National Yang Ming Chiao Tung University
Computer Architecture & System Lab

Lecture 13: Multicores

CS10014 Computer Organization

Department of Computer Science
Tsung Tai Yeh
Thursday: 1:20 pm– 3:10 pm
Classroom: EC-022

Acknowledgements and Disclaimer

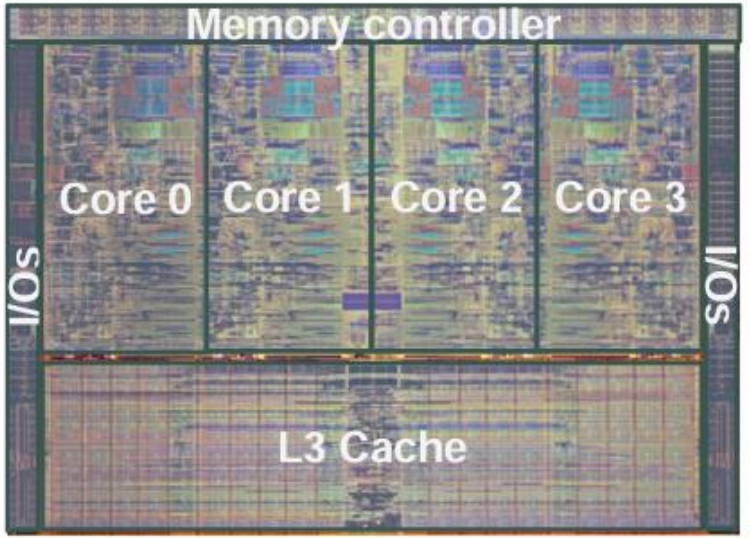
- Slides were developed in the reference with
  - CS 61C at UC Berkeley
    - https://inst.eecs.berkeley.edu/~cs61c/sp23/
  - 6.888 at MIT
    - https://courses.csail.mit.edu/6.888/spring13/
  - CIS510 at Upenn
    - https://www.cis.upenn.edu/~cis5710/spring2019/

Outline

- Multi-core Processor
- Pipelining Optimization
- Superscalar Processor
- Hardware Multi-Threading
- Vector Processors
- Graphics Processors

Multi-core Processor (1/4)

- A single core can only be so fast
  - Limited clock frequency
  - Limited instruction-level parallelism
- What if we need even more computing power?
  - Use multiple cores!



Intel Quad-core “core i7”

Multi-core Processor (2/4)

- **Application domains for multiprocessors**
  - **Scientific computing/super-computing**
    - Example: weather simulation, protein folding
    - Each processor computes for a part of data
  - **Server workloads**
    - Example: airline reservation database
    - Many concurrent updates, searches, lookups, queries
    - Processors handle different requests
  - But software must be written to expose parallelism

Multi-core Processor (3/4)

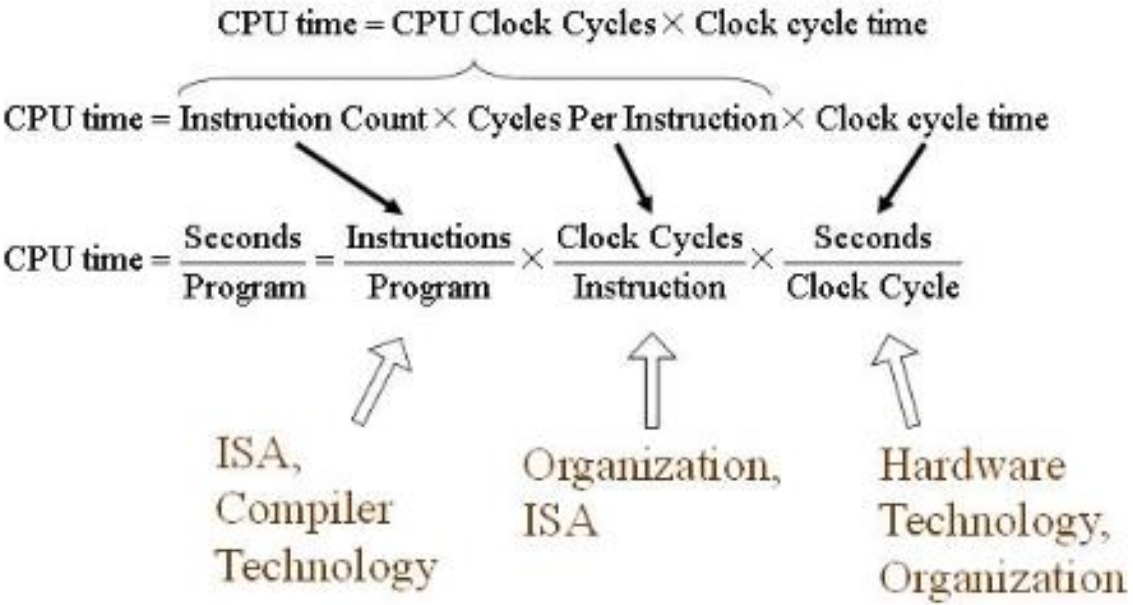
- **Multi-core & energy**
  - Explicit parallelism (multicore) is highly energy efficient
  - Example: Intel's Xscale:
    - 1GHz -> 200 MHz reduces energy used by 30X
    - What if we used 5 Xscales at 200 MHz?
    - Similar performance as 1GHz Xscale, but 1/6th the energy
    - 5 cores * 1/30th = 1/6th

Multi-core Processor (4/4)

- **Amdahl's law**
  - Consider a task with a "parallel" and "serial" portion
    - What is the speedup with N cores?
      - Speedup (n, p, s) = (s + p) / (s + (p/n))
      - p is "parallel percentage", s is "serial percentage"
    - What about infinite cores?
      - Speedup (p, s) = (s + p) / s = 1/s
    - Example: can optimize 50% of program A
      - Only yields a **2X** speedup

Pipelined Processors (1/5)

- **Microprocessor performance**
  - Iron law of performance

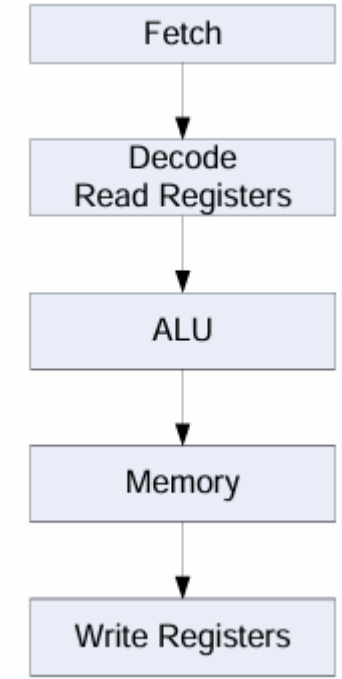


Pipelined Processors (2/5)

- **Microprocessor performance**
  - $CPI = CPI_{ideal} + CPI_{stall}$
    - CPI_{ideal}: cycles per instruction if no stall
  - CPI_{stall} contributors
    - Data dependences: RAW, WAR, WAW
    - Structural hazards
    - Control hazards: branches, exceptions
    - Memory latency: cache misses

