Design of Digital Circuits

Lecture 17: Out-of-Order, DataFlow, Superscalar Execution

Prof. Onur Mutlu
ETH Zurich

Spring 2018
27 April 2018

Agenda for Today & Next Few Lectures

- Single-cycle Microarchitectures
- Multi-cycle and Microprogrammed Microarchitectures
- Pipelining
- Issues in Pipelining: Control & Data Dependence Handling, State Maintenance and Recovery, …
- Out-of-Order Execution
- Other Execution Paradigms

Reminder: Optional Homeworks

- Posted online
  - 3 Optional Homeworks

- Optional

- Good for your learning

- https://safari.ethz.ch/digitaltechnik/spring2018/doku.php?id=homeworks

Readings for Today

- Smith and Sohi, "The Microarchitecture of Superscalar Processors," Proceedings of the IEEE, 1995
  - More advanced pipelining
  - Interrupt and exception handling
  - Out-of-order and superscalar execution concepts

- H&H Chapters 7.8 and 7.9

- Optional:
  - Kessler, "The Alpha 21264 Microprocessor," IEEE Micro 1999.

Lecture Announcement

- Monday, April 30, 2018
- 16:15-17:15
- CAB G 61
- Apéro after the lecture ☺

- Prof. Wen-Mei Hwu (University of Illinois at Urbana-Champaign)
- D-INFK Distinguished Colloquium
- **Innovative Applications and Technology Pivots – A Perfect Storm in Computing**

- https://www.inf.ethz.ch/news-and-events/colloquium/event-detail.html?eventFeedId=40447

General Suggestion

- Attend the **Computer Science Distinguished Colloquia**

- Happens on some Mondays
  - 16:15-17:15, followed by an apero

- https://www.inf.ethz.ch/news-and-events/colloquium.html

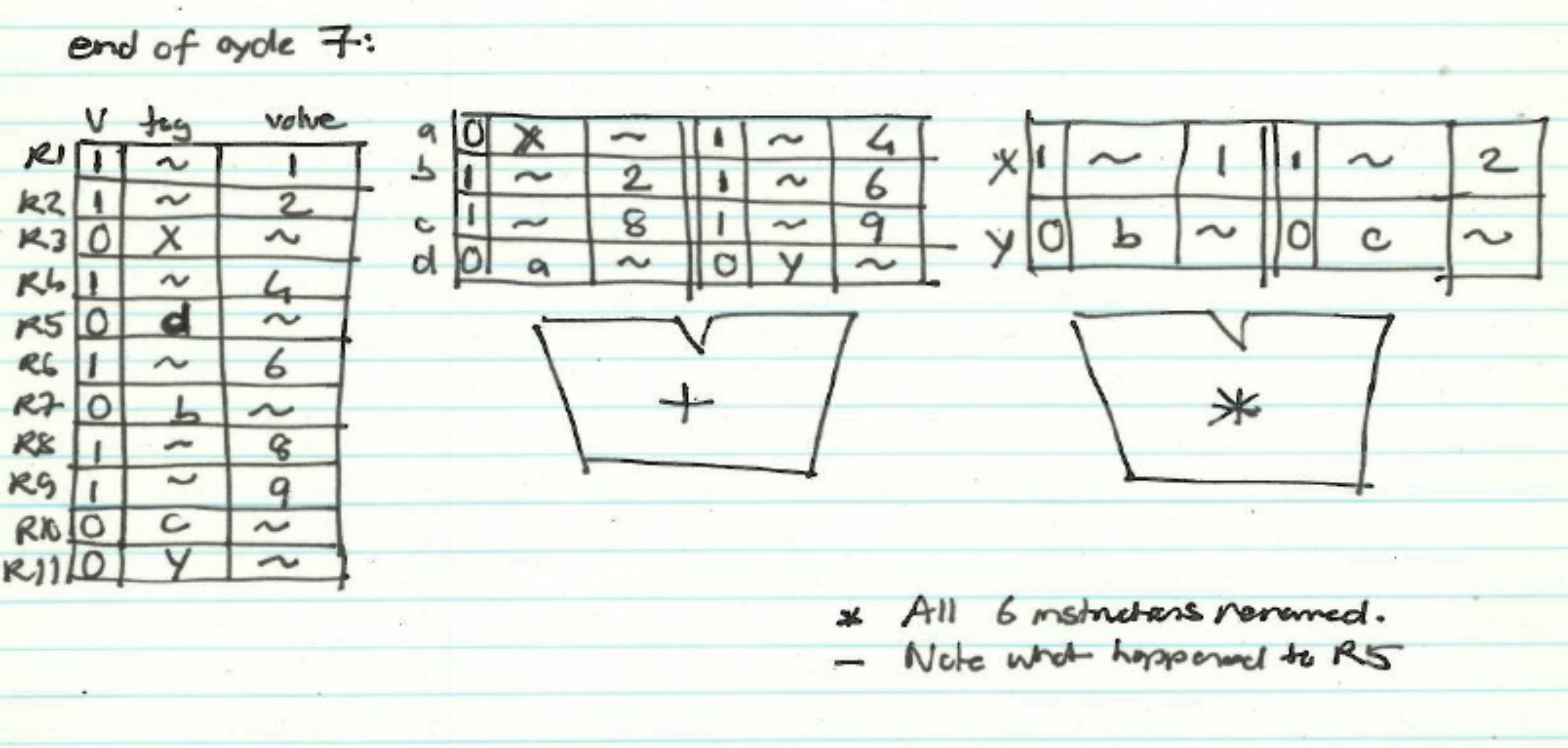
- Great way of learning about key developments in the field
  - And, meeting leading researchers

Out-of-Order Execution (Dynamic Instruction Scheduling)

Review: Dataflow Graph for Our Example

```
MUL   R3 ← R1, R2
ADD   R5 ← R3, R4
ADD   R7 ← R2, R6
ADD   R10 ← R8, R9
MUL   R11 ← R7, R10
ADD   R5 ← R5, R11
```

Review: State of RAT and RS in Cycle 7



end of cycle 7:

	V	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	~

a	0	x	~	1	~	4
b	1	~	2	1	~	6
c	1	~	8	1	~	9
d	0	a	~	0	y	~

x	1	~	1	1	~	2
y	0	b	~	0	c	~

* All 6 instructions renamed.
– Note what happened to R5

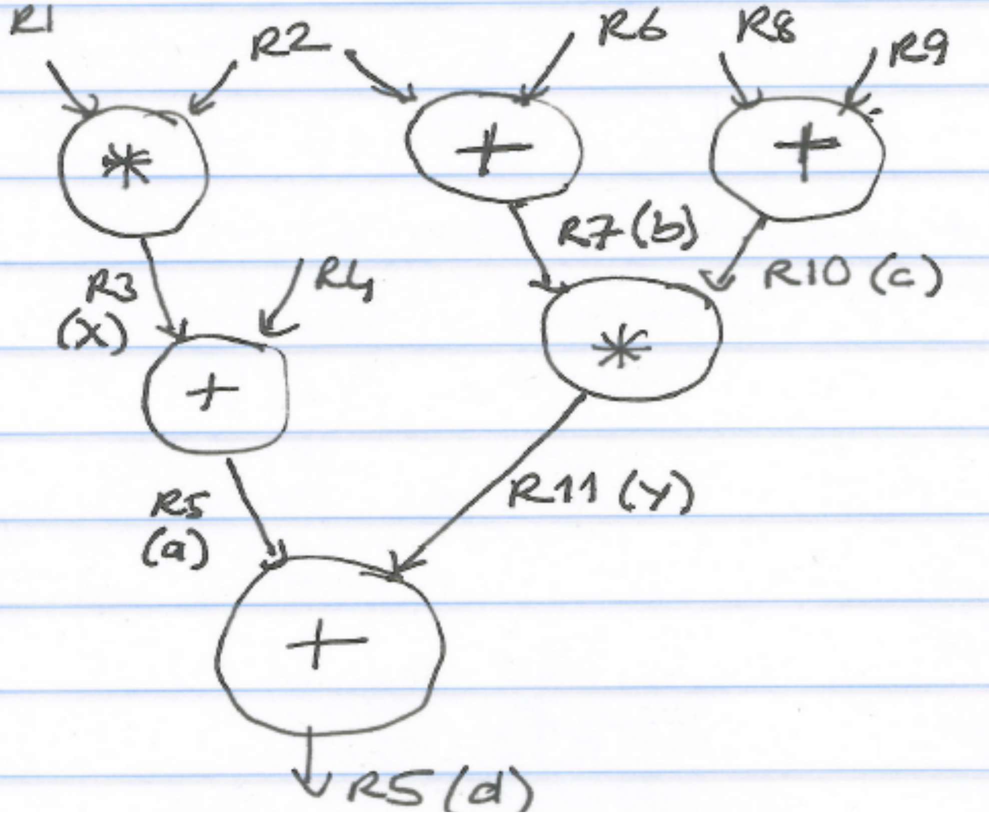
Review: Corresponding Dataflow Graph

```
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



Some More Questions (Design Choices)

- When is a reservation station entry deallocated?
- Exactly when does an instruction broadcast its tag?
- Should the reservation stations be dedicated to each functional unit or global across functional units?
  - Centralized vs. Distributed: What are the tradeoffs?
- Should reservation stations and ROB store data values or should there be a centralized physical register file where all data values are stored?
  - What are the tradeoffs?
- Many other design choices for OoO engines

For You: An Exercise, w/ Precise Exceptions

