B4M36ESW – Efficient software

Lecture 1: Introduction, modern computer architecture, compiler, profiling

Michal Sojka
sojkam1@fel.cvut.cz

About this course

- **Michal Sojka**
  - C/C++, embedded systems, operating systems
- **David Šišlák**
  - Java, servers, ...
- **Scope**
  - Writing fast programs
  - Single (multi-core) computer, no distributed systems/clouds
  - Interaction between software and hardware

Lecture outline I.

1. Intro: how to write efficient programs, modern computer architectures, energy consumption
2. Benchmarking, metrics, statistics, WCET, timestamping, profiling (perf, *trace, cachegrind)
3. Program execution – virtual machine, byte-code, Java compiler, JIT compiler, relation to machine code, byte-code analysis, Java byte-code decompilation, compiler optimalization, program performance analysis
4. Scalable synchronization – from mutexes to RCU (read-copy-update), transactional memory, scalable API, SIM commutativity
5. JVM concurrency – parallel data accesses, lock monitoring, atomic operations, lock-less/block-free data structures, non-blocking algorithms (fronta, zásobník, množina, slovník)
6. Data serialization – JSON, XML, protobufs, AVRO, cap'n'proto, mmap/shared memory

Lecture outline II.

7. Memory access – cache memory, dynamic memory allocation (malloc, NUMA, …)
8. Efficient servers, C10K problem, non-blocking IO, native cache memory in JVM
9. Representation of objects in JVM – definition loading, materialization of class definition, class initialization, instance initialization, class loader, class finalization, freeing of class definitions
10. JVM memory management – memory organization, data representation, memory management algorithms and their parameters
11. Type of links to instances in Java, efficient cache memory, static and dynamic memory analysis, data structures with reduced memory management overheads, bloom filters
12. Virtualization (IOMMU, SR-IOV, PCI pass-through, virtio, …)
13. Program execution – C compiler (restrict qualifier, optimization), SIMD

Grading

- **Exercise: 60 points**
  - 7× small task
  - semestral work (both C and Java)
  - **Minimum 30 points**
- **Exam:**
  - Written test: 30 points
  - Voluntary oral exam: 10 points
  - **Minimum: 20 points**

Your participation

- There are many techniques how to make your program more efficient
- We will cover only few techniques in this course
- Hardware is still evolving – what was efficient in the past may no longer work today
- We are open to discussion

Efficient software

Efficient software

- **There is no theory of how to write efficient software**
- **Writing efficient software is about:**
  - Knowledge of all layers involved
  - Experience in knowing when and how performance can be a problem
  - Skill in detecting and zooming in on the problems
  - A good dose of common sense
- **Best practices**
  - Patterns that occur regularly
  - Typical mistakes

Fundamental theorem of software engineering

"All problems in computer science can be solved by another level of indirection"

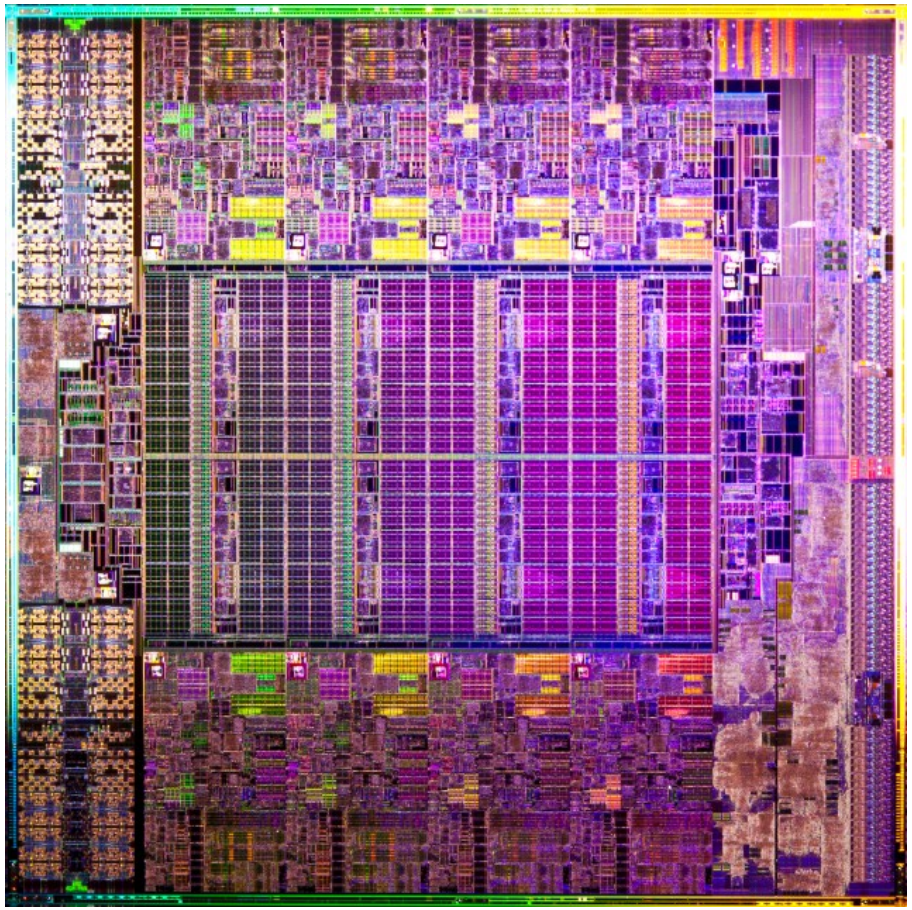
"...except for the problem of too many layers of indirection."

