

Lecture 19: SIMD and GPU

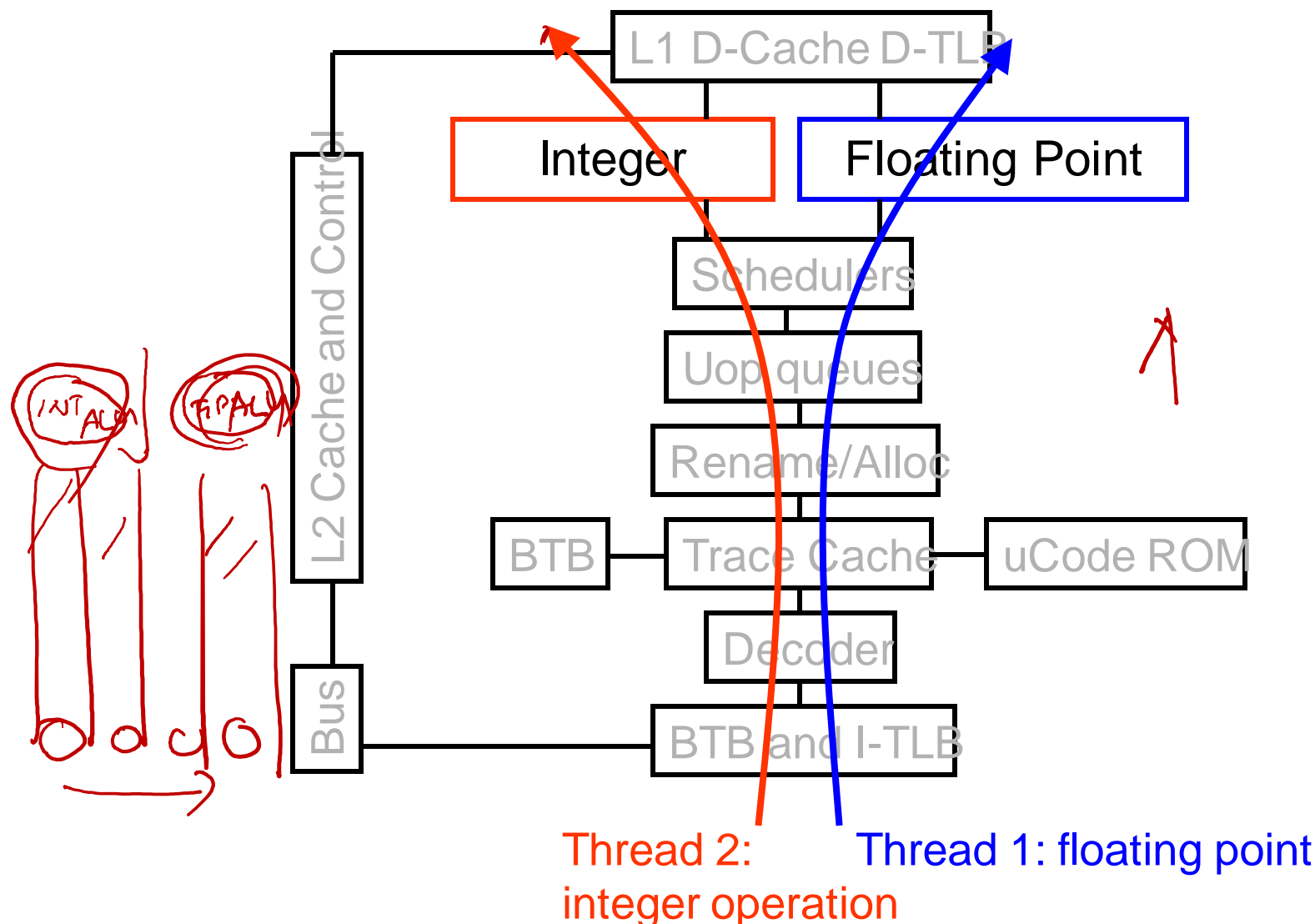
Acknowledgement: Many slides were adapted from Prof. Mutulu's ECE740 Computer Architecture at ETH with his generous permission. Also some slides were from Prof. Hall's CS4961 at U of Utha.

Review: Why Multi-core?

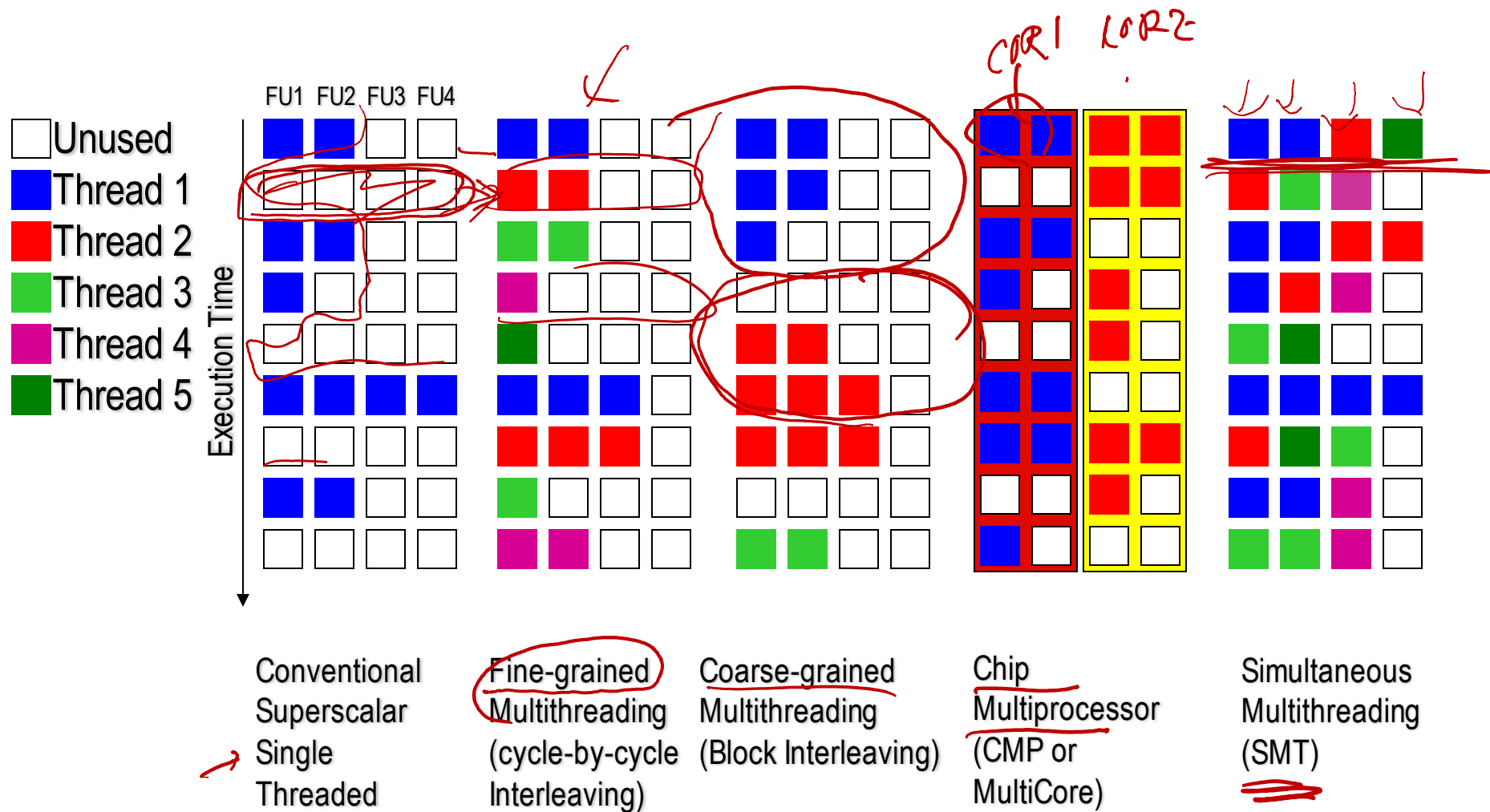
- Never ending story ...
 - ◆ Complex Applications
 - ◆ Faster Computation
 - ◆ How far did we go with uniprocessors?
- **Parallel Processors** now play a major role
 - ◆ Logical way to improve performance
 - Connect multiple microprocessors
 - ◆ Not much left with ILP exploitation
 - ◆ Server and embedded software have parallelism
- Multiprocessor architectures will become increasingly attractive
 - ◆ Due to slowdown in advances of uniprocessors

Review: Simultaneous Multi-Threading (SMT)

