18-447
Computer Architecture
Lecture 17: GPUs, VLIW, Systolic Arrays

Prof. Onur Mutlu
Carnegie Mellon University
Spring 2014, 2/26/2014

Midterm I Next Week

- March 5, during class time
- We will likely take the entire 2 hours: 12:30-2:30pm
- Please come early

- Closed book, closed notes
- No electronic devices allowed
- You can bring one letter-sized cheat sheet

- The exam will test understanding, not memorization

Lab 4 Reminder

- Lab 4a out
  - Branch handling and branch predictors

- Lab 4b out
  - Fine-grained multithreading

- Due March 21st

- You have 4 weeks!
- Get started very early – Exam and S. Break are on the way
- Finish Lab 4a first and check off
- Finish Lab 4b next and check off
- Do the extra credit

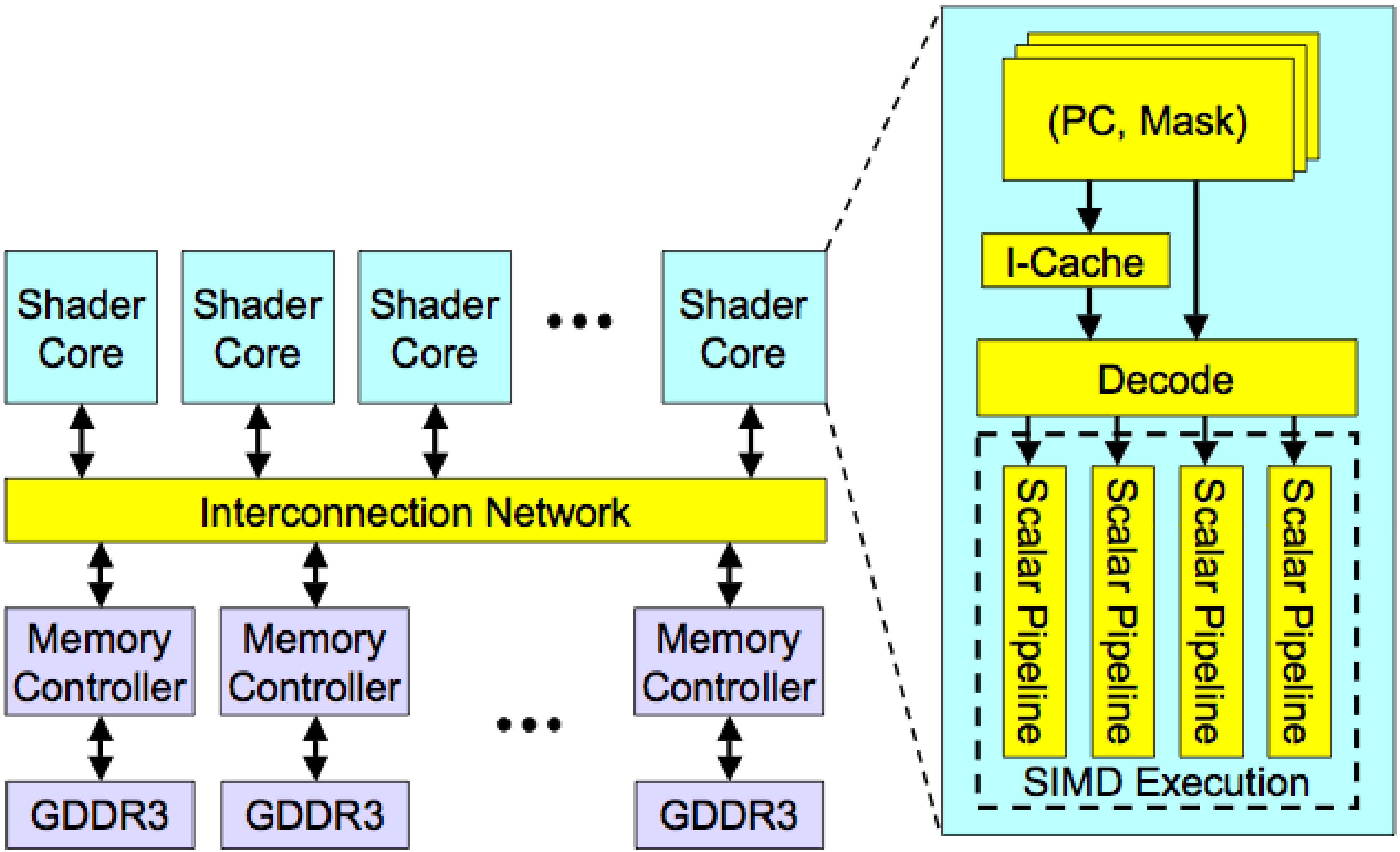
GPU Readings

- Required
  - Lindholm et al., "NVIDIA Tesla: A Unified Graphics and Computing Architecture," IEEE Micro 2008.
  - Fatahalian and Houston, "A Closer Look at GPUs," CACM 2008.

- Recommended
  - Narasiman et al., "Improving GPU Performance via Large Warps and Two-Level Warp Scheduling," MICRO 2011.
  - Fung et al., "Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow," MICRO 2007.
  - Jog et al., "Orchestrated Scheduling and Prefetching for GPGPUs," ISCA 2013.

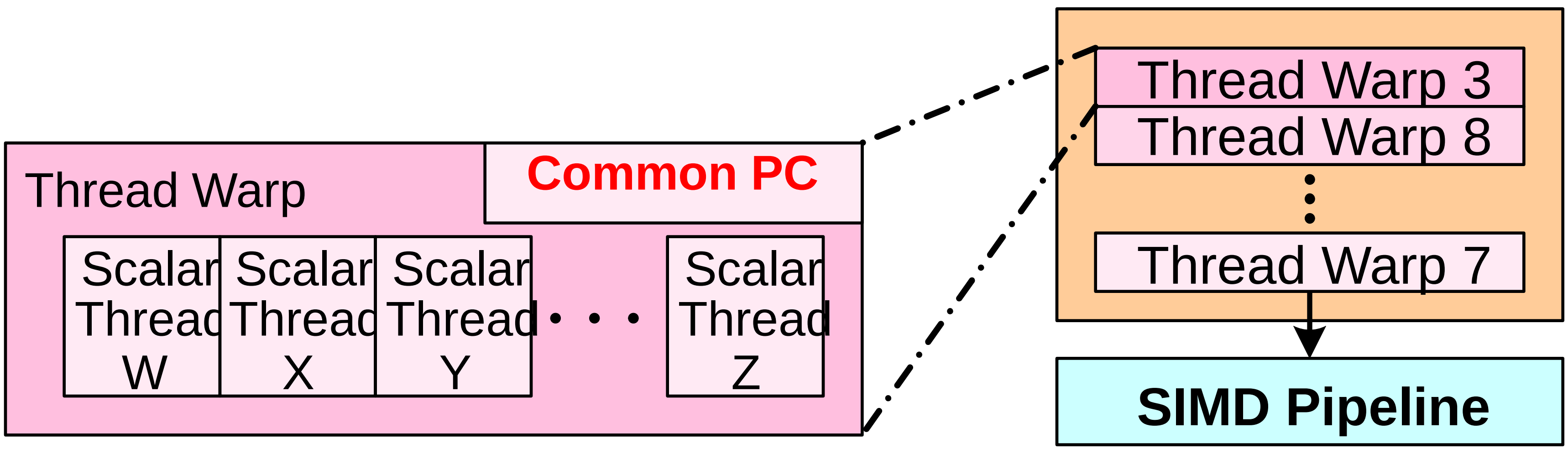
Graphics Processing Units
SIMD not Exposed to Programmer (SIMT)

High-Level View of a GPU



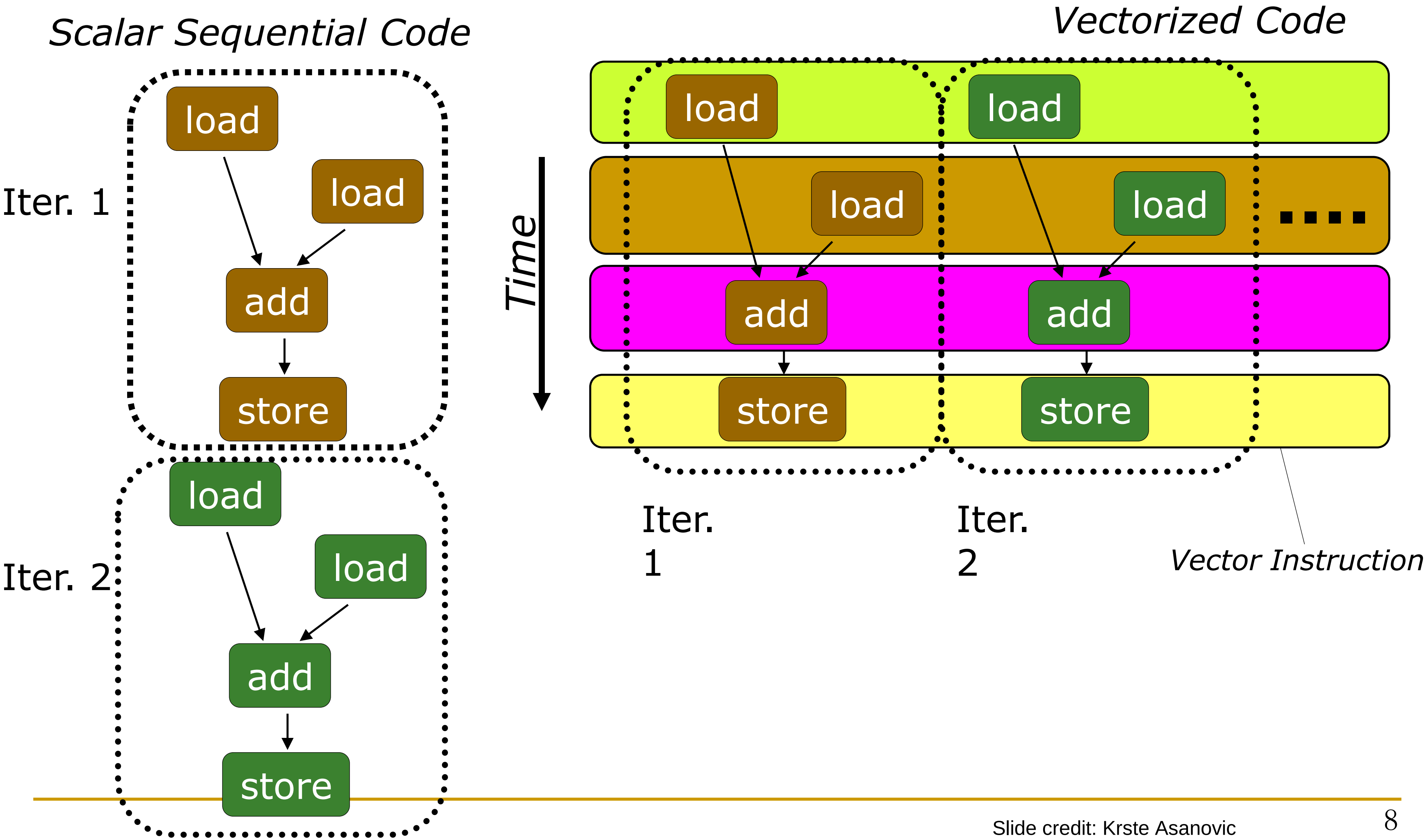
Concept of "Thread Warps" and SIMT

- Warp: A set of threads that execute the same instruction (on different data elements) → SIMT (Nvidia-speak)
- All threads run the same code
- Warp: The threads that run lengthwise in a woven fabric …



