UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

Computer Architecture

COMPUTER ARCHITECTURE AND OPERATING SYSTEMS

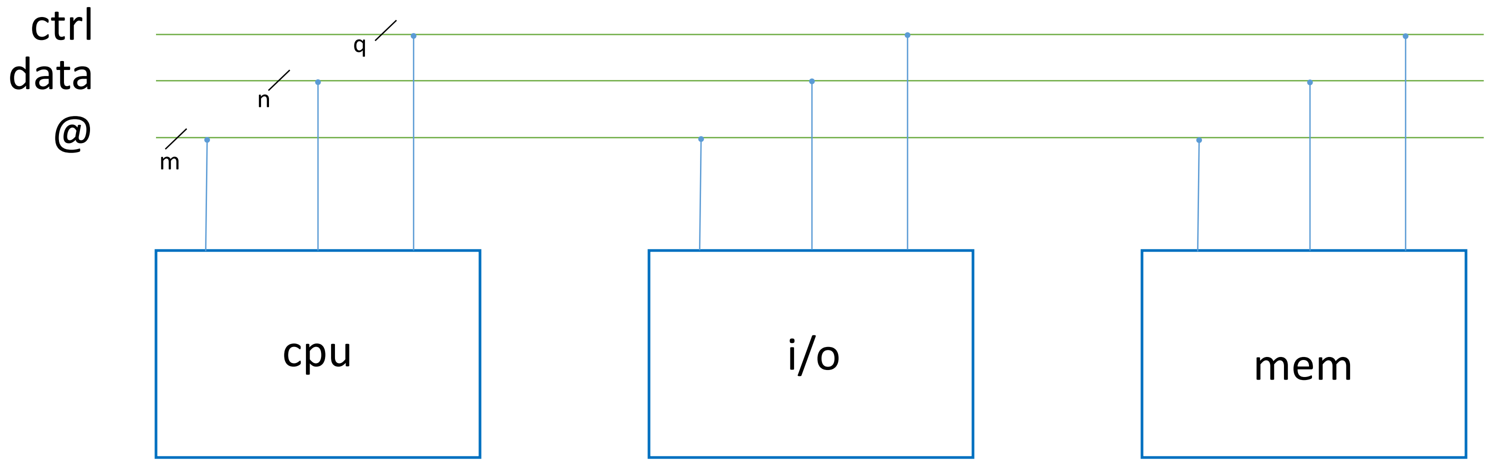
2025/26 Spring Term

Jordi Fornés

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Departament d'Arquitectura de Computadors

Computer organisation

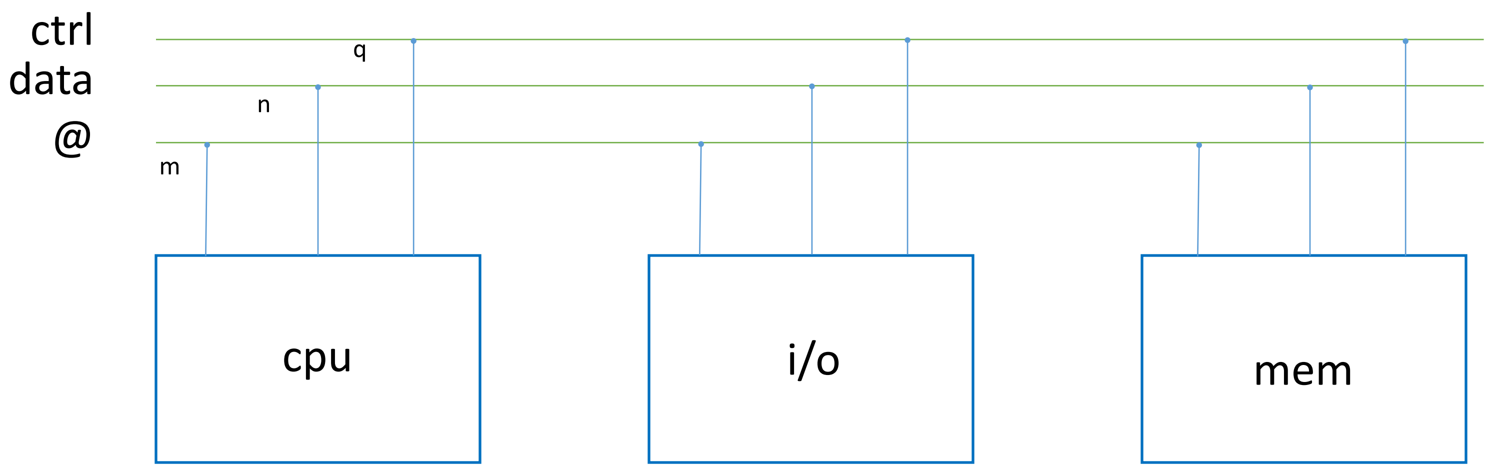
- The so called *von Neumann* architecture



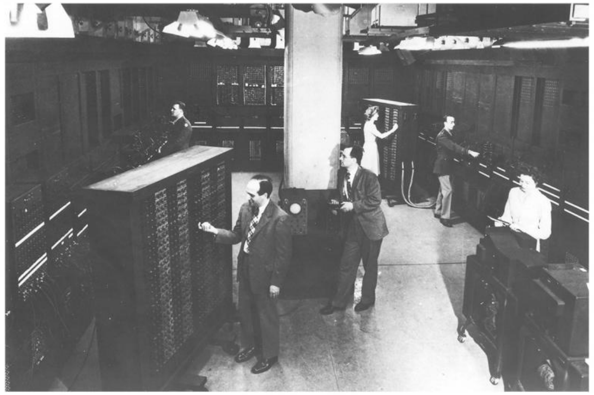
https://en.wikipedia.org/wiki/John_von_Neumann

Computer organisation

- The so called *von Neumann* architecture



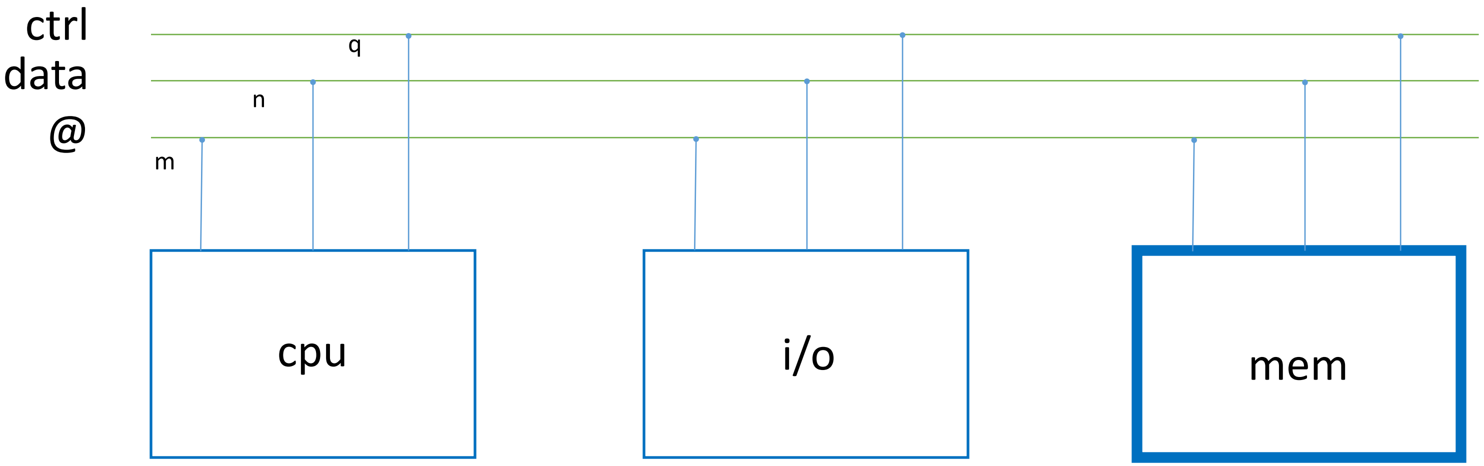
https://en.wikipedia.org/wiki/John_von_Neumann



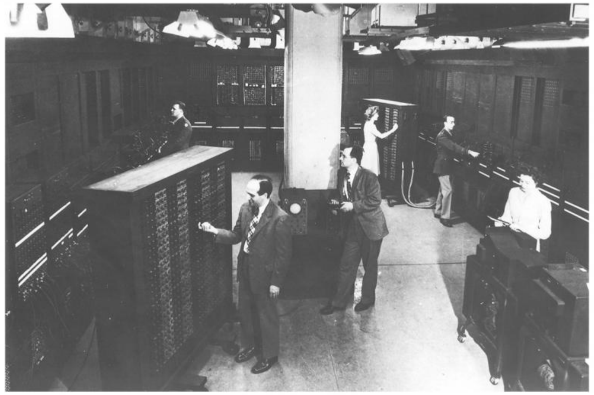
https://www.fi.edu/case-files/mauchly-and-eckert

Computer organisation

- The so called *von Neumann* architecture

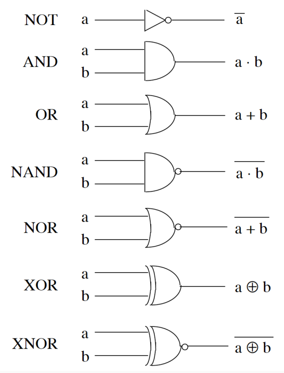


https://en.wikipedia.org/wiki/John_von_Neumann



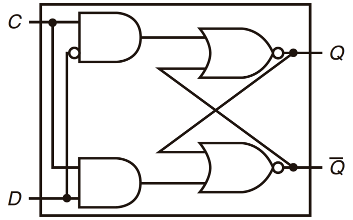
https://www.fi.edu/case-files/mauchly-and-eckert

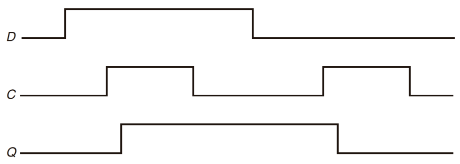
Memory components



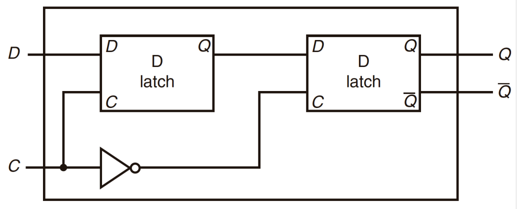
a	NOT
0	1
1	0

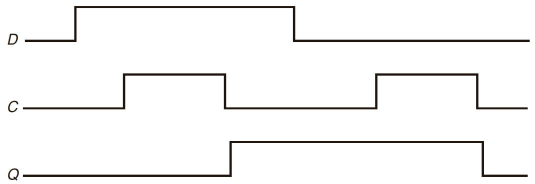
a	b	AND	OR	NAND	NOR	XOR	XNOR
0	0	0	0	1	1	0	1
0	1	0	1	1	0	1	0
1	0	0	1	1	0	1	0
1	1	1	1	0	0	0	1





- **D latch**
  - store the state value unless the clock input C is asserted
  - When C is asserted the value of input D replaces the value of Q





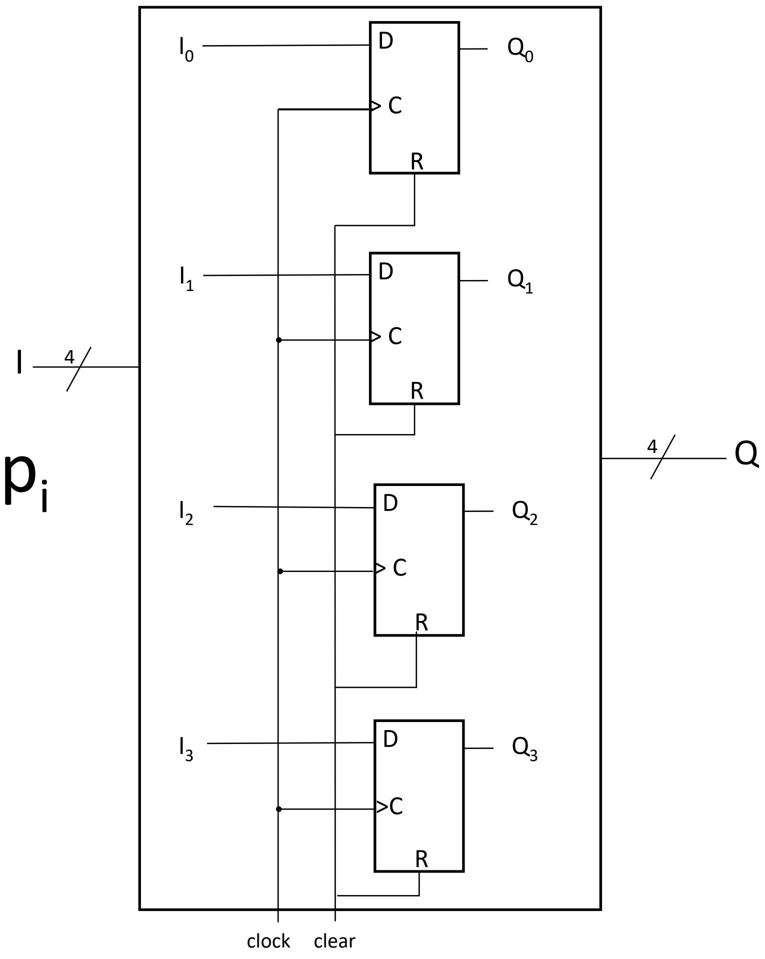
- **flip-flop**
  - The output is equal to the value of the stored state
  - The internal state is changed only on clock edge

Source: Patterson, David. *Computer organization and design.* Morgan Kaufmann 2021.

