An Introduction to the hardware of your PC

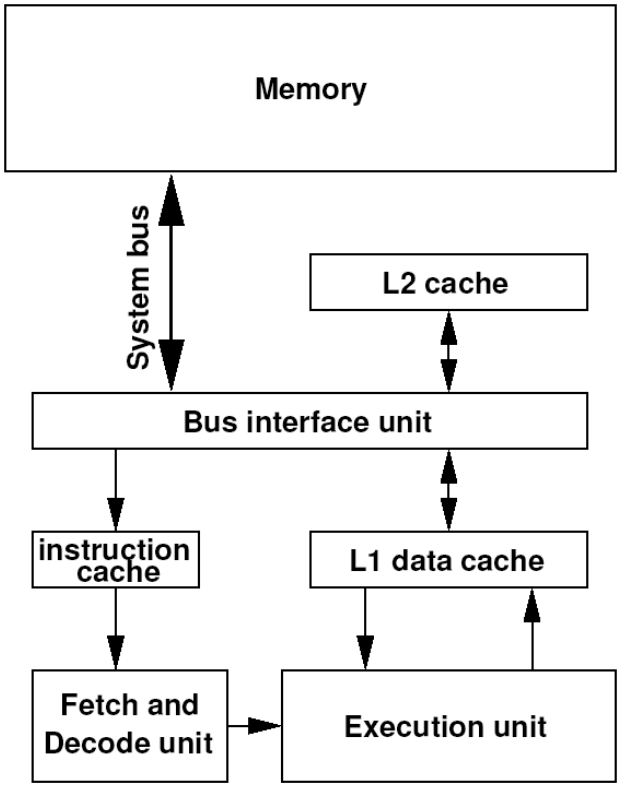
Know your tools!

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 specifed 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

Types of CPUs

- CISC (complex instruction set)
- RISC (reduced instruction set)
- post-RISC (superscalar)
- EPIC (explicitly parallel instruction set)
- Vector
- GPUs

CISC CPUs

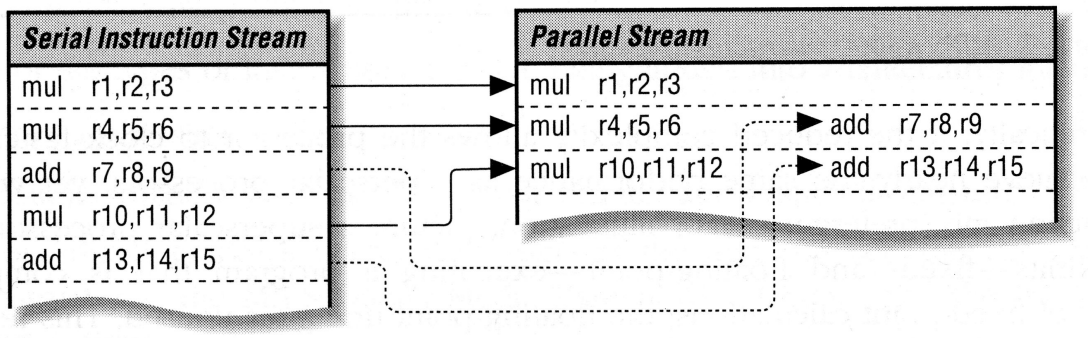
- Complex instruction set
  - Many high-level instructions (example: sin-cos-instruction)
  - Take many cycles to execute
    - High clock rate does not tell everything
- Examples
  - Intel IA-32/EM64T
  - AMD x86_64
- Advantage
  - High level instructions makes assembly language programming easy
- Disadvantage
  - 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/P JUGENE: PowerPC 450 850 MHz
- Advantages
  - fast and can be pipelined
  - Small and use little power
- Disadvantage
  - More machine language instructions needed

Superscalar and post-RISC processors

- Have more than one pipeline
  - Can execute instructions in parallel
  - Can reorder instructions
  - Even better execution speed
- But also get more complex than simple RISC processors



