

Operating System

Chapter 8

Contents

1. Introduction

1.1 Types of OS

2.1 Scheduling

2.2 Different States of a process

3.1 Memory Management and its necessities

3.2 Swapping

3.3 Partitioning

3.4 Paging and Page Table

4.1 Demand Paging

4.2 Improving Paging

1.Introduction

Why do we need to study Computer Architecture?

- Understanding Computer Functionality
- Hardware Design and Optimization
- Software Development
- Compatibility and Portability
- Troubleshooting and Debugging
- System Administration
- Security
- Innovations and Advances
- Career Opportunities
- Academic Pursuits

Why do we need an OS?

- **Convenience :** It becomes easier for a programmer to code
- **Resource Management :**
 - Allocating memory for the user programs
 - Scheduling time for the user programmers

Computer Hardware and Software Structure

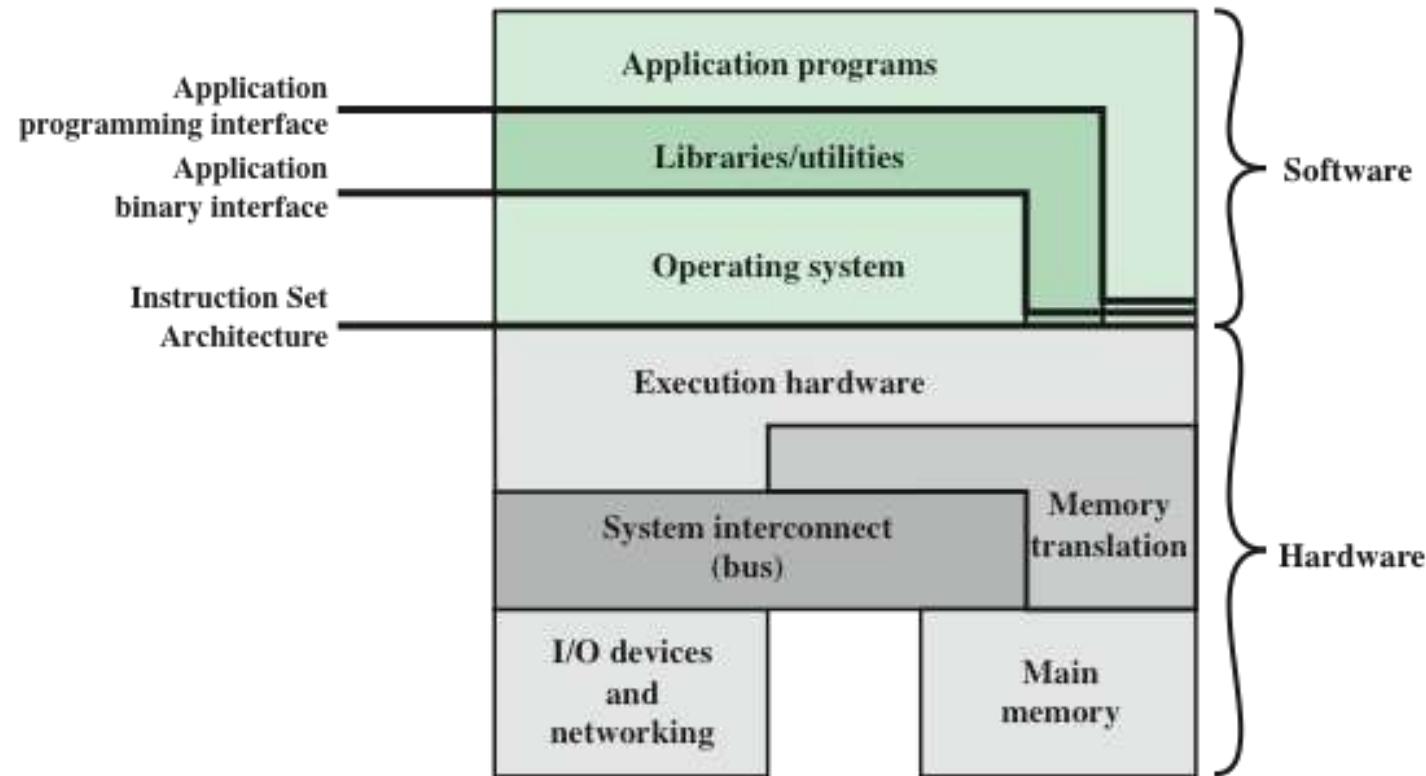


Figure 8.1 Computer Hardware and Software Structure

The services provided by the Operating System

- **Helping in writing codes** - OS has different utility functions. These functions can be called by our programs . For example i/o functions.
- **Helping in program execution** – Allocating RAM, I/O devices.
- **Controlled access to files** – Helps in determining which program has permission to access a file.
- **Error Detection and Response** – Detecting an error and taking appropriate measures.
- **Interrupt** : Contains functions for handling different interrupts.
- **Accounting** – Keeping log of CPU usage, error logs.

The OS as a resource manager:

A program for running other programs:

- OS functions the same way as **any other computer program**.
- It gives **control to the processor** for running other programs, the processor **returns back the control** to it.
- **Timer:**
 - Whenever the control is released to another programmer (PC points to the start of that program)
 - A timer is started
 - Once the timer becomes zero
 - The control is returned back to the OS (i.e the PC points to the OS code)

The OS as a resource manager:

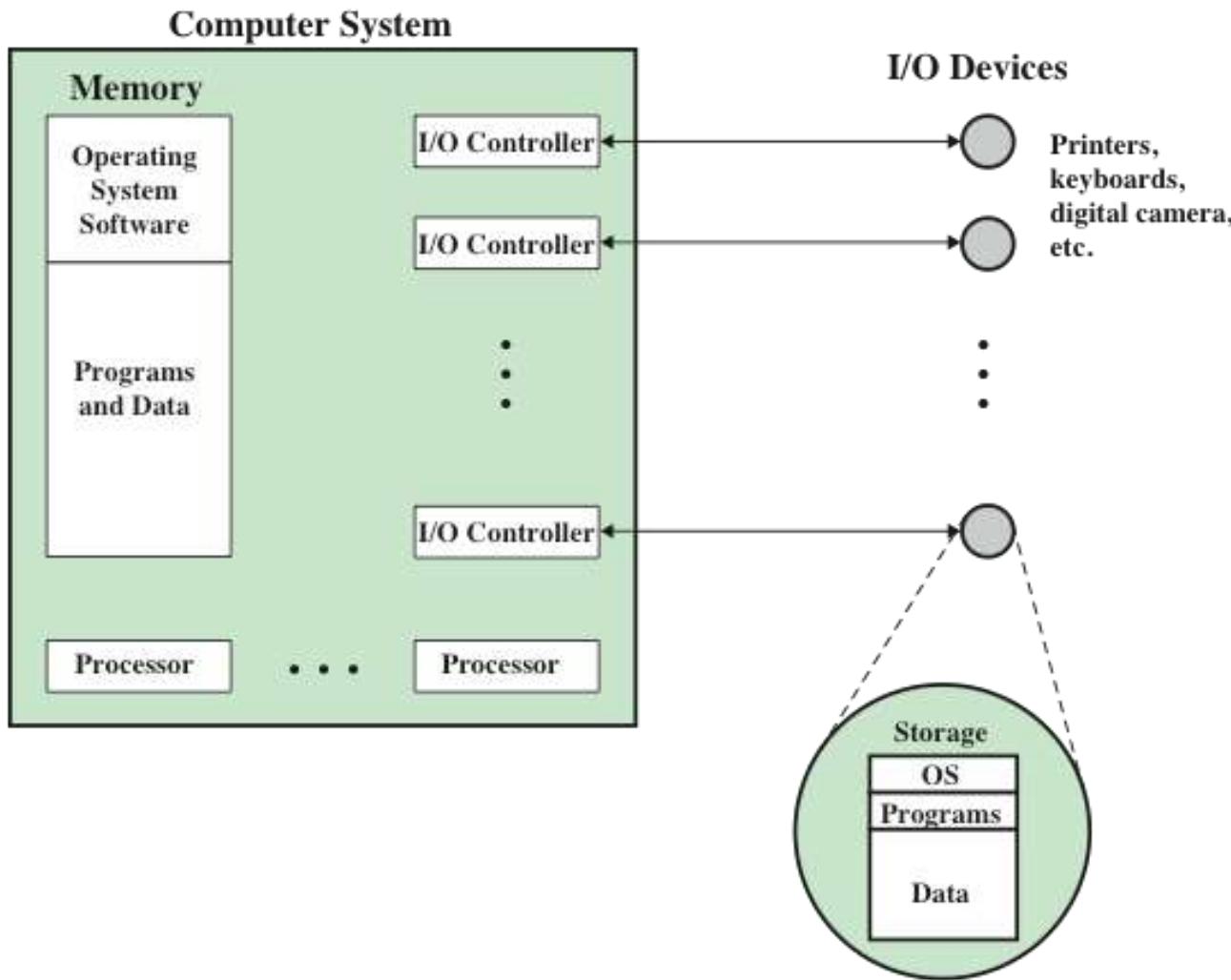


Figure 8.2 The Operating System as Resource Manager

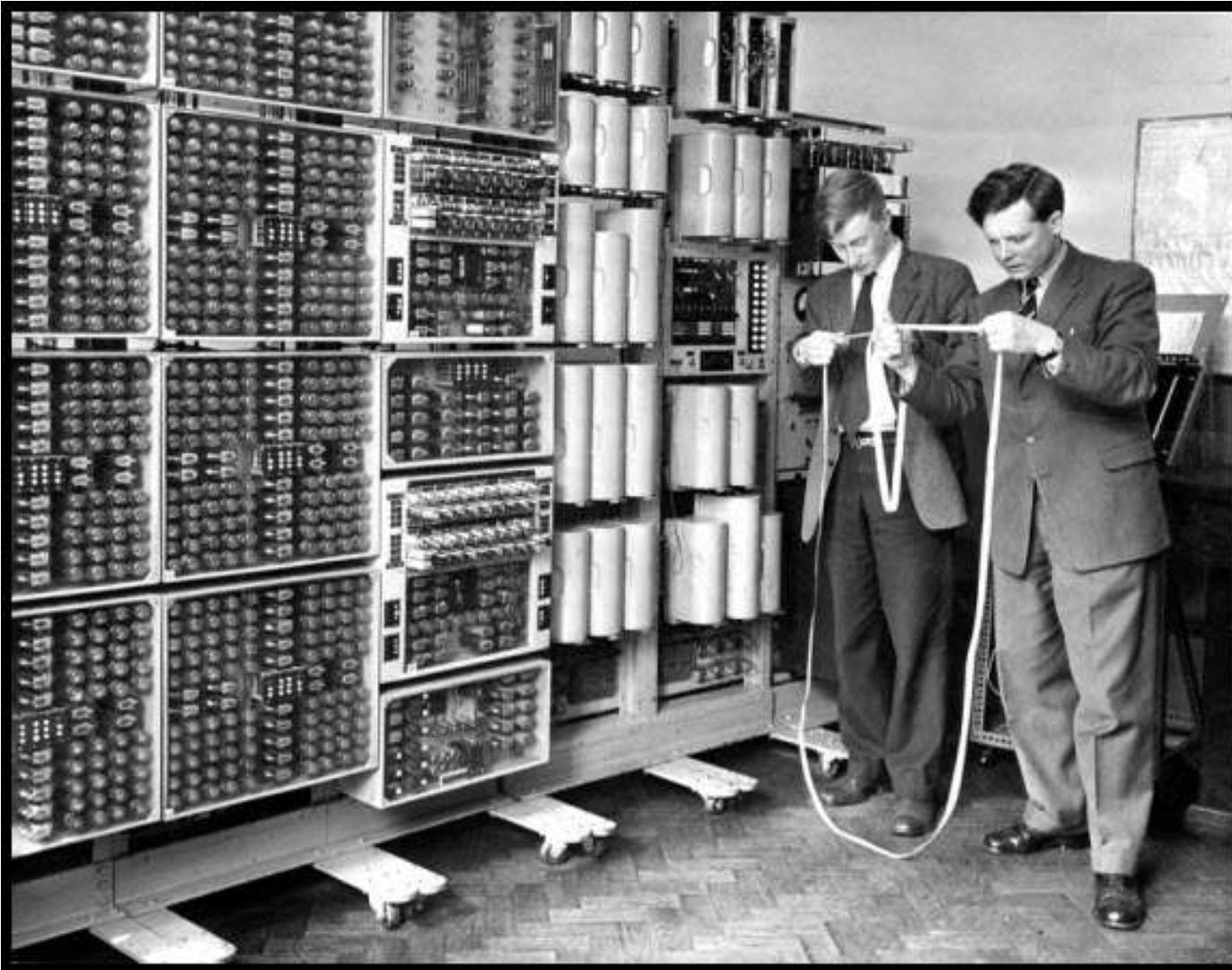
Types of Operating Systems

- Interactive system
 - The user/programmer interacts directly with the computer to request the execution of a job or to perform a transaction
 - User may, depending on the nature of the application, communicate with the computer during the execution of the job
- Batch system
 - Opposite of interactive
 - The user's program is batched together with programs from other users and submitted by a computer operator
 - After the program is completed results are printed out for the user

Types of Operating Systems

- Multiprogramming
 - Keep processor as busy as possible
 - Work on more than one program at a time
 - Processor switch rapidly among the programs
- Uniprogramming
 - Works only one program at a time

Early Systems



Early Systems

- From the late 1940s to the mid-1950s the programmer interacted directly with the computer hardware – there was no OS
 - Processors were run from a console consisting of display lights, toggle switches, some form of input device and a printer
- Problems:
 - Scheduling
 - Sign-up sheets were used to reserve processor time
 - This could result in wasted computer idle time if the user finished early
 - If problems occurred the user could be forced to stop before resolving the problem
 - Setup time
 - A single program could involve
 - Loading the compiler plus the source program into memory
 - Saving the compiled program
 - Loading and linking together the object program and common functions

2.1.Scheduling

Process

Process : basically refers to a running program and the components that are helping it to run

- A program in execution
- Registers being used
- Program counters
- Other meta info (process id, running time, children ids, etc)

Meta data maintained for a process by the OS

- **Identifier** – every process has a unique id
- **State** – we will see this soon – each process has a state
- **Priority** – some of the processes have higher priorities (e.g – device driver programs)
- **Program Counter** – the program counter value of that process
- **Starting and ending location** of the process code
- **Register data** – the register values
- **I/O status** - Was the process waiting for some input
- **Log** – Other information (running time, no. of executions etc.)

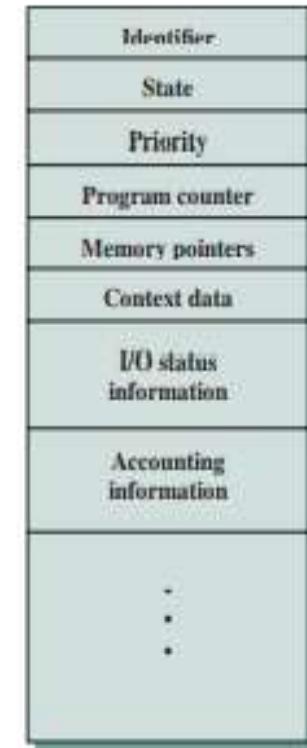
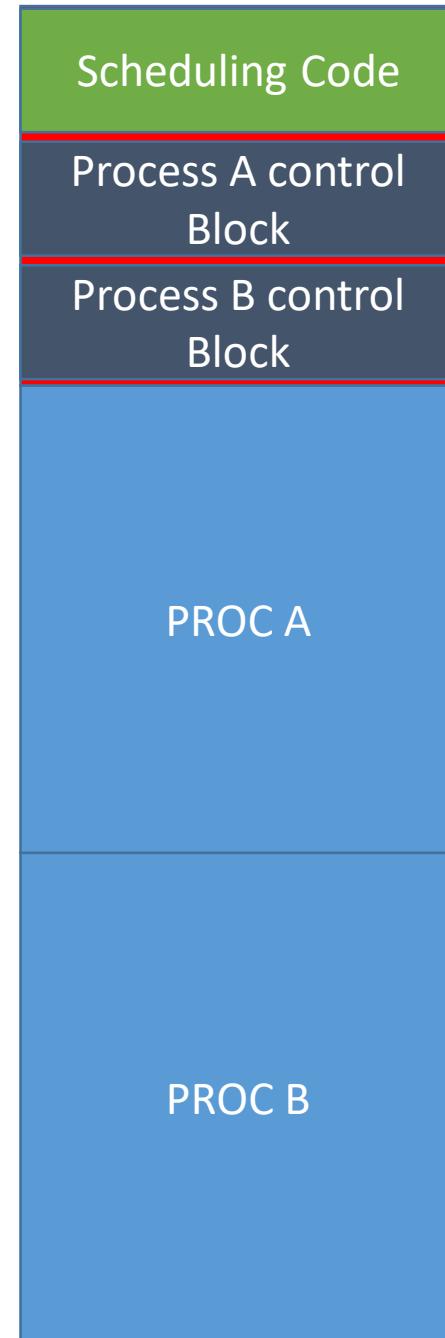
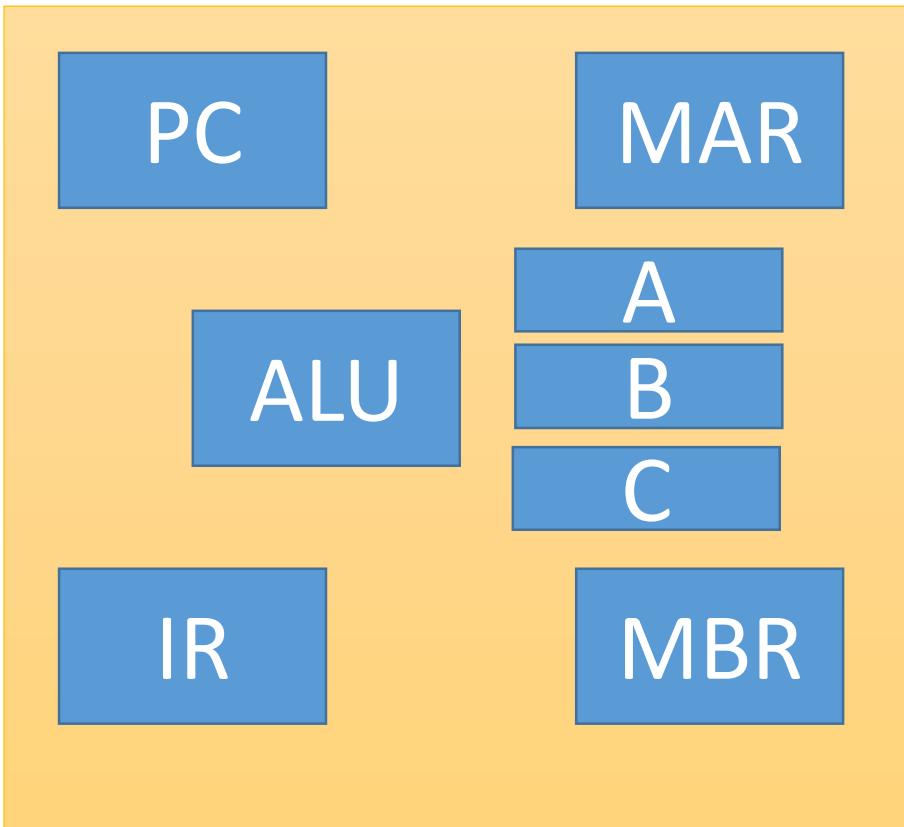


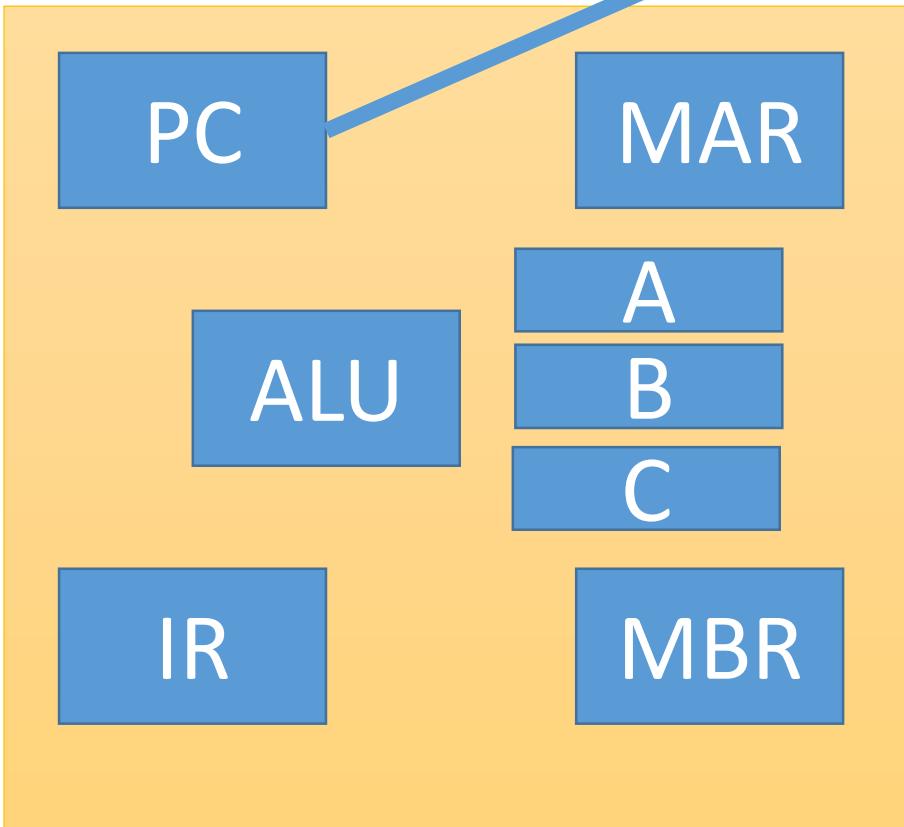
Figure 8.8 Process Control Block

(The memory of the OS where all the info of a process is stored is called the **process control block**.)

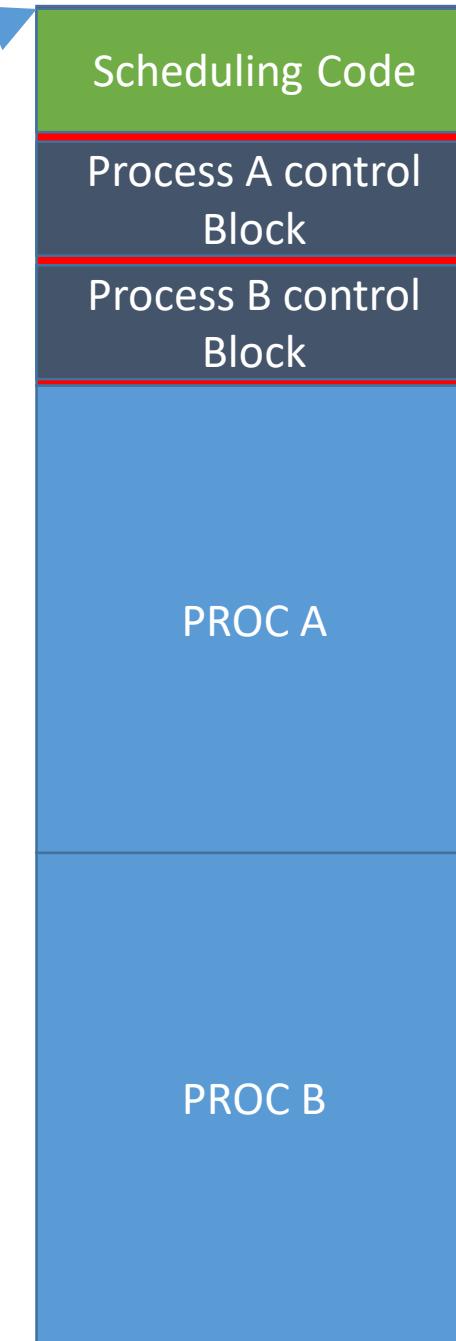
Putting together the Timer and PCB



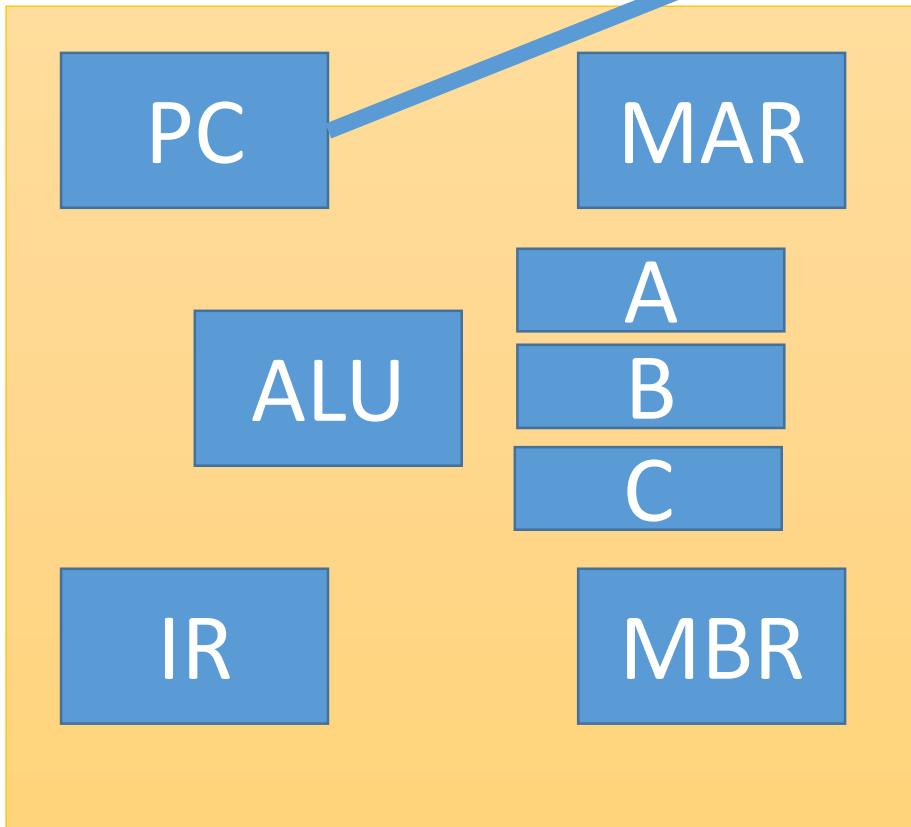
Putting together the Timer and PCB



TIMER



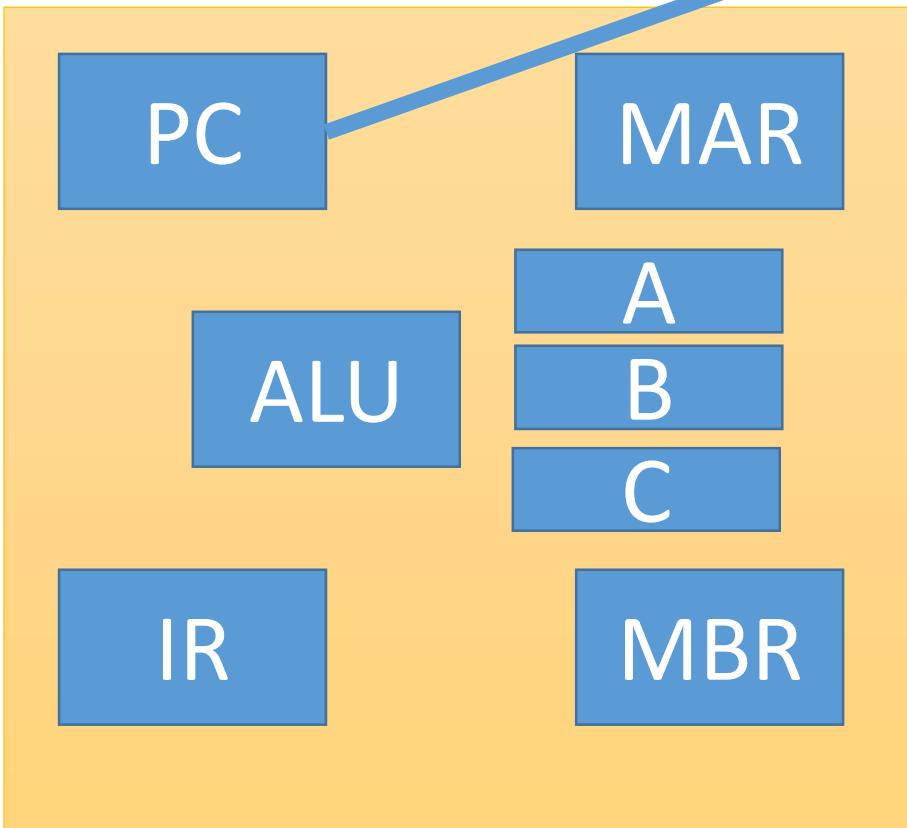
Putting together the Timer and PCB



TIMER



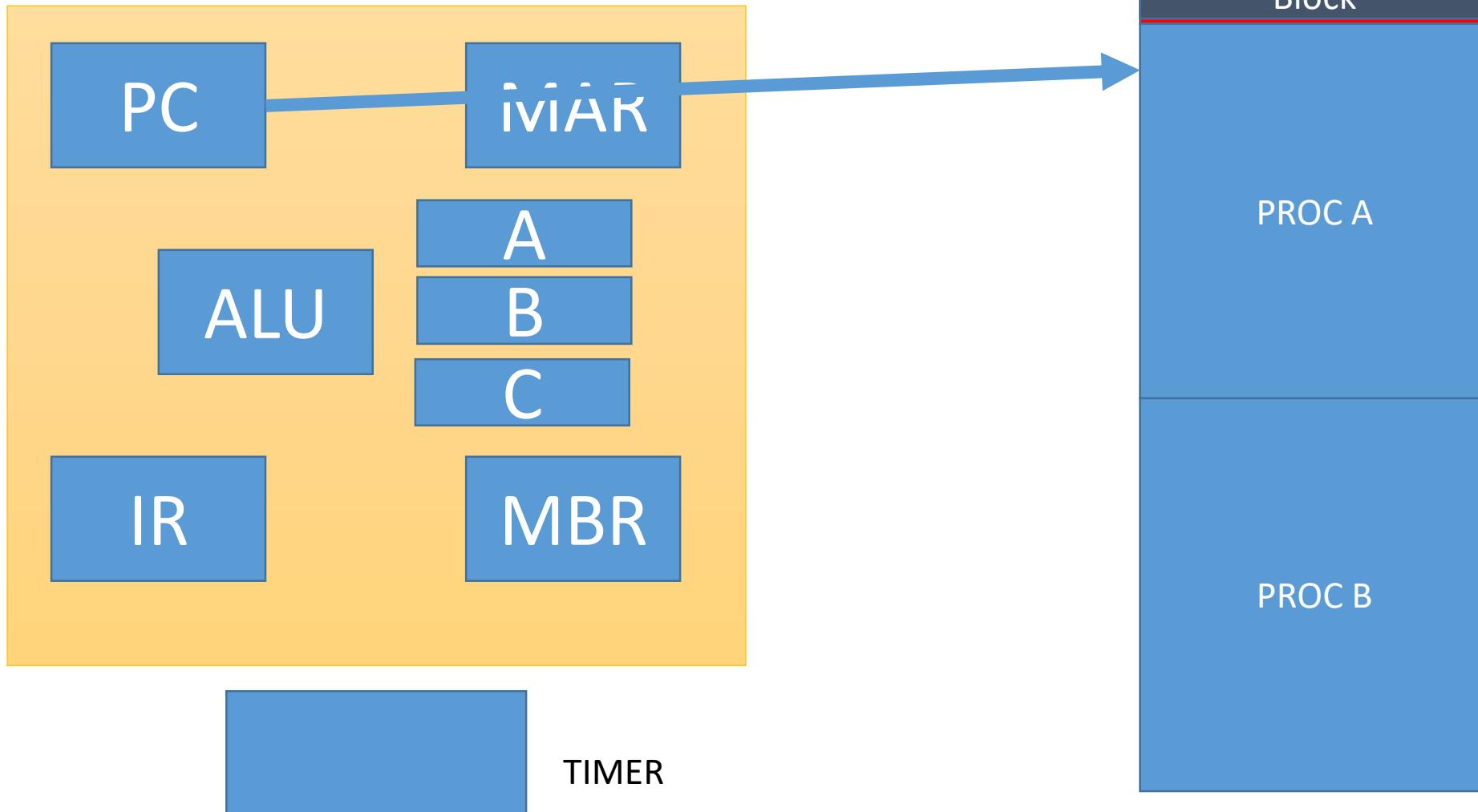
Putting together the Timer and PCB



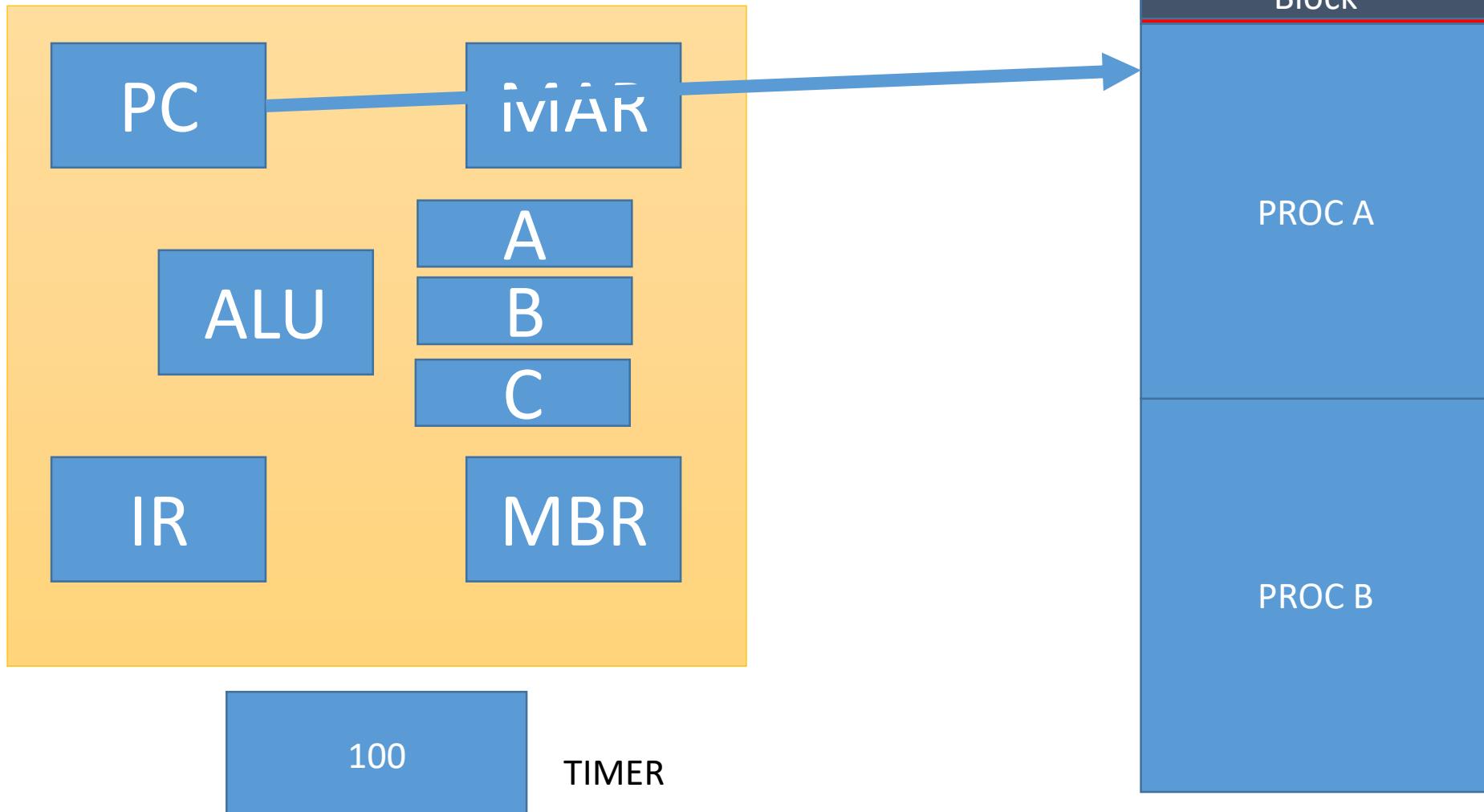
TIMER



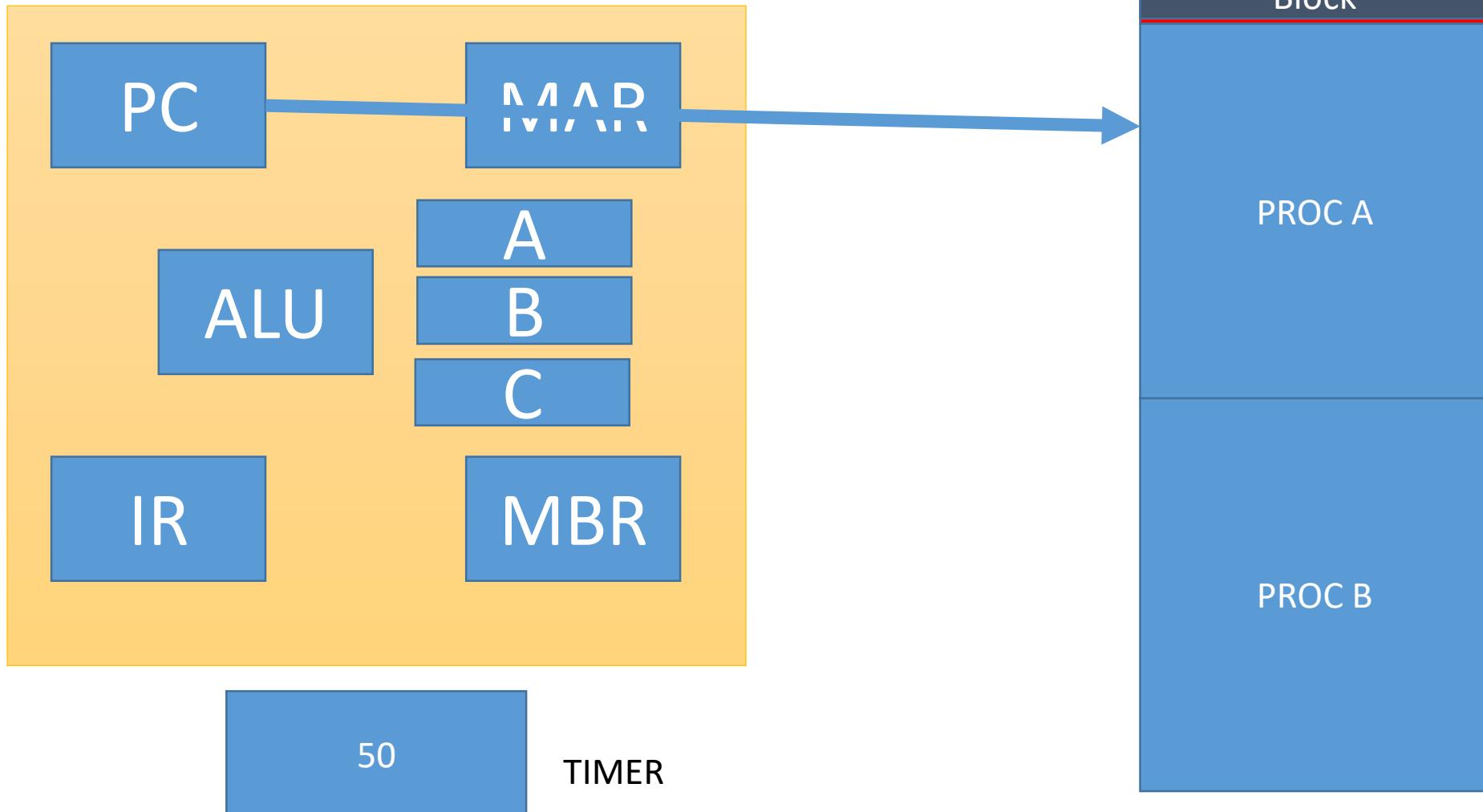
Putting together the Timer and PCB



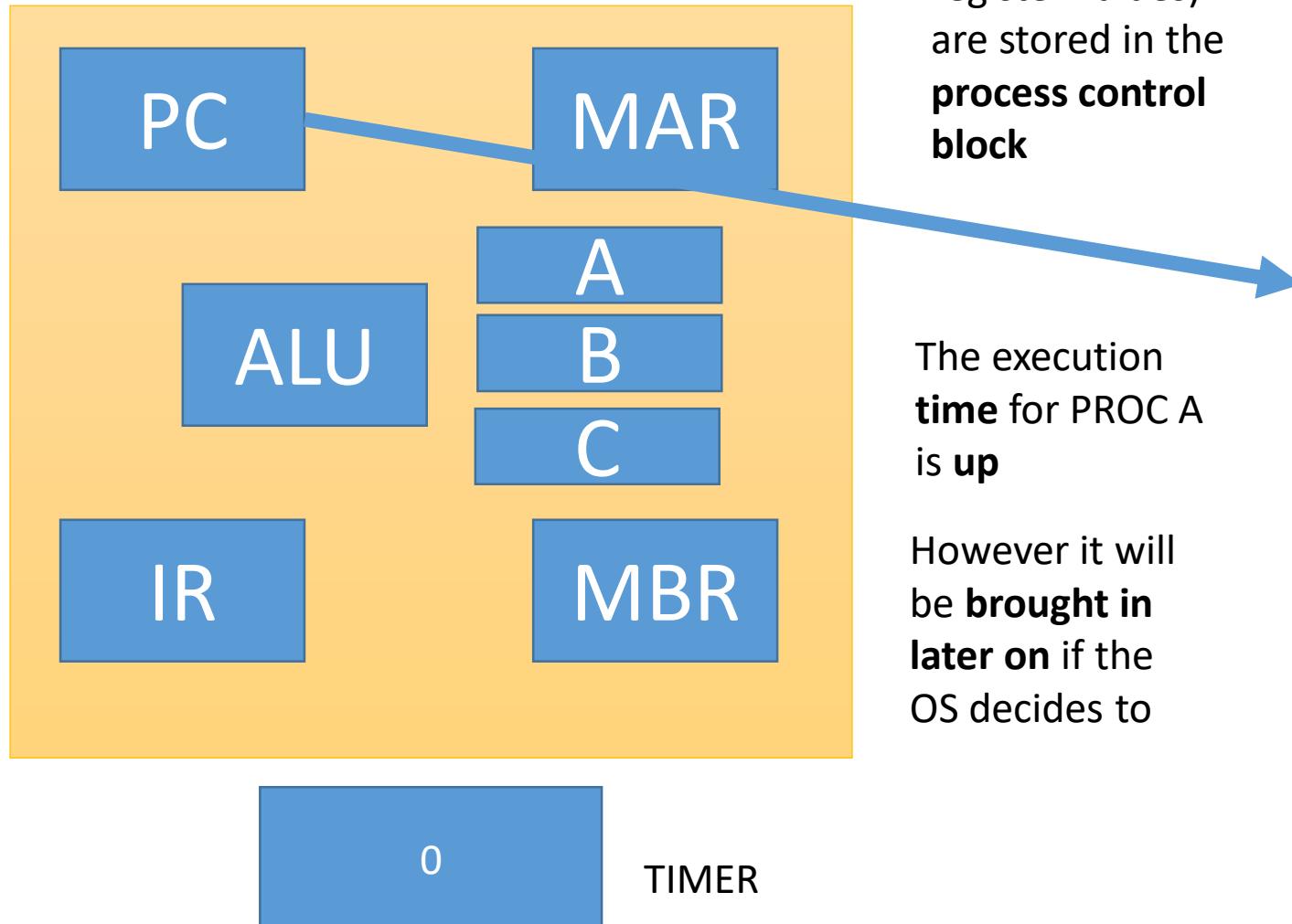
Putting together the Timer and PCB



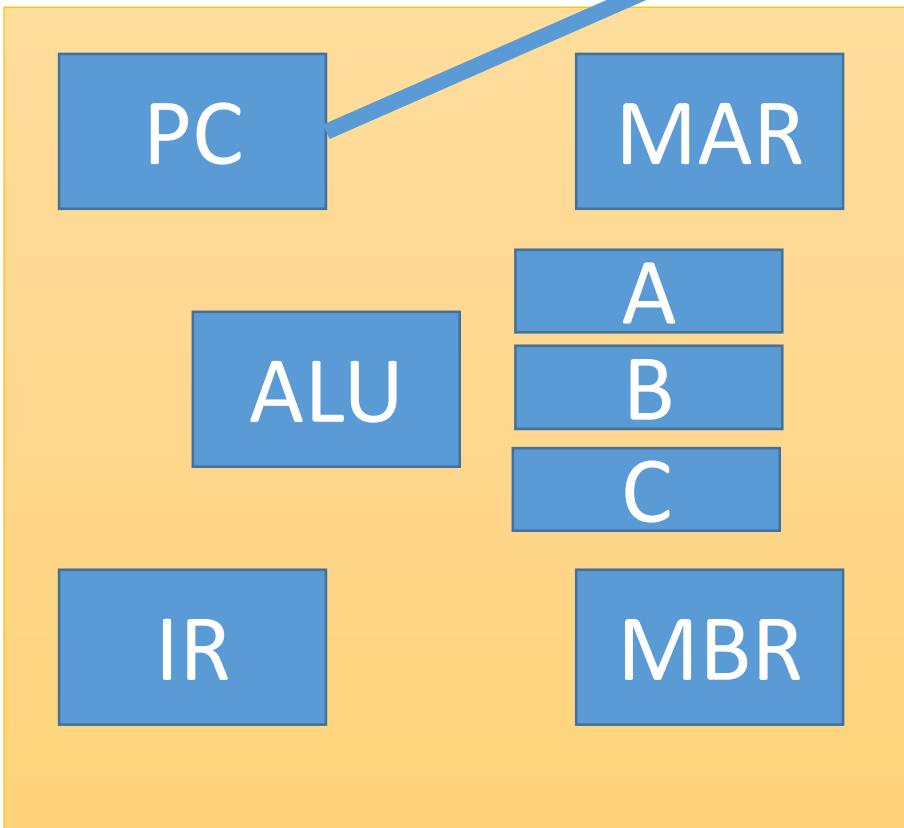
Putting together the Timer and PCB



Putting together the Timer and PCB



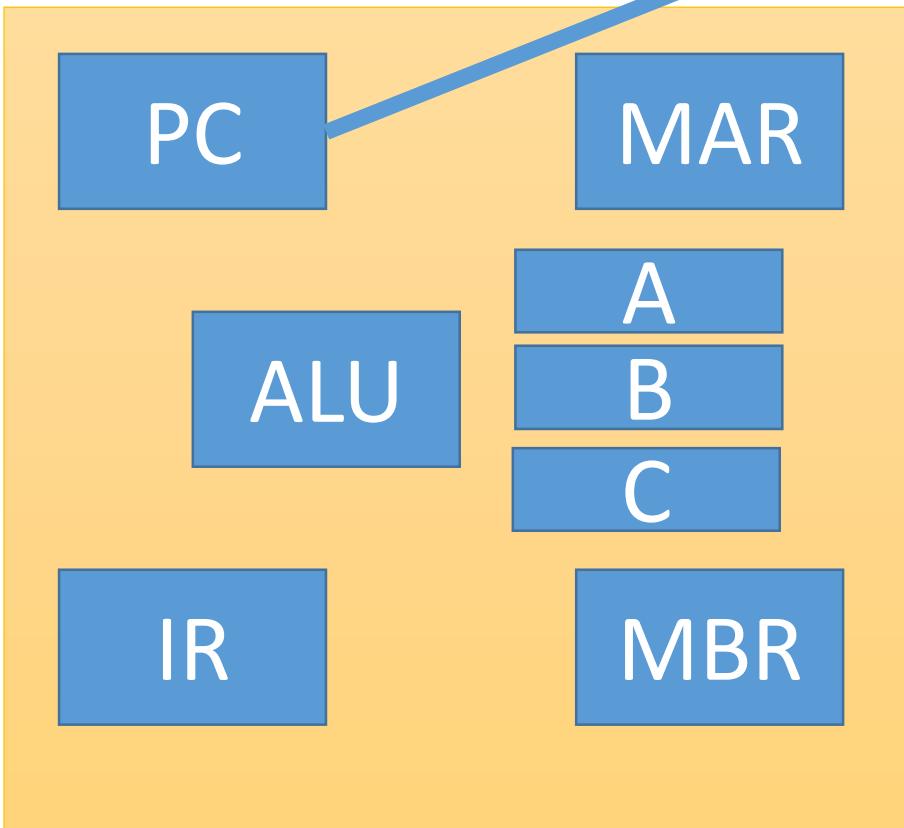
Putting together the Timer and PCB



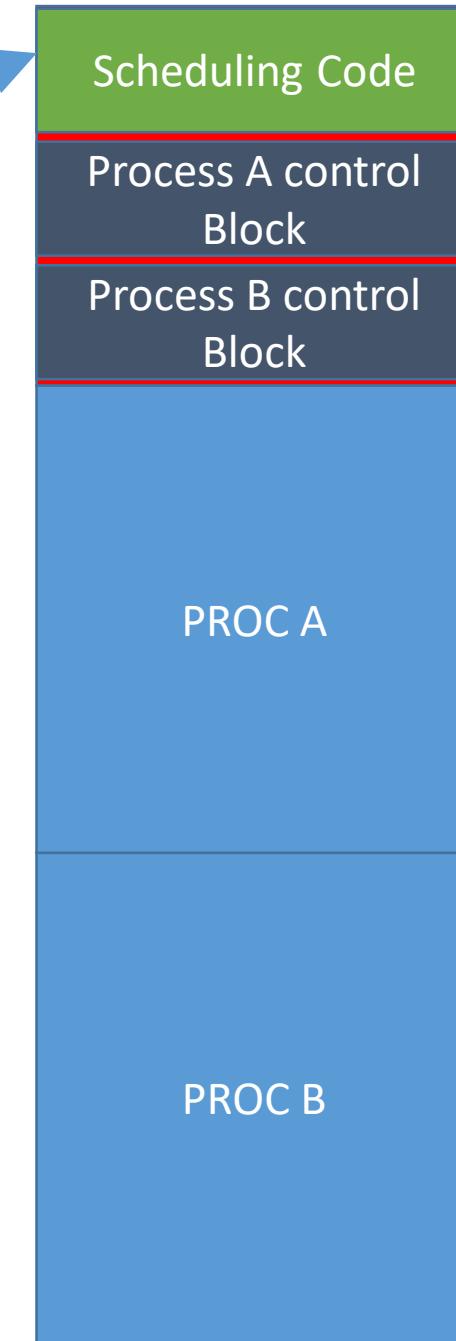
TIMER



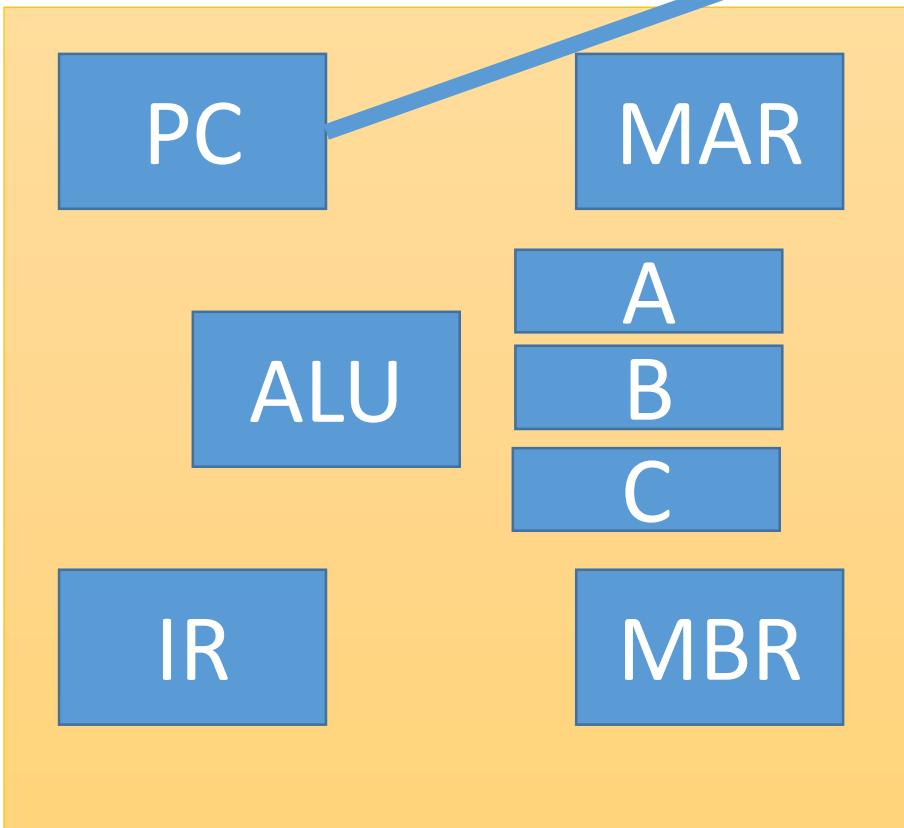
Putting together the Timer and PCB



TIMER



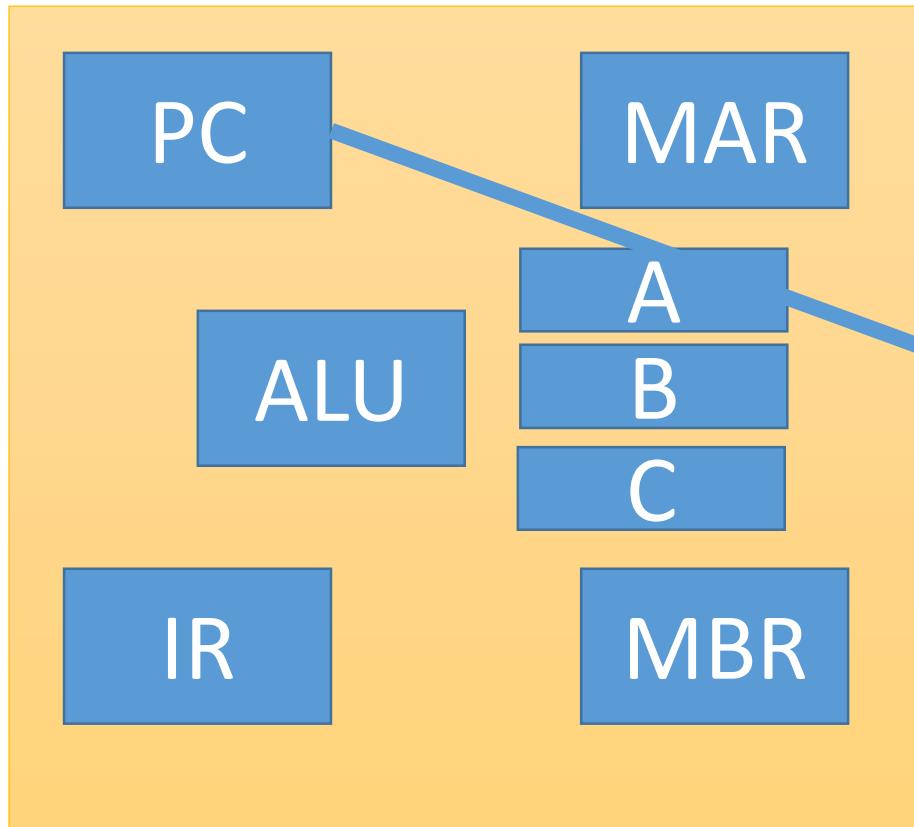
Putting together the Timer and PCB



TIMER



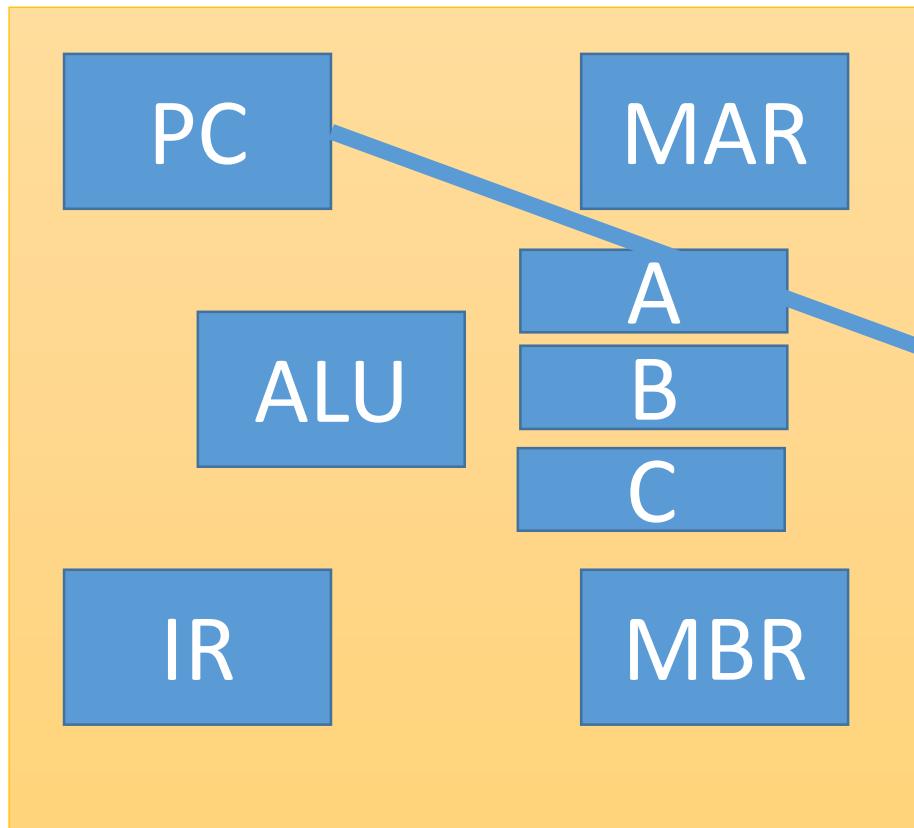
Putting together the Timer and PCB



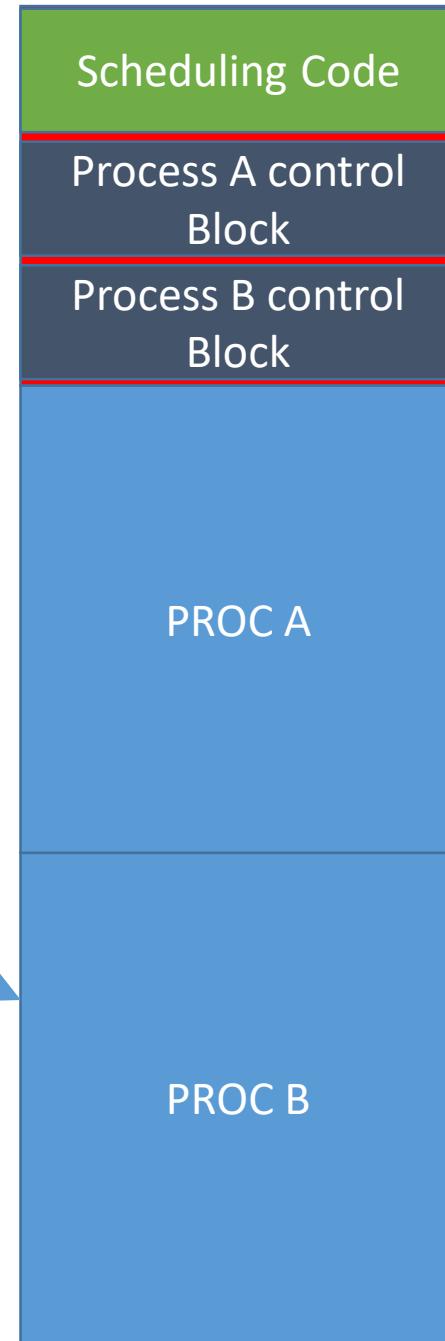
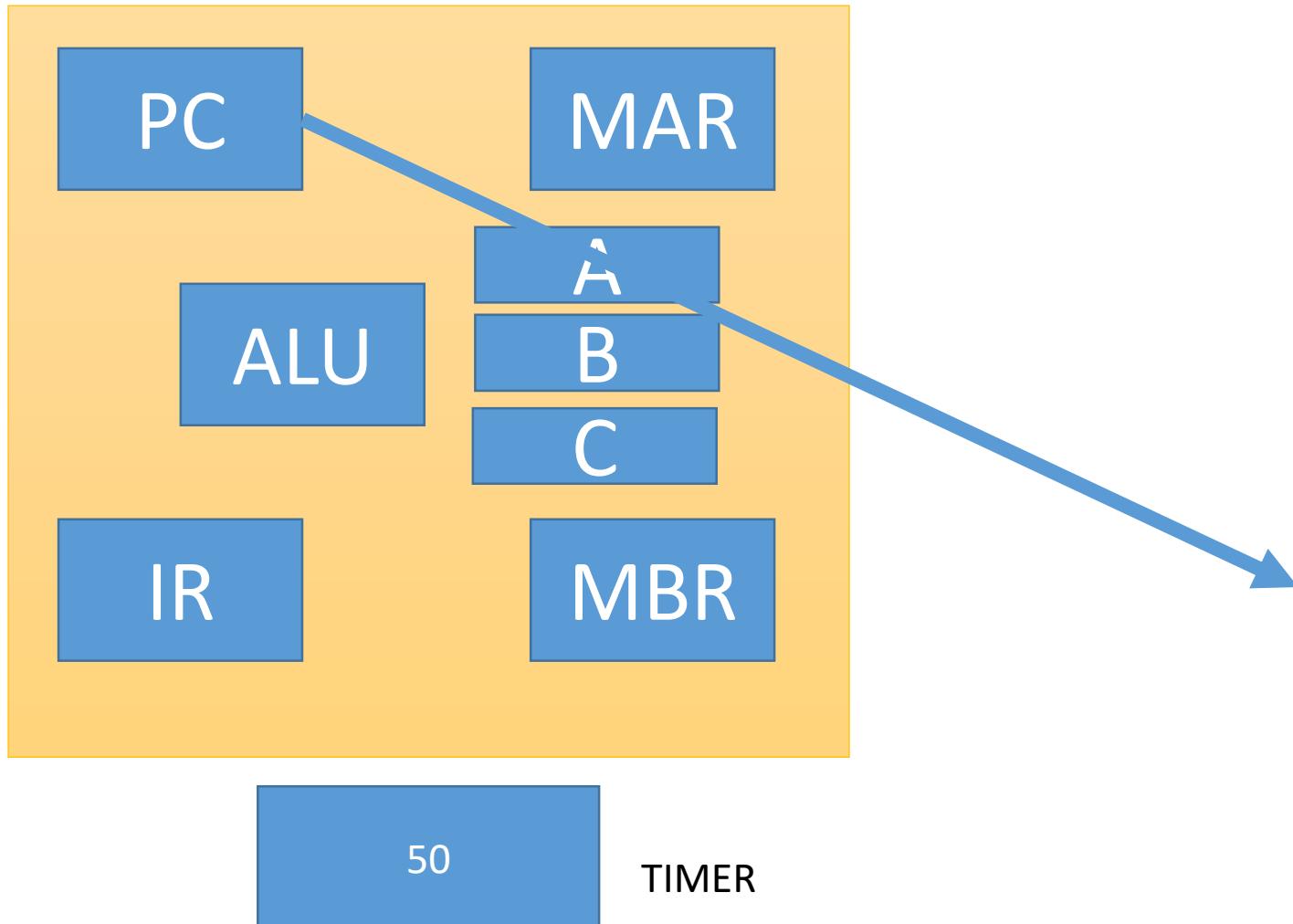
TIMER



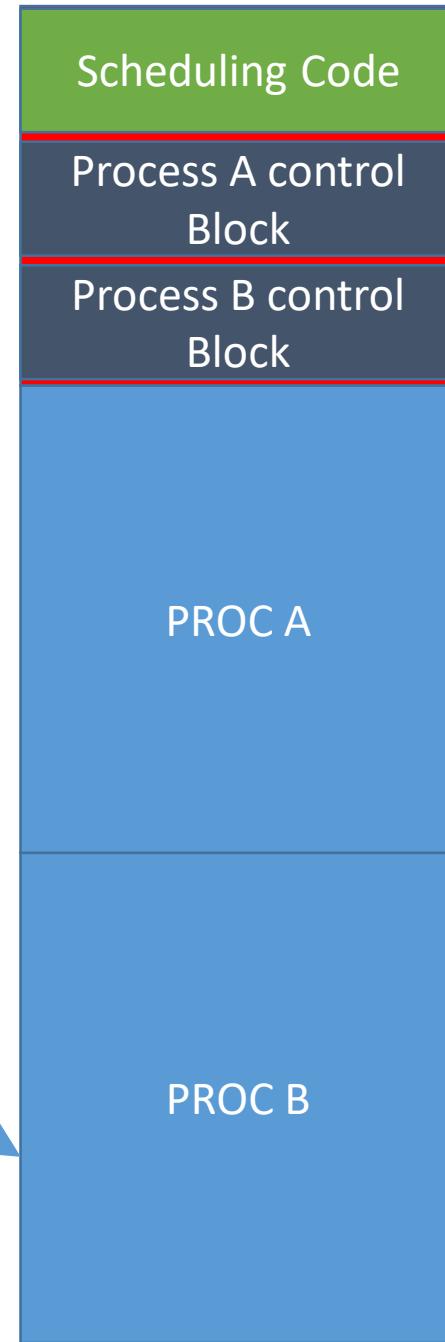
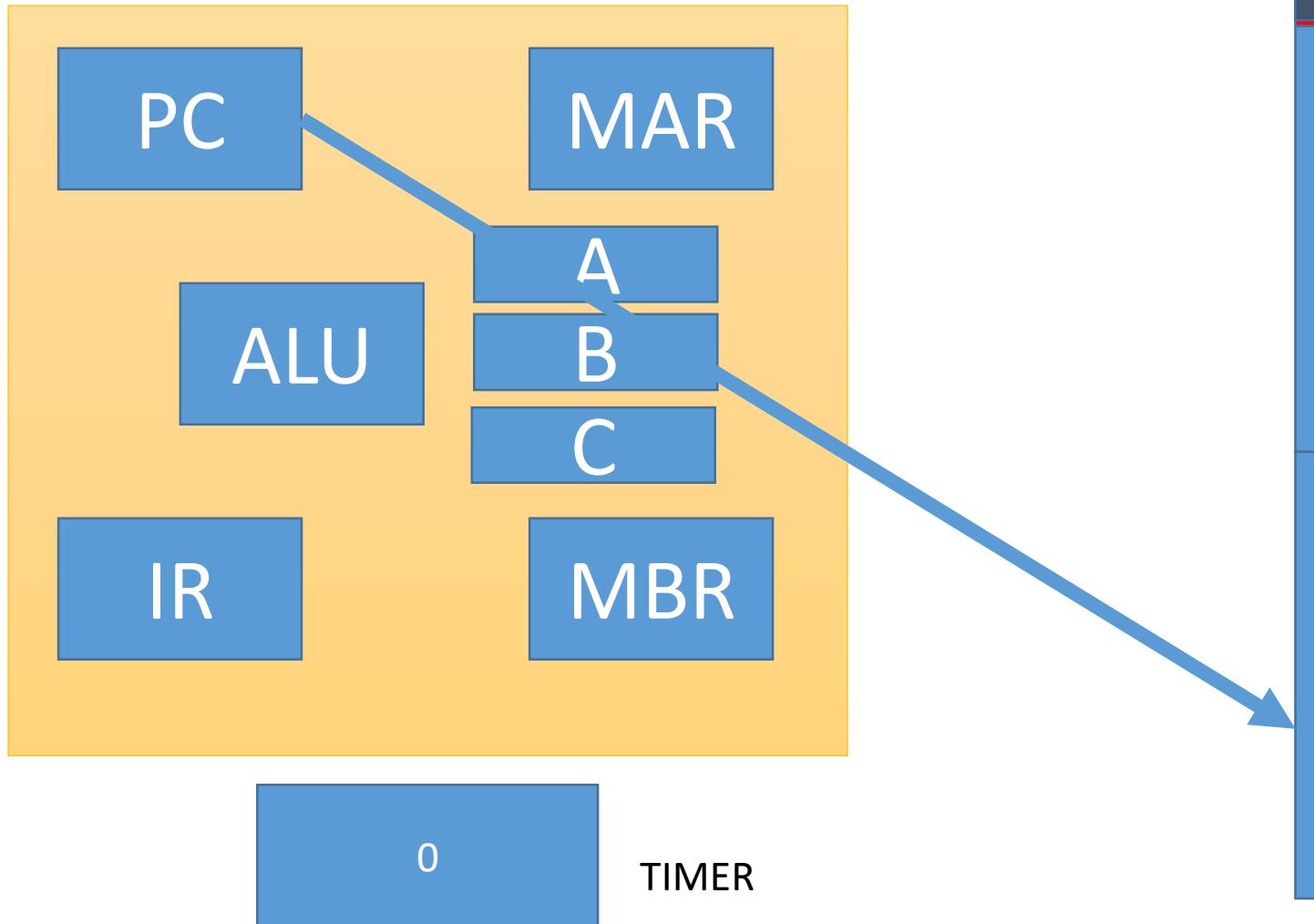
Putting together the Timer and PCB



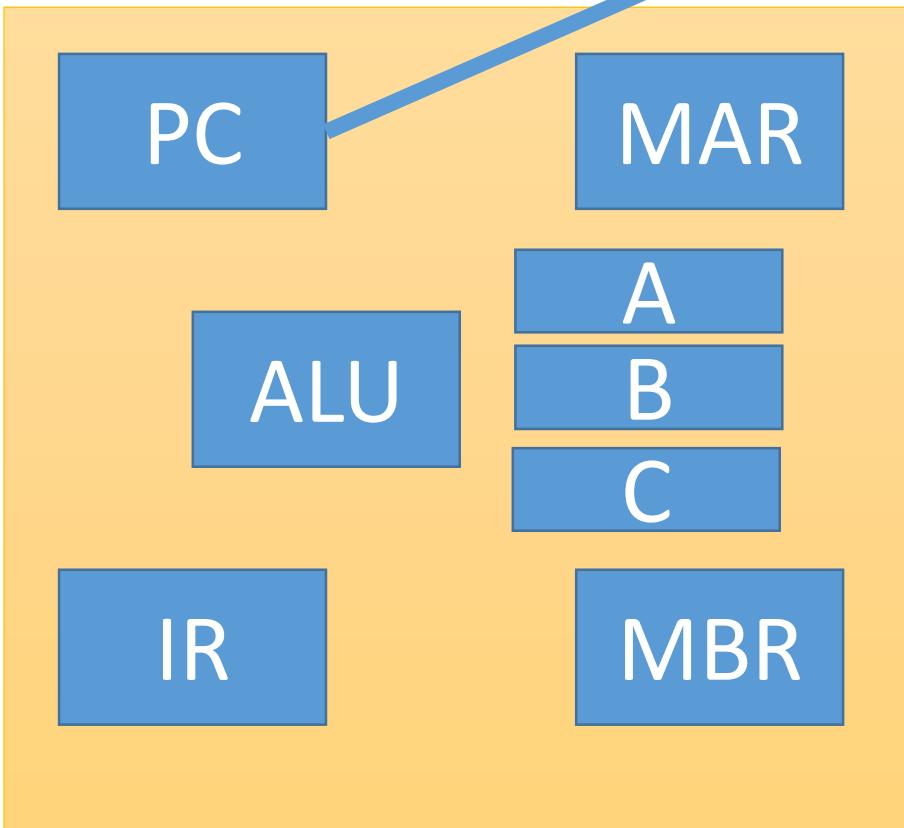
Putting together the Timer and PCB



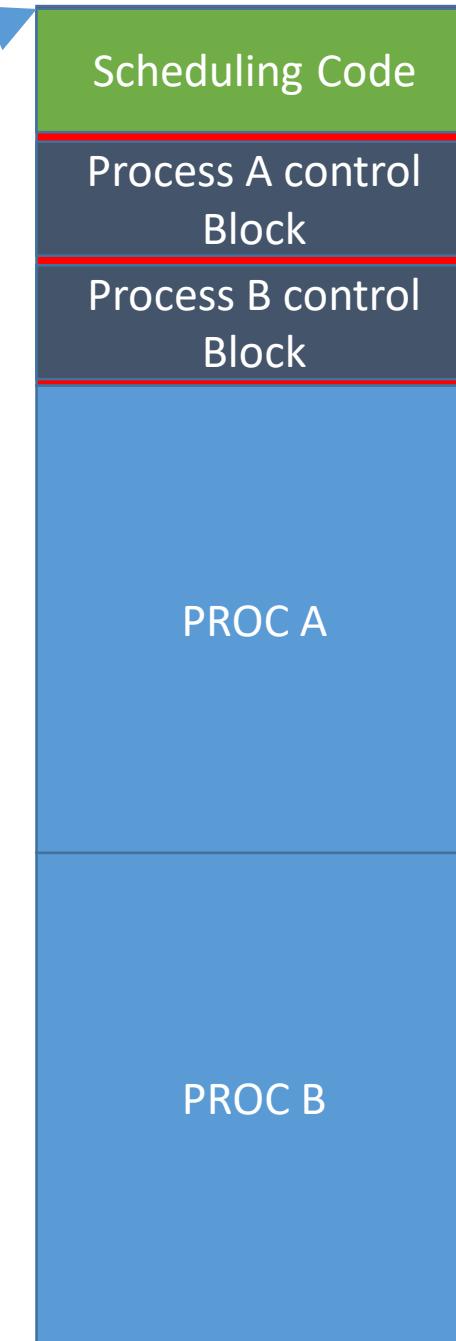
Putting together the Timer and PCB



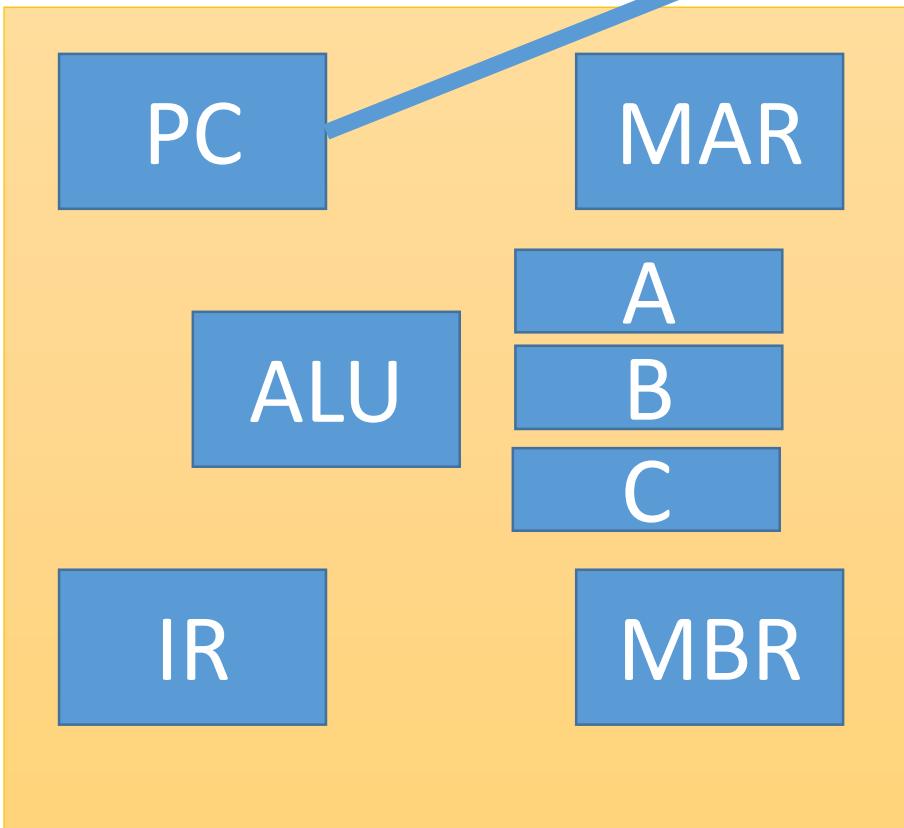
Putting together the Timer and PCB



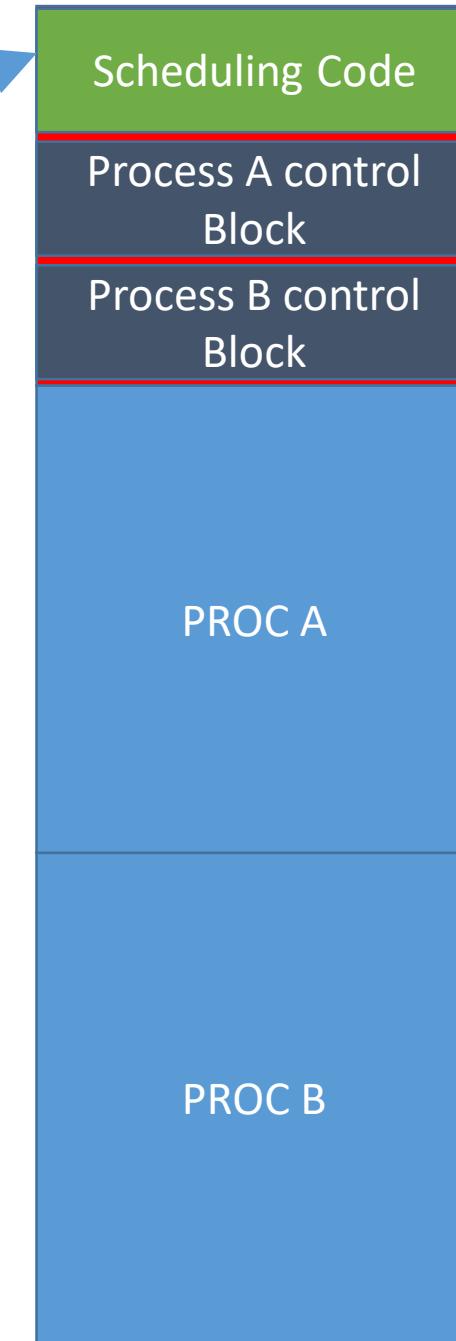
TIMER



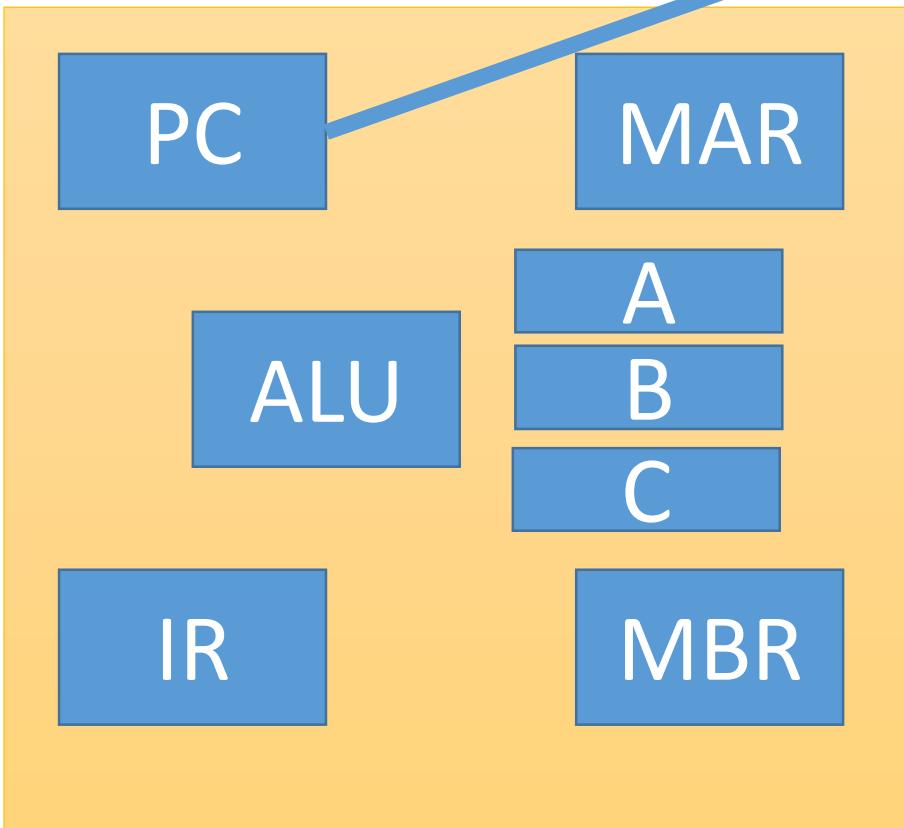
Putting together the Timer and PCB



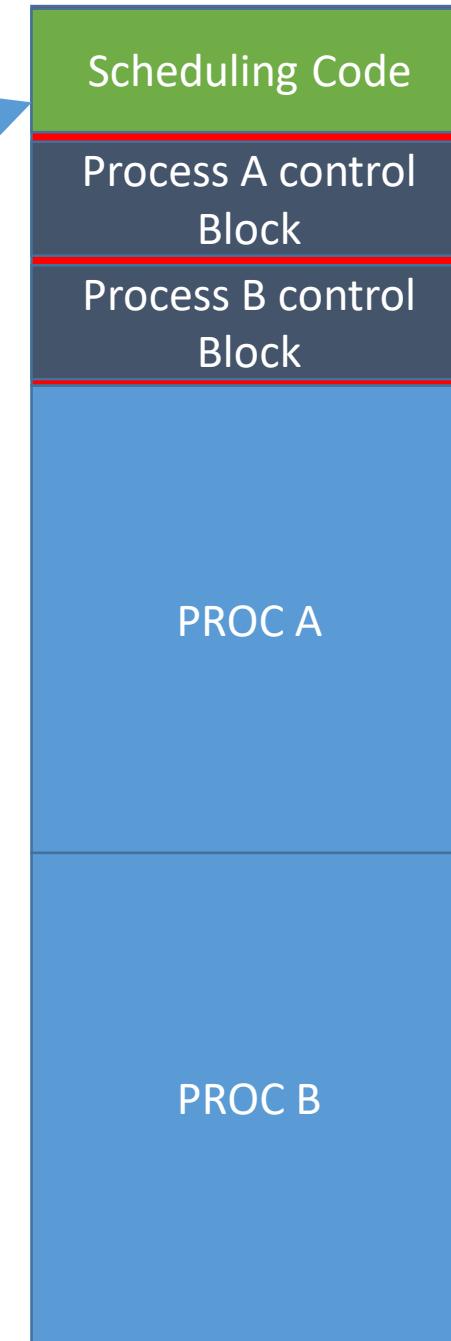
TIMER



Putting together the Timer and PCB



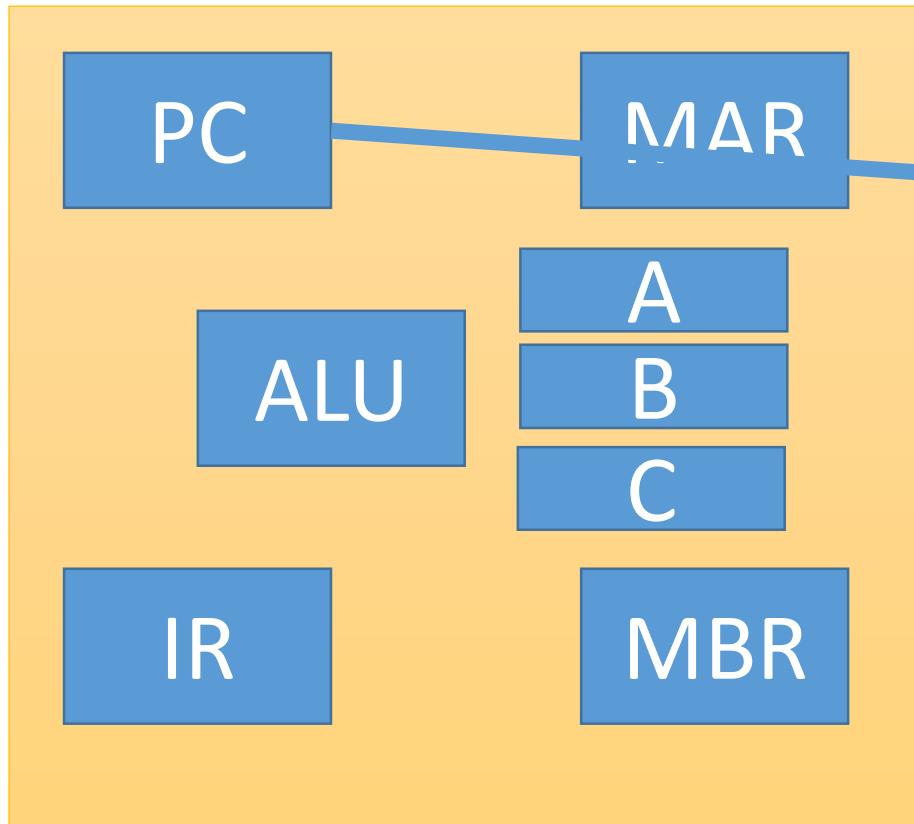
TIMER



Putting together the Timer and PCB

The Proc A
Has been
rescheduled
to run again

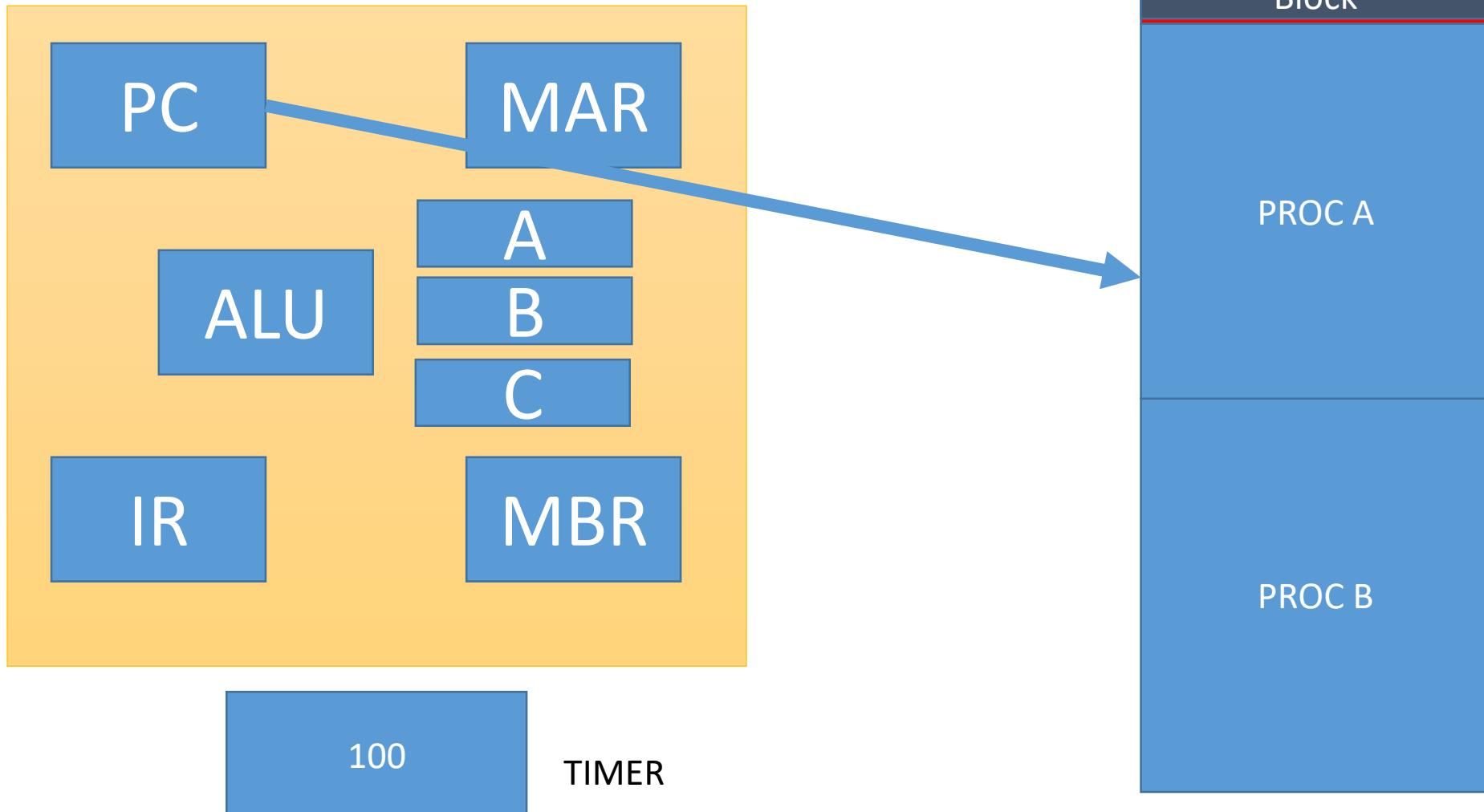
Since all its
information ,
Were
preserved, it
is starting
From the
same place
where it was
before



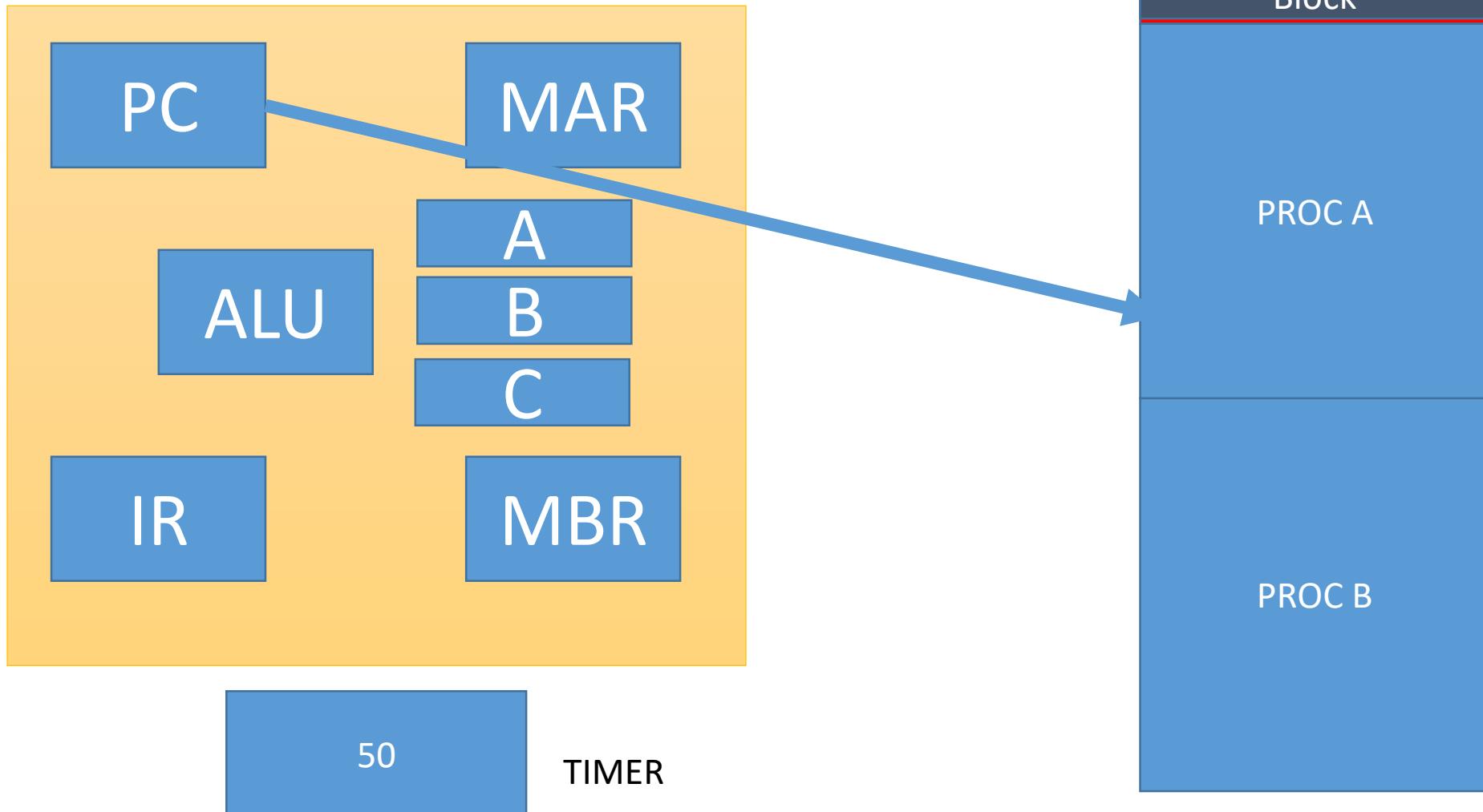
TIMER



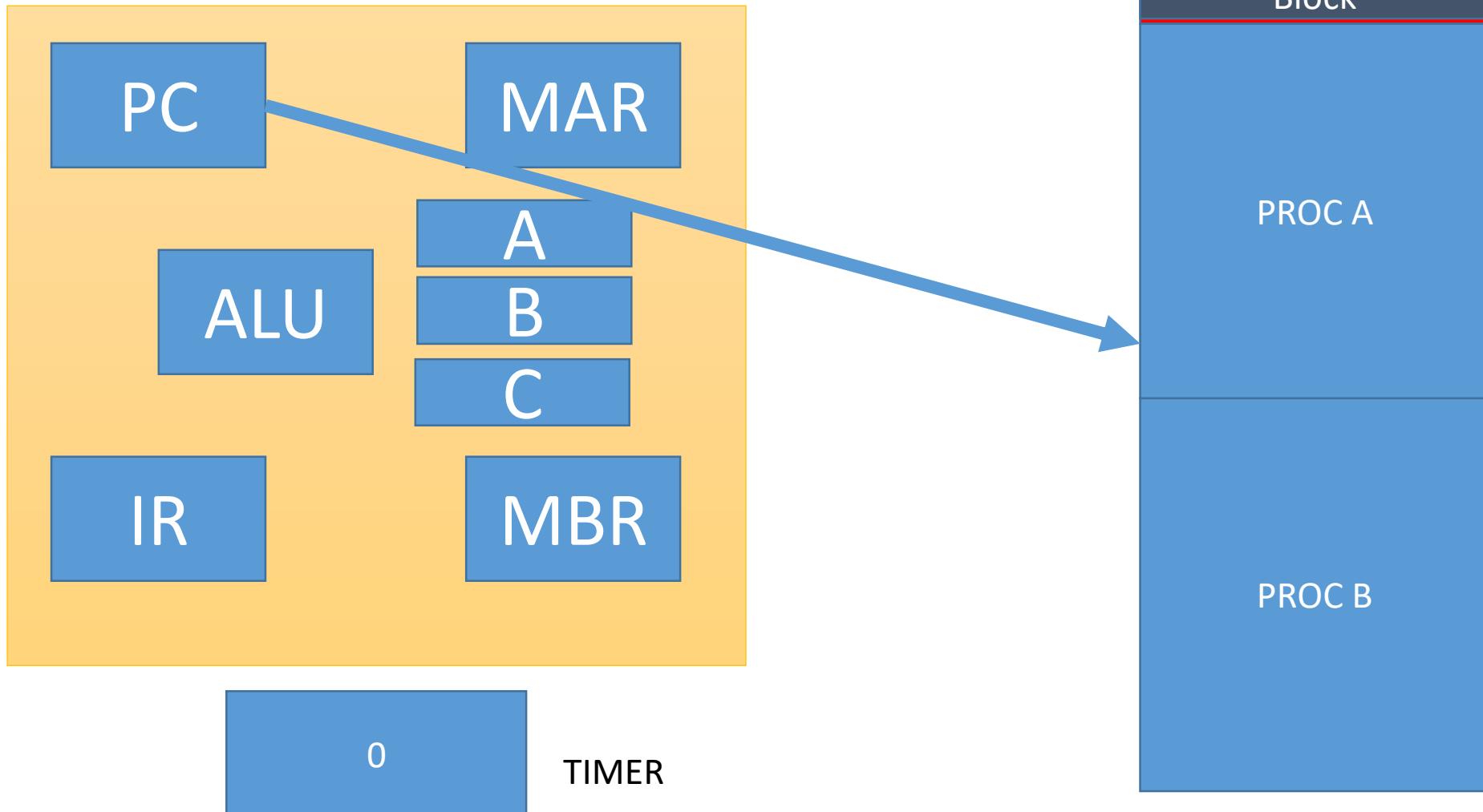
Putting together the Timer and PCB



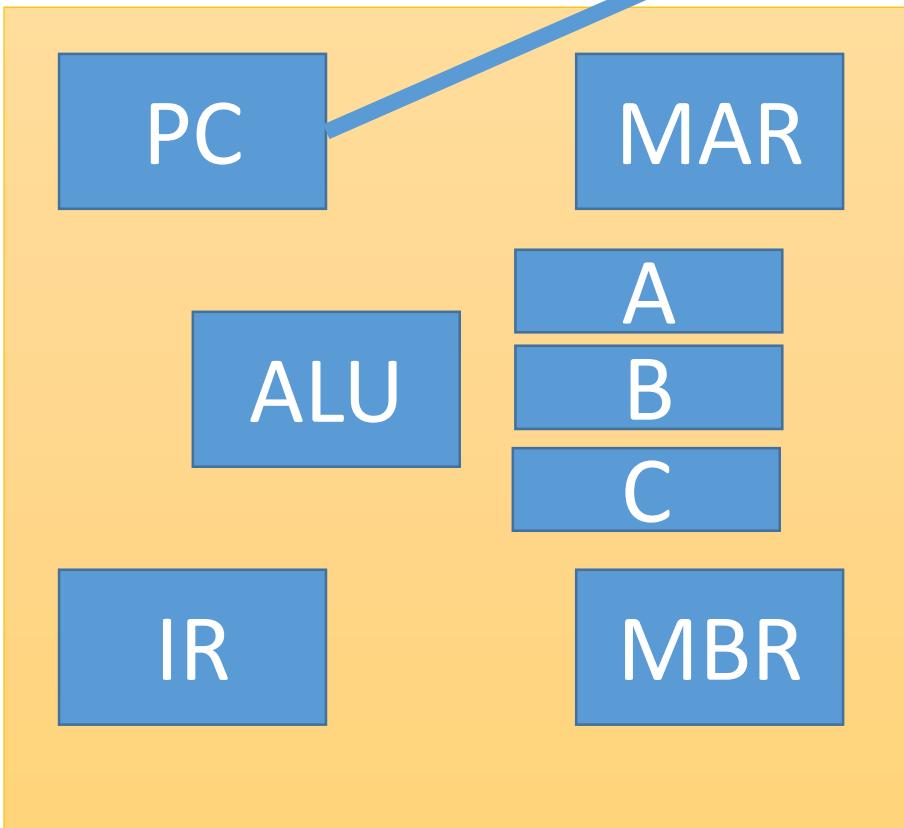
Putting together the Timer and PCB



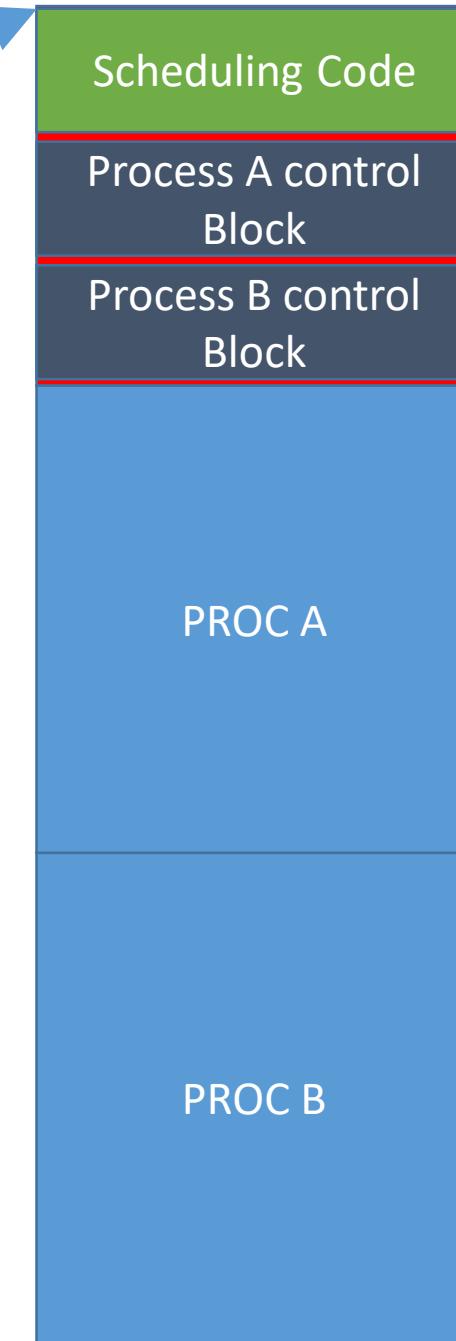
Putting together the Timer and PCB



Putting together the Timer and PCB



TIMER



2.2.Different States of a process

Different States of a Process

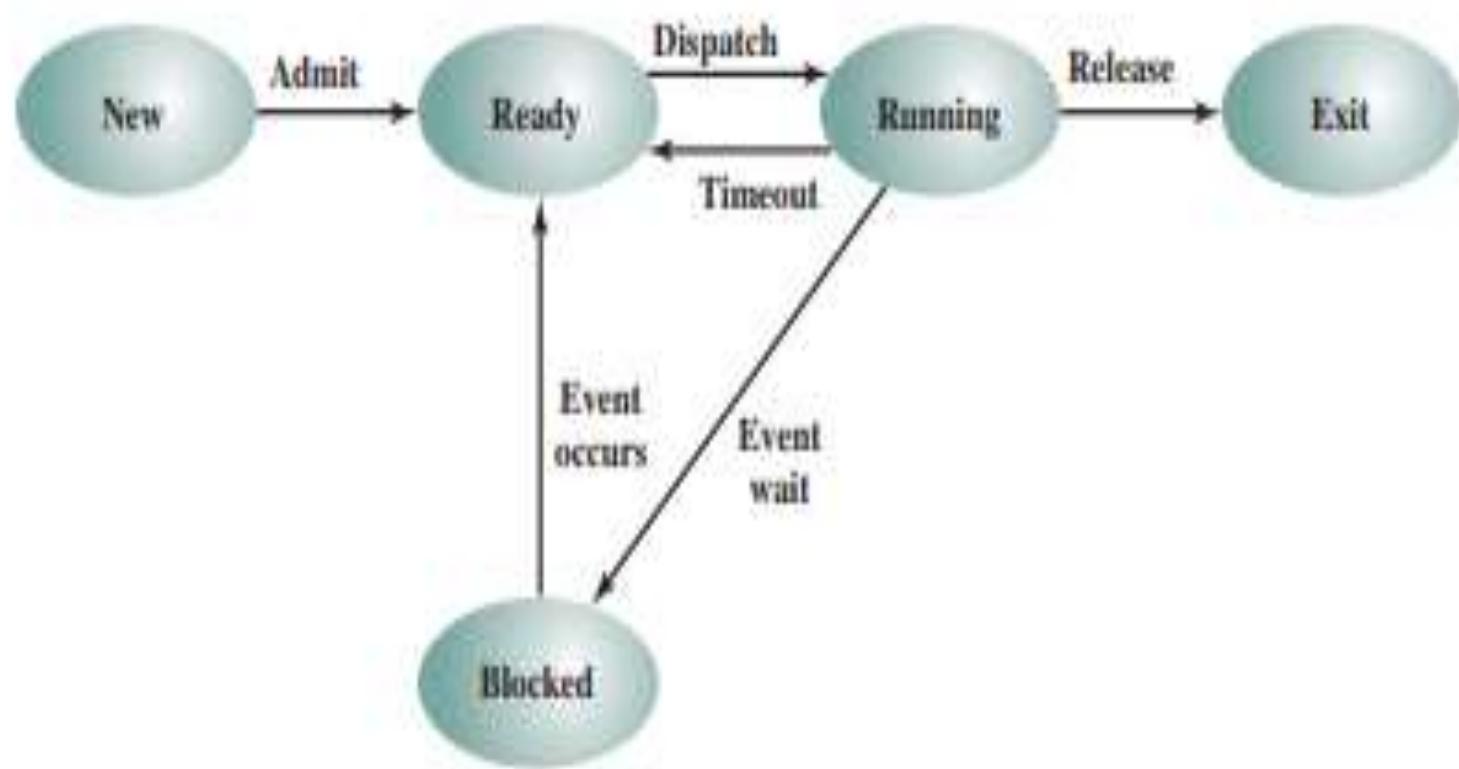


Figure 8.7 Five-State Process Model

New- program clicked to be initiated.

Ready-The process's code is in the memory along with meta data in PCB

Running-The process is being executed in processor

Blocked- Not running, but it may run later on, its necessary information are stored before switching to another processor.

Exit-The process has terminated, soon its code and meta data will be evicted.

Putting the bits and bytes of scheduling together

The data structure maintained by OS:

OS maintains some queues.

Long term queue - List of programs that are **waiting to enter the system**. (say the user has clicked on 4 different programs)

This queue is used by the **high level scheduler**. Based on this queue, it will allocate memory for process codes and hence bring them into the **ready state**.

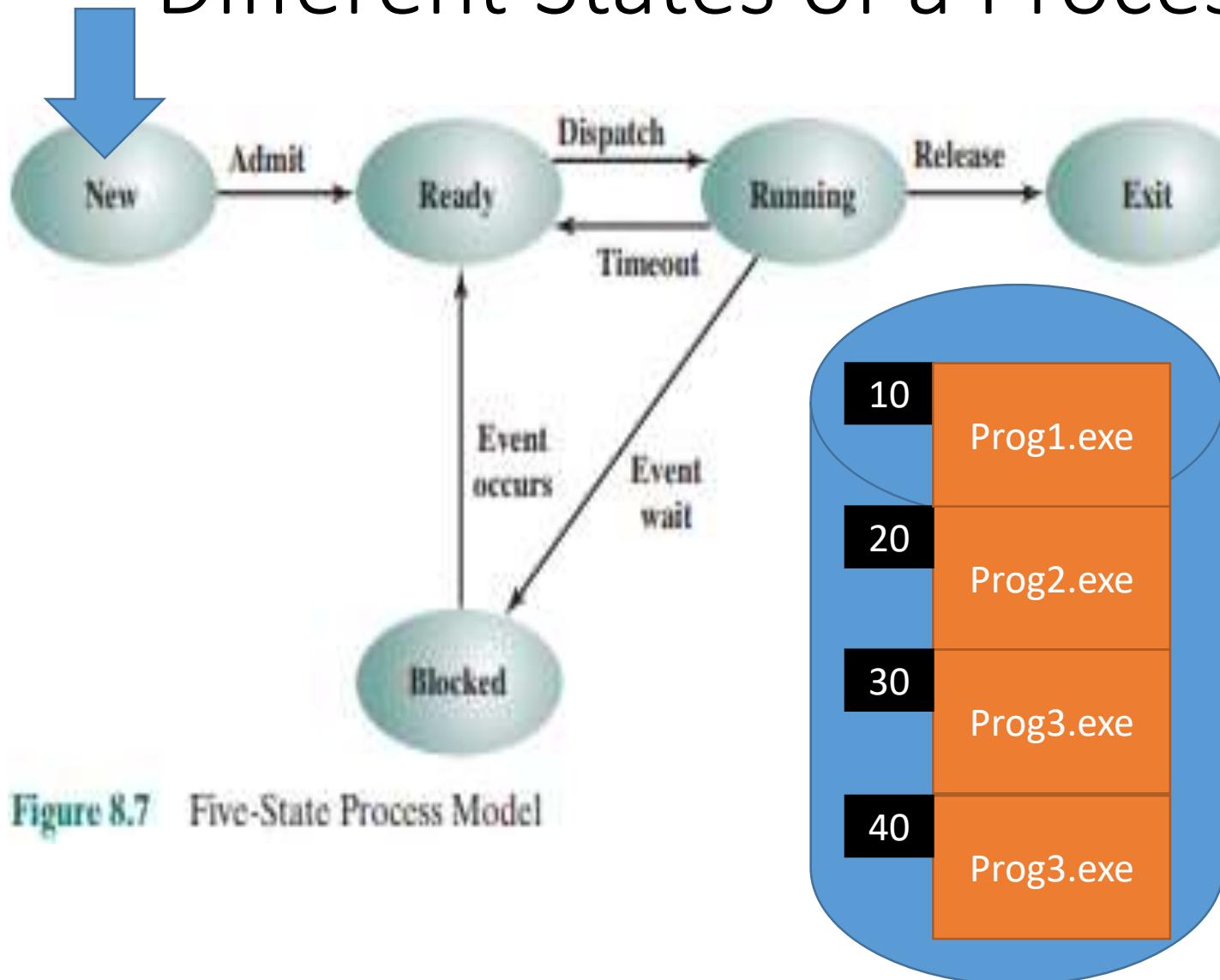
Short term queue - Consists of all the processes in the **ready state**.

The short term scheduler will pick which process to run next.

Intermediate queue: A blocked process which was running in the memory, but blocked due to some operations and is no longer in the memory (we will see this in detail **later**)

I/O Queue - Multiple processes may **request** for an **input/output device** at a time. Hence these requests are inserted into a queue. From which later on the access is given.

Different States of a Process

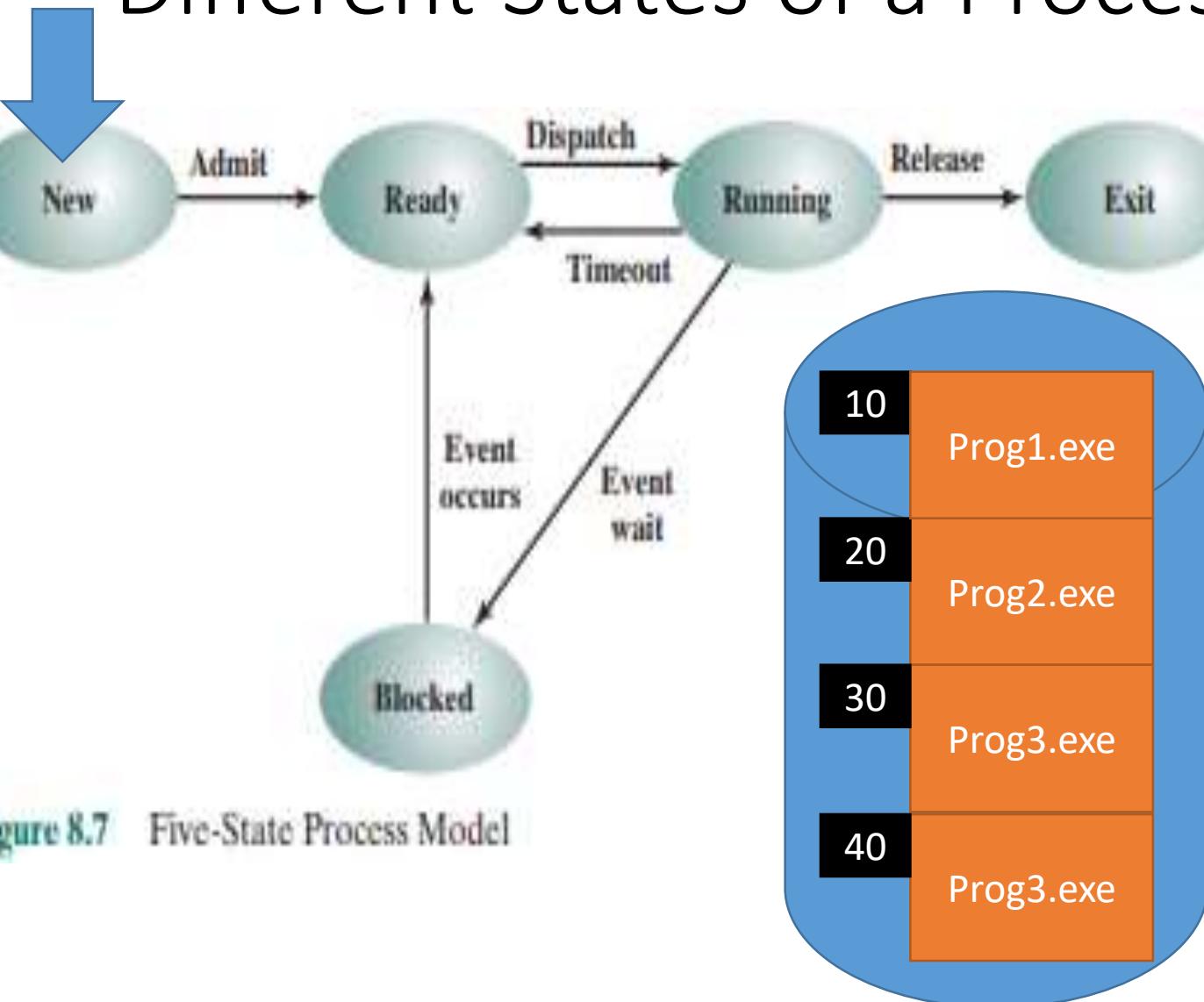


Suppose the user wants to run **prog1.exe**



Figure 8.7 Five-State Process Model

Different States of a Process



Its address will be provided to the **long term queue** of the OS.

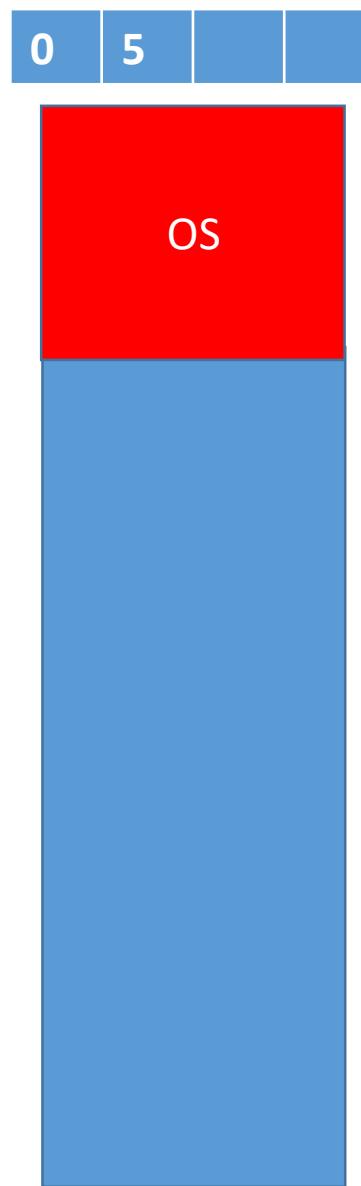
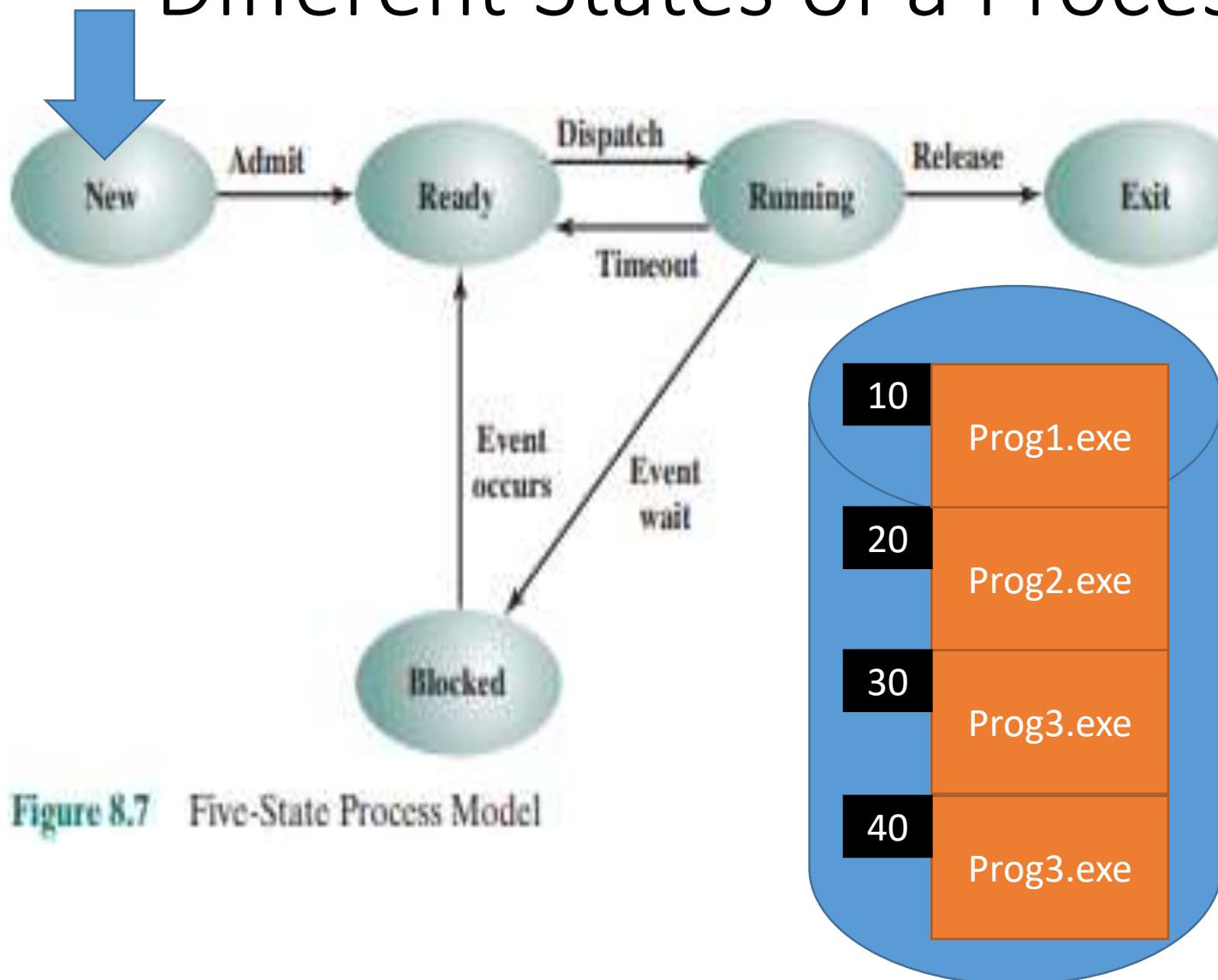


Figure 8.7 Five-State Process Model

Different States of a Process



Its address will be provided to the **long term queue** of the OS.

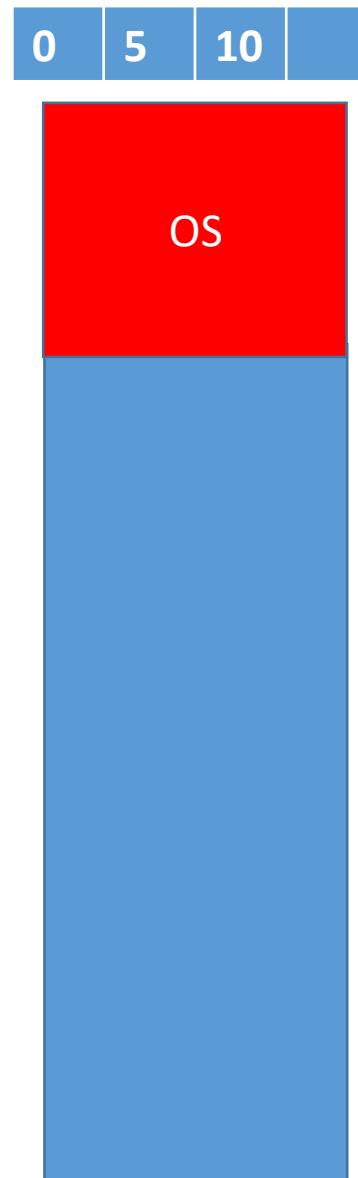
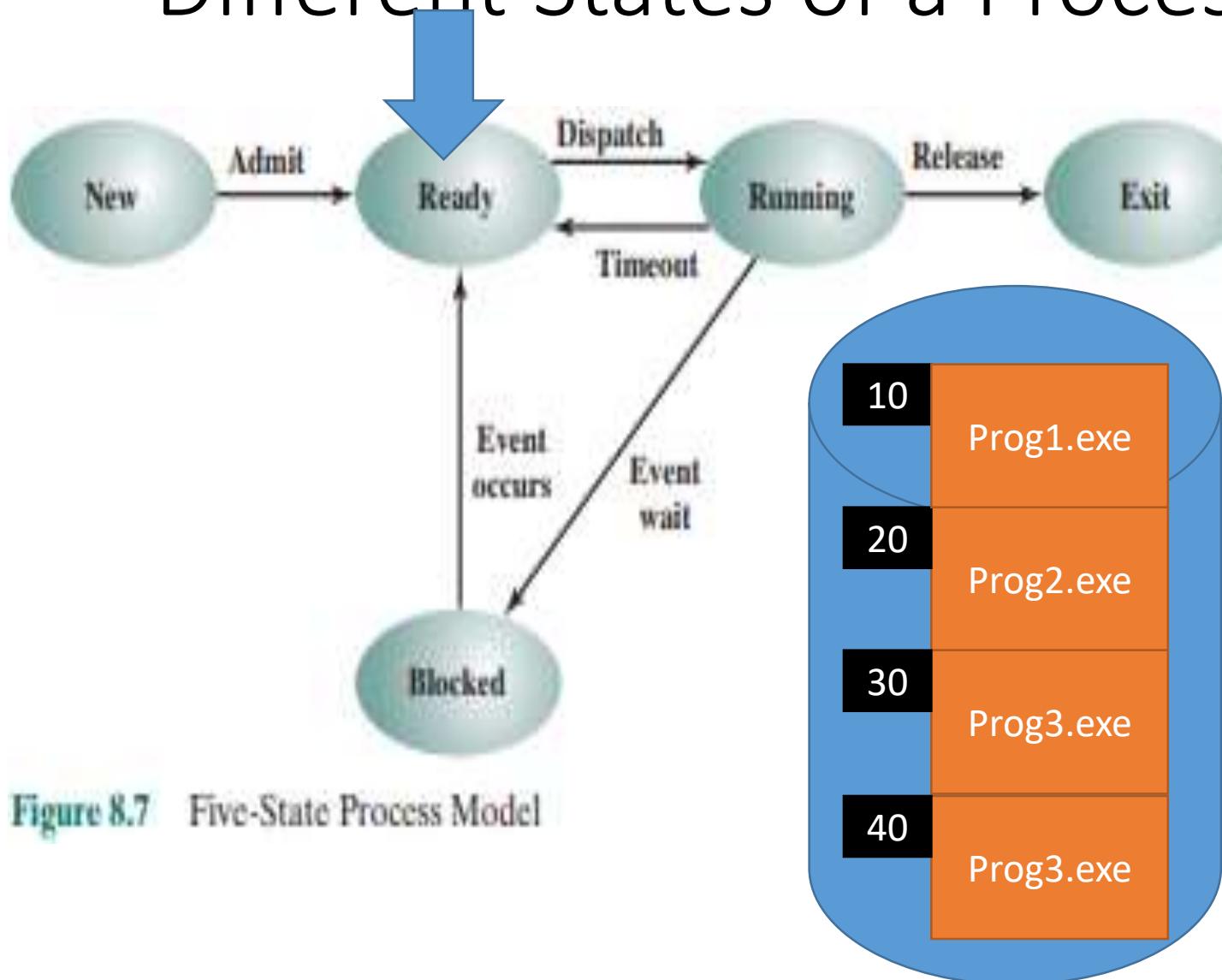


Figure 8.7 Five-State Process Model

Different States of a Process



Its address will be provided to the **long term queue** of the OS.

