

CoE3DR4

Computer Organization

Chapter 7: Multicores, Multiprocessors and Clusters



Outline

- 7.1 Introduction
- 7.2 Difficulty of Parallel Processing Programs
- 7.3 Shared Memory Multiprocessors
- 7.4 Clusters and Message-Passing Multiprocessors
- 7.5 Hardware Multithreading
- 7.6 SISD, MIMD, SIMD, SPMD and Vector
- 7.7 GPU
- 7.8 Multiprocessor Network Topologies

7.1 Introduction

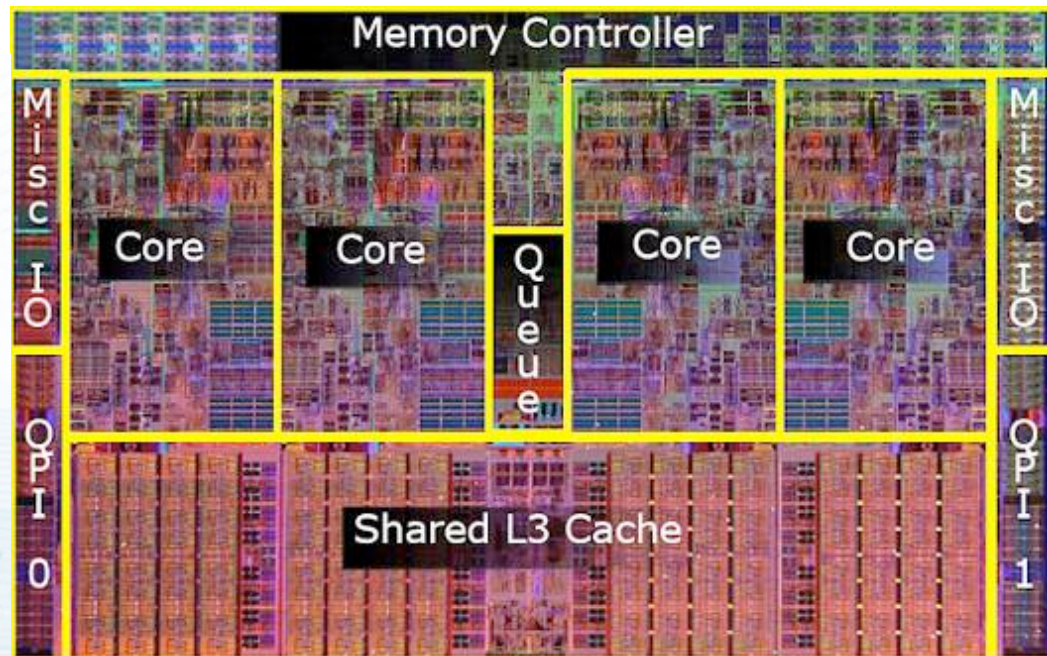
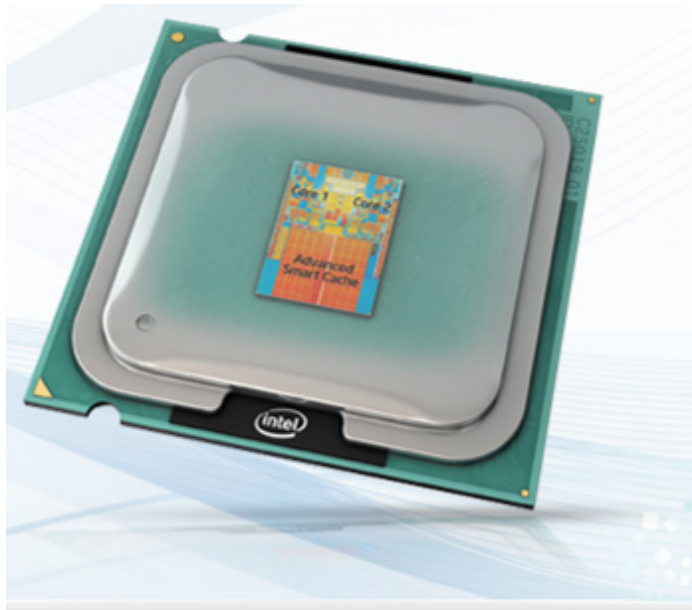
- Multiprocessing: the use of multiple concurrent processes or processors in a system as opposed to a single process or processor at any one instant.
- This is actually good because if one processor fails then the other one can take over

Multiprocessor System

- Power and Speed has become a major issue for Datacenters and Microprocessors
- Thus replacing large inefficient processors with many smaller, efficient processors delivers better performance per watt
- This is know as Multiprocessor System, a computer system with at least two or more processor

7.2 Difficulty of Parallel Processing Programs

- Multi-Core: two or more processors have been attached
 - enhanced performance
 - reduced power consumption and
 - more efficient simultaneous processing of multiple tasks



But is it easy?

