

Computer Architectures Should Go Brrrrrrr

August 28, 2024

Eric Quinnell, Ph.D.

Tesla AI Hardware

@divBy_zero

Topics and Caveats

Topics

- **Intro** – Architectures should go brrrrrr
- **RISC-V RVC** – critique of variable length instructions
- **RISC-V RVV** – critique of uarch feasibility
- **SPEC vs JIT architectures** – P6 vs Firestorm
- **Q&A**

Caveats

- I do not speak on behalf of Tesla. My opinions are my own and are expressed merely to educate new engineers on why they're wrong.
- My critiques are my attempt to contribute to errors I see in RISC-V ISA development. Dojo uses a custom set of RISC-V, I want this to succeed.
- All content presented here is public domain. Any speculation on CPU reverse-engineering is mine alone, but probably right

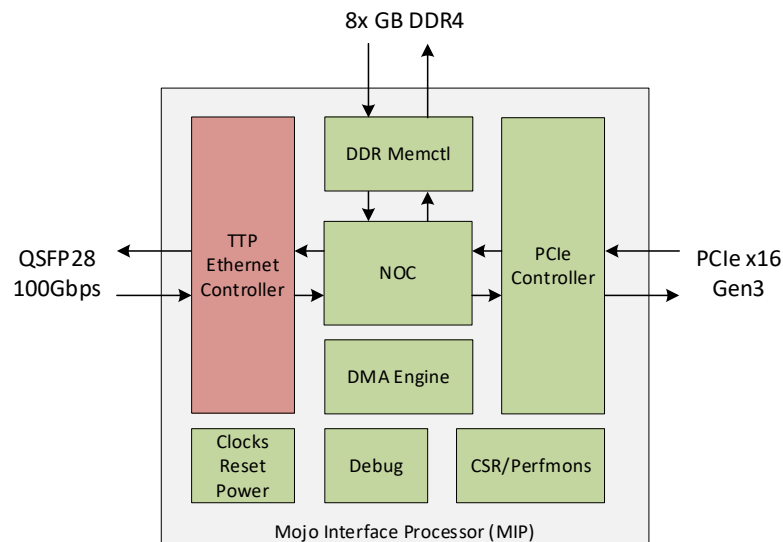
The “dumber” design is the smarter design.

