Topic 4

Advanced Processors III

Xinfei Guo
xinfei.guo@sjtu.edu.cn

June 6th, 2022

T4 learning goals

- Advanced Processors
  - Section I: Superpipelined & Superscalar Pipelines
  - Section II & III: Out-of-order (OoO) Pipelines

Recall: Types of Dependencies

- **RAW** (Read After Write) = "true dependence" (true)

  mul r0 * r1 ➔ **r2**

  …

  add **r2** + r3 ➔ r4

- **WAW** (Write After Write) = "output dependence" (false)

  mul r0 * r1 ➔ **r2**

  …

  add r1 + r3 ➔ **r2**

- **WAR** (Write After Read) = "anti-dependence" (false)

  mul r0 * **r1** ➔ r2

  …

  add r3 + r4 ➔ **r1**

- WAW & WAR are "false", Can be **totally eliminated** by "renaming"

Also Have Dependencies via Memory

- **If value in "r2" and "r3" is the same…**
- RAW (Read After Write) – True dependency

  st r1 → **[r2]**

  …

  ld **[r3]** → r4
- WAW (Write After Write)

  st r1 → **[r2]**

  …

  st r4 → **[r3]**
- WAR (Write After Read)

  ld **[r2]** → r1

  …

  st r4 → **[r3]**

WAR/WAW are "false dependencies"

- But can't rename memory in same way as registers
  - **Why? Addresses are not known at rename**
- Need to use other tricks

Let's Start with Just Stores

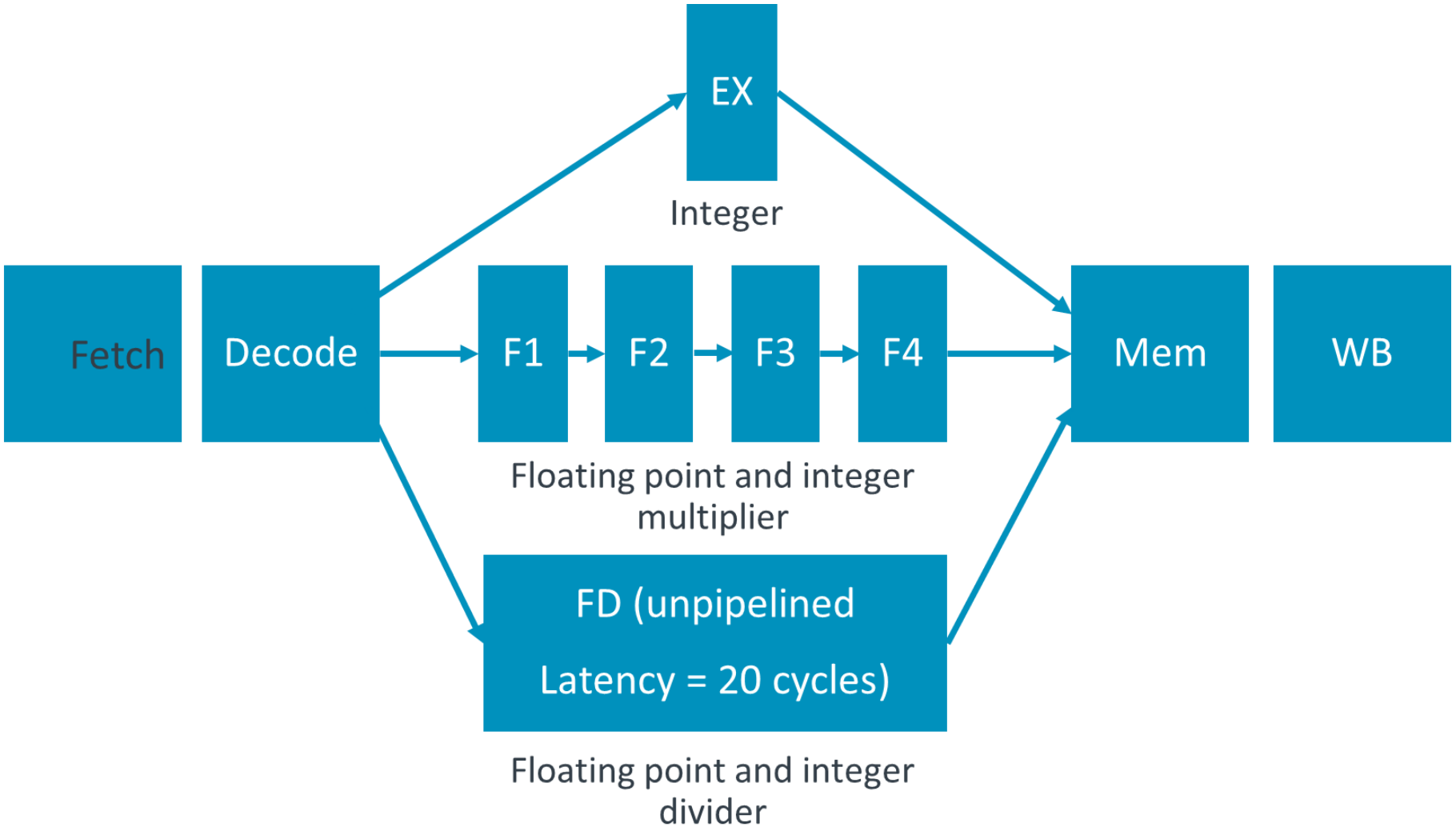
- Stores: Write data cache, not registers
  - Can we rename memory?
- No (at least not easily)
  - Cache writes unrecoverable
- Solution: write stores into cache only when certain
  - When are we certain? At "commit"

Diversified Pipelines

- It is impractical to require that all instructions execute in a single cycle.
- We also want to avoid sending all instructions down a single long pipeline.
- We can instead introduce multiple (or “diversified”) execution pipelines.
- Can potentially introduce WAR hazards.

source: ARM

Diversified Pipelines



source: ARM

Handling Stores

	0	1	2	3	4	5	6	7	8	9	10	11	12
mul p1 * p2 → p3	F	Di	I	R R	X_1	X_2	X_3	X_4	W	C			
jump-not-zero p3	F	Di					I	R R	X	W	C		
st p5 → [p3+4]		F	Di				I	R R	X	M	W	C	
st p4 → [p6+8]		F	Di	I?									

- Can "st p4 → [p6+8]" issue in cycle 3?
  - Its register inputs are ready…

Reminder

- Lab #4 Lecture (**RISC-V ISA**)
- HW #2 Due next Monday
- Lab #3 Due this Friday
- Review slides and read the book

Quick Quiz

A special unit used to govern the OoO of the instructions is ?

A. Commitment unit

B. Temporal unit

C. Monitor

D. Supervisory unit

Problem #1: Out-of-Order Stores

	0	1	2	3	4	5	6	7	8	9	10	11	12
mul p1 * p2 ➜ p3	F	Di	I	R R	X_1	X_2	X_3	X_4	W	C			
jump-not-zero p3	F	Di					I	R R	X	W	C		
st p5 ➜ [p3+4]		F	Di				I	R R	X	**M**	W	C	
st p4 ➜ [p6+8]		F	Di	**I?**	**RR**	**X**	**M**	**W**				C	

- Can "st p4 ➜ [p6+8]" write the cache in cycle 6?
  - "st p5 ➜ [p3+4]" has not yet executed
- What if p3+4 == p6+8?
  - The two stores write the same address! WAW dependency!
  - Not known until their "X" stages (cycle 5 & 8)
- Unappealing solution: all stores execute in-order
- We can do better…

Problem #2: Speculative Stores

	0	1	2	3	4	5	6	7	8	9	10	11	12
mul p1 * p2 → p3	F	Di	I	R R	X_1	X_2	X_3	X_4	W	C			
jump-not-zero p3	F	Di					I	R R	X	W	C		
st p5 → [p3+4]		F	Di				I	R R	X	M	W	C	
st p4 → [p6+8]		F	Di	**I?**	**RR**	**X**	**M**	**W**				**C**	

- Can "st p4 → [p6+8]" write the cache in cycle 6?
  - Store is still "speculative" at this point
- What if "jump-not-zero" is mis-predicted?
  - Not known until its "X" stage (cycle 8)
- How does it "undo" the store once it hits the cache?
  - Answer: it can't; stores write the cache only at **commit**
  - Guaranteed to be non-speculative at that point

Store Queue (SQ)

- Solves two problems
  - Allows for recovery of speculative stores
  - Allows out-of-order stores
- Store Queue (SQ)
  - **At dispatch, each store is given a slot in the Store Queue**
  - First-in-first-out (FIFO) queue
  - Each entry contains: “address”, “value”, and “bday”
- Operation:
  - Dispatch (in-order): allocate entry in SQ (stall if full)
  - Execute (out-of-order): write store value into store queue
  - Commit (in-order): read value from SQ and write into data cache
  - Branch recovery: remove entries from the store queue
- Also solves problems with loads

Store Queue Operation

	0	1	2	3	4	5	6	7	8	9	10	11	12
fdiv p1 / p2 → p9	F	Di	I	R R	X_1	X_2	X_3	X_4	X_5	X_6	W	C	
st p4 → [p5+4]	F	Di	I	R R	X	S Q						C	
st p3 → [p6+8]		F	Di	I	R R	X	S Q						C

- Stores write to SQ, not M
  - similar to register renaming, where we allocated a new physical register for each insn

Memory Forwarding

	0	1	2	3	4	5	6	7	8	9	10	11	12
fdiv p1 / p2 → p9	F	Di	I	R R	X_1	X_2	X_3	X_4	X_5	X_6	W	C	
st p4 → [p5+4]	F	Di	I	R R	X	S Q						C	
st p3 → [p6+8]		F	Di	I	R R	X	S Q						C
ld [p7] → p8		F	Di	I?	R R	X	M_1	M_2	W				C

- Can "ld [p7] → p8" issue and begin execution?
  - Why or why not?

Memory Forwarding

	0	1	2	3	4	5	6	7	8	9	10	11	12
fdiv p1 / p2 ➜ p9	F	Di	I	R R	X_1	X_2	X_3	X_4	X_5	X_6	W	C	
st p4 ➜ [p5+4]	F	Di	I	R R	X	S Q						C	
st p3 ➜ [p6+8]		F	Di	I	R R	X	S Q						C
ld [p7] ➜ p8		F	Di	I?	R R	X	M_1	M_2	W				C

- Can “ld [p7] ➜ p8” issue and begin execution?
  - Why or why not?
- If the load reads from either of the stores’ addresses…
  - Load must get correct value, but stores don’t write cache until commit…
- Solution: “memory forwarding”
  - Load also searches the Store Queue (in parallel with cache access)
  - Conceptually like register bypassing, but different implementation
    - Why? Addresses unknown until execute

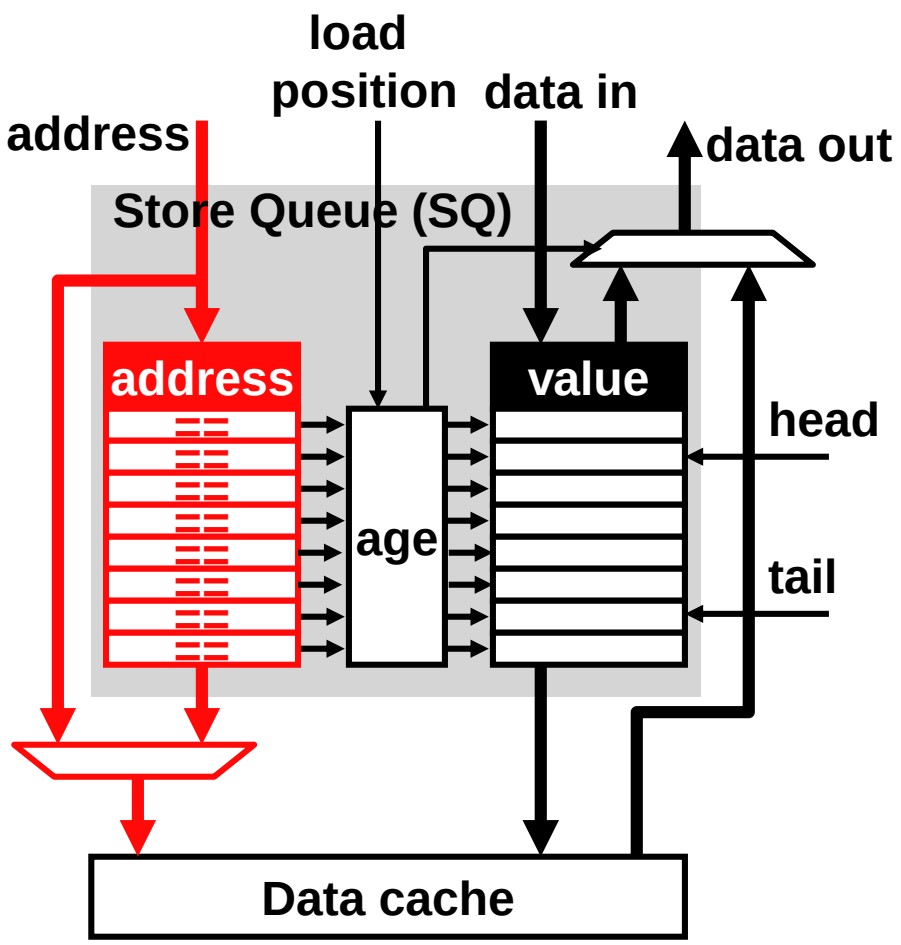
Problem #3: WAR Hazards

	0	1	2	3	4	5	6	7	8	9	10	11	12
mul p1 * p2 → p3	F	Di	I	R R	X_1	X_2	X_3	X_4	W	C			
jump-not-zero p3	F	Di					I	R R	X	W	C		
ld [p3+4] → p5		F	Di				I	R R	X	M_1	M_2	W	C
st p4 → [p6+8]		F	Di	I	RR	X	S Q						C

