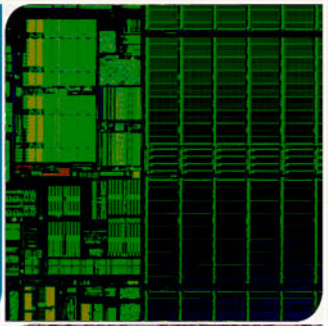
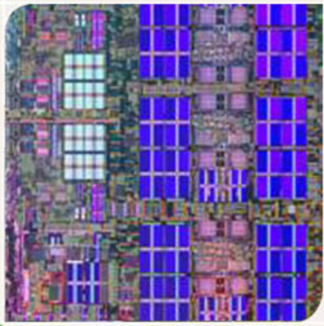
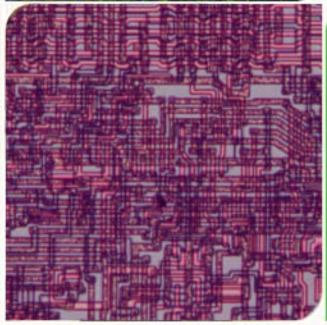
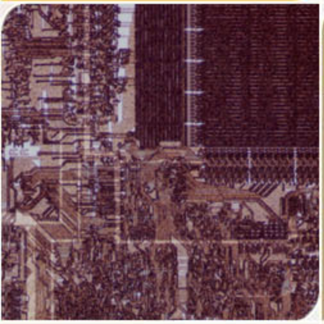





CS4290/CS6290

Fall 2011

Prof. Hyesoon Kim



Georgia Tech College of Computing

Thanks to Prof. Loh & Prof. Prvulovic

Multiprocessing

- Flynn's Taxonomy of Parallel Machines
  - How many Instruction streams?
  - How many Data streams?
- SISD: Single I Stream, Single D Stream
  - A uniprocessor
- SIMD: Single I, Multiple D Streams
  - Each "processor" works on its own data
  - But all execute the same instrs in lockstep
  - E.g. a vector processor or MMX, CUDA

Flynn's Taxonomy

- MISD: Multiple I, Single D Stream
  - Not used much
  - Stream processors are closest to MISD
- MIMD: Multiple I, Multiple D Streams
  - Each processor executes its own instructions and operates on its own data
  - This is your typical off-the-shelf multiprocessor (made using a bunch of "normal" processors)
  - Includes multi-core processors

Flynn's Classical Taxonomy

SISD Single Instruction, Singe Data	SIMD Single Instruction, Multiple Data
MISD Multiple Instruction, Single Data	MIMD Multiple Instruction, Multiple Data

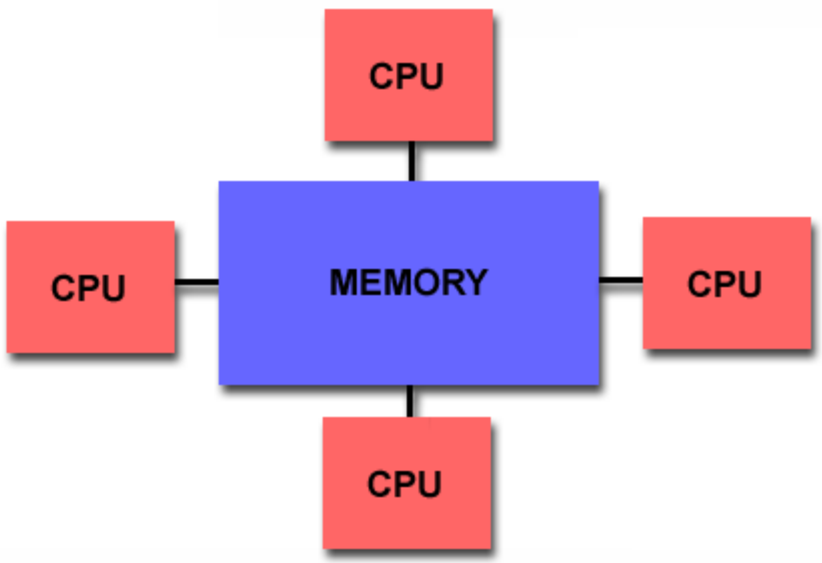
Multiprocessors

- Why do we need multiprocessors?
  - Uniprocessor speed keeps improving
  - But there are things that need even more speed
    - Wait for a few years for Moore's law to catch up?
    - Or use multiple processors and do it now?
- Multiprocessor software problem
  - Most code is sequential (for uniprocessors)
    - MUCH easier to write and debug
  - Correct parallel code very, very difficult to write
    - ***Efficient*** and correct is even harder
    - Debugging even more difficult (Heisenbugs)

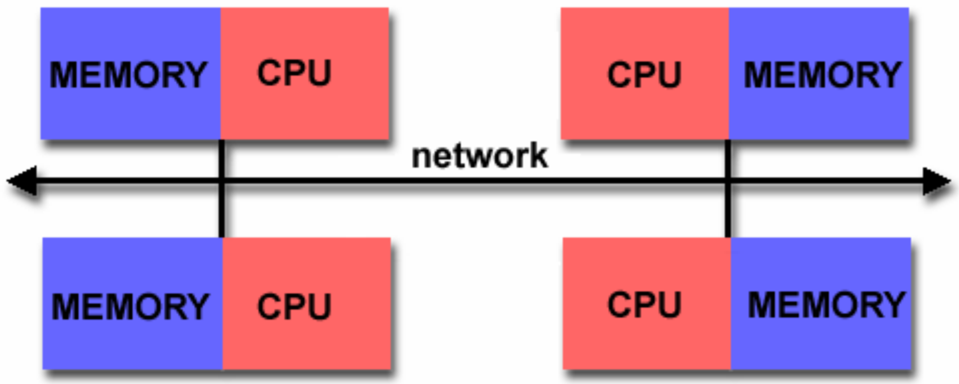
ILP limits reached?

MIMD Multiprocessors

Centralized Shared Memory



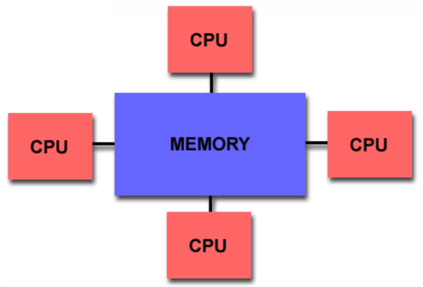
Distributed Memory



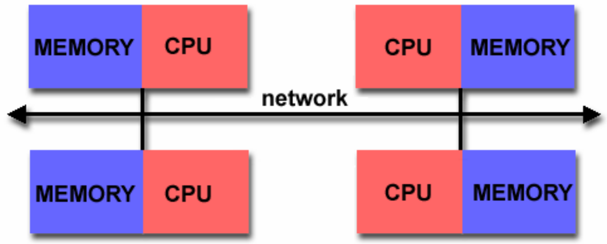
Centralized-Memory Machines

- Also “Symmetric Multiprocessors” (SMP)
- “Uniform Memory Access” (UMA)
  - All memory locations have similar latencies
  - Data sharing through memory reads/writes
  - P1 can write data to a physical address A,
    P2 can then read physical address A to get that data

- Problem: Memory Contention
  - All processor share the one memory
  - Memory bandwidth becomes bottleneck
  - Used only for smaller machines
    - Most often 2,4, or 8 processors



