18-447 Computer Architecture Lecture 15: Dataflow and SIMD

Prof. Onur Mutlu
Carnegie Mellon University

Spring 2013, 2/20/2013

Reminder: Homework 3

- Homework 3
  - Due Feb 25
  - REP MOVS in Microprogrammed LC-3b, Pipelining, Delay Slots, Interlocking, Branch Prediction

Lab Assignment 3 Due March 1

- Lab Assignment 3
  - Due Friday, March 1
  - Pipelined MIPS implementation in Verilog
  - All labs are individual assignments
  - No collaboration; please respect the honor code

  - Extra credit: Optimize for execution time!
    - Top assignments with lowest execution times will get extra credit.
    - And, it will be fun to optimize…

Course Feedback Sheet

- Was due Feb 15, in class
- But, please still turn it in
- We would like your honest feedback on the course

Readings for Today

- Lindholm et al., "NVIDIA Tesla: A Unified Graphics and Computing Architecture," IEEE Micro 2008.

- Fatahalian and Houston, "A Closer Look at GPUs," CACM 2008.

- Stay tuned for more readings…

Readings for Next Week

- Virtual Memory

- Section 5.4 in Patterson & Hennessy
- Section 8.8 in Hamacher et al.

Last Lecture

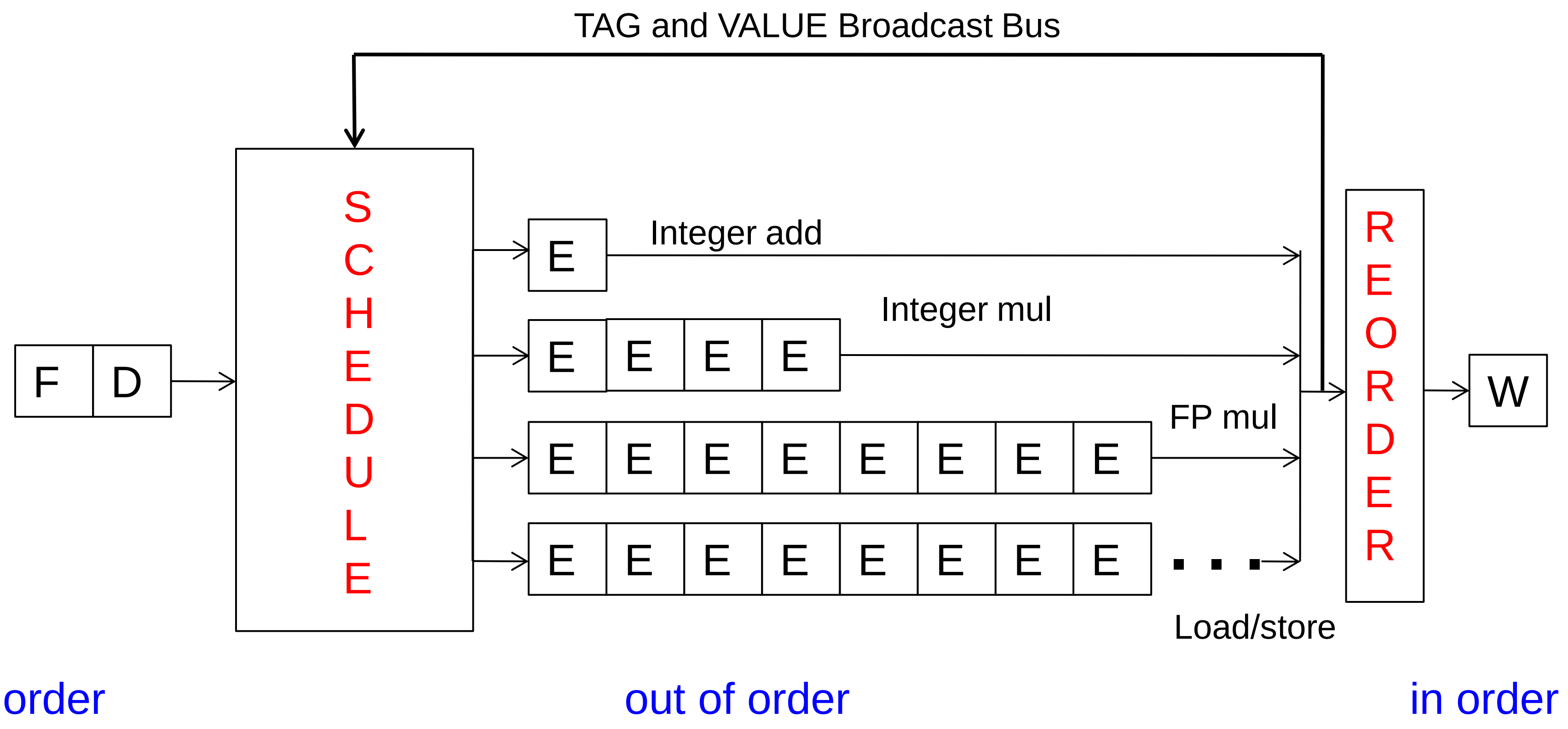
- Out-of-order execution
  - Tomasulo's algorithm
  - Example
  - OoO as restricted dataflow execution

Today

- Wrap up out-of-order execution
  - Memory dependence handling
  - Alternative designs

Out-of-Order Execution (Dynamic Instruction Scheduling)

Review: Out-of-Order Execution with Precise Exceptions



- Hump 1: Reservation stations (scheduling window)
- Hump 2: Reordering (reorder buffer, aka instruction window or active window)

Review: Enabling OoO Execution, Revisited

1. Link the consumer of a value to the producer
   - Register renaming: Associate a "tag" with each data value

2. Buffer instructions until they are ready
   - Insert instruction into reservation stations after renaming

3. Keep track of readiness of source values of an instruction
   - Broadcast the "tag" when the value is produced
   - Instructions compare their "source tags" to the broadcast tag → if match, source value becomes ready

4. When all source values of an instruction are ready, dispatch the instruction to functional unit (FU)
   - Wakeup and select/schedule the instruction

Review: Summary of OOO Execution Concepts

- Register renaming eliminates false dependencies, enables linking of producer to consumers
- Buffering enables the pipeline to move for independent ops
- Tag broadcast enables communication (of readiness of produced value) between instructions
- Wakeup and select enables out-of-order dispatch

Review: Registers versus Memory, Revisited

- So far, we considered register based value communication between instructions

- What about memory?

- What are the fundamental differences between registers and memory?
  - Register dependences known statically – memory dependences determined dynamically
  - Register state is small – memory state is large
  - Register state is not visible to other threads/processors – memory state is shared between threads/processors (in a shared memory multiprocessor)

Review: Memory Dependence Handling (I)

- Need to obey memory dependences in an out-of-order machine
  - and need to do so while providing high performance

- Observation and Problem: Memory address is not known until a load/store executes

- Corollary 1: Renaming memory addresses is difficult
- Corollary 2: Determining dependence or independence of loads/stores need to be handled after their execution
- Corollary 3: When a load/store has its address ready, there may be younger/older loads/stores with undetermined addresses in the machine

Review: Memory Dependence Handling (II)

- When do you schedule a load instruction in an OOO engine?
  - Problem: A younger load can have its address ready before an older store's address is known
  - Known as the memory disambiguation problem or the unknown address problem

- Approaches
  - Conservative: Stall the load until all previous stores have computed their addresses (or even retired from the machine)
  - Aggressive: Assume load is independent of unknown-address stores and schedule the load right away
  - Intelligent: Predict (with a more sophisticated predictor) if the load is dependent on the/any unknown address store

Handling of Store-Load Dependencies

- A load's dependence status is not known until all previous store addresses are available.

- How does the OOO engine detect dependence of a load instruction on a previous store?
  - Option 1: Wait until all previous stores committed (no need to check)
  - Option 2: Keep a list of pending stores in a store buffer and check whether load address matches a previous store address

- How does the OOO engine treat the scheduling of a load instruction wrt previous stores?
  - Option 1: Assume load dependent on all previous stores
  - Option 2: Assume load independent of all previous stores
  - Option 3: Predict the dependence of a load on an outstanding store

Memory Disambiguation (I)

- Option 1: Assume load dependent on all previous stores

  + No need for recovery

  -- Too conservative: delays independent loads unnecessarily

- Option 2: Assume load independent of all previous stores

  + Simple and can be common case: no delay for independent loads

  -- Requires recovery and re-execution of load and dependents on misprediction

- Option 3: Predict the dependence of a load on an outstanding store

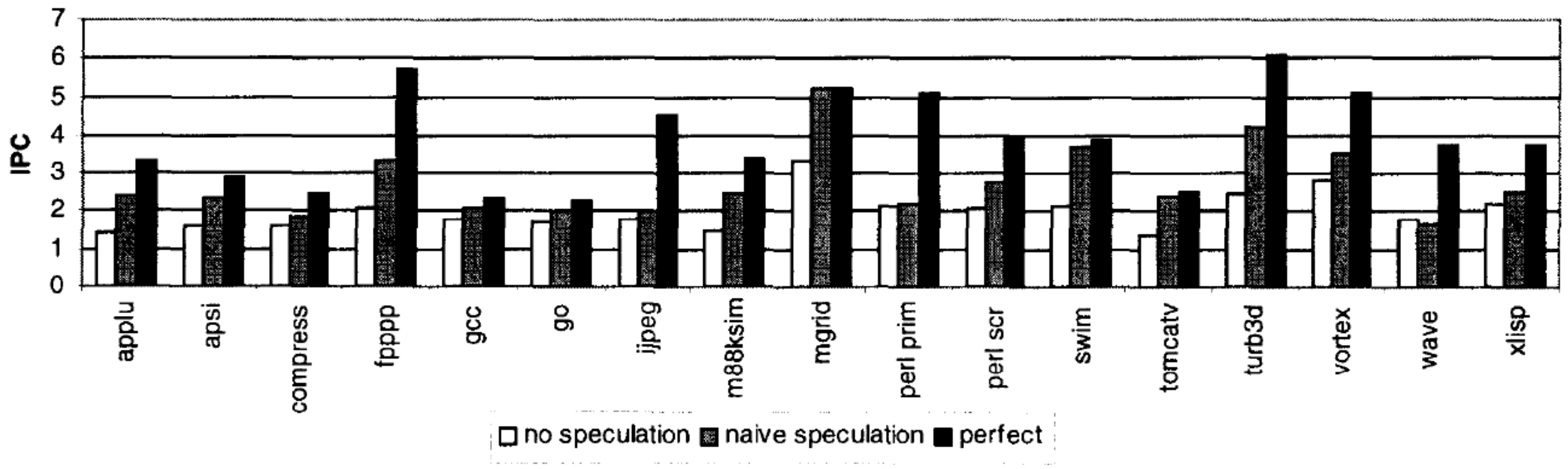
  + More accurate. Load store dependencies persist over time

  -- Still requires recovery/re-execution on misprediction

  - Alpha 21264 : Initially assume load independent, delay loads found to be dependent
  - Moshovos et al., "Dynamic speculation and synchronization of data dependences," ISCA 1997.
  - Chrysos and Emer, "Memory Dependence Prediction Using Store Sets," ISCA 1998.

Memory Disambiguation (II)

- Chrysos and Emer, "Memory Dependence Prediction Using Store Sets," ISCA 1998.



- Predicting store-load dependencies important for performance
- Simple predictors (based on past history) can achieve most of the potential performance

Food for Thought for You

- Many other design choices

- Should reservation stations be centralized or distributed?
  - What are the tradeoffs?

- Should reservation stations and ROB store data values or should there be a centralized physical register file where all data values are stored?
  - What are the tradeoffs?

- Exactly when does an instruction broadcast its tag?

- …

More Food for Thought for You

- How can you implement branch prediction in an out-of-order execution machine?
  - Think about branch history register and PHT updates
  - Think about recovery from mispredictions
    - How to do this fast?

- How can you combine superscalar execution with out-of-order execution?
  - These are different concepts
  - Concurrent renaming of instructions
  - Concurrent broadcast of tags

- How can you combine superscalar + out-of-order + branch prediction?

Recommended Readings

- Kessler, "The Alpha 21264 Microprocessor," IEEE Micro, March-April 1999.
- Boggs et al., "The Microarchitecture of the Pentium 4 Processor," Intel Technology Journal, 2001.
- Yeager, "The MIPS R10000 Superscalar Microprocessor," IEEE Micro, April 1996
- Tendler et al., "POWER4 system microarchitecture," IBM Journal of Research and Development, January 2002.

Other Approaches to Concurrency (or Instruction Level Parallelism)

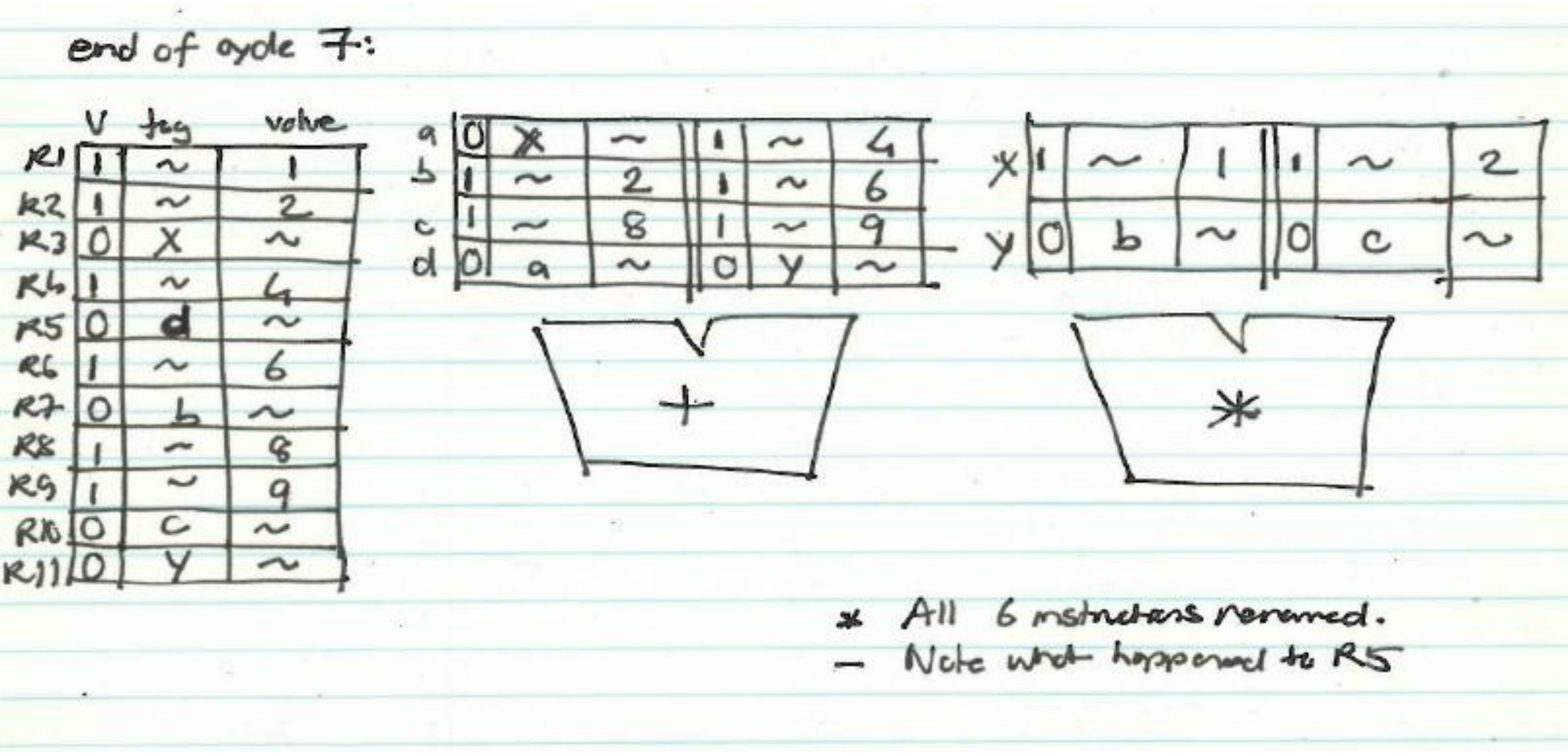
Approaches to (Instruction-Level) Concurrency

- Out-of-order execution
- Dataflow (at the ISA level)
- SIMD Processing
- VLIW

- Systolic Arrays
- Decoupled Access Execute

Data Flow:
Exploiting Irregular Parallelism

Remember: State of RAT and RS in Cycle 7



end of cycle 7:

	V	tag	value
R1	1	~	1
R2	1	~	2
R3	0	X	~
R4	1	~	4
R5	0	d	~
R6	1	~	6
R7	0	b	~
R8	1	~	8
R9	1	~	9
R10	0	c	~
R11	0	y	~

a	0	x	~	1	~	4
b	1	~	2	1	~	6
c	1	~	8	1	~	9
d	0	a	~	0	y	~

x	1	~	1	1	~	2
y	0	b	~	0	c	~

* All 6 instructions renamed.
– Note what happened to R5

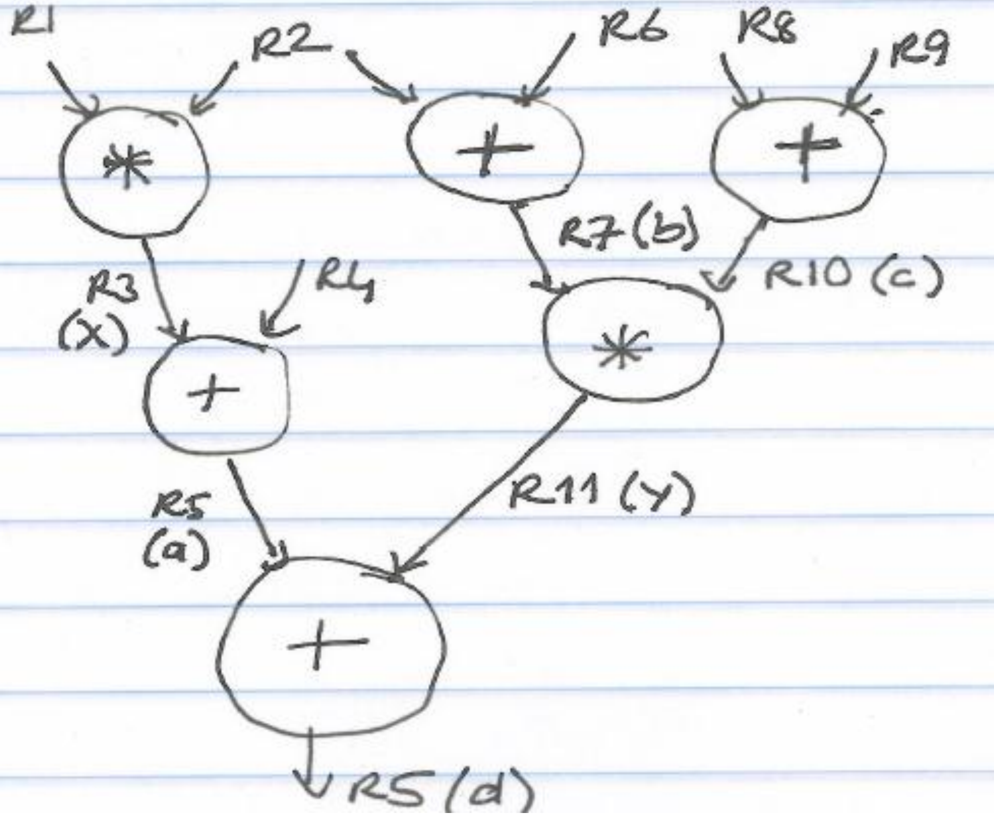
Remember: Dataflow Graph

```
MUL R1, R2 → R3 (x)
ADD R3, R4 → R5 (a)
ADD R2, R6 → R7 (b)
ADD R8, R9 → R10 (c)
MUL R7, R10 → R11 (y)
ADD R5, R11 → R5 (d)
```

Dataflow graph

Nodes: operations performed by the instruction

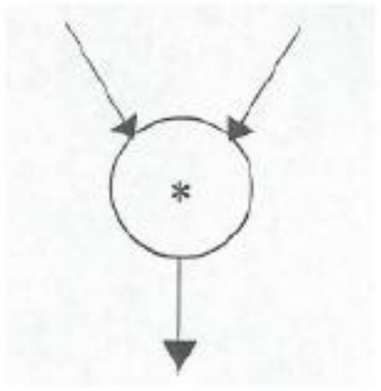
Arcs: tags in Tomasulo's algorithm

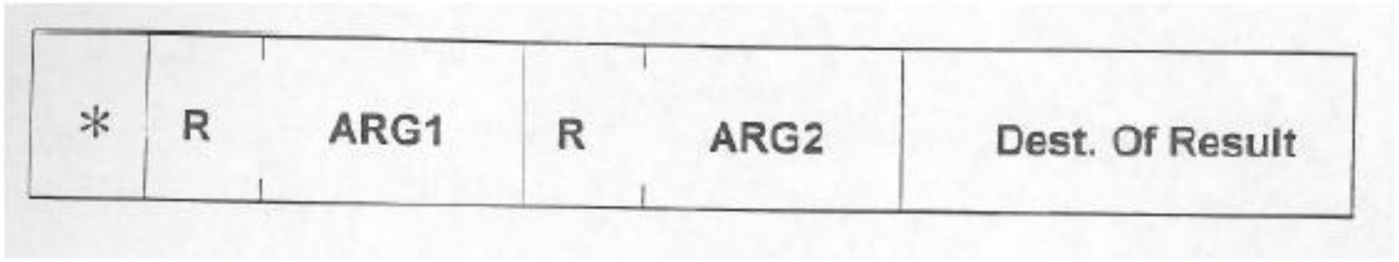


Review: More on Data Flow

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

- Data flow node and its ISA representation

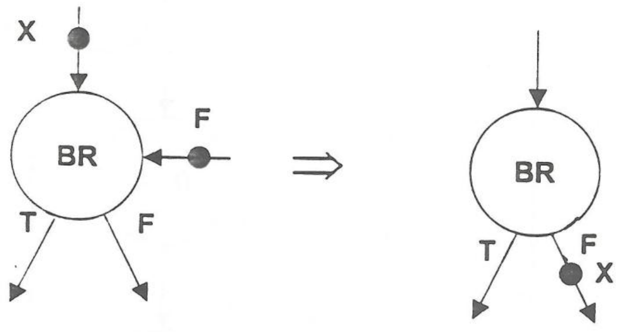




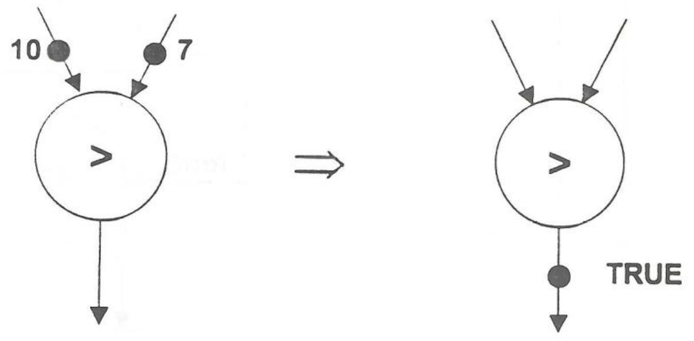
*	R	ARG1	R	ARG2	Dest. Of Result

Data Flow Nodes

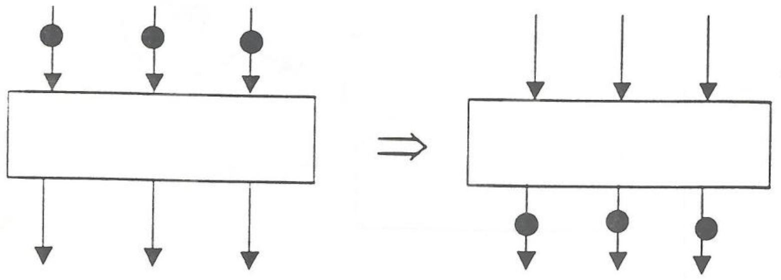
✱*Conditional*



✱*Relational*

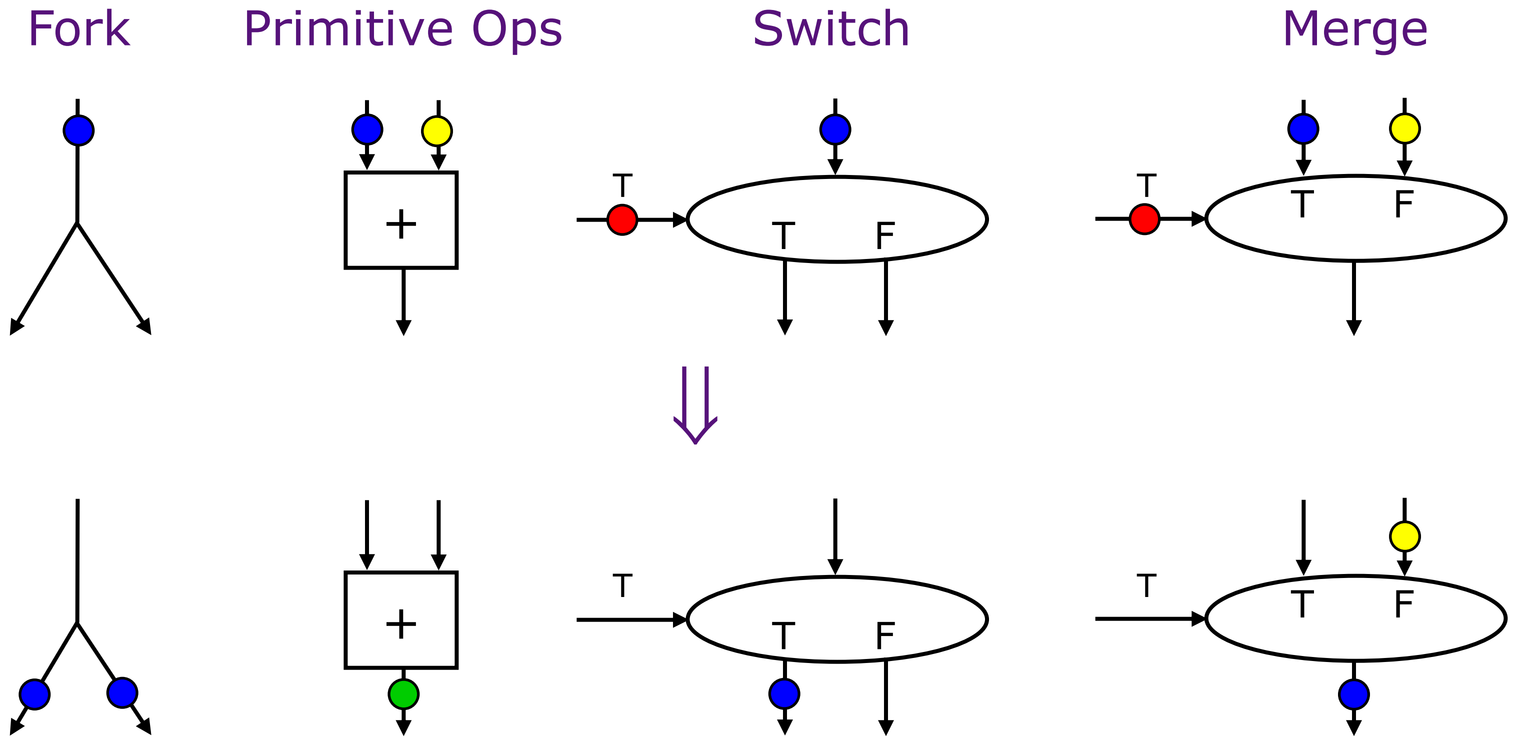


✱**Barrier Synch**



Dataflow Nodes (II)

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



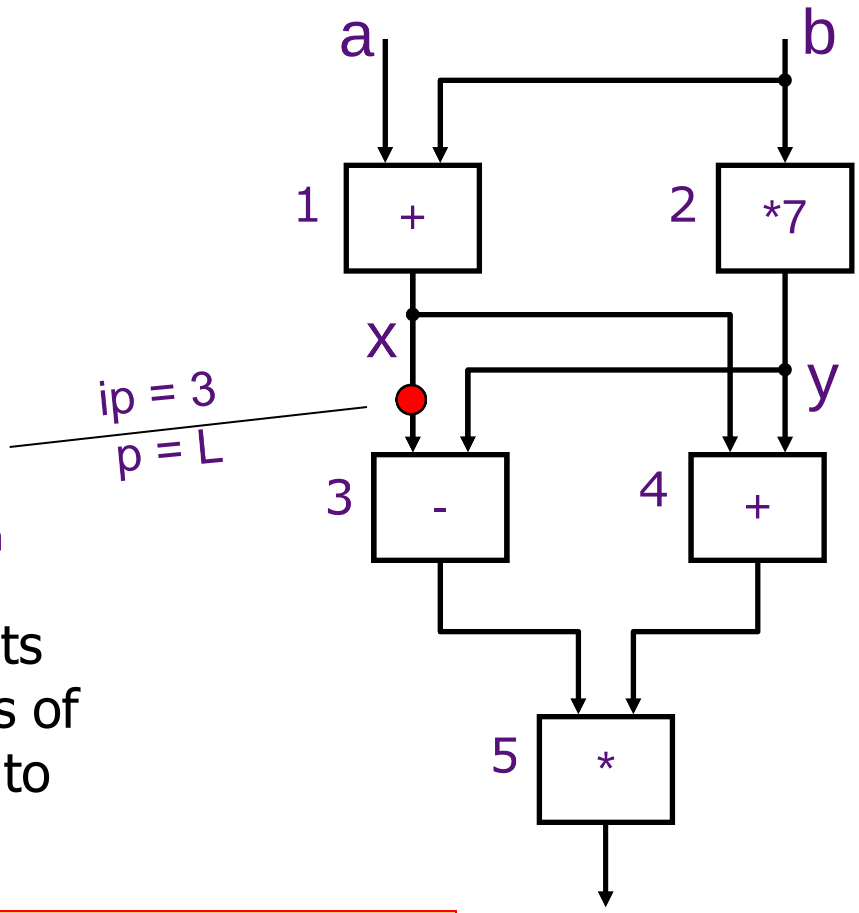
Dataflow Graphs

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

- Values in dataflow graphs are represented as tokens

token < ip , p , v >

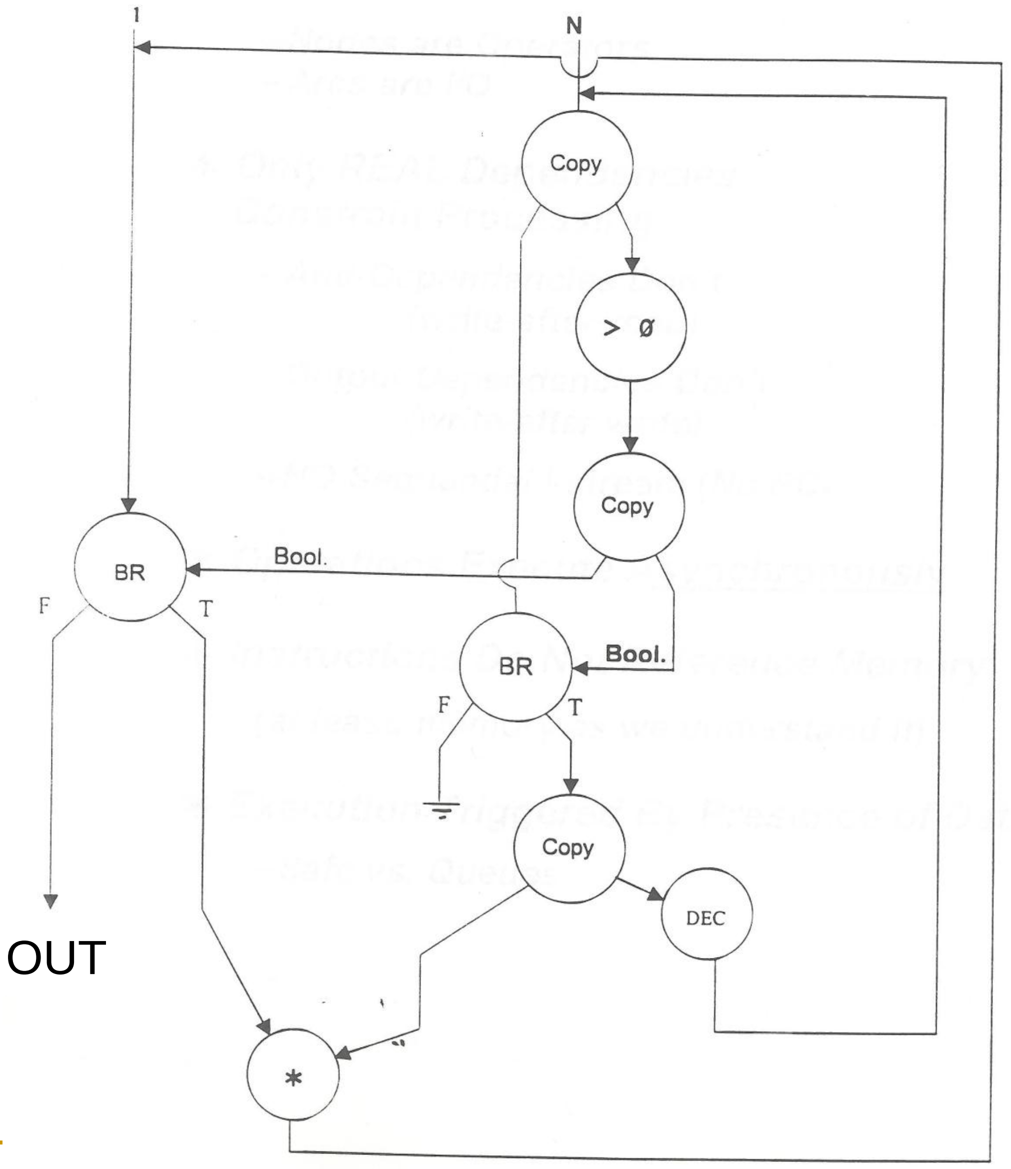
instruction ptr   port   data



- 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

```
a := x + y
b := a × a
c := 4 — a
```