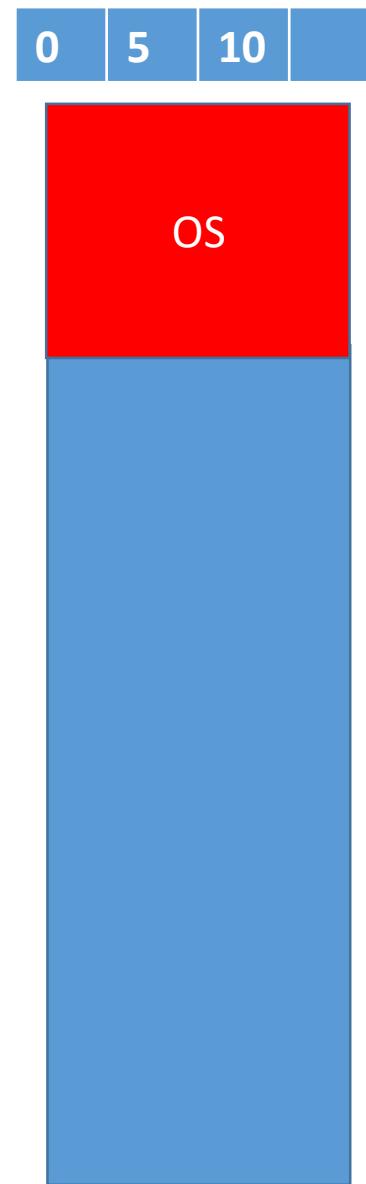


Figure 8.7 Five-State Process Model

Now its code and metadata
Are provided into the RAM

Different States of a Process

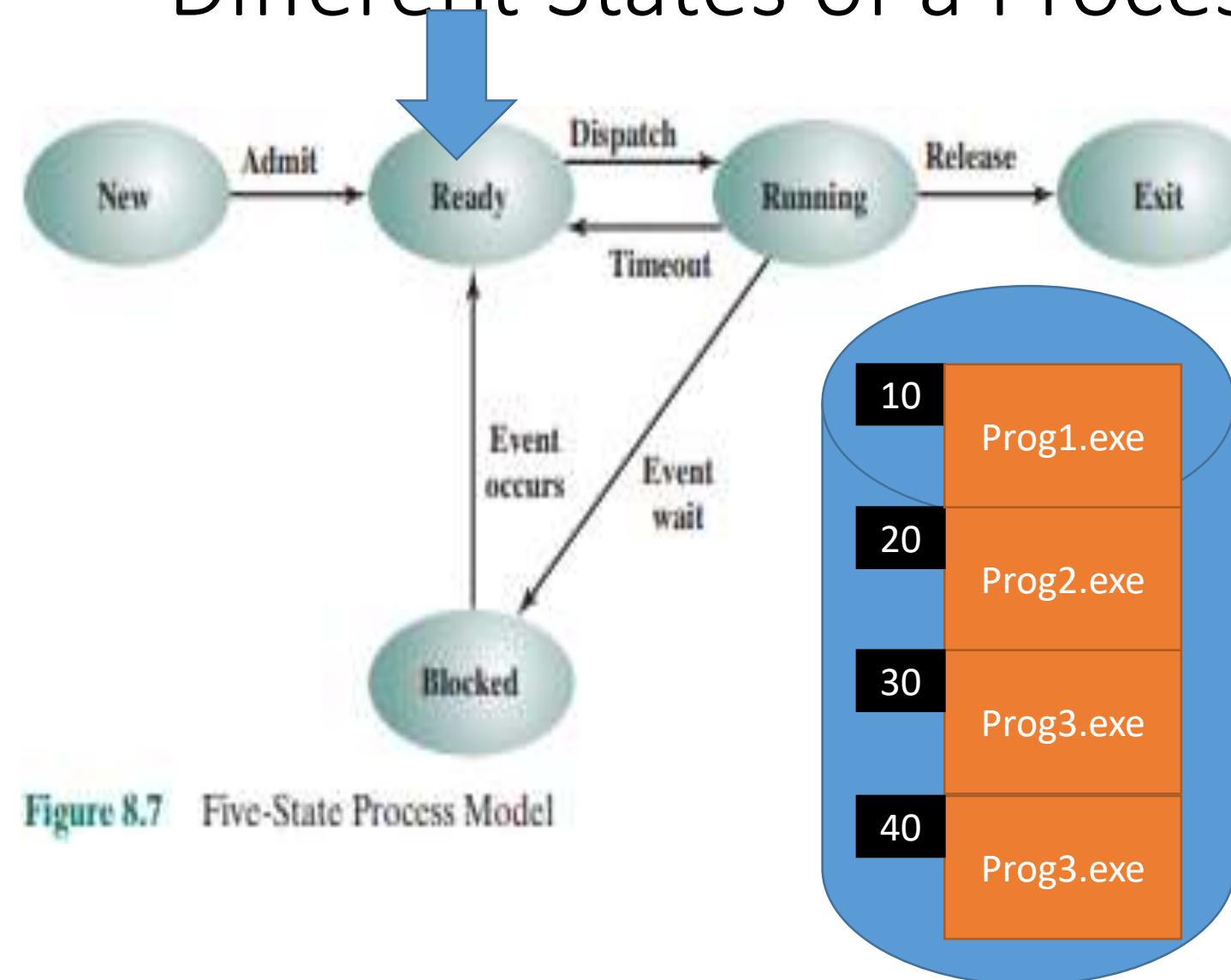
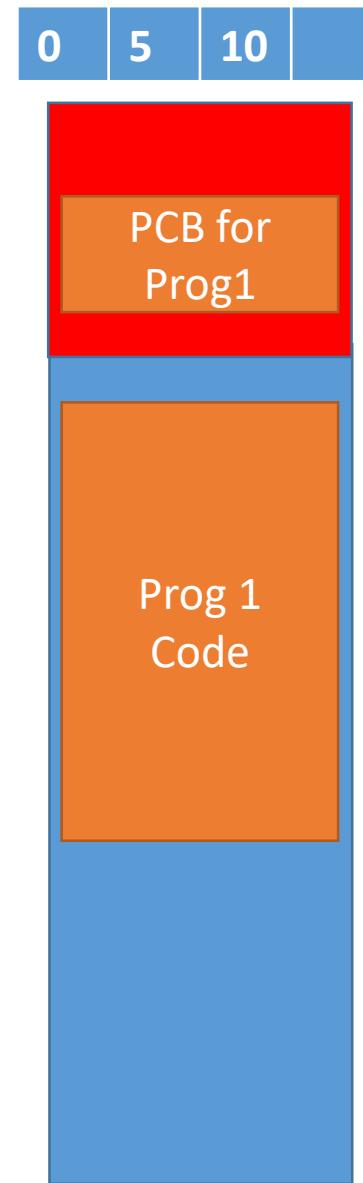


Figure 8.7 Five-State Process Model



Different States of a Process

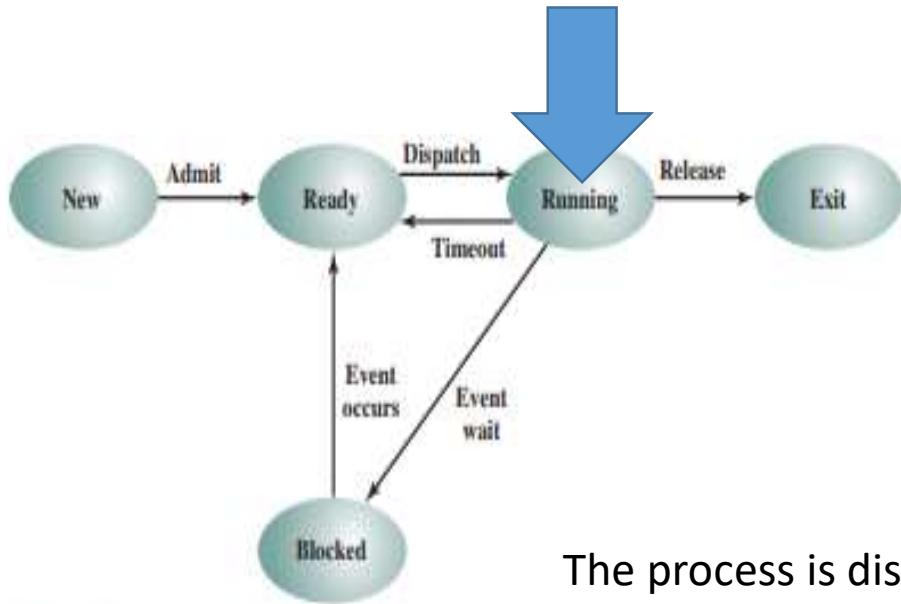


Figure 8.7 Five-State Process Model

The process is dispatched by the scheduler.

i.e the PC begins to take in instruction from it



Different States of a Process

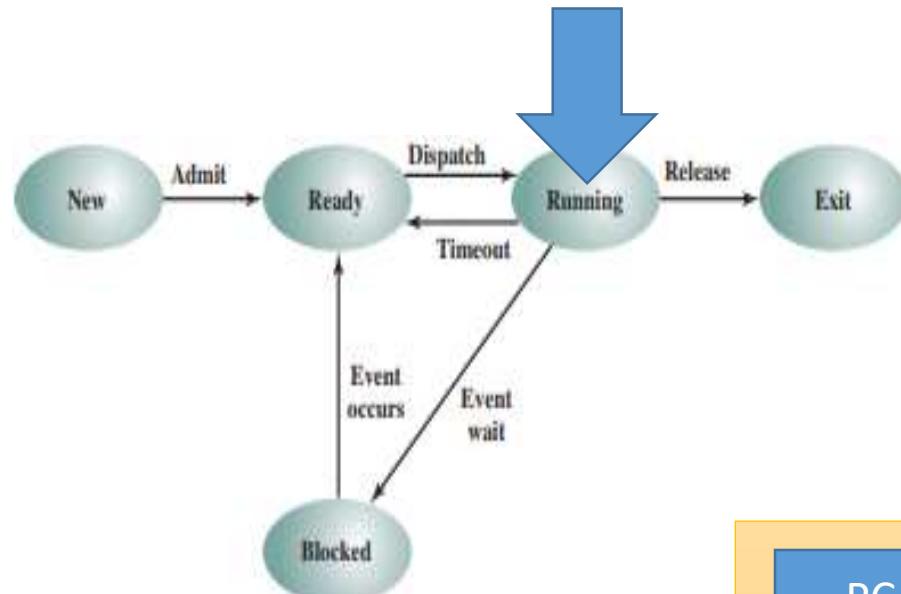
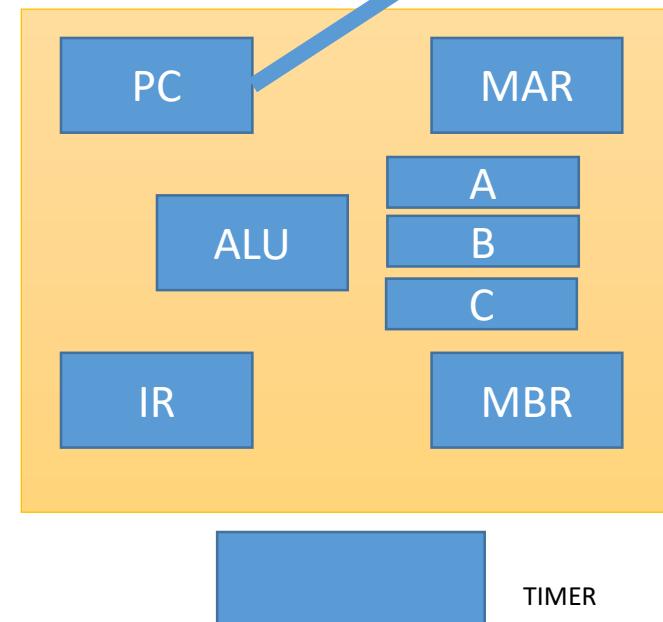


Figure 8.7 Five-State Process Model



Different States of a Process

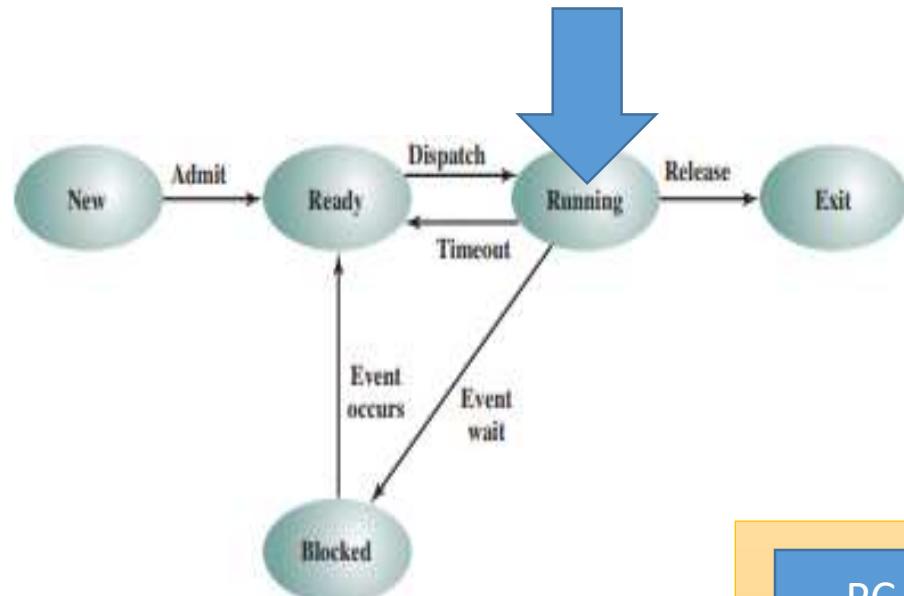
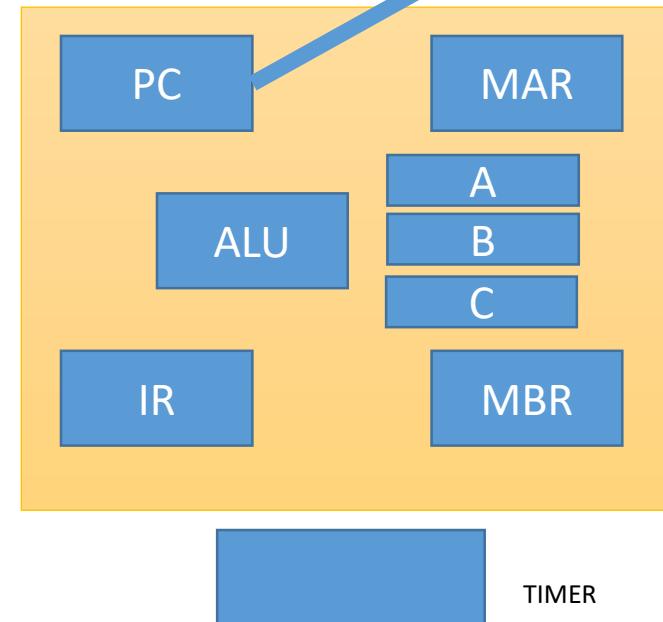


Figure 8.7 Five-State Process Model



Different States of a Process

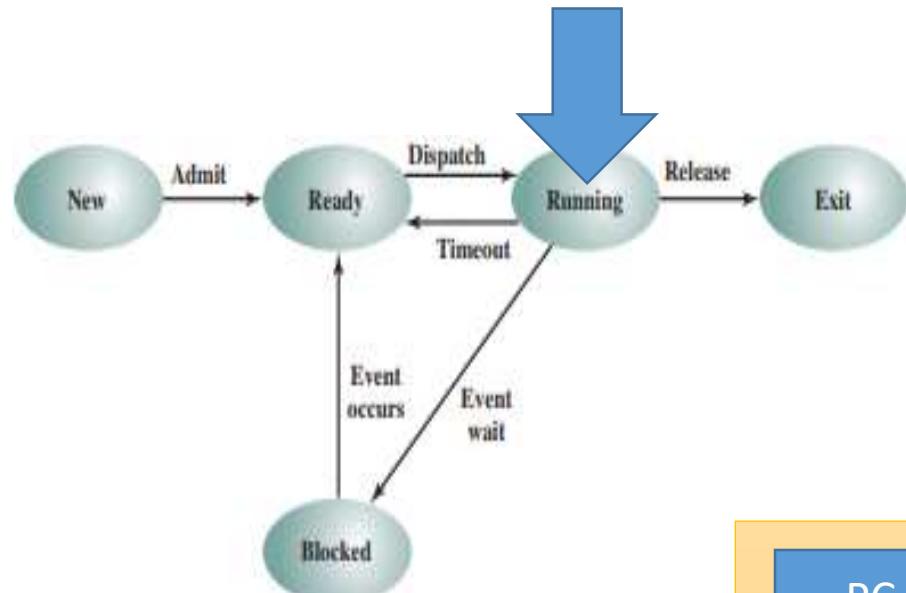
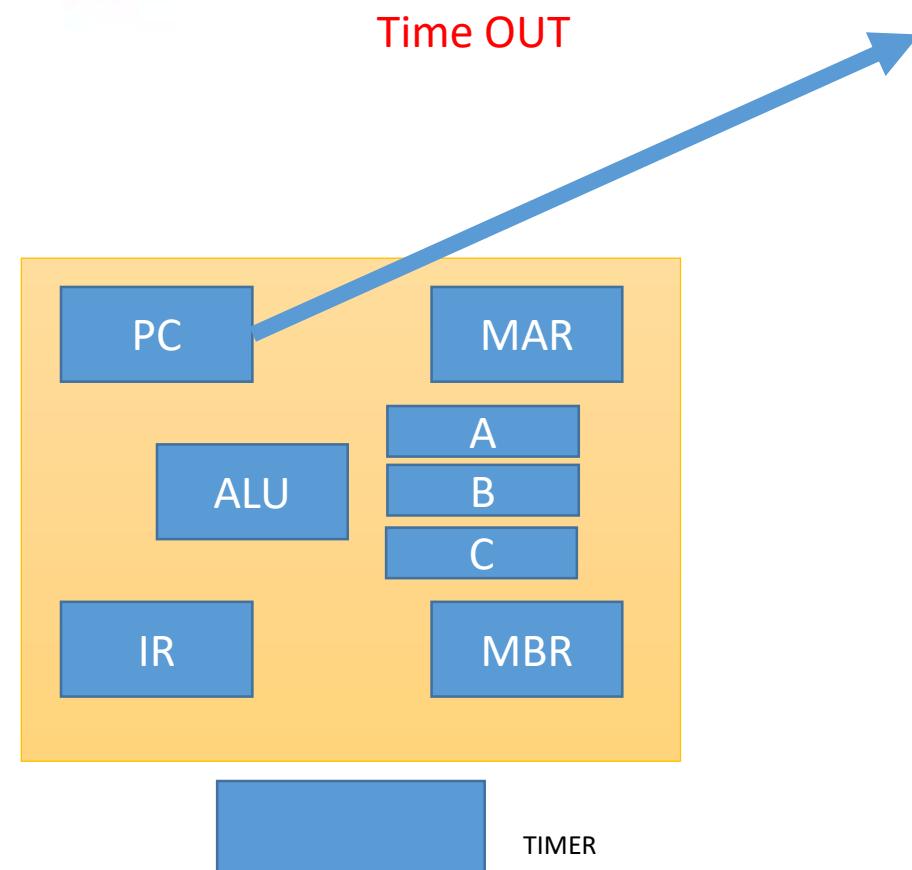
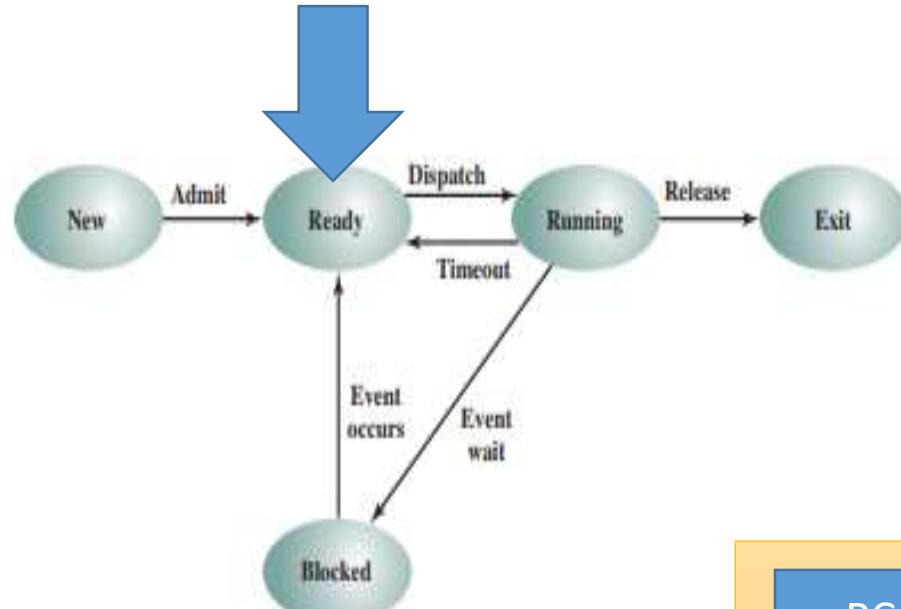


Figure 8.7 Five-State Process Model



Different States of a Process



The process is in ready state again

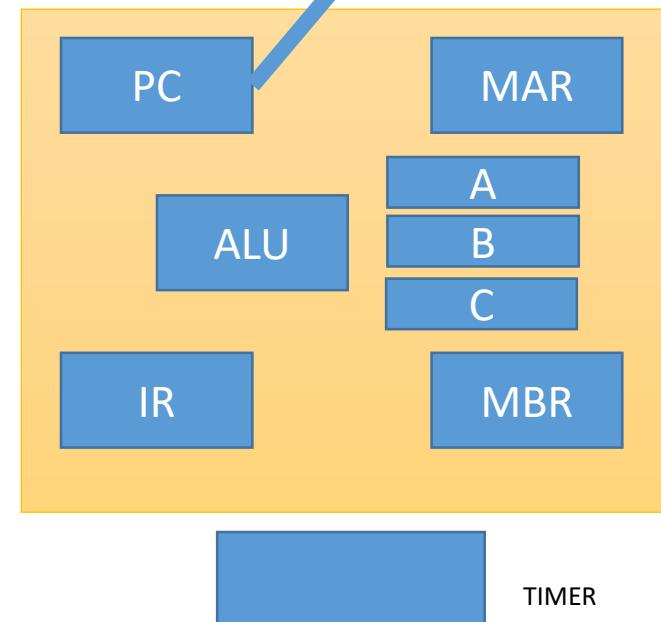
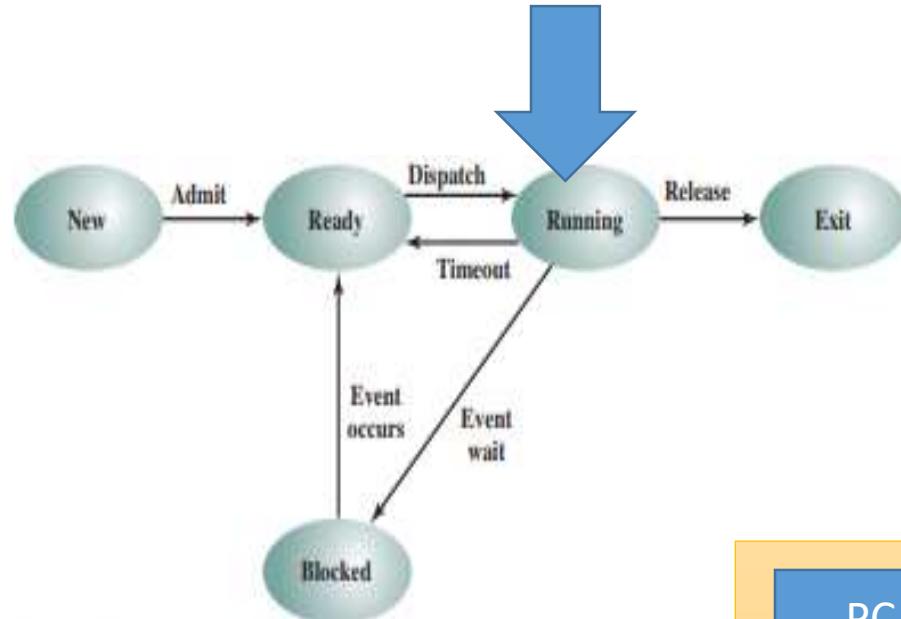


Figure 8.7 Five-State Process Model

Different States of a Process



Lets assume,
that the
scheduler
decided to run
this process
again

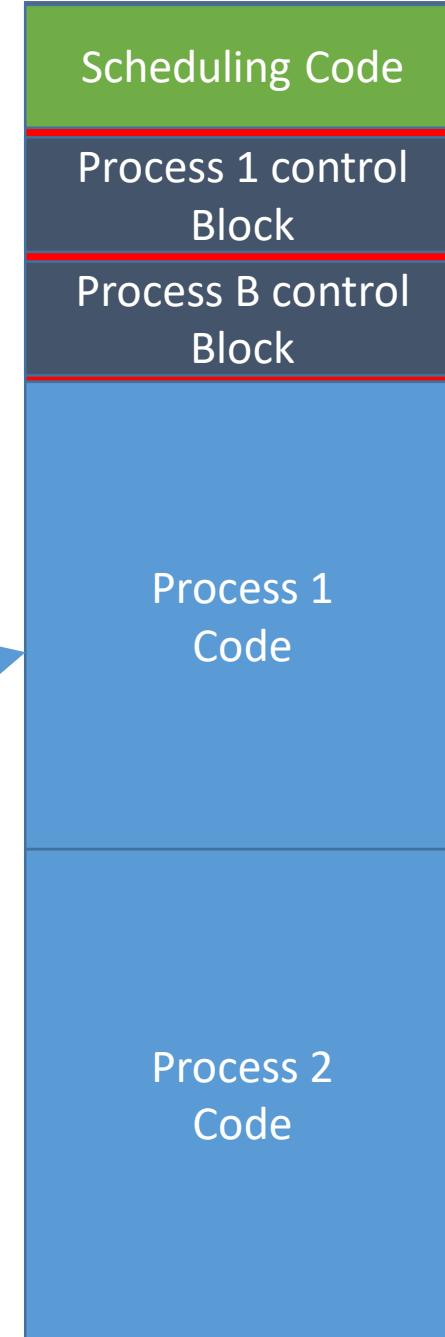
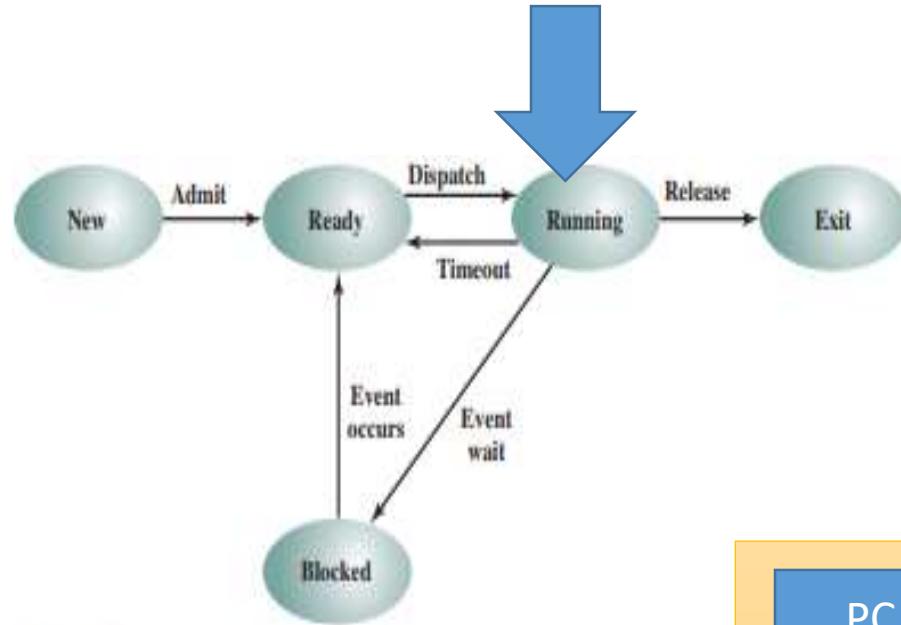


Figure 8.7 Five-State Process Model

Different States of a Process



Lets assume,
that the
scheduler
decided to run
this process
again

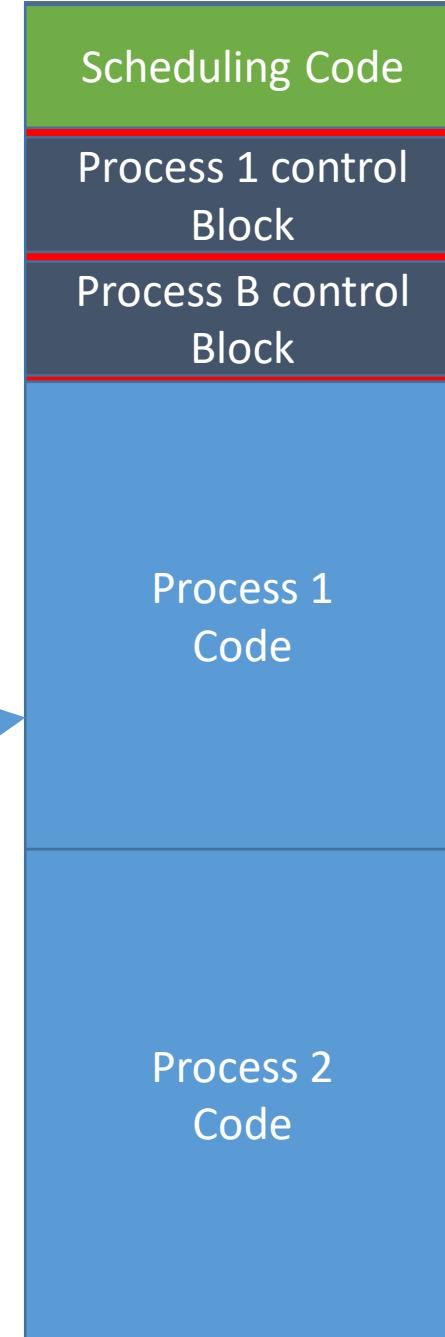
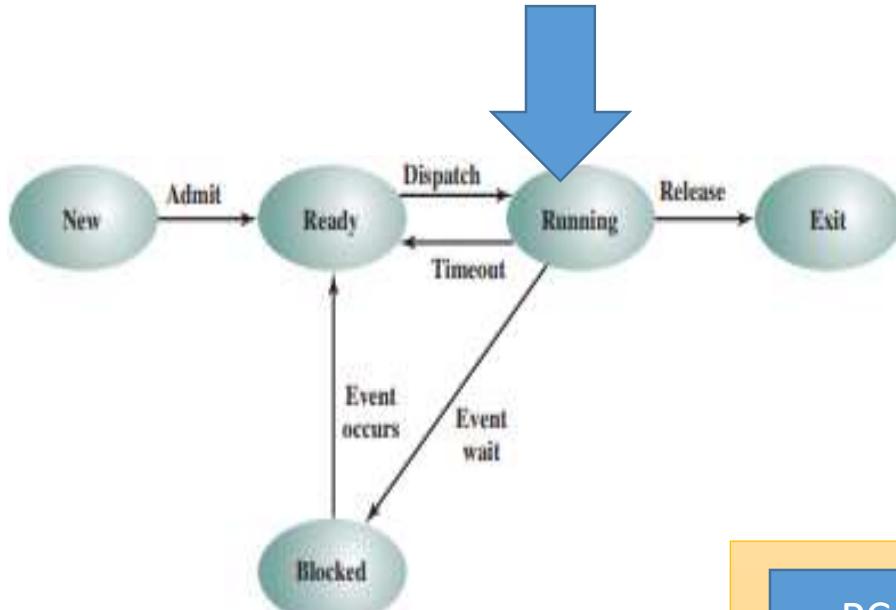


Figure 8.7 Five-State Process Model

Different States of a Process



Lets assume,
that the
scheduler
decided to run
this process
again

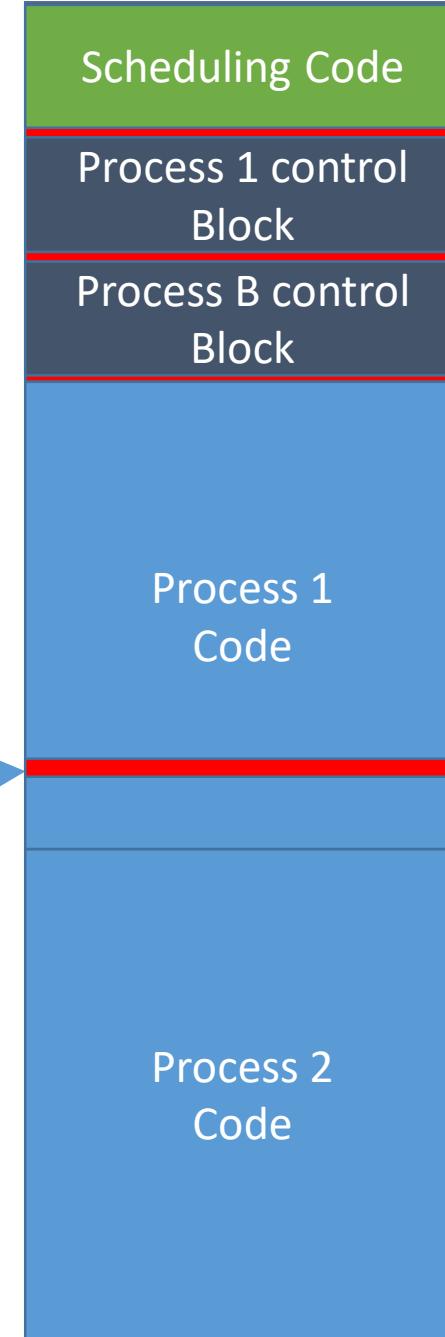


Figure 8.7 Five-State Process Model

Different States of a Process

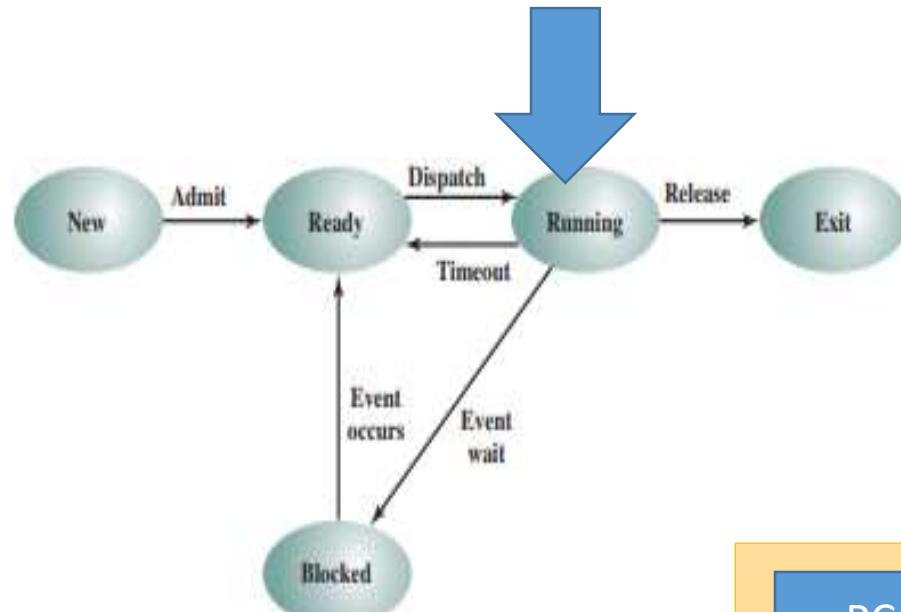
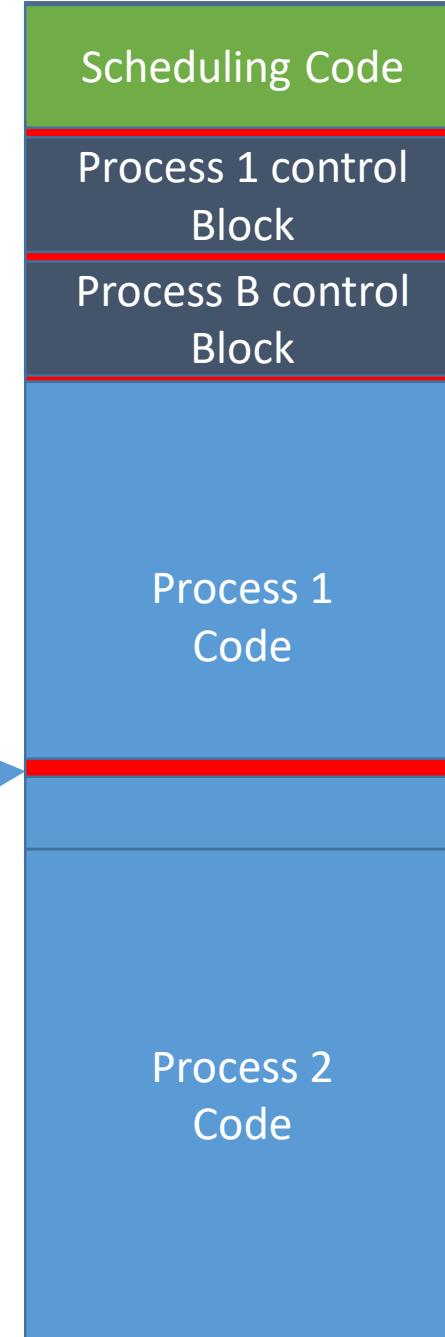
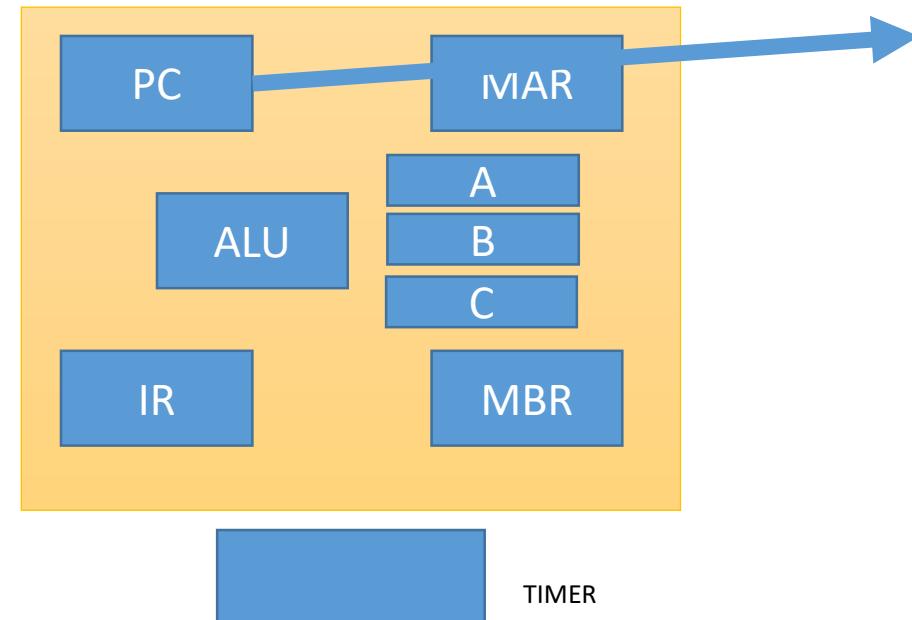


Figure 8.7 Five-State Process Model



This is a line
Of code that
Requires **action**
To be taken
By the **user**

Different States of a Process

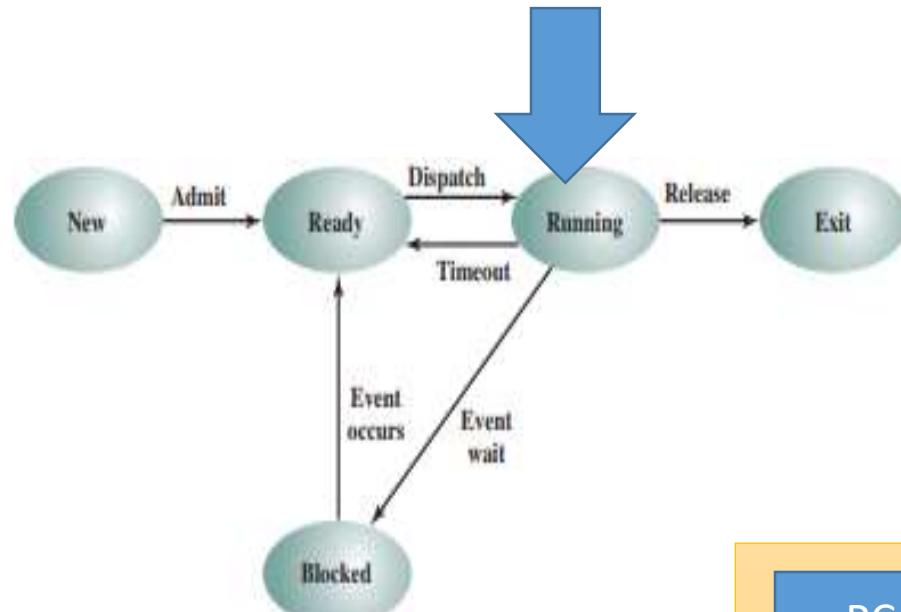
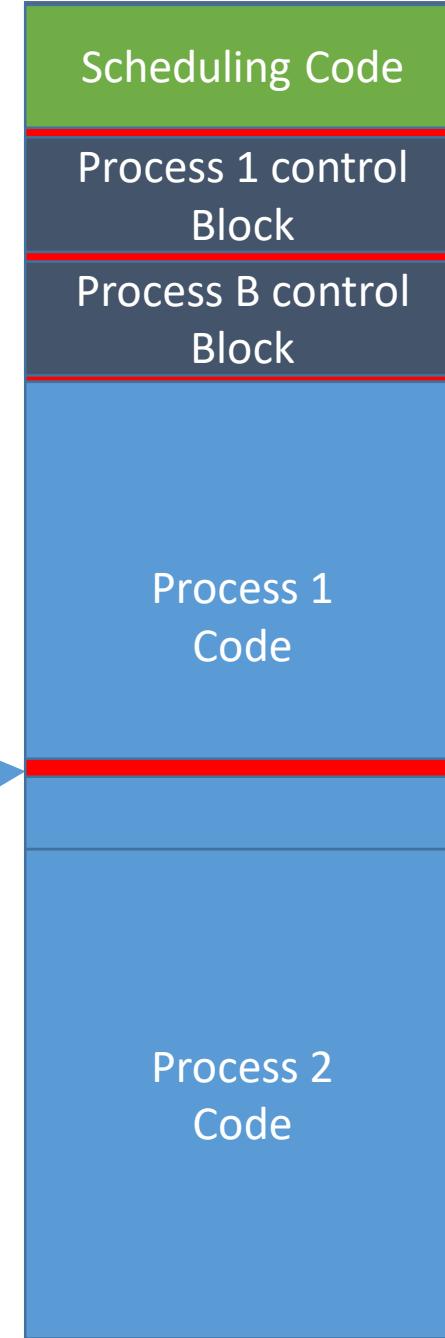
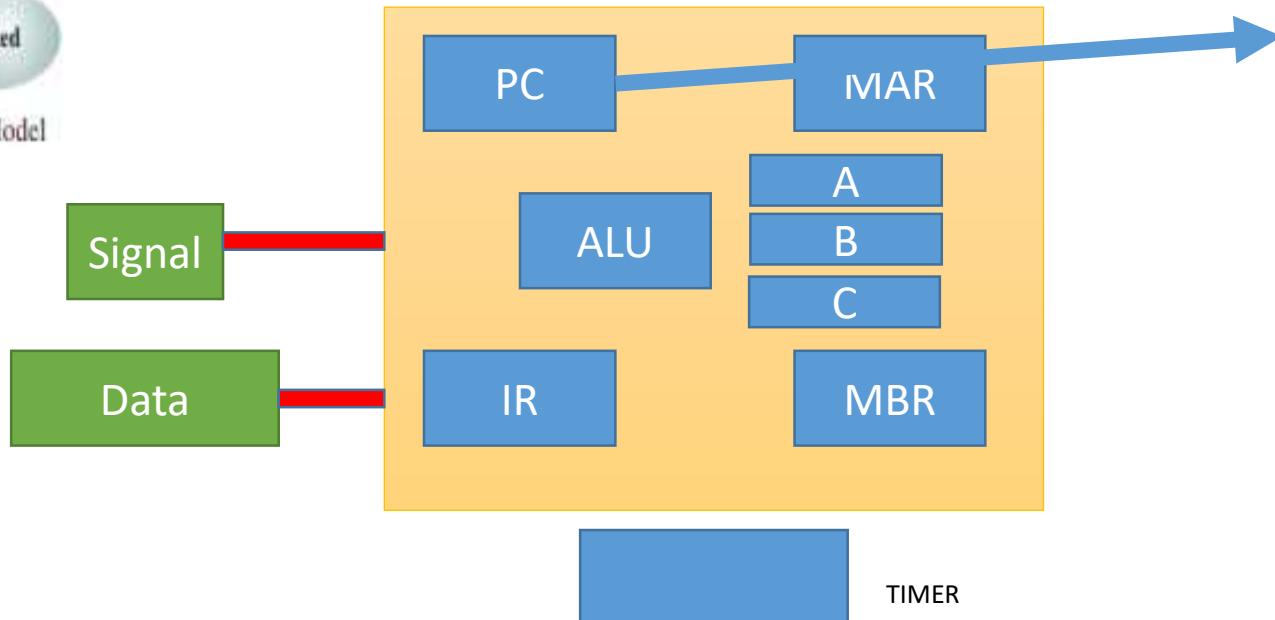


Figure 8.7 Five-State Process Model



This is a line
Of code that
Requires action
To be taken
By the user

Different States of a Process

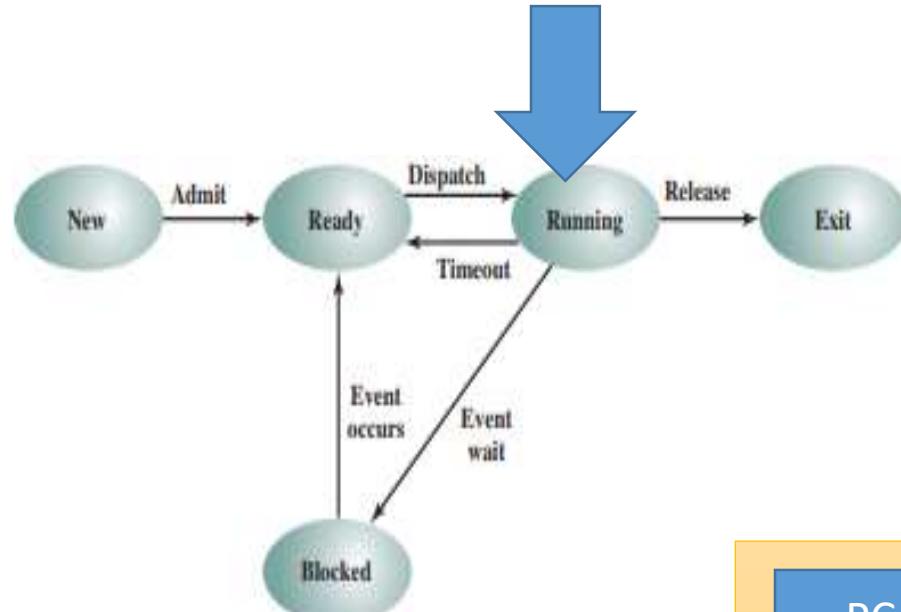
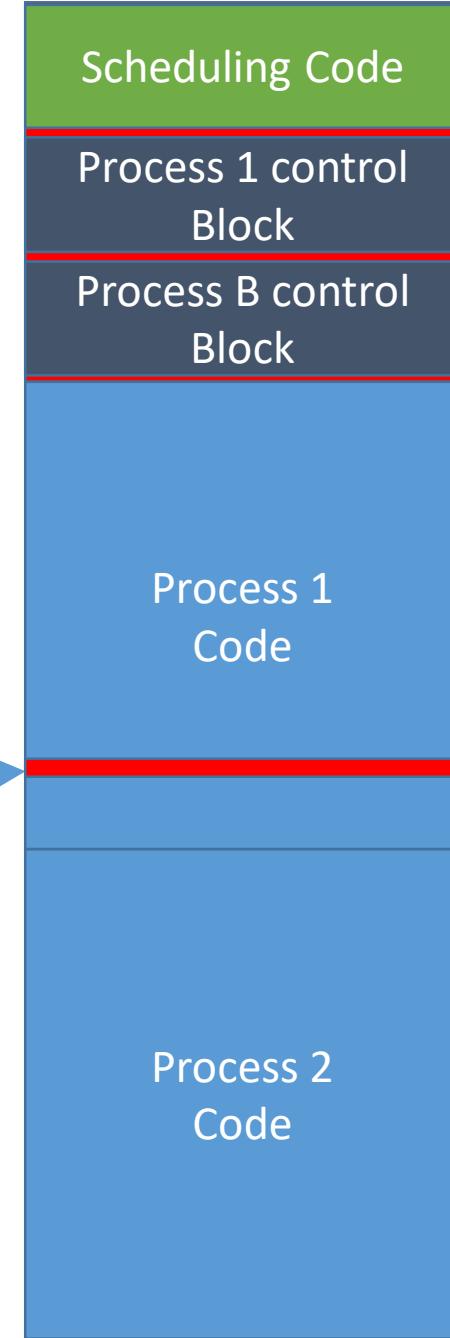
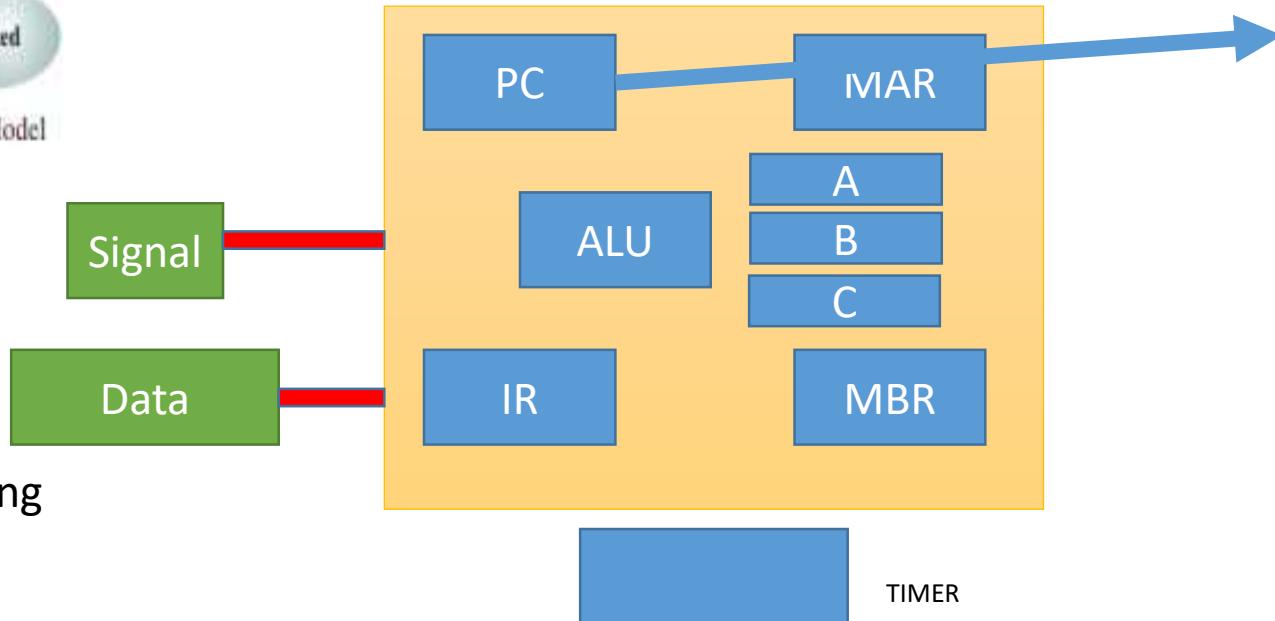


Figure 8.7 Five-State Process Model

Let us assume
That the device
Connected is a
Sensor which
Provides **data** along
With a **signal**



This is a line
Of code that
Requires **action**
To be taken
By the **user**

Which basically,
Means that the
Processor will
Have to **wait**
Before executing the
Next line.

Different States of a Process

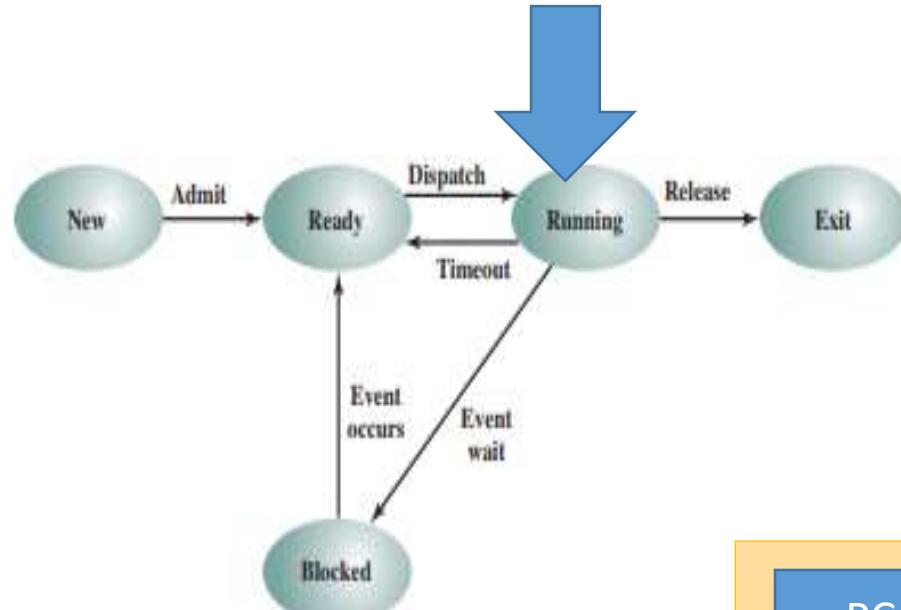
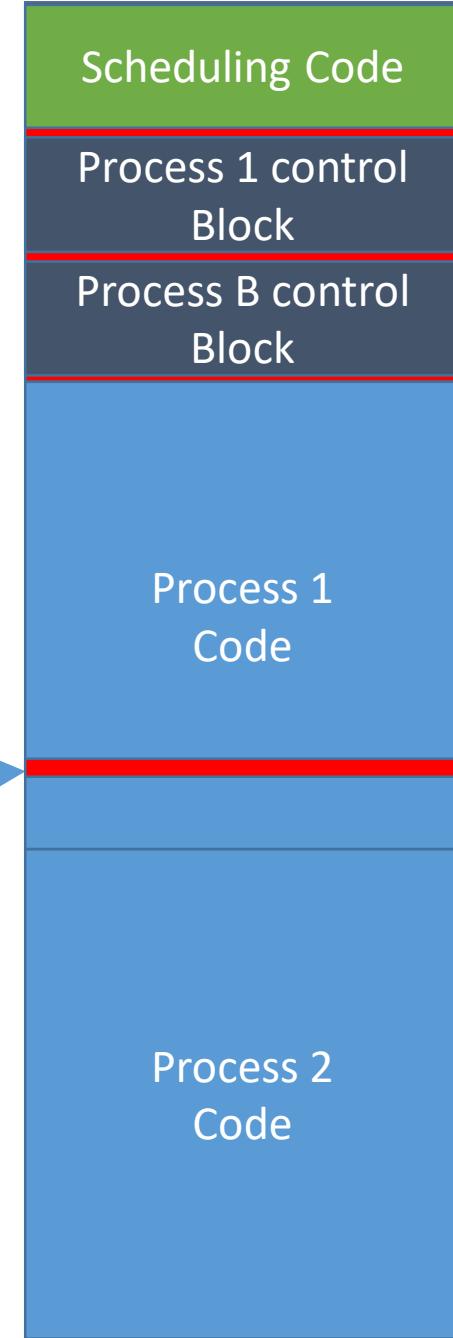
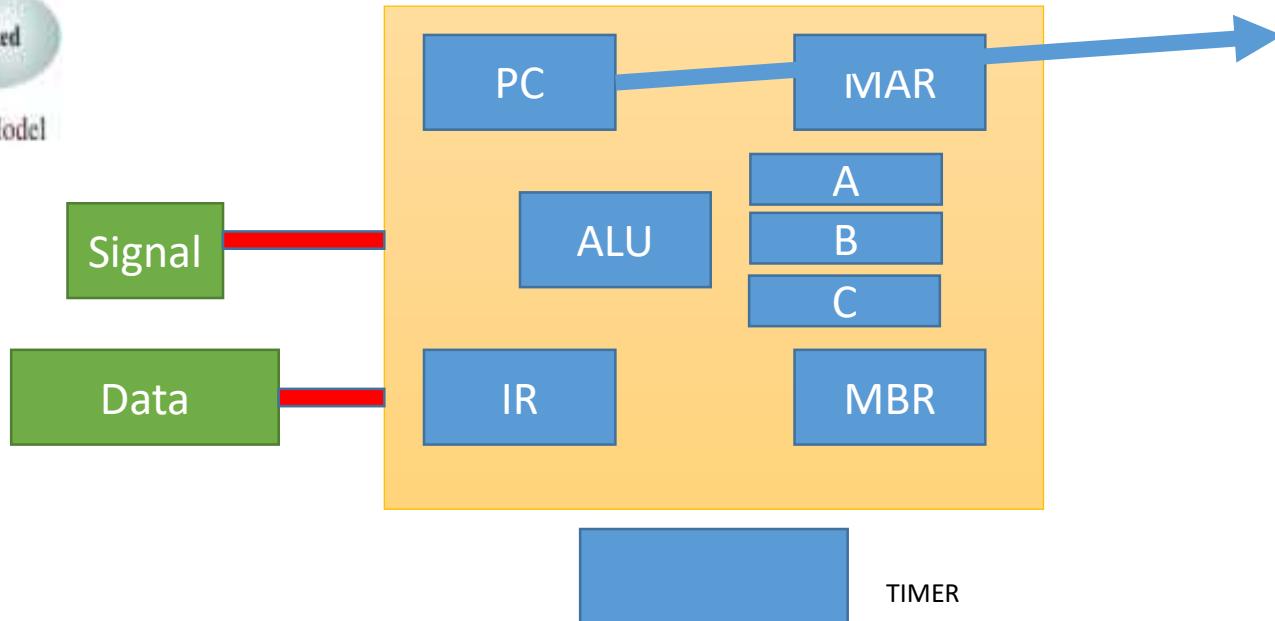


Figure 8.7 Five-State Process Model



Different States of a Process

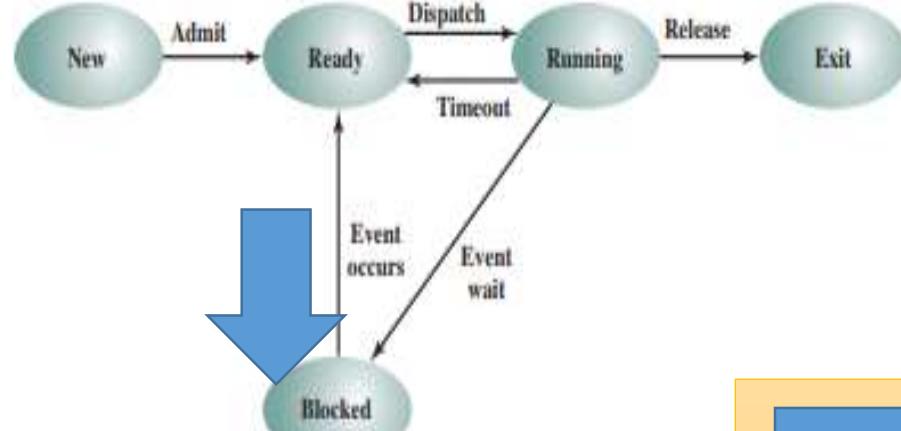
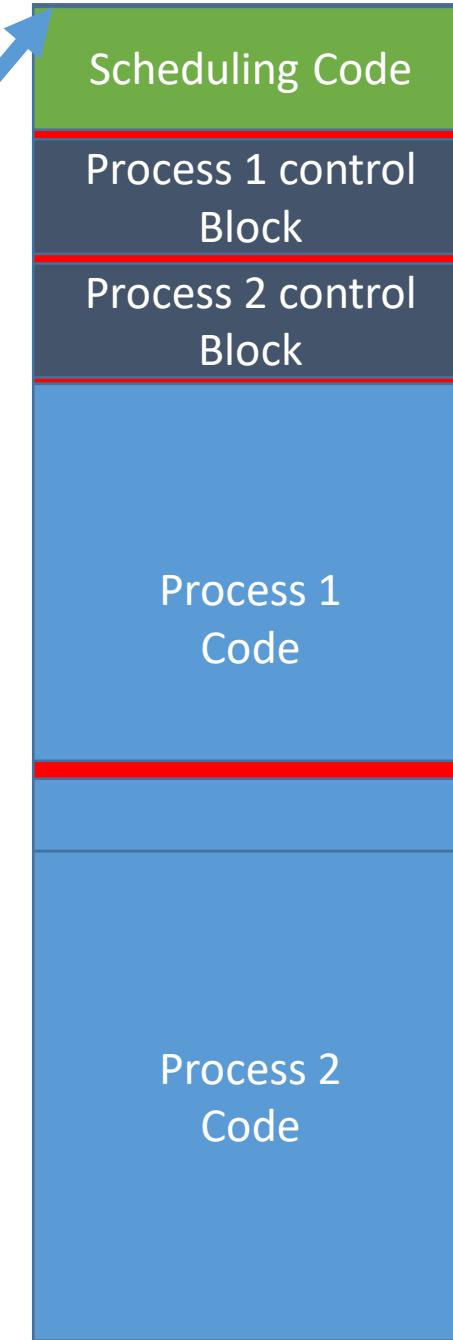
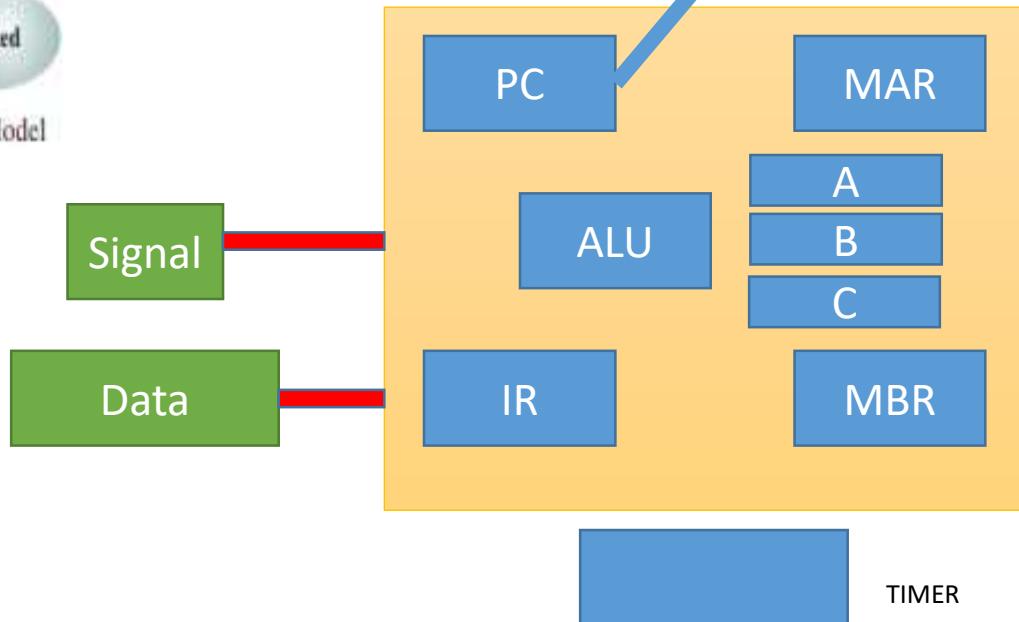


Figure 8.7 Five-State Process Model



Time runs Out for this Instruction!

Since it was Waiting for an Event, the OS Marks it as blocked

Different States of a Process

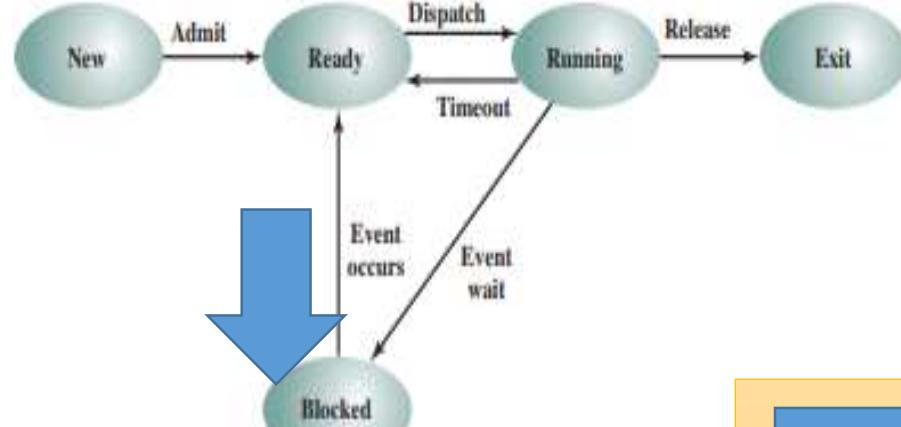
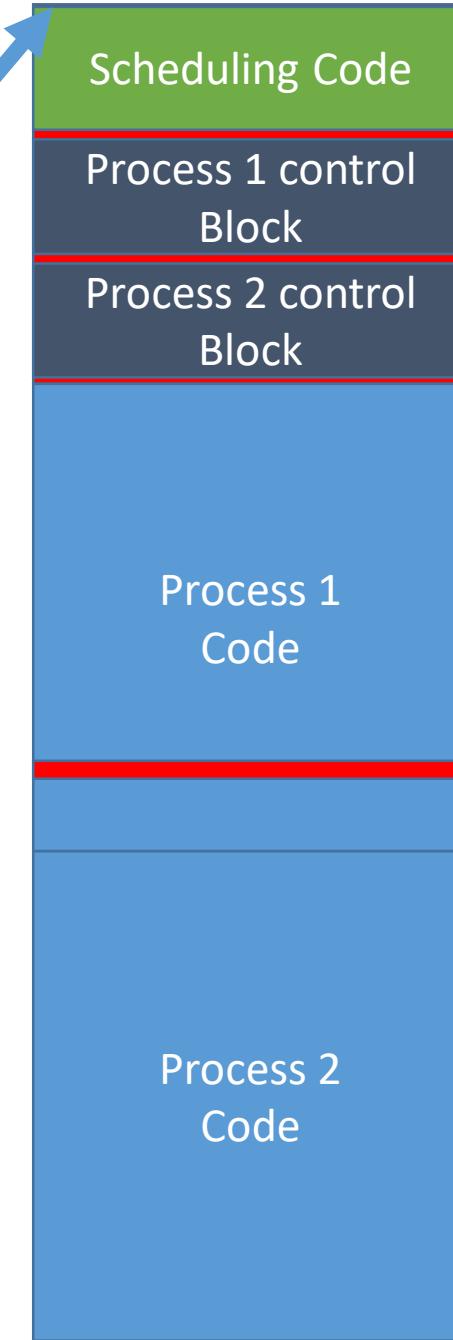
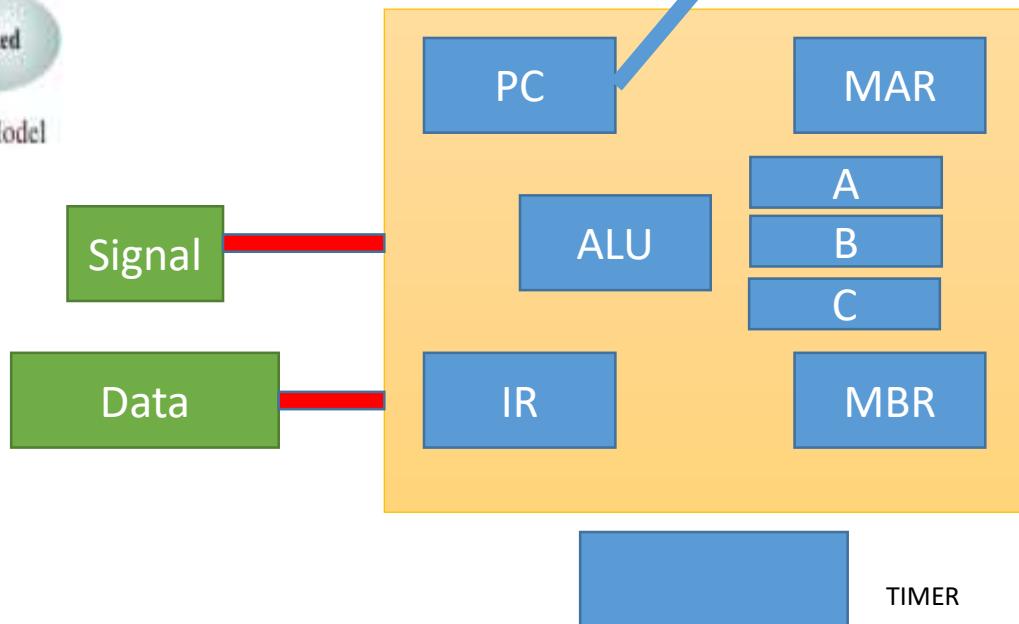


Figure 8.7 Five-State Process Model



The scheduler will Schedule process 1
Again only if it Gets the **input And the **signal****

Different States of a Process

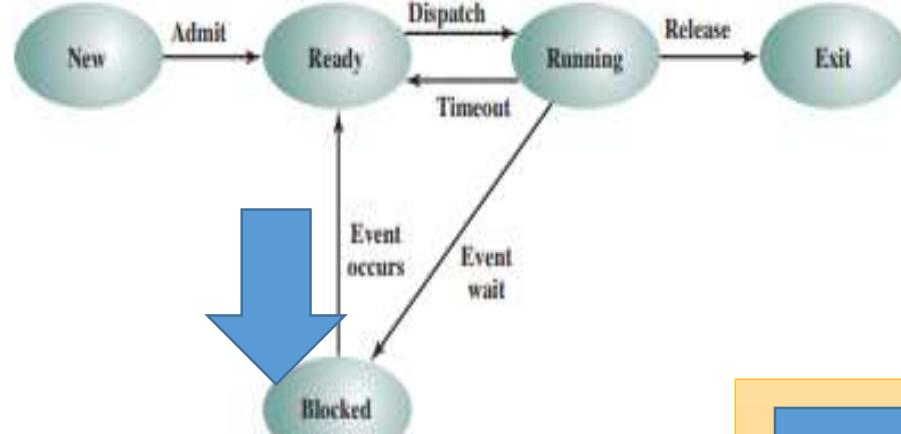
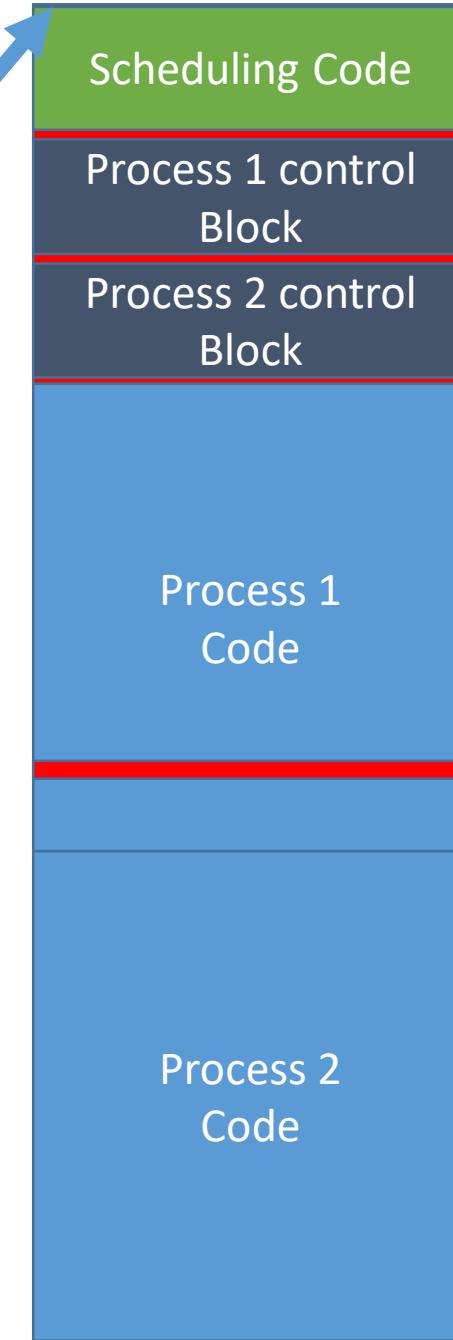
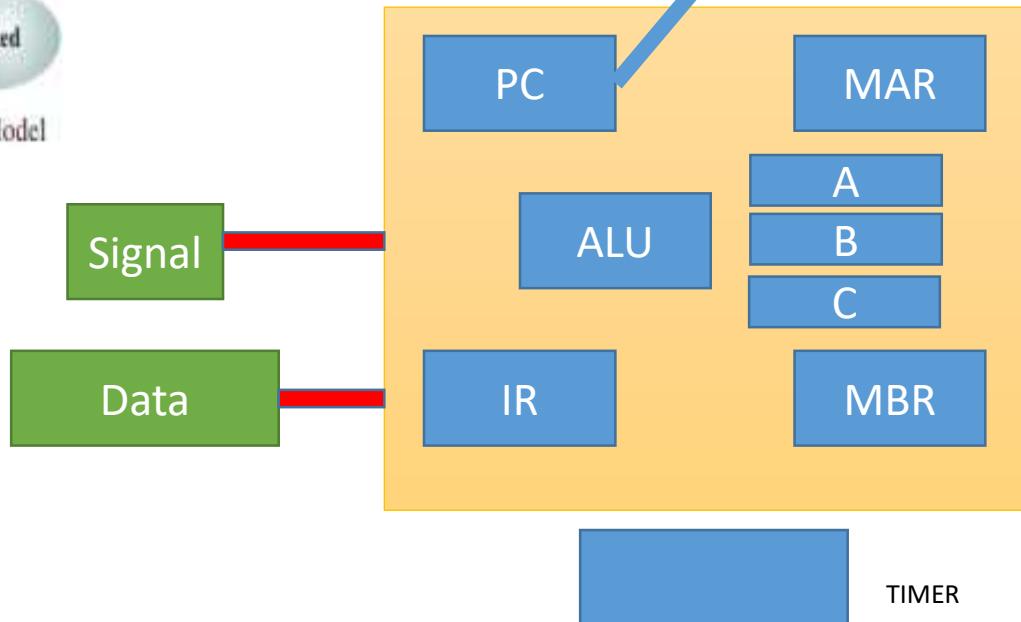


Figure 8.7 Five-State Process Model



The scheduler will Schedule process 1 Again only if it Gets the input And the signal

Every time before Scheduling a process, It will **check whether**, The **blocked process** Events occurred or not

Different States of a Process

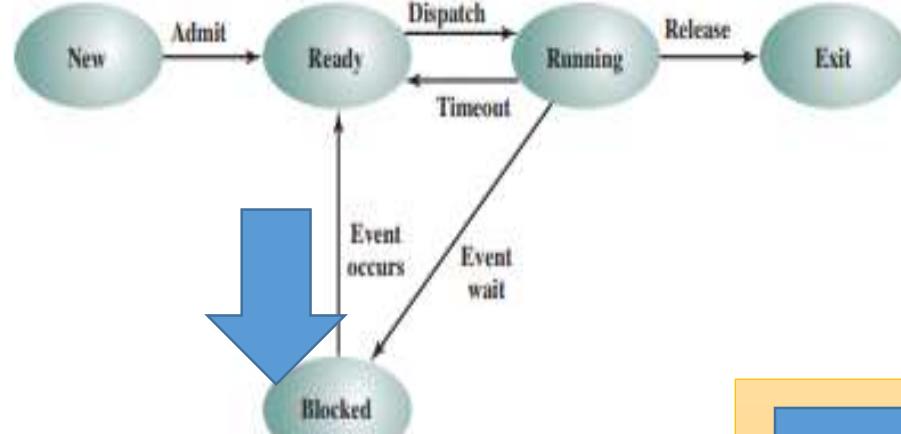
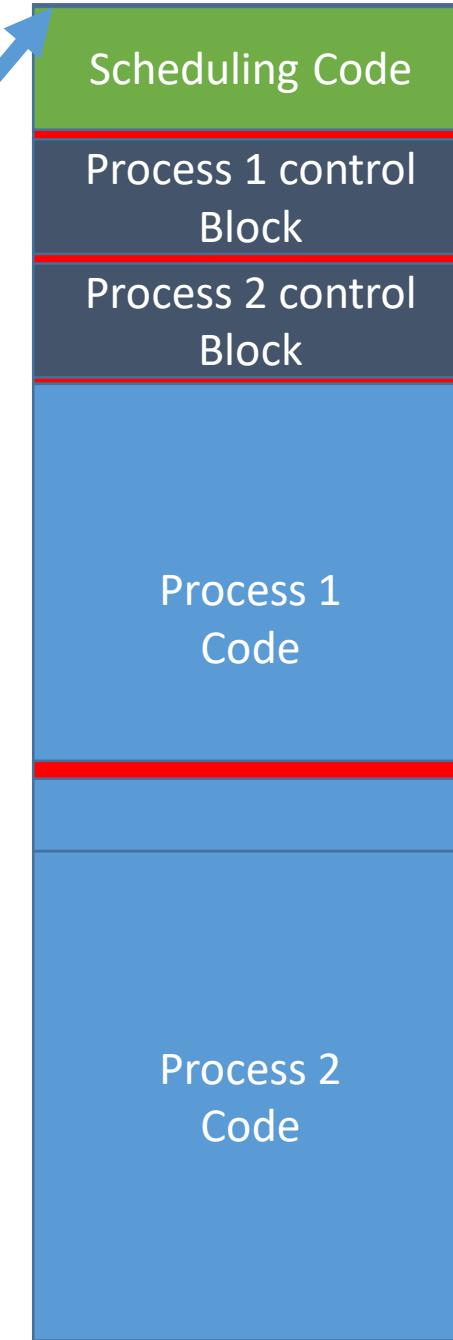
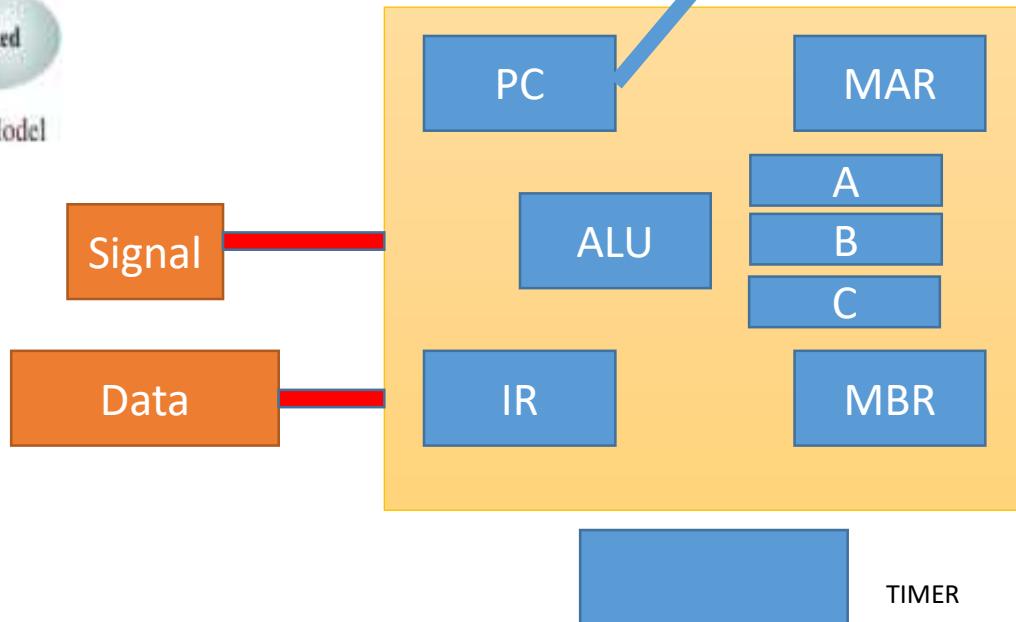


Figure 8.7 Five-State Process Model

If the **particular**
Event occurs



The scheduler will
Schedule process 1
Again only if it
Gets the input
And the signal

Every time
before
Scheduling a
process,
It will check
whether,
The blocked
process
Events occurred
or not

Different States of a Process

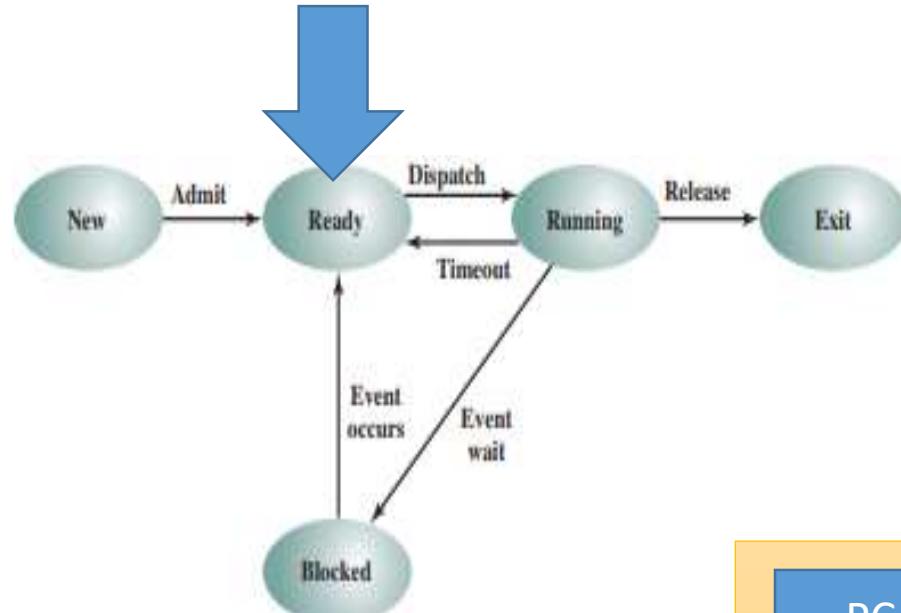
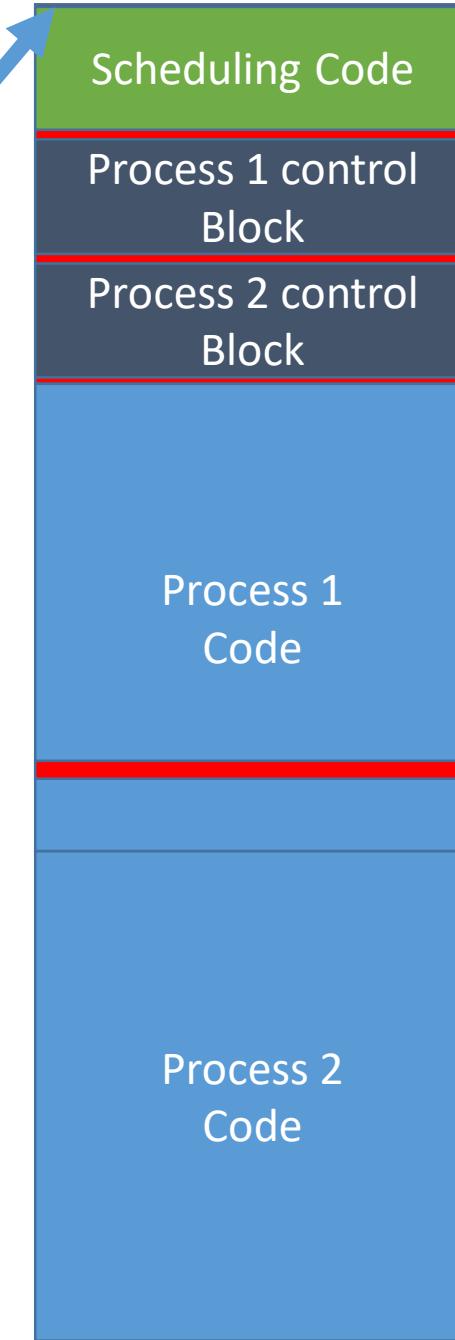
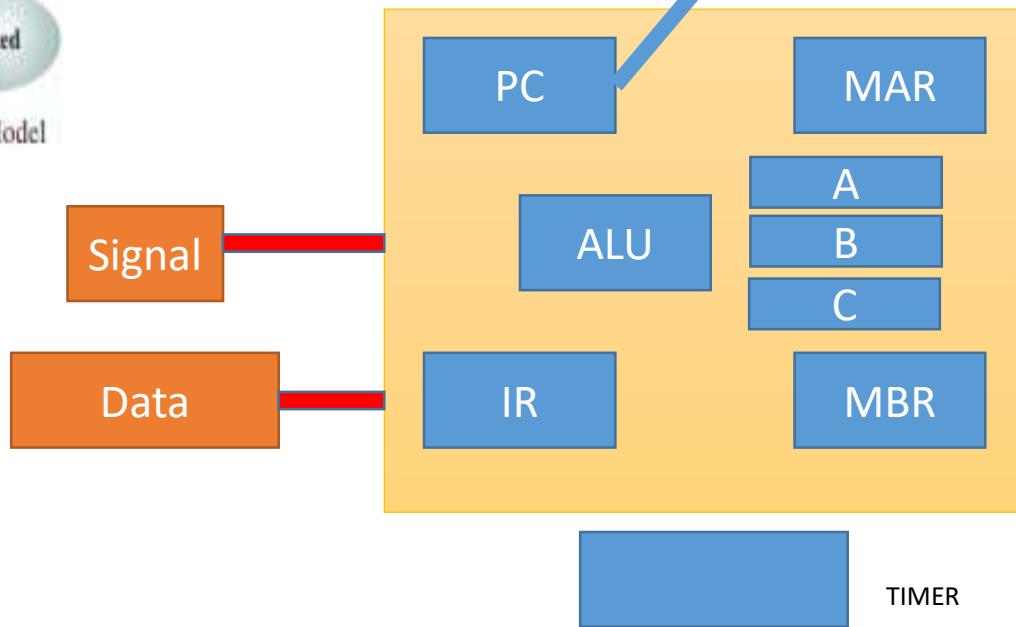


Figure 8.7 Five-State Process Model

If the particular Event occurs
It will be marked As ready again



The scheduler will Schedule process 1 Again only if it Gets the input And the signal

Every time before Scheduling a process, It will check whether, The blocked process Events occurred or not

Different States of a Process

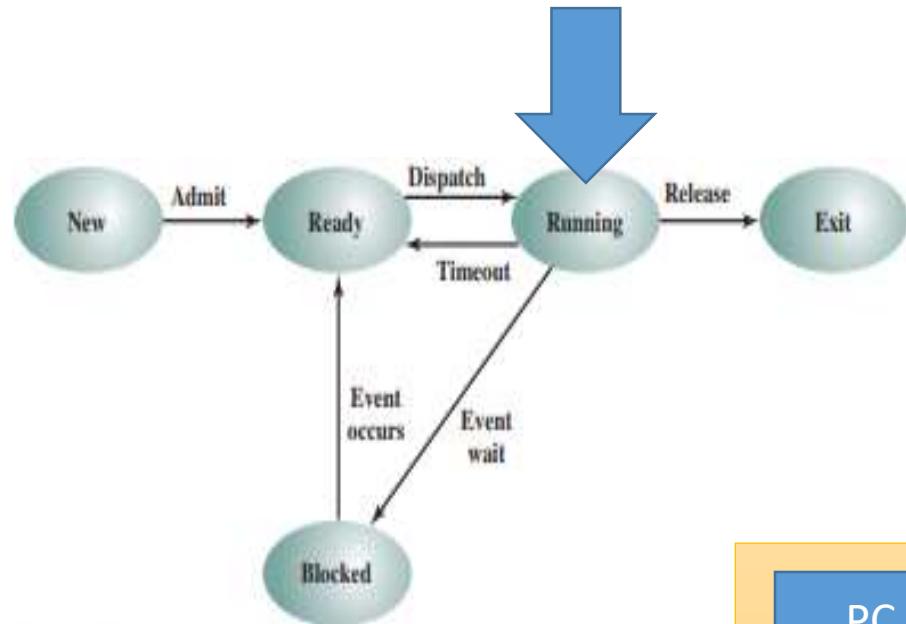
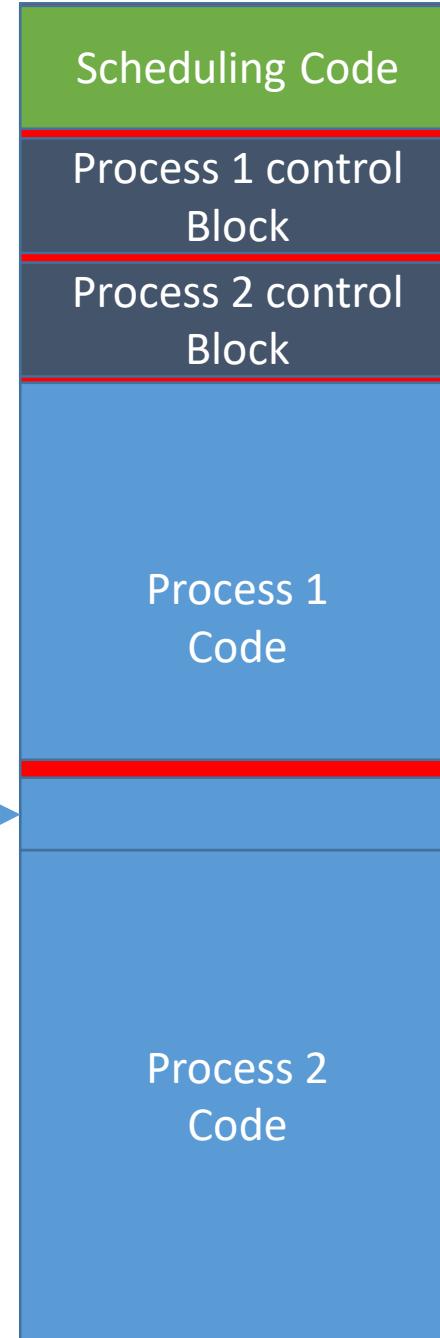
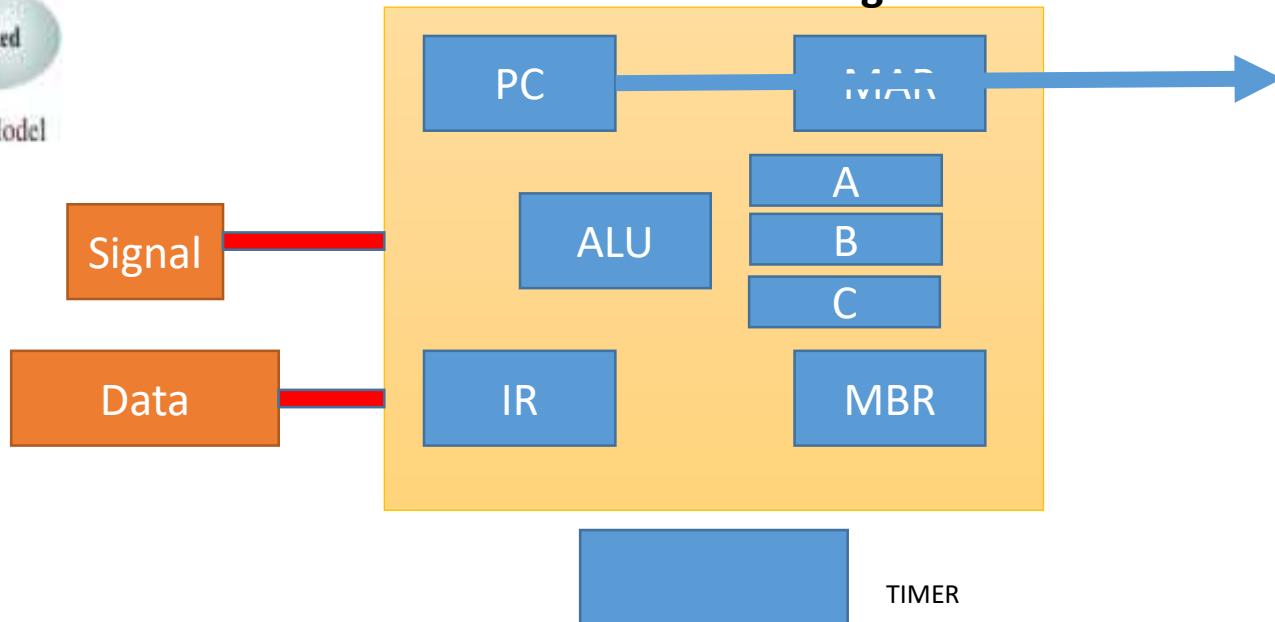


Figure 8.7 Five-State Process Model

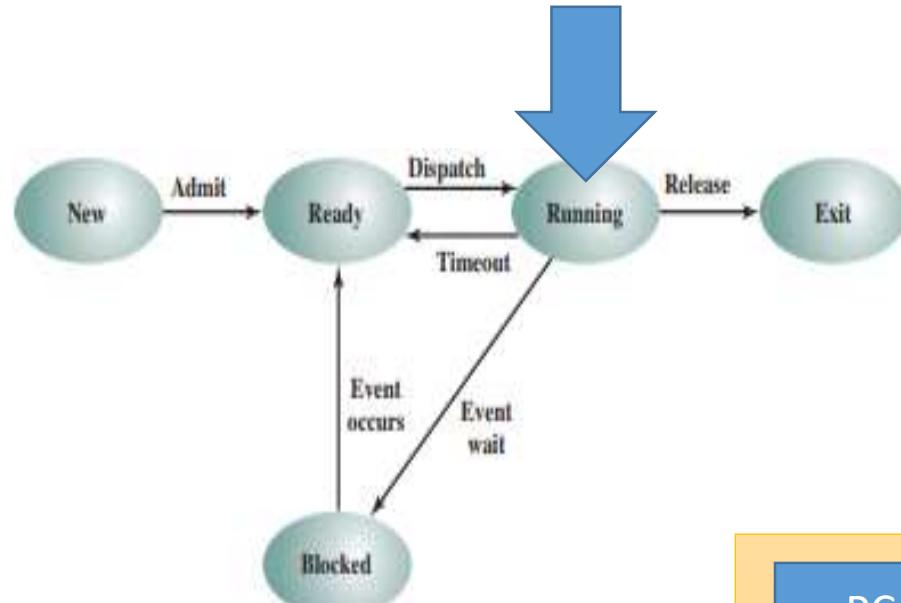
If the particular Event occurs
It will be marked As ready again



The scheduler will Schedule process 1 Again only if it Gets the input And the signal

Every time before Scheduling a process, It will check whether, The blocked process Events occurred or not

Different States of a Process



Once its **final Line of code**
Has been
executed

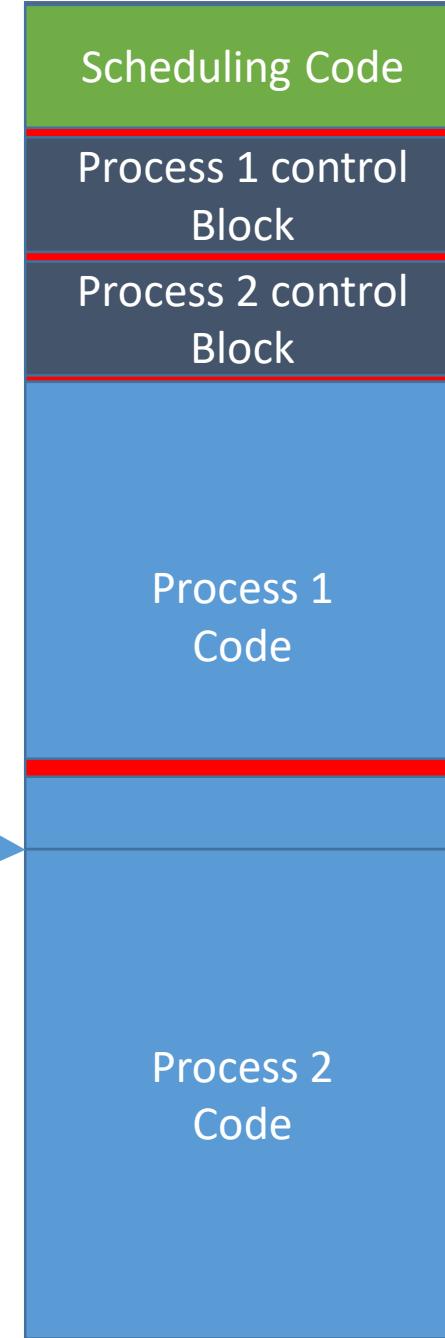
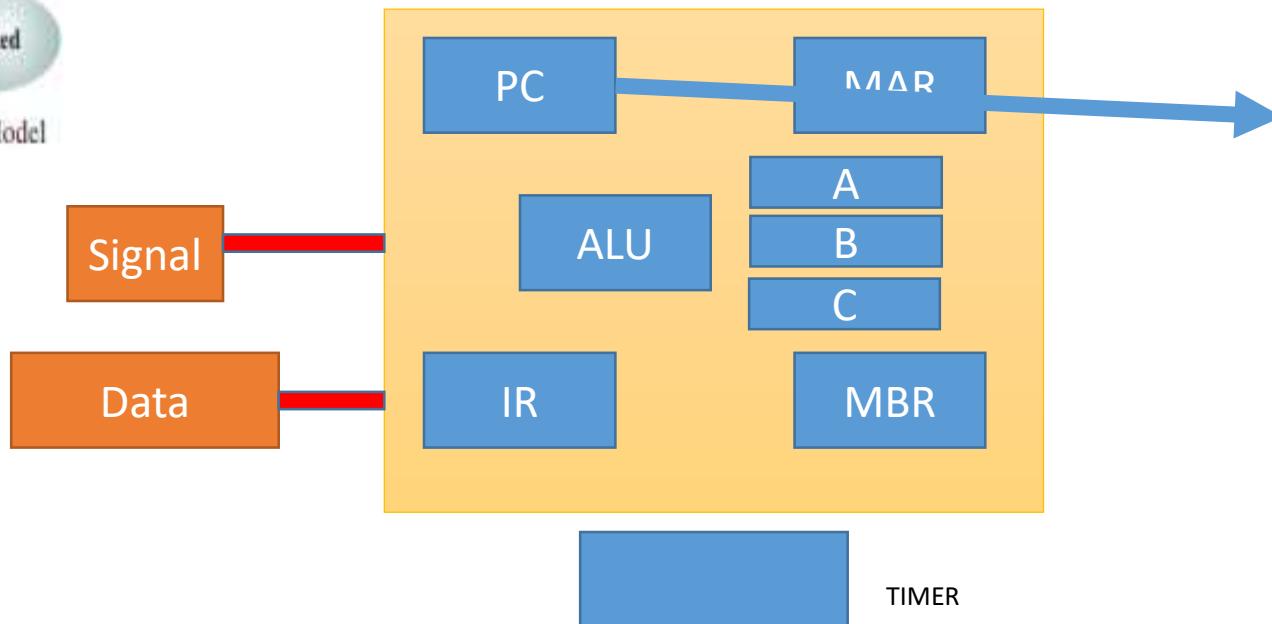
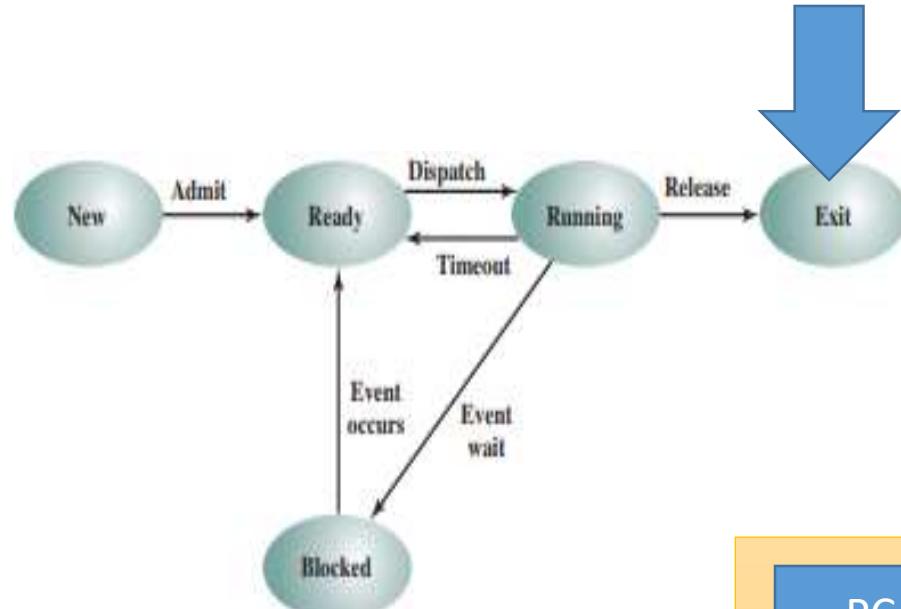
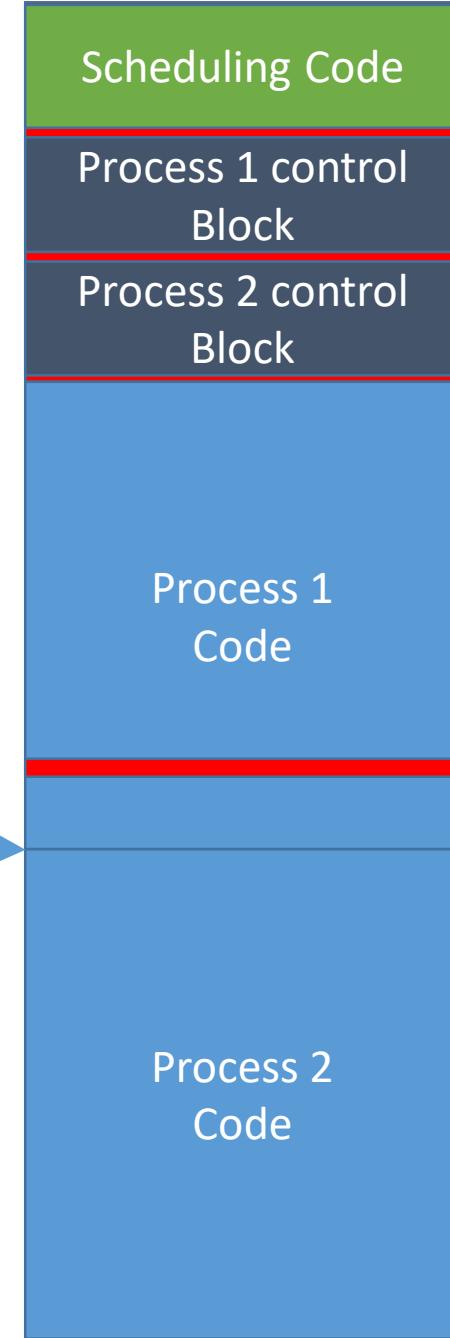
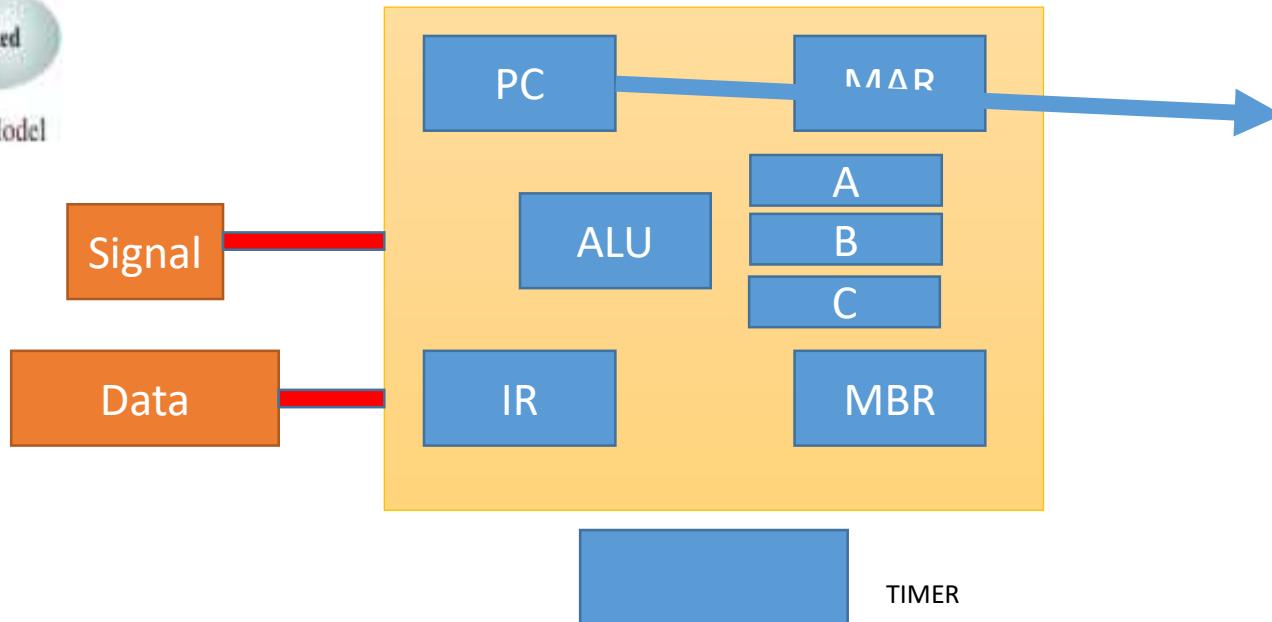


Figure 8.7 Five-State Process Model

Different States of a Process



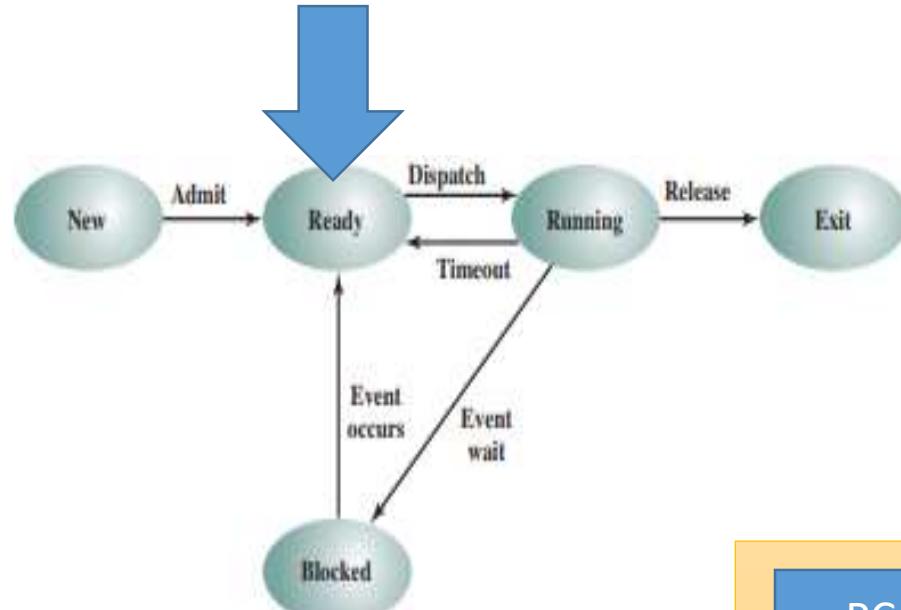
Once its final
Line of code
Has been
executed



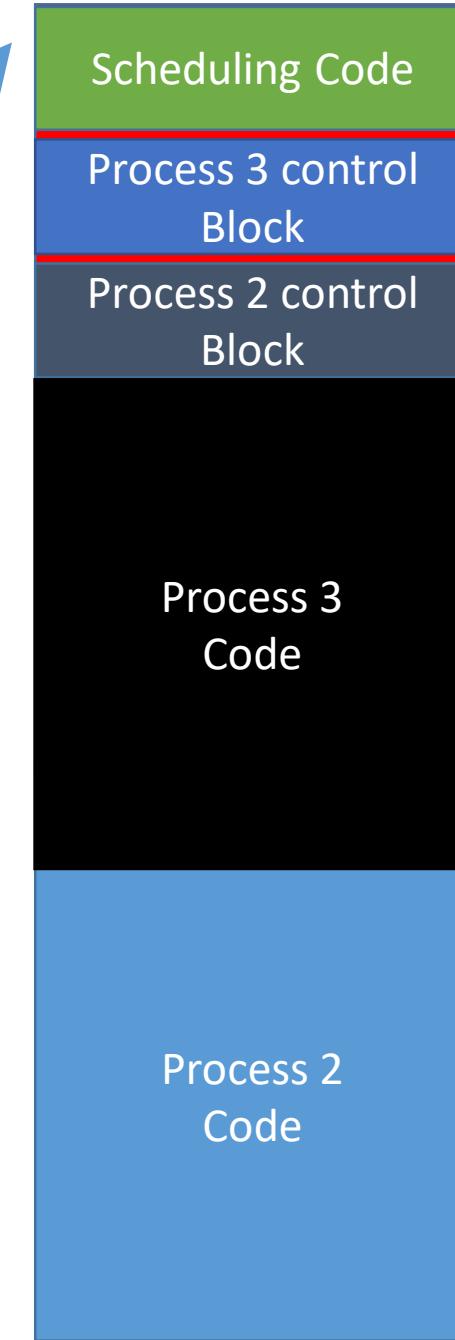
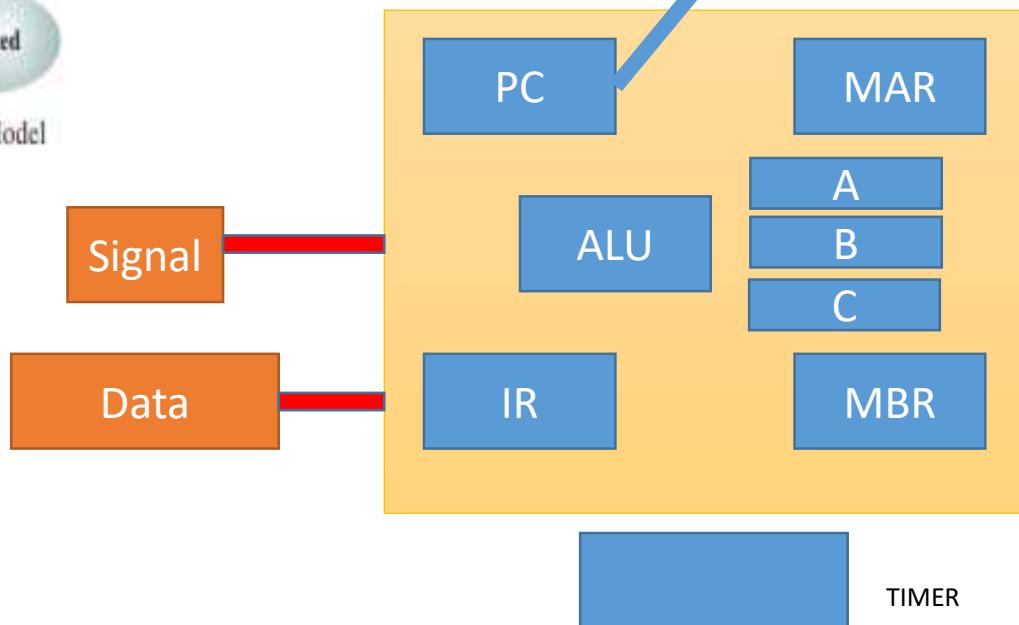
The memory
having the code
and PCB can be
declared as **free
memory**, and
can
Be **overwritten**
by **other
processes**.

Figure 8.7 Five-State Process Model

Different States of a Process



Once its final
Line of code
Has been
executed



The memory having the code and PCB can be declared as free memory, and can Be overwritten by other processes.

This state of the process is known as **Exit state**.

Different Types of Scheduling

Long Term Scheduling

- Basically determining which instructions will be brought from the New state to the Ready state.
- i.e. from the Storage (Hard Disc) to the Memory

Short Term Scheduling

- Determines which process will be getting the processor
- i.e. which process will go from ready state to running state

3.1 Memory Management and its necessities

Memory Management

- If multiple programs were not being run simultaneously.
 - And there was a lot of memory space.
 - We could reserve just one part of the memory for OS and the other part for the processes.
-
- But multiple processes, finite memory -> efficient memory management required.
 - A situation may occur where all the processes are blocked

Memory Management

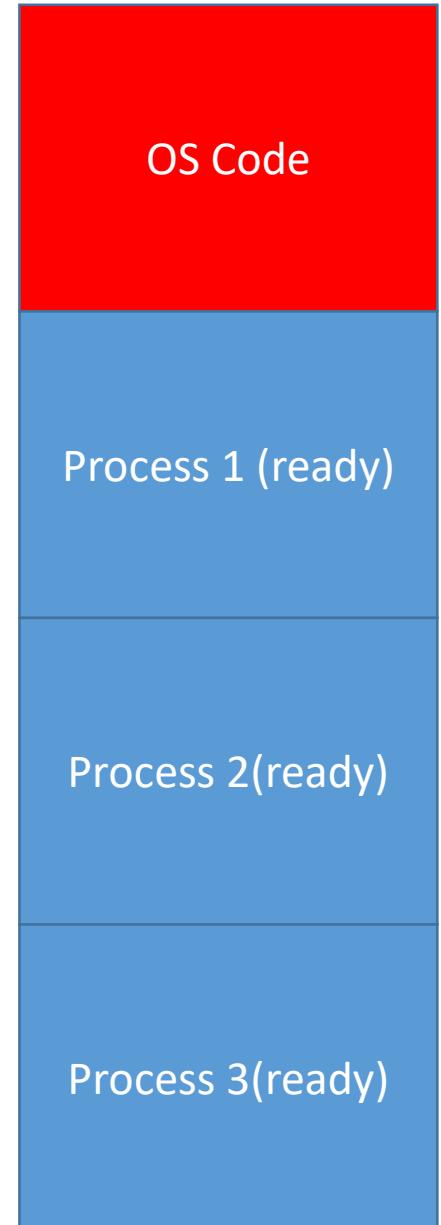
- If **multiple** programs were **not** being run simultaneously.
 - And there was a **lot of memory space**.
 - We could reserve just one part of the memory for **OS** and the other part for the **processes**.
-
- But multiple processes, finite memory -> efficient memory management required.
 - A situation may occur where all the processes are blocked

Memory Management

- If multiple programs were not being run simultaneously.
 - And there was a lot of memory space.
 - We could reserve just one part of the memory for OS and the other part for the processes.
-
- But multiple processes, **finite memory** -> **efficient memory management required**.
 - A situation may occur where all the **processes are blocked**

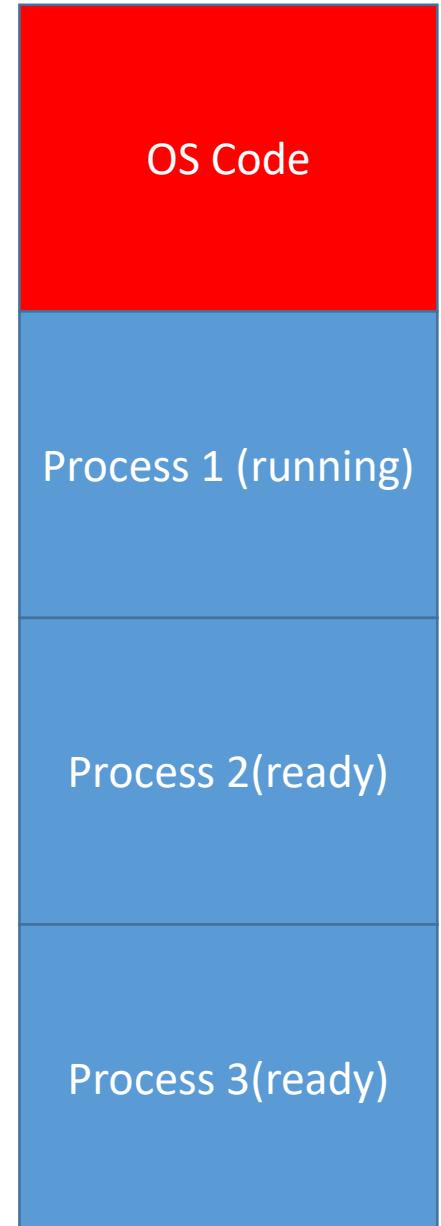
Memory Management

- If multiple programs were not being run simultaneously.
 - And there was a lot of memory space.
 - We could reserve just one half of the memory for OS and the other half for the process.
-
- But multiple processes, finite memory -> efficient memory management required.
 - A situation may occur where all the processes are blocked



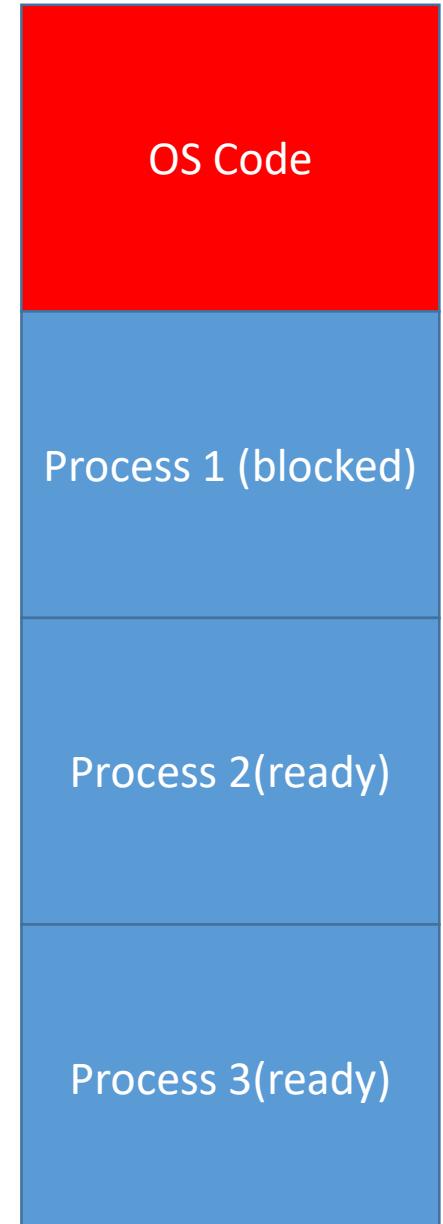
Memory Management

- If multiple programs were not being run simultaneously.
 - And there was a lot of memory space.
 - We could reserve just one half of the memory for OS and the other half for the process.
-
- But multiple processes, finite memory -> efficient memory management required.
 - A situation may occur where all the processes are blocked



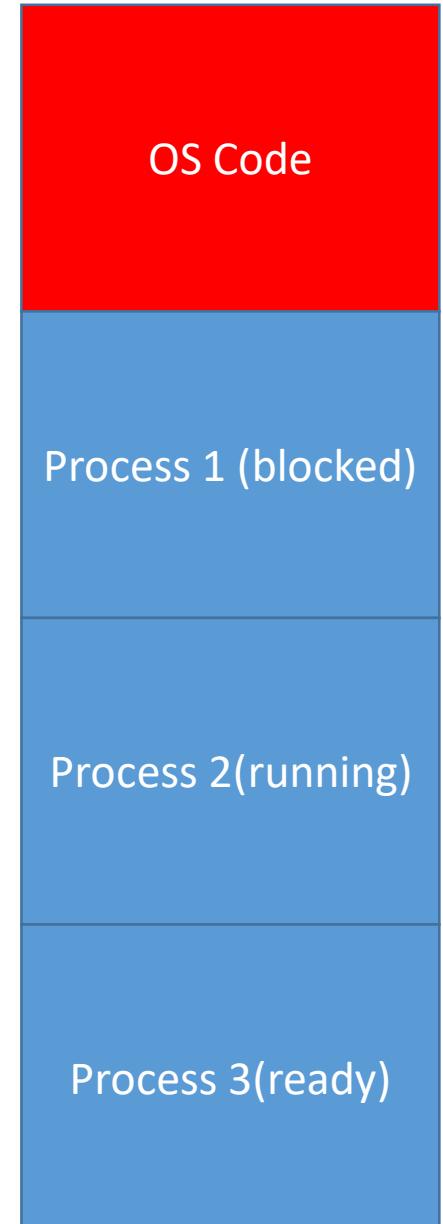
Memory Management

- If multiple programs were not being run simultaneously.
 - And there was a lot of memory space.
 - We could reserve just one half of the memory for OS and the other half for the process.
-
- But multiple processes, finite memory -> efficient memory management required.
 - A situation may occur where all the processes are blocked



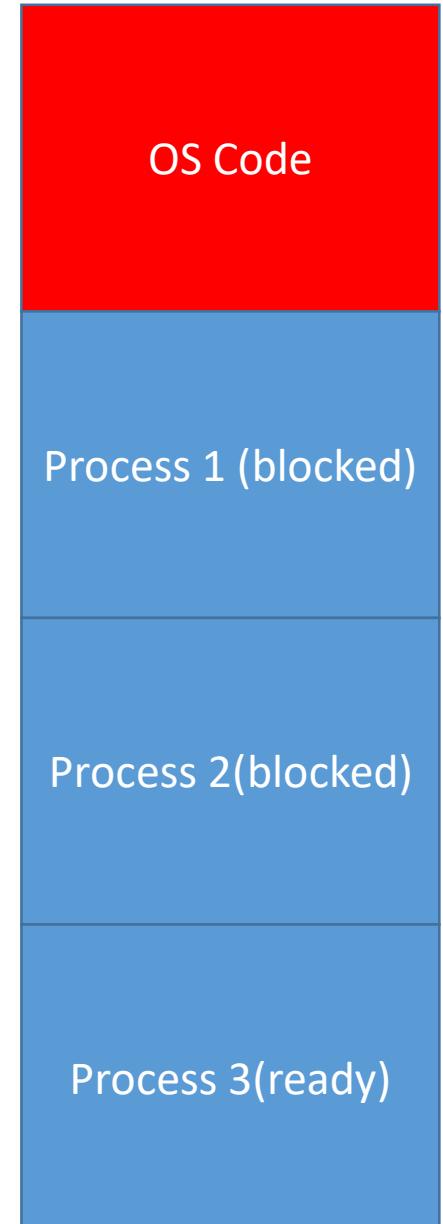
Memory Management

- If multiple programs were not being run simultaneously.
 - And there was a lot of memory space.
 - We could reserve just one half of the memory for OS and the other half for the process.
-
- But multiple processes, finite memory -> efficient memory management required.
 - A situation may occur where all the processes are blocked



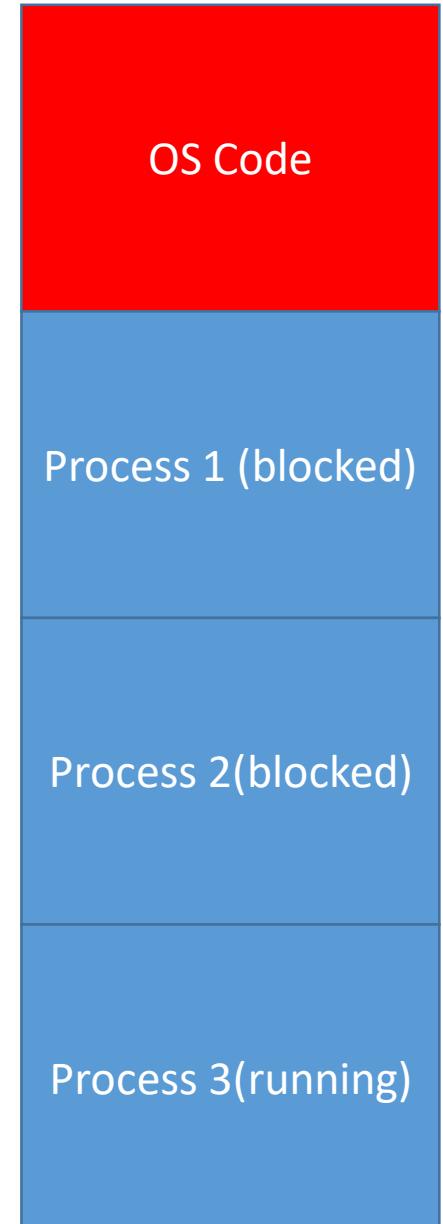
Memory Management

- If multiple programs were not being run simultaneously.
 - And there was a lot of memory space.
 - We could reserve just one half of the memory for OS and the other half for the process.
-
- But multiple processes, finite memory -> efficient memory management required.
 - A situation may occur where all the processes are blocked



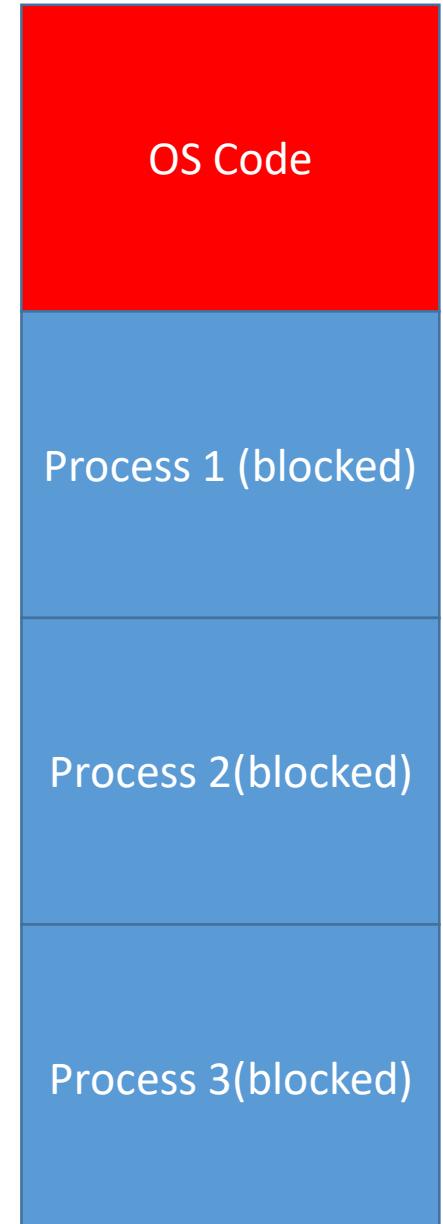
Memory Management

- If multiple programs were not being run simultaneously.
 - And there was a lot of memory space.
 - We could reserve just one half of the memory for OS and the other half for the process.
-
- But multiple processes, finite memory -> efficient memory management required.
 - A situation may occur where all the processes are blocked



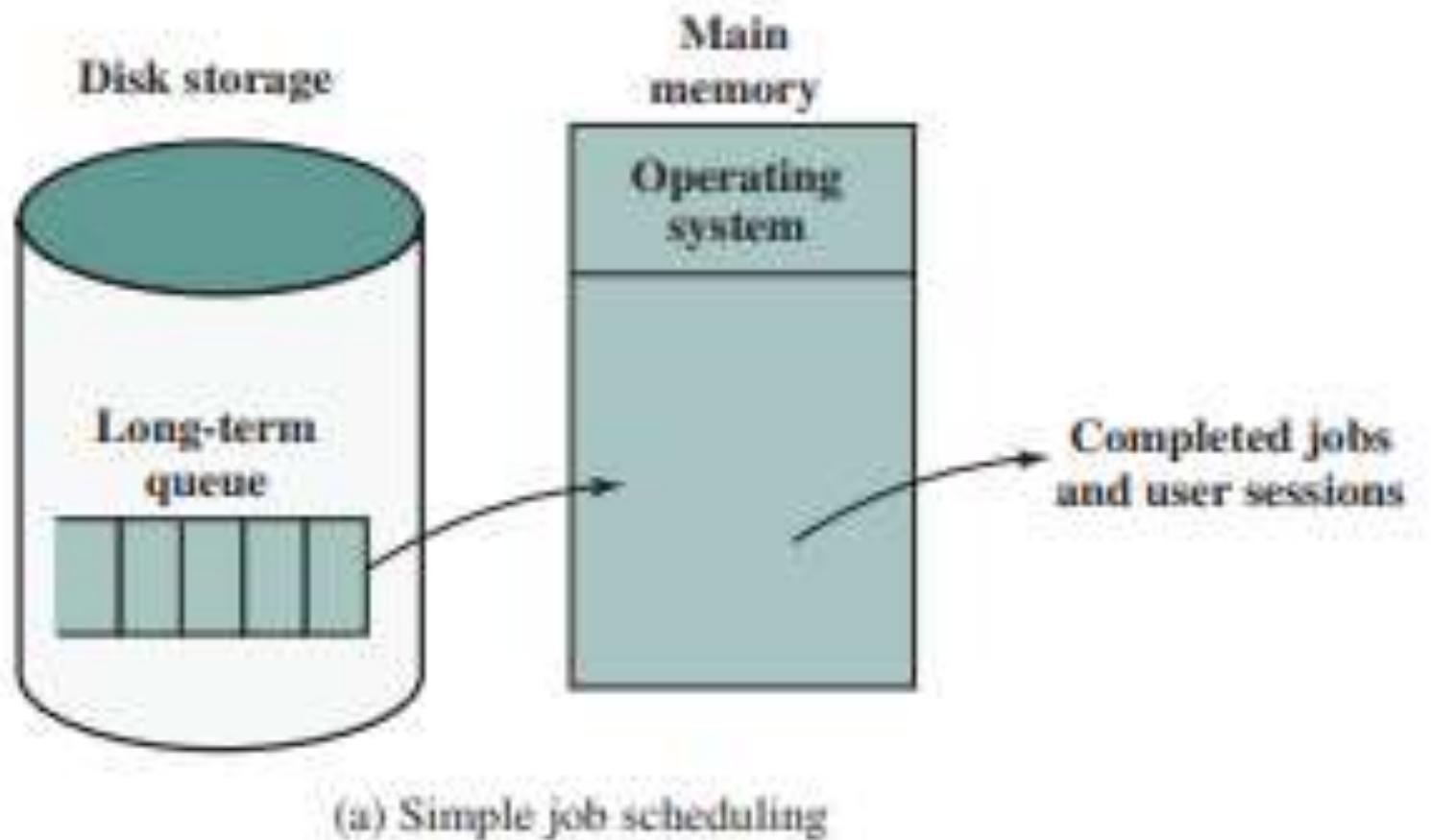
Memory Management

- If multiple programs were not being run simultaneously.
 - And there was a lot of memory space.
 - We could reserve just one half of the memory for OS and the other half for the process.
-
- But multiple processes, finite memory -> efficient memory management required.
 - A situation may occur where all the processes are blocked



3.2 Swapping

Solution: Swapping



WE ARE AQUAINTED
WITH THIS
SCENARIO

Solution: Swapping (Cont.)

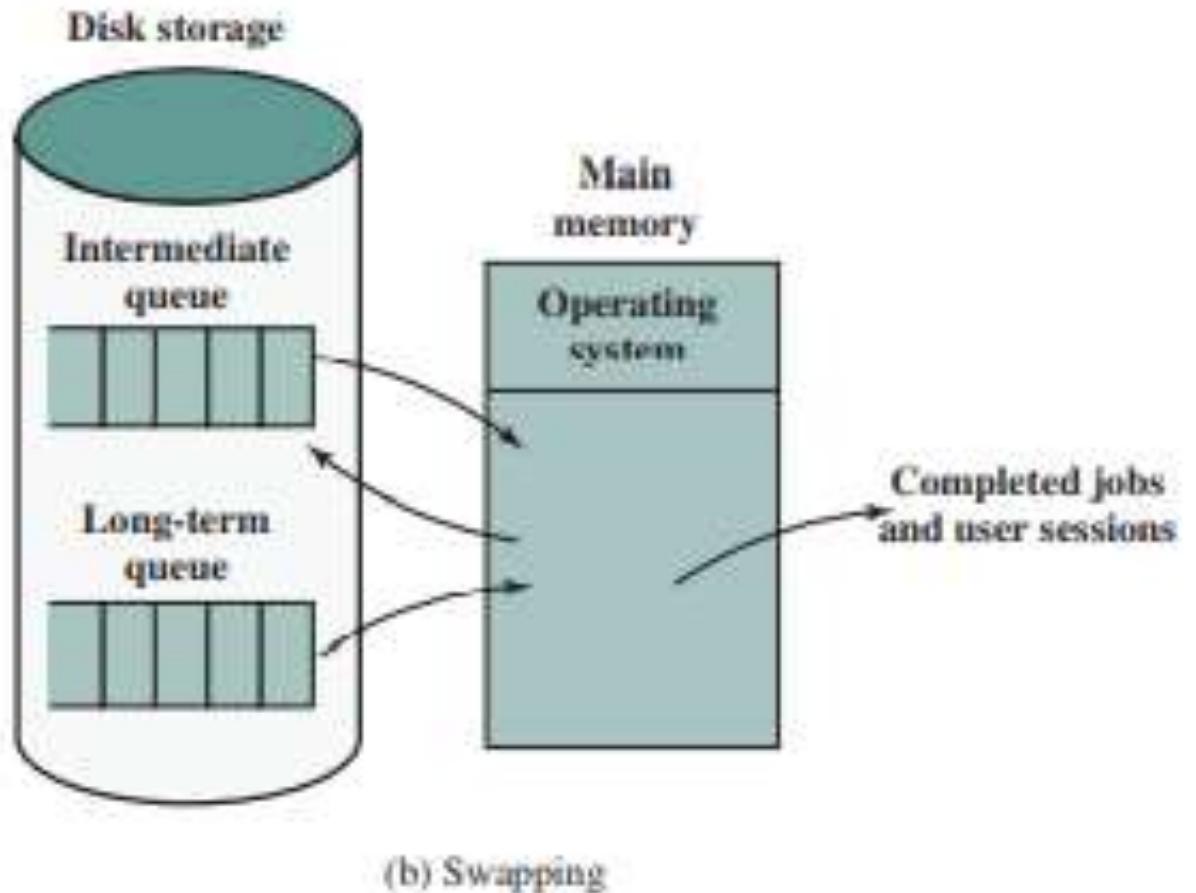
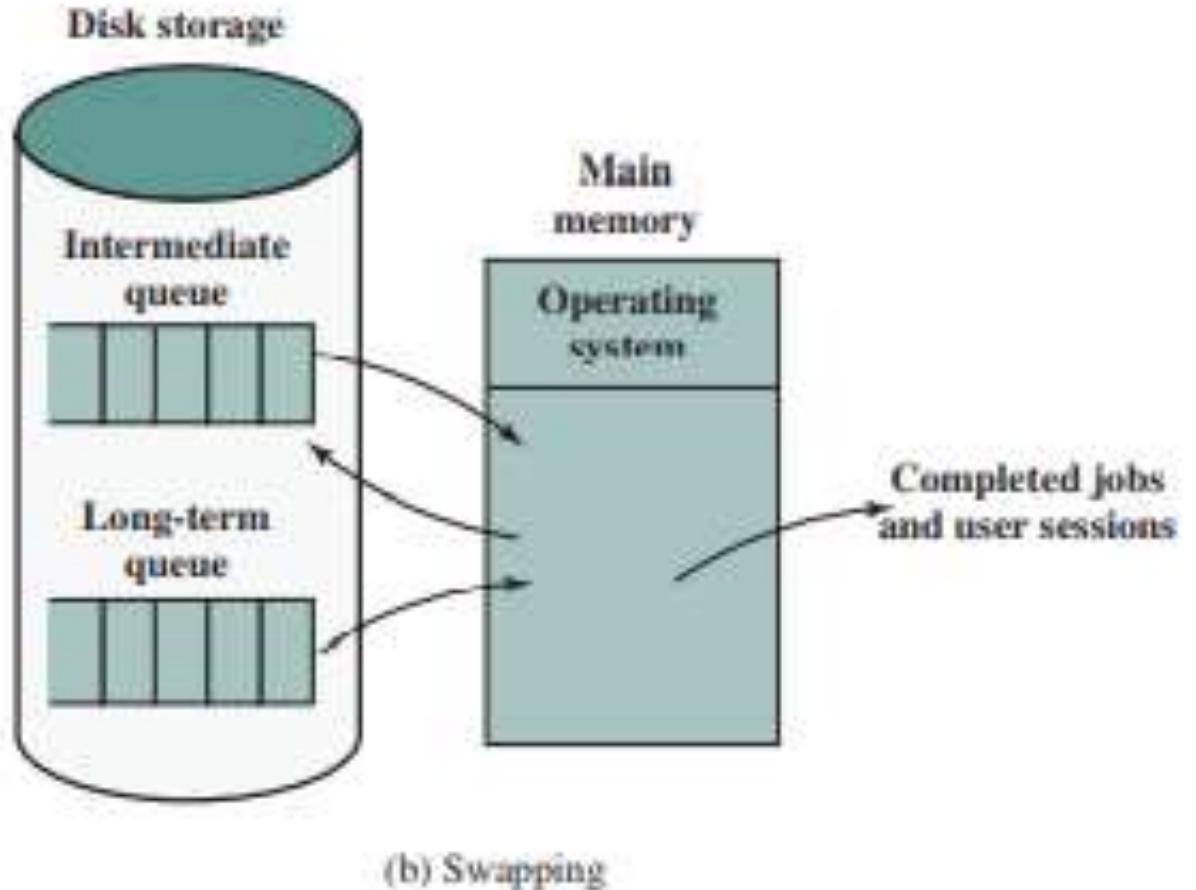


Figure 8.12 The Use of Swapping



Solution: Swapping (Cont.)



This process is **Occupying space**. And will be **idle** for **A long time** compared To the non-blocked Processes.



Figure 8.12 The Use of Swapping

Solution: Swapping (Cont.)

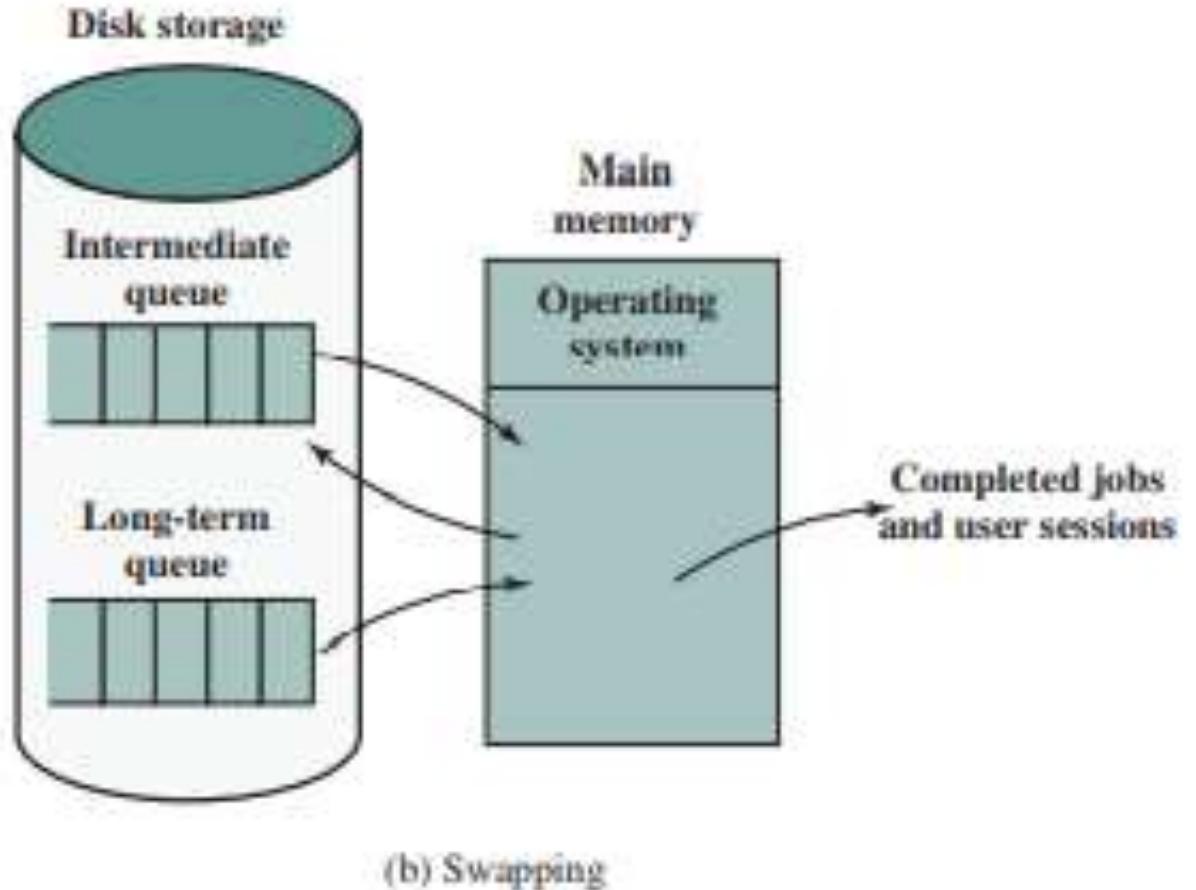


Figure 8.12 The Use of Swapping

This process is
Occupying space.
And will be idle for
A long time compared
To the non-blocked
Processes.

We can insert
Another process
In place of this one
Using the addresses in
The long term queue



Solution: Swapping (Cont.)

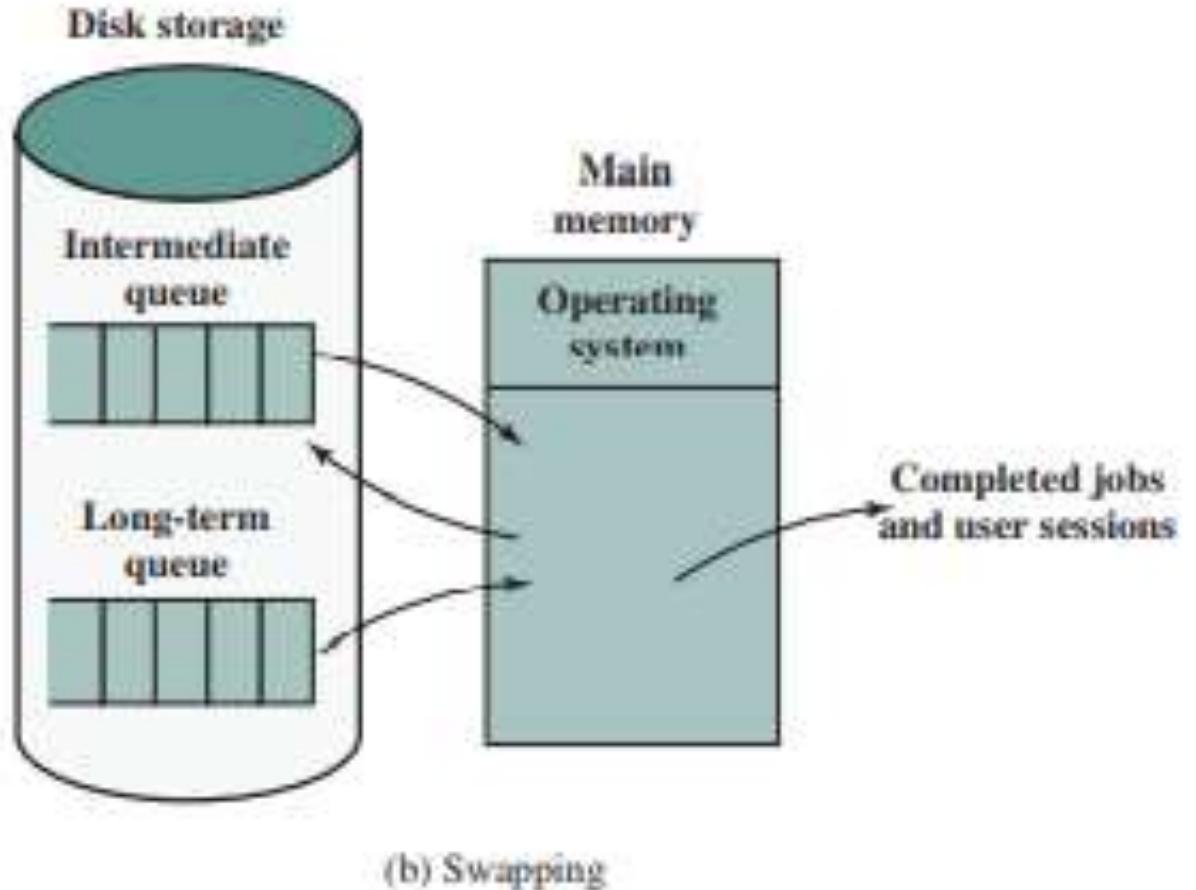


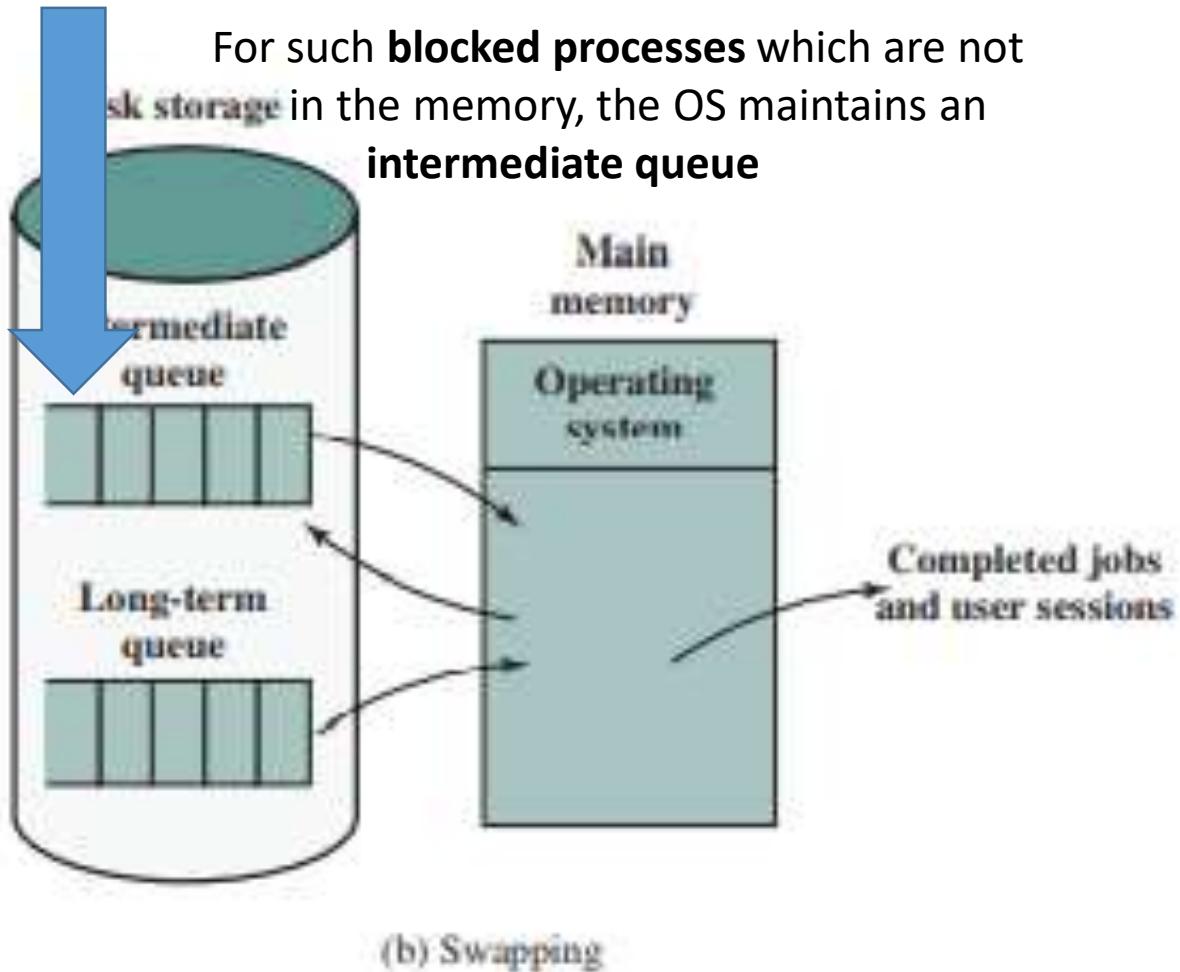
Figure 8.12 The Use of Swapping

This process is Occupying space. And will be idle for A long time compared To the non-blocked Processes.

We can insert Another process In place of this one Using the addresses in The long term queue



Solution: Swapping (Cont.)



This process is Occupying space. And will be idle for A long time compared To the non-blocked Processes.

We can insert Another process In place of this one Using the addresses in The long term queue



Figure 8.12 The Use of Swapping

Solution: Swapping (Cont.)

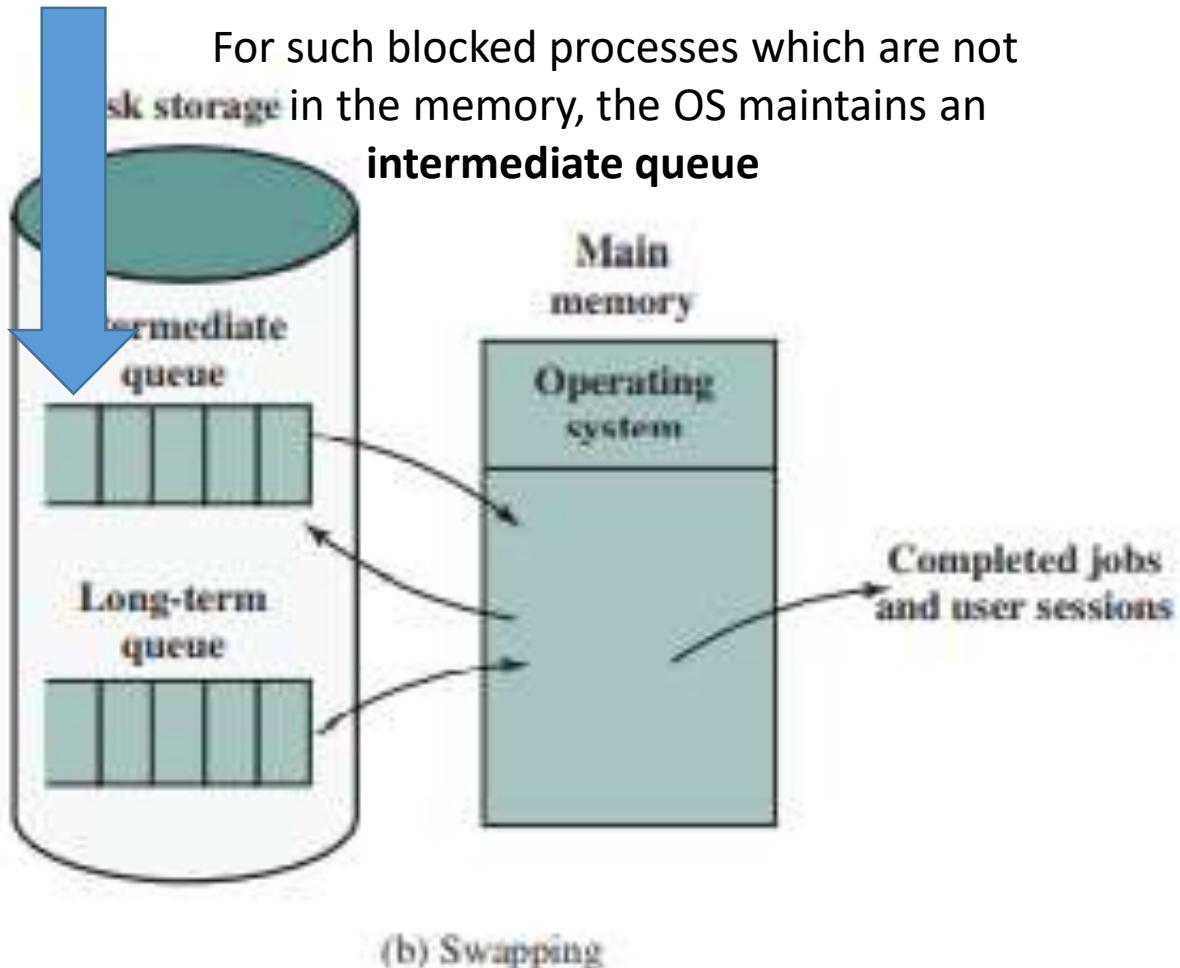


Figure 8.12 The Use of Swapping

This process is Occupying space. And will be idle for A long time compared To the non-blocked Processes.

We can insert Another process In place of this one Using the addresses in The long term queue

Later on, If the OS's scheduler finds the process to be **unlockable**, then it will simply load its code into the memory.
Remove its info from the **intmdt** queue and mark it as **ready**.



- Swapping is I/O operation so there is a chance of making things worse
- But disk I/O is comparatively faster than other I/O (printer or tape etc.)
- Usually improve performance

3.3 Partitioning

Partitioning

- We have understood swapping.
- Basically it is referring to swapping the blocked process codes in the RAM with the new process codes in the disc.
- (Though it is not an actual swap as no write operation is done in the DISC)
- However, every time the OS tries to bring in a **new process**, it will have to be **careful** about whether or not its code is becoming **overlapped** with another process's code.
- There is **no organized approach** for this so far.
- Hence we introduce the concept of Partitioning .

