Performance-optimised computing – Week 2.

Instruction set, Processors, Memory access,

Dr. Bakay Árpád – Ericcson

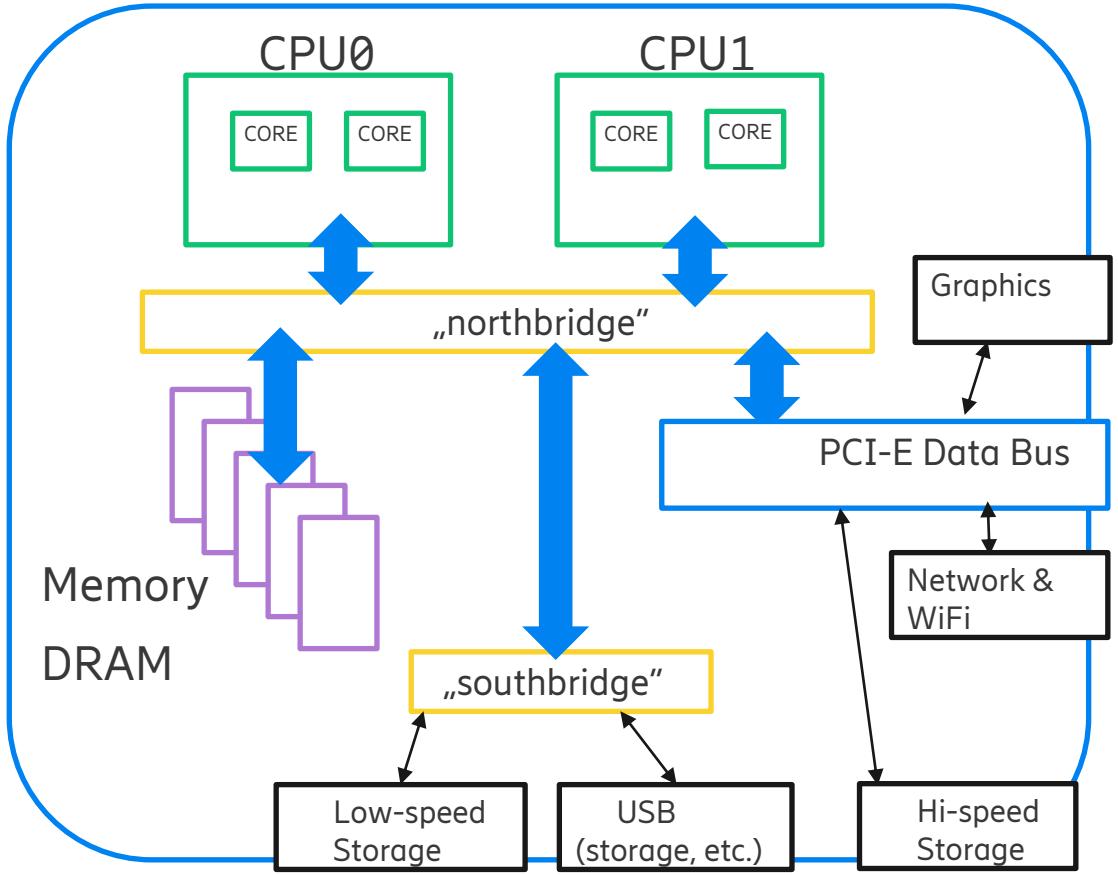
Fall Semester, 2024

Typical Computer Architecture

server or desktop, around 2010

- CPU Core and X86 Architecture
- CPU as a unit
- Memory
- Storage
- Networking

- Software, etc.