Layers of indirection in today's systems

- **Hardware**
  - microcode, ISA
  - virtual memory, MMU
  - buses, arbiters
- **Software**
  - operating system kernel
  - compiler
  - language run-time
  - application frameworks

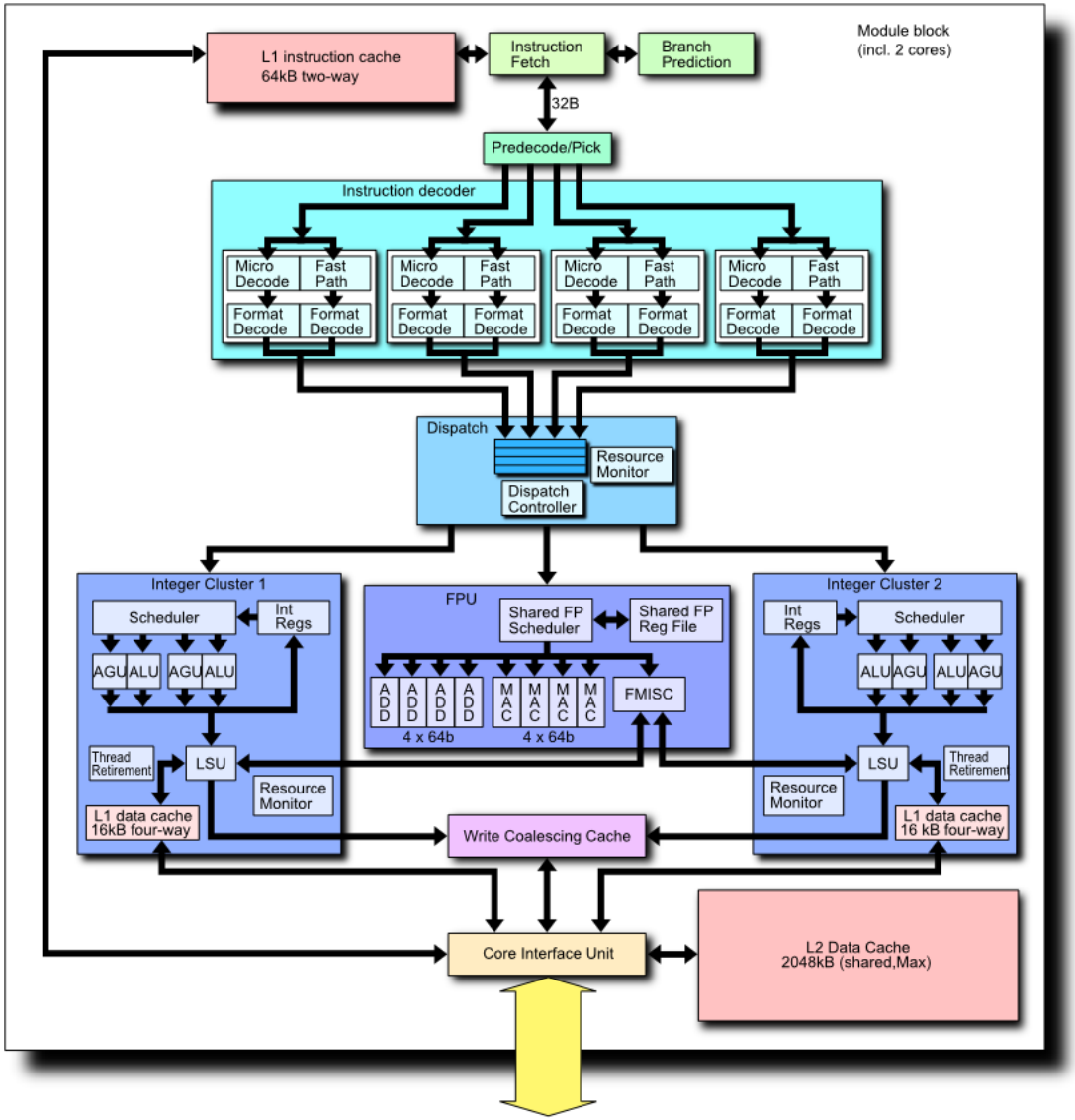
Hardware optimizations

- **Done by hardware manufacturers**
  - Programmers need to know how to use them properly
- **Instruction-level parallelism**
  - e.g. 2 integer, 2 floating point, 1 MMX/SSE units working in parallel
  - vector instruction (SIMD)
  - Cache hierarchy
  - Prefetching of data
  - Branch prediction

