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

Von Neumann Execution Model

Program is a linear series of addressable instructions

- send PC to memory
- next instruction to execute depends on what happened during the execution of the current instruction

Next instruction to be executed is pointed to by the PC

Operands reside in a centralized, global memory (GPRs)

Dataflow Execution Model

Instructions are already in the processor:

Operands arrive from a producer instruction

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

Instruction-Level Parallelism (ILP)

Fine-grained parallelism

Obtained by:

- instruction overlap (later, as in a pipeline)
- executing instructions in parallel (later, with multiple instruction issue)

In contrast to:

- **loop-level** parallelism (medium-grained)
- **process-level** or **task-level** or **thread-level** parallelism (coarse-grained)

Instruction-Level Parallelism (ILP)

Can be exploited when instruction operands are **independent** of each other, for example,

- two instructions are independent if their operands are different
- an example of independent instructions

ld R1, 0(R2)
or R7, R3, R8

Each thread (program) has a fair amount of potential ILP

- very little can be exploited on today's computers
- researchers trying to increase it

Dependences

data dependence: arises from the flow of values through programs

- consumer instruction gets a value from a producer instruction
- determines the order in which instructions can be executed

```
ld R1, 32(R3)
add R3, R1, R8
```

name dependence: instructions use the same register but no flow of data between them

- **antidependence**
- **output dependence**

```
ld R1, 32(R3)
add R3, R1, R8
ld R1, 16 (R3)
```

Dependences

control dependence

- arises from the flow of control
- instructions after a branch depend on the value of the branch's condition variable

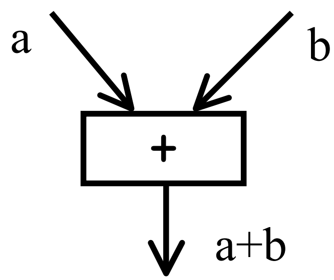
	`beqz R2, target`
	`lw r1, 0(r3)`
`target:`	`add r1, ...`

Dependences inhibit ILP

Dataflow Execution

All computation is **data-driven**.

- binary represented as a directed graph
  - nodes are operations
  - values travel on arcs



- WaveScalar instruction

`opcode`	`destination1`	`destination2`	...

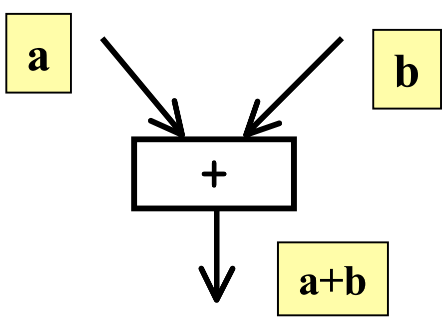
Dataflow Execution

Data-dependent operations are connected, producer to consumer

Code & initial values loaded into memory

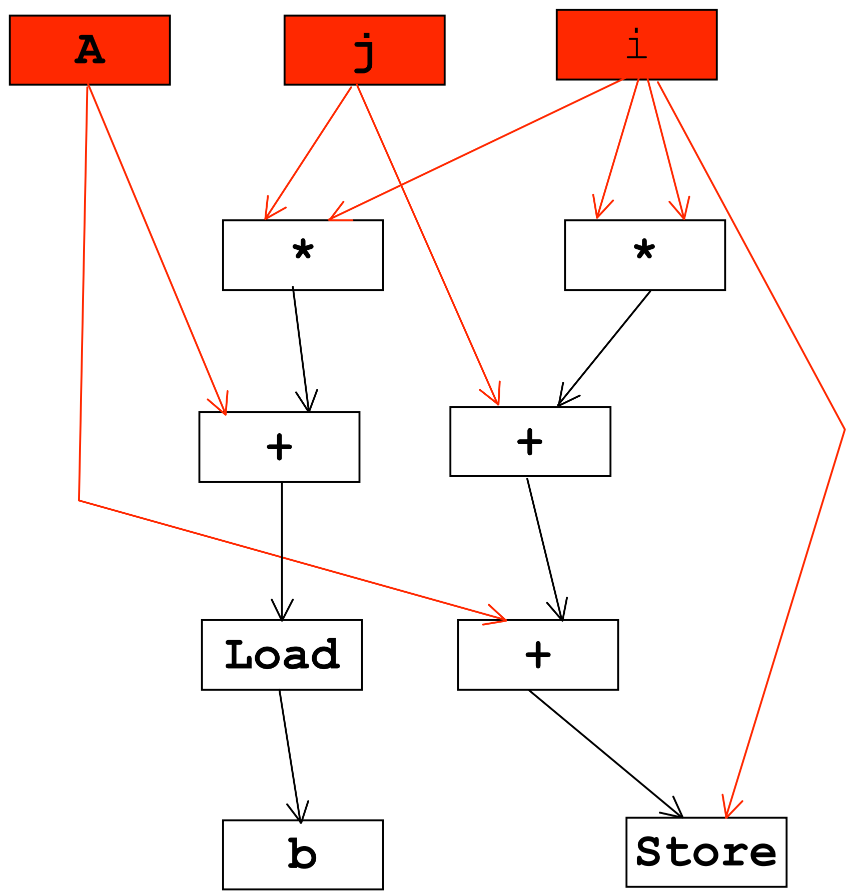
Execute according to the **dataflow firing rule**