Loop Iterations as Threads

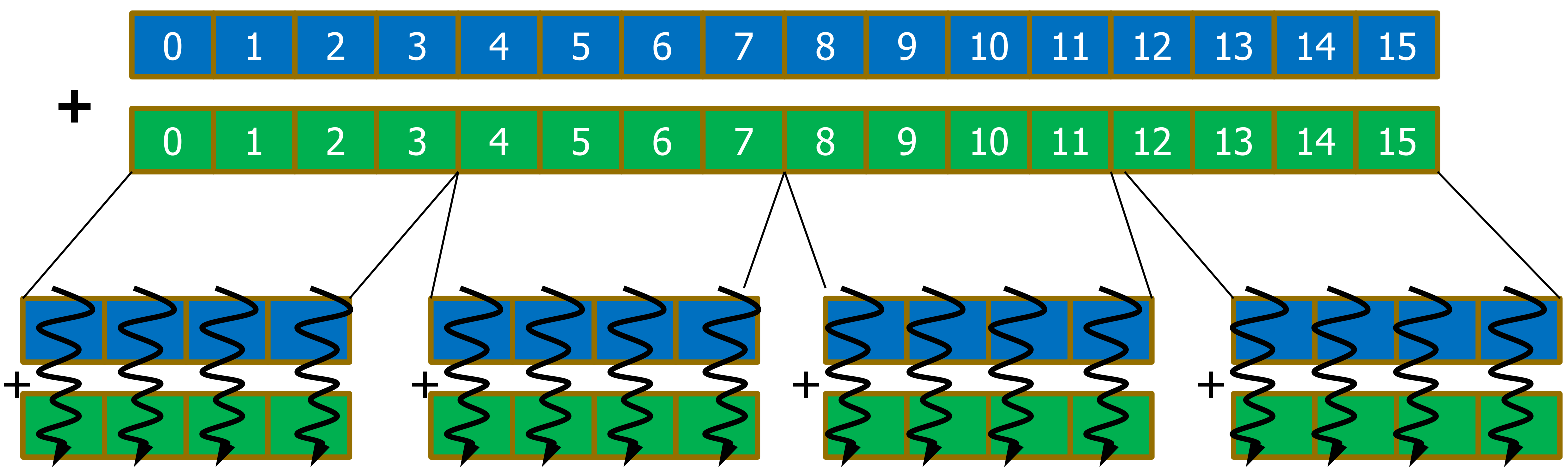
```
for (i=0; i < N; i++)
    C[i] = A[i] + B[i];
```



SIMT Memory Access

- Same instruction in different threads uses thread id to index and access different data elements

Let's assume N=16, blockDim=4 → 4 blocks



Slide credit: Hyesoon Kim

Sample GPU SIMT Code (Simplified)

CPU code

```
for (ii = 0; ii < 100; ++ii) {
C[ii] = A[ii] + B[ii];
}
```

CUDA code

```
// there are 100 threads
__global__ void KernelFunction(…) {
  int tid = blockDim.x * blockIdx.x + threadIdx.x;
  int varA = aa[tid];
  int varB = bb[tid];
  C[tid] = varA + varB;
}
```

Slide credit: Hyesoon Kim

Sample GPU Program (Less Simplified)

CPU Program

```
void add matrix
 ( float *a, float* b, float *c, int N) {
    int index;
    for (int i = 0; i < N; ++i)
       for (int j = 0; j < N; ++j) {
          index = i + j*N;
          c[index] = a[index] + b[index];
         }
 }

int main () {

   add matrix (a, b, c, N);
}
```

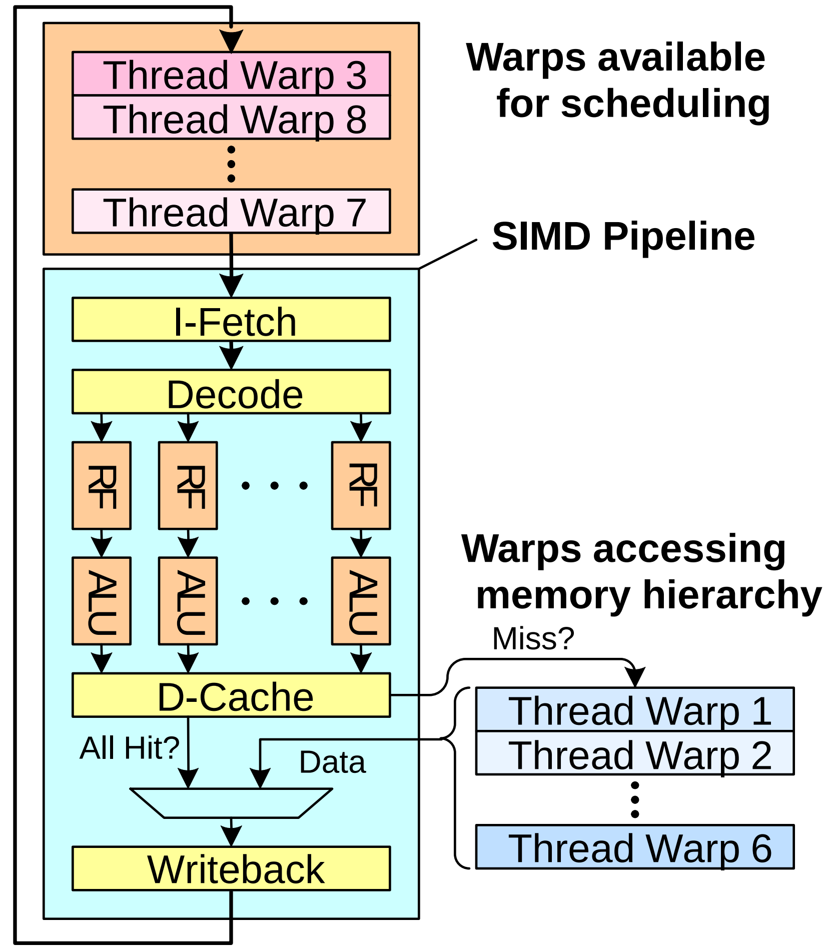
GPU Program

```
__global__  add_matrix
  ( float *a, float *b, float *c, int N) {
int i = blockIdx.x *  blockDim.x + threadIdx.x;
Int j = blockIdx.y * blockDim.y  + threadIdx.y;
int index = i + j*N;
 if (i < N && j < N)
   c[index] = a[index]+b[index];
}

Int main() {
  dim3 dimBlock( blocksize, blocksize) ;
  dim3 dimGrid (N/dimBlock.x, N/dimBlock.y);
  add_matrix<<<dimGrid, dimBlock>>>( a, b, c, N);
}
```

Latency Hiding with "Thread Warps"

- Warp: A set of threads that execute the same instruction (on different data elements)
- Fine-grained multithreading
  - One instruction per thread in pipeline at a time (No branch prediction)
  - Interleave warp execution to hide latencies
- Register values of all threads stay in register file
- FGMT enables long latency tolerance
  - Graphics has millions of pixels



Warp-based SIMD vs. Traditional SIMD

- Traditional SIMD contains a single thread
  - Lock step
  - Programming model is SIMD (no threads) → SW needs to know vector length
  - ISA contains vector/SIMD instructions

- Warp-based SIMD consists of multiple scalar threads executing in a SIMD manner (i.e., same instruction executed by all threads)
  - Does not have to be lock step
  - Each thread can be treated individually (i.e., placed in a different warp) → programming model not SIMD
    - SW does not need to know vector length
    - Enables memory and branch latency tolerance
  - ISA is scalar → vector instructions formed dynamically
  - Essentially, it is SPMD programming model implemented on SIMD hardware

SPMD

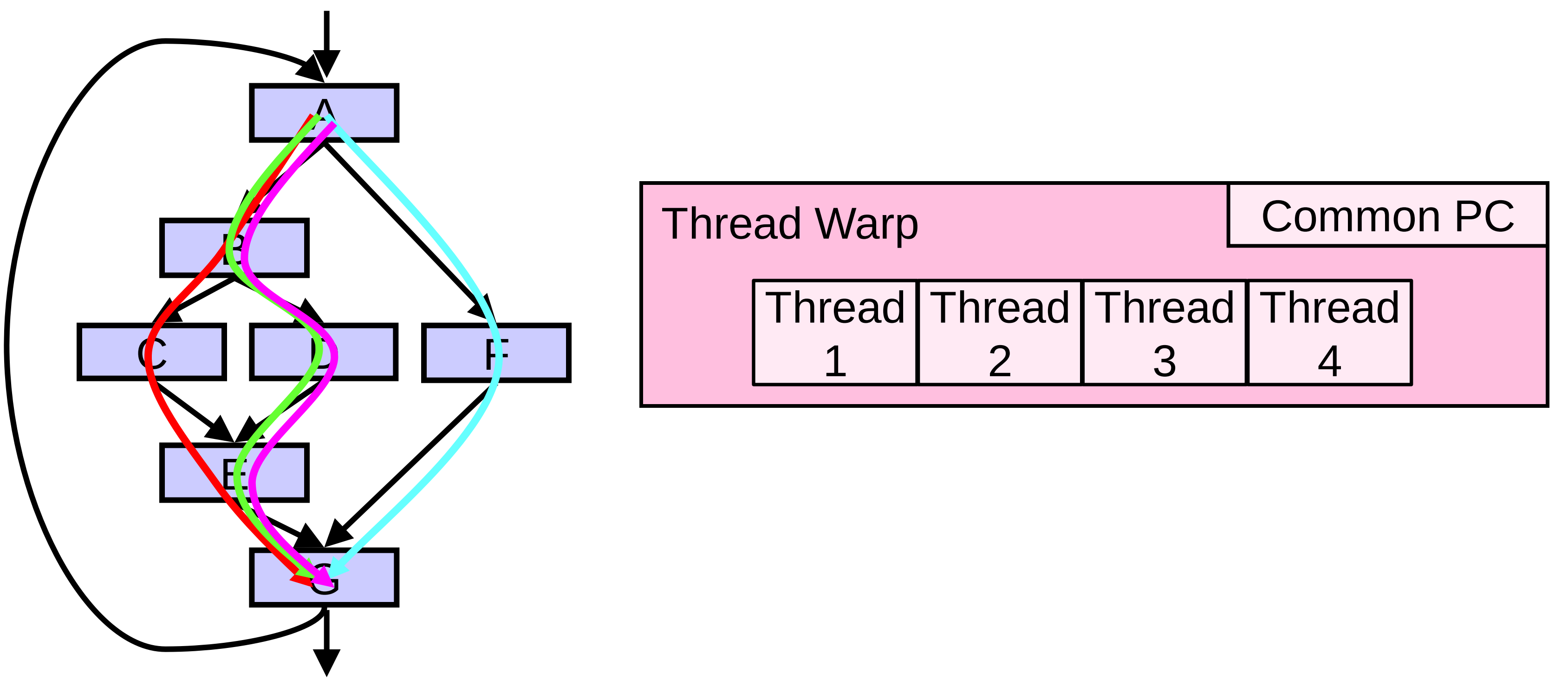
- Single procedure/program, multiple data
  - This is a programming model rather than computer organization

- Each processing element executes the same procedure, except on different data elements
  - Procedures can synchronize at certain points in program, e.g. barriers

- Essentially, multiple instruction streams execute the same program
  - Each program/procedure can 1) execute a different control-flow path, 2) work on different data, at run-time
  - Many scientific applications are programmed this way and run on MIMD computers (multiprocessors)
  - Modern GPUs programmed in a similar way on a SIMD computer

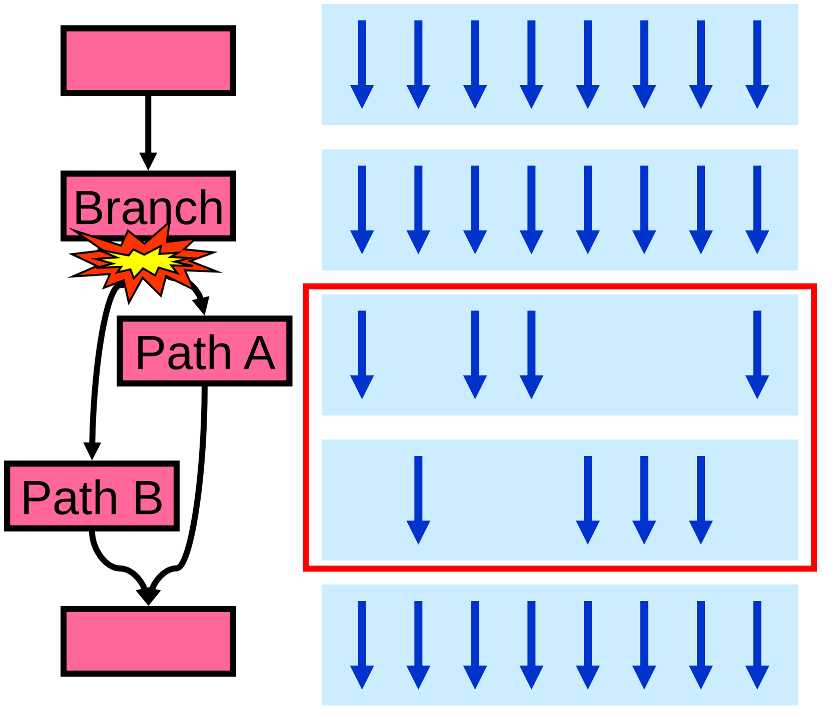
Branch Divergence Problem in Warp-based SIMD