- SMT processor: both threads can run concurrently



Review: Multi-Threading Paradigm



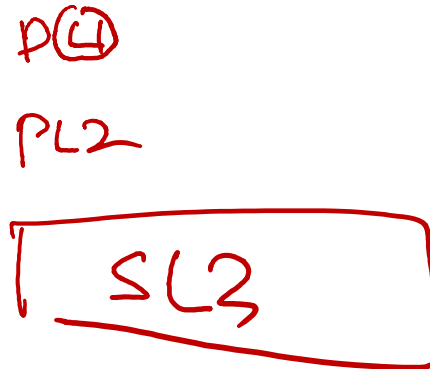
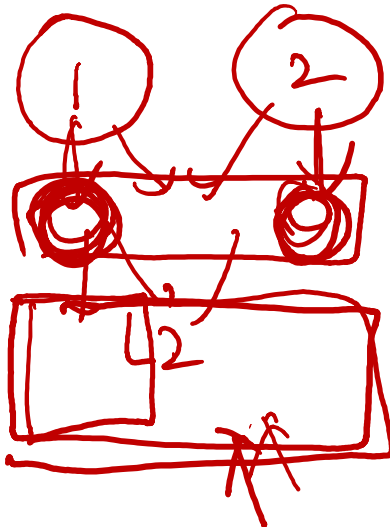
Review: Combining Multi-core and SMT

- cores can be SMT-enabled (or not)
- different combinations:
 - ✓ single-core, non-SMT: standard uniprocessor
 - ✓ single-core, w/ SMT
 - ✓ multi-core, non-SMT
 - ✓ multi-core, w/ SMT: our fish machines
- number of SMT threads
 - ✓ 2, 4, or sometimes 8 simultaneous threads
- Intel calls them “hyper-threads”



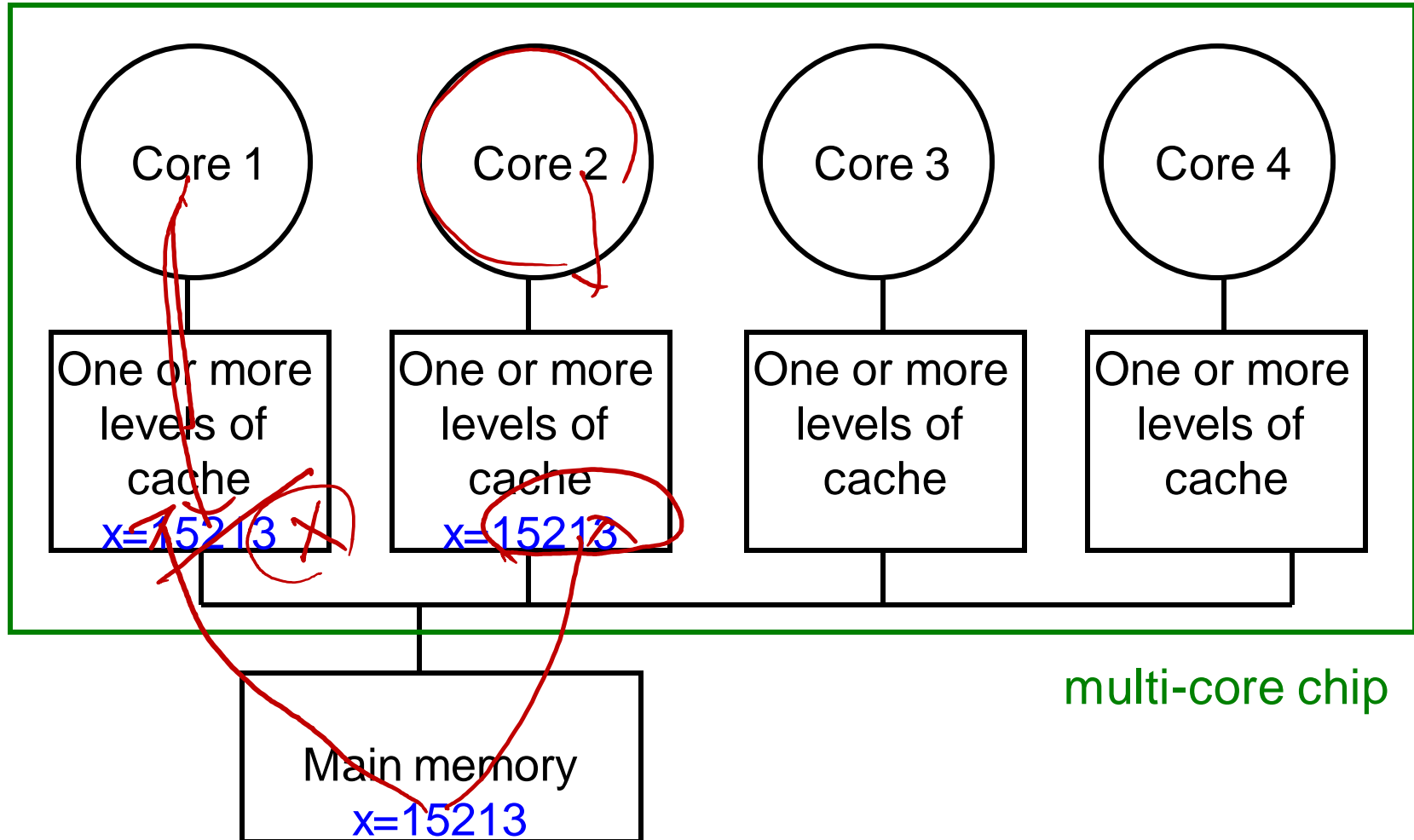
Review: Private vs shared caches

- advantages of private:
 - ✓ they are closer to core, so faster access
 - ✓ reduces contention
- advantages of shared:
 - ✓ threads on different cores can share the same cache data
 - ✓ more cache space available if a single (or a few) high-performance thread runs on the system



Review: Cache Coherence Problem

- core 2 reads x

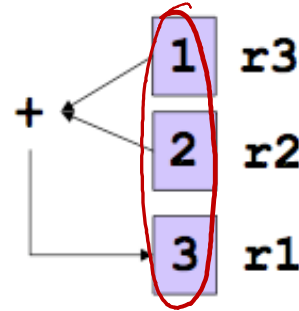


Flynn's Taxonomy of Computers

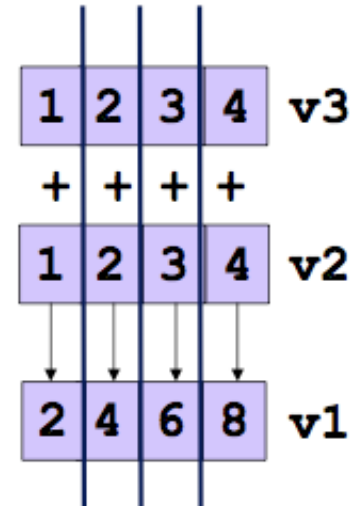
- Mike Flynn, “**Very High-Speed Computing Systems**,” Proc. of IEEE, 1966
- **SISD**: single instruction operates on single data element
- **SIMD**: single instruction operates on multiple data elements
 - vector processor, GPU
- ~~MISD: multiple instructions operate on single data element~~
- **MIMD**: multiple instructions operate on multiple data elements (multiple instruction streams)
 - multiprocessor
 - multithreaded processor