```
MUL   R3 ← R1, R2
ADD   R5 ← R3, R4
ADD   R7 ← R2, R6
ADD   R10 ← R8, R9
MUL   R11 ← R7, R10
ADD   R5 ← R5, R11
```

F	D	E	R	W

- Assume ADD (4 cycle execute), MUL (6 cycle execute)
- Assume one adder and one multiplier
- How many cycles
    - in a non-pipelined machine
    - in an in-order-dispatch pipelined machine with reorder buffer (no forwarding and full forwarding)
    - in an out-of-order dispatch pipelined machine with reorder buffer (full forwarding)

Out-of-Order Execution with Precise Exceptions

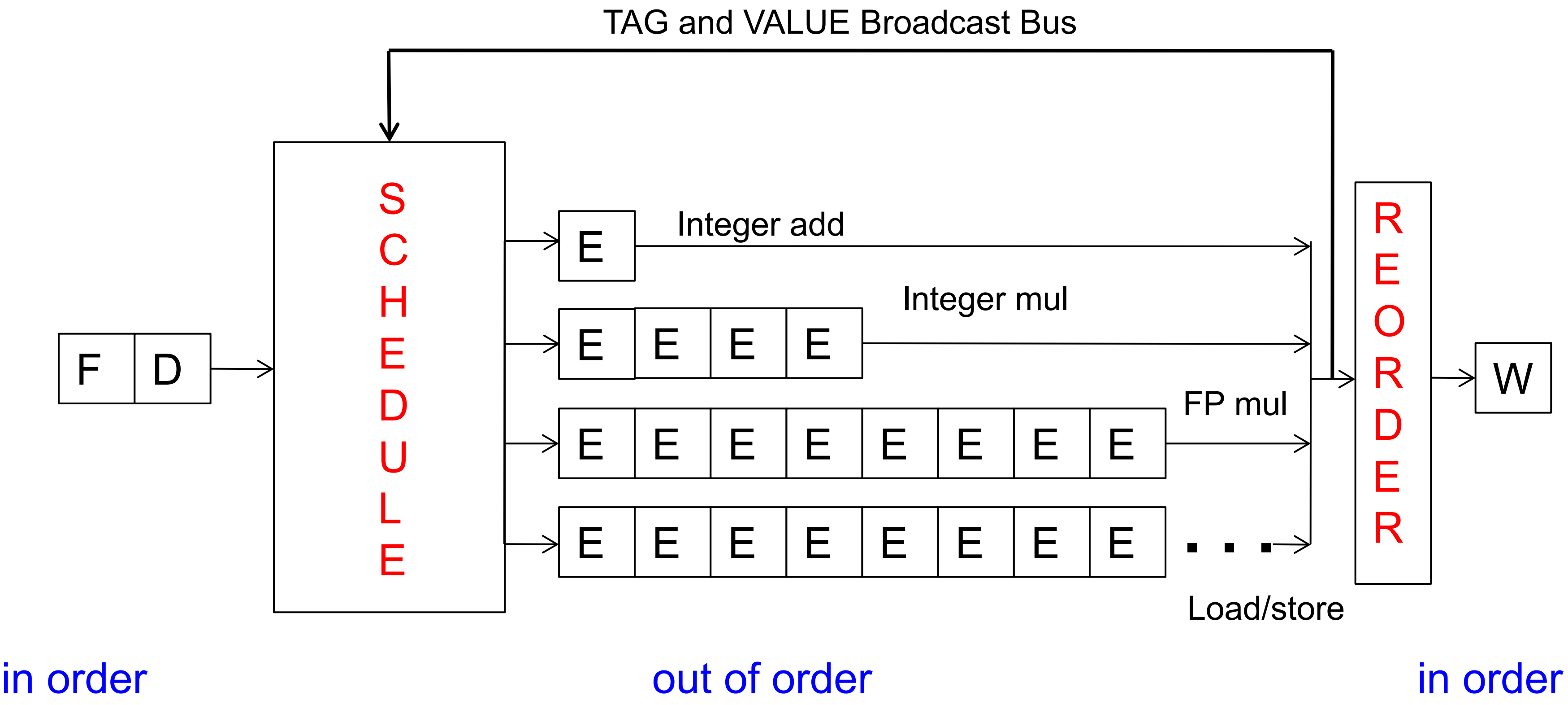
- Idea: Use a reorder buffer to reorder instructions before committing them to architectural state

- An instruction updates the RAT when it completes execution
  - Also called frontend register file

- An instruction updates a **separate** architectural register file when it retires
  - i.e., when it is the oldest in the machine and has completed execution
  - In other words, the architectural register file is always updated in program order

- On an exception: flush pipeline, copy architectural register file into frontend register file

Out-of-Order Execution with Precise Exceptions



- Hump 1: Reservation stations (scheduling window)
- Hump 2: Reordering (reorder buffer, aka instruction window or active window)

Two Humps in a Modern Pipeline

TAG and VALUE Broadcast B

Photo credit: http://true-wildlife.blogspot.ch/2010/10/bactrian-camel.html

Modern OoO Execution w/ Precise Exceptions

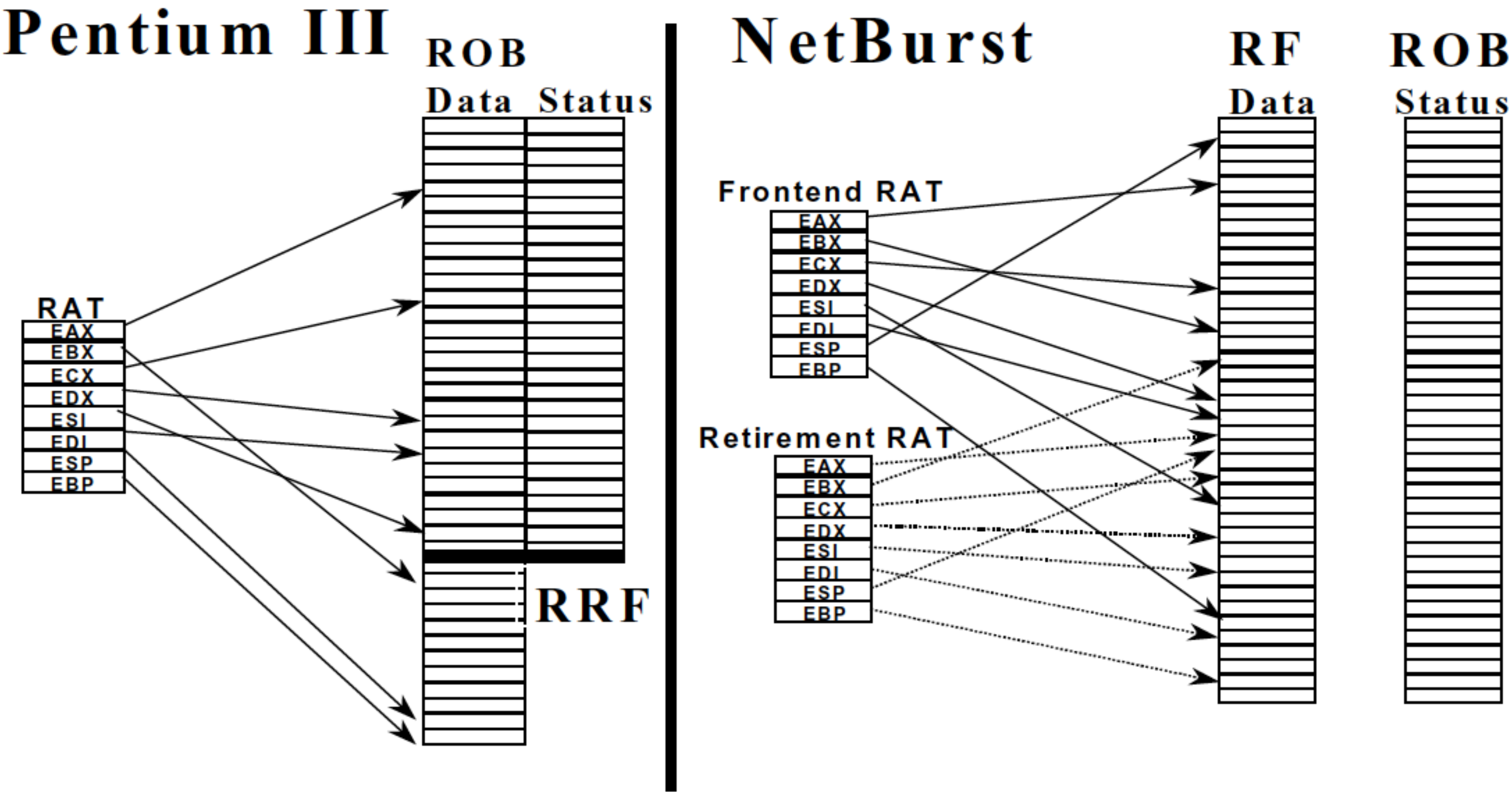
- Most modern processors use the following

- Reorder buffer to support in-order retirement of instructions

- A single register file to store all registers
  - Both speculative and architectural registers
  - INT and FP are still separate

- Two register maps
  - Future/frontend register map → used for renaming
  - Architectural register map → used for maintaining precise state

An Example from Modern Processors



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

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 dependencies, enables linking of producer to consumers

- Buffering enables the pipeline to move for independent ops

- 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
  - which piece?

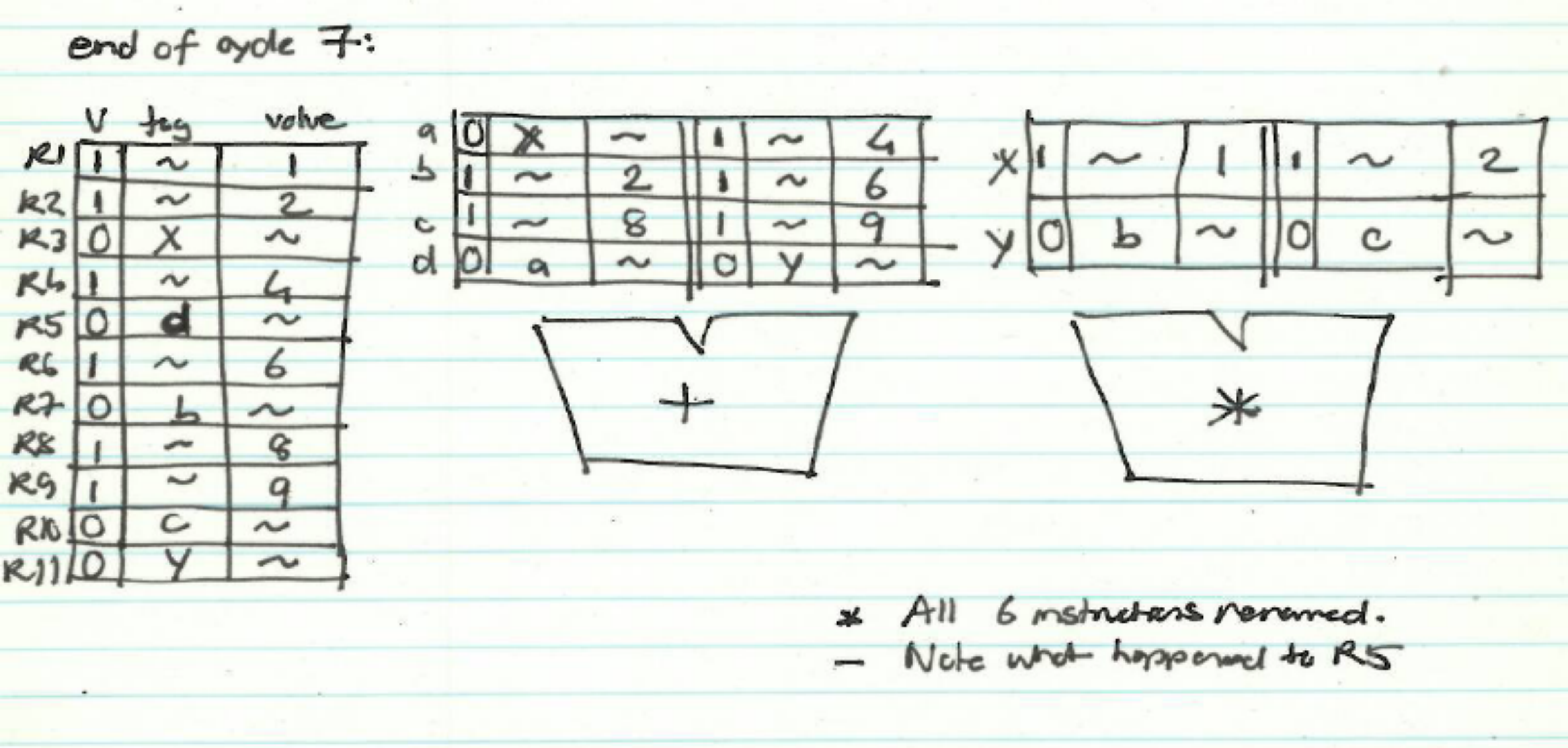
- The dataflow graph is limited to the **instruction window**
  - Instruction window: all decoded but not yet retired instructions

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

Recall: Dataflow Graph for Our Example