- It is not the hardware, that makes parallelism hard. It is that too few application programs have been rewritten to complete tasks sooner on multiprocessors.
 - must get better performance
 - must get better efficiency
 - how to write code for multiprocessors
 - may need to rewrite software

The Difficulty of Creating Parallel Processing Programs

- Writing parallel software is difficult, and difficulty increases as # of cores/chip increases.
- *MUST* achieve better performance and efficiency from a parallel processing program on a multiprocessor
 - Otherwise, a faster single processor can complete the task of increasing performance (increased clock speed)

The Difficulty of Creating Parallel Processing Programs

cont'd

- Challenges:
 - Scheduling
 - the way various processes are assigned in multitasking and multiprocessing
 - Load balancing
 - refers to balancing a workload amongst multiple processors
 - Synchronization
 - establishing consistency among data on multiple processors and the continuous harmonization of the data over time
 - Communication between processes

Amdahl's Law

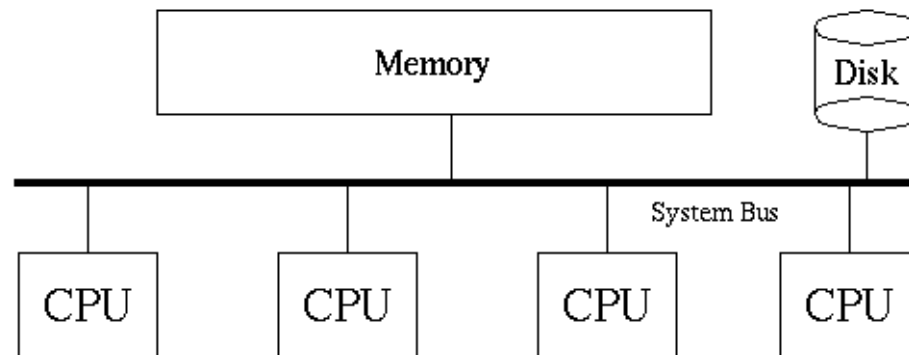
- Sequential part can limit speedup
 - $T_{\text{new}} = T_{\text{parallelizable}} / \text{Processors} + T_{\text{sequential}}$
- Do examples on the board

Strong Scaling vs. Weak Scaling

- Strong Scaling:
 - Speedup achieved on a multi-processor WITHOUT increasing the size of the problem
- Weak Scaling:
 - Speedup achieved on a multi-processor while increasing the size of the problem proportional to the increase in the number of processors

7.3 Shared Memory Multiprocessors

- SMP: shared memory multiprocessor
 - Single physical address space across all processors
 - Synchronization is the process of coordinating the behaviour of two or more process, which may be running on different processors
 - Synchronization mechanisms (eg: LOCK) allows access to data to only one processor at a time to avoid data inconsistencies



Shared Memory Machine

Shared Memory Multiprocessors

Memory access time

- UMA (uniform)
 - Takes same amount of time to access main memory regardless of which processor has requested data
- NUMA (non-uniform)
 - Memory access times are faster than others depending on which processor has requested the data

A simple Parallel Processing Program for Shared Address Space

- Suppose we want to sum 100,000 numbers on 100 processor
 - Pn is the number that identifies the processor 0-99
 - We divide 1000 numbers per processor

```
sum[Pn] = 0;
```

```
for(i=1000*Pn; i<1000*(Pn+1); i=i+1)
```

```
    sum[Pn] = sum[Pn] + A[i];
```

- Now need to add these partial sums
 - Reduction: divide and conquer
 - Half of the processors add pairs, then quarter, and so on, until we have final sum
 - Processors must synchronize between reduction steps

A simple Parallel Processing Program.... Cont'd

```
half = 100;
```

```
repeat
```

```
  synch();
```

```
  if (half%2 != 0 && Pn == 0)
```

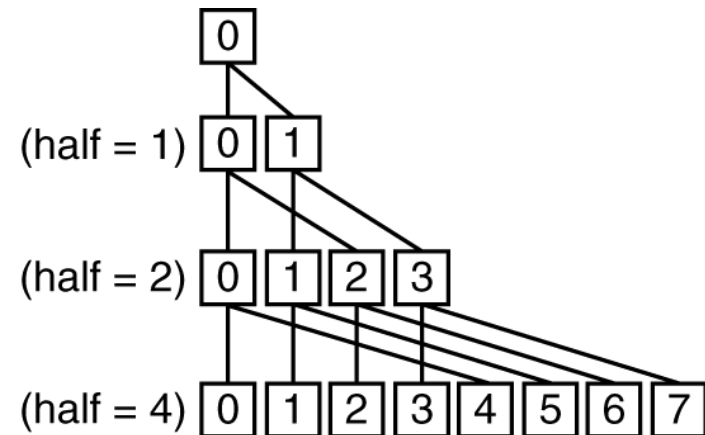
```
    sum[0] = sum[0] + sum[half-1];
```

```
    /* Conditional sum needed when half is odd; Processor 0  
    gets missing element */
```

```
    half = half/2; /* dividing line on who sums */
```

```
    if (Pn < half) sum[Pn] = sum[Pn] + sum[Pn+half];
```

```
until (half == 1);
```