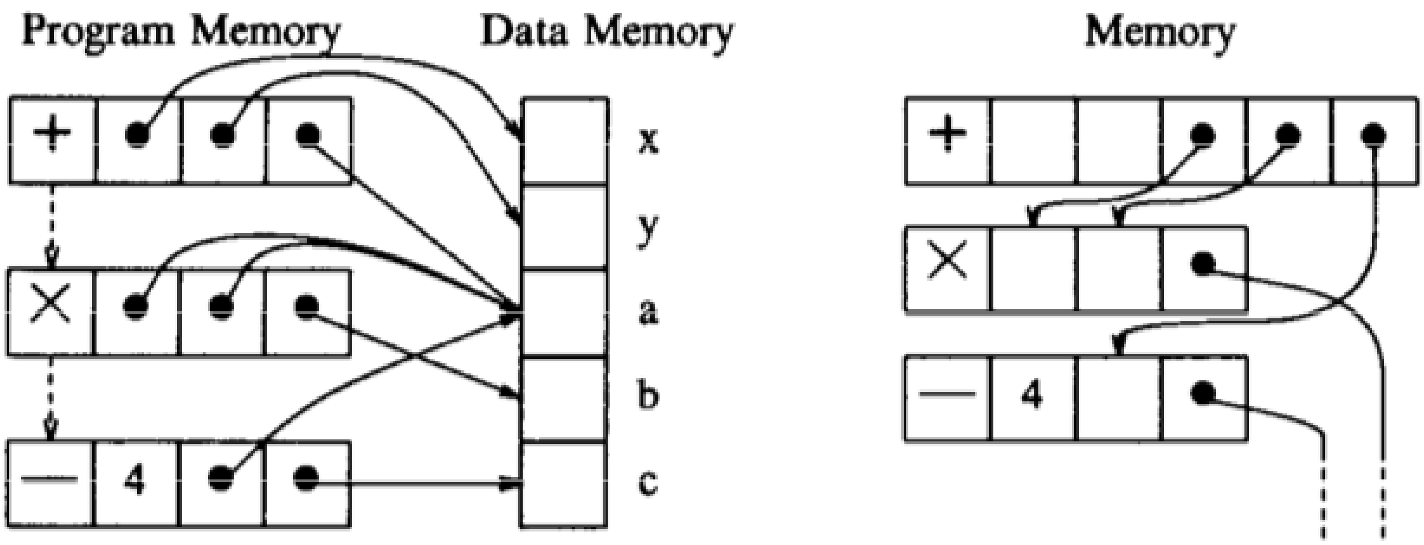
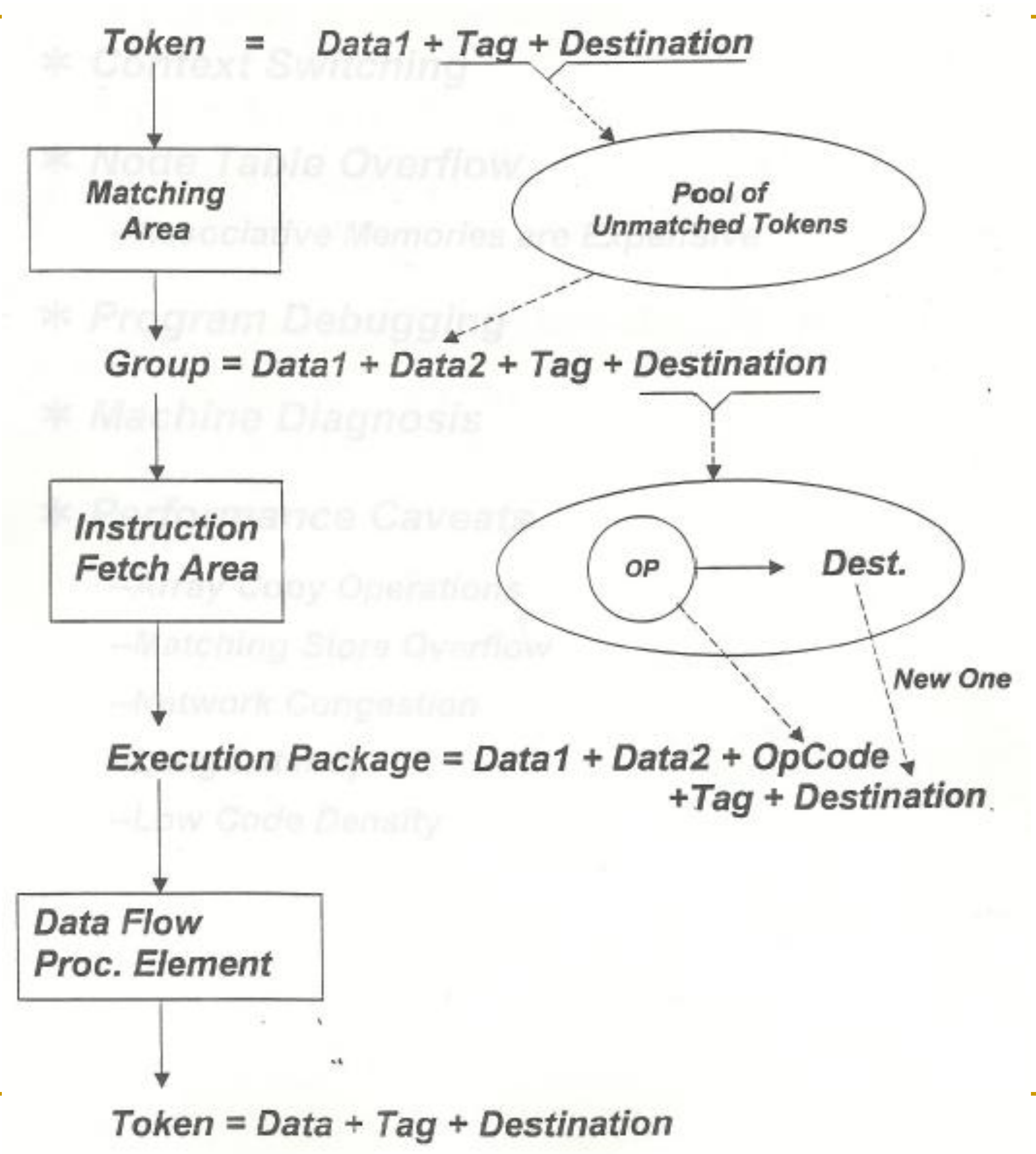


Figure 2. A comparison of control flow and dataflow programs. On the left a control flow program for a computer with memory-to-memory instructions. The arcs point to the locations of data that are to be used or created. Control flow arcs are indicated with dashed arrows; usually most of them are implicit.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

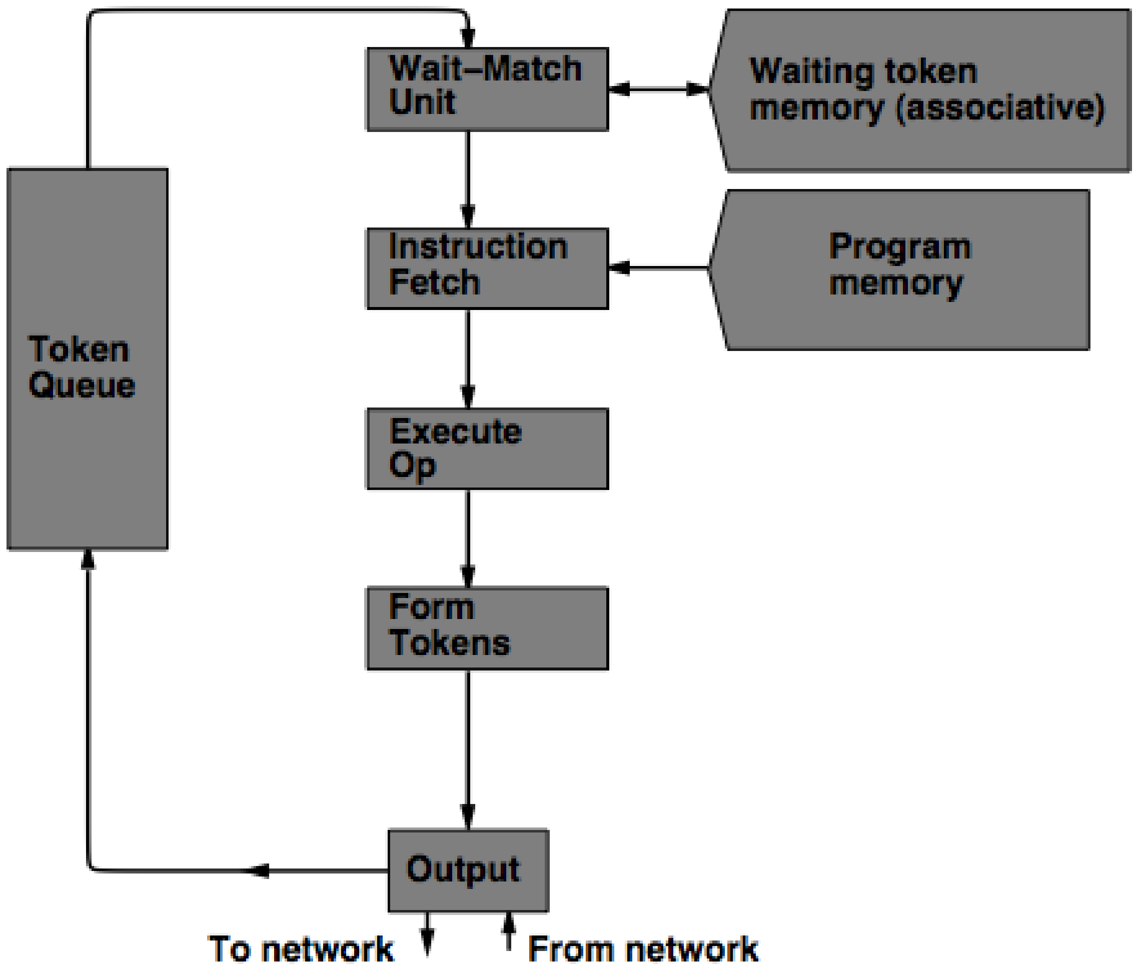
Data Flow Characteristics

- 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

A Dataflow Processor


Token = Data1 + Tag + Destination
Matching Area
Pool of Unmatched Tokens
Group = Data1 + Data2 + Tag + Destination
Instruction Fetch Area
OP
Dest.
New One
Execution Package = Data1 + Data2 + OpCode +Tag + Destination
Data Flow Proc. Element
Token = Data + Tag + Destination

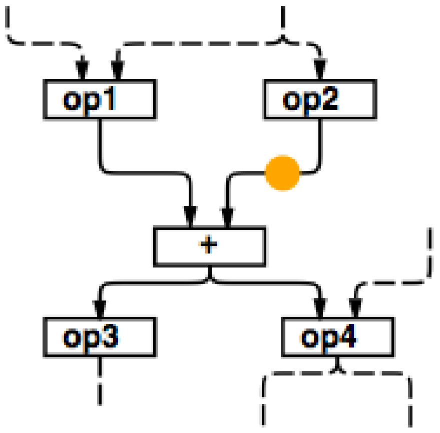
MIT Tagged Token Data Flow Architecture



- 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

Conceptual



Encoding of token:

A "packet" containing:

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

Encoding of graph

Program memory:

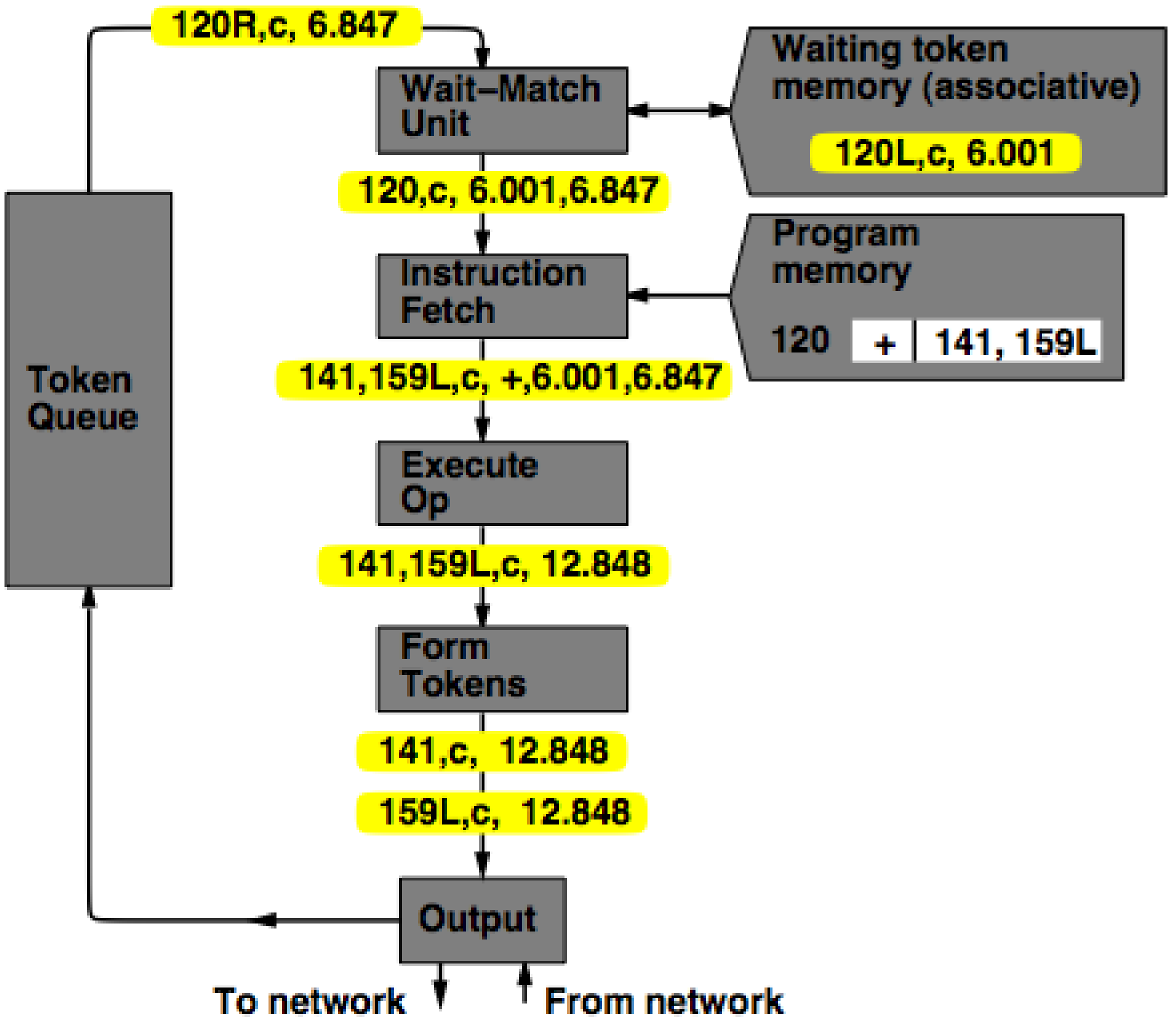
	Op–code	Destination(s)
109	op1	120L
113	op2	120R
120	+	141, 159L
141	op3	...
159	op4	... , ...

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

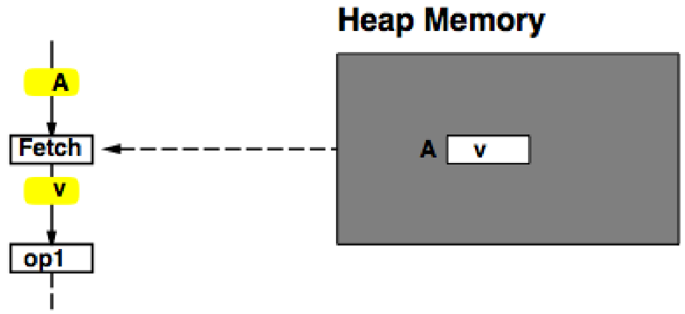
120R	Destination instruction address, Left/Right port
Ctxt	Context Identifier
6.847	Value

TTDA Data Flow Example

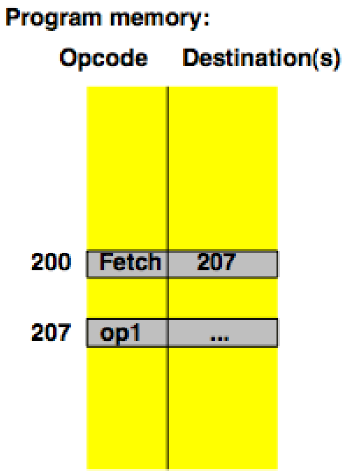


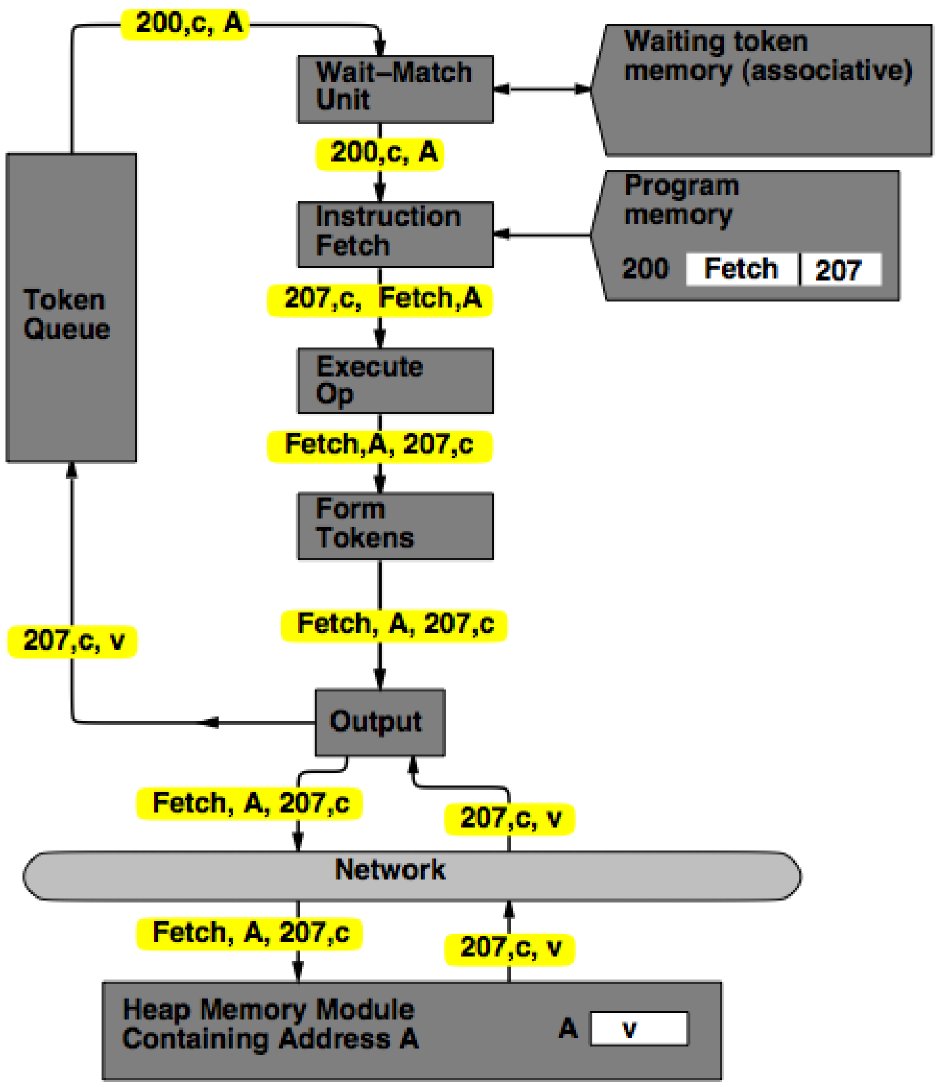
TTDA Data Flow Example

Conceptual:

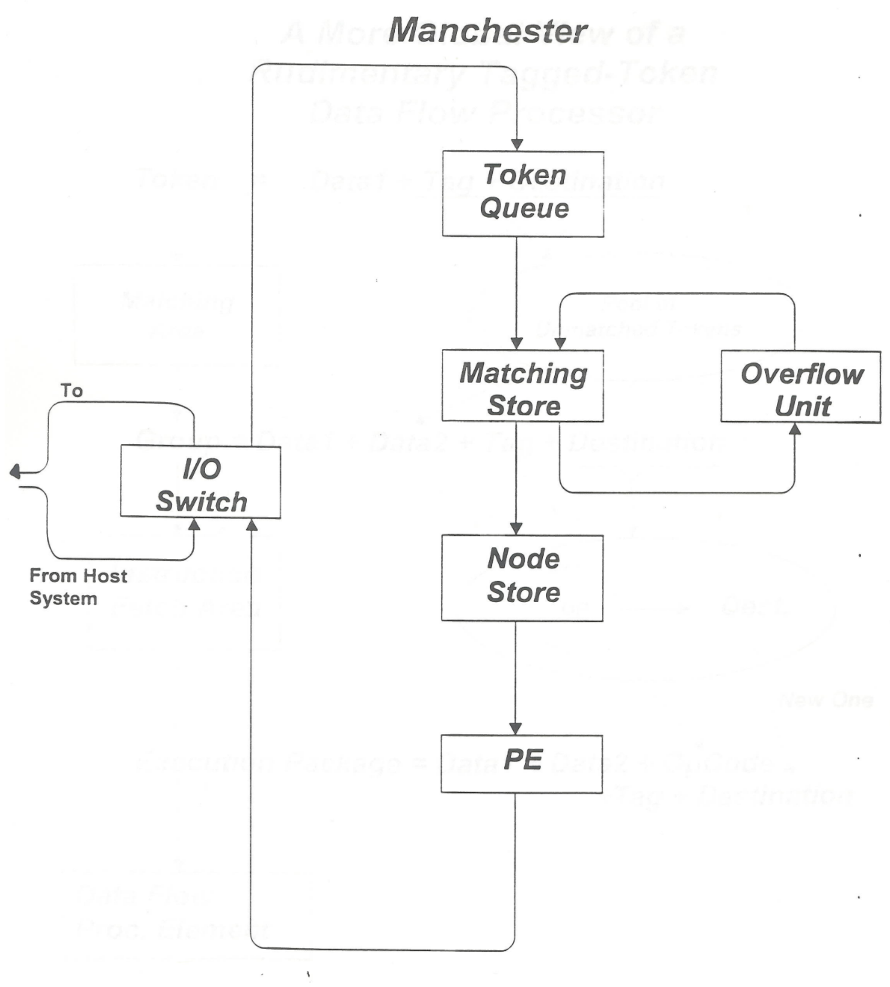


Encoding of graph:





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

Data Flow Advantages/Disadvantages

- Advantages
  - Very good at exploiting irregular parallelism
  - Only real dependencies constrain processing

- Disadvantages
  - No precise state
    - Interrupt/exception handling is difficult
    - Debugging very difficult
  - Bookkeeping overhead (tag matching)
  - Too much parallelism? (Parallelism control needed)
    - Overflow of tag matching tables
  - Implementing dynamic data structures difficult

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 very successful
    - Out of order execution
    - Hwu and Patt, "HPSm, a high performance restricted data flow architecture having minimal functionality," ISCA 1986.

Further Reading on Data Flow

- ISA level dataflow
  - Gurd et al., "The Manchester prototype dataflow computer," CACM 1985.

- Microarchitecture-level dataflow:
  - Hwu and Patt, "HPSm, a high performance restricted data flow architecture having minimal functionality," ISCA 1986.

Vector Processing: Exploiting Regular (Data) Parallelism

Flynn's Taxonomy of Computers

- Mike Flynn, "Very High-Speed Computing Systems," Proc. of IEEE, 1966

- SISD: Single instruction operates on single data element
- SIMD: Single instruction operates on multiple data elements
  - Array processor
  - Vector processor
- MISD: Multiple instructions operate on single data element
  - Closest form: systolic array processor, streaming processor
- MIMD: Multiple instructions operate on multiple data elements (multiple instruction streams)
  - Multiprocessor
  - Multithreaded processor

Data Parallelism

- Concurrency arises from performing the same operations on different pieces of data
  - Single instruction multiple data (SIMD)
  - E.g., dot product of two vectors
- Contrast with data flow
  - Concurrency arises from executing different operations in parallel (in a data driven manner)
- Contrast with thread (“control”) parallelism
  - Concurrency arises from executing different threads of control in parallel
- SIMD exploits instruction-level parallelism
  - Multiple instructions concurrent: instructions happen to be the same

SIMD Processing

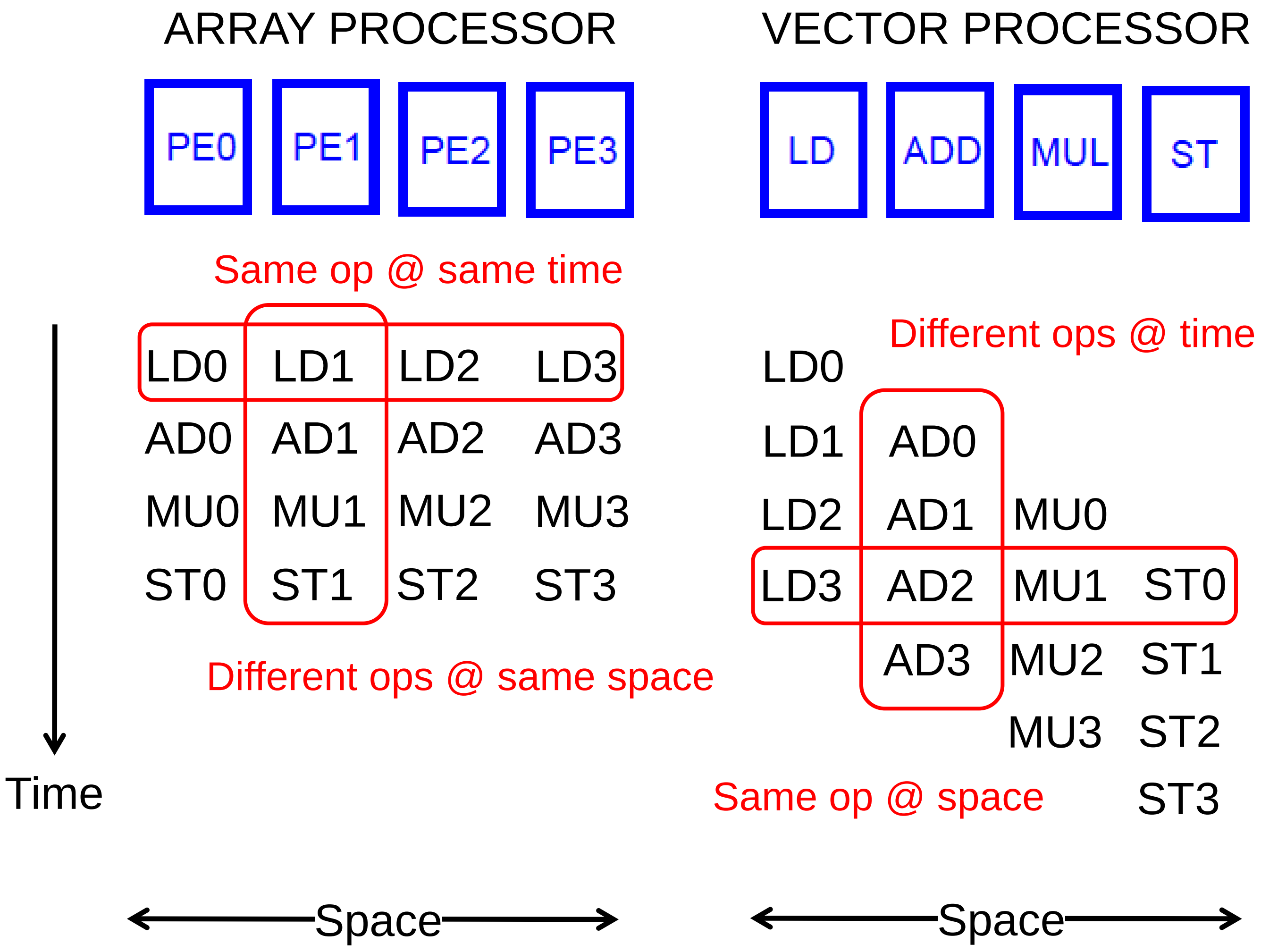
- Single instruction operates on multiple data elements
  - In time or in space
- Multiple processing elements

- Time-space duality
  - Array processor: Instruction operates on multiple data elements at the same time
  - Vector processor: Instruction operates on multiple data elements in consecutive time steps

Array vs. Vector Processors

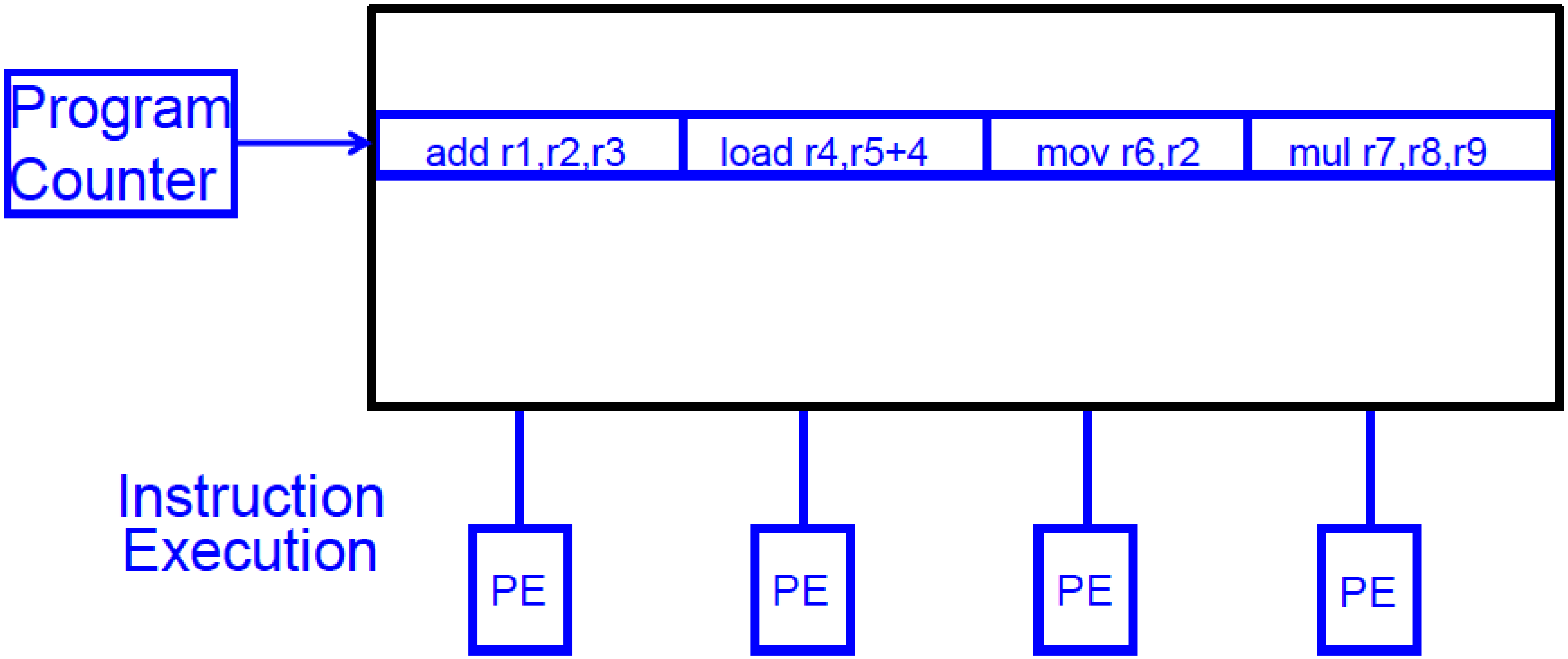
Instruction Stream

LD	VR ← A[3:0]
ADD	VR ← VR, 1
MUL	VR ← VR, 2
ST	A[3:0] ← VR



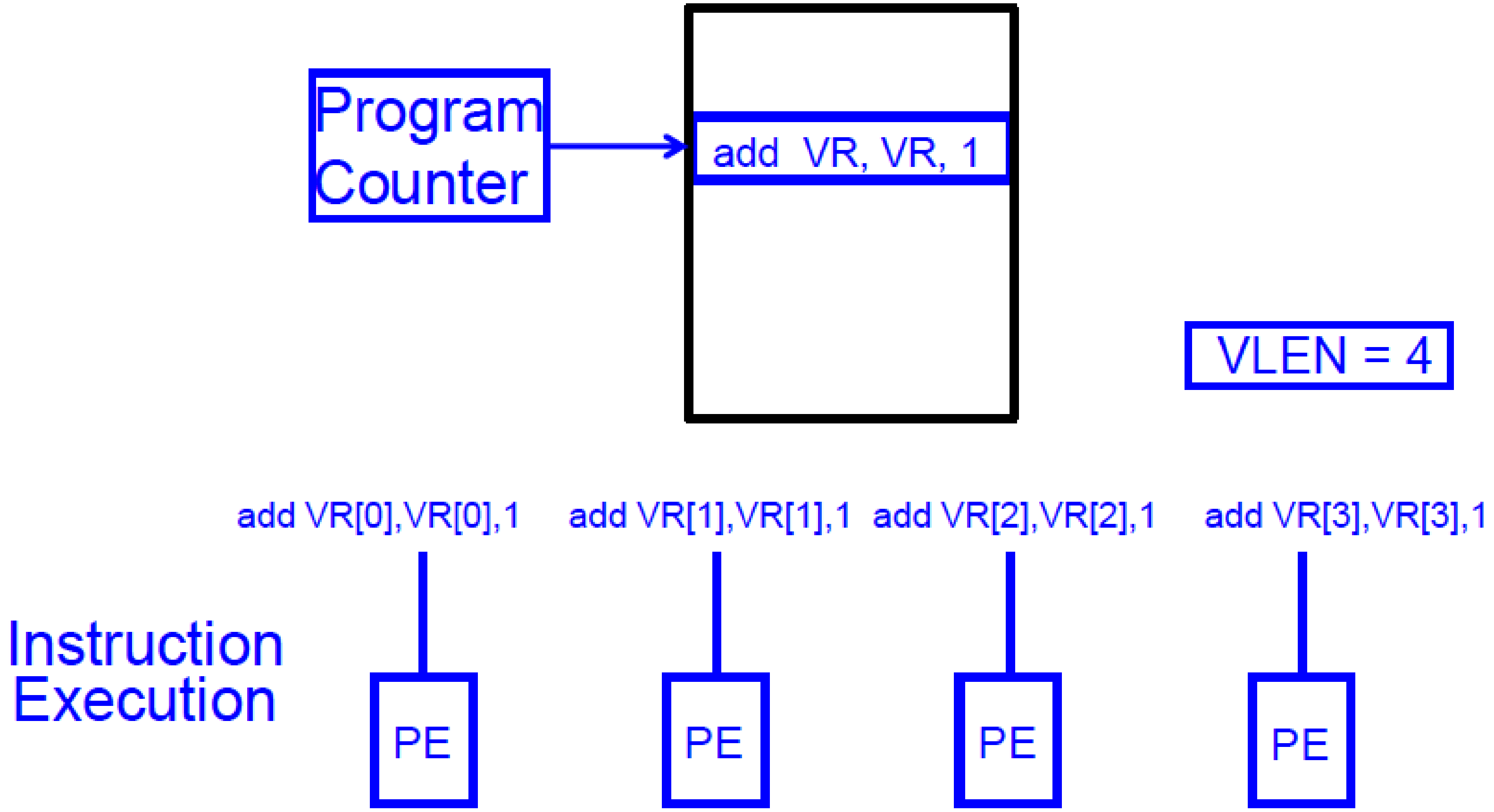
SIMD Array Processing vs. VLIW

- VLIW



SIMD Array Processing vs. VLIW

- Array processor



Vector Processors

- A vector is a one-dimensional array of numbers
- Many scientific/commercial programs use vectors

```
for (i = 0; i<=49; i++)
    C[i] = (A[i] + B[i]) / 2
```

- A vector processor is one whose instructions operate on vectors rather than scalar (single data) values
- Basic requirements
  - Need to load/store vectors → vector registers (contain vectors)
  - Need to operate on vectors of different lengths → vector length register (VLEN)
  - Elements of a vector might be stored apart from each other in memory → vector stride register (VSTR)
    - Stride: distance between two elements of a vector

Vector Processors (II)

- A vector instruction performs an operation on each element in consecutive cycles
  - Vector functional units are pipelined
  - Each pipeline stage operates on a different data element

- Vector instructions allow deeper pipelines
  - No intra-vector dependencies → no hardware interlocking within a vector
  - No control flow within a vector
  - Known stride allows prefetching of vectors into cache/memory

Vector Processor Advantages

+ No dependencies within a vector
  - Pipelining, parallelization work well
  - Can have very deep pipelines, no dependencies!