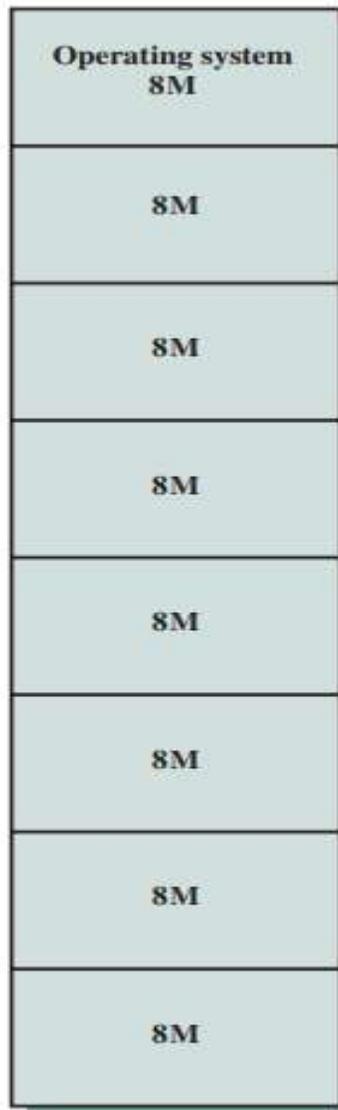
Partitioning

- Equal sized partitions
- Variable sized partitions

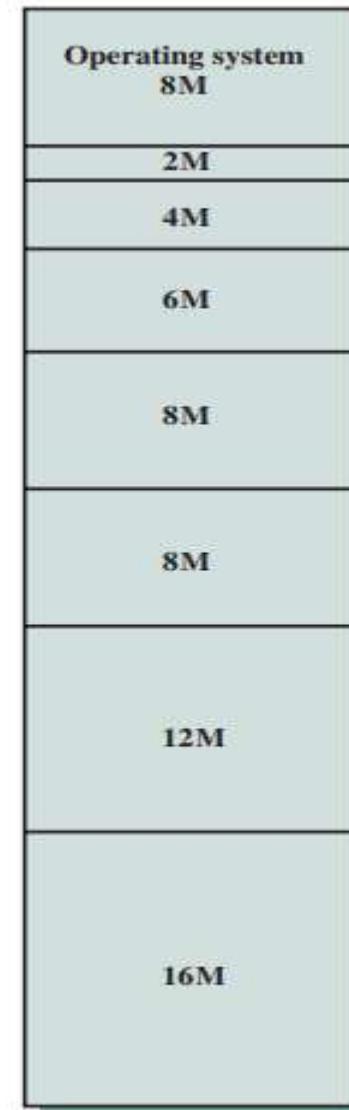
Now it is not **required to check**
Other processes,

The OS just needs to keep info about
Whether a **partition is filled up or not.**

Then while placing a new process, it
Just has to place the **codes** in the
Unoccupied partitions.



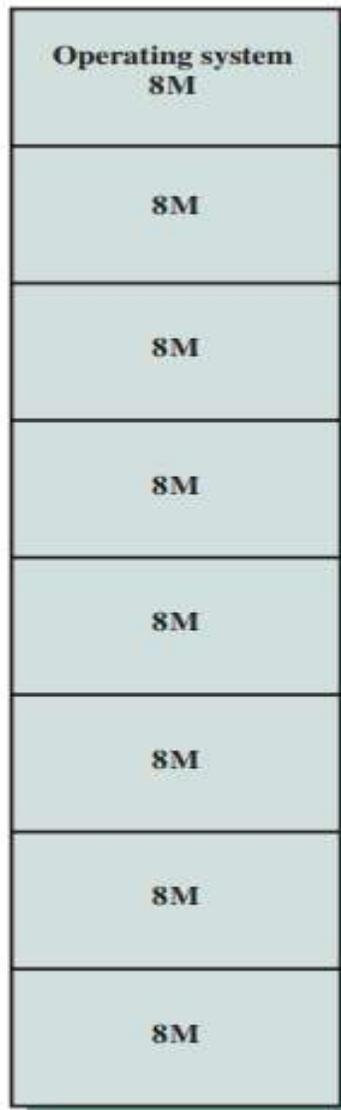
(a) Equal-size partitions



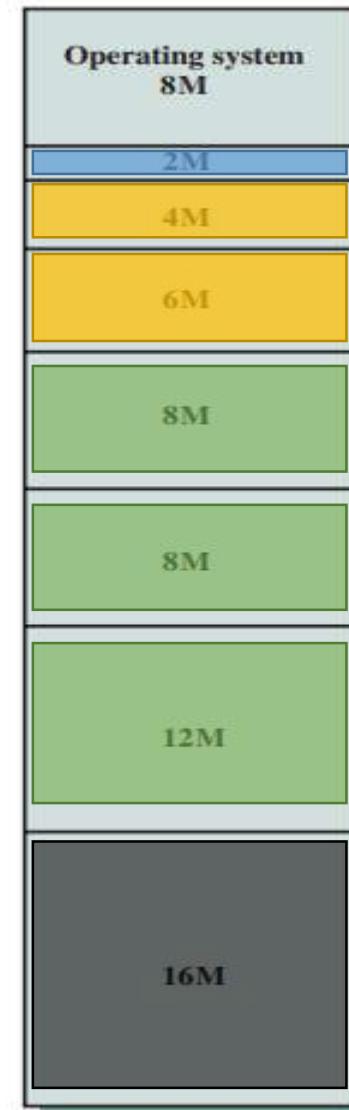
(b) Unequal-size partitions

Partitioning

- Equal sized partitions
- Variable sized partitions



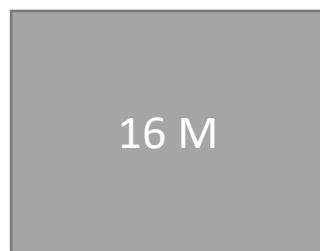
(a) Equal-size partitions



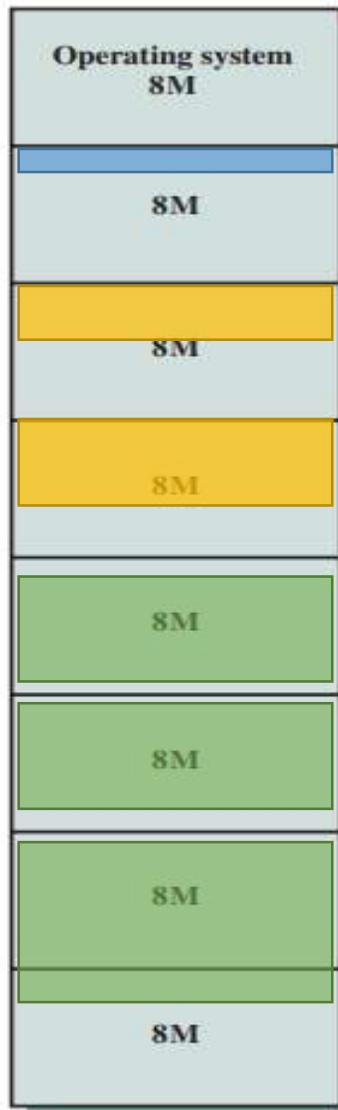
(b) Unequal-size partitions

Partitioning

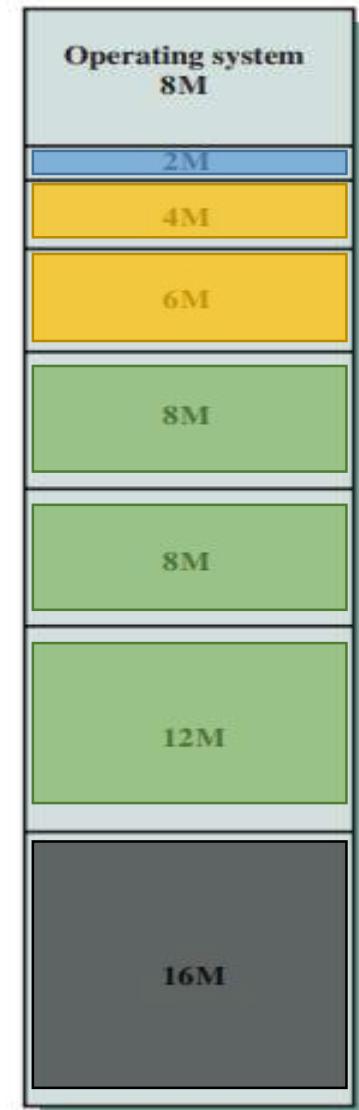
- Equal sized partitions
- Variable sized partitions



Variable sized partitions are better



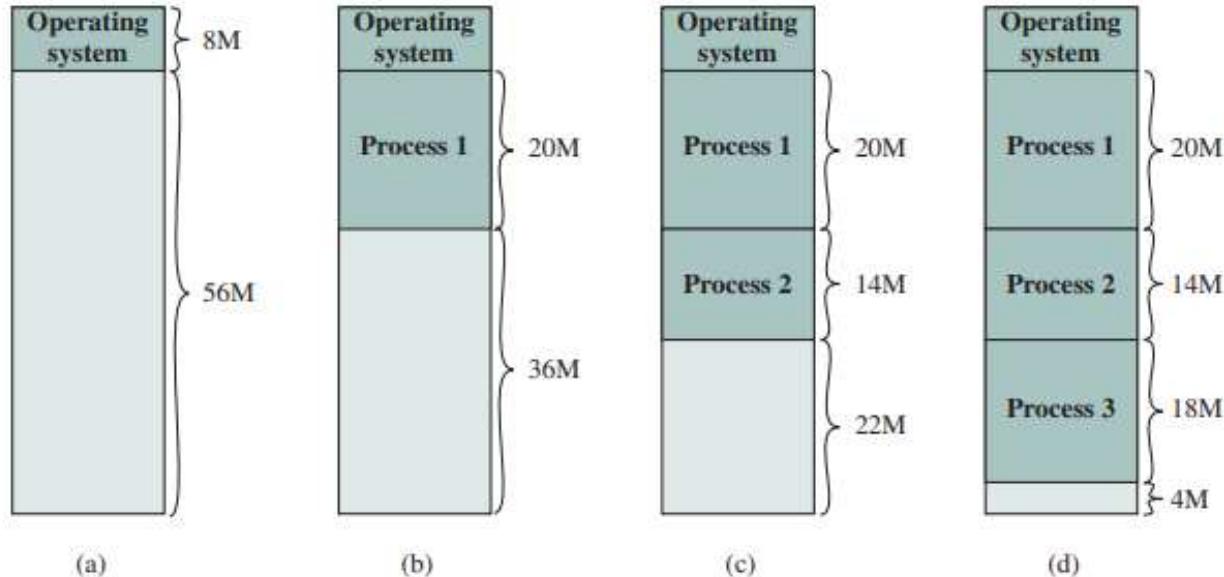
(a) Equal-size partitions



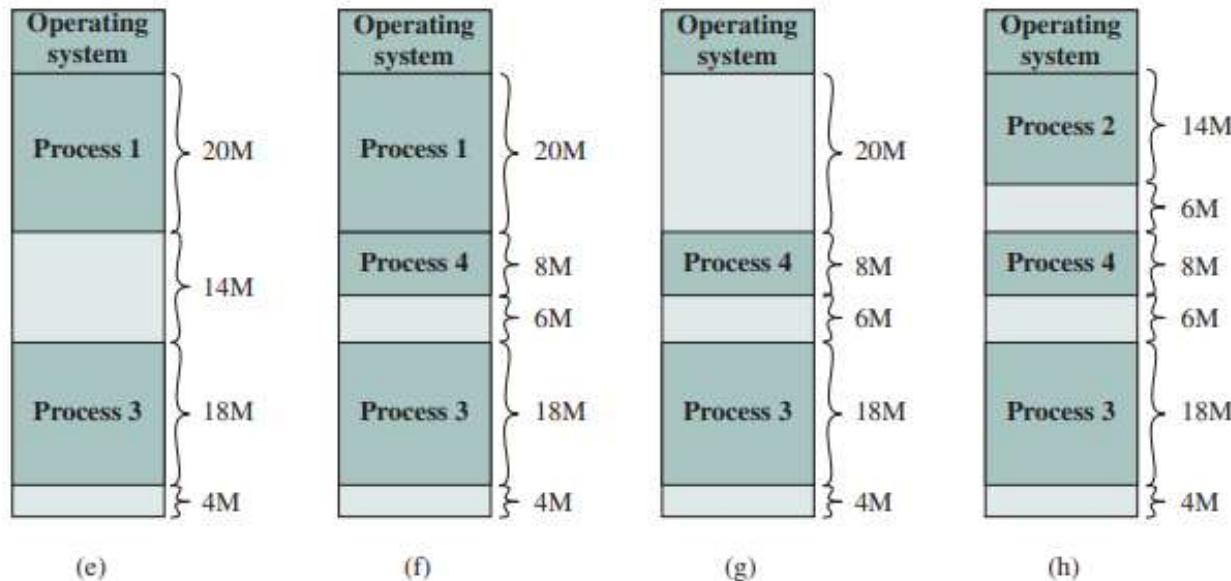
(b) Unequal-size partitions

Partitioning

- Issues with **variable sized partitions**



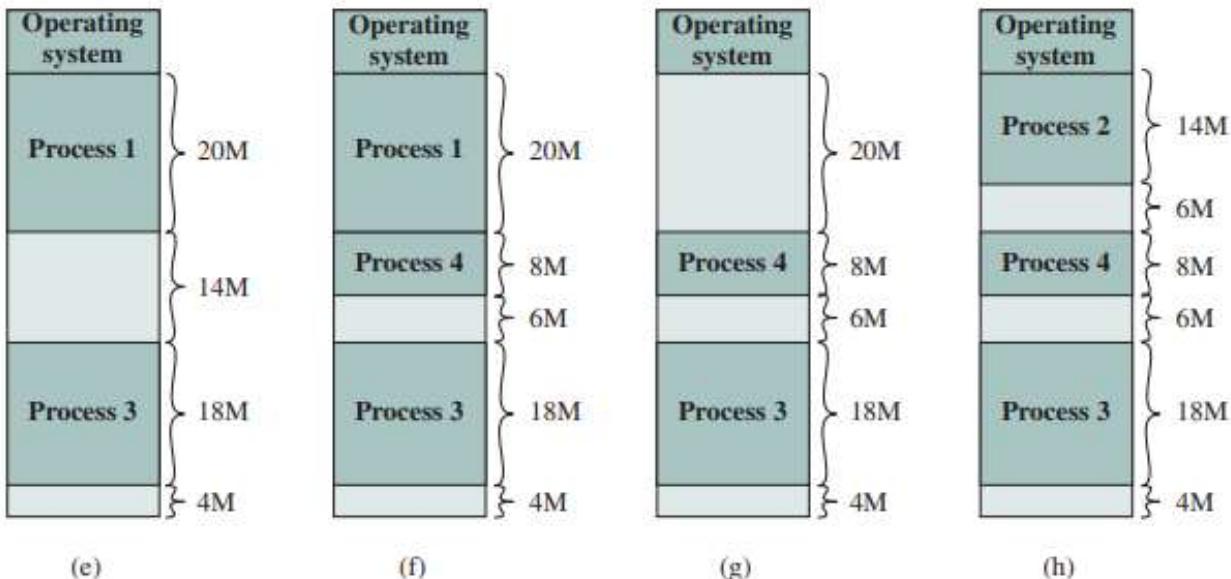
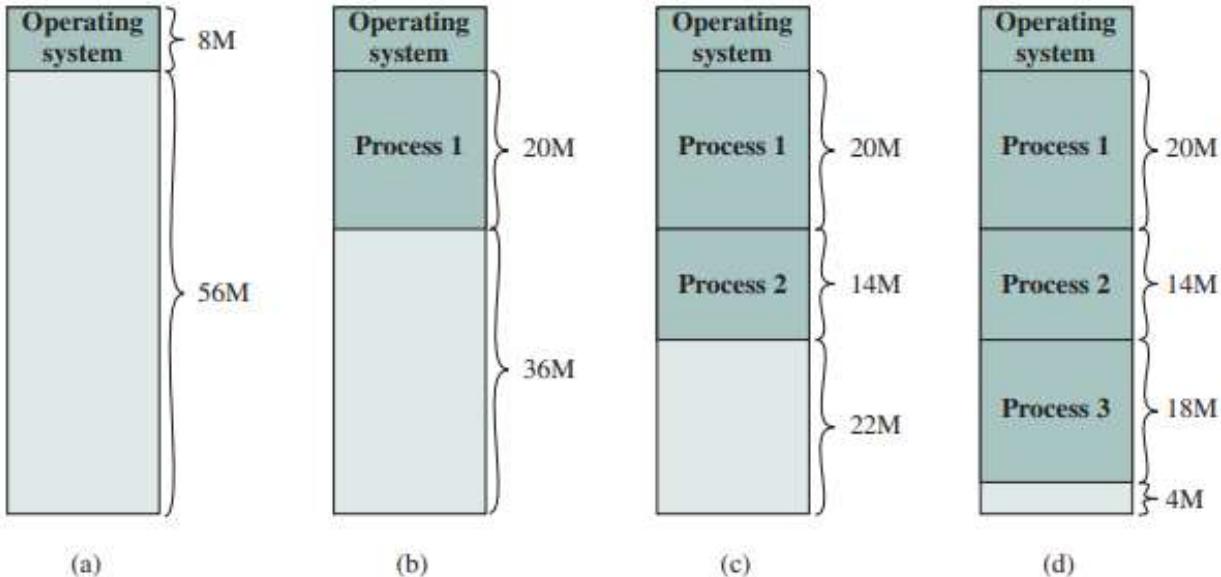
- **FRAGMENTATION**
- Leads to small holes in memory
- Compaction is required but time consuming



Partitioning

- **SOLUTION:**

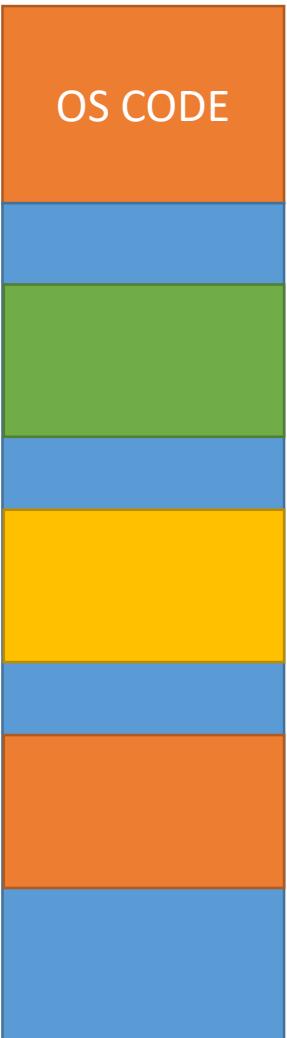
Dynamic
variable
partitions with
compaction



Partitioning

- SOLUTION:

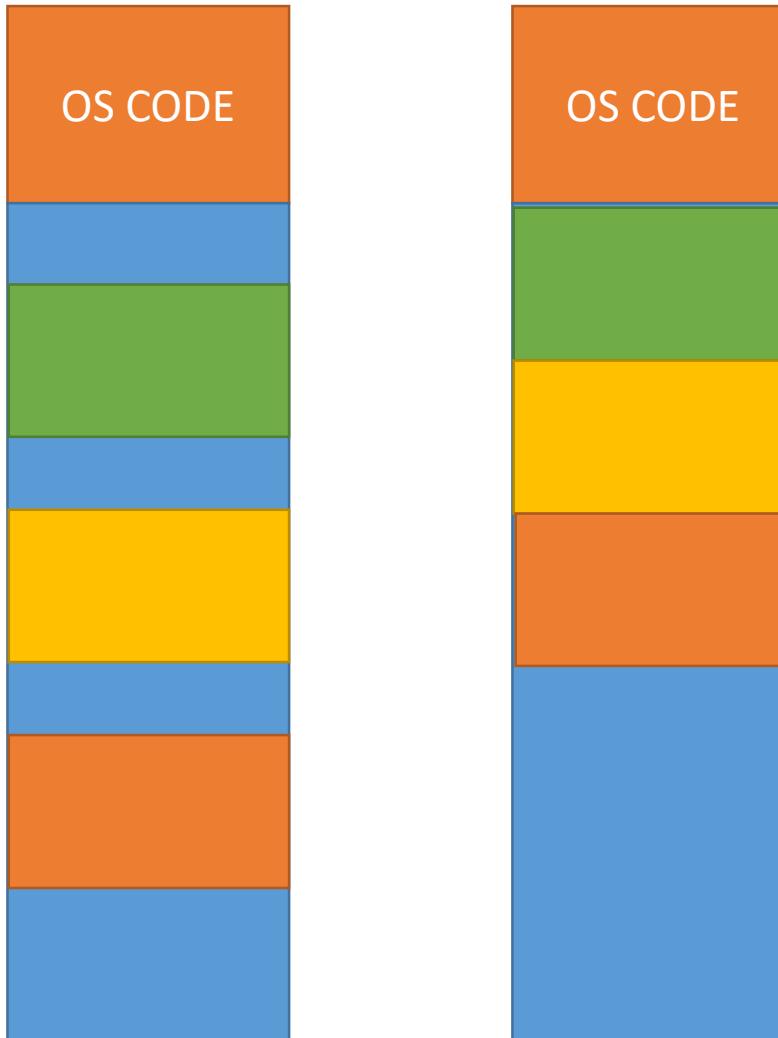
Dynamic
variable
partitions with
compaction



Partitioning

- SOLUTION:

Dynamic
variable
partitions with
compaction

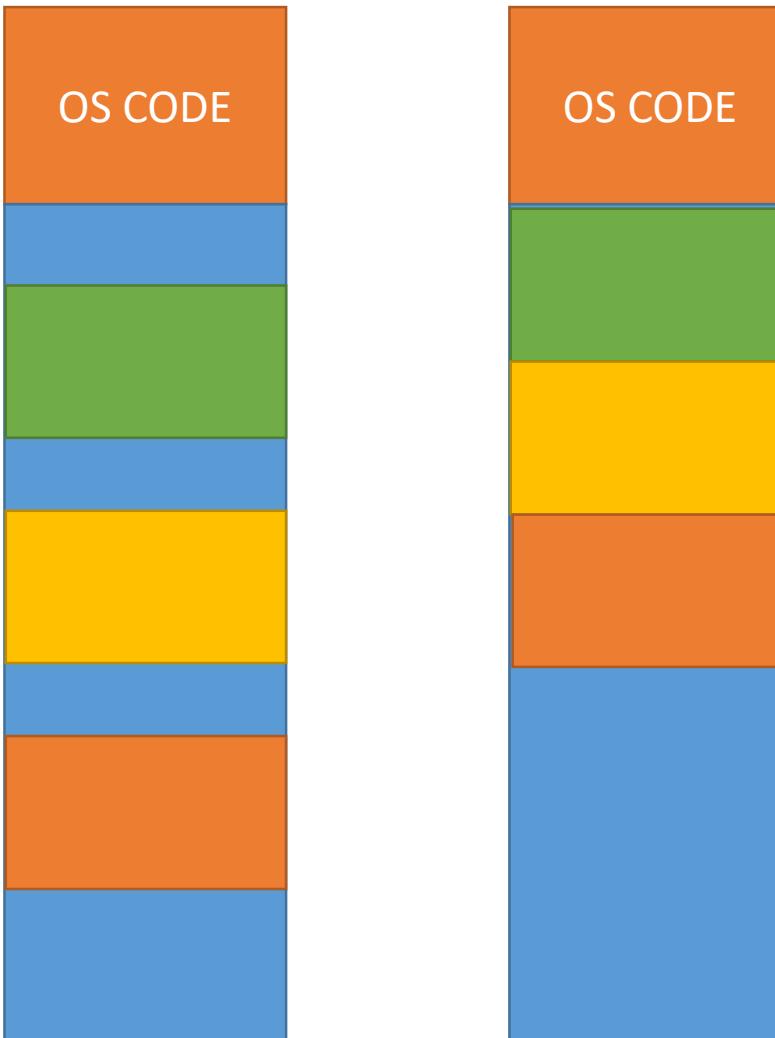


Partitioning

- SOLUTION:

Dynamic
variable
partitions with
compaction

Compaction is
requiring several
memory read
write operations



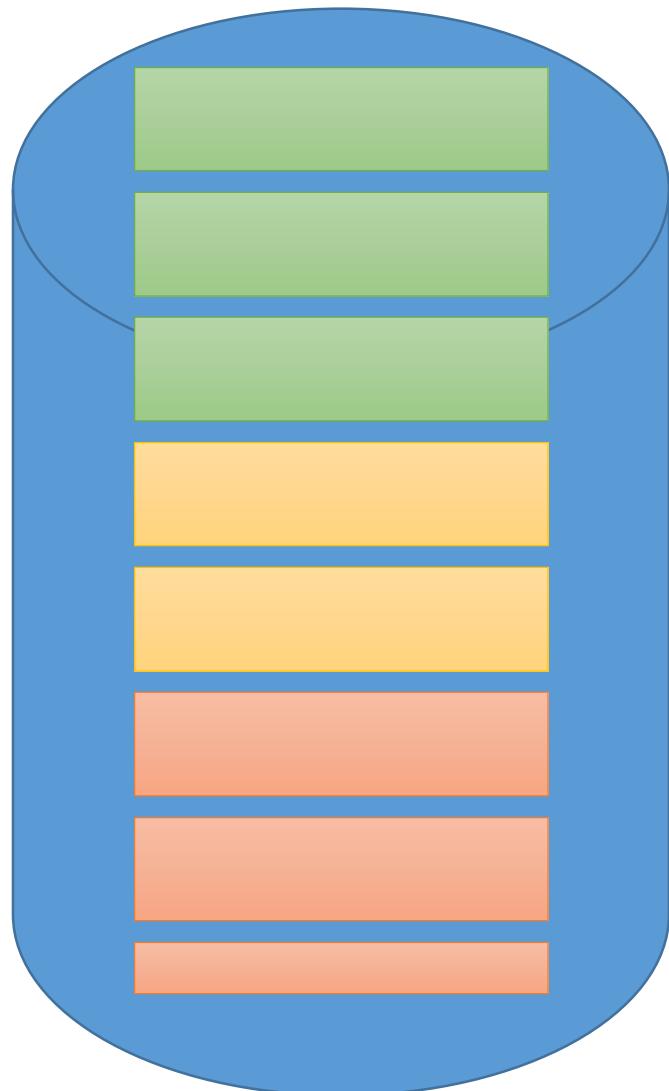
Note that :
We need to use
Displacement addressing
Here for the process codes
as the physical
Processes can be placed
at any part of the memory.

3.4 Paging and Page Tables

A more efficient approach : Paging

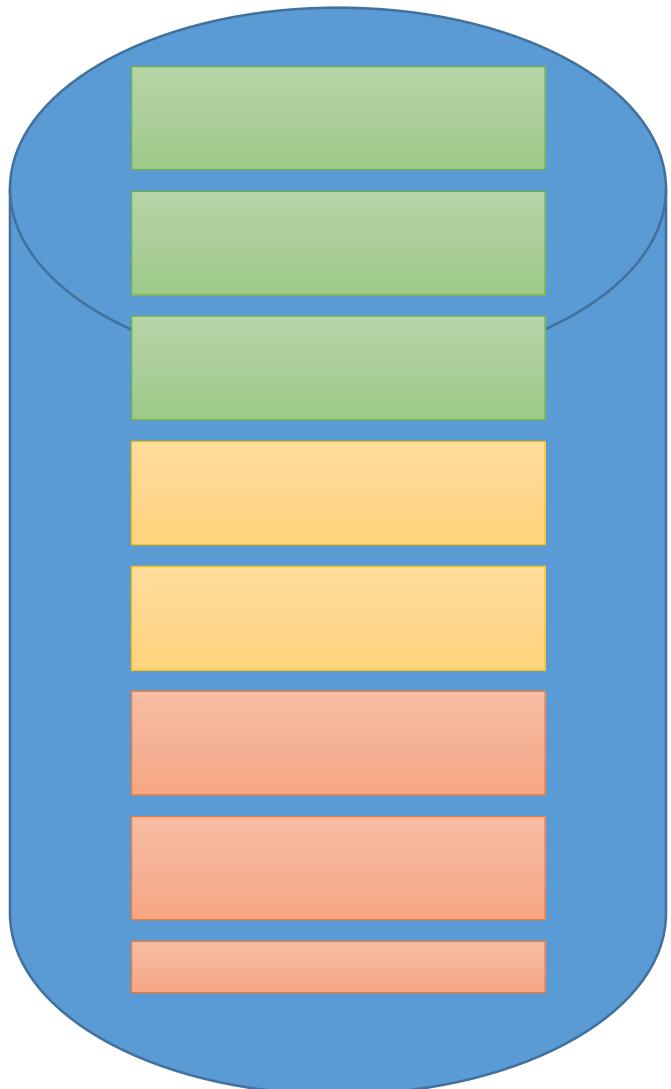
- Both of the fixed and unfixed **partitioning** mechanisms are **inefficient**. May result in waste of a lot of space.
- Compaction requires several **memory read write operations**.
- Thus we try another mechanism called **Paging**.

A more efficient approach : Paging



Programs with **fixed sized pages**

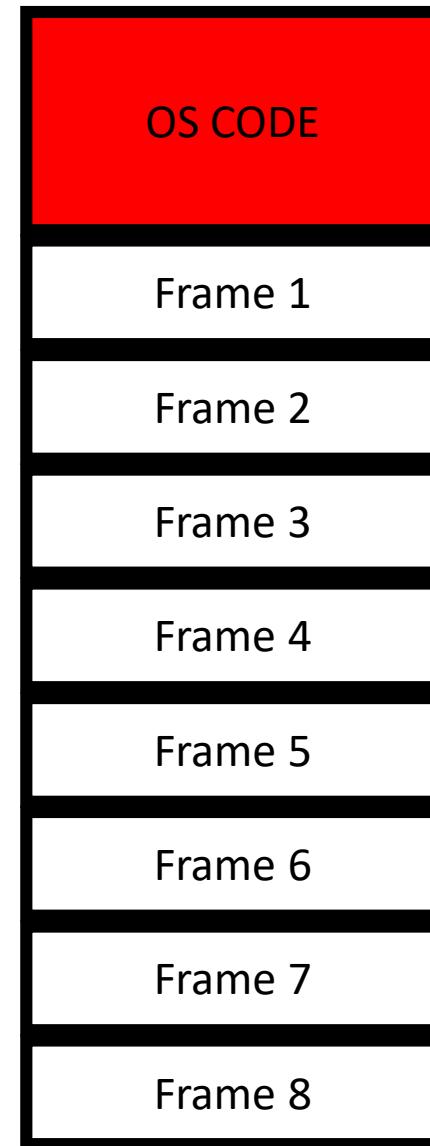
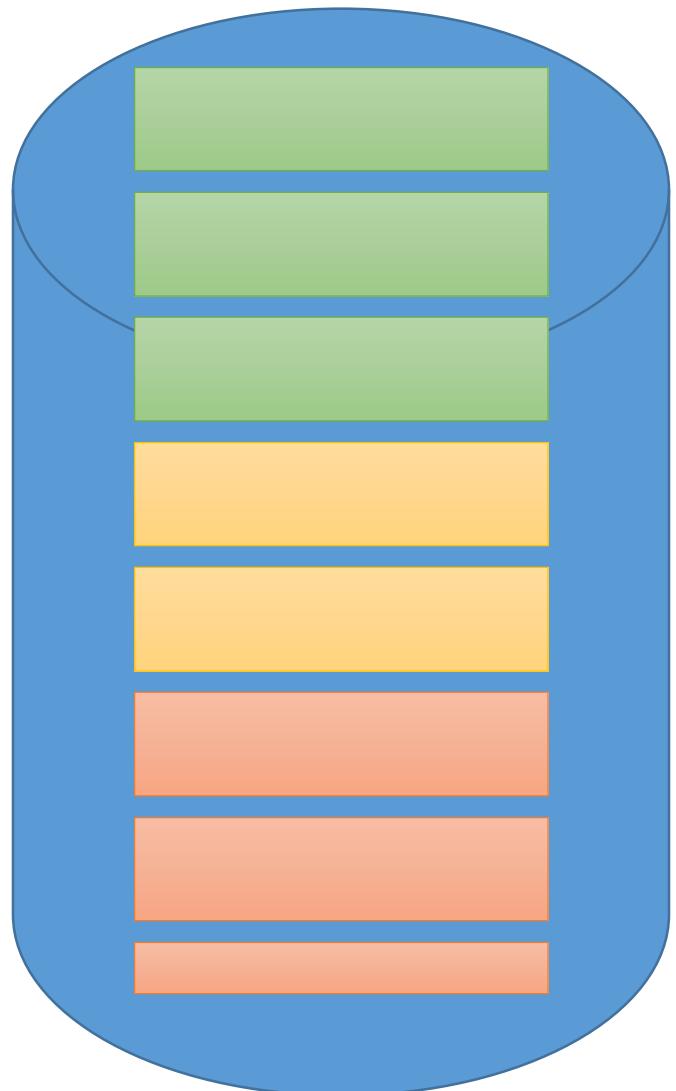
A more efficient approach : Paging



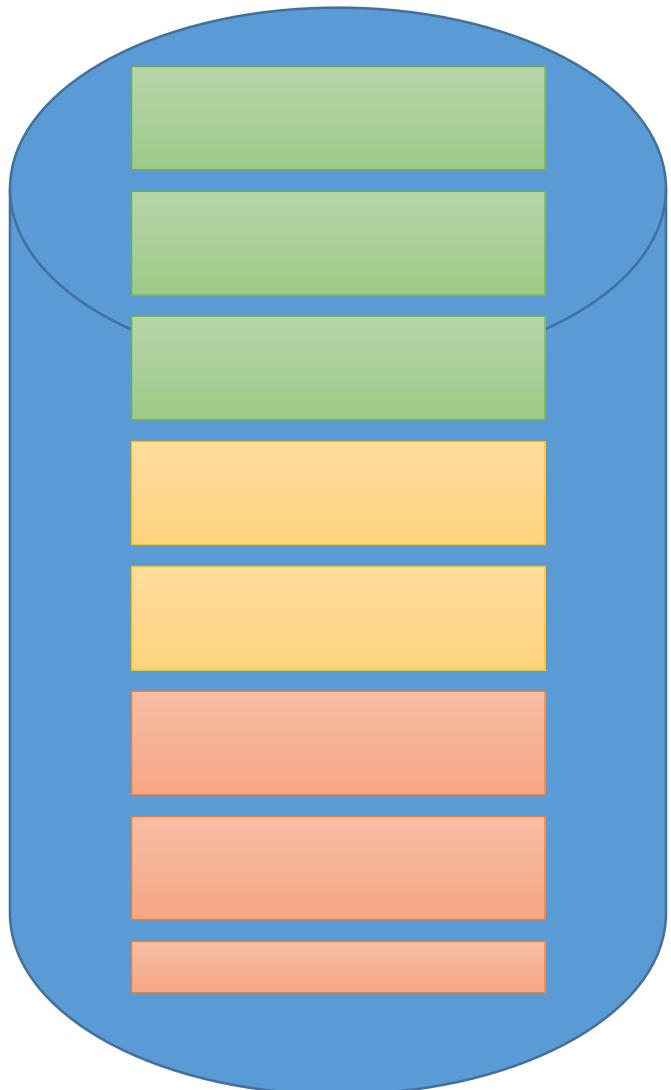
Programs with fixed sized pages

Note that some programs may have
Partial page sizes.

A more efficient approach : Paging

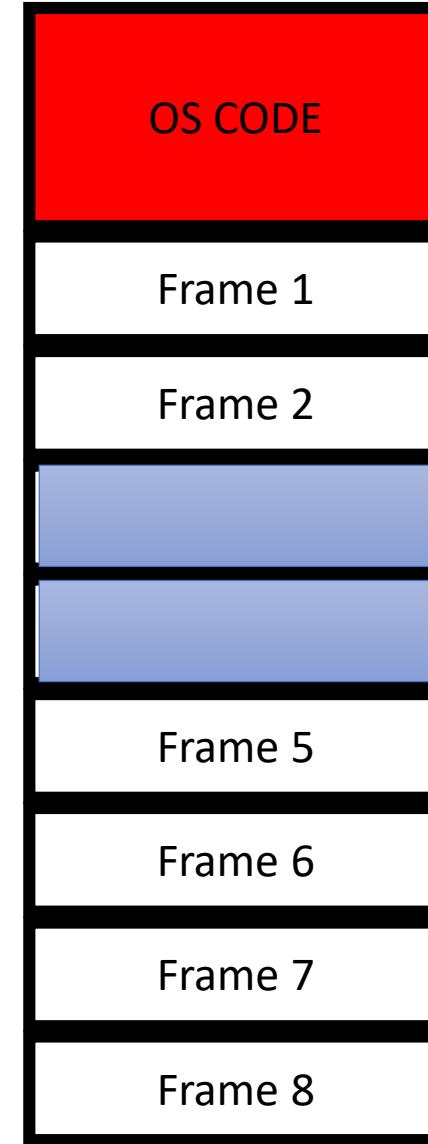


A more efficient approach : Paging

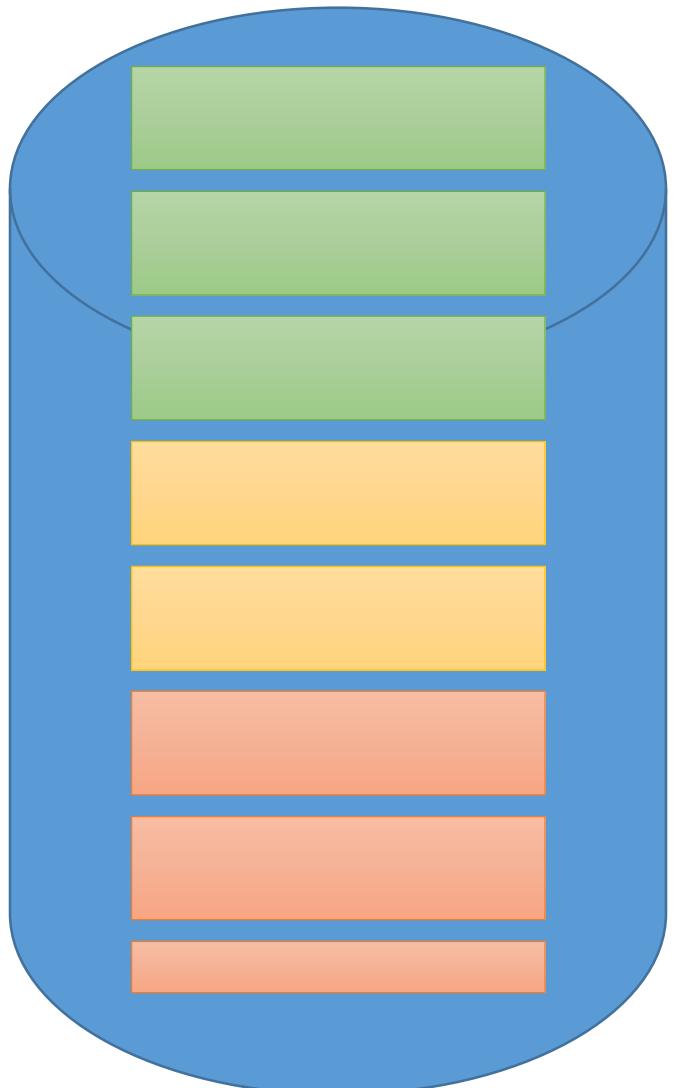


Available
chunks of
memory
referred to as
frames

Assume that
This program
With **two pages**
Was already there

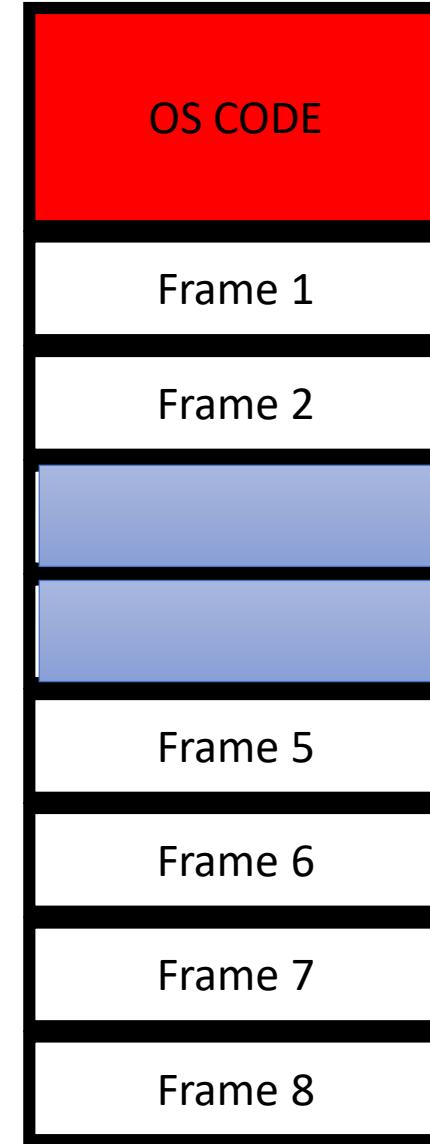


A more efficient approach : Paging

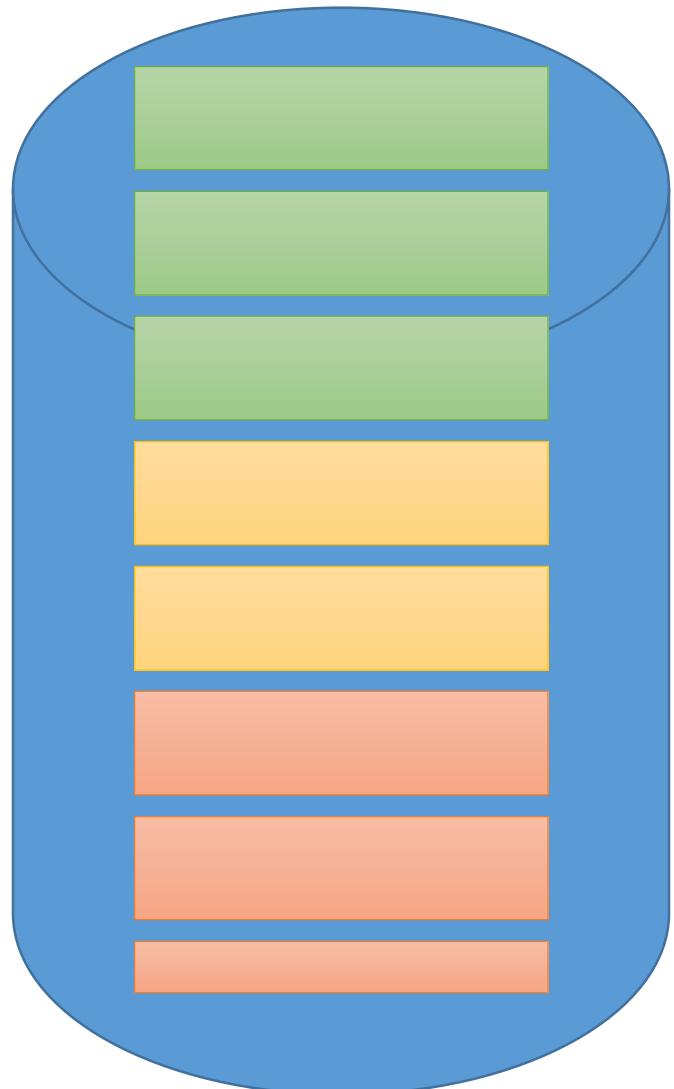


We want to **bring in**
This program

Assume that
This program
With two pages
Was already there

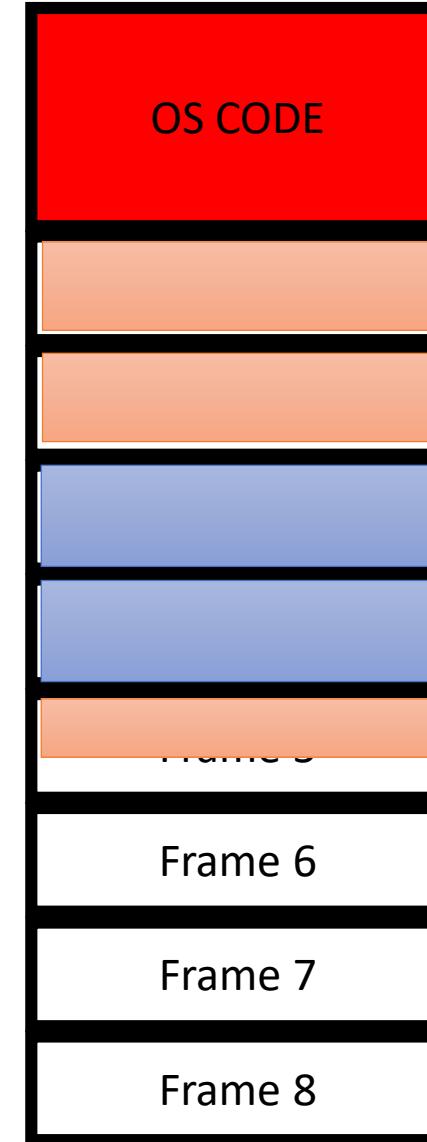


A more efficient approach : Paging

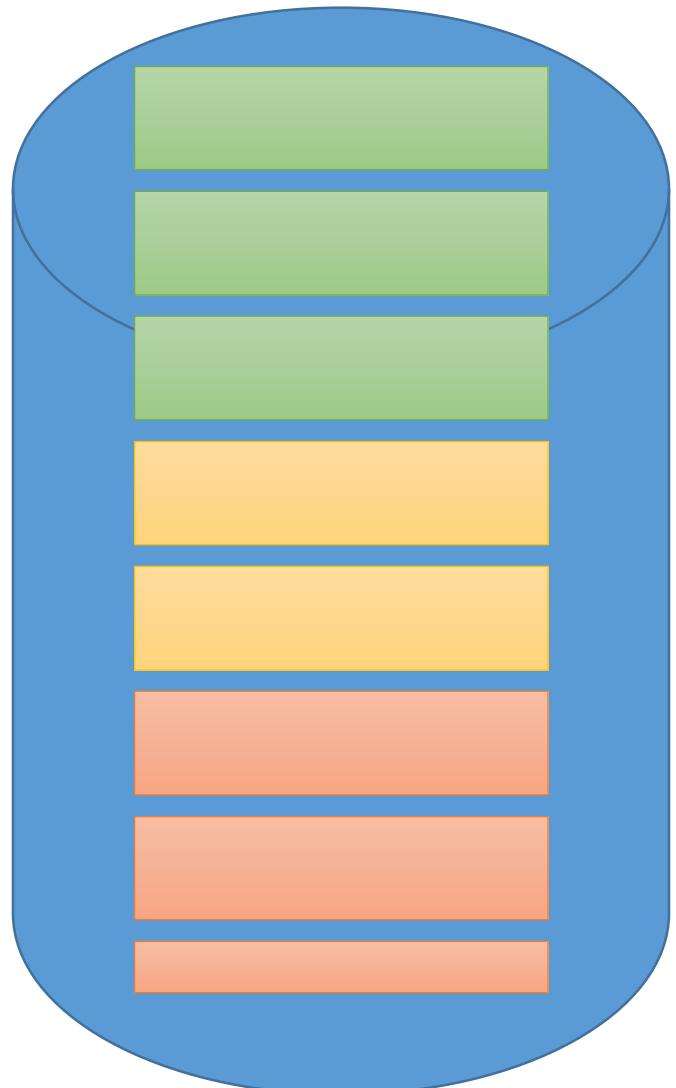


We want to bring in
This program

Assume that
This program
With two pages
Was already there



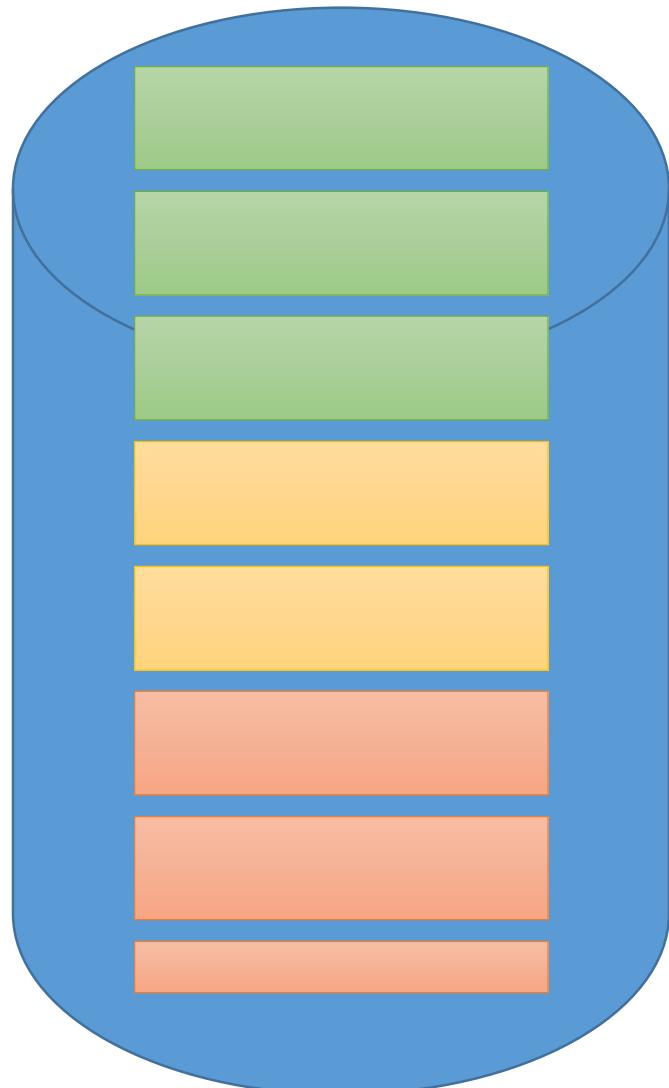
A more efficient approach : Paging



Assume that
This program
With two pages
Was already there



A more efficient approach : Paging



We want to bring in
This program

Assume that
This program
With two pages
Was already there



But observe that
The code is no longer
Present in a
Sequential
manner

Very little wastage

Page table

- A **mapping** is required to be maintained.
- For keeping track :
 - Which **page** of a program is in which **frame**
 - We need to use a **new kind of addressing mode**.
 - Now when a processor gets an instruction with an address it will be in the form **(page no, relative address)**
 - it will change it into the format **(frame, relative address)** using the page table.
- For **each process**, OS will maintain a **page table**.

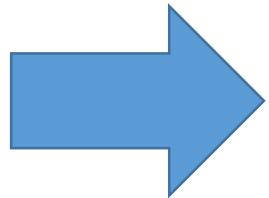
Example

```
CMP AX,BX  
JGE #100  
MOV AX, BX  
ADD AX, BX  
LOAD AX, #200
```

....
...
...
...

Example

```
CMP AX,BX  
JGE #100  
MOV AX, BX  
ADD AX, BX  
LOAD AX, #200  
....  
....  
....  
....
```



The
Program
Is divided
Into pages

```
CMP AX,BX  
JGE #100  
MOV AX, BX  
ADD AX, BX  
LOAD AX, #200
```

....
....
....
....
....

....
....
....
....
....
....

Example

```
CMP AX,BX  
JGE #100  
MOV AX, BX  
ADD AX, BX  
LOAD AX, #200  
....  
....  
....  
....
```

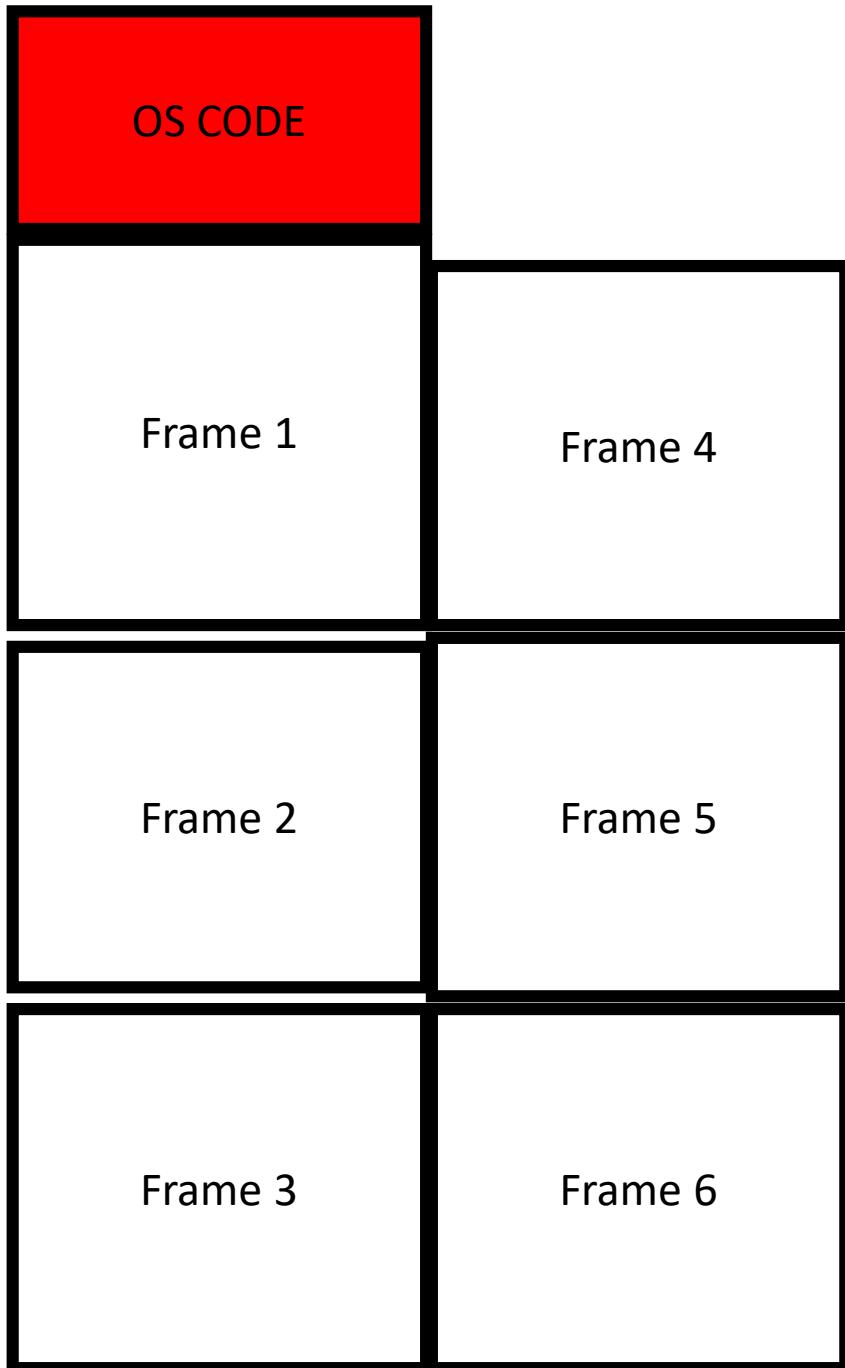
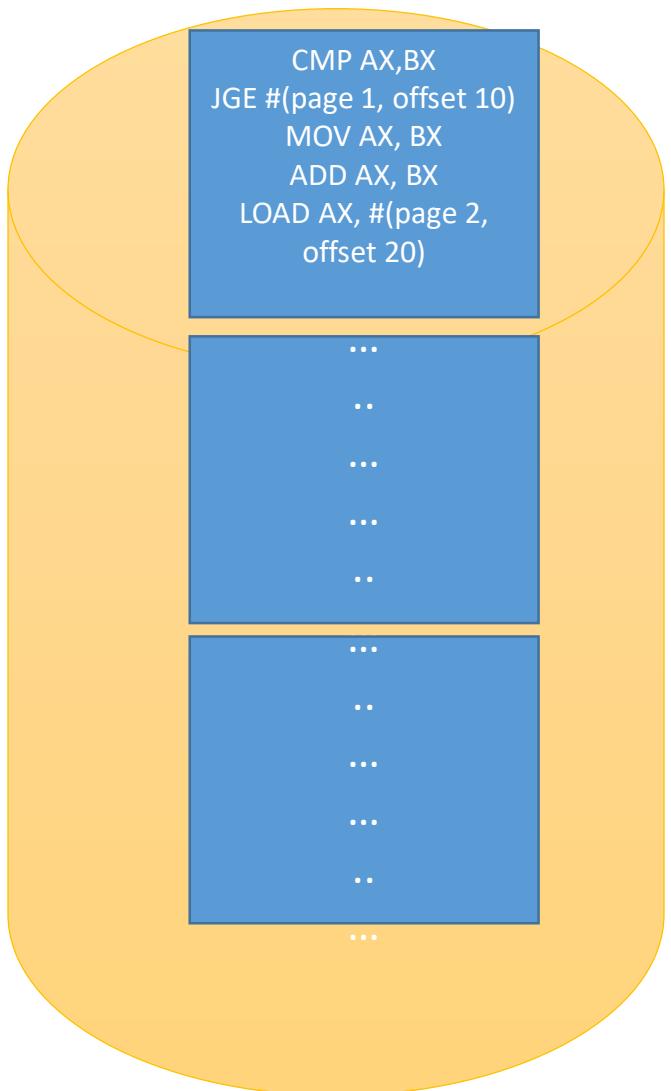
The
Program
Is divided
Into pages

```
CMP AX,BX  
JGE #100  
MOV AX, BX  
ADD AX, BX  
LOAD AX, #200
```

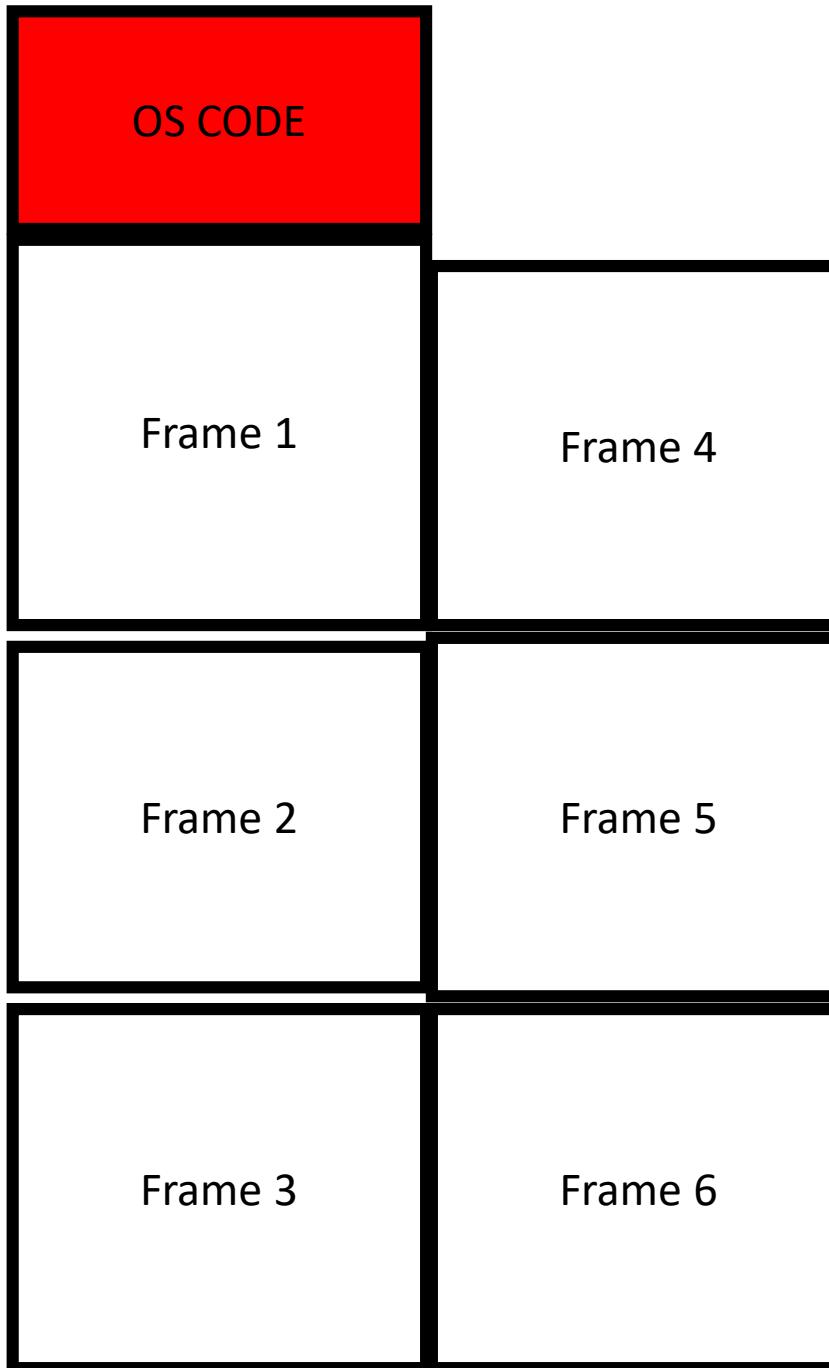
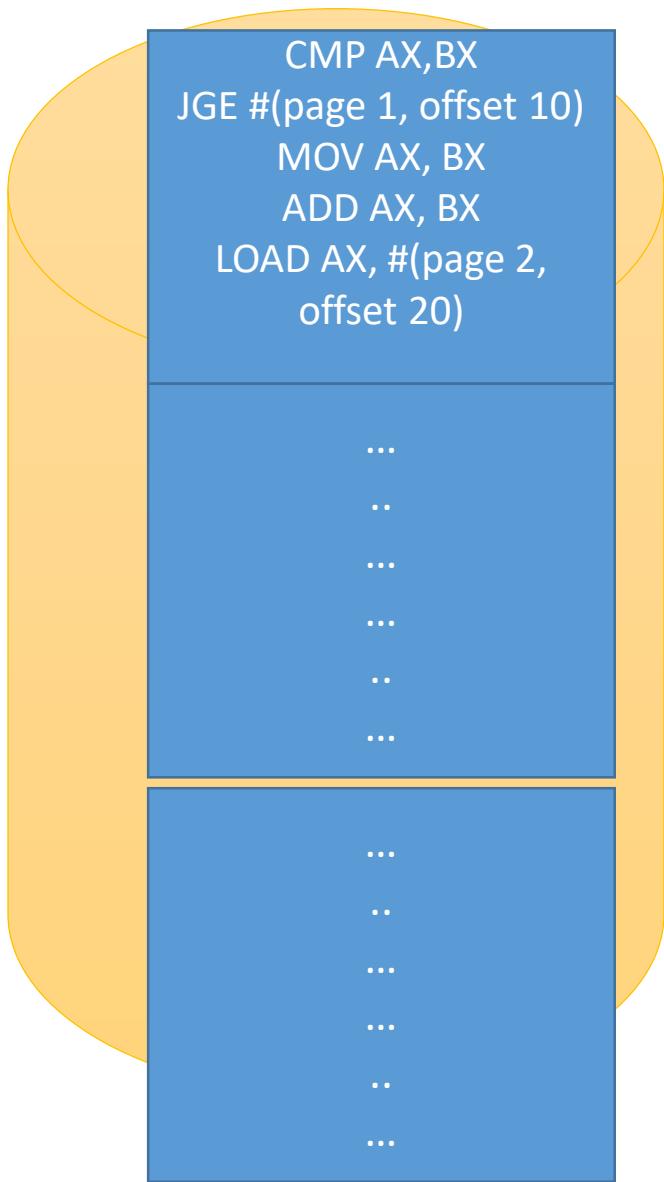
Observe
that
The
addresses
are
written
In a new
format
By the
assembler

```
CMP AX,BX  
JGE #(page 1, offset 10)  
MOV AX, BX  
ADD AX, BX  
LOAD AX, #(page 2, offset 20)
```

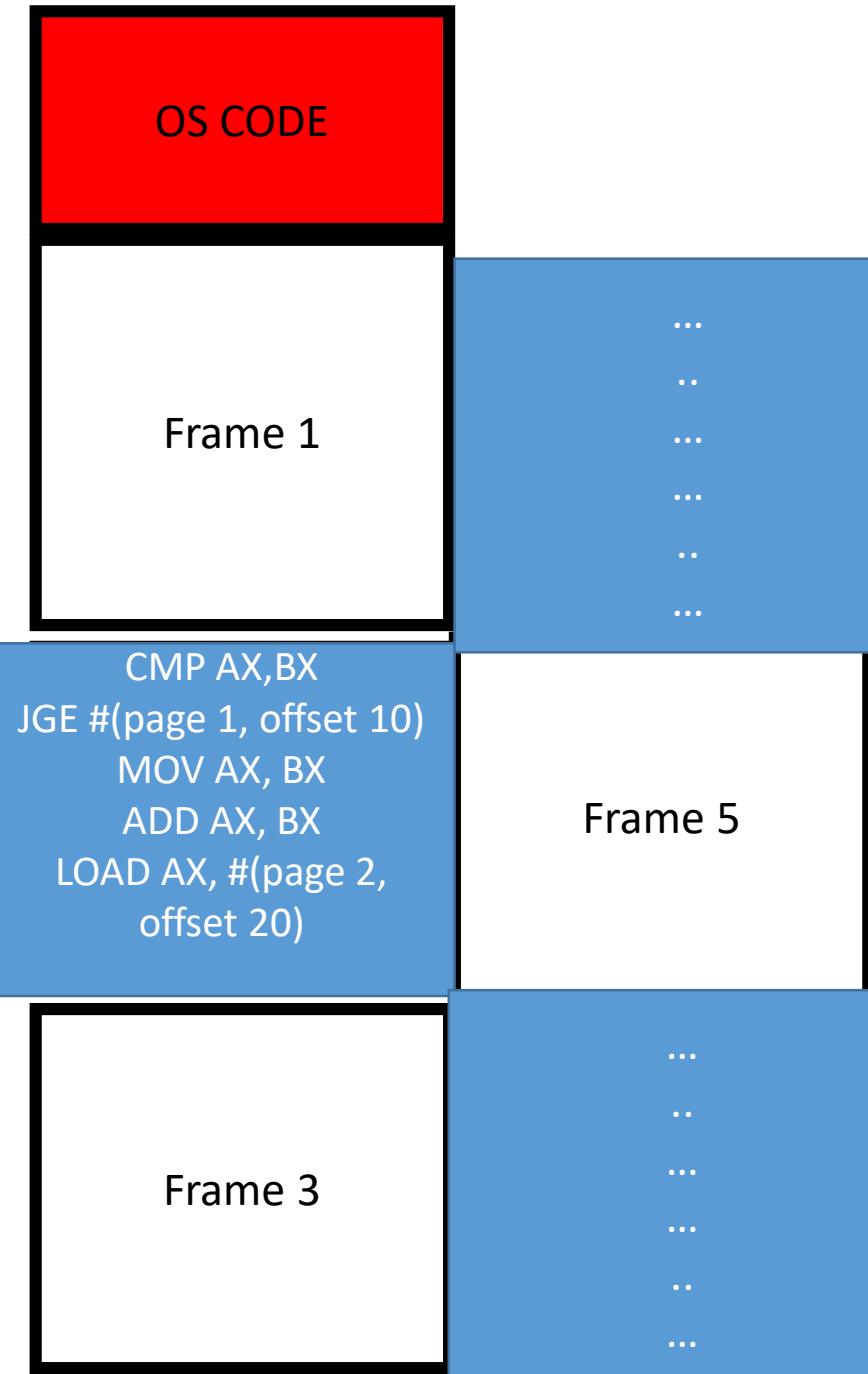
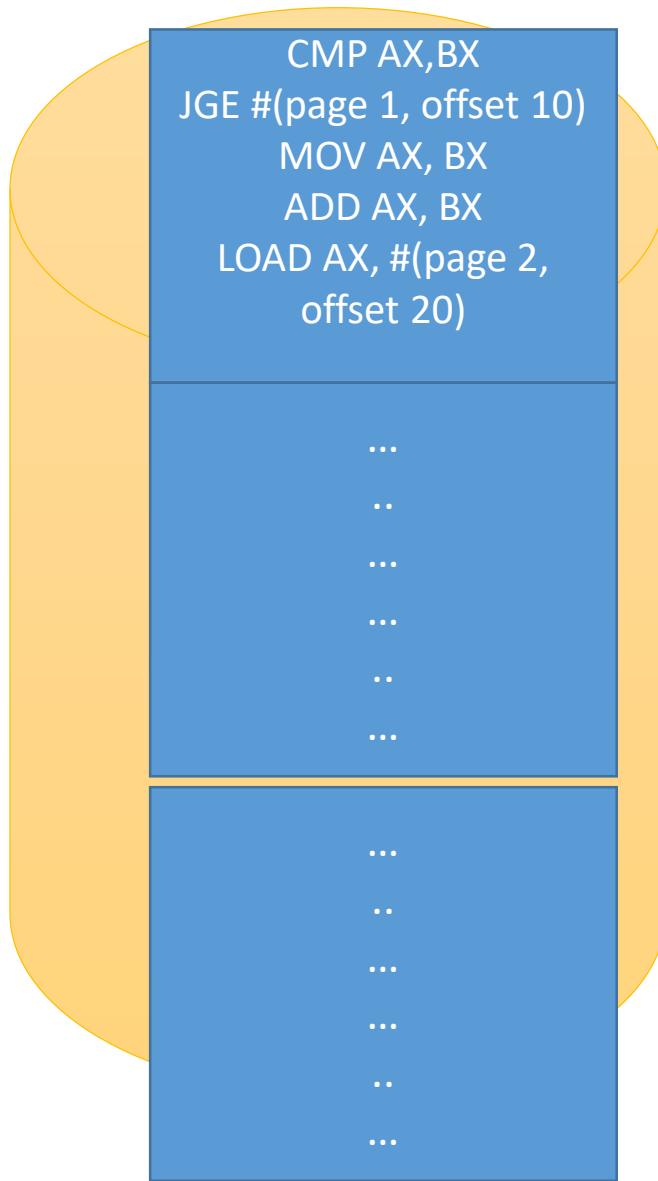
A more efficient approach : Paging



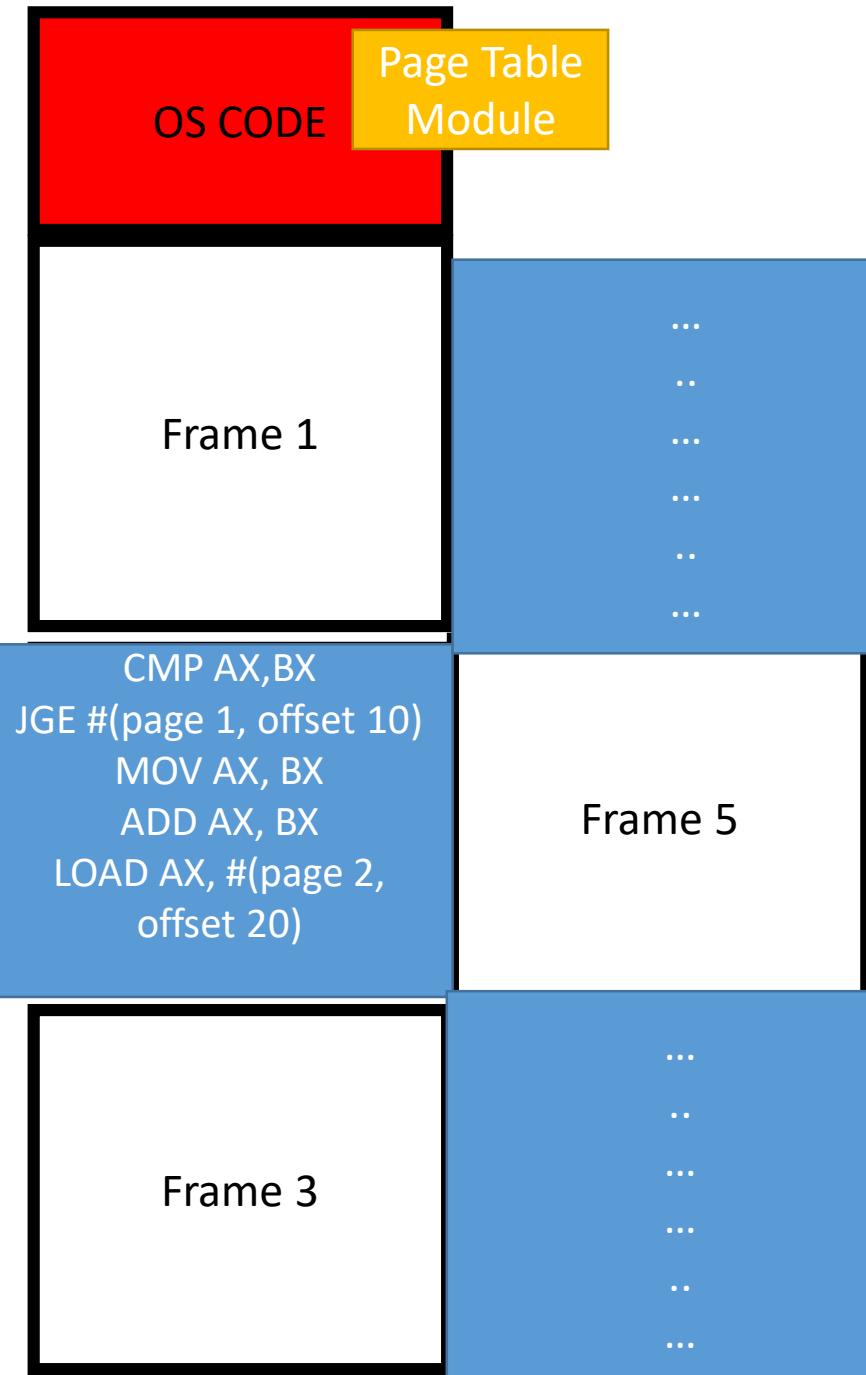
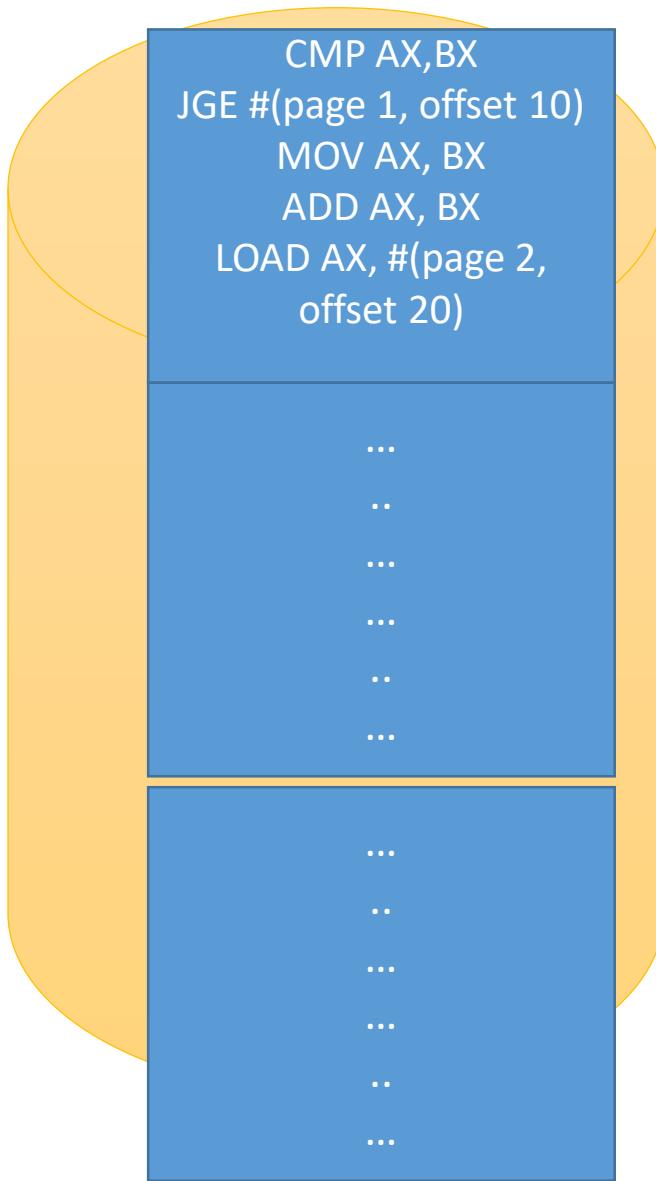
A more efficient approach : Paging



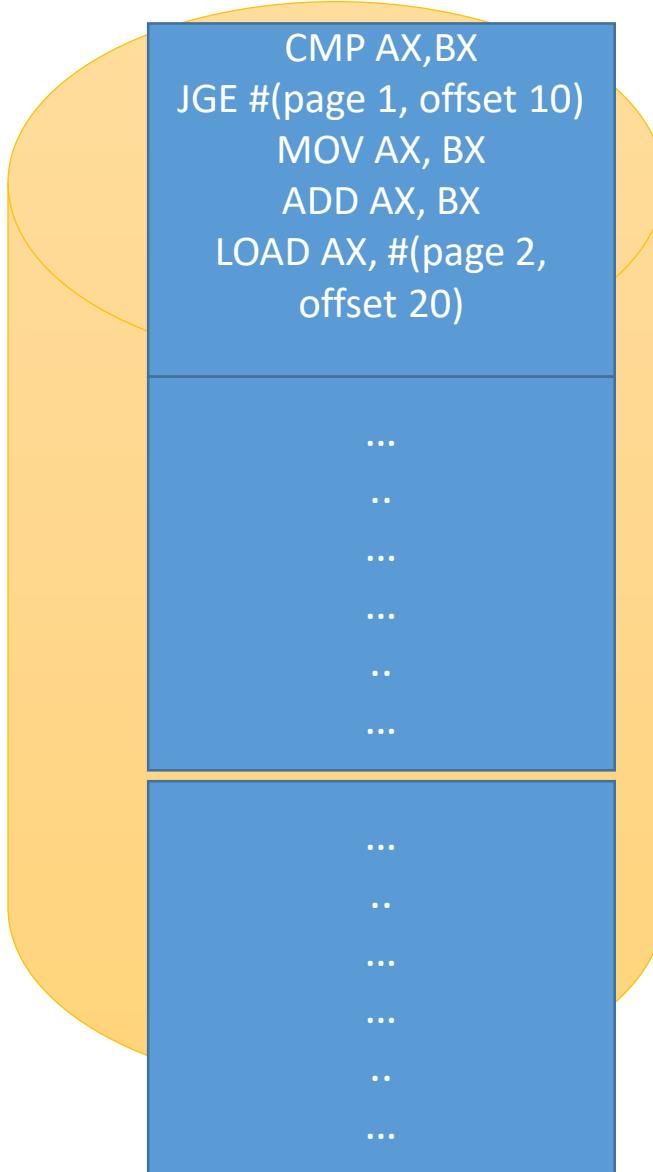
A more efficient approach : Paging



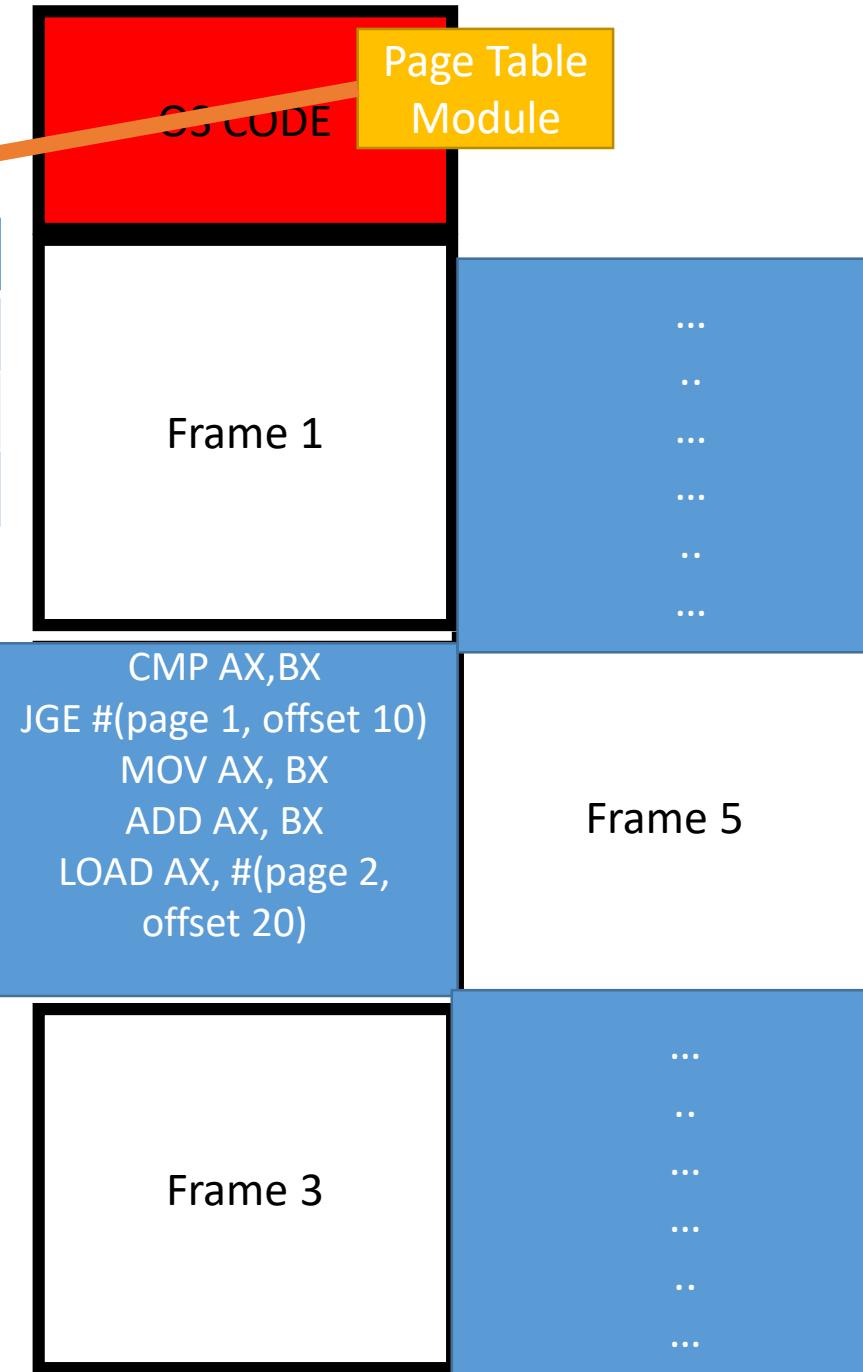
A more efficient approach : Paging



A more efficient approach : Paging



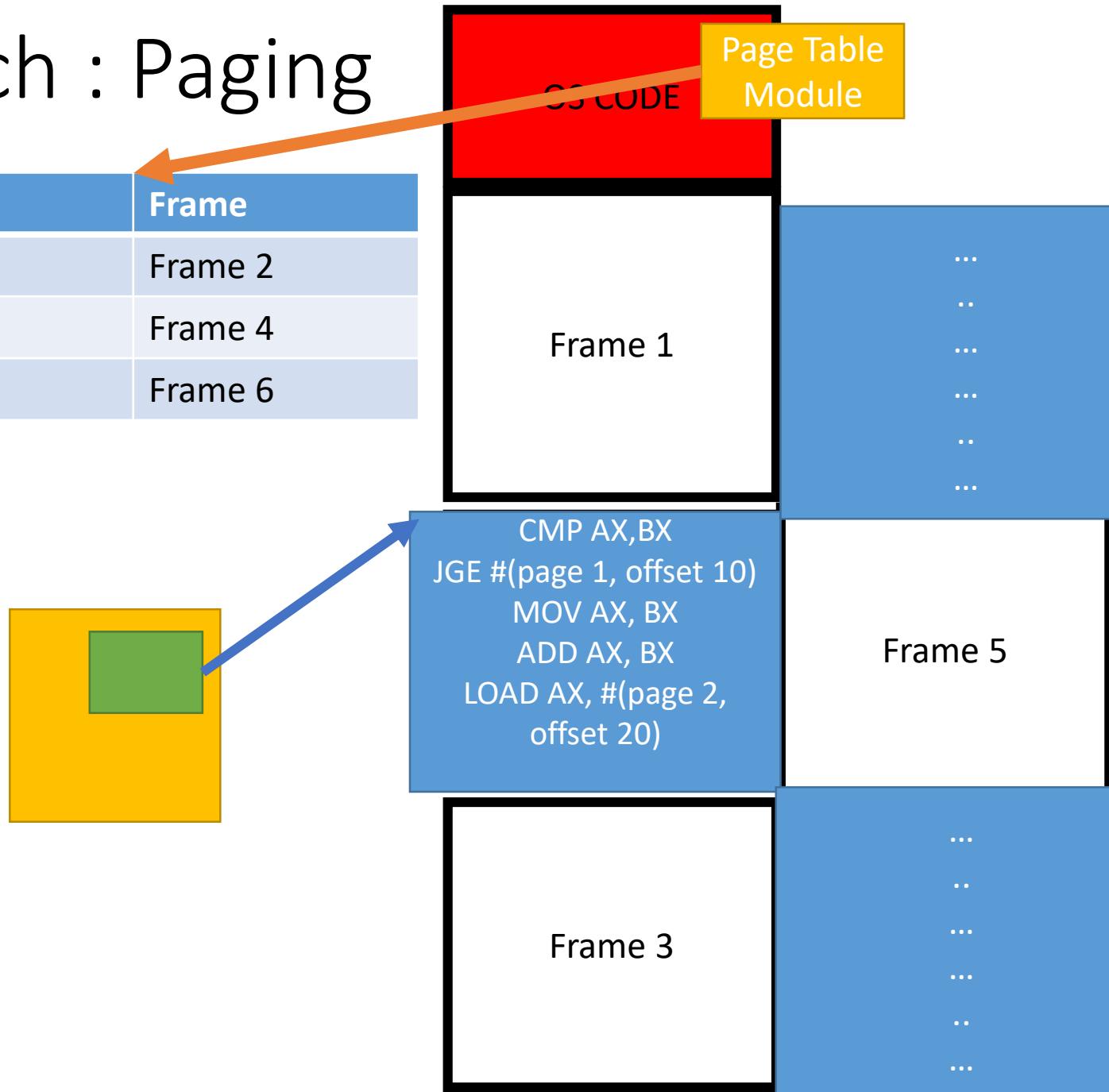
| Page | Frame |
|--------|---------|
| Page 1 | Frame 2 |
| Page 2 | Frame 4 |
| Page 3 | Frame 6 |



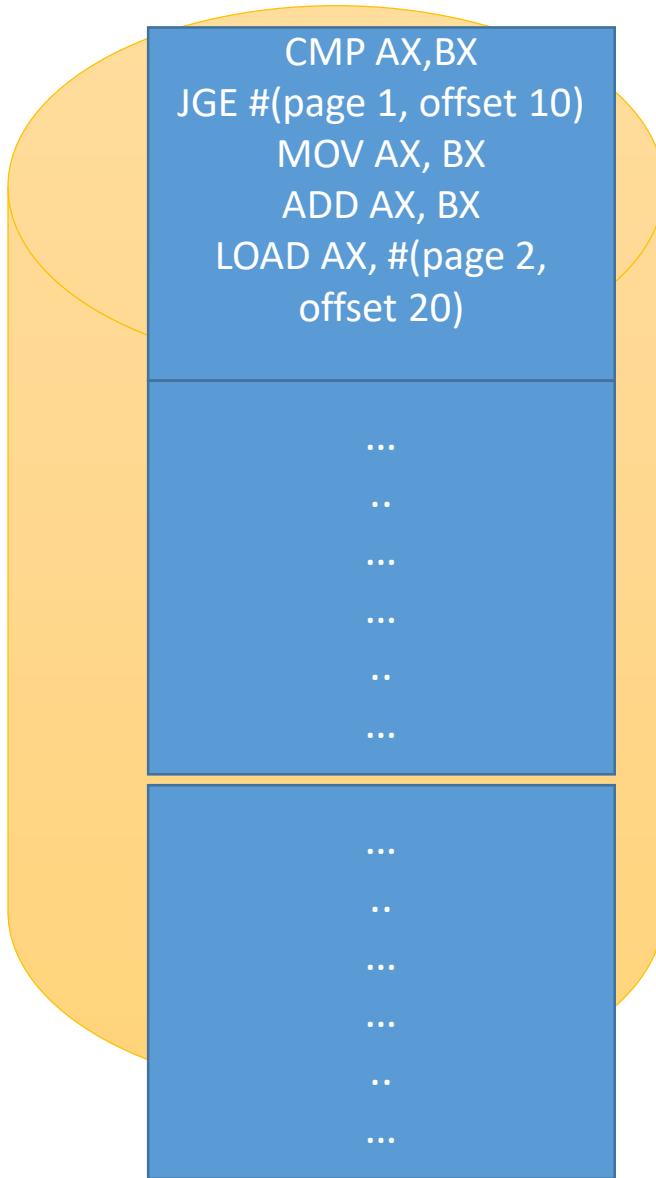
A more efficient approach : Paging



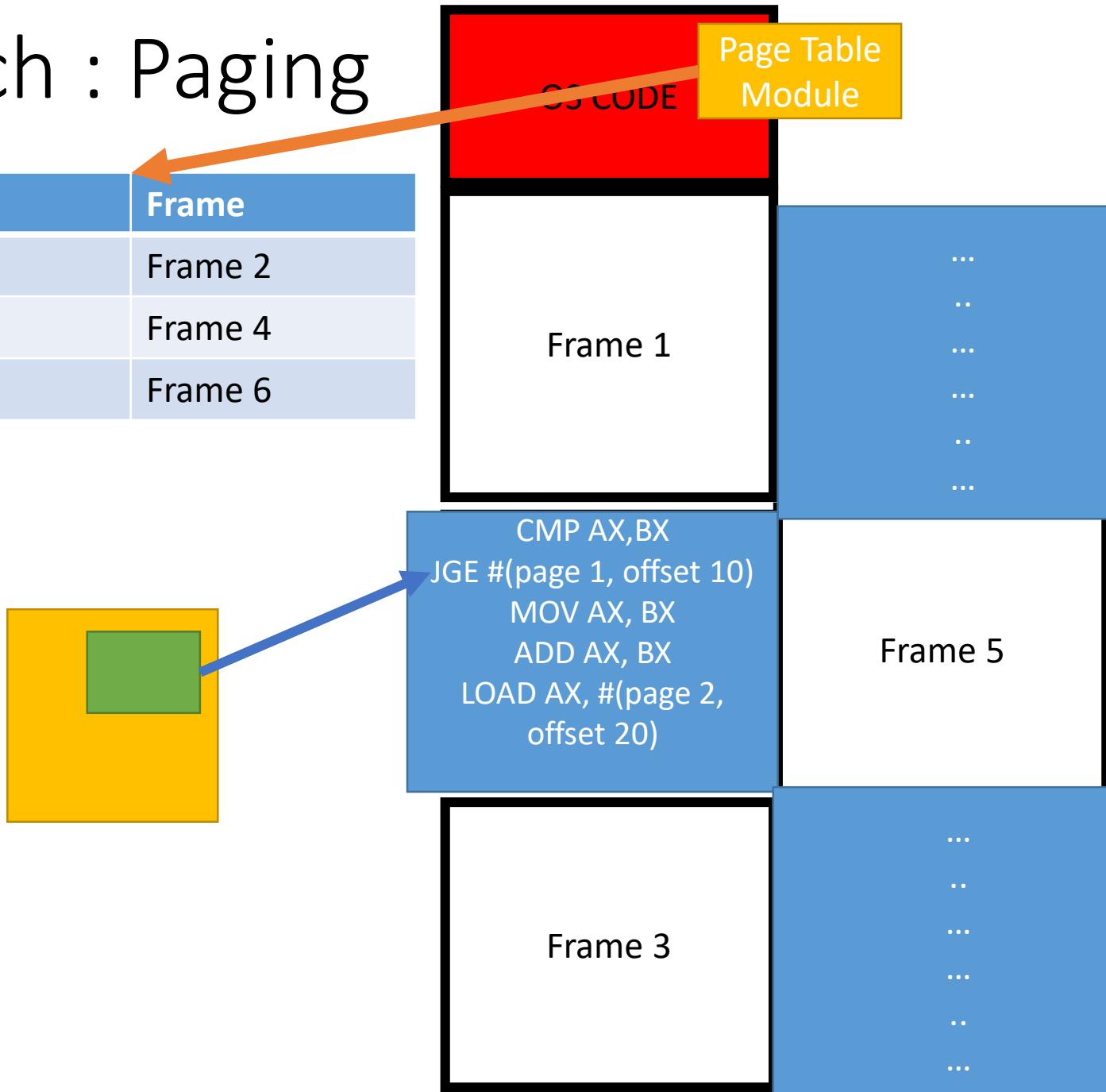
| Page | Frame |
|--------|---------|
| Page 1 | Frame 2 |
| Page 2 | Frame 4 |
| Page 3 | Frame 6 |



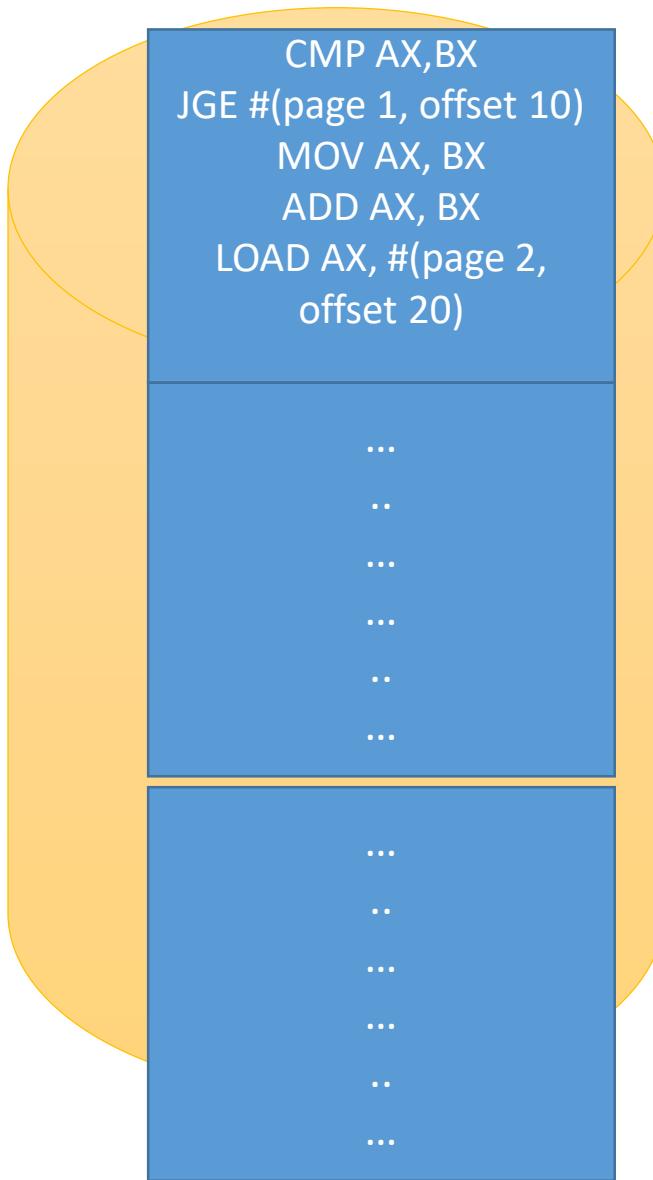
A more efficient approach : Paging



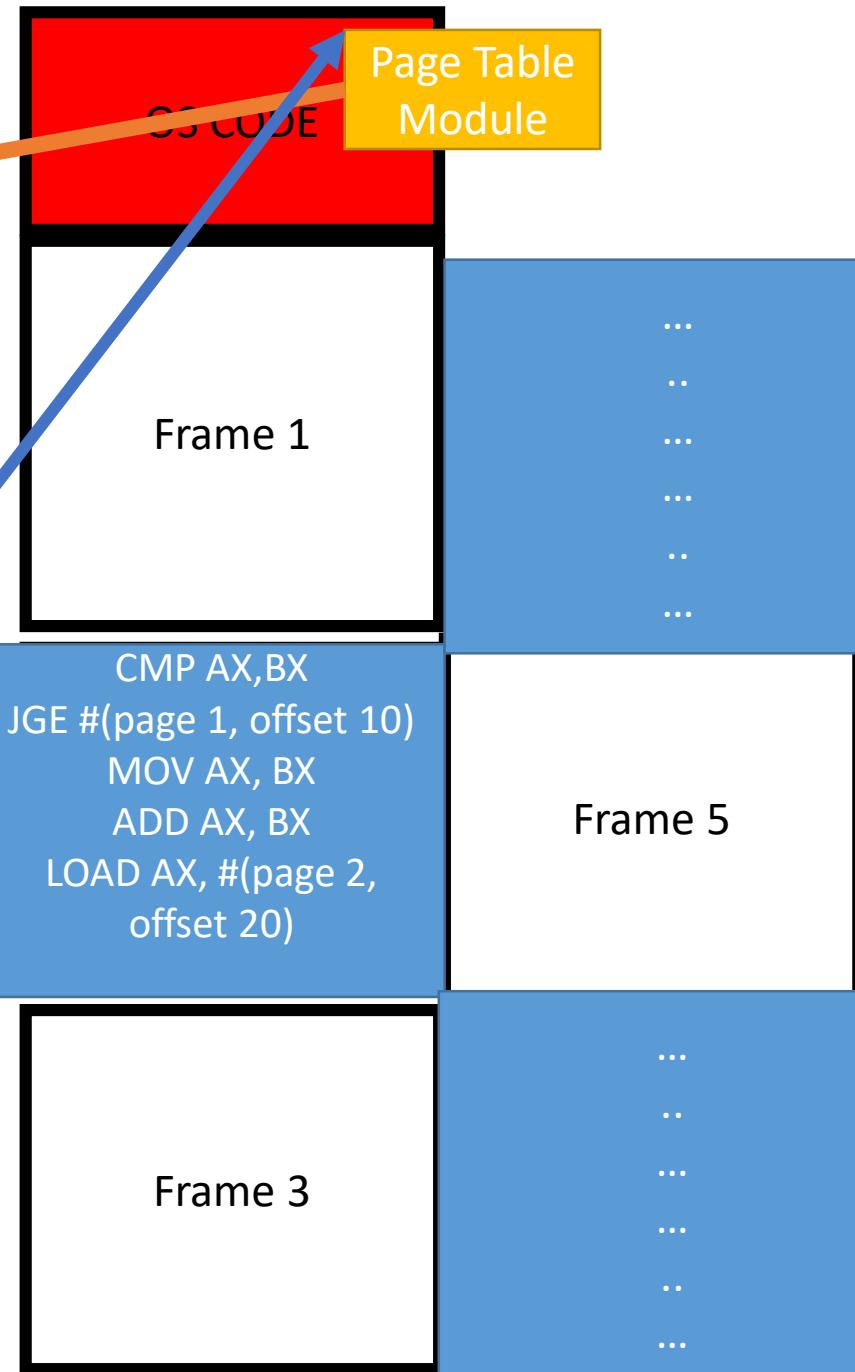
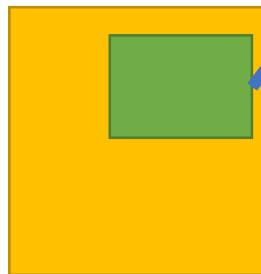
| Page | Frame |
|--------|---------|
| Page 1 | Frame 2 |
| Page 2 | Frame 4 |
| Page 3 | Frame 6 |



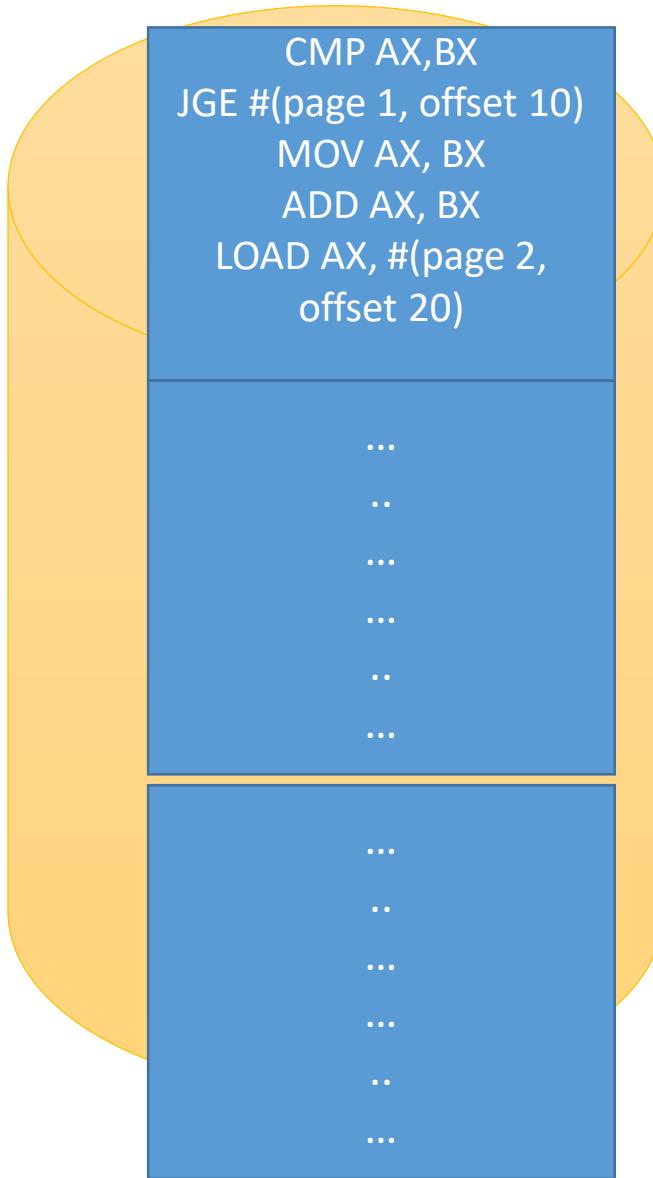
A more efficient approach : Paging



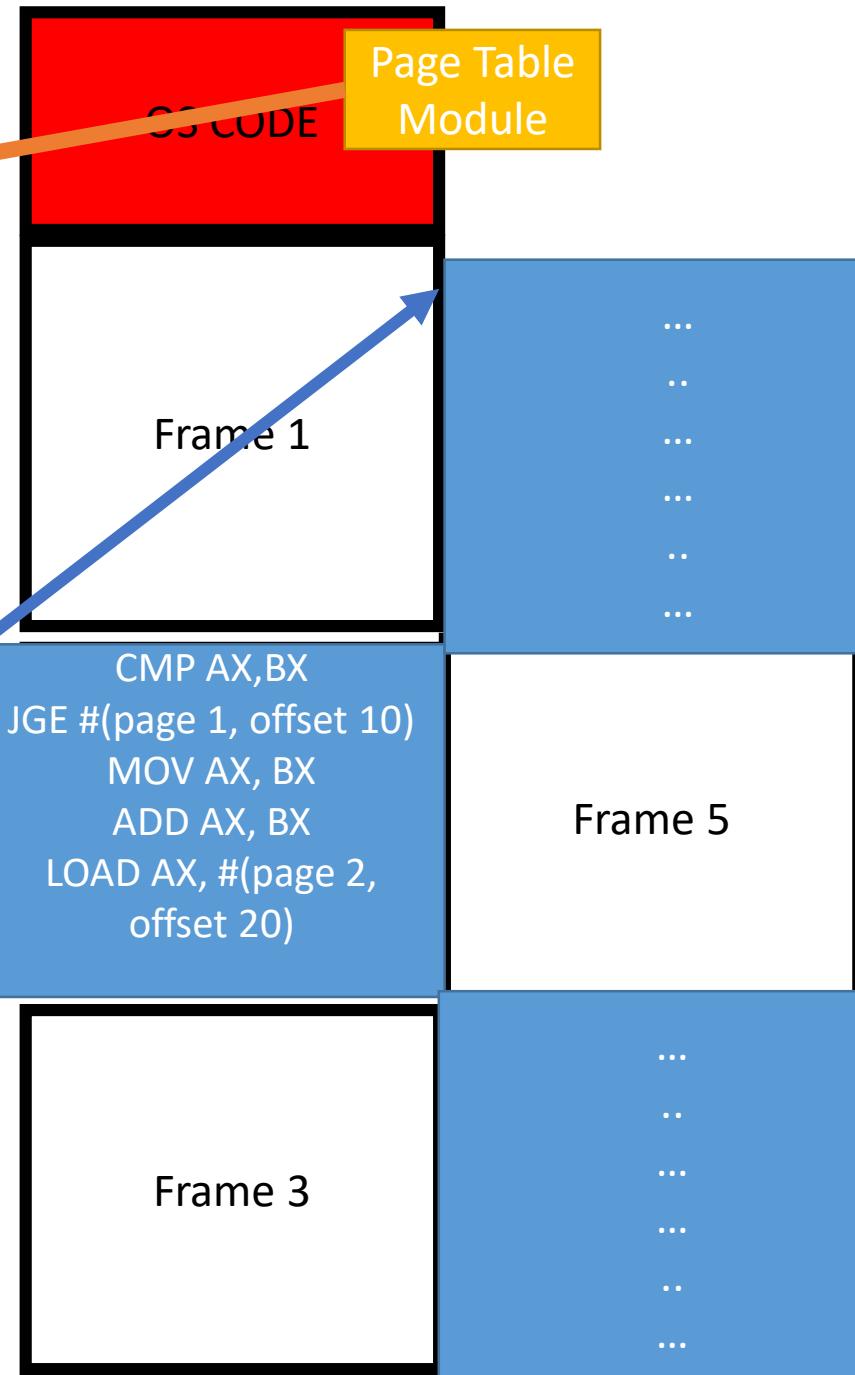
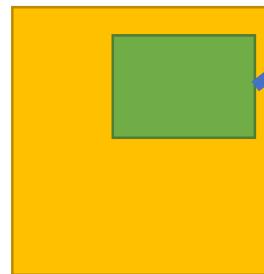
| Page | Frame |
|--------|---------|
| Page 1 | Frame 2 |
| Page 2 | Frame 4 |
| Page 3 | Frame 6 |



A more efficient approach : Paging



| Page | Frame |
|--------|---------|
| Page 1 | Frame 2 |
| Page 2 | Frame 4 |
| Page 3 | Frame 6 |



Demand Paging

- It is not wise to bring in **all the pages of a program** (as we did with the 3 pages of the previous program)
- We should bring in a program's page only when it is **required**.
- **Principle of locality** : A program's execution is confined to some particular pages (because of loops, functions, conditions).
- That is the **execution time of few pages** are way more compared to the **other pages** of the program

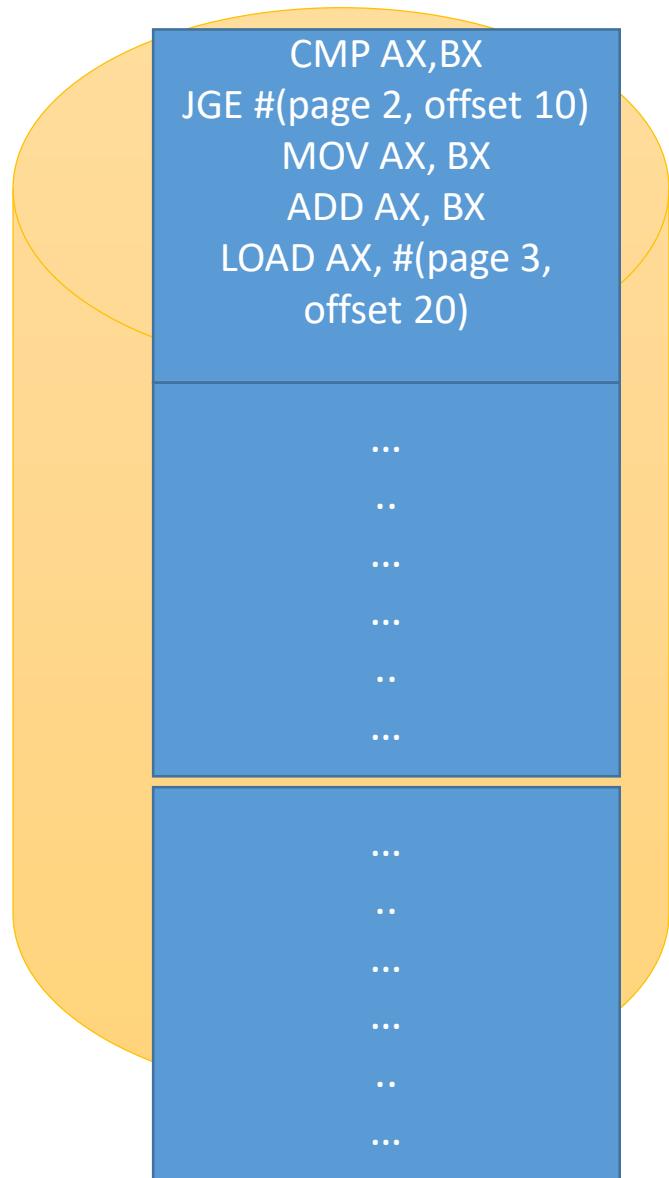
4.1 Demand Paging

Demand Paging

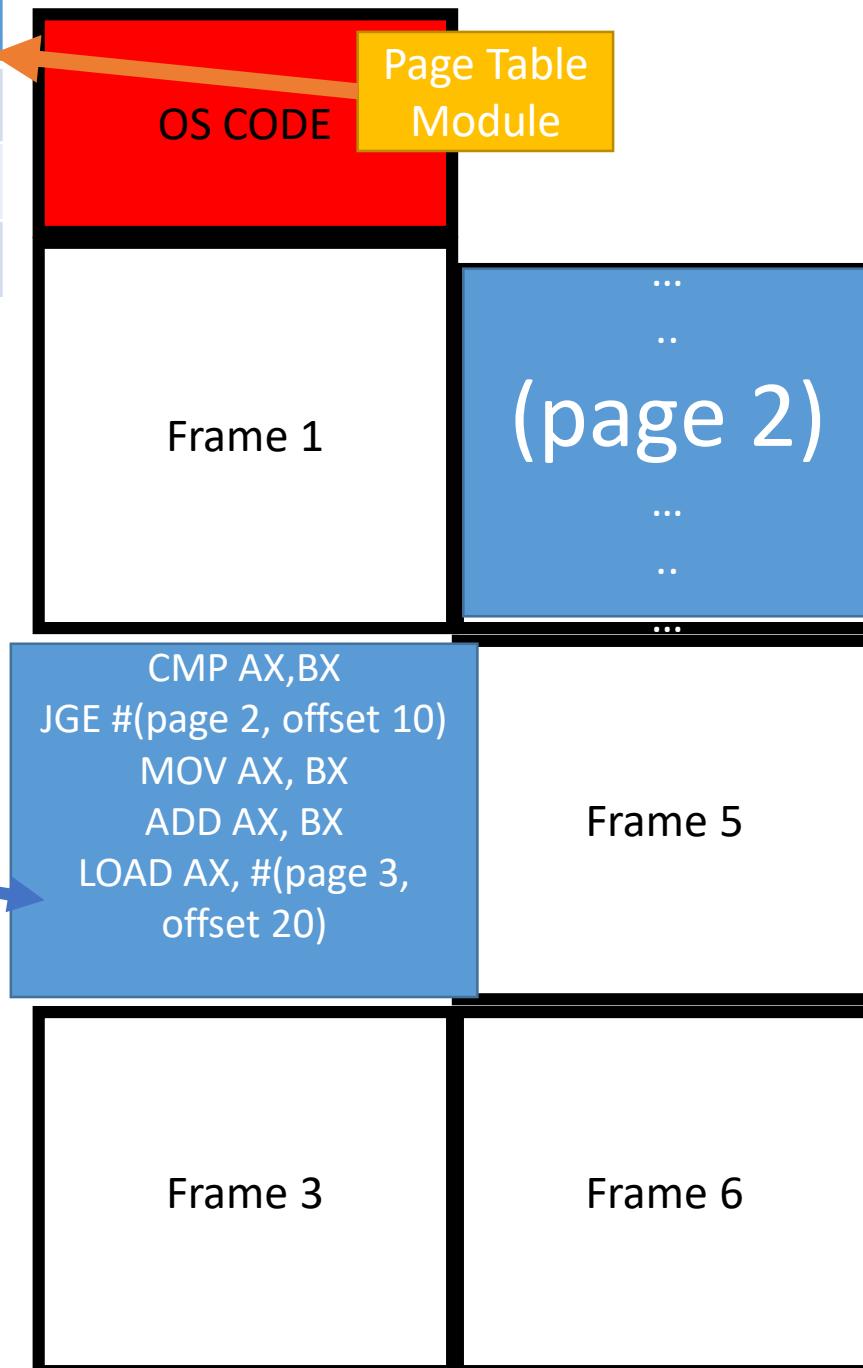
Three terms related to Demand Paging

- Page Fault
- Replacement Algorithm
- Thrashing

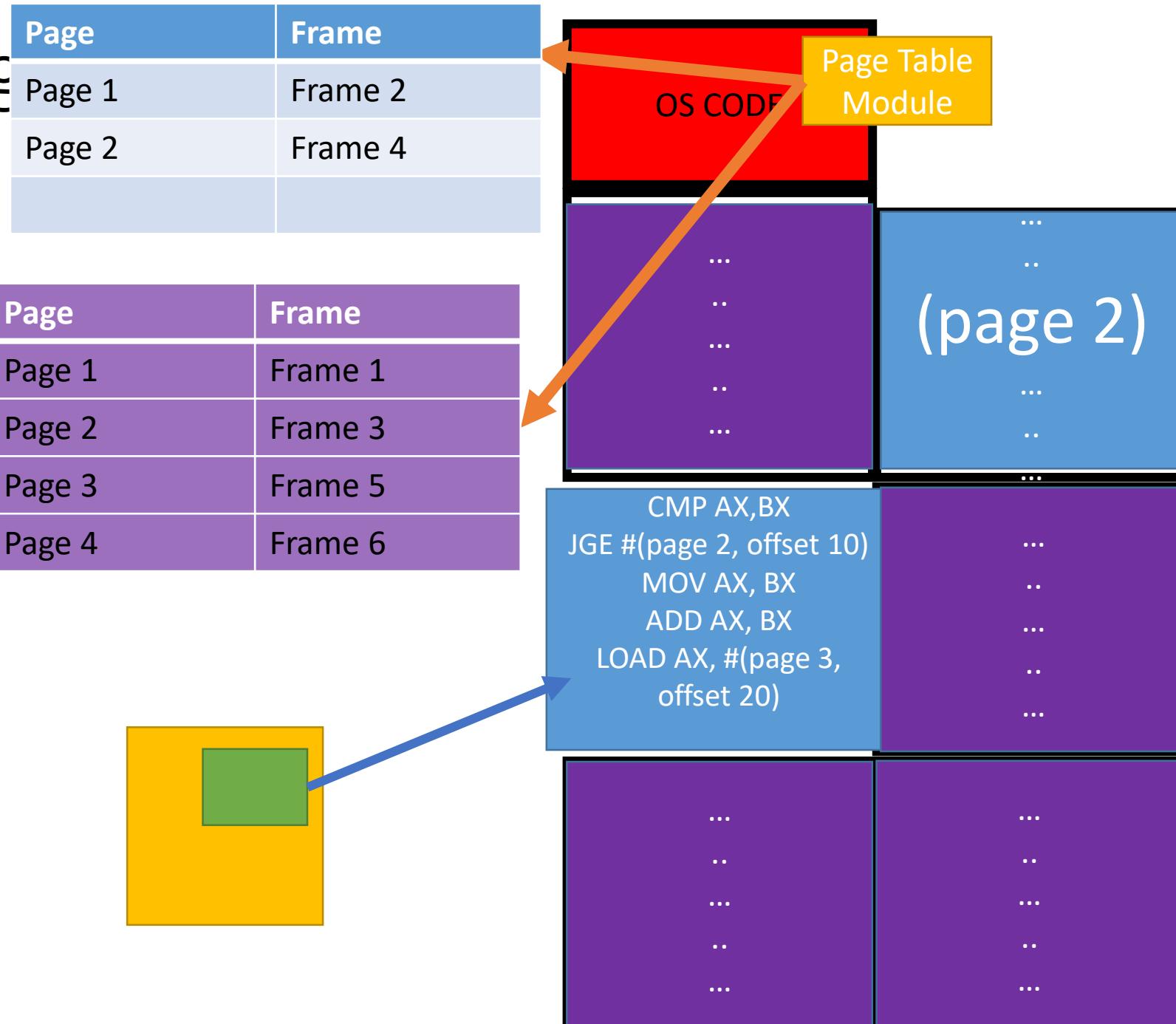
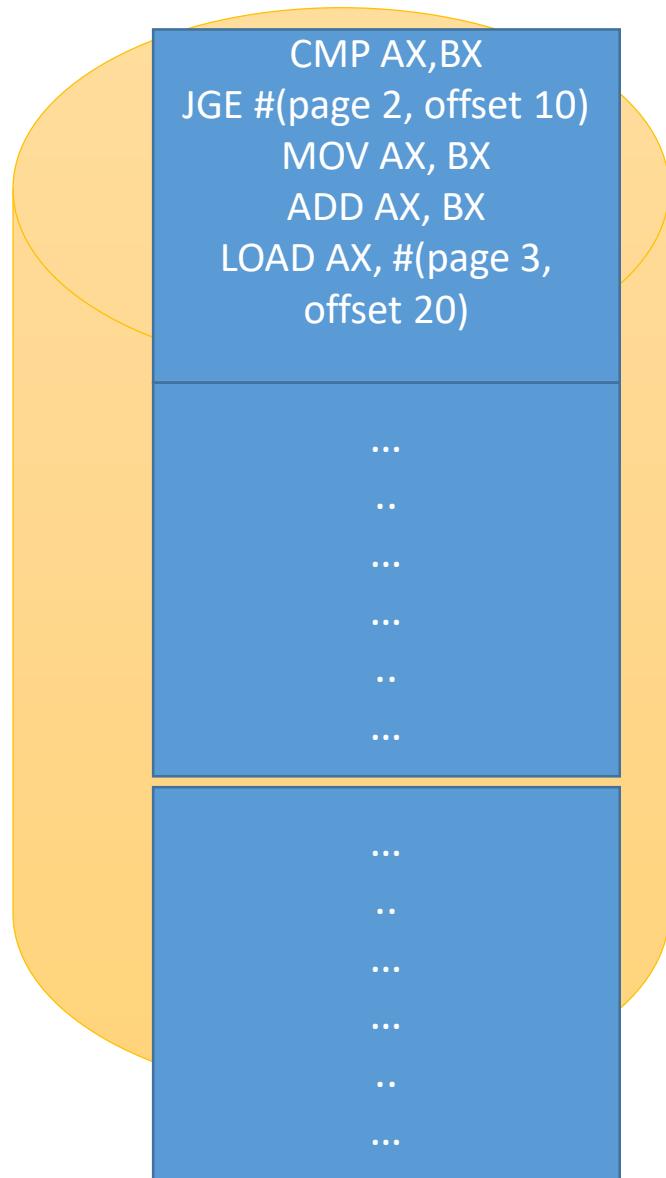
Demand Paging



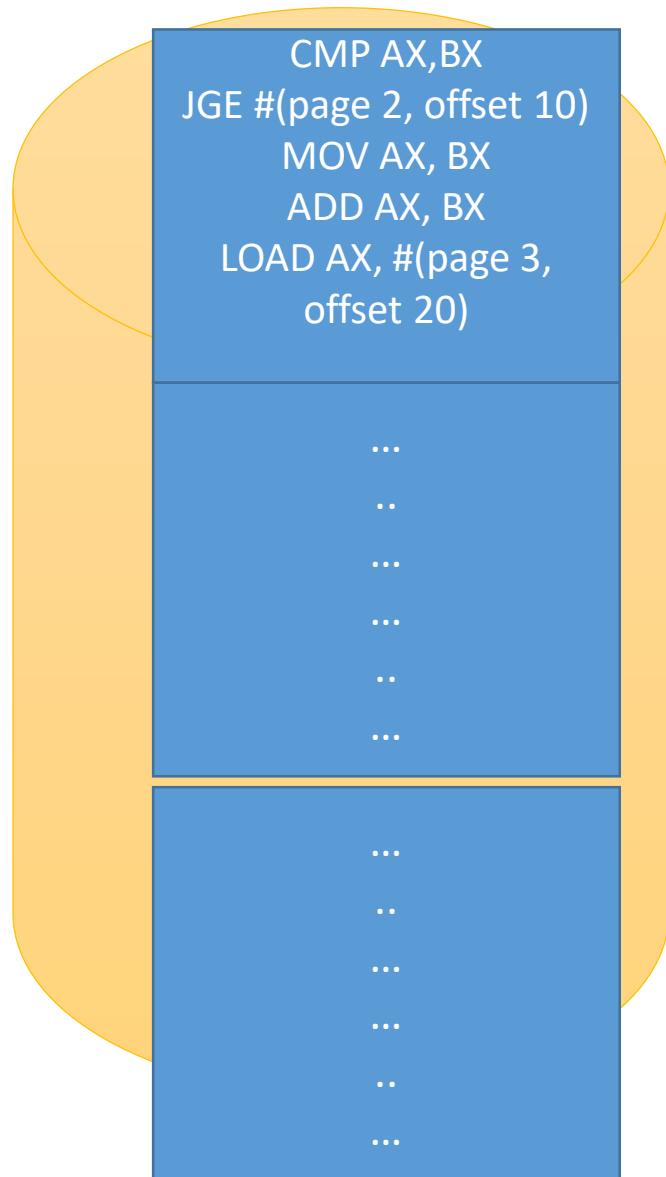
| Page | Frame |
|--------|---------|
| Page 1 | Frame 2 |
| Page 2 | Frame 4 |
| | |



Demand Paging



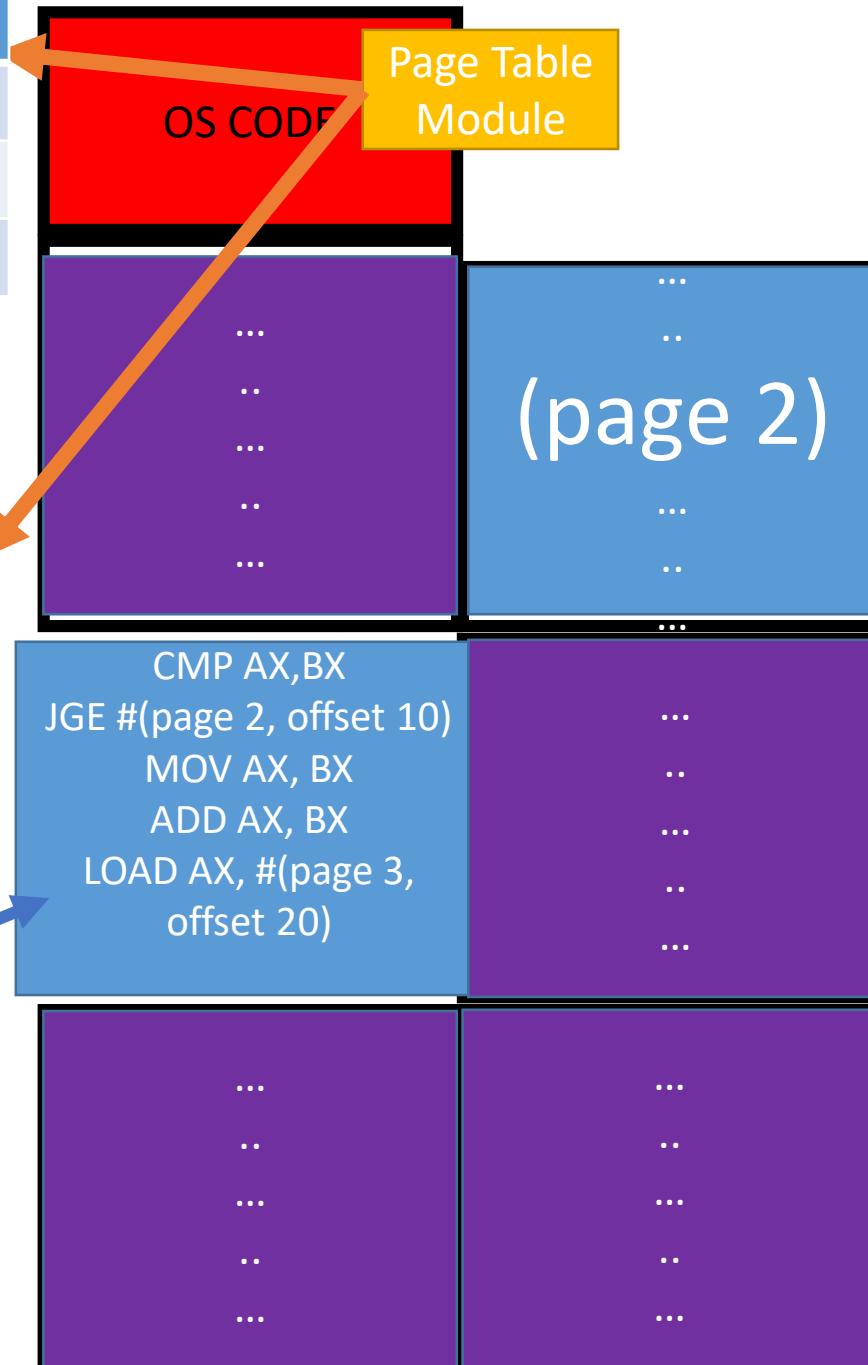
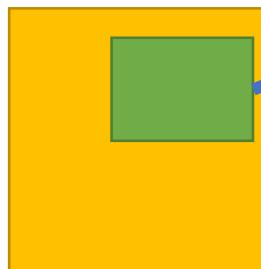
Demand Paging



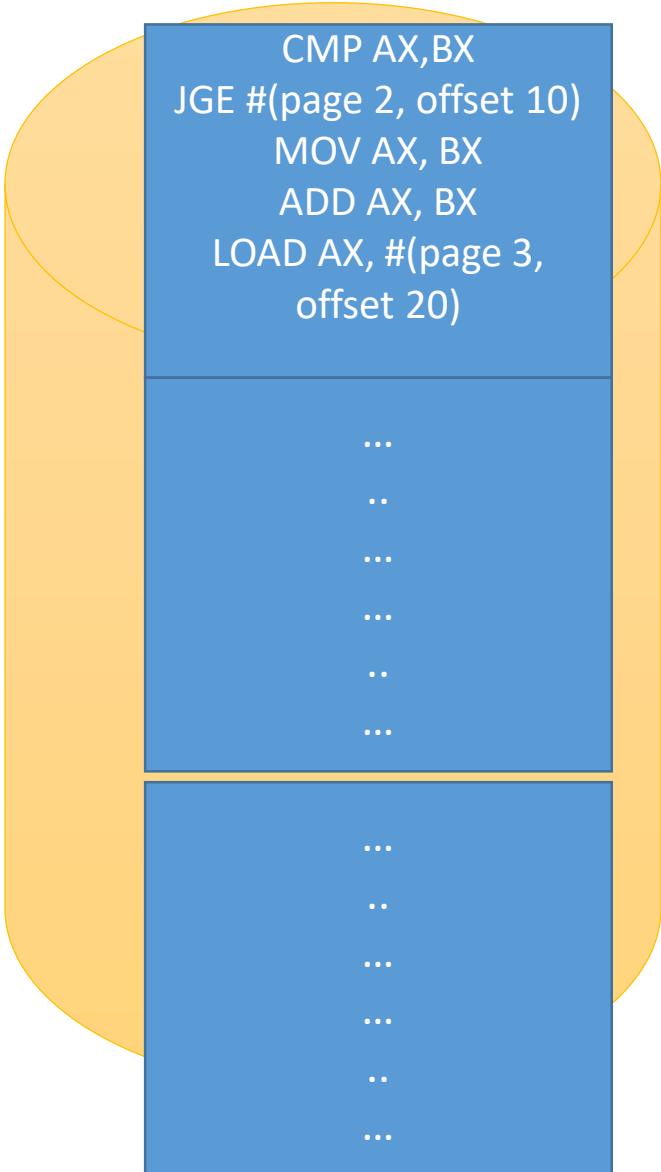
| Page | Frame |
|--------|---------|
| Page 1 | Frame 2 |
| Page 2 | Frame 4 |
| | |

| Page | Frame |
|--------|---------|
| Page 1 | Frame 1 |
| Page 2 | Frame 3 |
| Page 3 | Frame 5 |
| Page 4 | Frame 6 |

Observe that this instruction is
Requiring data from a page that is not
In the memory (**page 3**)



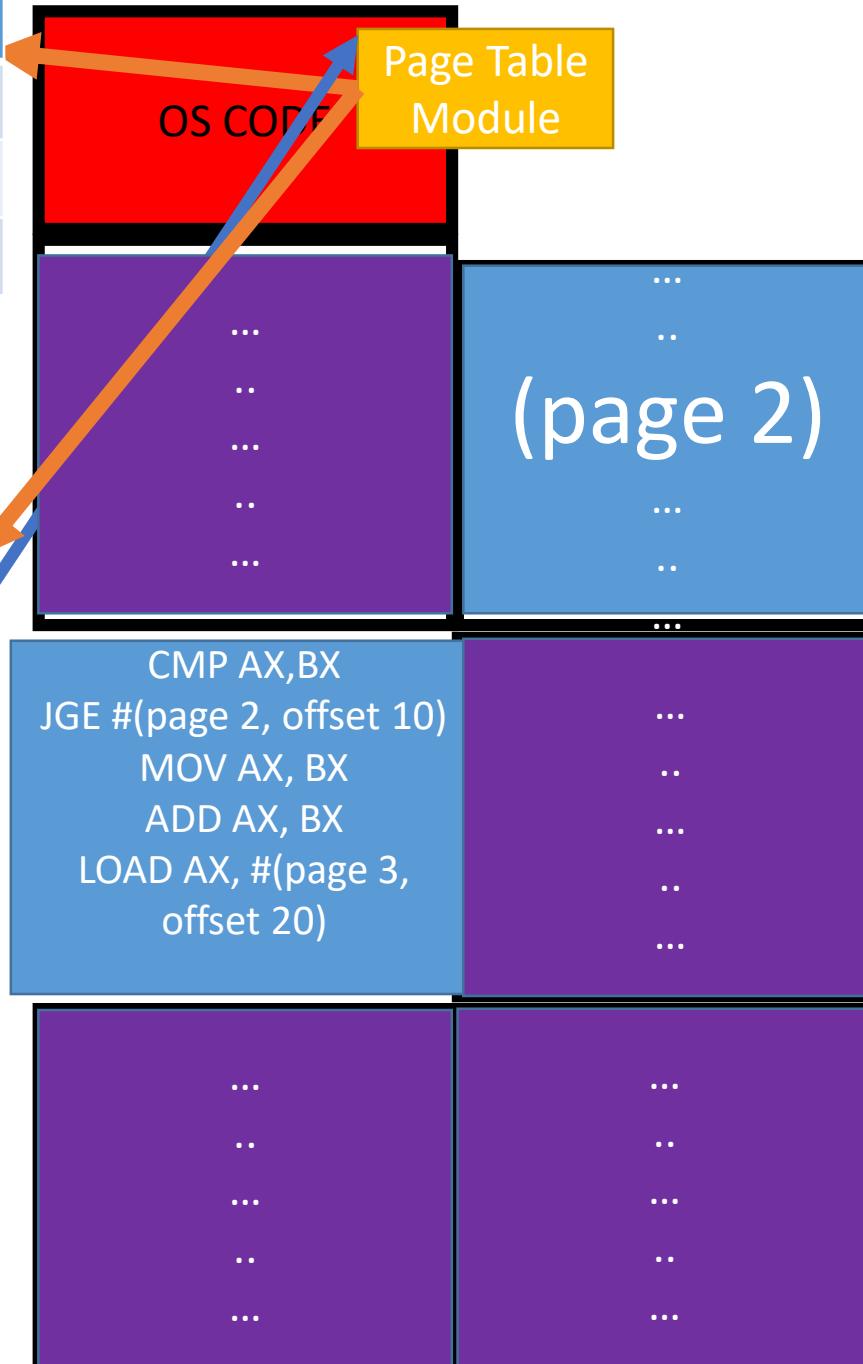
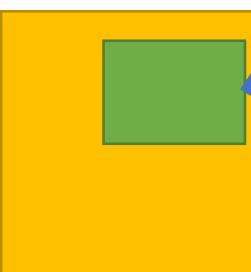
Demand Paging



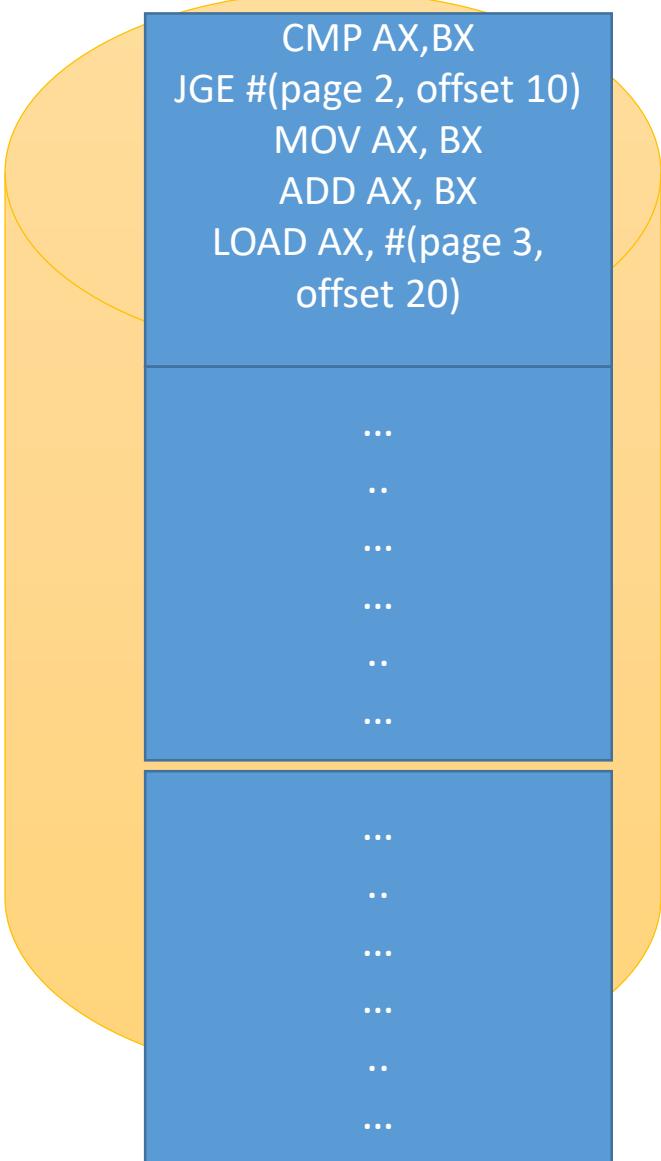
| Page | Frame |
|--------|---------|
| Page 1 | Frame 2 |
| Page 2 | Frame 4 |
| | |

| Page | Frame |
|--------|---------|
| Page 1 | Frame 1 |
| Page 2 | Frame 3 |
| Page 3 | Frame 5 |
| Page 4 | Frame 6 |

The Processor goes to the OS's page
Page table module



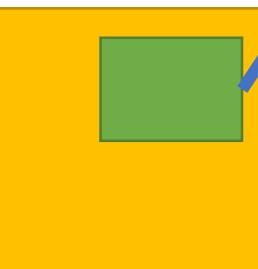
Demand Paging



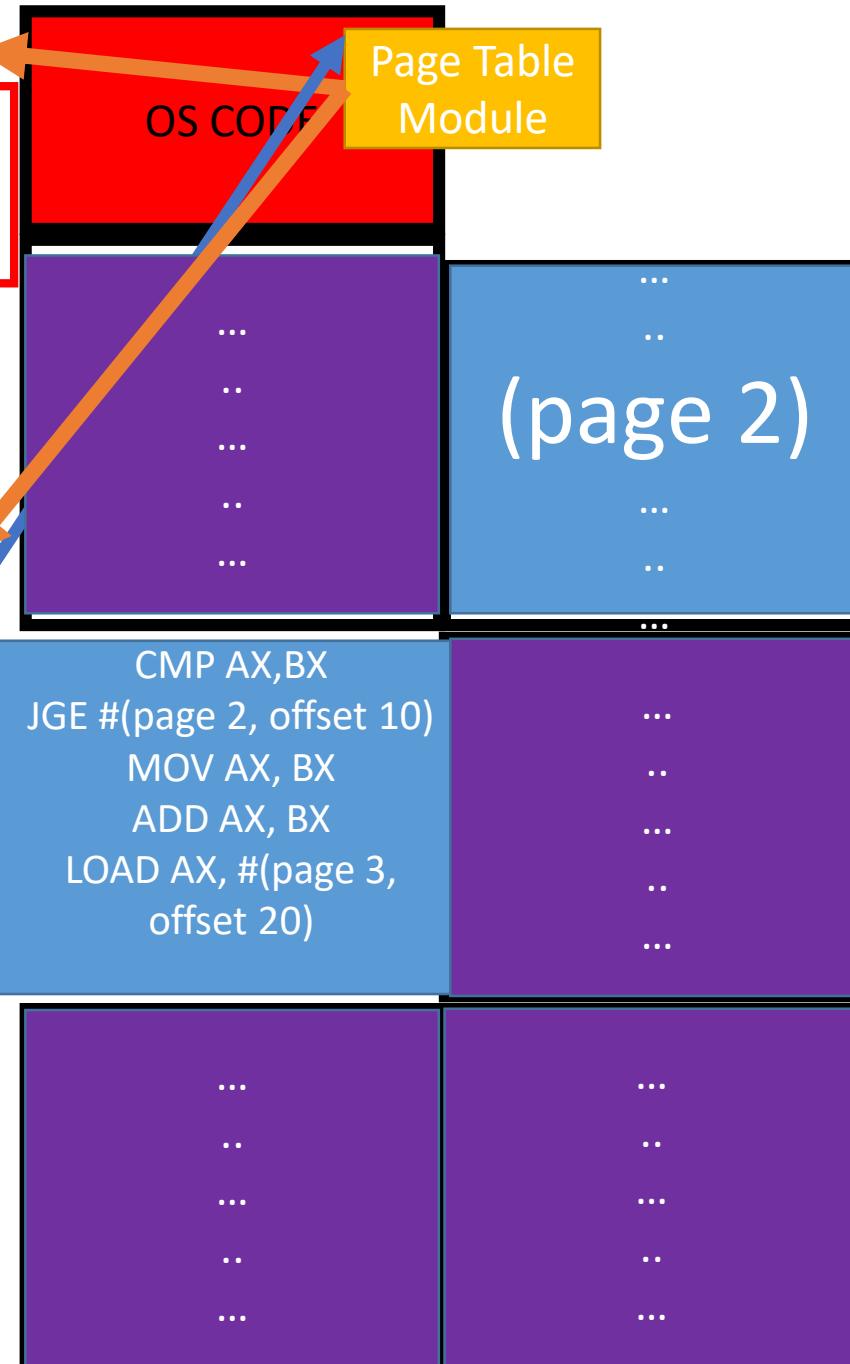
| Page | Frame |
|--------|---------|
| Page 1 | Frame 2 |
| Page 2 | Frame 4 |

| Page | Frame |
|--------|---------|
| Page 1 | Frame 1 |
| Page 2 | Frame 3 |
| Page 3 | Frame 5 |
| Page 4 | Frame 6 |