Ik intends to highlight items I see as overly c

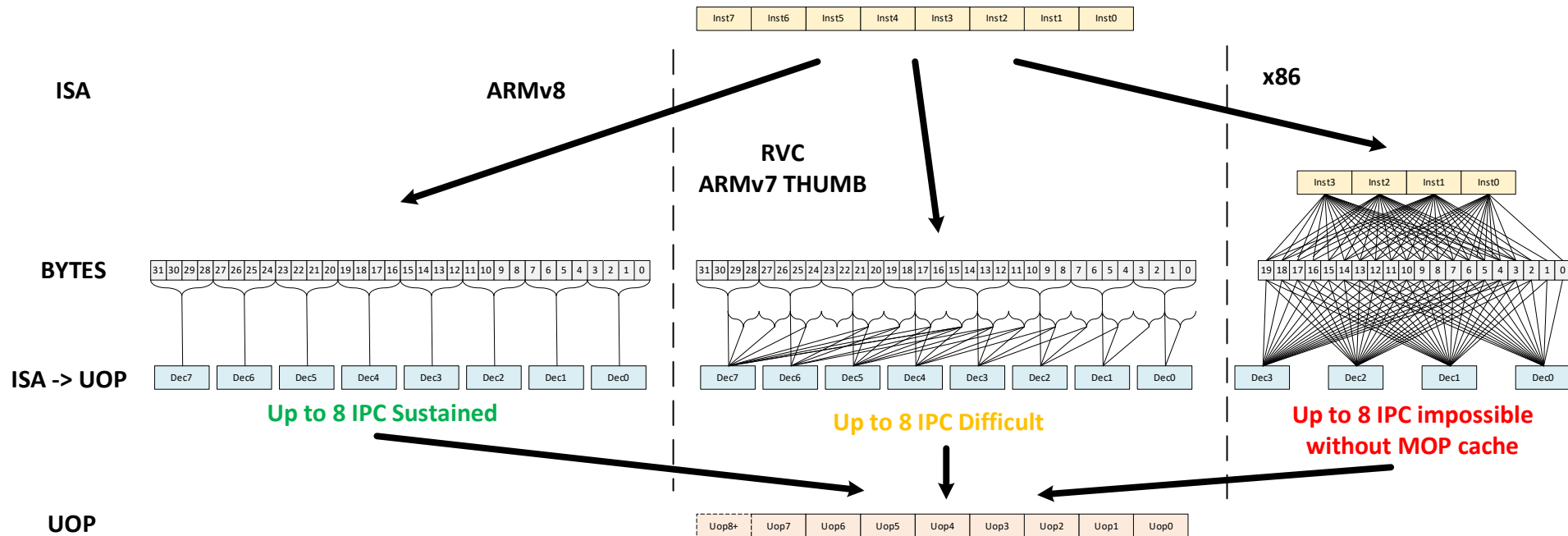
“Mojo” 100Gbps Dumb-NIC

From my presentation at yesterday's HotChips 2024. “DumbNIC”. I practice what I preach.

Feature	Spec
Ethernet Speed	100Gbps QSFP
PCI-e	Gen3 x16
Memory	8GB DDR4
Power	<20W max
Reliability	5-year tested
DMA engine	Dojo DMA
CPU+OS	None
Active Links	512 unique, 2-way, LRU



RISC-V RVC – The cost of variable length decode



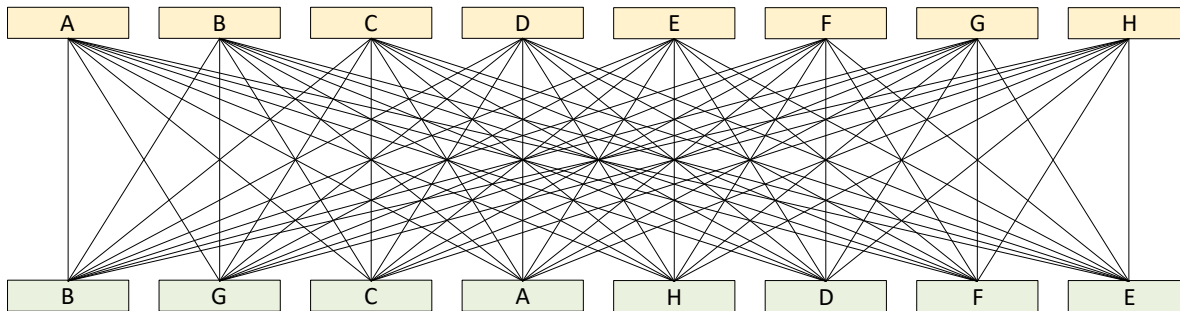
Permutes – Destroyer of Chips

```

for (i=0; i<7; i++) begin
  casez(sel[i][2:0]) begin
    3'b000 : out[i] = A;
    3'b001 : out[i] = B;
    3'b010 : out[i] = C;
    3'b011 : out[i] = D;
    3'b100 : out[i] = E;
    3'b101 : out[i] = F;
    3'b110 : out[i] = G;
    3'b111 : out[i] = H;
    default : out[i] = aint_gonna_make_timing;
  endcase
end

```

- It's not the FLOPs, it's the connections limiting your performance.
- N-to-N element selection is conceptually easy and tremendously difficult to implement well.
- Modern nodes are "wire dominant" -- i.e. the reduction of geometric sizes provides less cross-sectional area for each wire, increasing resistivity.
- e.g. x86's VPERMPS ARM's VTBL/VTRN/VEXT/VZIP AI/NN layer graphs, **var length decode**