- SPMD Execution on SIMD Hardware
  - NVIDIA calls this “Single Instruction, Multiple Thread” (“SIMT”) execution

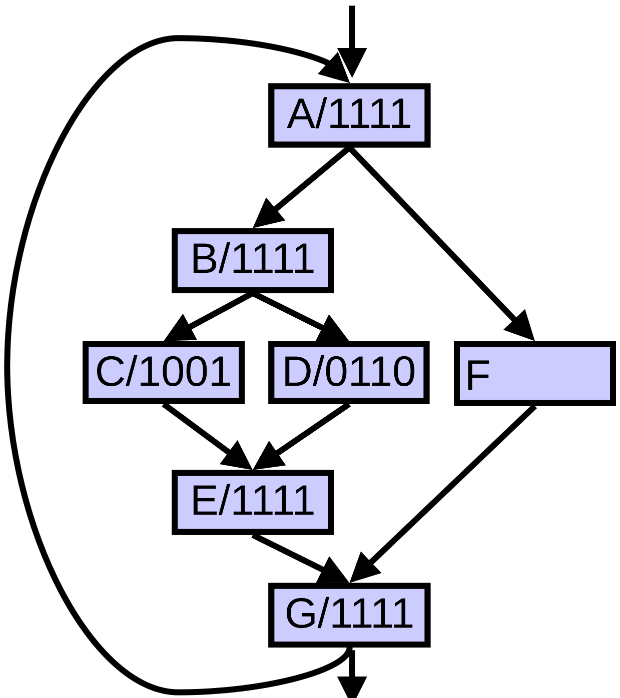


Control Flow Problem in GPUs/SIMD

- GPU uses SIMD pipeline to save area on control logic.
  - Group scalar threads into warps
- Branch divergence occurs when threads inside warps branch to different execution paths.

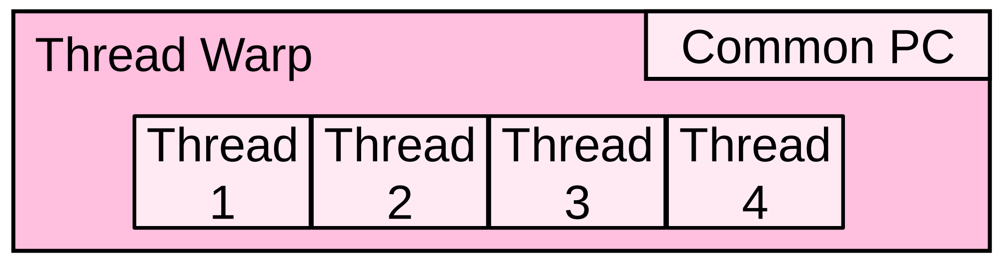


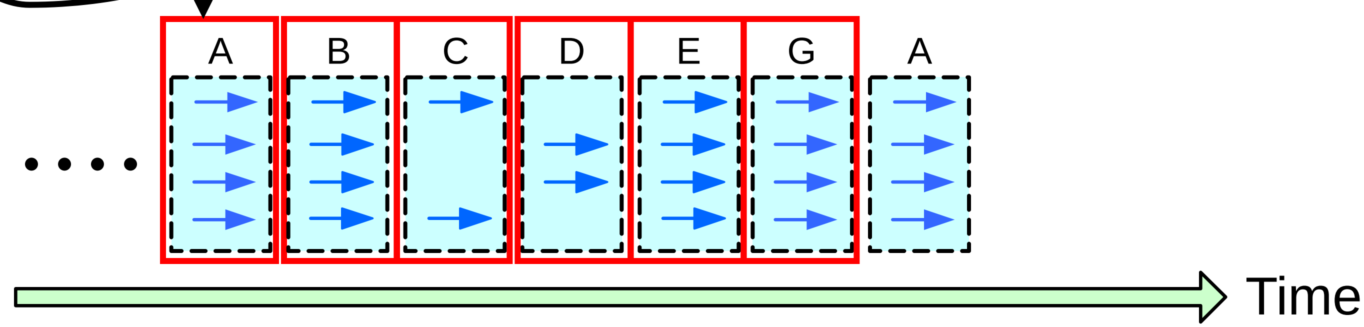
Branch Divergence Handling (I)



Stack

	Reconv. PC	Next PC	Active Mask
TOS →	-	E	1111
TOS →	E	D	0110
TOS →	E	E	1001

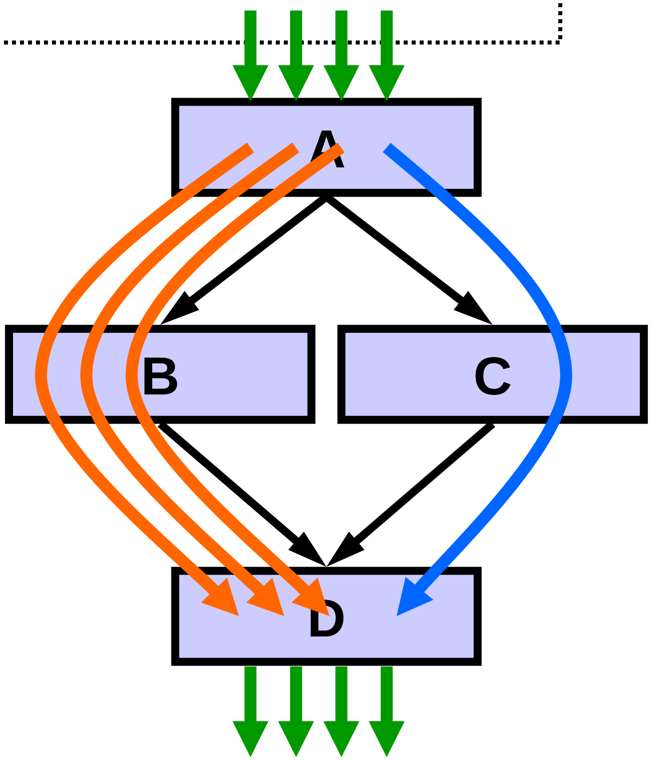




Slide credit: Tor Aamodt

Branch Divergence Handling (II)

```
A;
if (some condition) {
   B;
} else {
   C;
}
D;
```



One per warp

Control Flow Stack

	Next PC	Recv PC	Amask
TOS →	D	--	1111
	B	D	1110
	D	D	0001

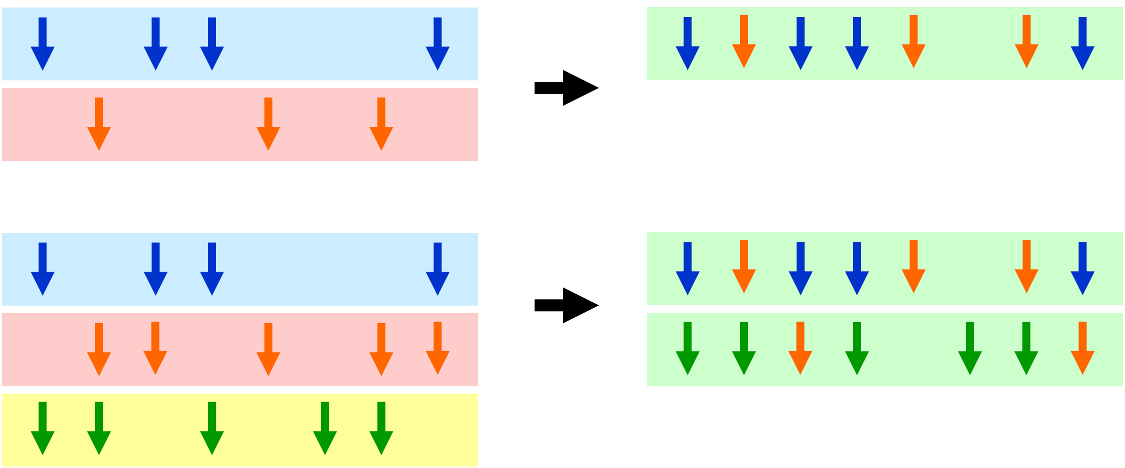
Execution Sequence

A	C	B	D
1	0	1	1
1	0	1	1
1	0	1	1
1	1	0	1

Time →

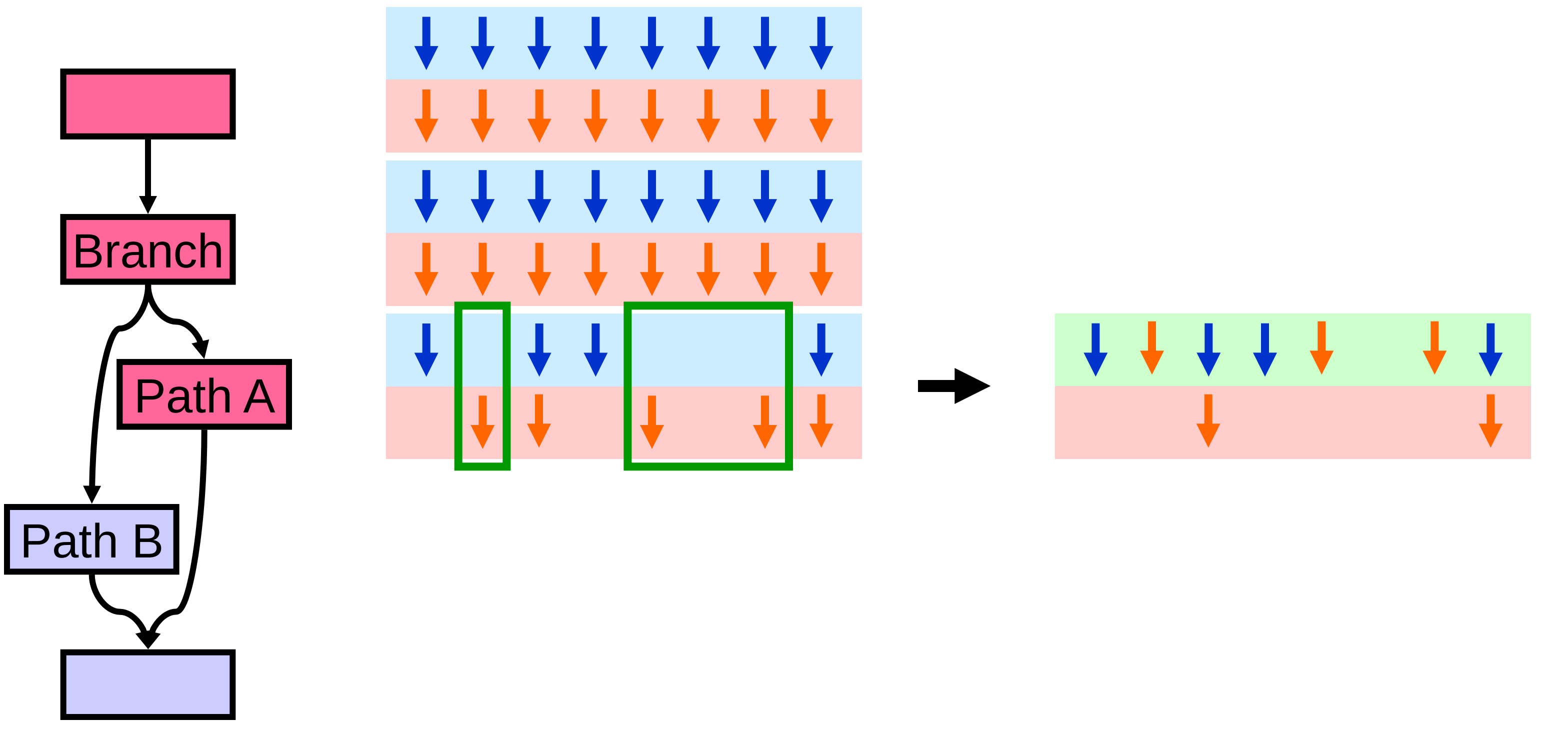
Dynamic Warp Formation/Merging

- Idea: Dynamically merge threads executing the same instruction (after branch divergence)
- Form new warp at divergence
  - Enough threads branching to each path to create full new warps



