**Von Neumann Execution Model**

Fetch:
- send PC to memory
- transfer instruction from memory to CPU
- increment PC

Decode & read ALU input sources

Execute
- an ALU operation
- memory operation
- branch target calculation

Store the result in a register or memory

Execution is comprised of a linear series of addressable instructions
- next instruction to be executed is pointed to by the PC
- send PC to memory
- next instruction to execute depends on what happened during the execution of the current instruction

Instruction operands reside in a centralized processor memory (GPRs)
Dataflow Execution Model

Instructions & initial input values are already in the processor:

Operands arrive from a producer instruction via a network

Check to see if all an instruction’s operands are there

Execute
  • an ALU operation
  • memory operation
  • branch target calculation

Send the result
  • to the consumer instructions or memory

Dataflow Execution Model

Execution is driven by the availability of input operands
  • operands are consumed
  • output is generated
  • no PC

Result operands are passed directly to consumer instructions
  • no register file
Promise of Dataflow Parallelism

Motivation:

- exploit instruction-level parallelism on a massive scale
- more fully utilize all processing elements

Believed this was possible if:

1. expose instruction-level parallelism by using a functional-style programming language
   - no side effects wrt generating new values
   - only restrictions were producer-consumer
2. scheduled code for execution on the hardware greedily
3. hardware support for data-driven execution
**Dataflow Execution**

All computation is **data-driven**.

- Binary is represented as a directed graph of data dependences
  - Nodes are operations executing in a logical processor
  - Values travel on arcs

```
+      
a \rightarrow b  
\downarrow 
\text{a+b}
```

- WaveScalar instruction

```
\text{opcode destination1 destination2}
```

Data-dependent operations are connected, producer to consumer

Code & initial values loaded into memory

Execute according to the **dataflow firing rule**

- When operands of an instruction have arrived on all input arcs, instruction may execute
- Value on input arcs is removed
- Computed value placed on output arc
Dataflow Example

\[ A[j + i \times i] = i; \]
\[ b = A[i \times j]; \]

Spring 2012  
CSE 471 - Dataflow Machines
Dataflow Example

\[ A[j + i*i] = i; \]
\[ b = A[i*j]; \]

Dataflow Execution

Control

- **steer (\(\rho\))**
  - value
  - T path
  - F path

- **merge (\(\phi\))**
  - T path value
  - F path value

- execute one path after the condition variable is known (steer)
- execute both paths & pass one set of values at the end (merge)
- convert control dependence to data dependence
WaveScalar Control

\[ \rho \text{ (steer)} \]

\[ \phi \text{ (merge)} \]

\[
\begin{align*}
\text{if } (A > 0) & : D = C + B; \\
\text{else} & : D = C - E; \\
F &= D + 1;
\end{align*}
\]

ISA for a Dataflow Computer

Instructions
- operation
- names of destination instructions

Data packets, called Tokens
- value
- tag to identify the operand & match it with its fellow operands in the same dynamic instruction
  - architecture dependent
    - instruction number
    - iteration number
    - activation/context number (for functions, especially recursive)
    - thread number
- Dataflow computer executes a program by receiving, matching tags, computing & sending out tokens.
Types of Dataflow Computers

**static:**
- one copy of each instruction
- no simultaneously active iterations, no recursion

**dynamic**
- multiple copies of each instruction
- better performance from increased ILP
- gate counting technique to prevent instruction explosion

**k-bounding**
- extra instruction with K tokens on its input arc; passes a token to 1st instruction of a loop iteration
- 1st instruction consumes a token (needs one extra operand to execute)
- last instruction in loop iteration produces another token at end of iteration
- limits active iterations to k
### Canonical Dataflow Computer

![Diagram of a canonical dataflow computer](image)

### Problems with Dataflow Computers

1. Memory ordering
   - dataflow cannot guarantee a correct ordering of memory operations

2. Language compatibility
   - dataflow computer programmers could not use mainstream programming languages, such as C
   - could not handle "complex" data structures
   - developed special languages in which order didn’t matter
Problems with Dataflow Computers

3. Scalability:
   - big token store
     - side-effect-free programming language with no mutable data structures
       - each update creates a new data structure
       - 1000 tokens for 1000 data items even if the same value
     - slow access
       - aggravated by the state of processor technology at the time
       - associative search impossible; accessed with slower hash function
       - delays in processing (only so many functional units, arbitration both for PEs and storing of result, long wires)

Dataflow Example

\[ A[j + i*i] = i; \]
\[ b = A[i*j]; \]
Example to Illustrate the Memory Ordering Problem

A[j + i*i] = i;

b = A[i*j];
Example to Illustrate the Memory Ordering Problem

\[ A[j + i*i] = i; \]
\[ b = A[i*j]; \]

Load-store ordering issue

Partial Solutions

Solutions led away from pure dataflow execution

Data representation in memory

- **I-structures:**
  - write once; read many times
  - early reads are deferred until the write

- **M-structures:**
  - multiple reads & writes, but they must alternate
  - reusable structures which could hold multiple values
Partial Solutions

Local (register) storage for back-to-back instructions

Frames within the token store for a sequence of instructions
  • example: each frame stores the data for one iteration or one thread
  • not have to search entire token store (use an offset to the frame)

Physically partition token store & place each partition with a PE
  • dataflow execution within coarse-grain threads