Distributed-Memory Machines



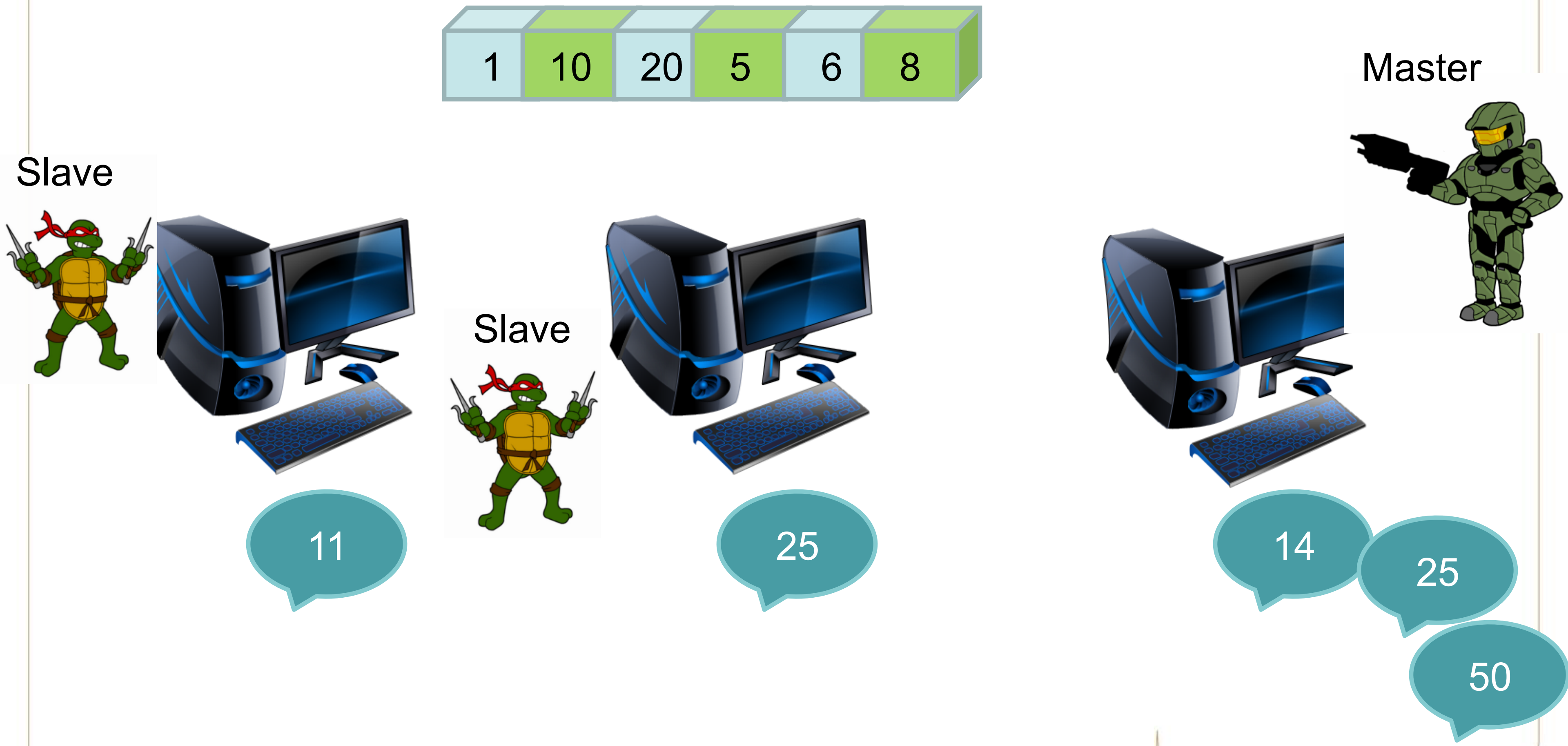
- Two kinds
  - Distributed Shared-Memory (DSM)
    - All processors can address all memory locations
    - Data sharing like in SMP
    - Also called **NUMA (non-uniform memory access)**
    - Latencies of different memory locations can differ (local access faster than remote access)
  - Message-Passing
    - A processor can directly address only local memory
    - To communicate with other processors, must explicitly send/receive messages
    - Also called multicomputers or clusters
- Most accesses local, so less memory contention (can scale to well over 1000 processors)

Georgia Tech College of Computing

Message-Passing Machines

- A cluster of computers
  - Each with its own processor and memory
  - An interconnect to pass messages between them
  - Producer-Consumer Scenario:
    - P1 produces data D, uses a SEND to send it to P2
    - The network routes the message to P2
    - P2 then calls a RECEIVE to get the message
  - Two types of send primitives
    - Synchronous: P1 stops until P2 confirms receipt of message
    - Asynchronous: P1 sends its message and continues
  - Standard libraries for message passing:
    Most common is MPI – Message Passing Interface

Message Passing Example



Message Passing: A Program

- Calculating the sum of array elements

```
#define ASIZE 1024
#define NUMPROC 4
double myArray[ASIZE/NUMPROC];
double mySum=0;
for(int i=0;i<ASIZE/NUMPROC;i++)
  mySum+=myArray[i];
if(myPID=0){
  for(int p=1;p<NUMPROC;p++){
    int pSum;
     recv(p,pSum);
    mySum+=pSum;
  }
  printf("Sum: %lf\n",mySum);
}else
  send(0,mySum);
```

