Computer Architecture: Dataflow/Systolic Arrays

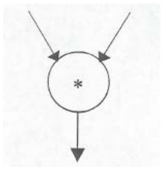
Prof. Onur Mutlu (editted by seth)
Carnegie Mellon University

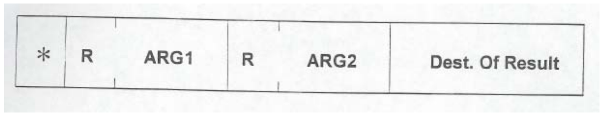
Data Flow

- 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: Obeys the control-flow model
  - Uarch level: A window of instructions executed in data-flow order → execute an instruction when its operands become available

Data Flow

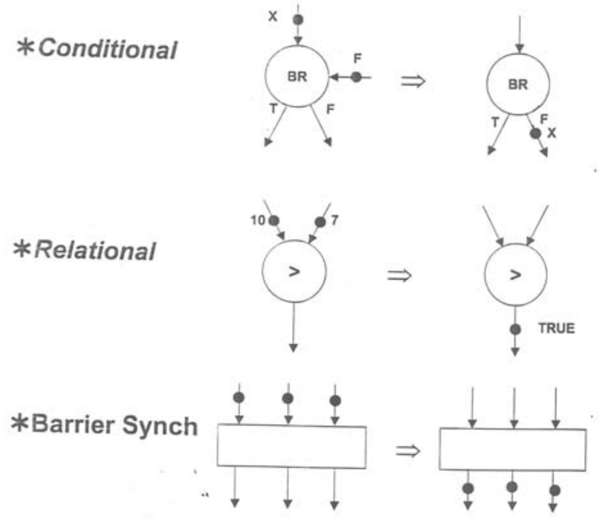
- In a data flow machine, a program consists of data flow nodes
- A data flow node fires (fetched and executed) when all its inputs are ready
  - i.e. when all inputs have tokens
- Data flow node and its ISA representation





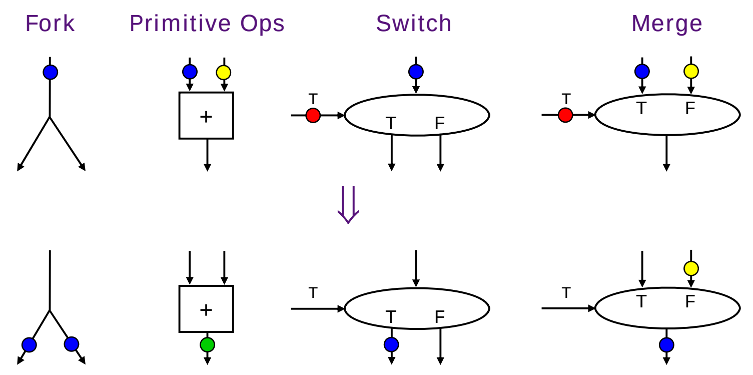
*	R	ARG1	R	ARG2	Dest. Of Result

Data Flow Nodes



Data Flow Nodes (II)

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



Dataflow Graphs

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

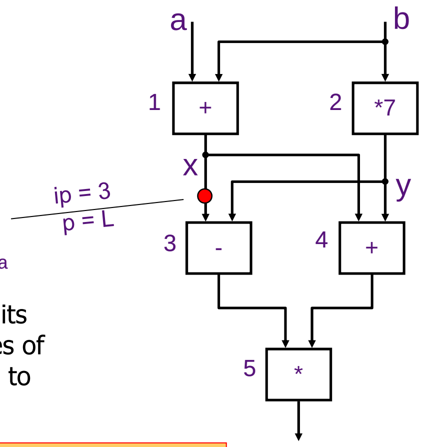
- Values in dataflow graphs are represented as tokens

token < ip , p , v >

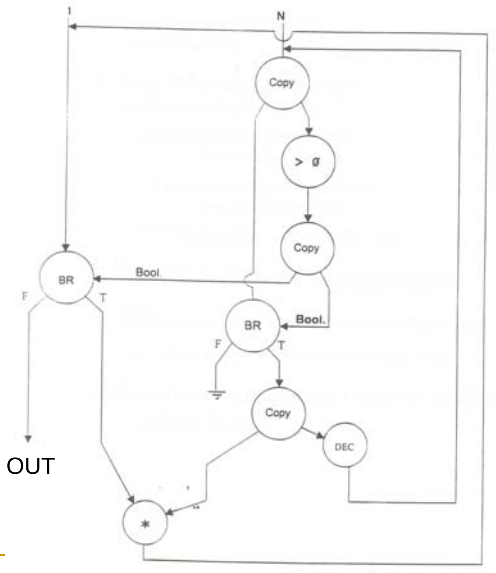
instruction ptr port data

- An operator executes when all its input tokens are present; copies of the result token are distributed to the destination operators

no separate control flow



Example Data Flow Program



Control Flow vs. Data Flow

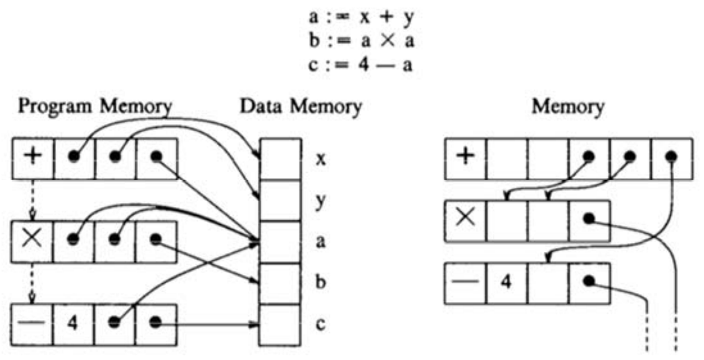


Figure 2. A comparison of control flow and dataflow programs. On the left a control flow program for a computer with memory-to-memory instructions. The arcs point to the locations of data that are to be used or created. Control flow arcs are indicated with dashed arrows; usually most of them are implicit. In the equivalent dataflow program on the right only one memory is involved. Each instruction contains pointers to all instructions that consume its results.