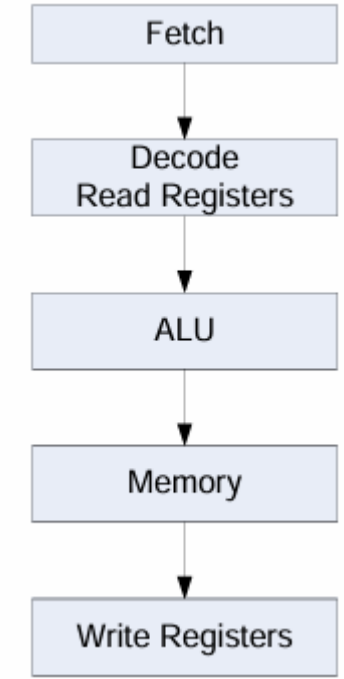
Pipelined Processors (3/5)

- **5-stage pipelined processors**
  - Advantages
    - CPI_{ideal} is 1 (pipelining)
    - No WAW or WAR hazards
    - Simple elegant
      - Still used in ARM & MIPs processors



Pipelined Processors (4/5)

- **5-stage pipelined processors**
  - Shortcomings
    - Upper performance bound is CPI = 1
    - High latency instructions not handle well
      - 1 stage for accesses to large cache or multiplier
      - Clock cycle is high
    - Unnecessary stalls due to rigid pipeline
      - If one instruction stalls anything behind it stalls



Pipelined Processors (5/5)

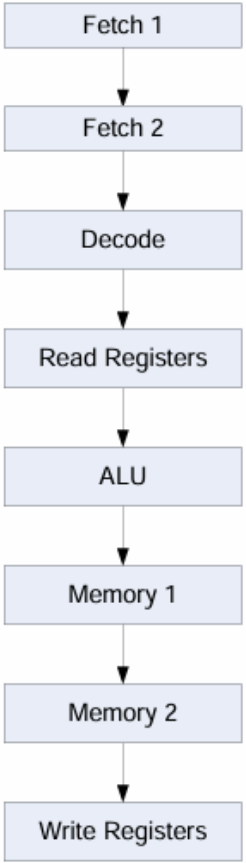
- **Improving 5-stage pipeline performance**
  - Higher clock frequency (lower CCT): **deeper pipelines**
    - Overlap more instructions
  - Higher CPI_{ideal}: **wider pipeline**
    - Insert multiple instruction in parallel in the pipeline
  - Lower CPI_{stall}:
    - **Diversified pipelines** for different functional units
    - **Out-of-order execution**
  - Balance conflicting goals
    - Deeper & wider pipelines => more control hazards
    - **Branch prediction**

Outline

- Multi-core Processor
- Pipelining Optimization
- Superscalar Processor
- Hardware Multi-Threading
- Vector Processors
- Graphics Processors

Pipelining Optimization (1/4)

- **Deeper pipelines**
  - Idea: break up instruction into N pipeline stages
    - Ideal CCT = 1/N compared to non-pipelined
  - **Other motivation for deep pipelines**
    - Not all basic operations have the same latency
      - Integer ALU, FP ALU, cache access
    - Difficult to fit them in one pipeline stage
      - CCT must be large enough to fit the longest one
    - Break some of them into multiple pipeline stages
      - E.g., data cache access in 2 stages, FP add in 2 stage, FP mul in 3 stage

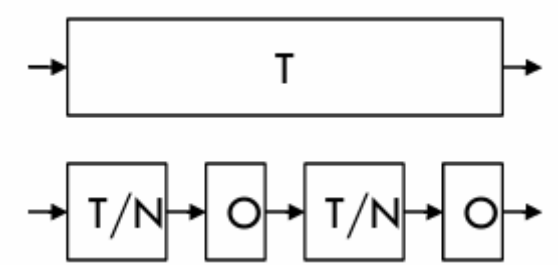


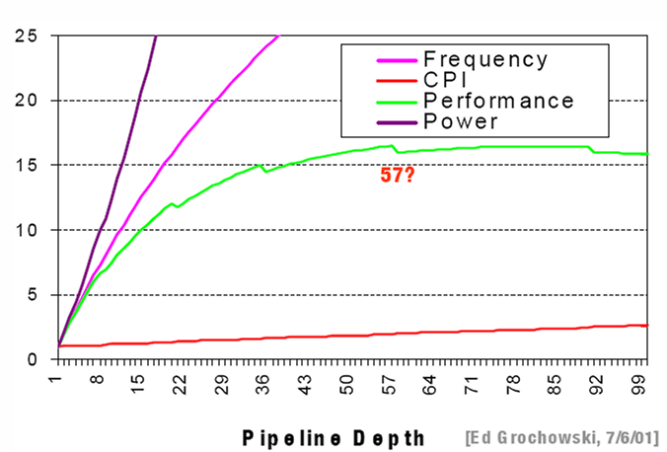
Pipelining Optimization (2/4)

- **Limits to pipeline depth**
  - Each pipeline stage introduces some overhead (O)
    - Delay of pipeline registers
    - Inequalities in work per stage
      - Cannot break up work into stages at arbitrary points
    - Clock skew
      - Clocks to different registers may not be perfectly aligned

Pipelining Optimization (3/4)

- **Limits to pipeline depth**
  - If original CCT was T, with N stages CCT is T/N + O
    - If N -> ∞, speedup = T/(T/N+O) -> T/O
    - Assuming that IC and CPI stay constant
  - Eventually overhead dominates and deeper pipelines have diminishing returns





Pipelining Optimization (4/4)

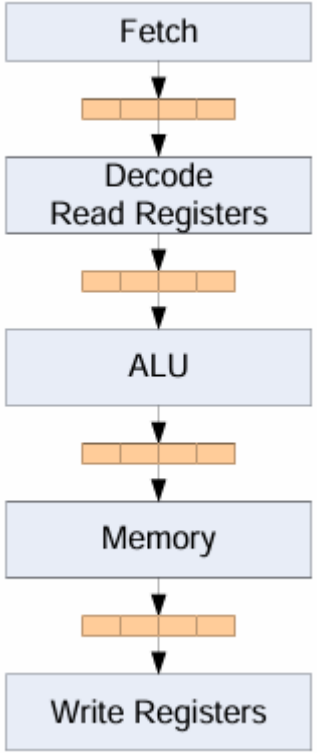
- **Deep pipelines review**
  - Advantages: higher clock frequency
    - The workhorse behind multi-GHz processors
  - Cost:
    - Complexity: more forwarding & stall cases
  - Disadvantages
    - More overlapping -> more dependencies -> more stall
      - CPI_{stall} grows **due to data and control hazards**
      - Clock overhead becomes increasing important

Outline

- Multi-core Processor
- Pipelining Optimization
- Superscalar Processor
- Hardware Multi-Threading
- Vector Processors
- Graphics Processors

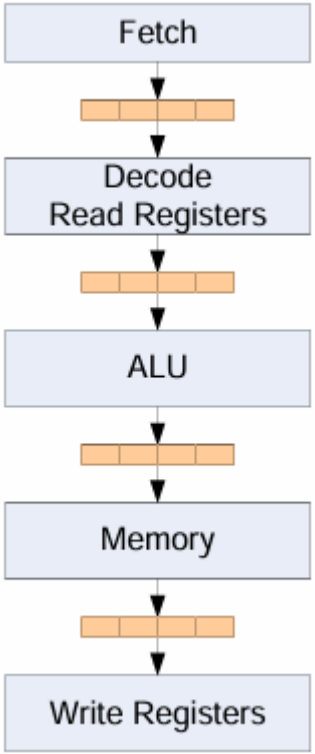
Superscalar Processor (1/7)

- **Superscalar (Wider) pipelines**
  - Idea: operate on N instructions each clock cycle
    - Known as wide or superscalar pipelines
    - $CPI_{ideal} = 1/N$
  - Options (from simpler to harder)
    - One integer and one floating-point instruction
    - Any N = 2 instructions
    - Any N = 4 instructions
    - Any N = ? Instructions
      - What are the limits here?