Must manually split the array

"Master" processor adds up partial sums and prints the result

"Slave" processors send their partial results to master

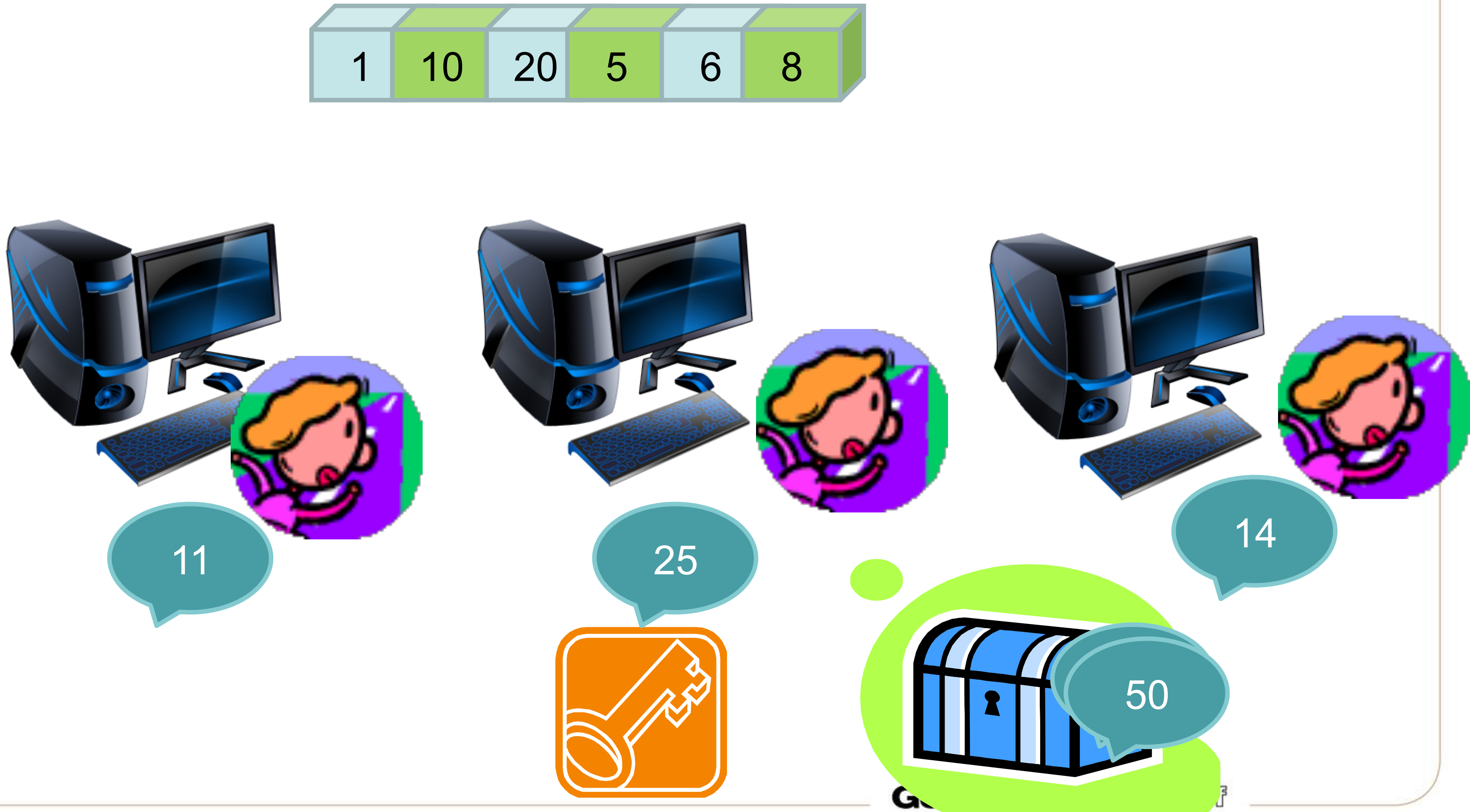
Message Passing Pros and Cons

- Pros
  - Simpler and cheaper hardware
  - Explicit communication makes programmers aware of costly (communication) operations
- Cons
  - Explicit communication is painful to program
  - Requires manual optimization
    - If you want a variable to be local and accessible via LD/ST, you must declare it as such
    - If other processes need to read or write this variable, you must explicitly code the needed sends and receives to do this

Communication Performance

- Metrics for Communication Performance
  - Communication Bandwidth
  - Communication Latency
    - Sender overhead + transfer time + receiver overhead
  - Communication latency hiding
- Characterizing Applications
  - Communication to Computation Ratio
    - Work done vs. bytes sent over network
    - Example: 146 bytes per 1000 instructions

Shared Memory Example



Shared Memory: A Program

- Calculating the sum of array elements

```
#define ASIZE 1024
#define NUMPROC 4
shared double array[ASIZE];
shared double allSum=0;
shared mutex sumLock;
double mySum=0;
for(int i=myPID*ASIZE/NUMPROC;i<(myPID+1)*ASIZE/NUMPROC;i++)
  mySum+=array[i];
lock(sumLock);
allSum+=mySum;
unlock(sumLock);
if(myPID=0)
  printf("Sum: %lf\n",allSum);
```

Array is shared

Each processor sums up "its" part of the array

Each processor adds its partial sums to the final result

"Master" processor prints the result

Georgia Tech College of Computing

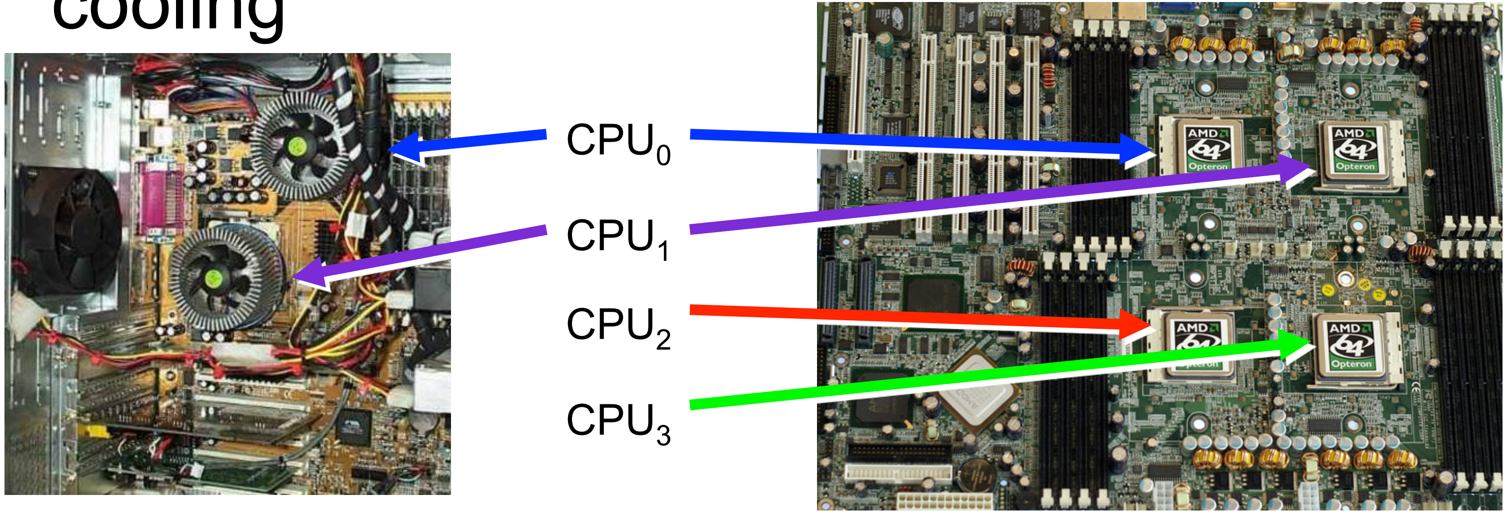
Shared Memory Pros and Cons

- Pros
  - Communication happens automatically
  - More natural way of programming
    - Easier to write correct programs and gradually optimize them
  - No need to manually distribute data
    (but can help if you do)
- Cons
  - Needs more hardware support
  - Easy to write correct, but inefficient programs
    (remote accesses look the same as local ones)

MULTI PROCESSORS

Implementing MP Machines

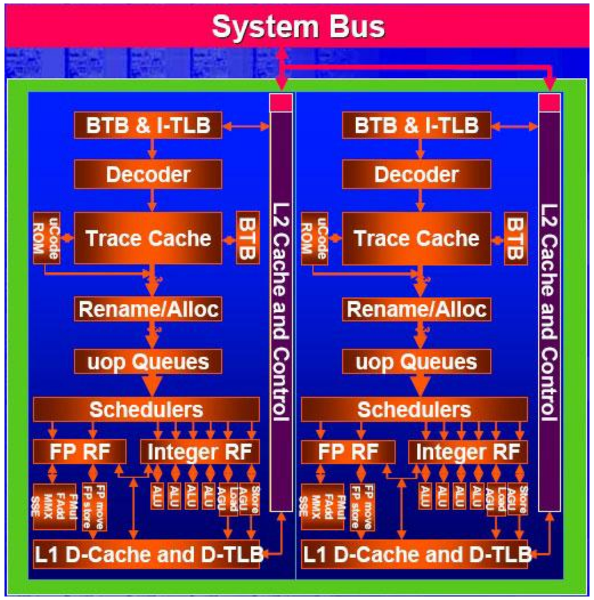
- One approach: add sockets to your MOBO
  - minimal changes to existing CPUs
  - power delivery, heat removal and I/O not too bad since each chip has own set of pins and cooling



