

# Computers

# Computer Structure

*Grau en Ciència i Enginyeria de Dades*

---

**Xavier Martorell, Xavier Verdú**

Facultat d'Informàtica de Barcelona (FIB)

Universitat Politècnica de Catalunya (UPC)

2019-2020 Q2

# Creative Commons License

---

This work is under a Creative Commons Attribution 4.0 Unported License



The details of this license are publicly available at  
<https://creativecommons.org/licenses/by-nc-nd/4.0>

# Table of Contents

---

- Computer structure. Components
  - Processor – The chip
    - CPU / core
    - Multi- and many-core
    - Simultaneous multithreading / hyperthreading
  - Memory – and memory hierarchy
    - Data / instruction caches
    - Main memory

# Table of Contents (cont.)

---

- Computer structure. Components
  - Input/Output components
    - Buses. Peripherals
  - Storage and file systems
    - Disks
  - Networking
    - Connexions and protocols

# Computation - arithmetic

- From Session 1 – Data Representation

**Addition:**

$$\begin{array}{r}
 10110010 \\
 + 01011001 \\
 \hline
 100001011
 \end{array}$$

Carry bit 1

$128+32+16+2 = 178$   
 $64+16+8+1 = 89$   
 $256 + 8 + 2 + 1 = 267$   
 $8 + 2 + 1 = 11$

**Subtraction:**

$$\begin{array}{r}
 10110010 \\
 - 01011001 \\
 \hline
 01011001
 \end{array}$$

Carry bit 0

$128+32+16+2 = 178$   
 $64+16+8+1 = 89$   
 $64 + 16 + 8 + 1 = 89$

**AND:**

$$\begin{array}{r}
 10110010 \\
 \& 01011001 \\
 \hline
 00010000
 \end{array}$$

Carry bit 0

$128+32+16+2 = 178$   
 $64+16+8+1 = 89$   
 $16$

**OR:**

$$\begin{array}{r}
 10110010 \\
 \vee 01011001 \\
 \hline
 11111011
 \end{array}$$

Carry bit 0

$128+32+16+2 = 178$   
 $64+16+8+1 = 89$   
 $128+64+32+16+8+2+1 = 251$

**XOR:**

$$\begin{array}{r}
 10110010 \\
 \oplus 01011001 \\
 \hline
 11101011
 \end{array}$$

Carry bit 0

$128+32+16+2 = 178$   
 $64+16+8+1 = 89$   
 $128+64+32+8+2+1 = 235$   
 $128+32+16+2 = 178$   
 $64+8+4+1 = 77$

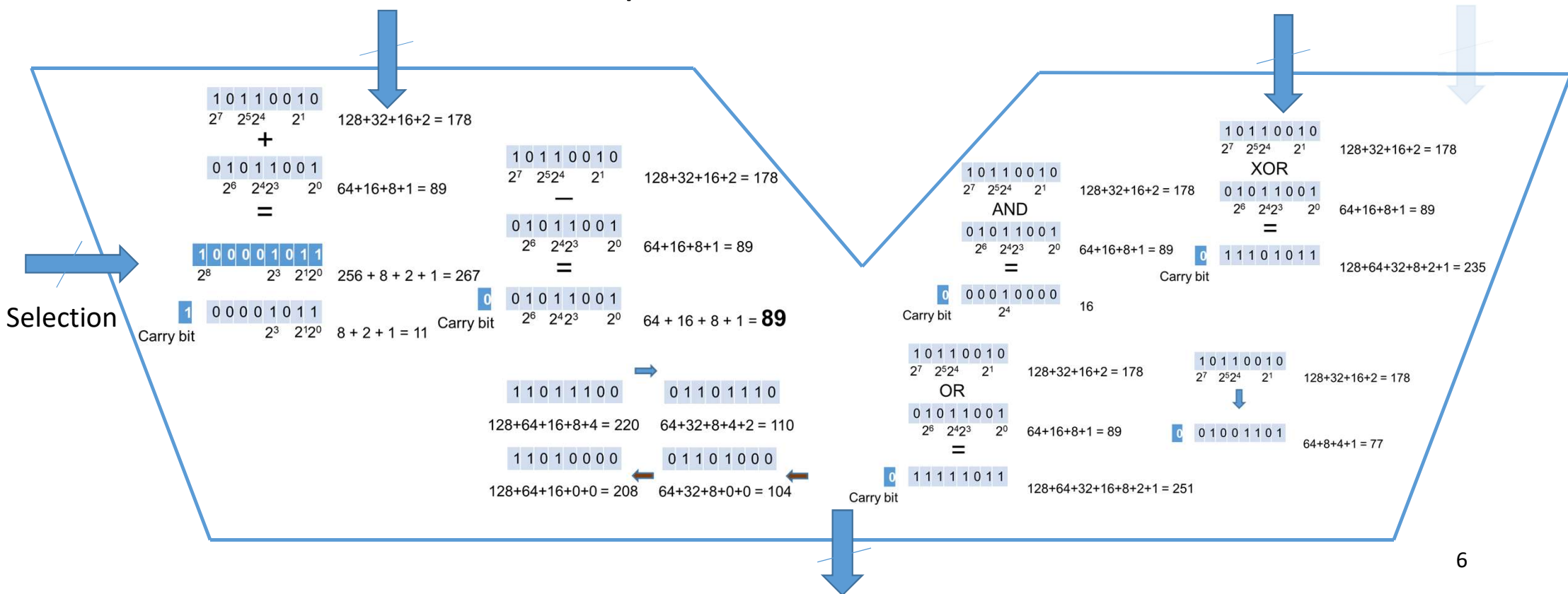
**Shifts:**

Right shift:  $11011100 \rightarrow 01101110$   
 $128+64+16+8+4 = 220$      $64+32+8+4+2 = 110$

Left shift:  $11010000 \leftarrow 01101000$   
 $128+64+16+0+0 = 208$      $64+32+8+0+0 = 104$

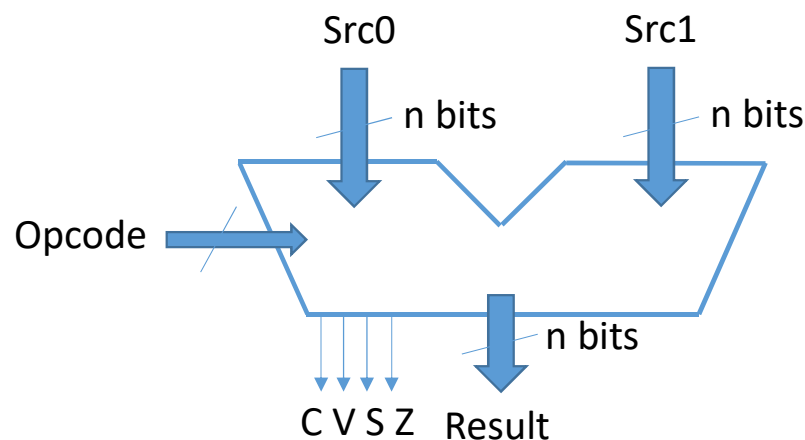
# Computation - arithmetic

- From Session 1 – Data Representation



# Grouping operations together

- Arithmetic and Logic Unit – ALU



- Comparisons are implemented with the subtraction and looking at the flag bits

[https://en.wikipedia.org/wiki/Truth\\_table](https://en.wikipedia.org/wiki/Truth_table)

Opcode				Operation
0	0	0	0	Src0 + Src1
0	0	0	1	Src0 – Src1
0	0	1	0	Src0 * Src1
0	0	1	1	Src0 / Src1
0	1	0	0	Shift left (Src0) by Src1
0	1	0	1	Shift right (Src0) by Src1
0	1	1	0	Rotate left (Src0) by Src1
0	1	1	1	Rotate right (Src0) by Src1
1	0	0	0	Src0 AND Src1
1	0	0	1	Src0 OR Src1
1	0	1	0	Src0 XOR Src1
1	0	1	1	NOT(Src0)
1	1	0	0	NOT(Src1)
1	1	x	x	Reserved for future use

# Computation – sequence of operations

---

- How can we implement a sequence of...
  - Provide operands and operator selection
  - Compute
  - Save the result [somewhere]
  - Read operands [from somewhere]
  - Provide operands and operator selection
  - Compute
  - Save the result [somewhere]
  - ...