- What if "p3+4 == p6 + 8"?
  - WAR: need to make sure that load doesn't read store's result
  - Need to get values based on "program order" not "execution order"
- Bad solution: require all stores/loads to execute in-order
- Good solution: add "age" fields to store queue (SQ)
  - Loads read from **youngest older (than load) matching** store
  - Another reason the SQ is a FIFO queue

Memory Forwarding via Store Queue

- Store Queue (SQ)
  - Holds all in-flight stores
  - CAM: fast search
  - Age logic: determine youngest matching store older than load
- Store rename/dispatch
  - Allocate entry in SQ
- Store execution
  - Update SQ
    - Address + Data
- Load execution
  - Search SQ identify youngest older matching store
    - Match? Read SQ
    - No Match? Read cache



Store Queue (SQ)

- On load execution, select the store that is:
  - To same address as load
  - Older than the load (before the load in program order)
- Of these matching stores, select the youngest
  - The store to the same address that immediately precedes the load

When Can Loads Execute?

	0	1	2	3	4	5	6	7	8	9	10	11	12
mul p1 * p2 → p3	F	Di	I	R R	X_1	X_2	X_3	X_4	W	C			
jump-not-zero p3	F	Di					I	R R	**X**	W	C		
st p5 → [p3+4]		F	Di				I	R R	X	S Q	C		
ld [p6+8] → p7		F	Di	**I?**	**RR**	**X**	**M_1**	**M_2**	**W**			C	

- Can "ld [p6+8] → p7" issue in cycle 3
  - Why or why not?

When Can Loads Execute?

	0	1	2	3	4	5	6	7	8	9	10	11	12
mul p1 * p2 → p3	F	Di	I	R R	X_1	X_2	X_3	X_4	W	C			
jump-not-zero p3	F	Di					I	R R	**X**	W	C		
st p5 → [p3+4]		F	Di				I	R R	X	S Q	C		
ld [p6+8] → p7		F	Di	**I?**	**RR**	**X**	**M_1**	**M_2**	**W**			C	

- Aliasing! Does p3+4 == p6+8?
  - If no, load should get value from memory
    - **Can it start to execute?**
  - If yes, load should get value from store
    - By reading the store queue?
    - **But the value isn't put into the store queue until cycle 9**
- **Key challenge**: don't know addresses until execution!
  - One solution: require all loads to wait for all earlier (prior) stores

Conservative Load Scheduling

- Conservative load scheduling:
  - All older stores have executed
    - Some architectures: split store address / store data
      - Only requires knowing addresses (not the store values)
  - Advantage: always safe
  - Disadvantage: performance (limits ILP)

Conservative Load Scheduling

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
ld [p1] → p4	F	Di	I	Rr	X	M_1	M_2	W	C							
ld [p2] → p5	F	Di	I	Rr	X	M_1	M_2	W	C							
add p4, p5 → p6		F	Di			I	Rr	X	W	C						
st p6 → [p3]		F	Di				I	Rr	X	SQ	C					
ld [p1+4] → p7			F	Di				I	Rr	X	M_1	M_2	W	C		
ld [p2+4] → p8			F	Di				I	Rr	X	M_1	M_2	W	C		
add p7, p8 → p9				F	Di						I	Rr	X	W	C	
st p9 → [p3+4]				F	Di							I	Rr	X	SQ	C

- Conservative load scheduling: can't issue ld [p1+4] until cycle 7!
- Might as well be an in-order machine on this example
- Can we do better? How?

Optimistic Load Scheduling

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
ld [p1] → p4	F	Di	I	Rr	X	M_1	M_2	W	C							
ld [p2] → p5	F	Di	I	Rr	X	M_1	M_2	W	C							
add p4, p5 → p6		F	Di			I	Rr	X	W	C						
st p6 → [p3]		F	Di				I	Rr	X	SQ	C					
ld [p1+4] → p7			F	Di	I	Rr	X	M_1	M_2	W	C					
ld [p2+4] → p8			F	Di	I	Rr	X	M_1	M_2	W		C				
add p7, p8 → p9				F	Di			I	Rr	X	W	C				
st p9 → [p3+4]				F	Di				I	Rr	X	SQ	C			

Optimistic load scheduling: can actually benefit from out-of-order!

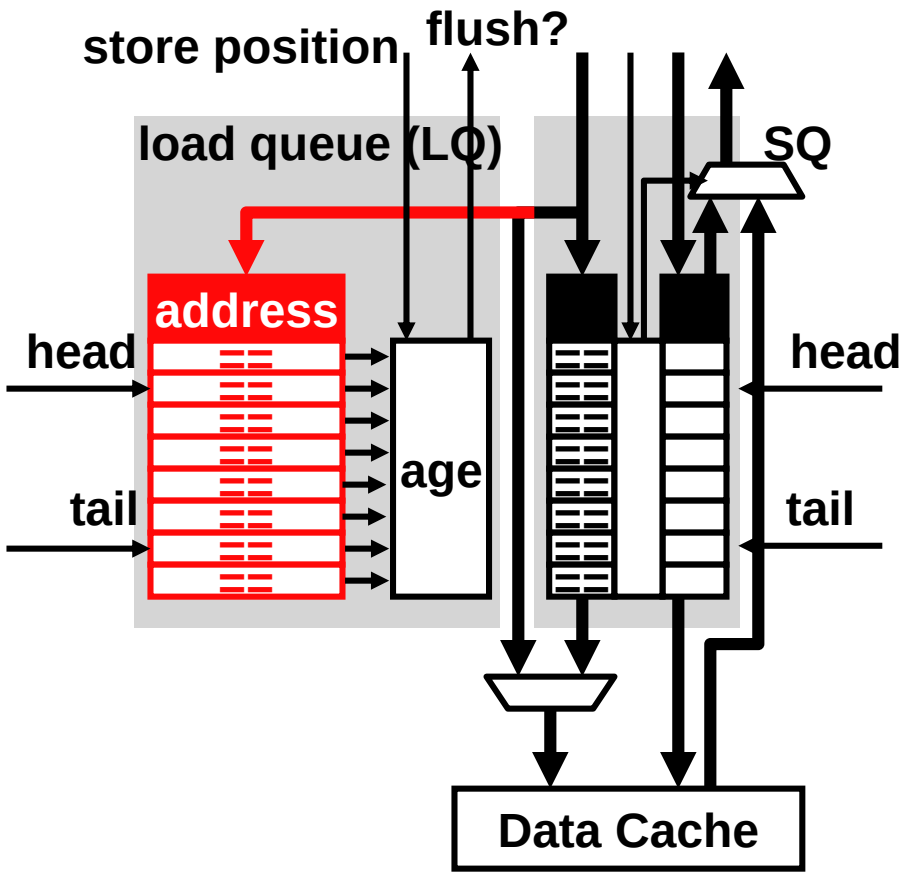
Let's speculate!

Load Speculation

- Speculation requires three things…..
  1. When do we speculate?
  2. How do we detect a mis-speculation?
  3. How do we recover from mis-speculations?
     - Squash offending load and all newer insns
     - Similar to branch mis-prediction recovery

Load Queue

- Detects load ordering violations
- Load execution: Write address into LQ
  - Also note any store forwarded from
- Store execution: Search LQ
  - Younger load with same addr?
    - Did younger load forward from younger store?



Store Queue + Load Queue

- Store Queue: handles forwarding, allows OoO stores
  - Entry per store (allocated @ dispatch, deallocated @ commit)
  - Written by stores (@ execute)
  - Searched by loads (@ execute)
  - Read from SQ to write data cache (@ commit)
- Load Queue: detects ordering violations
  - Entry per load (allocated @ dispatch, deallocated @ commit)
  - Written by loads (@ execute)
  - Searched by stores (@ execute)
- Both together
  - Allows aggressive load scheduling
  - Stores don’t constrain load execution

Optimistic Load Scheduling Problem

- Allows loads to issue before older stores
  - Increases ILP
  - \+ Good: When no conflict, increases performance
  - \- Bad: Conflict => squash => worse performance than waiting
- Can we have our cake AND eat it too?

Predictive Load Scheduling

- Predict which loads must wait for stores

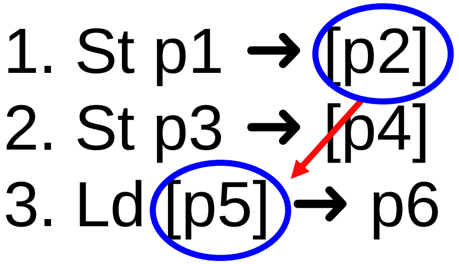
- Fool me once, shame on you-- fool me twice?
  - Loads default to aggressive
  - Keep table of load PCs that have been caused squashes
    - Schedule these conservatively
  - \+ Simple predictor
  - \- Makes “bad” loads wait for **all** older stores

- More complex predictors used in practice
  - Predict which stores loads should wait for
  - “Store Sets” paper on Canvas (Reading Material > Store Set Paper.pdf)

Load/Store Queue Examples

Initial State

(Stores to different addresses)



1. St p1 → [p2]
2. St p3 → [p4]
3. Ld [p5] → p6

RegFile

p1	5
p2	100
p3	9
p4	200
p5	100
p6	---
p7	---
p8	---

Load Queue

Bdy	**Addr**

Store Queue

Bdy	**Addr**	**Val**

Cache

Addr	**Val**
100	13
200	17

RegFile

p1	5
p2	100
p3	9
p4	200
p5	100
p6	---
p7	---
p8	---

Load Queue

Bdy	**Addr**

Store Queue

Bdy	**Addr**	**Val**

Cache

Addr	**Val**
100	13
200	17

RegFile

p1	5
p2	100
p3	9
p4	200
p5	100
p6	---
p7	---
p8	---

Load Queue

Bdy	**Addr**

Store Queue

Bdy	**Addr**	**Val**

Cache

Addr	**Val**
100	13
200	17

Good Interleaving

1. St p1 → [p2]
2. St p3 → [p4]
3. Ld [p5] → p6

(Shows importance of address check)

1. St p1 → [p2]

RegFile

p1	5
p2	100
p3	9
p4	200
p5	100
p6	---
p7	---
p8	---

Load Queue

Bdy	Addr

Store Queue

Bdy	Addr	Val
1	100	5

Cache

Addr	Val
100	13
200	17

2. St p3 → [p4]

RegFile

p1	5
p2	100
p3	9
p4	200
p5	100
p6	---
p7	---
p8	---

Load Queue

Bdy	Addr

Store Queue

Bdy	Addr	Val
1	100	5
2	200	9

Cache

Addr	Val
100	13
200	17

3. Ld [p5] → p6

RegFile

p1	5
p2	100
p3	9
p4	200
p5	100
p6	**5**
p7	---
p8	---

Load Queue

Bdy	Addr
3	100

Store Queue

Bdy	Addr	Val
1	100	5
2	200	9

Cache

Addr	Val
100	13
200	17

Different Initial State

(All to same address)

1. St p1 → [p2]
2. St p3 → [p4]
3. Ld [p5] → p6

RegFile

p1	5
p2	100
p3	9
p4	100
p5	100
p6	---
p7	---
p8	---

Load Queue

Bdy	**Addr**

Store Queue

Bdy	**Addr**	**Val**

Cache

Addr	**Val**
100	13
200	17

RegFile

p1	5
p2	100
p3	9
p4	100
p5	100
p6	---
p7	---
p8	---

Load Queue

Bdy	**Addr**

Store Queue

Bdy	**Addr**	**Val**

Cache

Addr	**Val**
100	13
200	17

RegFile

p1	5
p2	100
p3	9
p4	100
p5	100
p6	---
p7	---
p8	---

Load Queue

Bdy	**Addr**

Store Queue

Bdy	**Addr**	**Val**

Cache

Addr	**Val**
100	13
200	17

Good Interleaving #1

1. St p1 → [p2]
2. St p3 → [p4]
3. Ld [p5] → p6

(Program Order)

