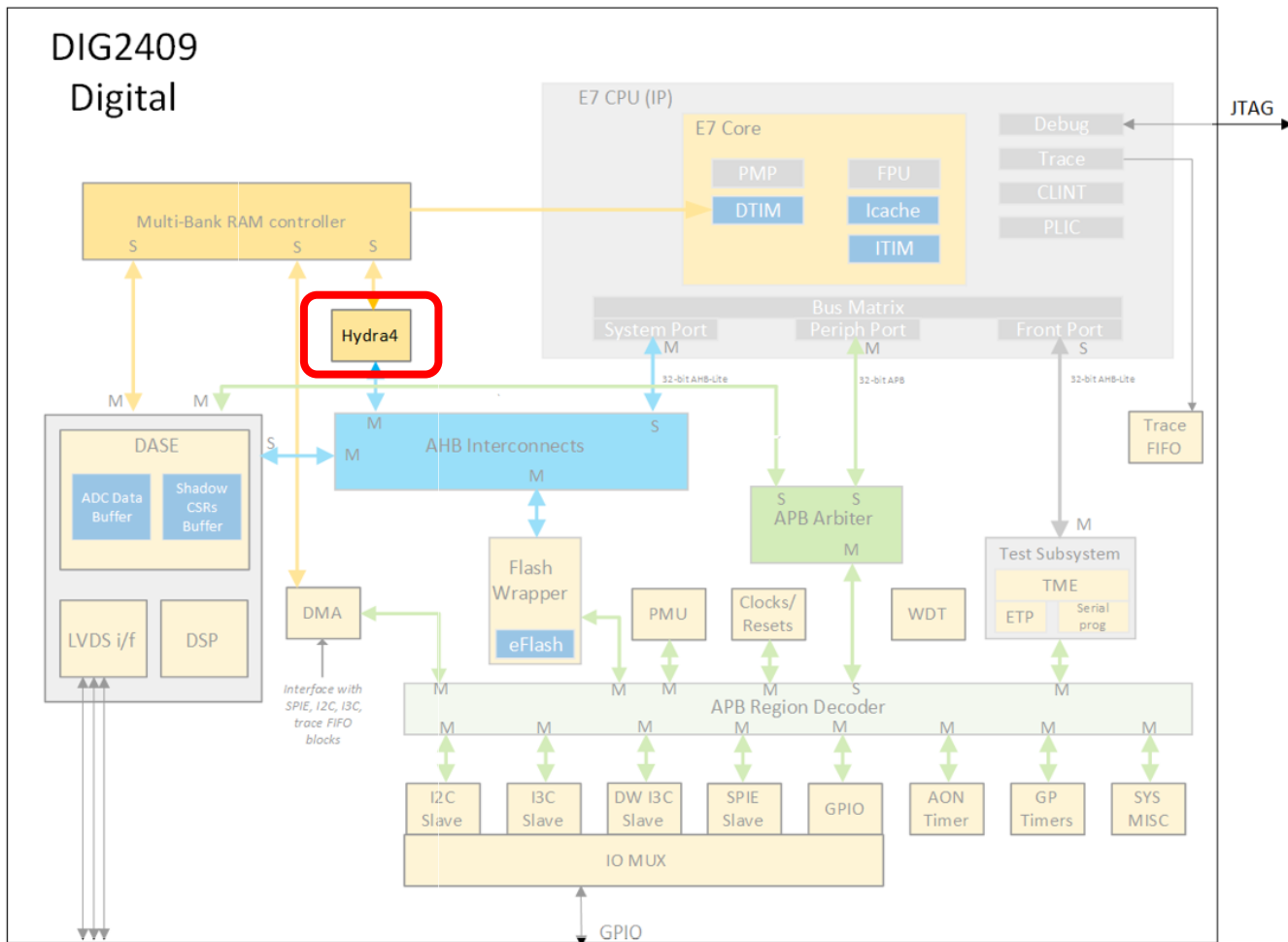
Block diagram



hydra4.v

In Byrd

DIG2409 Digital



Port list: Clocks and control

Name	Dir	Domain	From/to	Description
hclk	In	Clock	clk_rst	Master clock
hclk_rst_n	In	hclk	clk_rst	Async reset, deassert sync'd to hclk
hydra_hclk_en	Out	hclk	clk_rst	Hydra asks to enable hclk clock gate
host_clk	In	Clock	clk_rst	Host interface clock
host_clk_rst_n	In	host_clk	clk_rst	Async reset, deassert sync'd to host_clk
hydra_host_clk_en	Out	host_clk	clk_rst	Hydra asks to enable host_clk clock gate
hclk_is_host_clk	In	hclk	clk_rst	Analogous to "!pclk_is_high_en"
hydra_awake	Out	hclk	HSO	Hydra is running and needs a clock
hydra_intr	Out	host_clk	CPU	Interrupt request to host CPU
scan_en	In	hclk	Test logic	Scan enable (for internal clock gates)

See the comments in hydra4.v for details about all these signals

All the rest of Hydra's ports involve either host interfacing or memory interfacing

Clock gating

By default, Hydra 4 includes internal clock gates (SYNA_CLK_GATE) on its hclk and host_clk clock inputs

- Chip logic can (and should) connect these to pure ungated clock sources

Hydra 4 exports hydra_hclk_en and hydra_host_clk_en signals to allow a chip to locate Hydra's clock gating in the chip's central clock generator instead, if desired

- Define ``HYDRA_EXTERNAL_CLOCK_GATING` in `hydra4_config.vh` to use this option
- There is a combinational logic path from the host interface (e.g., AHB HSEL and HADDR) to `hydra_host_clk_en` that turns on `host_clk` during cycles when the host accesses a Hydra register

The `hydra_awake` signal is used to force the chip's master oscillator awake

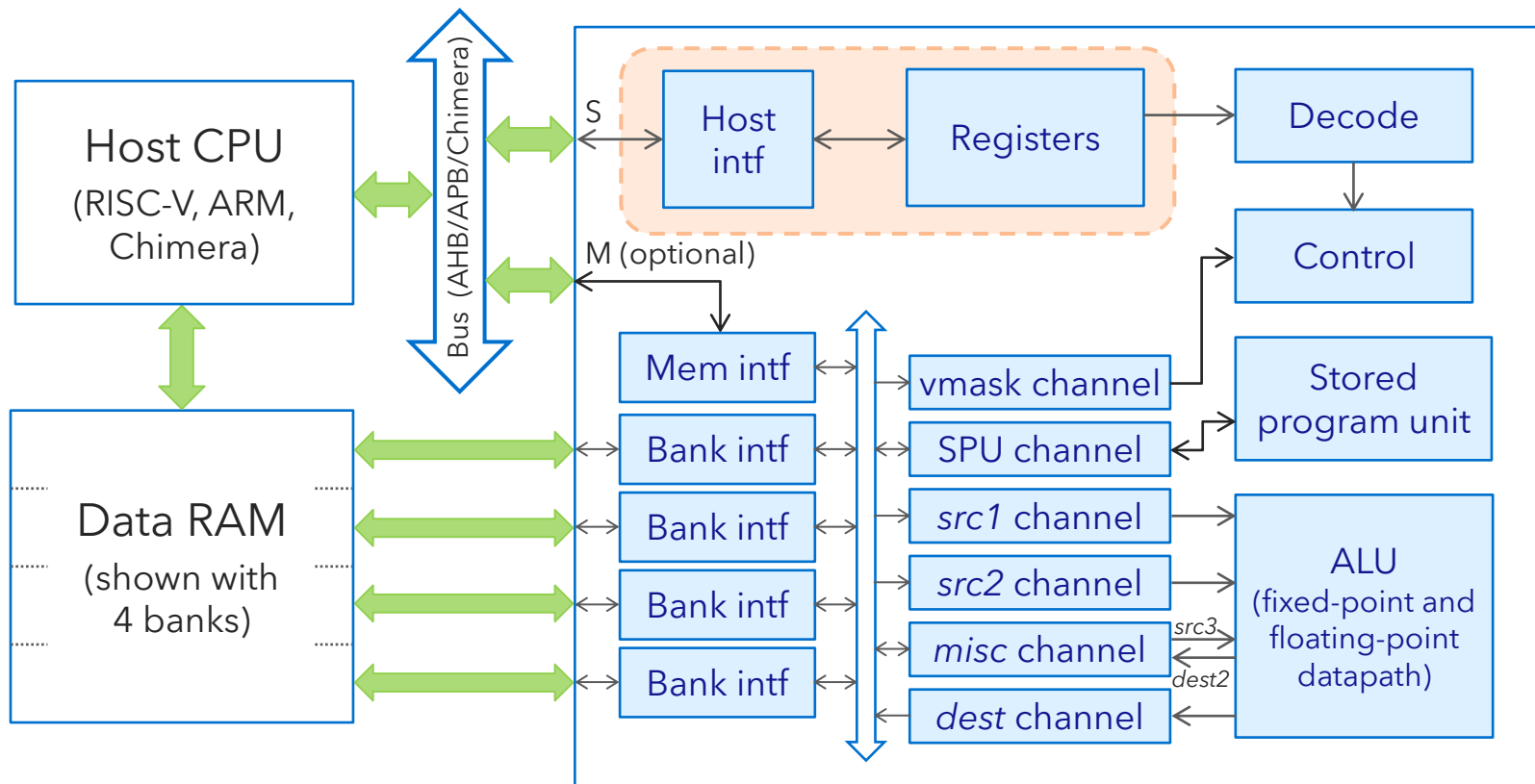
- ... in case the rest of the chip goes to sleep while Hydra is still operating on a vector
- This signal comes straight from a flip-flop so that it can cleanly interface to asynchronous oscillator circuits

Hydra 4

Architecture overview

- Introduction
- Using Hydra
- **Hardware and implementation**
 - Block diagram
 - **Host interface**
 - Decode and control
 - Memory interfaces
 - DMA channels
 - Arithmetic unit (ALU)
 - Stored program unit (SPU)

Block diagram



hydra4.v

Host interface

Hydra 4 supports a choice of Chimera, AHB, APB, or “Direct” interface to the host CPU

The Chimera-style interface is *much* easier to integrate than Hydra 3’s

- All Chimera-facing signals are registered on `host_clk == pclk_master`, not on `hclk`
- Greatly simplifies synthesis when `pclk` runs at $\frac{1}{2}$ `hclk` rate (or generally, $1/N$ `hclk`)

The AHB and APB interfaces comply with AMBA standards

- They also accommodate AHB HCLK or APB PCLK running at `host_clk == hclk` or $1/N$ `hclk` rate
- Byrd and SmartBridge use AHB, running at full speed (`host_clk == hclk`)

The Direct-style interface uses a simple handshake-free, un-pipelined protocol

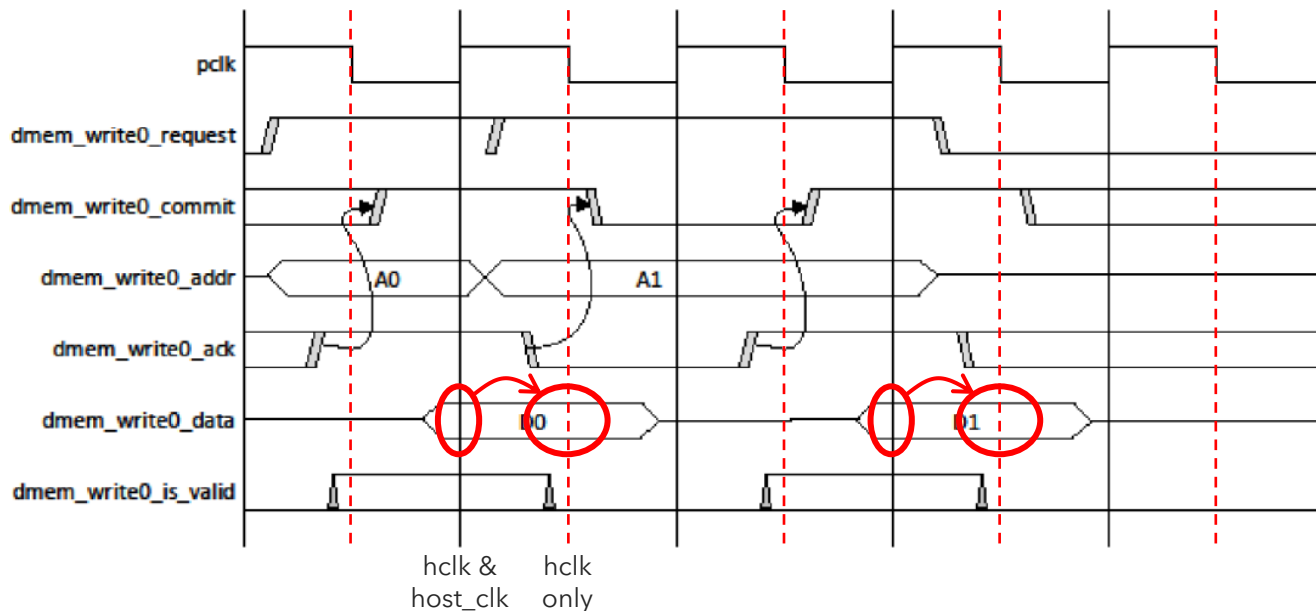
Note that in non-Chimera hosts, peripheral register reads/writes tend to be slow

- For example, Byrd’s E76 CPU has a few cycles of latency for operations on its AHB system port
- The Xtensa CPU on Prometheus/Triton runs its peripheral bus at $\frac{1}{4}$ the CPU clock rate
- Unavoidable: This fact motivated Hydra 4’s opcodes-with- Vn ’s design, and also its SPU

Host interface: Chimera-style

Input signals no longer require multicycle_paths

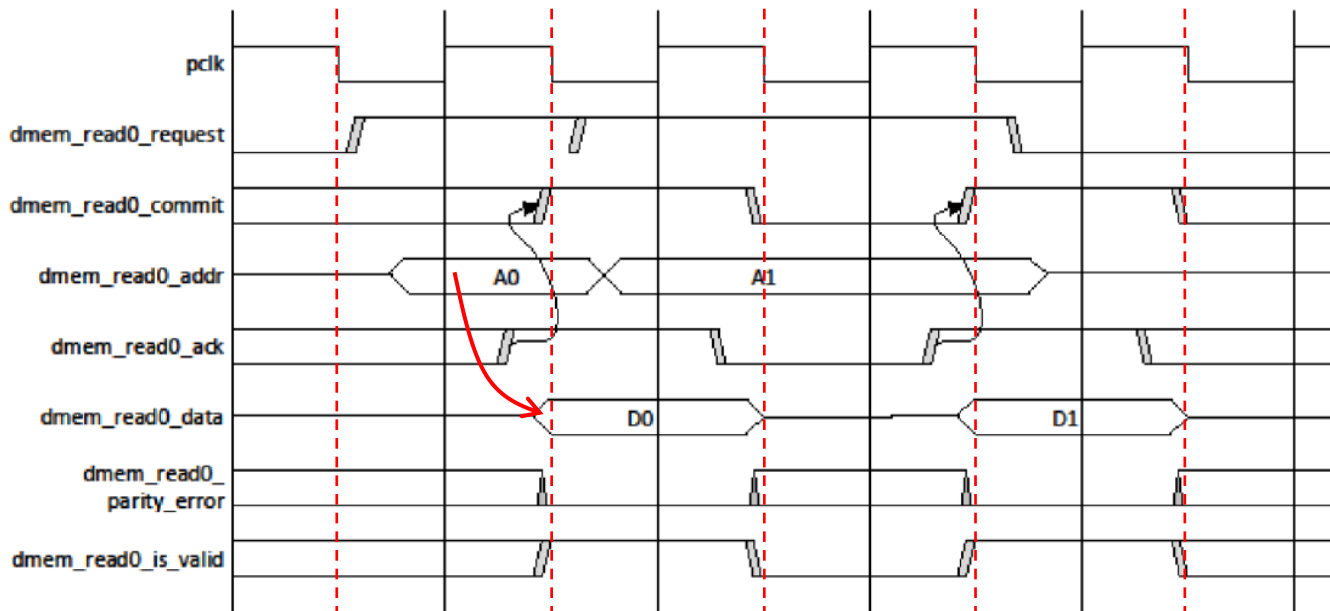
- dmem_read0/write0_addr, dmem_write0_data, etc., are sampled on the host_clk rising edge (i.e., on the *true* pclk rising edge)
- Write data is captured in a host_clk-domain write buffer and applied during the next hclk cycle