Memory components

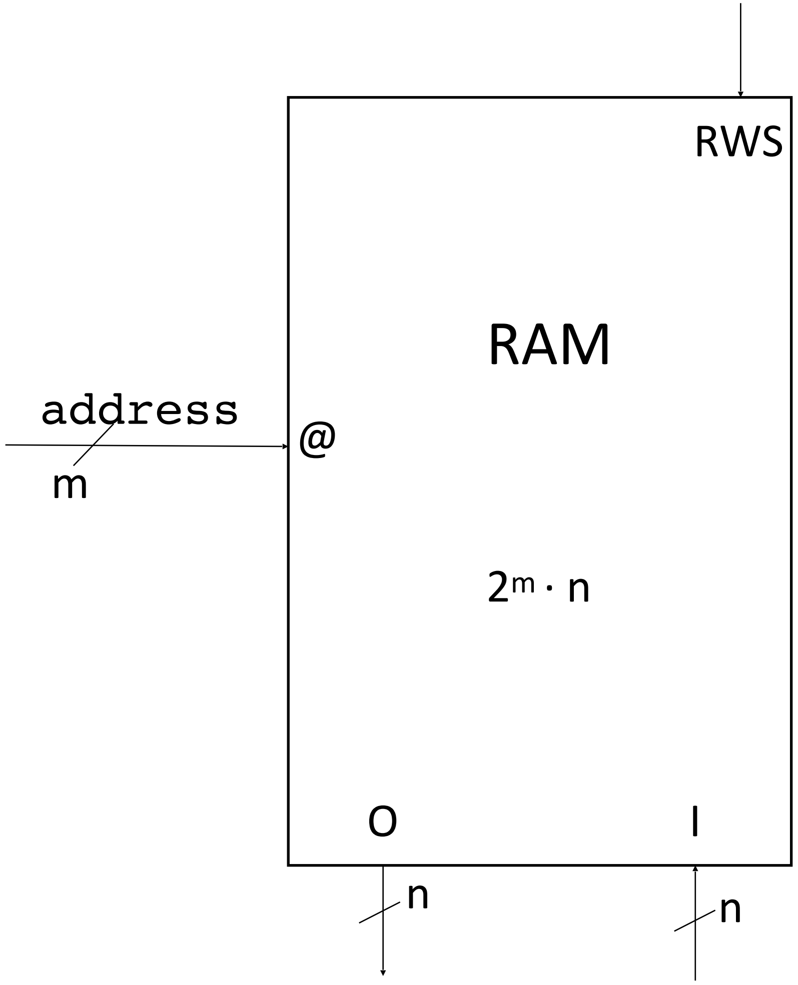
- A register is a flip-flop with several bits
  - A n-bits register consists n flip-flops with n inputs, n outputs and 1 clock
  - Various types of registers are available commercially
- In a shift register the output of the flip-flop$_i$ is connected to the input of the flip-flop$_{i+1}$
- A register file is an array of registers
  - Each register can be read by supplying a its register number

4-bit register



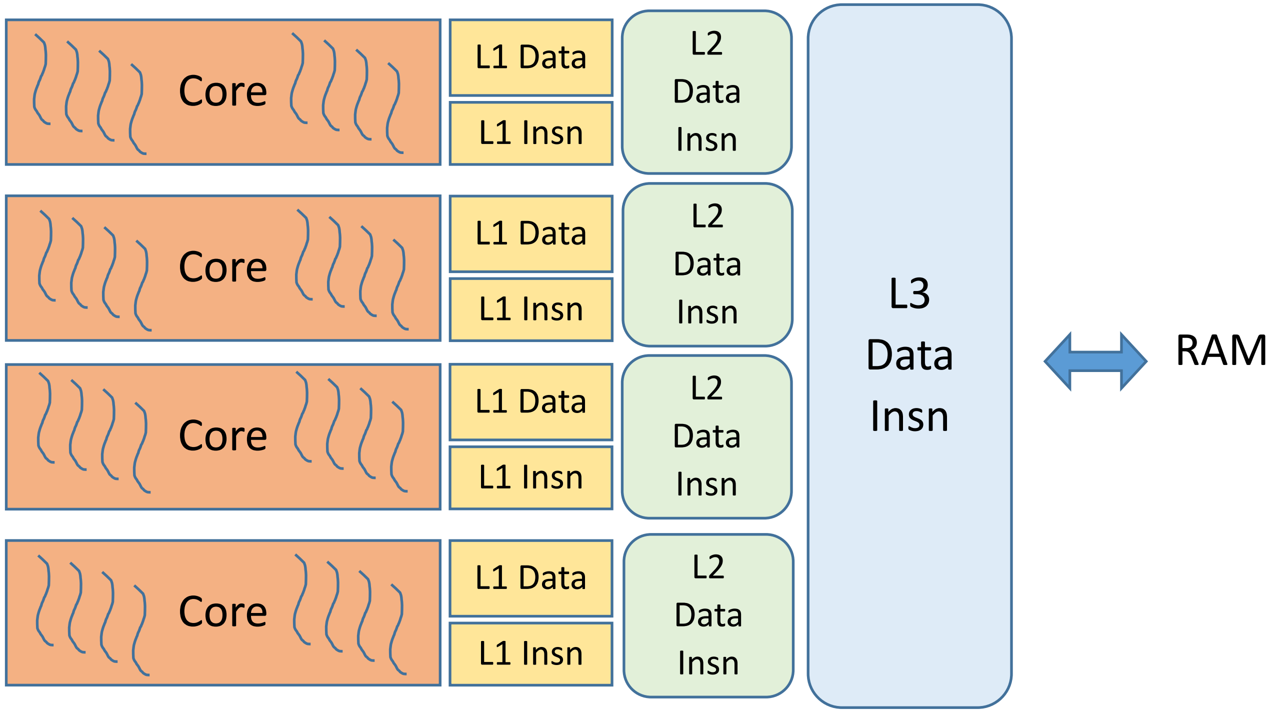
Memory components

- Random Access Memory
  - Larger amounts of memory than registers
  - Slower access than registers
  - Organised as arrays of 2^m rows of n bits
    - m bits needed to select a row
    - Read Write Selector (RWS) control bit
      - RWS = 0, RAM reads the `address` and the contents are available in O
      - RWS=1, RAM writes I at the `address`



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

Memory organisation

- Endianness
  - The order of byte wise values in memory
- Big-Endian
  - Byte with **most** significant value: stored first (lowest memory address)
  - Data networking and mainframes
  - Motorola 68000 and PowerPC G5 are big-endian
- Little-Endian
  - Byte with **least** significant value: stored first (lowest memory address)
  - x86 Intel and AMD64 processors family and most microprocessors
- Some architectures support both
  - E.g. Arm and IBM POWER in full, recent x86 and x86-64 have limited support (movbe)

E.g. 1234_{10}= 04 $D2_{16}$

@	data
0x0000:	04
0x0001:	D2

@	data
0x0000:	D2
0x0001:	04

Big endian

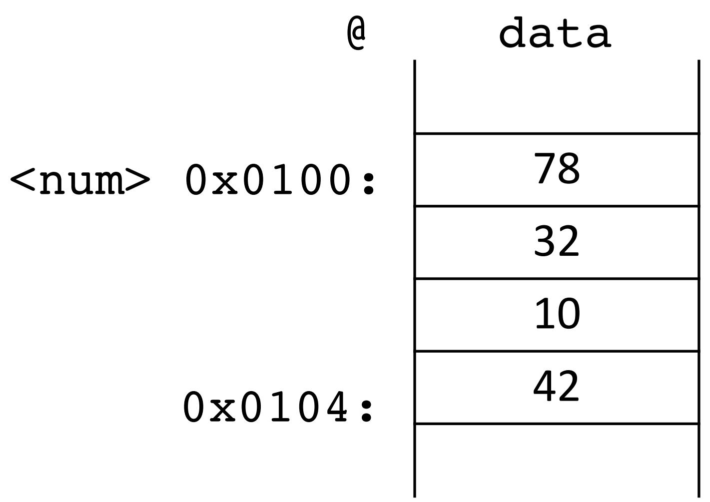
- The location address points to the Big end of the number
  - Like writing the left-to-right

`num=0x42103278`

@	data
<num> 0x0100:	42
	10
	32
0x0104:	78

Little Endian

- The location address points to the Little endian of the number
  - Like writing the bytes right-to-left



num=0x42103278

https://en.wikipedia.org/wiki/Endianness

Endianness in Python

- Handling binary data
  - stored in files
  - or from network connections

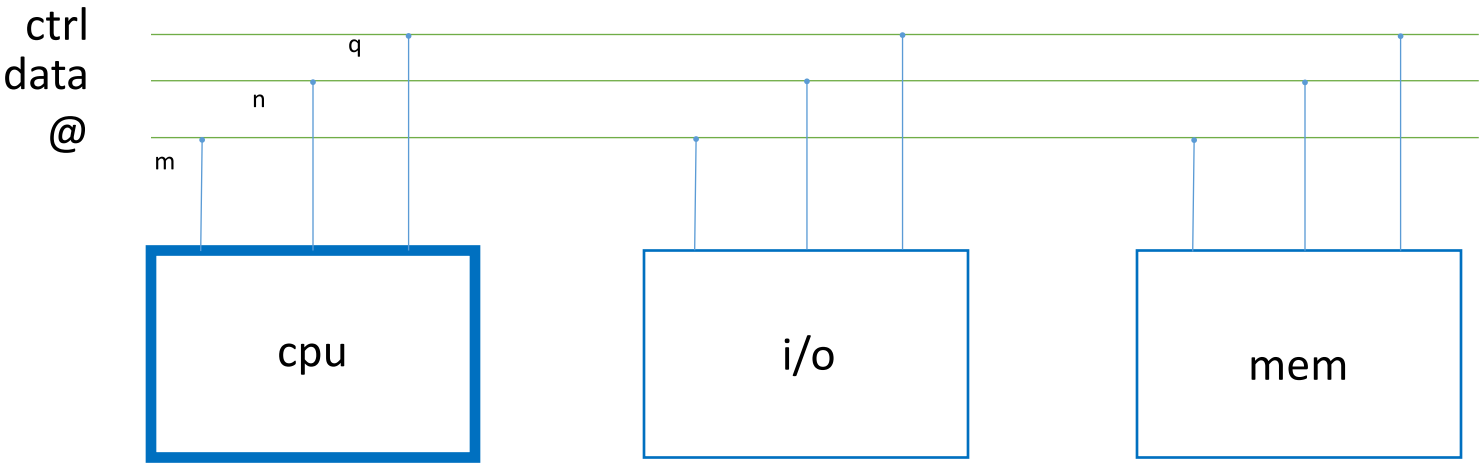
Character	Byte order	Size	Alignment
@	native	native	native
=	native	standard	none
<	little-endian	standard	none
>	big-endian	standard	none
!	network (= big-endian)	standard	none

```
>>> import sys
>>> sys.byteorder
'little'
>>> from struct import *
>>> pack('>hhl', 1, 2, 3)
b'\x00\x01\x00\x02\x00\x00\x00\x03'
>>> pack('<hhl', 1, 2, 3)
b'\x01\x00\x02\x00\x03\x00\x00\x00'
>>> calcsize('hhl')
16
```

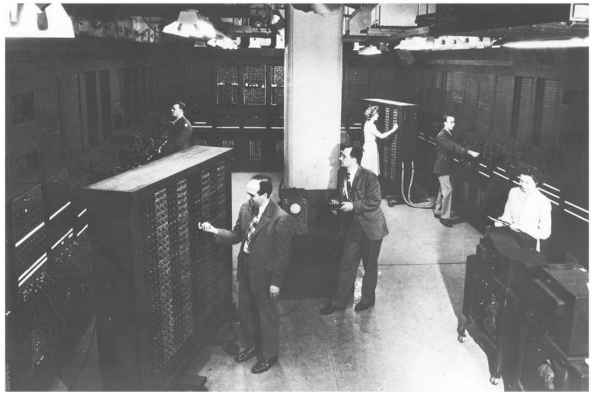
https://docs.python.org/3/library/struct.html

