Digital Design & Computer Arch.

Lecture 17a: Dataflow & Superscalar Execution

Prof. Onur Mutlu

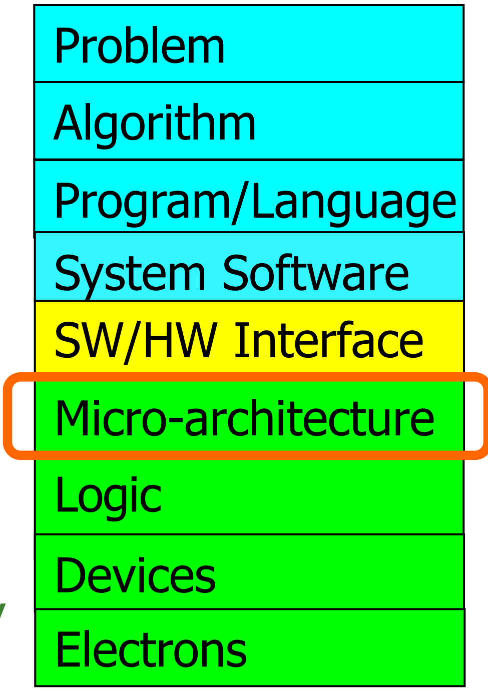
ETH Zürich

Spring 2022

29 April 2022

Roadmap for Today (and Past 2-3 Weeks)

- Prior to last week: Microarchitecture Fundamentals
  - Single-cycle Microarchitectures
  - Multi-cycle Microarchitectures
- Last week: Pipelining & Precise Exceptions
  - Pipelining
  - Pipelined Processor Design
    - Control & Data Dependence Handling
    - Precise Exceptions: State Maintenance & Recovery
- This+next week: Out-of-Order & Superscalar Execution
  - Out-of-Order Execution
  - Dataflow & Superscalar Execution
  - Branch Prediction



Readings

- This week
  - Smith and Sohi, "The Microarchitecture of Superscalar Processors," Proceedings of the IEEE, 1995
  - H&H Chapters 7.8 and 7.9
  - McFarling, "Combining Branch Predictors," DEC WRL Technical Report, 1993.
  - Optional: Kessler, "The Alpha 21264 Microprocessor," IEEE Micro 1999.

Out-of-Order Execution (Restricted Dataflow) **Wrap Up**

Recall: OoO Execution w/ Precise Exceptions

- Most modern processors use the following
- Reorder buffer to support in-order retirement of instructions
- A single register file (physical RF) to store **all registers**
  - Both speculative and architectural registers
  - INT and FP are still separate
- Two register **maps** store **pointers** to the physical RF
  - Future/frontend register map → used for renaming
  - Architectural register map → used for maintaining precise state
- This design avoids value replication in RSs, ROB, etc.

Recall: OoO Execution w/ Precise Exceptions (II)

Pointers to PRF

Register	PR
R1	18
R2	13
R3	10
R4	22
R5	14
R6	19
R7	17
R8	20
R9	3
R10	4
R11	1

Frontend Register Map

PR	Value
PR1	1
PR2	2
PR3	3
PR4	4
PR5	5
PR6	6
PR7	7
PR8	8
PR9	9
PR10	10
PR11	11
PR12	12
PR13	13
PR14	14
PR15	15
PR16	16
PR17	17
PR18	18
PR19	19
PR20	20
PR21	21
PR22	22

Physical Register File (PRF)

Reorder Buffer (ROB)

Entry 0					
Entry 1					
Entry 2					
Entry 8					
Entry 13					
Entry 14					
Entry 15					

Centralized Value Storage

Pointers to PRF

Register	PR
R1	12
R2	2
R3	10
R4	22
R5	5
R6	9
R7	11
R8	20
R9	7
R10	6
R11	1

Architectural Register Map

Recall: OoO Execution w/ Precise Exceptions (III)

At Decode/Rename: Allocate DestPR to Architectural DestReg (RS, ROB)
At Decode/Rename: Read and Update **Frontend Register Map**

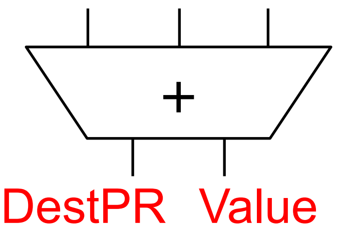
RS for ADD Unit

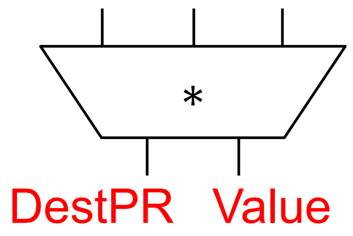
	Source 1	Source 2
	PR	PR
a		
b		
c		
d		

RS for MUL Unit

	Source 1	Source 2
	PR	PR
a		
b		
c		
d		

Before Execution: Access **Physical Register File** to Get Source Values

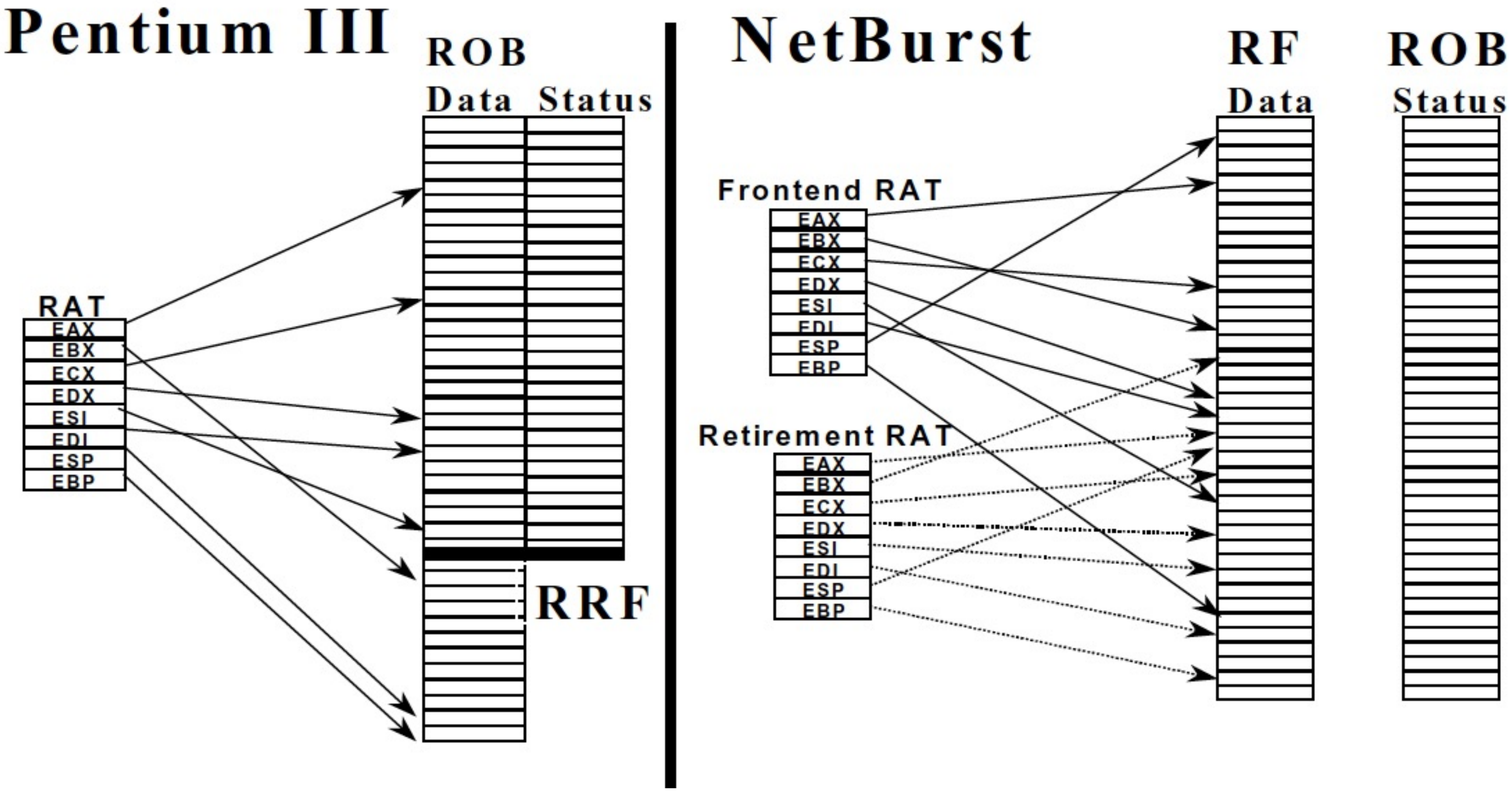




After Execution: Access **Physical Register File** to Write Result Values

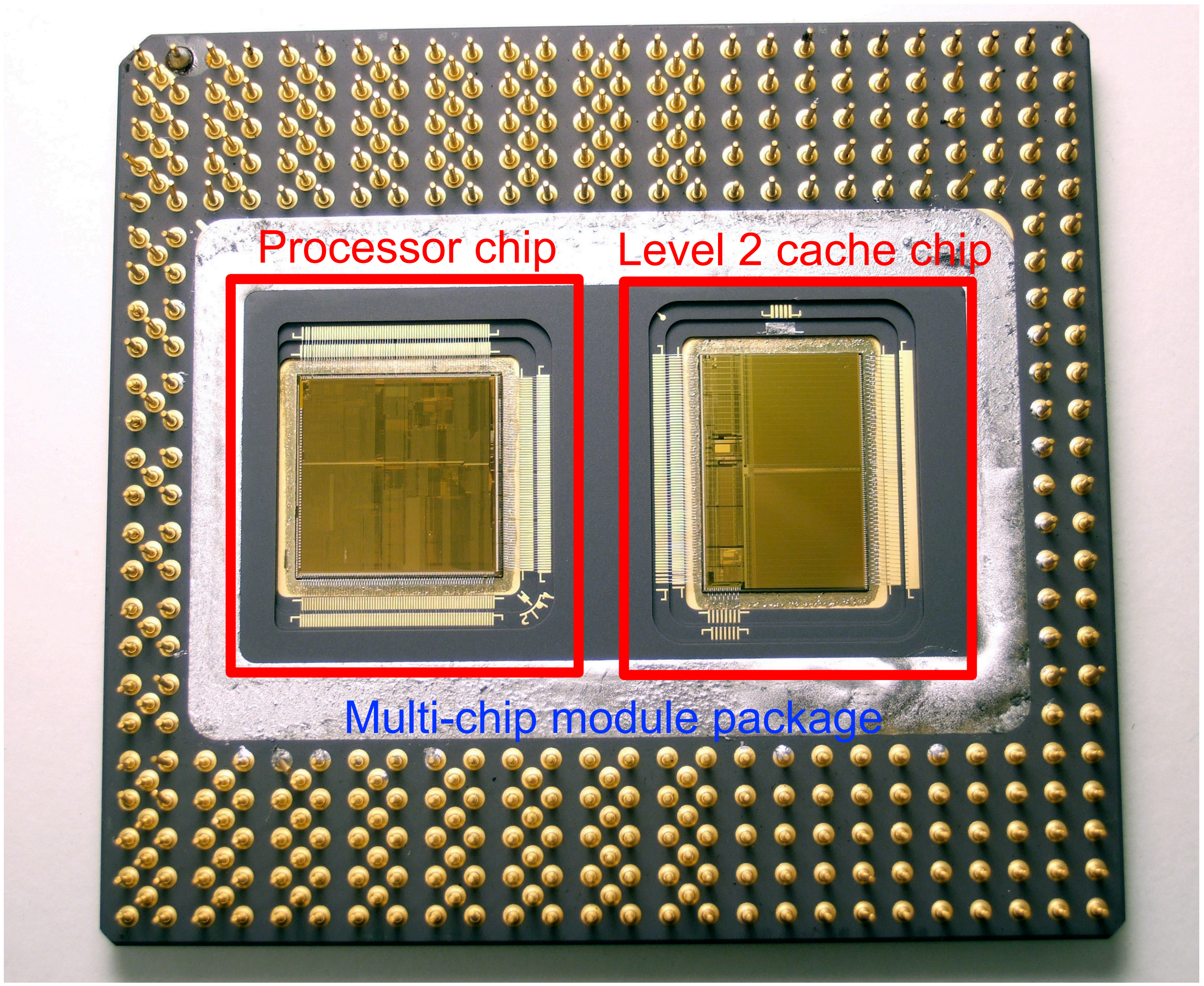
At Retirement: Update Architectural Register Map with DestPR

Recall: Examples from Modern Processors



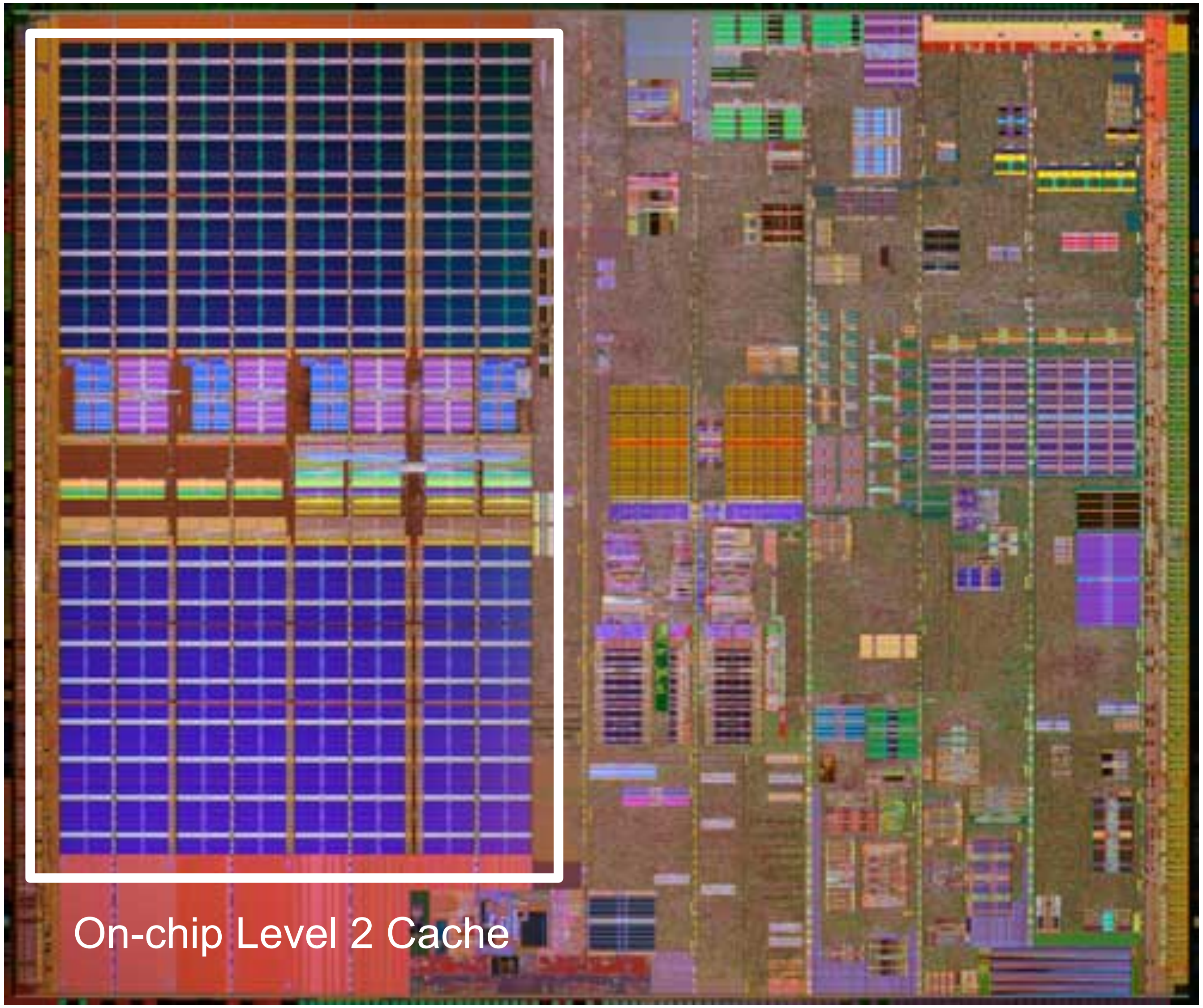
Boggs et al., "The Microarchitecture of the Pentium 4 Processor,"
Intel Technology Journal, 2001.