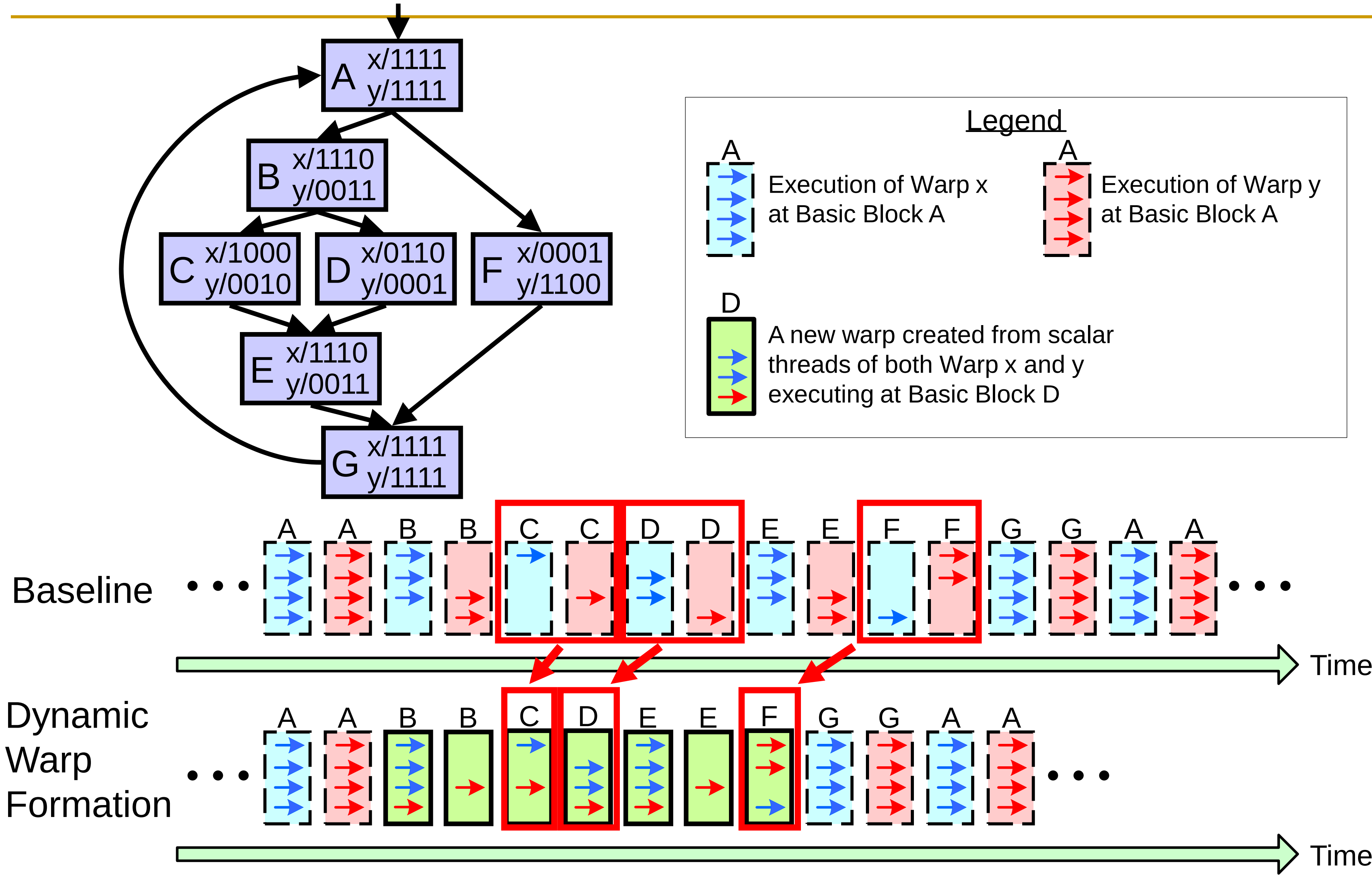
Dynamic Warp Formation/Merging

- Idea: Dynamically merge threads executing the same instruction (after branch divergence)



- Fung et al., "Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow," MICRO 2007.

Dynamic Warp Formation Example



Slide credit: Tor Aamodt

What About Memory Divergence?

- Modern GPUs have caches
- Ideally: Want all threads in the warp to hit (without conflicting with each other)
- Problem: One thread in a warp can stall the entire warp if it misses in the cache.

- Need techniques to
  - Tolerate memory divergence
  - Integrate solutions to branch and memory divergence

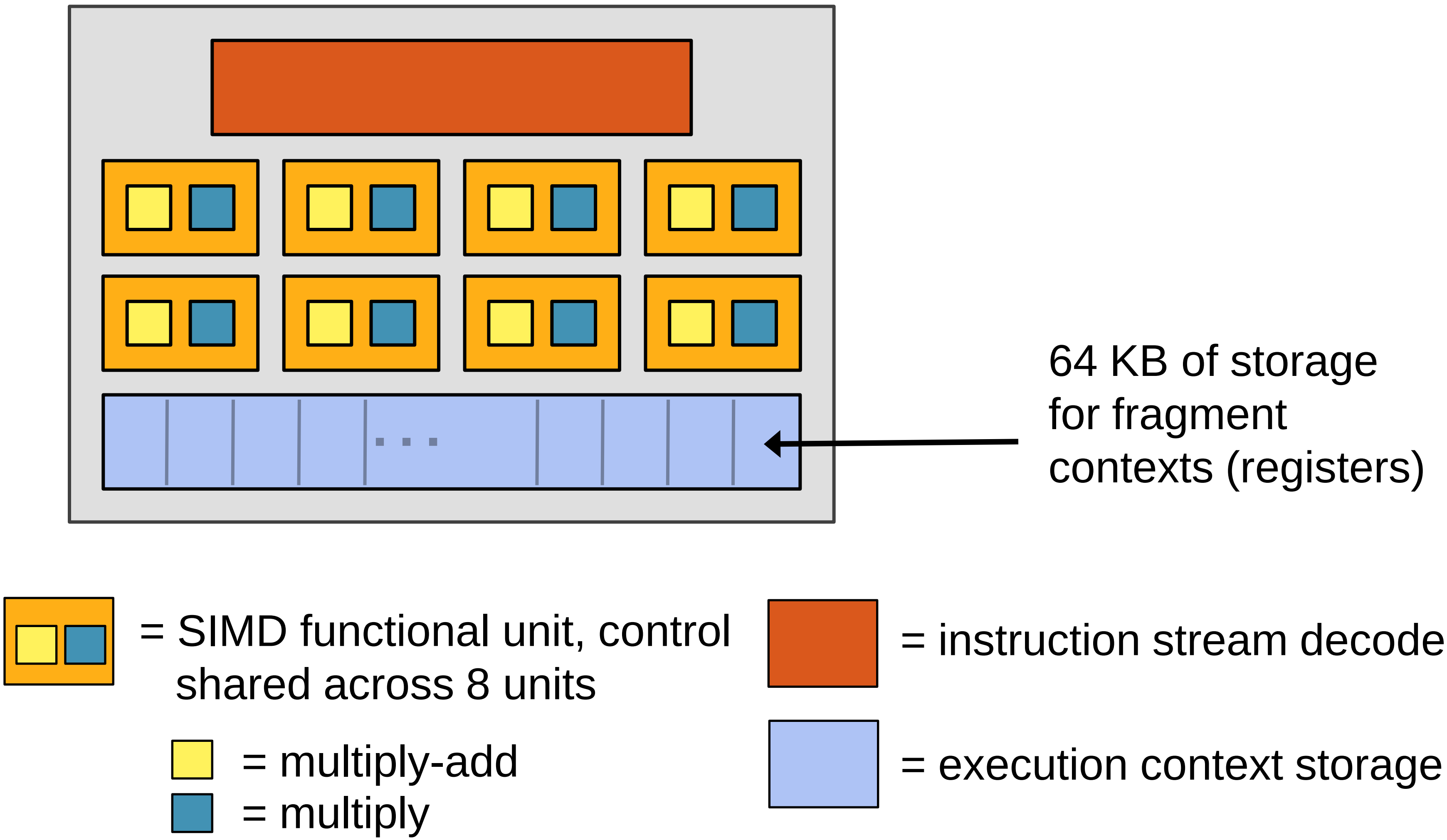
NVIDIA GeForce GTX 285

- NVIDIA-speak:
  - 240 stream processors
  - "SIMT execution"

- Generic speak:
  - 30 cores
  - 8 SIMD functional units per core

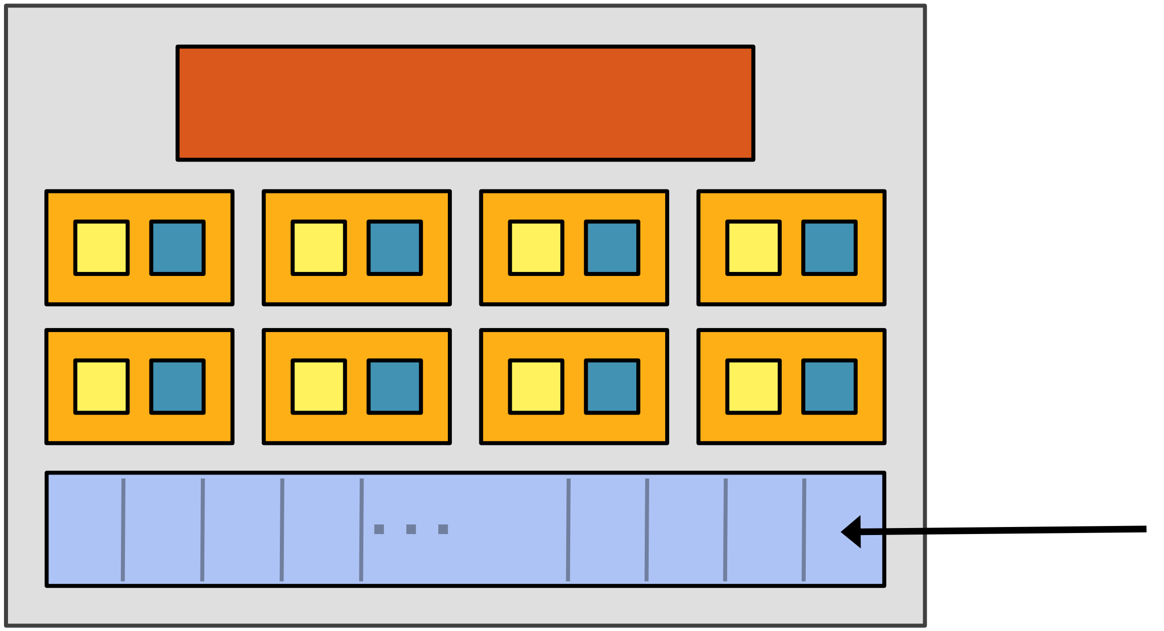
Slide credit: Kayvon Fatahalian

NVIDIA GeForce GTX 285 “core”