```
MUL   R3 ← R1, R2
ADD   R5 ← R3, R4
ADD   R7 ← R2, R6
ADD   R10 ← R8, R9
MUL   R11 ← R7, R10
ADD   R5 ← R5, R11
```

Recall: State of RAT and RS in Cycle 7



end of cycle 7:

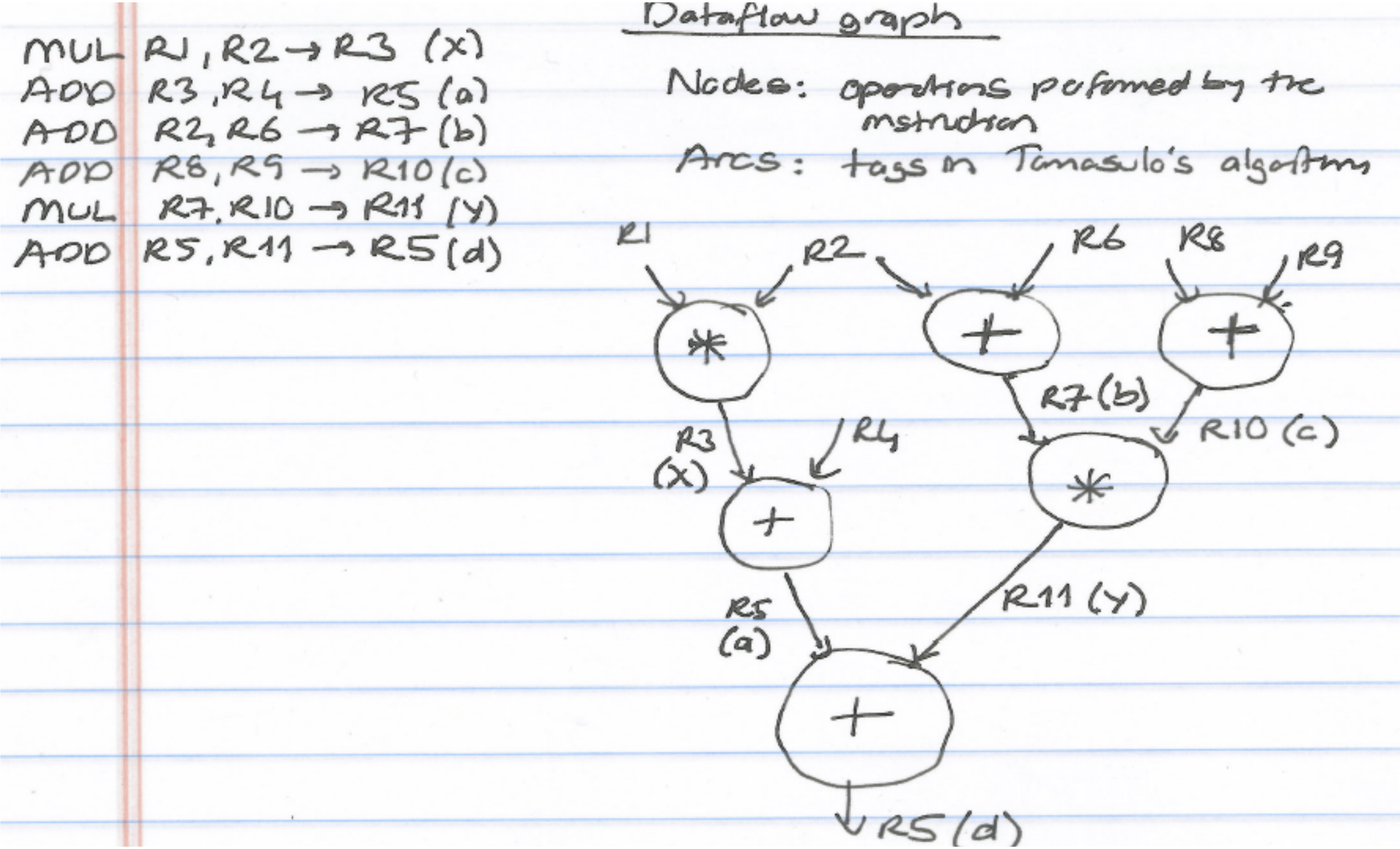
	V	tag	value
R1	1	~	1
R2	1	~	2
R3	0	X	~
R4	1	~	4
R5	0	d	~
R6	1	~	6
R7	0	b	~
R8	1	~	8
R9	1	~	9
R10	0	c	~
R11	0	y	~

a	0	x	~	1	~	4
b	1	~	2	1	~	6
c	1	~	8	1	~	9
d	0	a	~	0	y	~

x	1	~	1	1	~	2
y	0	b	~	0	c	~

- * All 6 instructions renamed.
- – Note what happened to R5

Recall: 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

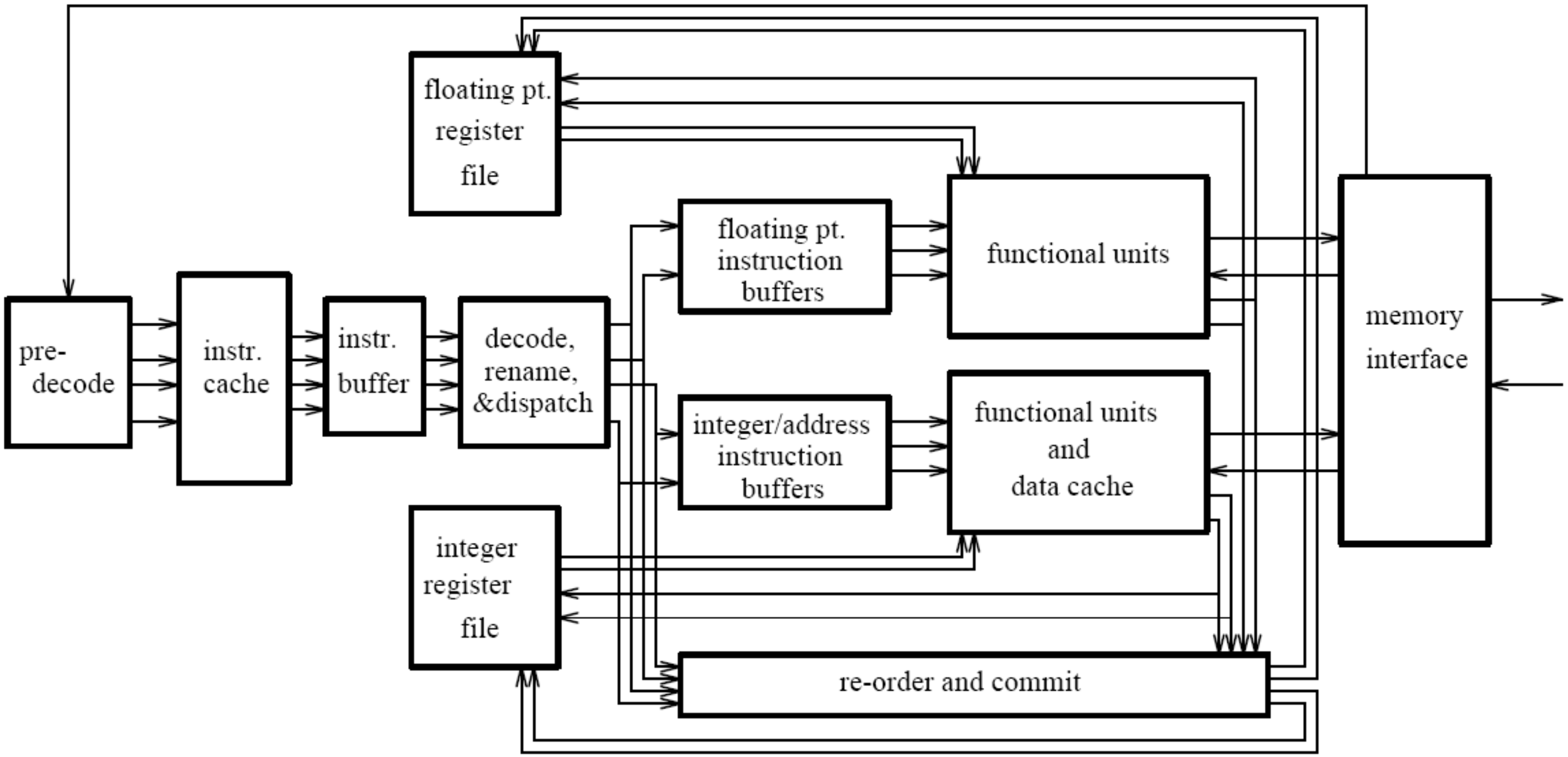
Arcs: tags in Tomasulo's algorithm

Questions to Ponder

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

- What if an instruction takes 500 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?
    - Active/instruction window size: determined by both scheduling window and reorder buffer size

General Organization of an OOO Processor



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

A Modern OoO Design: Intel Pentium 4

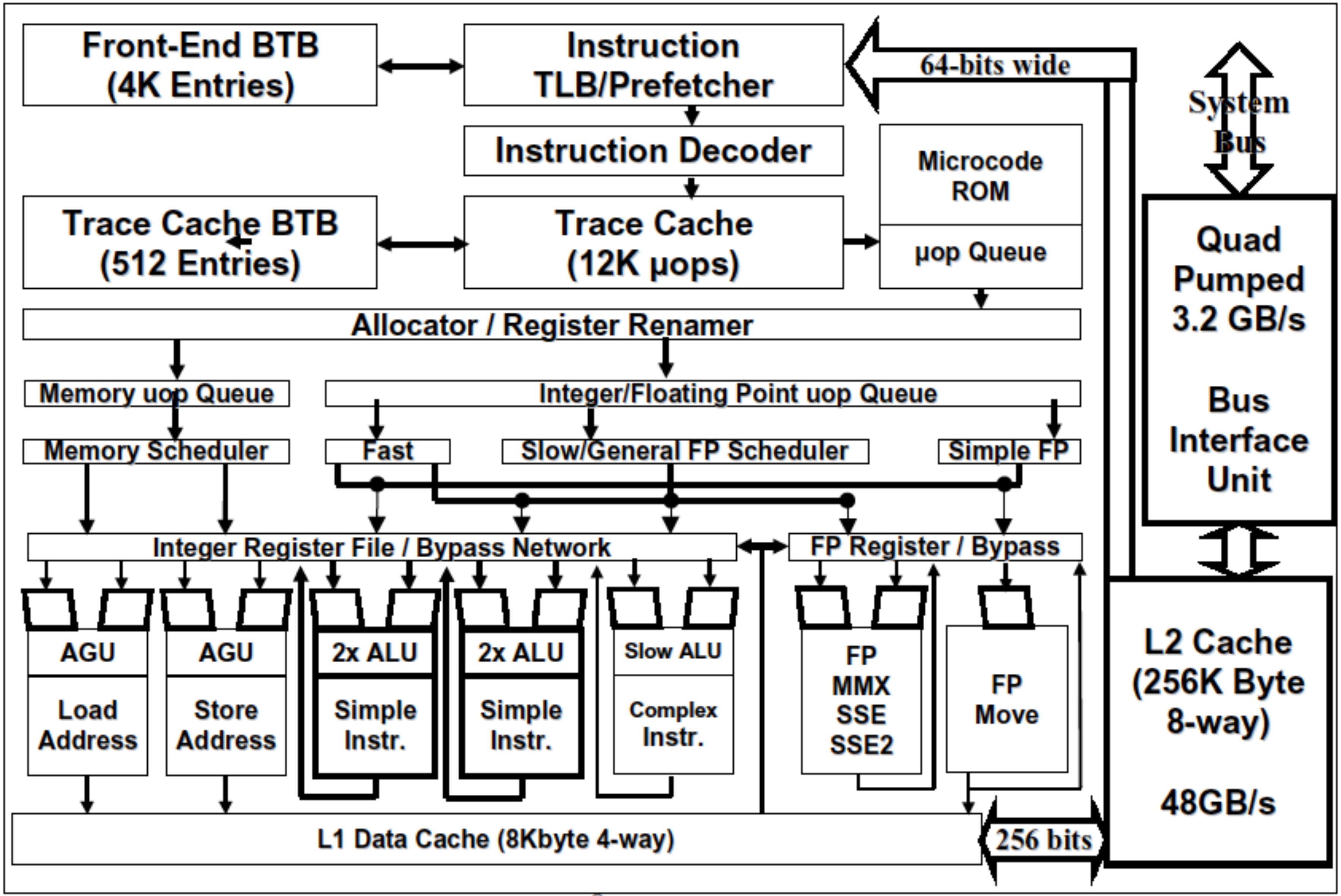
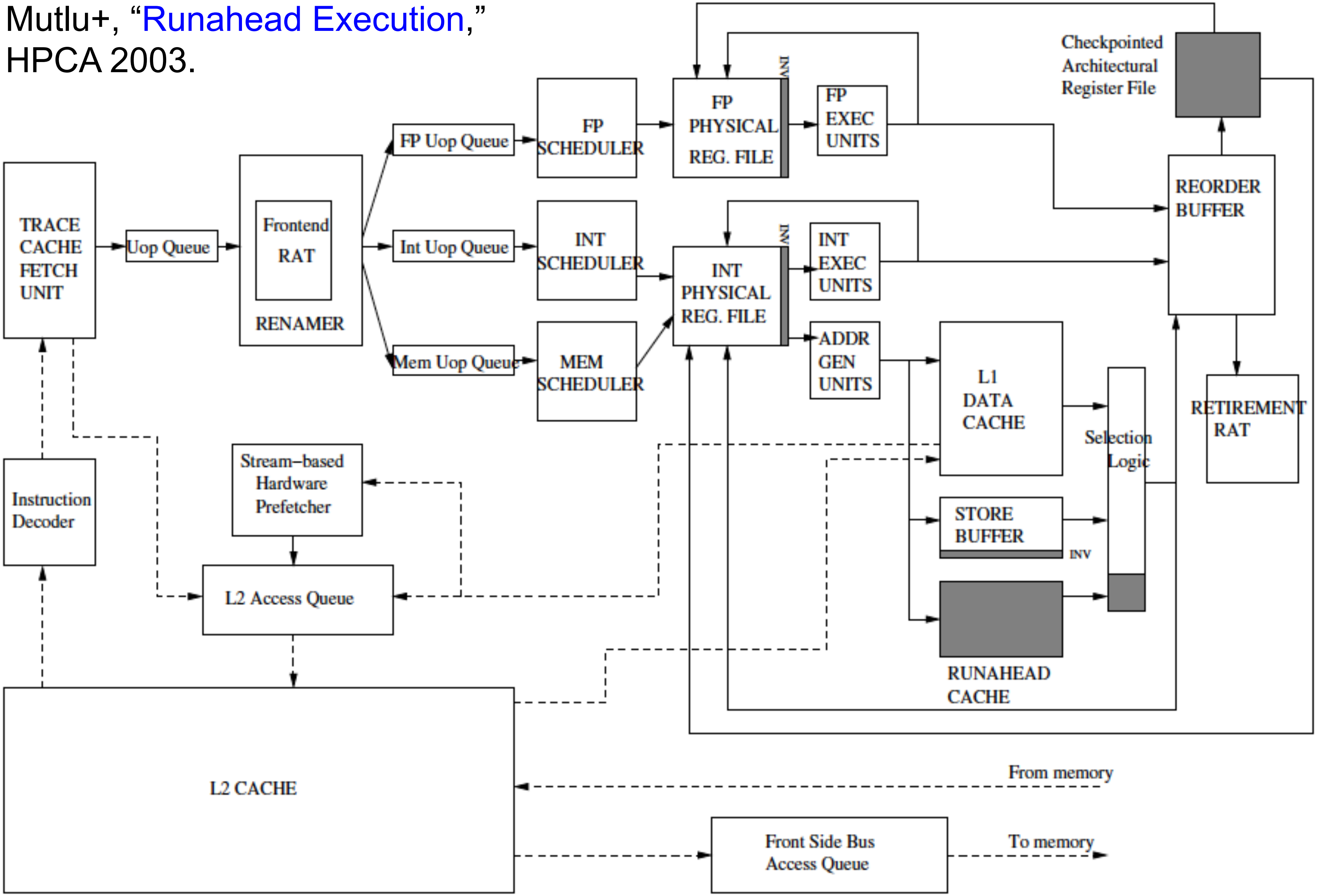


Figure 4: Pentium® 4 processor microarchitecture

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

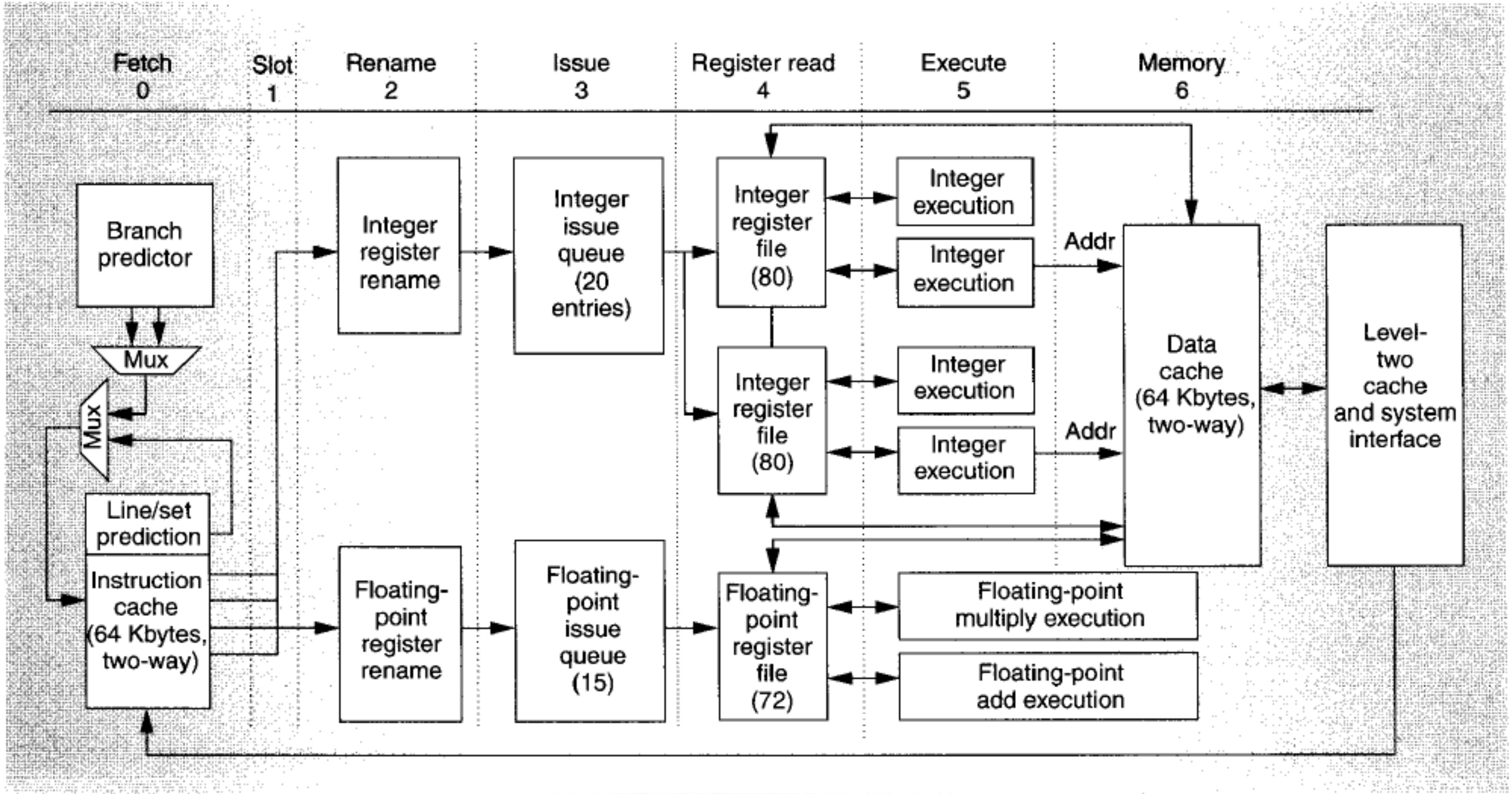
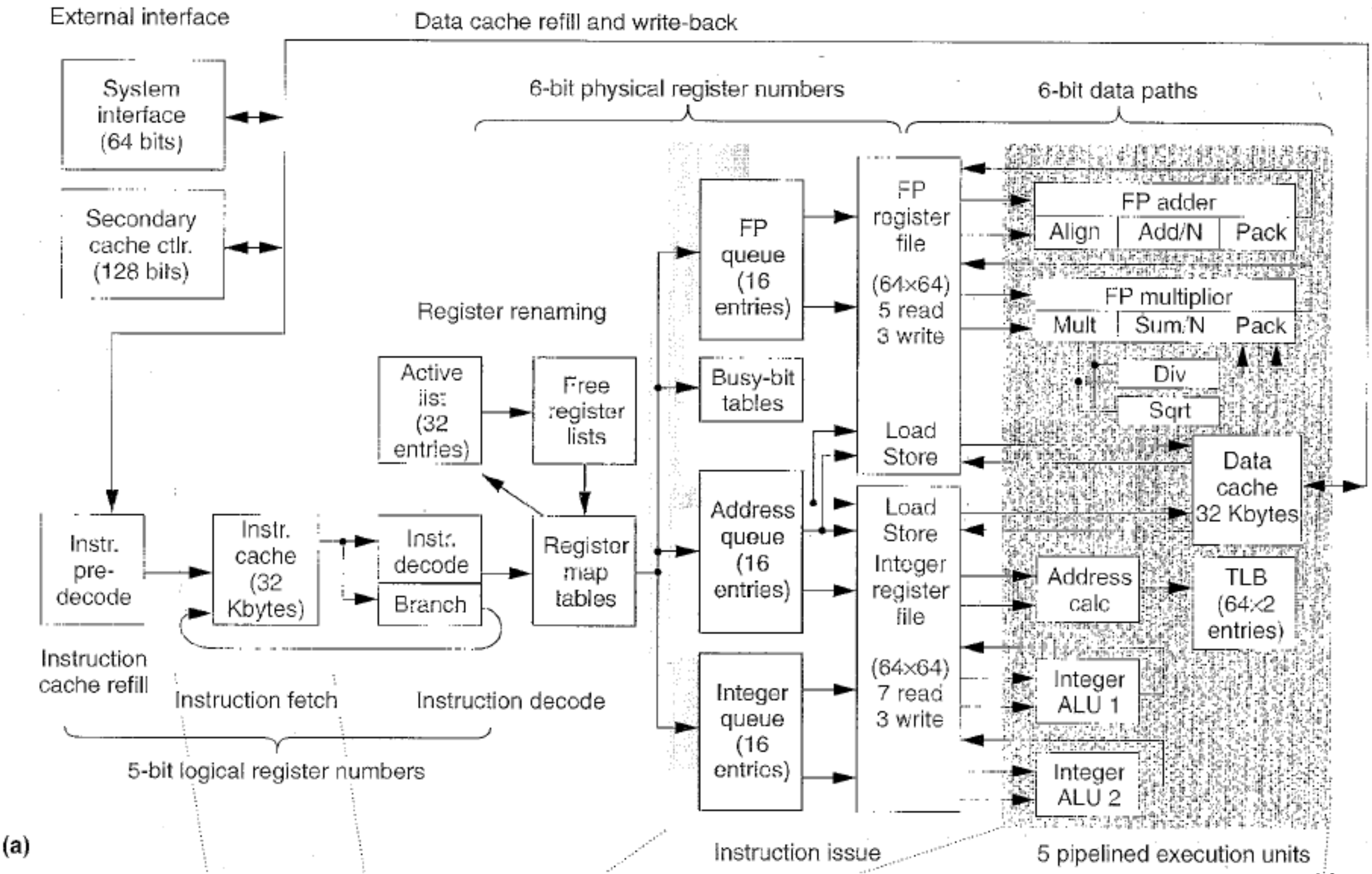

Fetch
0
Slot
1
Rename
2
Issue
3
Register read
4
Execute
5
Memory
6
Branch predictor
Mux
Mux
Line/set prediction
Instruction cache (64 Kbytes, two-way)
Integer register rename
Integer issue queue (20 entries)
Integer register file (80)
Integer register file (80)
Integer execution
Integer execution
Integer 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.

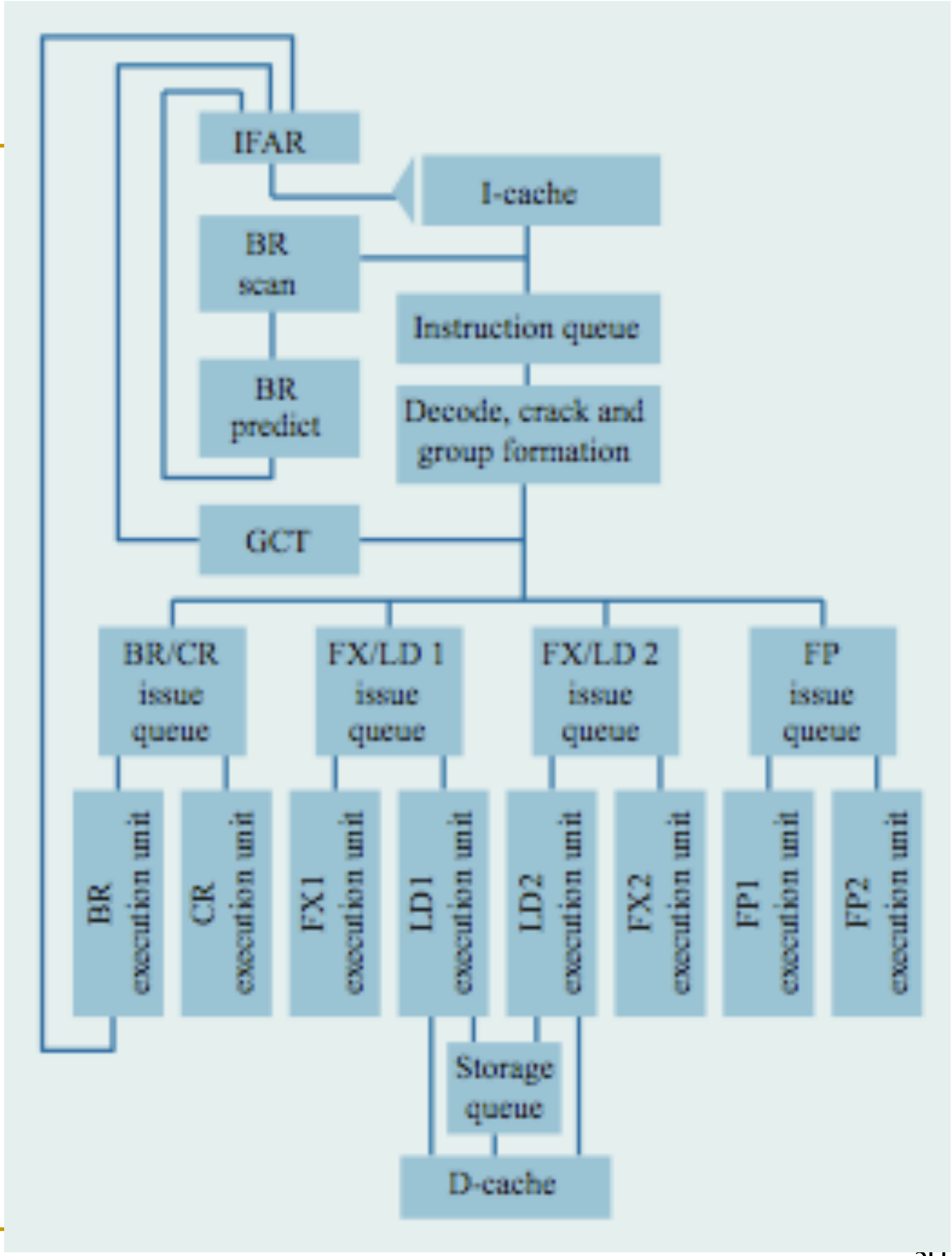
MIPS R10000



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.

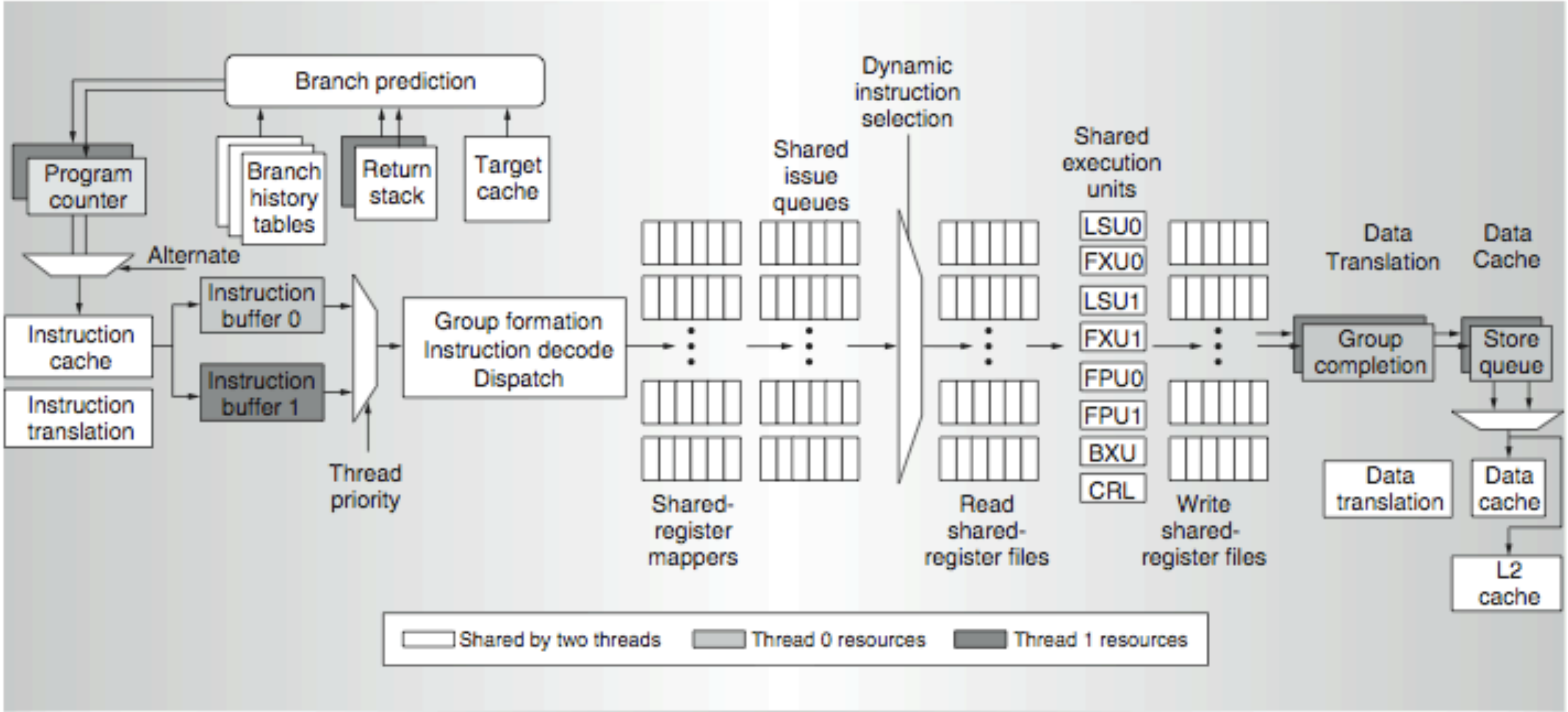


Figure 4. Power5 instruction data flow (BXU = branch execution unit and CRL = condition register logical execution unit).

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 need to be handled *after* their (partial) execution
