18-447
Computer Architecture
Lecture 13: Out-of-Order Execution and Data Flow

Prof. Onur Mutlu
Carnegie Mellon University
Spring 2015, 2/16/2015

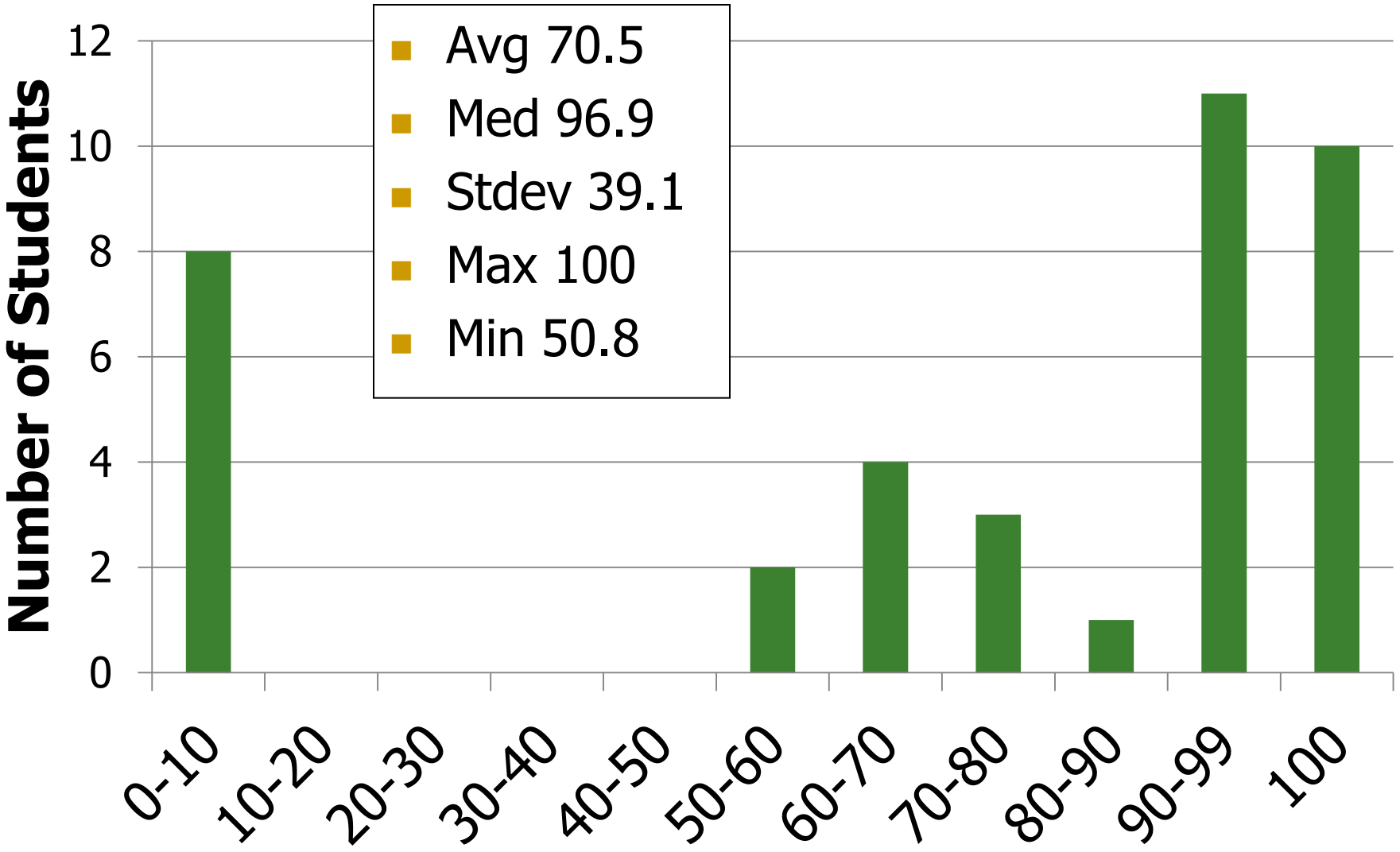
Agenda for Today & Next Few Lectures

- Single-cycle Microarchitectures
- Multi-cycle and Microprogrammed Microarchitectures
- Pipelining
- Issues in Pipelining: Control & Data Dependence Handling, State Maintenance and Recovery, …
- Out-of-Order Execution
- Issues in OoO Execution: Load-Store Handling, …
- Alternative Approaches to Instruction Level Parallelism

Reminder: Announcements

- Lab 3 due this Friday (Feb 20)
  - Pipelined MIPS
  - Competition for high performance
    - You can optimize both cycle time and CPI
    - Document and clearly describe what you do during check-off
- Homework 3 due Feb 25
  - A lot of questions that enable you to learn the concepts via hands-on exercise
  - Remember this is all for your benefit (to learn and prepare for exams)
    - HWs have very little contribution to overall grade
    - Solutions to almost all questions are online anyway

Lab 2 Grade Distribution



Lab 2 Extra Credits

- Complete and correct:
  - Terence An
  - Jared Choi
- Almost correct:
  - Pete Ehrett
  - Xiaofan Li
  - Amanda Marano
  - Ashish Shrestha
- Almost-1 correct:
  - Sohil Shah

Readings Specifically 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

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

Readings for Next Lecture

- SIMD Processing
- Basic GPU Architecture
- Other execution models: VLIW, DAE, Systolic Arrays

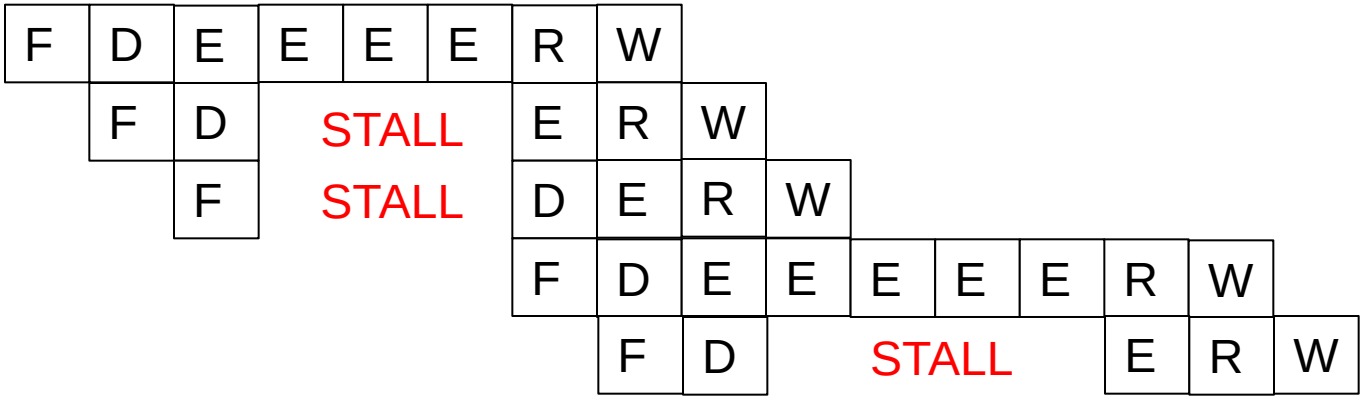
- Lindholm et al., "NVIDIA Tesla: A Unified Graphics and Computing Architecture," IEEE Micro 2008.
- Fatahalian and Houston, "A Closer Look at GPUs," CACM 2008.

Recap of Last Lecture

- Maintaining Speculative Memory State (Ld/St Ordering)
- Out of Order Execution (Dynamic Scheduling)
  - Link Dependent Instructions: Renaming
  - Buffer Instructions: Reservation Stations
  - Track Readiness of Source Values: Tag (and Value) Broadcast
  - Schedule/Dispatch: Wakeup and Select
- Tomasulo's Algorithm
- OoO Execution Exercise with Code Example: Cycle by Cycle
- OoO Execution with Precise Exceptions
- Questions on OoO Implementation
  - Where data is stored? Single physical register file vs. reservation stations
  - Critical path, renaming IDs, …
- OoO Execution as Restricted Data Flow
- Reverse Engineering the Data Flow Graph

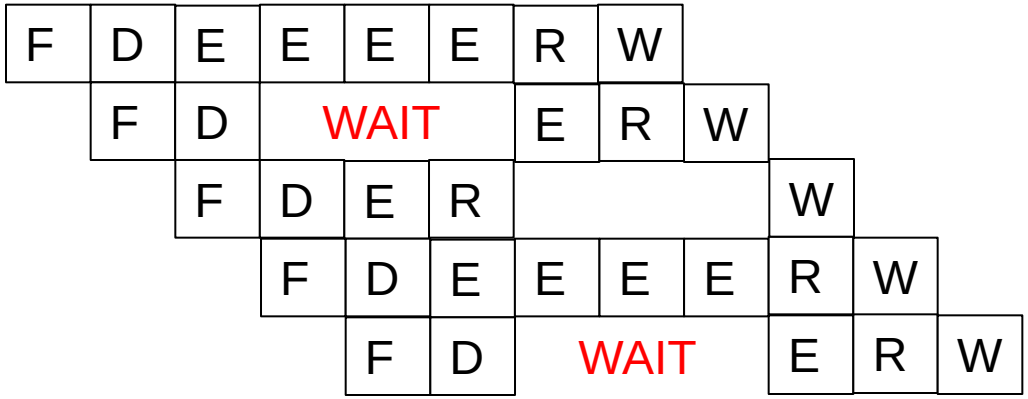
Review: In-order vs. Out-of-order Dispatch

- In order dispatch + precise exceptions:



IMUL R3 ← R1, R2
ADD R3 ← R3, R1
ADD R1 ← R6, R7
IMUL R5 ← R6, R8
ADD R7 ← R3, R5

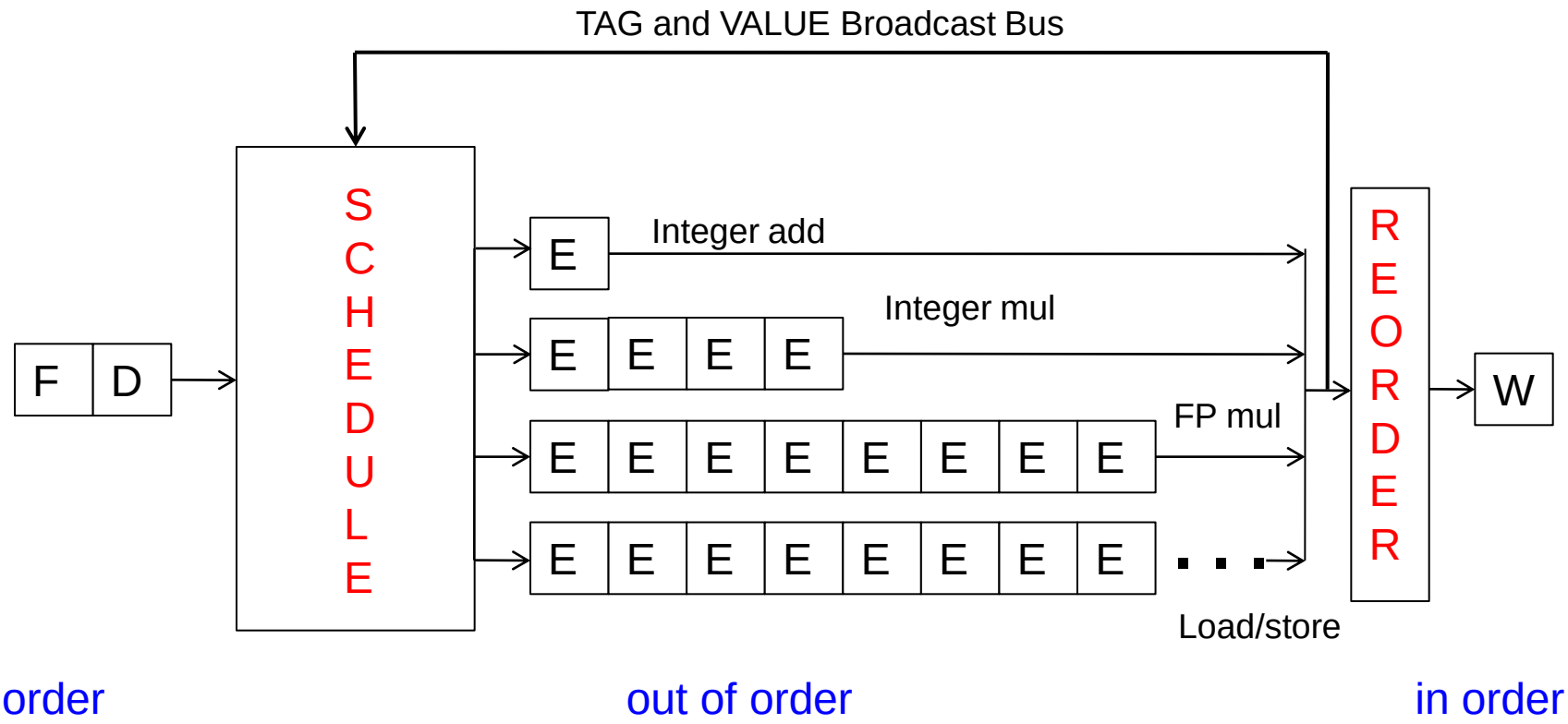
- Out-of-order dispatch + precise exceptions:



This slide is actually correct

- 16 vs. 12 cycles

Review: Out-of-Order Execution with Precise Exceptions



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

Review: 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

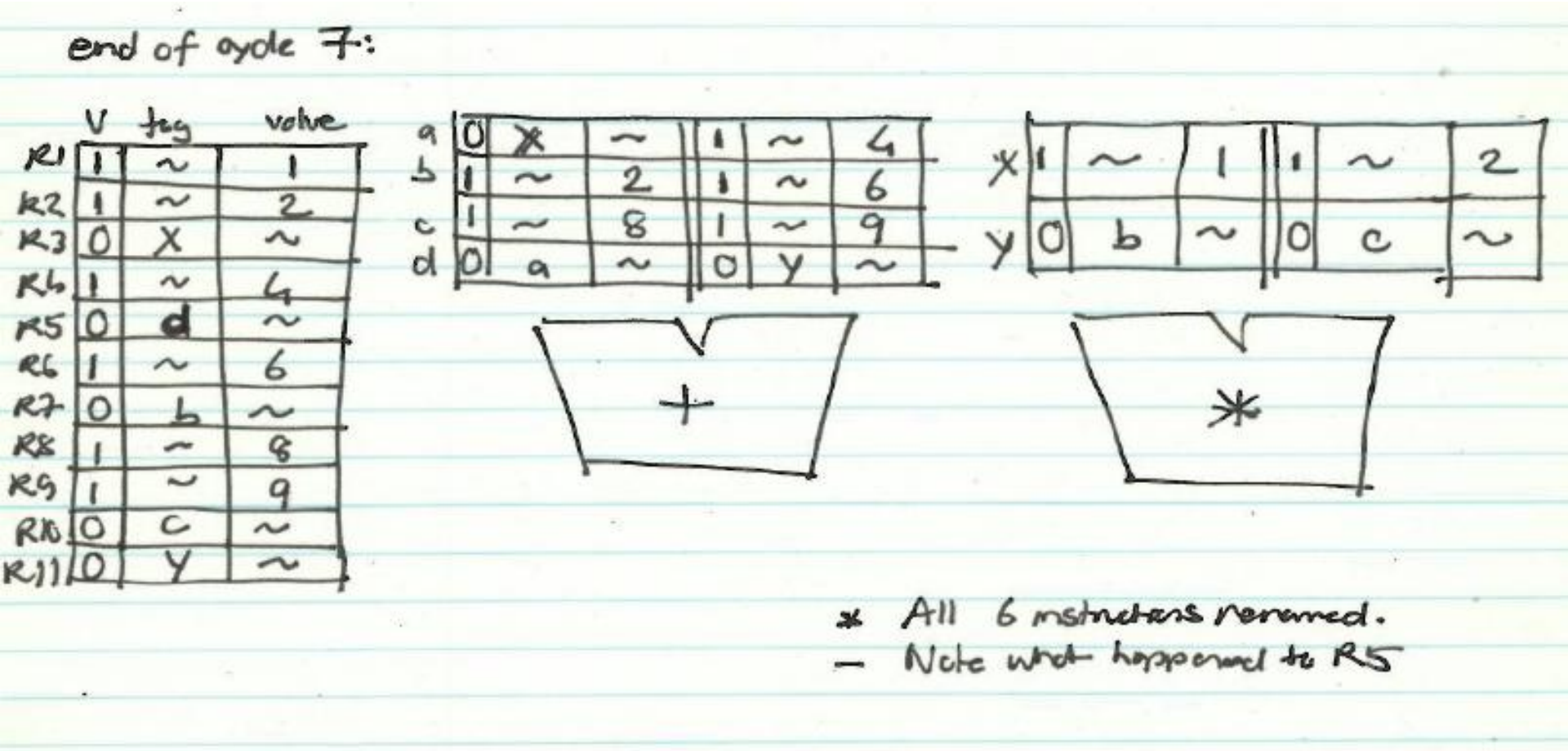
Review: Summary of OOO Execution Concepts

- 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

Review: 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

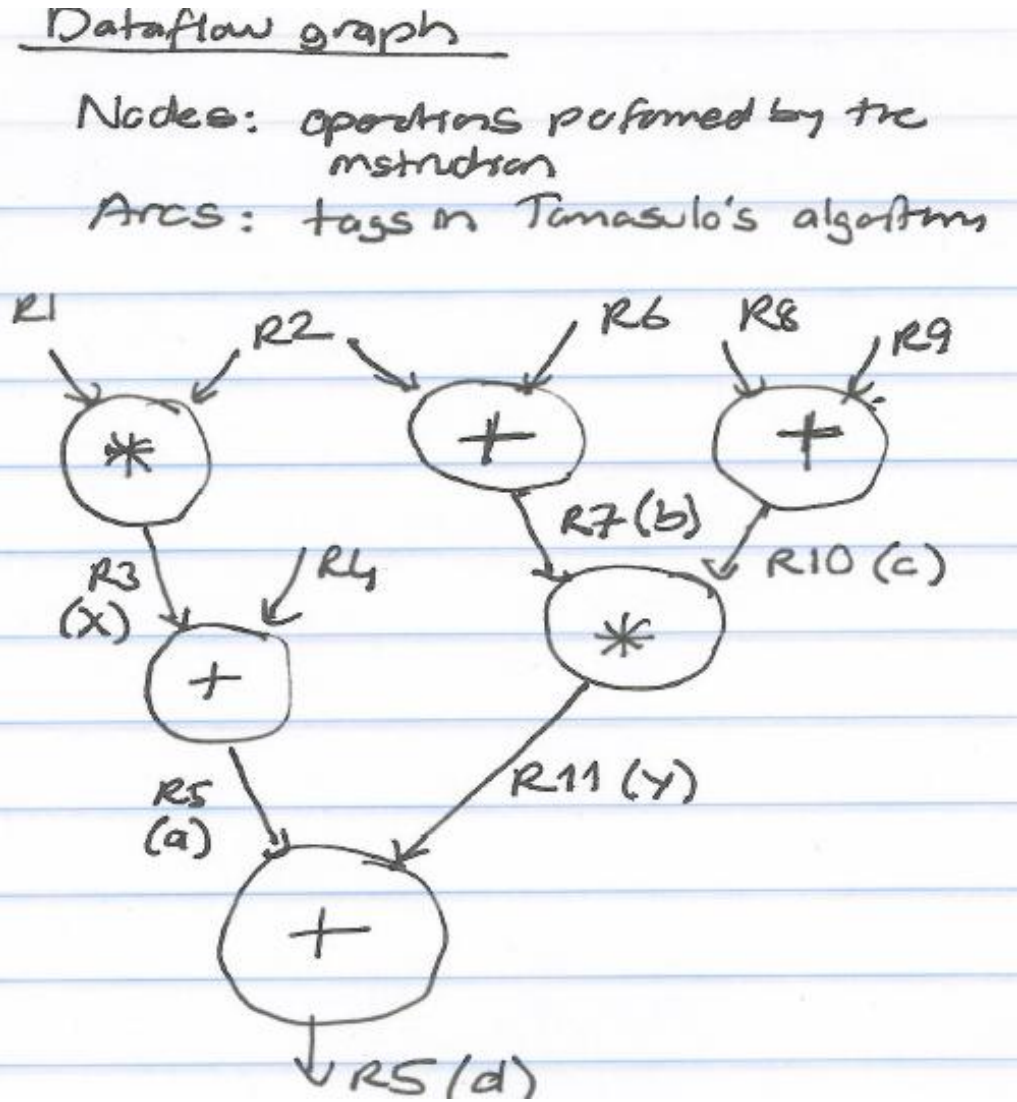
Review: State of RAT and RS in Cycle 7



All our in-class drawings are at:
http://www.ece.cmu.edu/~ece447/s15/lib/exe/fetch.php?media=447_tomasulo.pdf