1. St p1 → [p2]

RegFile

p1	5
p2	100
p3	9
p4	100
p5	100
p6	---
p7	---
p8	---

Load Queue

Bdy	Addr

Store Queue

Bdy	Addr	Val
1	100	5

Cache

Addr	Val
100	13
200	17

2. St p3 → [p4]

RegFile

p1	5
p2	100
p3	9
p4	100
p5	100
p6	---
p7	---
p8	---

Load Queue

Bdy	Addr

Store Queue

Bdy	Addr	Val
1	100	5
2	100	9

Cache

Addr	Val
100	13
200	17

3. Ld [p5] → p6

RegFile

p1	5
p2	100
p3	9
p4	100
p5	100
p6	**9**
p7	---
p8	---

Load Queue

Bdy	Addr
3	100

Store Queue

Bdy	Addr	Val
1	100	5
2	100	9

Cache

Addr	Val
100	13
200	17

Good Interleaving #2

(Stores reordered)

1. St p1 → [p2]
2. St p3 → [p4]
3. Ld [p5] → p6

2. St p3 → [p4]

RegFile

p1	5
p2	100
p3	9
p4	100
p5	100
p6	---
p7	---
p8	---

Load Queue

Bdy	Addr

Store Queue

Bdy	Addr	Val
1		
2	100	9

Cache

Addr	Val
100	13
200	17

1. St p1 → [p2]

RegFile

p1	5
p2	100
p3	9
p4	100
p5	100
p6	---
p7	---
p8	---

Load Queue

Bdy	Addr

Store Queue

Bdy	Addr	Val
1	100	5
2	100	9

Cache

Addr	Val
100	13
200	17

3. Ld [p5] → p6

RegFile

p1	5
p2	100
p3	9
p4	100
p5	100
p6	**9**
p7	---
p8	---

Load Queue

Bdy	Addr
3	100

Store Queue

Bdy	Addr	Val
1	100	5
2	100	9

Cache

Addr	Val
100	13
200	17

Bad Interleaving #1

1. St p1 → [p2]
2. St p3 → [p4]
3. Ld [p5] → p6

(Load reads the cache)

3. Ld [p5] → p6

RegFile

p1	5
p2	100
p3	9
p4	100
p5	100
p6	13
p7	---
p8	---

Load Queue

Bdy	Addr
3	100

Store Queue

Bdy	Addr	Val
1		
2		

Cache

Addr	Val
100	13
200	17

2. St p3 → [p4]

RegFile

p1	5
p2	100
p3	9
p4	100
p5	100
p6	13
p7	---
p8	---

Load Queue

Bdy	Addr
3	100

Store Queue

Bdy	Addr	Val
2	100	9

Cache

Addr	Val
100	13
200	17

Bad Interleaving #2

1. St p1 → [p2]
2. St p3 → [p4]
3. Ld [p5] → p6

(Load gets value from wrong store)

1. St p1 → [p2]

RegFile

p1	5
p2	100
p3	9
p4	100
p5	100
p6	---
p7	---
p8	---

Load Queue

Bdy	Addr	Bdy of Fwd. Store

Store Queue

Bdy	Addr	Val
1	100	5

Cache

Addr	Val
100	13
200	17

3. Ld [p5] → p6

RegFile

p1	5
p2	100
p3	9
p4	100
p5	100
p6	**5**
p7	---
p8	---

Load Queue

Bdy	Addr	Bdy of Fwd. Store
3	100	1

Store Queue

Bdy	Addr	Val
1	100	5
2		

Cache

Addr	Val
100	13
200	17

2. St p3 → [p4]

RegFile

p1	5
p2	100
p3	9
p4	100
p5	100
p6	5
p7	---
p8	---

Load Queue

Bdy	Addr	Bdy of Fwd. Store
3	100	1

Store Queue

Bdy	Addr	Val
1	100	5
2	100	9

Cache

Addr	Val
100	13
200	17

Bad/Good Interleaving

(Load gets value from correct store, but does it work?)

1. St p1 → [p2]
2. St p3 → [p4]
3. Ld [p5] → p6

2. St p3 → [p4]

RegFile

p1	5
p2	100
p3	9
p4	100
p5	100
p6	---
p7	---
p8	---

Load Queue

Bdy	Addr	Bdy of Fwd. Store

Store Queue

Bdy	Addr	Val
1		
2	100	9

Cache

Addr	Val
100	13
200	17

3. Ld [p5] → p6

RegFile

p1	5
p2	100
p3	9
p4	100
p5	100
p6	**9**
p7	---
p8	---

Load Queue

Bdy	Addr	Bdy of Fwd. Store
3	100	2

Store Queue

Bdy	Addr	Val
1		
2	100	9

Cache

Addr	Val
100	13
200	17

1. St p1 → [p2]

RegFile

p1	5
p2	100
p3	9
p4	100
p5	100
p6	9
p7	---
p8	---

Load Queue

Bdy	Addr	Bdy of Fwd. Store
3	100	2

Store Queue

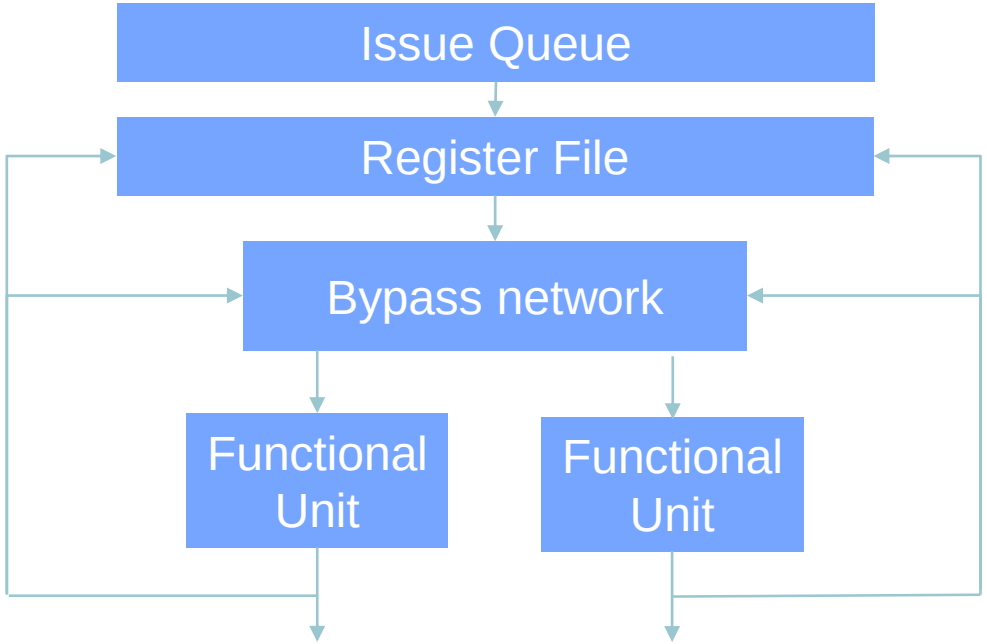
Bdy	Addr	Val
1	100	5
2	100	9

Cache

Addr	Val
100	13
200	17

Superscalar Processors: Data Forwarding (Bypass) Network

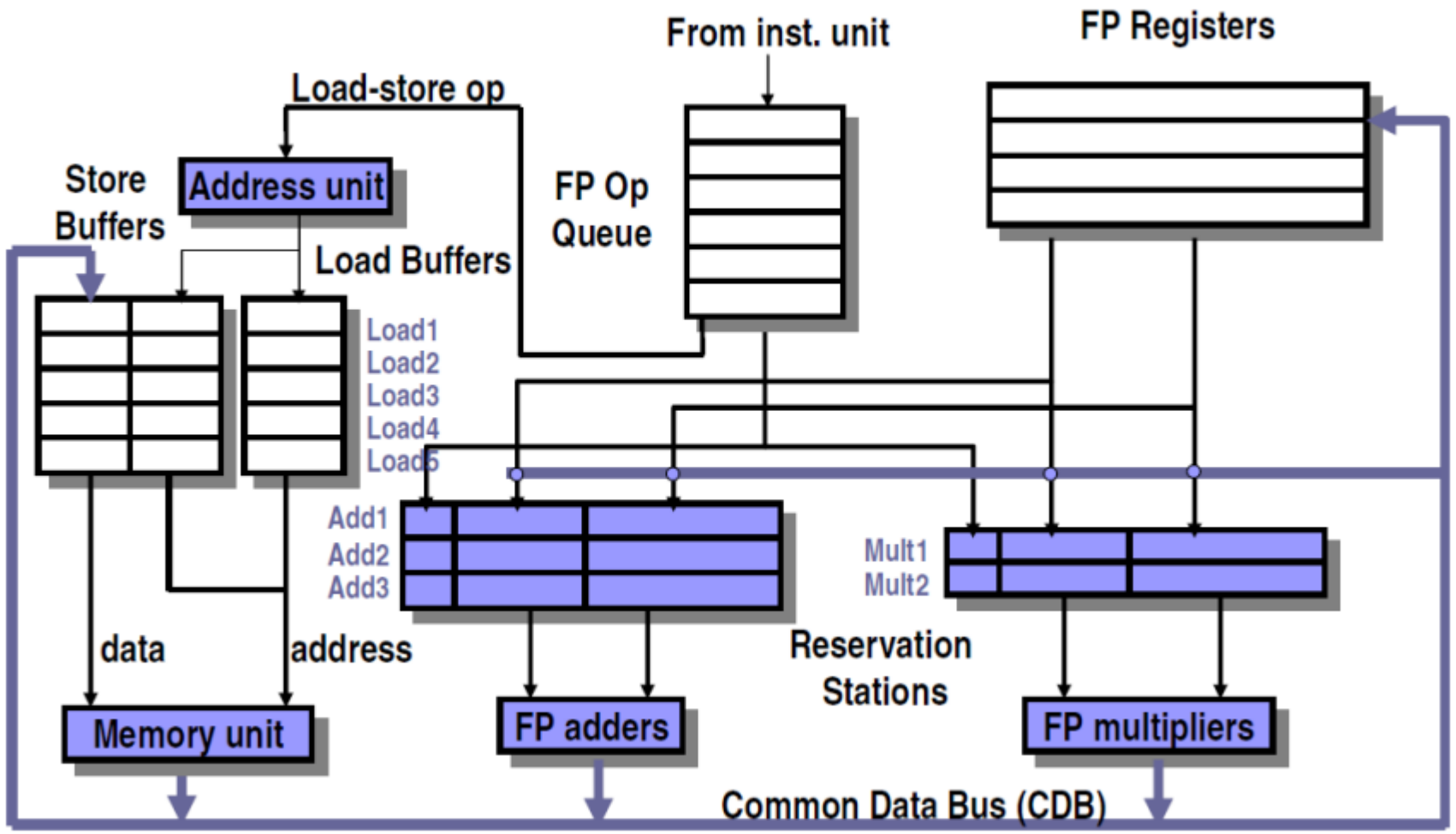
- Data forwarding in a scalar pipeline is relatively simple, consisting of a few extra buses and multiplexers.
- In a superscalar processor, we have many parallel functional units and may need to forward any recently generated results to the input of any functional unit. For example:



OoO & Superscalar

- Dynamic scheduling and multiple issue are orthogonal
  - e.g. Pentium 4: dynamically scheduled 5-way superscalar
  - Two dimensions
    - **N**: superscalar width (number of parallel operations)
    - **W**: window size (number of reservation stations)

Putting it all together



Tomasulo Organization

Adapted from Nanjing U, CS252@UC Berkeley and CS 246@Harvard University

Tomasulo's Algorithm/Organization

- Used in IBM 360/91 Machines (Late 60s)
- Key concept: Reservation Stations (RS)
- Added renaming in hardware
- Can eliminate WAW and WAR hazards
- Earlier: Scoreboarding
- Small number of floating point registers prevented interesting compiler scheduling of operations