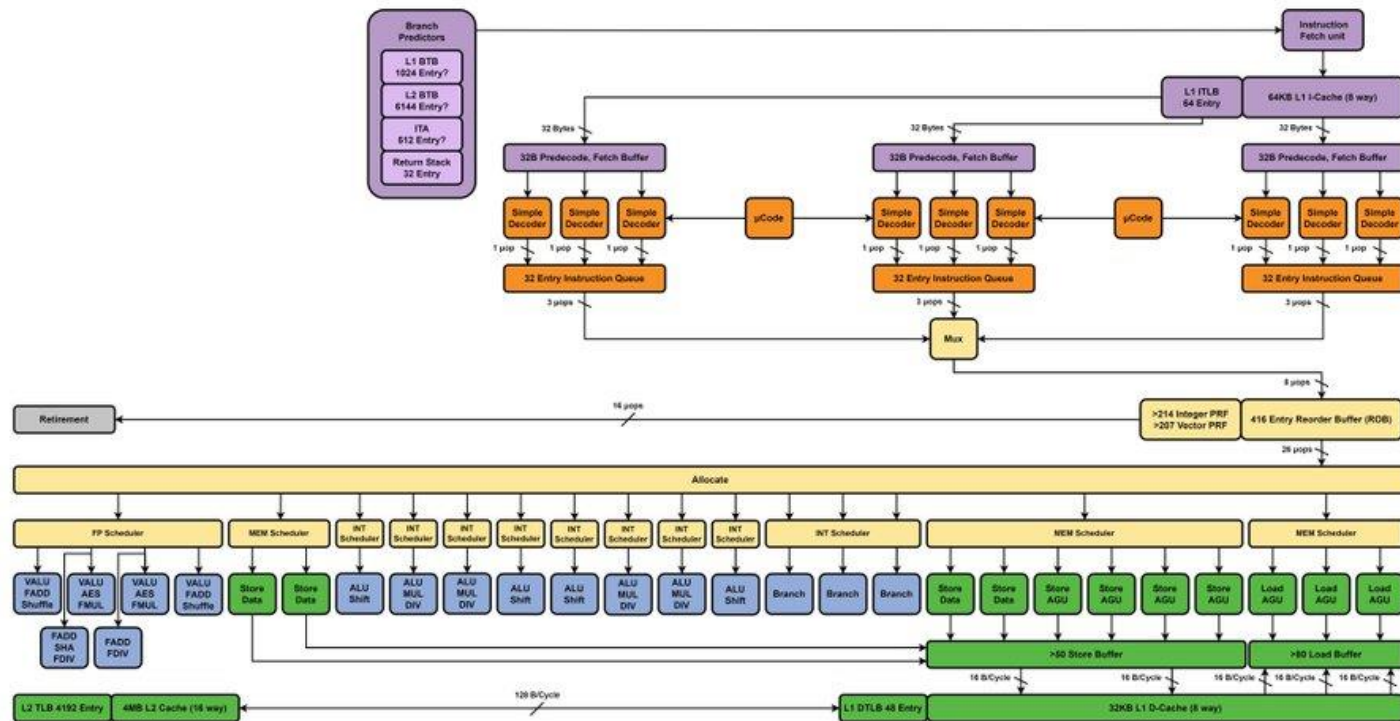
- [illegible]

<https://x.com/Cardyak>

<https://x.com/Cardyak>

Skymont

By Cardyak



RISC-V RVC – Keep it optional

- There are many more consequences of var-length I didn't have time to cover – namely redirection to variable places in a cacheline – this directly negatively affects BTB coverage
- RVC will have trouble achieving >4 IPC without an Op Cache
 - Even if you had a 50% code reduction, it will not pay for many fold losses in IPC
 - Real Code does not fit in the OpCache
- RVC has obvious benefits in embedded space where I\$ is very small
- Keep the extension optional, not required!
- Bibliography and Results – 3x from 1960s and 1x self-citation?

Bibliography

- [1] G. M. Amdahl, G. A. Blaauw, and Jr. F. P. Brooks. Architecture of the IBM System/360. *IBM Journal of R. & D.*, 8(2), 1964.
- [2] Werner Buchholz, editor. *Planning a computer system: Project Stretch*. McGraw-Hill Book Company, 1962.
- [3] James E. Thornton. Parallel operation in the Control Data 6600. In *Proceedings of the October 27-29, 1964, Fall Joint Computer Conference, Part II: Very High Speed Computer Systems*, AFIPS '64 (Fall, part II), pages 33–40, 1965.
- [4] Andrew Waterman. Improving energy efficiency and reducing code size with RISC-V compressed. Master's thesis, University of California, Berkeley, 2011.