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)
```



Restricted Data Flow

- An out-of-order machine is a "restricted data flow" machine
  - Dataflow-based execution is restricted to the microarchitecture level
  - ISA is still based on von Neumann model (sequential execution)

- Remember the data flow model (at the ISA level):
  - Dataflow model: An instruction is fetched and executed in data flow order
  - i.e., when its operands are ready
  - i.e., there is no instruction pointer
  - Instruction ordering specified by data flow dependence
    - Each instruction specifies "who" should receive the result
    - An instruction can "fire" whenever all operands are received

Review: 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?

Questions to Ponder

- Why is OoO execution beneficial?
  - What if all operations take 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

Registers versus Memory, Revisited

- So far, we considered register based value communication between instructions
- 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 dependent on all previous stores

  + No need for recovery

  -- Too conservative: delays independent loads unnecessarily

- Option 2: Assume load 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

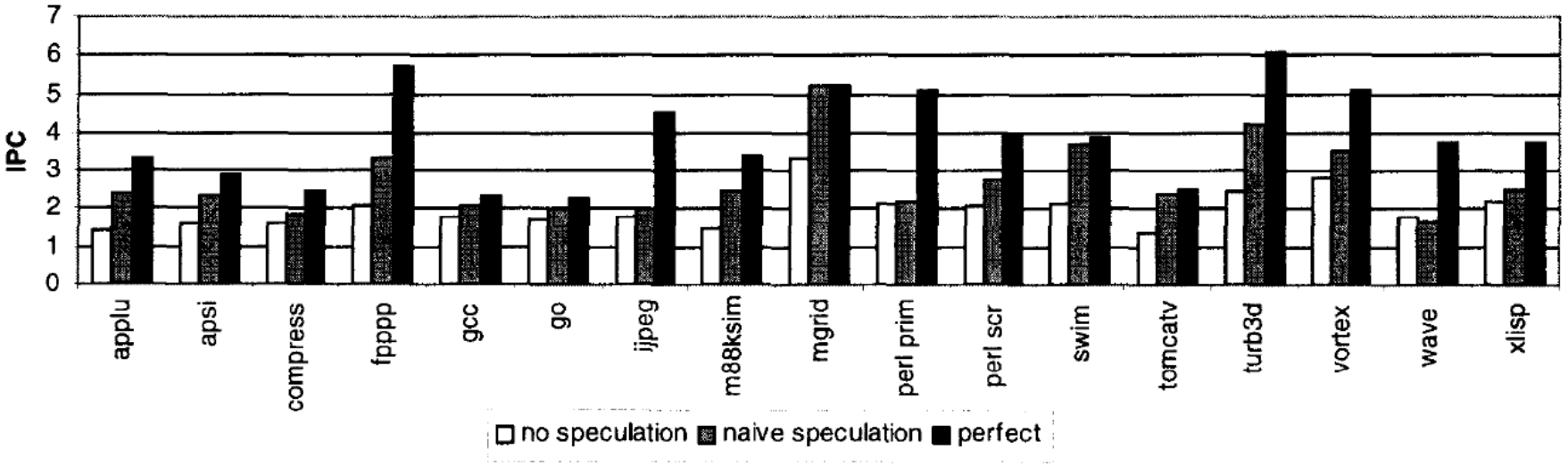
  + More accurate. Load store dependencies persist over time

  -- 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?

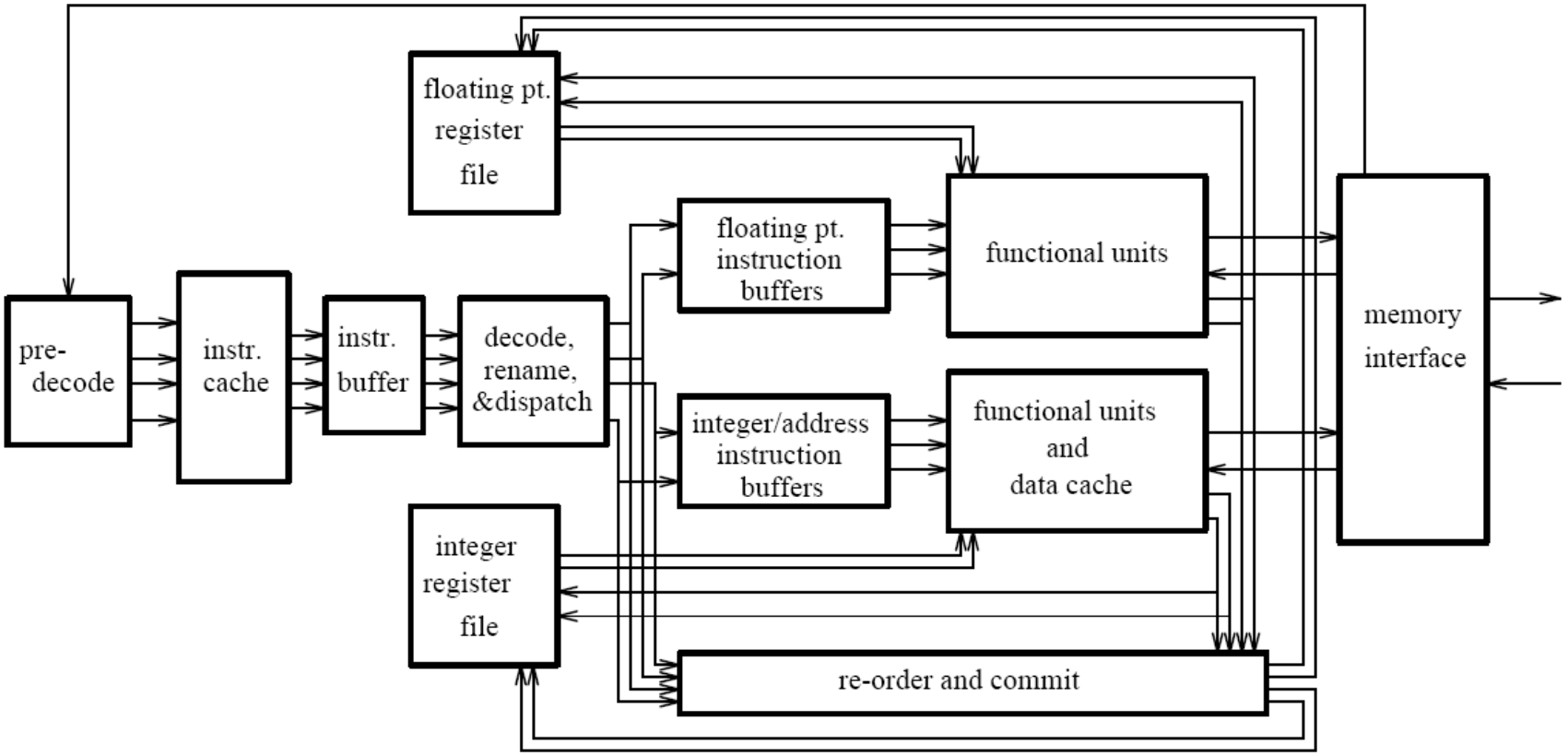
Food for Thought for You

- Many other design choices
- Should reservation stations be centralized or distributed across functional units?
  - 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?
- Exactly when does an instruction broadcast its tag?
- …

More Food for Thought for You

- How can you implement branch prediction in an out-of-order execution machine?
  - Think about branch history register and PHT updates
  - Think about recovery from mispredictions
    - How to do this fast?
- How can you combine superscalar execution with out-of-order execution?