Instruction Set and Machine Code

- Each processor architecture has a well-defined set of „commands" supported – this is the **instruction set**
  - Contians dozens (or hundreds of) instructions which execute simple operations, e.g
    - Move data between memory and CPU-internal registers, or between registers
    - Add/Mul/Subtract/Divide 2 integer or floating-point numbers stored in registers
    - Branch to a location if previous operation resulted in `0`
    - Jump to a location saving current address in stack (call), jump back to a location on the stack (return)
  - **Instructions are encoded** into a few (1 to 15) bytes, and placed contigiously into a memory section used for „code" -> x86 Assembly/Machine Language Conversion – Wikibooks
    - **„Machine code"** are sequences of instructions.
  - Most operations refer to 1, 2 (or 3) **registers**, which hold „hot" data in the CPU,
    - The fastest access possible (about 100x faster than main memory)
- Full X86/64 CPU documentation is free and published (...but long):
  - Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4
  - 1. Basic architecture, 2. Instruction set, 3. Programming Gudie, 4. Model-specific info

X64 Data: Registers, Memory and Immediate

- **General purpose registers** are for 64 bit integers (16 regs)
  - Names in doc (and in assembly) AX, BX, CX, DX, DI, SI, BP, SP + R8 .. R15,
  - Accessible in different widths (8bits: AL,/AH, 16: AX, 32: EAX, 64: RAX)
- **IP (or RIP)** is s special (automatically incremented) register pointing to the next instruction
- **Vector registers** are 64-512 bytes (8,16 or 32 regs) and applicable for integer AND for floating point, also for „vector" (SIMD) operations
- **Floating point regs**: st0-st7 + control + status (aliased on vector regs).
- Various **status** and **control** registers

- Memory is a contigious byte store of up to 2^64 bytes

- Immediate: Input data (constants) may be encoded into the instructions.

