Computer Architecture: Out-of-Order Execution II

Prof. Onur Mutlu
Carnegie Mellon University

A Note on This Lecture

- These slides are partly from 18-447 Spring 2013, Computer Architecture, Lecture 15

- Video of that lecture:
  - http://www.youtube.com/watch?v=f-XL4BNRoBA until 50:00

Last Lecture

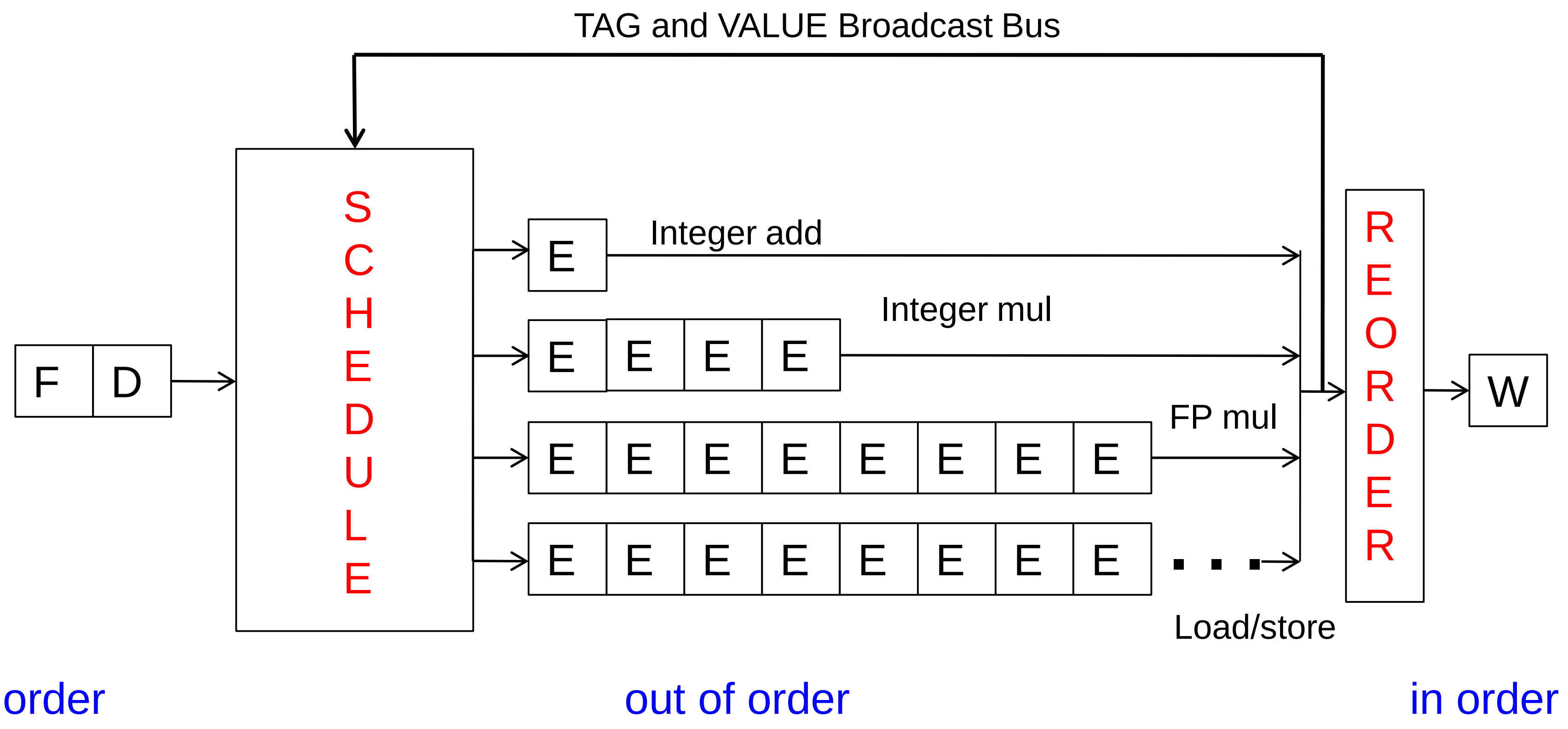
- Out-of-order execution
  - Tomasulo's algorithm
  - Example
  - OoO as restricted dataflow execution

Today

- Wrap up out-of-order execution
  - Memory dependence handling
  - Alternative designs

Out-of-Order Execution (Dynamic Instruction Scheduling)

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: 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)

Review: 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 execution
- Corollary 3: When a load/store has its address ready, there may be younger/older loads/stores with undetermined addresses in the machine

Review: 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 Dependencies

- 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)
  - 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

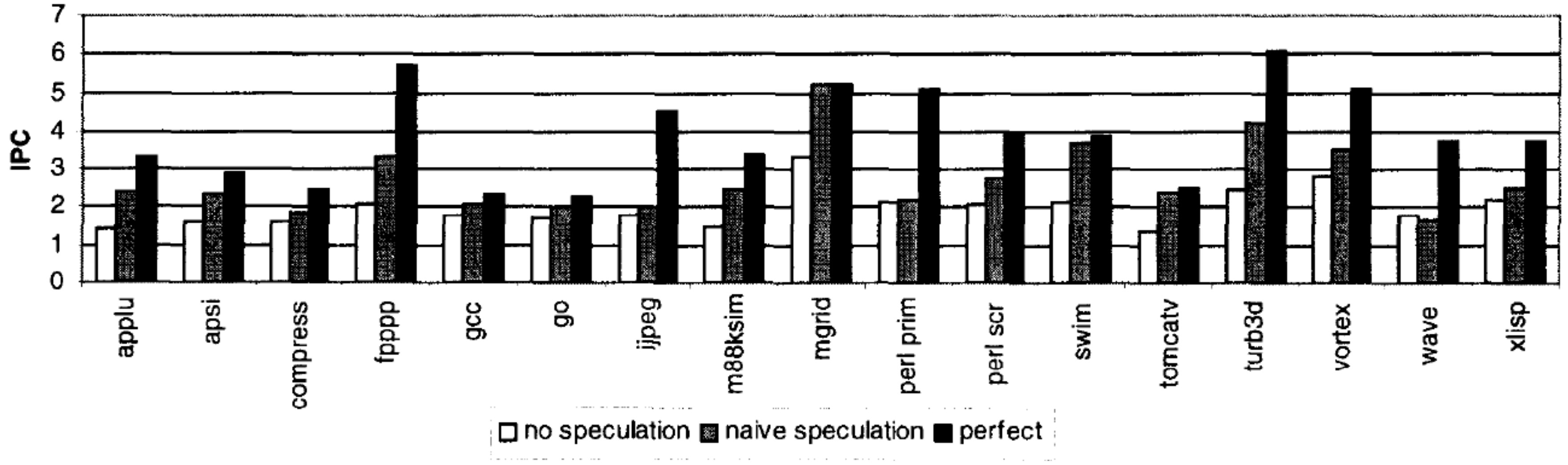
  + More accurate. Load store dependencies persist over time