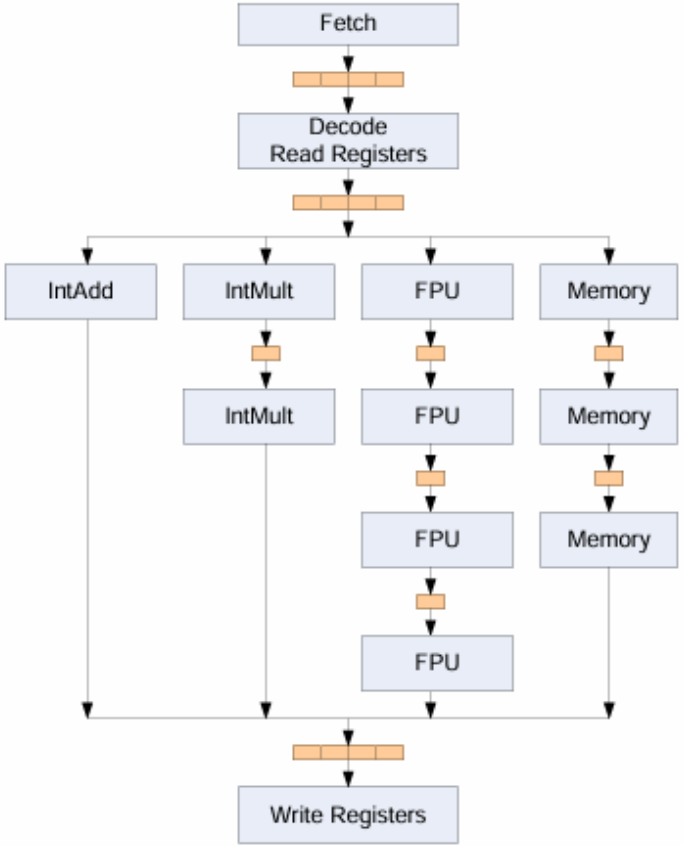
Superscalar Processor (2/7)

- **Superscalar (Wider) pipelines**
  - Advantages: Lower CPI_{ideal} (1/N)
  - Cost
    - Need wider path to instruction cache
    - Need more ALUs, register file ports…
    - Complexity: more forwarding & stall cases to check
  - Disadvantages
    - Parallel execution -> more dependencies -> more stalls
    - CPI_{stall} grows due to data and control hazards



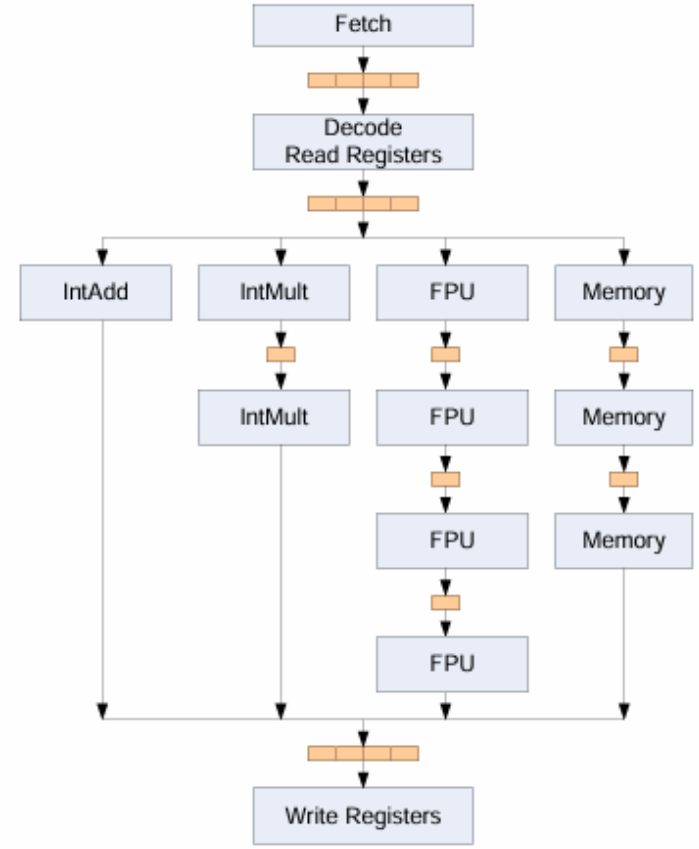
Superscalar Processor (3/7)

- **Diversified pipelines**
  - Idea: decouple the execution portion of the pipeline for different instructions
  - Common approach
    - Separate pipelines for simple integer, integer multiply, FP, LD/ST



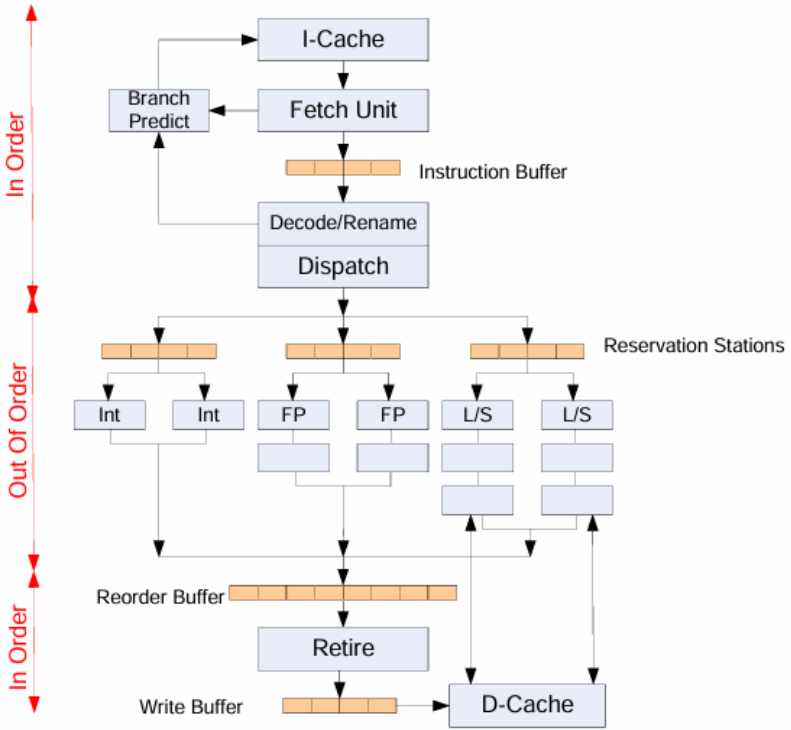
Superscalar Processor (4/7)

- **Diversified pipelines**
  - Advantage
    - Avoid unnecessary stalls
      - E.g. slow FP instruction does not block independent integer instruction
  - Disadvantages
    - WAW hazards



Superscalar Processor (5/7)

- **Superscalar out-of-order processor**



Superscalar Processor (6/7)

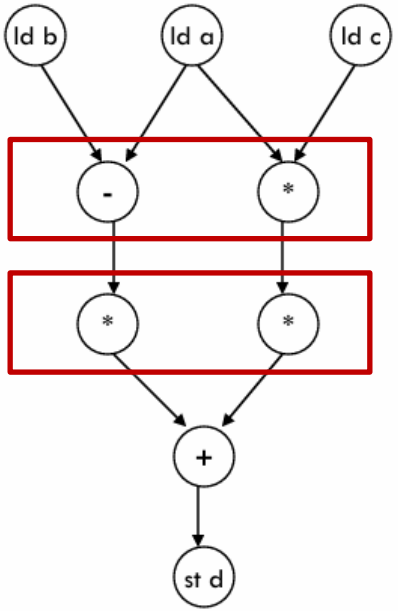
- **Instruction Level Parallelism (ILP)**

$$D = 3(a - b) + 7ac$$

- **Sequential execution order**

```
ld a
ld b
sub a-b
mul 3(a-b)
ld c
mul ac
mul 7ac
add 3(a-b)+7ac
st d
```

