

## An Introduction to the hardware of your PC

---

Know your tools!

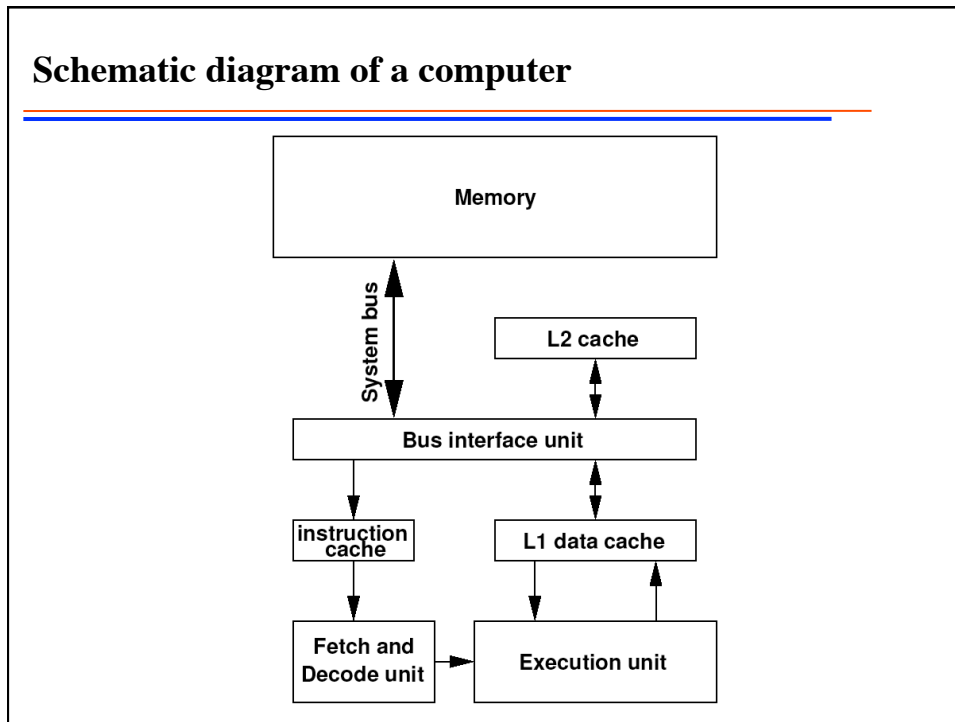
We need to understand what the computer does before we can write fast programs

## Understanding hardware is important

---

- ◆ Steps in executing a program
  - ◆ We write our code in a high-level language
  - ◆ The compiler translates the program to machine language
  - ◆ The computer executes the machine language program
  
- ◆ We want to write a fast program
  - ◆ Need to understand hardware limitations
  - ◆ Need to understand what the compiler does
  
- ◆ This week
  - ◆ Introduction to main hardware components
  - ◆ Understanding the limitations

## Schematic diagram of a computer



## Components of the CPU

- ◆ The main components of the central processing unit (CPU) are:
  - ◆ Memory controller
    - ◆ Manages loading from and storing to memory
  - ◆ Registers
    - ◆ Can store integer or floating point numbers
    - ◆ Values can be set to specified constants
    - ◆ Values can be loaded from or stored into memory
  - ◆ Arithmetic and logical units (ALU)
    - ◆ Performs arithmetic operations and comparisons
    - ◆ Operates on values in the registers (very fast)
    - ◆ On some CPUs they can operate on contents of memory (slow)
  - ◆ Fetch and decode unit
    - ◆ Fetches the next instruction from memory
    - ◆ Interprets the numerical value of the instruction and decides what to do
    - ◆ Dispatches operations to ALU and memory controller to perform the operation
- ◆ Be aware that modern CPUs are more complex (see later)

## Machine code and assembly language

---

- ◆ The CPU performs instructions read from memory
  - ◆ Instructions are given in machine code
  - ◆ These are just numbers which are interpreted as instructions
  - ◆ Ugly and nearly impossible to interpret
- ◆ Assembly language
  - ◆ Is a one-to-one translation from machine code to a readable text form
  - ◆ Is non-portable: differs depending on CPU-type
- ◆ Typical instructions
  - ◆ Load values into registers
  - ◆ Load data from memory into register or store registers into memory
  - ◆ Perform arithmetic and logical instructions on registers
  - ◆ Jump (branch) to another instruction

## CISC CPUs

---

- ◆ Complex instruction set
  - ◆ Many high-level instructions (example: sin-cos-instruction)
  - ◆ Take many cycles to execute
    - ◆ High clock rate does not tell everything
- ◆ Examples
  - ◆ Intel IA-32
  - ◆ AMD x86\_64
- ◆ Advantage
  - ◆ High level instructions makes assembly language programming easy
- ◆ Disadvantage
  - ◆ Very complex CPU for high level instructions