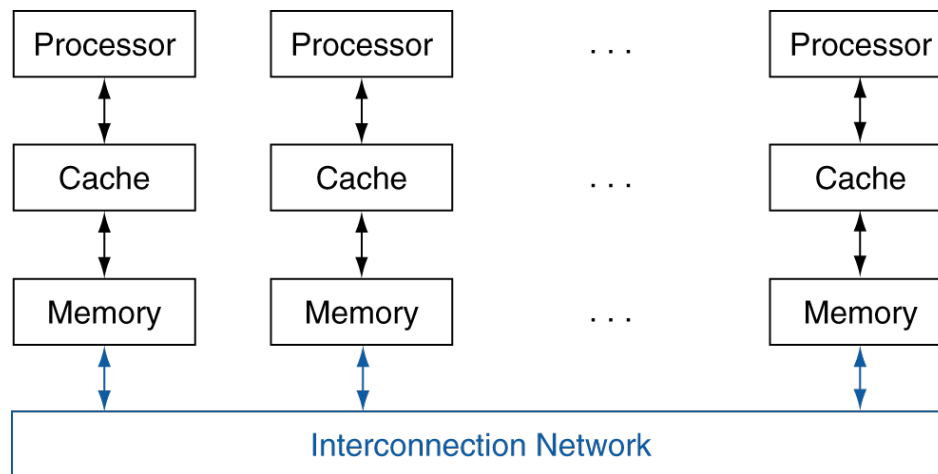


7.4 Clusters and Message-Passing Multiprocessors

- The alternative approach to sharing an address space is for the processors to each have their own private physical address space. Need to communicate.
- Message Passing: Communication between multiple processors by explicitly sending and receiving information.
- Send/Receive message routine: A routine used by a processor in machines to pass to/accept from another processor

Clusters and Other Message-Passing Multiprocessors

- Each processor has private physical address space
- Hardware sends/receives messages between processors through I/O interconnect via standard network switches and cable

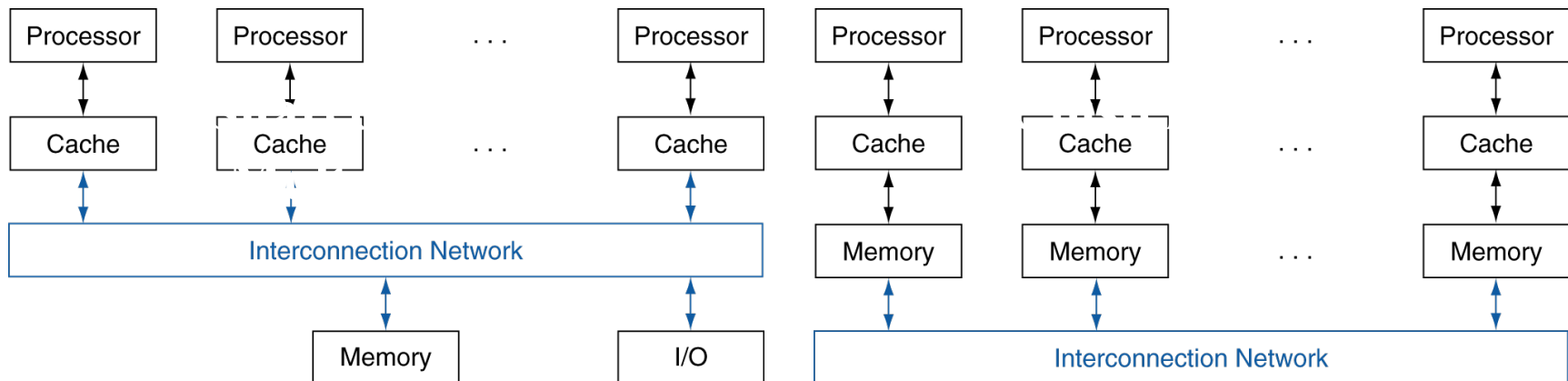


Clusters

- Clusters: Collections of computers connected via I/O over standard network switches to form a message-passing multiprocessor.
- Drawbacks of Clusters
 - High cost of administration
 - Lower bandwidth and higher latency of the interconnects relative to multicore processors
 - Overhead in the division of memory (e.g., n copies of the OS)
- Advantages
 - Easier to replace a machine without bringing down the system
 - Easier to expand the system

Cluster Computing Drawbacks

- Cost of administering a cluster of N machines is same as cost of administering N independent machines
- Cost of administering an SMP (shared memory processors) with N processors is the same as administering a single machine



7.5 Hardware Multithreading

- Thread: smallest schedulable unit
- On a single processor, **multithreading** generally occurs by time-division multiplexing: the processor switches between different threads.
- On a multiprocessor or multi-core system, the threads or tasks will generally run at the same time, with each processor or core running a particular thread or task
- Three different hardware Multithreading architectures are
 - 1) Fine-Grained Multithreading
 - 2) Coarse – Grained Multithreading
 - 3) Simultaneous Multithreading

Hardware Multithreading

- Performing multiple threads of execution in parallel
 - Each thread must have a duplicate of register file and PC
 - Memory can be shared via VM mechanisms
- Fine-grain multithreading
 - Switching between threads after each cycle
 - Interleaved execution of multiple threads
 - When one thread stalls, the next thread is executed
 - **Advantage:** can hide throughput losses that arise from stalls
 - **Disadvantage:** slows down the execution of individual threads b/c of stalls during exec of other threads

Fine Grained Multithreading

- A higher performance type of multithreading is where the processor switches threads every CPU cycle. For example:
 - Cycle i : an instruction from thread A is issued
 - Cycle $i+1$: an instruction from thread B is issued
 - Cycle $i+2$: an instruction from thread C is issued
- This type of interleaving is often done in round robin fashion, skipping any threads that are stalled at that time
- Primary advantage of this architecture is that it can hide the throughput losses that arise from both long and short stalls