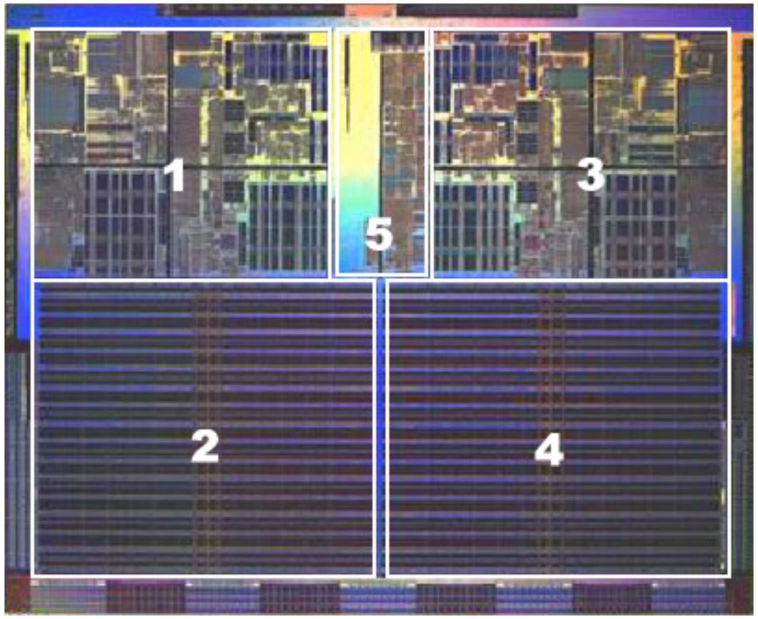
Pictures found from google images

Chip-Multiprocessing

- Simple SMP on the same chip



Intel "Smithfield" Block Diagram

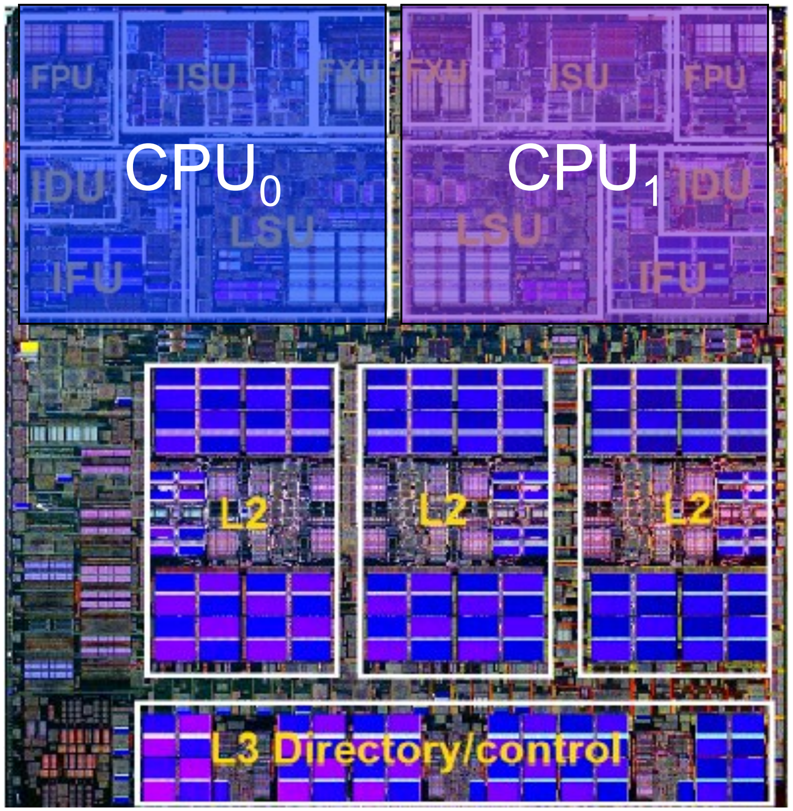


AMD Dual-Core Athlon FX

Pictures found from google images

Shared Caches

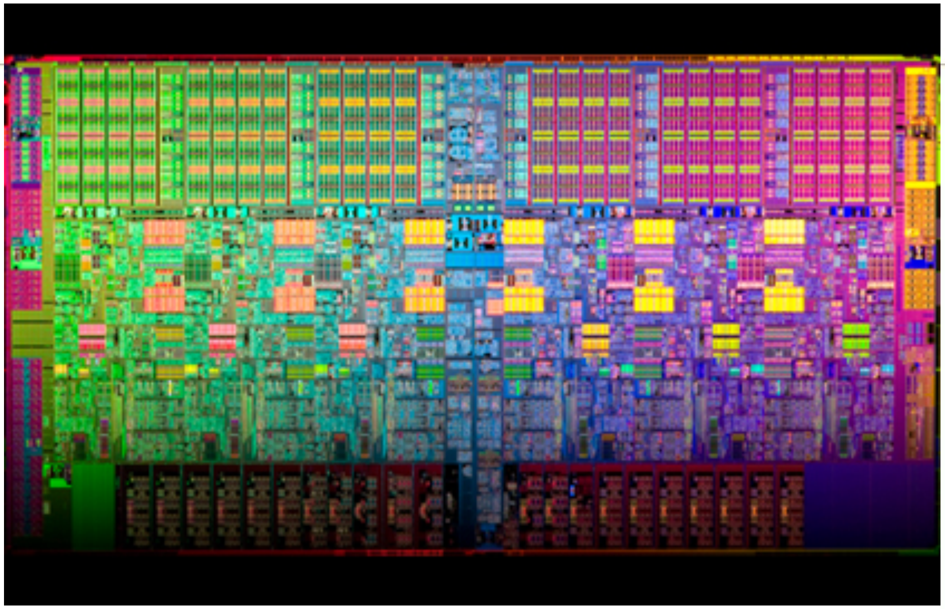
- Resources can be shared between CPUs
  – ex. IBM Power 5

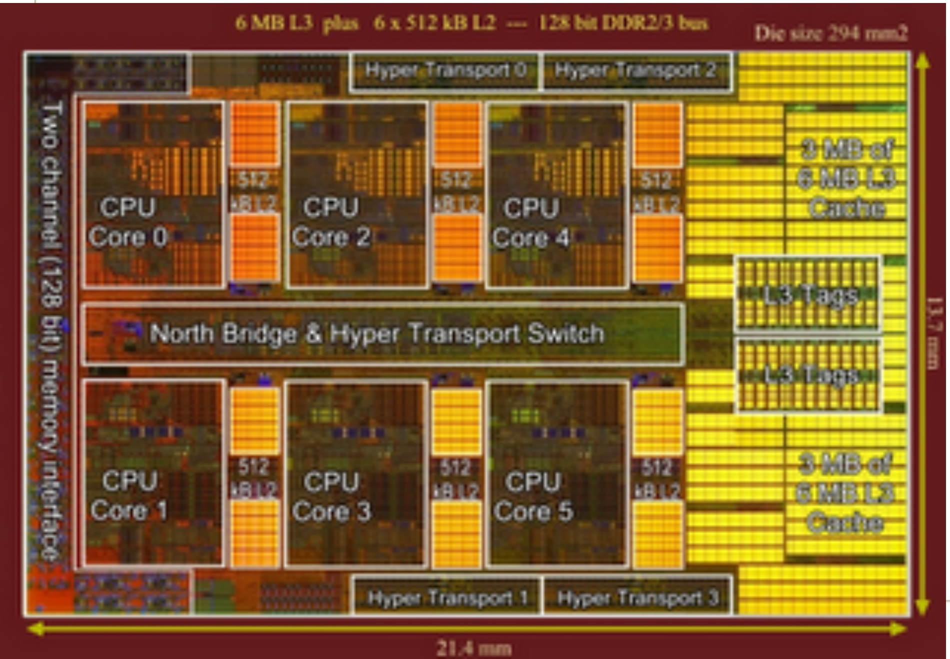


L2 cache shared between both CPUs (no need to keep two copies coherent)

L3 cache is also shared (only tags are on-chip; data are off-chip)

6-core world!