Intel Pentium Pro (1995)



By Moshen - http://en.wikipedia.org/wiki/Image:Pentiumpro_moshen.jpg, CC BY-SA 2.5, https://commons.wikimedia.org/w/index.php?curid=2262471

Intel Pentium 4 (2000)



https://www.anandtech.com/show/1621/3

Enabling OoO Execution, Revisited

1. Link the consumer of a value to the producer
   - Register renaming: Associate a “tag” with each data value

2. Buffer instructions until they are ready
   - Insert instruction into reservation stations after renaming

3. Keep track of readiness of source values of an instruction
   - Broadcast the “tag” when the value is produced
   - Instructions compare their “source tags” to the broadcast tag → if match, source value becomes ready

4. When all source values of an instruction are ready, dispatch the instruction to functional unit (FU)
   - Wakeup and select/schedule the instruction

Summary of OOO Execution Concepts

- Register renaming eliminates false dependences, enables linking of producer to consumers
- Buffering in reservation stations enables the pipeline to move (i.e., not stall) for independent instructions
- Tag broadcast enables communication (of readiness of produced value) between instructions
- Wakeup and select enables out-of-order dispatch

OOO Execution: Restricted Dataflow

- An out-of-order engine dynamically builds the dataflow graph of a piece of the program

- The dataflow graph is limited to the **instruction window**
  - Instruction window: all decoded but not yet retired instructions

- Can we do it for the whole program?
  - In other words, how can we have a large instruction window?
  - Why would we like to?
- Can we do it efficiently with Tomasulo's algorithm?

Recall: State of RAT and RS in Cycle 7

Slightly harder tasks for you:

1. **Draw the dataflow graph for the executing code**
2. **Provide the executing code in sequential order**

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	0	x	
R4	1		4
R5	0	d	
R6	1		6
R7	0	b	
R8	1		8
R9	1		9
R10	0	c	
R11	0	y	

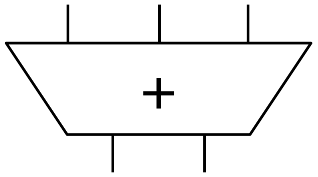
Register Alias Table

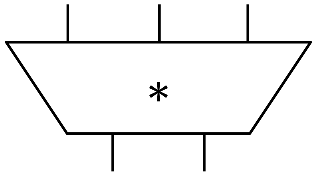
RS for ADD Unit

	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
a	0	x		1	~	4
b	1	~	2	1	~	6
c	1	~	8	1	~	9
d	0	a		0	y	

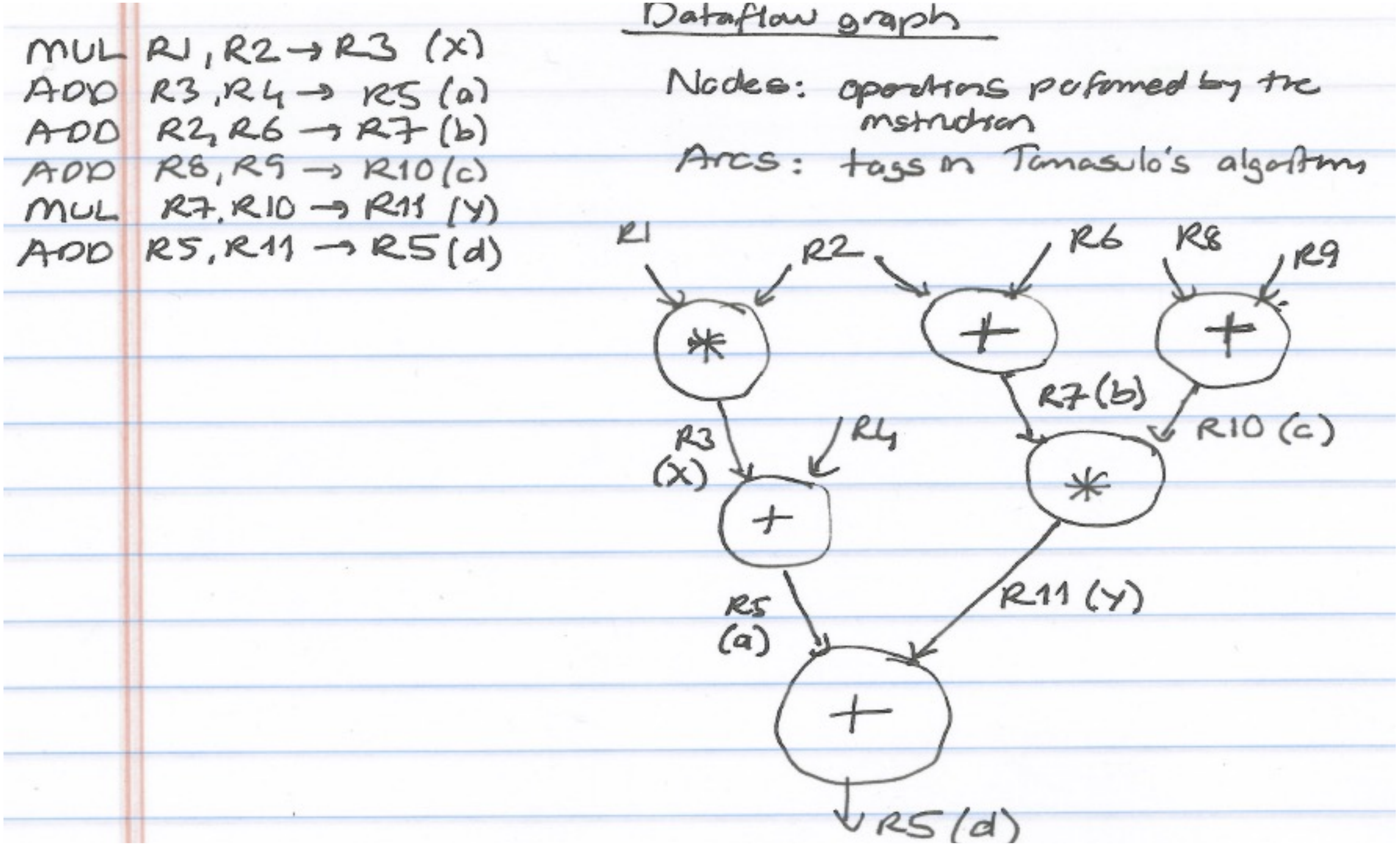
RS for MUL Unit

	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
x	1	~	1	1	~	2
y	0	b		0	c	
z						
t						





Recall: Reverse Engineered Dataflow Graph



```
MUL  R1,R2 → R3 (x)
ADD  R3,R4 → R5 (a)
ADD  R2,R6 → R7 (b)
ADD  R8,R9 → R10 (c)
MUL  R7,R10 → R11 (y)
ADD  R5,R11 → R5 (d)
```

Dataflow graph

Nodes: operations performed by the instruction

Arcs: tags in Tomasulo's algorithm

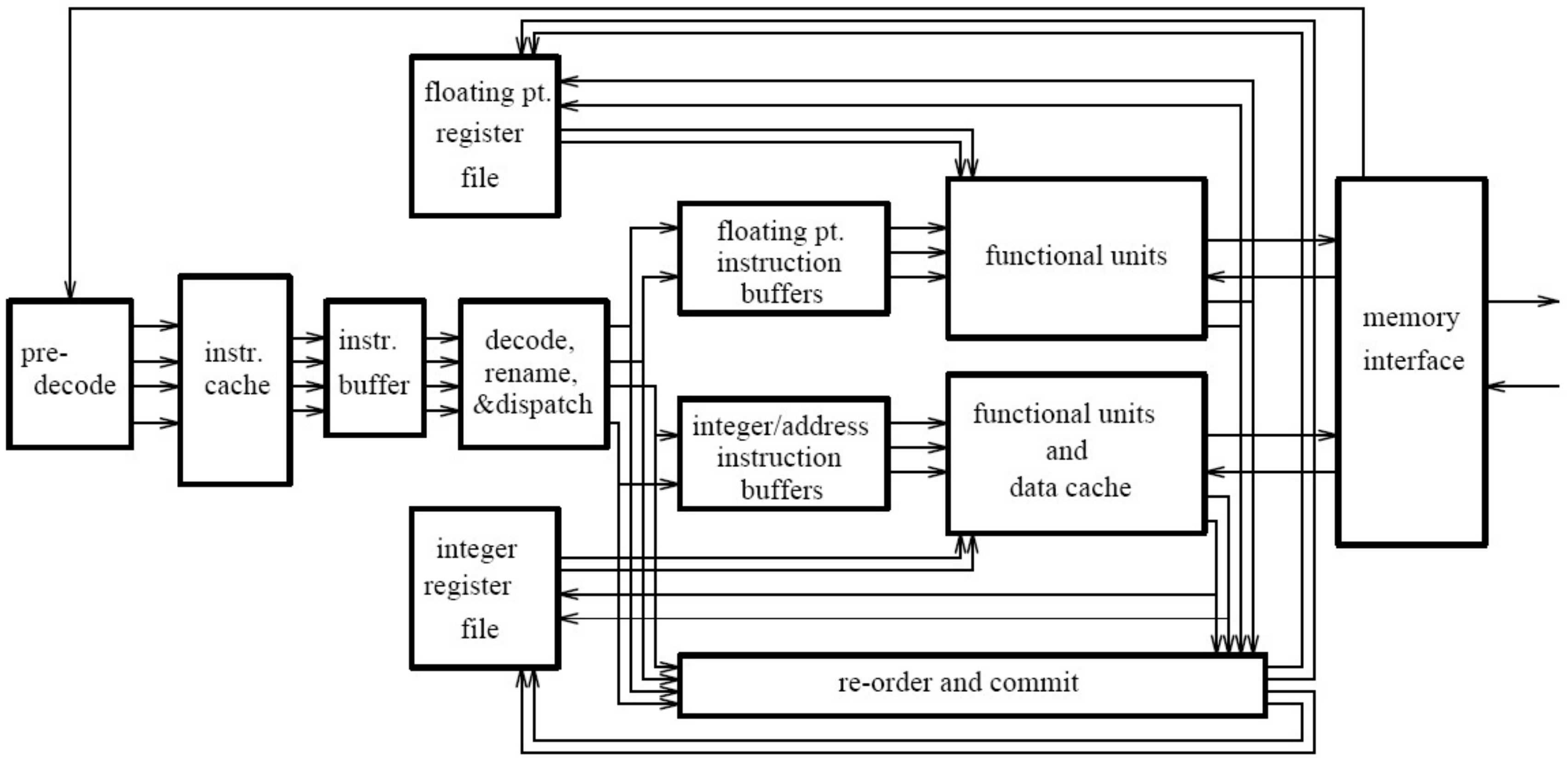
We can "easily" reverse-engineer the dataflow graph of the executing code!

Questions to Ponder

- Why is OoO execution beneficial?
  - Latency tolerance: OoO execution tolerates the latency of multi-cycle operations by executing independent operations concurrently
  - What if all operations take a single cycle?

- What if an instruction takes 1000 cycles?
  - How large of an instruction window do we need to continue decoding?
  - How many cycles of latency can OoO tolerate?
  - What limits the latency tolerance scalability of Tomasulo's algorithm?
    - Instruction window size: how many decoded but not yet retired instructions you can keep in the machine

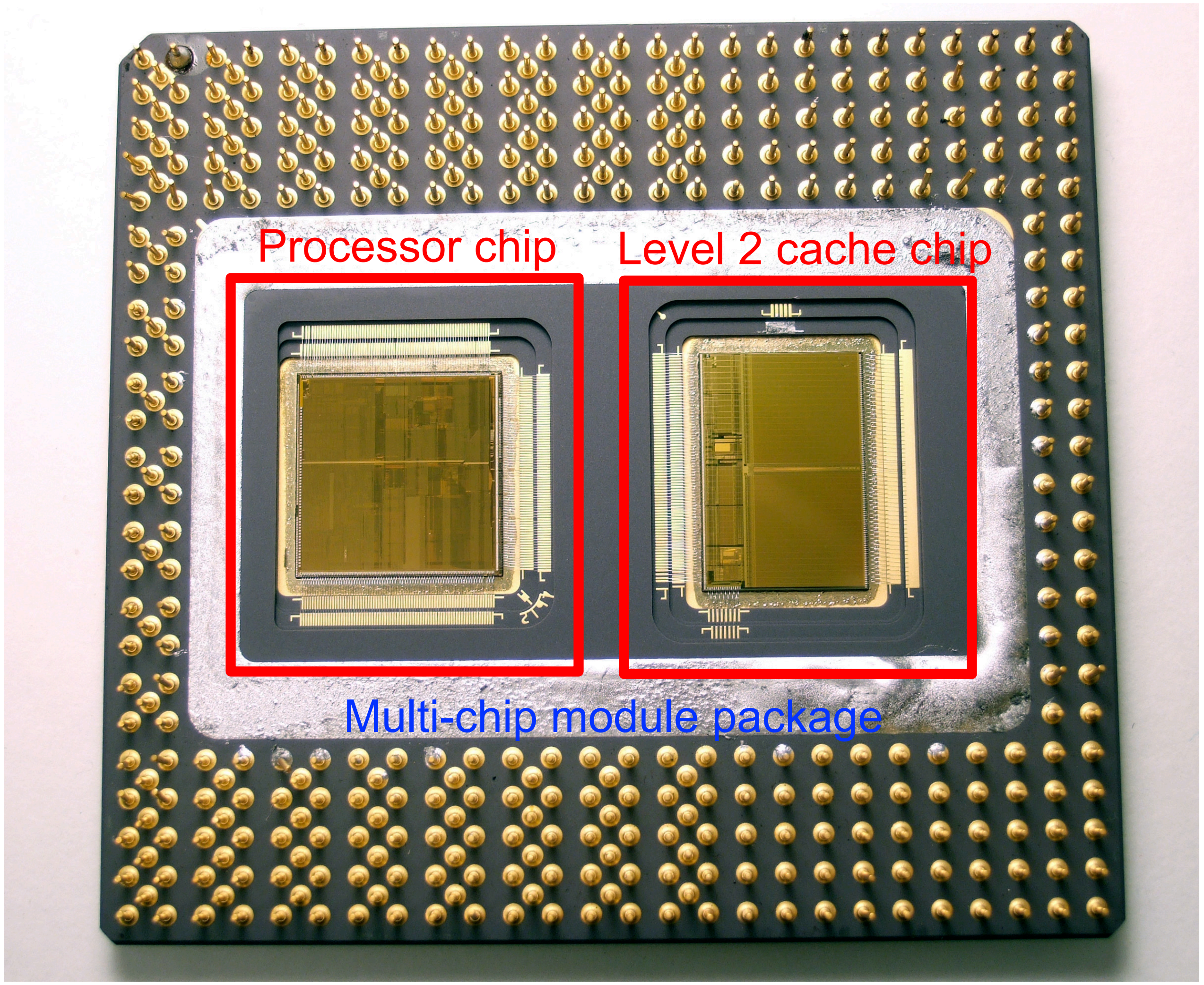
General Organization of an OOO Processor