Hardware Multithreading cont'd

- Coarse-grain multithreading
 - Only switch on long stall (L2-cache miss)
 - Less likely to slow down the execution of an individual thread since instructions from other threads will only be issued when a thread encounters a long stall
 - **Advantage:** Reduces the penalty of high cost stalls when pipeline refill is negligible compared to stall time
 - **Disadvantage:** Limited in its ability to overcome throughput losses from short stalls eg, data hazards

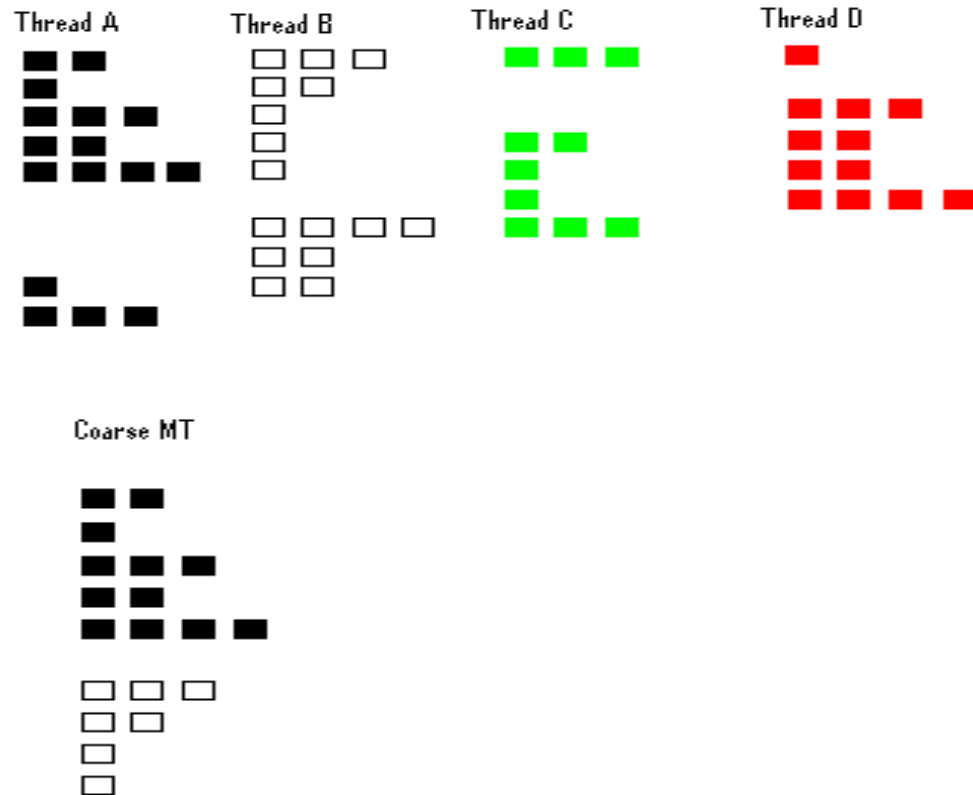
Coarse Grained Multithreading

- This change relieves the need to have thread switching and the slowdown in the execution of individual thread is less
- The major draw back of this hardware is the throughput losses from shorter stalls
- When a stall occurs the pipeline has to be emptied or frozen and the new thread that begins execution has to fill up the pipeline before starting the execution
- Due to this coarse grained architecture is more useful for reducing the penalty of high cost stalls where pipeline refill is negligible compared to the stall time

Coarse Grained Multithreading

- In this version of hardware multithreading thread is switched on costly stalls, such as second level cache miss
- Cycle i : instruction j from thread A is issued
- Cycle $i+1$: instruction $j+1$ from thread A is issued
- Cycle $i+2$: instruction $j+2$ from thread A is issued, load instruction which misses in all caches
- Cycle $i+3$: thread scheduler invoked, switches to thread B
- Cycle $i+4$: instruction k from thread B is issued
- Cycle $i+5$: instruction $k+1$ from thread B is issued

