Memory Systems

Memory hierarchy, cache memory, virtual memory

Gloria Beraldo (gloria.beraldo@dei.unipd.it)
Department of Information Engineering, University of Padova

Topics:

- Memory hierarchy and access time
- Locality of reference
- Types of cache memories and their usage
- Virtual memory

Book reference:

- Chapter 12

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

Writing methods in the cache

- So far we have considered the words in the cache as copies of those in main memory used to speed up the reading access.
- However, the matter changes if we also want to write a word that obviously cannot be written only in the cache

- **Write-through method:**
  - The result is always written to both the cache and the main memory
  - Symmetry between cache and central memory
  - It can slow down operations

- **Copy-back method:**
  - The result is written in the cache in case of Cache Hit
  - The result is written in the memory in case of Cache Miss
  - No symmetry between cache ←→ memory, some locations in the cache may no be longer valid

Dirty bit to control the writing

- In the case of the copy-back method, it is neccesary to keep track whether an item has been written or not in the cache
- Why?
  - To write in the memory only when it is necessary and not every time there is a Cache Miss
- A control bit, called **dirty bit,** is used:
  - If the dirty bit is equal to 1, it means that the LINE in cache has been written (and must also be written in memory)
  - If the dirty bit is equal to 0, it means you don’t need to write in the memory

Dirty bit & Copy-back method

Consider the following basic program below and we suppose to have a fully associative cache with 2 lines and 4 words per line

PROGRAM

LD	R0	R1	
DEC	R5	R5	
ST	R3	R5	
INC	R1	R1	
ADD	R3	R3	R4
LD	R6	R2	
ADI	R2	R2	4
BRZ	R5	2	
JMP	R6		
INC	R7	R2	

REGISTERS

R0	R1	R2	R3	R4	R5	R6	R7
12	3	12	3	5	4	0	0

RAM

0x0000	4
0x0001	43
0x0002	54
0x0003	73
0x0004	22
0x0005	34
0x0006	5
0x0007	23
0x0008	67
0x0009	12
0x000A	3
0x000B	12
0x000C	4
0x000D	34
0x000E	23
0x000F	240
0x0010	5
0x0011	65
0x0012	234
0x0013	23
0x0014	67
0x0015	78
0x0016	18
0x0017	72
0x0018	52
0x0019	12
0x001A	42
0x001B	8
0x001C	35
0x001D	49
0x001E	51
0x001F	86

Dirty bit & Copy-back method

Consider the following basic program below and we suppose to have a fully associative cache with 2 lines and 4 words per line

PROGRAM

LD	R0	R1	
DEC	R5	R5	
ST	R3	R5	
INC	R1	R1	
ADD	R3	R3	R4
LD	R6	R2	
ADI	R2	R2	4
BRZ	R5	2	
JMP	R6		
INC	R7	R2	

Notice that the LD and ST instructions require the access to the memory (cache and/or RAM)

The following slides show only a partial execution focusing on these instructions.

Check the whole execution in the .xlsx file in the course page

Dirty bit & Copy-back method

Consider the following basic program below and we suppose to have a fully associative cache with 2 lines and 4 words per line that uses FIFO policy as replacement method.

WE DESIGN THE CACHE

NB: No direct mapping between cache and RAM (i.e., no index only TAG) because it is a fully associative cache

RAM

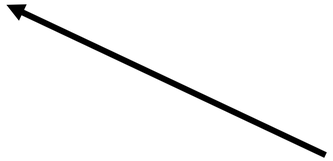
	0x0000	4
	0x0001	43
	0x0002	54
	0x0003	73
	0x0004	22
	0x0005	34
	0x0006	5
	0x0007	23
	0x0008	67
	0x0009	12
	0x000A	3
	0x000B	12
	0x000C	4
	0x000D	34
	0x000E	23
	0x000F	240
	0x0010	5
	0x0011	65
	0x0012	234
	0x0013	23
	0x0014	67
	0x0015	78
	0x0016	18
	0x0017	72
	0x0018	52
	0x0019	12
	0x001A	42
	0x001B	8
	0x001C	35
	0x001D	49
	0x001E	51
	0x001F	86

Dirty bit & Copy-back method

Consider the following basic program below and we suppose to have a fully associative cache with 2 lines and 4 words per line that uses FIFO policy as replacement method.

PROGRAM

LD	R0	R1	
DEC	R5	R5	
ST	R3	R5	
INC	R1	R1	
ADD	R3	R3	R4
LD	R6	R2	
ADI	R2	R2	4
BRZ	R5	2	
JMP	R6		
INC	R7	R2	



Current instruction

RAM

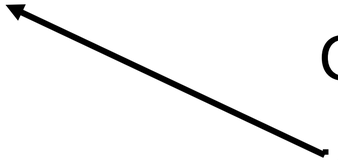
	0x0000	4
	0x0001	43
	0x0002	54
	0x0003	73
	0x0004	22
	0x0005	34
	0x0006	5
	0x0007	23
	0x0008	67
	0x0009	12
	0x000A	3
	0x000B	12
	0x000C	4
	0x000D	34
	0x000E	23
	0x000F	240
	0x0010	5
	0x0011	65
	0x0012	234
	0x0013	23
	0x0014	67
	0x0015	78
	0x0016	18
	0x0017	72
	0x0018	52
	0x0019	12
	0x001A	42
	0x001B	8
	0x001C	35
	0x001D	49
	0x001E	51
	0x001F	86

Dirty bit & Copy-back method

Consider the following basic program below and we suppose to have a fully associative cache with 2 lines and 4 words per line that uses FIFO policy as replacement method.

PROGRAM

LD	R0	R1		
DEC	R5	R5		
ST	R3	R5		
INC	R1	R1		
ADD	R3	R3	R4	
LD	R6	R2		
ADI	R2	R2		4
BRZ	R5		2	
JMP	R6			
INC	R7	R2		



Current instruction

R0= M[R1]

R0 = M[3] = 73

MISS!

4	43	54	73

RAM

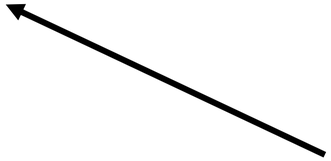
	0x0000	4
	0x0001	43
	0x0002	54
	0x0003	73
	0x0004	22
	0x0005	34
	0x0006	5
	0x0007	23
	0x0008	67
	0x0009	12
	0x000A	3
	0x000B	12
	0x000C	4
	0x000D	34
	0x000E	23
	0x000F	240
	0x0010	5
	0x0011	65
	0x0012	234
	0x0013	23
	0x0014	67
	0x0015	78
	0x0016	18
	0x0017	72
	0x0018	52
	0x0019	12
	0x001A	42
	0x001B	8
	0x001C	35
	0x001D	49
	0x001E	51
	0x001F	86

Dirty bit & Copy-back method

Consider the following basic program below and we suppose to have a fully associative cache with 2 lines and 4 words per line that uses FIFO policy as replacement method.

PROGRAM

LD	R0	R1	
DEC	R5	R5	
ST	R3	R5	
INC	R1	R1	
ADD	R3	R3	R4
LD	R6	R2	
ADI	R2	R2	4
BRZ	R5	2	
JMP	R6		
INC	R7	R2	



Current instruction

4	43	54	73

RAM

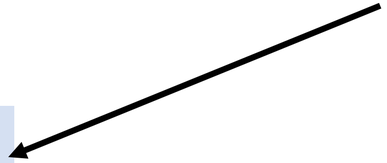
	0x0000	4
	0x0001	43
	0x0002	54
	0x0003	73
	0x0004	22
	0x0005	34
	0x0006	5
	0x0007	23
	0x0008	67
	0x0009	12
	0x000A	3
	0x000B	12
	0x000C	4
	0x000D	34
	0x000E	23
	0x000F	240
	0x0010	5
	0x0011	65
	0x0012	234
	0x0013	23
	0x0014	67
	0x0015	78
	0x0016	18
	0x0017	72
	0x0018	52
	0x0019	12
	0x001A	42
	0x001B	8
	0x001C	35
	0x001D	49
	0x001E	51
	0x001F	86

Dirty bit & Copy-back method

Consider the following basic program below and we suppose to have a fully associative cache with 2 lines and 4 words per line that uses FIFO policy as replacement method.

PROGRAM

LD	R0	R1	
DEC	R5	R5	
ST	R3	R5	
INC	R1	R1	
ADD	R3	R3	R4
LD	R6	R2	
ADI	R2	R2	4
BRZ	R5	2	
JMP	R6		
INC	R7	R2	



Current instruction

M[R3]= R5

M[3] = 3

HIT!

Write only in the cache

Dirty_bit = 1

4	43	54	3

RAM

	0x0000	4
	0x0001	43
	0x0002	54
	0x0003	73
	0x0004	22
	0x0005	34
	0x0006	5
	0x0007	23
	0x0008	67
	0x0009	12
	0x000A	3
	0x000B	12
	0x000C	4
	0x000D	34
	0x000E	23
	0x000F	240
	0x0010	5
	0x0011	65
	0x0012	234
	0x0013	23
	0x0014	67
	0x0015	78
	0x0016	18
	0x0017	72
	0x0018	52
	0x0019	12
	0x001A	42
	0x001B	8
	0x001C	35
	0x001D	49
	0x001E	51
	0x001F	86

Dirty bit & Copy-back method

Consider the following basic program below and we suppose to have a fully associative cache with 2 lines and 4 words per line that uses FIFO policy as replacement method.

