The models we have examined all assumed:
- Instructions are fetched and retired in sequential, control flow order.

This is part of the Von-Neumann model of computation:
- Single program counter
- Sequential execution
- Control flow determines fetch, execution, commit order

What about out-of-order execution?
- Architecture level: Obey the control-flow model
- Uarch level: A window of instructions executed in data-flow order → execute an instruction when its operands become available

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:

*Conditional

*Relational

*Barrier Synch

Data Flow Nodes
Data Flow Nodes (II)

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

```
Fork  Primitive Ops  Switch  Merge
```

Dataflow Graphs

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

- Values in dataflow graphs are represented as tokens
- An operator executes when all its input tokens are present; copies of the result token are distributed to the destination operators

Example Data Flow Program

```
a := x + y
b := a * a
c := 4 - a
```

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.
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

Static Dataflow Machine: Instruction Templates

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Destination 1</th>
<th>Destination 2</th>
<th>Operand 1</th>
<th>Operand 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>3L</td>
<td>4L</td>
<td></td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>3R</td>
<td>4R</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>5L</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>5R</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*</td>
<td></td>
<td>* out</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

presence bits

Each arc in the graph has an operand slot in the program

Static Dataflow Machine (Dennis+, ISCA 1974)

- Many such processors can be connected together
- Programs can be statically divided among the processors

Static Data Flow Machines

- Mismatch between the model and the implementation
  - The model requires *unbounded FIFO token queues* per arc but the architecture provides storage for one token per arc
  - The architecture *does not ensure FIFO order* in the reuse of an operand slot

- The static model *does not support*
  - Reentrant code (and code sharing)
    - Function calls
    - Loops
  - Data Structures
  - Non-strict functions
  - Latency hiding
Exploiting All The Parallelism

```plaintext
Def Vsum A, B
    {  C = array(1,n);
       For j From 1 To n Do
    } In
    C;
```

Dynamic Dataflow Architectures

- 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

```
allocate C
fetch A
fetch B
+ +
store
```

Static versus Dynamic Dataflow Machines

MIT Tagged Token Data Flow Architecture

- Resource Manager Nodes
  - responsible for Function allocation (allocation of context/frame 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 --&gt; Waiting Token Mem, bubble (no-op forwarded)

TTDA Data Flow Example

- **Function Calls:**
  - Need extra mechanism to direct the output token of the function to the proper calling site
  - Usually done by sending special token containing the return node address

Encoding of graph:

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

Encoding of token:

- A "packet" containing:
  - Destination instruction address, Left/Right port
  - Value
**Concept of Tagging**
- Each invocation receives a separate tag

**Procedure Linkage Operators**
- Procedure linkage operators include:
  - Get frame
  - Extract tag
  - Change Tag 0
  - Change Tag 1
  - Change Tag n

**Graph for f**
- Fork: token in frame 0, token in frame 1

**Loops and Function Calls Summary**

**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 arc (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

- Write-multiple-times data structures
- How can you support them in a dataflow machine?
  - M-Structures
    - Can you implement a linked list?
  - What are the ordering semantics for writes and reads?
- Imperative vs. functional languages
  - Side effects and mutable state vs.
  - No side effects and no mutable state
TTDA Data Flow Example

Conceptual:

Heap Memory

Encoding of graph:

Program memory:
Opcode Destination(s)

200 Fetch 307
207 ip+r ... 

Waiting token memory (associative)

Token Queue

200, c, A

Instruction Fetch

207, c, A

Fetch, A, 207, c

Network

207, c, v

207, c, v

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

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

Monsoon Processor (ISCA 1990)

Manchester Data Flow Machine
A Frame in Dynamic Dataflow

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>+</td>
<td>1</td>
<td>3L, 4L</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>*</td>
<td>2</td>
<td>3R, 4R</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td>3</td>
<td>5L</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>+</td>
<td>4</td>
<td>5R</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>*</td>
<td>5</td>
<td>out</td>
<td></td>
</tr>
</tbody>
</table>

Program

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.

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
  - 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 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

Model 2 Example: Macro Dataflow

- Data flow execution of large blocks, control flow within a block

Benefits of Control Flow within Data Flow

- Strongly-connected block: Strongly-connected subgraph of the dataflow graph

- Executed without interruption. Atomic: all or none.