+ Each instruction generates a lot of work
  - Reduces instruction fetch bandwidth

+ Highly regular memory access pattern
  - Interleaving multiple banks for higher memory bandwidth
  - Prefetching

+ No need to explicitly code loops
  - Fewer branches in the instruction sequence

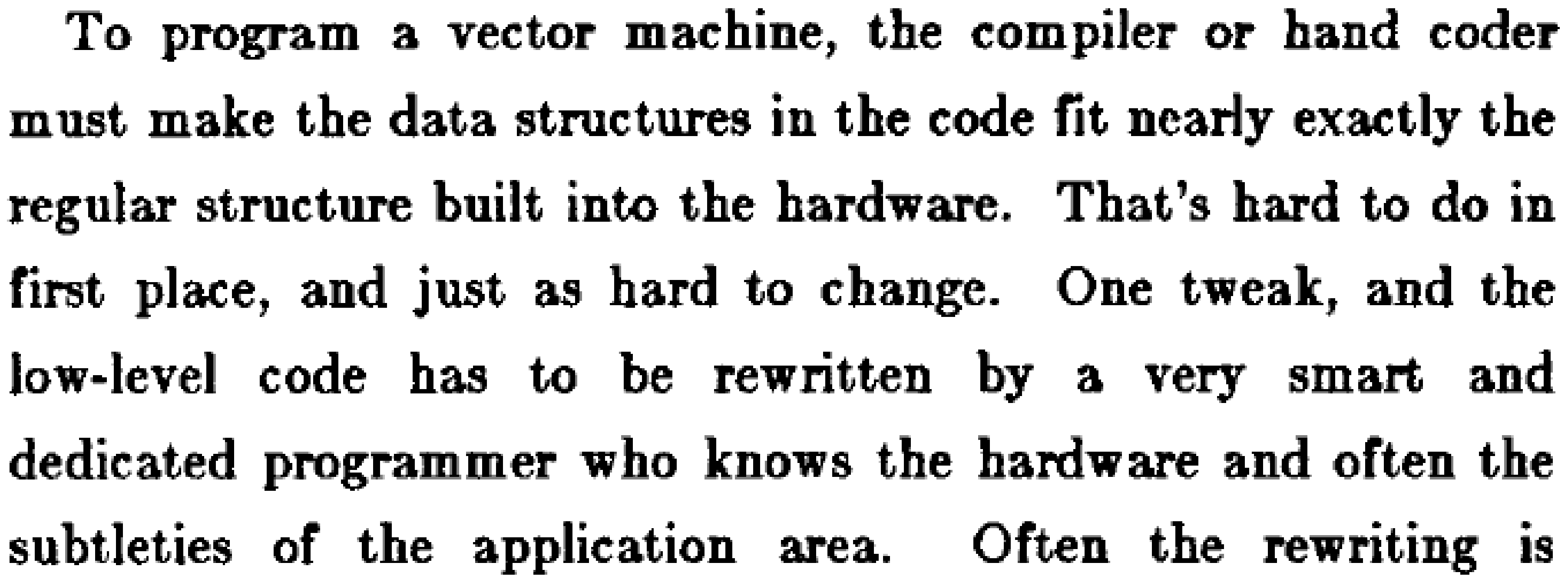
Vector Processor Disadvantages

-- Works (only) if parallelism is regular (data/SIMD parallelism)

++ Vector operations

-- Very inefficient if parallelism is irregular

-- How about searching for a key in a linked list?



To program a vector machine, the compiler or hand coder must make the data structures in the code fit nearly exactly the regular structure built into the hardware. That's hard to do in first place, and just as hard to change. One tweak, and the low-level code has to be rewritten by a very smart and dedicated programmer who knows the hardware and often the subtleties of the application area. Often the rewriting is

Fisher, "Very Long Instruction Word architectures and the ELI-512," ISCA 1983. 53

Vector Processor Limitations

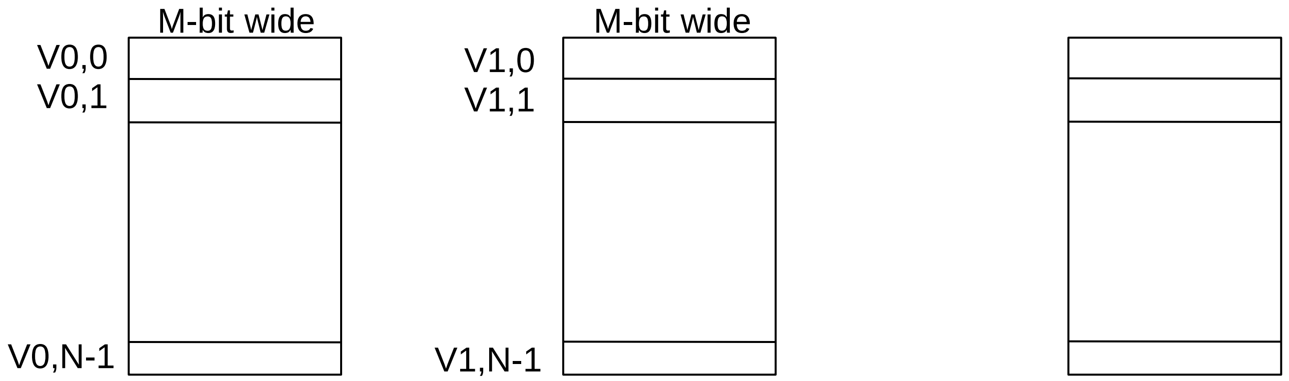
-- Memory (bandwidth) can easily become a bottleneck, especially if

1. compute/memory operation balance is not maintained
2. data is not mapped appropriately to memory banks

We did not cover the following slides in lecture. These are for your preparation for the next lecture.

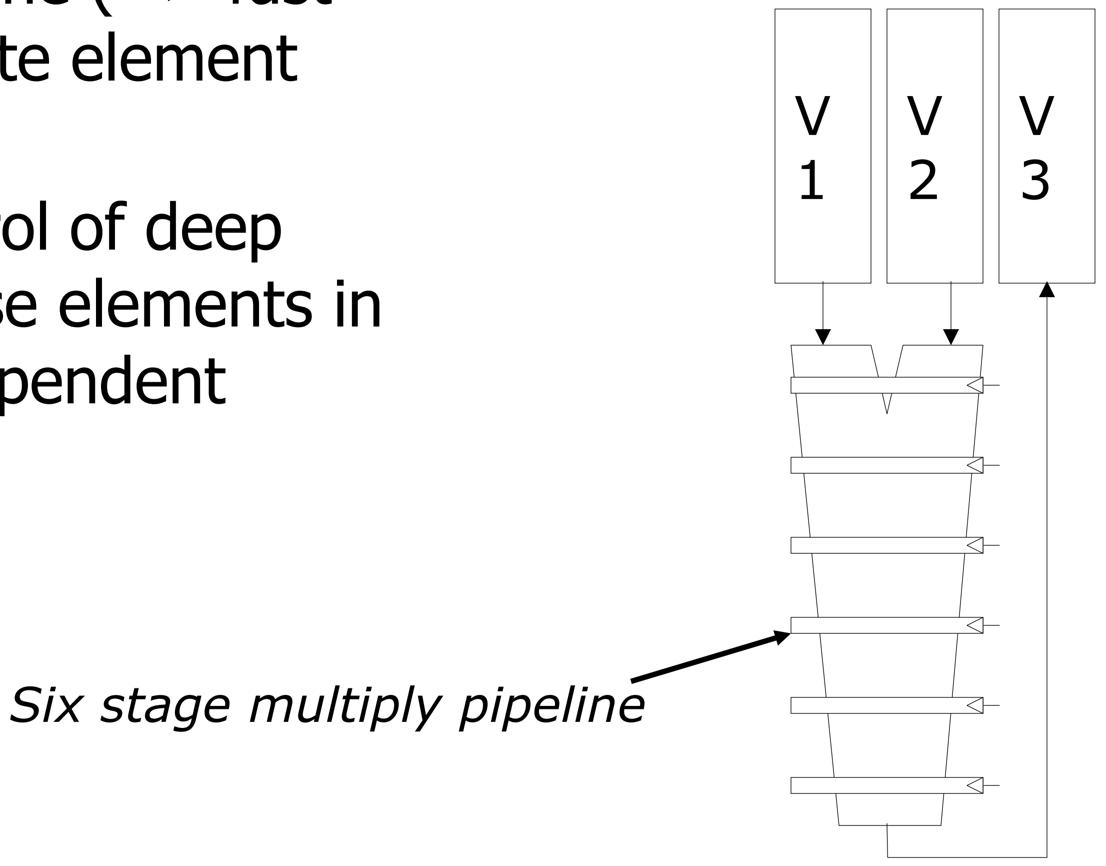
Vector Registers

- Each vector data register holds N M-bit values
- Vector control registers: VLEN, VSTR, VMASK
- Vector Mask Register (VMASK)
  - Indicates which elements of vector to operate on
  - Set by vector test instructions
    - e.g., VMASK[i] = (V_k[i] == 0)
- Maximum VLEN can be N
  - Maximum number of elements stored in a vector register



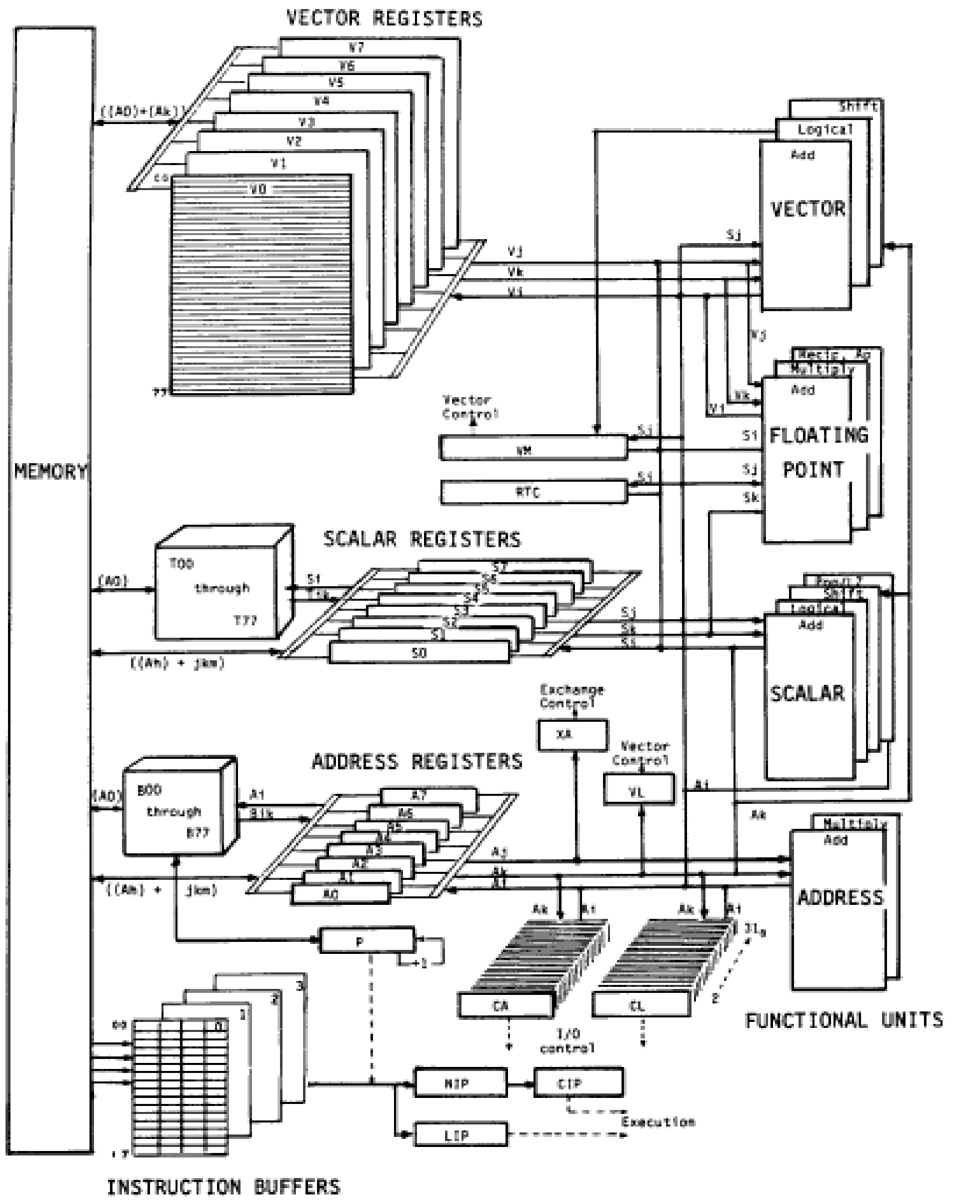
Vector Functional Units

- Use deep pipeline (=> fast clock) to execute element operations
- Simplifies control of deep pipeline because elements in vector are independent



V3 <- v1 * v2

Vector Machine Organization (CRAY-1)

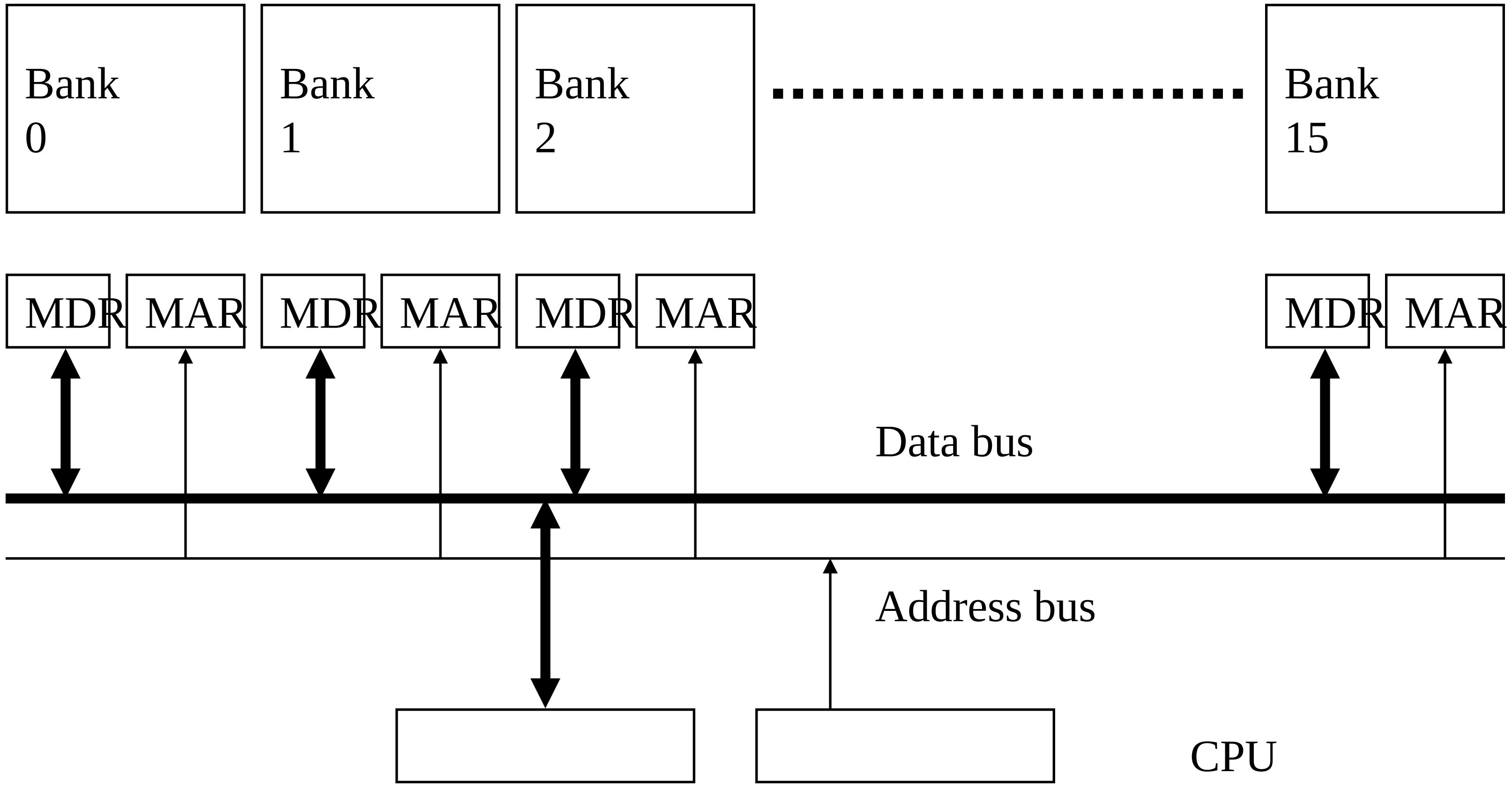


- CRAY-1
- Russell, "The CRAY-1 computer system," CACM 1978.