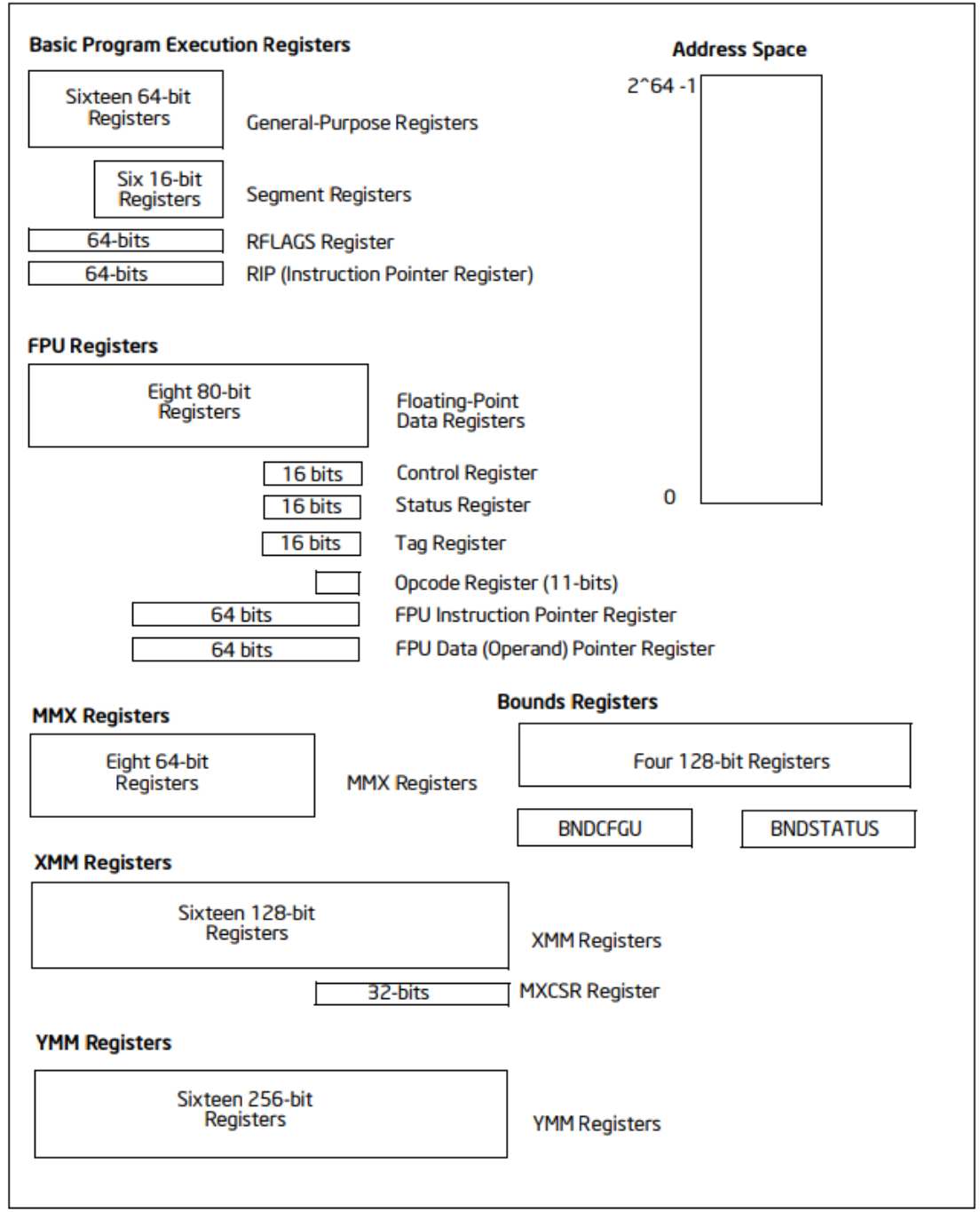


Figure 3-2. 64-Bit Mode Execution Environment

Data types in registers (or in memory)

- Byte (8bit), Word (16bit), Doubleword(32bit), Quadword (64bit)
  - Instructions may handle these as „signed“ (2-complement) or „unsigned“
- Addresses
- Floats with Half (1+9+6), Single (1+23+8, Double (1+11+52) or Extended (1+63[+1]+15) precision
    - Plus NaN and +/- Infine
- Special cases
    - BCD numbers packed/unpacked
    - Bit fields, Strings

- Memory (and registers) do not remember the type of data they contain. It is encoded in the instructions.

Overview of Instructions

- Basic
  - Data move instructions (MOV)
  - Arithmetic, bit-logical and shift
  - Conditional or unconditional branch
    - Conditional uses the FLAGS register set by previous instructions
    - Relative (offset) or absolute value to set IP
  - Stack operations (PUSH and POP) and subroutine calls CALL [in caller] and RET [in callee])
    - These use SP (a.k.a. RSP) register, which builds a FIFO (with mixed content of data and return addresses)
- Complex instructions
  - E.g: „Fused Multiply and Add“, „Inverse SQRT“, „REP/LOOP“+„MOVS“/“CMPS“, „FSIN/FCOS/FPTAN“, AES encryption, hyperthreading
- Best online resource: [x86 and amd64 instruction reference (felixcloutier.com)]

