Dataflow Architecture: Pure, Hybrid, and Spatial

15-740 FALL'19

NATHAN BECKMANN

Today

Historical tour of dataflow

Concepts & early architectures

“Hybrid dataflow” architectures

- Academic machines that outperform best from industry!

Modern “spatial dataflow” architectures

“Pure” Dataflow

Dataflow: The “Big Idea”

Program ⇔ Dataflow graph

Execution limited only by true dependences!

Implications:

- Massive parallelism ➔ Scales great!
- Massive storage ➔ Scales badly!
- No program counter ➔ Hard to program ?

```
input a, b
   y := (a+b)/x
   x := (a*(a+b))+b
output y, x
```

Example Dataflow Program

"A Preliminary Architecture for a Dataflow Processor" – J. Dennis (1975)

Dataflow building blocks

Figure 6. Links of the basic data-flow language.

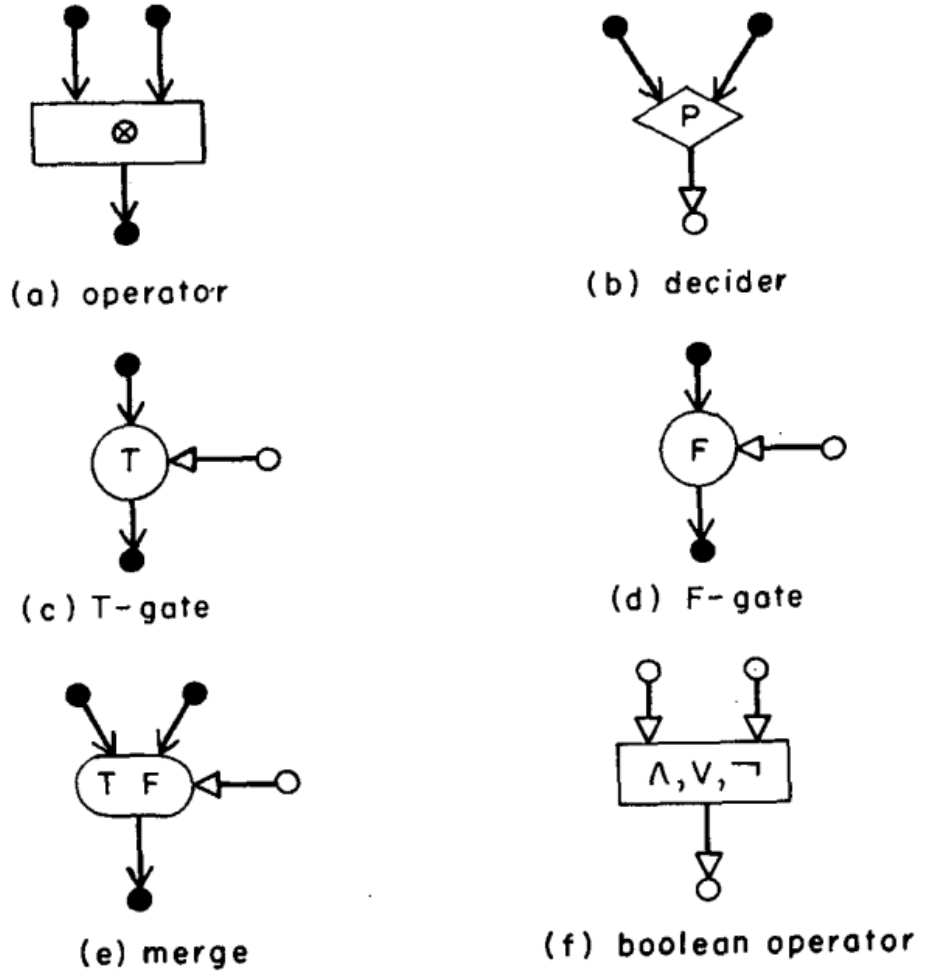


Figure 7. Actors of the basic data-flow language.

```
input y, x
n := 0
while y < x do
   y := y + x
   n := n + 1
   end
output y, n
```

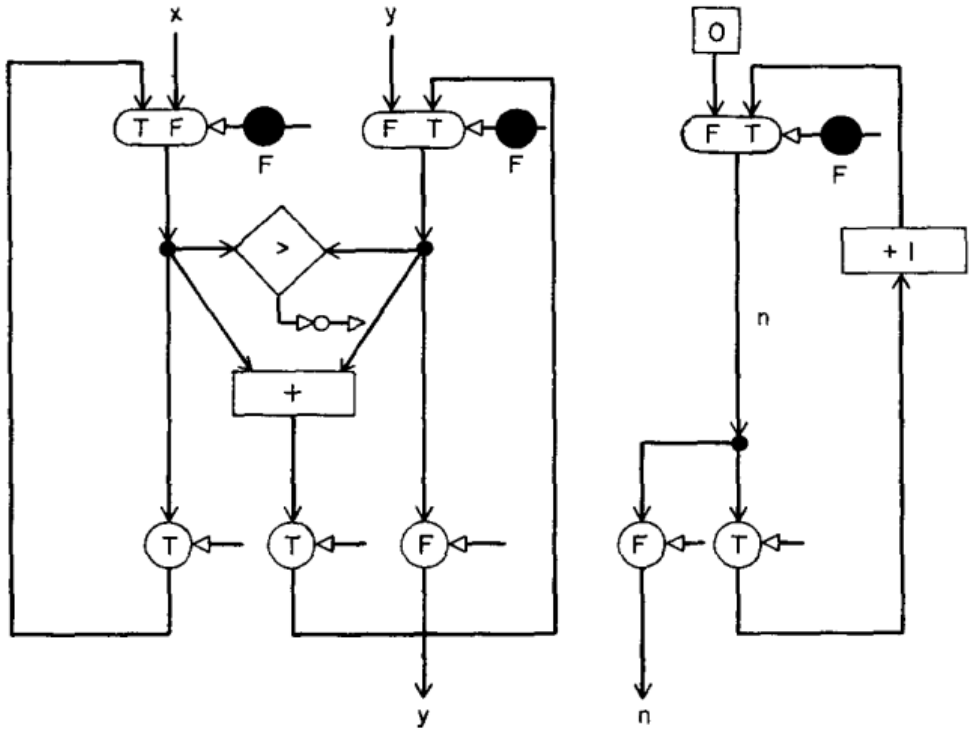


Figure 8. Data-flow representation of the basic program.

Example Dataflow Program w/ Control

[Dennis, ‘75]

Implementing dataflow

"Spatial" dataflow

- Directly implement DFG in hardware
- Instructions map onto an array of Pes
- Each PE has links to other Pes
- Problem? Limited scalability!

"Pure" dataflow

- Multiplex instructions onto small number of Pes
- PEs communicate through memory

Dennis's proposed architecture

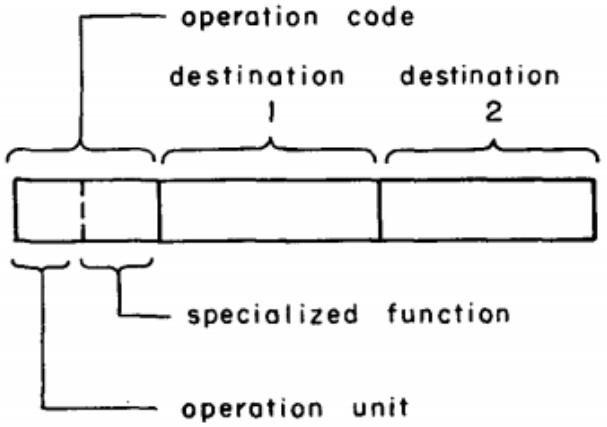


Figure 4. Instruction format.