  -- 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

Food for Thought for You

- Many other design choices

- Should reservation stations be centralized or 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?

- 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?

Recommended Readings

- 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.

The following slides are for your benefit

Other Approaches to Concurrency (or Instruction Level Parallelism)

Approaches to (Instruction-Level) Concurrency

- Out-of-order execution
- Dataflow (at the ISA level)
- SIMD Processing
- VLIW

- Systolic Arrays
- Decoupled Access Execute

Data Flow:
Exploiting Irregular Parallelism

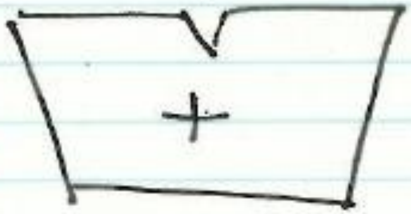
Remember: State of RAT and RS in Cycle 7

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

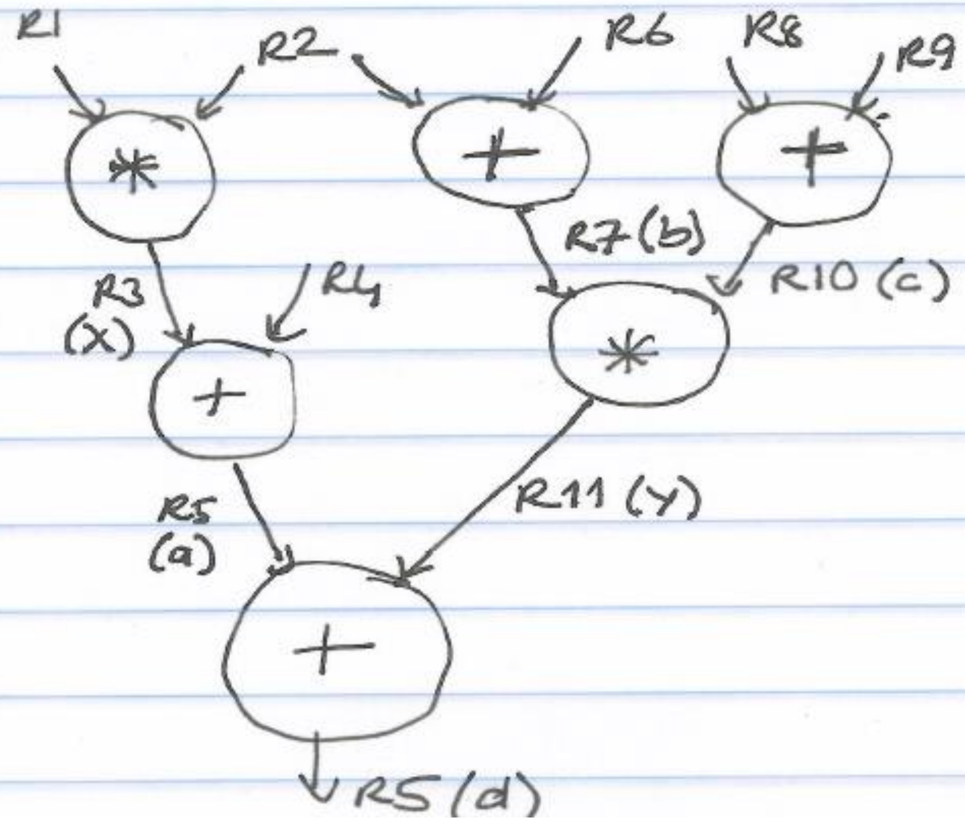
Remember: Dataflow Graph

```
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

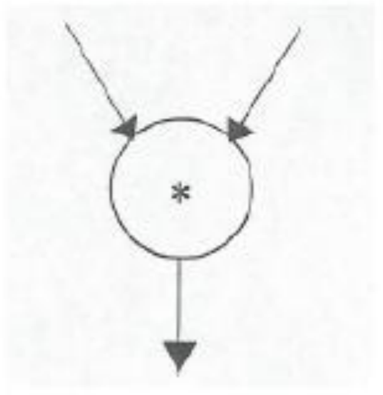
Arcs: tags in Tomasulo's algorithm

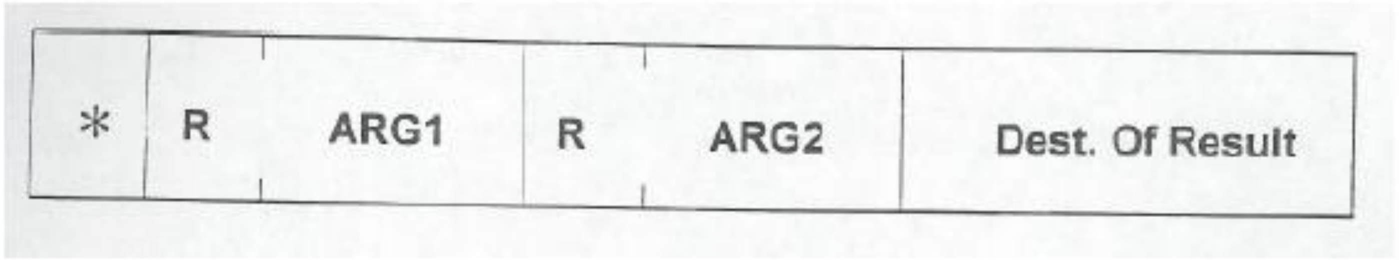


Review: More on Data Flow

- 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

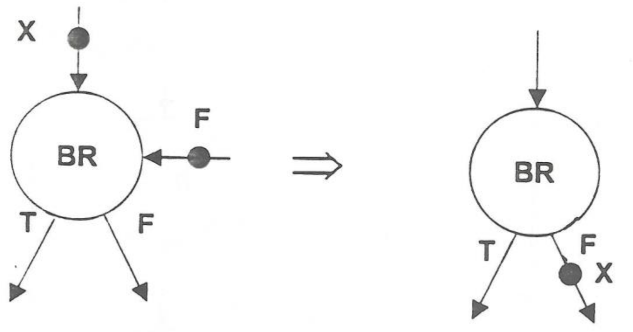