RISC-V Vector Extensions (RVV) – how do I build this?

6.4. Example of stripmining and changes to SEW

The SEW and LMUL settings can be changed dynamically to provide high throughput on mixed-width operations in a single loop.

```
# Example: Load 16-bit values, widen multiply to 32b, shift 32b result
# right by 3, store 32b values.
# On entry:
# a0 holds the total number of elements to process
# a1 holds the address of the source array
# a2 holds the address of the destination array

loop:
    vsetvli a3, a0, e16, m4, ta, ma # vtype = 16-bit integer vectors;
                                   # also update a3 with v1 (# of elements this iteration)

    vle16.v v4, (a1)               # Get 16b vector
    slli t1, a3, 1                  # Multiply # elements this iteration by 2 bytes/source element
    add a1, a1, t1                  # Bump pointer
    vwmul.vx v8, v4, x10            # Widening multiply into 32b in <v8--v15>

    vsetvli x0, x0, e32, m8, ta, ma # Operate on 32b values
    vsrl.vi v8, v8, 3              # Shift right by 3

    vse32.v v8, (a2)               # Store vector of 32b elements
    slli t1, a3, 2                  # Multiply # elements this iteration by 4 bytes/destination element
    add a2, a2, t1                  # Bump pointer
    sub a0, a0, a3                  # Decrement count by v1
    bnez a0, loop                  # Any more?
```

I'm rather confused by the target machine of RVV. Perhaps I misunderstand the intent

Is this intended for a CPU?

If this is targeting an Out of Order core, how does this snippet run > 4 IPC?

<https://github.com/riscv/riscv-v-spec/blob/master/v-spec.adoc#sec-vector-extensions>

Vtype Implementation Questions

6.4. Example of stripmining and changes to SEW

The SEW and LMUL settings can be changed dynamically to provide high throughput on mixed-width operations in a single loop.

```
# Example: Load 16-bit values, widen multiply to 32b, shift 32b result
# right by 3, store 32b values.
# On entry:
# a0 holds the total number of elements to process
# a1 holds the address of the source array
# a2 holds the address of the destination array

loop:
    vsetvli a3, a0, e16, m4, ta, ma # vtype = 16-bit integer vectors;
                                     # also update a3 with vl (# of elements this iteration)
    vle16.v v4, (a1)                # Get 16b vector
    slli t1, a3, 1                   # Multiply # elements this iteration by 2 bytes/source element
    add a1, a1, t1                   # Bump pointer
    vwmul.vx v8, v4, x10             # Widening multiply into 32b in <v8--v15>

    vsetvli x0, x0, e32, m8, ta, ma # Operate on 32b values
    vsrl.vi v8, v8, 3
    vse32.v v8, (a2)                # Store vector of 32b elements
    slli t1, a3, 2                   # Multiply # elements this iteration by 4 bytes/destination element
    add a2, a2, t1                   # Bump pointer
    sub a0, a0, a3                   # Decrement count by vl
    bnez a0, loop                   # Any more?
```

Vtype Questions

- 6x src/dest registers?! So CISC then?
- Is the vtype CSR renamed or serializing?
- Are ta and ma used as dynamic predicate masks like SVE?
 - Are predicate bits therefore renamed?
 - Are they also an additional source for each scheduler to pick?
 - Does VL and masks live in INT or FPU? Do renamed predicates need to bounce between both?
- Element predication in SVE is a large reason for its loss in performance vs NEON when <512 bits
- Look up NEON vs SVE memcpy issues

<https://github.com/riscv/riscv-v-spec/blob/master/v-spec.adoc#sec-vector-extensions>

LMUL Implementation Questions

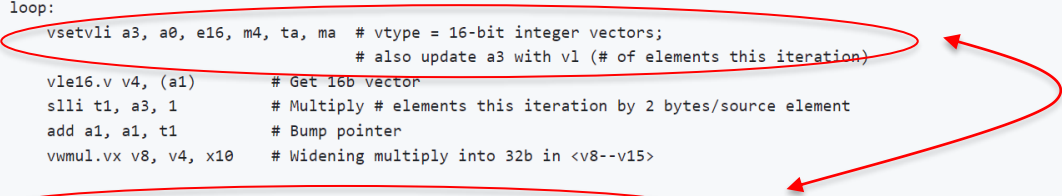
6.4. Example of stripmining and changes to SEW