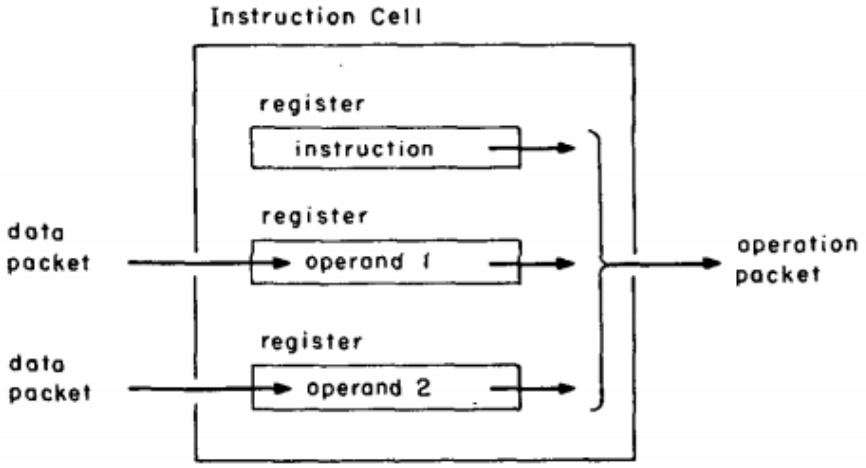


Figure 3. Operation of an Instruction Cell.

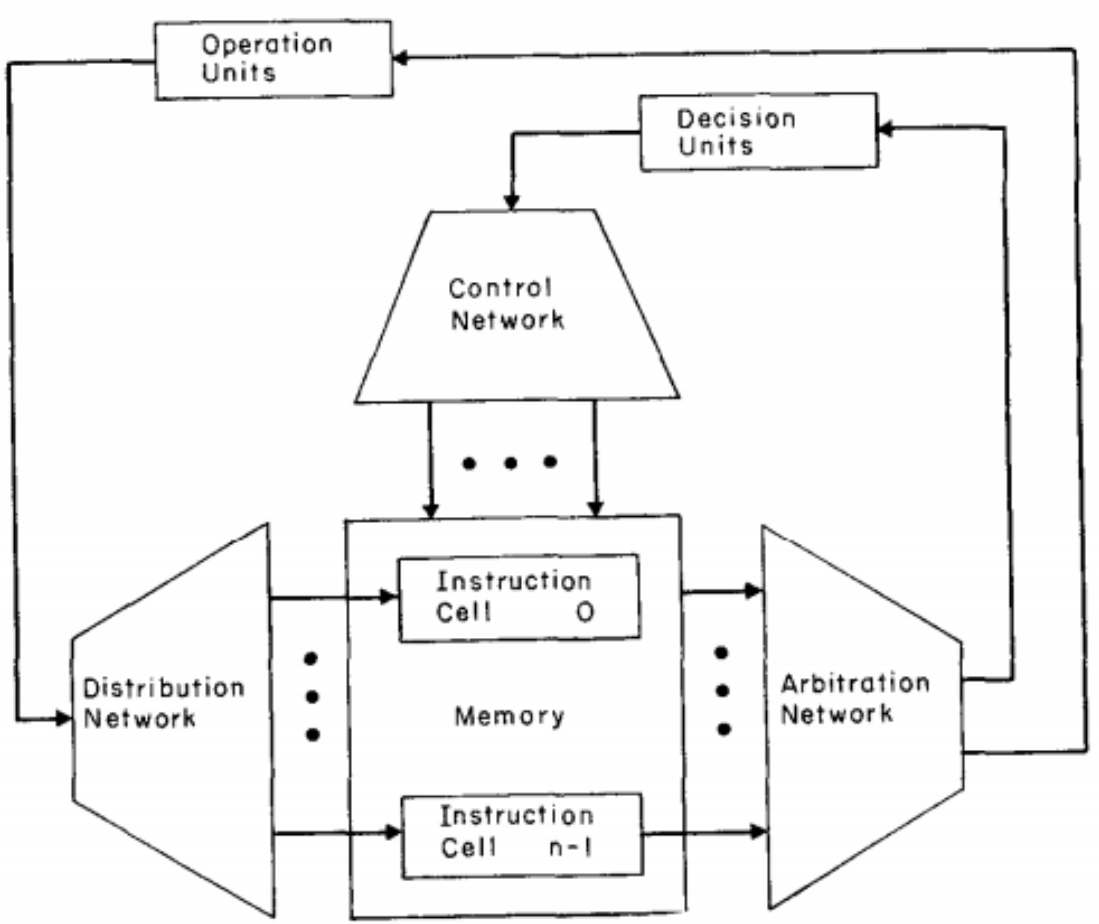


Figure 9. Organization of a basic data-flow processor without two-level memory.

Limitations of Dennis's dataflow processor

Communication through memory ➔ High power, high latency

How do you program it?

Instructions unique memory location ➔ No recursion!

“Executing a Program on the MIT Tagged-Token Dataflow Architecture” – Arvind & Nikhil (1990)

Id programming language

- Functional (seen as major disadvantage)
- Compiles to dynamic dataflow graphs
- Adds “I-structures” to deal with arrays

Tagged-Token Dataflow Architecture (TTDA) hardware

- Tokens (data values) are “tagged” with the instruction that generated them
- Allows recursion, parallelism across loops, etc

Id example

Adding vectors:

```
Def vsum A B =
    { C = array (1,n) ;
      {For j From 1 To n Do
        C[j] = A[j] + B[j]}
    In
      C } ;
```

Abundant parallelism:

- Across loop iterations
- Between *array allocation*, the *return statement* and *function execution* (!!!)
- E.g., one valid execution of vsum: Execute half the sums, then allocate the array, then return it, then finish the loop execution

Calling vsum: vsum e1 e2

- This function can return *before its inputs are ready*!

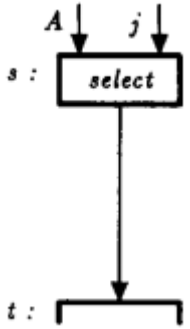
I-structure

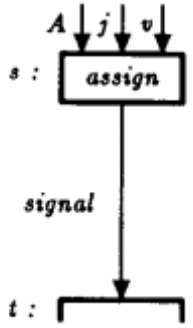
What does it mean to return an array before its full?

Array is an *I-structure*, a “tagged array”

- Each array element is written at most once
- Reads are delayed until the value is ready

Two new operators:





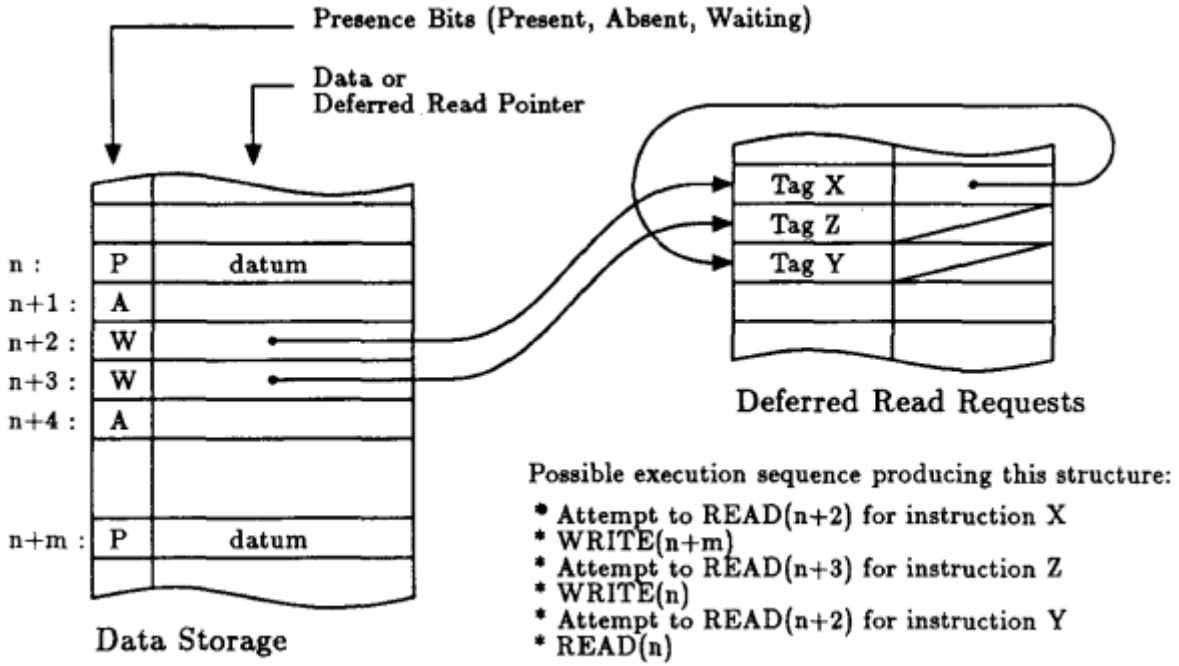


Fig. 7. *I*-structure memory.

I-structure example

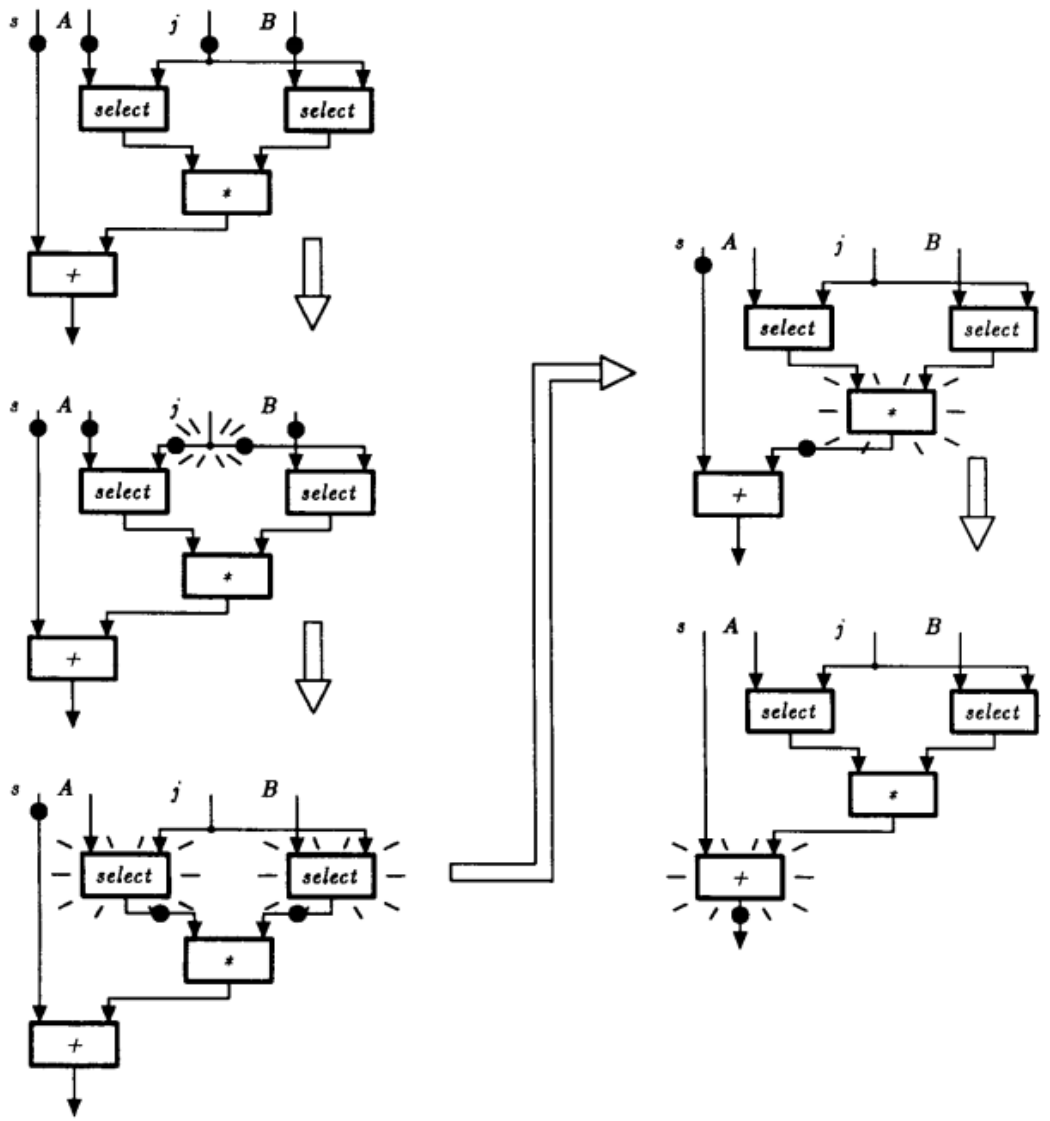


Fig. 3. A firing sequence for "$s+A[i]*B[i]$."

Dynamic dataflow graphs & tagged tokens

All tokens are tagged with a *context* specifying which function invocation they represent

Tokens: $\langle c.s,\ v\rangle_p$

- c – context
- s – destination instruction
- v – value

Output inherits its input context

$$\langle c.s,\ v1\rangle_l \times \langle c.s,\ v2\rangle_r \Rightarrow \langle c.t,\ (v1\ \text{op}\ v2)\rangle$$

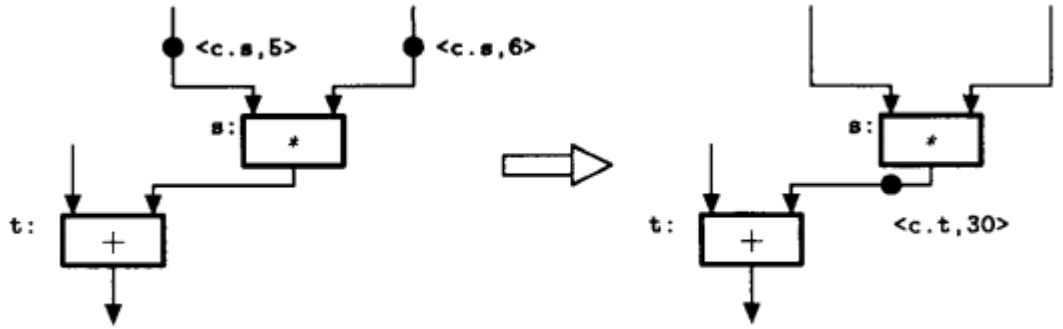


Fig. 5. Firing rule for "*" operator.