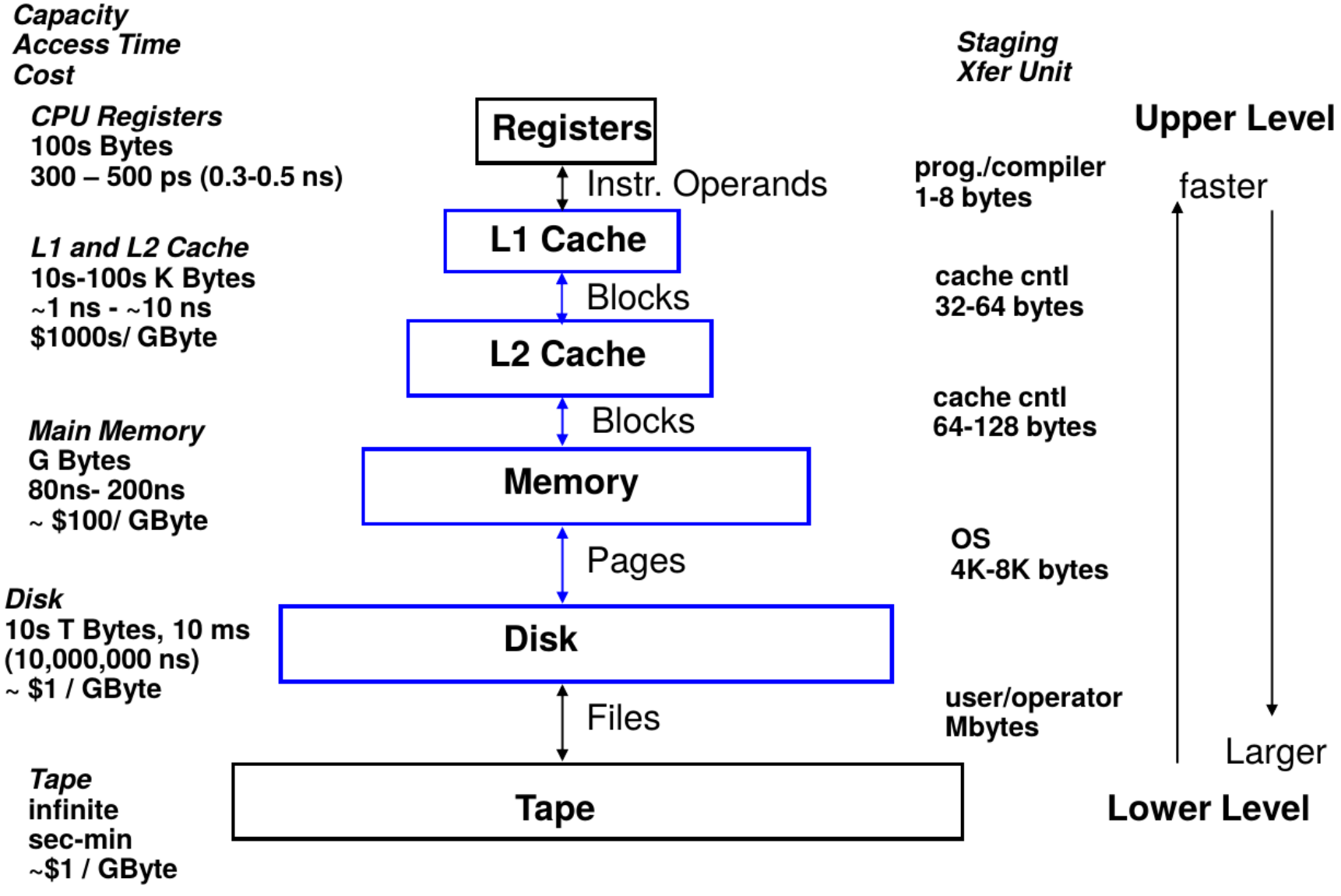


Intel Xeon E5 (Source: extremetech.com)