The SEW and LMUL settings can be changed dynamically to provide high throughput on mixed-width operations in a single loop.

```
# Example: Load 16-bit values, widen multiply to 32b, shift 32b result
# right by 3, store 32b values.
# On entry:
# a0 holds the total number of elements to process
# a1 holds the address of the source array
# a2 holds the address of the destination array

loop:
    vsetvli a3, a0, e16, m4, ta, ma # vtype = 16-bit integer vectors;
                                   # also update a3 with v1 (# of elements this iteration)
    vle16.v v4, (a1)                # Get 16b vector
    slli t1, a3, 1                   # Multiply # elements this iteration by 2 bytes/source element
    add a1, a1, t1                   # Bump pointer
    vwmul.vx v8, v4, x10             # Widening multiply into 32b in <v8--v15>

    vsetvli x0, x0, e32, m8, ta, ma # Operate on 32b values
    vsrl.vi v8, v8, 3
    vse32.v v8, (a2)                 # Store vector of 32b elements
    slli t1, a3, 2                   # Multiply # elements this iteration by 4 bytes/destination element
    add a2, a2, t1                   # Bump pointer
    sub a0, a0, a3                   # Decrement count by v1
    bnez a0, loop                   # Any more?
```



LMUL Questions

- Are LMUL > 1 instructions atomic?
- When LMUL changes size, how does the renamer handle that?
- When LMUL and SEW change size, does the load/store, write-combine queue, prefetcher, and cache replacement algorithms get updated?

<https://github.com/riscv/riscv-v-spec/blob/master/v-spec.adoc#sec-vector-extensions>

LMUL Implementation Questions

6.4. Example of stripmining and changes to SEW

The SEW and LMUL settings can be changed dynamically to provide high throughput on mixed-width operations in a single loop.

```
# Example: Load 16-bit values, widen multiply to 32b, shift 32b result
# right by 3, store 32b values.
# On entry:
# a0 holds the total number of elements to process
# a1 holds the address of the source array
# a2 holds the address of the destination array

loop:
    vsetvli a3, a0, e16, m4, ta, ma # vtype = 16-bit integer vectors;
                                   # also update a3 with v1 (# of elements this iteration)

    vle16.v v4, (a1)               # Get 16b vector
    slli t1, a3, 1                  # Multiply # elements this iteration by 2 bytes/source element
    add a1, a1, t1                  # Bump pointer
    vwmul.vx v8, v4, x10            # Widening multiply into 32b in <v8--v15>

    vsetvli x0, x0, e32, m8, ta, ma # Operate on 32b values
    vsrl.vi v8, v8, 3
    vse32.v v8, (a2)               # Store vector of 32b elements
    slli t1, a3, 2                  # Multiply # elements this iteration by 4 bytes/destination element
    add a2, a2, t1                  # Bump pointer
    sub a0, a0, a3                  # Decrement count by v1
    bnez a0, loop                  # Any more?
```

Vector Execution Questions

- I assume this executes the entire defined vector as a single instruction (i.e. like AVX-512 single instruction execution)?
- If so, without an equivalent and synchronized datapath, this will be ucode, yes?
- With variable LMUL and SEW, it **all** becomes microcode? How is this not CISC?
- Variable SEW requires different datapaths for different element packing

<https://github.com/riscv/riscv-v-spec/blob/master/v-spec.adoc#sec-vector-extensions>

RISC-V RVV – Don't repeat SVE, repeat AVX

- My understanding may be completely incorrect, but as a micro-architect I see no viable pathway to build RVV at any level of out-of-order performance
- The ISA discussions looks identical to those for SVE, where optimizations for assembly and SW simplification completely overlook what it would take to build the machine.
- SVE is losing to NEON below 512 bits. NEON didn't expand to 512 bits, but if it did, it would outperform SVE
- Predicated execution and branch predictors don't mix
- I\$ footprint should not be the concern here, unrolled GEMM loops go brrrrrrr
- Is RVV completely abandoning the RISC philosophy of simplicity? All I see are ucode and CISC instructions.
- I'm trying to give you honest feedback, RVV needs a microarchitect to feedback real machine consequences of this proposal
- Or I'm just completely wrong, definitely a real possibility