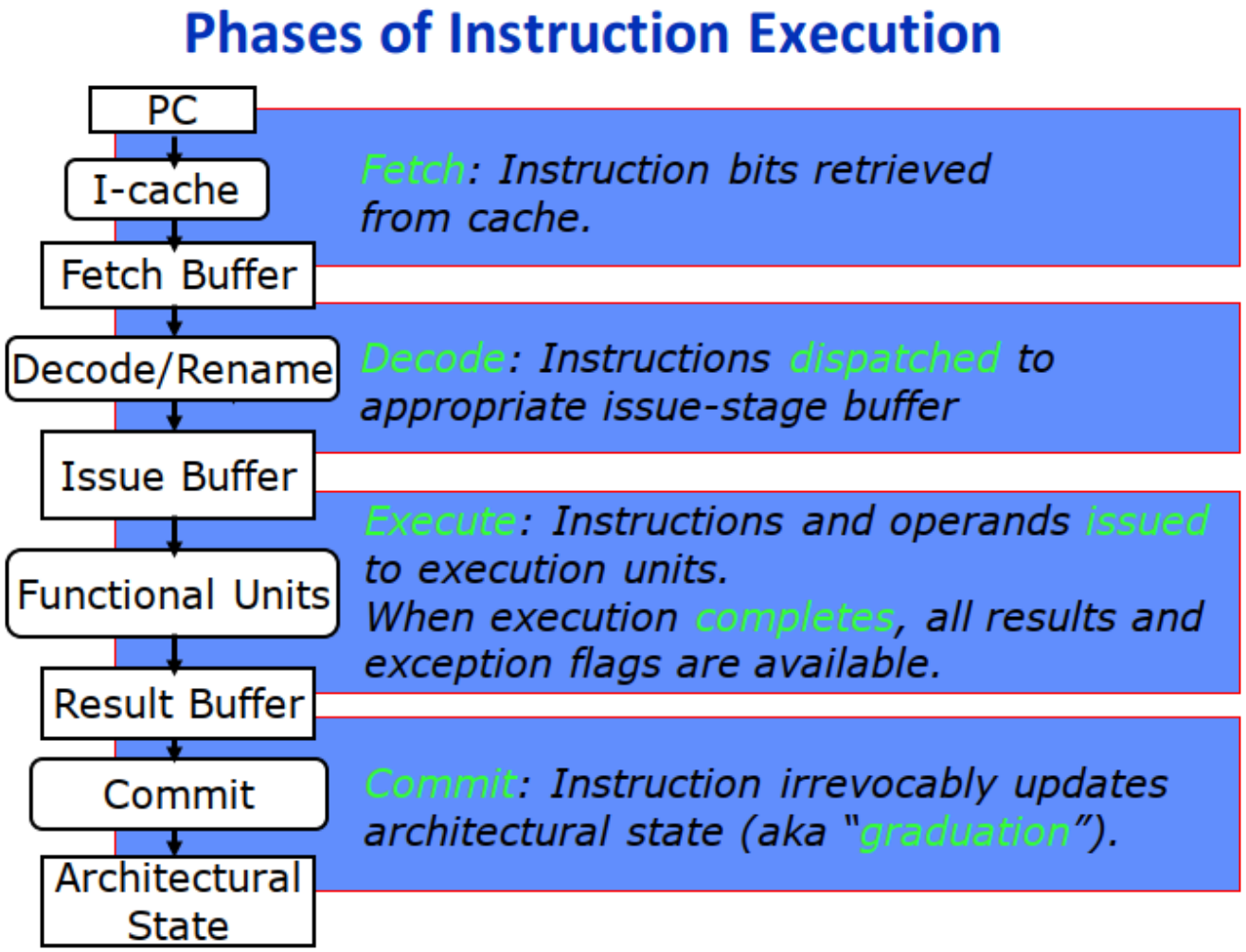
Mr. Robert Tomasulo

source: Wiki

Data Buses in Tomasulo Algorithm

- Normal data bus: data + destination (“go to” bus)
- Common data bus (CDB): data + source (“come from” bus)
  - 64 bits of data + 4 bits of Functional Unit source address
  - Write if matches expected Functional Unit (produces result)
  - Does the broadcast - CDB broadcasts results rather than waiting on registers
    - Functional units can access the result of any operation without involving a floating-point-register, allowing multiple units waiting on a result to proceed without waiting to resolve contention for access to register file read ports.

Summary Again



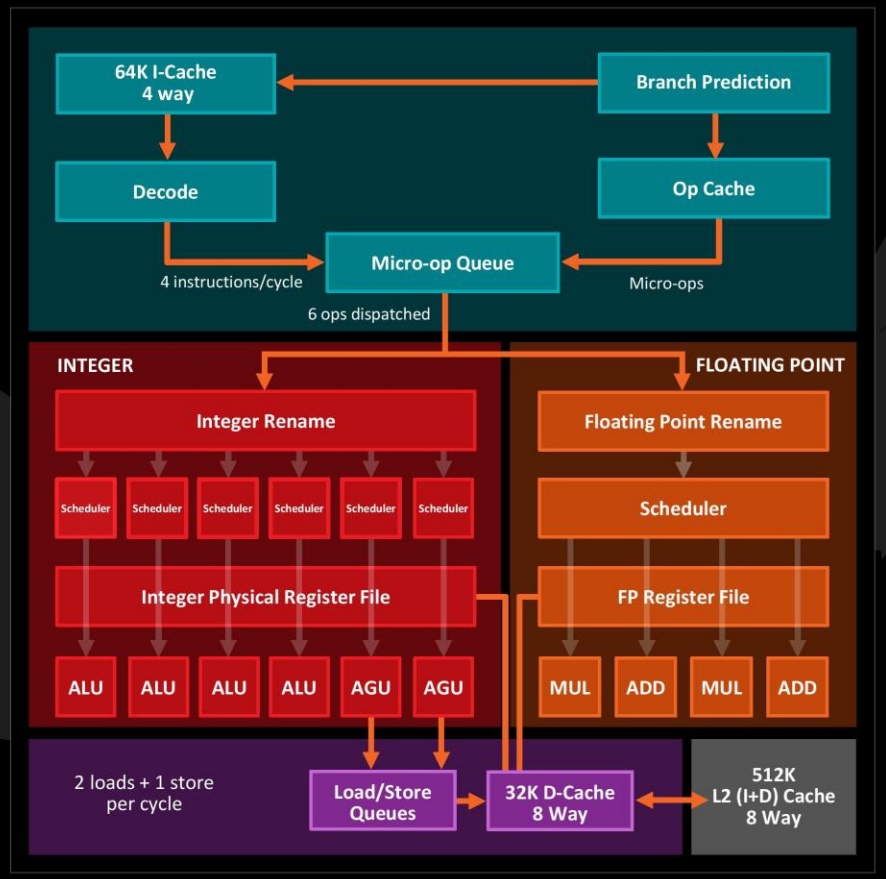
Issue Queue
Reservation Station

ROB

Real-world OoO Examples

Modern OoO desktop/server CPU

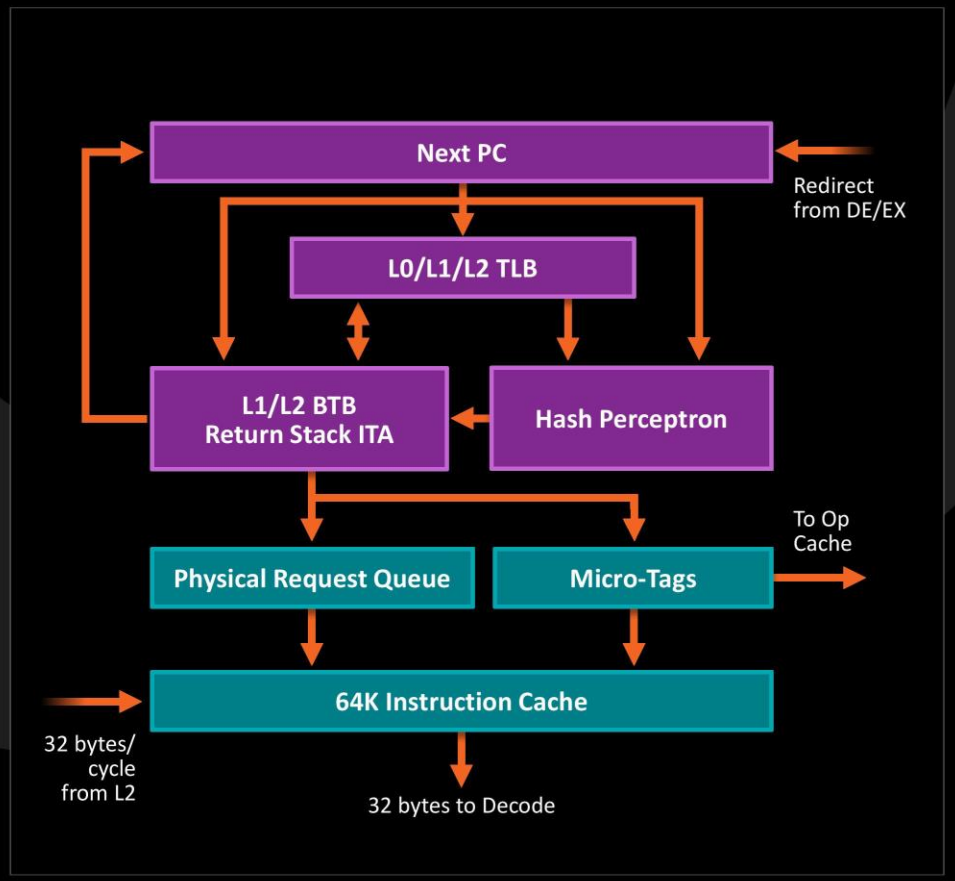
- AMD Zen microarchitecture (2016)
  - AMD slides from HotChips 2016 conference follow



AMD

ZEN MICROARCHITECTURE

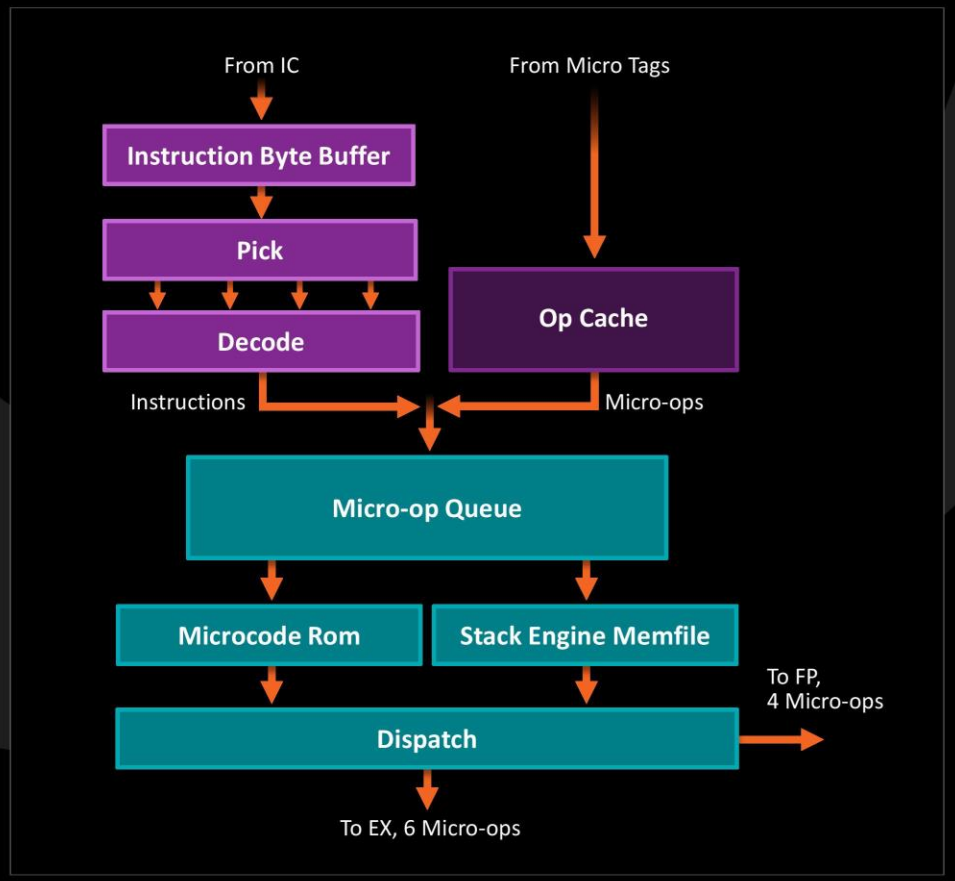
- Fetch Four x86 instructions
- Op Cache instructions
- 4 Integer units
  - Large rename space – 168 Registers
  - 192 instructions in flight/8 wide retire
- 2 Load/Store units
  - 72 Out-of-Order Loads supported
- 2 Floating Point units x 128 FMACs
  - built as 4 pipes, 2 Fadd, 2 Fmul
- I-Cache 64K, 4-way
- D-Cache 32K, 8-way
- L2 Cache 512K, 8-way
- Large shared L3 cache
- 2 threads per core