AMD Bulldozer CPU



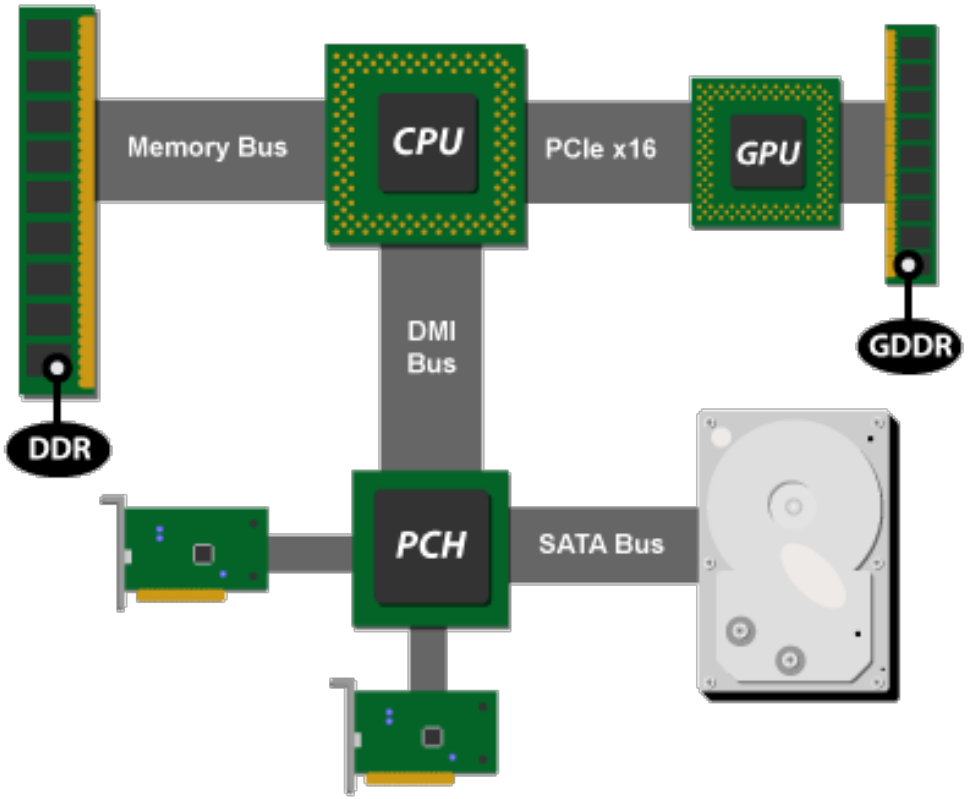
Memory hierarchy



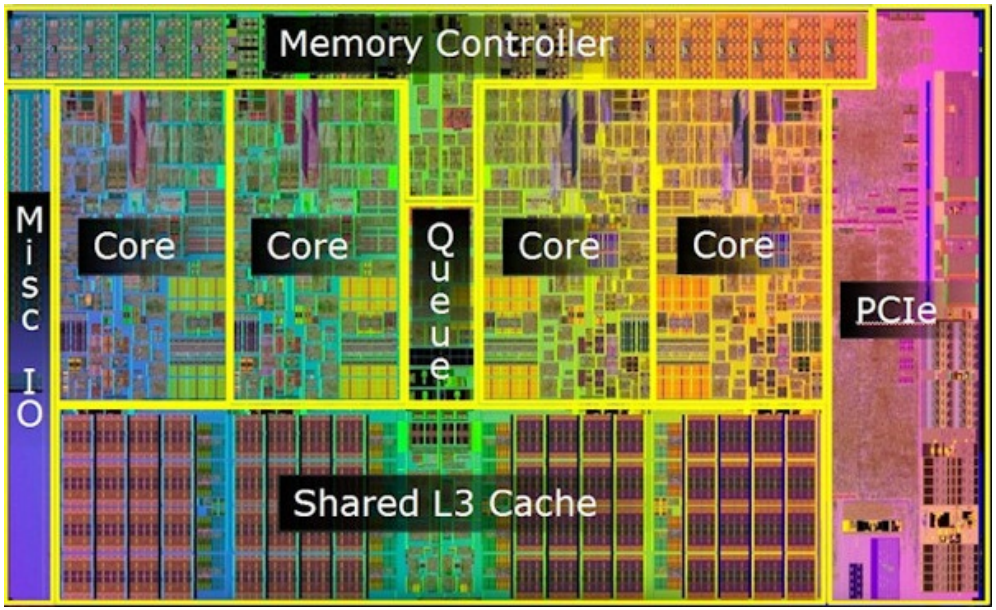
Latencies in computer systems

Event	Latency	Scaled
1 CPU cycle	0.3 ns	1 s
Level 1 cache access	0.9 ns	3 s
Level 2 cache access	2.8 ns	9 s
Level 3 cache access	12.9 ns	43 s
Main memory access (DRAM, from CPU)	120 ns	6 min
Solid-state disk I/O (flash memory)	50–150 ns	2–6 days
Rotational disk I/O	1–10 ms	1–12 months
Internet: San Francisco to New York	40 ms	4 years
Internet: San Francisco to United Kingdom	81 ms	8 years
Internet: San Francisco to Australia	183 ms	19 years
TCP packet retransmit	1–3 s	105–117 years
OS virtualization (container) system reboot	4 s	423 years
SCSI command timeout	30 s	3 millennia
HW virtualization system reboot	40 s	4 millennia
Physical server system reboot	5 m	32 millenia

Intel-based system (single socket)

