Introduction to HPC

Lecture 2

Jakub Gałecki

Agenda

- CPU architecture crashcourse
- Assembly language – just a taste
- The problem with branching
- **Memory cache**
- The TLB
- Memory alignment
- Case study: the Goto algorithm

CPU architecture – a primer

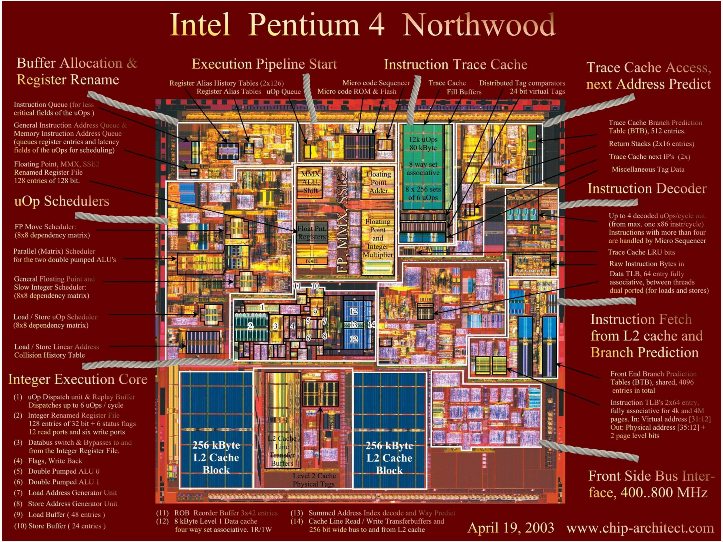
The CPU

CPU = Central Processing Unit

It is the brain of the computer

But that’s a bit vague…

The CPU


Intel Pentium 4 Northwood
Buffer Allocation & Register Rename
Instruction Queue (for less critical fields of the uOps)
General Instruction Address Queue & Memory Instruction Address Queue (queues register entries and latency fields of the uOps for scheduling)
Floating Point, MMX, SSE2 Renamed Register File 128 entries of 128 bit.
uOp Schedulers
FP Move Scheduler: (8x8 dependency matrix)
Parallel (Matrix) Scheduler for the two double pumped ALU's
General Floating Point and Slow Integer Scheduler: (8x8 dependency matrix)
Load / Store uOp Scheduler: (8x8 dependency matrix)
Load / Store Linear Address Collision History Table
Integer Execution Core
(1) uOp Dispatch unit & Replay Buffer Dispatches up to 6 uOps / cycle
(2) Integer Renamed Register File 128 entries of 32 bit + 6 status flags 12 read ports and six write ports
(3) Databus switch & Bypasses to and from the Integer Register File.
(4) Flags, Write Back
(5) Double Pumped ALU 0
(6) Double Pumped ALU 1
(7) Load Address Generator Unit
(8) Store Address Generator Unit
(9) Load Buffer (48 entries)
(10) Store Buffer (24 entries)
(11) ROB Reorder Buffer 3x42 entries
(12) 8 kByte Level 1 Data cache four way set associative. 1R/1W
(13) Summed Address Index decode and Way Predict
(14) Cache Line Read / Write Transferbuffers and 256 bit wide bus to and from L2 cache
Execution Pipeline Start
Register Alias History Tables (2x126)
Register Alias Tables
uOp Queue
Micro code Sequencer
Micro code ROM & Flash
Instruction Trace Cache
Trace Cache
Fill Buffers
Distributed Tag comparators
24 bit virtual Tags
MMX ALU, Shift
FP, MMX, SSE2
Floating Point Adder
Float Pnt. Registers
rom
Floating Point and Integer Multiplier
12k uOps 80 kByte
8 way set associative
8 x 256 sets of 6 uOps
1
2
3
4
5
6
7
8
9
10
11
12
13
14
12
256 kByte L2 Cache Block
L2 Cache Line Transfer Buffers
Level 2 Cache Physical Tags
256 kByte L2 Cache Block
Trace Cache Access, next Address Predict
Trace Cache Branch Prediction Table (BTB), 512 entries.
Return Stacks (2x16 entries)
Trace Cache next IP's (2x)
Miscellaneous Tag Data
Instruction Decoder
Up to 4 decoded uOps/cycle out. (from max. one x86 instr/cycle) Instructions with more than four are handled by Micro Sequencer
Trace Cache LRU bits
Raw Instruction Bytes in
Data TLB, 64 entry fully associative, between threads dual ported (for loads and stores)
Instruction Fetch from L2 cache and Branch Prediction
Front End Branch Prediction Tables (BTB), shared, 4096 entries in total
Instruction TLB's 2x64 entry, fully associative for 4k and 4M pages. In: Virtual address [31:12] Out: Physical address [35:12] + 2 page level bits
Front Side Bus Interface, 400..800 MHz
April 19, 2003
www.chip-architect.com