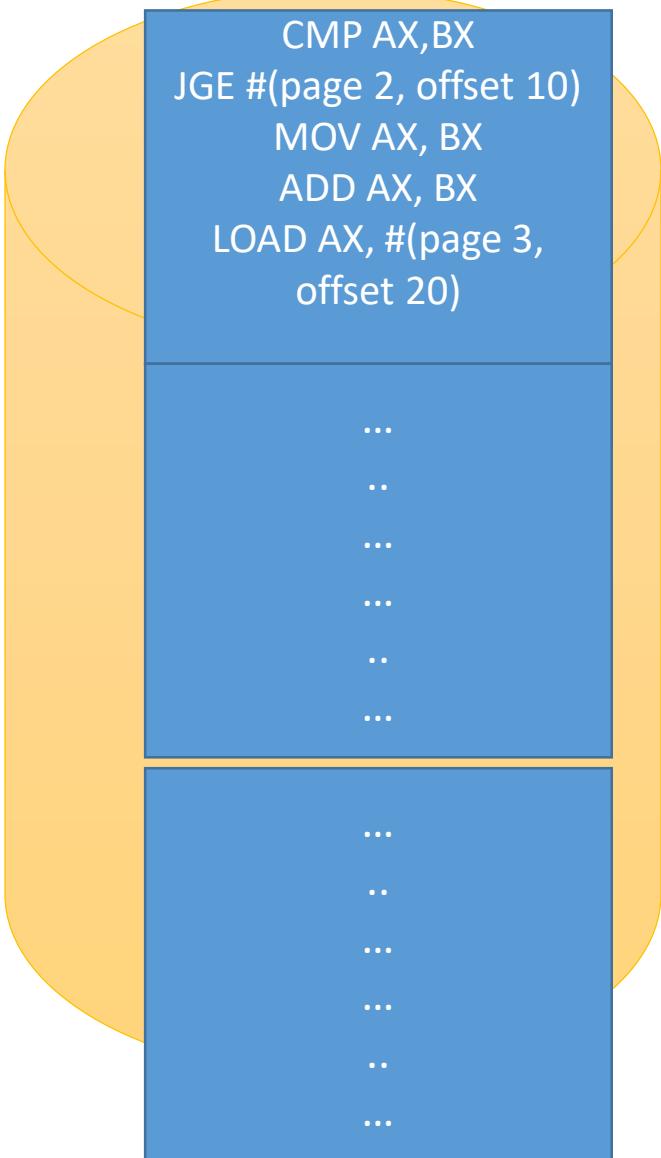
The Processor goes to the OS's page
Page table module



But no data of page 3 is there:
This situation is called **PAGE FAULT**



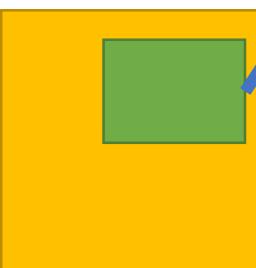
Demand Paging



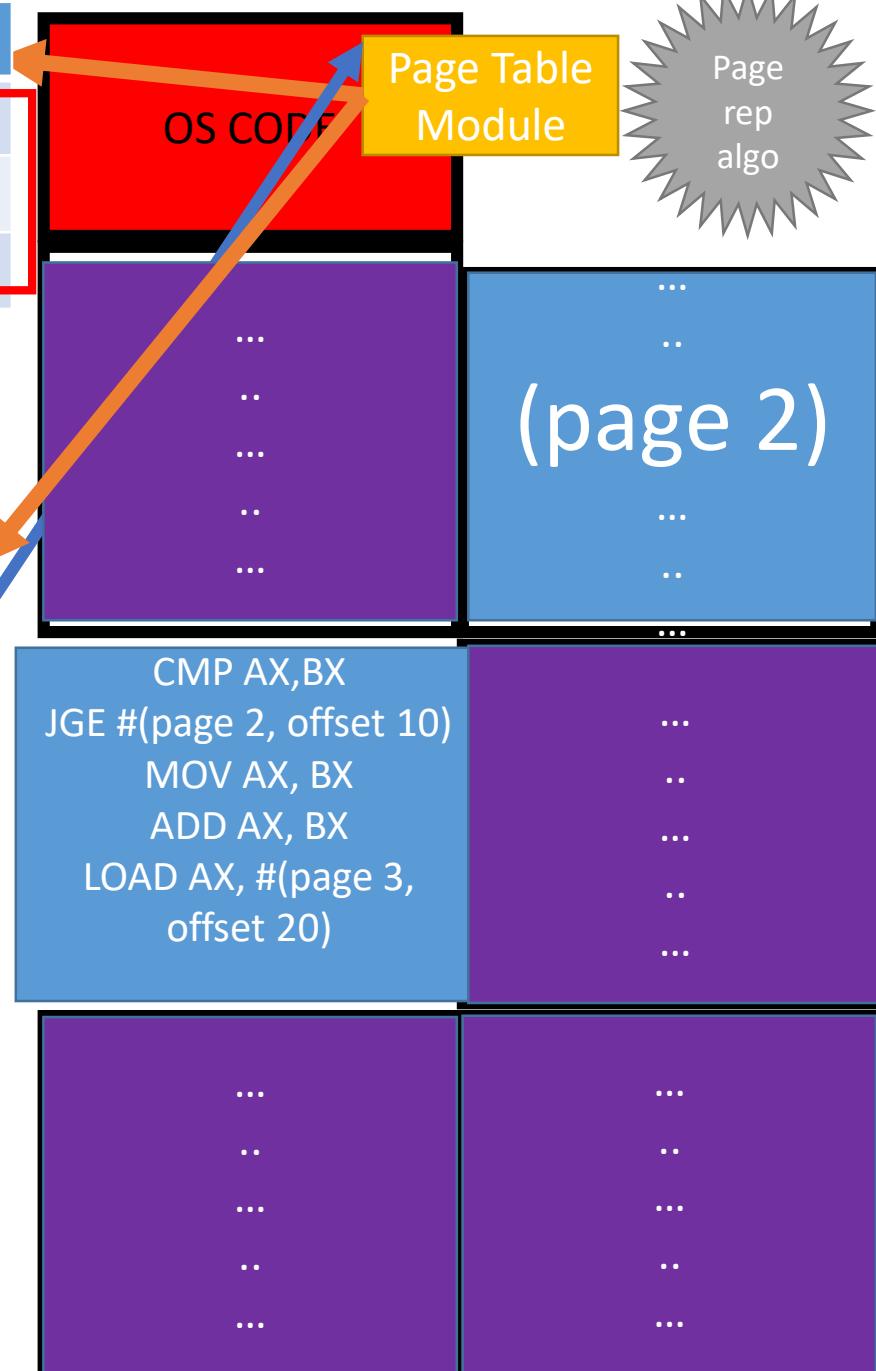
| Page | Frame |
|--------|---------|
| Page 1 | Frame 2 |
| Page 2 | Frame 4 |

| Page | Frame |
|--------|---------|
| Page 1 | Frame 1 |
| Page 2 | Frame 3 |
| Page 3 | Frame 5 |
| Page 4 | Frame 6 |

The Processor goes to the OS's page
Page table module

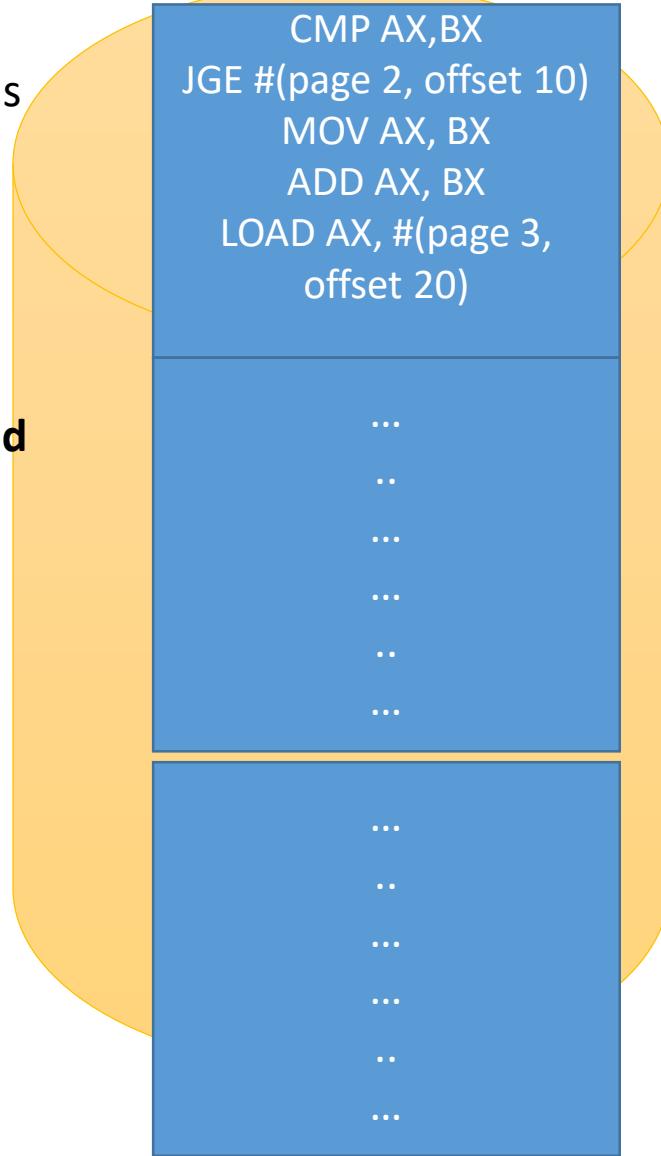


But no data of page 3 is there



Demand Paging

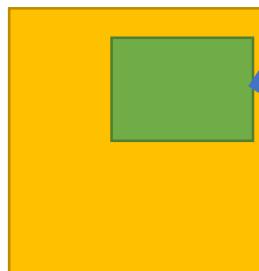
Since, no empty Frame is there, one of the Pages will be replaced



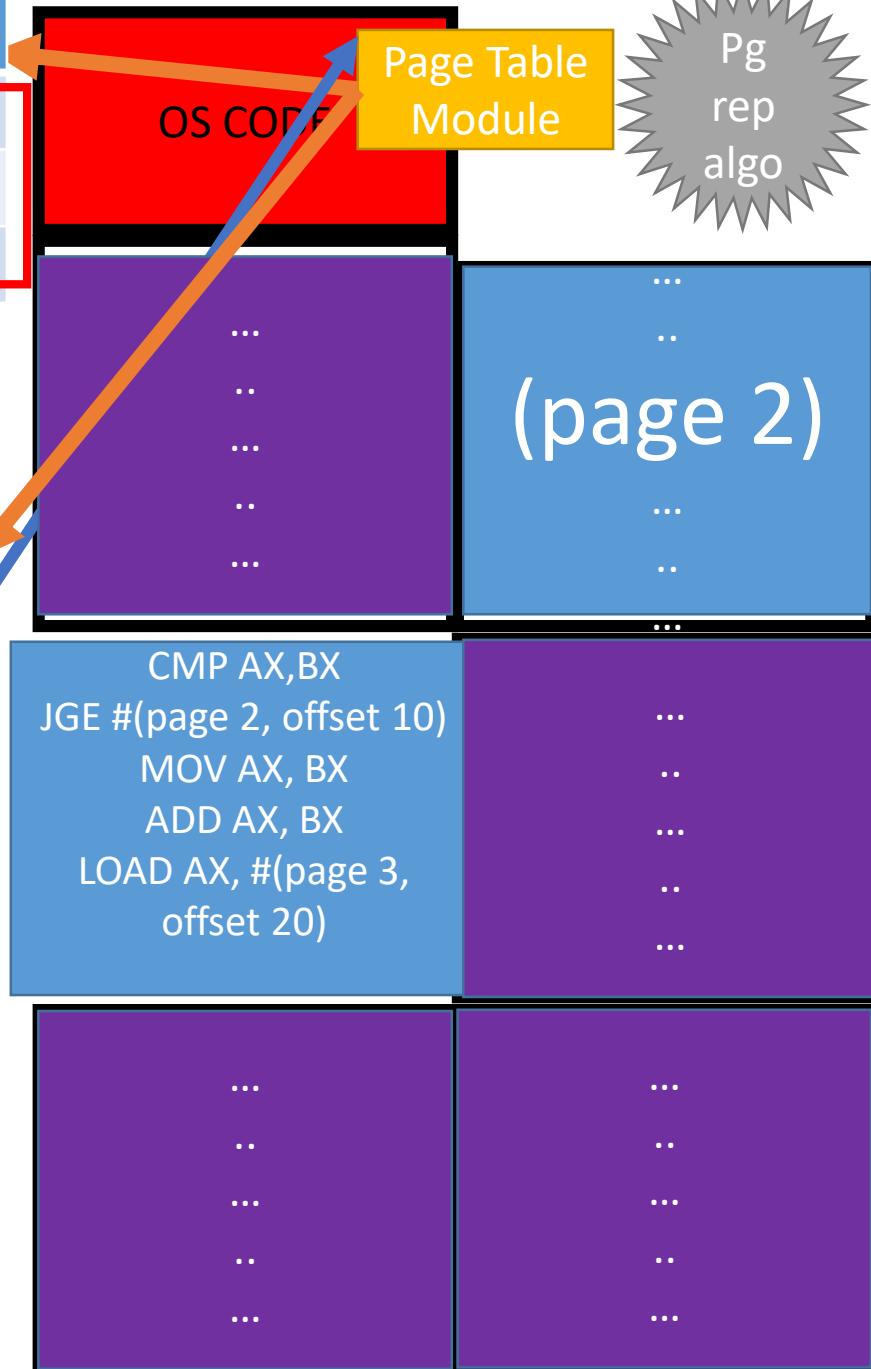
| Page | Frame |
|--------|---------|
| Page 1 | Frame 2 |
| Page 2 | Frame 4 |

| Page | Frame |
|--------|---------|
| Page 1 | Frame 1 |
| Page 2 | Frame 3 |
| Page 3 | Frame 5 |
| Page 4 | Frame 6 |

The Processor goes to the OS's page Page table module



But no data of page 3 is there



Demand Paging

Since, no empty Frame is there, one of the Pages will be replaced

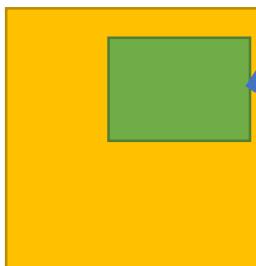


The new Page is Placed inside And the The page Tables are updated

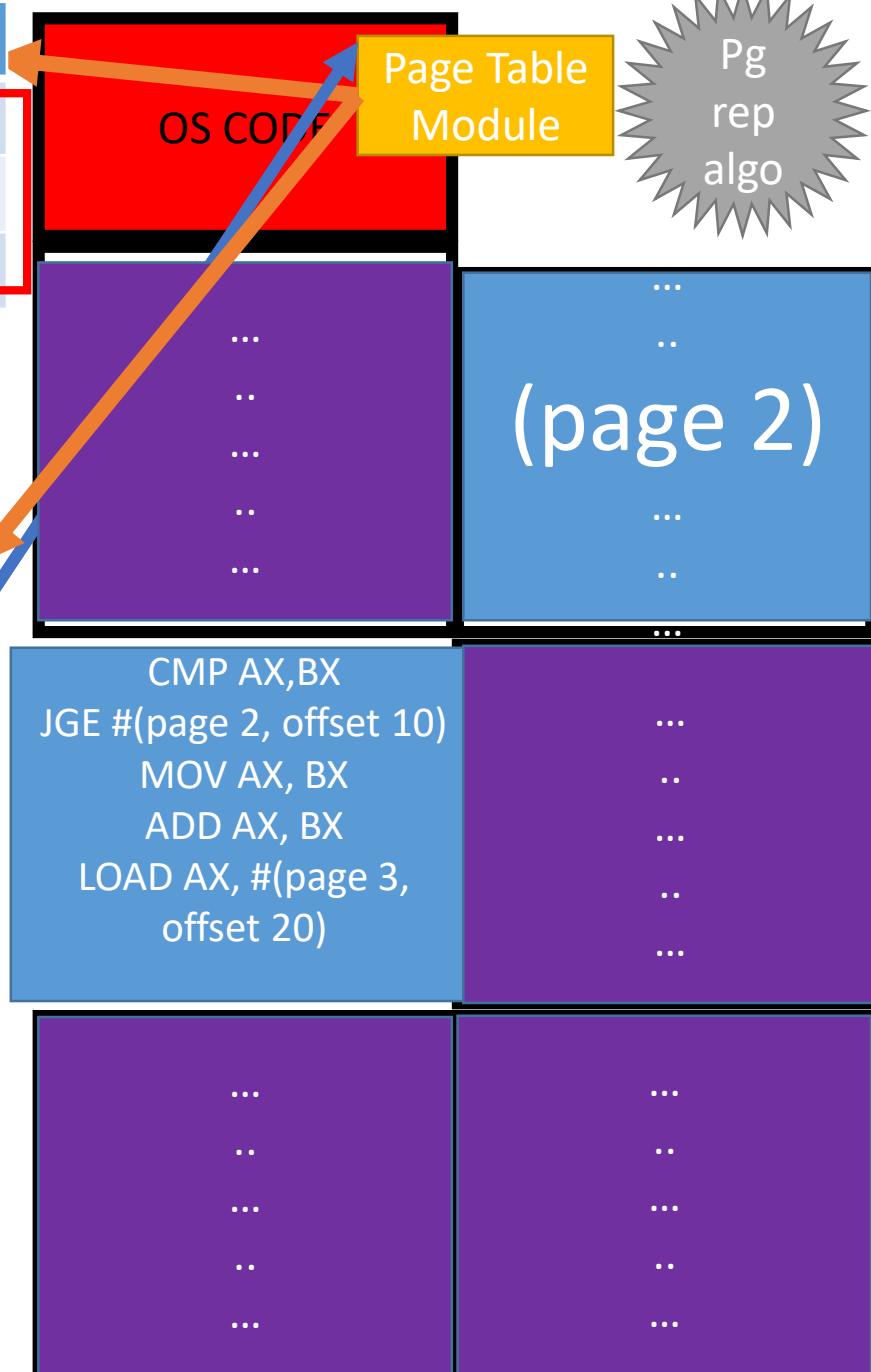
| Page | Frame |
|--------|---------|
| Page 1 | Frame 2 |
| Page 2 | Frame 4 |

| Page | Frame |
|--------|---------|
| Page 1 | Frame 1 |
| Page 2 | Frame 3 |
| Page 3 | Frame 5 |
| Page 4 | Frame 6 |

The Processor goes to the OS's page Page table module



But no data of page 3 is there



Pg rep algo

Demand Paging

Since, no empty Frame is there, one of the Pages will be replaced

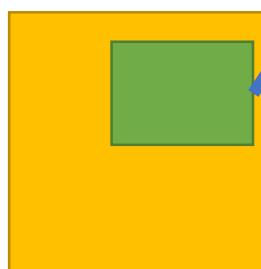


The new Page is Placed inside And the The page Tables are updated

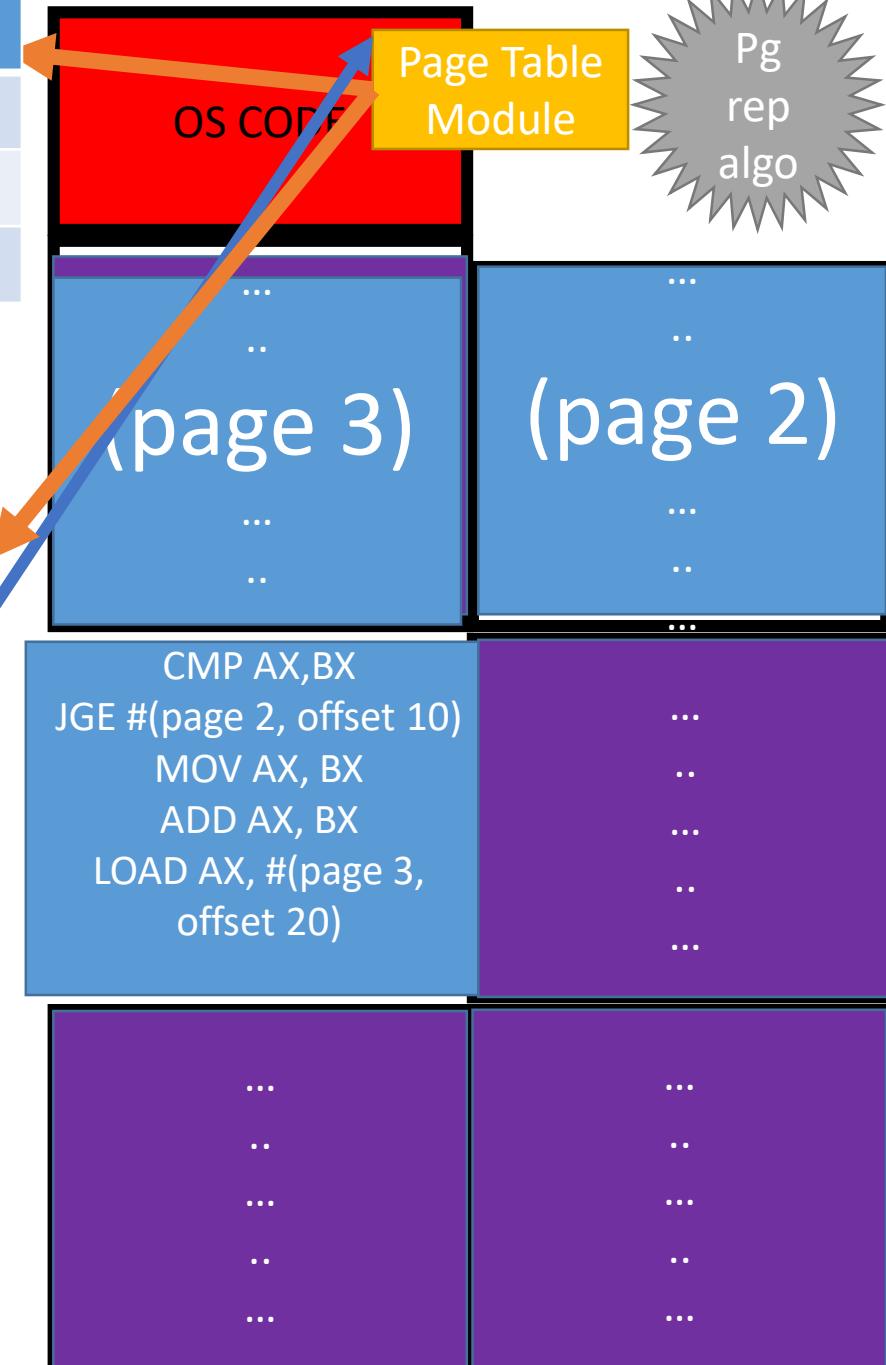
| Page | Frame |
|--------|---------|
| Page 1 | Frame 2 |
| Page 2 | Frame 4 |
| Page 3 | Frame 1 |

| Page | Frame |
|--------|---------|
| Page 2 | Frame 3 |
| Page 3 | Frame 5 |
| Page 4 | Frame 6 |

The Processor goes to the OS's page Page table module



But no data of page 3 is there



Pg rep algo

Demand Paging

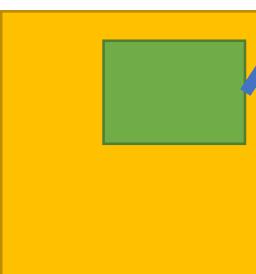
Using
The new
Entry
The
process
Execution
Is
continued
By
updating
The
program
counter



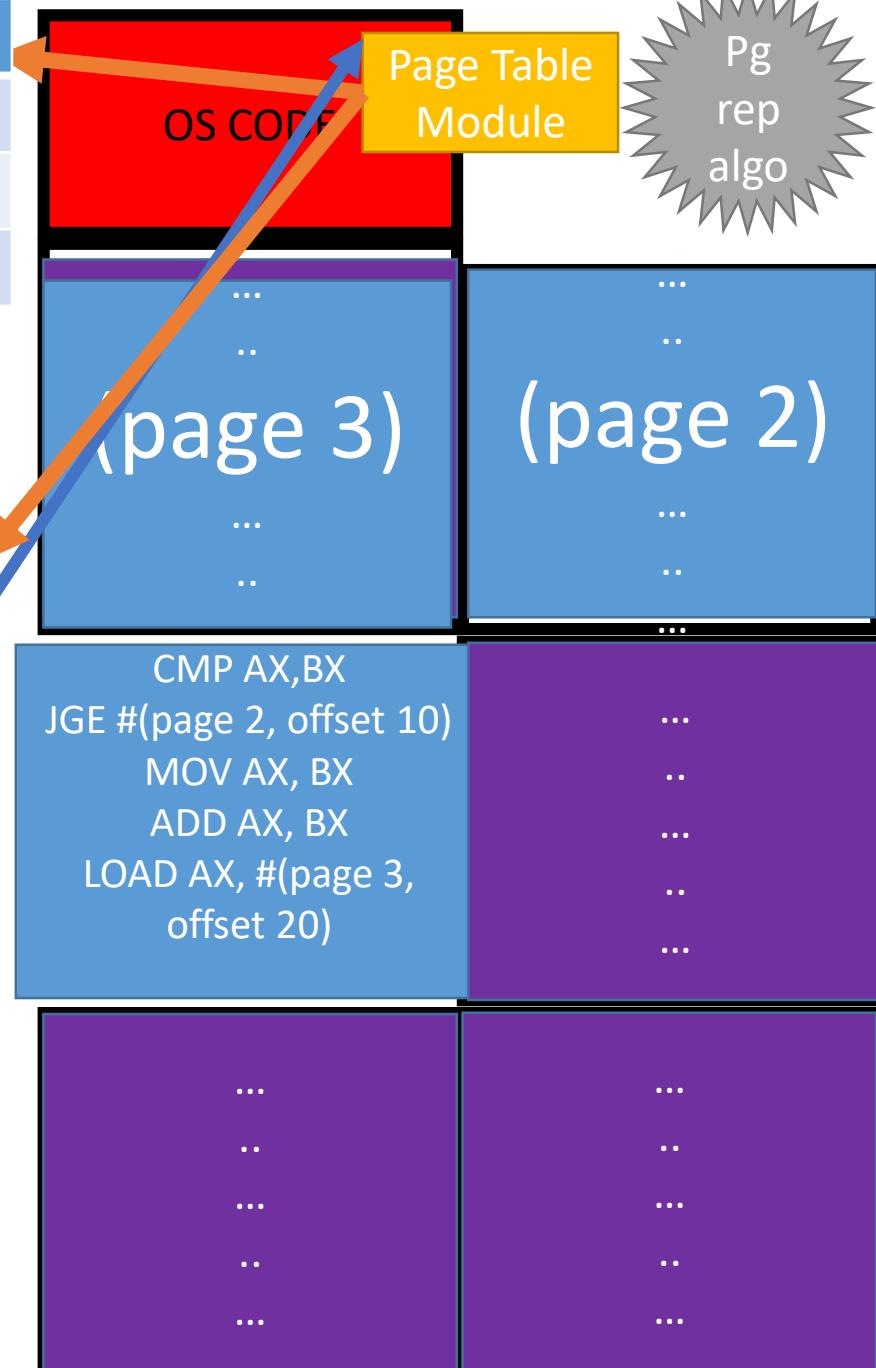
| Page | Frame |
|--------|---------|
| Page 1 | Frame 2 |
| Page 2 | Frame 4 |
| Page 3 | Frame 1 |

| Page | Frame |
|--------|---------|
| Page 2 | Frame 3 |
| Page 3 | Frame 5 |
| Page 4 | Frame 6 |

The Processor goes to the OS's page
Page table module



But no data of page 3 is there



Pg
rep
algo

Demand Paging

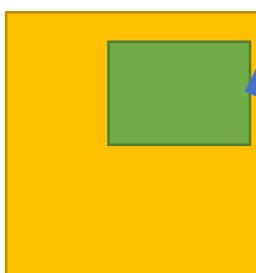
Using
The new
Entry
The
process
Execution
Is
continued
By
updating
The
program
counter



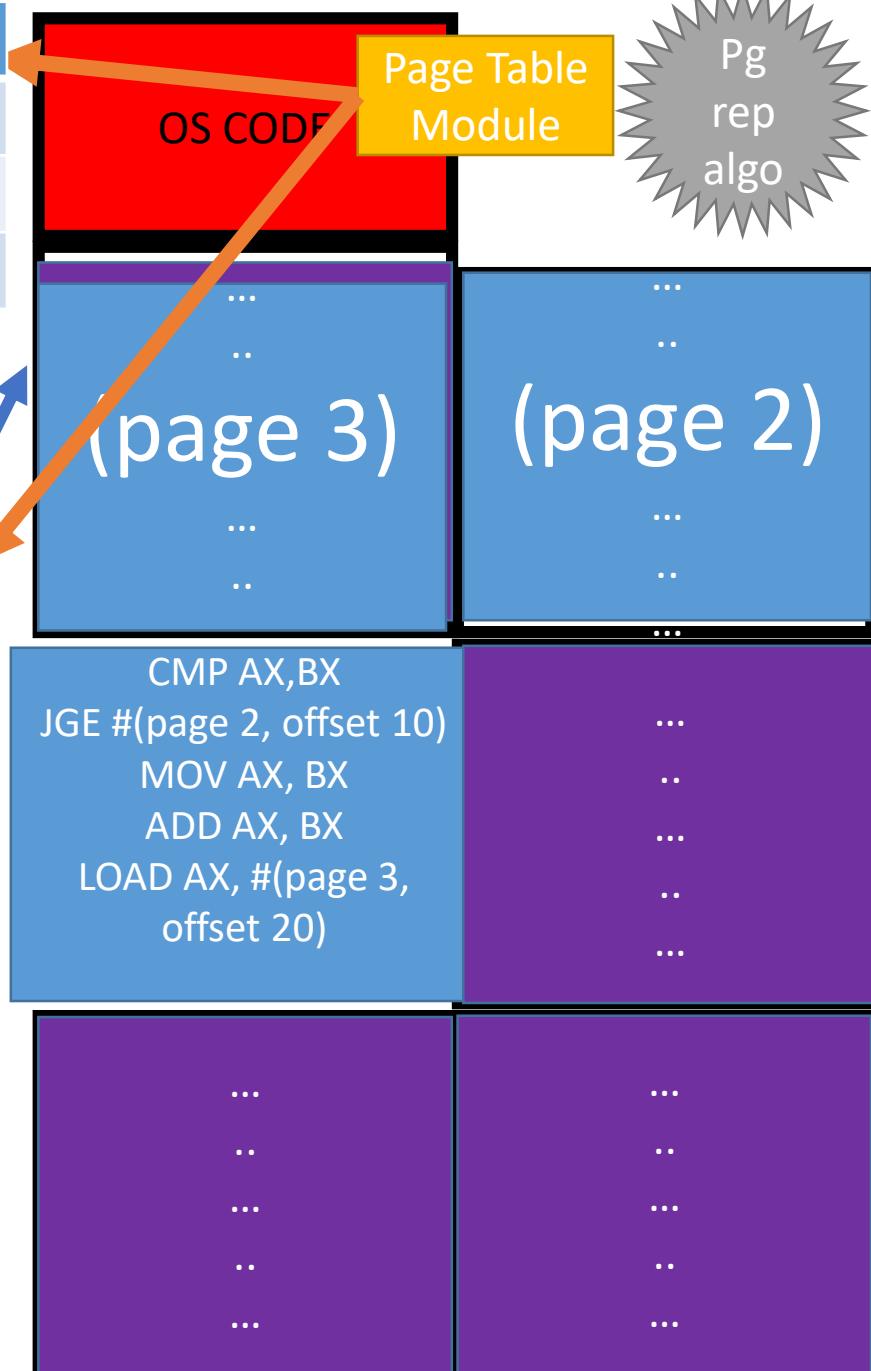
| Page | Frame |
|--------|---------|
| Page 1 | Frame 2 |
| Page 2 | Frame 4 |
| Page 3 | Frame 1 |

| Page | Frame |
|--------|---------|
| Page 2 | Frame 3 |
| Page 3 | Frame 5 |
| Page 4 | Frame 6 |

The Processor goes to the OS's page
Page table module

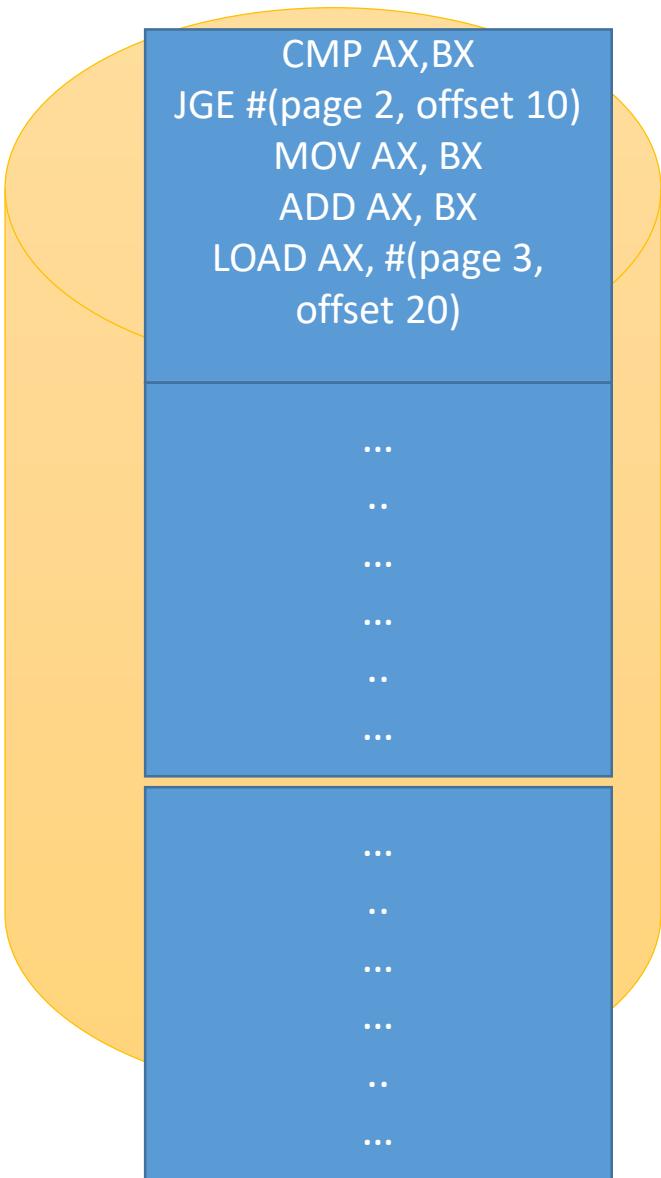


But no data of page 3 is there



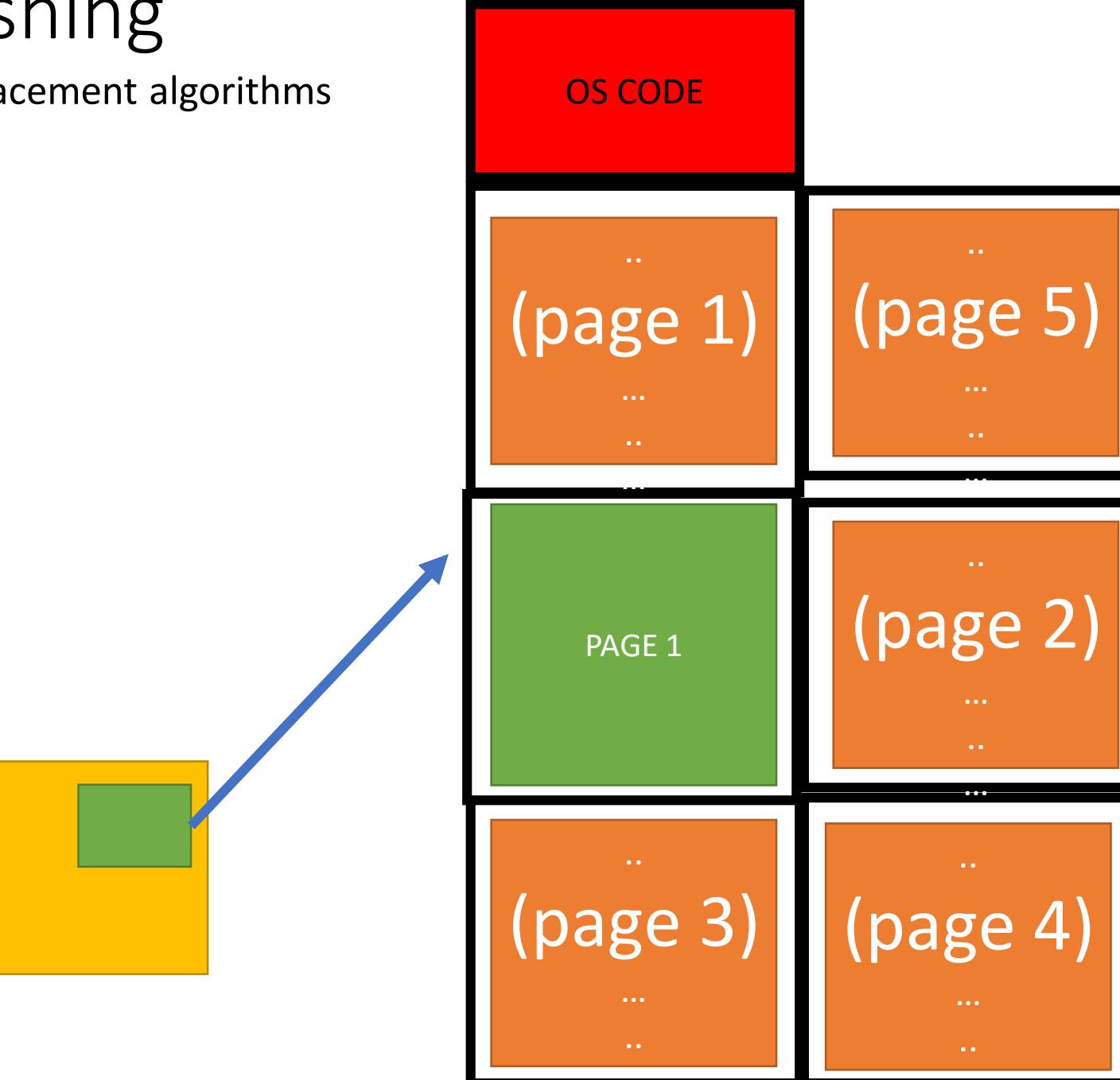
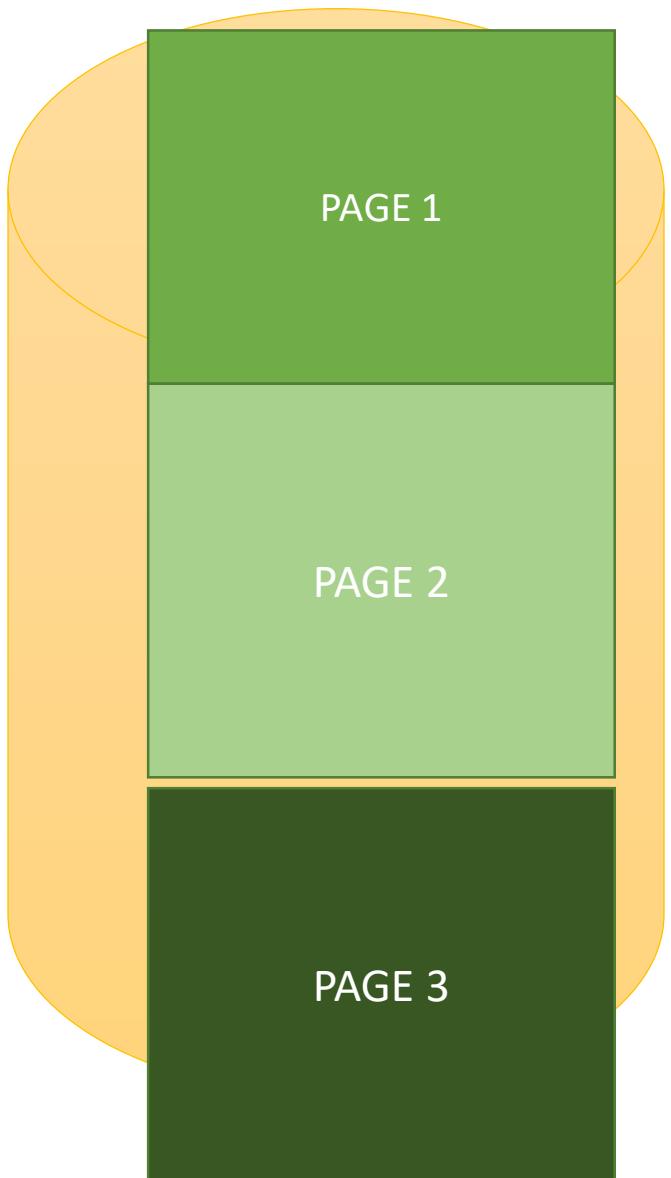
Demand Paging : Thrashing

- This happens due to **inefficient Page replacement algorithms**



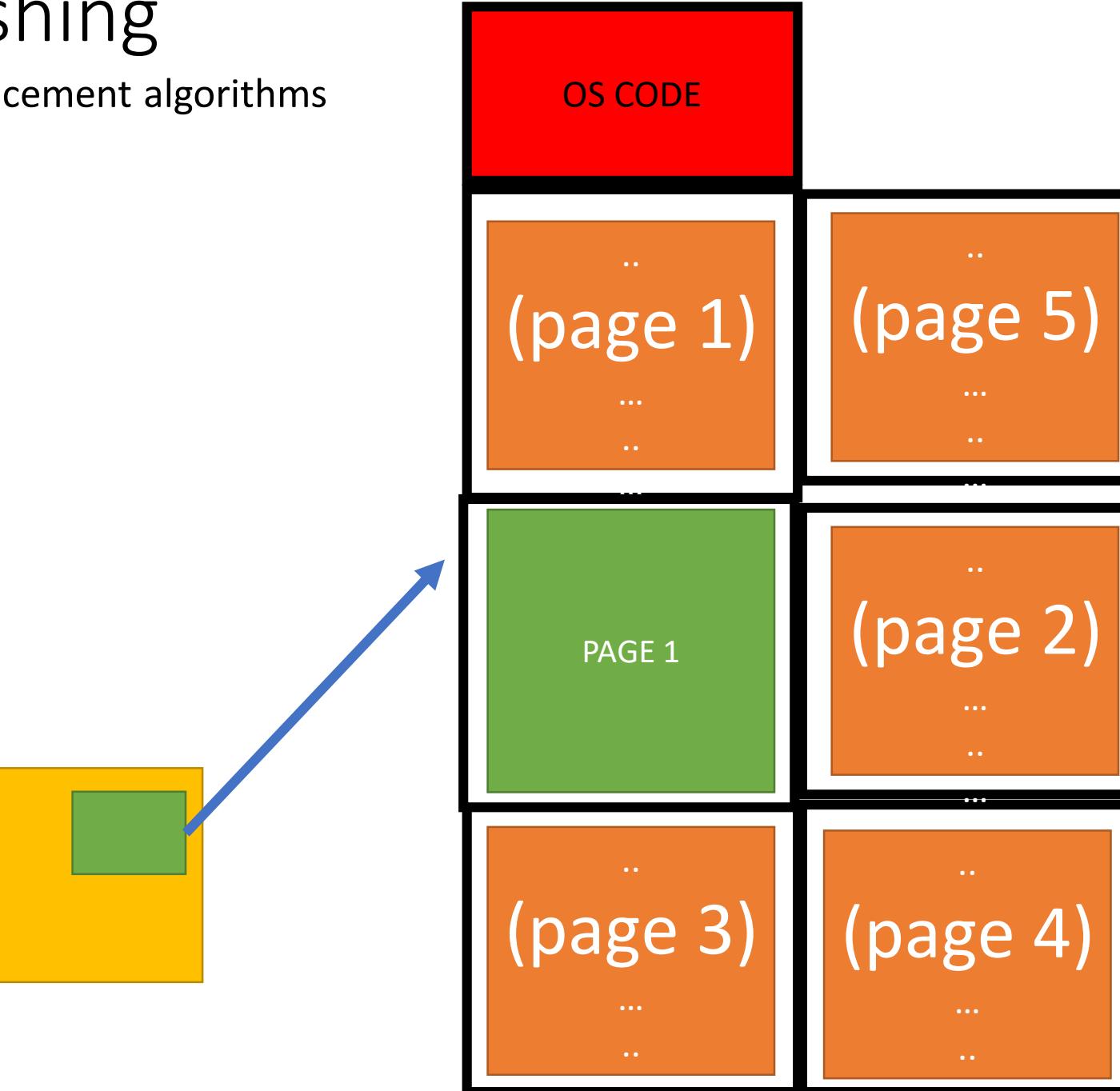
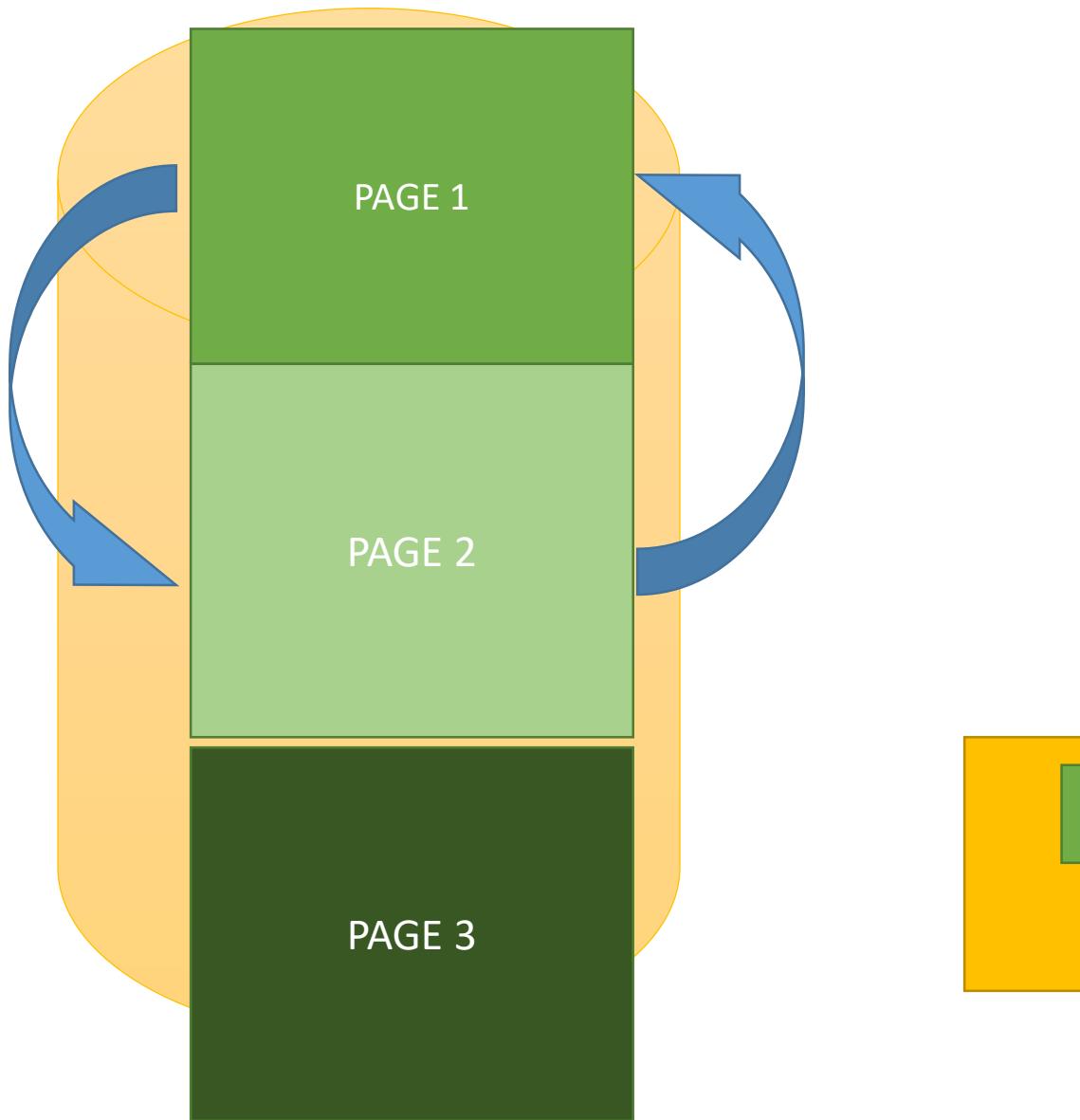
Demand Paging : Thrashing

- This happens due to inefficient Page replacement algorithms



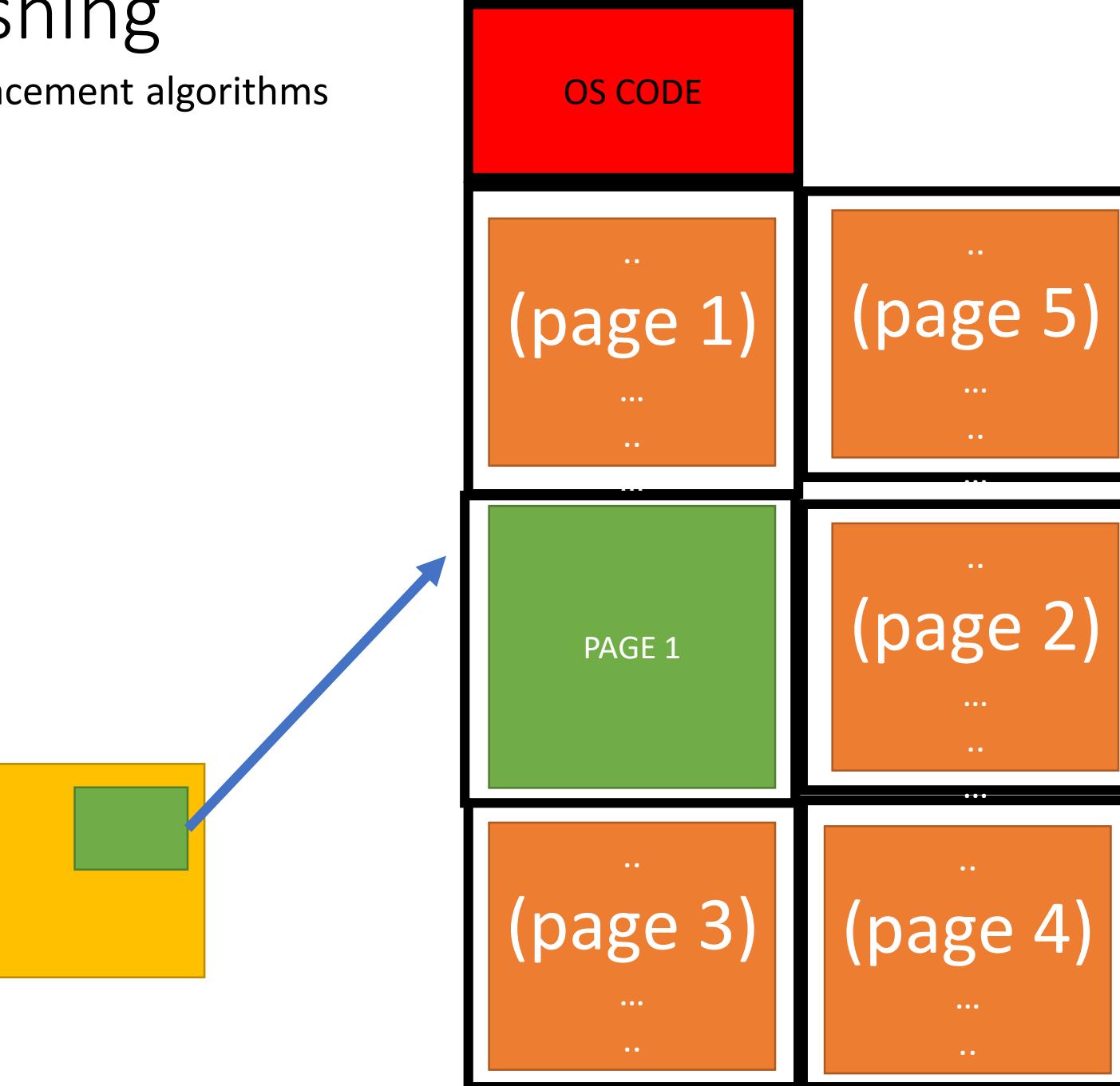
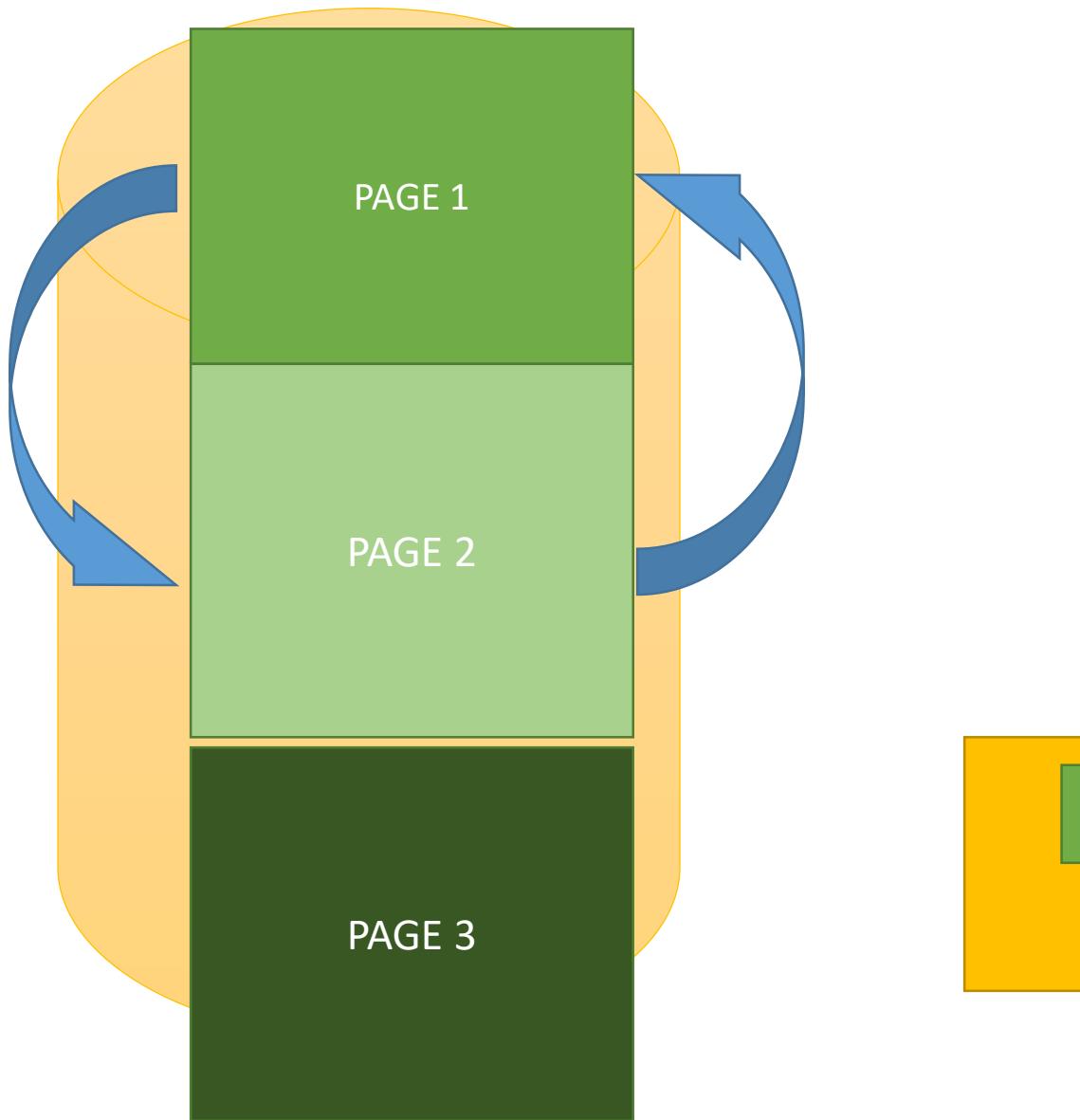
Demand Paging : Thrashing

- This happens due to inefficient Page replacement algorithms



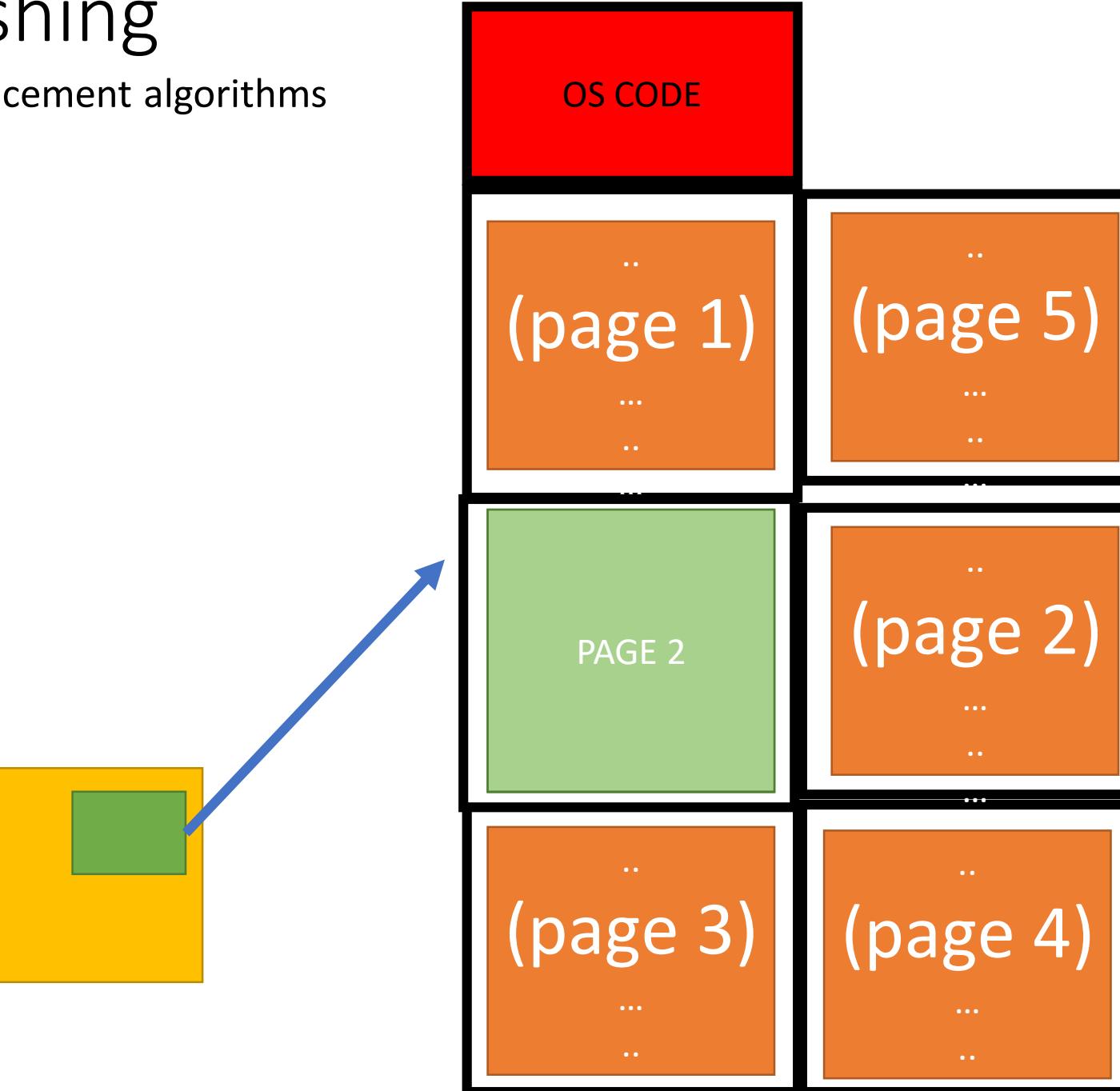
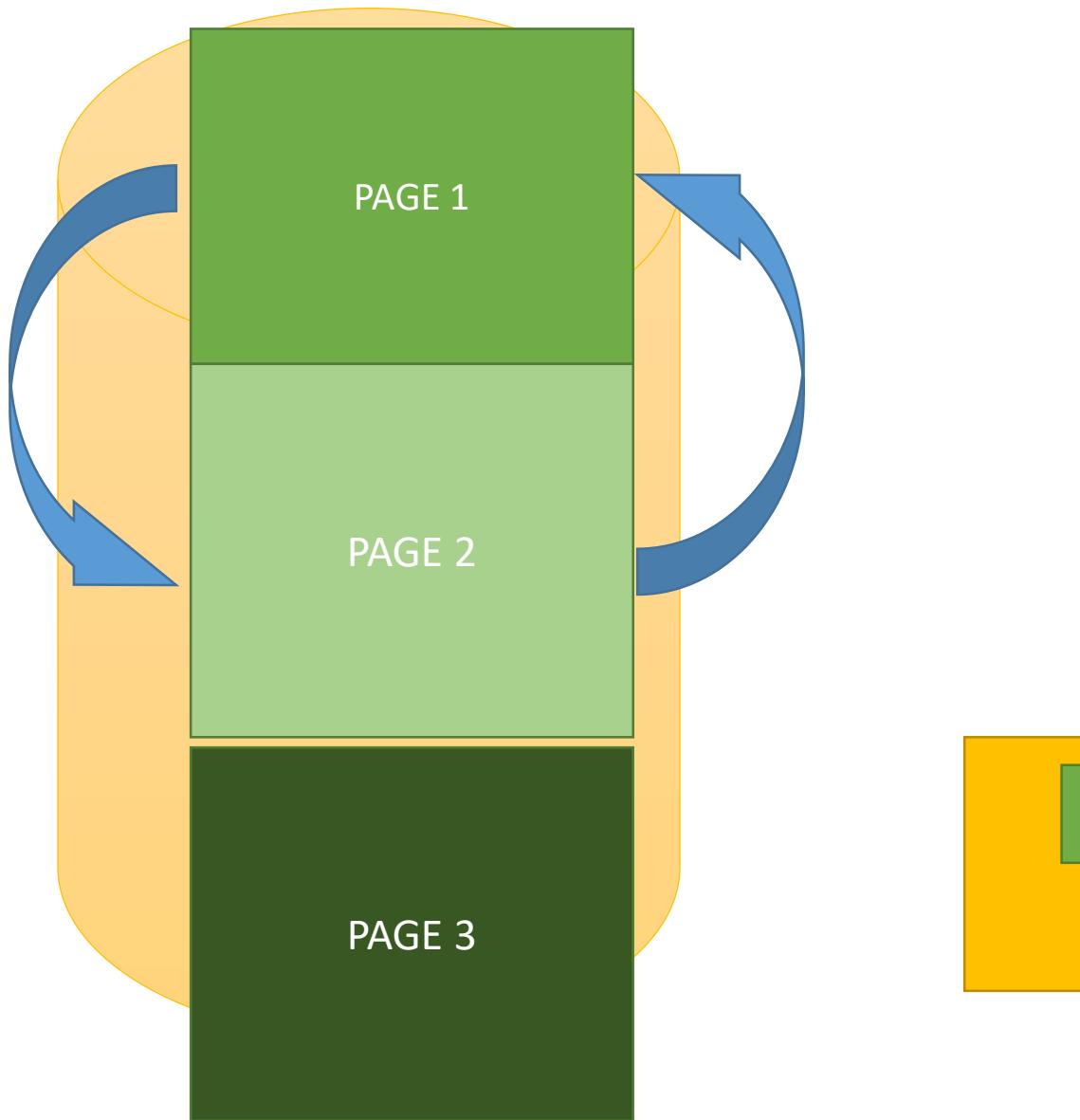
Demand Paging : Thrashing

- This happens due to inefficient Page replacement algorithms



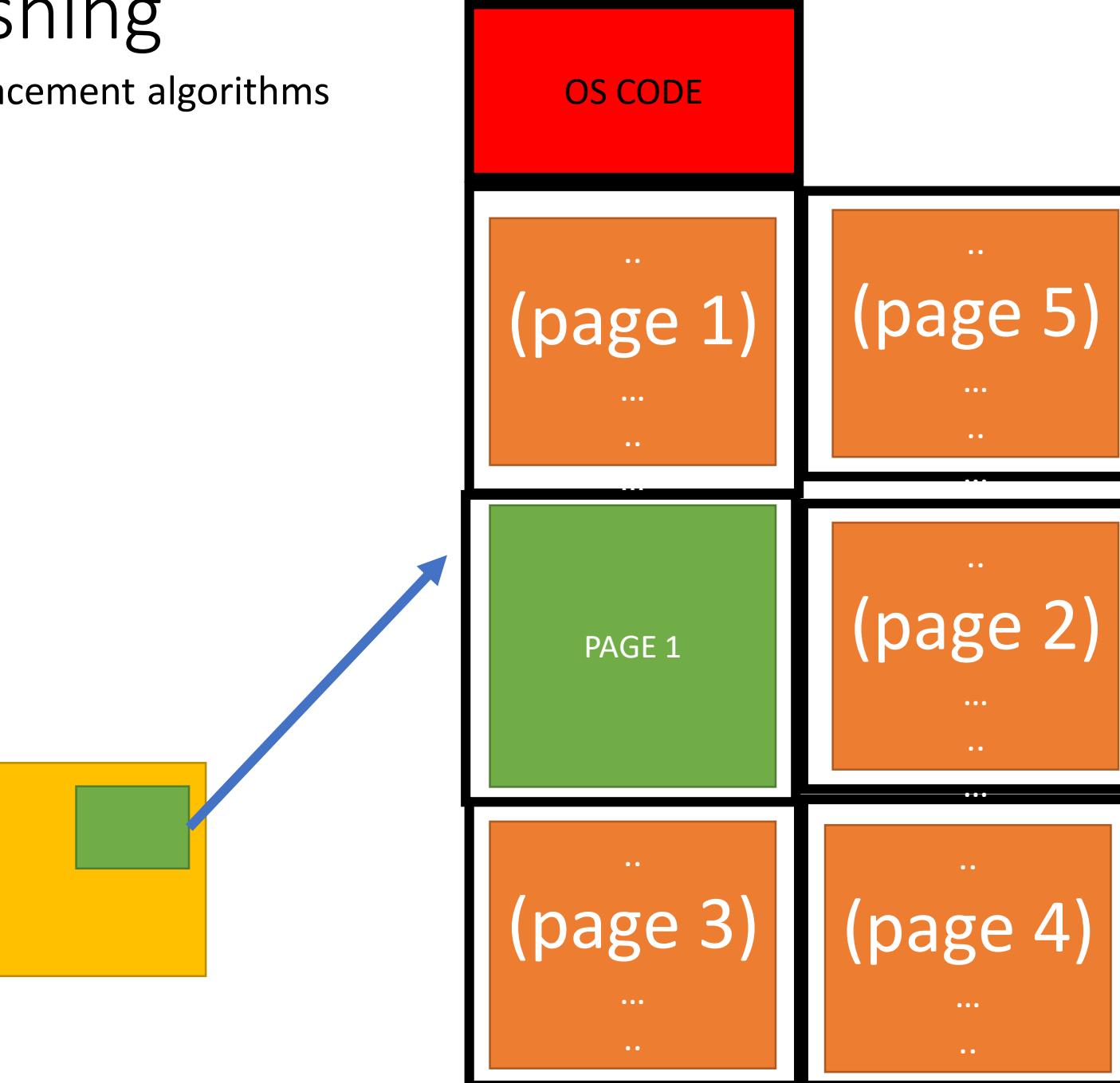
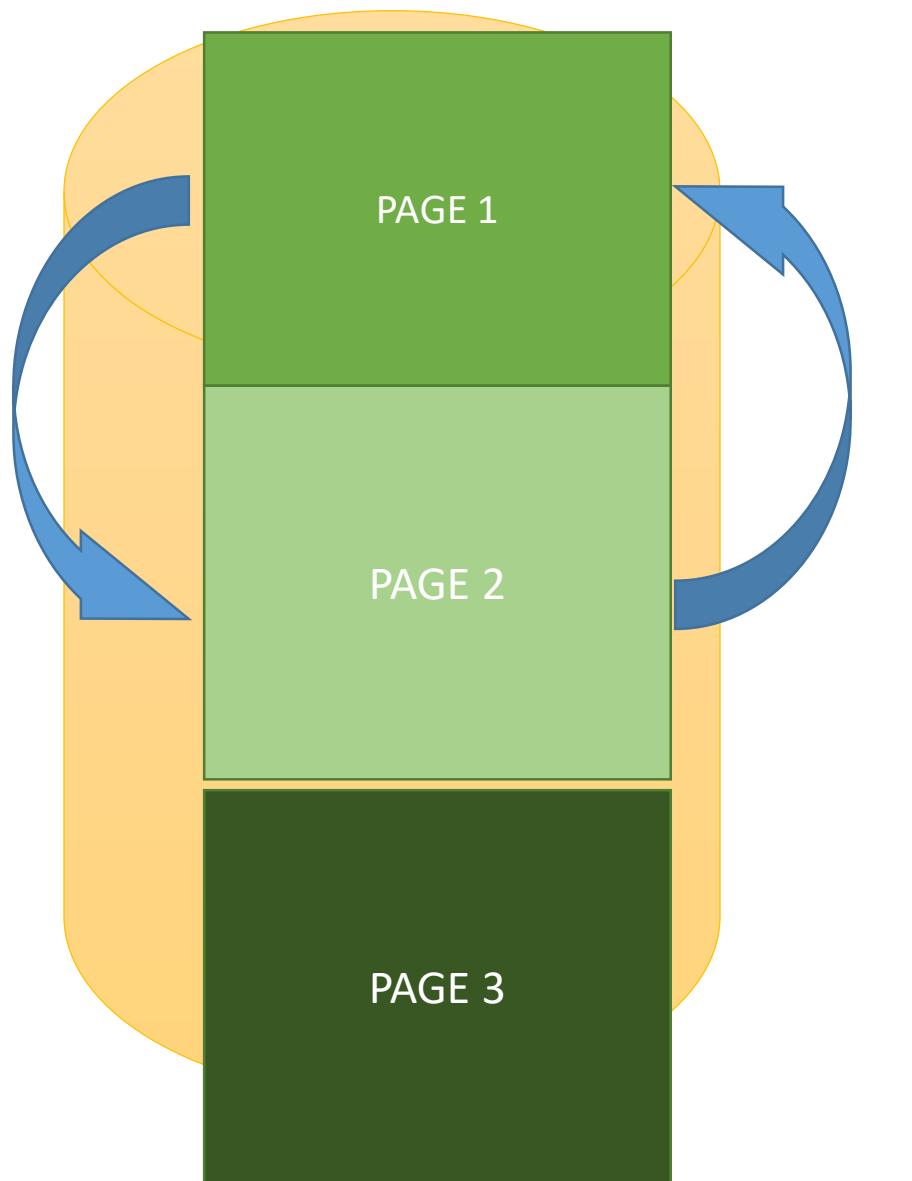
Demand Paging : Thrashing

- This happens due to inefficient Page replacement algorithms



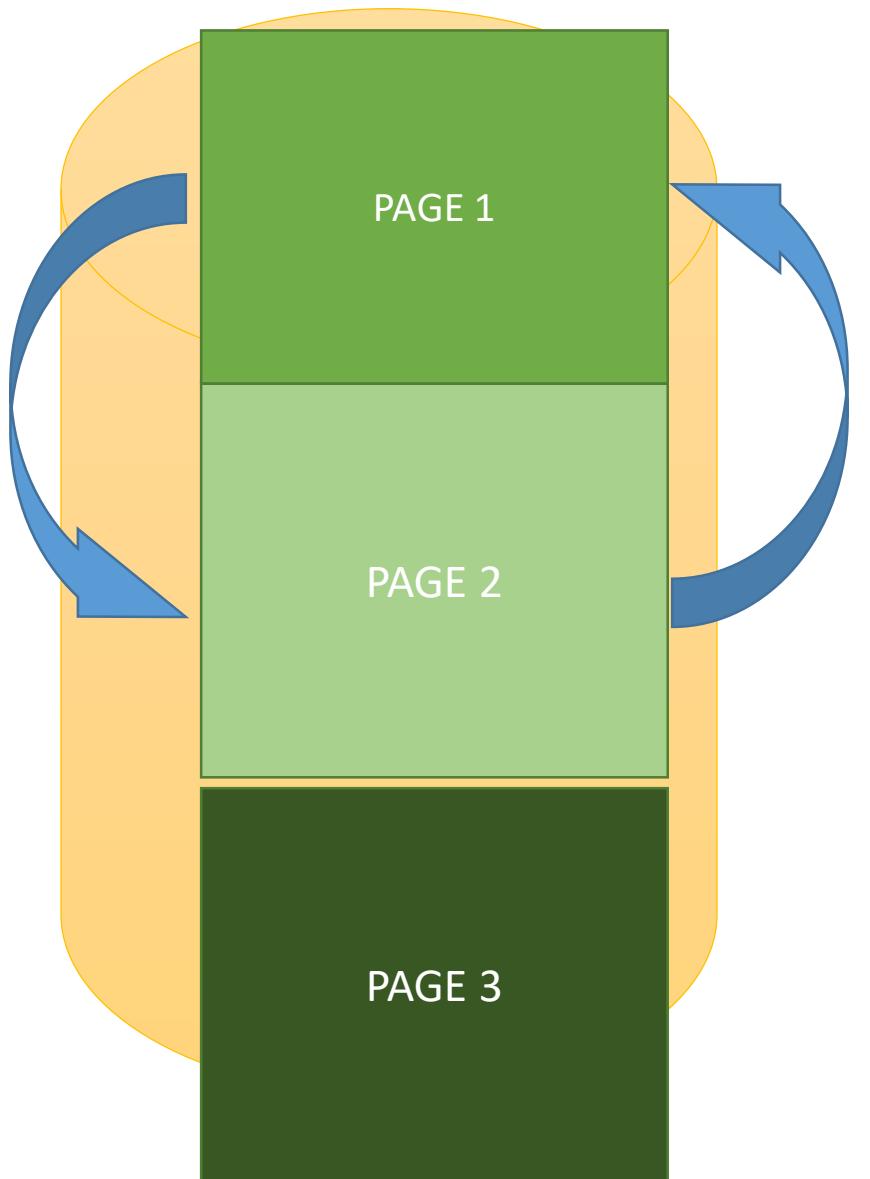
Demand Paging : Thrashing

- This happens due to inefficient Page replacement algorithms

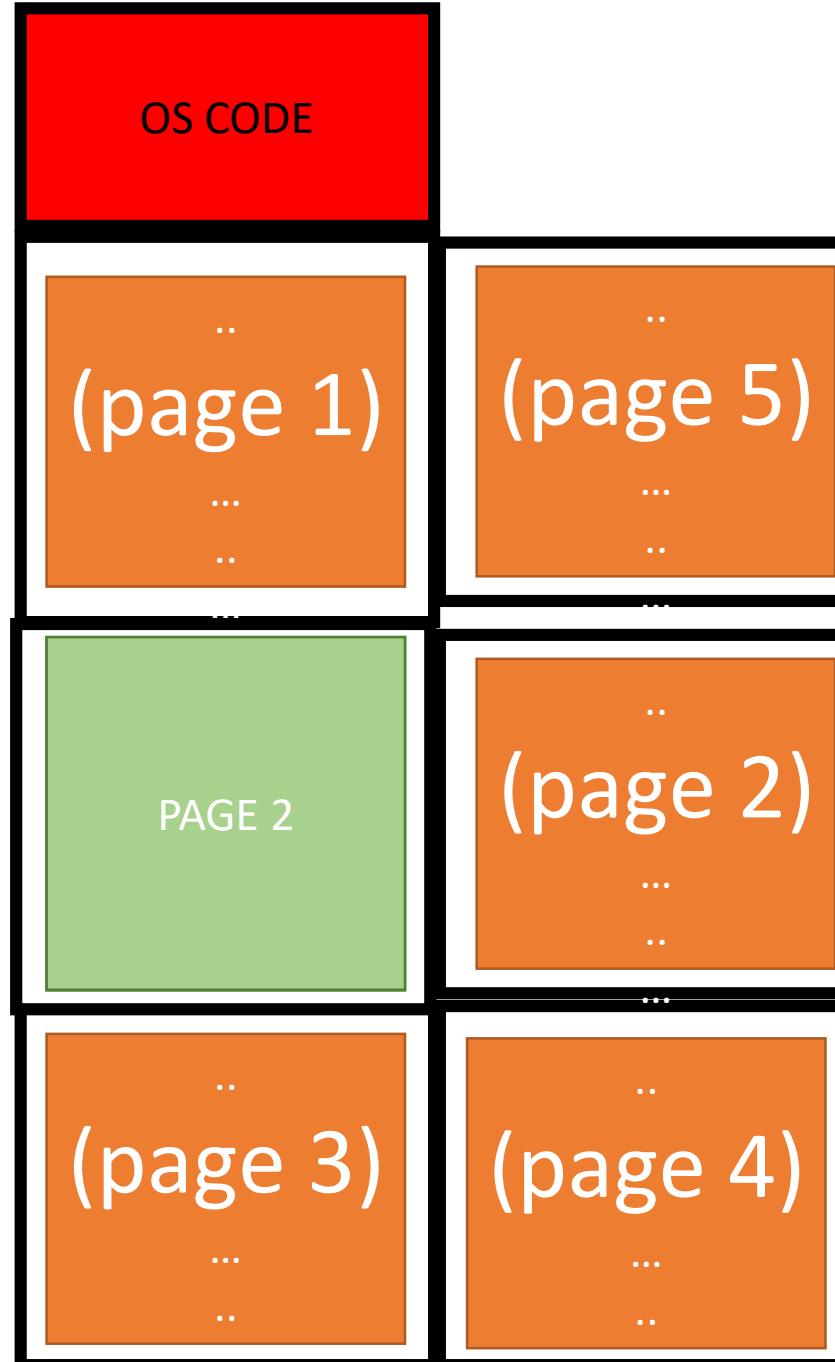
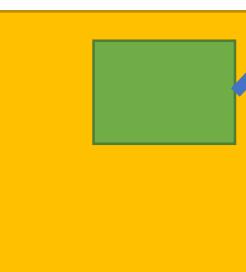


Demand Paging : Thrashing

- This happens due to inefficient Page replacement algorithms



Thrashing:
Results in
More memory
Read write operations
And hence **slows down**
The system



4.2 Improving Paging

Inverted Page Table

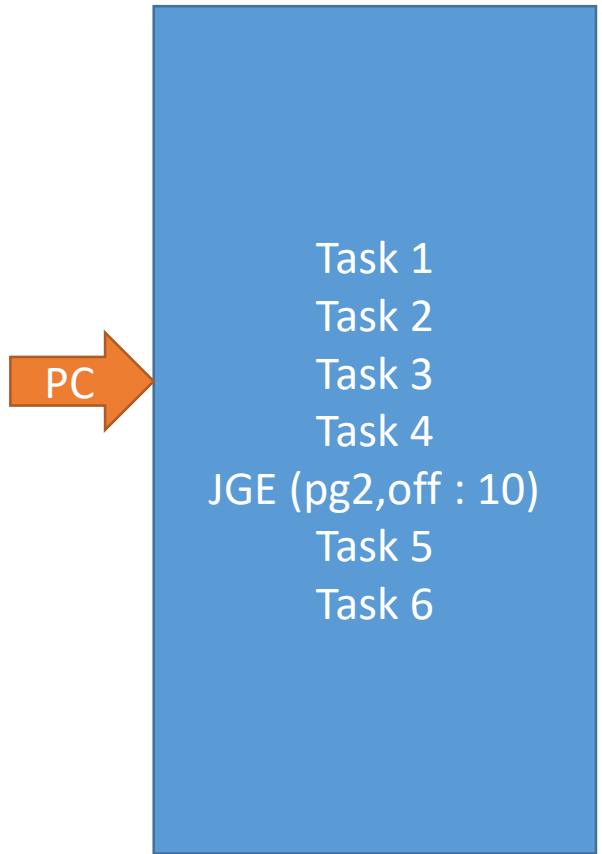
Translation Lookaside Buffer

Combining TLB with Cache

Inverted Page Table

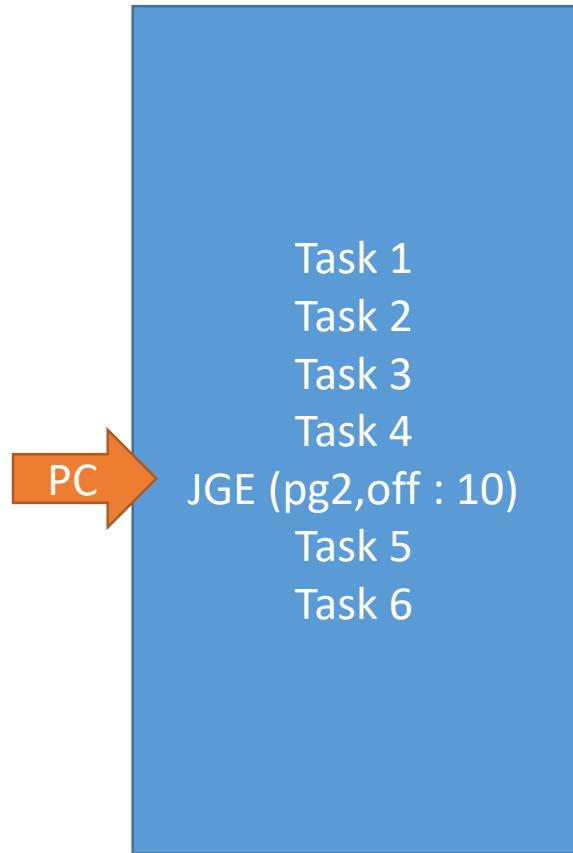
- So far we have seen that:
 - If there are **n processes**, then there will be **n corresponding page tables**
 - This takes up a **lot** of space in the **memory** of OS.
- Solution for this:
 - A new kind of table
 - Known as : **Inverted page table**
- Characteristics of Inverted Page Table
 - **Page to Frame** mapping
 - **Process ID** tracking (this feature allows the use of **only 1 table**)

Inverted Page Table



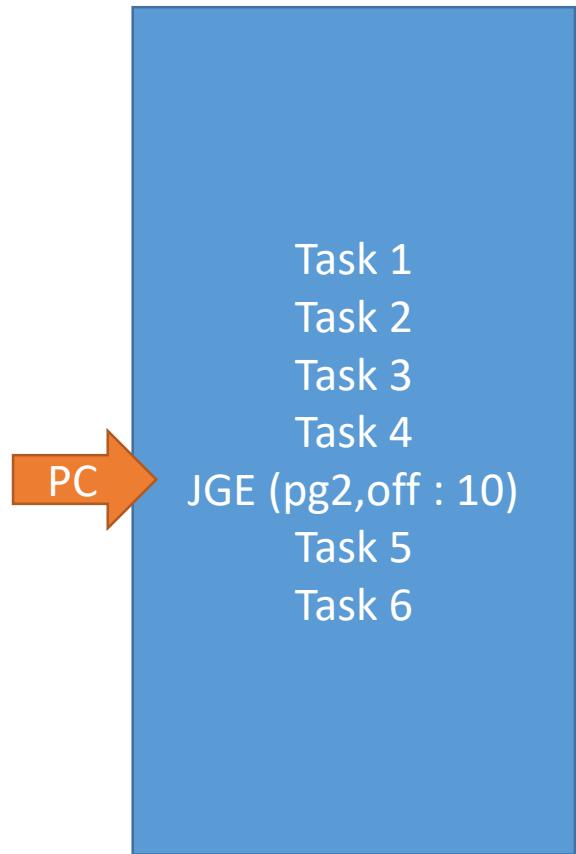
This is the page 1 of a process with process ID 1

Inverted Page Table



- The processor will shift control to OS.
- OS code will contain a **hash function**

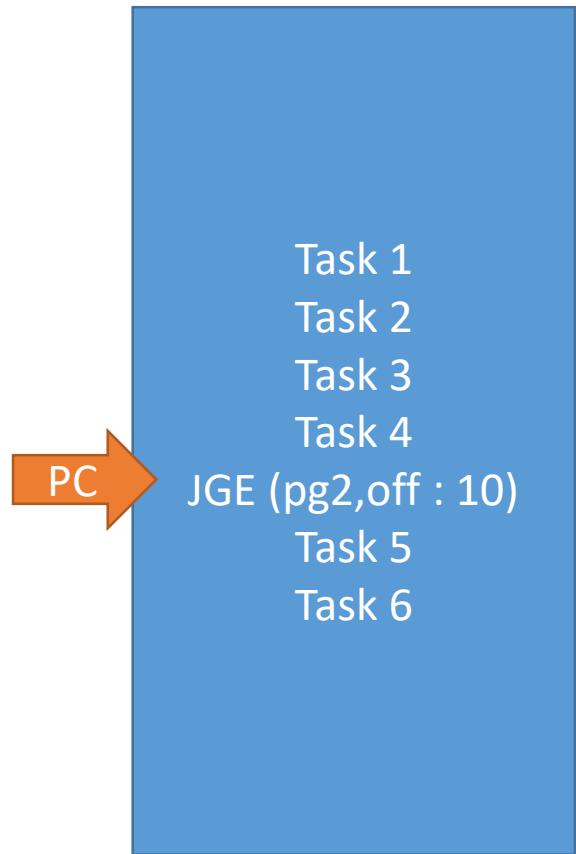
Inverted Page Table



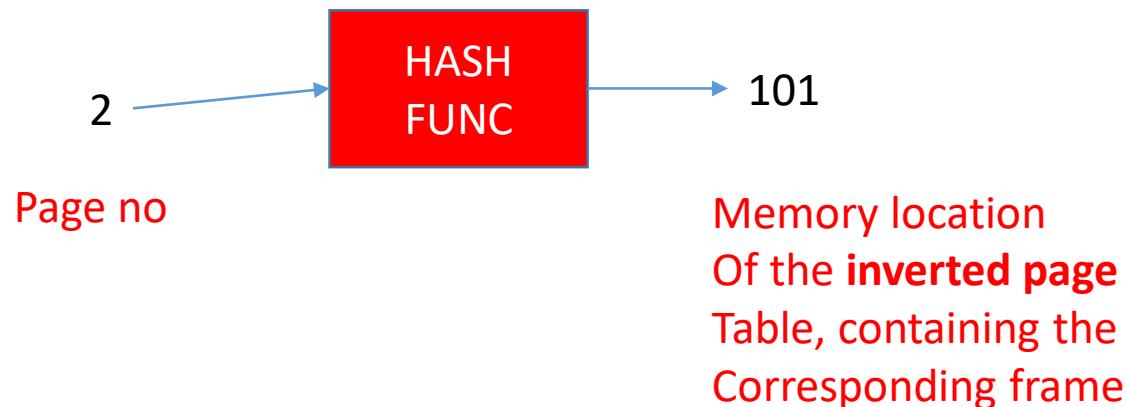
- The processor will shift control to OS.
- OS code will contain a hash function



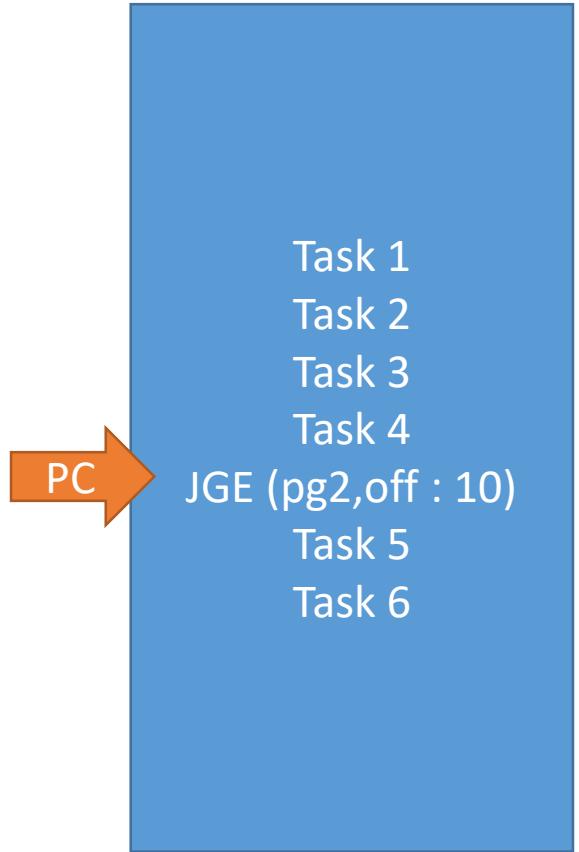
Inverted Page Table



- The processor will shift control to OS.
- OS code will contain a hash function

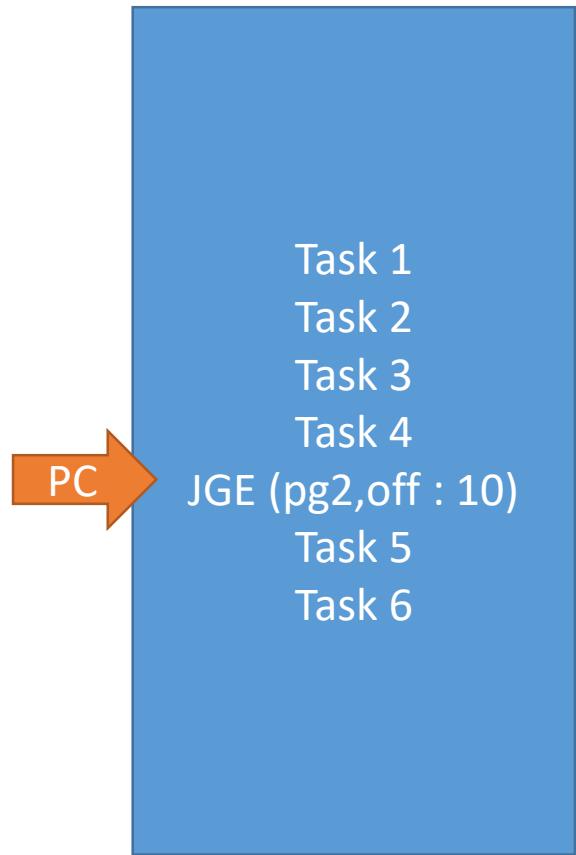


Inverted Page Table



OS will
Use this
Value of **101**
And **search**
The **inverted**
Page table

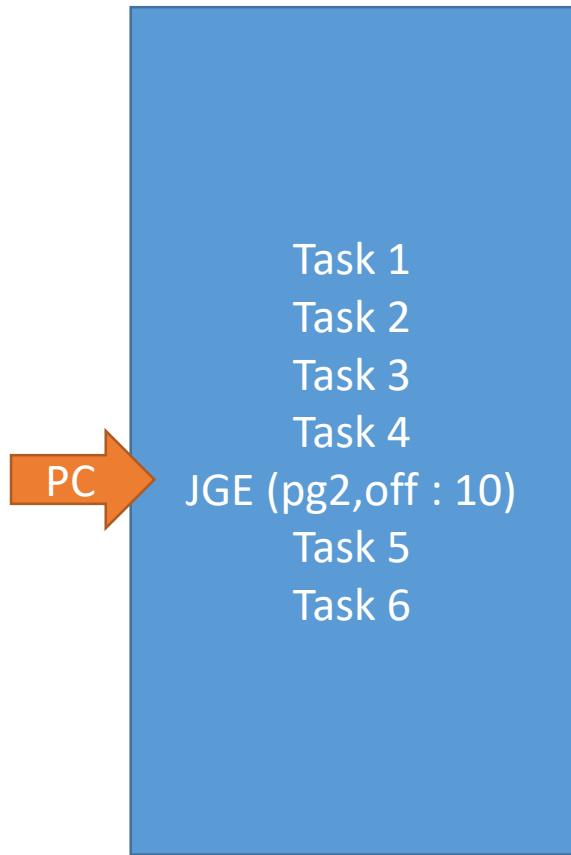
Inverted Page Table



OS will
Use this
Value of 101
And search
The inverted
Page table

| Memory Locn | Process ID | Frame | Chain |
|-------------|------------|-------|-------|
| 101 | 1 | 20 | - |
| 102 | | | |
| 103 | | | |
| 104 | | | |
| 105 | | | |
| 106 | | | |
| 107 | | | |

Inverted Page Table



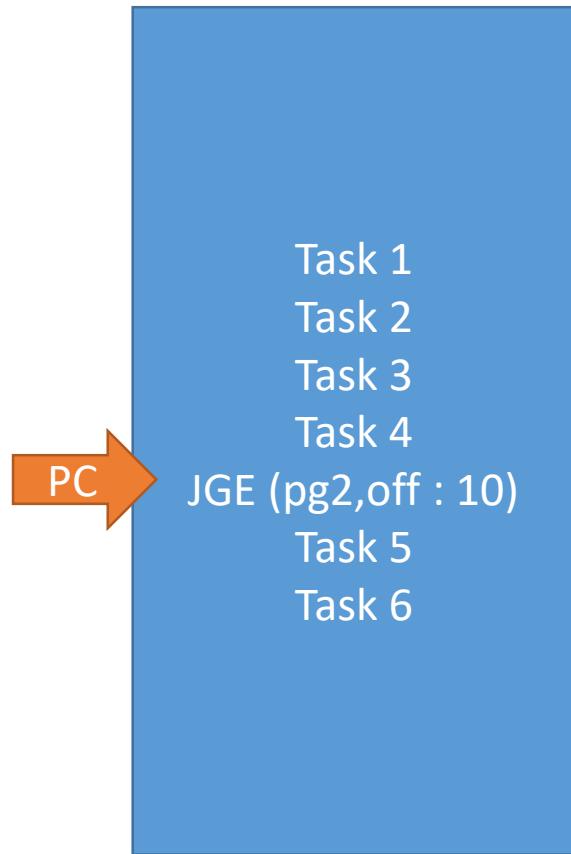
OS will
Use this
Value of 101
And search
The inverted
Page table

So for this process with ID 1, **the page**
Is located in the 20th frame

| Memory Locn | Process ID | Frame | Chain |
|-------------|------------|-------|-------|
| 101 | 1 | 20 | - |
| 102 | | | |
| 103 | | | |
| 104 | | | |
| 105 | | | |
| 106 | | | |
| 107 | | | |

Inverted Page Table

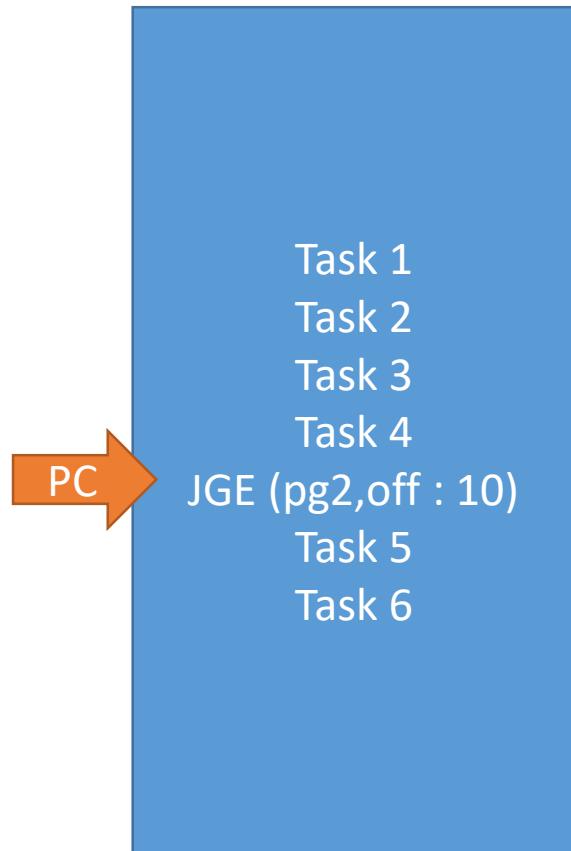
BUT WHAT IF THERE WAS NO ENTRY?



| Memory Locn | Process ID | Frame | Chain |
|-------------|------------|-------|-------|
| 101 | | | - |
| 102 | | | |
| 103 | | | |
| 104 | | | |
| 105 | | | |
| 106 | | | |
| 107 | | | |

Inverted Page Table

BUT WHAT IF THERE WAS NO ENTRY?



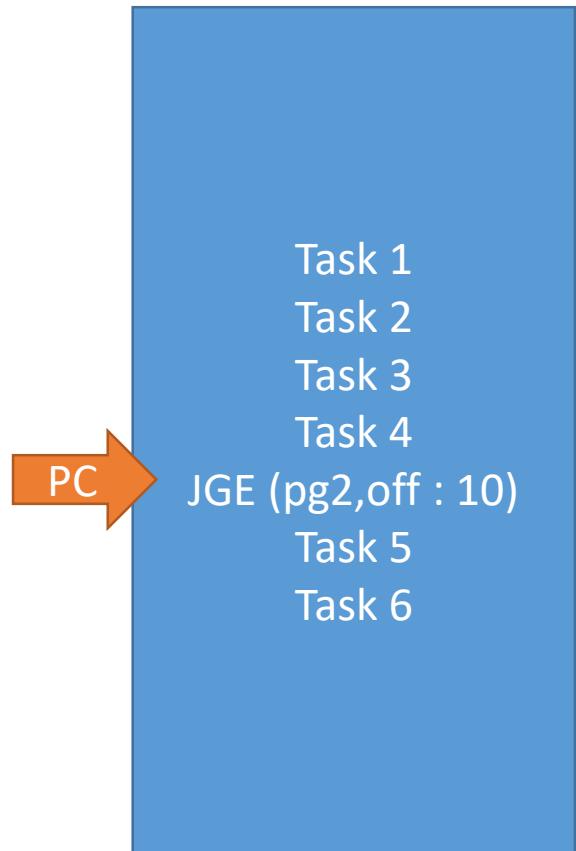
OS will
Use this
Value of 101
And search
The inverted
Page table

| Memory Locn | Process ID | Frame | Chain |
|-------------|------------|-------|-------|
| 101 | | | - |
| 102 | | | |
| 103 | | | |
| 104 | | | |
| 105 | | | |
| 106 | | | |
| 107 | | | |

It would mean that, the **page 2 is
Not in the memory yet**

Inverted Page Table

BUT WHAT IF THERE WAS NO ENTRY?



OS will
Use this
Value of 101
And search
The inverted
Page table

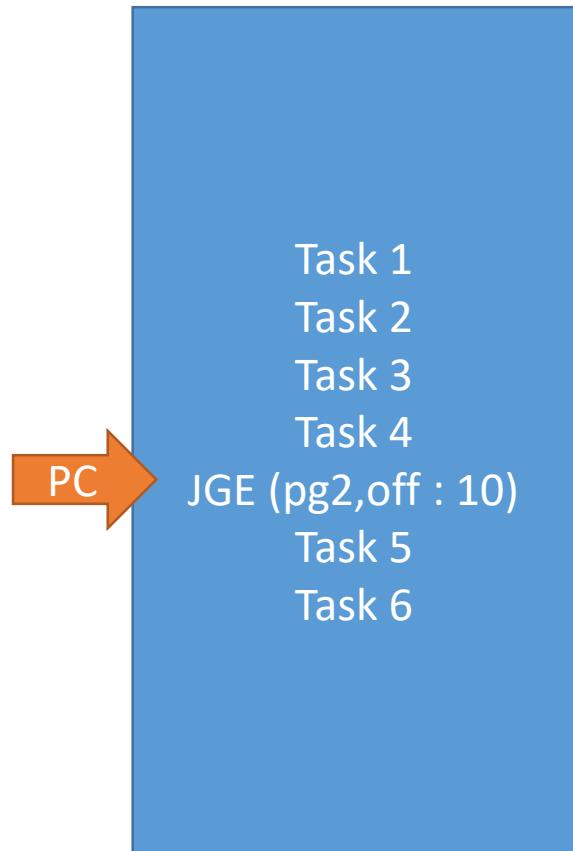
| Memory Locn | Process ID | Frame | Chain |
|-------------|------------|-------|-------|
| 101 | | | - |
| 102 | | | |
| 103 | | | |
| 104 | | | |
| 105 | | | |
| 106 | | | |
| 107 | | | |

It would mean that, the page 2 is
Not in the memory yet

That page will have to brought in

- **Empty frame** –place it there
- No empty frame- use **page replacement algo**

Inverted Page Table



OS will
Use this
Value of 101
And search
The inverted
Page table

BUT WHAT IF THERE WAS NO ENTRY?

Then we would **update the table** with the
Info about the frame it has been placed

| Memory Locn | Process ID | Frame | Chain |
|-------------|------------|-------|-------|
| 101 | 1 | 30 | -- |
| 102 | | | |
| 103 | | | |
| 104 | | | |
| 105 | | | |
| 106 | | | |
| 107 | | | |

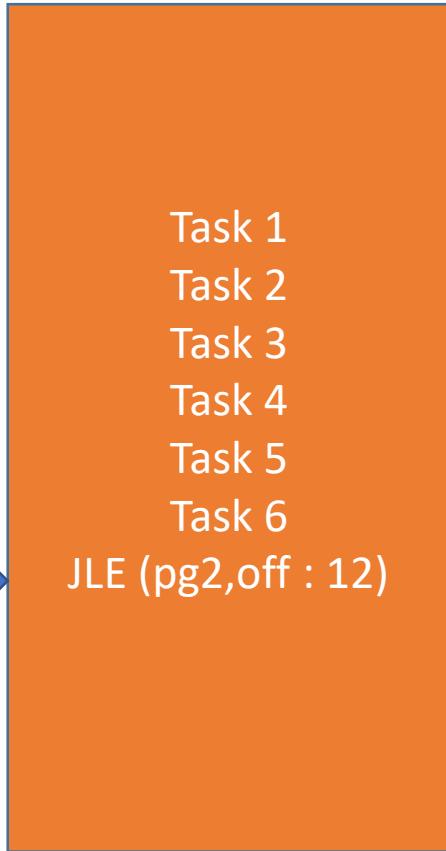
It would mean that, the page 2 is
Not in the memory yet

That page will have to brought in

- Empty frame –place it there
- No empty frame- use page replacement algo

Inverted Page Table (The Chain field)

PC →

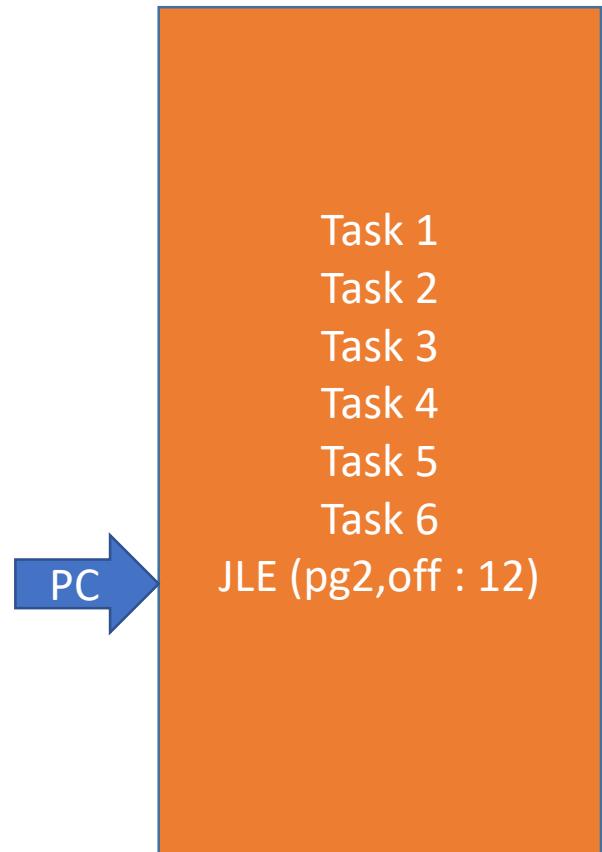


PROCESS WITH ID 2

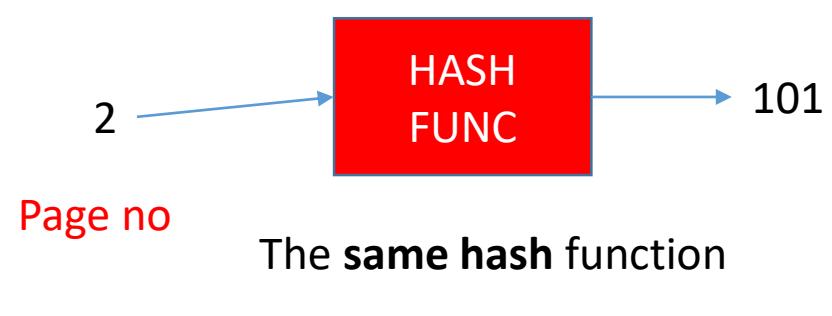
| Memory Locn | Process ID | Frame | Chain |
|-------------|------------|-------|-------|
| 101 | 1 | 30 | -- |
| 102 | | | |
| 103 | | | |
| 104 | | | |
| 105 | | | |
| 106 | | | |
| 107 | | | |

Let us assume that its **page 2** is not in the table

Inverted Page Table (The Chain field)

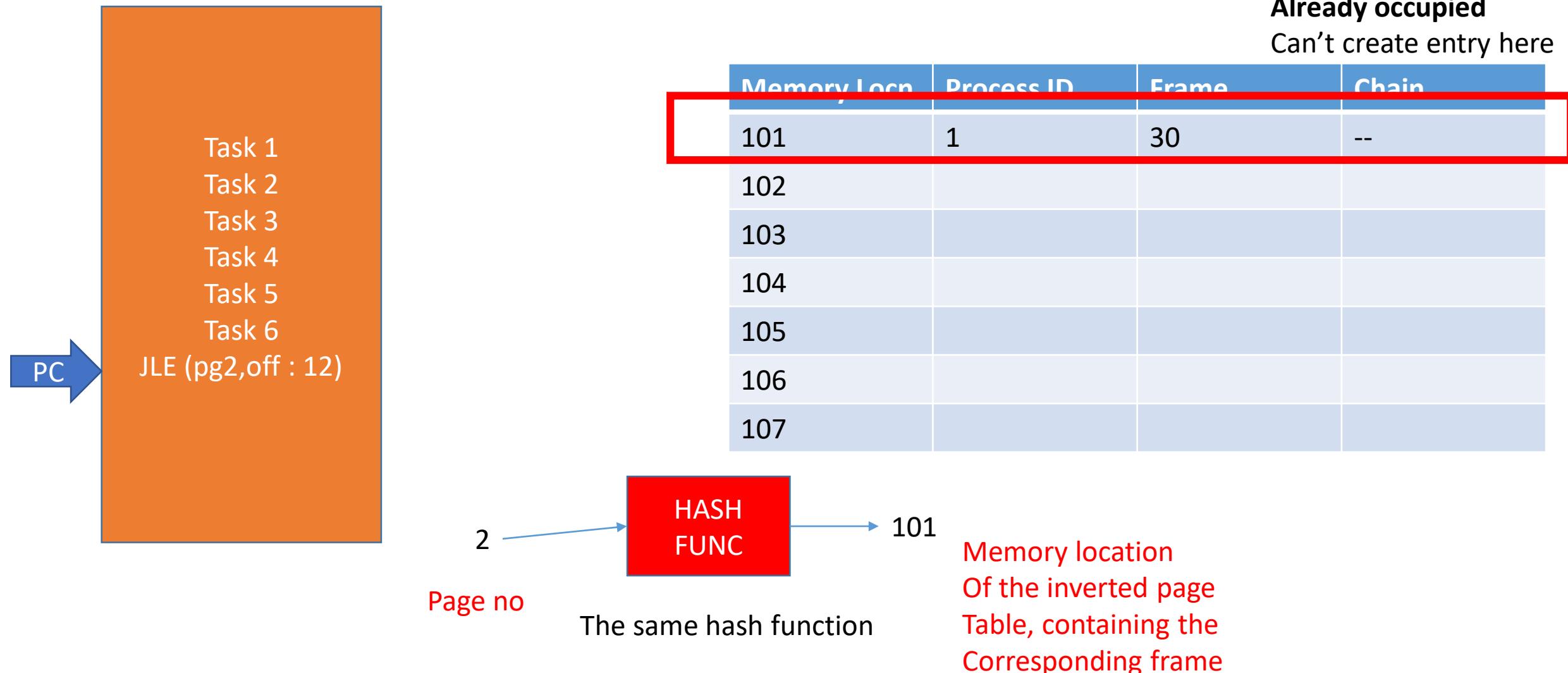


| Memory Locn | Process ID | Frame | Chain |
|-------------|------------|-------|-------|
| 101 | 1 | 30 | -- |
| 102 | | | |
| 103 | | | |
| 104 | | | |
| 105 | | | |
| 106 | | | |
| 107 | | | |

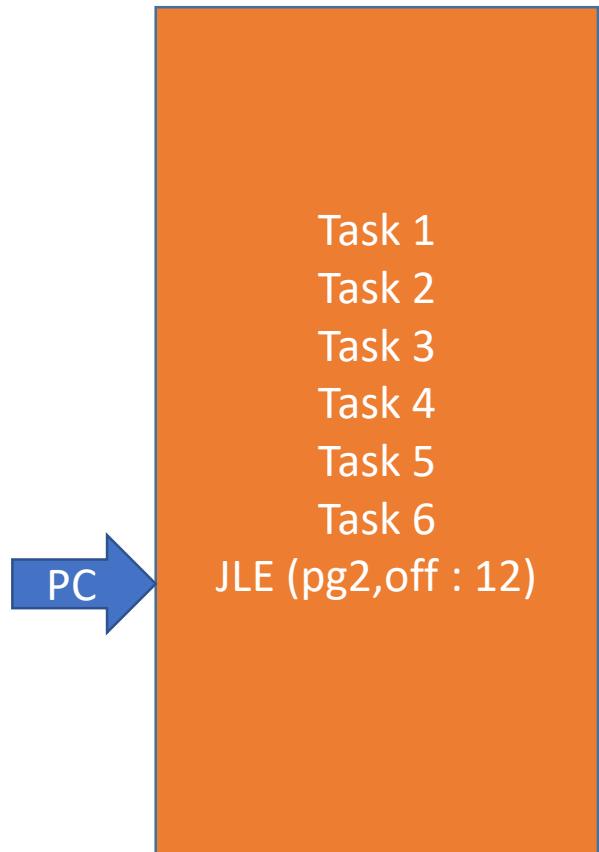


Memory location
Of the inverted page
Table, containing the
Corresponding frame

Inverted Page Table (The Chain field)

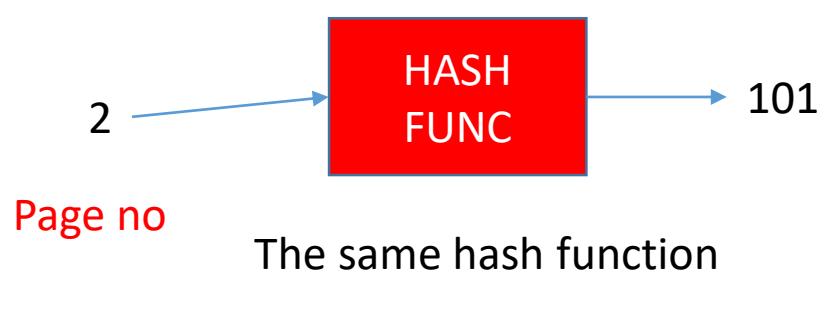


Inverted Page Table (The Chain field)



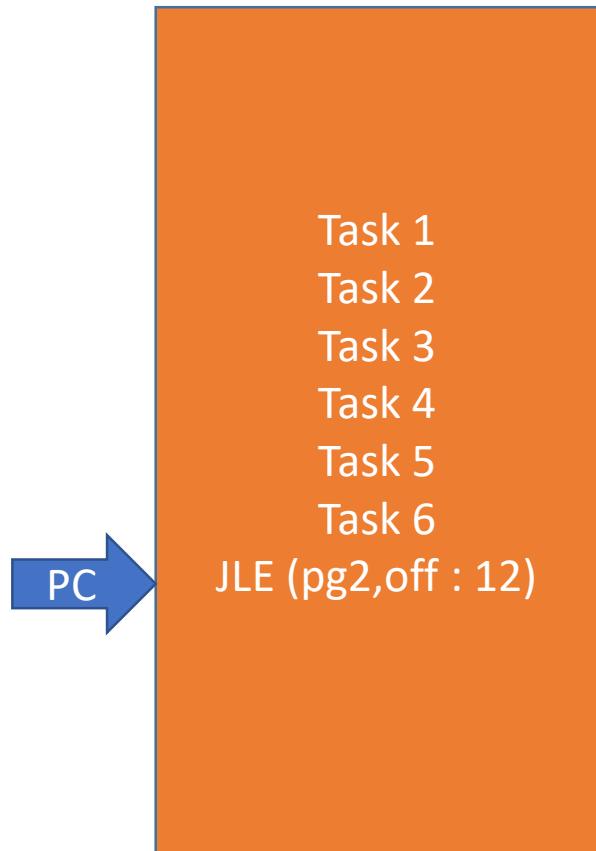
Search for the **nearest location**, that is **not occupied**
With page frame mapping data

| Memory Locn | Process ID | Frame | Chain |
|-------------|------------|-------|-------|
| 101 | 1 | 30 | -- |
| 102 | | | |
| 103 | | | |
| 104 | | | |
| 105 | | | |
| 106 | | | |
| 107 | | | |



Memory location
Of the inverted page
Table, containing the
Corresponding frame

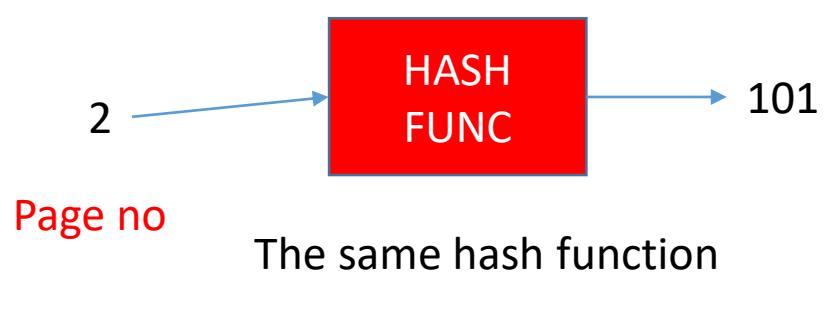
Inverted Page Table (The Chain field)



Let us **assume** that **these**
rows
Were already occupied

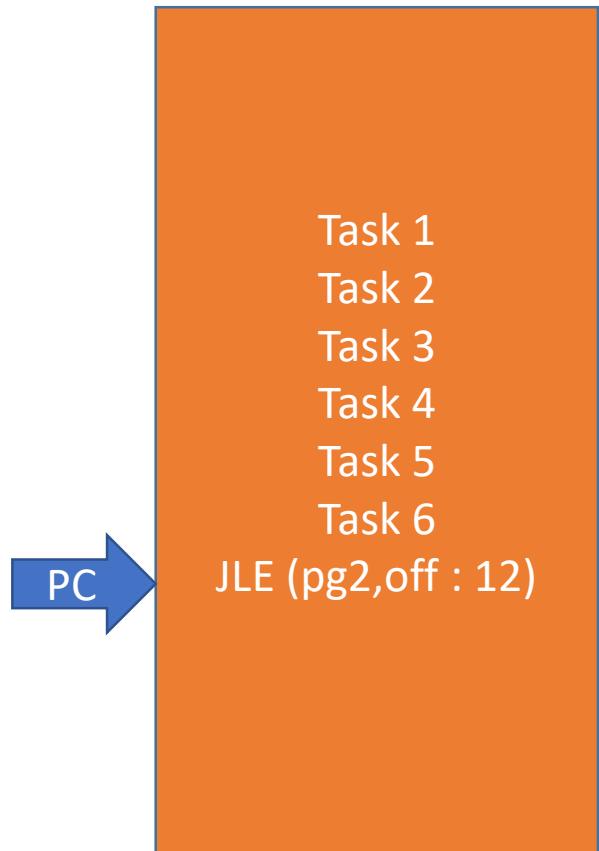
Search for the nearest location, that is not occupied
With page frame mapping data

| Memory Locn | Process ID | Frame | Chain |
|-------------|------------|-------|-------|
| 101 | 1 | 30 | -- |
| 102 | occupied | | |
| 103 | occupied | | |
| 104 | Occupied | | |
| 105 | occupied | | |
| 106 | | | |
| 107 | occupied | | |



Memory location
Of the inverted page
Table, containing the
Corresponding frame

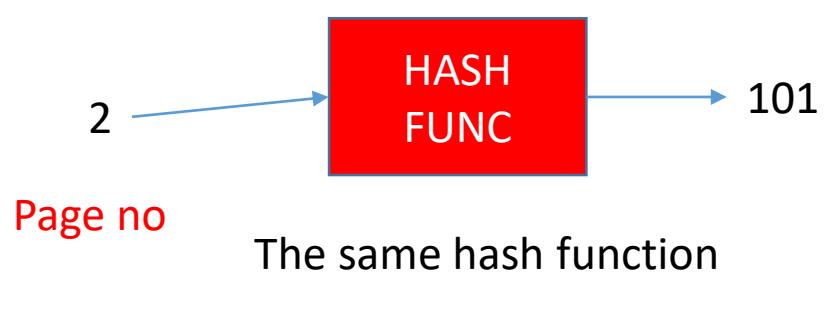
Inverted Page Table (The Chain field)



Let us assume that these rows
Were already occupied

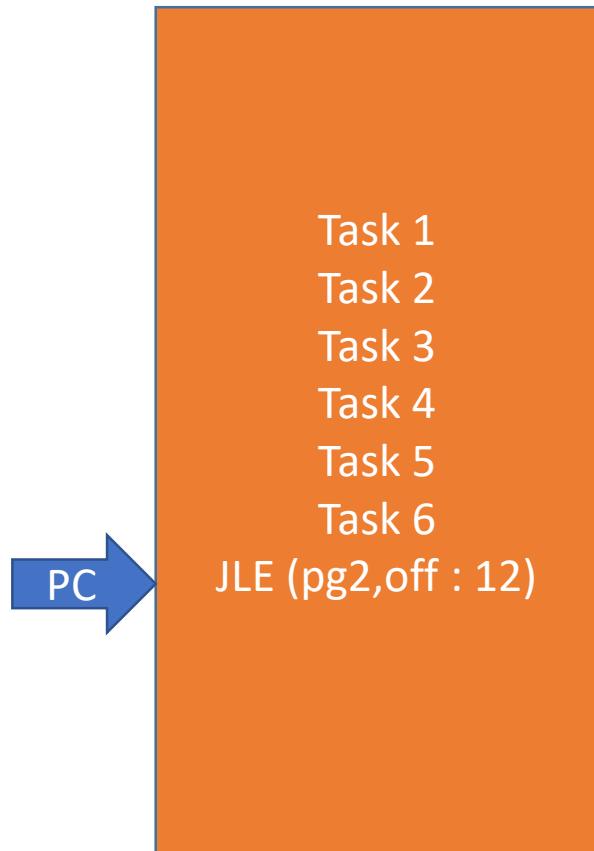
Search for the nearest location, that is not occupied
With page frame mapping data

| Memory Locn | Process ID | Frame | Chain |
|-------------|------------|-------|-------|
| 101 | 1 | 30 | -- |
| 102 | occupied | | |
| 103 | occupied | | |
| 104 | Occupied | | |
| 105 | occupied | | |
| 106 | | | |
| 107 | occupied | | |



Memory location
Of the inverted page
Table, containing the
Corresponding frame

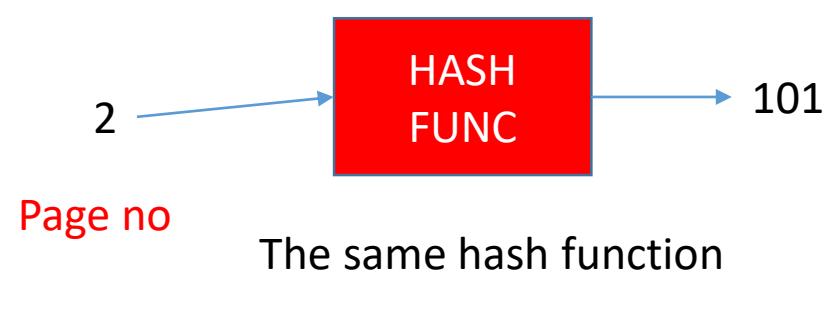
Inverted Page Table (The Chain field)



Let us assume that these rows
Were already occupied

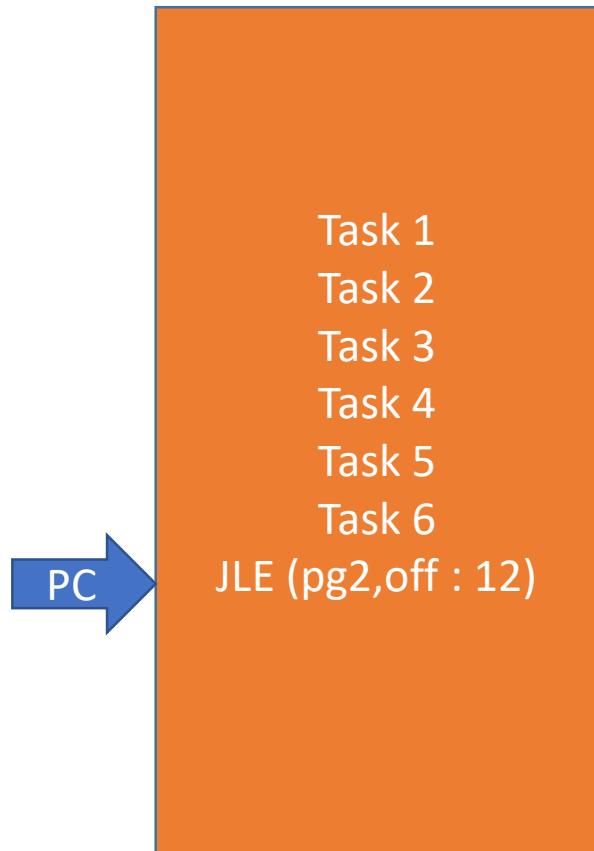
Search for the nearest location, that is not occupied
With page frame mapping data

| Memory Locn | Process ID | Frame | Chain |
|-------------|------------|-------|-------|
| 101 | 1 | 30 | 106 |
| 102 | occupied | | |
| 103 | occupied | | |
| 104 | Occupied | | |
| 105 | occupied | | |
| 106 | 2 | 45 | - |
| 107 | occupied | | |



Memory location
Of the inverted page
Table, containing the
Corresponding frame

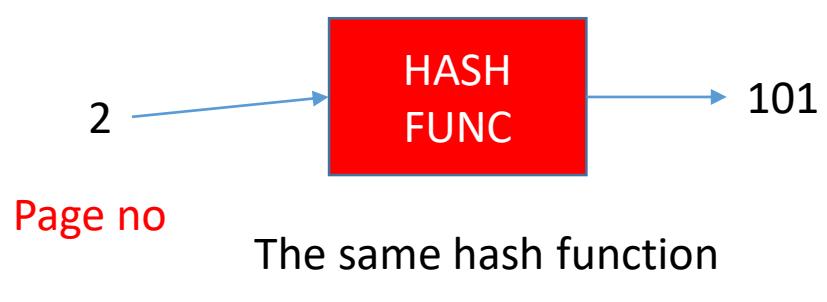
Inverted Page Table (The Chain field)



Let us assume that these rows
Were already occupied

Search for the nearest location, that is not occupied
With page frame mapping data

| Memory Locn | Process ID | Frame | Chain |
|-------------|------------|-------|-------|
| 101 | 1 | 30 | 106 |
| 102 | occupied | | |
| 103 | occupied | | |
| 104 | Occupied | | |
| 105 | occupied | | |
| 106 | 2 | 45 | - |
| 107 | occupied | | |

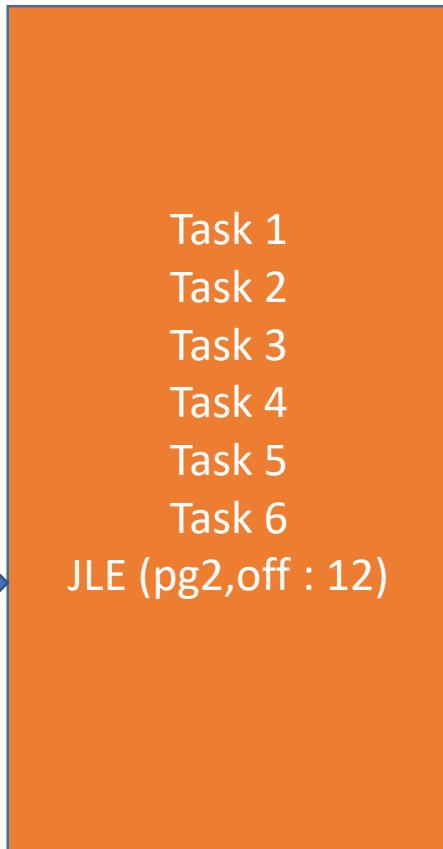


Memory location
Of the inverted page
Table, containing the
Corresponding frame

The page
Rep algo placed
2nd page of process 2
In the 45th frame

Inverted Page Table (The Chain field)

PC →

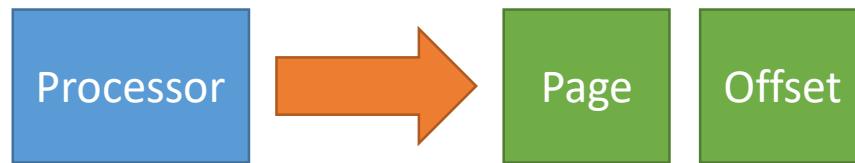


| Memory Locn | Process ID | Frame | Chain |
|-------------|------------|-------|-------|
| 101 | 1 | 30 | 106 |
| 102 | occupied | | |
| 103 | occupied | | |
| 104 | Occupied | | |
| 105 | occupied | | |
| 106 | 2 | 45 | - |
| 107 | occupied | | |

Next time when **page 2** for **process 2** will be searched. The **entry will be available** with the frame info

Translation Lookaside Buffer

Translation Lookaside Buffer



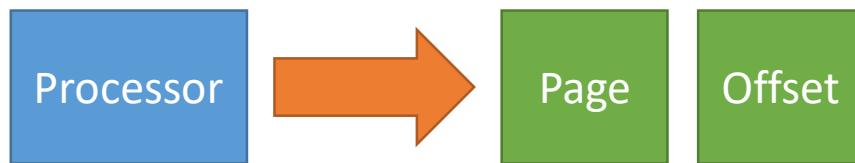
The processor
Requires
Instruction/
Data from this
location

| ID | Page | Frame |
|----|------|-------|
| | | |
| | | |
| | | |
| | | |

| ID | Frame | Chain |
|----|-------|-------|
| | | |
| | | |
| | | |
| | | |

Translation Lookaside Buffer

TLB: A h/w with the **recently used** Pages and corresponding mapping



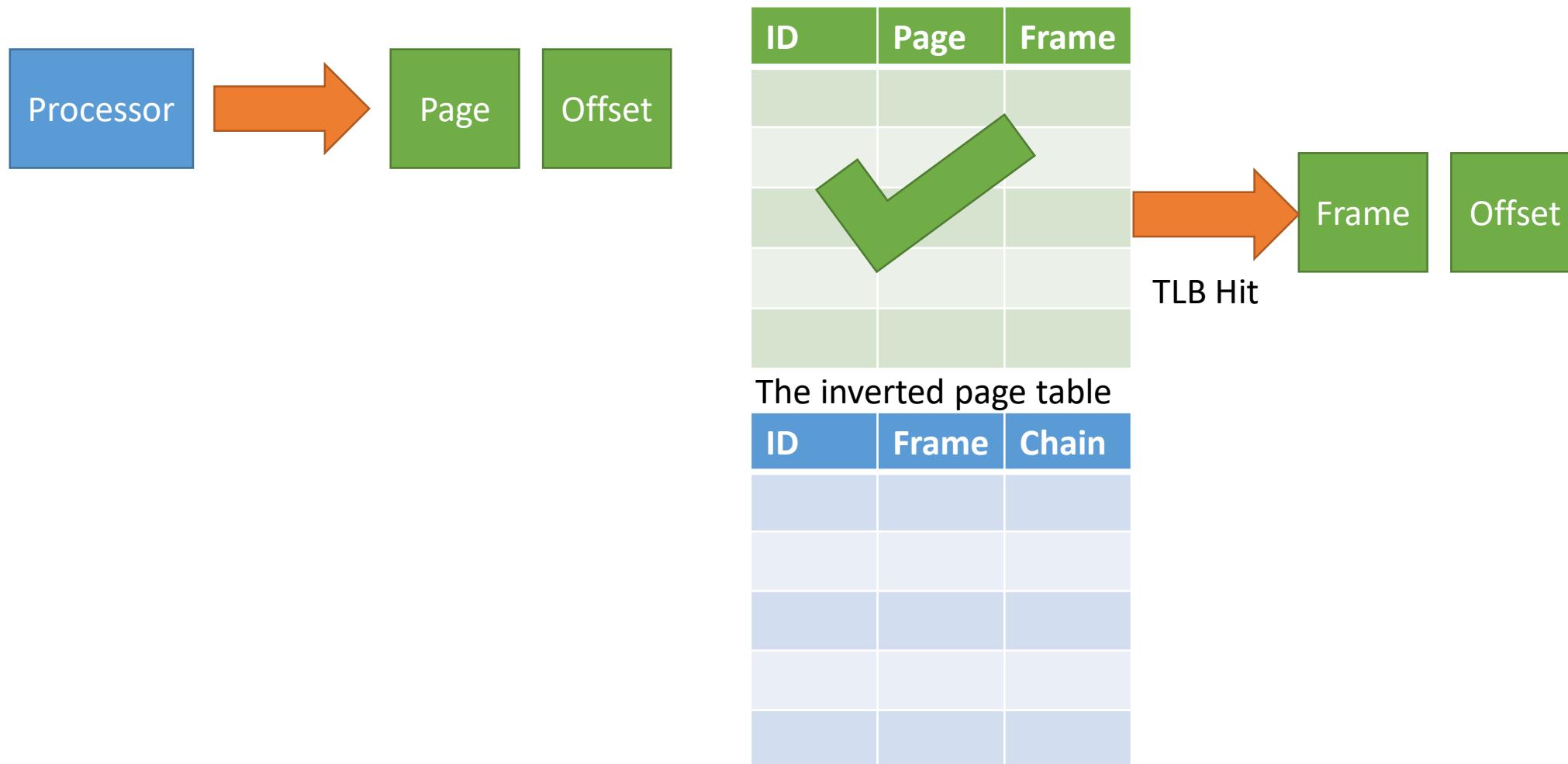
| ID | Page | Frame |
|----|------|-------|
| | | |
| | | |
| | | |
| | | |

The **inverted page table**

| ID | Frame | Chain |
|----|-------|-------|
| | | |
| | | |
| | | |
| | | |

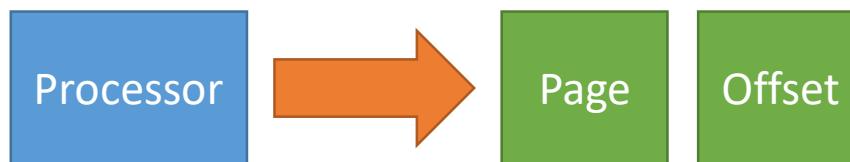
Translation Lookaside Buffer

TLB: A h/w with the recently used Pages and corresponding mapping



Translation Lookaside Buffer

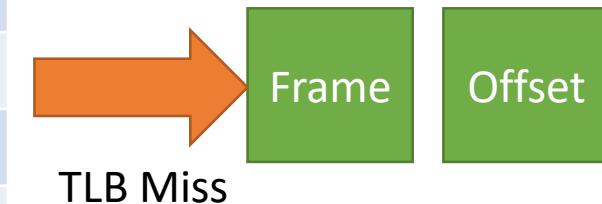
TLB: A h/w with the recently used Pages and corresponding mapping



| ID | Page | Frame |
|----|------|-------|
| | | |
| | | |
| | | |

The inverted page table

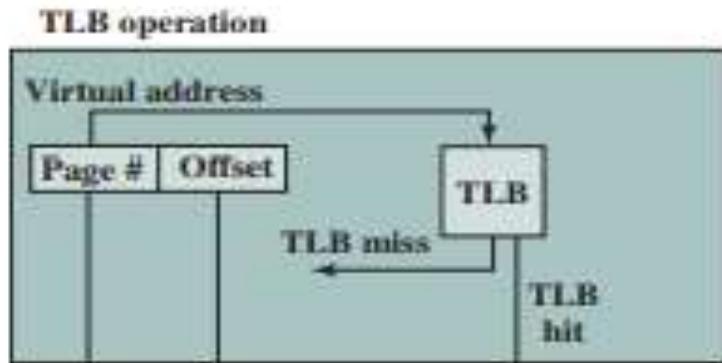
| ID | Frame | Chain |
|----|-------|-------|
| | | |
| | | |
| | | |



BASICALLY
TLB is a like a **cache**
For page frame mapping

Combining TLB with Cache

1.The processor requires a data from a particular page and offset

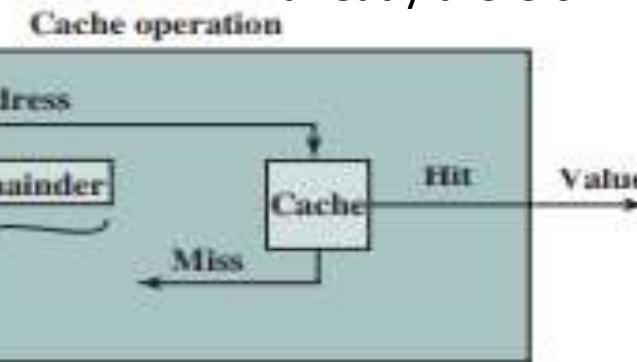


3.If it is not found, then the page table is checked

4. If it is not here, the page is brought in and the table is updated

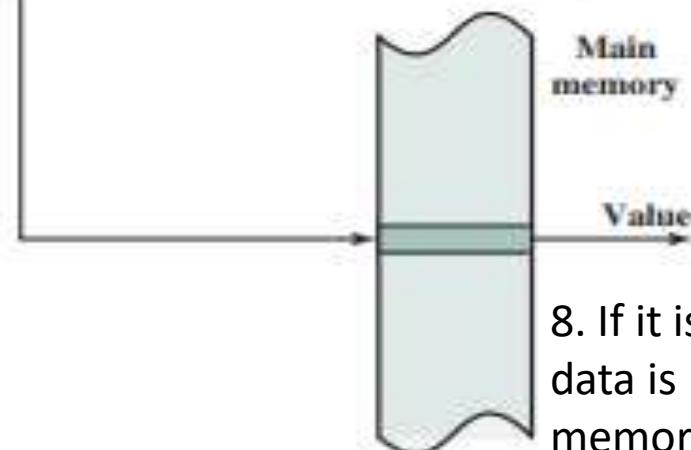
2.It searches the TLB for the physical location (frame)

7. The cache is checked to see if the data/instruction of that physical location is already there or not



5. Either way, The frame and Offset is found

6. Which means the physical location in the RAM is found



8. If it is not there, then the data is brought in from the memory

School Teacher :- You will learn this
in college

Professor :- You have learnt this in
your school

Me :-

Comp Archi
OS Teacher

School Teacher :- You will learn this
in college

Professor :- You have learnt this in
your school

Me :-



CSE 21

Comp Archi

OS Teacher

CSE 21

School Teacher :- You will learn this
in college

Professor :- You have learnt this in
your school

Me :-



THE END

CSE 213

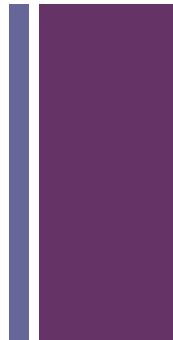
Computer Architecture

Lecture 9: Computer Arithmetic

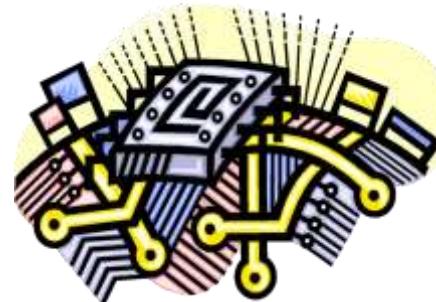
Military Institute of Science
and Technology



Arithmetic & Logic Unit (ALU)



- Part of the computer that actually performs arithmetic and logical operations on data
- All of the other elements of the computer system are there mainly to bring data into the ALU for it to process and then to take the results back out
- Based on the use of simple digital logic devices that can store binary digits and perform simple Boolean logic operations



ALU Inputs and Outputs

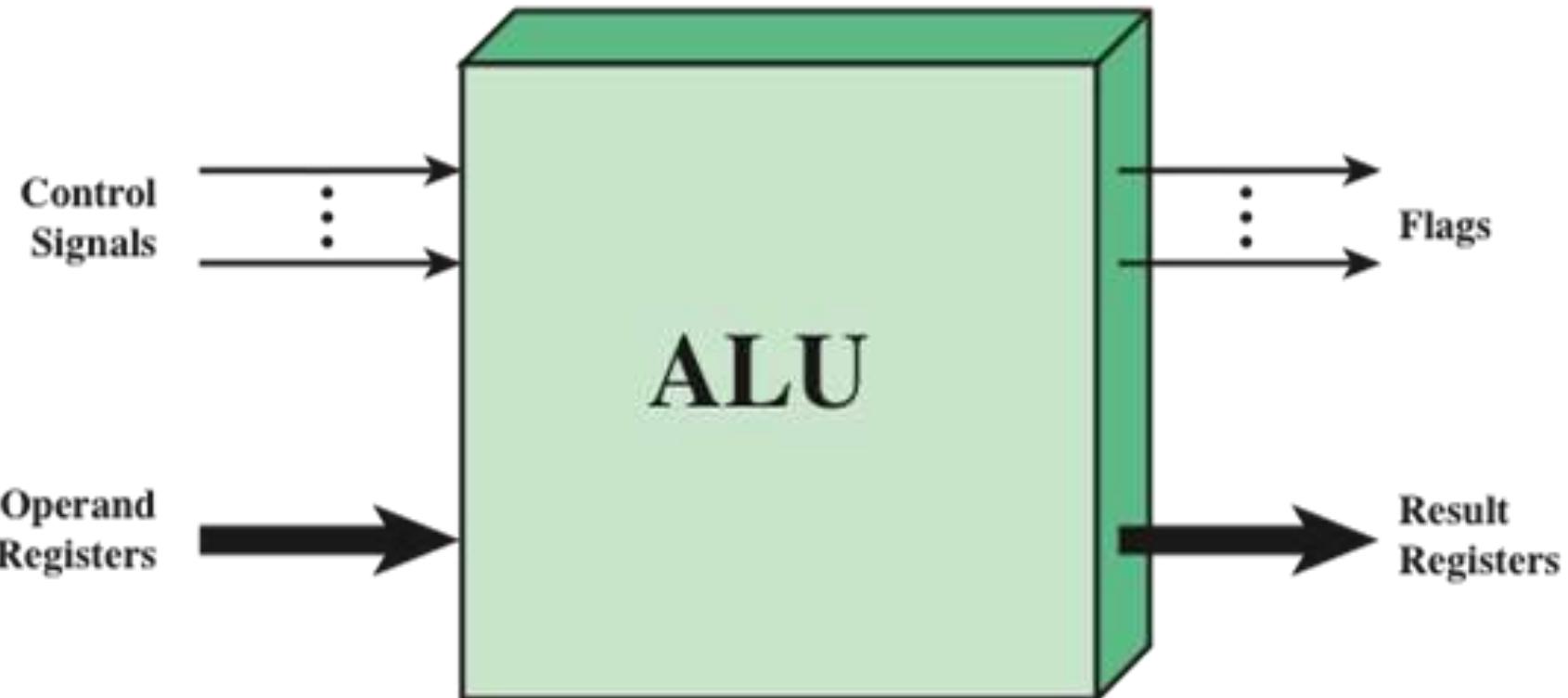


Figure 10.1 ALU Inputs and Outputs

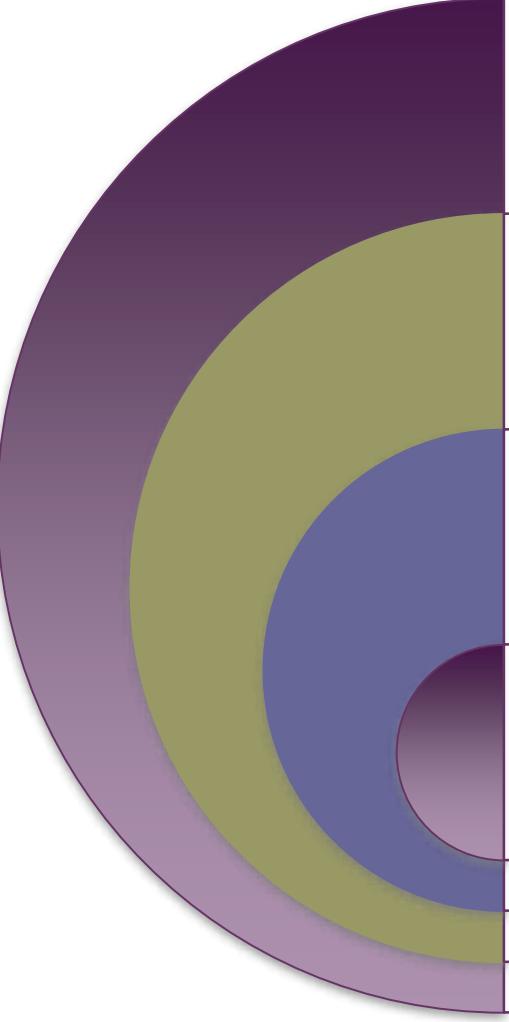
+

Integer Representation



- In the binary number system arbitrary numbers can be represented with:
 - The digits zero and one
 - The minus sign (for negative numbers)
 - The period, or *radix point* (for numbers with a fractional component)
- For purposes of computer storage and processing we do not have the benefit of special symbols for the minus sign and radix point
- Only binary digits (0,1) may be used to represent numbers

Sign-Magnitude Representation



There are several alternative conventions used to represent negative as well as positive integers

- All of these alternatives involve treating the most significant (leftmost) bit in the word as a sign bit
- If the sign bit is 0 the number is positive
- If the sign bit is 1 the number is negative

Sign-magnitude representation is the simplest form that employs a sign bit

Drawbacks:

- Addition and subtraction require a consideration of both the signs of the numbers and their relative magnitudes to carry out the required operation
- There are two representations of 0

Because of these drawbacks, sign-magnitude representation is rarely used in implementing the integer portion of the ALU



Twos Complement Representation

- Uses the most significant bit as a sign bit
- Differs from sign-magnitude representation in the way that the other bits are interpreted

| | |
|--|--|
| Range | -2_{n-1} through $2_{n-1} - 1$ |
| Number of Representations of Zero | One |
| Negation | Take the Boolean complement of each bit of the corresponding positive number, then add 1 to the resulting bit pattern viewed as an unsigned integer. |
| Expansion of Bit Length | Add additional bit positions to the left and fill in with the value of the original sign bit. |
| Overflow Rule | If two numbers with the same sign (both positive or both negative) are added, then overflow occurs if and only if the result has the opposite sign. |
| Subtraction Rule | To subtract B from A , take the twos complement of B and add it to A . |

Table 10.1 Characteristics of Twos Complement Representation and Arithmetic

Table 10.2

Alternative Representations for 4-Bit Integers

| Decimal Representation | Sign-Magnitude Representation | Twos Complement Representation | Biased Representation |
|------------------------|-------------------------------|--------------------------------|-----------------------|
| +8 | — | — | 1111 |
| +7 | 0111 | 0111 | 1110 |
| +6 | 0110 | 0110 | 1101 |
| +5 | 0101 | 0101 | 1100 |
| +4 | 0100 | 0100 | 1011 |
| +3 | 0011 | 0011 | 1010 |
| +2 | 0010 | 0010 | 1001 |
| +1 | 0001 | 0001 | 1000 |
| +0 | 0000 | 0000 | 0111 |
| -0 | 1000 | — | — |
| -1 | 1001 | 1111 | 0110 |
| -2 | 1010 | 1110 | 0101 |
| -3 | 1011 | 1101 | 0100 |
| -4 | 1100 | 1100 | 0011 |
| -5 | 1101 | 1011 | 0010 |
| -6 | 1110 | 1010 | 0001 |
| -7 | 1111 | 1001 | 0000 |
| -8 | — | 1000 | — |

Geometric Depiction of Twos Complement Integers

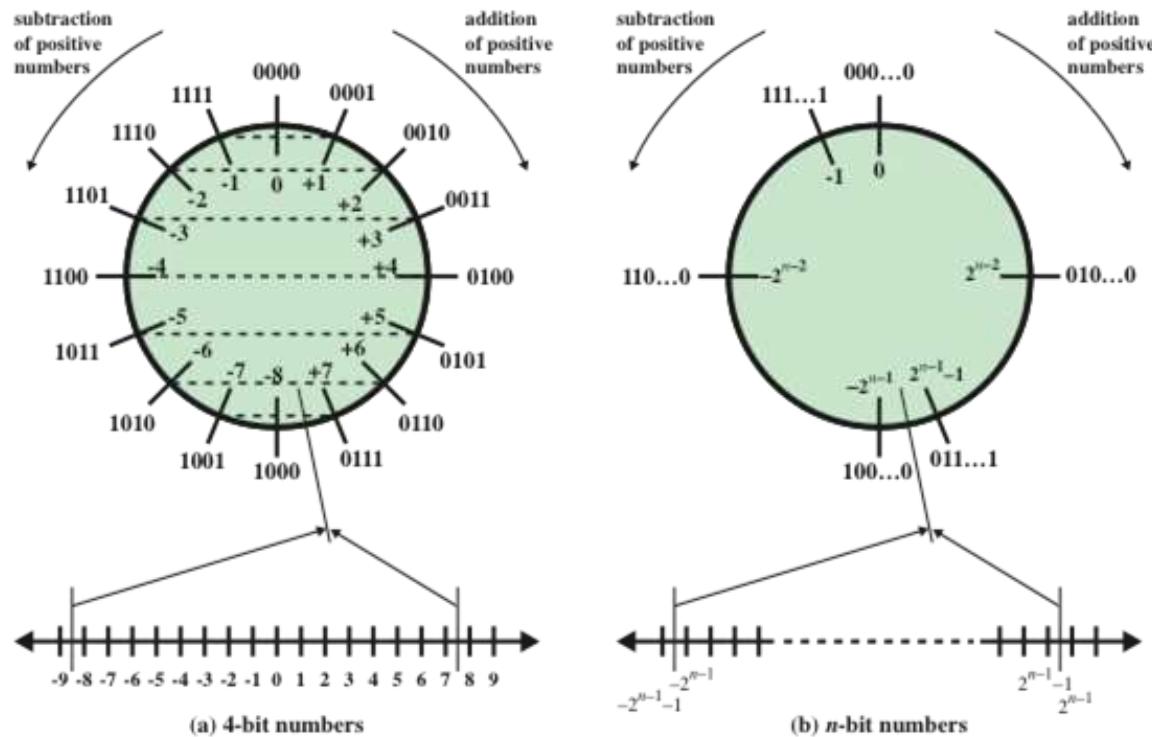


Figure 10.5 Geometric Depiction of Twos Complement Integers



Range Extension

- Range of numbers that can be expressed is extended by increasing the bit length
- In sign-magnitude notation this is accomplished by moving the sign bit to the new leftmost position and fill in with zeros
- This procedure will not work for twos complement negative integers
 - Rule is to move the sign bit to the new leftmost position and fill in with copies of the sign bit
 - For positive numbers, fill in with zeros, and for negative numbers, fill in with ones
 - This is called *sign extension*



Negation

- Twos complement operation
 - Take the Boolean complement of each bit of the integer (including the sign bit)
 - Treating the result as an unsigned binary integer, add 1

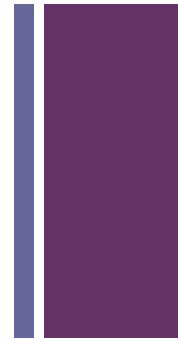
$$\begin{array}{r} +18 = 00010010 \text{ (twos complement)} \\ \text{bitwise complement} = 11101101 \\ + \quad \quad \quad 1 \\ \hline 11101110 = -18 \end{array}$$

- The negative of the negative of that number is itself:

$$\begin{array}{r} -18 = 11101110 \text{ (twos complement)} \\ \text{bitwise complement} = 00010001 \\ + \quad \quad \quad 1 \\ \hline 00010010 = +18 \end{array}$$

+

Negation Special Case 1



0 = 00000000 (twos complement)

Bitwise complement = 11111111

Add 1 to LSB
+ _____ 1

Result 100000000

Overflow is ignored, so:

$$-0 = 0$$

+

Negation Special Case 2

$$-128 = 10000000 \text{ (twos complement)}$$

$$\text{Bitwise complement} = 0111111$$

$$\begin{array}{r} \text{Add 1 to LSB} \\ + \quad \quad \quad 1 \end{array}$$

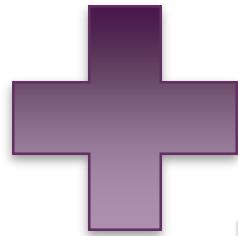
$$\text{Result} \quad \quad \quad 10000000$$

So:

$$-(-128) = -128 \times$$

Monitor MSB (sign bit)

It should change during negation



Addition

| | |
|---|--|
| $\begin{array}{r} 1001 = -7 \\ +0101 = 5 \\ \hline 1110 = -2 \end{array}$ | $\begin{array}{r} 1100 = -4 \\ +0100 = 4 \\ \hline 10000 = 0 \end{array}$ |
| (a) $(-7) + (+5)$ | (b) $(-4) + (+4)$ |
| $\begin{array}{r} 0011 = 3 \\ +0100 = 4 \\ \hline 0111 = 7 \end{array}$ | $\begin{array}{r} 1100 = -4 \\ +1111 = -1 \\ \hline 11011 = -5 \end{array}$ |
| (c) $(+3) + (+4)$ | (d) $(-4) + (-1)$ |
| $\begin{array}{r} 0101 = 5 \\ +0100 = 4 \\ \hline 1001 = \text{Overflow} \end{array}$ | $\begin{array}{r} 1001 = -7 \\ +1010 = -6 \\ \hline 10011 = \text{Overflow} \end{array}$ |
| (e) $(+5) + (+4)$ | (f) $(-7) + (-6)$ |

Figure 10.3 Addition of Numbers in Twos Complement Representation



Overflow

OVERFLOW RULE:

If two numbers are added, and they are both positive or both negative, then overflow occurs if and only if the result has the opposite sign.