  - These are different concepts
  - Concurrent renaming of instructions
  - Concurrent broadcast of tags
- How can you combine superscalar + out-of-order + branch prediction?

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

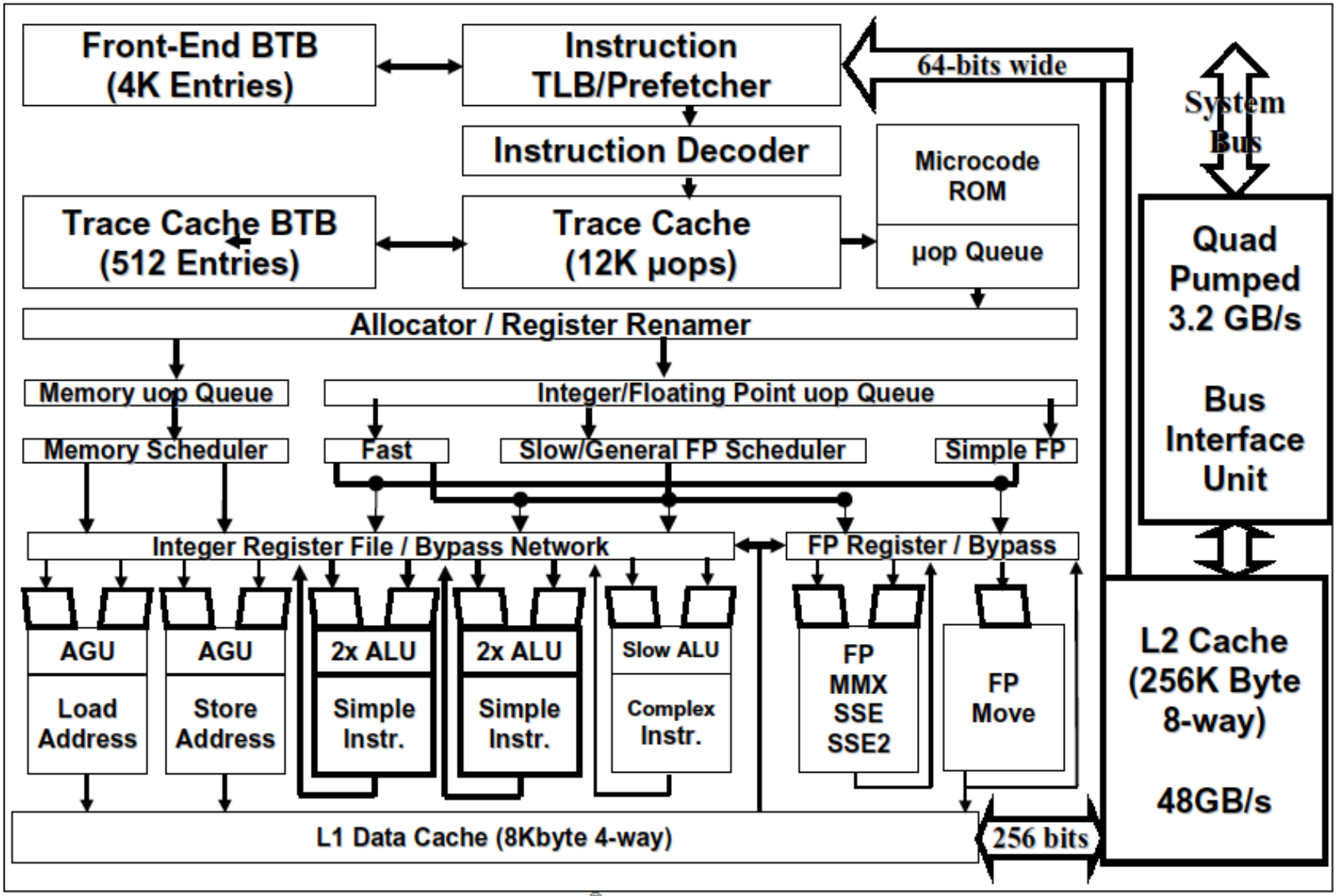
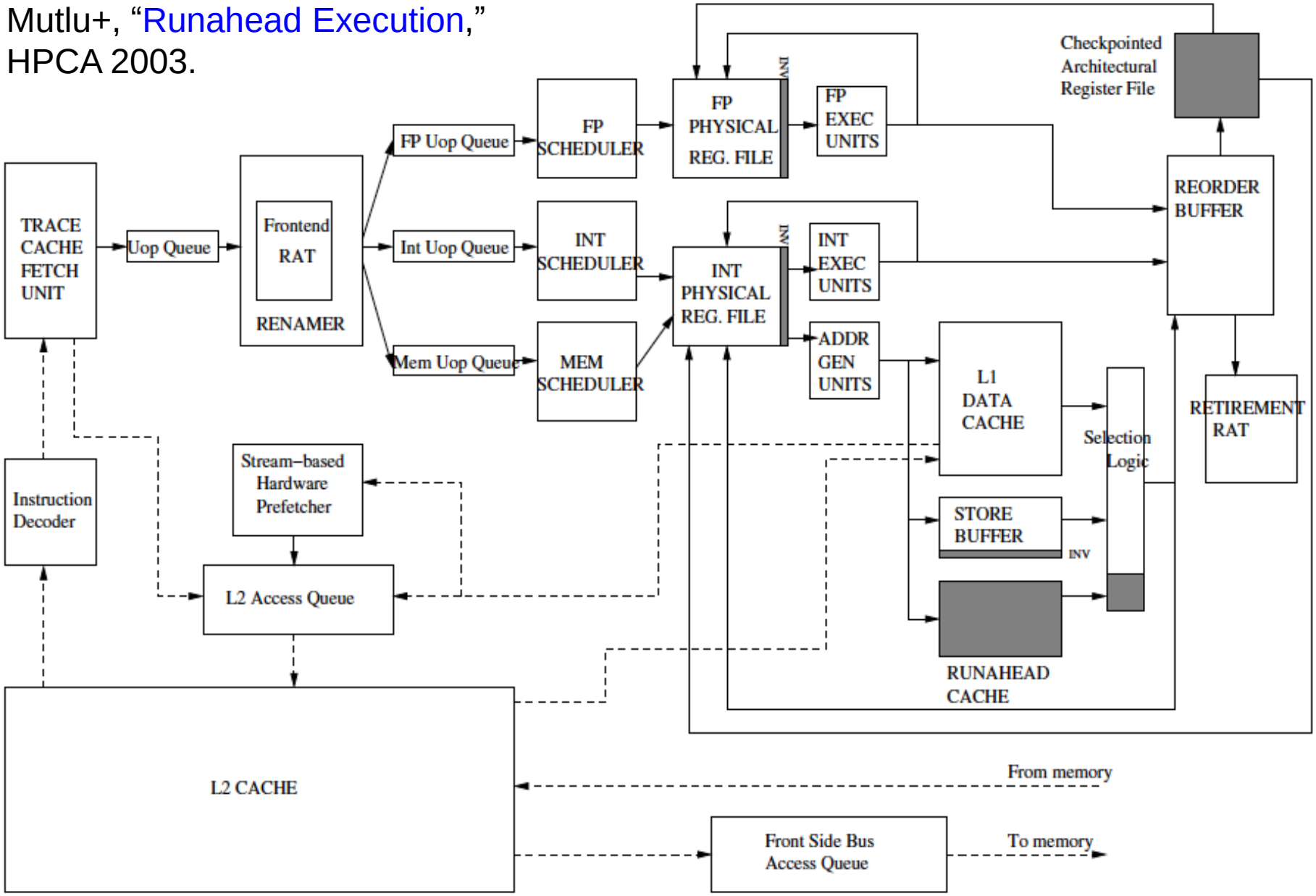


Figure 4: Pentium® 4 processor microarchitecture

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

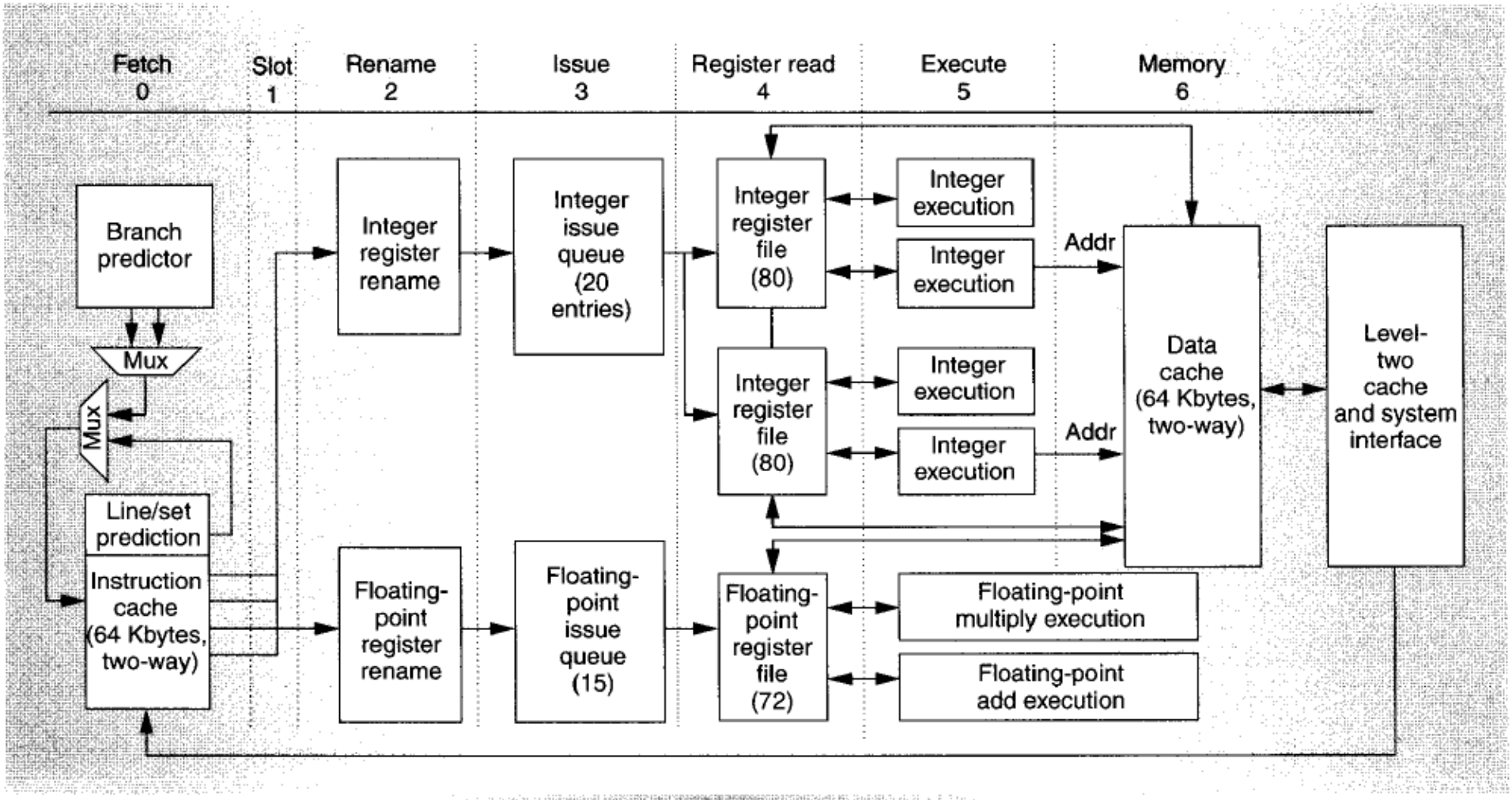
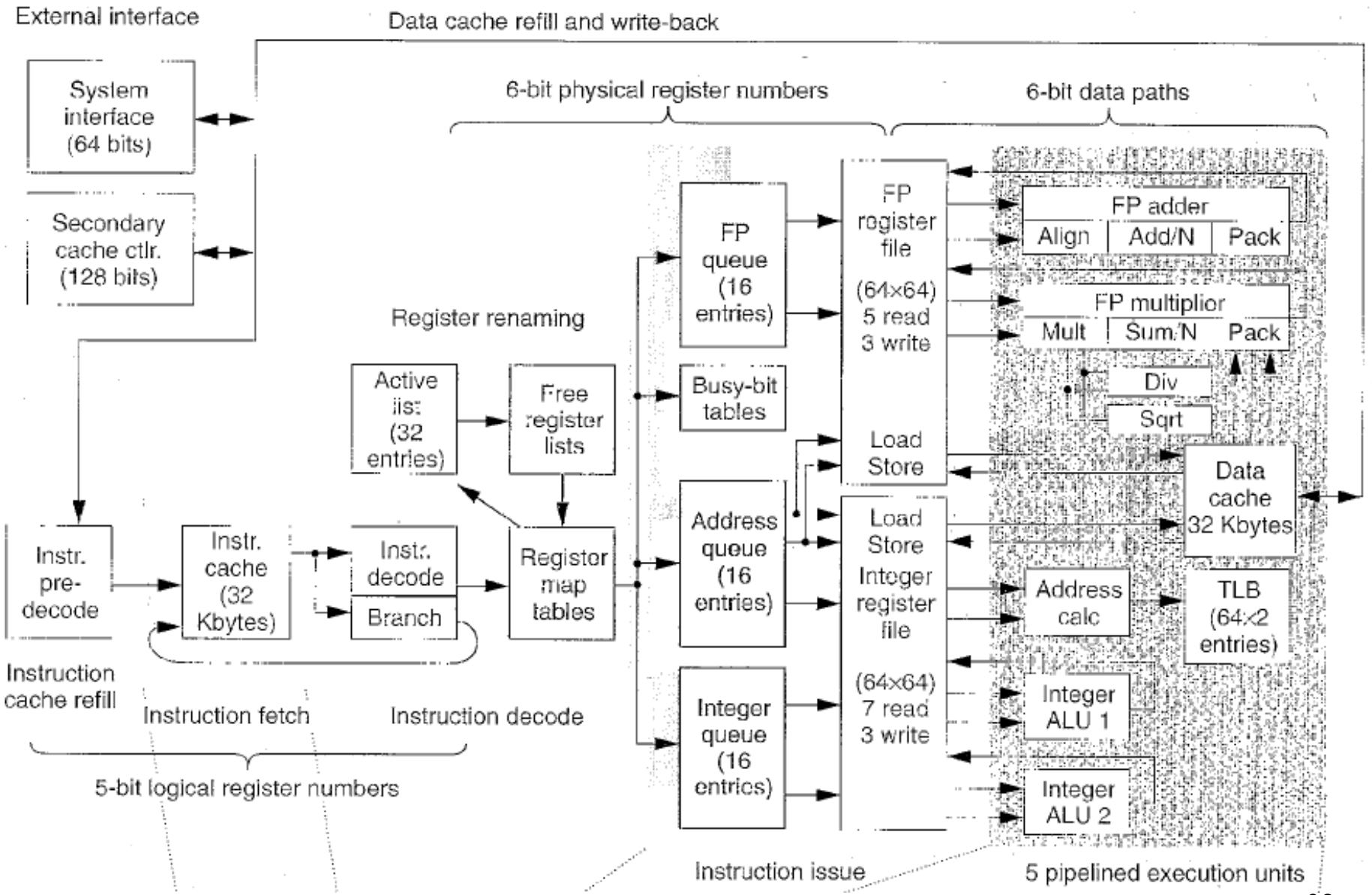


Figure 2. Stages of the Alpha 21264 instruction pipeline.

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

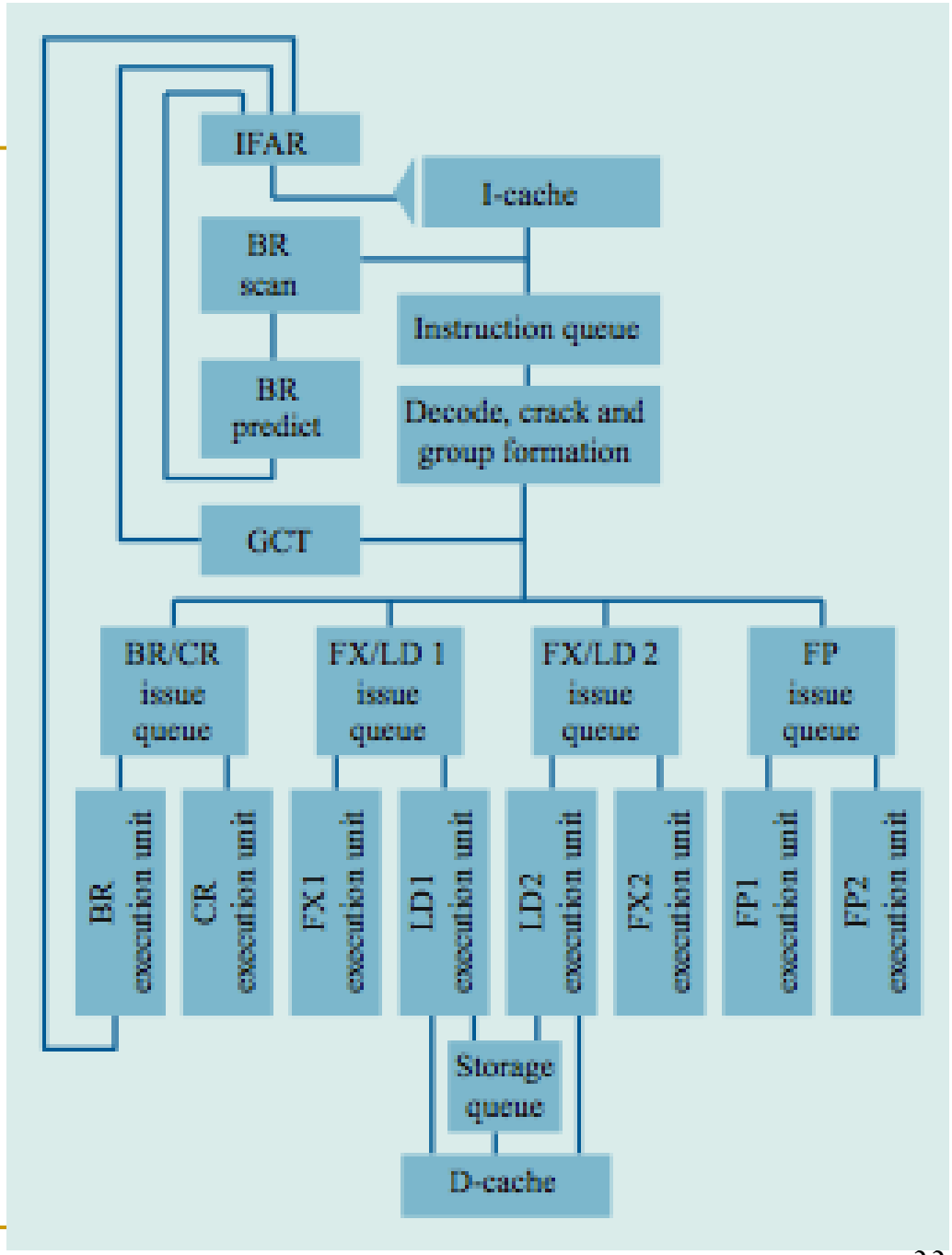
MIPS R10000



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.

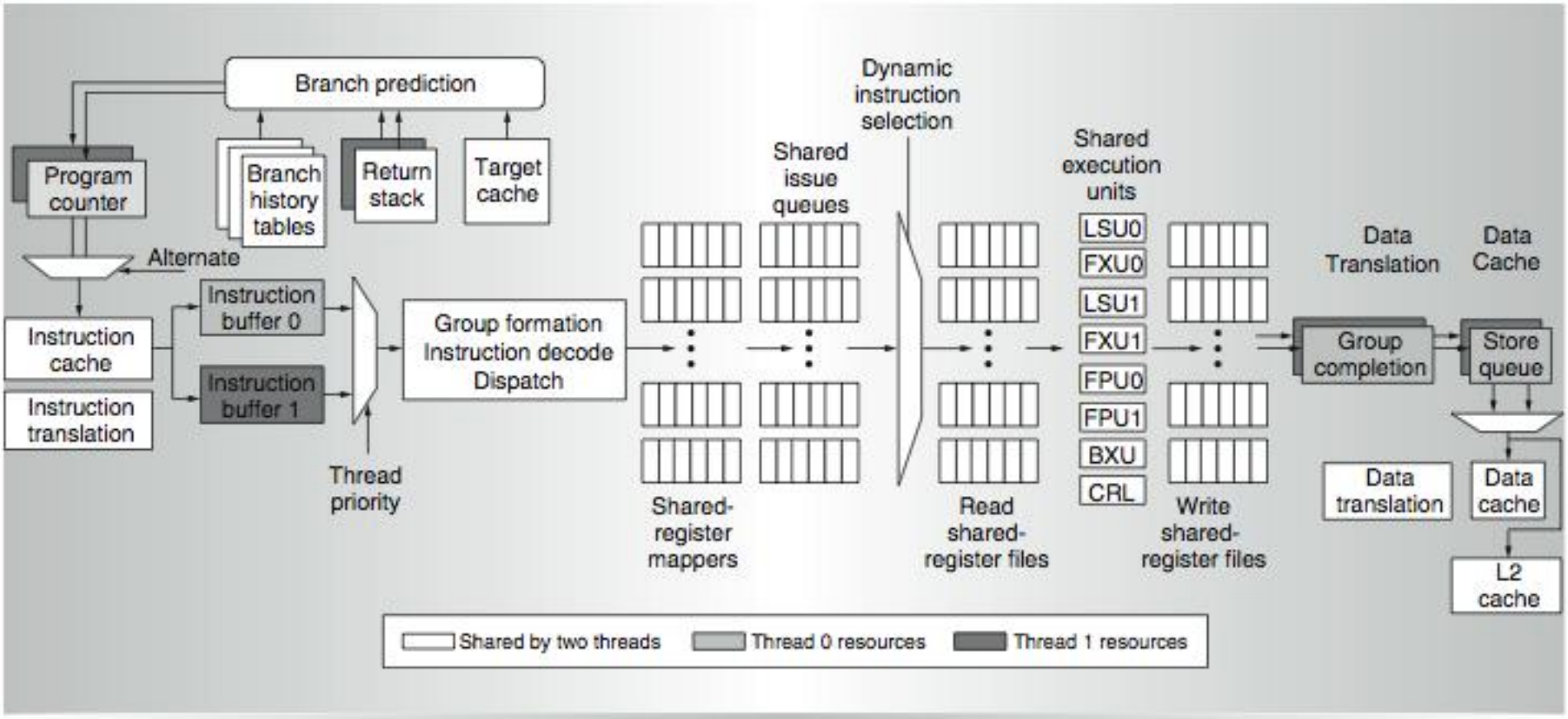


Figure 4. Power5 instruction data flow (BXU = branch execution unit and CRL = condition register logical execution unit).

Recommended Readings

- Out-of-order execution processor designs
- Kessler, "The Alpha 21264 Microprocessor," IEEE Micro, March-April 1999.
- Boggs et al., "The Microarchitecture of the Pentium 4 Processor," Intel Technology Journal, 2001.
- Yeager, "The MIPS R10000 Superscalar Microprocessor," IEEE Micro, April 1996
- Tendler et al., "POWER4 system microarchitecture," IBM Journal of Research and Development, January 2002.

And More Readings…

- Stark et al., "On Pipelining Dynamic Scheduling Logic," MICRO 2000.
- Brown et al., "Select-free Instruction Scheduling Logic," MICRO 2001.
- Palacharla et al., "Complexity-effective Superscalar Processors," ISCA 1997.

Other Approaches to Concurrency (or Instruction Level Parallelism)

Approaches to (Instruction-Level) Concurrency

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

Data Flow: Exploiting Irregular Parallelism

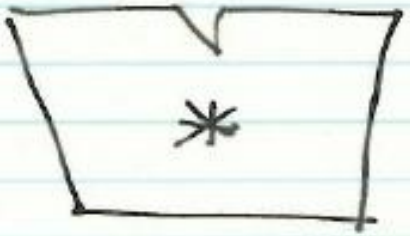
Remember: State of RAT and RS in Cycle 7

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

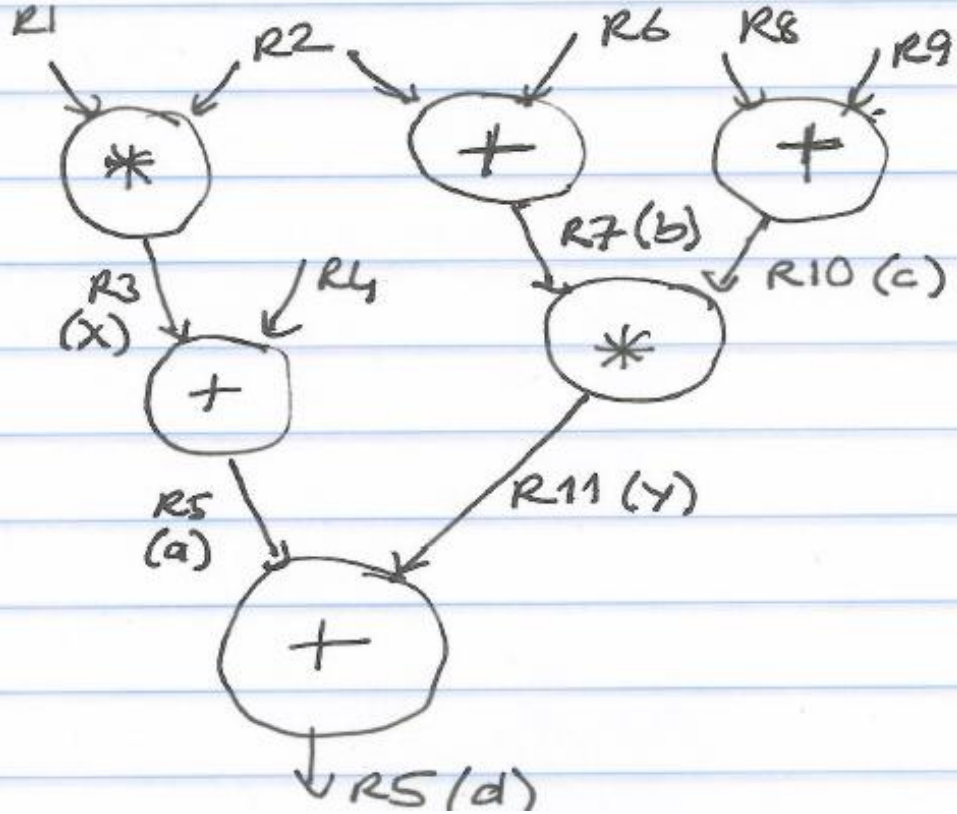
Remember: Dataflow Graph