Georgia Tech | College of Computing

Benefits?

- Cheaper than mobo-based SMP
  - all/most interface logic integrated on to main chip (fewer total chips, single CPU socket, single interface to main memory)
  - less power than mobo-based SMP as well (communication on-die is more power-efficient than chip-to-chip communication)
- Performance
  - on-chip communication is faster
- Efficiency
  - potentially better use of hardware resources than trying to make wider/more OOO single-threaded CPU

Performance vs. Power

- 2x CPUs not necessarily equal to 2x performance
- 2x CPUs → ½ power for each
  - maybe a little better than ½ if resources can be shared
- Back-of-the-Envelope calculation:
  - 3.8 GHz CPU at 100W
  - Dual-core: 50W per CPU
  - $P \propto V^3$: $V_{orig}{}^3 / V_{CMP}{}^3 = 100W/50W \rightarrow V_{CMP} = 0.8\ V_{orig}$
  - $f \propto V$: $f_{CMP} = 3.0GHz$

Benefit of SMP: Full power budget per socket!

Georgia Tech College of Computing

Performance vs. Power (2)

- So what's better?
  - One 3.8 GHz CPU?
  - Or a dual-core running at 3.0 GHz?
- Depends on workloads
  - If you have one program to run, the 3.8GHz CPU will run it in 79% of the time
  - If you have two programs to run, then:
    - 3.8GHz CPU: 79% for one, or 158% for both
    - Dual 3.0GHz CPU: 100% for both in parallel

Question

- Dual Core: total power 200W frequency: 2GHz
- With the same power budget if we have 4 cores, what should be the frequency of each core?

Multithreaded Processors

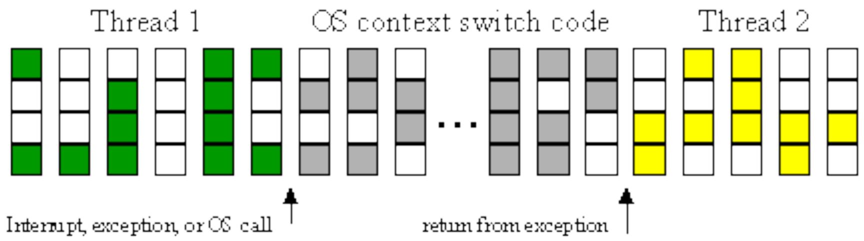
- Single thread in superscalar execution: dependences cause most of stalls
- Idea: when one thread stalled, other can go
- Different granularities of multithreading
  - Coarse MT: can change thread every few cycles
  - Fine MT: can change thread every cycle
  - Simultaneous Multithreading (SMT)
    - Instrs from different threads even in the same cycle
    - AKA **Hyperthreading**

Simultaneous Multi-Threading

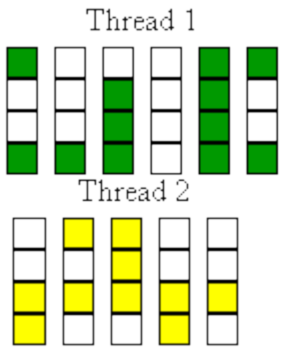
- Uni-Processor: 4-6 wide, lucky if you get 1-2 IPC
  - poor utilization
- SMP: 2-4 CPUs, but need independent tasks
  - else poor utilization as well
- SMT: Idea is to use a single large uni-processor as a multi-processor

SMT (2)

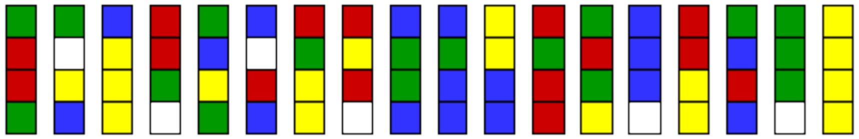
Regular CPU



CMP



SMT (4 threads)



2x HW Cost

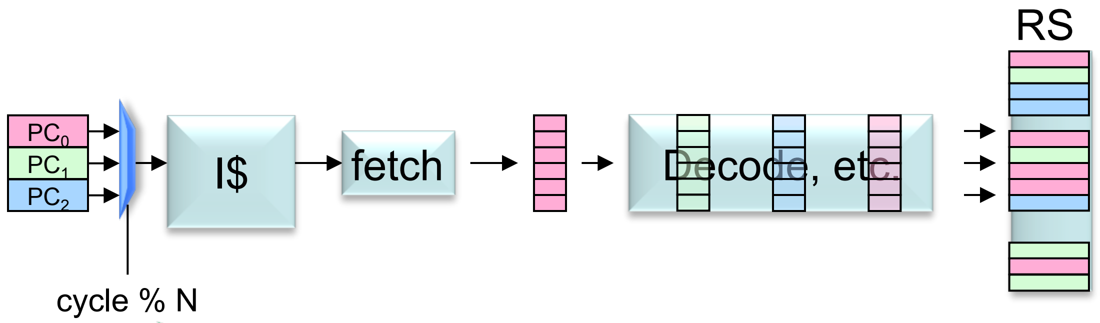
Approx 1x HW Cost

Overview of SMT Hardware Changes

- For an N-way (N threads) SMT, we need:
  - Ability to fetch from N threads
  - N sets of architectural registers (including PCs)
  - N rename tables (RATs)
  - N virtual memory spaces
  - Front-end: branch predictor?: no, RAS? :yes
- But we don't need to replicate the entire OOO execution engine (schedulers, execution units, bypass networks, ROBs, etc.)