- Scalar and vector modes
- 8 64-element vector registers
- 64 bits per element
- 16 memory banks
- 8 64-bit scalar registers
- 8 24-bit address registers

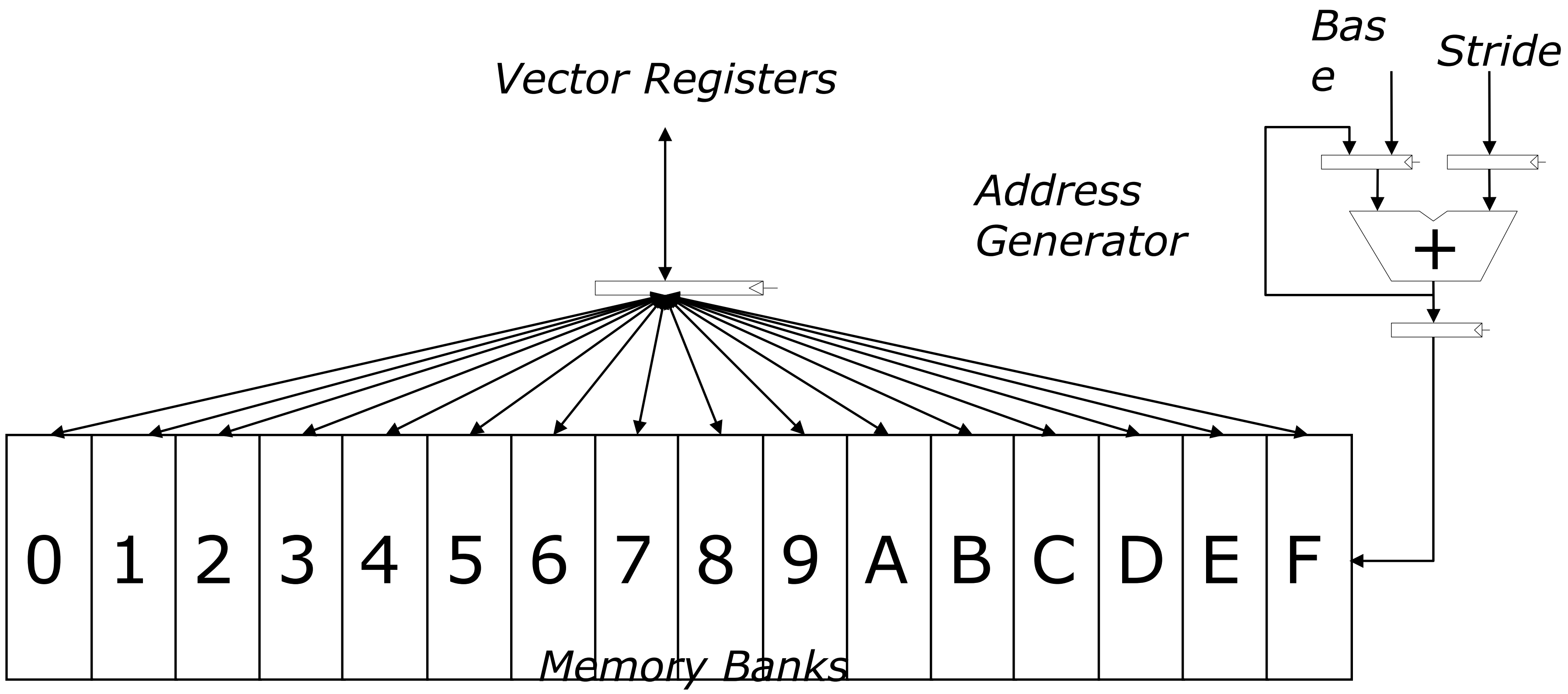
Memory Banking

- Example: 16 banks; can start one bank access per cycle
- Bank latency: 11 cycles
- Can sustain 16 parallel accesses if they go to different banks



Slide credit: Derek Chiou

Vector Memory System



Slide credit: Krste Asanovic

Scalar Code Example

- For I = 0 to 49
  - C[i] = (A[i] + B[i]) / 2

- Scalar code

```
     MOVI R0 = 50                1
     MOVA R1 = A                 1          304 dynamic instructions
     MOVA R2 = B                 1
     MOVA R3 = C                 1
X:   LD R4 = MEM[R1++]           11   ;autoincrement addressing
     LD R5 = MEM[R2++]           11
     ADD R6 = R4 + R5            4
     SHFR R7 = R6 >> 1           1
     ST MEM[R3++] = R7           11
     DECBNZ --R0, X              2    ;decrement and branch if NZ
```

Scalar Code Execution Time

- Scalar execution time on an in-order processor with 1 bank
  - First two loads in the loop cannot be pipelined: 2*11 cycles
  - 4 + 50*40 = 2004 cycles

- Scalar execution time on an in-order processor with 16 banks (word-interleaved)
  - First two loads in the loop can be pipelined
  - 4 + 50*30 = 1504 cycles

- Why 16 banks?
  - 11 cycle memory access latency
  - Having 16 (>11) banks ensures there are enough banks to overlap enough memory operations to cover memory latency

Vectorizable Loops

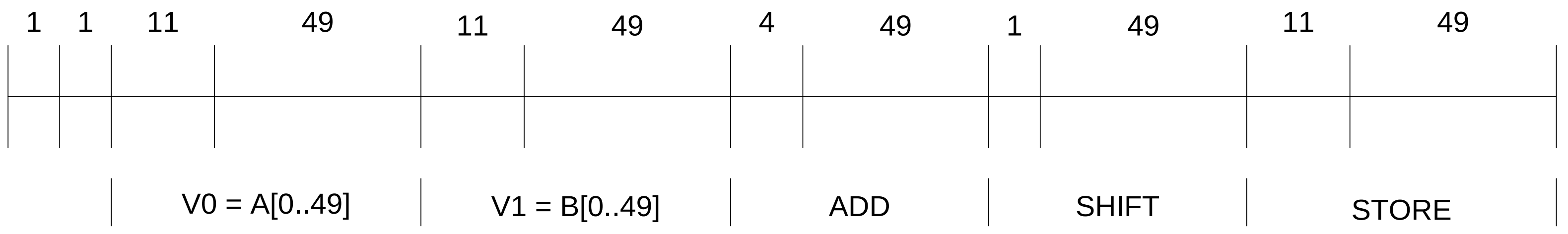
- A loop is vectorizable if each iteration is independent of any other
- For I = 0 to 49
  - C[i] = (A[i] + B[i]) / 2
- Vectorized loop:

7 dynamic instructions

MOVI VLEN = 50	1
MOVI VSTR = 1	1
VLD V0 = A	11 + VLN - 1
VLD V1 = B	11 + VLN – 1
VADD V2 = V0 + V1	4 + VLN - 1
VSHFR V3 = V2 >> 1	1 + VLN - 1
VST C = V3	11 + VLN – 1

Vector Code Performance

- No chaining
  - i.e., output of a vector functional unit cannot be used as the input of another (i.e., no vector data forwarding)
- One memory port (one address generator)
- 16 memory banks (word-interleaved)

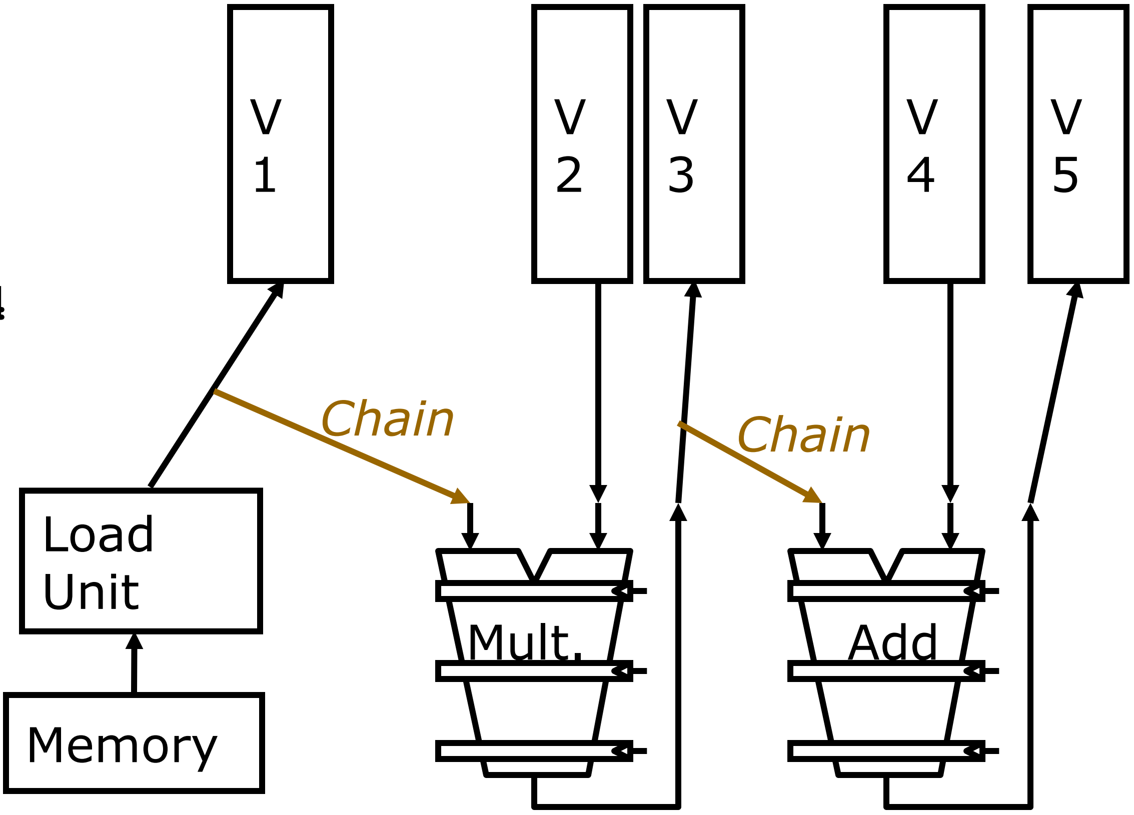


- 285 cycles

Vector Chaining

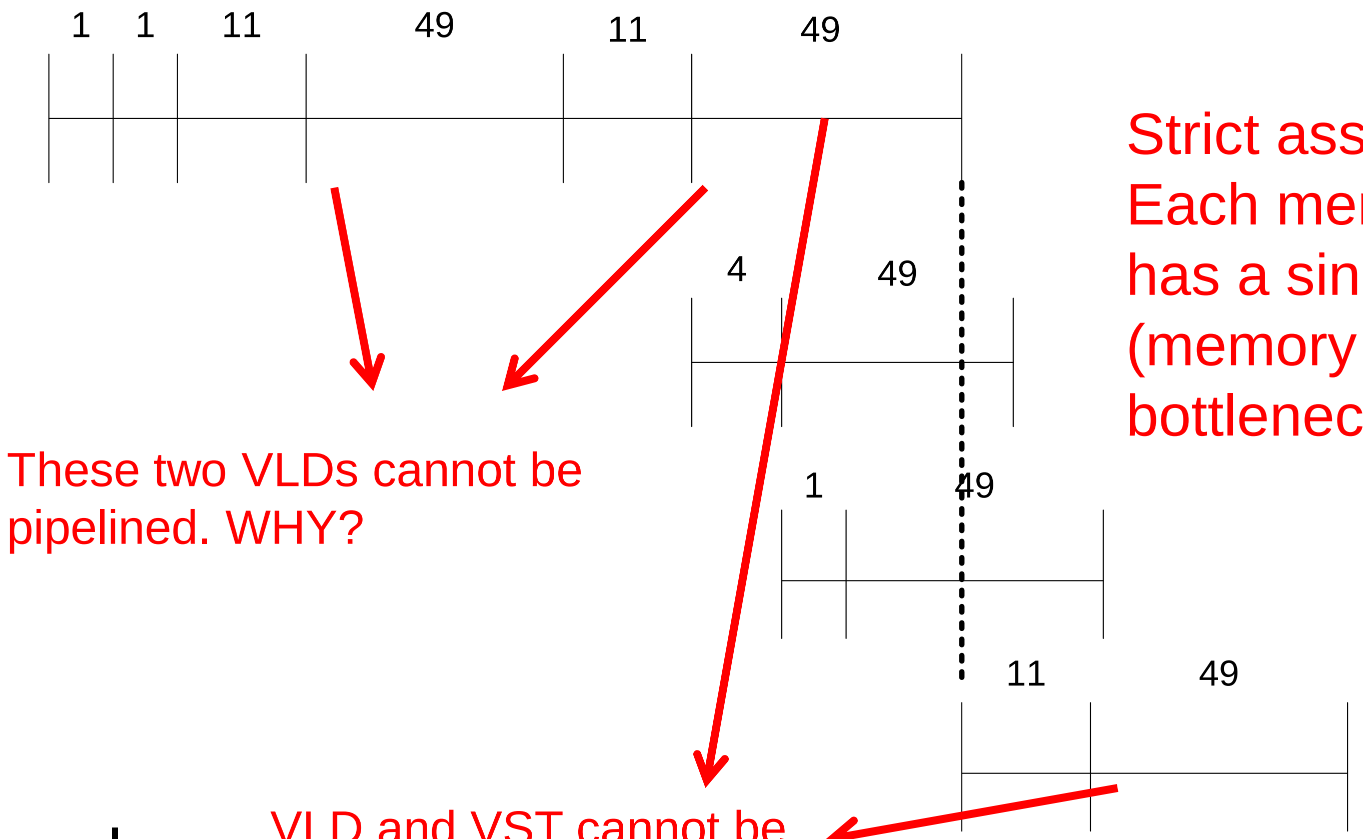
- Vector chaining: Data forwarding from one vector functional unit to another

```
LV   v1
MULV v3,v1,v2
ADDV v5, v3, v4
```



Vector Code Performance - Chaining

- Vector chaining: Data forwarding from one vector functional unit to another



These two VLDs cannot be pipelined. WHY?

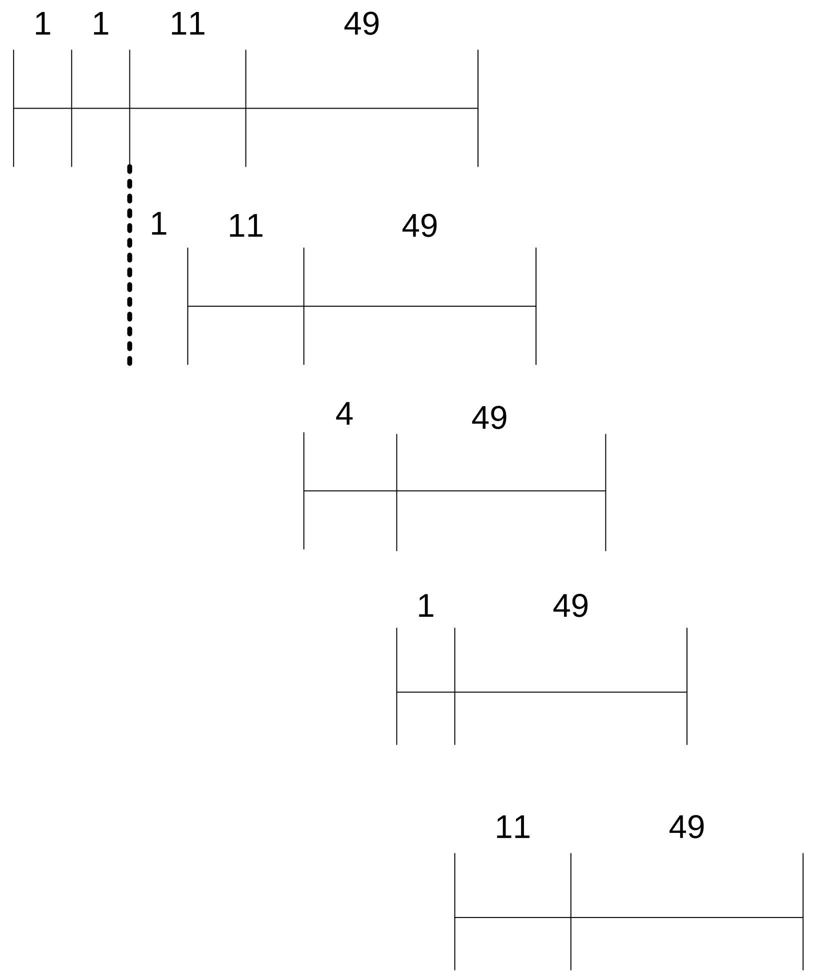
Strict assumption: Each memory bank has a single port (memory bandwidth bottleneck)

VLD and VST cannot be pipelined. WHY?

- 182 cycles

Vector Code Performance – Multiple Memory Ports

- Chaining and 2 load ports, 1 store port in each bank



- 79 cycles

Questions (I)

- What if # data elements > # elements in a vector register?
  - Need to break loops so that each iteration operates on # elements in a vector register
    - E.g., 527 data elements, 64-element VREGs
    - 8 iterations where VLEN = 64
    - 1 iteration where VLEN = 15 (need to change value of VLEN)
  - Called vector stripmining

- What if vector data is not stored in a strided fashion in memory? (irregular memory access to a vector)
  - Use indirection to combine elements into vector registers
  - Called scatter/gather operations

Gather/Scatter Operations

Want to vectorize loops with indirect accesses:

```
for (i=0; i<N; i++)
    A[i] = B[i] + C[D[i]]
```

Indexed load instruction (*Gather*)

```
LV vD, rD         # Load indices in D vector
LVI vC, rC, vD    # Load indirect from rC base
LV vB, rB         # Load B vector
ADDV.D vA,vB,vC   # Do add
SV vA, rA         # Store result
```

Gather/Scatter Operations

- Gather/scatter operations often implemented in hardware to handle sparse matrices
- Vector loads and stores use an index vector which is added to the base register to generate the addresses

Index Vector	Data Vector	Equivalent
1	3.14	3.14
3	6.5	0.0
7	71.2	6.5
8	2.71	0.0
		0.0
		0.0
		0.0
		71.2
		2.7

Conditional Operations in a Loop

- What if some operations should not be executed on a vector (based on a dynamically-determined condition)?

```
loop:       if (a[i] != 0) then b[i]=a[i]*b[i]
            goto loop
```

- Idea: Masked operations
  - VMASK register is a bit mask determining which data element should not be acted upon

    ```
    VLD V0 = A
    VLD V1 = B
    VMASK = (V0 != 0)
    VMUL V1 = V0 * V1
    VST B = V1
    ```

  - Does this look familiar? This is essentially predicated execution.