```
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

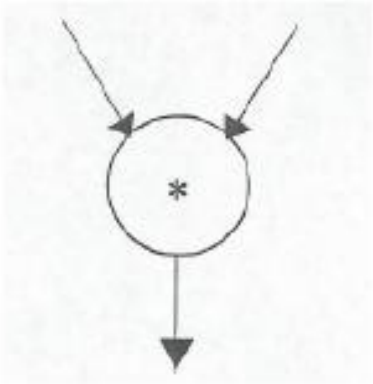
Arcs: tags in Tomasulo's algorithm

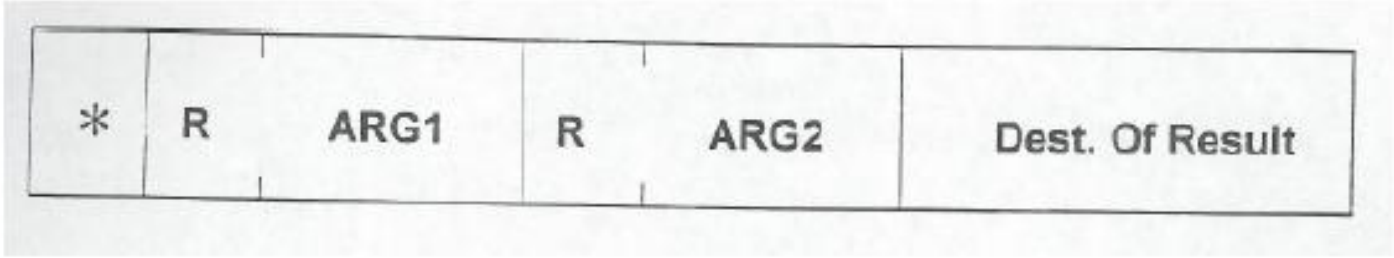


Review: More on Data Flow

- In a data flow machine, a program consists of data flow nodes
  - A data flow node fires (fetched and executed) when all it inputs are ready
    - i.e. when all inputs have tokens

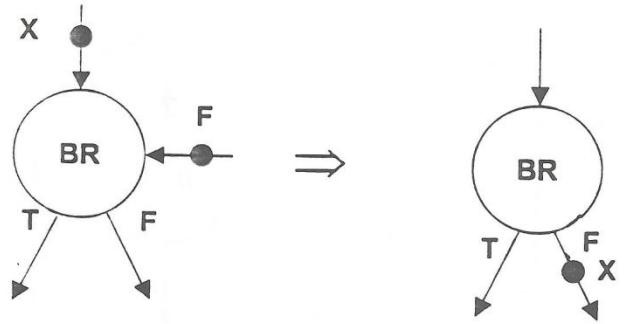
- Data flow node and its ISA representation



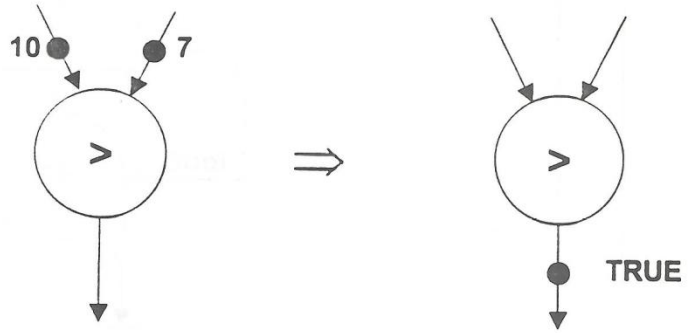


Data Flow Nodes

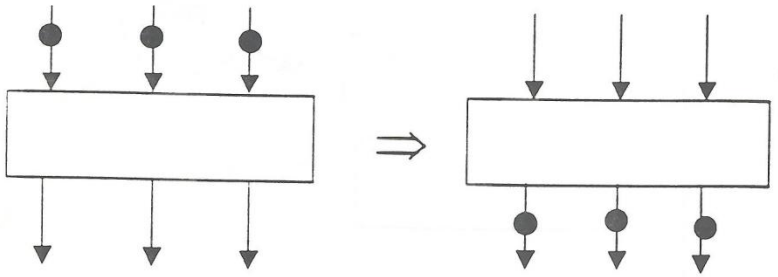
*Conditional



*Relational

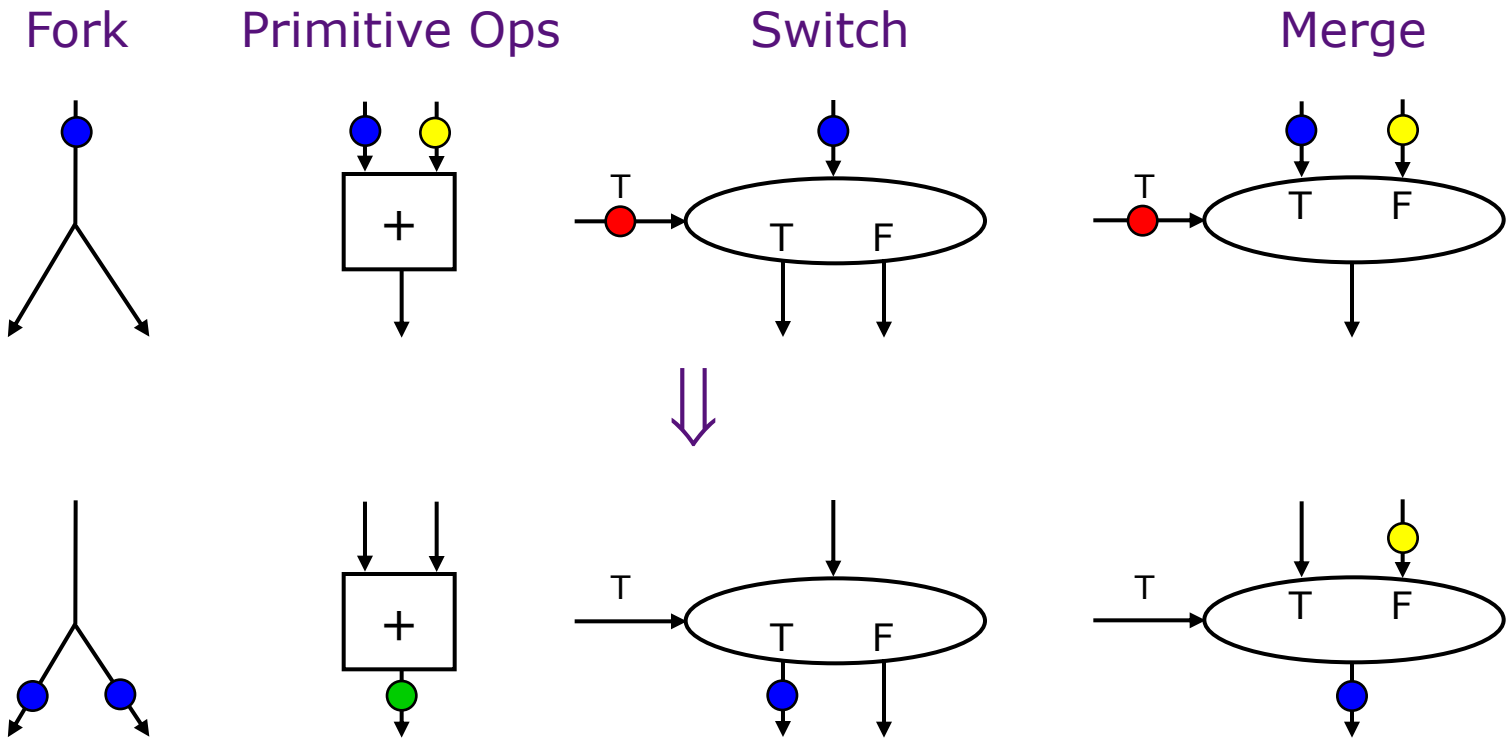


*Barrier Synch



Dataflow Nodes (II)

- A small set of dataflow operators can be used to define a general programming language



Dataflow Graphs

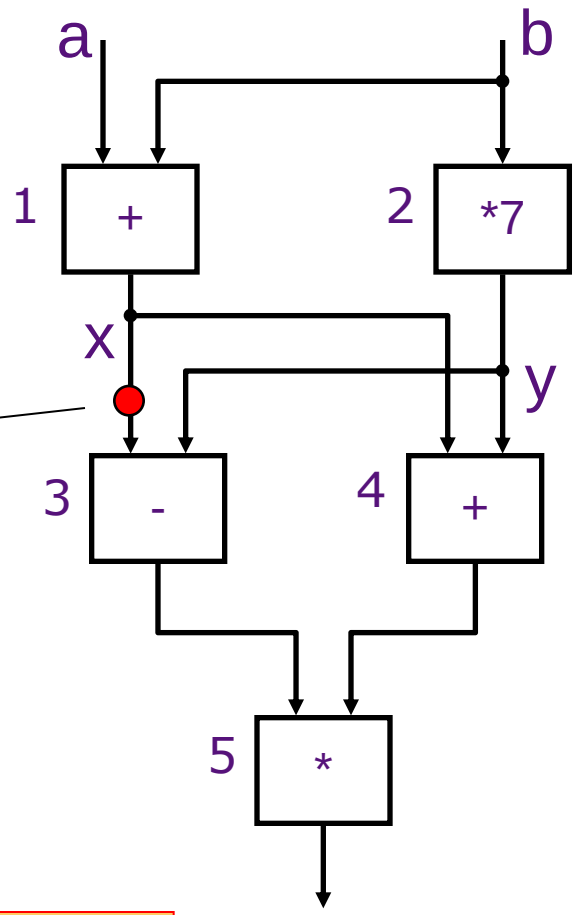
```
{x = a + b;
 y = b * 7
in
   (x-y) * (x+y)}
```