Slide credit: Kayvon Fatahalian

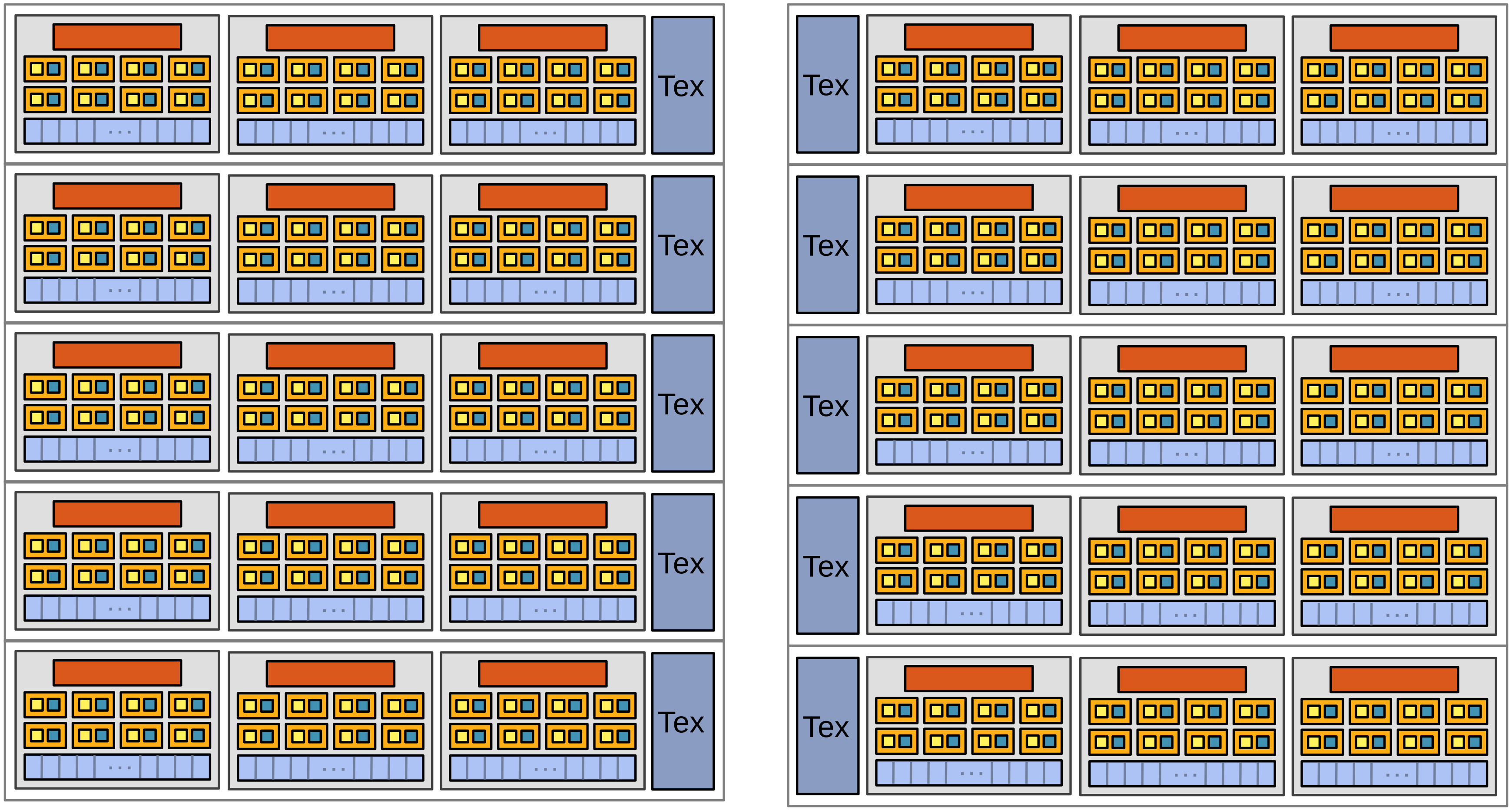
NVIDIA GeForce GTX 285 “core”



- Groups of 32 threads share instruction stream (each group is a Warp)
- Up to 32 warps are simultaneously interleaved
- Up to 1024 thread contexts can be stored

Slide credit: Kayvon Fatahalian

NVIDIA GeForce GTX 285



30 cores on the GTX 285: 30,720 threads

Slide credit: Kayvon Fatahalian

VLIW and DAE

Remember: SIMD/MIMD Classification of Computers

- Mike Flynn, "Very High Speed Computing Systems," Proc. of the 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?
- MIMD: Multiple instructions operate on multiple data elements (multiple instruction streams)
  - Multiprocessor
  - Multithreaded processor

SISD Parallelism Extraction Techniques

- We have already seen
  - Superscalar execution
  - Out-of-order execution

- Are there simpler ways of extracting SISD parallelism?
  - VLIW (Very Long Instruction Word)
  - Decoupled Access/Execute

VLIW

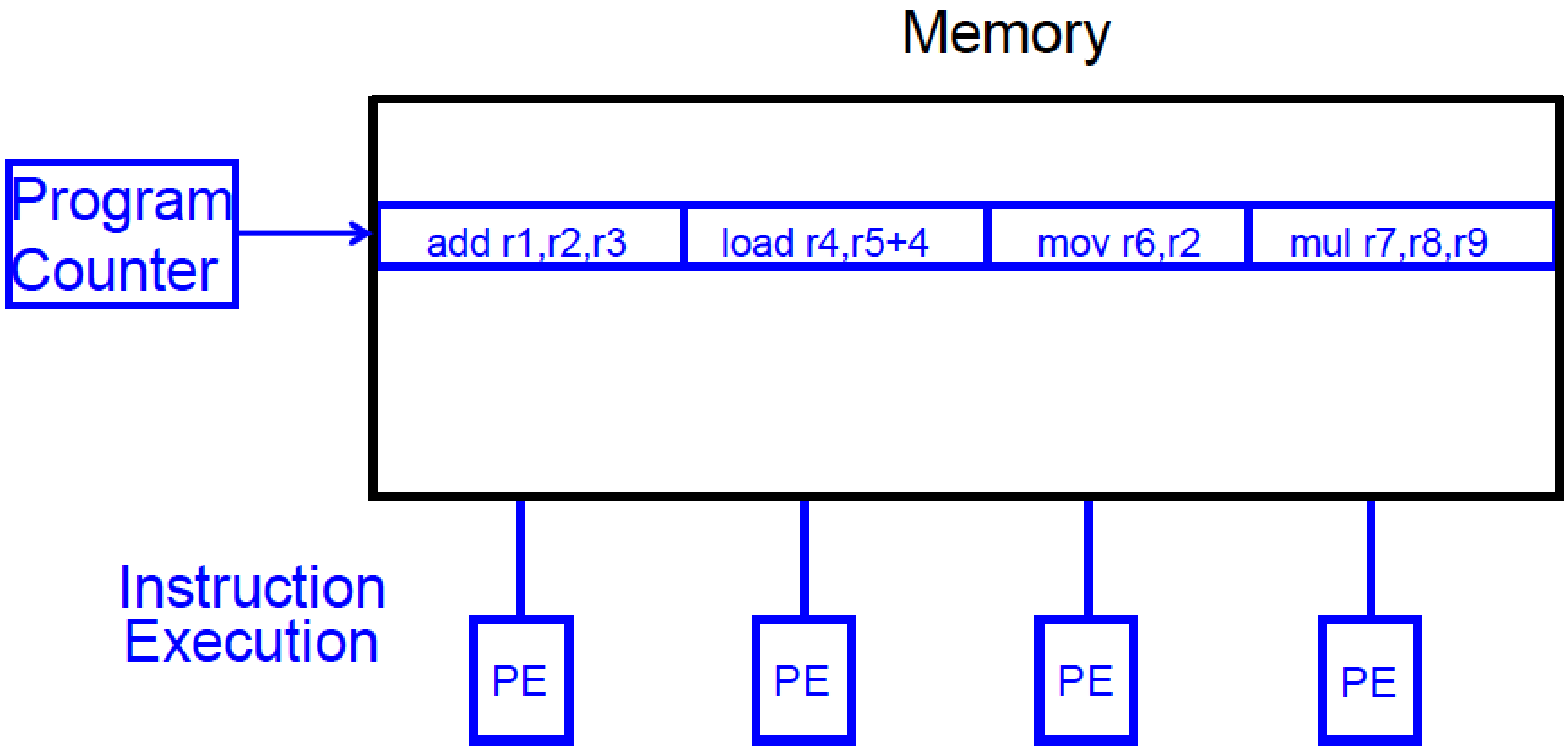
VLIW (Very Long Instruction Word)

- A very long instruction word consists of multiple independent instructions packed together by the compiler
  - Packed instructions can be logically unrelated (contrast with SIMD)

- Idea: Compiler finds independent instructions and statically schedules (i.e. packs/bundles) them into a single VLIW instruction

- Traditional Characteristics
  - Multiple functional units
  - Each instruction in a bundle executed in lock step
  - Instructions in a bundle statically aligned to be directly fed into the functional units

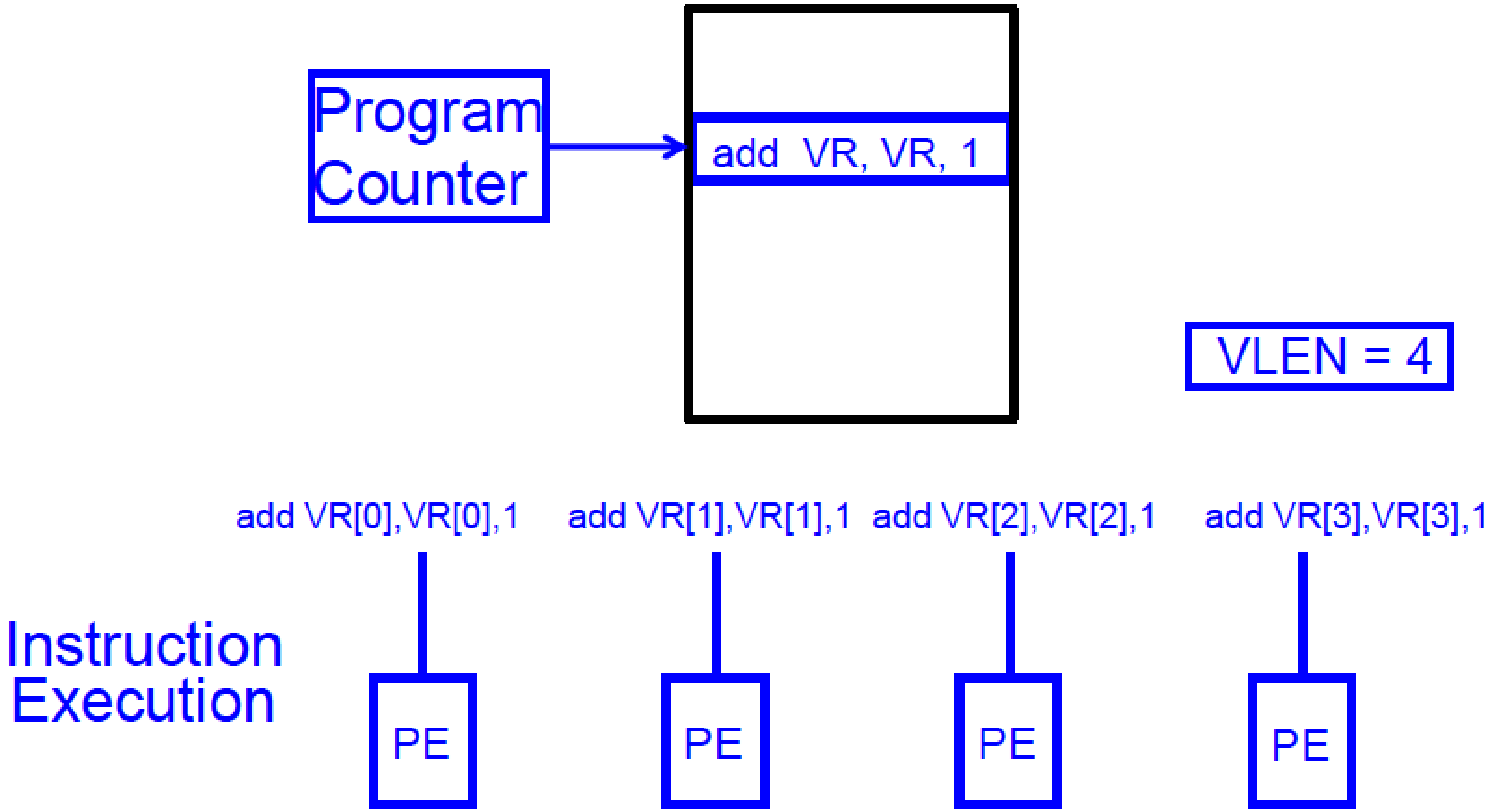
VLIW Concept



- Fisher, "Very Long Instruction Word architectures and the ELI-512," ISCA 1983.
  - ELI: Enormously longword instructions (512 bits)

SIMD Array Processing vs. VLIW

- Array processor



VLIW Philosophy

- Philosophy similar to RISC (simple instructions and hardware)
  - Except multiple instructions in parallel

- RISC (John Cocke, 1970s, IBM 801 minicomputer)
  - Compiler does the hard work to translate high-level language code to simple instructions (John Cocke: control signals)
    - And, to reorder simple instructions for high performance
  - Hardware does little translation/decoding → very simple

- VLIW (Fisher, ISCA 1983)
  - Compiler does the hard work to find instruction level parallelism
  - Hardware stays as simple and streamlined as possible
    - Executes each instruction in a bundle in lock step
    - Simple → higher frequency, easier to design

VLIW Philosophy (II)

More formally, VLIW architectures have the following properties:

There is one central control unit issuing a single long instruction per cycle.