PROGRAM

LD	R0	R1	
DEC	R5	R5	
ST	R3	R5	
INC	R1	R1	
ADD	R3	R3	R4
LD	R6	R2	
ADI	R2	R2	4
BRZ	R5	2	
JMP	R6		
INC	R7	R2	

Current instruction

4	43	54	3

Dirty_bit = 1

RAM

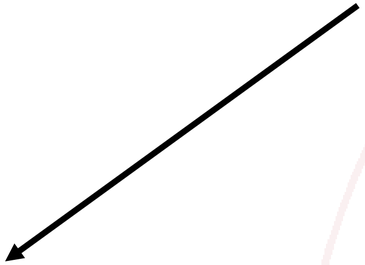
	0x0000	4
	0x0001	43
	0x0002	54
	0x0003	73
	0x0004	22
	0x0005	34
	0x0006	5
	0x0007	23
	0x0008	67
	0x0009	12
	0x000A	3
	0x000B	12
	0x000C	4
	0x000D	34
	0x000E	23
	0x000F	240
	0x0010	5
	0x0011	65
	0x0012	234
	0x0013	23
	0x0014	67
	0x0015	78
	0x0016	18
	0x0017	72
	0x0018	52
	0x0019	12
	0x001A	42
	0x001B	8
	0x001C	35
	0x001D	49
	0x001E	51
	0x001F	86

Dirty bit & Copy-back method

Consider the following basic program below and we suppose to have a fully associative cache with 2 lines and 4 words per line that uses FIFO policy as replacement method.

PROGRAM

LD	R0	R1	
DEC	R5	R5	
ST	R3	R5	
INC	R1	R1	
ADD	R3	R3	R4
LD	R6	R2	
ADI	R2	R2	4
BRZ	R5	2	
JMP	R6		
INC	R7	R2	



Current instruction

R6= M[R2]

R6 = M[12] = 4

MISS!

4	34	23	240
4	43	54	3

Dirty_bit = 1

RAM

	0x0000	4
	0x0001	43
	0x0002	54
	0x0003	73
	0x0004	22
	0x0005	34
	0x0006	5
	0x0007	23
	0x0008	67
	0x0009	12
	0x000A	3
	0x000B	12
	0x000C	4
	0x000D	34
	0x000E	23
	0x000F	240
	0x0010	5
	0x0011	65
	0x0012	234
	0x0013	23
	0x0014	67
	0x0015	78
	0x0016	18
	0x0017	72
	0x0018	52
	0x0019	12
	0x001A	42
	0x001B	8
	0x001C	35
	0x001D	49
	0x001E	51
	0x001F	86

Dirty bit & Copy-back method

Consider the following basic program below and we suppose to have a fully associative cache with 2 lines and 4 words per line that uses FIFO policy as replacement method.

PROGRAM

LD	R0	R1		
DEC	R5	R5		
ST	R3	R5		
INC	R1	R1		
ADD	R3	R3	R4	
LD	R6	R2		
ADI	R2	R2		4
BRZ	R5		2	
JMP	R6			
INC	R7	R2		

Current instruction

4	34	23	240
4	43	54	3

Dirty_bit = 1

RAM

	0x0000	4
	0x0001	43
	0x0002	54
	0x0003	73
	0x0004	22
	0x0005	34
	0x0006	5
	0x0007	23
	0x0008	67
	0x0009	12
	0x000A	3
	0x000B	12
	0x000C	4
	0x000D	34
	0x000E	23
	0x000F	240
	0x0010	5
	0x0011	65
	0x0012	234
	0x0013	23
	0x0014	67
	0x0015	78
	0x0016	18
	0x0017	72
	0x0018	52
	0x0019	12
	0x001A	42
	0x001B	8
	0x001C	35
	0x001D	49
	0x001E	51
	0x001F	86

Dirty bit & Copy-back method

Consider the following basic program below and we suppose to have a fully associative cache with 2 lines and 4 words per line that uses FIFO policy as replacement method.

PROGRAM

LD	R0	R1		
DEC	R5	R5		
ST	R3	R5		
INC	R1	R1		
ADD	R3	R3	R4	
LD	R6	R2		
ADI	R2	R2		4
BRZ	R5		2	
JMP	R6			
INC	R7	R2		

Current instruction

R0= M[R1]

R0 = M[4] = 22

REPLACEMENT

NB: We need to update RAM first, dirty bit is 1

4	34	23	240
4	43	54	3

Dirty_bit = 1

RAM

	0x0000	4
	0x0001	43
	0x0002	54
	0x0003	73
	0x0004	22
	0x0005	34
	0x0006	5
	0x0007	23
	0x0008	67
	0x0009	12
	0x000A	3
	0x000B	12
	0x000C	4
	0x000D	34
	0x000E	23
	0x000F	240
	0x0010	5
	0x0011	65
	0x0012	234
	0x0013	23
	0x0014	67
	0x0015	78
	0x0016	18
	0x0017	72
	0x0018	52
	0x0019	12
	0x001A	42
	0x001B	8
	0x001C	35
	0x001D	49
	0x001E	51
	0x001F	86

3

Dirty bit & Copy-back method

Consider the following basic program below and we suppose to have a fully associative cache with 2 lines and 4 words per line that uses FIFO policy as replacement method.

PROGRAM

LD	R0	R1		
DEC	R5	R5		
ST	R3	R5		
INC	R1	R1		
ADD	R3	R3	R4	
LD	R6	R2		
ADI	R2	R2		4
BRZ	R5		2	
JMP	R6			
INC	R7	R2		

Current instruction

R0= M[R1]

R0 = M[4] = 22

REPLACEMENT

FIFO policy as replacement method

4	34	23	240
22	34	5	23

Dirty_bit = 0

RAM

	0x0000	4
	0x0001	43
	0x0002	54
	0x0003	73
	0x0004	22
	0x0005	34
	0x0006	5
	0x0007	23
	0x0008	67
	0x0009	12
	0x000A	3
	0x000B	12
	0x000C	4
	0x000D	34
	0x000E	23
	0x000F	240
	0x0010	5
	0x0011	65
	0x0012	234
	0x0013	23
	0x0014	67
	0x0015	78
	0x0016	18
	0x0017	72
	0x0018	52
	0x0019	12
	0x001A	42
	0x001B	8
	0x001C	35
	0x001D	49
	0x001E	51
	0x001F	86

Dirty bit & Copy-back method

Consider the following basic program below and we suppose to have a fully associative cache with 2 lines and 4 words per line that uses FIFO policy as replacement method.

PROGRAM

LD	R0	R1		
DEC	R5	R5		
ST	R3	R5		
INC	R1	R1		
ADD	R3	R3	R4	
LD	R6	R2		
ADI	R2	R2		4
BRZ	R5		2	
JMP	R6			
INC	R7	R2		

Current instruction

4	34	23	240
22	34	5	23

Dirty_bit = 0

RAM

	0x0000	4
	0x0001	43
	0x0002	54
	0x0003	73
	0x0004	22
	0x0005	34
	0x0006	5
	0x0007	23
	0x0008	67
	0x0009	12
	0x000A	3
	0x000B	12
	0x000C	4
	0x000D	34
	0x000E	23
	0x000F	240
	0x0010	5
	0x0011	65
	0x0012	234
	0x0013	23
	0x0014	67
	0x0015	78
	0x0016	18
	0x0017	72
	0x0018	52
	0x0019	12
	0x001A	42
	0x001B	8
	0x001C	35
	0x001D	49
	0x001E	51
	0x001F	86

Dirty bit & Copy-back method

Consider the following basic program below and we suppose to have a fully associative cache with 2 lines and 4 words per line that uses FIFO policy as replacement method.

PROGRAM

LD	R0	R1		
DEC	R5	R5		
ST	R3	R5		
INC	R1	R1		
ADD	R3	R3	R4	
LD	R6	R2		
ADI	R2	R2		4
BRZ	R5		2	
JMP	R6			
INC	R7	R2		

Current instruction

M[R3]= R5

M[8] = 2

MISS!