Major CPU architectures – a new split

- Intel's P6 microarchitecture was a revolution in CPU design and was the gold standard for almost 3 decades
- Since the advent of LLVM, code has turned into “indirect threading garbage”.
 - This evolution is necessary – to span ISAs, run on embedded OS's (e.g. Chrome), and allow SW to “just work” without coding to the metal
- Apple, starting in A9 and expanding even to today, shows a mastery of a new uarch design philosophy.
 - A14 Firestorm seems to be the gold standard in this new world
- Out – high frequency bursts, IPC defined by decode width, variable length
- In – 1-2GHz sustained, IPC limited by memory sub system, indirect branch heavy, metric tons of int ALUs, 4x LDs+, brute force width for bursts

Rehash – “Real Code” is Indirect Threading

```

start:
    ip = &thread // points to '&i_pushA'
    jump (*ip) // follow pointers to 1st instruction of 'push', DO NOT advance ip yet
thread:
    &i_pushA
    &i_pushB
    &i_add
    ...
i_pushA:
    &push
    &A
i_pushB:
    &push
    &B
i_add:
    &add
push:
    *sp++ = *(*ip + 1) // Look 1 past start of indirect block for operand address
    jump *(*++ip) // advance ip in thread, jump through next indirect block to next subroutine
add:
    addend1 = *--sp
    addend2 = *--sp
    *sp++ = addend1 + addend2
    jump *(*++ip)

```

https://en.wikipedia.org/wiki/Threaded_code#Indirect_threading

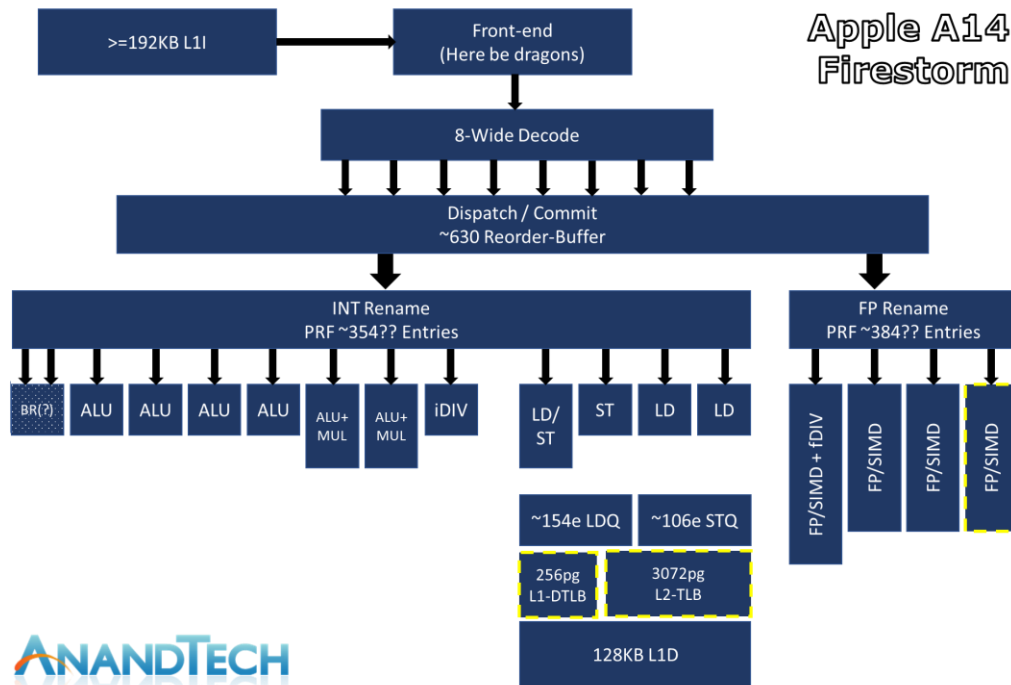
For Whom are you building machines?