The CPU

But that was too specific…

The computer

Let’s start by placing things in context. The (modern) computer consists of:

- **The CPU**
- **RAM**
- **GPU** (last 2 lectures)
- The hard drive (whether HDD or SSD is irrelevant to us)
- I/O devices
- …

The CPU

Fundamentally, the CPU performs the following tasks:

- Fetches instructions
- Decodes instructions
- Executes instructions

How does it know where to fetch the instructions from: the program counter.

Examples of instructions:

- arithmetic operation
- read/write from/to memory
- **conditional jump**

Instructions usually operate on operands (arguments)

Registers

Small (e.g. 64 bit) volatile memory units

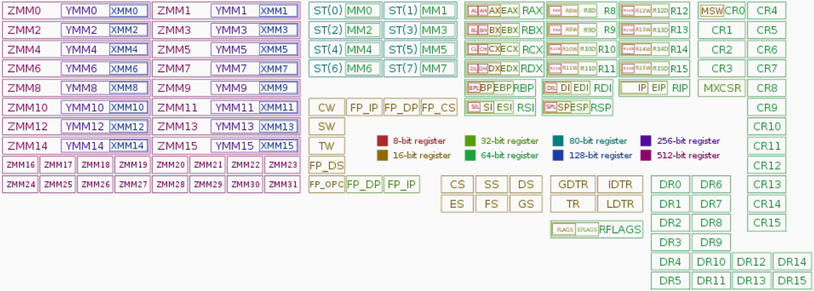
Most instructions involve data stored in registers

Registers have zero latency to access

Examples:

- General purpose
- RFLAGS
- Control
- Debug
- Vector (spoiler alert)

Registers of the x86-64 ISA



Hello, World! in x86 assembly

```
.LC0:
        .string "Hello, World!\n"
main:
        push    rbp
        mov     rbp, rsp
        mov     edi, OFFSET FLAT:.LC0
        call    puts
        mov     eax, 0
        pop     rbp
        ret
```

https://godbolt.org/z/or5Tz3cEb

The pipeline

Transistor reaction speed is not instantaneous.

- Gate delay: d
- Desired clock rate: f
- Theoretical max gate chain length: $1/df$

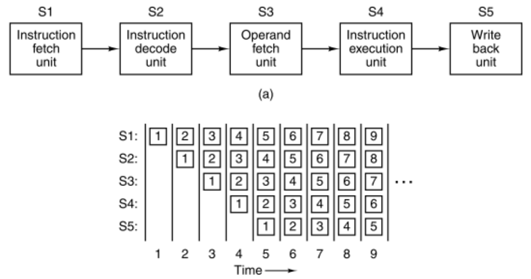
If we want fast clock frequencies, we have a hard, physical limit on the complexity of our circuit.

The solution: pipelining

We can break the instructions down into stages and execute 1 stage per cycle. Different stages of subsequent instructions are executed concurrently!

The pipeline

Simplified example, 5 stage pipeline:



Problem: what happens when instruction $n+1$ depends on the result of instruction n?