Write only in the RAM

4	34	23	240
22	34	5	23

RAM

	0x0000	4
	0x0001	43
	0x0002	54
	0x0003	73
	0x0004	22
	0x0005	34
	0x0006	5
	0x0007	23
	0x0008	67
	0x0009	12
	0x000A	3
	0x000B	12
	0x000C	4
	0x000D	34
	0x000E	23
	0x000F	240
	0x0010	5
	0x0011	65
	0x0012	234
	0x0013	23
	0x0014	67
	0x0015	78
	0x0016	18
	0x0017	72
	0x0018	52
	0x0019	12
	0x001A	42
	0x001B	8
	0x001C	35
	0x001D	49
	0x001E	51
	0x001F	86

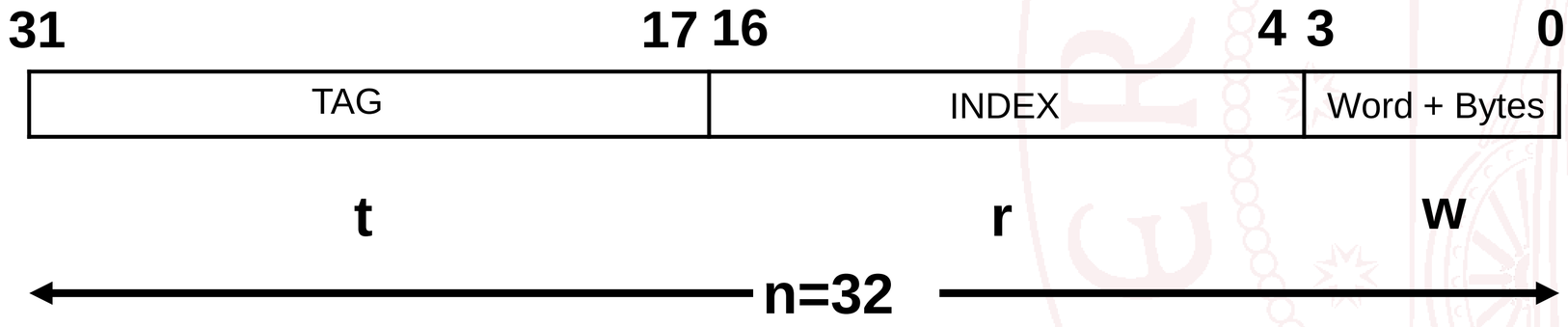
2

Example of 2-way set-associative with 4-word lines: write-through method

- A 256 KB cache memory, 4 words for the lines and the address is composed of 32 bits
- $N_{bytes} = 256 \cdot 1024 = 262144\ bytes$
  - $N_{bytes \times word} = 4\ bytes$ → each word contains 4 bytes
- $N_{words} = \frac{N_{bytes}}{N_{bytes \times word}} = \frac{262144}{4} = 65536\ words$
- $N_{bytes \times line} = 16$ → $N_{words \times line} = 4$
- $N_{lines} = \frac{N_{words}}{N_{words \times line}} = \frac{65536}{4} = 16384$
- $N_{ways} = 2$ → $N_{lines \times way} = 8192$
- $n - w - r = t = 32 - 4 - 13 = 15$

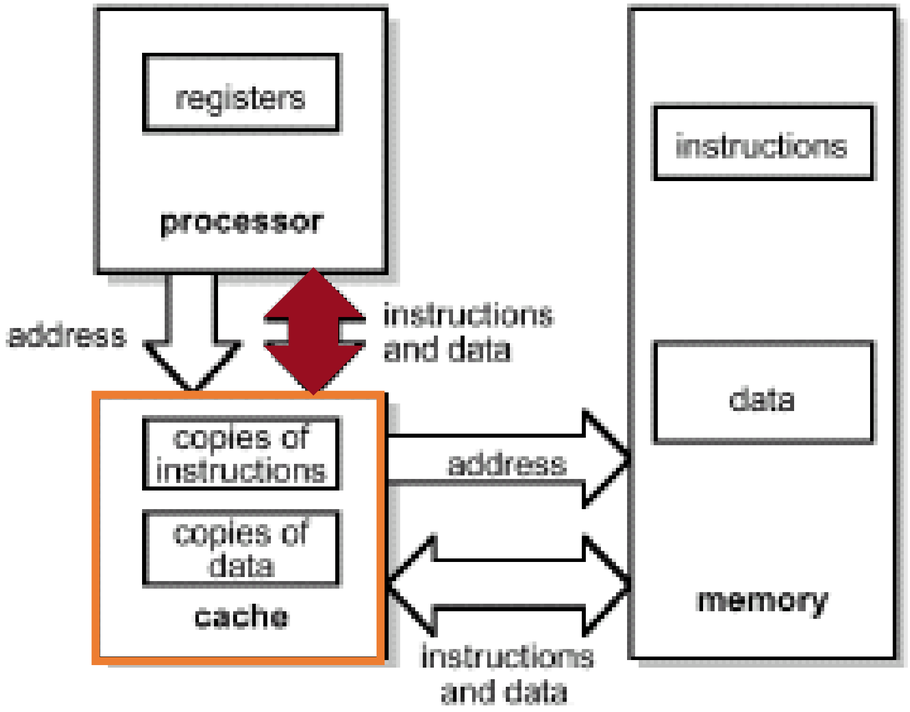
To identify the16 bytes, we need:
$\log_2 16 = 4\ bits$

To identify the 8192 lines, we need:
$\log_2 8192 = 13\ bits$



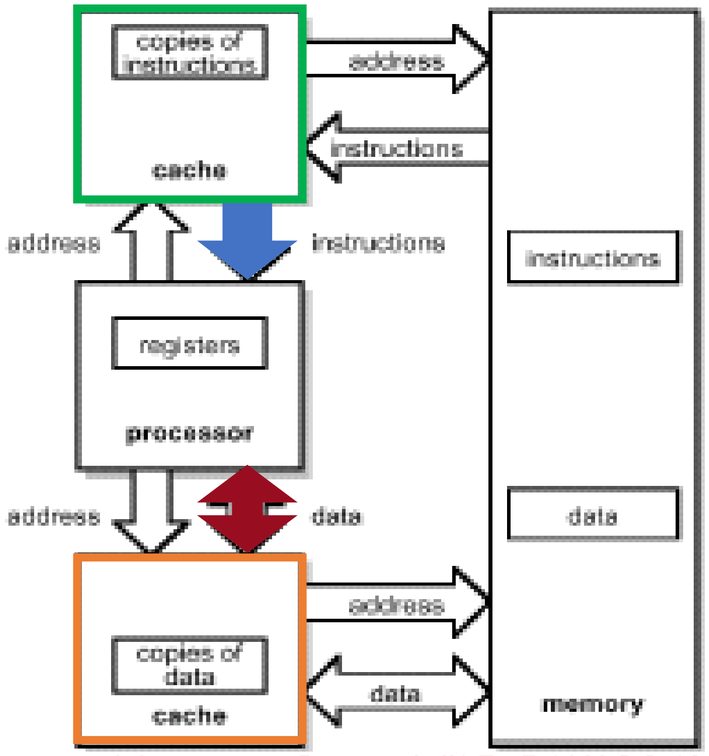
Single or separate cache for instructions and data

Single cache



- Data and instructions in the same cache
- Flexibility and higher cache hit
- 1 access per clock cycle

Separate cache

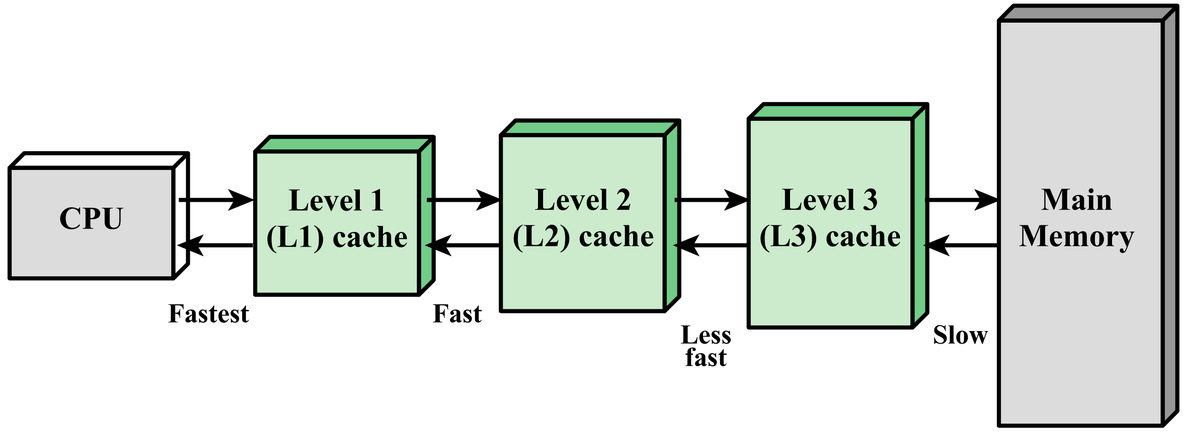


- Data and instructions in two separate caches
- Different design for each cache
- Each cache is easier
- 2 accesses per clock cycle (istructions and data)