# Computation – programs

- Compute the sum of two vectors
  - Vectors = data; data is stored in memory

```
#pragma omp parallel for
for (i=0; i < N; i++) {
    c[i] = a[i] + b[i];
}
```

```
move #0, i
while (i < N) {
    load r1, a[i]
    load r2, b[i]
    add r1, r2, r3
    store r3, c[i]
    i++
}
```

```
move #0, r8
while (r8 < N) {
    load r1, a[r8]
    load r2, b[r8]
    add r1, r2, r3
    store r3, c[r8]
    add #1, r8
}
```

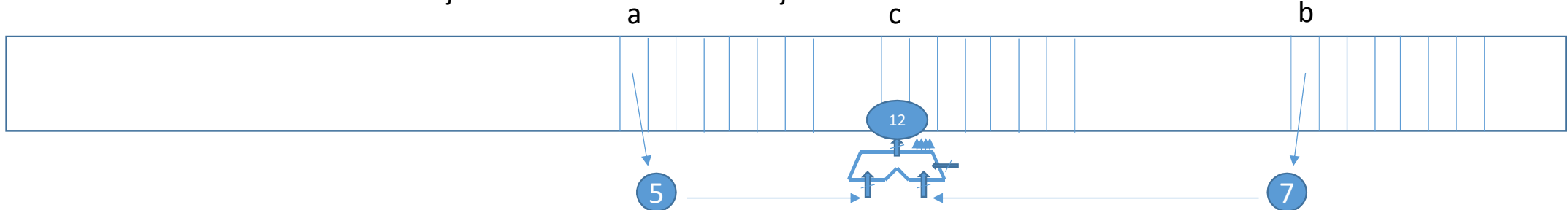
```
move #a, r16
move #b, r17
move #c, r18
move #0, r8
```

loop:

```
cmp #N, r8
ble endloop
load r1, r16[r8]
load r2, r17[r8]
add r1, r2, r3
store r3, r18[r8]
add #1, r8
bra loop
```

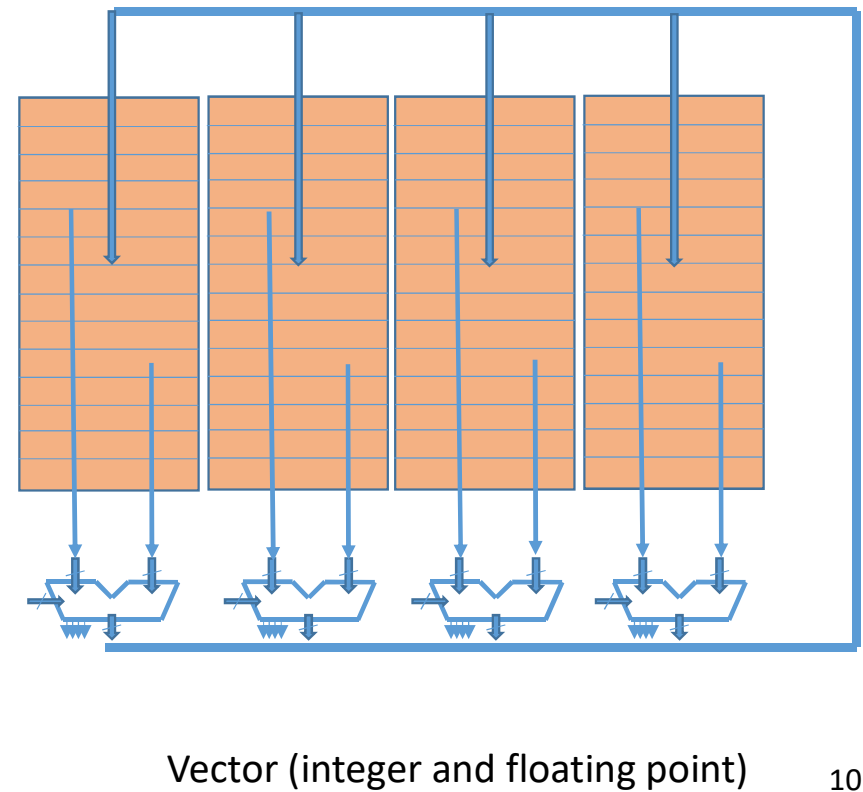
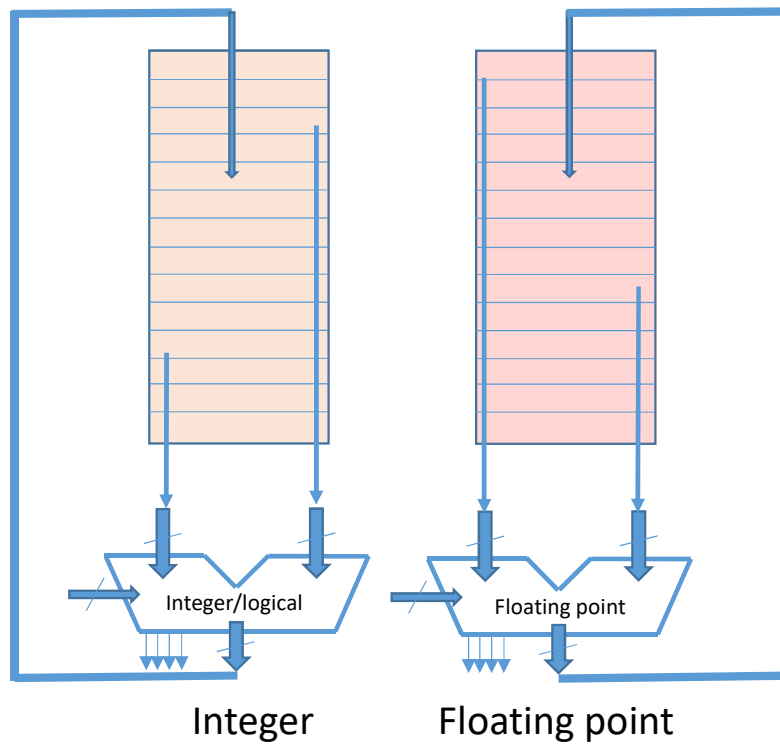
endloop:

...  
b



# Data operations

- ALUs and data **registers** banks



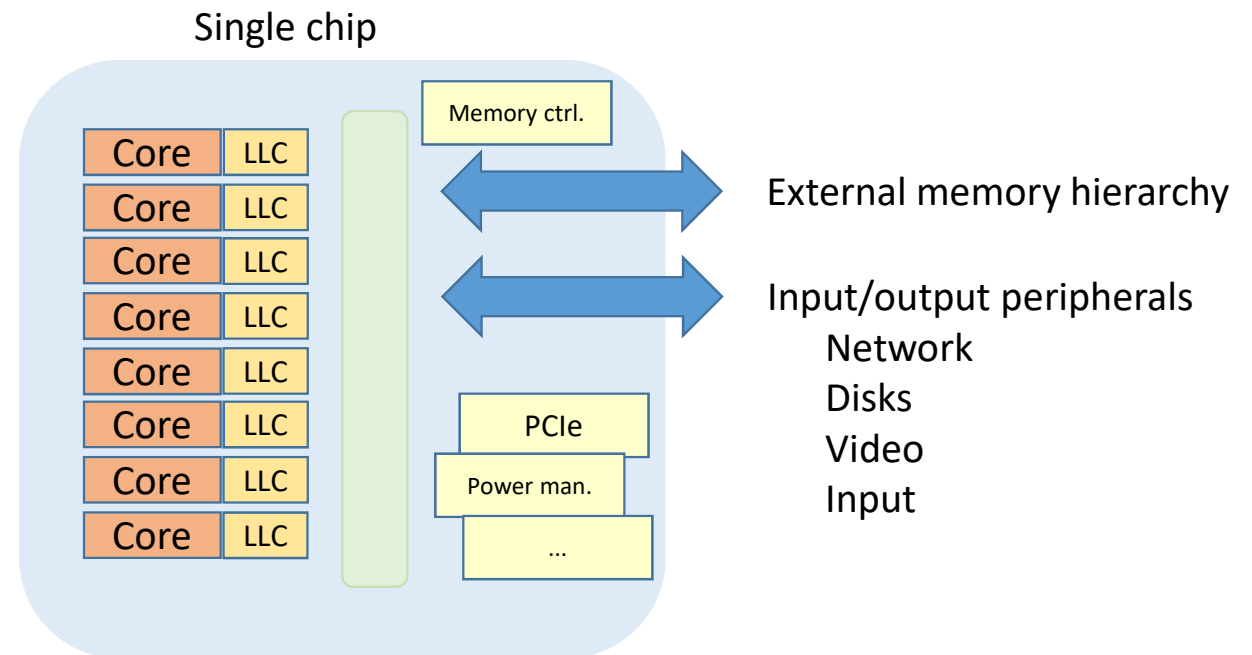
# Table of Contents

---

- Binary arithmetic and logic operations
- **Computer structure. Components**
  - Processor – The chip
    - CPU / core
    - Multi- and many-core
    - Simultaneous multithreading / hyperthreading
  - Memory – and memory hierarchy
    - Data / instruction caches
    - Main memory

# Processors

- M Chips
  - N cores/chip
  - T threads/core
- LLC – last level cache memory



# Processors

---

- What do we need?
  - A program – sequence of instructions
    - Or multiple sequences... iif concurrent/parallel
  - Data – operands should reach the instructions
- **Exercise**... where should we store instructions and data?
- **Exercise**... how do we generate executable programs?

# Hardware Thread

- Each hardware thread independently...