EPIC and Intel IA-64

- Explicitly Parallel Instruction set
  - E.g. Intel Itanium (IA-64)
- The machine language can specify which instructions can run simultaneously
  - CPU simplified since no automatic detection of independent instructions
  - Compilers get harder to write

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

RISC, CISC and GPUs in modern supercomputers

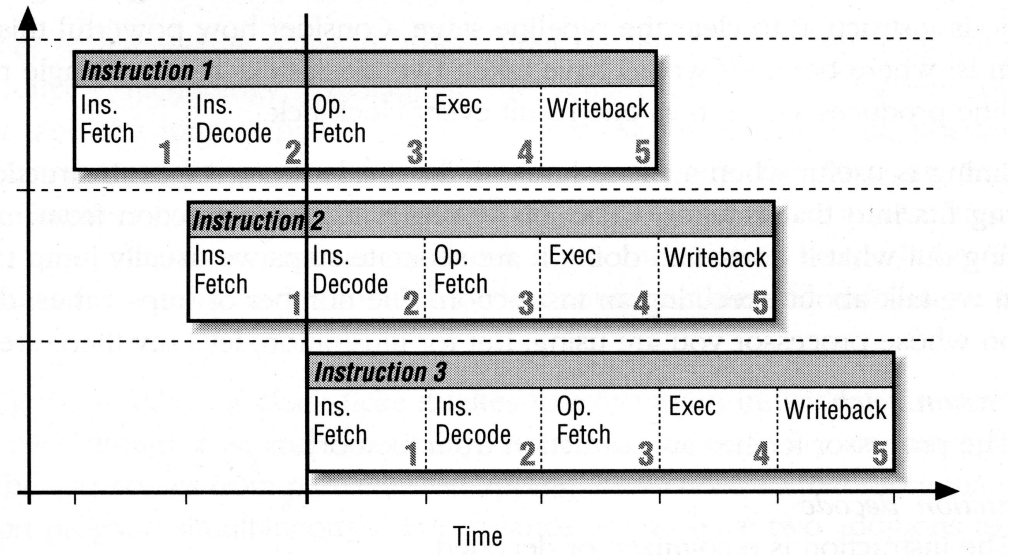
Processor Family	Count	Share %	Rmax Sum (GF)	Rpeak Sum (GF)	Processor Sum
Power	42	8.40 %	5833813	7343434	1418576
NEC	1	0.20 %	122400	131072	1280
Sparc	2	0.40 %	139110	152247	15104
Intel IA-64	5	1.00 %	269498	317132	50416
Intel EM64T	401	80.20 %	19276748	31534715	2664464
AMD x86_64	49	9.80 %	6793114	8991896	981621
Totals	**500**	**100%**	**32434683.70**	**48470495.53**	**5131461**

Source: top500.org

#	Name	CPU type	Vendor
1	Jaguar	Opteron	Cray
2	Nebulae	Xeon + Tesla	Dawning
3	Roadrunner	Opteron + Cell	IBM
4	Kraken	Opteron	Cray
5	Jugene	PowerPC	IBM