Coarse Grained Multithreading



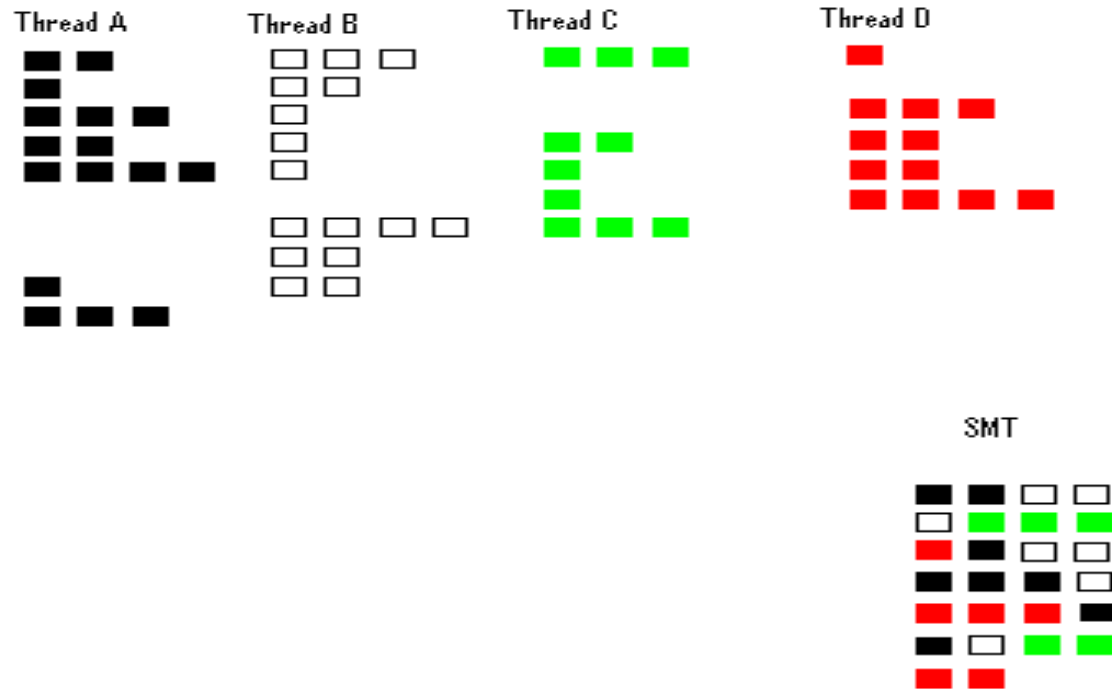
Simultaneous Multithreading

- In multiple-issue dynamically scheduled processor
 - Instruction are scheduled from multiple threads
 - Make use of thread level parallelism and instruction level parallelism
 - Instructions from independent threads execute when functional units are available eg, IF, ID, MEM etc...
- Example: Sun UltraSPARC T2 (Niagara 2)
 - Fine-grain multi-threading

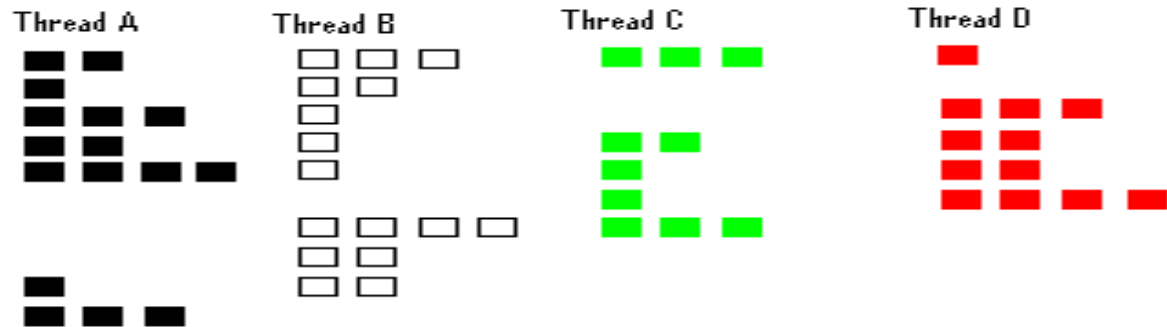
Simultaneous Multithreading

- In simultaneous multithreading, instructions from more than one thread can be executing in any given pipeline stage at a time
- This is done without great changes to the basic processor architecture: the main additions needed are the ability to fetch instructions from multiple threads in a cycle, and a larger register file to hold data from multiple threads
- The technique is really an efficiency solution and there is inevitable increased conflict on shared resources