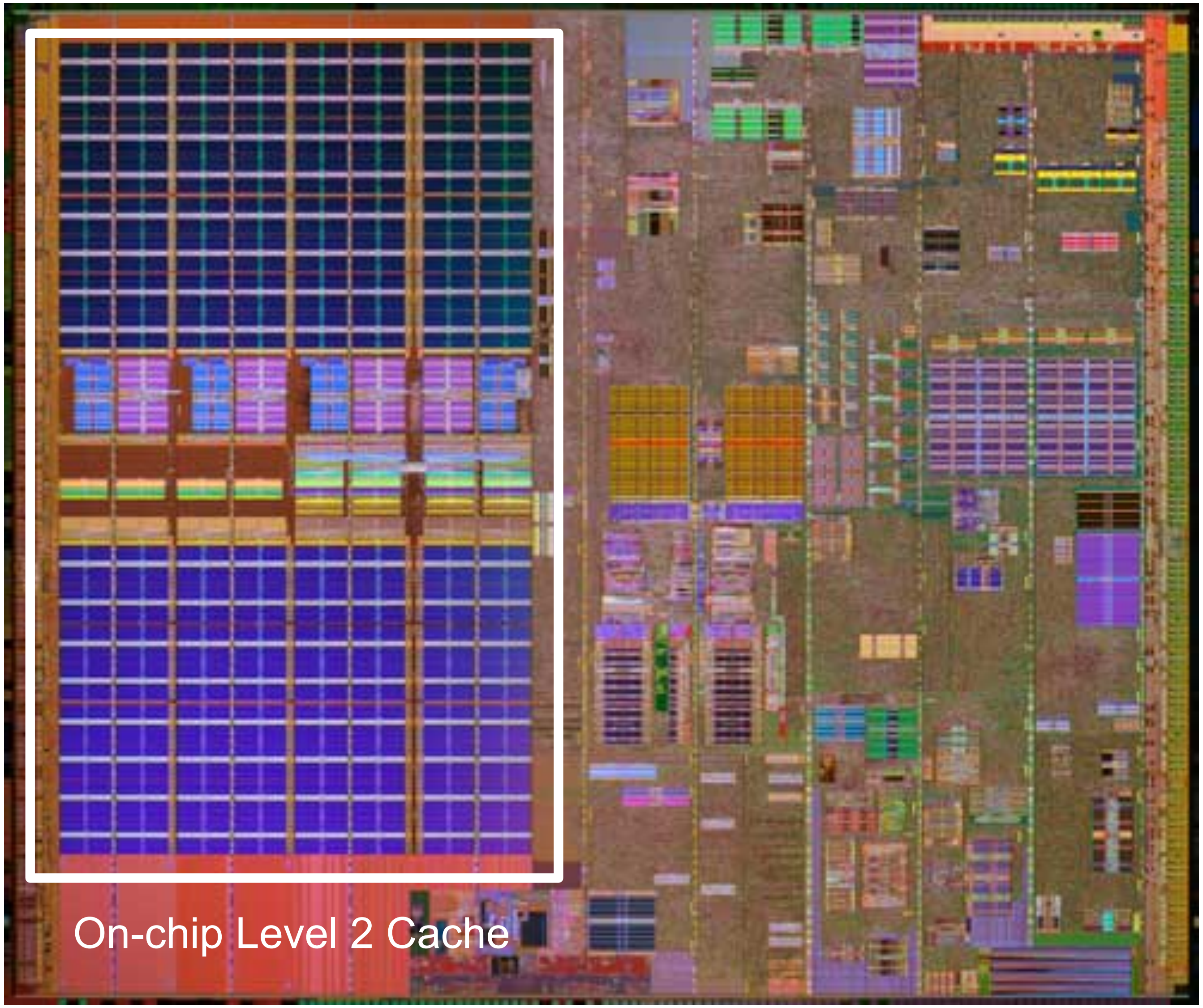
floating pt. register file
floating pt. instruction buffers
functional units
memory interface
pre-decode
instr. cache
instr. buffer
decode, rename, &dispatch
integer/address instruction buffers
functional units and data cache
integer register file
re-order and commit

- Smith and Sohi, "The Microarchitecture of Superscalar Processors," Proc. IEEE, Dec. 1995.

Intel Pentium Pro (1995)



Intel Pentium 4 (2000)



https://www.anandtech.com/show/1621/3

A Modern OoO Design: Intel Pentium 4

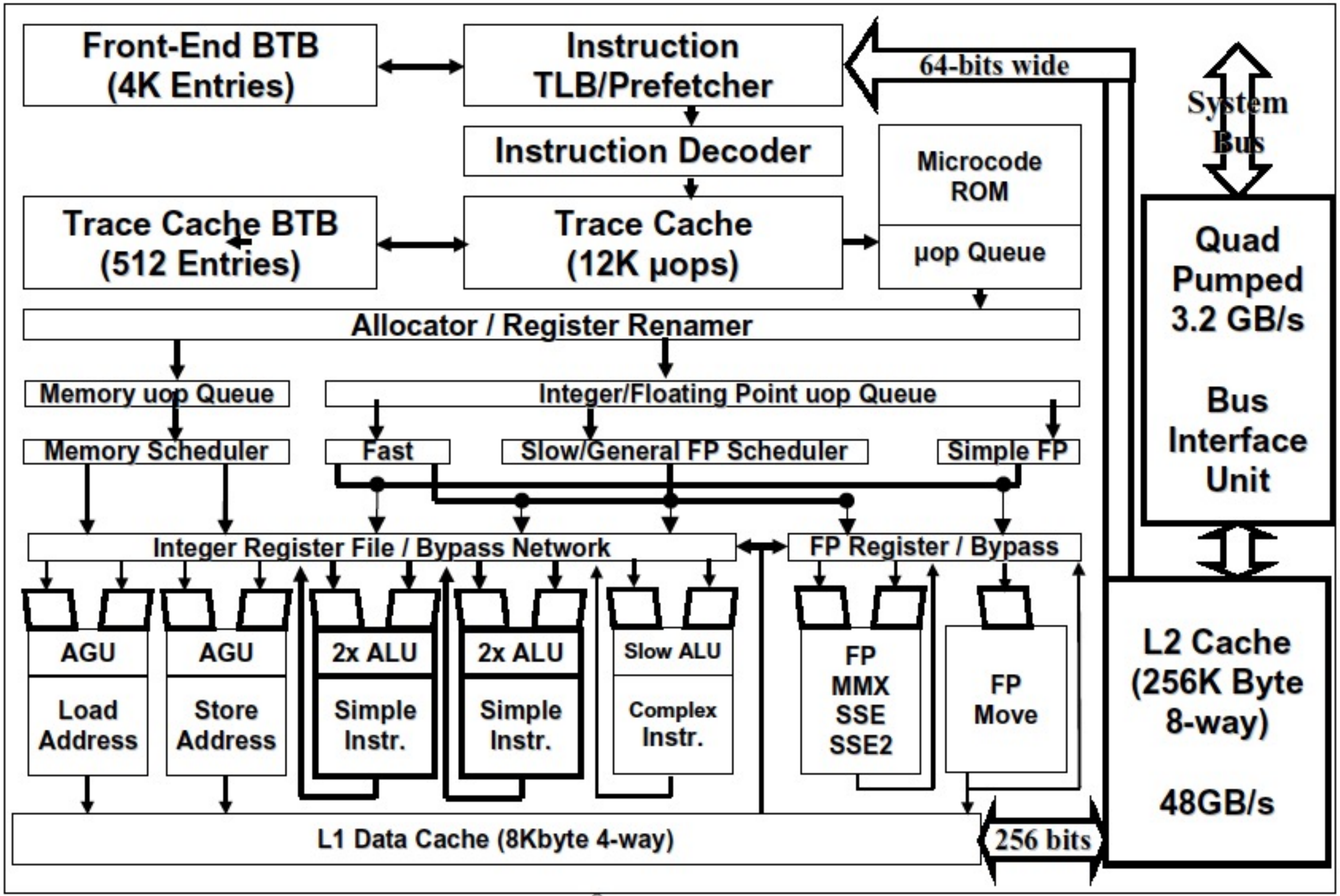
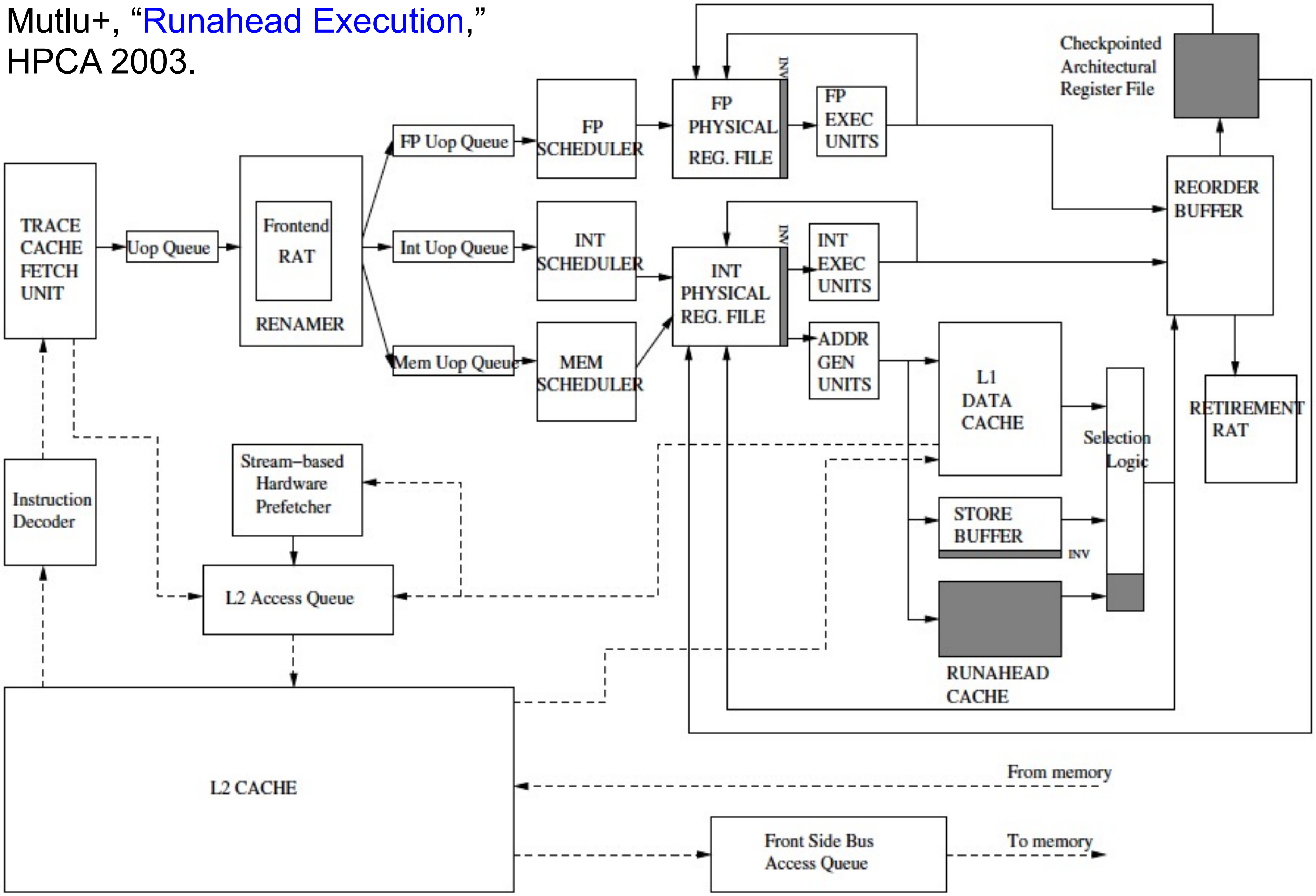


Figure 4: Pentium® 4 processor microarchitecture

Boggs et al., "The Microarchitecture of the Pentium 4 Processor," Intel Technology Journal, 2001.

Intel Pentium 4 Simplified

Mutlu+, “Runahead Execution,” HPCA 2003.



Alpha 21264

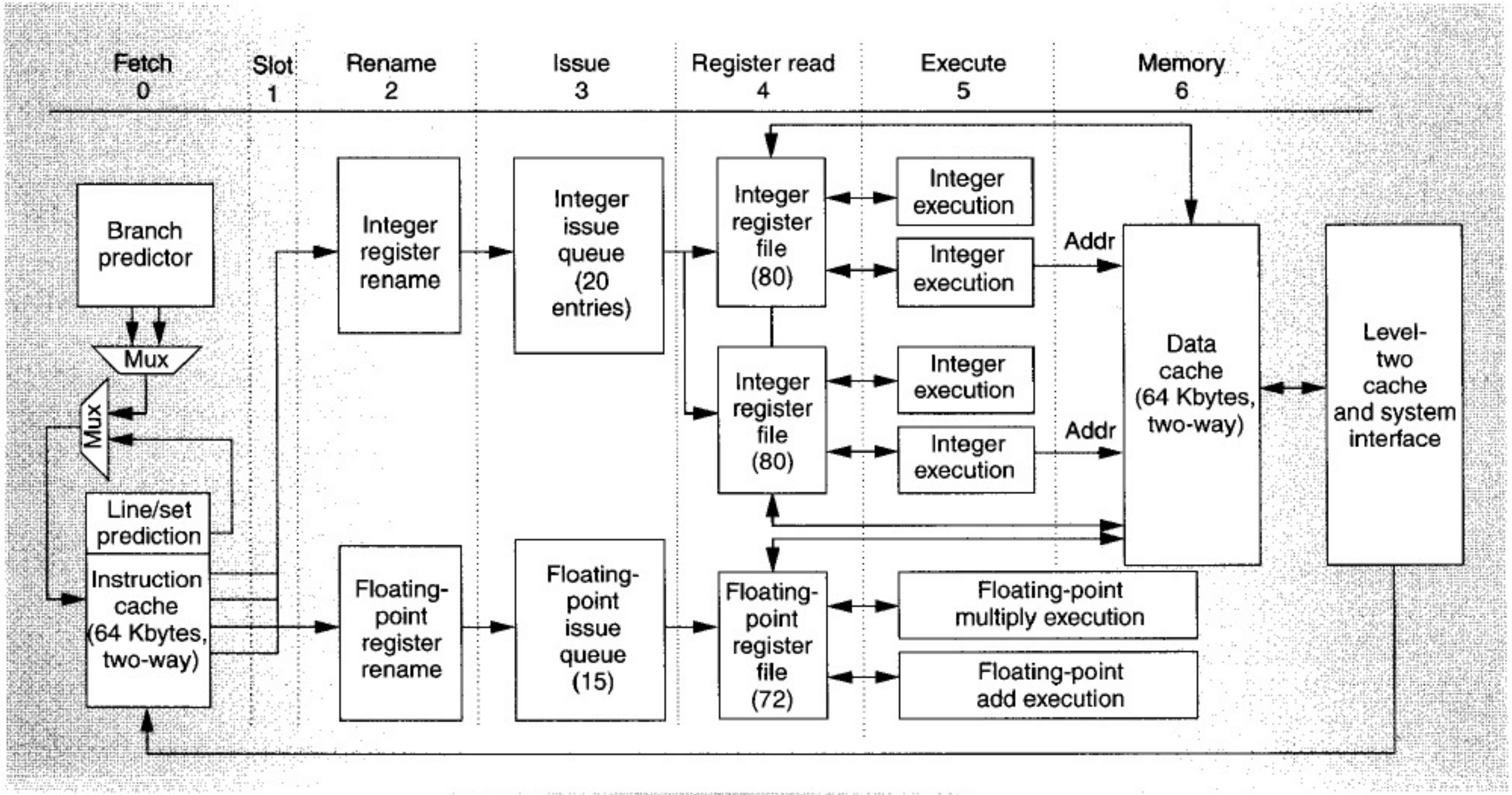
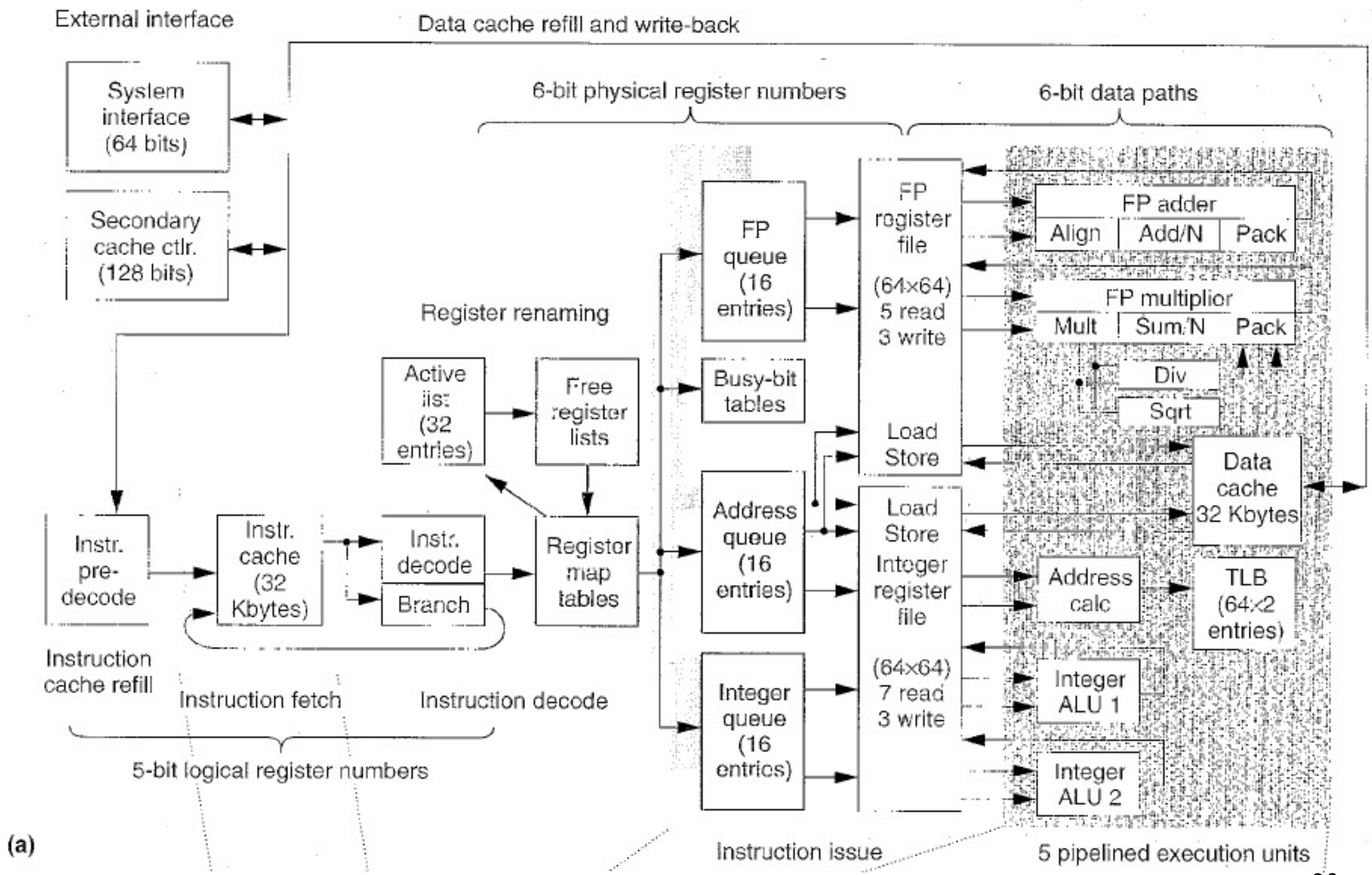

Fetch 0
Slot 1
Rename 2
Issue 3
Register read 4
Execute 5
Memory 6
Branch predictor
Mux
Mux
Line/set prediction
Instruction cache (64 Kbytes, two-way)
Integer register rename
Integer issue queue (20 entries)
Integer register file (80)
Integer register file (80)
Integer execution
Integer execution
Integer execution
Integer execution
Addr
Addr
Data cache (64 Kbytes, two-way)
Level-two cache and system interface
Floating-point register rename
Floating-point issue queue (15)
Floating-point register file (72)
Floating-point multiply execution
Floating-point add execution

Figure 2. Stages of the Alpha 21264 instruction pipeline.