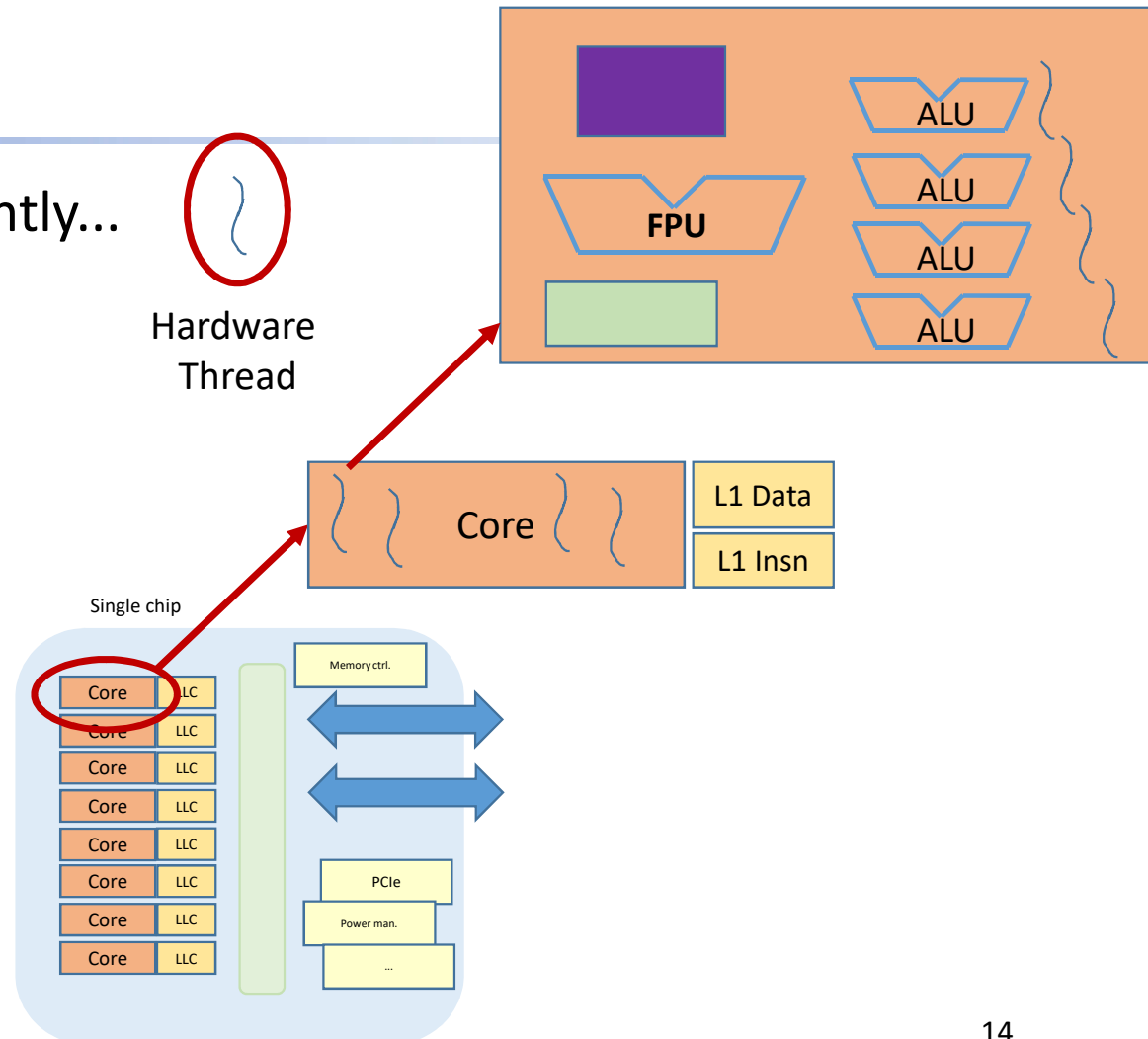
- Fetches instructions\*
- Decodes
- Issues load memory accesses\*
- Executes\*
- Stores results\*

\* When executing a single thread per core, then such a thread has all core resources available!

- Memory bandwidth
- Functional units

- Multithreading

- Execute multiple threads in parallel




# Software Thread

---

- The instruction flow of a given running program. Any program has at least one thread.

- Single-Threaded

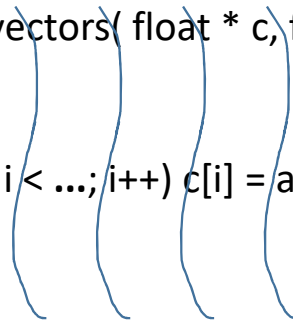
```
void add_vectors( float * c, float * a, float * b )  
{  
    int i;  
    for (i=0; i < N; i++) c[i] = a[i] + b[i];  
}
```



*The thread executes  
from 0 to N*

- Multi-Threaded: execute multiple threads in parallel or concurrently

```
void add_vectors( float * c, float * a, float * b )  
{  
    int i;  
    for (i=...; i < ...; i++) c[i] = a[i] + b[i];  
}
```

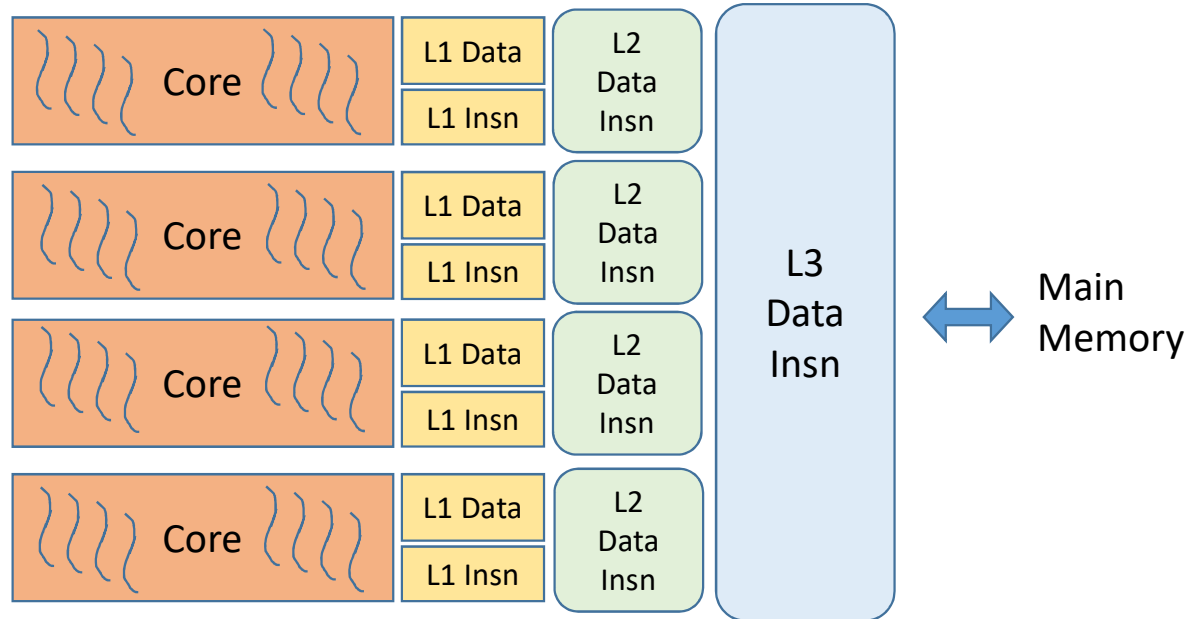


*Every thread executes  
N/4 iterations of the loop*

# Memory hierarchy

- Instruction and data caches

- Level 1 – L1
- Level 2 – L2
- Level 3 – L3 --- LLC (usually)



## Examples

AMD

Private L1  
Shared L2  
Shared L3

Intel

Private L1  
Private L2  
Shared L3



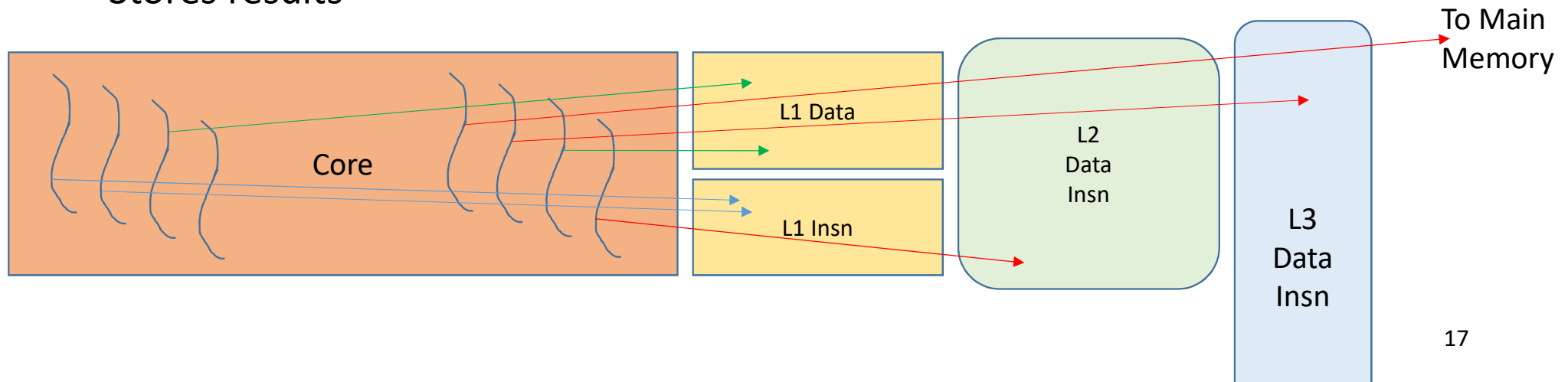
# Hardware multithreading

- Each hardware thread independently...

- Fetches instructions\*
- Decodes
- Issues load memory accesses\*
- Executes\*
- Stores results\*

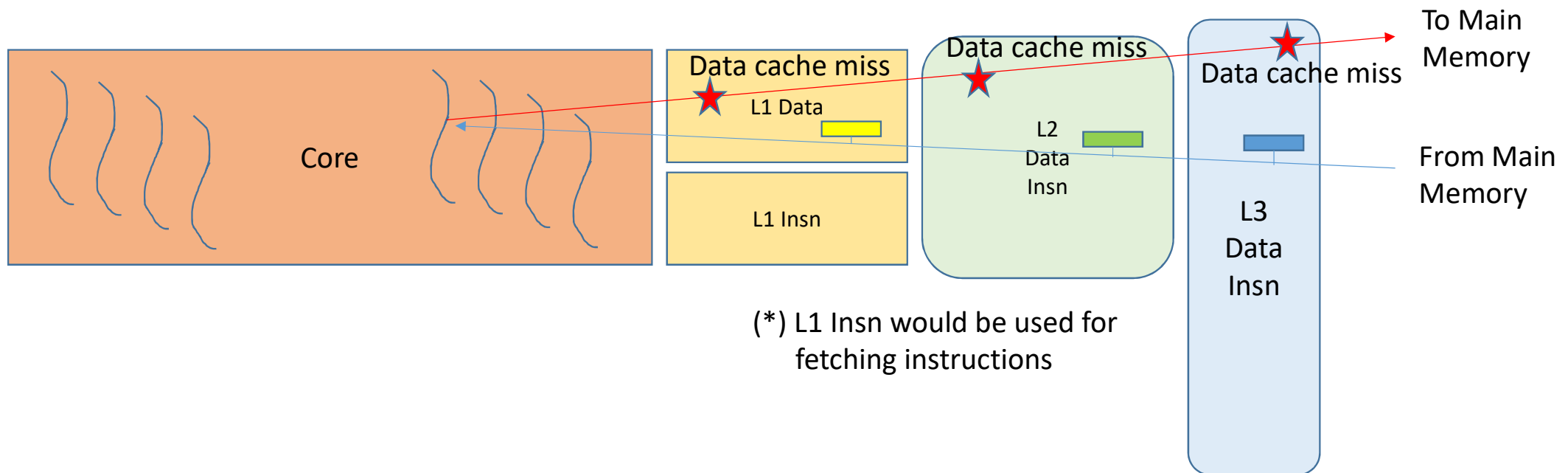
\* When executing a single thread per core, then such a thread has all core resources available!