Scalar vs. SIMD in Multimedia Extensions

Scalar: `add r1, r2, r3`

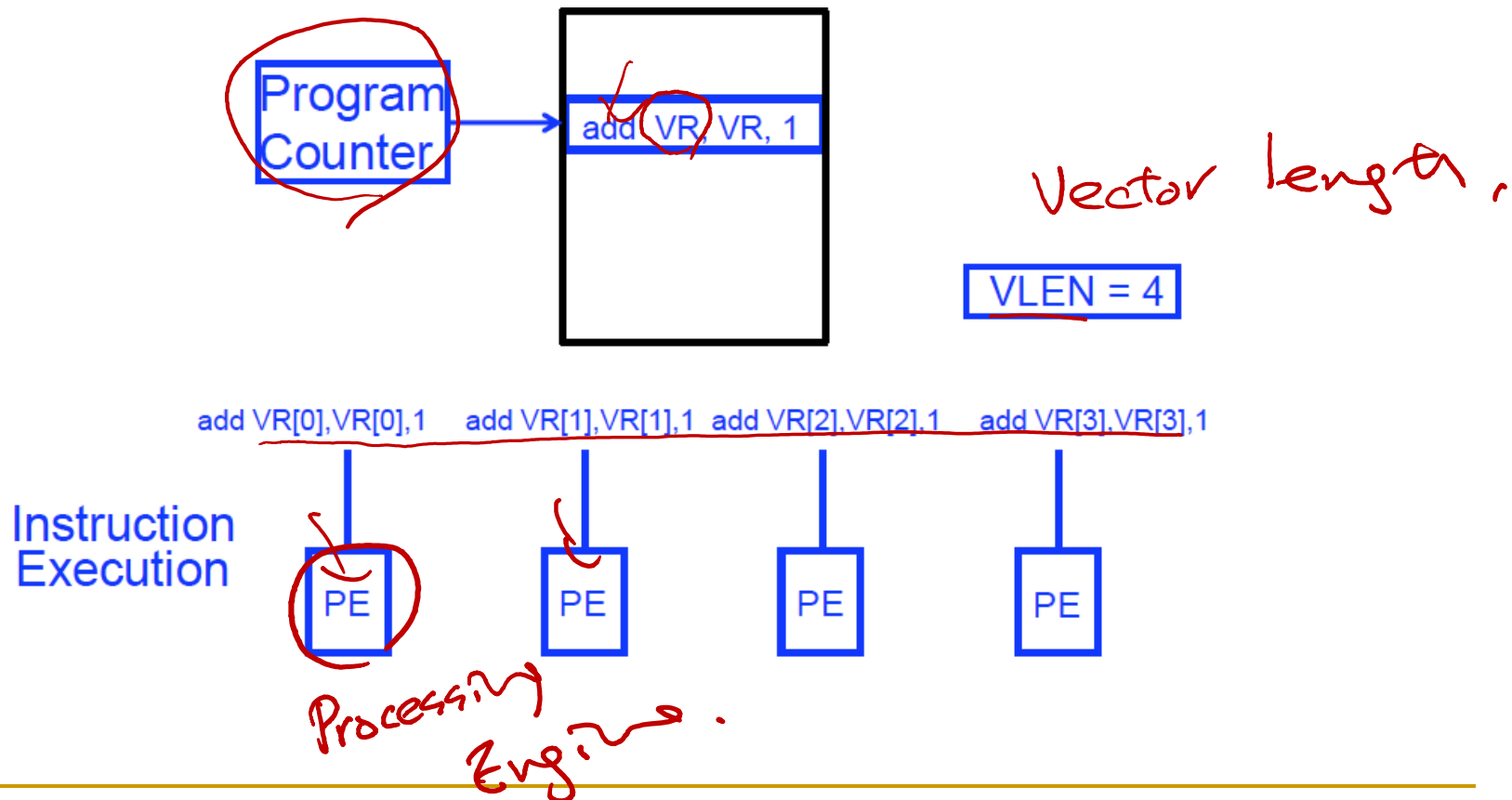


SIMD: `vadd<sws> v1, v2, v3`



SIMD Processing

- single instruction controls simultaneous execution of many processing elements
 - lockstep basis



SIMD Application

image

- ❑ graphics : 3D games, movies
- ❑ image recognition
- ❑ video encoding/decoding : JPEG, MPEG4

sound

- ❑ encoding/decoding: IP phone, MP3
- ❑ speech recognition
- ❑ digital signal processing: cell phones

scientific applications

- ❑ array-based data-parallel computation, another level of parallelism
-

Characteristics of Multimedia Applications

- regular data access pattern
 - data items are contiguous in memory
- short data types
 - 8, 16, 32 bits
- data streaming through a series of processing stages
 - □ some temporal reuse for such data streams
- sometimes ...
 - many constants
 - short iteration counts
 - requires saturation arithmetic

Why SIMD

SIMD

+ more parallelism when parallelism is abundant

+ SIMD in addition to ILP

+ simple design

+ replicated functional units

+ energy efficient

+ only needs one instruction for many data operations

+ attractive for personal mobile devices

+ small die area

+ no heavily ported register files

+ die area: +MAX-2(HP): 0.1% +VIS(Sun): 3.0%

- must be explicitly exposed to the hardware

- by the compiler or by the programmer

1
2
3
4
1

Exploiting TLP with SIMD Execution

■ benefit:

- multiple ALU ops → one SIMD op
- multiple ld/st ops → one wide mem op

■ what are the overheads:

- packing and unpacking:
 - rearrange data so that it is contiguous
- alignment overhead
 - accessing data from the memory system so that it is aligned to a "superword" boundary
- control flow
 - control flow may require executing all paths

Characteristics of SIMD

- independent ALU ops

$$\left\{ \begin{array}{l} R = R + XR * 1.08327 \\ G = G + XG * 1.89234 \\ B = B + XB * 1.29835 \end{array} \right.$$



$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} R \\ G \\ B \end{bmatrix} + \begin{bmatrix} XR \\ XG \\ XB \end{bmatrix} * \begin{bmatrix} 1.08327 \\ 1.89234 \\ 1.29835 \end{bmatrix}$$

Characteristics of SIMD

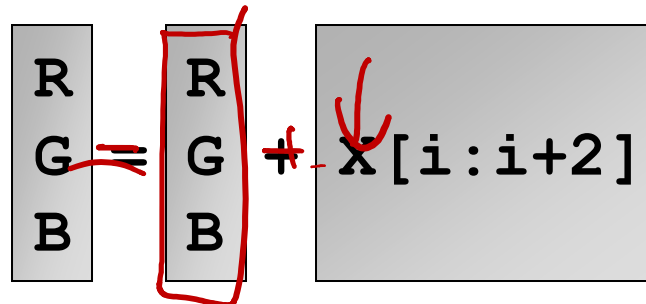
- adjacent memory references



24)

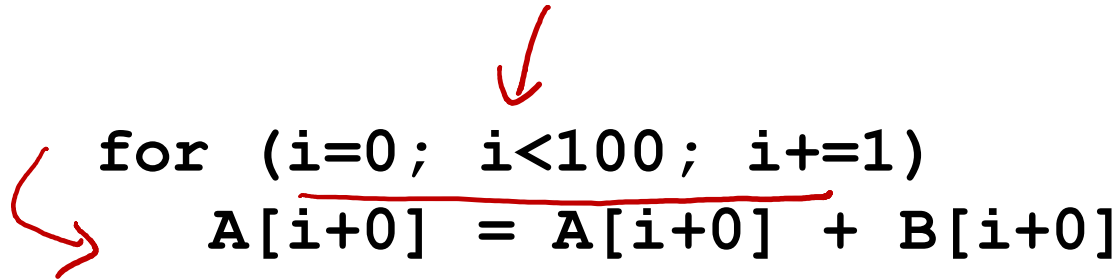
R = R + X[i+0]
G = G + X[i+1]
B = B + X[i+2]

3 LD + 3 ADD



Characteristics of SIMD

- vectorizable loops



A code snippet illustrating a vectorizable loop. The loop is written in C-style syntax: `for (i=0; i<100; i+=1)`. A red arrow points down to the loop condition `i<100`. Below the loop, the assignment statement `A[i+0] = A[i+0] + B[i+0]` is shown. A red arrow points from the left to the first `A[i+0]`, and a red underline is drawn under the entire assignment statement `A[i+0] = A[i+0] + B[i+0]`.

```
for (i=0; i<100; i+=1)
    A[i+0] = A[i+0] + B[i+0]
```

Characteristics of SIMD

- vectorizable loops

~~for (i=0; i<100; i+=4)~~

A[i+0] = A[i+0] + B[i+0]

A[i+1] = A[i+1] + B[i+1]

A[i+2] = A[i+2] + B[i+2]

A[i+3] = A[i+3] + B[i+3]



for (i=0; i<100; i+=4)

A[i:i+3] = B[i:i+3] + C[i:i+3]

Characteristics of SIMD

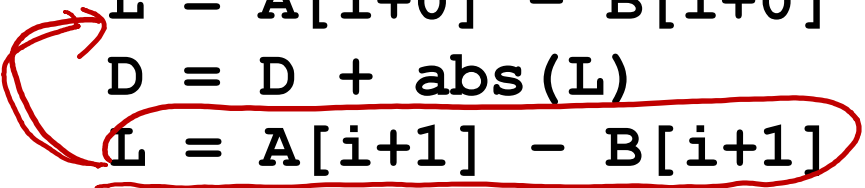
- partially vectorizable loops

```
for (i=0; i<16; i+=1)  
    → L = A[i+0] - B[i+0]  
    → D = D + abs(L)
```

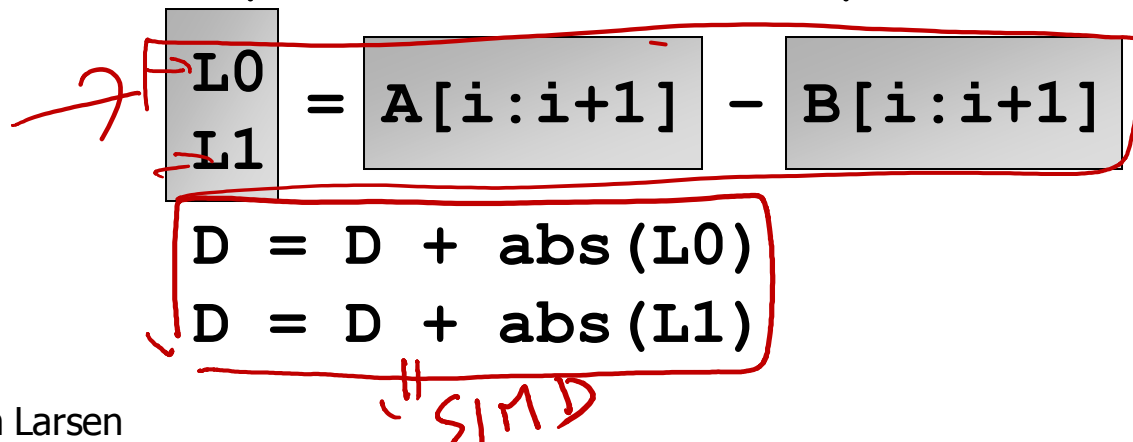
Characteristics of SIMD

- partially vectorizable loops

```
for (i=0; i<16; i+=2)
  L = A[i+0] - B[i+0]
  D = D + abs(L)
  L = A[i+1] - B[i+1]
  D = D + abs(L)
```