Kessler, "The Alpha 21264 Microprocessor," IEEE Micro, March-April 1999.

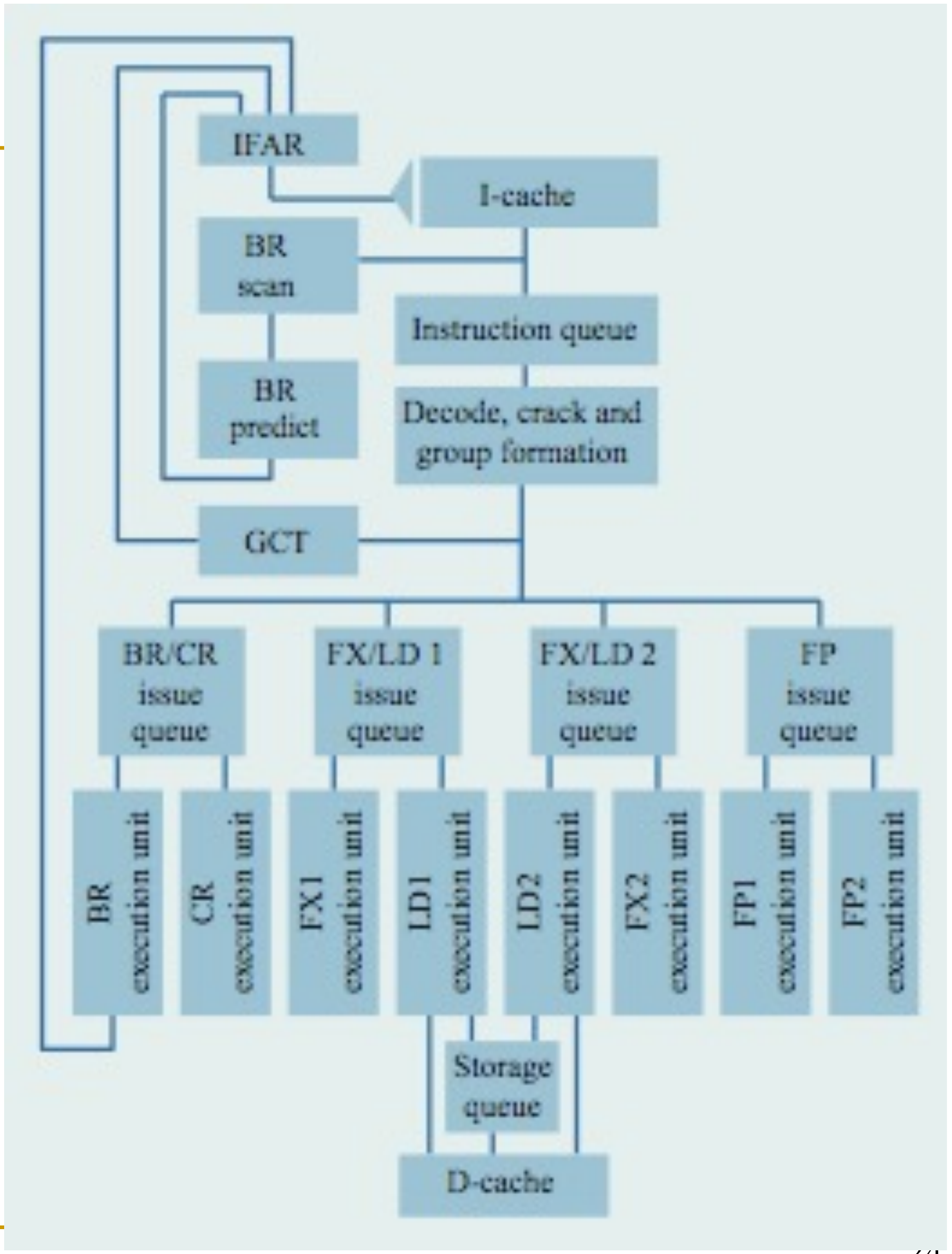
MIPS R10000



Yeager, "The MIPS R10000 Superscalar Microprocessor," IEEE Micro, April 1996

IBM POWER4

- Tendler et al., "POWER4 system microarchitecture," IBM J R&D, 2002.



IBM POWER4

- 2 cores, out-of-order execution
- 100-entry instruction window in each core
- 8-wide instruction fetch, issue, execute
- Large, local+global hybrid branch predictor
- 1.5MB, 8-way L2 cache
- Aggressive stream based prefetching

IBM POWER5

- Kalla et al., "IBM Power5 Chip: A Dual-Core Multithreaded Processor," IEEE Micro 2004.

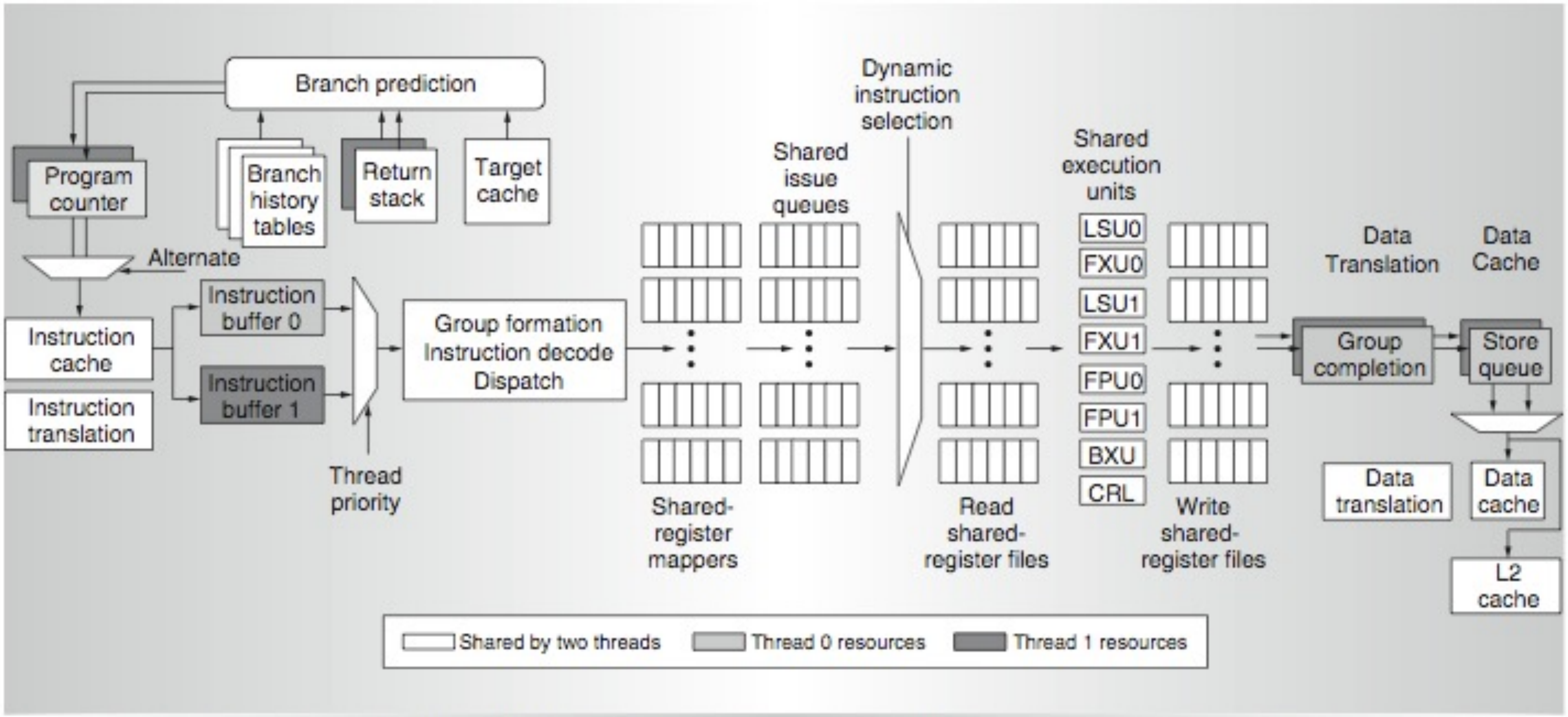
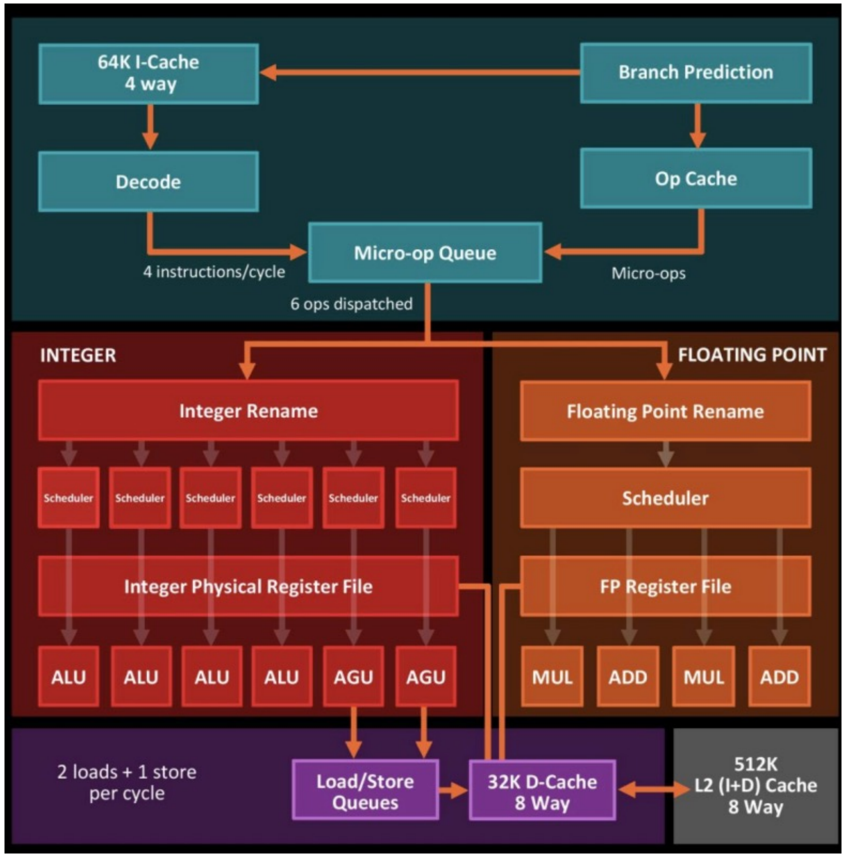
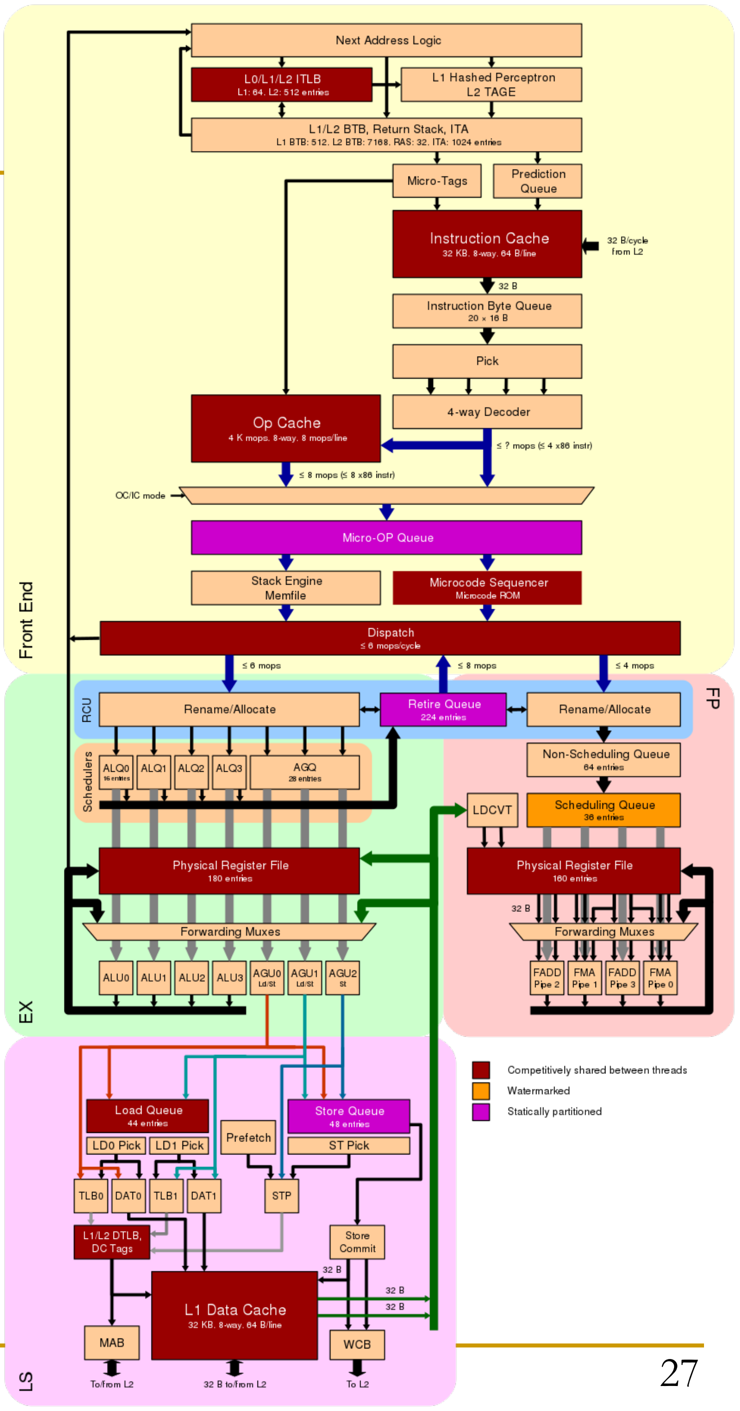


Figure 4. Power5 instruction data flow (BXU = branch execution unit and CRL = condition register logical execution unit).