- **Data-flow execution order**



Superscalar Processor (7/7)

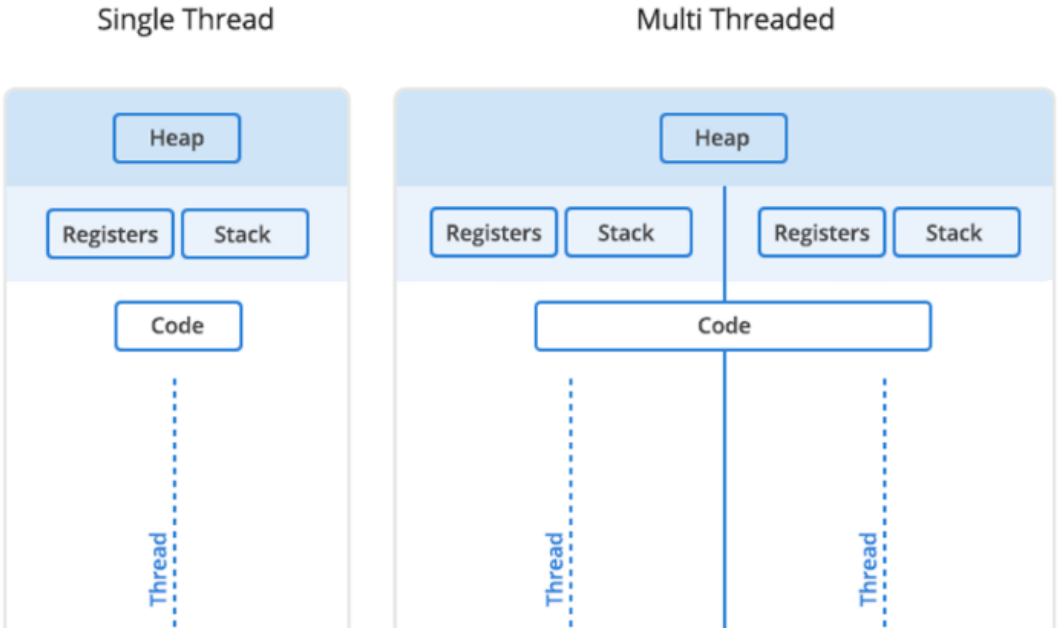
- **Challenges of Superscalar Processors**
  - Clock frequency: Getting close to pipeline limits
    - Clocking overheads, CPI degradation
  - Branch prediction & memory latency limit the practical benefits of out-of-order (OOO) execution
  - Power grows super-linearly with higher clock & more OOO logic
  - Design complexity grows exponentially with issue width
  - Limit ILP -> **must exploit TLP and DLP**
    - **Thread-Level Parallelism**: Multithreading and multicore
    - **Data-Level Parallelism**: SIMD instructions

Multi-Threading (1/5)

- **Software "thread":** Independent flows of execution
  - "Per-thread" state
    - Context state: PC, registers
    - Stack (per-thread local variables)
  - "Shared" state: global variables, heap, etc.
  - Threads generally share the same memory space
    - A process is like a thread, but with its own memory space
    - Java has thread support built in, C/C++ use the pthreads lib

Multi-Threading (2/5)

- **Software "thread":** Independent flows of execution
  - A thread is the unit of execution within a process



Multi-Threading (3/5)

- **Software "thread":** Independent flows of execution
  - **Thread**: instruction stream with own PC and data
  - Each thread has all the state (instructions, data, PC, register state, …) necessary to allow it to execute
  - System software (the O.S.) manages threads
  - "Thread scheduling", "context switching"
  - In single-core system, all threads share one processor
    - Hardware timer interrupt occasionally triggers O.S.
    - Quickly swapping threads gives illusion of concurrent execution

Multi-Threading (4/5)

- **Shared Memory Programming Model**
  - Programmer explicitly creates multiple threads
  - All loads & stores to a single shared memory space
    - Each thread has its own stack frame for local variables
    - All memory shared, accessible by all threads
  - Multi-threading is commonly used
    - Handling user interaction (GUI programming)
    - Handling I/O latency
    - Expressing parallel work via **Thread-Level Parallelism (TLP)**

Multi-Threading (5/5)

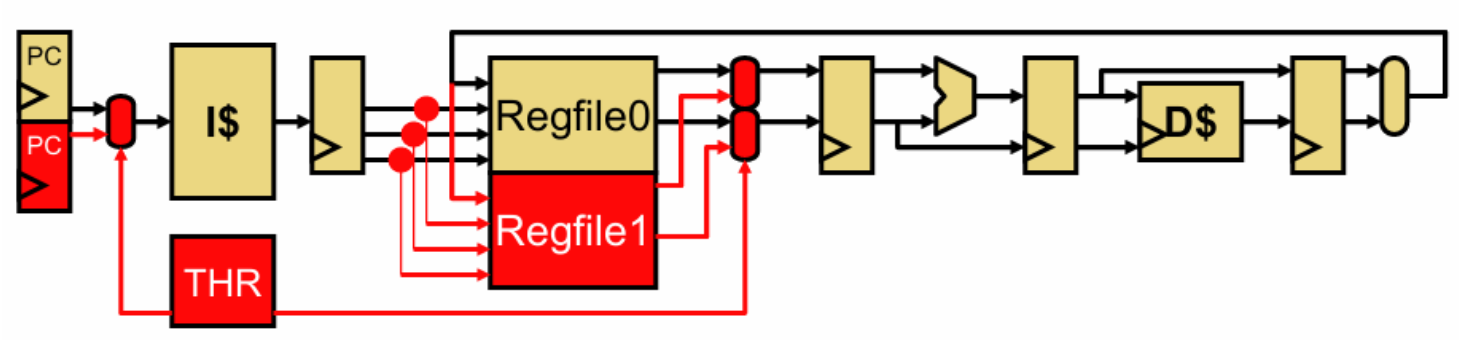
- **Shared memory issues**
  - **Cache coherence**
    - If cores have private (non-shared) caches
    - How to make writes to one cache “show” up in others?
  - **Parallel programming**
    - How does the programmer express the parallelism?
  - **Synchronization**
    - How to regulate access to shared data?
    - How to implement “locks”?
  - **Memory consistency models**
    - How to reconcile shared memory with compiler optimizations, store buffers, and out-of-order execution?