SMT Fetch

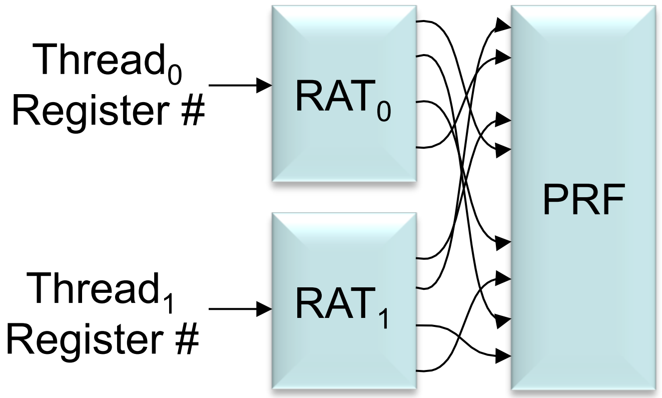
- Multiplex the Fetch Logic



Can do simple round-robin between active threads, or favor some over the others based on how much each is stalling relative to the others

SMT Rename

- Thread #1's R12 != Thread #2's R12
  - separate name spaces
  - need to disambiguate



SMT Issue, Exec, Bypass, …

- No change needed

Thread 0:

Add R1 = R2 + R3
Sub R4 = R1 – R5
Xor R3 = R1 ^ R4
Load R2 = 0[R3]

Thread 1:

Add R1 = R2 + R3
Sub R4 = R1 – R5
Xor R3 = R1 ^ R4
Load R2 = 0[R3]

After Renaming

Thread 0:

Add T12 = T20 + T8
Sub T19 = T12 – T16
Xor T14 = T12 ^ T19
Load T23 = 0[T14]

Thread 1:

Add T17 = T29 + T3
Sub T5 = T17 – T2
Xor T31 = T17 ^ T5
Load T25 = 0[T31]

Shared RS Entries

Sub T5 = T17 – T2
Add T12 = T20 + T8
Load T25 = 0[T31]
Xor T14 = T12 ^ T19
Load T23 = 0[T14]
Sub T19 = T12 – T16
Xor T31 = T17 ^ T5
Add T17 = T29 + T3

Georgia Tech College of Computing

SMT Cache

- Each process has own virtual address space
  - TLB must be thread-aware
    - translate (thread-id,virtual page) → physical page
  - Virtual portion of caches must also be thread-aware
    - VIVT cache must now be (virtual addr, thread-id)-indexed, (virtual addr, thread-id)-tagged
    - Similar for VIPT cache
    - No changes needed if using PIPT cache (like L2)

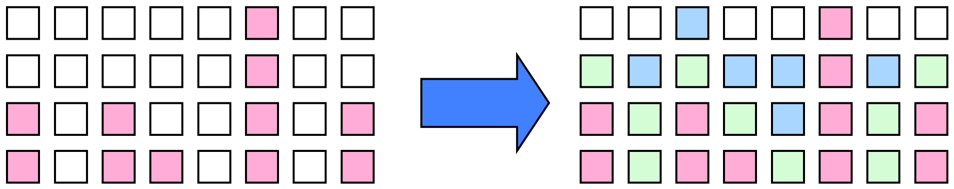
SMT Commit

- Register File Management
  - ARF/PRF organization
    - need one ARF per thread

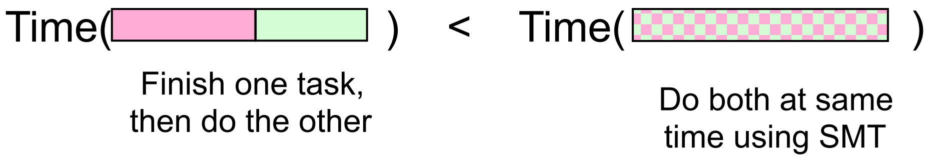
- Need to maintain interrupts, exceptions, faults on a per-thread basis
  - like OOO needs to appear to outside world that it is in-order, SMT needs to appear as if it is actually N CPUs

SMT Performance

- When it works, it fills idle "issue slots" with work from other threads; throughput improves

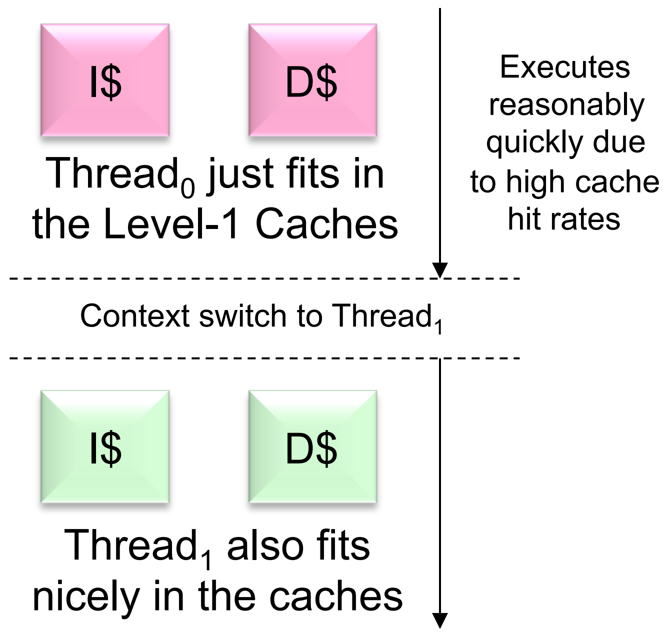


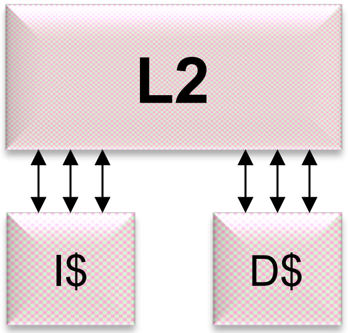
- But sometimes it can cause performance degradation!


Time( ) < Time( )
Finish one task, then do the other
Do both at same time using SMT

How?

- Cache thrashing


I$
D$
Thread$_0$ just fits in the Level-1 Caches
Executes reasonably quickly due to high cache hit rates
Context switch to Thread$_1$
I$
D$
Thread$_1$ also fits nicely in the caches

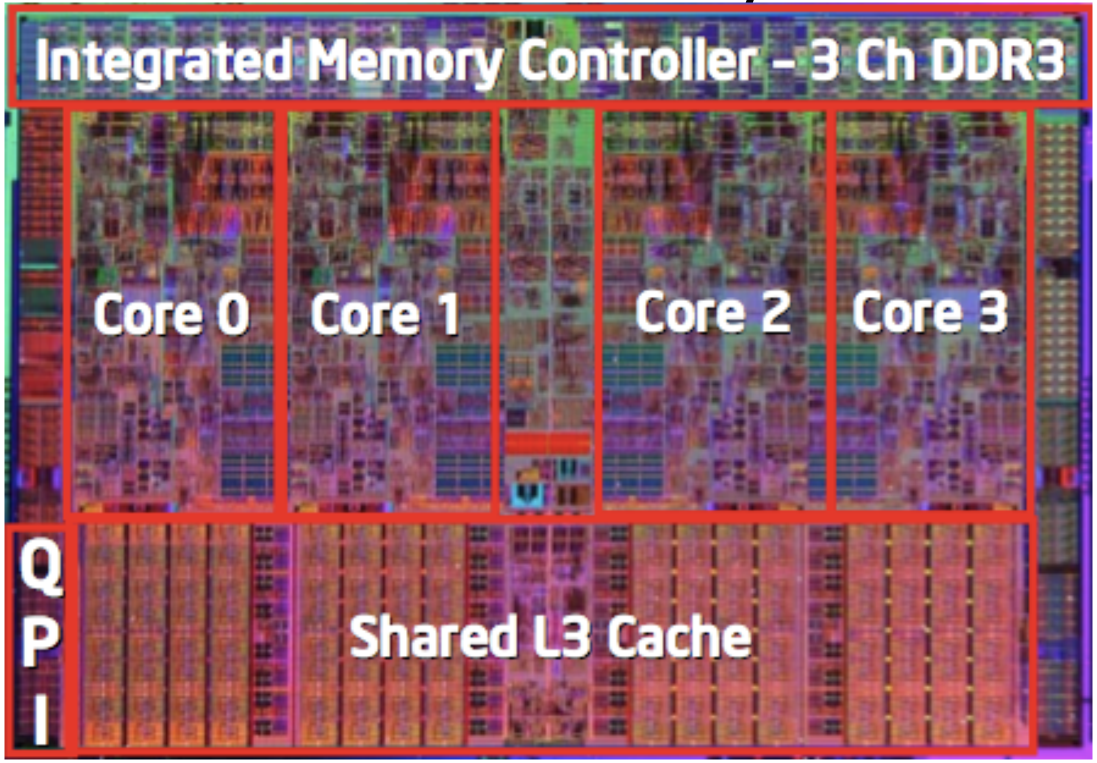


Caches were just big enough to hold one thread's data, but not two thread's worth