- Corollary 3: When a load/store has its address ready, there may be younger/older loads/stores with undetermined 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 the/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

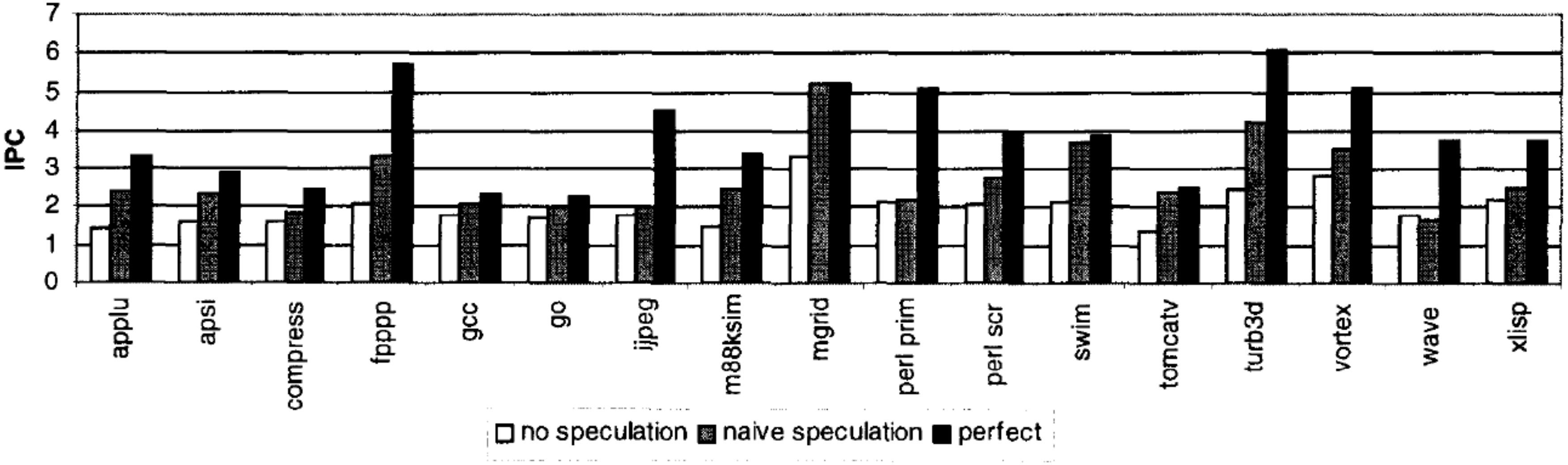
  + More accurate. Load store dependencies persist over time

  -- 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 dependencies 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 an 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

- When a later load instruction generates its address, it:
  - searches the reorder buffer (or 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

Other Approaches to Concurrency (or Instruction Level Parallelism)

Approaches to (Instruction-Level) Concurrency

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

Review: Data Flow:
Exploiting Irregular Parallelism

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 under the hood (while preserving sequential ISA semantics) have been very successful
  - Out of order execution is the prime example

Pure Data Flow Advantages/Disadvantages

- Advantages
  - Very good at exploiting irregular parallelism
  - Only real dependencies 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)
  - …

Approaches to (Instruction-Level) Concurrency

- Pipelining
- Out-of-order execution
- Dataflow (at the ISA level)
- Superscalar Execution
- VLIW
- Fine-Grained Multithreading
- 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

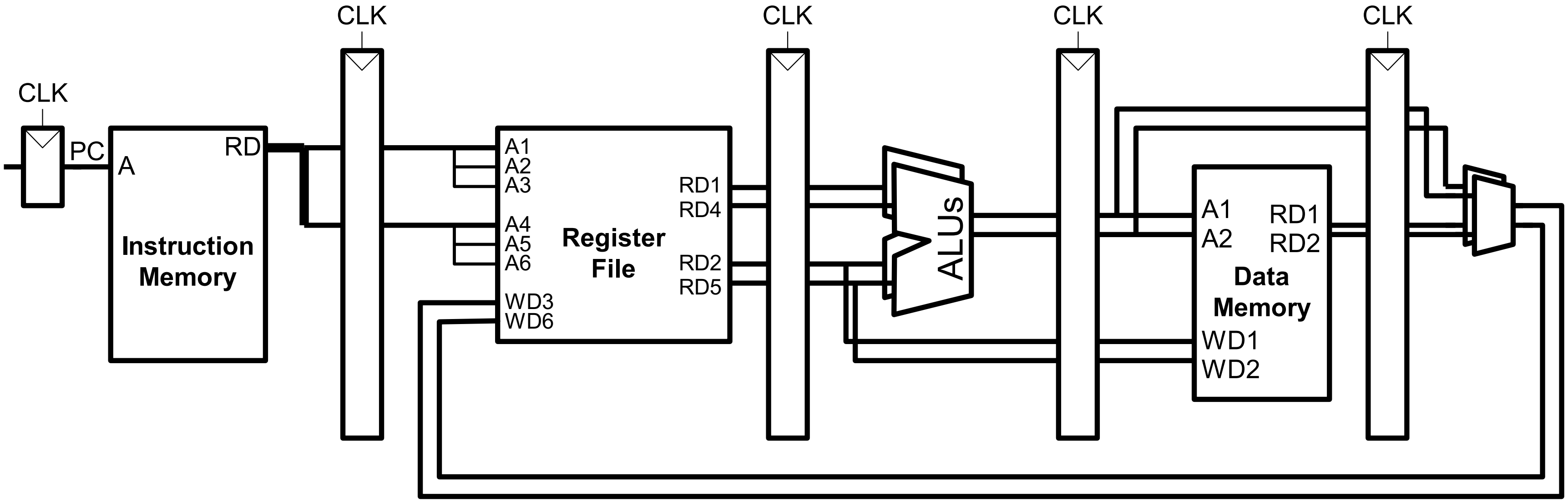
- Hardware performs the dependence checking between concurrently-fetched instructions

- 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 issue multiple instructions at per cycle

- Dependencies make it tricky to issue multiple instructions at once

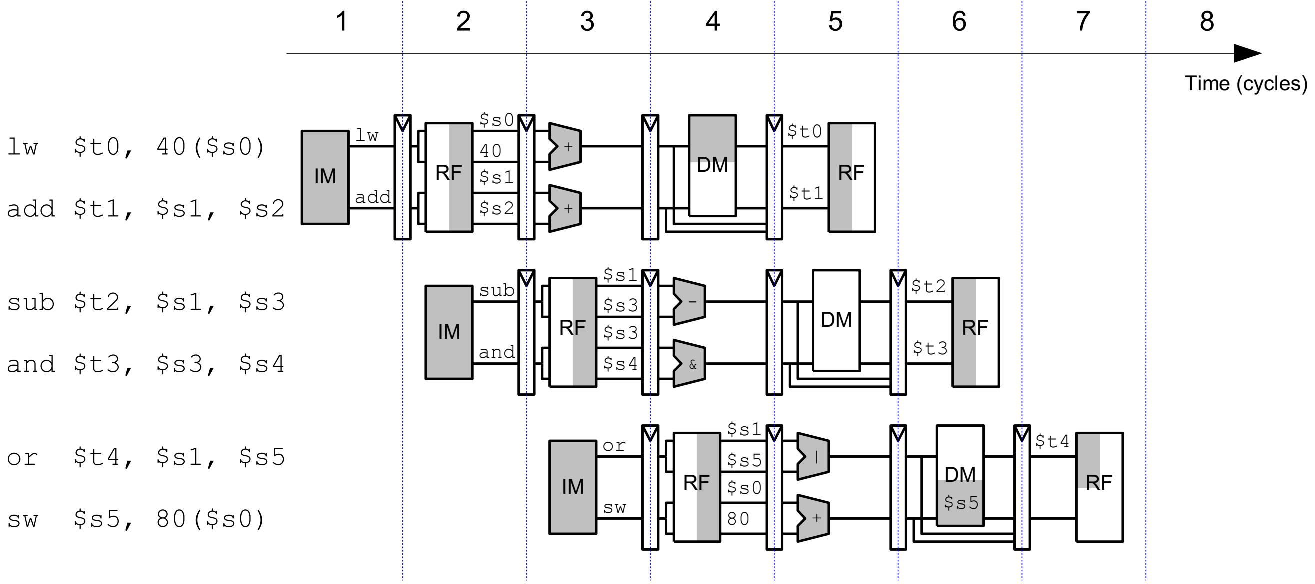


Here: Ideal IPC = 2

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

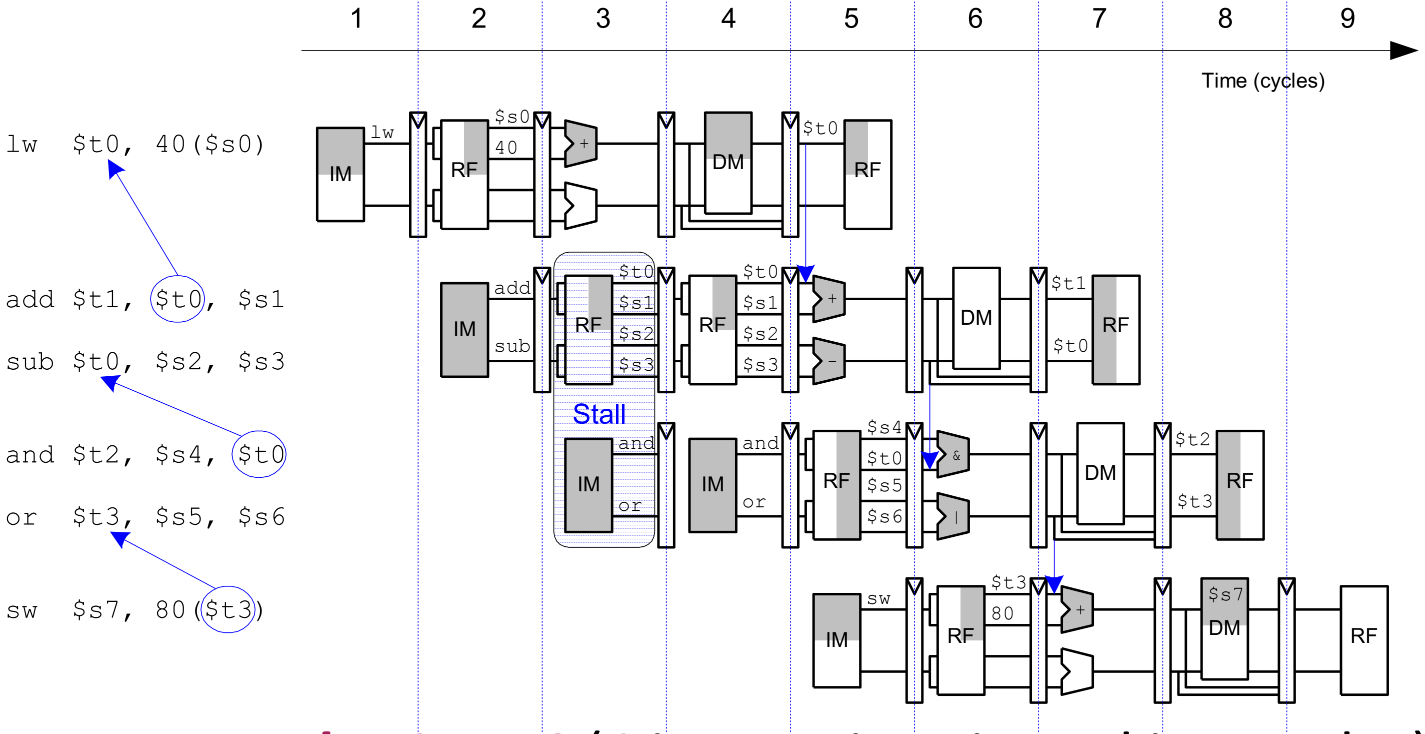


Actual IPC = 2 (6 instructions issued in 3 cycles)

Superscalar Performance with Dependencies