Host interface: Chimera-style

Output signals are now stable throughout the pclk period

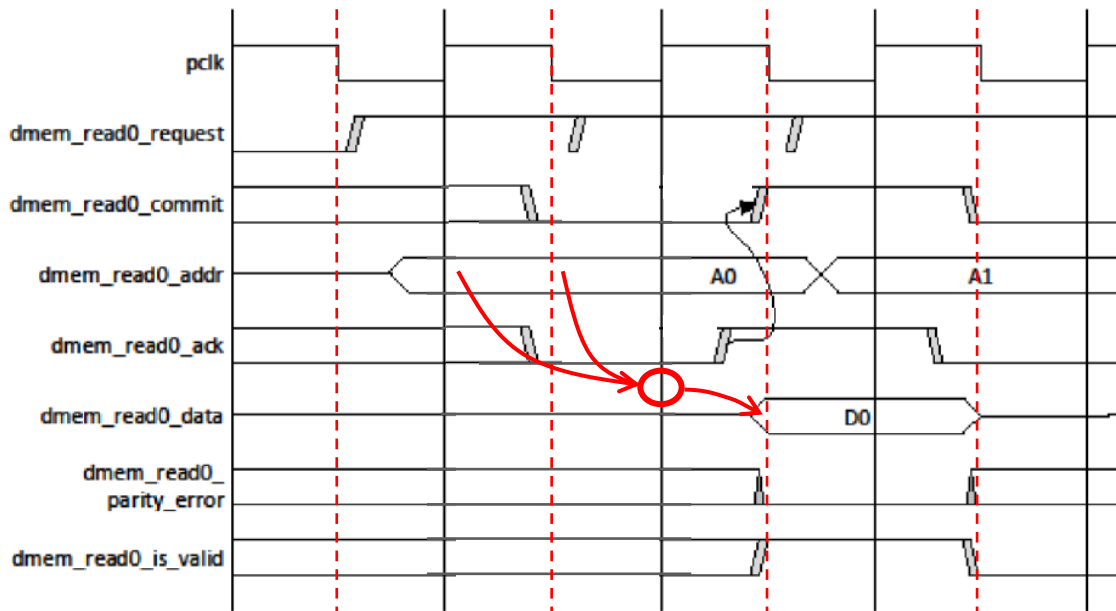
- dmem_read0_data, dmem_read0/write0_ack, hydra_intr, hydra_awake, etc., change only on the host_clk rising edge, driven by host_clk-domain registers
- Hydra registers such as VL, that are only written by the host, are clocked by host_clk



Host interface: Chimera-style

Some registers (such as STATUS) really must operate in the hclk domain

- These are captured in a host_clk-domain read buffer and delivered after one wait state



Host interface signal timing

Hydra 4 aims for the host interface to have the simplest possible signal timing

- Every input goes only to flops clocked on host_clk
- Every output comes only from flops clocked on host_clk
- Not much logic delay in between the ports and those registers

Port list: Chimera-style host interface (writing)

Name	Dir	Domain	From/to	Description
dmem_write0_request	In	host_clk	Addr dec	Standard Chimera signal
dmem_write0_ack[1:0]	Out	host_clk	Addr dec	Standard Chimera signal
dmem_write0_commit	In	host_clk	Addr dec	Standard Chimera signal
dmem_write0_addr_is_hydra	In	host_clk	Addr dec	Hydra block address decode
dmem_write0_addr_low[5:0]	In	host_clk	Addr dec	Low address bits within Hydra block
dmem_write0_size[1:0]	In	host_clk	Addr dec	Standard Chimera signal
dmem_write0_data[31:0]	In	host_clk	Addr dec	Standard Chimera signal
dmem_write0_guest_safe	Out	host_clk	Addr dec	Standard Chimera signal

Port list: Chimera-style host interface (reading)

Name	Dir	Domain	From/to	Description
dmem_read0_request	In	host_clk	Addr dec	Standard Chimera signal
dmem_read0_ack[1:0]	Out	host_clk	Addr dec	Standard Chimera signal
dmem_read0_addr_is_hydra	In	host_clk	Addr dec	Hydra block address decode
dmem_read0_addr_low[5:0]	In	host_clk	Addr dec	Low address bits within Hydra block
dmem_read0_size[1:0]	In	host_clk	Addr dec	Standard Chimera signal
dmem_read0_data[31:0]	Out	host_clk	Addr dec	Standard Chimera signal
dmem_read0_guest_safe	Out	host_clk	Addr dec	Standard Chimera signal
hydra_mutex_release_out	Out	host_clk	CPU	Standard Chimera signal

Host interface: AHB

The host interface complies with the AMBA 5 AHB standard ("AHB-Lite")

- Again, all signals operate in the host_clk domain
- A write buffer and a read buffer transfer data between the hclk and host_clk domains
- Write buffer forwarding ensures the host sees the correct value if it writes a register and immediately reads it back

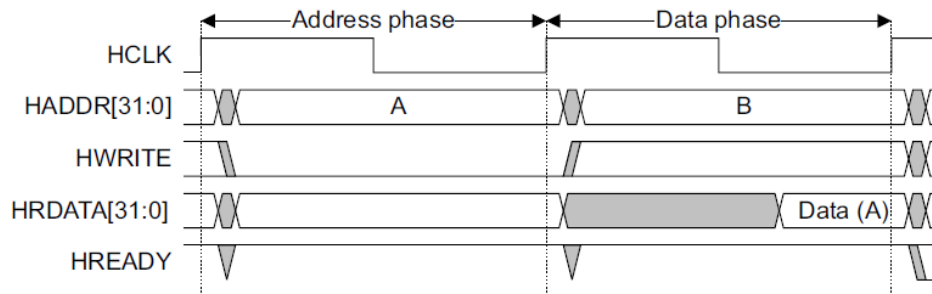


Figure 3-1 Read transfer

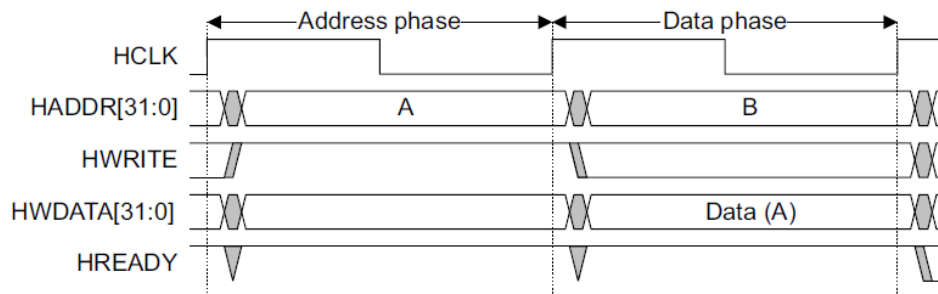


Figure 3-2 Write transfer

Host interface: AHB

Hydra's AHB host interface occasionally generates wait states to delay its response to the host's read or write request

- ⇒ When a read must go through the read buffer
- ⇒ When the host reads IRQ on the very next cycle after writing to OPCODE (gives time to decode the opcode to report errors)
- ⇒ For optional handshaking on writes to OPCODE and reads to G0 and G1

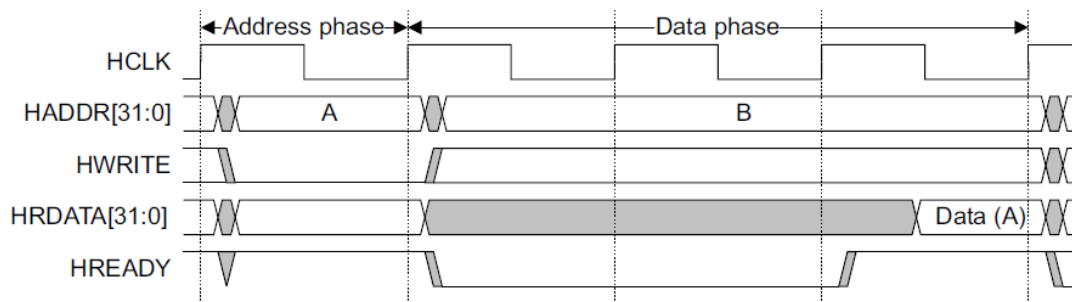


Figure 3-3 Read transfer with two wait states

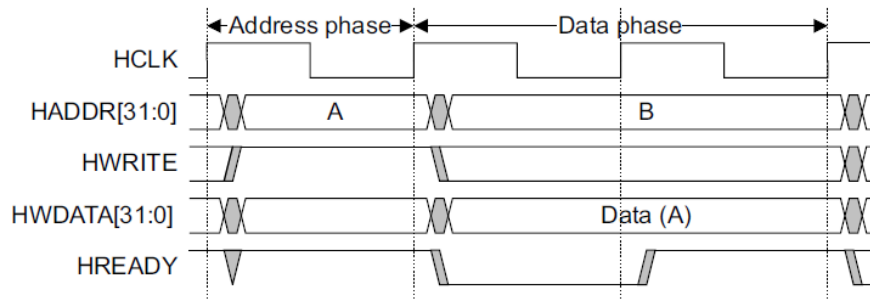


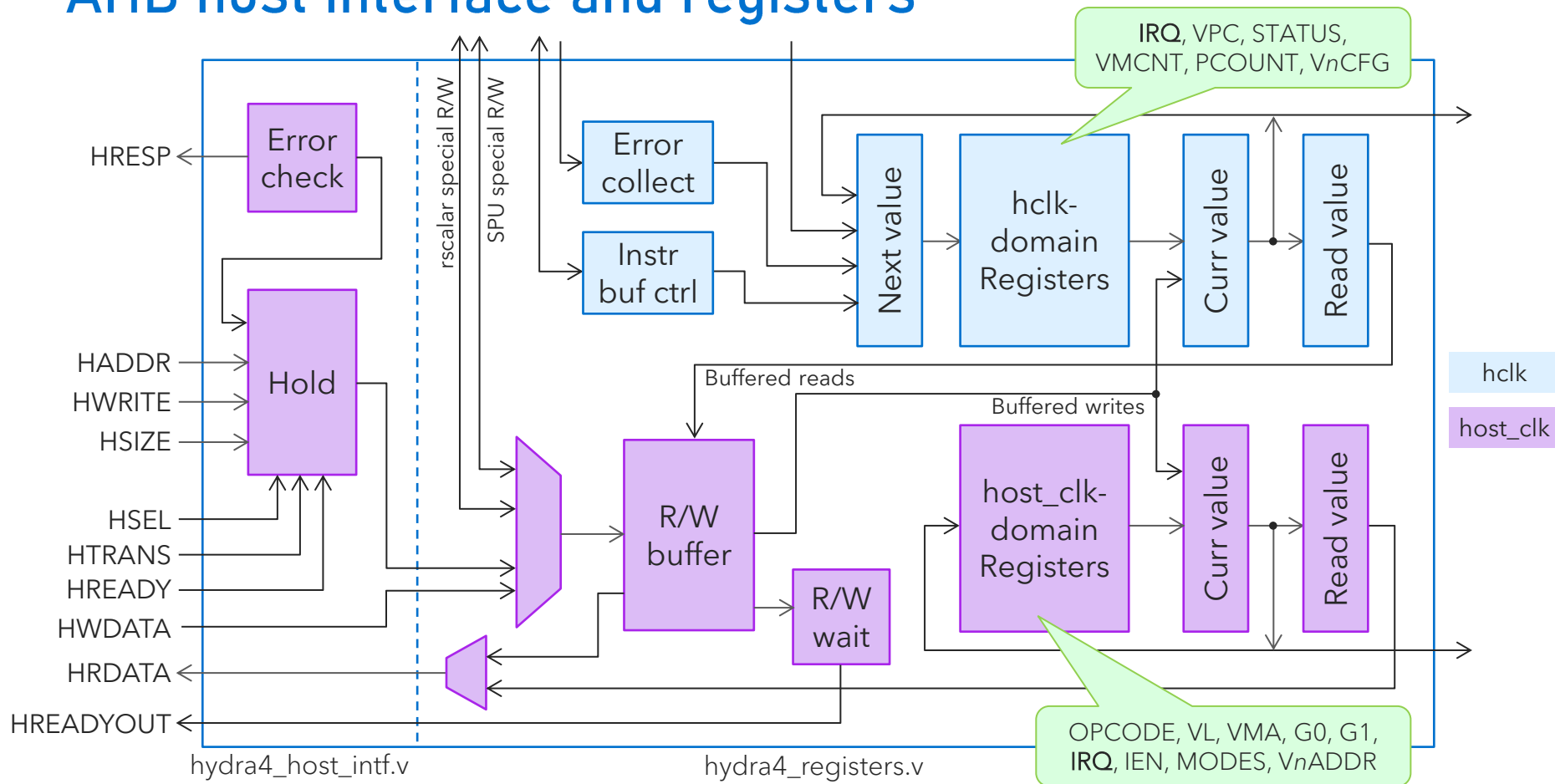
Figure 3-4 Write transfer with one wait state

Port list: AHB host interface

Name	Dir	Domain	From/to	Description
ahb_hsel	In	host_clk	AHB mux	Standard AHB signal
ahb_hready	In	host_clk	AHB mux	Standard AHB signal
ahb_haddr[10:0]	In	host_clk	AHB mux	Standard AHB signal (low bits only)
ahb_hprot[3:0]	In	host_clk	AHB mux	Standard AHB signal
ahb_hsize[2:0]	In	host_clk	AHB mux	Standard AHB signal
ahb_htrans[1:0]	In	host_clk	AHB mux	Standard AHB signal
ahb_hwrite	In	host_clk	AHB mux	Standard AHB signal
ahb_hwdata[31:0]	In	host_clk	AHB mux	Standard AHB signal
ahb_hrdata[31:0]	Out	host_clk	AHB mux	Standard AHB signal
ahb_hreadyout	Out	host_clk	AHB mux	Standard AHB signal
ahb_hresp	Out	host_clk	AHB mux	Standard AHB signal

The width of ahb_addr[] is given by `HYDRA_HOST_ADDR_WIDTH in hydra4_config.vh, any number from 7 to 31

AHB host interface and registers



Host interface: APB

Hydra offers an APB host interface option

- ... but Hydra discourages using it, since APB is a slow protocol!

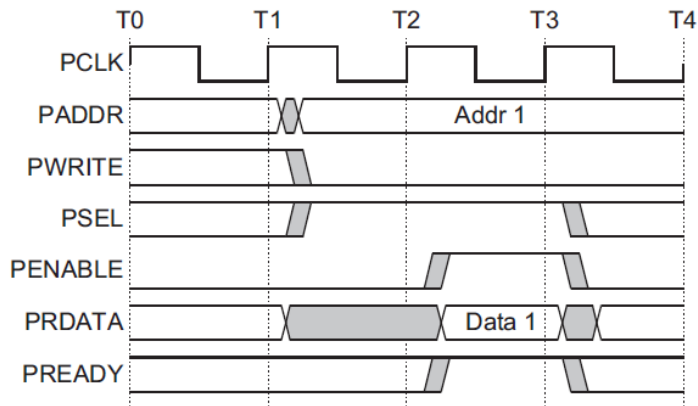


Figure 3-4 Read transfer with no wait states

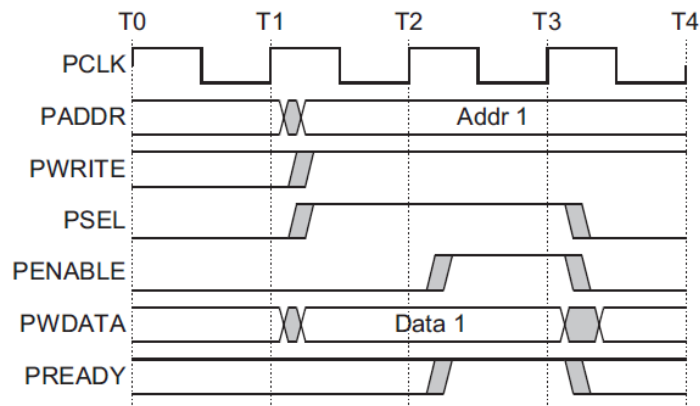


Figure 3-1 Write transfer with no wait states

Port list: APB host interface

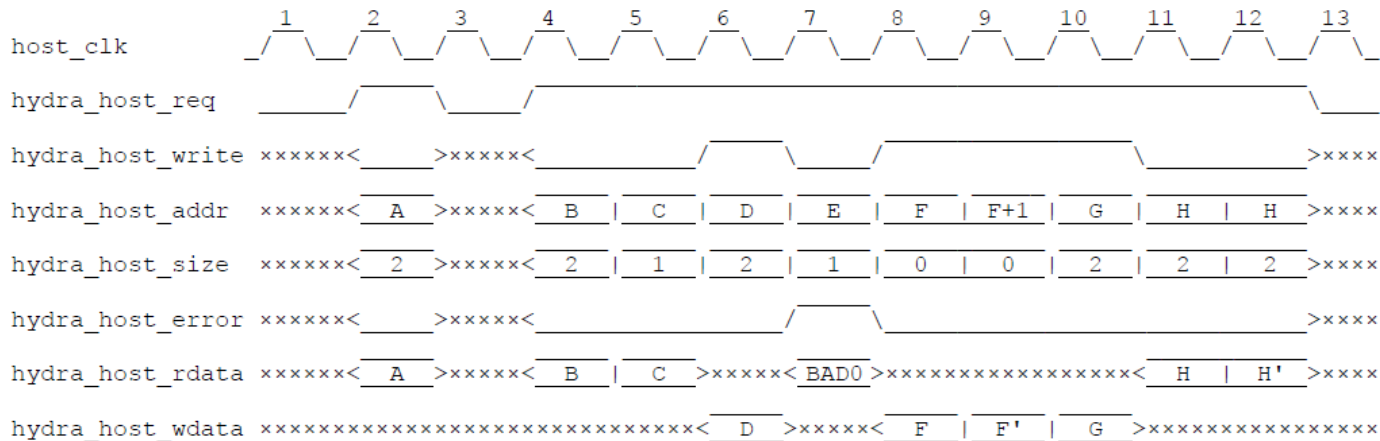
Name	Dir	Domain	From/to	Description
apb_psel	In	host_clk	APB bridge	Standard APB signal
apb_penable	In	host_clk	APB bridge	Standard APB signal
apb_paddr[10:0]	In	host_clk	APB bridge	Standard APB signal (low bits only)
apb_pprot[2:0]	In	host_clk	APB bridge	Standard APB signal
apb_pstrb[3:0]	In	host_clk	APB bridge	Standard APB signal
apb_pwrite	In	host_clk	APB bridge	Standard APB signal
apb_pwdata[31:0]	In	host_clk	APB bridge	Standard APB signal
apb_prdata[31:0]	Out	host_clk	APB bridge	Standard APB signal
apb_pready	Out	host_clk	APB bridge	Standard APB signal
apb_pslverr	Out	host_clk	APB bridge	Standard APB signal