- Data flow node and its ISA representation



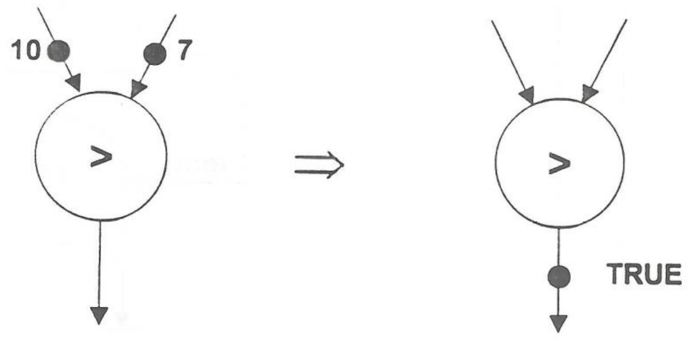


Data Flow Nodes

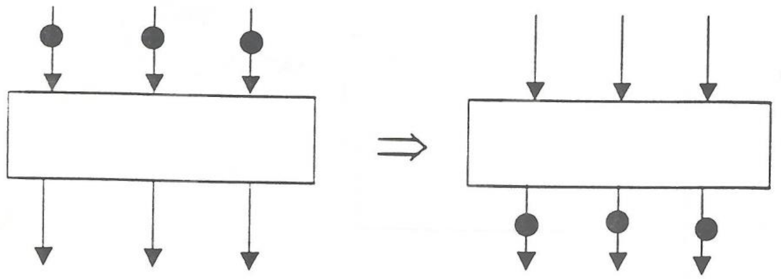
*Conditional



*Relational

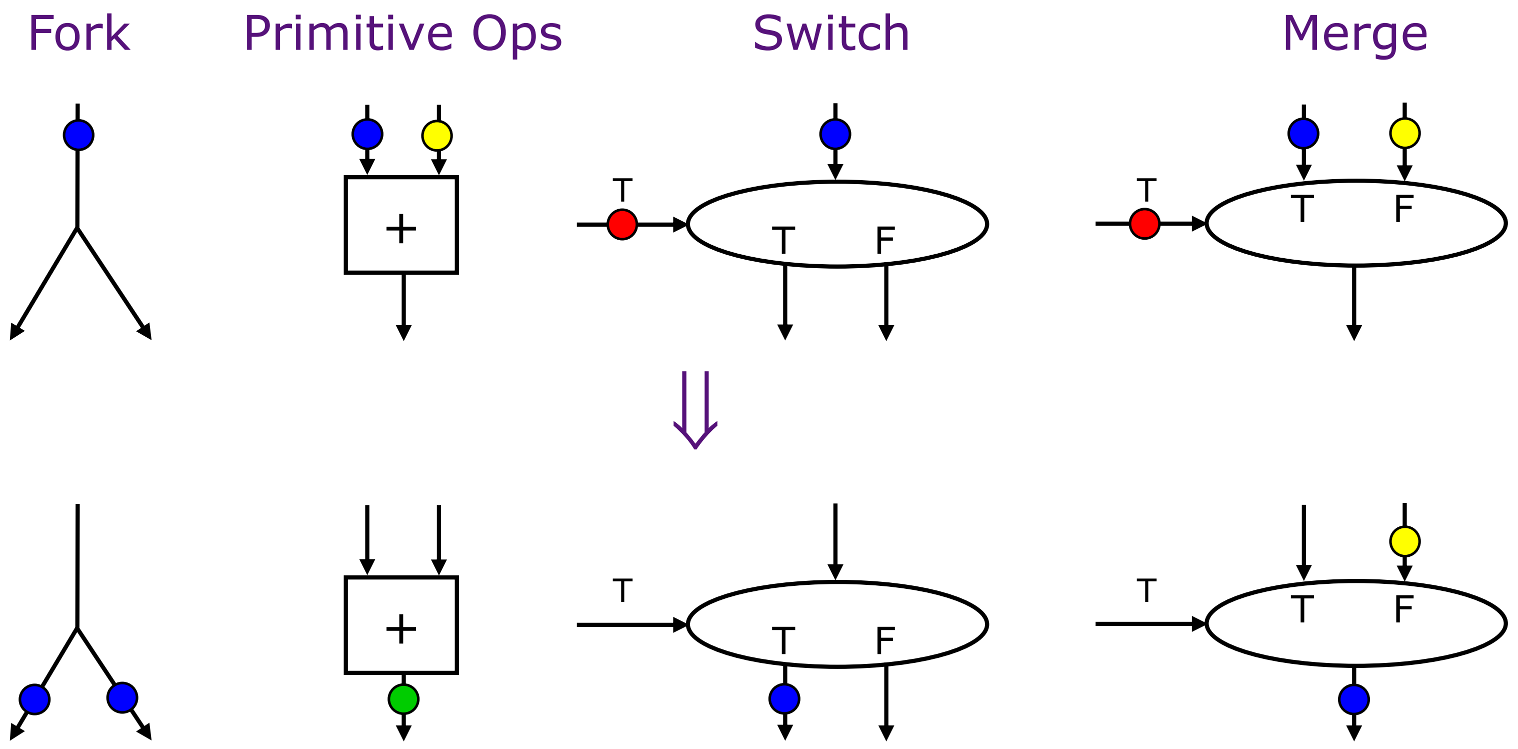


*Barrier Synch



Dataflow Nodes (II)

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



Dataflow Graphs

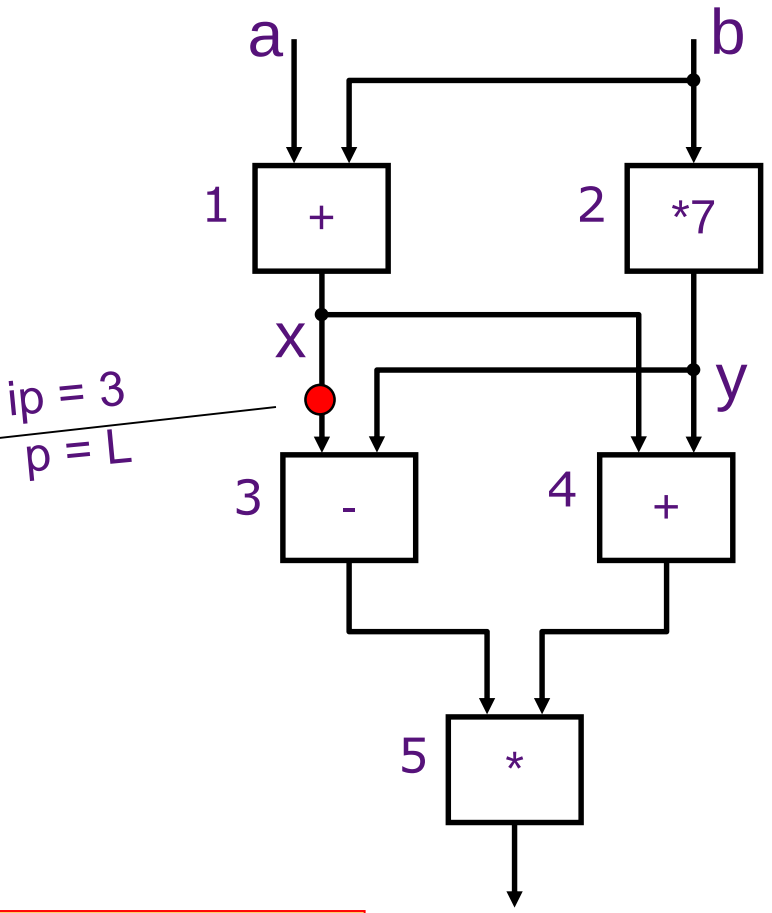
```
{x = a + b;
 y = b * 7
in
   (x-y) * (x+y)}
```

- Values in dataflow graphs are represented as tokens

token < ip , p , v >

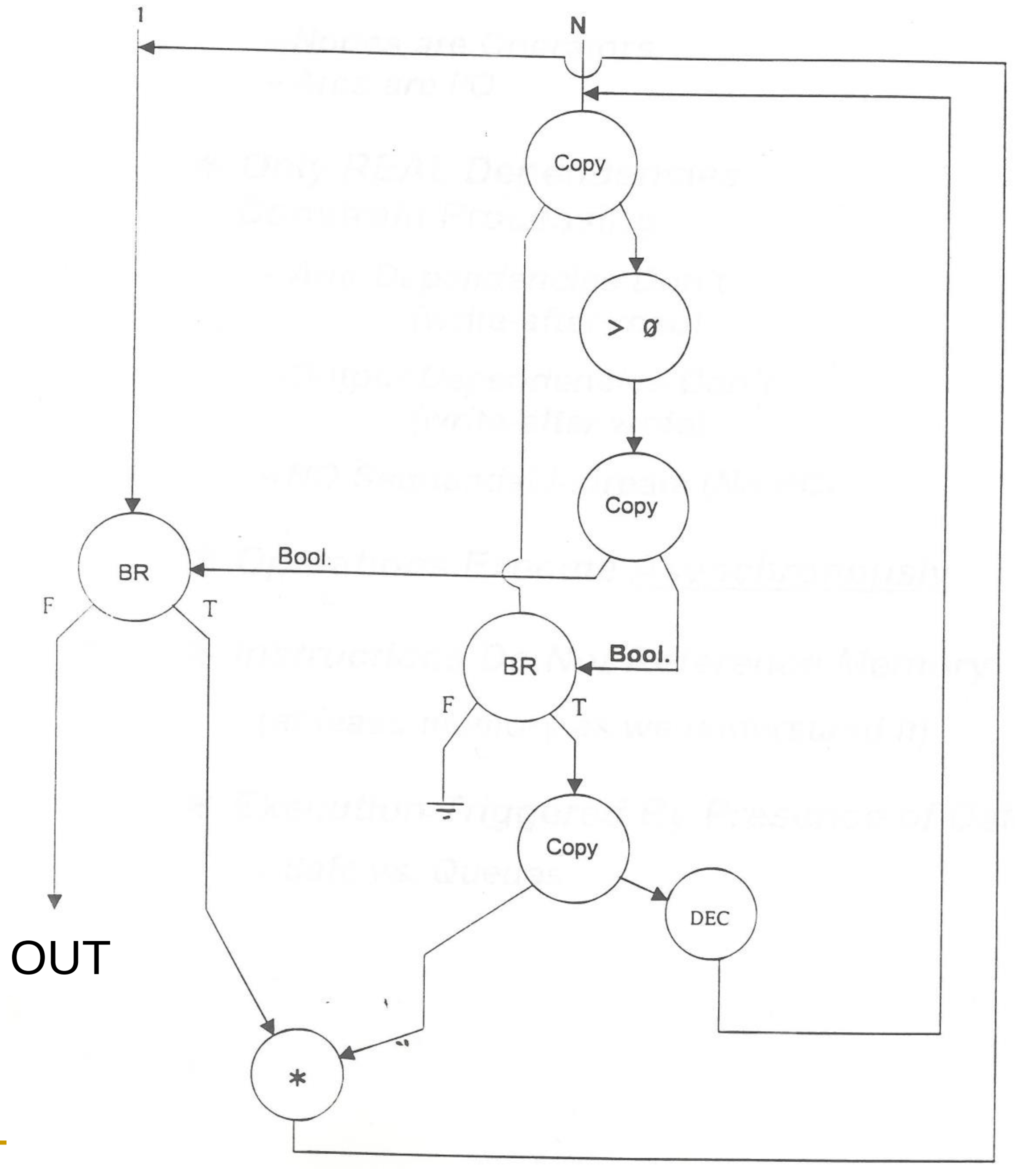
instruction ptr port data

- 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