Now both threads have significantly higher cache miss rates

SMT+CMP

- Intel’s Nehalemn
- Each core is 2-way SMT



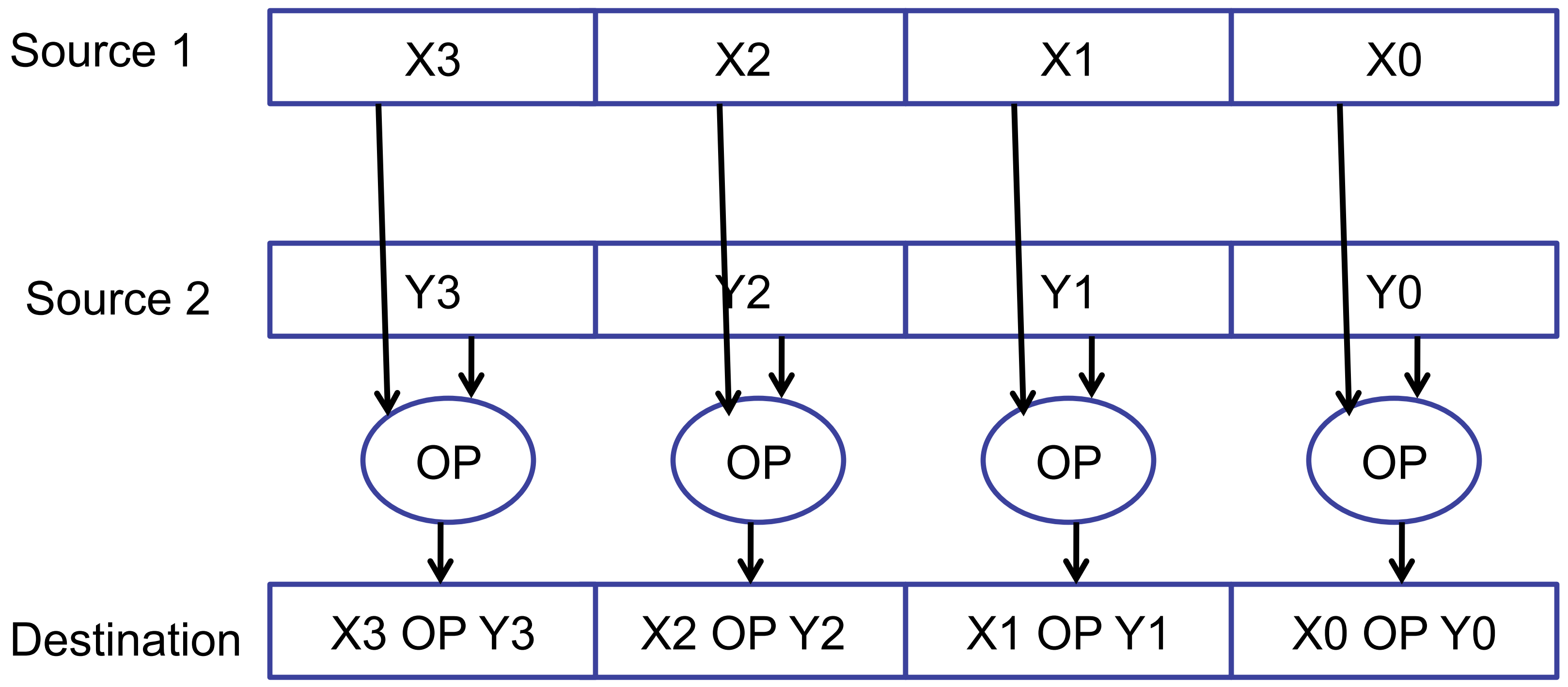
http://www.intel.com/technology/product/demos/turboboost/demo.htm?iid=tech_tb+demo

SIMD

SIMD Model

- Texas C62xx, IA32 (SSE), AMD K6, CUDA, Xbox..
- Early SIMD machines: e.g.) CM-2 (large distributed system)
  - Lack of vector register files and efficient transposition support in the memory system.
  - Lack of irregular indexed memory accesses
- Modern SIMD machines:
  - SIMD engine is in the same die

SIMD Execution Model



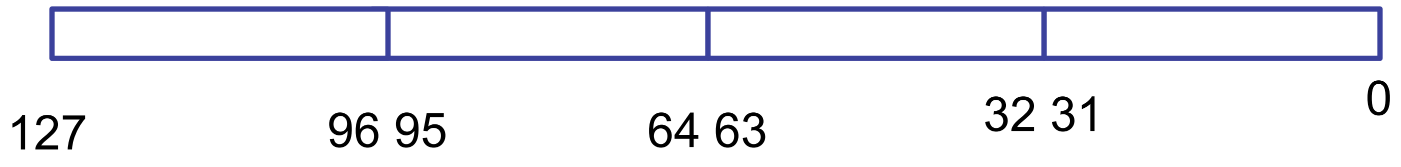
```
for (ii = 0; ii < 4; ii++)
x[ii] = y[ii]+z[ii];
```

⇒

SIMD_ADD(X, Y, Z)

Intel' SSE (Streaming SIMD Extensions)

- New data type
  - 128-bit packed single-precision floating-point data type



- Packed/Scalar singe-precision floating-point instruction
- 64-bit SIMD integer instruction
- State management instructions
- Cacheability control, prefetch, and memory ordering instructions

SSE2/SSE3/SSE4

- Add new data types
- Add more complex SIMD instructions
- Additional vector registers
- Additional cacheability-control and instruction-ordering instructions.