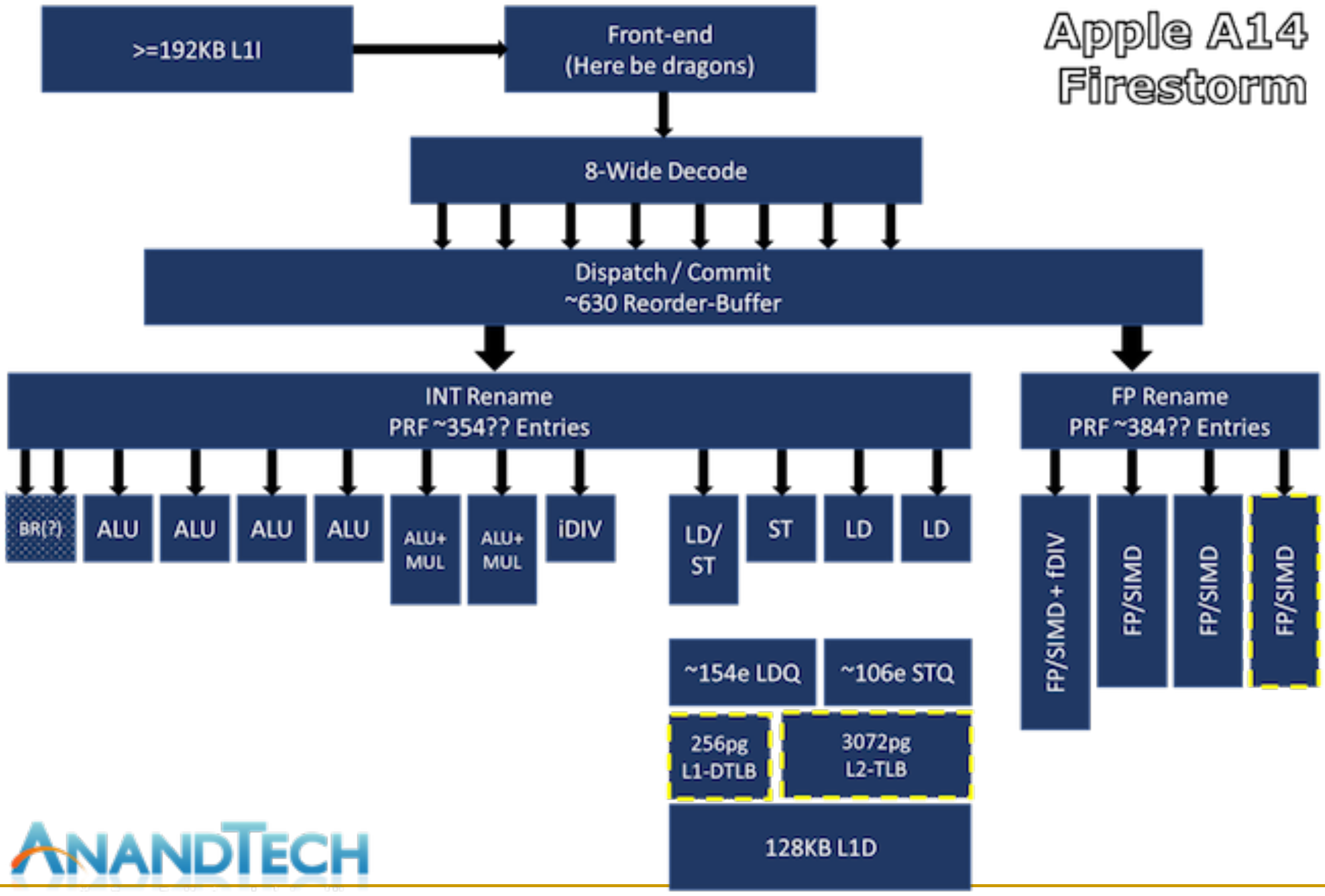
AMD Zen/Zen2? (2019)





https://en.wikichip.org/wiki/amd/microarchitectures/zen_2
https://courses.engr.illinois.edu/cs433/fa2020/slides/mini-project-amd-zen.pdf

Apple M1 Firestorm? (2020)



https://www.anandtech.com/show/16226/apple-silicon-m1-a14-deep-dive/2

Out-of-Order Execution Tradeoffs

- Advantages
  - Latency tolerance: Allows independent instructions to execute and complete in the presence of long-latency operations
    → Higher performance than in-order execution
  - Irregular parallelism: Dynamically finds and exploits parallel operations in a program
    → Difficult to find/exploit such parallelism statically
- Disadvantages
  - Higher complexity
    - Potentially lengthens critical path delay → clock cycle time
  - More hardware resources needed
- Recall: Execution time of an entire program
  - **{# of instructions} x {Average CPI} x {clock cycle time}**

Other Approaches to Concurrency (or Instruction Level Parallelism)

Approaches to (Instruction-Level) Concurrency

- Pipelining
- Fine-Grained Multithreading
- Out-of-order Execution
- Dataflow (at the ISA level)
- Superscalar Execution
- VLIW
- SIMD Processing (Vector and array processors, GPUs)
- Decoupled Access Execute
- Systolic Arrays

Review: Data Flow:
Exploiting Irregular Parallelism

Recall: OOO Execution: **Restricted Dataflow**

- An out-of-order engine dynamically builds the dataflow graph of a piece of the program
- The dataflow graph is limited to the **instruction window**
  - Instruction window: all decoded but not yet retired instructions
- Can we do it for the whole program?
  - In other words, how can we have a large instruction window?
- Can we do it efficiently with Tomasulo’s algorithm?

Recall: State of RAT and RS in Cycle 7

Slightly harder tasks for you:

1. **Draw the dataflow graph for the executing code**
2. **Provide the executing code in sequential order**

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	0	x	
R4	1		4
R5	0	d	
R6	1		6
R7	0	b	
R8	1		8
R9	1		9
R10	0	c	
R11	0	y	

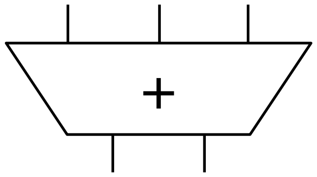
Register Alias Table

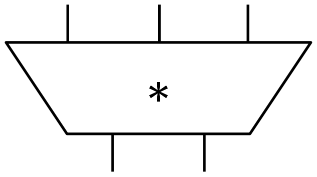
RS for ADD Unit

	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
a	0	x		1	~	4
b	1	~	2	1	~	6
c	1	~	8	1	~	9
d	0	a		0	y	

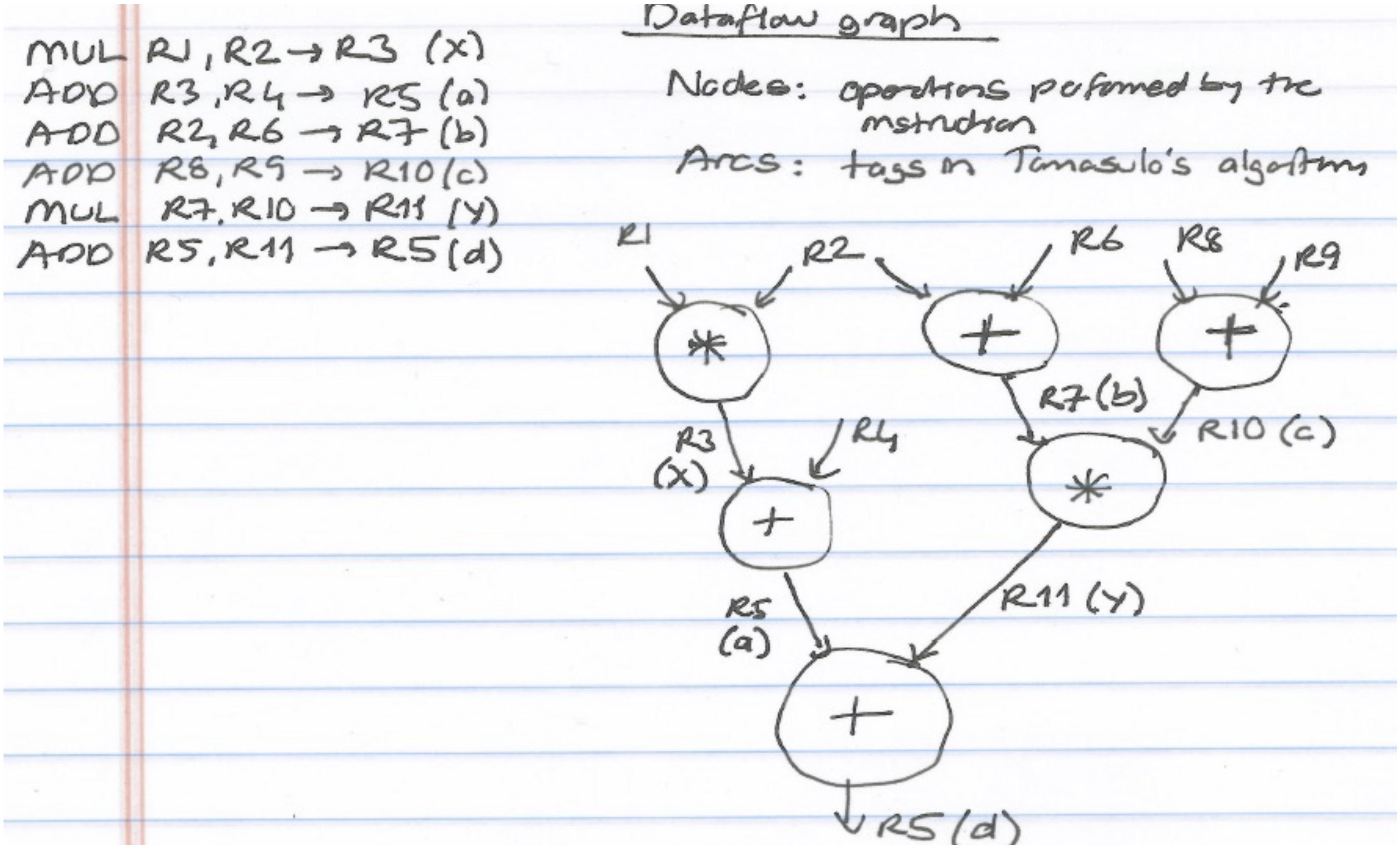
RS for MUL Unit

	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
x	1	~	1	1	~	2
y	0	b		0	c	
z						
t						





Recall: Reverse Engineered Dataflow Graph



```
MUL R1, R2 → R3 (x)
ADD R3, R4 → R5 (a)
ADD R2, R6 → R7 (b)
ADD R8, R9 → R10 (c)
MUL R7, R10 → R11 (y)
ADD R5, R11 → R5 (d)
```

Dataflow graph

Nodes: operations performed by the instruction

Arcs: tags in Tomasulo's algorithm

We can "easily" reverse-engineer the dataflow graph of the executing code!

Data Flow Summary

- Availability of data determines order of execution
- A data flow node fires when its sources are ready
- Programs represented as data flow graphs (of nodes)

- Data Flow at the ISA level has **not** been (as) successful

- **Data Flow** implementations at the **microarchitecture** level (while preserving von Neumann model semantics) have been **very successful**
  - Out of order execution is the prime example

- Data Flow mapping of programs to reconfigurable hardware substrates (FPGAs) has also been successful

Recall: ISA-level Tradeoff: Program Counter

- Do we want a Program Counter (PC or IP) in the ISA?
  - Yes: Control-driven, sequential execution
    - An instruction is executed when the PC points to it
    - PC automatically changes sequentially (except for control flow instructions) → sequential
  - No: Data-driven, parallel execution
    - An instruction is executed when all its operand values are available → dataflow

- Tradeoffs: MANY high-level ones
  - Ease of programming (for average programmers)?
  - Ease of compilation?
  - Performance: Extraction of parallelism?
  - Hardware complexity?

Pure Data Flow Advantages/Disadvantages

- Advantages
  - Very good at exploiting irregular parallelism
    - Only real dependences constrain processing
    - More parallelism can be exposed than Von Neumann model
- Disadvantages
  - No precise state semantics
    - Debugging very difficult
    - Interrupt/exception handling is difficult (what is precise state semantics?)
  - Too much parallelism? (Parallelism control needed)
  - High bookkeeping overhead (tag matching, data storage)
  - How to enable mutable data structures
  - …

Recall: ISA vs. Microarchitecture Level Tradeoff

- A similar tradeoff (control vs. data-driven execution) can be made at the microarchitecture level

- ISA: Specifies how the **programmer sees** the instructions to be executed
  - Programmer sees a sequential, control-flow execution order vs.
  - Programmer sees a dataflow execution order

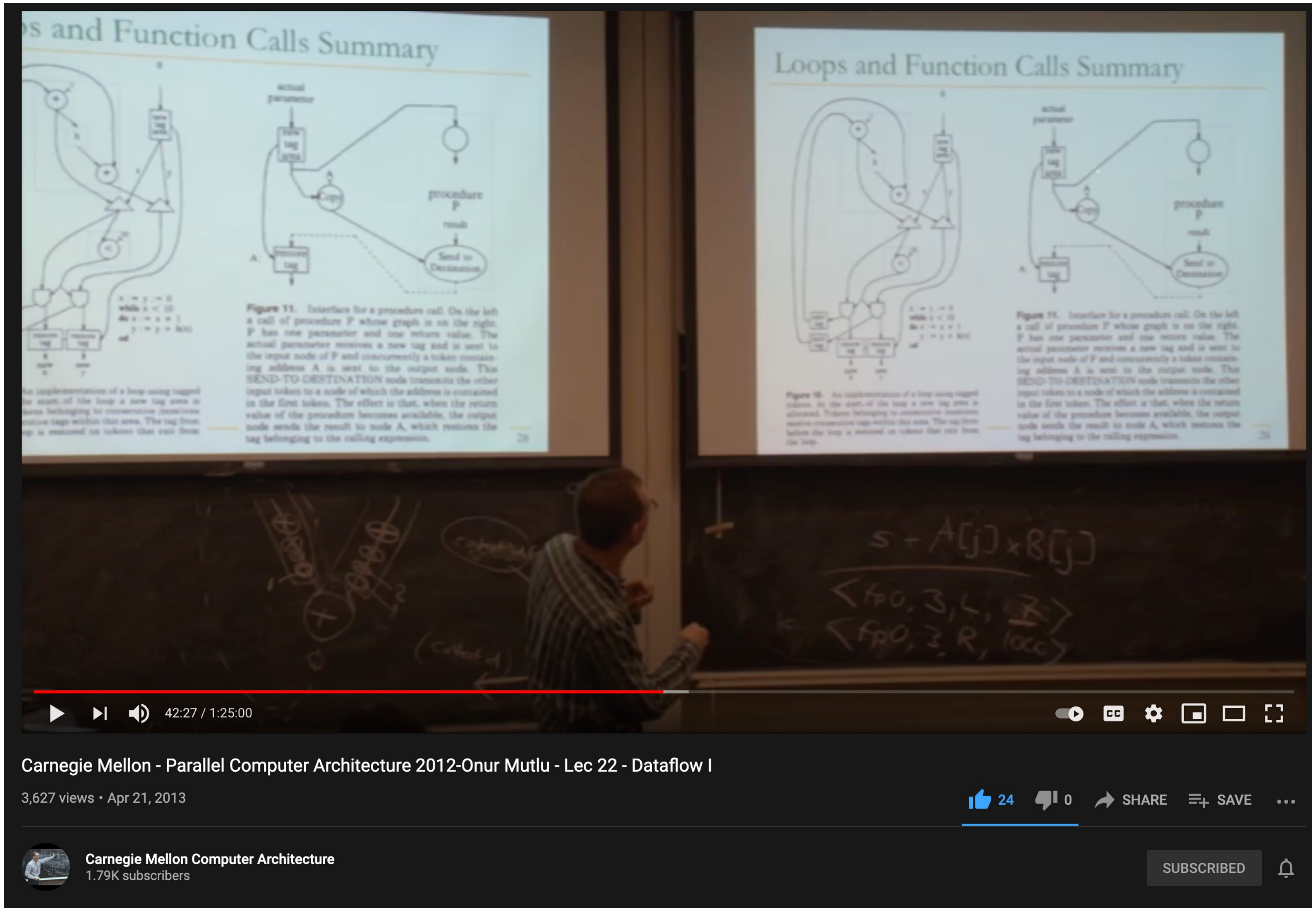
- Microarchitecture: How the **underlying implementation actually executes** instructions
  - Microarchitecture can execute instructions in any order as long as it obeys the semantics specified by the ISA when making the instruction results visible to software
    - Programmer should see the order specified by the ISA

Readings & Lectures on Data Flow Model

- Dennis and Misunas, "A preliminary architecture for a basic data-flow processor," ISCA 1974.
- Gurd et al., "The Manchester prototype dataflow computer," CACM 1985.