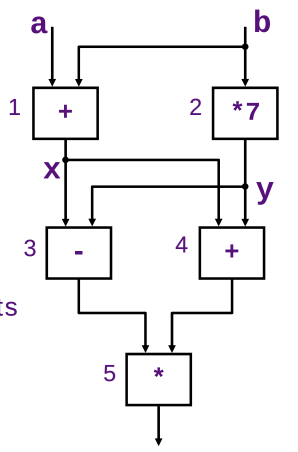
Static Dataflow

- 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 its output arcs
- Dennis and Misunas, "A Preliminary Architecture for a Basic Data Flow Processor," ISCA 1974.

Static Dataflow Machine:
Instruction Templates

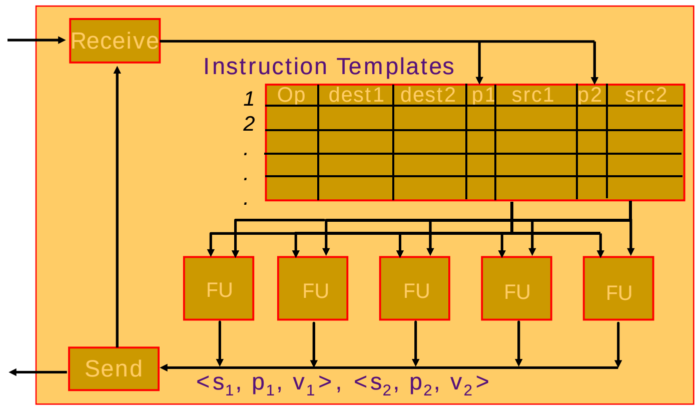
	Opcode	Destination 1	Destination 2	Operand 1	Operand 2
1	+	3L	4L		
2	*	3R	4R		
3	-	5L			
4	+	5R			
5	*	out			

Presence bits



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

Static Dataflow Machine (Dennis+, ISCA 1974)



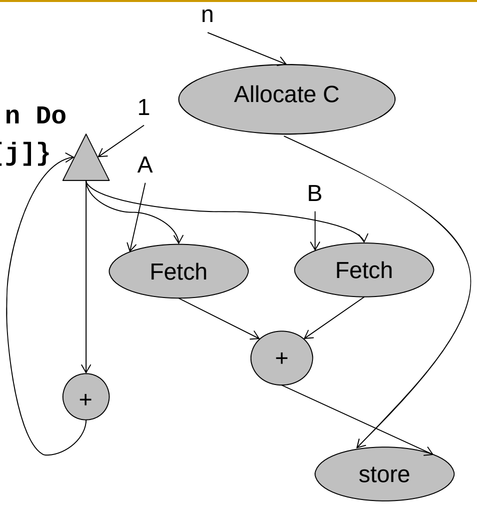
- Many such processors can be connected together
- Programs can be statically divided among the processors

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

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

a token <fp, ip, port, data>

frame pointer

instruction pointer

Static versus Dynamic Dataflow Machines

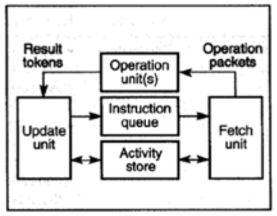


Figure 1. The basic organization of the static dataflow model.

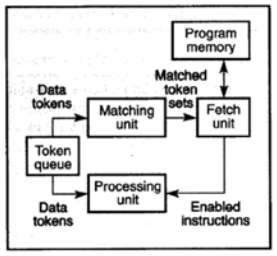
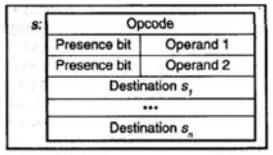
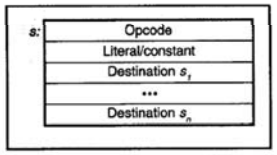


Figure 3. The general organization of the dynamic dataflow model.



s:	Opcode	
	Presence bit	Operand 1
	Presence bit	Operand 2
	Destination s_1	
	•••	
	Destination s_n	

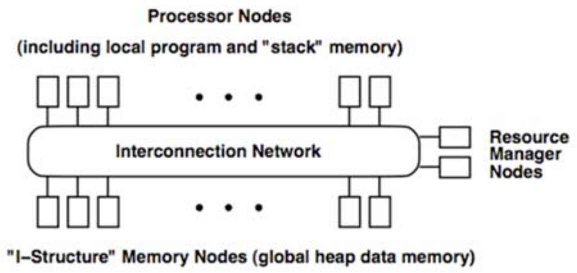
Figure 2. An instruction template for the static dataflow model.



s:	Opcode
	Literal/constant
	Destination s_1
	•••
	Destination s_n

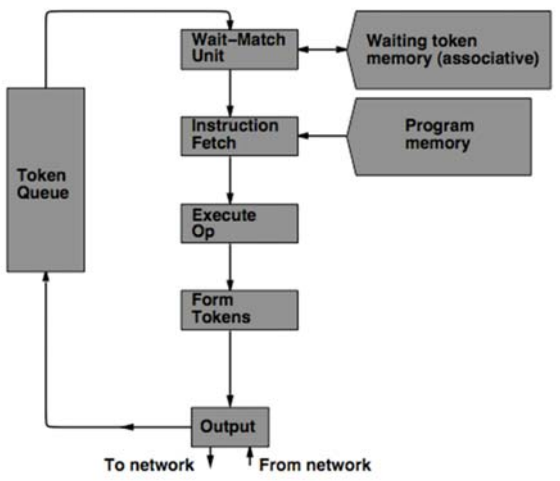
Figure 4. An instruction format for the dynamic dataflow model.