Timing of Instructions

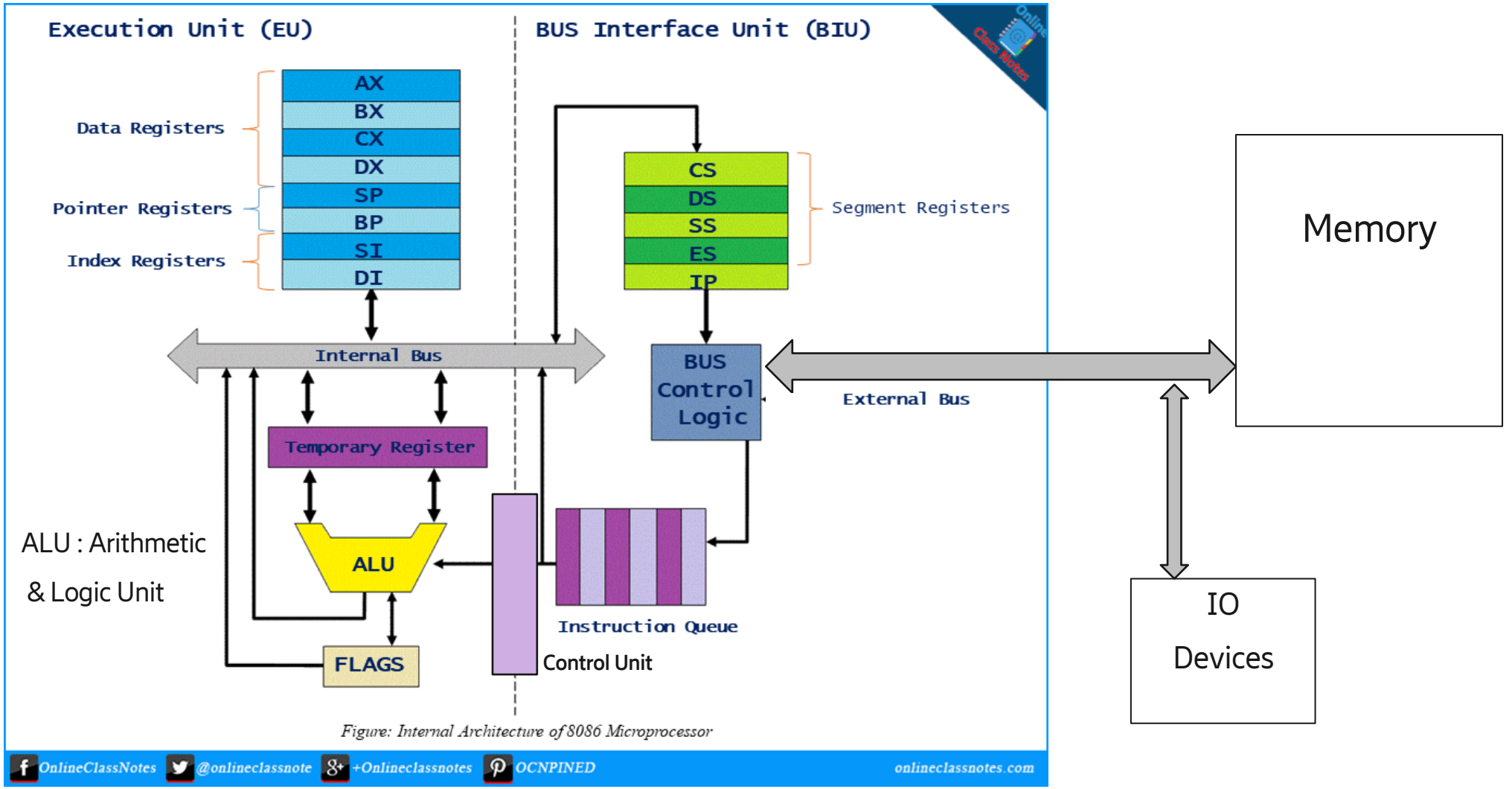
- Instructions take **>=1 clock cycles** if executed in sequence, (+ 4-6 may be executed in parallel)
  - Shortest is integer MOV/ADD/CMP register to register, or with immediate values
    - When memory is involved, this can become longer.
  - Multiplication is aceptable (4 cycles), also with floats.
  - Integer division is 15-20,
  - Floating point math functions > 50
  - Best online resource: https://www.agner.org/optimize/instruction_tables.pdf
- Instructions are processed in several phases, called a **pipeline**
  - Fetch instr -> decode -> read regs -> execute -> access memory -> store in regs
  - The pipeline may be delayed, e.g. if slow memory access occurs (e.g 20 ns -> CPU 60 cycles)
  - Dependencies (e.g add AX, 4; add BX, AX. also delay the pipeline
  - Branches cause problems, as a branch invalidates the pipeline (losing 15-20 clock cycles), unless it is correctly **predicted** -> this is provided by **branch predictors** and **branch target buffers**
- In-CPU autonomous optimizations, e.g.
  - Instruction reordering -> requires traching of dependencies.
  - Multiple (up to 4) execution units (e.g. ALU)