Hazards

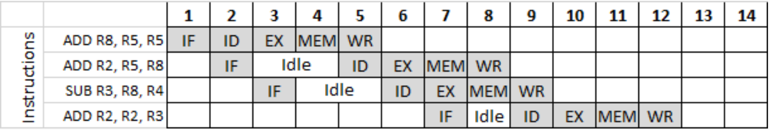
Pipelining introduces potential delays when subsequent operations depend on one another

- Structural hazard – resource conflict
- Data hazard – logical dependency between instructions
    - Read-after-write
    - Write-after-read
    - Write-after-write
- Control hazard – control flow depends on result of previous instruction

We should keep these in mind when programming, although the hardware and compiler do most of the heavy lifting.

Example

What kind of hazard is this?



		1	2	3	4	5	6	7	8	9	10	11	12	13	14
Instructions	ADD R8, R5, R5	IF	ID	EX	MEM	WR									
	ADD R2, R5, R8		IF	Idle		ID	EX	MEM	WR						
	SUB R3, R8, R4			IF	Idle		ID	EX	MEM	WR					
	ADD R2, R2, R3							IF	Idle	ID	EX	MEM	WR		

How to avoid hazards?

- Structural hazards: get better CPU (sorry)
- Data hazards: compiler optimization, out-of-order execution, register renaming, inline assembly if we’re feeling dangerous
- Control hazards: branch prediction, write better code (stay tuned)

A branching example

```
.LC0:
        .string "Hello, World!"
.LC1:
        .string "So many arguments :o"
main:
        sub     rsp, 8
        cmp     edi, 1
        jle     .L6
        mov     edi, OFFSET FLAT:.LC1
        call    puts
.L3:
        xor     eax, eax
        add     rsp, 8
        ret
.L6:
        mov     edi, OFFSET FLAT:.LC0
        call    puts
        jmp     .L3
```

`https://godbolt.org/z/5sWsYcvTd`

Branch prediction, speculative execution

The problem with branching: the CPU doesn't even know which instruction to fetch until some previous instruction executes

Control hazard == ”Data hazard on steroids”

The solution: take a guess and see what happens

- Correct guess: no stall, no performance penalty
- Incorrect guess: pipeline flush, undo changes – expensive

We need to try to be predictable.

Complete talk on the subject: `https://youtu.be/g-WPhYREFjk`

Live demo

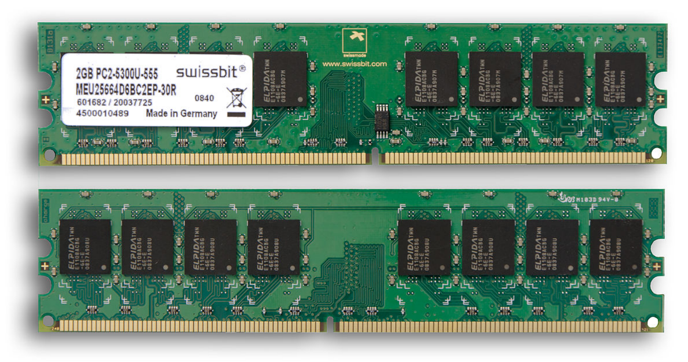
Accessing memory

DRAM

DRAM == Dynamic Random Access Memory

Very large – up to hundreds of GB

Very slow to access – hundreds of cycles

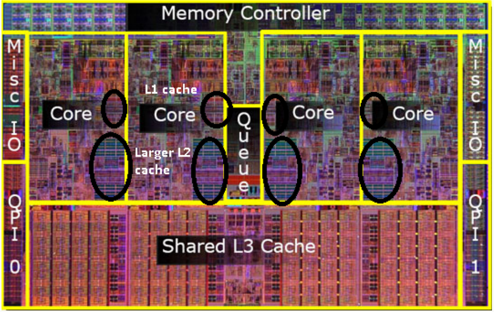


Cache

Cache == fast, on-die memory

Usually several levels, nowadays: L1I, L1D, L2, L3

Smaller →faster



Memory latency

This is, of course, dependent on the specific CPU model, but the order of magnitude for modern CPUs is as follows:

Memory hierarchy component	Latency [cycles]
Register	0
L1 Cache	4
L2 Cache	10-25
L3 Cache	~40
Main memory	200+

Source: Bakhvalov, D. (2020). Performance Analysis and Tuning on Modern CPUs.