MIT Tagged Token Data Flow Architecture


Processor Nodes
(including local program and "stack" memory)
Interconnection Network
Resource Manager Nodes
"I–Structure" Memory Nodes (global heap data memory)

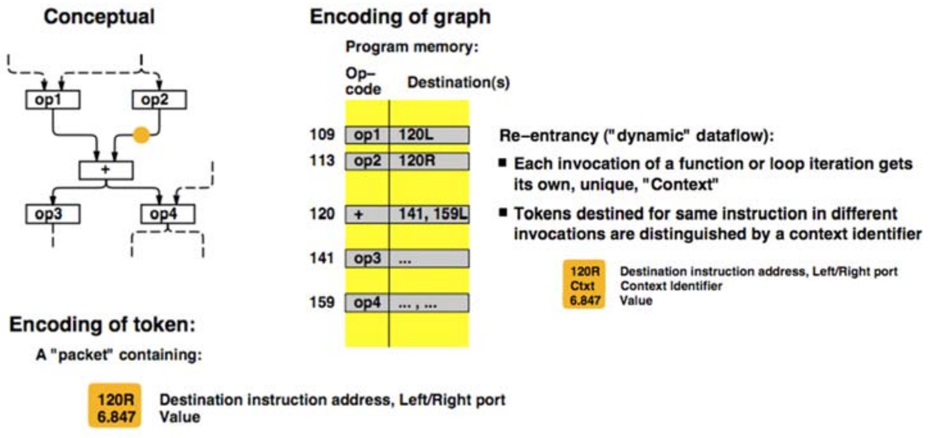
- Resource Manager Nodes
    - responsible for Function allocation (allocation of context/frame identifiers), Heap allocation, etc.

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 ––> Waiting Token Mem, bubble (no-op forwarded)

TTDA Data Flow Example



	Op-code	Destination(s)
109	op1	120L
113	op2	120R
120	+	141, 159L
141	op3	...
159	op4	... , ...

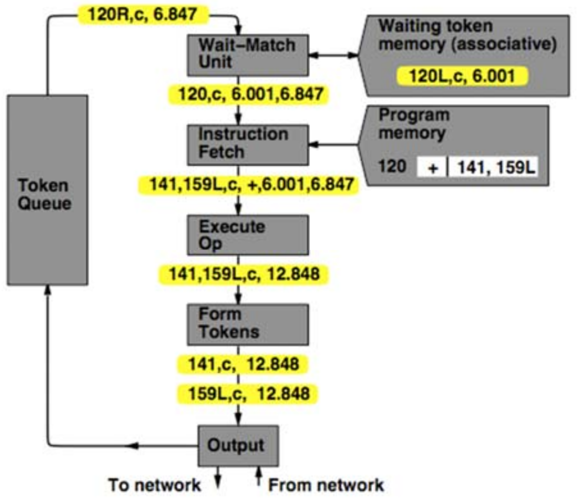
Re–entrancy ("dynamic" dataflow):
- Each invocation of a function or loop iteration gets its own, unique, "Context"
- Tokens destined for same instruction in different invocations are distinguished by a context identifier

Encoding of token:

A "packet" containing:

120R 6.847 — Destination instruction address, Left/Right port; Value

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

Concept of Tagging

- Each invocation receives a separate tag

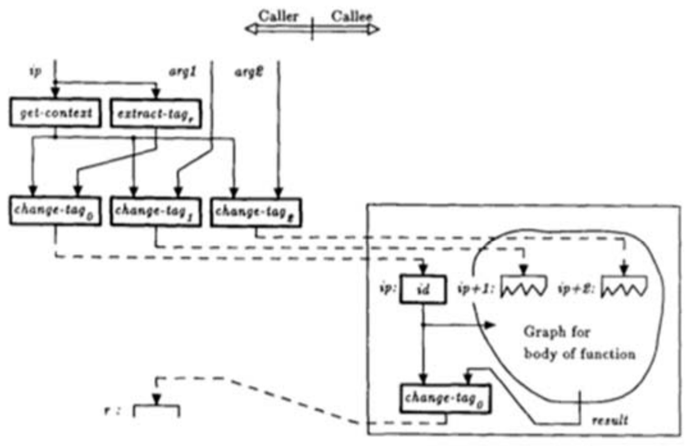
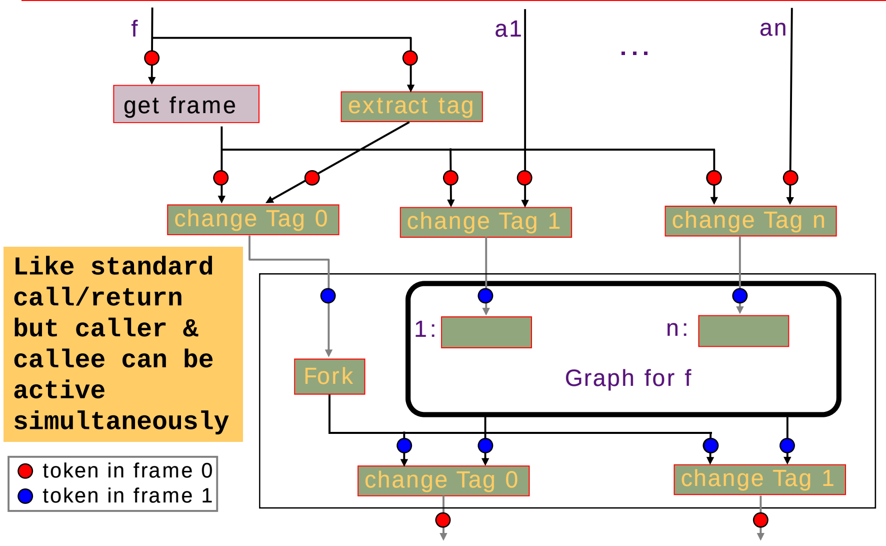