```
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



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

Superscalar Tradeoffs

- Advantages
  - Higher IPC (instructions per cycle)
- Disadvantages
  - Higher complexity for dependency checking
    - Require checking within a pipeline stage
    - Renaming becomes more complex in an OoO processor
  - More hardware resources needed

Design of Digital Circuits

Lecture 17: Out-of-Order, DataFlow, Superscalar Execution

Prof. Onur Mutlu

ETH Zurich

Spring 2018

27 April 2018

We did not cover the following slides in lecture. These are for your preparation for the next lecture.

Approaches to (Instruction-Level) Concurrency

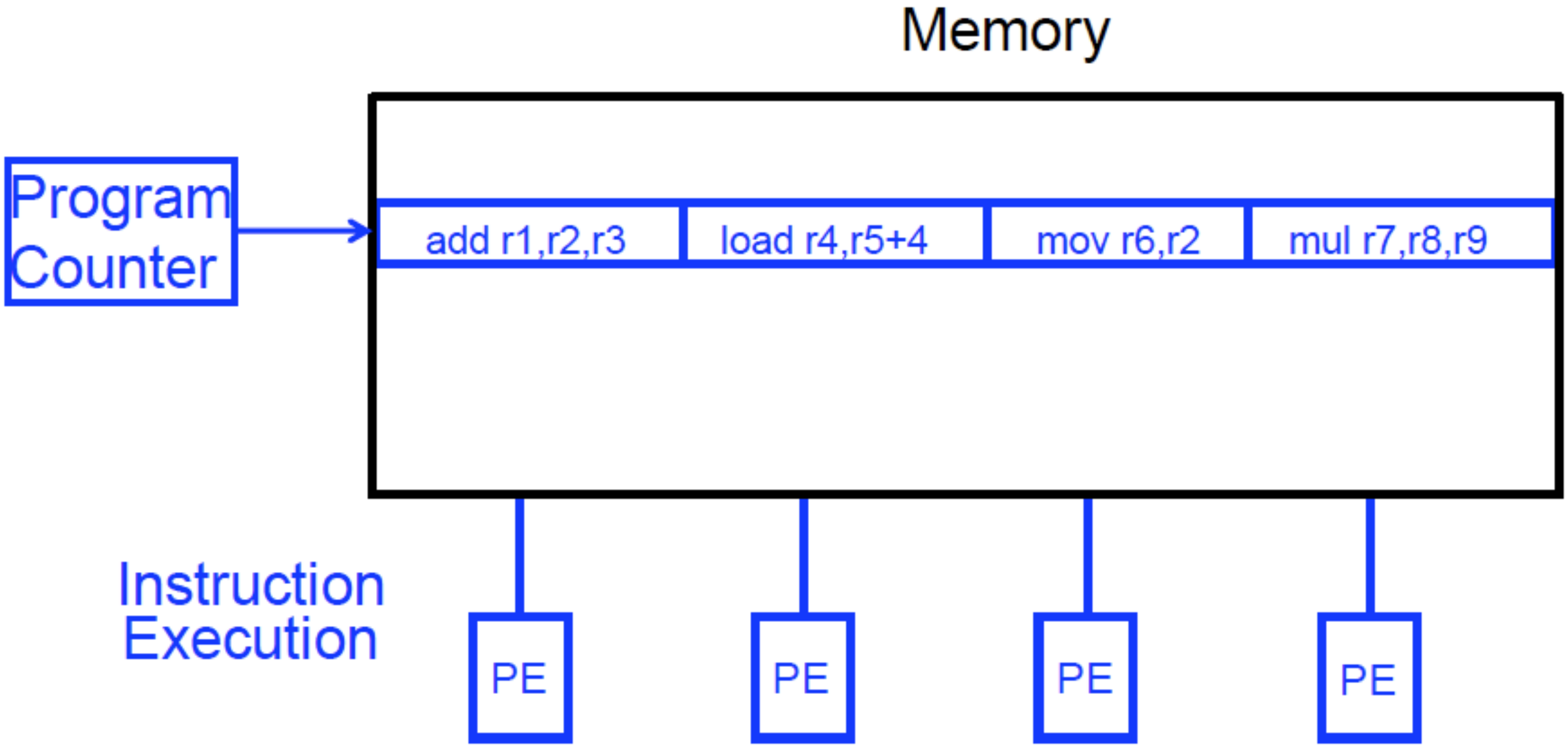
- Pipelining
- Out-of-order execution
- Dataflow (at the ISA level)
- Superscalar Execution
- VLIW
- Fine-Grained Multithreading
- SIMD Processing (Vector and array processors, GPUs)
- Decoupled Access Execute
- Systolic Arrays

VLIW

VLIW Concept

- Superscalar
  - Hardware fetches multiple instructions and checks dependencies between them
- VLIW (Very Long Instruction Word)
  - Software (compiler) packs independent instructions in a larger "instruction bundle" to be fetched and executed concurrently
  - Hardware fetches and executes the instructions in the bundle concurrently
- No need for hardware dependency checking between concurrently-fetched instructions in the VLIW model

VLIW Concept



- Fisher, "Very Long Instruction Word architectures and the ELI-512," ISCA 1983.
  - ELI: Enormously longword instructions (512 bits)

VLIW (Very Long Instruction Word)

- A very long instruction word consists of multiple independent instructions packed together by the compiler
  - Packed instructions can be logically unrelated (contrast with SIMD/vector processors, which we will see soon)

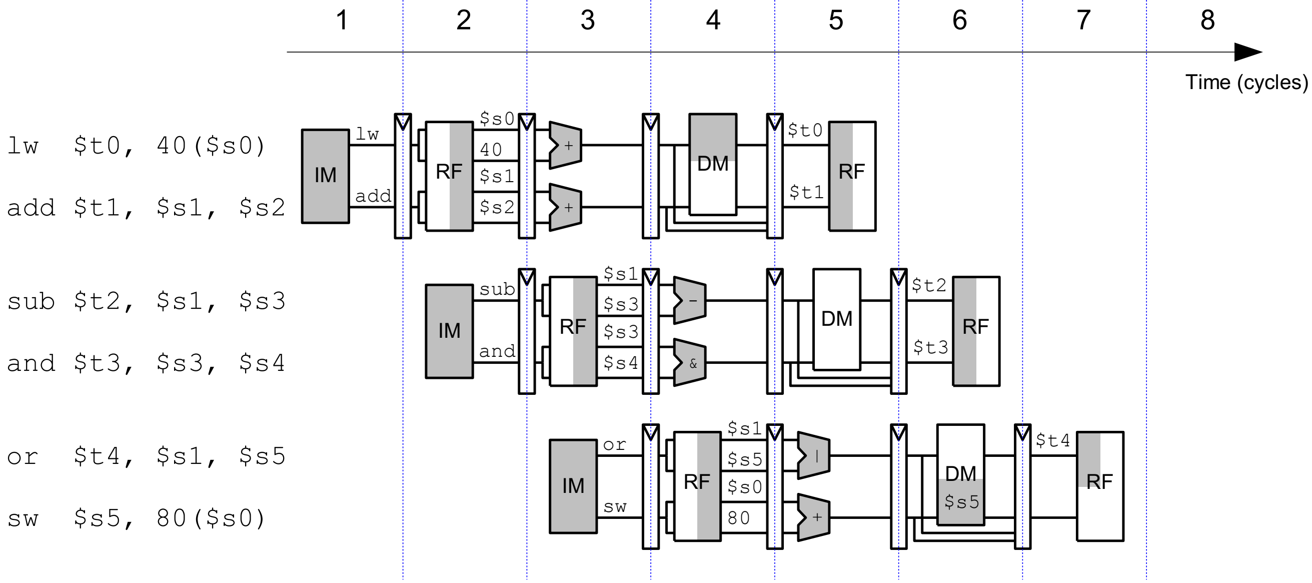
- Idea: Compiler finds independent instructions and statically schedules (i.e. packs/bundles) them into a single VLIW instruction

- Traditional Characteristics
  - Multiple functional units
  - All instructions in a bundle are executed in lock step
  - Instructions in a bundle statically aligned to be directly fed into the functional units