Another Example with Masking

```
for (i = 0; i < 64; ++i)
    if (a[i] >= b[i]) then c[i] = a[i]
    else c[i] = b[i]
```

A	B	VMASK
1	2	0
2	2	1
3	2	1
4	10	0
-5	-4	0
0	-3	1
6	5	1
-7	-8	1

Steps to execute loop

1. Compare A, B to get VMASK
2. Masked store of A into C
3. Complement VMASK
4. Masked store of B into C

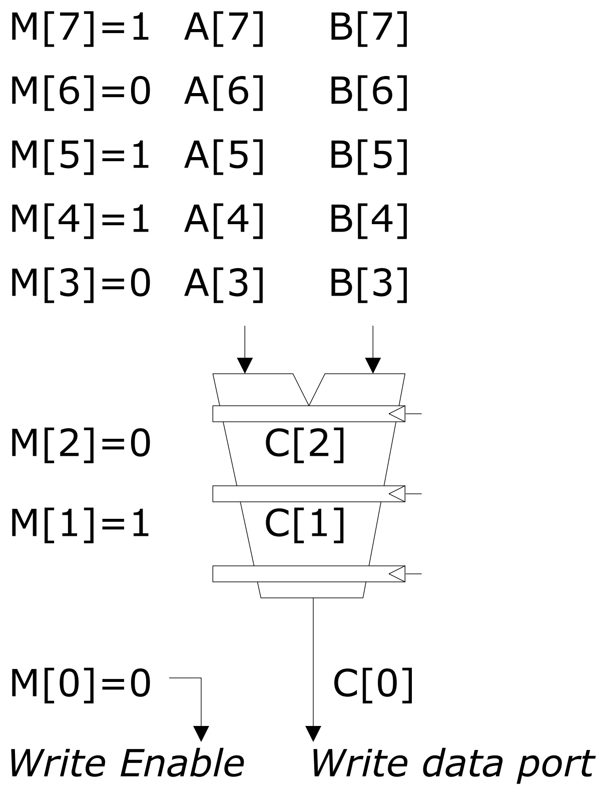
Masked Vector Instructions

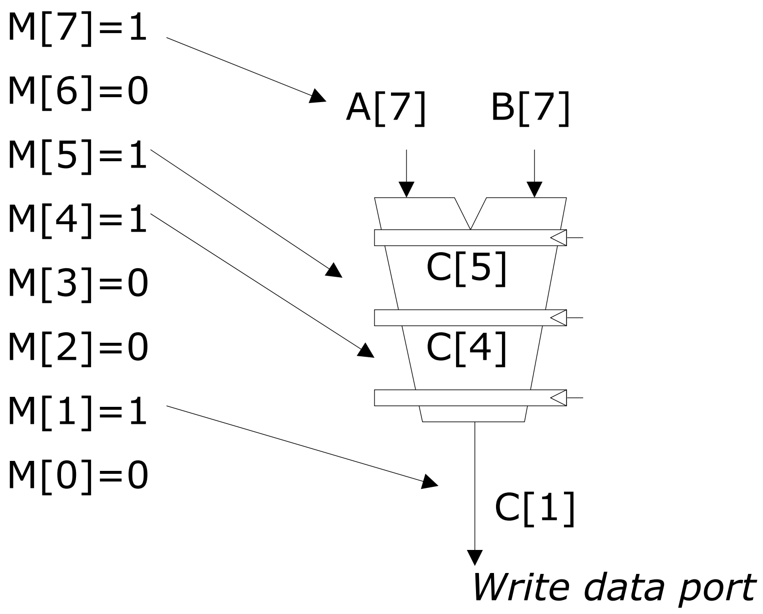
Simple Implementation

– execute all N operations, turn off result writeback according to mask

Density-Time Implementation

– scan mask vector and only execute elements with non-zero masks





Slide credit: Krste Asanovic

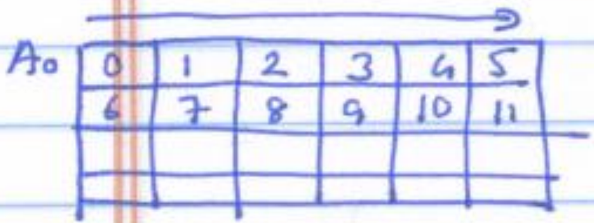
Some Issues

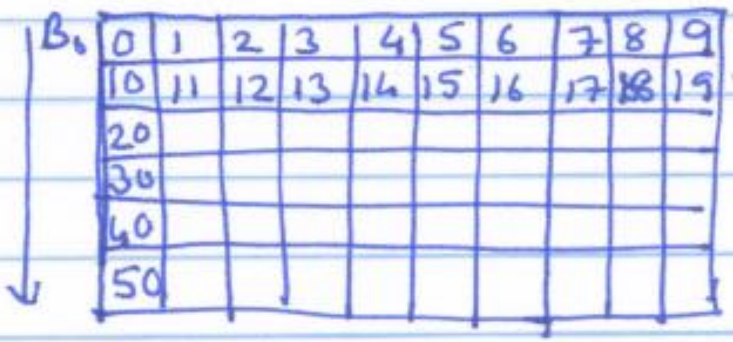
- Stride and banking
  - As long as they are relatively prime to each other and there are enough banks to cover bank access latency, consecutive accesses proceed in parallel

- Storage of a matrix
  - Row major: Consecutive elements in a row are laid out consecutively in memory
  - Column major: Consecutive elements in a column are laid out consecutively in memory
  - You need to change the stride when accessing a row versus column

Matrix multiplication

A & B, both in row major order





$A_{4\times6} \cdot B_{6\times10} \rightarrow C_{4\times10}$ (dot products of rows & colums of A & B)

A: Load A0 into a vector register V1

→ each time you need to increment the address by 1 to access the next column

→ First matrix accesses have a <u>stride of 1</u>

B: Load B0 into a vector register V2

→ each time you need to increment by 10

→ <u>stride of 10</u>

Different strides can lead to bank conflicts.

→ How do you minimize them?

Array vs. Vector Processors, Revisited

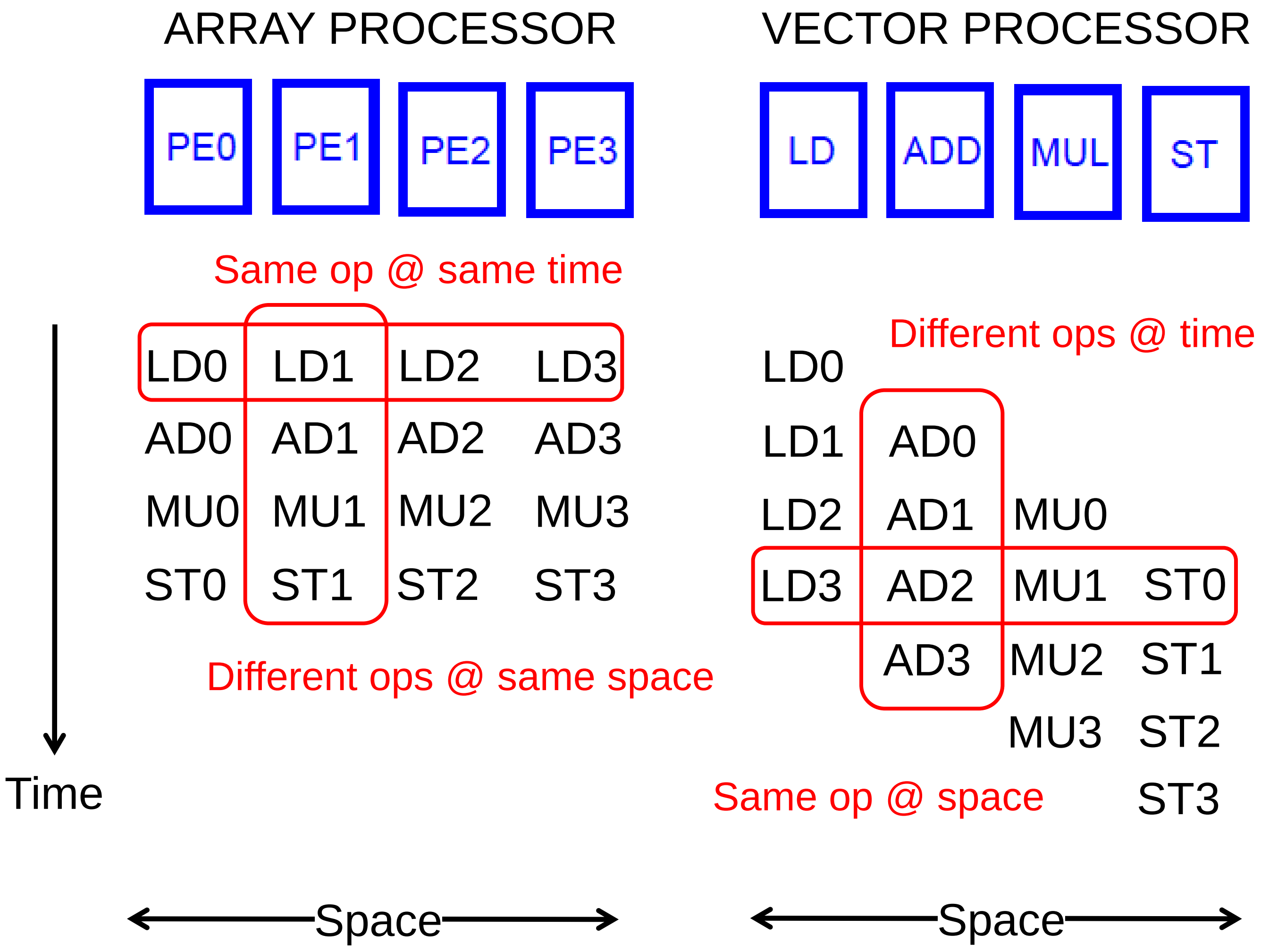
- Array vs. vector processor distinction is a "purist's" distinction

- Most "modern" SIMD processors are a combination of both
  - They exploit data parallelism in both time and space

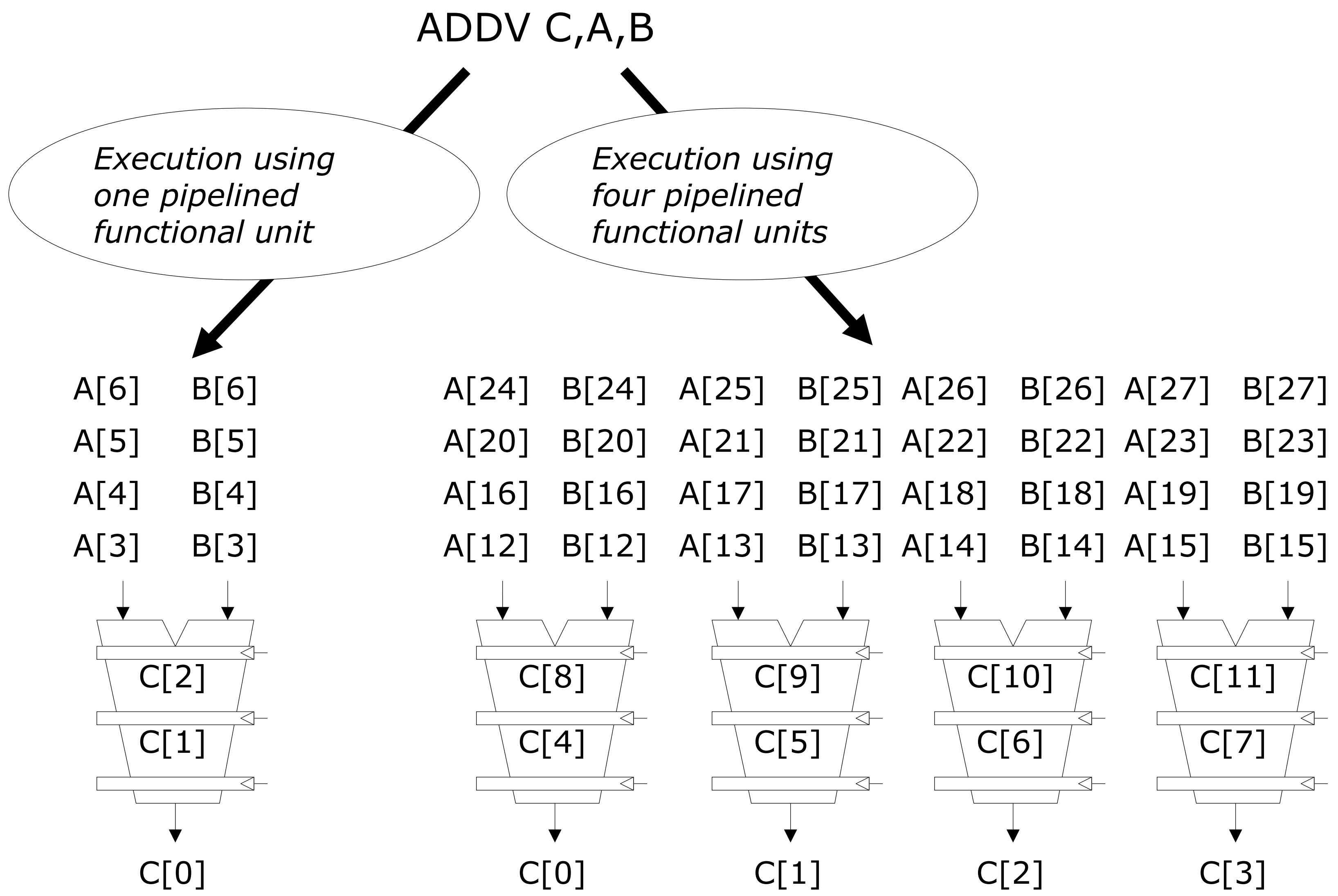
Remember: Array vs. Vector Processors

Instruction Stream

LD	VR ← A[3:0]
ADD	VR ← VR, 1
MUL	VR ← VR, 2
ST	A[3:0] ← VR



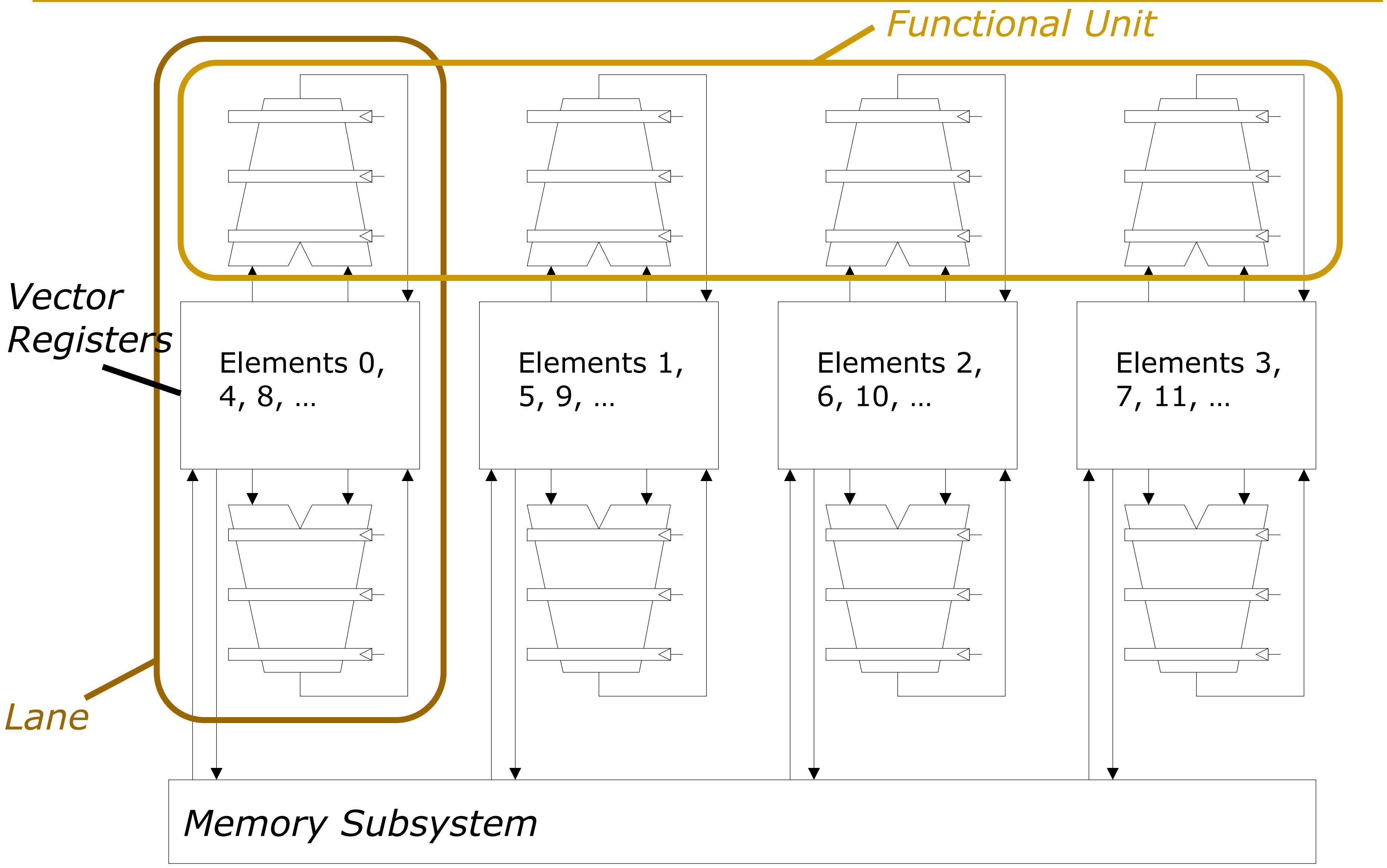
Vector Instruction Execution



ADDV C,A,B
Execution using one pipelined functional unit
Execution using four pipelined functional units
A[6] B[6]
A[5] B[5]
A[4] B[4]
A[3] B[3]
C[2]
C[1]
C[0]
A[24] B[24] A[25] B[25] A[26] B[26] A[27] B[27]
A[20] B[20] A[21] B[21] A[22] B[22] A[23] B[23]
A[16] B[16] A[17] B[17] A[18] B[18] A[19] B[19]
A[12] B[12] A[13] B[13] A[14] B[14] A[15] B[15]
C[8] C[9] C[10] C[11]
C[4] C[5] C[6] C[7]
C[0] C[1] C[2] C[3]