## RISC CPUs

---

- ◆ Reduced instruction set
  - ◆ Only low level instructions
    - ◆ E.g. load from memory into register, add values in registers, ...
  - ◆ But very fast execution speed (few cycles per instruction)
  - ◆ Many registers in the CPU
- ◆ Example:
  - ◆ IBM Power and PowerPC
    - ◆ E.g. IBM BlueGene/Q: PowerPC At 1.6GHz
- ◆ Advantages
  - ◆ fast and can be pipelined
  - ◆ Small and use little power
- ◆ Disadvantage
  - ◆ More machine language instructions needed

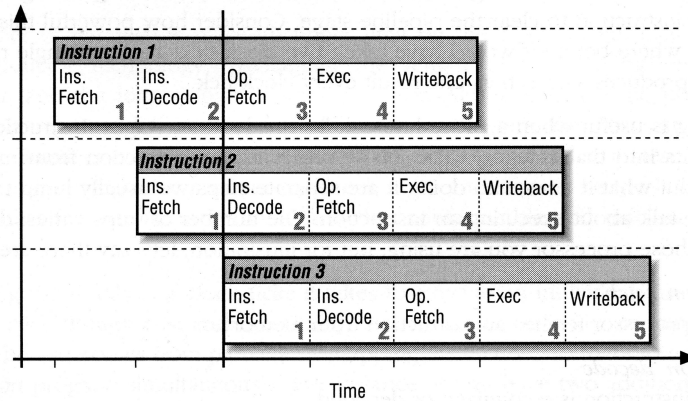
## GPUs

---

- ◆ General-purpose GPUs offer immense floating point performance
  - ◆ Performance gain often >10x in actual application
- ◆ Example: NVIDIA Tesla C870
  - ◆ 128 cores
  - ◆ over 500 GFlop/s (fastest Xeon: 107 GFlop/s)
- ◆ SIMD (Single Instruction Multiple Data) programming style
- ◆ Harder to program
  - ◆ Many cores
  - ◆ Inhomogeneous and small memory
  - ◆ Transfer between CPU and GPU is expensive
- ◆ Programming environments: CUDA, OpenCL

## Pipelining

- ◆ Is used to speed up execution
  - ◆ Second (independent) instruction can be started before first one finishes



## Example of a pipeline

- ◆ Imagine a loop
 

```
for (int i=0; i <102400; ++i)
  a[i]=b[i]+c[i];
```
- ◆ Consecutive iterations are independent and can be executed in parallel after unrolling

```
for (int i=0; i <102400; i+=4){
  a[i]=b[i]+c[i];
  a[i+1]=b[i+1]+c[i+1];
  a[i+2]=b[i+2]+c[i+2];
  a[i+3]=b[i+3]+c[i+3];
}
```

## Let us look at some examples

---

- ◆ Check out week 6 from the repository
- ◆ Example1: simpleadd.C
  - ◆ Add two floating point numbers
- ◆ Example 2: loopadd.C
  - ◆ Add two arrays of floating point numbers

