Computer Architecture

Design and Implementation

Jan Thorbecke

Contents

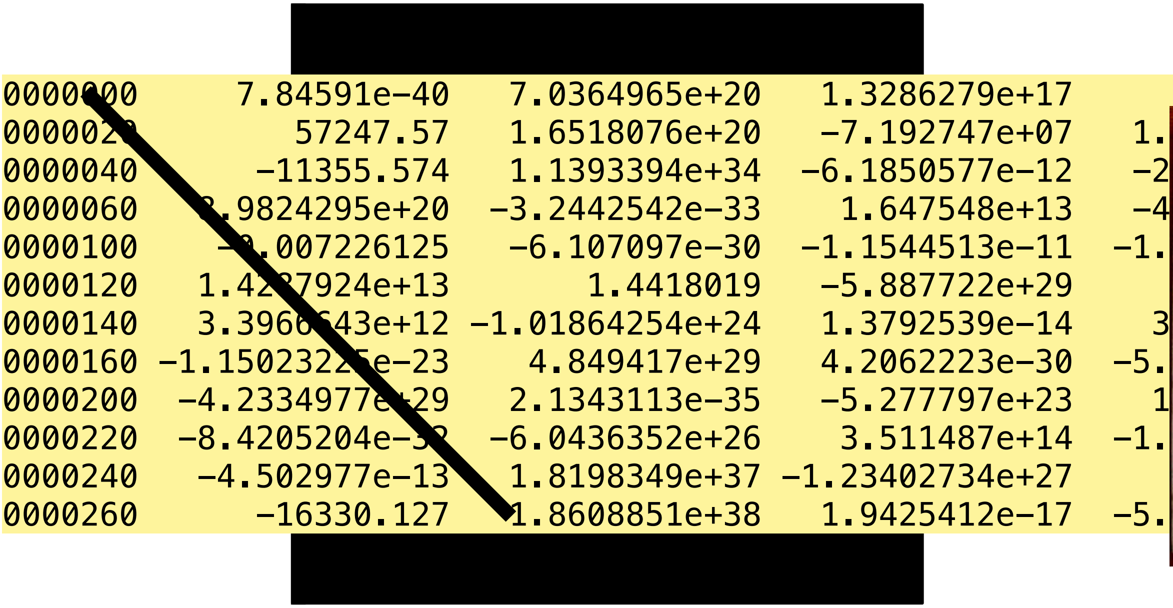
- Building Blocks
- Costs
- Memory hierarchy
- CPU
- Multi-core CPU
- Future hardware

computer operational view

$$f_1^- = \Theta_a R f_1^+,$$
$$f_1^{+\star} - f_{1,d}^{+\star} = -\Theta_b R f_1^{-\star}$$

input

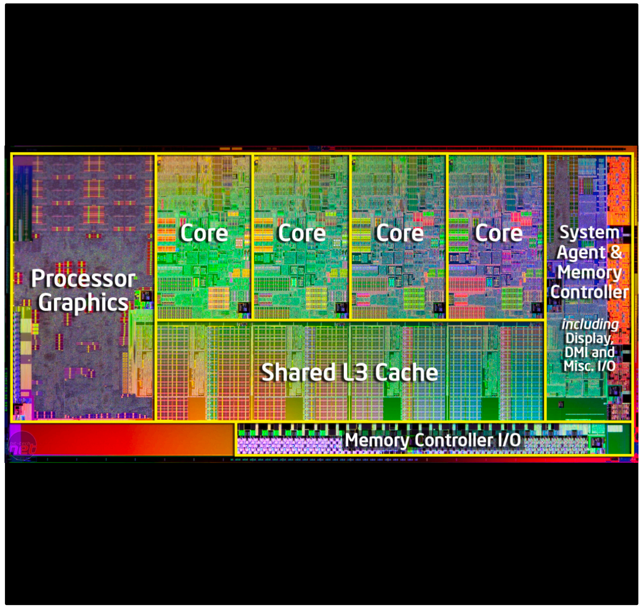
```
&inout
restartflag=`last'
output_length=1
output_offset=64
restart_length=1restart_le
restart_offset=64
logfile_outfreq=10
use_means=.false/
```



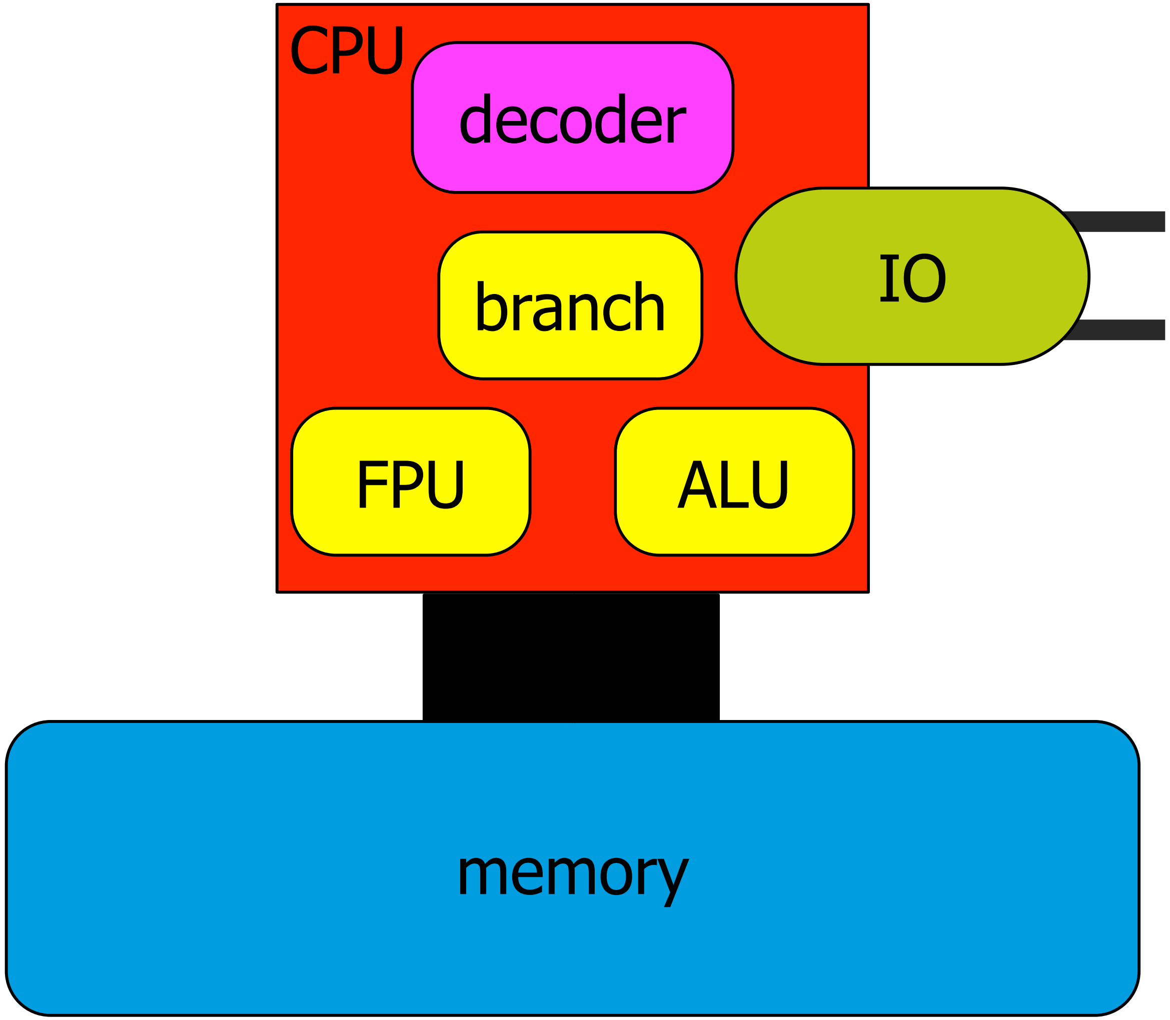
```
0000000     7.84591e-40   7.0364965e+20   1.3286279e+17        447423.97
0000020        57247.57   1.6518076e+20   -7.192747e+07   1.
0000040      -11355.574   1.1393394e+34  -6.1850577e-12   -2
0000060   2.9824295e+20  -3.2442542e-33    1.647548e+13   -4
0000100    -0.007226125   -6.107097e-30  -1.1544513e-11  -1.
0000120   1.4227924e+13       1.4418019   -5.887722e+29
0000140   3.3966643e+12 -1.01864254e+24   1.3792539e-14    3
0000160 -1.15023225e-23    4.849417e+29   4.2062223e-30  -5.
0000200  -4.2334977e+29   2.1343113e-35   -5.277797e+23    1
0000220  -8.4205204e-32  -6.0436352e+26    3.511487e+14  -1.
0000240   -4.502977e-13   1.8198349e+37 -1.23402734e+27
0000260      -16330.127   1.8608851e+38   1.9425412e-17  -5.
```

output

What’s in the black box?

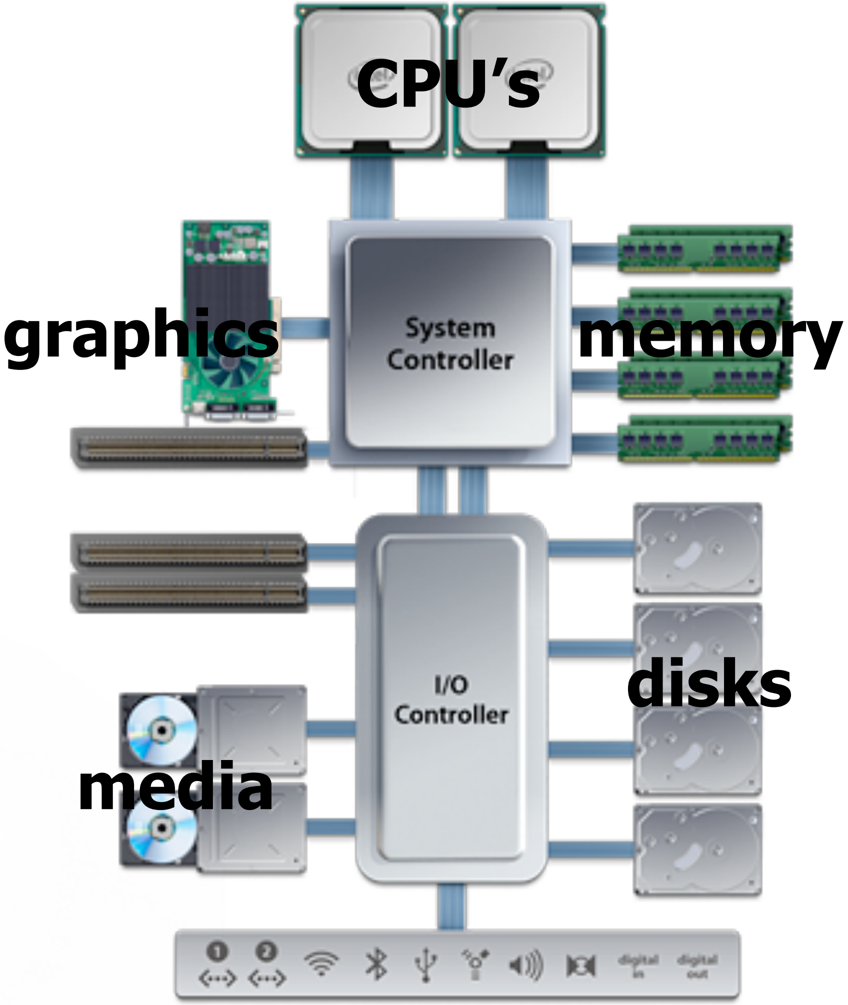


Lets build a compute system



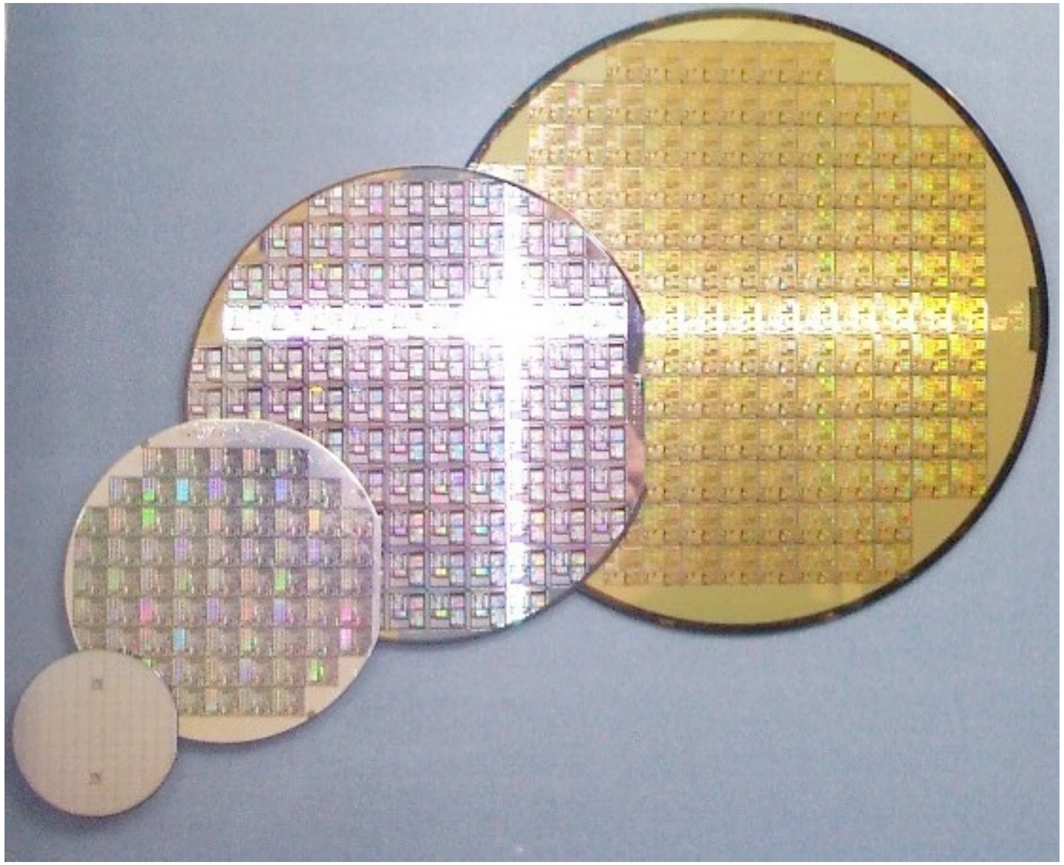
This is called a Von Neumann Architecture

A Typical Compute Server

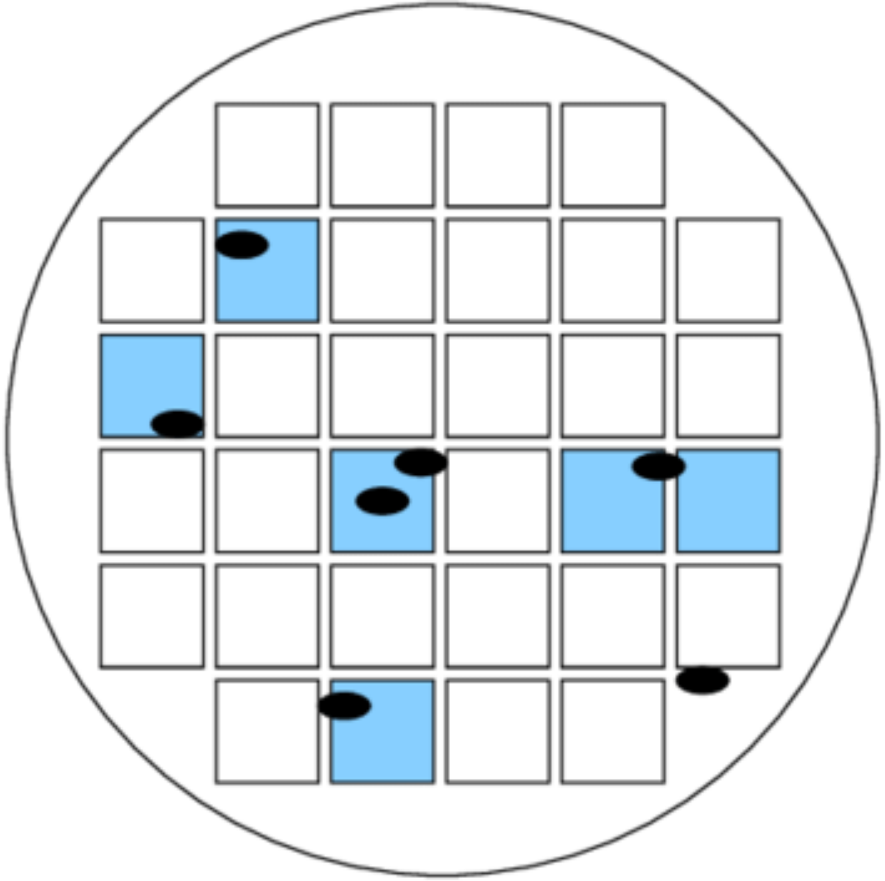


Wafers and dies:production of chips

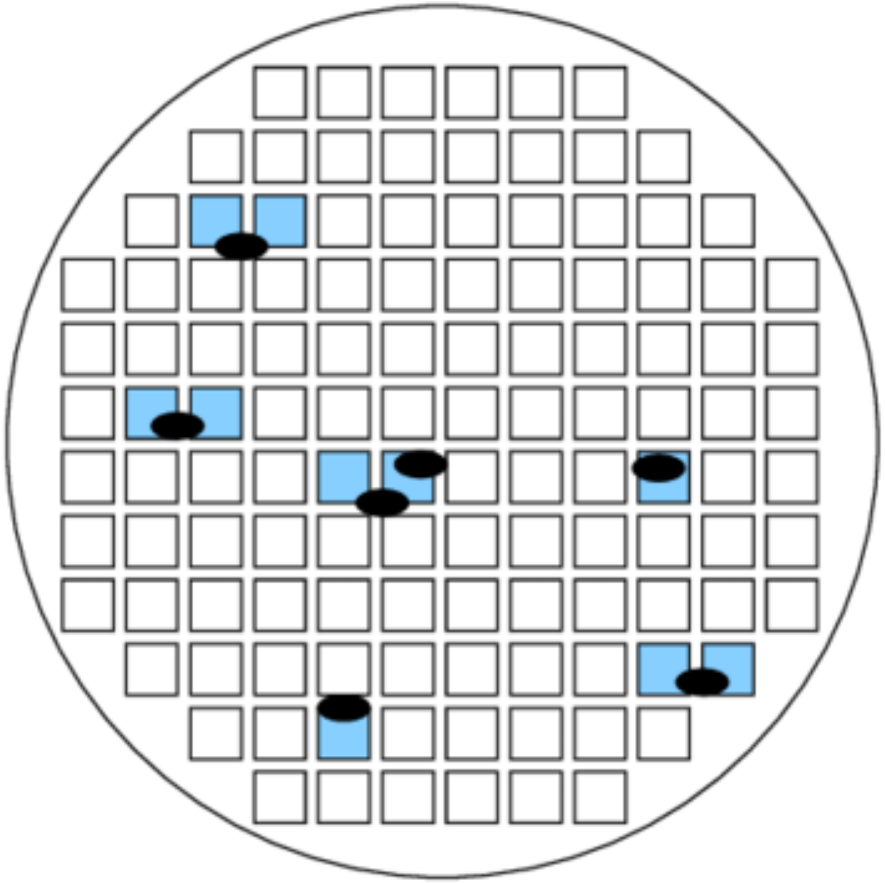
An entire wafer is produced and chopped into dies that undergo testing and packaging



Yield, defect, density and die size

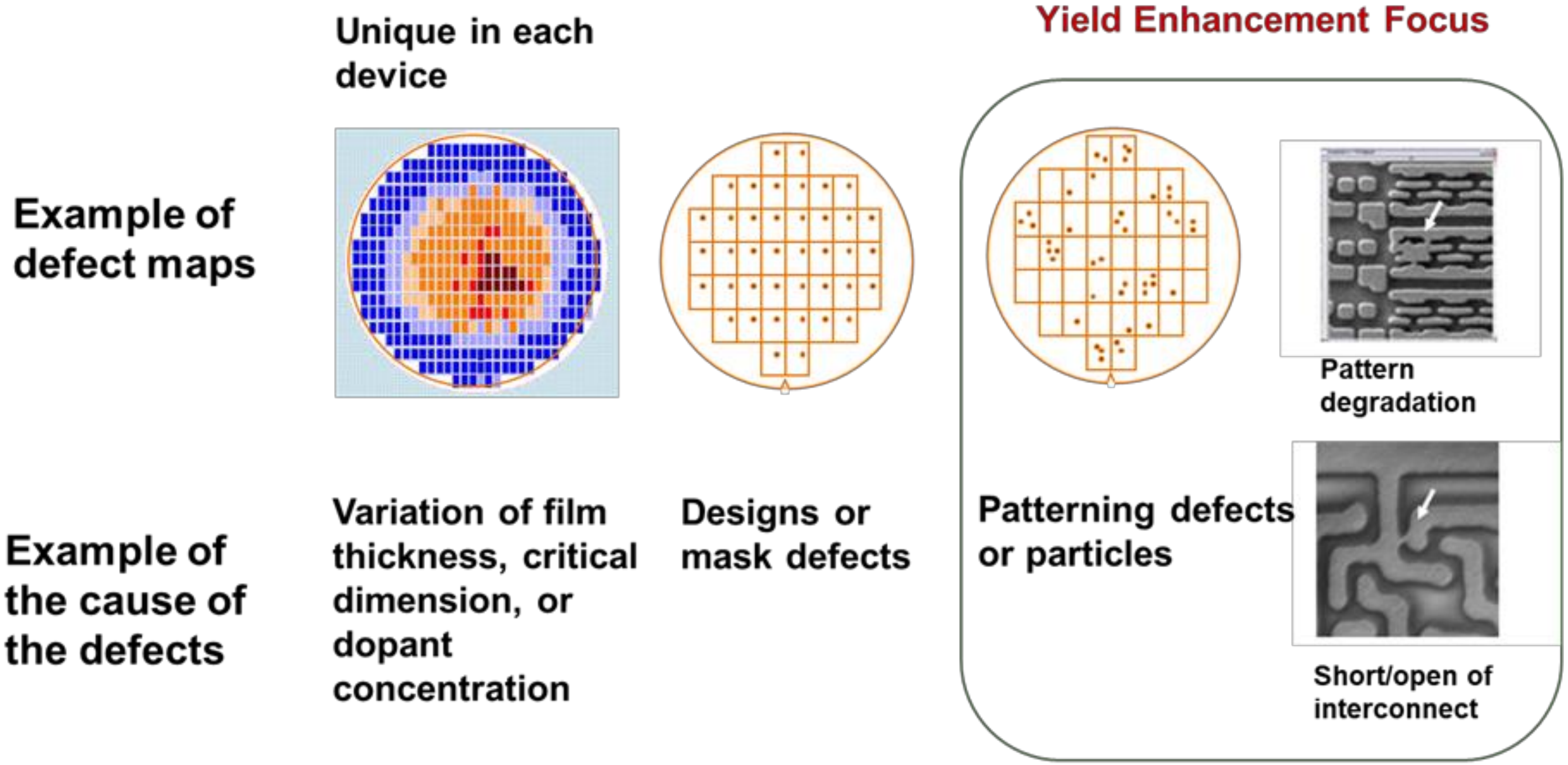


Wafer 1 : 32 gross die, 6 lost die, 81.25% yield



Wafer 2 : 120 gross die, 10 lost die, 91.6% yield

Die errors



Unique in each device

Yield Enhancement Focus

Example of defect maps

Pattern degradation

Example of the cause of the defects

Variation of film thickness, critical dimension, or dopant concentration

Designs or mask defects

Patterning defects or particles

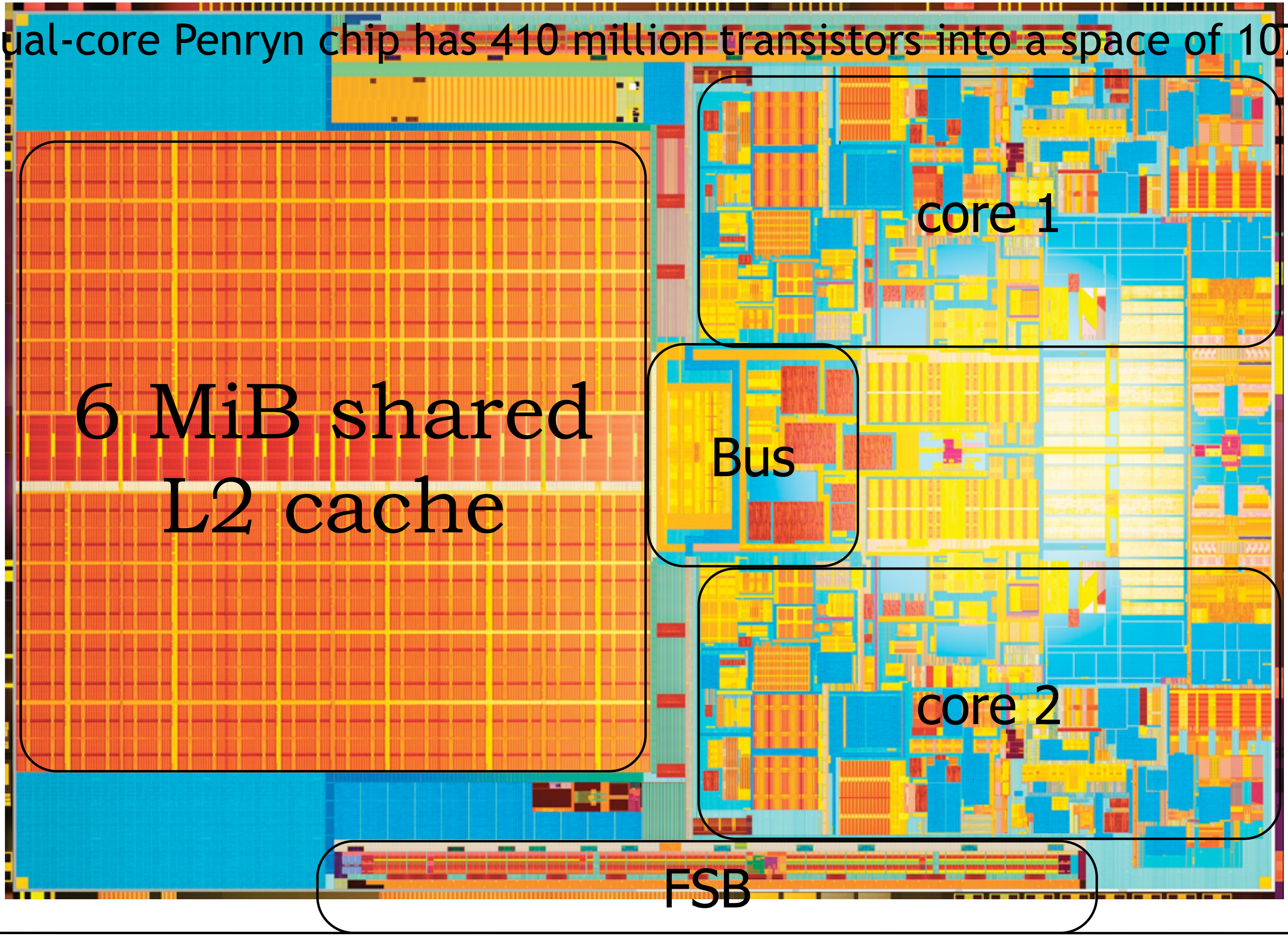
Short/open of interconnect

IEEE IRDS: INTERNATIONAL ROADMAP FOR DEVICES AND SYSTEMS. 2021 UPDATE “YIELD ENHANCEMENT”.

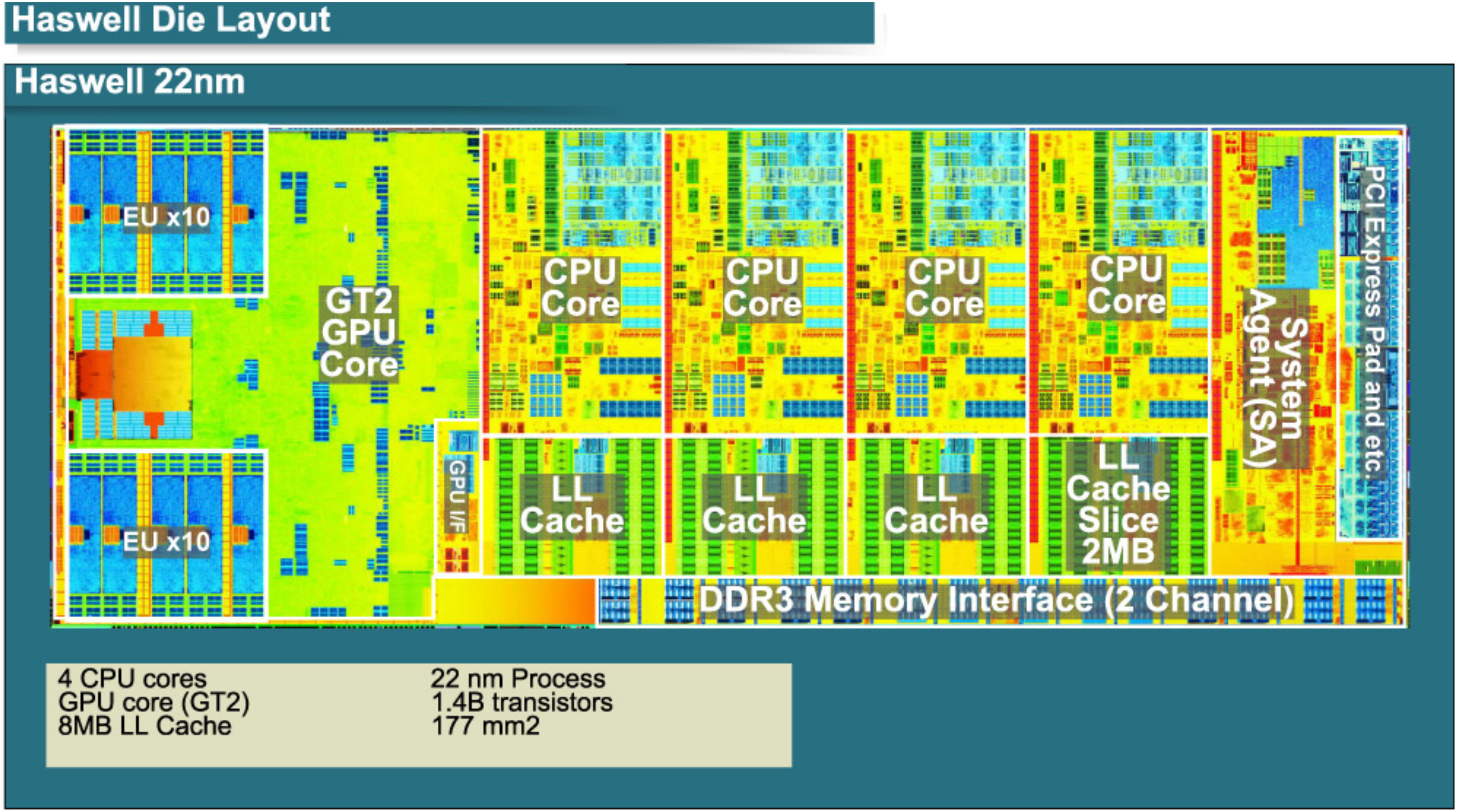
14 nm introduced in 2017

Intel Penryn dual-core die (45 nm)

Each dual-core Penryn chip has 410 million transistors into a space of 107 mm^2



Haswell quad core 22nm 2013



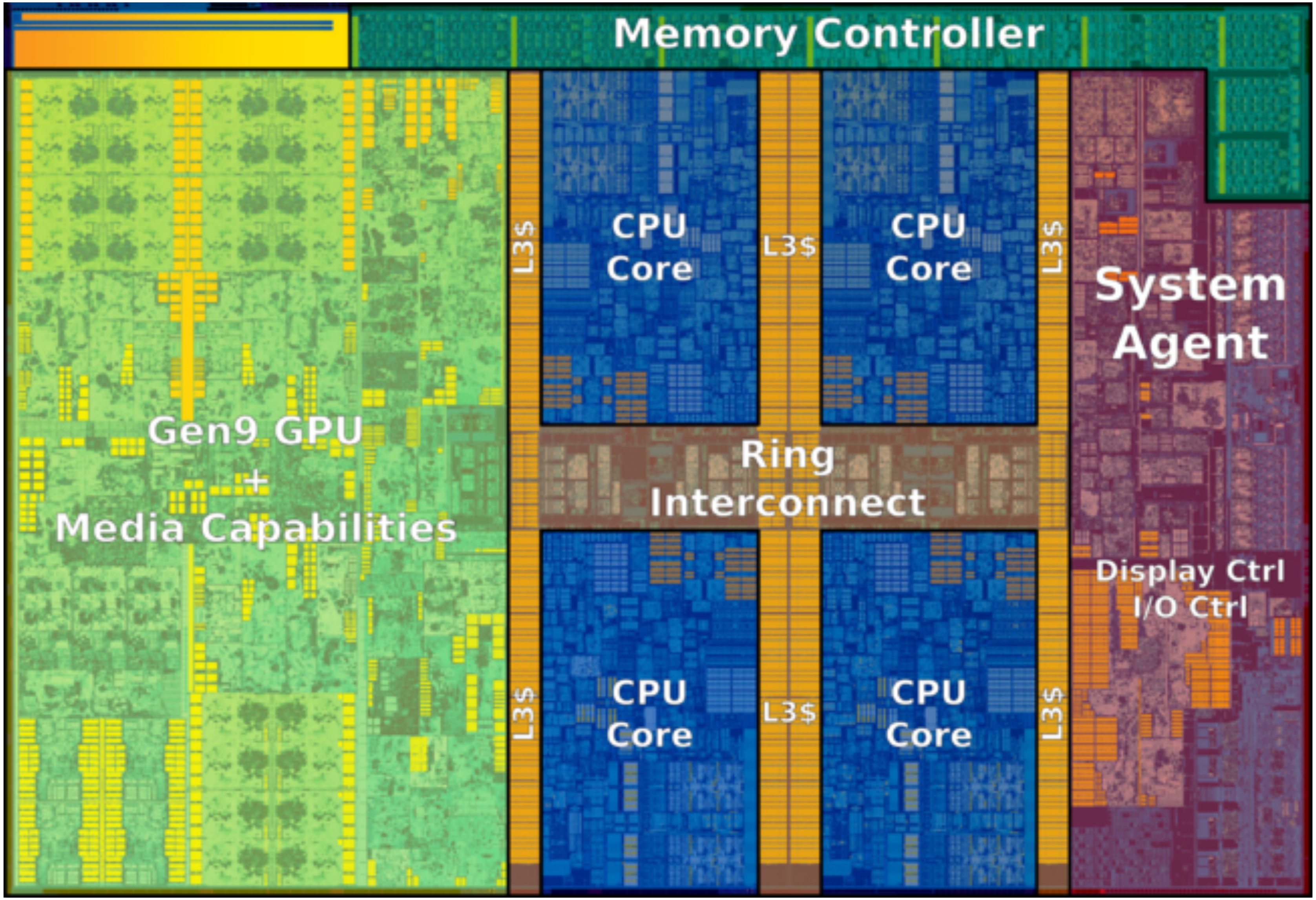
Haswell Die Layout

Haswell 22nm

4 CPU cores GPU core (GT2) 8MB LL Cache	22 nm Process 1.4B transistors 177 mm2

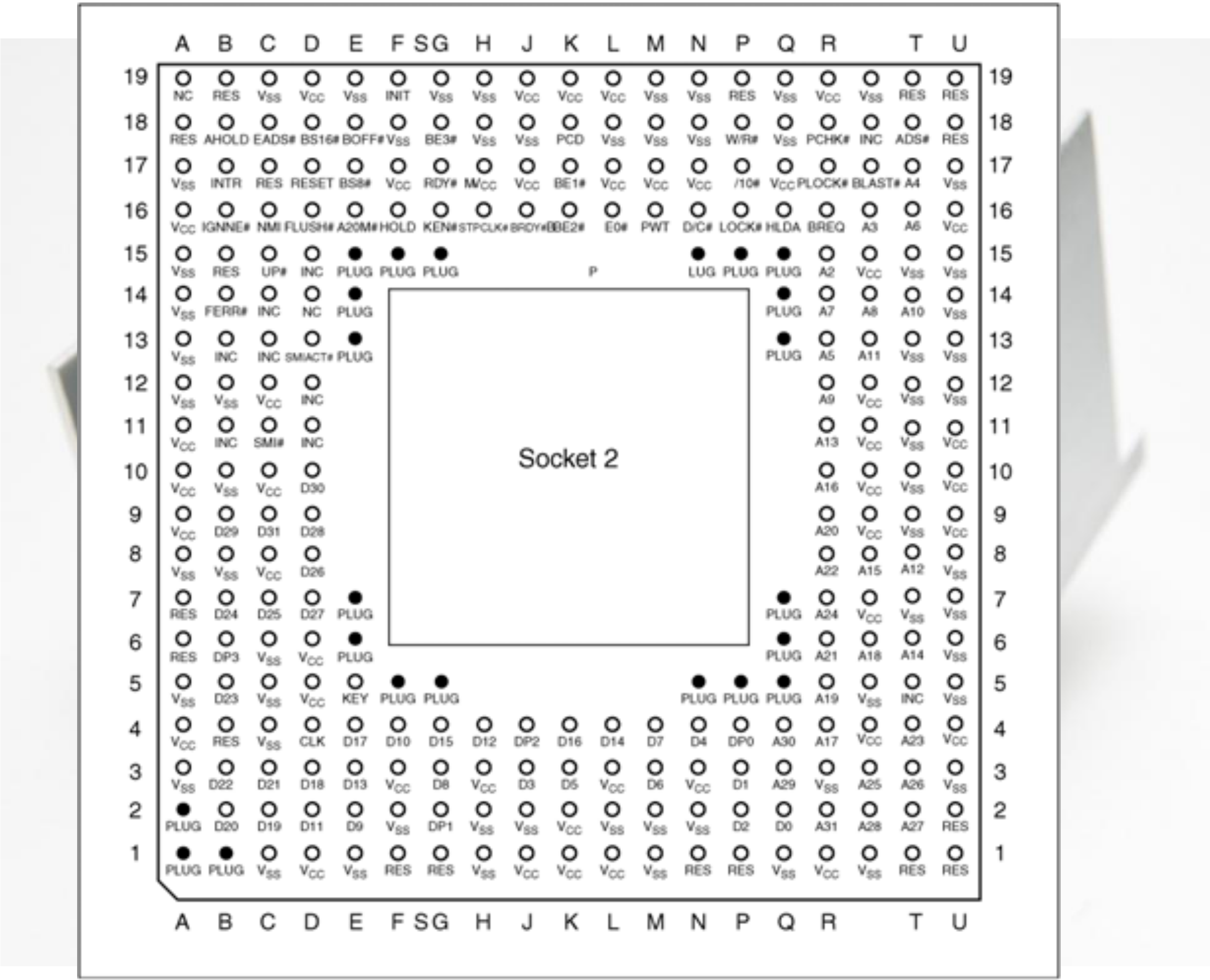
Copyright (c) 2013 Hiroshige Goto All rights reserved.

Skylake 14 nm 2018



Haswell Chip

Cascade Lake



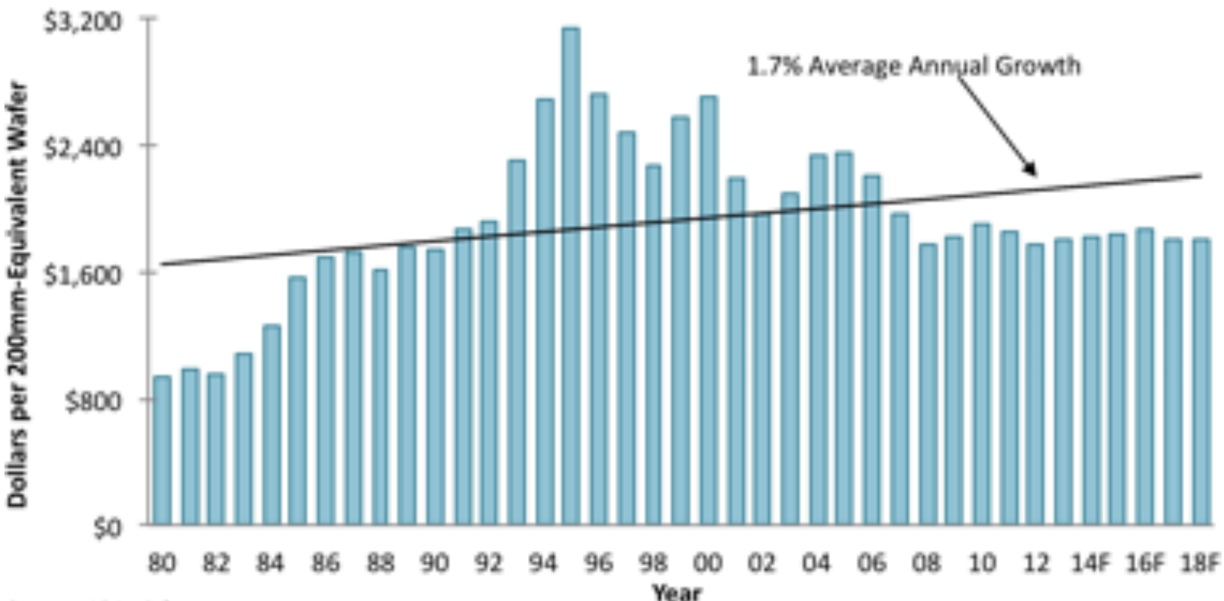
	A	B	C	D	E	F	SG	H	J	K	L	M	N	P	Q	R		T	U	
19	NC	RES	V_{SS}	V_{CC}	V_{SS}	INIT	V_{SS}	V_{SS}	V_{CC}	V_{CC}	V_{CC}	V_{SS}	V_{SS}	RES	V_{SS}	V_{CC}	V_{SS}	RES	RES	19
18	RES	AHOLD	EADS#	BS16#	BOFF#	V_{SS}	BE3#	V_{SS}	V_{SS}	PCD	V_{SS}	V_{SS}	V_{SS}	W/R#	V_{SS}	PCHK#	INC	ADS#	RES	18
17	V_{SS}	INTR	RES	RESET	BS8#	V_{CC}	RDY#	M/$_{CC}$	V_{CC}	BE1#	V_{CC}	V_{CC}	V_{CC}	/10#	V_{CC}	PLOCK#	BLAST#	A4	V_{SS}	17
16	V_{CC}	IGNNE#	NMI	FLUSH#	A20M#	HOLD	KEN#	STPCLK#	BRDY#	BBE2#	E0#	PWT	D/C#	LOCK#	HLDA	BREQ	A3	A6	V_{CC}	16
15	V_{SS}	RES	UP#	INC	PLUG	PLUG	PLUG			P			LUG	PLUG	PLUG	A2	V_{CC}	V_{SS}	V_{SS}	15
14	V_{SS}	FERR#	INC	NC	PLUG										PLUG	A7	A8	A10	V_{SS}	14
13	V_{SS}	INC	INC	SMIACT#	PLUG										PLUG	A5	A11	V_{SS}	V_{SS}	13
12	V_{SS}	V_{SS}	V_{CC}	INC												A9	V_{CC}	V_{SS}	V_{SS}	12
11	V_{CC}	INC	SMI#	INC												A13	V_{CC}	V_{SS}	V_{CC}	11
10	V_{CC}	V_{SS}	V_{CC}	D30												A16	V_{CC}	V_{SS}	V_{CC}	10
9	V_{CC}	D29	D31	D28												A20	V_{CC}	V_{SS}	V_{CC}	9
8	V_{SS}	V_{SS}	V_{CC}	D26												A22	A15	A12	V_{SS}	8
7	RES	D24	D25	D27	PLUG										PLUG	A24	V_{CC}	V_{SS}	V_{SS}	7
6	RES	DP3	V_{SS}	V_{CC}	PLUG										PLUG	A21	A18	A14	V_{SS}	6
5	V_{SS}	D23	V_{SS}	V_{CC}	KEY	PLUG	PLUG						PLUG	PLUG	PLUG	A19	V_{SS}	INC	V_{SS}	5
4	V_{CC}	RES	V_{SS}	CLK	D17	D10	D15	D12	DP2	D16	D14	D7	D4	DP0	A30	A17	V_{CC}	A23	V_{CC}	4
3	V_{SS}	D22	D21	D18	D13	V_{CC}	D8	V_{CC}	D3	D5	V_{CC}	D6	V_{CC}	D1	A29	V_{SS}	A25	A26	V_{SS}	3
2	PLUG	D20	D19	D11	D9	V_{SS}	DP1	V_{SS}	V_{SS}	V_{CC}	V_{SS}	V_{SS}	V_{SS}	D2	D0	A31	A28	A27	RES	2
1	PLUG	PLUG	V_{SS}	V_{CC}	V_{SS}	RES	RES	V_{SS}	V_{CC}	V_{CC}	V_{CC}	V_{SS}	RES	RES	V_{SS}	V_{CC}	V_{SS}	RES	RES	1
	A	B	C	D	E	F	SG	H	J	K	L	M	N	P	Q	R		T	U	

Socket 2

Integrated Circuit Cost Examples

- A 30 cm diameter wafer cost $200-$700 in 2015
- Such a wafer yields about 366 good 1 cm^2 dies and 1014 good 0.49 cm^2 dies (note the effect of area and yield)

IC Revenue per Wafer Start Trends (1980-2018)

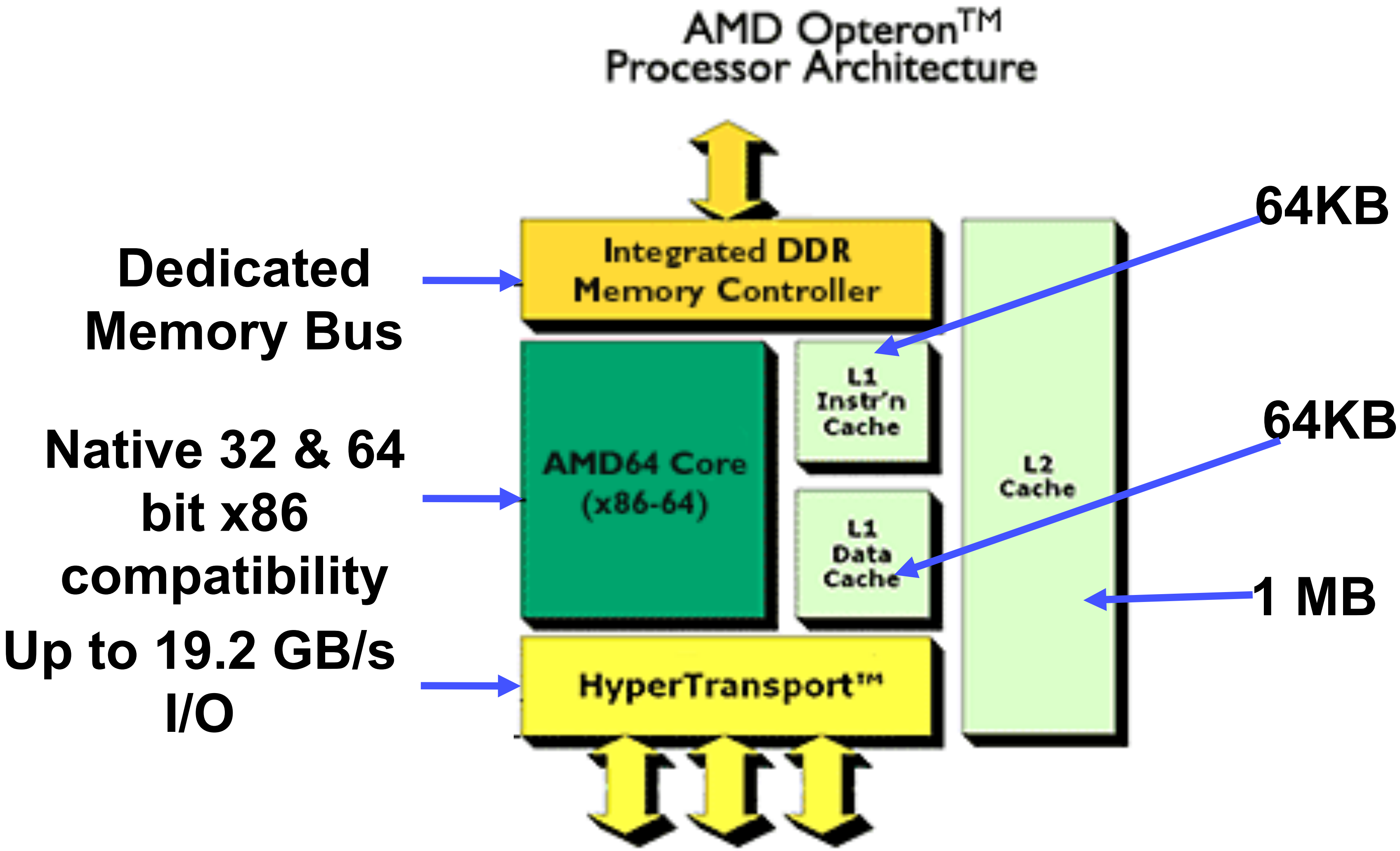


Source: IC Insights

Progress

Intel generation	manufacturing process	transistor count	die size
Pentium (P5)	0.80µm	3.1M	294 mm^2
Pentium 4	0.18µm	42M	217 mm^2
Nehalem 4-core	45 nm	731M	263 mm^2
SandyBridge 8-core	32 nm	2270M	434 mm^2
Haswell 18-core	22 nm	5560M	661 mm^2
Broadwell 22-core	14 nm	7200M	456 mm^2
Sapphire Rapids 56-core	10 nm	~20000M	1600 mm^2

The AMD Opteron Processor



Dedicated Memory Bus

Native 32 & 64 bit x86 compatibility

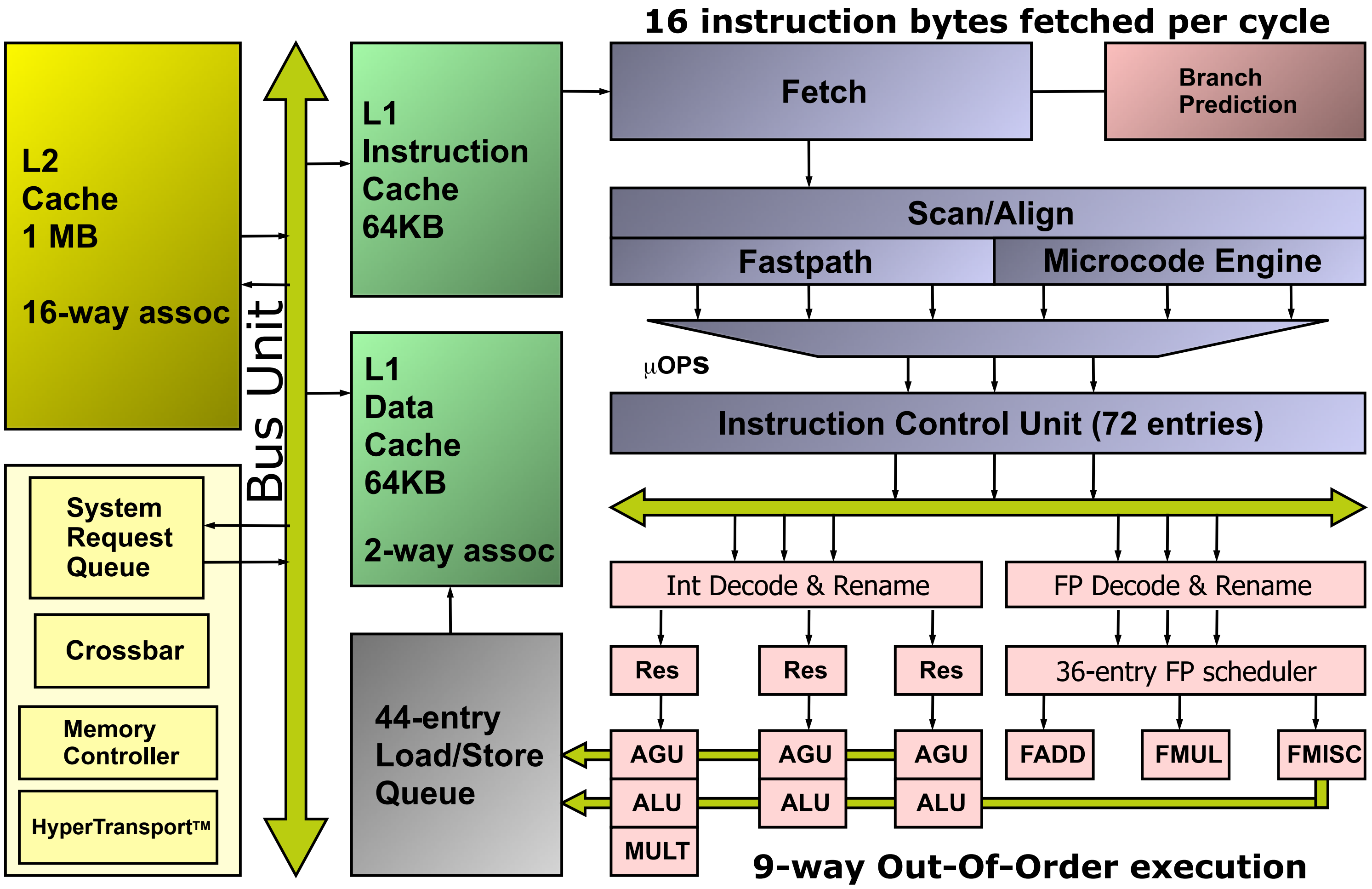
Up to 19.2 GB/s I/O

64KB

64KB

1 MB

AMD Opteron Processor



- 36 entry FPU instruction scheduler
- 64-bit/80-bit FP Realized throughput (1 Mul + 1 Add)/cycle: 1.9 FLOPs/cycle
- 32-bit FP Realized throughput (2 Mul + 2 Add)/cycle: 3.4+ FLOPs/cycle

Instruction sets

- Instructions to tell the hardware what to do.
- Brief overview of instructions before we dive deeper into the hardware.

Instruction sets

- ISA: An instruction set, or instruction set architecture (ISA), is the part of the compute hitecture related to programming, including the native data types, ins registers, addressing modes, memory architecture, interrupt and ex andling, and external I/O. An ISA includes a specification of the se des (machine language), the native commands implemented by a particular n.
- **Categories of ISA**
  - CISC (X86)
  - RISC
  - VLIW
  - MISC
  - EPIC
  - vector processor
  - SIMD
  - Flynn's Taxonomy
  - orthogonal instruction set

Don’t worry we are not going to program in it.

ISA Considerations

- Code size
    - Long instructions take more time to fetch
    - Longer instructions require a larger memory
        - Important in small devices, e.g., cell phones
- Number of instructions (IC)
    - Reducing IC reduce execution time
        - At a given CPI (clocks cycles per instruction) and frequency
- Code "simplicity"
    - Simple HW implementation
        - Higher frequency and lower power
    - Code optimization can better be applied to "simple code"

CISC

- **Definition**: Pronounced "sisk" and standing for Complex Instruction Set Computer, is a Microprocessor Architecture that aims at achieving complex operations with single instructions and favors the richness of the instruction set (typically as many as 200 unique instructions) over the speed with which individual instructions are executed.

Why should I know about CISC?

- Today's computers still use processors which are based on CISC designs
- It has been a prominent architecture since 1978 (x86)
  - x86_64: 64 bit version of the x86 instruction set

RISC

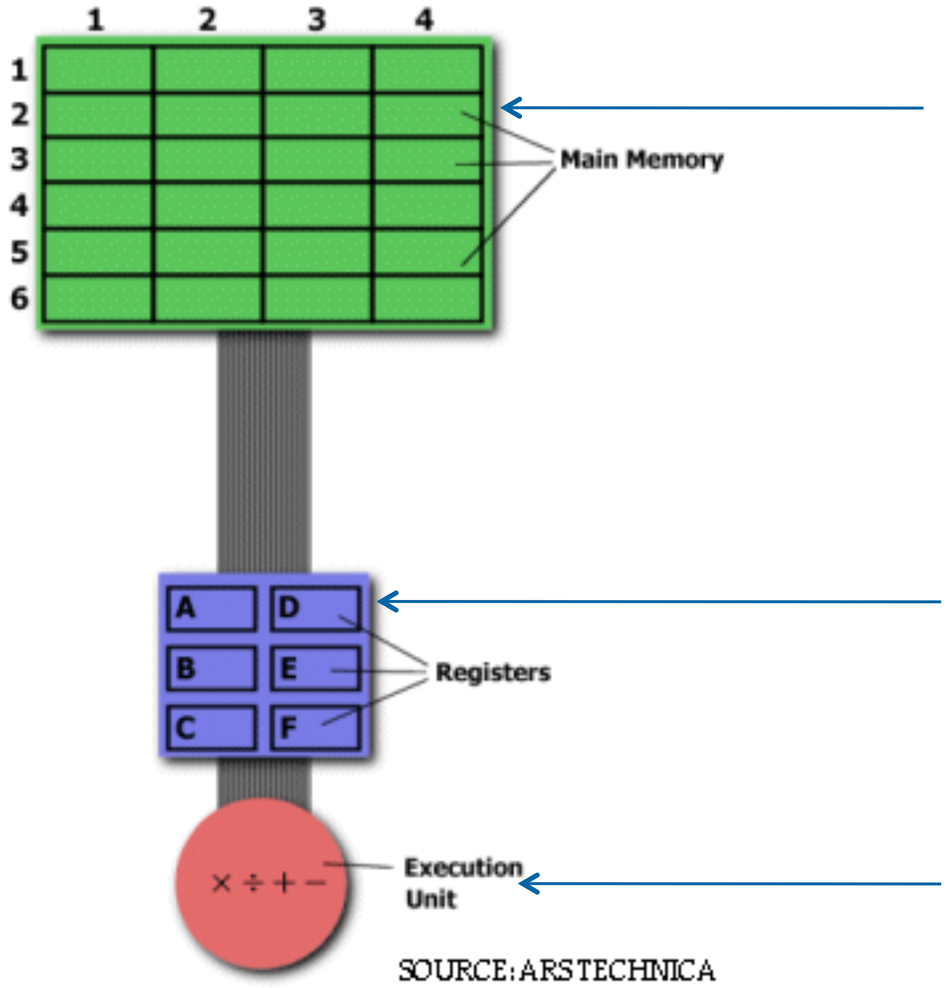
- RISC - Reduced Instruction Set Computer
    - The idea: simple instructions enable fast hardware
    - load-store architecture

- Characteristic
    - A small instruction set, with only a few instructions formats
    - Simple instructions
        - execute simple tasks
        - Most of them require a single cycle (with pipeline)

    - ALU operations on registers only
        - Memory is accessed using Load and Store instructions only
        - Many orthogonal registers
    - Fixed length instructions

- Examples: MIPS™, Sparc™, Alpha™, Power™

RISC/CISC Example



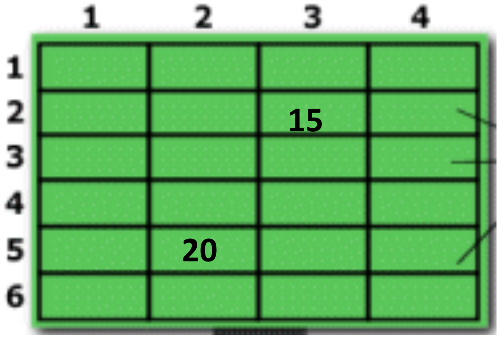
SOURCE: ARSTECHNICA

Main Memory

General Purpose Registers

ALU

Consider following task of Multiplication



	1	2	3	4
1				
2			15	
3				
4				
5		20		
6				

Operands:

M[2:3] = operand 1 (15)
M[5:2] = operand 2 (20)

Task : Multiplication

Result: M[2:3] <= result

The CISC Approach

- Instruction :

    MULT 2:3, 5:2

Operations:

- Loads the two operands into separate registers
- Multiplies the operands in the execution unit
- Then stores the product in the some temporary register
- Stores value back to memory location 2:3

- MULT is what is known as a "complex instruction."
- Operates directly on the computer's memory banks
- Does not require the programmer to explicitly call any loading or storing functions.
- closely resembles a command in a higher level language.

    e.g. a ‘C’ statement "a = a * b."

The RISC Approach

- Instructions :

```
LW          A, 2:3
LW          B, 5:2
MULT        A, B
SW 2:3      A
```

- Operations:
- Load operand1 into register A
- Load operand2 into register B
- Multiply the operands in the execution unit and store result in A
- Store value of A back to memory location 2:3

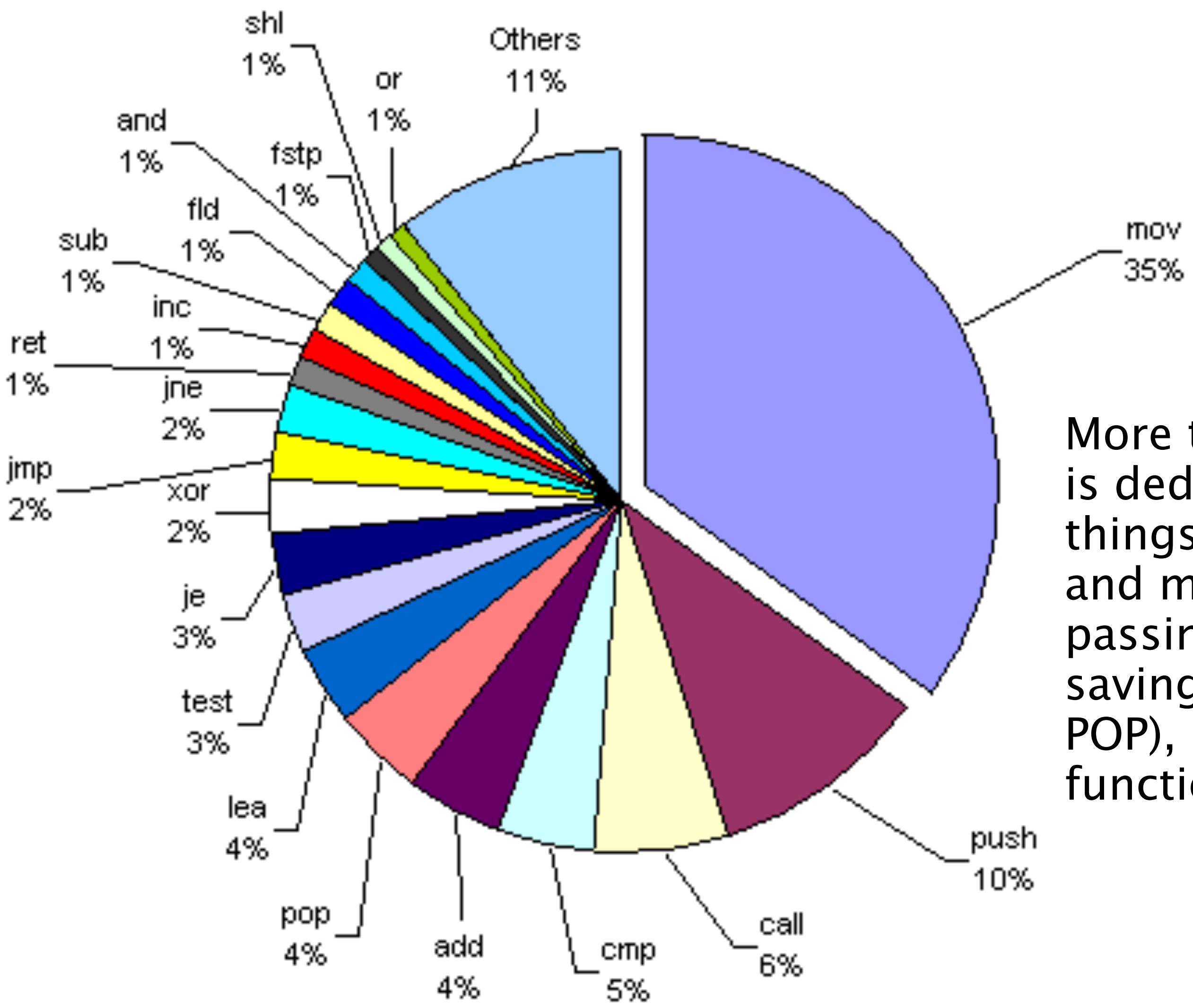
- These set of Instructions is known as a “Reduced Instructions."
- Cannot Operate directly on the computer's memory banks
- Requires the programmer to explicitly call any loading or storing functions.
- RISC processors only use simple instructions that can be executed within one clock cycle

Developments

- The terms RISC and CISC have become less meaningful with the continued evolution of both CISC and RISC designs and implementations.

- Modern x86 processors also decode and split more complex instructions into a series of smaller internal "micro-operations" which can thereby be executed in a pipelined (parallel) fashion, thus achieving high performance on a much larger subset of instructions.

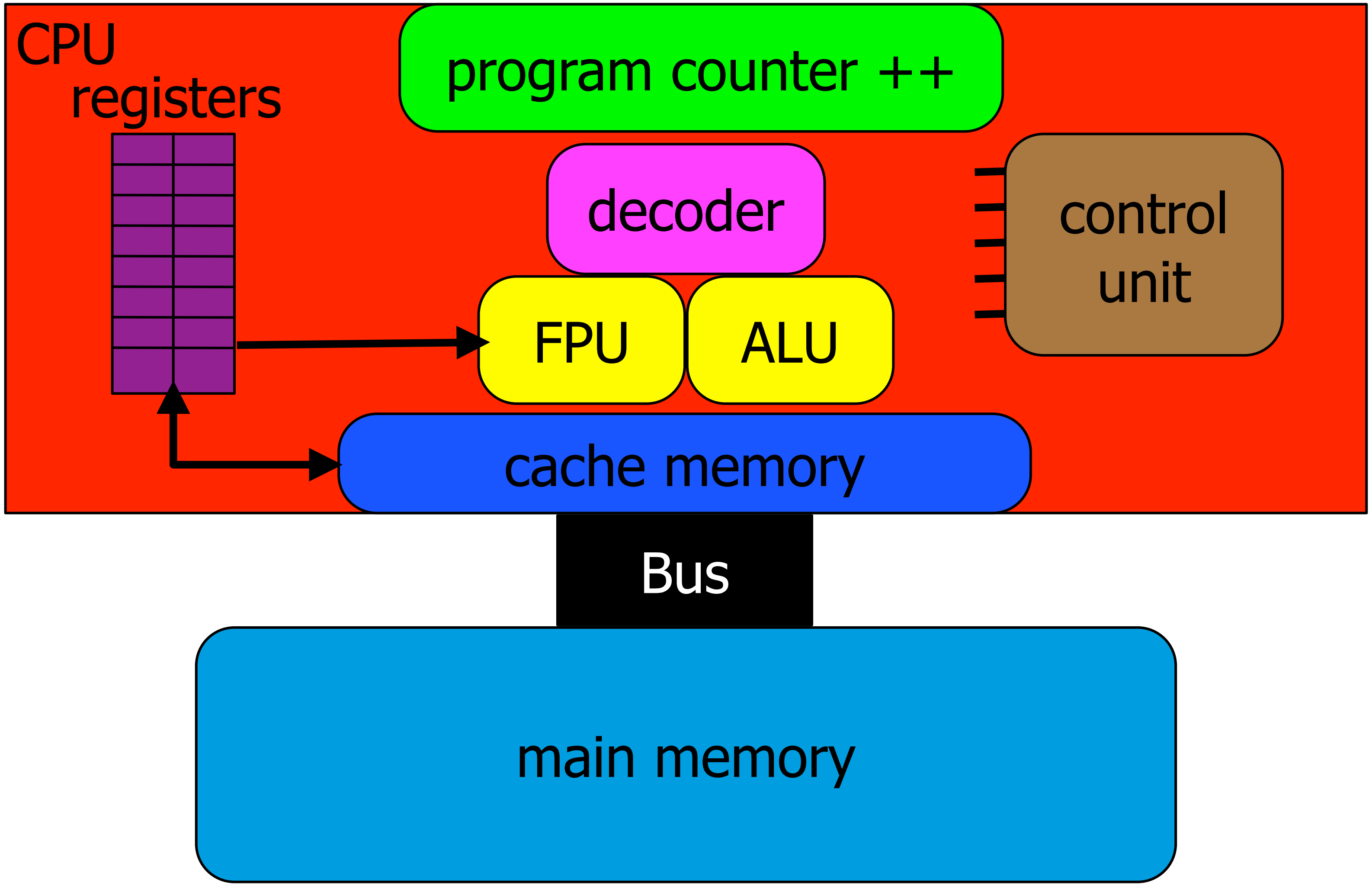
Top 20 instructions of x86



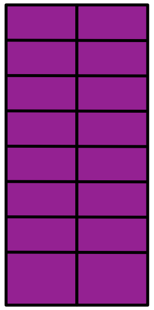
More than 50% of all code is dedicated to moving things between registers and memory (MOV), passing arguments, saving registers (PUSH, POP), and calling functions (CALL).

Computer Architecture

A Simple Computer Architecture



Register Array



- All modern CPU’s have an array of registers
  - usually at least 32 general purpose registers (128 bit wide)
  - frequently some registers have dedicated use
- Characteristics of registers
  - usually contain one computer word
  - can be accessed in one CPU cycle
- Functions of registers
  - serve as source of operands
  - serve as destination of results
  - temporarily store intermediate results
  - serve as index registers to access arrays (stack pointer)
- Specialized registers
  - floating point registers
  - store constants ….frequently used values

The Program Counter (PC)

program counter ++

- stores address of next instruction to execute
- must be incremented after each instruction
- may be changed by function call or jump
- controls flow of program execution

Arithmetic Logic Unit (ALU)

ALU

- performs arithmetic and logical functions
- works on integers
- add, subtract, multiply, divide, complement, shift…etc.
- function performed is determined by the control signals received
- will have input and output latches to hold operands and results

Floating Point Unit (FPU)

FPU

- mainly for addition, multiplication and sometimes division
- works on floating point numbers
- higher order functions like divide, sqrt are emulated in software: for example using a series expansion approximation based on the basic operations add and mul
- has its own set of registers it can use
- SSE and AVX instructions (SIMD) can do more than one operation in a clock cycle

Memory Buffer (Cache)

cache memory

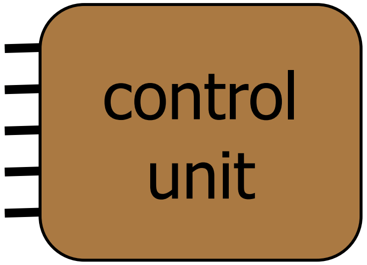
- A distinct memory positioned between the CPU and Main Memory
- holds values to be transferred between main memory and the CPU
- both data or instructions can be stored in cache
- values to be written to memory
- machines are capable of transferring more than a single word called a cache line; usually 8 bytes (64 bits)

Decoder

decoder

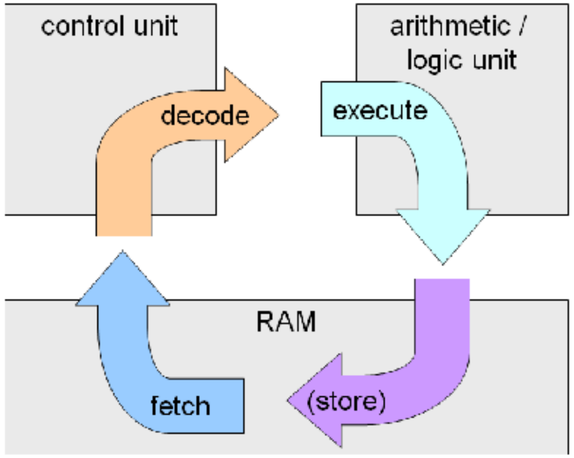
- Decode the instructions that are sent to the microprocessor.
- It can decode and optimise the order of instructions before it sends them to the execution unit to be run.

Control Unit



- provides control signals necessary to control the hardware of the CPU
- control signals are needed to control functions of various hardware units and to direct the flow of information within the CPU.
- Directs the operation of the processor: It tells the computer's memory, arithmetic/logic unit and input and output devices how to respond to a program's instructions

The Fetch-Execute Cycle



- The steps that the **control unit** carries out in executing a program are:

  (1) Fetch the next instruction to be executed from memory.

  (2) Decode the opcode.

  (3) Read operand(s) from main memory, if any.

  (4) Execute the instruction and store results.

  (5) Go to step 1.

  This is known as the **fetch-execute** cycle.

Memory Unit

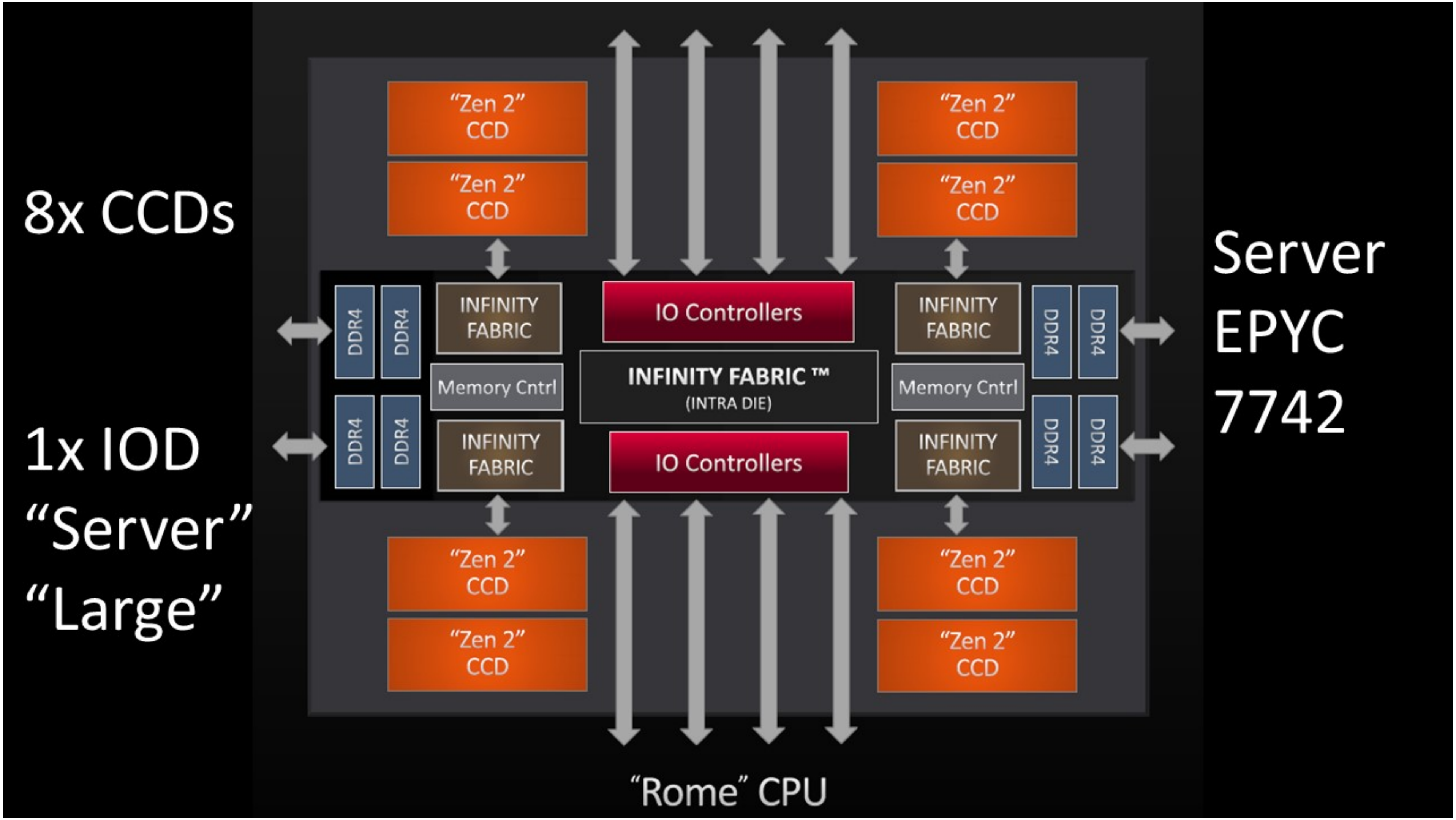
main memory

- Main Memory
- used to store programs (instructions) and data
- volatile: requires power to maintain the stored information
- usually uses DRAM… Dynamic Random Access Memory
- most memory is byte addressable
  - can retrieve a single byte per memory access
  - can be organised to access a full word or multiple words per access.

Bus structure

- CPU bus structure
- a bus is an ‘path’ connecting the various functional units within the CPU
- capable of transmitting one entire word in parallel
- will consist of one word length of ‘wires’ or data paths
- the CPU will have multiple buses to improve the information transfer options within the CPU to maximize the flexibility and parallelism of the system

Memory direct connected



8x CCDs

1x IOD
“Server”
“Large”

Server
EPYC
7742

“Rome” CPU

Memory Design

- Types of memory
  - DRAM
  - SRAM
- Access speed
  - latency
  - bandwidth
- Amount of storage
- Fabrication costs

Memory differences

- Volatile: fast access but non permanent
  - static RAM
  - dynamic RAM (must be refreshed regularly)
- Permanent writable: (very) slow write access but permanent
  - magnetic (hard-drive, magnetic tape, etc)
  - SSD (Solid State Drive)
  - FLASH (page write access)
  - EEPROM
  - CD, DVD
- Permanent non writable
  - ROM
  - PROM

Memory Types volatile

- SDRAM: Synchronous Dynamic-RAM used for main memory
- SRAM: Static-RAM used for cache
- Registers: direct accessible

Memory locations are arranged linearly in consecutive order. Each numbered locations corresponds to a **word**. The unique number that identifies each word is referred to as its address.

DRAM memory cell

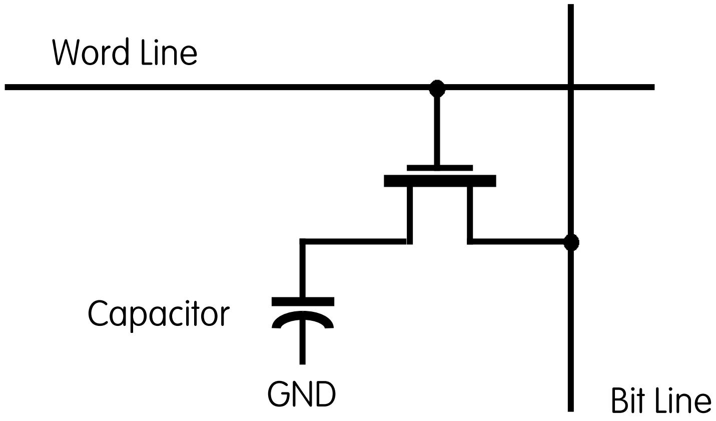
- Word line selects cell for reading or writing

  To write, the bit line is charged with logic 1 or 0

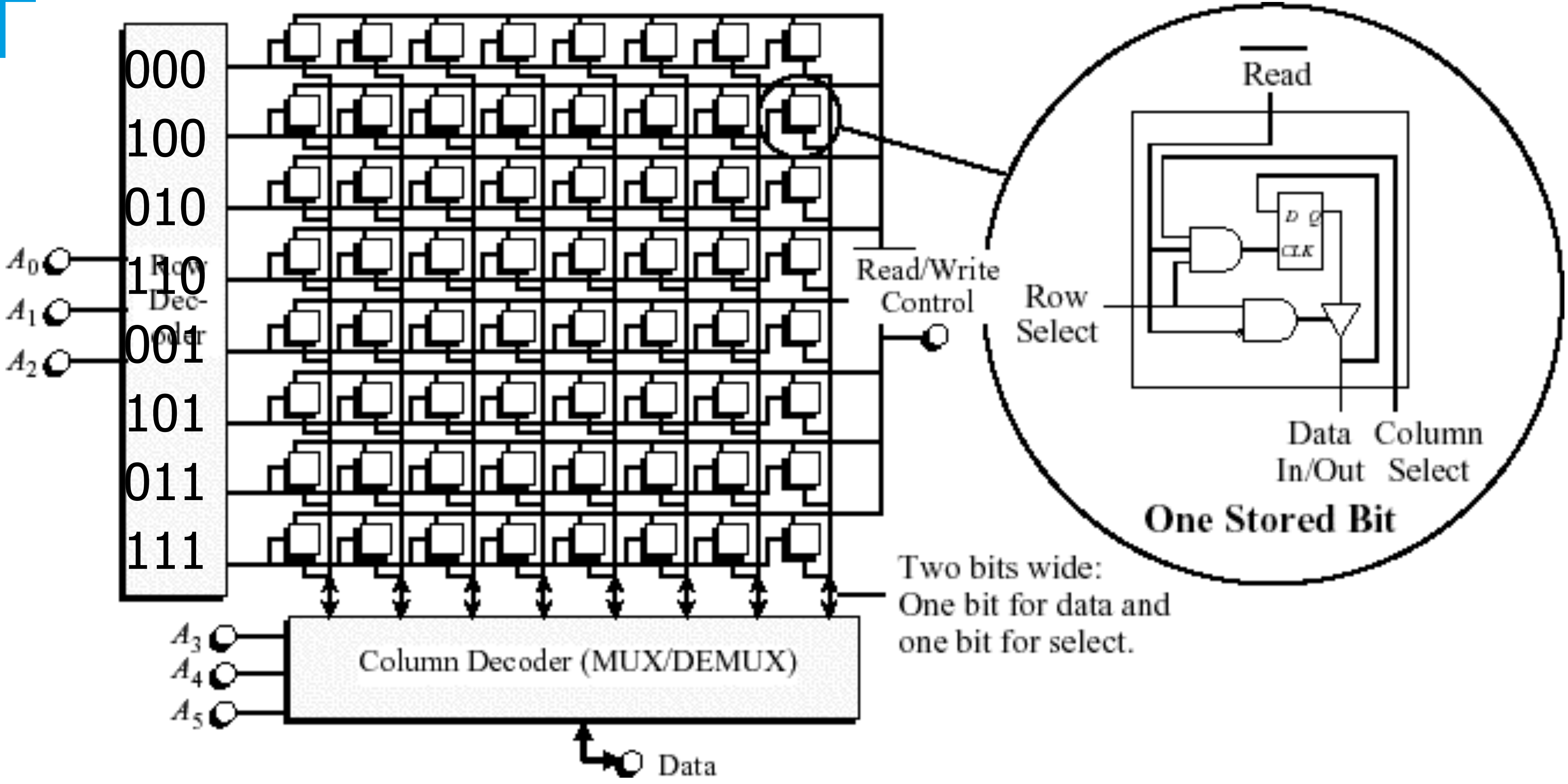
  To read, sensitive amplifier circuits detect small changes in bit line.