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];
}
```

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

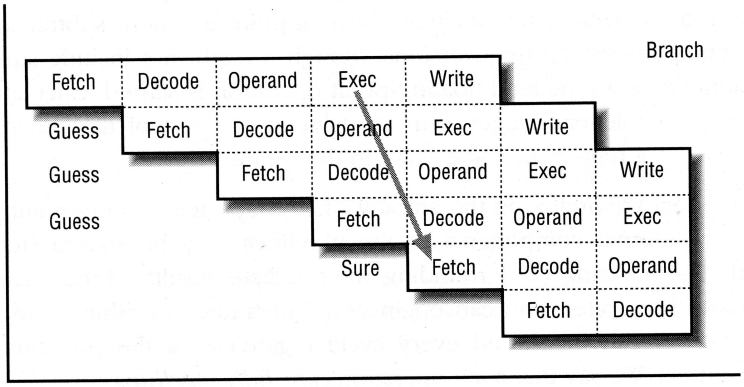
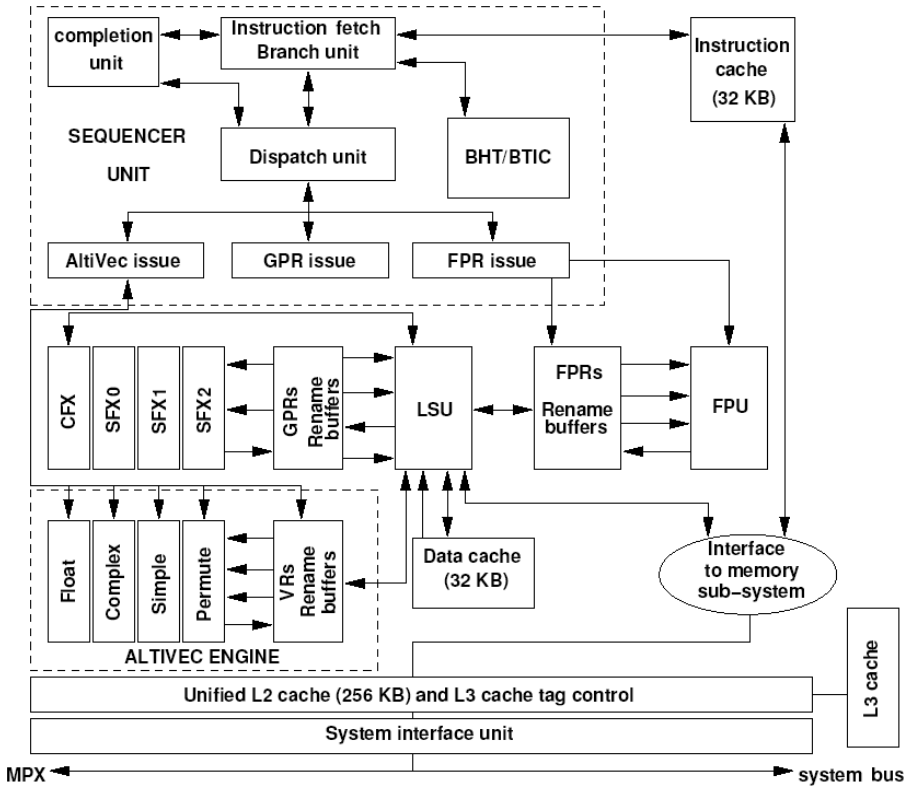


Diagram of a PowerPC G4 CPU



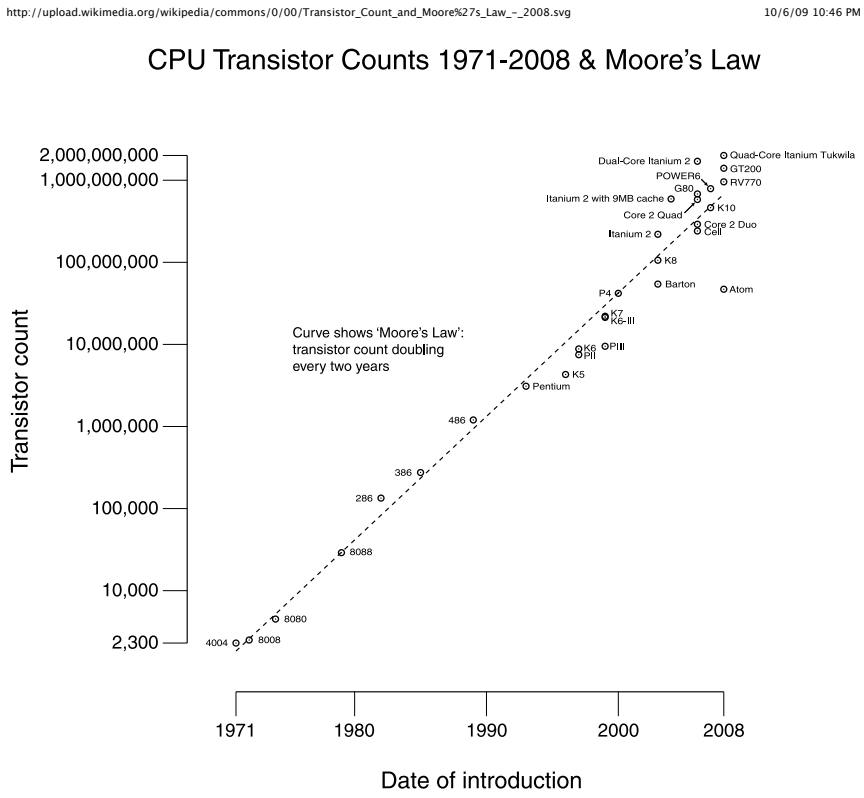
Summary of CPUs

- 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

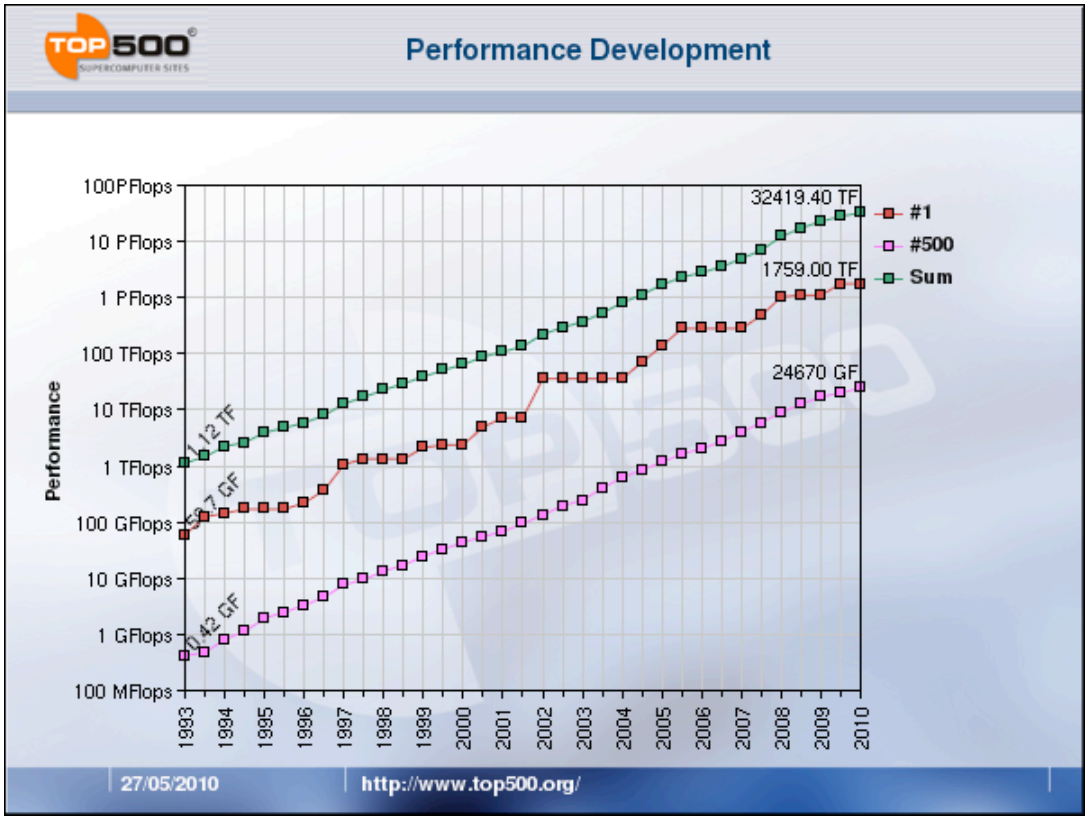
Moore's law

- "The number of transistors on a chip doubles every 18 months"
  - More transistors means smaller transistors
  - Smaller transistors => shorter distances => faster signals
  - Smaller transitors => 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 for Intel CPUs