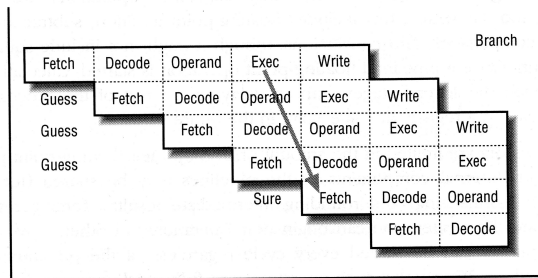
## Looking at the assembly code

---

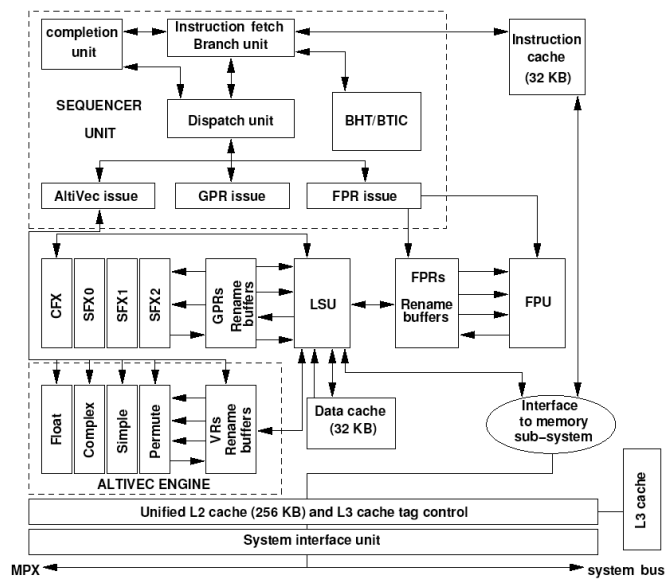
- ◆ Run, one after the other
  - ◆ `g++ -save-temps -c -O3 simpleadd.C`
  - ◆ `g++ -save-temps -c -O3 loopadd.C`
  - ◆ `g++ -save-temps -c -O3 -funroll-loops loopadd.C`
- ◆ Let us take a look at the created intermediate \*.s files
- ◆ simpleadd.s
  - ◆ Can you understand the addition?
  - ◆ Can you see the difference between RISC (Mac) and CISC (Pentium)?
- ◆ loopadd.s
  - ◆ Can you see the loop?
  - ◆ Can you see the unrolling and potential for pipelining?

## Branch prediction

- ◆ At each branch (*if*-statement, ...) the pipelines stall
  - ◆ Have to wait for end of execution before starting one of the branches
- ◆ Solution: branch prediction
  - ◆ Predict (clever compiler, clever hardware) which branch is more likely
    - ◆ E.g. in loop will usually repeat the loop
  - ◆ Start executing more likely branch
    - ◆ If correct prediction: pipeline runs on without any cost
    - ◆ If wrong prediction: abort pipeline and start right branch



## Diagram of a PowerPC G4 CPU



## Summary of CPUs

---

- ◆ Several types of architectures
  - ◆ CISC, RISC, GPUs
- ◆ Differences are disappearing:
  - ◆ All modern CPUs are superscalar.
  - ◆ They have SIMD units (e.g. SSE).
  - ◆ They may internally convert instructions from CISC to RISC.
- ◆ Hybrid systems (CPU+GPU) will very likely be the future of high performance computing
- ◆ We have very fast CPUs, but the rest of the system cannot keep up with the speed

## Moore's law

---

- ◆ "The number of transistors on a chip doubles every 18 months"
  - ◆ More transistors means smaller transistors
  - ◆ Smaller transistors => shorter distances => faster signals
  - ◆ Smaller transistors => fewer charges => faster switching
  - ◆ Thus also the CPU speed increases exponentially
- ◆ Has worked for the past 30 years!
- ◆ How long will it continue?
  - ◆ Current prototype chips at 10 GHz
  - ◆ Insulating layers only 4 atoms thick!
  - ◆ Can we still reduce the size??
  - ◆ Moore's law will probably stop working in the next decade
  - ◆ Software optimization will become more important



## Moore's law

Moore's Law ran smoothly until 2002, when the gap between performance and gate count started to appear.

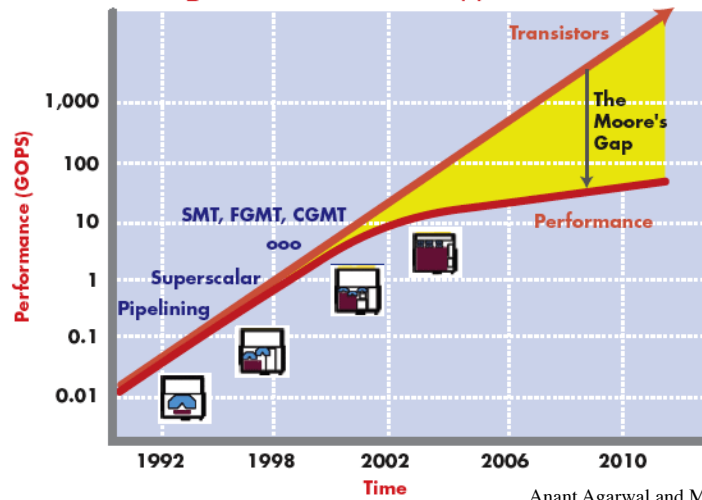
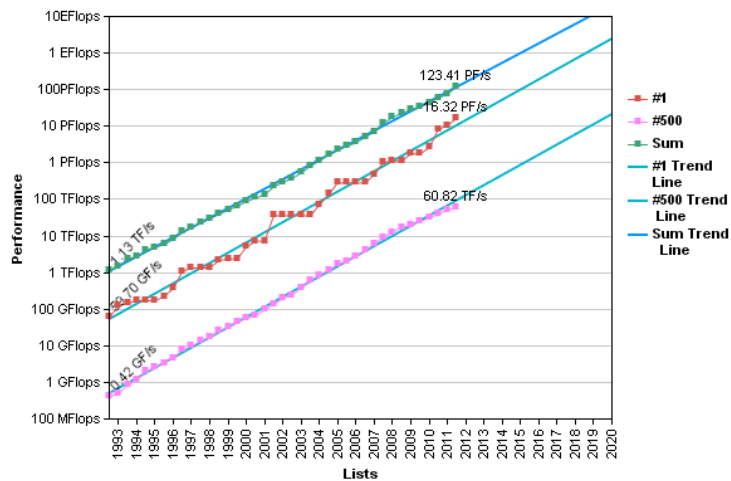