```
a := x + y
b := a × a
c := 4 — a
```

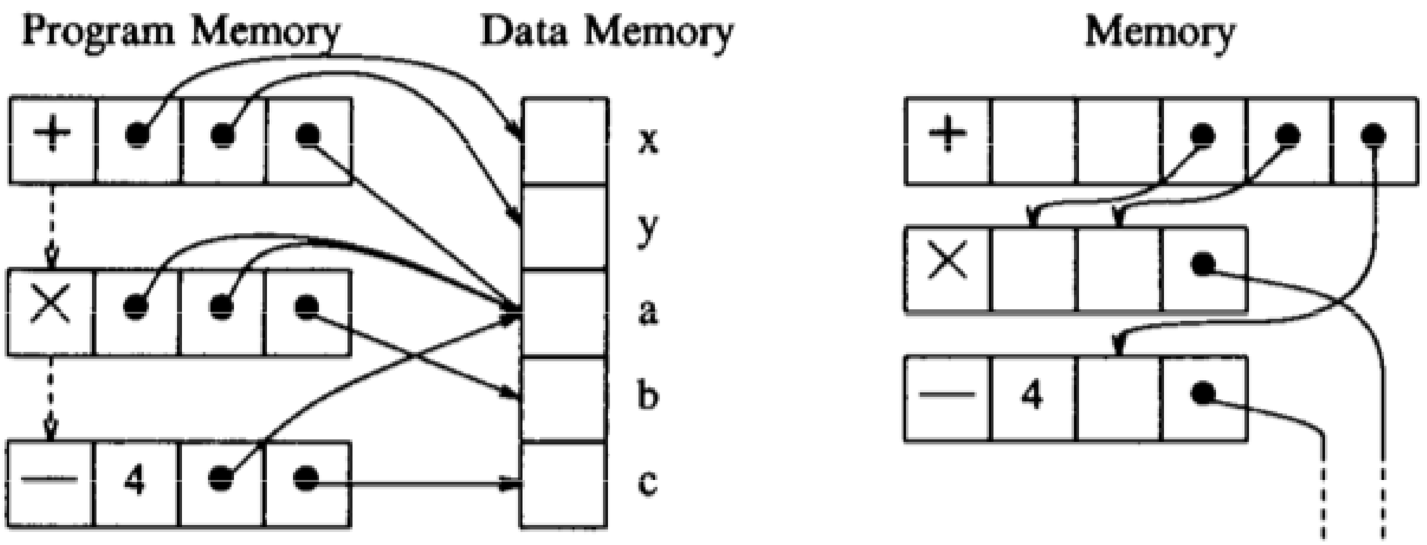
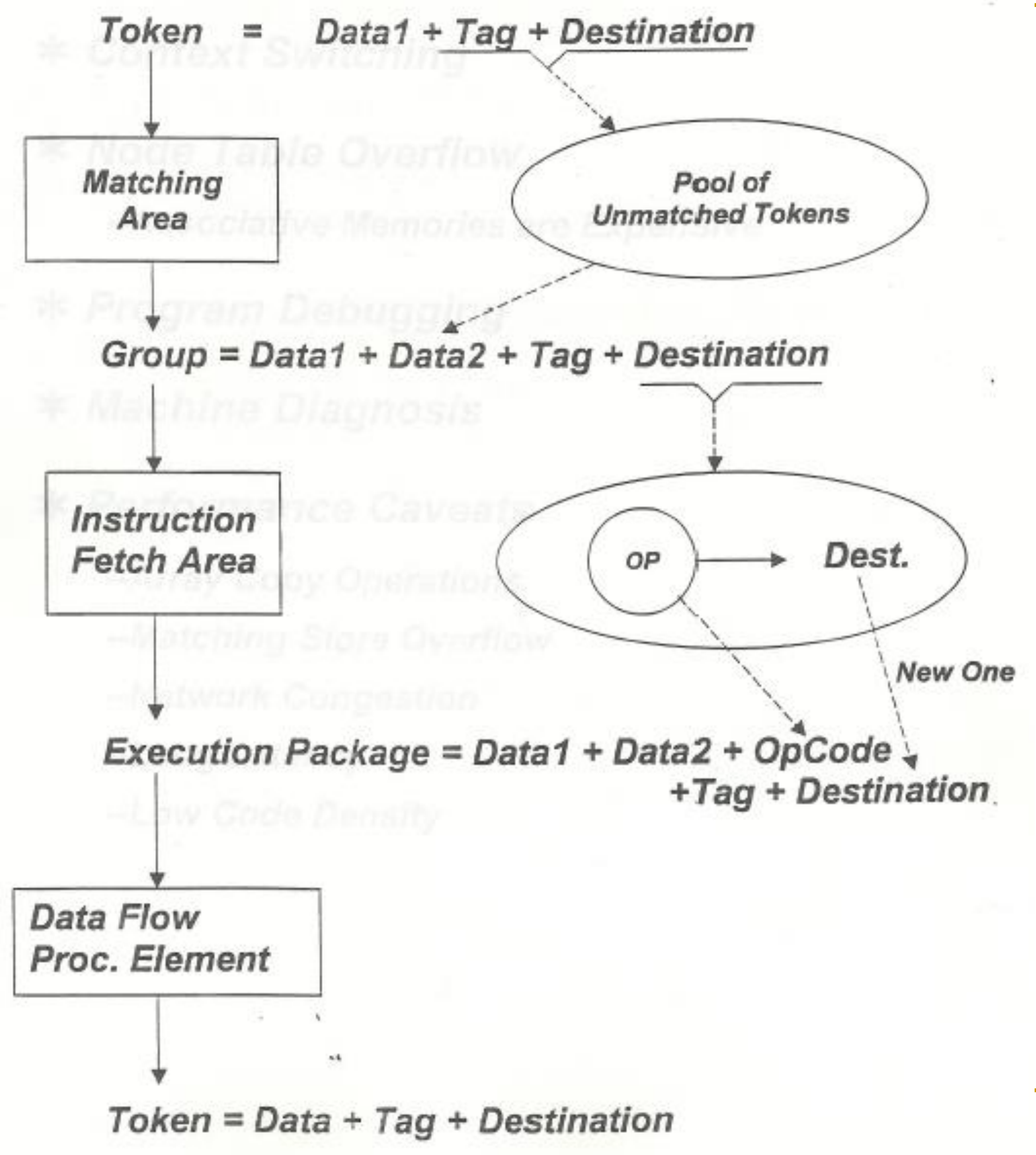


Figure 2. A comparison of control flow and dataflow programs. On the left a control flow program for a computer with memory-to-memory instructions. The arcs point to the locations of data that are to be used or created. Control flow arcs are indicated with dashed arrows; usually most of them are implicit.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

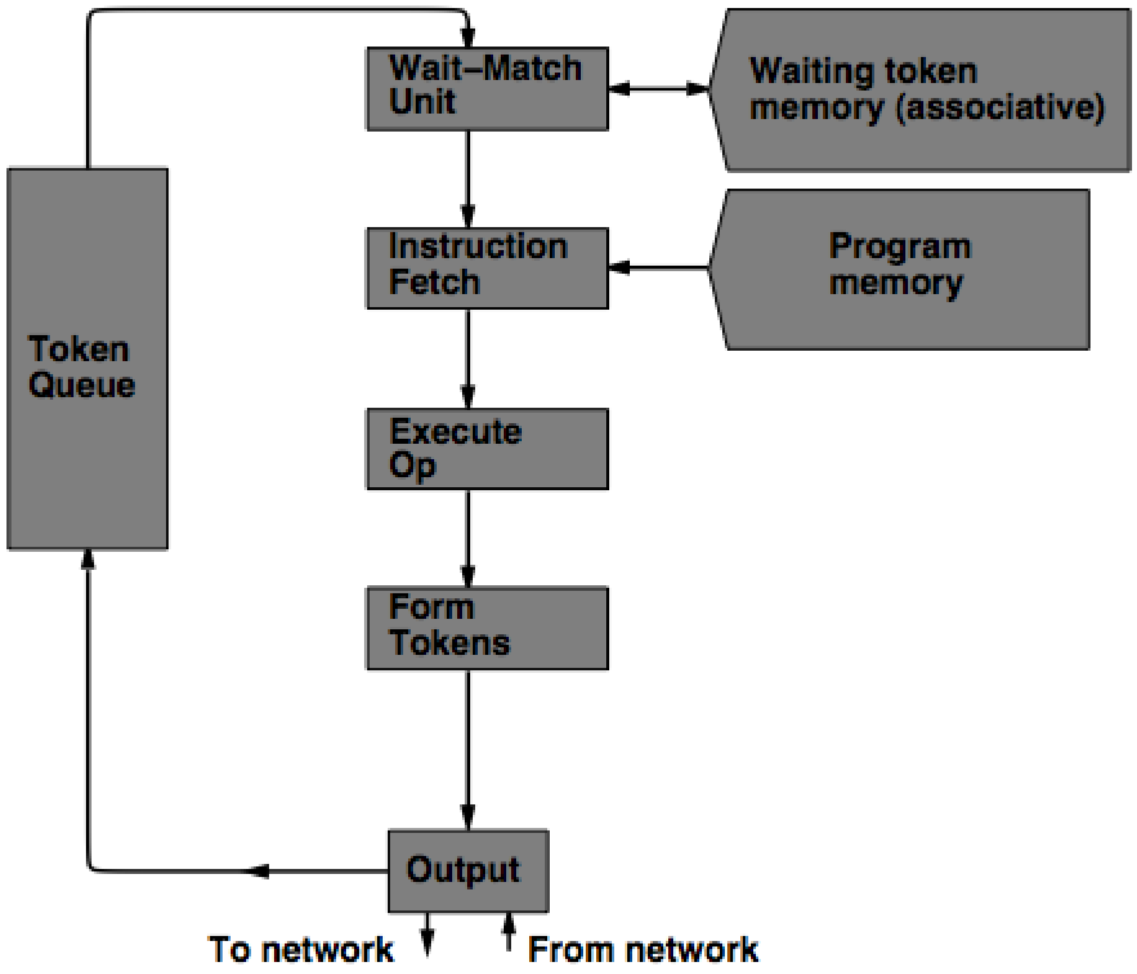