Rule



Subtraction

SUBTRACTION RULE:

To subtract one number (subtrahend) from another (minuend), take the twos complement (negation) of the subtrahend and add it to the minuend.

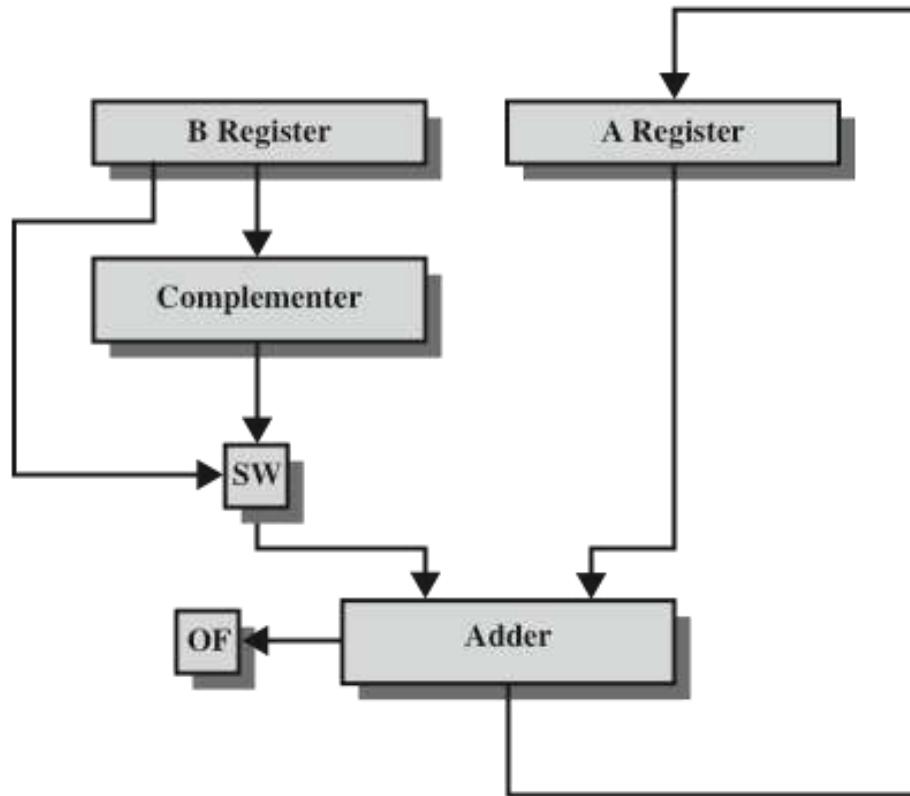
Rule

Subtraction

| | |
|---|--|
| $\begin{array}{r} 0010 = 2 \\ +1001 = -7 \\ \hline 1011 = -5 \end{array}$ | $\begin{array}{r} 0101 = 5 \\ +1110 = -2 \\ \hline 10011 = 3 \end{array}$ |
| (a) $M = 2 = 0010$ $S = 7 = 0111$ $-S = 1001$ | (b) $M = 5 = 0101$ $S = 2 = 0010$ $-S = 1110$ |
| $\begin{array}{r} 1011 = -5 \\ +1110 = -2 \\ \hline 11001 = -7 \end{array}$ | $\begin{array}{r} 0101 = 5 \\ +0010 = 2 \\ \hline 0111 = 7 \end{array}$ |
| (c) $M = -5 = 1011$ $S = 2 = 0010$ $-S = 1110$ | (d) $M = 5 = 0101$ $S = -2 = 1110$ $-S = 0010$ |
| $\begin{array}{r} 0111 = 7 \\ +0111 = 7 \\ \hline 1110 = \text{Overflow} \end{array}$ | $\begin{array}{r} 1010 = -6 \\ +1100 = -4 \\ \hline 10110 = \text{Overflow} \end{array}$ |
| (e) $M = 7 = 0111$ $S = -7 = 1001$ $-S = 0111$ | (f) $M = -6 = 1010$ $S = 4 = 0100$ $-S = 1100$ |

Figure 10.4 Subtraction of Numbers in Twos Complement Representation ($M - S$)

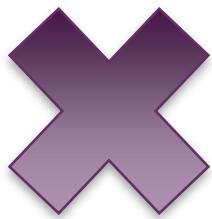
Hardware for Addition and Subtraction



OF = overflow bit

SW = Switch (select addition or subtraction)

Figure 10.6 Block Diagram of Hardware for Addition and Subtraction



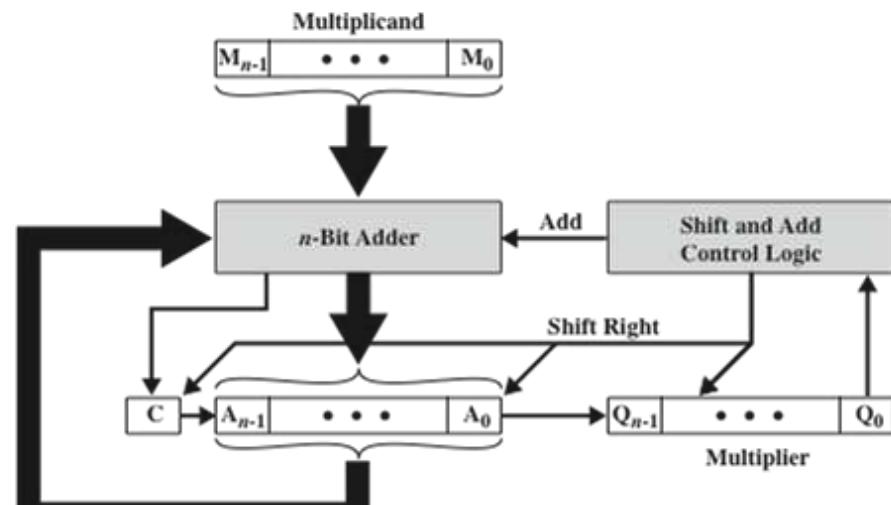
Multiplication

| | | |
|---------------|--------------------------|-------------------------|
| 1011 | Multiplicand (11) | |
| $\times 1101$ | Multiplier (13) | |
| <hr/> 1011 | } | |
| 0000 | | Partial products |
| 1011 | | |
| <hr/> 1011 | | Product (143) |

Figure 10.7 Multiplication of Unsigned Binary Integers



Hardware Implementation of Unsigned Binary Multiplication



(a) Block Diagram

| C | A | Q | M | | Initial Values |
|---|------|------|------|-------|----------------|
| 0 | 0000 | 1101 | 1011 | | |
| 0 | 1011 | 1101 | 1011 | Add | First Cycle |
| | 0101 | 1110 | 1011 | Shift | |
| 0 | 0010 | 1111 | 1011 | Shift | Second Cycle |
| | 1101 | 1111 | 1011 | Add | |
| 0 | 0110 | 1111 | 1011 | Shift | Third Cycle |
| | 0001 | 1111 | 1011 | Add | |
| 1 | 1000 | 1111 | 1011 | Shift | Fourth Cycle |

(b) Example from Figure 9.7 (product in A, Q)

Figure 10.8 Hardware Implementation of Unsigned Binary Multiplication

Flowchart for Unsigned Binary Multiplication

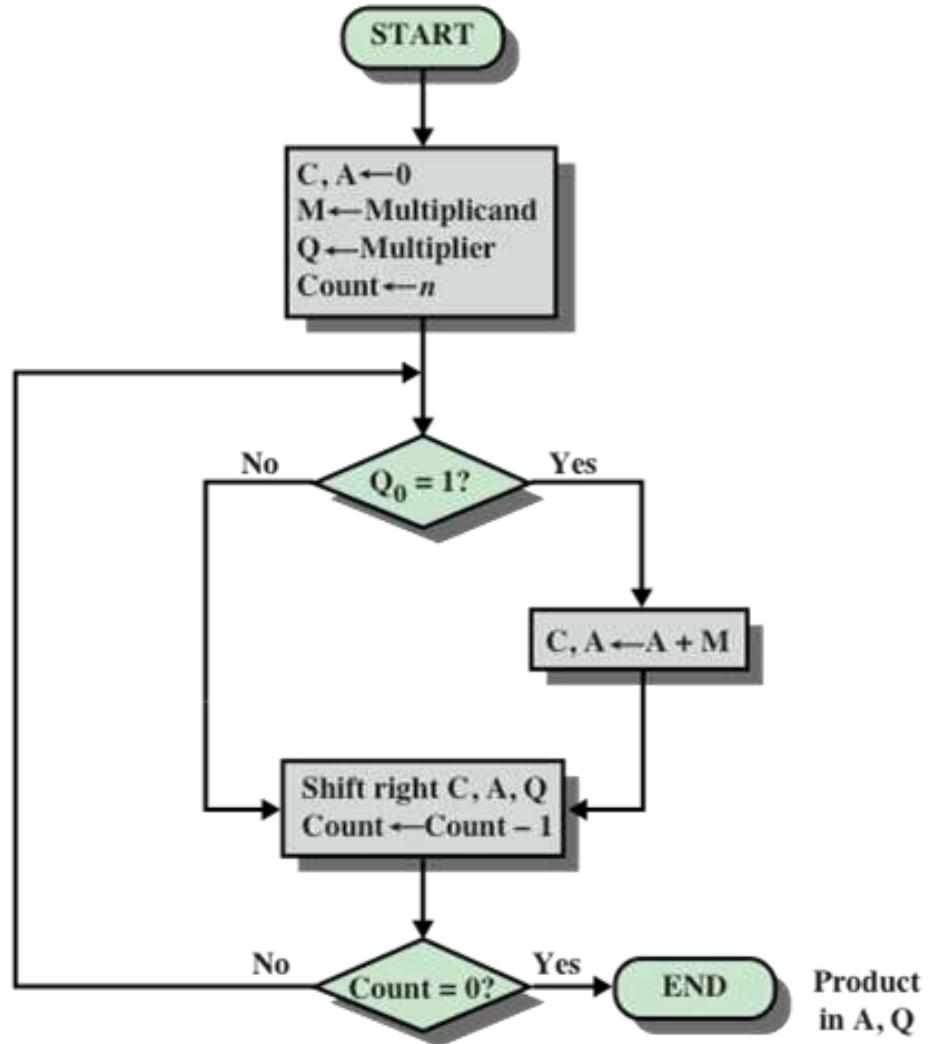


Figure 10.9 Flowchart for Unsigned Binary Multiplication

Twos Complement Multiplication

$$\begin{array}{r} 1011 \\ \times 1101 \\ \hline 00001011 & 1011 \times 1 \times 2^0 \\ 00000000 & 1011 \times 0 \times 2^1 \\ 00101100 & 1011 \times 1 \times 2^2 \\ \hline 01011000 & 1011 \times 1 \times 2^3 \\ \hline 10001111 & \end{array}$$

Figure 10.10 Multiplication of Two Unsigned 4-Bit Integers Yielding an 8-Bit Result

Comparison

$$\begin{array}{r} 1001 \quad (9) \\ \times 0011 \quad (3) \\ \hline 00001001 \quad 1001 \times 2^0 \\ 00010010 \quad 1001 \times 2^1 \\ 00011011 \quad (27) \end{array}$$

(a) Unsigned integers

$$\begin{array}{r} 1001 \quad (-7) \\ \times 0011 \quad (3) \\ \hline 11111001 \quad (-7) \times 2^0 = (-7) \\ 11110010 \quad (-7) \times 2^1 = (-14) \\ 11101011 \quad (-21) \end{array}$$

(b) Twos complement integers

Figure 10.11 Comparison of Multiplication of Unsigned and Twos Complement Integers

Booth's

Algorithm

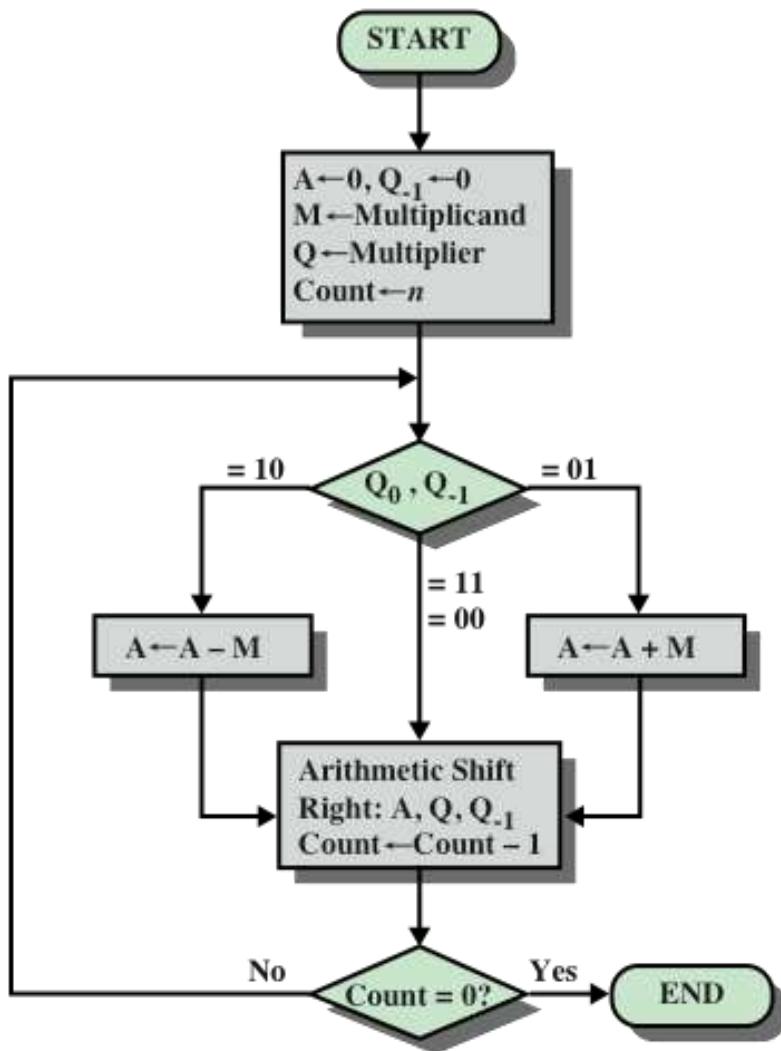


Figure 10.12 Booth's Algorithm for Twos Complement Multiplication

Example of Booth's Algorithm

| A | Q | Q_{-1} | M | Initial Values | |
|------|------|----------|------|----------------------|--------------|
| 0000 | 0011 | 0 | 0111 | | |
| 1001 | 0011 | 0 | 0111 | $A \leftarrow A - M$ | First Cycle |
| 1100 | 1001 | 1 | 0111 | Shift | |
| 1110 | 0100 | 1 | 0111 | Shift | Second Cycle |
| 0101 | 0100 | 1 | 0111 | $A \leftarrow A + M$ | |
| 0010 | 1010 | 0 | 0111 | Shift | Third Cycle |
| 0001 | 0101 | 0 | 0111 | Shift | |
| | | | | | Fourth Cycle |

Figure 10.13 Example of Booth's Algorithm (7× 3)

Examples Using Booth's Algorithm

| | |
|---|---|
| $\begin{array}{r} 0111 \\ \times 0011 \\ \hline 11111001 \\ 00000000 \\ \hline 000111 \\ \hline 00010101 \end{array}$ <p>(0) 1-0 1-1 0-1 (21)</p> | $\begin{array}{r} 0111 \\ \times 1101 \\ \hline 11111001 \\ 0000111 \\ \hline 111001 \\ \hline 11101011 \end{array}$ <p>(0) 1-0 0-1 1-0 (-21)</p> |
|---|---|

(a) $(7) \times (3) = (21)$

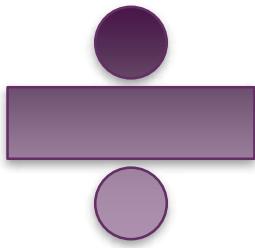
(b) $(7) \times (-3) = (-21)$

| | |
|--|---|
| $\begin{array}{r} 1001 \\ \times 0011 \\ \hline 00000111 \\ 00000000 \\ \hline 111001 \\ \hline 11101011 \end{array}$ <p>(0) 1-0 1-1 0-1 (-21)</p> | $\begin{array}{r} 1001 \\ \times 1101 \\ \hline 00000111 \\ 1111001 \\ 000111 \\ \hline 00010101 \end{array}$ <p>(0) 1-0 0-1 1-0 (21)</p> |
|--|---|

(c) $(-7) \times (3) = (-21)$

(d) $(-7) \times (-3) = (21)$

Figure 10.14 Examples Using Booth's Algorithm



Division

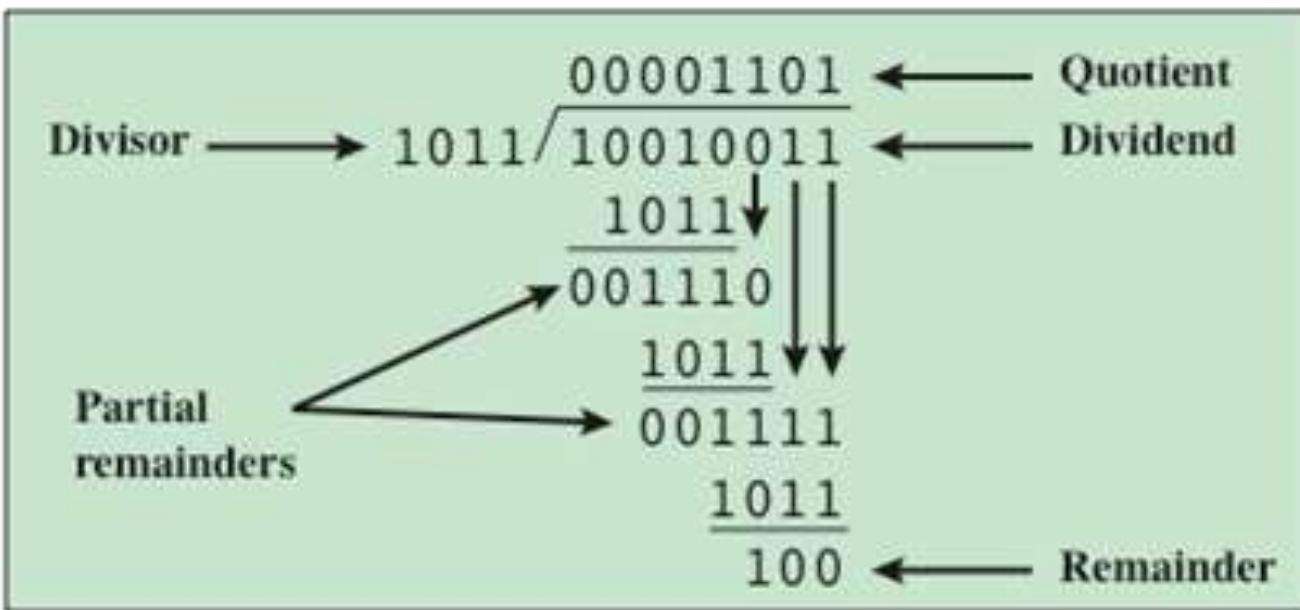


Figure 10.15 Example of Division of Unsigned Binary Integers

+

Flowchart for Unsigned Binary Division

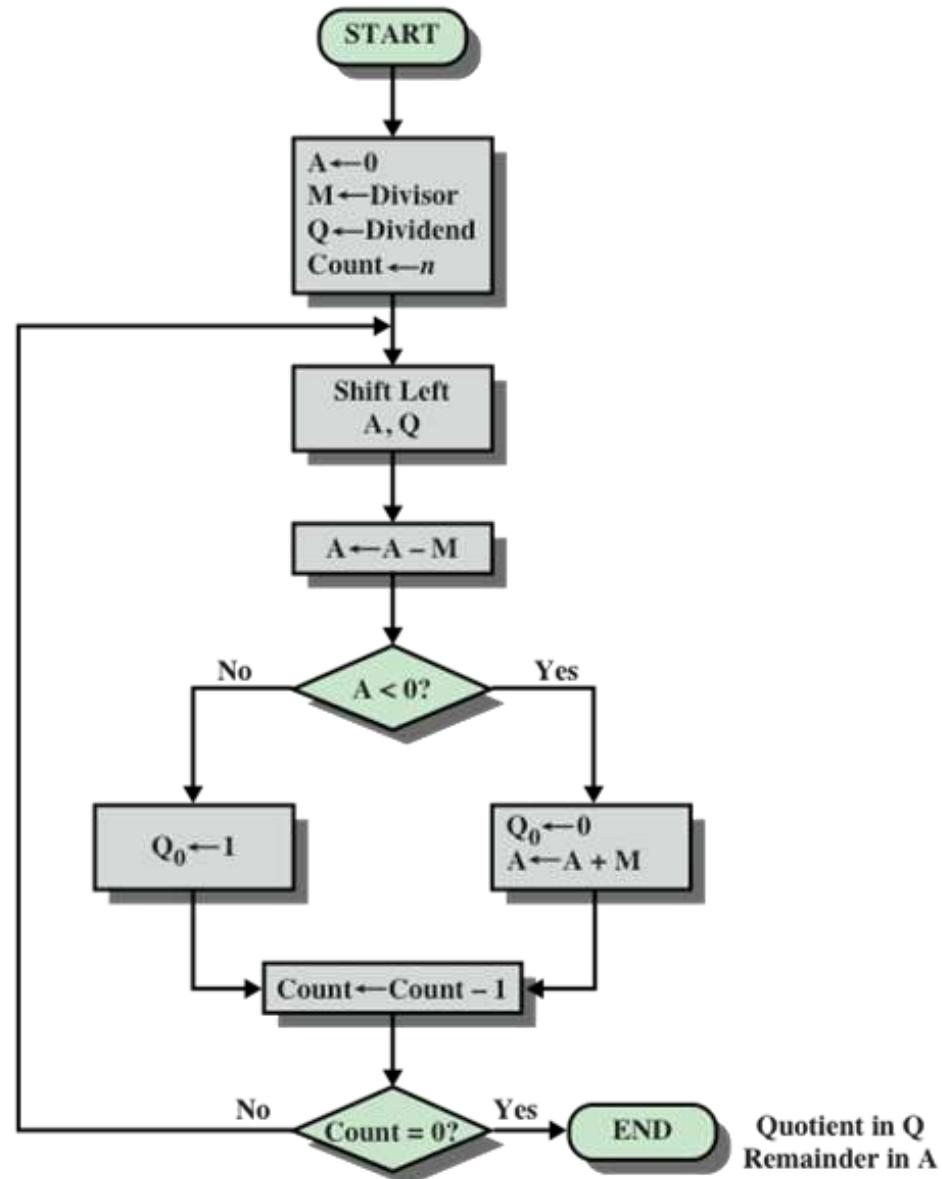


Figure 10.16 Flowchart for Unsigned Binary Division



Flowchart for Unsigned Binary Division

The algorithm can be summarized as follows:

1. Load the twos complement of the divisor into the M register; that is, the M register contains the negative of the divisor.
Load the dividend into the A, Q registers. The dividend must be expressed as a *2n-bit positive number*. Thus, for example, the 4-bit 0111 becomes 00000111.
2. Shift A, Q left 1 bit position.



Flowchart for Unsigned Binary Division

3. Perform $A = A - M$. This operation subtracts the divisor from the contents of A.
4.
 - a. If the result is nonnegative (most significant bit of A = 0), then set $Q_0 = 1$.
 - b. If the result is negative (most significant bit of A = 1), then set $Q_0 = 0$ and restore the previous value of A.
5. Repeat steps 2 through 4 as many times as there are bit positions in Q.
6. The remainder is in A and the quotient is in Q.

Example of Restoring Twos Complement Division

| A | Q | |
|-------------------------------------|------|--|
| 0000 | 0111 | Initial value |
| 0000 <u>1101</u> 1101 0000 | 1110 | Shift Use twos complement of 0011 for subtraction Subtract Restore, set $Q_0 = 0$ |
| 0001 <u>1101</u> 1110 0001 | 1100 | Shift Subtract Restore, set $Q_0 = 0$ |
| 0011 <u>1101</u> 0000 | 1000 | Shift |
| 0001 <u>1101</u> 1110 0001 | 1001 | Subtract, set $Q_0 = 1$ |
| 0001 <u>1101</u> 1110 0001 | 0010 | Shift Subtract Restore, set $Q_0 = 0$ |

Figure 10.17 Example of Restoring Twos Complement Division (7/3)

Instruction Sets : Characteristics and Functions

Chapter 12

Machine Instruction Characteristics

Machine Instruction Characteristics

Usually a machine instruction has the following components:

- **Operation Code / Opcode :** ADD AX,BX
- **Source Operand Reference:** MOV AX, BX
- **Result Operand Reference:** MOV AX, BX
- **Next Instruction Reference:**
 - An instruction may be a **direct branching instruction** like : JMP Label
 - After all non - branching instructions, the PC gets incremented.

Machine Instruction Characteristics

1. The nature of the operands/data source and dest. in a machine instruction :

- Memory : MOV AX, #1
- Register : MOV AX, BX
- Immediate ADD AX, 1
- Input output Device :
 - Non - memory mapped i/o device : MOV AX, Port 1 (Similar to register)
 - Memory –mapped i/o device : MOV AX, # 12

Machine Instruction Characteristics

2. Instruction Representation:

There can be different representations.

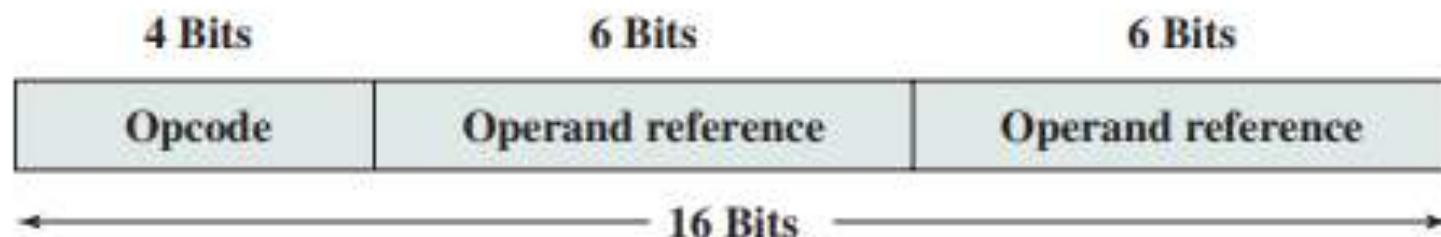


Figure 12.2 A Simple Instruction Format

Machine Instruction Characteristics

3. Generation of machine level languages from high level languages

Consider the following instruction (in high level): $X=X+Y;$

The compiler , from a very high level perspective does this:

- Assignment of memory locations for the different variables:
 - X - memory location 513
 - Y - memory location 514
- Generation of the assembly code
 - Load AX,#513
 - Load BX,#514
 - ADD AX,BX
 - STOR #513,AX

Machine Instruction Characteristics

4. The tasks carried out by machine language instructions

Data processing: Arithmetic and logic instructions.

Add AX,BX

Data storage: Storage of data in memory/registers/ports

LOAD AX,#19 // load AX from #19

STOR AX,#19 // store in #19 from AX

Data movement: Movement of data within processor's registers.

MOV AX,BX

Control: Test and branch instructions

CMP AX,BX

JGE LABEL

Machine Instruction Characteristics

5. The Number of operands

It usually is concerned with 4 operands :

- **Operand SrcOp1 SrcOp1 DestOp NextAddr**

However It can be more :

- Some machines can have instructions dealing with 17 opearands at a time.
- (e.g : The ARM architecture)

Machine Instruction Characteristics

6. One High level instruction, different addressing formats:

$$\text{Programs to Execute } Y = \frac{A - B}{C + (D \times E)}$$

| <u>Instruction</u> | <u>Comment</u> |
|--------------------|---------------------------|
| SUB Y, A, B | $Y \leftarrow A - B$ |
| MPY T, D, E | $T \leftarrow D \times E$ |
| ADD T, T, C | $T \leftarrow T + C$ |
| DIV Y, Y, T | $Y \leftarrow Y \div T$ |

(a) Three-address instructions

| <u>Instruction</u> | <u>Comment</u> |
|--------------------|---------------------------|
| MOVE Y, A | $Y \leftarrow A$ |
| SUB Y, B | $Y \leftarrow Y - B$ |
| MOVE T, D | $T \leftarrow D$ |
| MPY T, E | $T \leftarrow T \times E$ |
| ADD T, C | $T \leftarrow T + C$ |
| DIV Y, T | $Y \leftarrow Y \div T$ |

(b) Two-address instructions

| <u>Instruction</u> | <u>Comment</u> |
|--------------------|-----------------------------|
| LOAD D | $AC \leftarrow D$ |
| MPY E | $AC \leftarrow AC \times E$ |
| ADD C | $AC \leftarrow AC + C$ |
| STOR Y | $Y \leftarrow AC$ |
| LOAD A | $AC \leftarrow A$ |
| SUB B | $AC \leftarrow AC - B$ |
| DIV Y | $AC \leftarrow AC \div Y$ |
| STOR Y | $Y \leftarrow AC$ |

(c) One-address instructions

Machine Instruction Characteristics

6. One High level instruction, different addressing formats:

$$\text{Programs to Execute } Y = \frac{A - B}{C + (D \times E)}$$

BUT THE MOST IMPORTANT ONE IS:

ZERO ADDRESS INSTRUCTION FORMAT

Machine Instruction Characteristics

7. Tradeoff between instruction formats with more addresses and instruction formats with less addresses

- Instructions with more addresses mean less code is required to be written. (we will see this in detail in chapter 15).
- Ensures flexibility for the programmer (if he codes in assembly language).
- Complex hardware is required.

Instruction Set Design

1. What questions will you be asking yourself when building your processor?

- **Operation repertoire** – how many and which operations
 - In your DSD project for example you may choose ADD, SUB, AND , OR
- **Data Types** – how the memory and register values will be interpreted
 - 1001 – can be considered as **+9** if your instruction thinks of it as an unsigned integer.
 - 1001 – can be considered as **-7** if your instruction thinks of it as a signed integer.
- **Instruction Format** – length , opcode size, operand size

1. What questions will you be asking yourself when building your processor?

- **Registers** – how many registers you will want to keep
 - **Addressing** – the mode of addressing you want to follow.
 - It may be direct addressing for some opcodes like ADD #1, #2
 - It may be direct addressing for some opcodes like ADD_ #1, #2
- Or
- Your entire processor may just work based in direct addressing
 - Your entire processor may just work based in indirect addressing

2. Some sample operations that you may consider while designing your processor

| Type | Operation Name | Description |
|---------------|-----------------|---|
| Data transfer | Move (transfer) | Transfer word or block from source to destination |
| | Store | Transfer word from processor to memory |
| | Load (fetch) | Transfer word from memory to processor |
| | Exchange | Swap contents of source and destination |
| | Clear (reset) | Transfer word of 0s to destination |
| | Set | Transfer word of 1s to destination |
| | Push | Transfer word from source to top of stack |
| | Pop | Transfer word from top of stack to destination |

2. Some sample operations that you may consider while designing your processor (cont.)

| | | |
|------------|-----------|---------------------------------------|
| Arithmetic | Add | Compute sum of two operands |
| | Subtract | Compute difference of two operands |
| | Multiply | Compute product of two operands |
| | Divide | Compute quotient of two operands |
| | Absolute | Replace operand by its absolute value |
| | Negate | Change sign of operand |
| | Increment | Add 1 to operand |
| | Decrement | Subtract 1 from operand |

2. Some sample operations that you may consider while designing your processor (cont.)

| | | |
|---------|-----------------------|--|
| Logical | AND | Perform logical AND |
| | OR | Perform logical OR |
| | NOT | (complement) Perform logical NOT |
| | Exclusive-OR | Perform logical XOR |
| | Test | Test specified condition; set flag(s) based on outcome |
| | Compare | Make logical or arithmetic comparison of two or more operands; set flag(s) based on outcome |
| | Set Control Variables | Class of instructions to set controls for protection purposes, interrupt handling, timer control, etc. |
| | Shift | Left (right) shift operand, introducing constants at end |
| | Rotate | Left (right) shift operand, with wraparound end |

2. Some sample operations that you may consider while designing your processor (cont.)

| | | |
|---------------------|--------------------|---|
| Transfer of control | Jump (branch) | Unconditional transfer; load PC with specified address |
| | Jump Conditional | Test specified condition; either load PC with specified address or do nothing, based on condition |
| | Jump to Subroutine | Place current program control information in known location; jump to specified address |
| | Return | Replace contents of PC and other register from known location |
| | Execute | Fetch operand from specified location and execute as instruction; do not modify PC |
| | Skip | Increment PC to skip next instruction |
| | Skip Conditional | Test specified condition; either skip or do nothing based on condition |
| | Halt | Stop program execution |
| | Wait (hold) | Stop program execution; test specified condition repeatedly; resume execution when condition is satisfied |
| | No operation | No operation is performed, but program execution is continued |

3. More on transfer of Control

Remember the instruction JGE LABEL?

And a function that we saw in Chapter 14 (PROC MyFunc)

Instructions for Transfer of Control are required because:

- **Decision making**
- Use an instruction **more than once**
- Help the **programmer** in coding

3. More on transfer of Control (Cont.)

Mainly 3 types of instructions for transfer of control

- a) Branch
- b) Skip
- c) Procedure Call

a. Branch Instructions:

Branching instruction characteristics:

- It must have an **address**.
- It can be **conditional or unconditional**
- **Unconditional branching example :**
 - JMP Label
- **Conditional branching example :**
 - [ADD] [CMP] [SUB] AX,BX
 - JE X
 - or
 - BRE R1,R2, Label
- You can branch in the forward as well as the backward direction

b. Skip Instructions

Another transfer of control instruction.

MOV AX, 100

LOOP_START_LABEL

TASK 1

TASK 2

DEC AX

ISZ AX // if AX is zero, skip next instruction, else do not

JMP LOOP_START_LABEL // similar to goto

out of loop task 1

out of loop task 1

c. Procedure Call Instructions

- Economy and modularity
- Consists of a call instruction and a return instruction
- Function calls can be made from diff locations
- Nesting is of function calls is possible too.

c. Procedure Call Instructions (cont.)

- Note that in order to **pass parameters** to an assembly language function or to **receive values** back from it.
- The assembly language **programmer** must take the necessary steps.
- It is **not simple** like the high level languages.

c. Procedure Call Instructions (cont.)

- Three approaches for passing and receiving data from a function:
 - i. Using registers
 - ii. Using some memory location
 - iii. Using the stack

i. Using registers



The first approach:

We want to give the function a number.

We want it to return the twice the number.

We give it the number using the AX register.

We get the output using the BX register.



Port



Task 1

Task 2

MOV AX, 2

CALL PYPROC

MOV BX, Port1

Task1

PROC MYPROC:

ADD AX, AX

MOV BX, AX

END MYPROC

i. Using registers

The first approach:

We want to give the function a number.

We want it to return the twice the number.

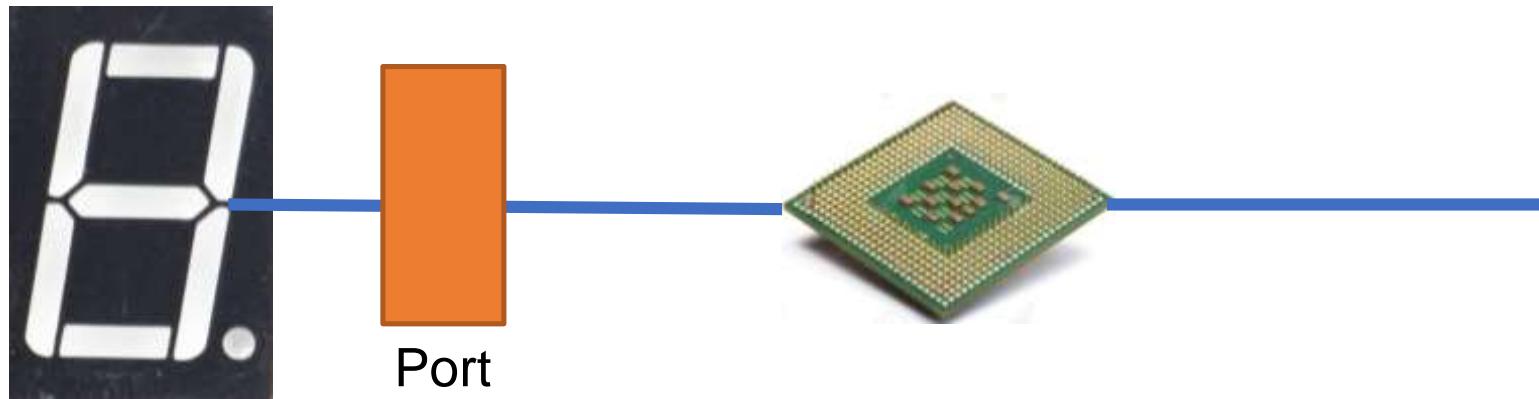
We give it the number using the AX register.

We get the output using the BX register.



```
Task 1  
Task 2  
MOV AX, 2  
CALL PYPROC  
MOV BX, Port1  
Task1
```

```
PROC MYPROC:  
ADD AX, AX  
MOV BX, AX  
END MYPROC
```



i. Using registers

The first approach:

We want to give the function a number.

We want it to return the twice the number.

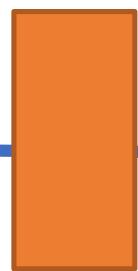
We give it the number using the AX register.

We get the output using the BX register.

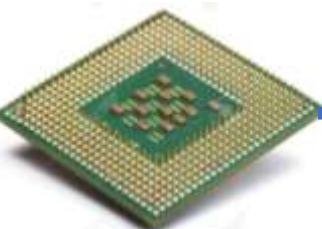


```
Task 1  
Task 2  
MOV AX, 2  
CALL PYPROC  
MOV BX, Port1  
Task1
```

```
PROC MYPROC:  
ADD AX, AX  
MOV BX, AX  
END MYPROC
```



Port



i. Using registers

The first approach:

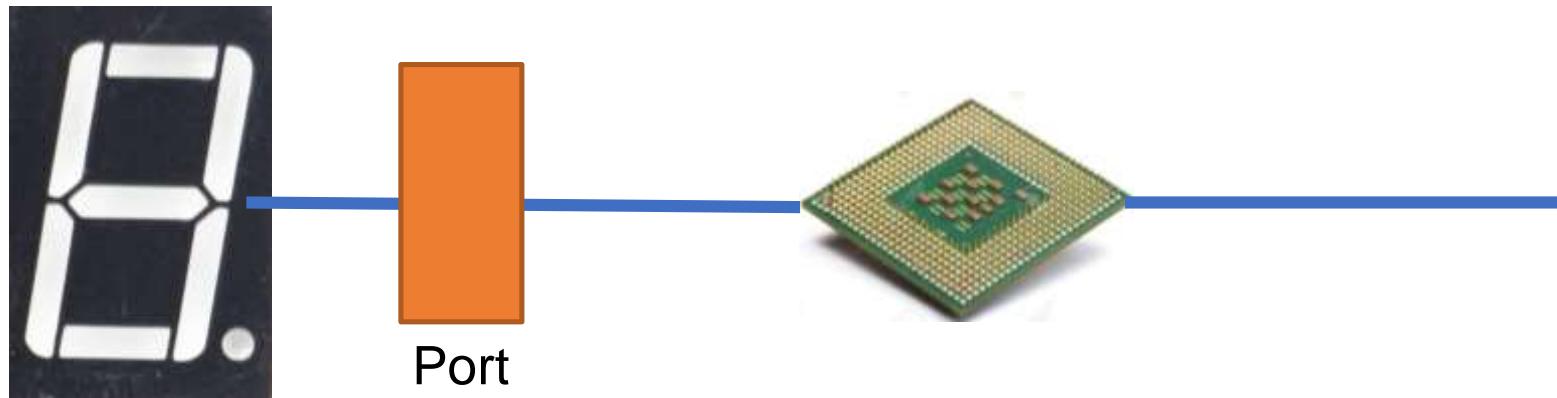
We want to give the function a number.



We want it to return the twice the number.

We give it the number using the AX register.

We get the output using the BX register.



Task 1
Task 2
MOV AX, 2
CALL PYPROC
MOV BX, Port1
Task1

PROC MYPROC:
ADD AX, AX
MOV BX, AX
END MYPROC

i. Using registers

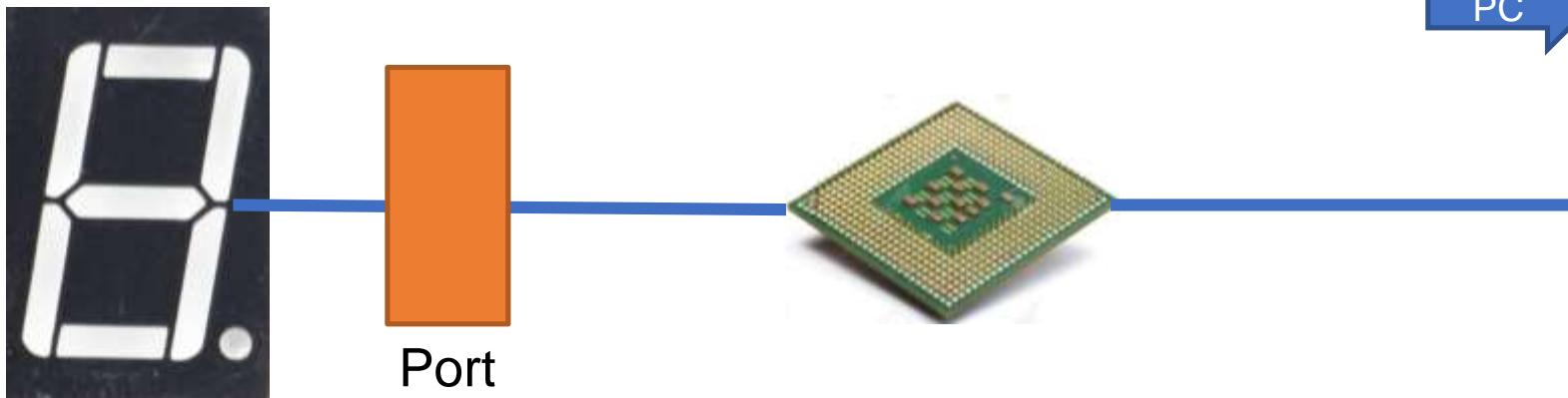
The first approach:

We want to give the function a number.

We want it to return the twice the number.

We give it the number using the AX register.

We get the output using the BX register.



```
Task 1  
Task 2  
MOV AX, 2  
CALL PYPROC  
MOV BX, Port1  
Task1
```

```
PROC MYPROC:  
ADD AX, AX  
MOV BX, AX  
END MYPROC
```

i. Using registers

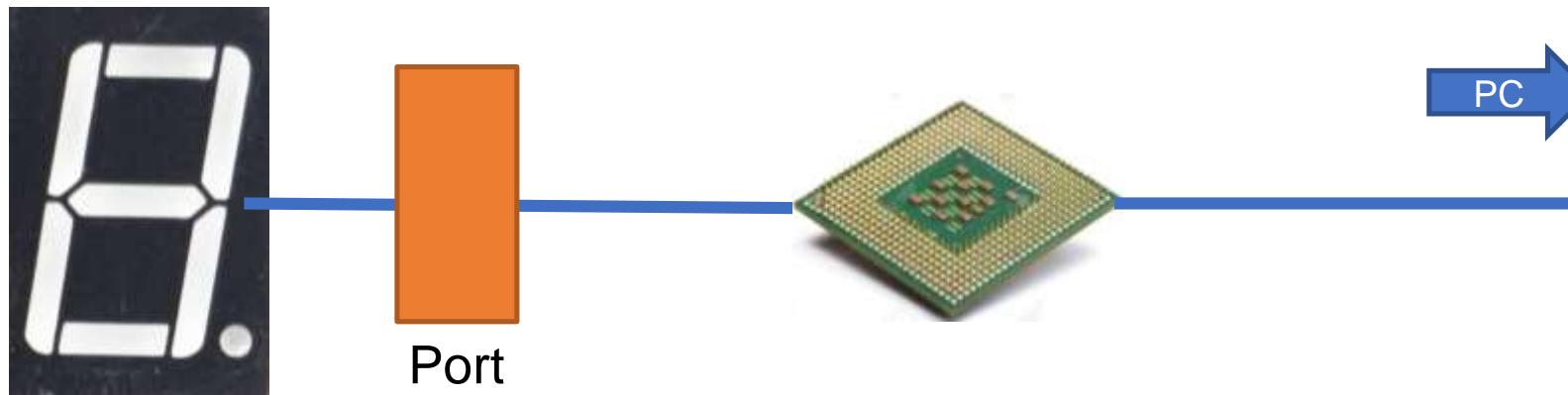
The first approach:

We want to give the function a number.

We want it to return the twice the number.

We give it the number using the AX register.

We get the output using the BX register.



```
Task 1  
Task 2  
MOV AX, 2  
CALL PYPROC  
MOV BX, Port1  
Task1
```

```
PROC MYPROC:  
ADD AX, AX  
MOV BX, AX  
END MYPROC
```

i. Using registers

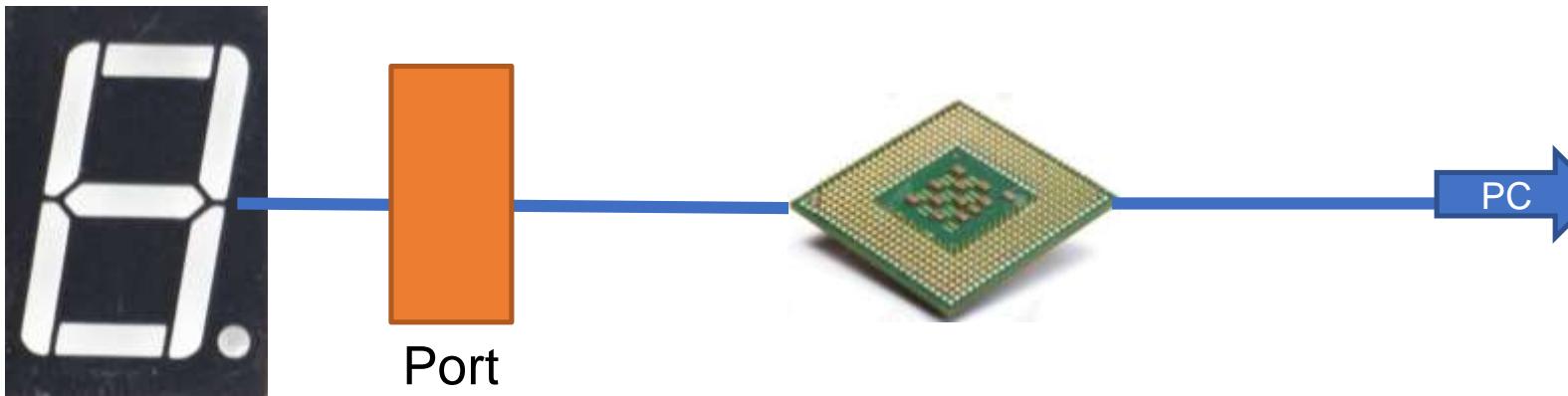
The first approach:

We want to give the function a number.

We want it to return the twice the number.

We give it the number using the AX register.

We get the output using the BX register.



```
Task 1  
Task 2  
MOV AX, 2  
CALL PYPROC  
MOV BX, Port1  
Task1
```

```
PROC MYPROC:  
ADD AX, AX  
MOV BX, AX  
END MYPROC
```

i. Using registers

The first approach:

We want to give the function a number.

We want it to return the twice the number.

We give it the number using the AX register.

We get the output using the BX register.

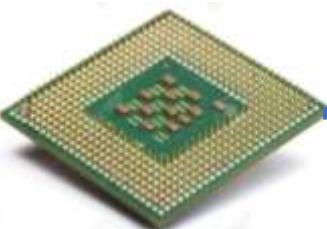
PC →

```
Task 1  
Task 2  
MOV AX, 2  
CALL PYPROC  
MOV BX, Port1  
Task1
```

```
PROC MYPROC:  
ADD AX, AX  
MOV BX, AX  
END MYPROC
```



Port



i. Using registers

The first approach:

We want to give the function a number.

We want it to return the twice the number.

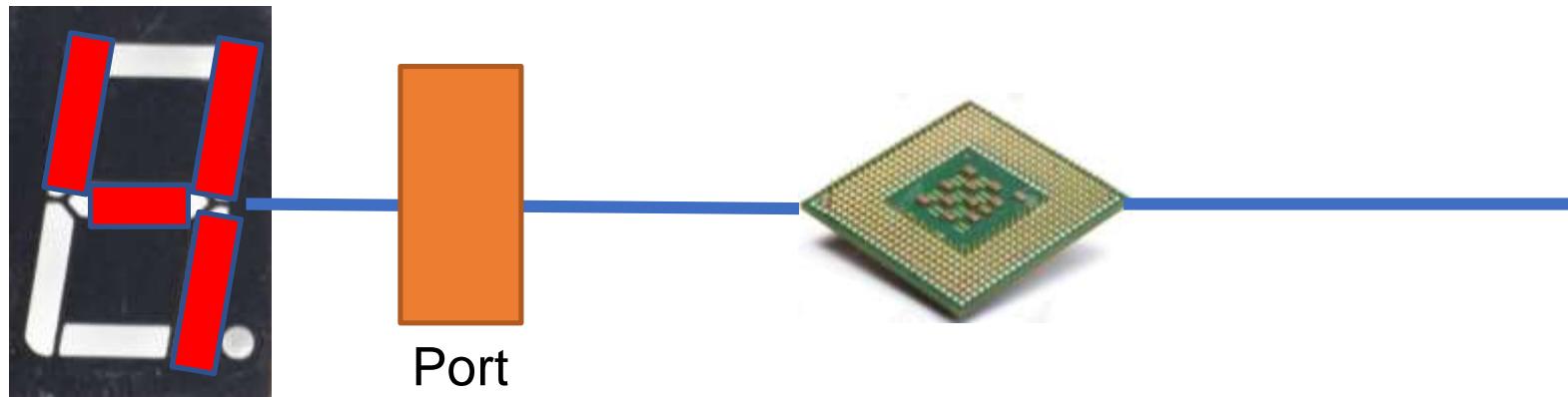
We give it the number using the AX register.

We get the output using the BX register.



```
Task 1  
Task 2  
MOV AX, 2  
CALL PYPROC  
MOV BX, Port1  
Task1
```

```
PROC MYPROC:  
ADD AX, AX  
MOV BX, AX  
END MYPROC
```



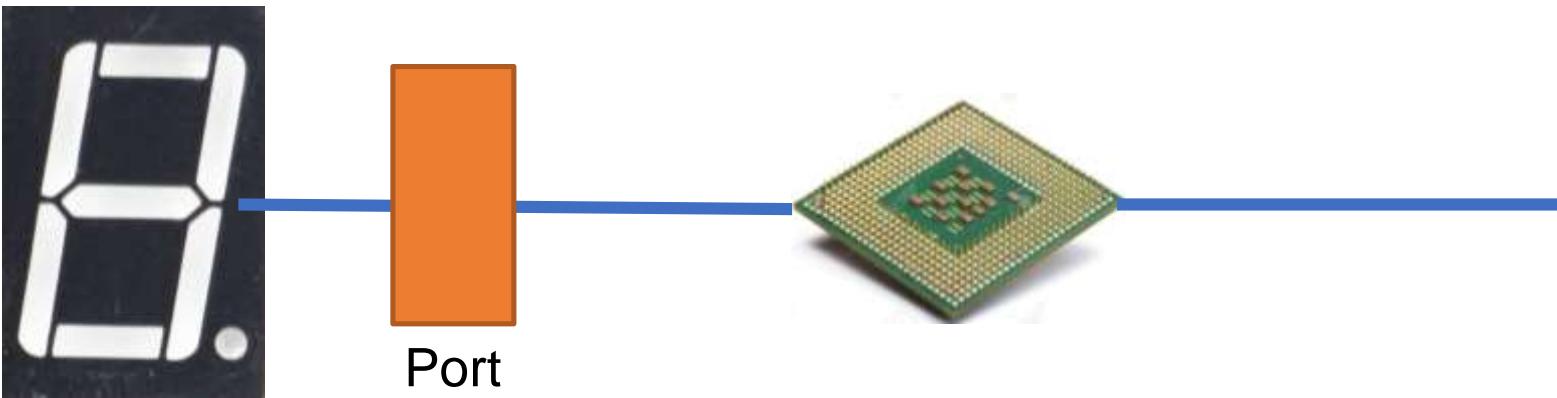
i. Using registers

The first approach:

What does the processor do when it encounters:

- CALL MYPROC
- and END MYPROC ?

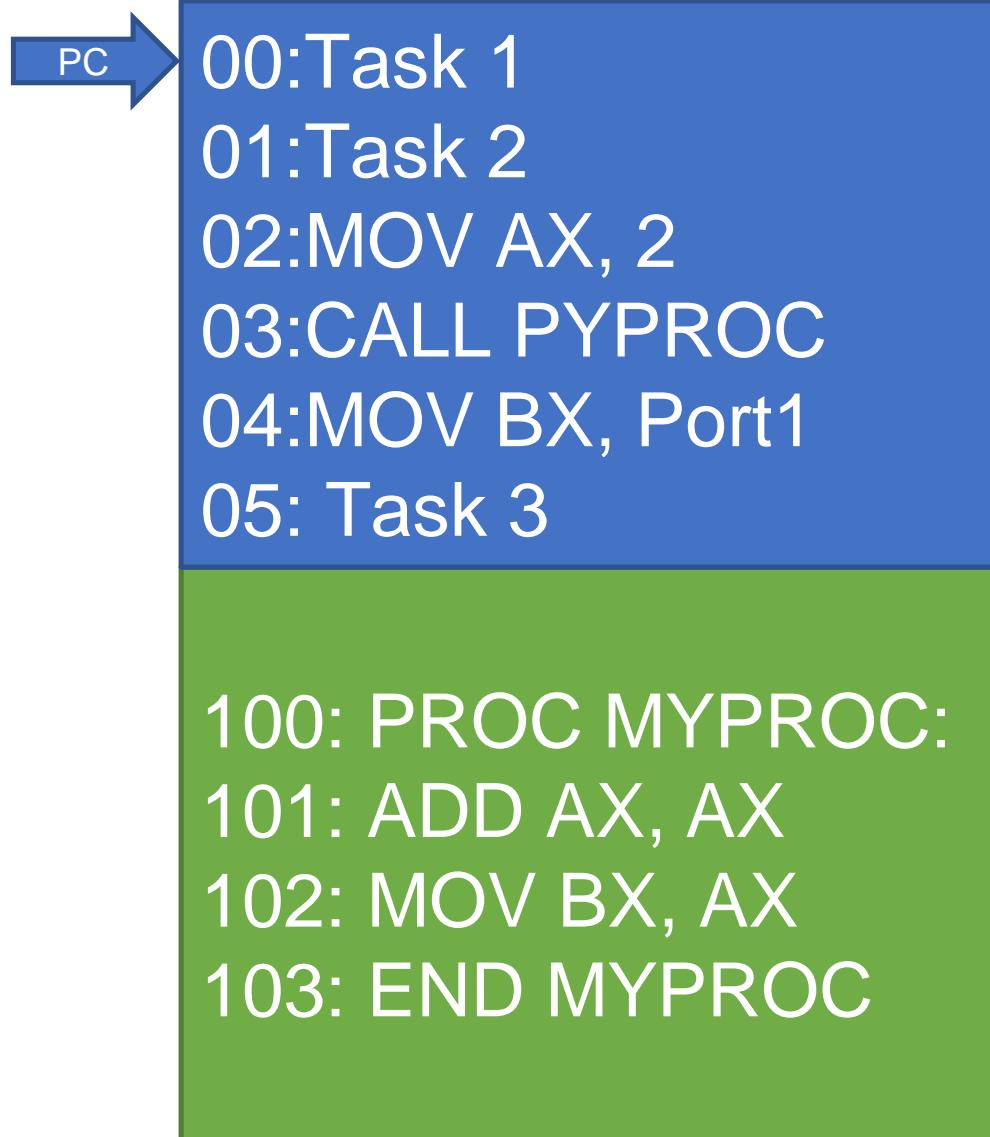
A stack is running in the background for this.



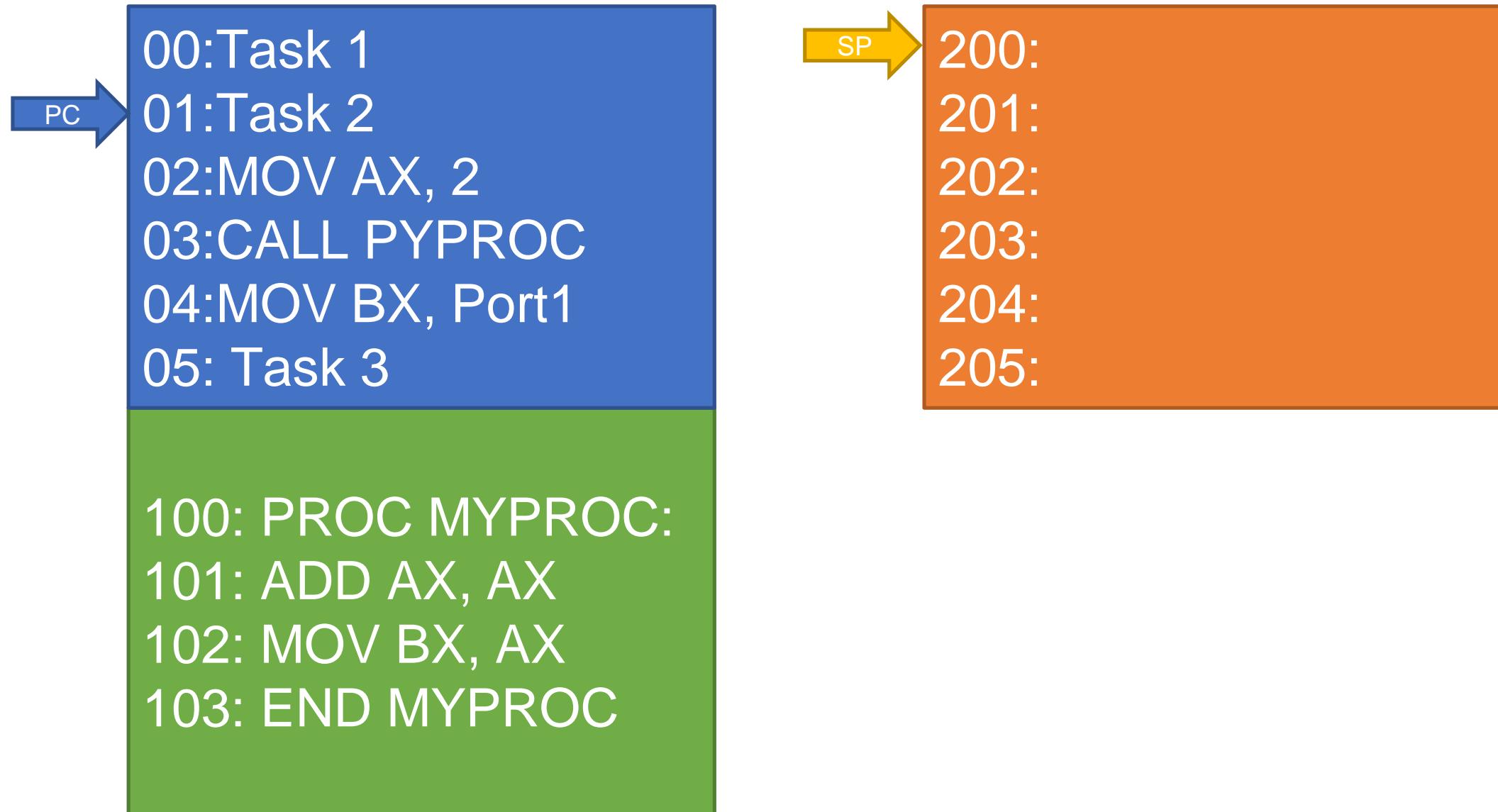
Task 1
Task 2
MOV AX, 2
CALL PYPROC
MOV BX, Port1
Task1

PROC MYPROC:
ADD AX, AX
MOV BX, AX
END MYPROC

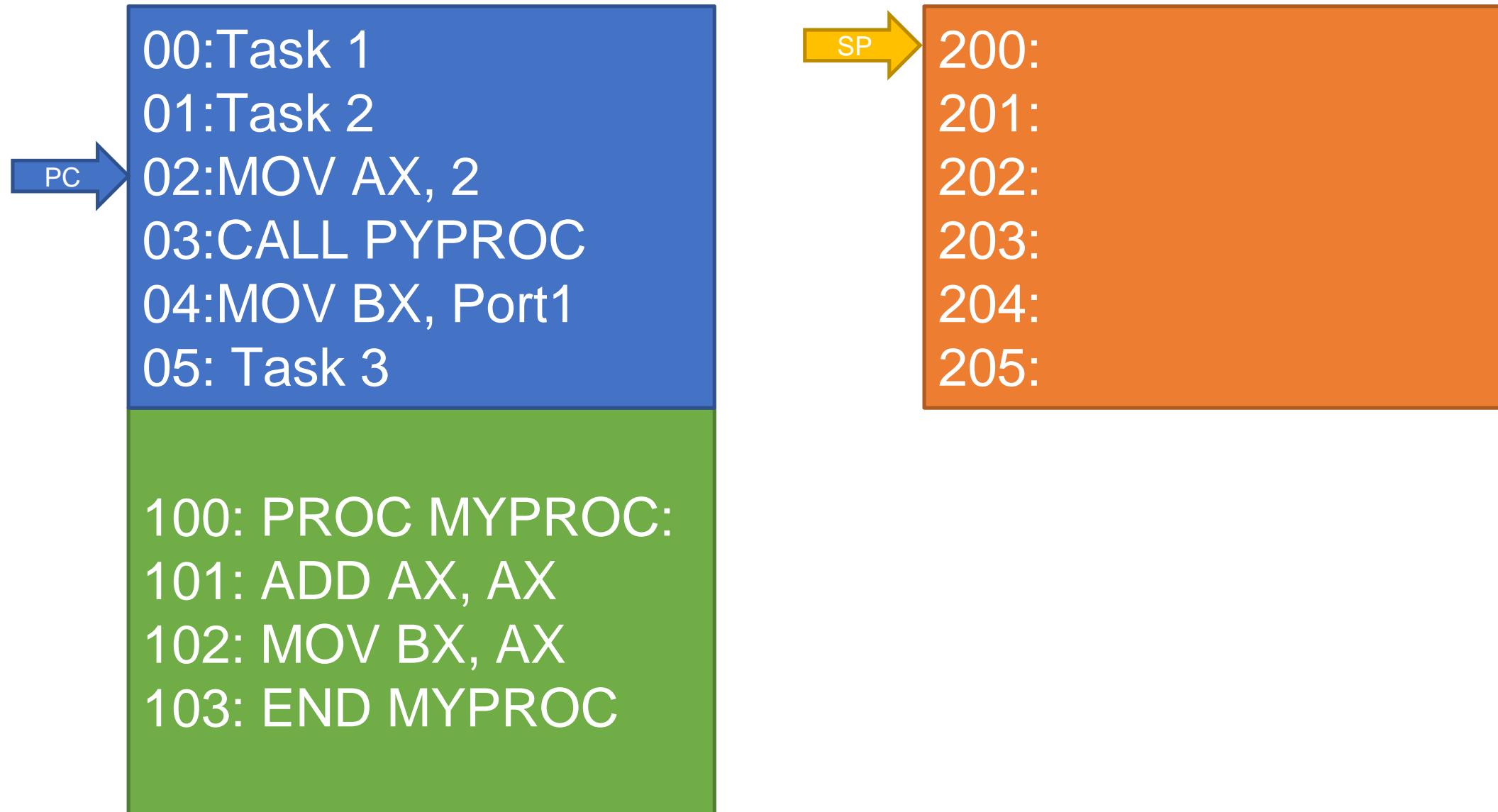
The Stack in function calls



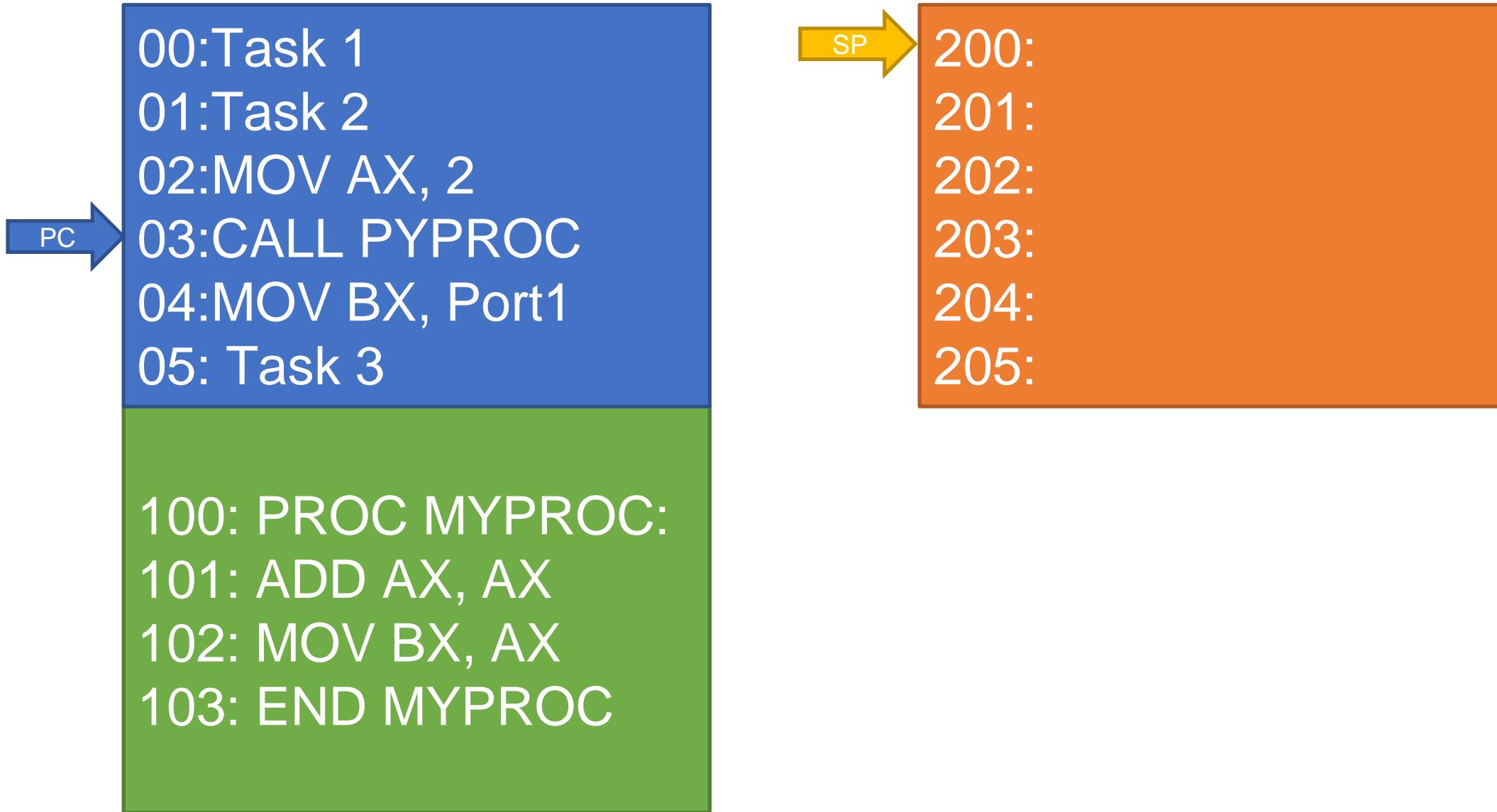
The Stack in function calls (cont.)



The Stack in function calls (cont.)



The Stack in function calls (cont.)



The Stack in function calls (cont.)

```
00:Task 1  
01:Task 2  
02:MOV AX, 2  
03:CALL PYPROC  
04:MOV BX, Port1  
05: Task 3
```



```
200: 04 //the return address  
201:  
202:  
203:  
204:  
205:
```



```
100: PROC MYPROC:  
101: ADD AX, AX  
102: MOV BX, AX  
103: END MYPROC
```

The Stack in function calls (cont.)

```
00:Task 1  
01:Task 2  
02:MOV AX, 2  
03:CALL PYPROC  
04:MOV BX, Port1  
05: Task 3
```



```
200: 04 //the return address  
201:  
202:  
203:  
204:  
205:
```



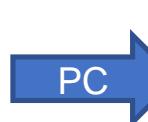
```
100: PROC MYPROC:  
101: ADD AX, AX  
102: MOV BX, AX  
103: END MYPROC
```

The Stack in function calls (cont.)

```
00:Task 1  
01:Task 2  
02:MOV AX, 2  
03:CALL PYPROC  
04:MOV BX, Port1  
05: Task 3
```

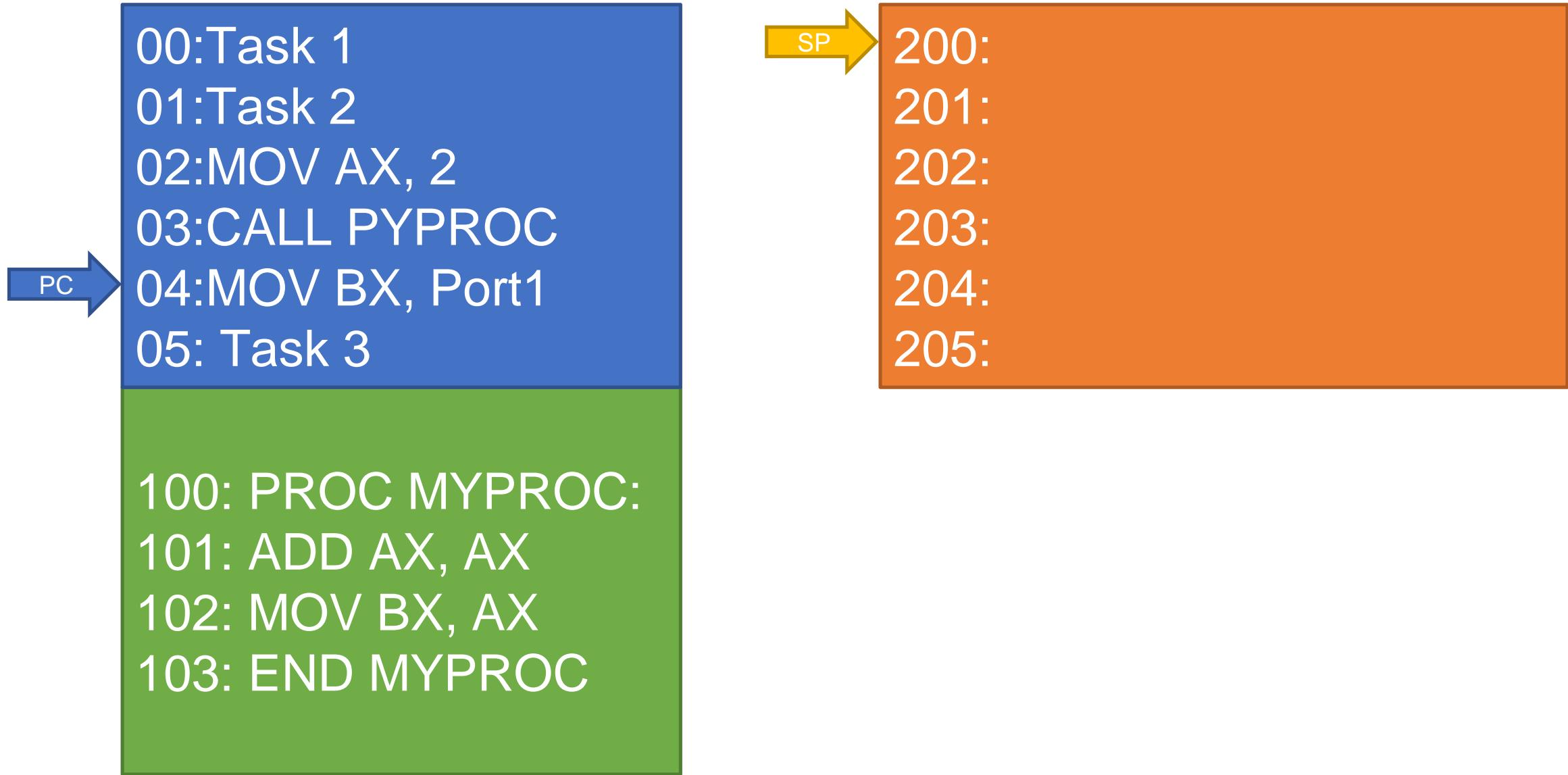


```
200: 04 //the return address  
201:  
202:  
203:  
204:  
205:
```

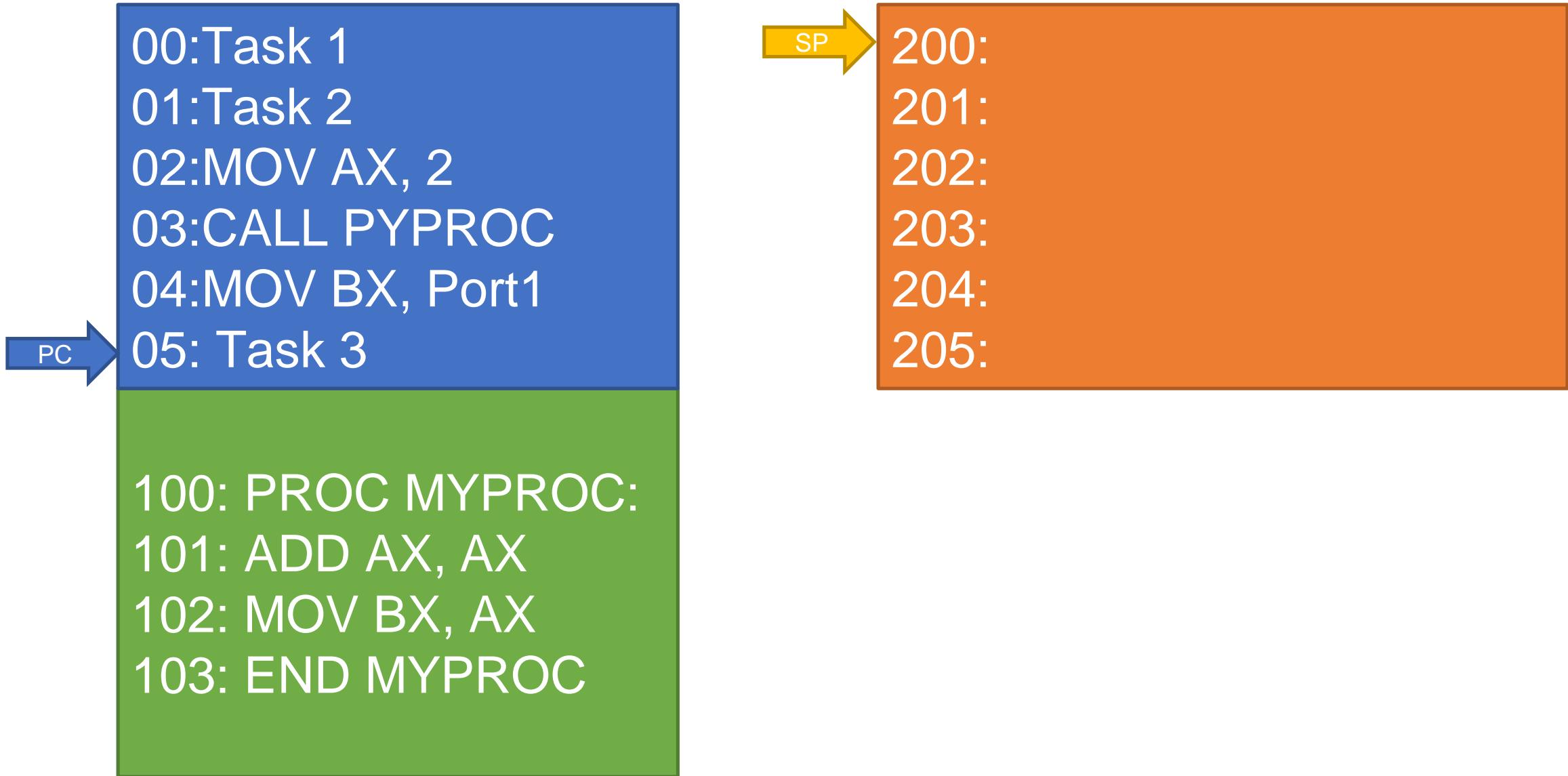


```
100: PROC MYPROC:  
101: ADD AX, AX  
102: MOV BX, AX  
103: END MYPROC
```

The Stack in function calls (cont.)



The Stack in function calls (cont.)



ii. Using dedicated memory space

The second approach:

We want to give the function a number.

We want it to return the twice the number.

We give it the number using the 12th mem locn.

We get the output using the 14th mem locn.



Task 1
Task 2
MOV #12, 2
CALL PYPROC
MOV #14, Port1
Task1

PROC MYPROC:
ADD #12, #12
MOV #14, #12
END MYPROC



Port

ii. Using dedicated memory space

The second approach:

We want to give the function a number.

We want it to return the twice the number.

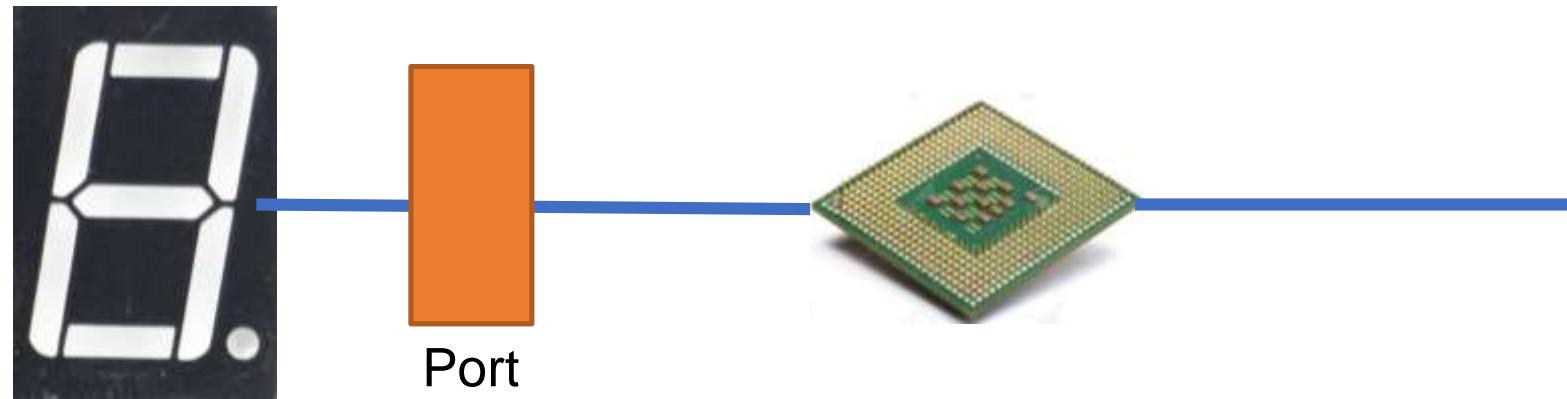
We give it the number using the 12th mem locn.

We get the output using the 14th mem locn.



```
Task 1  
Task 2  
MOV #12, 2  
CALL PYPROC  
MOV #14, Port1  
Task1
```

```
PROC MYPROC:  
ADD #12, #12  
MOV #14, #12  
END MYPROC
```



ii. Using dedicated memory space

The second approach:



We want to give the function a number.

We want it to return the twice the number.

We give it the number using the 12th mem locn.

We get the output using the 14th mem locn.

```
Task 1  
Task 2  
MOV #12, 2  
CALL PYPROC  
MOV #14, Port1  
Task1
```

```
PROC MYPROC:  
ADD #12, #12  
MOV #14, #12  
END MYPROC
```



Port

ii. Using dedicated memory space

The second approach:

We want to give the function a number.



We want it to return the twice the number.

We give it the number using the 12th mem locn.

We get the output using the 14th mem locn.



Task 1
Task 2
MOV #12, 2
CALL PYPROC
MOV #14, Port1
Task1

PROC MYPROC:
ADD #12, #12
MOV #14, #12
END MYPROC

ii. Using dedicated memory space

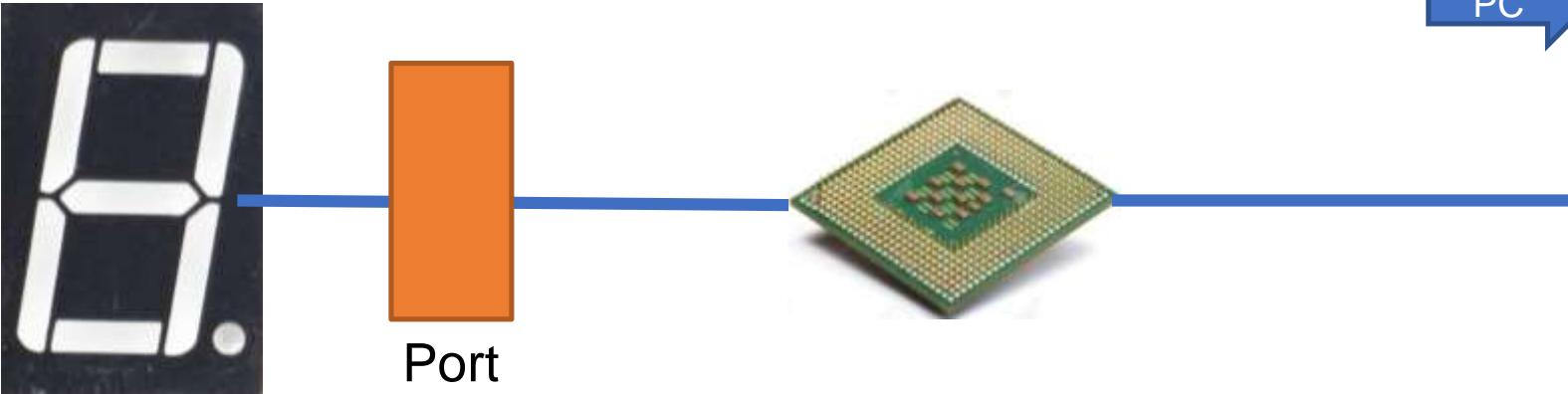
The second approach:

We want to give the function a number.

We want it to return the twice the number.

We give it the number using the 12th mem locn.

We get the output using the 14th mem locn.



```
Task 1  
Task 2  
MOV #12, 2  
CALL PYPROC  
MOV #14, Port1  
Task1
```

```
PROC MYPROC:  
ADD #12, #12  
MOV #14, #12  
END MYPROC
```

ii. Using dedicated memory space

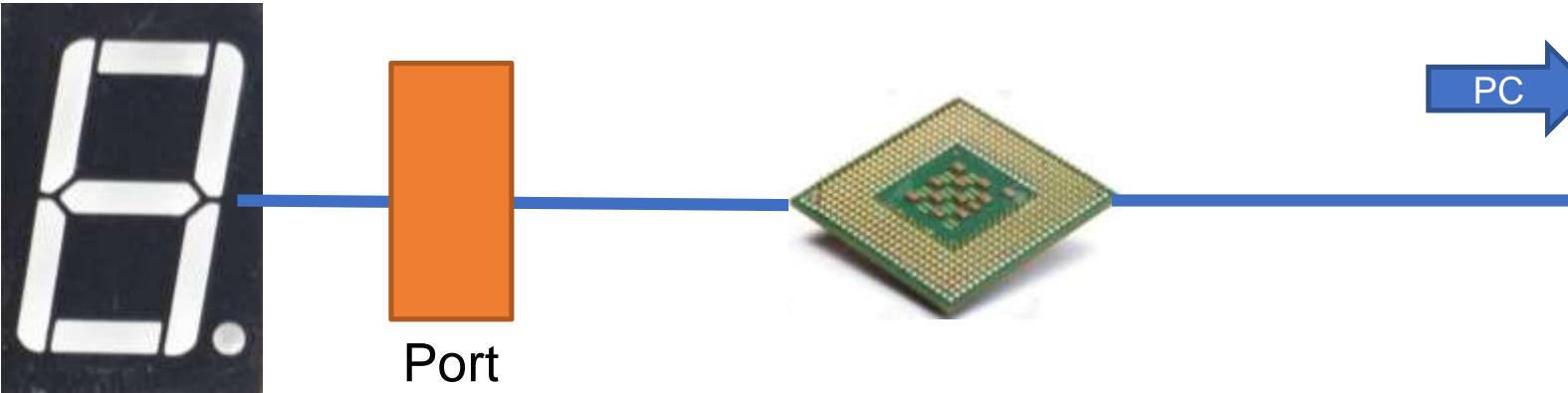
The second approach:

We want to give the function a number.

We want it to return the twice the number.

We give it the number using the 12th mem locn.

We get the output using the 14th mem locn.



```
Task 1  
Task 2  
MOV #12, 2  
CALL PYPROC  
MOV #14, Port1  
Task1
```

```
PROC MYPROC:  
ADD #12, #12  
MOV #14, #12  
END MYPROC
```

ii. Using dedicated memory space

The second approach:

We want to give the function a number.

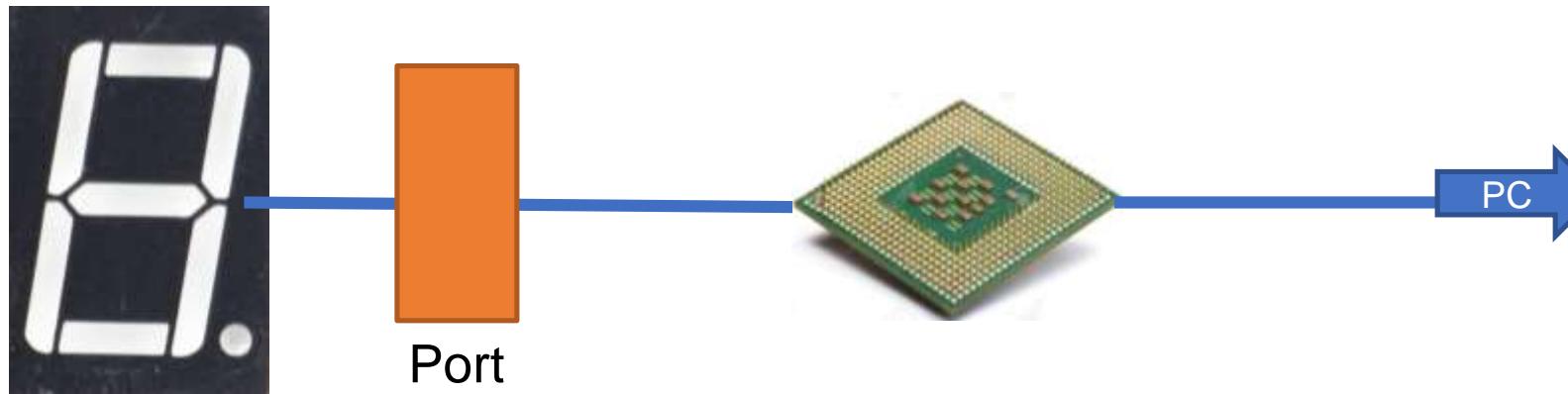
We want it to return the twice the number.

We give it the number using the 12th mem locn.

We get the output using the 14th mem locn.

```
Task 1  
Task 2  
MOV #12, 2  
CALL PYPROC  
MOV #14, Port1  
Task1
```

```
PROC MYPROC:  
ADD #12, #12  
MOV #14, #12  
END MYPROC
```



ii. Using dedicated memory space

The second approach:

We want to give the function a number.

We want it to return the twice the number.

We give it the number using the 12th mem locn.

We get the output using the 14th mem locn.

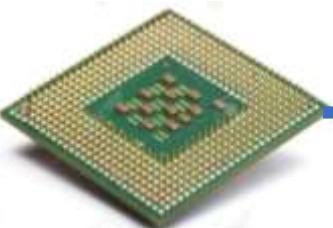


```
Task 1  
Task 2  
MOV #12, 2  
CALL PYPROC  
MOV #14, Port1  
Task1
```

```
PROC MYPROC:  
ADD #12, #12  
MOV #14, #12  
END MYPROC
```



Port



ii. Using dedicated memory space

The second approach:

We want to give the function a number.

We want it to return the twice the number.

We give it the number using the 12th mem locn.

We get the output using the 14th mem locn.

PC →

```
Task 1  
Task 2  
MOV #12, 2  
CALL PYPROC  
MOV #14, Port1  
Task1
```

```
PROC MYPROC:  
ADD #12, #12  
MOV #14, #12  
END MYPROC
```

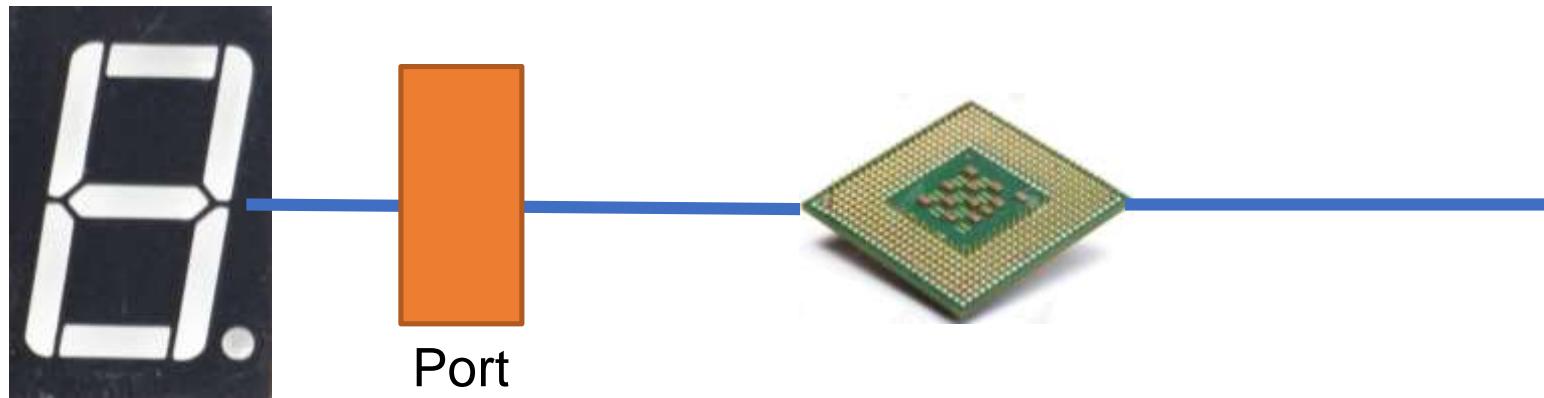


Port

ii. Using dedicated memory space

The second approach:

Note that in this approach also, the **stack** is being used to store the value of the program counter and implement function call return.



```
Task 1  
Task 2  
MOV #12, 2  
CALL PYPROC  
MOV #14, Port1  
Task1
```

```
PROC MYPROC:  
ADD #12, #12  
MOV #14, #12  
END MYPROC
```

iii. Using the stack

In order to understand how stacks work for **passing arguments** (in addition to helping in **returning** back from functions).

Let us consider two functions whose tasks are as follows:

Func1 : Takes two numbers and adds them and then increments it by 1

Func2 : Takes a number and increases it by 1.

Func1 uses Func2, hence we will see an example of nested function here.

00: Task 1
01: Task 2
02: PUSH 2
03: PUSH 4
04: CALL FUNC1
05: POP
06: POP BX
07: MOV BX, Port1
08: Task 3

100: PROC FUNC1:
101: ADD [SP-3],[SP-2]
102: PUSH [SP-3]
103 : CALL FUNC2
104 : MOV [SP-4], [SP-1]
105 : POP
105 : END FUNC1

200: PROC FUNC2:
201: INC [SP-2]
202: END FUNC2

200:
201:
202:
203:
204:
205:

00: Task 1
01: Task 2
02: PUSH 2
03: PUSH 4
04: CALL FUNC1
05: POP
06: POP BX
07: MOV BX, Port1
08: Task 3

200:
201:
202:
203:
204:
205:

100: PROC FUNC1:
101: ADD [SP-3],[SP-2]
102: PUSH [SP-3]
103 : CALL FUNC2
104 : MOV [SP-4], [SP-1]
105 : POP
105 : END FUNC1

200: PROC FUNC2:
201: INC [SP-2]
202: END FUNC2

00: Task 1
01: Task 2
02: PUSH 2
03: PUSH 4
04: CALL FUNC1
05: POP
06: POP BX
07: MOV BX, Port1
08: Task 3

100: PROC FUNC1:
101: ADD [SP-3],[SP-2]
102: PUSH [SP-3]
103 : CALL FUNC2
104 : MOV [SP-4], [SP-1]
105 : POP
105 : END FUNC1

200: PROC FUNC2:
201: INC [SP-2]
202: END FUNC2

200:
201:
202:
203:
204:
205:

00: Task 1
01: Task 2
02: PUSH 2
03: PUSH 4
04: CALL FUNC1
05: POP
06: POP BX
07: MOV BX, Port1
08: Task 3

200: 2
201:
202:
203:
204:
205:

100: PROC FUNC1:
101: ADD [SP-3],[SP-2]
102: PUSH [SP-3]
103 : CALL FUNC2
104 : MOV [SP-4], [SP-1]
105 : POP
105 : END FUNC1

200: PROC FUNC2:
201: INC [SP-2]
202: END FUNC2

00: Task 1
01: Task 2
02: PUSH 2
03: PUSH 4
04: CALL FUNC1
05: POP
06: POP BX
07: MOV BX, Port1
08: Task 3

100: PROC FUNC1:
101: ADD [SP-3],[SP-2]
102: PUSH [SP-3]
103 : CALL FUNC2
104 : MOV [SP-4], [SP-1]
105 : POP
105 : END FUNC1

200: PROC FUNC2:
201: INC [SP-2]
202: END FUNC2

200: 2
201: 4
202:
203:
204:
205:

```
00: Task 1  
01: Task 2  
02: PUSH 2  
03: PUSH 4  
04: CALL FUNC1  
05: POP  
06: POP BX  
07: MOV BX, Port1  
08: Task 3
```



```
100: PROC FUNC1:  
101: ADD [SP-3],[SP-2]  
102: PUSH [SP-3]  
103 : CALL FUNC2  
104 : MOV [SP-4], [SP-1]  
105 : POP  
105 : END FUNC1
```

```
200: 2  
201: 4  
202: 5  
203:  
204:  
205:
```



```
200: PROC FUNC2:  
201: INC [SP-2]  
202: END FUNC2
```

```
00: Task 1  
01: Task 2  
02: PUSH 2  
03: PUSH 4  
04: CALL FUNC1  
05: POP  
06: POP BX  
07: MOV BX, Port1  
08: Task 3
```

→

```
100: PROC FUNC1:  
101: ADD [SP-3],[SP-2]  
102: PUSH [SP-3]  
103 : CALL FUNC2  
104 : MOV [SP-4], [SP-1]  
105 : POP  
105 : END FUNC1
```

```
200: PROC FUNC2:  
201: INC [SP-2]  
202: END FUNC2
```

```
200: 6  
201: 4  
202: 5  
203:  
204:  
205:
```

```
00: Task 1  
01: Task 2  
02: PUSH 2  
03: PUSH 4  
04: CALL FUNC1  
05: POP  
06: POP BX  
07: MOV BX, Port1  
08: Task 3
```

```
100: PROC FUNC1:  
101: ADD [SP-3],[SP-2]  
102: PUSH [SP-3]  
103 : CALL FUNC2  
104 : MOV [SP-4], [SP-1]  
105 : POP  
105 : END FUNC1
```

```
200: PROC FUNC2:  
201: INC [SP-2]  
202: END FUNC2
```

```
200: 6  
201: 4  
202: 5  
203: 6  
204:  
205:
```

```
00: Task 1  
01: Task 2  
02: PUSH 2  
03: PUSH 4  
04: CALL FUNC1  
05: POP  
06: POP BX  
07: MOV BX, Port1  
08: Task 3
```

```
100: PROC FUNC1:  
101: ADD [SP-3],[SP-2]  
102: PUSH [SP-3]  
103 : CALL FUNC2  
104 : MOV [SP-4], [SP-1]  
105 : POP  
105 : END FUNC1
```

→
200: PROC FUNC2:
201: INC [SP-2]
202: END FUNC2

```
200: 6  
201: 4  
202: 5  
203: 6  
204: 104  
205:
```

```
00: Task 1  
01: Task 2  
02: PUSH 2  
03: PUSH 4  
04: CALL FUNC1  
05: POP  
06: POP BX  
07: MOV BX, Port1  
08: Task 3
```

```
100: PROC FUNC1:  
101: ADD [SP-3],[SP-2]  
102: PUSH [SP-3]  
103 : CALL FUNC2  
104 : MOV [SP-4], [SP-1]  
105 : POP  
105 : END FUNC1
```

```
200: 6  
201: 4  
202: 5  
203: 7  
204: 104  
205:
```



```
00: Task 1  
01: Task 2  
02: PUSH 2  
03: PUSH 4  
04: CALL FUNC1  
05: POP  
06: POP BX  
07: MOV BX, Port1  
08: Task 3
```

```
100: PROC FUNC1:  
101: ADD [SP-3],[SP-2]  
102: PUSH [SP-3]  
103 : CALL FUNC2  
104 : MOV [SP-4], [SP-1]  
105 : POP  
105 : END FUNC1
```

```
200: 6  
201: 4  
202: 5  
203: 7  
204:  
205:
```



```
00: Task 1  
01: Task 2  
02: PUSH 2  
03: PUSH 4  
04: CALL FUNC1  
05: POP  
06: POP BX  
07: MOV BX, Port1  
08: Task 3
```

```
100: PROC FUNC1:  
101: ADD [SP-3],[SP-2]  
102: PUSH [SP-3]  
103 : CALL FUNC2  
104 : MOV [SP-4], [SP-1]  
105 : POP  
105 : END FUNC1
```



```
200: 7  
201: 4  
202: 5  
203: 7  
204:  
205:
```

00: Task 1
01: Task 2
02: PUSH 2
03: PUSH 4
04: CALL FUNC1
05: POP
06: POP BX
07: MOV BX, Port1
08: Task 3

100: PROC FUNC1:
101: ADD [SP-3],[SP-2]
102: PUSH [SP-3]
103 : CALL FUNC2
104 : MOV [SP-4], [SP-1]
105 : POP
105 : END FUNC1

200: PROC FUNC2:
201: INC [SP-2]
202: END FUNC2

200: 7
201: 4
202: 5
203:
204:
205:

00: Task 1
01: Task 2
02: PUSH 2
03: PUSH 4
04: CALL FUNC1
05: POP
06: POP BX
07: MOV BX, Port1
08: Task 3

200: 7
201: 4
202:
203:
204:
205:

100: PROC FUNC1:
101: ADD [SP-3],[SP-2]
102: PUSH [SP-3]
103 : CALL FUNC2
104 : MOV [SP-4], [SP-1]
105 : POP
105 : END FUNC1

200: PROC FUNC2:
201: INC [SP-2]
202: END FUNC2

00: Task 1
01: Task 2
02: PUSH 2
03: PUSH 4
04: CALL FUNC1
05: POP
06: POP BX
07: MOV BX, Port1
08: Task 3



100: PROC FUNC1:
101: ADD [SP-3],[SP-2]
102: PUSH [SP-3]
103 : CALL FUNC2
104 : MOV [SP-4], [SP-1]
105 : POP
105 : END FUNC1

200: PROC FUNC2:
201: INC [SP-2]
202: END FUNC2



200: 7
201:
202:
203:
204:
205:

```
00: Task 1  
01: Task 2  
02: PUSH 2  
03: PUSH 4  
04: CALL FUNC1  
05: POP  
06: POP BX  
07: MOV BX, Port1  
08: Task 3
```



```
200:  
201:  
202:  
203:  
204:  
205:
```

```
100: PROC FUNC1:  
101: ADD [SP-3],[SP-2]  
102: PUSH [SP-3]  
103 : CALL FUNC2  
104 : MOV [SP-4], [SP-1]  
105 : POP  
105 : END FUNC1
```

```
200: PROC FUNC2:  
201: INC [SP-2]  
202: END FUNC2
```

00: Task 1
01: Task 2
02: PUSH 2
03: PUSH 4
04: CALL FUNC1
05: POP
06: POP BX
07: MOV BX, Port1
08: Task 3

200:
201:
202:
203:
204:
205:

100: PROC FUNC1:
101: ADD [SP-3],[SP-2]
102: PUSH [SP-3]
103 : CALL FUNC2
104 : MOV [SP-4], [SP-1]
105 : POP
105 : END FUNC1

200: PROC FUNC2:
201: INC [SP-2]
202: END FUNC2



00: Task 1
01: Task 2
02: PUSH 2
03: PUSH 4
04: CALL FUNC1
05: POP
06: POP BX
07: MOV BX, Port1
08: Task 3

200:
201:
202:
203:
204:
205:

100: PROC FUNC1:
101: ADD [SP-3],[SP-2]
102: PUSH [SP-3]
103 : CALL FUNC2
104 : MOV [SP-4], [SP-1]
105 : POP
105 : END FUNC1

200: PROC FUNC2:
201: INC [SP-2]
202: END FUNC2



Thank You

Chapter 13

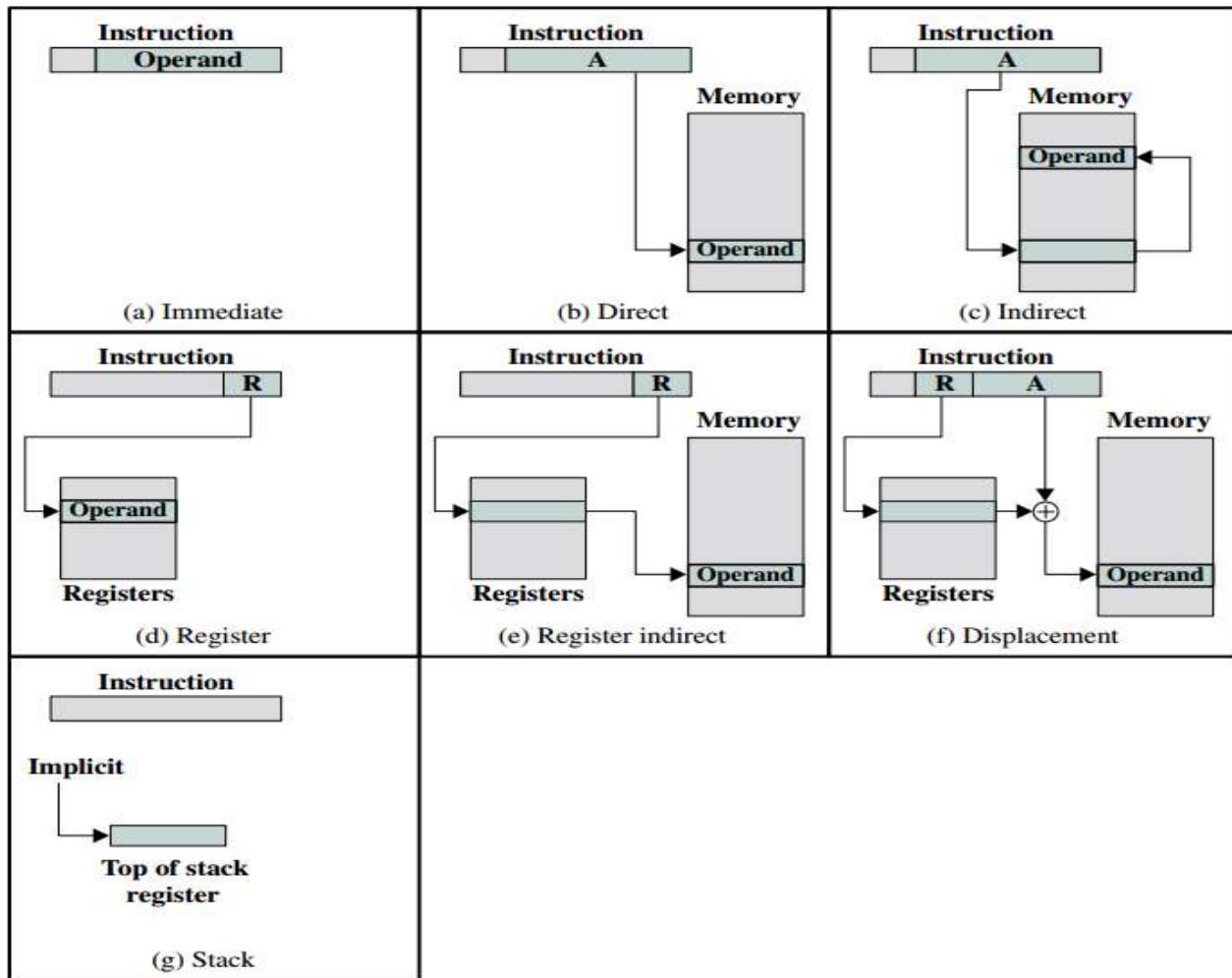


Figure 13.1 Addressing Modes

Implicit

- We don't mention anything
- Processor knows, from where to get data
- E.g : CMA
- No operand is mentioned here, but
- Implicitly AX is being complemented

CMA

Instruction Register

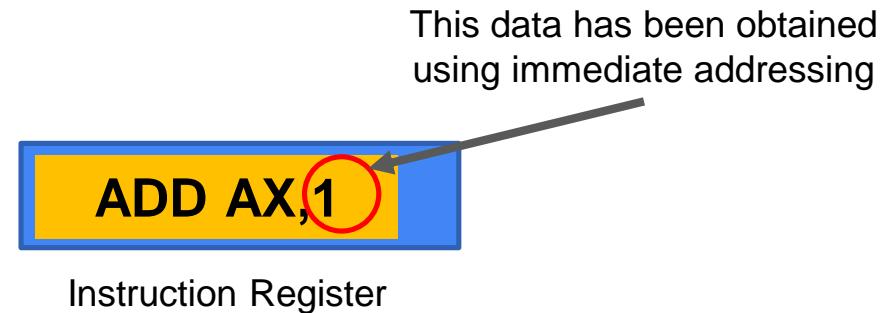
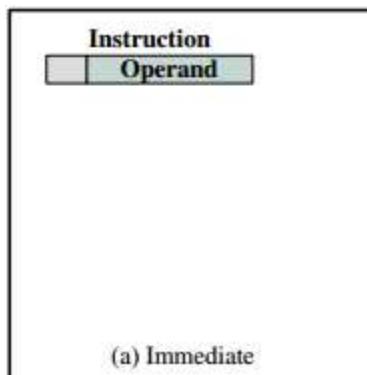
Immediate

- Instruction has opcode and operands.
- The operands are present inside the instruction,
- Hence no need to fetch it from the memory or register.



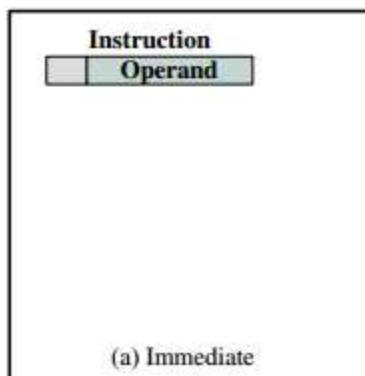
Immediate

- Instruction has opcode and operands.
- The operands are present inside the instruction,
- Hence no need to fetch it from the memory or register.



Immediate

- Instruction has opcode and operands.
- The operands are present inside the instruction,
- Hence no need to fetch it from the memory or register.



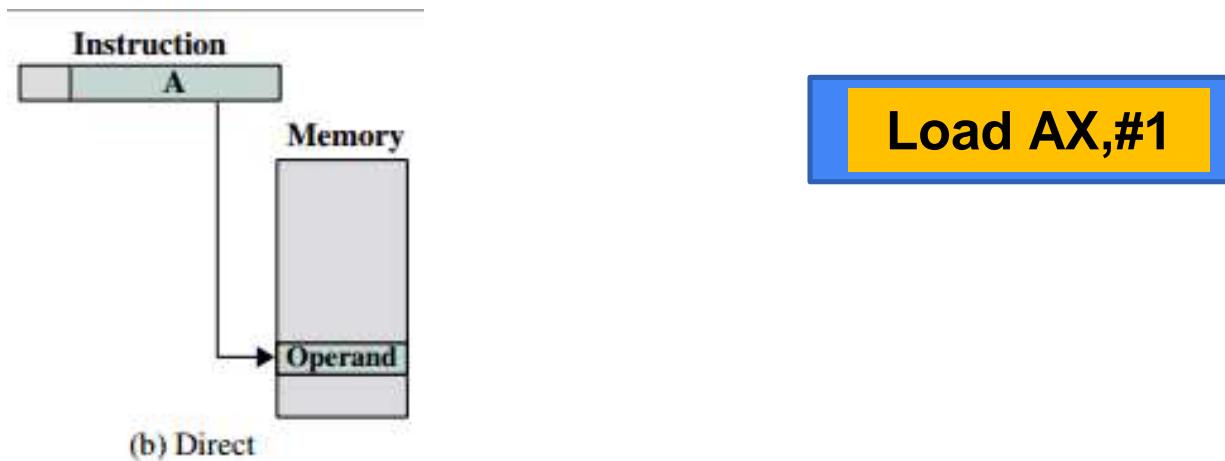
Instruction Register



However, the data from the AX register has been obtained using Register (Direct) addressing

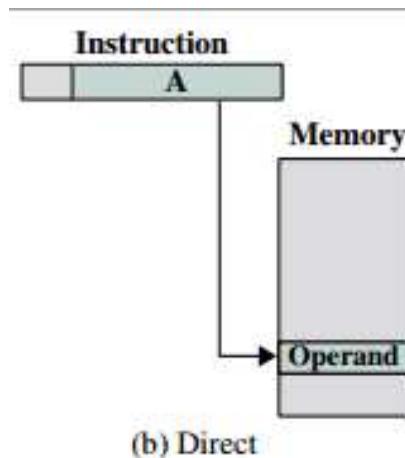
Direct Addressing Mode:

- Instruction has opcode and
- The address of the memory location, where the operand is
- Here it is at the Ath location.



Direct Addressing Mode:

- Instruction has opcode and
- The address of the memory location, where the operand is
- Here it is at the Ath location.



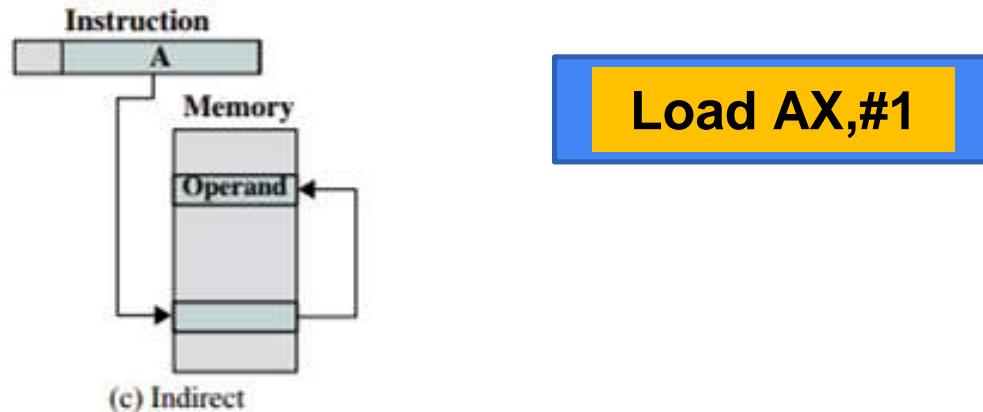
Load AX, #1

Data is needed to be brought in from
the 1st address in the memory

Indirect Addressing Mode

Similar to pointer to pointer.

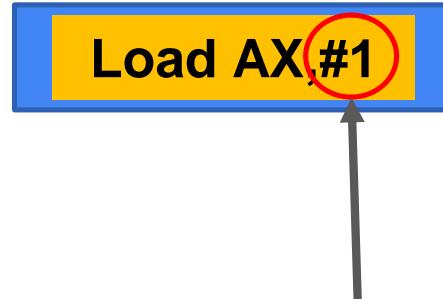
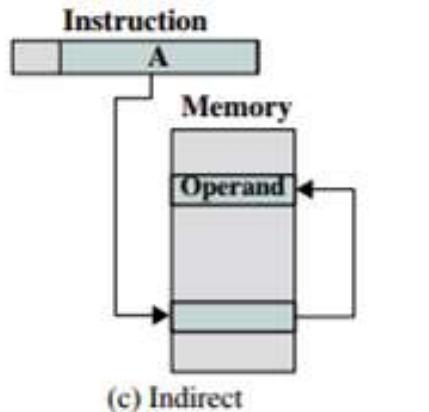
Address of a location which has address of the operand



Indirect Addressing Mode

Similar to pointer to pointer.

Address of a location which has address of the operand

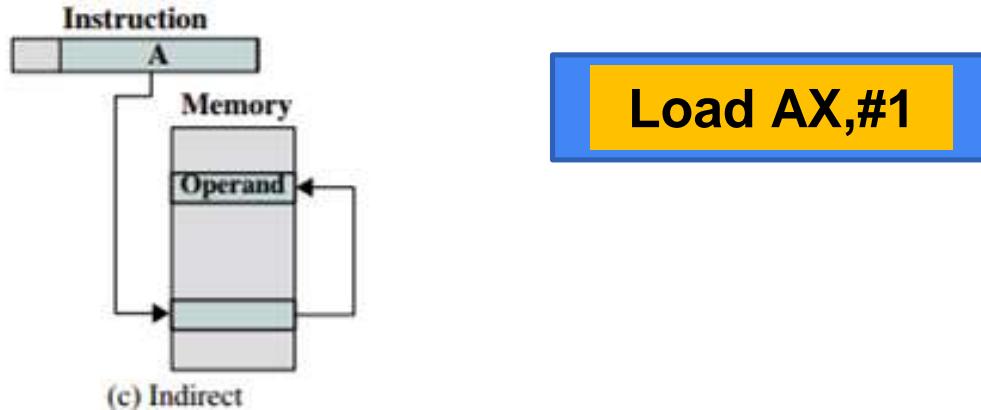


Data is needed to bring in via
the 1st address in the memory

Indirect Addressing Mode

Similar to pointer to pointer.

Address of a location which has address of the operand

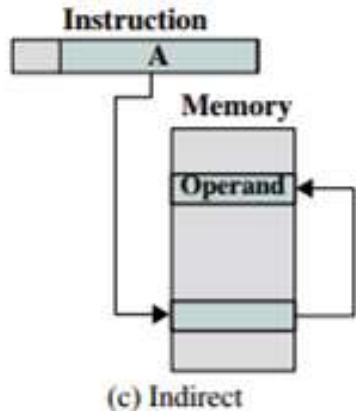


| | |
|---|-----|
| 0 | |
| 1 | 6 |
| 2 | |
| 3 | |
| 4 | |
| 5 | |
| 6 | 100 |
| 7 | |

Indirect Addressing Mode

Similar to pointer to pointer.

Address of a location which has address of the operand



Load AX,#1

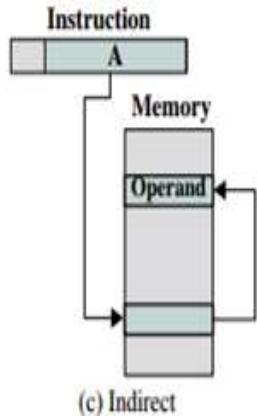
So the register AX will be loaded with
the number.....

100

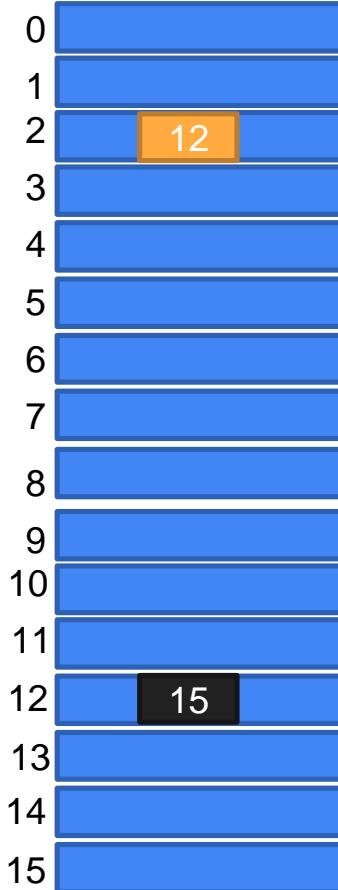
| | |
|---|-----|
| 0 | |
| 1 | 6 |
| 2 | |
| 3 | |
| 4 | |
| 5 | |
| 6 | 100 |
| 7 | |

Indirect Addressing Mode

Why do we need this?



The Instruction register is 8 bits

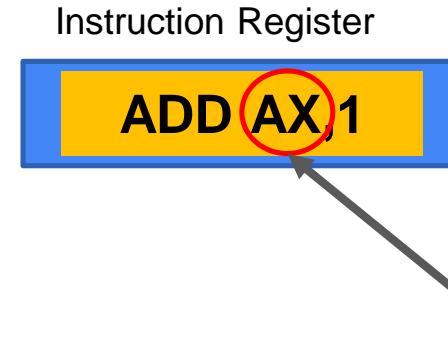
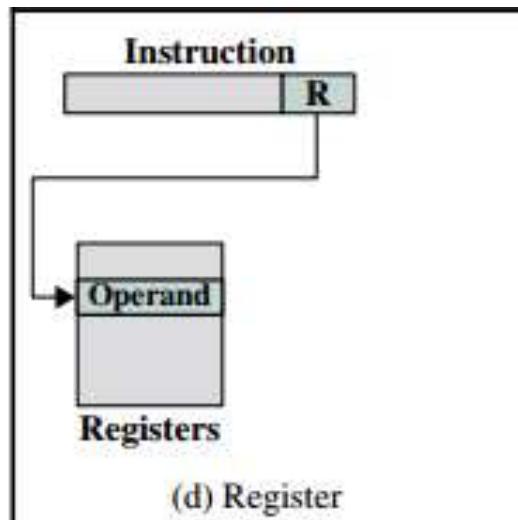


Register Addressing

Each processor has a set of registers

Based on register no, the register is selected

Each register has a unique identifier

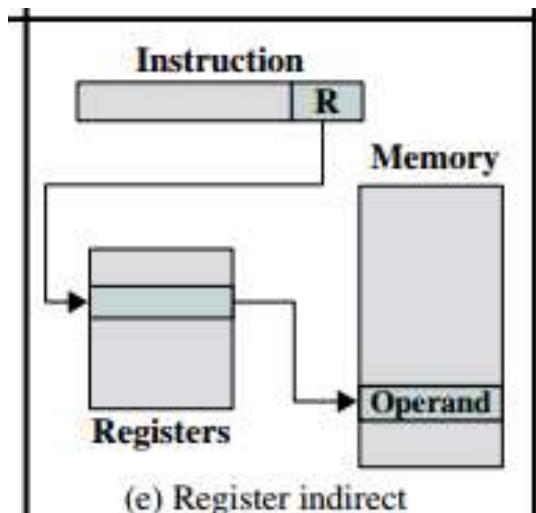


The data from the AX register has been obtained via Register (Direct) addressing

Register Indirect Addressing

Pointer to a pointer concept

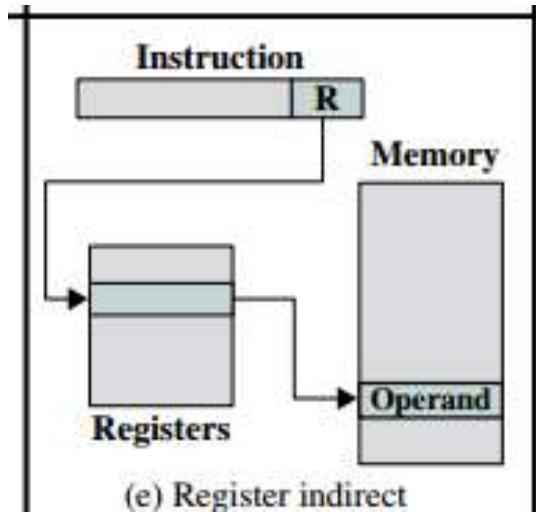
Register pointing to a register address of the operand



Register Indirect Addressing

Pointer to a pointer concept

Register pointing to a register
address of the operand

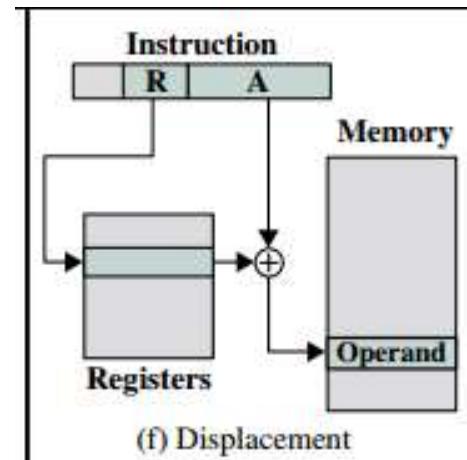


The Instruction register is 8 bits



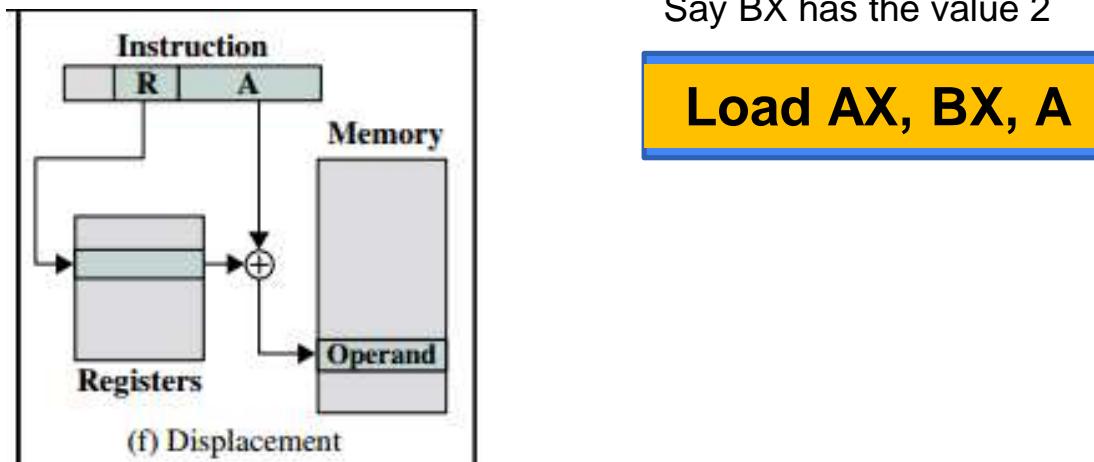
Displacement Addressing

- Instruction will contain the register and address.
- Register content and the address content are combined
- The combined result is the actual physical address



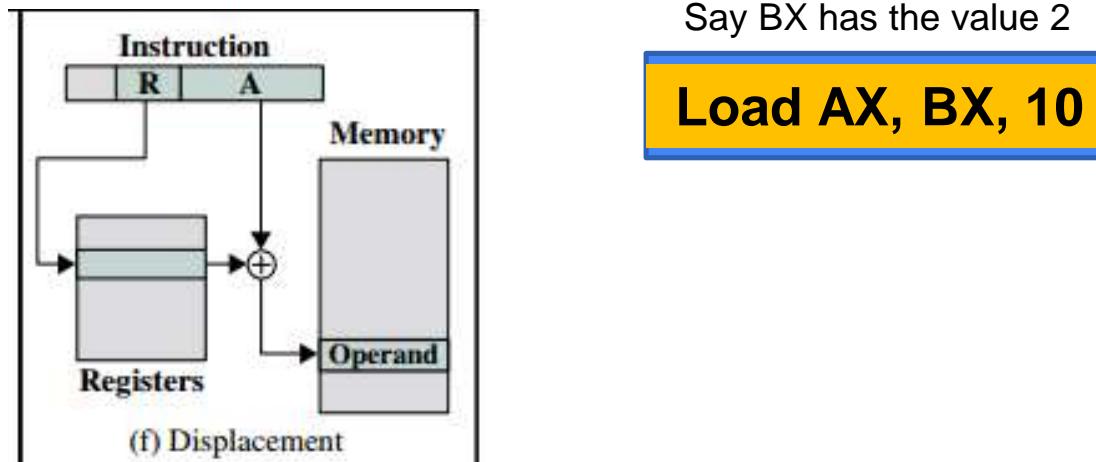
Displacement Addressing

- Instruction will contain the register and address.
- Register content and the address content are combined
- The combined result is the actual physical address



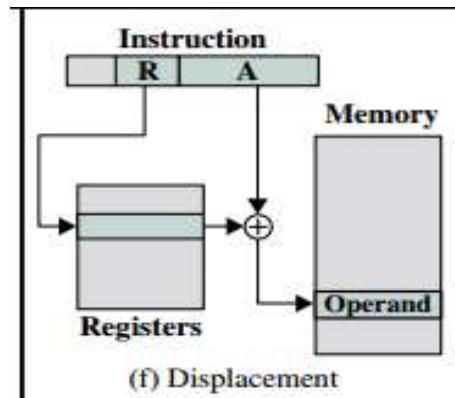
Displacement Addressing

- Instruction will contain the register and address.
- Register content and the address content are combined
- The combined result is the actual physical address



Displacement Addressing

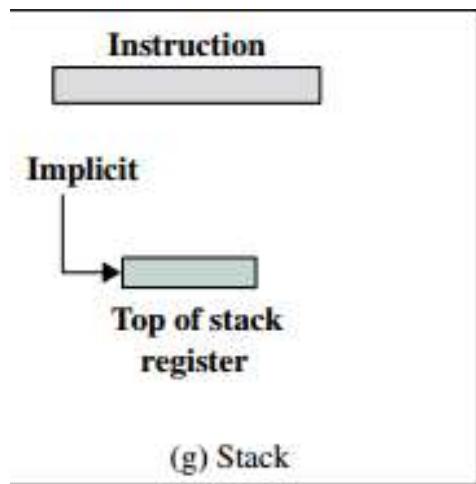
- **Relative Displacement Addressing:** Register is a program counter.
- **Base Register Displacement Addressing :** base address of a segment will be in register, and A is the offset. $2000+20 \rightarrow 2020$.
- **Indexing Displacement Addressing :** A is the address of the array. R has the index value. Adding will give us the physical address.