Host interface: Direct-style

Hydra's Direct-style host interface is a simple interface for use in chips that do not support the Chimera, AHB, or APB protocol

Very simple: No "ack" signals; no wait states; no AHB-style pipelining

Every request takes place, and completes, within one self-contained host_clk cycle



Port list: Direct host interface

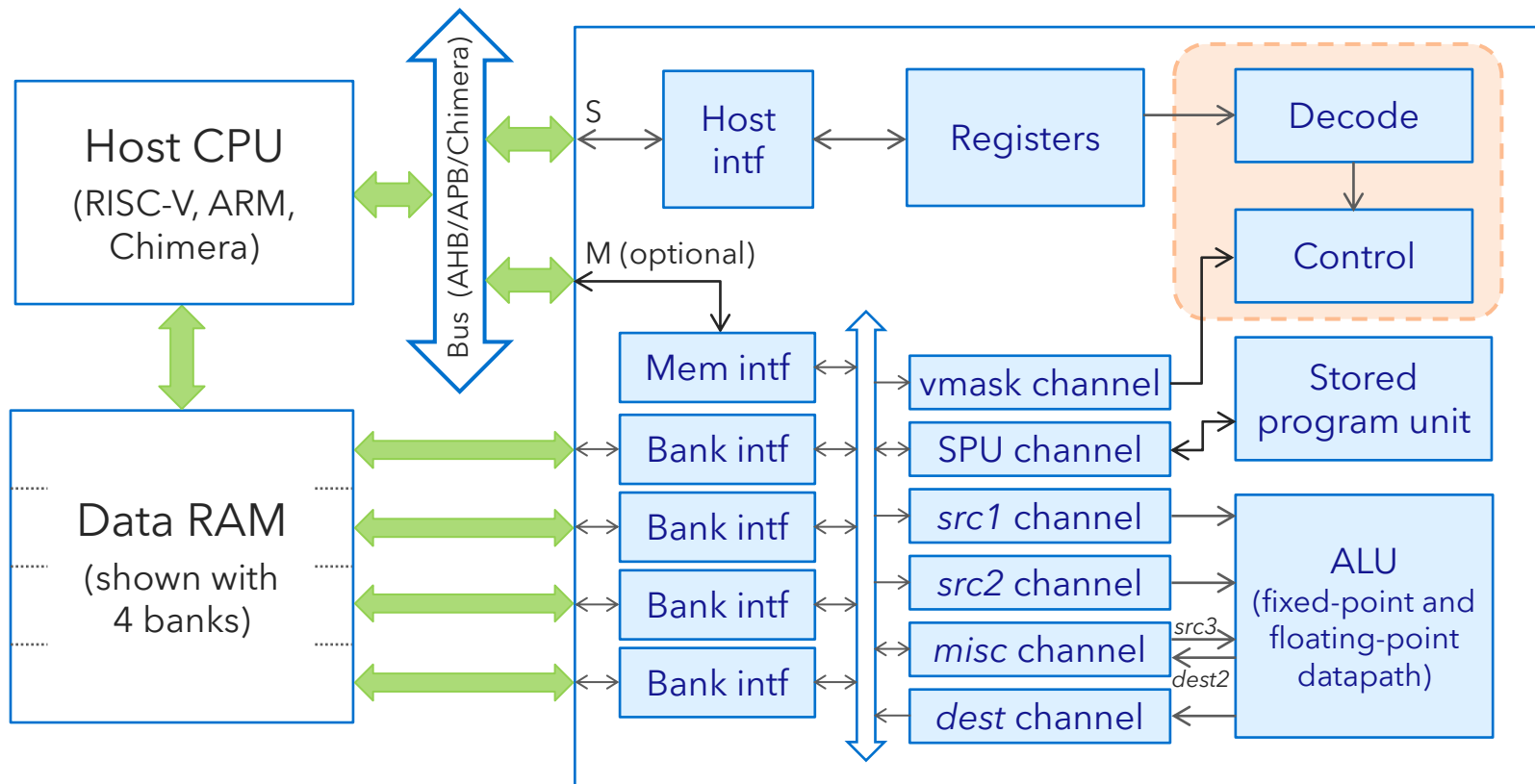
Name	Dir	Domain	From/to	Description
hydra_host_req	In	host_clk	Periph bus	Request a read or write
hydra_host_write	In	host_clk	Periph bus	This request is for a write vs read
hydra_host_addr[6:0]	In	host_clk	Periph bus	Byte address of register (low bits only)
hydra_host_size[1:0]	In	host_clk	Periph bus	Access size (0=byte, 1=short, 2=long)
hydra_host_priv	In	host_clk	Periph bus	High if access is from privileged code
hydra_host_wdata[31:0]	In	host_clk	Periph bus	Data to write on this cycle
hydra_host_rdata[31:0]	Out	host_clk	Periph bus	Data that was read on this cycle
hydra_host_error	Out	host_clk	Periph bus	High if this access request is invalid

Hydra 4

Architecture overview

- Introduction
- Using Hydra
- **Hardware and implementation**
 - Block diagram
 - Host interface
 - **Decode and control**
 - Memory interfaces
 - DMA channels
 - Arithmetic unit (ALU)
 - Stored program unit (SPU)

Block diagram



hydra4.v

Instruction decoding

The Decode block receives the 30-bit opcode when the host writes to OP CODE

- Also the current values of “instruction buffered” registers: VL, VMA, G0, G1, MODES.ROUND
- And the four VnCFG’s and VnADDR’s selected by the low bits of the opcode:

‘00jj rrss mmff oooo eeee bbbb aaaa dddd’



The decoder validates the instruction

- Adjusts IRQ.ERROR quickly enough so the host can see the error code, even if it reads IRQ immediately after writing to OP CODE

The decoder pre-processes the instruction to simplify downstream processing

- For example, instructions that write to the vector mask are converted to a $K=1$ packed vector *dest*
- Instructions written to COMPAT_OPCODE are fully translated from Hydra 3 to Hydra 4

The decoder captures the instruction into a holding buffer

- This represents the first stage of the instruction buffer as seen by the user

Instruction control

The Control block receives the instruction from the decoder when it is ready

- If Hydra was idle when the user wrote to OP CODE: On the very next hclk
- If Hydra was already running an instruction: When the current instruction finishes

The controller captures the instruction for execution

- This is the second stage of the instruction buffer
- The instruction buffer is expensive! $228 + 6 \cdot W$ flops for each stage
- But double-buffering the instructions greatly improves performance
- The host can set up the next vector operation while the current operation is still running
- In most cases, double-buffering completely eliminates this instruction setup overhead

Instruction control

The controller sends the instruction to the various channels and the ALU

- Currently, there is not much additional processing or “state” in the controller itself

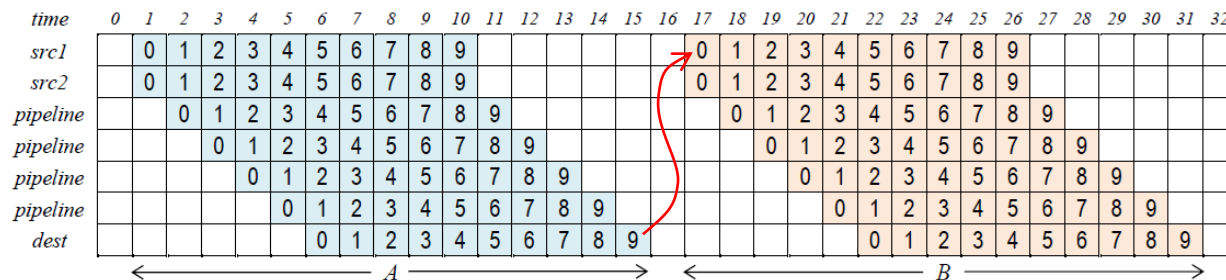
Eventually, the controller will implement “instruction joining”

- Hydra can begin reading the next instruction’s *src* operands while the current instruction is still finishing up in the ALU and *dest* channel
- This feature is defined in the *Hydra 4 User’s Guide*, but not yet implemented in Hydra 4.0–4.2
 - There was not enough time for it in the Byrd and SmartBridge project schedules

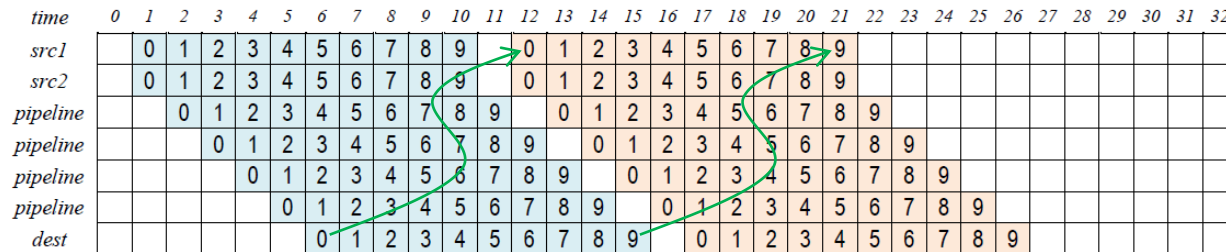
Instruction joining

Hydra 3's and 4.2's pipeline can do only one instruction at a time

- Instruction A must complete all its *dest* writes before B can read any *src*
- Works correctly even if any $dest_A[i]$ is the same address as any $src_B[j]$



Hydra 4.x will be able to optimize this in cases when $dest_A$ and src_B do not overlap, or when they have only a "simple" overlap

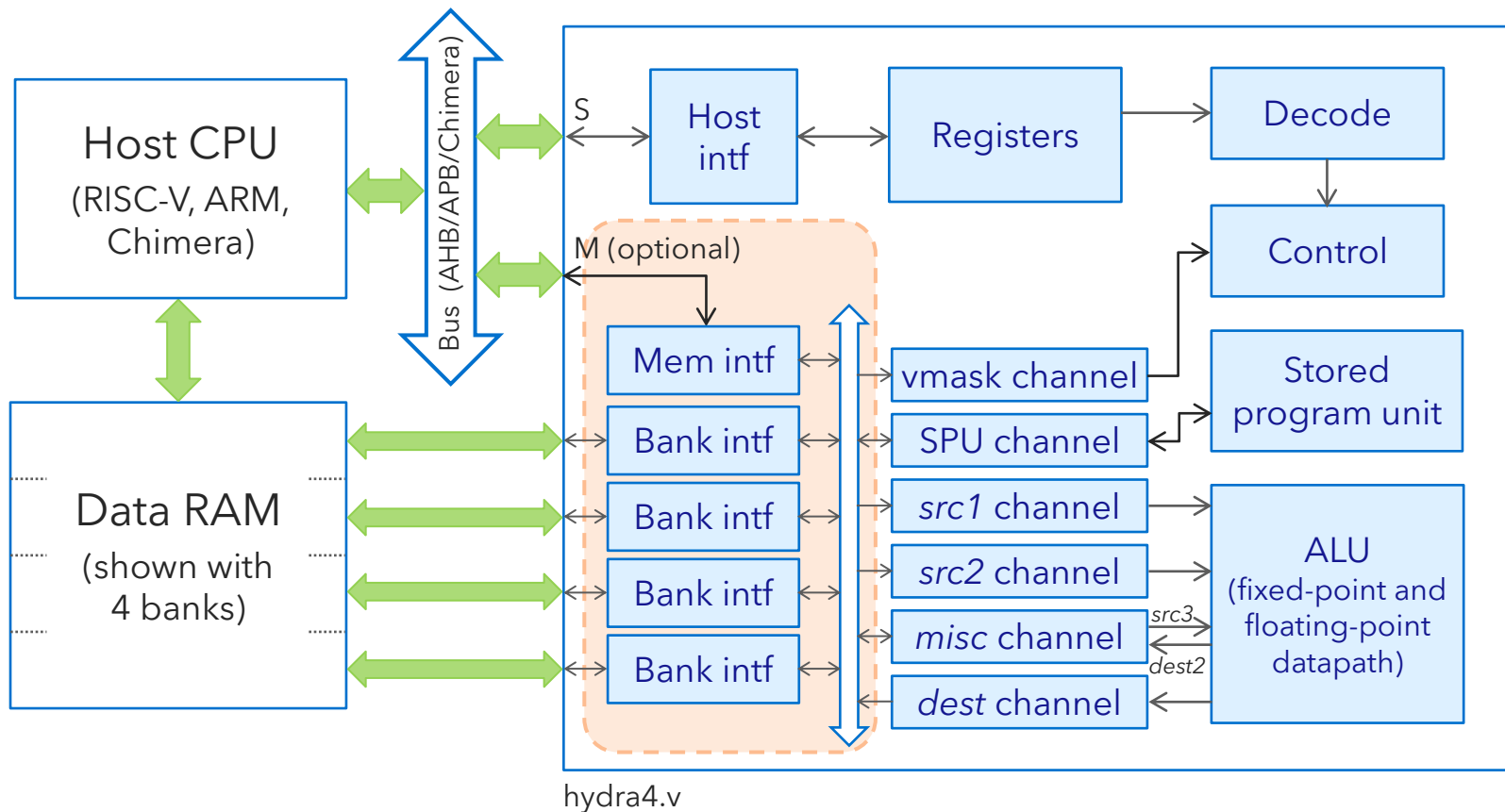


Hydra 4

Architecture overview

- Introduction
- Using Hydra
- **Hardware and implementation**
 - Block diagram
 - Host interface
 - Decode and control
 - **Memory interfaces**
 - DMA channels
 - Arithmetic unit (ALU)
 - Stored program unit (SPU)

Block diagram



Memory interfaces

By default, Hydra 4 exports one memory interface for each RAM bank

- Each bank interface connects directly to one banked RAM
- `hydra4_config.vh` configures the number of banks (Byrd and SmartBridge have 4 banks)
- It also configures the address pattern for each bank

Preferably, a chip arranges its data RAM in *interleaved* banks

- I.e., the bank address patterns are configured to use the low address bits (bits 3:2 or bits 4:2) to select the bank

Arbitration among banks takes place inside the Hydra core

- Each bank interface sees *only* addresses in that bank
- The arbitration is isolated by a pipeline stage from port-level bus signal timing

Every bank may do an 8-, 16-, or 32-bit-wide read or write on every cycle

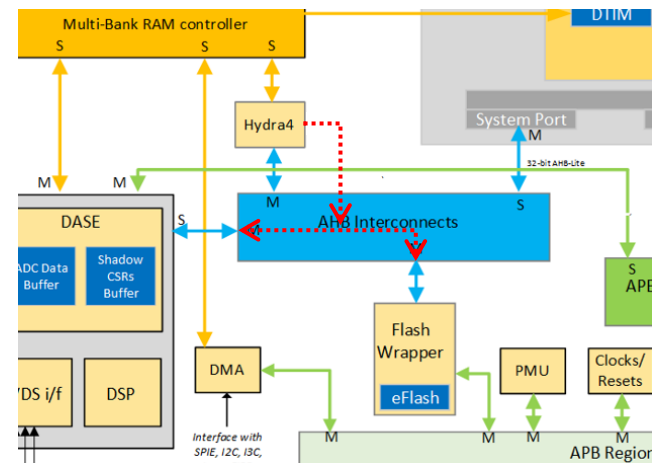
Memory interfaces

If an address does not fall in any bank...

- By default, this is flagged as a bus error
- Hydra 4 has an optional "General Memory port"
- If it is present, non-bank accesses go out on this port
- Typically the General Memory port would go to the chip's bus matrix, e.g., to allow Hydra to "DMA" into peripherals, Flash, or other memories
- (Byrd A0 and SB7900 do not have a general memory port: No need to DMA into DAQ/DASE or other peripherals; uncached Flash is too slow for direct access to be useful)
- (But Byrd B0 uses it to connect Hydra to SYS_MEM RAM)

If W is configured with just enough address bits to cover the entire data RAM (e.g., as it is on Byrd and SmartBridge) ...