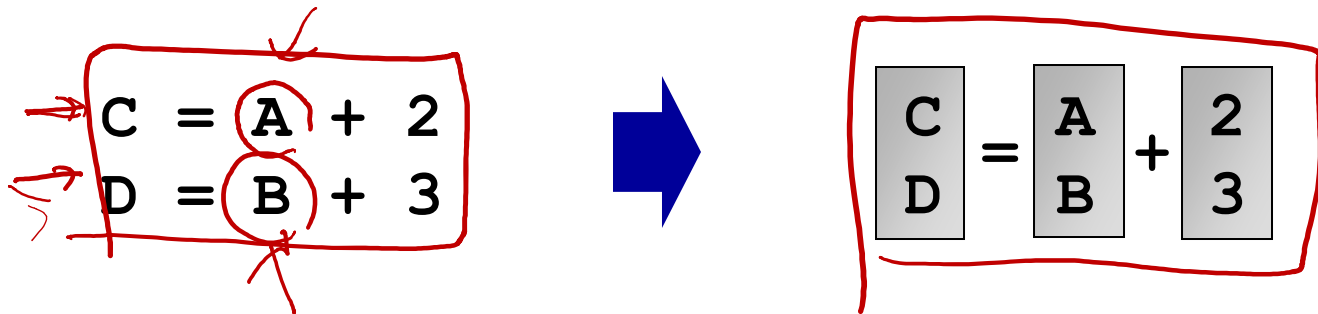
```
for (i=0; i<16; i+=2)
  L0 = A[i:i+1] - B[i:i+1]
  L1 = A[i:i+1] - B[i:i+1]
  D = D + abs(L0)
  D = D + abs(L1)
```



Programming Complexity Issues

- high level: use compiler
 - may not always be successful
 - low level: use intrinsics or inline assembly tedious and error prone
 - data must be aligned and adjacent in memory
 - unaligned data may produce incorrect results
 - may need to copy to get adjacency (overhead)
 - control flow introduces complexity and inefficiency
-

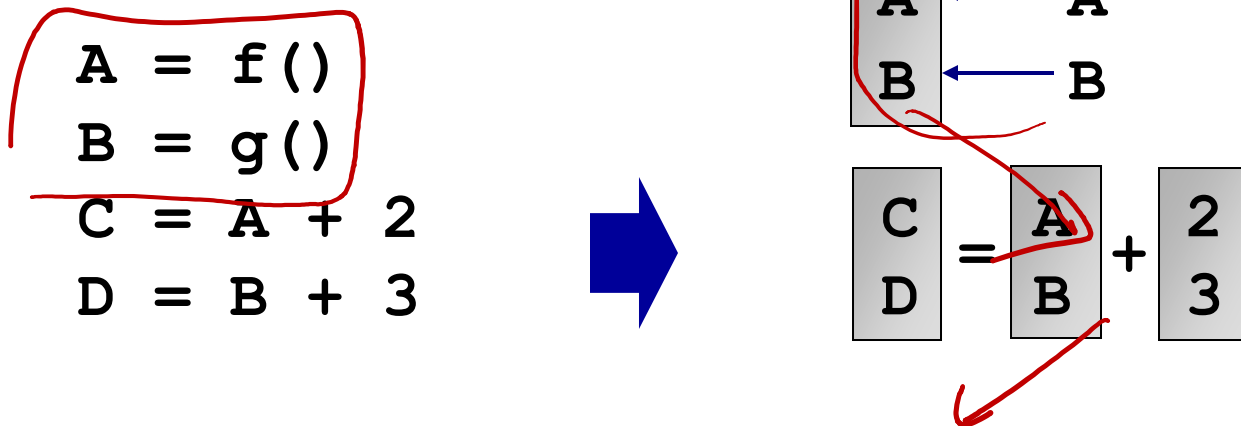
Packing/Unpacking Costs



slide source: Sam Larsen

Packing/Unpacking Costs

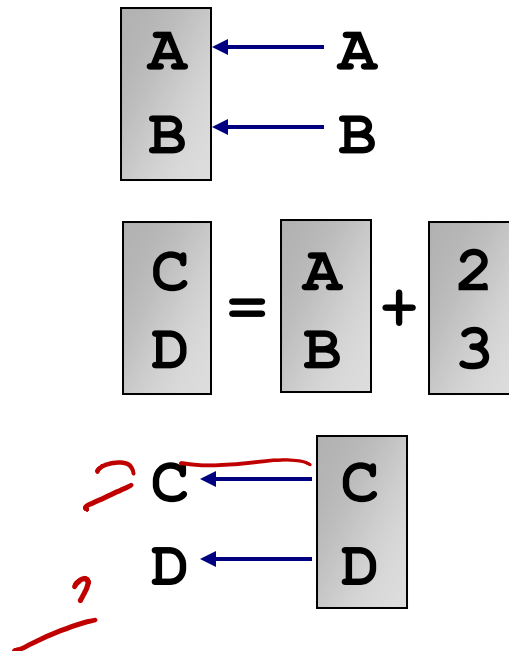
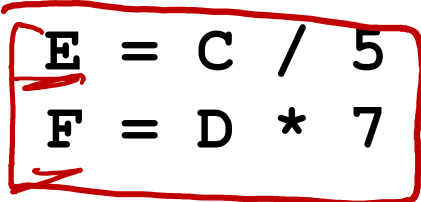
- packing source operands
 - copying into contiguous memory



Packing/Unpacking Costs

- packing source operands
 - copying into contiguous memory
- unpacking destination operands
 - copying back to location

```
A = f()  
B = g()  
C = A + 2  
D = B + 3  
E = C / 5  
F = D * 7
```



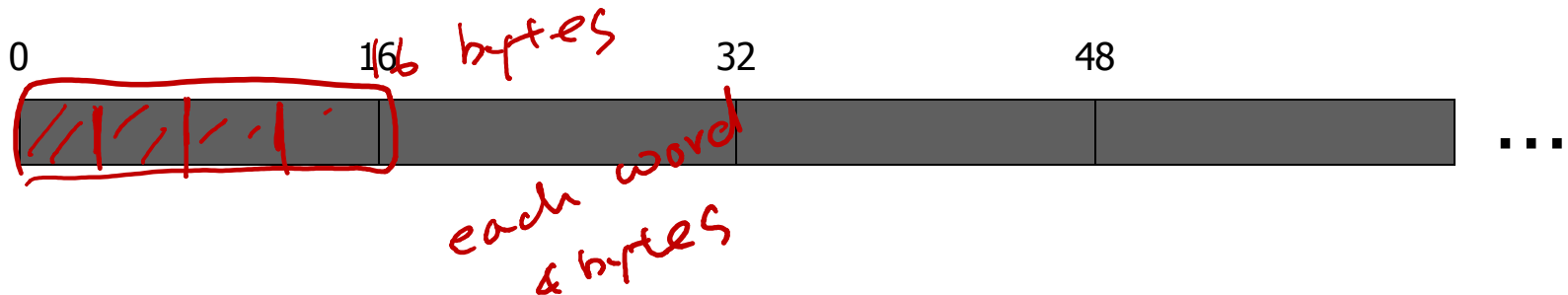
slide source: Sam Larsen

Alignment Code Generation

- aligned memory access
 - the address is always a multiple of 16 bytes
 - just one superword load or store instruction

```
float a[64];  
for (i=0; i<64; i+=4)  
    Va = a[i:i+3];
```

4 FP values.



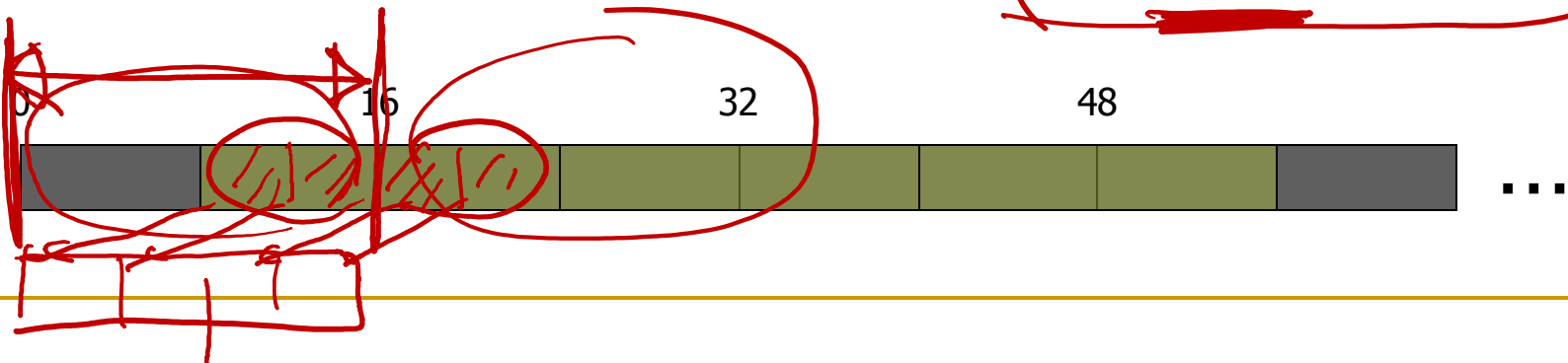
Alignment Code Generation (cont.)