Fig. 6. Dataflow graph for function call and return linkage.

Procedure Linkage Operators



Like standard call/return but caller & callee can be active simultaneously

● token in frame 0
● token in frame 1

Loops and Function Calls Summary

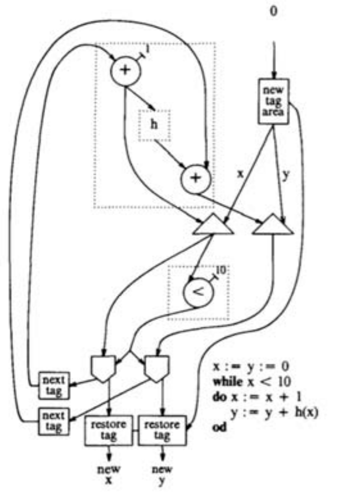


Figure 10. An implementation of a loop using tagged tokens. At the start of the loop a new tag area is allocated. Tokens belonging to consecutive iterations receive consecutive tags within this area. The tag from before the loop is restored on tokens that exit from the loop.

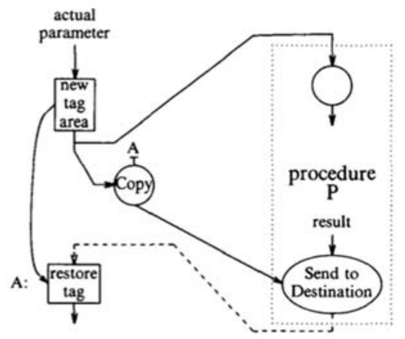


Figure 11. Interface for a procedure call. On the left a call of procedure P whose graph is on the right. P has one parameter and one return value. The actual parameter receives a new tag and is sent to the input node of P and concurrently a token containing address A is sent to the output node. This SEND-TO-DESTINATION node transmits the other input token to a node of which the address is contained in the first token. The effect is that, when the return value of the procedure becomes available, the output node sends the result to node A, which restores the tag belonging to the calling expression.

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

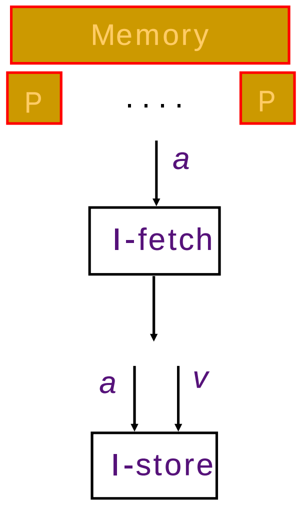
Data Structures in Dataflow

- 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

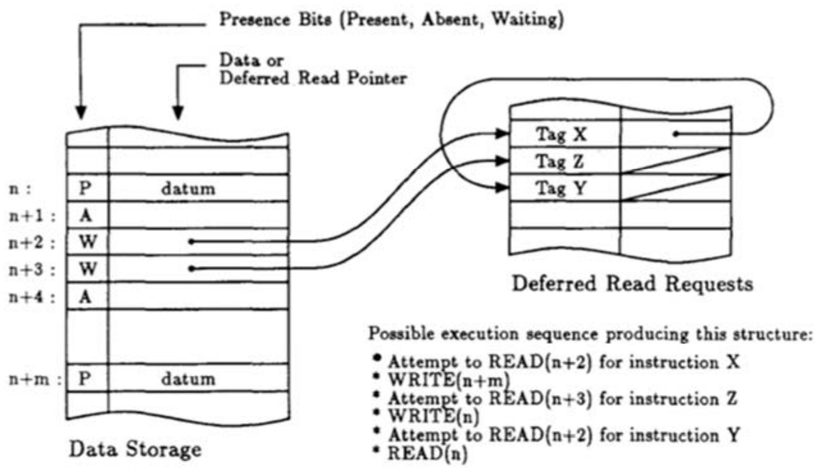

Presence Bits (Present, Absent, Waiting)
Data or Deferred Read Pointer
n :
n+1 :
n+2 :
n+3 :
n+4 :
n+m :
P
A
W
W
A
P
datum
datum
Tag X
Tag Z
Tag Y
Deferred Read Requests
Data Storage
Possible execution sequence producing this structure:
• Attempt to READ(n+2) for instruction X
• WRITE(n+m)
• Attempt to READ(n+3) for instruction Z
• WRITE(n)
• Attempt to READ(n+2) for instruction Y
• READ(n)

Fig. 7. *I*-structure memory.