Computer organisation

- The so called *von Neumann* architecture

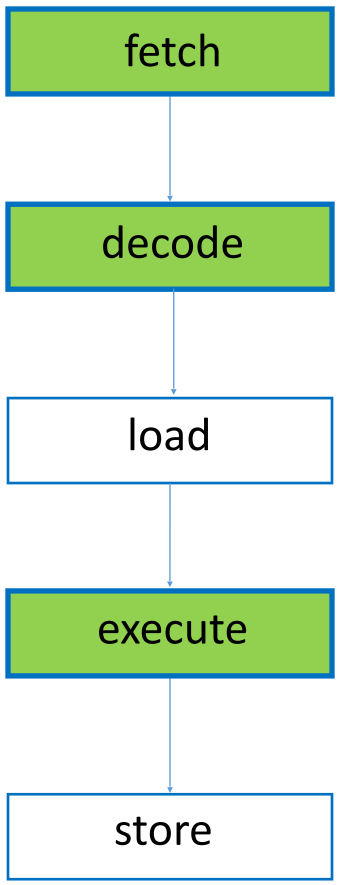


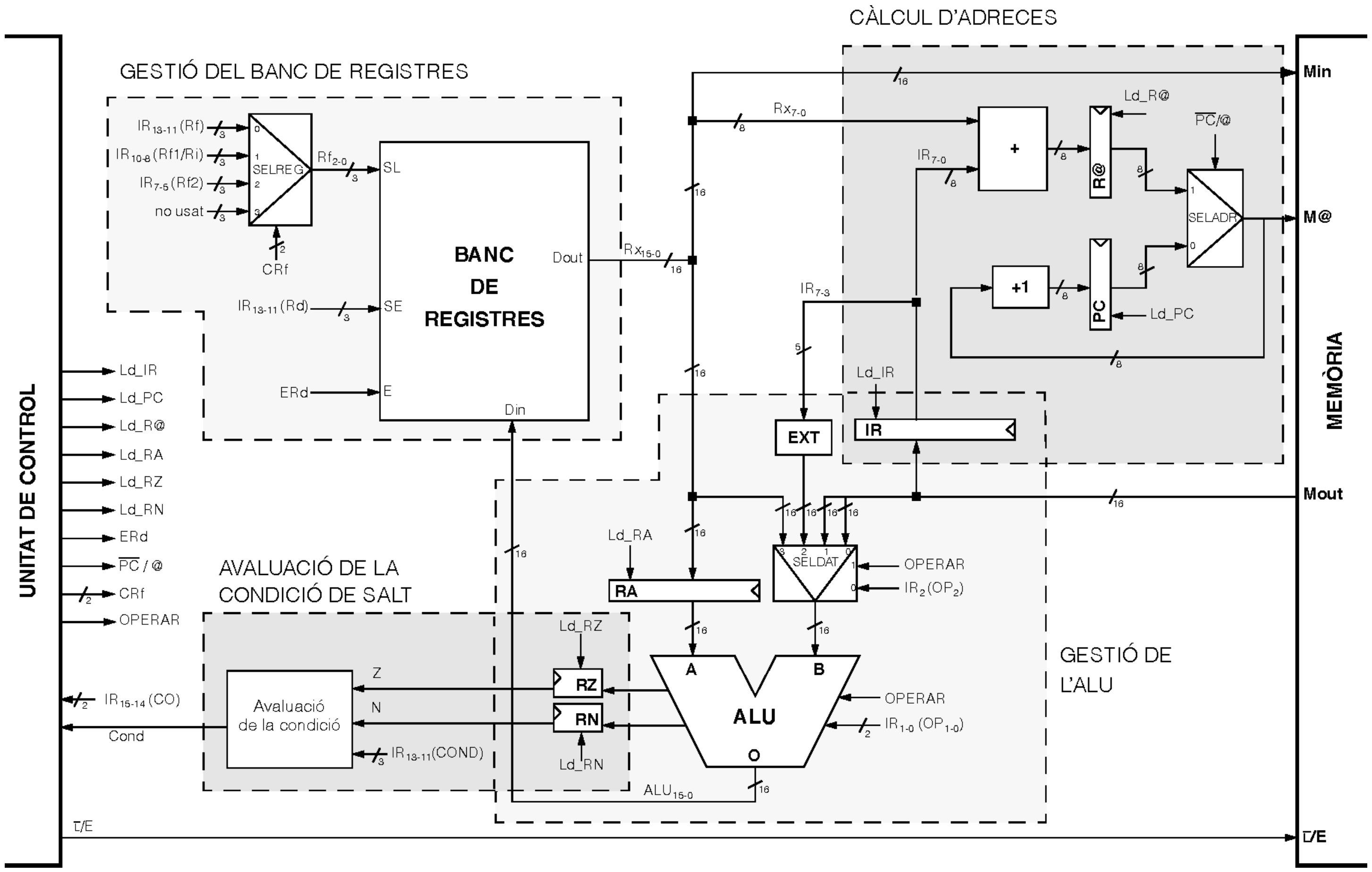
https://en.wikipedia.org/wiki/John_von_Neumann



https://www.fi.edu/case-files/mauchly-and-eckert

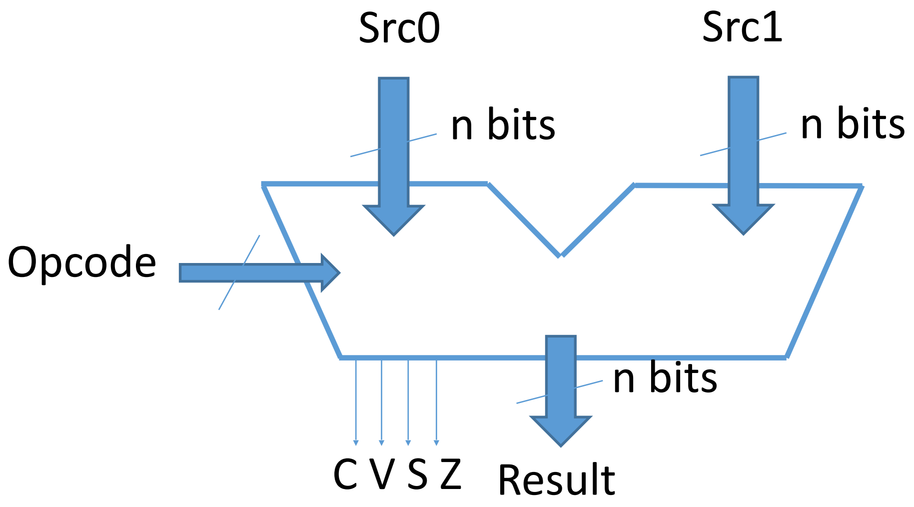
Central Processor Unit





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

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 - programs

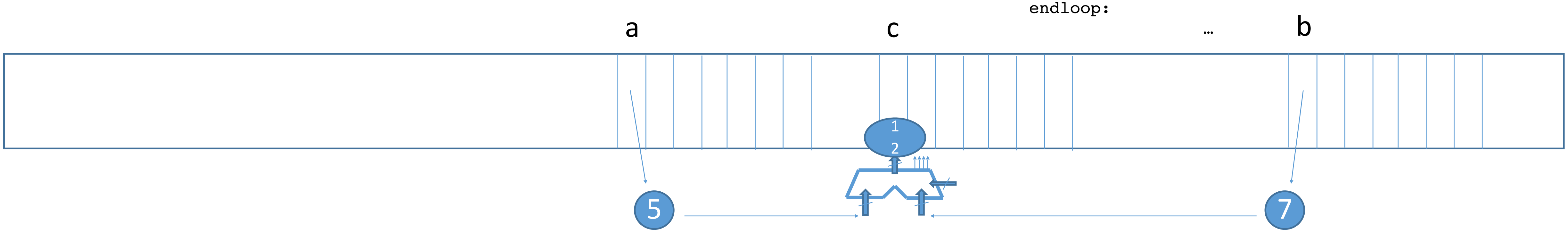
- Compute the sum of two vectors
  - Vectors = data;   data is stored in memory

```
>>> import numpy as np
>>> a = np.array([5, 2, 3, 4, 5])
 >>> b = np.array([7, 7, 8, 9, 10])
>>> a + b
array([12,  9, 11, 13, 15])
```

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

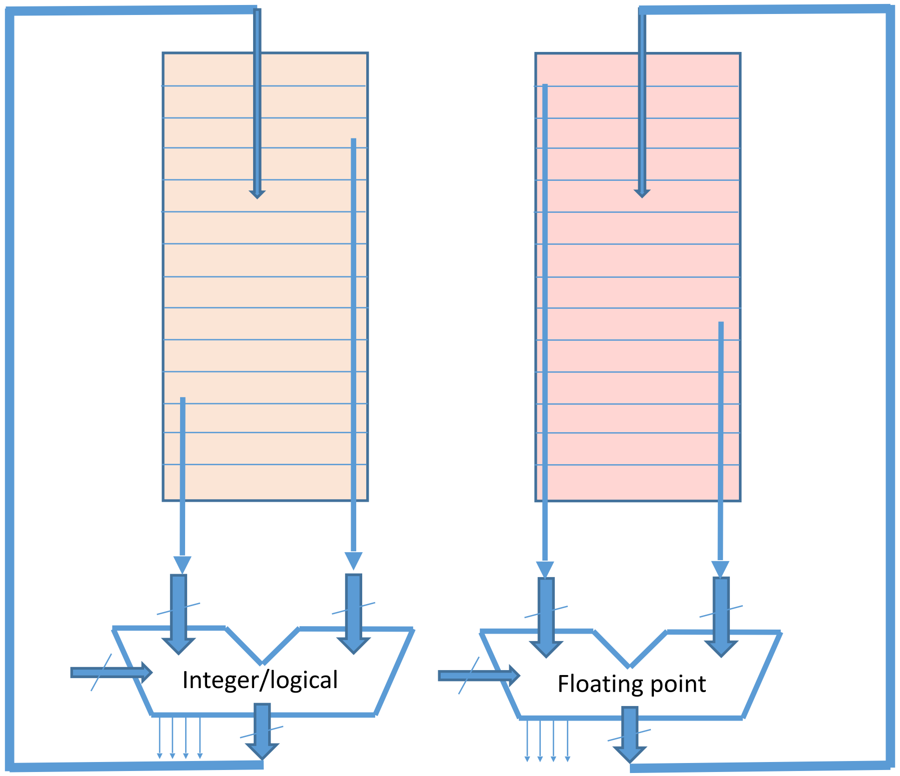
```
move   #0, r8
while    (r8 < N) {
    load      r1, a[r8]
    load      r2, b[r8]
    add       r1,r2,r3
    store     r3, c[r8]
    add       #1,r8
}
```

```
          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:
```