Stack addressing (0 addressing)

ADD // means add the top two elements of a stack

SUB // means subtract the top two elements of a stack



You will be fully clarified about this after starting
Chapter 12.

Comparison between all the addressing modes

Table 13.1 Basic Addressing Modes

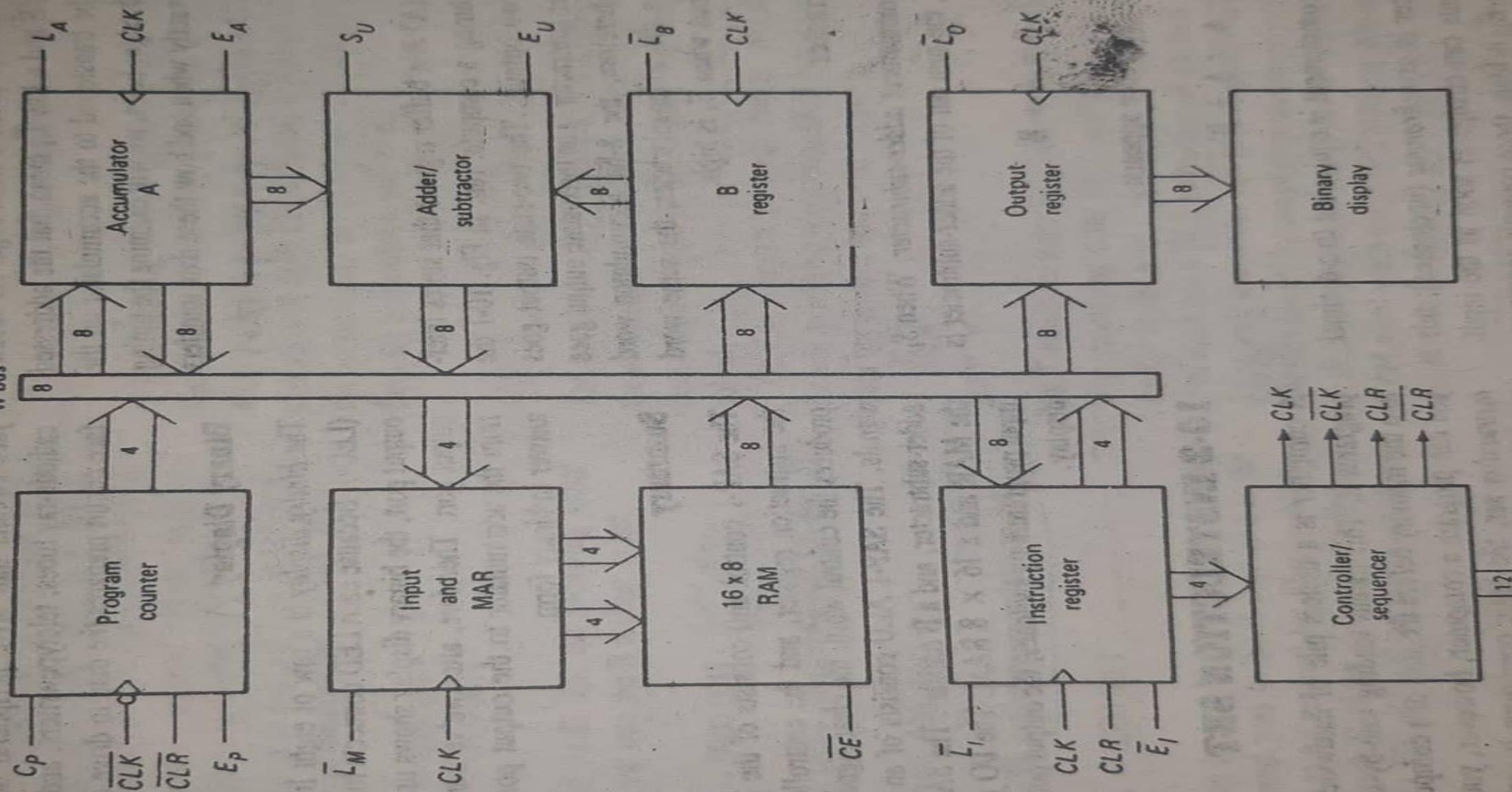
| Mode | Algorithm | Principal Advantage | Principal Disadvantage |
|-------------------|-------------------|---------------------|----------------------------|
| Immediate | Operand = A | No memory reference | Limited operand magnitude |
| Direct | EA = A | Simple | Limited address space |
| Indirect | EA = (A) | Large address space | Multiple memory references |
| Register | EA = R | No memory reference | Limited address space |
| Register indirect | EA = (R) | Large address space | Extra memory reference |
| Displacement | EA = A + (R) | Flexibility | Complexity |
| Stack | EA = top of stack | No memory reference | Limited applicability |

Thank You

CHAPTER 14

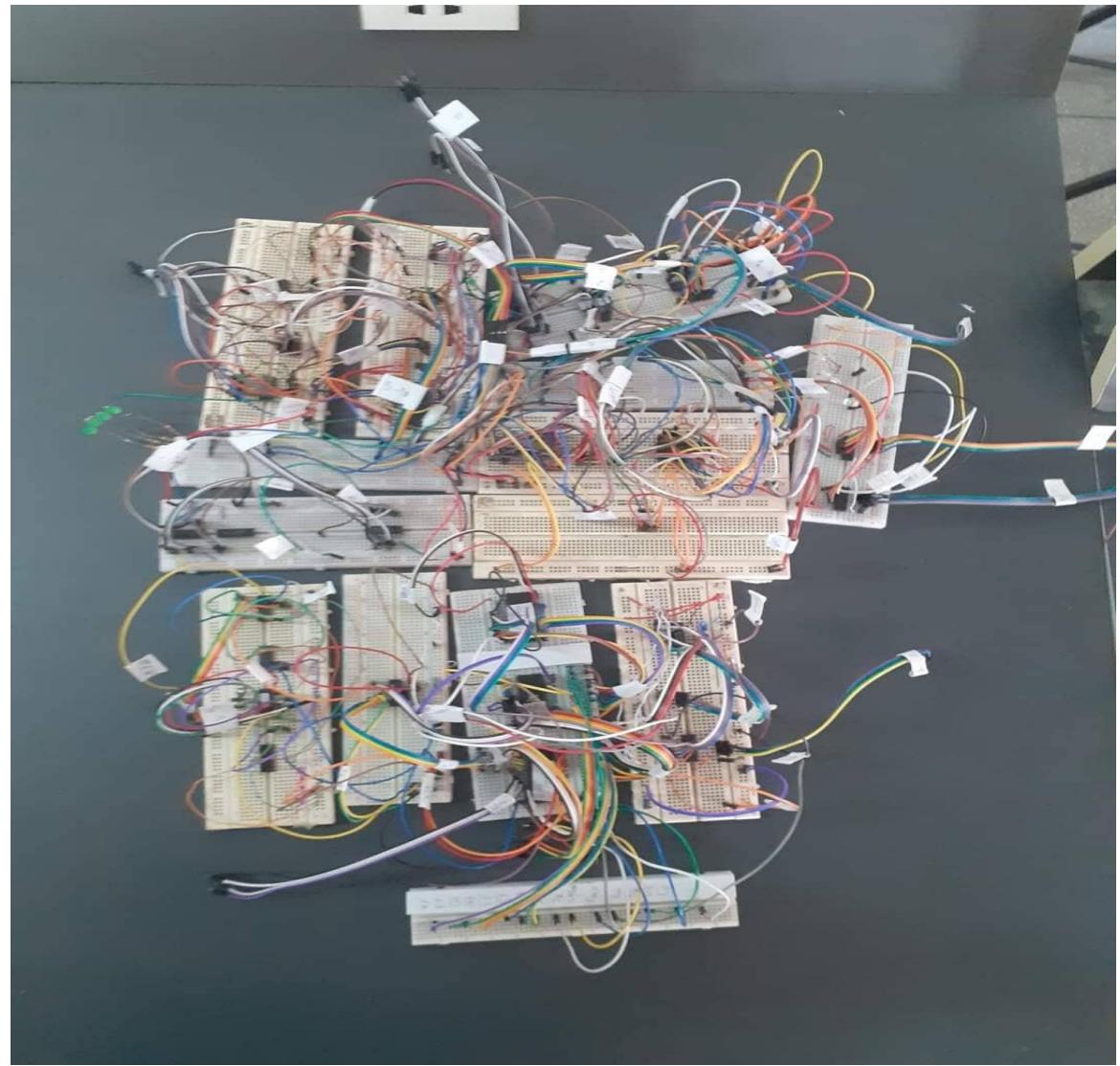
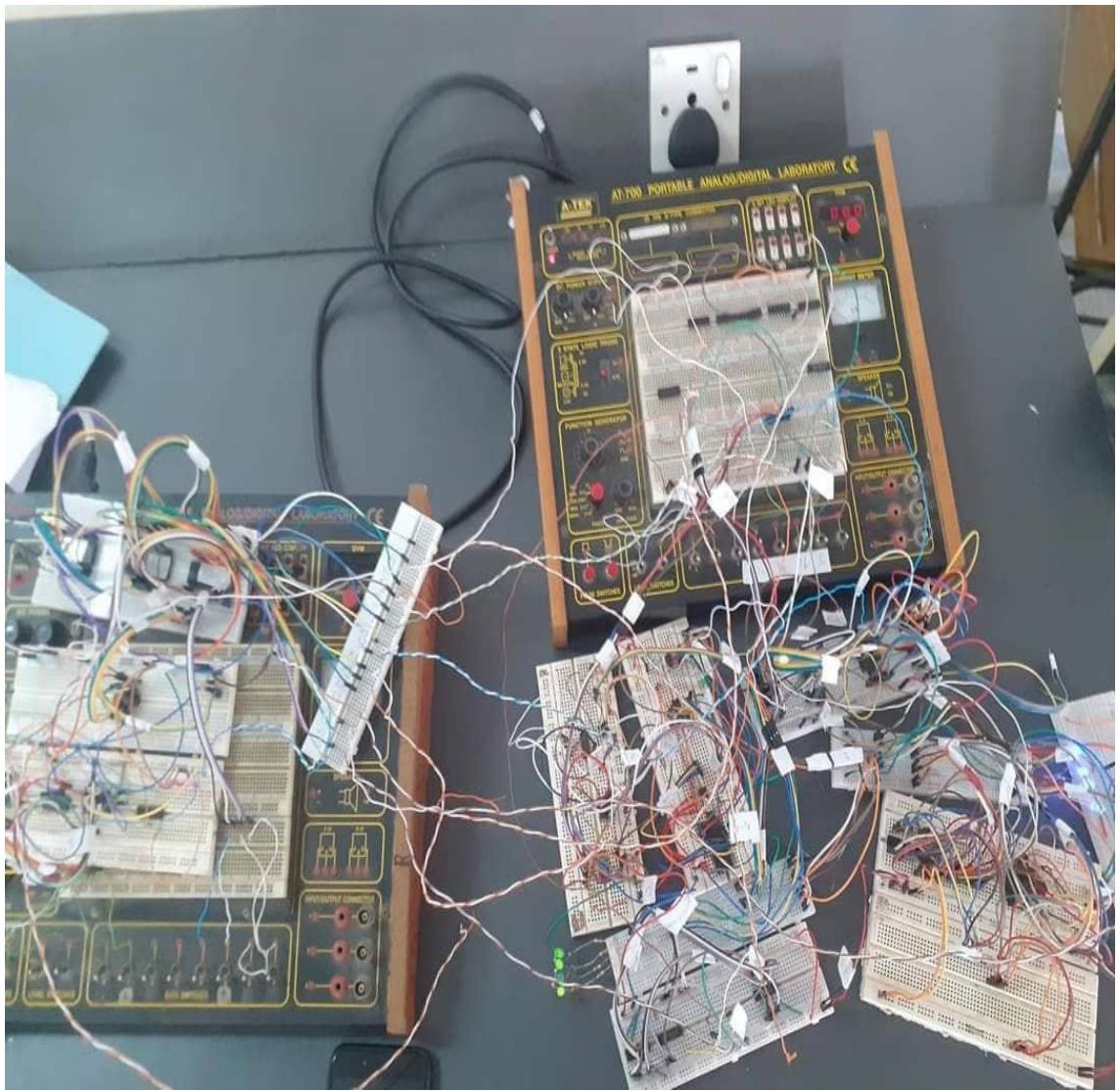
Processor Structure and Function

W bus



•1 SAP-1 architecture.

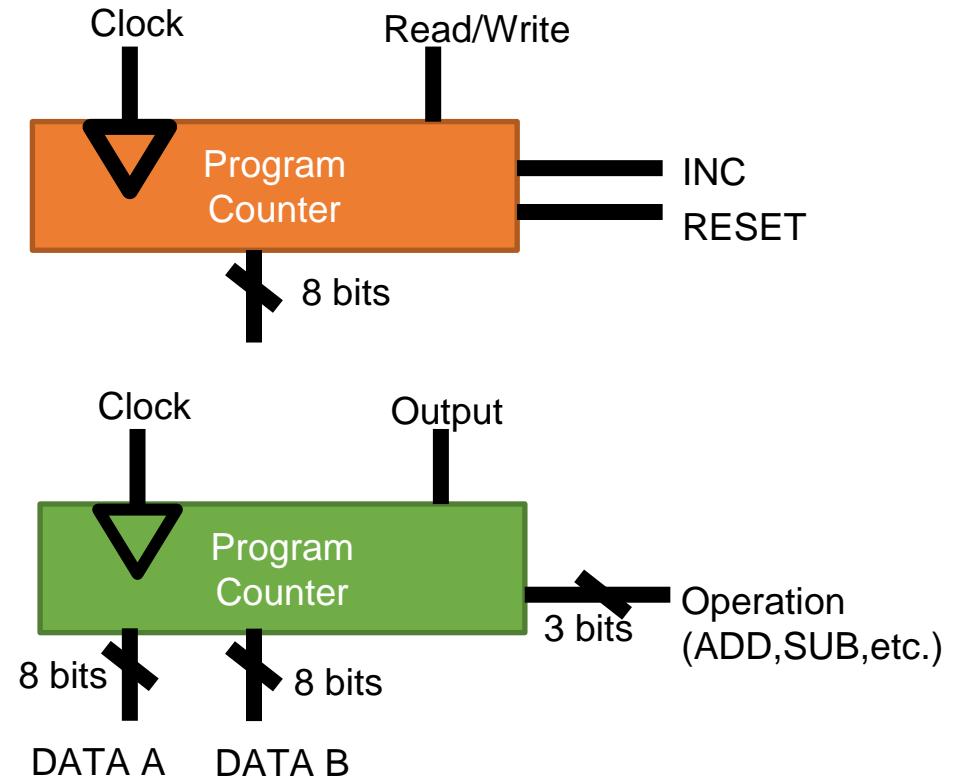
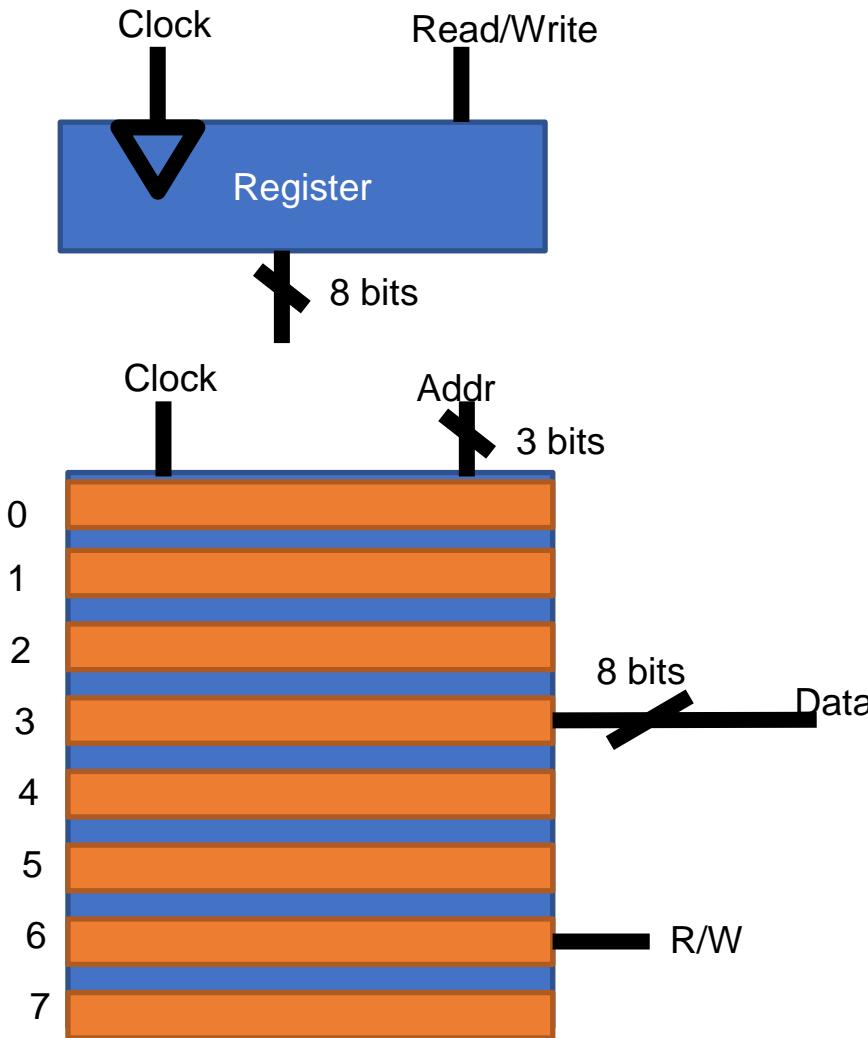
$$C_p E_p \overline{L_M} \overline{CE} \overline{L_I} \overline{E_A} S_U E_U \overline{L_B} \overline{L_0}$$

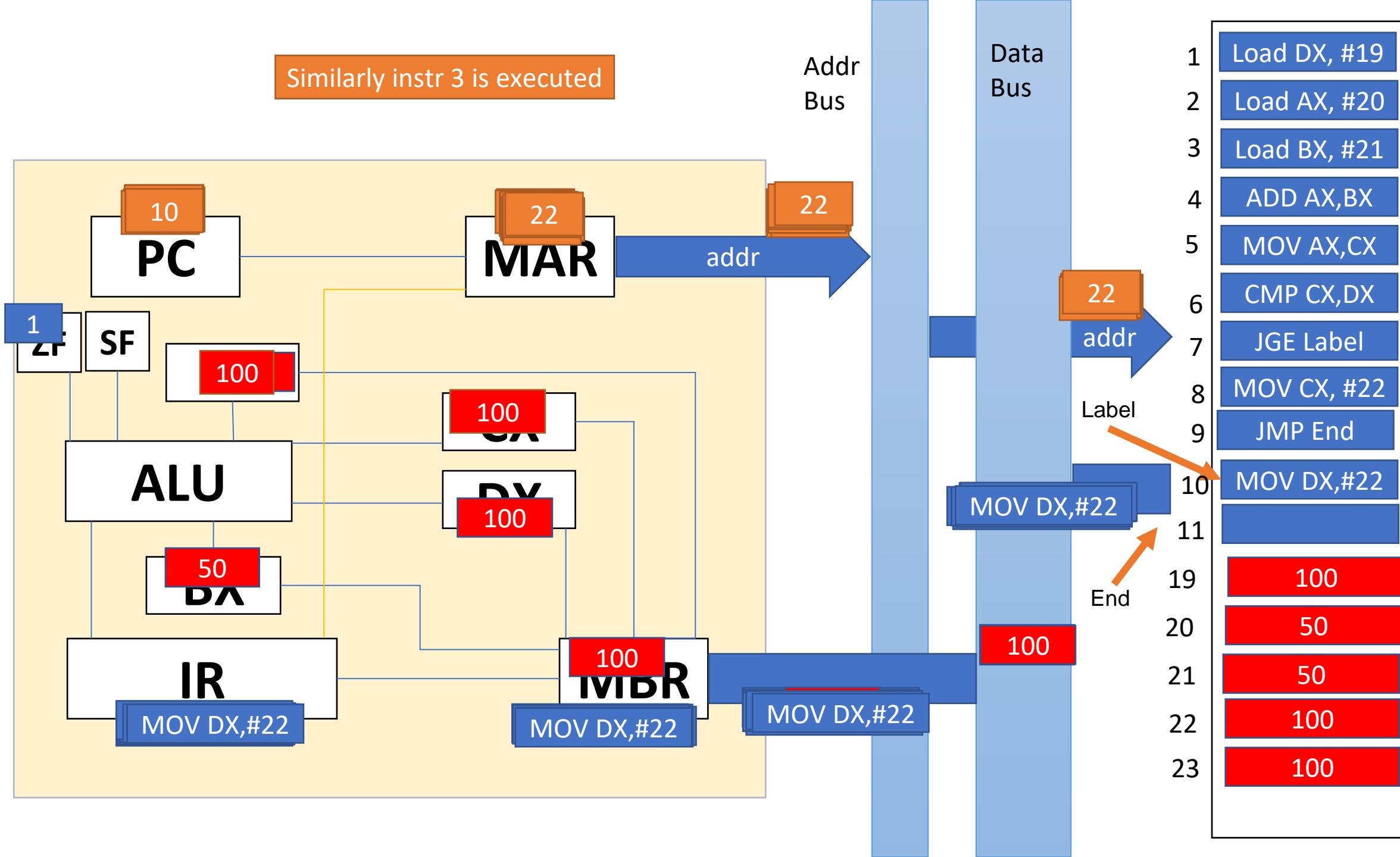


1. Lets see how a processor works

1. Basic Components
2. Processor Simulation
3. 5 Main functions

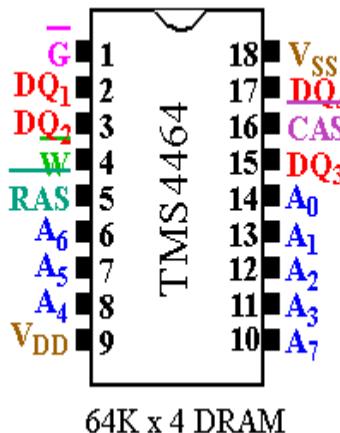
The components of a processor



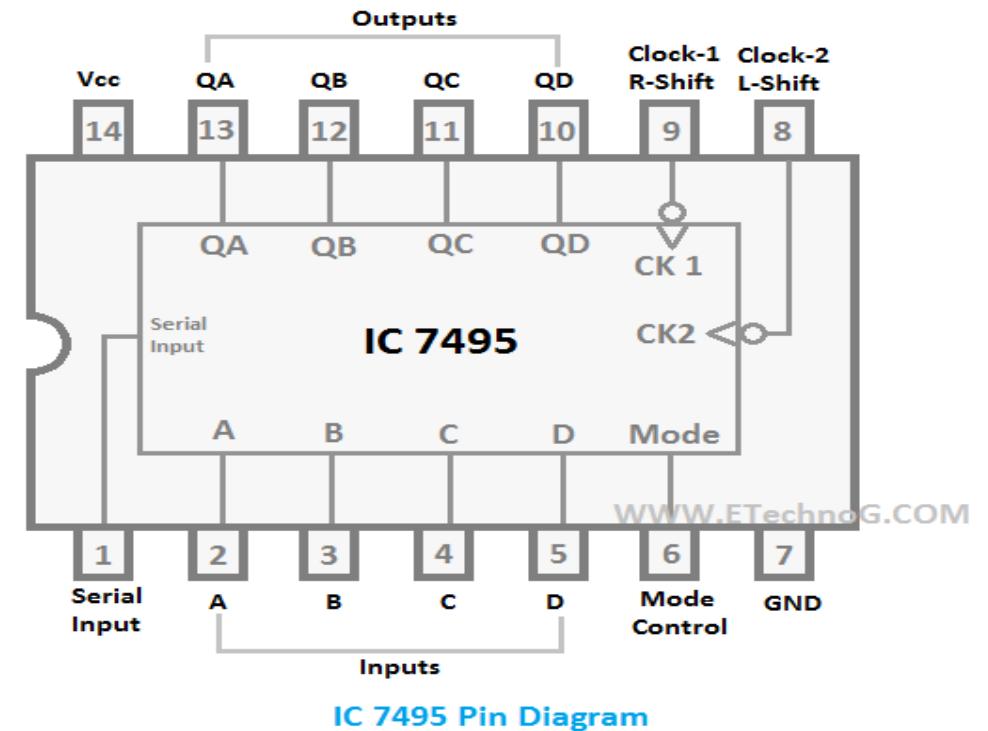


A more detailed view

The components of a processor again



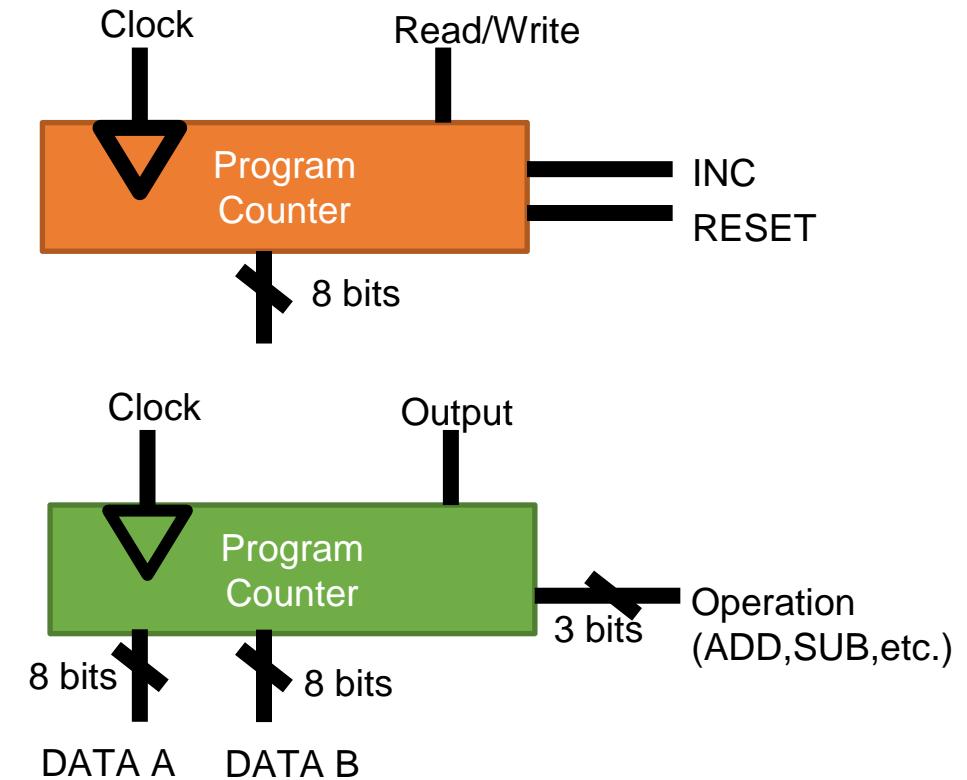
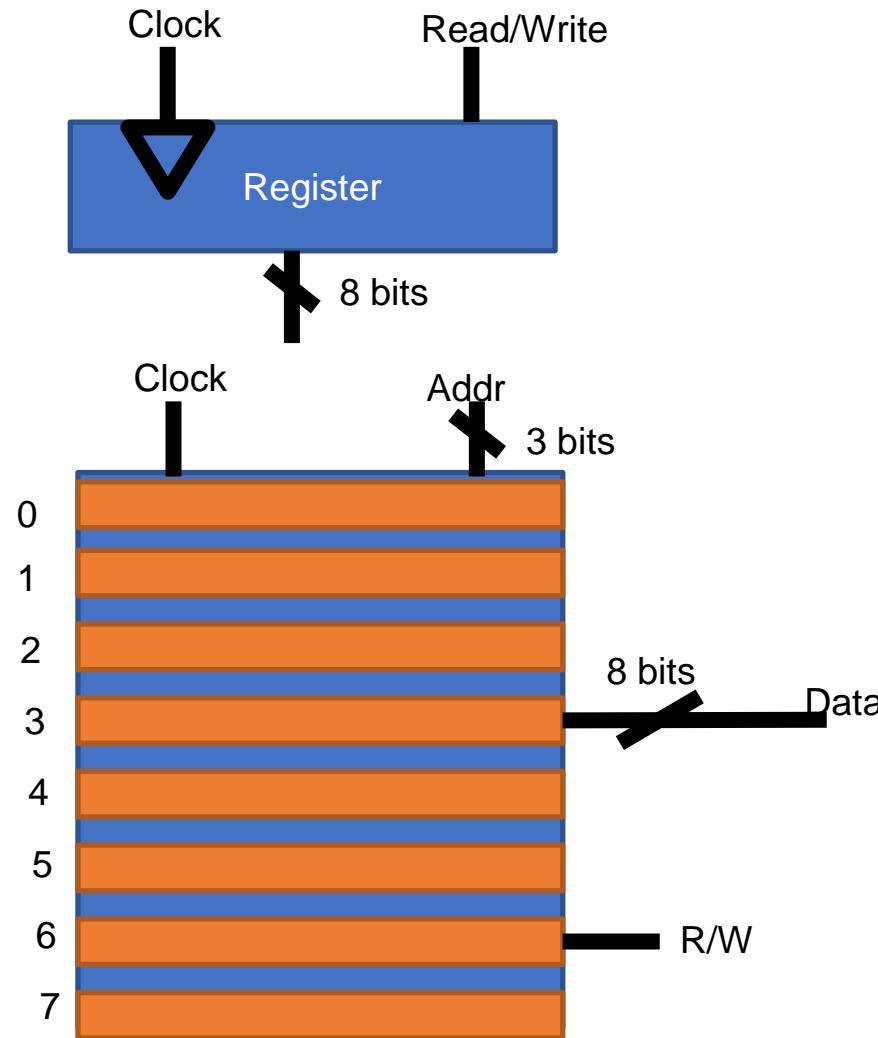
| Pin(s) | Function |
|----------------------------------|-----------------------|
| A ₀ -A ₇ | Address |
| DQ ₀ -DQ ₄ | Data In/Data Out |
| RAS | Row Address Strobe |
| CAS | Column Address Strobe |
| G | Output Enable |
| W | Write Enable |



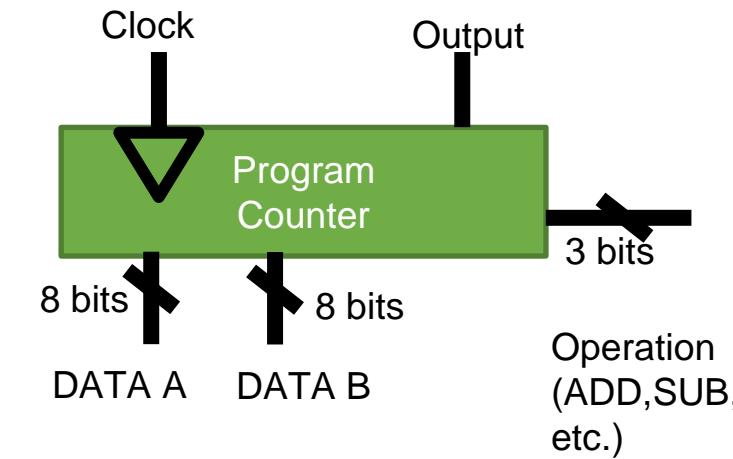
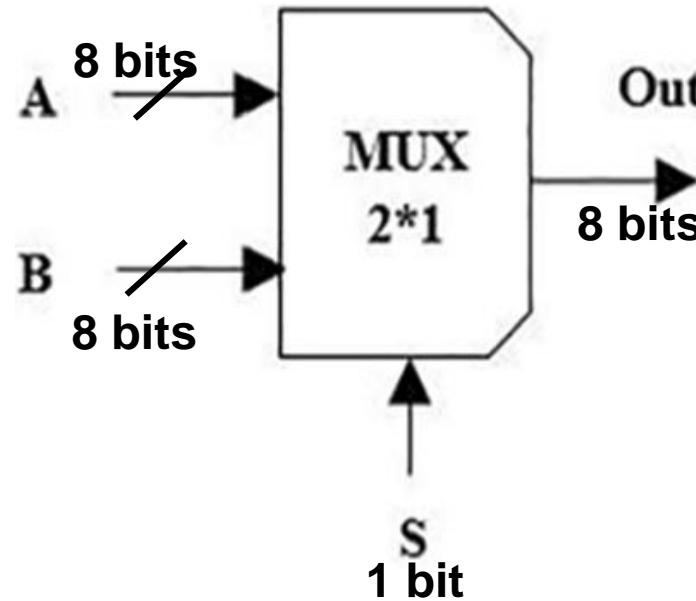
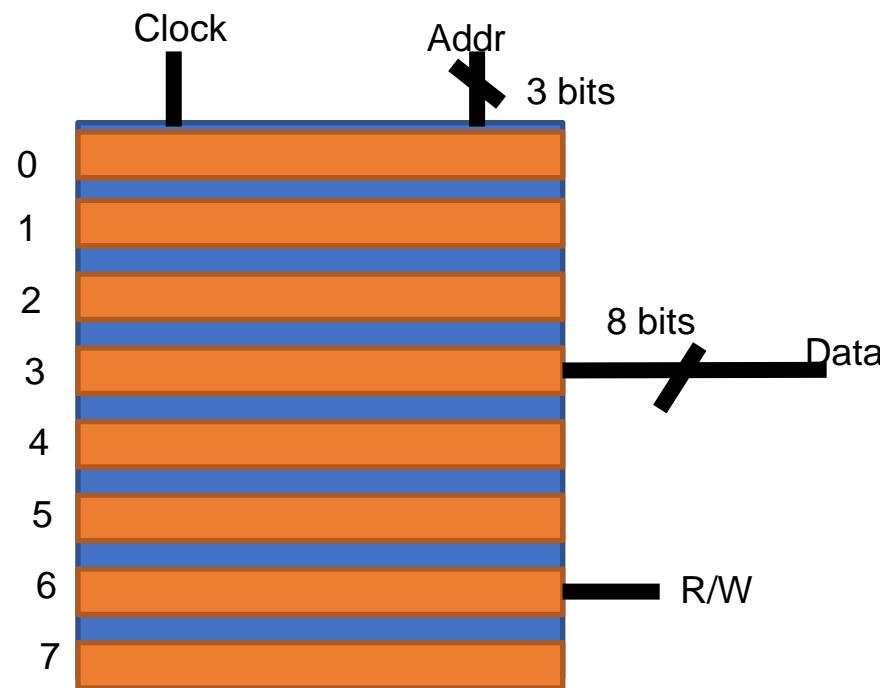
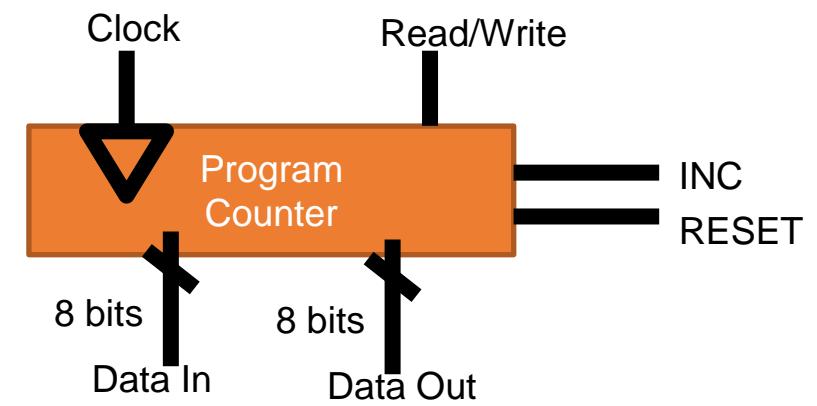
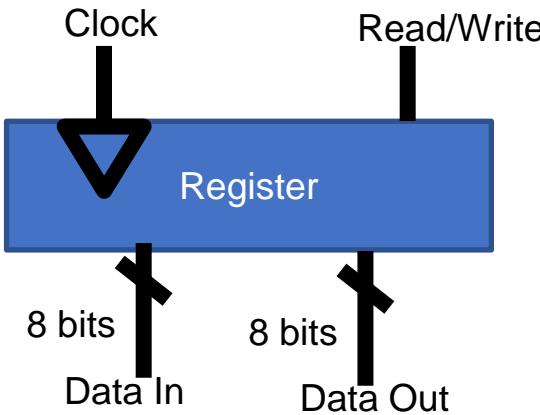
Memory with one path for both data input and output

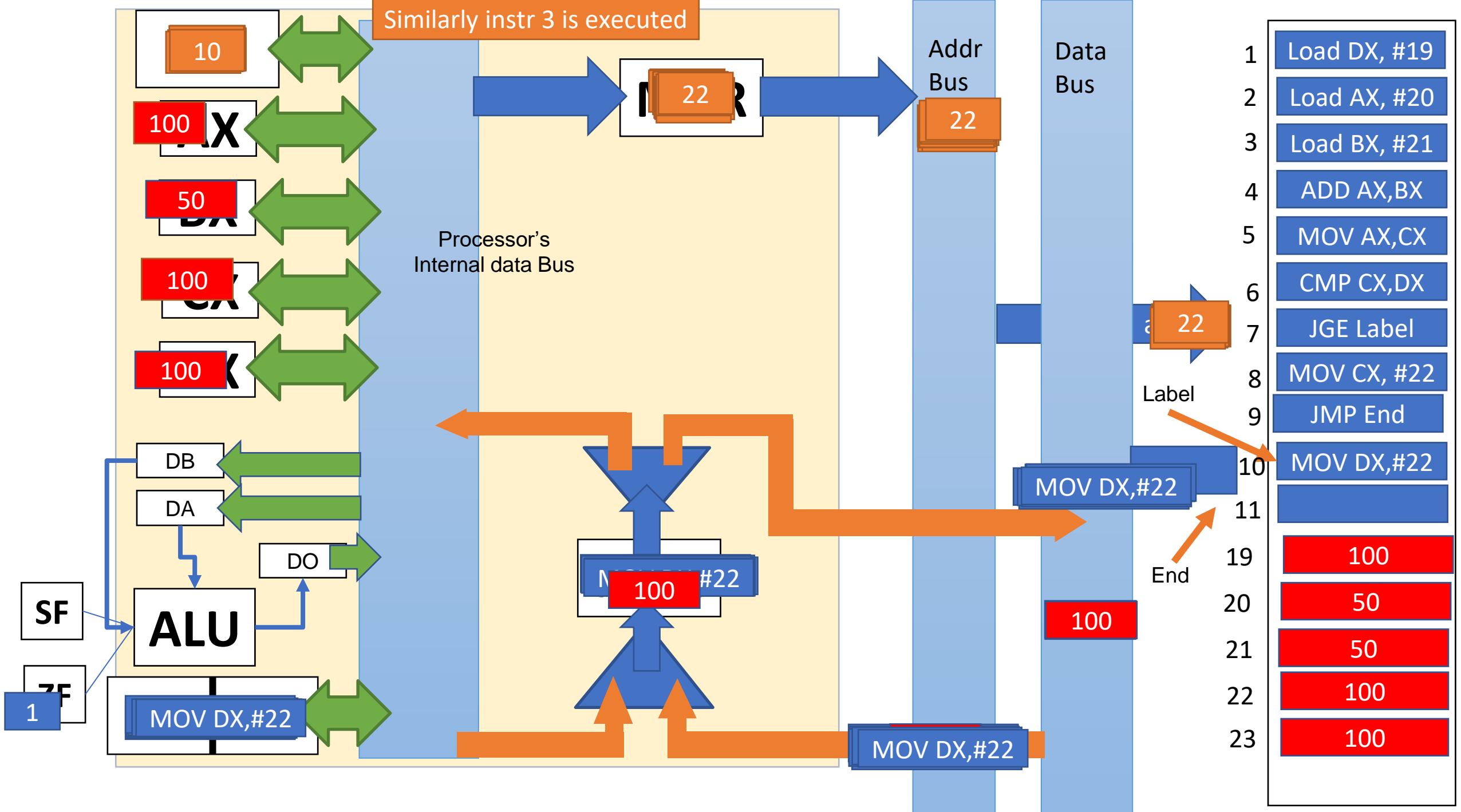
Memory for different paths for input and output

The components of a processor again



The components of a processor again





5 Main functions

1. Fetch Instruction (from memory)
2. Interpret Instruction (in IR)
3. Fetch data (from memory/registers)
4. Process Data (ALU and registers)
5. Write Data (to memory/registers)

2. Registers

- a. User Visible Registers
- b. Control and Status Registers
- c. Program Status Word (PSW)
- d. Memory and register Organization
- e. Three examples of register organization

a. User Visible Registers

General purpose registers:

Used as operand in any opcode.

Data register

MOV AX, 1

Address register:

Segment pointer

Index register

Stack pointer

MOV DS,AX

a. User Visible Registers (Cont.)

Some points about **registers** and **machine language** (how it is influenced)

- **Implicit registers** -
 - Not mentioned opcode
 - CMA – Complement the Accumulator
 - Reduces the programmer's flexibility.
- **Register No-** Each register has a unique address (we will see this later again).More register means more operand in Machine Language
- **Register Length** – two registers can make one register
- **Flags** – Reflects the result of the last ALU operation. Used for branching

a. User Visible Registers (Cont.)

Condition Codes

Adv:

1. Comparison done using a simple command
2. Branching becomes really easy
3. Multiway branching possible

Dis:

1. Complexity for the hardware
2. Extra hardware connections required
3. Problems in the Pipeline.
4. Tasks can be done without conditional code as well . (Some finds them redundant: **SUB** instead of **CMP**)

b. Control and Status Registers -not used by us during coding

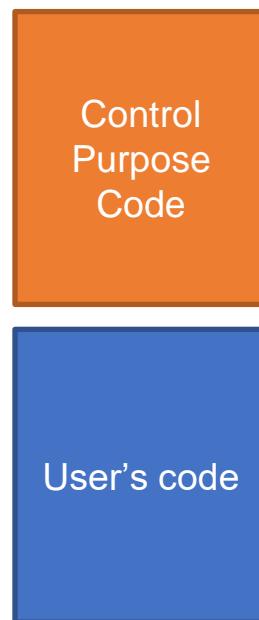
- Some points about the Control and Status registers:
- **PC** – usually stores next instruction. But branching code may update it.
- **IR** – instruction is inserted into it. Opcode and operand are analysed by the Control Unit
- **MAR** – Used for placing the address in the address bus, selects RAM location
- **MBR** – Places data in the bus , reads data from the bus
- **Input and output**- Used to deal with peripherals

c. Program Status Word (PSW)

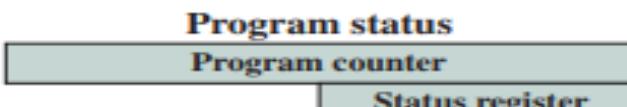
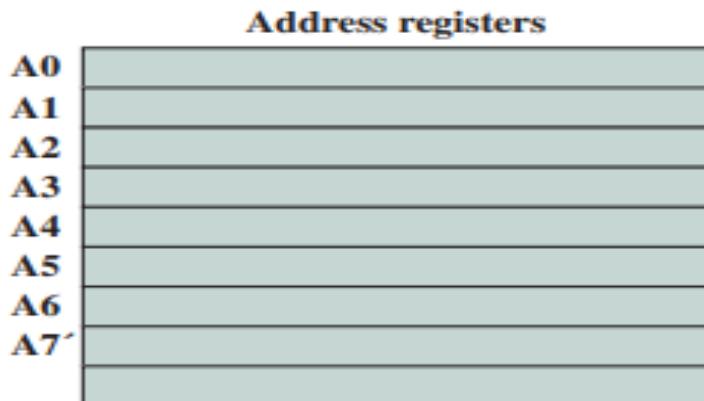
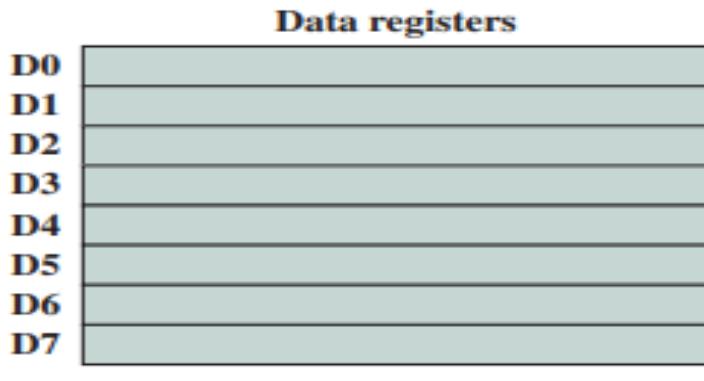
- Sign, Zero, Carry, Equal, Overflow, Interrupt, **Supervisor – VIP or non VIP instructions**
- Last one is there for OS
- Attached with the ALU

d.Memory Organisation

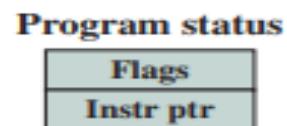
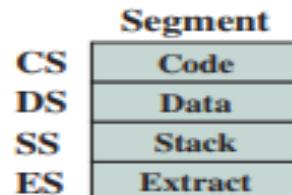
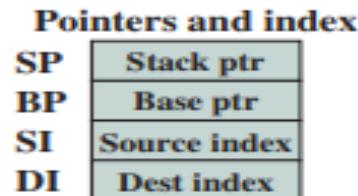
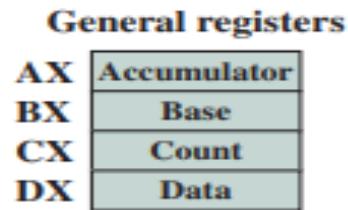
- First portion used for control purposes. Next portion – user code.



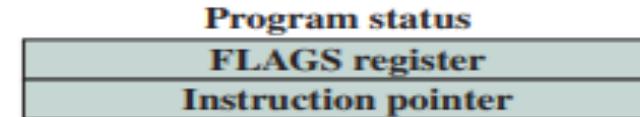
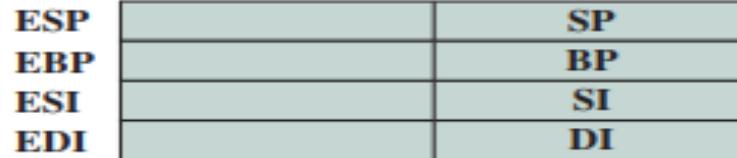
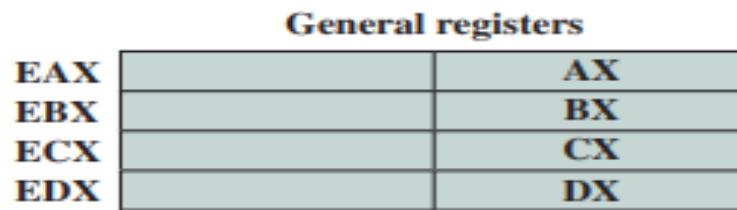
Three Examples of Register Organization



(a) MC68000



(b) 8086



(c) 80386—Pentium 4

Loop uses CX

CMA uses AX

Multiply uses AX

3. Instruction Cycle

Indirect Cycle

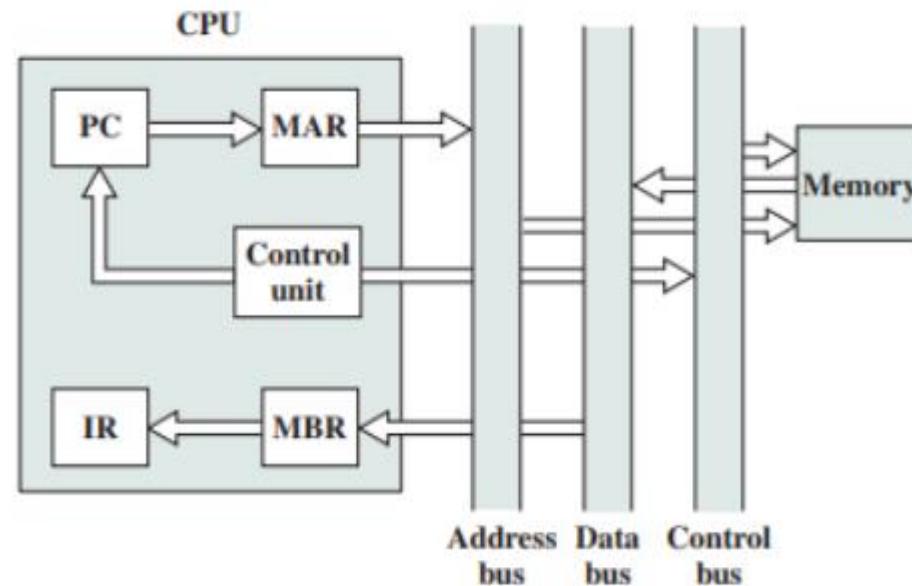
Direct Cycle

Interrupt Cycle

Data Flow in 3 Cases

- Data Flow Direct Cycle
- Data Flow Indirect Cycle
- Interrupt Cycle:

Data Flow Direct/Indirect Cycle



MBR = Memory buffer register

MAR = Memory address register

IR = Instruction register

PC = Program counter

Figure 14.6 Data Flow, Fetch Cycle

Data Flow Interrupt Cycle

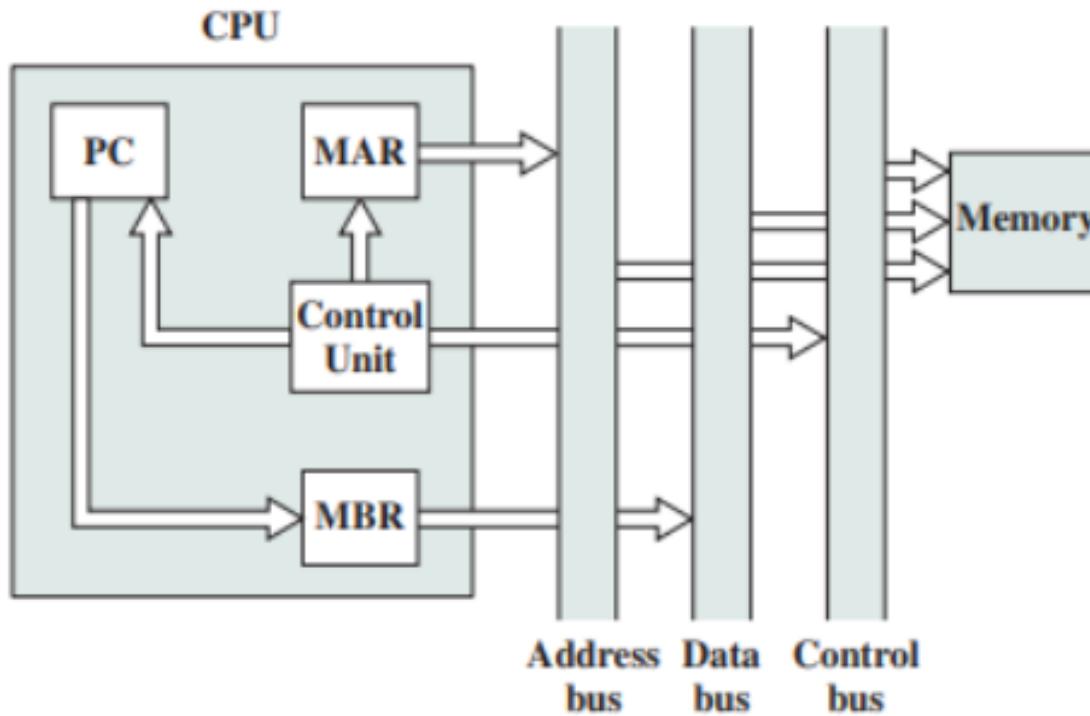
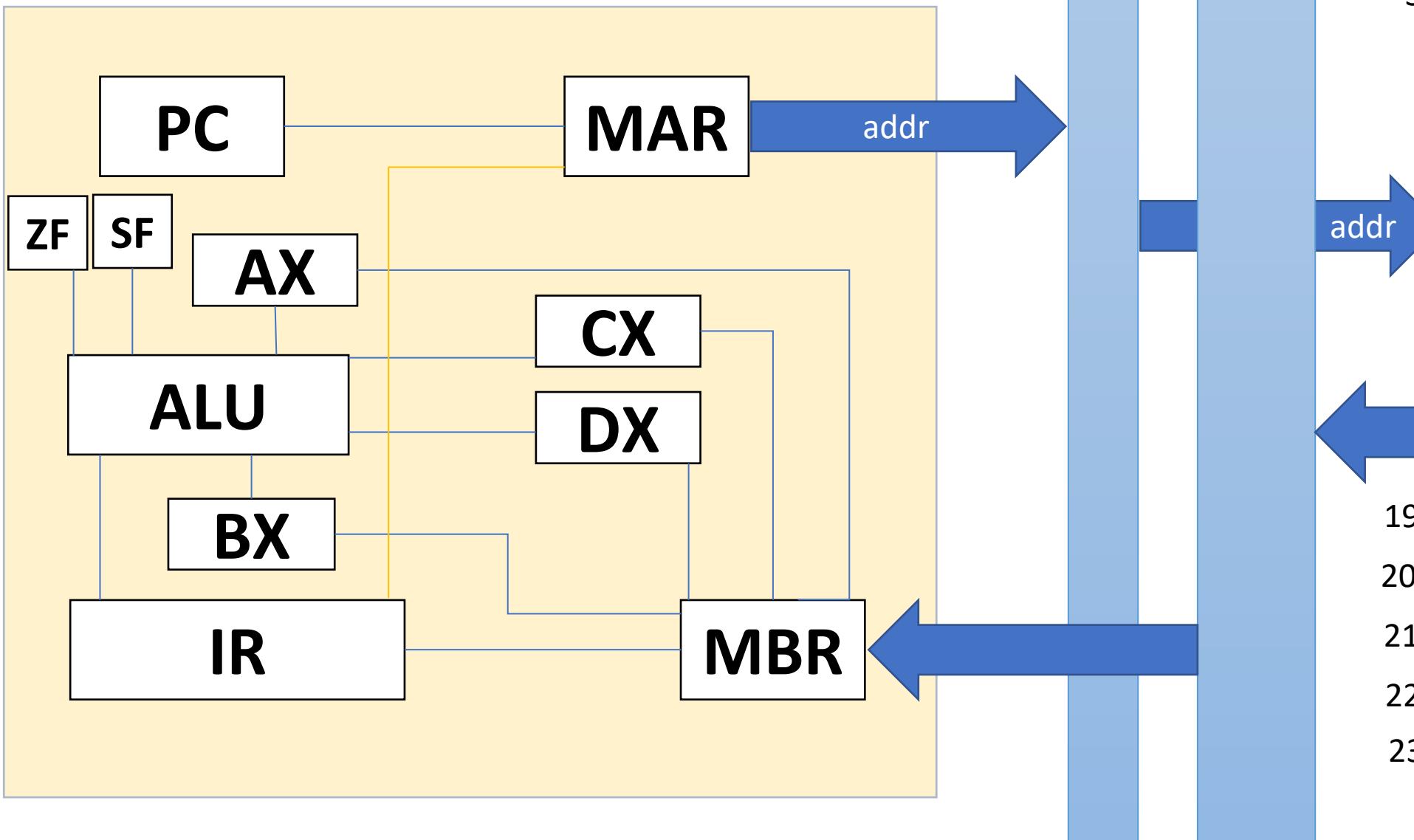
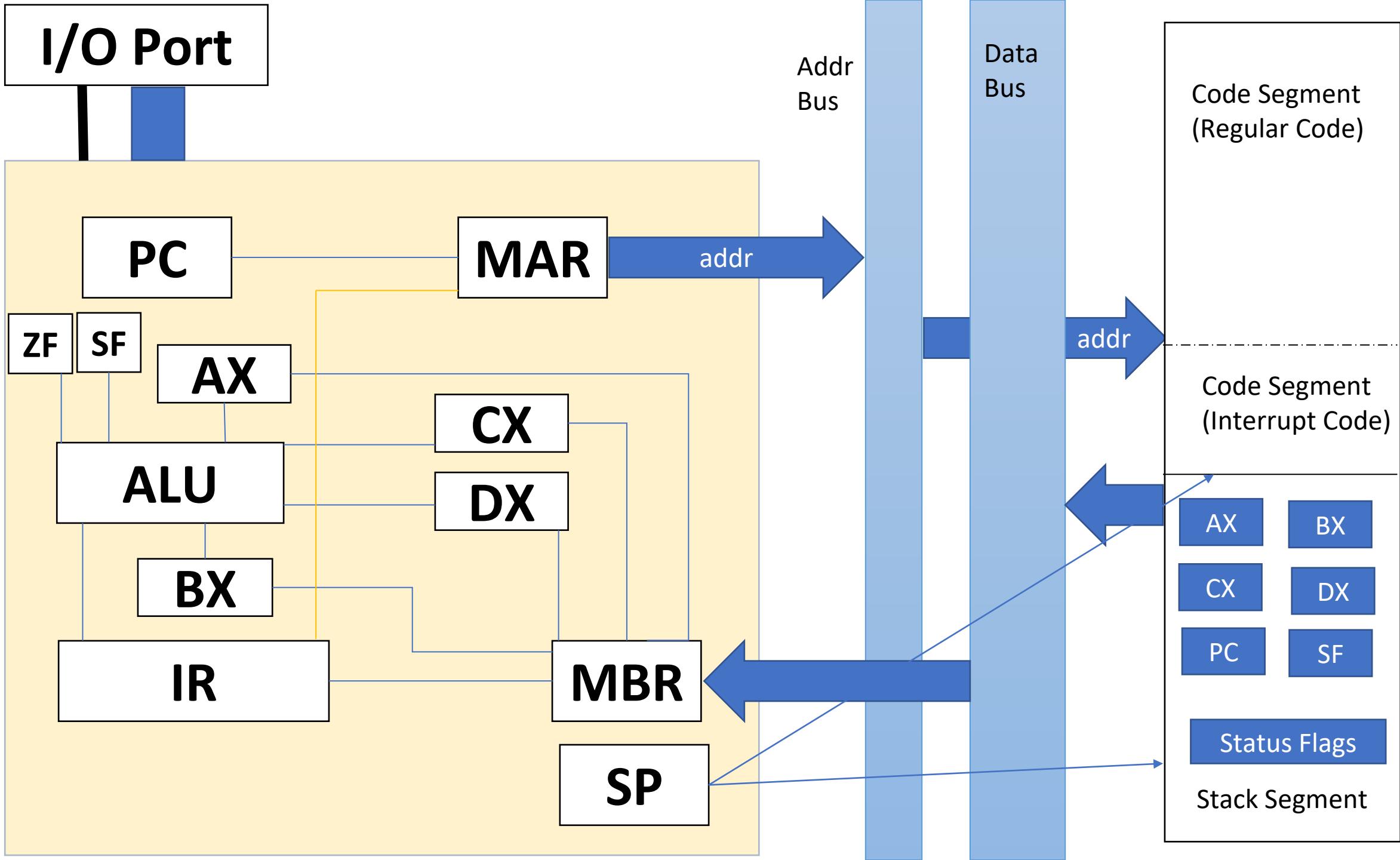


Figure 14.8 Data Flow, Interrupt Cycle

Understanding Interrupt



| | |
|----|--------------|
| 1 | Load DX, #19 |
| 2 | Load AX, #20 |
| 3 | Load BX, #21 |
| 4 | ADD AX,BX |
| 5 | MOV CX,AX |
| 6 | CMP CX,DX |
| 7 | JGE LABEL |
| 8 | MOV CX, #22 |
| 9 | LABEL: |
| 10 | MOV DX,#22 |
| 11 | 100 |
| 12 | 50 |
| 13 | 50 |
| 14 | MOV DX,#12 |
| 15 | MOV DX,#12 |



4. Instruction Pipelining

2 Stage Pipelining

6 Stage Pipelining

Some Problems Associated with pipelining

More Stage more Efficiency?

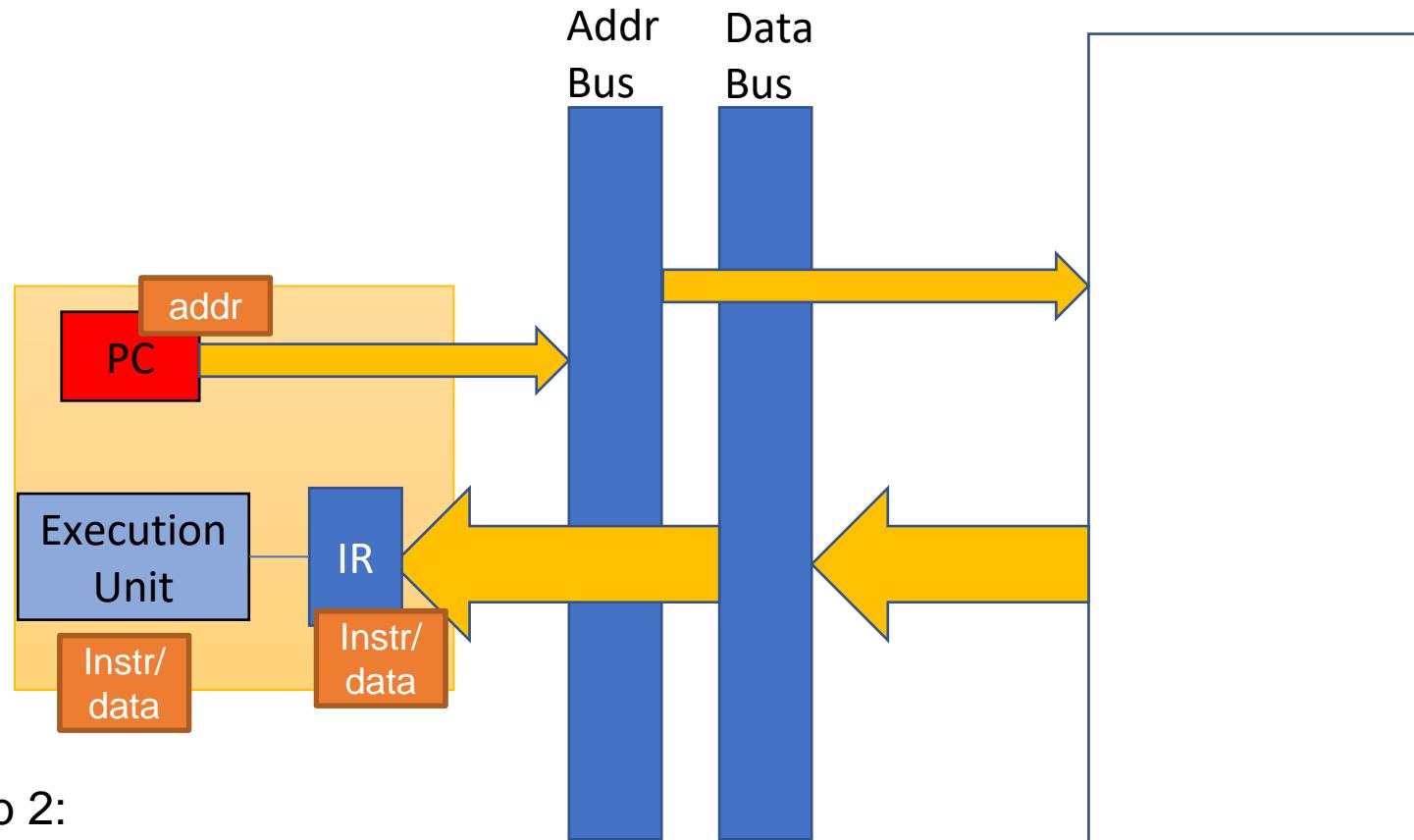
Pipeline Performance



2 Stage Instruction Pipelining

But some pros:

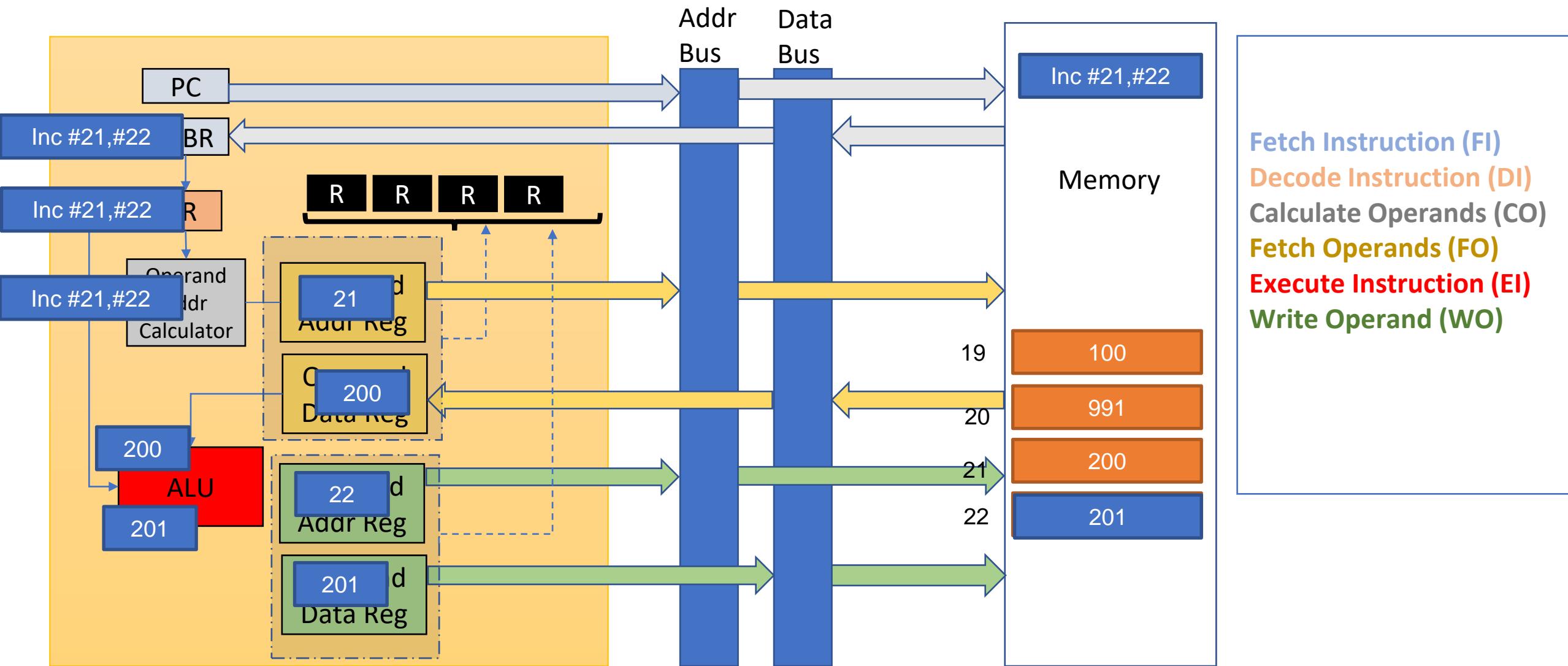
1. Execution may be slower than fetching time.
2. Conditional/branching instruction – which one to fetch?



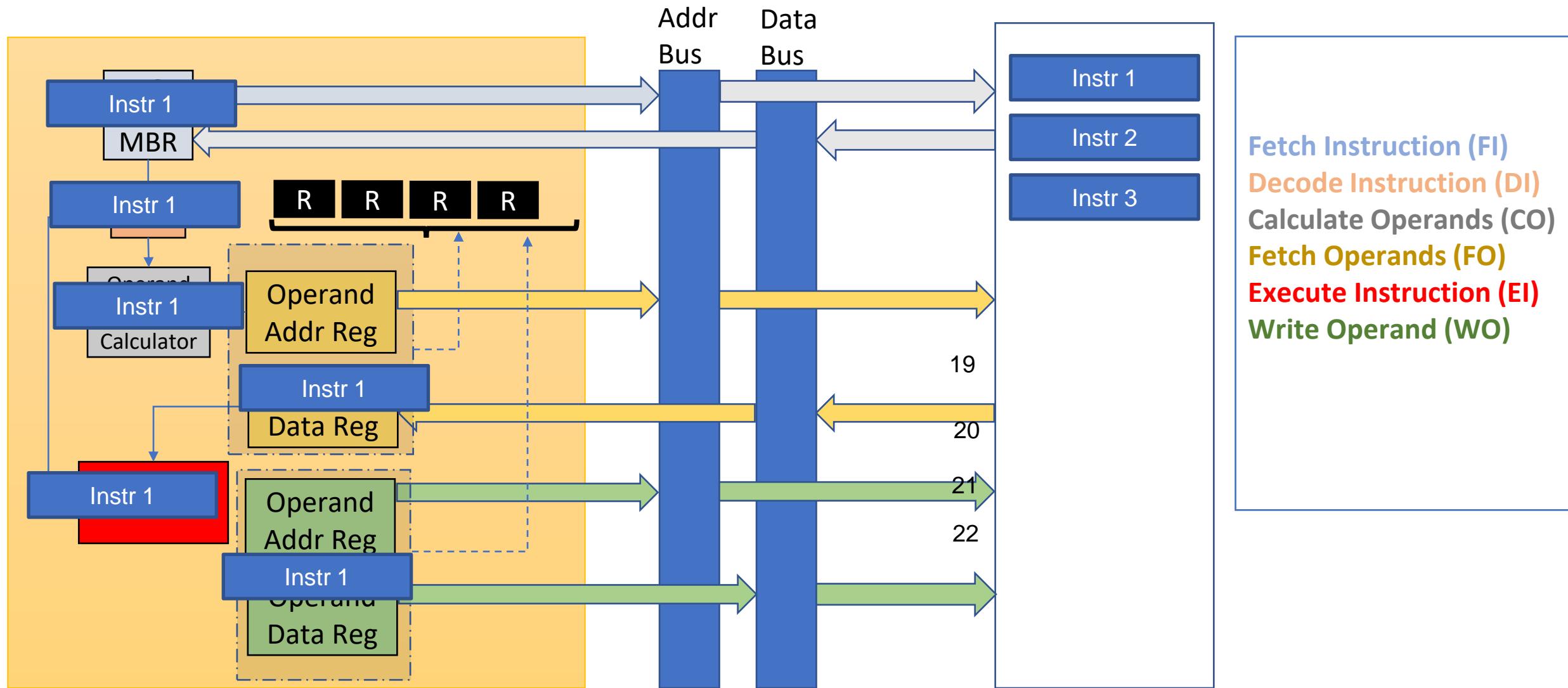
A simple solution to prob 2:

1. Bring in the next instruction
2. If that is used . Great! . Otherwise need to bring in the other branch

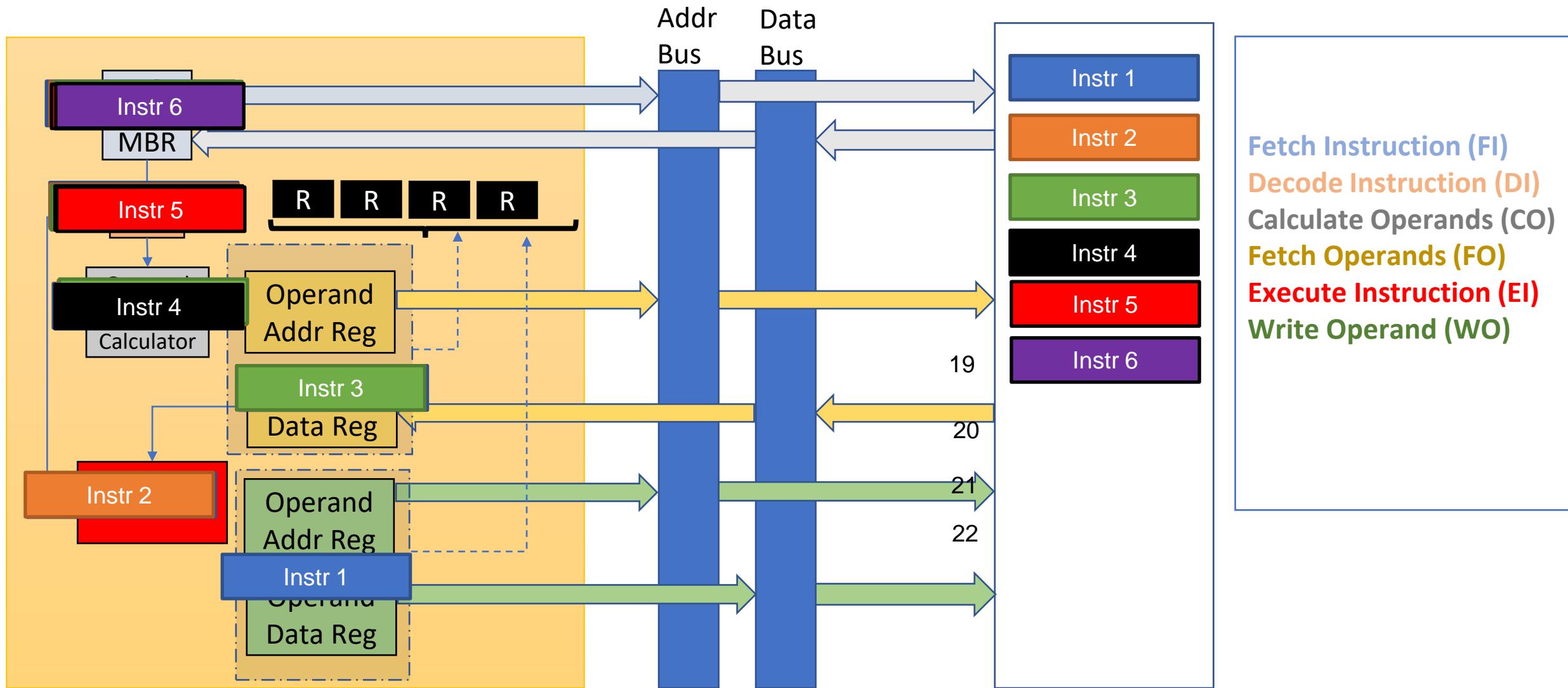
6 Stage Instruction Pipelining



6 Stage Instruction Pipelining



6 Stage Instruction Pipelining



6 Stage Instruction Pipelining (54 unit->14 unit)

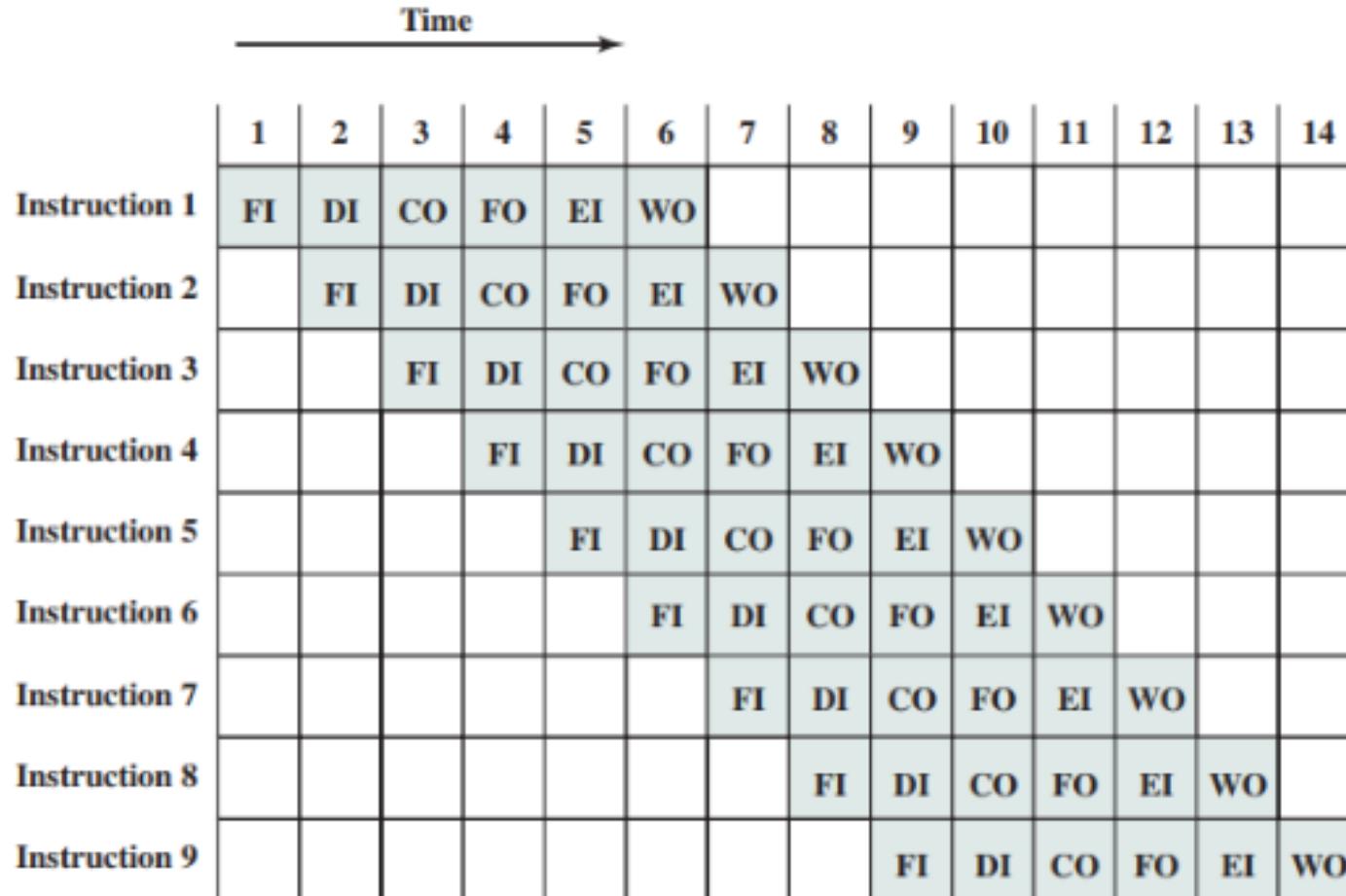
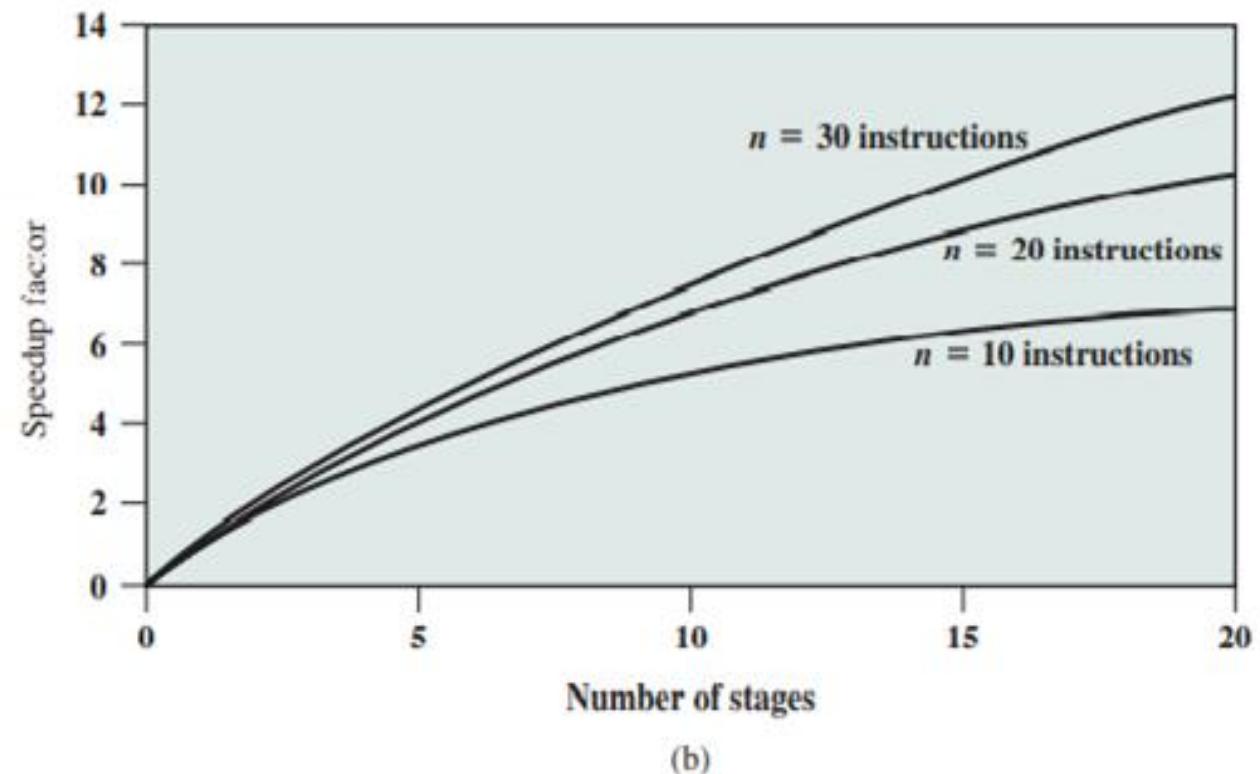
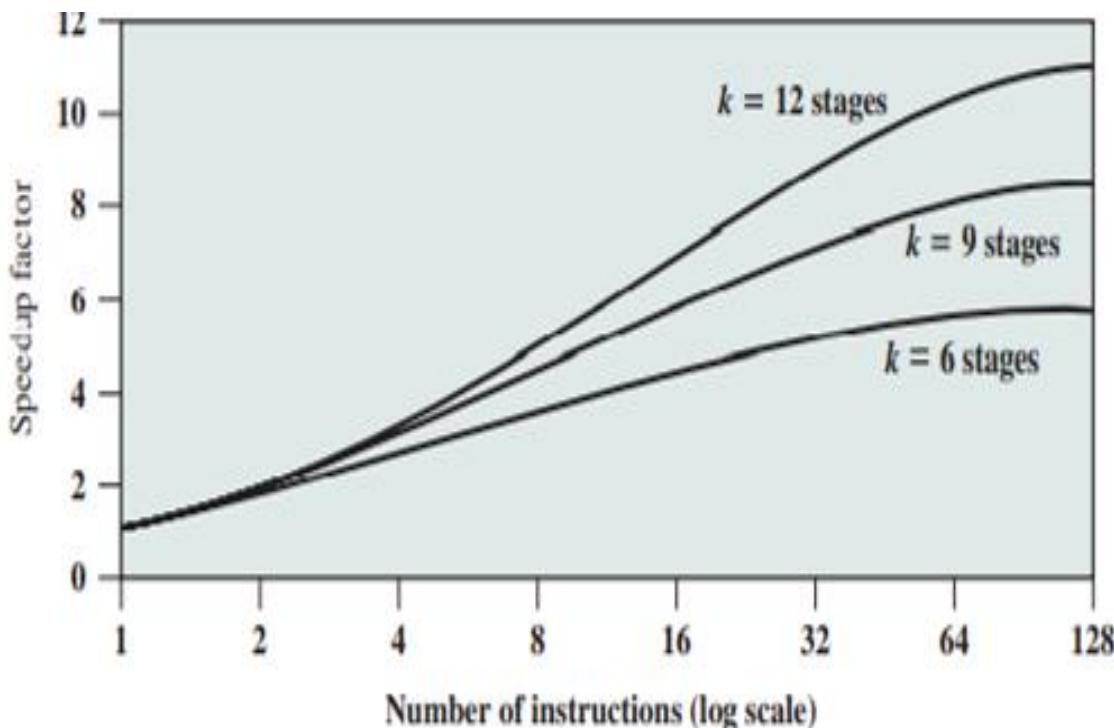


Figure 14.10 Timing Diagram for Instruction Pipeline Operation

Pipeline Performance

- Speedup factor – a measure denoting how much improvement a pipeline has brought about.
- Some mathematical terms (next slide)



(b)

Pipeline Performance

$$\tau = \max_i[\tau_i] + d = \tau_m + d \quad 1 \leq i \leq k$$

$$T_{k,n} = [k + (n - 1)]\tau$$

$$14 = [6 + (9 - 1)]$$

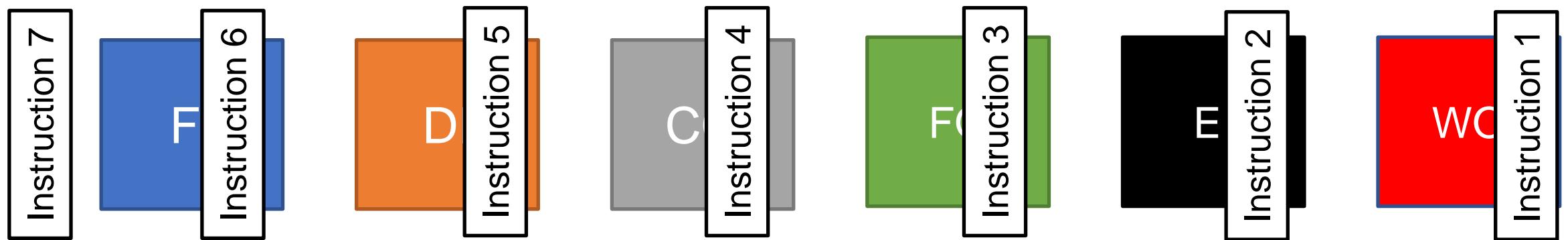
$$S_k = \frac{T_{1,n}}{T_{k,n}} = \frac{nk\tau}{[k + (n - 1)]\tau} = \frac{nk}{k + (n - 1)}$$

Pipeline Performance

$$\tau = \max_i[\tau_i] + d = \tau_m + d \quad 1 \leq i \leq k$$

Cycle Time :

- Time required for all the instructions to advance by 1 stage.



Pipeline Performance

$$T_{k,n} = [k + (n - 1)]\tau$$

Time needed to execute n instructions in a k stage pipeline :

- Time required for all the instructions to advance by 1 stage.

$$14 = [6 + (9 - 1)]$$

Pipeline Performance

$$S_k = \frac{T_{1,n}}{T_{k,n}} = \frac{nk\tau}{[k + (n - 1)]\tau} = \frac{nk}{k + (n - 1)}$$

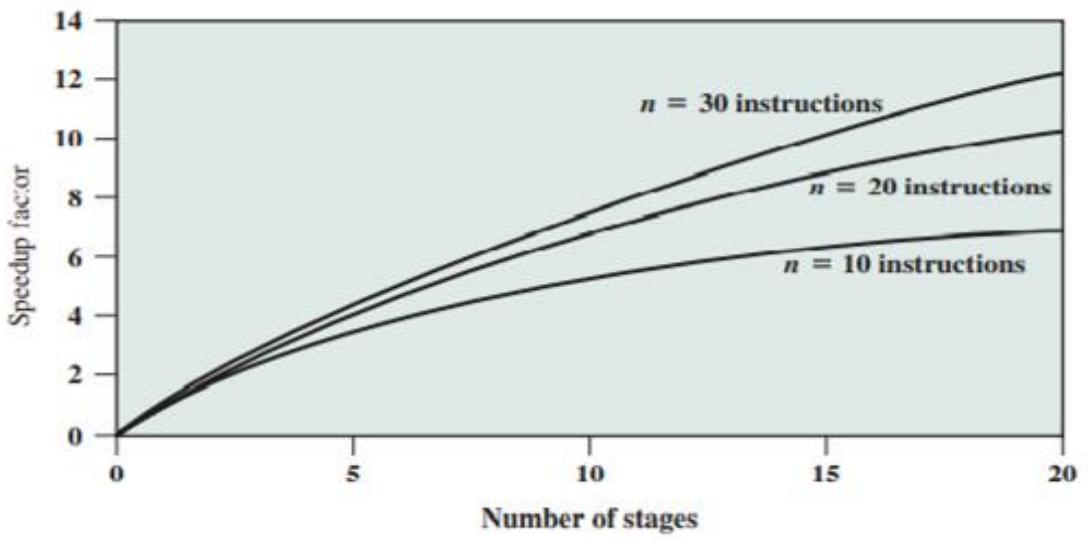
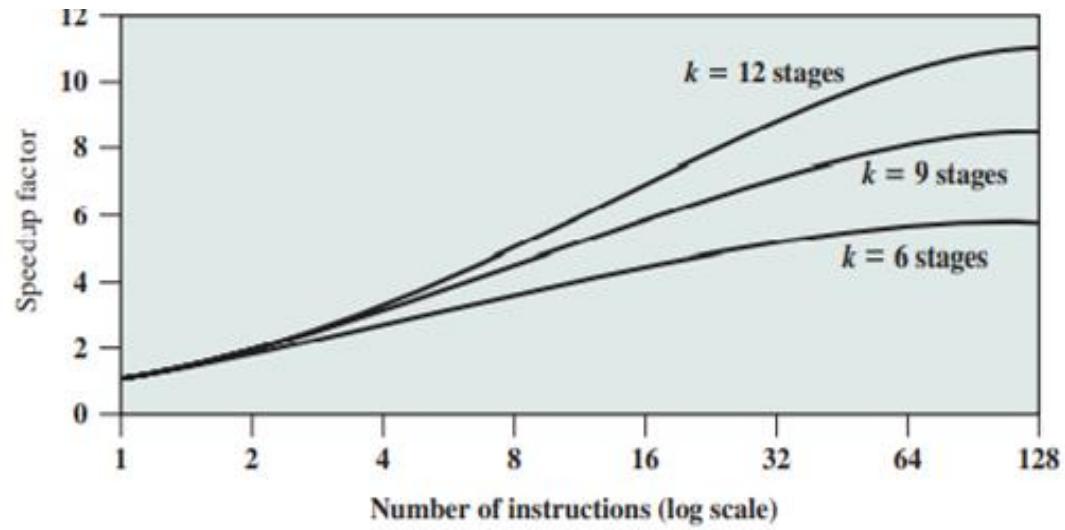
Speed up Factor :

- A ratio
- (Time required for n instructions to complete **without using** the pipeline)/(Time required for n instructions to complete **by using** the pipeline)

Pipeline Performance

Throughput

Throughput for pipelined execution = Number of instructions executed per unit time



- As no of instructions increase
 - speed up factor increase
 - If k (pipeline division) increases, speed up factor increases
- As no of stages increase:
 - Speed up factor increases
 - If n (the number of instructions) increases, the speed up factor increases.

Some downsides of this approach

- Some instruction **stage** may be **unused**,
 - for example **Load AX, #10** ,no memory write operation is taking place here
- **FI,FO,WO** – These instructions block the address bus. So cant be executed at the same time.
- If all stages are **not equal duration** – waiting
- **Register contents** of one instruction may be **altered** by another instruction
- **Conditional branching** - causes difficulty (next slide)

6 Stage Instruction Pipelining (Conditional Branching Problem)

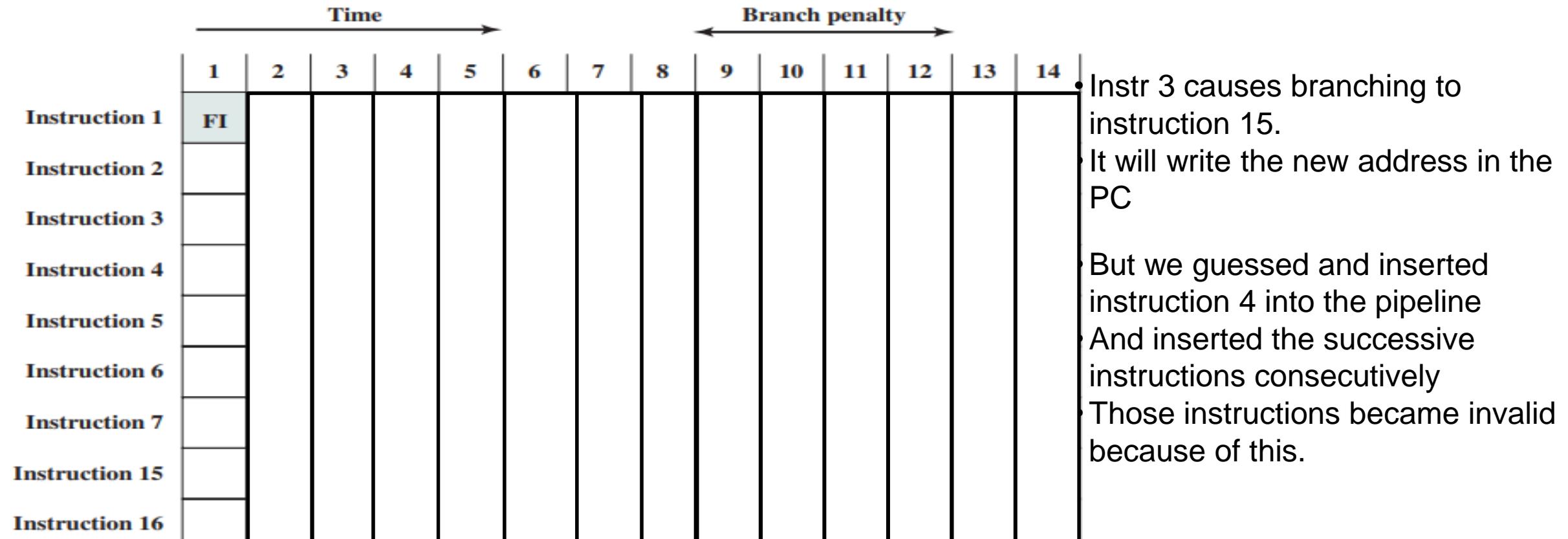


Figure 14.11 The Effect of a Conditional Branch on Instruction Pipeline Operation

INSTRUCTION 3 is a branching instruction

Do more stages in a pipeline mean more efficiency?

Not necessarily

- If two instructions are **logically dependent**, then problems may arise.
This also causes overhead in moving data
- More **control logic** required
- **Latching delay** - Pipeline buffers are slow at times

5. Pipelining Hazard

1. Resource Hazard
2. Data Hazard (3 types)
3. Control Hazard (and 2 ways of dealing with it)

Resource Hazard:

- Also known as **structural hazard**
- Ideal case : (For I1 , FO is taking data from the registers)

| | Clock cycle | | | | | | | | |
|----|-------------|----|----|----|----|----|----|----|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| I1 | FI | DI | FO | EI | WO | | | | |
| I2 | | FI | DI | FO | EI | WO | | | |
| I3 | | | FI | DI | FO | EI | WO | | |
| I4 | | | | FI | DI | FO | EI | WO | |

(a) Five-stage pipeline, ideal case

Resource Hazard (Cont.):

- Also known as structural hazard
- Hazard case : (For I1 , FO is taking data from the memory, Hence I3 starts a bit late)

| | | Clock cycle | | | | | | | | |
|-------------|----|-------------|----|------|----|----|----|----|----|----|
| | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| Instruction | I1 | FI | DI | FO | EI | WO | | | | |
| | I2 | | FI | DI | FO | EI | WO | | | |
| | I3 | | | Idle | FI | DI | FO | EI | WO | |
| | I4 | | | | | FI | DI | FO | EI | WO |

(b) II source operand in memory

Figure 14.15 Example of Resource Hazard

Data Hazard:

- Suppose there are two instructions. The data of the first instruction is used by the second instruction.
- If the instructions are executed **without pipeline**- no problem
- But we are **pipelining**

```
ADD EAX, EBX /* EAX = EAX + EBX
```

```
SUB ECX, EAX /* ECX = ECX - EAX
```

Data Hazard:

I2 needs the data from I1, so it before fetching, it waits for that data to be written

ADD EAX, EBX /* EAX = EAX + EBX

SUB ECX, EAX /* ECX = ECX - EAX

Clock cycle

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---------------------|----|----|----|------|----|----|----|----|----|----|
| ADD EAX, EBX | FI | DI | FO | EI | WO | | | | | |
| SUB ECX, EAX | | FI | DI | Idle | | FO | EI | WO | | |
| I3 | | | FI | | | DI | FO | EI | WO | |
| I4 | | | | | | FI | DI | FO | EI | WO |

Figure 14.16 Example of Data Hazard

Three types of data hazards based on dependencies:

Read after write (true dependency):

- Need to wait till the data is written, say I1 will write the data.
- Then the written data is to be read, I2 will read the data
- The previous case is the example.

Write after read (anti dependency):

- An instruction will read a data from a place, after that an instruction will write the data
- I1 will read the data
- I2 will write at that place once I1 has finished reading

Write after write (output dependency):

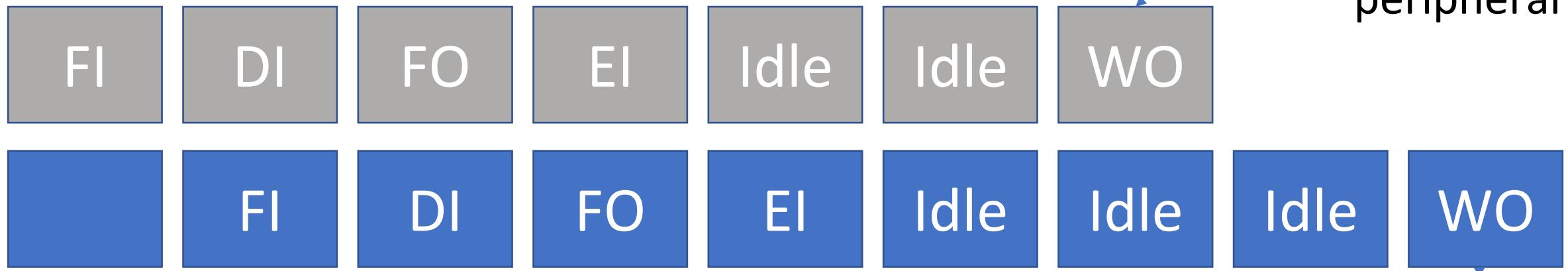
- I1 will write first.
- I2 will write second.
- If it the sequence is not followed, then there will be a hazard

**BASICALLY DATA HAZARD REFERS WAITING FOR DATA
Because of some form of dependency**

Write after Read



Write After Write



Writes to an external port register, it will be read by a peripheral

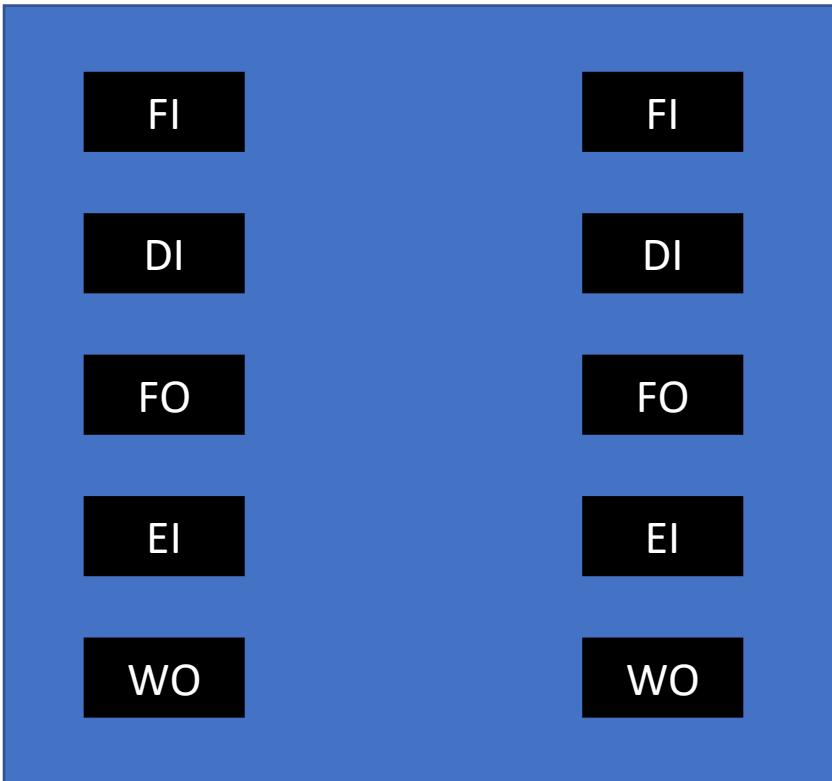
Also writes to the external port register, it will be read next by a peripheral

Control Hazard

- **Wrong instruction** brought in due to branching
- The selected instruction is to be **discarded**
- Thus **waste of time** (branch penalty)

Control Hazard (Dealing with branches)

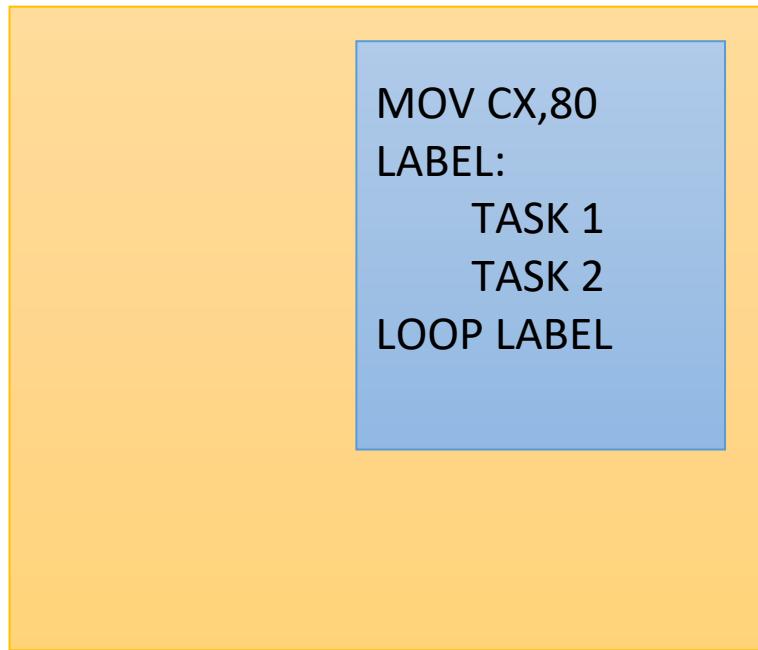
Multiple Streams



Two branches can be
fed through two
streams.

Loop Buffer

- N most recently fetched instructions are taken.
- If any branching instruction arrives, the loop buffer is checked



Loop Buffer

Benefits:

- Memory access not needed, less time
- If all the instructions of a loop falls into it -> super efficient

6. Branch Prediction

5 approaches

3 of them are static

2 of them are dynamic

Predict never taken/ always taken (1st/2nd)

1st two static approaches –

- assumes a branch may be taken or not taken
- Statistics show that more than 50% of the time branch is taken
- So bringing in the branch is better than bringing in the sequential path.

But paged machine-

- page fault may occur (the first instruction of the branch is in another page)
- Necessary avoidance mechanism can be taken.

3rd Approach

- 3rd static approach - Certain opcodes will take branch.
- 75% success rate

Dynamic Approaches

Taken not taken switch(the history bit)

| Branch Instruction | History that will help in prediction |
|--------------------|--------------------------------------|
| JGE LABEL | 0 (branch not taken) |
| JZ LABEL2 | 1 (branch taken) |
| JNZ LABEL3 | 1 (branch taken) |
| --- | --- |
| --- | --- |

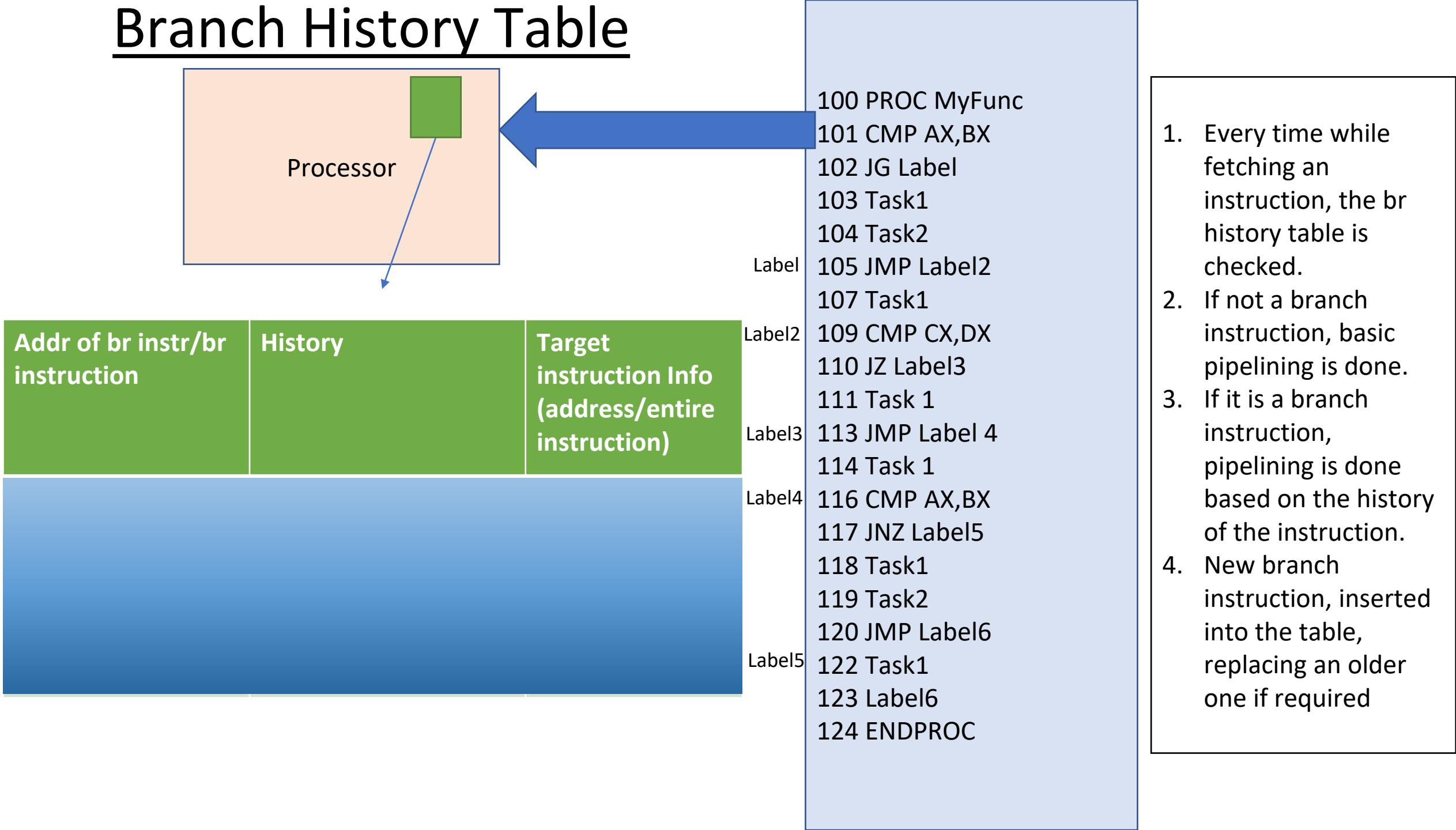
A temporary high-speed storage is maintained. This can be done using a separate h/w or a cache. If an erroneous prediction occurs twice, then the bit is changed. More than 1 bit can be used to keep more info

Dynamic Approaches

Branch History Table

| Address of a branching instruction | History about its behavior | Target Instruction Information |
|---|-----------------------------------|---------------------------------------|
| | | |
| | | |
| | | |
| | | |
| | | |

Branch History Table



Thank you

Chapter 15 : Reduced Instruction Set Computers

Semantic Gap

Semantic gap:

There are differences between the operations provided in HLLs (High Level Languages) and those provided in the computer architecture at low level.

CISC and RISC

- Complex Instruction Set Computer
- Reduced Instruction Set Computer

First Approach (CISC)

- **Code** in high level language is written
- It is the **compiled** by a compiler
- It generates **complex** Instructions which are comparatively few in number

Complex Instructions mean:

- Requires **complex** hardwares
- **Variable** instruction size
- **Different clock cycles** for different instructions

Another Approach (RISC):

- **Code** in high level language is written
- It is the **compiled** by a compiler
- It generates **simple Instructions** which are comparatively more in number

Simple Instructions mean:

- Requires comparatively **simpler** hardware
- **Same instruction size**
- **Same clock cycles** for almost all instructions

The Core distinction

| A task done using a CISC processor instruction(s) | Same task done using a RISC processor instruction(s) |
|---|---|
| ADD #10, #11, #12 | LOAD R1, # 10 LOAD R2, # 11 LOAD R3, # 12 ADD R2,R3 MOV R1,R2 |

RISC vs CISC

| SN | CISC | RISC |
|----|--|---|
| 1 | Emphasis on hardware | Emphasis on software (compiler) |
| 2 | Complex instructions requiring several clock cycles. | Simple instructions requiring fewer clock cycles. |
| 3 | Load and Store instructions are incorporated in the main instructions. ADD #10,#11,#12 | Separate Load and Store instructions are required. Load AX, #12 |
| 4 | Slower as more clock cycle per instruction | Faster as each instruction takes less number of clock cycles. |

RISC vs CISC (cont.)

| SN | CISC | RISC |
|-----------|--|--|
| 5 | Main objective is less machine code | Main objective is simplicity of hardware and more speed per instruction. |
| 6 | More hardware oriented | More software oriented (i.e compiler oriented) |
| 7 | Variable instruction size | Fixed instruction size |
| 8 | Uses complex addressing modes | Uses simple addressing modes |
| 9 | Expensive hardware design | Economic hardware design |

Using Registers : Load, Operate, Store

- Load and **store** operations deliver the data to the **registers**.
- RISC processors tend to make the **most use of registers** and keeps the **partial results** into the registers (not in the memory)
- Then the registers' data are taken in and **operated** on by the **ALU**.
- But this tendency of RISC processor creates an **increased demand** for **registers**
- Hence, it is required to use the **registers** in an **efficient manner**.

Optimizing Register Usage

Two approaches are there to optimize register usage:

The **software based** approach:

- **Analyze** the code.
- Check register **usage period**.
- Try to use as **few registers** as possible (using **graph coloring**.)

The **hardware approach** is

- We can simply add **more register**, but this is not a feasible solution.
- We can use registers using an arrangement called **The Circular Buffer Organization of Overlapping Windows**.

The Software based approach

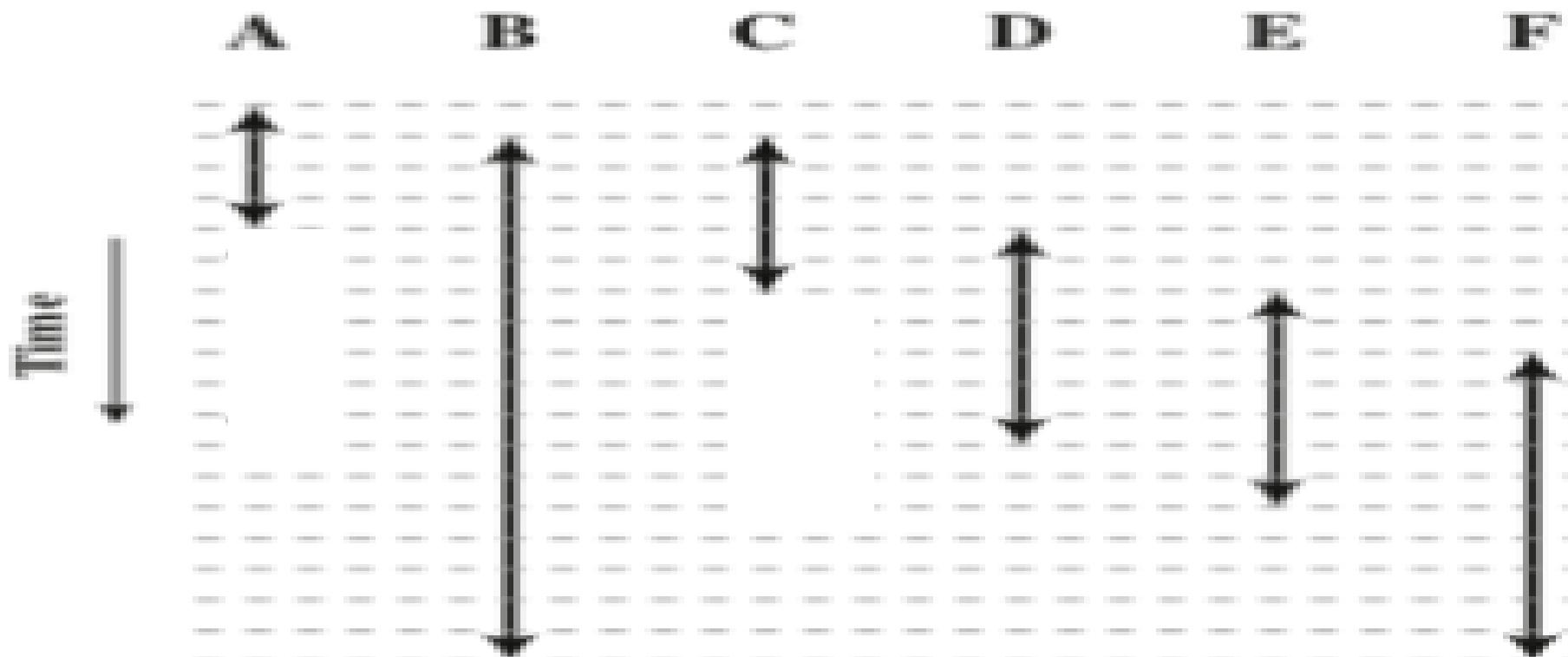
Let us observe the following scenario:

- The code you wrote uses **6 registers**.
- But by **analyzing** that code, it can be seen that,
- **4 registers** can get the job done.

The Software based approach (cont.)

After analyzing the code, the following usage information is obtained about the registers.

Basically showing **the period** during which the register will be containing required data

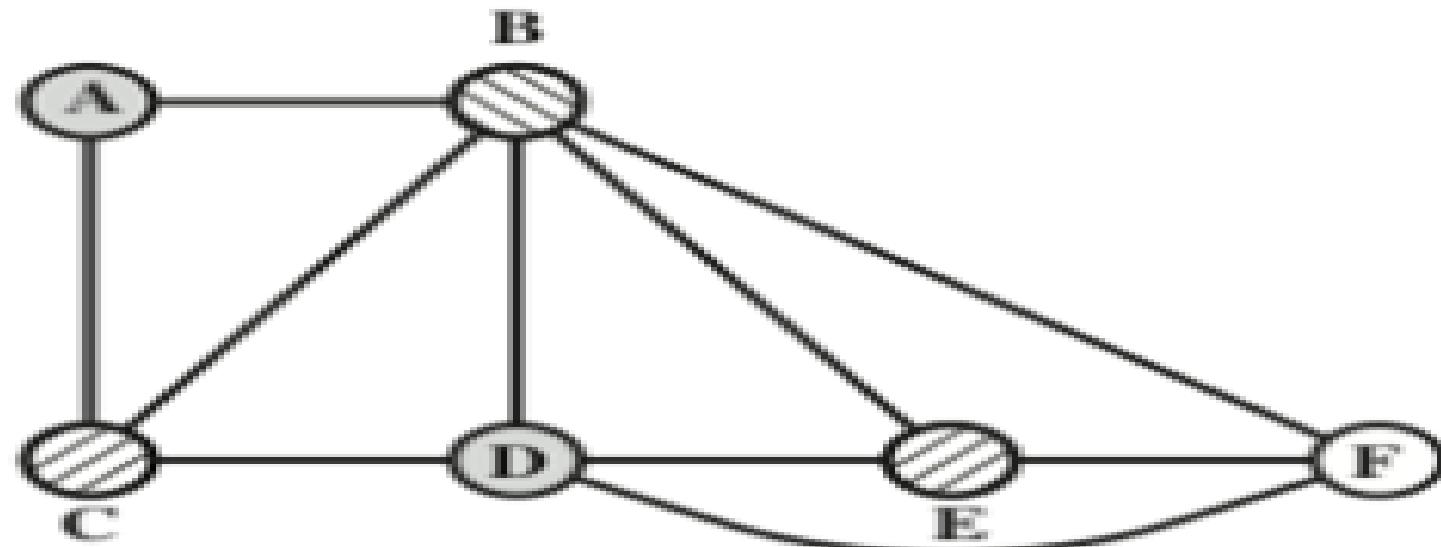


The Software based approach (cont.)

Based on the **overlapping register** usage, a **graph** can be generated:

The Software based approach (cont.)

Based on the overlapping register usage, a graph can be generated:



(b) Register interference graph

Brute Force based but quick approach you can use for graph coloring

- Coloring with the lowest number

1. Color first vertex with first color.

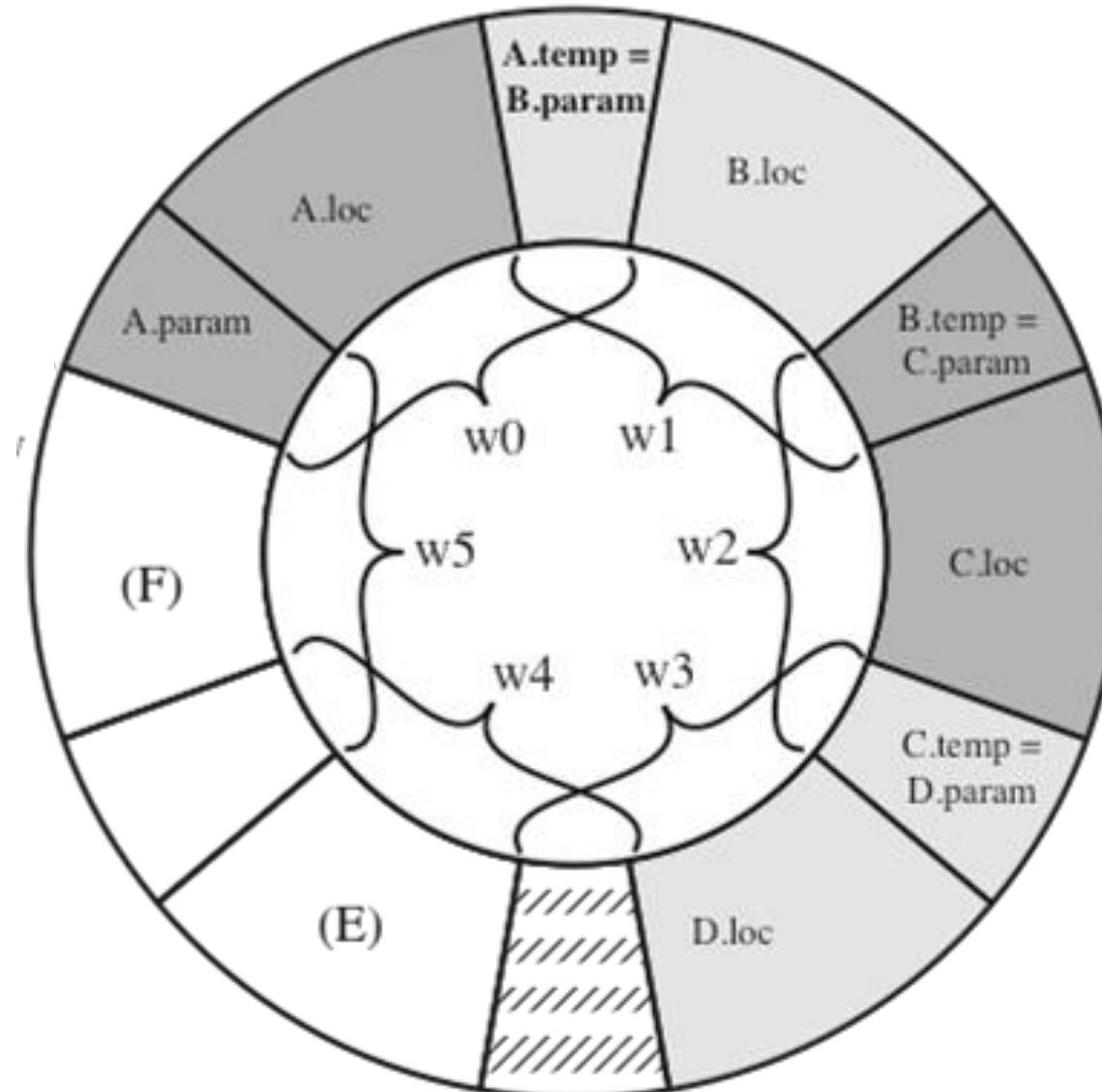
2. Do following for remaining $V-1$ vertices.

..... a) Consider the currently picked vertex and color it with the lowest numbered color that has not been used on any previously colored vertices adjacent to it. If all previously used colors appear on vertices adjacent to v , assign a new color to it.

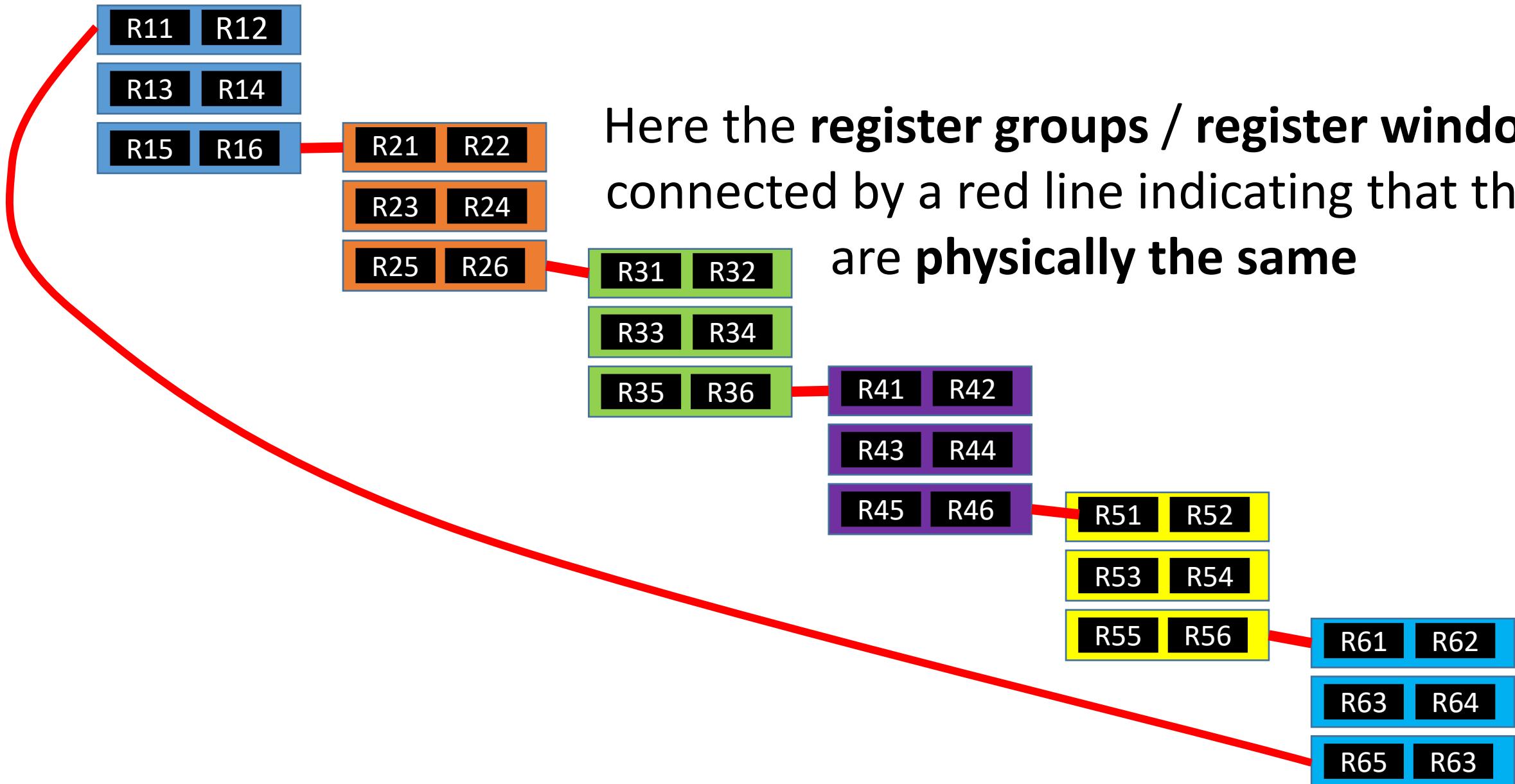
The Hardware based Approach:

- Recall that we can use **registers** to **pass parameters** to a function.
- But this approach has some limitations:
 - Limited by the **number of registers**.
 - Not as simple as using stack for **passing parameters**.
 - Not simple to use if **nested function** calls occur.
- Let us use some more registers and **reorganize**

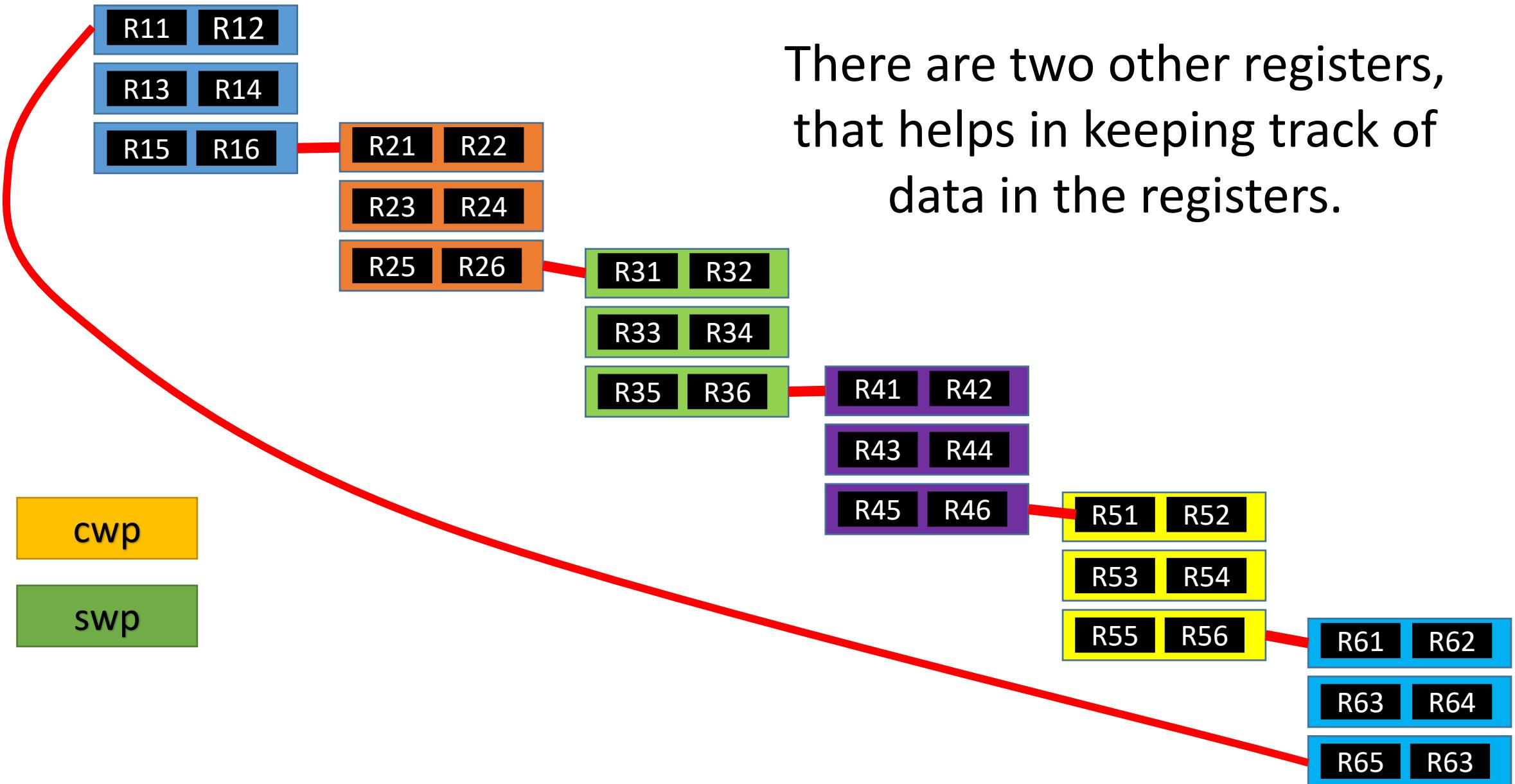
Circular Buffer Organization of Overlapping Windows



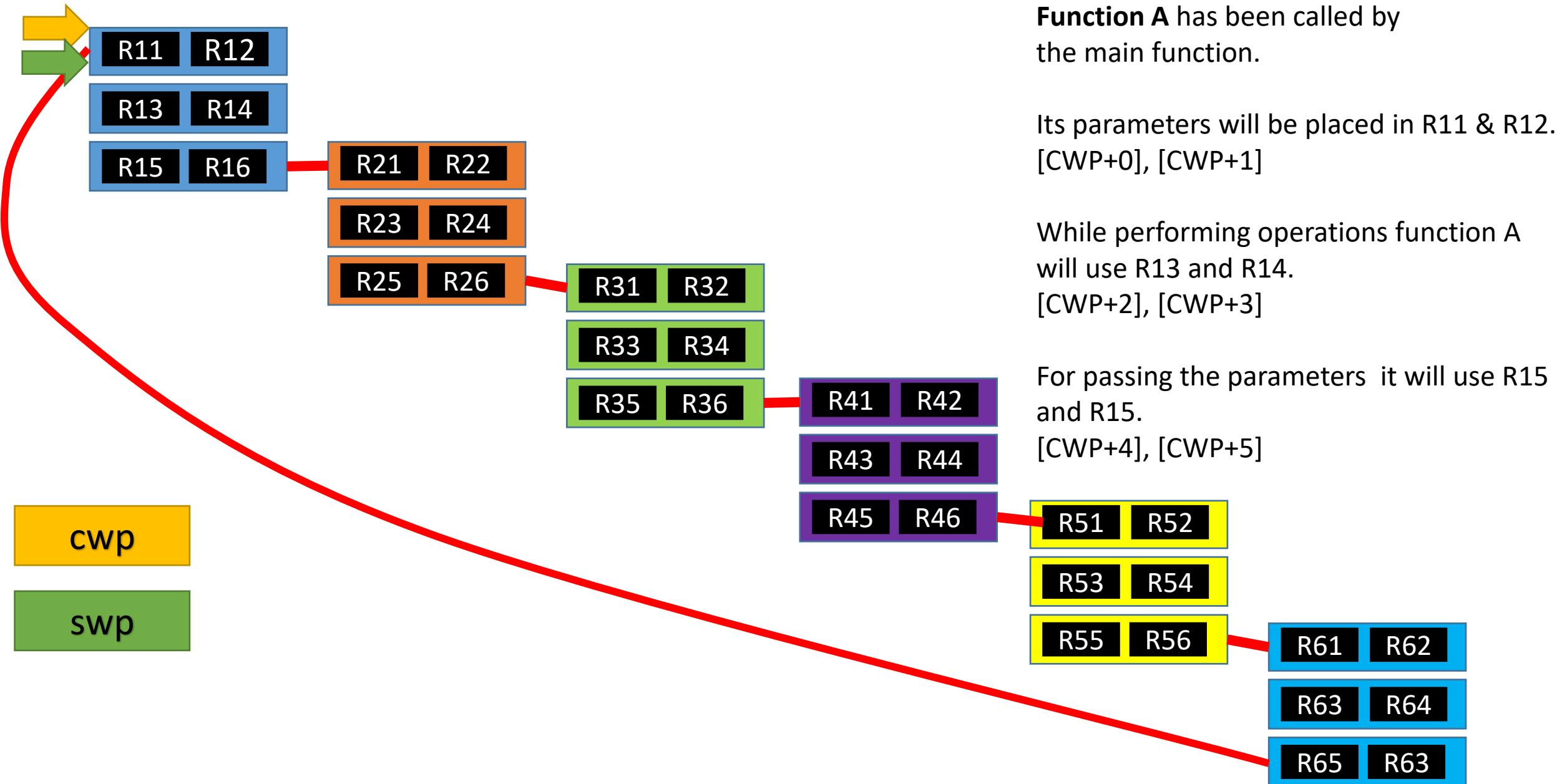
Circular Buffer Organization of Overlapping Windows (Cont.)



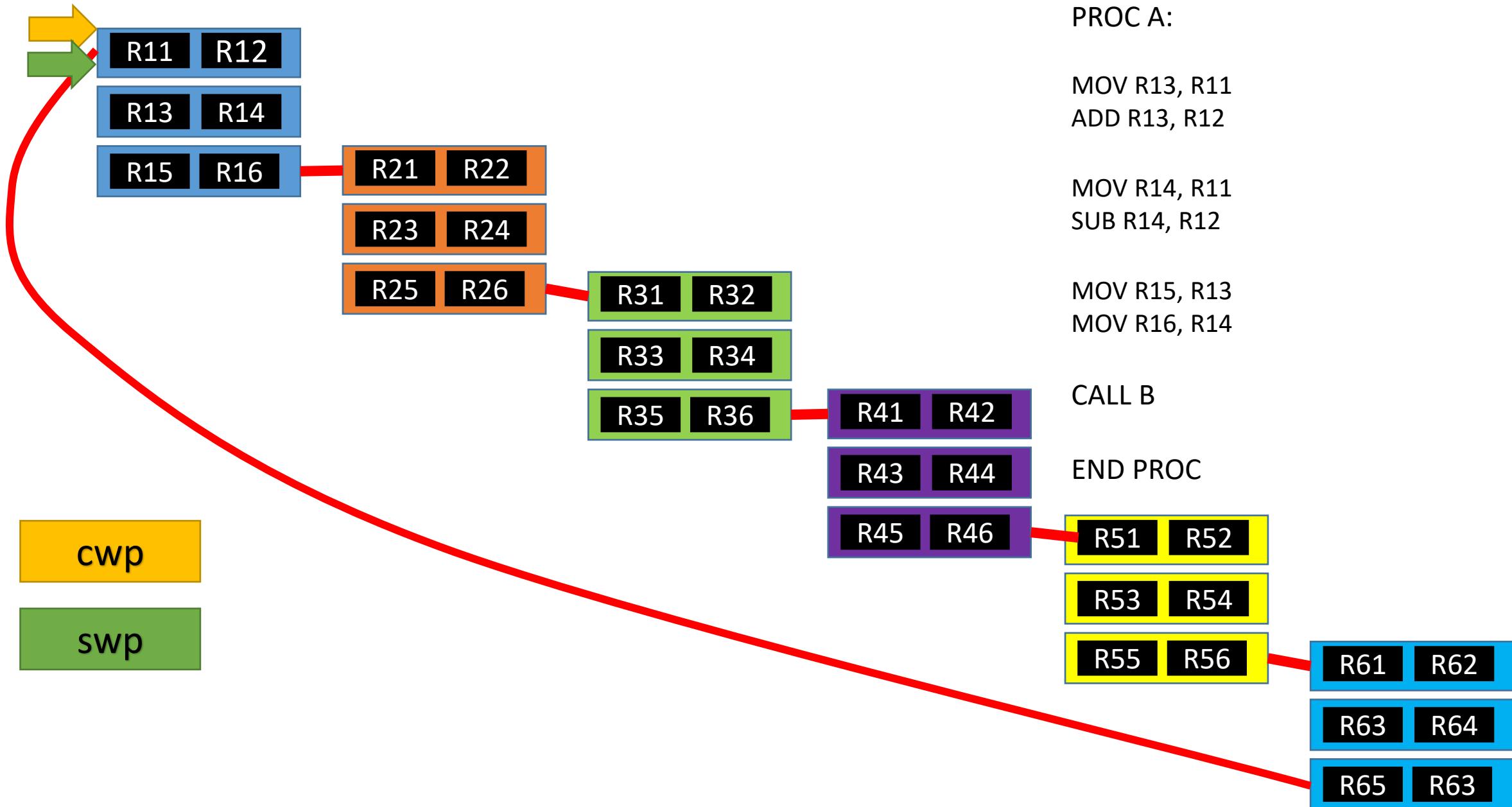
Circular Buffer Organization of Overlapping Windows (Cont.)