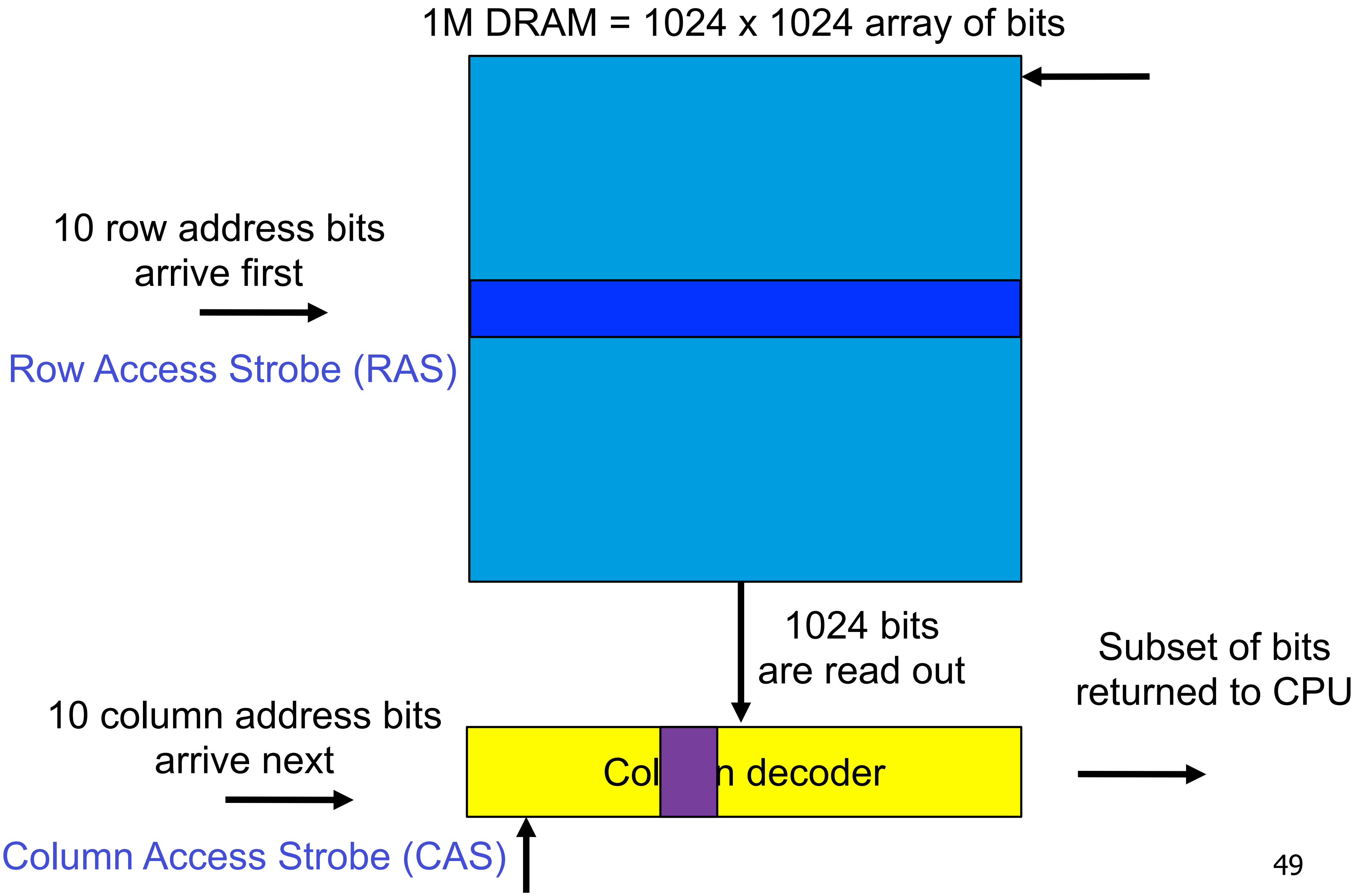
- storage cells consist of
  one capacitor and transistor per data bit

  Reading discharges the capacitor.



2-1/2D Organization of a 64-Word by One-Bit RAM





1M DRAM = 1024 x 1024 array of bits

10 row address bits
arrive first

Row Access Strobe (RAS)

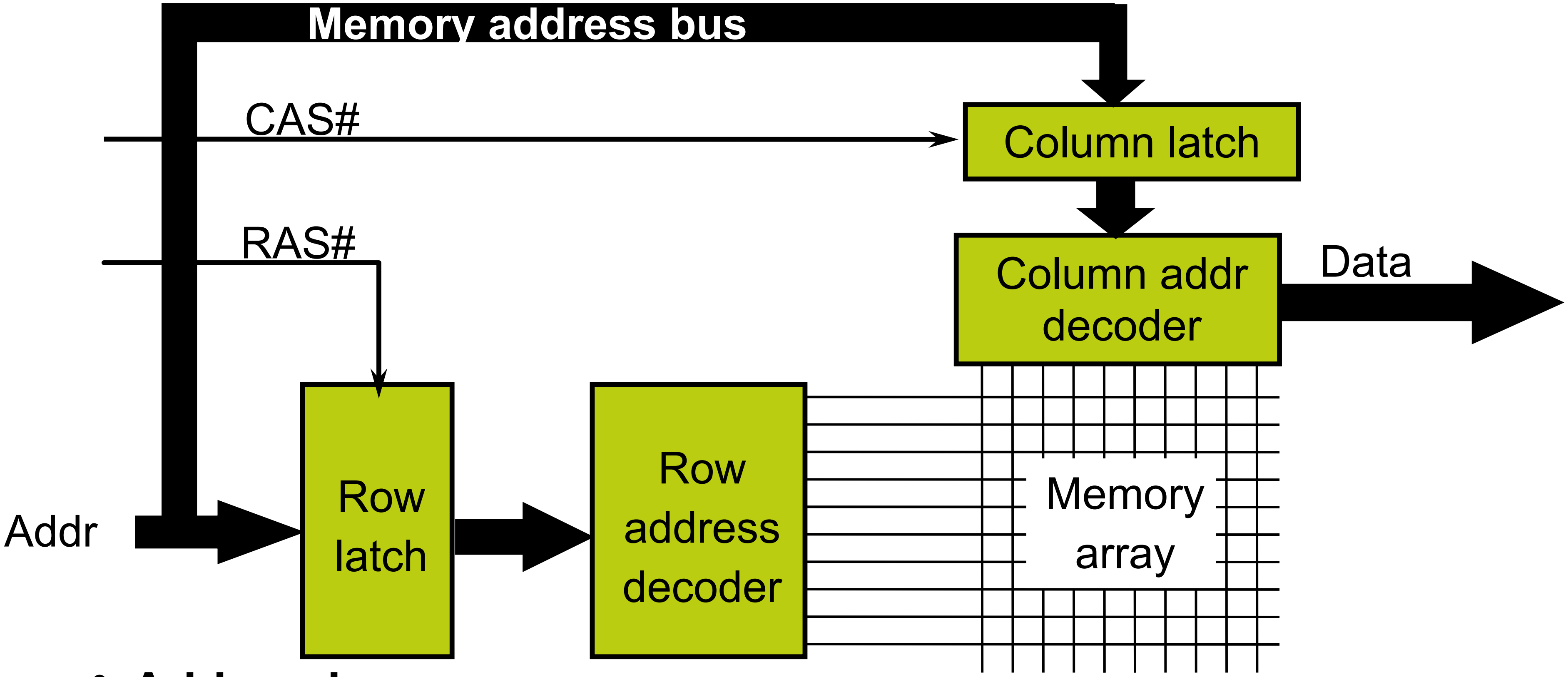
1024 bits
are read out

Subset of bits
returned to CPU

10 column address bits
arrive next

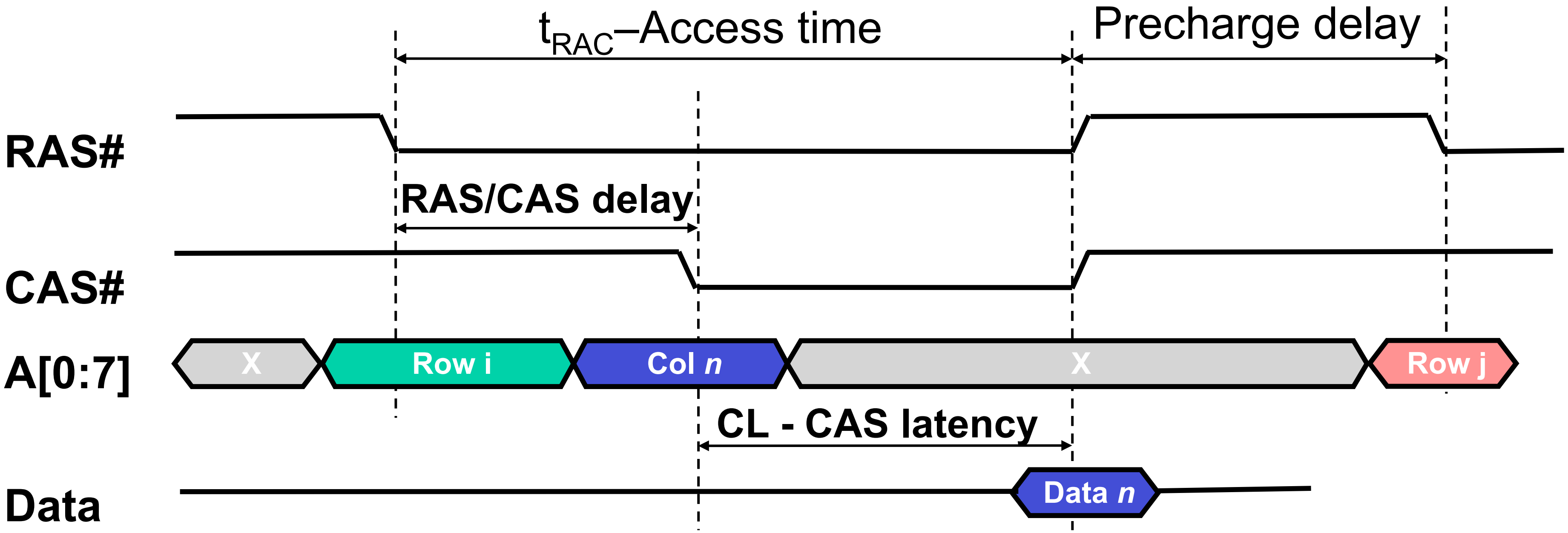
Column Access Strobe (CAS)

Basic DRAM chip



- **Addressing sequence**
  - Row address and then RAS# asserted
  - RAS# to CAS# delay
  - Column address and then CAS# asserted
  - DATA transfer

Addressing sequence



- **Access sequence**
  - Put row address on data bus and assert RAS#
    - Wait for RAS# to CAS# delay (t_{RCD})
  - Put column address on data bus and assert CAS#
  - DATA transfer
  - Pre-charge

RAM Latency: "tCAS-tRCD-tRP-tRAS"

- **tCAS**

The number of clock cycles needed to access a certain column of Data in SDRAM. CAS Latency, or simply CAS, is known as Column Address Strobe Latency, sometimes referred to as **tCL**.

tRCD (RAS to CAS Delay)

The number of Clock cycles needed between a Row Address Strobe (RAS) and a CAS. It is the time required between the computer defining the row and column of the given memory block and the actual read or write to that location. Stands for Row address to Column address Delay.

tRP (RAS Precharge)

The number of clock cycles needed to terminate access to an open row of memory, and open access to the next row. Stands for Row precharge time.

tRAS

The minimum number of clock cycles needed to access a certain row of data in RAM between the data request and the precharge command. Known as Active to Precharge Delay.

RAM speeds are given by the four numbers above. So, for example, latency values given as 2.5-3-3-8 would indicate tCAS=2.5, tRCD=3, tRP=3, tRAS=8. (Note that 0.5 values of latency (such as 2.5) are only possible in Double data rate RAM, where two parts of each clock cycle are used)

Latency

- Memory latency is traditionally quoted using two measures:
  - **access time** is the time between when a read is requested and when the desired word arrives
  - **cycle time** is the minimum time between requests

Cycle time is greater than access time because the memory needs the address lines to be stable between accesses.

DRAM Properties

- The RAS and CAS bits share the same pins on the chip (multiplex)
  - Column Address Strobe dictates how many clocks the memory waits before sending data on.

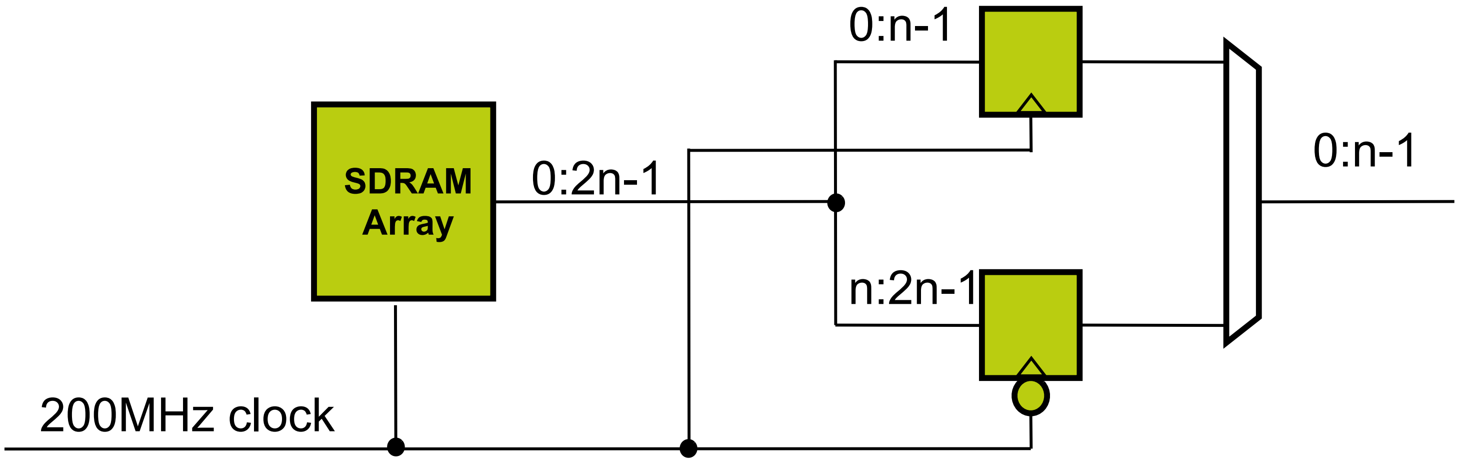
- Each bit loses its value after a while – hence, each bit has to be refreshed periodically:

  This is done by reading each row and writing the value back (hence, dynamic random access memory) – causes variability in memory access time

- SDRAM runs Synchronously with the clock of the processor and the system bus.

DDR-SDRAM

- **2n-prefetch architecture**
  - The DRAM cells are clocked at the same speed as SDR SDRAM
  - Internal data bus is twice the width of the external data bus
  - Data capture occurs twice per clock cycle
    - Lower half of the bus sampled at clock rise
    - Upper half of the bus sampled at clock fall



- **Uses 2.5V** (vs. 3.3V in SDRAM)
  - Reduced power consumption

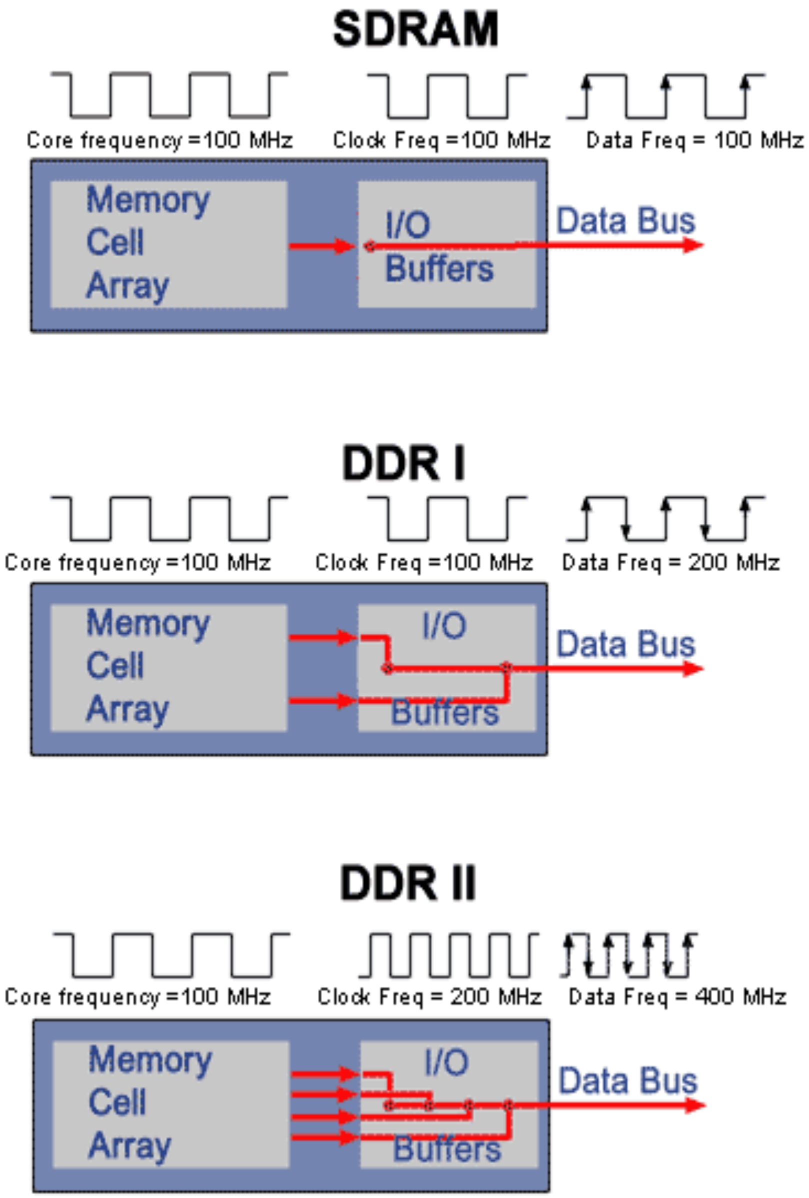
DIMMs

- **DIMM: Dual In-line Memory Module**
  - A small circuit board that holds memory chips

- **64-bit wide data path (72 bit with parity)**
  - Single sided: 9 chips, each with 8 bit data bus
    - 512 Mbit / chip × 8 chips ⇒ 512 Mbyte per DIMM
  - Dual sided: 18 chips, each with 4 bit data bus
    - 256 Mbit / chip × 16 chips ⇒ 512 Mbyte per DIMM

DDR2

- **DDR2 achieves high-speed using 4-bit prefetch architecture**
  - SDRAM cells read/write 4× the amount of data as the external bus
  - DDR2-533 cell works at the same frequency as a DDR266 SDRAM or a PC133 SDRAM cell
- **This method comes at a price of increased latency for lower clocks**
  - DDR2-based systems may perform worse than DDR1-based systems



DDR3

- 30% a power consumption reduction compared to DDR2
  - 1.5 V supply voltage, compared to DDR2's 1.8 V or DDR's 2.5 V
  - 90 nanometer fabrication technology

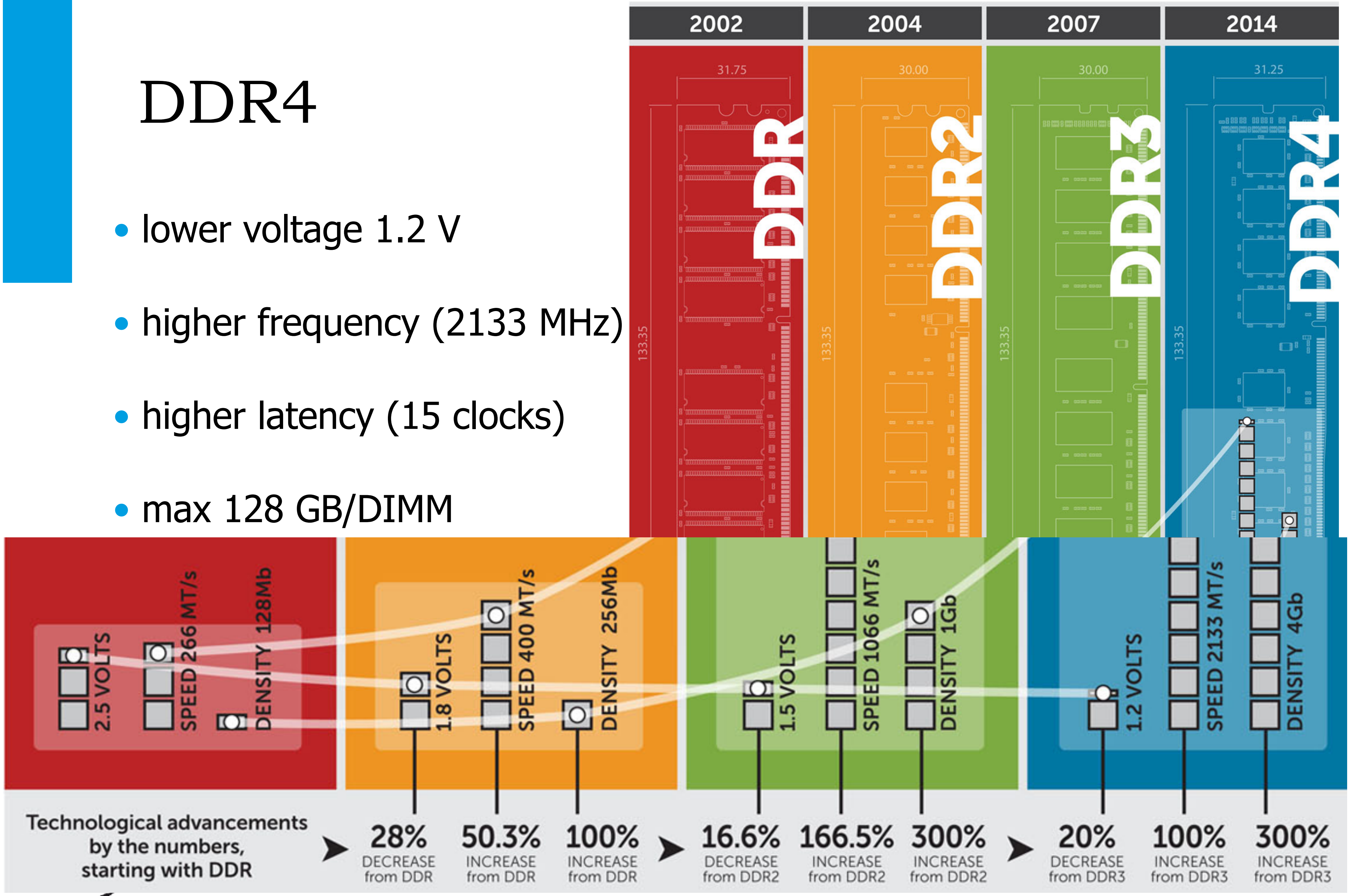
- Higher bandwidth
  - 8 bit deep prefetch buffer (vs. 4 bit in DDR2 and 2 bit in DDR)

- Transfer data rate
  - Effective clock rate of 800–1600 MHz using both rising and falling edges of a 400–800 MHz I/O clock.
  - DDR2: 400–800 MHz using a 200–400 MHz I/O clock
  - DDR: 200–400 MHz based on a 100–200 MHz I/O clock

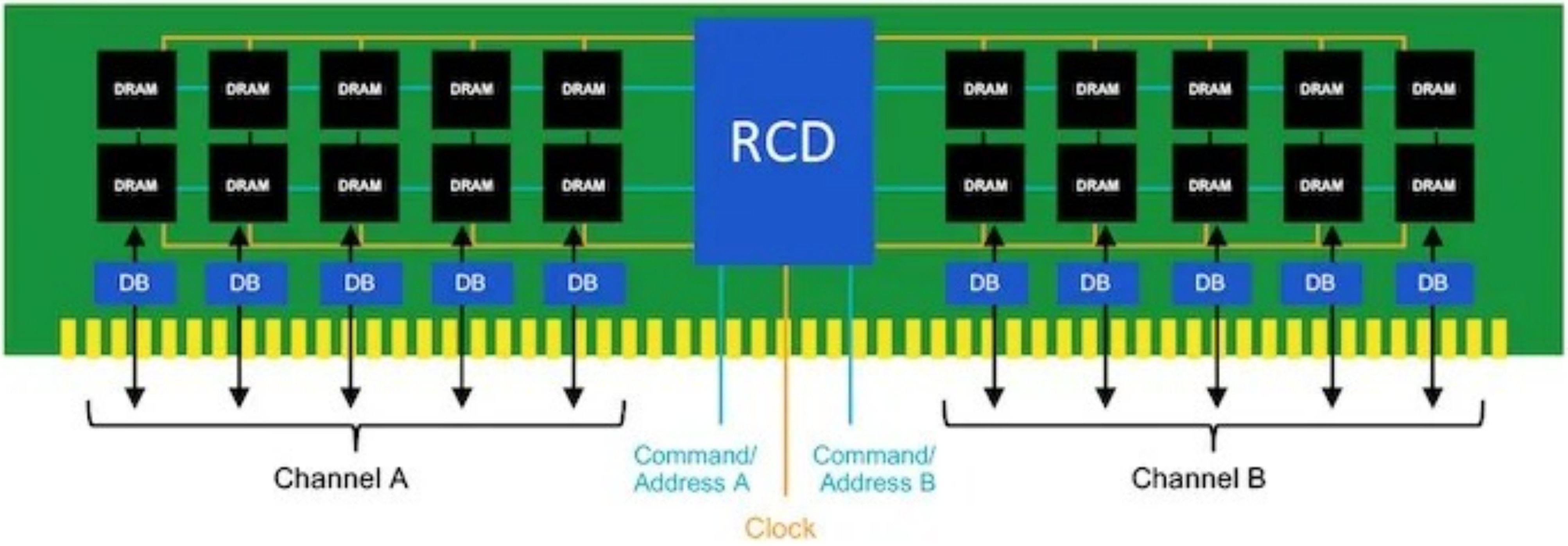
- DDR3 DIMMs
  - 240 pins, the same number as DDR2, and are the same size
  - Electrically incompatible, and have a different key notch location

DDR4

- lower voltage 1.2 V
- higher frequency (2133 MHz)
- higher latency (15 clocks)
- max 128 GB/DIMM



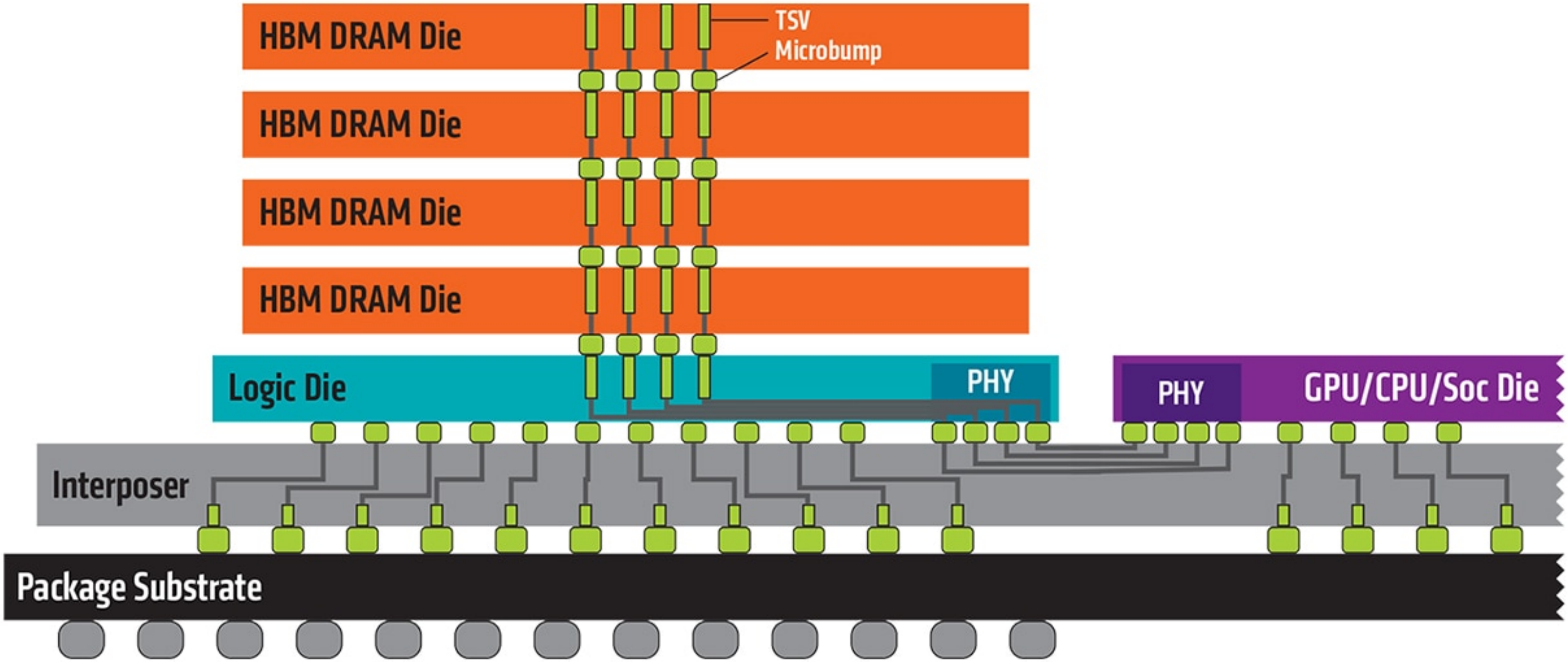
DDR5



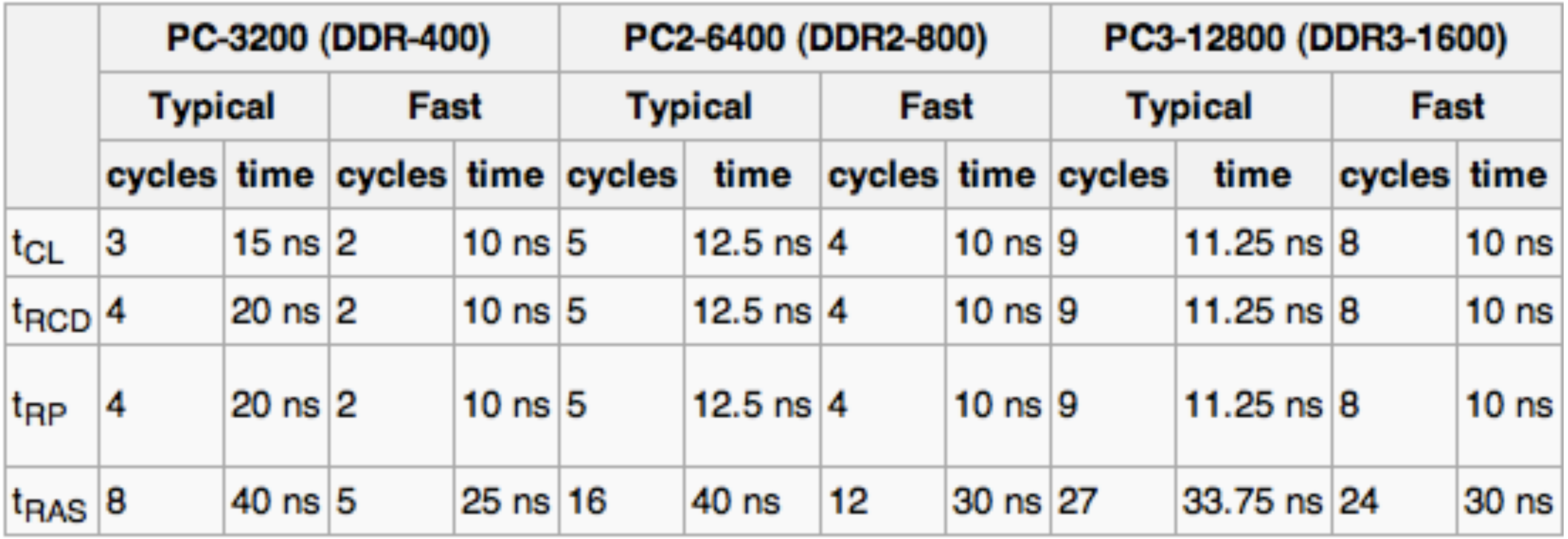
DRAM
RCD
DB
Channel A
Command/ Address A
Clock
Command/ Address B
Channel B

High Bandwidth Memory (HBM)

- DRAM stacked memory



Memory latency over generations

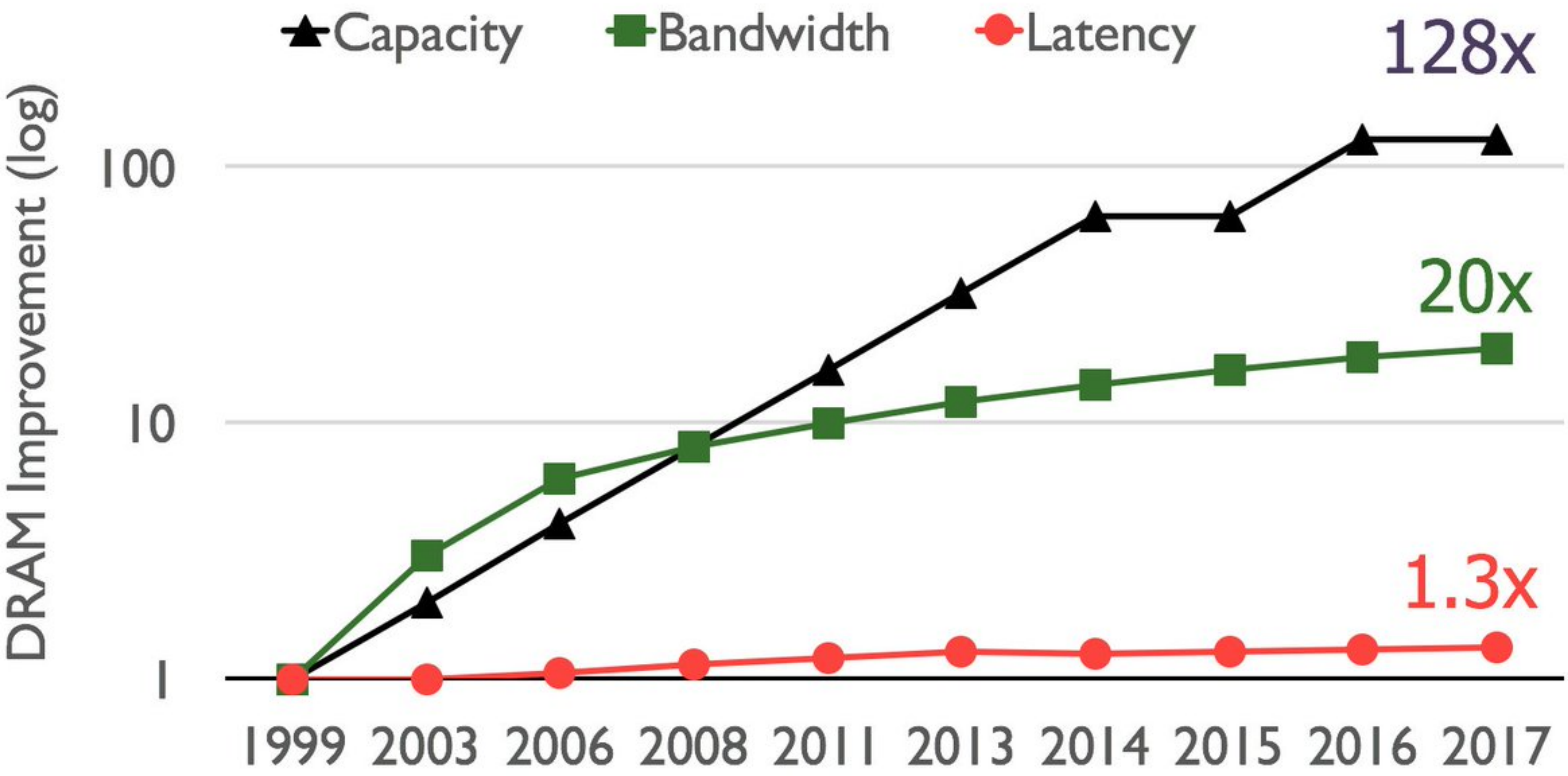


	PC-3200 (DDR-400)				PC2-6400 (DDR2-800)				PC3-12800 (DDR3-1600)			
	Typical		Fast		Typical		Fast		Typical		Fast	
	cycles	time	cycles	time	cycles	time	cycles	time	cycles	time	cycles	time
t_{CL}	3	15 ns	2	10 ns	5	12.5 ns	4	10 ns	9	11.25 ns	8	10 ns
t_{RCD}	4	20 ns	2	10 ns	5	12.5 ns	4	10 ns	9	11.25 ns	8	10 ns
t_{RP}	4	20 ns	2	10 ns	5	12.5 ns	4	10 ns	9	11.25 ns	8	10 ns
t_{RAS}	8	40 ns	5	25 ns	16	40 ns	12	30 ns	27	33.75 ns	24	30 ns

- It is worth noting that the latency improvement over 11 years is not that large. However, the DDR3 memory does achieve 32 times higher bandwidth.

- http://en.wikipedia.org/wiki/Dynamic_random_access_memory

Memory trends



Limits on DRAM performance

- Read cycle time, the time between successive read operations. This time decreased from 10 ns for 100 MHz SDRAM to 5 ns for DDR-400, but has remained relatively unchanged through DDR2-800 and DDR3-1600 generations. However, the achievable bandwidth has increased rapidly.

- Another limit is the CAS latency, the time between supplying a column address and receiving the corresponding data. Again, this has remained relatively constant at 10–15 ns through the last few generations of DDR SDRAM.

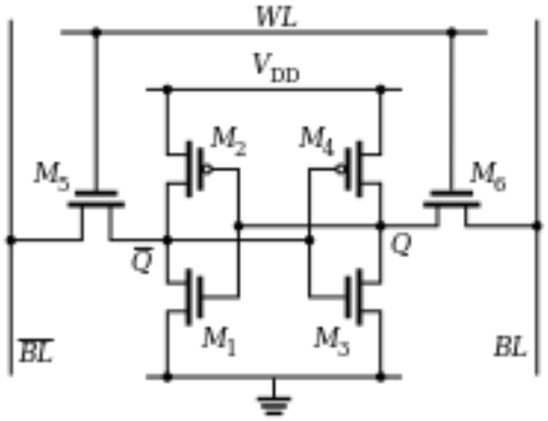
- The benefits of SDRAM's internal buffering come from its ability to interleave operations to multiple banks of memory, thereby increasing effective bandwidth.

SRAM – Static RAM

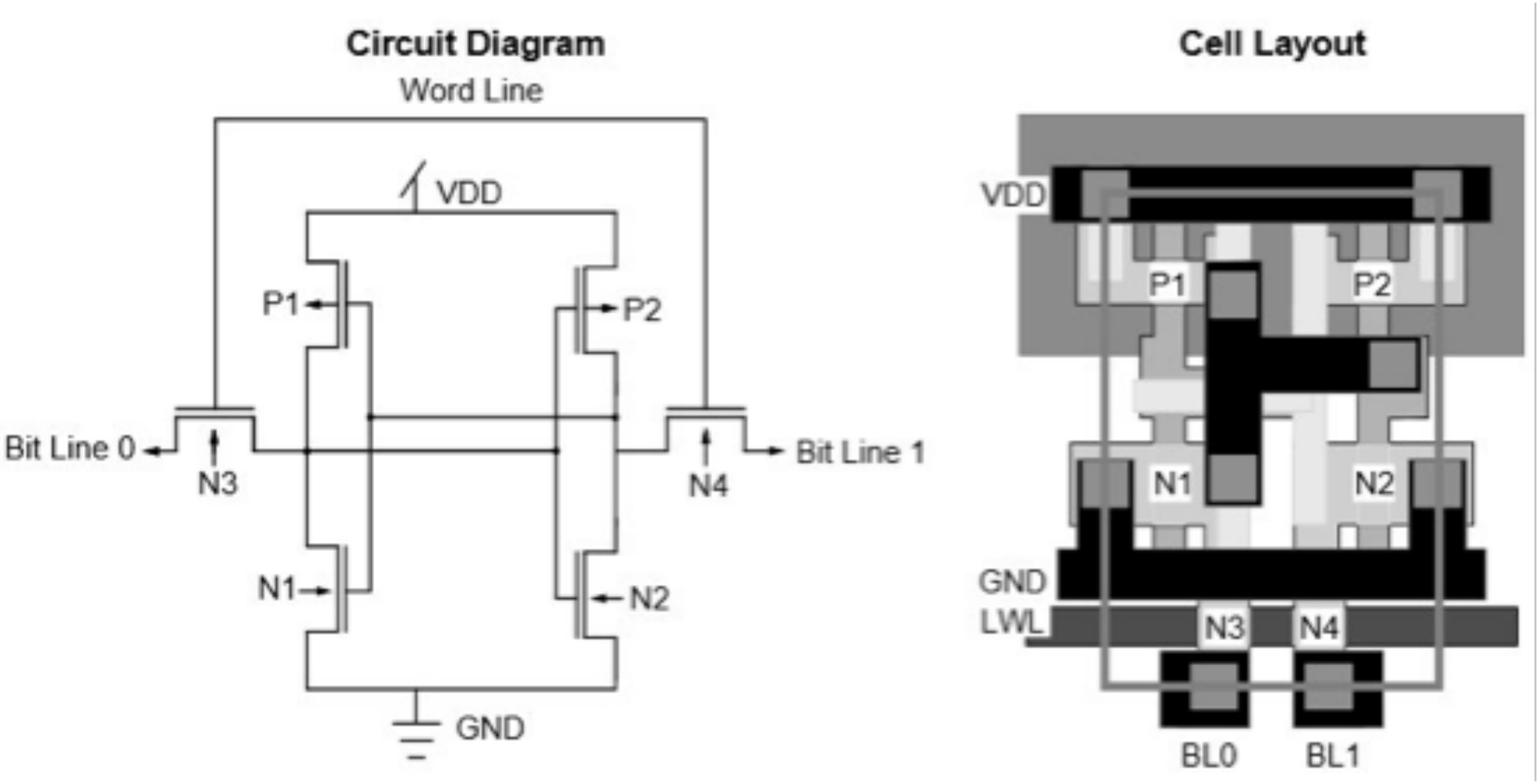
- **Static RAM** uses a completely different technology. In static RAM, a form of flip-flop holds each bit of memory. A flip-flop for a memory cell takes four or six transistors along with some wiring, but never has to be refreshed. This makes static RAM significantly faster than dynamic RAM. However, because it has more parts, a static memory cell takes up a lot more space on a chip than a dynamic memory cell. Therefore, you get less memory per chip, and that makes static RAM a lot more expensive.

- http://computer.howstuffworks.com/ram3.htm

-



SRAM



Circuit Diagram

Cell Layout

SRAM – Static RAM

- True random access
- High speed, low density, high power
- No refresh
- Address not multiplexed

- **DDR SRAM**
  - 2 READs or 2 WRITEs per clock
  - Common or Separate I/O
  - DDRII: 200MHz to 333MHz Operation; Density: 18/36/72Mb+

- **QDR SRAM**
  - Two separate DDR ports: one read and one write
  - One DDR address bus: alternating between the read address and the write address
  - QDRII: 250MHz to 333MHz Operation; Density: 18/36/72Mb+

Summary Random Access Memory

- Dynamic RAM (DRAM)
  - Each bit is stored in a capacitor
  - Uses one capacitor and one transistor per bit
  - Slower, but takes up less space in a chip
  - Must be refreshed periodically (milliseconds), since the capacitor leaks

- Static RAM (SRAM)
  - Each bit is stored in a type of flip-flop
  - Typically takes four or six transistors per bit
  - Faster, but takes up more space in a chip
  - Retains information as long as power is supplied
  - equal access time.

SRAM vs. DRAM

	DRAM – Dynamic RAM	SRAM – Static RAM
Refresh	Regular refresh (~1% time)	No refresh needed
Address	Address muxed: row+ column	Address not multiplexed
Access	Not true "Random Access"	True "Random Access"
density	High (1 Transistor/bit)	Low (6 Transistor/bit)
Power	low	high
Speed	slow	fast
Price/bit	low	high
Typical usage	Main memory	cache

Trends in Memory

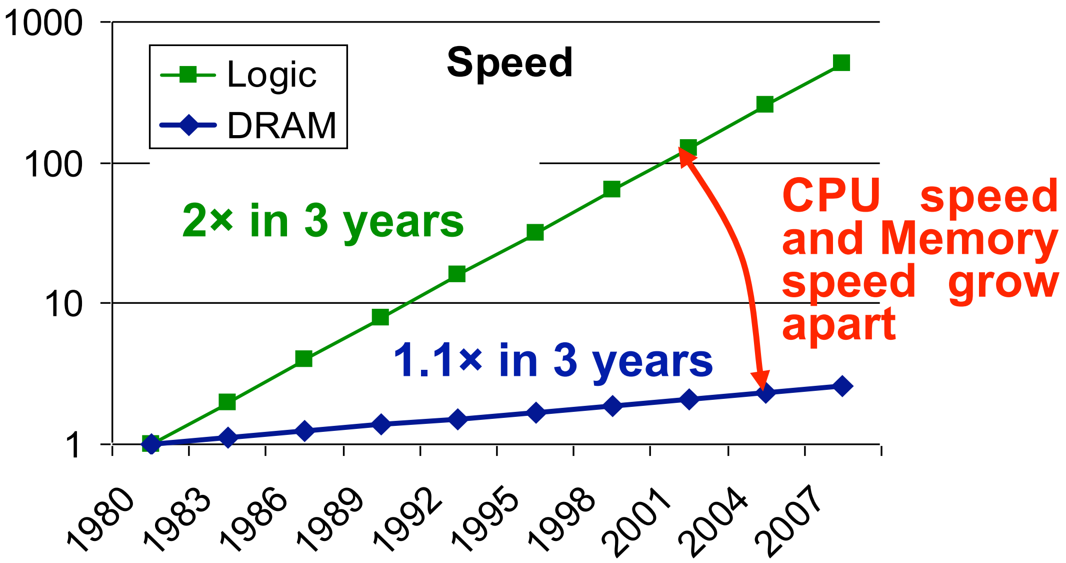
Technology Trends

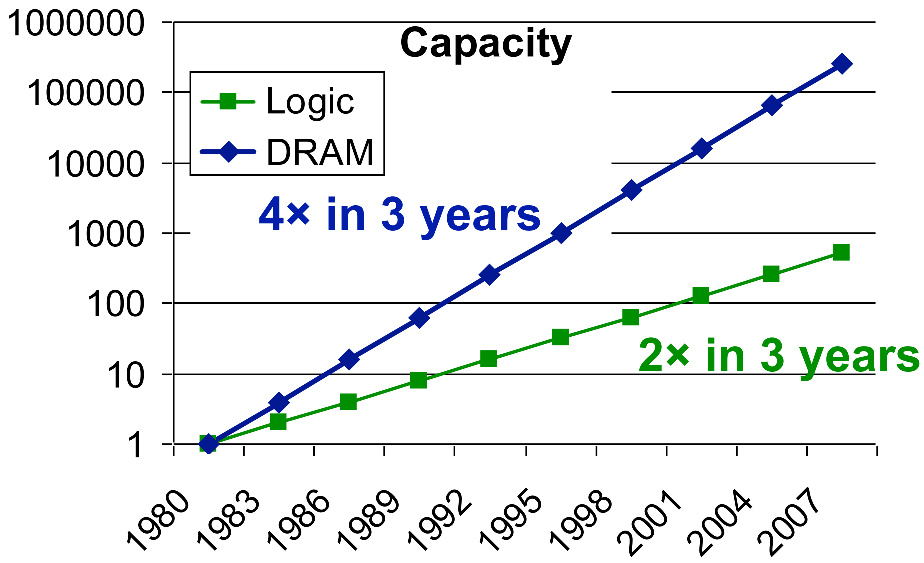
- Improvements in technology (smaller devices)
  => DRAM capacities double every two years

- Time to read data out of the array improves by only 5% every year
  => high memory latency (also called the memory wall!)

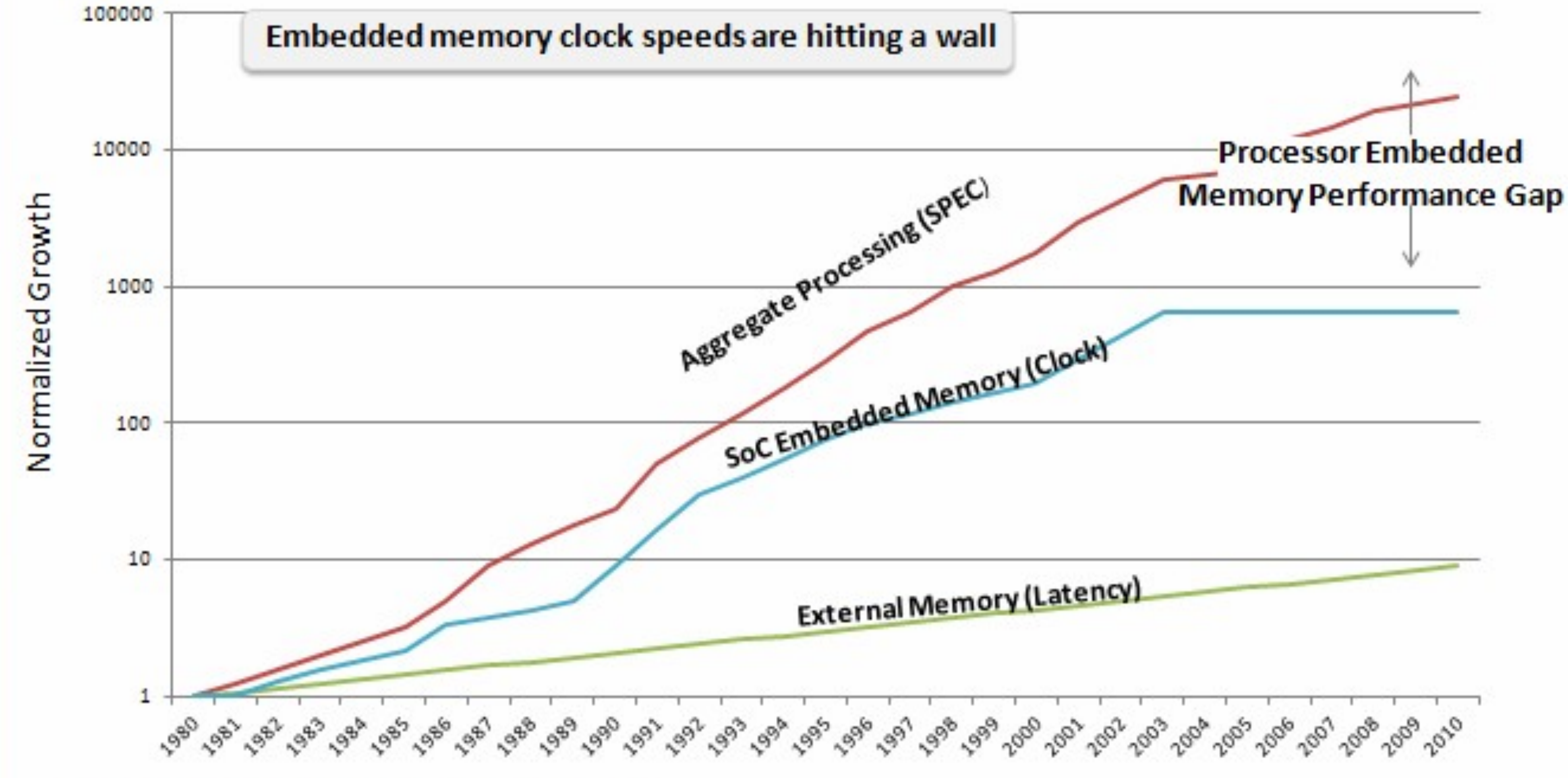
- Time to read data out of the column decoder improves by 10% every year
  => influences bandwidth

Technology Trends and Performance



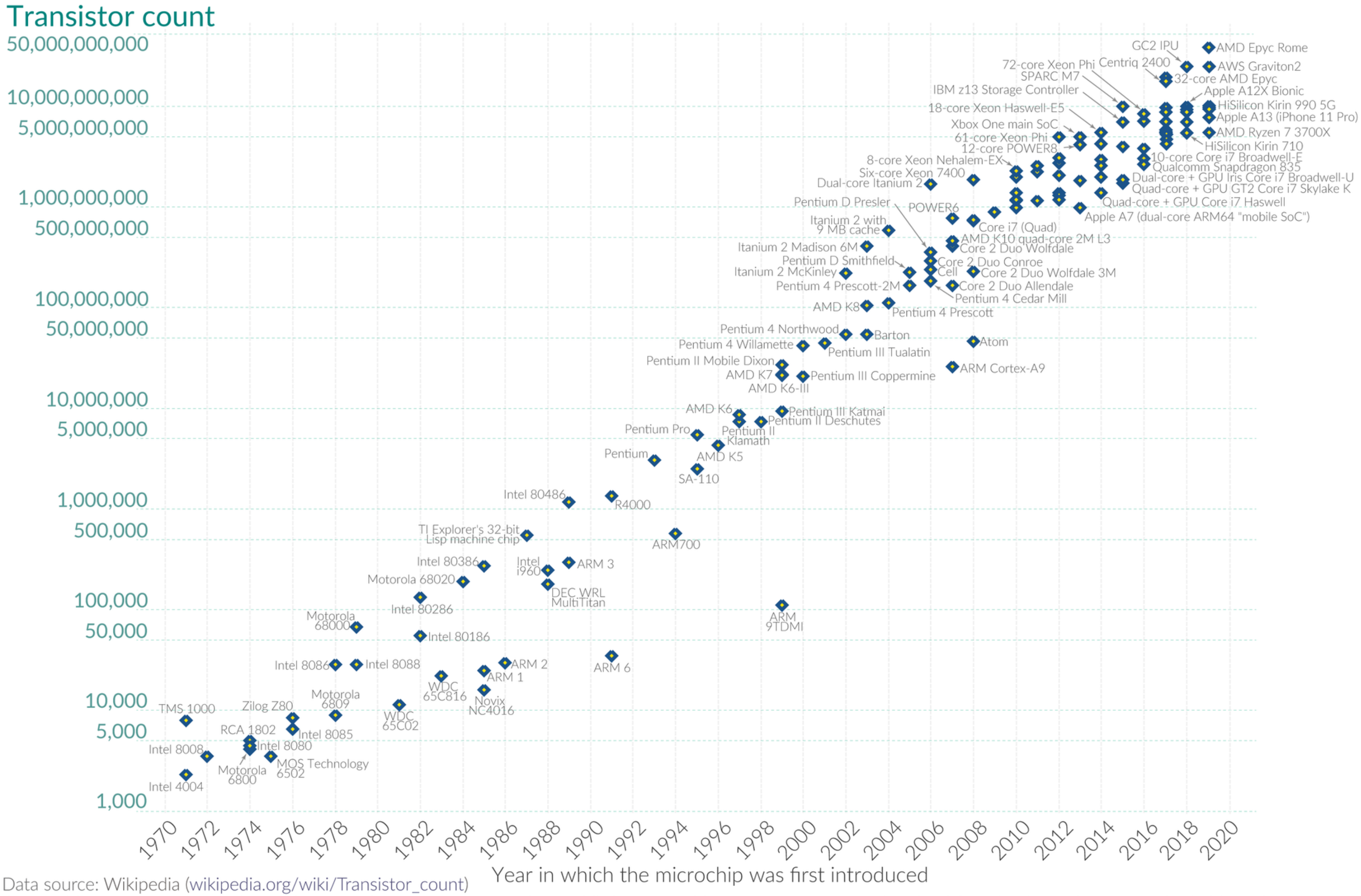


- Computing capacity: 4× per 3 years
- Moore's Law: Performance is doubled every ~18 months



*Source: Hennessy and Patterson, 5th Edition

Moore’s Law



Data source: Wikipedia (wikipedia.org/wiki/Transistor_count)

OurWorldinData.org – Research and data to make progress against the world’s largest problems.

Cost of DRAM Generations

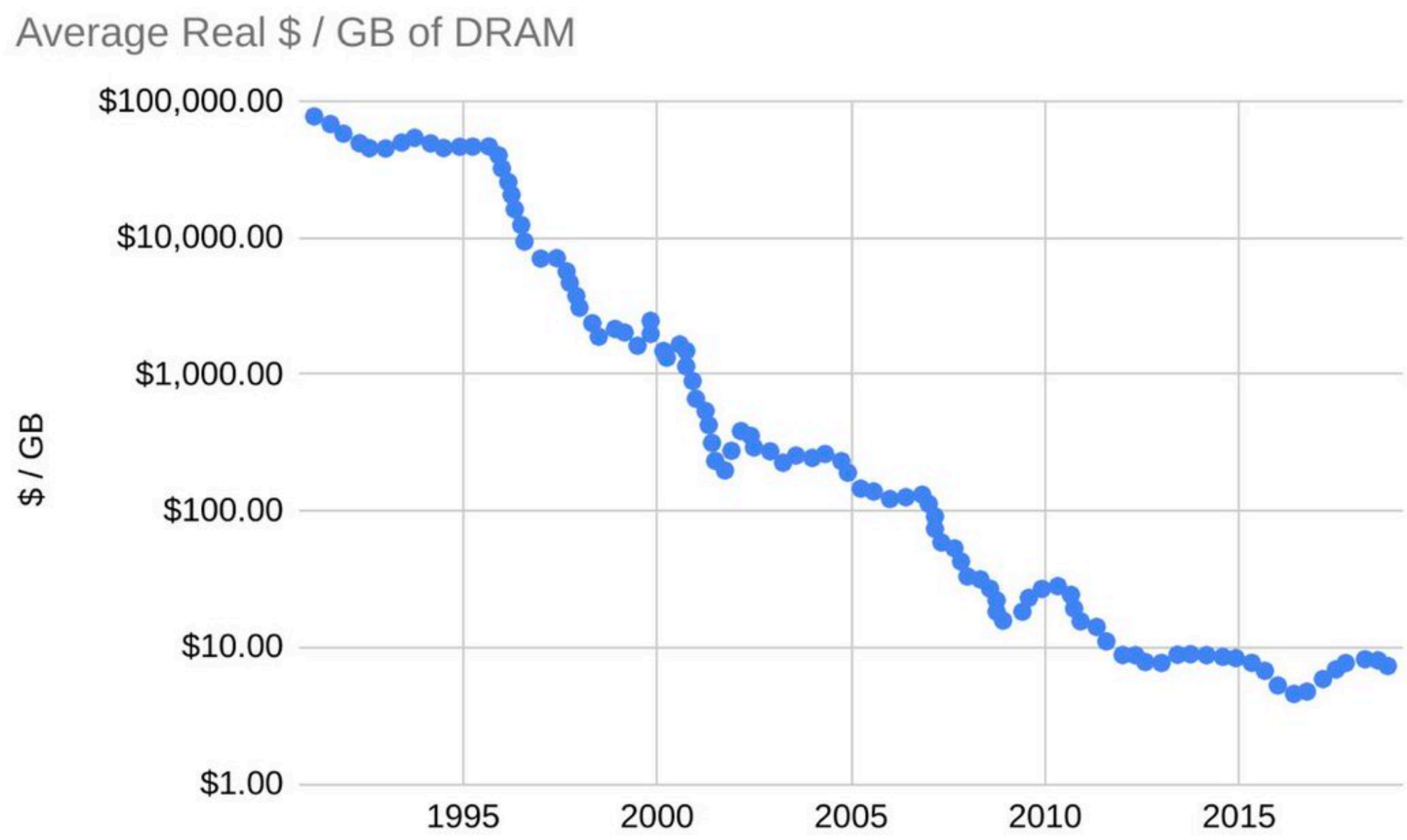


Figure 2: Average $ / GB of DRAM from 1991 to 2019 according to Objective Analysis. Dollars are 2020 dollars.

How to increase memory Bandwidth and Latency?

- By increasing the memory width (number of memory chips and the connecting bus), more bytes can be transferred together – increases cost

- Interleaved memory – since the memory is composed of many chips, multiple operations can happen at the same time – a single address is fed to multiple chips, allowing to read sequential words in parallel

➡ most increases have already been used and tried...., still a memory bottleneck due to latency;…. MEMORY WALL

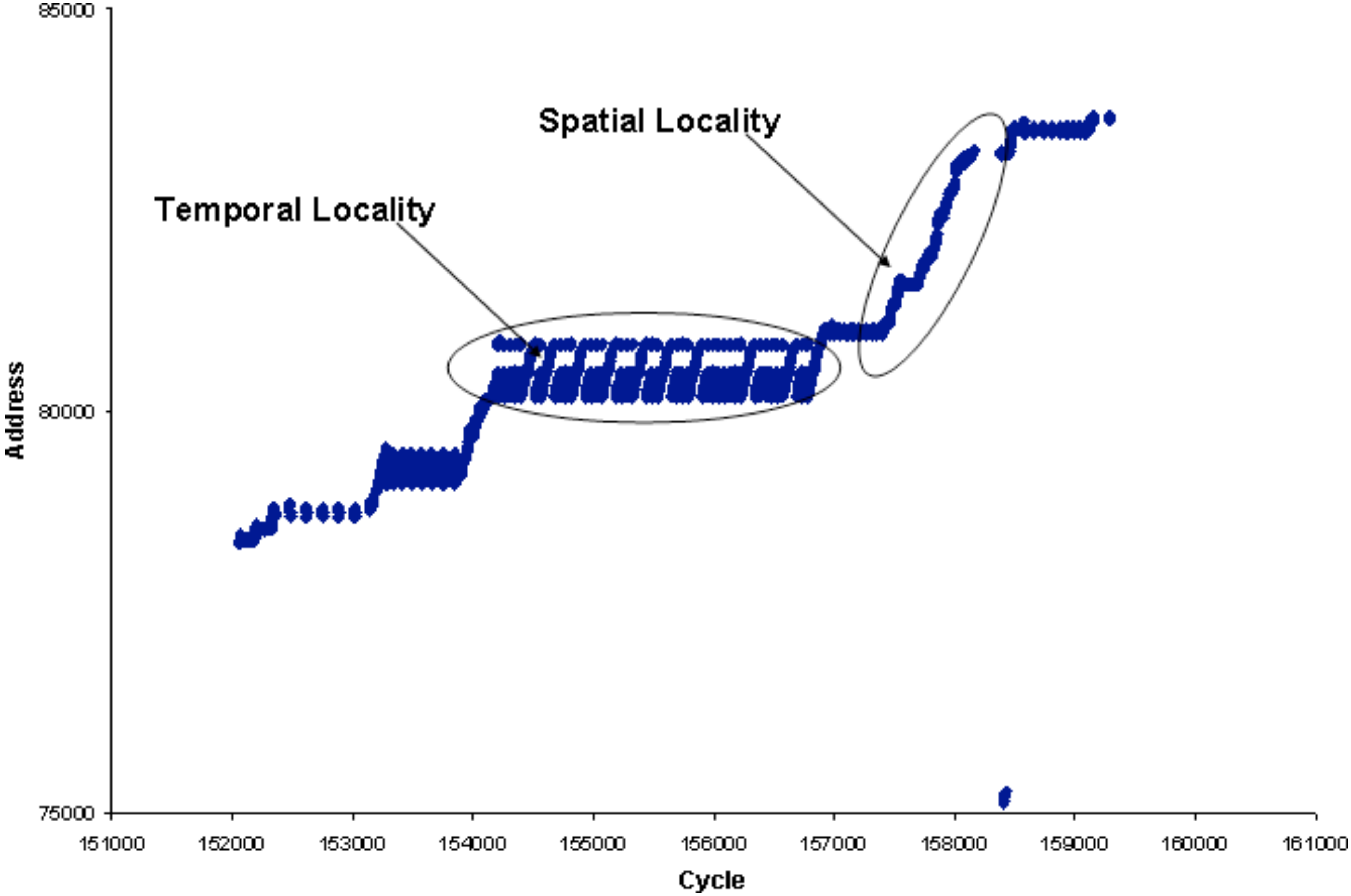
➡ How is memory used in a program?

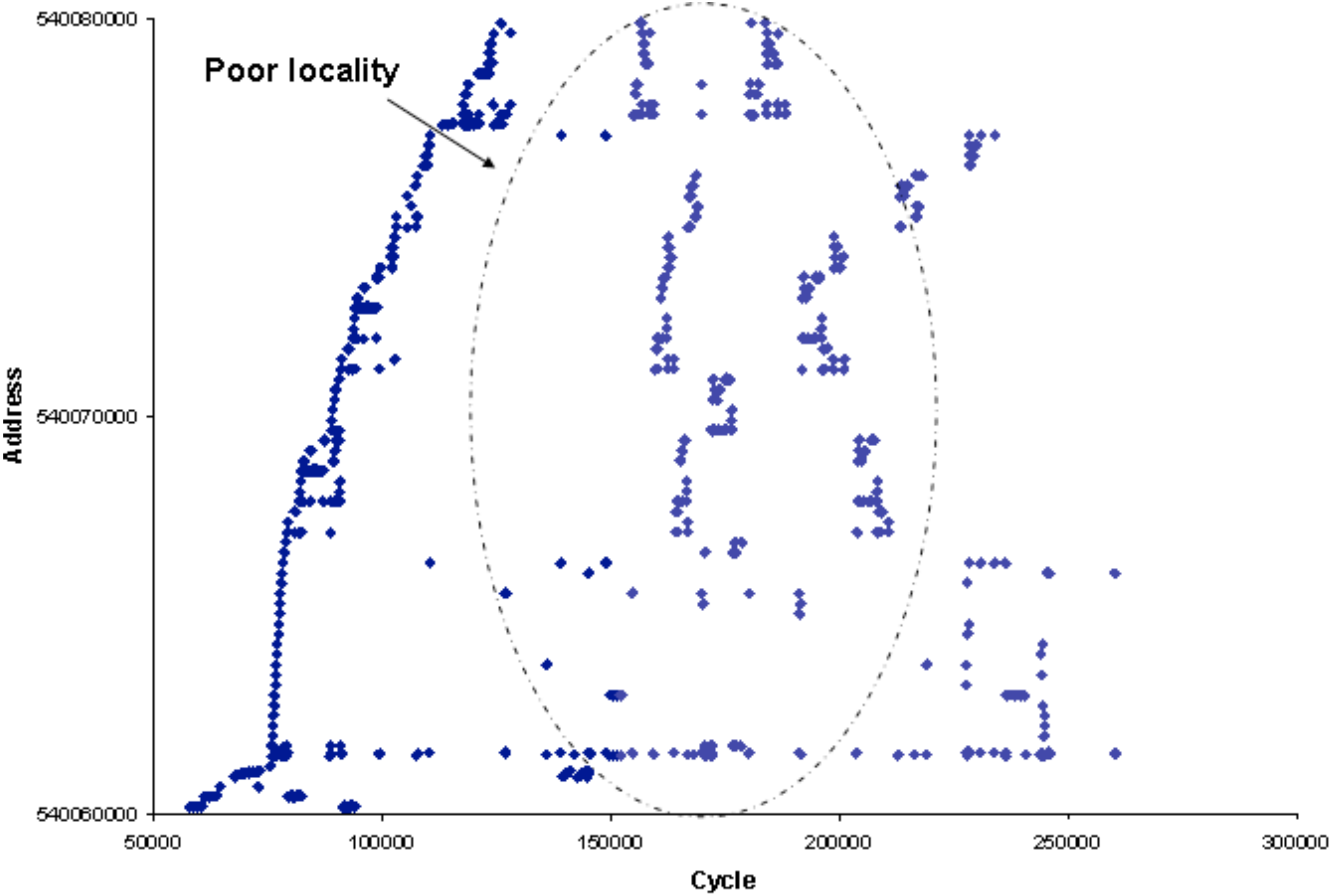
Observation: Principle of Locality

- Programs tend to reuse data and instructions they have used recently.
- A widely held rule of thumb is that a program spends **90**% of its execution time in only **10**% of the code. An implication of locality is that we can predict with reasonable accuracy what instructions and data a program will use in the near future based on its accesses in the recent past.

Two different types of locality have been observed:

- **Temporal** locality states that recently accessed items are likely to be accessed in the near future.
- **Spatial** locality says that items whose addresses are near one another tend to be referenced close together in time.


85000
80000
75000
Address
151000
152000
153000
154000
155000
156000
157000
158000
159000
160000
161000
Cycle
Temporal Locality
Spatial Locality


Poor locality
540080000
540070000
540060000
Address
50000
100000
150000
200000
250000
300000
Cycle

Using Locality by Caching

- main memory **latency** (which affects the cache miss penalty) is the primary concern of the **cache**, while main memory **bandwidth** is the primary concern of **multiprocessors** and I/O.

  it is generally easier to improve memory bandwidth with new organizations than it is to reduce latency.

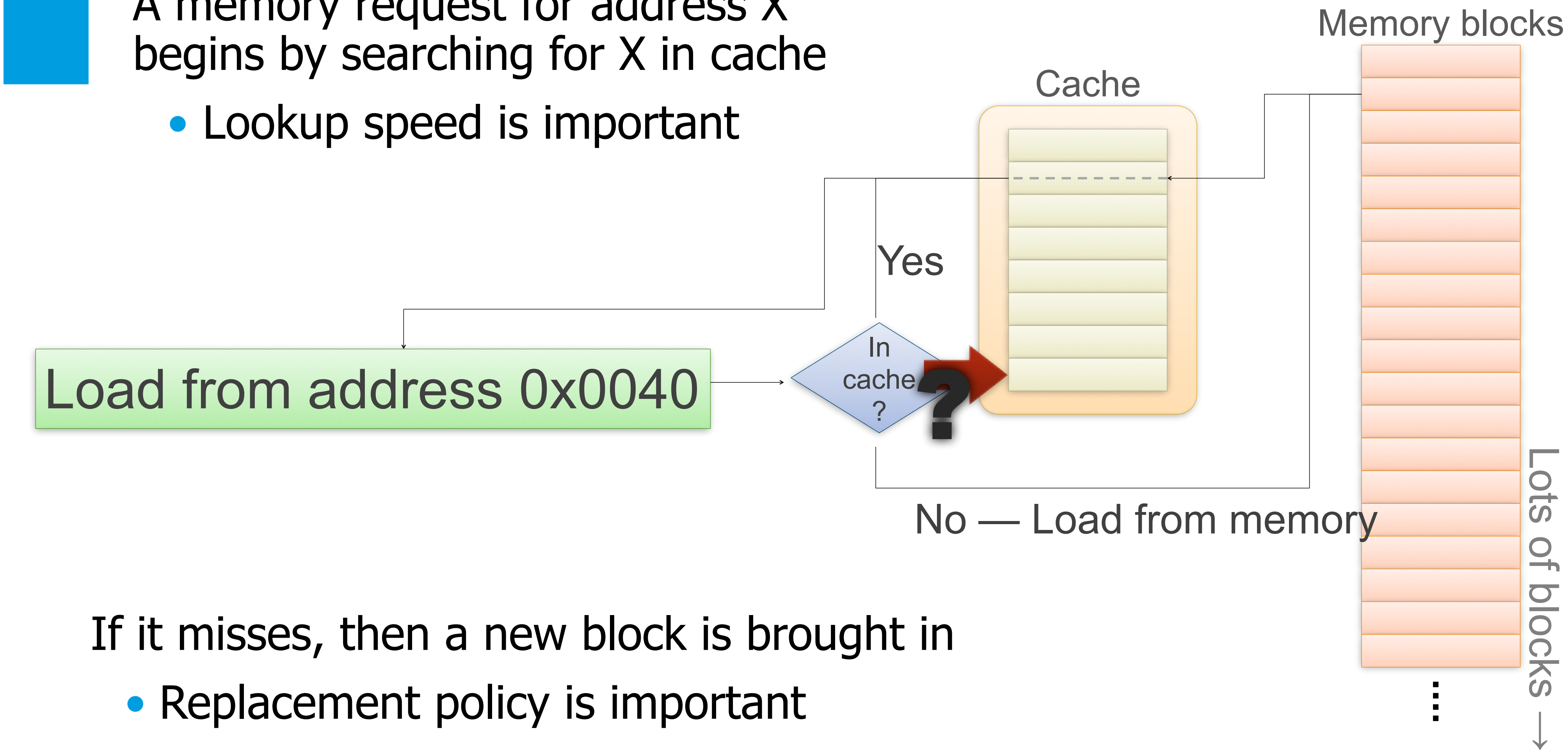
Cache operation

- CPU requests contents of memory location
- Check cache for this data
- If present, get from cache (fast)
- If not present, read required block from main memory to cache
- Then deliver data from cache to CPU
- Cache includes tags to identify which block of main memory is in each cache slot

Cache Lines

A memory request for address X begins by searching for X in cache

- Lookup speed is important



If it misses, then a new block is brought in

- Replacement policy is important

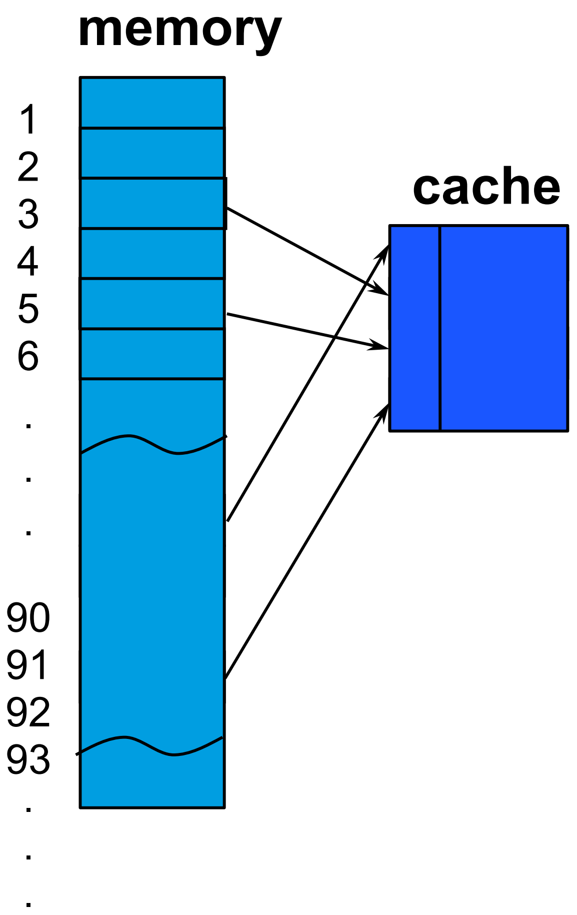
Cache – Main Idea

- The cache holds a small part of the entire memory
  - Need to map parts of the memory into the cache

- Main memory is (logically) partitioned into blocks
  - Typical block size is 32 to 64 bytes
  - Blocks are aligned

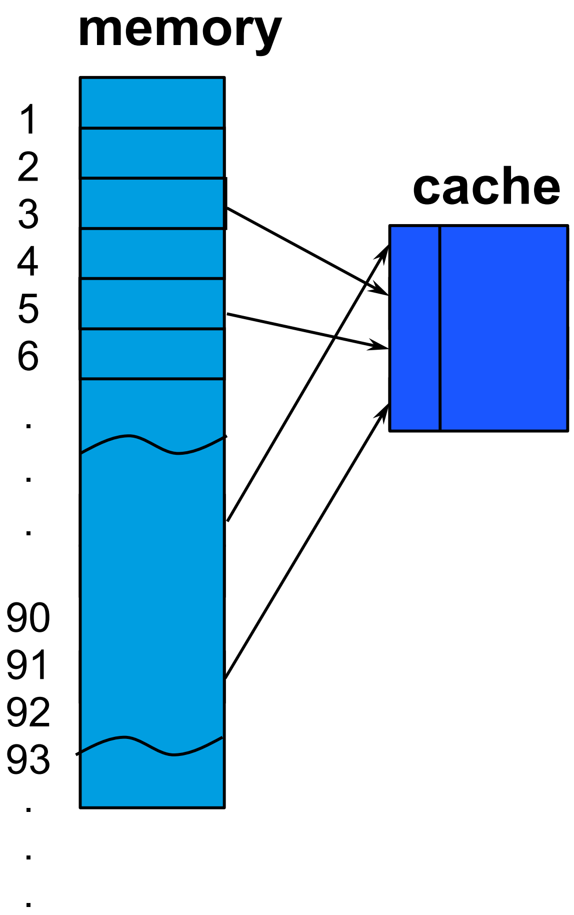
- Cache partitioned to cache lines
  - Each cache line holds a block
  - Only a subset of the blocks is mapped to the cache at a given time
  - The cache views an address as

Block #	offset

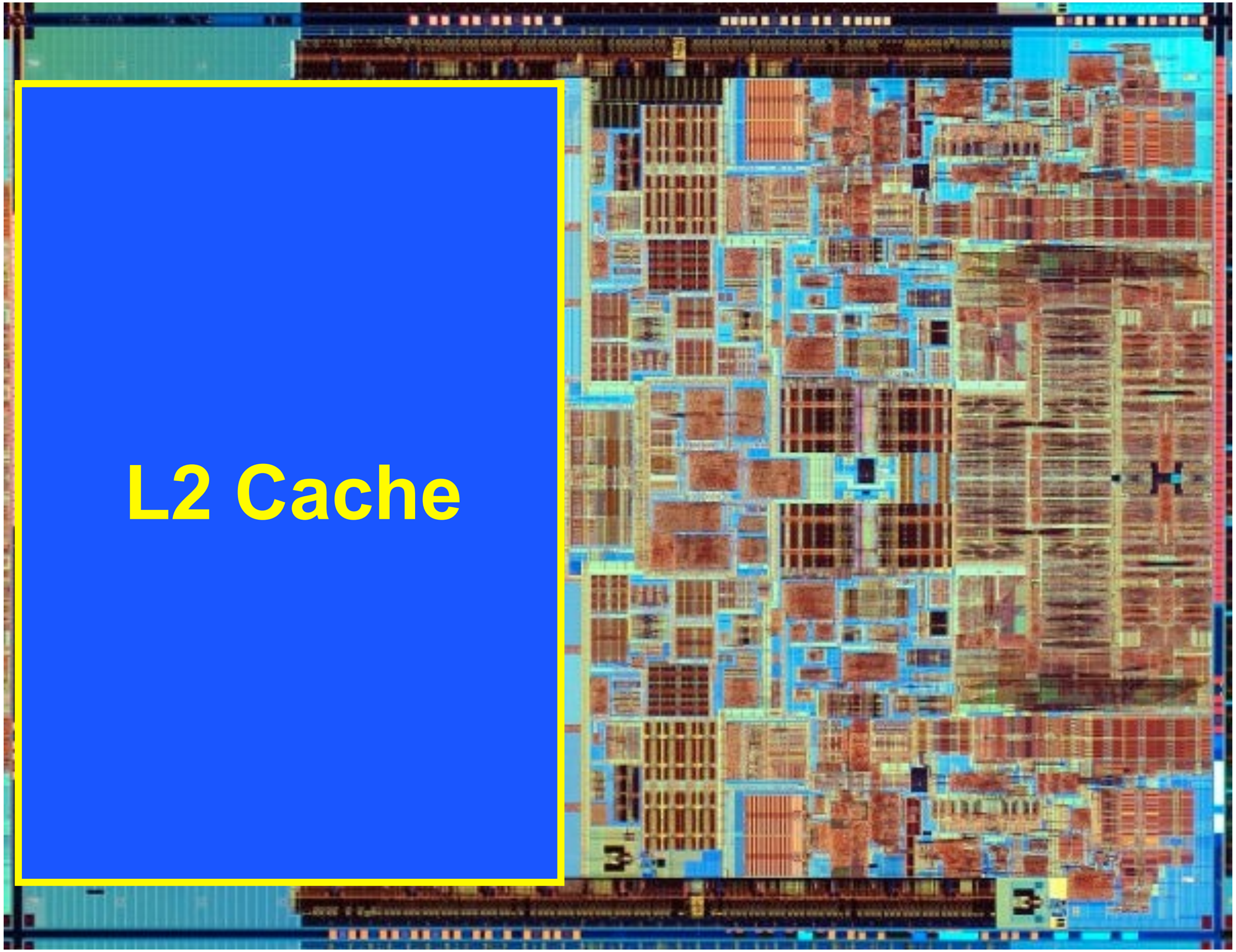


Cache Lookup

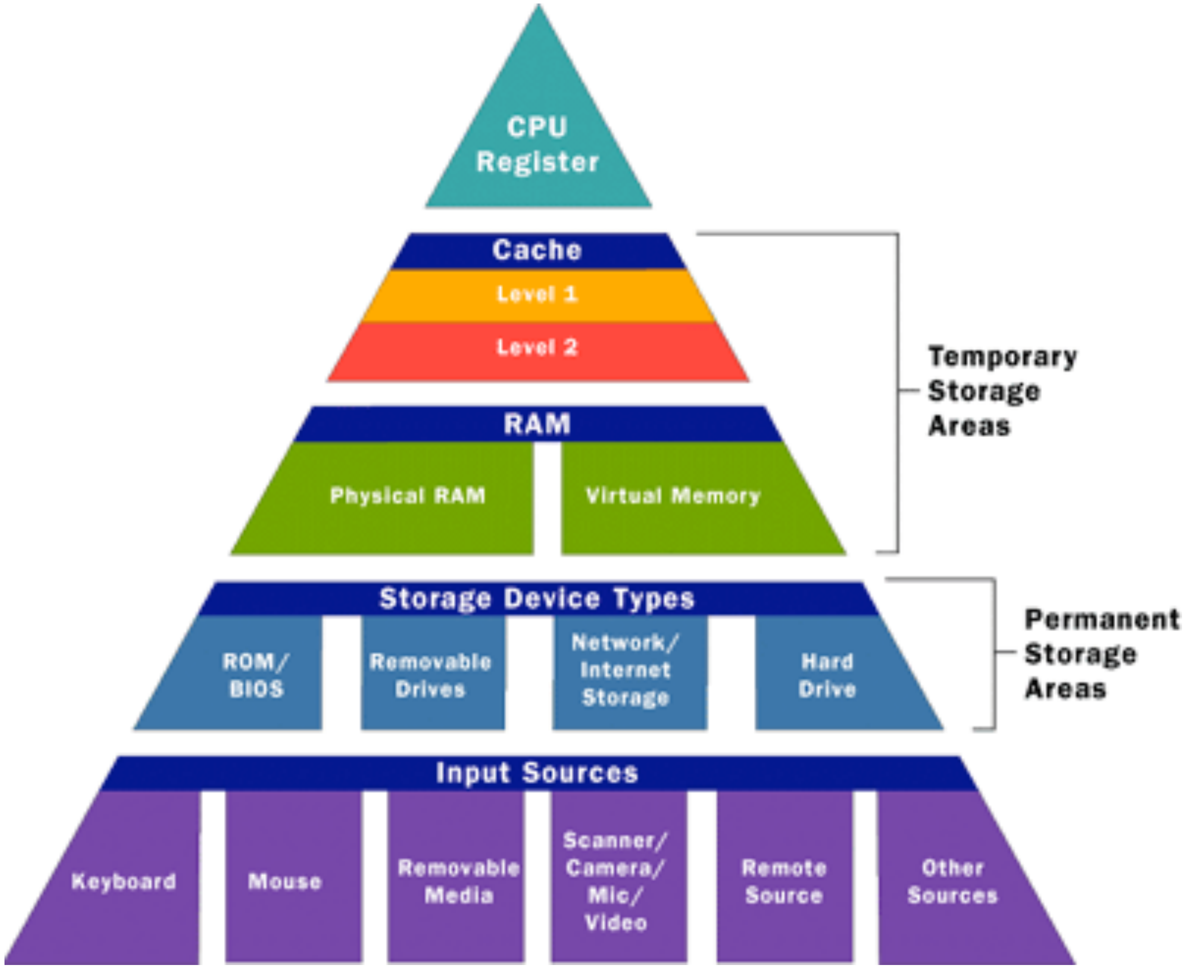
- Cache hit
  - Block is mapped to the cache –
    return data according to block's offset
- Cache miss
  - Block is not mapped to the cache
    ⇒ do a cache line fill
    - Fetch block into fill buffer
      - may require few bus cycle
    - Write fill buffer into cache
  - May need to remove another block
    from the cache to make room for the new block