Assembly language

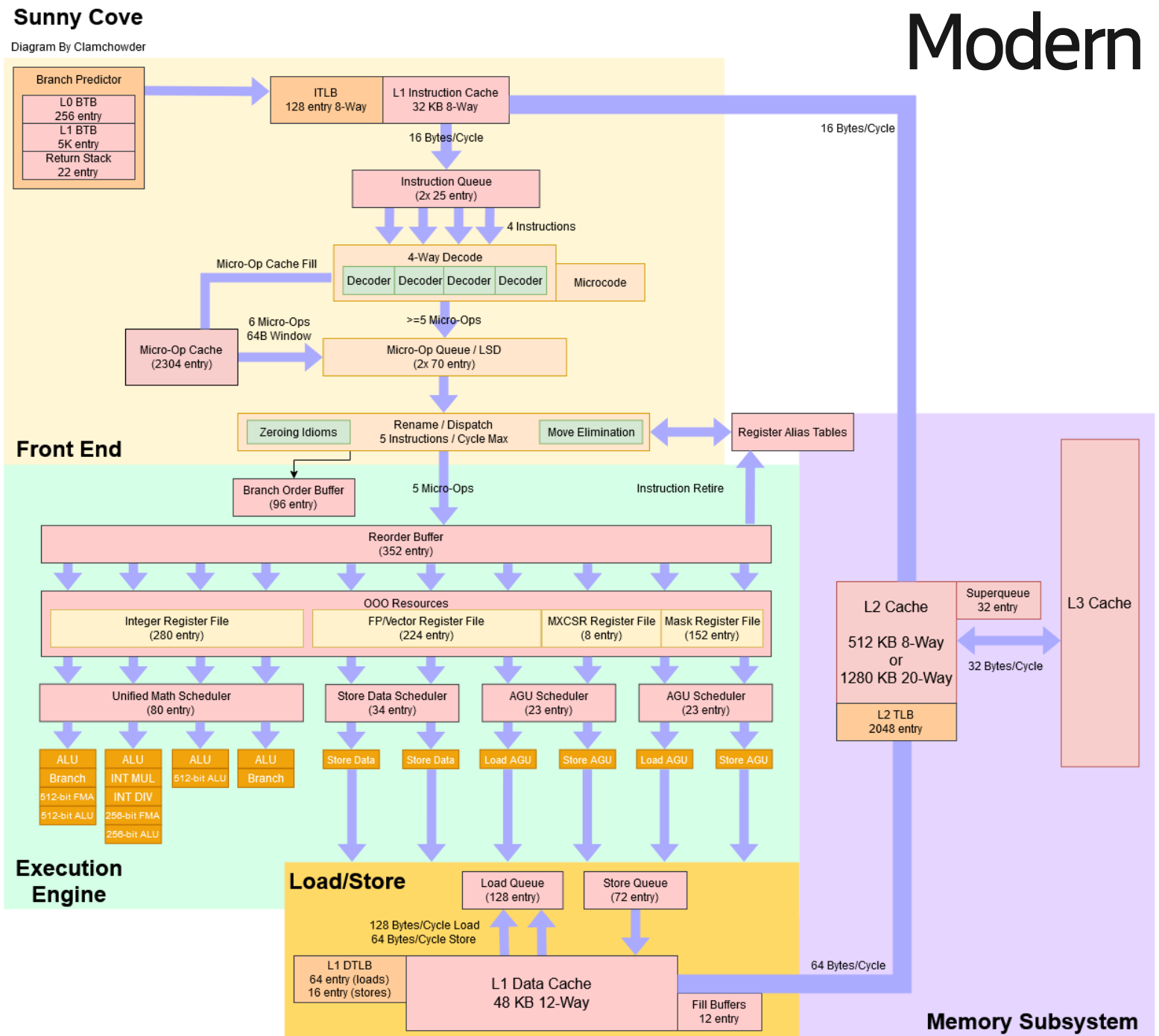
- „Mnemonic"-s assigned to instruction types
- Directly and easily convertible into machine code, no optimizations, etc.
  - Disassembly is also possible
- A syntax is defined for addressing and options
- Jumps use labels
  - No need to calculate offsets manually
  - Absolute addresses are resolved by linker „GNU ld)
- Machine code and assembly can be seen togethet with **objdump –D**

```
0000000000400430 <_init>:
  400430:       48 83 ec 08             sub    $0x8,%rsp
  400434:       48 8b 05 bd 0b 20 00    mov    0x200bbd(%rip),%rax        # 600ff8 <__gmon_start__>
  40043b:       48 85 c0                test   %rax,%rax
  40043e:       74 02                   je     400442 <_init+0x12>
  400440:       ff d0                   callq  *%rax
  400442:       48 83 c4 08             add    $0x8,%rsp
  400446:       c3                      retq
```

Intel 8086 architecture – 1978!!