AMD

FETCH

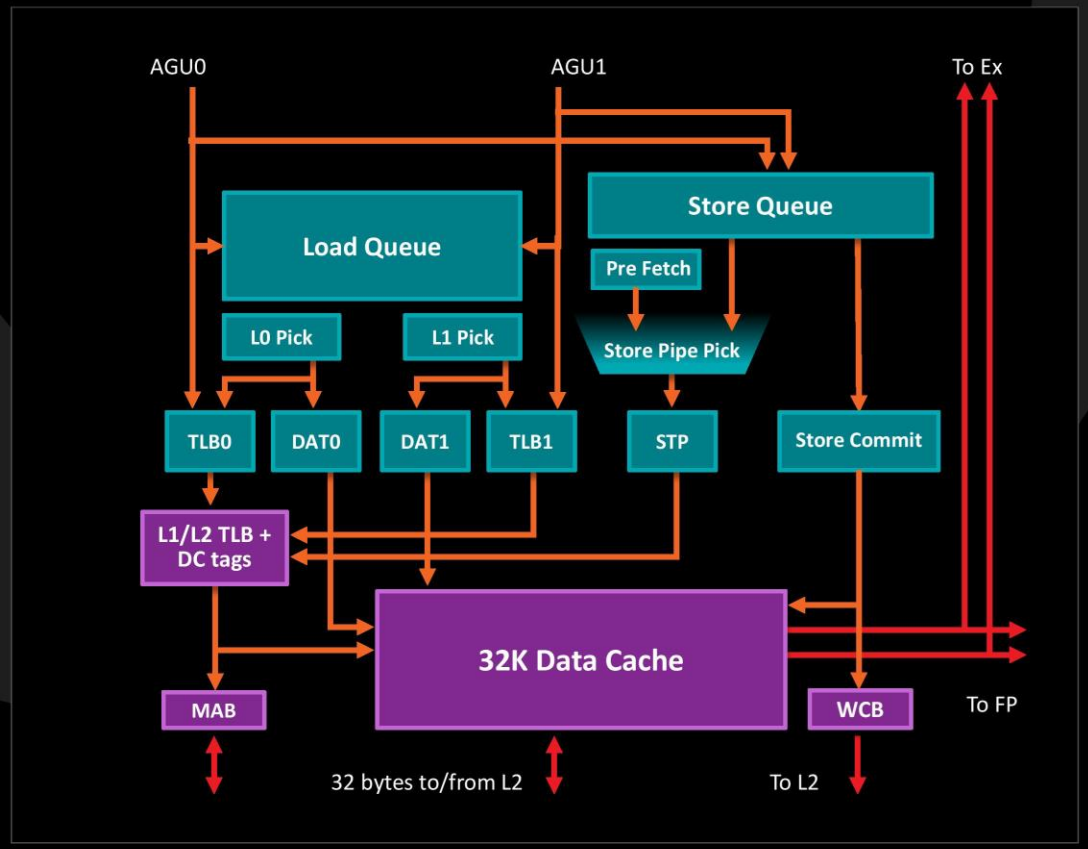
- Decoupled Branch Prediction
- TLB in the BP pipe
  - 8 entry L0 TLB, all page sizes
  - 64 entry L1 TLB, all page sizes
  - 512 entry L2 TLB, no 1G pages
- 2 branches per BTB entry
- Large L1 / L2 BTB
- 32 entry return stack
- Indirect Target Array (ITA)
- 64K, 4-way Instruction cache
- Micro-tags for IC & Op cache
- 32 byte fetch



AMD

DECODE

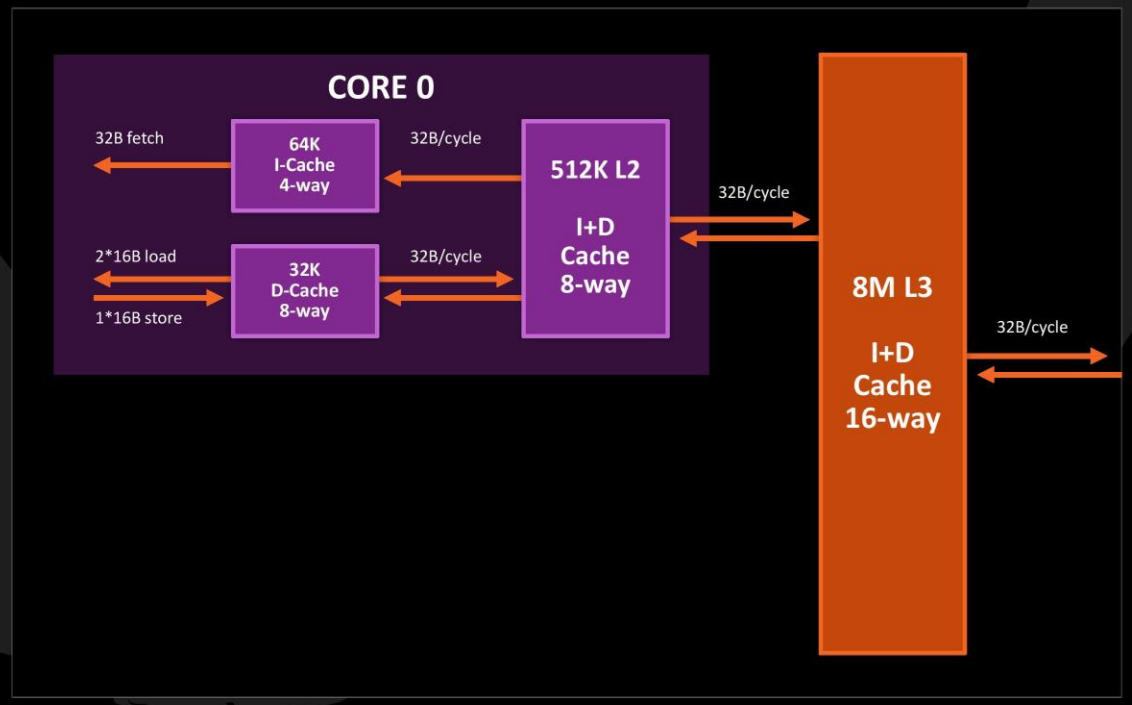
- Inline Instruction-length Decoder
- Decode 4 x86 instructions
- Op cache
- Micro-op Queue
- Stack Engine
- Branch Fusion
- Memory File for Store to Load Forwarding



LOAD/STORE AND L2

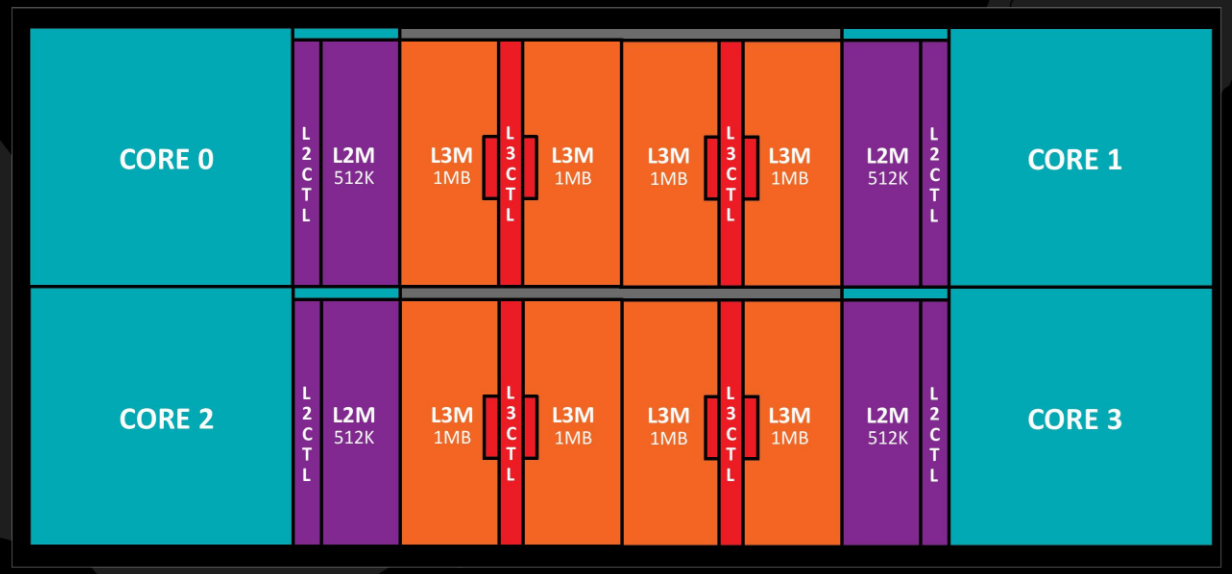
- 72 Out of Order Loads
- 44 entry Store Queue
- Split TLB/Data Pipe, store pipe
- 64 entry L1 TLB, all page sizes
- 1.5K entry L2 TLB, no 1G pages
- 32K, 8 way Data Cache
  - Supports two 128-bit accesses
- Optimized L1 and L2 Prefetchers
- 512K, private (2 threads), inclusive L2

AMD



ZEN CACHE HIERARCHY

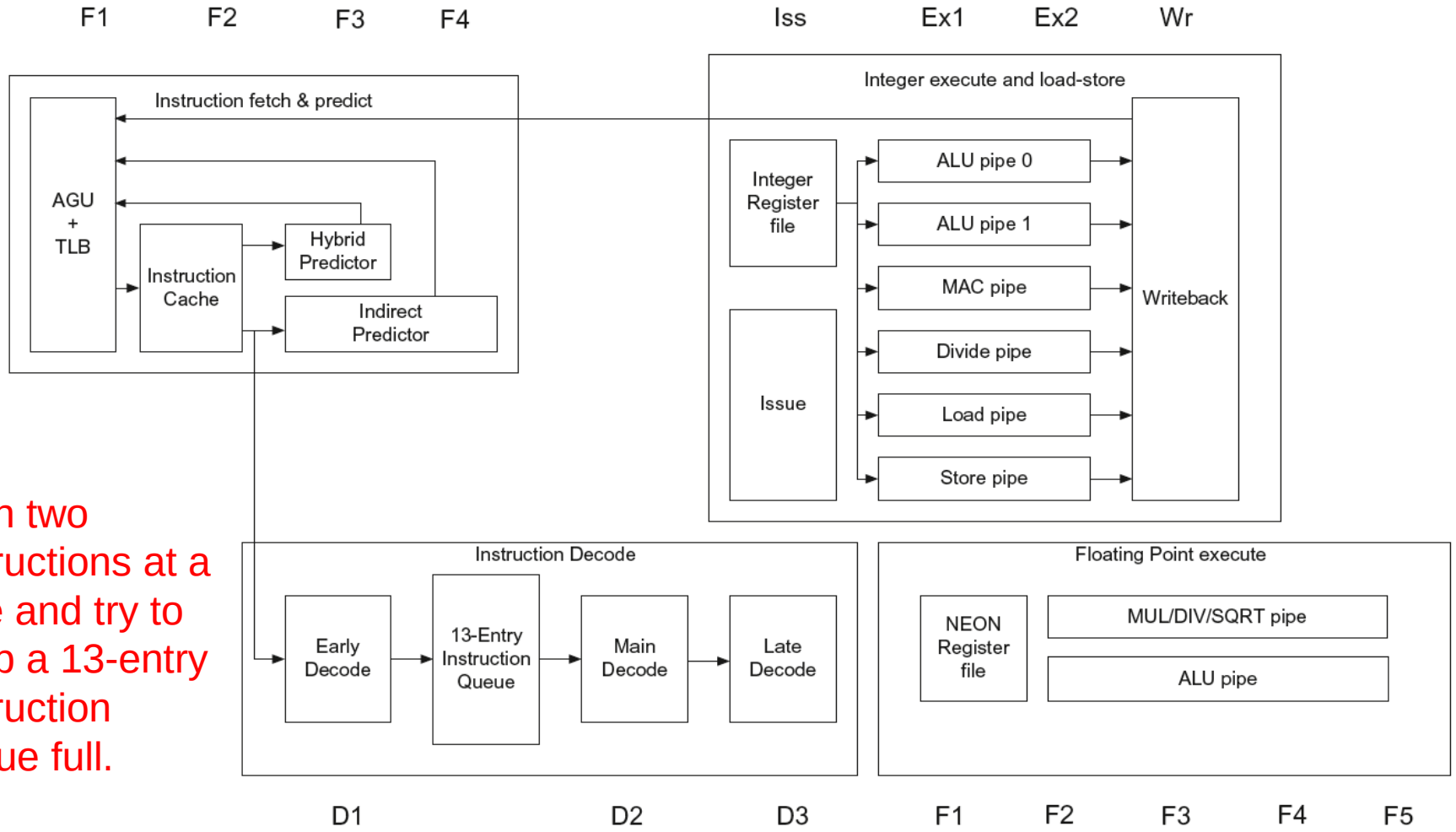
- Fast private 512K L2 cache
- Fast shared L3 cache
- High bandwidth enables prefetch improvements
- L3 is filled from L2 victims
- Fast cache-to-cache transfers
- Large Queues for Handling L1 and L2 misses