- Benefits of the atomic block:
  - Dependent or independent instructions can execute back to back → improved processing element utilization
  - Exploits locality with registers → reduced comm. delay
  - No need for token matching within the block → simpler, less overhead
  - No need for token circulation (which is slow) within the block
  - Easier to implement serialization and critical sections

Macro Dataflow Program Example

- Incorporate threads into dataflow: statically ordered instructions; when the first instruction is fired, the remaining instructions execute without interruption

Figure 1 An Example of a Strongly Connected Block.

Figure 7. Organization of a macro-dataflow processing element.

Macro Dataflow Machine Example

Model 1 Example: Restricted Data Flow

- Data flow execution under sequential semantics and precise exceptions

Systolic Arrays

Why Systolic Architectures?

- Idea: Data flows from the computer memory in a rhythmic fashion, passing through many processing elements before it returns to memory

- Similar to an assembly line
  - Different people work on the same car
  - Many cars are assembled simultaneously
  - Can be two-dimensional

- Why? Special purpose accelerators/architectures need
  - Simple, regular designs (keep # unique parts small and regular)
  - High concurrency → high performance
  - Balanced computation and I/O (memory access)

Systolic Architectures

- Basic principle: Replace a single PE with a regular array of PEs and carefully orchestrate flow of data between the PEs → achieve high throughput w/o increasing memory bandwidth requirements

- Differences from pipelining:
  - Array structure can be non-linear and multi-dimensional
  - PE connections can be multidirectional (and different speed)
  - PEs can have local memory and execute kernels (rather than a piece of the instruction)

Systolic Computation Example

- Convolution
  - Used in filtering, pattern matching, correlation, polynomial evaluation, etc ...
  - Many image processing tasks

Given the sequence of weights \( \{w_1, w_2, \ldots, w_k\} \) and the input sequence \( \{x_1, x_2, \ldots, x_n\} \), compute the result sequence \( \{y_1, y_2, \ldots, y_{n+k-1}\} \) defined by

\[
y_i = w_1x_i + w_2x_{i+1} + \cdots + w_kx_{i+k-1}
\]
Systolic Computation Example: Convolution

- \( y_1 = w_1x_1 + w_2x_2 + w_3x_3 \)
- \( y_2 = w_1x_2 + w_2x_3 + w_3x_4 \)
- \( y_3 = w_1x_3 + w_2x_4 + w_3x_5 \)

Figure 8. Design W1: systolic convolution array (a) and cell (b) where \( w_i \)'s stay and \( x_i \)'s and \( y_i \)'s move systolically in opposite directions.

Worthwhile to implement adder and multiplier separately to allow overlapping of add/mul executions.

More Programmability

- Each PE in a systolic array
  - Can store multiple "weights"
  - Weights can be selected on the fly
  - Eases implementation of, e.g., adaptive filtering

- Taken further
  - Each PE can have its own data and instruction memory
  - Data memory \( \rightarrow \) to store partial/temporary results, constants
  - Leads to stream processing, pipeline parallelism
    - More generally, staged execution

Pipeline Parallelism

Figure 1. (a) The code of a loop, (b) Each iteration is split into 3 pipeline stages: A, B, and C. Iteration i comprises A_i, B_i, C_i. (c) Sequential execution of 4 iterations. (d) Parallel execution of 6 iterations using pipeline parallelism on a three-core machine. Each stage executes on one core.
File Compression Example

![Diagram of file compression algorithm executed using pipeline parallelism]

Systolic Array

- **Advantages**
  - Makes multiple uses of each data item → reduced need for fetching/refetching
  - High concurrency
  - Regular design (both data and control flow)

- **Disadvantages**
  - Not good at exploiting irregular parallelism
  - Relatively special purpose → need software, programmer support to be a general purpose model

The WARP Computer

- HT Kung, CMU, 1984-1988
- Linear array of 10 cells, each cell a 10 Mflop programmable processor
- Attached to a general purpose host machine
- HLL and optimizing compiler to program the systolic array
- Used extensively to accelerate vision and robotics tasks

The WARP Computer

Models and Architectures

- In-order scalar Von-Neumann
- OoO scalar Von-Neumann
- SIMD
- Vector
- SPMD
- Static Dataflow
- Dynamic Dataflow
- Stream processing
- Systolic