- More detailed Lecture Video & Slides on DataFlow:
  - http://www.youtube.com/watch?v=D2uue7izU2c
  - http://www.ece.cmu.edu/~ece740/f13/lib/exe/fetch.php?media=onur-740-fall13-module5.2.1-dataflow-part1.ppt

Lecture Video on Dataflow



42:27 / 1:25:00
Carnegie Mellon - Parallel Computer Architecture 2012-Onur Mutlu - Lec 22 - Dataflow I
3,627 views • Apr 21, 2013
24
0
SHARE
SAVE
Carnegie Mellon Computer Architecture
1.79K subscribers
SUBSCRIBED

http://www.youtube.com/watch?v=D2uue7izU2c

Approaches to (Instruction-Level) Concurrency

- Pipelining
- Fine-Grained Multithreading
- Out-of-order Execution
- Dataflow (at the ISA level)
- Superscalar Execution
- VLIW
- SIMD Processing (Vector and array processors, GPUs)
- Decoupled Access Execute
- Systolic Arrays

Superscalar Execution

Superscalar Execution

- Idea: Fetch, decode, execute, retire multiple instructions per cycle
  - N-wide superscalar → N instructions per cycle

- Need to add the hardware resources for doing so

- Hardware performs the dependence checking between concurrently-fetched instructions

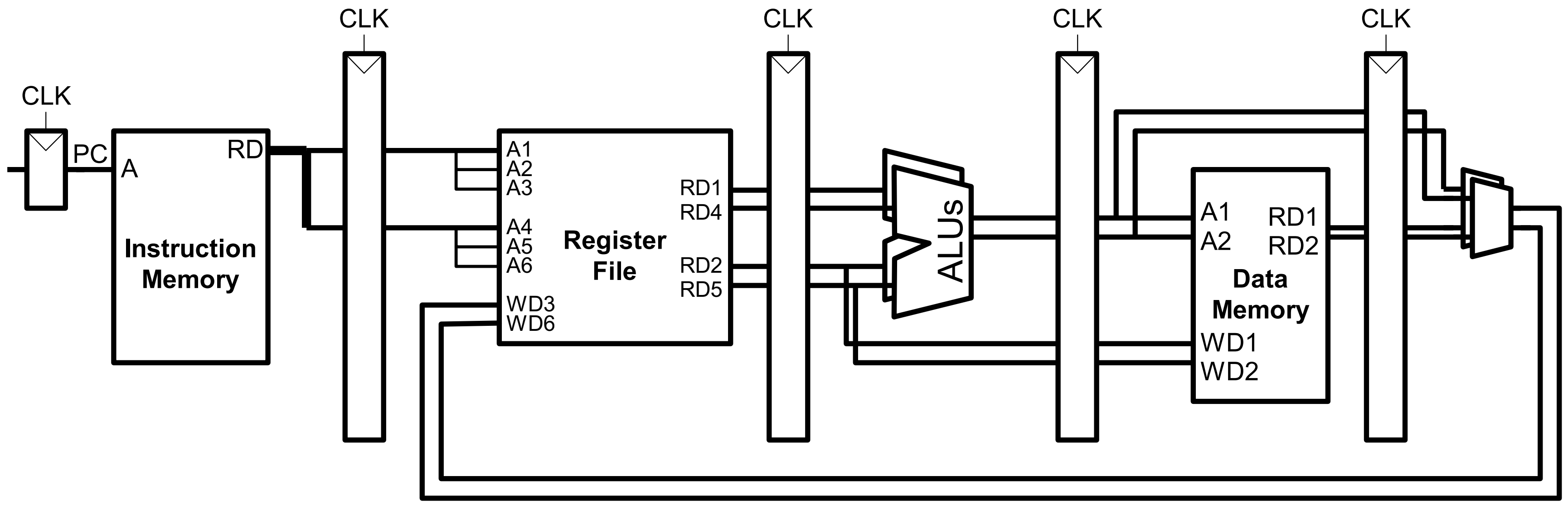
- **Superscalar** execution and **out-of-order** execution are **orthogonal concepts**
  - Can have all four combinations of processors:

    [in-order, out-of-order] x [scalar, superscalar]

In-Order Superscalar Processor Example

- Multiple copies of datapath: Can fetch/decode/execute multiple instructions per cycle

- Dependences make it tricky to dispatch multiple instructions in the same cycle
  - Need dependence detection between concurrently-fetched instructions

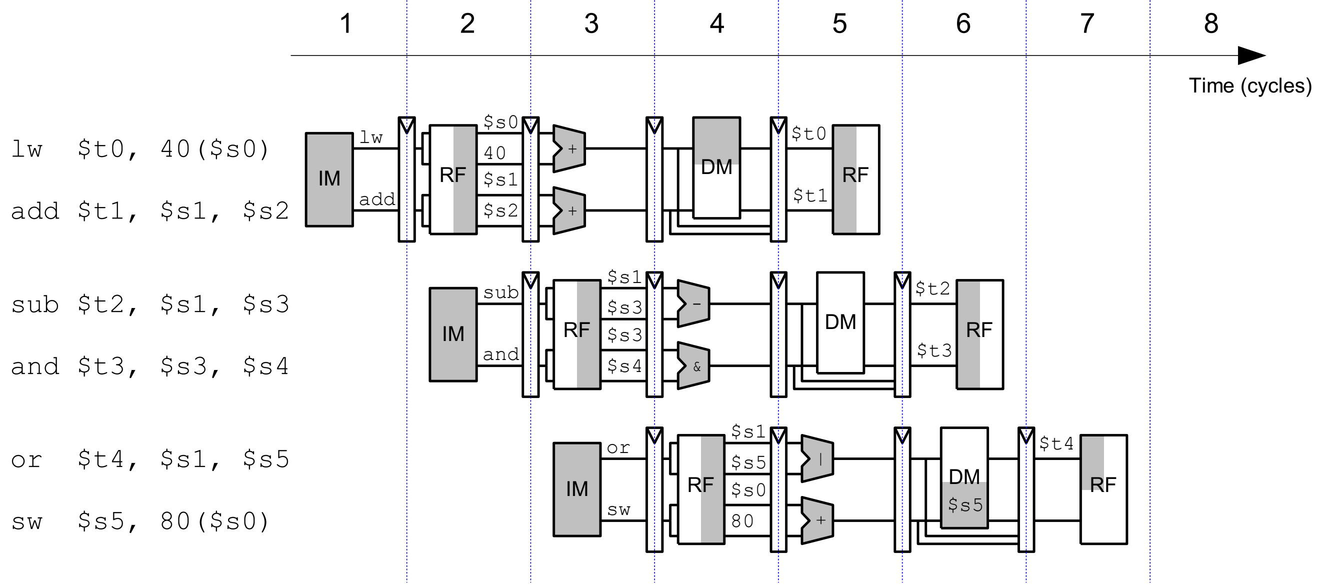


Here: Ideal IPC = 2

In-Order Superscalar Performance Example

```
lw  $t0, 40($s0)
add $t1, $s1, $s2
sub $t2, $s1, $s3
and $t3, $s3, $s4
or  $t4, $s1, $s5
sw  $s5, 80($s0)
```

Ideal IPC = 2



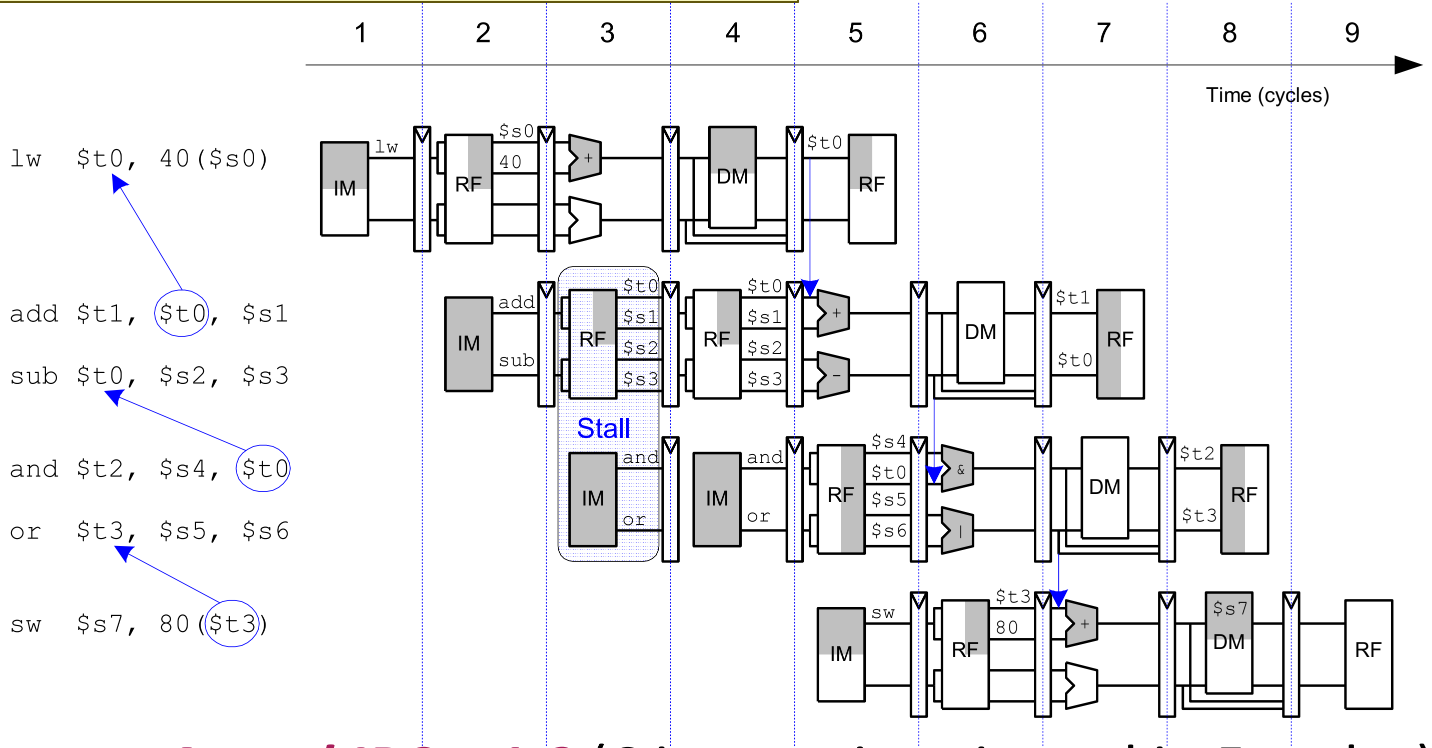
Actual IPC = 2 (6 instructions issued in 3 cycles)

Superscalar Performance with Dependences

```
lw  $t0, 40($s0)
add $t1, $t0, $s1
sub $t0, $s2, $s3
and $t2, $s4, $t0
or  $t3, $s5, $s6
sw  $s7, 80($t3)
```

Ideal IPC = 2

Can you reorder the instructions to get IPC = 2?



Actual IPC = 1.2 (6 instructions issued in 5 cycles)

Review: How to Handle Data Dependences

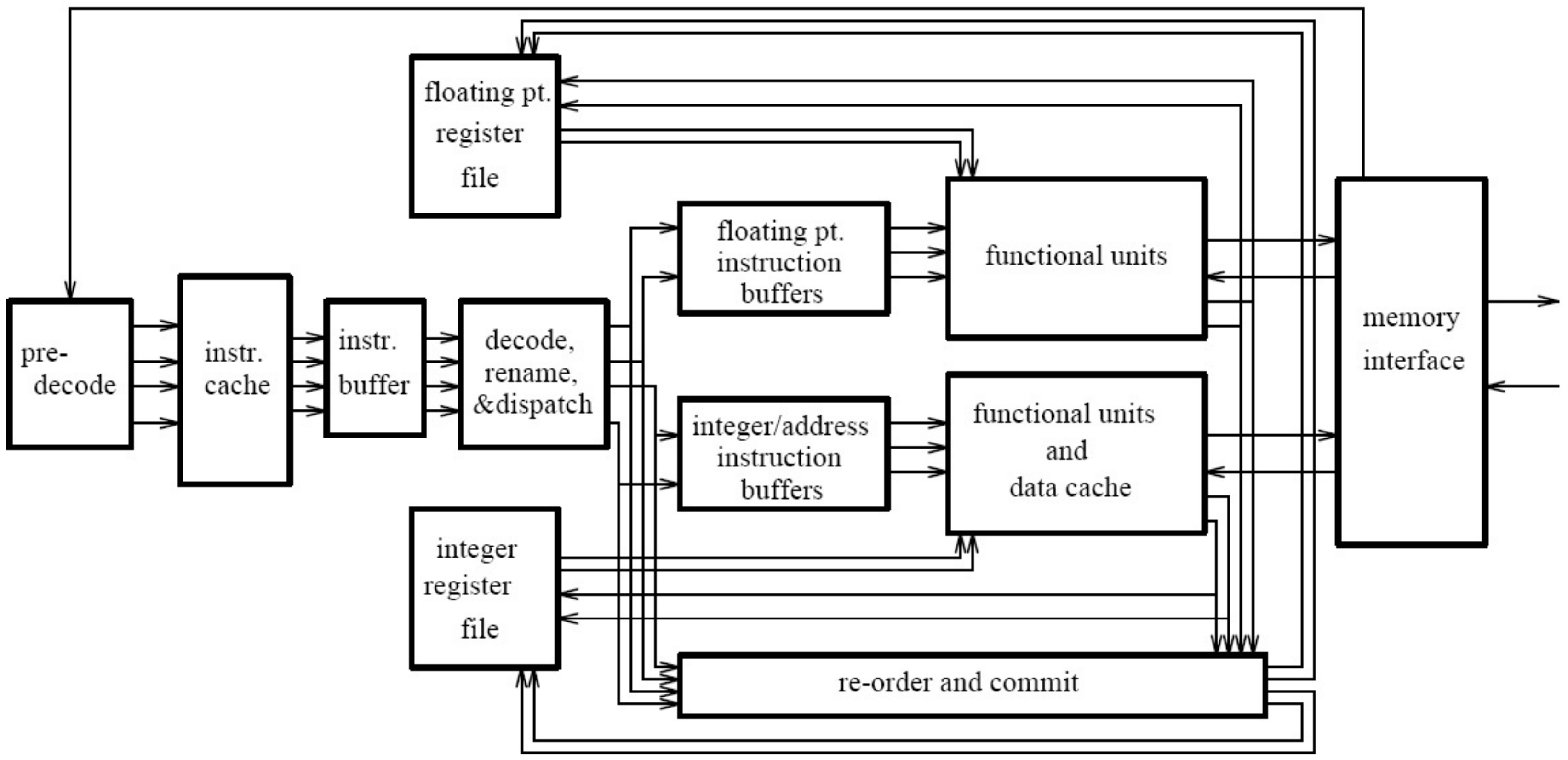
- Six fundamental ways of handling flow dependences
  - Detect and wait until value is available in register file
  - Detect and forward/bypass data to dependent instruction
  - Detect and eliminate the dependence at the software level
    - No need for the hardware to detect dependence
  - Detect and move it out of the way for independent instructions
  - Predict the needed value(s), execute "speculatively", and verify
  - Do something else (fine-grained multithreading)
    - No need to detect

- Can employ all these in superscalar processors

Superscalar Execution Tradeoffs

- Advantages
  - Higher instruction throughput
    - Higher IPC: instructions per cycle (i.e., lower CPI)
- Disadvantages
  - Higher complexity for dependence checking
    - Requires dependence checking between concurrent instructions
    - Register renaming becomes more complex in an OoO processor
    - Potentially lengthens critical path delay → clock cycle time
  - More hardware resources needed
- Recall: Execution time of an entire program
  - **{# of instructions} x {Average CPI} x {clock cycle time}**

