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
- from the ALU or memory

Program is 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

Operands reside in a centralized, global memory (GPRs)
Dataflow Execution Model

Instructions 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
Dataflow Computers

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

Believed this was possible if:
- expose instruction-level parallelism by using a functional-style programming language
  - no side effects; only restrictions were producer-consumer
- scheduled code for execution on the hardware greedily
- hardware support for data-driven execution

Dataflow Execution

All computation is **data-driven**.
- binary is represented as a directed graph
  - nodes are operations
  - values travel on arcs
- WaveScalar instruction

```
+   a     b
  \
  \     \
  \  a+b
  \    \
  \  opcode | destination1 | destination2
```
Dataflow Execution

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*i] = i;
b = A[i*j];
Dataflow Example

A[j + i*i] = i;
b = A[i*j];
### Dataflow Execution

**Control**
- **steer** ($\rho$)
- **merge** ($\phi$)

- Convert control dependence to data dependence with value-steering instructions
- Execute one path after condition variable is known (steer)
  - or
- Execute both paths & pass values at end (merge)

![Dataflow Execution Diagram]

### WaveScalar Control

**$\rho$ (steer)**

```plaintext
if (A > 0)
  D = C + B;
else
  D = C - E;
F = D + 1;
```

**$\phi$ (merge)**

```plaintext
>0 + -
```

![WaveScalar Control Diagram]
Dataflow Computer ISA

Instructions
- operation
- destination instructions

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

Types of Dataflow Computers

static:
- one copy of each instruction
- no simultaneously active iterations, no recursion
Types of Dataflow Computers

dynamic
• multiple copies of each instruction
• better performance
• 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 loop body
• 1st instruction of loop body consumes a token (needs one extra operand to execute)
• last instruction in loop body produces another token at end of iteration
• limits active iterations to k

Prototypical Early Dataflow Computer

Original implementations were centralized.

Performance cost
• large token store (long access)
• long wires
• arbitration both for PEs and storing of result
Problems with Dataflow Computers

Language compatibility
- dataflow cannot guarantee a correct ordering of memory operations
- dataflow computer programmers could not use mainstream programming languages, such as C
- developed special languages in which order didn’t matter

Scalability: large 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
  - aggravated by the state of processor technology at the time
    - delays in processing (only so many functional units, arbitration delays, etc.) meant delays in operand arrival
    - associative search impossible; accessed with slower hash function

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 of sequential instruction execution

- create “frames”, each of which stored the data for one iteration or one thread
- not have to search entire token store (offset to frame)
- like having dataflow execution among coarse-grain threads rather than instructions

Physically partition token store & place each partition with a PE