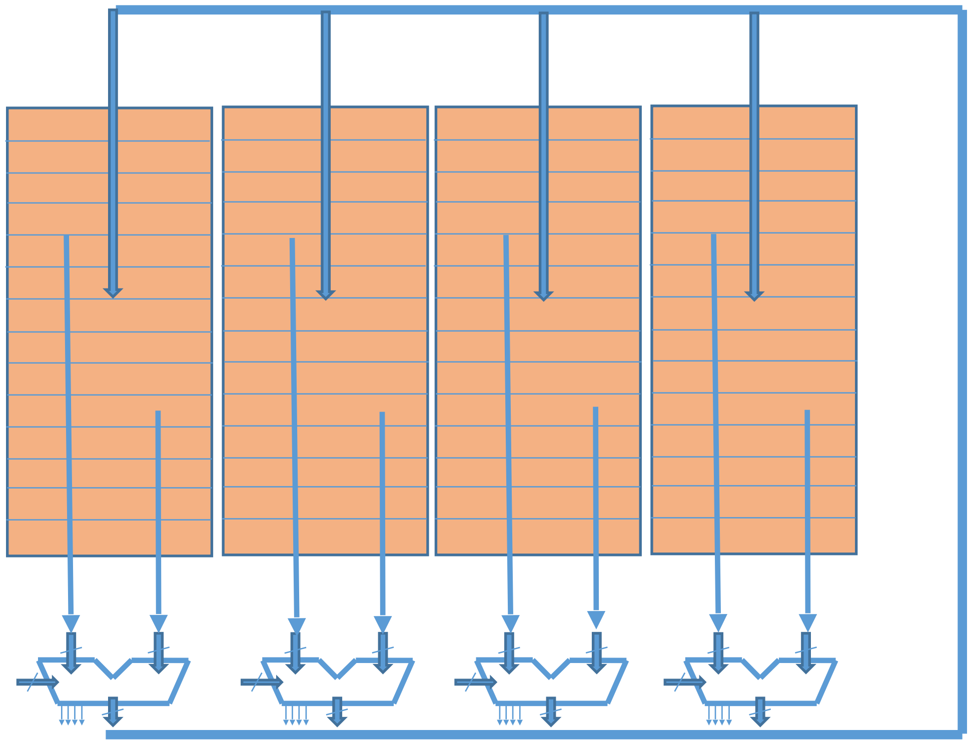


Data operations

▶ALUs and data file **registers**



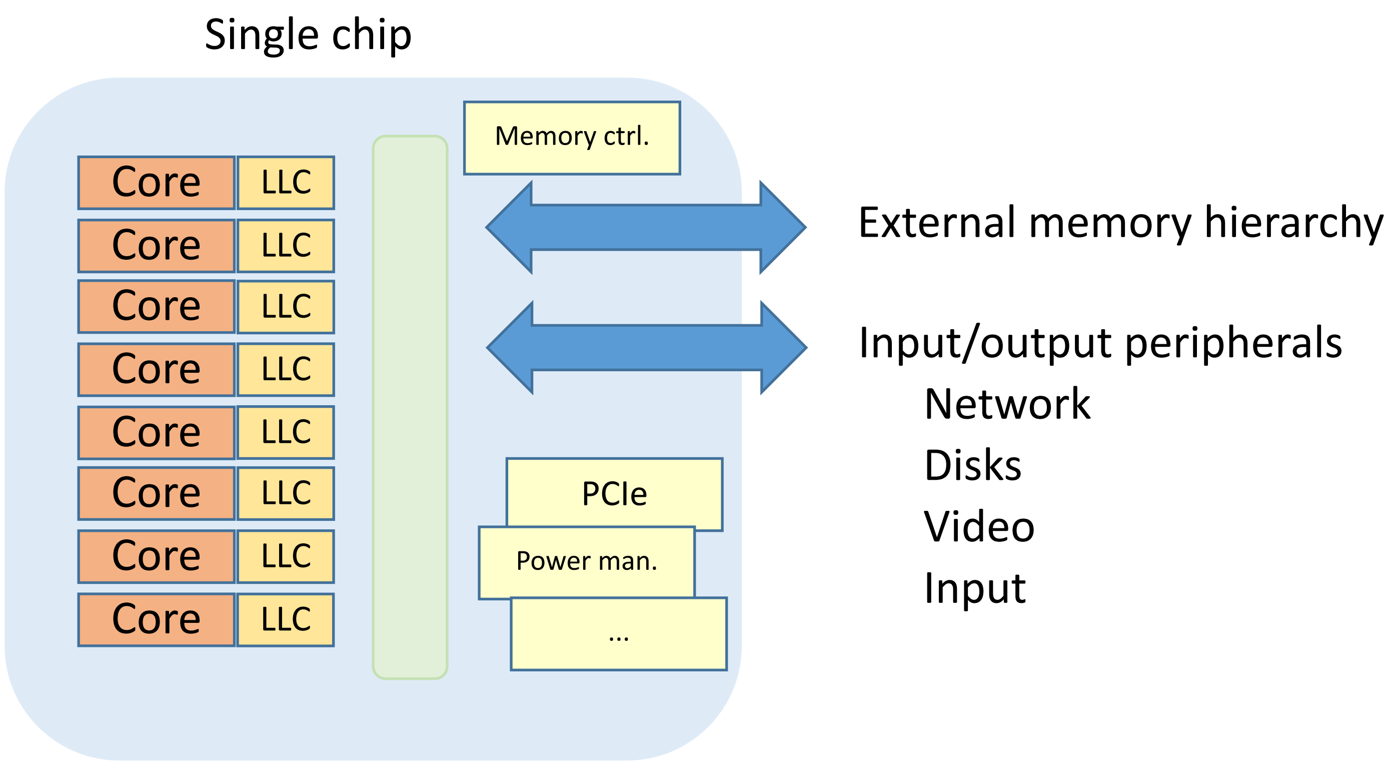
Integer Floating point



Vector (integer and floating point)

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

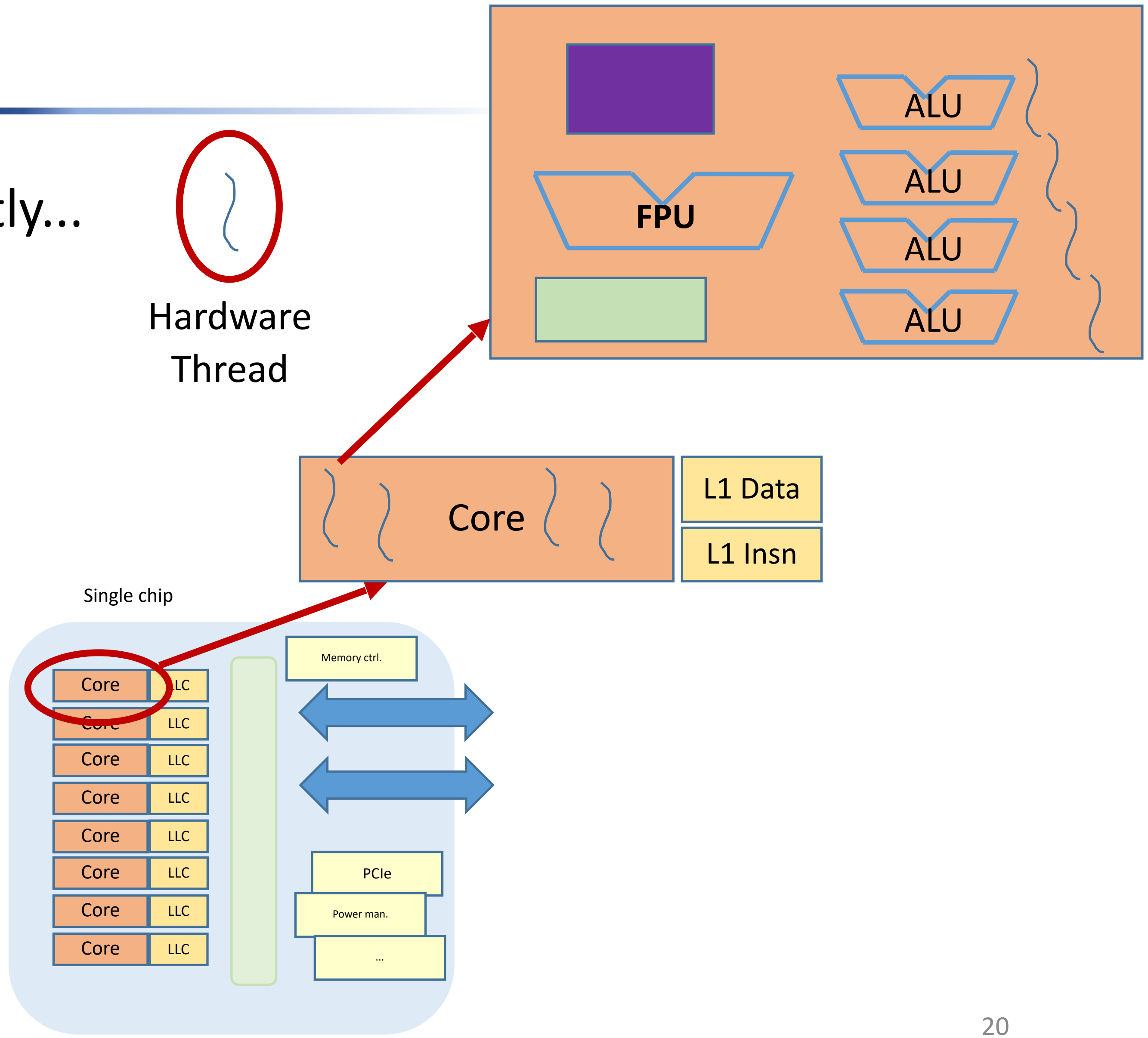
- Exercise... where should we store instructions and data?
- Exercise... how do we generate executable programs?

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

```
def add_vectors(a,b,c):
    for i in range(0, N):
        c[i] = a[i] + b[i]
    return c
```

The thread executes from 0 to N

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

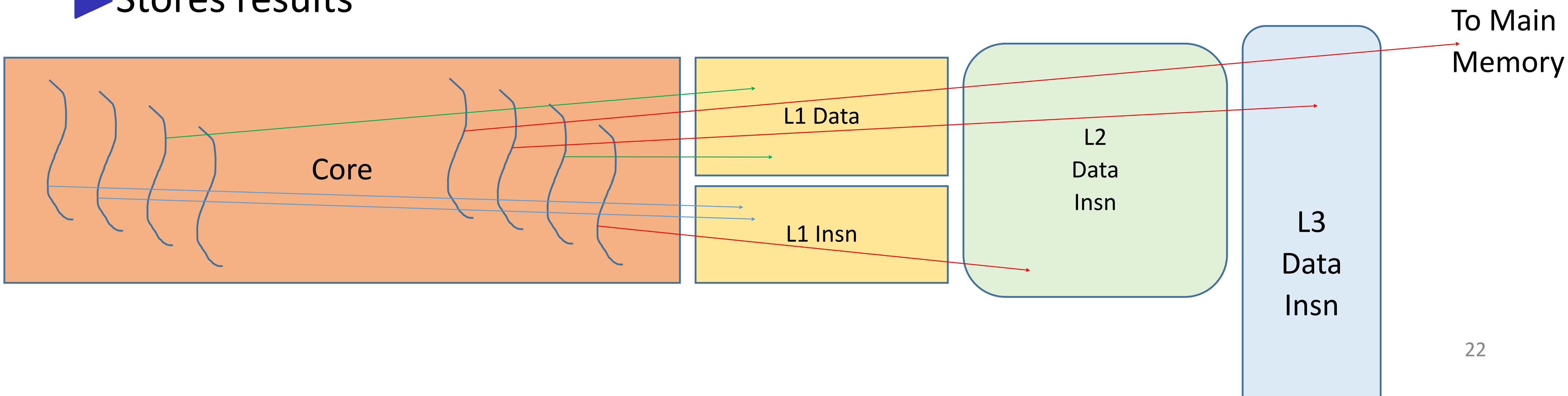
```
def add_vectors(a,b,c):
    for i in range(…, …):
        c[i] = a[i] + b[i]
    return c
```

Every thread executes N/4 iterations of the loop

Hardware multithreading

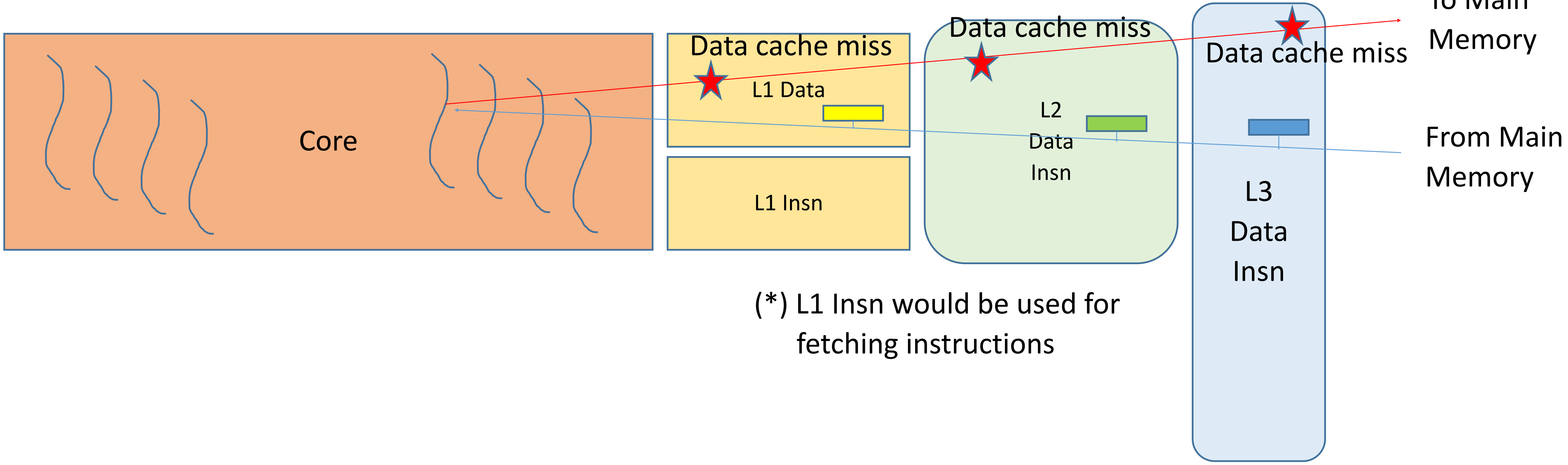
- Each hardware thread independently...
  - Fetches instructions*
  - Decodes
  - Issues load memory accesses*
  - Executes*
  - Stores results*

* When executing a single thread per core, then such a thread has all core resources available!
- Memory bandwidth
- Functional units