Dynamic dataflow graphs & tagged tokens – function linkage

Special operators required to manipulate tags when calling a function

$\text{extract\_tag}_r : \langle c.s, \_ \rangle \Rightarrow \langle c.t, c.r \rangle.$

- Construct return continuation

$\text{change\_tag}_j : \langle c.s, c'.t' \rangle_l \times \langle c.s, v \rangle_r \Rightarrow \langle c'.(t' + j), v \rangle_l.$

- Change a token's tag

$\text{get\_context} : \langle c.s, f \rangle \Rightarrow \langle c.t, \text{new\_}c.f \rangle.$

- Allocate a new context for callee

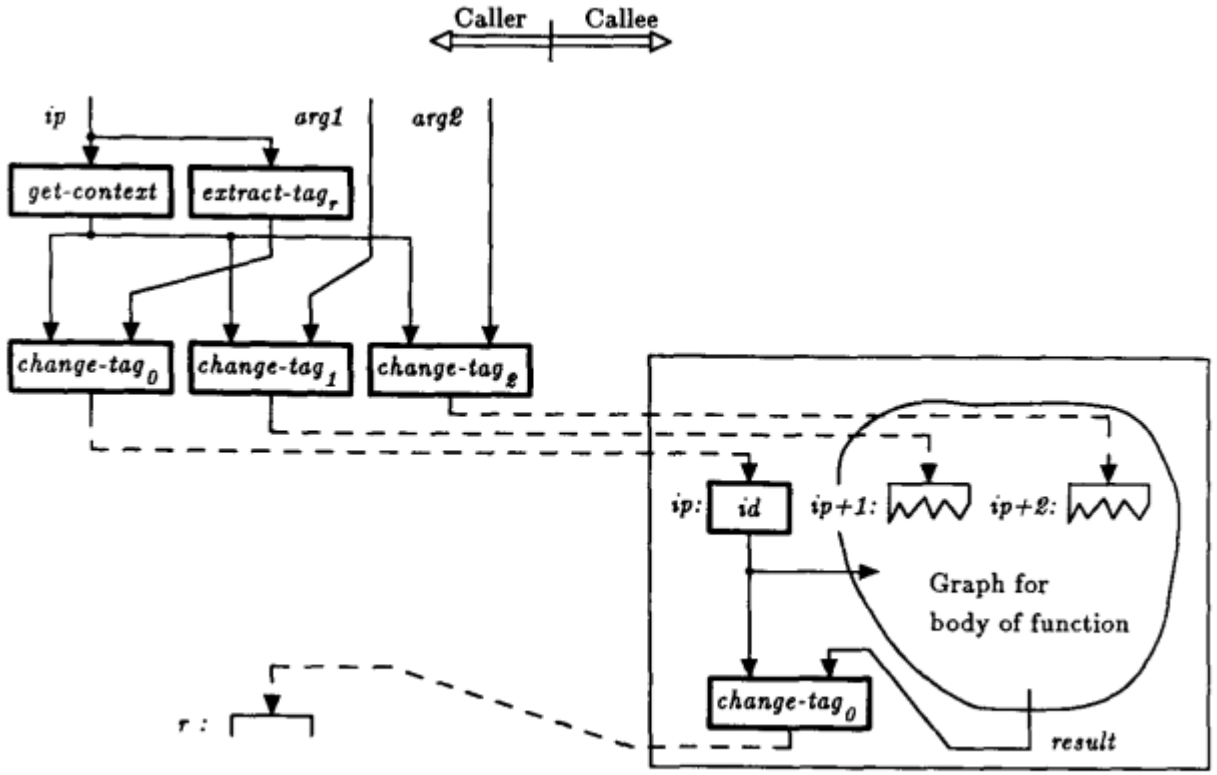


Fig. 6. Dataflow graph for function call and return linkage.

Dynamic dataflow graphs & tagged tokens – conditionals

```
Def ip A B = { s = 0 ;
               j = 1
             In
               {While (j <= n) Do
                    Next j = j + 1 ;
                    Next s = s + A[j] * B[j]
                Finally s }} ;
```

Similar approach to Dennis

"switch" operator controls what fires

Tags let many loop iterations execute in parallel!

- "D" operator allocates a new context for each iteration
- "D_reset" resets to original context for final result
- Simpler & faster than "get_context" implementation

They add loop throttling to *limit parallelism* (!!!)

- This is not trivial (deadlock if done naively, lowers perf)

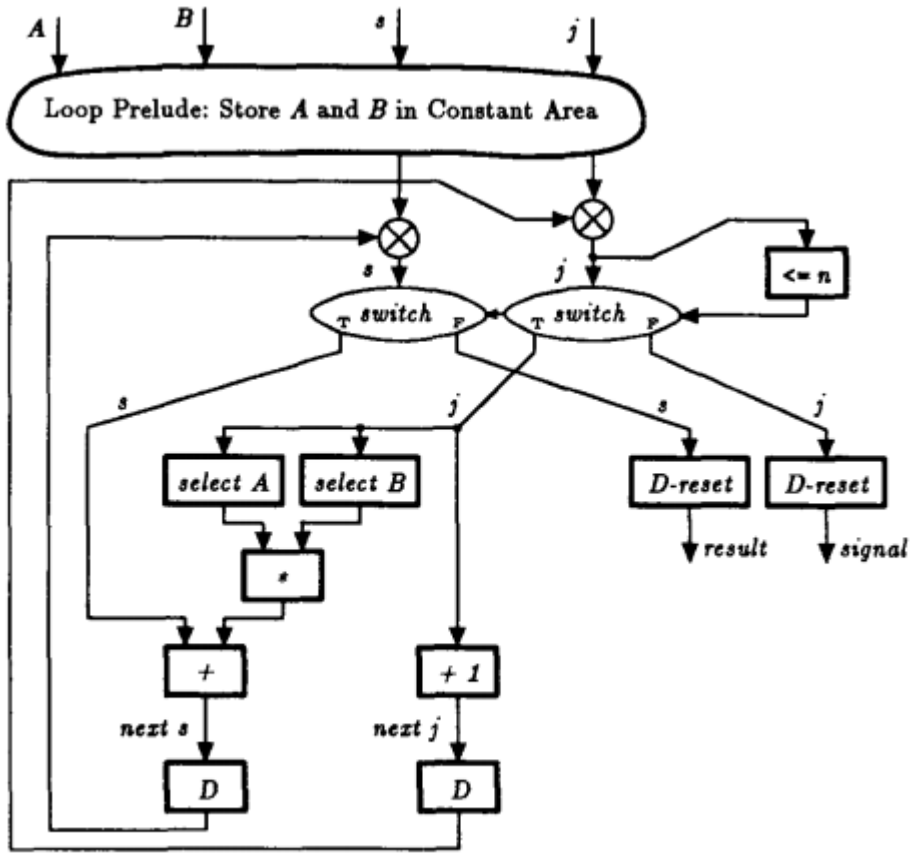


Fig. 13. Dataflow graph for a loop.

TTDA

"Except in special signal processing architectures, one should *never* think of the dataflow graph as representing physical wiring between function modules."

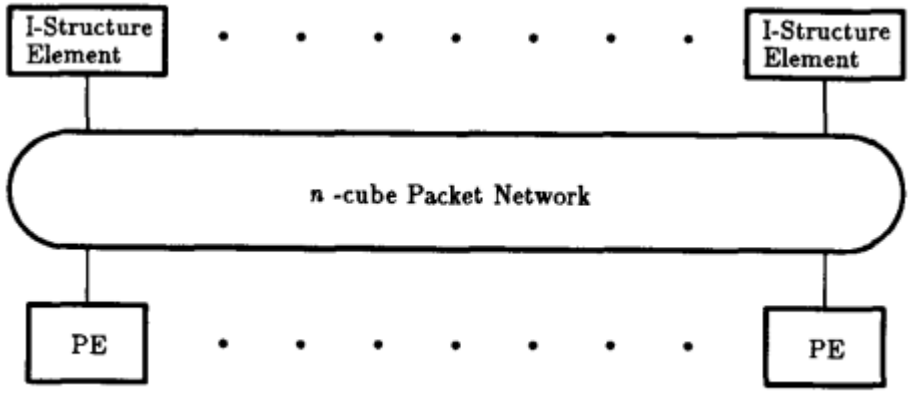


Fig. 17. Top-level view of TTDA.