- Governments use Fortran
- HPC, Gaming use C/C++ (and Excel spreadsheets)
- AI uses MACs and Python. (That's pretty much it.)
- The Internet, VMs use JS and JITs (**this is most of the world**)
 - <https://browserbench.org/Speedometer2.0/>
 - That's ~1B instructions / iteration to sort a 100-deep text list.
 - Try it on your phone and laptop right now, see which is faster
 - Indirect branches and integer ALUs
 - This is for SW's dev benefit, not HW. There are exponentially more of them (SW), they win.
- We knew about this phenomenon in 2005. What is SPEC benchmarking again?
 - <https://www.amazon.com/Virtual-Machines-Versatile-Platforms-Architecture/dp/1558609105>
- The newest CPU uarchs are **heavy** with indirect-branch target storage, algorithms like IT-TAGE (Seznec).

Firestorm is the LLVM/JS/JIT gold standard

- **Massive Caches**
 - Not really a mystery – seen from space
 - Likely the greatest contributor to low energy – extremely fewer accesses to DRAM than competitors
 - No other ARM competitor comes even close to paying the area for this significant advantage
- Measured massive Indirect Branch Target chains
- Tons of simple ALUs, 3xLD to feed it
- Extremely short mis-predict pipe (IPC >> frequency)
- **No OP cache**, 8-Wide decode fixed-width instructions.
 - Why not 10-wide? Why not more? Fixed width decode is cheap
- Vertical Integration with SW
 - Adding cute ISA instructions does not make up for lacking this
- SMP, not SMT