Detailed memory access

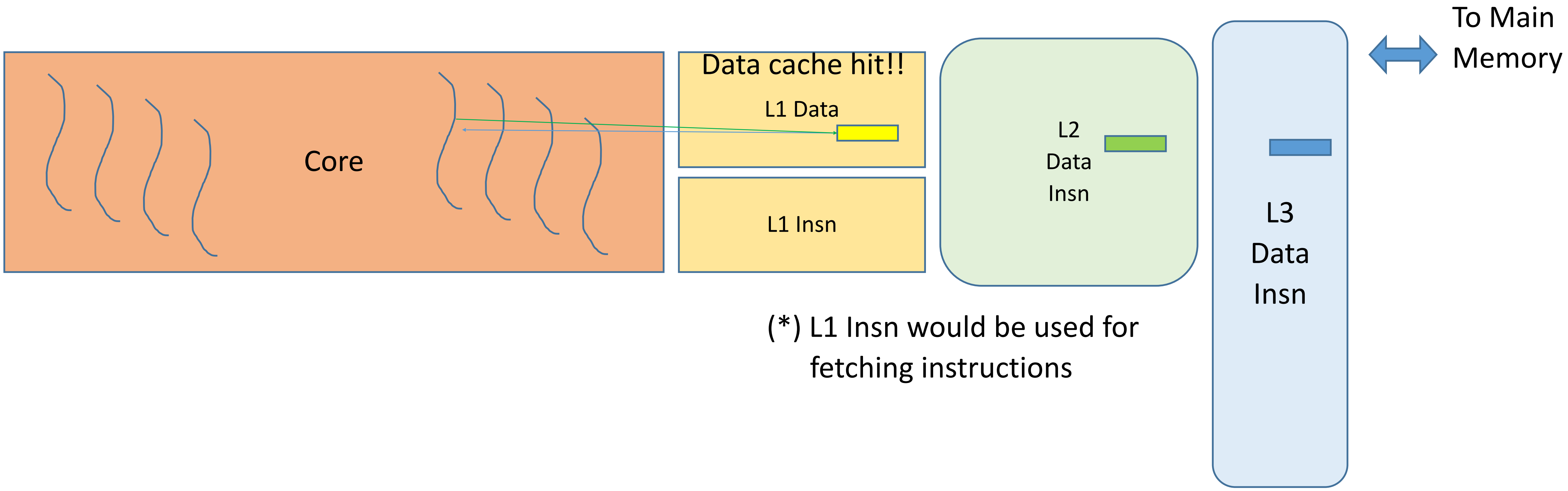
- ▶Load instruction, data is not on the caches
  - ▶Also, fetching instructions, instructions are not on the caches(*)



(*) L1 Insn would be used for fetching instructions

Detailed memory access

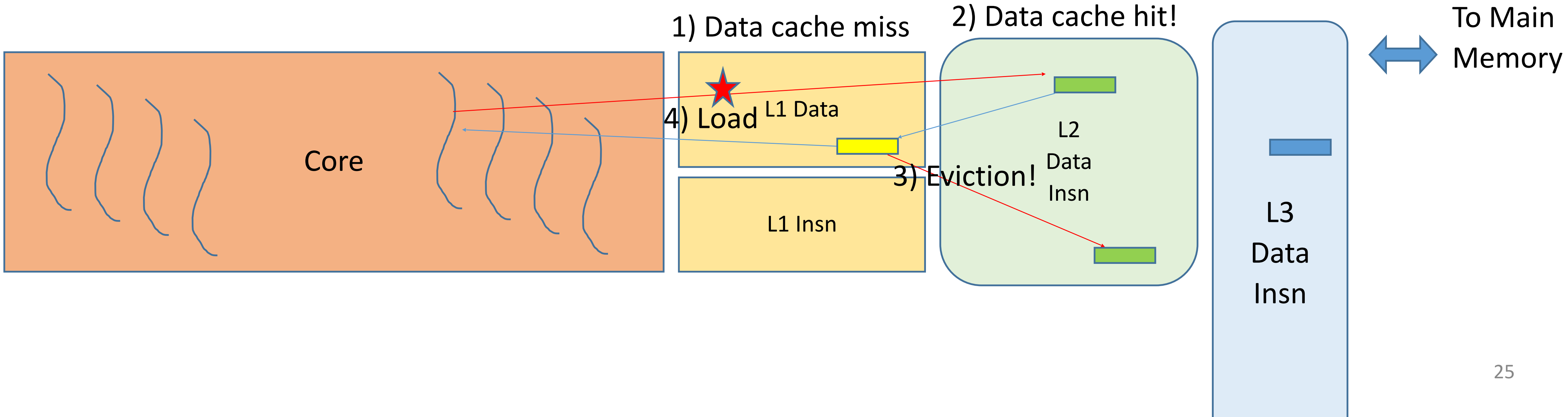
- ▶Load instruction, data present in L1 Data
  - ▶Also, fetching instructions, instruction present in L1 Insn (*)



(*) L1 Insn would be used for fetching instructions

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

```
def add_vectors(a,b,c):
    for i in range(0, N):
        c[i] = a[i] + b[i]
    return c
```

```
def mul_vectors(a,b,c):
    for i in range(0, N):
        c[i] = a[i] * b[i]
    return c
```

https://en.wikipedia.org/wiki/Consistency_model

Code generation details

```
_add_vectors:
        subq    $8, %rsp
        cmpl    $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
        cmpl    %eax, _N(%rip)
        jg      L5
L6:
        addq    $8, %rsp
        ret
```

prologue/ entering function

init index

load a

add/mul a, b

store c

inc index

compare index to N

epilogue/ leaving function

```
_mult_vectors:
        subq    $8, %rsp
        cmpl    $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
        cmpl    %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:
        movslq  %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
```

Processors / threads execute on a cycle by cycle basis

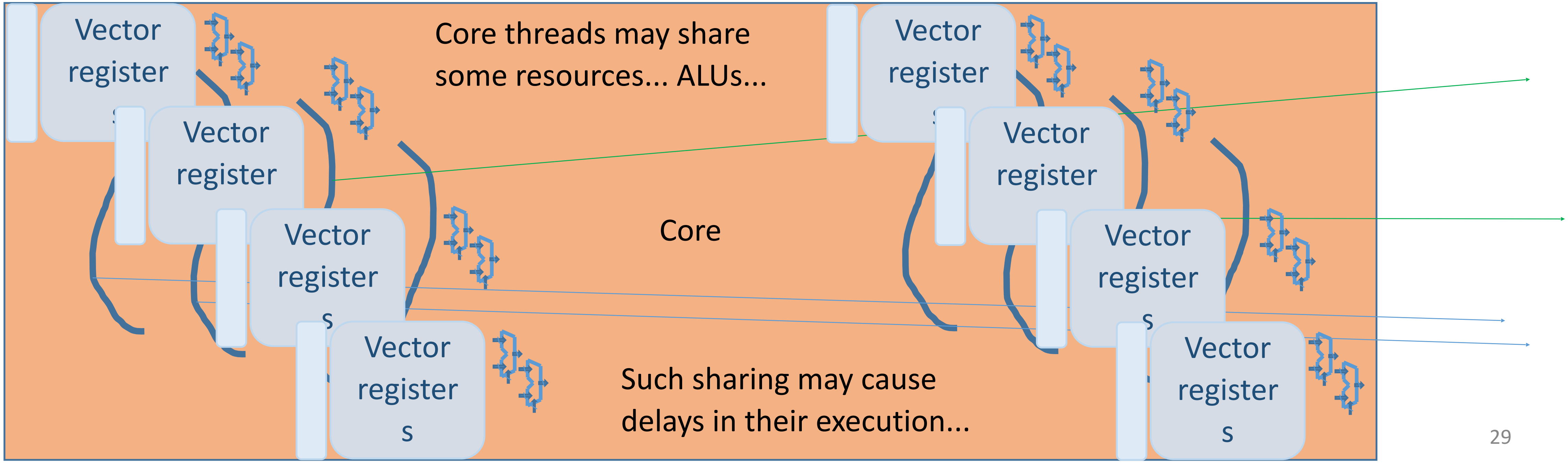
1.x – 2.x instructions per cycle

init index — immediate loads may take 1-10 cycles

load a	loads may take 1 – 200 cycles (L1 ... Main mem)
add a, b	stores may be less costly... Store buffer
store c	integer instructions 1-4 cycles
inc index	floating point instructions 8-30 cycles
compare index to N	jump instructions 1-20 cycles

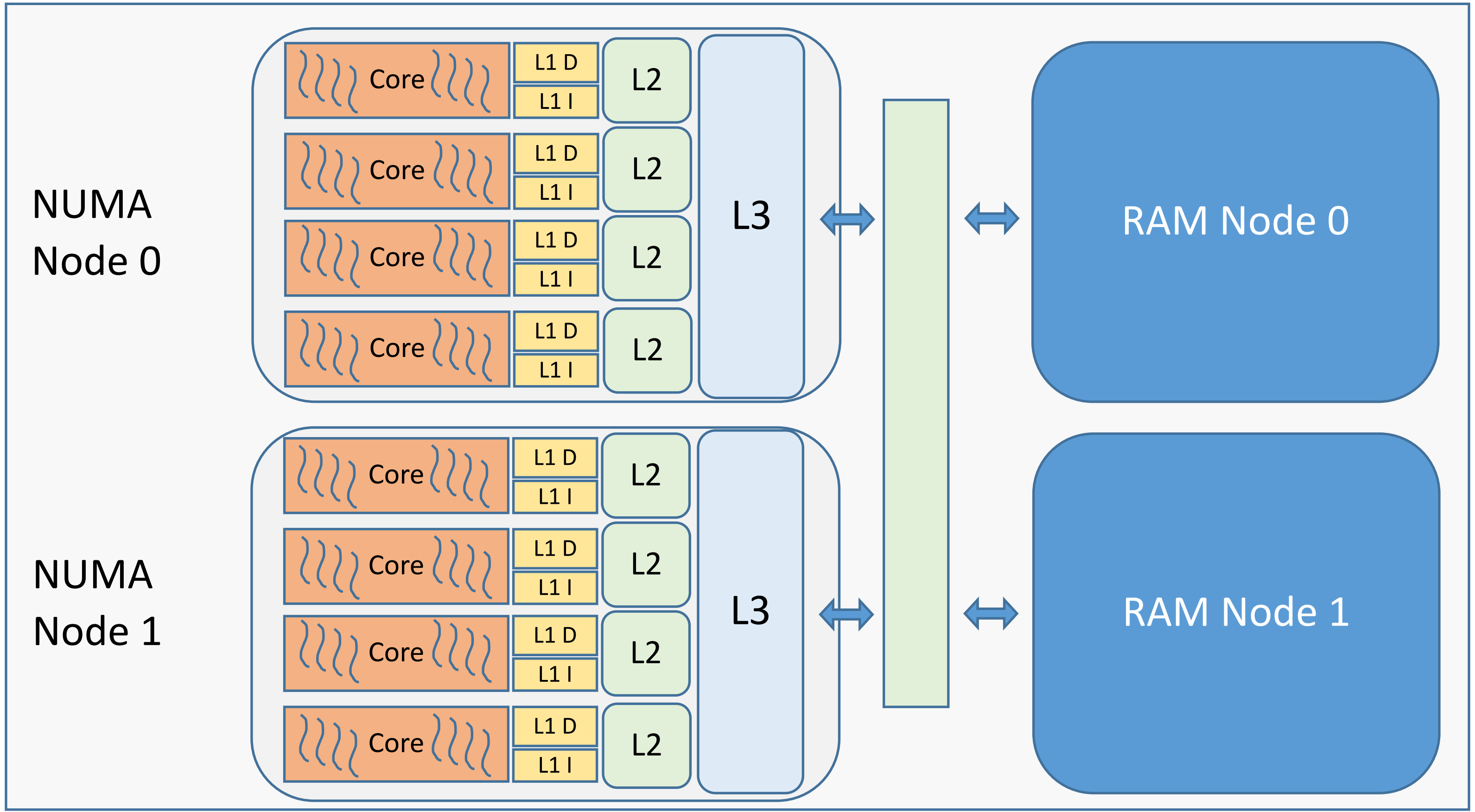
Core details

- 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



L1 Insn/Data	1 cycle
L2	15 cycles
L3	60 cycles
Main memory	
Local node	200 cycles
Remote node	250 cycles