- misaligned memory access
 - the address is always a non-zero constant offset away from the 16 byte boundaries.
 - static alignment: for a misaligned load, issue two adjacent aligned loads followed by a merge.

```
float a[64];  
for (i=0; i<60; i+=4)  
    Va = a[i+2:i+5];
```



```
float a[64];  
for (i=0; i<60; i+=4)  
    V1 = a[i:i+3];  
    V2 = a[i+4:i+7];  
    Va = merge(V1, V2, 8);
```



SIMD Variations

- SIMD processing units in CPU
 - MMX: Multimedia Extensions (1996)
 - repurpose 64-bit floating point registers
 - eight 8-bit integer ops or four 16-bit integer ops
 - SSE: Streaming SIMD Extensions (1999, 2001, 2004, 2007)
 - separate 128-bit registers
 - eight 16-bit ops, four 32-bit ops, or two 64-bit ops
 - single-precision floating point arithmetic
 - AVX: Advanced Vector Extension (2010)
 - doubles the width to 256 bits, i.e., four 64-bit integer/fp ops,
 - extendible to 512 and 1024 bits for future generations
 - operands must be consecutive and aligned memory locations

Intel Pentium MMX Operations

- media applications operate on data types narrower than the native word size
 - graphics systems use 8 bits per primary color
 - audio samples use 8-16 bits

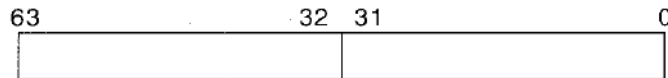
*You can
have a look
at this
later.*



(a)



(b)



(c)



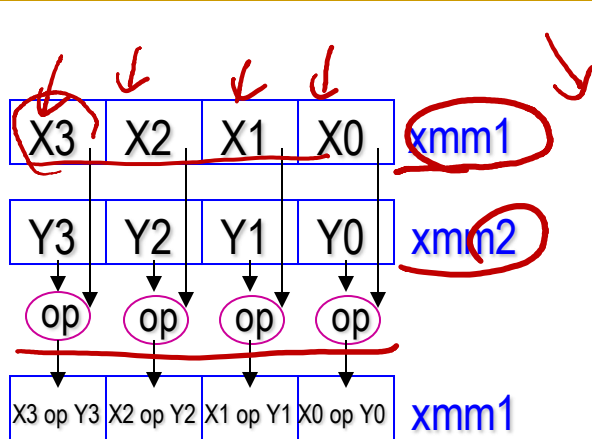
(d)

8 8-bit bytes
4 16-bit words
2 32-bit doublewords
1 64-bit quadword

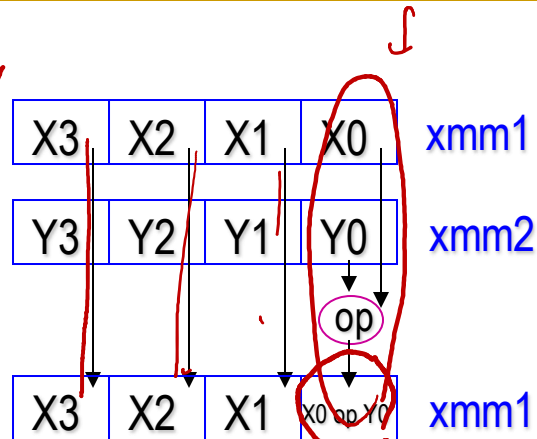
Peleg and Weiser, “**MMX Technology Extension to the Intel Architecture**,”
IEEE Micro, 1996.

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

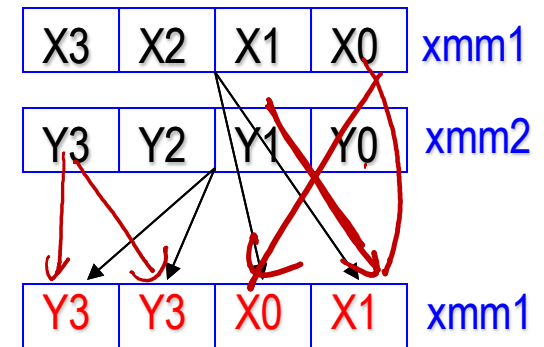
Examples of Using SSE



Packed SP FP operation
(e.g. `ADDPS xmm1, xmm2`)



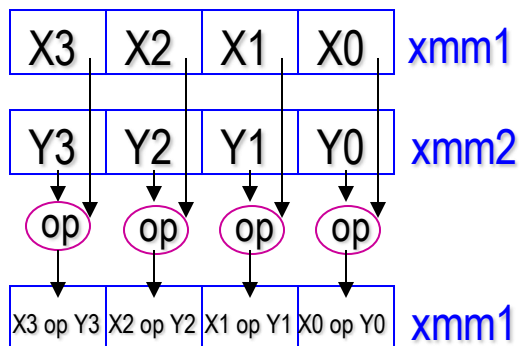
Scalar SP FP operation
(e.g. `ADDSS xmm1, xmm2`)



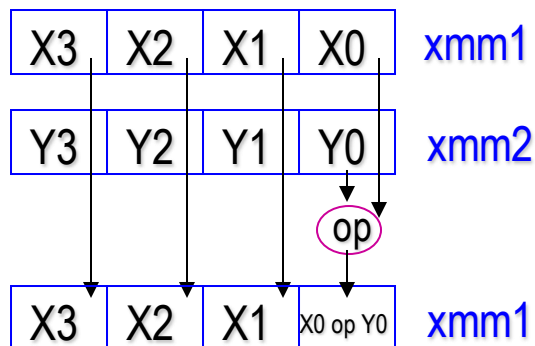
Shuffle FP operation (8-bit imm)
(e.g. `SHUFPS xmm1, xmm2, 0x11`)

Examples of Using SSE and SSE2

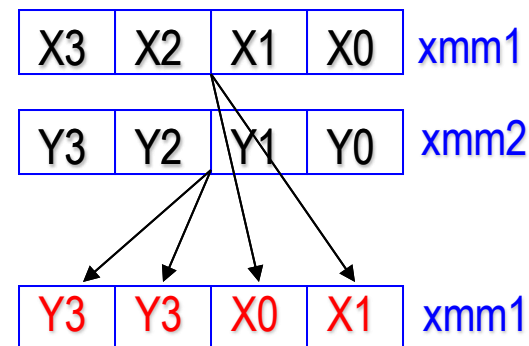
<http://www.songho.ca/misc/sse/sse.html>



Packed **SP** FP operation
(e.g. **ADDPS** xmm1, xmm2)

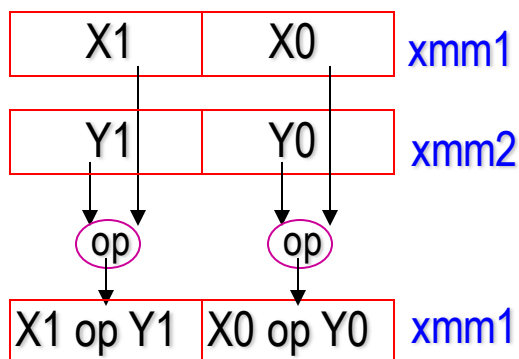


Scalar **SP** FP operation
(e.g. **ADDSS** xmm1, xmm2)

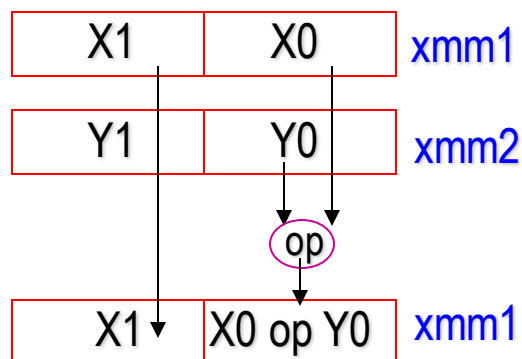


Shuffle **FP** operation (8-bit imm)
(e.g. **SHUFPS** xmm1, xmm2, **0xf1**)

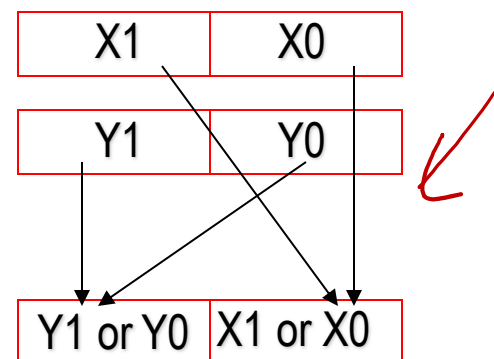
SSE2



Packed **DP** FP operation
(e.g. **ADDPD** xmm1, xmm2)