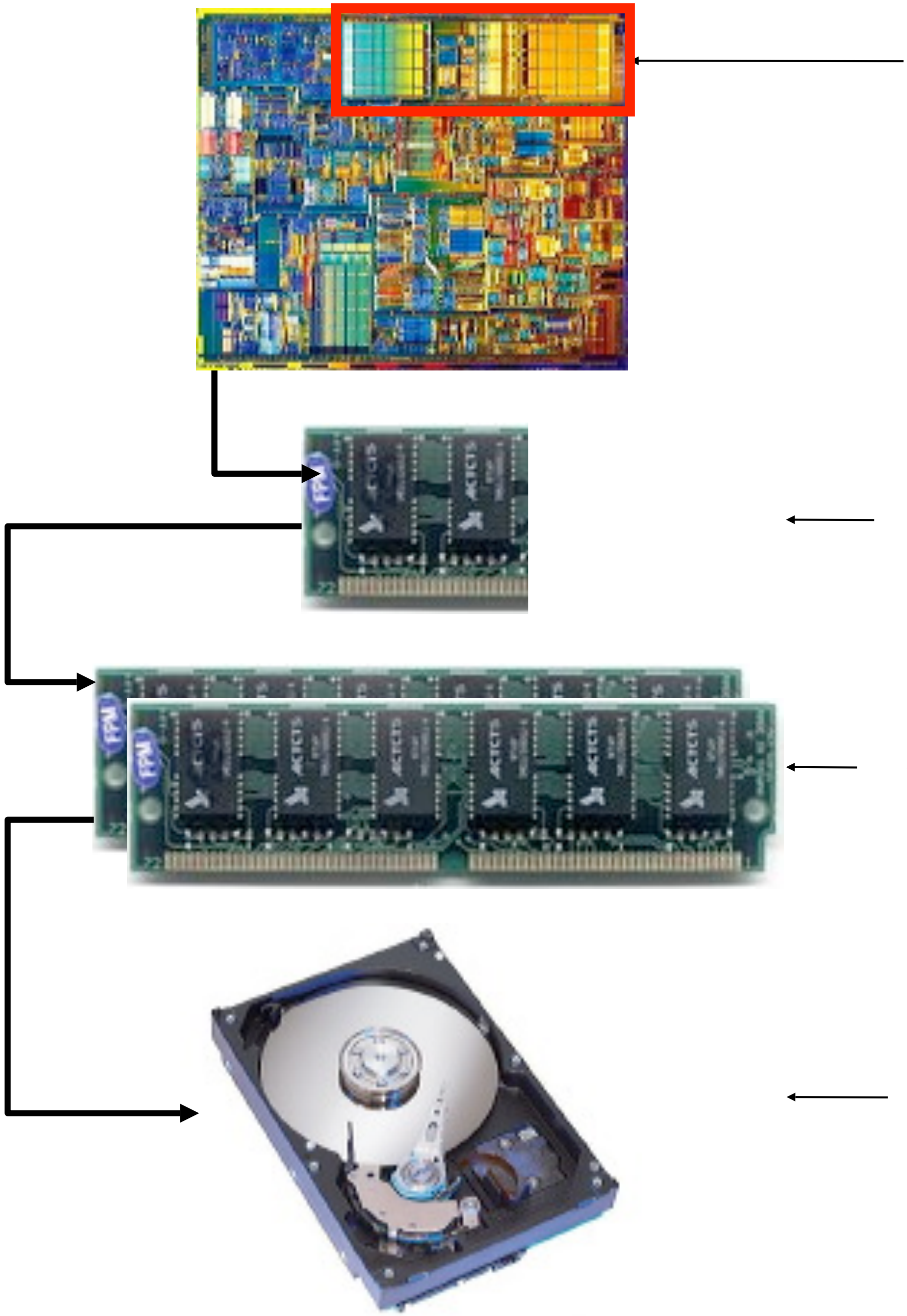


Core 2 Duo Die Photo



Memory Hierarchy

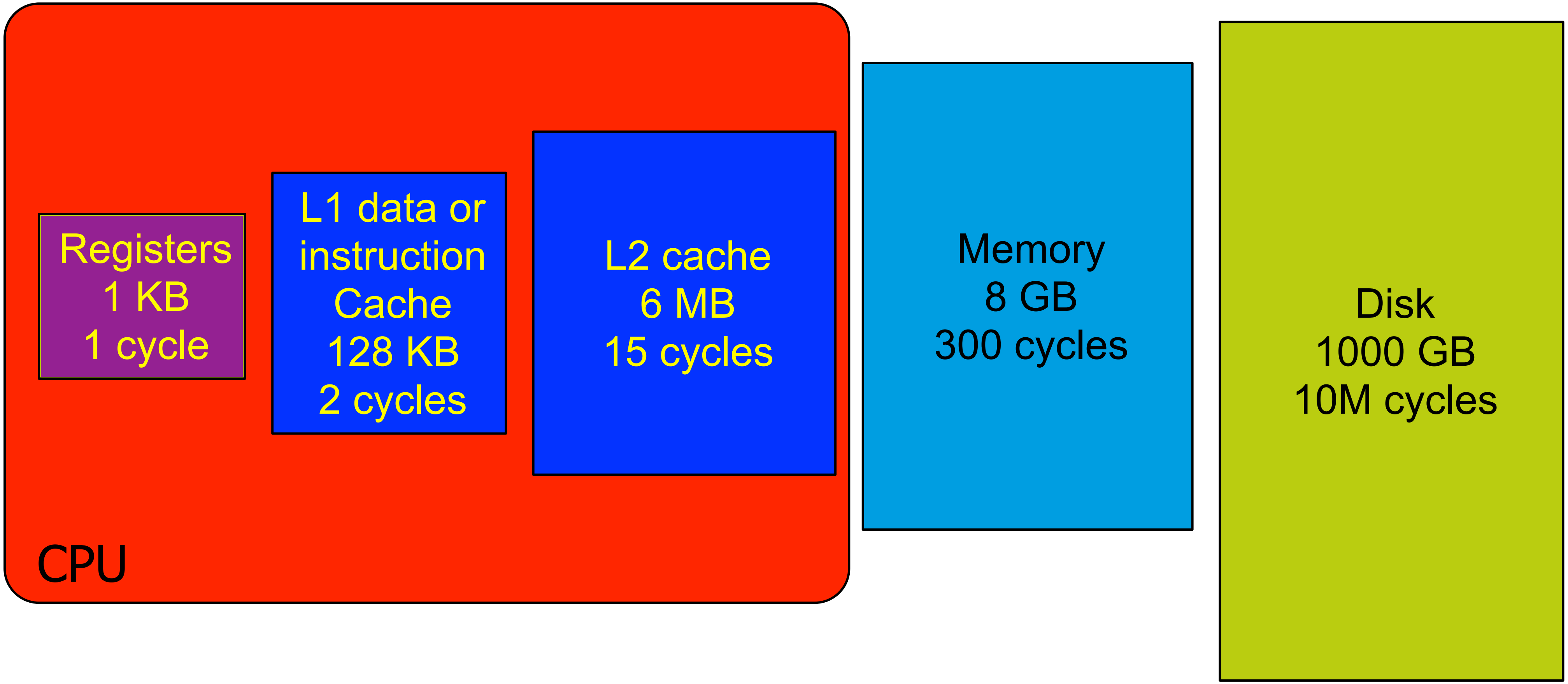




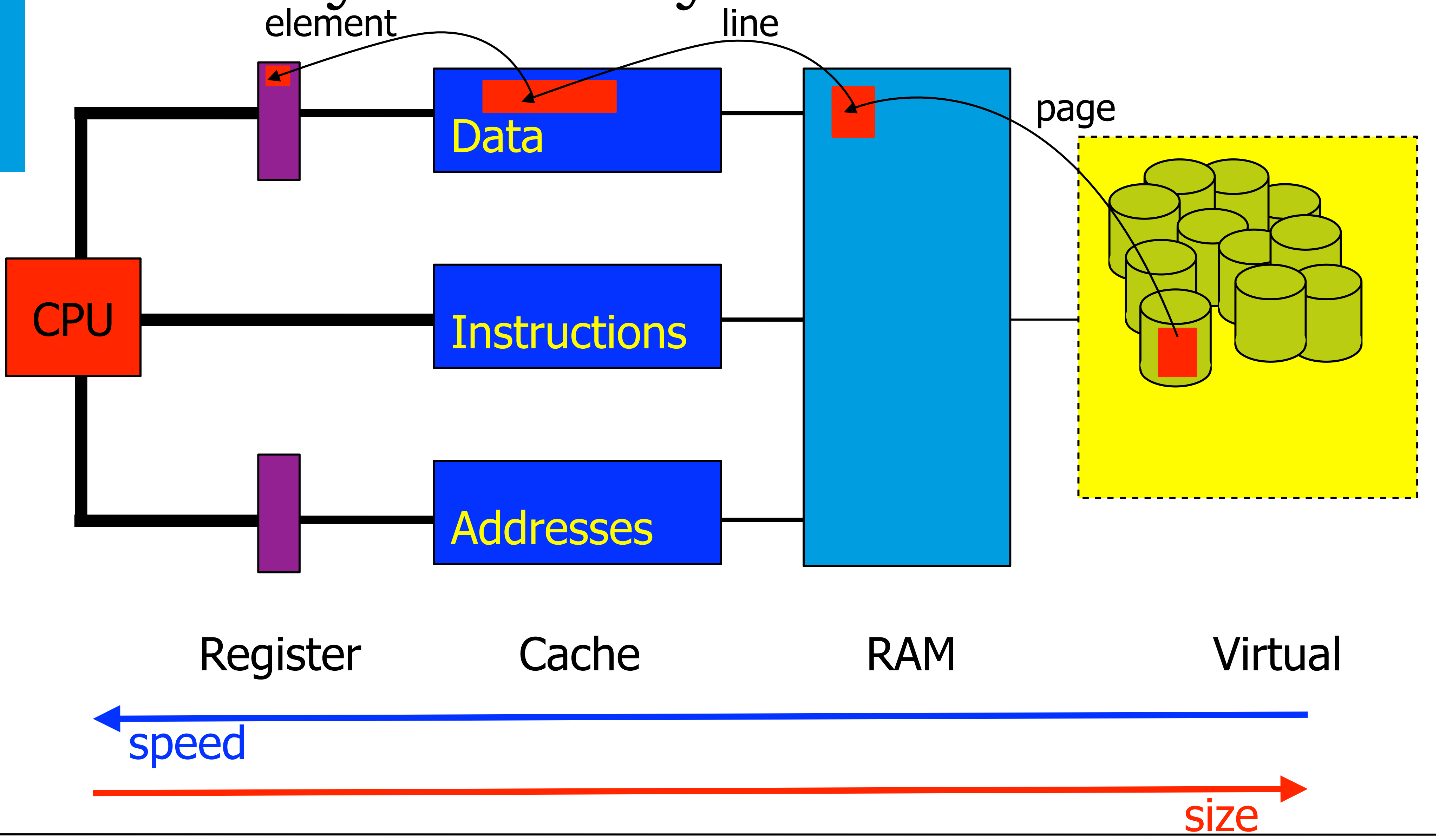
	Cost	Access time
on-chip cache KBs		< 1ns
off-chip cache MBs	2.5 $/MB	5ns
main memory GBs	0.07 $/MB	60ns
Disk TBs	0.0004 $/MB	10,000,000ns

Memory Hierarchy

As you go further, capacity and latency increase



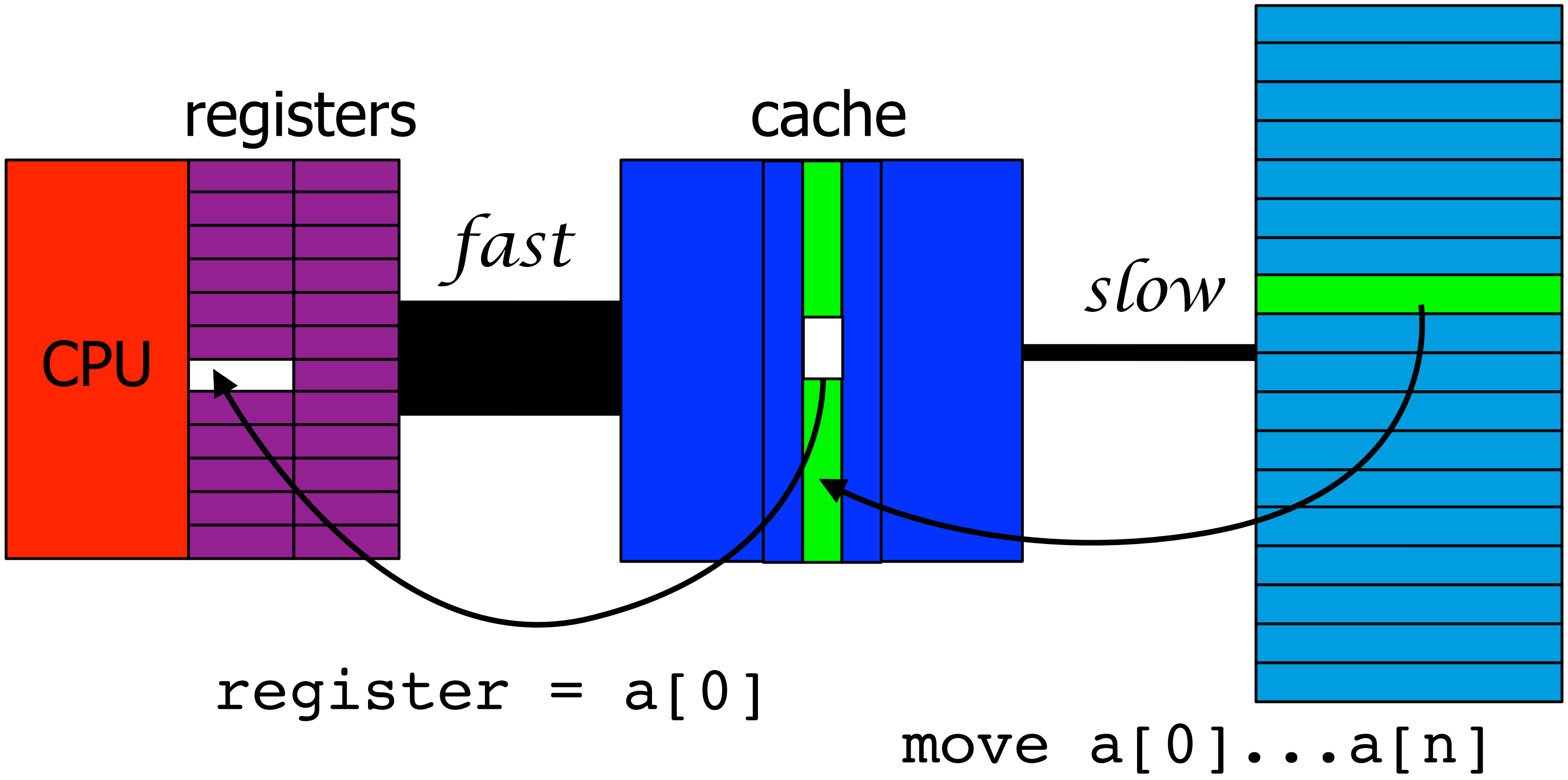
Memory Hierarchy



Cache Lines

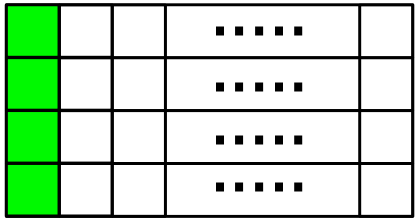
Typically more than one element at once is transferred

`x = a[0]`

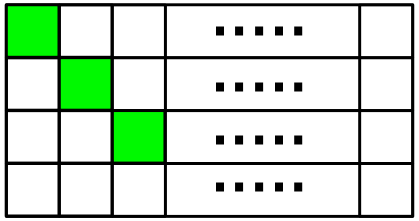


Cache lines

- The unit of transfer is called a cache line
- A cache line consists of consecutive memory locations



Like this



Not like this

- The size of a cache line is architecture dependent
  - AMD Barcelona/Shanghai 64 Bytes
  - Intel Nehalem 64 Bytes

Memory Level Issues

- Caches are working copies, true image is in main memory

- Cache exploits temporal proximity
  - recent data/instruction likely to be used again

- Where does true image of data/code reside?
  - When cache is written to, how is memory image updated?

- A cache is not big enough to store all data
  - How is cache organized and addressed?

- How is data replaced within cache?
  - This is called cache replacement policy, and will be discussed next

Direct Mapped Caches

- Replacement Formula:

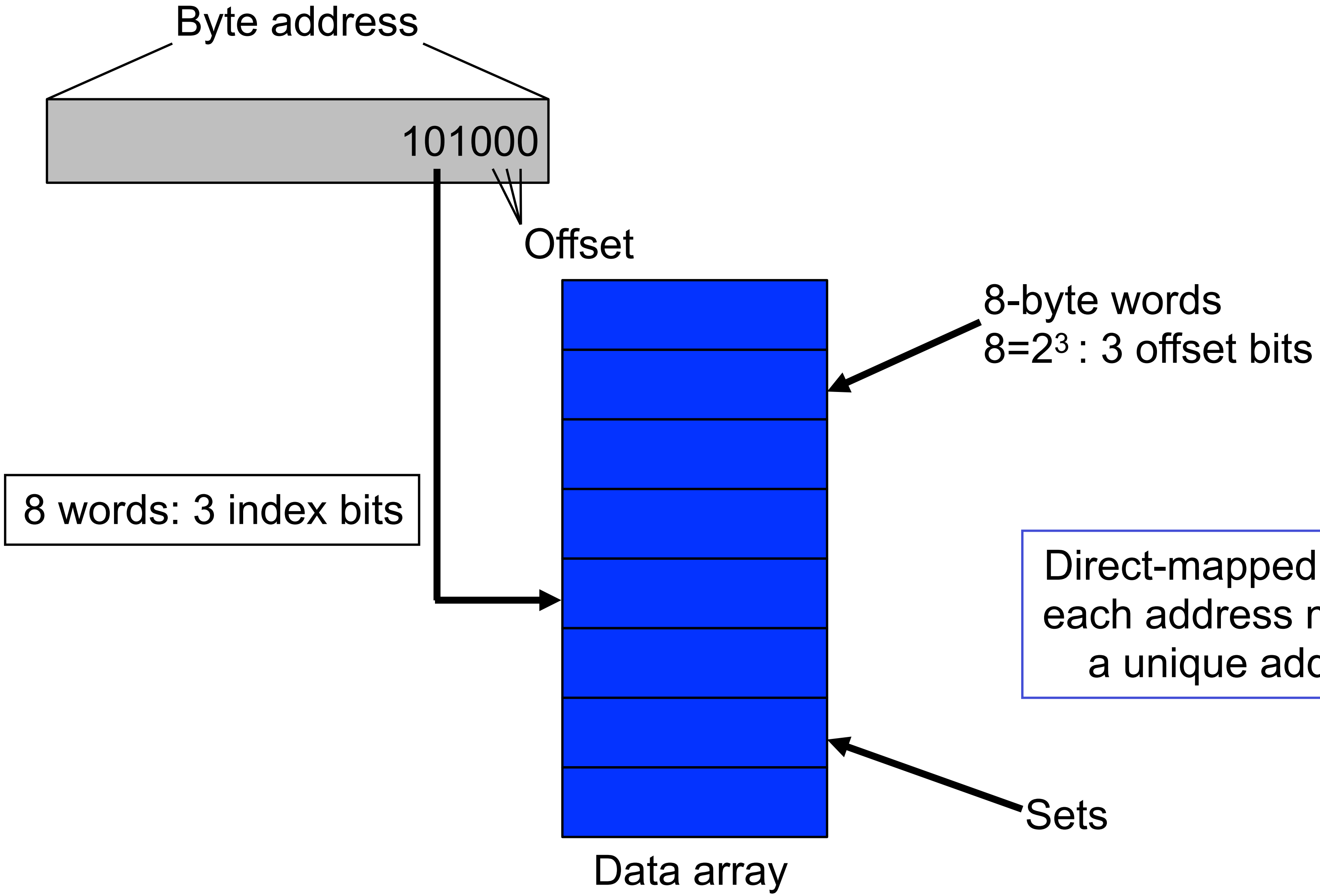
cache location = (memory address) modulo (cache size in lines)

- This maps a memory location from main memory directly to a position in the cache.

Cache, Example

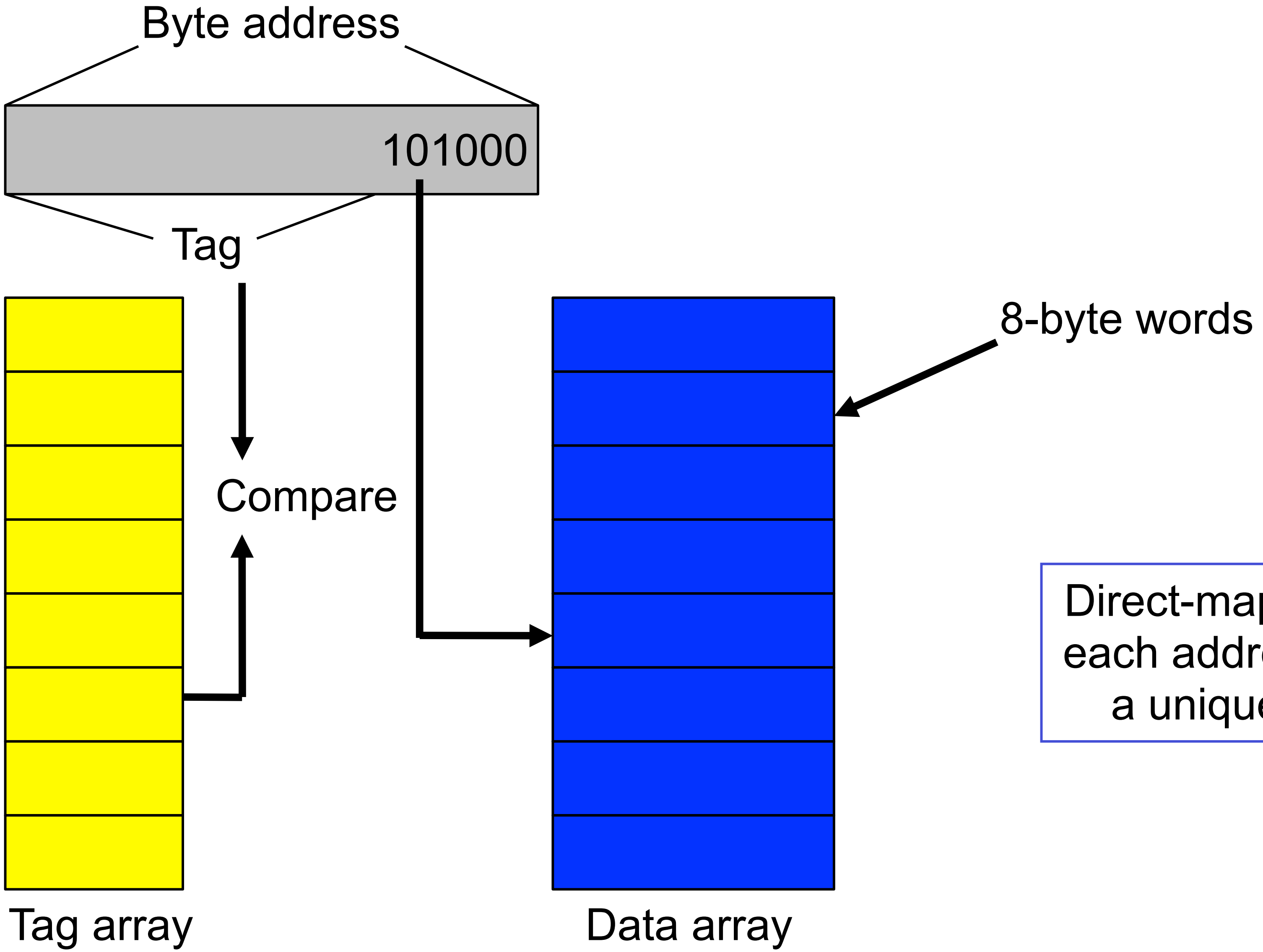
- 64 byte cache-size
- Each Cache “line” or “block” holds one word (8 bytes)
  - total cache can store 8 words (=64 bytes)
- Byte in cache is addressed by lowest three bits of address
- Cache line is addressed by next 3 bits in address
- Each Cache line has a “tag” matching the remaining 26 bits of the memory address

Accessing the Cache



Direct-mapped cache: each address maps to a unique address

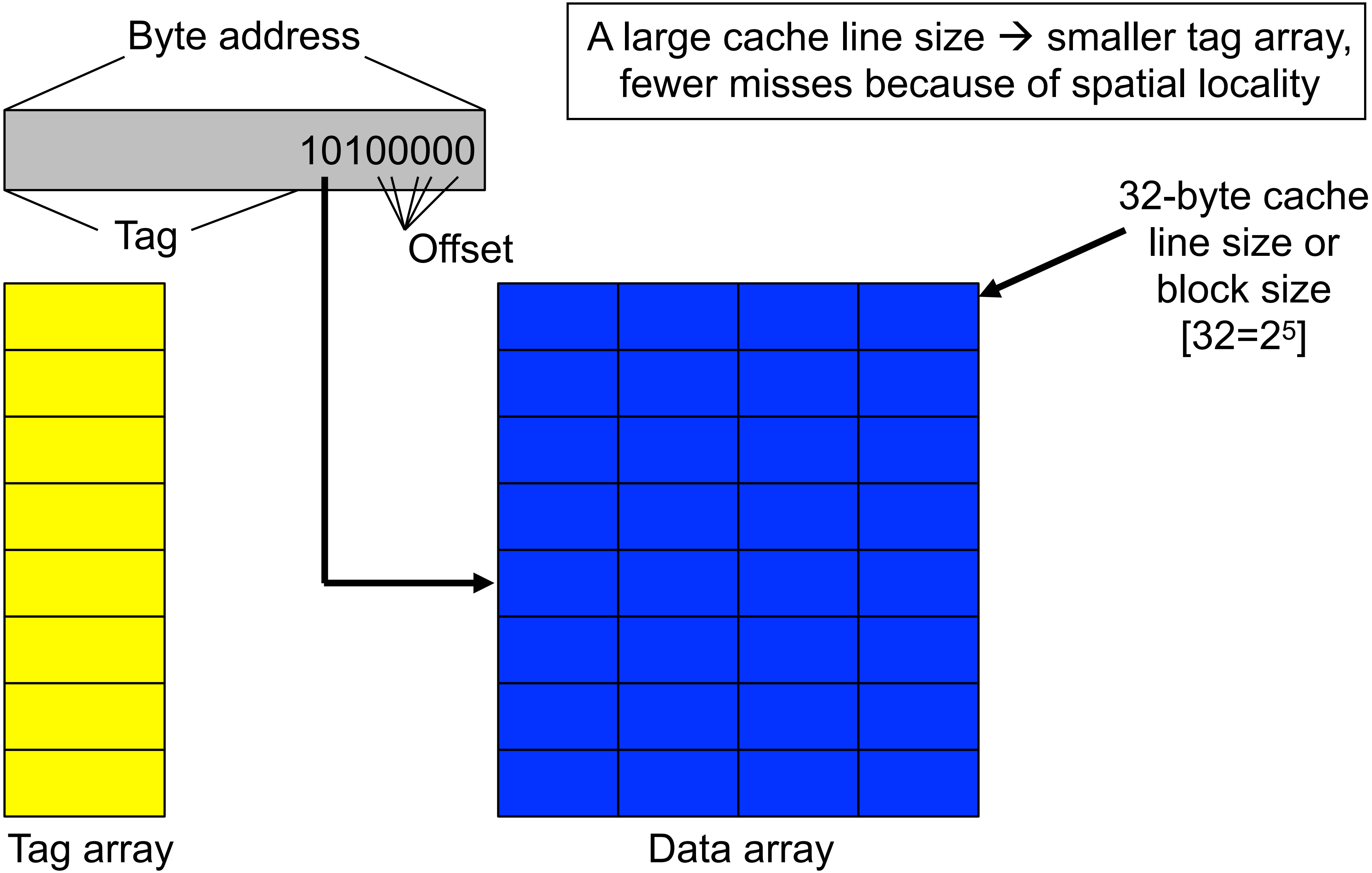
The Tag Array



Direct-mapped cache: each address maps to a unique address

Increasing Line Size

A large cache line size → smaller tag array, fewer misses because of spatial locality



Direct Mapped Caches

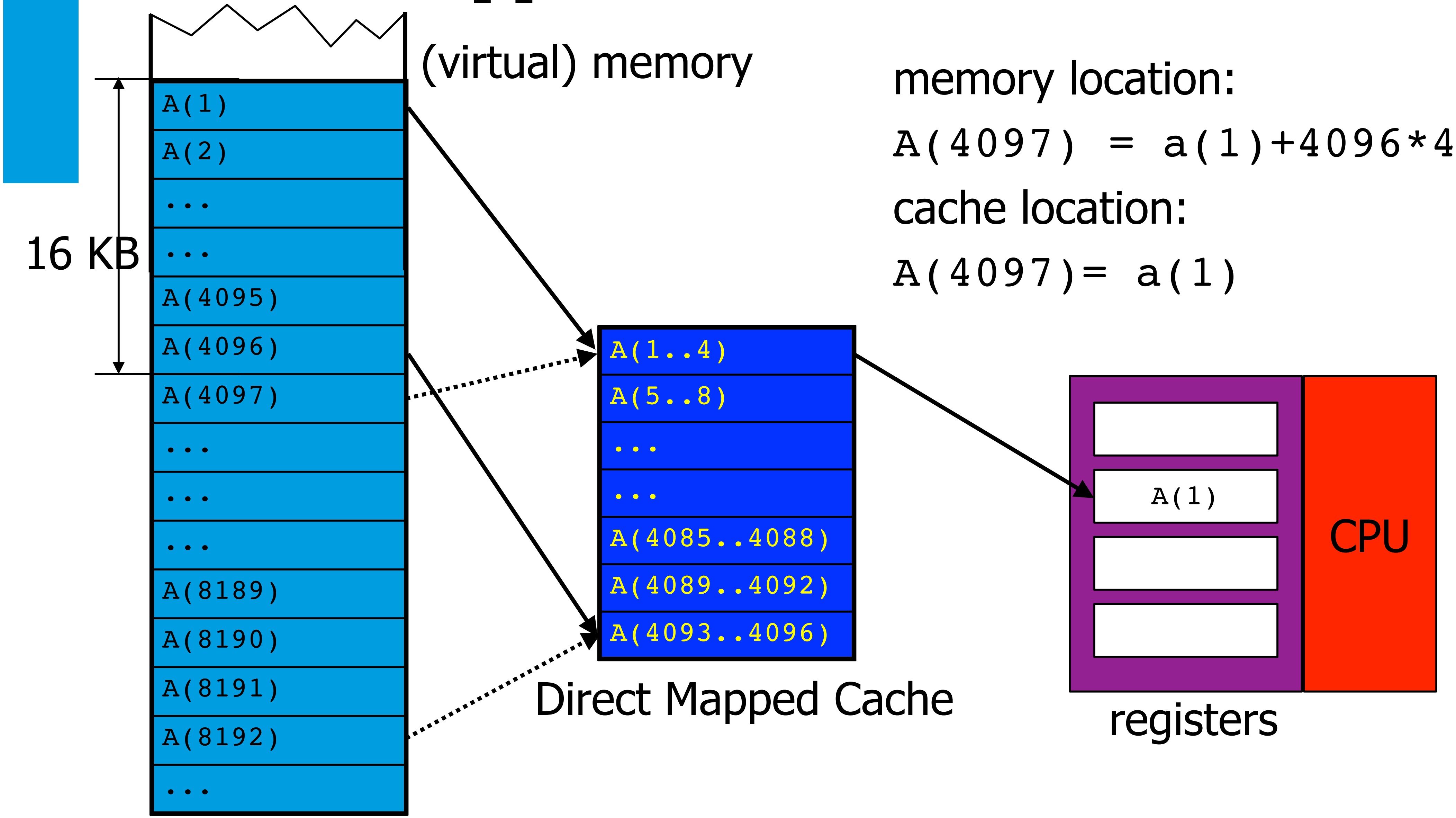
- Replacement Formula:

cache location = (memory address) modulo (cache size in lines)

An example:

- Assume that a cache line is 4 words (=16 Bytes)
- Cache size = 16 KB = 16 (line size) * 1024 (# of lines) Bytes
- This corresponds to 4096 32-bit words
- Example: element 5000 goes to cache line ((5000%1024)%4=226)
- We have to load an array A with 8192 32-bit elements:
  i.e. twice the size of the cache

Direct Mapped Caches



memory location:

A(4097) = a(1)+4096*4

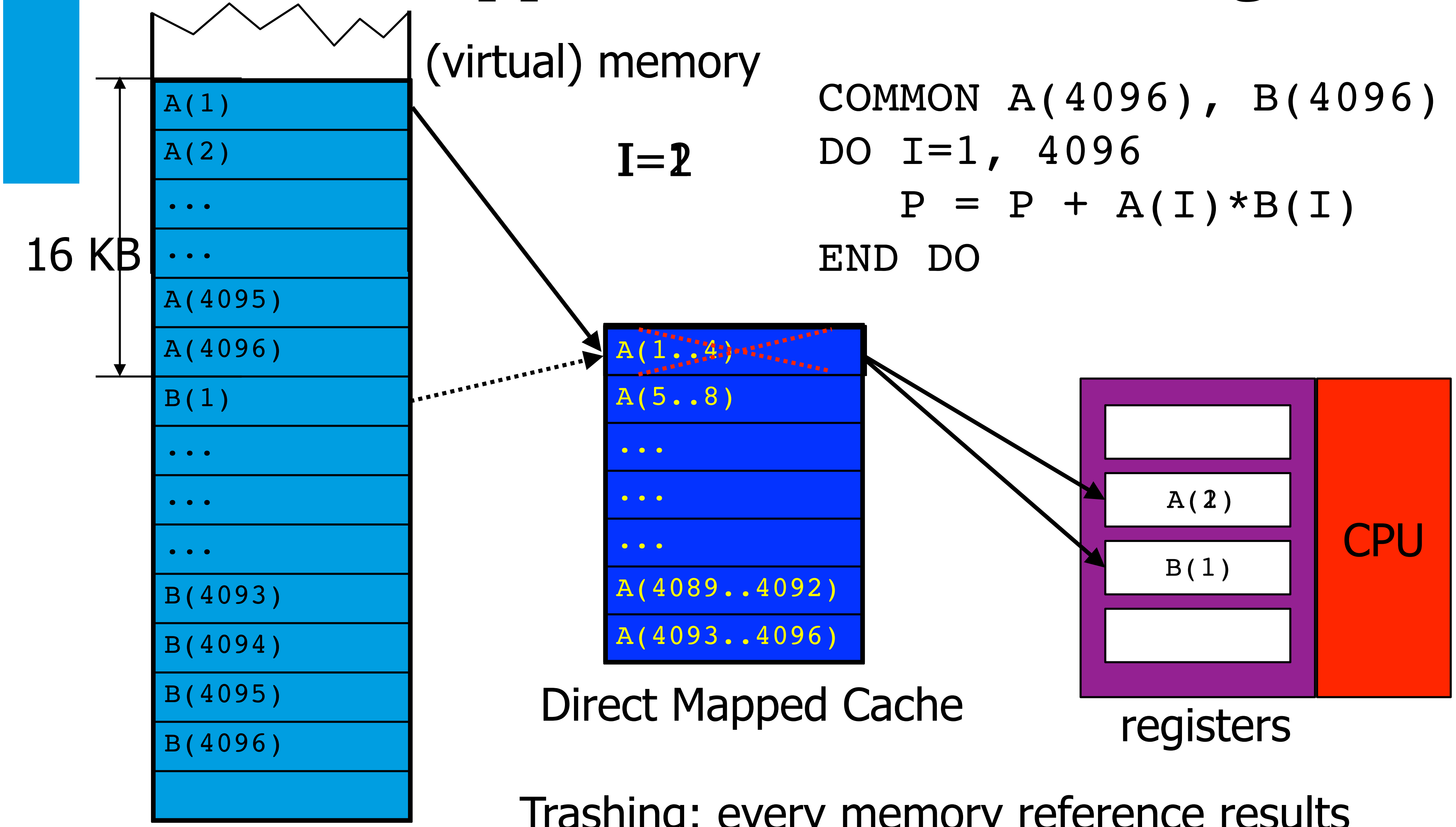
cache location:

A(4097)= a(1)

Direct mapped caches - Trashing

- A well known side-effect of this design: data elements that are soon needed are overwritten (trashing)
- Especially when multiple arrays are involved direct mapping can become very inefficient
- Often the only remedy is to modify the memory mapping, but this can be non-trivial
- There is a solution....

Direct Mapped Caches - Trashing



```
COMMON A(4096), B(4096)
DO I=1, 4096
   P = P + A(I)*B(I)
END DO
```

Trashing: every memory reference results in a cache miss

Fully Associative Caches

Fully Associative:

- The replacement is now based upon a Least-Recently Used (LRU) algorithm:
  - Data that is oldest (touched) is removed
  - In many cases it makes sense to do
  - It greatly helps when working with multiple arrays
  - Takes longer time to find if a line is already in the cache

Fully Associative Caches

- Why are not all caches then fully associative?

Because of cost

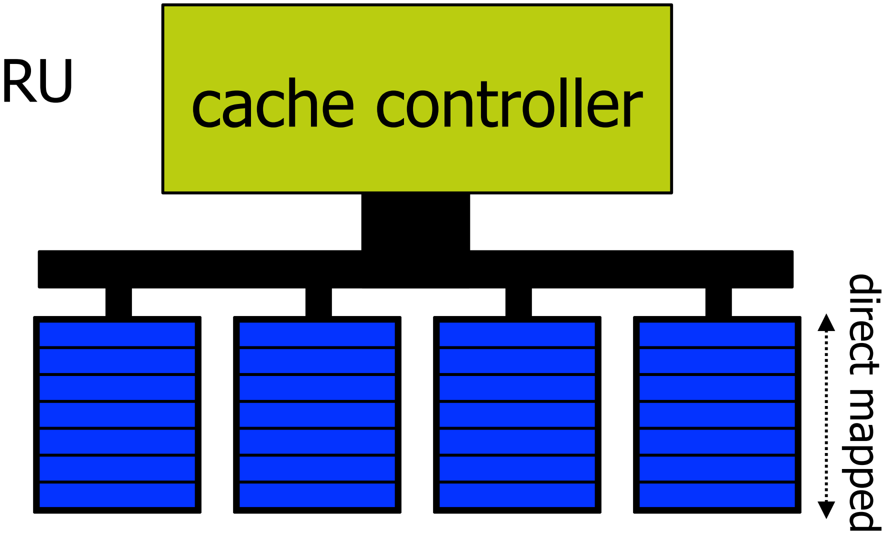
- Luckily, a clever alternative (polder model) exists.

Set Associative Caches

Set Associative:

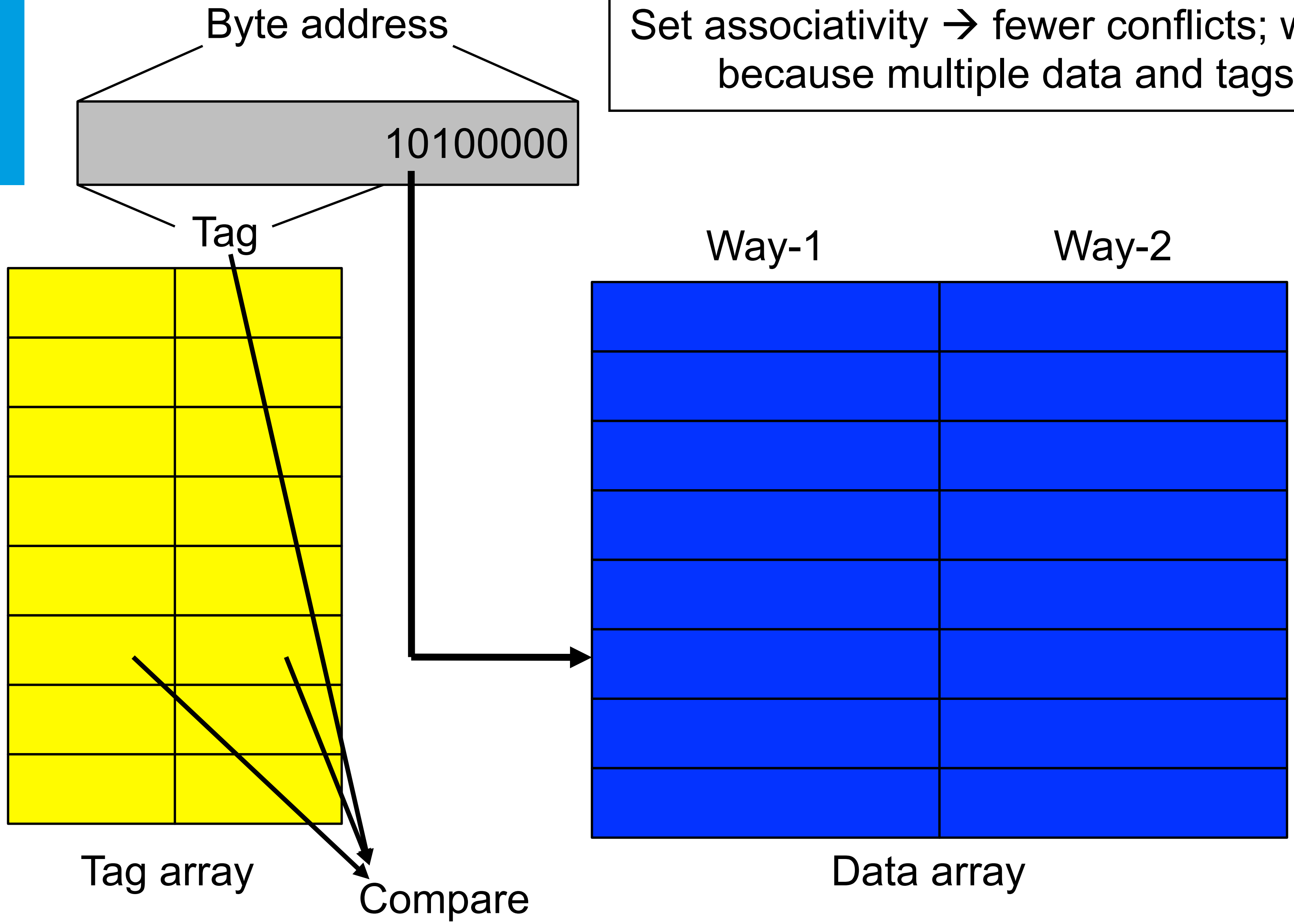
- The cache contains several direct mapped caches
- Data can go into one of these caches (called a 'set')
- The choice of a set is often (semi-) LRU

4-way set associative cache



Associativity

Set associativity → fewer conflicts; wasted power because multiple data and tags are read

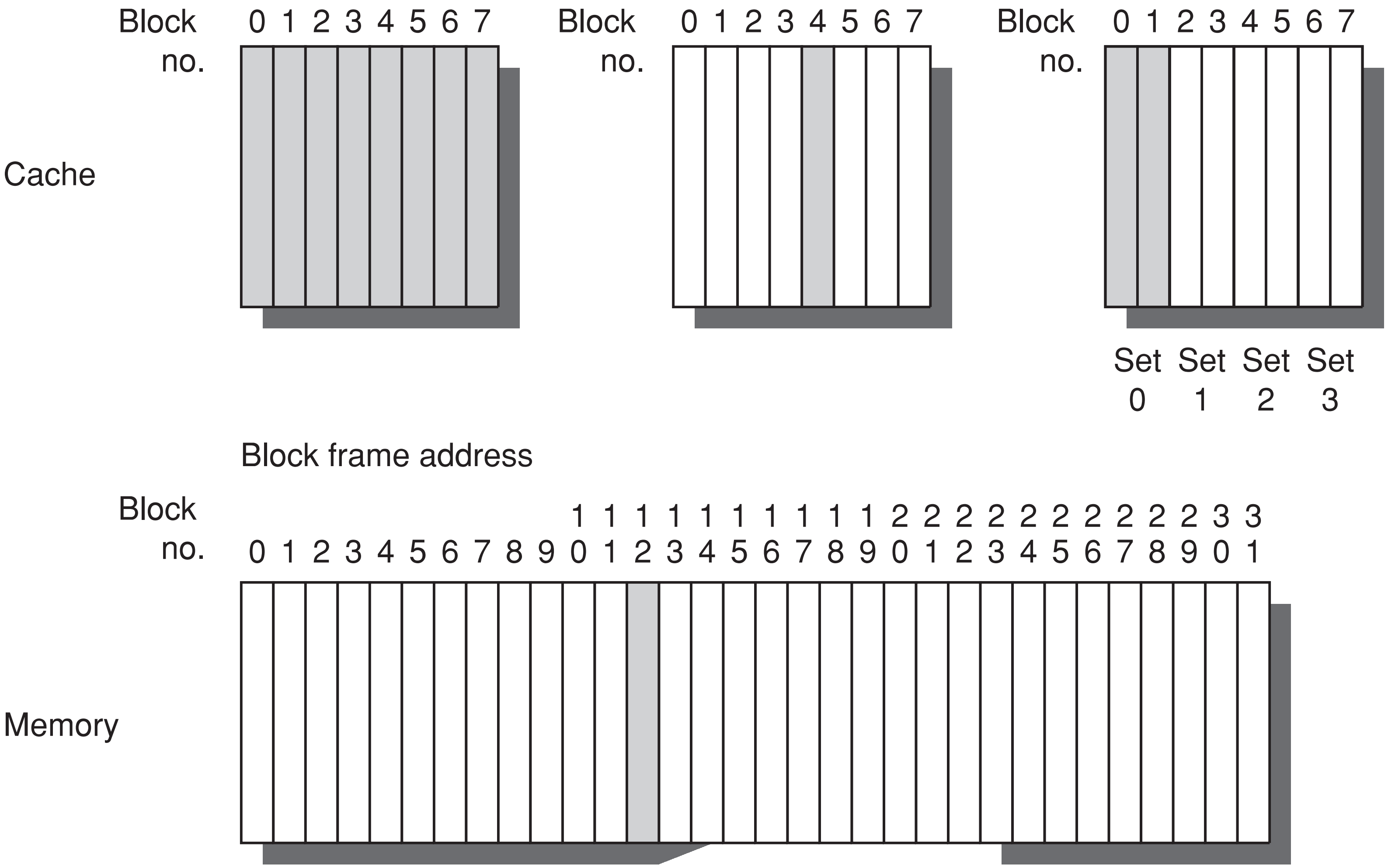


trashing can still occur within one set !

Fully associative:
block 12 can go
anywhere

Direct mapped:
block 12 can go
only into block 4
(12 mod 8)

Set associative:
block 12 can go
anywhere in set 0
(12 mod 4)



- **Direct Mapped Cache:** The simplest way to allocate the cache to the system memory is to determine how many cache lines there are (16,384 in our example) and just chop the system memory into the same number of chunks. Then each chunk gets the use of one cache line. This is called *direct mapping*. So if we have 64 MB of main memory addresses, each cache line would be shared by 4,096 memory addresses (64 M divided by 16 K).
- **Fully Associative Cache:** Instead of hard-allocating cache lines to particular memory locations, it is possible to design the cache so that any line can store the contents of any memory location. This is called *fully associative mapping*.
- **N-Way Set Associative Cache:** "N" here is a number, typically 2, 4, 8 etc. This is a compromise between the direct mapped and fully associative designs. In this case the cache is broken into sets where each set contains "N" cache lines, let's say 4. Then, each memory address is assigned a set, and can be cached in any one of those 4 locations within the set that it is assigned to. In other words, *within each set* the cache is associative, and thus the name.

  This design means that there are "N" possible places that a given memory location may be in the cache. The trade-off is that there are "N" times as many memory locations competing for the same "N" lines in the set. Let's suppose in our example that we are using a 4-way set associative cache. So instead of a single block of 16,384 lines, we have 4,096 sets with 4 lines in each. Each of these sets is shared by 16,384 memory addresses (64 M divided by 4 K) instead of 4,096 addresses as in the case of the direct mapped cache. So there is more to share (4 lines instead of 1) but more addresses sharing it (16,384 instead of 4,096).

Cache Mapping

Cache Type	**Hit Ratio**	**Search Speed**
Direct Mapped	Good	Best
Fully Associative	Best	Moderate
N-way Set Associative (N>1)	Very Good Better as N increases	Good Worse as N increases

Multi-Level Caches

- If one works well, why not use the same trick again !

- The L2 and L3 have properties that are different from L1
  - access time is not as critical for L2 as it is for L1 (every load/store/instruction accesses the L1)
  - the L2 is much larger and can consume more power per access

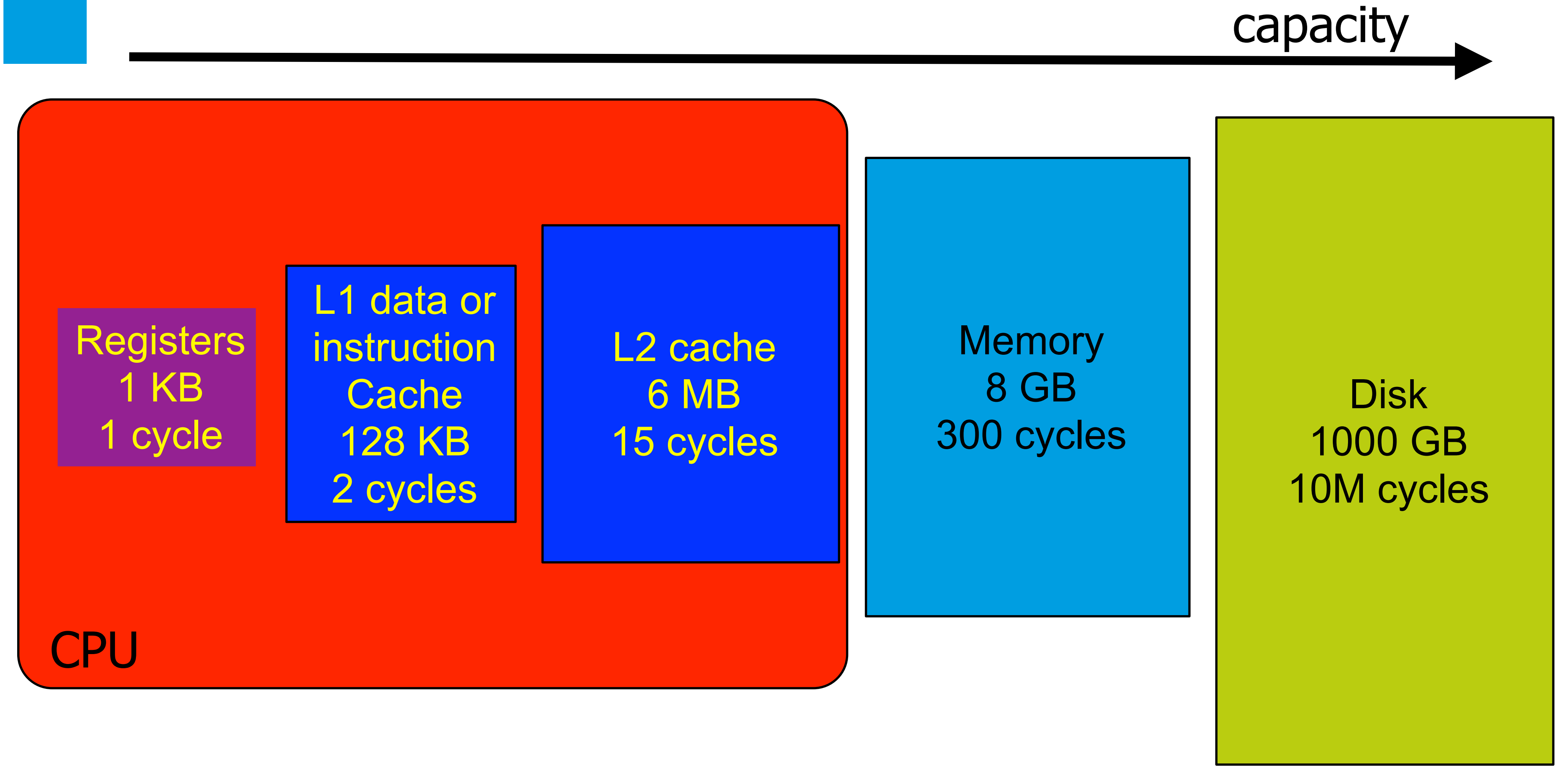
- Hence, they can adopt alternative design choices
  - serial tag and data access
  - high associativity

Cache levels

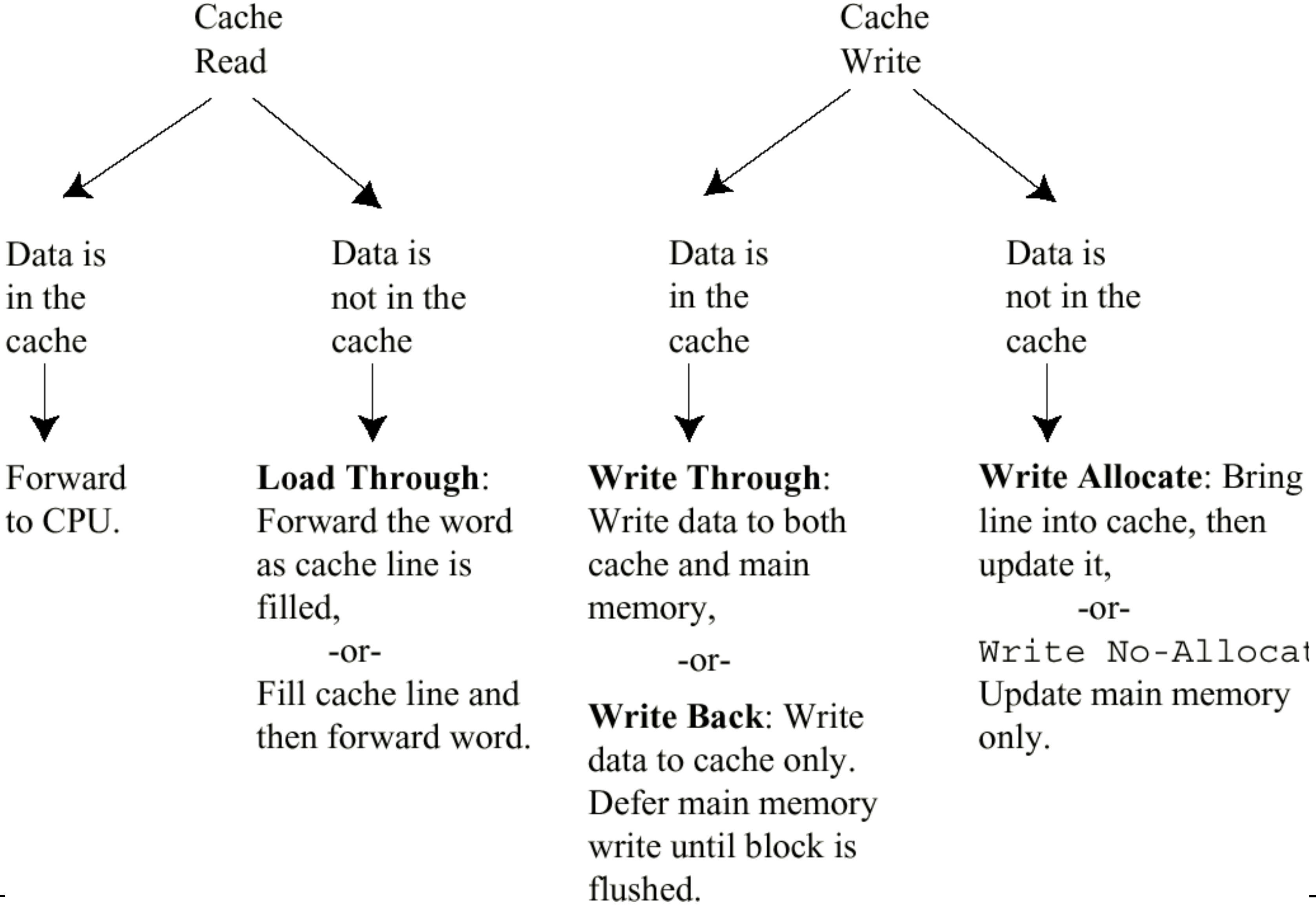
- L1
  - design for minimal hit time
- L2
  - design for low miss rate to avoid access to main memory
- L3
  - design for sharing with other cores
- L1 usually smaller than L2
- L1 block size (number of sets) smaller than L2
- L3 much larger than L1

Memory Hierarchy

As you go further away from the CPU, capacity and latency increase



Cache Read and Write Policies



Cache Read

Data is in the cache

Forward to CPU.

Data is not in the cache

Load Through: Forward the word as cache line is filled,

-or-

Fill cache line and then forward word.

Cache Write

Data is in the cache

Write Through: Write data to both cache and main memory,

-or-

Write Back: Write data to cache only. Defer main memory write until block is flushed.

Data is not in the cache

Write Allocate: Bring line into cache, then update it,

-or-

Write No-Allocat

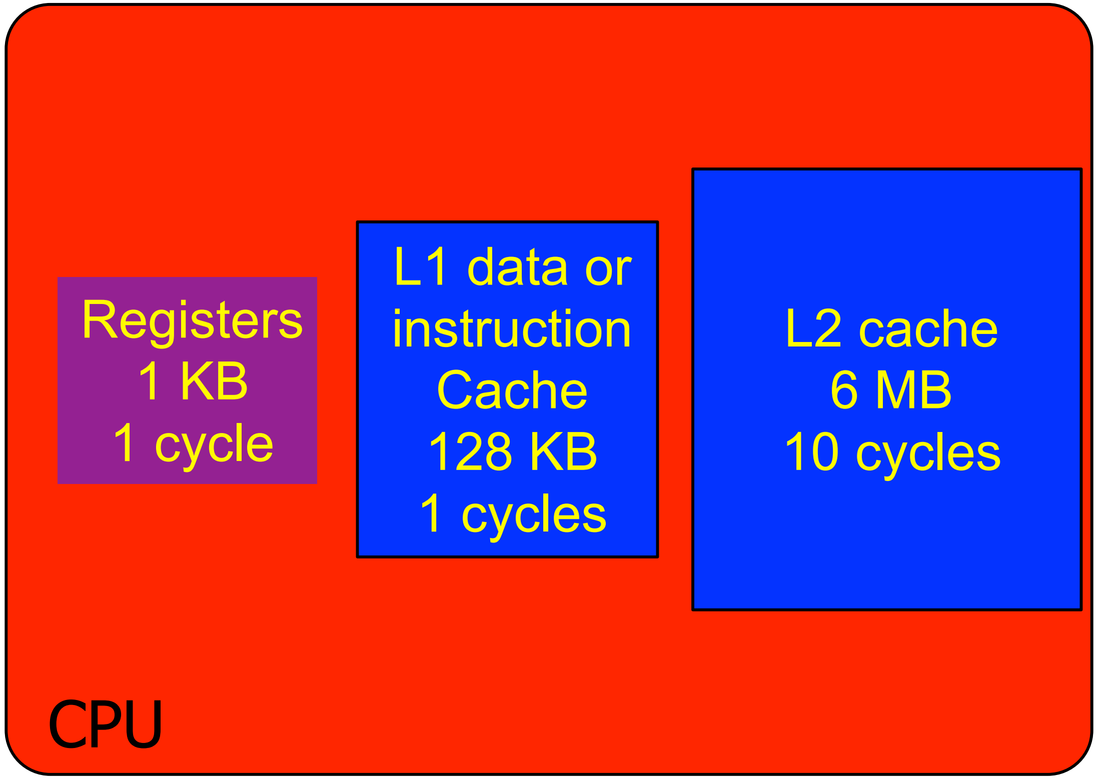
Update main memory only.

Cache Summary - hit/miss

- Cache Hit
  - Item is found in the cache
  - CPU continues at full speed
  - Need to verify valid and tag match

- Cache Miss
  - Item must be retrieved from memory
  - Whole Cache line is retrieved
  - CPU stalls for memory access

L1 cache hit rate



Load data 100 times

100%	hit rate in L1: 100 cycles	
99%	hit rate in L1: 109 cycles	9% slower
95%	hit rate in L1: 145 cycles	45% slower

Cache misses

- cache misses take time
  - cache filling and emptying takes take
  - getting data from main memory to L3, L2 and L1
  - replacement policy

- Can we do something else while waiting for data to arrive in the cache?

Tolerating Miss Penalty

- **Out of order execution**: can do other useful work while waiting for the miss – can have multiple cache misses – cache controller has to keep track of multiple outstanding misses (non-blocking cache)

- Hardware and software prefetching into prefetch buffers – aggressive prefetching can increase contention for buses

Those techniques will be discussed later today.

Optimal cache performance

Re-use data in the cache

- This is using temporal locality

Use all the data in one cache line

- This is using spatial locality

Time for some exercise

- Bandwidth and or Latency
- …./HPCourse/lat_mem_rd

One More Thing…. about memory

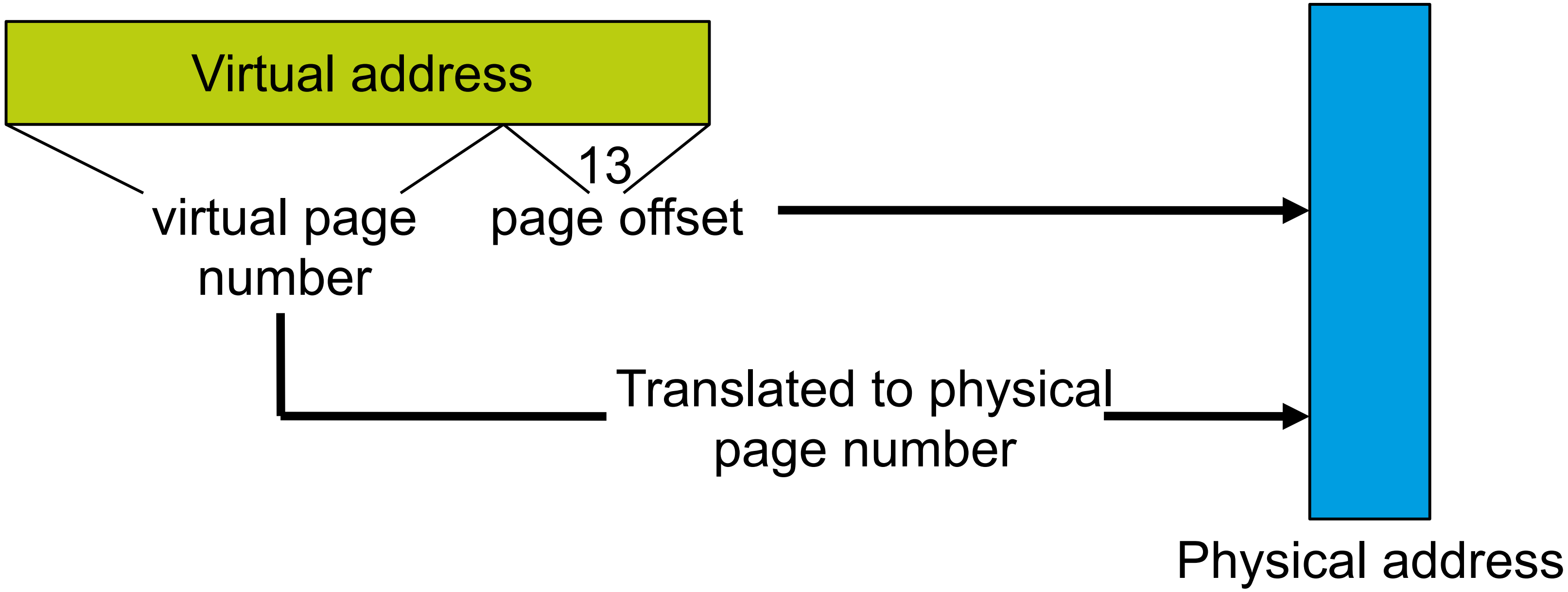
Virtual Memory

- Modern programs operate in “virtual memory”
  - Each program thinks it has all of memory to itself
  - Fixed sized blocks (“pages”) vs variable sized blocks (“segments”)
- Virtual Memory benefit
  - Allow a program that is larger than physical memory to run
  - Programmer does not have to manually create overlays
  - Allow many programs to share limited physical memory
- Virtual Memory asks for additional work:
  - Each virtual memory reference must be translated into a physical memory reference

Address Translation

- The virtual and physical memory are broken up into pages

8KB page size



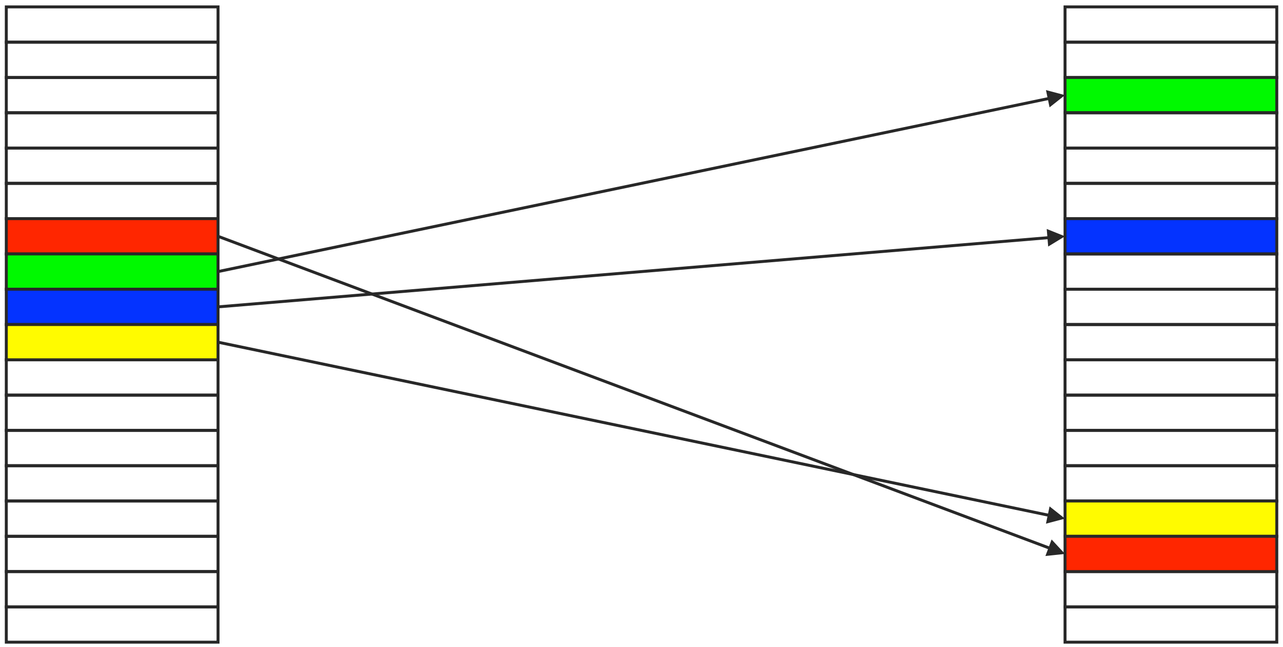
Basic page has a size of 4 KB

Virtual and Physical memory

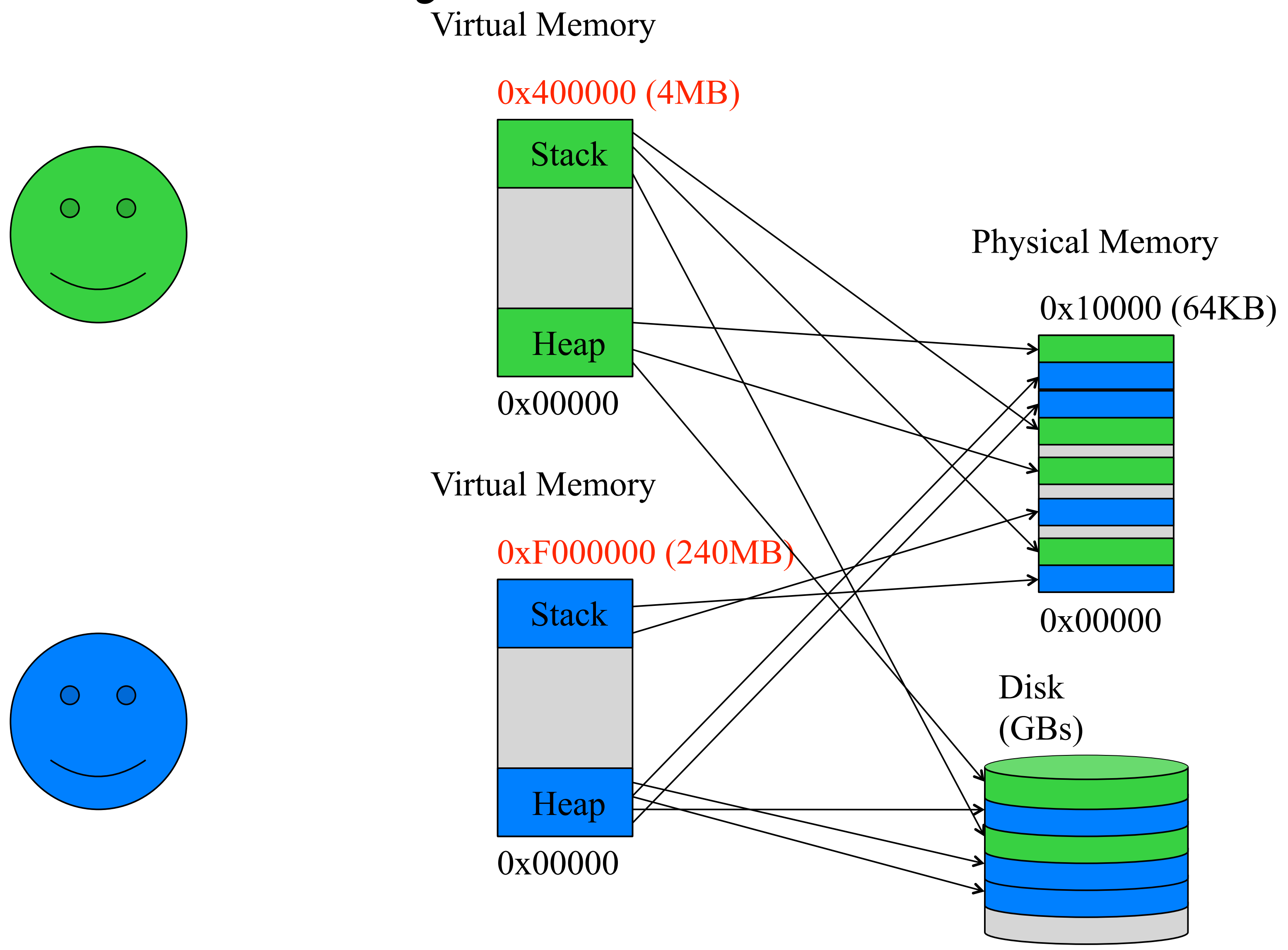
Virtual Memory

Address Translation

Physical Memory



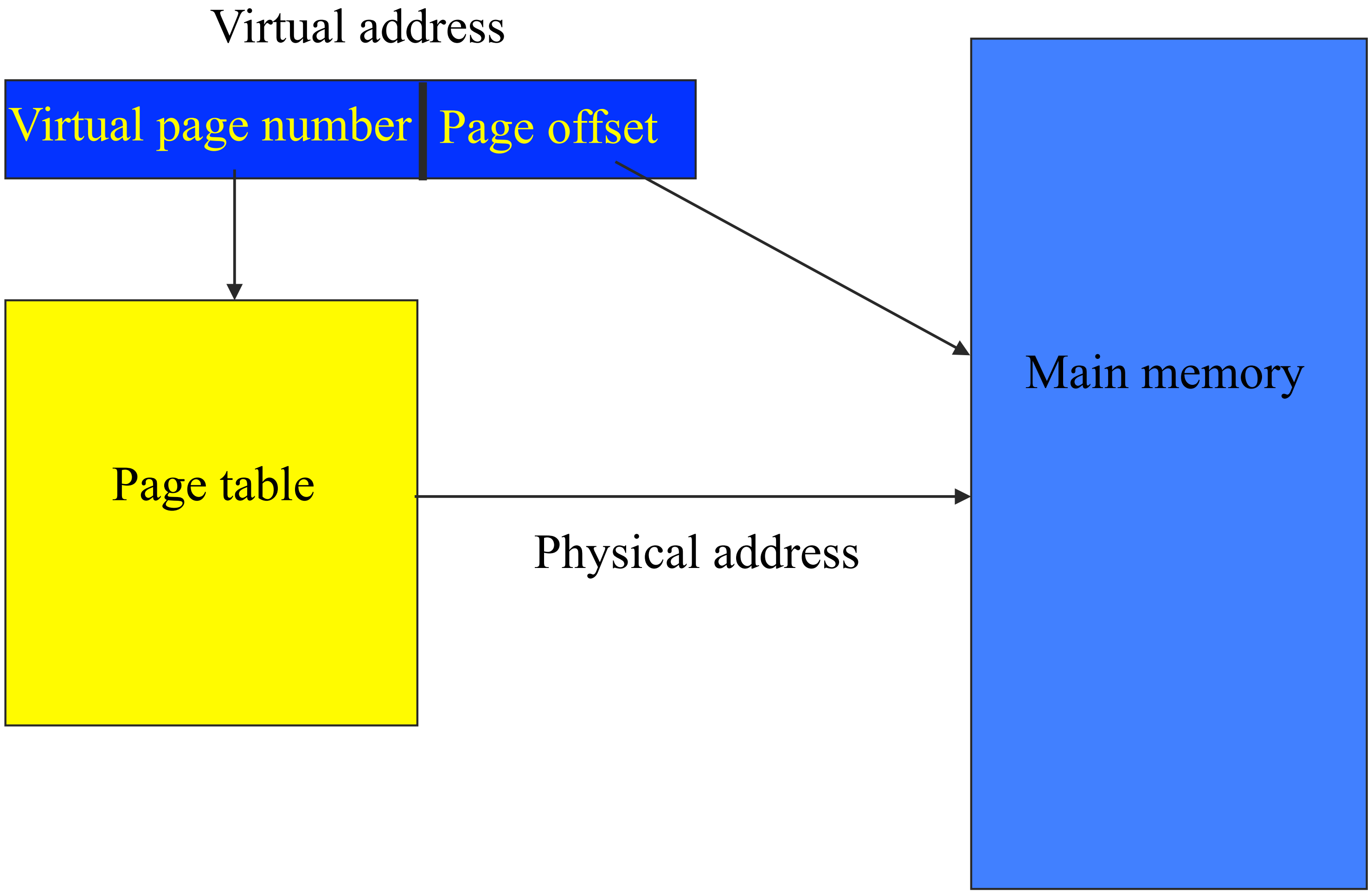
Virtual memory



The TLB cache

- In a virtual memory based system, the **virtual address** needs to be translated to a **physical address** by the kernel.
- This address translation is typically a costly operation
- Therefore translations are:
  - Performed on a virtual memory page basis
  - Buffered in a cache (with the hope to re-use them)
- This cache is often called Translation Lookaside Buffer or TLB for short

Look-up Table



Virtual address

Virtual page number

Page offset

Page table

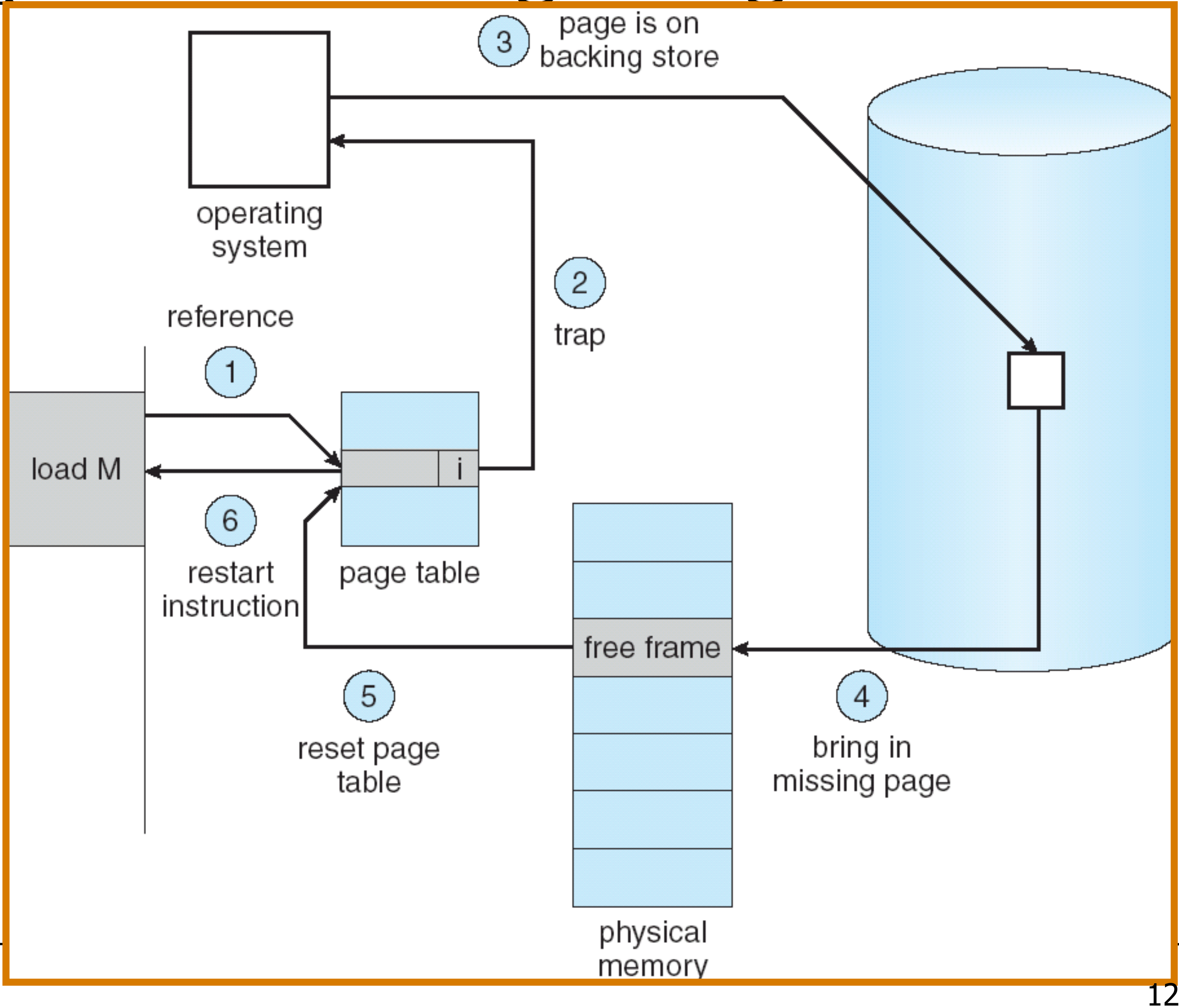
Main memory

Physical address

Typical TLB size

- Size: 8 - 4,096 entries
- Hit time: 1 clock cycle
- Miss penalty: 10 - 100 clock cycles
- Miss rate: 0.01 - 1%
- Larger pages allow:
  - Wider memory coverage
  - With fewer address translations
- Varying pages sizes can lead to fragmentation
  - E.g., a memory-hungry program might work with 2 MB pages, but can oversubscribe available memory with 16 MB pages

Steps in Handling a Page Fault

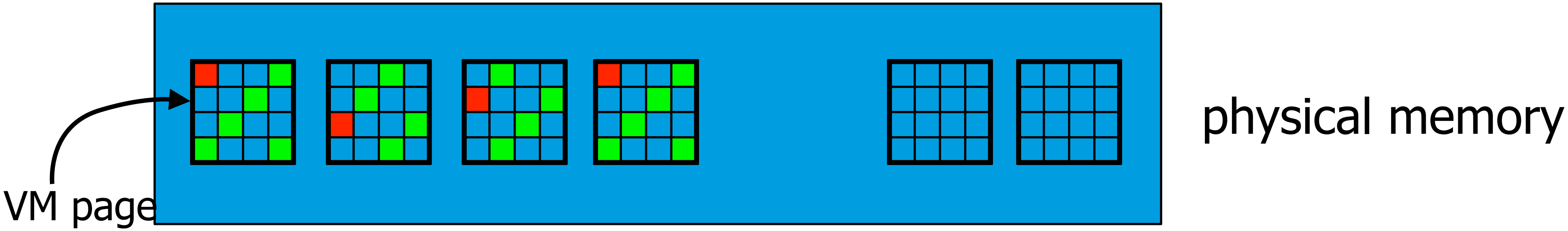


The TLB cache

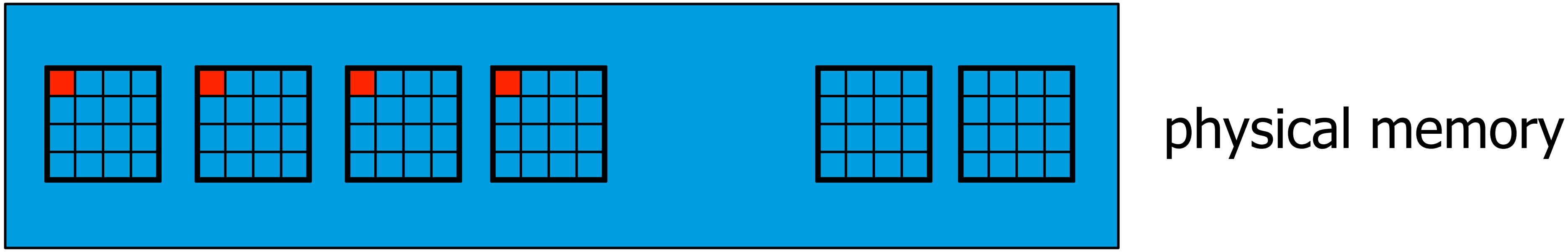
🟥 = new TLB entry created

🟩 = address already mapped

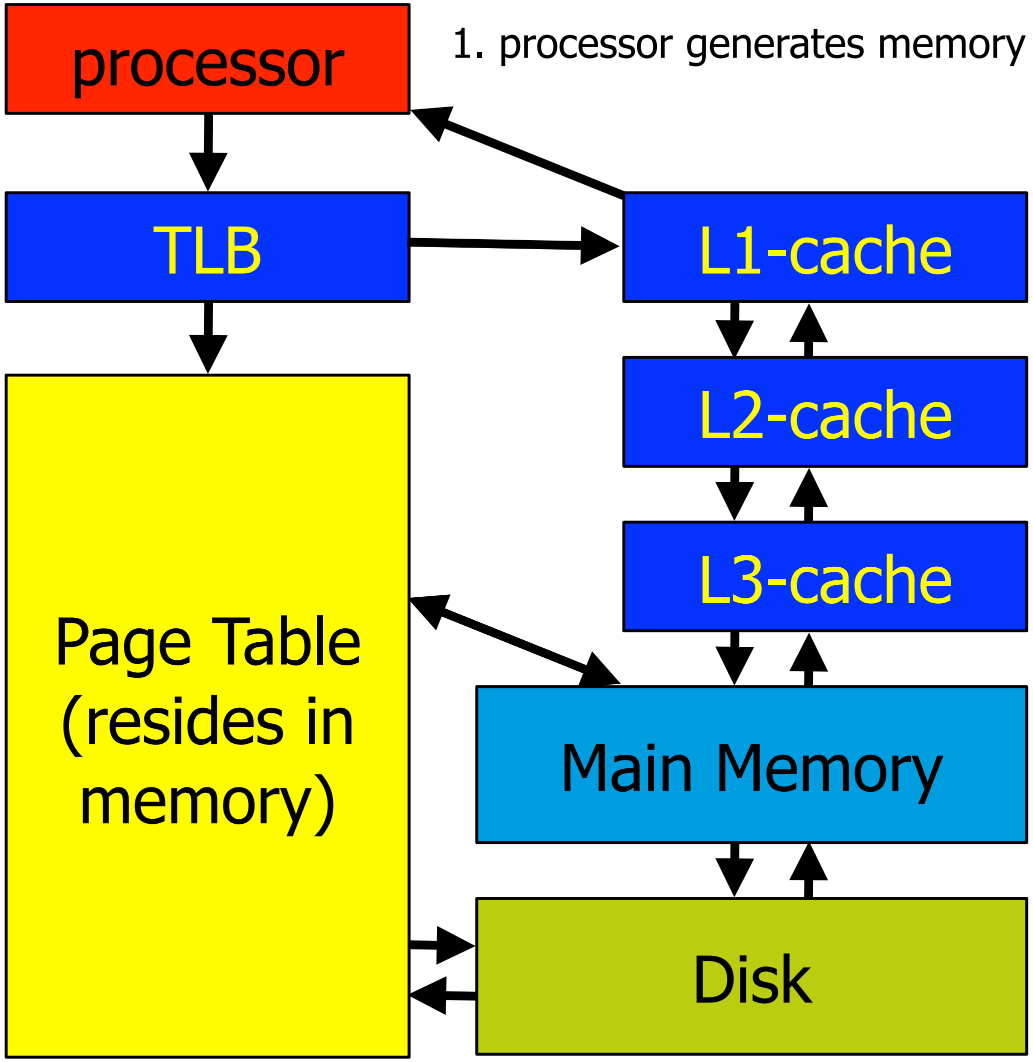
bad for the TLB
non unit stride through the data



VERY bad for the TLB
strides through the data which exceed the page size



Putting it all together



1. processor generates memory reference (ld or st)

2. check TLB if virtual mapping is present

3. If virtual mapping is not present, go to page table, retrieve mapping, and update TLB with the mapping

4. L1 cache miss

5. L2 cache miss

6. L3 cache miss

7. main memory hit

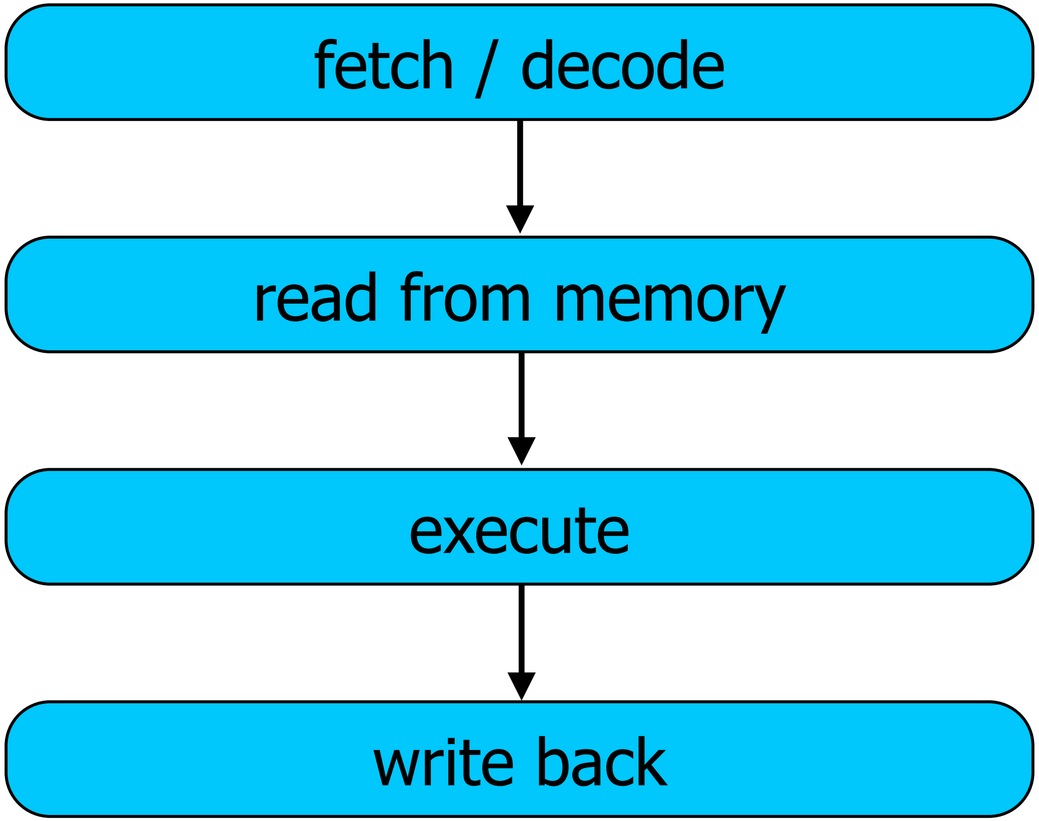
8. If there is a main memory miss, then a page fault would be generated.