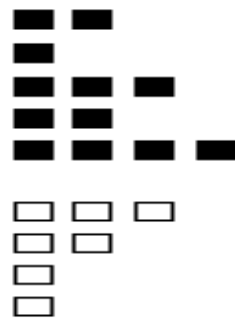
Simultaneous Multithreading



Hardware MultiThreading



Coarse MT



Fine MT



SMT



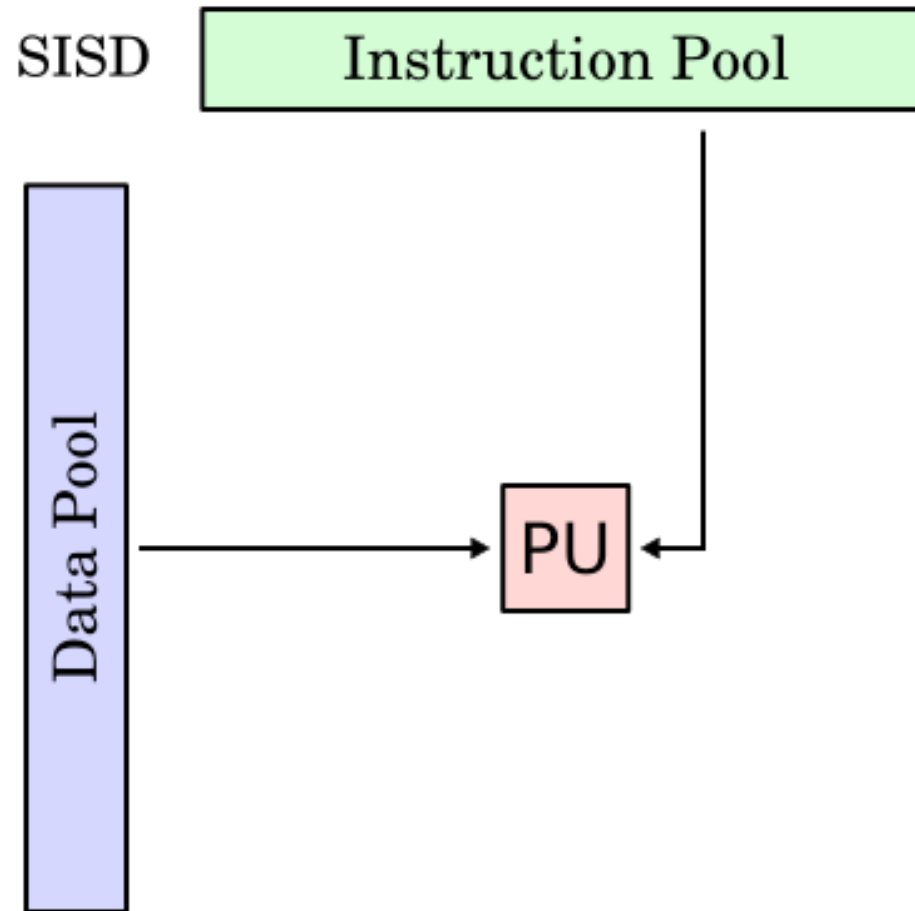
7.6 SISD, MIMD, SIMD, SPMD, and Vector

- SISD and MIMD are based on the number of instruction streams and the number of data streams.
- SIMD computers on the other hand operate on vectors of data streams (e.g., 64 ALUs to form 64 sums within a single clock cycle).
- An important advantage of this method is to reduce size of program.
- SIMD works when there is a lot of data-level parallelism
- Data level parallelism: operating on independent data

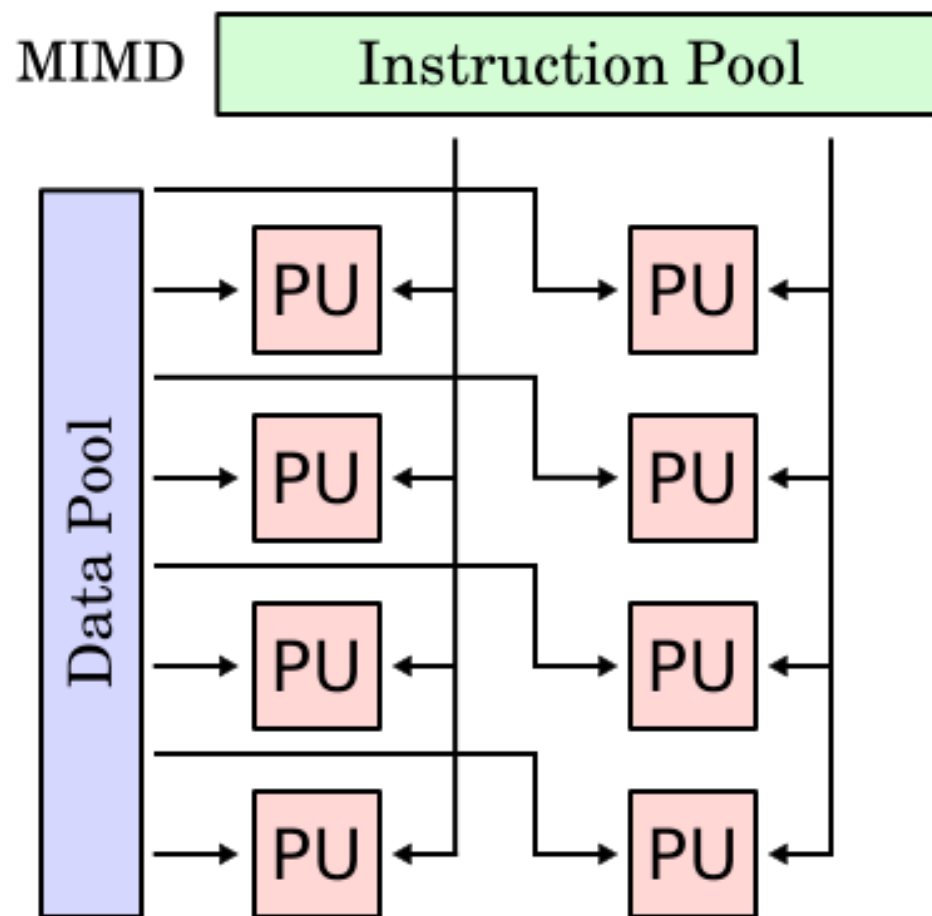
SISD, MIMD, SIMD, SPMD, and Vector

		Data Streams	
		Single	Multiple
Instruction Streams	Single	SISD: Intel Pentium 4	SIMD: SSE instructions of x86
	Multiple	MISD: No examples	MIMD: Intel Xeon e5345