Moore's law for supercomputers

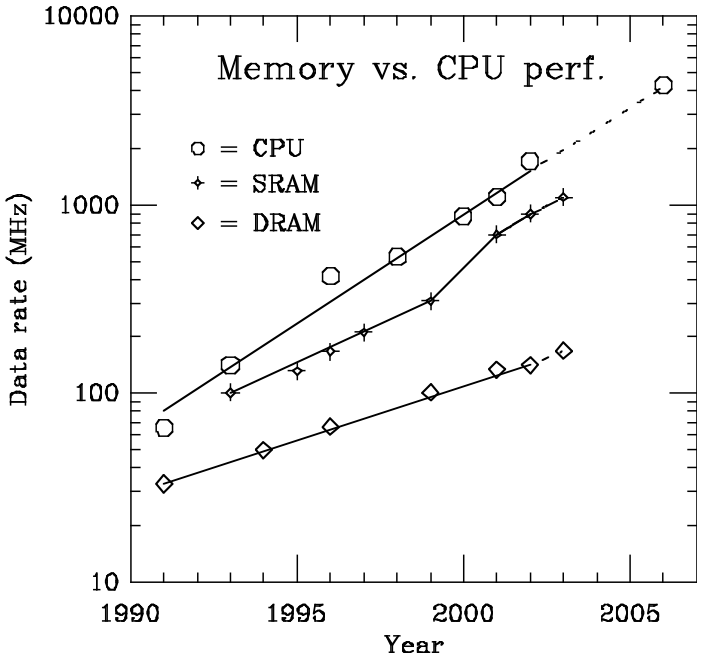


How about the other components of a computer?

- 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 then 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!

Improving memory speed by using caches

	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

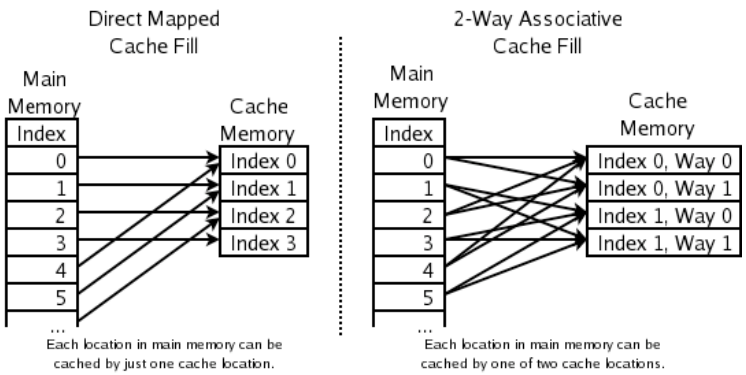
AMD "Barcelona" @ 2GHz

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

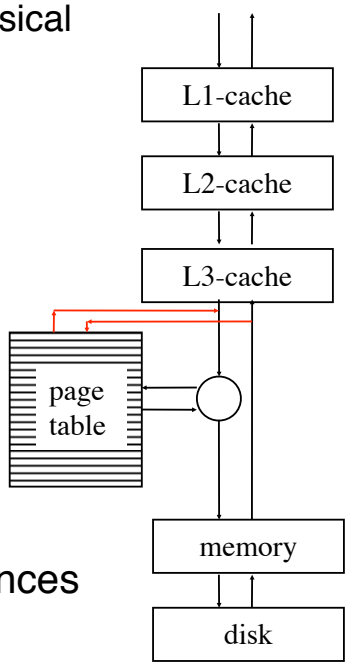
- Exercise 1:
  - Write a program to measure the number and size of caches in your machine
- Exercise 2 (bonus):
  - Write a program to determine the type of associativity of your L1-cache. Is it
    - Direct mapped?
    - *n*-way associative?
    - Fully associative?

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



TLB/Page sizes

Architecture	TLB Size	Page Size	TLB Coverage
VAX	64-256	512B	32-128kB
ia32 / x86 (typical)	32-32+64	4kB+4MB	128-128+256kB
MIPS	96-128	4kB-16MB	384kB - ...
SPARC	64	8kB-4MB	512kB - ...
Alpha	32-128+128	8kB-4MB	256kB - ...
RS/6000	32+128	4kB	128+512kB
Power4/G5	128	4kB+16MB	512kB - ...
PA-8000	96+96	4kB-64MB	
Itanium	64+96	4kB-4GB	

Not grown much in 20 years!

T. Roscoe, A. Baumann

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