Figure 2

Anant Agarwal and Markus Levy

## Moore's law for supercomputers

Projected Performance Development



## How about the other components of a computer?

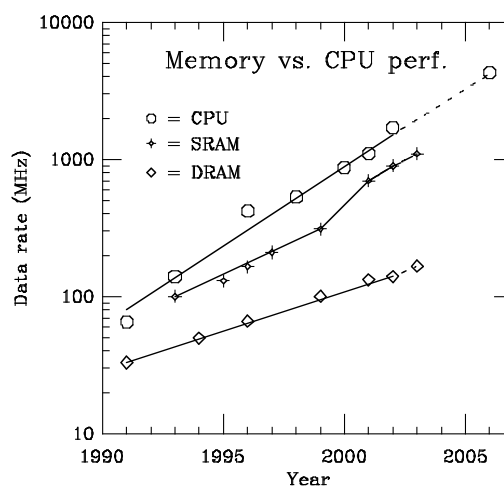
---

- ◆ Transistor density doubles every 18 months
- ◆ PC speed doubles every 2-2.5 years
  - ◆ Are now as fast as supercomputers were a decade ago
- ◆ Supercomputer speed doubles every year
  - ◆ PCs will not catch up with supercomputers
- ◆ But the rest of the system does not catch up
  - ◆ RAM speed increases slower
  - ◆ Disk speed increases even slower

## Memory versus CPU speed

---

- ◆ DRAM has gotten cheap over the past decades but not much faster



## Memory (RAM)

---

- ◆ SRAM (static random access memory)
  - ◆ Very fast access but very expensive
  - ◆ Data stored in state of transistors (flip-flop)
  - ◆ Data stays as long as there is power
  
- ◆ DRAM (dynamic random access memory)
  - ◆ Much cheaper than SRAM but slower
  - ◆ Data stored in tiny capacitor which discharge slowly
  - ◆ Capacitors need to be recharged regularly (hence dynamic)
  
- ◆ SDRAM (synchronous dynamic random access memory)
  - ◆ Variant of DRAM, with a clock synchronized with caches,
  - ◆ allows faster reading of successive data

## Faster RAM technologies

---

- ◆ DDR RAM (double data rate)
  - ◆ Can send data twice per clock cycle
  - ◆ Send data on rising and falling edge of clock signal
  
- ◆ DRDRAM (Rambus DRAM)
  - ◆ Adds fast logic to RAM chips to allow faster data exchange between CPU and memory
  - ◆ For more information see <http://rambus.org>
  - ◆ Market share negligible
  
- ◆ Interleaved memory systems
  - ◆ Use more than one bank of memory chips
  - ◆ Used in vector machines and most 64-bit systems
  - ◆ Can read simultaneously from each bank
    - ◆ increases bandwidth
    - ◆ Does not change latency (access time)

## Improving memory speed by using caches

---

- ◆ Are added to speed up memory access (Opteron Barcelona)
  - ◆ Many GByte of slow DRAM
  - ◆ 2 MByte of fast and expensive L3-Cache
  - ◆ 512 kByte of even faster and more expensive L2-Cache per core
  - ◆ 2x64 kByte of the fastest and most expensive L1-Cache (instruction and data cache) per core
- ◆ Problems needing little memory will run faster!

## Comparison of memory/cache speeds

---

	Cycles	Normalised
L1 cache	2	1
L2 cache	15	7.5
L3 cache	75	37.5
Other L1/L2	130	65
Memory	~300	~150
1-hop remote L3	190	95
2-hop remote L3	260	130