- Then every possible address falls in one of the banks
- ... and a General Memory port is not needed or useful



Memory protocol options

Hydra 4 offers two types of signaling for its RAM interfaces:

- Direct style (similar to Hydra 3's RAM signaling)
- AHB style

The main difference is whether write data and "ack" are pipelined

- This also affects the behavior of request/addr/size signals during wait states

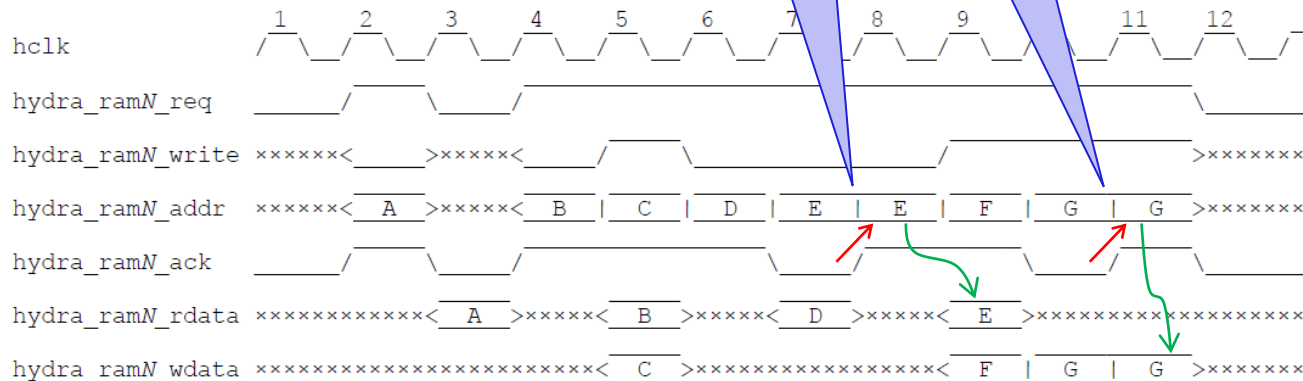
The "Direct" method is more efficient and easier to work with

- Very similar to the "tightly coupled RAM" interface of many commercial MCUs
- Byrd and SmartBridge use it for their direct Hydra-to-RAM interfaces instead of AHB-style
- Note that this signaling convention is *not* the same as Chimera's dmem_read/write bus signaling
- But it is much like how Hydra 3 accessed the RAM in prior Chimera-based chips

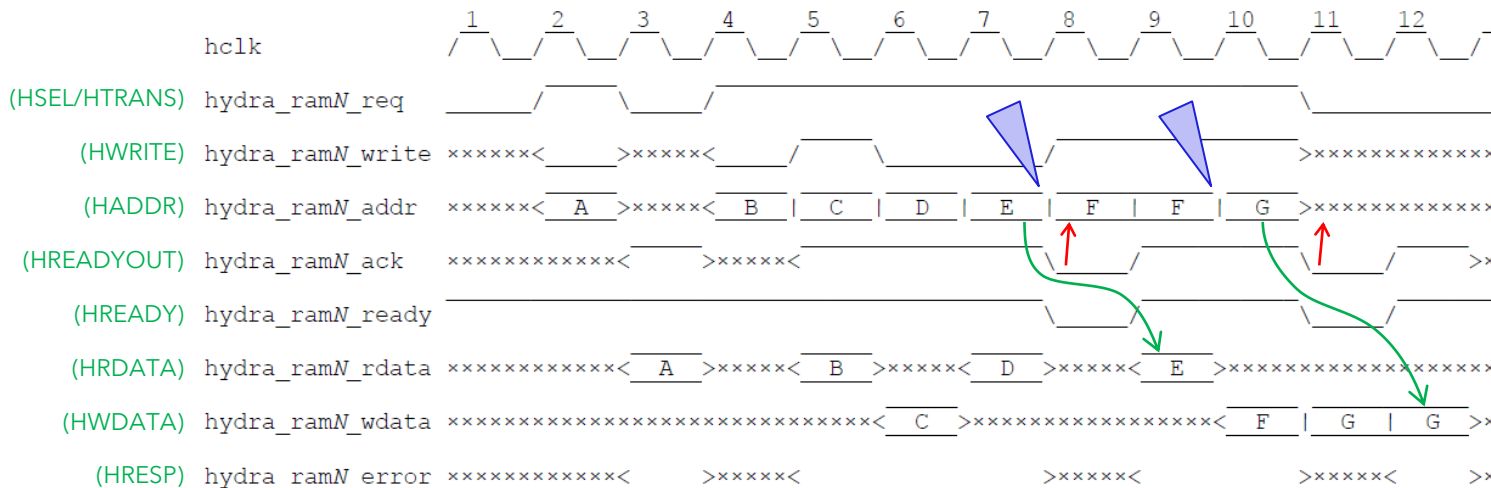
The Banked Memory and General Memory interfaces can each independently choose either Direct or AHB signaling

Memory interface signals

Direct style ⇒



AHB style ⇒



Memory interface signal timing

The memory interface operates in the hclk clock domain

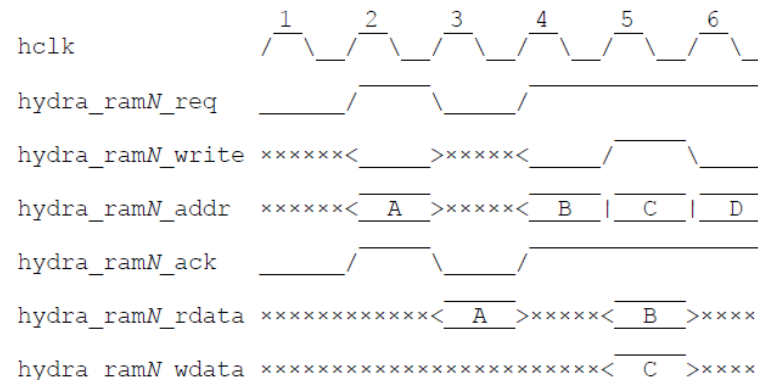
Most memory interface signals have little logic delay between ports and flops

One exception: `hydra_ramN_ack`

- Hydra has significant logic between this port and its internal flops
- Is this acceptable? (Initial synthesis results say “yes”)
- These are among the worst Hydra timing paths in the synthesis report (but better than the worst E76 paths)
- Hydra’s RTL has an option to pipeline this signal, but it is expensive (many gates and flops) so we didn’t use it for Byrd A; we did need it for Byrd B

Previously, the worst timing path was from `hydra_ramN_rdata` to the many channel FIFOs

- This has already been fixed with a pipeline stage (also expensive – optional, but needed in Byrd!)



Port list: RAM bank interface (if Direct-style)

Name	Dir	Domain	From/to	Description
hydra_ramN_req	Out	hclk	RAM bank <i>N</i>	Request a read or write
hydra_ramN_ack[1:0]	In	hclk	RAM bank <i>N</i>	RAM can perform the current request
hydra_ramN_addr[W-1:0]	Out	hclk	RAM bank <i>N</i>	<i>W</i> = `HYDRA_WIDTH_W in config.vh
hydra_ramN_write	Out	hclk	RAM bank <i>N</i>	Current request is a write
hydra_ramN_size[1:0]	Out	hclk	RAM bank <i>N</i>	Current request data size (8, 16, or 32)
hydra_ramN_rdata[31:0]	In	hclk	RAM bank <i>N</i>	RAM read data (delayed 1 hclk)
hydra_ramN_wdata[31:0]	Out	hclk	RAM bank <i>N</i>	Current RAM write data
hydra_ramN_ready	Out	hclk	–	(Unused for Direct-style)
hydra_ramN_error	In	hclk	–	(Supported but typically unused and tied low for direct-style memories)

Repeated for each $N = 0 \dots \text{`HYDRA_NUM_BANKS}-1$

These signals operate much like the old Hydra 3 RAM interface signals

- (Except that bank arbitration happens inside Hydra instead of in the RAM controller)

Port list: RAM bank interface (if AHB-style)

Name	Dir	Domain	From/to	Description
hydra_ramN_req	Out	hclk	RAM bank N	Connect to AHB HSEL
hydra_ramN_ack	In	hclk	RAM bank N	Connect to AHB HREADYOUT
hydra_ramN_addr[W-1:0]	Out	hclk	RAM bank N	Connect to AHB HADDR[W-1:0]
hydra_ramN_write	Out	hclk	RAM bank N	Connect to AHB HWRITE
hydra_ramN_size[2:0]	Out	hclk	RAM bank N	Connect to AHB HSIZE (8, 16, or 32 bits)
hydra_ramN_rdata[31:0]	In	hclk	RAM bank N	Connect to AHB HRDATA
hydra_ramN_wdata[31:0]	Out	hclk	RAM bank N	Connect to AHB HWDATA
hydra_ramN_ready	Out	hclk	RAM bank N	Connect to AHB HREADY
hydra_ramN_error	In	hclk	RAM bank N	Connect to AHB HRESP

Repeated for each $N = 0 \dots \text{`HYDRA_NUM_BANKS}-1$

Note: hydra_ramN_wdata, _ready, _error are delayed 1 hclk relative to Direct-style

Also connect HBURST = 3'b000, HPROT = 4'b0001, HMASTLOCK = 1'b0
and HTRANS = { hydra_ramN_req, 1'b0 }

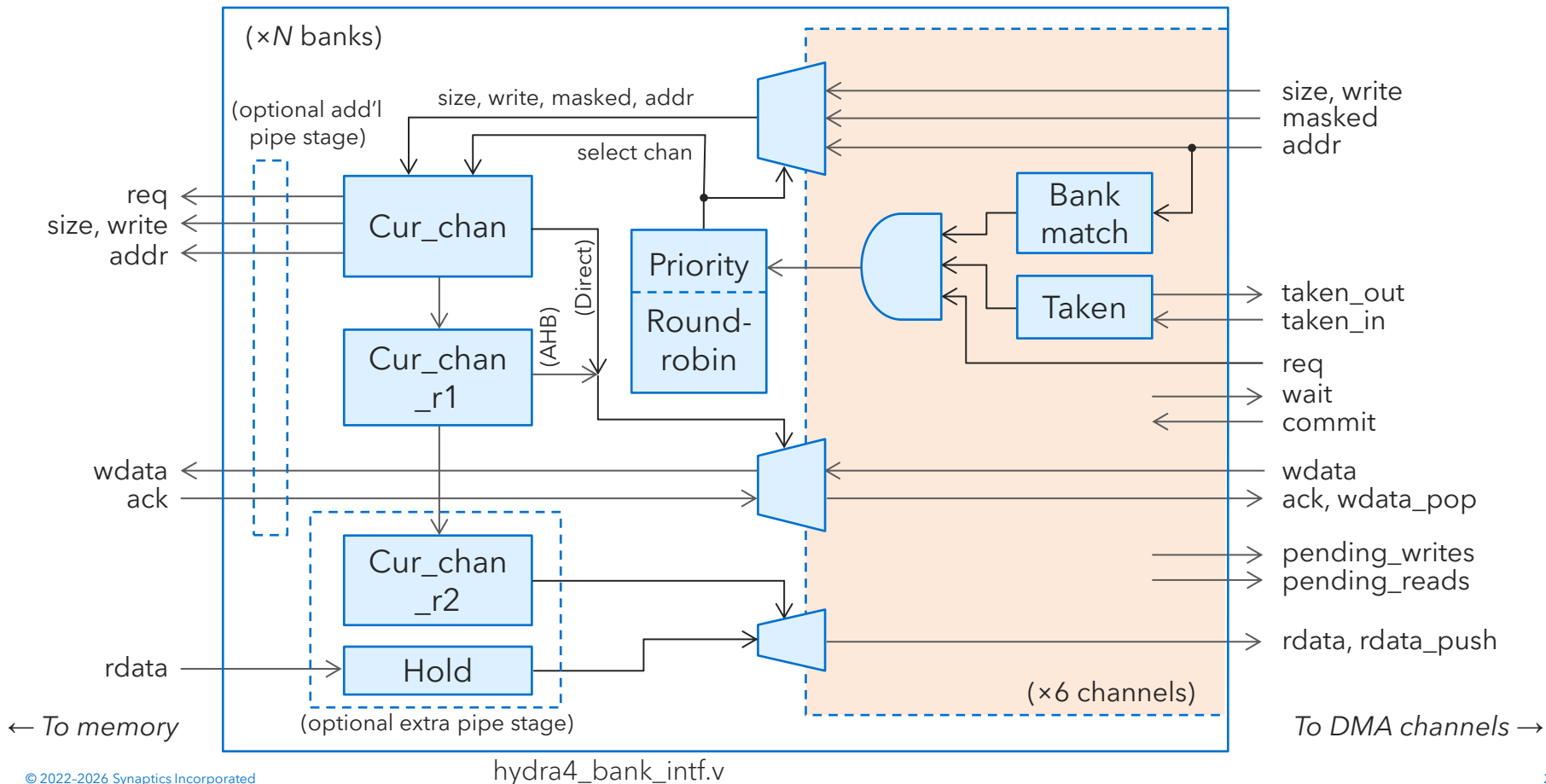
Port list: General Direct-style master interface

Name	Dir	Domain	From/to	Description
gen_mem_req	Out	hclk	Gen RAM	Request a read or write
gen_mem_ack[1:0]	In	Hclk	Gen RAM	RAM can perform the current request
gen_mem_addr[W-1:0]	Out	hclk	Gen RAM	$W = \text{`HYDRA_WIDTH_W in config.vh}$
gen_mem_prot	Out	hclk	Gen RAM	Current request is a privileged access
gen_mem_write	Out	hclk	Gen RAM	Current request is a write
gen_mem_size[1:0]	Out	hclk	Gen RAM	Current request data size (8, 16, or 32)
gen_mem_rdata[31:0]	In	hclk	Gen RAM	RAM read data (delayed 1 hclk)
gen_mem_wdata[31:0]	Out	hclk	Gen RAM	Current RAM write data
gen_mem_error	In	hclk	Gen RAM	(Supported but typically unused and tied low for direct-style memories)

Port list: General AHB master interface

Name	Dir	Domain	From/to	Description
gen_mem_hsel	Out	hclk	AHB mux	Standard AHB signal
gen_mem_hready	Out	Hclk	AHB mux	Standard AHB signal
gen_mem_haddr[W-1:0]	Out	hclk	AHB mux	Standard AHB signal
gen_mem_hprot[3:0]	Out	hclk	AHB mux	Standard AHB signal
gen_mem_hsize[2:0]	Out	hclk	AHB mux	Standard AHB signal (8, 16, or 32 bits)
gen_mem_htrans[1:0]	Out	hclk	AHB mux	Standard AHB signal (IDLE or NONSEQ)
gen_mem_hburst[2:0]	Out	hclk	AHB mux	Standard AHB signal (always 3'b000)
gen_mem_hwrite	Out	hclk	AHB mux	Standard AHB signal
gen_mem_hmastlock	Out	hclk	AHB mux	Standard AHB signal (always 1'b0)
gen_mem_hwdata[31:0]	Out	hclk	AHB mux	Standard AHB signal
gen_mem_hrdata[31:0]	In	hclk	AHB mux	Standard AHB signal
gen_mem_hreadyout	In	hclk	AHB mux	Standard AHB signal
gen_mem_hresp	In	hclk	AHB mux	Standard AHB signal