- Memory bandwidth
- Functional units



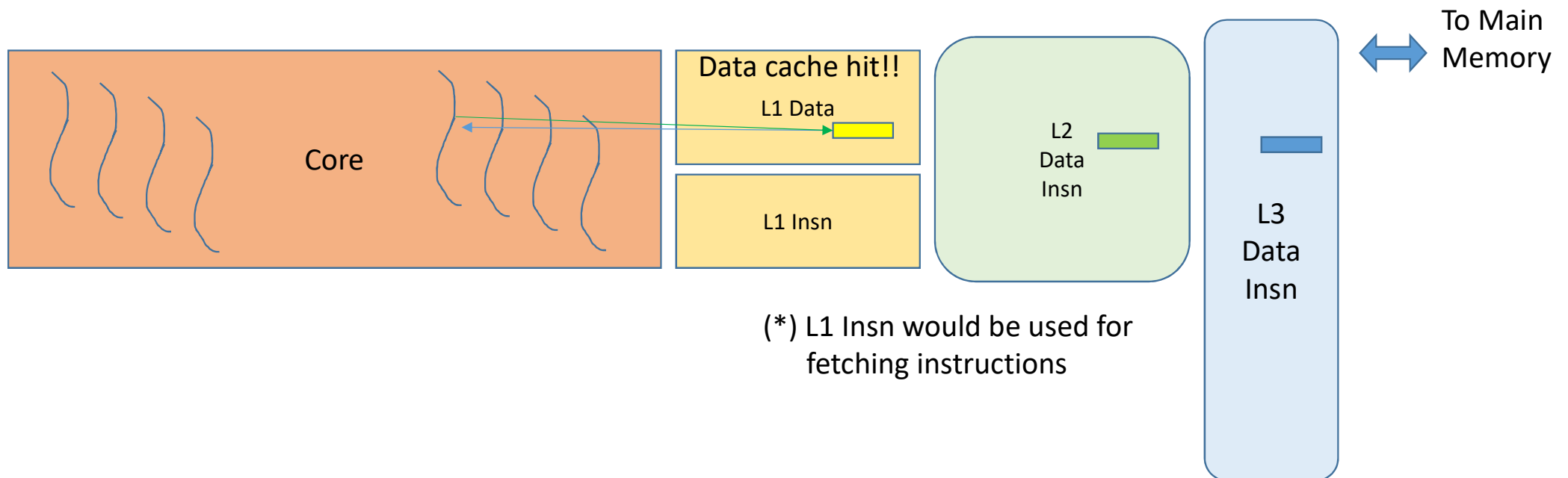
# Detailed memory access

- Load instruction, data is not on the caches
  - Also, fetching instructions, instructions are not on the caches(\*)



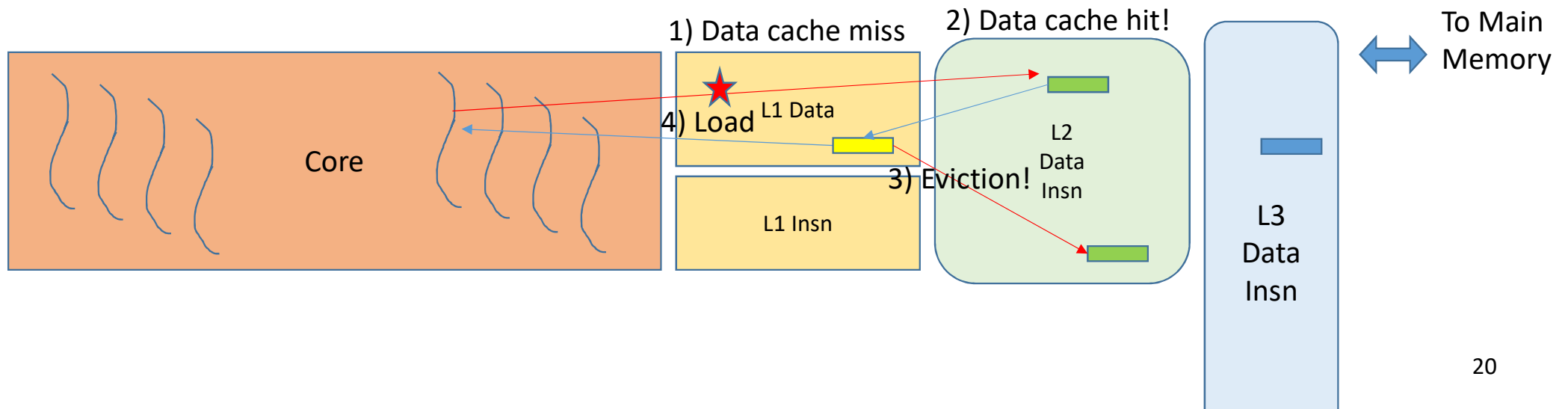
# Detailed memory access

- Load instruction, data present in L1 Data
  - Also, fetching instructions, instruction present in L1 Insn (\*)



# Detailed memory access

- Cache management is a complex hardware feature
  - What happens when the cache is already full of data... and the core needs to bring more?
    - Cache eviction... Last recently used data may be evicted to the next cache level



# Sample code

---

- Computing on vectors  $a$ ,  $b$ , and  $c$
- Accesses reference main memory locations, not cache locations
  - Cache memories are transparently managed by the hardware
  - **Memory coherency**: any read from any processor to a particular memory @, returns the most recently written value to that @
  - **Memory consistency**: ensure writes to different memory @ will be seen in the correct order from all processors

```
void add_vectors( float * c, float * a, float * b )  
{  
    int i;  
    for (i=0; i < N; i++) c[i] = a[i] + b[i];  
}
```

```
void mult_vectors( float * c, float * a, float * b )  
{  
    int i;  
    for (i=0; i < N; i++) c[i] = a[i] * b[i];  
}
```

# Code generation details

```

_add_vectors:
    subq    $8, %rsp
    cmp1    $0, _N(%rip)
    jle     L6
    movl    $0, %eax

L5:
    movslq  %eax,%r9
    salq    $2, %r9
    movss   (%rdx,%r9), %xmm0
    addss   (%r8,%r9), %xmm0
    movss   %xmm0, (%rcx,%r9)
    addl    $1, %eax
    cmp1    %eax, _N(%rip)
    jg      L5

L6:
    addq    $8, %rsp
    ret
  
```

```

        _mult_vectors:
        prologue/
        entering function

        init index

        load a
        add/mul a, b
        store c
        inc index
        compare index to N

        epilogue/
        leaving function
  
```

```

    subq    $8, %rsp
    cmp1    $0, _N(%rip)
    jle     L11
    movl    $0, %eax

L10:
    movslq  %eax,%r9
    salq    $2, %r9
    movss   (%rdx,%r9), %xmm0
    mulss   (%r8,%r9), %xmm0
    movss   %xmm0, (%rcx,%r9)
    addl    $1, %eax
    cmp1    %eax, _N(%rip)
    jg      L10

L11:
    addq    $8, %rsp
    ret
  
```

# Code execution details

\_add\_vectors:

```
subq    $8, %rsp
cmpl    $0, _N(%rip)
jle     L6
movl    $0, %eax
```

L5:

```
movsq   %eax,%r9
salq    $2, %r9
movss   (%rdx,%r9), %xmm0
addss   (%r8,%r9), %xmm0
movss   %xmm0, (%rcx,%r9)
addl    $1, %eax
cmpl    %eax, _N(%rip)
jg      L5
```