- Values in dataflow graphs are represented as tokens

token < ip , p , v >

instruction ptr — port — data

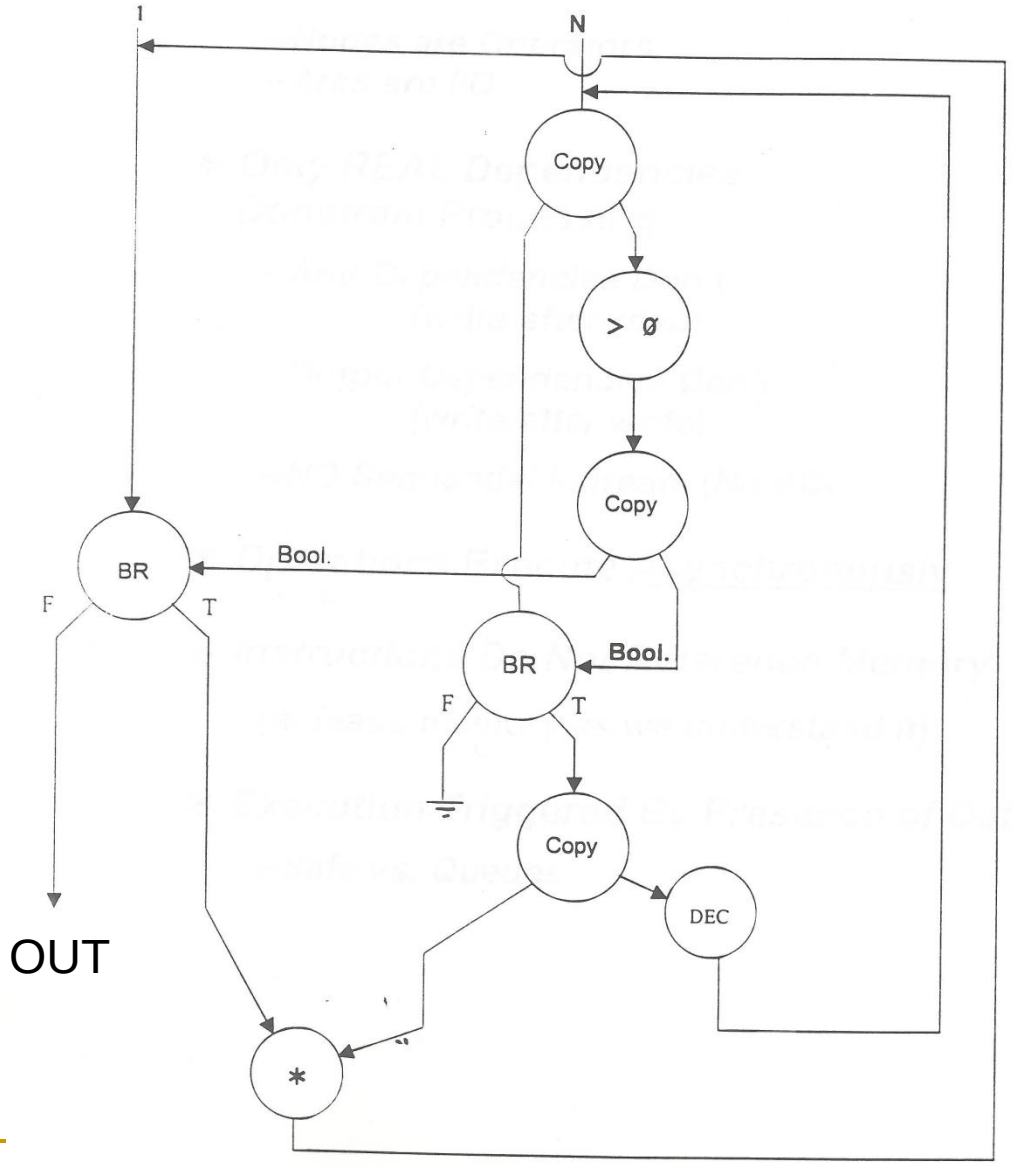
ip = 3
p = L



- An operator executes when all its input tokens are present; copies of the result token are distributed to the destination operators

no separate control flow

Example Data Flow Program



Control Flow vs. Data Flow

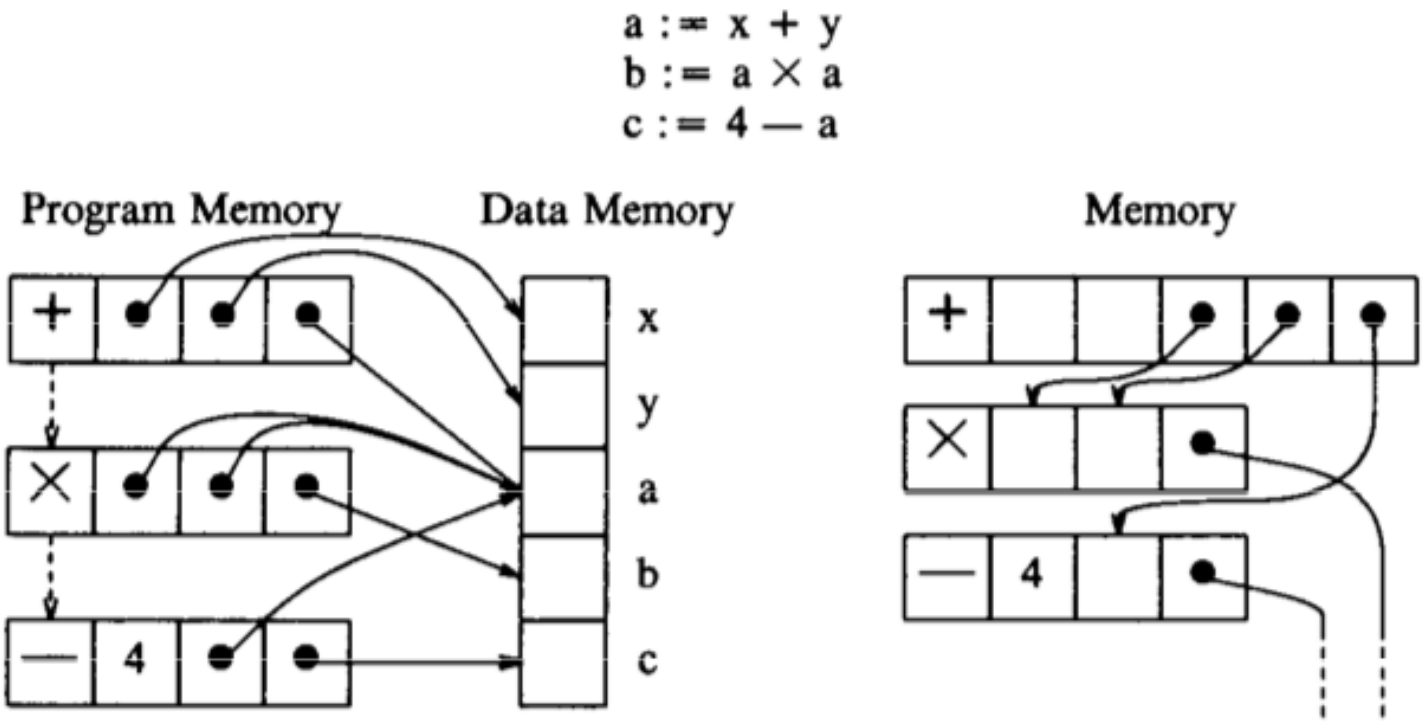


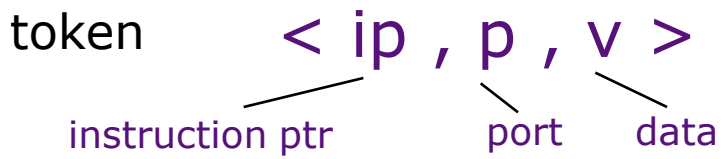
Figure 2. A comparison of control flow and dataflow programs. On the left a control flow program for a computer with memory-to-memory instructions. The arcs point to the locations of data that are to be used or created. Control flow arcs are indicated with dashed arrows; usually most of them are implicit. In the equivalent dataflow program on the right only one memory is involved. Each instruction contains pointers to all instructions that consume its results.

Data Flow Characteristics

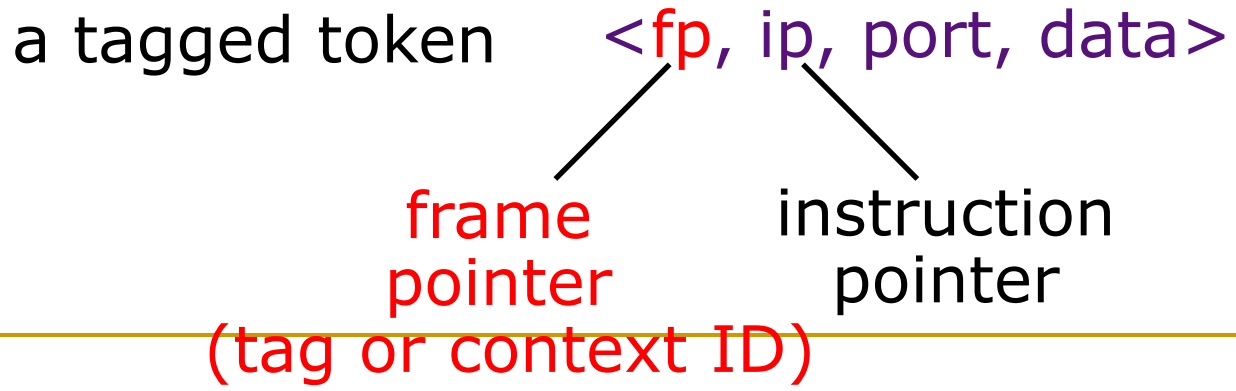
- Data-driven execution of instruction-level graphical code
  - Nodes are operators
  - Arcs are data (I/O)
  - As opposed to control-driven execution
- Only real dependencies constrain processing
- No sequential instruction stream
  - No program counter
- Execution triggered by the presence/readiness of data
- Operations execute asynchronously

What About Loops and Function Calls?

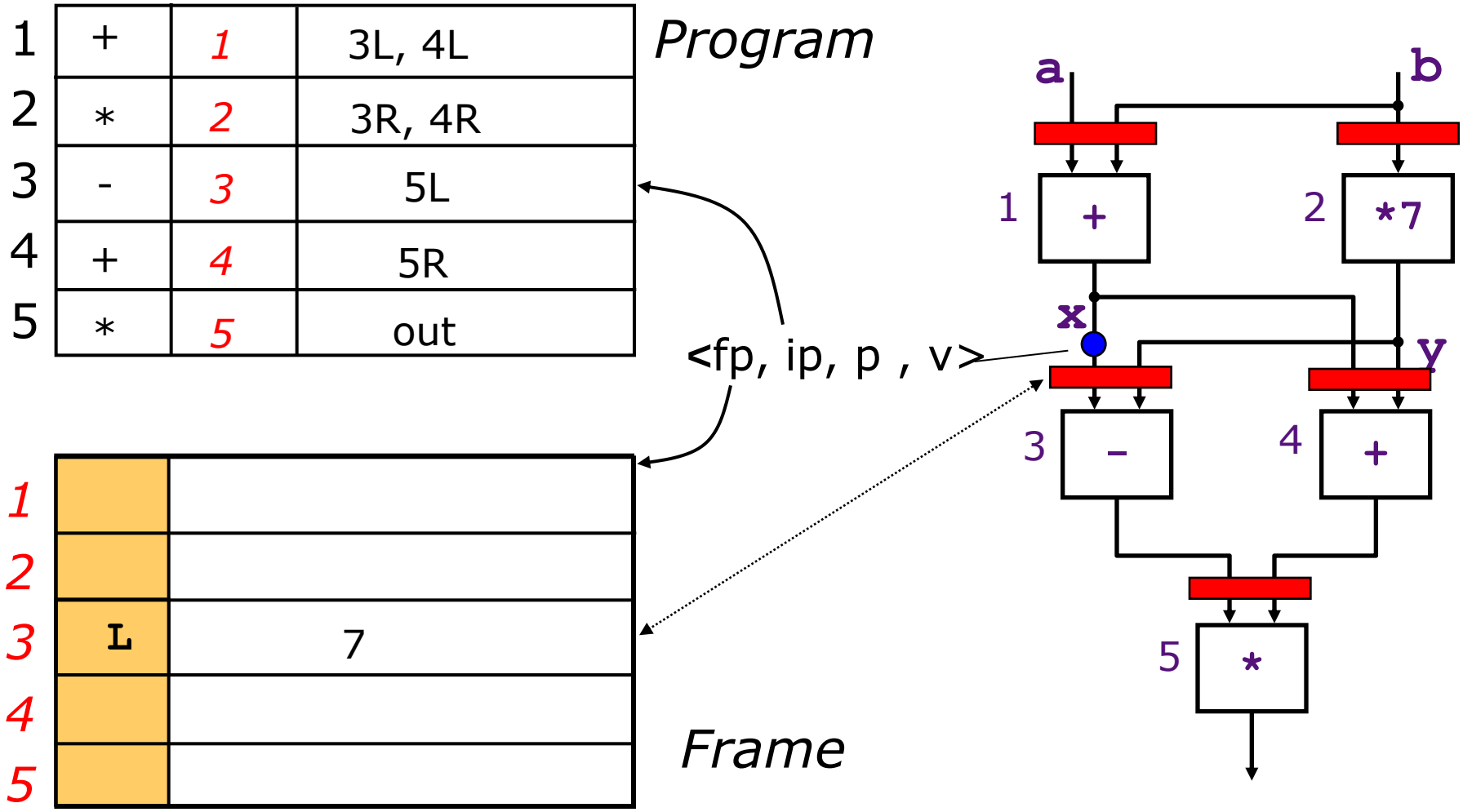
- Problem: Multiple dynamic instances can be active for the same instruction (i.e., due to loop iteration or invocation of function from different location)
- IP is not enough to distinguish between these different dynamic instances of the same static instruction