CPU COMPLEX

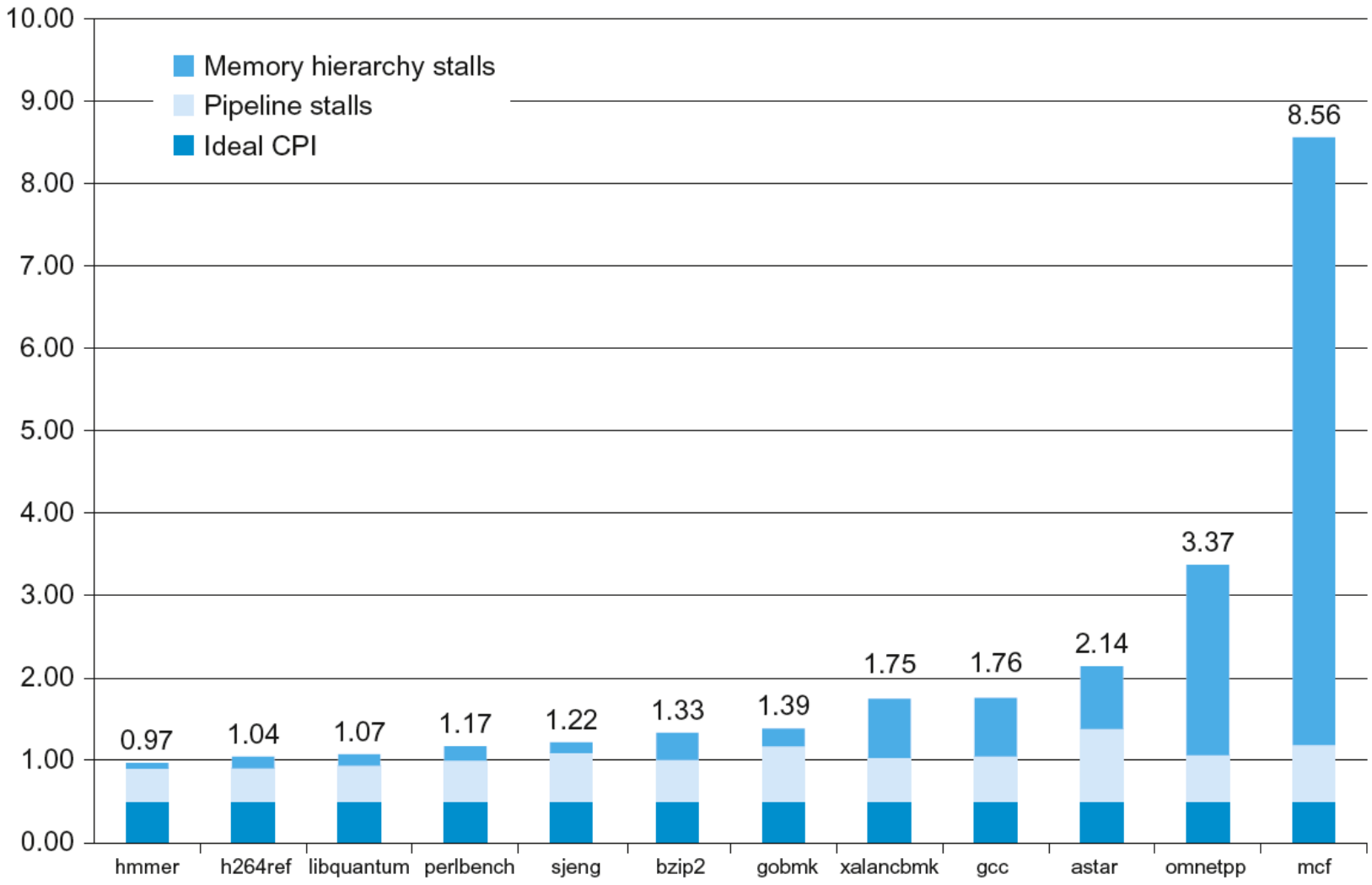
- A CPU complex (CCX) is four cores connected to an L3 Cache.
- The L3 Cache is 16-way associative, 8MB, mostly exclusive of L2.
- The L3 Cache is made of 4 slices, by low-order address interleave.
- Every core can access every cache with same average latency

ARM Cortex-A53 Pipeline



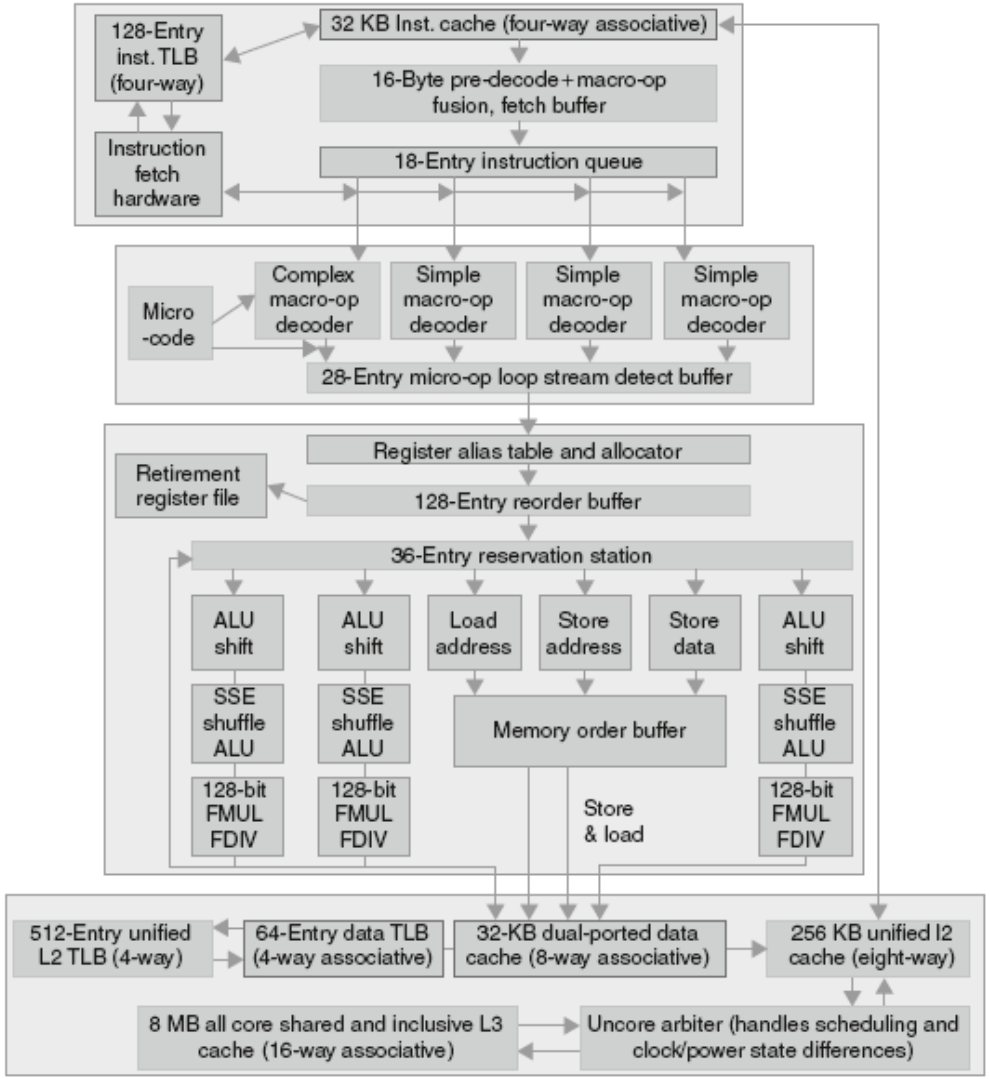
fetch two instructions at a time and try to keep a 13-entry instruction queue full.

ARM Cortex-A53 Performance

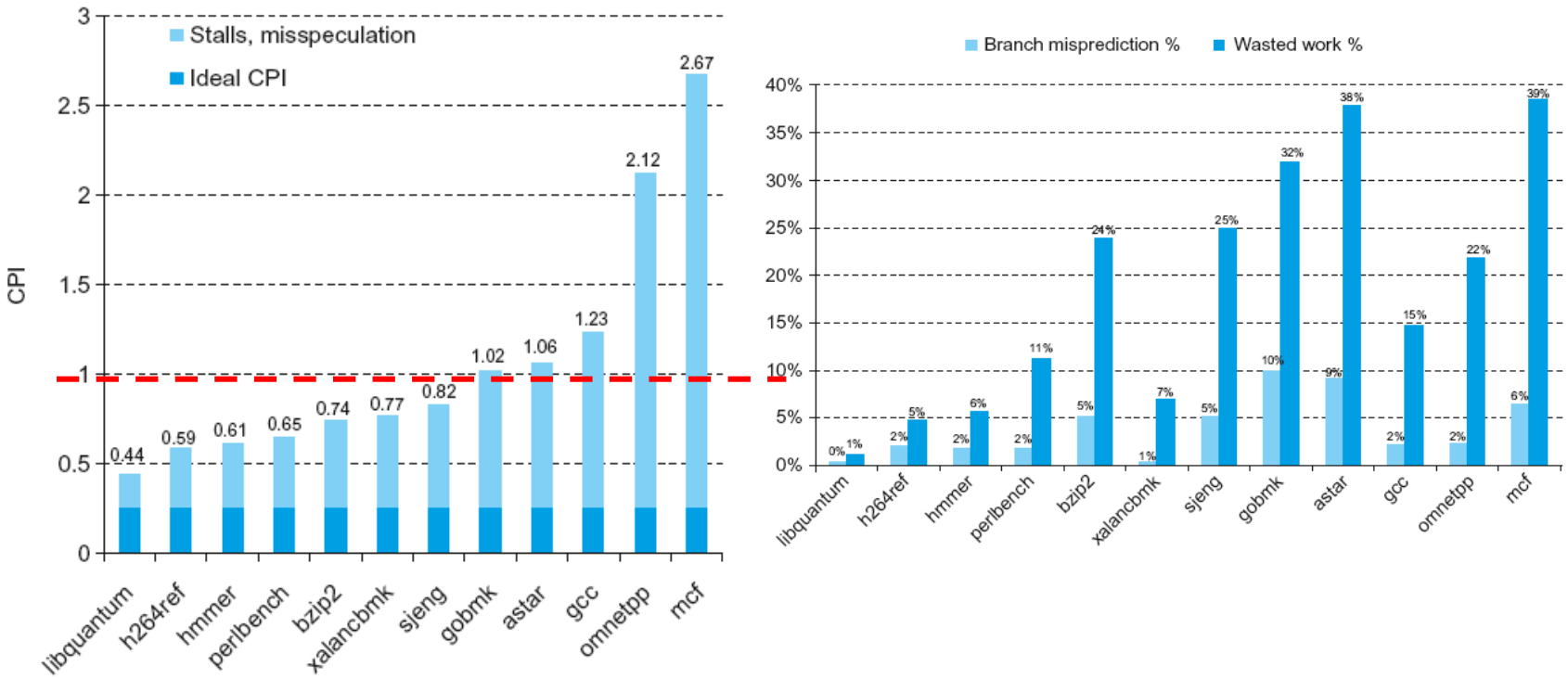


Core i7 Pipeline

- 6 separate functional units
  - 3x ALU
  - 3 for memory operations
- 20-24 stage pipeline
- Aggressive branch prediction and other optimizations
- Massive out-of-order capability
  - Can reorder up to 128 micro-operation instructions!
- And yet it still barely averages a 1 on CPI!