Dynamic Data Structures

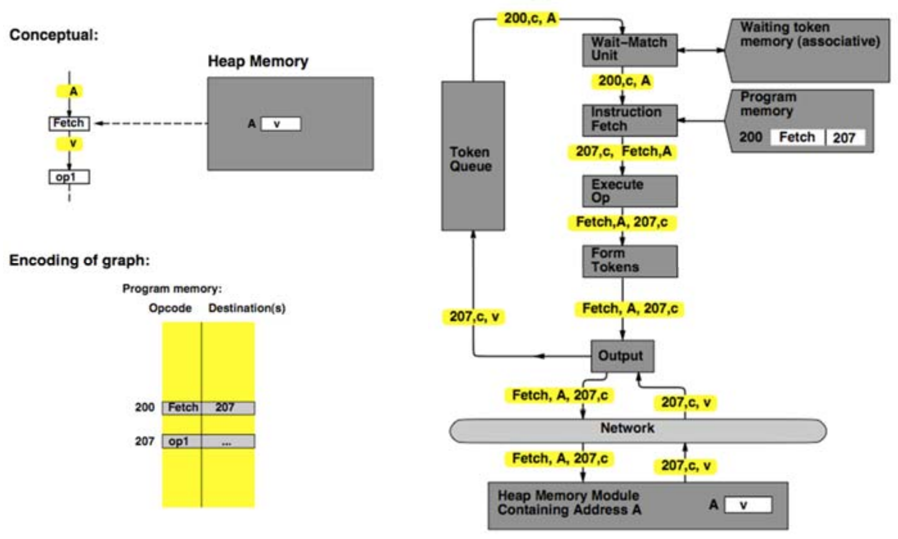
- Write-multiple-times data structures
- How can you support them in a dataflow machine?
  - M-Structures
  - Can you implement a linked list?

- What are the ordering semantics for writes and reads?

- Imperative vs. functional languages
  - Side effects and mutable state

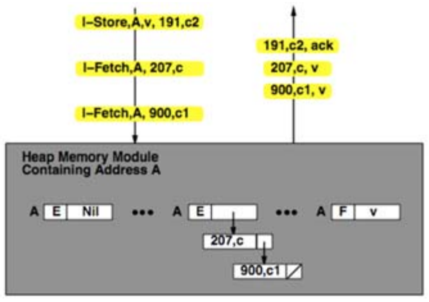
    vs.
  - No side effects and no mutable state

TTDA Data Flow Example

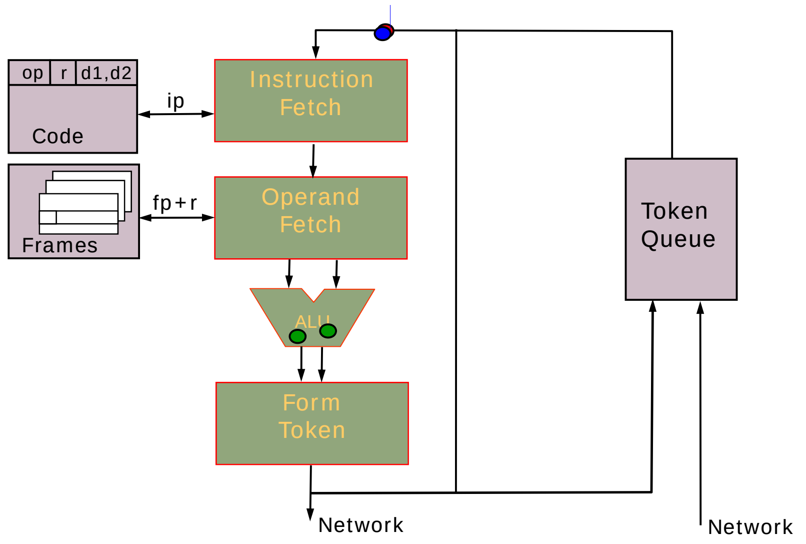


TTDA Synchronization

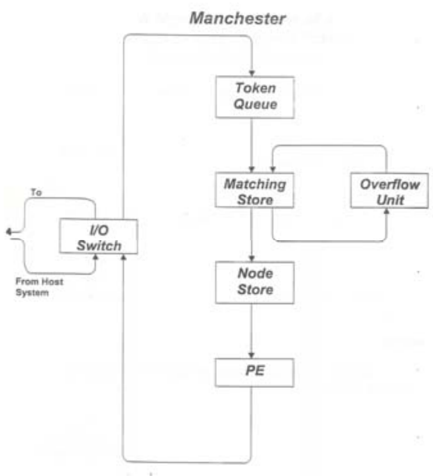
- Heap memory locations have FULL/EMPTY bits
- if the heap location is EMPTY, heap memory module queues request at that location When "I–Fetch" request arrives (instead of "Fetch"),
- Later, when "I–Store" arrives, pending requests are discharged
- No busy waiting
- No extra messages



Monsoon Processor (ISCA 1990)



Manchester Data Flow Machine

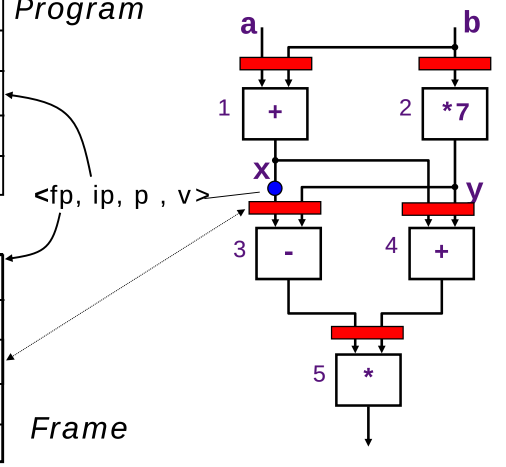


- Matching Store: Pairs together tokens destined for the same instruction
- Large data set → overflow in overflow unit
- Paired tokens fetch the appropriate instruction from the node store