Outline

- Multi-core Processor
- Pipelining Optimization
- Superscalar Processor
- Hardware Multi-Threading
- Vector Processors
- Graphics Processors

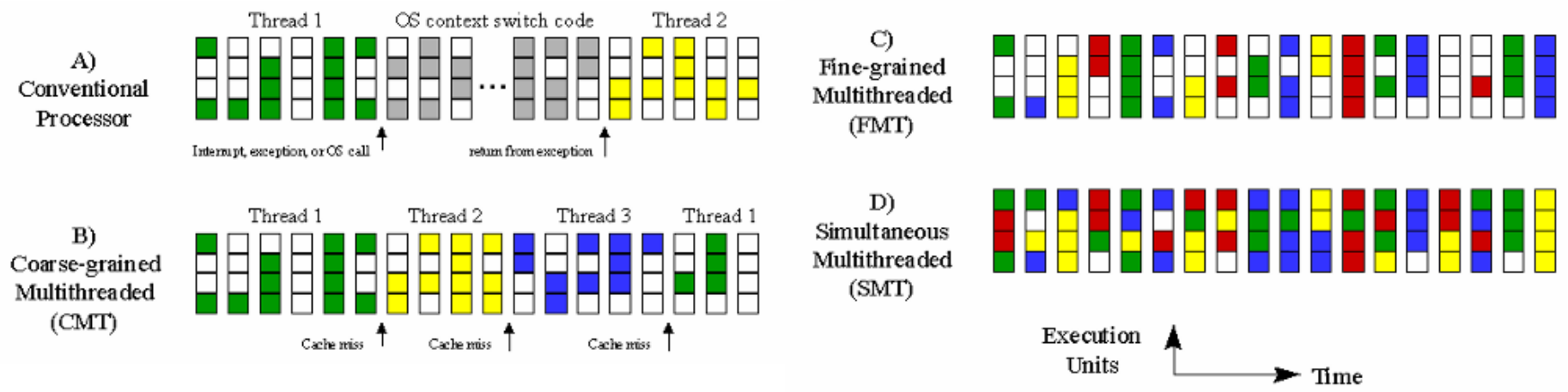
Hardware Multi-Threading (1/5)

- A **hardware thread** is a sequential stream of instructions
- **Hardware Multi-threading (MT)**
  - Multiple hardware threads dynamically share a single pipeline
  - Replicate only per-thread structures: program counter & registers
  - Hardware interleaves instructions



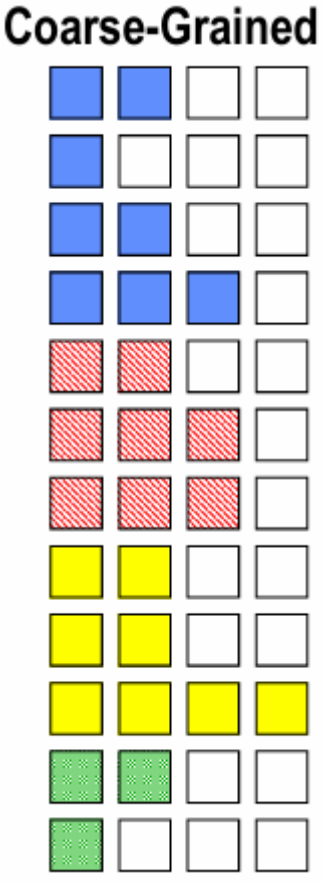
Hardware Multi-Threading (2/5)

- **Motivation**
  - Super-scalar hardware underutilized on stalls
  - Use TLP to increase utilization



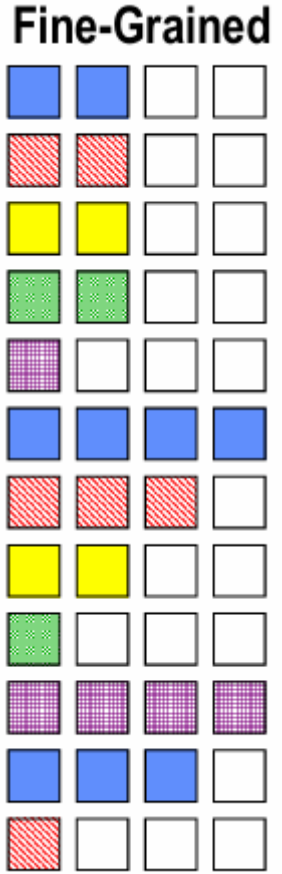
Hardware Multi-Threading (3/5)

- **Coarse-grained multithreading**
  - Switches threads **only on costly stalls**, such as L2 cache misses
  - Advantages
    - Relieves need to have very fast thread-switching
  - Disadvantages
    - Throughput losses from shorter stalls, due to start-up costs
    - New thread must fill pipeline before instructions complete
  - Better for reducing penalty of high cost stalls, where pipeline refill << stall time



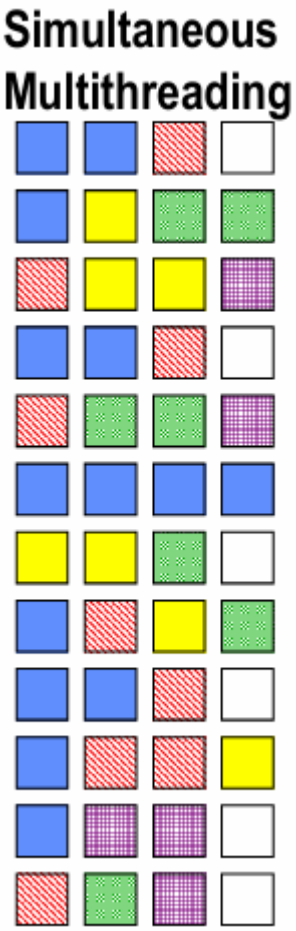
Hardware Multi-Threading (4/5)

- **Fine-grained multithreading**
  - Switches threads on each instruction
    - The execution of multiple threads to be interleaved
  - Advantages
    - Hide both short and long stalls, since instructions from other threads execute when one stalls
  - Disadvantages
    - Slows down execution of individual threads
    - A thread without stalls will be delayed by instructions from other threads



Hardware Multi-Threading (5/5)

- **Simultaneous multithreading (SMT)**
  - Exploiting TLP in a single processor core
  - Each clock, core chooses instructions from multiple threads to run on ALUs
  - Needs one context per thread
  - Benefits
    - Increasing throughput from concurrent execution
    - Dynamic scheduling
    - No partitioning of many resources
    - E.g. Intel Hyper-threading



SIMD Processing (1/2)

- Single Instruction Multiple Data (SIMD)
  - The instruction sequence applies for multiple elements
  - Vector processing -> amortize instruction costs (fetch, decode …) across multiple operations
  - Require regular data parallelism (no or minimal divergence)
- Exploiting SIMD
  - Explicit & low-level, using vector intrinsics
  - Explicit & high-level convey parallel semantics (e.g. foreach)
  - Implicit: parallel compiler infers loop dependencies

SIMD Processing (2/2)

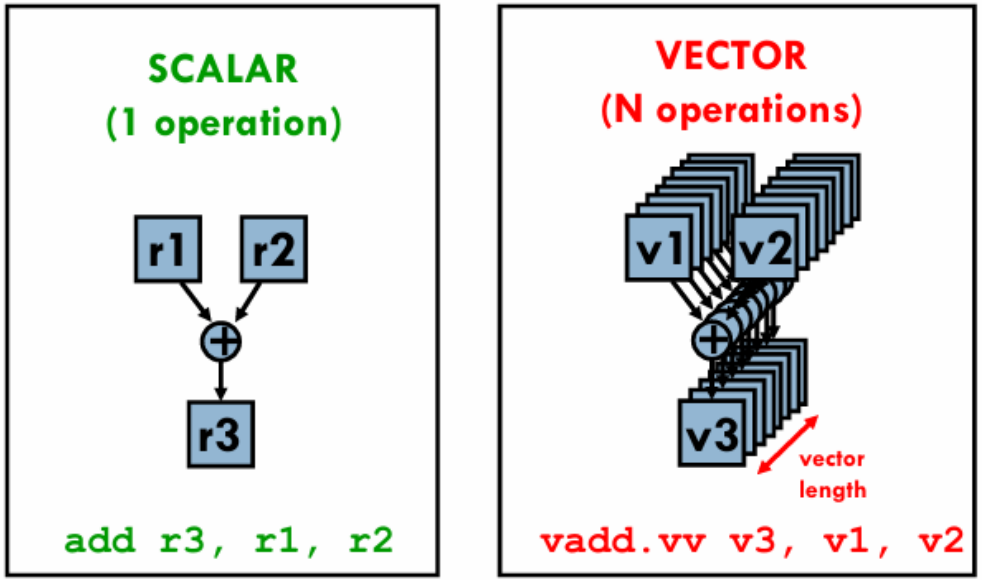
- SIMD Extensions on Modern CPUs
  - SSE: 128-bit operands (4x32-bit or 2x64-bit)
  - AVX: 256-bit operands (8x32-bit or 4x64-bit)
  - Explicit SIMD: parallelization performed at compile time