Going back to the CPU

Tricks a modern CPU can do

- Instruction Level Parallelism
  - Superscalar
  - Out of Order Execution
  - Pipelining
  - Branch Prediction
  - vector instructions
- Multithreading
- Pre Fetching

Fetch-Execution cycle



fetch / decode

read from memory

execute

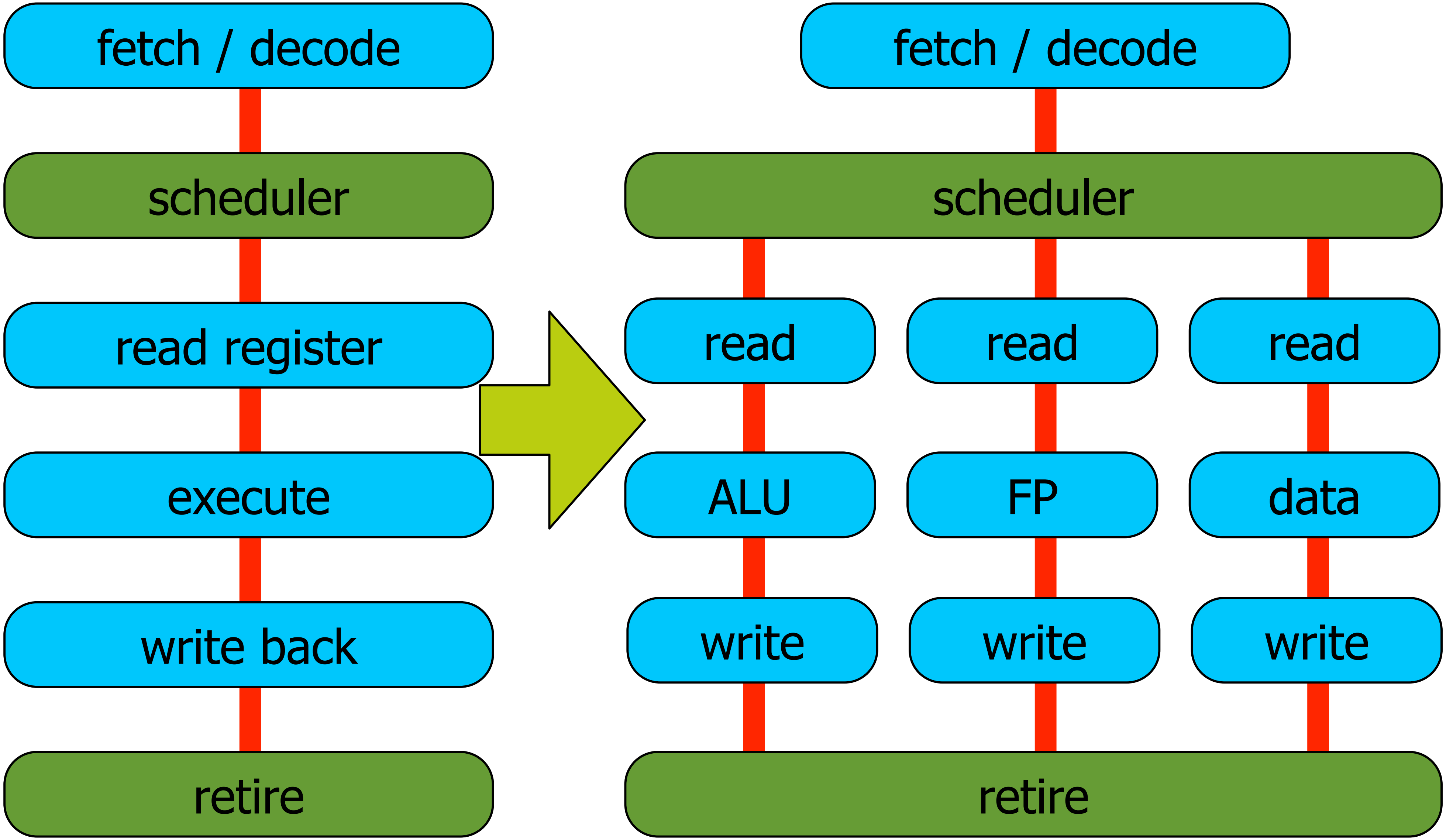
write back

Superscalar

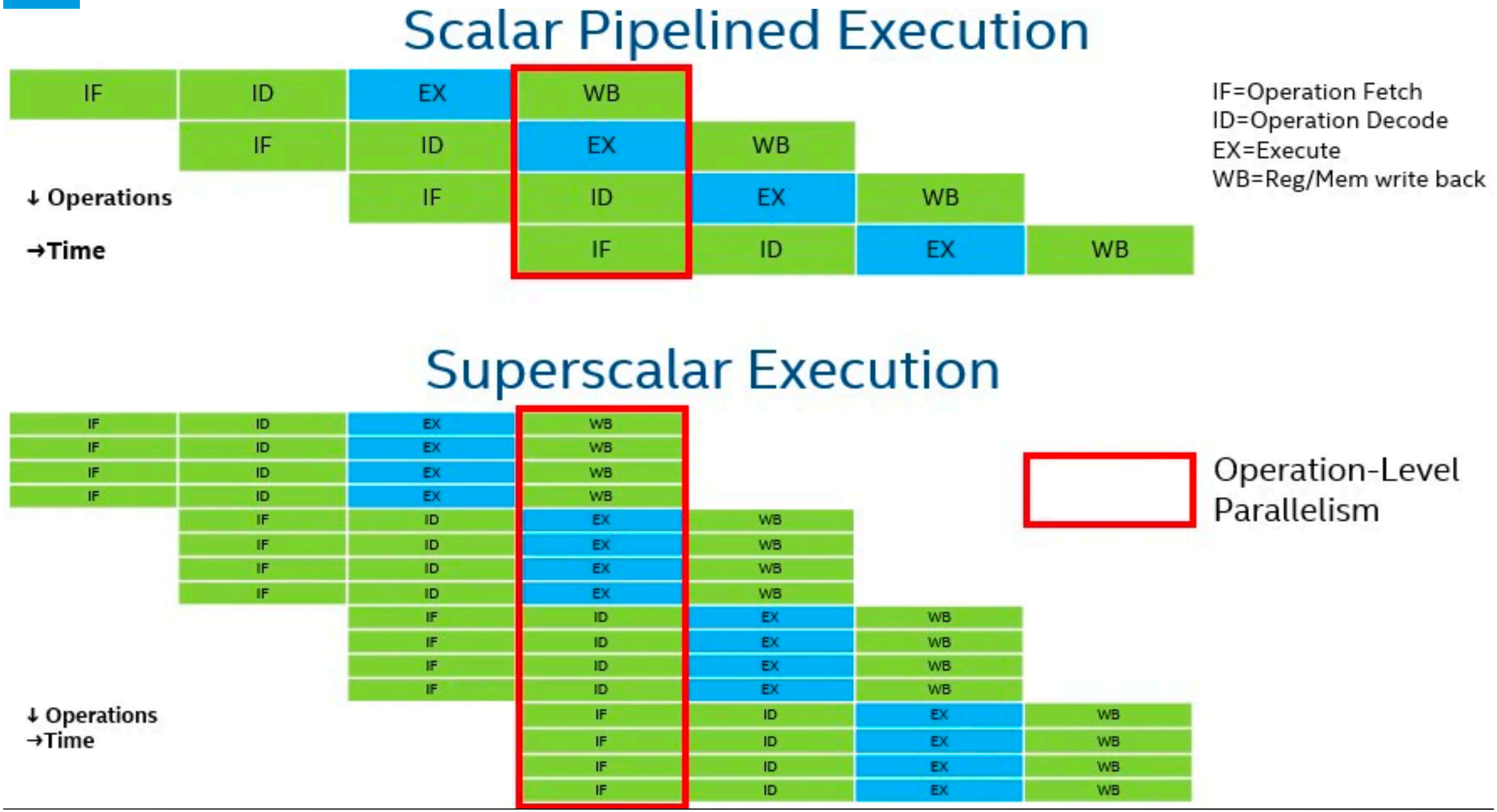
- A superscalar processor executes more than one instruction during a clock cycle by simultaneously dispatching multiple instructions to redundant functional units on the processor.

- Each functional unit is not a separate CPU core but an execution resource within a single CPU such as an arithmetic logic unit (ALU), a bit shifter, or a multiplier.

Superscalar



Superscalar



Scalar Pipelined Execution

IF	ID	EX	WB			
	IF	ID	EX	WB		
		IF	ID	EX	WB	
			IF	ID	EX	WB

↓ Operations

→Time

IF=Operation Fetch
ID=Operation Decode
EX=Execute
WB=Reg/Mem write back

Superscalar Execution

IF	ID	EX	WB			
IF	ID	EX	WB			
IF	ID	EX	WB			
IF	ID	EX	WB			
	IF	ID	EX	WB		
	IF	ID	EX	WB		
	IF	ID	EX	WB		
	IF	ID	EX	WB		
		IF	ID	EX	WB	
		IF	ID	EX	WB	
		IF	ID	EX	WB	
		IF	ID	EX	WB	
			IF	ID	EX	WB
			IF	ID	EX	WB
			IF	ID	EX	WB
			IF	ID	EX	WB

□ Operation-Level Parallelism

↓ Operations
→Time

Is Superscalar Good Enough ?

- A superscalar processor can fetch, decode, execute and retire instructions in parallel
  - Can execute only independent instructions in parallel
- But … adjacent instructions are usually dependent
  - The utilization of the second pipe is usually low
  - There are algorithms in which both pipes are highly utilized
- Solution: out-of-order execution
  - Execute instructions based on “data flow” rather than program order
  - Still need to keep the semantics of the original program

Out of Order Execution

- Make use of cycles that would otherwise be wasted by a certain type of costly delay. Most modern CPU designs include support for out of order execution.

- The key concept of OoO processing is to allow the processor to avoid a class of stalls that occur when the data needed to perform an operation are unavailable.

In Order

1.Instruction fetch.

2.If input operands are available (in registers for instance), the instruction is dispatched to the appropriate functional unit. If one or more operands is unavailable during the current clock cycle (generally because they are being fetched from memory), the processor stalls until they are available.

3.The instruction is executed by the appropriate functional unit.

4.The functional unit writes the results back to the register file.

Out of Order

1.Instruction fetch

2.Instruction dispatch to an instruction queue (also called instruction buffer or reservation stations).

3.The instruction waits in the queue until its input operands are available. The instruction is allowed to leave the queue before earlier, older instructions.

4.The instruction is issued to the appropriate functional unit and executed.

5.The results are queued.

6.Only after all older instructions have their results written back to the register file, then this result is written back to the register file. This is called the graduation or retire stage.

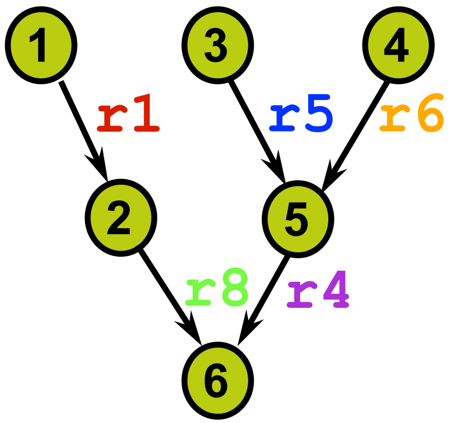
Data Flow Analysis

- Example:

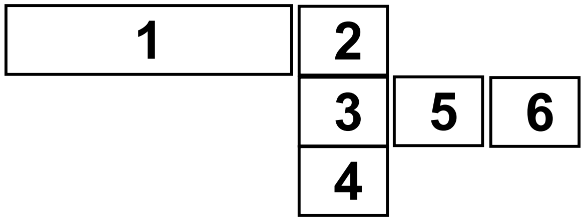
```
(1) r1 ← r4 / r7 ; assume divide takes 20 cycles

(2) r8 ← r1 + r2
(3) r5 ← r5 + 1
(4) r6 ← r6 - r3
(5) r4 ← r5 + r6
(6) r7 ← r8 * r4
```

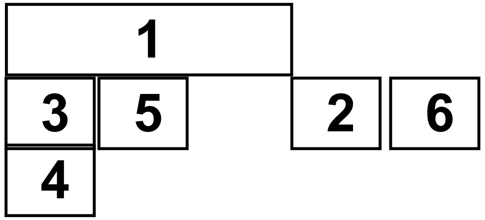
Data Flow Graph



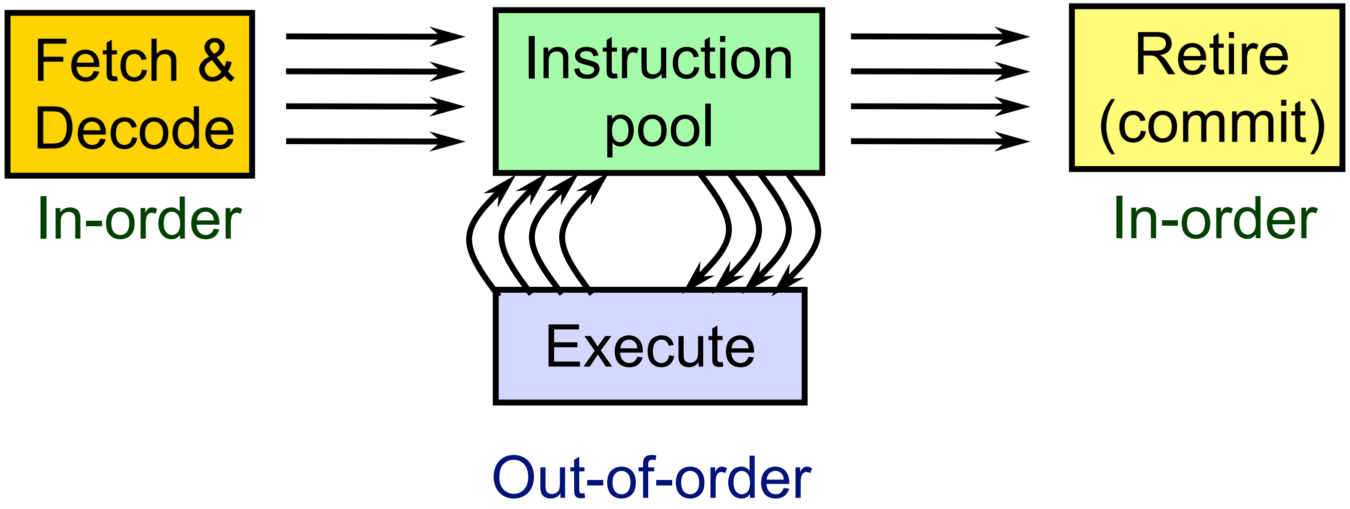
In-order execution



Out-of-order execution



OOOE – General Scheme



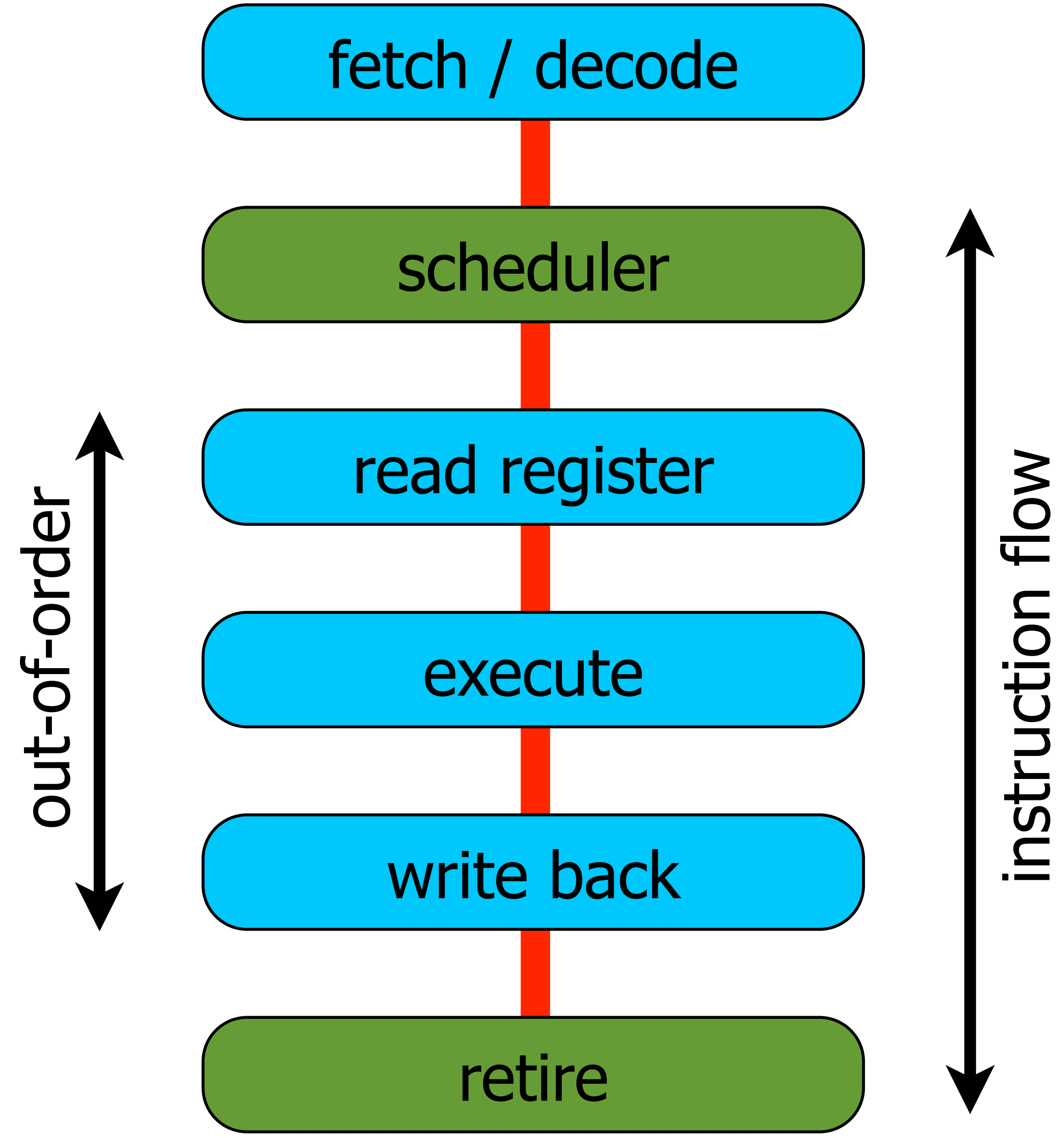
- Fetch & decode instructions in parallel but in order, to fill inst. pool
- Execute ready instructions from the instructions pool
  - All the data required for the instruction is ready
  - Execution resources are available
- Once an instruction is executed
  - signal all dependent instructions that data is ready
- Commit instructions in parallel but in-order
  - Can commit an instruction only after all preceding instructions (in program order) have committed

Out Of Order Execution – Summary

- **Advantages**
  - Help exploit Instruction Level Parallelism (ILP)
  - Help cover latencies (e.g., cache miss, divide)
  - Superior/complementary to compiler scheduler
    - Dynamic instruction window
    - Reg Renaming: can use more than the number architectural registers

- **Complex micro-architecture**
  - Complex scheduler
  - Requires reordering mechanism (retirement) in the back-end for:
    - Precise interrupt resolution
    - Mis-prediction / speculation recovery
    - Memory ordering

Out of order


fetch / decode
scheduler
read register
execute
write back
retire
out-of-order
instruction flow

Pipelining

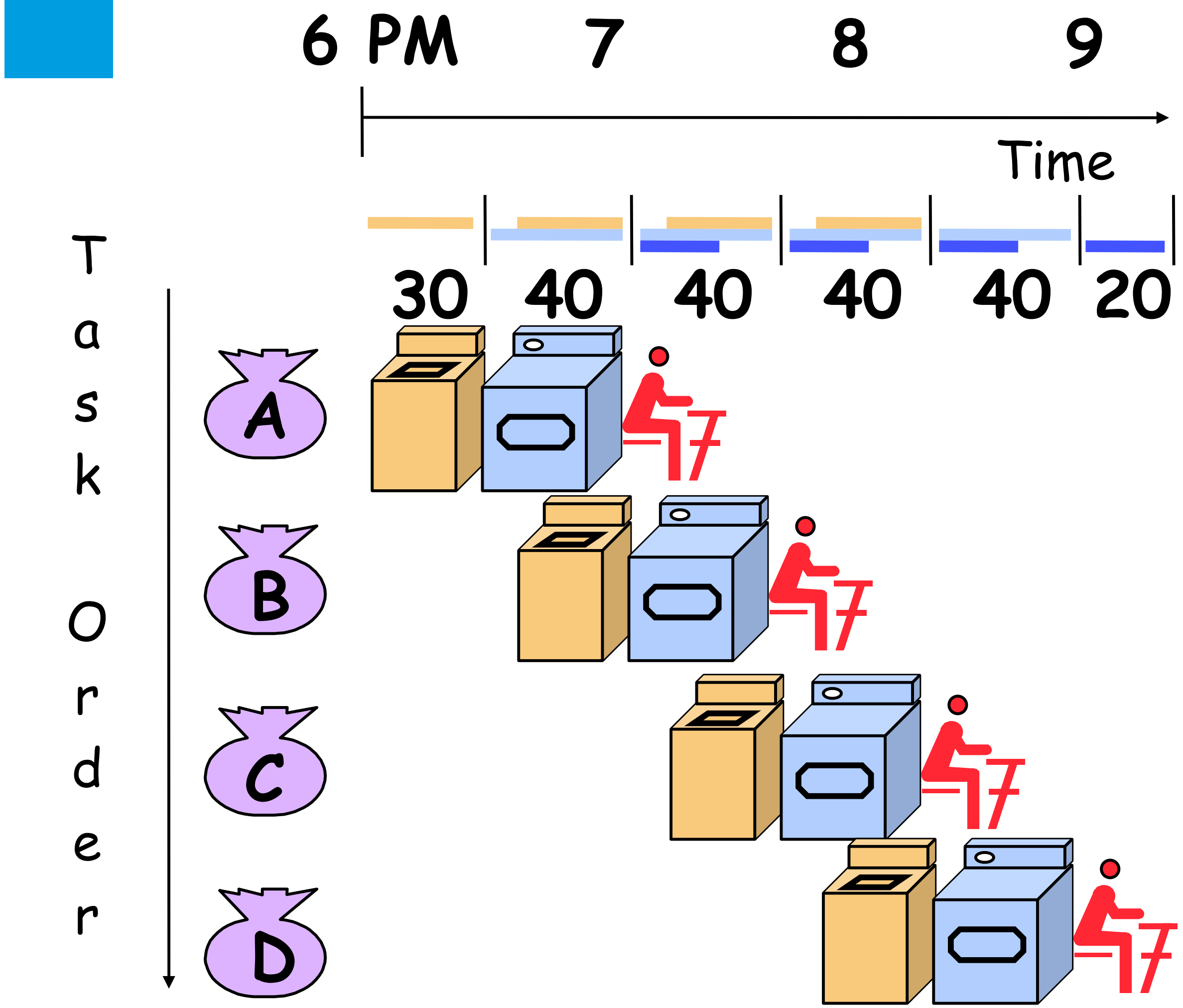
Pipelining

- Pipelining is a technique whereby multiple instructions are overlapped in execution.
- It takes advantage of parallelism that exists among the actions needed to execute an instruction.
- Each step in the pipeline completes a part of the instruction.
- In this way the clock period can be reduced. For example, the RISC pipeline is broken into five stages.

Pipelining

Dave Patterson's Laundry example: 4 people doing laundry

wash (30 min) + dry (40 min) + fold (20 min) = 90 min Latency



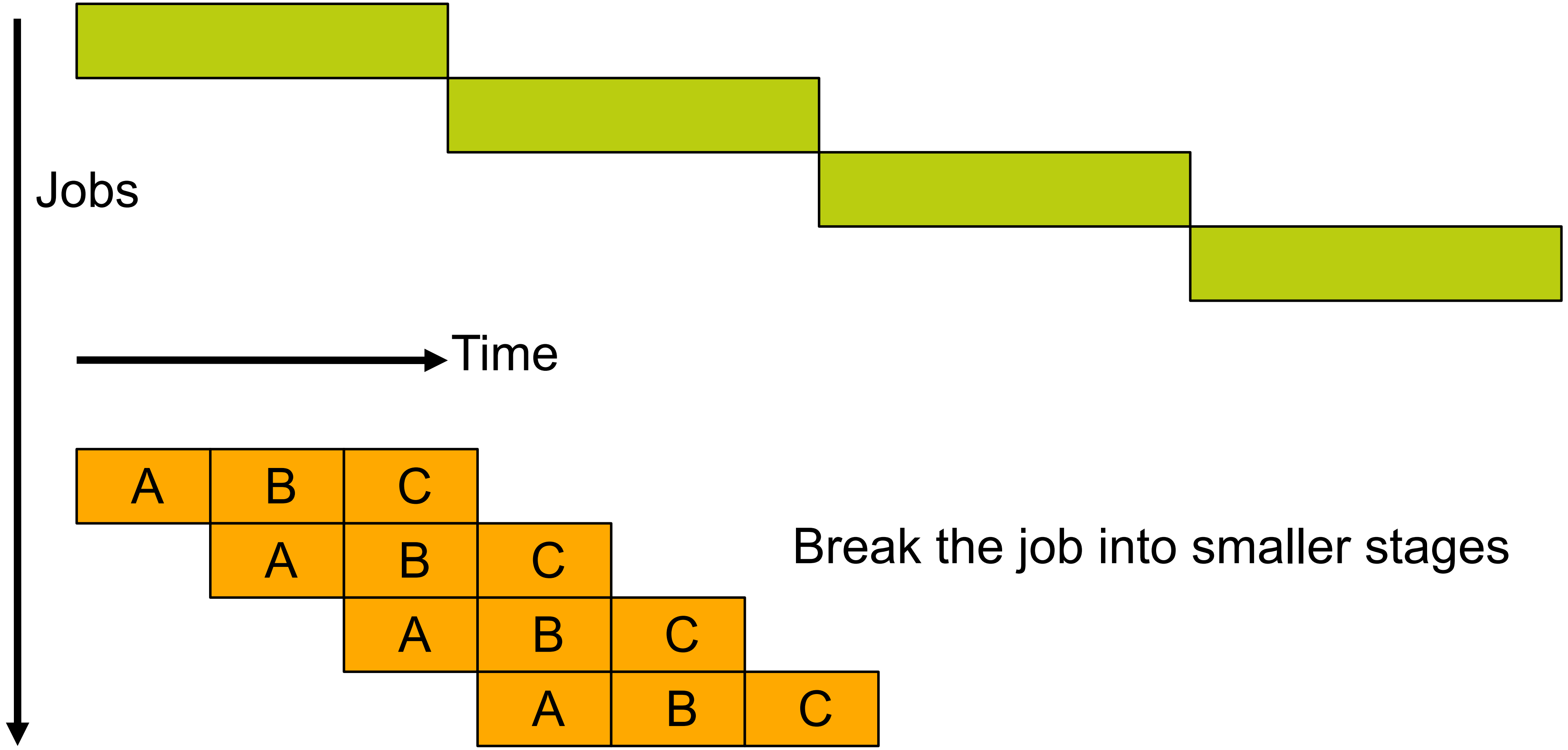
- In this example:
  - Sequential execution takes 4 * 90min = 6 hours
  - Pipelined execution takes 30+4*40+20 = 3.5 hours
- Pipelining helps bandwidth but not latency (90 min)

slide from Berkeley course cs194

The Assembly Line

Unpipelined

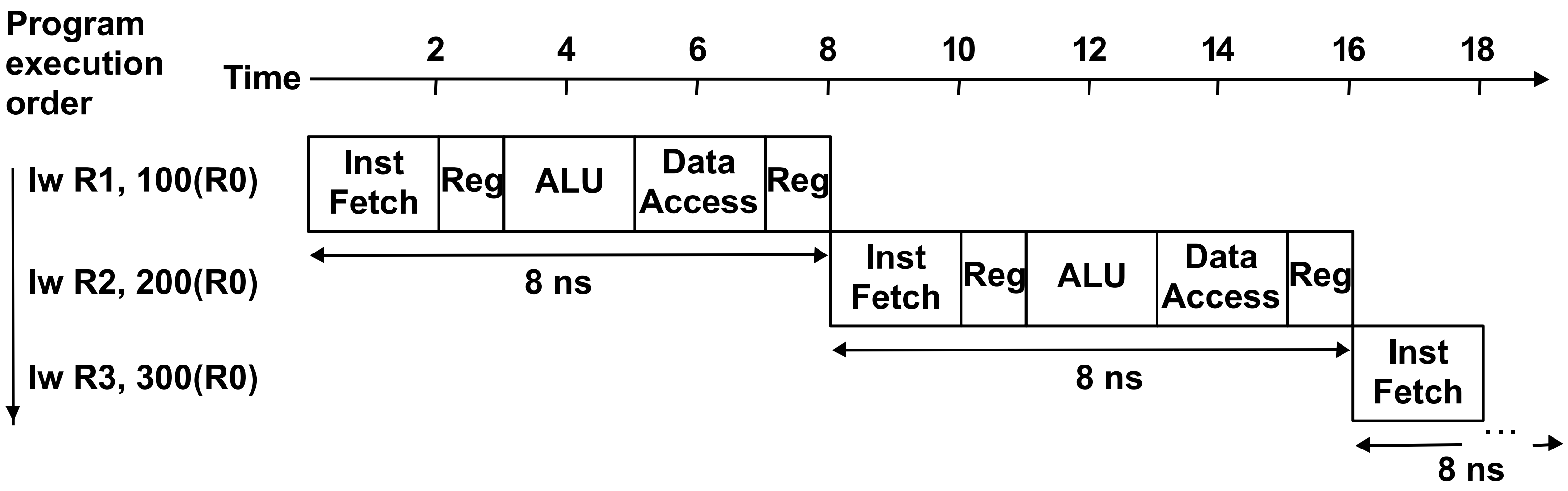
Start and finish a job before moving to the next

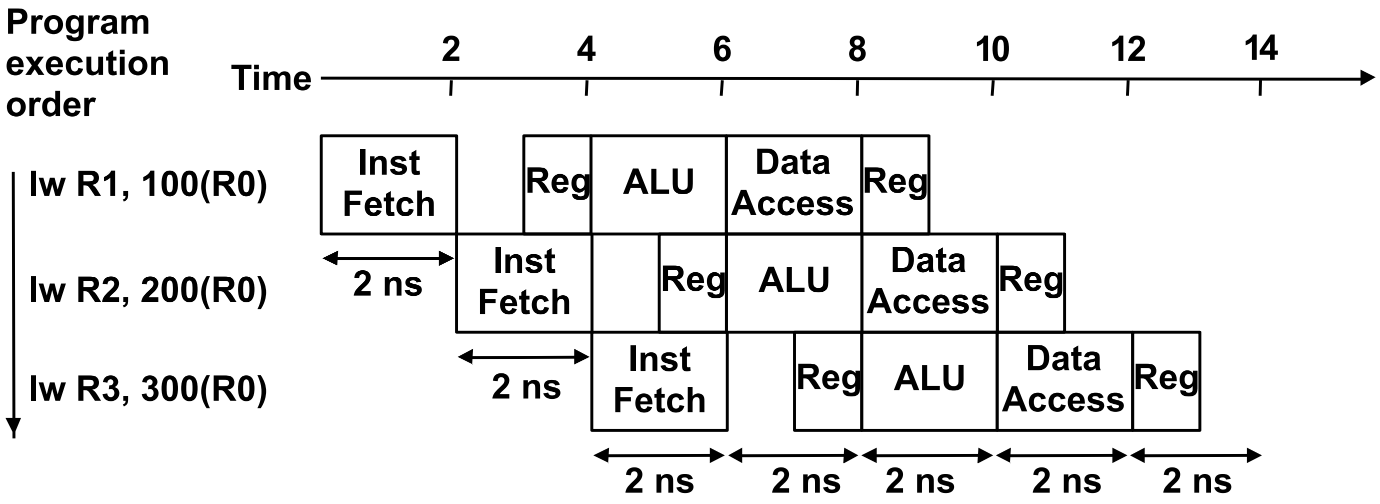


Break the job into smaller stages

Pipelined

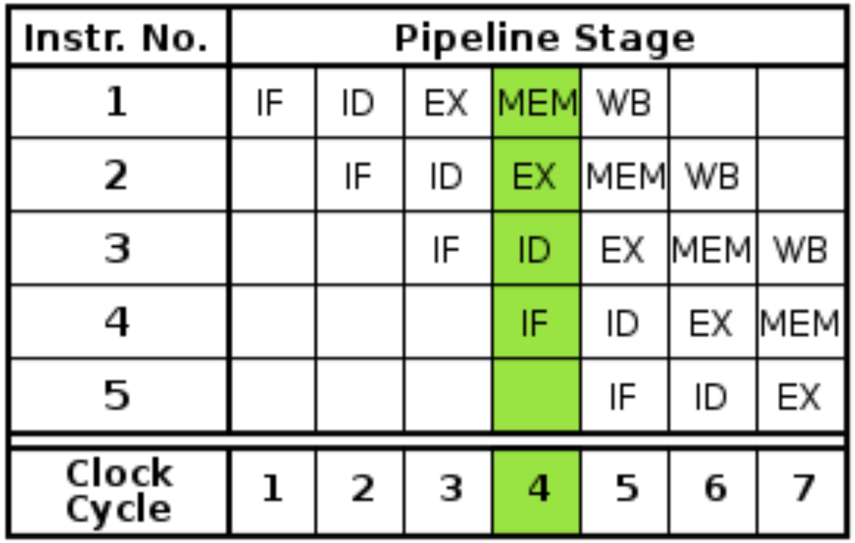
Pipelining Instructions





Ideal speedup is number of stages in the pipeline.

Pipeline example

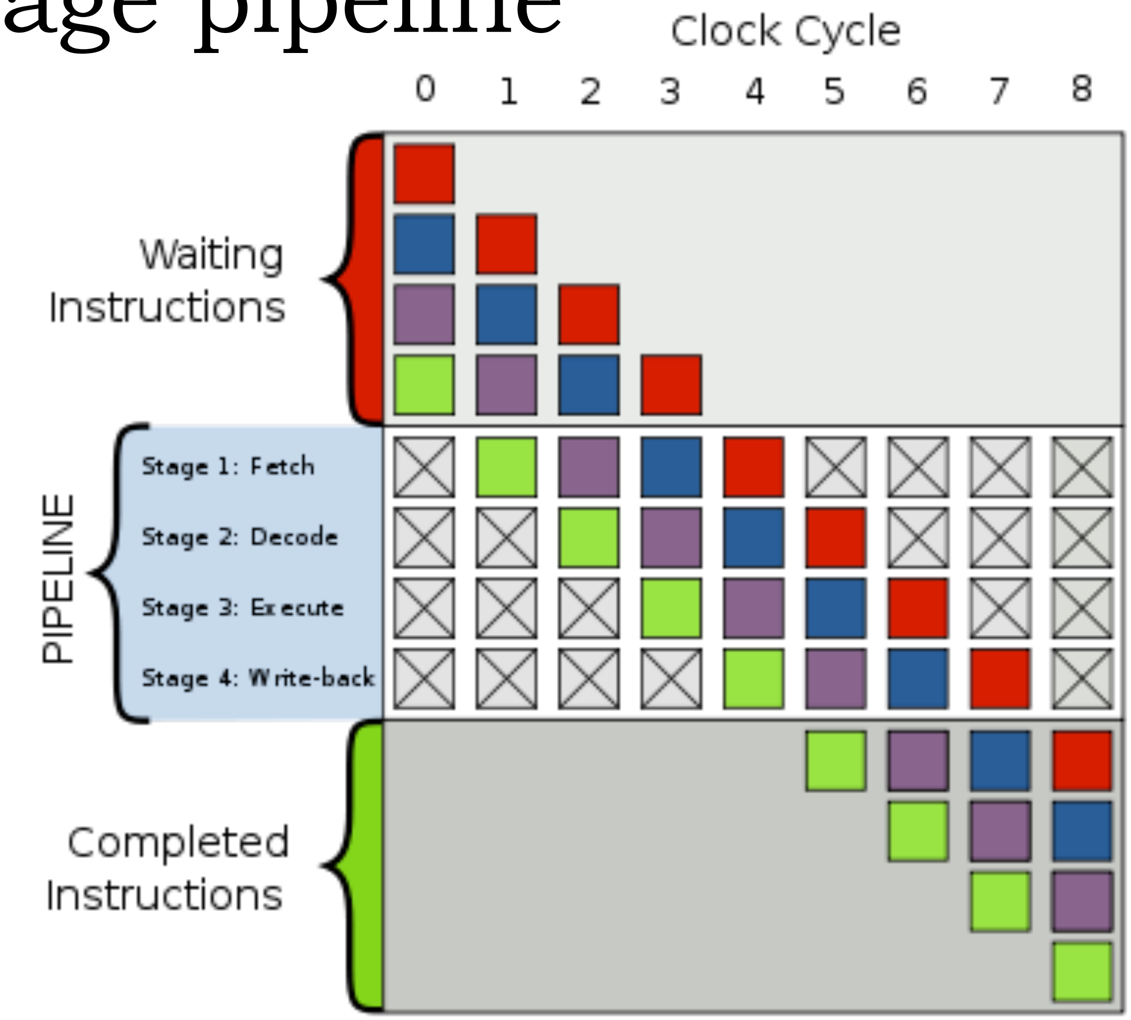


Instr. No.	Pipeline Stage						
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX
Clock Cycle	1	2	3	4	5	6	7

Basic five-stage pipeline in a RISC machine:

- IF = Instruction Fetch,
- ID = Instruction Decode,
- EX = Execute, compute address or operation (add)
- MEM = Memory access, read or store address
- WB = Register write back.

4-stage pipeline



from: http://en.wikipedia.org/wiki/Instruction_pipeline

Pipelining

- Pipelining does not reduce the latency of single task, it increases the throughput of entire workload
- Potential speedup = Number of pipe stages
  - Pipeline rate is limited by the slowest pipeline stage
    - ⇒ Partition the pipe to many pipe stages
    - ⇒ Make the longest pipe stage to be as short as possible
    - ⇒ Balance the work in the pipe stages
- Pipeline adds overhead (e.g., latches)
  - Time to “fill” pipeline and time to “drain” it reduces speedup
  - Stall for dependencies
    - ⇒ Too many pipe-stages start to loose performance
- IPC (Instructions Per Clock) of an ideal pipelined machine is 1
  - Every clock one instruction finishes

Pipeline hazards

- Pipelining introduces an extra layer of complexity than can lead to other problems (called pipeline hazards). Some of the hazards can be solved by adding additional hardware.

- This is for specialist in hardware design and we will not discuss this in much detail.

Pipeline Hazards

- **Structural Hazards**: Hardware doesn’t support two instructions in the same cycle

  - **Data Hazards**: Instructions can’t be executed since the source data is not available since still computed by a preceding instruction

  - **Load-Use Hazards**: source data is not available since data memory load instruction has not yet completed

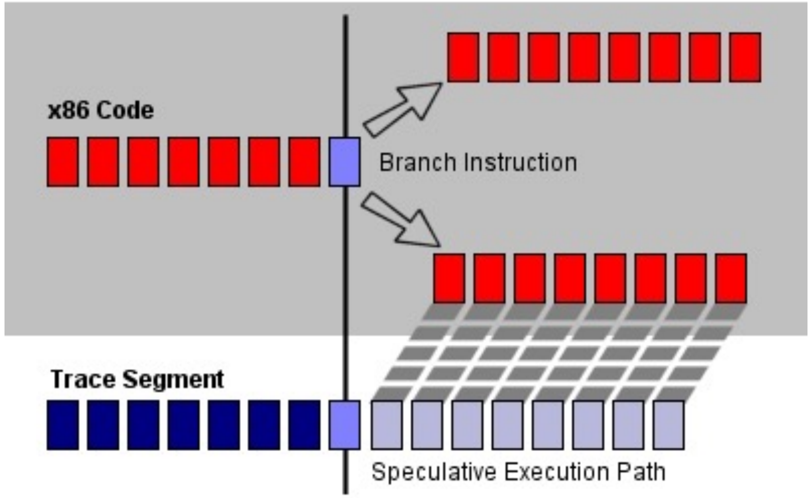
  - **Branch Hazards**: due to a branch (condition) in the code, the pipeline must wait until the next instruction is determined.

Branches

Branches examples

- Instructions which can alter the flow of instruction execution in a program
  - `if (a[i]> 1.0) {}`
  - `do_work(a, n1, n2, parm);`
  - `for (i=0; i<n, i++) {}`
  - `while (eps >= 1e-3) {}`
  - `return;`

Techniques for handling branches



- Stalling
  - waiting for condition to be computed and then continue with correct branch
- Predication
  - All possible branch paths are executed, the correct path is kept and all others are thrown away.
- Prediction
  - try to predict the next if statement based on previous if statement(s).

Multi-threading

Thread-Level Parallelism (TLP)

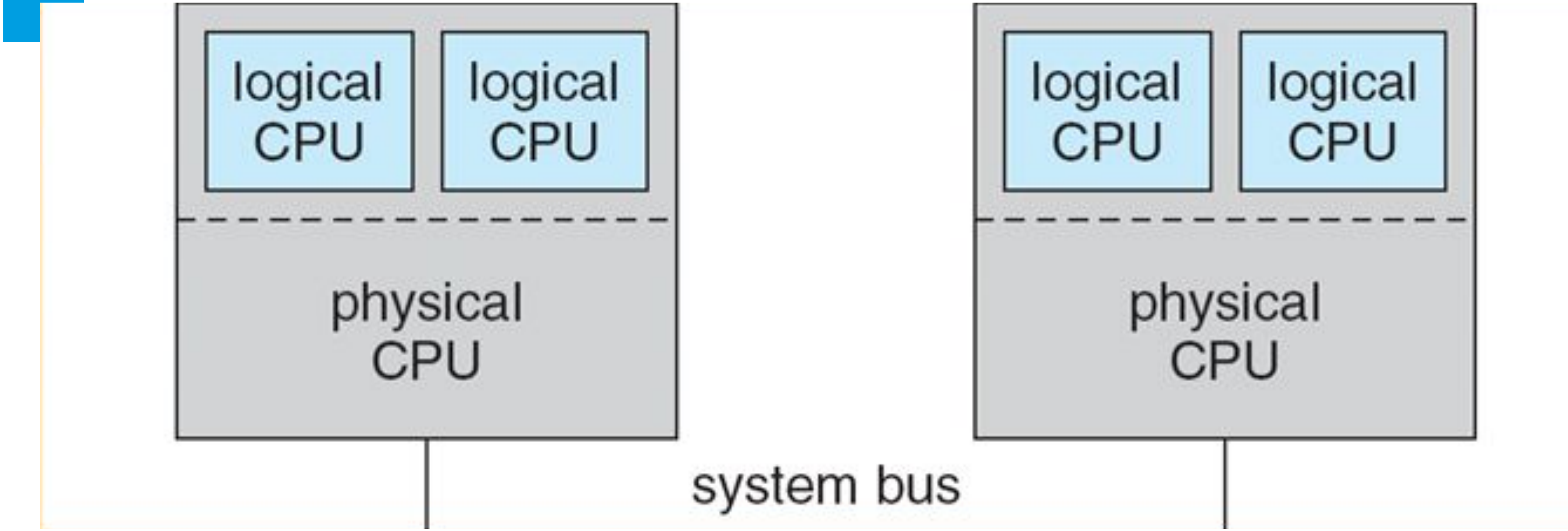
- Motivation: a single thread leaves a processor under-utilized for most of the time

- Strategies for thread-level parallelism:

- multiple threads share the same large processor =>
  - reduces under-utilization, efficient resource allocation
  - Simultaneous Multi-Threading (SMT)

- each thread executes on its own mini processor =>
  - simple design, low interference between threads
  - Chip Multi-Processing (CMP)

Simultaneous Multi-Threading



Hyper-threading (HT) Technology

- HT is SMT
  - Makes a single processor appear as 2 *logical processors* = *threads*

- Each thread keeps a its own architectural state
  - General-purpose registers
  - Control and machine state registers

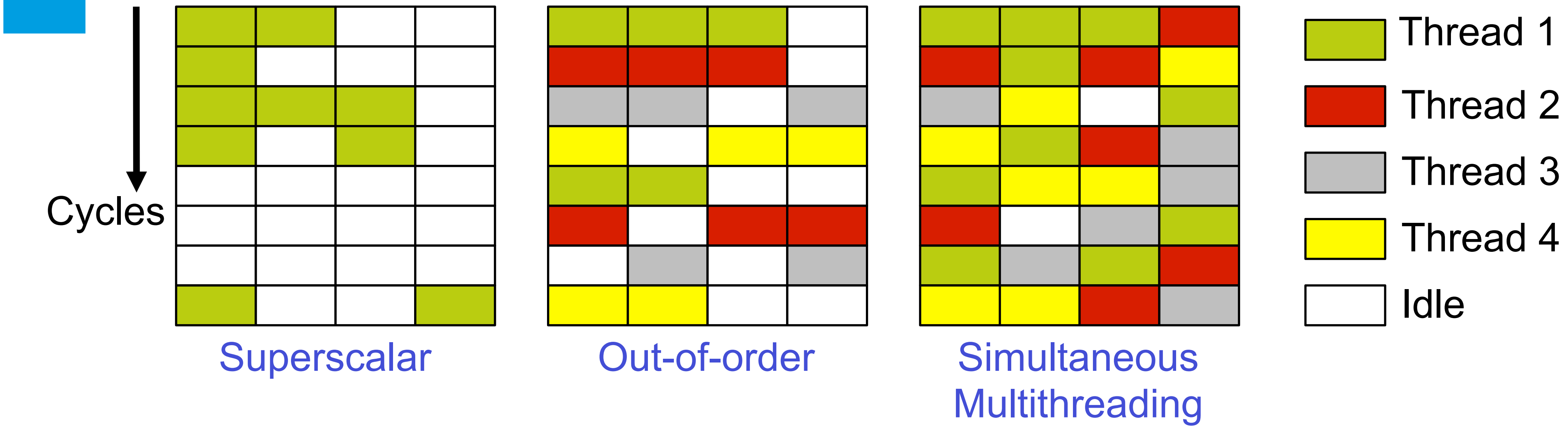
- Each thread has its own interrupt controller
  - Interrupts sent to a specific logical processor are handled only by it

- OS views logical processors (threads) as physical processors
  - Schedule threads to logical processors as in a multiprocessor system

- From a micro-architecture perspective
  - Thread share a single set of physical resources
    - caches, execution units, branch predictors, control logic, and buses

How are Resources Shared?

Each box represents an issue slot for a functional unit. Peak throughput is 4 IPC.



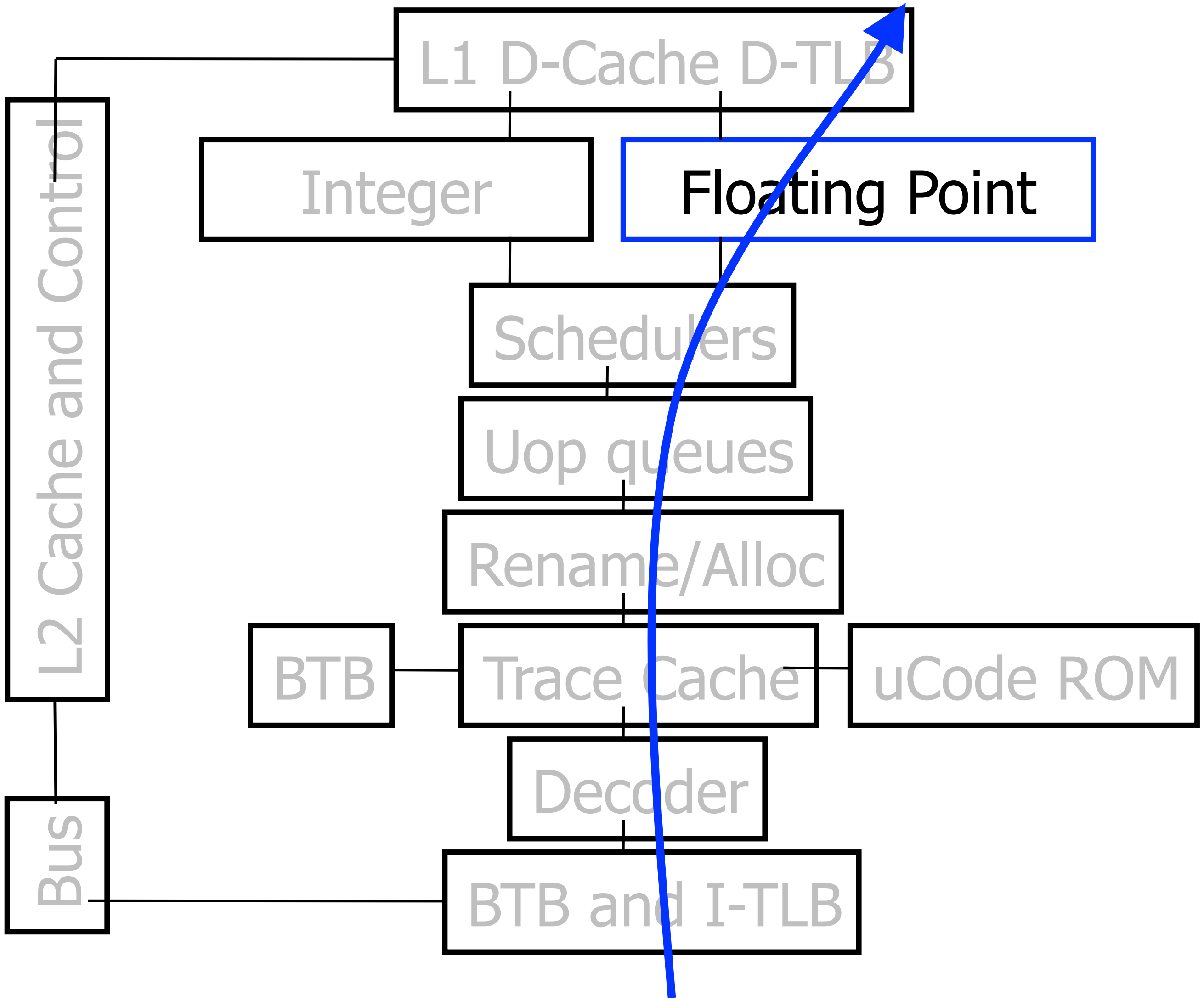
🟩	🟩	□	□
🟩	□	□	□
🟩	🟩	🟩	□
🟩	□	🟩	□
□	□	□	□
□	□	□	□
□	□	□	□
🟩	□	□	🟩

🟩	🟩	🟩	□
🟥	🟥	🟥	□
▨	▨	□	▨
🟨	□	🟨	🟨
🟩	🟩	□	□
🟥	□	🟥	🟥
□	▨	□	▨
🟨	🟨	□	□

🟩	🟩	🟩	🟥
🟥	🟩	🟥	🟨
▨	🟨	□	🟩
🟨	🟩	🟥	▨
🟩	🟨	🟨	▨
🟥	□	▨	🟩
🟩	▨	🟩	🟥
🟨	🟨	🟥	▨

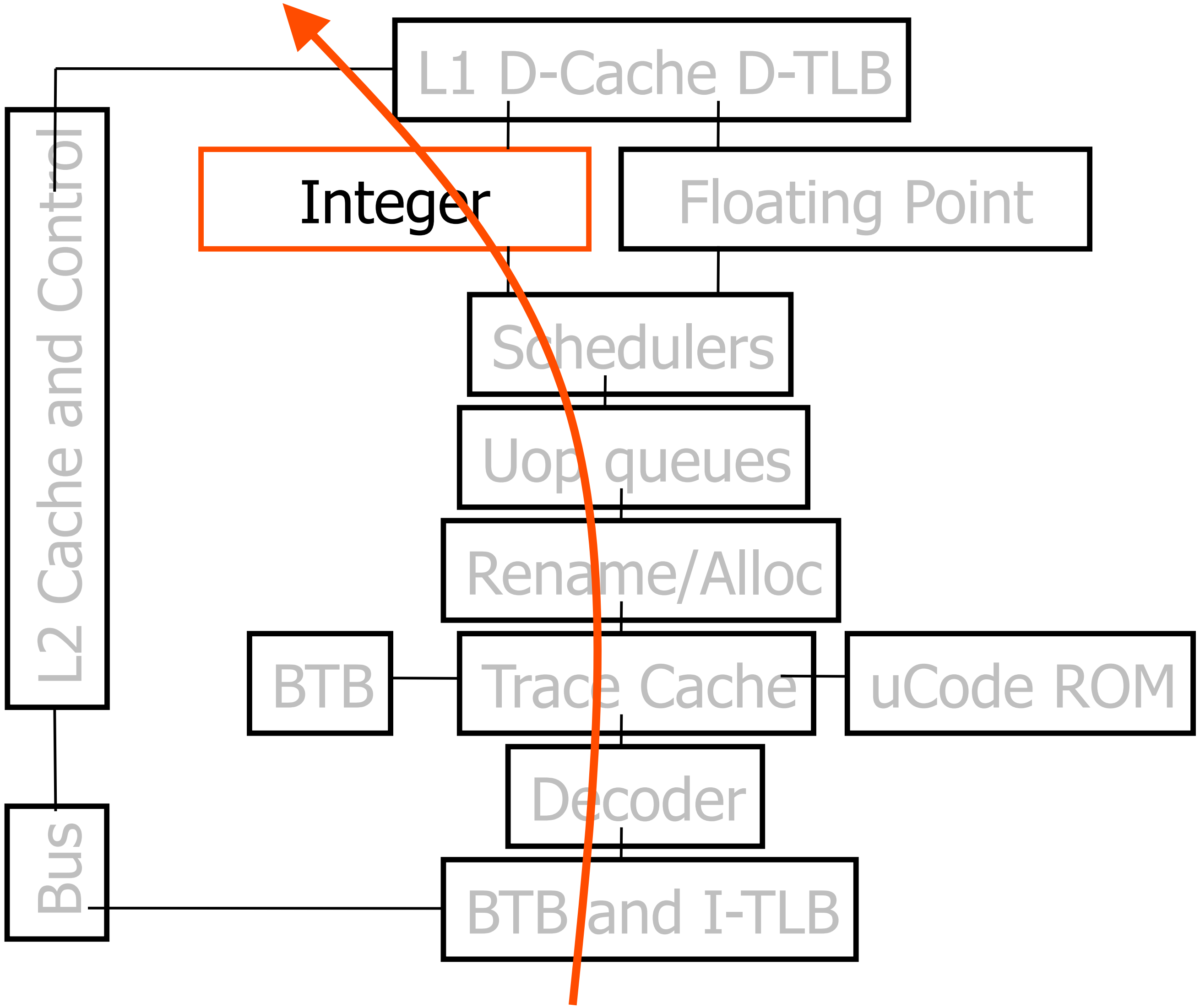
- Superscalar processor has high under-utilization – not enough work every cycle, especially when there is a cache miss
- Out-of-order can only issue instructions from a single thread in a cycle – can not find max work every cycle, but cache misses can be tolerated
- Simultaneous multithreading can issue instructions from any thread every cycle – has the highest probability of finding work for every issue slot

Without SMT, only a single thread can run at any given time



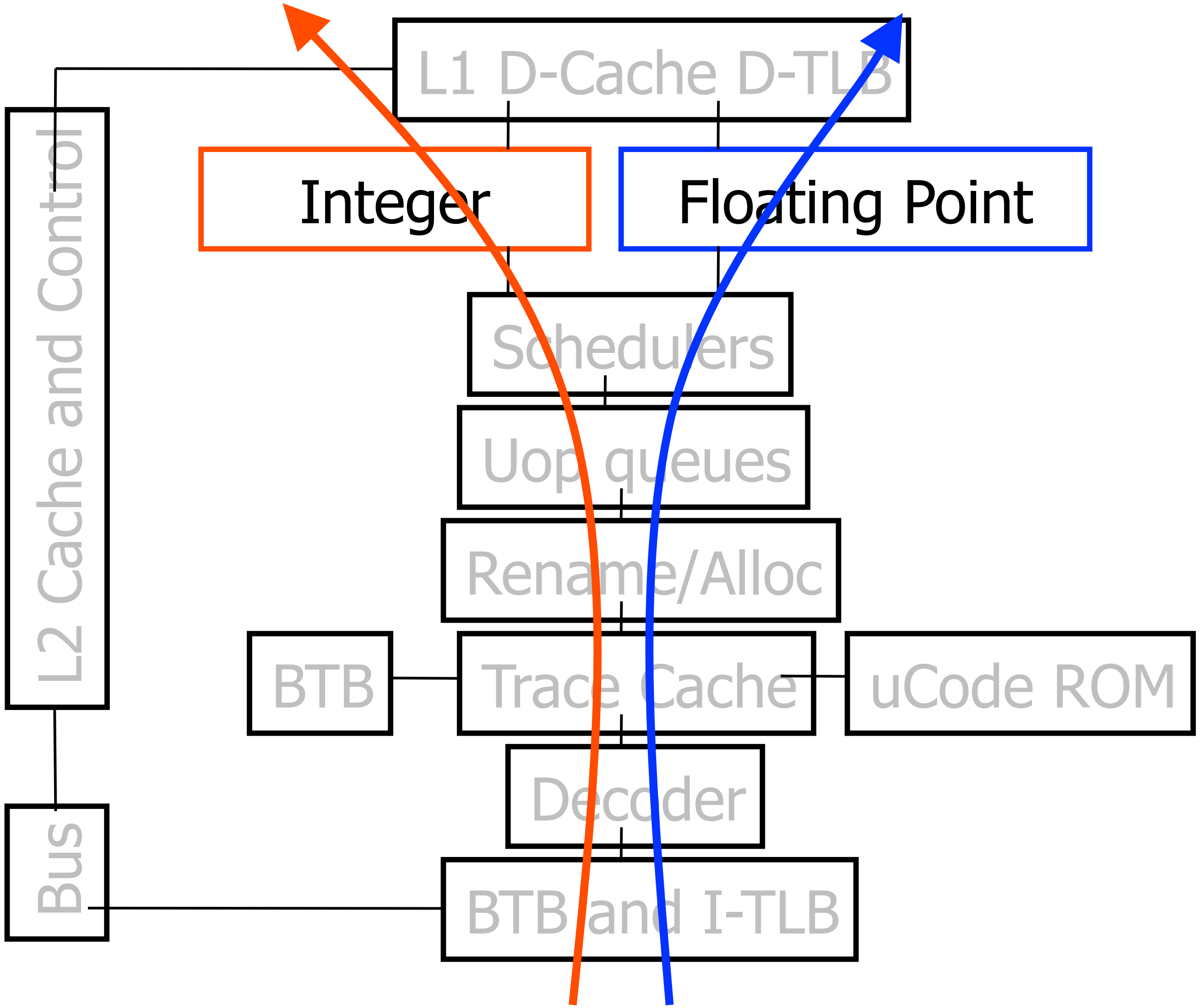
Thread 1: floating point

Without SMT, only a single thread can run at any given time



Thread 2:
integer operation

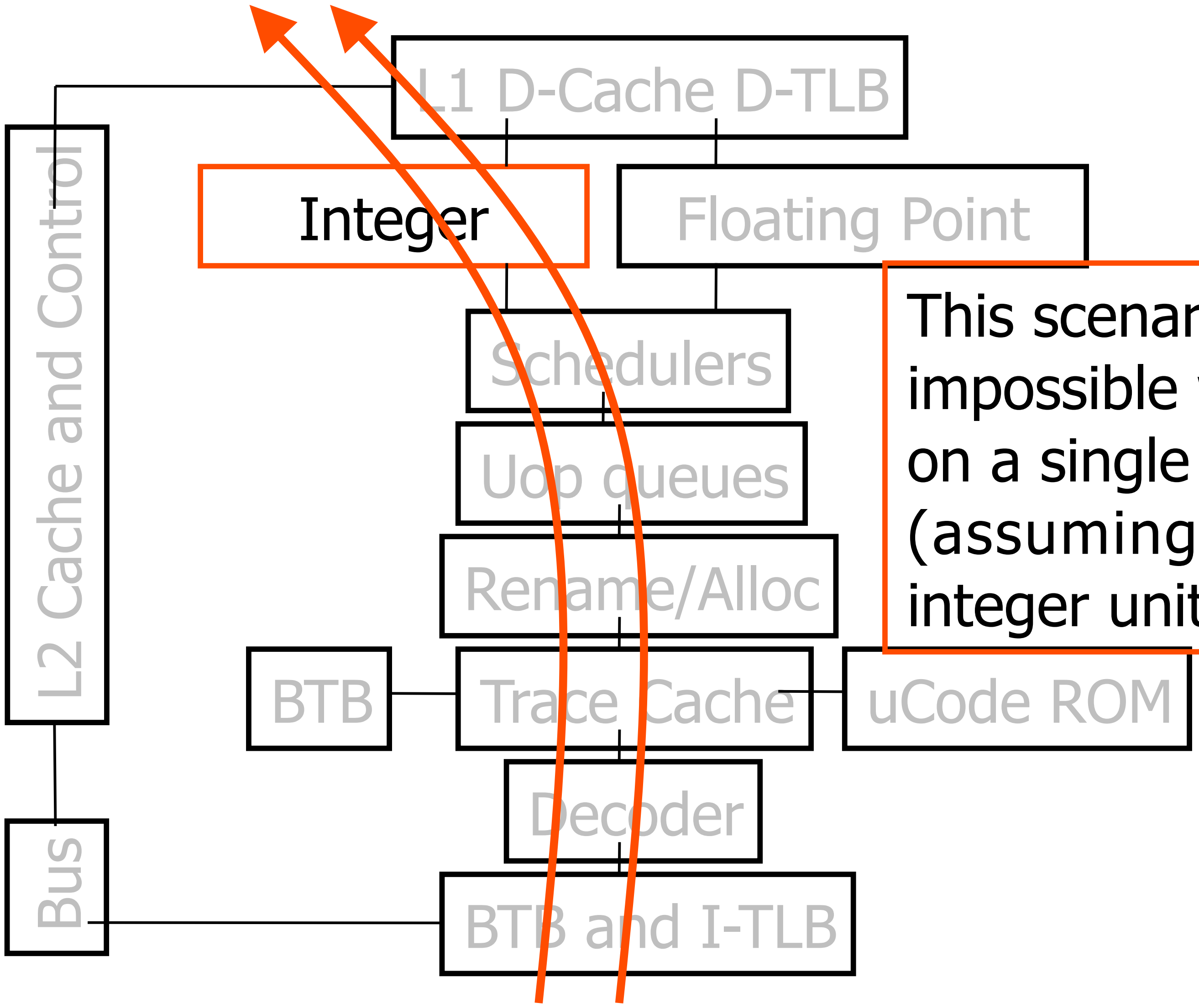
SMT processor: both threads can run concurrently



Thread 2: integer operation

Thread 1: floating point

But: Can't simultaneously use the same functional unit



This scenario is impossible with SMT on a single core (assuming a single integer unit)

Thread 1 Thread 2

IMPOSSIBLE

Multi-Threading story will be continued

Prefetching

- Instruction Prefetching
  - On a cache miss, prefetch sequential cache lines into stream buffers
  - Branch predictor directed prefetching
    - Let branch predictor run ahead
- Data Prefetching - predict future data accesses
  - Next sequential
  - Stride
  - General pattern
- Software Prefetching
  - Special prefetching instructions
- Prefetching relies on extra memory bandwidth
  - Otherwise it slows down demand fetches

➡difficult to do prefectching correct as a programmer

Putting things together

- Loop unrolling

Example: Loop Scheduling

- Using 5-stage in-order pipeline
- The compiler's job is to minimize stalls
- Assume:
  - load has a two-cycle latency (1 stall cycle for the consumer that immediately follows),
  - FP ALU feeding another => 3 stall cycles,
  - FP ALU feeding a store => 2 stall cycles,
  - int ALU feeding a branch => 1 stall cycle,
  - one delay slot after a branch.

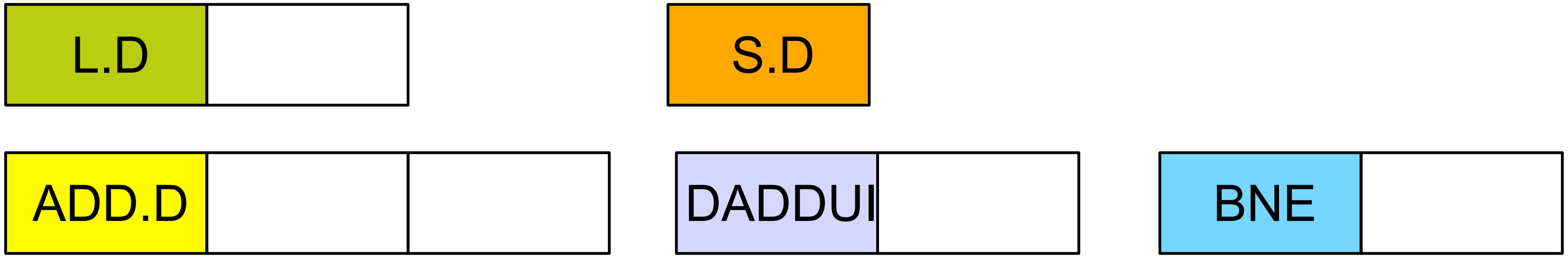
Loop Example

```
for (i=1000; i>0; i--)
   x[i] = x[i] + s;
```

Source code

```
Loop:    L.D       F0, 0(R1)        ; F0 = array element
         ADD.D     F4, F0, F2       ; add scalar
         S.D       F4, 0(R1)        ; store result
         DADDUI    R1, R1,# -8      ; decrement address pointer
         BNE       R1, R2, Loop     ; branch if R1 != R2
```

Assembly code



Loop Example

```
for (i=1000; i>0; i--)
    x[i] = x[i] + s;
```

Source code

```
Loop:   L.D      F0, 0(R1)       ; F0 = array element
        stall
        ADD.D    F4, F0, F2      ; add scalar
        stall
        stall
        S.D      F4, 0(R1)       ; store result
        DADDUI   R1, R1,# -8     ; decrement address pointer
        stall
        BNE      R1, R2, Loop    ; branch if R1 != R2
        stall
```

10-cycle schedule

L.D		ADD.D			S.D	DADDUI		BNE	

Smart Schedule

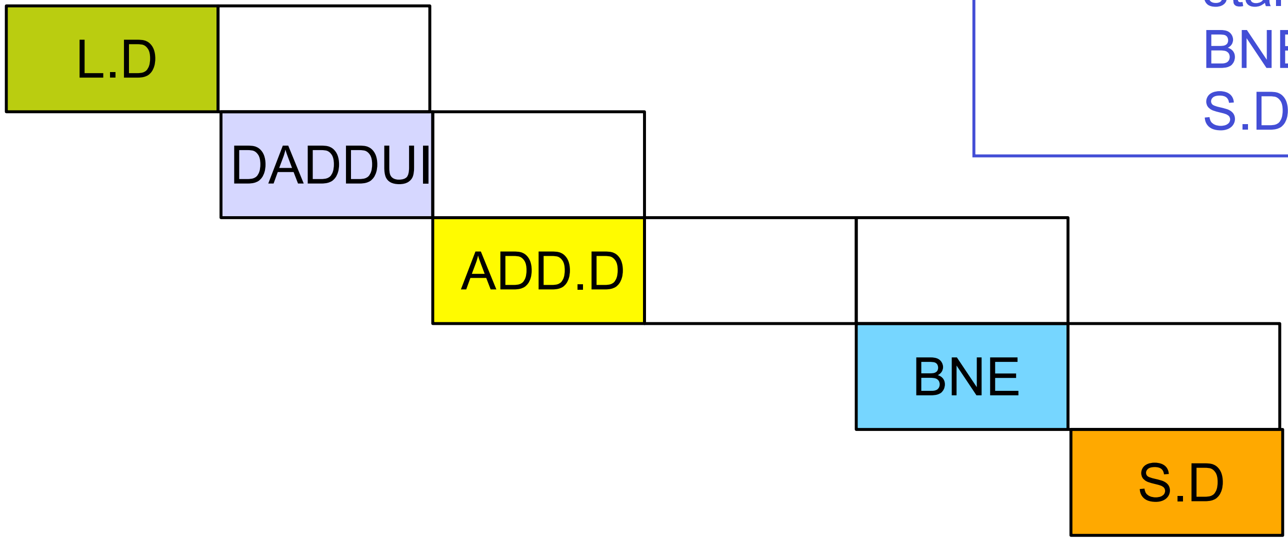
```
Loop:   L.D      F0, 0(R1)
        stall
        ADD.D    F4, F0, F2
        stall
        stall
        S.D      F4, 0(R1)
        DADDUI   R1, R1,# -8
        stall
        BNE      R1, R2, Loop
        stall
```

```
Loop:   L.D      F0, 0(R1)
        DADDUI   R1, R1,# -8
        ADD.D    F4, F0, F2
        stall
        BNE      R1, R2, Loop
        S.D      F4, 8(R1)
```

- By re-ordering instructions, it takes 6 cycles per iteration instead of 10

Smart Schedule

```
Loop:   L.D      F0, 0(R1)
        DADDUI   R1, R1,# -8
        ADD.D    F4, F0, F2
        stall
        BNE      R1, R2, Loop
        S.D      F4, 8(R1)
```



Loop overhead (BNE, DADDUI): 2 instructions
Actual work (the LD, ADD.D, and S.D): 3 instructions

Can we somehow get execution time to be 3 cycles per iteration?

Loop Unrolling: solution on user level

- Reduces the number of ‘end of loop’ checks.
- Increases program size
- Requires more registers
- To unroll an n-iteration loop by degree k, we will need (n/k) iterations of the larger loop, followed by (n mod k) iterations of the original loop.

Loop Unrolling (4 times)

```
Loop:   L.D      F0, 0(R1)
        ADD.D    F4, F0, F2
        S.D      F4, 0(R1)
        L.D      F6, -8(R1)
        ADD.D    F8, F6, F2
        S.D      F8, -8(R1)
        L.D      F10,-16(R1)
        ADD.D    F12, F10, F2
        S.D      F12, -16(R1)
        L.D      F14, -24(R1)
        ADD.D    F16, F14, F2
        S.D      F16, -24(R1)
        DADDUI   R1, R1, #-32
        BNE      R1,R2, Loop
```

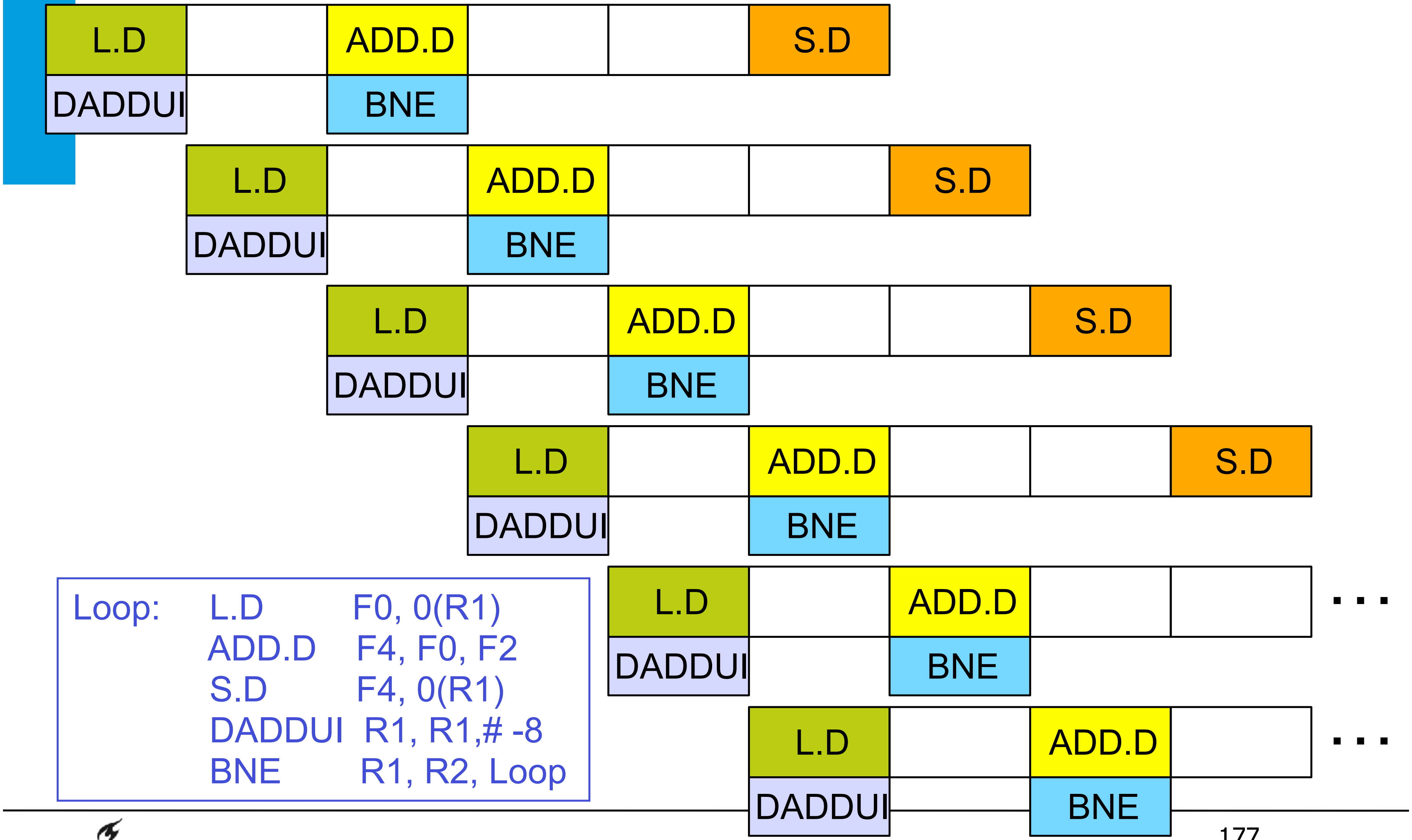
- Loop overhead: 2 instrs; Work: 12 instrs
- How long will the above schedule take to complete?