A Frame in Dynamic Dataflow

Program

1	+	*1*	3L, 4L
2	*	*2*	3R, 4R
3	-	*3*	5L
4	+	*4*	5R
5	*	*5*	out



Frame

1		
2		
3	L	7
4		
5		

Need to provide storage for only one operand/operator

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

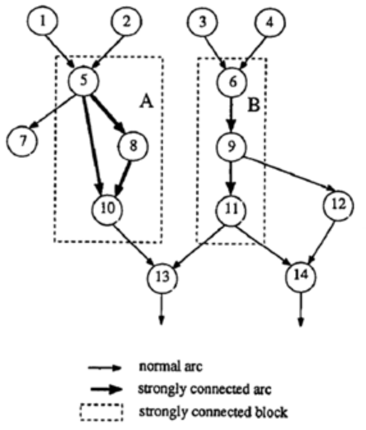


Figure 1 An Example of a Strongly Connected Block.

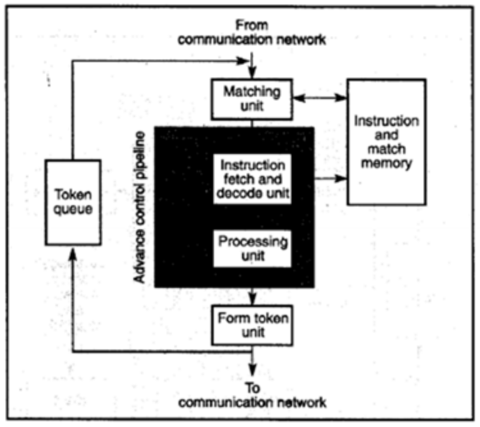


Figure 7. Organization of a macro-dataflow processing element.

Sakai et al., "An Architecture of a Dataflow Single Chip Processor," ISCA 1989.

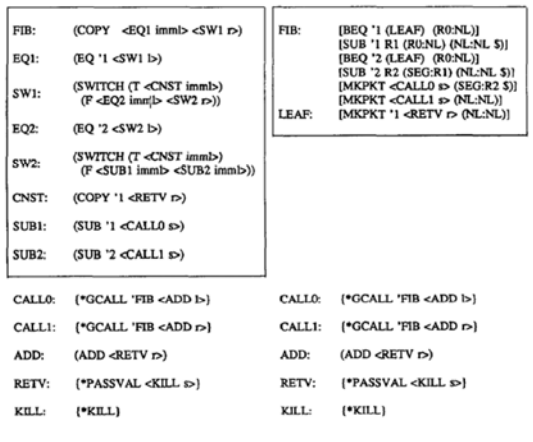
Benefits of Control Flow within Data Flow

- 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



```
FIB:    (COPY   <EQ1 imml> <SW1 r>)
EQ1:    (EQ '1 <SW1 l>)
SW1:    (SWITCH (T <CNST imml>)
          (F <EQ2 imml> <SW2 r>))
EQ2:    (EQ '2 <SW2 l>)
SW2:    (SWITCH (T <CNST imml>)
          (F <SUB1 imml> <SUB2 imml>))
CNST:   (COPY '1 <RETV r>)
SUB1:   (SUB '1 <CALL0 s>)
SUB2:   (SUB '2 <CALL1 s>)

CALL0:  {*GCALL 'FIB <ADD l>}
CALL1:  {*GCALL 'FIB <ADD r>}
ADD:    (ADD <RETV r>)
RETV:   {*PASSVAL <KILL s>}
KILL:   {*KILL}
```

(a) A Normal Dataflow Program

```
FIB:    [BEQ '1 (LEAF)  (R0:NL)]
        [SUB '1 R1 (R0:NL) (NL:NL $)]
        [BEQ '2 (LEAF)  (R0:NL)]
        [SUB '2 R2 (SEG:R1) (NL:NL $)]
        [MKPKT <CALL0 s> (SEG:R2 $)]
        [MKPKT <CALL1 s> (NL:NL)]
LEAF:   [MKPKT '1 <RETV r> (NL:NL)]

CALL0:  {*GCALL 'FIB <ADD l>}
CALL1:  {*GCALL 'FIB <ADD r>}
ADD:    (ADD <RETV r>)
RETV:   {*PASSVAL <KILL s>}
KILL:   {*KILL}
```

(b) A Strongly Connected Dataflow Program

Figure 5 FIBONACCI Program.

Macro Dataflow Machine Example

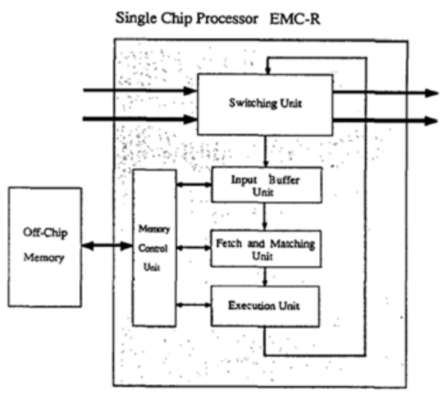
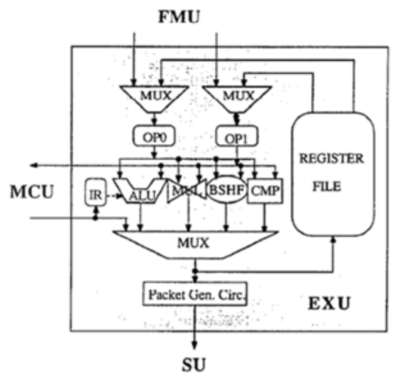


Figure 6 Block Diagram of the EMC-R.