VLIW Performance Example (2-wide bundles)

```
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)

VLIW Lock-Step Execution

- Lock-step (all or none) execution: If any operation in a VLIW instruction stalls, all instructions stall

- In a truly VLIW machine, the compiler handles all dependency-related stalls, hardware does not perform dependency checking
  - What about variable latency operations?

VLIW Philosophy

- Philosophy similar to RISC (simple instructions and hardware)
  - Except multiple instructions in parallel

- RISC (John Cocke, 1970s, IBM 801 minicomputer)
  - Compiler does the hard work to translate high-level language code to simple instructions (John Cocke: control signals)
    - And, to reorder simple instructions for high performance
  - Hardware does little translation/decoding → very simple

- VLIW (Josh Fisher, ISCA 1983)
  - Compiler does the hard work to find instruction level parallelism
  - Hardware stays as simple and streamlined as possible
    - Executes each instruction in a bundle in lock step
    - Simple → higher frequency, easier to design

Commercial VLIW Machines

- Multiflow TRACE, Josh Fisher (7-wide, 28-wide)
- Cydrome Cydra 5, Bob Rau
- Transmeta Crusoe: x86 binary-translated into internal VLIW
- TI C6000, Trimedia, STMicro (DSP & embedded processors)
  - Most successful commercially

- Intel IA-64
  - Not fully VLIW, but based on VLIW principles
  - EPIC (Explicitly Parallel Instruction Computing)
  - Instruction bundles can have dependent instructions
  - A few bits in the instruction format specify explicitly which instructions in the bundle are dependent on which other ones

VLIW Tradeoffs

- Advantages

  + No need for dynamic scheduling hardware → simple hardware

  + No need for dependency checking within a VLIW instruction → simple hardware for multiple instruction issue + no renaming

  + No need for instruction alignment/distribution after fetch to different functional units → simple hardware

- Disadvantages

  -- Compiler needs to find N independent operations per cycle

    -- If it cannot, inserts NOPs in a VLIW instruction

    -- Parallelism loss AND code size increase

  -- Recompilation required when execution width (N), instruction latencies, functional units change (Unlike superscalar processing)

  -- Lockstep execution causes independent operations to stall

    -- No instruction can progress until the longest-latency instruction completes

VLIW Summary

- VLIW simplifies hardware, but requires complex compiler techniques
- Solely-compiler approach of VLIW has several downsides that reduce performance

  -- Too many NOPs (not enough parallelism discovered)

  -- Static schedule intimately tied to microarchitecture

    -- Code optimized for one generation performs poorly for next

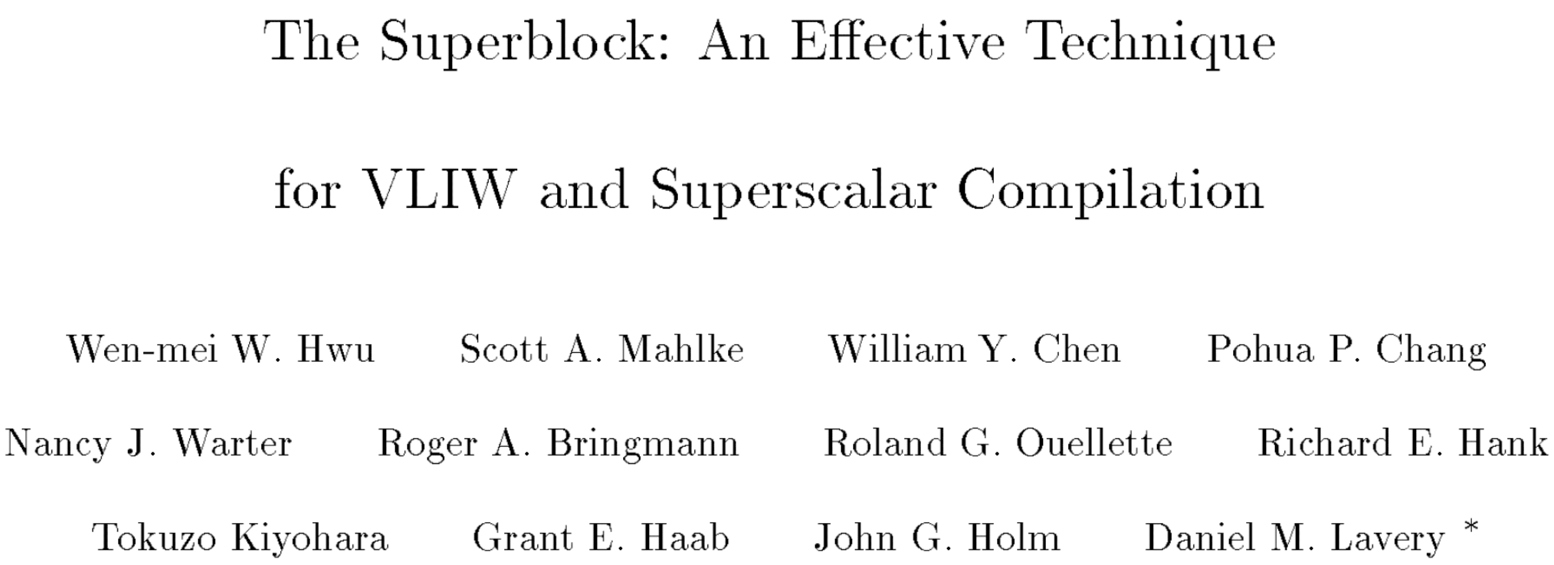
  -- No tolerance for variable or long-latency operations (lock step)

++ Most compiler optimizations developed for VLIW employed in optimizing compilers (for superscalar compilation)

- Enable code optimizations

++ VLIW successful when parallelism is easier to find by the compiler (traditionally embedded markets, DSPs)

An Example Work: Superblock



The Superblock: An Effective Technique

for VLIW and Superscalar Compilation

Wen-mei W. Hwu Scott A. Mahlke William Y. Chen Pohua P. Chang

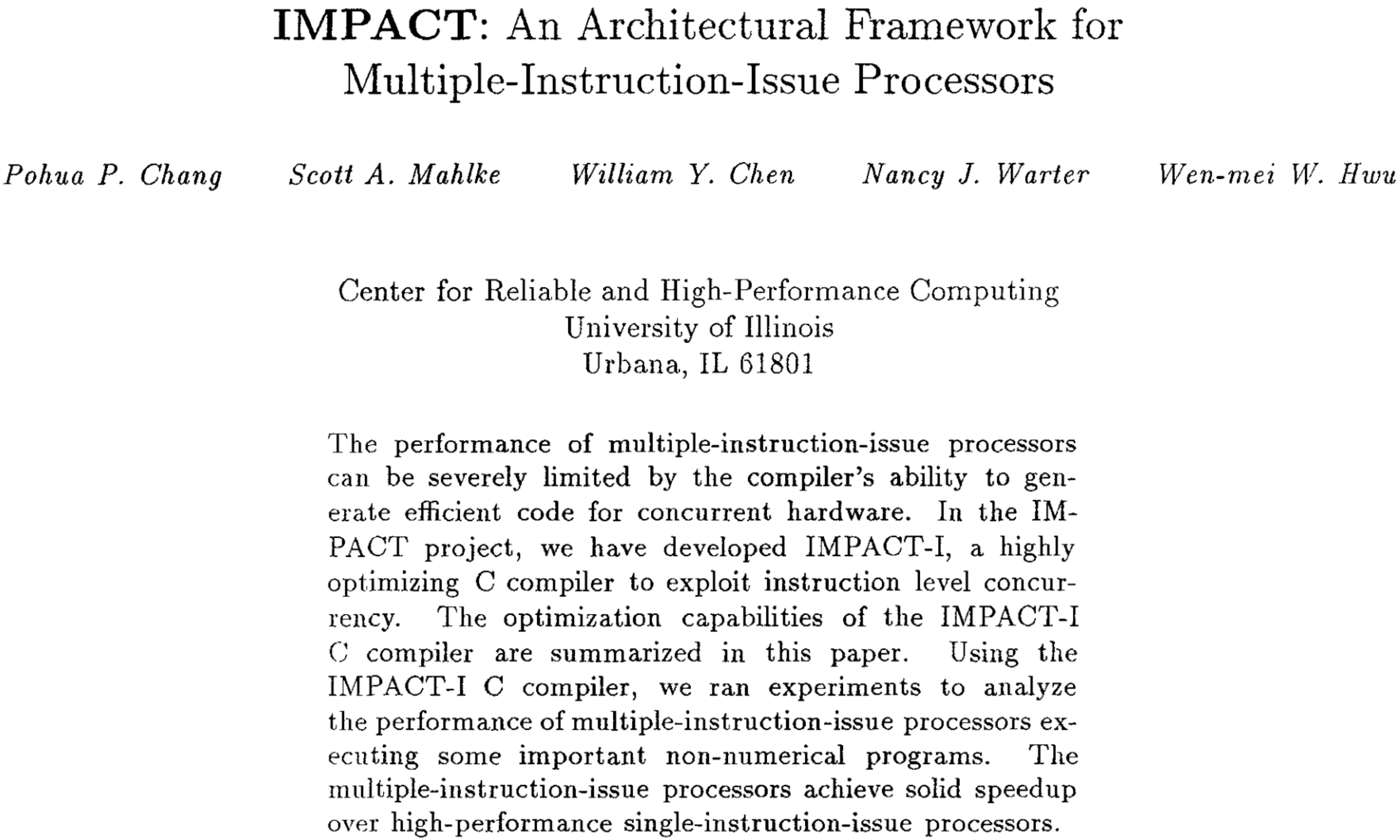
Nancy J. Warter Roger A. Bringmann Roland G. Ouellette Richard E. Hank

Tokuzo Kiyohara Grant E. Haab John G. Holm Daniel M. Lavery *

Hwu et al., The superblock: An effective technique for VLIW and superscalar compilation.
The Journal of Supercomputing, 1993.

- Lecture Video on Static Instruction Scheduling
  - https://www.youtube.com/watch?v=isBEVkIjgGA

Another Example Work: IMPACT



IMPACT: An Architectural Framework for Multiple-Instruction-Issue Processors

Pohua P. Chang *Scott A. Mahlke* *William Y. Chen* *Nancy J. Warter* *Wen-mei W. Hwu*

Center for Reliable and High-Performance Computing
University of Illinois
Urbana, IL 61801