Scheduled and Unrolled Loop

```
Loop:   L.D      F0, 0(R1)
        L.D      F6, -8(R1)
        L.D      F10,-16(R1)
        L.D       F14, -24(R1)
        ADD.D    F4, F0, F2
        ADD.D    F8, F6, F2
        ADD.D    F12, F10, F2
        ADD.D    F16, F14, F2
        S.D      F4, 0(R1)
        S.D      F8, -8(R1)
        DADDUI   R1, R1, # -32
        S.D      F12, 16(R1)
        BNE       R1,R2, Loop
        S.D      F16, 8(R1)
```

- Execution time: 14 cycles or 3.5 cycles per original iteration

Pipeline has similar efficiency as unrollin



L.D ADD.D S.D

DADDUI BNE

L.D ADD.D S.D

DADDUI BNE

L.D ADD.D S.D

DADDUI BNE

L.D ADD.D S.D

DADDUI BNE

L.D ADD.D …

DADDUI BNE

L.D ADD.D …

DADDUI BNE

```
Loop:   L.D       F0, 0(R1)
        ADD.D     F4, F0, F2
        S.D       F4, 0(R1)
        DADDUI  R1, R1,# -8
        BNE       R1, R2, Loop
```

Software Pipelining

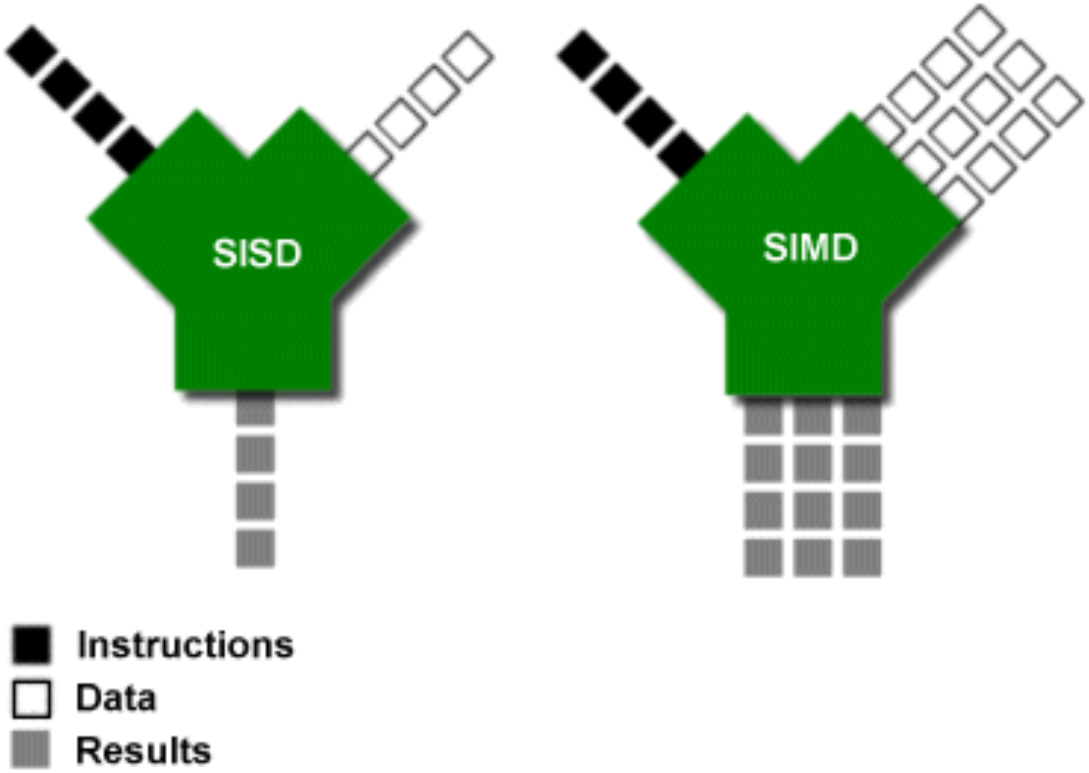
```
Loop:   L.D       F0, 0(R1)
        ADD.D     F4, F0, F2
        S.D       F4, 0(R1)
        DADDUI    R1, R1,# -8
        BNE       R1, R2, Loop
```

```
Loop:   S.D       F4, 16(R1)
        ADD.D     F4, F0, F2
        L.D       F0, 0(R1)
        DADDUI    R1, R1,# -8
        BNE       R1, R2, Loop
```

- Advantages: achieves nearly the same effect as loop unrolling, but without the code expansion – an unrolled loop may have inefficiencies at the start and end of each iteration, while a sw-pipelined loop is almost always in steady state – a sw-pipelined loop can also be unrolled to reduce loop overhead

- Disadvantages: does not reduce loop overhead, may require more registers

Vector instructions SSE2 SSE3 SSE4 AVX ...



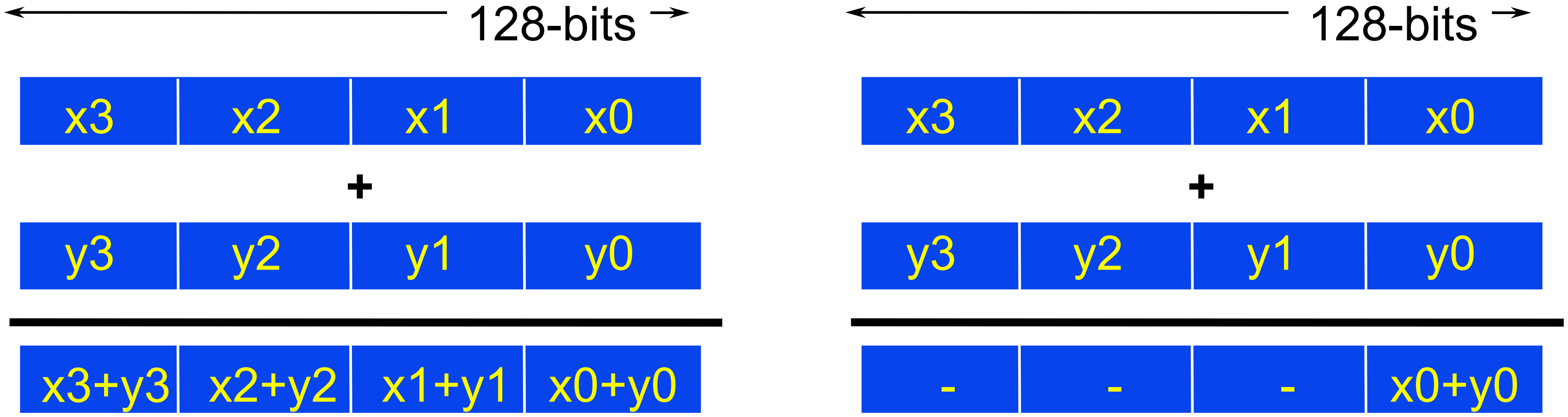
Software Specific Extensions

- Extend arch to accelerate exec of specific apps

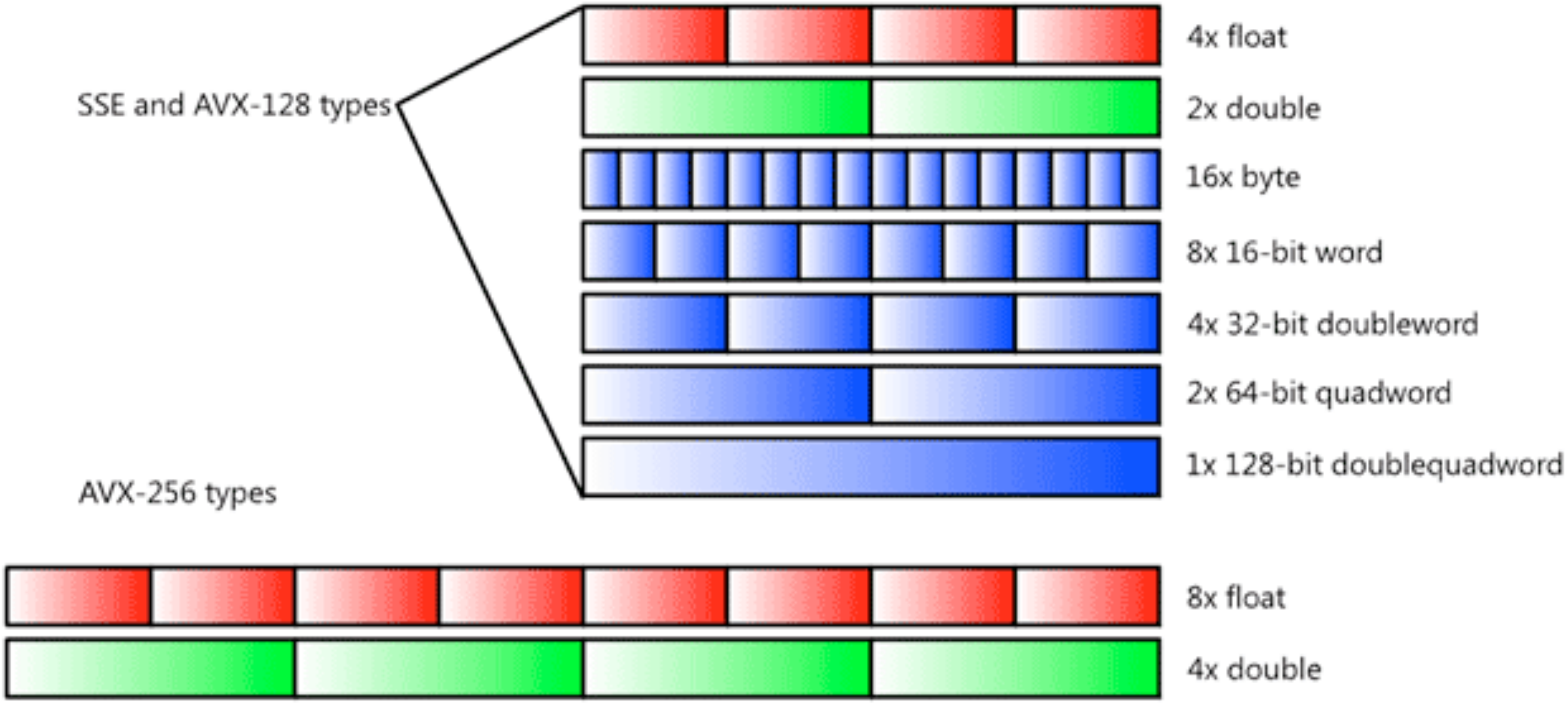
- Example: SSE™ – Streaming SIMD Extensions
  - 128-bit packed (vector) / scalar single precision FP (4×32)
  - Introduced on Pentium® III on ’99
  - 8 new 128 bit registers (XMM0 – XMM7)
  - Accelerates graphics, video, scientific calculations, …

- Packed vectorized:

Scalar:



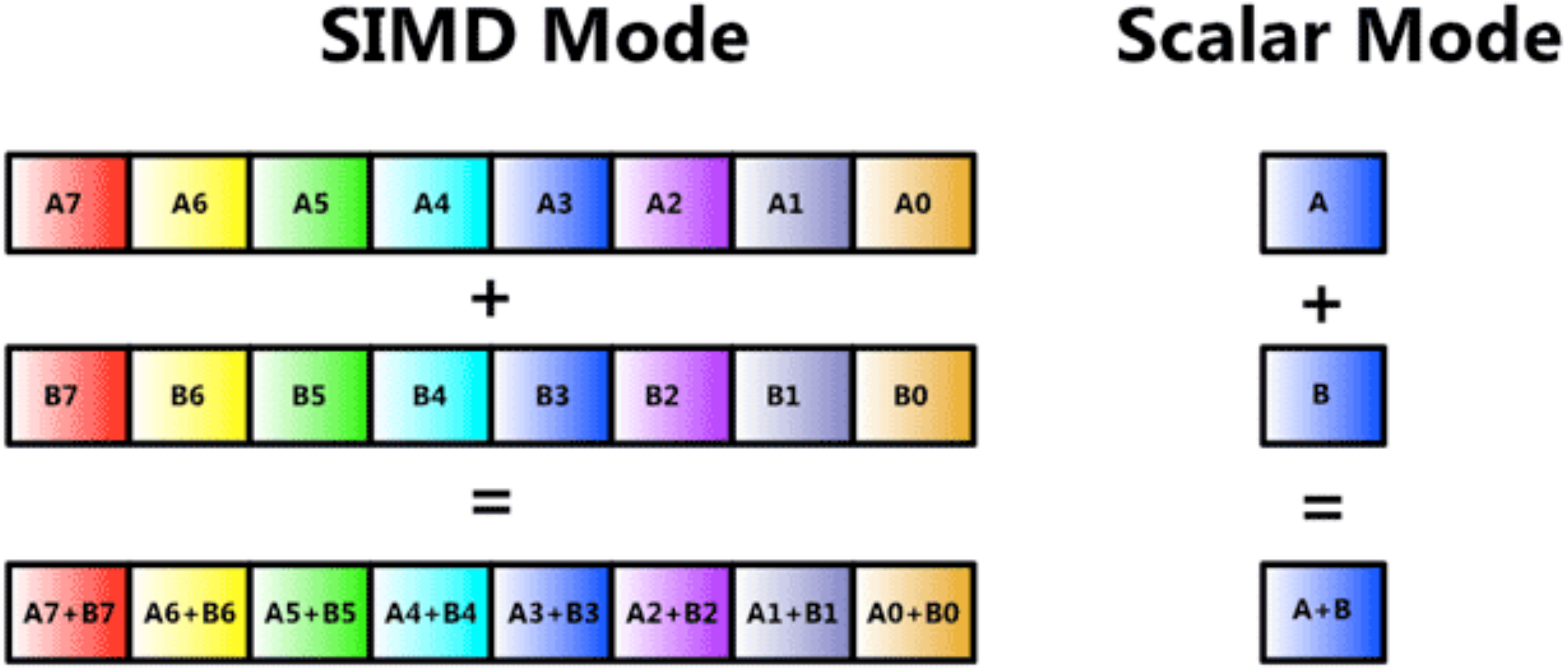
successor AVX



History of vector instructions

		width	Int.	SP	DP
1997	MMX	64	✔		
1999	SSE	128	✔	✔(x4)	
2001	SSE2	128	✔	✔	✔(x2)
2004	SSE3	128	✔	✔	✔
2006	SSSE 3	128	✔	✔	✔
2006	SSE 4.1	128	✔	✔	✔
2008	SSE 4.2	128	✔	✔	✔
2011	AVX	256	✔	✔(x8)	✔(x4)
2013	AVX2	256	✔	✔	✔
2019	AVX-512	512	✔	✔(x16)	✔(x8)

AVX2 2014


SIMD Mode
Scalar Mode
A7
A6
A5
A4
A3
A2
A1
A0
A
+
+
B7
B6
B5
B4
B3
B2
B1
B0
B
=
=
A7+B7
A6+B6
A5+B5
A4+B4
A3+B3
A2+B2
A1+B1
A0+B0
A+B

Dependencies

not all loops can be vectorised

- At start of a vector instruction the (4-8) values of the arrays are copied into vector registers.

- These (4-8) array values are then used by a vector functional unit and produces (4-8) output values.

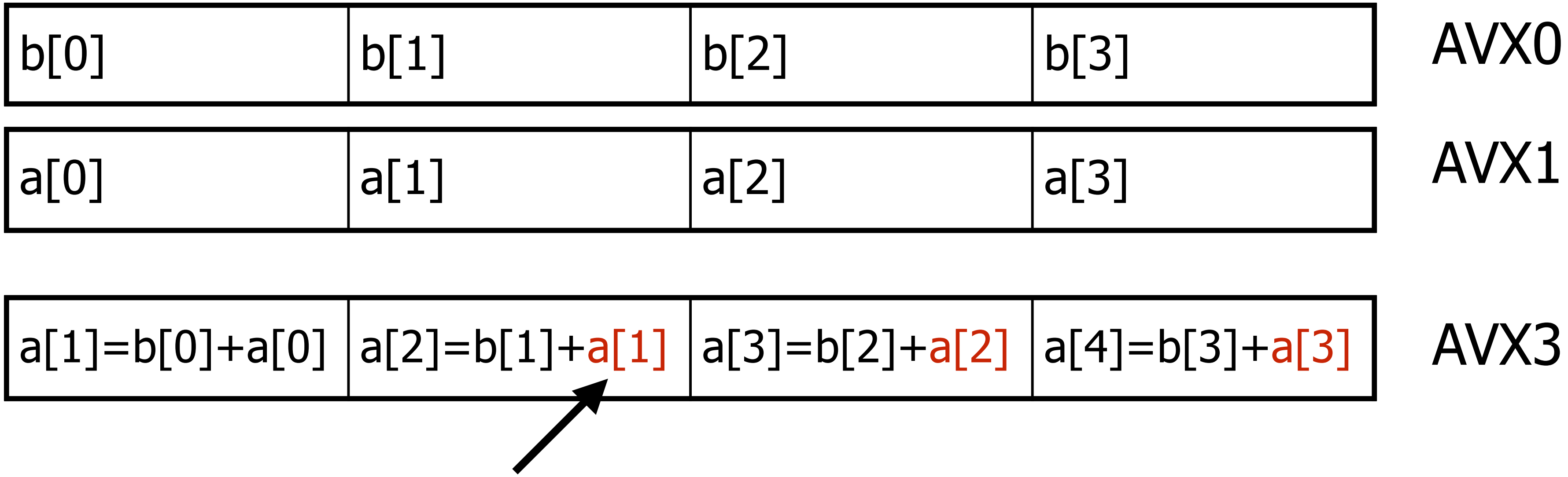
- Within these (4-8) values of the array there must be no dependencies. The vector functional unit uses the values in the registers.

Vectorization wrong usage

```
for (j=0; j<N; j++) {
        a[j+1] = a[j] + b[j];
}
```

```
a[1]=a[0]+b[0]

a[2]=b[1]+a[1]
a[2]=b[1]+a[0]+b[0]
```



b[0]	b[1]	b[2]	b[3]	AVX0
a[0]	a[1]	a[2]	a[3]	AVX1
a[1]=b[0]+a[0]	a[2]=b[1]+a[1]	a[3]=b[2]+a[2]	a[4]=b[3]+a[3]	AVX3

This a[1] is still the original value of a[1] (≠ a[0]+b[0])

Loop Dependencies

- If a loop only has dependencies within an iteration, the loop is considered parallel => multiple iterations can be executed together so long as order within an iteration is preserved
- If a loop has dependencies across iterations, it is not parallel and these dependencies are referred to as “loop-carried”
- Not all loop-carried dependencies imply lack of parallelism

Examples

```
For (i=1000; i>0; i=i-1)
    x[i] = x[i] + s;
```

```
For (i=1; i<=100; i=i+1) {
    A[i+1] = A[i] + C[i];       S1
    B[i+1] = B[i] + A[i+1];     S2
}
```

```
For (i=1; i<=100; i=i+1) {
    A[i] = A[i] + B[i];         S1
    B[i+1] = C[i] + D[i];       S2
}
```

```
For (i=1000; i>0; i=i-1)
    x[i] = x[i-3] + s;          S1
```

Examples

```
For (i=1000; i>0; i=i-1)
     x[i] = x[i] + s;
```

No dependences

```
For (i=1; i<=100; i=i+1) {
     A[i+1] = A[i] + C[i];          S1
     B[i+1] = B[i] + A[i+1];        S2
}
```

S2 depends on S1 in the same iteration
S1 depends on S1 from prev iteration
S2 depends on S2 from prev iteration

```
For (i=1; i<=100; i=i+1) {
     A[i] = A[i] + B[i];            S1
     B[i+1] = C[i] + D[i];          S2
}
```

S1 depends on S2 from prev iteration

```
For (i=1000; i>0; i=i-1)
     x[i] = x[i-3] + s;          S1
```

S1 depends on S1 from 3 prev iterations
Referred to as a recursion
Dependence distance 3; limited parallelism

Constructing Parallel/Vector Loops

If loop-carried dependencies are not cyclic (S1 depending on S1 is cyclic), loops can be restructured to be parallel

```
For (i=1; i<=100; i=i+1) {
    A[i] = A[i] + B[i];            S1
    B[i+1] = C[i] + D[i];          S2
}
```

```
A[1] = A[1] + B[1];
For (i=1; i<=99; i=i+1) {
    B[i+1] = C[i] + D[i];              S3
    A[i+1] = A[i+1] + B[i+1];          S4
}
B[101] = C[100] + D[100];
```

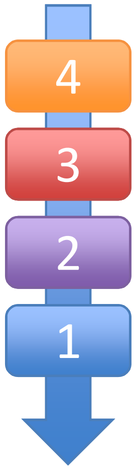
S1 depends on S2 from prev iteration

S4 depends on S3 of same iteration

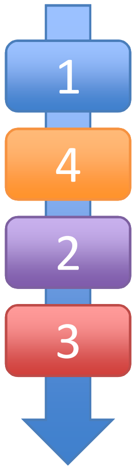
Summary

- Superscalar: start several instructions per cycle.
- Our of order: reshuffle instructions for optimal use of all functional units.
- Pipelining: work on instructions in parallel.
- Vectorization: parallel computation on short arrays.

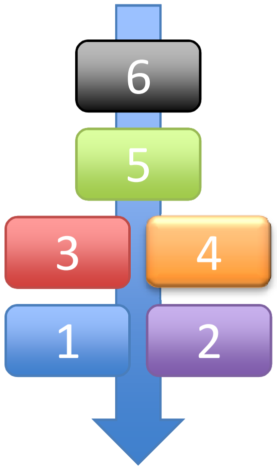
Summary



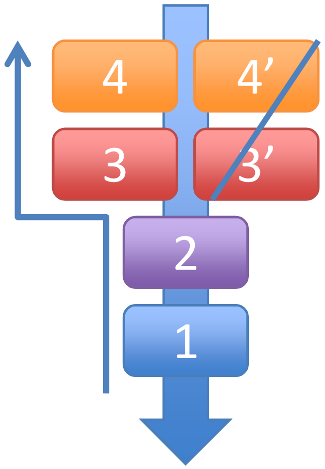
Pipelining



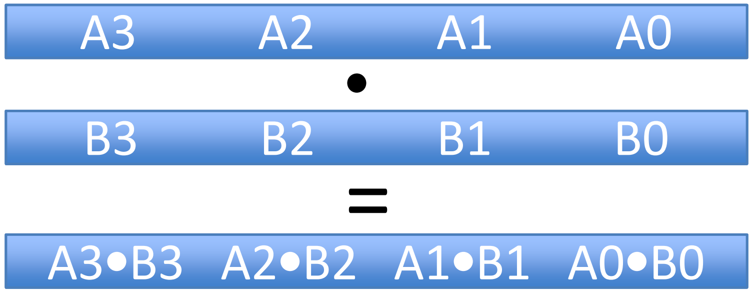
Out-of-order



Superscalarity



Branch Prediction

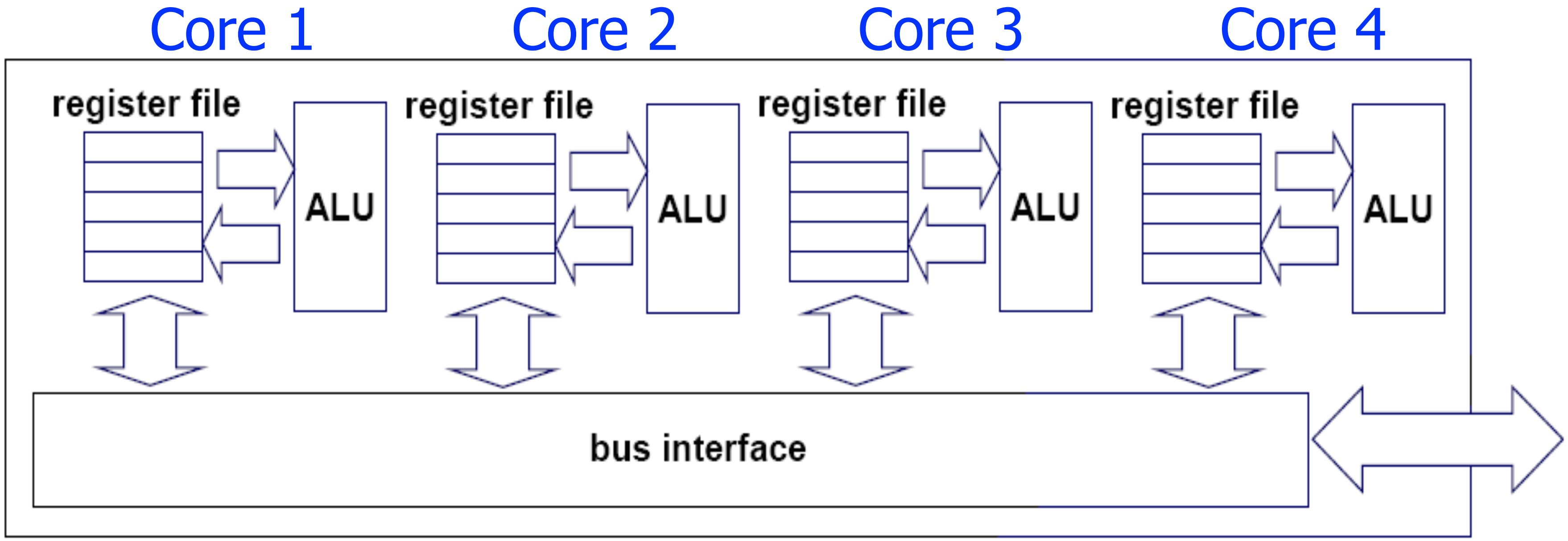


SIMD

Multi core and all the above

Multi-core architectures

- A trend in computer architecture since ~2012: Replicate multiple processor cores on a single die.



Multi-core CPU chip

The memory hierarchy and cores

- If simultaneous multithreading (software solution) only:
    - all caches shared

- Multi-core chips (hardware solution):
    - L1 caches private
    - L2 caches private in some architectures and shared in others
    - L3 shared cache among the cores

- Main memory is always shared between all cores.

Private vs shared caches?

- Advantages/disadvantages?

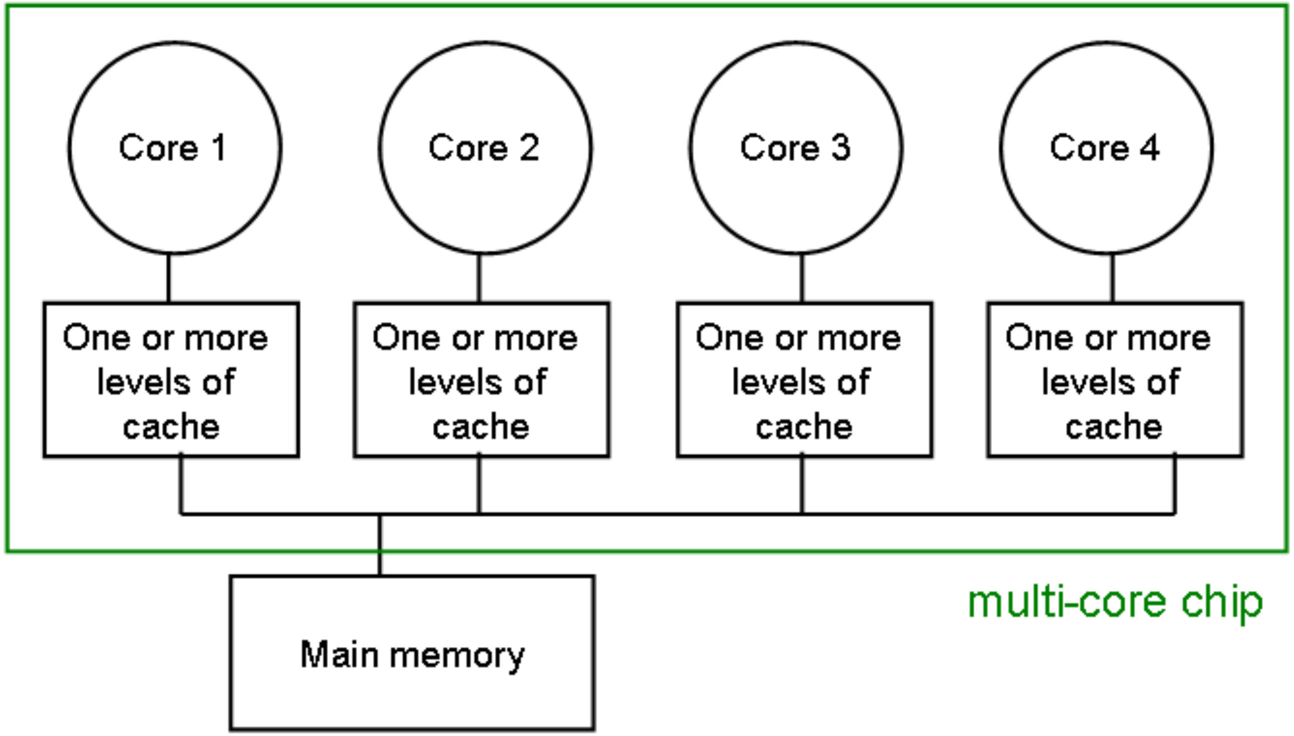
Private vs shared caches

- Advantages of private:
  - They are closer to core, so faster access
  - Reduces contention

- Advantages of shared:
  - Threads on different cores can share the same cache data
  - More cache space available if a single (or a few) high-performance thread runs on the system

The cache coherence problem

- Since we have private caches:
  How to keep the data consistent across caches?
- Each core should perceive the memory as a monolithic array, shared by all the cores


Core 1
Core 2
Core 3
Core 4
One or more levels of cache
One or more levels of cache
One or more levels of cache
One or more levels of cache
multi-core chip
Main memory

The cache coherence problem

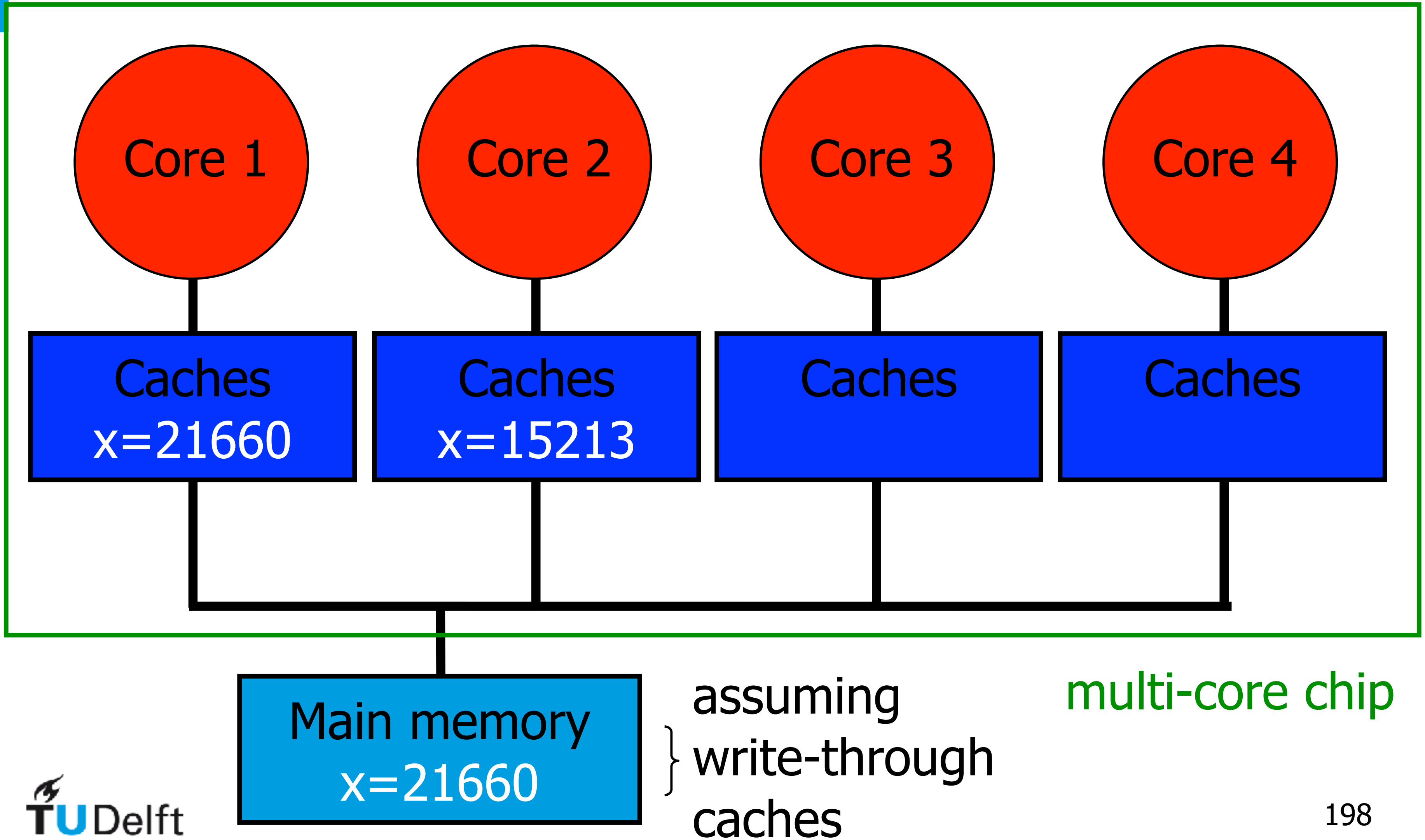
Suppose variable x initially contains 15213

Core 1 reads x

Core 2 reads x

Core 1 writes to x, setting it to 21660

Core 2 attempts to read x… gets a stale copy



assuming write-through caches

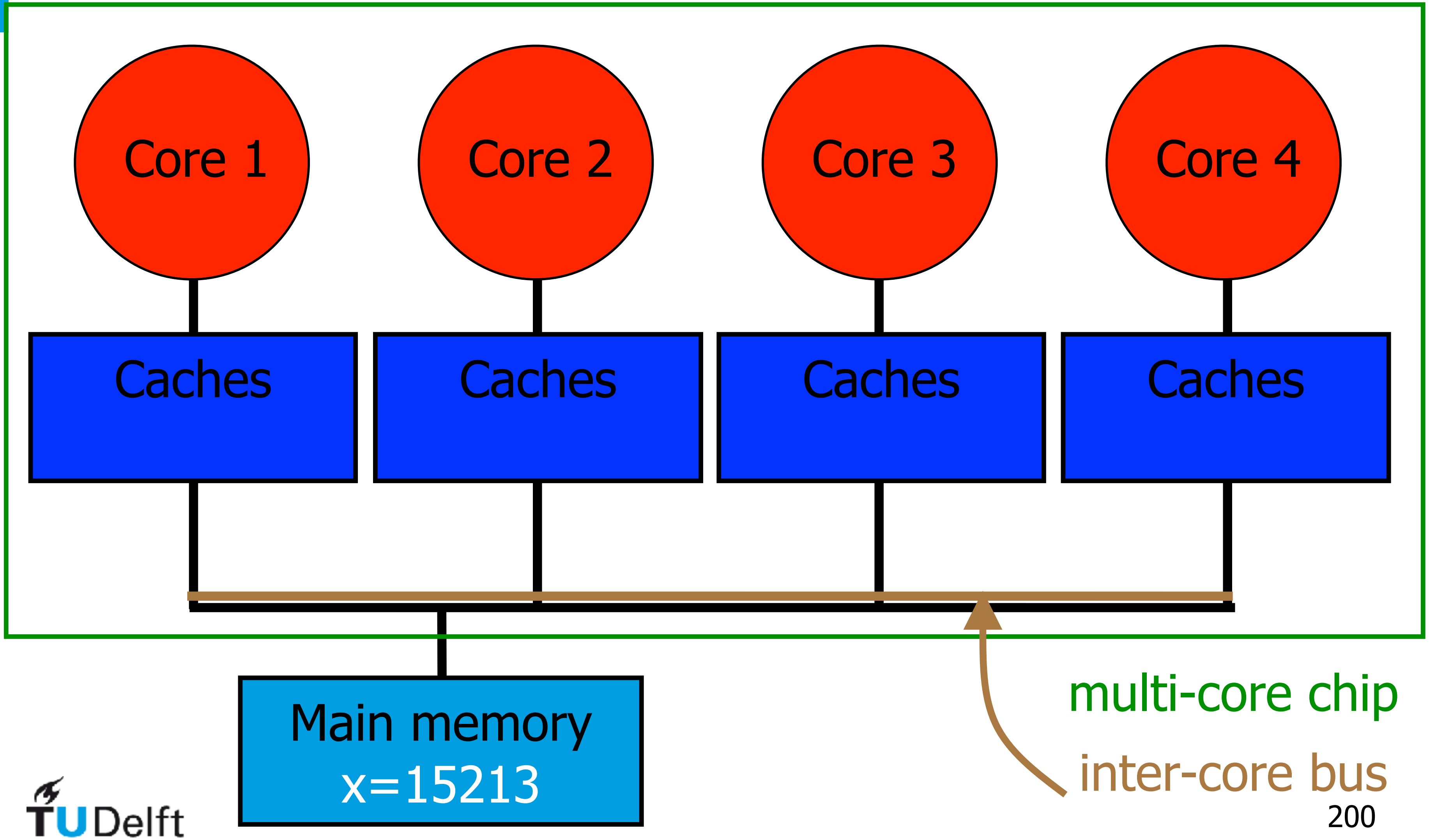
TU Delft

Solutions for cache coherence

- This is a general problem with multiprocessors, not limited just to multi-core
- There exist many solution algorithms, coherence protocols, etc.

- A simple solution:
  invalidation-based protocol with *snooping*

Inter-core bus



multi-core chip

inter-core bus

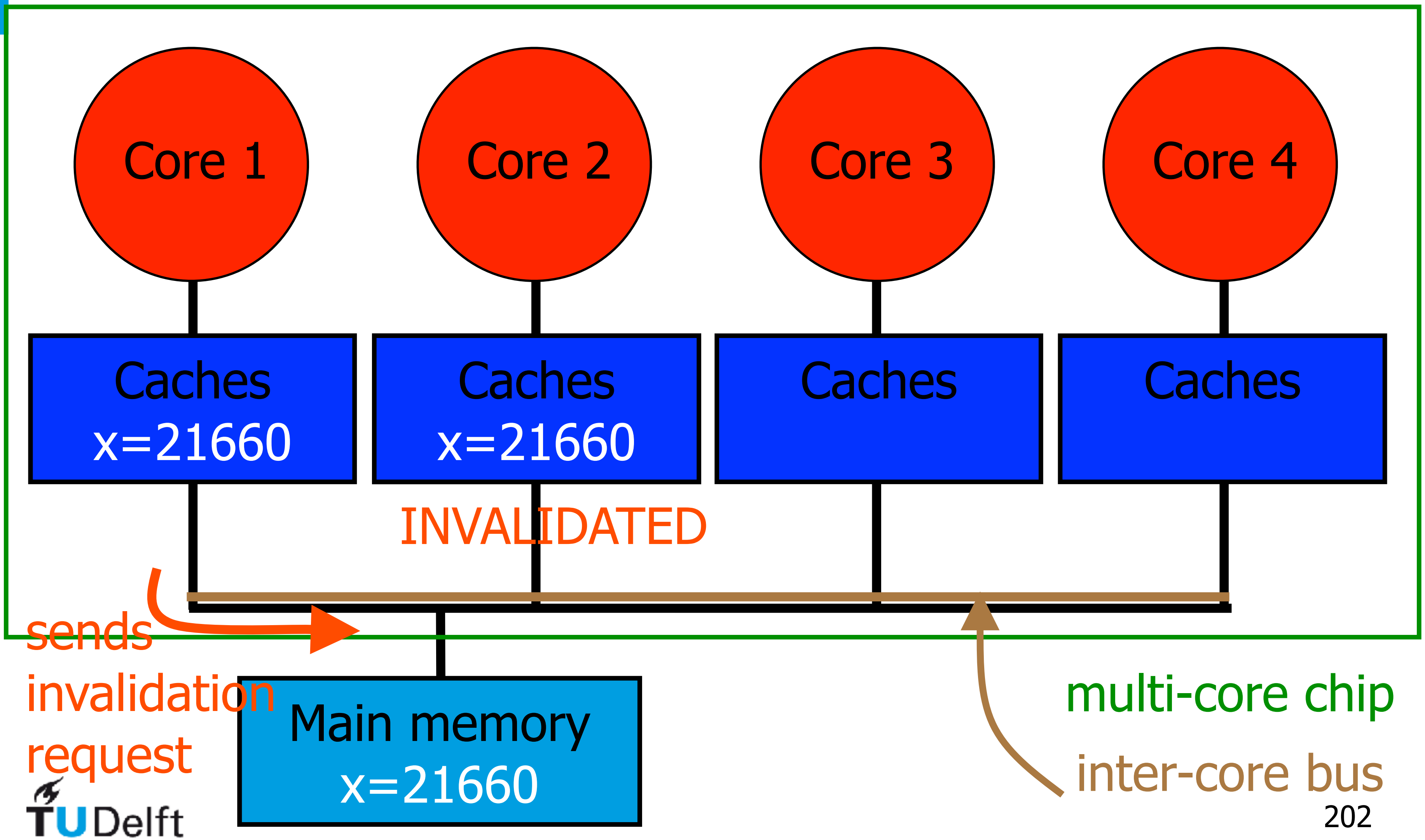
Invalidation protocol with snooping

- Invalidation:
  If a core writes to a data item, all other copies of this data item in other caches are *invalidated*

- Snooping:
  All cores continuously “snoop” (monitor) the bus connecting the cores.

The cache coherence problem

[illegible] read x [illegible] copy.



Core 1

Core 2

Core 3

Core 4

Caches
x=21660

Caches
x=21660

Caches

Caches

INVALIDATED

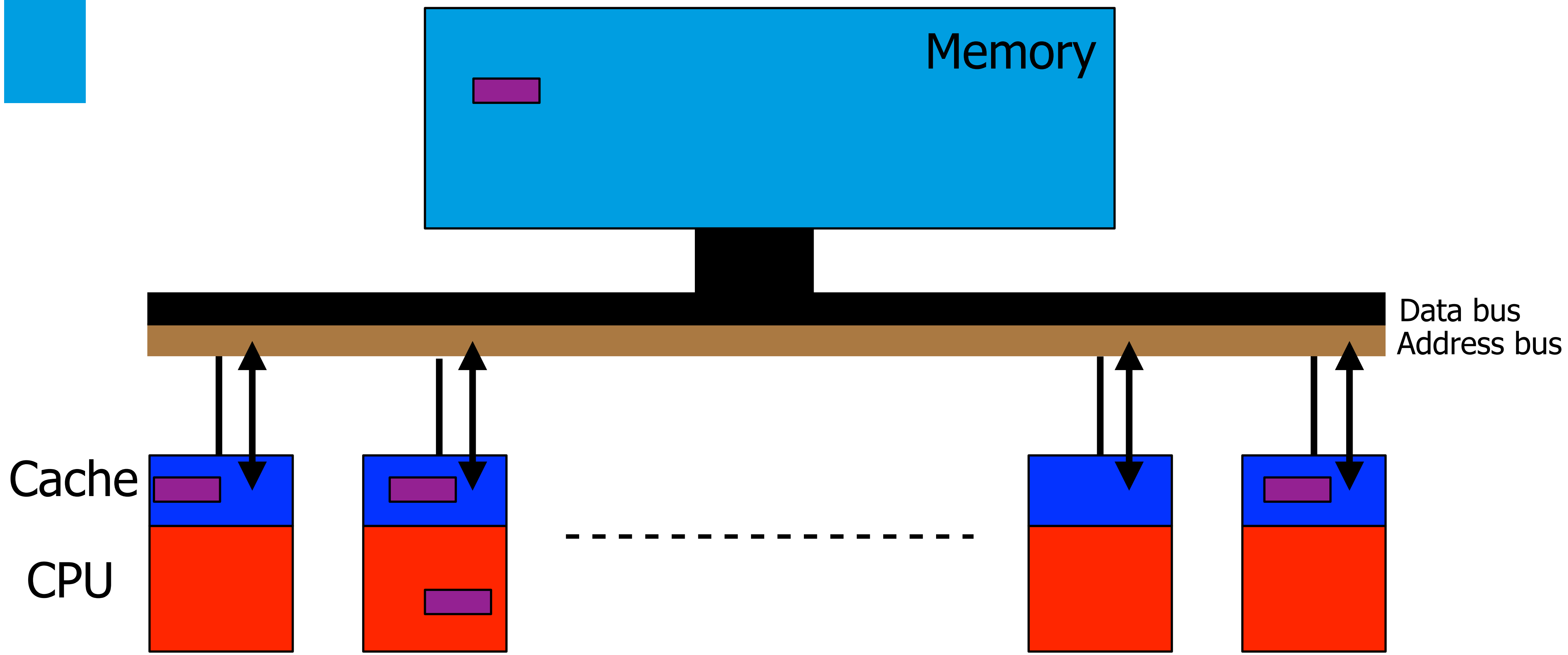
sends invalidation request

Main memory
x=21660

multi-core chip

inter-core bus

Cache Coherence - Snooping



With a snooping protocol, ALL address traffic on the bus is monitored by ALL processors

Cache Coherence Protocols

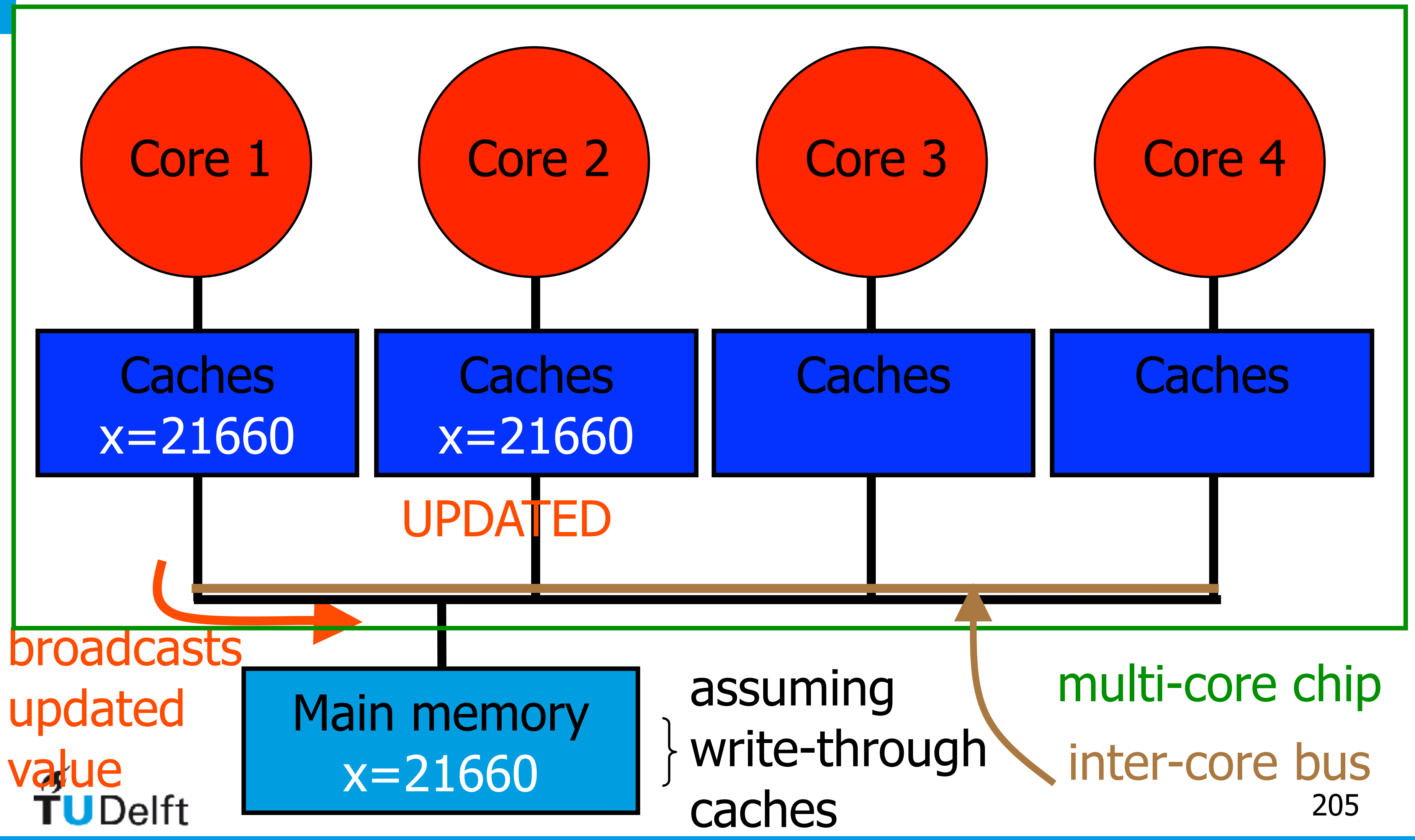
- Directory-based: A single location (directory) keeps track of the sharing status of a block of memory

- Snooping: Every cache block is accompanied by the sharing status of that block – all cache controllers monitor the shared bus so they can update the sharing status of the block, if necessary

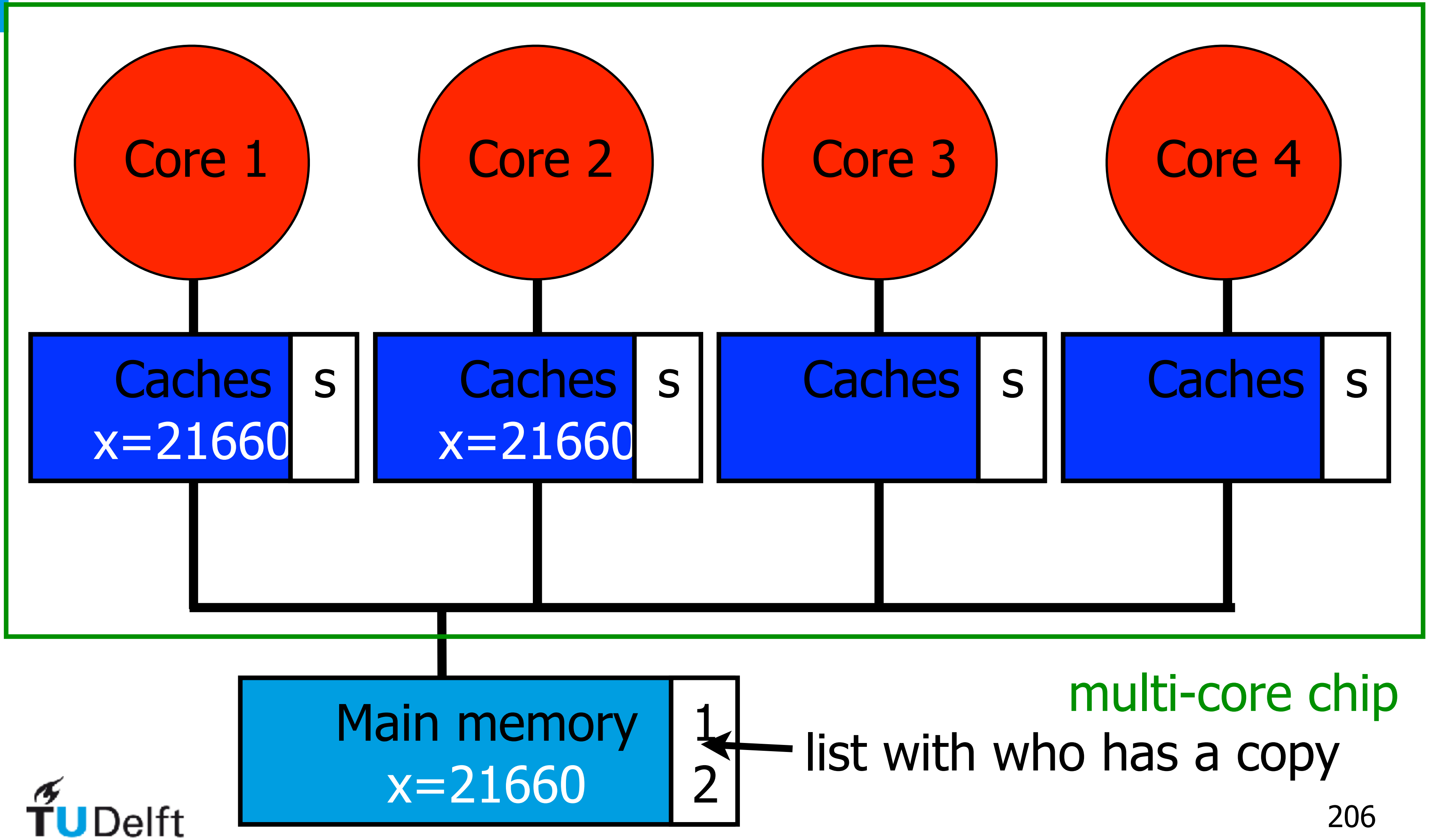
- Write-invalidate: a processor gains exclusive access of a block before writing by invalidating all other copies
- Write-update: when a processor writes, it updates other shared copies of that block

Alternative to invalidate protocol: update protocol

Core 1 writes x=21660:



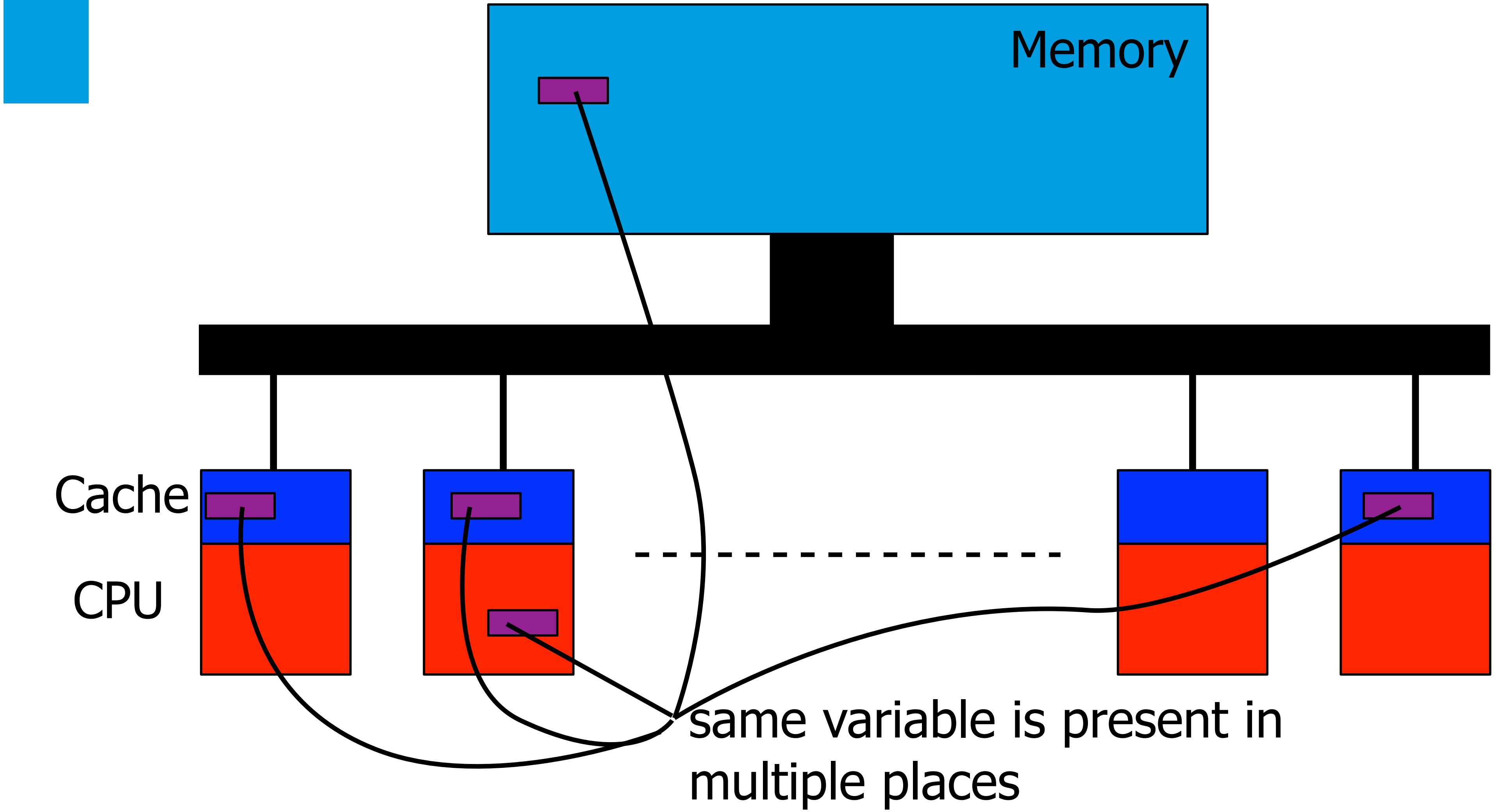
Alternative to snoop: Directory based coherency



multi-core chip

list with who has a copy

cache coherency ensures that one always gets the right value ... regardless of where the data is



Invalidation vs Update

- Multiple writes to the same location
  - invalidation: only the first time
  - update: must broadcast each write
    (which includes new variable value)

- Invalidation generally performs better:
  it generates less bus traffic

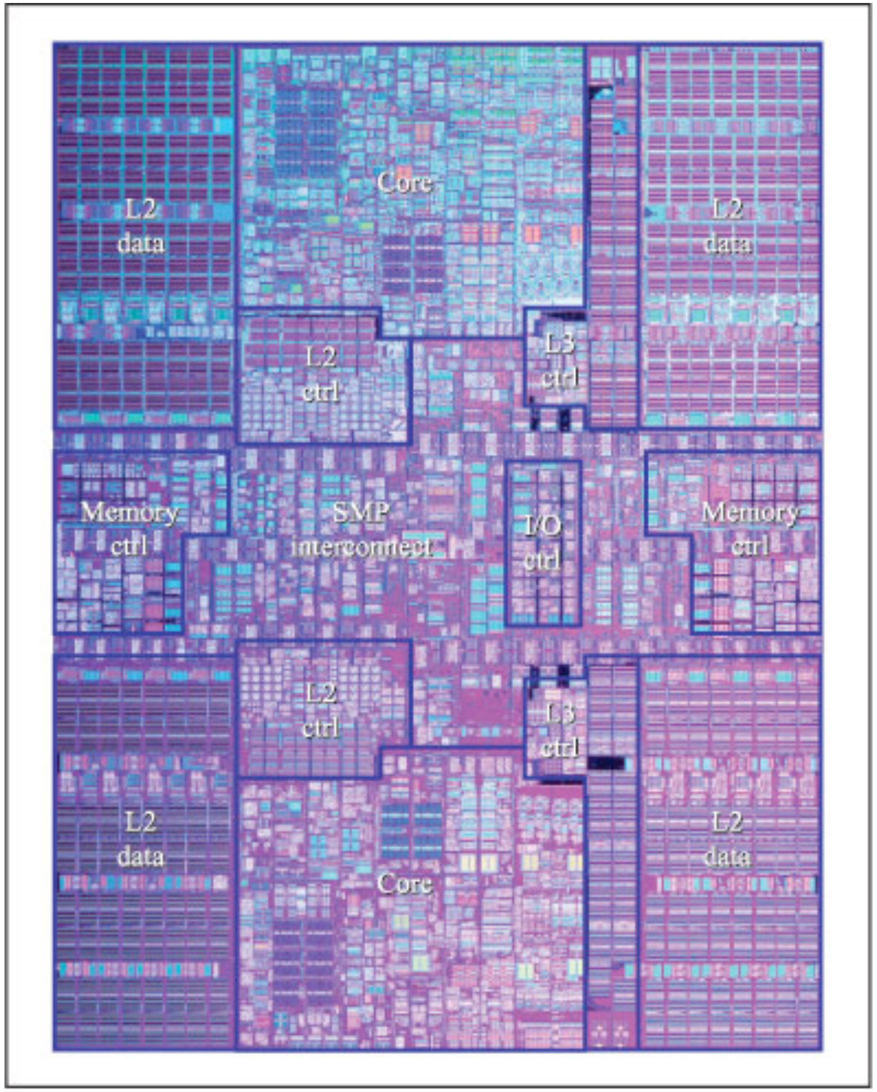
Current and Future Hardware

Modern CPU's and Systems

- IBM Power
- Intel Xeon
- AMD Zen and Ryzen
- GPGPU's
- FPGA's
- ARM

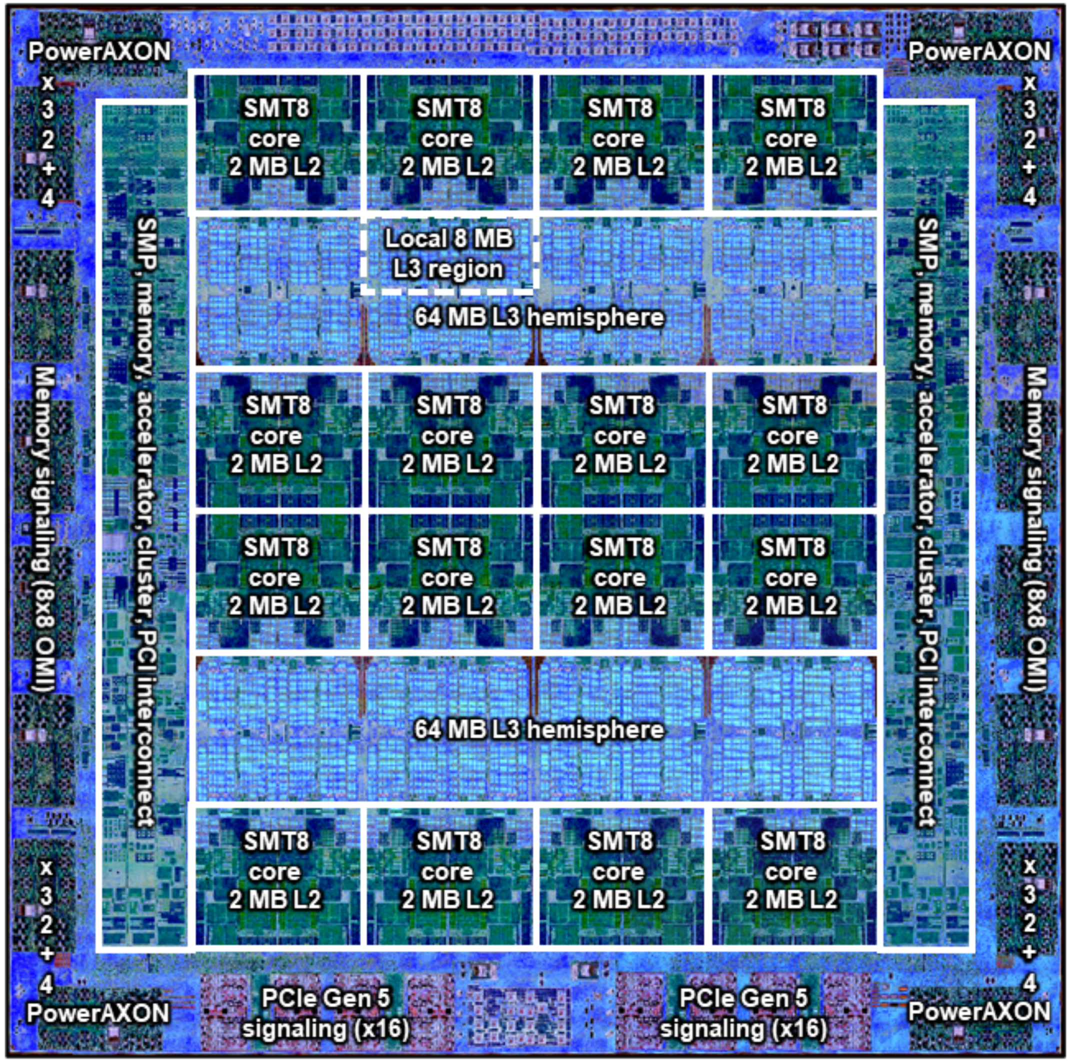
Power6

- Decimal Floating point unit
- In Order execution:
  - faster clock (4.7 GHz)
  - larger cache
  - SMT can deal with unused units
  - but compiler has to make good code!



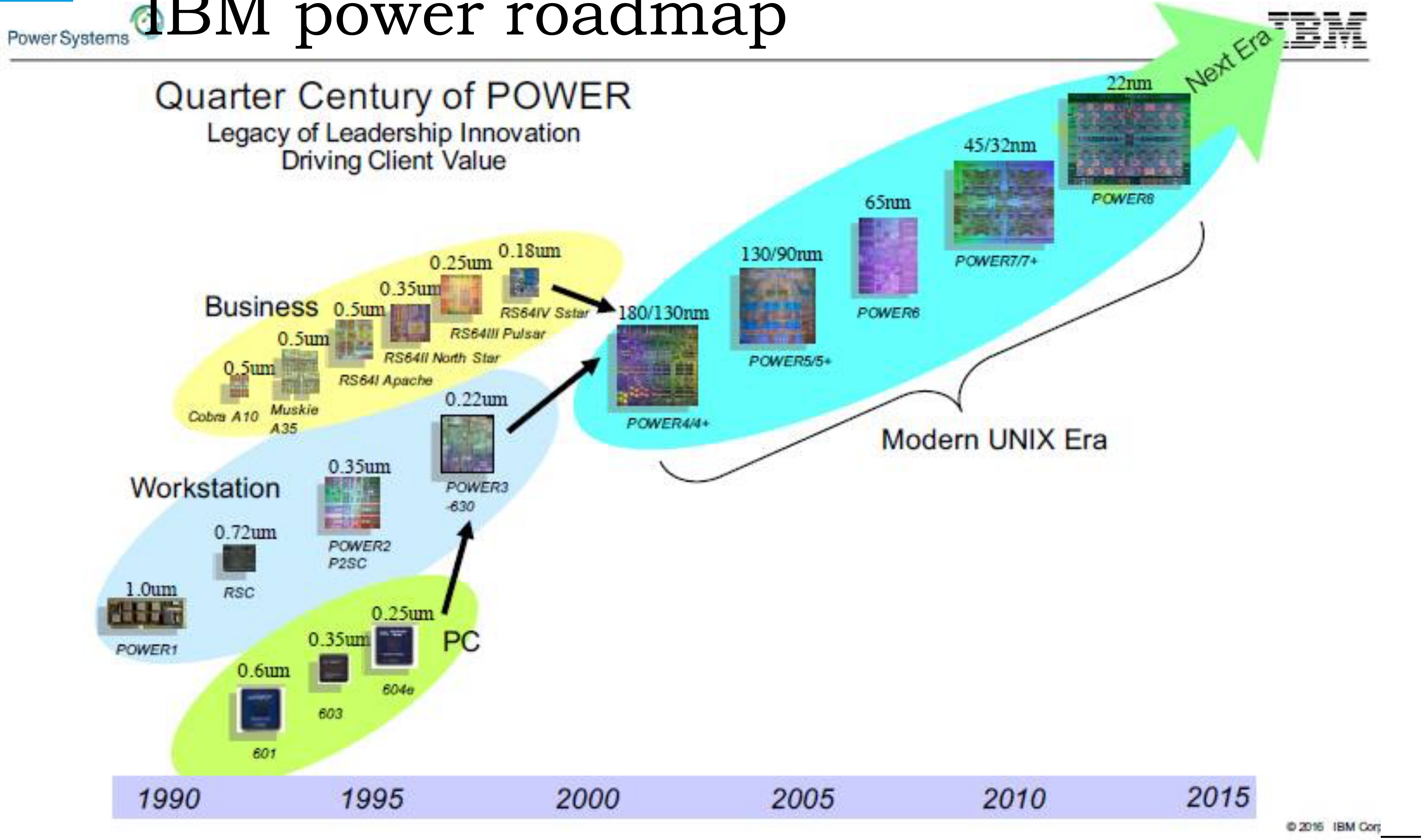
Power 10

8 SMT, 7 nm


PowerAXON
PowerAXON
x 32 + 4
x 32 + 4
SMT8 core 2 MB L2
Local 8 MB L3 region
64 MB L3 hemisphere
SMP, memory, accelerator, cluster, PCI interconnect
SMP, memory, accelerator, cluster, PCI interconnect
Memory signaling (8x8 OMI)
Memory signaling (8x8 OMI)
64 MB L3 hemisphere
x 32 + 4
x 32 + 4
PowerAXON
PCIe Gen 5 signaling (x16)
PCIe Gen 5 signaling (x16)
PowerAXON

Each socket holds one Power10 single chip module (SCM). An SCM can contain 10, 12, or 15 Power10 processor cores.

IBM power roadmap



Power Systems

IBM

Quarter Century of POWER

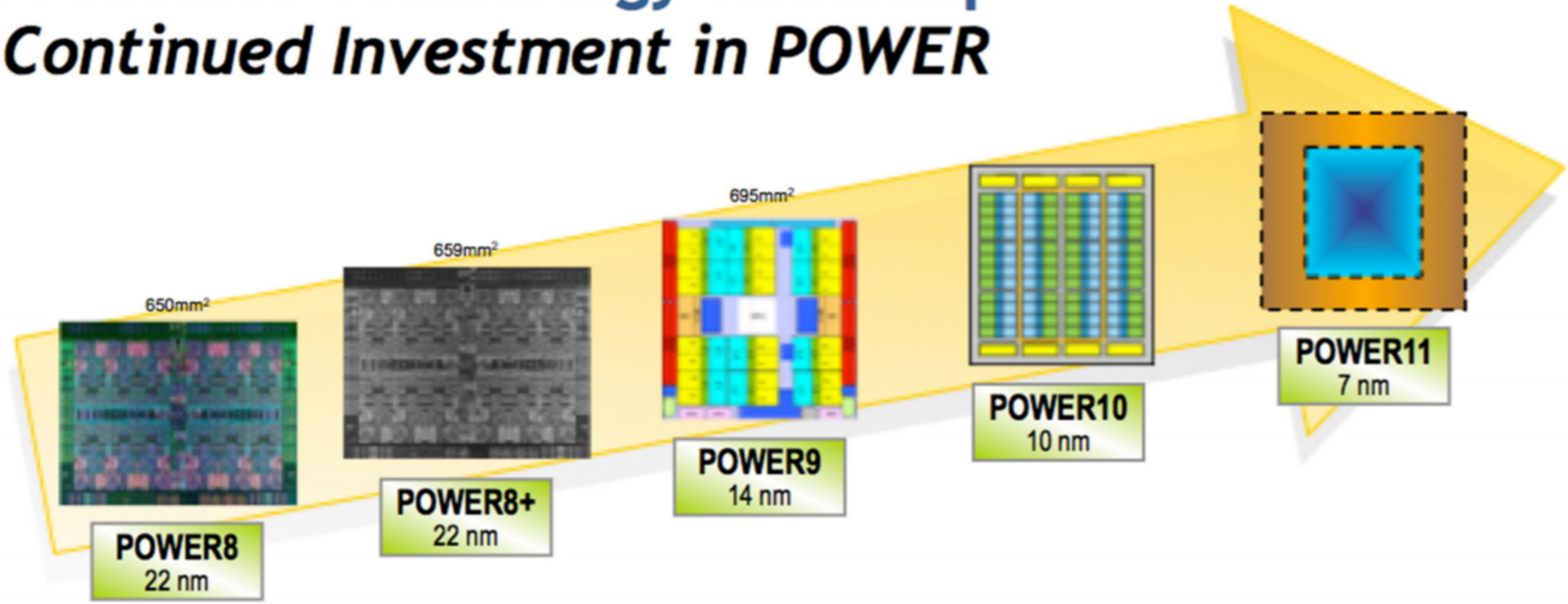
Legacy of Leadership Innovation
Driving Client Value

© 2016 IBM Corp

TUDelft

Processor Technology Roadmap

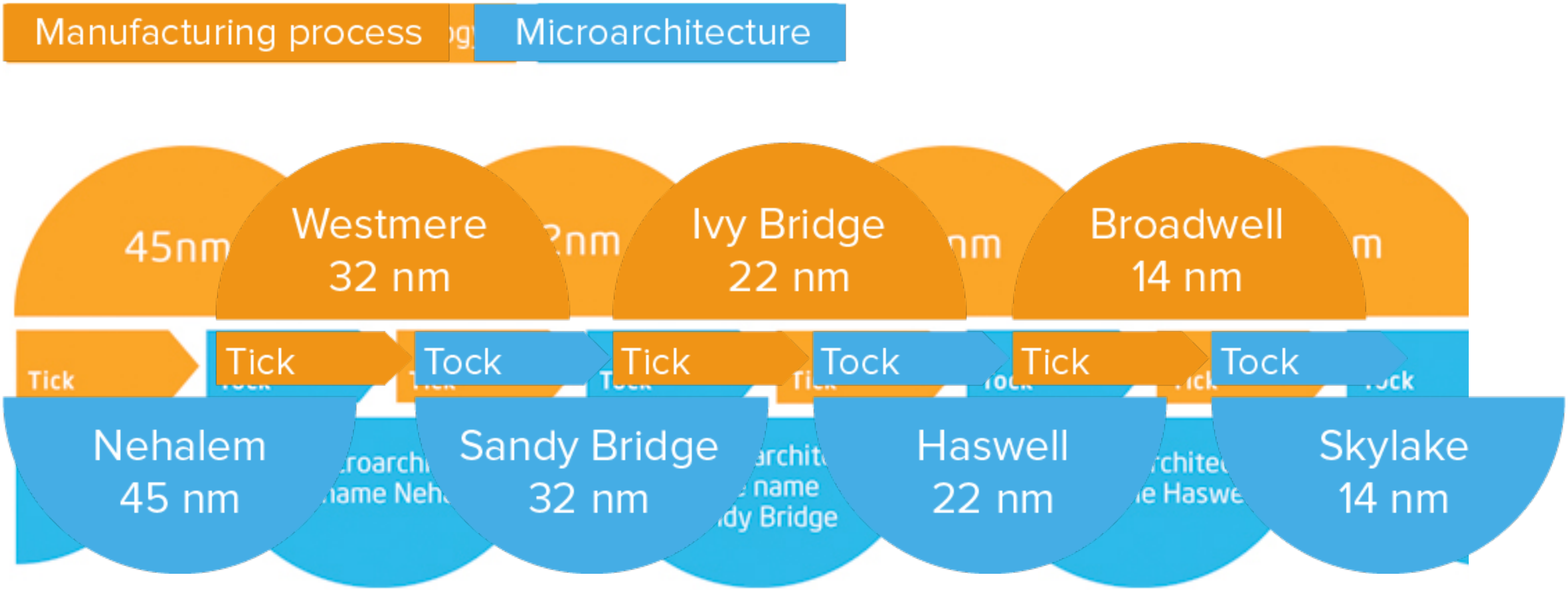
Continued Investment in POWER



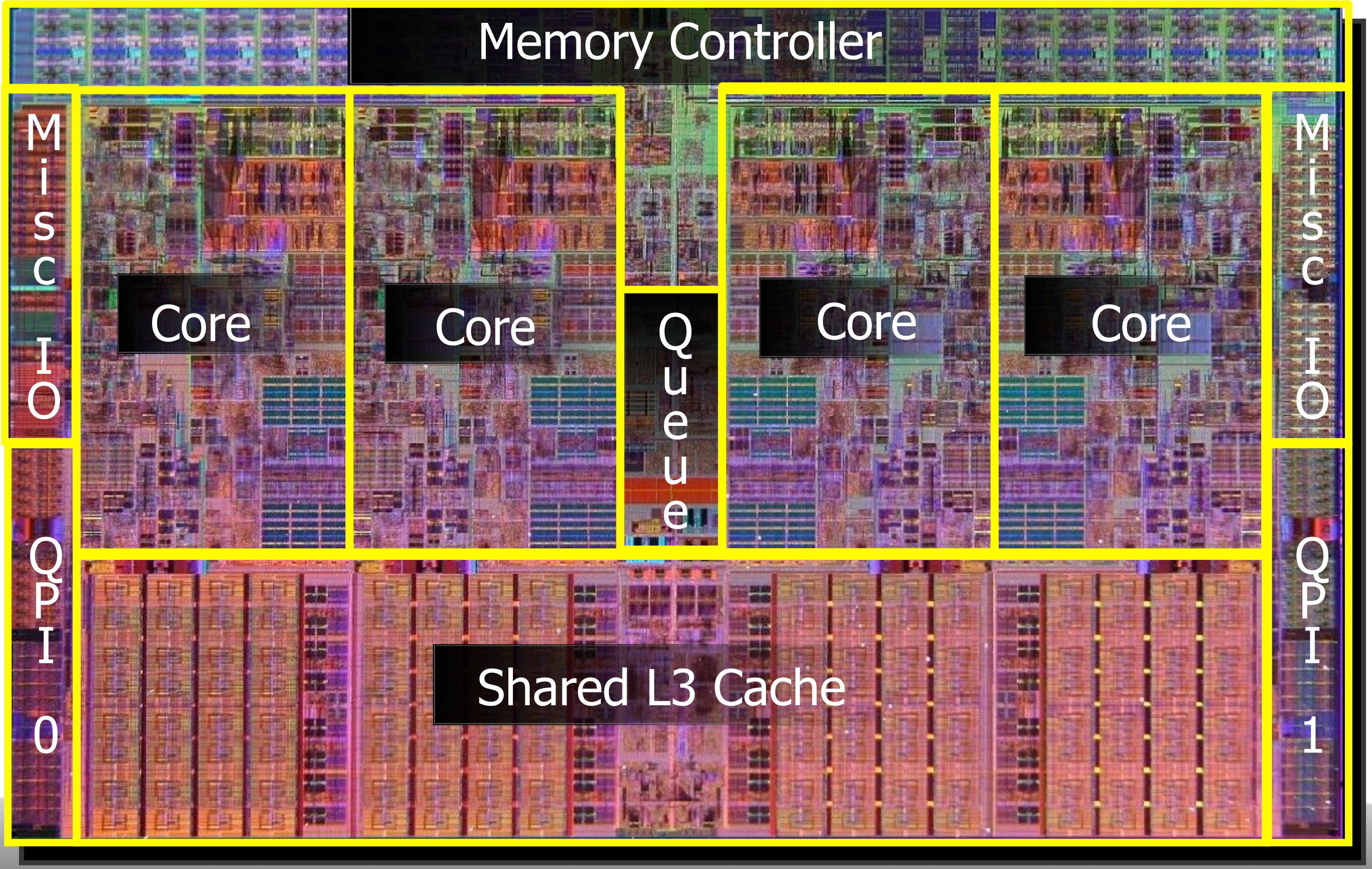
2014	2016	2017	2019-2020	Future
- 12 Cores - SMT8 - 2X DPFP - PCIE Gen 3 - Coprocessor (CAPI) - Enhanced Prefetch	- NVLINK1.0 - 2X CAPI	- 24 Cores - New µArchitecture - Direct-attach DDR4 - Gen4 PCIe - CAPI 2.0 - NVLINK2.0	- 48 Cores - New µArchitecture - Enhanced Memory - OpenCAPI - NVLINK3	- >48 Cores - New µArchitecture - 2x SIMD width - Future NVLINK - Future OpenCAPI

Intel

Tick-Tock


Manufacturing process
Microarchitecture
Westmere
32 nm
Ivy Bridge
22 nm
Broadwell
14 nm
Tick
Tock
Tick
Tock
Tick
Tock
Nehalem
45 nm
Sandy Bridge
32 nm
Haswell
22 nm
Skylake
14 nm
45nm
Tick

The Intel Nehalem (Core architecture) Processor



QPI: Intel® QuickPath Interconnect (Intel® QPI)

A Modular Design for Flexibility

Skylake 2018

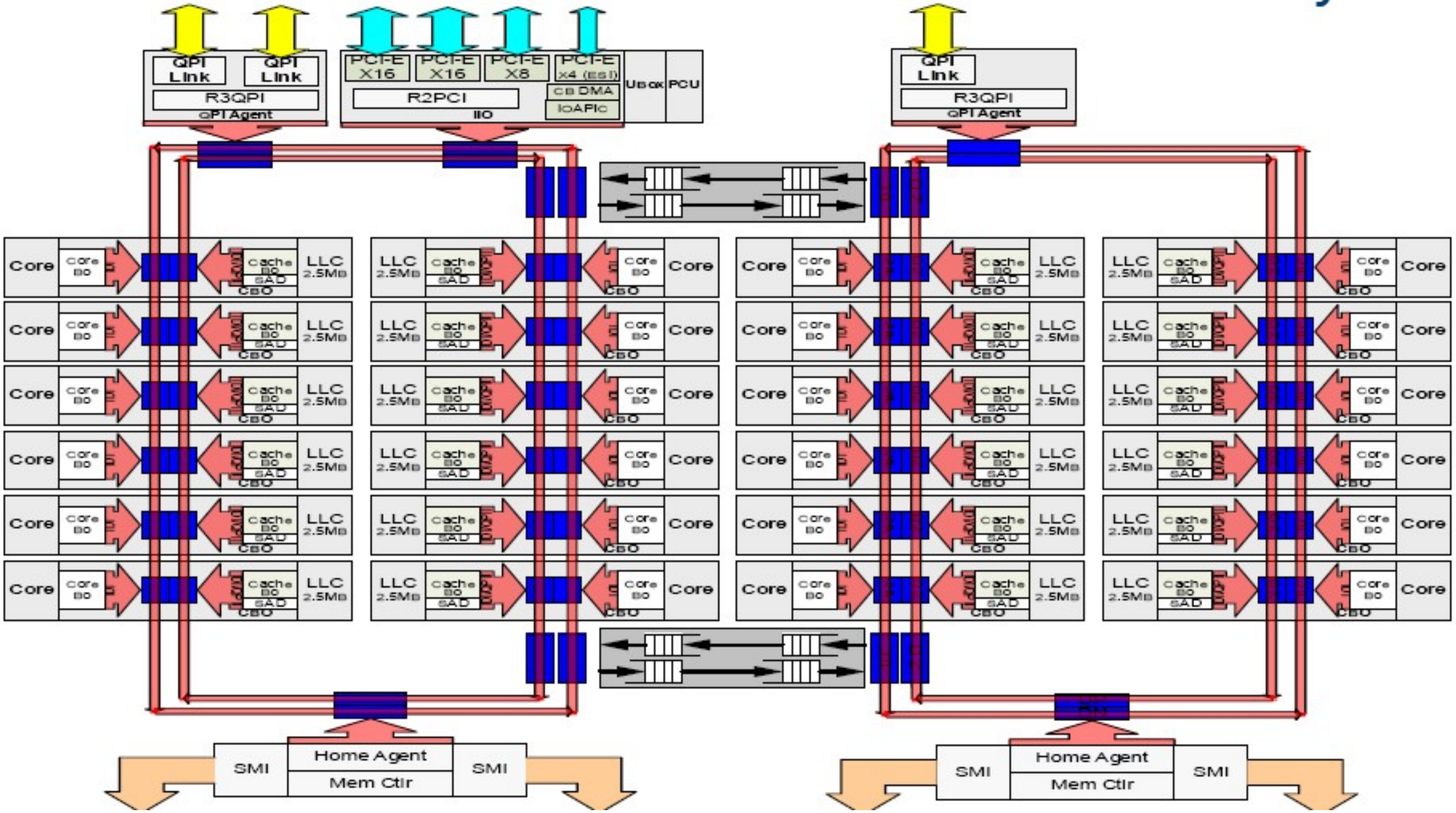
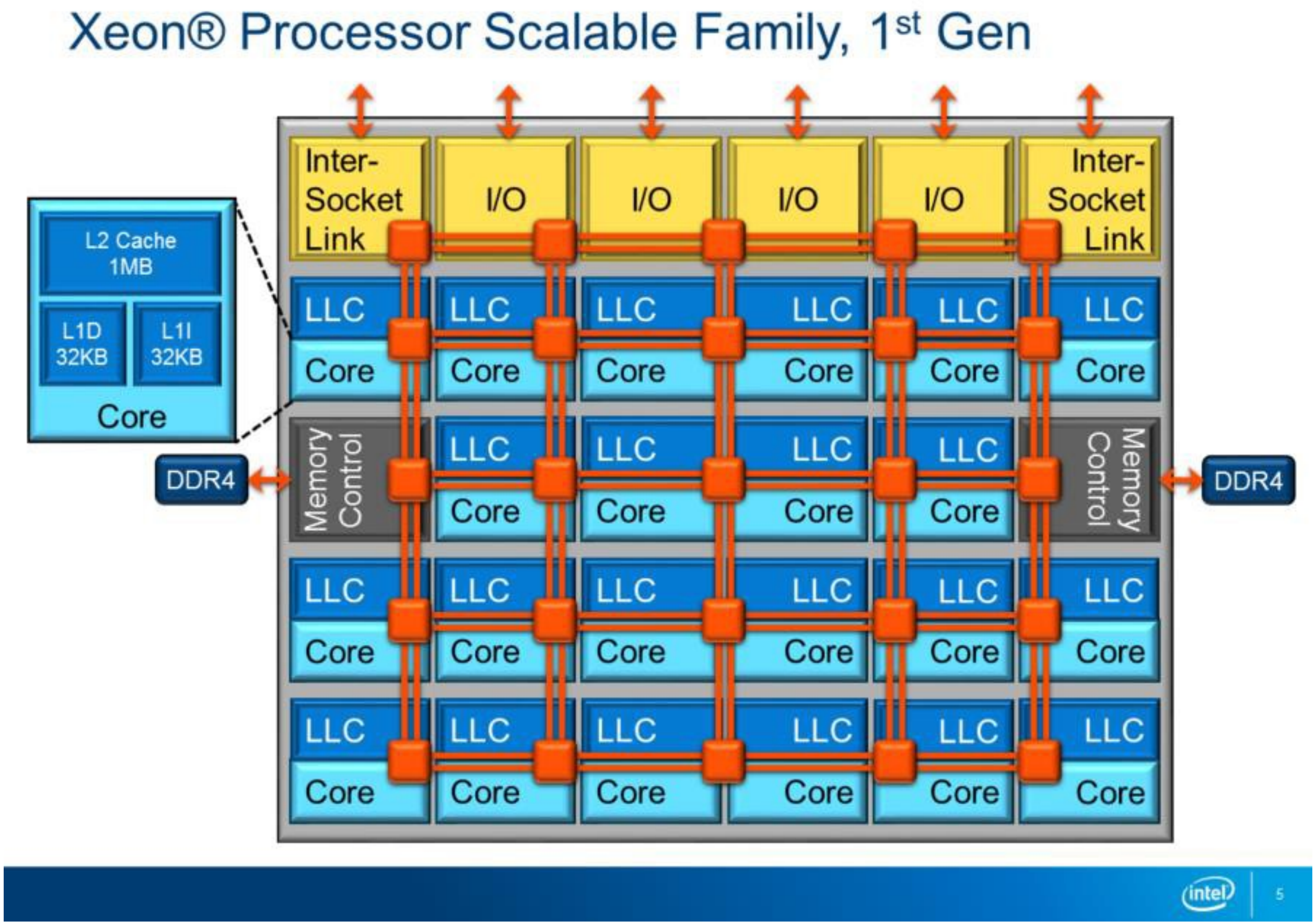


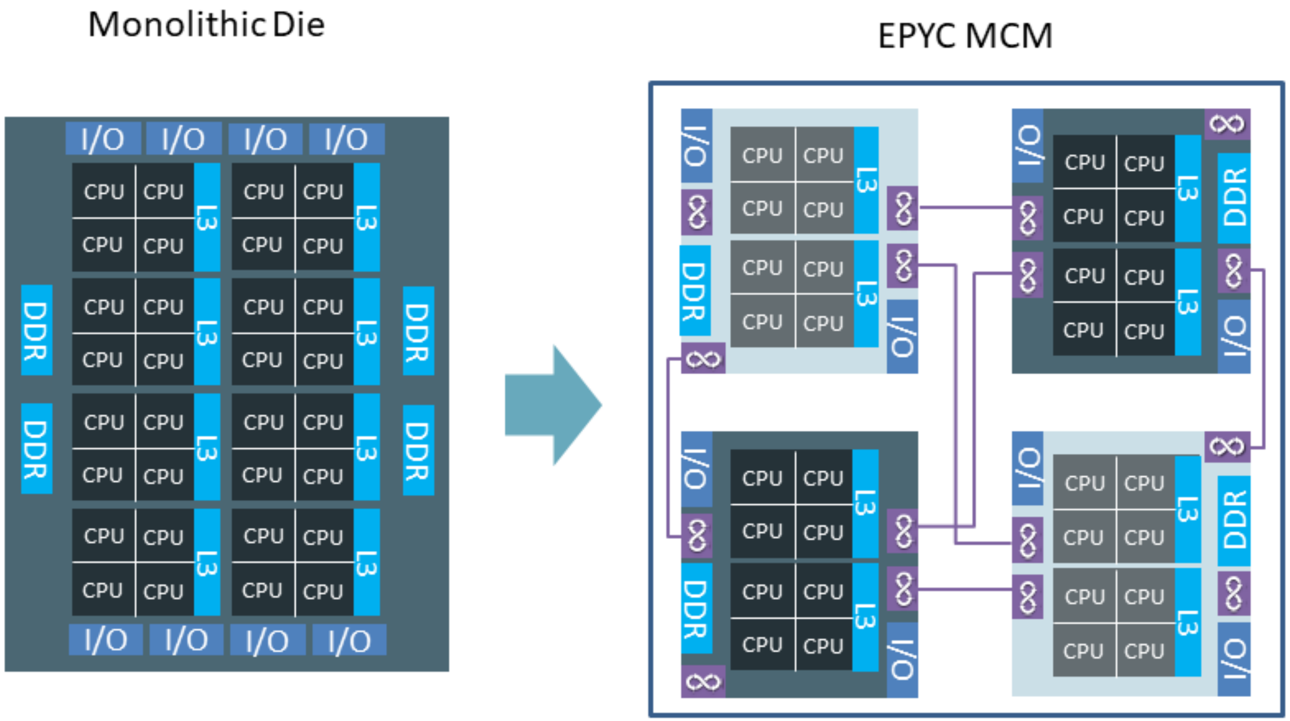
Figure 4: Intel Skylake-X Mesh Fabric



Source: Intel

AMD

AMD design


Monolithic Die

EPYC MCM

32C Die Cost

1.0X

4 x 8C Die Cost

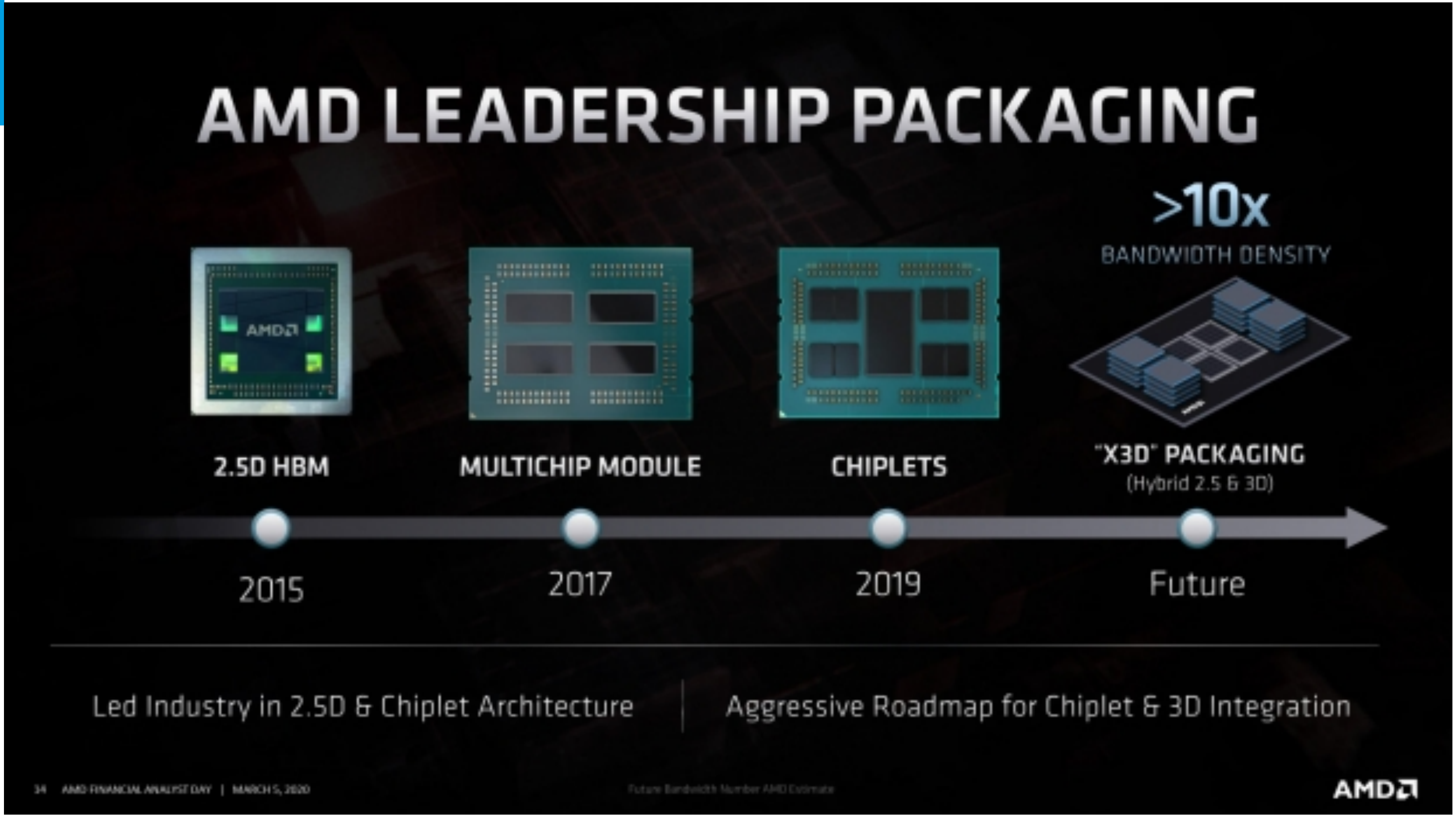
0.59X[1]

[1] Based on AMD internal yield model using historical defect density data for mature technologies.