AMD "Barcelona" @ 2GHz

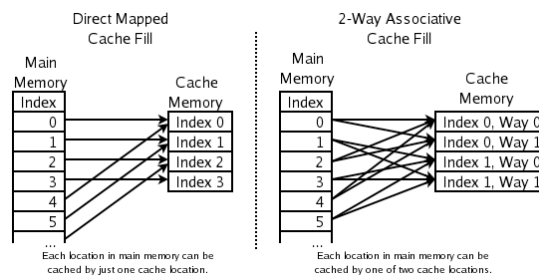
T. Roscoe, A. Baumann

## How does a cache work?

- ◆ CPU requests a word (e.g. 4 bytes) from memory
  - ◆ A full “cache line” (Opteron: 64 bytes) is read from memory and stored in the cache
  - ◆ The first word is sent to the CPU
- ◆ CPU requests another word from memory
  - ◆ Cache checks whether it has already read that part as part of the previous cache line
  - ◆ If yes, it the word is sent quickly from cache to CPU
  - ◆ If not, a new cache line is read
- ◆ Once the cache is full, the oldest data is overwritten
- ◆ *Locality of memory references are important for speed*

## Types of caches

- ◆ Direct mapped
  - ◆ Each memory location can be stored only in one cache location
  - ◆ “cache trashing” occurs if we access in strides of the cache size, always replacing the previous date
- ◆  $n$ -way associative
  - ◆ Each memory location can be stored in  $n$  cache locations
  - ◆ Better performance, more expensive
- ◆ Fully associative
  - ◆ Each memory location can be stored anywhere
  - ◆ Best but most expensive



## Exercises about caches

---

- ◆ Exercise 1:
  - ◆ Write a program to measure the number and size of caches in your machine
  
- ◆ Exercise 2 (bonus):
  - ◆ Write a program to determine the type of associativity of your L1-cache. Is it
    - ◆ Direct mapped?
    - ◆  $n$ -way associative?
    - ◆ Fully associative?

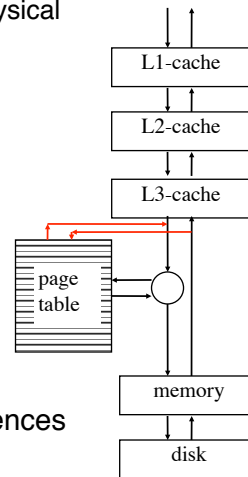
## Virtual memory: memory is actually even slower

---

- ◆ What if more than one program runs on a machine?
- ◆ What if we need more memory than we have RAM?
  
- ◆ Solution 1: virtual memory
  - ◆ Programs run in a “logical” address space
  - ◆ Hardware maps “logical” to “physical” address
  
- ◆ Solution 2: swap space
  - ◆ Some physical memory may be on a hard disk
  - ◆ If accessed it is first read from disk into memory
  - ◆ This is even slower!

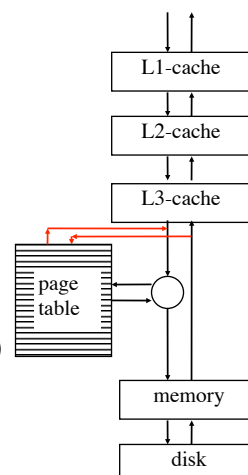
## Virtual memory logic:

- ◆ Memory is organized in “pages” of e.g. 4 Kbyte
  - ◆ Addresses are translated from logical to physical address space
  - ◆ Lookup in page table
    - ◆ If in memory, access to memory
    - ◆ If on disk, read from disk first (slow!!!)
- ◆ Access to page table needs reading from memory
- ◆ Solution: translation lookaside buffer (TLB)
  - ◆ Is a cache for the page table
- ◆ It is again important to keep memory references local



## Virtual memory: the worst case

- ◆ Request an address
- ◆ Cache miss in L1
- ◆ Cache miss in L2
- ◆ Cache miss in L3
- ◆ Lookup physical address
  - ◆ Cache miss in TLB
  - ◆ Request page table entry
  - ◆ Load page table from memory (slow)
- ◆ Page fault in page table
  - ◆ Store a page to disk (extremely slow)
  - ◆ Create and initialize a new page (very slow)
  - ◆ Load page from disk (extremely slow)
- ◆ Load value from memory (slow)
- ◆ Try to reuse data as much as possible



## What has changed in last few years?

Model	Freq.	Release	Benchmark
Intel Pentium 4	1.4 GHz	November 2000	0.25x
Intel Pentium 4 HT 471	3.8 GHz	June 2005	1x
Intel Core i7 - 970	3.2 GHz	July 2010	15x

source: [cpubenchmark.net](http://cpubenchmark.net)

- ◆ The performance used to be increased only by increasing the clock frequency
  - ◆ very expensive in terms of power
    - other components cannot reach the same operating frequency
- ◆ Nowadays performance is increased more efficiently:
  - ◆ Better pipelining and branch prediction
  - ◆ Better use of caches
  - ◆ Parallelism (multicore chipsets)