The performance of multiple-instruction-issue processors can be severely limited by the compiler's ability to generate efficient code for concurrent hardware. In the IMPACT project, we have developed IMPACT-I, a highly optimizing C compiler to exploit instruction level concurrency. The optimization capabilities of the IMPACT-I C compiler are summarized in this paper. Using the IMPACT-I C compiler, we ran experiments to analyze the performance of multiple-instruction-issue processors executing some important non-numerical programs. The multiple-instruction-issue processors achieve solid speedup over high-performance single-instruction-issue processors.

Chang et al., IMPACT: an architectural framework for multiple-instruction-issue processors. ISCA 1991.

Approaches to (Instruction-Level) Concurrency

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

Recall: How to Handle Data Dependences

- Anti and output dependences are easier to handle
  - write to the destination in one stage and in program order

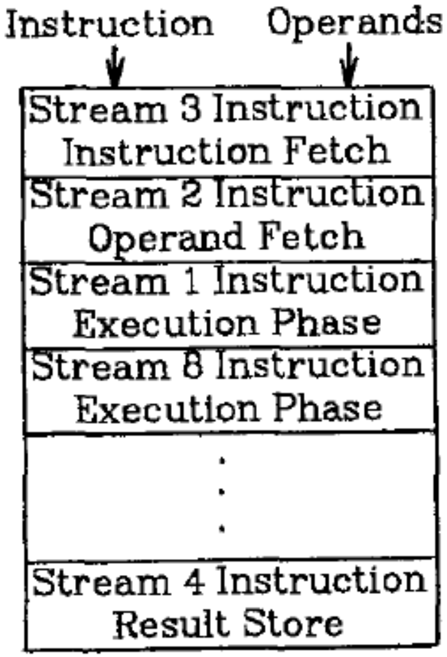
- Flow dependences are more interesting

- Five 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
  - Predict the needed value(s), execute “speculatively”, and verify
  - Do something else (fine-grained multithreading)
    - No need to detect

Fine-Grained Multithreading

Fine-Grained Multithreading

- Idea: Hardware has multiple thread contexts (PC+registers). Each cycle, fetch engine fetches from a different thread.
  - By the time the fetched branch/instruction resolves, no instruction is fetched from the same thread
  - Branch/instruction resolution latency overlapped with execution of other threads’ instructions



+ No logic needed for handling control and data dependences within a thread

-- Single thread performance suffers

-- Extra logic for keeping thread contexts

-- Does not overlap latency if not enough threads to cover the whole pipeline

Fine-Grained Multithreading (II)

- Idea: Switch to another thread every cycle such that no two instructions from a thread are in the pipeline concurrently

- Tolerates the control and data dependency latencies by overlapping the latency with useful work from other threads
- Improves pipeline utilization by taking advantage of multiple threads

- Thornton, "Parallel Operation in the Control Data 6600," AFIPS 1964.
- Smith, "A pipelined, shared resource MIMD computer," ICPP 1978.

Fine-Grained Multithreading: History

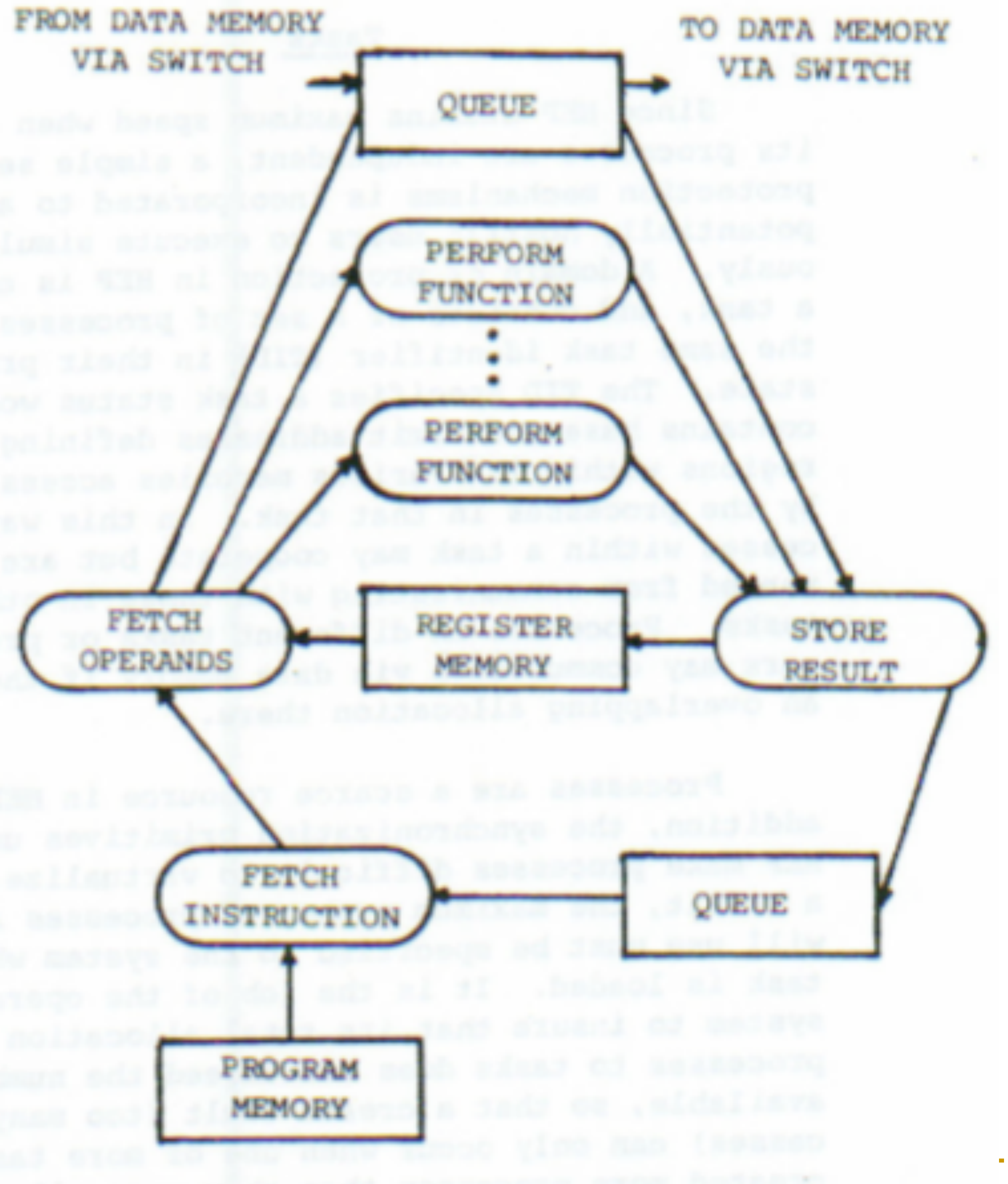
- CDC 6600's peripheral processing unit is fine-grained multithreaded
  - Thornton, "Parallel Operation in the Control Data 6600," AFIPS 1964.
  - Processor executes a different I/O thread every cycle
  - An operation from the same thread is executed every 10 cycles

- Denelcor HEP (Heterogeneous Element Processor)
  - Smith, "A pipelined, shared resource MIMD computer," ICPP 1978.
  - 120 threads/processor
  - available queue vs. unavailable (waiting) queue for threads
  - each thread can have only 1 instruction in the processor pipeline; each thread independent
  - to each thread, processor looks like a non-pipelined machine
  - system throughput vs. single thread performance tradeoff

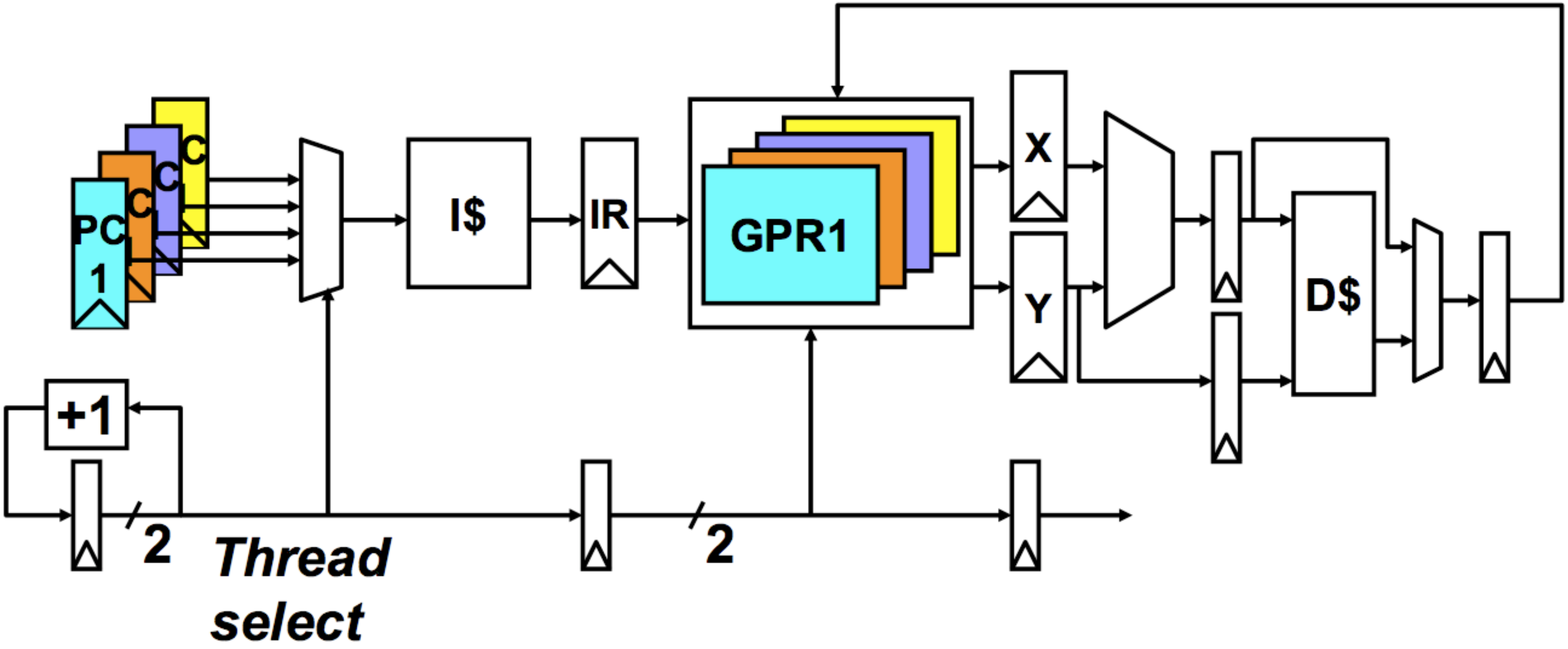
Fine-Grained Multithreading in HEP

- Cycle time: 100ns
- 8 stages → 800 ns to complete an instruction
  - assuming no memory access
- No control and data dependency checking

Burton Smith
(1941-2018)

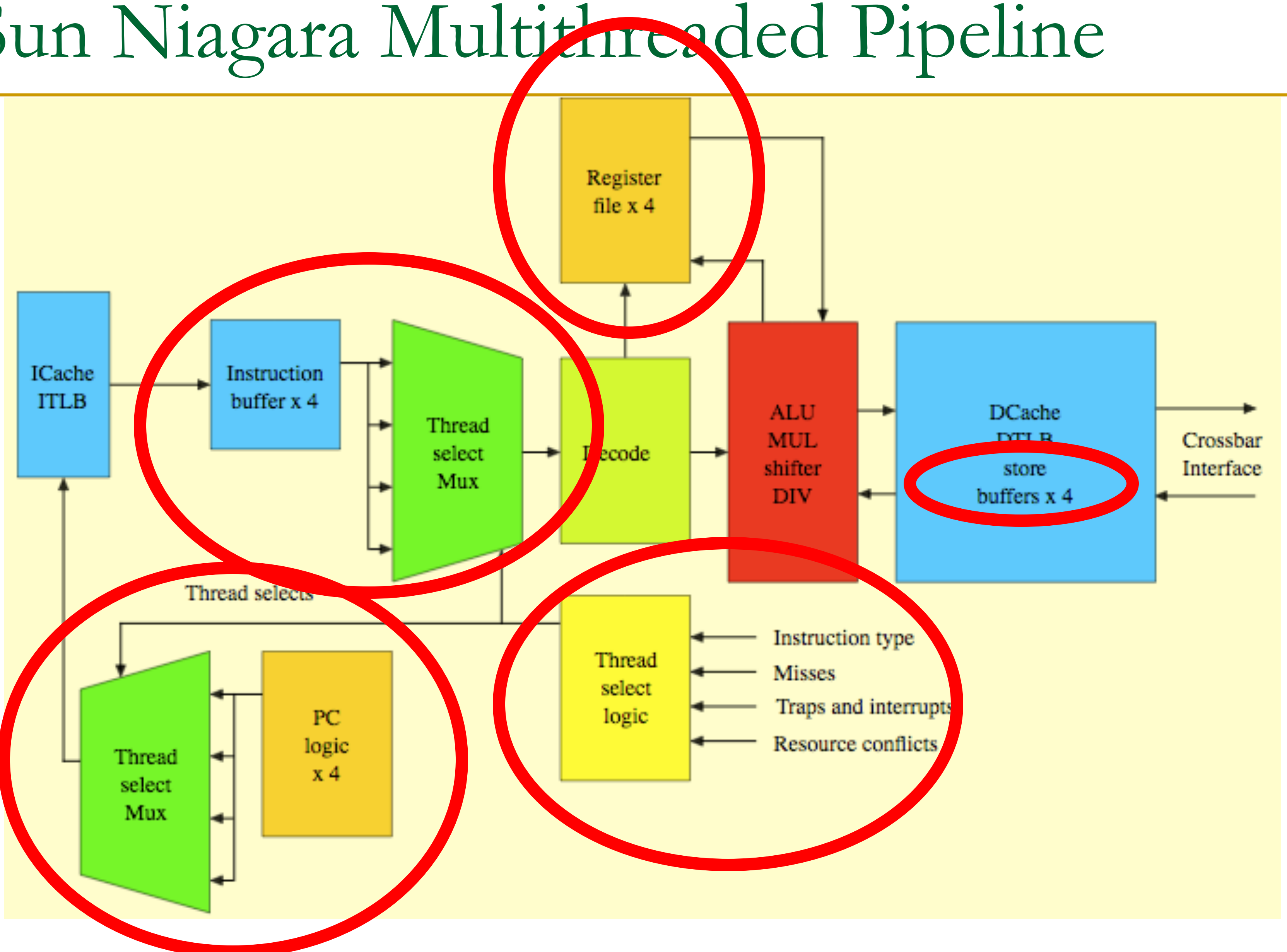


Multithreaded Pipeline Example



Slide credit: Joel Emer

Sun Niagara Multithreaded Pipeline



Kongetira et al., "Niagara: A 32-Way Multithreaded Sparc Processor," IEEE Micro 2005.

Fine-grained Multithreading

- Advantages

  + No need for dependency checking between instructions
  (only one instruction in pipeline from a single thread)

  + No need for branch prediction logic

  + Otherwise-bubble cycles used for executing useful instructions from different threads

  + Improved system throughput, latency tolerance, utilization

- Disadvantages