IR : Instruction Register OPi : Operand Register i

Figure 8 Execution Unit Organization.

Macro Dataflow Pipeline Organization

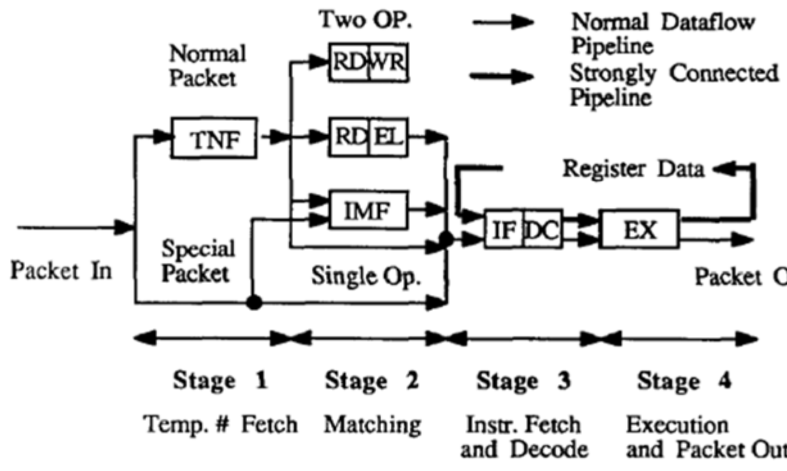


Figure 9 Pipeline Organization of the EMC-R.

Model 1 Example: Restricted Data Flow

- Data flow execution under sequential semantics and precise exceptions

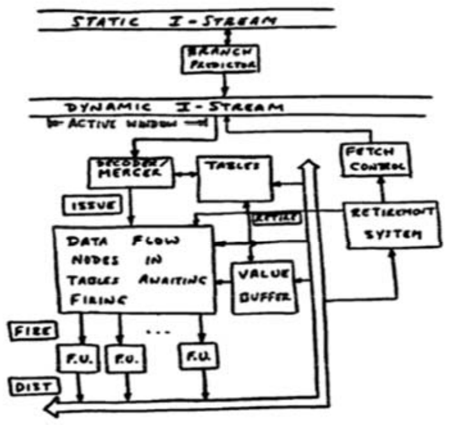


FIGURE 1.

Patt et al., "HPS, a new microarchitecture: rationale and introduction," MICRO 1985.

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

- H. T. Kung, "Why Systolic Architectures?," IEEE Computer 1982.

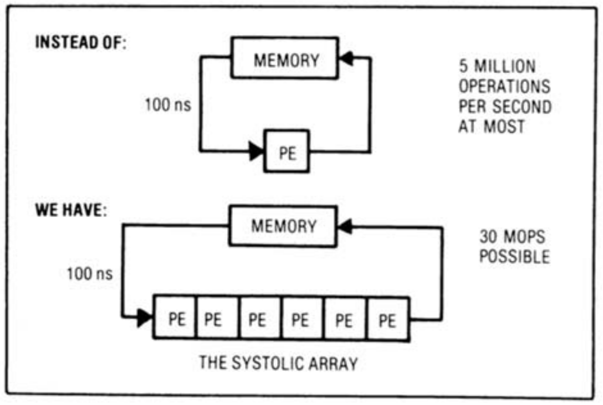


Figure 1. Basic principle of a systolic system.

Memory: heart
PEs: cells

Memory pulses data through cells

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

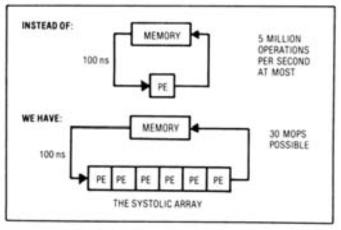


Figure 1. Basic principle of a systolic system.

- 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 $\{y_1, y_2, \ldots, y_{n+1-k}\}$
defined by

$$y_i = w_1x_i + w_2x_{i+1} + \ldots + w_kx_{i+k-1}$$

Systolic Computation Example: Convolution

- y1 = w1x1 + w2x2 + w3x3
- y2 = w1x2 + w2x3 + w3x4
- y3 = w1x3 + w2x4 + w3x5

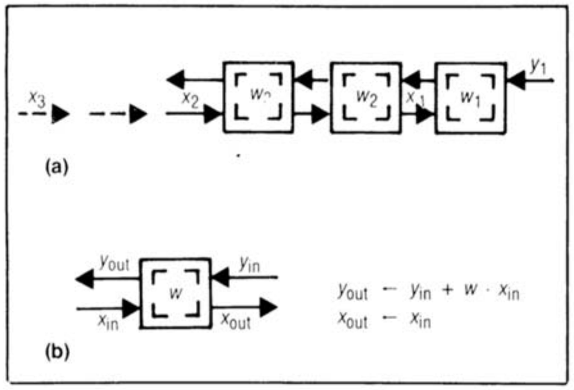


Figure 8. Design W1: systolic convolution array (a) and cell (b) where w_i's stay and x_i's and y_i's move systolically in opposite directions.