- Solution: Distinguish between different instances by creating new tags/frames (at the beginning of new iteration or call)

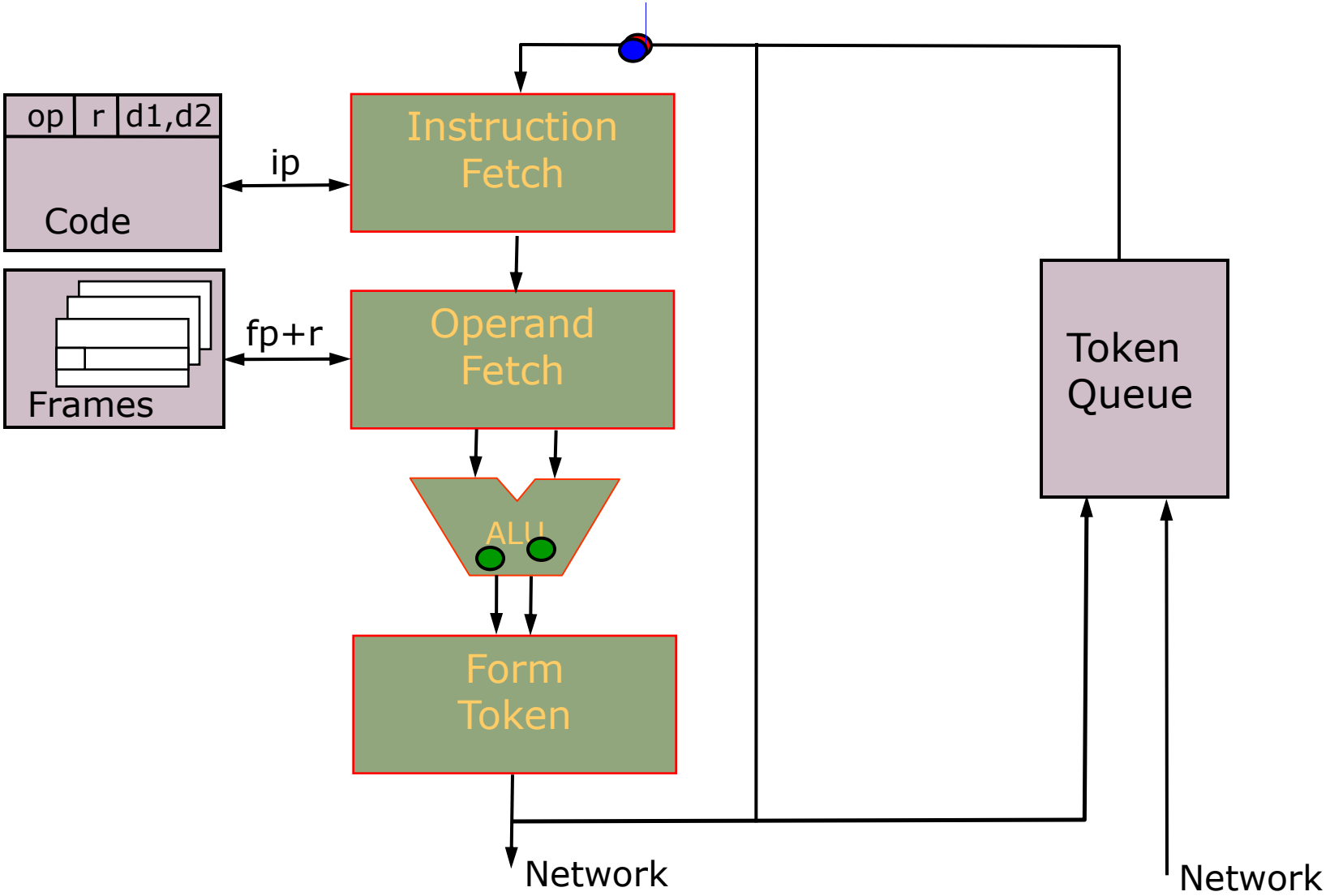


An Example Frame and Execution

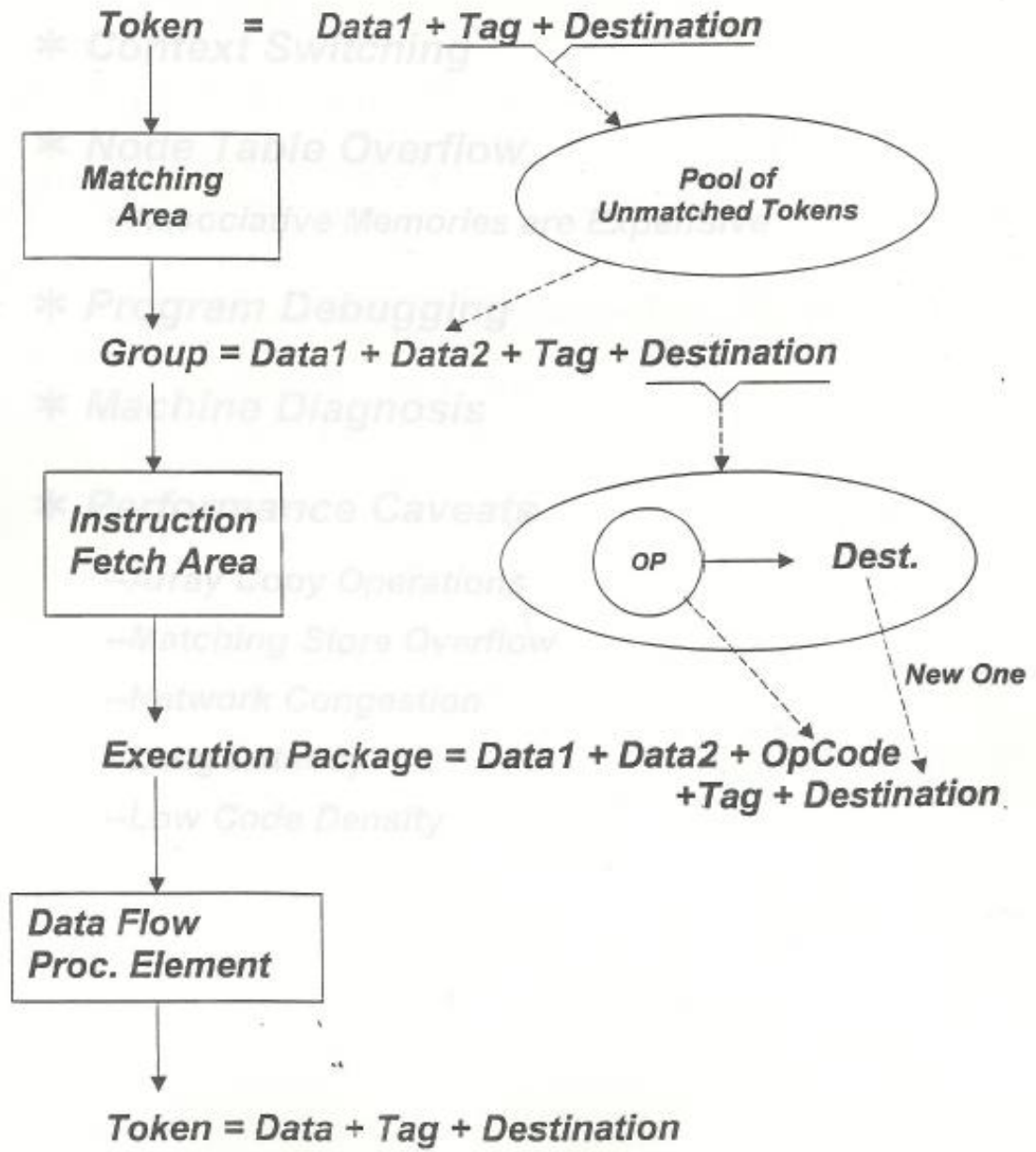


Need to provide storage for only one operand/operator

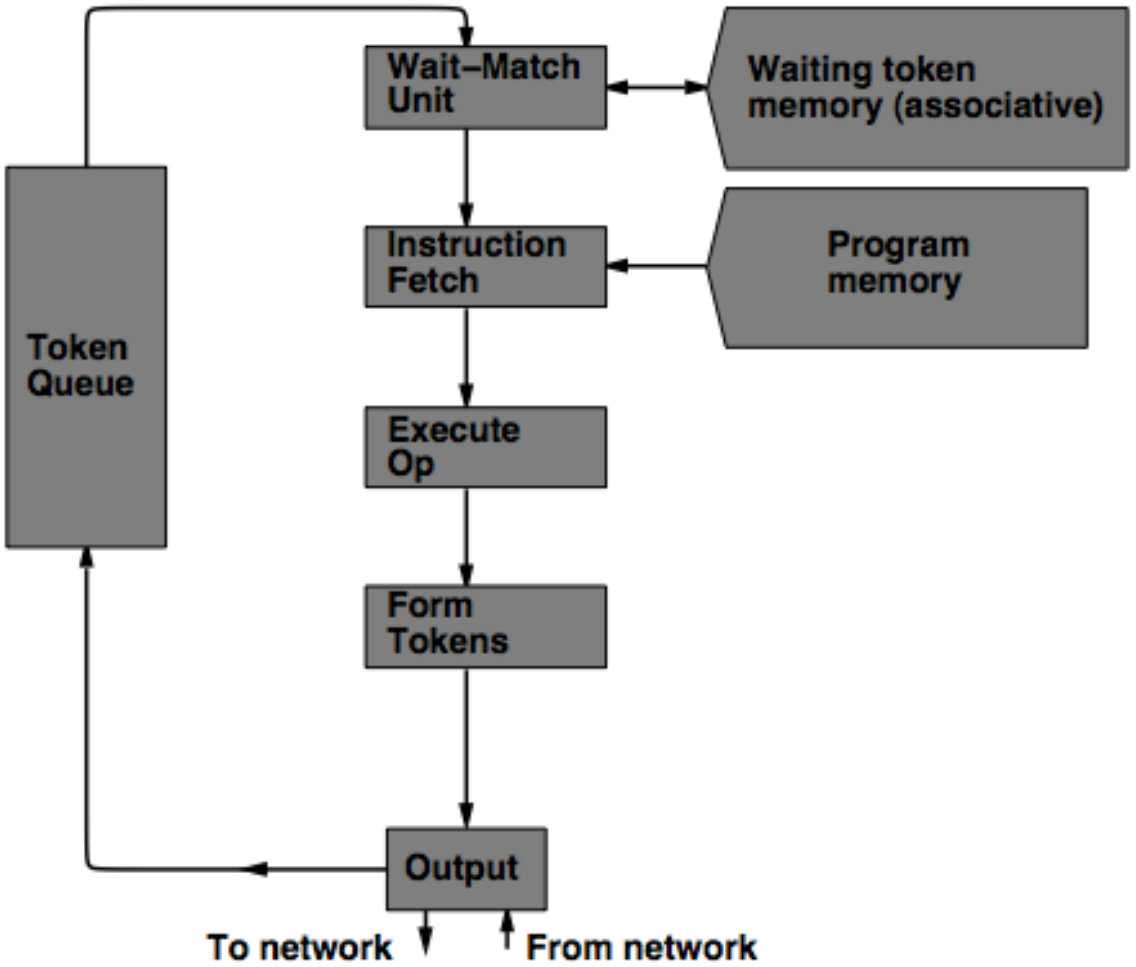
Monsoon Dataflow Processor [ISCA 1990]



A Dataflow Processor



MIT Tagged Token Data Flow Architecture

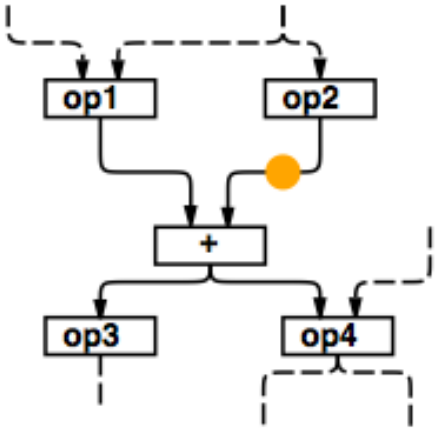


Wait–Match Unit: try to match incoming token and context id and a waiting token with same instruction address

- Success: Both tokens forwarded, fetch instruction
- Fail: Incoming token stored in Waiting Token Memory, bubble inserted

TTDA Data Flow Example

Conceptual



Encoding of graph

Program memory:

	Op–code	Destination(s)
109	op1	120L
113	op2	120R
120	+	141, 159L
141	op3	...
159	op4	... , ...