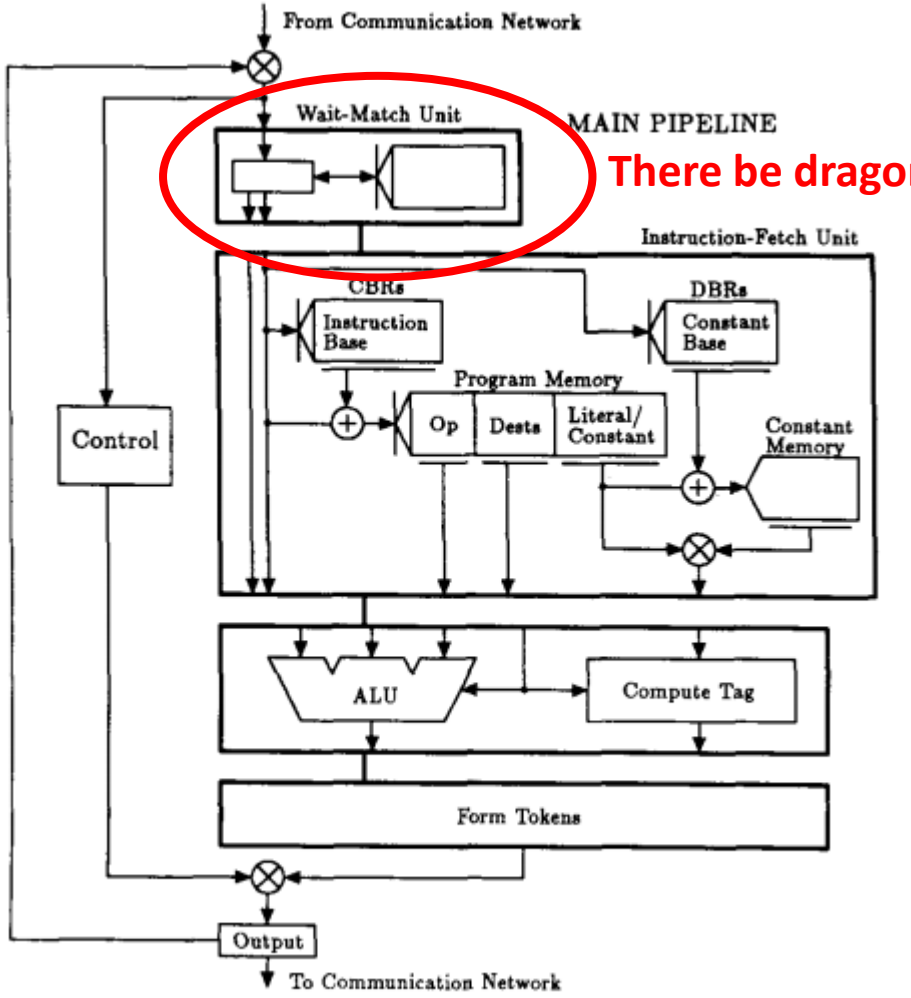


Fig. 18. A processing element.

Pure Dataflow Recap

ADVANTAGES

Elegance

Naturally parallel

- Avoids ILP limits of von Neumann designs

DISADVANTAGES

Hard to implement

- Too much parallelism!
- Tons of state to buffer
- Frequent associative lookups

Hard to program

- Unfamiliar functional programming languages
- No program counter – debugging & exceptions are challenging

Hybrid Dataflow

“Monsoon: an Explicit Token-Store Architecture” – Papadopoulos & Culler (1990)

Token store does not scale

Causes deadlock when it runs out of space

But most tokens are sent *locally* within a single function

Big Idea: Exploit this locality to improve scalability!

- Each function has its *activation frame* for token storage (storage + presence bits)
- Compiler instruction outputs onto static offsets within this frame

Programs consist of a tree (not stack) of activation frames

Significant *conceptual bridge* between dataflow + conventional architecture

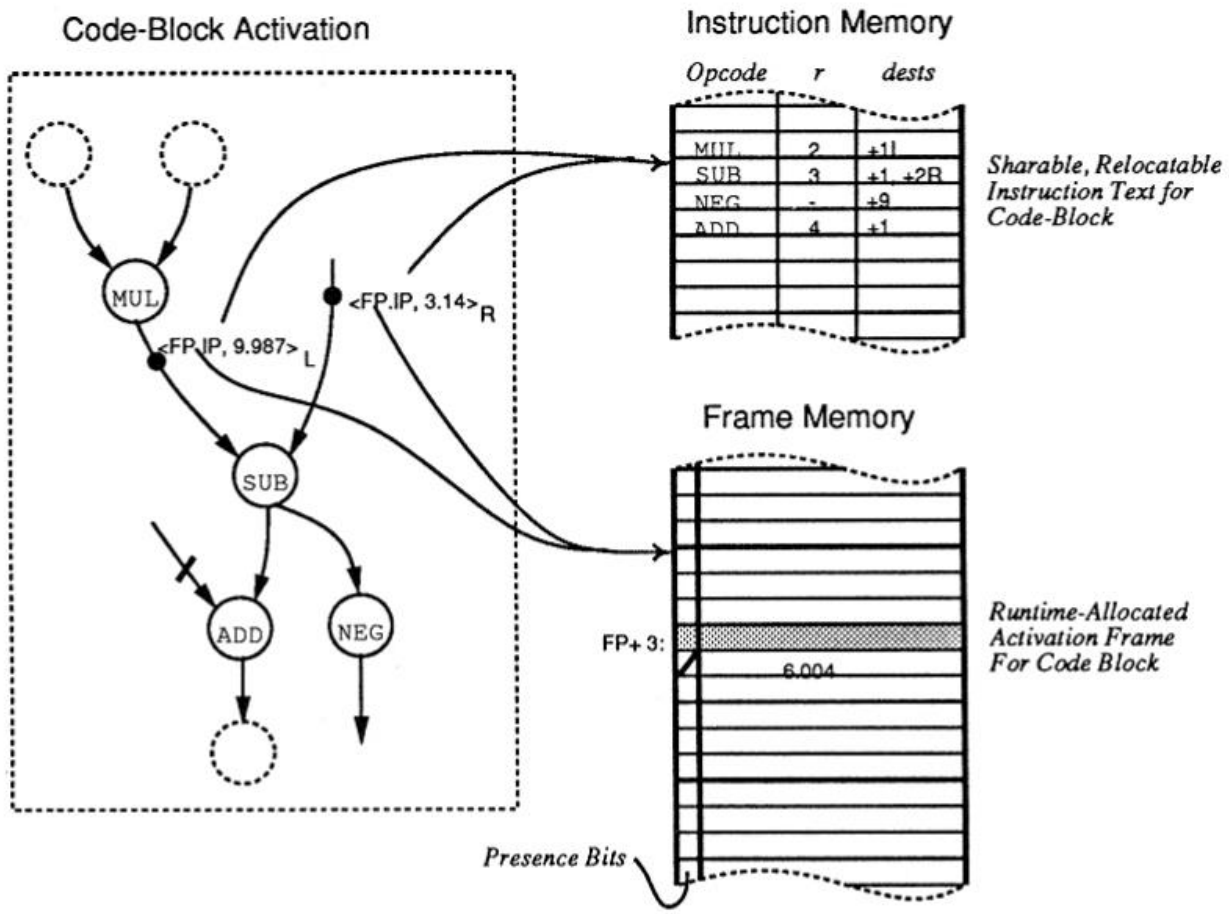
Monsoon example

Instructions are shared across invocations

Specify operations and destinations

Frame is unique to each invocation

Contains only values + presence bits



Monsoon hardware

(Read paper for details)