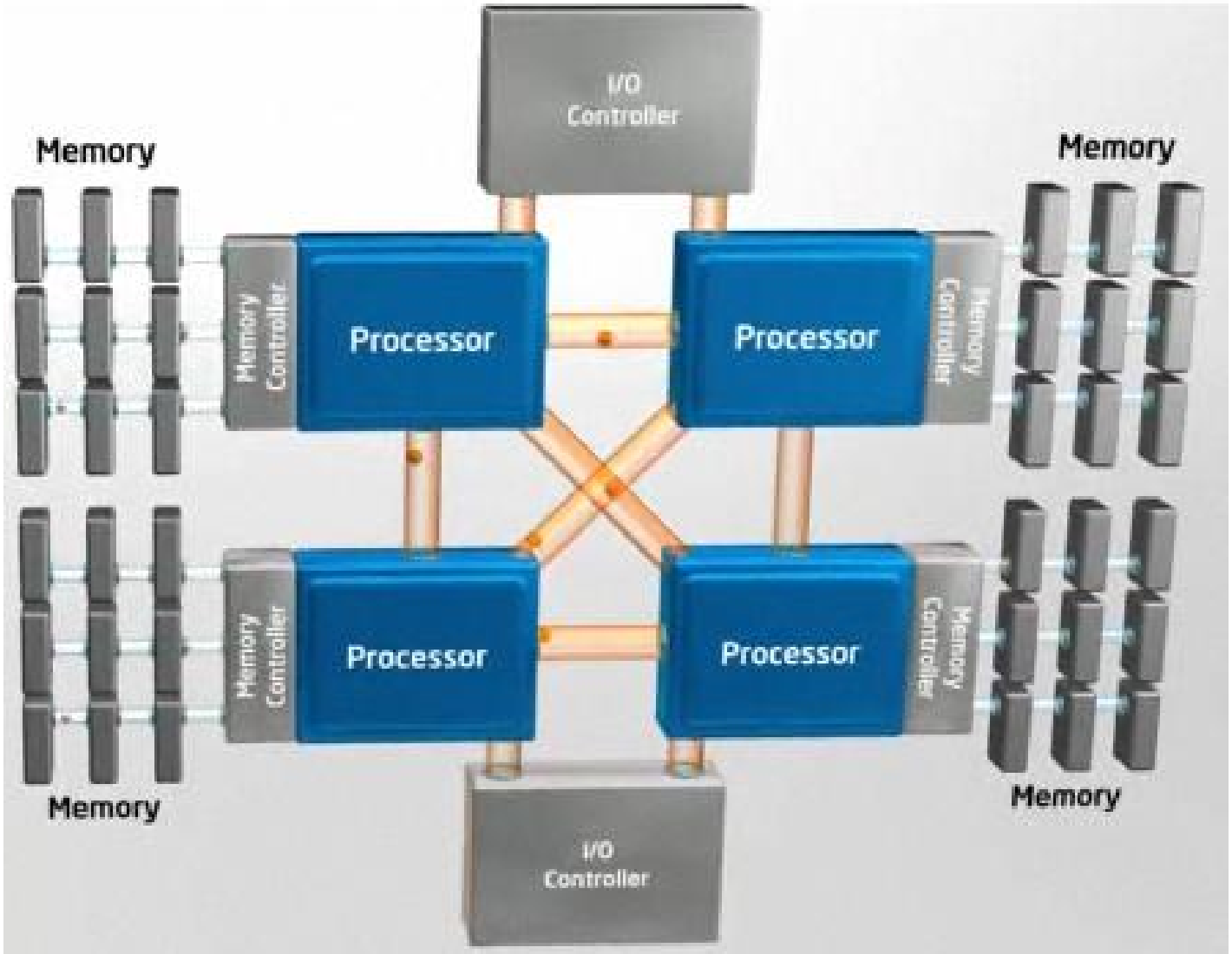


Intel's P55 platform. Source: ArsTechnica

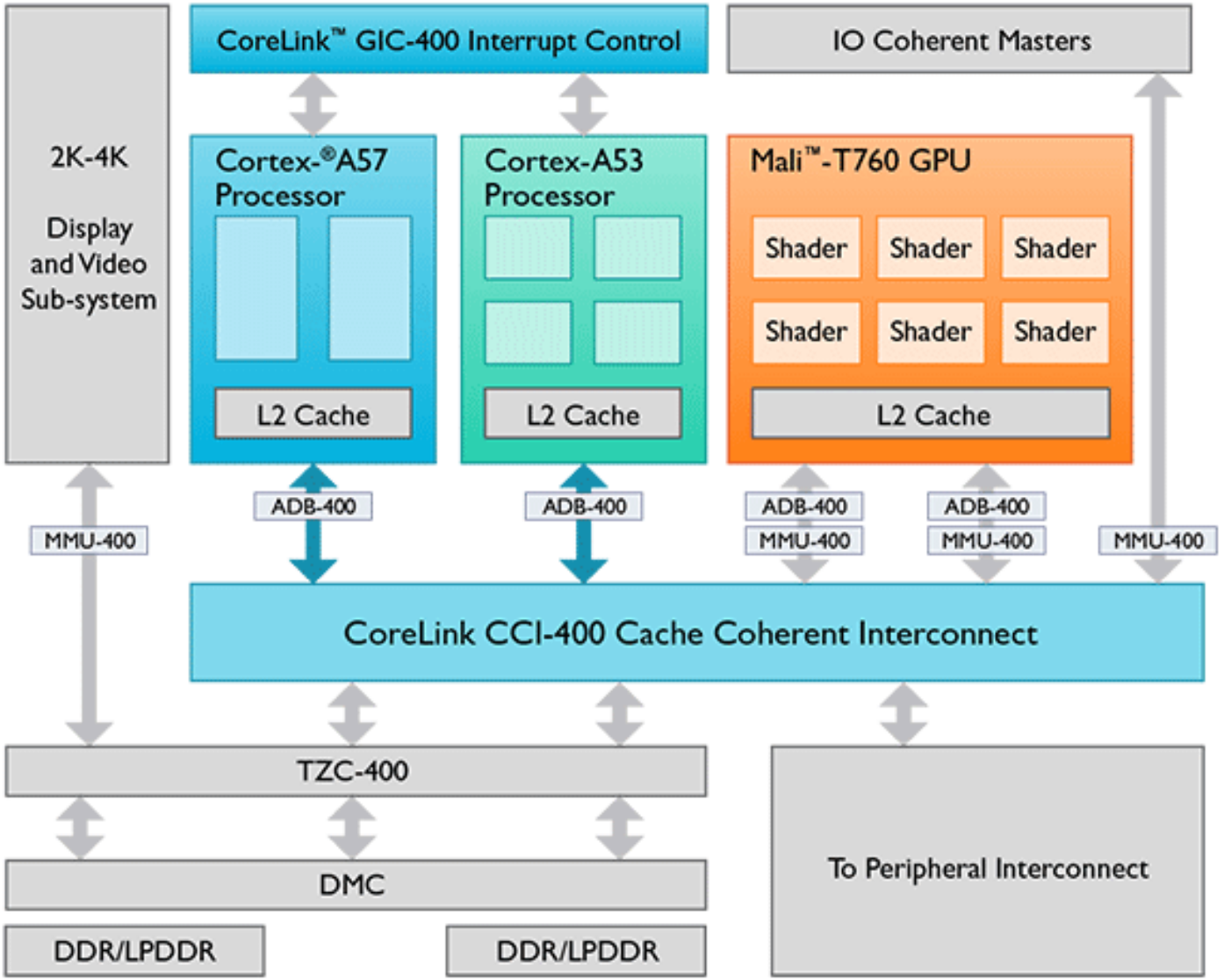


Lynnfield CPU. Source: Intel

Non-Uniform Memory Access (NUMA)