Outline

- Multi-core Processor
- Pipelining Optimization
- Superscalar Processor
- Hardware Multi-Threading
- Vector Processors
- Graphics Processors

Vector Processors (1/12)

- Scalar processors operate on single numbers (scalars)
- Vector processors operate on linear sequences of numbers (vector)

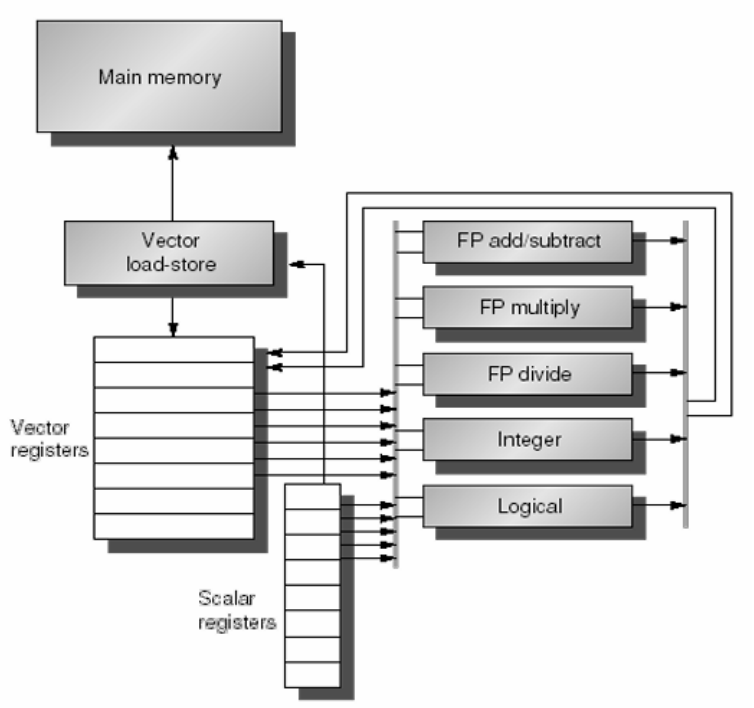


Vector Processors (2/12)

- What's in a vector processor?
  - A scalar processor
    - Scalar register file (32 registers)
    - Scalar functional units (arithmetic, load/store, etc.)
  - A vector register file (a 2D register array)
    - Each register is an array of elements
    - E.g. 32 registers with 32 64-bit elements per register
    - **MVL = maximum vector length** = max # of elements per register
  - A set of vector functional units
    - Integer, FP, load/store, etc..

Vector Processors (3/12)

- What's in a vector processor?



Vector Processors (4/12)

- Basic vector ISA

Instr.	Operands	Operation	Comment
VADD.**VV**	V1,V2,V3	V1=V2+V3	vector + vector
VADD.**SV**	V1,**R0**,V2	V1=**R0**+V2	scalar + vector
VMUL.VV	V1,V2,V3	V1=V2*V3	vector x vector
VMUL.SV	V1,R0,V2	V1=R0*V2	scalar x vector
VLD	V1,R1	V1=M[R1...R1+63]	load, stride=1
VLD**S**	V1,R1,**R2**	V1=M[R1...R1+**63*R2**]	load, stride=R2
VLD**X**	V1,R1,**V2**	V1=M[R1+**V2**$_i$,i=0..63]	indexed("gather")
VST	V1,R1	M[R1...R1+63]=V1	store, stride=1
VST**S**	V1,R1,**R2**	V1=M[R1...R1+**63*R2**]	store, stride=R2
VST**X**	V1,R1,**V2**	V1=M[R1+**V2**$_i$,i=0..63]	indexed("scatter")

+ regular scalar instructions...

Vector Processors (5/12)

- Advantages of vector ISAs
  - Compact: single instruction defines N operations
    - Amortizes the cost of instruction fetch/decode/issue
    - Also reduces the frequency of branches
  - Parallel: N operations are (data) parallel
    - No dependencies
    - No need for complex hardware to detect parallelism
    - Can execute in parallel assuming N parallel datapaths

Vector Processors (6/12)

- Advantages of vector ISAs
  - Expressive: memory operations describe patterns
    - Continuous or regular memory access pattern
    - Can prefetch or accelerate using wide/multi-banked memory
    - Can amortize high latency for 1st element over large sequential pattern

Vector Processors (7/12)

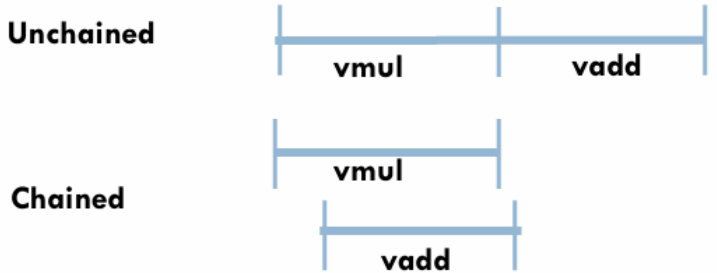
- **Vector Length (VL)**
  - Basic: Fixed vector length (typically in narrow SIMD)
- **Vector-length (VL) register**
  - Control the length of any vector operation, including vector loads and stores
    - E.g. vadd.vv with VL = 10 <->
      for (i = 0; i < 10; i++) V1[i] = V2[i] + V3[i]
    - VL can be set up to MVL (e.g. 32)

Vector Processors (8/12)

- **Optimization 1: Chaining**
  - Suppose the following code with VL = 32

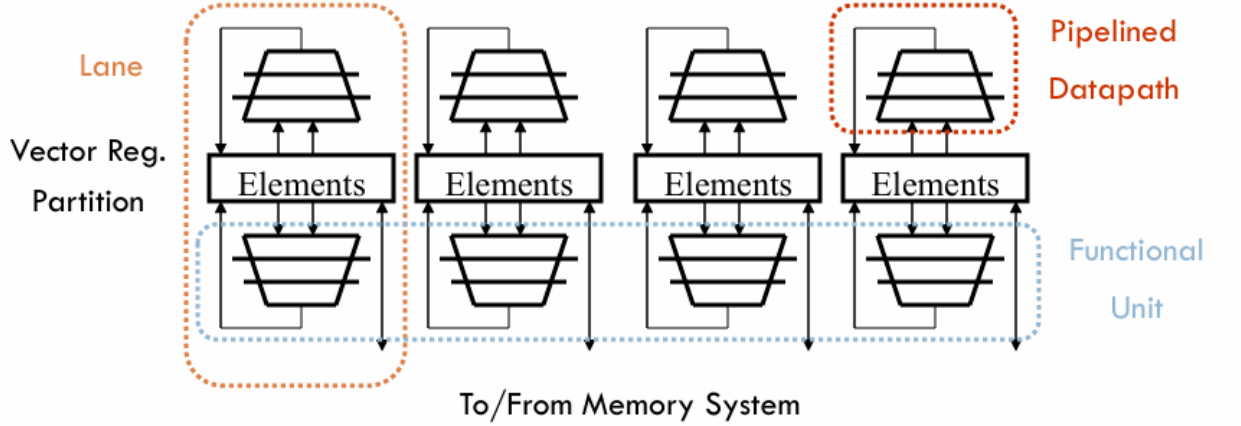
```
vmul.vv    V1,V2,V3
vadd.vv    V4,V1,V5      # very long RAW hazard
```