Core i7 Performance

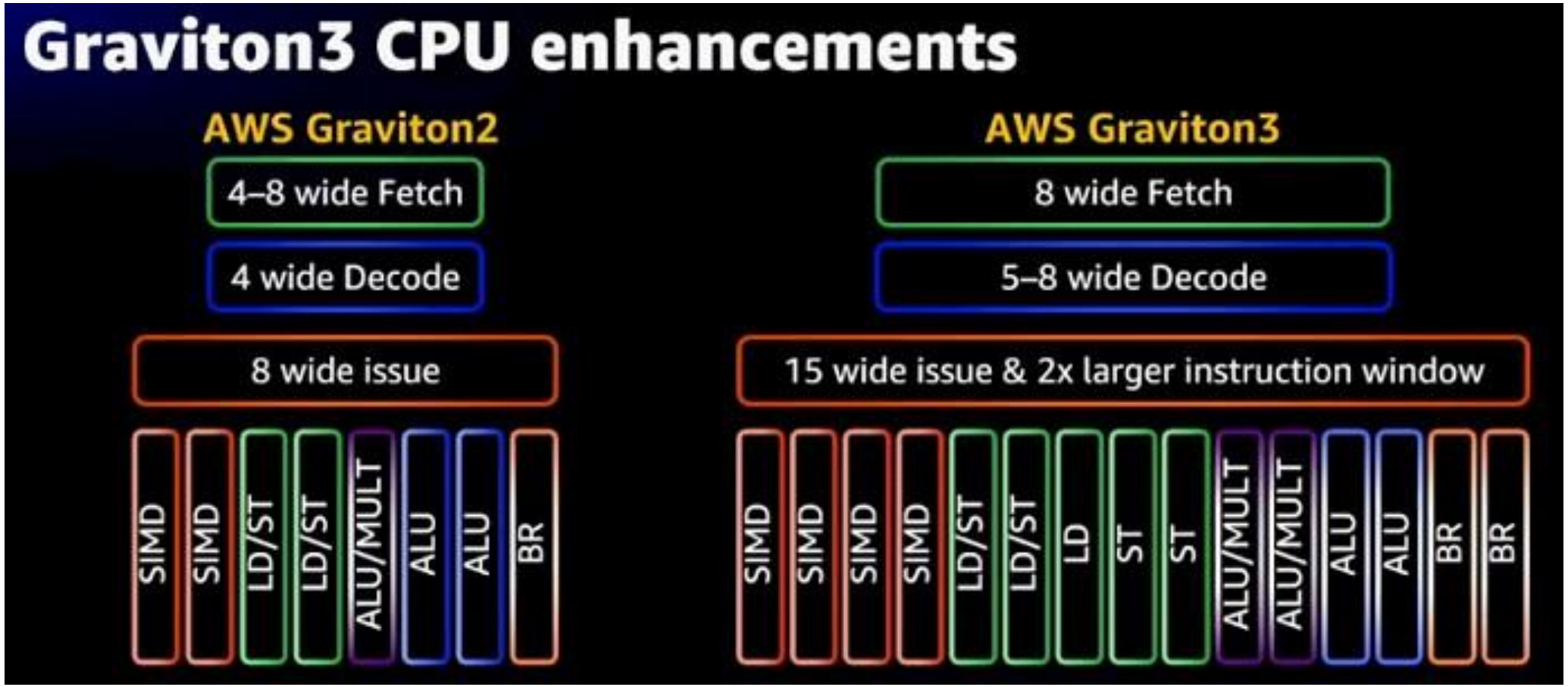


Power Efficiency

- Complexity of dynamic scheduling and speculations requires power
- Multiple simpler cores may be better

Microprocessor	Year	Clock Rate	Pipeline Stages	Issue width	Out-of-order/ Speculation	Cores	Power
i486	1989	25MHz	5	1	No	1	5W
Pentium	1993	66MHz	5	2	No	1	10W
Pentium Pro	1997	200MHz	10	3	Yes	1	29W
P4 Willamette	2001	2000MHz	22	3	Yes	1	75W
P4 Prescott	2004	3600MHz	31	3	Yes	1	103W
Core	2006	2930MHz	14	4	Yes	2	75W
UltraSparc III	2003	1950MHz	14	4	No	1	90W
UltraSparc T1	2005	1200MHz	6	1	No	8	70W

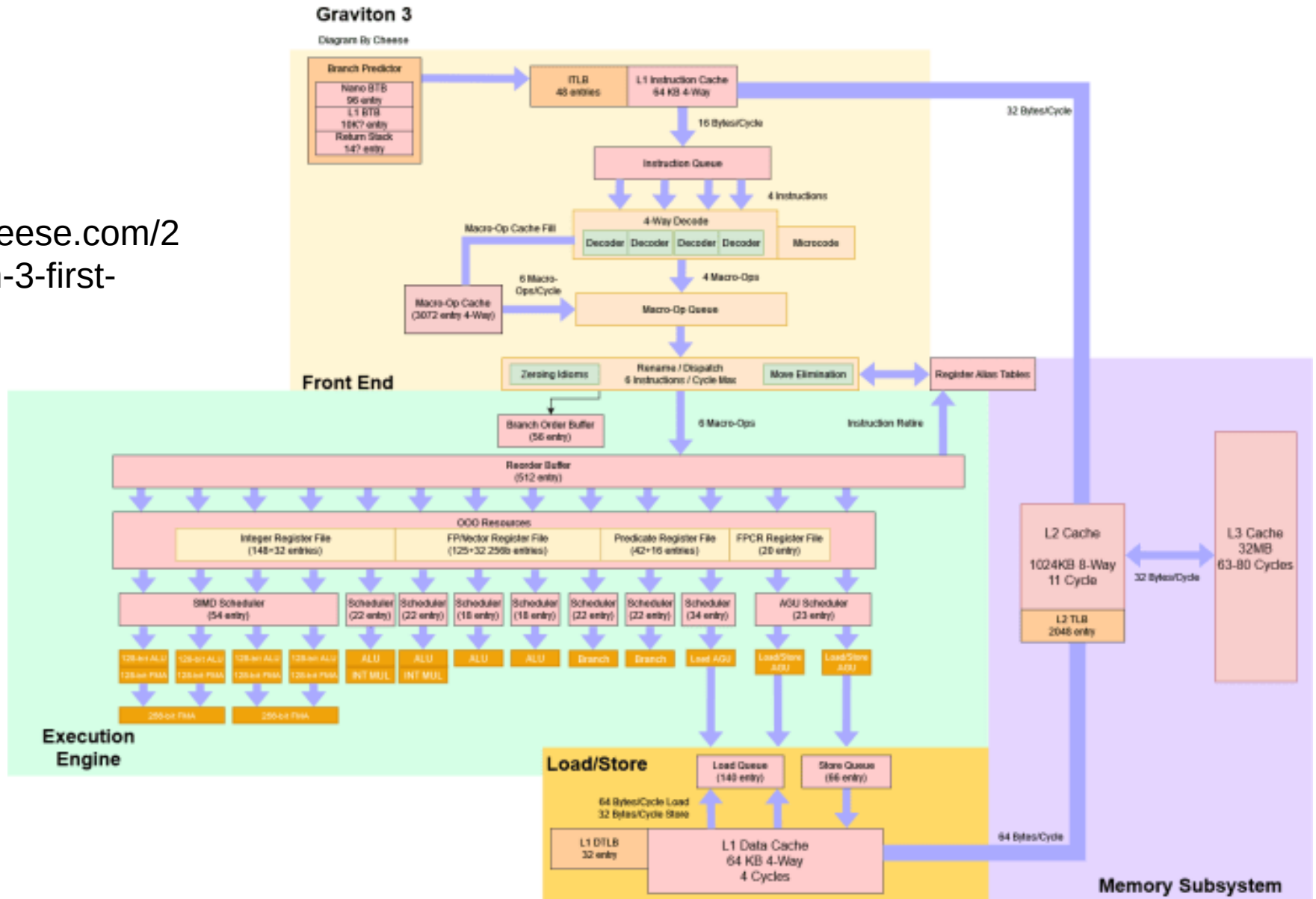
Amazon Graviton 3 (May 2022)



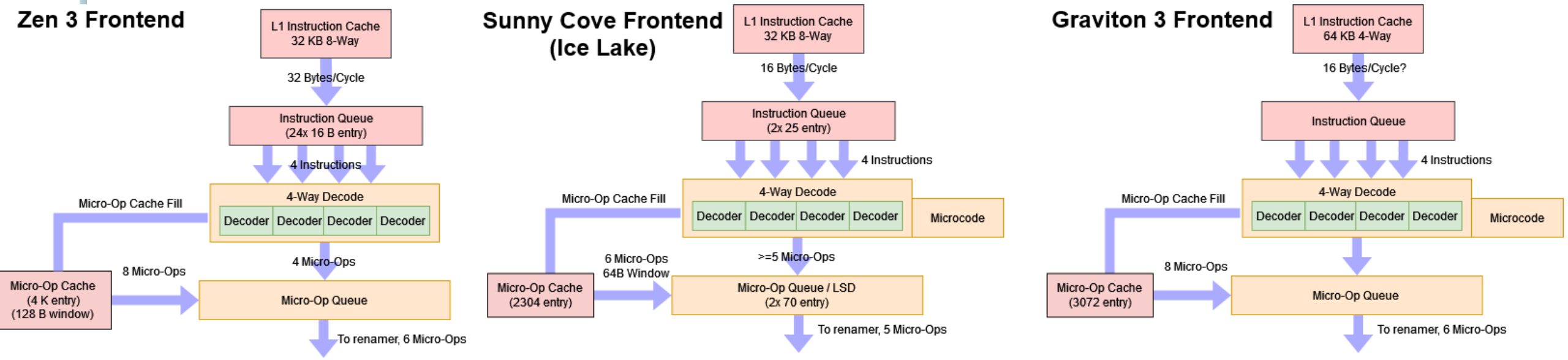
https://www.nextplatform.com/2022/01/04/inside-amazons-graviton3-arm-server-processor/

Amazon Graviton 3 (May 2022)

https://chipsandcheese.com/2022/05/29/graviton-3-first-impressions/

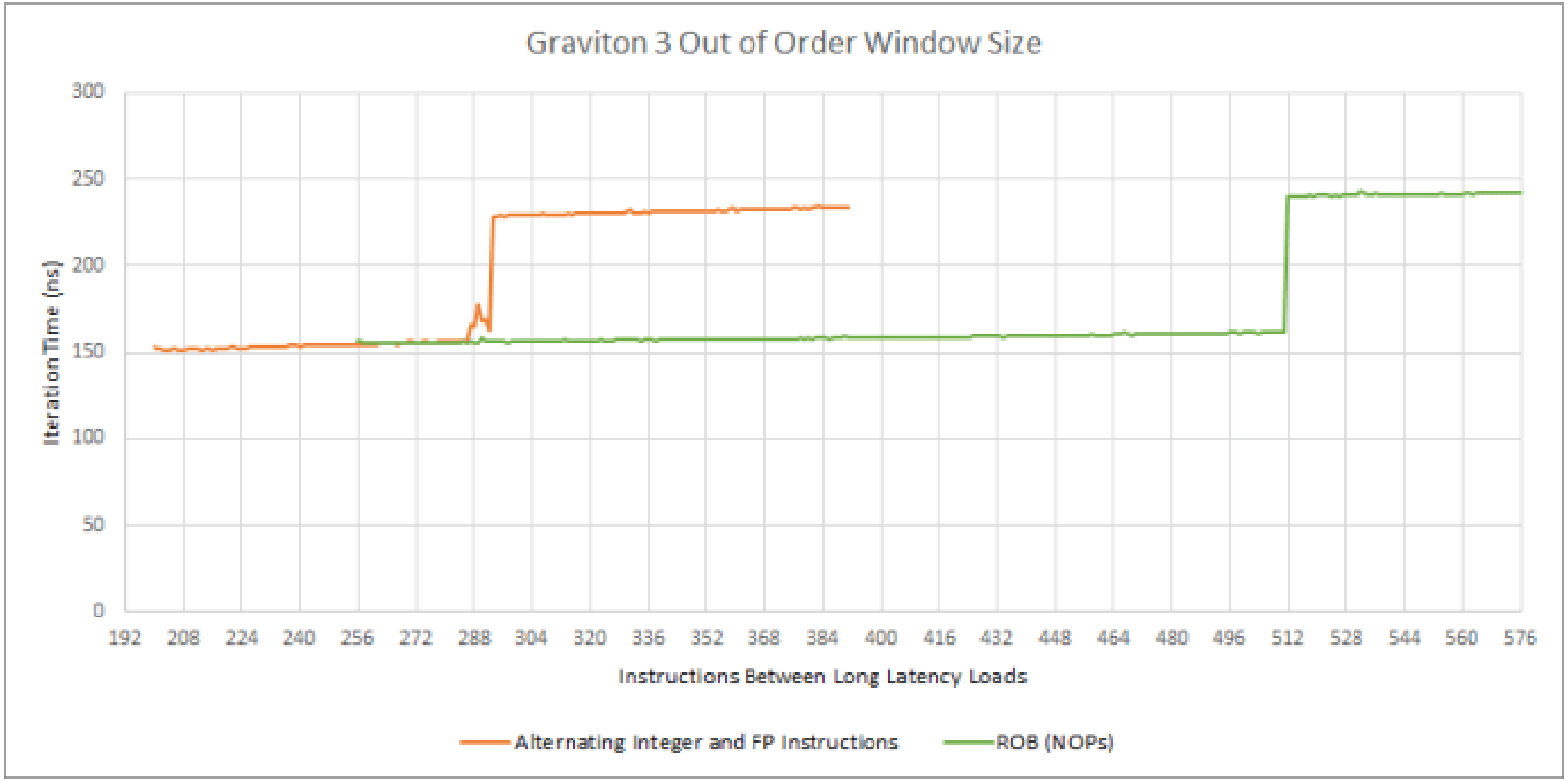


Amazon Graviton 3 (May 2022)