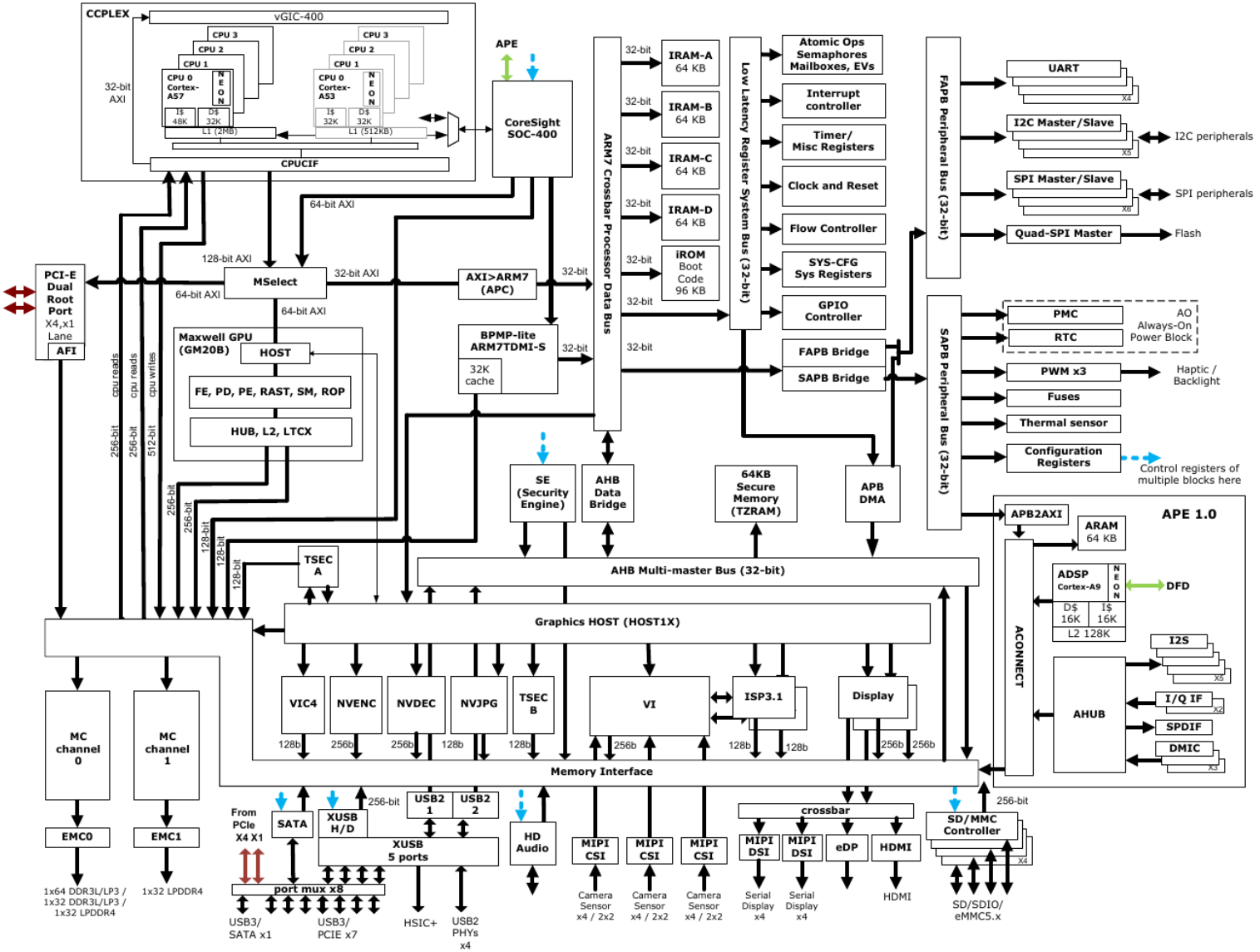
Embedded multi-core system (SoC)



Source: ARM

Nvidia Tegra X1

More detailed diagram of an embedded SoC



Energy is the new speed

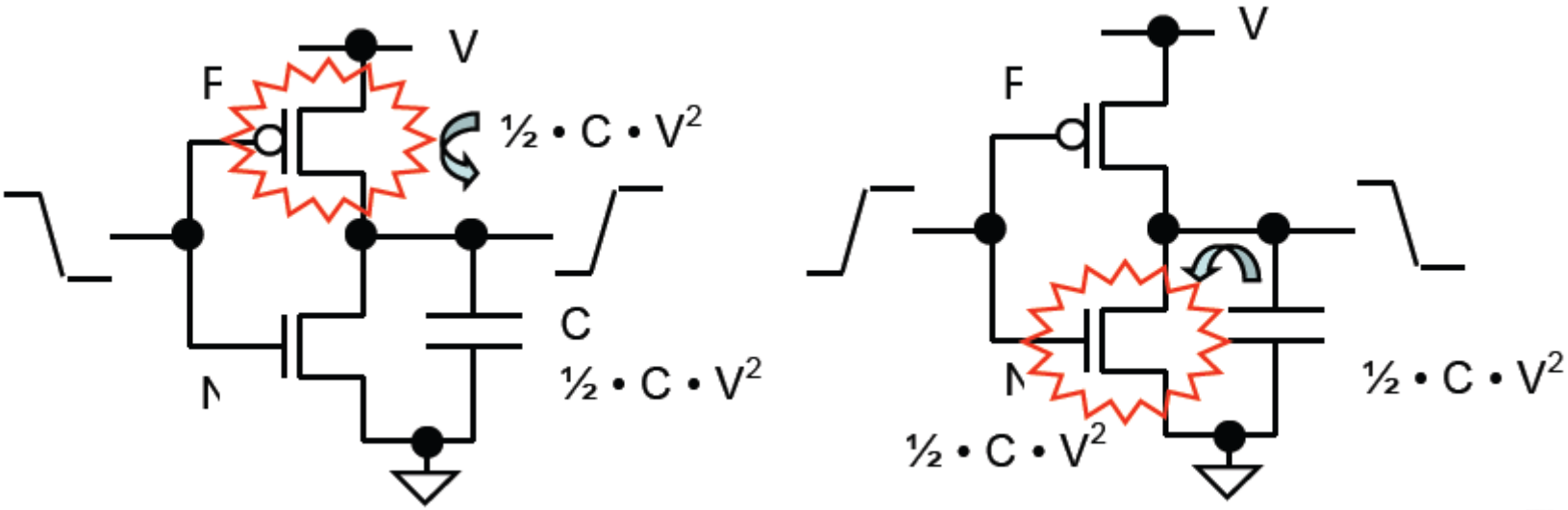
- Today, we no longer want just fast software
- We also care about heating and battery life of our mobile phones
- Good news: Fast software is also energy efficient