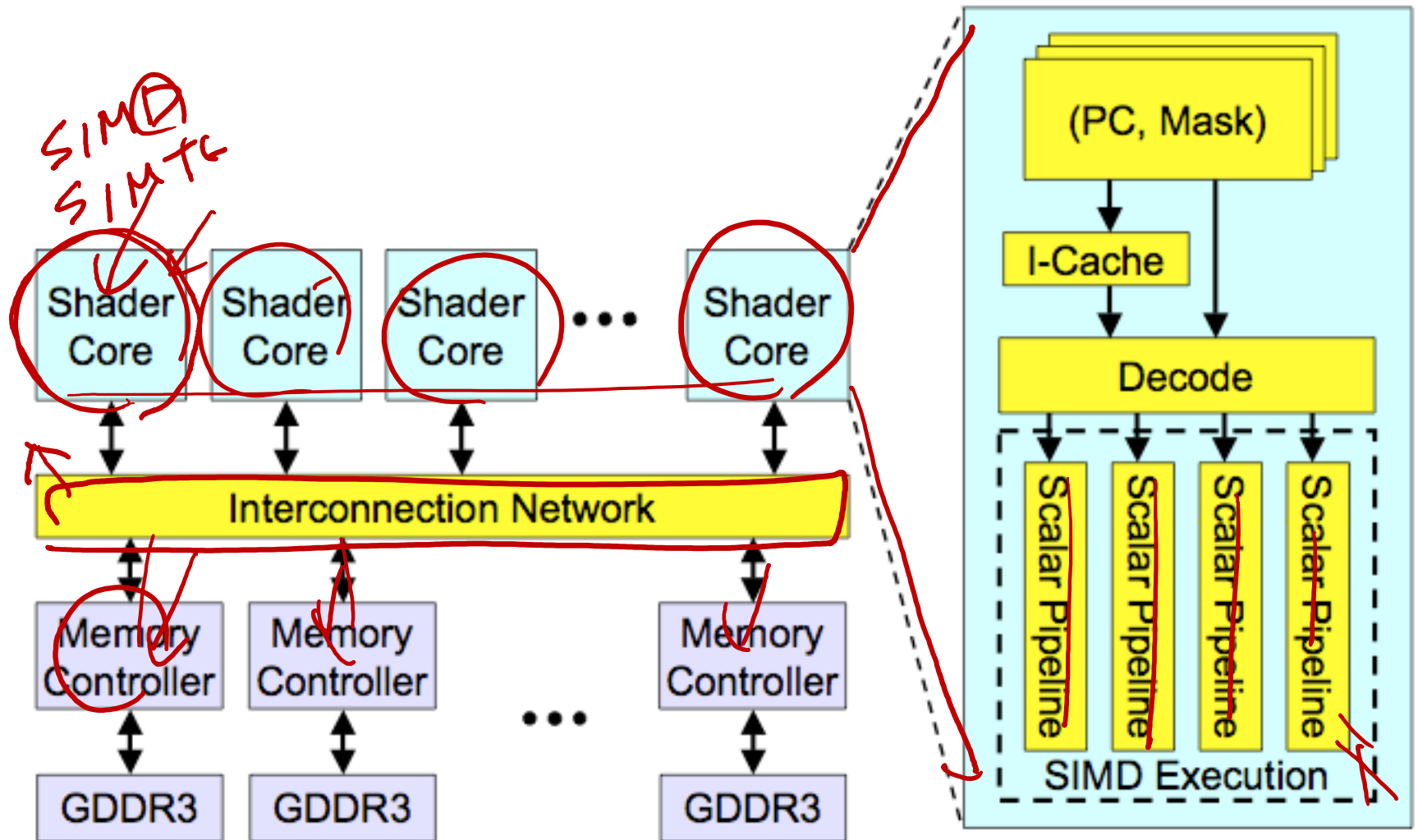


Scalar **DP** FP operation
(e.g. **ADDSD** xmm1, xmm2)



Shuffle **DP** operation (2-bit imm)
(e.g. **SHUFPD** xmm1, xmm2, **imm2**)

High-Level View of a GPU



Sample GPU SIMT Code (Simplified)

CPU code

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

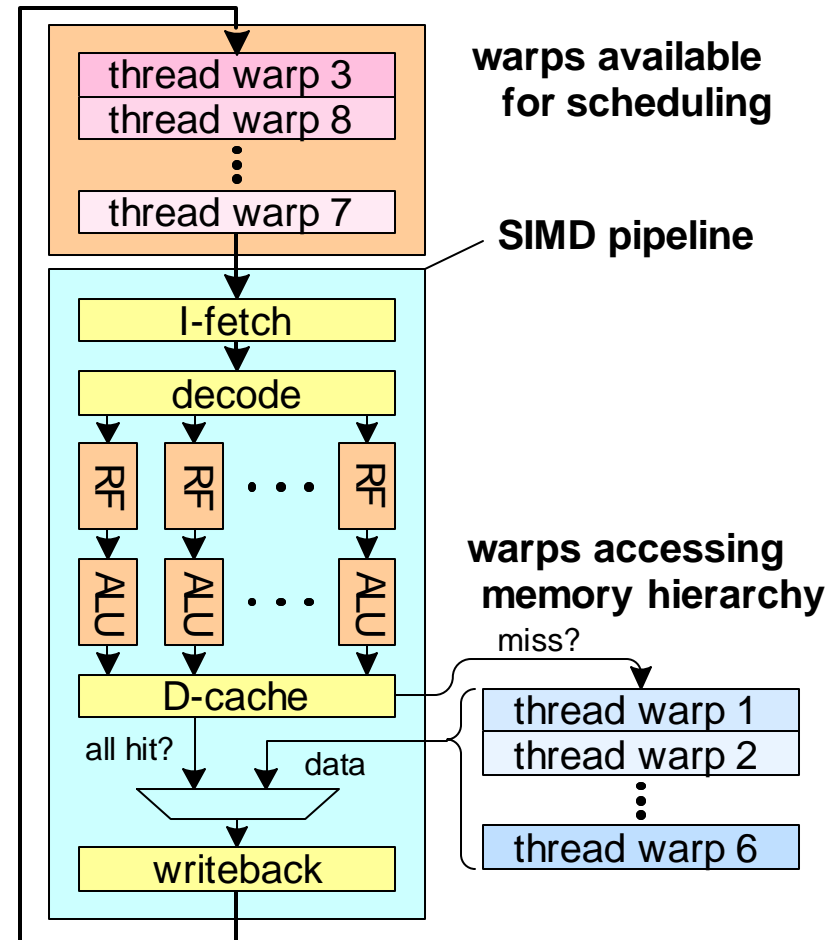


CUDA code

```
// 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;  
}
```

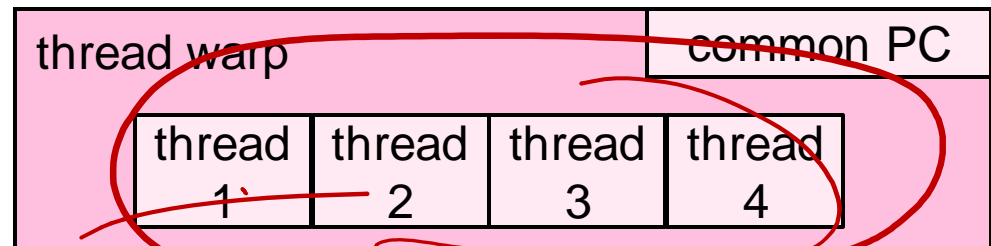
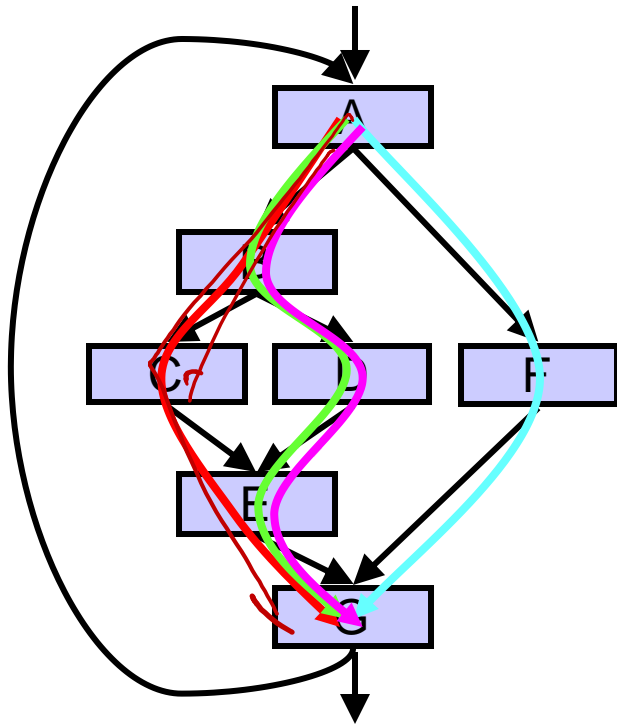

Latency Hiding with “Thread Warps”

- warp: a set of threads that execute the same instruction (on different data elements)
SIMD
- 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
- no OS context switching
- memory latency hiding
 - graphics has millions of pixels