Each long instruction consists of many tightly coupled independent operations.

Each operation requires a small, statically predictable number of cycles to execute.

Operations can be pipelined. These properties distinguish VLIWs from multiprocessors (with large asynchronous tasks) and dataflow machines (without a single flow of control, and without the tight coupling). VLIWs have none of the required regularity of a vector processor, or true array processor.

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

Commercial VLIW Machines

- Multiflow TRACE, Josh Fisher (7-wide, 28-wide)
- Cydrome Cydra 5, Bob Rau
- Transmeta Crusoe: x86 binary-translated into internal VLIW
- TI C6000, Trimedia, STMicro (DSP & embedded processors)
  - Most successful commercially

- Intel IA-64
  - Not fully VLIW, but based on VLIW principles
  - EPIC (Explicitly Parallel Instruction Computing)
  - Instruction bundles can have dependent instructions
  - A few bits in the instruction format specify explicitly which instructions in the bundle are dependent on which other ones

VLIW Tradeoffs

- Advantages
  + No need for dynamic scheduling hardware → simple hardware
  + No need for dependency checking within a VLIW instruction → simple hardware for multiple instruction issue + no renaming
  + No need for instruction alignment/distribution after fetch to different functional units → simple hardware

- Disadvantages
  -- Compiler needs to find N independent operations
    -- If it cannot, inserts NOPs in a VLIW instruction
    -- Parallelism loss AND code size increase
  -- Recompilation required when execution width (N), instruction latencies, functional units change (Unlike superscalar processing)
  -- Lockstep execution causes independent operations to stall
    -- No instruction can progress until the longest-latency instruction completes

VLIW Summary

- VLIW simplifies hardware, but requires complex compiler techniques
- Solely-compiler approach of VLIW has several downsides that reduce performance

  -- Too many NOPs (not enough parallelism discovered)

  -- Static schedule intimately tied to microarchitecture

    -- Code optimized for one generation performs poorly for next

  -- No tolerance for variable or long-latency operations (lock step)

++ Most compiler optimizations developed for VLIW employed in optimizing compilers (for superscalar compilation)

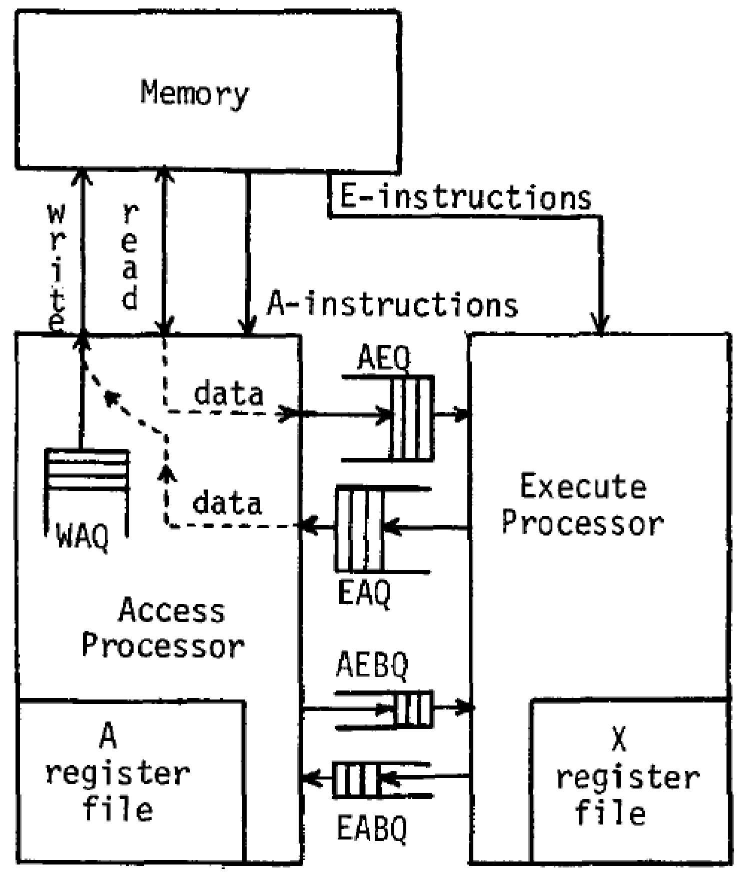
- Enable code optimizations

++ VLIW successful in embedded markets, e.g. DSP

DAE

Decoupled Access/Execute

- Motivation: Tomasulo's algorithm too complex to implement
  - 1980s before HPS, Pentium Pro
- Idea: Decouple operand access and execution via two separate instruction streams that communicate via ISA-visible queues.
- Smith, "Decoupled Access/Execute Computer Architectures," ISCA 1982, ACM TOCS 1984.



Decoupled Access/Execute (II)

- Compiler generates two instruction streams (A and E)
  - Synchronizes the two upon control flow instructions (using branch queues)

```
      q = 0.0
      Do 1  k = 1, 400
1     x(k) = q + y(k) * (r * z(k+10) + t * z(k+11))
```

Fig. 2a. Lawrence Livermore Loop 1 (HYDRO EXCERPT)

```
       A7 ← -400           . negative loop count
       A2 ← 0              . initialize index
       A3 ← 1              . index increment
       X2 ← r              . load loop invariants
       X5 ← t              . into registers
loop:  X3 ← z + 10, A2     . load z(k+10)
       X7 ← z + 11, A2     . load z(k+11)
       X4 ← X2 *f X3       . r*z(k+10)-flt. mult.
       X3 ← X5 *f X7       . t * z(k+11)
       X7 ←  y, A2         . load y(k)
       X6 ← X3 +f X4       . r*z(x+10)+t*z(k+11))
       X4 ← X7 *f X6       . y(k) * (above)
       A7 ← A7 + 1         . increment loop counter
       x, A2 ← X4          . store into x(k)
       A2 ← A2 + A3        . increment index
       JAM  loop           . Branch if A7 < 0
```

Fig. 2b. Compilation onto CRAY-1-like architecture

```
      Access                  Execute
        .
        .
        .
AEQ ← z + 10, A2         X4 ← X2 *f AEQ
AEQ ← z + 11, A2         X3 ← X5 *f AEQ
AEQ ← y, A2              X6 ← X3 +f X4
A7 ← A7 + 1             EAQ ← AEQ *f X6
x, A2 ← EAQ                         .
A2  ← A2+ A3                        .
        .                           .
        .
        .
```

Fig. 2c. Access and execute programs for straight-line section of loop

Decoupled Access/Execute (III)

- Advantages:

  + Execute stream can run ahead of the access stream and vice versa

    + If A takes a cache miss, E can perform useful work

    + If A hits in cache, it supplies data to lagging E

    + Queues reduce the number of required registers

  + Limited out-of-order execution without wakeup/select complexity

- Disadvantages:

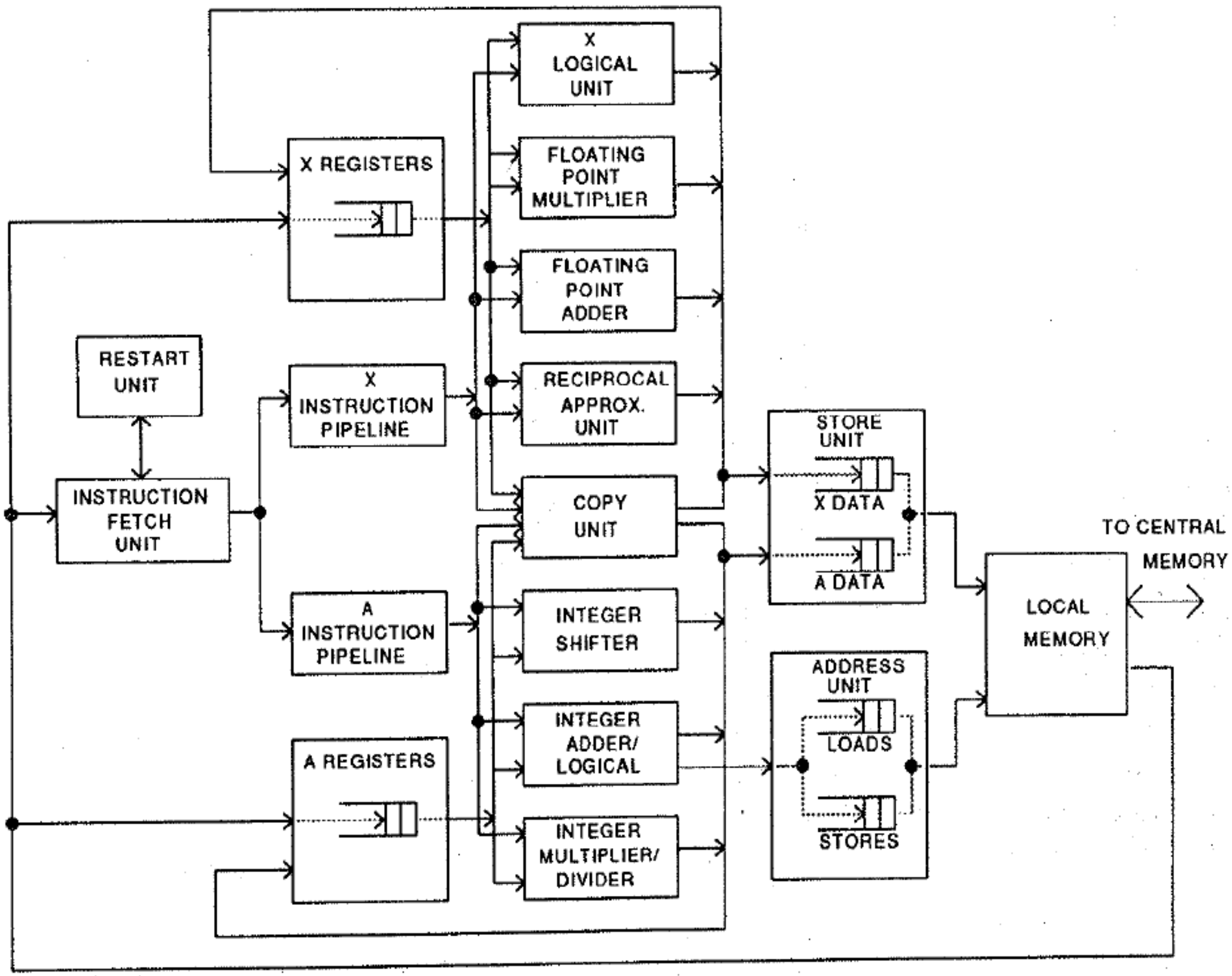
  -- Compiler support to partition the program and manage queues

    -- Determines the amount of decoupling

  -- Branch instructions require synchronization between A and E

  -- Multiple instruction streams (can be done with a single one, though)

Astronautics ZS-1



- Single stream steered into A and X pipelines
- Each pipeline in-order

- Smith et al., "The ZS-1 central processor," ASPLOS 1987.

- Smith, "Dynamic Instruction Scheduling and the Astronautics ZS-1," IEEE Computer 1989.