Memory interfaces

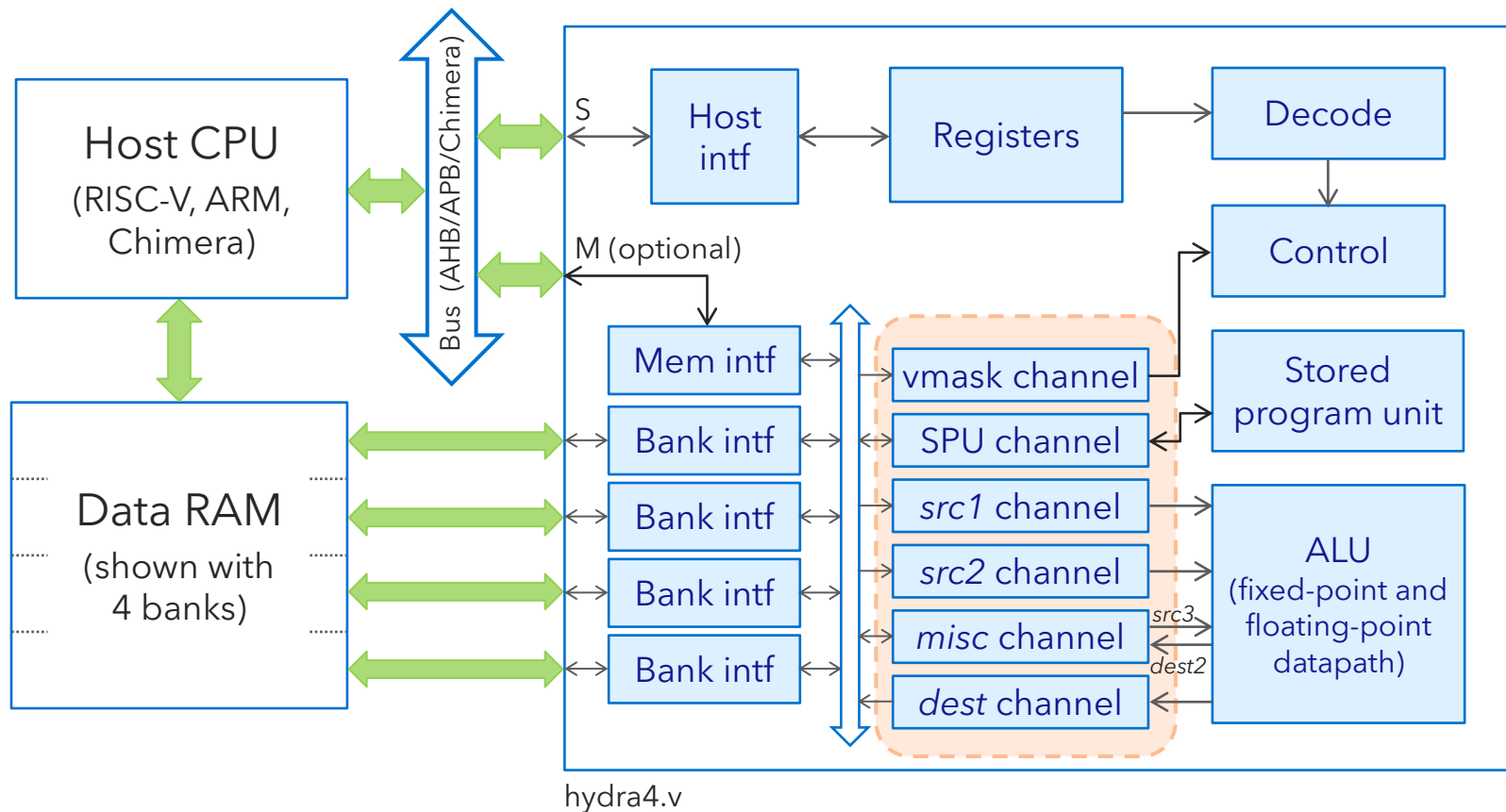


Hydra 4

Architecture overview

- Introduction
- Using Hydra
- **Hardware and implementation**
 - Block diagram
 - Host interface
 - Decode and control
 - Memory interfaces
 - **DMA channels**
 - Arithmetic unit (ALU)
 - Stored program unit (SPU)

Block diagram



DMA channels

Hydra 4 internally has six channels for transferring data to or from memory

- A read-only *src1* channel
- A read-only *src2* channel
- A write-only *dest* channel
- A “misc” channel which can either read *src3* or write *dest2* (depending on the instruction)
- A special read-only channel for reading vector masks
- A special channel for reading SPU programs and performing SPU loads/stores

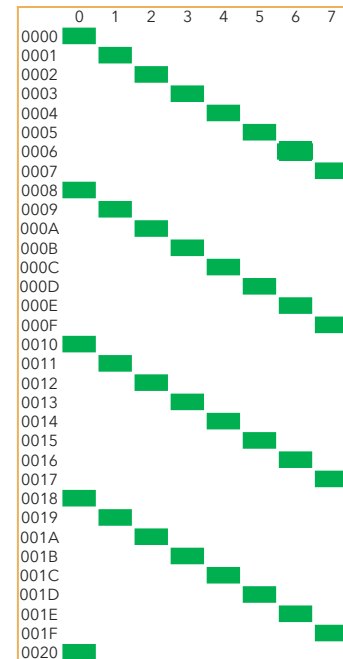
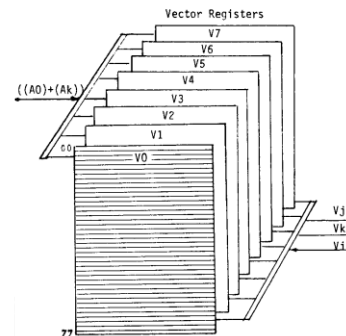
These channels contend with each other for access to the RAM bank interfaces

- Each channel wants to talk to one bank according to its current RAM address
- When several channels all want to use the same bank, Hydra uses a priority rule first, then round-robin arbitration if RAM contention persists

DMA channels

As shown earlier, Hydra (like the Cray-1 and other vector machines) needs to perform multiple parallel reads and writes per cycle to keep its data pipeline busy

- Interleaved banks of RAM allow Hydra to do this automatically, with no user intervention
- Hydra does not require users to manage the location or alignment of vectors in RAM
- Hydra's channel FIFOs make this possible

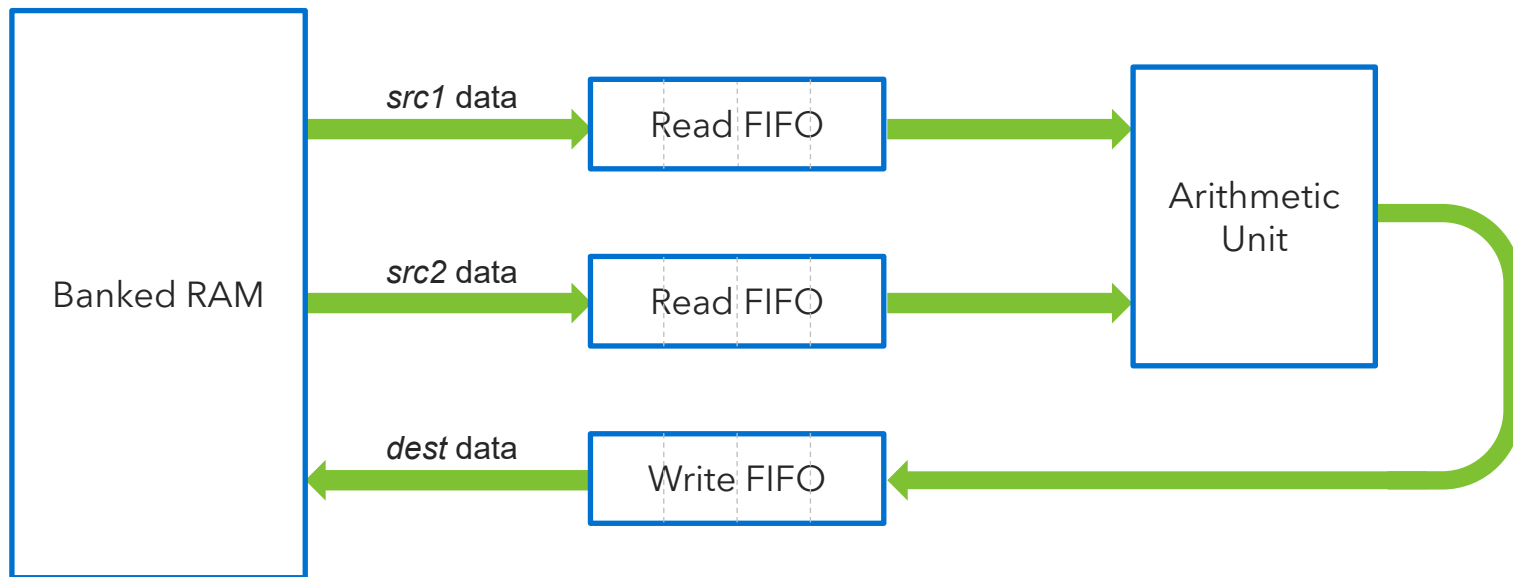


Fetch B	B[0]	B[1]	B[2]	B[3]	B[4]	B[5]	B[6]	B[7]	B[8]	B[9]				
Fetch C		C[0]	C[1]	C[2]	C[3]	C[4]	C[5]	C[6]	C[7]	C[8]	C[9]			
Exec (1)			[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]		
Exec (2)				[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	
Write A					A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]

RAM read/write channels with FIFOs

Hydra's several channels to RAM operate simultaneously and independently

Each RAM channel has a FIFO (first-in/first-out) data buffer



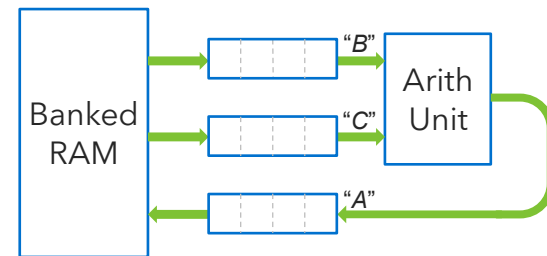
RAM read/write channels with FIFOs

The read and write FIFOs create some “slack” in the pipeline

This makes Hydra extremely tolerant of arbitrary vector alignments

Hydra starts by reading both source vectors simultaneously

- If the arrays’ alignment in RAM is such that $B[i]$ falls in a different bank than $C[i]$ for every i ...
- ... and likewise there is no conflict when writing to $A[i]$...
- ... then everything is simple:

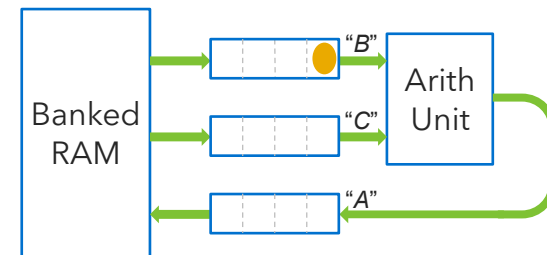


Fetch B	B[0]	B[1]	B[2]	B[3]	B[4]	B[5]	B[6]	B[7]	B[8]	B[9]			
Fetch C	C[0]	C[1]	C[2]	C[3]	C[4]	C[5]	C[6]	C[7]	C[8]	C[9]			
Exec (1)		[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]		
Exec (2)			[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	
Write A				A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]

RAM read/write channels with FIFOs

Suppose $B[i]$ lands in the same bank as $C[i]$, because B and C are similarly aligned in RAM

- Hydra will fetch $B[0]$ on the first cycle ...
- ... but its attempt to fetch $C[0]$ at the same time will stall due to a bank conflict
- Hydra will hold $B[0]$ in its channel FIFO while it tries to read $C[0]$ again on the next cycle
- The "B" RAM channel simultaneously reads $B[1]$, which also succeeds because $B[1]$ and $C[0]$ fall in different banks
 - They must, since $B[1]$ and $B[0]$ are in different interleaved banks while $B[0]$ and $C[0]$ were in the same bank



Fetch B **B[0]** B[1]

Fetch C **stall** → **C[0]**

Exec (1)

Exec (2)

Write A

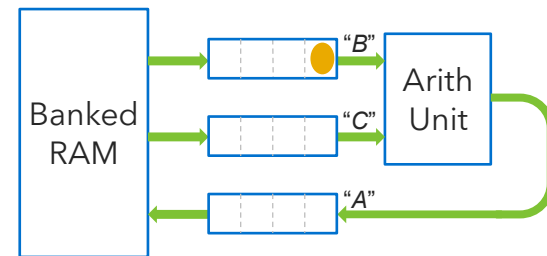
RAM read/write channels with FIFOs

Hydra quickly falls into a pattern fetching $B[i+1]$ together with $C[i]$ on each cycle " i ", free of bank conflicts

- The "B" FIFO always holds one data value:
It pushes and pops one value on every cycle
- The "C" FIFO always passes its data straight through

No special attention is required by the user

Firmware can casually allocate arrays like B and C without micro-managing their location and alignment



Fetch B	B[0]	B[1]	B[2]	B[3]	B[4]	B[5]	B[6]	B[7]	B[8]	B[9]				
Fetch C	stall → C[0]	C[1]	C[2]	C[3]	C[4]	C[5]	C[6]	C[7]	C[8]	C[9]				
Exec (1)		[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]			
Exec (2)			[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]		
Write A					A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]

RAM read/write channels with FIFOs

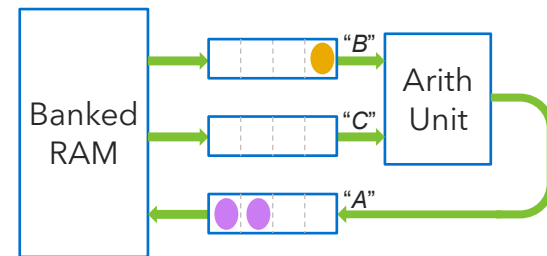
Now suppose $A[0]$ also lands so that it would conflict with the reads of $B[4]$ and $C[4]$

Hydra absorbs two results into its write-channel FIFO as it waits for a chance to write to A

Again, it naturally falls into a pattern where RAM accesses go at full speed with no conflicts

For these particular vector alignments, the steady state is:

- FIFO "B" always holds one value, pushing and popping each time;
- FIFO "C" always passes straight through;
- FIFO "A" always holds two values, pushing and popping each time



Fetch B	B[0]	B[1]	B[2]	B[3]	B[4]	B[5]	B[6]	B[7]	B[8]	B[9]				
Fetch C	stall →	C[0]	C[1]	C[2]	C[3]	C[4]	C[5]	C[6]	C[7]	C[8]	C[9]			
Exec (1)			[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]		
Exec (2)				[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	
Write A					stall →	stall →	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]

RAM read/write channels with FIFOs

Hydra shares access to the RAM with the host CPU and (in some chips) with other DMA peripherals

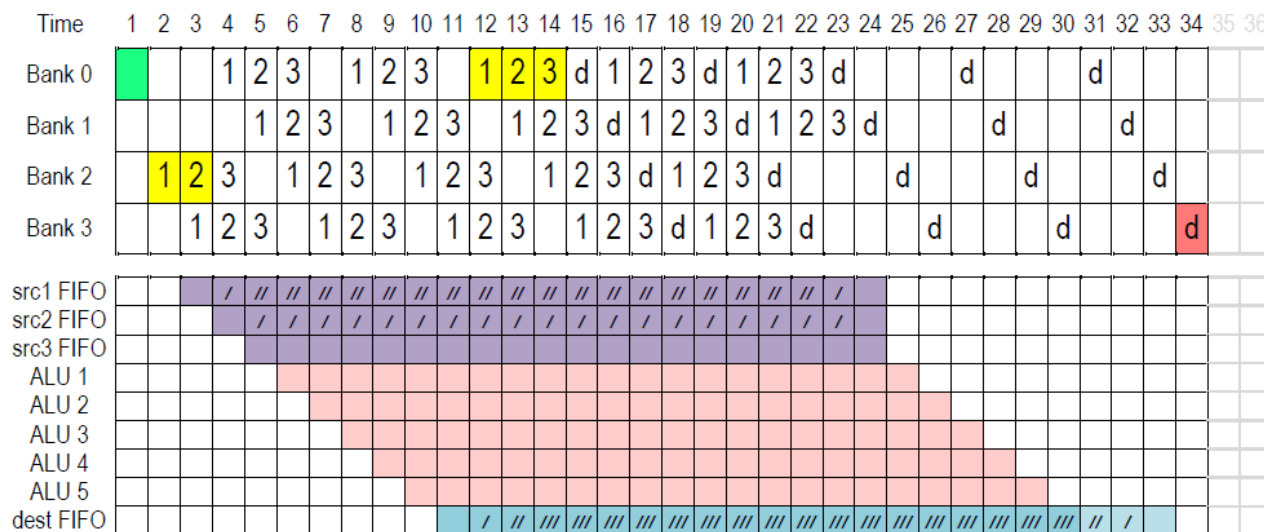
- The host CPU and DMA get top priority, and Hydra's FIFOs soak up the resulting stalls naturally
- ⇒ Each RAM-read channel issues "read" requests to RAM as fast as it can go
- ⇒ The RAM-write channel deposits results to RAM as fast as the RAM can take them
- ⇒ If a FIFO ever fills to capacity (~4 items), the pipeline stalls upstream to relieve "back pressure"
- ⇒ If a FIFO ever runs empty, the pipeline stalls downstream until more data is available

Fetch B	B[0]	B[1]	B[2]	B[3]	B[4]	B[5]	B[6]	B[7]	stall → B[8]	B[9]				
Fetch C	stall → C[0]	C[1]	C[2]	C[3]	C[4]	stall → C[5]	C[6]	C[7]	C[8]	C[9]				
Exec (1)		[0]	[1]	[2]	[3]	[4]	—	[5]	[6]	[7]	[8]	[9]		
Exec (2)			[0]	[1]	[2]	[3]	[4]	—	[5]	[6]	[7]	[8]	[9]	
Write A				A[0]	A[1]	A[2]	A[3]	A[4]	—	A[5]	A[6]	A[7]	stall → A[8]	A[9]

Bank contention for linear vectors

This timing diagram (from actual simulation results, as shown in the *Hydra User's Guide*) shows a VMADD instruction with four linear (stride-1) vectors

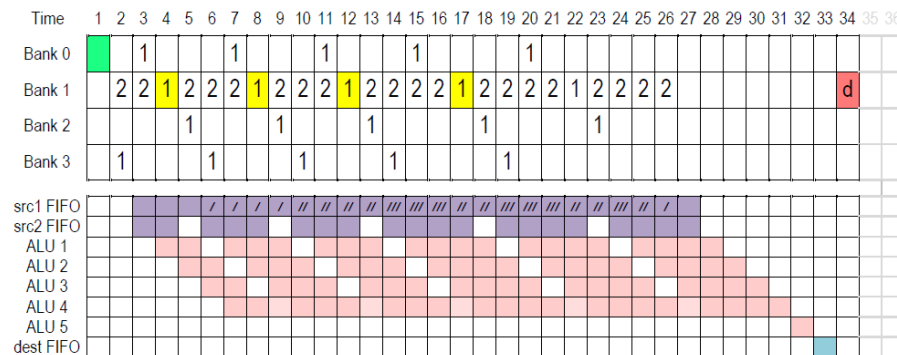
- The vector base addresses have been chosen to create worst-case bank contention
- All three source vectors start in bank 2, causing two bank-conflict stalls at the start (in yellow)
- The *dest* vector starts in bank 0, which happens to be busy serving *src1*, then *src2*, then *src3* at time 12 when the first result emerges from the pipeline: This causes three more stalls
- Once the FIFOs have absorbed the stalls, operation continues at full speed