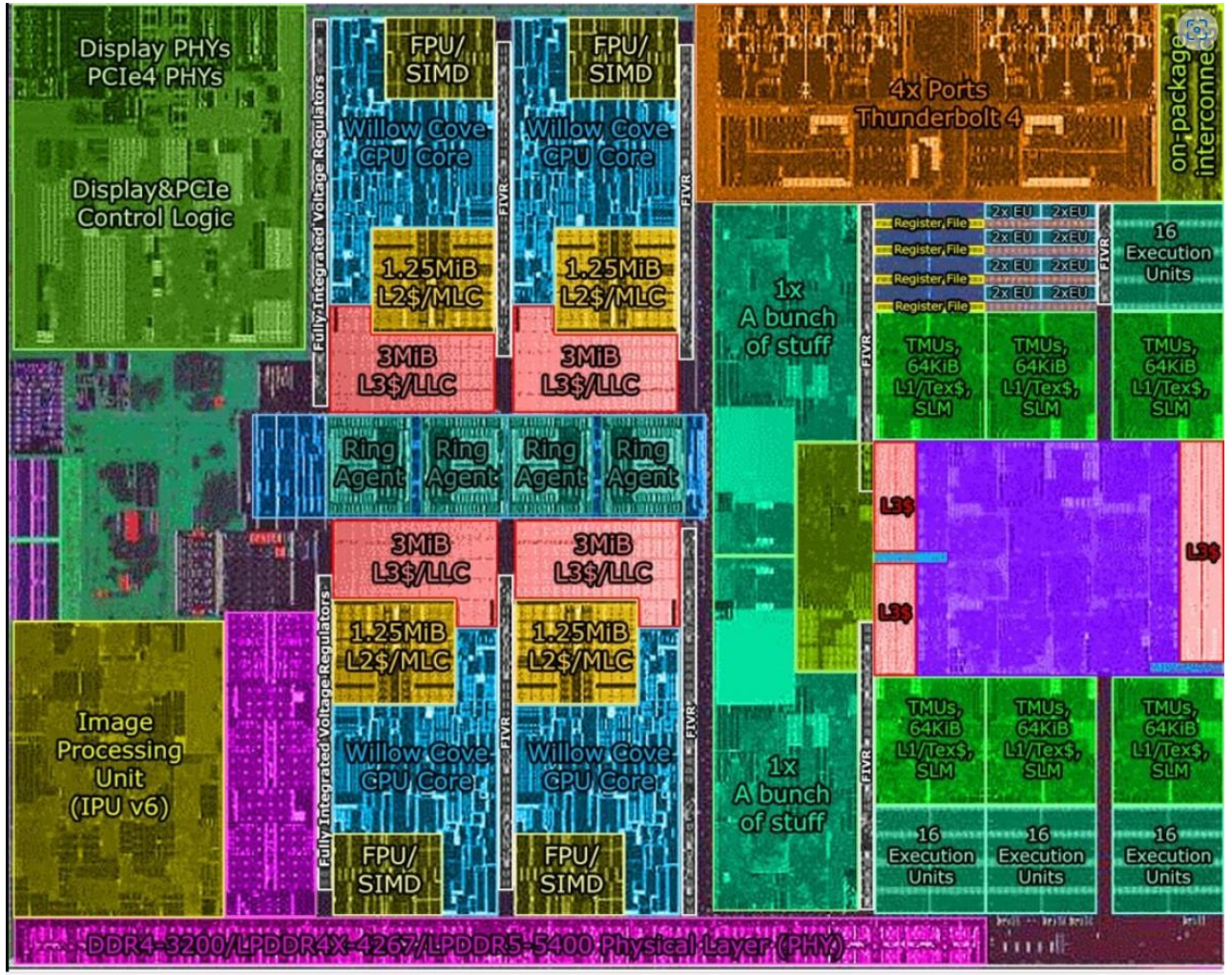


Modern Intel X64 Core (2019)



- Hundreds of 64 & 512 bit registers
- 3 layer caches
- Parallel inst decode and µOp cache
- 4x parallel ALU-s + 2 ‚AGU'
- Reordering
- Branch predictor
- Reordering