L6:

```
addq    $8, %rsp
ret
```

**init index**

**load a**

**add a, b**

**store c**

**inc index**

**compare index to N**

Processors / threads execute on a cycle by cycle basis  
1.x – 2.x instructions per cycle

immediate loads may take 1-10 cycles

loads may take 1 – 200 cycles (L1 ... Main mem)

stores may be less costly... Store buffer

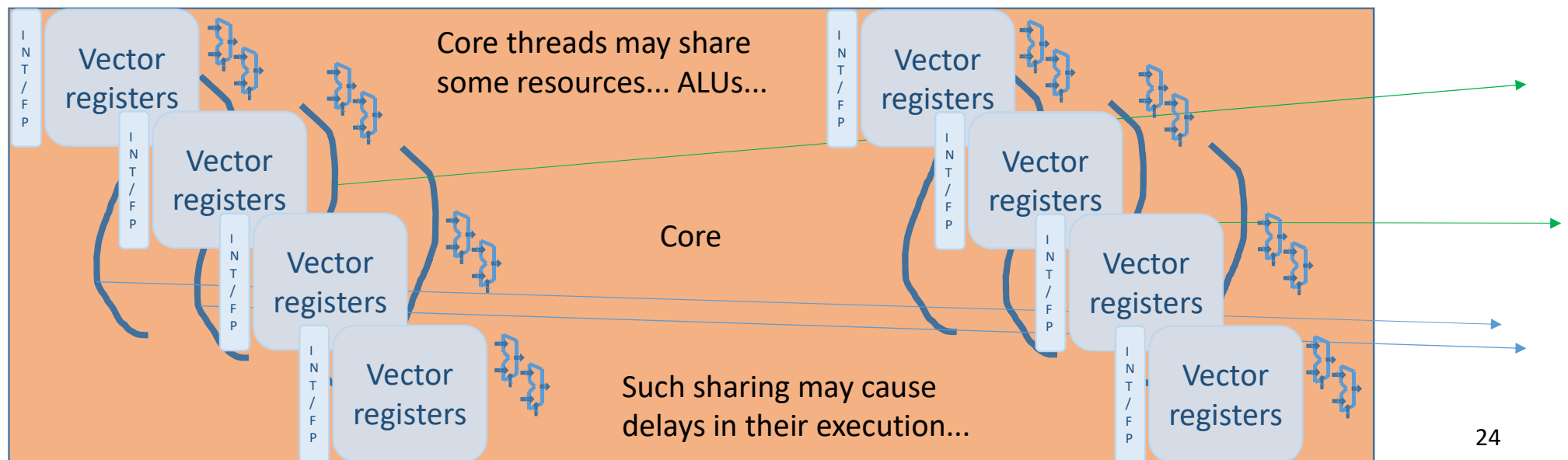
integer instructions      1-4 cycles

floating point instructions 8-30 cycles

jump instructions      1-20 cycles

# Core details

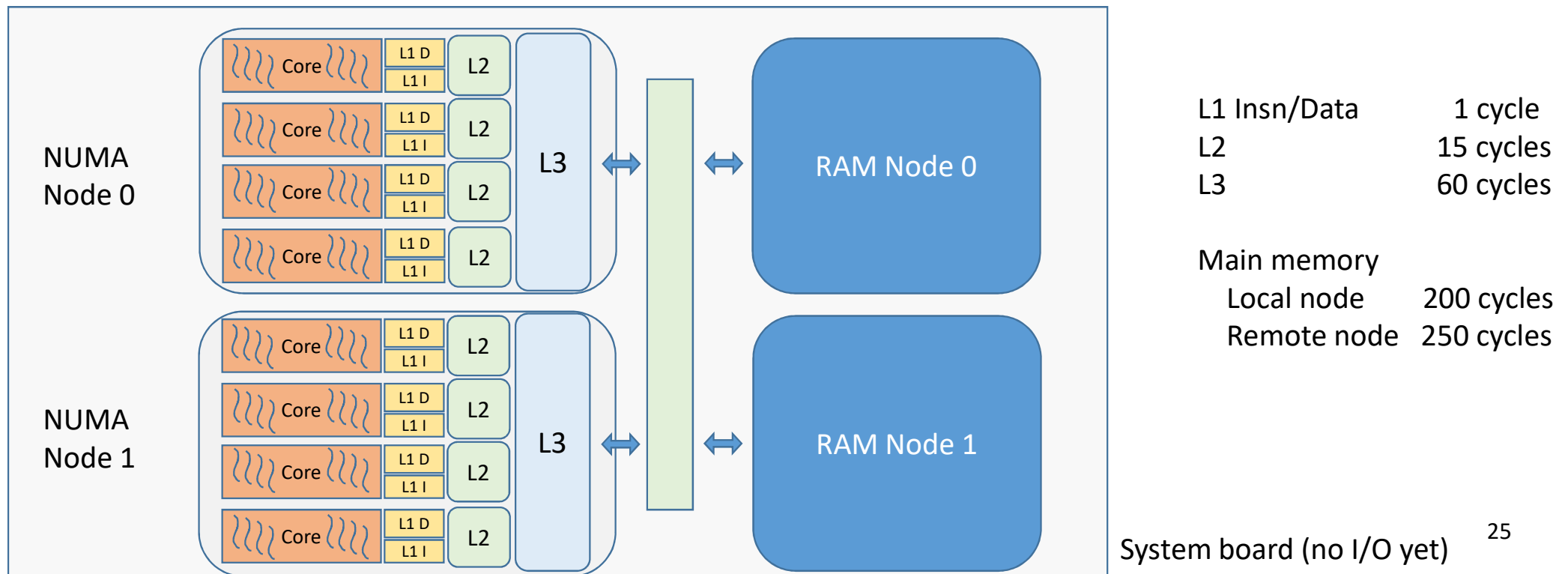
- Instructions need the use of **registers** for bringing data to the thread
  - Load/store instructions bring data from memory (also mov, add, mul...)
  - Computation instructions use the ALUs to process data (add, mul...)
  - Control instructions break the execution sequence (conditionally...)





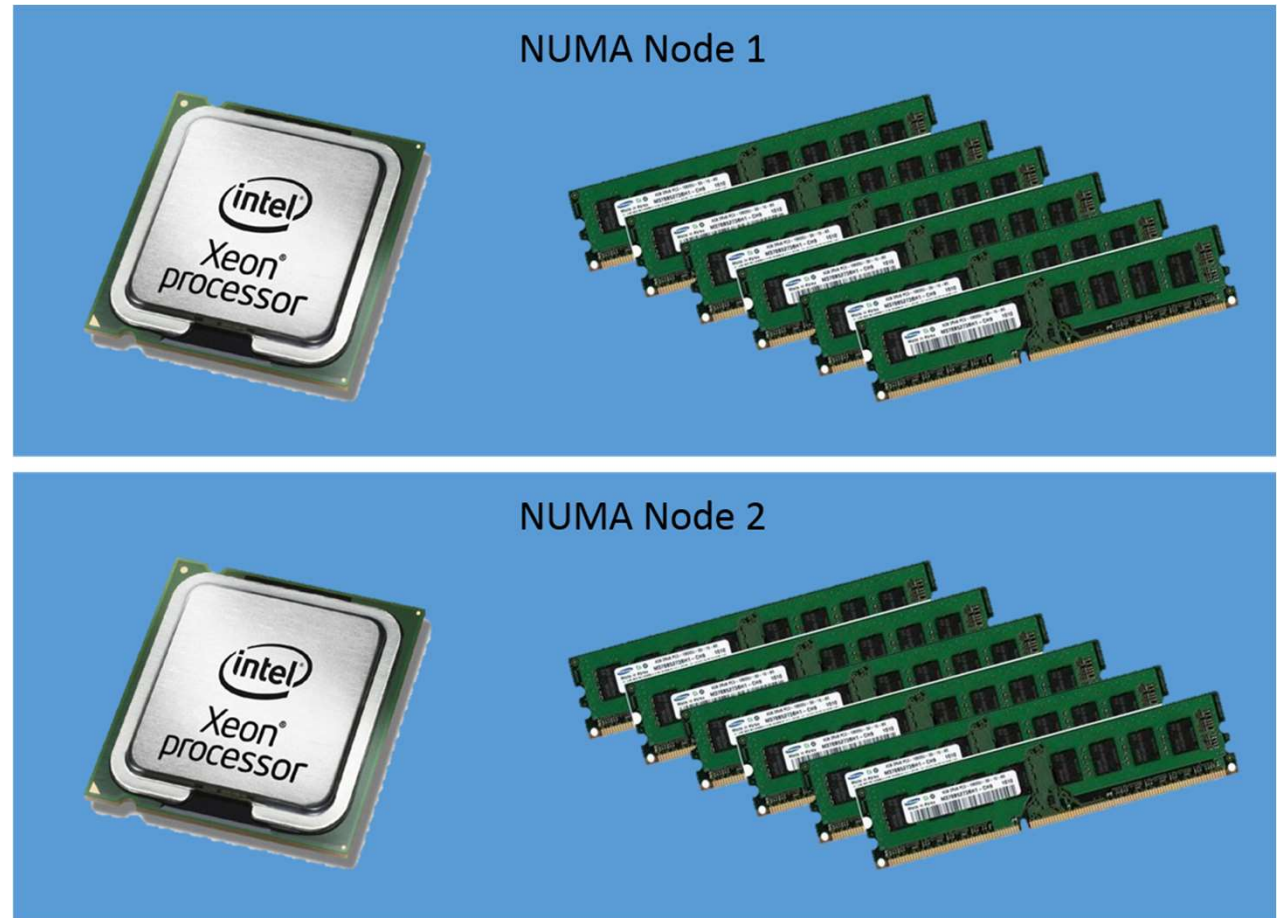
# Complete processor/memory system

- Most usually, systems have two or more chips
  - NUMA – Non-Uniform Memory Access