- 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



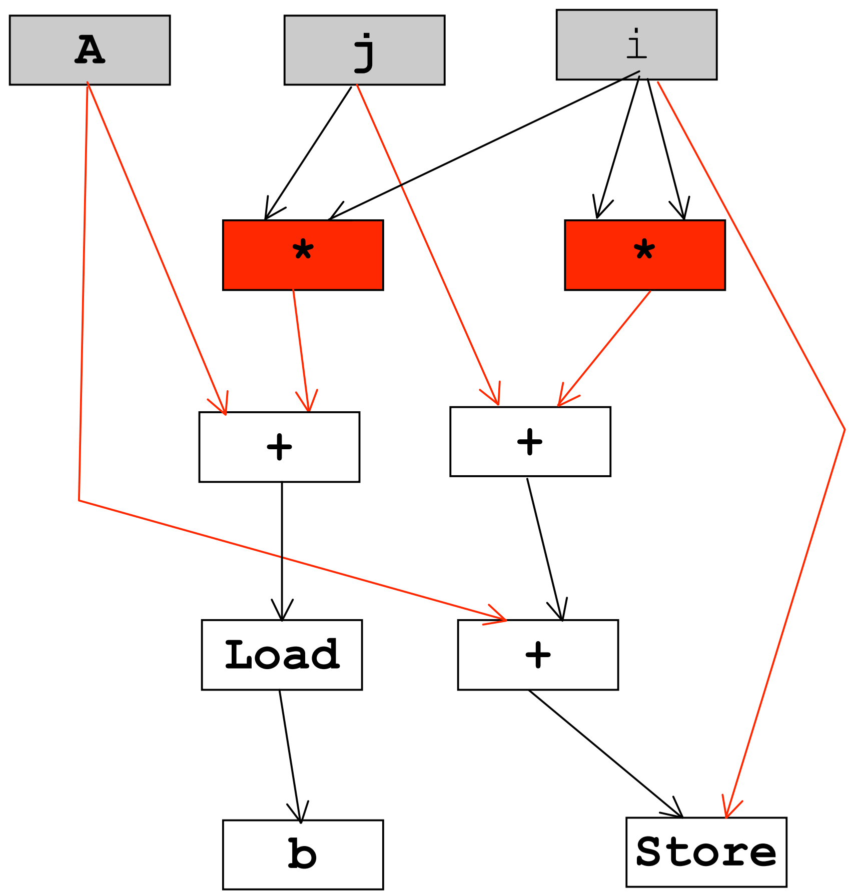
Dataflow Example

```
A[j + i*i] = i;

b = A[i*j];
```