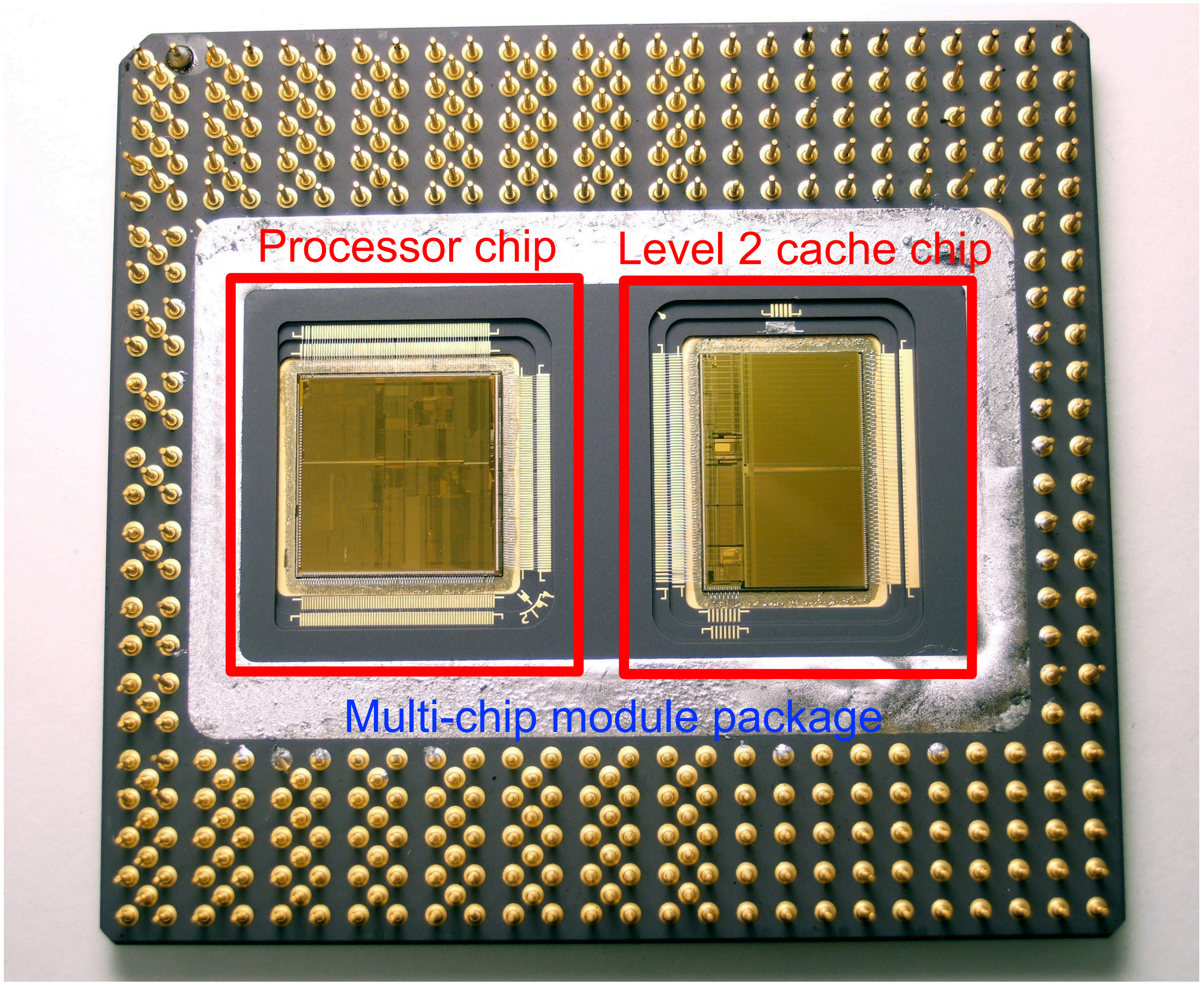
General Organization of a Superscalar+OoO Processor



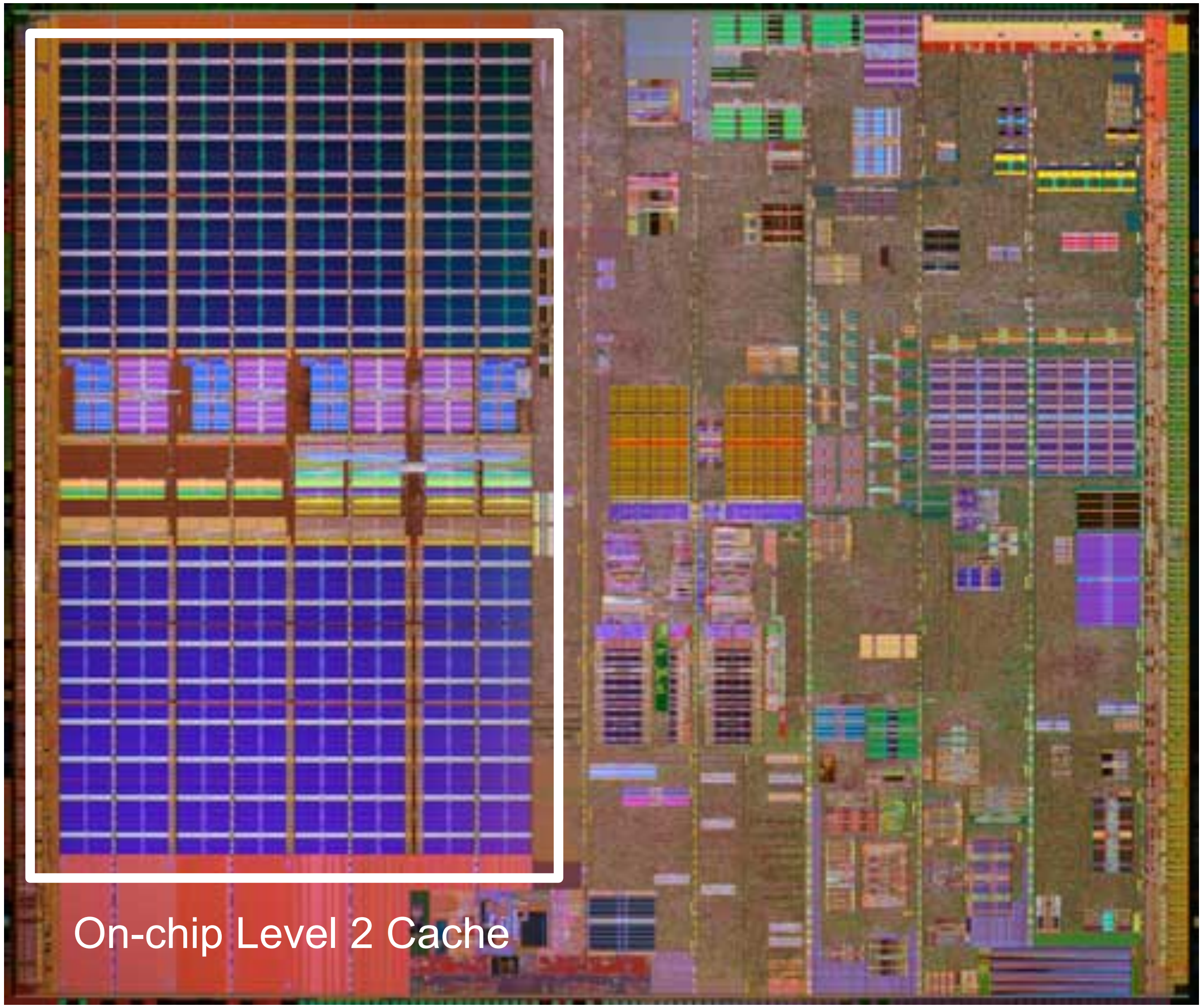
floating pt. register file
pre-decode
instr. cache
instr. buffer
decode, rename, &dispatch
floating pt. instruction buffers
functional units
integer/address instruction buffers
functional units and data cache
memory interface
integer register file
re-order and commit

- Smith and Sohi, "The Microarchitecture of Superscalar Processors," Proc. IEEE, Dec. 1995.

Intel Pentium Pro (1995)

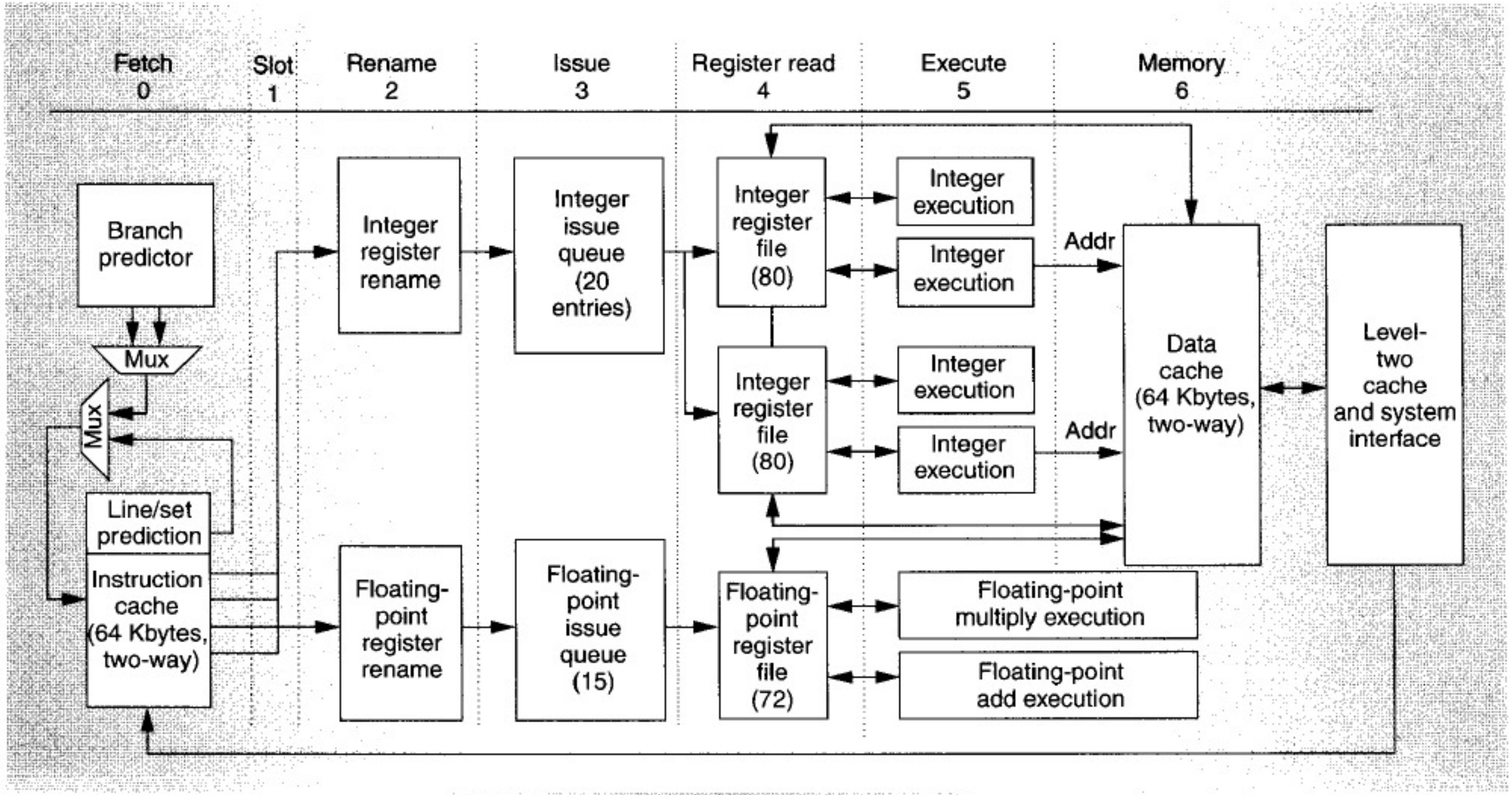


Intel Pentium 4 (2000)



https://www.anandtech.com/show/1621/3

Alpha 21264

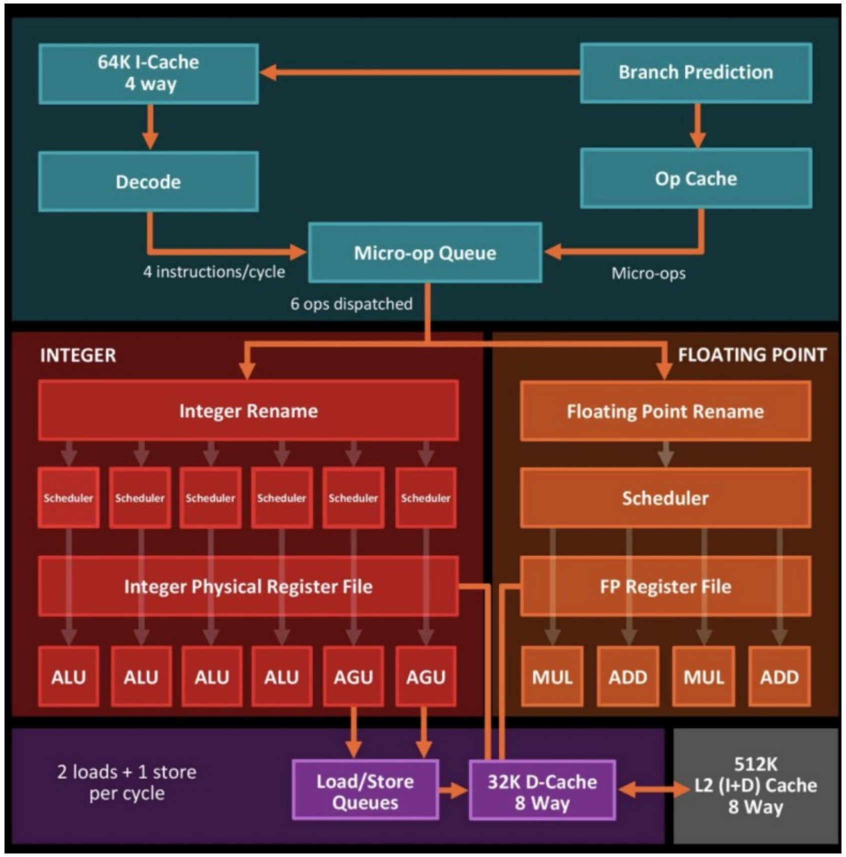


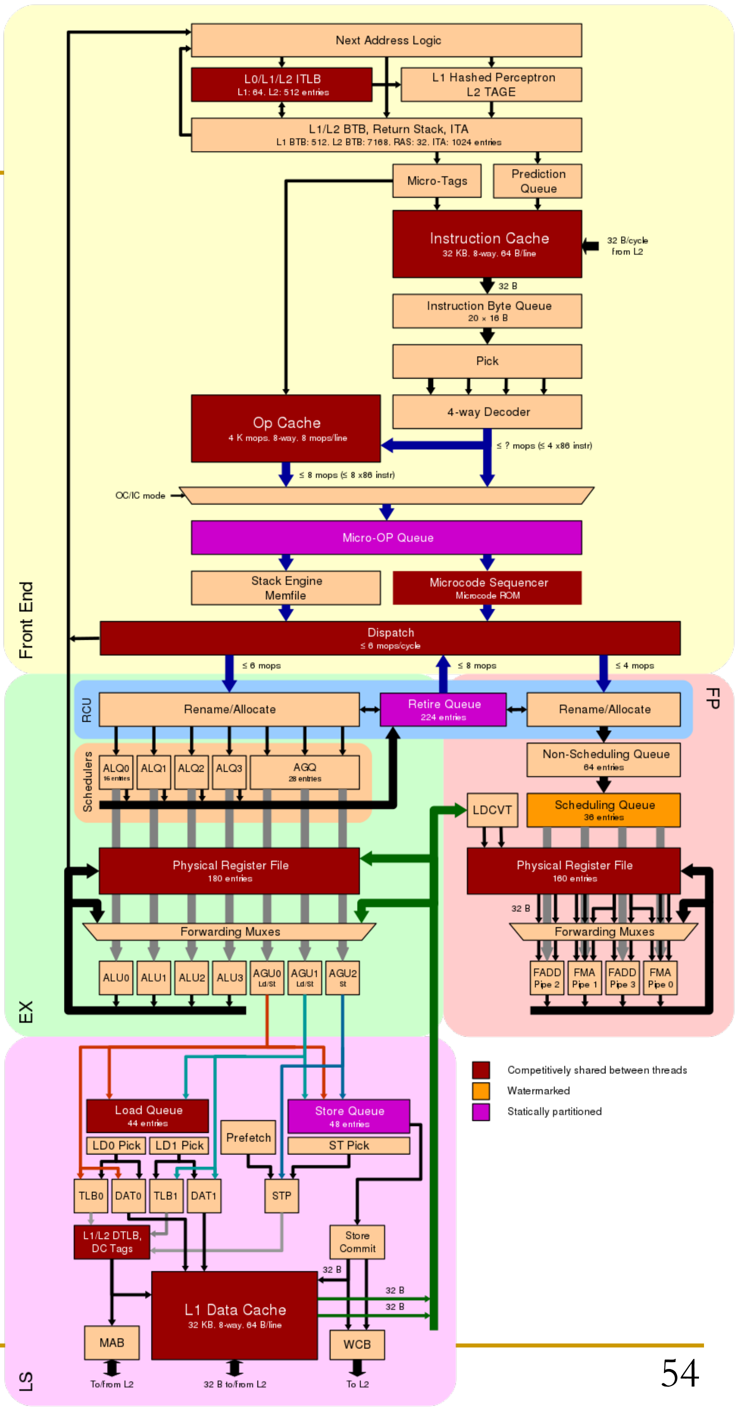
Fetch 0
Slot 1
Rename 2
Issue 3
Register read 4
Execute 5
Memory 6
Branch predictor
Mux
Mux
Line/set prediction
Instruction cache (64 Kbytes, two-way)
Integer register rename
Integer issue queue (20 entries)
Integer register file (80)
Integer register file (80)
Integer execution
Integer execution
Integer execution
Integer execution
Addr
Addr
Data cache (64 Kbytes, two-way)
Level-two cache and system interface
Floating-point register rename
Floating-point issue queue (15)
Floating-point register file (72)
Floating-point multiply execution
Floating-point add execution

Figure 2. Stages of the Alpha 21264 instruction pipeline.