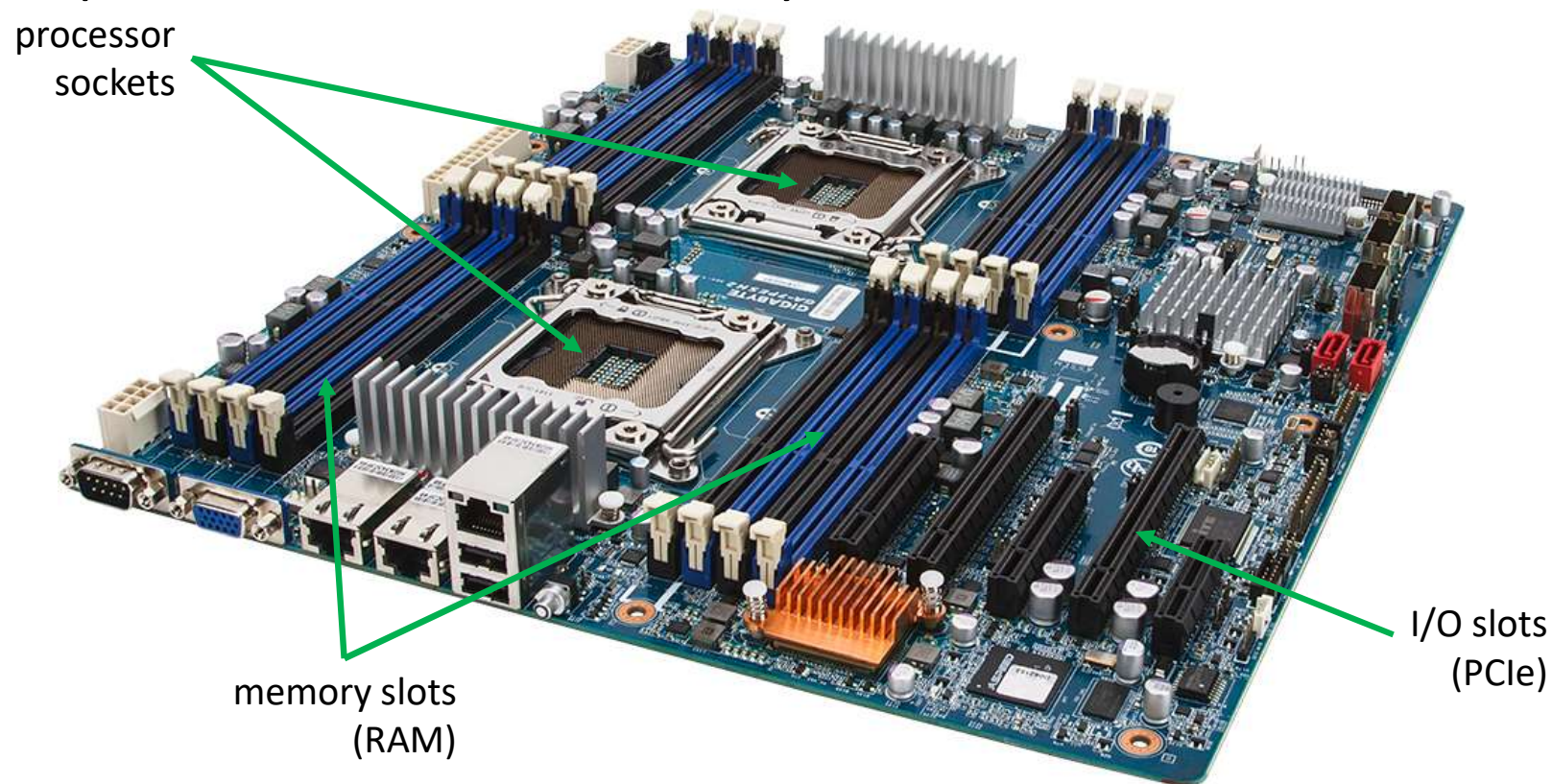
# Example of multiprocessor motherboard

- Schematics
  - Two processors
  - Two memory nodes
- **Exercise**... where are L1I, L1D, L2, L3?



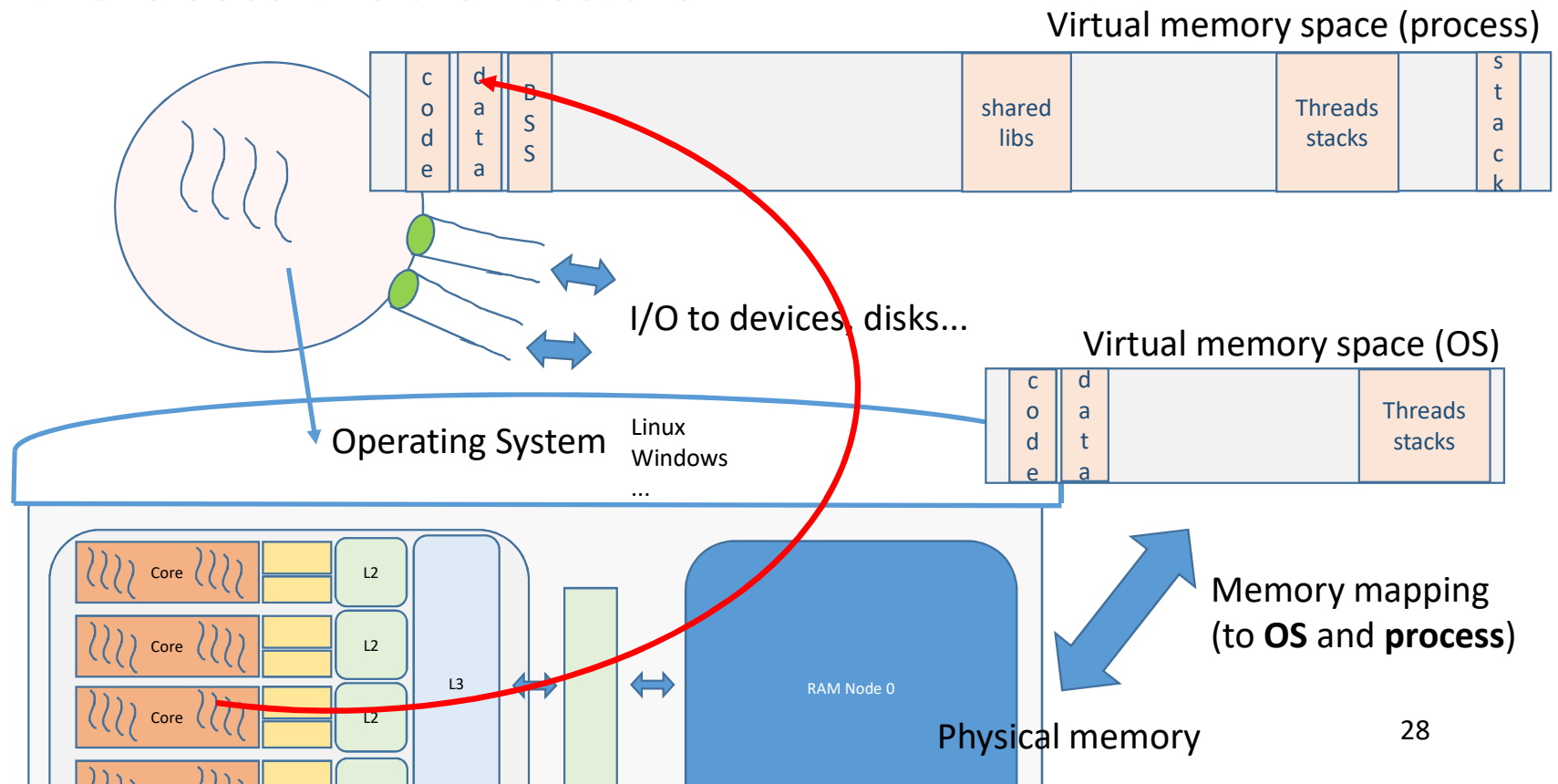
# Example of multiprocessor motherboard

- Two processors and two memory nodes



# Software/hardware mapping

- How the software uses this architecture?



# Current processor chips

---

- Intel Xeon E7 v4 family

<https://ark.intel.com>

Intel processor descriptions

- 14 nm technology
- 24 cores / hyperthreading (2), 2.2 – 3.4 GHz.
- L3 cache 60MB.
- MAX CPU supported 8 sockets
- 3.07 TB. MAX RAM 1866 MHz., 4 memory channels
- PCIe x4, x8, x16

# Current processor chips

---

- IBM Power 9

<https://www.ibm.com/it-infrastructure/power/power9>

- 14 nm technology
- 24 cores / SMT (8), 3.0 – 4.0 GHz.
- L1 caches 32+32 KB.
- L2 cache 512 KB.
- L3 cache 120MB.
- MAX CPU supported 4-8 and more sockets
- 2 TB MAX RAM DDR4
- PCIe v4 x4, x8, x16

# Current processor chips

---

- Intel KNL – Xeon Phi 72x5
  - 14 nm technology
  - 72 cores 1.5 – 1.6 GHz.
  - L2 cache 36 MB.
  - MAX CPU supported 1 socket?
  - 384 GB. MAX RAM DDR4
  - PCIe v3 x4, x8, x16

intel-xeon-phi-processor-7295-16gb-1-5-ghz-72-core

# Current processor chips

---

- ARM Cortex-A77

<https://www.arm.com/products/silicon-ip-cpu/cortex-a/cortex-a77>

- 7 nm technology
- aarch64 – ARMv8-A
- 4-8 cores
- DynamIQ Technology – (big-LITTLE)

- ARM Cortex-A72 – A64FX (Fujitsu)

- 7 nm
- ARMv8.2
- 48 cores
- 512-bit SIMD Scalable Vector Extensions (SVE)



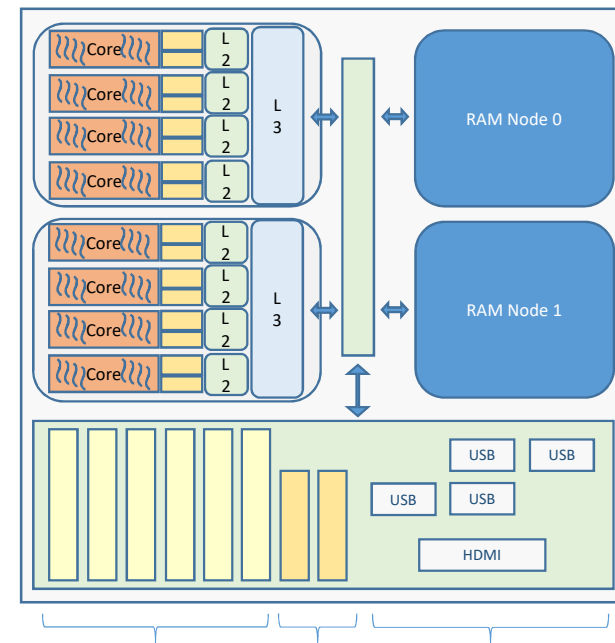
# Table of Contents (cont.)