Source: AMD

https://www.amd.com/system/files/2018-03/AMD-Optimizes-EPYC-Memory-With-NUMA.pdf

AMD Chiplets



AMD LEADERSHIP PACKAGING

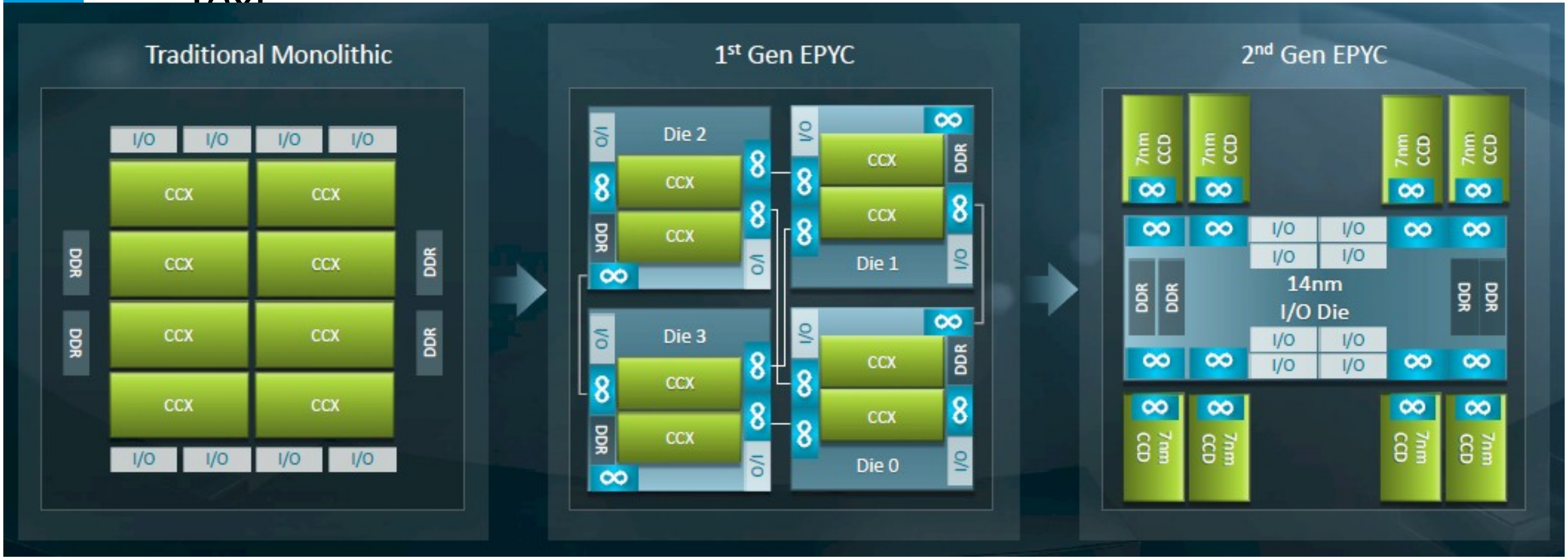
>10x
BANDWIDTH DENSITY
2.5D HBM
MULTICHIP MODULE
CHIPLETS
"X3D" PACKAGING
(Hybrid 2.5 & 3D)
2015
2017
2019
Future

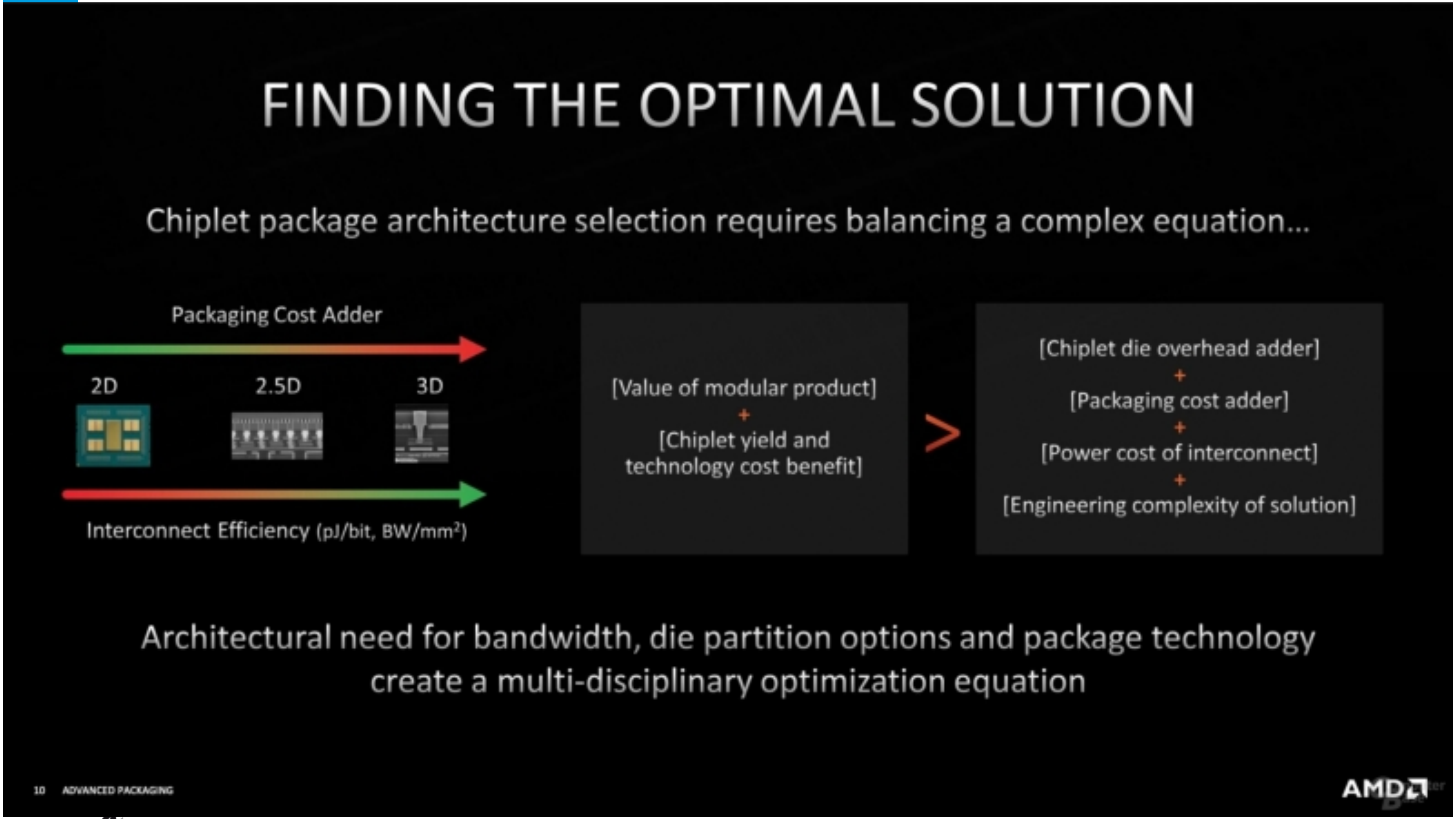
Led Industry in 2.5D & Chiplet Architecture | Aggressive Roadmap for Chiplet & 3D Integration

34 AMD FINANCIAL ANALYST DAY | MARCH 5, 2020

Future Bandwidth Number AMD Estimate

Text


Traditional Monolithic
I/O
CCX
DDR
1st Gen EPYC
Die 2
Die 3
Die 1
Die 0
CCX
I/O
DDR
∞
2nd Gen EPYC
7nm CCD
∞
I/O
DDR
14nm I/O Die



FINDING THE OPTIMAL SOLUTION

Chiplet package architecture selection requires balancing a complex equation…

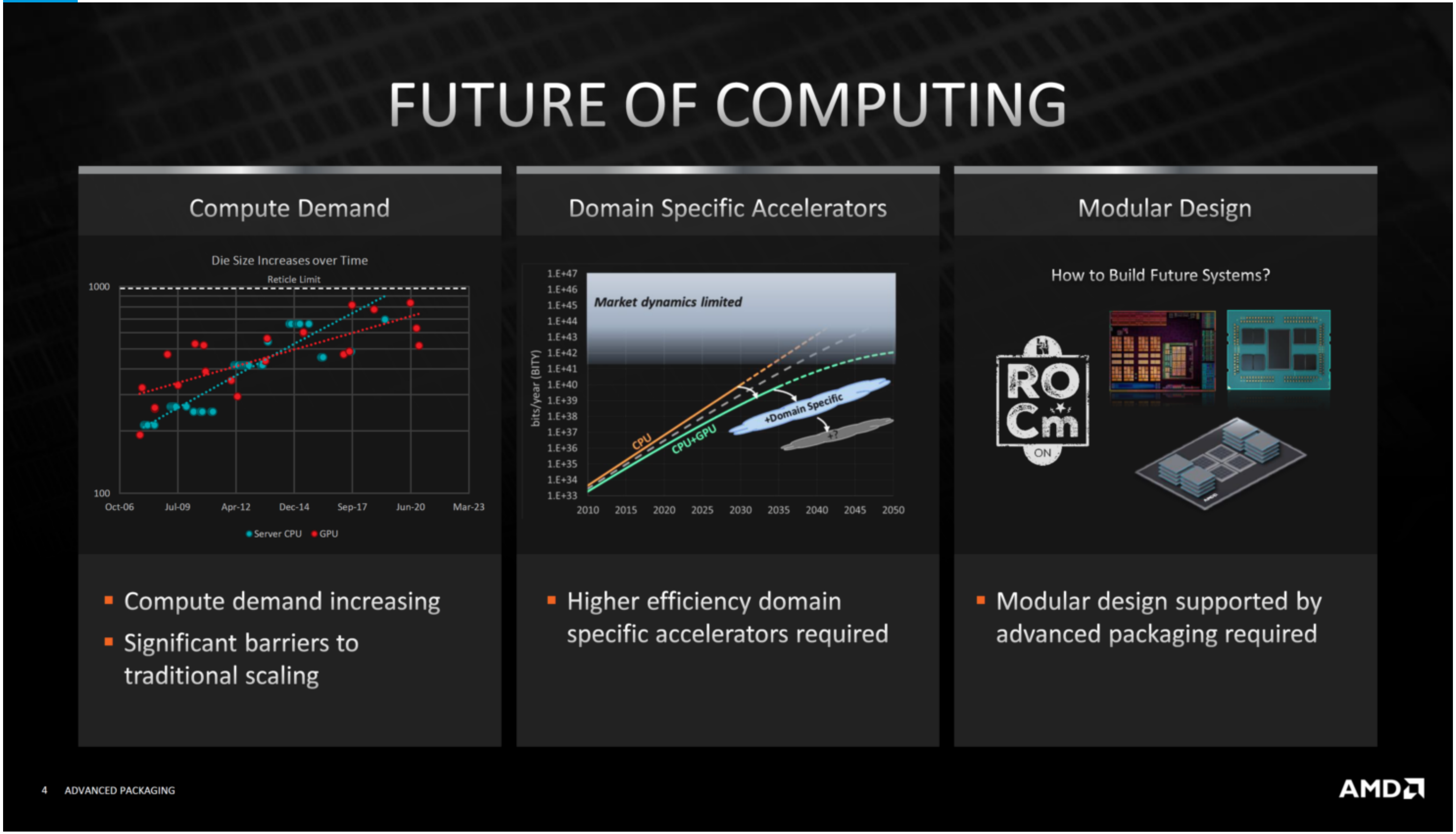
[Value of modular product]
+
[Chiplet yield and technology cost benefit]

>

[Chiplet die overhead adder]
+
[Packaging cost adder]
+
[Power cost of interconnect]
+
[Engineering complexity of solution]

Architectural need for bandwidth, die partition options and package technology create a multi-disciplinary optimization equation

AMD marketing



FUTURE OF COMPUTING

Compute Demand

- Compute demand increasing
- Significant barriers to traditional scaling

Domain Specific Accelerators

- Higher efficiency domain specific accelerators required

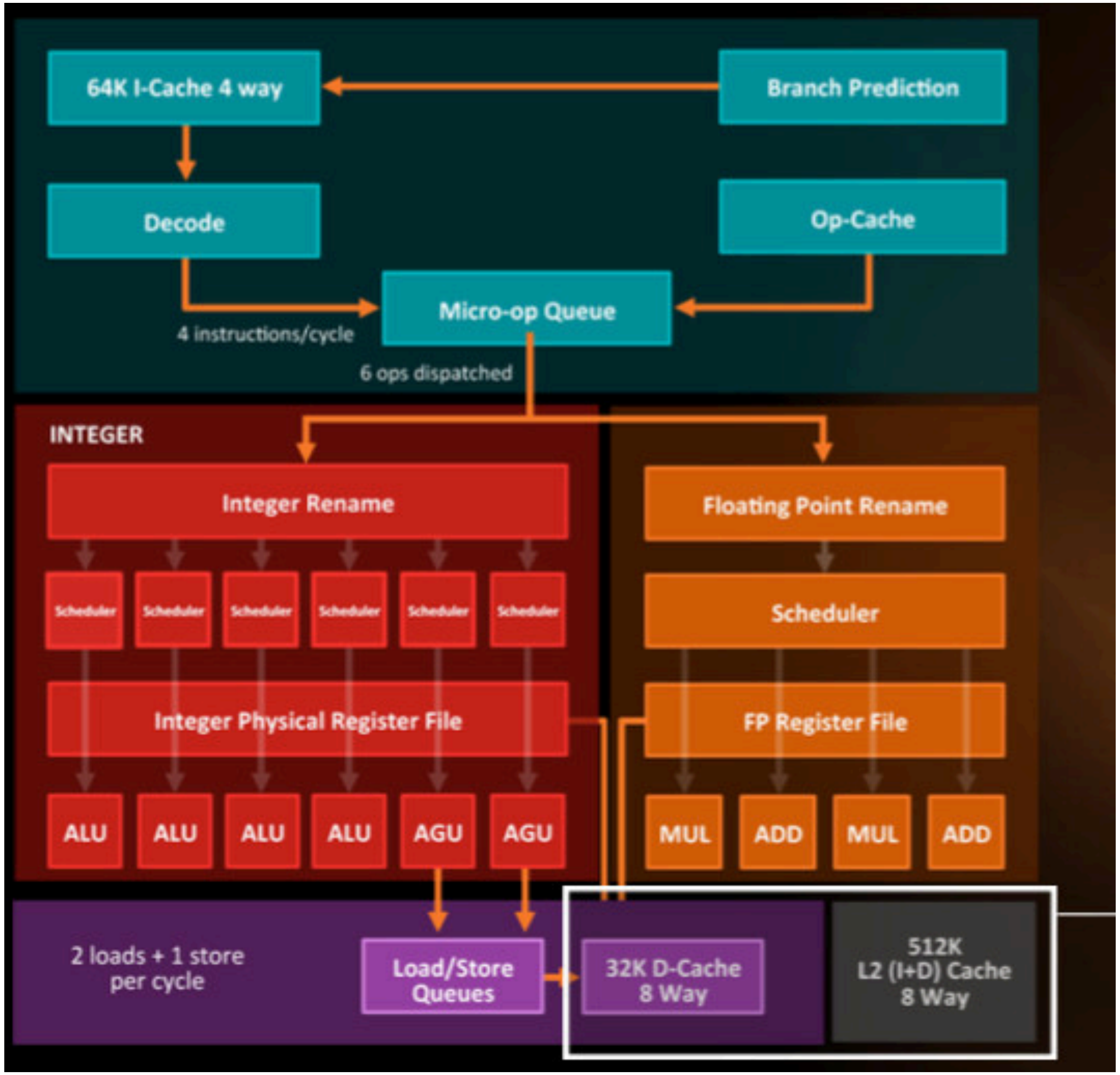
Modular Design

How to Build Future Systems?

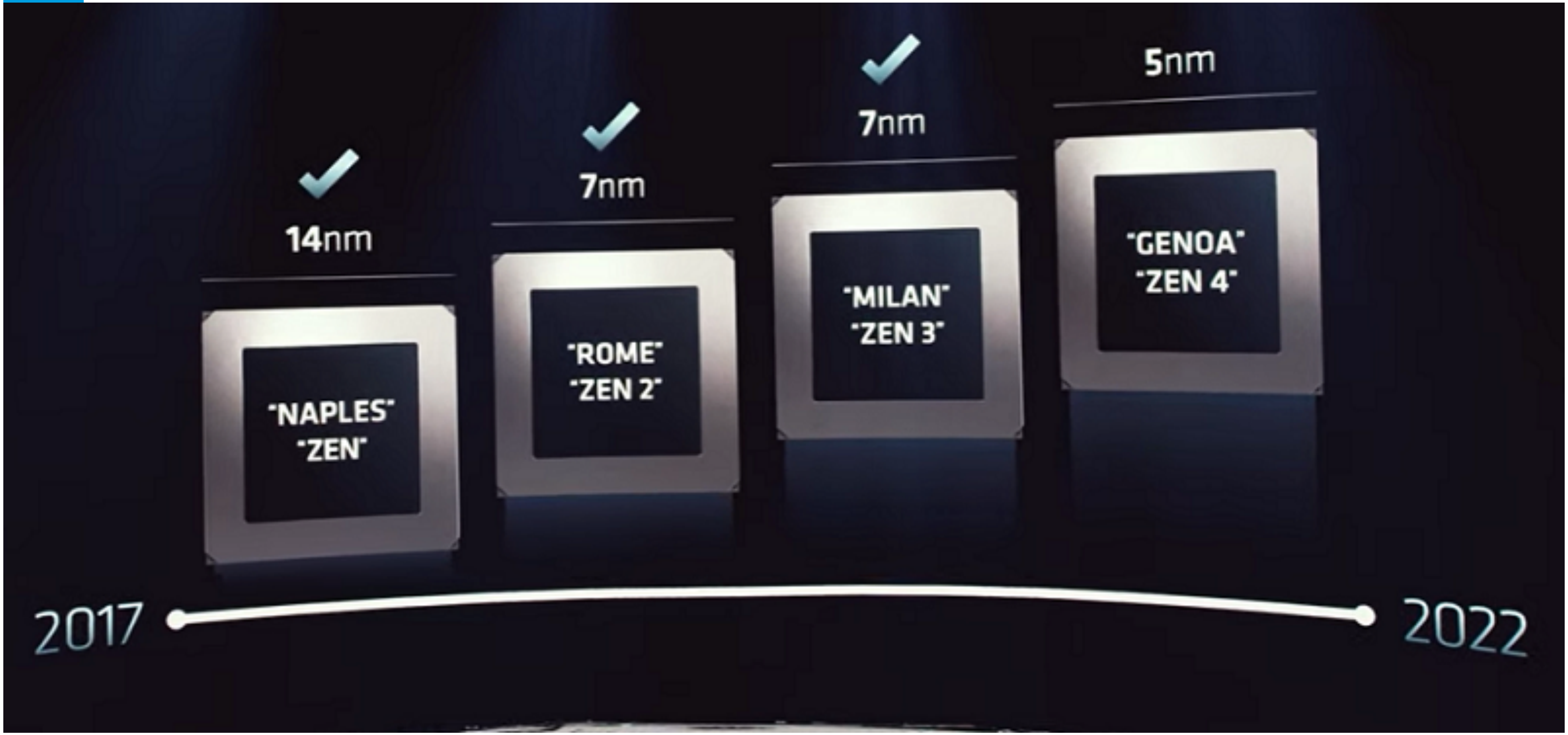
- Modular design supported by advanced packaging required

AMD

AMD ZEN 2017


64K I-Cache 4 way
Branch Prediction
Decode
Op-Cache
Micro-op Queue
4 instructions/cycle
6 ops dispatched
INTEGER
Integer Rename
Floating Point Rename
Scheduler
Scheduler
Scheduler
Scheduler
Scheduler
Scheduler
Scheduler
Integer Physical Register File
FP Register File
ALU
ALU
ALU
ALU
AGU
AGU
MUL
ADD
MUL
ADD
2 loads + 1 store per cycle
Load/Store Queues
32K D-Cache 8 Way
512K L2 (I+D) Cache 8 Way

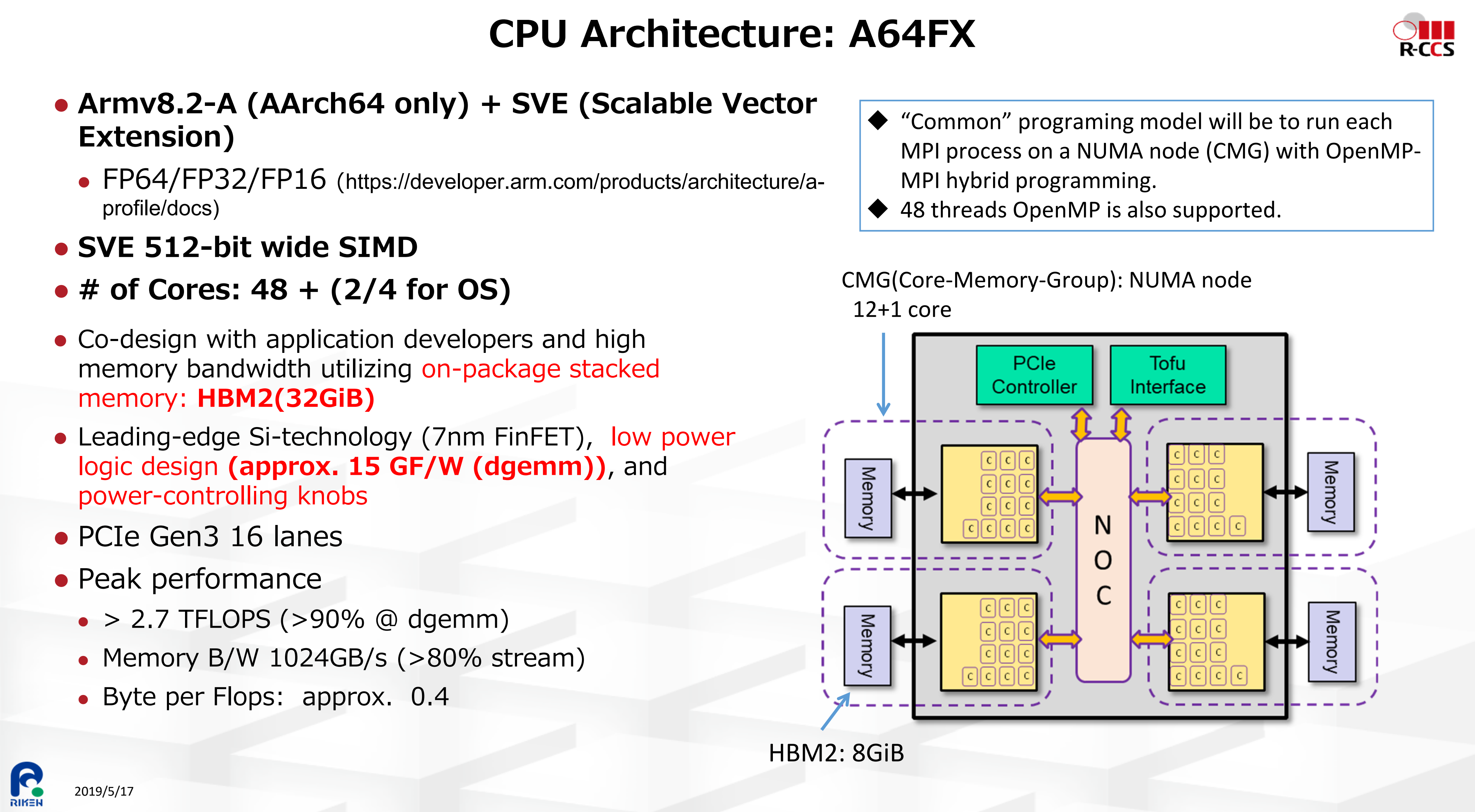
AMD: Rome (Zen2) and Milan (Zen3)



ARM: Advanced RISC Machines

- ARM only licenses its technology as intellectual property, rather than manufacturing its own CPUs.
- Companies making processors based on ARM's designs.
  Intel, Apple, Samsung,Texas Instruments, Analog Devices, Atmel, Freescale, Nvidia, Qualcomm, STMicroelectronics and Renesas have all licensed ARM technology.
- Design focussed on low power consumption and mainly used in handheld devices (also called phones).

ARM Fujitsu A64FX



CPU Architecture: A64FX

- **Armv8.2-A (AArch64 only) + SVE (Scalable Vector Extension)**
  - FP64/FP32/FP16 (https://developer.arm.com/products/architecture/a-profile/docs)
- **SVE 512-bit wide SIMD**
- **# of Cores: 48 + (2/4 for OS)**
- Co-design with application developers and high memory bandwidth utilizing on-package stacked memory: **HBM2(32GiB)**
- Leading-edge Si-technology (7nm FinFET), low power logic design **(approx. 15 GF/W (dgemm))**, and power-controlling knobs
- PCIe Gen3 16 lanes
- Peak performance
  - > 2.7 TFLOPS (>90% @ dgemm)
  - Memory B/W 1024GB/s (>80% stream)
  - Byte per Flops: approx. 0.4

◆ “Common” programing model will be to run each MPI process on a NUMA node (CMG) with OpenMP-MPI hybrid programming.

◆ 48 threads OpenMP is also supported.

CMG(Core-Memory-Group): NUMA node
12+1 core

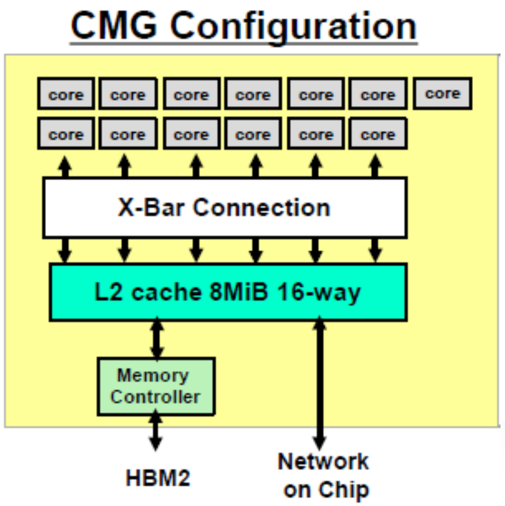
HBM2: 8GiB

2019/5/17

ARM Fujitsu A64FX

CMG (Core Memory Group)

- CMG: 13 cores (12+1) and L2 cache (8MiB 16way) and memory controller for HBM2 (8GiB)
- X-bar connection in a CMG maximize efficiency for throughput of L2 (>115 GB/s for R, >57 GB/s for W)
- Assistant core is dedicated to run OS demon, I/O, etc
- 4 CMGs support cache coherency by ccNUMA with on-chip directory (> 115GB/s x 2 for inter-CMGs)



CMG Configuration

core core core core core core core

core core core core core core

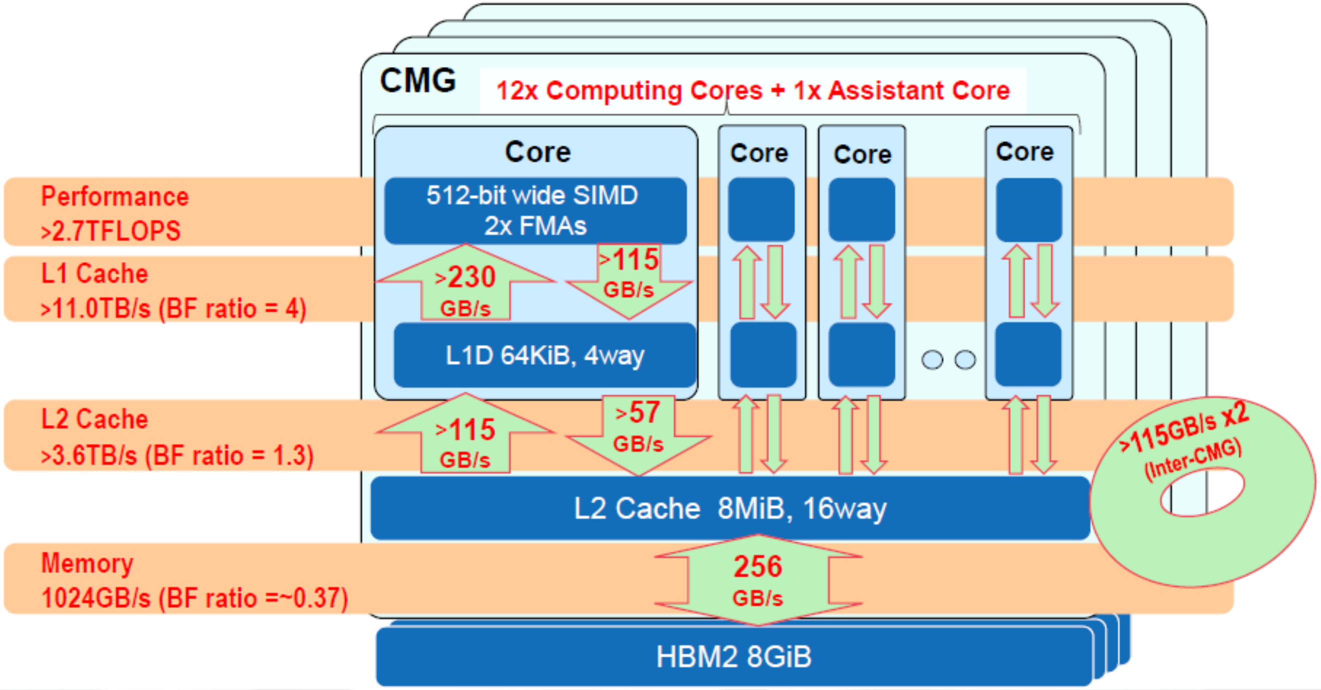
X-Bar Connection

L2 cache 8MiB 16-way

Memory Controller

HBM2

Network on Chip



CMG

12x Computing Cores + 1x Assistant Core

Core

512-bit wide SIMD 2x FMAs

>230 GB/s

>115 GB/s

L1D 64KiB, 4way

Core

Core

Core

Performance >2.7TFLOPS

L1 Cache >11.0TB/s (BF ratio = 4)

L2 Cache >3.6TB/s (BF ratio = 1.3)

>115 GB/s

>57 GB/s

L2 Cache 8MiB, 16way

>115GB/s x2 (Inter-CMG)

Memory 1024GB/s (BF ratio =~0.37)

256 GB/s

HBM2 8GiB

Figures from the slide presented in Hotchips 30 by Fujitsu

ShenWei SW26010 Microprocessor

- 4x (64 CPE + 1 MPE) cores
- 64-bit RISC
- 1.45 GHz
- 256 bit vectors
- each core 8 flops/cycle => 3.06 Tflop/s
- no-cache
- 128-bit system bus
- DDR3 2133, 4 channels, max 32 GB
- 6 Gflop/Watt => very energy efficient (most Xeon's ~2 Gflop/Watt)

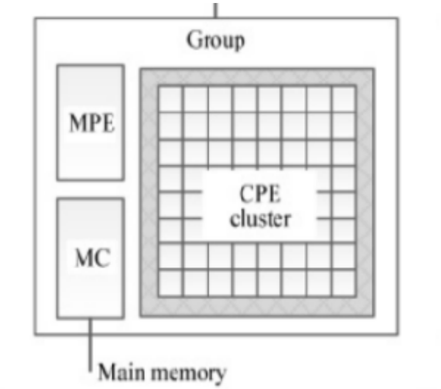


Figure 1: Core Group for Node

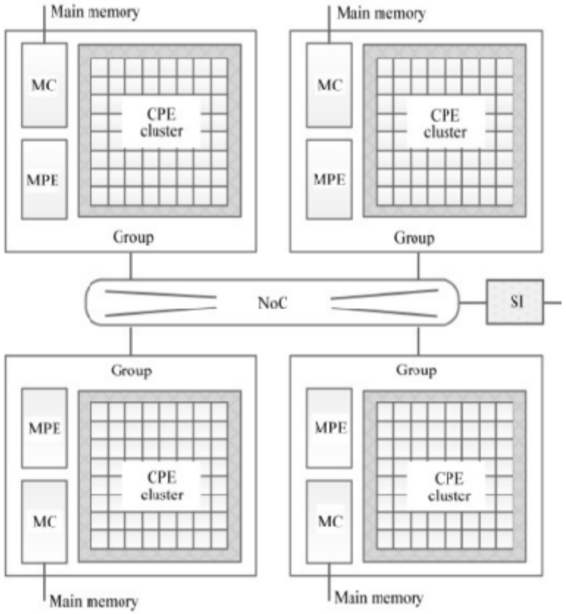
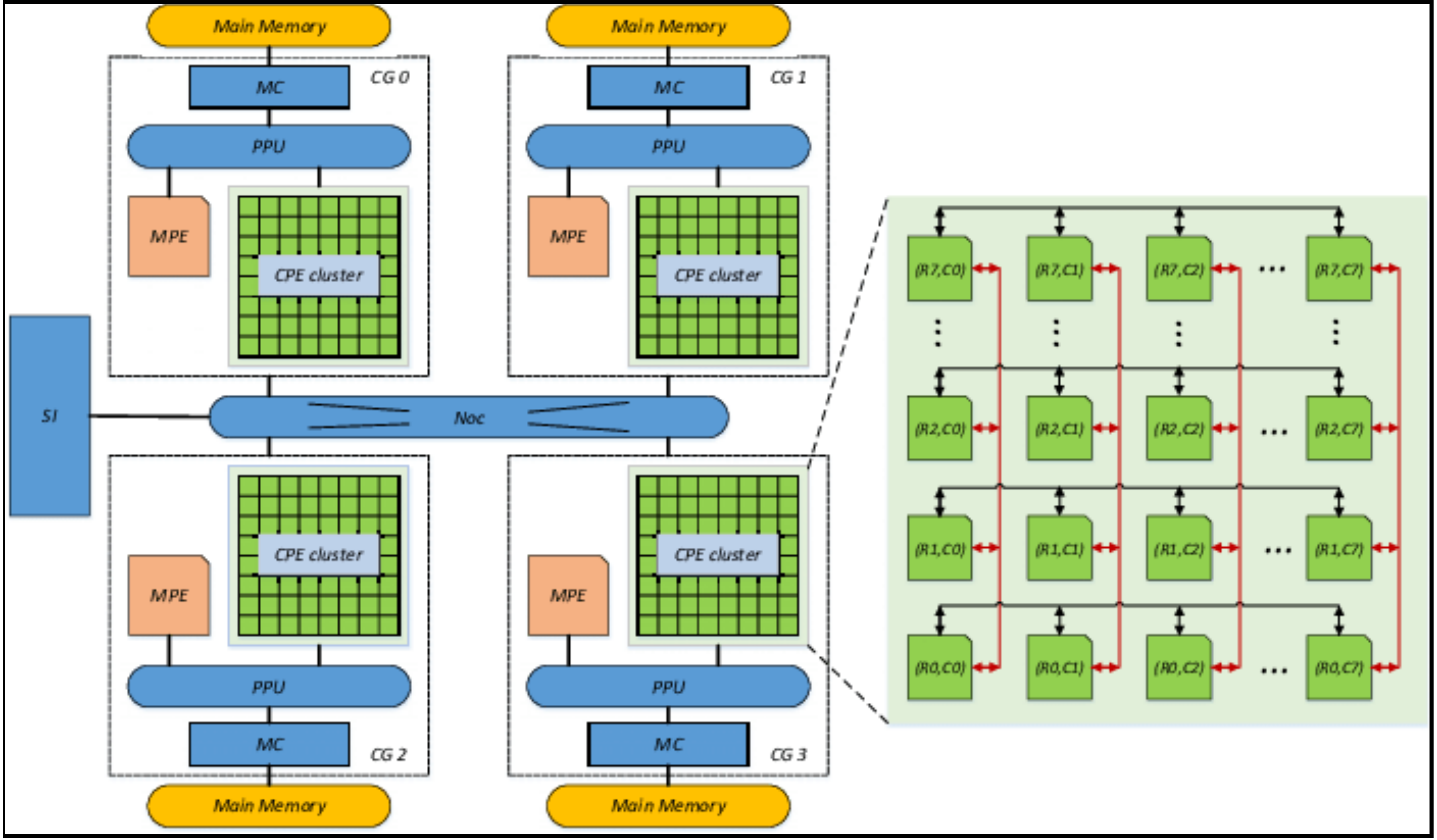


Figure 2: Basic Layout of a Node


Main Memory
Main Memory
MC
CG 0
MC
CG 1
PPU
PPU
MPE
CPE cluster
MPE
CPE cluster
SI
Noc
MPE
CPE cluster
MPE
CPE cluster
PPU
PPU
MC
CG 2
MC
CG 3
Main Memory
Main Memory
(R7,C0)
(R7,C1)
(R7,C2)
(R7,C7)
(R2,C0)
(R2,C1)
(R2,C2)
(R2,C7)
(R1,C0)
(R1,C1)
(R1,C2)
(R1,C7)
(R0,C0)
(R0,C1)
(R0,C2)
(R0,C7)

Observation: More Processors per socket

- Intel IceLake: 2021, 36 cores
- AMD Rome: 2020, 64 cores, 8 modules
- IBM Power 10: 2021, 15 cores SMT=8
- Arm: 16+ cores

Alternatives to multi-core CPU

- GP-GPU: graphical cards
- Intel Ponte Vecchio ~GPU accelerator
- FPGA
- Special computational boards

GP-GPU

- Use graphical processors for computational work
  - enormous market (games) creates cheap products
  - flops are cheap: communication is expensive

- Nvidia
- first generation (G80)
  - just a graphical card which also runs some codes
- second generation (tesla, G200)
  - graphical card with floating point runs more codes
- third generation (fermi)
  - HPC card which can also do graphics very well
- fourth generation (kepler)
  - HPC card with cache

AMD story

- Movie:

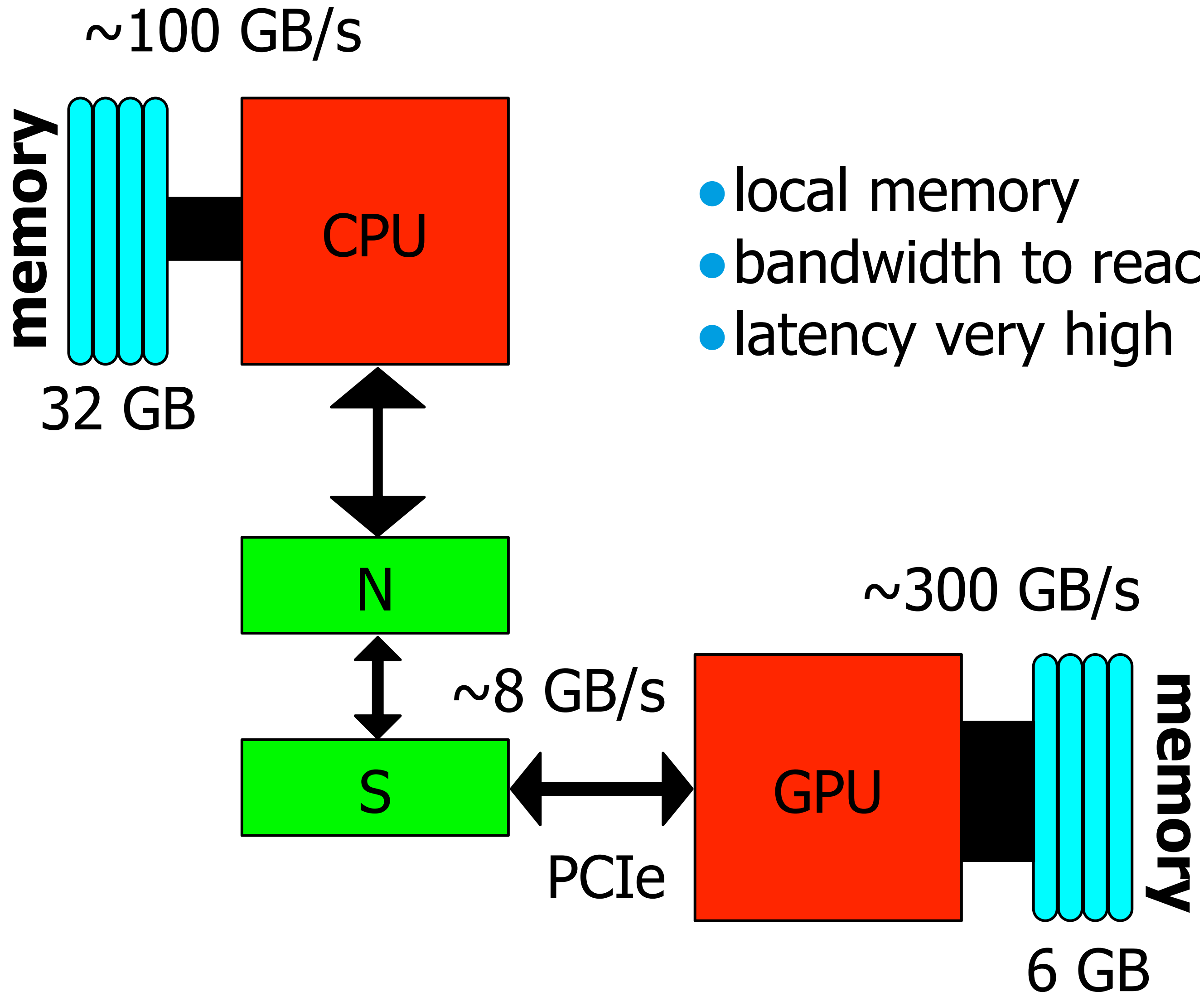
  AMD Building Blocks- A Look Inside Your Personal Computer

http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2012/10/HSA_TIRIAS_Whitepaper_Final_1-28-14.pdf

GPU vs CPU

GPU	**CPU**
simple architecture	complex architecture
many cores > 1000	a few cores 12-64
energy efficient flops per core	high energy flops per core
specialized computing	general computing
GDDR	DDR
in-order	out-of-order, superscalar
	branch prediction, SMT

CPU connected to GPU



- local memory
- bandwidth to reach GPU fine
- latency very high

GP-GPU

- GPU usage
  - Algorithms and applications using the Fast Fourier Transform
  - Audio processing and DSP
  - Digital image and video processing
  - Raytracing
  - Weather forecasting
  - Neural networks
  - Molecular modeling
  - Database operations
  - Reverse Time Migration (Finite Difference)

A nice introduction can be found at:
http://en.wikipedia.org/wiki/GPGPU]

Nvidia

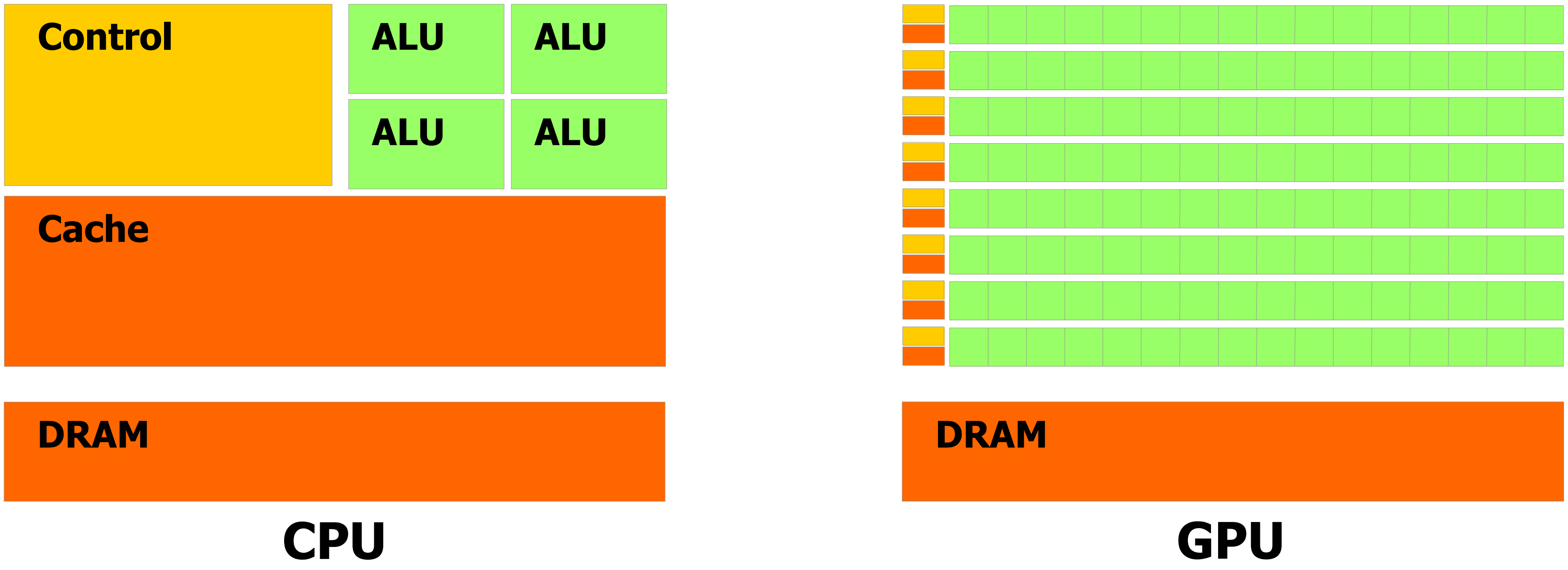
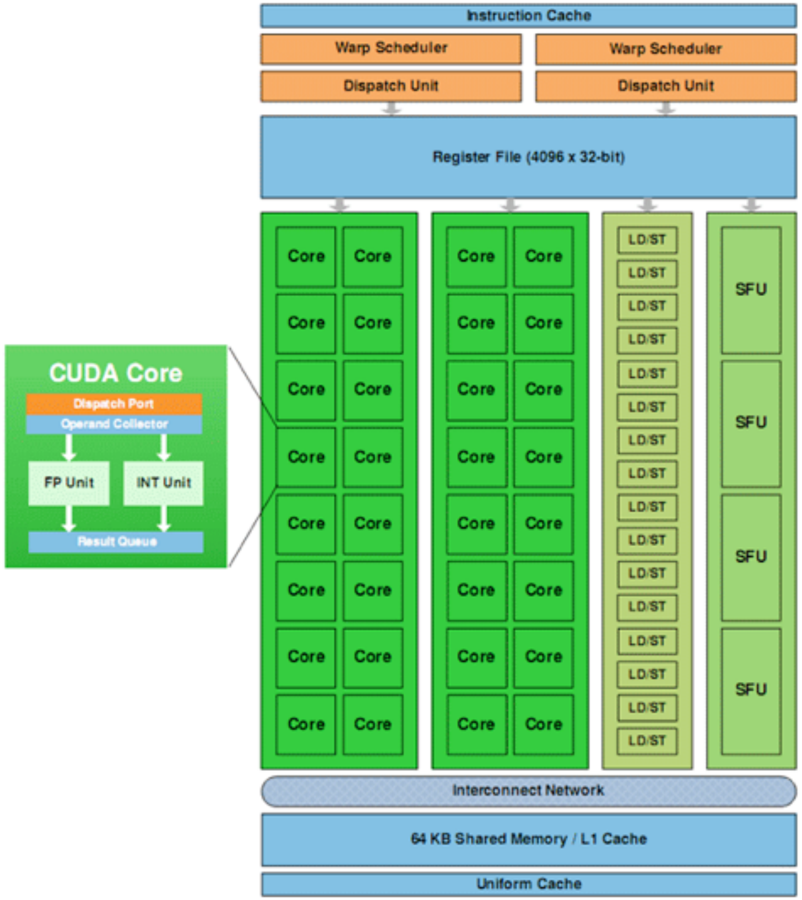
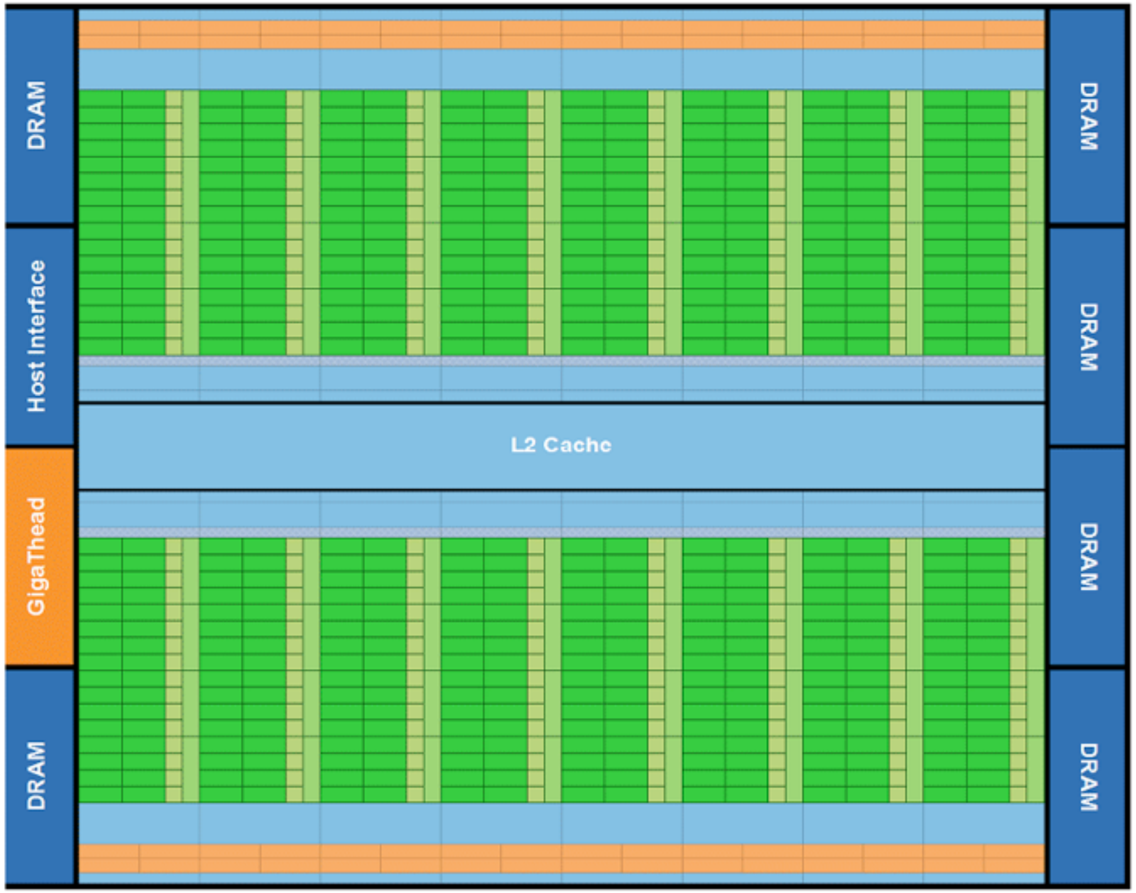


Figure 1-2. The GPU Devotes More Transistors to Data Processing

http://www.nvidia.com/object/cuda_develop.html

Fermi (2010)


Instruction Cache
Warp Scheduler
Warp Scheduler
Dispatch Unit
Dispatch Unit
Register File (4096 x 32-bit)
Core
LD/ST
SFU
CUDA Core
Dispatch Port
Operand Collector
FP Unit
INT Unit
Result Queue
Interconnect Network
64 KB Shared Memory / L1 Cache
Uniform Cache


DRAM
Host Interface
GigaThread
DRAM
L2 Cache
DRAM
DRAM
DRAM
DRAM

NVLink

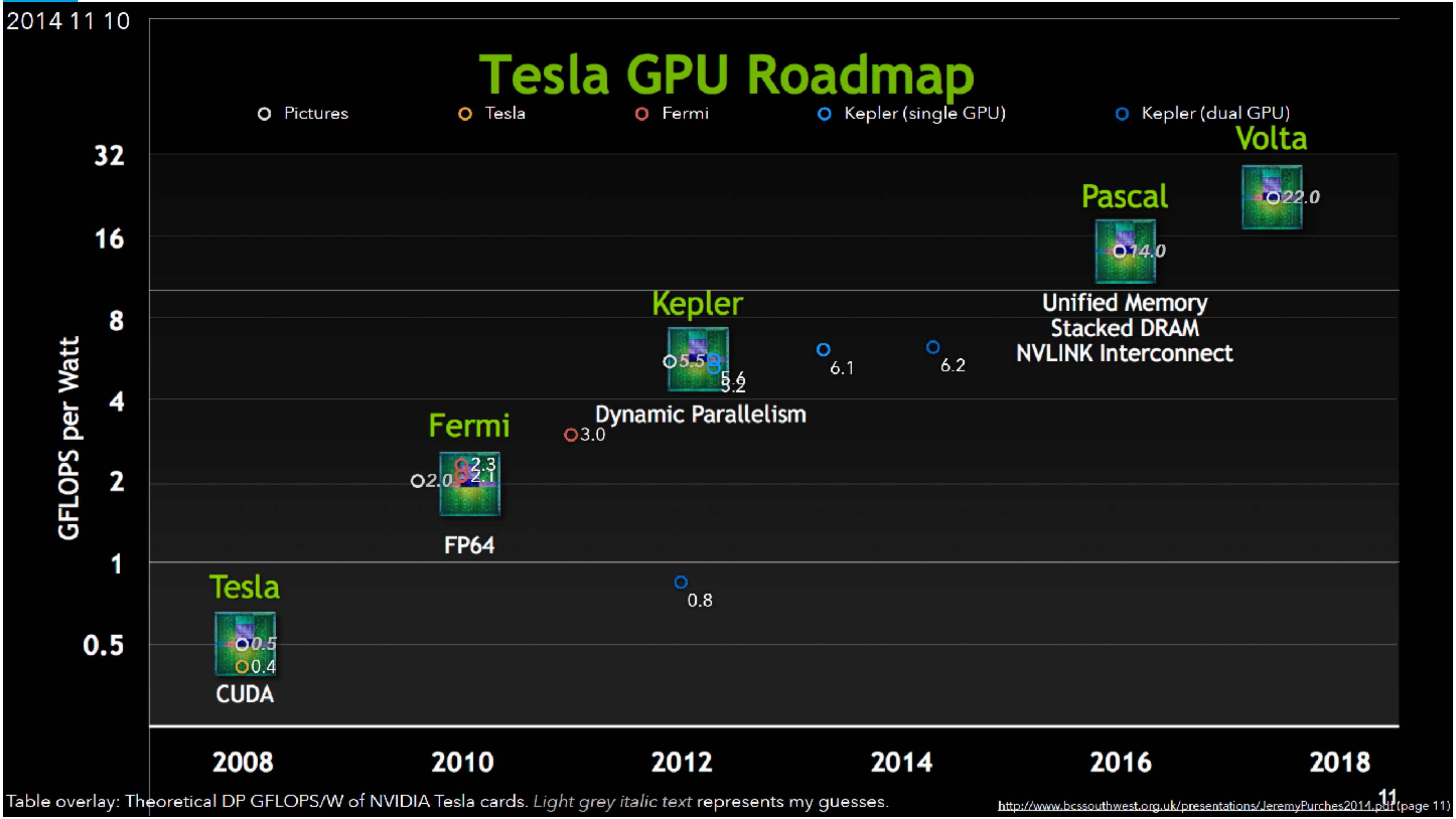
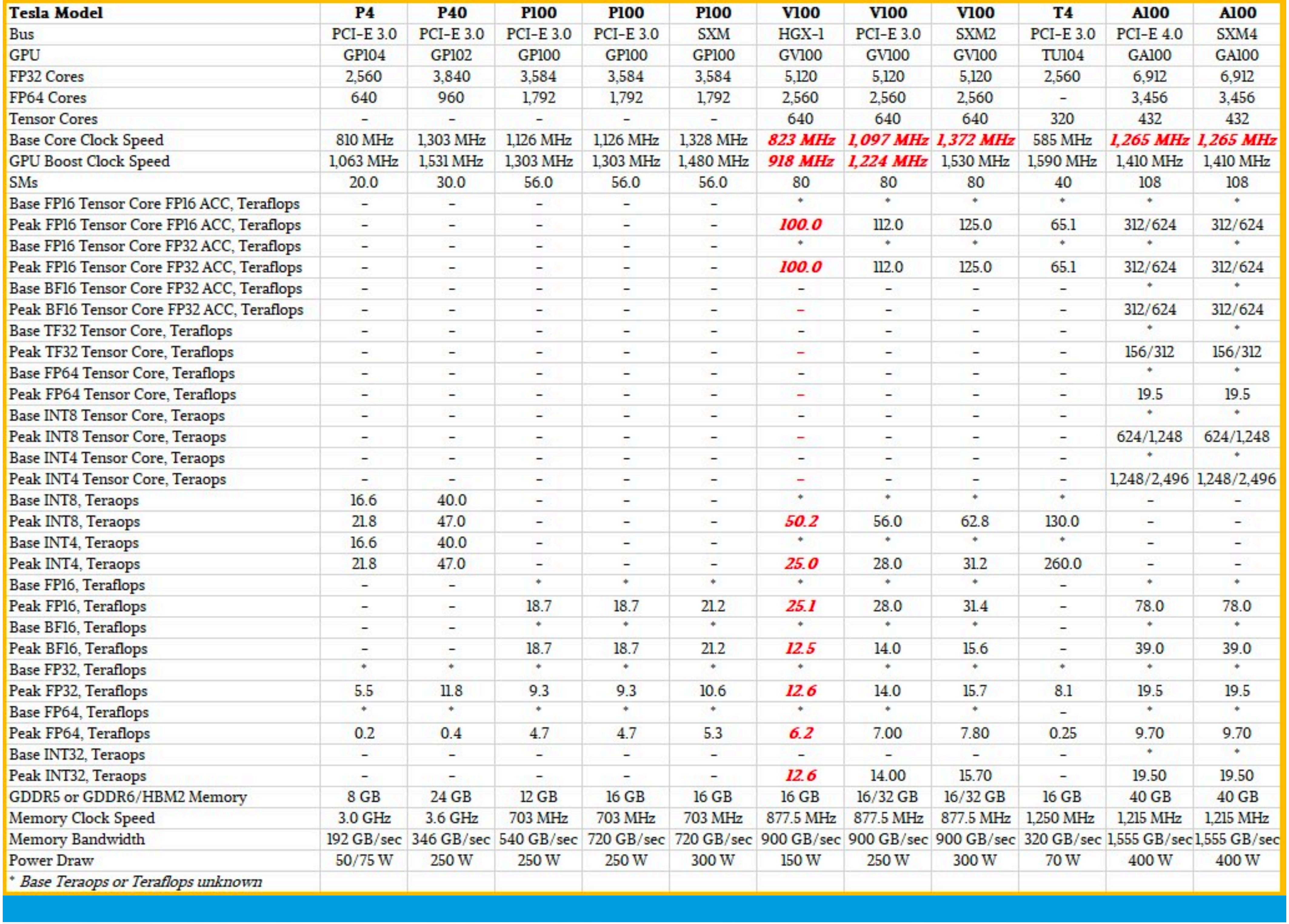


Table overlay: Theoretical DP GFLOPS/W of NVIDIA Tesla cards. *Light grey italic text* represents my guesses.

11 http://www.bcssouthwest.org.uk/presentations/JeremyPurches2014.pdf (page 11)



Tesla Model	P4	P40	P100	P100	P100	V100	V100	V100	T4	A100	A100
Bus	PCI-E 3.0	PCI-E 3.0	PCI-E 3.0	PCI-E 3.0	SXM	HGX-1	PCI-E 3.0	SXM2	PCI-E 3.0	PCI-E 4.0	SXM4
GPU	GP104	GP102	GP100	GP100	GP100	GV100	GV100	GV100	TU104	GA100	GA100
FP32 Cores	2,560	3,840	3,584	3,584	3,584	5,120	5,120	5,120	2,560	6,912	6,912
FP64 Cores	640	960	1,792	1,792	1,792	2,560	2,560	2,560	-	3,456	3,456
Tensor Cores	-	-	-	-	-	640	640	640	320	432	432
Base Core Clock Speed	810 MHz	1,303 MHz	1,126 MHz	1,126 MHz	1,328 MHz	***823 MHz***	***1,097 MHz***	***1,372 MHz***	585 MHz	***1,265 MHz***	***1,265 MHz***
GPU Boost Clock Speed	1,063 MHz	1,531 MHz	1,303 MHz	1,303 MHz	1,480 MHz	***918 MHz***	***1,224 MHz***	1,530 MHz	1,590 MHz	1,410 MHz	1,410 MHz
SMs	20.0	30.0	56.0	56.0	56.0	80	80	80	40	108	108
Base FP16 Tensor Core FP16 ACC, Teraflops	-	-	-	-	-	*	*	*	*	*	*
Peak FP16 Tensor Core FP16 ACC, Teraflops	-	-	-	-	-	***100.0***	112.0	125.0	65.1	312/624	312/624
Base FP16 Tensor Core FP32 ACC, Teraflops	-	-	-	-	-	*	*	*	*	*	*
Peak FP16 Tensor Core FP32 ACC, Teraflops	-	-	-	-	-	***100.0***	112.0	125.0	65.1	312/624	312/624
Base BF16 Tensor Core FP32 ACC, Teraflops	-	-	-	-	-	-	-	-	-	*	*
Peak BF16 Tensor Core FP32 ACC, Teraflops	-	-	-	-	-	-	-	-	-	312/624	312/624
Base TF32 Tensor Core, Teraflops	-	-	-	-	-	-	-	-	-	*	*
Peak TF32 Tensor Core, Teraflops	-	-	-	-	-	-	-	-	-	156/312	156/312
Base FP64 Tensor Core, Teraflops	-	-	-	-	-	-	-	-	-	*	*
Peak FP64 Tensor Core, Teraflops	-	-	-	-	-	-	-	-	-	19.5	19.5
Base INT8 Tensor Core, Teraops	-	-	-	-	-	-	-	-	-	*	*
Peak INT8 Tensor Core, Teraops	-	-	-	-	-	-	-	-	-	624/1,248	624/1,248
Base INT4 Tensor Core, Teraops	-	-	-	-	-	-	-	-	-	*	*
Peak INT4 Tensor Core, Teraops	-	-	-	-	-	-	-	-	-	1,248/2,496	1,248/2,496
Base INT8, Teraops	16.6	40.0	-	-	-	*	*	*	*	-	-
Peak INT8, Teraops	21.8	47.0	-	-	-	***50.2***	56.0	62.8	130.0	-	-
Base INT4, Teraops	16.6	40.0	-	-	-	*	*	*	*	-	-
Peak INT4, Teraops	21.8	47.0	-	-	-	***25.0***	28.0	31.2	260.0	-	-
Base FP16, Teraflops	-	-	*	*	*	*	*	*	-	*	*
Peak FP16, Teraflops	-	-	18.7	18.7	21.2	***25.1***	28.0	31.4	-	78.0	78.0
Base BF16, Teraflops	-	-	*	*	*	*	*	*	-	*	*
Peak BF16, Teraflops	-	-	18.7	18.7	21.2	***12.5***	14.0	15.6	-	39.0	39.0
Base FP32, Teraflops	*	*	*	*	*	*	*	*	*	*	*
Peak FP32, Teraflops	5.5	11.8	9.3	9.3	10.6	***12.6***	14.0	15.7	8.1	19.5	19.5
Base FP64, Teraflops	*	*	*	*	*	*	*	*	-	*	*
Peak FP64, Teraflops	0.2	0.4	4.7	4.7	5.3	***6.2***	7.00	7.80	0.25	9.70	9.70
Base INT32, Teraops	-	-	-	-	-	-	-	-	-	*	*
Peak INT32, Teraops	-	-	-	-	-	***12.6***	14.00	15.70	-	19.50	19.50
GDDR5 or GDDR6/HBM2 Memory	8 GB	24 GB	12 GB	16 GB	16 GB	16 GB	16/32 GB	16/32 GB	16 GB	40 GB	40 GB
Memory Clock Speed	3.0 GHz	3.6 GHz	703 MHz	703 MHz	703 MHz	877.5 MHz	877.5 MHz	877.5 MHz	1,250 MHz	1,215 MHz	1,215 MHz
Memory Bandwidth	192 GB/sec	346 GB/sec	540 GB/sec	720 GB/sec	720 GB/sec	900 GB/sec	900 GB/sec	900 GB/sec	320 GB/sec	1,555 GB/sec	1,555 GB/sec
Power Draw	50/75 W	250 W	250 W	250 W	300 W	150 W	250 W	300 W	70 W	400 W	400 W
** Base Teraops or Teraflops unknown*											

Nvidia Ampere (2021)

- Threads: xxx
  - Streaming Multiprocessor (SM) has 64 FP32 units
  - There are 128 SM's

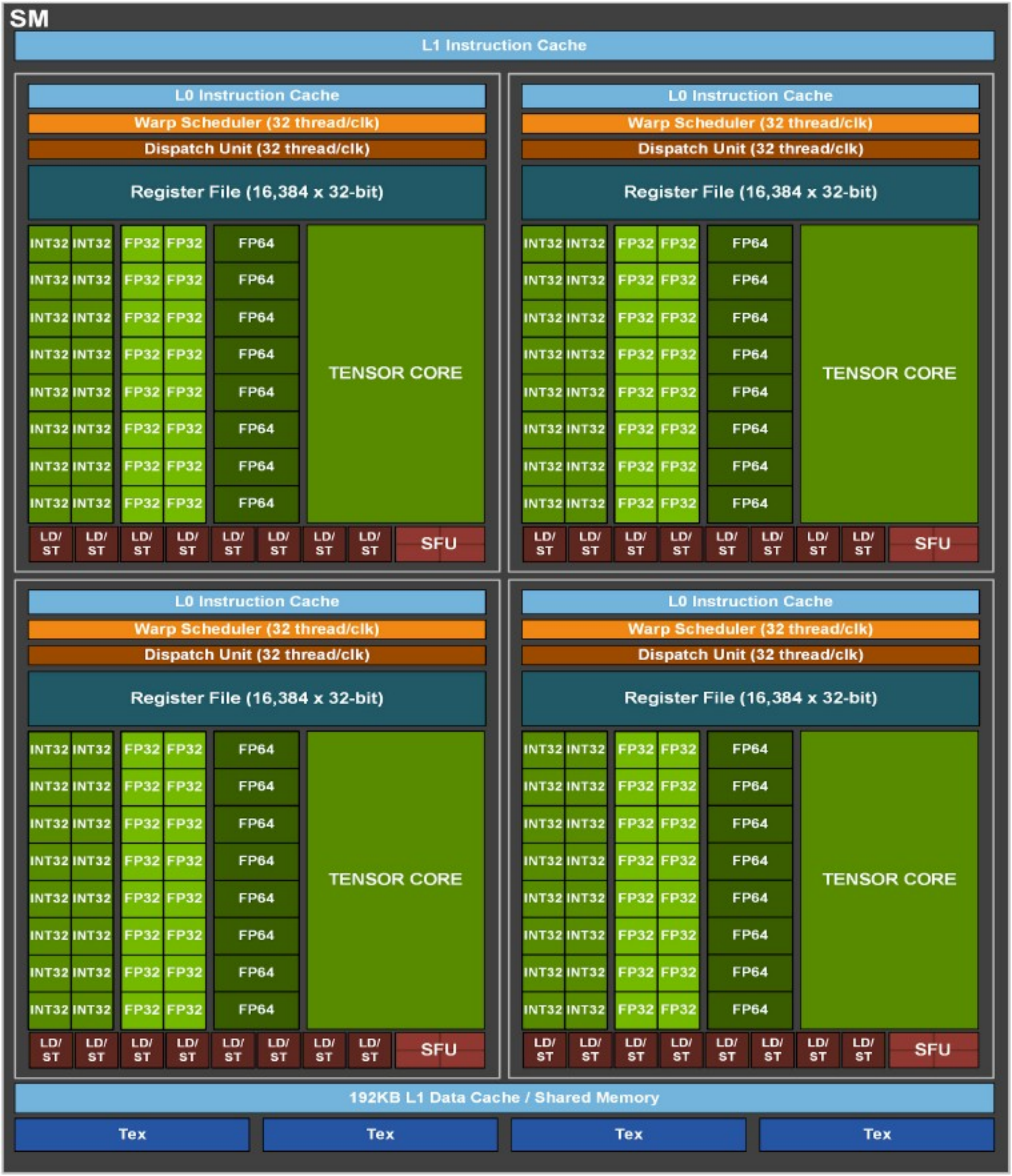
- Mixed floating point format
  - 64, 32 and 16-bit FP, Tensor cores …

- Memory 1000 GB/s of

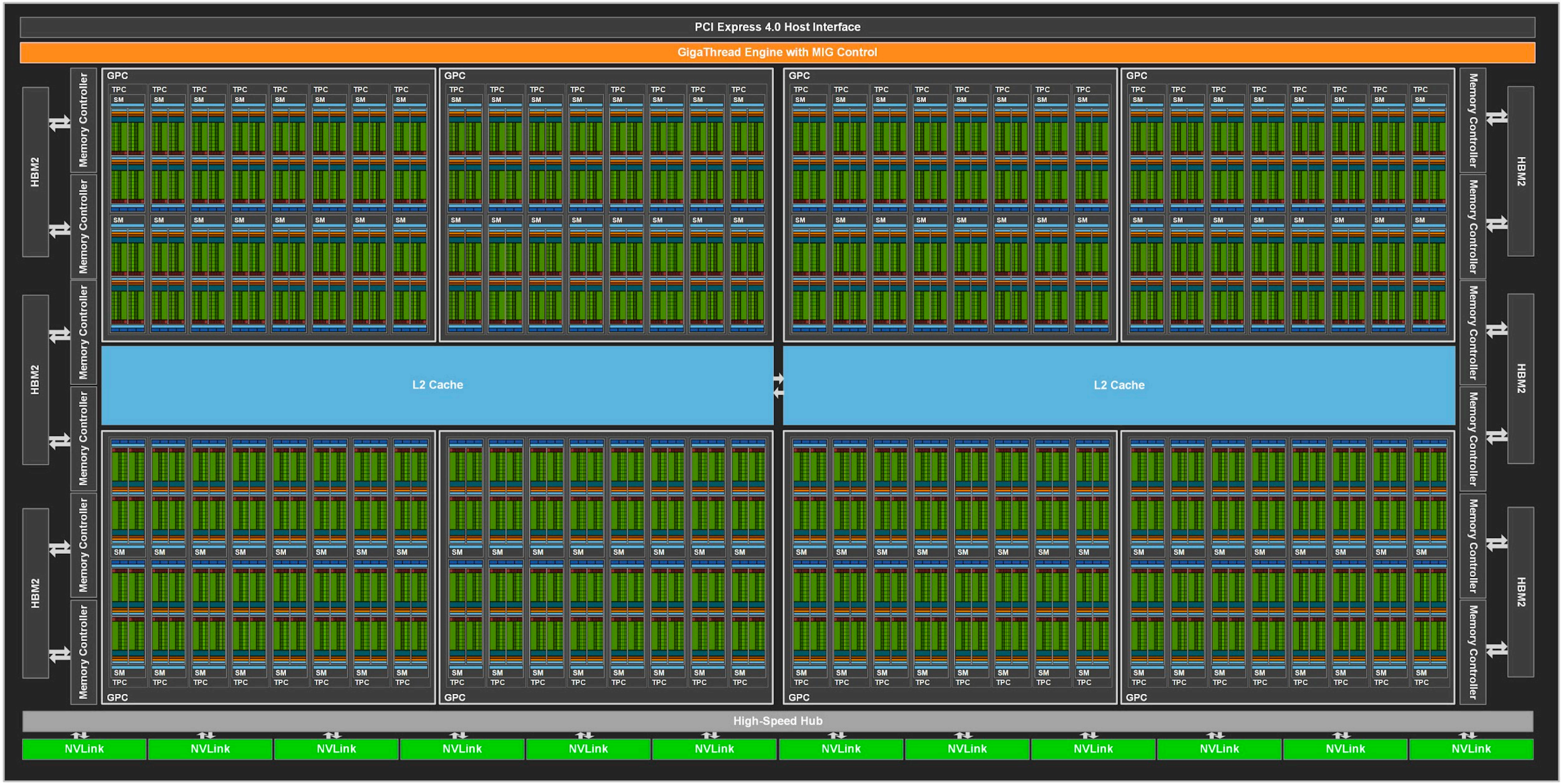
- 16 / 32 GB HBM2 (High bandwidth Memory)

- NVLink: to directly access memory of another GPU or CPU

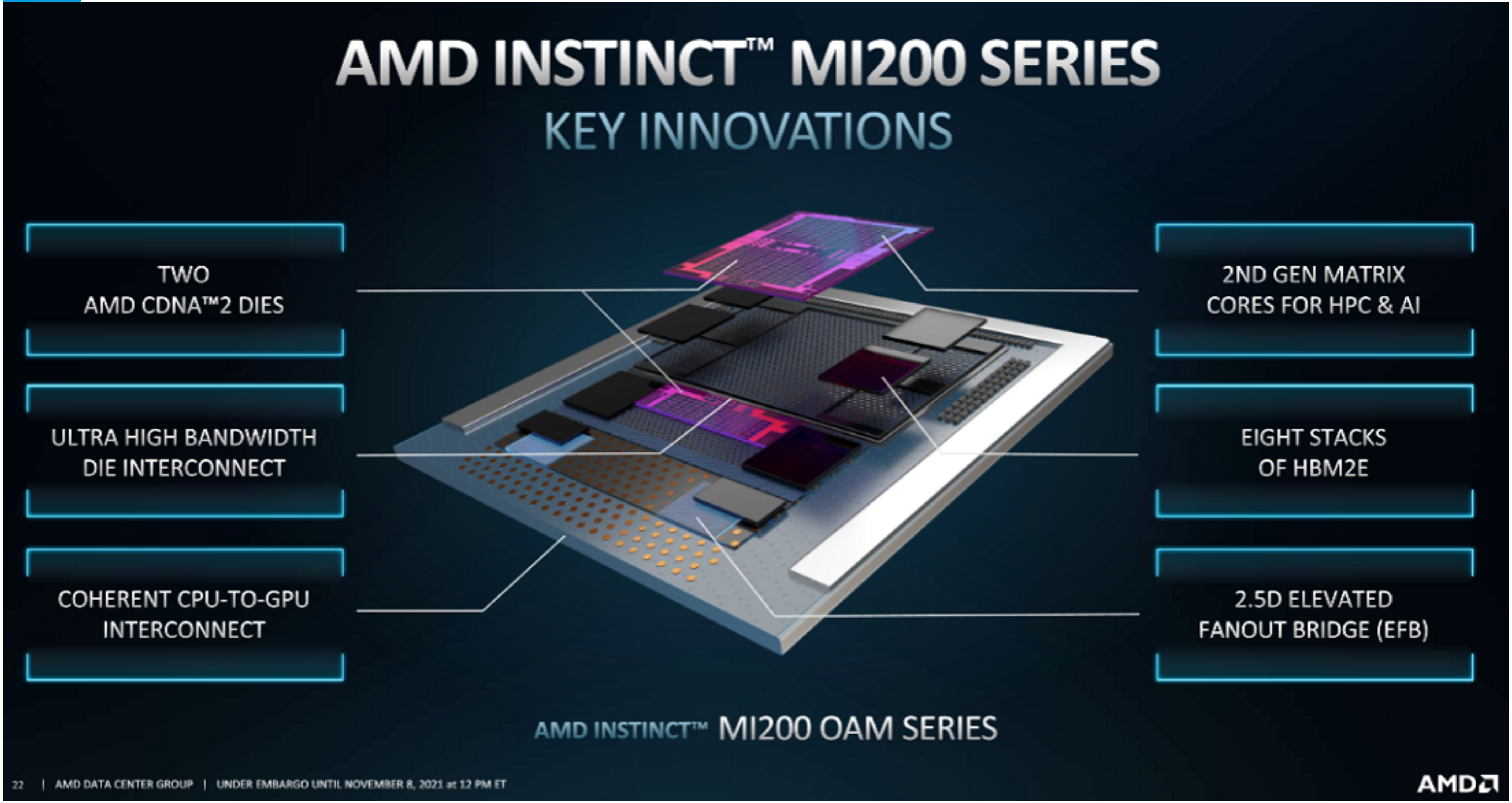
- 7 nm FinFET


SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
INT32
FP32
FP64
TENSOR CORE
LD/ ST
SFU
192KB L1 Data Cache / Shared Memory
Tex
Tex
Tex
Tex

Ampere’s all 128 SM’s



AMD GPU's: MI200



AMD INSTINCT™ MI200 SERIES

KEY INNOVATIONS

- TWO AMD CDNA™2 DIES
- ULTRA HIGH BANDWIDTH DIE INTERCONNECT
- COHERENT CPU-TO-GPU INTERCONNECT
- 2ND GEN MATRIX CORES FOR HPC & AI
- EIGHT STACKS OF HBM2E
- 2.5D ELEVATED FANOUT BRIDGE (EFB)

AMD INSTINCT™ MI200 OAM SERIES

22 | AMD DATA CENTER GROUP | UNDER EMBARGO UNTIL NOVEMBER 8, 2021 at 12 PM ET

AMD

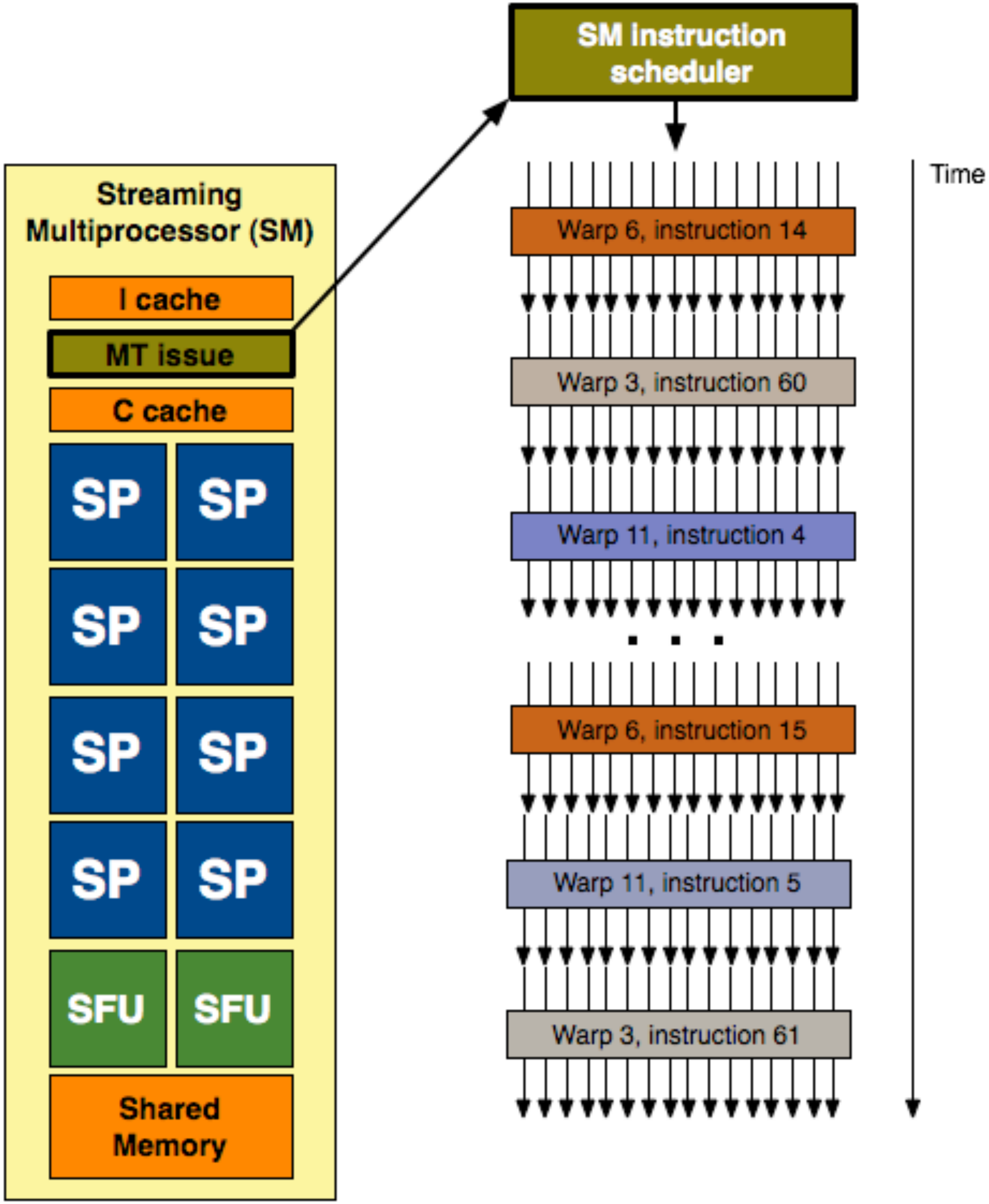
How does a GPU hide memory latency

- GPU’s issue instructions in order
  - Issue stalls when instruction arguments are not ready

- GPUs switch between threads to hide latency
  - context switch is free: thread state is partitioned (large register file), not stored / restored

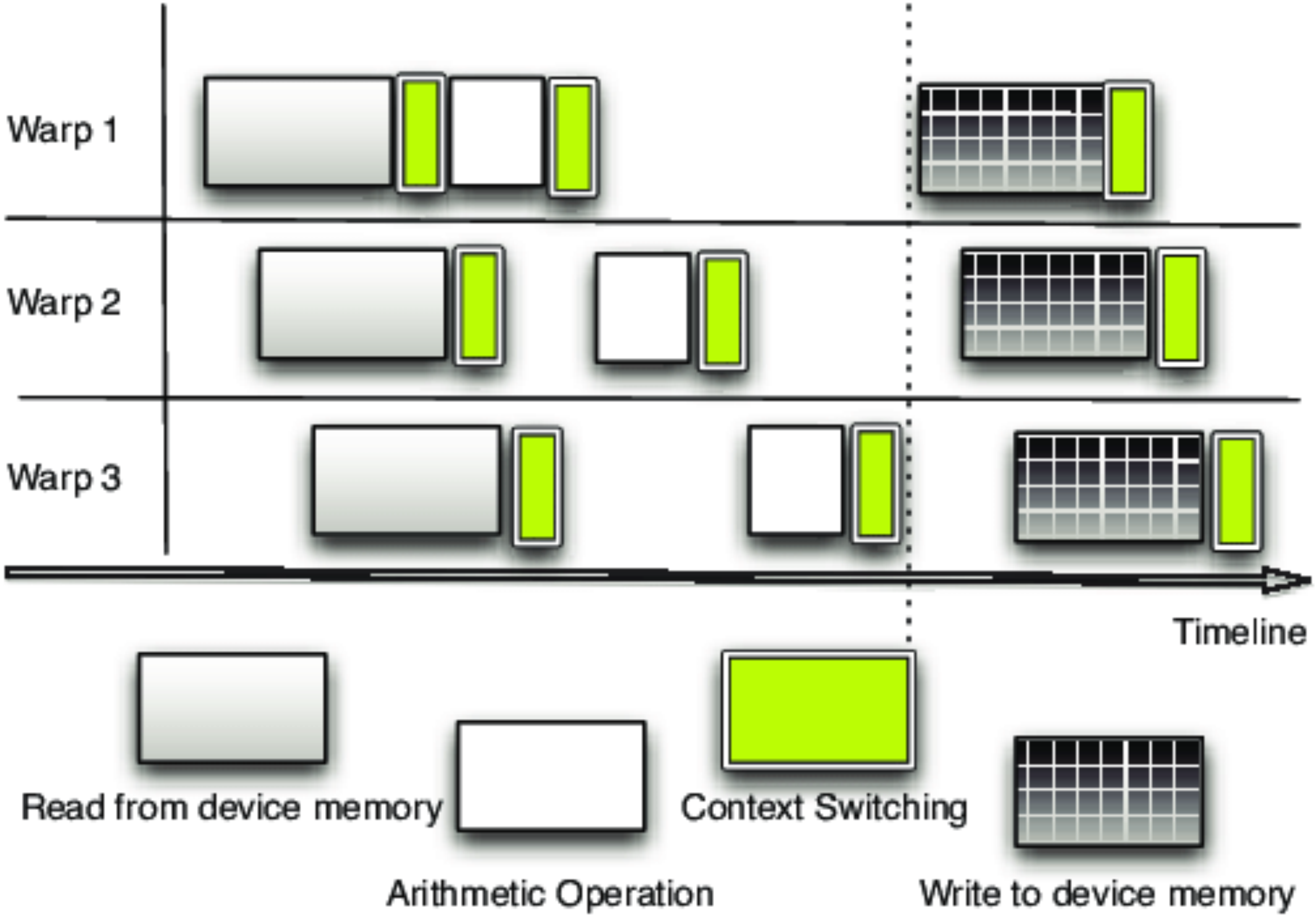
- Need enough threads to hide latency and saturate the memory bus.