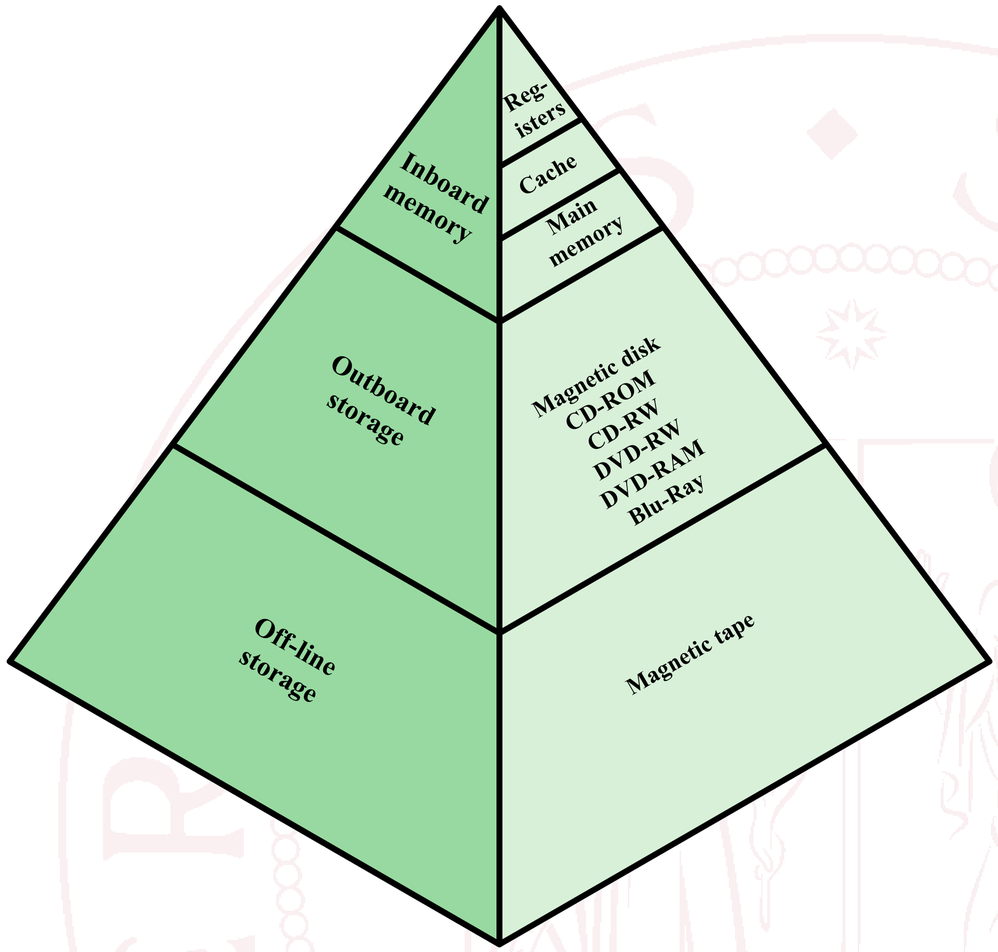
Multiple-Level Caches (1)

- It is possible to add **additional levels of cache**:
  - A 1st level cache, almost always integrated in the same processor chip, with very fast access;
  - A 2nd level cache, sometimes external to the processor chip, with quick access;
  - sometimes even a 3° level cache.
- The **closest levels** to the processor will contain the data that the processor will need in the immediate future;
- The data will be needed later are contained in the **farther levels** which are slower but more capacious.

Multiple-Level Caches (2)



(b) Three-level cache organization



Memory management techniques (1)

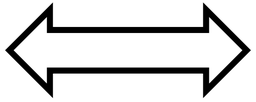
Program A

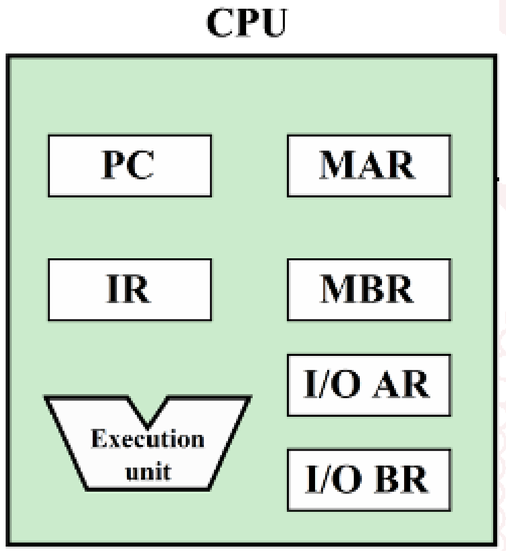
```
int main(int argc, char** argv) {
   int a = 0;
   const int NIter = 100;
   for (unsigned int i=0; i++; i<NIter) {
      a = a+1;
   }
   return 0;
}
```

in: /home/beraldo/main.c

- We coded a simple program A
- The program has been compiled and linked, the executable has been saved to the hard disk
- How does the operating system launch the program?

Solution 1:





- The instructions are read/written directly from/to the hard drive
- Extremely slow
- $V_{CPU} \ggg V_{I/O}$

Not feasible solution

Memory management techniques (2)

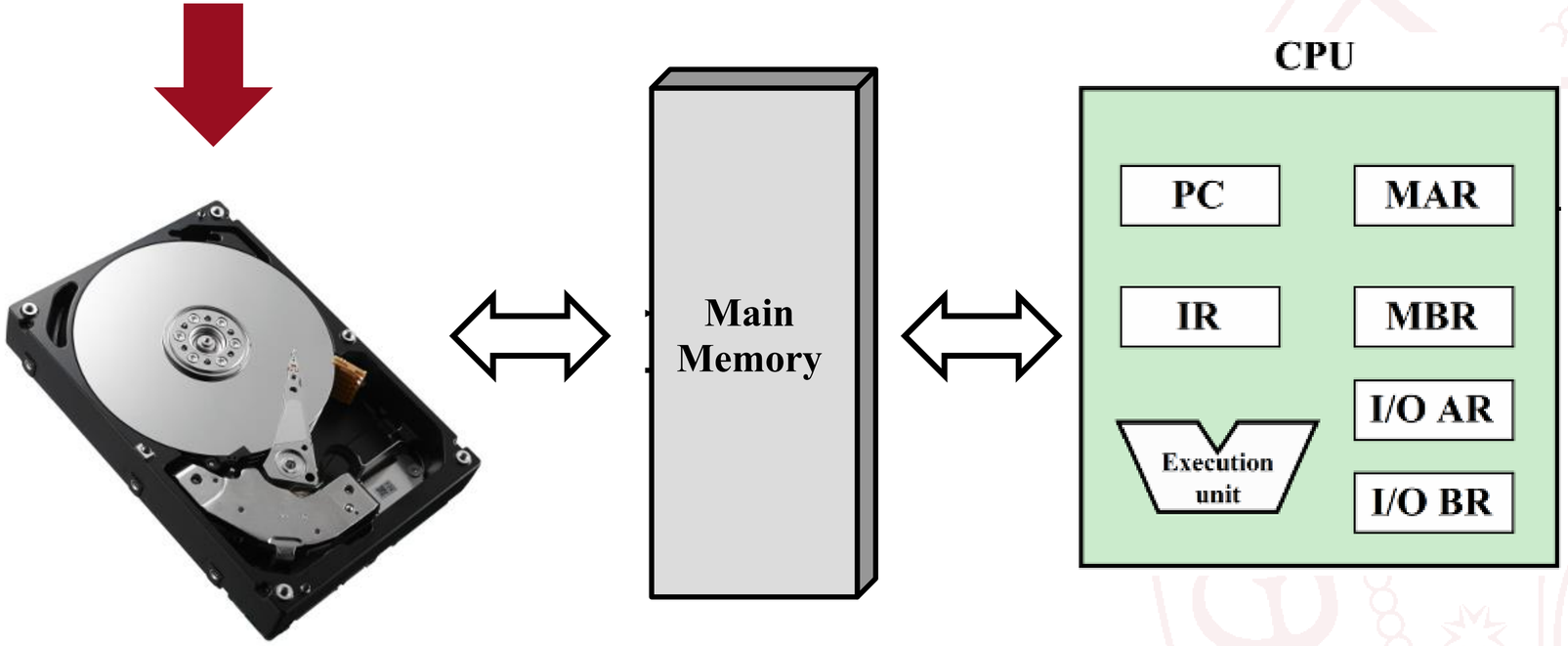
Program A

```
int main(int argc, char** argv) {
   int a = 0;
   const int NIter = 100;
   for (unsigned int i=0; i++; i<NIter) {
      a = a+1;
   }
   return 0;
}
```

in: /home/beraldo/main.c

- We coded a simple program A
- The program has been compiled and linked, the executable has been saved to the hard disk
- How does the operating system launch the program?

Solution 2:



- The program is loaded into memory (**process**)
- The CPU reads/writes in memory
- Faster access
- $V_{CPU} \gg V_{MEM}$

Memory management techniques (3)

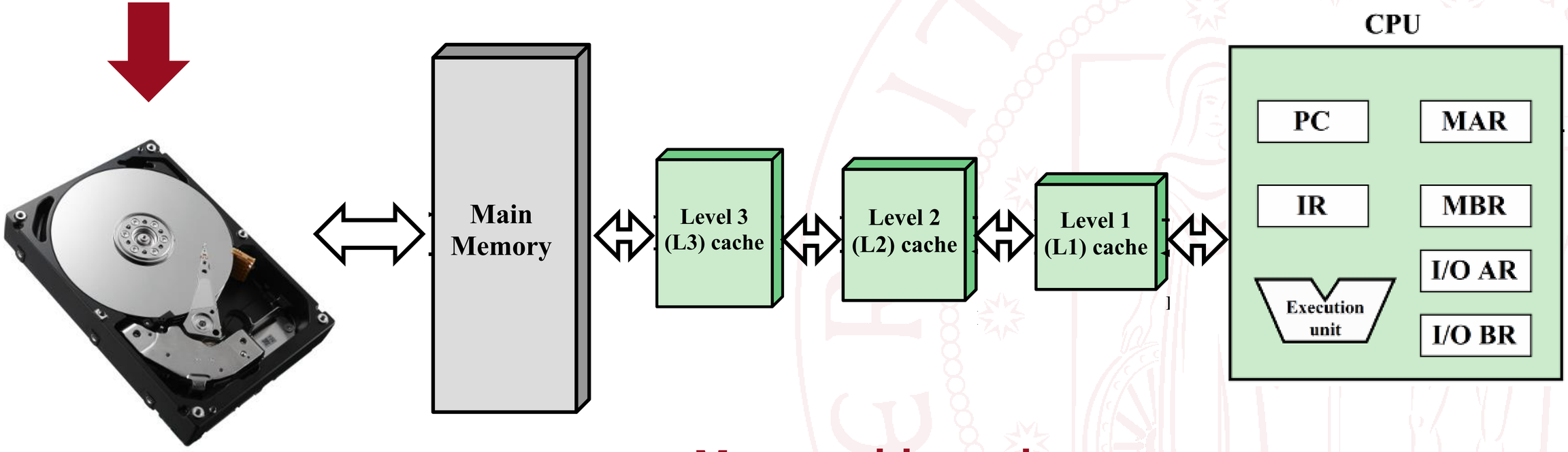
Program A

```
int main(int argc, char** argv) {
   int a = 0;
   const int NIter = 100;
   for (unsigned int i=0; i++; i<NIter) {
      a = a+1;
   }
   return 0;
}
```

in: /home/beraldo/main.c

- We coded a simple program A
- The program has been compiled and linked, the executable has been saved to the hard disk
- How does the operating system launch the program?