Power consumption of CMOS circuits

- **Two components:**
  - Static dissipation
    - leakage current through P-N junctions etc.
    - higher voltage → higher static dissipation
  - Dynamic dissipation
    - charging and discharging of load capacitance (useful + parasitic)
    - short-circuit current

$$P_{total} = P_{static} + P_{dyn}$$

Dynamic power consumption, gate delay

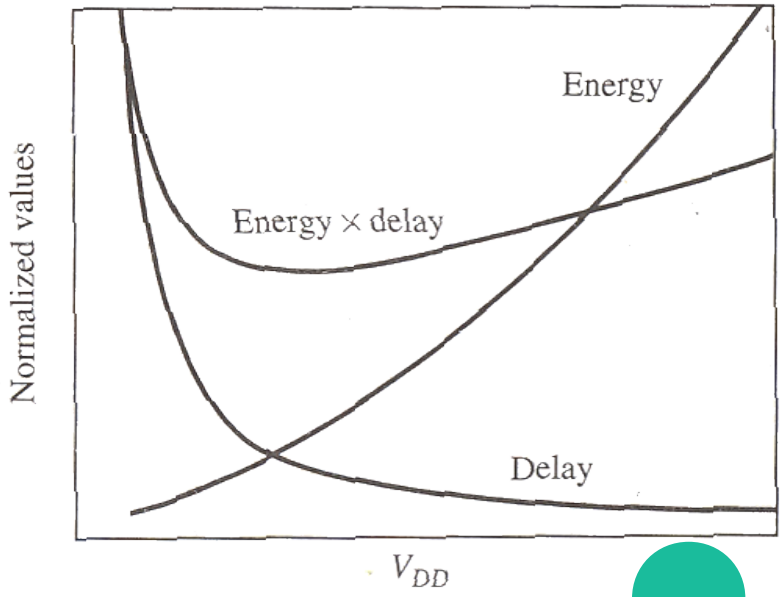


$$P_{dyn} = a \cdot C \cdot V_{dd}^2 \cdot f$$

- a – activity factor
- f – frequency

$$t = \frac{\gamma \cdot C \cdot V_{dd}}{(V_{dd} - V_T)^2} \approx \frac{1}{V_{dd}}$$

- Low power ⇒ slow
- Voltage and frequency must be related



Methods to reduce power/energy consumption

- use better technology/smaller gates
- use better placing and routing on the chip
- reduce power supply V_{DD}
- reduce frequency

} dynamic voltage & frequency scaling (DVFS)
Note: ramp-up latency

- reduce activity (clock gating)
- **use better algorithms and/or data structures**

Software

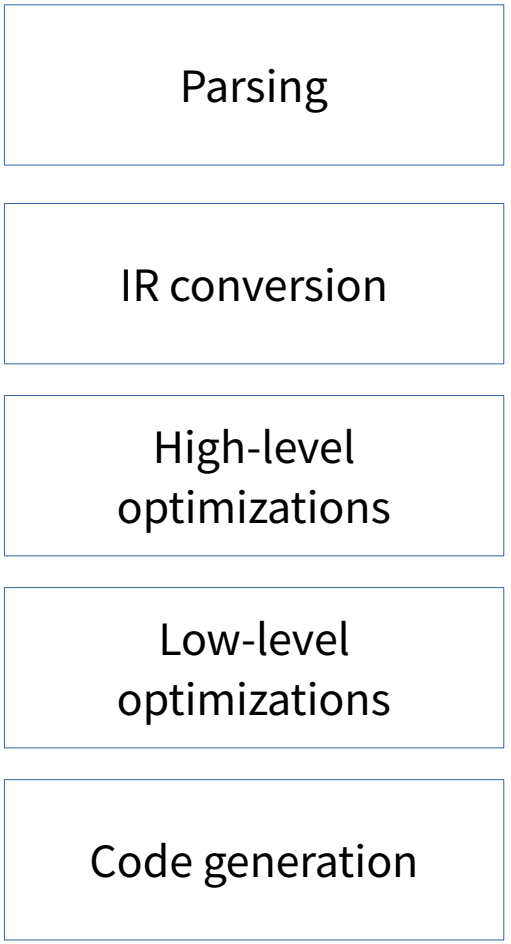
Bentley's Rules (Writing Efficient Programs)

- **Space-for-Time Rules**
  - Data Structure Augmentation
  - Store Precomputed Results
  - Caching
- **Time-for-Space Rules**
  - Packing
  - Interpreters

http://www.new-npac.org/projects/cdroms/cewes-1999-06-vol1/nhse/hpccsurvey/orgs/sgi/bentley.html

C/C++ compiler

- gcc, clang (LLVM), icc, ...
- Parsing → syntax tree
- Intermediate representation
- High-level optimizations – HW independent
- Low-level optimizations – HW dependent
- Code generation: IR → machine code



GCC high-level optimizations

- **Dead code elimination (if (0))**
- **Elimination of unused variables**
- **Constant propagation**
  - void func(int i) { if (i!=0) { ... } }
  - func(0); // Nothing happens
- **Variable propagation to expressions**
  - x = a + const1;
  - if (x == const2) goto ... else goto ...
  - if (a == (const2 - const1)) goto ... else goto …
- **Elimination of subsequent stores (a=1; a=2)**
- **Loop optimization (operations are replaced by SIMD instructions (MMX, SSE) etc.)**
- **Simplification of built-in functions (e.g. memcpy).**
- **Tail call (at the end of a function) can be replaced by a jump.**

GCC low-level optimizations

- **Common subexpression elimination – intermediate values are stored in temporary variables/registers.**
- **Selections of addressing modes with respect to their "price"**
- **Loop optimization (unrolling, modulo scheduling, …)**
- **Combining multiple operations to one instruction**
- **Allocation of correct registers for operands and variables, decision of what will be stored on the stack and what in the registers.**
  - Variables can be moved between stack and registers during execution
- **Instruction reordering for faster execution (optimal use of multiple ALU units in the CPU)**