Text



SM instruction scheduler
Streaming Multiprocessor (SM)
I cache
MT issue
C cache
SP
SP
SP
SP
SP
SP
SP
SP
SFU
SFU
Shared Memory
Time
Warp 6, instruction 14
Warp 3, instruction 60
Warp 11, instruction 4
. . .
Warp 6, instruction 15
Warp 11, instruction 5
Warp 3, instruction 61

T


Warp 1
Warp 2
Warp 3
Timeline
Read from device memory
Arithmetic Operation
Context Switching
Write to device memory

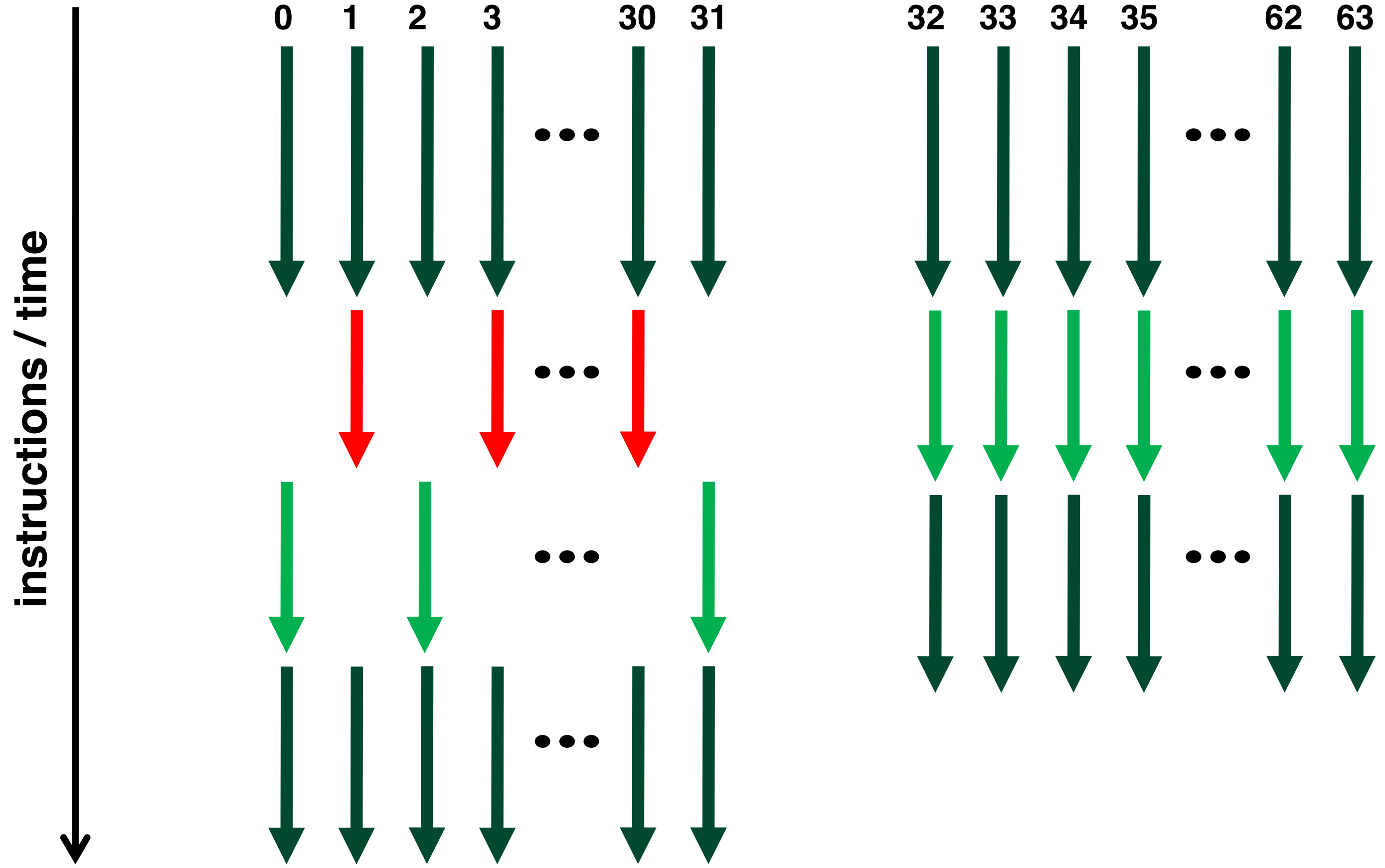
GPU instructions

- Single-Instruction Multiple-Threads (SIMT) model

- A single instruction is issued for a warp (thread-vector) at a time
    - NVIDIA GPU: warp = a vector of 32 threads
    - AMD GPU: wavefront = a vector ot 64 threads

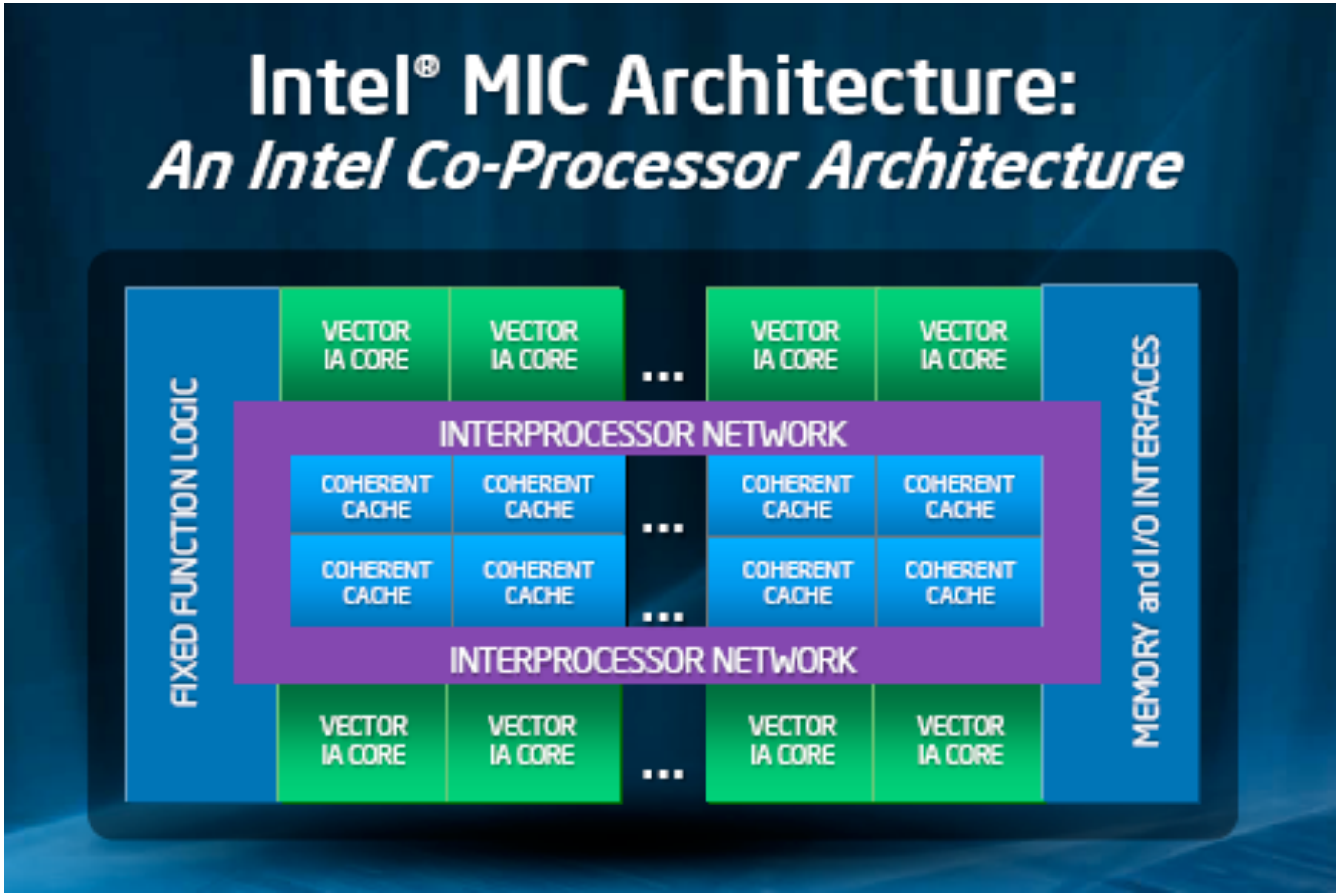
  - warp = group of 32 threads that always execute same instructions simultaneously.

Execution diverges within a warp



© NVIDIA 2011

Intel’s first answer to GPU’s: MIC

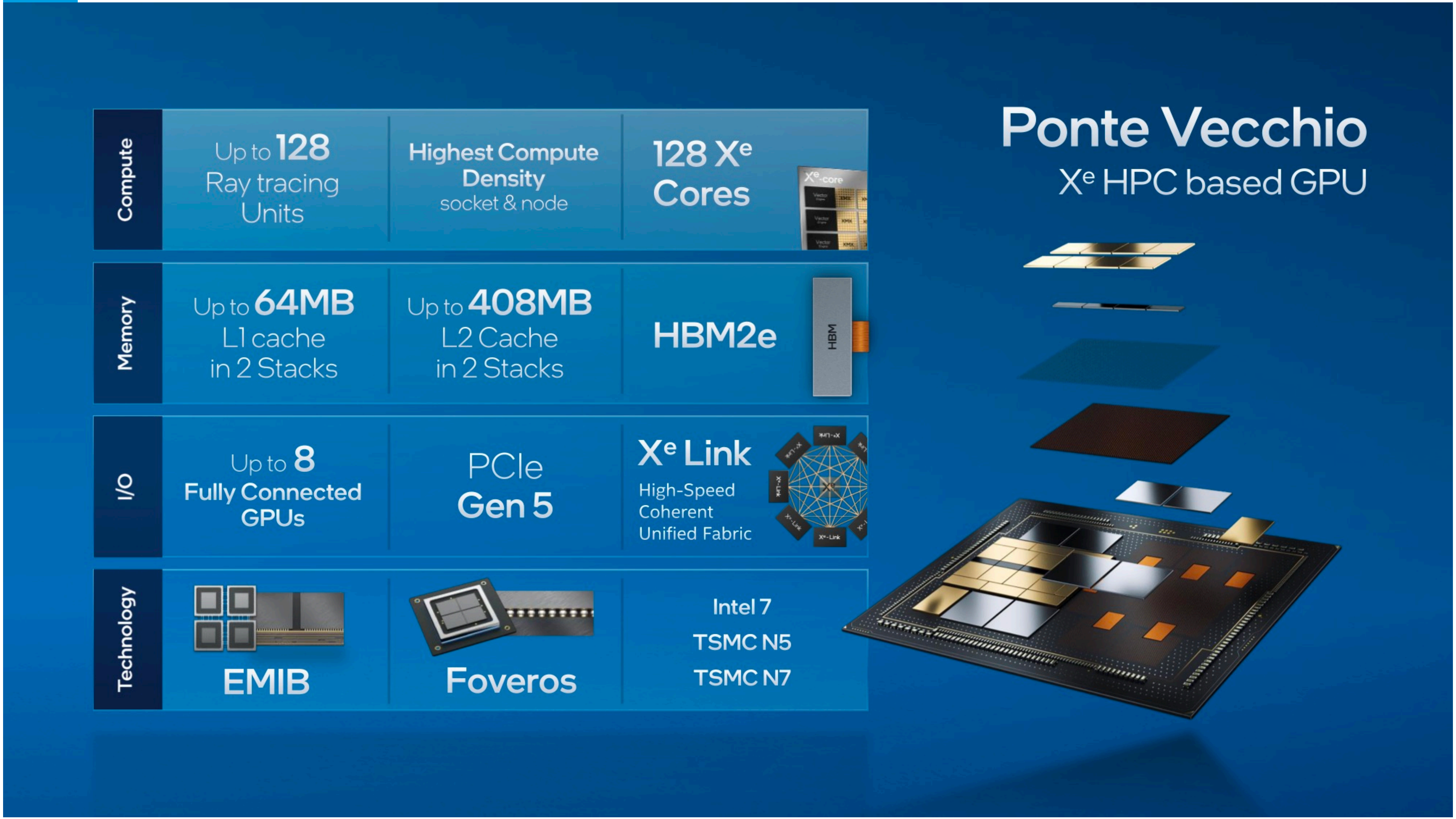


16 (single) flops/cycle

Many Integrated Cores, X86 based

Knight Ferry: 32 cores at 1.2 GHz, linked via PCIe

Intel’s latest Ponte Vecchio

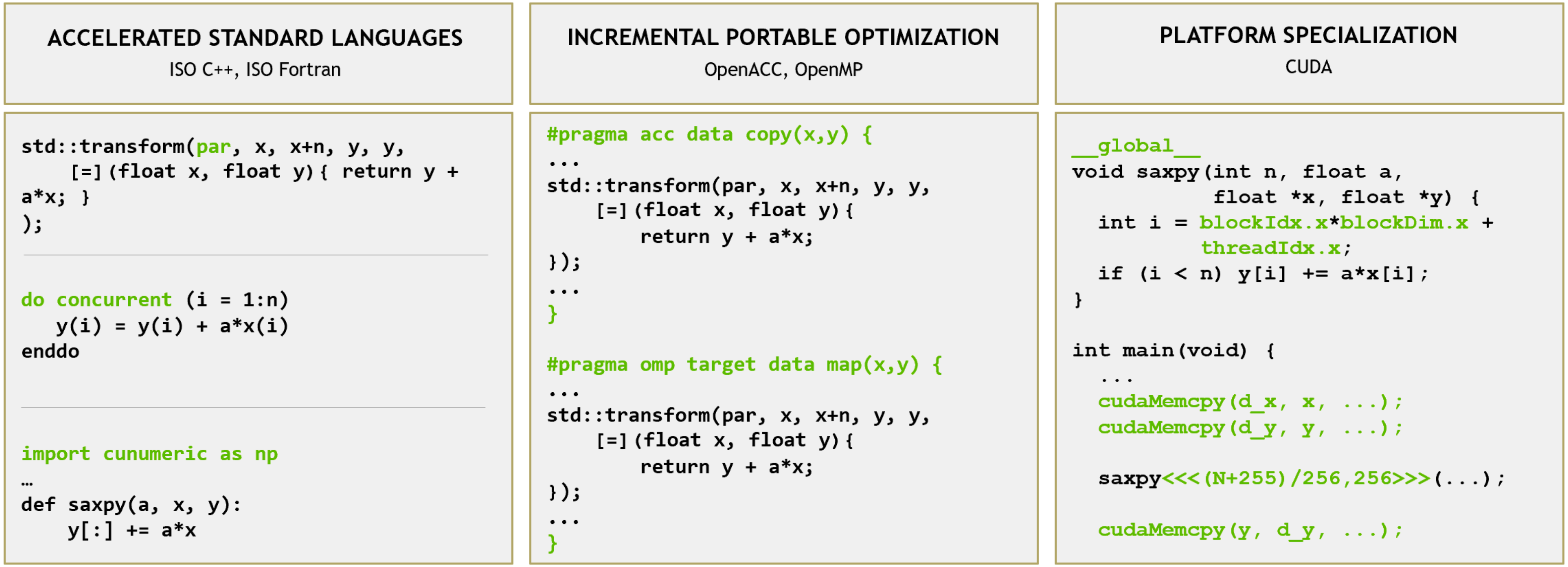


Ponte Vecchio

X^e HPC based GPU

Compute	Up to **128** Ray tracing Units	**Highest Compute Density** socket & node	**128 X^e Cores**
Memory	Up to **64MB** L1 cache in 2 Stacks	Up to **408MB** L2 Cache in 2 Stacks	**HBM2e**
I/O	Up to **8** **Fully Connected GPUs**	PCIe **Gen 5**	**X^e Link** High-Speed Coherent Unified Fabric
Technology	**EMIB**	**Foveros**	Intel 7 TSMC N5 TSMC N7

Programming Nvidia GPU’s



ACCELERATED STANDARD LANGUAGES

ISO C++, ISO Fortran

```
std::transform(par, x, x+n, y, y,
    [=](float x, float y){ return y +
a*x; }
);
```

```
do concurrent (i = 1:n)
   y(i) = y(i) + a*x(i)
enddo
```

```
import cunumeric as np
…
def saxpy(a, x, y):
    y[:] += a*x
```

INCREMENTAL PORTABLE OPTIMIZATION

OpenACC, OpenMP

```
#pragma acc data copy(x,y) {
...
std::transform(par, x, x+n, y, y,
    [=](float x, float y){
        return y + a*x;
});
...
}
```

```
#pragma omp target data map(x,y) {
...
std::transform(par, x, x+n, y, y,
    [=](float x, float y){
        return y + a*x;
});
...
}
```

PLATFORM SPECIALIZATION

CUDA

```
__global__
void saxpy(int n, float a,
           float *x, float *y) {
  int i = blockIdx.x*blockDim.x +
          threadIdx.x;
  if (i < n) y[i] += a*x[i];
}

int main(void) {
  ...
  cudaMemcpy(d_x, x, ...);
  cudaMemcpy(d_y, y, ...);

  saxpy<<<(N+255)/256,256>>>(...);

  cudaMemcpy(y, d_y, ...);
```

OpenACC

- Open standard for addressing the acceleration of Fortran, C and C++ applications

    Originally designed by Cray, PGI and Nvidia

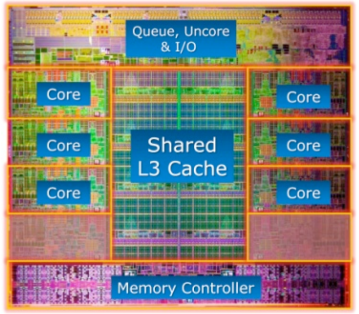
    Directives can be ignored on systems without accelerator

    Can be used to target accelerators from Nvidia, AMD and Intel

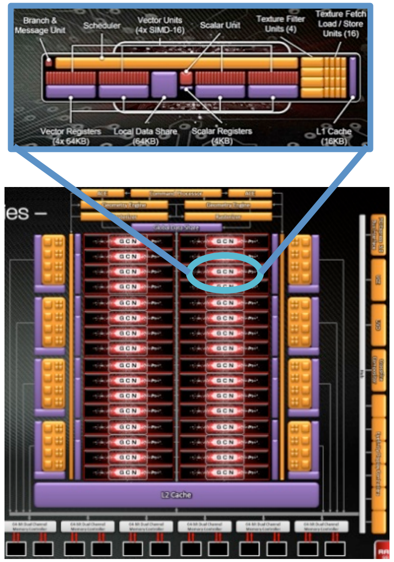
    http://www.openacc-standard.org/

Which accelerator?

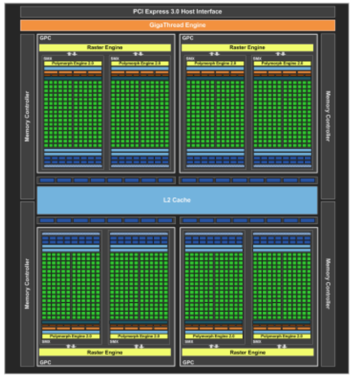
- HPC market too small and likely that only one accelerator will dominate HPC
- Vector based accelerators
- Flops/Watt an important factor



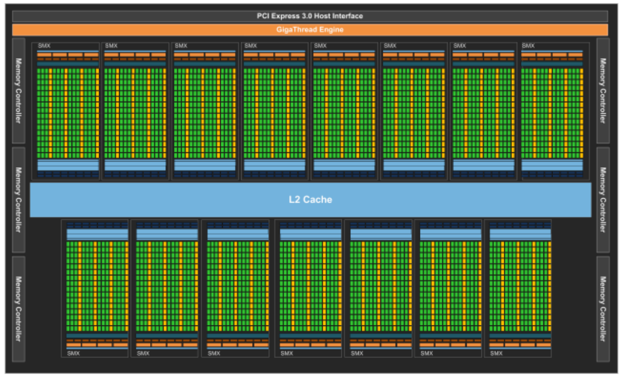
Intel Sandy Bridge
Core I7 3960X
(6 core)



AMD 7970
(1,280 work-item)



NVIDIA Fermi
(1,536 CUDA cores)



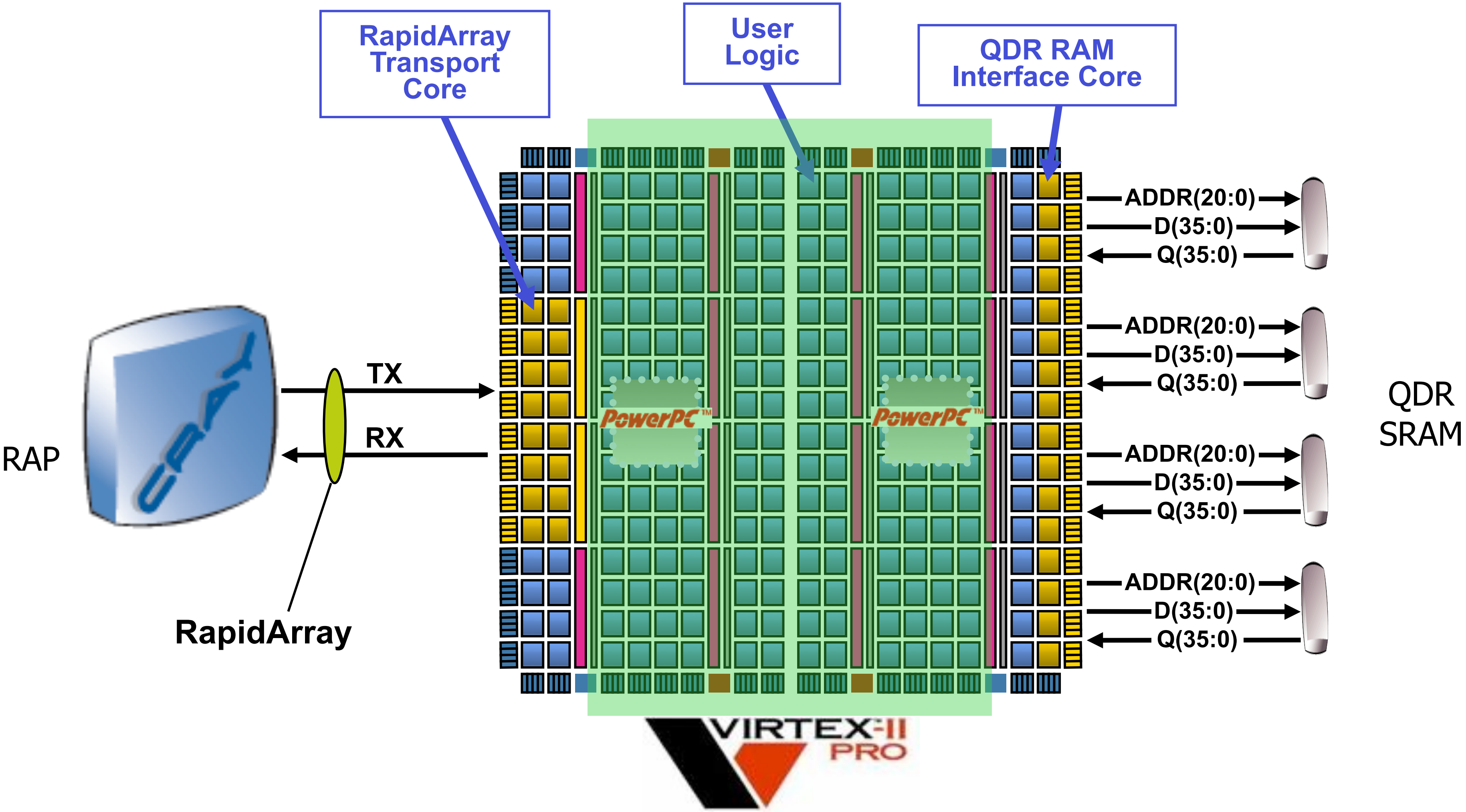
NVIDIA Kepler K20
(2,880 CUDA cores)

Parallel programming skills are needed to program these chips.

FPGA

- Field-programmable gate array
- Adjust the architecture to the needs of your algorithm
- Invented 1984
- Used heavily in embedded and real-time systems
- Occasionally Use in supercomputers like Cray XD1, SGI RASC, Convey, SRC computing
- Programmability!
- An overview can be found at: [http://en.wikipedia.org/wiki/Field-programmable_gate_array]

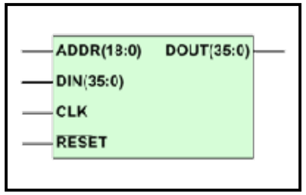
Application Acceleration Interface



- XC2VP30 running at 200 MHz.
- 4 QDR II RAM with over 400 HSTL-I I/O at 200 MHz DDR (400 MTransfers/s).
- 16 bit simplified HyperTransport I/F at 400 MHz DDR (800 MTransfers/s.)
- QDR and HT I/F take up <20 % of XC2VP30. The rest is available for user applications.

FPGA Development Flow

Cores

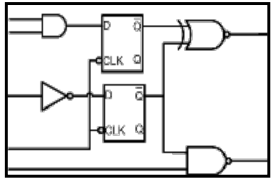


RAP I/F,
QDR RAM I/F

HDL

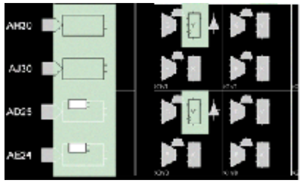
VHDL,
Verilog,
C

Synthesize



Synplicity,
Leonardo,
Precision,
Xilinx ISE

Implement



Xilinx ISE

Metadat
a

0100010101
1010101011
0100101011
0101011010
1001110101
0110101010

Binary File

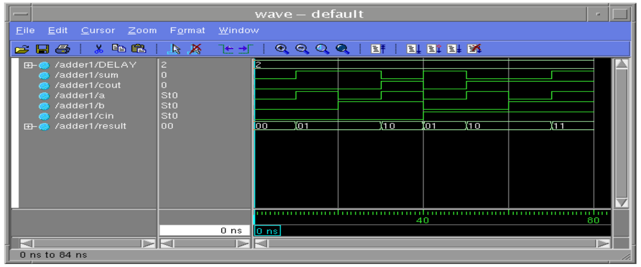
Download

From Command line
or Application

Verify

Xilinx
ChipScope

Simulate



```
/adder1/DELAY    2
/adder1/sum      0
/adder1/cout     0
/adder1/a        St0
/adder1/b        St0
/adder1/cin      St0
/adder1/result   00
```

0 ns to 84 ns

Modelsim

Looking at the future

The future in 2000

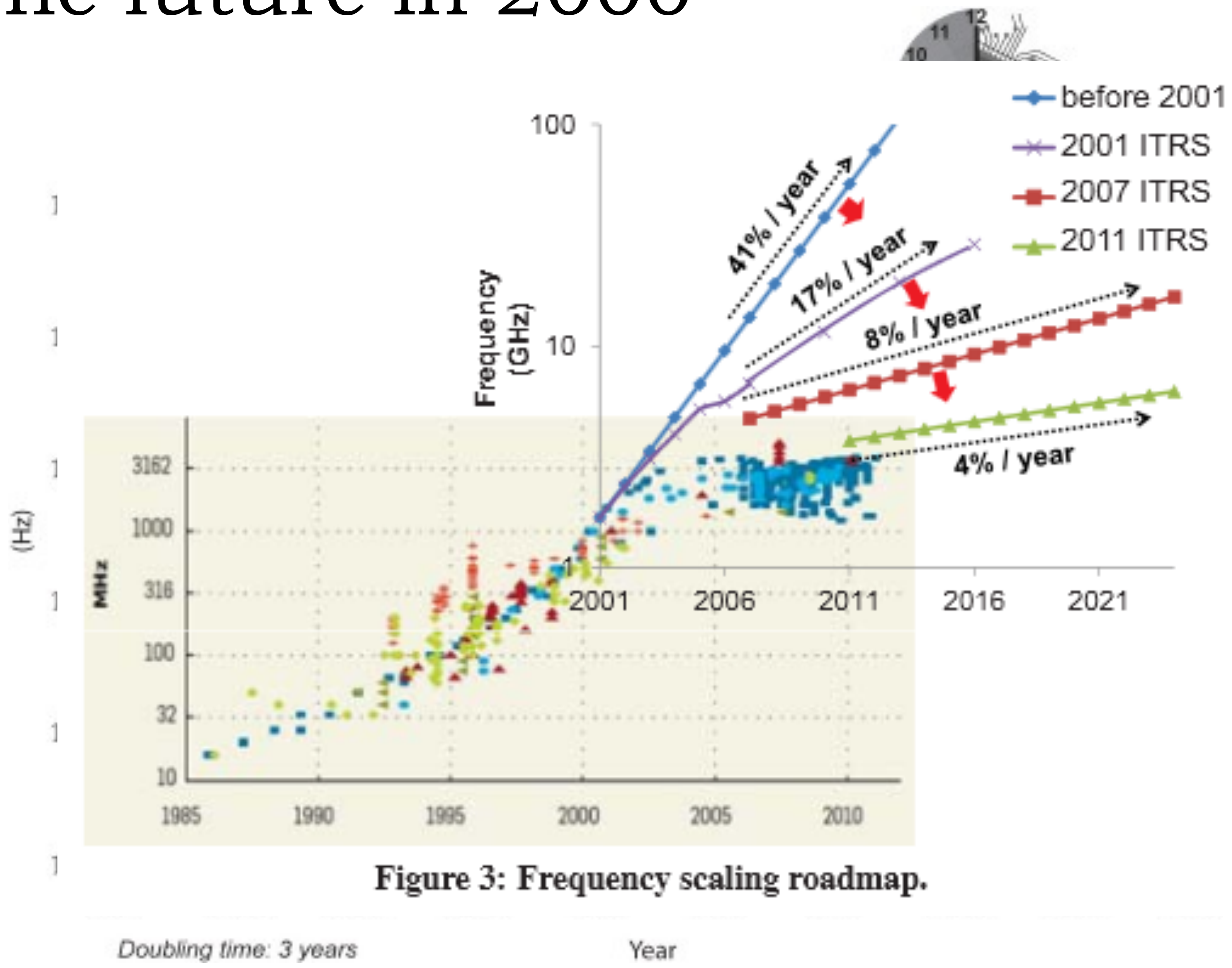
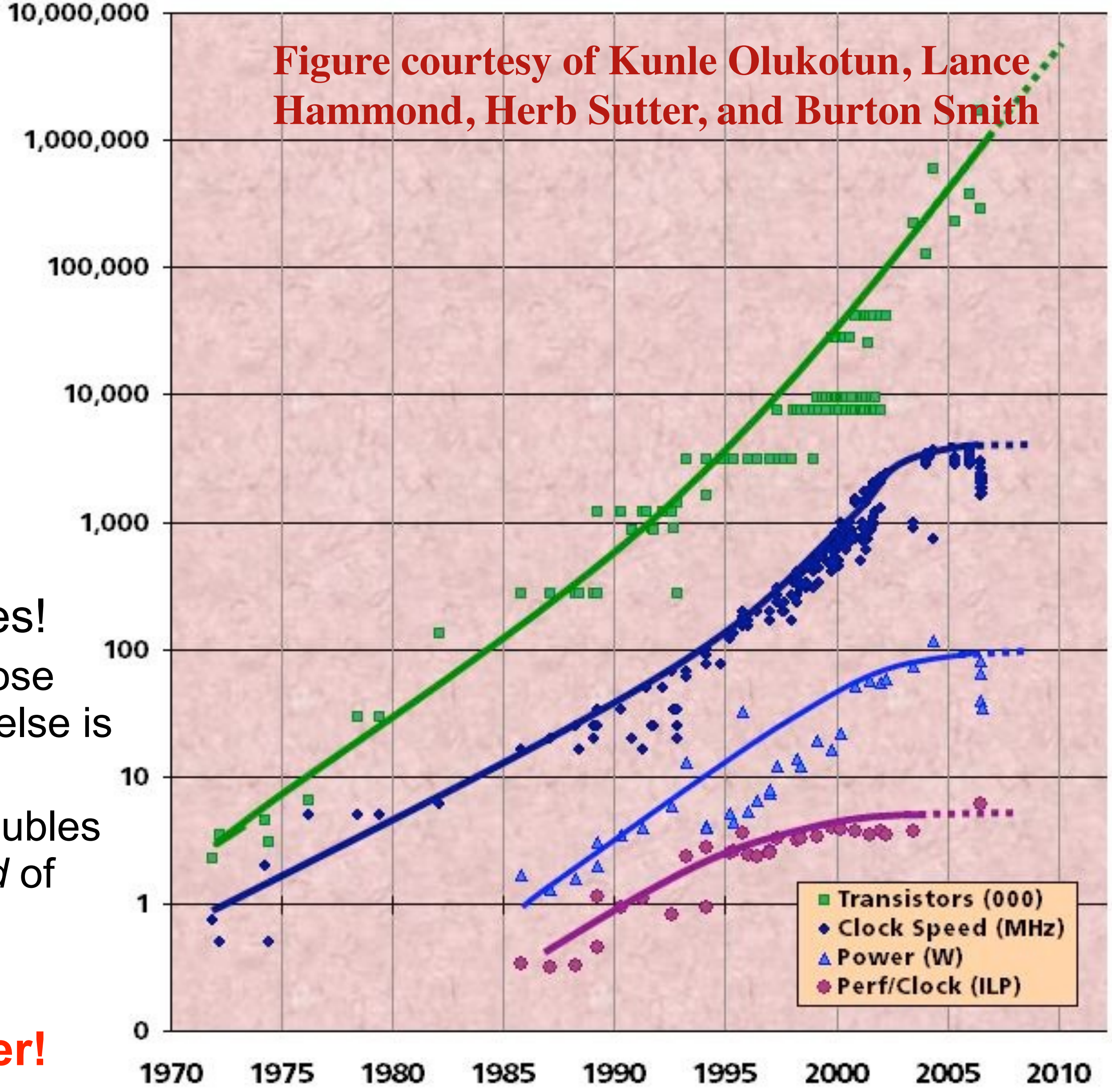


Figure 3: Frequency scaling roadmap.

Looming Power Crisis

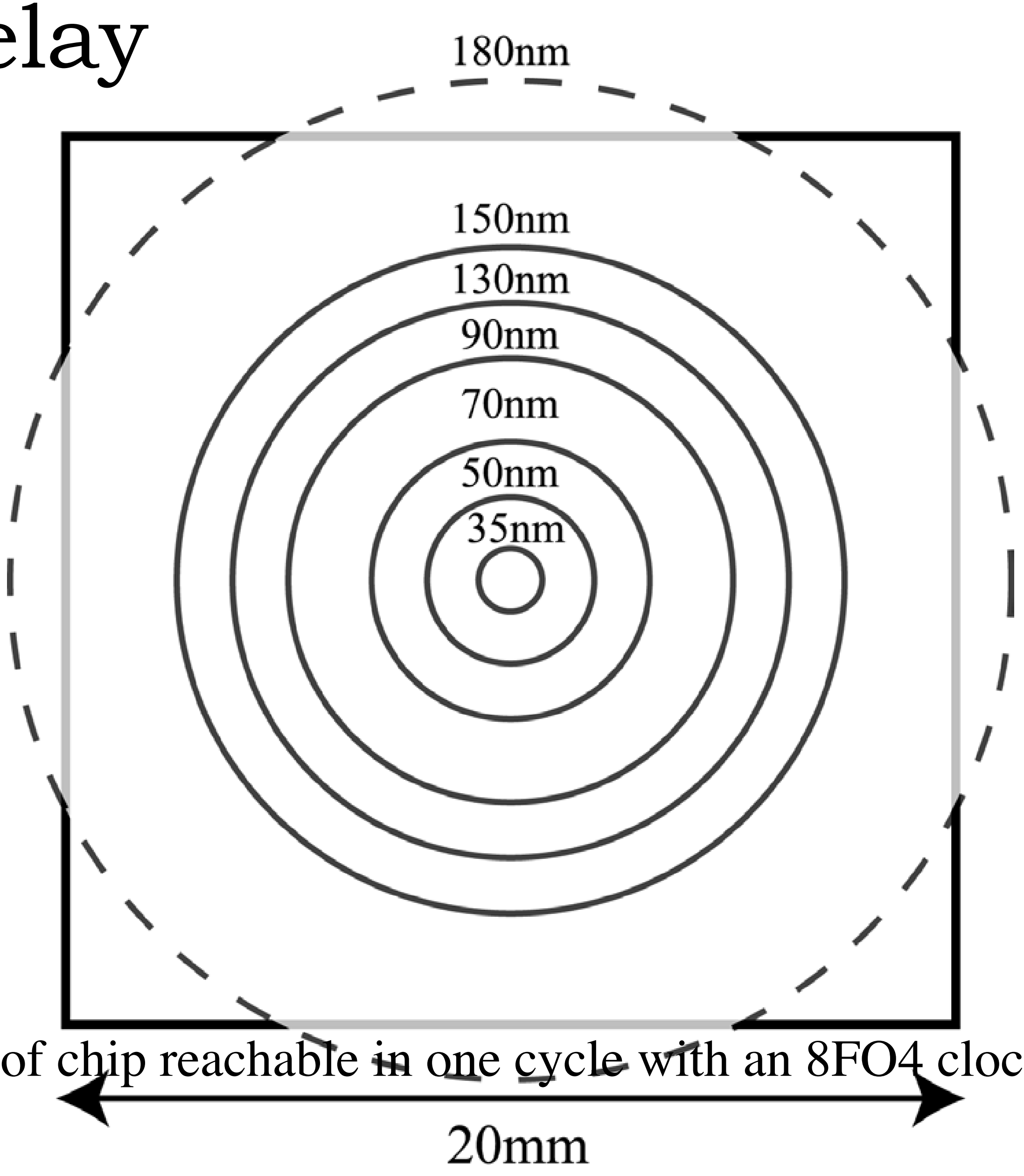
- New Constraints
  - Power limits clock rates
  - Cannot squeeze more performance from ILP *(complex cores)* either!
- But Moore's Law continues!
  - What to do with all of those transistors if everything else is flat-lining?
  - Now, #cores per chip doubles every 18 months *instead* of clock frequency!
- **The "Free Lunch" is over!**



Limitation of chip design

- Size
- Speed
- Power

Wire delay



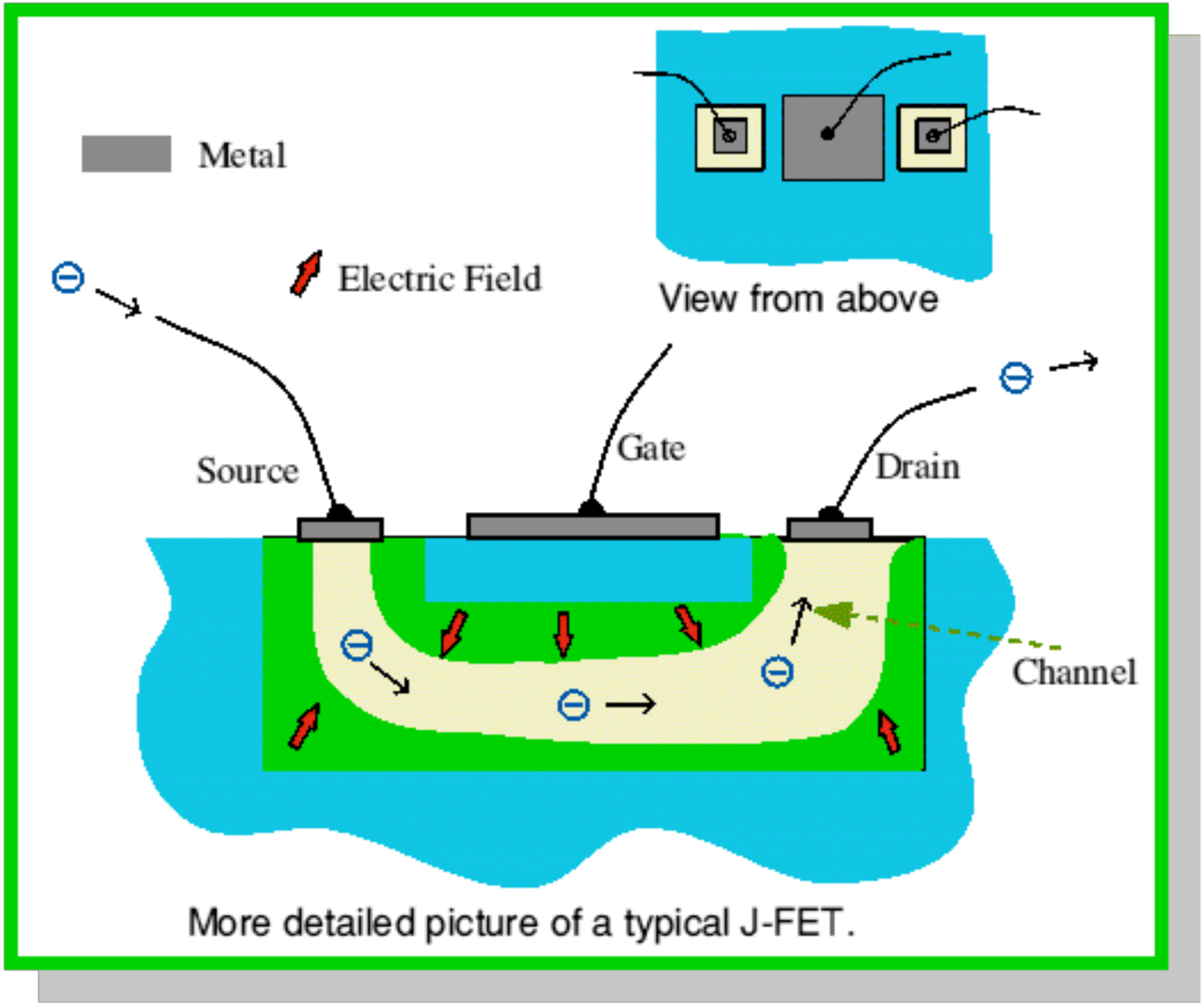
Projected fraction of chip reachable in one cycle with an 8FO4 clock period.

Processor Technology Trends

- Shrinking of transistor sizes: 250nm (1997) => 130nm (2002) => 65nm (2007) => 45nm (2009) => 32nm (2010) => 22 nm (2011/12) => 14 nm (2017/18) => 7 nm (2019/20)

- Transistor density increases by 35% per year and die size increases by 10-20% per year… more cores!

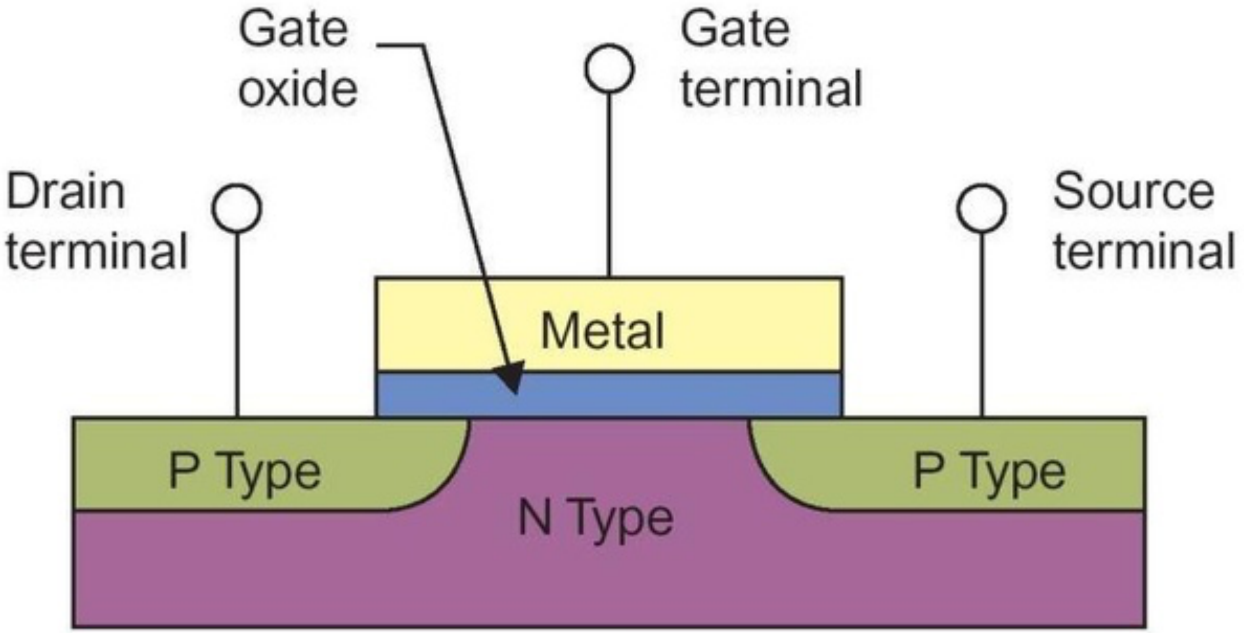
- Transistor speed improves linearly with size (complex equation involving: voltages, resistance's, capacitances, …) and lead to clock speed improvements!



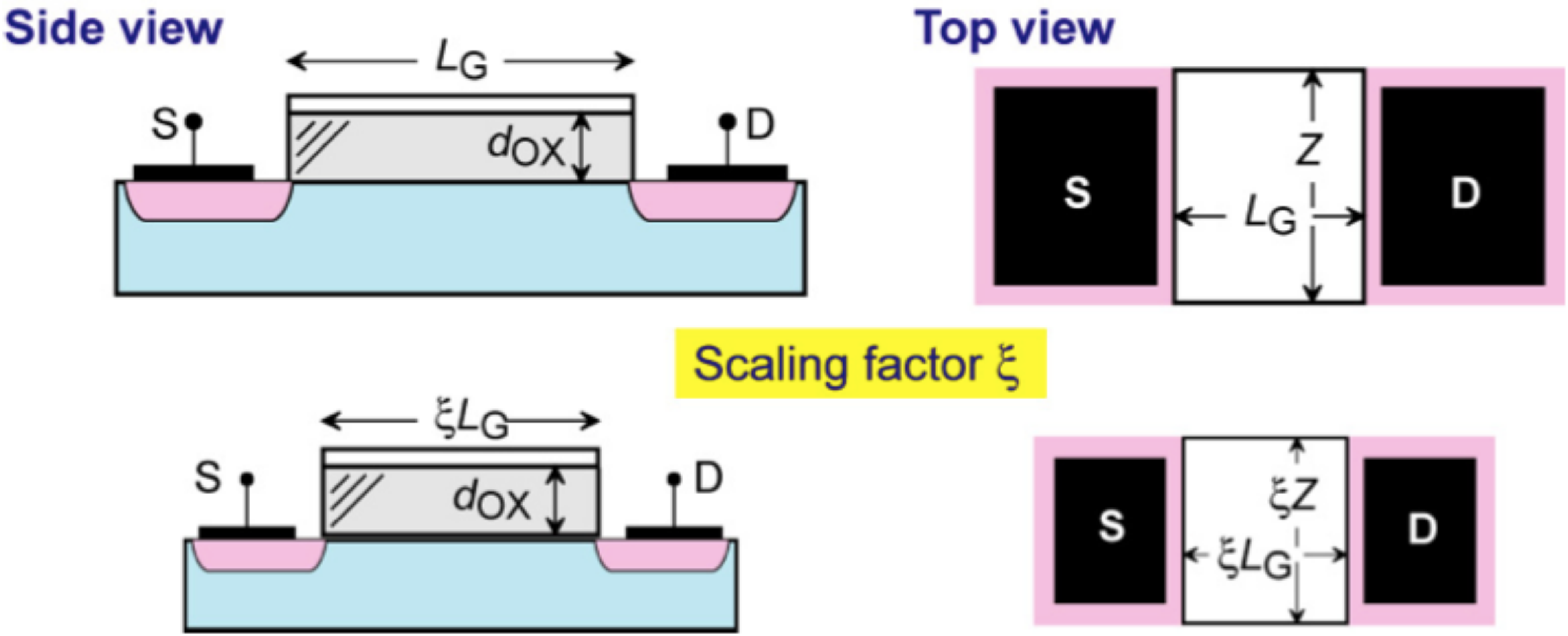
More detailed picture of a typical J-FET.

MOSFET

- Transistor shrinking leads to
  - area of gate gets smaller
  - thinner gate-oxide gives stronger electric fields that allows faster switching (higher processor clock).
  - at 45 nm the gate di-electric is 0.9 nm thick : size of a single SiO^2 molecule.



Shrinking transistors

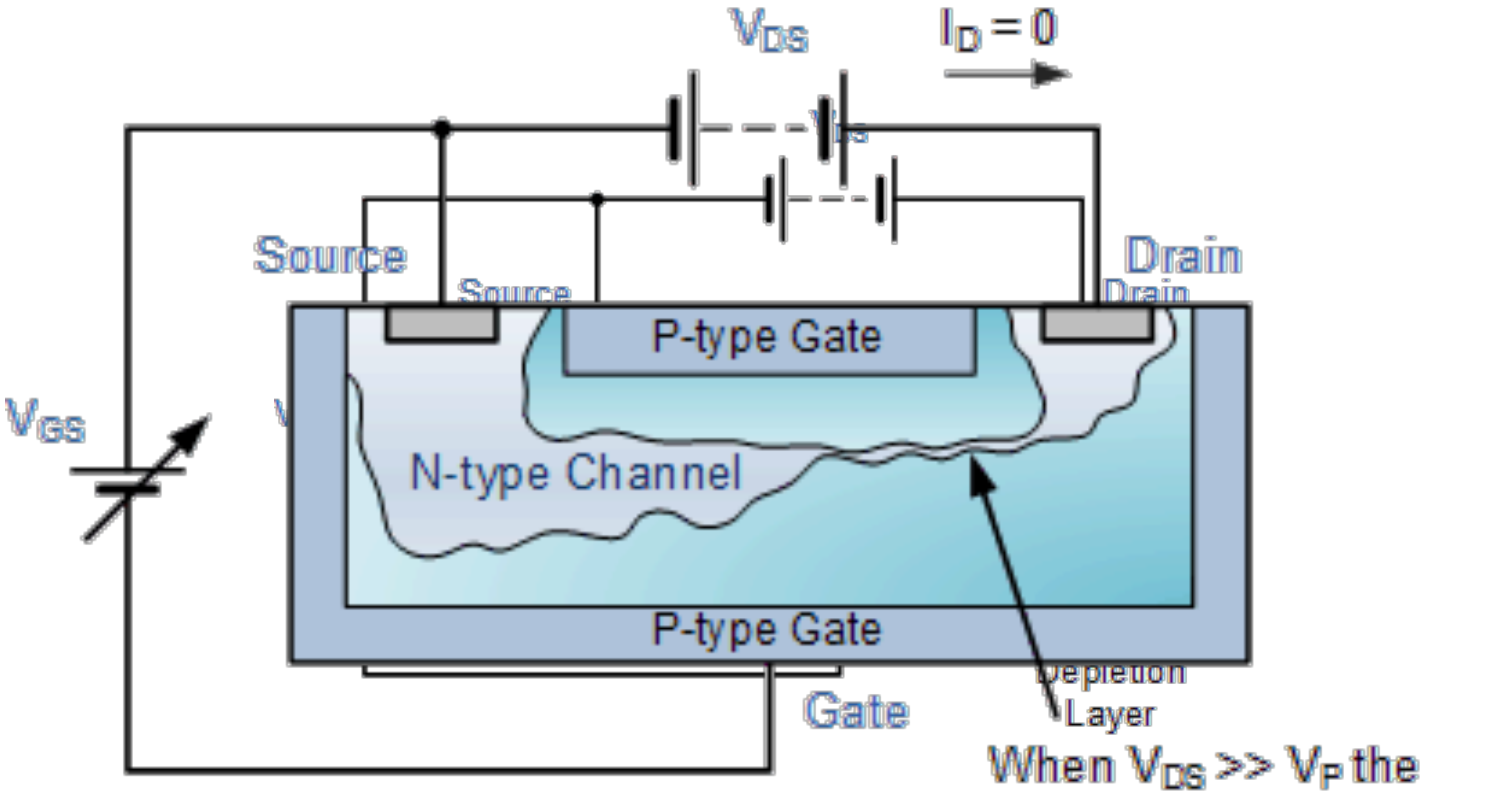


Side view

Top view

Scaling factor ξ

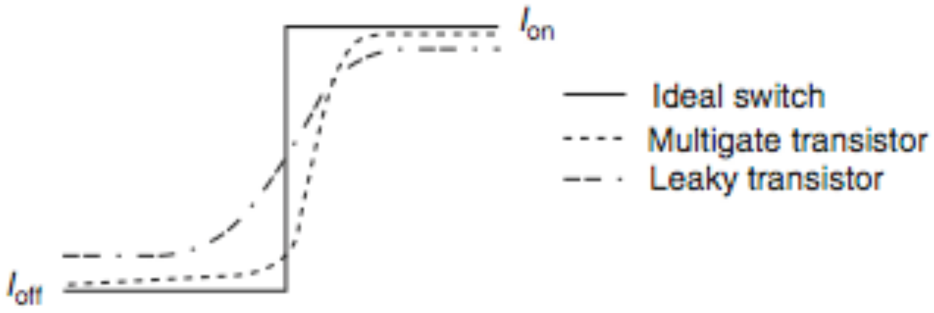
JFET-transistor



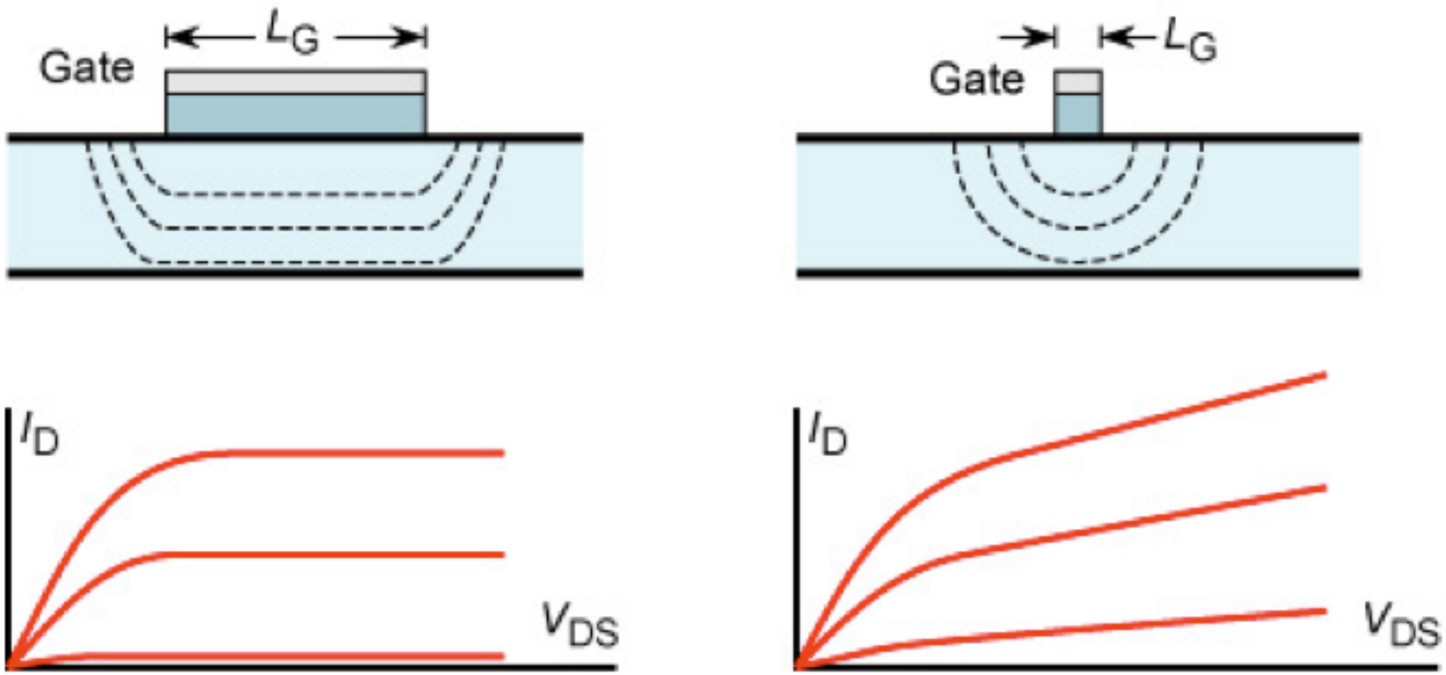
When $V_{DS} >> V_P$ the channel closes "Pinched-off"

Leakage

- The gate is the electrical connection that controls the MOS switch. The gate is separated from the rest of the MOS transistor by an insulating layer. As this layer gets thinner, the transistor performance improves. However, at a certain point, the gate is so thin that it leaks electrons.



short channel effects



Short-channel effects:

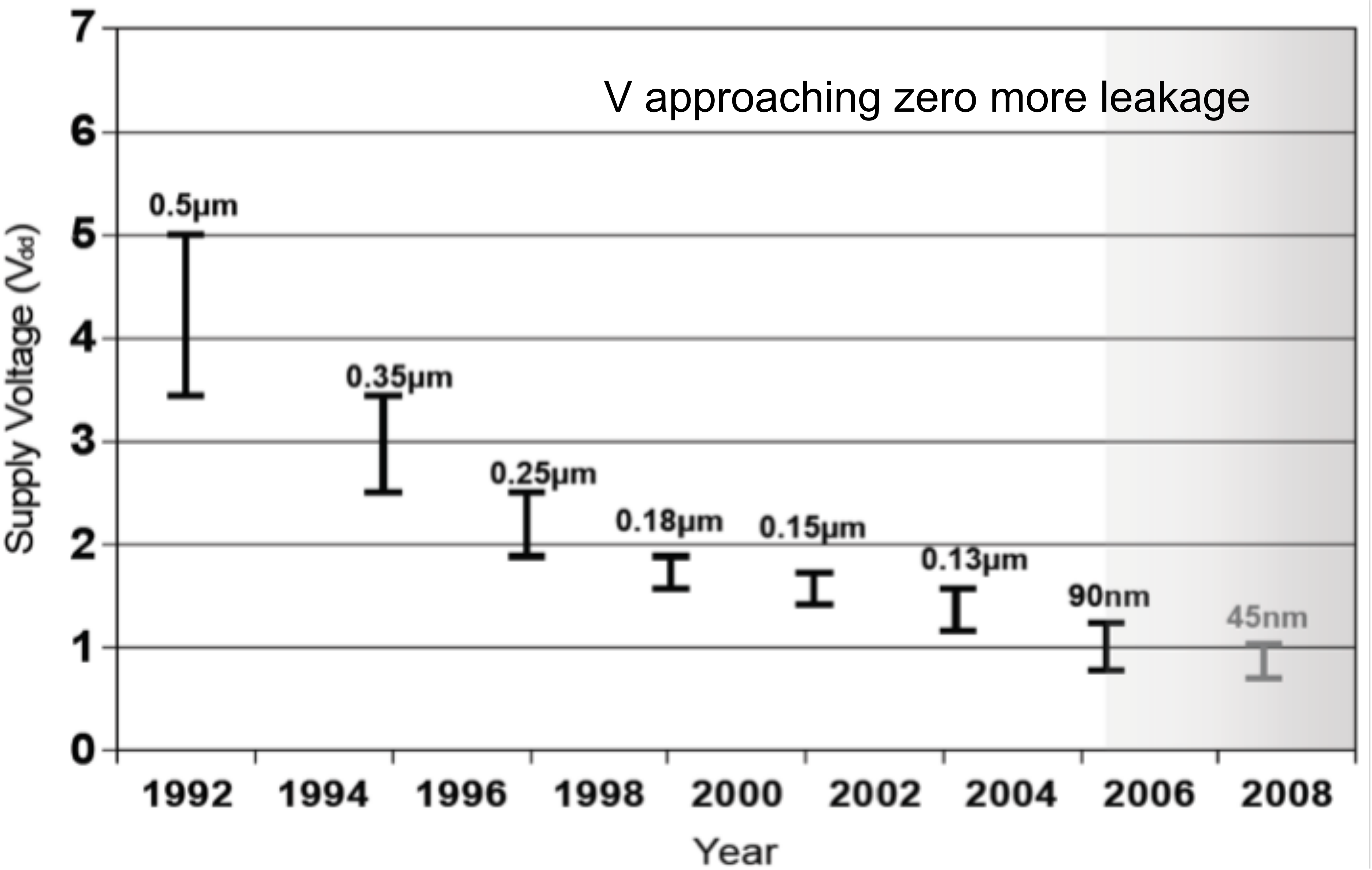
Threshold-voltage shift

Lack of pinch-off

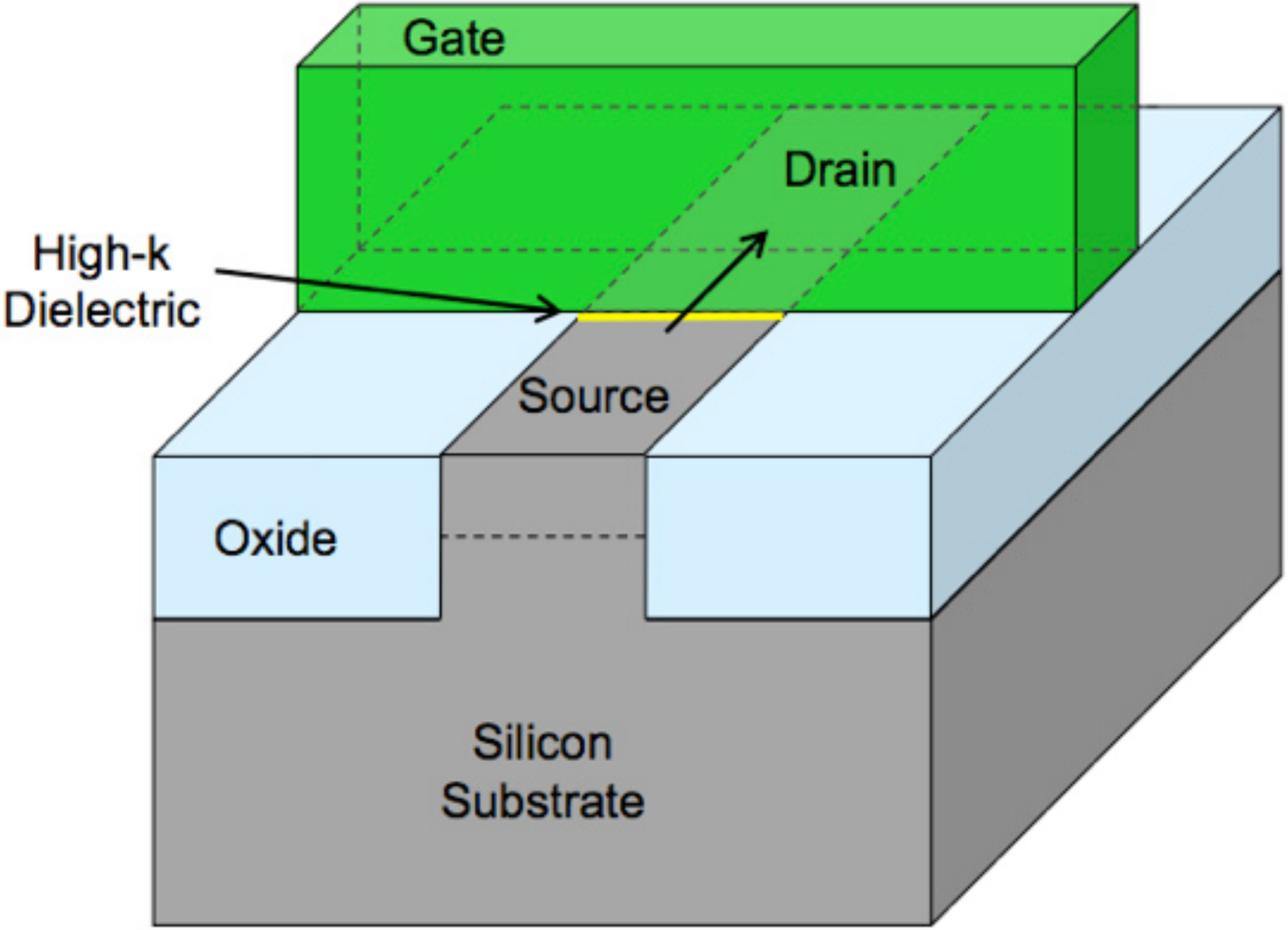
Increased leakage current

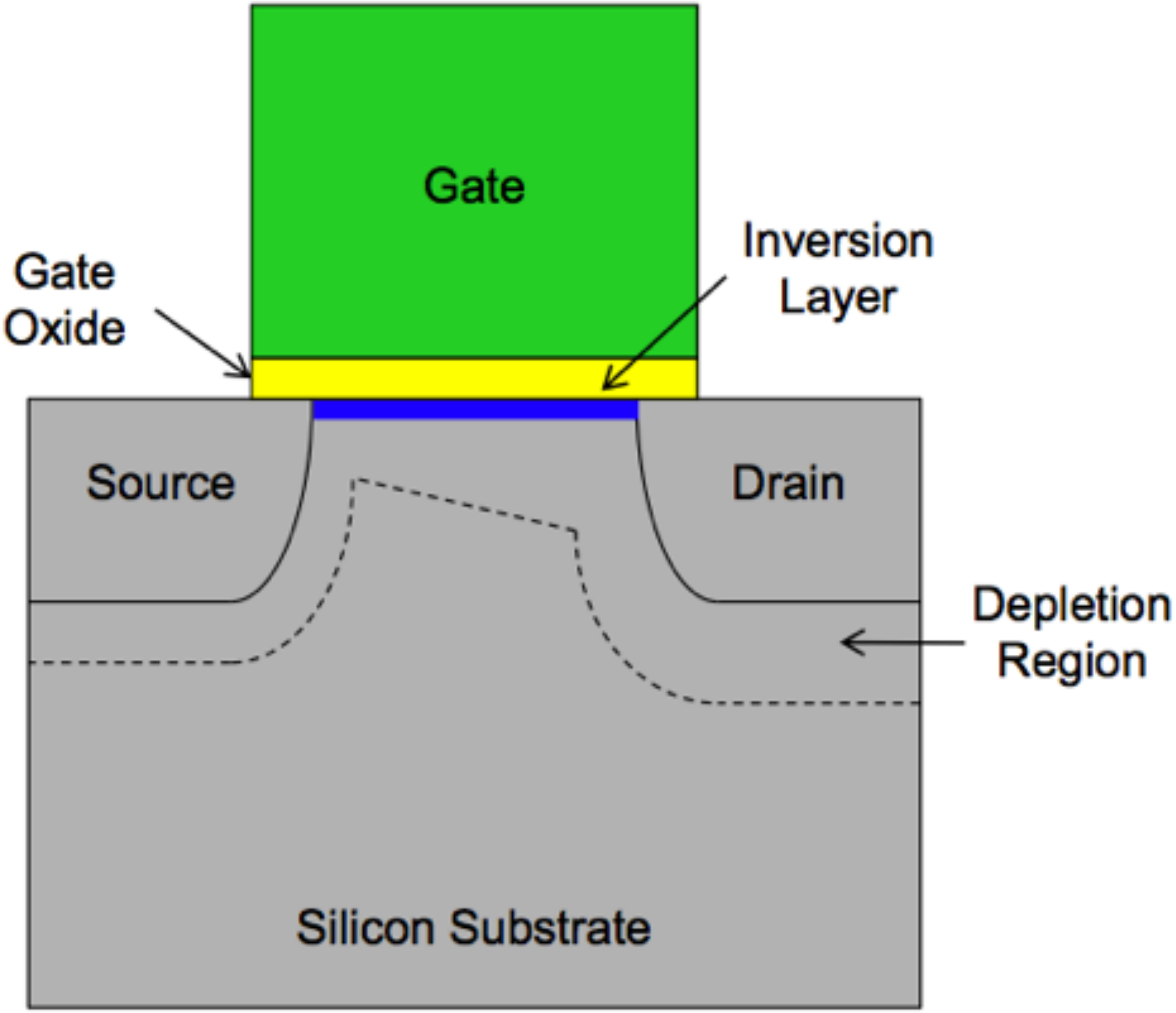
Increase of output conductance

Element Size x Supply Voltage



Source:Semiconductor Industry Association (SIA), US


Gate
Drain
High-k
Dielectric
Source
Oxide
Silicon
Substrate


Gate
Gate Oxide
Inversion Layer
Source
Drain
Depletion Region
Silicon Substrate

- Time for a Disney movie

Physical limitations: Power

- The most difficult problem is to control power dissipation.

  ~280 watts is considered a maximum power output of a processor.

  As we pack more transistors, the power output goes up and better cooling is necessary.

  current leakage increases with smaller chip design

  Power = $C * f * V^2$ ~ area * frequency * Voltage^2

What is Happening Now?