Solution 2 bis:



Memory hierarchy

Memory management techniques (4)

Program A

```
int main(int argc, char** argv) {
   int a = 0;
   const int NIter = 100;
   for (unsigned int i=0; i++; i<NIter) {
      a = a+1;
   }
   return 0;
}
```

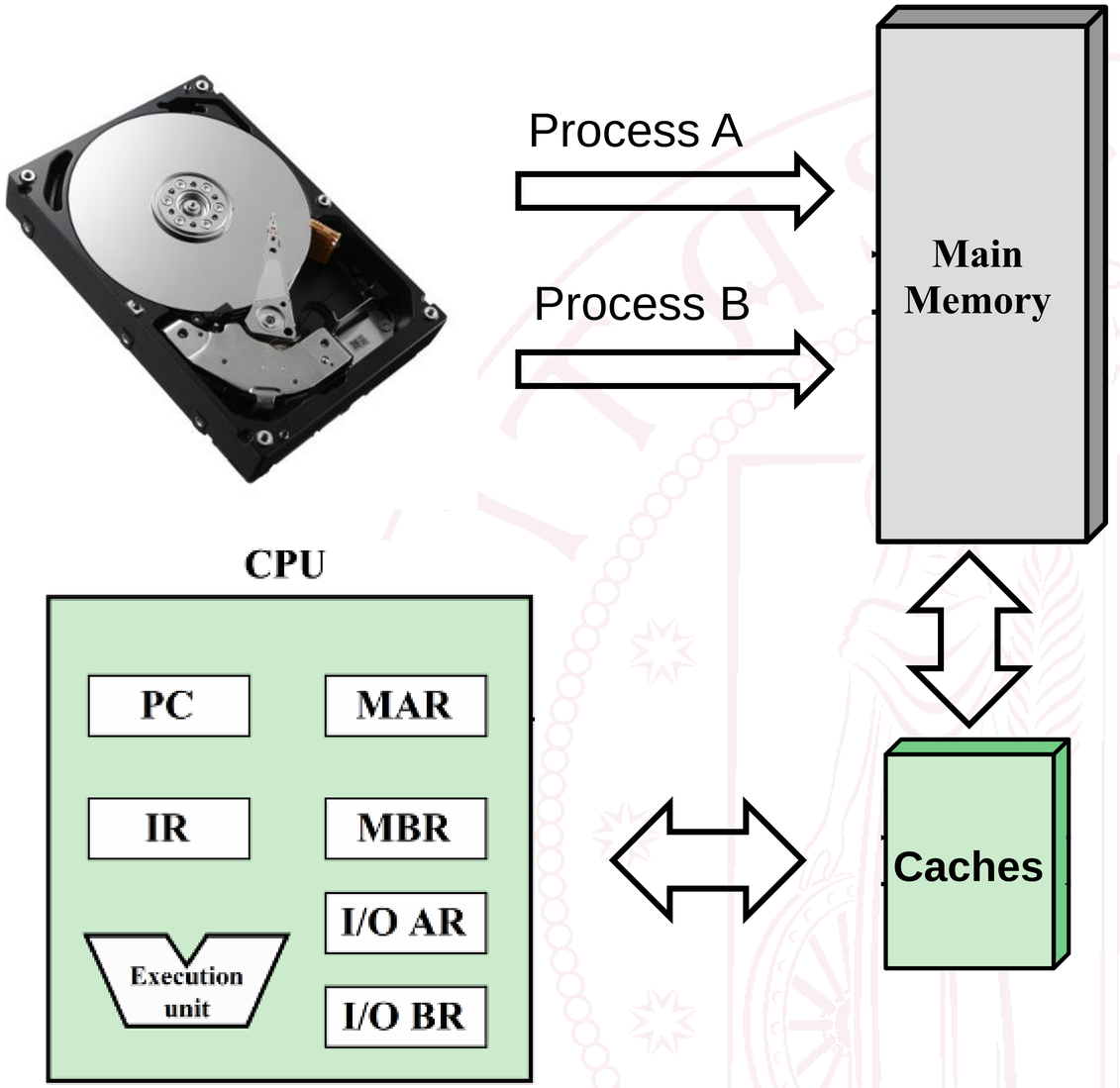
in: /home/beraldo/mainA.c

Program B

```
int main(int argc, char** argv) {
   const int NElem = 100;
   std::vector v(NElem);
   for (unsigned int i=0; i++; i<NElem) {
      v[i] = i;
   }
   return 0;
}
```

in: /home/beraldo/mainB.c

- We want to load two programs into memory in parallel



Memory management techniques (5)

Program A

```
int main(int argc, char** argv) {
   int a = 0;
   const int NIter = 100;
   for (unsigned int i=0; i++; i<NIter) {
      a = a+1;
   }
   return 0;
}
```

8 MB

in: /home/beraldo/mainA.c

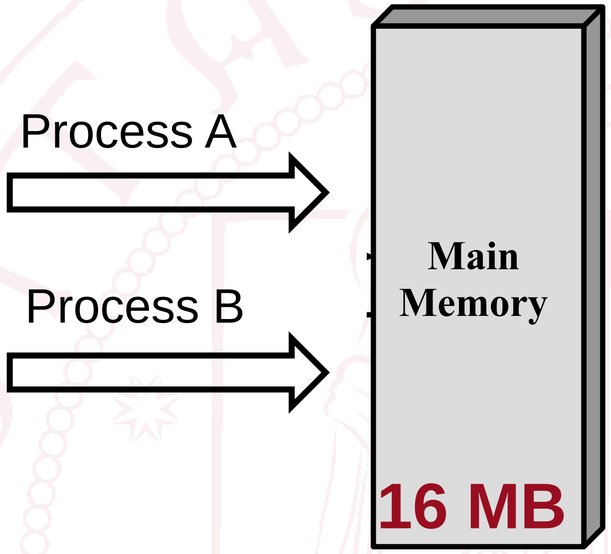
Program B

```
int main(int argc, char** argv) {
   const int NElem = 100;
   std::vector v(NElem);
   for (unsigned int i=0; i++; i<NElem) {
      v[i] = i;
   }
   return 0;
}
```

16 MB

in: /home/beraldo/mainB.c

- **Let's assume** that the first process needs 8 MB of memory space
- **Let's assume** that the second process needs 16 MB of memory space
- **Let's assume** that the memory capacity is 16MB



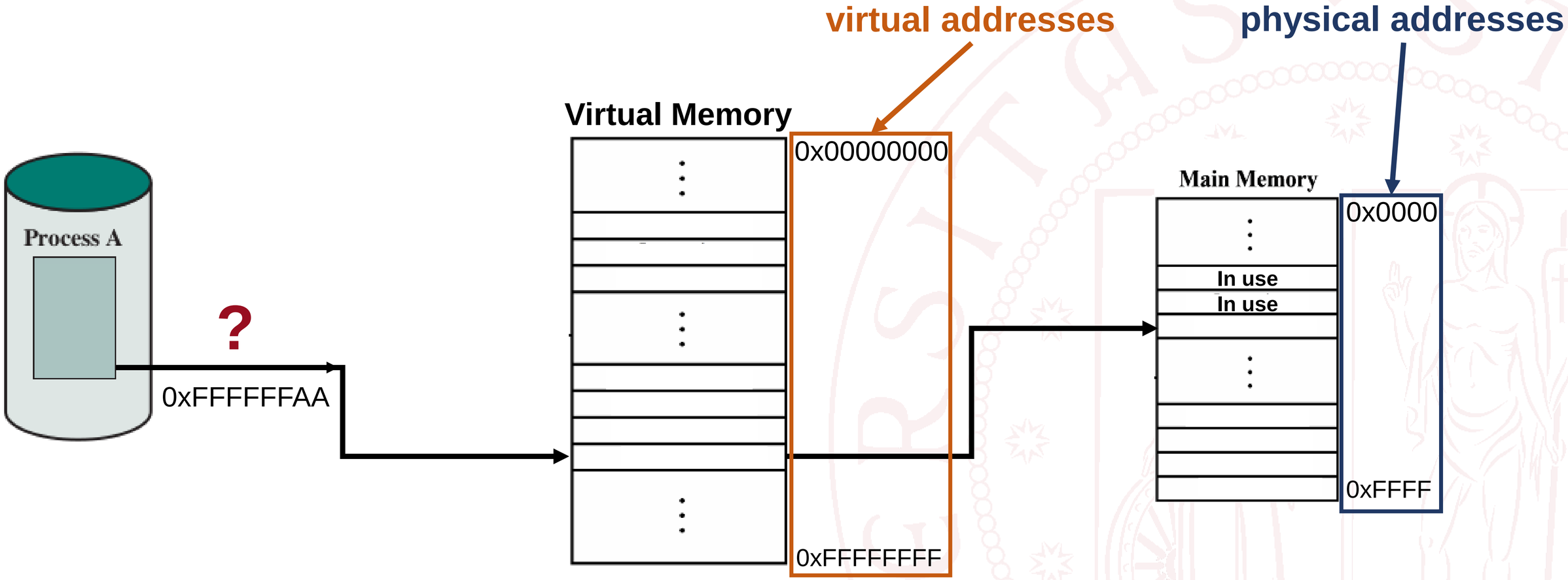
Solutions:

- Extend the memory→ it is costly
- **Optimize the memory management→ virtual memory**

- The two processes cannot be simultaneously loaded into the memory
- The operating system should wait for the the end of A to load B

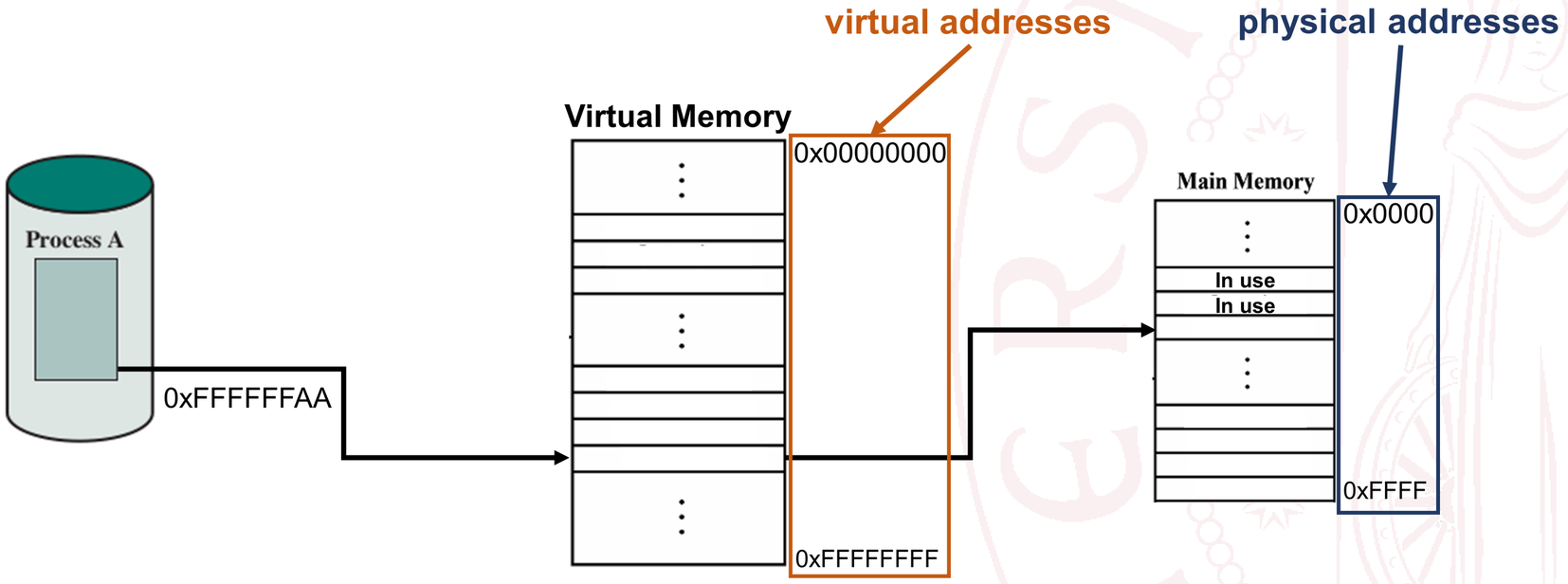
Virtual memory (1)

- Let’s imagine that the process A needs the memory address from 0xFFFFFFAA
- However, we have limited memory with locations ranging from 0x0000 to 0xFFFF
- Our memory is almost entirely «free»
- We can then **remap** the address required by the process into free memory cells



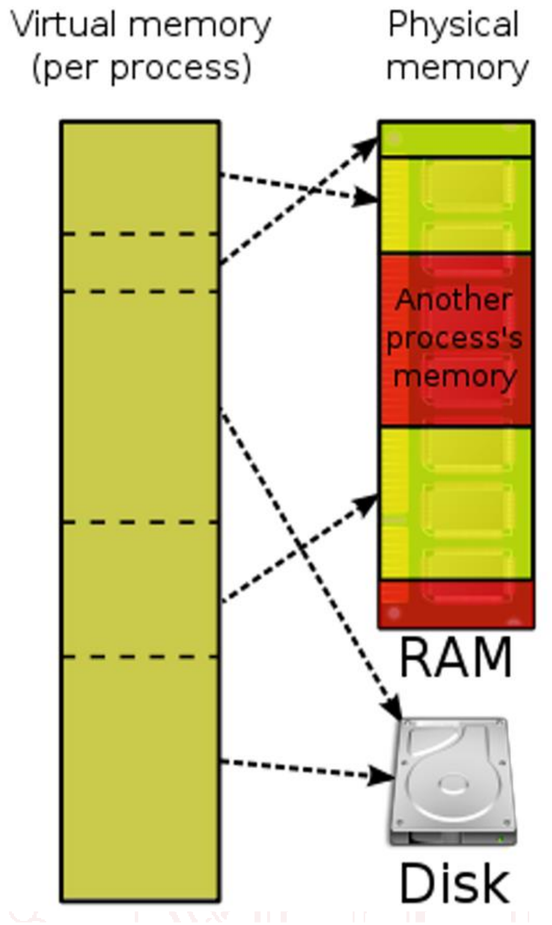
Virtual memory (2)

- The programmer «access» always and only to the **virtual addresses**
- The **physical accesses** are automatically computed by the hardware/software
- The **virtual memory** is addressed by the virtual addresses
- Virtual memory can be much larger than the physical memory, thanks to the optimization of its management (e.g., via paging)
- The virtual memory can:
    1. Use processes that require more memory than the physical available
    2. Use memory areas shared by multiple programs



Virtual memory (3)

- Extends the memory capacity of the main memory by using a portion of the disk drive.
- Allows the system to run programs that are bigger than the size of main memory
- Paging is implemented in this system.

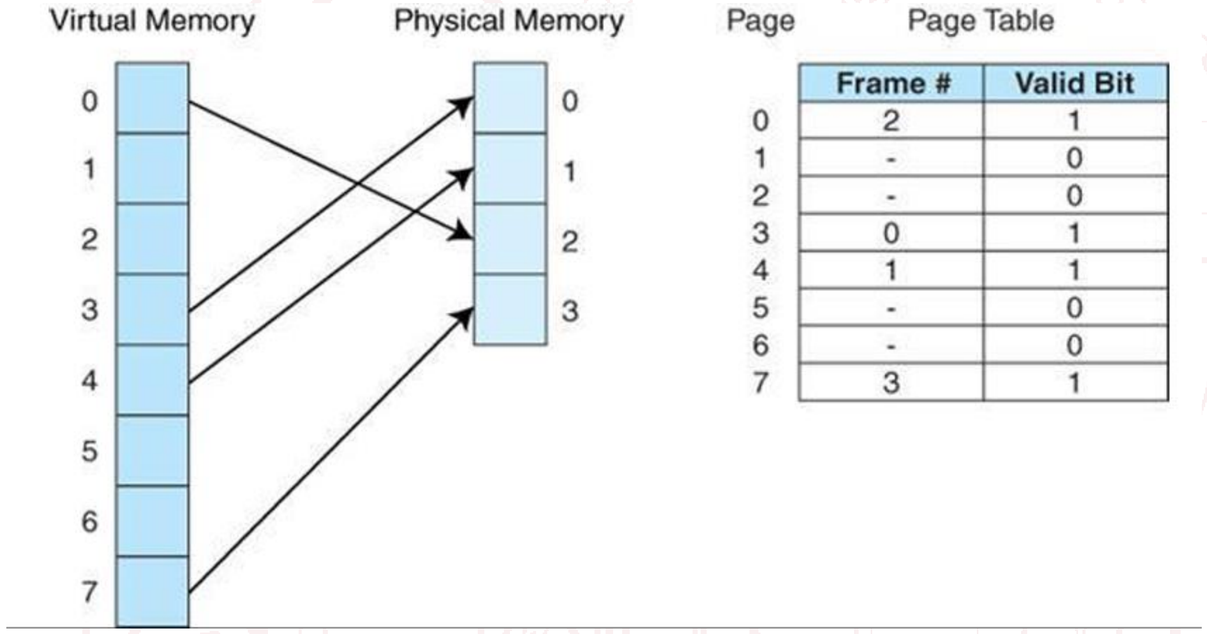


Virtual memory (3)

- Extends the memory capacity of the main memory by using a portion of the disk drive.
- Allows the system to run programs that are bigger than the size of main memory
- Paging is implemented in this system.