Systolic Computation Example: Convolution

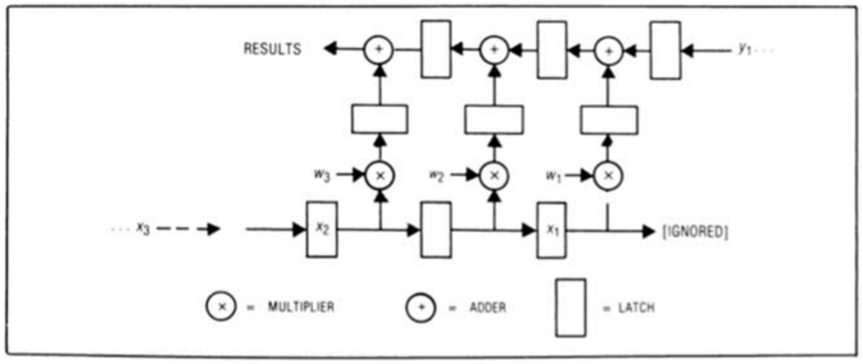


Figure 10. Overlapping the executions of multiply and add in design W1.

- 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 → to store partial/temporary results, constants
  - Leads to stream processing, pipeline parallelism
    - More generally, staged execution

Pipeline Parallelism

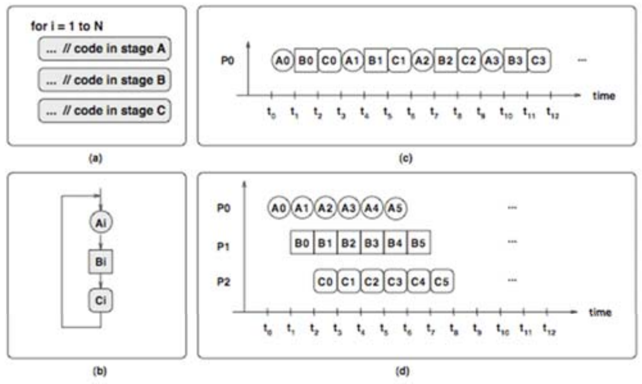


Figure 1. (a) The code of a loop, (b) Each iteration is split into 3 pipeline stages: A, B, and C. Iteration i comprises Ai, Bi, Ci. (c) Sequential execution of 4 iterations. (d) Parallel execution of 6 iterations using pipeline parallelism on a three-core machine. Each stage executes on one core.

File Compression Example

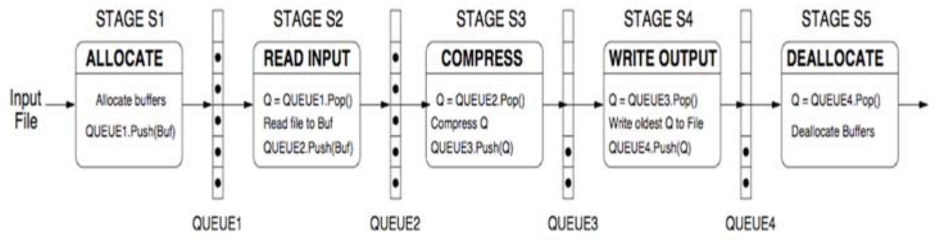


Figure 3. File compression algorithm executed using pipeline parallelism

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
- Annaratone et al., "Warp Architecture and Implementation," ISCA 1986.
- Annaratone et al., "The Warp Computer: Architecture, Implementation, and Performance," IEEE TC 1987.

The WARP Computer

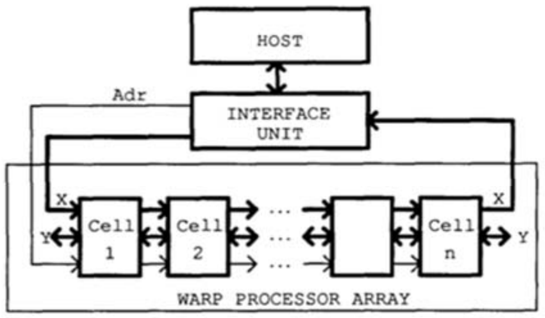


Figure 1: Warp system overview

The WARP Computer

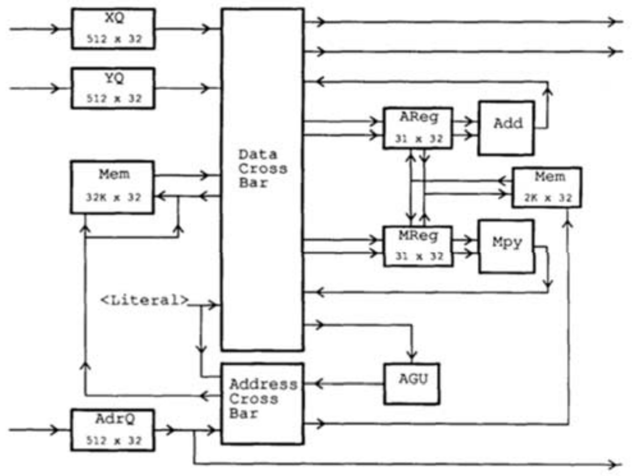


Figure 2: Warp cell data path

Models and Architectures

- In-order scalar Von-Neumann
- OoO scalar Von-Neumann
- SIMD
- Vector
- SPMD

- Static Dataflow
- Dynamic Dataflow
- Stream processing
- Systolic