- Moore’s Law
  - Silicon lithography will improve by 2x every 18 mth
  - Double number of transistors per chip every 18 mth

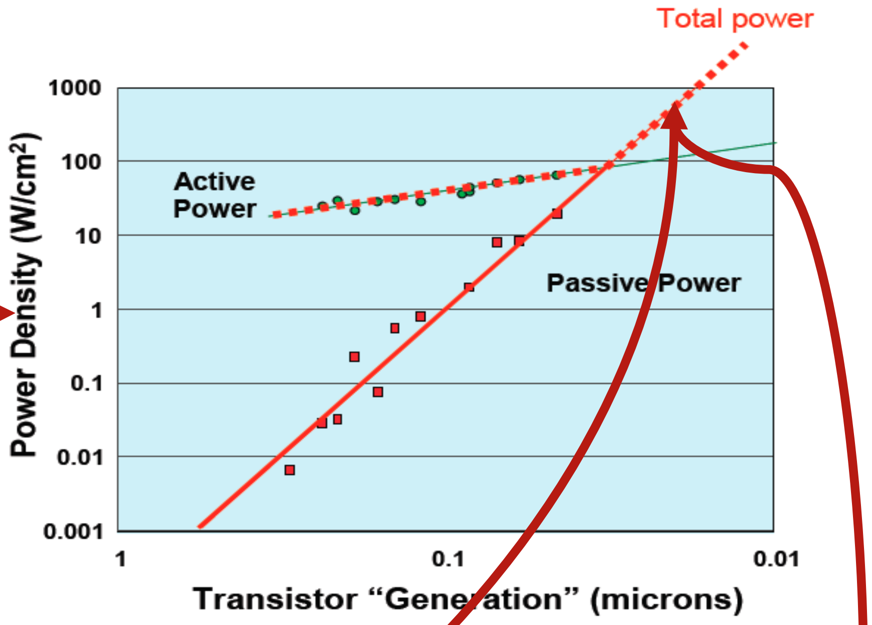
- CMOS Power

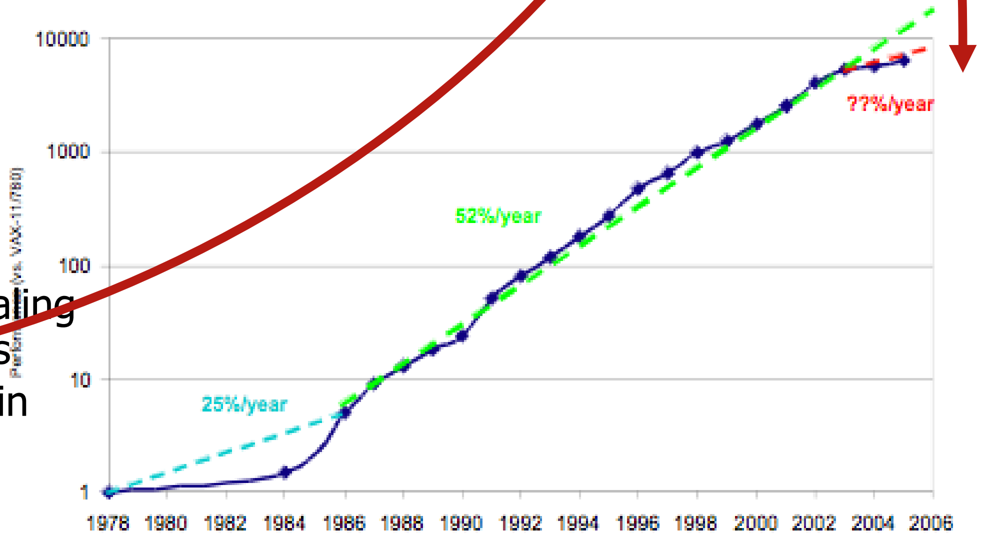
  Total Power = $V^2 * f * C + V * I_{leakage}$

  active power passive power

  - As we reduce feature size Capacitance (C) decreases proportionally to transistor size
  - Enables increase of clock frequency (f) proportionally to Moore’s law lithography improvements, with same power use
  - This is called “Fixed Voltage Clock Frequency Scaling” *(Borkar `99)*

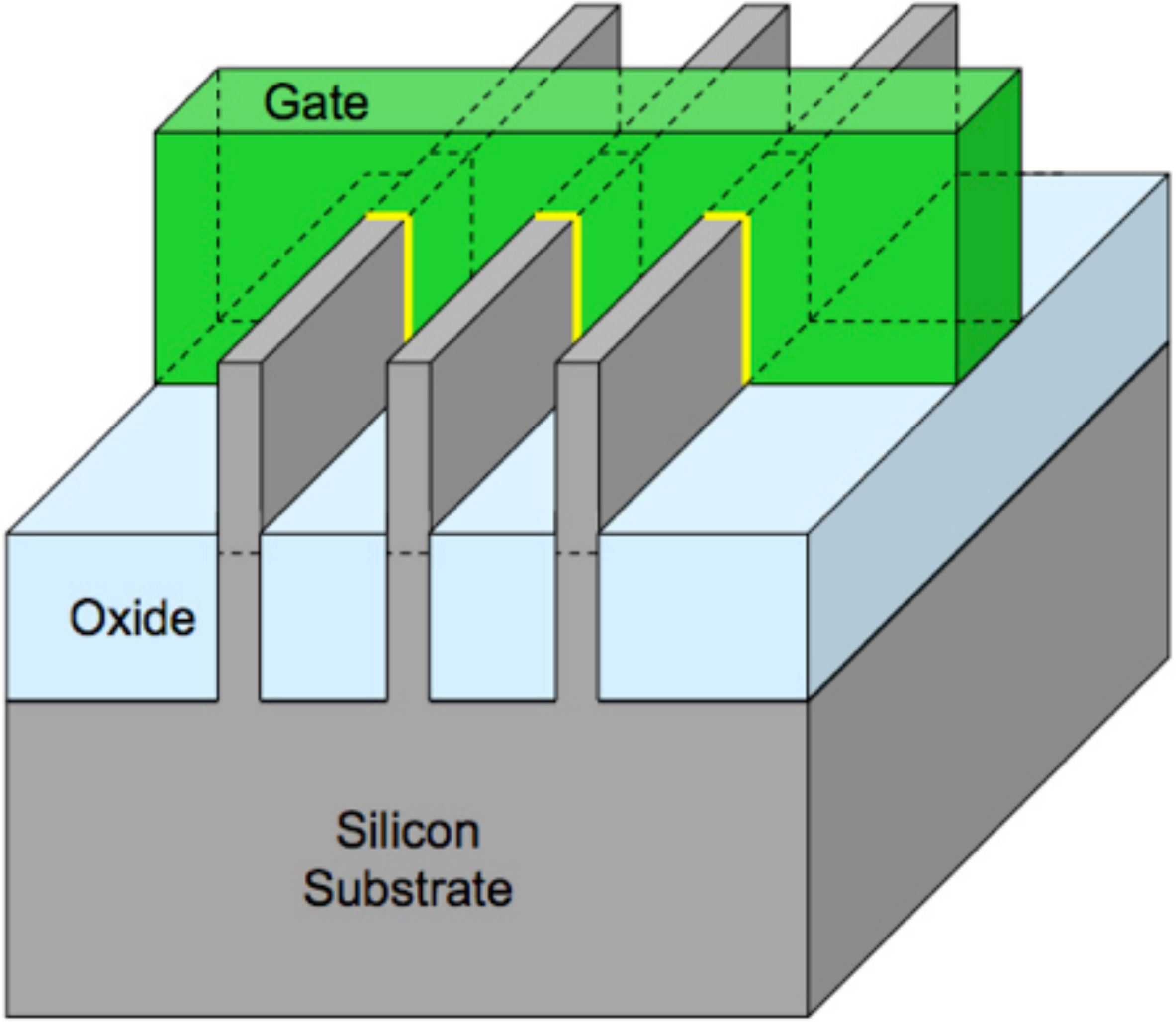
- Since ~90nm
  - $V^2 * f * C \sim= V * I_{leakage}$
  - Can no longer take advantage of frequency scaling because passive power ($V * I_{leakage}$) dominates
  - Result is recent clock-frequency stall reflected in Patterson Graph at right



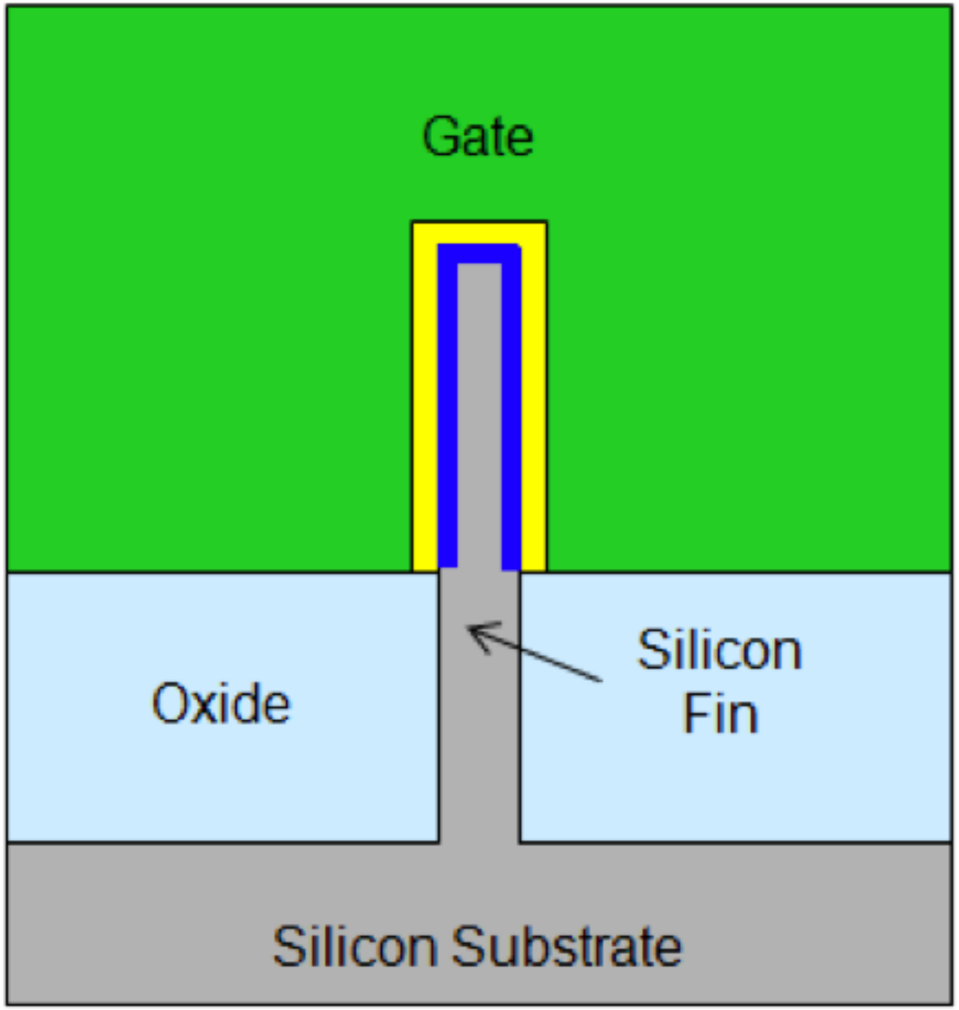


SPEC_Int benchmark performance since 1978 from Patterson & Hennessy Vol 4.

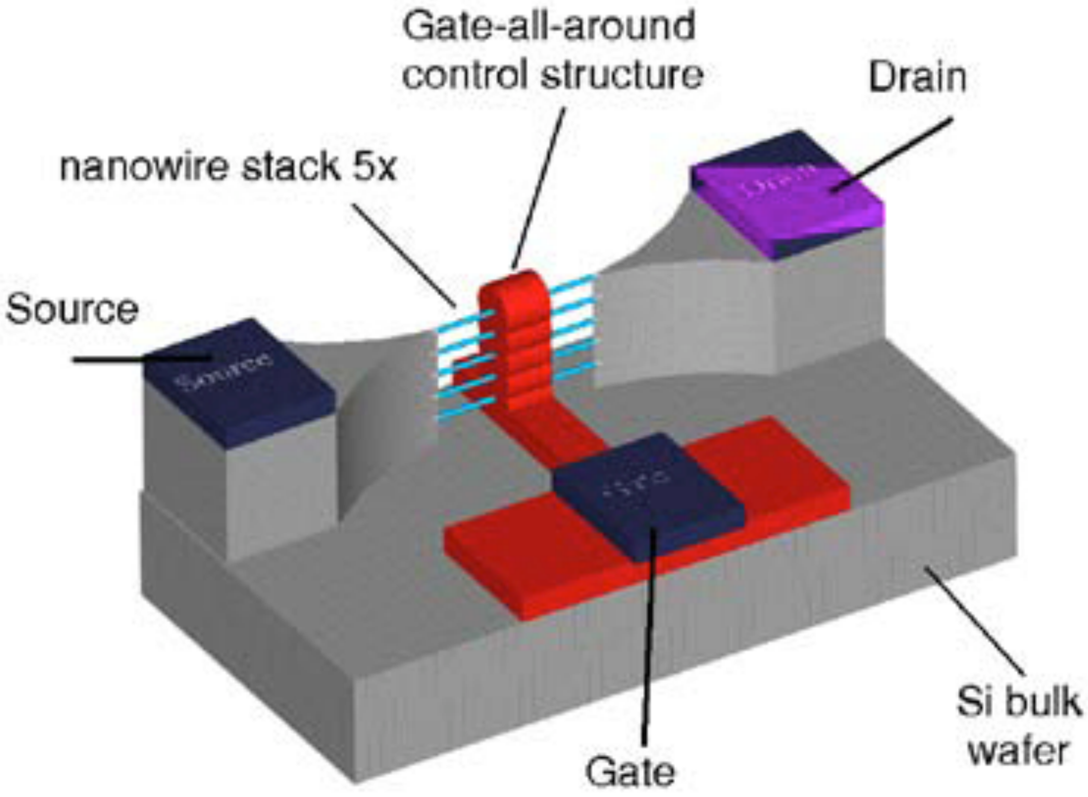
Developments Transistors



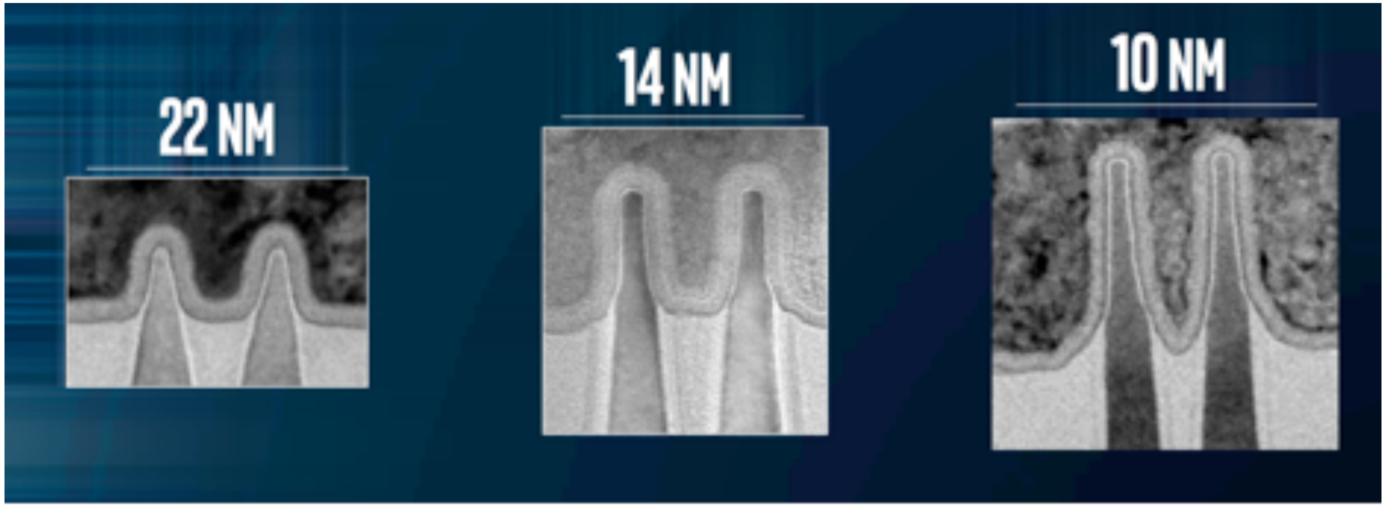
fin-FET

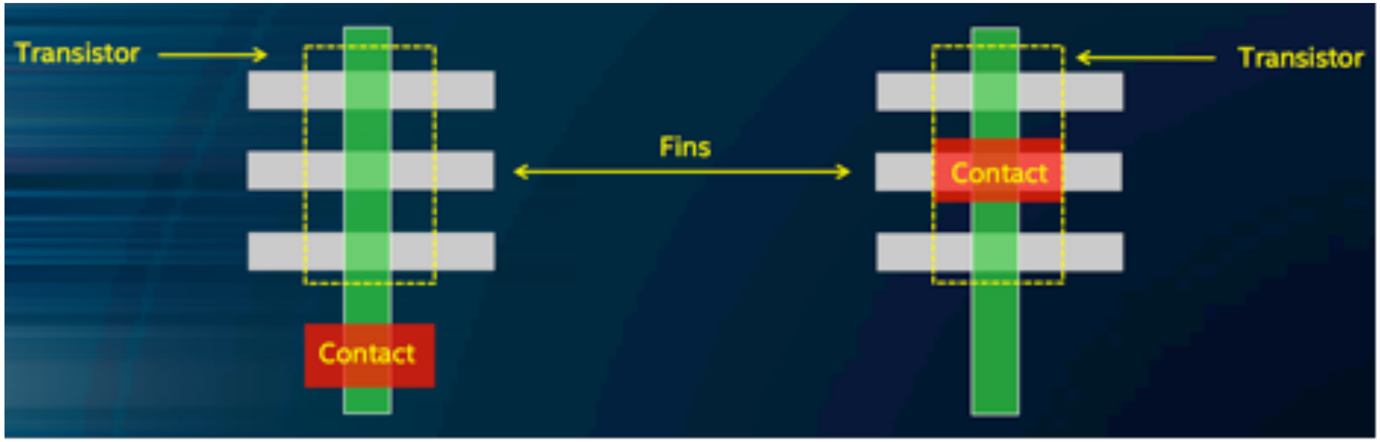


gate-all-around (2020)



The nm story


22 NM
14 NM
10 NM


Transistor
Fins
Contact
Transistor
Contact

Where are we headed and Why?

- Modern trends:
  - Clock speed improvements are not increasing
    - power constraints
    - already doing less work per stage
  - Difficult to further optimize a single core for performance
  - Multi-cores: each new processor generation will accommodate more cores
  - Integrated of functionality on the die:
    - memory controller
    - direct connect to other processor(s)
    - PCI
    - network interface chip (NIC)…

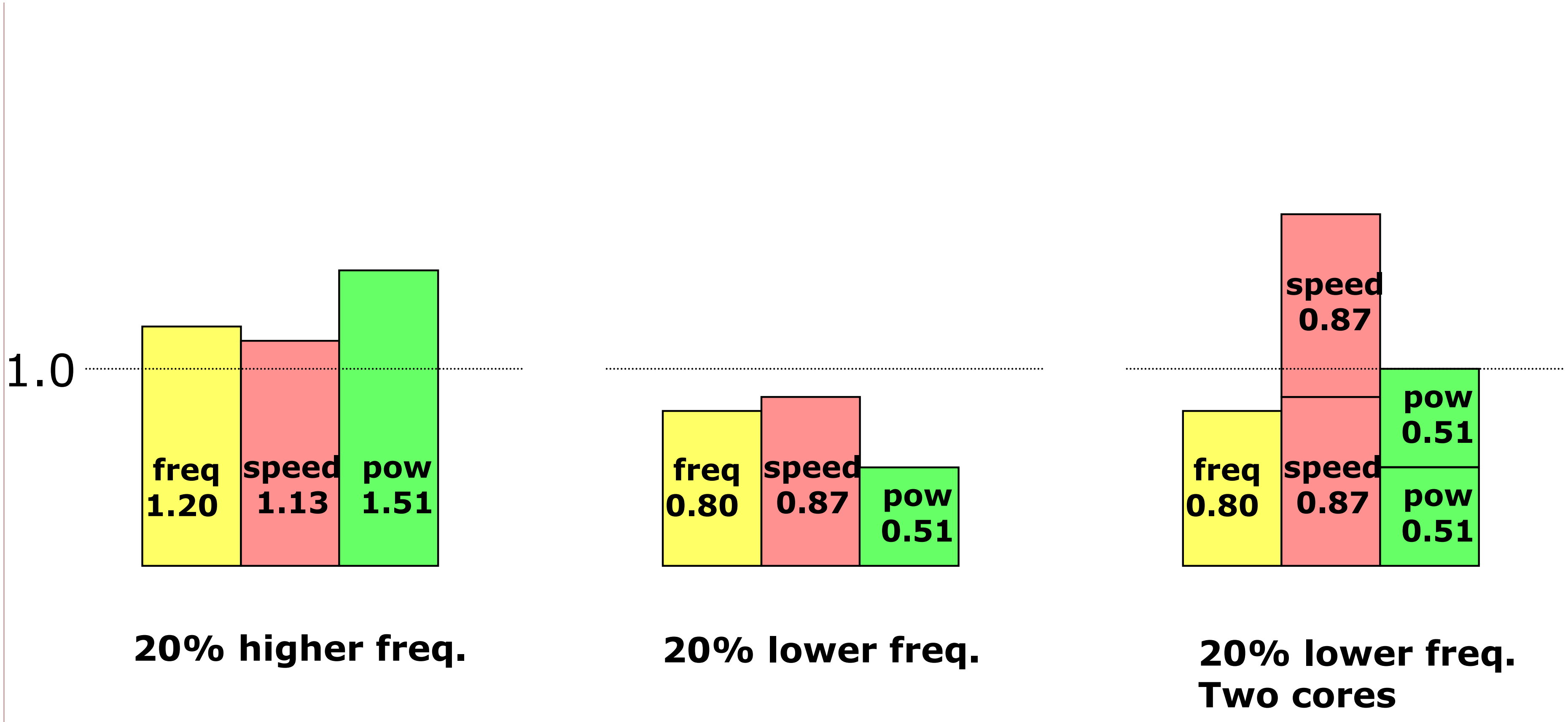
Why multi-core

- Not enough ILP (Instruction Level Parallelism), adding more will not get faster runtimes
  - all ILP has already been explored the last 20 years

- Signal propagation delay >> transistor delay

- Power consumption $P_{active} \sim C * f * V^2 \sim f^3$

Frequency Scaling



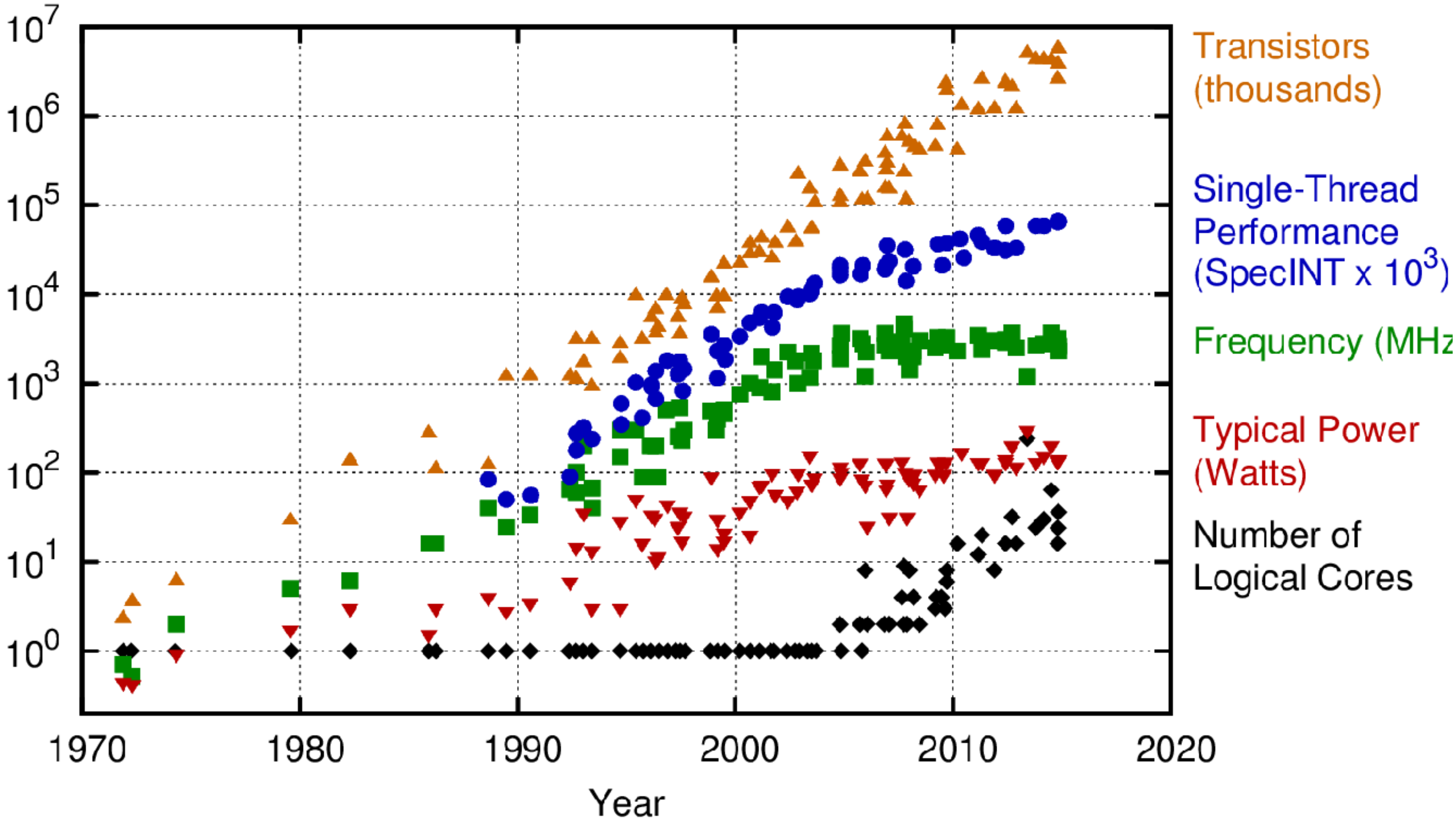
Trends

- **Frequency scaling is now prevented by physical constraints**
  - Heat (too much of it and too hard to dissipate)
  - Power Consumption (too high)
  - Current leakage problems

- **Future performance gains will come from**
  - Hyperthreading
  - Multicore
  - Cache

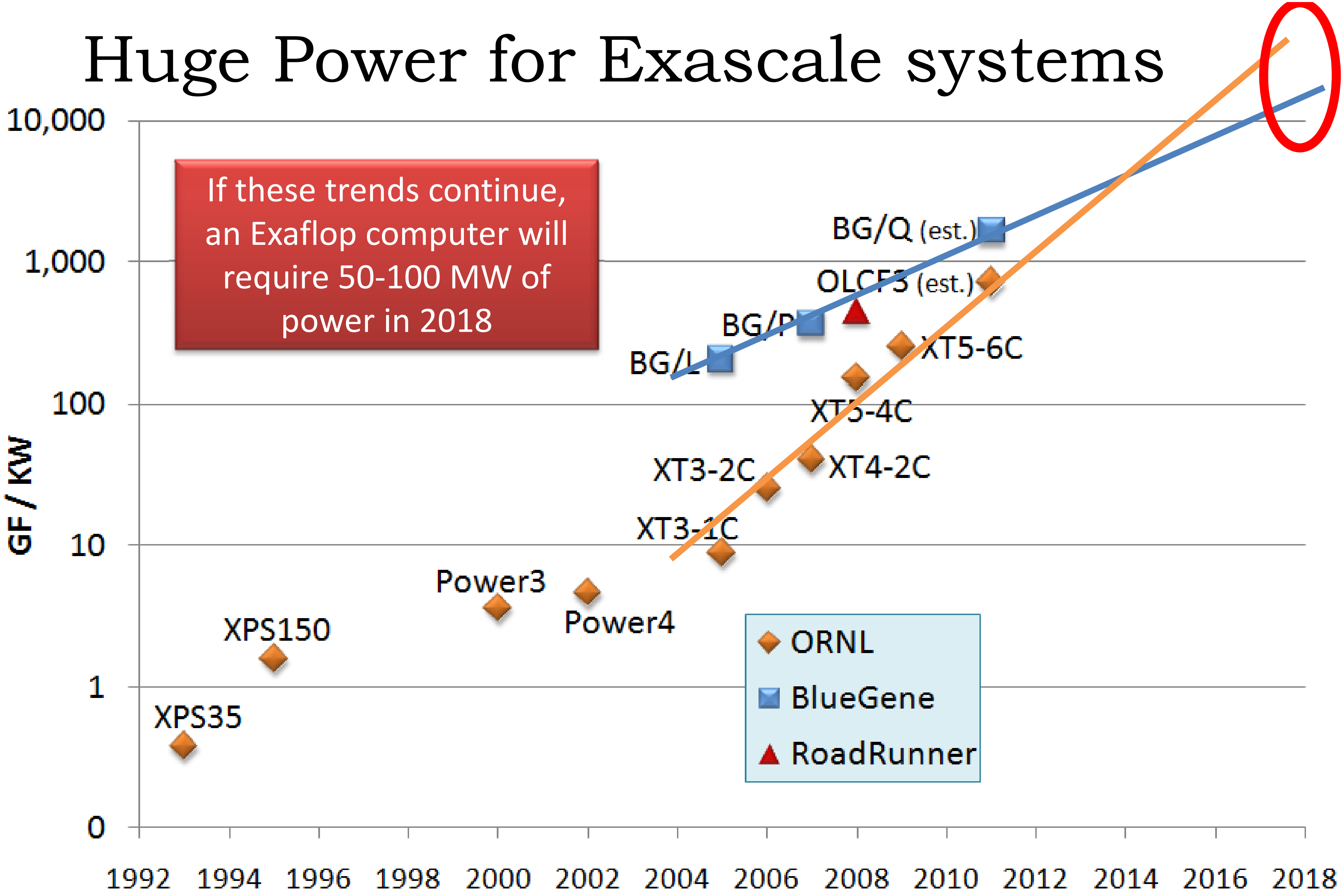
- **This requires better and parallel software !**

40 Years of Microprocessor Trend Data



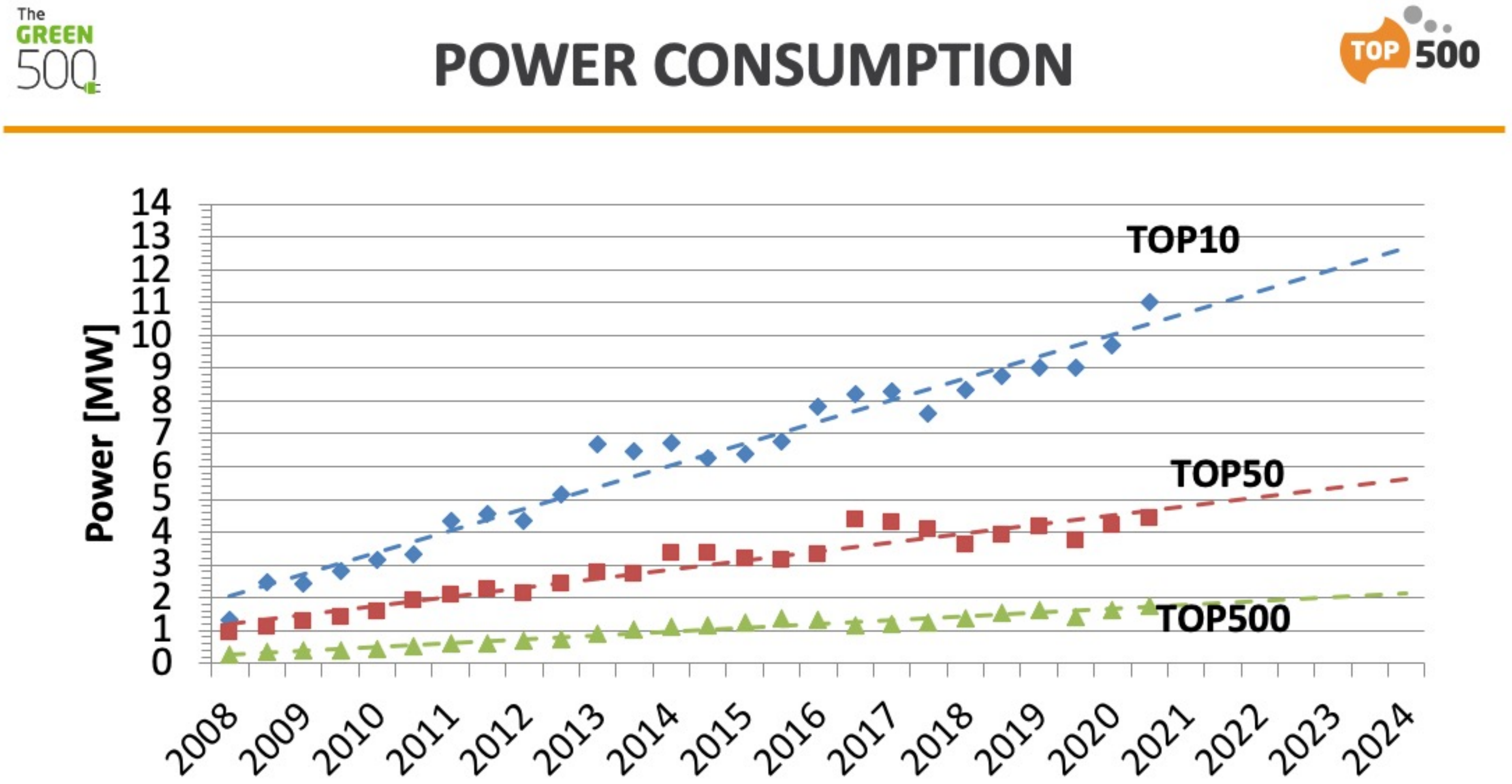
Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2015 by K. Rupp

Huge Power for Exascale systems



If these trends continue, an Exaflop computer will require 50-100 MW of power in 2018

It’s not that bad



POWER CONSUMPTION

Power management on modern cores

- PM ensure that cores do not overheat and remain functional for a longer time.
- Modern processors (x86) tend to be power limited rather than frequency limited
- Different workloads (i.e., executed instruction sequences) will generate different amounts of power consumption in the processor. This can grow quite large.
- Current processors from AMD and Intel contain dedicated microcontrollers that administer power management.
  If changes in the operating scenario cause any one parameter to approach its limit, the controller must throttle the processor’s performance to compensate. These adjustments can happen every millisecond.

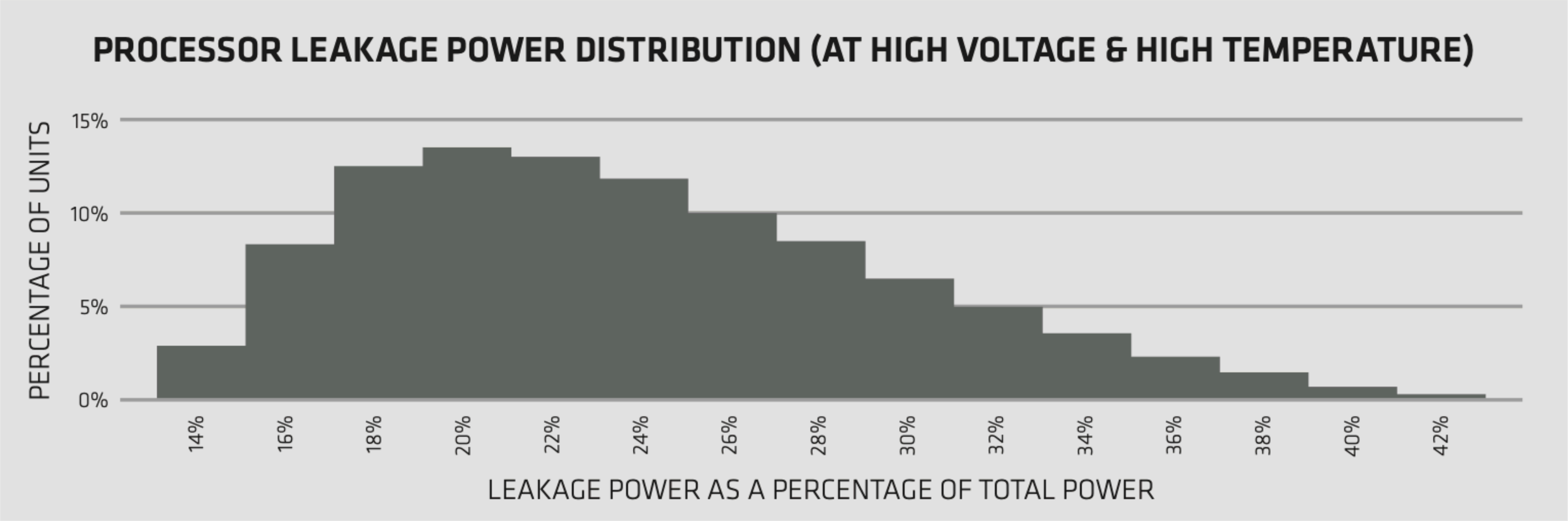


Figure 1- Leakage power distribution for an undisclosed AMD product based on a 14nm FinFET process.

- hardware performance varies ~30% within the same processor
- The same factors that are required to make transistors switch faster (higher frequency) also increase leakage.

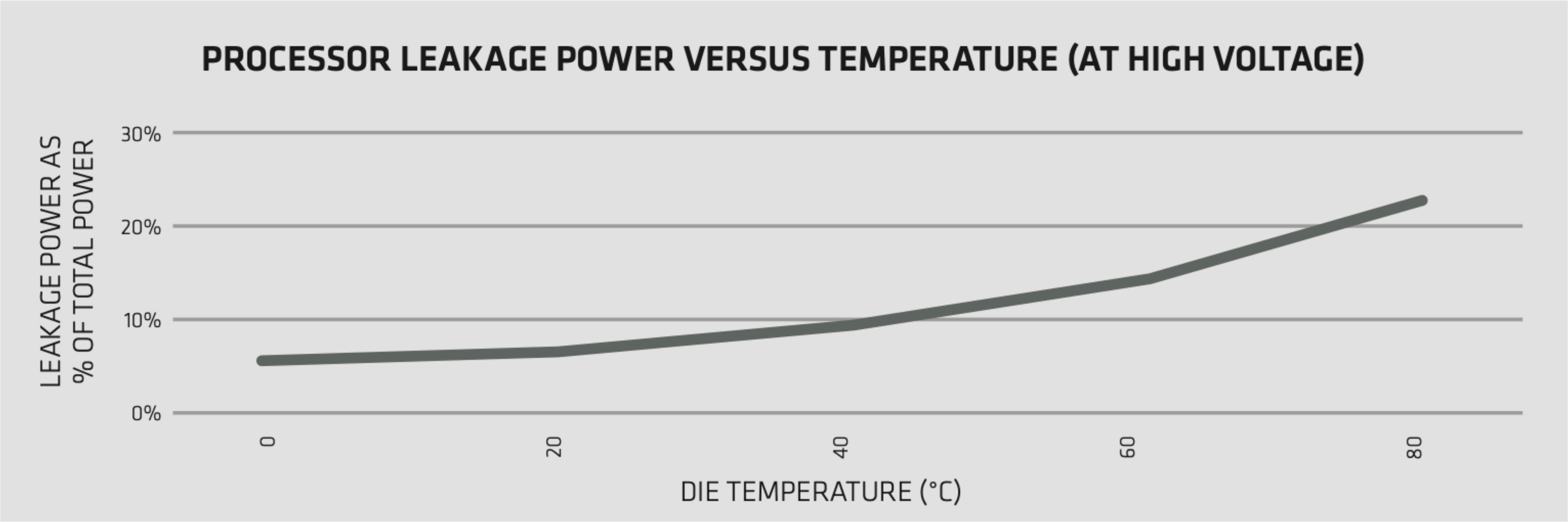
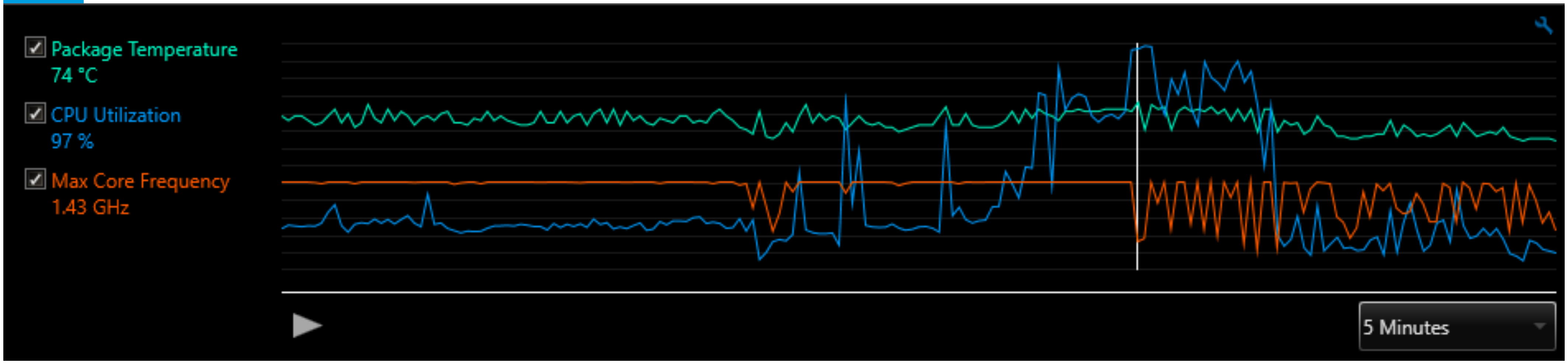


Figure 2 - Leakage power over temperature for a typical sample of an undisclosed AMD product based on a 14nm FinFET process.

- Heat affects transistor operating characteristics

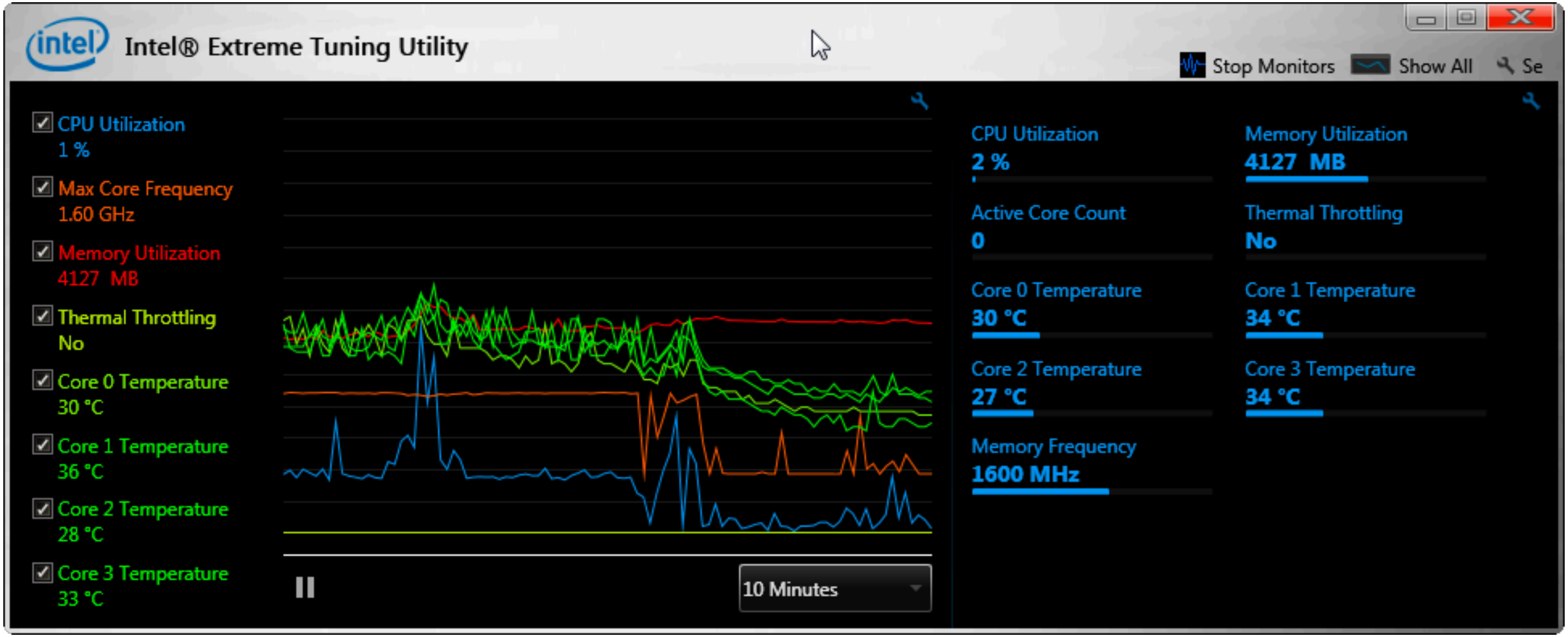


☑ Package Temperature
74 °C

☑ CPU Utilization
97 %

☑ Max Core Frequency
1.43 GHz

5 Minutes



Intel® Extreme Tuning Utility

☑ CPU Utilization
1 %

☑ Max Core Frequency
1.60 GHz

☑ Memory Utilization
4127 MB

☑ Thermal Throttling
No

☑ Core 0 Temperature
30 °C

☑ Core 1 Temperature
36 °C

☑ Core 2 Temperature
28 °C

☑ Core 3 Temperature
33 °C

10 Minutes

CPU Utilization
2 %

Memory Utilization
4127 MB

Active Core Count
0

Thermal Throttling
No

Core 0 Temperature
30 °C

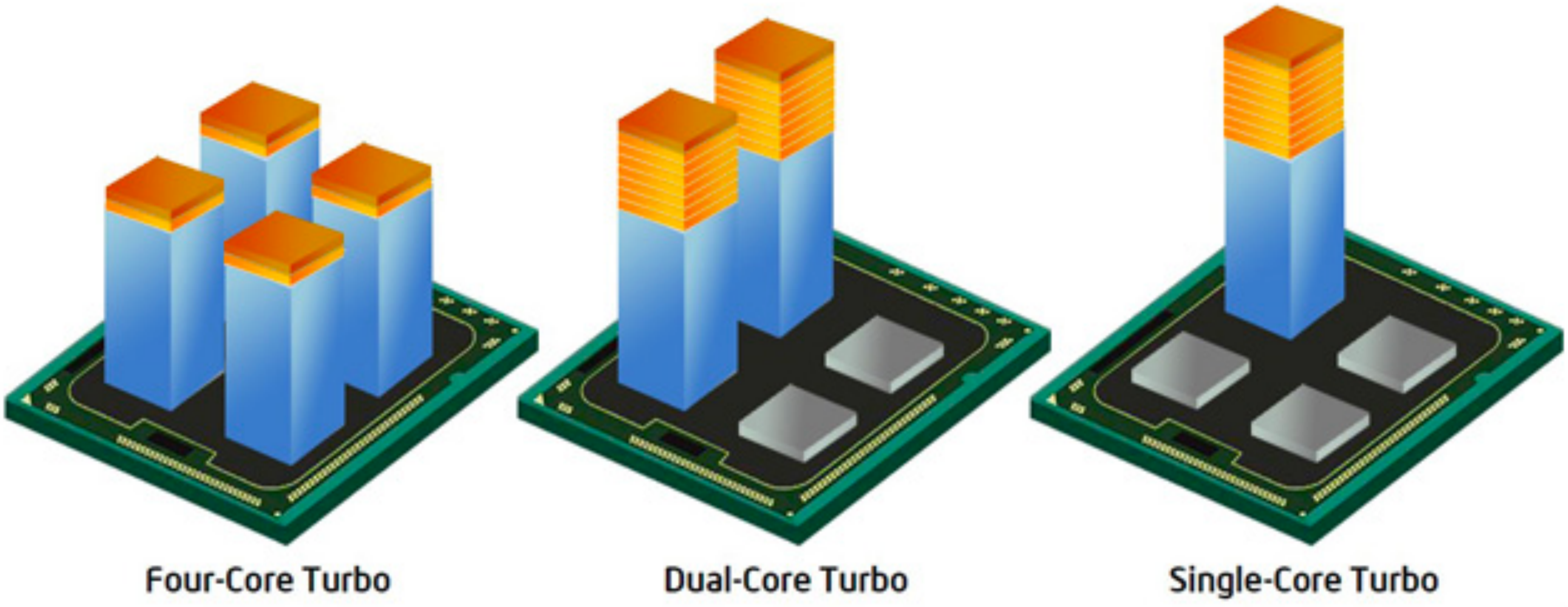
Core 1 Temperature
34 °C

Core 2 Temperature
27 °C

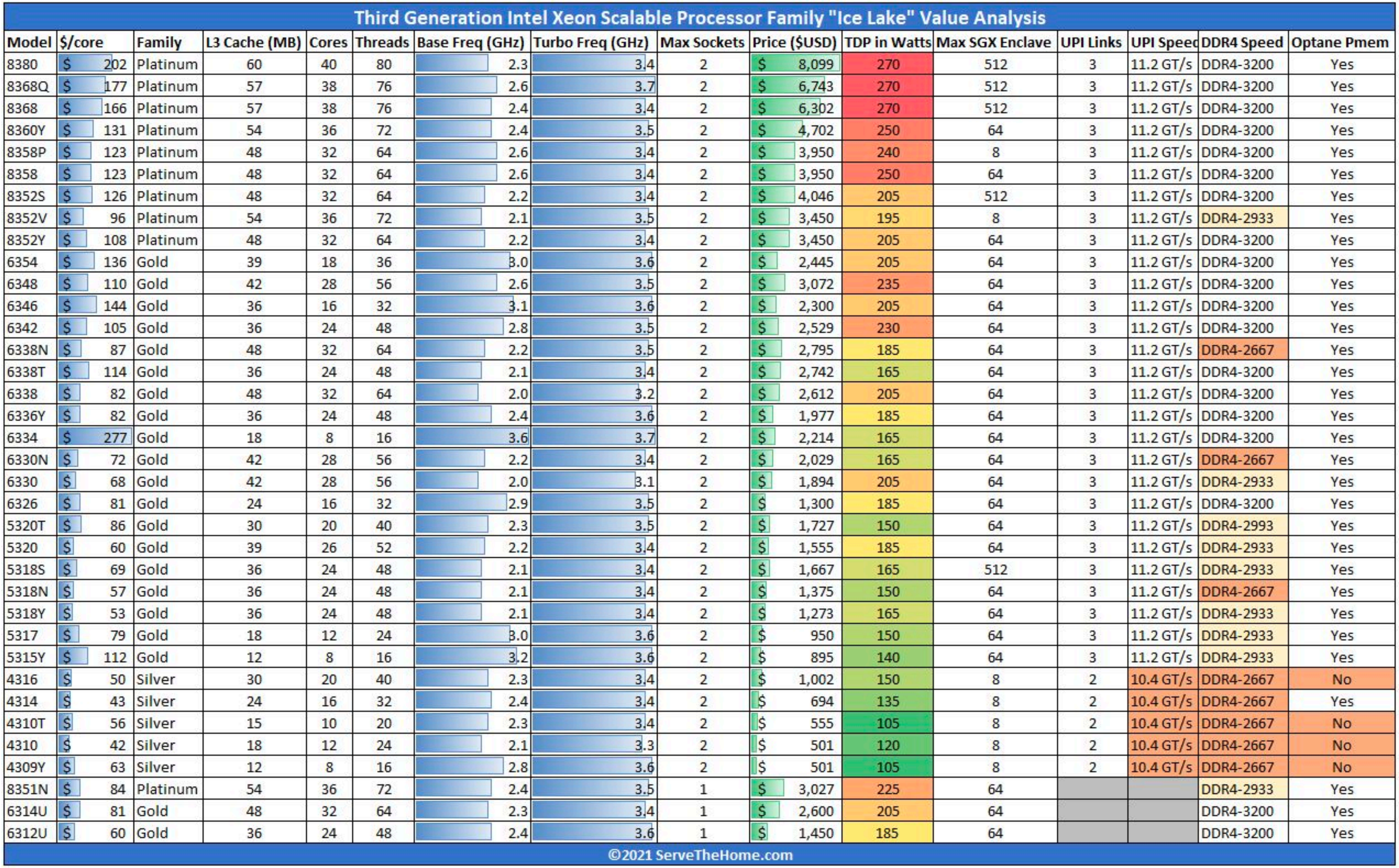
Core 3 Temperature
34 °C

Memory Frequency
1600 MHz

turbo-boost



- allows the power management controller to dynamically provide the best performance (frequency) possible for the specific operating scenario in real-time.



Third Generation Intel Xeon Scalable Processor Family "Ice Lake" Value Analysis

Model	$/core	Family	L3 Cache (MB)	Cores	Threads	Base Freq (GHz)	Turbo Freq (GHz)	Max Sockets	Price ($USD)	TDP in Watts	Max SGX Enclave	UPI Links	UPI Speed	DDR4 Speed	Optane Pmem
8380	$ 202	Platinum	60	40	80	2.3	3.4	2	$ 8,099	270	512	3	11.2 GT/s	DDR4-3200	Yes
8368Q	$ 177	Platinum	57	38	76	2.6	3.7	2	$ 6,743	270	512	3	11.2 GT/s	DDR4-3200	Yes
8368	$ 166	Platinum	57	38	76	2.4	3.4	2	$ 6,302	270	512	3	11.2 GT/s	DDR4-3200	Yes
8360Y	$ 131	Platinum	54	36	72	2.4	3.5	2	$ 4,702	250	64	3	11.2 GT/s	DDR4-3200	Yes
8358P	$ 123	Platinum	48	32	64	2.6	3.4	2	$ 3,950	240	8	3	11.2 GT/s	DDR4-3200	Yes
8358	$ 123	Platinum	48	32	64	2.6	3.4	2	$ 3,950	250	64	3	11.2 GT/s	DDR4-3200	Yes
8352S	$ 126	Platinum	48	32	64	2.2	3.4	2	$ 4,046	205	512	3	11.2 GT/s	DDR4-3200	Yes
8352V	$ 96	Platinum	54	36	72	2.1	3.5	2	$ 3,450	195	8	3	11.2 GT/s	DDR4-2933	Yes
8352Y	$ 108	Platinum	48	32	64	2.2	3.4	2	$ 3,450	205	64	3	11.2 GT/s	DDR4-3200	Yes
6354	$ 136	Gold	39	18	36	3.0	3.6	2	$ 2,445	205	64	3	11.2 GT/s	DDR4-3200	Yes
6348	$ 110	Gold	42	28	56	2.6	3.5	2	$ 3,072	235	64	3	11.2 GT/s	DDR4-3200	Yes
6346	$ 144	Gold	36	16	32	3.1	3.6	2	$ 2,300	205	64	3	11.2 GT/s	DDR4-3200	Yes
6342	$ 105	Gold	36	24	48	2.8	3.5	2	$ 2,529	230	64	3	11.2 GT/s	DDR4-3200	Yes
6338N	$ 87	Gold	48	32	64	2.2	3.5	2	$ 2,795	185	64	3	11.2 GT/s	DDR4-2667	Yes
6338T	$ 114	Gold	36	24	48	2.1	3.4	2	$ 2,742	165	64	3	11.2 GT/s	DDR4-3200	Yes
6338	$ 82	Gold	48	32	64	2.0	3.2	2	$ 2,612	205	64	3	11.2 GT/s	DDR4-3200	Yes
6336Y	$ 82	Gold	36	24	48	2.4	3.6	2	$ 1,977	185	64	3	11.2 GT/s	DDR4-3200	Yes
6334	$ 277	Gold	18	8	16	3.6	3.7	2	$ 2,214	165	64	3	11.2 GT/s	DDR4-3200	Yes
6330N	$ 72	Gold	42	28	56	2.2	3.4	2	$ 2,029	165	64	3	11.2 GT/s	DDR4-2667	Yes
6330	$ 68	Gold	42	28	56	2.0	3.1	2	$ 1,894	205	64	3	11.2 GT/s	DDR4-2933	Yes
6326	$ 81	Gold	24	16	32	2.9	3.5	2	$ 1,300	185	64	3	11.2 GT/s	DDR4-3200	Yes
5320T	$ 86	Gold	30	20	40	2.3	3.5	2	$ 1,727	150	64	3	11.2 GT/s	DDR4-2993	Yes
5320	$ 60	Gold	39	26	52	2.2	3.4	2	$ 1,555	185	64	3	11.2 GT/s	DDR4-2933	Yes
5318S	$ 69	Gold	36	24	48	2.1	3.4	2	$ 1,667	165	512	3	11.2 GT/s	DDR4-2933	Yes
5318N	$ 57	Gold	36	24	48	2.1	3.4	2	$ 1,375	150	64	3	11.2 GT/s	DDR4-2667	Yes
5318Y	$ 53	Gold	36	24	48	2.1	3.4	2	$ 1,273	165	64	3	11.2 GT/s	DDR4-2933	Yes
5317	$ 79	Gold	18	12	24	3.0	3.6	2	$ 950	150	64	3	11.2 GT/s	DDR4-2933	Yes
5315Y	$ 112	Gold	12	8	16	3.2	3.6	2	$ 895	140	64	3	11.2 GT/s	DDR4-2933	Yes
4316	$ 50	Silver	30	20	40	2.3	3.4	2	$ 1,002	150	8	2	10.4 GT/s	DDR4-2667	No
4314	$ 43	Silver	24	16	32	2.4	3.4	2	$ 694	135	8	2	10.4 GT/s	DDR4-2667	Yes
4310T	$ 56	Silver	15	10	20	2.3	3.4	2	$ 555	105	8	2	10.4 GT/s	DDR4-2667	No
4310	$ 42	Silver	18	12	24	2.1	3.3	2	$ 501	120	8	2	10.4 GT/s	DDR4-2667	No
4309Y	$ 63	Silver	12	8	16	2.8	3.6	2	$ 501	105	8	2	10.4 GT/s	DDR4-2667	No
8351N	$ 84	Platinum	54	36	72	2.4	3.5	1	$ 3,027	225	64			DDR4-2933	Yes
6314U	$ 81	Gold	48	32	64	2.3	3.4	1	$ 2,600	205	64			DDR4-3200	Yes
6312U	$ 60	Gold	36	24	48	2.4	3.6	1	$ 1,450	185	64			DDR4-3200	Yes

©2021 ServeTheHome.com

TU Delft

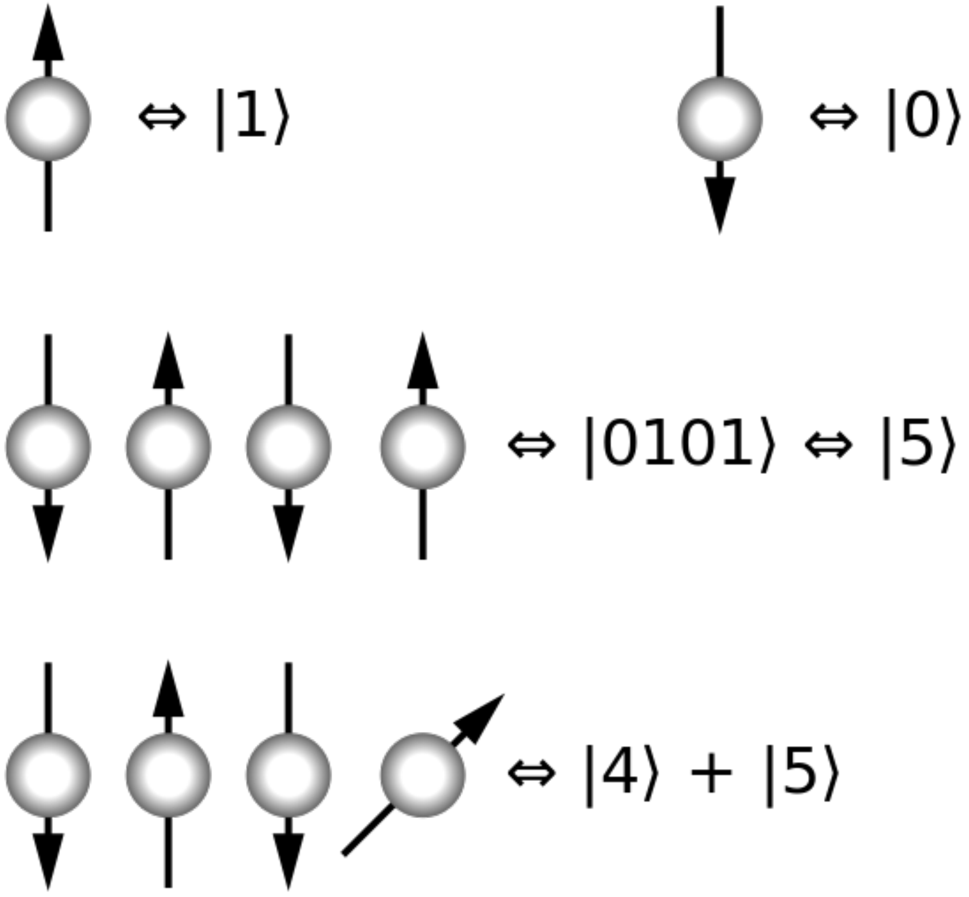
Questions

- Is Multicore really the answer?
  - FPGAs? Quantum computing?
  - What else might be waiting in the wings

- What about advances in circuit fabrication?
  - alternatives to Si: SOI, Hafnium doping, plastics
  - optical wires, photonic communication
  - superconducting

- What about memory?
  - Its starting to consume more space than CPU cores!
  - Packaging changes (3D Stacking? Optical Interfaces?)

Quantum computing



$\Leftrightarrow |1\rangle$ $\Leftrightarrow |0\rangle$

$\Leftrightarrow |0101\rangle \Leftrightarrow |5\rangle$

$\Leftrightarrow |4\rangle + |5\rangle$

qubits can be in a superposition of all the
clasically allowed states

Quantum computing

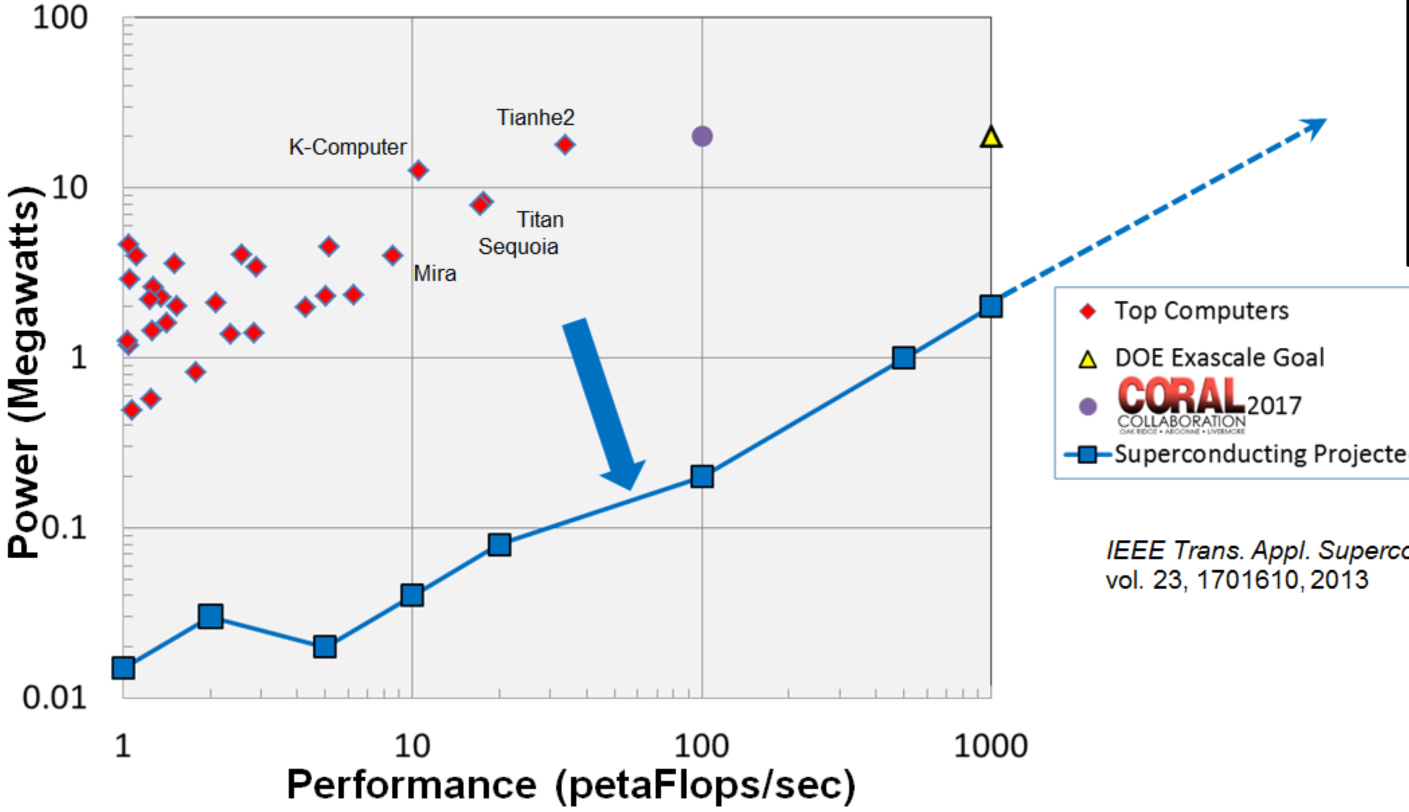
- Claims to solve NP-Complete Problems
  - traveling salesman problem
  - Graph Coloring Problem: can you color a graph using k ≥ 3 colors such that no adjacent vertices have the same color?
  - one algorthm can solve all NP-complete problems

- NP-Complete Problems
  - solution is easy to verify
  - number of compute steps grows exponentially with problem size

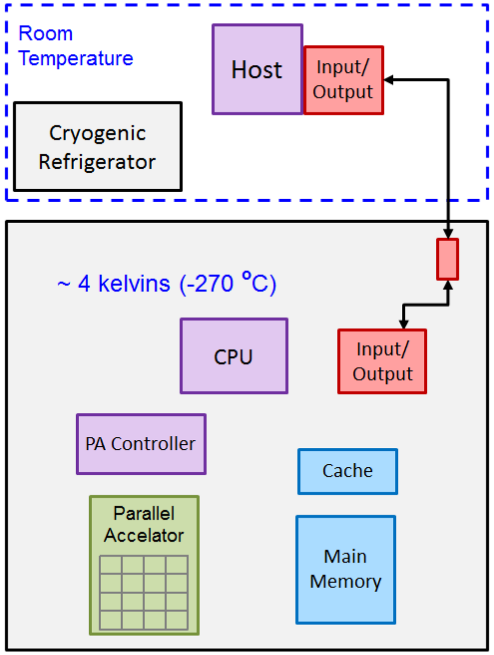
- Quantum computing also leads to a better understanding of quantum physics.

Superconducting

- Cryogenic Computer Complexity



IEEE Trans. Appl. Supercond., vol. 23, 1701610, 2013

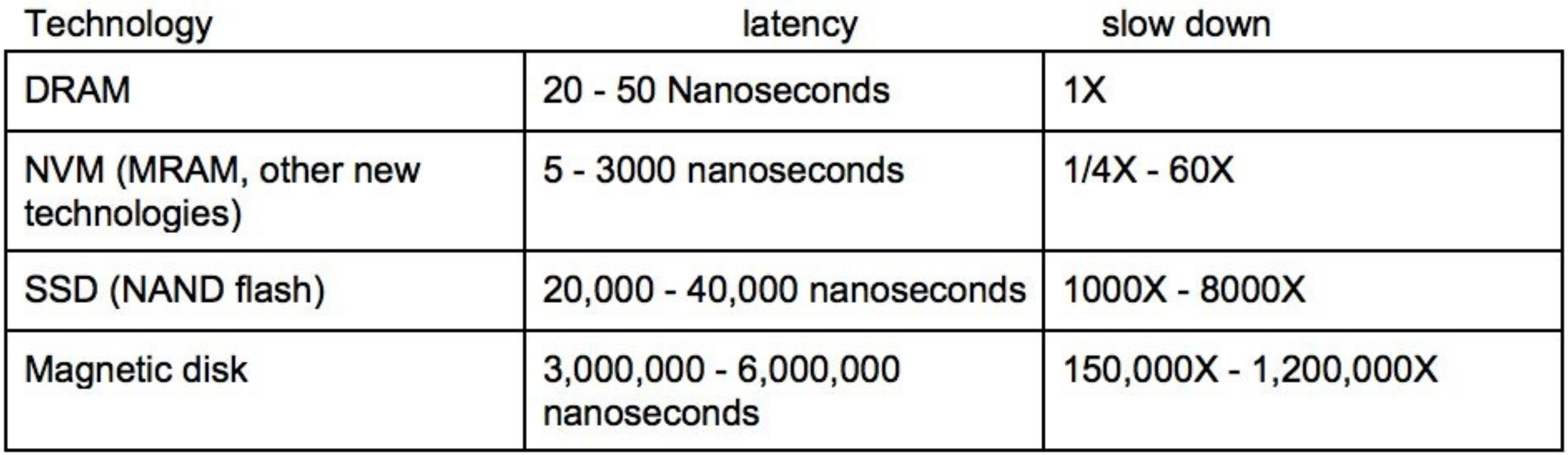


The hardest is to develop high-density, high-efficiency, low-latency, cryogenic memory.

Memory

- Extend Hierarchy with another layer between DRAM and HardDisk
    - SSD/FLASH layer

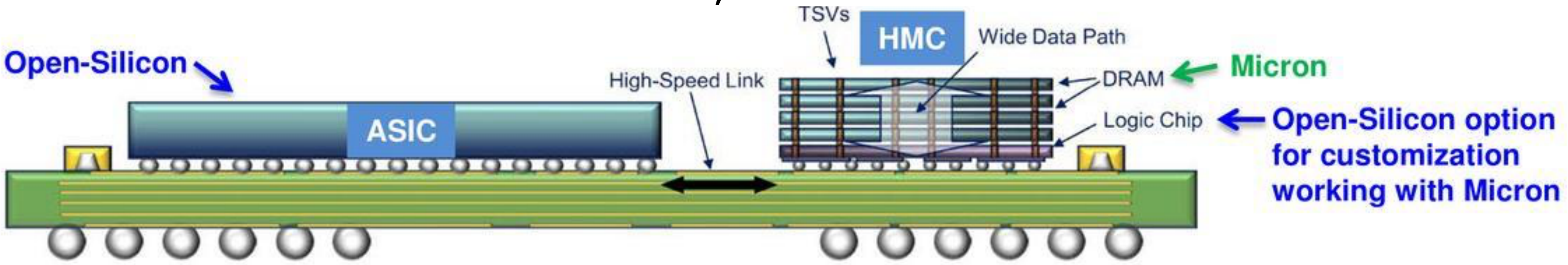
- Extend / Replace DRAM to non-volatile memory



Technology	latency	slow down
DRAM	20 - 50 Nanoseconds	1X
NVM (MRAM, other new technologies)	5 - 3000 nanoseconds	1/4X - 60X
SSD (NAND flash)	20,000 - 40,000 nanoseconds	1000X - 8000X
Magnetic disk	3,000,000 - 6,000,000 nanoseconds	150,000X - 1,200,000X

Developments in Memory

- 3D packing of memory elements
  - Hybrid Memory Cube (HMC)
    HMC uses standard DRAM cells, but it has more data banks.



  - High Bandwidth Memory (HBM)
    HMB2 DRAM chips are 8Gb each, and they can be stacked up to 8 high, yielding an 8GB 256GB/s lan

Complexity – Memory

2015

L1/L2 cache	~1 ns
L3 cache	~10 ns
Main memory	~100 ns / ~80 GB/s / ~100GB
NAND SSD	~100 usec / ~10 GB/s / ~1 TB
Fast HHD	~10 msec / ~100 MB/s / ~10 TB

2020

L1/L2 cache	~1 ns
L3 cache	~10 ns
HBM	~10 ns / ~1TB/s / ~10GB
Main memory	~100 ns / ~80 GB/s / ~100GB
NVM (3D Xpoint)	~1 usec / ~10GB/s / ~1TB
NAND SSD	~100 usec / ~10 GB/s / ~10 TB
Fast HHD	~10 msec / ~100 MB/s / ~100 TB

HPE The machine

- Memristors
  - fuse memory and storage,
  - flatten complex data hierarchies,
  - bring processing closer to the data,
  - embed security control points throughout the hardware and software stacks
  - crapping the distinction between storage and memory.

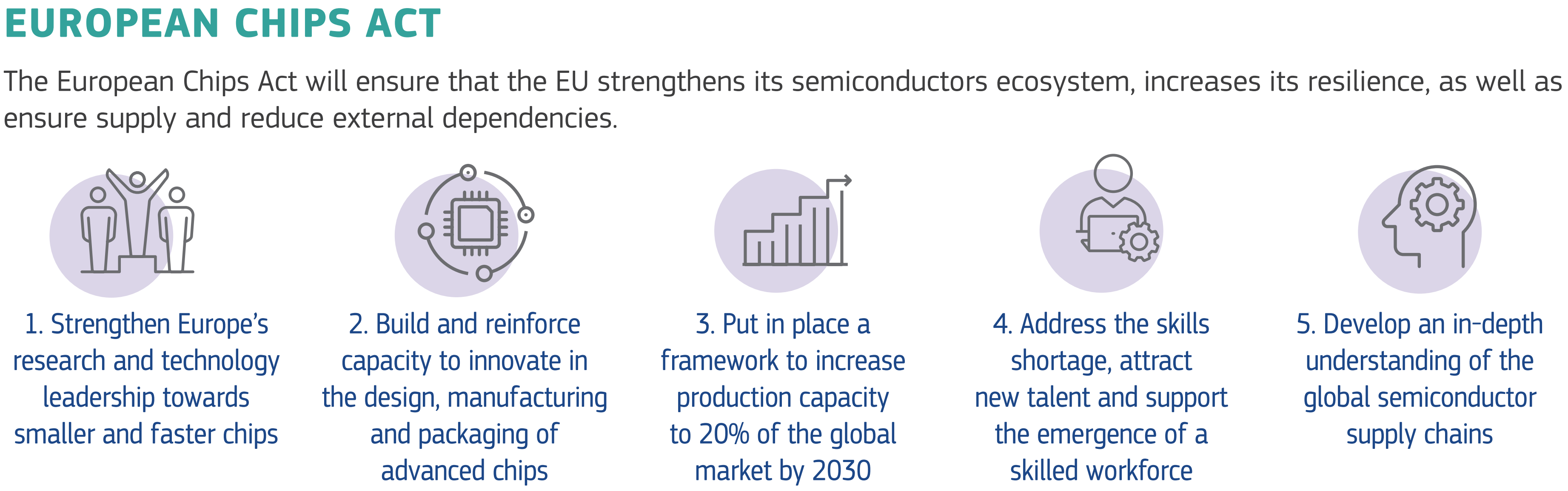
A single large store of memory based on HP’s memristors will both hold data and make it available for the processor.

A Dream not come true.

What about Europe?

European Chips Act (8-2-2022)

- €43 billion euros of public and private investments
- Become a leader in this field beyond research and technology in design, manufacturing and packaging of advanced chips, to secure its supply of semiconductors and to reduce its dependencies.

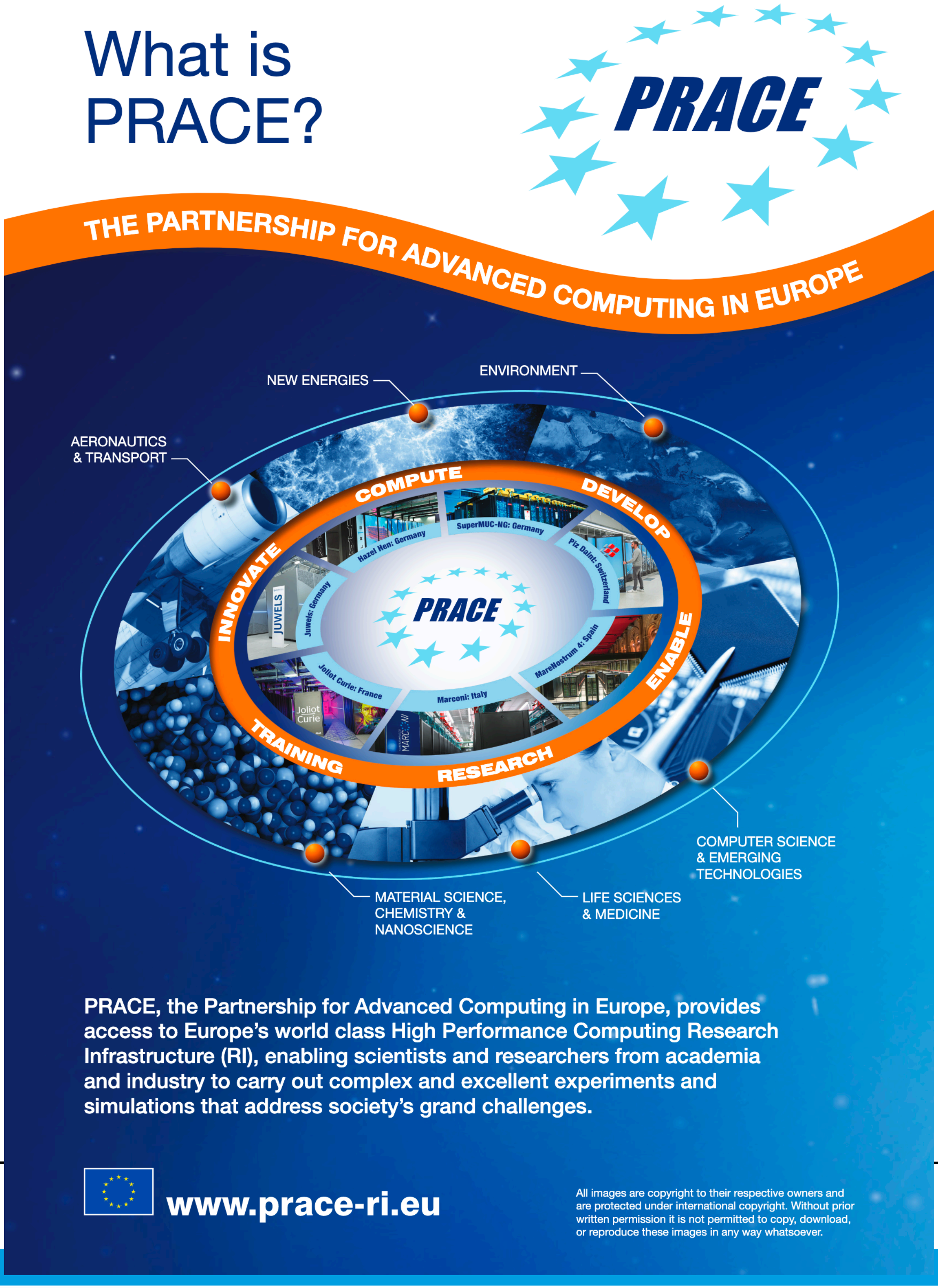


EUROPEAN CHIPS ACT

The European Chips Act will ensure that the EU strengthens its semiconductors ecosystem, increases its resilience, as well as ensure supply and reduce external dependencies.

1. Strengthen Europe's research and technology leadership towards smaller and faster chips
2. Build and reinforce capacity to innovate in the design, manufacturing and packaging of advanced chips
3. Put in place a framework to increase production capacity to 20% of the global market by 2030
4. Address the skills shortage, attract new talent and support the emergence of a skilled workforce
5. Develop an in-depth understanding of the global semiconductor supply chains

PRACE



What is PRACE?

PRACE, the Partnership for Advanced Computing in Europe, provides access to Europe's world class High Performance Computing Research Infrastructure (RI), enabling scientists and researchers from academia and industry to carry out complex and excellent experiments and simulations that address society's grand challenges.

www.prace-ri.eu

Advise for programmers

Get ready for multi-core

to be continued...

IO interface and hardware

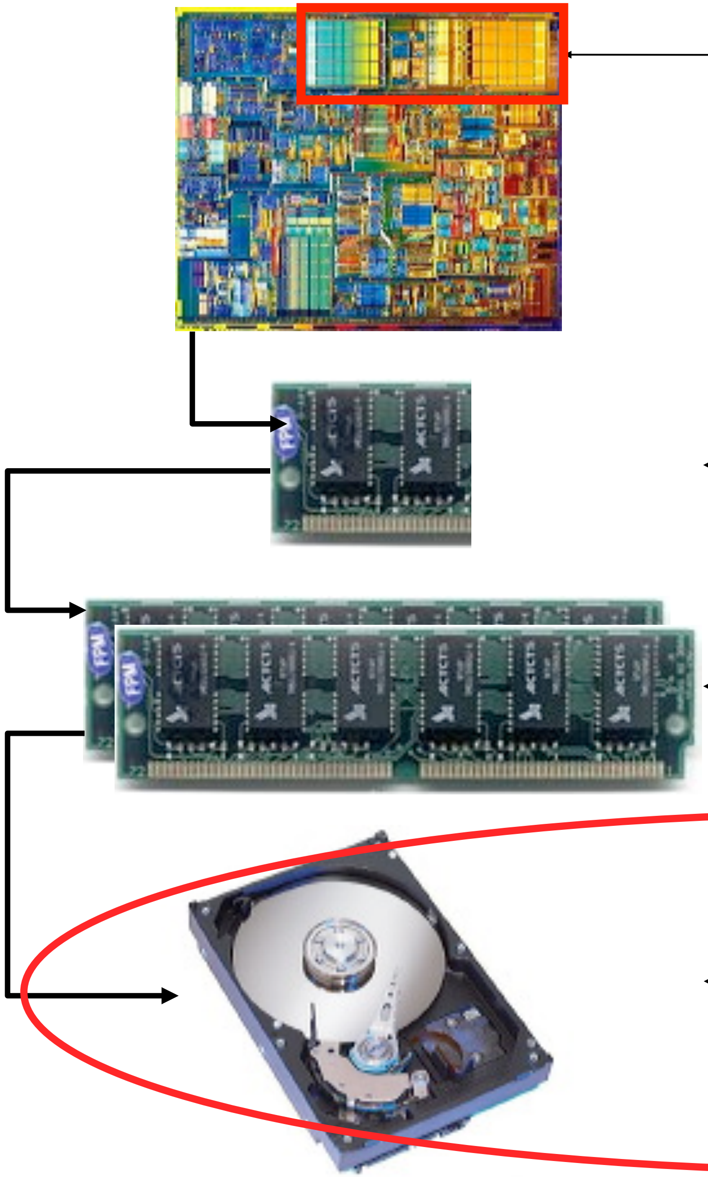
IO Interfaces

- CPU interface and interaction with IO storage device(s)
  - SATA (Serial ATA)
  - SCSI (Serial Attached S)
  - PCIe, NVMe

- Hard drive
  - magnetic disks
  - SSD

ATA: 16 wires of data in parallel
SATA: serial transport line through 2 wires

Hard Disks



	Cost	Access time
on-chip cache KBs		< 1ns
off-chip cache MBs	2.5 $/MB	5ns
main memory GBs	0.07 $/MB	60ns
Disk TBs	0.000008 $/MB	10,000,000ns

- Magnetic: rotating disk slow access
  - 125 MB/s max
  - cheap $0.04/GB
- Solid State Disks (SSD)
  - 250 MB/s read
  - Fast in (random) IO operations per seconds
  - expensive $0.5/GB

Exercise: 1 Cycles

- Counting cycles of basic operations; addition, multiplication, division, ...

- On your git clone: cd HPCourse/Cycles

- Check the README for instructions.

- Links: http://en.wikipedia.org/wiki/Streaming_SIMD_Extensions

Exercise 2: Memory hierarchy

- Measuring the memory bandwidth of your computer.
- On your git clone: cd HPCourse/LoadStore
- Check the README for instructions.
- The program produces an ASCII output file which contains the result.
- Results can be plotted with gnuplot, (set style data linespoints)
- Sent interesting results (ASCII files) to janth@xs4all.nl

Exercise 3: Memory latency

- Measuring the latency of memory hierarchy
- On your git clone: cd HPCourse/lat_mem_rd
- Check the README for instructions.
- The program produces an ASCII output file which contains the result: Mbytes, nanoseconds, cycles
- Sent interesting results (ASCII files) to janth@xs4all.nl
- additional information: http://www.bitmover.com/lmbench/lat_mem_rd.8.html