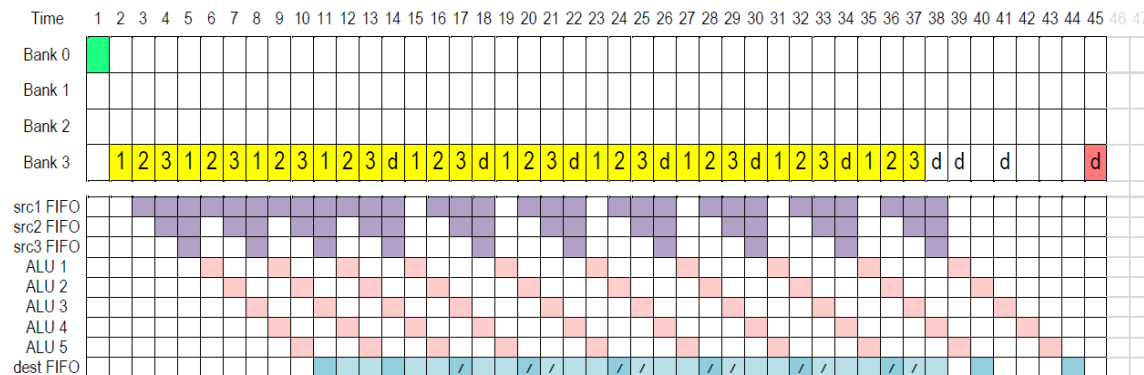
Bank contention for strided vectors

When a vector's stride is a multiple of 4 (actually, a multiple of the number of RAM banks), every element in it will fall in the same bank

- This can slow Hydra noticeably even if the other vectors are linear $\Rightarrow$



- And it can slow down dramatically if the vectors are aligned the same and all are stride-4! $\Rightarrow$



Channels: Other functions

The channel's sequencer counts cycles to determine when the channel is done

- ... and it counts the nested sub-cycles of matrices and convolutions

The address generator produces a sequence of RAM addresses for the selected addressing mode

- For linear mode and stride mode: Simple addition
- For matrix addressing: Add either the row stride or the column stride
- Special formulas for FFT addressing, matrix multiply, convolutions, etc.

Sketch of address generation for *src1* $\Rightarrow$ in 2D convolutions

for all $0 \leq i < R$:

for all $0 \leq j < C$:

$MVdest[i, j] \leftarrow accum(MVsrc1[i + u, j + v], MVsrc2[u, v] \text{ for } 0 \leq u < U \text{ and } 0 \leq v < V)$

```
outer_row_addr = base_addr;
for (count = 0; count < length; ) {
    outer_addr = outer_row_addr;
    for (column_count = 0; column_count < outer_width; ) {
        kernel_row_addr = outer_addr;
        count++, column_count++;
        for (kernel_count = 0; kernel_count < kernel_length; ) {
            cur_addr = kernel_row_addr;
            for (kernel_column_count = 0; kernel_column_count < kernel_width; ) {
                advance();
                kernel_count++, kernel_column_count++;
                if (kernel_count < kernel_length) {
                    if (kernel_column_count < kernel_width)
                        cur_addr += column_stride;
                    else
                        kernel_row_addr += row_stride;
                } else {
                    if (column_count < outer_width)
                        outer_addr += column_stride;
                    else
                        outer_row_addr += row_stride;
                }
            }
        }
    }
}
```

Channels: Other functions

Also...

A source unpacker or dest packer for packed addressing mode

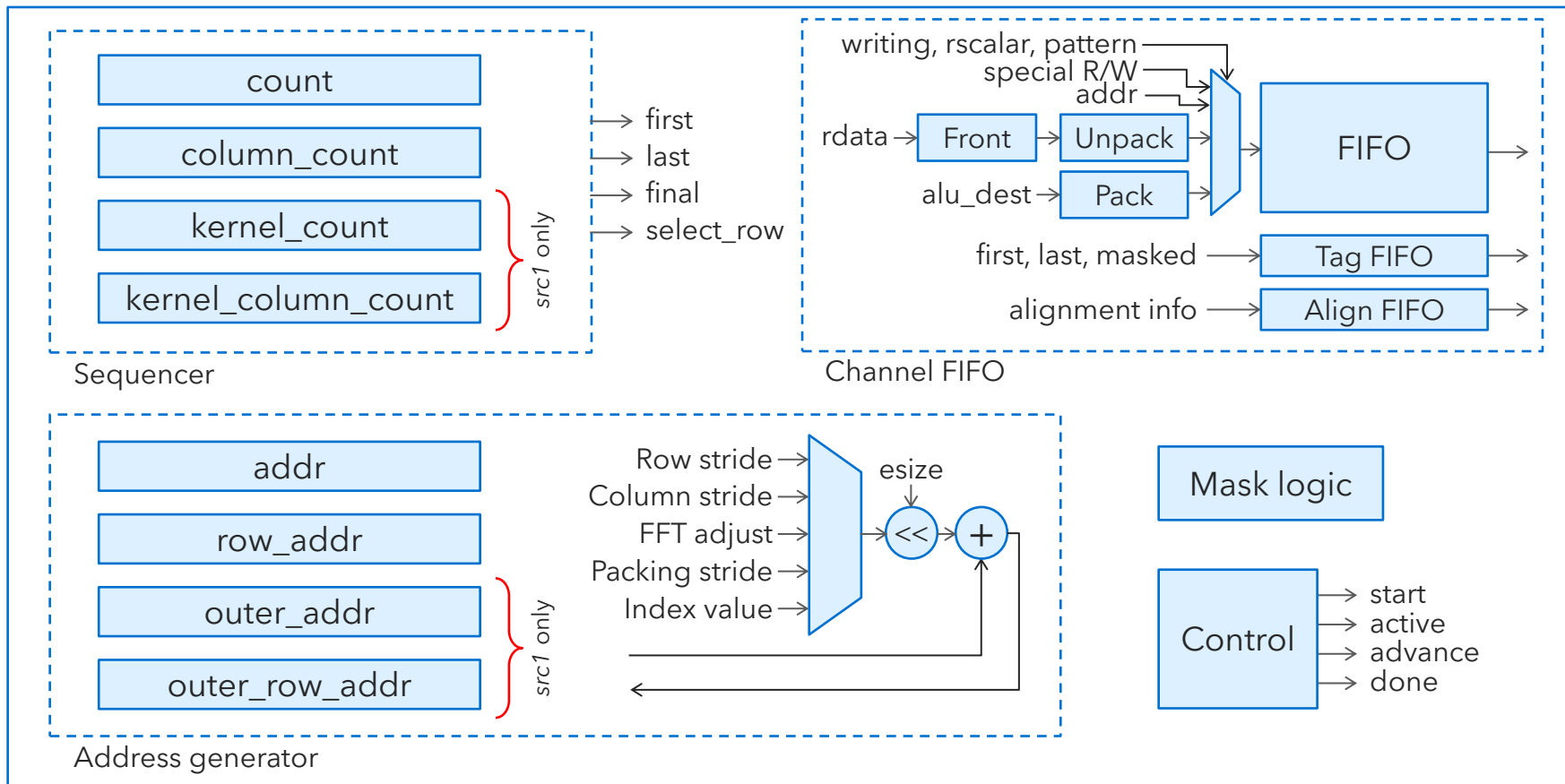
- It also realigns 16- or 8-bit data into the appropriate lanes of memory wdata/rdata
- And merges adjacent byte/short reads or writes into 32-bit accesses when possible
- And it is used for instructions that generate a vector mask

Logic to handle incoming vector masks

- With a parallel FIFO for “mask” bits and a few other control bits
- The channel suppresses the RAM read or RAM write for any masked-out vector element

Extra pipeline stages in several places to help meet 400 MHz timing

DMA channel (each of *src1*, *src2*, *dest*, *misc*)



Vector mask channel

There is a specialized, read-only channel for vector masks

It reads 32-bit words from the vector mask and holds the most recent two words

- This is enough to cover the current mask bit of every active *src* and *dest* channel
- It sends the relevant mask bits to all the channels depending on each channel's current count

Eventually, Hydra 4.x will support “vector mask optimization”

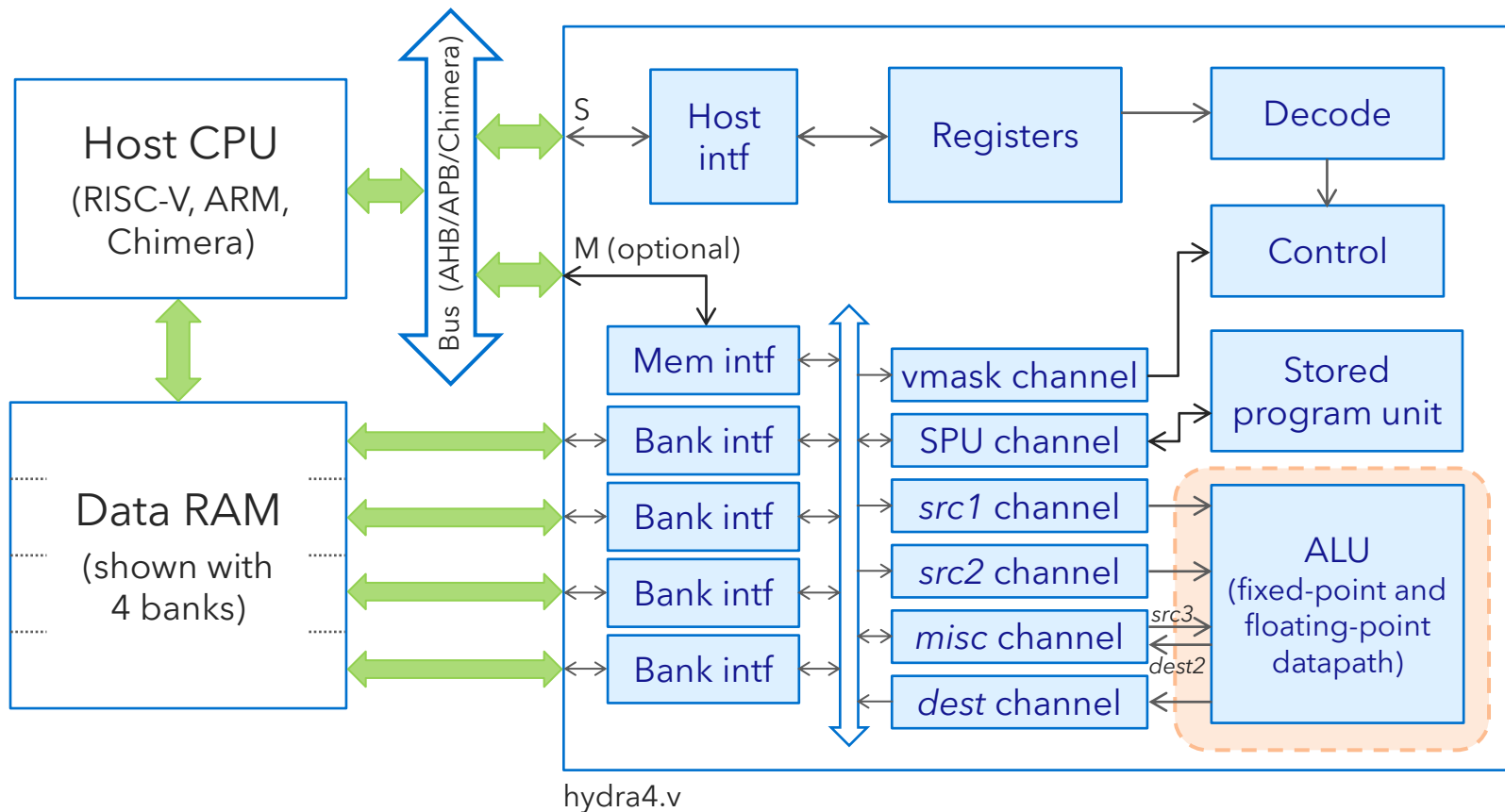
- Each channel can see the next *several* upcoming mask bits
- The channel can skip multiple iterations when conditions are right
- If the next 2^K mask bits are visible and they are all '0's, then linear or strided addressing can skip through 2^K iterations at once: Adjust the address by $(stride \ll K)$ instead of by $(stride)$
- If all the channels are able to skip in this way, Hydra can run quickly on “sparse” masked vectors
- Hydra 3's design made this impossible, but Hydra 4 is architected to allow it
- This feature is defined in the *Hydra 4 User's Guide*, but not yet implemented in Hydra 4.0–4.2

Hydra 4

Architecture overview

- Introduction
- Using Hydra
- **Hardware and implementation**
 - Block diagram
 - Host interface
 - Decode and control
 - Memory interfaces
 - DMA channels
 - **Arithmetic unit (ALU)**
 - Stored program unit (SPU)

Block diagram



Arithmetic unit

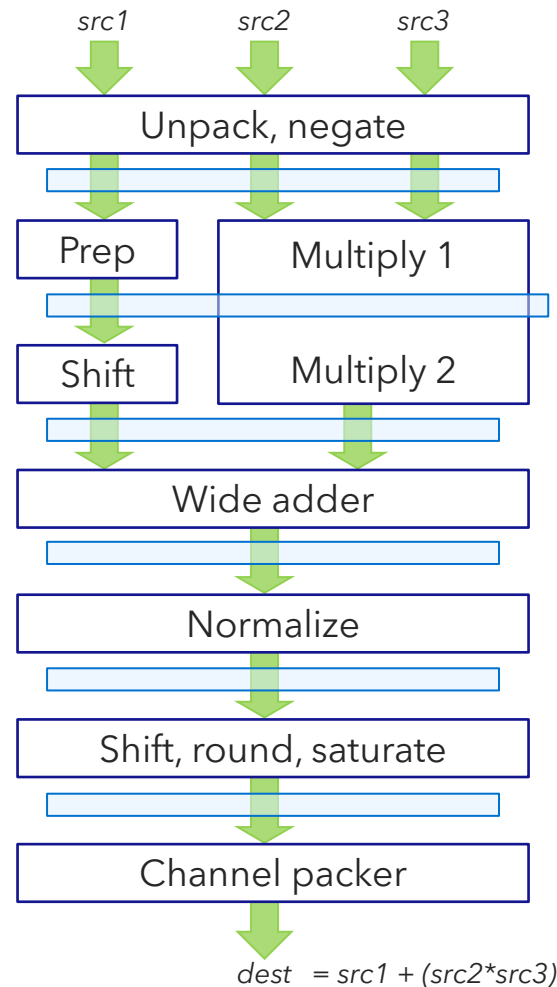
Hydra 4 has a single arithmetic pipeline for all data types (integer, fixed, float, Posit)

Hydra 4's pipeline has many more stages than Hydra 3's

- But actually, Hydra 3's float pipeline ran at half speed ...
- ... so it took four hclk's for a multiply-add
- ... and Hydra 4 is designed to clock significantly faster than before
- SiFive's E76 CPU has 5-cycle latency, too, for floating-point operations

It's a fact: A chip this fast needs a deep pipeline!