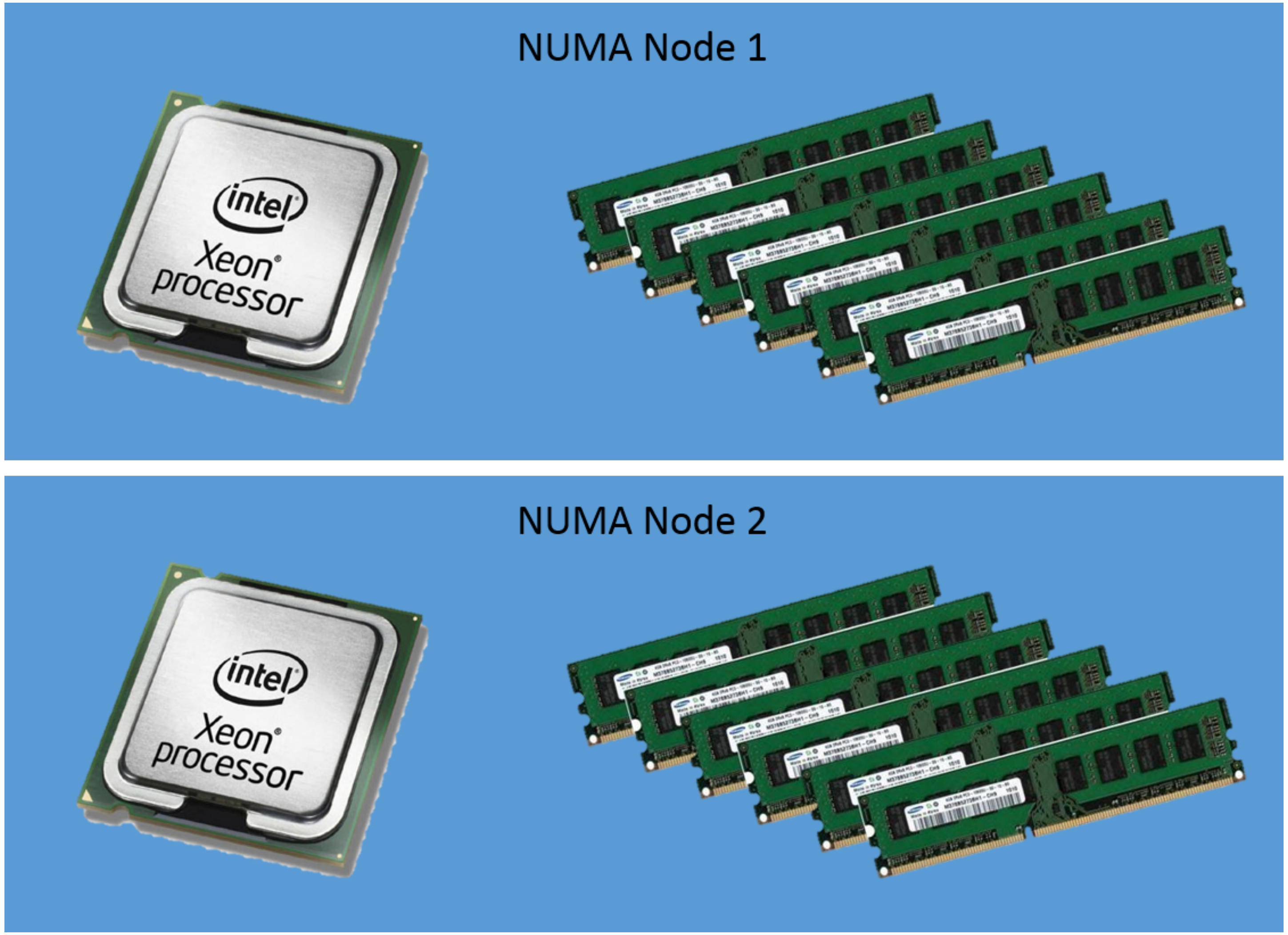
System board (no I/O yet)

https://richardjgreen.net/tags/hyper-v/

Example of multiprocessor motherboard

- Schematics
  - Two processors
  - Two memory nodes

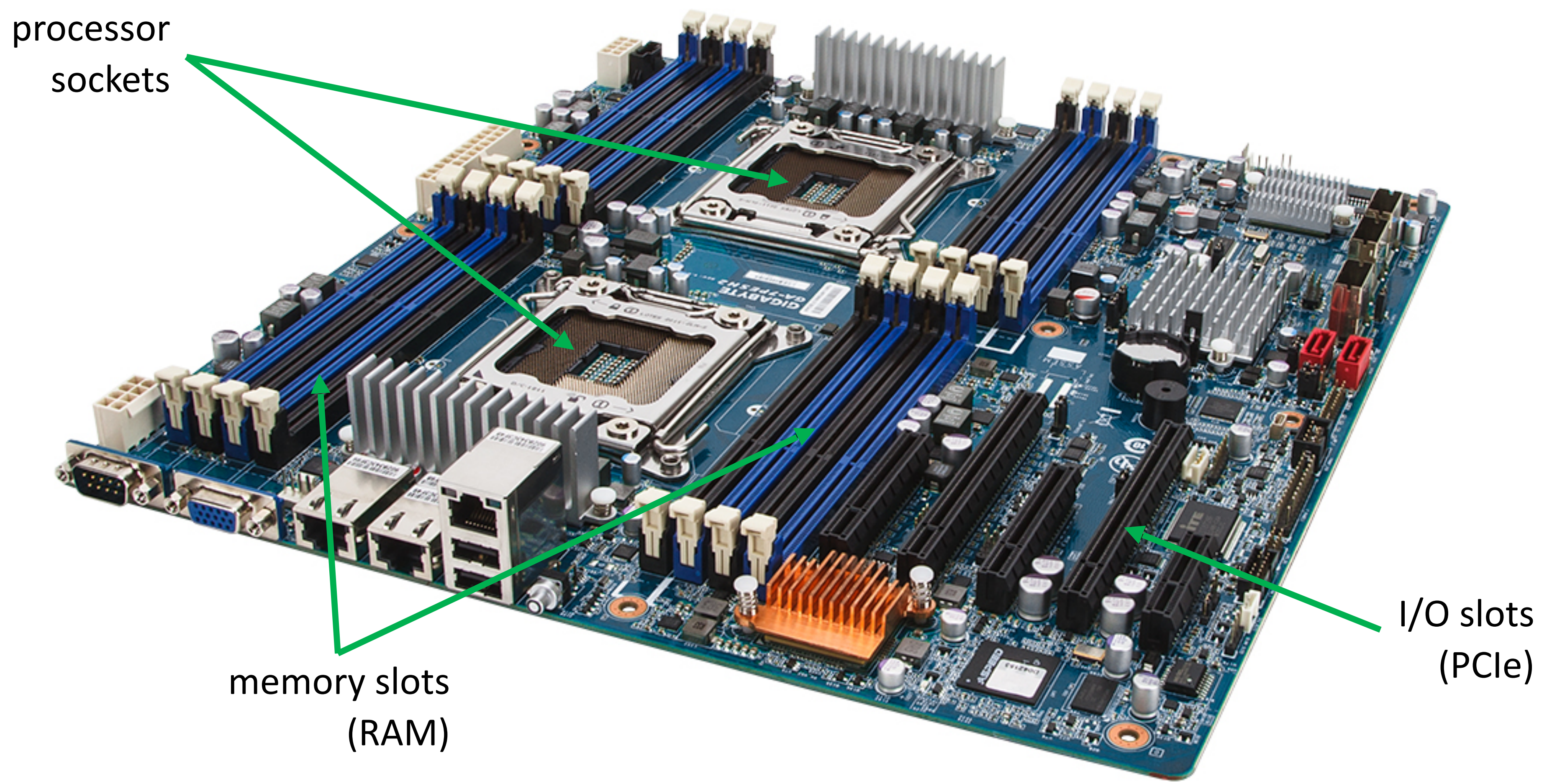
- Exercise... where are L1I, L1D, L2, L3?



https://community.mellanox.com/s/article/understanding-numa-node-for-performance-benchmarks

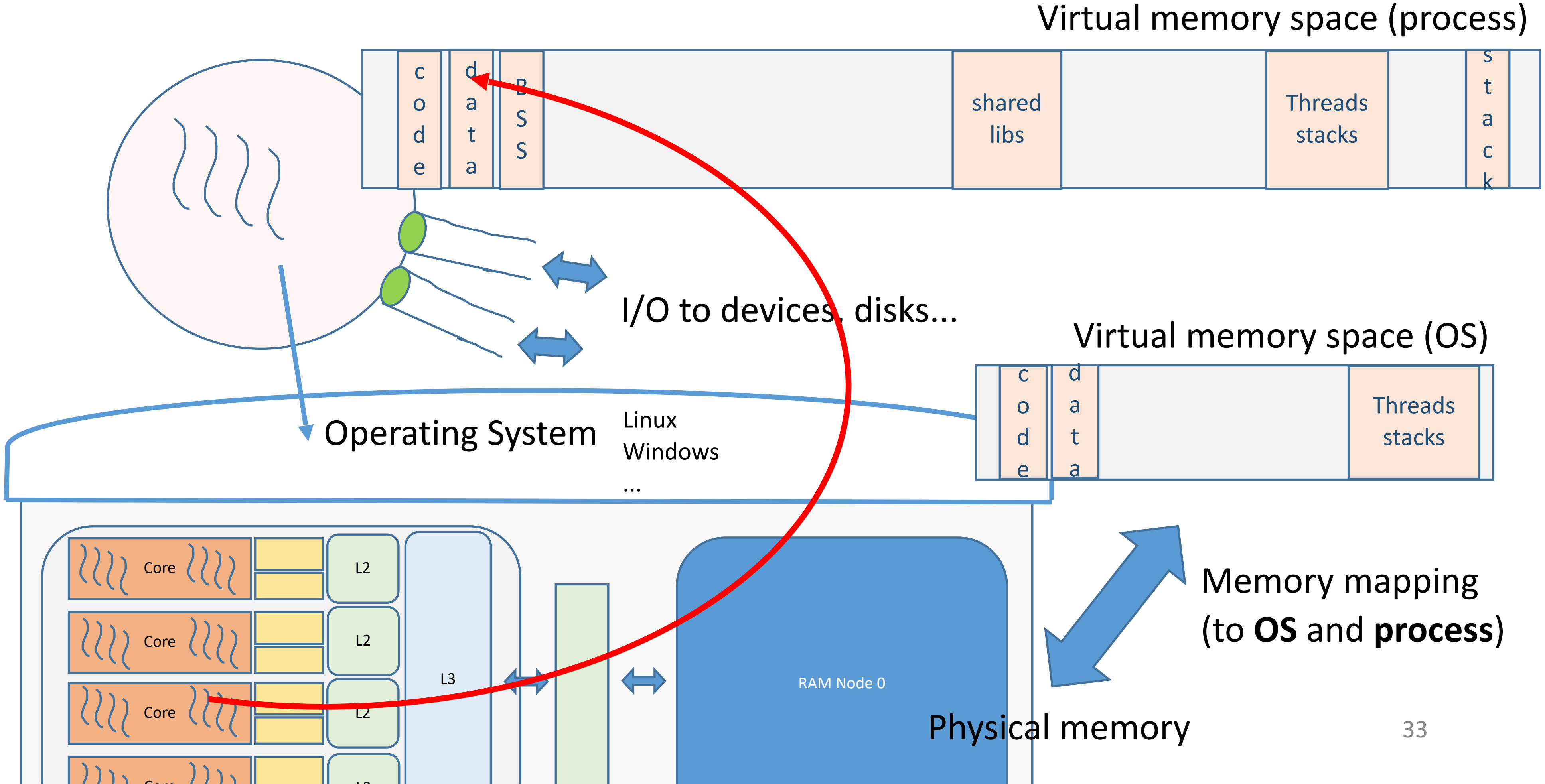
Example of multiprocessor motherboard

- Two processors and two memory nodes



Software/hardware mapping

- How the software uses this architecture?