  - **Chaining**
    - V1 is not a single entity but a group of individual elements
    - Pipeline forwarding can work on an element basis
    - Allow vector to chain to any other active vector operations => **more R/W ports**



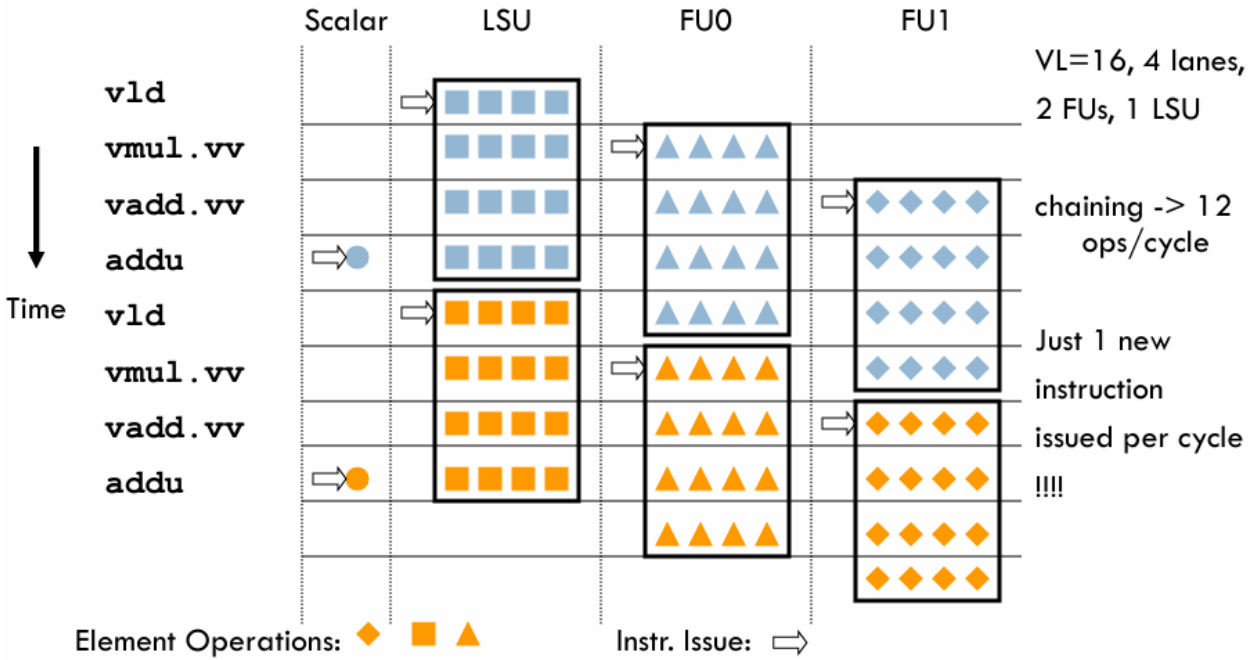
Vector Processors (9/12)

- **Optimization 2: Multiple Lanes**
  - Elements for each vector register interleaved across the lanes
  - Each lane receives identical control
  - Multiple element operations executed per cycle
  - No need for inter-lane communication for most vector insns



Vector Processors (10/12)

- **Chaining & Multi-lane example**



Vector Processors (11/12)

- **Optimization 3: Conditional Execution**
  - Suppose you want to vectorize this:

```
for (i=0; i<N; i++) if (A[i]!= B[i]) A[i] -= B[i];
```

  - Solution: Vector conditional execution (predication)
    - Add vector flag registers with single-bit elements (masks)
    - Use a vector compare to set a flag register
    - Use flag register as mask control for the vector sub
      - Add executed only for vector elements with corresponding flag element set

Vector Processors (12/12)

- **Optimization 3: Conditional Execution**
  - Solution: Vector conditional execution (predication)
    - Use flag register as mask control for the vector sub
      - Add executed only for vector elements with corresponding flag element set

```
vld          V1, Ra
vld          V2, Rb
vcmp.neq.vv  M0, V1, V2      # vector compare
vsub.vv      V3, V2, V1, M0  # conditional vadd
vst          V3, Ra
```

Outline

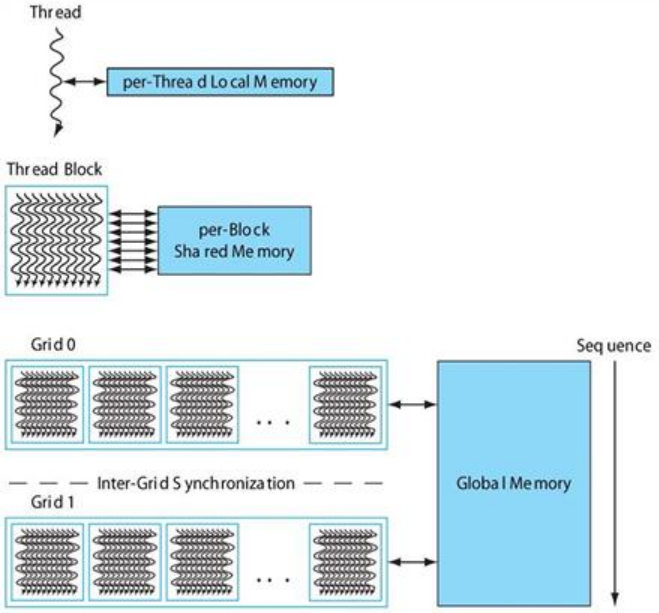
- Multi-core Processor
- Pipelining Optimization
- Superscalar Processor
- Hardware Multi-Threading
- Vector Processors
- Graphics Processors

Graphics Processors (1/8)

- Till mid 90s
  - VGA controllers used to accelerate some display functions
- Mid 90s to mid 00s
  - Fixed-function graphics accelerators for the OpenGL and DirectX APIs
  - 3D graphics: triangle setup & rasterization, texture mapping …
- Modern GPUs
  - Programmable multiprocessors optimized for data-parallel app
    - OpenGL/DirectX/CUDA/OpenCL …
  - Some fixed-function hardware (texture, raster ops …)

Graphics Processors (2/8)

- Software GPU Thread Model (CUDA)
  - Single-program multiple data (SPMD)
  - Each thread has local memory
  - Parallel threads packed in blocks
    - Access to per-block shared memory
    - Synchronize with barrier
  - Grids include independent groups



Graphics Processors (3/8)

- In SAXPY example
  - CUDA code launches 256 threads per block
    - Thread = 1 iteration of scalar loop (1 element in vector loop)
    - Block = body of vectorized loop (with VL = 256 in this ex.)
    - Grid = vectorizable loop