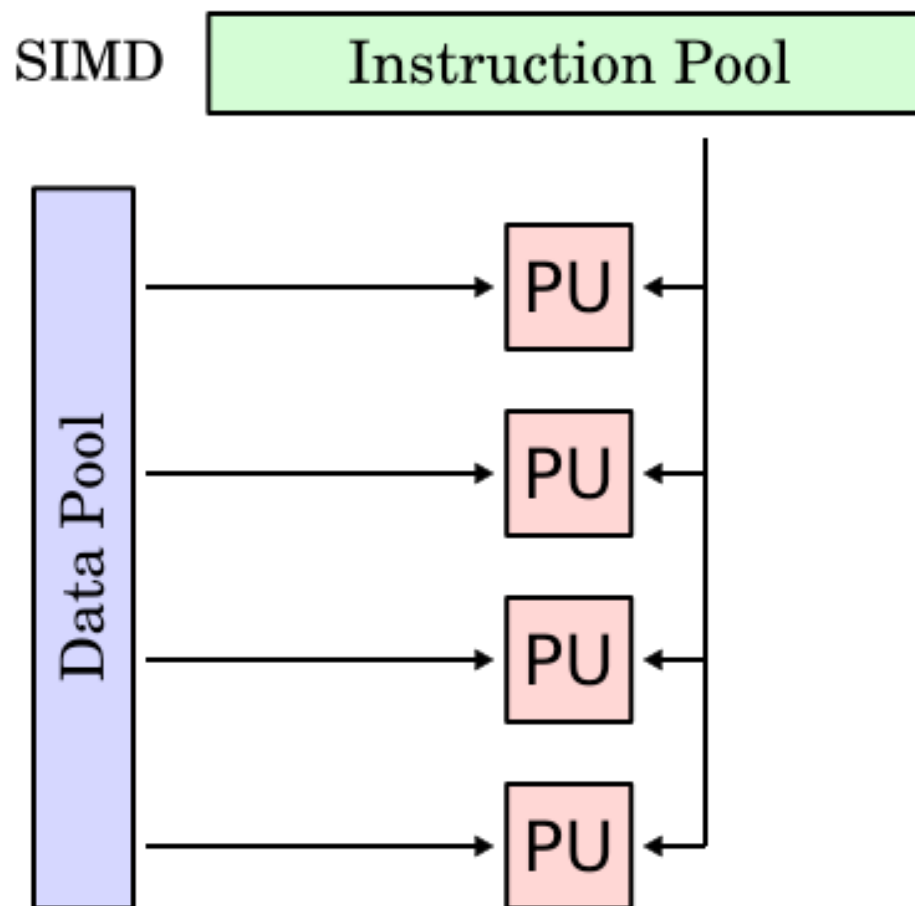
SIMD versus SISD



SIMD versus SISD



SIMD versus SISD



Multimedia Extensions

- The most widely used SIMD method includes MMX and SSE instructions that were added to improve performance of multimedia programs.
- Loads and stores are as wide as the widest ALU, so one can think of the same data transfer instruction as transferring either a single 64-bit data element or two 32-bit data elements, etc...
- SSE2 now supports the simultaneous execution of a pair of 64-bit floating point numbers as a result of the evolution of hardware.

Vector

- Vector processors: heavily pipelined processors that perform efficient operations on entire vectors
- Vectors are used in problems with lots of data-level parallelism.
- Vector architectures pipeline the ALU to get good performance at lower cost.
- Vector registers: specialized registers that can hold vector elements at one time
- Vector architecture might have 32 vector registers, each with 64 64-bit elements.

Vector Processors

- Highly pipelined function units
- Stream data from/to vector registers to units
- Data collected from memory into register. The results are stored from registers to memory
- Example: Vector extension to MIPS: 32×64 -element registers (64-bit elements)
- Vector instructions
 - `lv, sv`: load/store vector
 - `addv.d`: add vectors of double
 - `addvs.d`: add scalar to each element of vector of double
- Significantly reduces instruction-fetch bandwidth

Comparing vector to Conventional Code

- Lets investigate the effects of vector instructions and registers on MIPS code.
- The most dramatic difference between conventional MIPS code and vector MIPS code is that the vector processor greatly reduces the instruction bandwidth.
- This is a result of vector operations working on 64 elements and no overhead instructions are present in vector code. This reduction in executed instructions saves power.

Example: DAXPY ($Y = a * X + Y$)

- Conventional MIPS code

```
      l.d    $f0,a($sp)      ;load scalar a
      addiu  r4,$s0,#512     ;upper bound of what to load
loop: l.d    $f2,0($s0)      ;load x(i)
      mul.d  $f2,$f2,$f0     ;a * x(i)
      l.d    $f4,0($s1)      ;load y(i)
      add.d  $f4,$f4,$f2     ;a * x(i) + y(i)
      s.d    $f4,0($s1)      ;store into y(i)
      addiu  $s0,$s0,#8      ;increment index to x
      addiu  $s1,$s1,#8      ;increment index to y
      subu   $t0,r4,$s0      ;compute bound
      bne   $t0,$zero,loop   ;check if done
```

- Vector MIPS code

```
      l.d    $f0,a($sp)      ;load scalar a
      lv     $v1,0($s0)      ;load vector x
      mulvs.d $v2,$v1,$f0    ;vector-scalar multiply
      lv     $v3,0($s1)      ;load vector y
      addv.d $v4,$v2,$v3     ;add y to product
      sv     $v4,0($s1)      ;store the result
```

Vector vs. Scalar

- Vector instructions have important differences compared to conventional code:
- A single vector instruction accomplishes a great deal-equivalent to executing a loop.
- Hardware does not have to check for data hazards within a vector instruction
- Vector instructions have a known access pattern. Thus, the cost of latency to main memory is seen only once for the entire vector.
- Control hazards that arise from a loop branch is non-existent.

Vector vs. Scalar

- Vector architectures have an advantage in power and energy savings.
- Hardware need only check for data hazards between two vector instructions once per vector operand.
- Vector operations can be made faster than a sequence of scalar operations on the same number of data items.