---

- Computer structure. Components
  - Input/Output components
    - Buses. Peripherals
  - Storage and file systems
    - Disks
  - Networking
    - Connexions and protocols

# Input/Output components

- The I/O Bus extends the access to
  - Accelerators (GPUs, FPGAs)
  - Disks
  - Network
  - Human-Machine Interface Peripherals



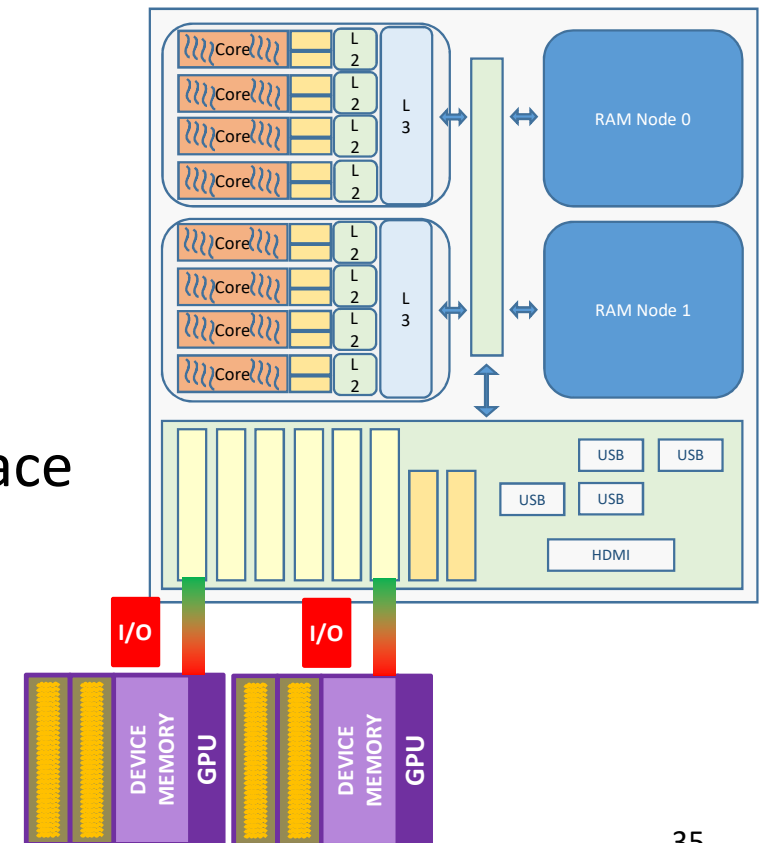
PCIe cards  
GPUs / FPGAs  
Networking

Sata  
Disks

HMI Peripherals  
Keyboard/mouse  
Video  
Audio

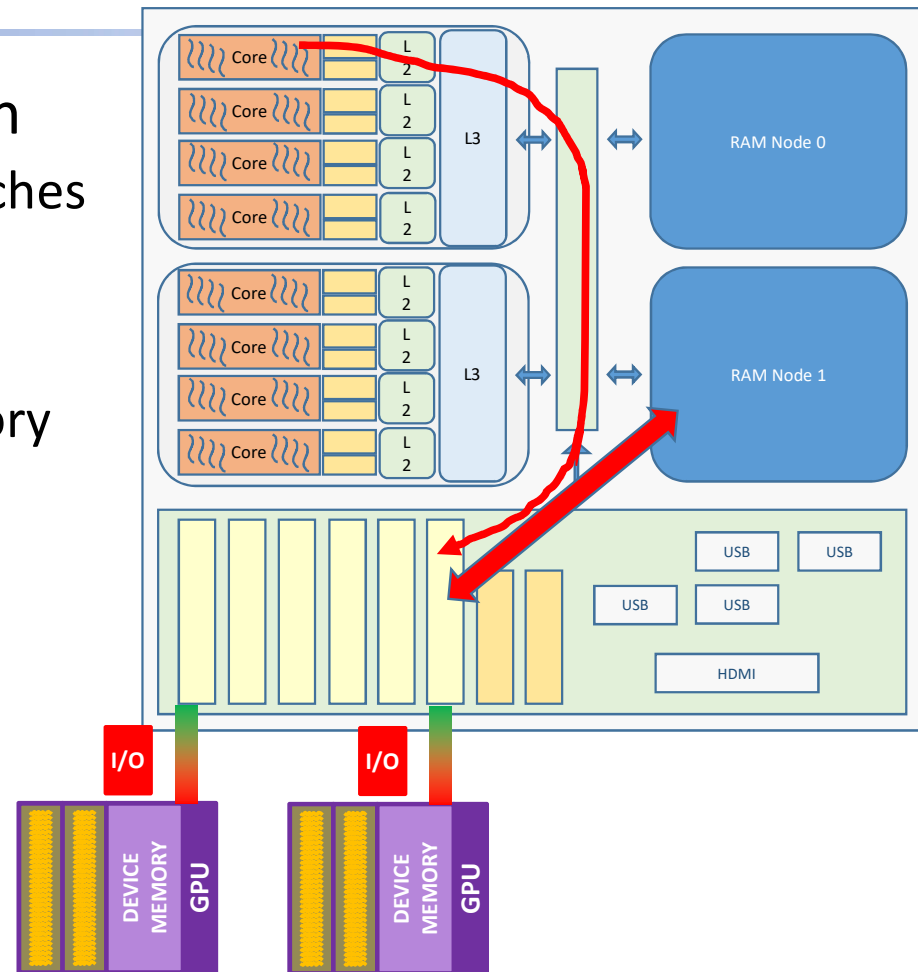
# Accelerators

- Devices attached for computation
  - Need to transfer
    - code and datato/from the device
  - Need to start/stop execution
  - Synchronization
- Configuration through mapped memory space
  - Use specific addresses to access the device's configuration registers
  - Access only allowed from the OS



# Access to accelerators/devices/peripherals

- Accesses to device configuration
  - Uncached – do not access the caches
    - Property of the memory mapping
- Accesses to device memory
  - Through specialized Direct Memory Access (DMA) engines