Apple A14
Firestorm



<https://images.anandtech.com/doci/16226/Firestorm.png>

New world examples

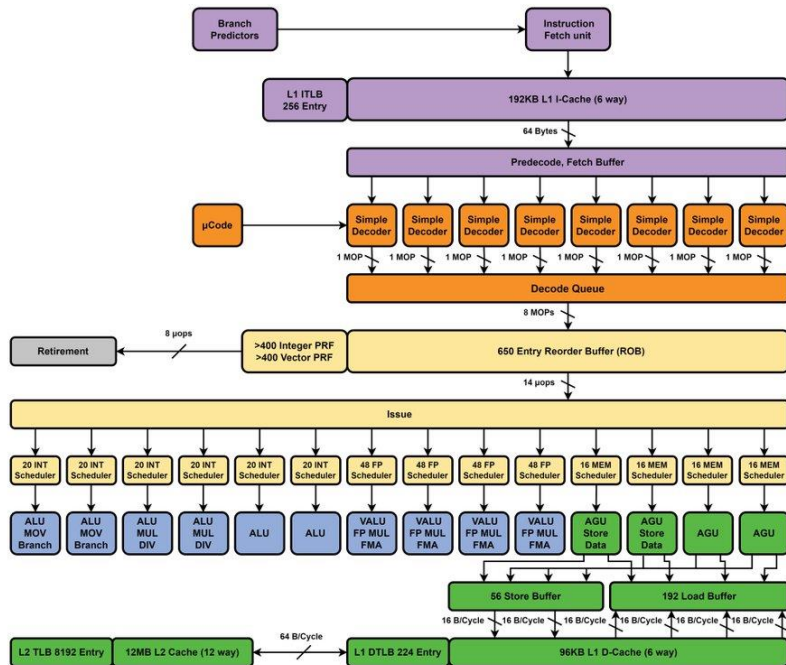
<https://x.com/Cardyak>

Qualcomm - 2024

Oryon

Microarchitecture Block Diagram

By Cardyak

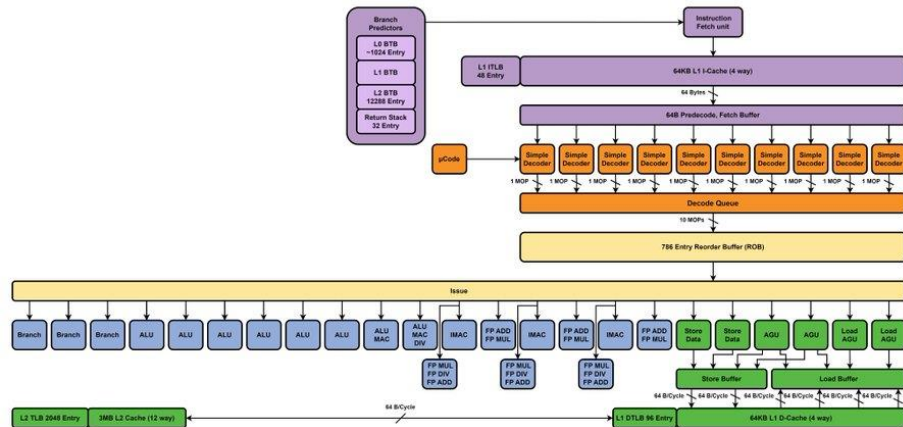


ARM - 2024

Cortex-X925

Microarchitecture Block Diagram

By Cardyak



Rehash -- SMT – the good, bad, and always ugly

I'm bringing this back up to applaud and agree with Intel's new Lion Cove architecture unveiled yesterday at HotChips2024

SMT bad when used on out-of-order machines

Claim: *SMT in an out-of-order, virtualized, non-shared context will be beaten by single-threaded SMP*

- 2x85% IPC SMP cores at ½ the size beat ~120% IPC SMT
- Full-context SMT takes >> 10% area in reality. Find a die shot (not a paper or claim) where this is untrue.
- “Noisy neighbor effect” with cross-polluting caches
- Spectre-style security problems
- HW-multi thread scheduling unable to predict SW multi-thread intent, forecasting, prefetching
- All threads must sleep before powering down
- Out-of-order resources lost to hold variable SMT retire state
- Turn off hyperthreading and see for yourself. Did you lose anything?

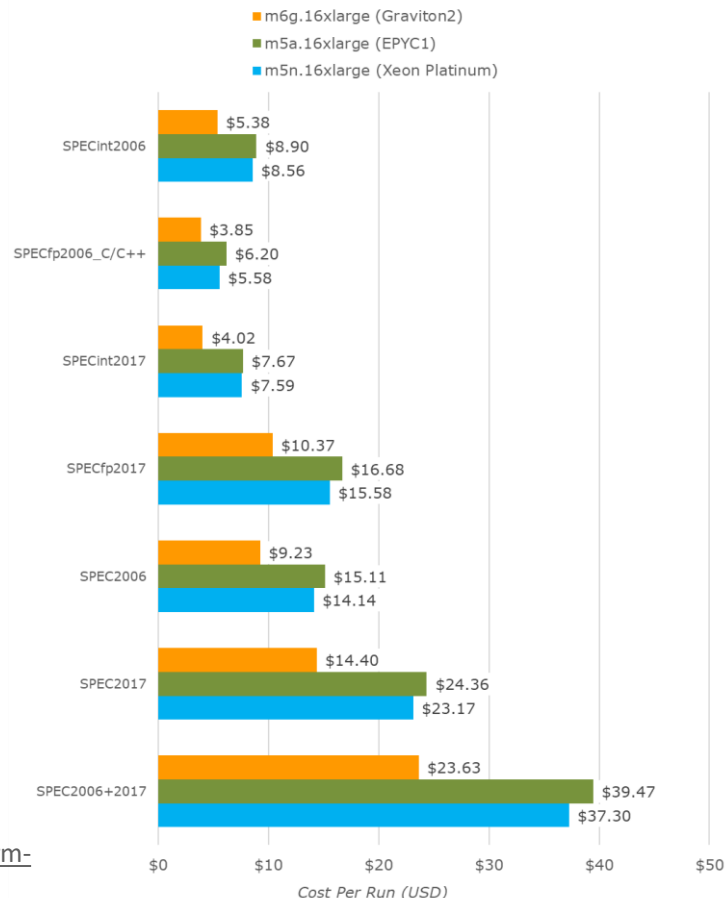
SMT good when used on in-order machines

Claim: *SMT in a shared-context, SW controlled, scalar and in-order throughput machine gains scale-able benefits of concurrent control and execution*

- [Qualcomm's DSP Hexagon](#)
- Tesla's Dojo AI D1

<https://www.anandtech.com/show/15578/cloud-clash-amazon-graviton2-arm-against-intel-and-amd/9>

Amazon EC2 Cost Per SPEC Test (64 rate/vCPU)



Car Picture