Data Flow Characteristics

- 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 I-stream
  - No program counter
- Operations execute asynchronously
- Execution triggered by the presence of data

A Dataflow Processor


Token = Data1 + Tag + Destination
Matching Area
Pool of Unmatched Tokens
Group = Data1 + Data2 + Tag + Destination
Instruction Fetch Area
OP
Dest.
New One
Execution Package = Data1 + Data2 + OpCode +Tag + Destination
Data Flow Proc. Element
Token = Data + Tag + Destination

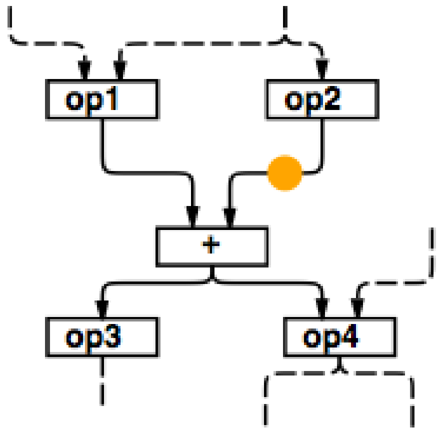
MIT Tagged Token Data Flow Architecture



- Wait–Match Unit: try to match incoming token and context id and a waiting token with same instruction address
  - Success: Both tokens forwarded
  - Fail: Incoming token ––> Waiting Token Mem, bubble (no-op forwarded)

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

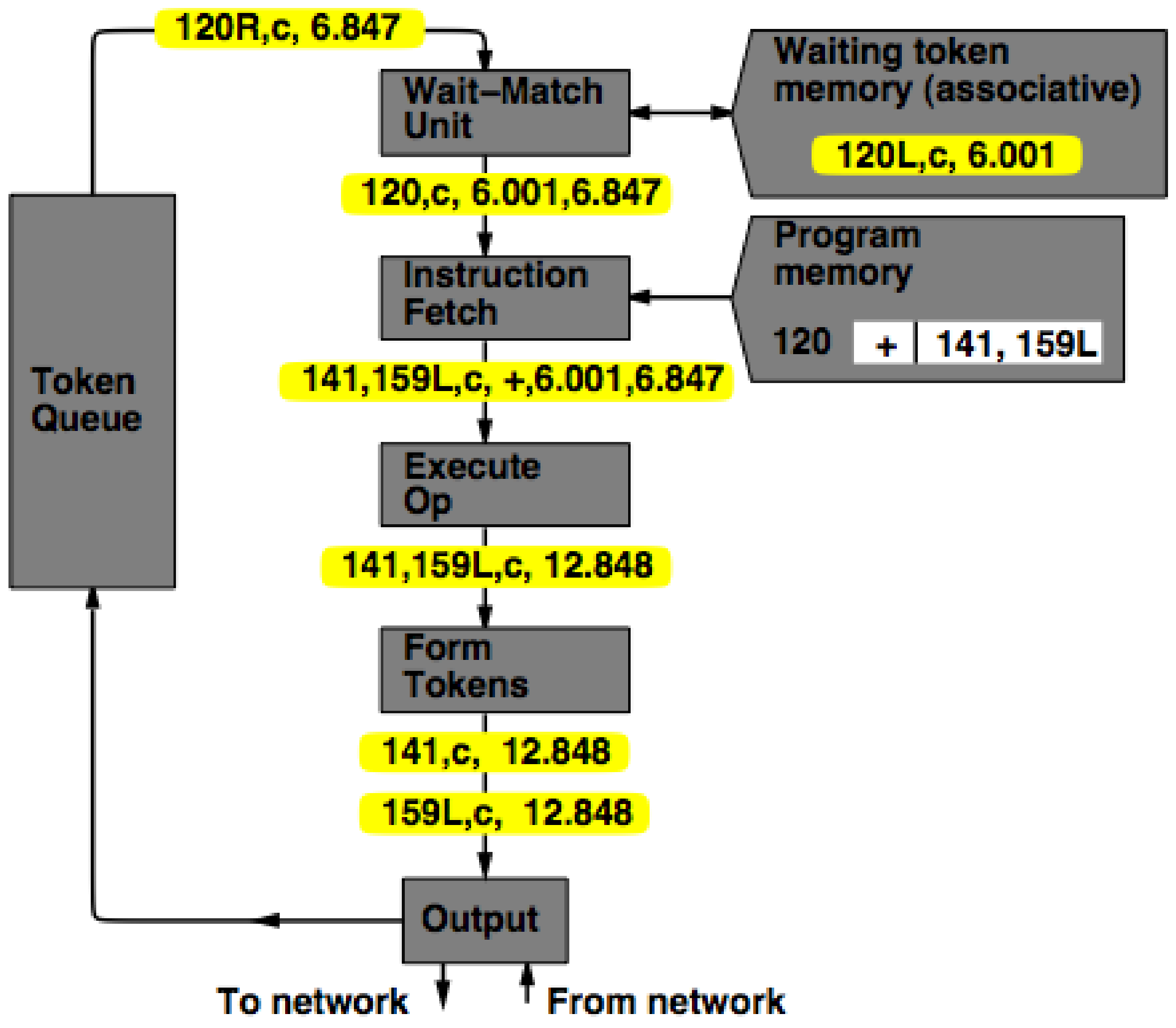
120R	Destination instruction address, Left/Right port
Ctxt	Context Identifier
6.847	Value

Encoding of token:

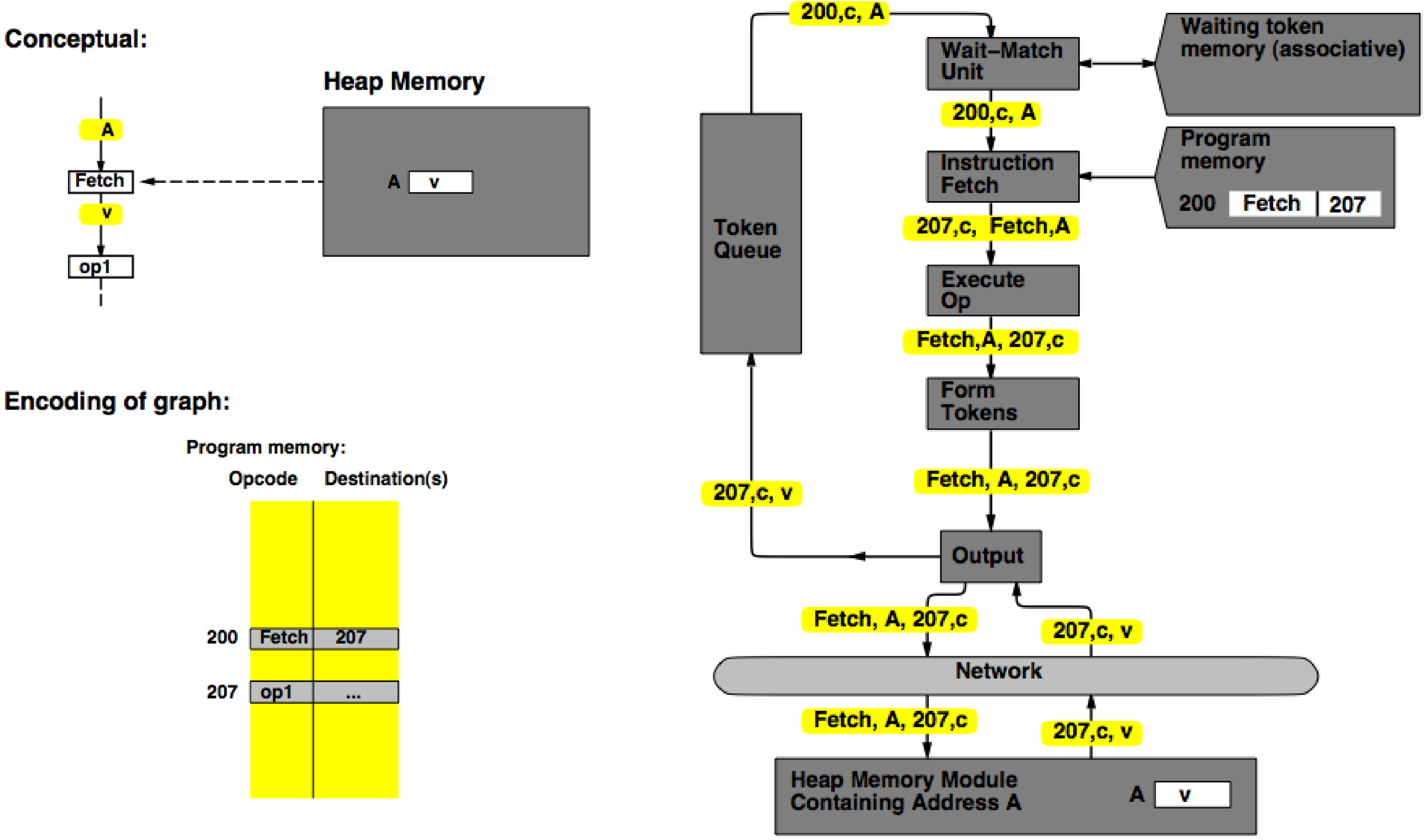