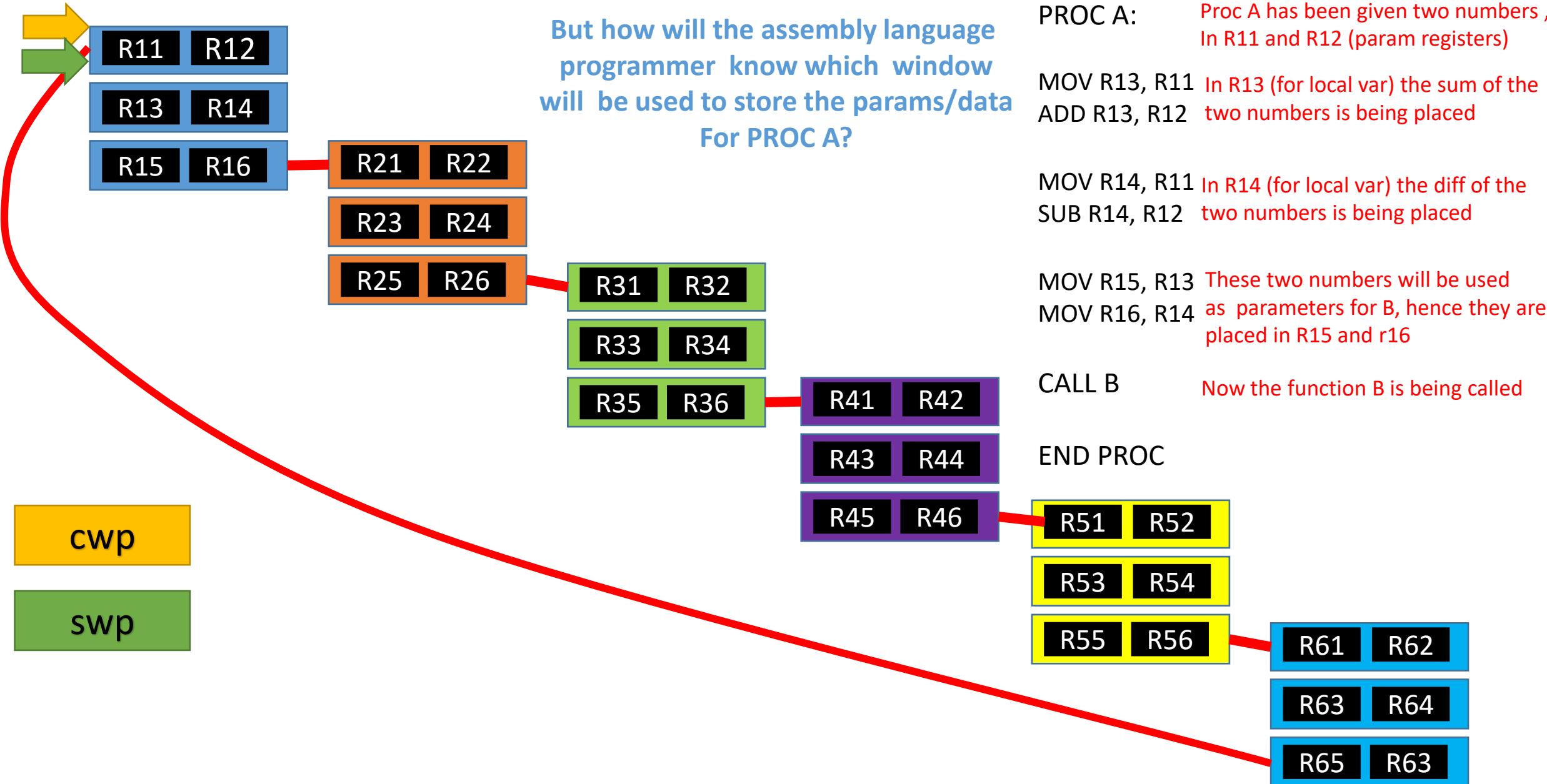
Circular Buffer Organization of Overlapping Windows (Cont.)



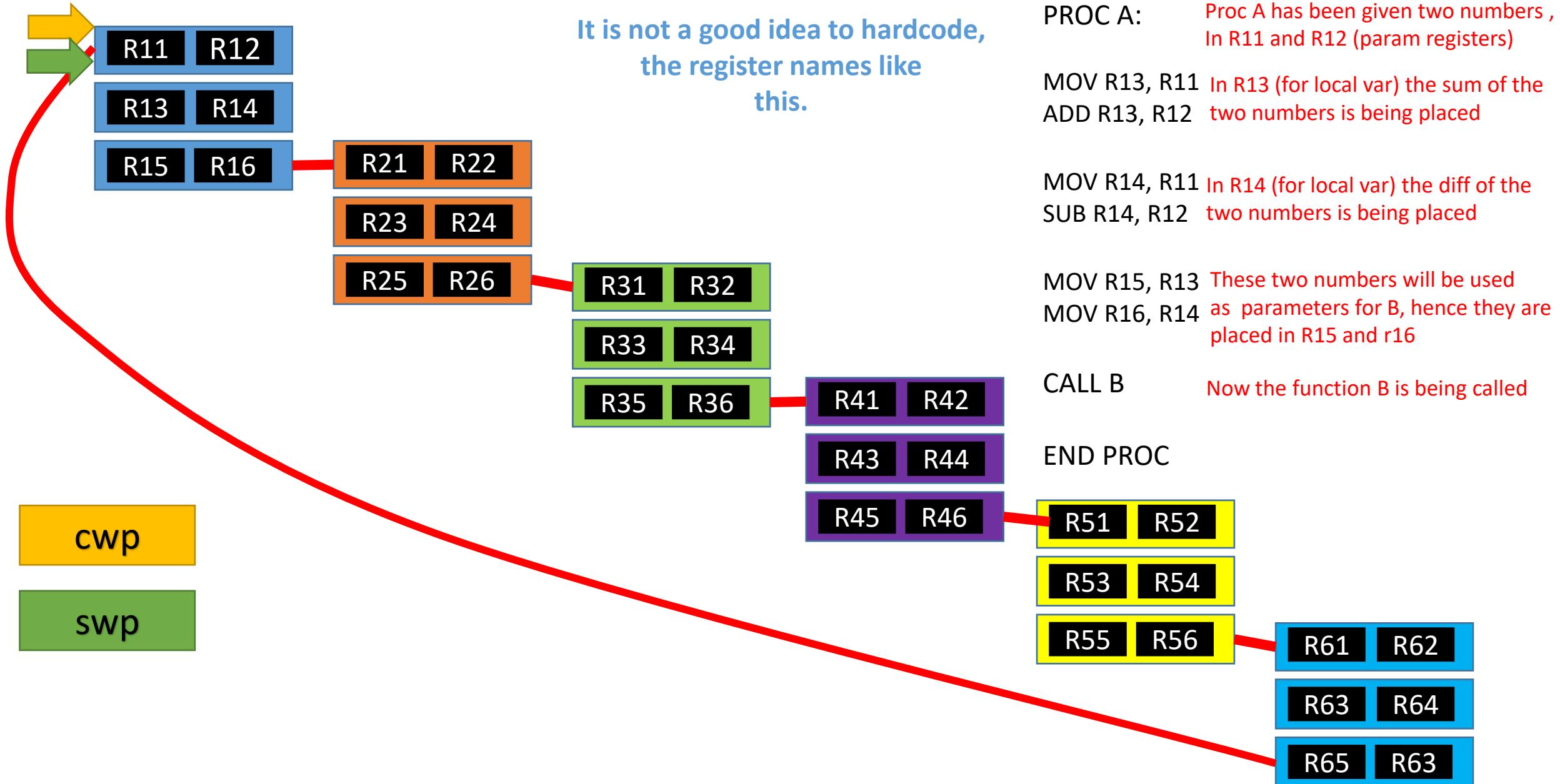
Circular Buffer Organization of Overlapping Windows (Cont.)



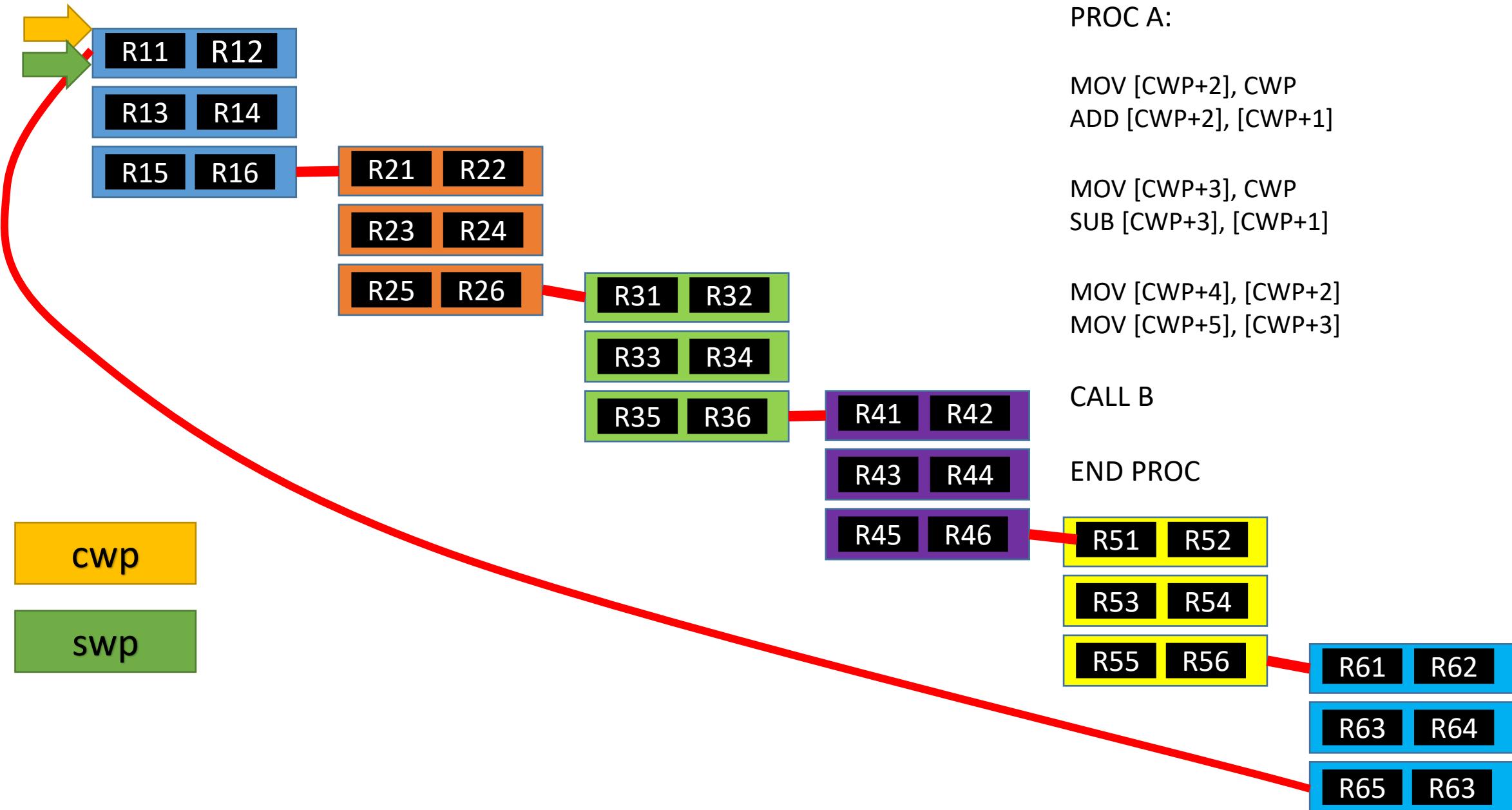
Circular Buffer Organization of Overlapping Windows (Cont.)



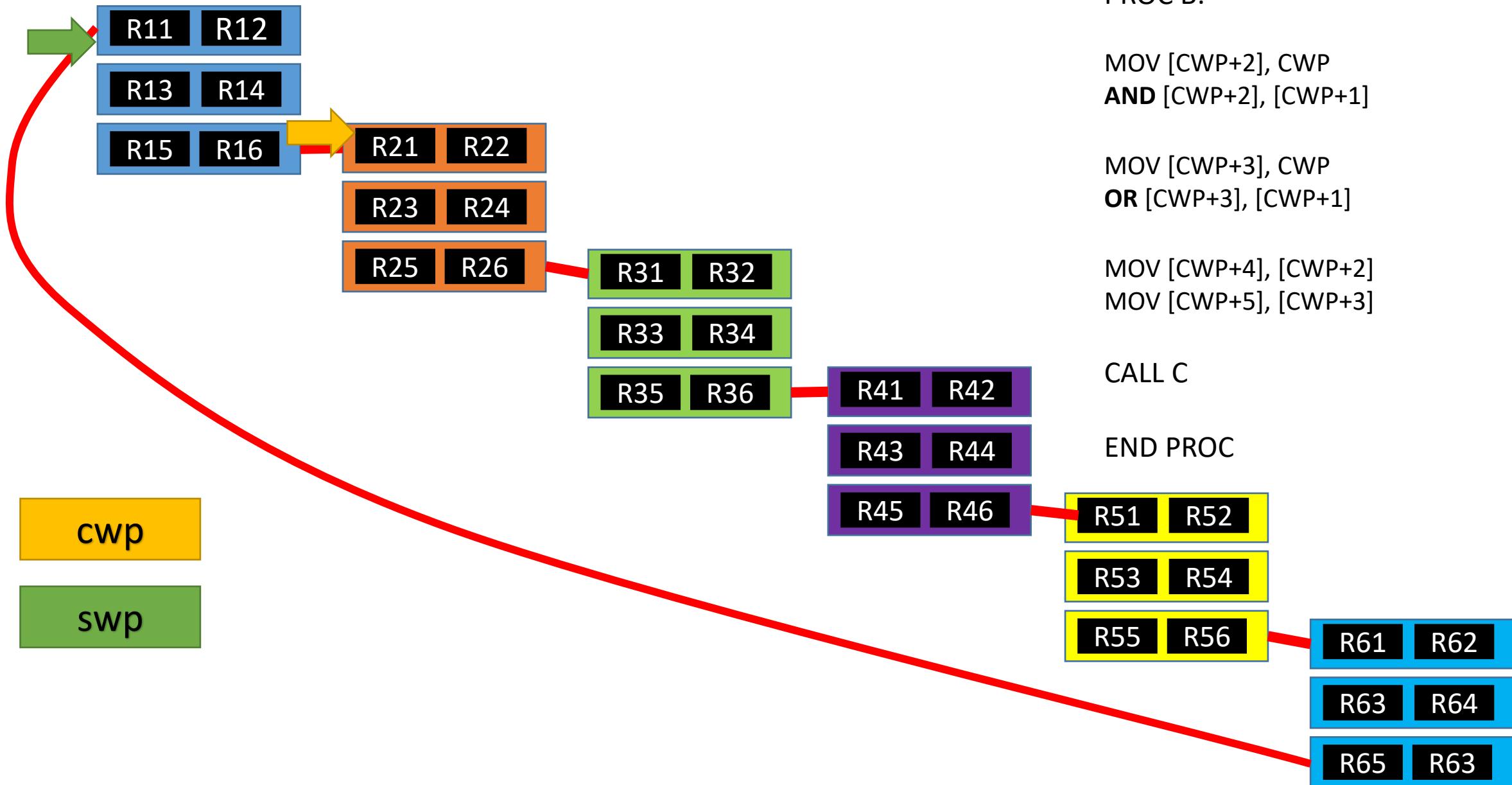
Circular Buffer Organization of Overlapping Windows (Cont.)



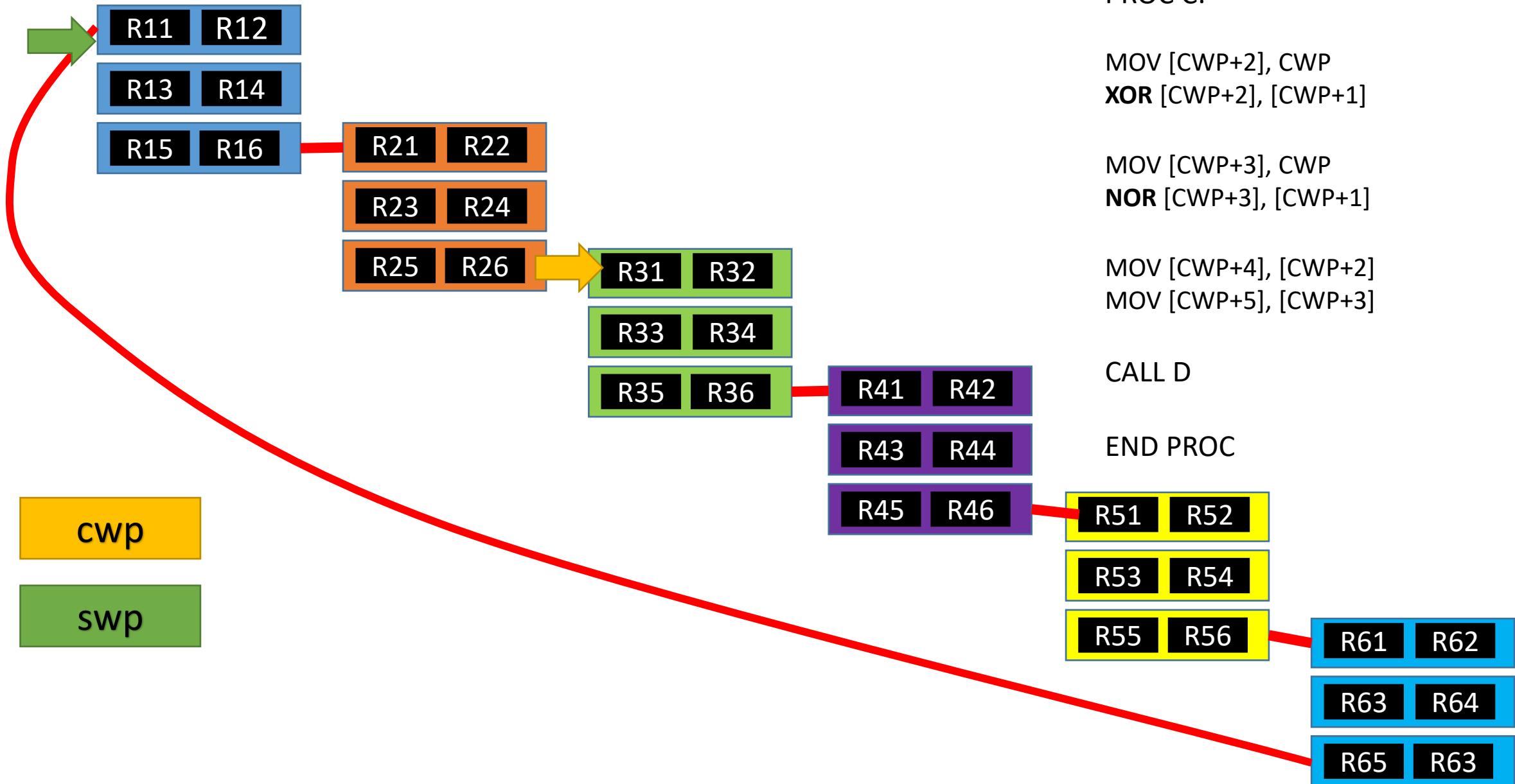
Circular Buffer Organization of Overlapping Windows (Cont.)



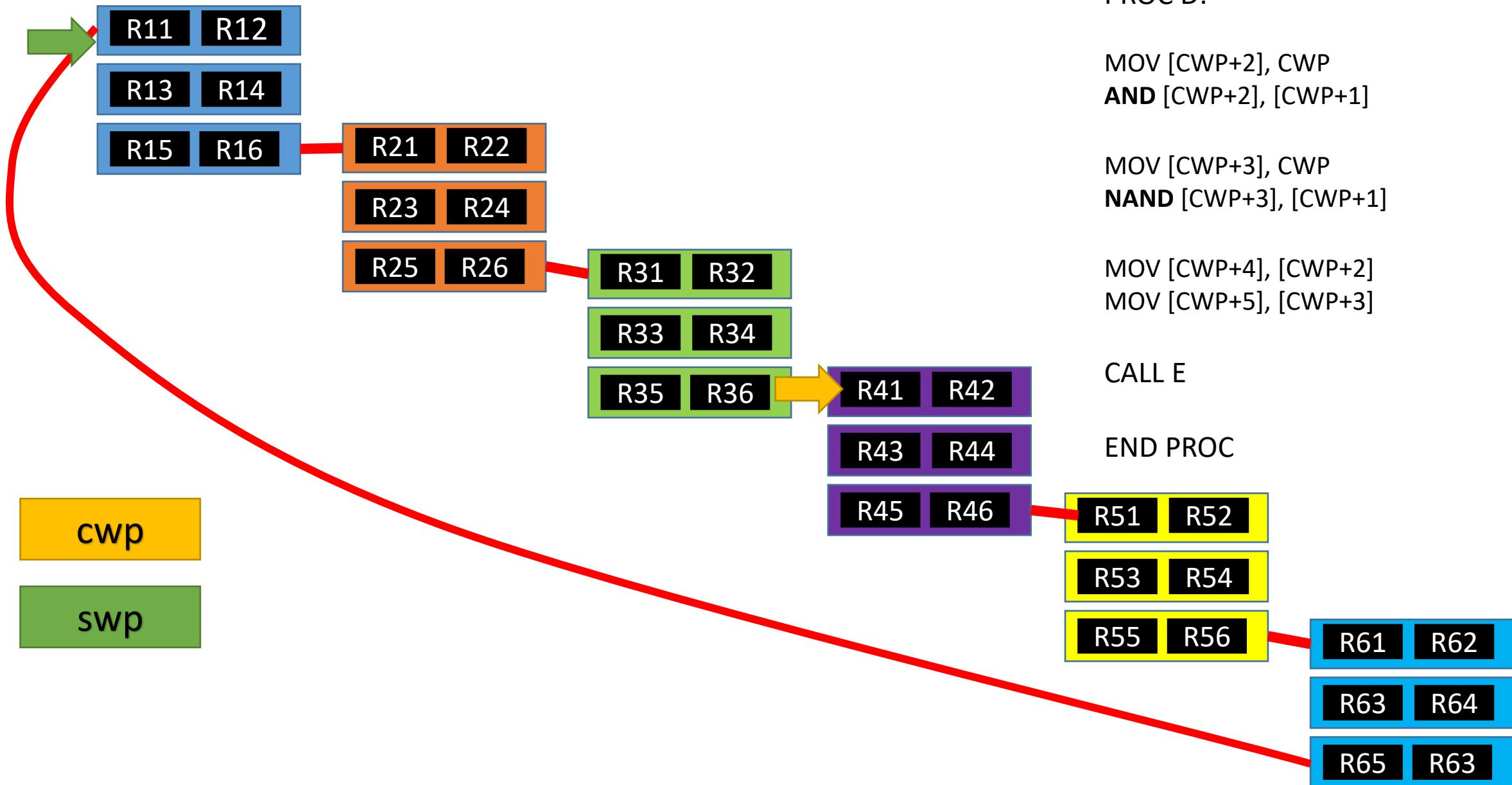
Circular Buffer Organization of Overlapping Windows (Cont.)



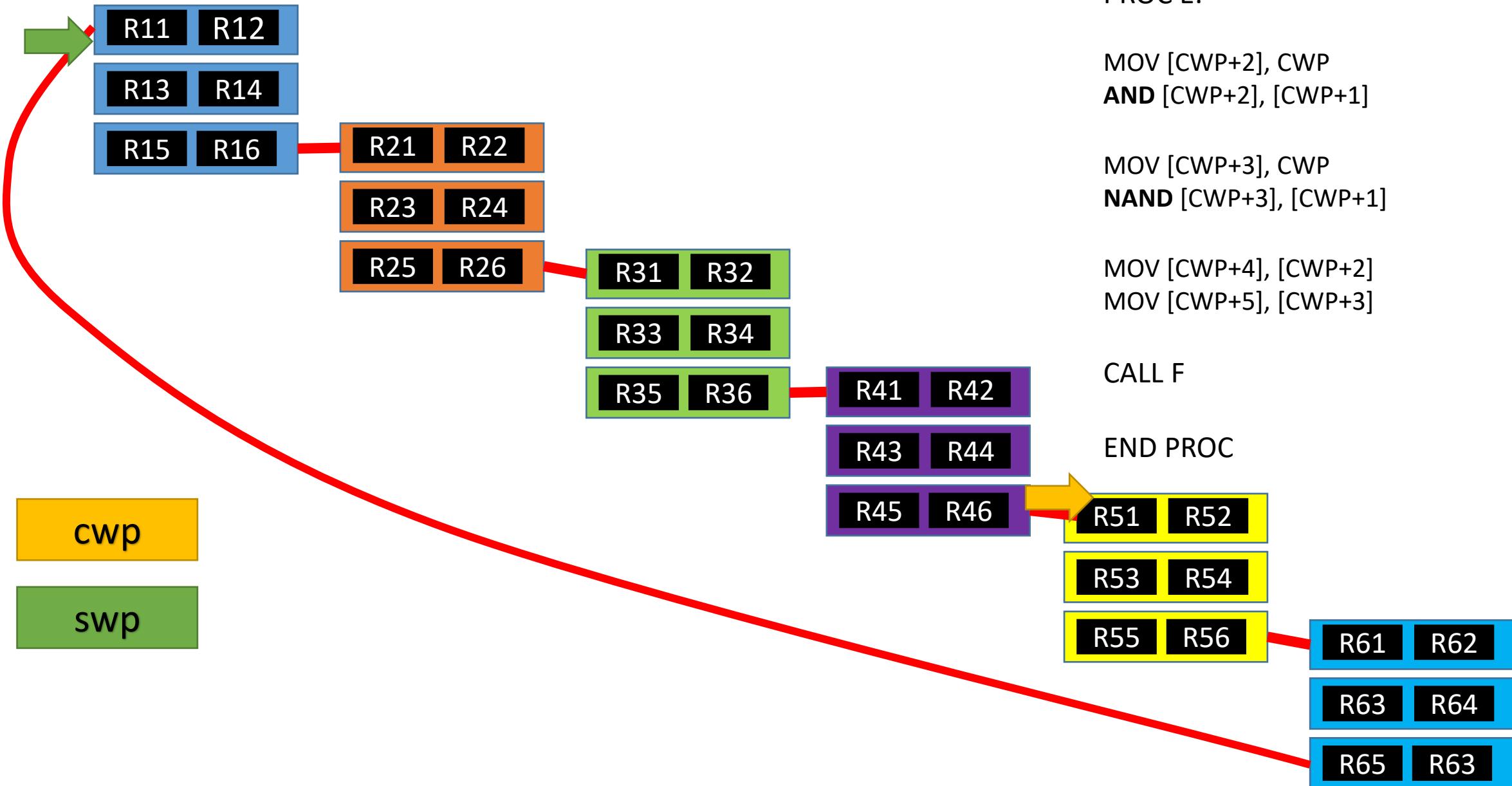
Circular Buffer Organization of Overlapping Windows (Cont.)



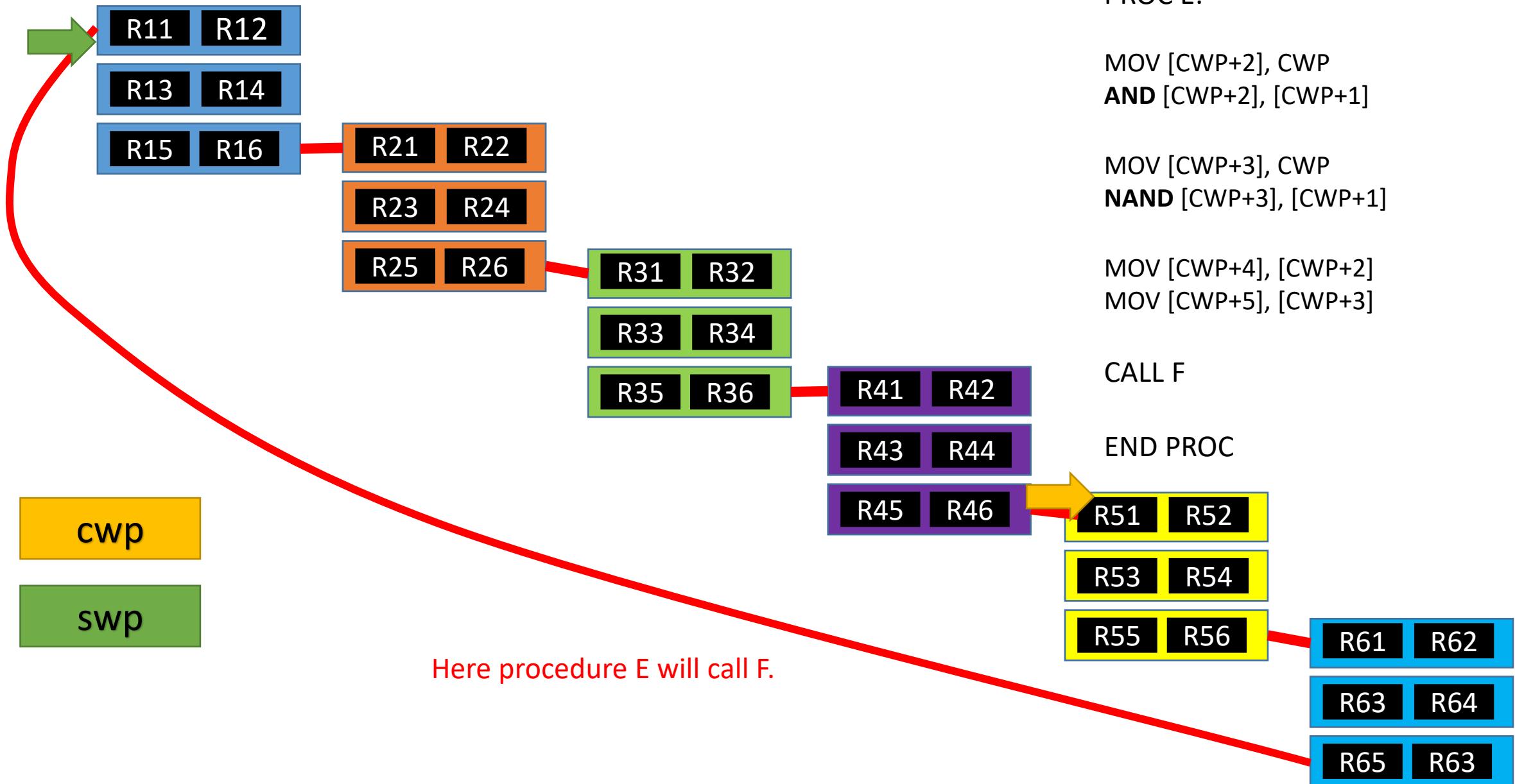
Circular Buffer Organization of Overlapping Windows (Cont.)



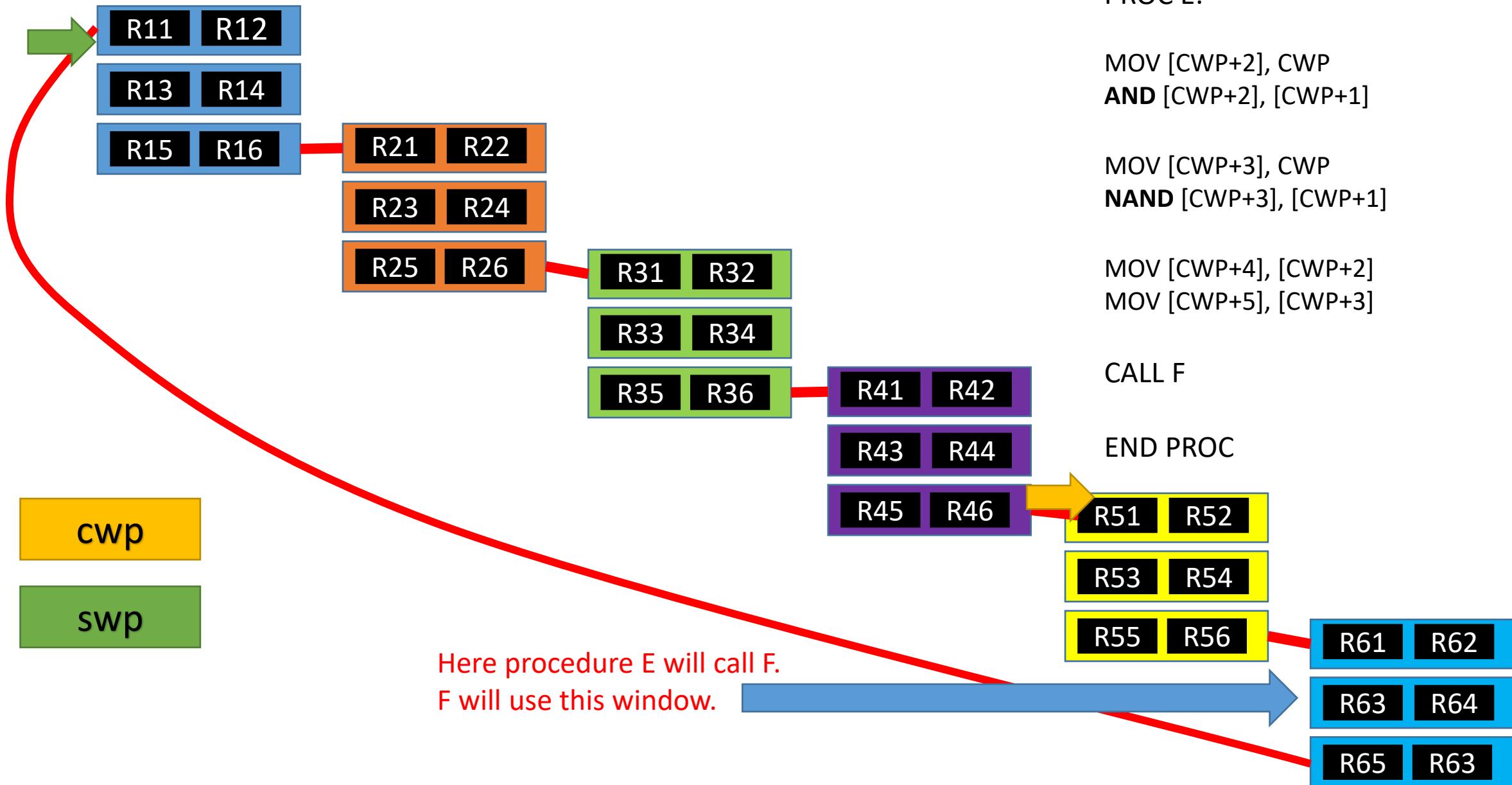
Circular Buffer Organization of Overlapping Windows (Cont.)



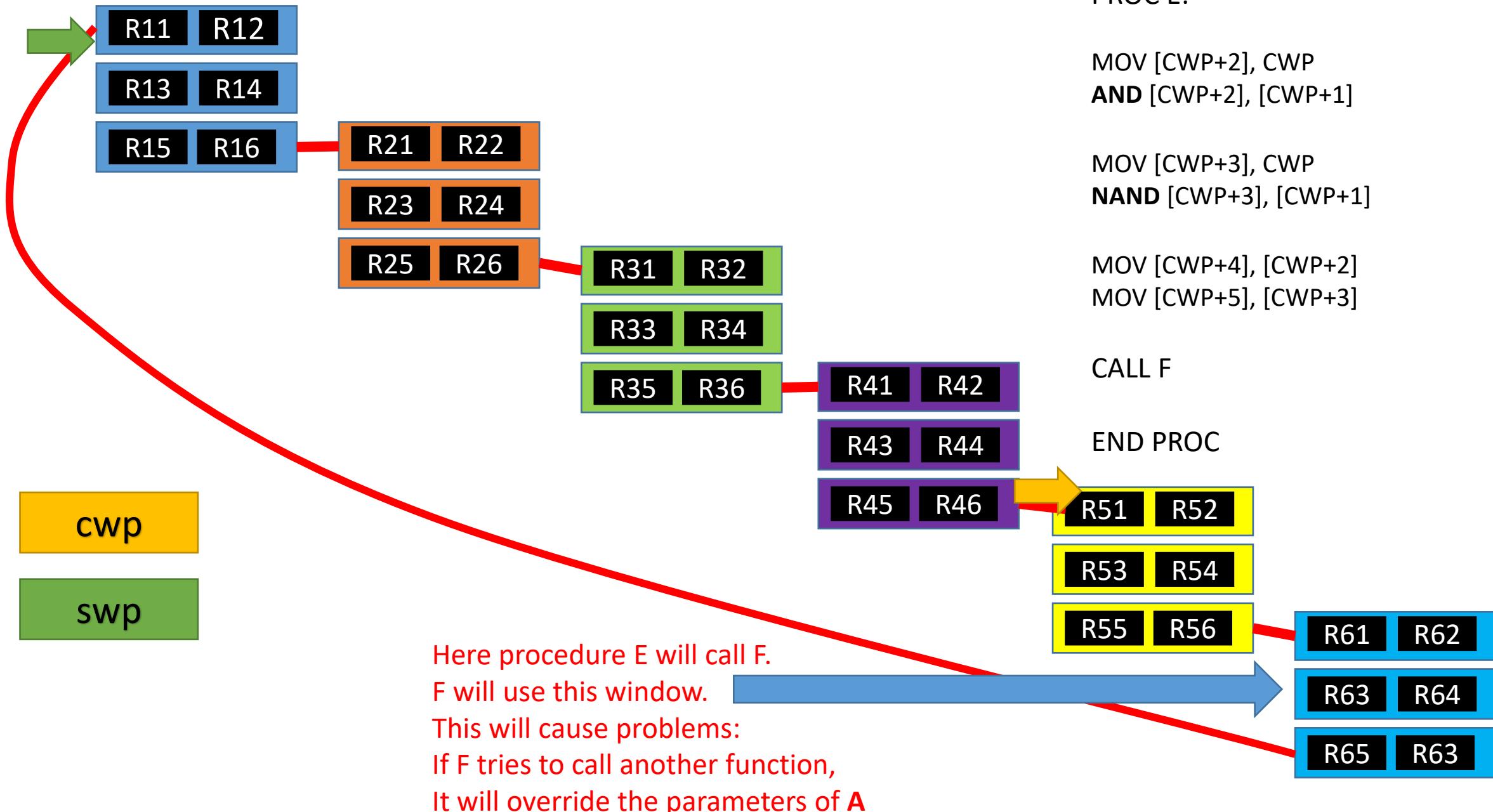
Circular Buffer Organization of Overlapping Windows (Cont.)



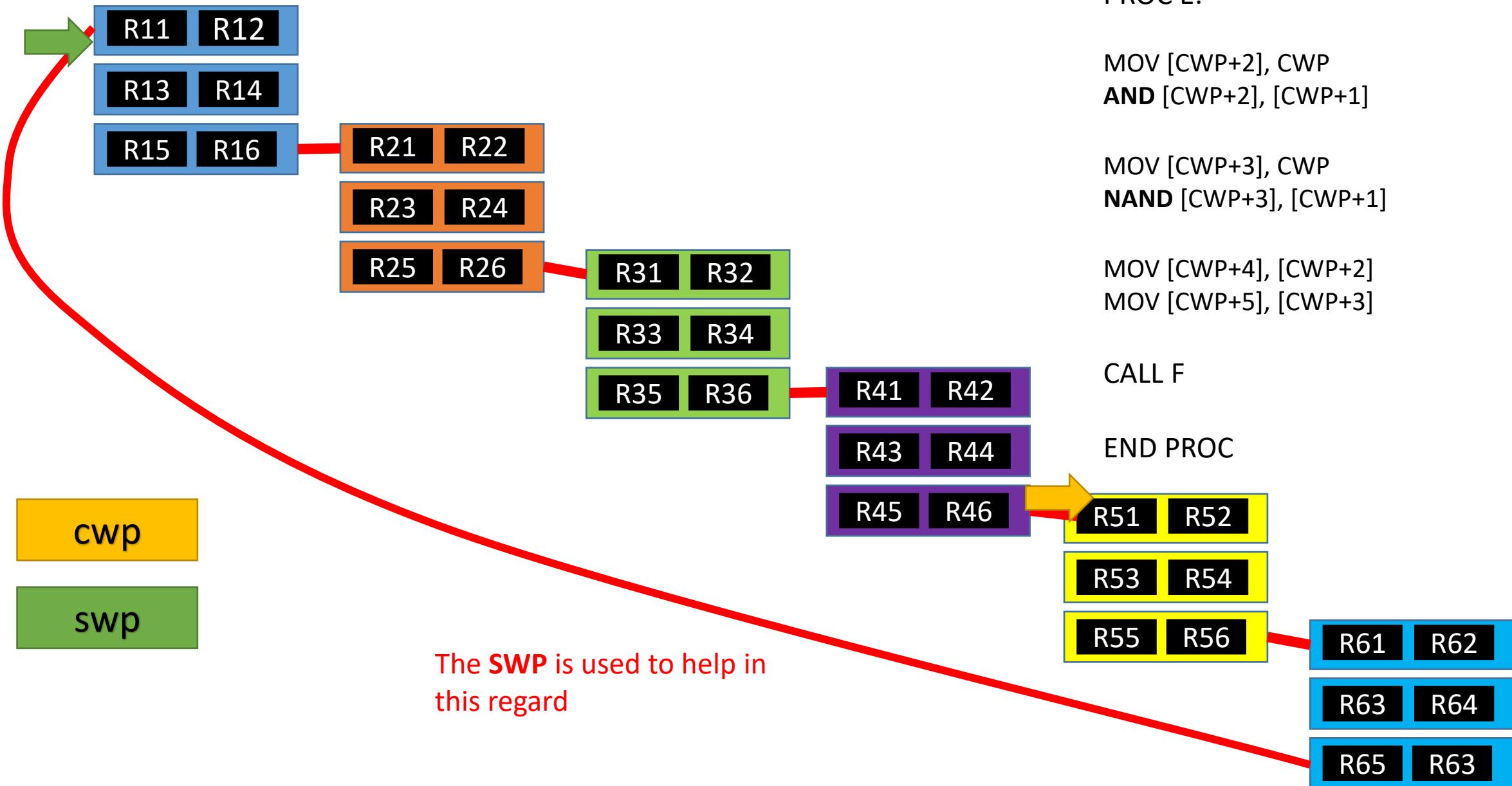
Circular Buffer Organization of Overlapping Windows (Cont.)



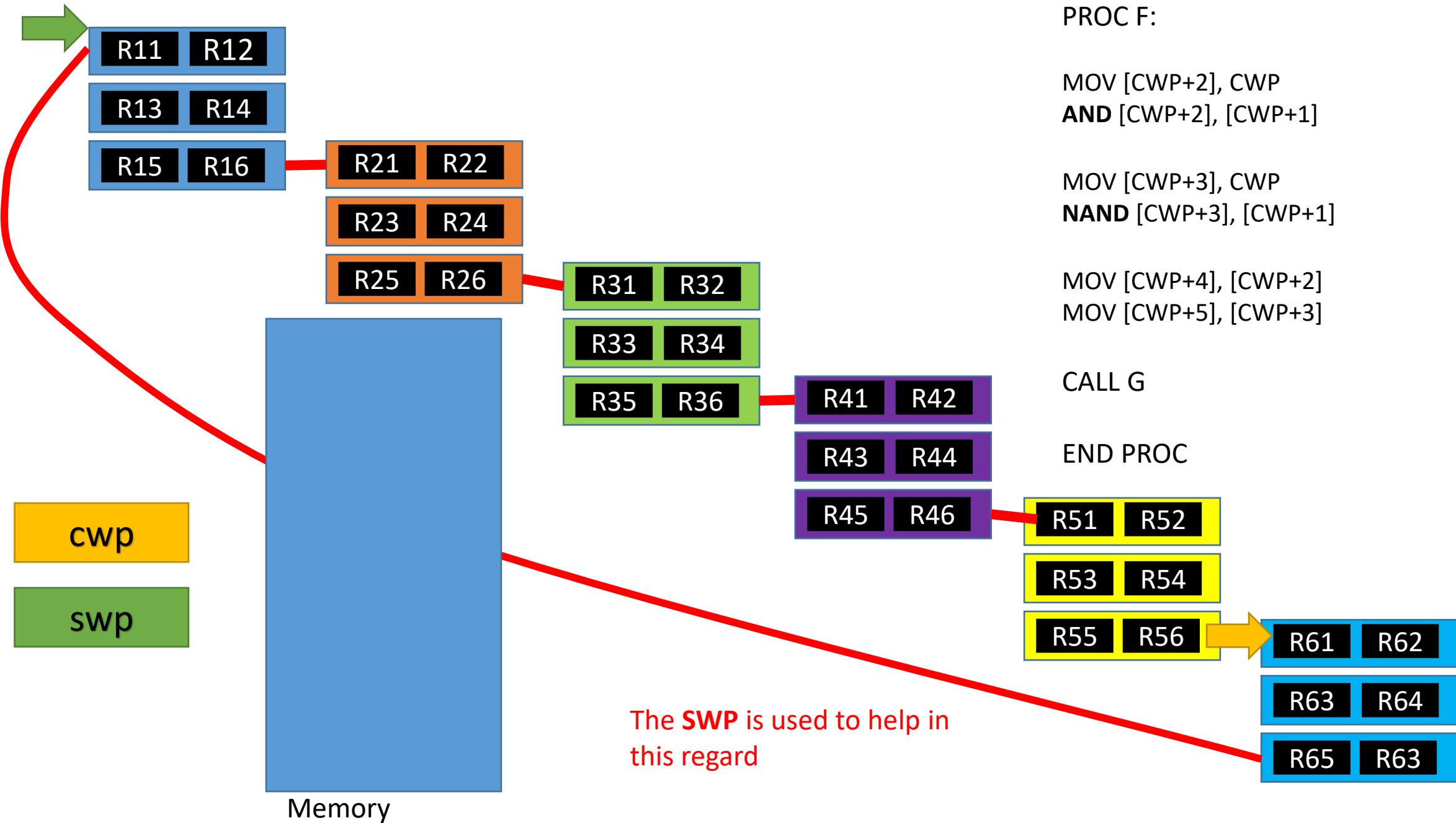
Circular Buffer Organization of Overlapping Windows (Cont.)



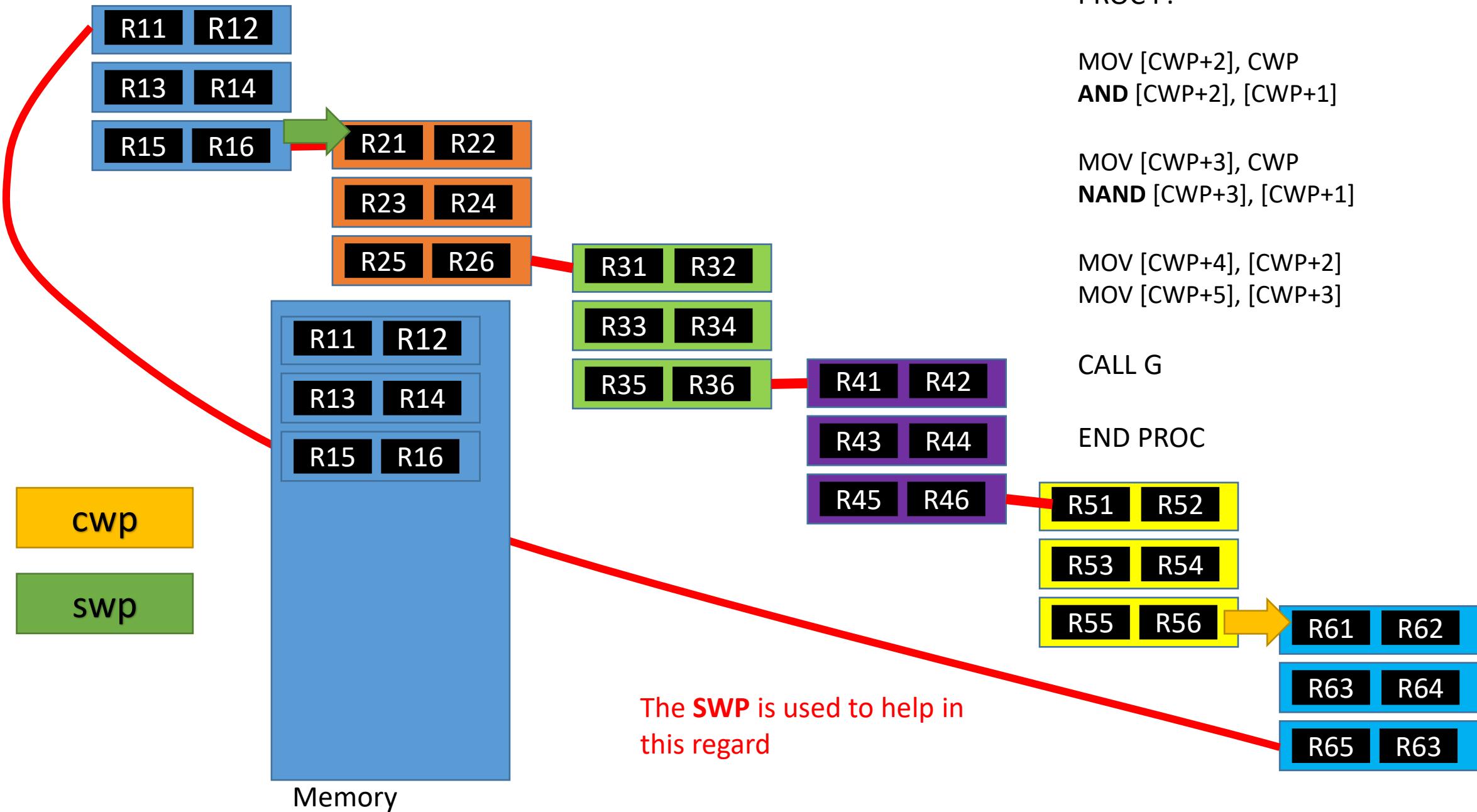
Circular Buffer Organization of Overlapping Windows (Cont.)



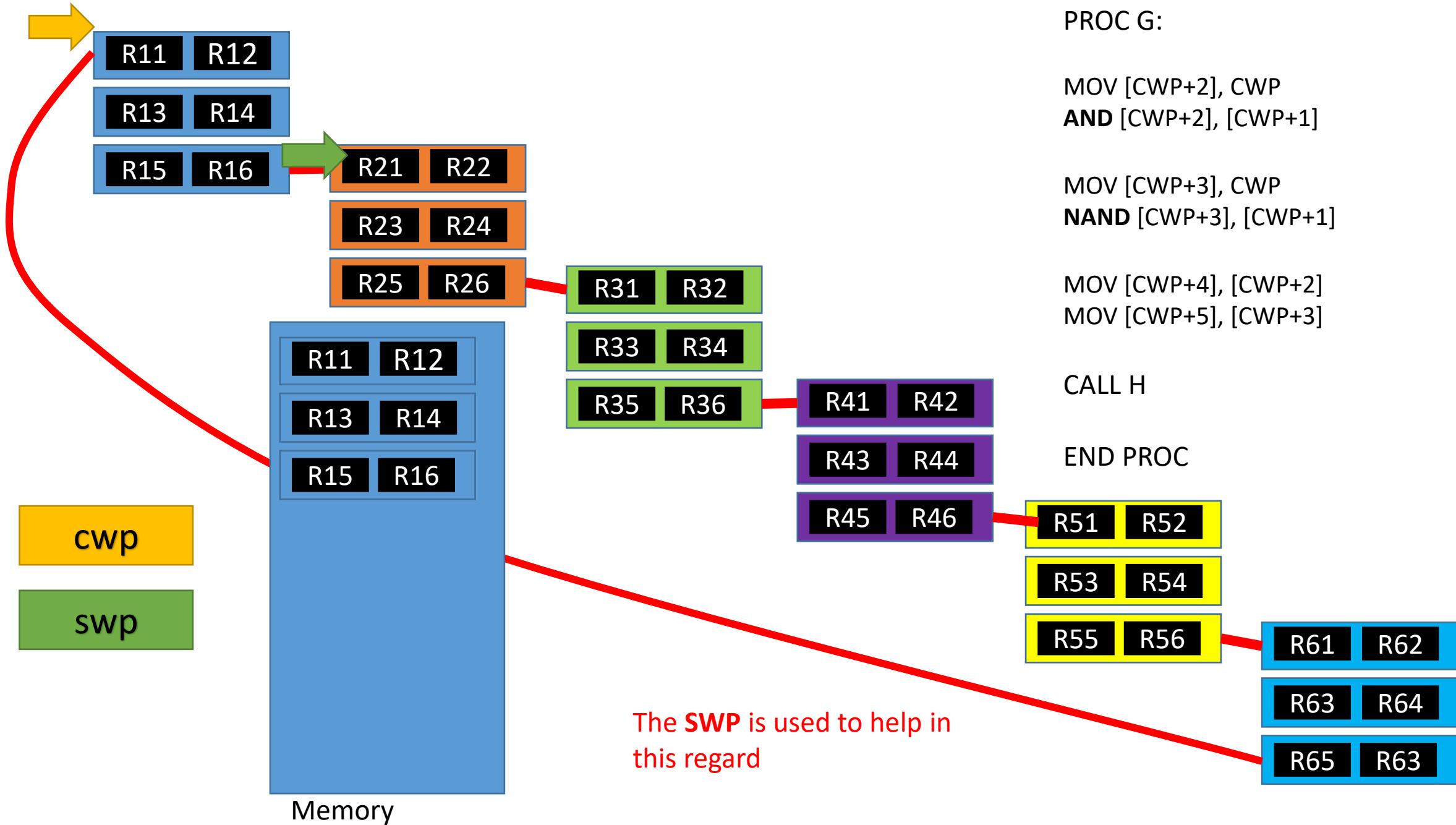
Circular Buffer Organization of Overlapping Windows (Cont.)



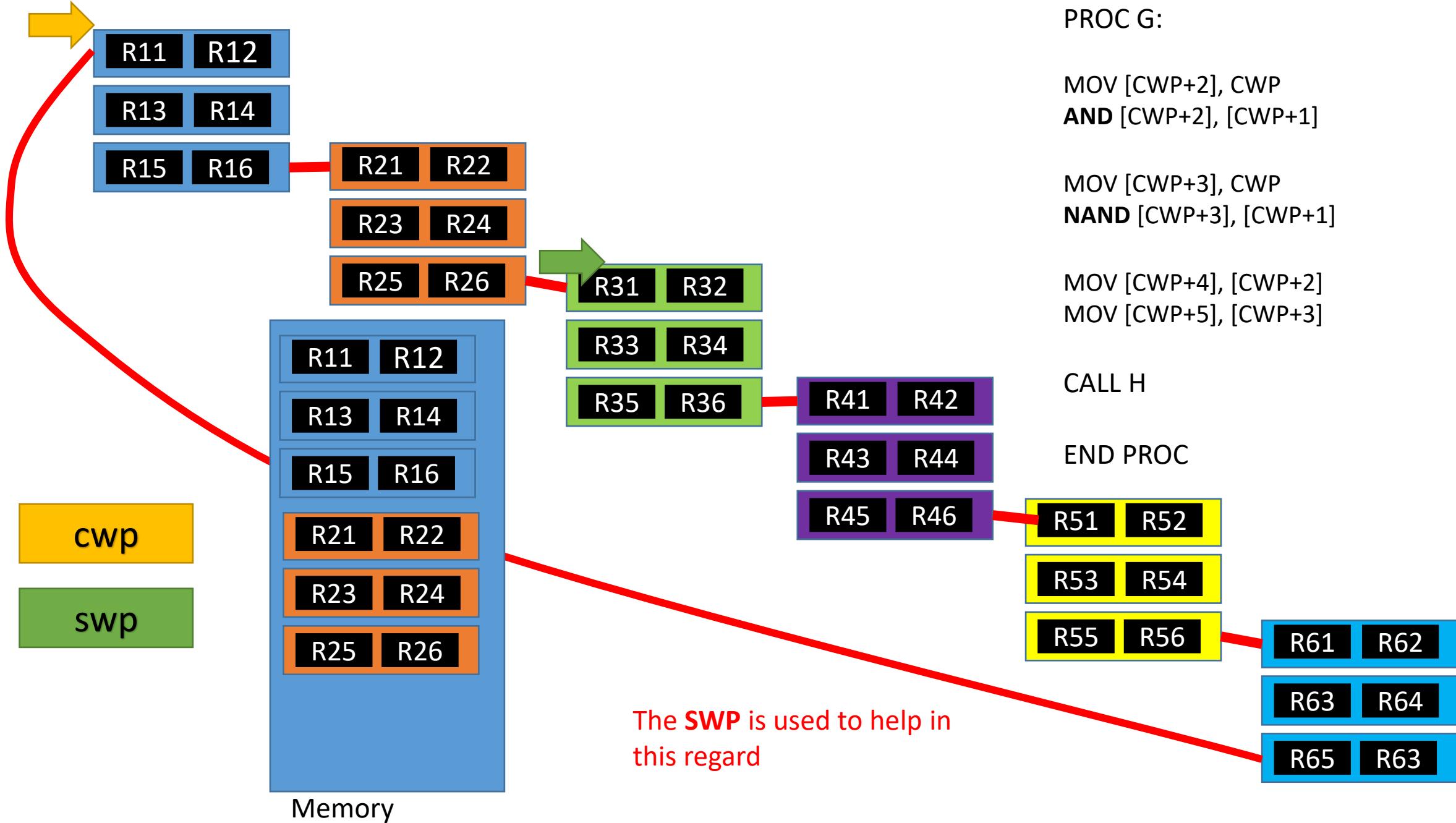
Circular Buffer Organization of Overlapping Windows (Cont.)



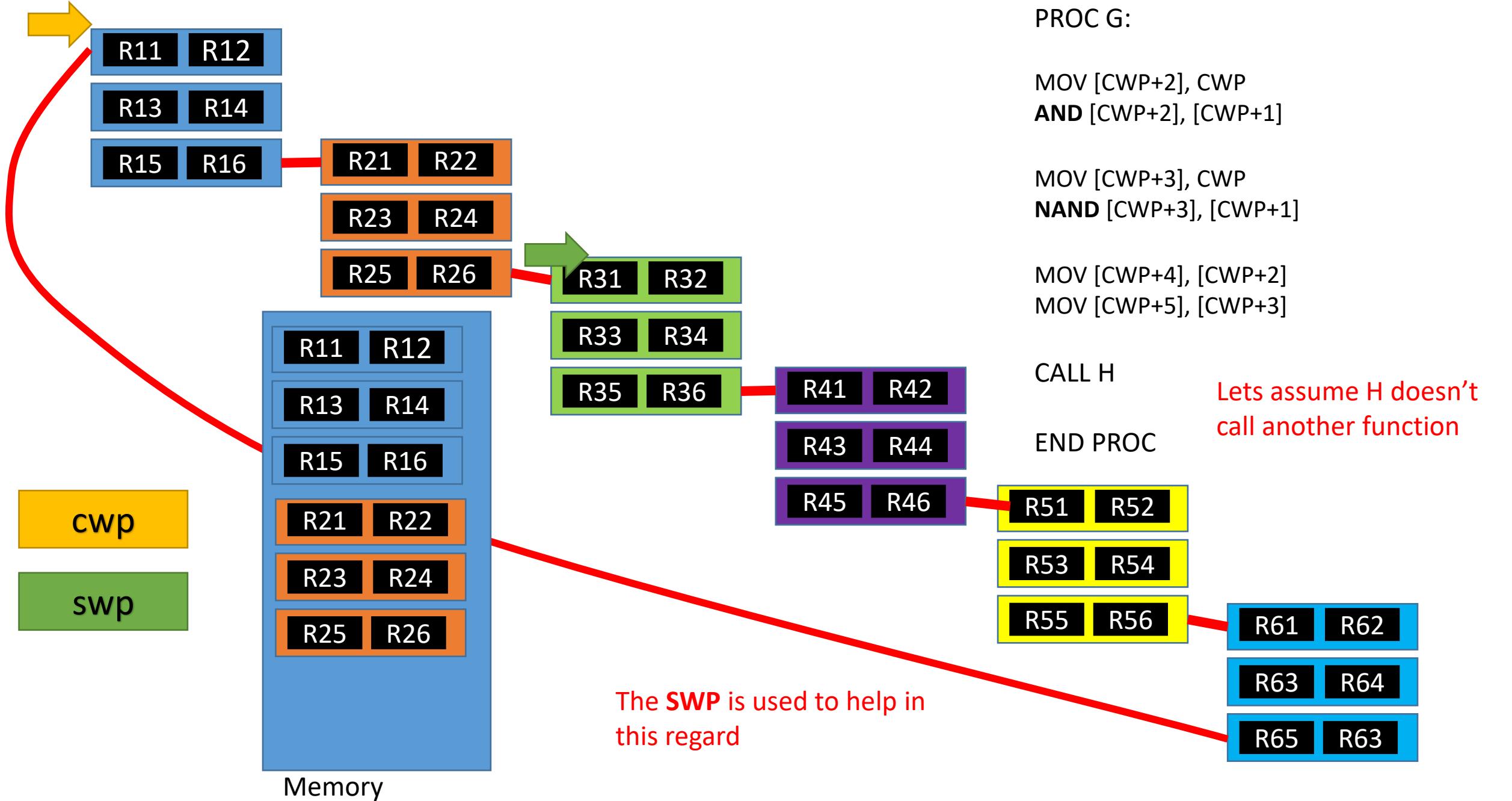
Circular Buffer Organization of Overlapping Windows (Cont.)



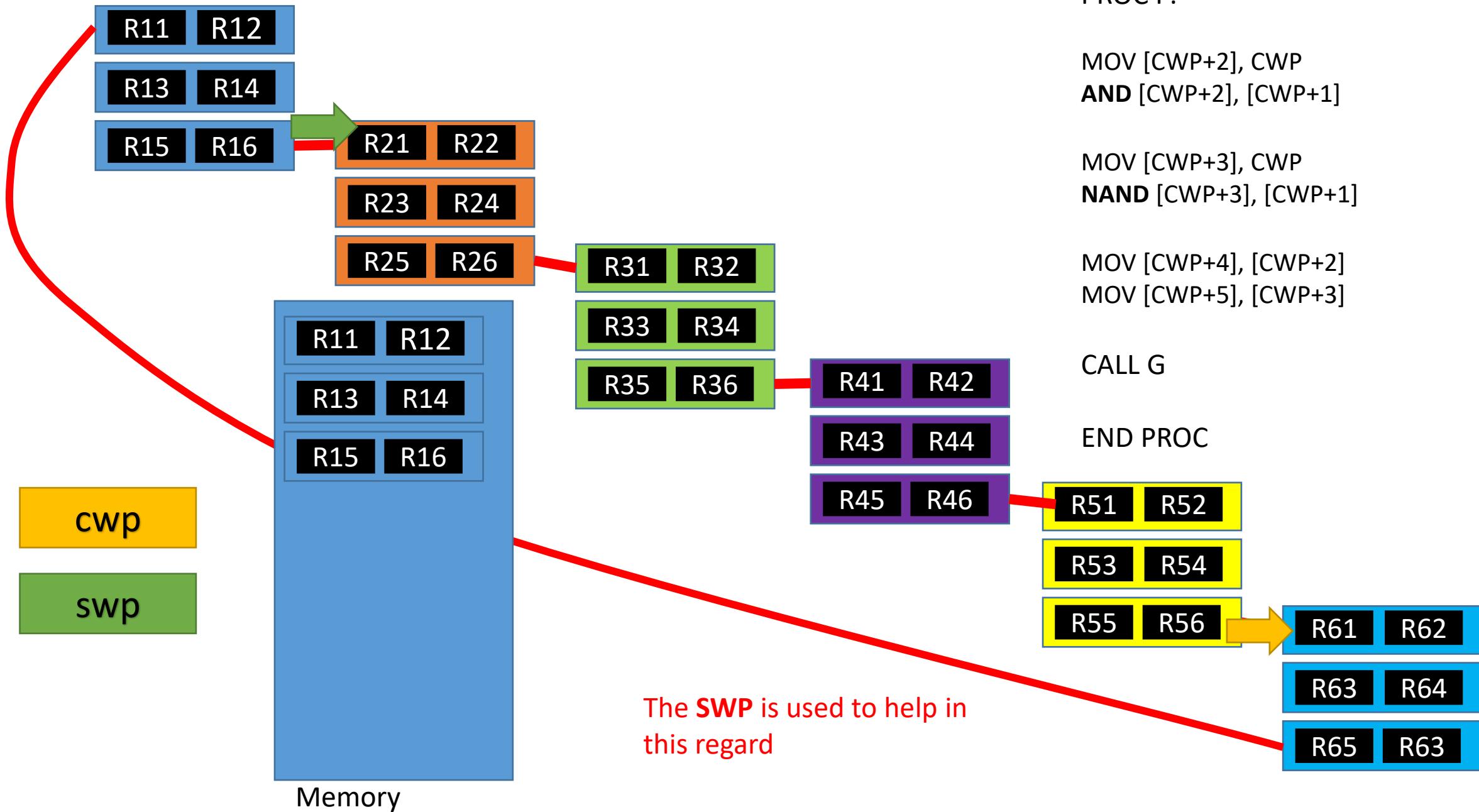
Circular Buffer Organization of Overlapping Windows (Cont.)



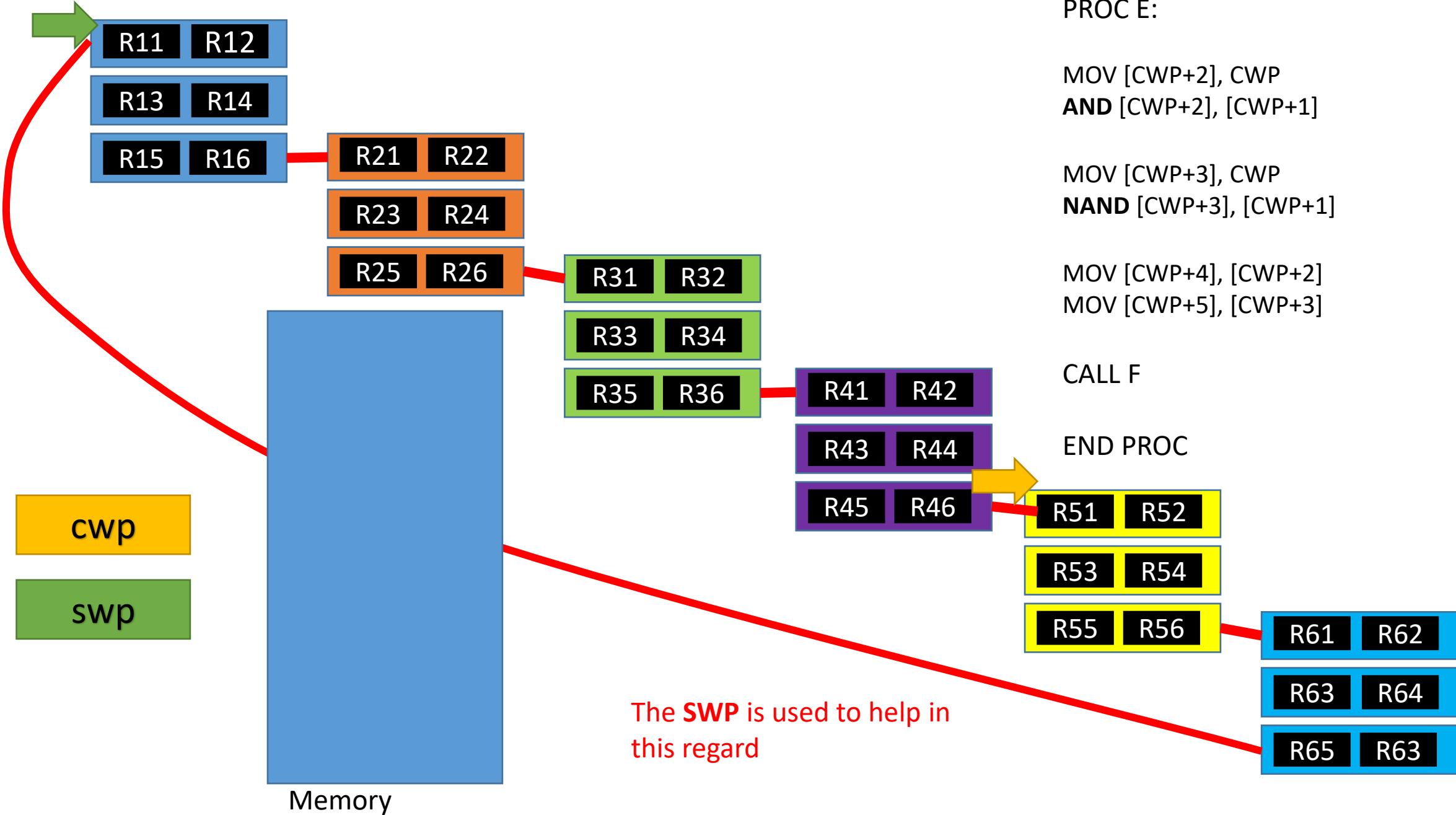
Circular Buffer Organization of Overlapping Windows (Cont.)



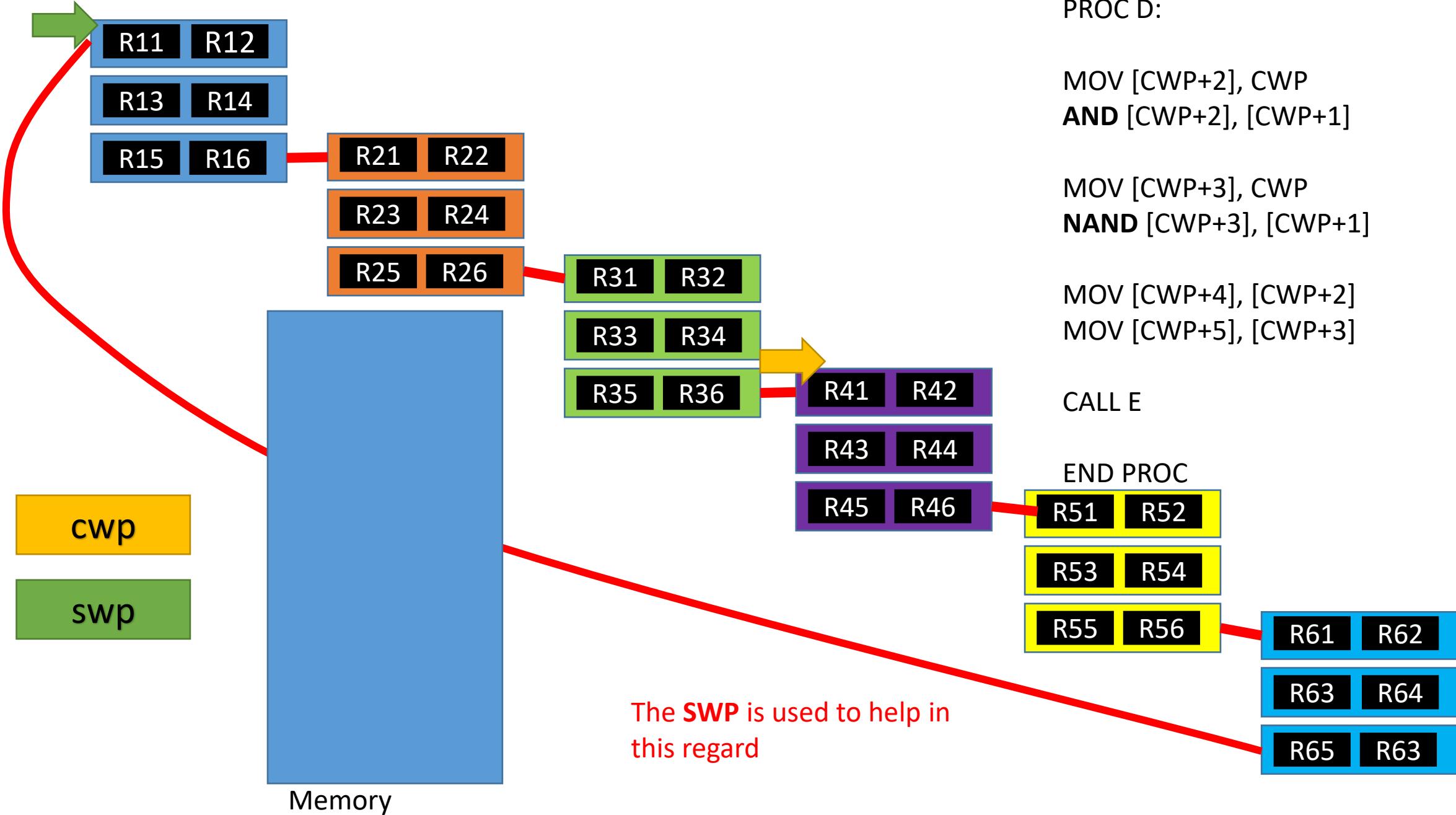
Circular Buffer Organization of Overlapping Windows (Cont.)



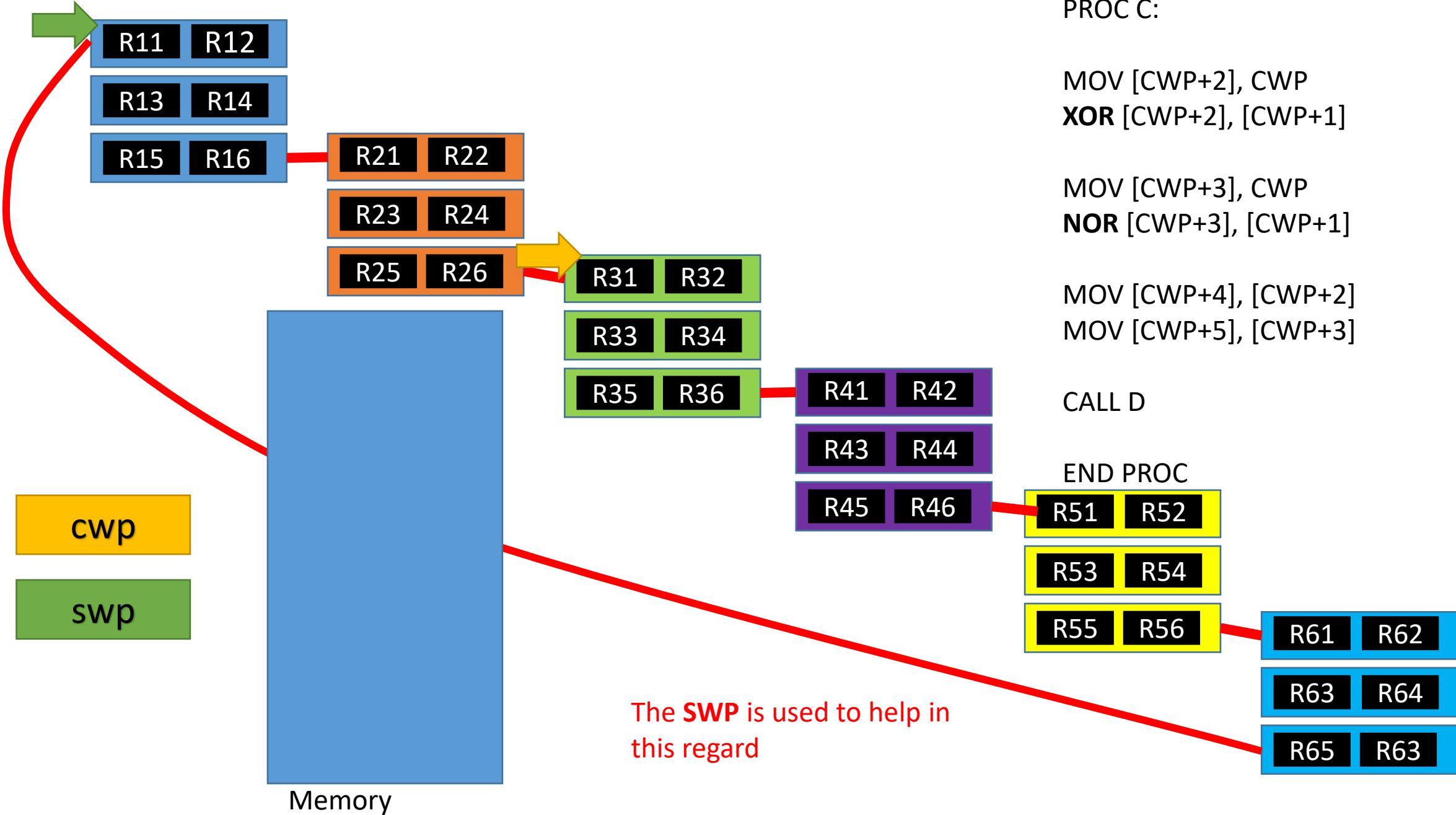
Circular Buffer Organization of Overlapping Windows (Cont.)



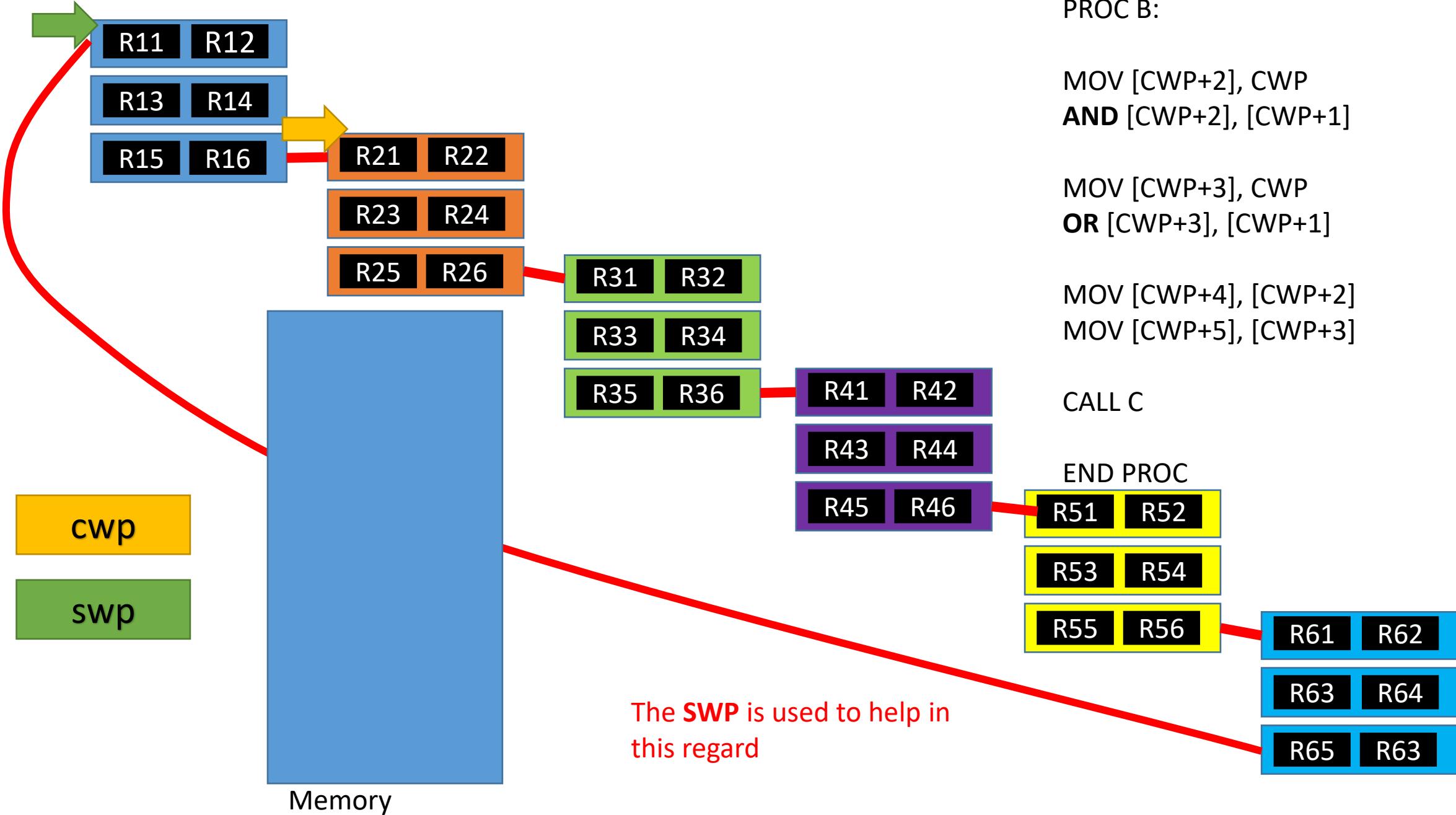
Circular Buffer Organization of Overlapping Windows (Cont.)



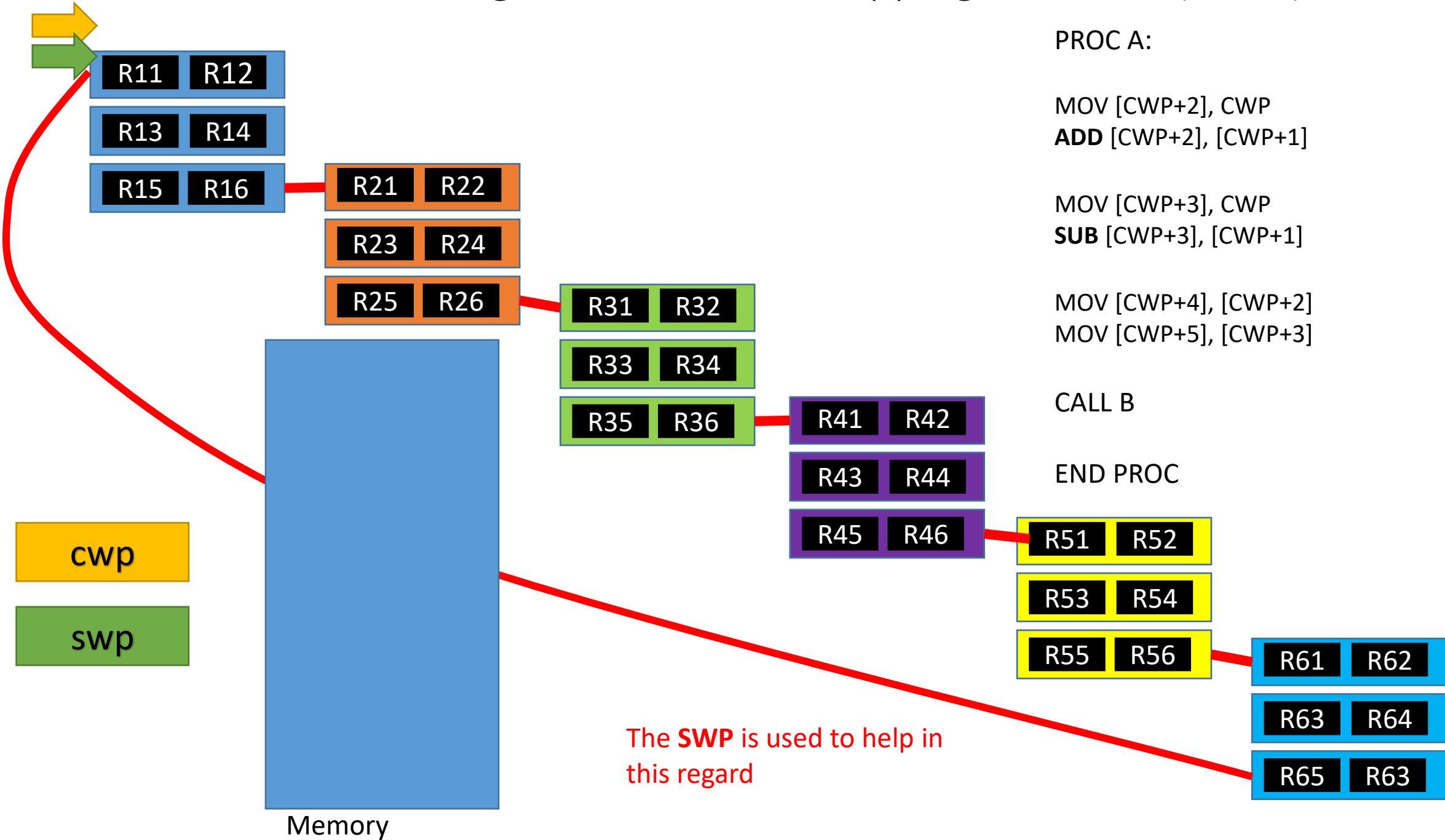
Circular Buffer Organization of Overlapping Windows (Cont.)



Circular Buffer Organization of Overlapping Windows (Cont.)



Circular Buffer Organization of Overlapping Windows (Cont.)



RISC Pipelining

RISC processors have a shorter pipeline. This is because:

1. The **instructions** are **simple**.
 - a) Each instruction performs a simple task.
 - b) Load, Store, Operate : Separately used instructions.
2. The arrangement of the **hardware** components is **simple**.

RISC Pipelining (Cont.)

I : Instruction Fetch

E: Execution (Add,Sub)

D : Read/write

Remember:

In RISC processors, after an operation (e.g ADD, SUB), data is written to the registers.

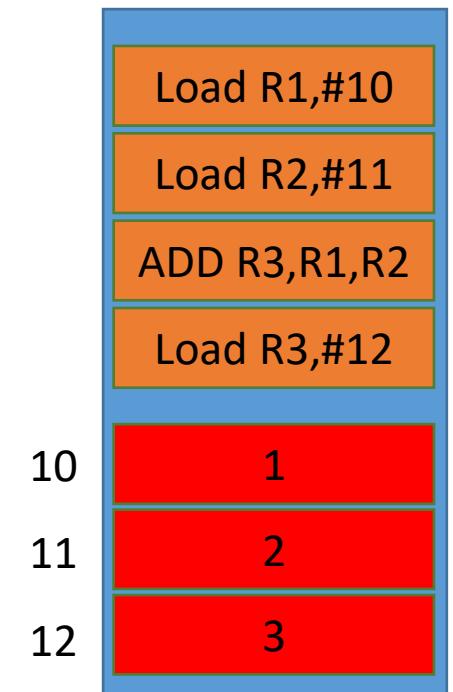
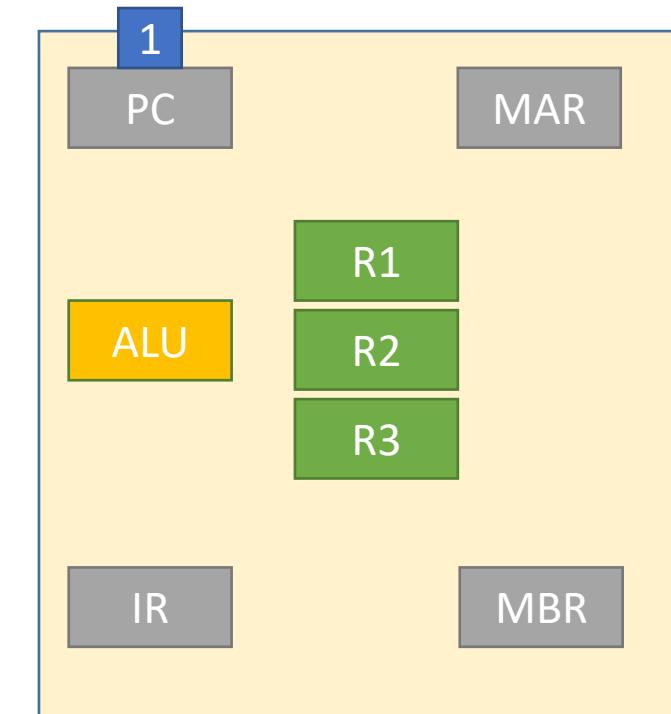
Later on data from the registers are transferred to the memory using another store operation.

RISC Pipelining Issues (Case 1)

| Time Units | 1 | 2 | 3 | 4 | 5 | 6 |
|-------------------|---|---|---|---|---|---|
| Load R1, #10 | I | E | D | | | |
| Load R2, #11 | | I | E | D | | |
| Add R3, R1, R2 | | | I | E | D | |
| Store R3. M | | | | I | E | D |
| | | | | | | |

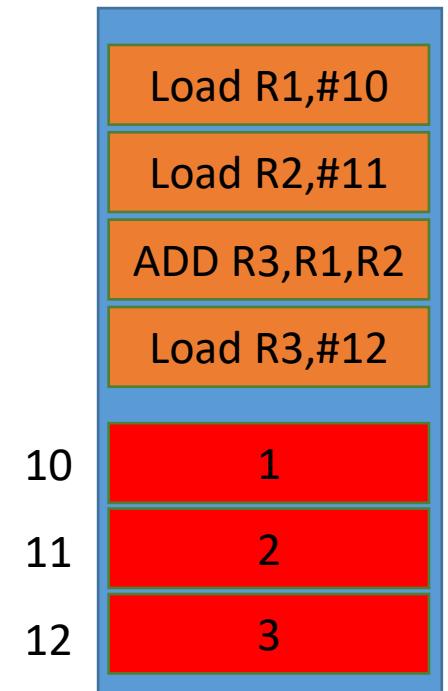
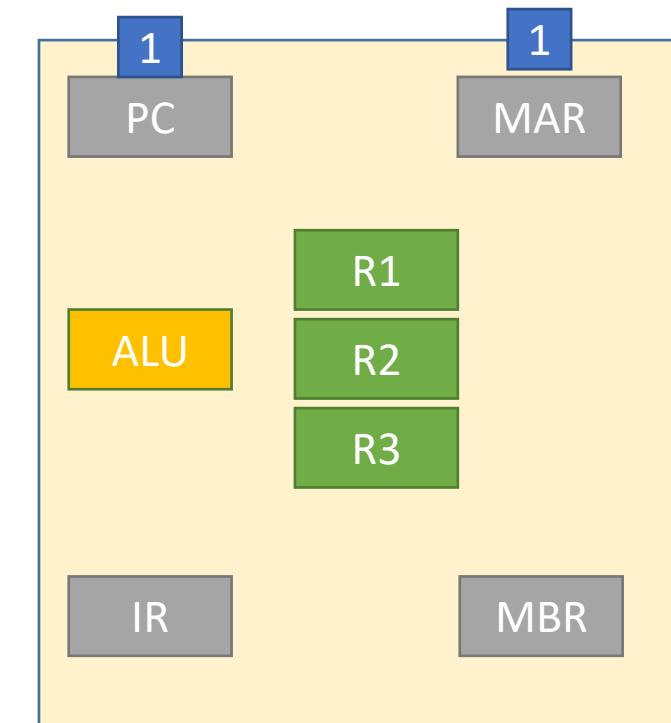
RISC Pipelining Issues (Case 1)

| Time Units | 1 | 2 | 3 | 4 | 5 | 6 |
|-------------------|---|---|---|---|---|---|
| Load R1, #10 | I | E | D | | | |
| Load R2, #11 | | I | E | D | | |
| Add R3, R1, R2 | | | I | E | D | |
| Store R3, M | | | | I | E | D |
| | | | | | | |



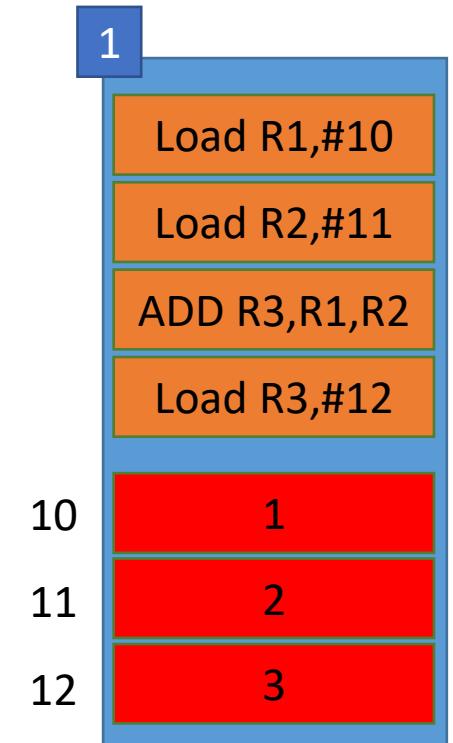
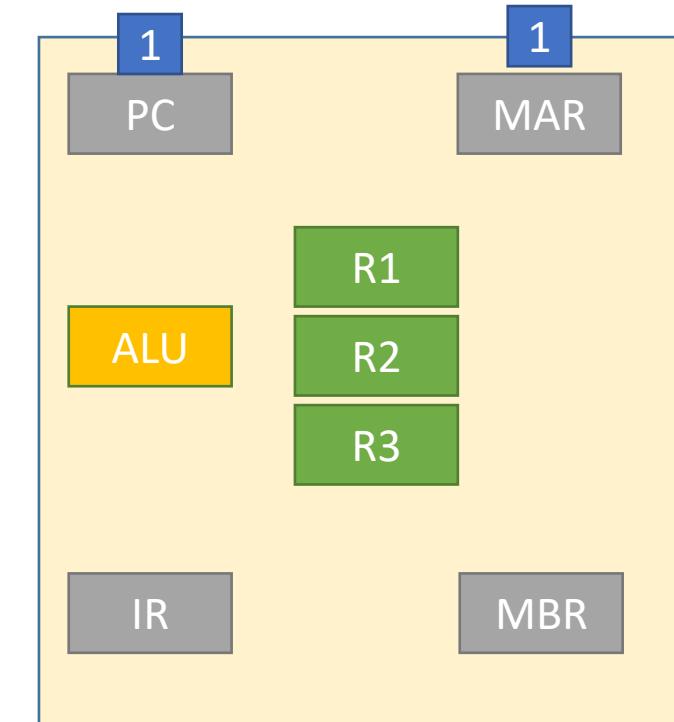
RISC Pipelining Issues (Case 1)

| Time Units | 1 | 2 | 3 | 4 | 5 | 6 |
|-------------------|---|---|---|---|---|---|
| Load R1, #10 | I | E | D | | | |
| Load R2, #11 | | I | E | D | | |
| Add R3, R1, R2 | | | I | E | D | |
| Store R3, M | | | | I | E | D |
| | | | | | | |



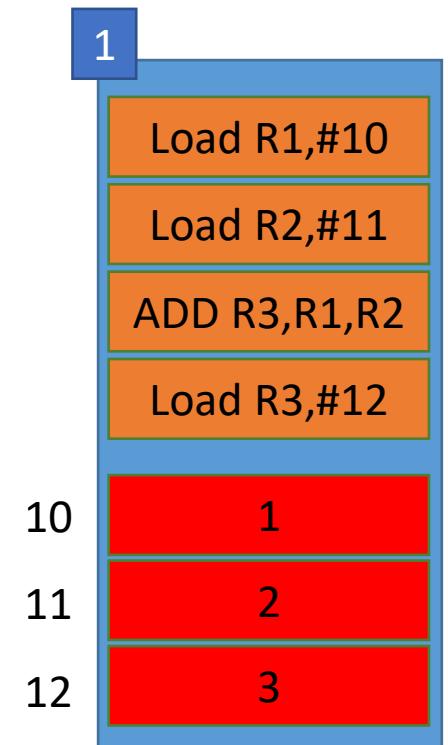
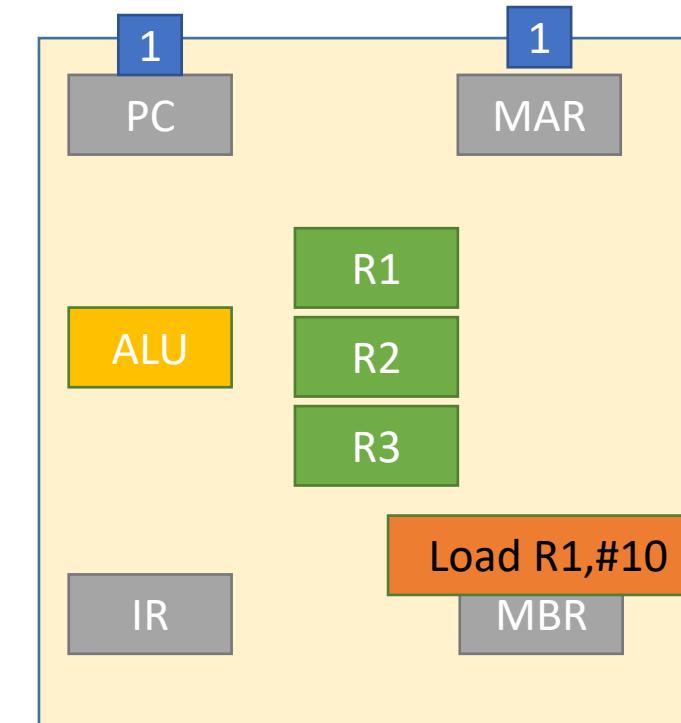
RISC Pipelining Issues (Case 1)

| Time Units | 1 | 2 | 3 | 4 | 5 | 6 |
|-------------------|---|---|---|---|---|---|
| Load R1, #10 | I | E | D | | | |
| Load R2, #11 | | I | E | D | | |
| Add R3, R1, R2 | | | I | E | D | |
| Store R3, M | | | | I | E | D |
| | | | | | | |



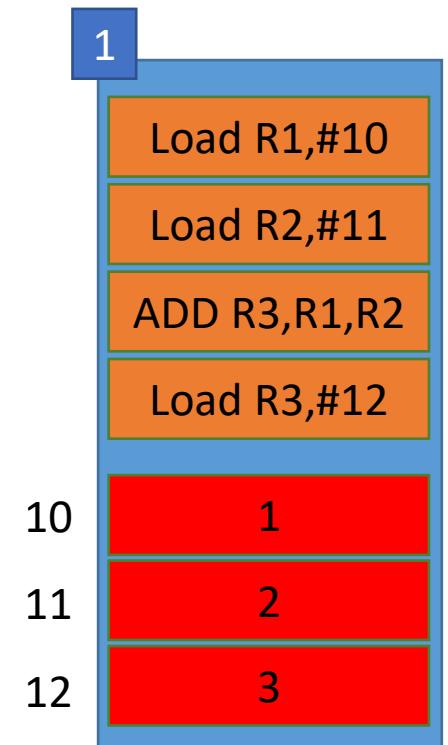
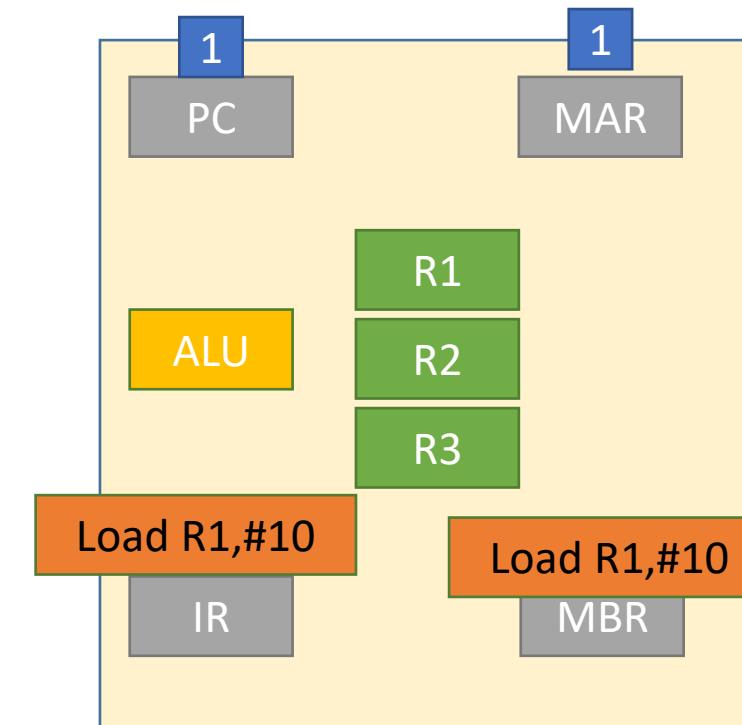
RISC Pipelining Issues (Case 1)

| Time Units | 1 | 2 | 3 | 4 | 5 | 6 |
|-------------------|---|---|---|---|---|---|
| Load R1, #10 | I | E | D | | | |
| Load R2, #11 | | I | E | D | | |
| Add R3, R1, R2 | | | I | E | D | |
| Store R3, M | | | | I | E | D |
| | | | | | | |



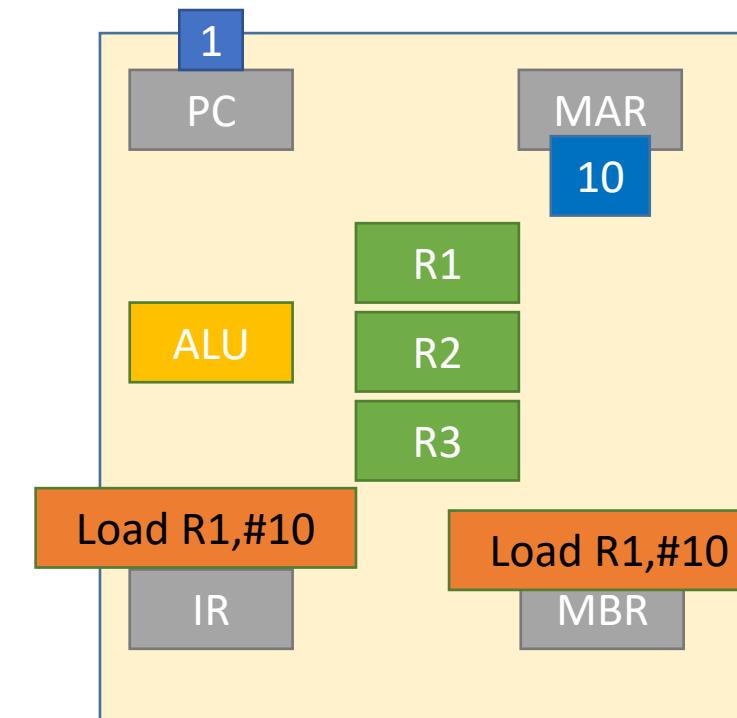
RISC Pipelining Issues (Case 1)

| Time Units | 1 | 2 | 3 | 4 | 5 | 6 |
|-------------------|---|---|---|---|---|---|
| Load R1, #10 | I | E | D | | | |
| Load R2, #11 | | I | E | D | | |
| Add R3, R1, R2 | | | I | E | D | |
| Store R3, M | | | | I | E | D |
| | | | | | | |



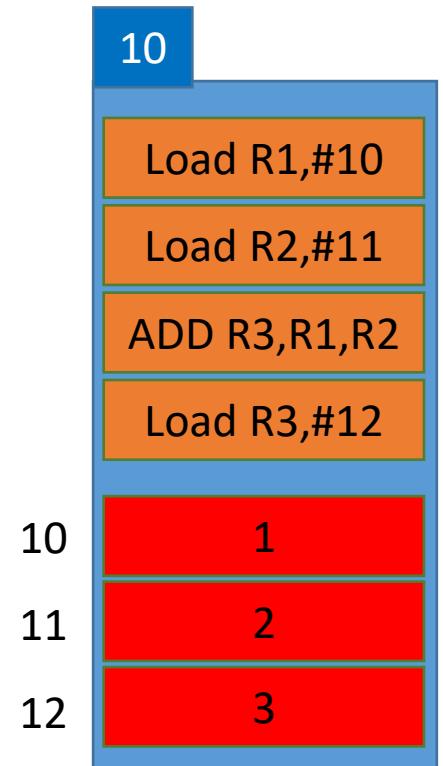
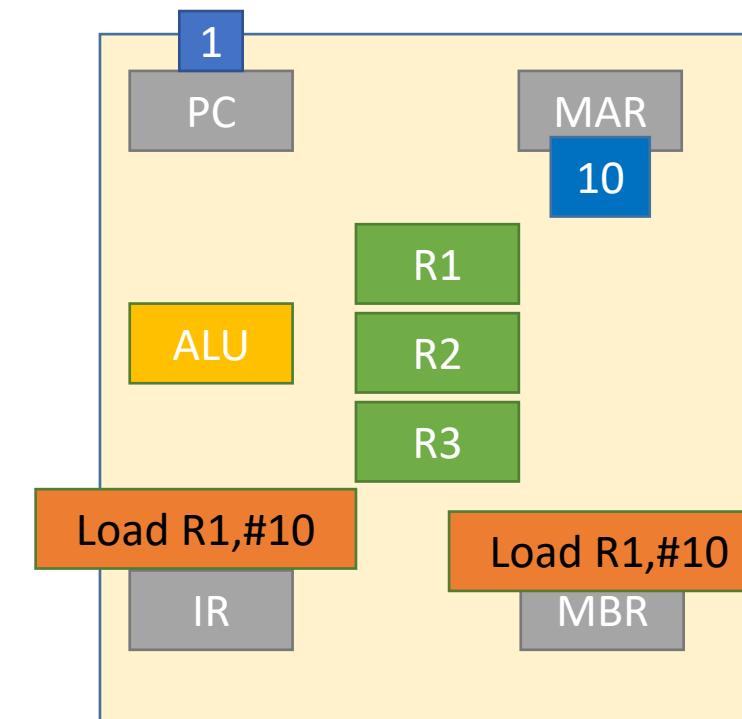
RISC Pipelining Issues (Case 1)

| Time Units | 1 | 2 | 3 | 4 | 5 | 6 |
|-------------------|---|---|---|---|---|---|
| Load R1, #10 | I | E | D | | | |
| Load R2, #11 | | I | E | D | | |
| Add R3, R1, R2 | | | I | E | D | |
| Store R3, M | | | | I | E | D |
| | | | | | | |



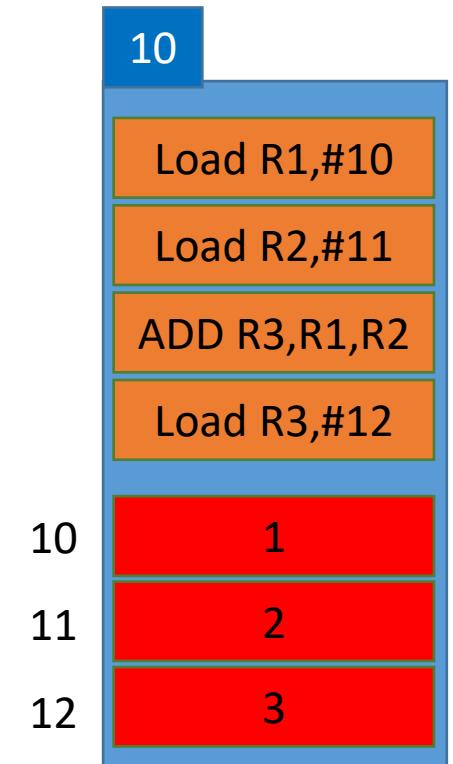
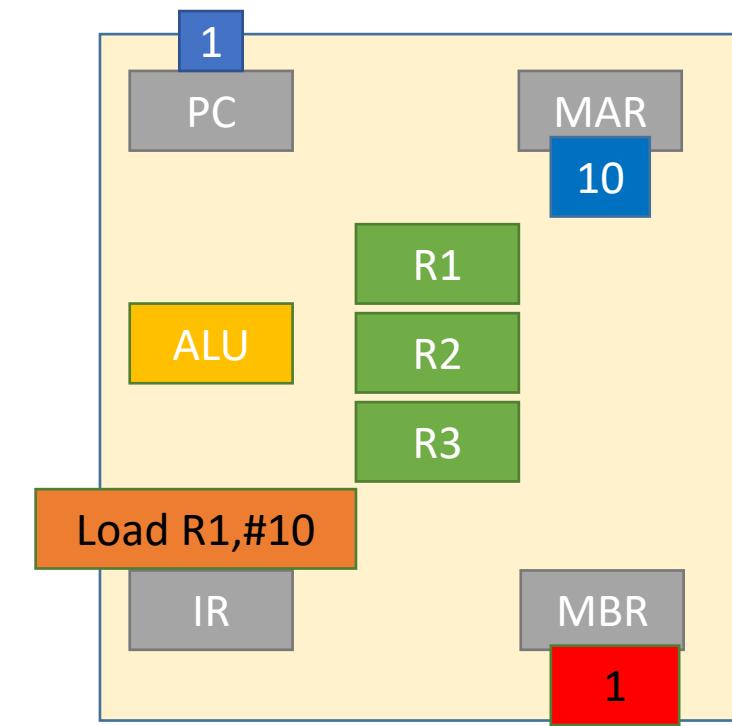
RISC Pipelining Issues (Case 1)

| Time Units | 1 | 2 | 3 | 4 | 5 | 6 |
|-------------------|---|---|---|---|---|---|
| Load R1, #10 | I | E | D | | | |
| Load R2, #11 | | I | E | D | | |
| Add R3, R1, R2 | | | I | E | D | |
| Store R3, M | | | | I | E | D |
| | | | | | | |



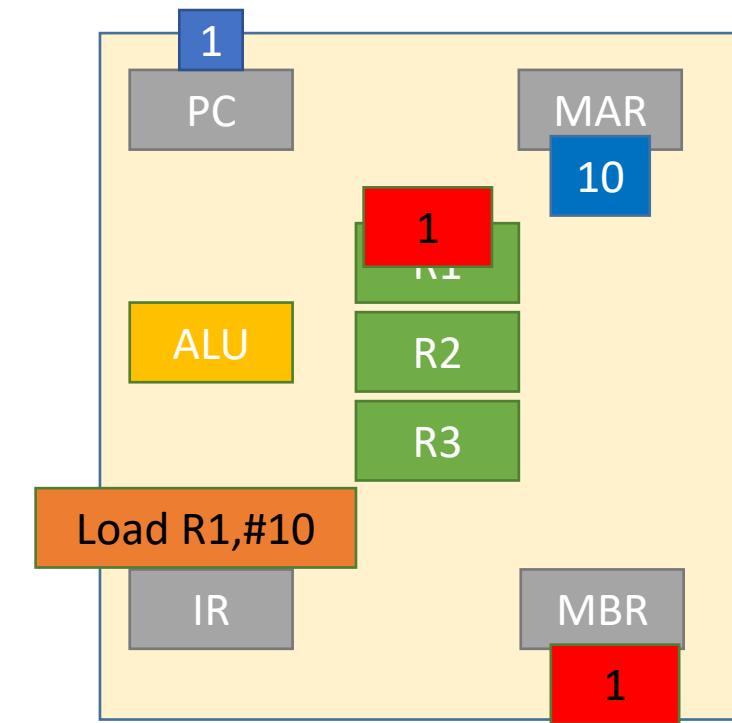
RISC Pipelining Issues (Case 1)

| Time Units | 1 | 2 | 3 | 4 | 5 | 6 |
|-------------------|---|---|---|---|---|---|
| Load R1, #10 | I | E | D | | | |
| Load R2, #11 | | I | E | D | | |
| Add R3, R1, R2 | | | I | E | D | |
| Store R3, M | | | | I | E | D |
| | | | | | | |



RISC Pipelining Issues (Case 1)

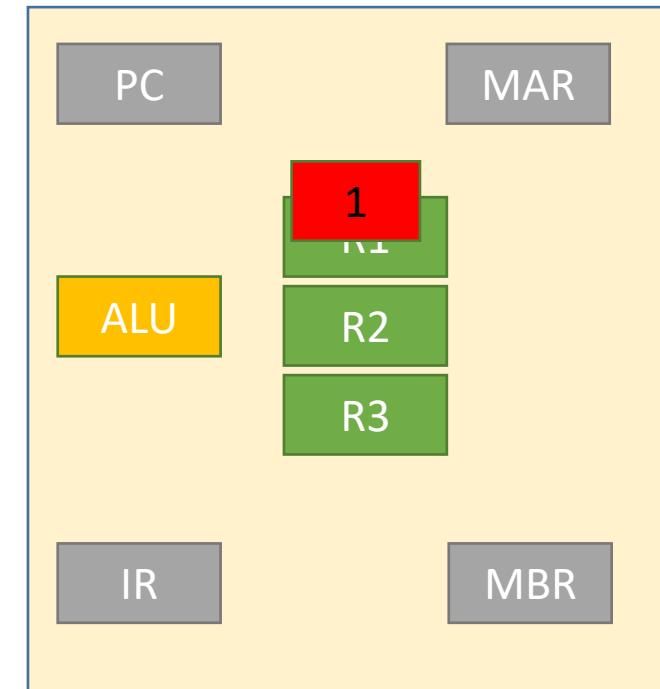
| Time Units | 1 | 2 | 3 | 4 | 5 | 6 |
|-------------------|---|---|---|---|---|---|
| Load R1, #10 | I | E | D | | | |
| Load R2, #11 | | I | E | D | | |
| Add R3, R1, R2 | | | I | E | D | |
| Store R3, M | | | | I | E | D |
| | | | | | | |



RISC Pipelining Issues (Case 1)

| Time Units | 1 | 2 | 3 | 4 | 5 | 6 |
|-------------------|---|---|---|---|---|---|
| Load R1, #10 | I | E | D | | | |
| Load R2, #11 | | I | E | D | | |
| Add R3, R1, R2 | | | I | E | D | |
| Store R3, M | | | | I | E | D |
| | | | | | | |

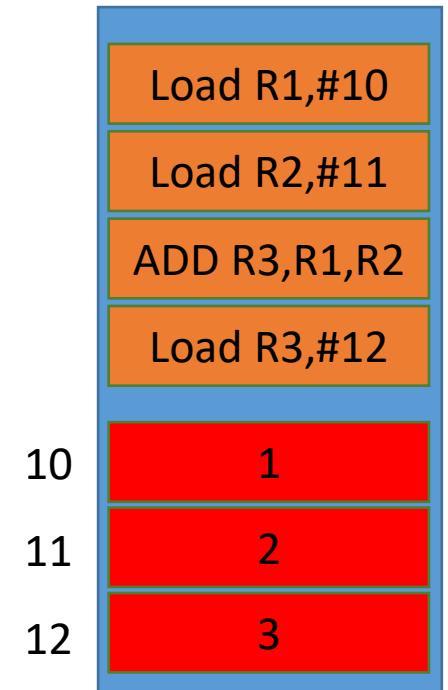
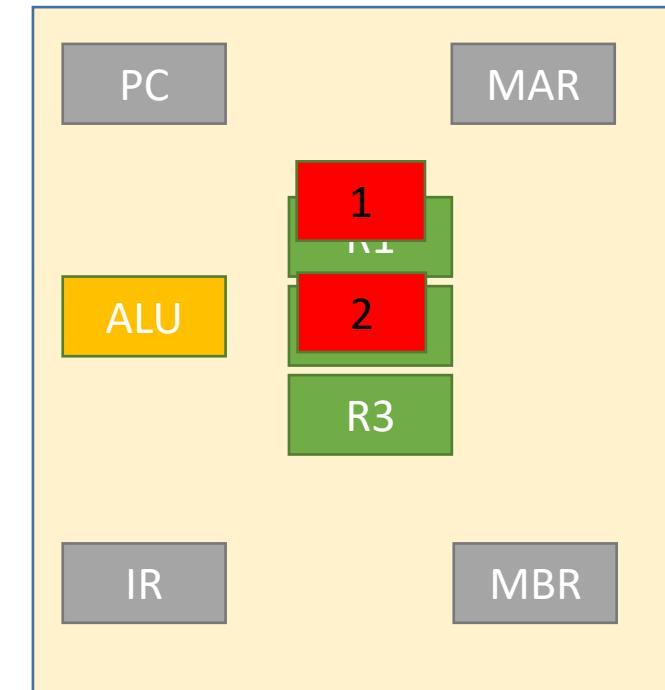
Similarly, Load R2, #11 will be executed



RISC Pipelining Issues (Case 1)

| Time Units | 1 | 2 | 3 | 4 | 5 | 6 |
|-------------------|---|---|---|---|---|---|
| Load R1, #10 | I | E | D | | | |
| Load R2, #11 | | I | E | D | | |
| Add R3, R1, R2 | | | I | E | D | |
| Store R3, M | | | | I | E | D |
| | | | | | | |

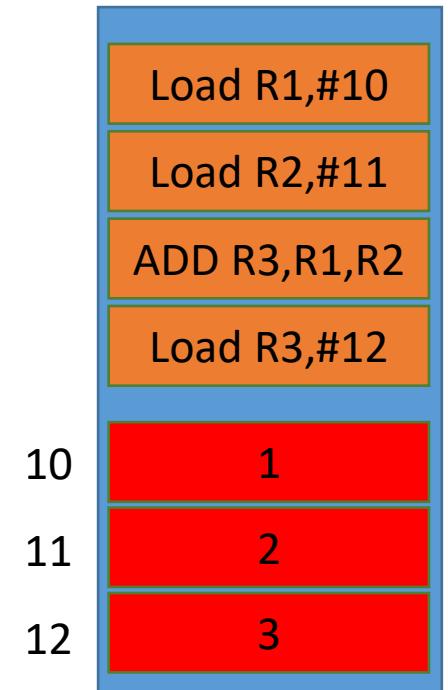
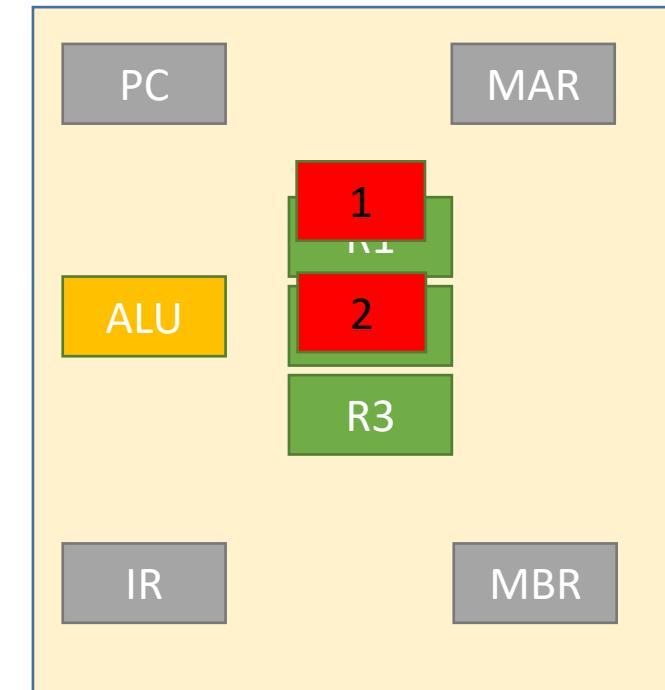
Similarly, Load R2, #11 will be executed



RISC Pipelining Issues (Case 1)

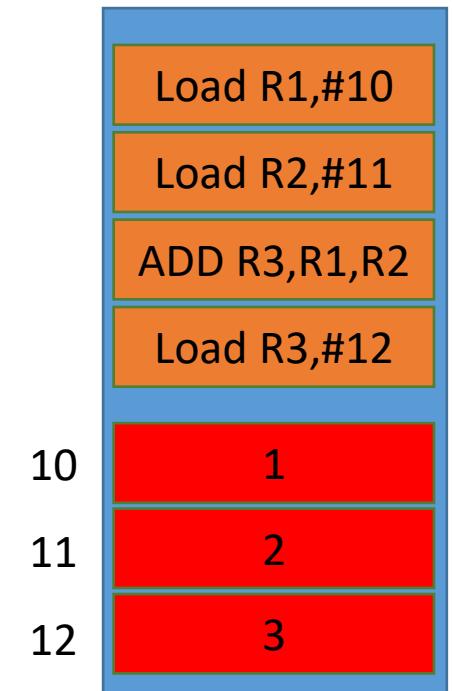
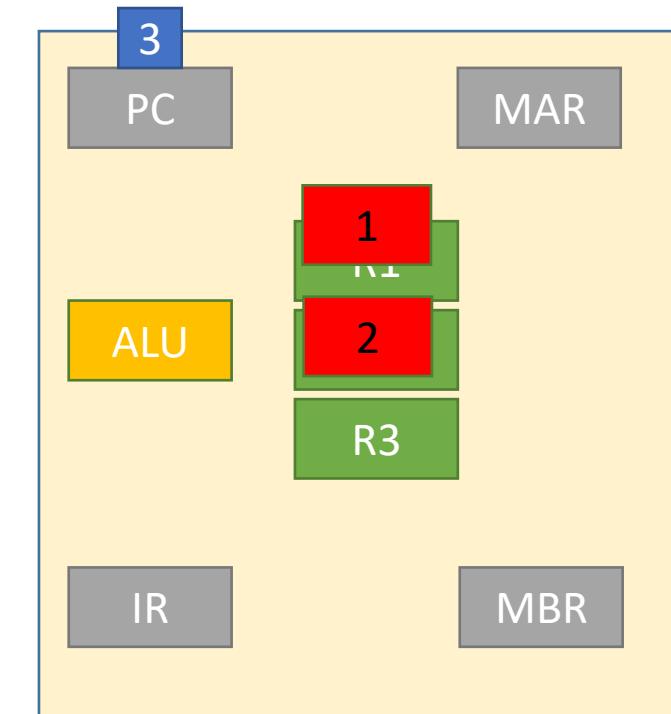
| Time Units | 1 | 2 | 3 | 4 | 5 | 6 |
|----------------|---|---|---|---|---|---|
| Load R1, #10 | I | E | D | | | |
| Load R2, #11 | | I | E | D | | |
| Add R3, R1, R2 | | | I | E | D | |
| Store R3, M | | | | I | E | D |
| | | | | | | |

ADD R3,R1,R2



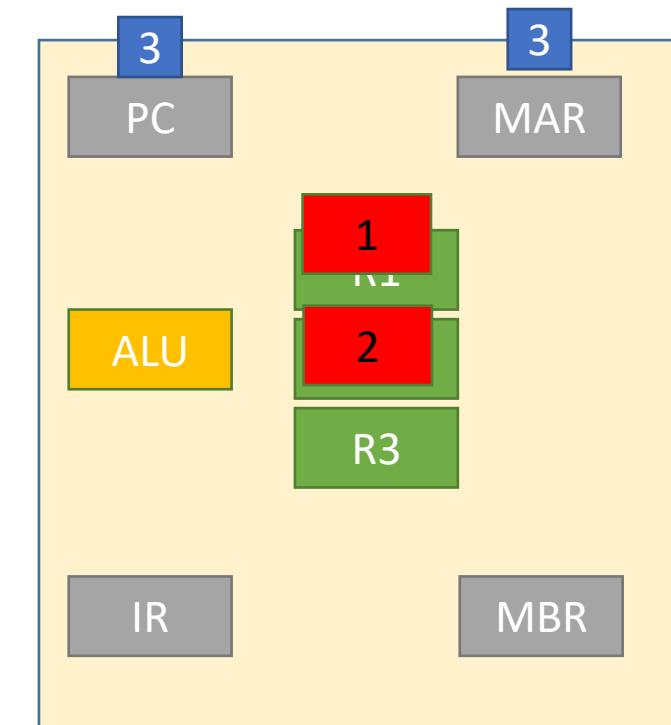
RISC Pipelining Issues (Case 1)

| Time Units | 1 | 2 | 3 | 4 | 5 | 6 |
|-------------------|---|---|---|---|---|---|
| Load R1, #10 | I | E | D | | | |
| Load R2, #11 | | I | E | D | | |
| Add R3, R1, R2 | | | I | E | D | |
| Store R3, M | | | | I | E | D |
| | | | | | | |



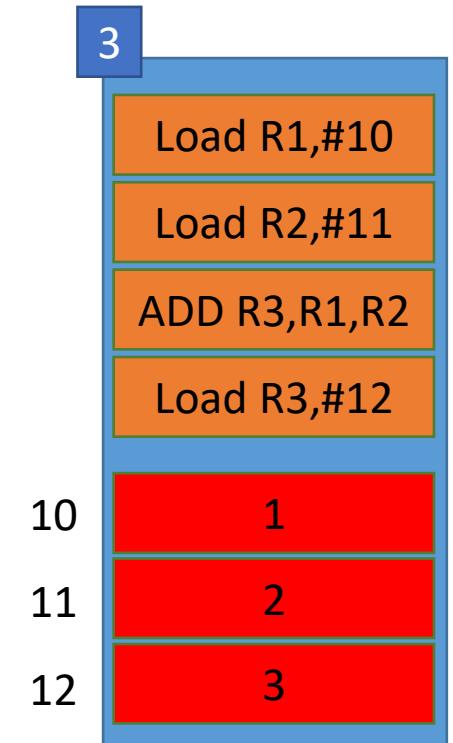
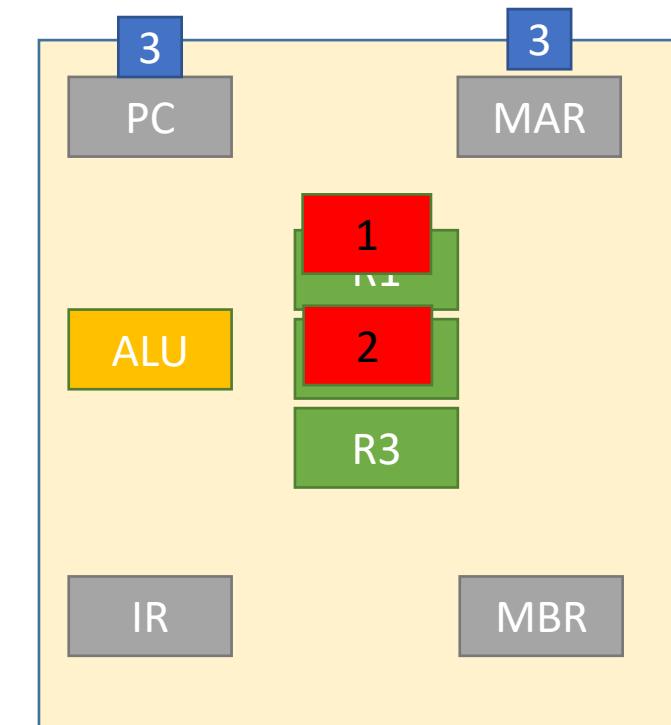
RISC Pipelining Issues (Case 1)

| Time Units | 1 | 2 | 3 | 4 | 5 | 6 |
|-------------------|---|---|---|---|---|---|
| Load R1, #10 | I | E | D | | | |
| Load R2, #11 | | I | E | D | | |
| Add R3, R1, R2 | | | I | E | D | |
| Store R3, M | | | | I | E | D |
| | | | | | | |



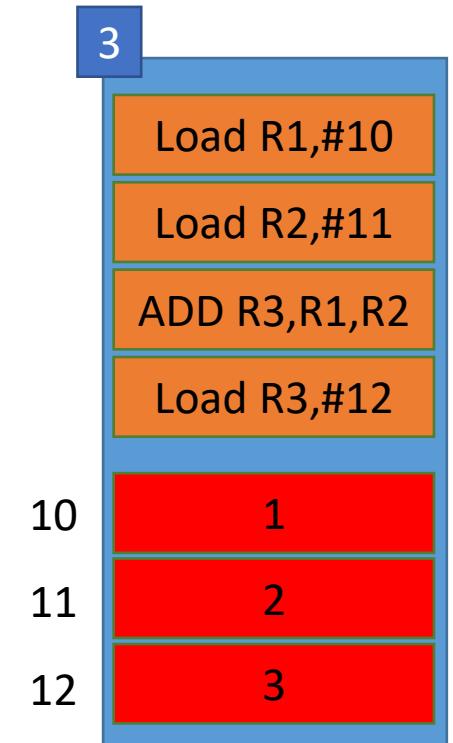
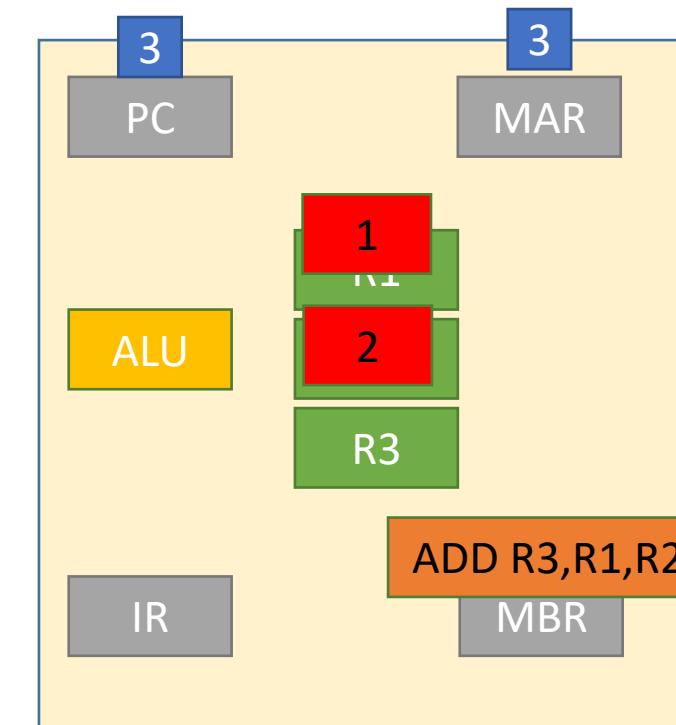
RISC Pipelining Issues (Case 1)

| Time Units | 1 | 2 | 3 | 4 | 5 | 6 |
|-------------------|---|---|---|---|---|---|
| Load R1, #10 | I | E | D | | | |
| Load R2, #11 | | I | E | D | | |
| Add R3, R1, R2 | | | I | E | D | |
| Store R3, M | | | | I | E | D |
| | | | | | | |



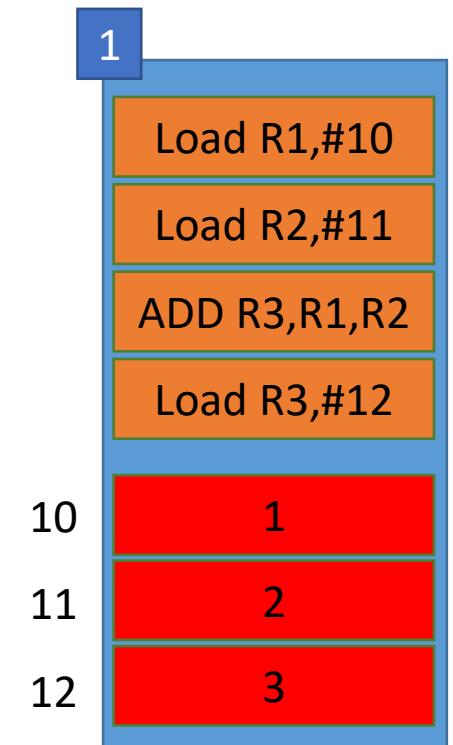
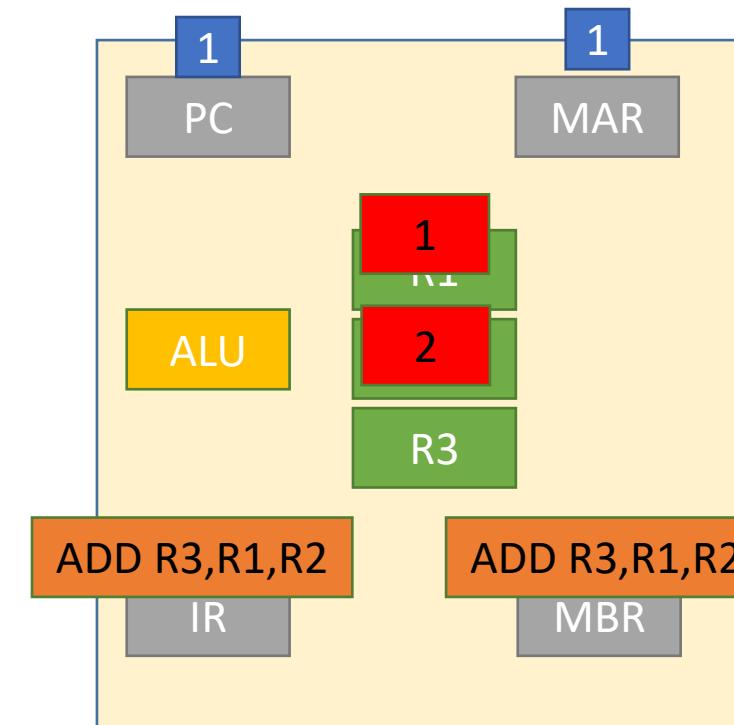
RISC Pipelining Issues (Case 1)

| Time Units | 1 | 2 | 3 | 4 | 5 | 6 |
|-------------------|---|---|---|---|---|---|
| Load R1, #10 | I | E | D | | | |
| Load R2, #11 | | I | E | D | | |
| Add R3, R1, R2 | | | I | E | D | |
| Store R3, M | | | | I | E | D |
| | | | | | | |



RISC Pipelining Issues (Case 1)

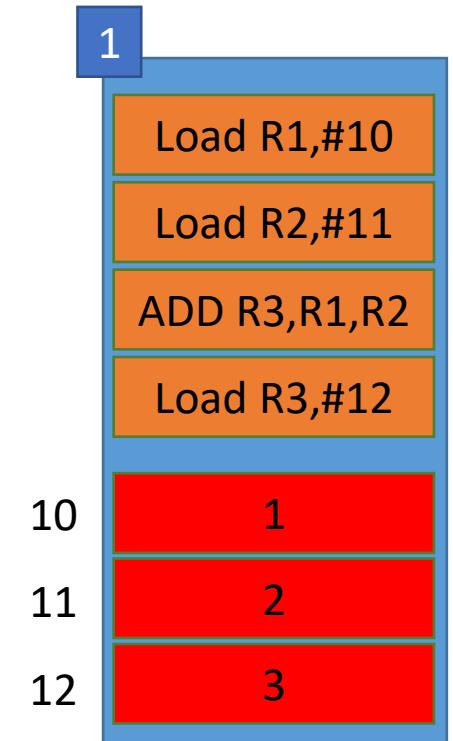
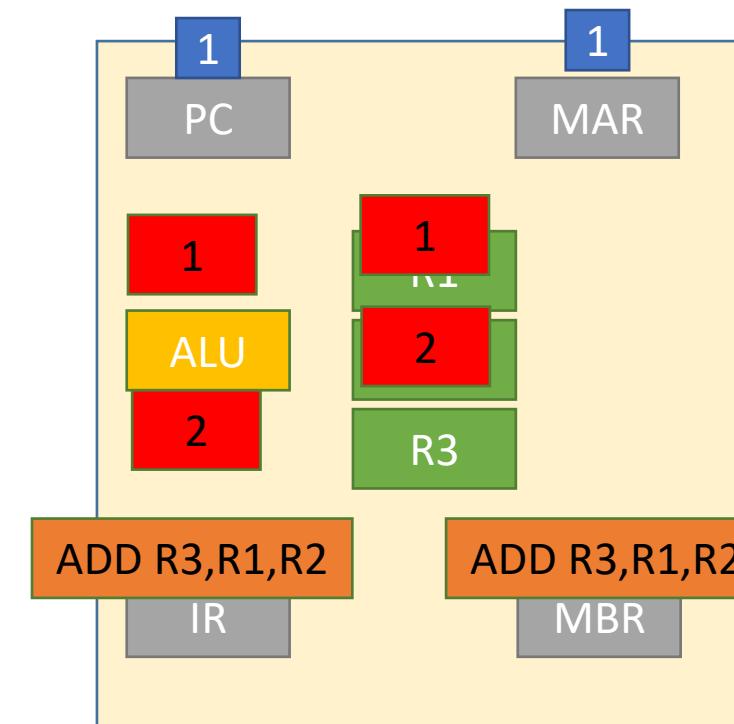
| Time Units | 1 | 2 | 3 | 4 | 5 | 6 |
|-------------------|---|---|---|---|---|---|
| Load R1, #10 | I | E | D | | | |
| Load R2, #11 | | I | E | D | | |
| Add R3, R1, R2 | | | I | E | D | |
| Store R3, M | | | | I | E | D |
| | | | | | | |



RISC Pipelining Issues (Case 1)

Recall that there were 3 buffer registers with ALU
(chapter 14)

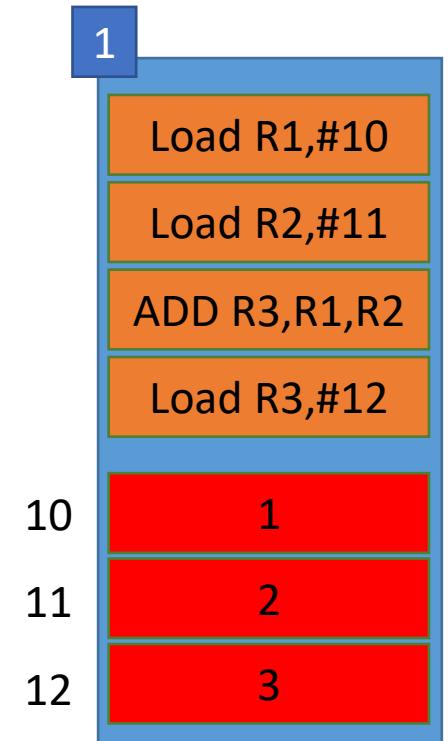
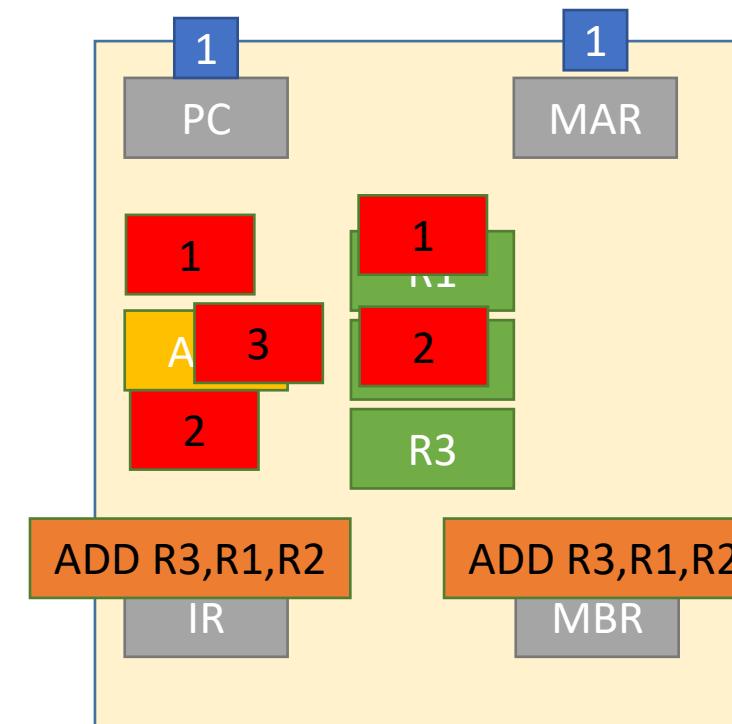
| Time Units | 1 | 2 | 3 | 4 | 5 | 6 |
|-------------------|---|---|---|---|---|---|
| Load R1, #10 | I | E | D | | | |
| Load R2, #11 | | I | E | D | | |
| Add R3, R1, R2 | | | I | E | D | |
| Store R3, M | | | | I | E | D |
| | | | | | | |



RISC Pipelining Issues (Case 1)

Recall that there were 3 buffer registers with ALU
(chapter 14)

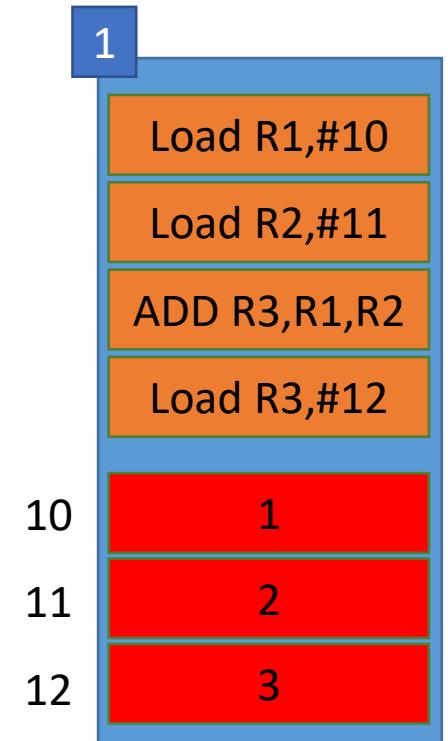
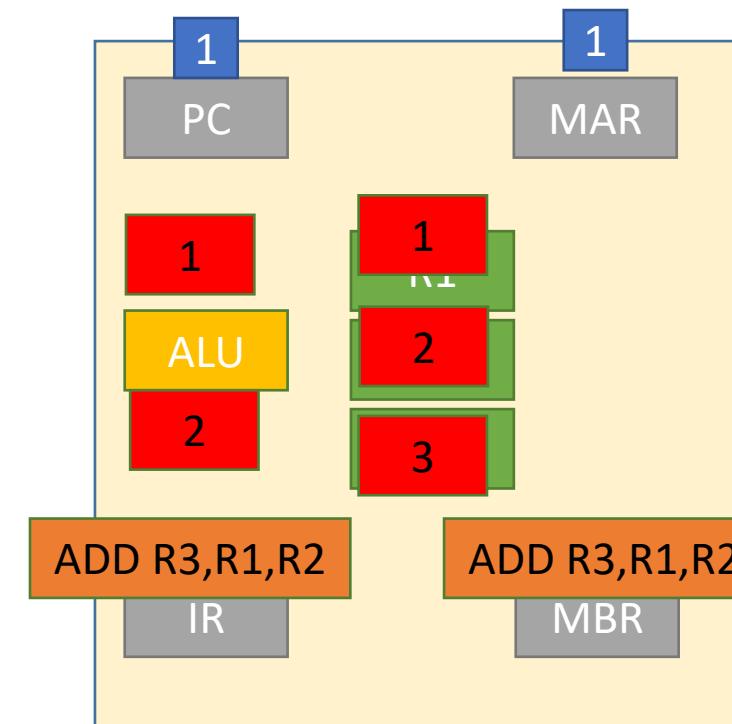
| Time Units | 1 | 2 | 3 | 4 | 5 | 6 |
|-------------------|---|---|---|---|---|---|
| Load R1, #10 | I | E | D | | | |
| Load R2, #11 | | I | E | D | | |
| Add R3, R1, R2 | | | I | E | D | |
| Store R3, M | | | | I | E | D |
| | | | | | | |



RISC Pipelining Issues (Case 1)

Recall that there were 3 buffer registers with ALU
(chapter 14)

| Time Units | 1 | 2 | 3 | 4 | 5 | 6 |
|-------------------|---|---|---|---|---|---|
| Load R1, #10 | I | E | D | | | |
| Load R2, #11 | | I | E | D | | |
| Add R3, R1, R2 | | | I | E | D | |
| Store R3, M | | | | I | E | D |
| | | | | | | |



RISC Pipelining **Issues** (Case 1)

- Have you realized yet what the issue is?