Cacheline

The cache does not operate on individual bytes, but rather on sets of bytes, called **cachelines**.

The size of a cacheline on modern CPUs is 64B.

This has consequences:

- Aligning data to cache can increase performance
- Accessing neighboring data is faster
- Potential pitfall for concurrent programs (false sharing)

How do I use this?

There is no instruction for “write N bytes from memory to LX cache”*

We have to structure our data access so that it is naturally cache-friendly

Spatial locality:

- Subsequent addresses are likely to be on the same cacheline
- The CPU can detect access patterns and prefetch our data

Temporal locality:

- Least recently used cacheline gets evicted first
- Data which was recently accessed is likely still in cache

Live demo

Virtual vs physical memory

Data is ultimately represented by electrons residing in the DRAM die – *physical* address

Our program references memory via *virtual* addresses

To de-conflict different processes, the OS *translates* virtual addresses to physical addresses

The CPU has special hardware which helps with translation

For improved efficiency, memory is divided into 4kB* *pages*

The TLB

TLB = Translation Lookaside Buffer

Cache for the page translation process

TLB size: 1536 pages

TLB hit time: $\leq$1 cycle

TLB miss penalty: 10-100 cycles

Memory thrashing for large working sets with random memory access

Usually not an issue

Alignment

We say address i is aligned to a (or has alignment a) iff

$$i \mod a = 0$$

where a must be a power of 2. For example:

- `0xa0` is aligned to 16
- `0x0777b2` is aligned to 2

CPUs are much better at accessing data which is aligned to its natural alignment, i.e., a multiple of its size.

For usual cases, this is handled by the compiler with padding: `https://godbolt.org/z/39aWbGoKW`.

We can use `alignas` or aligned allocation to override the defaults. We will soon see why this may be desired.

Case study: Goto algorithm

Author: Kazushige Goto (early 2000’s)

Matrix-matrix multiply algorithm explicitly catering to the 3 level cache memory hierarchy

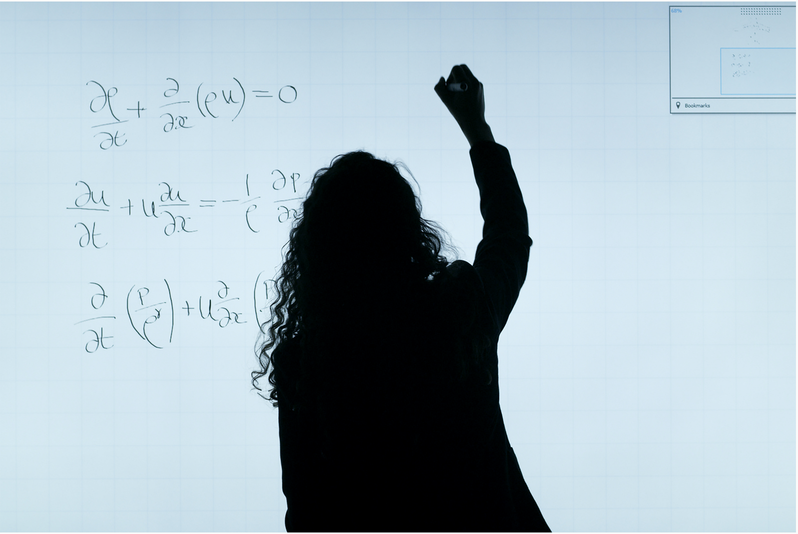
Slice & dice approach

General structure: simple, no CS PhD required

Micro-kernel: detailed knowledge of the CPU architecture is required

Fantastic explanation: `https://youtu.be/07SMaudtH6k`

To the whiteboard!



Summary

- CPU architecture 101
- Assembly 101
- CPUs are pipelined
- Avoid unpredictable branches
- Cache is king

Key takeaways

- → Want performance? Know your hardware!
- → The speed of feeding the data to the CPU is equaly as important as the speed of processing the data
- → Break down the problem, optimize the kernel

Q&A