Vector vs. Multimedia

- Vector – dozens of operations/instruction
- Multimedia – a couple of operations/instruction
- Flexibility – number of operands in vector operation is contained in a register
- Operands in a multimedia operation are embedded in the op code (more op cod)
- More efficient memory access in vector (can load entire block)

Vector vs. Multimedia Extensions

- A vector instruction specifies multiple operations. However, multimedia extensions typically specifies dozens of operations.
- A new large set of op-codes is added each time the “vector” length changes in the multimedia extension architecture.
- Vectors are a very efficient way to execute data parallel processing programs
- Vectors are better suited to compiler technology than multimedia extensions.

7.7

Introduction to Graphics Processing Units

- A major driving force for improving graphics processing was the computer game industry.
- Moore's law enabled video graphics controller chips to add functions to accelerate 2D and 3D graphics.
- As graphics processors increased in power, they earned the name *Graphics Processing Unit*
- There are key characteristics that differ GPU's from CPU's.

Graphics Processing Units

- GPU's do not need to be able to perform all the tasks of a CPU. The CPU-GPU combination is one example of heterogeneous multiprocessing.
- Each pixel fragment can be rendered independently.
- GPUs represent each vertex component as a 32-bit floating point number. Each of the four pixel components now represent a single precision floating-point number between 0.0 and 1.0
- A working set can be hundreds of megabytes and as such, there is a great deal of data-level parallelism in such tasks.

Different Styles of Architecture

- GPUs do not rely on multilevel caches to overcome the long latency to memory.
- GPUs rely on parallelism to obtain high performance.
- GPU is oriented towards bandwidth rather than latency.
- Given the reliance on many threads to have good bandwidth, CPUs have many parallel processors and thus all GPUs are highly multithreaded.
- GPUs now focus on having more scalar instructions to improve programmability and efficiency.

Introduction to NVIDIA GPU Architecture

- NVIDIA uses a GPU architecture called Tesla.
- The latest version of NVIDIA multiprocessors is called a GeForce 8800 GTX, which has 16 multiprocessors and a clock rate of 1.35 GHz.
- The peak single precision multiply-add performance of the 8800 GTX chip is:

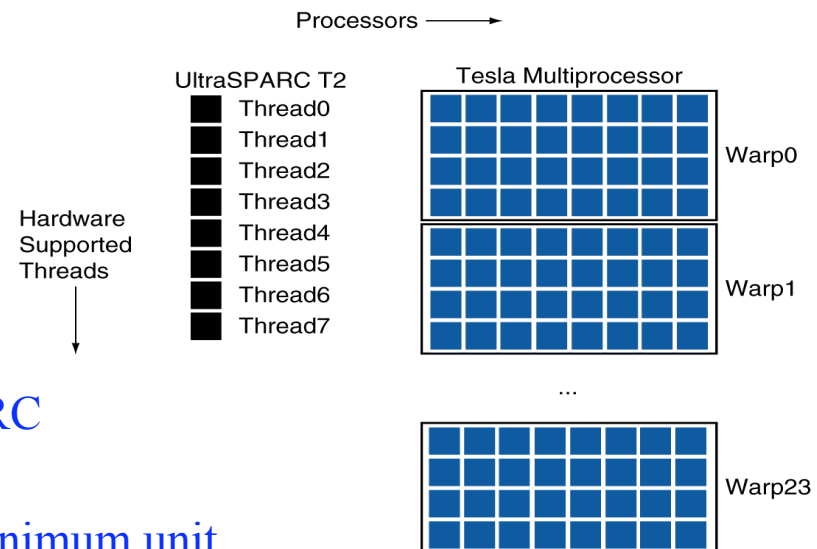
$$\begin{aligned} & 16\text{MPs} \times 8\text{SPs/MP} \times (2\text{FLOPs/instr})/\text{SP} \times 1 \\ & \text{instr/clock} \times 1.35 \times 10^9 \text{ clocks/seconds} \\ & = 345.6 \text{ GFLOPs/second} \end{aligned}$$

NVIDIA Introduction - Continued

- To hide memory latency, each streaming processor has hardware-supported threads.
- Each group of 32 threads is called a *wrap*
- A wrap is a unit of scheduling; up to 32 threads in a wrap execute in parallel in SIMD fashion.
- When threads of a wrap take diverging paths, the wrap sequentially executes both code paths with some inactive threads, which makes more active threads slower.
- For best performance, all 32 threads of a wrap need to execute together in parallel.

Sun UltraSPARC Vs. Tesla multiprocessor

- Tesla multiprocessor and Sun UltraSPARC are hardware multithreaded by scheduling threads over time, shown on the vertical axes of figure below:
- The Tesla multiprocessor uses fine-grained hardware multithreading to schedule 24 wraps over time. UltraSPARC schedules eight hardware-supported threads over time, one thread per cycle, shown vertically.
- The major difference is that UltraSPARC has one processor that can switch threads every clock cycle, while the minimum unit of switching wraps in the Tesla microprocessor in two cycles across eight streaming cores.



Putting GPUs into Perspective

- GPUs don't fit nicely into SIMD/MIMD model
- Conditional execution in a thread allows an illusion of MIM however, it results in performance degradation.
- Need to write general purpose code with care

Static: Discovered
at Compile Time

Dynamic: Discovered at
Runtime