Kessler, "The Alpha 21264 Microprocessor," IEEE Micro, March-April 1999.

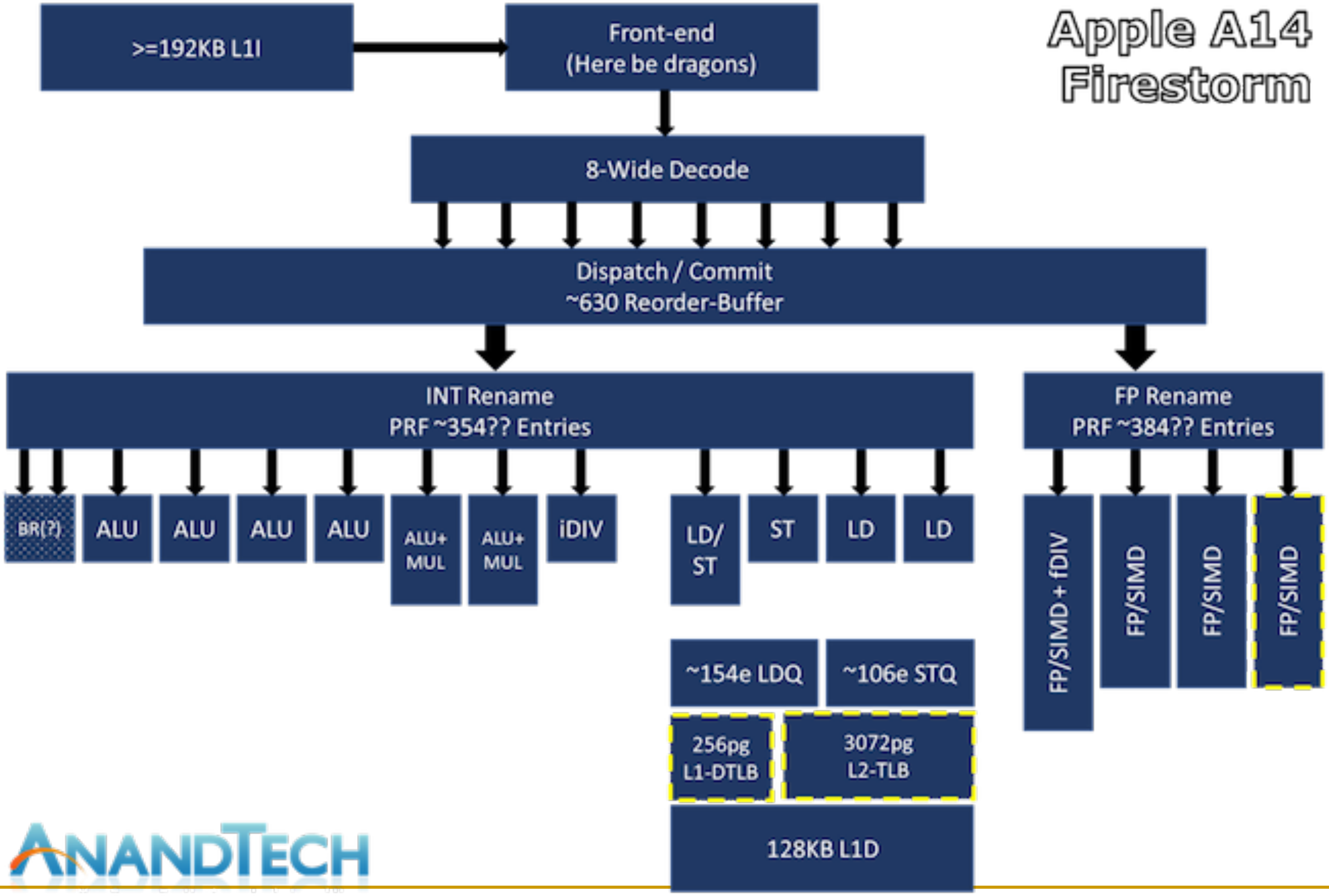
AMD Zen/Zen2? (2019)





https://en.wikichip.org/wiki/amd/microarchitectures/zen_2
https://courses.engr.illinois.edu/cs433/fa2020/slides/mini-project-amd-zen.pdf

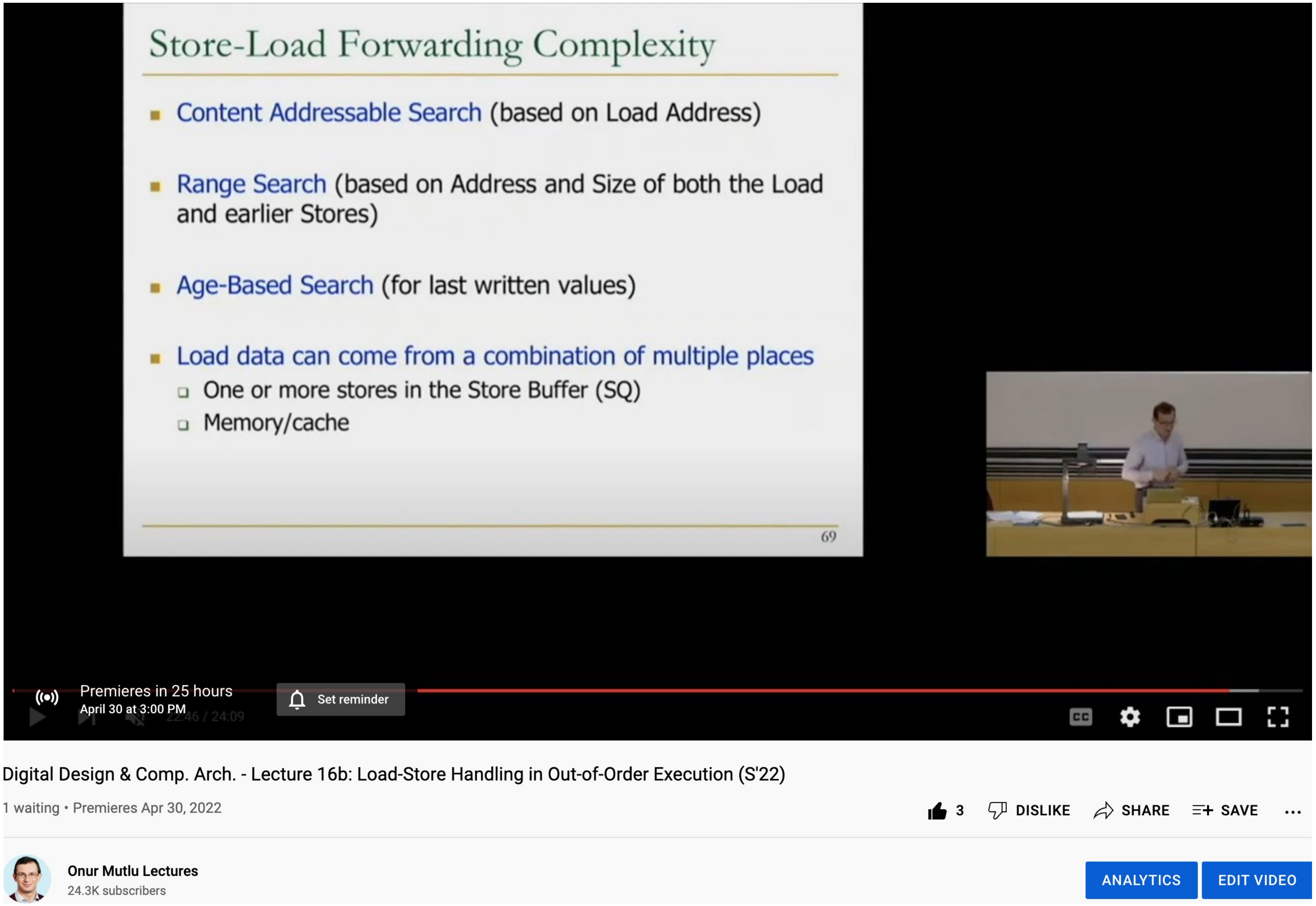
Apple M1 Firestorm? (2020)



https://www.anandtech.com/show/16226/apple-silicon-m1-a14-deep-dive/2

Backup Slides & Optional Video for: Handling Out-of-Order Execution of Loads and Stores

Lecture on Load-Store Handling in OoO



Store-Load Forwarding Complexity

- Content Addressable Search (based on Load Address)
- Range Search (based on Address and Size of both the Load and earlier Stores)
- Age-Based Search (for last written values)
- Load data can come from a combination of multiple places
  - One or more stores in the Store Buffer (SQ)
  - Memory/cache

69

Digital Design & Comp. Arch. - Lecture 16b: Load-Store Handling in Out-of-Order Execution (S'22)

1 waiting • Premieres Apr 30, 2022

https://www.youtube.com/watch?v=vulgJ3N1rlg&list=PL5Q2soXY2Zi97Ya5DEUpMpO2bbAoaG7c6&index=18

Digital Design & Computer Arch.

Lecture 17a: Dataflow & Superscalar Execution

Prof. Onur Mutlu

ETH Zürich

Spring 2022

29 April 2022

Handling Out-of-Order Execution of Loads and Stores

Registers versus Memory

- So far, we considered mainly registers as part of state
- What about memory?
- What are the fundamental differences between registers and memory?
  - Register dependences known statically – memory dependences determined dynamically
  - Register state is small – memory state is large
  - Register state is not visible to other threads/processors – memory state is shared between threads/processors (in a shared memory multiprocessor)

Memory Dependence Handling (I)

- Need to obey memory dependences in an out-of-order machine
  - and need to do so while providing high performance

- Observation and Problem: Memory address is not known until a load/store executes

- Corollary 1: Renaming memory addresses is difficult
- Corollary 2: Determining dependence or independence of loads/stores has to be handled *after* their (partial) execution
- Corollary 3: When a load/store has its address ready, there may be older/younger stores/loads with unknown addresses in the machine

Memory Dependence Handling (II)

- When do you schedule a load instruction in an OOO engine?
  - Problem: A younger load can have its address ready before an older store's address is known
  - Known as the memory disambiguation problem or the unknown address problem

- Approaches
  - Conservative: Stall the load until all previous stores have computed their addresses (or even retired from the machine)
  - Aggressive: Assume load is independent of unknown-address stores and schedule the load right away
  - Intelligent: Predict (with a more sophisticated predictor) if the load is dependent on any unknown address store

Handling of Store-Load Dependences

- **A load's dependence status is not known** until all previous store addresses are available.

- How does the OOO engine detect dependence of a load instruction on a previous store?
  - Option 1: Wait until all previous stores committed (no need to check for address match)
  - Option 2: Keep a list of pending stores in a store buffer and check whether load address matches a previous store address

- How does the OOO engine treat the scheduling of a load instruction wrt previous stores?
  - Option 1: Assume load dependent on all previous stores
  - Option 2: Assume load independent of all previous stores
  - Option 3: Predict the dependence of a load on an outstanding store

Memory Disambiguation (I)

- Option 1: Assume load is dependent on all previous stores

  + No need for recovery

  -- Too conservative: delays independent loads unnecessarily

- Option 2: Assume load is independent of all previous stores

  + Simple and can be common case: no delay for independent loads

  -- Requires recovery and re-execution of load and dependents on misprediction

- Option 3: Predict the dependence of a load on an outstanding store

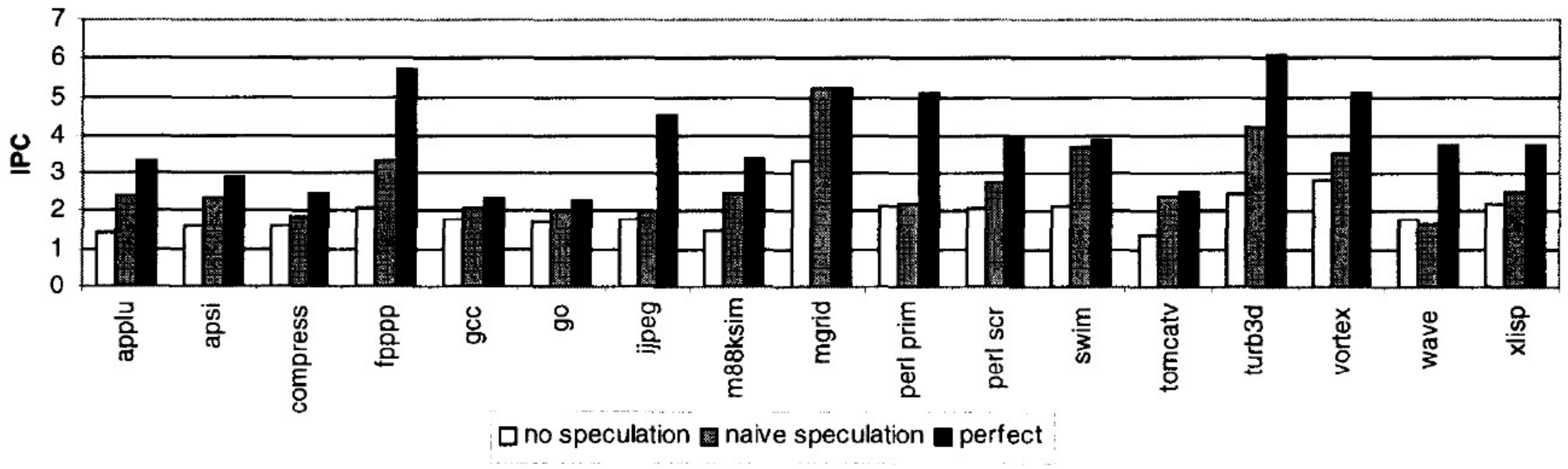
  + More accurate. Load store dependences persist over time

  -- Still requires recovery/re-execution on misprediction

  - Alpha 21264 : Initially assume load independent, delay loads found to be dependent
  - Moshovos et al., "Dynamic speculation and synchronization of data dependences," ISCA 1997.
  - Chrysos and Emer, "Memory Dependence Prediction Using Store Sets," ISCA 1998.

Memory Disambiguation (II)

- Chrysos and Emer, "Memory Dependence Prediction Using Store Sets," ISCA 1998.



- Predicting store-load dependences important for performance
- Simple predictors (based on past history) can achieve most of the potential performance

Data Forwarding Between Stores and Loads

- We cannot update memory out of program order
  → Need to buffer all store and load instructions in instruction window

- Even if we **know** all addresses of past stores when we generate the address of a load, two questions still remain:
  1. How do we check whether or not it is dependent on a store
  2. How do we forward data to the load if it is dependent on a store

- Modern processors use a LQ (load queue) and a SQ for this
  - Can be combined or separate between loads and stores
  - A load searches the SQ after it computes its address. Why?
  - A store searches the LQ after it computes its address. Why?

Out-of-Order Completion of Memory Ops

- When a store instruction finishes execution, it writes its address and data in its reorder buffer entry (or SQ entry)

- When a later load instruction generates its address, it:
  - searches the SQ with its address
  - accesses memory with its address
  - receives the value from the youngest older instruction that wrote to that address (either from ROB or memory)

- This is a complicated "search logic" implemented as a Content Addressable Memory
  - Content is "memory address" (but also need *size* and *age*)
  - Called **store-to-load forwarding logic**

Store-Load Forwarding Complexity

- Content Addressable Search (based on Load Address)
- Range Search (based on Address and Size of both the Load and earlier Stores)
- Age-Based Search (for last written values)
- Load data can come from a combination of multiple places
  - One or more stores in the Store Buffer (SQ)
  - Memory/cache