Built in collaboration with Motorola

Performed very well + was actually used by scientists for years afterward

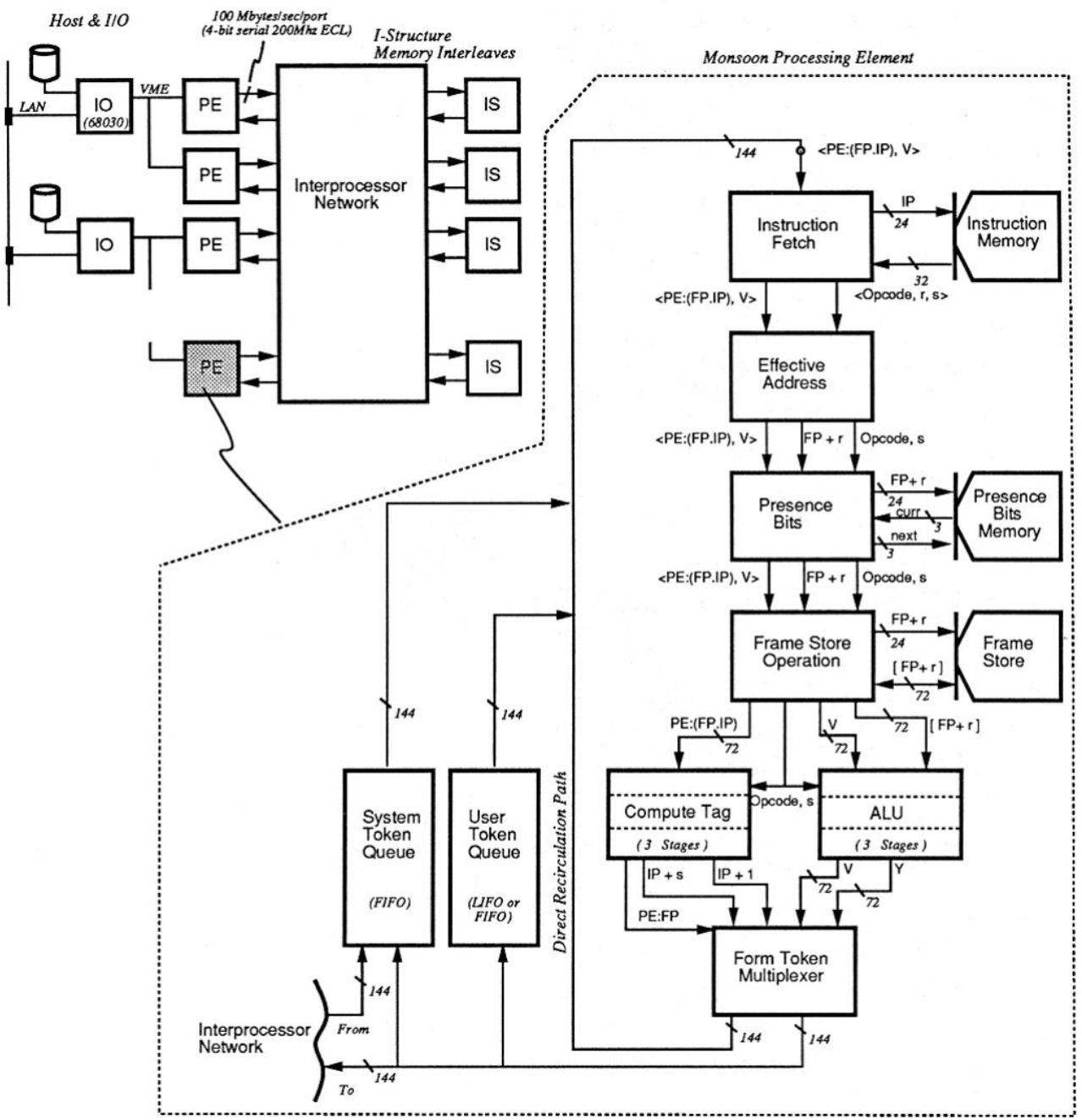


Figure 2: Monsoon Processing Element Pipeline

“WaveScalar” – Swanson et al (2003)

Began as an attempt to scale OOO processors

Similar to Monsoon, identifies **dataflow locality** as a major feature

- Most values soon after + nearby producing instruction
- Observation: OOO processors destroy dataflow locality (why?)

Dataflow machine with load-store architecture + sensible memory ordering

- No icky functional languages!

Tags tokens with a “wave number” & “WAVE-ADVANCE” operator

- ≈ contexts in TTDA

Memory ordering in WaveScalar

Compiler tags memory operations with a sequence number

Loads & stores form a chain representing control flow

- Notation: <prev, this, next>
- “?” means control flow makes prev/next ambiguous

Memory system then enforces ordering

- Only applies memory operation when there is no gap
- No “?”→”?” links allowed
  - “Memory-Nops” added to sequence legit operations (<3% ops)

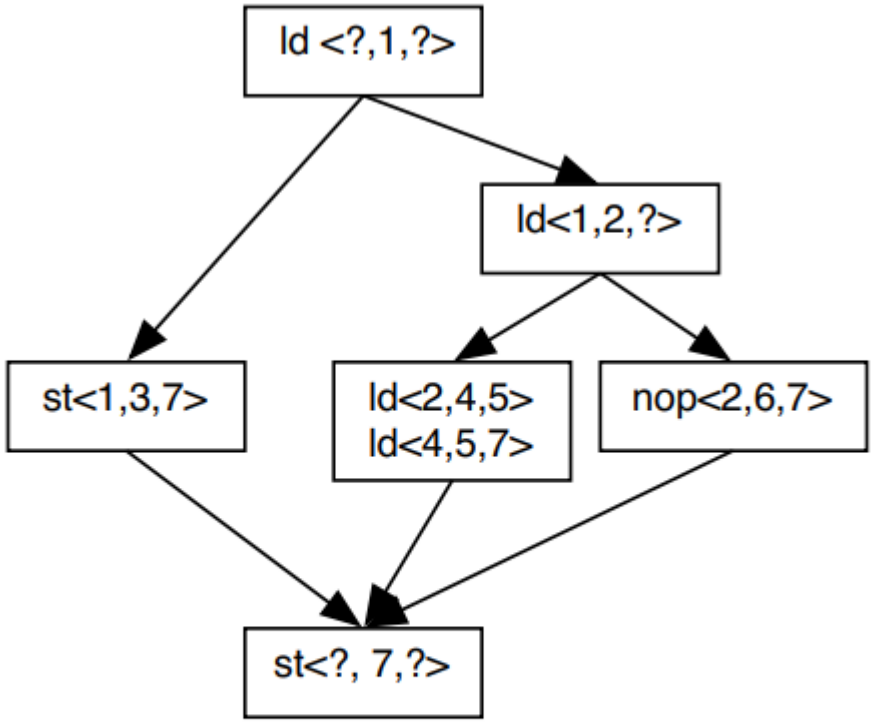


Figure 2: **Annotating memory operations.** A simple wave’s control flow graph showing the memory operations in each basic block and their links.