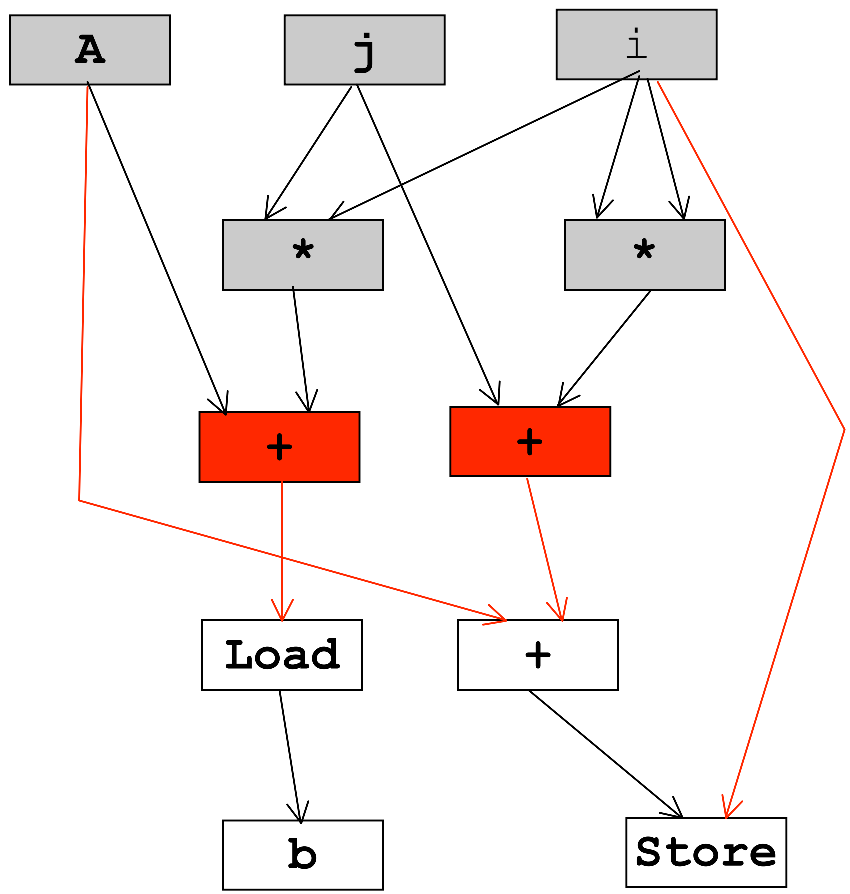
Dataflow Example

```
A[j + i*i] = i;

b = A[i*j];
```



Dataflow Example

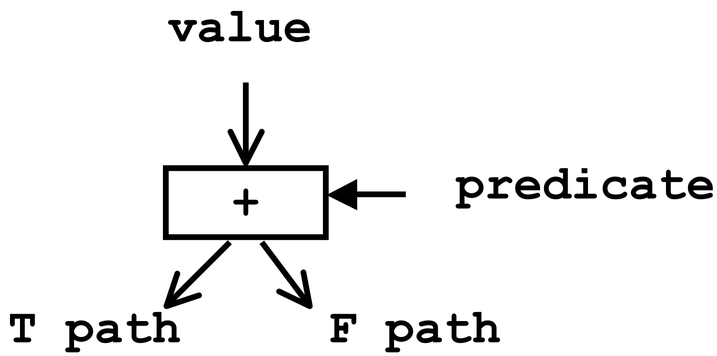
```
A[j + i*i] = i;

b = A[i*j];
```



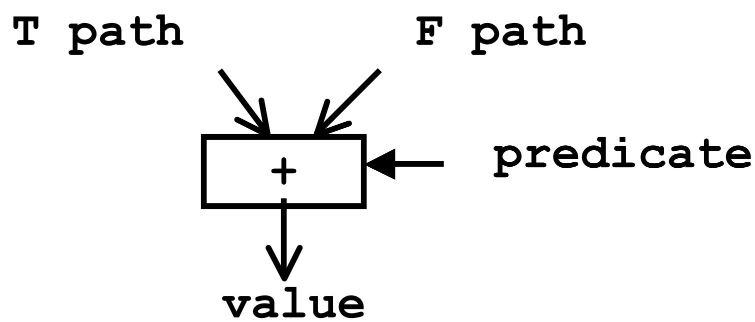
Dataflow Execution

Control

- Split (steer)



merge (ϕ)



- convert control dependence to data dependence with value-steering instructions
- execute one path after condition variable is known (split)

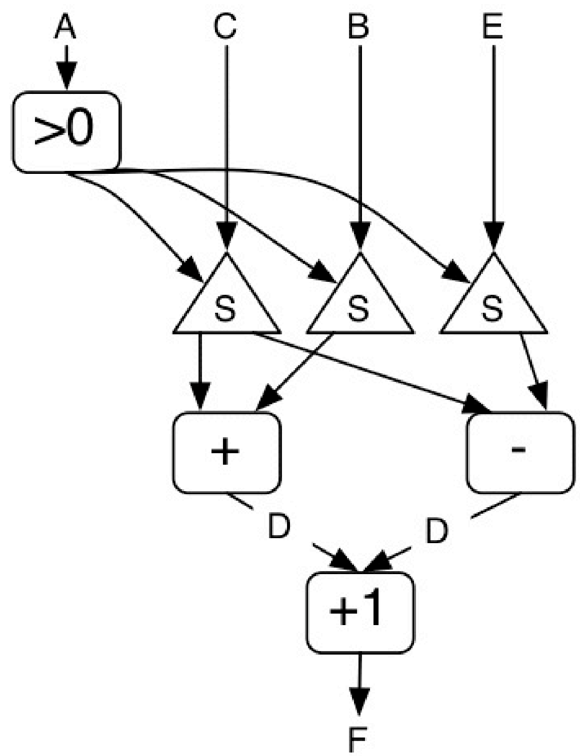
  or

- execute both paths & pass values at end (merge)