„Tiger Lake“ CPU with 4 x „Willow Cove“ Cores (10nm, 2021)



https://cdn.wccftech.com/wp-content/uploads/2020/08/Intel-Tiger-Lake-11th-Gen-Mobile-CPU-10nm-Die-Shot_2-1030x822.jpg

How to make use of these

- In 99% of cases compilers (e.g. modern C/C++ compilers) do a great job, i.e. they produce fast and efficinent code
- It is only worth optimizing a few little „hotspots" in code
  - Maybe there are no good candidate hotspots
  - Candidates are: spec data conversion (e.g. encoding/decoding, encryption), navigating in special collections/look-up tables, sort comparison methods, etc.
- Always give a chance to the C compiler: 1. use –O flags, 2. check if output is suboptimal, and 3. change only the smallest portion possible.
- Writing assembly files / complete methods from scratch is difficult; but **asm blocks** can be embedded into GNU C code:

```
int src = 1;
int dst;

asm ("mov %1, %0\n\t"
    "add $1, %0"
    : "=r" (dst)
    : "r" (src));

printf("%d\n", dst);
```

Inline assembly, and Assembly code - details

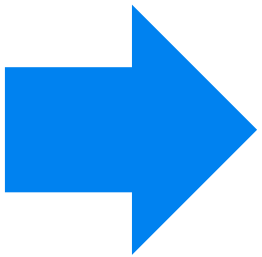
C code with asm block

```
int aaa = 59;
int ccc = 17;

asm volatile (

        "xor %%edx,%%edx\n\t"
        "mov $5,%%ebx\n\t"
        " add %1,%0\n\t"
        " imul %1\n\t"

        : "=r" (aaa)  // result will be a register, saved to a in the end
        : "r" (ccc)     // input data from variable c through a register
        : „edx", „ebx" // indicate regs used/overwritten in asm block
);
```



Compiler output

```
movl   $59, -44(%rbp)     // $59 is a constant 59
movl   $17, -36(%rbp)     //  rbp is the stack frame reg,
                          // -36 and -44 are the offsets

movl   -36(%rbp), %ecx    // select ecx  reg for $1, and load

xor %edx, %edx            // typical way to zero a reg
mov $5, %ebx              // load constant to reg
add %ecx,%eax             // eax is selected  for eax
imul %ebx                 // eax is used as src,
                             result goes to edx:eax

movl   %eax, -44(%rbp)    //  save to aaa var
```

Program for today: measure instruction latencies

- Generate code -> compile -> run several times -> summarize
- Parameters: <iteration_count> <IMUL-s per iteration> <second IMUL> <test_loops>

- E.g:
  - ./InlineAssembly.sh 1000000 10 X 10 -> 13 mSec
  - ./InlineAssembly.sh 1000000 110 X 10 -> 115 mSec

  - 1000000 x 100 IMULs: 100 mSec -> 1 IMUL: 1 nsec
  - CPU is 2.6 GHz -> 2.6 cycles / IMUL

Single instruction Assembly measured results

- Relative instruction times:

Instruction	Measured for 1M	Relative time	Remark
mov	73 µSec	0.26	Executed 4x parallel
add / sub	275 µSec	1	
add + add	280 µSec	1	Only for independent ops
mul / imul	1154 µSec	4.2	
div / idiv	7070 µSec	26	
div (after edx clear)	6270 µSec	23	Optimized in div µcode

- Check the CPU -> cat /proc/cpuinfo

  model name : Intel(R) Xeon(R) CPU E5-2697 v4 @ 2.30GHz - turbo: 3.50 Ghz.

  - cpu MHz measured with /proc/cpuinfo: 2.65 – 3.50 Ghz :
  - Clock cycle @3,50 Ghz: 285 psec, add instruction time: 286 psec

- Other CPU-s
  - Intel(R) Xeon(R) Gold 6240R CPU @ 2.40GHz -> 4.00 GHz for „turbo boost" -> down to 250 psec / clock cf. 273 psec
  - Intel Core i5-4210 CPU @1.70 GHz from 2014, with 270 GHz „Turbo Boost' -> down to 370 psec / clock cf. 378 ps / add

This is it for today - Mára ennyi!

- Next week, we investigate the caches and virtual memory in modern X64 processors