“WaveCache”

Instruction cache == processor

Instructions output is another instruction

Arrange instructions in an array of PEs

Cache instructions somewhere in the array

Route tokens to destination instruction in array

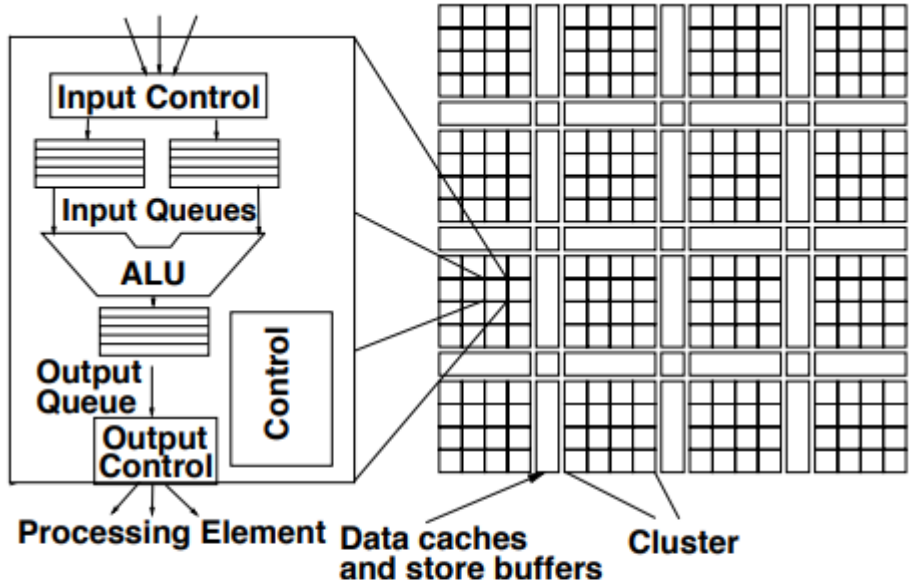


Figure 3: **A Simple WaveCache.** A simple architecture to execute the WaveScalar ISA. The WaveCache combines clusters of processing elements (left) with small data caches and store buffers to form a computing substrate (right).

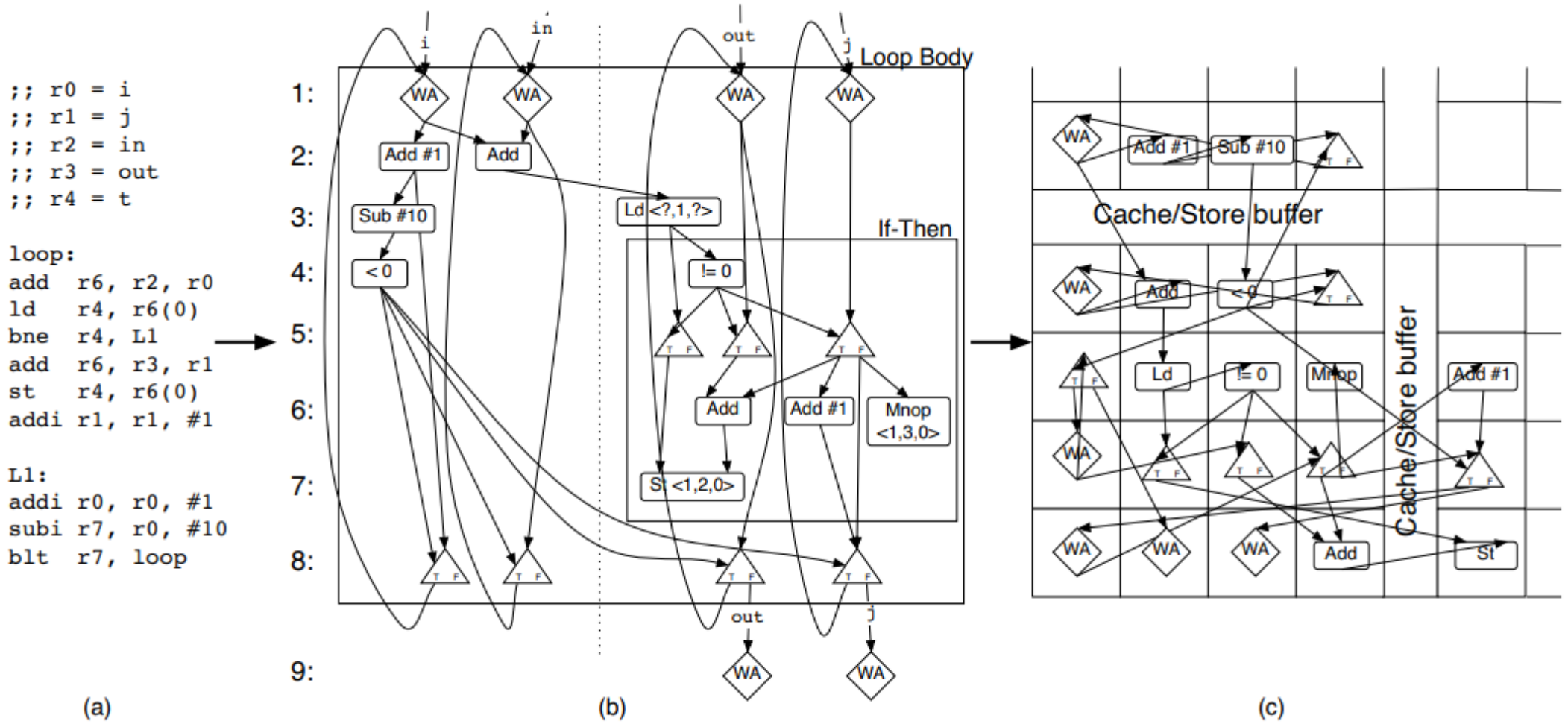


Figure 5: **WaveScalar example.** The RISC assembly (left) for the program fragment in Figure 4, the WaveScalar version (center), and the WaveScalar version mapped onto a small WaveCache (right). To clarify the discussion in the text, the numbers in the column label rows of instruction, and the vertical, dashed line divides the graph into two parts.

Spatial Dataflow

Spatial Network of Processing Elements

Idea: Directly wire operations together

- No buffering ➔ higher perf & efficiency

Reconfigure network & PEs to program device

Granularity is a major design choice

- How big is each PE
- How many PEs to have

- What are the tradeoffs?

Reconfigurable Logic

FPGA: Field programmable gate array

- Very fine-grain, bit-level reconfiguration (*too* configurable?)
- FUs are lookup tables (LUTs)
- Memory distributed in blocks (BRAMs)
- Originally used for fast ASIC prototyping, now a “general-purpose accelerator”
- FPGAs have been largely their own community for decades

CGRA: Coarse-grain reconfigurable array

- Sacrifice some configurability for efficiency
- Lots of recent architecture research
- Even FPGAs now have coarse-logic for efficiency (e.g., “DSP cores”)

TRIPS: “Scaling to the End of Silicon with EDGE Architectures” – Burger et al (2004)

(I highly recommend this magazine article … very approachable)

Intellectual background:

- Similar time as WaveScalar
- Sequential scaling was at its end
- *Polymorphism* would enable processors to target ILP, DLP, or TLP as available

TRIPS + EDGE architectures

EDGE: Explicit Data Graph Execution

TRIPS is a “VLIW hybrid-dataflow architecture”

- Instructions communicate directly to each other, no intermediate storage
- Compiler statically schedules instructions across an array of PEs
- Each spatial block is one “mega-instruction”, amortizing von Neumann overheads over lots of work

TRIPS Compilation Example

Each block emits exactly one branch (not shown) to determine next block to execute

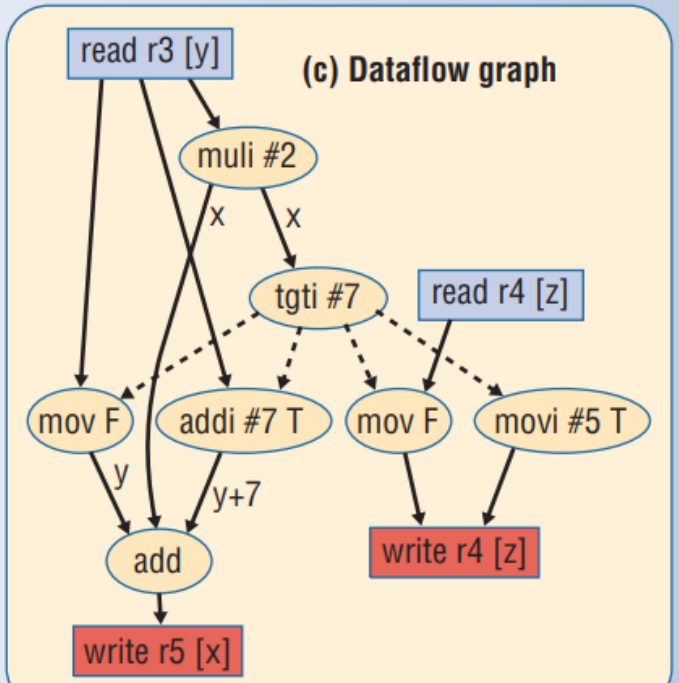
"tgti" is used to predicate execution of dependent instructions (shown as dashed lines)