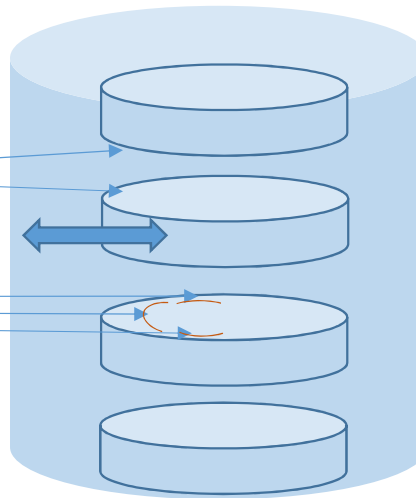


# Sata and HMI Peripherals

- Disks

- Addressed by

- Head
    - Cylinder
    - Sector

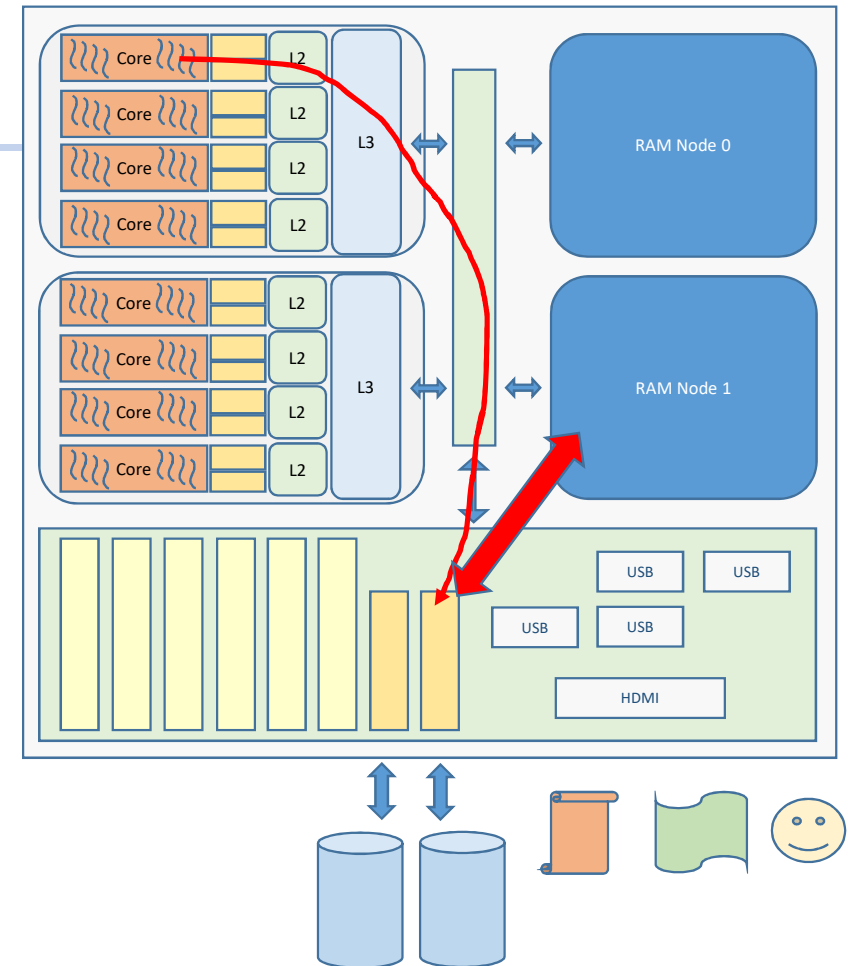


- Video

- With special video memory

- USB

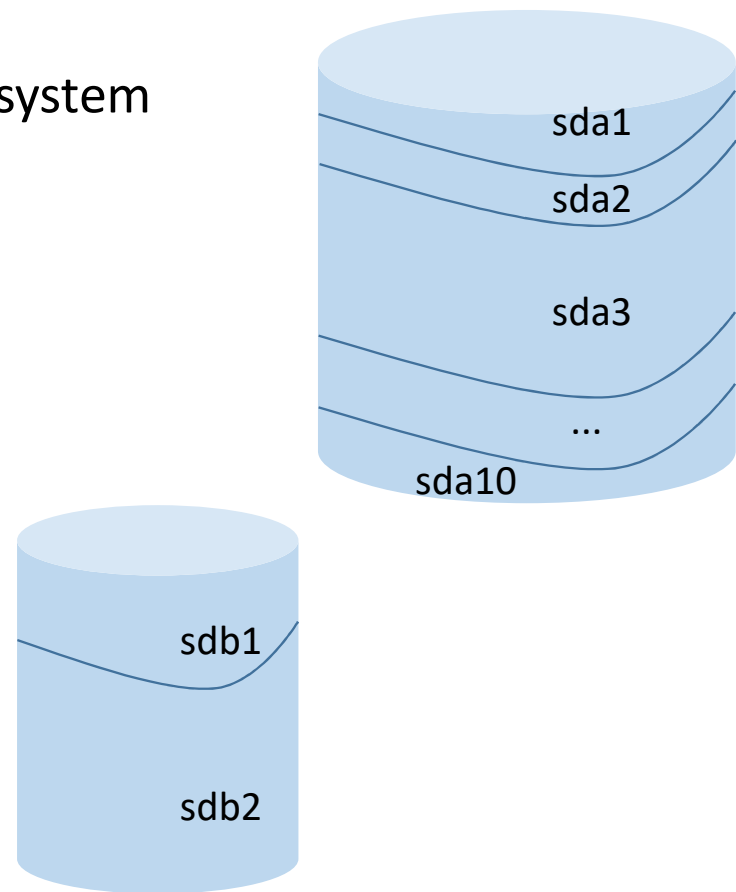
- Keyboard, mouse...



# Storage and file systems

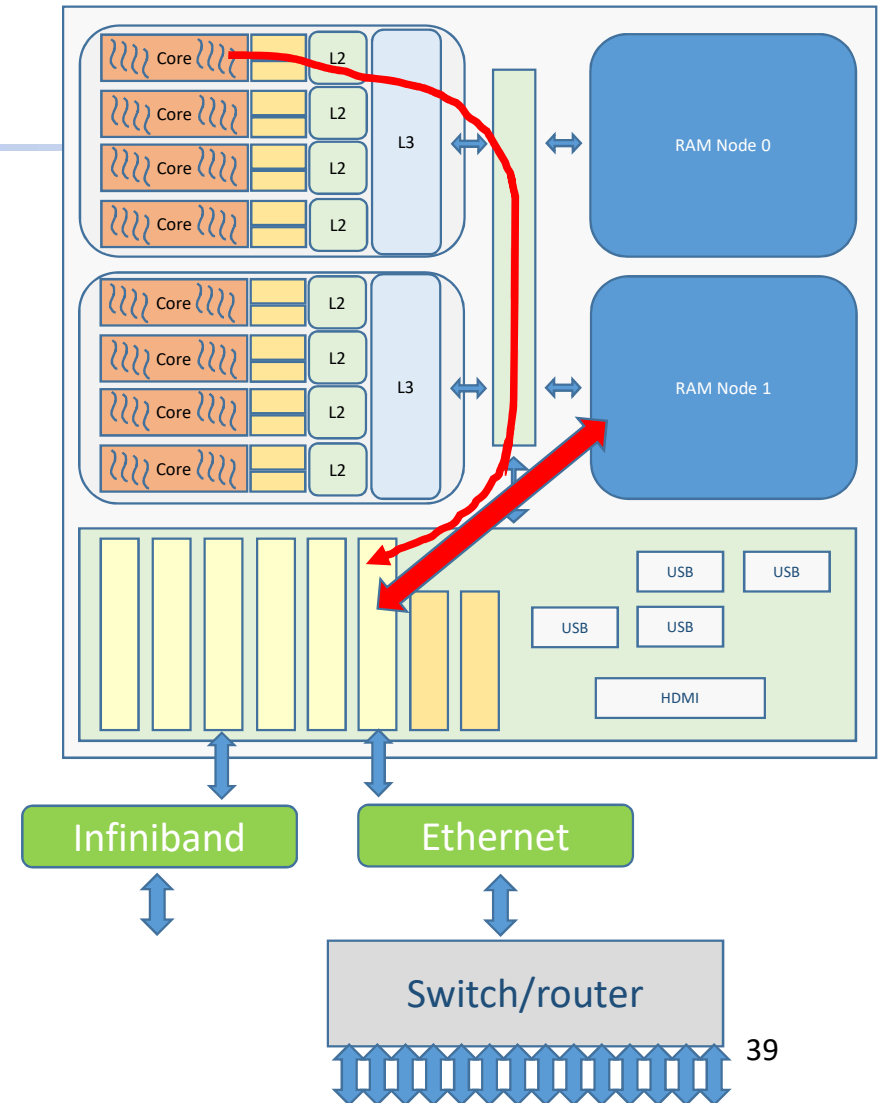
---

- Disks are usually split in partitions
  - Each partition can support a different file system
    - / (root)
    - /usr/local
    - /home
    - /opt
    - swap area
    - ...
  - Different types of file systems exist
    - Linux: ext4, ext3, ext2, btrfs, hfs, jfs, xfs...
    - Windows: ntfs, FAT, exFAT



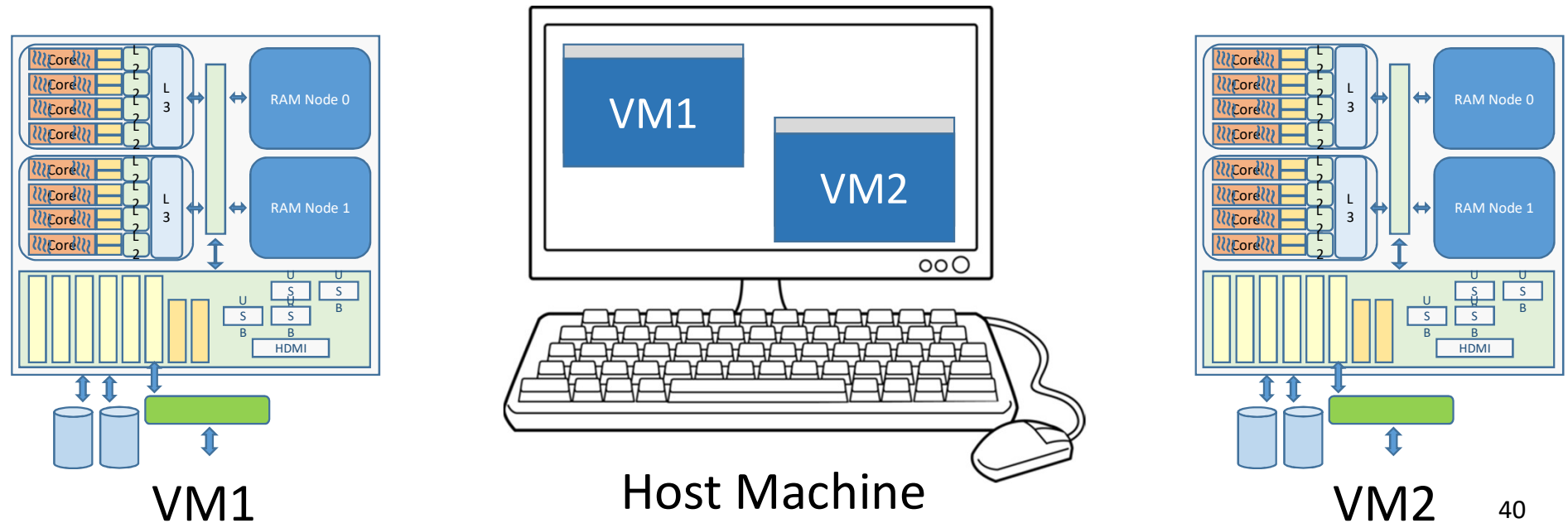
# Networking

- Send/receive information to
  - Servers
  - Network-attached disks
- Protocols
  - Low-level – ethernet packet
  - High-level – TCP/IP
- Control based on memory mapped configuration registers
  - Access from the OS
- Data transfers based on DMA engines



# Virtual Machine (VM)

- A software package virtualizes all physical resources of a computer
  - Virtualized resources can be emulated (simulates the real behavior) or linked to access real resources of the host machine
  - Multiple VMs can run on the same host. Everyone can run a different Operating System





# Bibliography

---

- Computer Organization and Design (5th Edition)
  - D. Patterson and J. Hennessy
  - [http://cataleg.upc.edu/record=b1431482~S1\\*cat](http://cataleg.upc.edu/record=b1431482~S1*cat)
    - Several chapters introduce different types of data

# Next steps

---

- Support to the programming environment
  - System libraries
  - Compilation environment
  - Operating system