A "packet" containing:

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

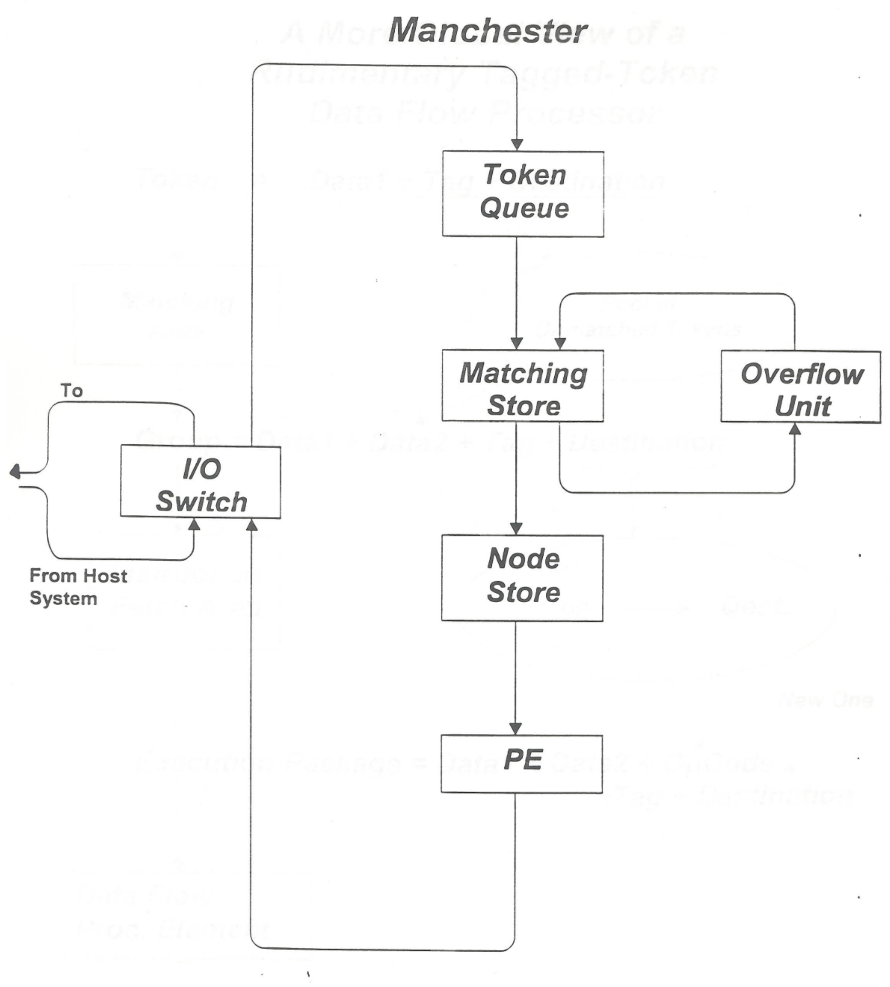
TTDA Data Flow Example



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
  - No precise state
    - Interrupt/exception handling is difficult
    - Debugging very difficult
  - Bookkeeping overhead (tag matching)
  - Too much parallelism? (Parallelism control needed)
    - Overflow of tag matching tables
  - Implementing dynamic data structures difficult

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