Astronautics ZS-1 Instruction Scheduling

- Dynamic scheduling
  - A and X streams are issued/executed independently
  - Loads can bypass stores in the memory unit (if no conflict)
  - Branches executed early in the pipeline
    - To reduce synchronization penalty of A/X streams
    - Works only if the register a branch sources is available

- Static scheduling
  - Move compare instructions as early as possible before a branch
    - So that branch source register is available when branch is decoded
  - Reorder code to expose parallelism in each stream
  - Loop unrolling:
    - Reduces branch count + exposes code reordering opportunities

Loop Unrolling

```
i = 1;
while ( i < 100 ) {
        a[i] = b[i+1] + (i+1)/m
        b[i] = a[i-1] - i/m
        i = i + 1
}
```

```
i = 1;
while ( i < 100 ) {
        a[i] = b[i+1] + (i+1)/m
        b[i] = a[i-1] - i/m

        a[i+1] = b[i+2] + (i+2)/m
        b[i+1] = a[i] - (i+1)/m
        i = i + 2
}
```

- Idea: Replicate loop body multiple times within an iteration

\+ Reduces loop maintenance overhead

  - Induction variable increment or loop condition test

\+ Enlarges basic block (and analysis scope)

  - Enables code optimization and scheduling opportunities

-- What if iteration count not a multiple of unroll factor? (need extra code to detect this)

-- Increases code size

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.

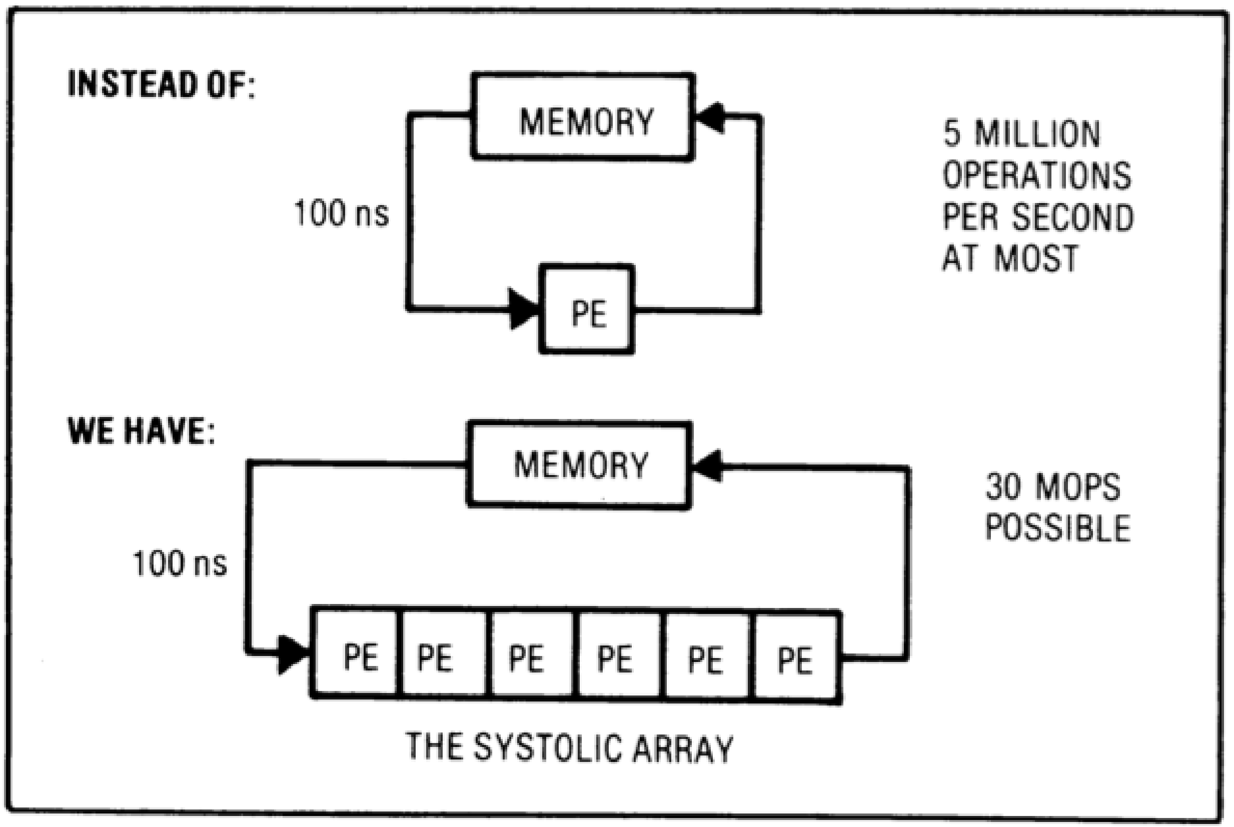


Figure 1. Basic principle of a systolic system.

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

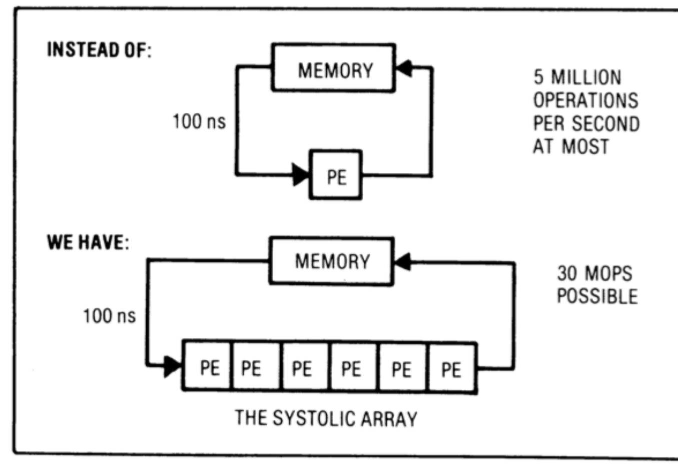


Figure 1. Basic principle of a systolic system.

- 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

$$y_i = w_1 x_i + w_2 x_{i+1} + \ldots + w_k x_{i+k-1}$$

Systolic Computation Example: Convolution

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

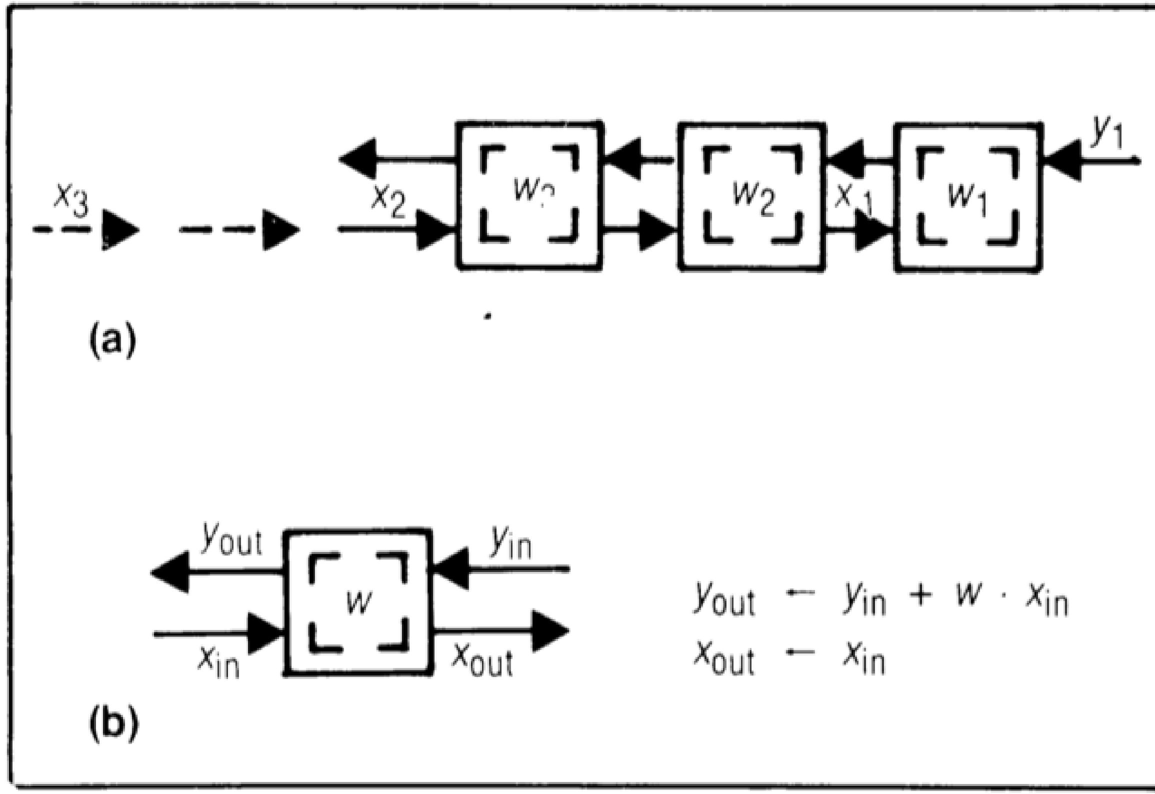


Figure 8. Design W1: systolic convolution array (a) and cell (b) where w_i's stay and x_i's and y_i's move systolically in opposite directions.

Systolic Computation Example: Convolution

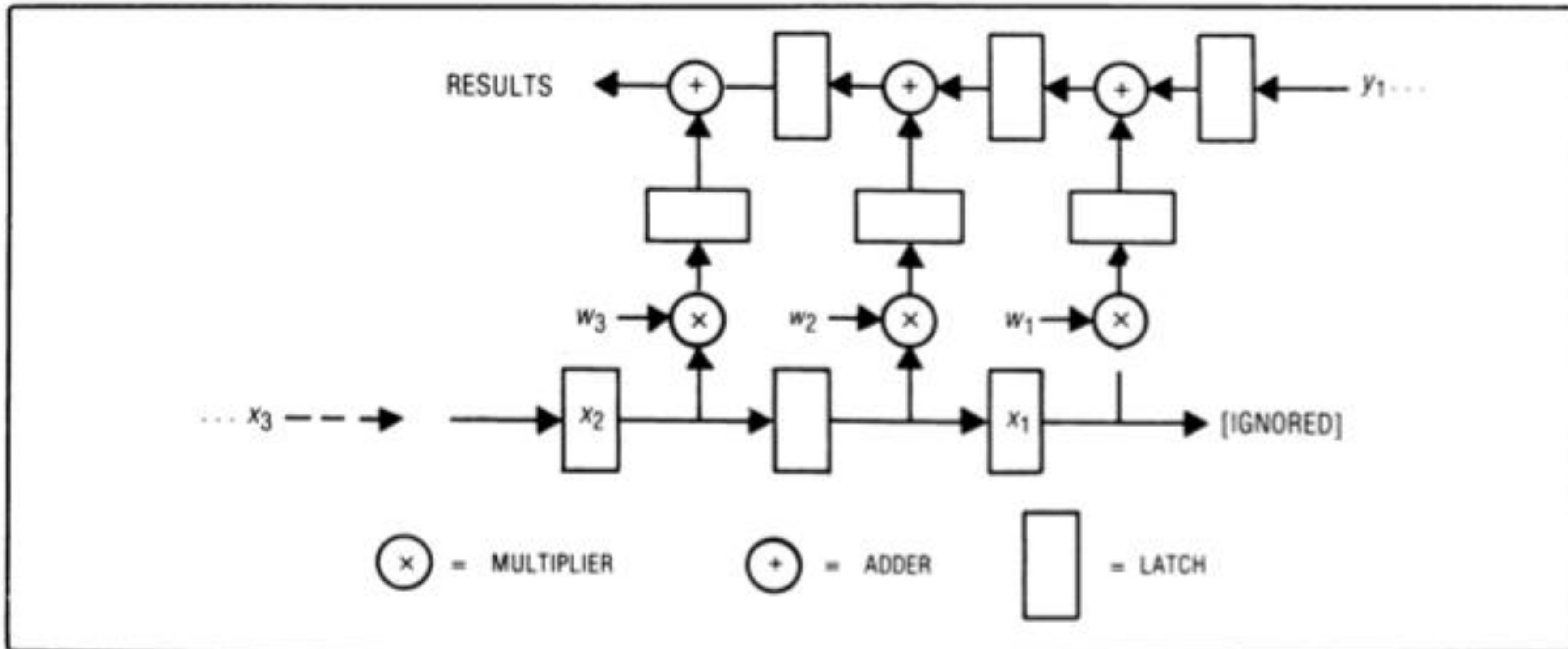


Figure 10. Overlapping the executions of multiply and add in design W1.