Current processor chips

- Intel Xeon E7 v4 family
  - 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

https://ark.intel.com   Intel processor descriptions

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

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

  - 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

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)

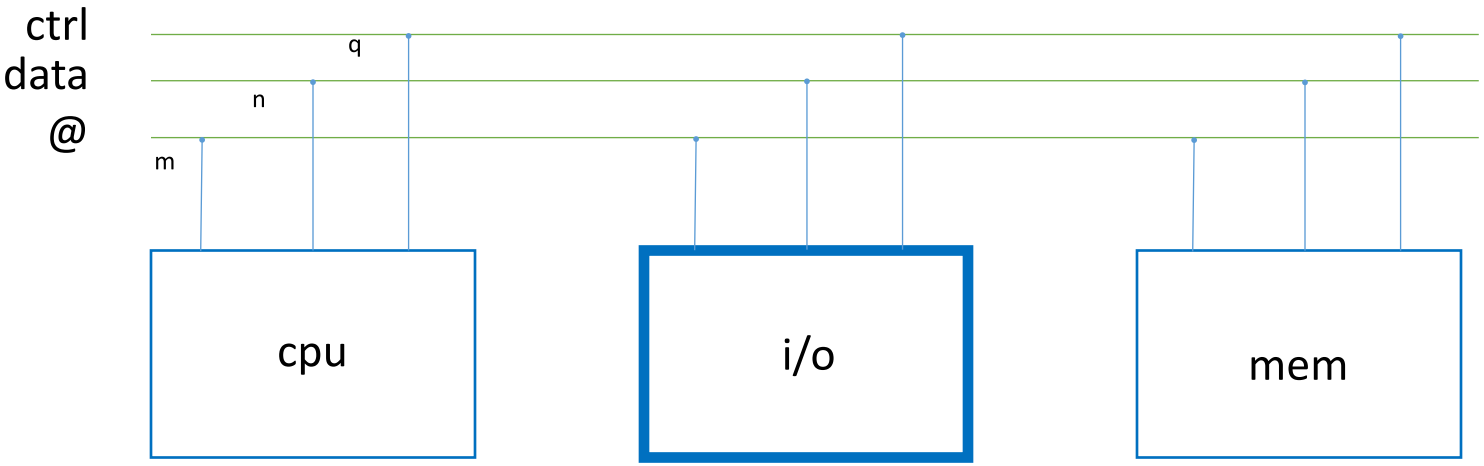
Current processor chips

- Apple M3
  - 3 nm technology
  - 4.05 GHz performance, 2.76 GHz efficiency.
  - aarch64 – ARMv8.6-A
  - 4 performance cores + 4 efficiency cores
  - L1 cache 192+128 KiB per performance core
  - L1 cache 128+64 KiB per efficiency core
  - L2 cache 16 MiB
  - RAM 8-24 GB
  - GPU 8-10 cores

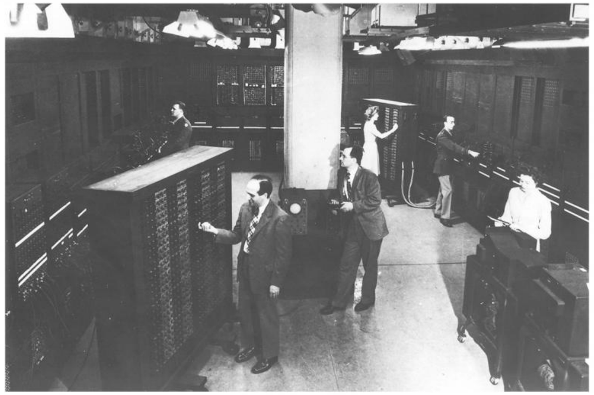
Apple unveils M3, M3Pro, and M3 Max

Computer organisation

- The so called *von Neumann* architecture



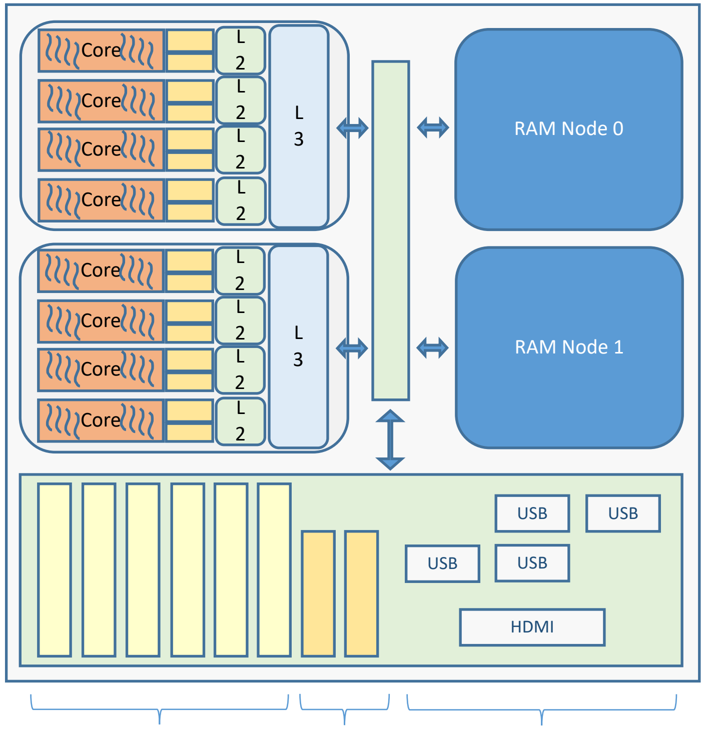
https://en.wikipedia.org/wiki/John_von_Neumann



https://www.fi.edu/case-files/mauchly-and-eckert

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

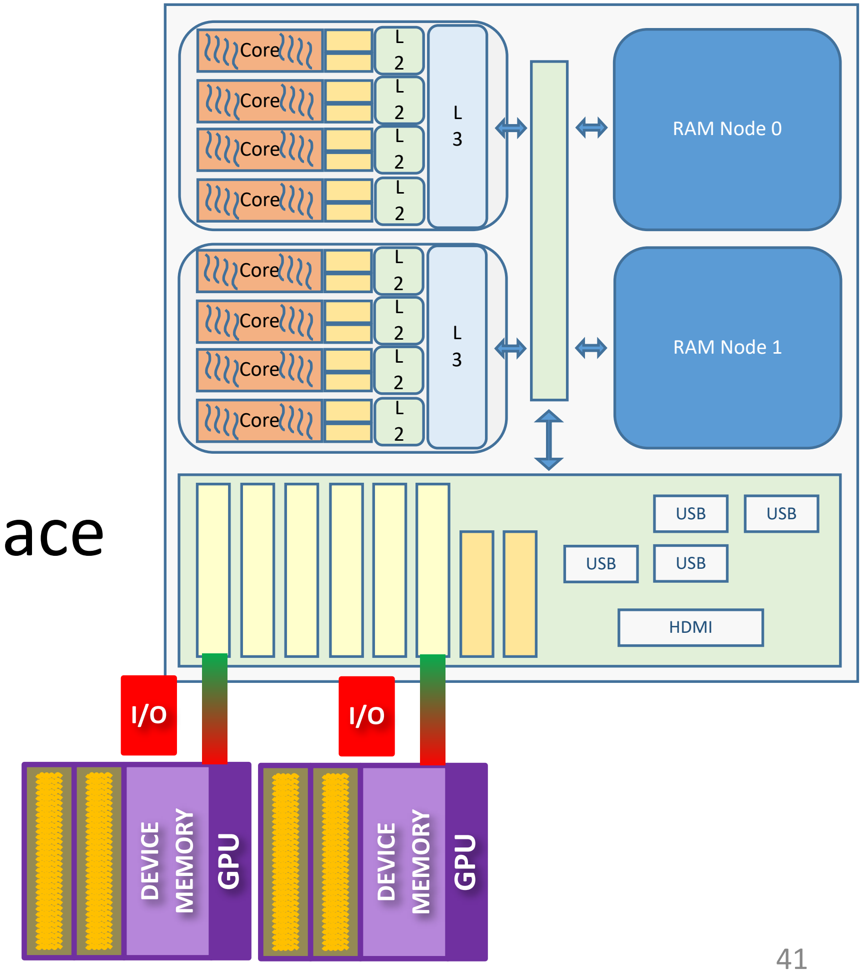
Sata
Disks

HMI Peripherals
Keyboard/mouse
Video
Audio

Accelerators

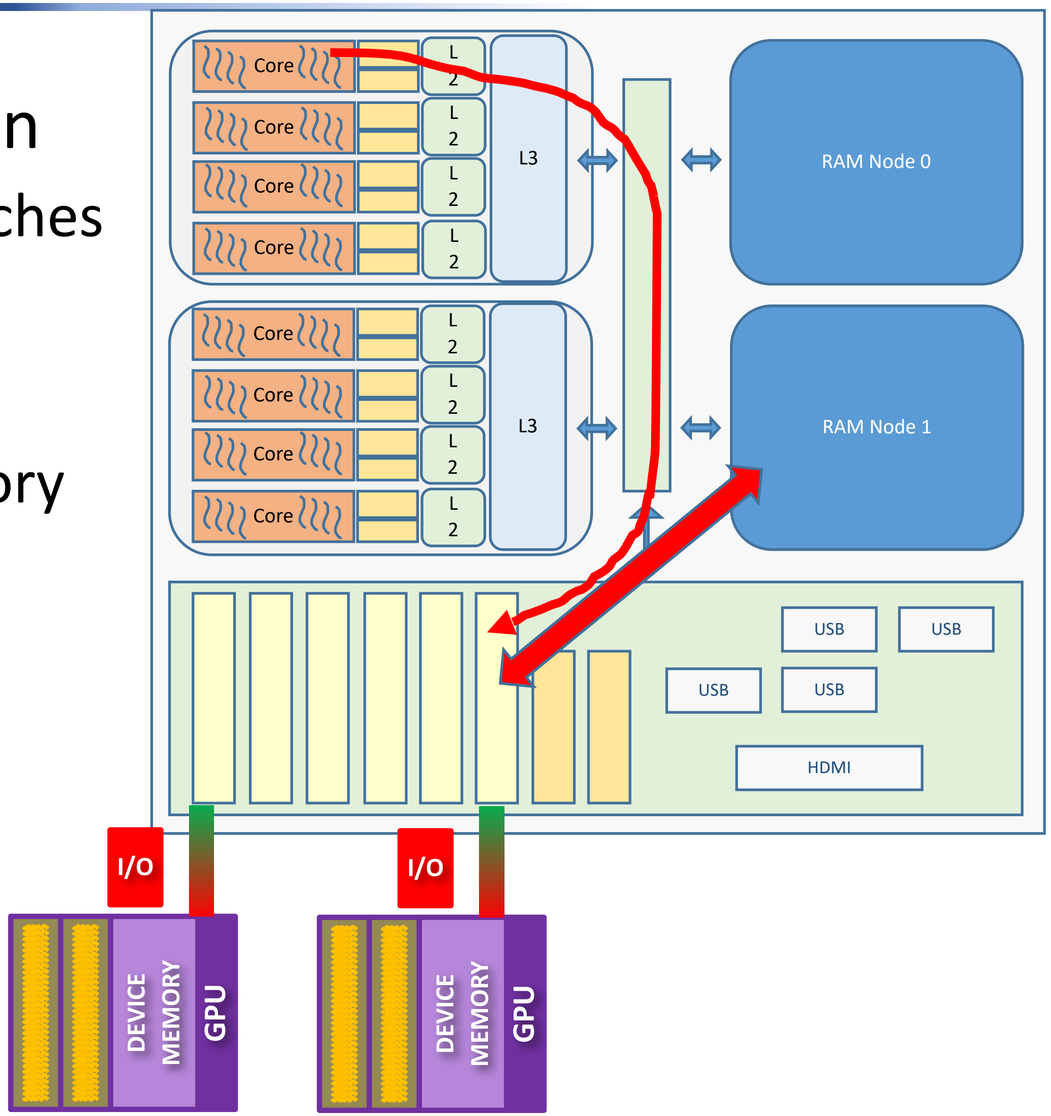
- Devices attached for computation
  - Need to transfer
    - code and data

    to/from the device
  - Need to start/stop execution
  - Synchronisation
- Configuration through mapped memory space
  - Use specific addresses to access the device's configuration registers
  - Access only allowed from the OS



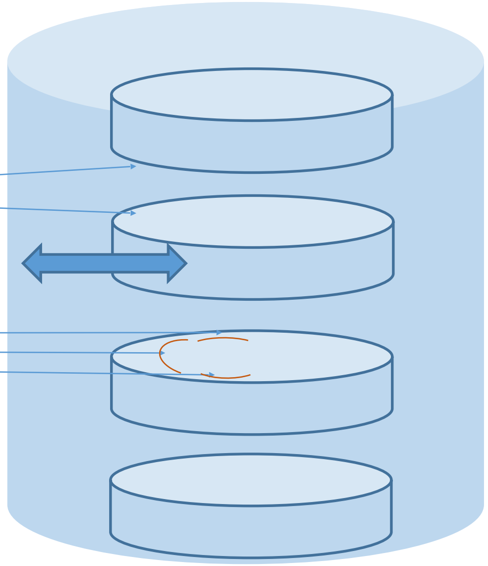
Access to accelerators/devices/peripherals