Re–entrancy ("dynamic" dataflow):

- Each invocation of a function or loop iteration gets its own, unique, "Context"
- Tokens destined for same instruction in different invocations are distinguished by a context identifier

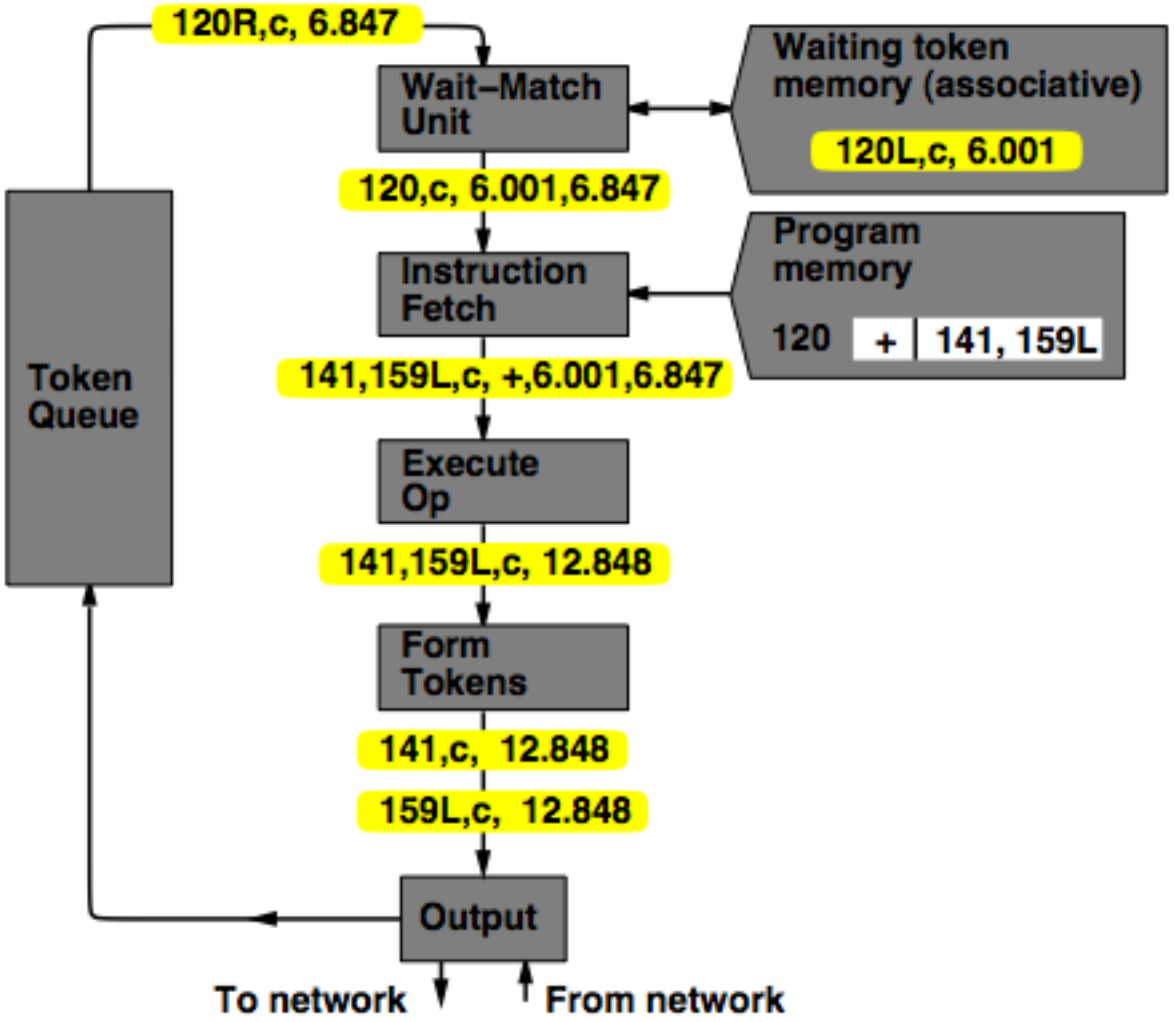
120R — Destination instruction address, Left/Right port
Ctxt — Context Identifier
6.847 — Value

Encoding of token:

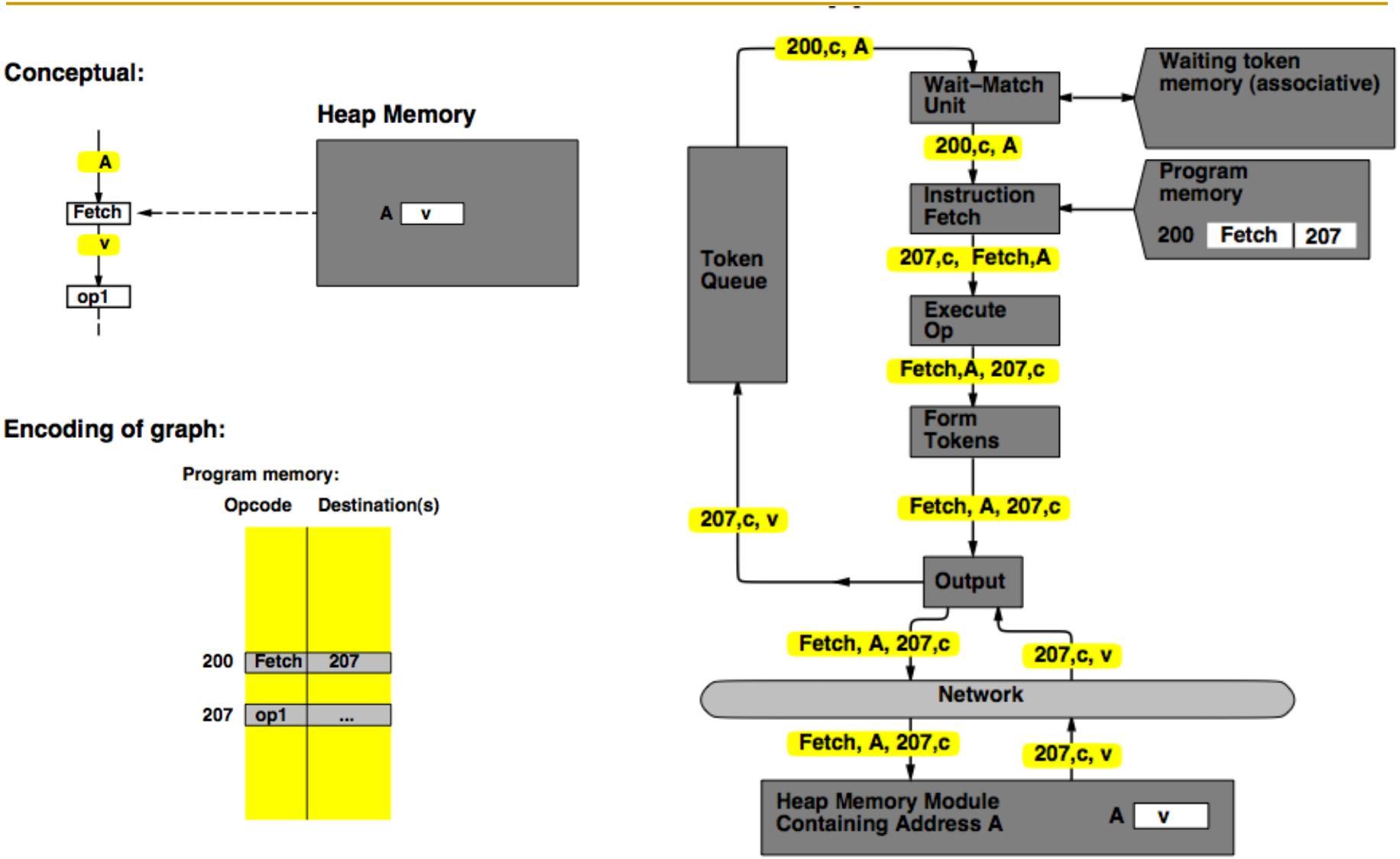
A "packet" containing:

120R — Destination instruction address, Left/Right port
6.847 — Value

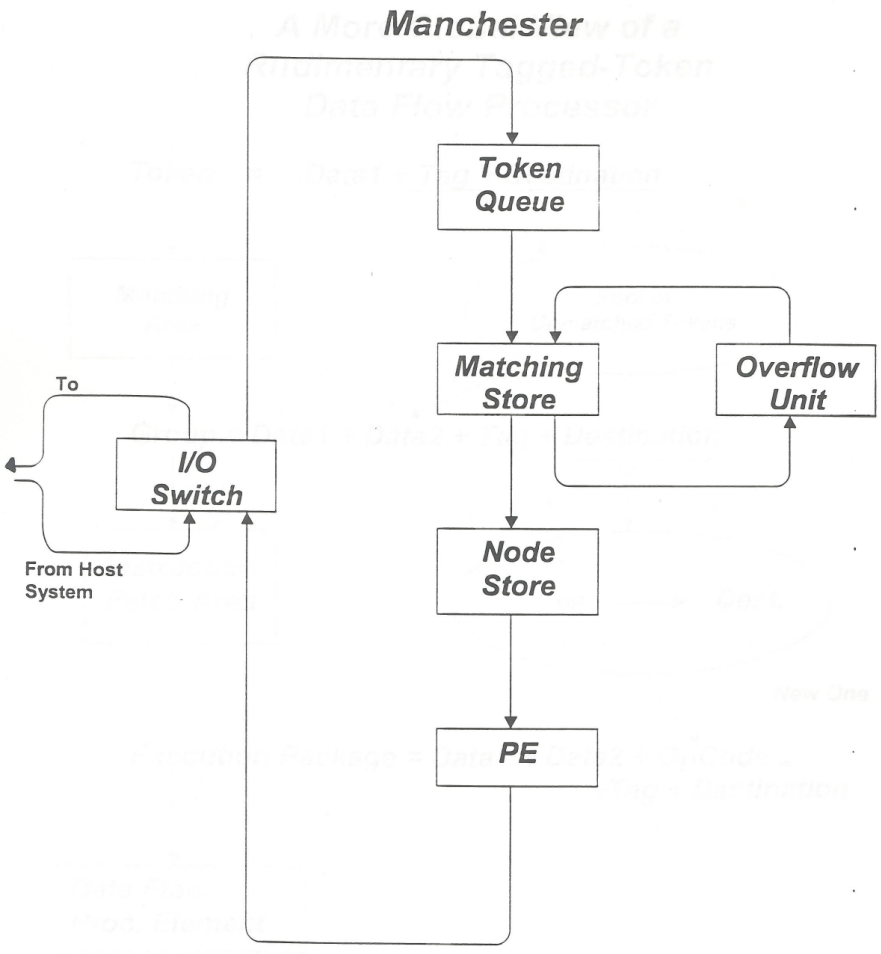
TTDA Data Flow Example



TTDA Data Flow Example



Manchester Data Flow Machine



- Matching Store: Pairs together tokens destined for the same instruction
- Large data set → overflow in overflow unit
- Paired tokens fetch the appropriate instruction from the node store

Data Flow Advantages/Disadvantages

- Advantages
  - Very good at exploiting irregular parallelism
  - Only real dependencies constrain processing

- Disadvantages
  - Debugging difficult (no precise state)
    - Interrupt/exception handling is difficult (what is precise state semantics?)
  - Implementing dynamic data structures difficult in pure data flow models
  - Too much parallelism? (Parallelism control needed)
  - High bookkeeping overhead (tag matching, data storage)
  - Instruction cycle is inefficient (delay between dependent instructions), memory locality is not exploited

Combining Data Flow and Control Flow

- Can we get the best of both worlds?
- Two possibilities
  - Model 1: Keep control flow at the ISA level, do dataflow underneath, preserving sequential semantics
  - Model 2: Keep dataflow model, but incorporate some control flow at the ISA level to improve efficiency, exploit locality, and ease resource management
    - Incorporate threads into dataflow: statically ordered instructions; when the first instruction is fired, the remaining instructions execute without interruption

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
  - Hwu and Patt, "HPSm, a high performance restricted data flow architecture having minimal functionality," ISCA 1986.

Further Reading on Data Flow

- ISA level dataflow
  - Gurd et al., "The Manchester prototype dataflow computer," CACM 1985.

- Microarchitecture-level dataflow:
- Patt, Hwu, Shebanow, "HPS, a new microarchitecture: rationale and introduction," MICRO 1985.
- Patt et al., "Critical issues regarding HPS, a high performance microarchitecture," MICRO 1985.
  - Hwu and Patt, "HPSm, a high performance restricted data flow architecture having minimal functionality," ISCA 1986.