(a) C code snippet

```
// y, z in registers

x = y*2;
of (x > 7){
   y + = 7;
   z = 5;
}
x + = y;

// x, z are live registers
```

(b) RISC assembly

```
// R0 contains 0
// R1 contains y
// R4 contains z
// R3 contains 7

muli R2, R1, 2          // x = y  * 2
ble R2, R3, L1          // if (x > 7)
addi R1, R1, #7         // y += 7
addi R4, R0, #5         // z = 5
L1: add R5, R2, R1      // x += y
```



(d) TRIPS instruction placement

Corresponding instruction positions:

[0,0,1]	[0,1,1]
[0,0,0]	[0,1,0]
[1,0,1]	[1,1,1]
[1,0,0]	[1,1,0]

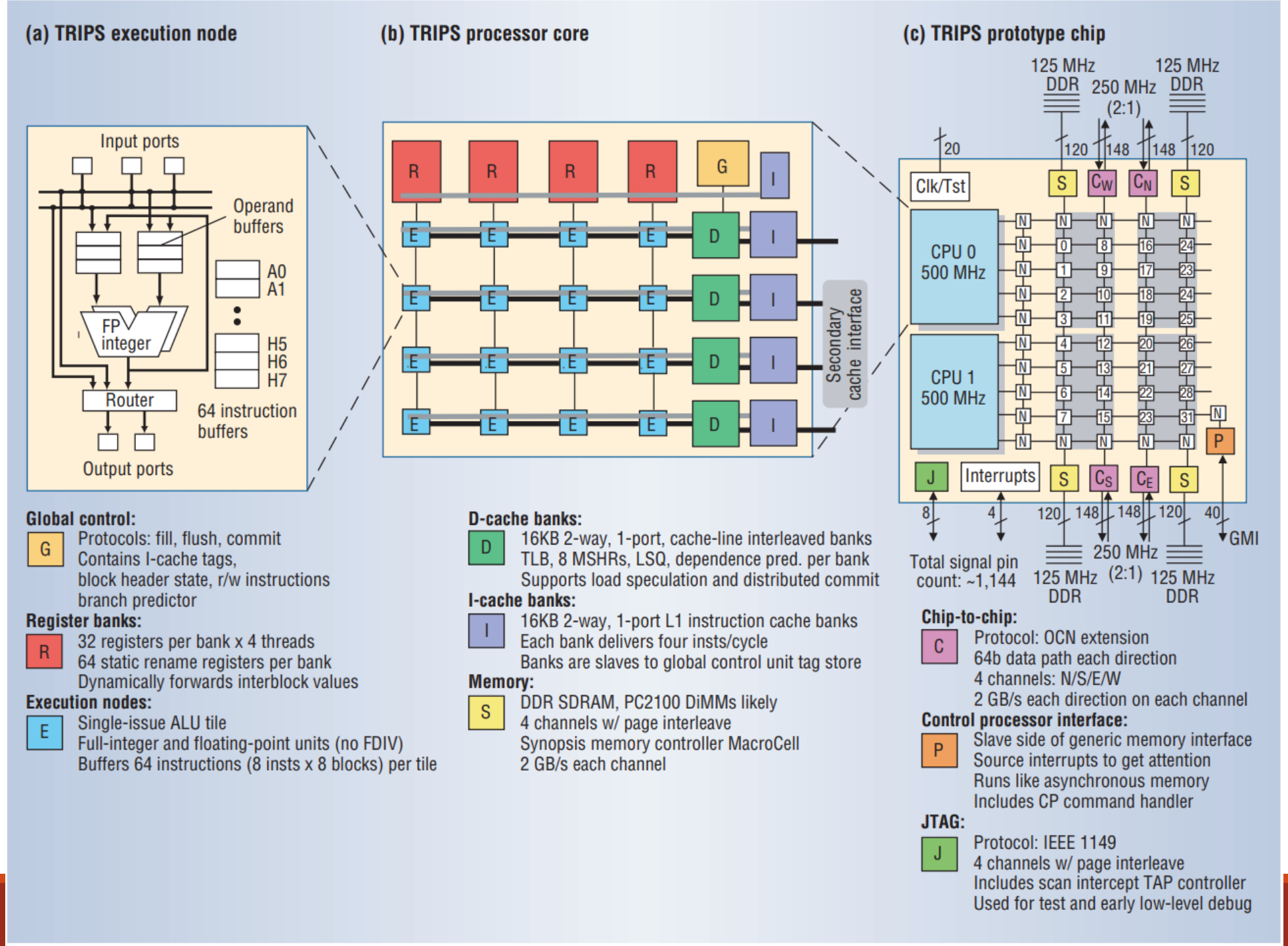
(e) TRIPS instruction block (2 x 2 x 2)

Block header

```
read r4, [1,0,0]          read r3, [0,1,1] [1,0,1]
w0: write r4              w1: write r5
```

Instruction block

```
add w1                    muli #2 [0,0,1] [0,1,0]
addi #7 [0,0,1]           tgti #7 [1,0,0] [0,0,0] [1,1,0] [1,1,1]

NOP                       movi #5 w0
mov [0,0,1]               mov w0
```


(a) TRIPS execution node
Input ports
Operand buffers
A0
A1
FP integer
H5
H6
H7
Router
64 instruction buffers
Output ports
(b) TRIPS processor core
R
G
I
E
D
Secondary cache interface
(c) TRIPS prototype chip
125 MHz DDR
250 MHz (2:1)
125 MHz DDR
20
120
148
148
120
Clk/Tst
S
CW
CN
S
CPU 0 500 MHz
CPU 1 500 MHz
N
P
J
Interrupts
CS
CE
8
4
40
GMI
Total signal pin count: ~1,144
250 MHz (2:1)
125 MHz DDR
125 MHz DDR
Global control:
G
Protocols: fill, flush, commit
Contains I-cache tags,
block header state, r/w instructions
branch predictor
Register banks:
R
32 registers per bank x 4 threads
64 static rename registers per bank
Dynamically forwards interblock values
Execution nodes:
E
Single-issue ALU tile
Full-integer and floating-point units (no FDIV)
Buffers 64 instructions (8 insts x 8 blocks) per tile
D-cache banks:
D
16KB 2-way, 1-port, cache-line interleaved banks
TLB, 8 MSHRs, LSQ, dependence pred. per bank
Supports load speculation and distributed commit
I-cache banks:
I
16KB 2-way, 1-port L1 instruction cache banks
Each bank delivers four insts/cycle
Banks are slaves to global control unit tag store
Memory:
S
DDR SDRAM, PC2100 DiMMs likely
4 channels w/ page interleave
Synopsis memory controller MacroCell
2 GB/s each channel
Chip-to-chip:
C
Protocol: OCN extension
64b data path each direction
4 channels: N/S/E/W
2 GB/s each direction on each channel
Control processor interface:
P
Slave side of generic memory interface
Source interrupts to get attention
Runs like asynchronous memory
Includes CP command handler
JTAG:
J
Protocol: IEEE 1149
4 channels w/ page interleave
Includes scan intercept TAP controller
Used for test and early low-level debug