We need data structure that maintain information concerning the location of each page, whether on disk or in memory

PAGE TABLE

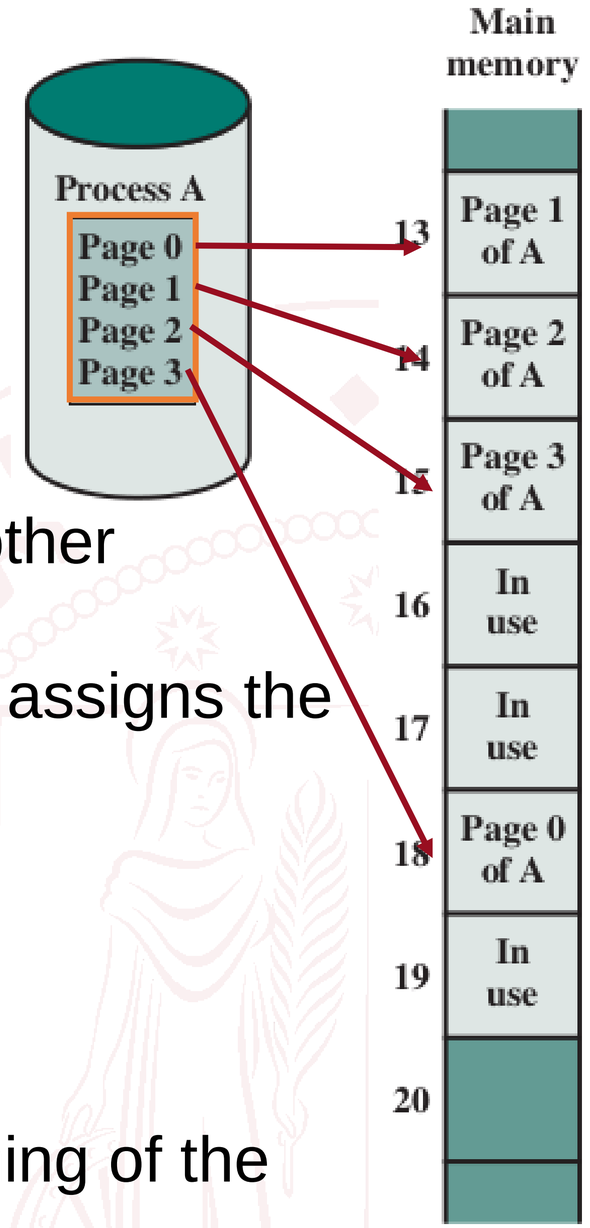


Page Table

Page	Frame #	Valid Bit
0	2	1
1	-	0
2	-	0
3	0	1
4	1	1
5	-	0
6	-	0
7	3	1

Paging (1)

- A process A in the hard drive
- The process A is divided into small blocks called **pages**
- The memory is also divided into blocks of the same size called **page frames**
- Some page frames in the memory may already be in use by other processes
- When the operating system loads the process into memory, it assigns the different pages to the free page frames



- The **physical address** is a real memory address
  - For the process A → 13, 14, 15, 18
- The **virtual address** is the address with respect to the beginning of the program
  - For the process A→ 1, 2, 3, 4

Paging (2)

- The physical address space in memory is divided into small **page frames** (e.g. 4 KB)
- The virtual address space is also divided into blocks of the same size, called **pages**

- Let's assume that the virtual address is 32 bits
- To address all the words in a 4KB page, we need $\log_2(4 \cdot 1024) = \boldsymbol{12\ bits}$
- The remaining 32-12=**20 bits** are used to address $2^{20}\ pages$ in the virtual memory
- Let's assume to have a 16 MB physical memory, I have $2^{12}\ page\ frames$ (24-12=12)
- The **page offset** refers to the word inside the page
- **Virtual page number**
- **Physical page frame number**

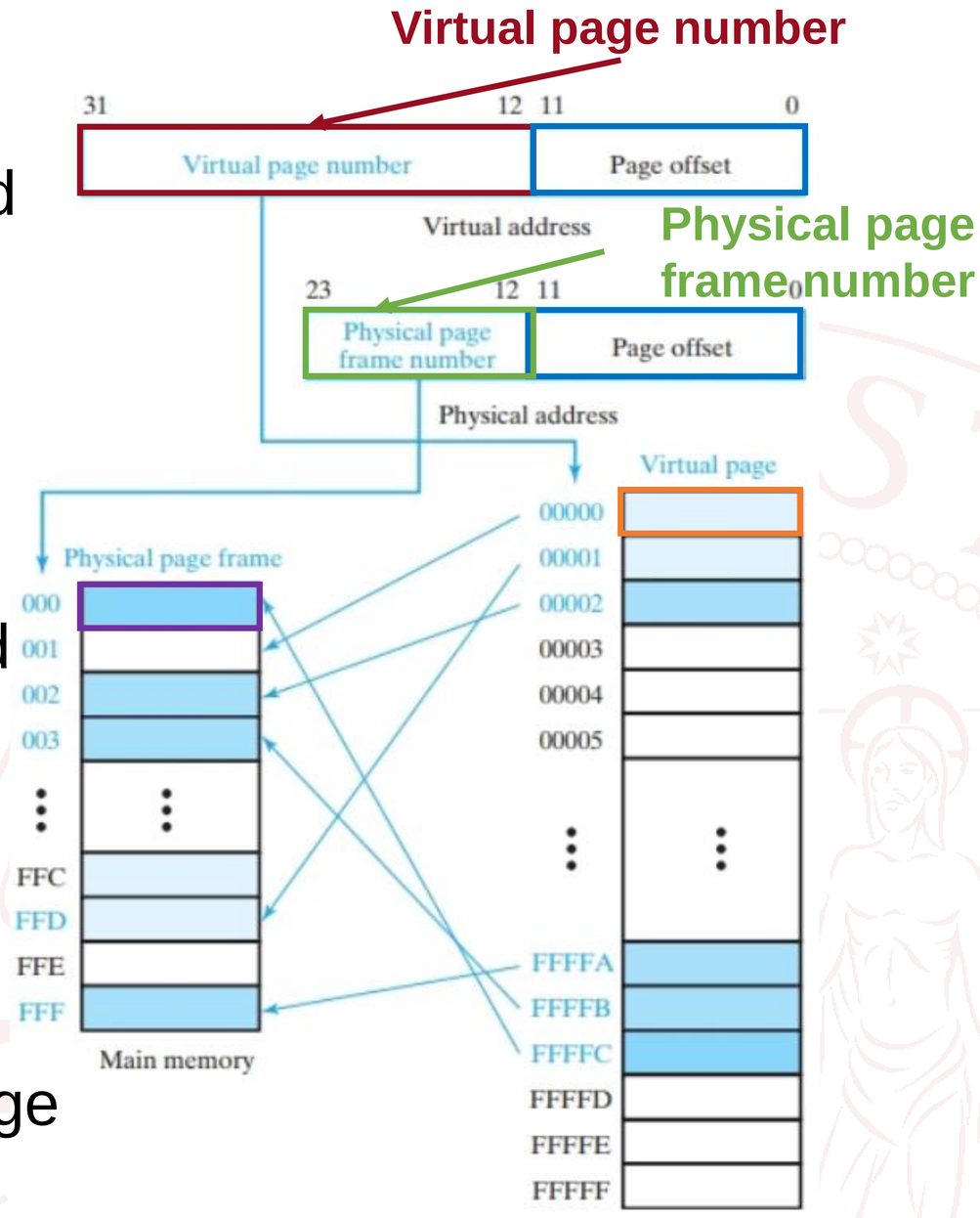
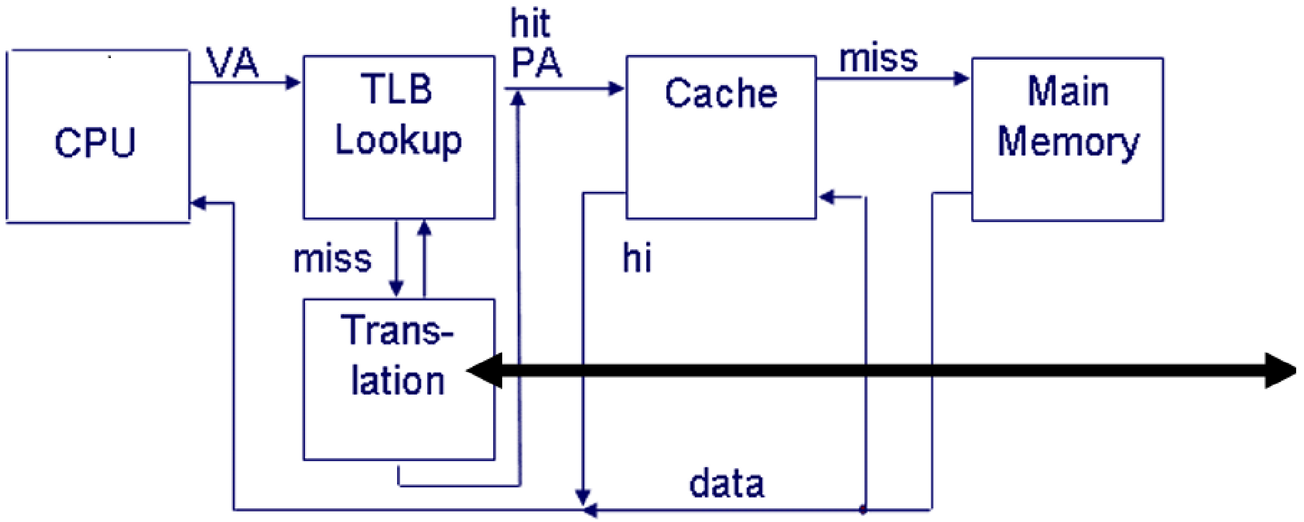