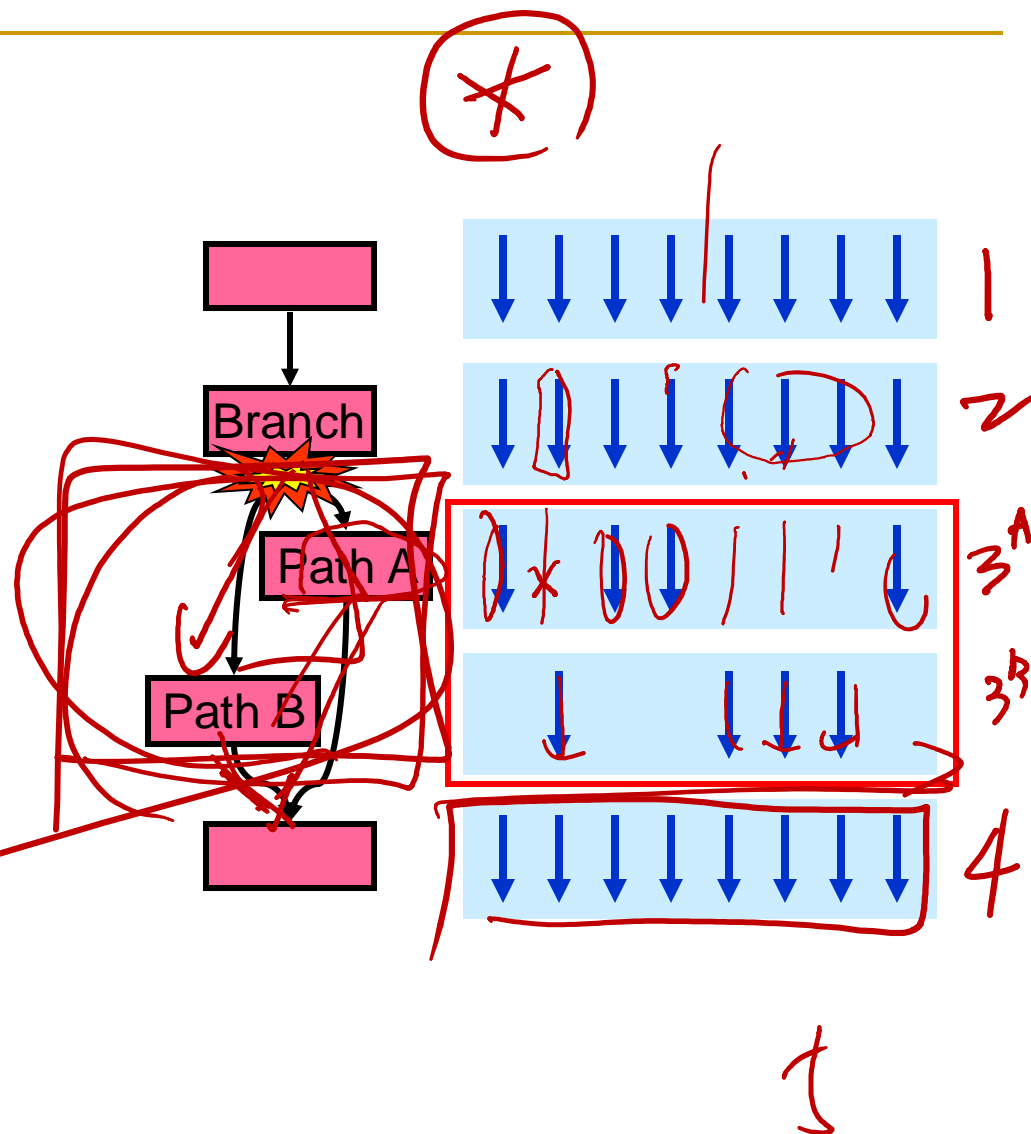
Branch Divergence Problem in Warp-based SIMD

- 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.

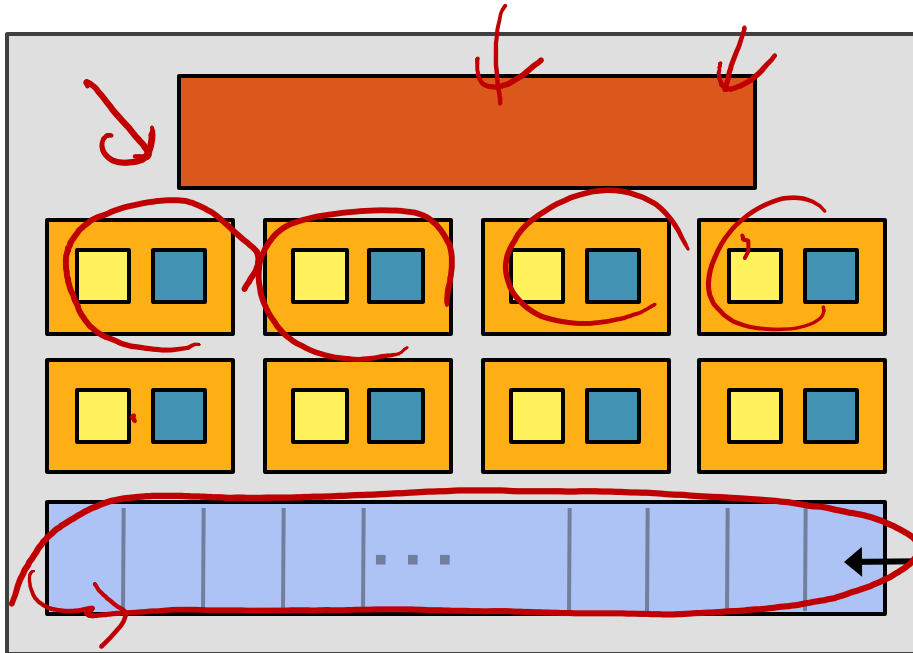


NVIDIA GeForce GTX 285

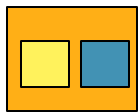
- NVIDIA-speak:
 - ❑ 240 stream processors
 - ❑ “SIMT execution”
- generic speak:
 - ❑ 30 cores
 - ❑ 8 SIMD functional units per core



NVIDIA GeForce GTX 285 “core”