- The long pipeline does not hurt throughput, but it affects instruction startup time



Arithmetic pipeline

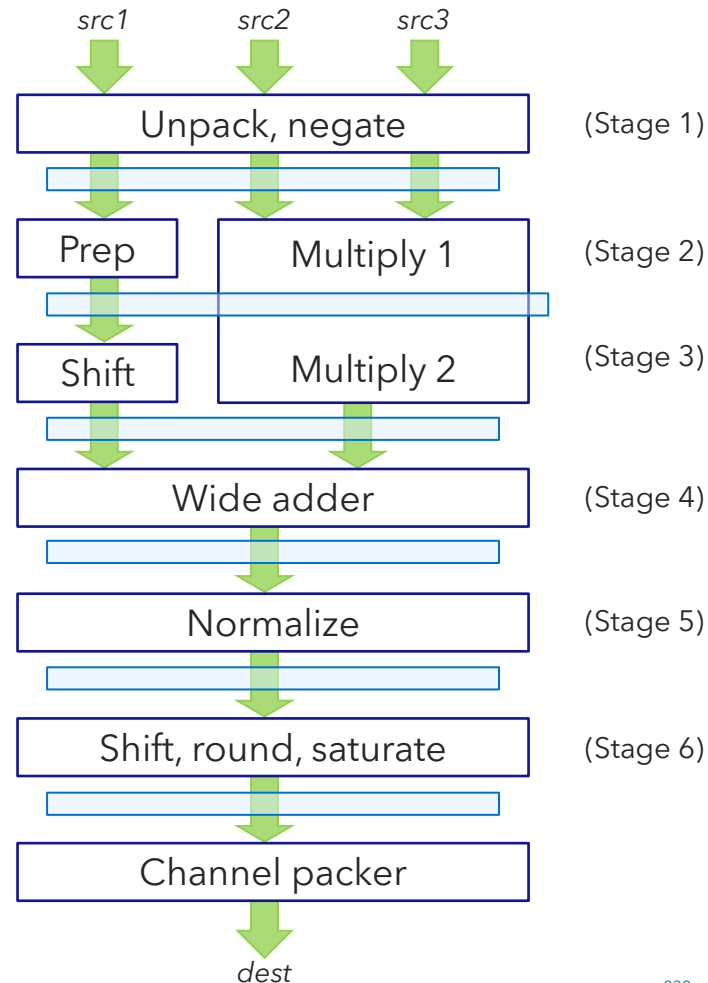
The ALU pipeline advances once per hclk cycle

It consumes one value from each source channel (*src1*, *src2*, *src3*) whenever all three channels have a value available in their channel FIFOs

- Otherwise (e.g., if RAM bank contention slows one *src*), a “bubble” passes through the pipeline for that cycle

seq_state	stage2	stage3	stage4	stage5	stage6
BEGIN	A, R				
NEXT	B, S	A*R			
NEXT	C, T	B*S	AR		
NEXT	-,-	C*T	BS	AR	
WAIT	D, U	-	CT	BS	AR (to dest)
NEXT	E, V	D*U	-	CT	BS (to dest)
NEXT	-,-	E*V	DU	-	CT (to dest)
WAIT	-,-	-	EV	DU	-
WAIT	F, W	-	-	EV	DU (to dest)
NEXT	G, X	F*W	-	-	EV (to dest)
NEXT	H, Y	G*X	FW	-	-
DONE	-,-	H*Y	GX	FW	-
DONE	-,-	-	HY	GX	FW (to dest)
DONE	-,-	-	-	HY	GX (to dest)
DONE	-,-	-	-	-	HY (to dest)
DONE	-,-	-	-	-	-

VMUL on 8-element vectors with 3 stall cycles



Arithmetic pipeline

When live data is in the output stage, it goes to the *dest* channel

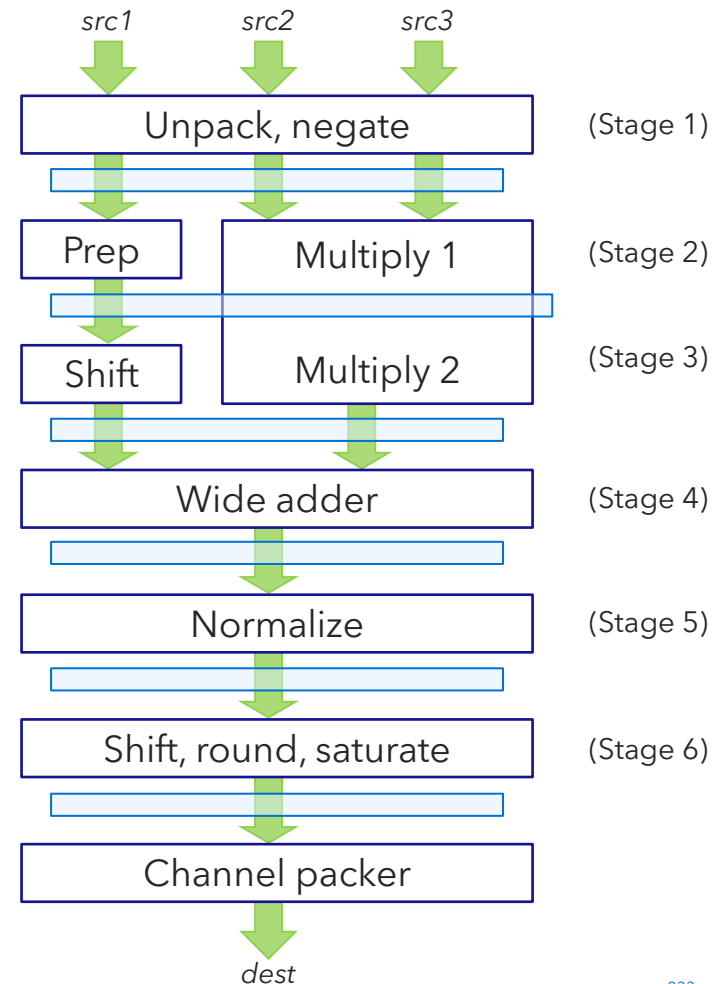
- Or the ALU pipeline stalls if the *dest* channel is not ready to accept data (*dest* FIFO is full)

For simple vector operations, data flows through at a rate of one multiply-add calculation per hclk

- (But with several hclk's of latency)

Some instructions (such as VCRC) have different execution patterns, controlled by a simple state machine in the ALU

- For simplicity, Hydra 4.0–4.2 uses the same six-stage pipeline structure for all instructions



Numbers

Hydra 4's ALU operates internally on a special internal float-like format

- A 9-bit integer exponent E and a 33-bit integer significand F
- The numeric value is $2^E * F$
- Both components are signed integers in two's complement notation
- Plus a few extra bits to encode the special values ± 0 , $\pm \infty$, and NaN
- This format can express all numbers that are representable in all of Hydra's data types

The ALU datapath converts all inputs to this form in its input stage

- Thus, Hydra instructions can have any mixture of data types: E.g., $\text{float32} \leftarrow \text{int32} * \text{float16}$

The datapath converts results back to the desired data type in its output stage

Internal number format

Format	Exponent (9 bits 2s-comp)	Significand (33 bits 2s-comp)	Notes
float32	Add bias	Signed fraction	To convert to internal format, negate if sign=1 and insert an explicit leading '1' if not denormal
bfloat16	Add bias	Signed fraction	
float16	Add bias	Signed fraction	To convert from this format, negate if negative, round, and normalize
int	0	Sign-extend	Significand can be treated as a 33-bit signed integer
unsigned	0	Zero-extend	
fixed SQ.k	-k	Sign-extend	
fixed UQ.k	-k	Zero-extend	
posit	Variable size	Sign-extend	Special conversion rules

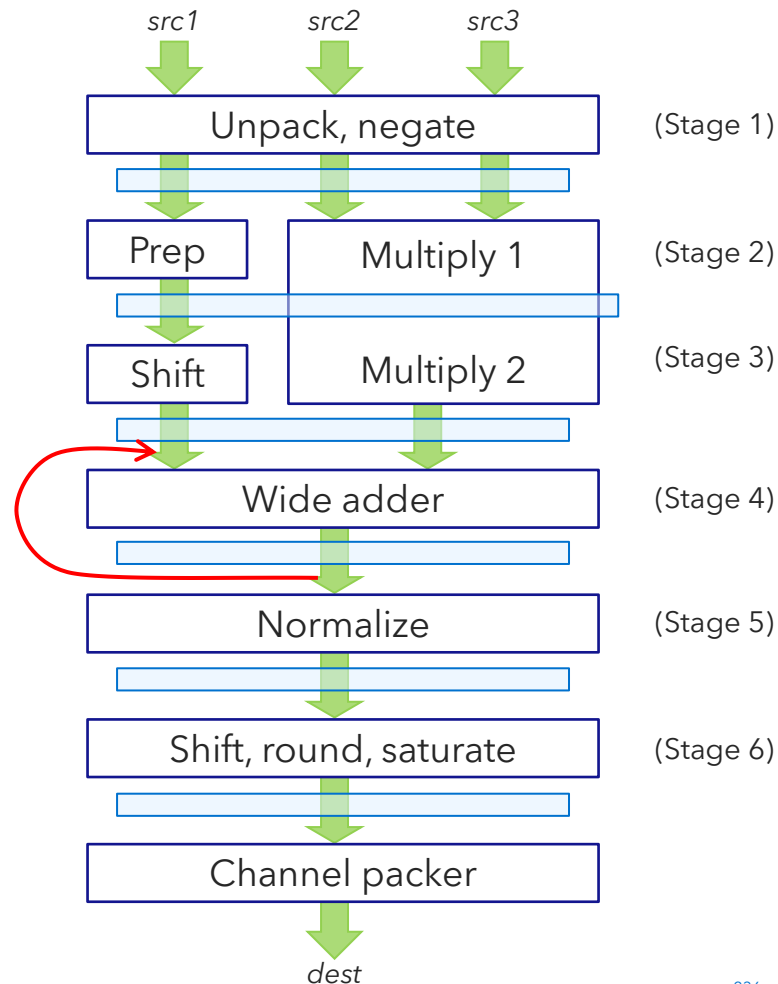
Multiply-accumulate

The VMAC instruction computes a sum of products

Challenge: How to accumulate efficiently when the latency through the pipeline is many cycles?

When (*src1*, *src2*, *src3*) are all integers or fixed-point numbers:

- The addition step is a simple integer add, which can be done with one-cycle latency
- A 96-bit accumulator calculates the mathematically exact sum (up to 2^{31} products of pairs of 33-bit signed numbers), giving a precise and deterministic result



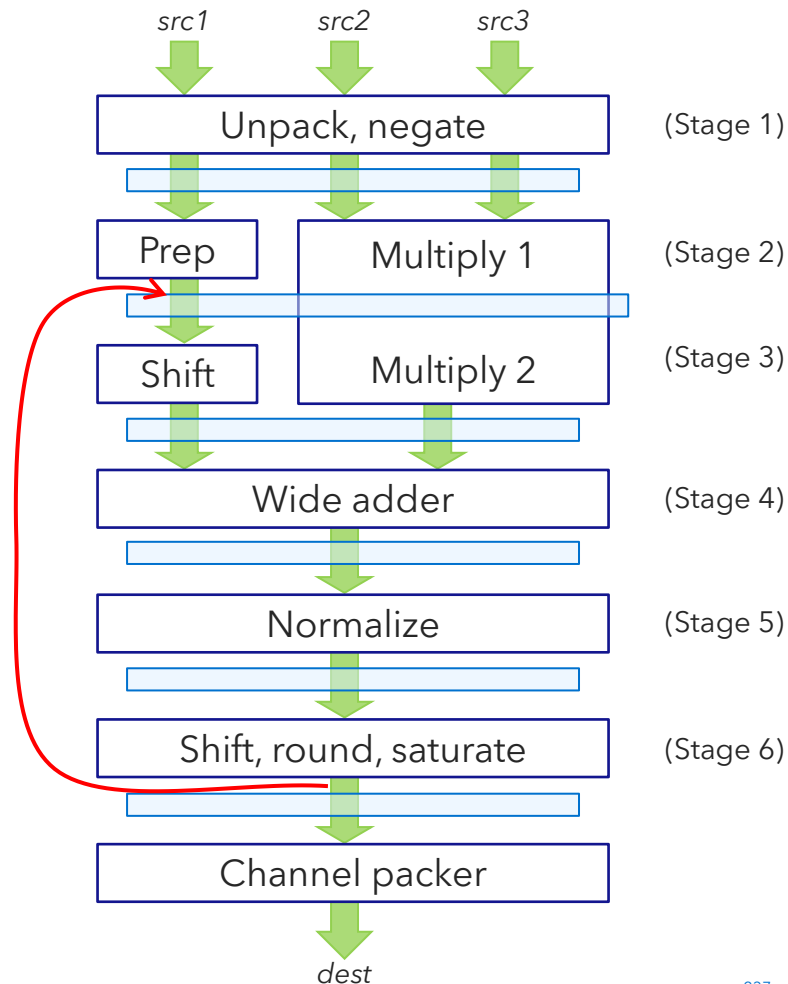
Multiply-accumulate

When floating-point inputs are involved:

- The natural loop latency around the “addend” side of the ALU is four cycles
- Hydra 4 offers several flavors of MAC instruction to deal with this

VOMAC (“ordered accumulate”):

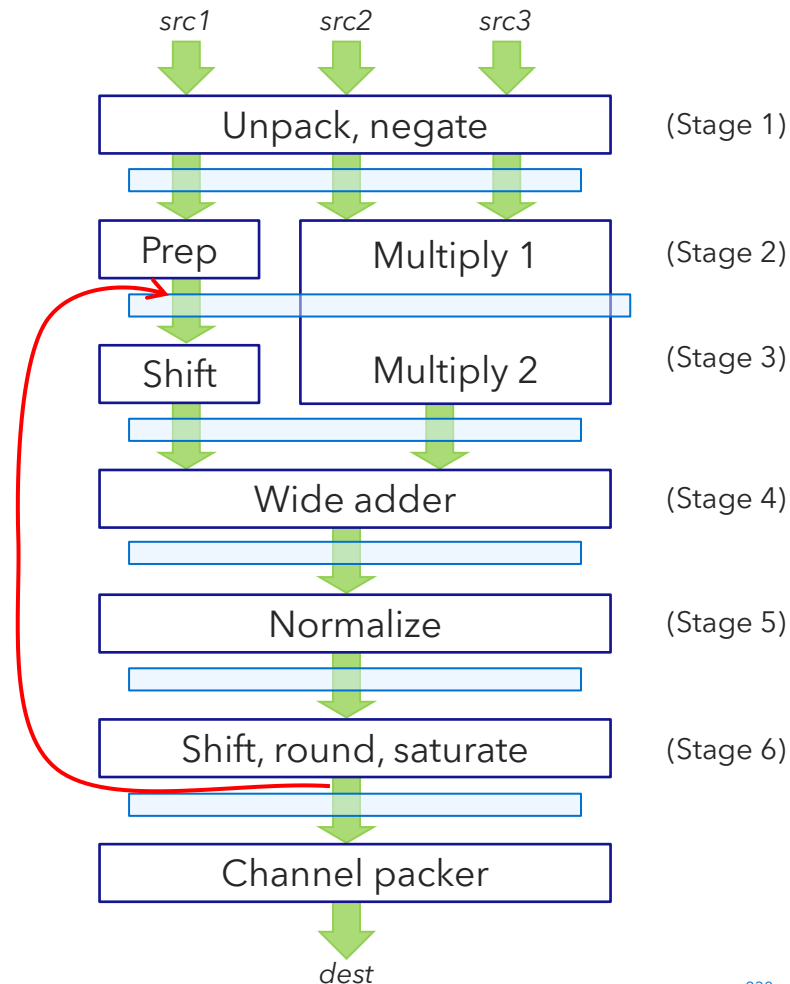
- Emulates a simple “for” loop iterating over the vector elements from left to right
- It really does take 4 cycles per iteration!
- VOMAC runs 4x slower than normal vector instructions
- Hydra provides the VOMAC instruction for completeness, but it is not satisfying and will rarely be used in practice



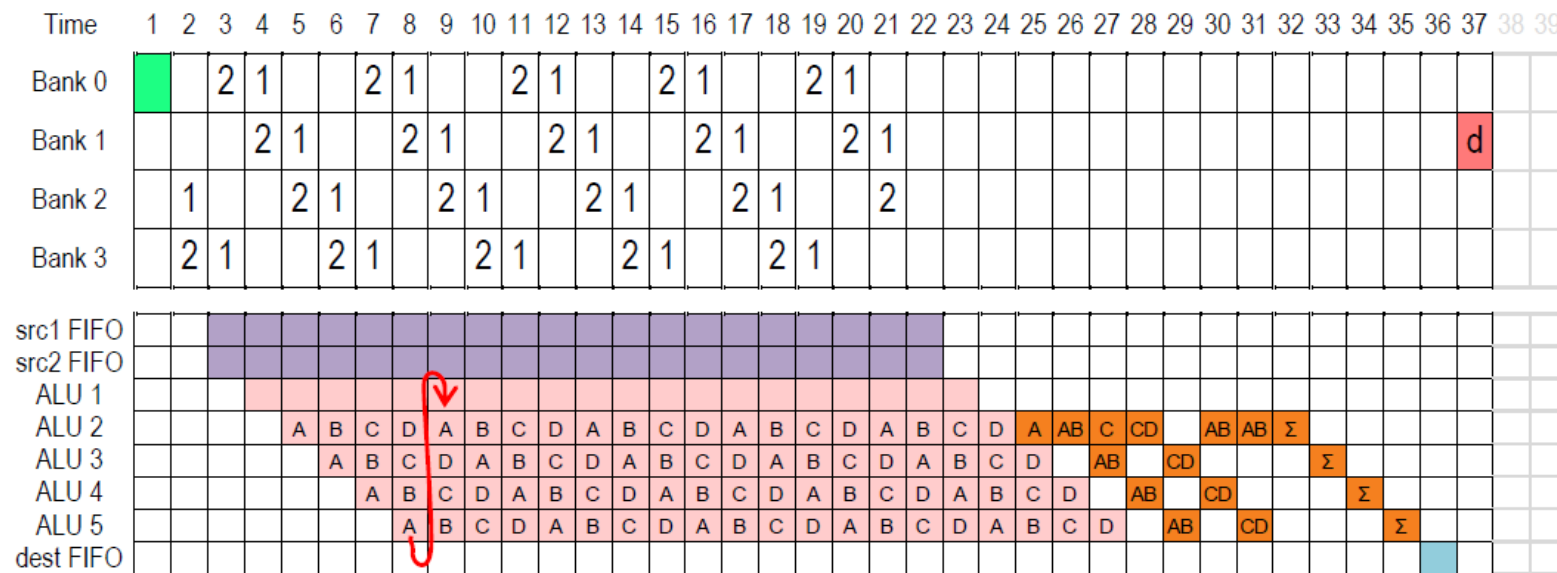
Multiply-accumulate

VDMAC (“deterministic accumulate”):