FIGURE 12-11
Virtual and Physical Address Fields and Mapping

Access to the page tables

To reduce memory accesses, let’s assume we have a cache that can directly translate virtual addresses into physical addresses

Such a cache is called **Translation Lookaside Buffer (TLB)**



TLB (Translation Lookaside Buffer)
A cache memory that stores recent translations of virtual memory to physical addresses for faster retrieval

Translation Lookaside Buffer

- It is a fully associative or k-way set-associative cache
- It compares the required **virtual page number** with those in cache

- It uses the **virtual page number** required from CPU as input
- It is compared with all the virtual page numbers in the cache
- If there is a match, and the valid bit is 1 (TLB Hit), the output is the corresponding **page frame number**
- If there is no match (TLB Miss), the page is searched in memory and loaded into the TLB
- If the page is not found, there is a **page fault** →the page is loaded from the hard disk to memory
- The effectiveness depends on the spatial and temporal locality of the pages in the TLB

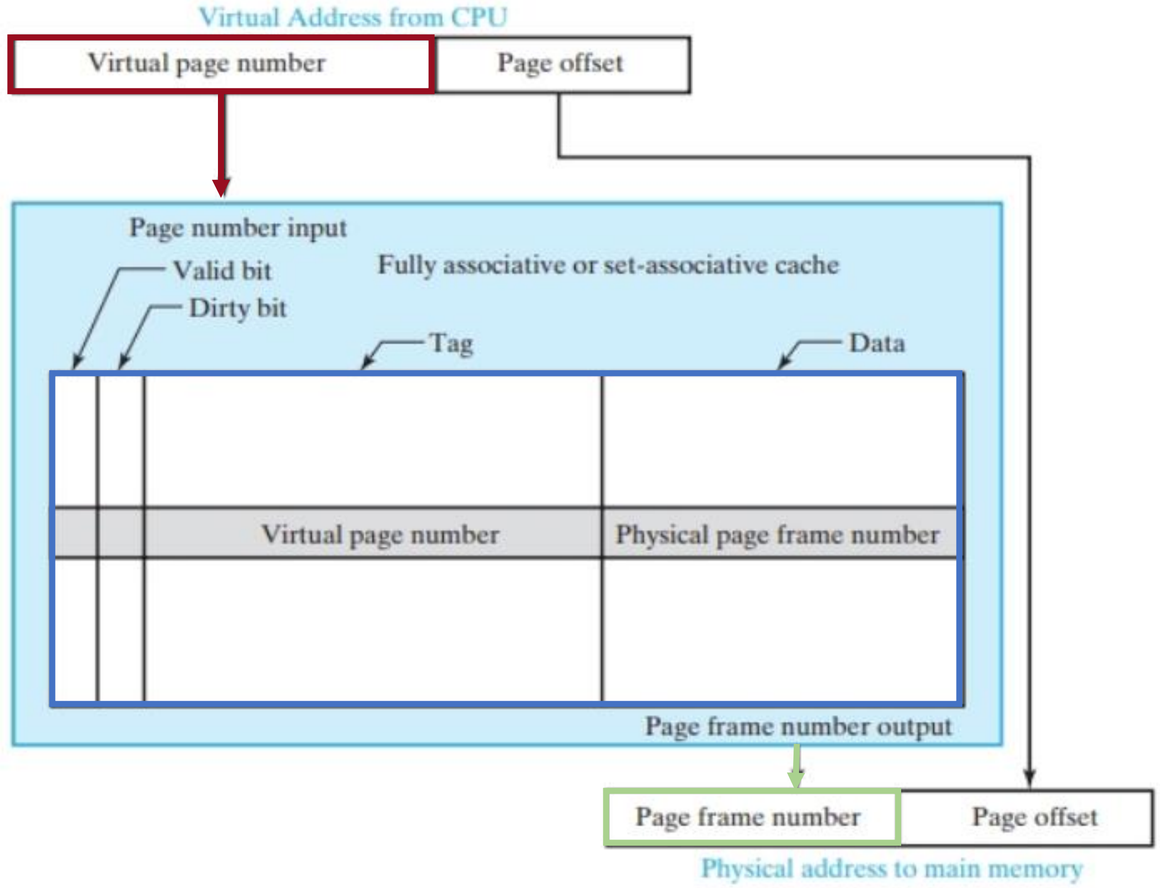


FIGURE 12-14
Example of Translation Lookaside Buffer

How it works

The requested address is broken down into **page** and **offset**.

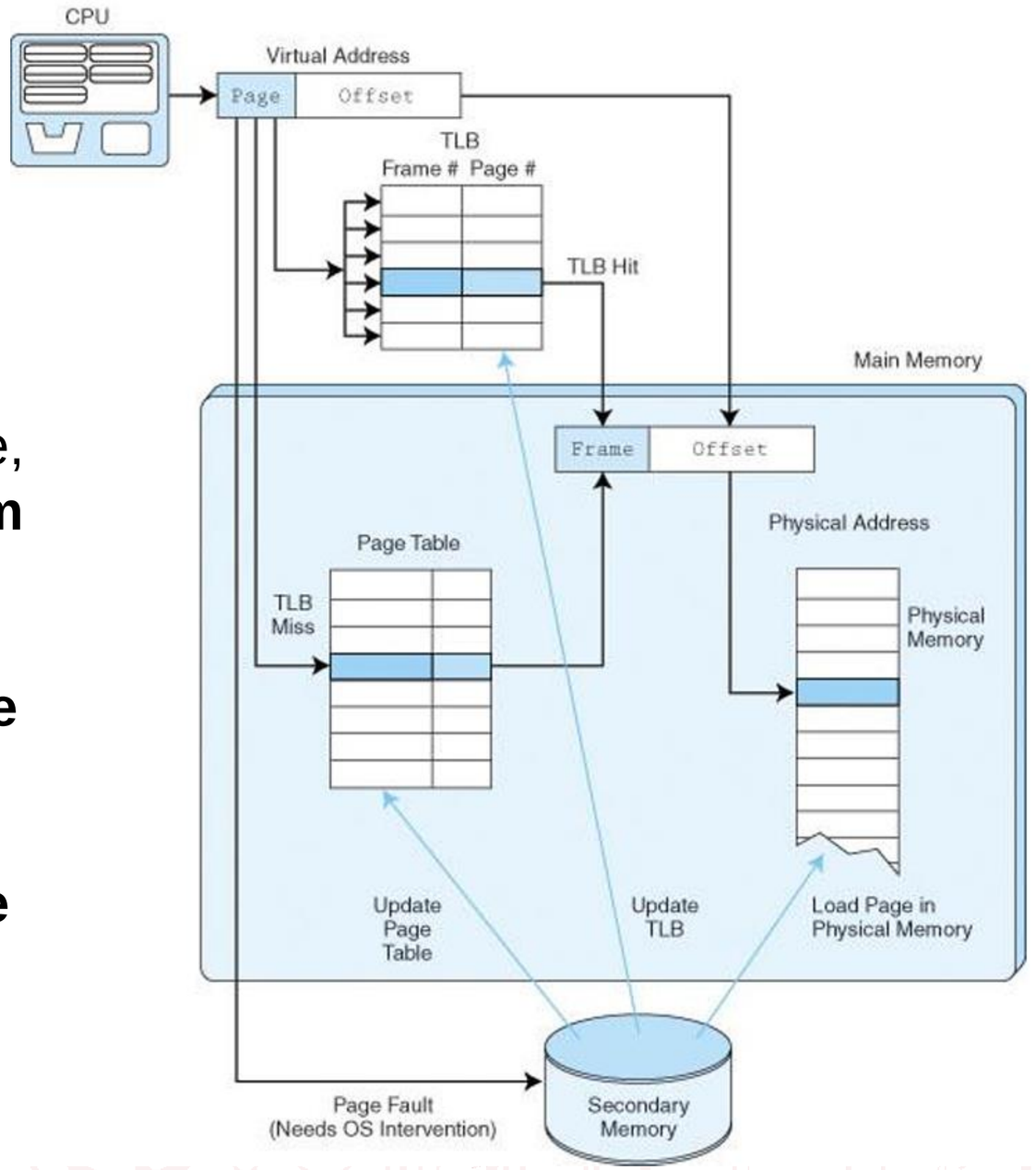
• Requested page is **searched in TLB**.

• If **there is** a frame number that matched the page, **TLB HIT** is obtained, and **data will be loaded from TLB**.

• Else, **page will be searched through page table** with direct mapping principle.

• If matched **page id is found in page table**, **Page Table Hit** is obtained and data will be loaded from Page Table. **TLB will be updated** accordingly.

• Else, Misses obtained, and **data will be loaded from secondary memory**.



Disclaimer

Figures from *Logic and Computer Design Fundamentals*,
Fifth Edition, GE Mano | Kime | Martin

Questions