Compiler flags (gcc, clang)

- **Optimization level: -O0, -O1, -O2, -O3, -Os (size)**
  - -O2 is considered “safe”, -O3 may be buggy
  - Individual optimization passes:
    - -ftree-ccp, -ffast-math, -fomit-frame-pointer, -ftree-vectorize
- **Code generation**
  - -fpic, -fpack-struct, -fshort-enums
  - Machine dependent
    - -march=core2, -mtune=native, -m32, -minline-all-stringops, ...
- **Debugging: -g**
- **“(p)info gcc” is your friend**

Do not trust the compiler :-)

- gcc -save-temps – saves intermediate files (assembler)
- objdump -d – disassembler
- objdump -dS – disassembler + source (needs gcc -g)

Example

```c
void vecadd(int * a, int * b, int * c, size_t n) {
    for (size_t i = 0; i < n; ++i) {
        a[i] += c[i];
        b[i] += c[i];
    }
}
```

veclib.c

```
gcc -Wall -g -O0 -march=core2 -o vecadd *.c
./vecadd       # time = 0.37
gcc -g -O2 -march=core2 -o veclib.o veclib.c
./vecadd       # time = 0.12 ~ 300% speedup
objdump -d veclib.o
```

```c
unsigned a[MM], b[MM], c[MM];

int main() {
    clock_t start,end;

    for (size_t i = 0; i < MM; ++i)
        a[i] = b[i] = c[i] = i;

    start = clock();
    vecadd(a, b, c, MM);
    end = clock();

    printf("time = %lf\n", (end – start)/
            (double)CLOCKS_PER_SEC);

    return 0;
}
```

vecadd.c

```
vecadd:
   xor     %eax,%eax
   test    %rcx,%rcx
   je      29 <vecadd+0x29>
   nopw    0x0(%rax,%rax,1)

   mov     (%rdx,%rax,4),%r8d
   add     %r8d,(%rdi,%rax,4)
   mov     (%rdx,%rax,4),%r8d
   add     %r8d,(%rsi,%rax,4)
   add     $0x1,%rax
   cmp     %rax,%rcx
   jne     10 <vecadd+0x10>
   retq
```

?

Pointer aliasing

- vecadd must work also when called as vecadd(a, a, a, MM)
- Pointer aliasing = multiple pointers of the same type can point to the same memory
  - prevents certain optimizations
- restrict qualifier = promise that pointer parameters of the same type can never alias

```
void vecadd(int * restrict a, int * restrict b, int * restrict c, size_t n)
{ ... }
```

- ./vecadd # time = 0.10, speedup 12%!

Profiling the code

Profiling the code

- “Premature optimization is the root of all evil”
  — D. Knuth
- Software is complex!
- We want to optimize the bottlenecks, not all code
- Real world codebases are big: Reading all the code is a waste of time (for optimizing)
- Profiling: Identifies where your code is slow

Bottlenecks

- **Sources**
  - code
  - memory
  - network
  - disk
  - ...

Profiling tools

In order to do:	You can use:
Manual instrumentation	printf and similar
Static instrumentation	gprof
Dynamic instrumentation	callgrind, cachegrind
Performance counters	oprofile, perf
Heap profiling	massif, google-perftools

- **Instrumentation = modifying the code the perform measurements**

Static instrumentation: gprof

- **gcc -pg ... -o program**
  - Adds profiling code to every function/basic block
- **./program**
  - generates gmon.out
- **gprof program**

```
Flat profile:

Each sample counts as 0.01 seconds.
%   cumulative self              self    total
time seconds    seconds calls s/call s/call name
33.86 15.52     15.52    1    15.52  15.52  func2
33.82 31.02     15.50    1    15.50  15.50  new_func1
33.29 46.27     15.26    1    15.26  30.75  func1
0.07  46.30     0.03                       main
```

Event sampling

- **Basic idea**
  - when an interesting event occurs, look at where program executes
  - result is histogram of addresses and event counts
- **Events**
  - time, cache miss, branch-prediction miss, page fault
- **Implementation**
  - timer interrupt → upon entry, program address is stored on stack
  - each event has counting register
    - when threshold is reached, an interrupt is generated

Performance counters

- Hardware inside the CPU (Intel, ARM, ...)
- Software can configure which events to count and when/whether to generate interrupts
- In many cases can be accessed from application code
- Documentation:
  - Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3: System Programming Guide
  - Intel® 64 and IA-32 Architectures Optimization Reference Manual
  - ARM® Architecture Reference Manual ARMv8, for ARMv8-A architecture profile

perf

- linux-tools package
- Can monitor both HW and SW events
- Can analyze:
  - single application
  - whole system
  - ...
- https://perf.wiki.kernel.org/

perf usage

- perf list
- perf stat -e cycles -e branch-misses -e branches -e cache-misses -e cache-references ./vecadd

```
Performance counter stats for './vecadd':

    1,898,543,656      cycles                                                  (79.98%)
          267,572      branch-misses             #    0.08% of all branches          (79.97%)
      348,090,074      branches                                                (79.95%)
       20,232,628      cache-misses              #   75.588 % of all cache refs      (80.51%)
       26,767,103      cache-references                                        (80.09%)

      0.619472916 seconds time elapsed
```

perf usage II.

- perf record -e cycles -e branch-misses ./vecadd
- perf report

Profiler-guided compilation

Excercise – ellipse detection

- **Passes**
  - scale the image
  - convert to gray
  - blur to have less details
  - find edges
  - find continuous components
    - if component looks roughly like ellipse
      - run RANSAC algorithm to fit the ellipse precisely

Excercise – ellipse detection

- RANSAC algorithm (Random sample consensus)