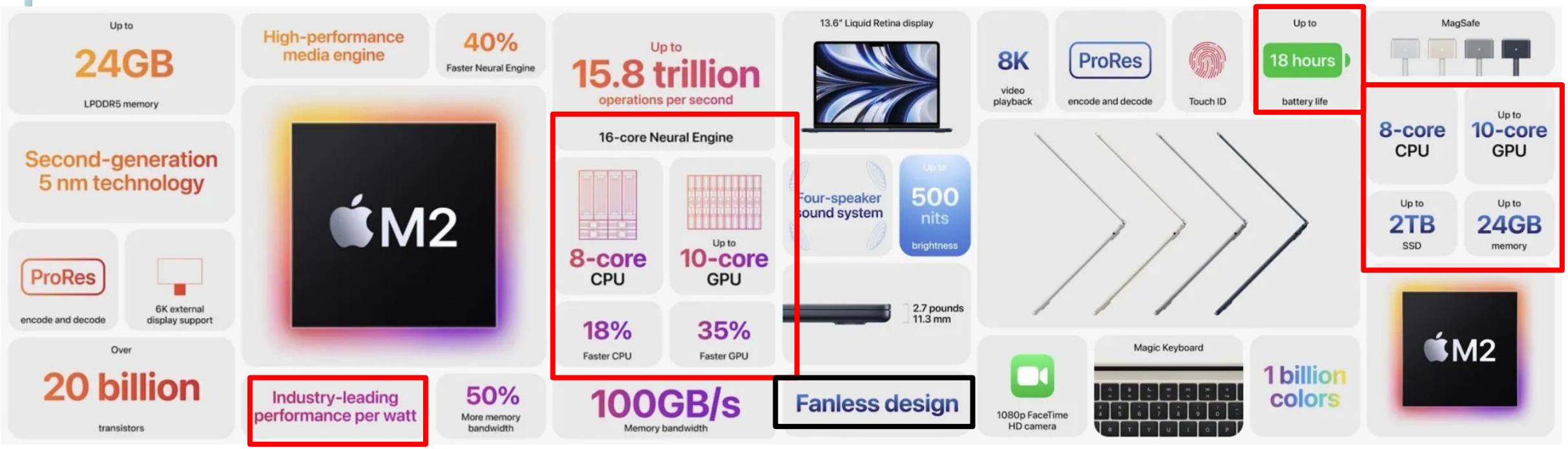
https://chipsandcheese.com/2022/05/29/graviton-3-first-impressions/

Amazon Graviton 3 (May 2022)



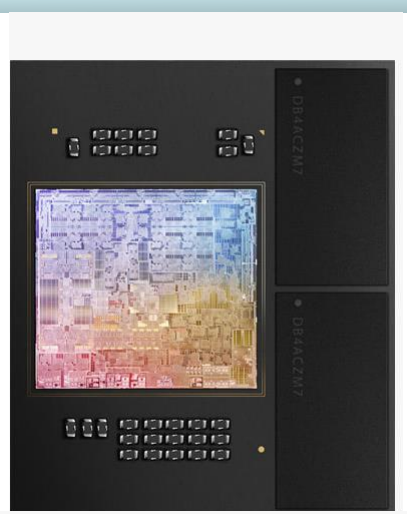
https://chipsandcheese.com/2022/05/29/graviton-3-first-impressions/

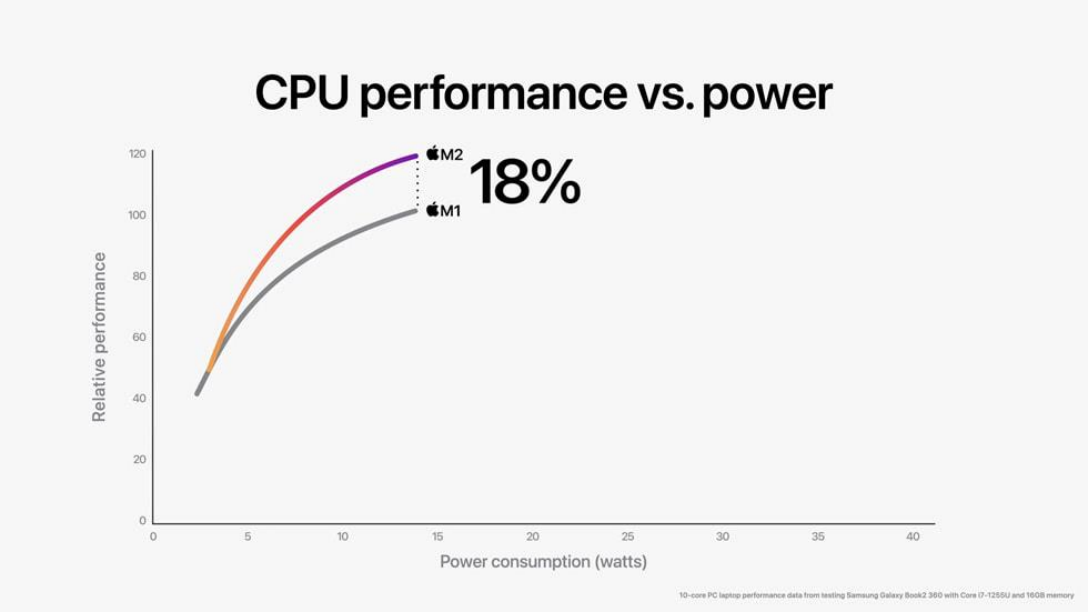
Apple M2 (June 2022)



source: Apple

Apple M2 (June 2022)





- More Transistors, More Memory
  - 18 percent greater multithreaded performance than M1
  - 20 billion transistors — 25 percent more than M1
  - with up to 24GB of fast unified memory, M2 can handle even larger and more complex workloads.
  - The higher performance per watt from M2 enables systems to have exceptional battery life, and run cool and quietly, even when playing graphics-intensive games or editing massive RAW images.

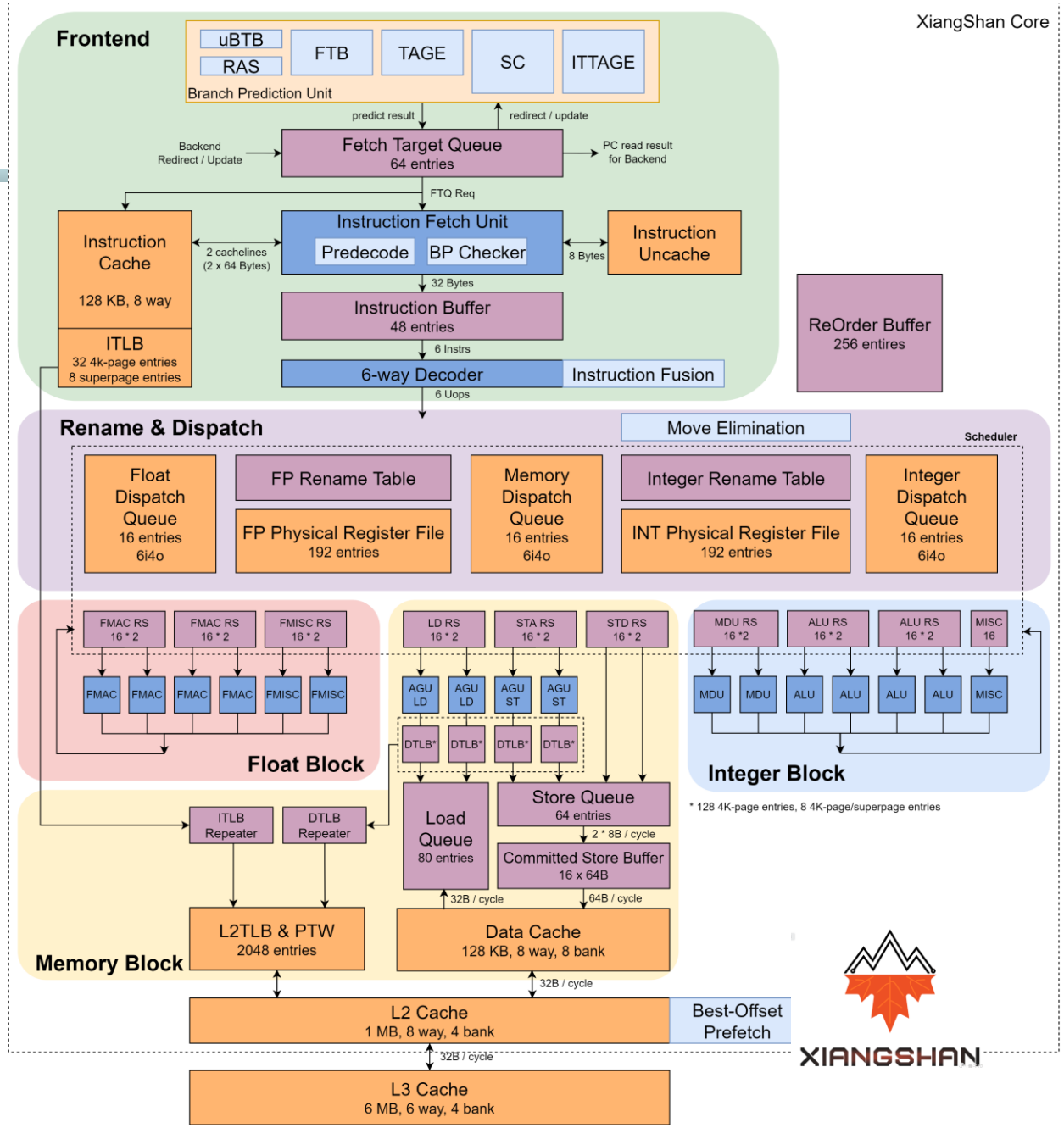
https://www.apple.com/newsroom/2022/06/apple-unveils-m2-with-breakthrough-performance-and-capabilities/

source: Apple

Xiangshan

Xiangshan: open-source high-performance RISC-V processor project developed in Institute of Computing Technology, Chinese Academy of Sciences

- **11**-stage, **6**-wide decode/rename
- **TAGE-SC-L** branch prediction
- **160** Int PRF + **160** FP PRF
- **192**-entry ROB, **64**-entry LQ, **48**-entry SQ
- **16**-entry RS for each FU
- **16KB** L1 Cache, **128KB** L1plus Cache for instruction
- **32KB** L1 Data Cache
- **32**-entry ITLB/DTLB, **4K**-entry STLB
- **1MB** inclusive L2 Cache



Source: https://github.com/OpenXiangShan/XiangShan

Limits to Superscalar Processors

Ultimately, the performance of a superscalar processor is limited by:

- Increasing hardware cost of extracting more ILP
- Memory bandwidth
  - Hard to keep pipelines full
- Limits to branch prediction and caches
  - Some dependencies are hard to eliminate
- Interconnect scaling
- Power consumption
- Some parallelism is hard to expose
  - Limited window size during instruction issue

OoO Summary

- **Advantages**
  - Help exploit Instruction Level Parallelism (ILP)
  - Help hide latencies (e.g., cache miss, divide)
  - Superior/complementary to inst. Scheduler in the compiler
    - Dynamic instruction window
- **Challenges**
  - Complex wakeup logic (instruction scheduler)
  - Complex logic needed to recover from mis-prediction
  - Runtime cost incurred when recovering from a mis-prediction

Where are we Heading?

- T5: Memory I

Acknowledgement

Slides in this topic are inspired in part by material developed and copyright by:

- ARM Courseware
- Prof. Shuai Wang @ Nanjing U
- Prof. Joe Devietti @ Upenn, CIS 571
- Prof. Ron Dreslinski @ UMich, EECS 470
- Prof. Hakim Weatherspoon @ Cornell, CS 3410
- Prof. Krste Asanovic @ UCB, CS252
- Xinfei Guo @ JI, VE370 2021 SU

Action Items

- HW#2 is due
- Reading Materials
  - Ch. 3.9 - 3.12
  - Ch. Appendix C.5, C.7