  - Extra hardware complexity: multiple hardware contexts (PCs, register files, …), thread selection logic

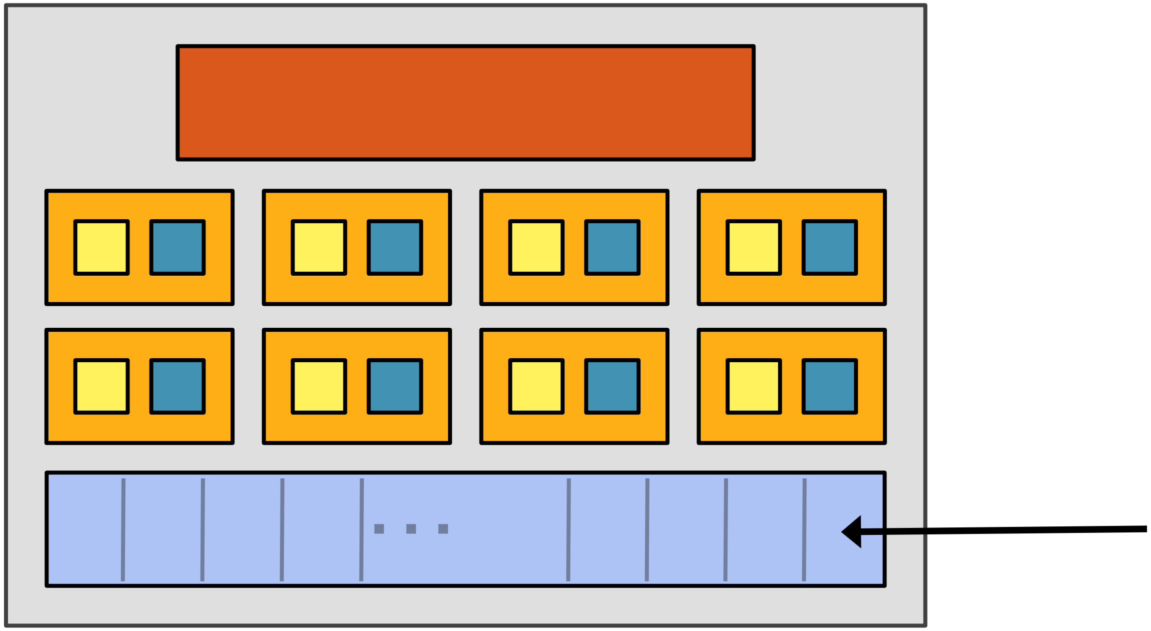
  - Reduced single thread performance (one instruction fetched every N cycles from the same thread)

  - Resource contention between threads in caches and memory

  - Some dependency checking logic *between* threads remains (load/store)

Modern GPUs are FGMT Machines

NVIDIA GeForce GTX 285 "core"



64 KB of storage for thread contexts (registers)

🟧 = data-parallel (SIMD) func. unit, control shared across 8 units

🟨 = multiply-add

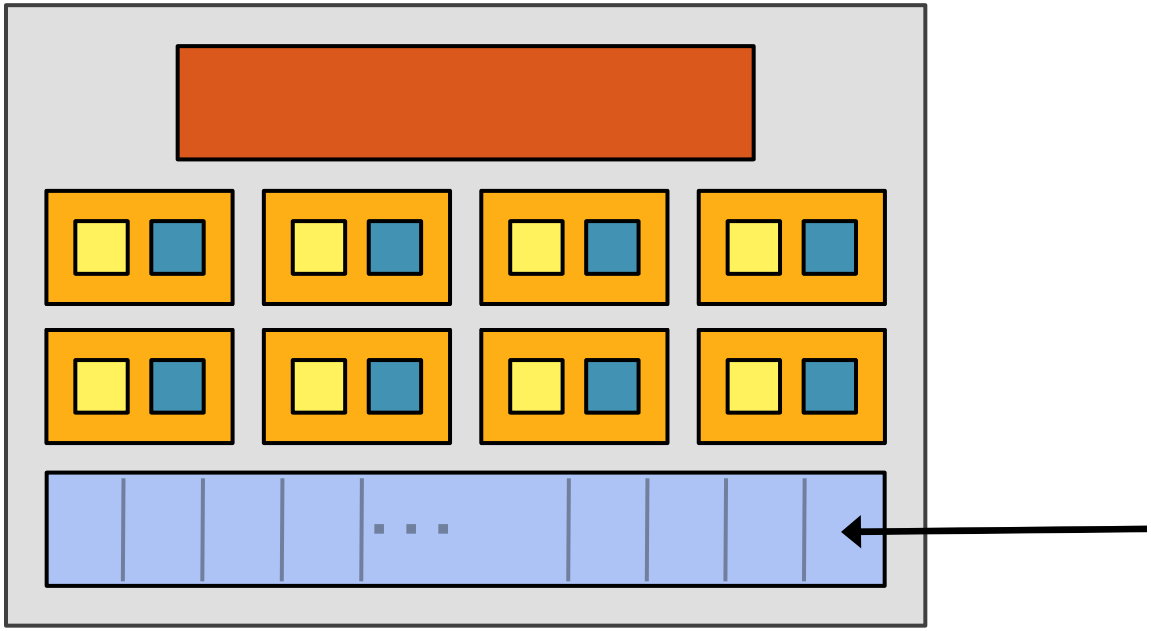
🟦 = multiply

🟥 = instruction stream decode

🟦 = execution context storage

Slide credit: Kayvon Fatahalian

NVIDIA GeForce GTX 285 "core"

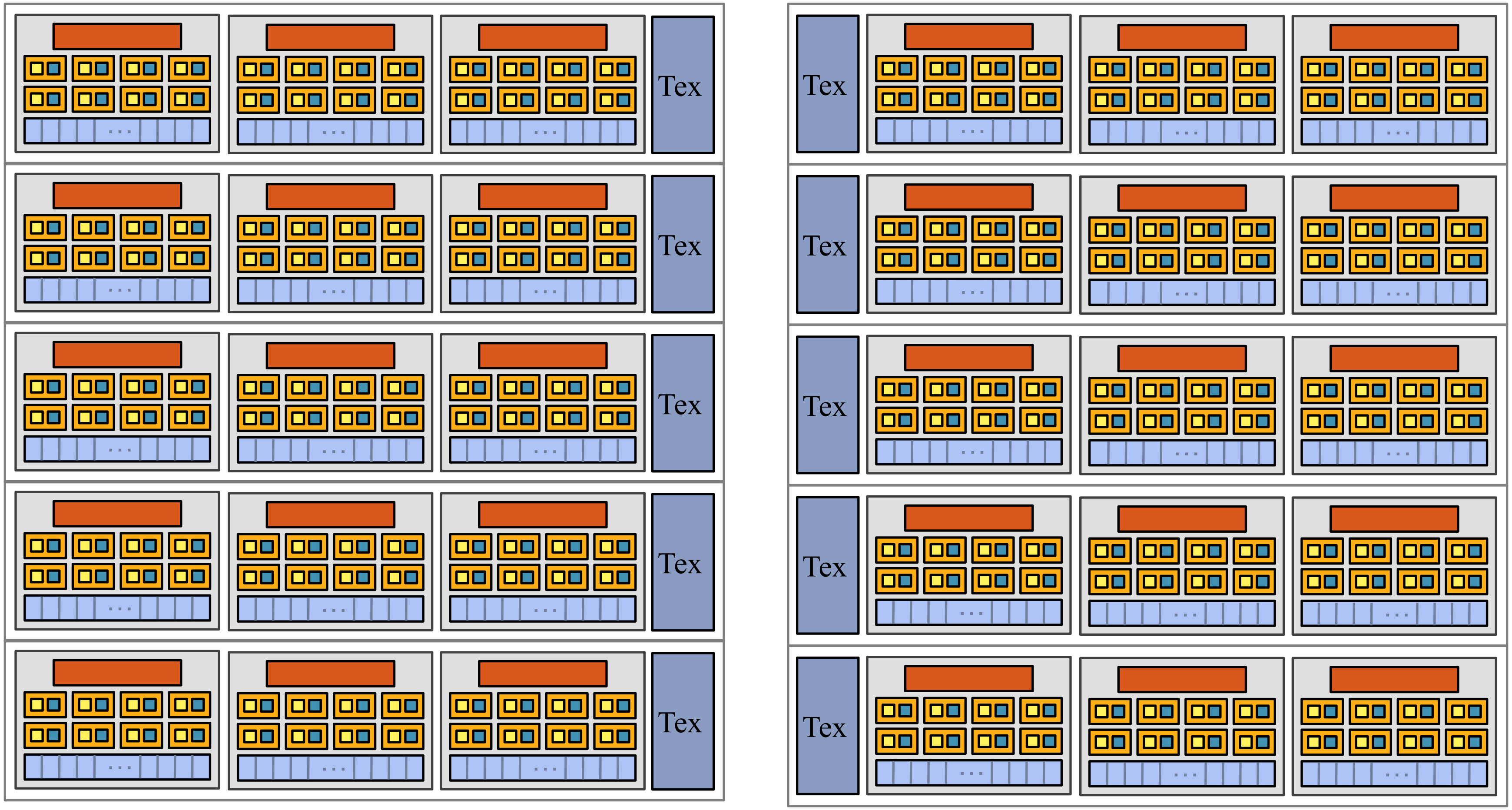


64 KB of storage for thread contexts (registers)

- Groups of 32 threads share instruction stream (each group is a Warp): they execute the same instruction on different data
- **Up to 32 warps are interleaved in an FGMT manner**
- Up to 1024 thread contexts can be stored

Slide credit: Kayvon Fatahalian

NVIDIA GeForce GTX 285

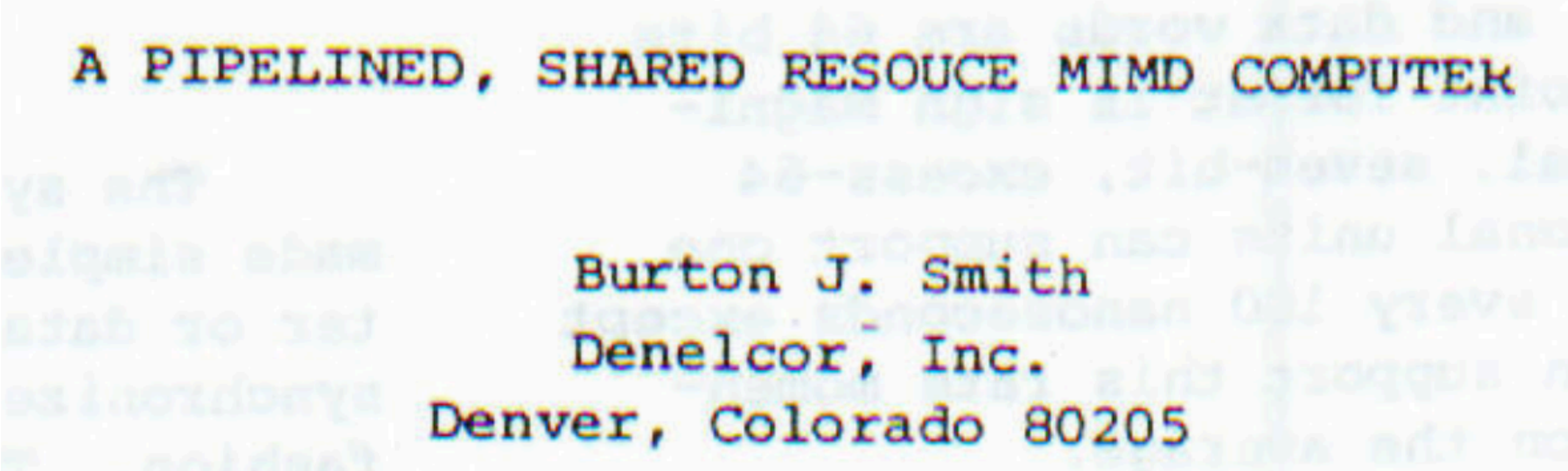


30 cores on the GTX 285: 30,720 threads

Slide credit: Kayvon Fatahalian

End of
Fine-Grained Multithreading

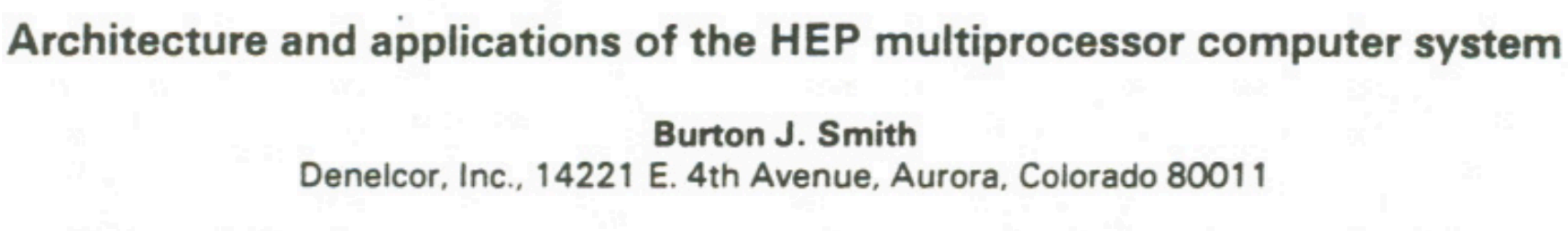
In Memory of Burton Smith



A PIPELINED, SHARED RESOUCE MIMD COMPUTER

Burton J. Smith
Denelcor, Inc.
Denver, Colorado 80205

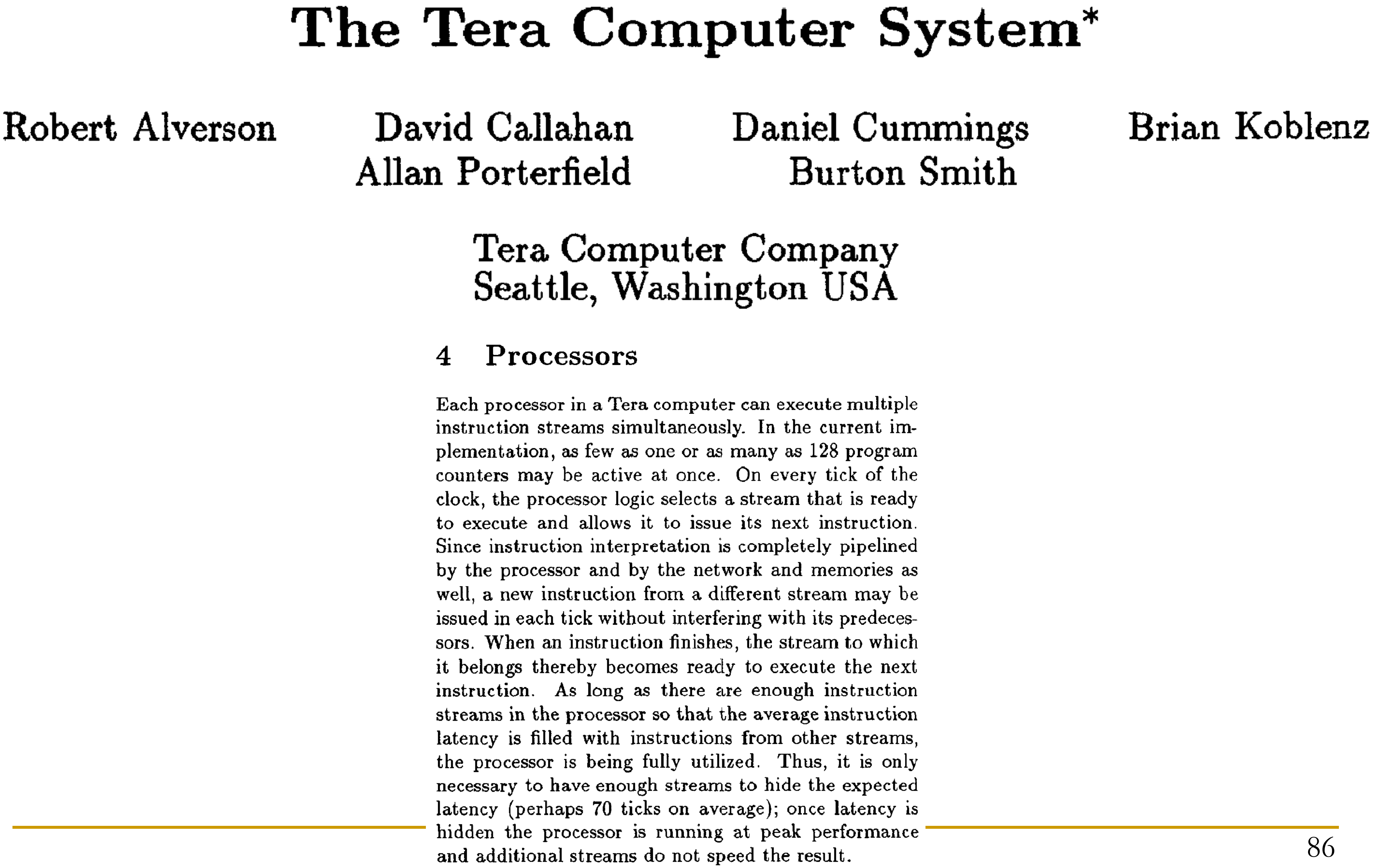
Burton Smith
(1941-2018)



Architecture and applications of the HEP multiprocessor computer system

Burton J. Smith
Denelcor, Inc., 14221 E. 4th Avenue, Aurora, Colorado 80011

In Memory of Burton Smith (II)



The Tera Computer System*

Robert Alverson David Callahan Daniel Cummings Brian Koblenz
Allan Porterfield Burton Smith

Tera Computer Company
Seattle, Washington USA

4 Processors

Each processor in a Tera computer can execute multiple instruction streams simultaneously. In the current implementation, as few as one or as many as 128 program counters may be active at once. On every tick of the clock, the processor logic selects a stream that is ready to execute and allows it to issue its next instruction. Since instruction interpretation is completely pipelined by the processor and by the network and memories as well, a new instruction from a different stream may be issued in each tick without interfering with its predecessors. When an instruction finishes, the stream to which it belongs thereby becomes ready to execute the next instruction. As long as there are enough instruction streams in the processor so that the average instruction latency is filled with instructions from other streams, the processor is being fully utilized. Thus, it is only necessary to have enough streams to hide the expected latency (perhaps 70 ticks on average); once latency is hidden the processor is running at peak performance and additional streams do not speed the result.

Wednesday Keynote (HiPEAC 2015)

Resource Management in PACORA

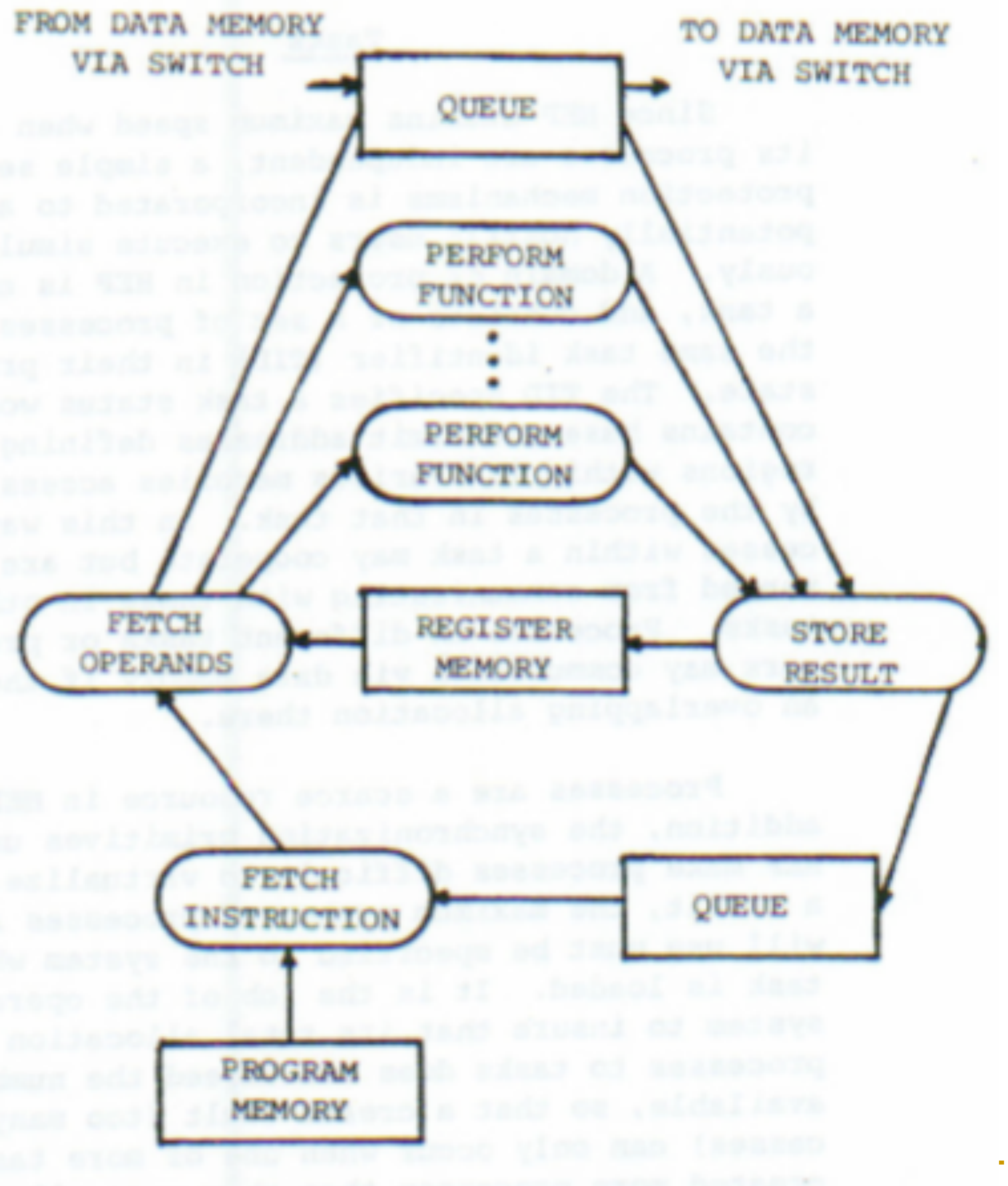
Burton J. Smith
Microsoft

Burton Smith

- Technical Fellow at Microsoft
- Past: Co-founder, chief scientist, chairman of Tera/Cray, Denelcor, Professor at Colorado
- Eckert-Mauchly Award in 1991, Seymour Cray Award, US National Academy of Engineering, AAAS/ACM/IEEE Fellow and many other honors
- Many wide-range contributions spanning architecture, system software, compilers, …, including:
  - Denelcor HEP, Tera MTA
  - fine-grained synchronization, communication, multithreading
  - parallel architectures, resource management, interconnection networks
  - …
- One I would like to share:
  - Smith, "A pipelined, shared resource MIMD computer", ICPP 1978.

Fine-grained Multithreading in HEP

- 128 processes (hardware contexts)
- Cycle time: 100ns
- 8 stages → 800 ns to complete an instruction
    - assuming no memory access
- No control and data dependency checking



Wednesday Keynote (HiPEAC 2015)

Resource Management in PACORA

Burton J. Smith
Microsoft