- 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

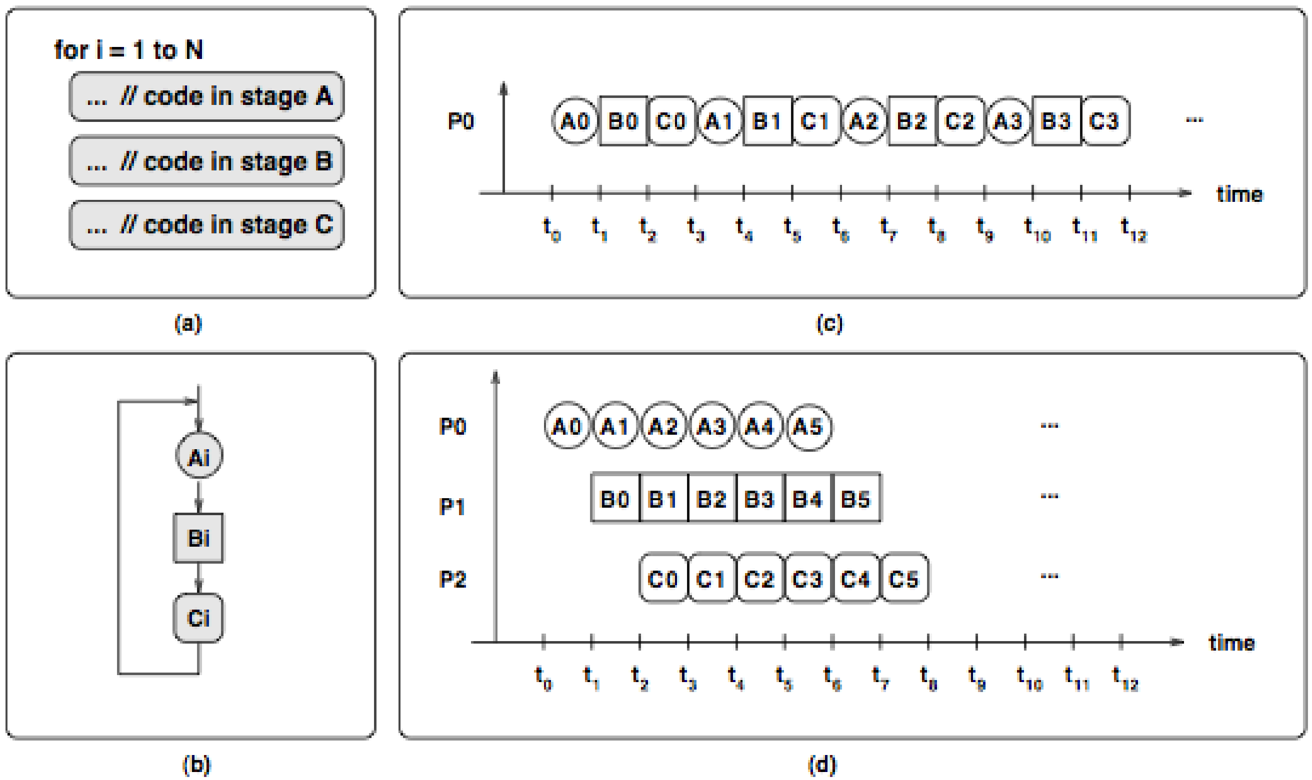


Figure 1. (a) The code of a loop, (b) Each iteration is split into 3 pipeline stages: A, B, and C. Iteration i comprises Ai, Bi, Ci. (c) Sequential execution of 4 iterations. (d) Parallel execution of 6 iterations using pipeline parallelism on a three-core machine. Each stage executes on one core.

File Compression Example

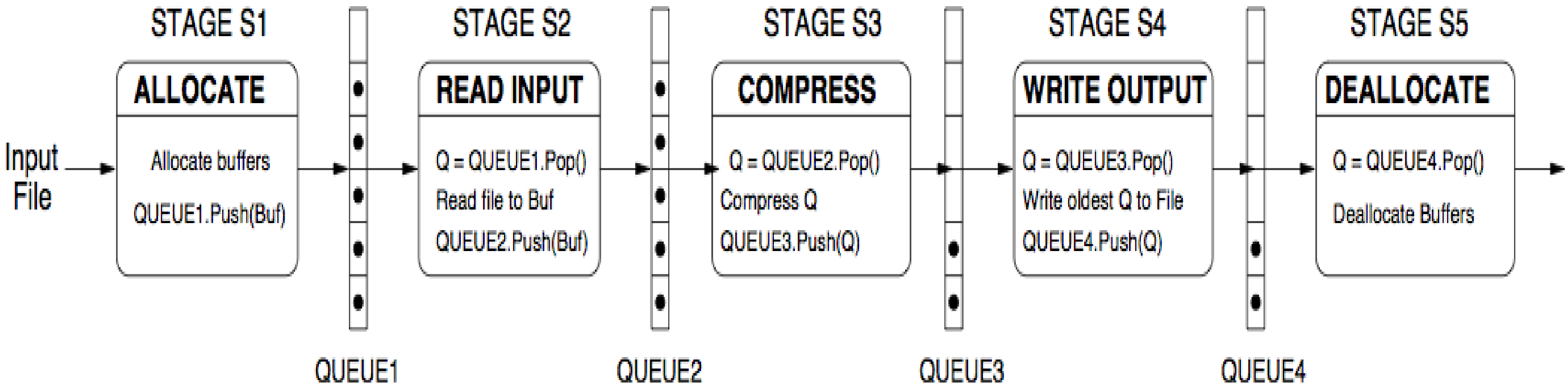


Figure 3. File compression algorithm executed using pipeline parallelism

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

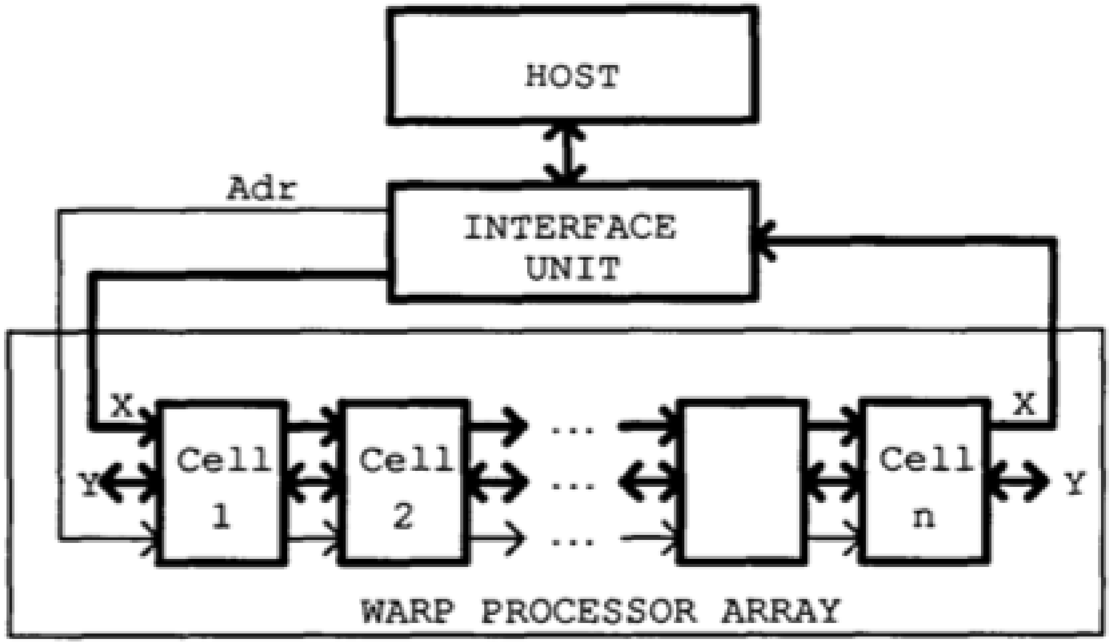


Figure 1: Warp system overview

The WARP Computer

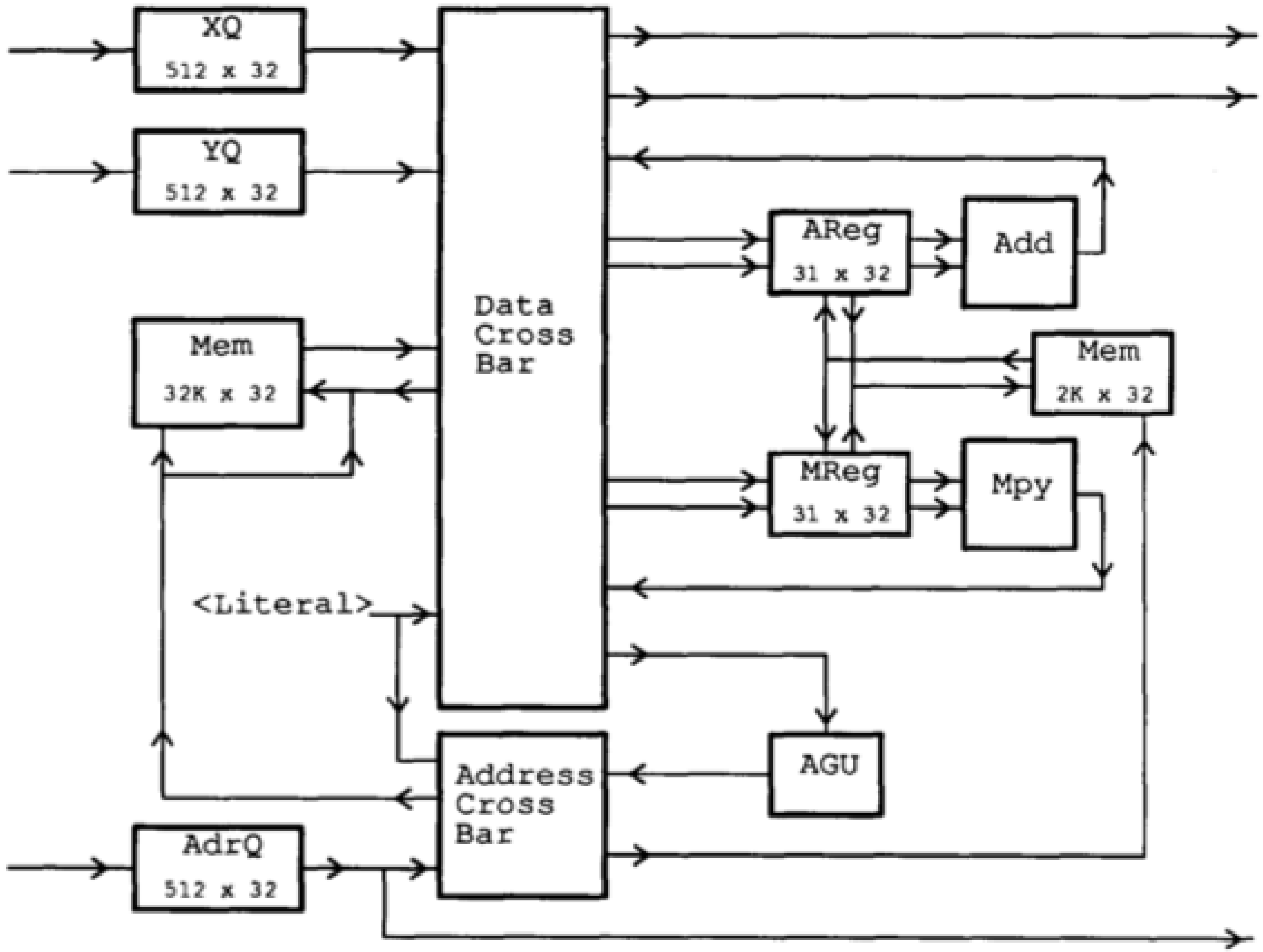


Figure 2: Warp cell data path

Systolic Arrays vs. SIMD

- Food for thought…

Some More Recommended Readings

- Recommended:
  - Fisher, "Very Long Instruction Word architectures and the ELI-512," ISCA 1983.
  - Huck et al., "Introducing the IA-64 Architecture," IEEE Micro 2000.
  - Russell, "The CRAY-1 computer system," CACM 1978.
  - Rau and Fisher, "Instruction-level parallel processing: history, overview, and perspective," Journal of Supercomputing, 1993.
  - Faraboschi et al., "Instruction Scheduling for Instruction Level Parallel Processors," Proc. IEEE, Nov. 2001.