- Runs at full speed
- Still recirculates the ALU with 4-cycle latency
- Result: Four partial sums will circulate around inside the ALU, each one summing 1/4 of all the elements
- At the end, VDMAC runs a few irregular cycles to collect and sum together the four partial sums
- Orchestrated by a simple state machine in the ALU’s control logic

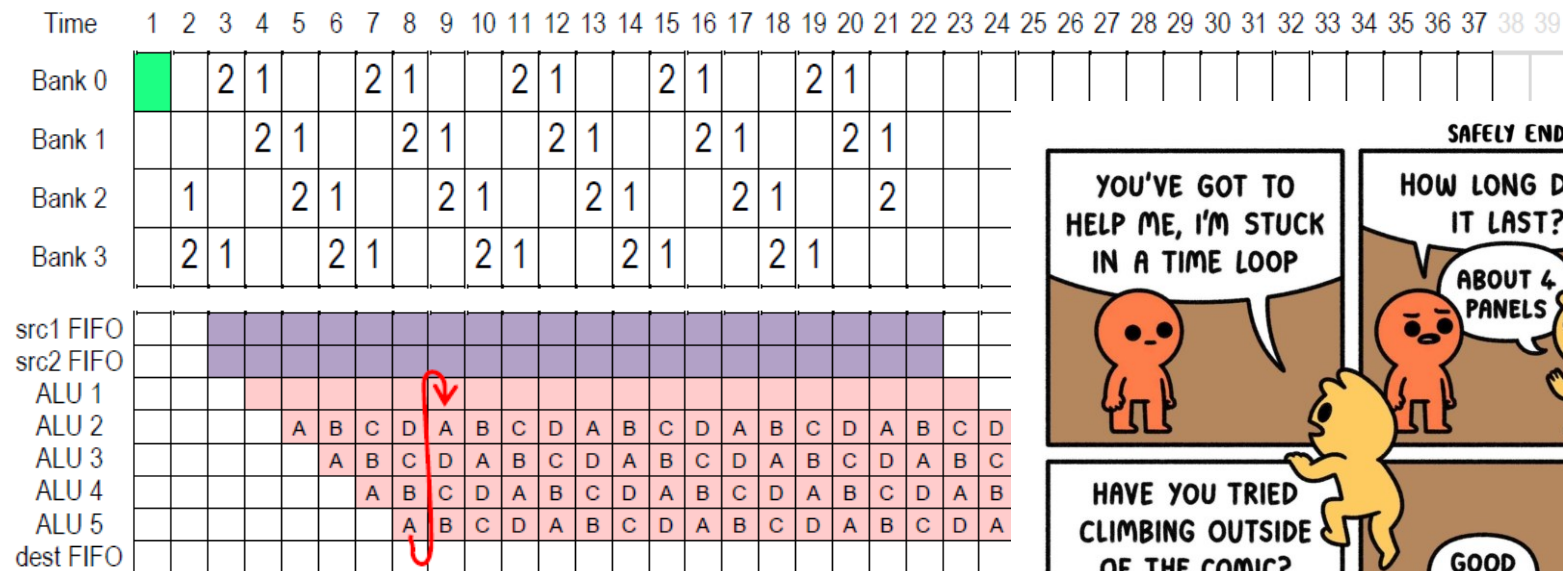


Multiply-accumulate: VDMAC sequencing



$$AB = A + B; \quad CD = C + D; \quad \Sigma = A + B + C + D$$

Multiply-accumulate: VDMAC sequencing



Multiply-accumulate: Rounding error

VDMAC will not give *quite* the same result as VOMAC

Their floating-point rounding errors will not match:

- Floating-point addition does not obey the associative law of “true” addition
- For example, “ $1.0 + (10^{-10}) - 1.0$ ” gives 0.0 instead of 10^{-10} when evaluated naturally (from left to right) using single-precision floats
- 10^{-10} is so small, it rounds away to nothing when added to 1.0 in single-precision
- Therefore, the calculation gets a very different answer than “ $1.0 - 1.0 + (10^{-10})$ ”, even though they are *mathematically* equivalent!

However, VDMAC’s result can be precisely reproduced by a reasonably simple and deterministic firmware loop

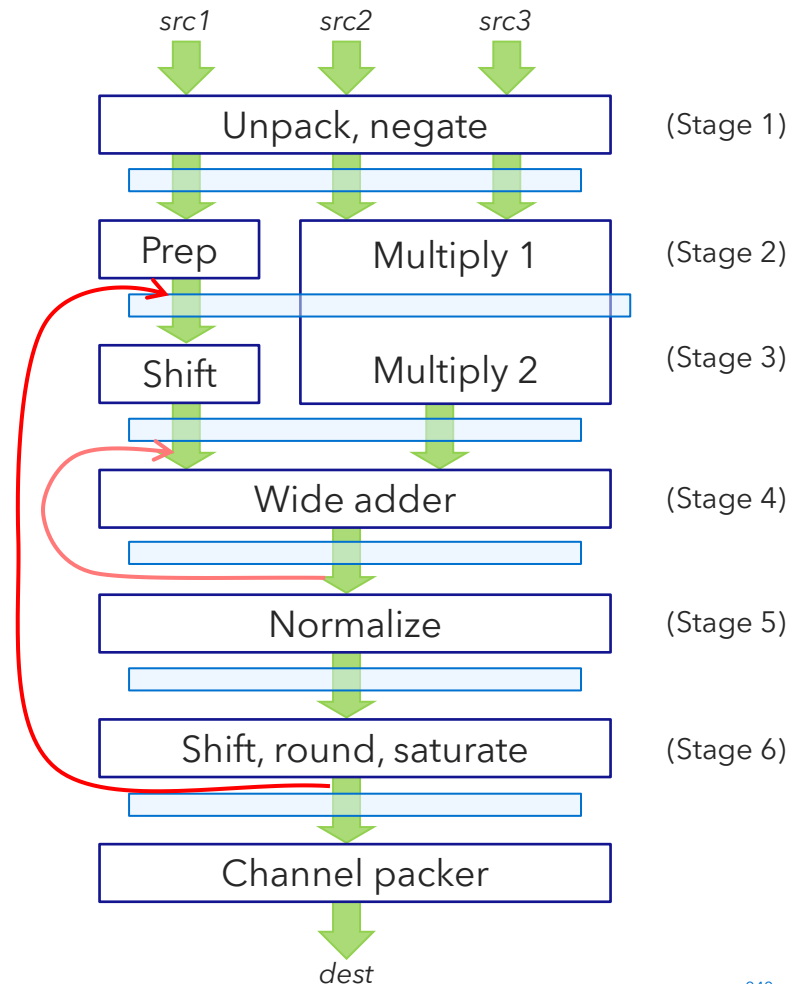
- Hence the “D” in “VDMAC”

```
float partial_sum[4] = { 0.0, 0.0, 0.0, 0.0 };
for (int i = 0; i < N; i++)
    partial_sum[i % 4] += vec[i];    // Or fmaf()
partial_sum[0] += partial_sum[1];
partial_sum[2] += partial_sum[3];
sum = partial_sum[0] + partial_sum[2];
```

Multiply-accumulate

VMAC ("fast accumulate"):

- Most ordinary multiply-accumulations should use the plain "VMAC" instruction
- Runs at full speed
- For fixed-point operands, it runs as shown earlier
- For floating-point operands, VMAC runs like VDMAC but (potentially) with higher precision accumulations that give better results ...
- ... but would be hard to reproduce exactly in host CPU firmware
- (Note that higher-precision float VMAC is not yet implemented; in Hydra 4.0-4.2, VMAC on floats currently works exactly the same as VDMAC)



ALU: Other operations

The *Hydra 4 User's Guide* defines a wide variety of operations on vectors:

- Bitwise operations
- Various flavors of MIN / MAX
- Hamming distance
- Cryptographic functions, etc.
- Find, compress, expand, gather, scatter, ...
- Fast medians
- Integer and float division in all rounding modes
- Fast SIMD treatment of 16-bit integer arithmetic
- Optimized treatment of sparse vector masks
- Padding for convolutions
- Instruction joining

The ones in **red** were not implemented in Hydra 4.0–4.2 on SmartBridge and Byrd

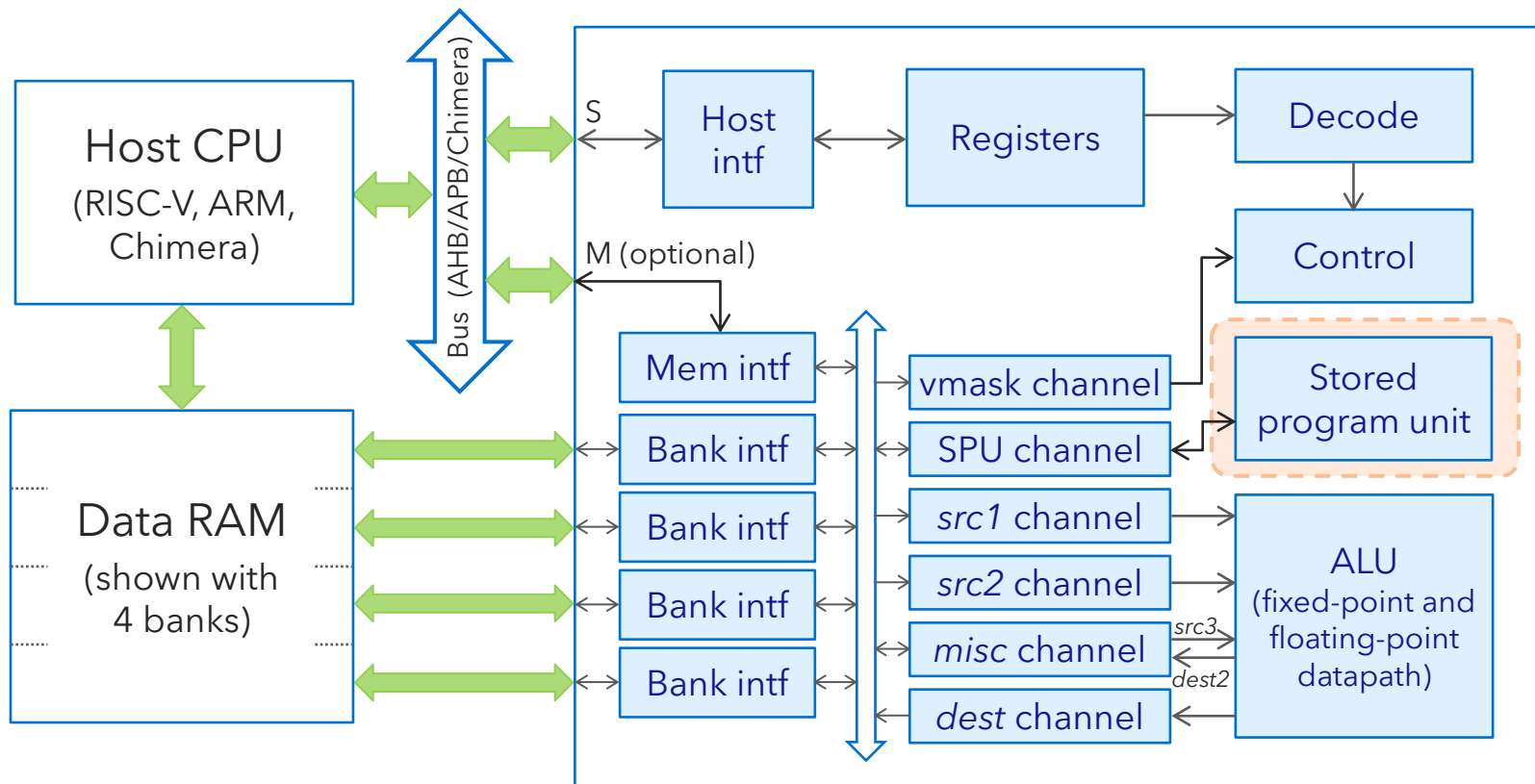
- Future chips can include these and other improvements in Hydra 4.3 or later

Hydra 4

Architecture overview

- Introduction
- Using Hydra
- **Hardware and implementation**
 - Block diagram
 - Host interface
 - Decode and control
 - Memory interfaces
 - DMA channels
 - Arithmetic unit (ALU)
 - **Stored program unit (SPU)**

Block diagram



hydra4.v

Stored program unit

The stored program unit is designed to be simple

It has a simple channel to read SPU instructions

- Normally it reads consecutive words into a FIFO buffer
- The decoder pops words from the FIFO and executes them
- A branch instruction flushes the FIFO and restarts the channel reading from a new address
- Load/store instructions stop the FIFO, run a single read or write, then restart the channel as if upon a branch

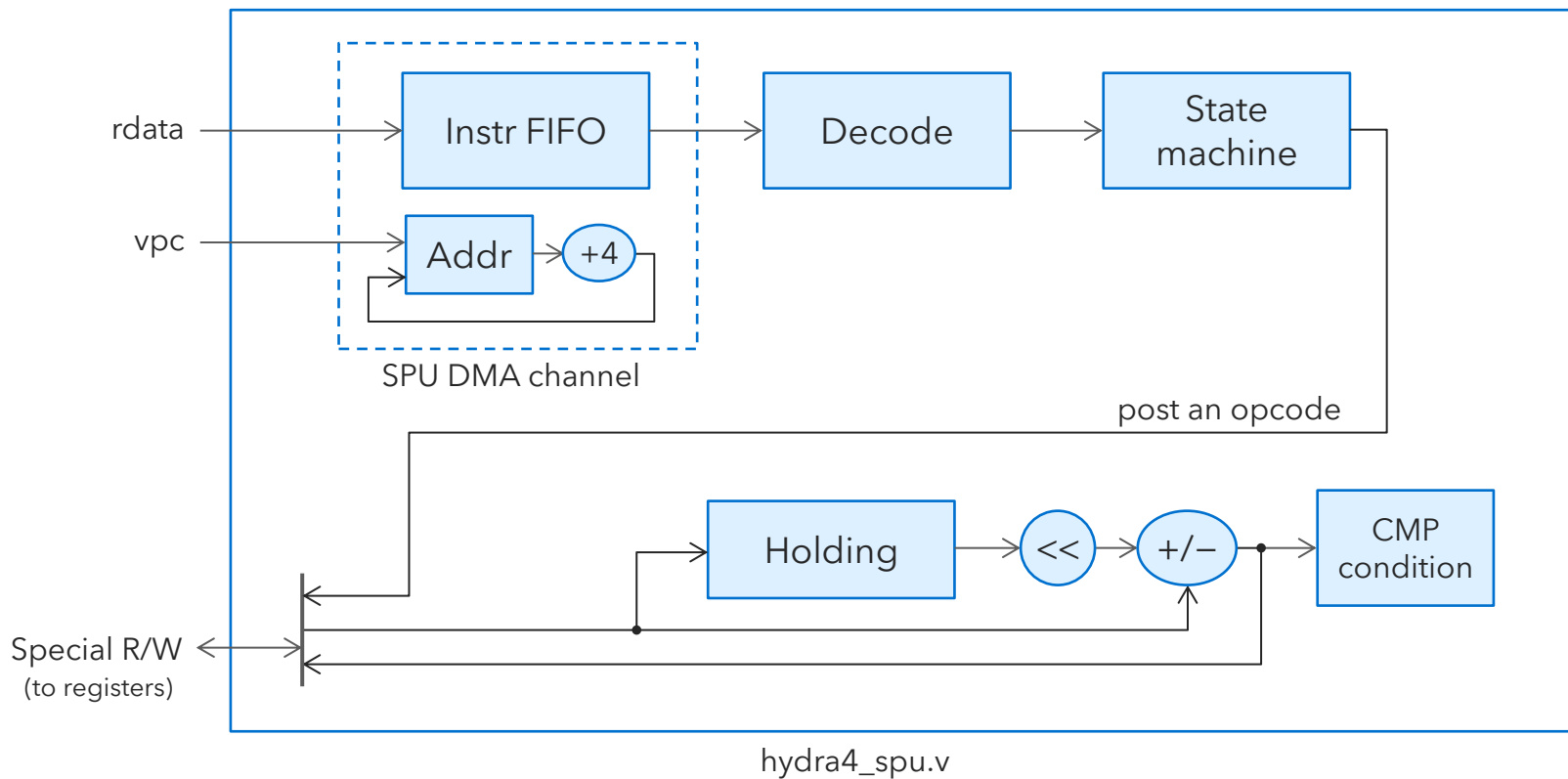
The SPU can read or write one Hydra register on each cycle using a “special R/W” interface to the registers block

SPU instructions that do more than one read or write are sequenced into several cycles

- The SPU is *not* meant to be a high-performance CPU!



Stored program unit



Thank You