Data Flow
In a data flow machine, a program consists of data flow nodes.

A data flow node fires (fetched and executed) when all its inputs are ready.
  - i.e. when all inputs have tokens.

Data flow node and its ISA representation.
Review: Static Dataflow

- Allows only one instance of a node to be enabled for firing

- A dataflow node is fired only when all of the tokens are available on its input arcs and no tokens exist on any of its output arcs

Review: Static versus Dynamic Dataflow Machines

Figure 1. The basic organization of the static dataflow model.

Figure 2. An instruction template for the static dataflow model.

Figure 3. The general organization of the dynamic dataflow model.

Figure 4. An instruction format for the dynamic dataflow model.
Allocate instruction templates, i.e., a frame, dynamically to support each loop iteration and procedure call
- termination detection needed to deallocate frames

The code can be shared if we separate the code and the operand storage
Figure 10. An implementation of a loop using tagged tokens. At the start of the loop a new tag area is allocated. Tokens belonging to consecutive iterations receive consecutive tags within this area. The tag from before the loop is restored on tokens that exit from the loop.

Figure 11. Interface for a procedure call. On the left a call of procedure P whose graph is on the right. P has one parameter and one return value. The actual parameter receives a new tag and is sent to the input node of P and concurrently a token containing address A is sent to the output node. This SEND-TO-DESTINATION node transmits the other input token to a node of which the address is contained in the first token. The effect is that, when the return value of the procedure becomes available, the output node sends the result to node A, which restores the tag belonging to the calling expression.
**Review: Control of Parallelism**

- **Problem:** Many loop iterations can be present in the machine at any given time
  - 100K iterations on a 256 processor machine can swamp the machine (thrashing in token matching units)
  - Not enough bits to represent frame id

- **Solution:** Throttle loops. Control how many loop iterations can be in the machine at the same time.
  - Requires changes to loop dataflow graph to inhibit token generation when number of iterations is greater than N
Data Structures

- Dataflow by nature has write-once semantics
- Each arc (token) represents a data value
- An arc (token) gets transformed by a dataflow node into a new token (token) → No persistent state...

- Data structures as we know of them (in imperative languages) are structures with persistent state
- Why do we want persistent state?
  - More natural representation for some tasks? (bank accounts, databases, ...)
  - To exploit locality
Data Structures in Dataflow

- Data structures reside in a structure store
  - tokens carry pointers

- I-structures: Write-once, Read multiple times or
  - allocate, write, read, ..., read, deallocate
  - No problem if a reader arrives before the writer at the memory location
I-Structures

Possible execution sequence producing this structure:

- Attempt to READ(n+2) for instruction X
- WRITE(n+m)
- Attempt to READ(n+3) for instruction Z
- WRITE(n)
- Attempt to READ(n+2) for instruction Y
- READ(n)

Fig. 7. I-structure memory.
Dynamic Data Structures

- Write-multiple-times data structures
- How can you support them in a dataflow machine?
  - Can you implement a linked list?

- What are the ordering semantics for writes and reads?

- Functional vs. imperative languages
  - Side effects and state vs. no side effects and no state
MIT Tagged Token Data Flow Architecture

- Resource Manager Nodes
  - responsible for Function allocation (allocation of context identifiers), Heap allocation, etc.
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**

- op1
- op2
- +
- op3
- op4

**Encoding of graph**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Destination(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>109</td>
<td>op1, 120L</td>
</tr>
<tr>
<td>113</td>
<td>op2, 120R</td>
</tr>
<tr>
<td>120</td>
<td>+, 141, 159L</td>
</tr>
<tr>
<td>141</td>
<td>op3, ...</td>
</tr>
<tr>
<td>159</td>
<td>op4, ...</td>
</tr>
</tbody>
</table>

**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

**Encoding of token:**

A "packet" containing:

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

- Ctxt 6.847
  - Context Identifier
  - Value
TTDA Data Flow Example
TTDA Data Flow Example

Conceptual:

Heap Memory

Encoding of graph:

Program memory:

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Destination(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>Fetch, 207</td>
</tr>
<tr>
<td>207</td>
<td>op1, ...</td>
</tr>
</tbody>
</table>

Wait-Match Unit

200, c, A

Instruction Fetch

207, c, Fetch, A

Execute Op

Fetch, A, 207, c

Form Tokens

Fetch, A, 207, c

Output

Network

Fetch, A, 207, c

Fetch Memory Module Containing Address A

A, v
TTDA Synchronization

- Heap memory locations have FULL/EMPTY bits
- If the heap location is EMPTY, heap memory module queues request at that location. When "I–Fetch" request arrives (instead of "Fetch"),
- Later, when "I–Store" arrives, pending requests are discharged
- No busy waiting
- No extra messages
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 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
- Single assignment languages and functional programming
  - E.g., SISAL in Manchester Data Flow Computer
  - No mutable state
Data Flow Advantages/Disadvantages

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

- **Disadvantages**
  - Interrupt/exception handling is difficult
    - Context switching overhead and precise state
  - High bookkeeping overhead (tag matching, data storage)
  - Debugging difficult (no precise state)
  - Too much parallelism? (Parallelism control needed)
    - Overflow of the node tables
  - Implementing dynamic data structures difficult
  - Instruction cycle is inefficient (delay between dependent instructions)
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 successful
  - Out of order execution
  - Hwu and Patt, “HPSm, a high performance restricted data flow architecture having minimal functionality,” ISCA 1986.
Combining Data Flow and Control Flow

- Can we get the best of both worlds?

- Two possibilities
  - Keep control flow at the ISA level, do dataflow underneath, preserving sequential semantics
  - Keep dataflow at the ISA level, do limited control flow underneath
    - Incorporate threads into dataflow: statically ordered instructions; when the first instruction is fired, the remaining instructions execute without interruption