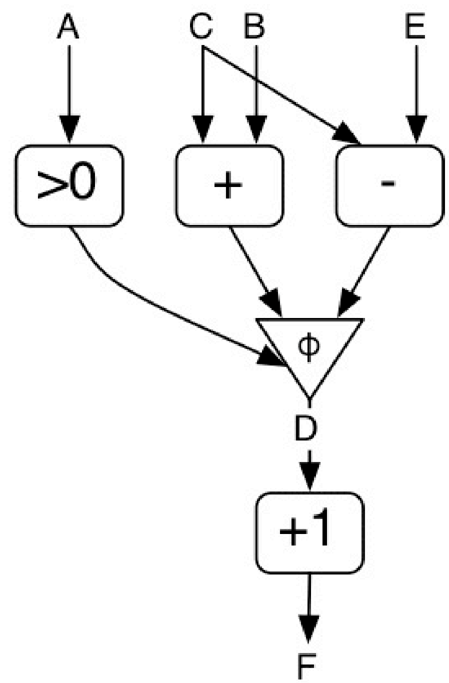
WaveScalar Control

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

steer



ϕ



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

dynamic

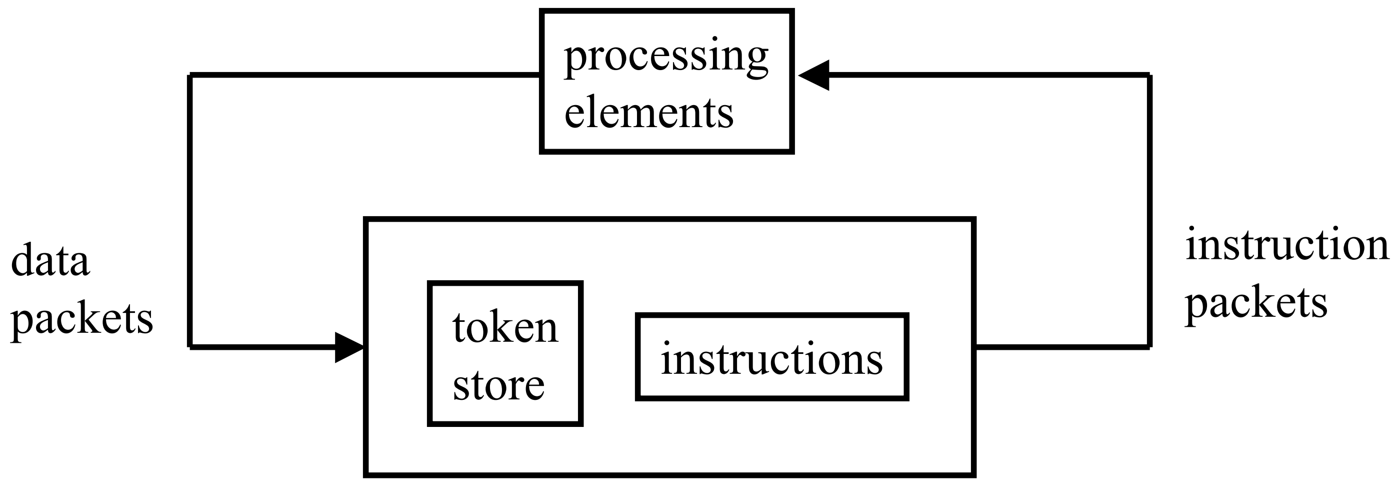
- multiple copies of each instruction
- better performance
- gate counting technique to prevent instruction explosion:

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 for PEs and return of result

Problems with Dataflow Computers

Language compatibility

- dataflow cannot guarantee a global 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
- associative search impossible; accessed with slower hash function
- aggravated by the state of processor technology at the time

More minor issues

- PE stalled for operand arrival
- Lack of operand locality

Partial Solutions

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

Local (register) storage for back-to-back instructions in a single thread

Cycle-level multithreading

Partial Solutions

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)
- dataflow execution among coarse-grain threads

Partition token store & place each partition with a PE

Many solutions led away from pure dataflow execution