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

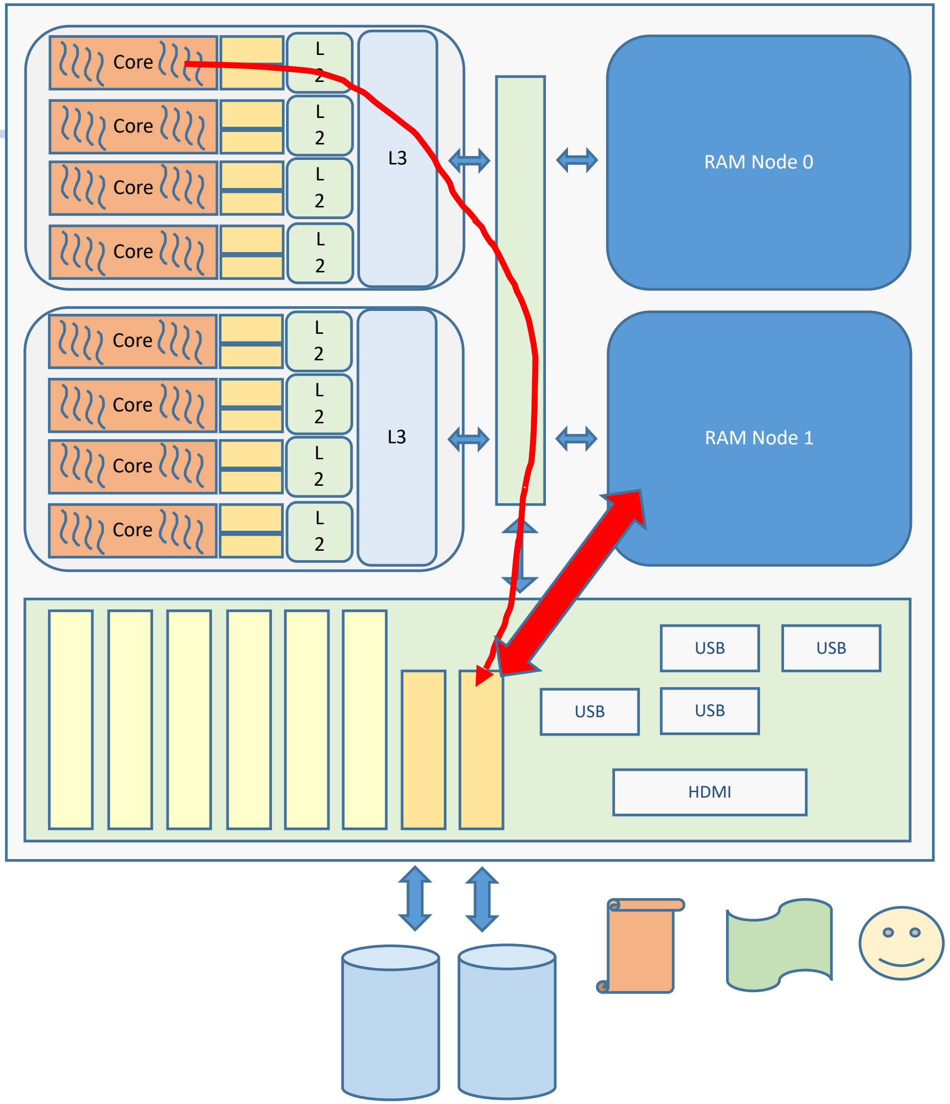


Sata and HMI Peripherals

- Disks
  - Addressed by
    - Head
    - Cylinder
    - Sector

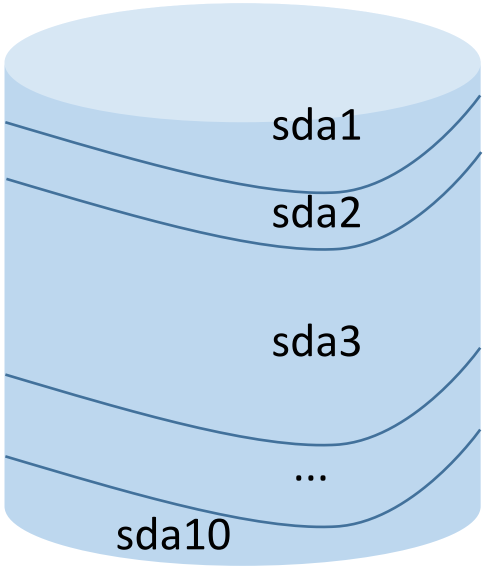


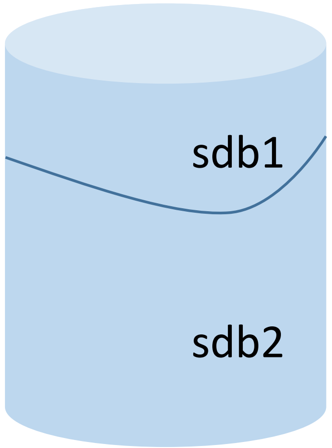
- Video
  - With special video memory
- USB
  - Keyboard, mouse...



Storage and file systems

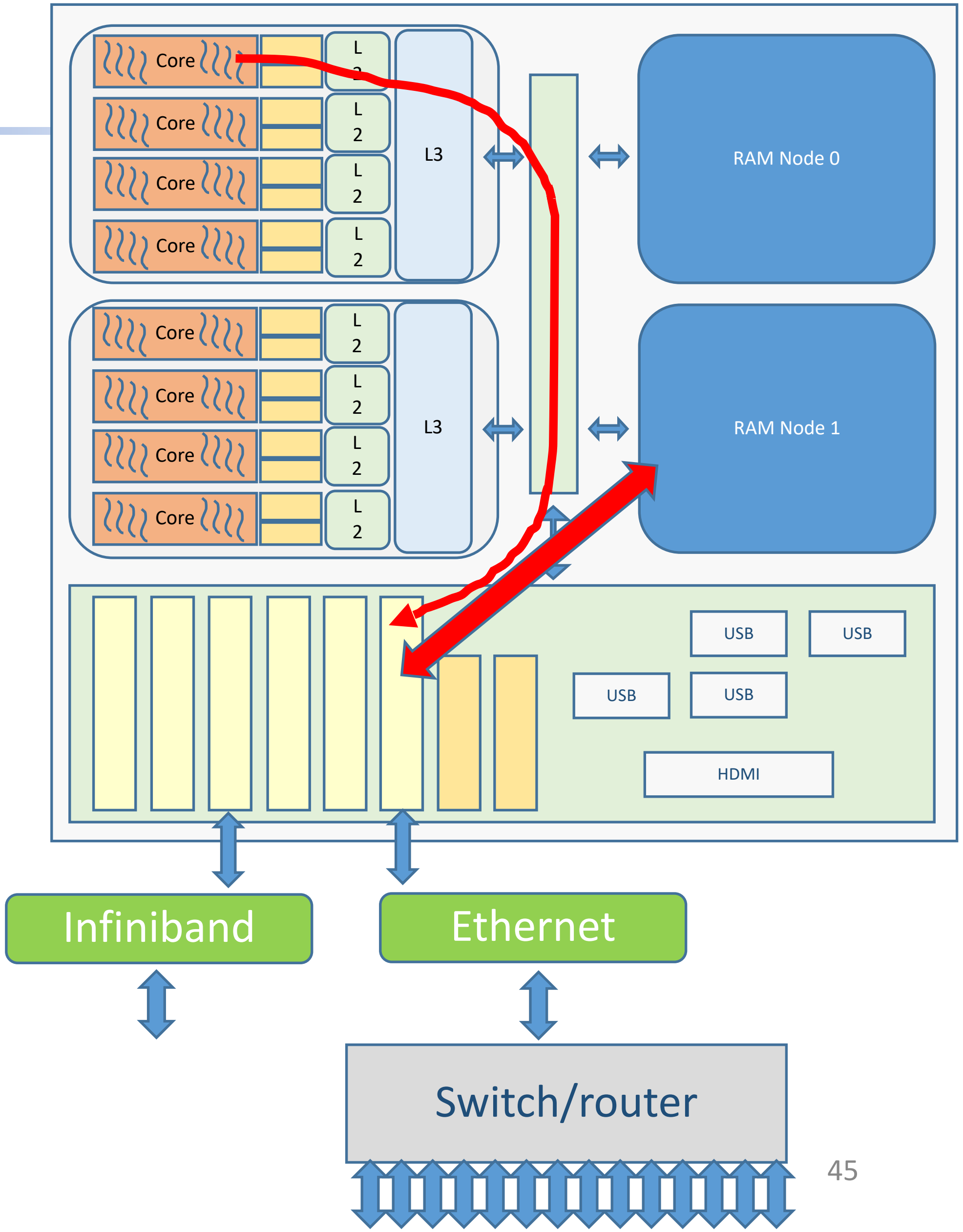
- Disks are usually split in partitions
  - Each partition can support a different file system
    - / (root)
    - /usr/local
    - /home
    - /opt
    - swap area
    - ...
  - Different types of file systems exist
    - Linux: ext4, ext3, ext2, btrfs, hfs, jfs, xfs...
    - Windows: ntfs, FAT, exFAT





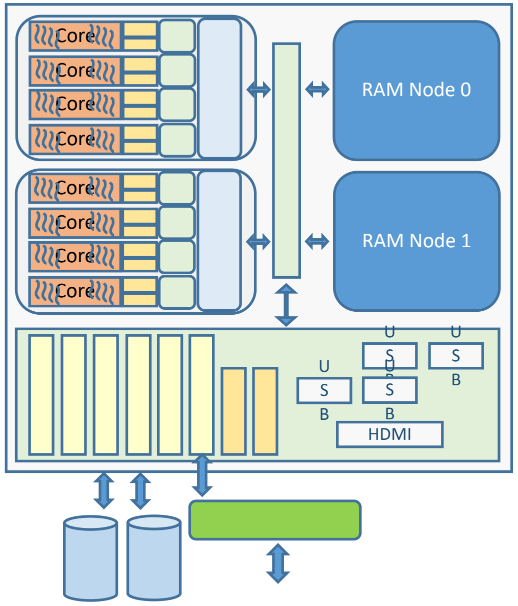
Networking

- 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

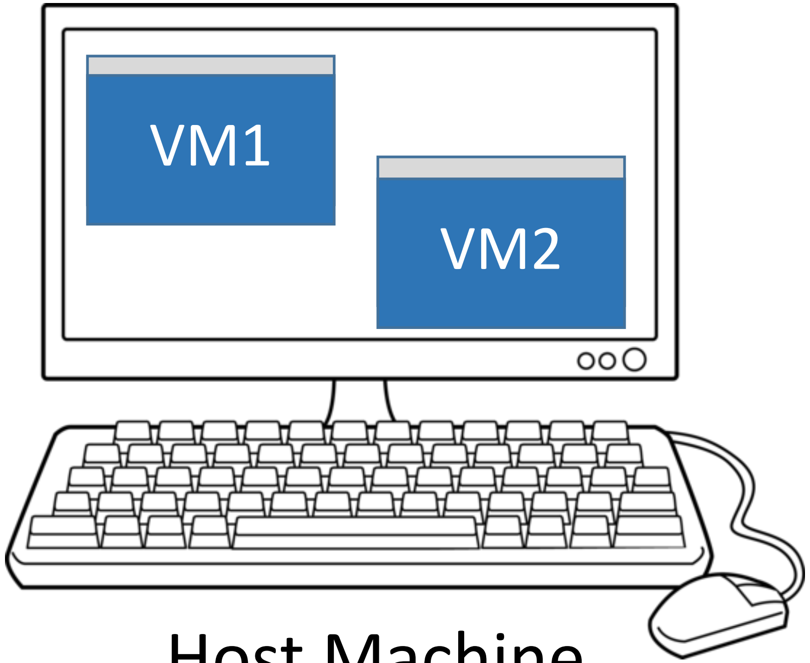


Virtual Machine (VM)

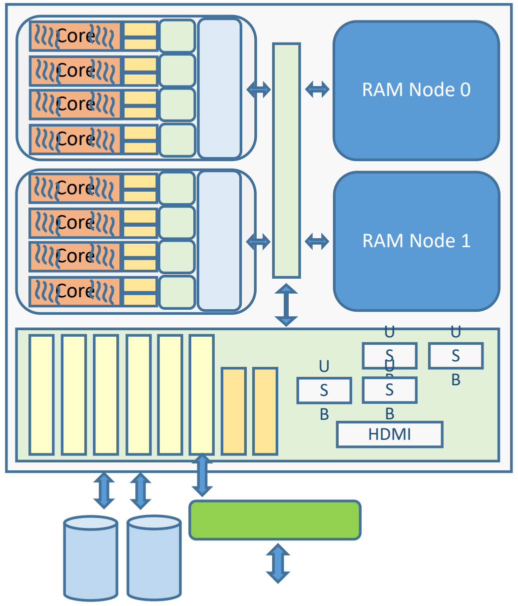
- A software package virtualises all physical resources of a computer
  - Virtualised resources can be emulated (simulates the real behaviour) or linked to access real resources of the host machine
  - Multiple VMs can run on the same host. Everyone can run a different Operating System



VM1



Host Machine



VM2

Bibliography

- Computer Organization and Design (6th Edition)
  - D. Patterson and J. Hennessy
  - https://cataleg.upf.edu/record=b1582632~S11*cat
    - Several chapters introduce different types of data
- Computer Systems. A programmer's perspective
  - Randal E. Bryant, David R. O'Hallaron
  - https://upfinder.upf.edu/iii/encore/record/C__Rb1318766
    - Chapters 4,5,6