Loop unrolling

```
for (i = 1; i < 12; i++) x[i] = j[i]+1;

 for (i = 1; i < 12; i=i+4)
{
     x[i] = j[i]+1;
      x[i+1] = j[i+1]+1;
      x[i+2] = j[i+2]+1;
       x[i+3] = j[i+3]+1;
}
```

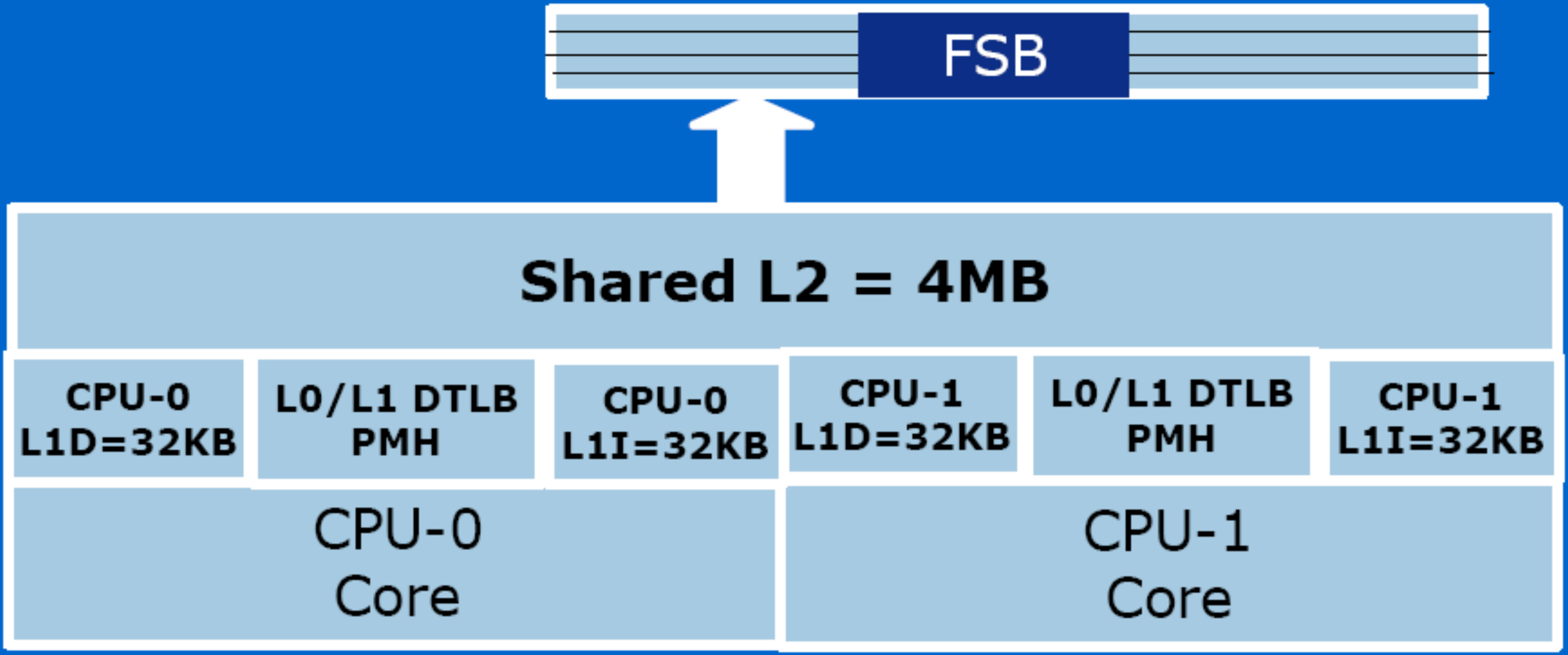
SSE ADD

Vectorization (SIMDzation)

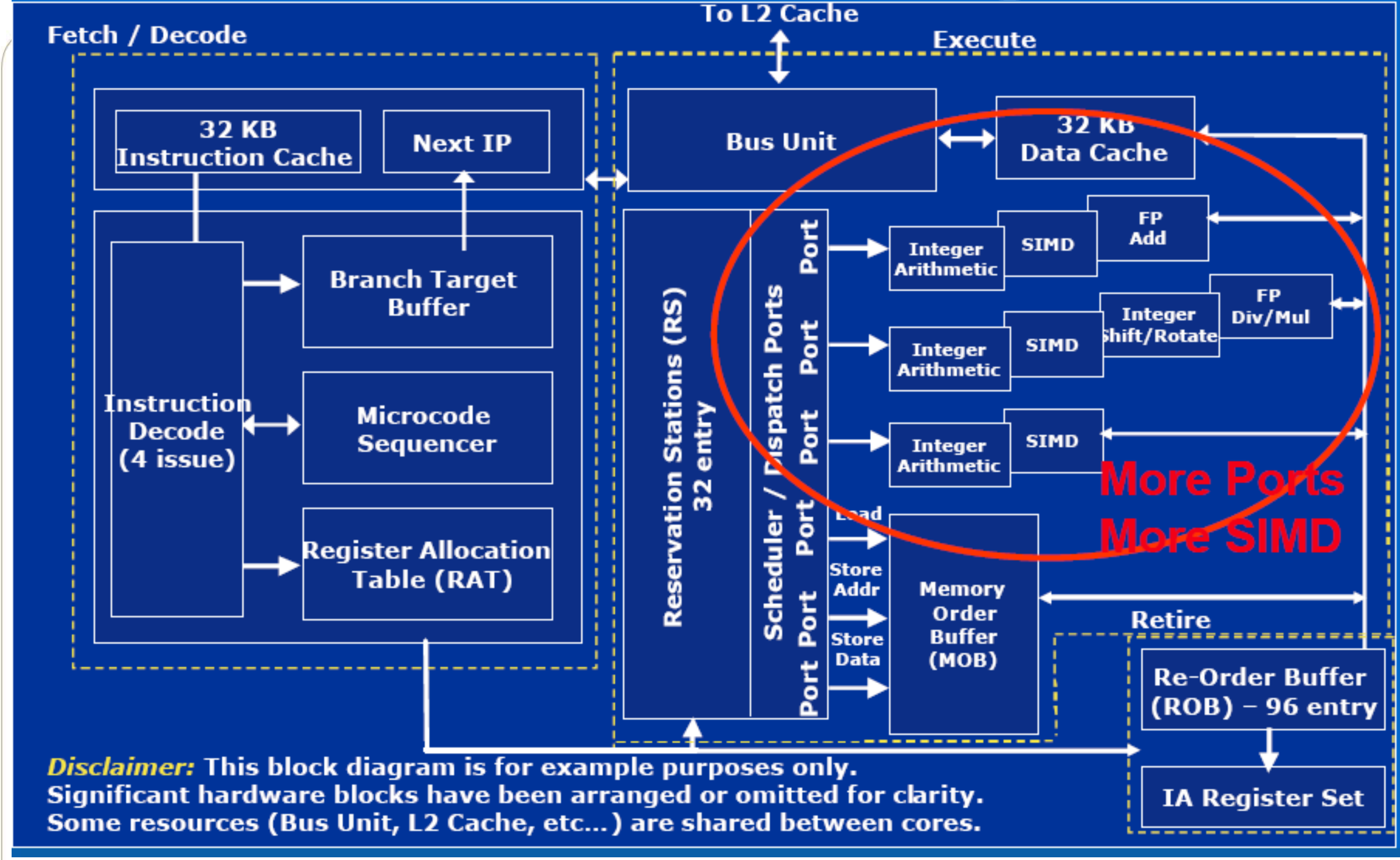
- Which code can be vectorized?

```
Case1: for (i = 0; i < 1024; i++)
    C[i] = A[i]*B[i];
Case 2: for (i=0;i<1024;i++)
        a[i] = a[i+k]-1;   k=3
Case 3: for (i=0;i<1024;i++)
        a[i] = a[i-k]-1;   k=3
```

Next Generation Micro Architecture Intel® Core™2 Duo Processor



Architecture Block Diagram



Disclaimer: This block diagram is for example purposes only.
Significant hardware blocks have been arranged or omitted for clarity.
Some resources (Bus Unit, L2 Cache, etc…) are shared between cores.

Georgia Tech | College of Computing