64 KB of storage
for fragment
contexts (registers)



= SIMD functional unit, control
shared across 8 units

■ = multiply-add
■ = multiply

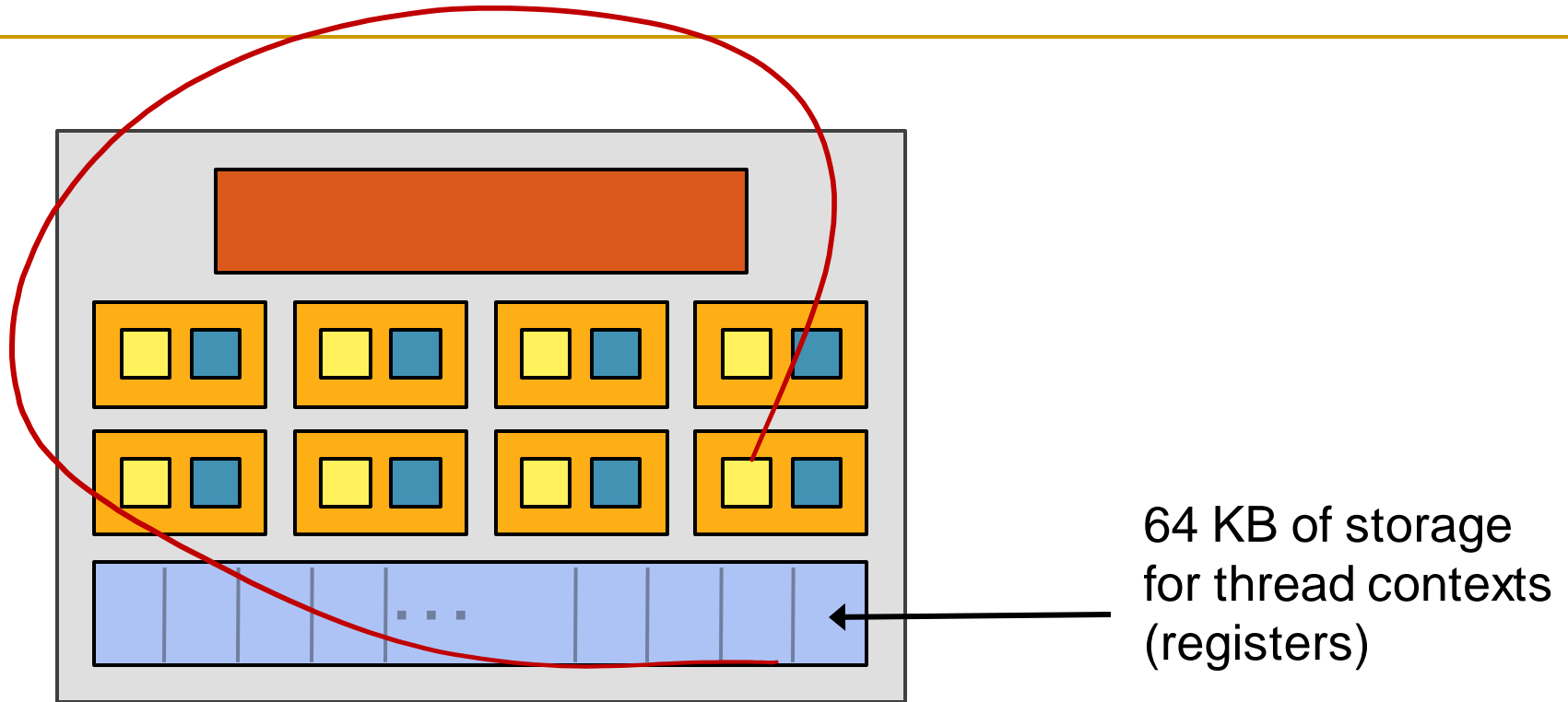


= instruction stream decode



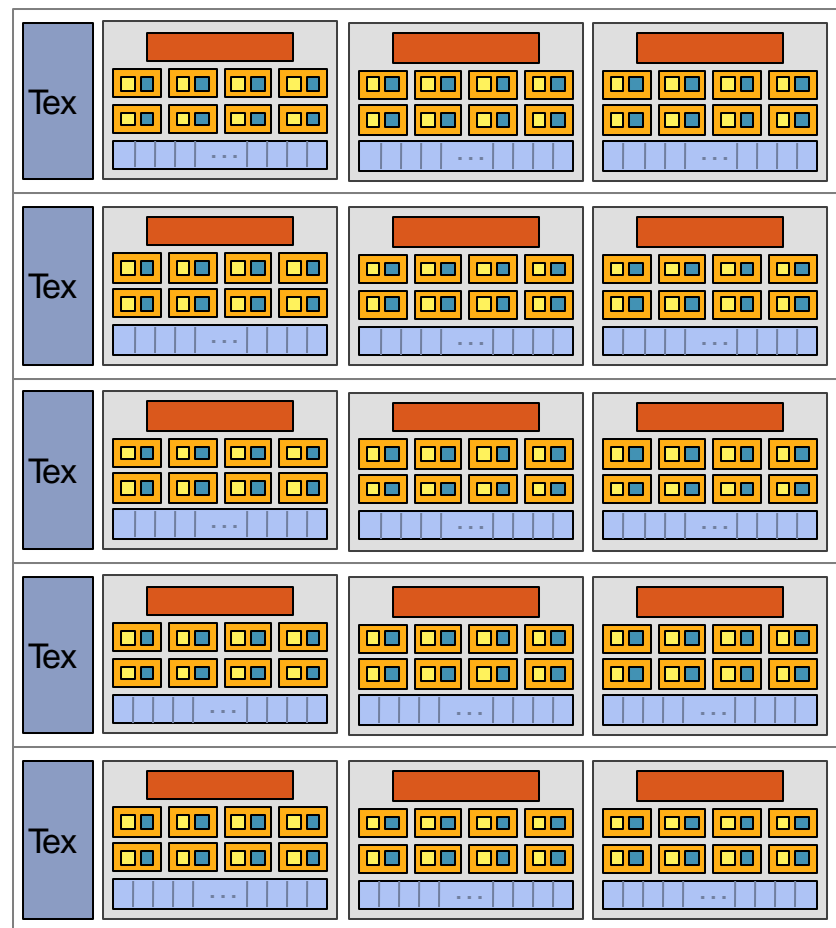
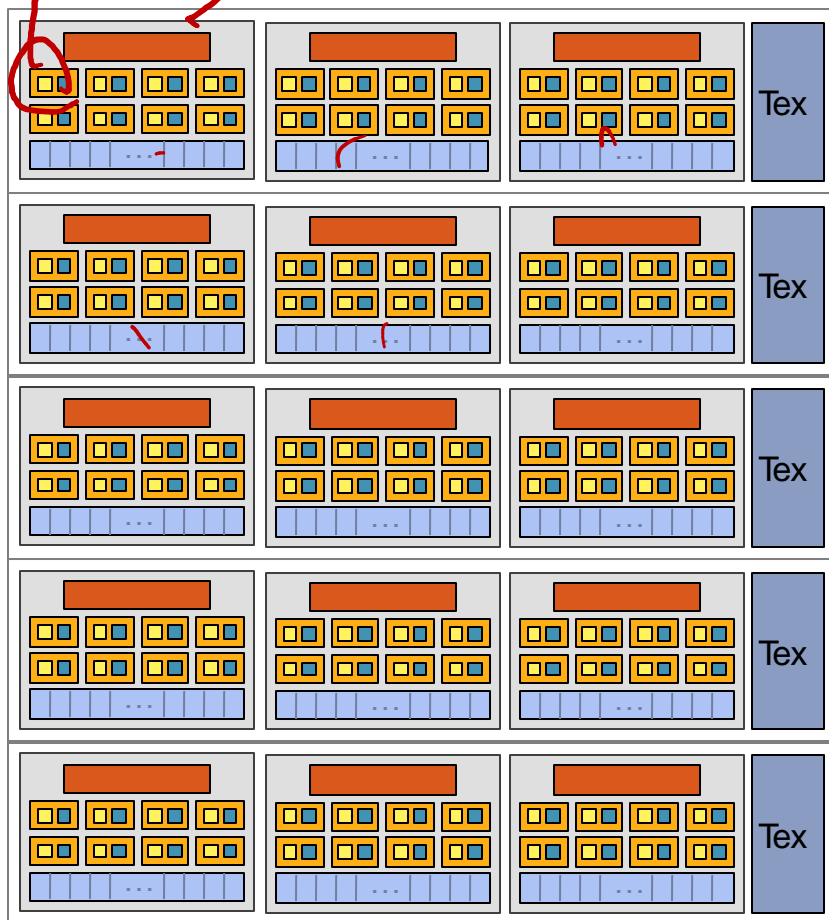
= execution context storage

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

NVIDIA GeForce GTX 285



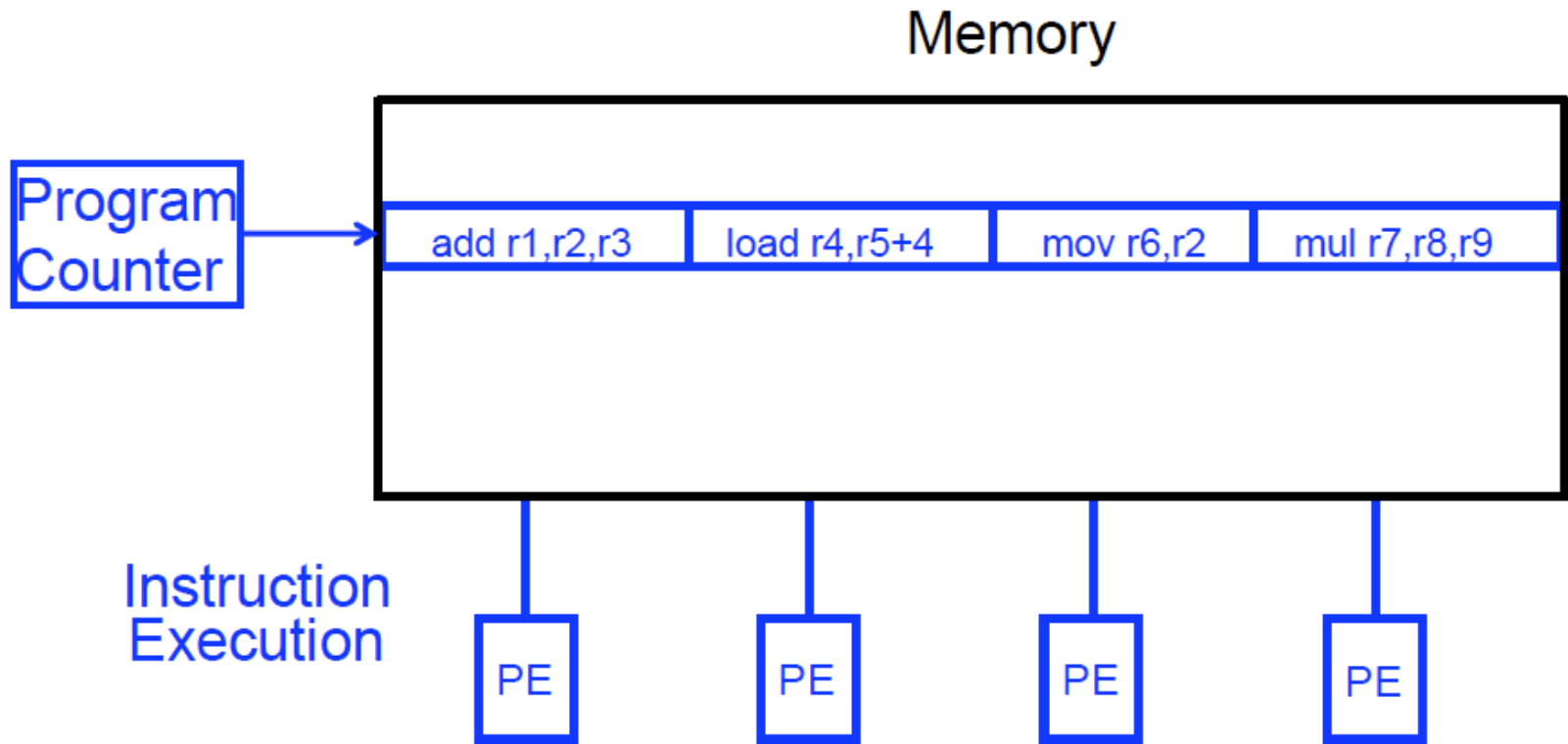
there are 30 of these things on the GTX 285: 30,720 threads

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)

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 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

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

Announcement

- next lecture: impact of technology
- MP assignment
 - ✓ MP3 final checkpoint due on 4/22 5pm ✕
 - ✓ 4/24 and 4/26 project review presentation