Slide credit: Krste Asanovic

Vector Unit Structure

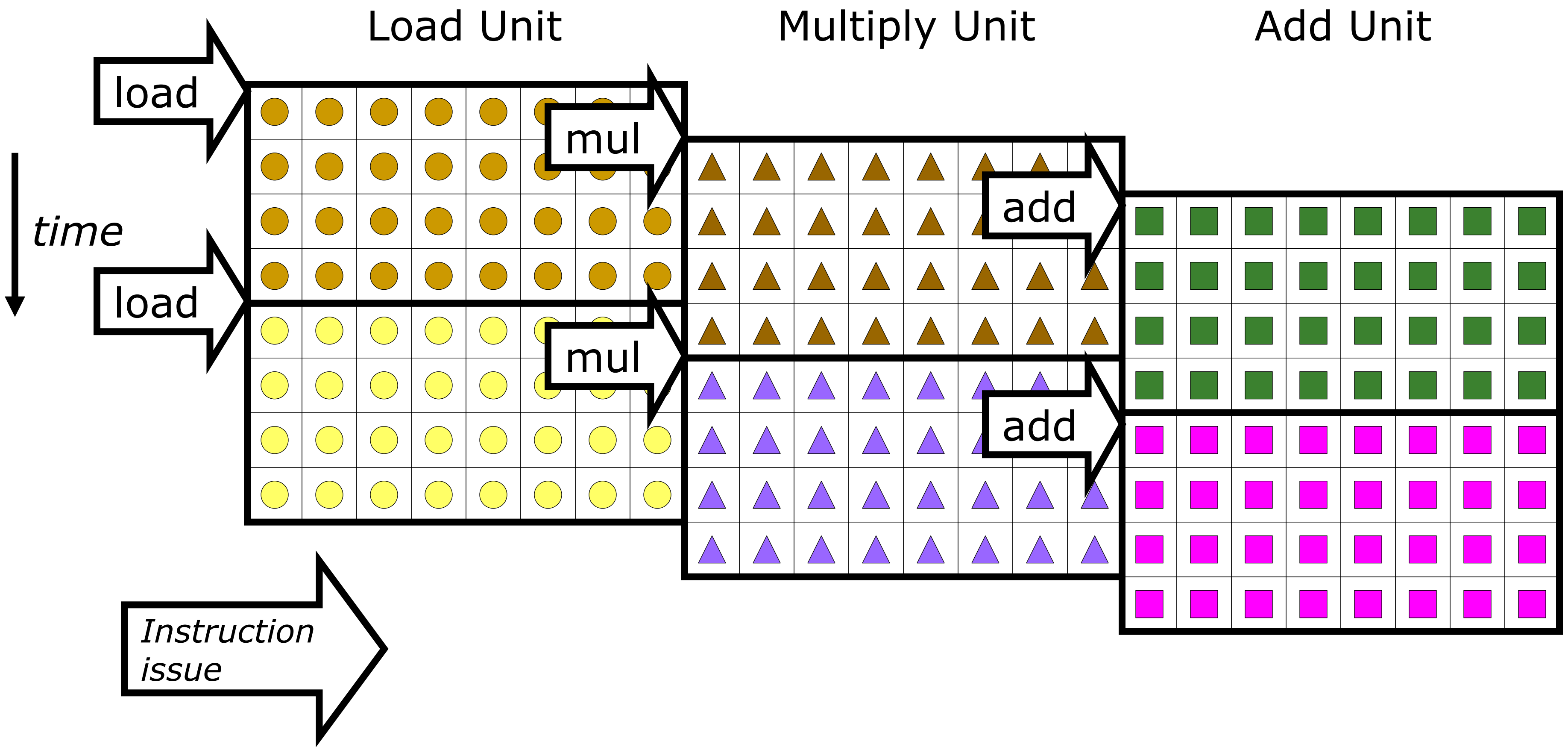


Slide credit: Krste Asanovic

Vector Instruction Level Parallelism

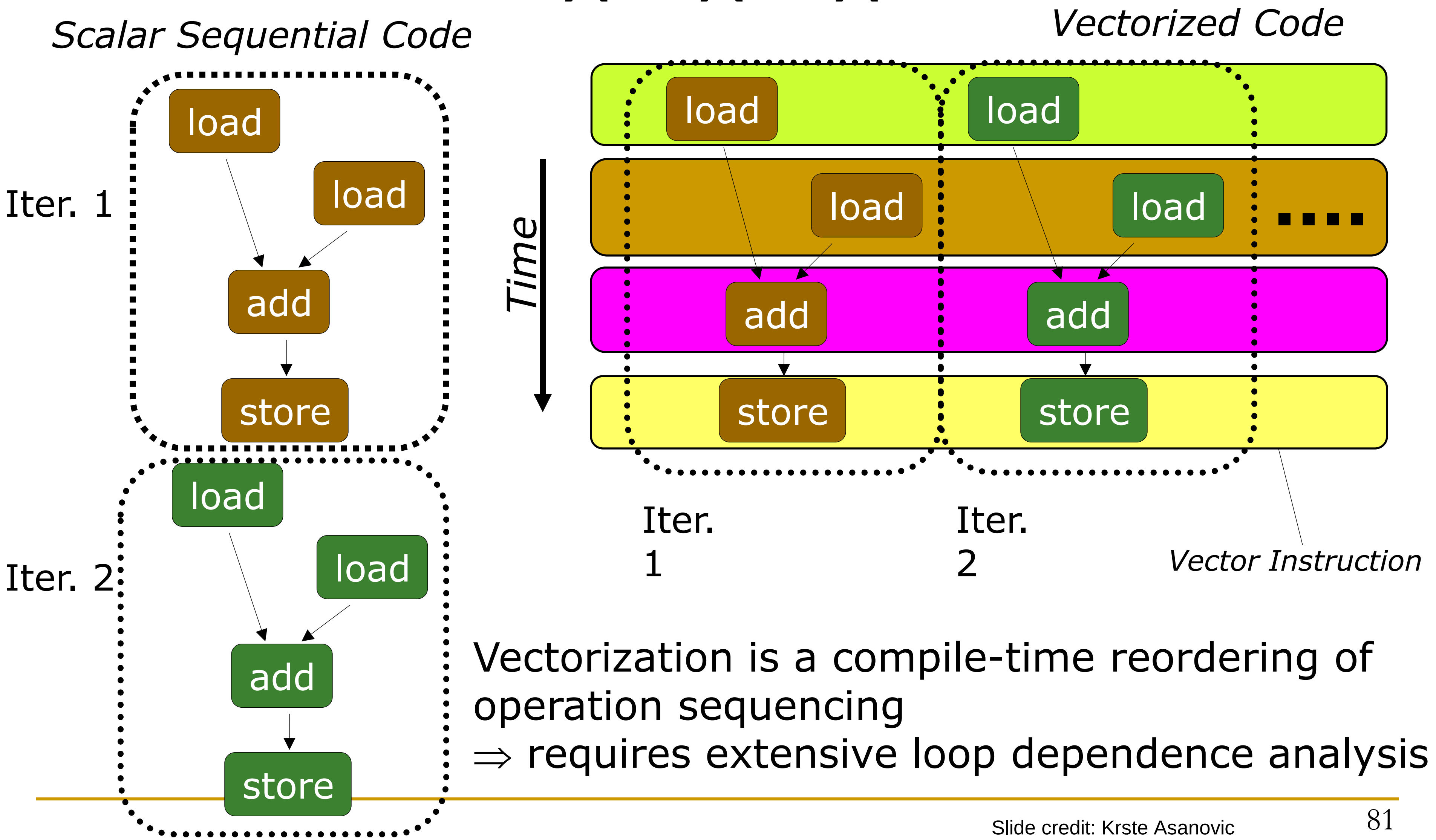
Can overlap execution of multiple vector instructions

- example machine has 32 elements per vector register and 8 lanes
- Complete 24 operations/cycle while issuing 1 short instruction/cycle



Automatic Code Vectorization

```
for (i=0; i < N; i++)
    C[i] = A[i] + B[i];
```



Vectorization is a compile-time reordering of operation sequencing
⇒ requires extensive loop dependence analysis

Slide credit: Krste Asanovic

Vector/SIMD Processing Summary

- Vector/SIMD machines good at exploiting regular data-level parallelism
  - Same operation performed on many data elements
  - Improve performance, simplify design (no intra-vector dependencies)

- Performance improvement limited by vectorizability of code
  - Scalar operations limit vector machine performance
  - Amdahl's Law
  - CRAY-1 was the fastest SCALAR machine at its time!

- Many existing ISAs include (vector-like) SIMD operations
  - Intel MMX/SSEn/AVX, PowerPC AltiVec, ARM Advanced SIMD

SIMD Operations in Modern ISAs

Intel Pentium MMX Operations

- Idea: One instruction operates on multiple data elements simultaneously
  - Ala array processing (yet much more limited)
  - Designed with multimedia (graphics) operations in mind

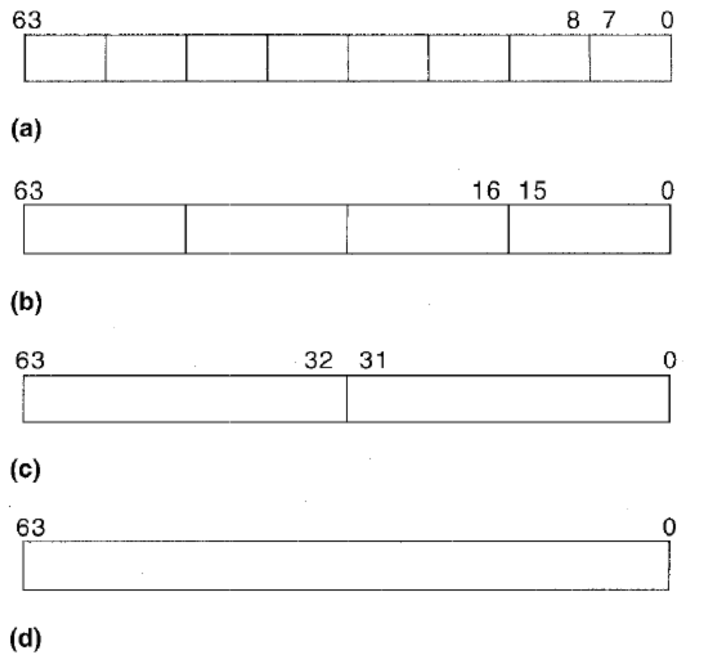


Figure 1. MMX technology data types: packed byte (a), packed word (b), packed doubleword (c), and quadword (d).

No VLEN register
Opcode determines data type:
8 8-bit bytes
4 16-bit words
2 32-bit doublewords
1 64-bit quadword

Stride always equal to 1.

Peleg and Weiser, "MMX Technology Extension to the Intel Architecture," IEEE Micro, 1996.

MMX Example: Image Overlaying (I)

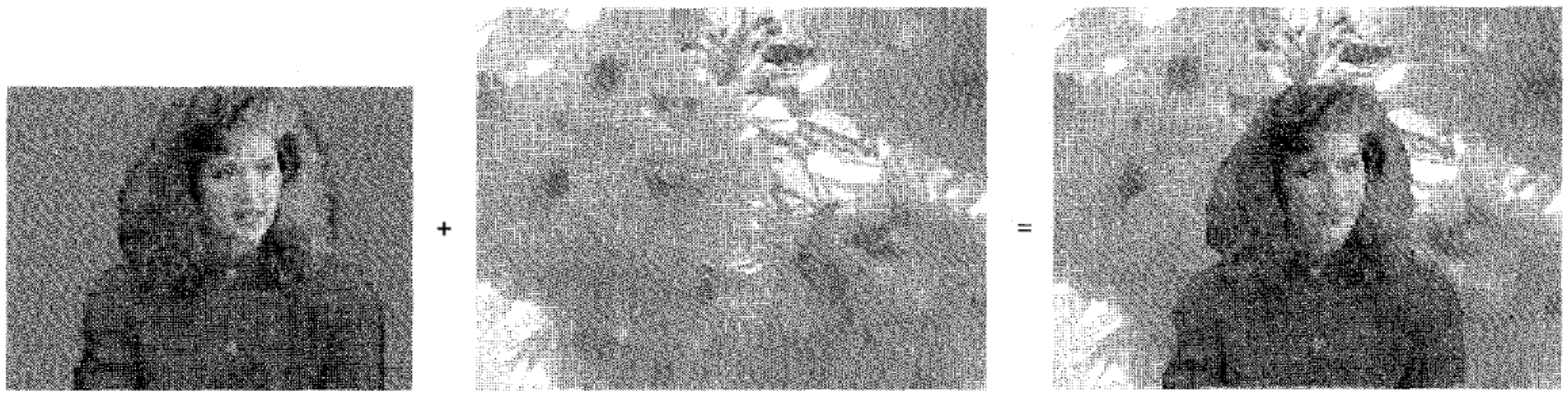
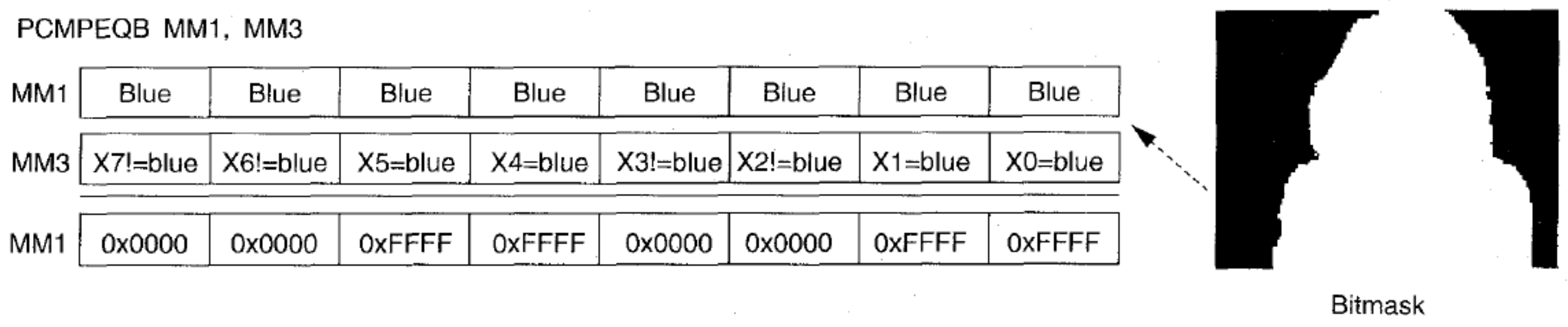


Figure 8. Chroma keying: image overlay using a background color.



PCMPEQB MM1, MM3

MM1	Blue	Blue	Blue	Blue	Blue	Blue	Blue	Blue
MM3	X7!=blue	X6!=blue	X5=blue	X4=blue	X3!=blue	X2!=blue	X1=blue	X0=blue
MM1	0x0000	0x0000	0xFFFF	0xFFFF	0x0000	0x0000	0xFFFF	0xFFFF

Figure 9. Generating the selection bit mask.

MMX Example: Image Overlaying (II)

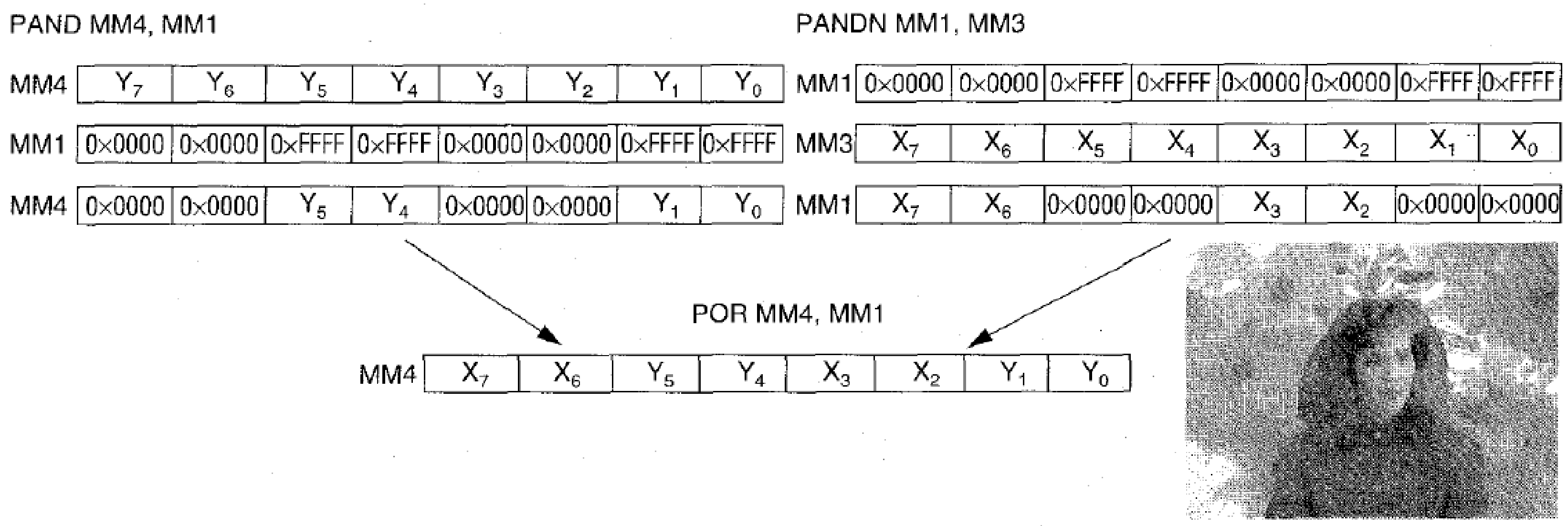

PAND MM4, MM1
MM4 | Y7 | Y6 | Y5 | Y4 | Y3 | Y2 | Y1 | Y0
MM1 | 0x0000 | 0x0000 | 0xFFFF | 0xFFFF | 0x0000 | 0x0000 | 0xFFFF | 0xFFFF
MM4 | 0x0000 | 0x0000 | Y5 | Y4 | 0x0000 | 0x0000 | Y1 | Y0
PANDN MM1, MM3
MM1 | 0x0000 | 0x0000 | 0xFFFF | 0xFFFF | 0x0000 | 0x0000 | 0xFFFF | 0xFFFF
MM3 | X7 | X6 | X5 | X4 | X3 | X2 | X1 | X0
MM1 | X7 | X6 | 0x0000 | 0x0000 | X3 | X2 | 0x0000 | 0x0000
POR MM4, MM1
MM4 | X7 | X6 | Y5 | Y4 | X3 | X2 | Y1 | Y0

Figure 10. Using the mask with logical MMX instructions to perform a conditional select.

```
Movq      mm3, mem1    /* Load eight pixels from
                          woman's image
Movq      mm4, mem2    /* Load eight pixels from the
                          blossom image
Pcmpeqb   mm1, mm3
Pand      mm4, mm1
Pandn     mm1, mm3
Por       mm4, mm1
```

Figure 11. MMX code sequence for performing a conditional select.