C Code

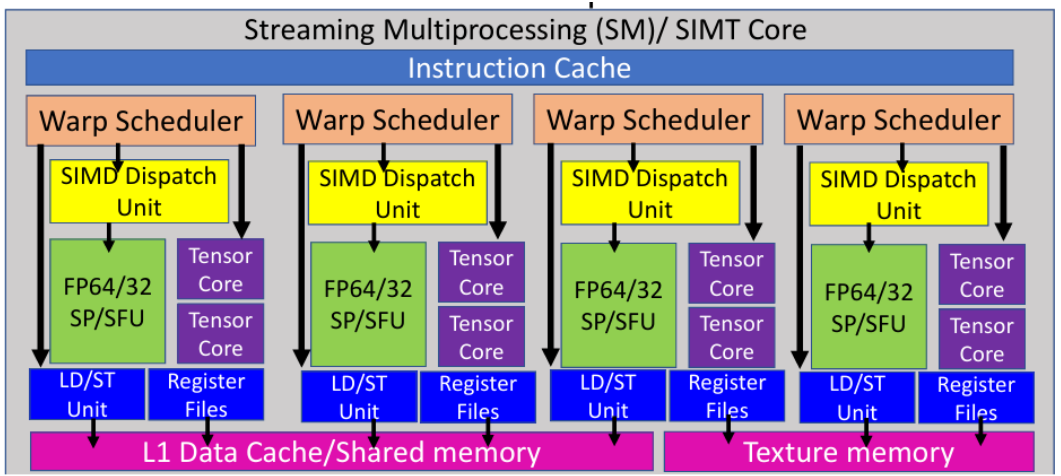
```
// Invoke DAXPY
daxpy(n, 2.0, x, y);
// DAXPY in C
void daxpy(int n, double a, double *x, double *y)
{
        for (int i = 0; i < n; ++i)
                y[i] = a*x[i] + y[i];
}
```

CUDA Code

```
// Invoke DAXPY with 256 threads per block
__host__
int nblocks = (n+ 255) / 256;
    daxpy<<<nblocks, 256>>>(n, 2.0, x, y);
// DAXPY in CUDA
__device__
void daxpy(int n, double a, double *x, double *y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n) y[i] = a*x[i] + y[i];
}
```

Graphics Processors (4/8)

- 15 SMX processors, shared L2, 6 memory controllers

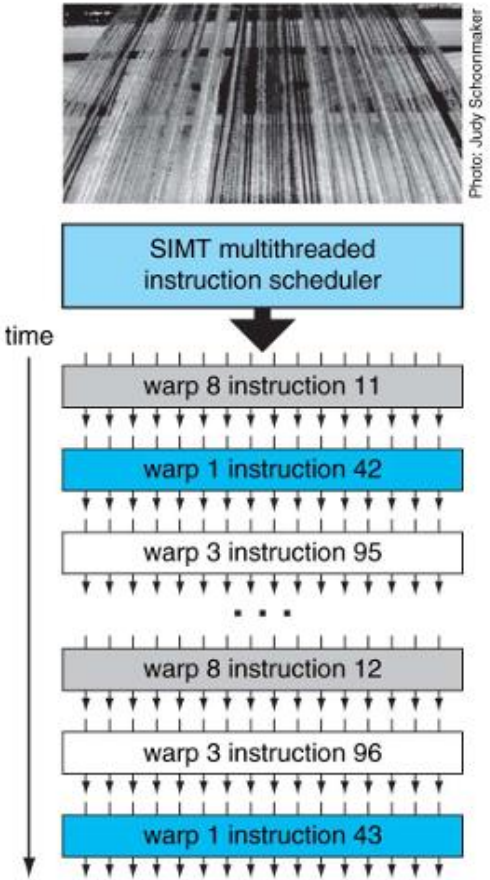


Graphics Processors (5/8)

- Cores are
  - Multithreaded
  - Data parallel
- Capabilities
  - 64K registers
  - 192 simple cores
    - Integer and SP FPU
  - 64 DP FPUs
- Scheduling
  - 4 warp schedulers, 2 instruction dispatch per warp

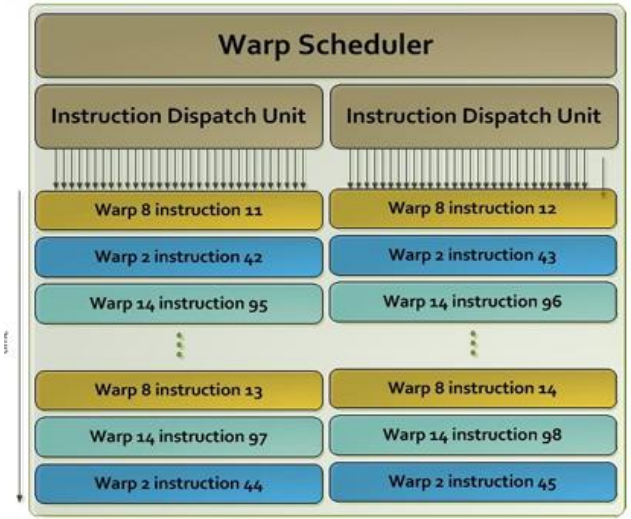
Graphics Processors (6/8)

- All threads can be independent
  - HW implements zero-overhead switching
- 32 threads are packed in warps
  - **Warp**: set of parallel threads that execute the same instruction-> data parallelism
  - 1 warp instruction keeps cores busy for multiple cycles
- SW thread blocks mapped to warps
  - When HW resources are available



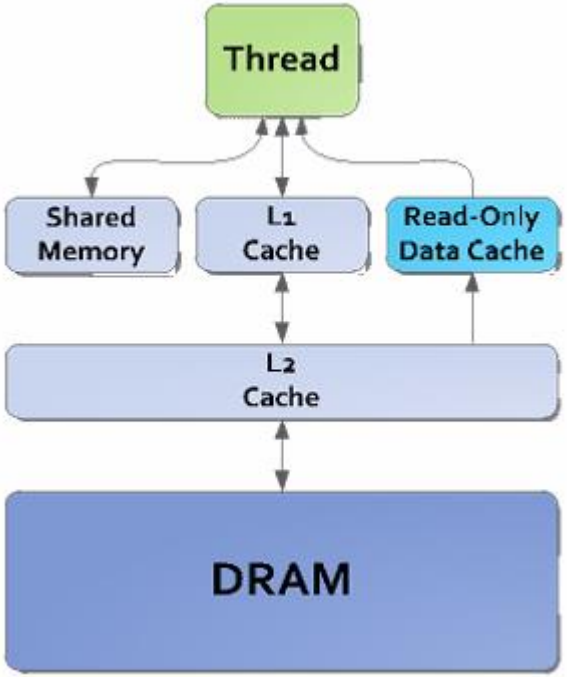
Graphics Processors (7/8)

- 64 warps per SMX
- 32 threads per warp
  - 64K registers/SMX
  - Up to 255 registers per threads (8 warps)
- Scheduling
  - 4 schedulers select 1 warp per cycle
  - 2 independent instructions issued per warp (double-pumped FUs)
  - Total bandwidth = 4 x 2 x 32 = 256 ops per cycle



Graphics Processors (8/8)

- Each SMX has 64KB of memory
  - Split between shared mem and L1 cache
  - 256 Bytes per access
  - 48KB read-only data cache
  - 1.5MB shared L2
    - Supports synchronization operations (atomicCAS, atomicADD …)
  - Throughput-oriented main memory
    - GDDRx standards



Conclusion

- Instruction-Level Parallelism (ILP)
  - Pipelining, super-scalar processor
- Thread-Level Parallelism (TLP)
  - Hardware multi-threading
- Data-Level Parallelism (DLP)
  - SIMD, Vector processor, GPU