RISC Pipelining Issues (Case 1)

| Time Units | 1 | 2 | 3 | 4 | 5 | 6 |
|-------------------|---|---|---|---|---|---|
| Load R1, #10 | I | E | D | | | |
| Load R2, #11 | | I | E | D | | |
| Add R3, R1, R2 | | | I | E | D | |
| Store R3. M | | | | I | E | D |
| | | | | | | |

The instructions are being
executed in a pipeline

This instruction
Requires R2 but its not
Ready during execution

RISC Pipelining Issues (Case 1) Solution?

**"To do nothing at all is the most difficult thing in the world.
the most difficult and the
most intellectual."**

- Oscar Wilde



RISC Pipelining (Case 1 - solution: Delay after Load)

| Time Units | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|-------------------|---|---|---|---|---|---|---|
| Load R1, #10 | I | E | D | | | | |
| Load R2, #11 | | I | E | D | | | |
| No Operation | | | I | E | D | | |
| Add R3, R1, R2 | | | | I | E | D | |
| Store R3,M | | | | | I | E | D |
| | | | | | | | |

In future you will encounter a NoOp Command

Now R2 is ready for this operation

RISC Pipelining (Case 1 - solution: Delay after Load)

| Time Units | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|-------------------|---|---|---|---|---|---|---|---|
| Load R1, #10 | I | E | D | | | | | |
| Load R2, #11 | | I | E | D | | | | |
| No Operation | | | I | E | D | | | |
| Add R3, R1, R2 | | | | I | E | D | | |
| No Operation | | | | | I | E | D | |
| Store R3,M | | | | | | I | E | D |

Note that, the same measure is to be taken for Store R3, M.

As it requires the contents of R3 to be fully ready before sending it to the memory

RISC Pipelining (Case 2)

| Time Units | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|------------------------------|---|---|---|---|---|---|---|---|
| Load R1, #10 | I | E | D | | | | | |
| Inc R2 | | I | E | D | | | | |
| Add R10,R20 | | | I | E | D | | | |
| Sub R11,R31 | | | | I | E | D | | |
| JGE X | | | | | I | E | D | |
| Some Task / Instruction at X | | | | | | I | E | D |

Depending on the change taken place here in the PC (after stage D)
Either the instruction after JGE X will be executed.
Or the instruction at X will be loaded.

RISC Pipelining (Case 2)

| Time Units | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|------------------------------|---|---|---|---|---|---|---|---|
| Load R1, #10 | I | E | D | | | | | |
| Inc R2 | | I | E | D | | | | |
| Add R10,R20 | | | I | E | D | | | |
| Sub R11,R31 | | | | I | E | D | | |
| JGE X | | | | | I | E | D | |
| Some Task / Instruction at X | | | | | | I | E | D |

Normally, the next instruction after JGE X will be brought in here (as the program counter keeps on increasing)

RISC Pipelining (Case 2)

Solution?

No operation

RISC Pipelining (Case 2 - Solution : Delay after branch)

| Time Units | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | | |
|---|---|---|---|---|---|---|---|---|---|--|
| Load R1, #10 | I | E | D | | | | | | | |
| Inc R2 | | I | E | D | | | | | | |
| Add R10,R20 | | | I | E | D | | | | | |
| Sub R11,R31 | | | | I | E | D | | | | |
| JGE X | | | | | I | E | D | | | |
| No Op | | | | | | I | E | D | | |
| No Op | | | | | | | I | E | D | |
| Instruction at X/ instruction after JGE X | | | | | | | I | E | D | |

JGE X is getting enough time to update the program counter

Hence the appropriate instruction will be loaded here.

Thank You

INSTRUCTION-LEVEL PARALLELISM & SUPERSCALAR PROCESSORS

Chapter 16

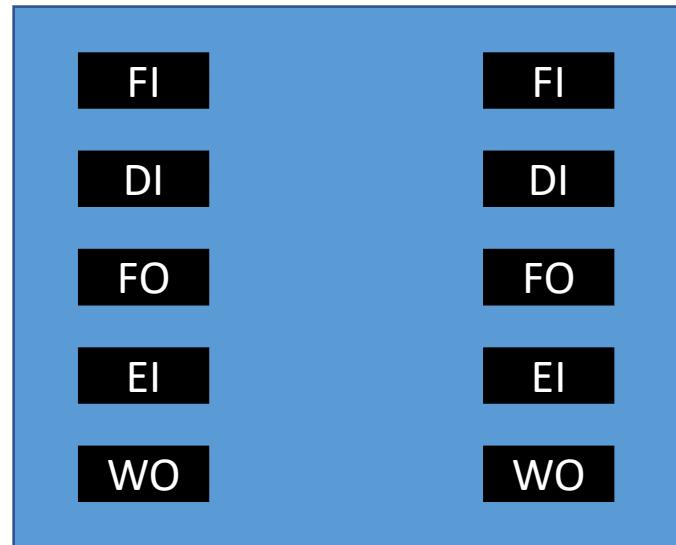
1. Introduction

Introduction

- In superscalar processors we execute instructions in multiple streams/pipelines.
- Also, instructions may be executed in an out of order fashion as well (this affects the WAW and WAR dependencies).
- If there are n-pipelines, there can be n+1 ALUs as well.

Introduction

- In superscalar processors we execute instructions in multiple streams/pipelines.
- Also, instructions may be executed in an out of order fashion as well (this affects the WAW and WAR dependencies).
- If there are n-pipelines, there can be n+1 ALUs as well.



Introduction

- In superscalar processors we execute instructions in multiple streams/pipelines.
- Also, instructions may be executed in an out of order fashion as well.
- If there are n-pipelines, there can be n+1 ALUs as well.

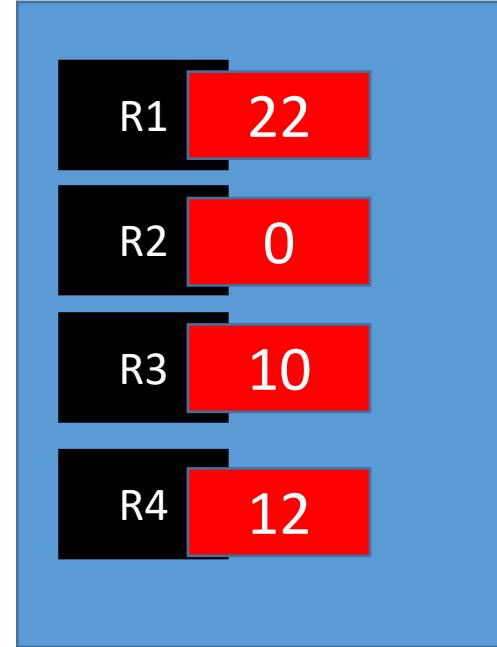
ADD R1, R3, R4
MOV R3, R2



Introduction

- In superscalar processors we execute instructions in multiple streams/pipelines.
- Also, instructions may be executed in an out of order fashion as well.
- If there are n-pipelines, there can be n+1 ALUs as well.

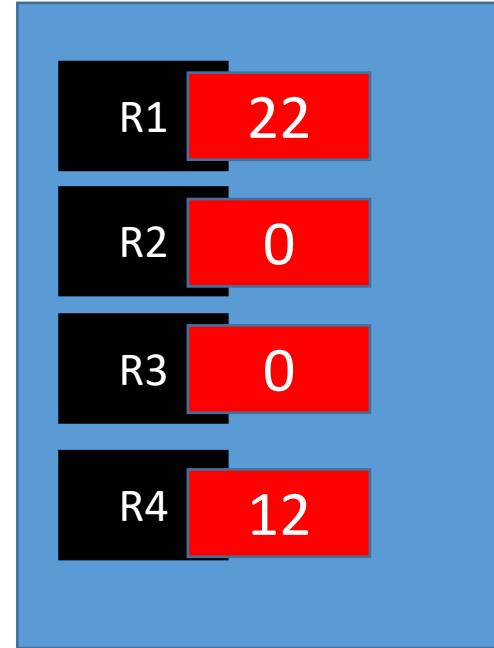
ADD R1, R3, R4
MOV R3, R2



Introduction

- In superscalar processors we execute instructions in multiple streams/pipelines.
- Also, instructions may be executed in an out of order fashion as well.
- If there are n-pipelines, there can be n+1 ALUs as well.

ADD R1, R3, R4
MOV R3, R2



Introduction

- In superscalar processors we execute instructions in multiple streams/pipelines.
- Also, instructions may be executed in an out of order fashion as well.
- If there are n-pipelines, there can be n+1 ALUs as well.

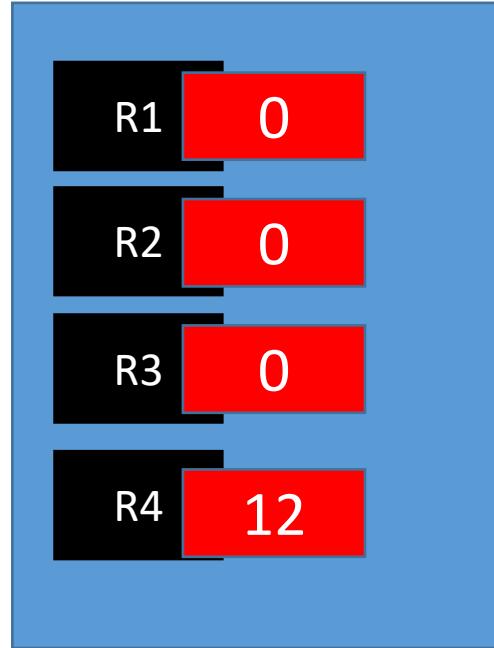
Out of order seq:
MOV R3, R2
ADD R1, R3, R4



Introduction

- In superscalar processors we execute instructions in multiple streams/pipelines.
- Also, instructions may be executed in an out of order fashion as well.
- If there are n-pipelines, there can be n+1 ALUs as well.

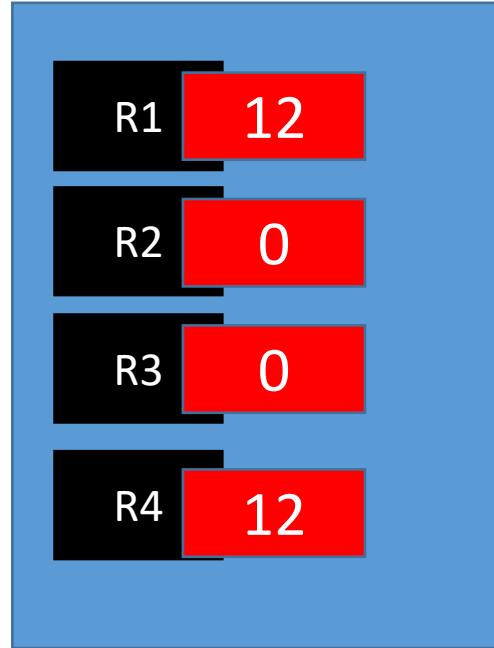
Out of order seq:
MOV R3, R2
ADD R1, R3, R4



Introduction

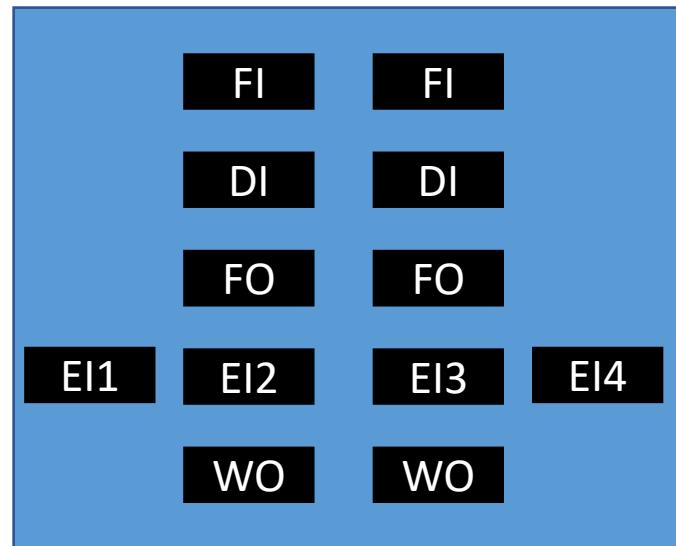
- In superscalar processors we execute instructions in multiple streams/pipelines.
- Also, instructions may be executed in an out of order fashion as well.
- If there are n-pipelines, there can be n+1 ALUs as well.

Out of order seq:
MOV R3, R2
ADD R1, R3, R4



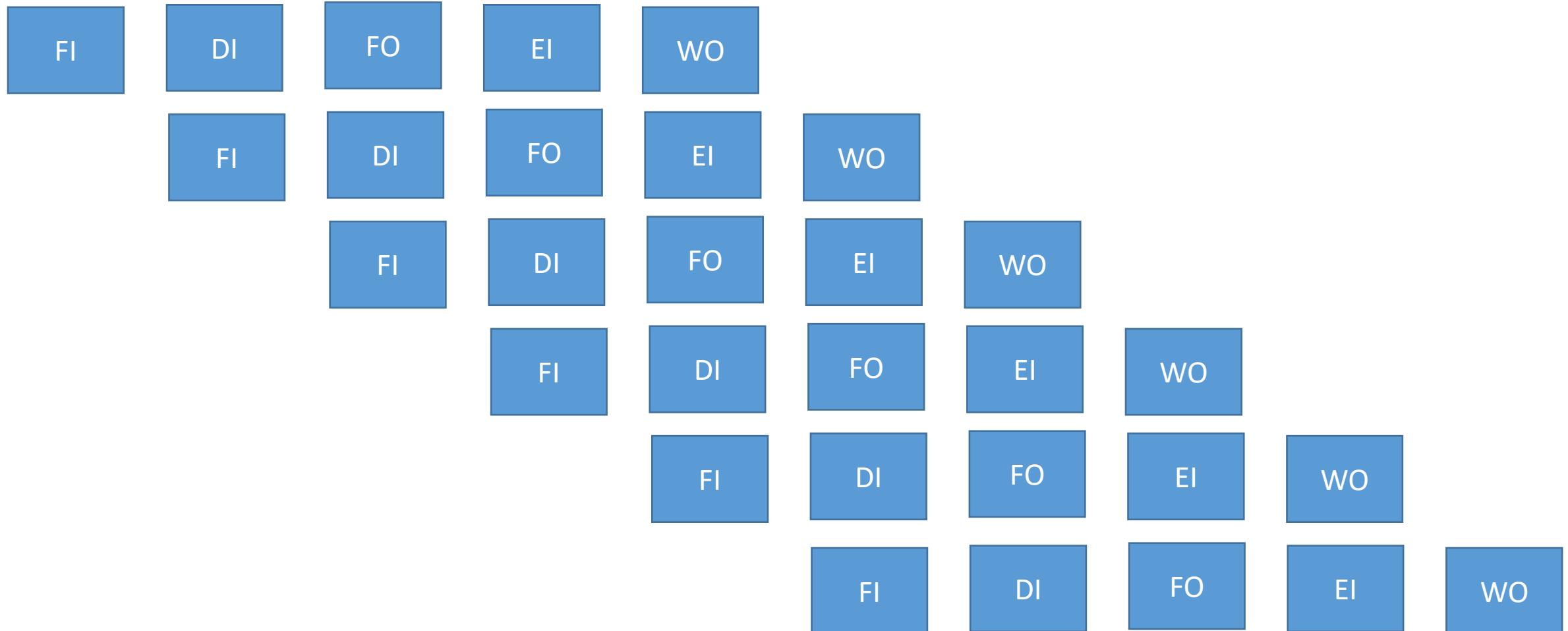
Introduction

- In superscalar processors we execute instructions in multiple streams/pipelines.
- Also, instructions may be executed in an out of order fashion as well (this affects the WAW and WAR dependencies).
- If there are n-pipelines, there can be n+1 (or more) ALUs as well.

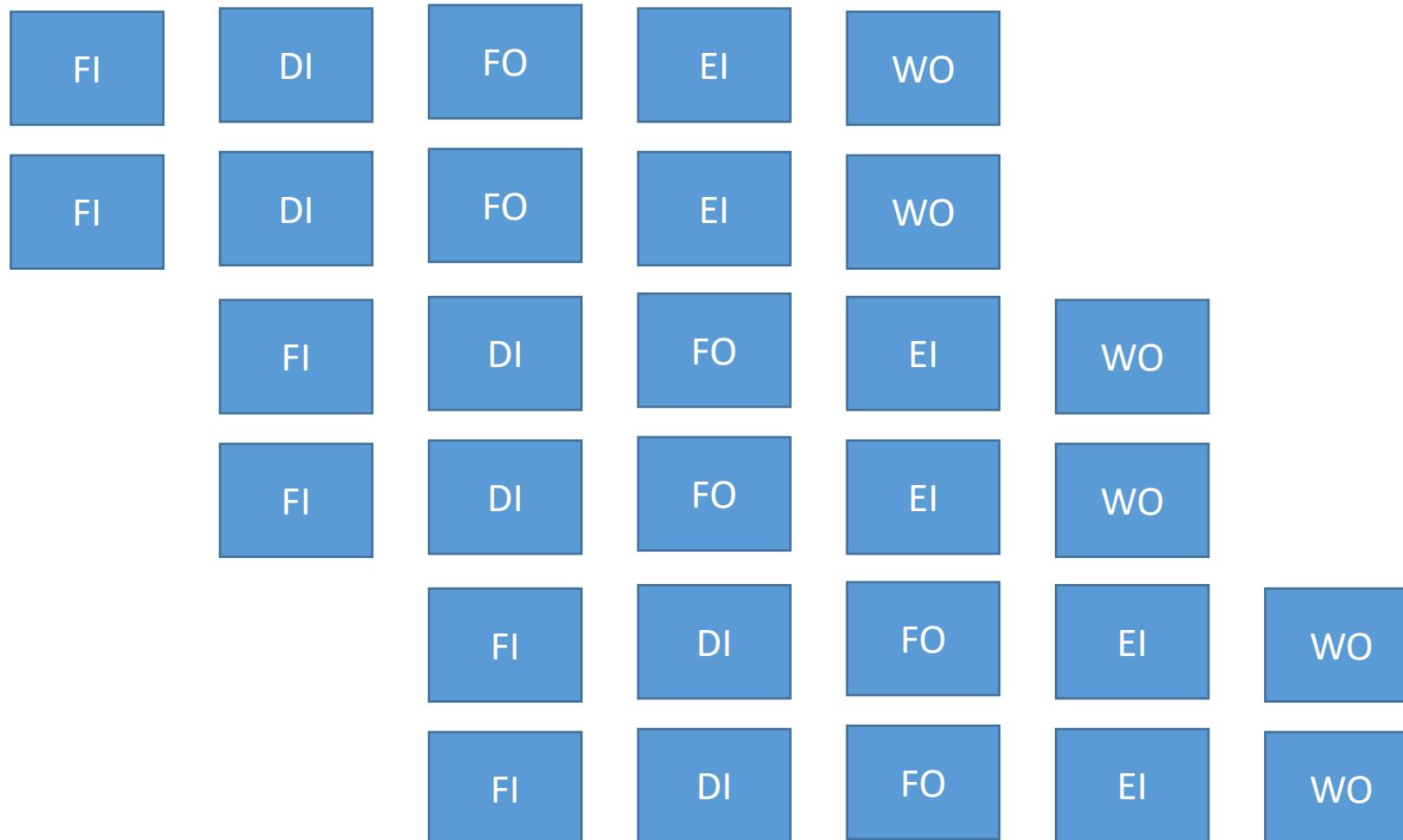


2.The Superscalar Pipeline

The type of pipelines you have seen so far:



The type of pipelines you have seen so far:



The type of pipelines you have seen so far:



There are two Program counters.

Bringing in instructions.

And being incremented together.

3. Resource Conflicts!

The core issue with Superscalar Processors

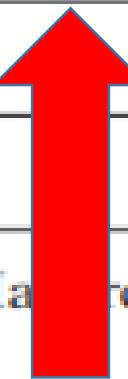
Recall the dependencies from chapter 14

- Read after Write (RAW)
- Write after Write (WAW)
- Write after Read (WAR)

Read After Write Dependency and Issues

| ADD EAX, EBX | FI | DI | FO | EI | WO | | | | | |
|---------------------|-----------|-----------|-----------|-------------|-----------|-----------|-----------|-----------|-----------|-----------|
| SUB ECX, EAX | | FI | DI | Idle | | FO | EI | WO | | |
| | | | FI | | | DI | FO | EI | WO | |
| | | | | | | FI | DI | FO | EI | WO |
| | | | | | | | | | | |

Figure 14.16 Example of Data Hazard



Also known as **pipeline stalling**

Write after Write (WAW)

- $R1 = R2 + R3$
- $R1 = R5 + R6$
- If executed sequentially, no issues.
- If executed out of order:
 - This will cause erroneous results to be delivered.
 - Order may change if the two instructions are fed through two different pipelines

Write after Read (WAR)

- $R1 = R2 + R3$
- $R2 = R5 + R6$
- If executed sequentially, no issues.
- If executed out of order:
 - This will cause erroneous results to be delivered.
 - Order may change if the two instructions are fed through two different pipelines

3. Instruction Level Parallelism and Machine Level Parallelism

Instruction Level Parallelism

- The instructions on the **left** are **independent**
- The instructions to the **right** are not **independent**
- Instruction level parallelism can be achieved easily with the instructions on the left (as orders do not matter)

Load R1 \leftarrow R2

Add R3 \leftarrow R3, "1"

Add R4 \leftarrow R4, R2

Add R3 \leftarrow R3, "1"

Add R4 \leftarrow R3, R2

Store [R4] \leftarrow R0

Instruction Level Parallelism

- The instructions on the **left** are **independent**
- The instructions to the **right** are not **independent**
- Instruction level parallelism can be achieved easily with the instructions on the left (as orders do not matter)

Load R1 \leftarrow R2

Add R3 \leftarrow R3, "1"

Add R4 \leftarrow R4, R2

Add R3 \leftarrow R3, "1"

Add R4 \leftarrow R3, R2

Store [R4] \leftarrow R0

RAW

Instruction Level Parallelism

- The instructions on the **left** are **independent**
- The instructions to the **right** are not **independent**
- Instruction level parallelism can be achieved easily with the instructions on the left (as orders do not matter)

Load R1 \leftarrow R2

Add R3 \leftarrow R3, "1"

Add R4 \leftarrow R4, R2

Add R3 \leftarrow R3, "1"

Add R4 \leftarrow R3, R2

Store [R4] \leftarrow R0

WAW

Machine Level parallelism

- **Ability of the processor** to take advantage of instruction level parallelism.
- It is determined by the **number of parallel pipelines** working at the same time.
- It is easy to enhance machine level parallelism if the instructions are fixed length

4. Policies and Protocols

A common solution for flow of data through shared resources

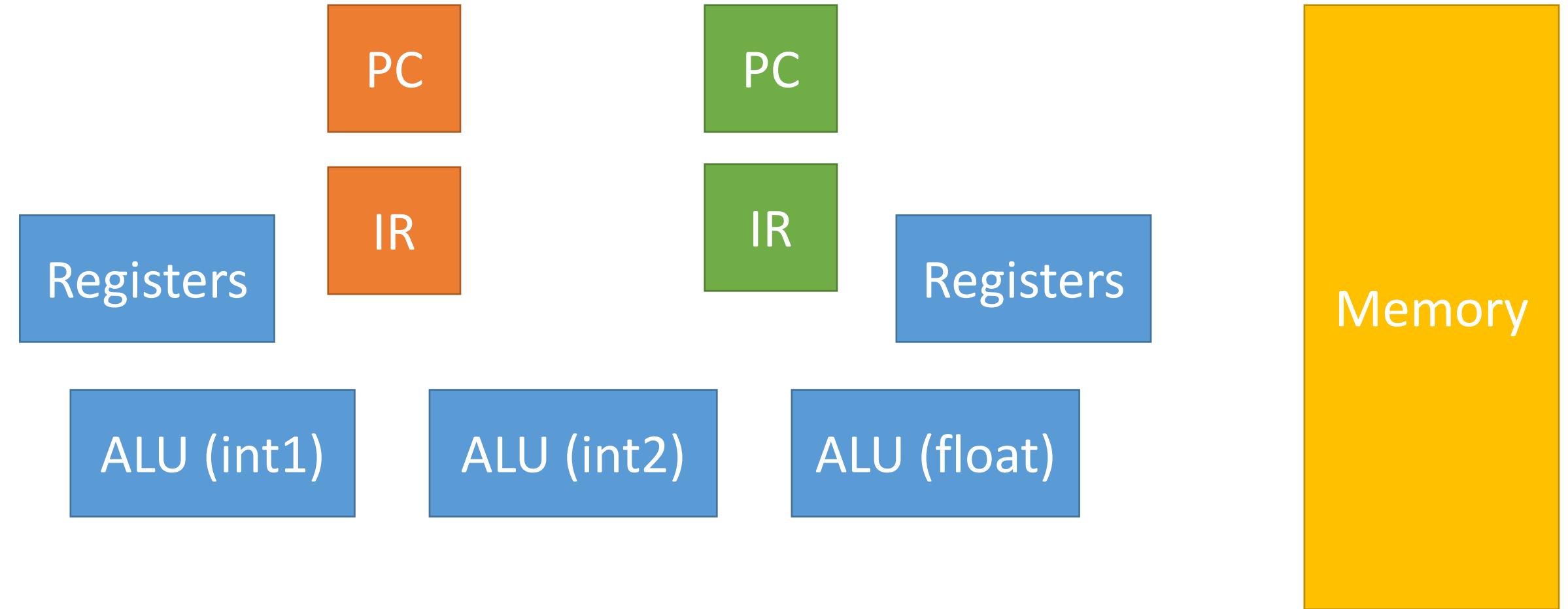
Instruction Issue Policy/Protocol

Basically referring to some rules that dictate the flow of data.

3 Protocols that we will study:

- In order issue, in order completion
- In order issue, out of order completion
- Out of order issue, out of order completion

Let us consider the following superscalar processor structure



Let us consider the following instructions

I1 : Needs **two clock cycles** in the execution stage, Int ALU

I2 : Int ALU

I3 : Requires the floating point ALU

I4 : Requires the floating point ALU

I5 : Depends on the value produced by I4 (RAW), requires Int2 ALU

I6 : Requires int2 ALU

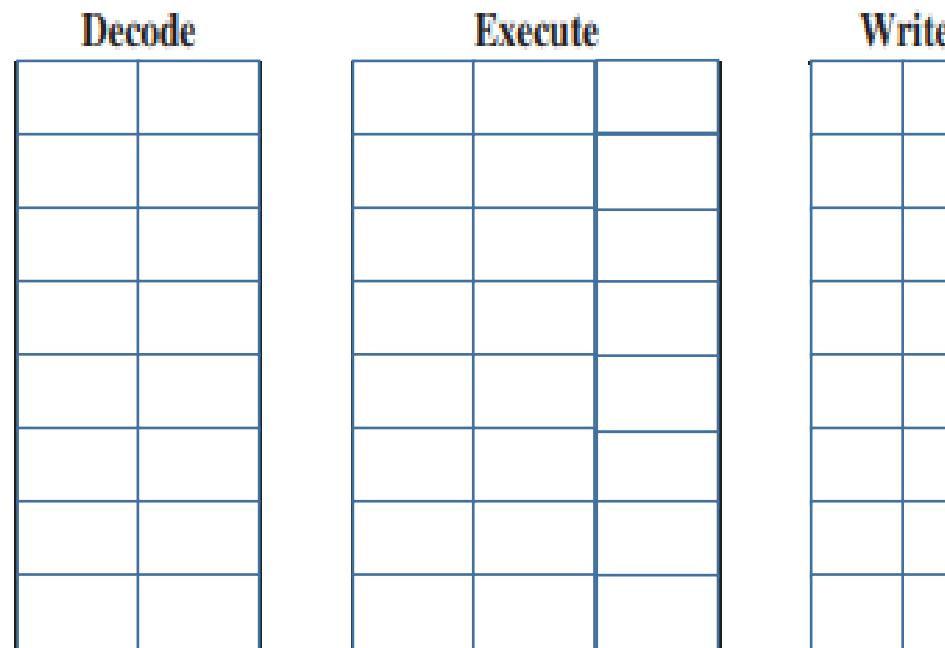
In order issue, in order completion

This is just a baseline policy, it is there for just comparison purposes:

1. Bring in two instructions at a time from the memory (the PCs are being incremented together)
2. If the required ALU is free, send it there.
3. If an instruction is being executed for more than 1 clock cycle, do not send in a new instruction.
4. The results of the instruction are to be written in order.
5. The results of two instructions are written out together.

In order issue, in order completion

1. Bring in **two instructions** at a time from the memory (the PCs are being incremented together)
2. If the required **ALU** is **free**, send it there.
3. If an instruction is being executed for **more than 1 clock cycle**, do not send in a new instruction.
4. The results of the instruction are to be **written in order**.

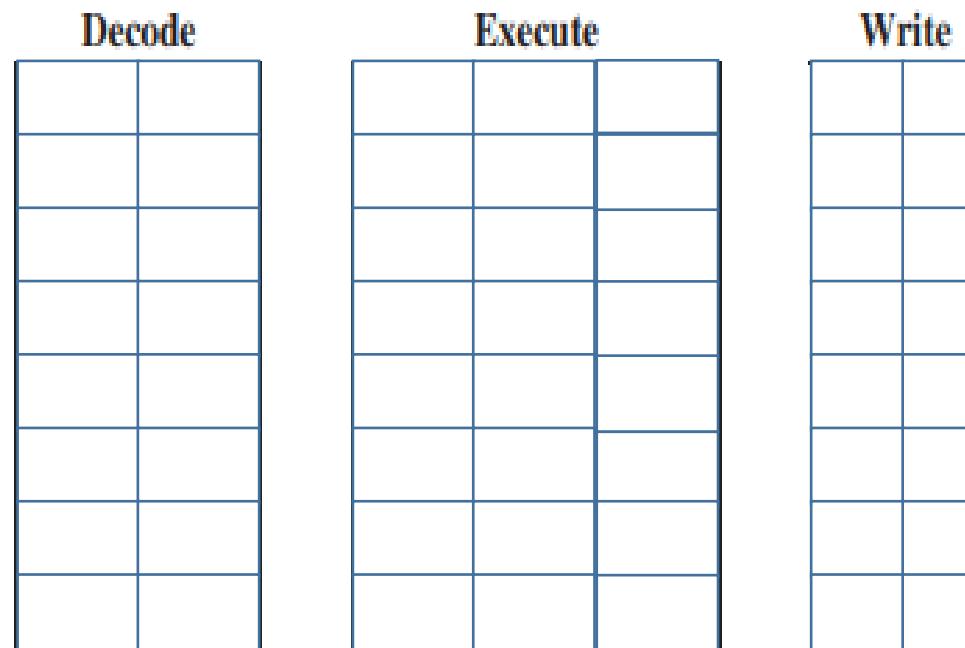


(a) In-order issue and in-order completion

- I1 needs **2 cycles**
- I3 and I4 conflicts for the same floating point unit
- I5 depends on the value produced by I4
- I5 and I6 conflict for the same integer unit

In order issue, in order completion

1. Bring in **two instructions** at a time from the memory (the PCs are being incremented together)
2. If the required **ALU** is **free**, send it there.
3. If an instruction is being executed for **more than 1 clock cycle**, do not send in a new instruction.
4. The results of the instruction are to be **written in order**.

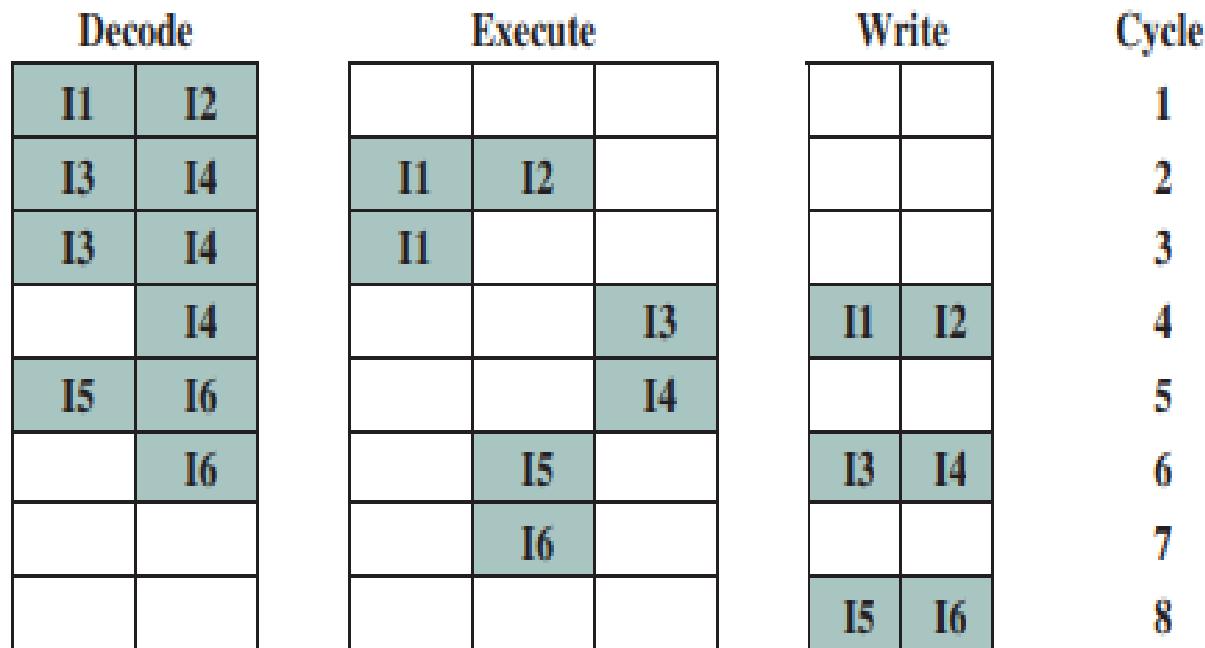


(a) In-order issue and in-order completion

- I1 needs **2 cycles**
- I3 and I4 conflicts for the same floating point unit
- I5 depends on the value produced by I4
- I5 and I6 conflict for the same integer unit

In order issue, in order completion

1. Bring in **two instructions** at a time from the memory (the PCs are being incremented together)
2. If the required **ALU** is **free**, send it there.
3. If an instruction is being executed for **more than 1 clock cycle**, do not send in a new instruction.
4. The results of the instruction are to be **written in order**.



(a) In-order issue and in-order completion

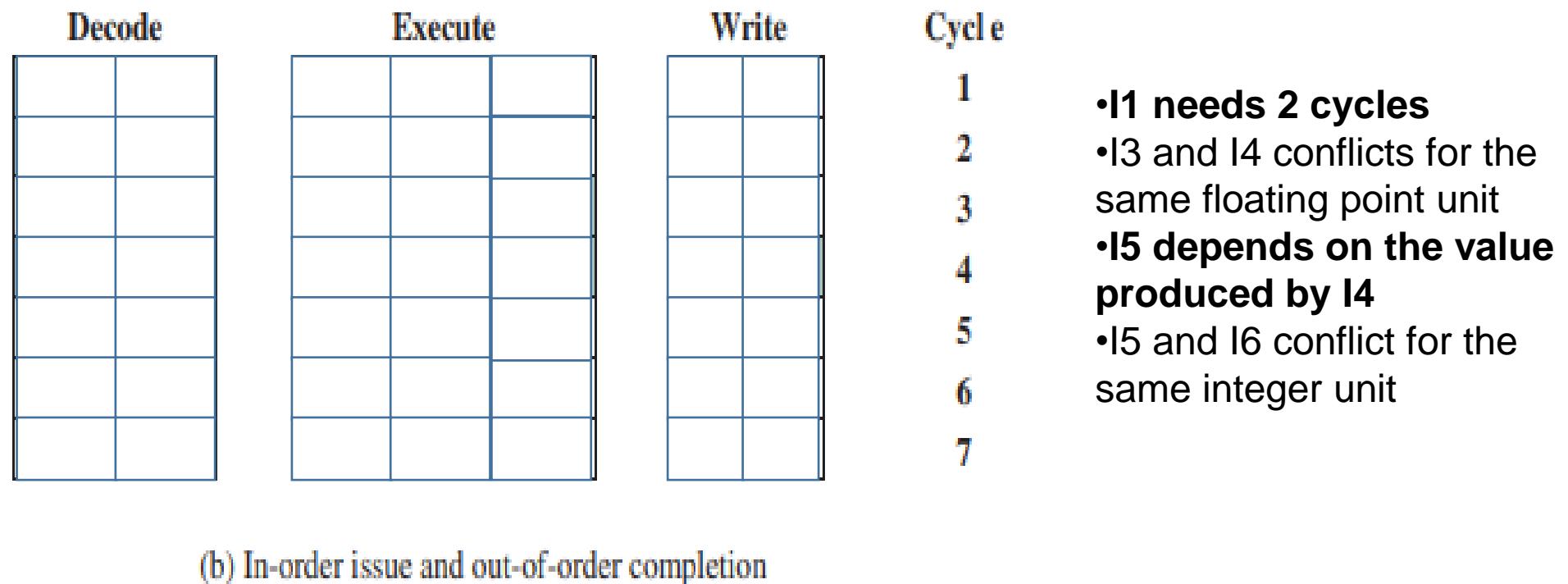
- I1 needs **2 cycles**
- I3 and I4 conflicts for the same floating point unit
- I5 depends on the value produced by I4
- I5 and I6 conflict for the same integer unit

In order issue, out of order completion

1. Bring in two instructions at a time from the memory (the PCs are being incremented together)
2. If the required ALU is free, send it there.
3. Write out the results as soon as it is done.

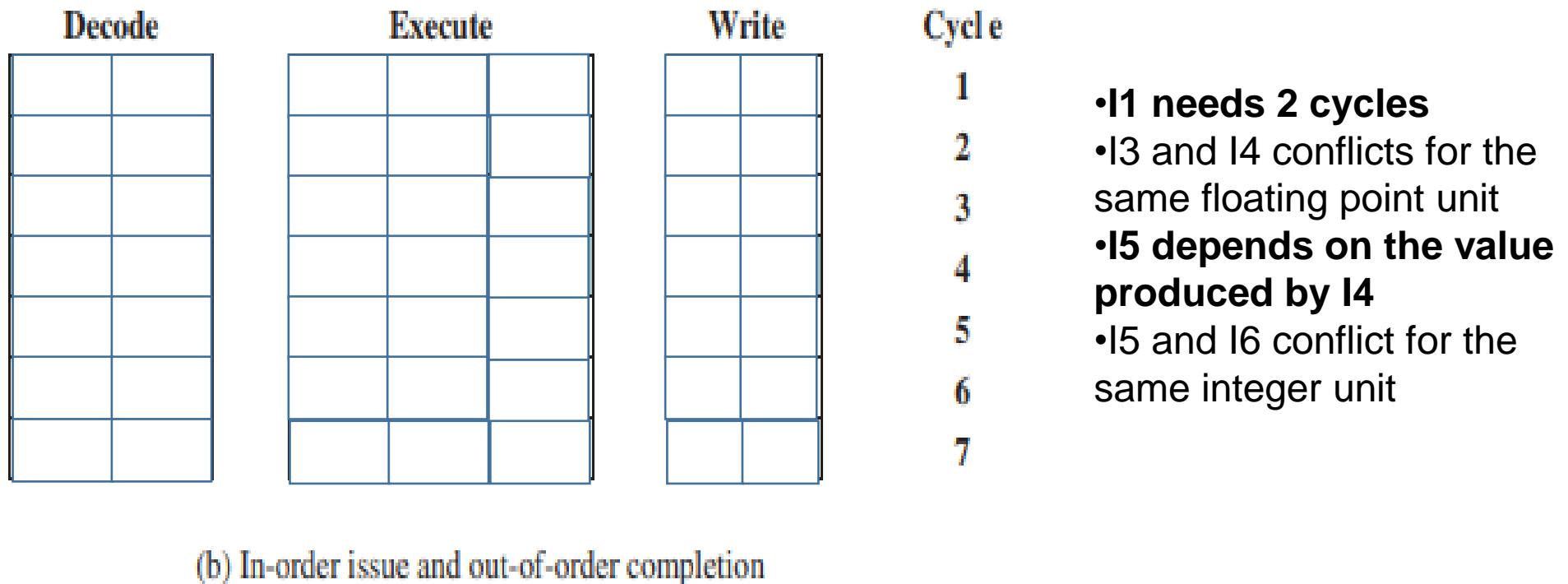
In order issue, in order completion

1. Bring in two instructions at a time from the memory (the PCs are being incremented together)
2. If the required ALU is free, send it there.
4. Write out the results as soon as it is done.



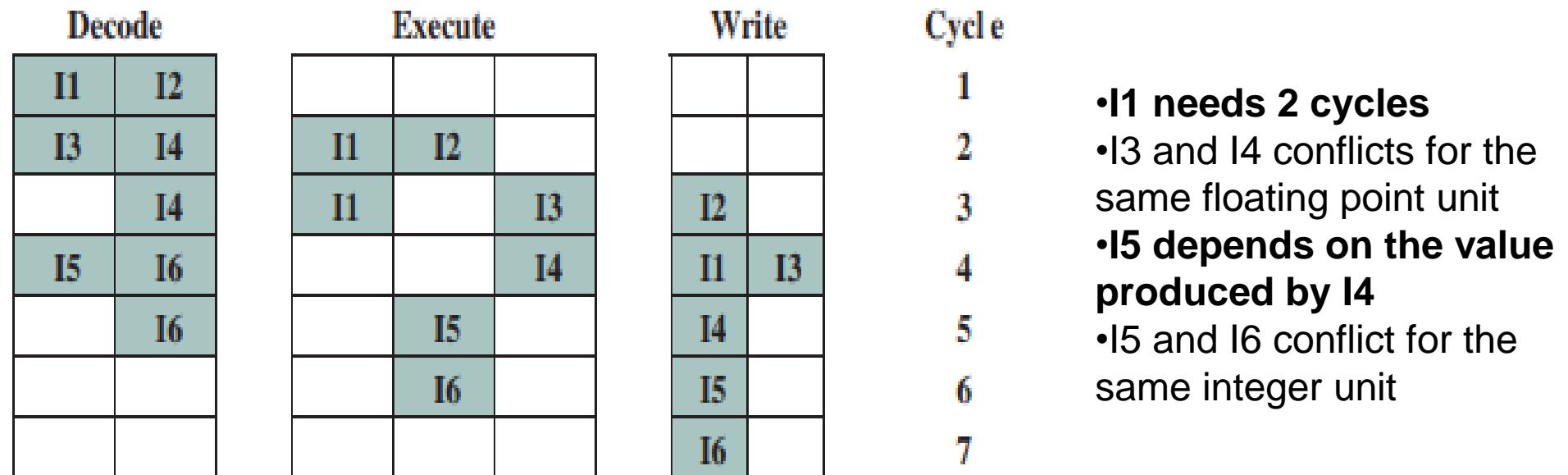
In order issue, in order completion

1. Bring in two instructions at a time from the memory (the PCs are being incremented together)
2. If the required ALU is free, send it there.
4. Write out the results as soon as it is done.



In order issue, in order completion

1. Bring in two instructions at a time from the memory (the PCs are being incremented together)
2. If the required ALU is free, send it there.
4. Write out the results as soon as it is done.



(b) In-order issue and out-of-order completion

- I1 needs 2 cycles
- I3 and I4 conflicts for the same floating point unit
- I5 depends on the value produced by I4
- I5 and I6 conflict for the same integer unit

Out of order issue, out of order completion

1. A buffer is there, for storing the instructions in an unordered manner.
2. From the buffer, the ready to be executed instruction is sent for execution
3. Bring in two instructions at a time from the memory (the PCs are being incremented together)
4. Write out the results as soon as it is done.

Out of order issue and out of order completion:

1. A buffer is there, for storing the instructions in an unordered manner.
2. From the buffer, the ready to be executed instruction is sent for execution
3. Bring in two instructions at a time from the memory (the PCs are being incremented together)
4. Write out the results as soon as it is done.

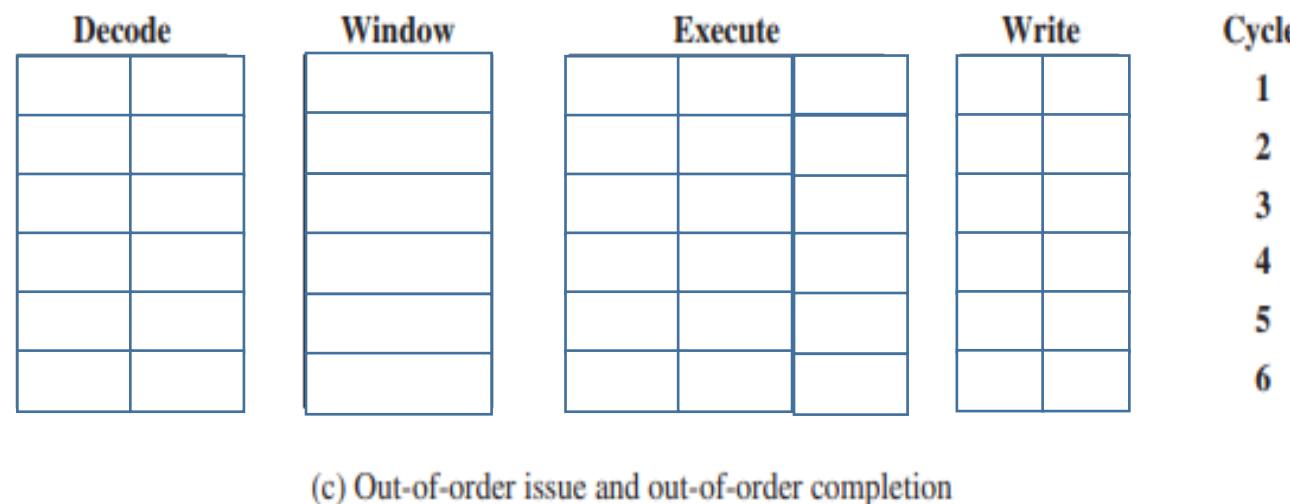


Figure 16.4 Superscalar Instruction Issue and Completion Policies

- I1 needs 2 cycles
- I3 and I4 conflicts for the same floating point unit
- I5 depends on the value produced by I4
- I5 and I6 conflict for the same integer unit

Out of order issue and out of order completion:

1. A buffer is there, for storing the instructions in an unordered manner.
2. From the buffer, the ready to be executed instruction is sent for execution
3. Bring in two instructions at a time from the memory (the PCs are being incremented together)
4. Write out the results as soon as it is done.

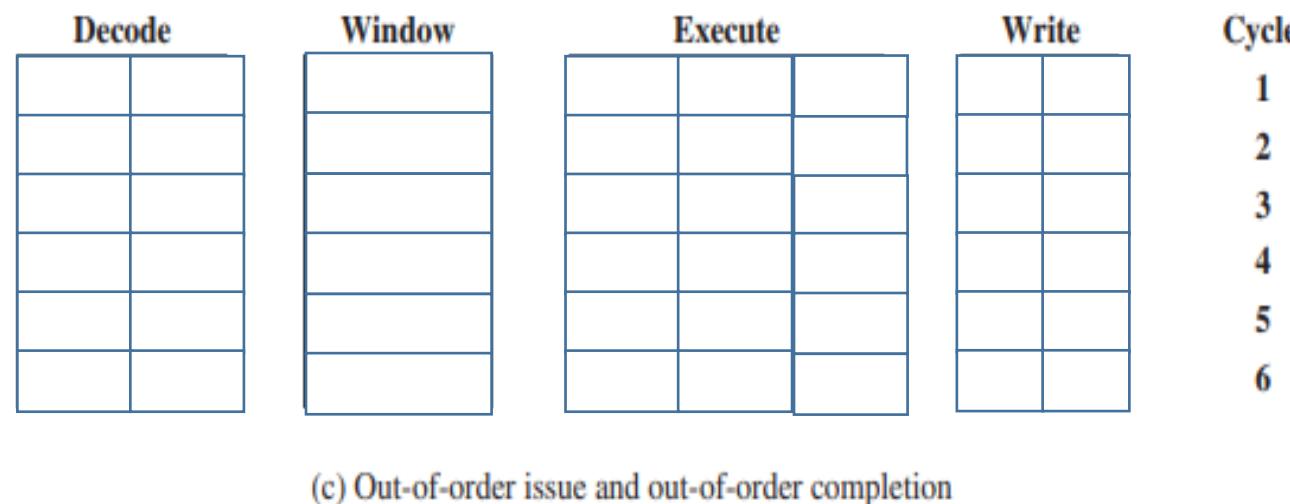
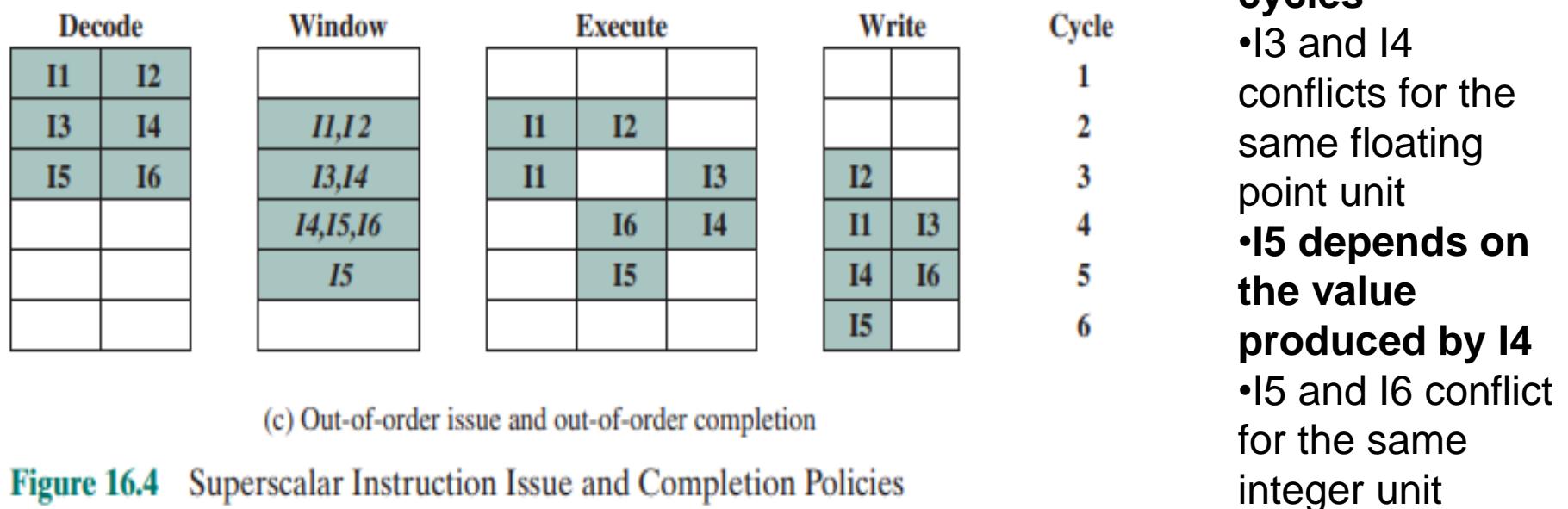


Figure 16.4 Superscalar Instruction Issue and Completion Policies

- I1 needs 2 cycles
- I3 and I4 conflicts for the same floating point unit
- I5 depends on the value produced by I4
- I5 and I6 conflict for the same integer unit

Out of order issue and out of order completion:

1. A buffer is there, for storing the instructions in an unordered manner.
2. From the buffer, the ready to be executed instruction is sent for execution
3. Bring in two instructions at a time from the memory (the PCs are being incremented together)
4. Write out the results as soon as it is done.



4. Register Renaming

Duplication of resources to avoid conflicts between out of order instructions

Example 1:

- MUL R2, R2, R2
- ADD R1,R1,R2
- MUL R2, R4, R4
- ADD R3, R3, R2
- MUL R2, R6, R6
- ADD R5, R5, R2

Example 1:

- MUL R2, R2, R2
- ADD R1,R1,R2
- MUL R2, R4, R4
- ADD R3, R3, R2
- MUL R2, R6, R6
- ADD R5, R5, R2
- $R2 = R2 * R2$
- $R1 = R1 + R2$
- $R2 = R4 * R4$
- $R3 = R3 + R2$
- $R2 = R6 * R6$
- $R5 = R5 + R2$

Example 2:

- I1: $R3 = R3 \text{ op } R5$
- I2: $R4 = R3 + 1$
- I3: $R3 = R5 + 1$
- I4: $R7 = R3 \text{ op } R4$

Example 3:

- I0: R3 + 1 -> R3
- I1: R3 + R2 -> R4
- I2: R3 op R4 -> R7
- I3: R0 -> R4

Thank You

Parallel Processing

Chapter 17

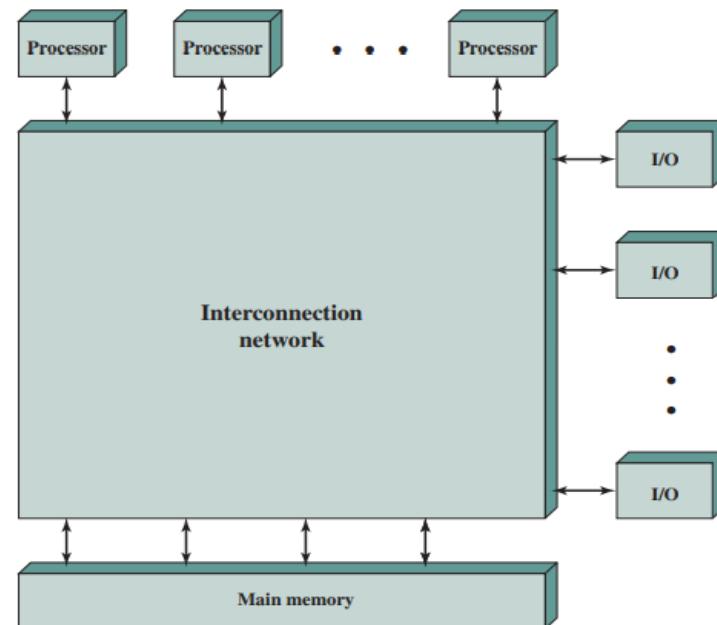
1. Introduction to Parallel Processing

Intro to Parallel Processing

- Multiple processors may share a **single memory**
- The processors share **messages** with each other.
- A system based on these criteria - symmetric multiprocessors (**SMP**) System

Intro to Parallel Processing

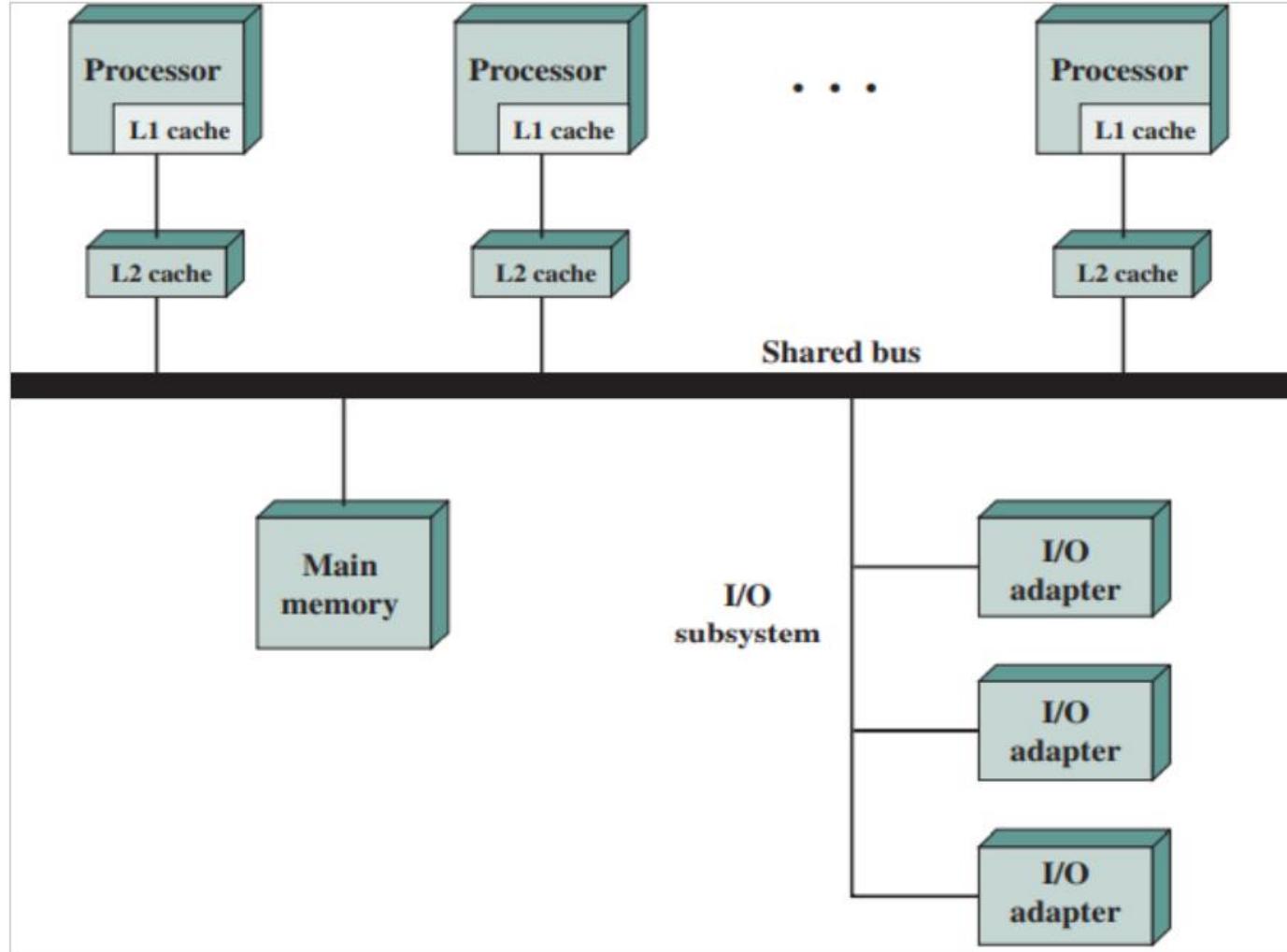
- Multiple processors may share a **single memory**
- The processors share **messages** with each other.
- A system based on these criteria - symmetric multiprocessors (**SMP**) System



What is an SMP system?

- Two or more **similar processors** will be there (helps in easier implementation and co-ordination)
- Main **memory** and i/o **devices** will be shared via **bus**
- Each processors will execute **instructions from the memory**
- **Integrated OS** – will run be run in one of the processors
- This OS will **schedule** the timing for **other processors**.
- This OS will **set** the tasks for **other processors**.
- **Note:** These controlling tasks are done by sending messages via bus

What is an SMP system?



The OS containing processor sets the PC, for the other processors (i.e it decides which processor will execute which part).

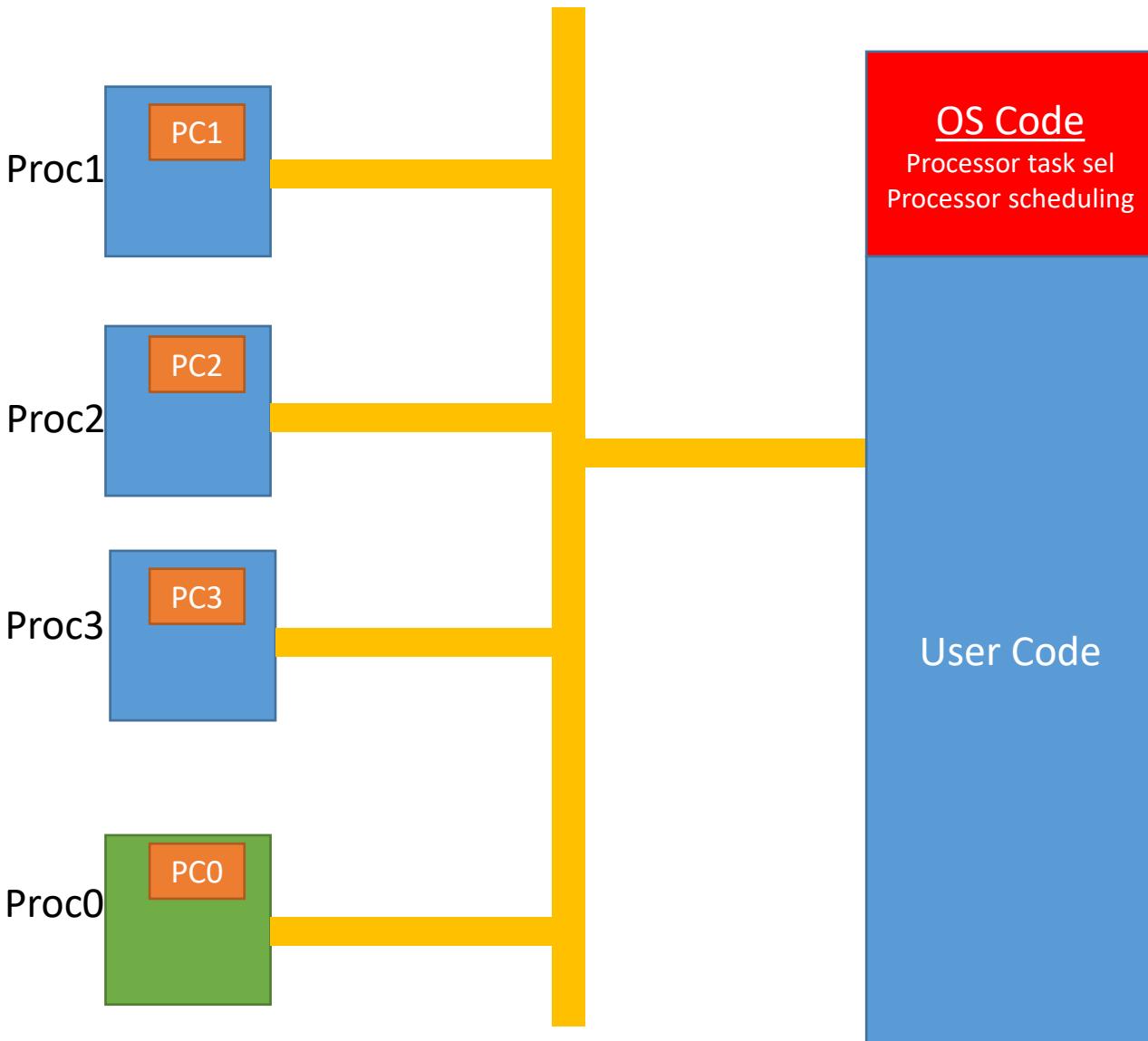
It also specifies other contents like Stack Pointer to ensure stacks for use during their respective function calls.

The OS containing processor also decides which processor will run next.

Such co-ordination is done by sending messages through the bus.

Note that there might be some common memory locations which may be used by more than 1 processor in parallel.

What is an SMP system?



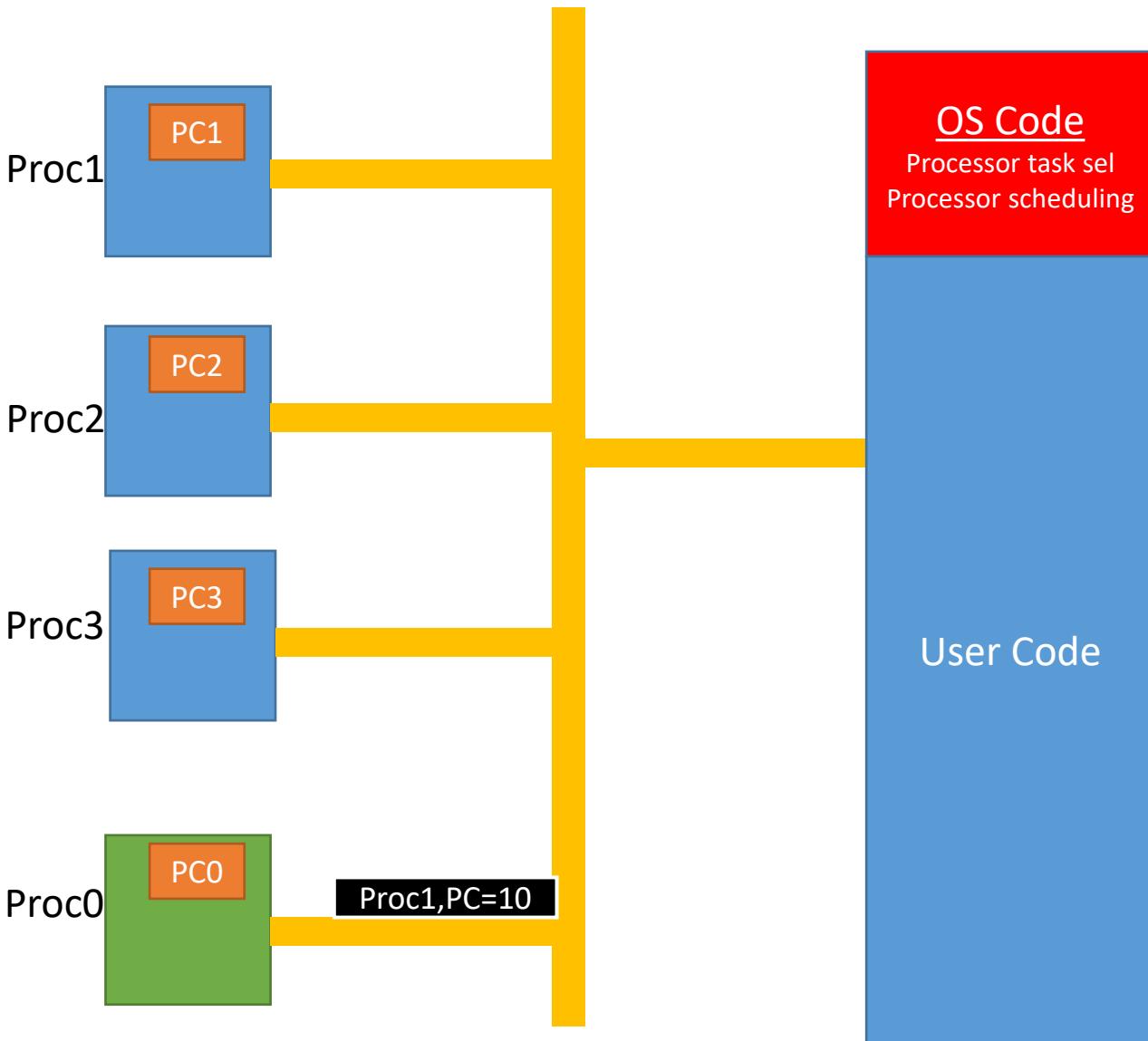
It also specifies other contents like Stack Pointer to ensure stacks for use during their respective function calls.

The OS containing processor also decides which processor will run next.

Such co-ordination is done by sending messages through the bus.

Note that there might be some common memory locations which may be used by more than 1 processor in parallel.

What is an SMP system?



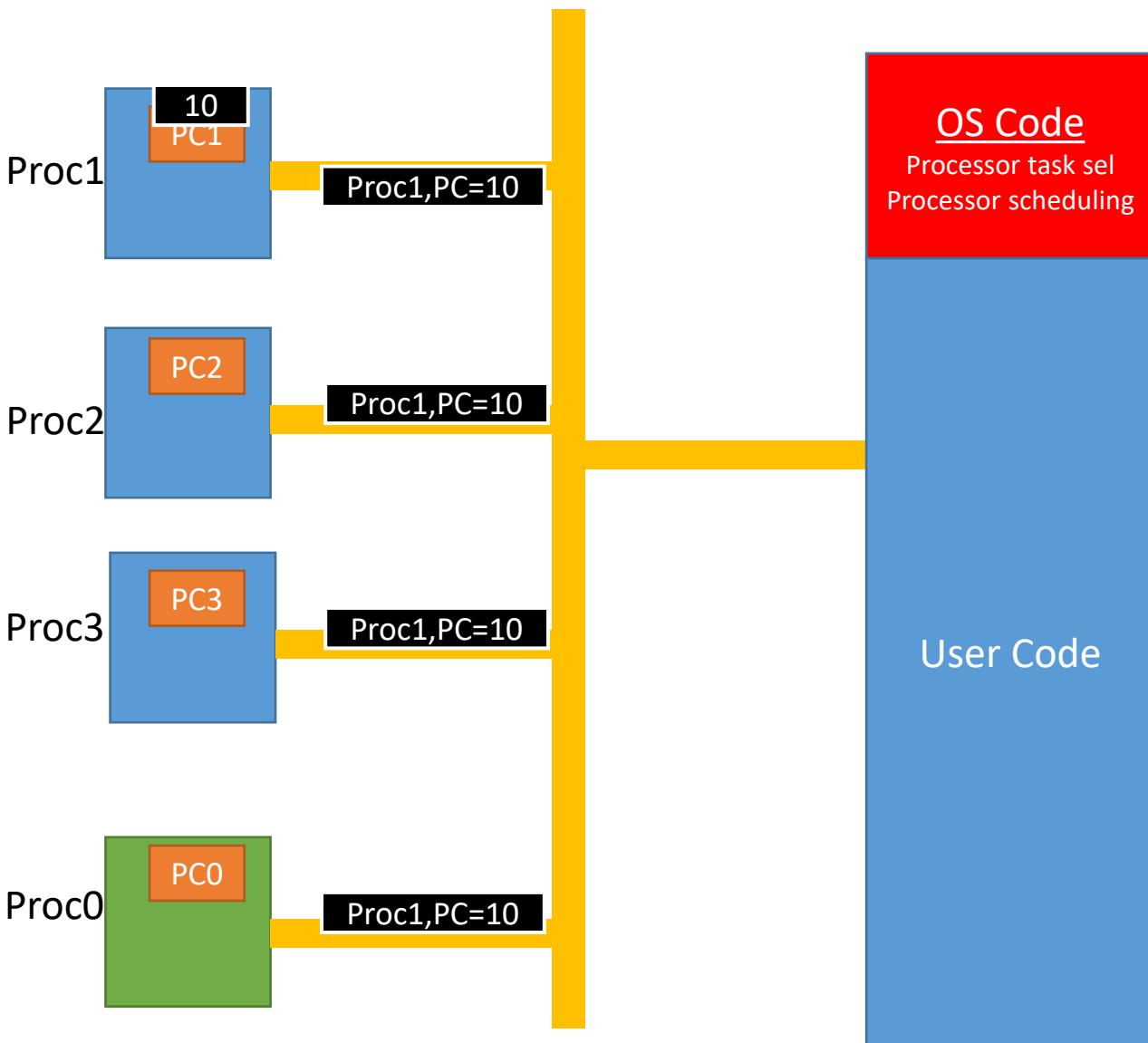
It also specifies other contents like Stack Pointer to ensure stacks for use during their respective function calls.

The OS containing processor also decides which processor will run next.

Such co-ordination is done by sending messages through the bus.

Note that there might be some common memory locations which may be used by more than 1 processor in parallel.

What is an SMP system?



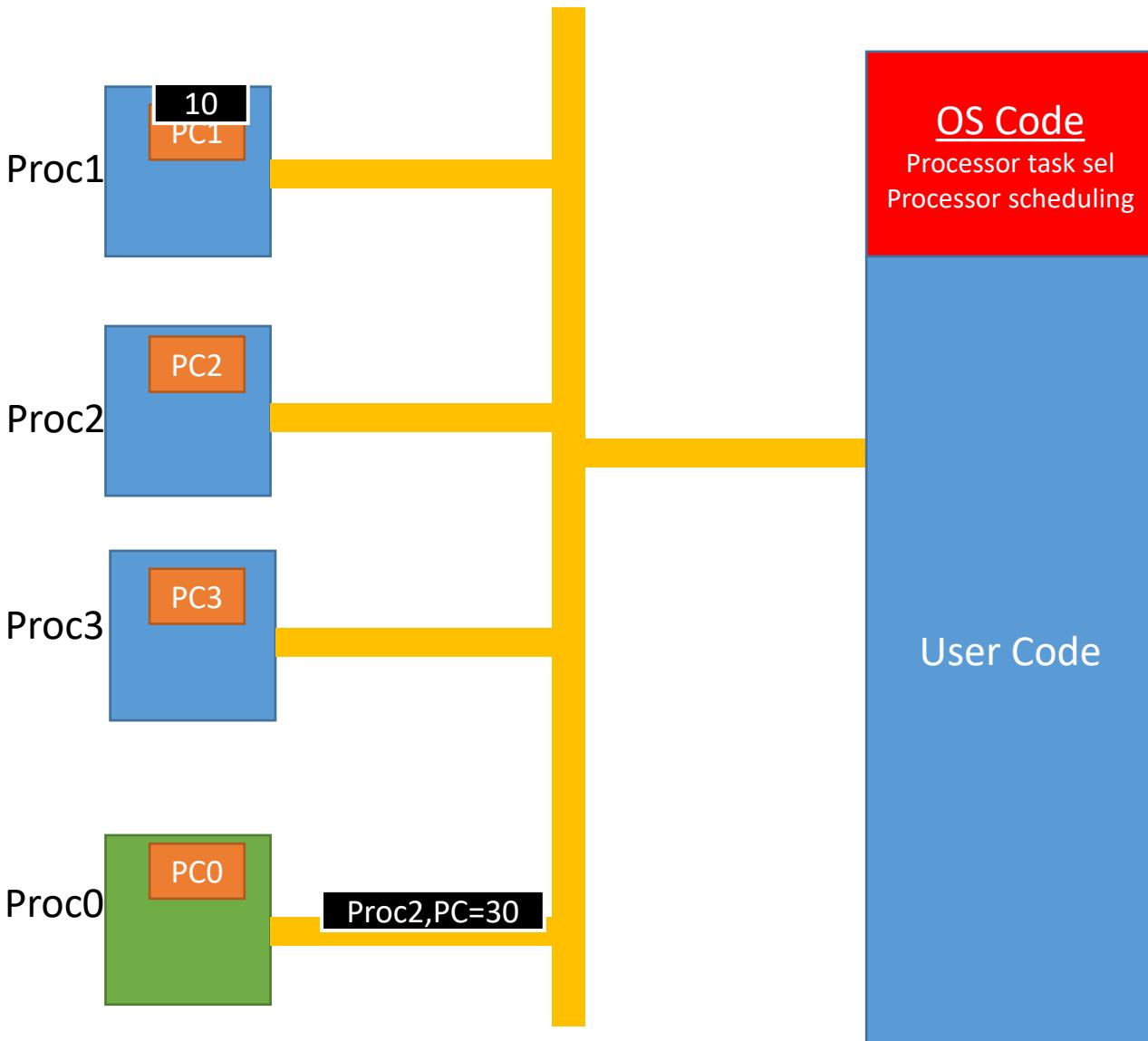
It also specifies other contents like Stack Pointer to ensure stacks for use during their respective function calls.

The OS containing processor also decides which processor will run next.

Such co-ordination is done by sending messages through the bus.

Note that there might be some common memory locations which may be used by more than 1 processor in parallel.

What is an SMP system?



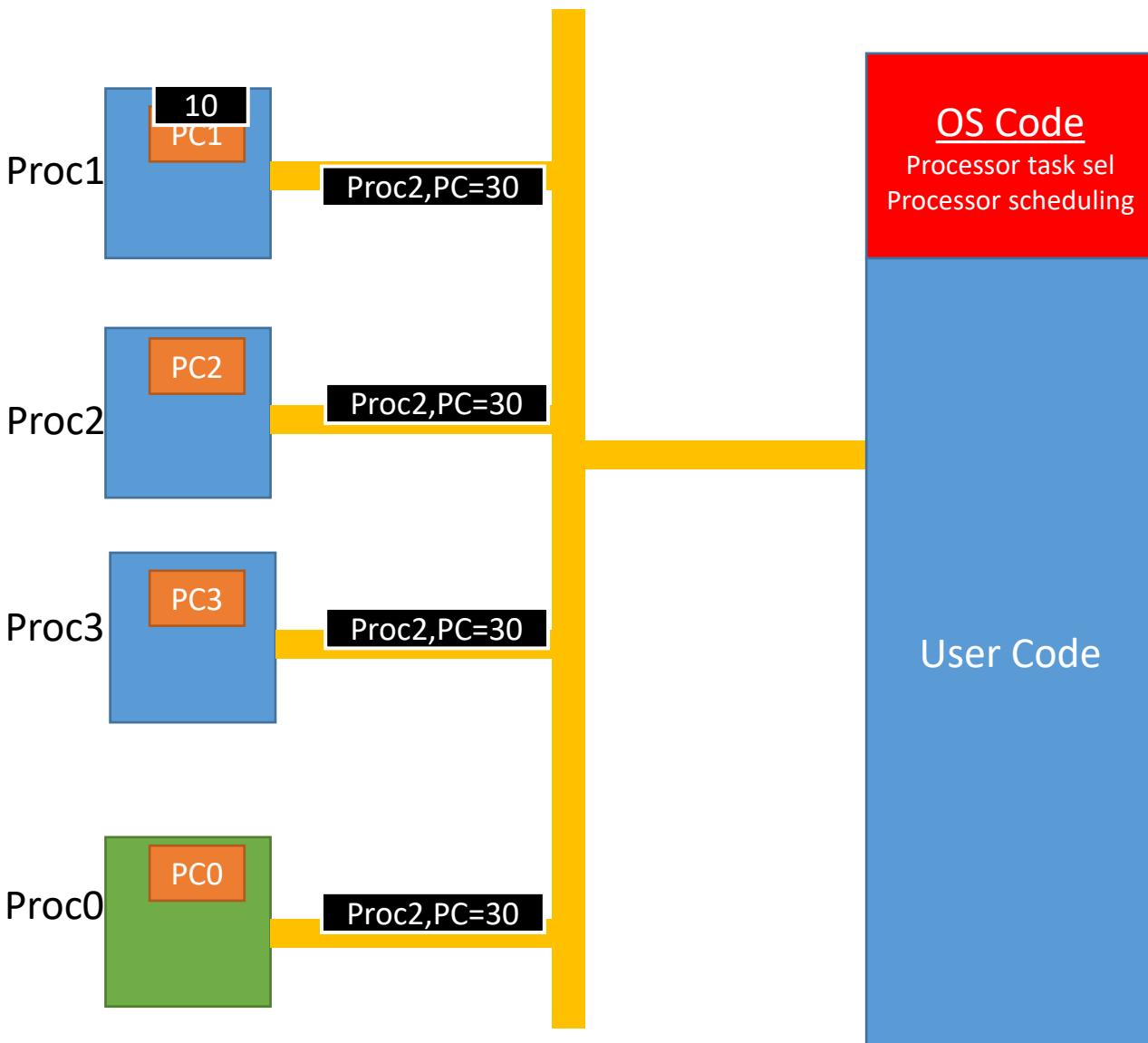
It also specifies other contents like Stack Pointer to ensure stacks for use during their respective function calls.

The OS containing processor also decides which processor will run next.

Such co-ordination is done by sending messages through the bus.

Note that there might be some common memory locations which may be used by more than 1 processor in parallel.

What is an SMP system?



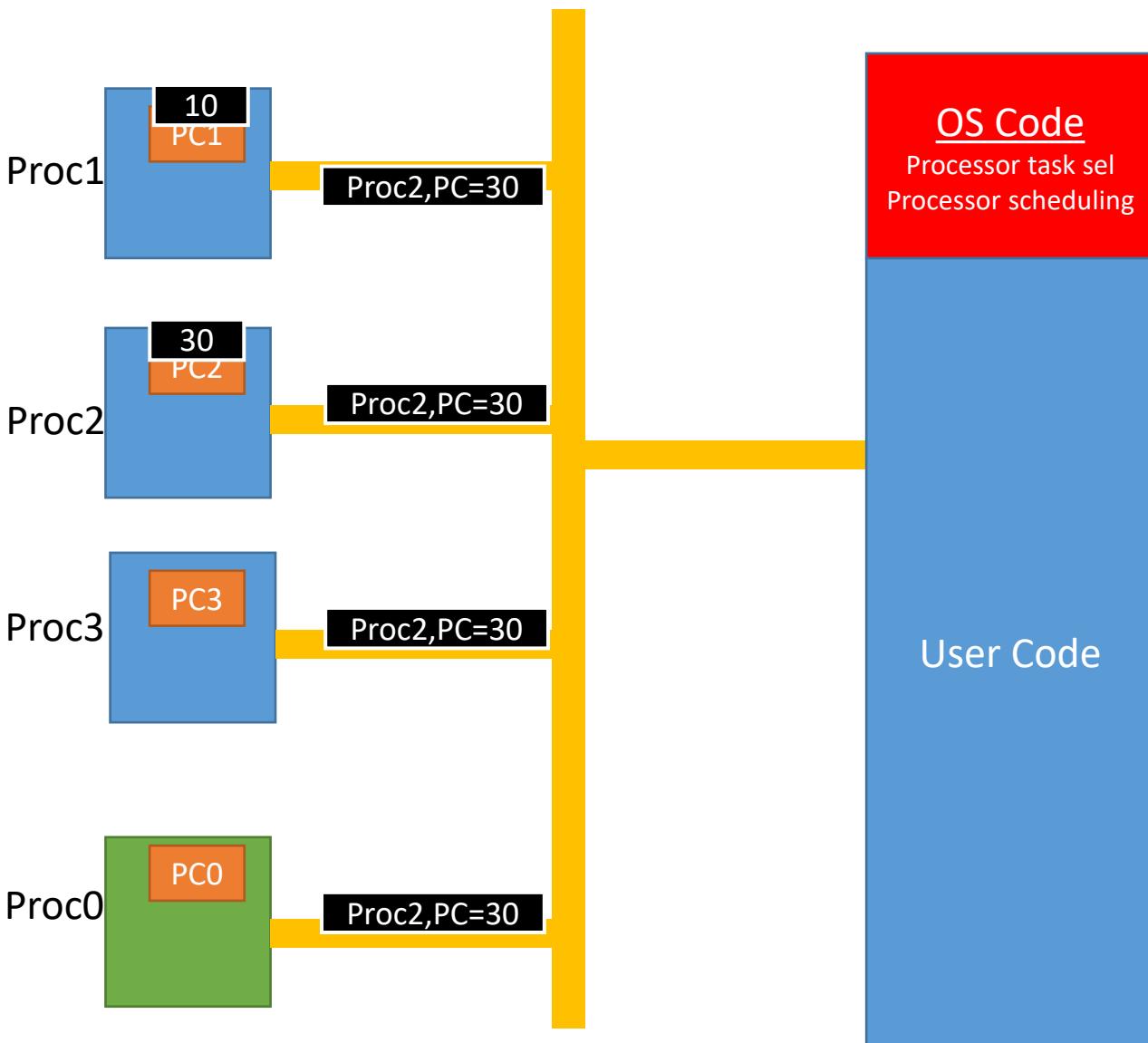
It also specifies other contents like Stack Pointer to ensure stacks for use during their respective function calls.

The OS containing processor also decides which processor will run next.

Such co-ordination is done by sending messages through the bus.

Note that there might be some common memory locations which may be used by more than 1 processor in parallel.

What is an SMP system?



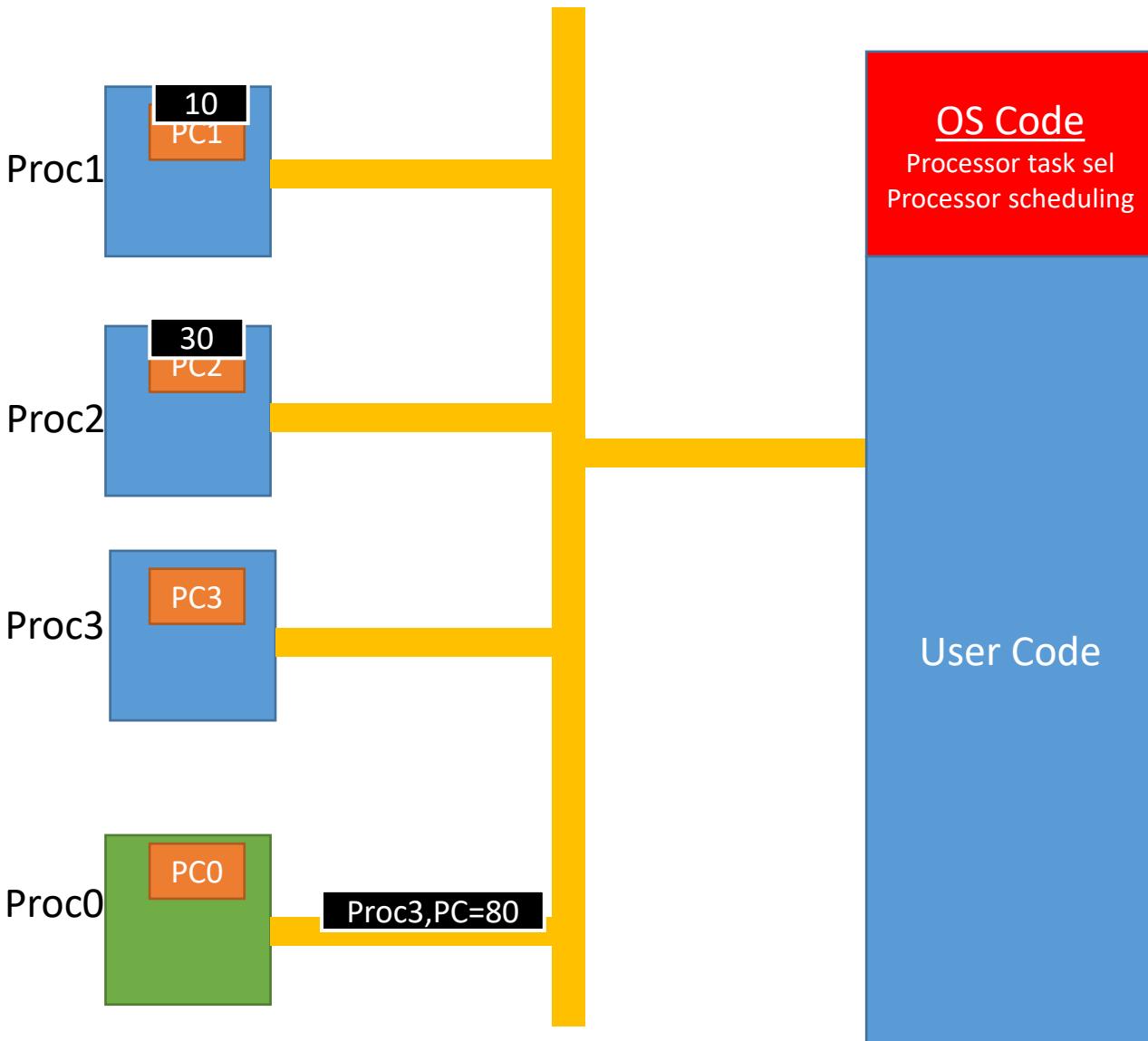
It also specifies other contents like Stack Pointer to ensure stacks for use during their respective function calls.

The OS containing processor also decides which processor will run next.

Such co-ordination is done by sending messages through the bus.

Note that there might be some common memory locations which may be used by more than 1 processor in parallel.

What is an SMP system?



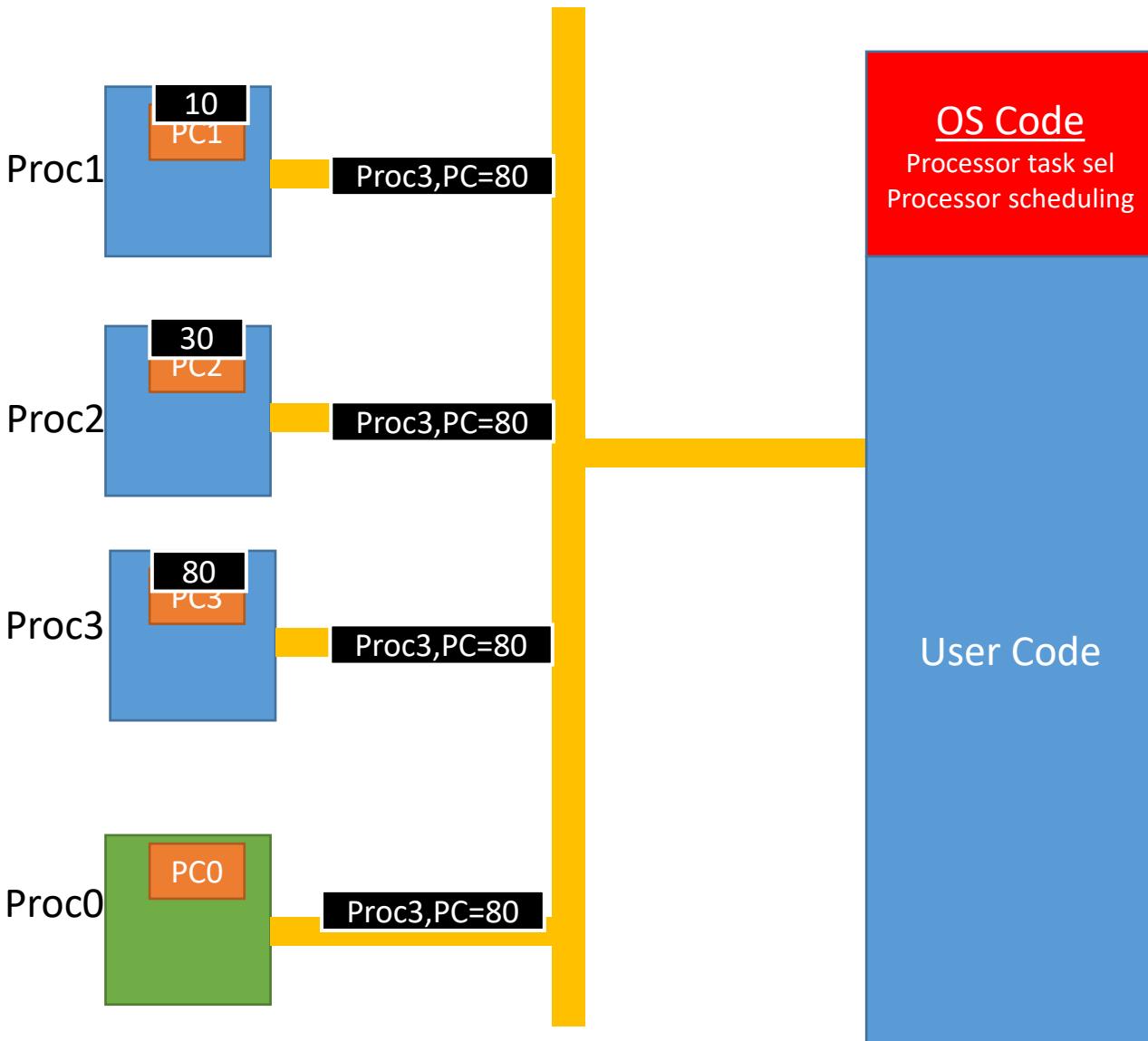
It also specifies other contents like Stack Pointer to ensure stacks for use during their respective function calls.

The OS containing processor also decides which processor will run next.

Such co-ordination is done by sending messages through the bus.

Note that there might be some common memory locations which may be used by more than 1 processor in parallel.

What is an SMP system?



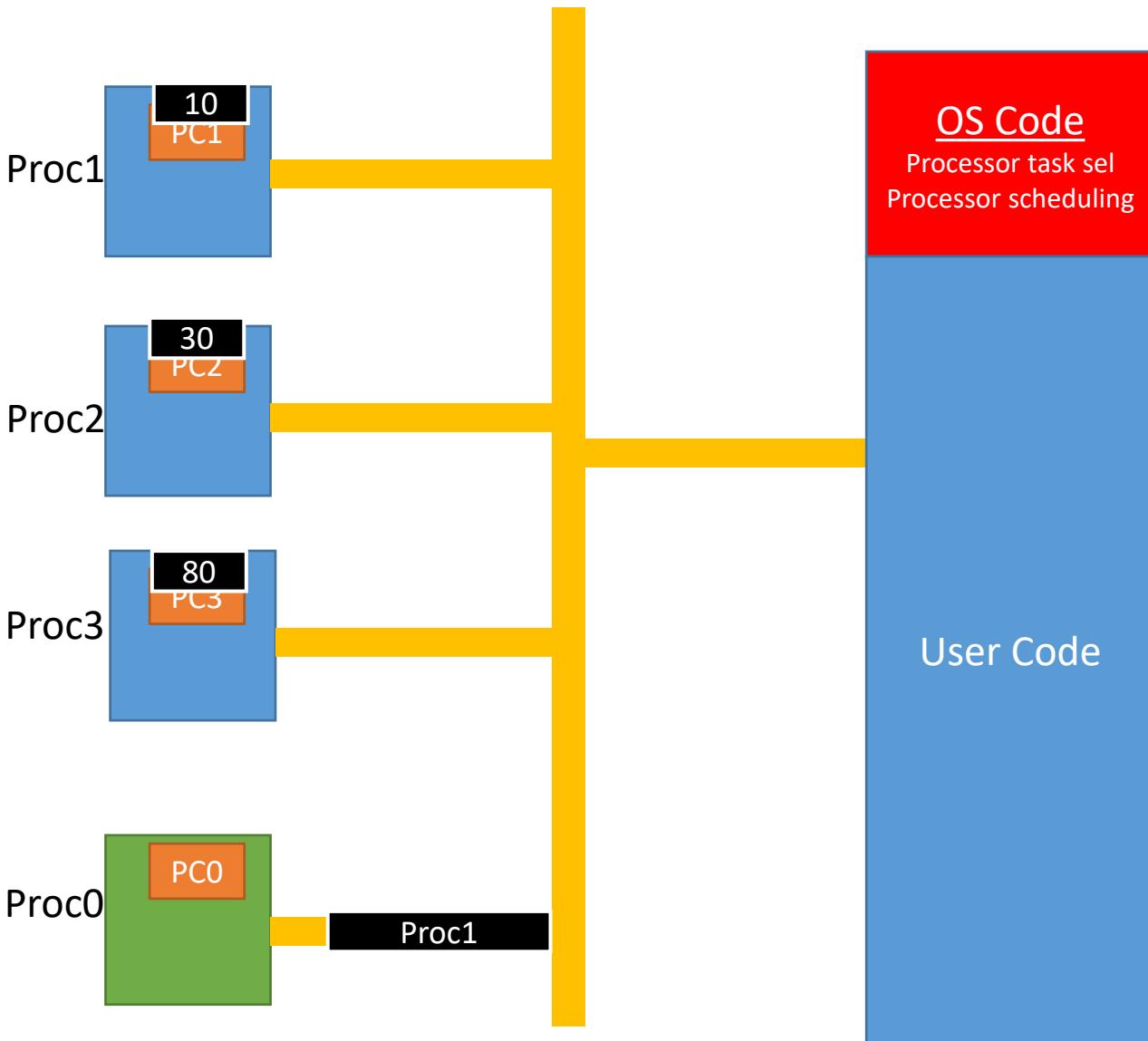
It also specifies other contents like Stack Pointer to ensure stacks for use during their respective function calls.

The OS containing processor also decides which processor will run next.

Such co-ordination is done by sending messages through the bus.

Note that there might be some common memory locations which may be used by more than 1 processor in parallel.

What is an SMP system?



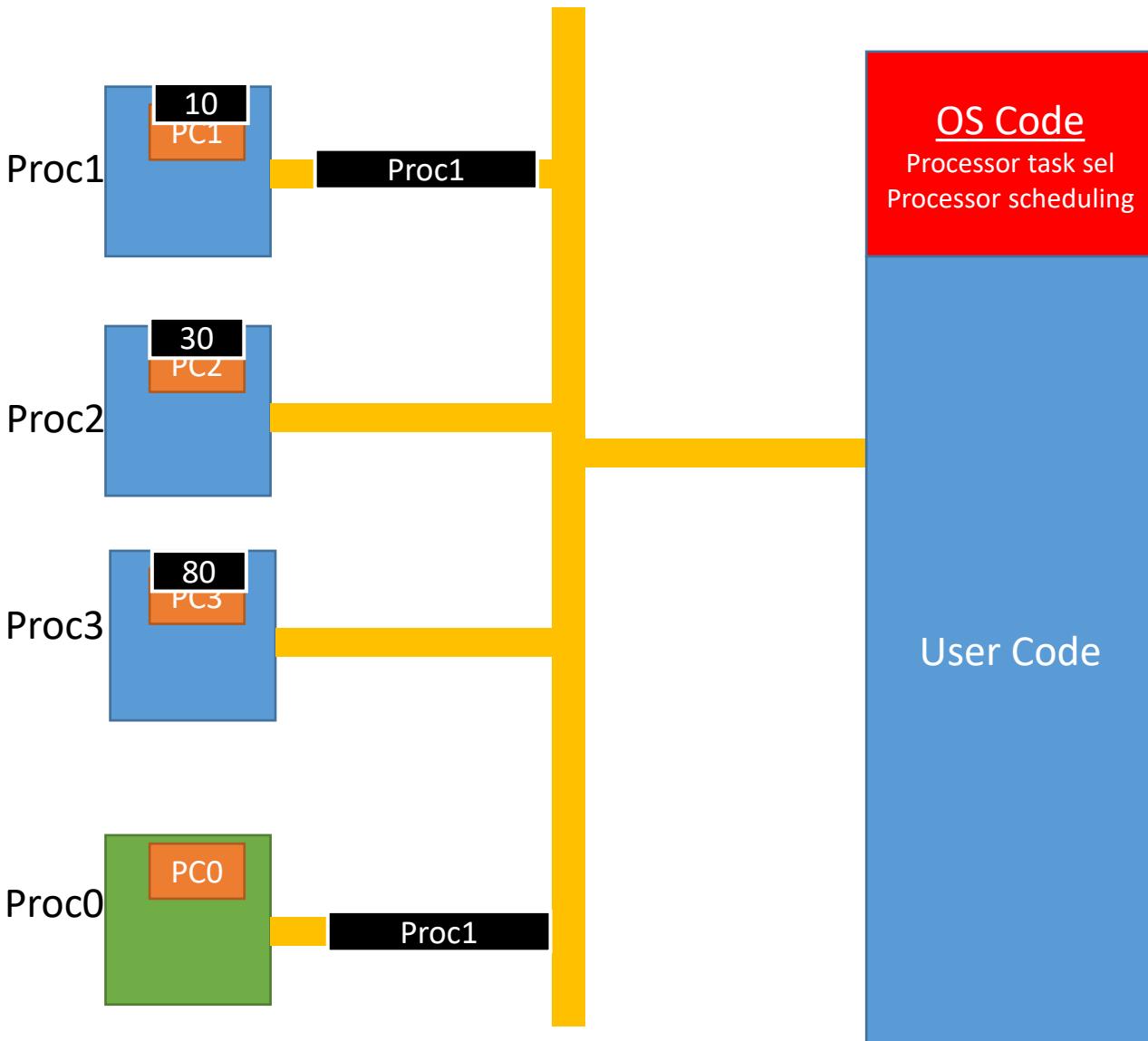
It also specifies other contents like Stack Pointer to ensure stacks for use during their respective function calls.

The OS containing processor also decides which processor will run next.

Such co-ordination is done by sending messages through the bus.

Note that there might be some common memory locations which may be used by more than 1 processor in parallel.

What is an SMP system?



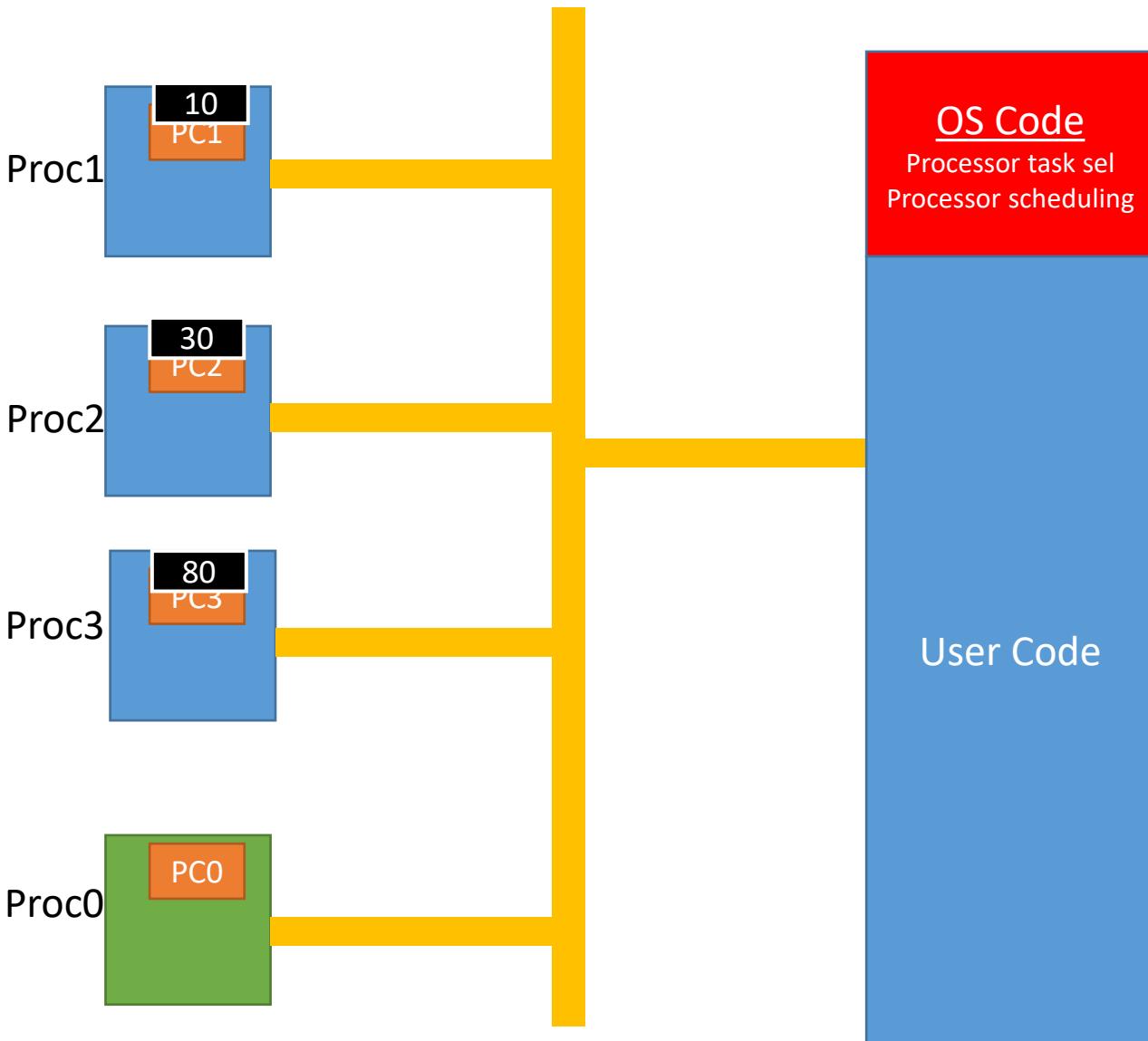
It also specifies other contents like Stack Pointer to ensure stacks for use during their respective function calls.

The OS containing processor also decides which processor will run next.

Such co-ordination is done by sending messages through the bus.

Note that there might be some common memory locations which may be used by more than 1 processor in parallel.

What is an SMP system?



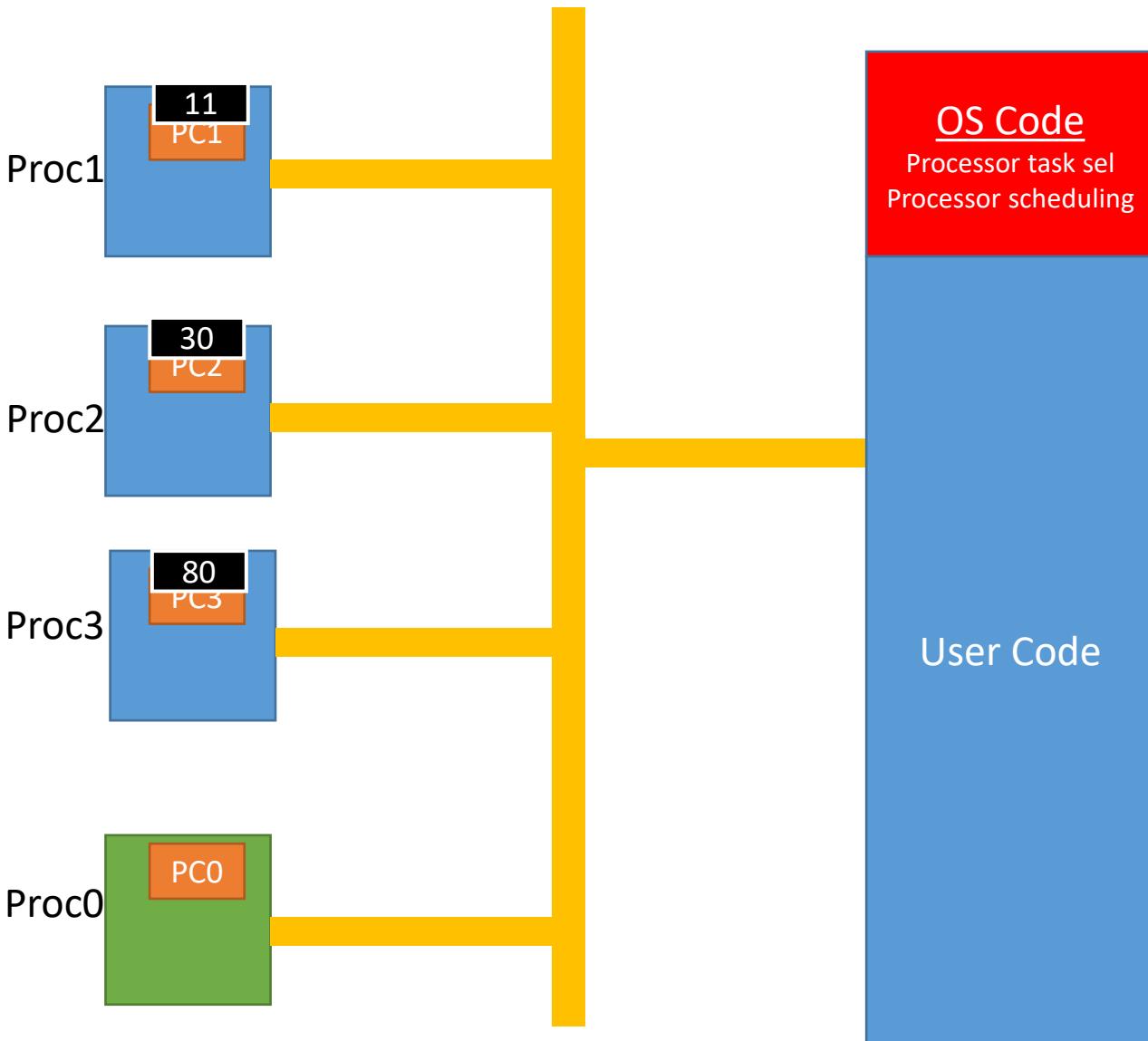
It also specifies other contents like Stack Pointer to ensure stacks for use during their respective function calls.

The OS containing processor also decides which processor will run next.

Such co-ordination is done by sending messages through the bus.

Note that there might be some common memory locations which may be used by more than 1 processor in parallel.

What is an SMP system?



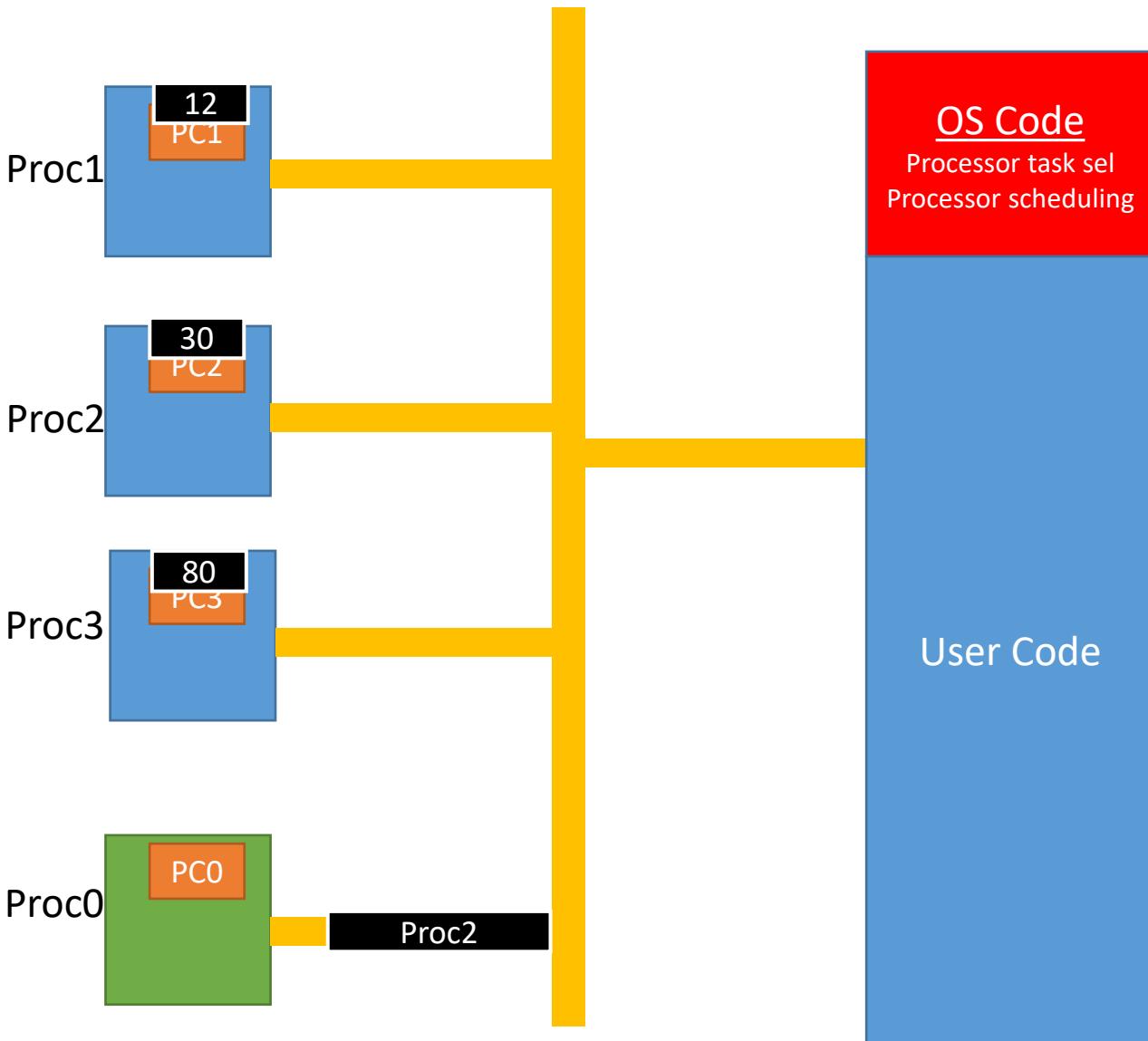
It also specifies other contents like Stack Pointer to ensure stacks for use during their respective function calls.

The OS containing processor also decides which processor will run next.

Such co-ordination is done by sending messages through the bus.

Note that there might be some common memory locations which may be used by more than 1 processor in parallel.

What is an SMP system?



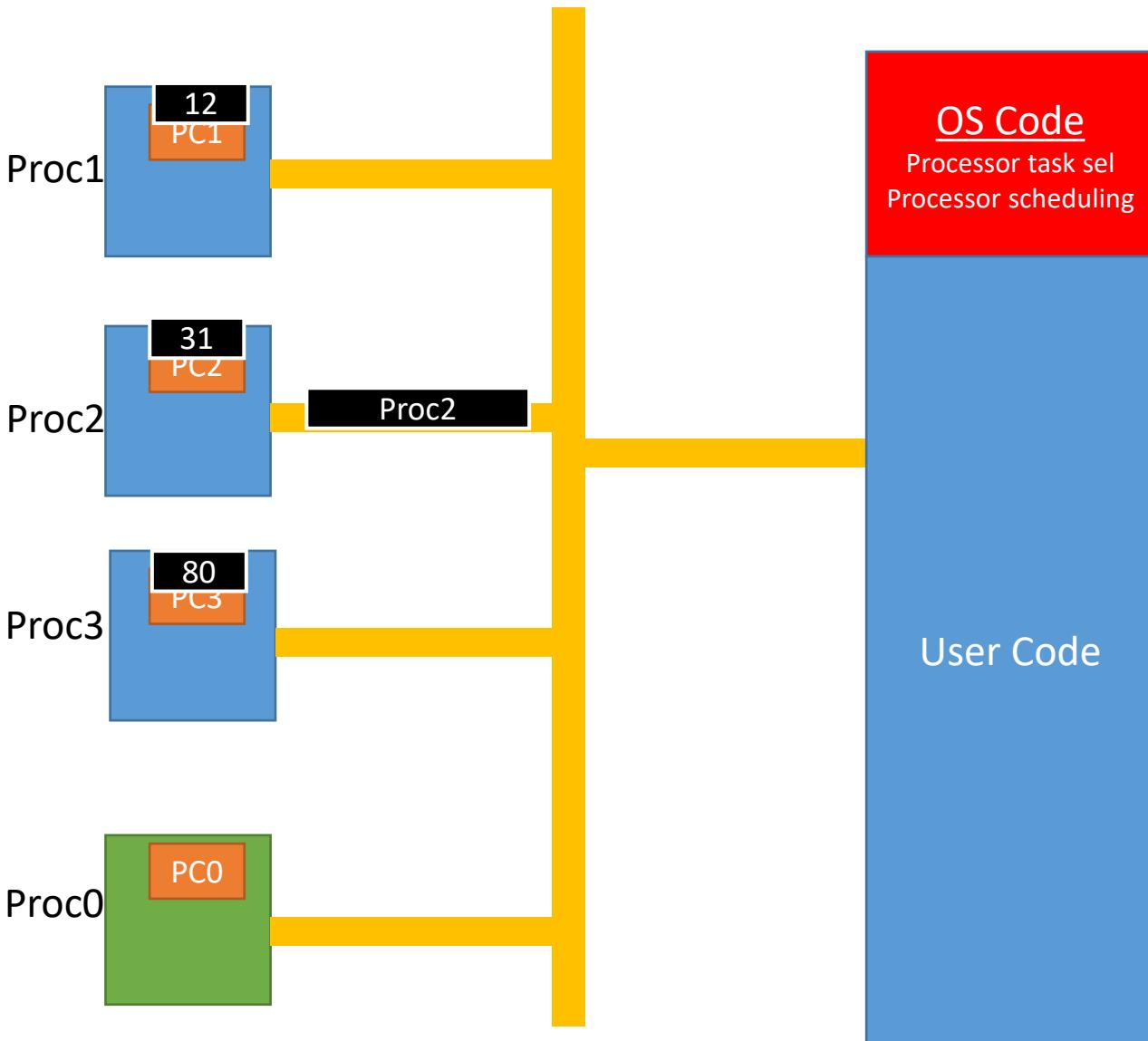
It also specifies other contents like Stack Pointer to ensure stacks for use during their respective function calls.

The OS containing processor also decides which processor will run next.

Such co-ordination is done by sending messages through the bus.

Note that there might be some common memory locations which may be used by more than 1 processor in parallel.

What is an SMP system?



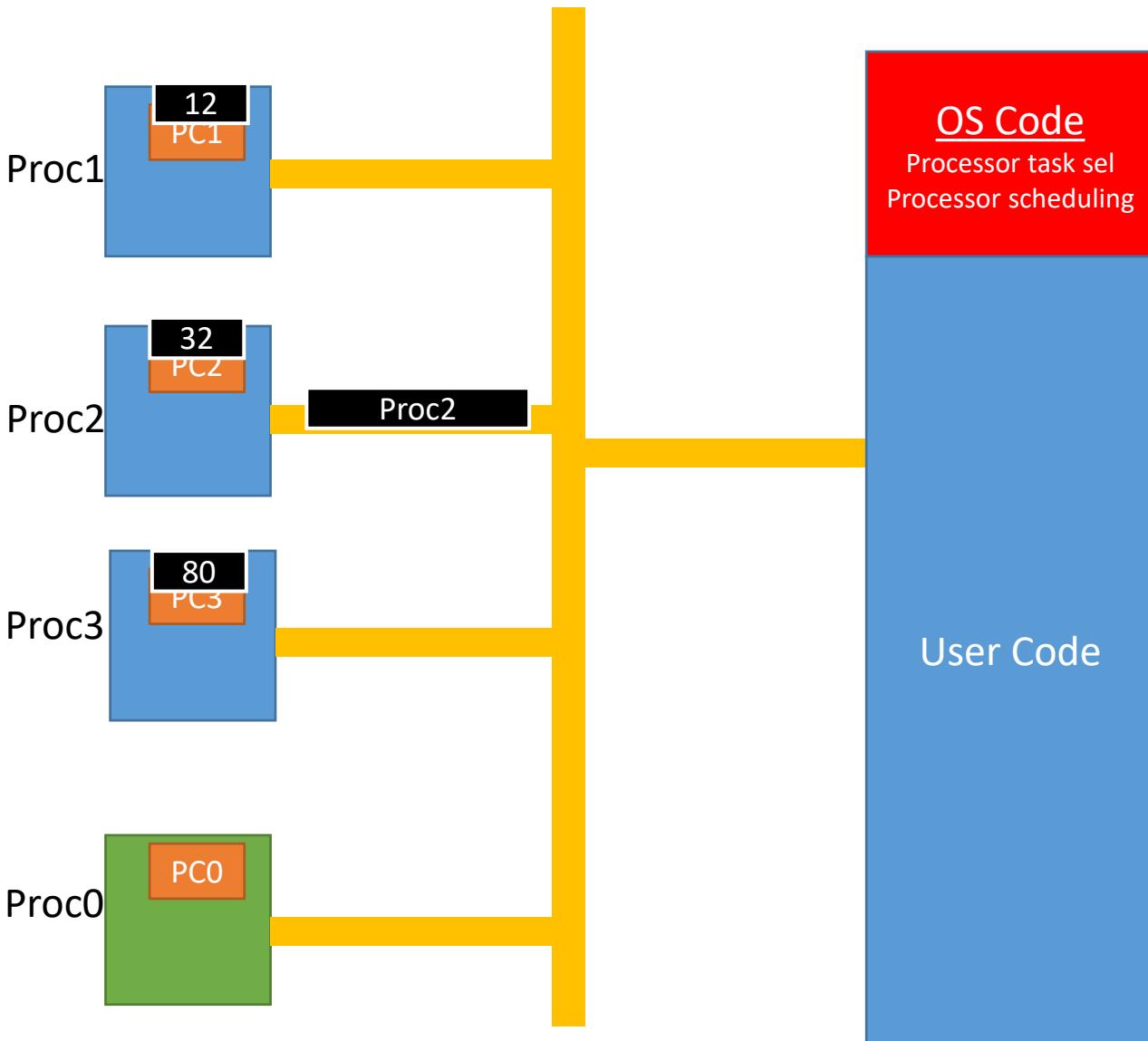
It also specifies other contents like Stack Pointer to ensure stacks for use during their respective function calls.

The OS containing processor also decides which processor will run next.

Such co-ordination is done by sending messages through the bus.

Note that there might be some common memory locations which may be used by more than 1 processor in parallel.

What is an SMP system?



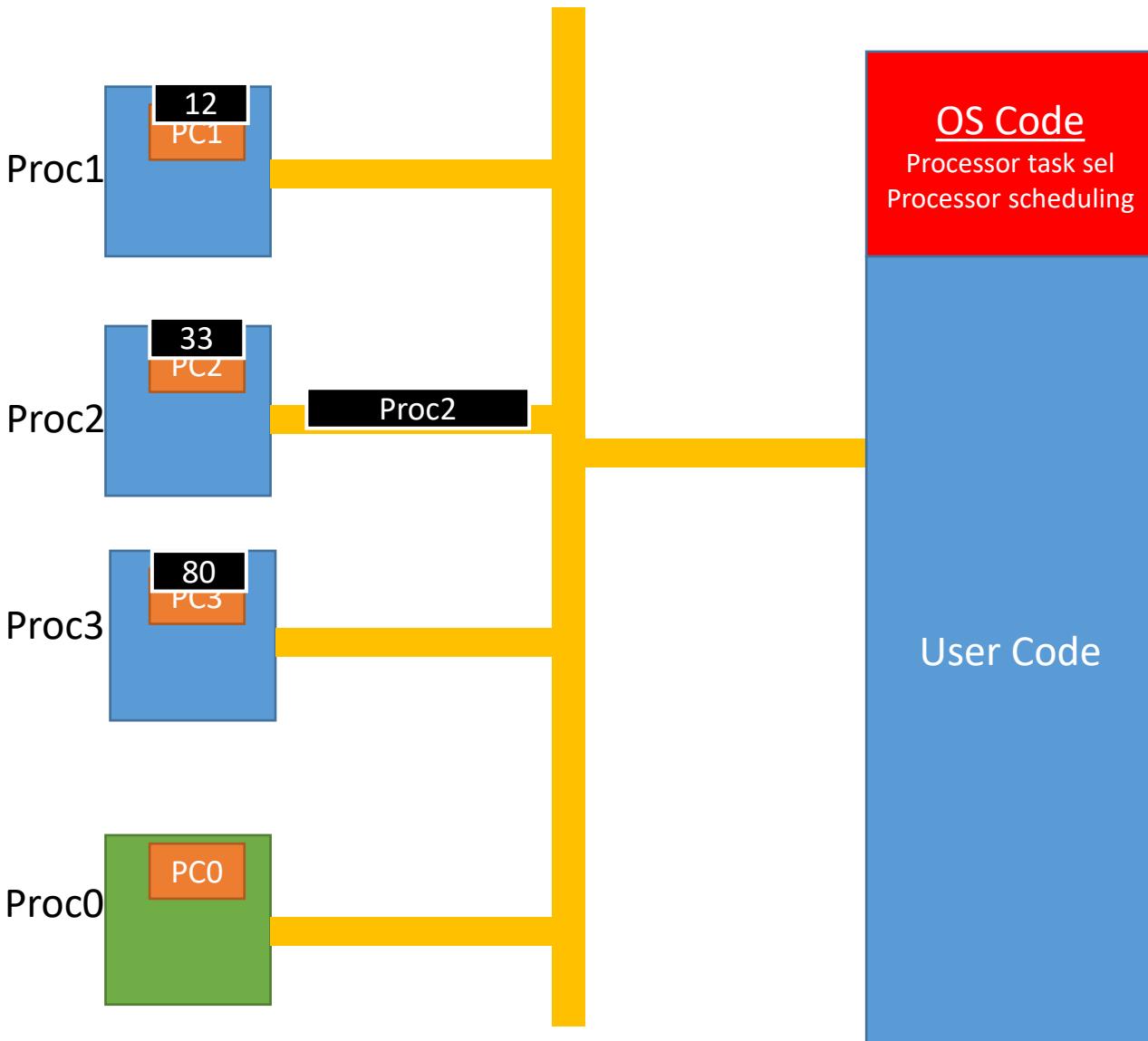
It also specifies other contents like Stack Pointer to ensure stacks for use during their respective function calls.

The OS containing processor also decides which processor will run next.

Such co-ordination is done by sending messages through the bus.

Note that there might be some common memory locations which may be used by more than 1 processor in parallel.

What is an SMP system?



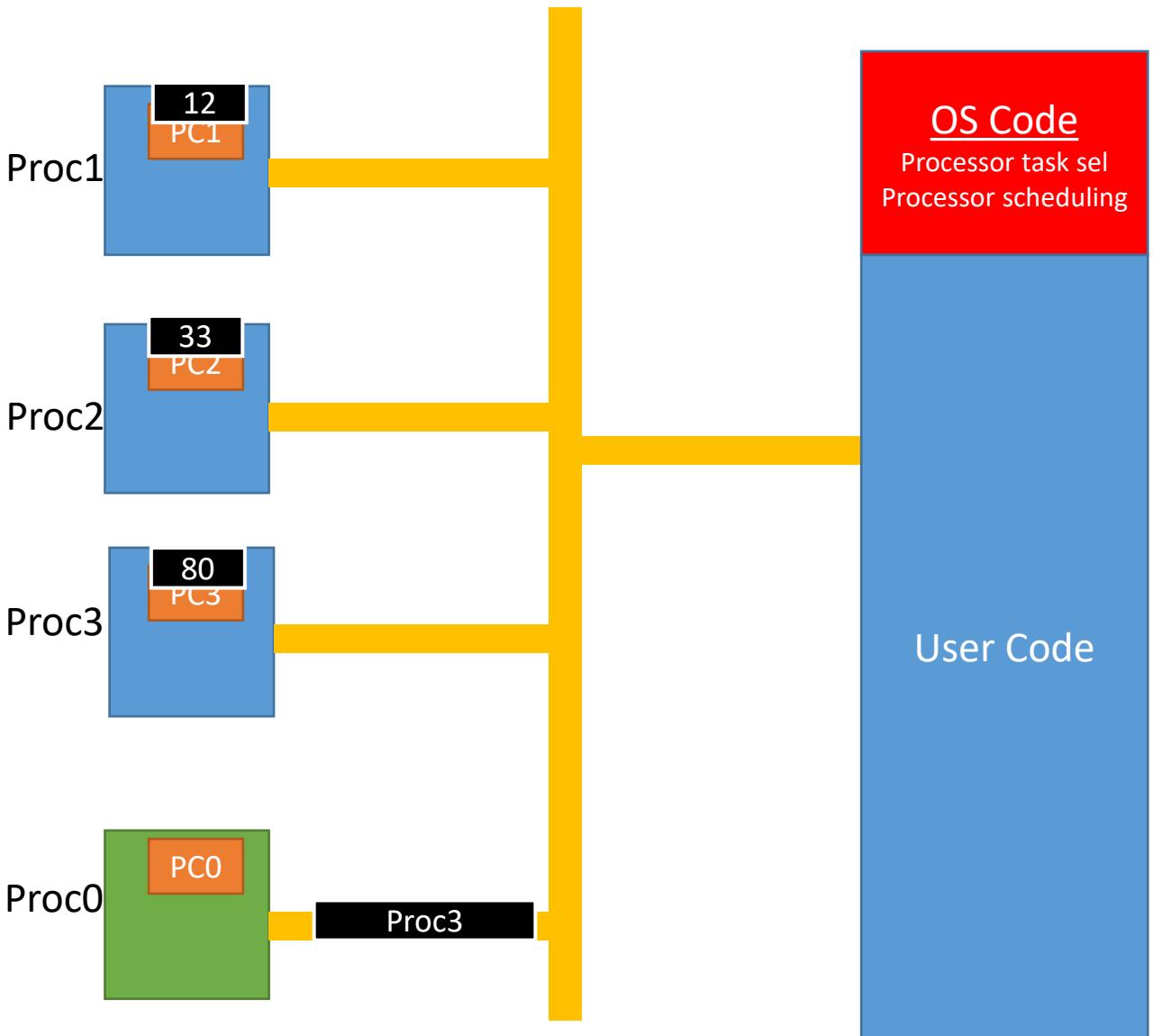
It also specifies other contents like Stack Pointer to ensure stacks for use during their respective function calls.

The OS containing processor also decides which processor will run next.

Such co-ordination is done by sending messages through the bus.

Note that there might be some common memory locations which may be used by more than 1 processor in parallel.

What is an SMP system?



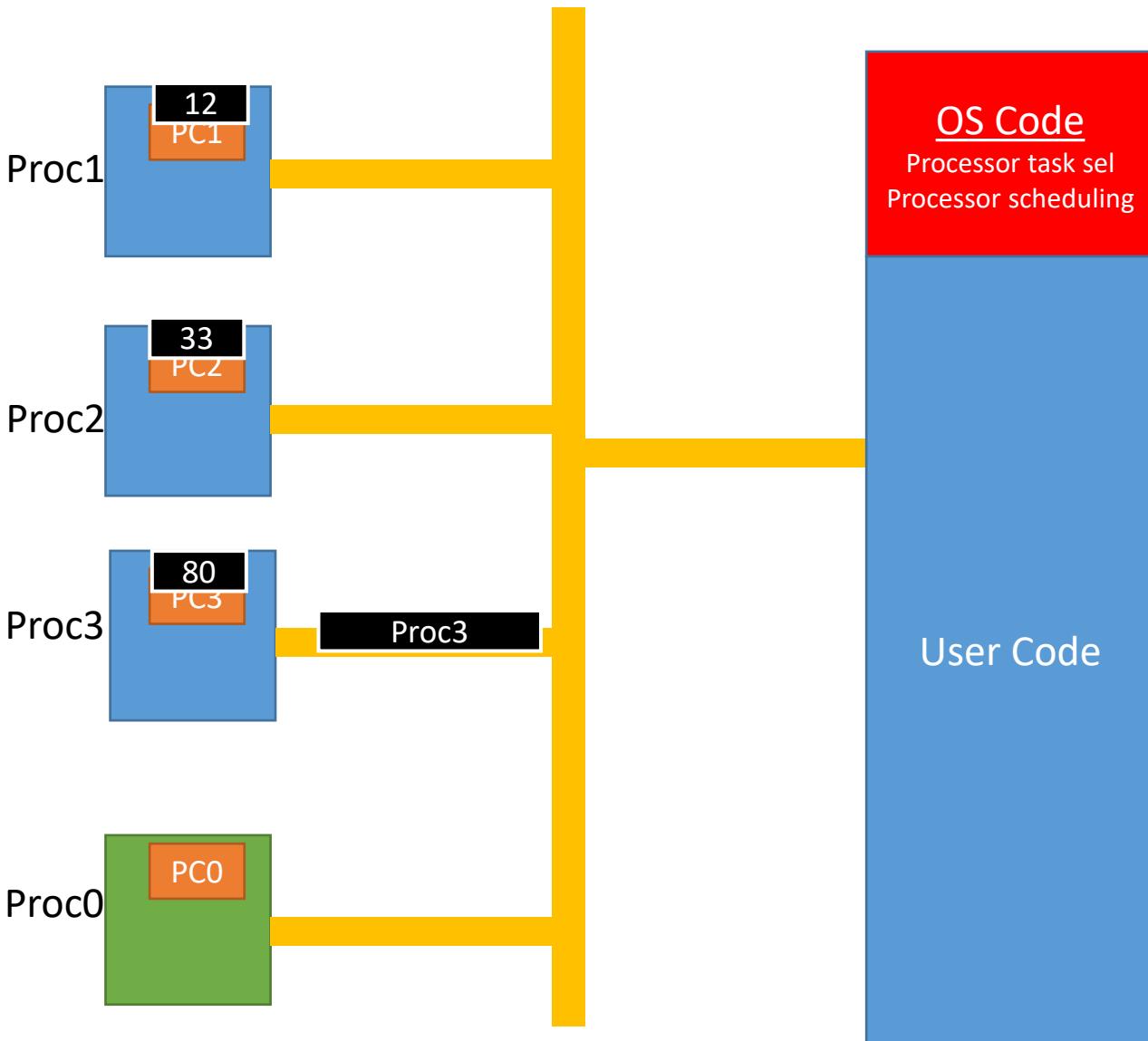
It also specifies other contents like Stack Pointer to ensure stacks for use during their respective function calls.

The OS containing processor also decides which processor will run next.

Such co-ordination is done by sending messages through the bus.

Note that there might be some common memory locations which may be used by more than 1 processor in parallel.

What is an SMP system?



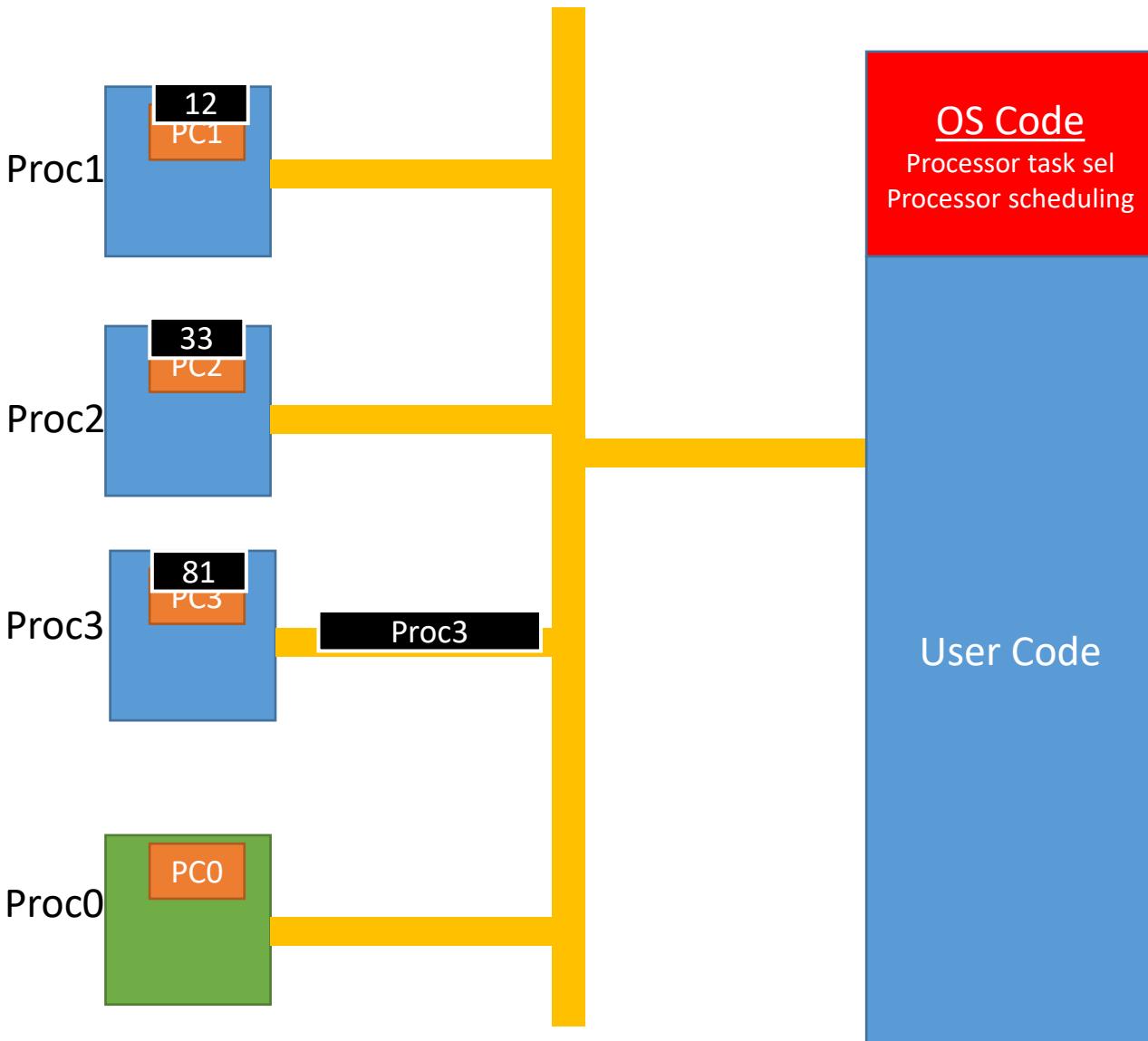
It also specifies other contents like Stack Pointer to ensure stacks for use during their respective function calls.

The OS containing processor also decides which processor will run next.

Such co-ordination is done by sending messages through the bus.

Note that there might be some common memory locations which may be used by more than 1 processor in parallel.

What is an SMP system?



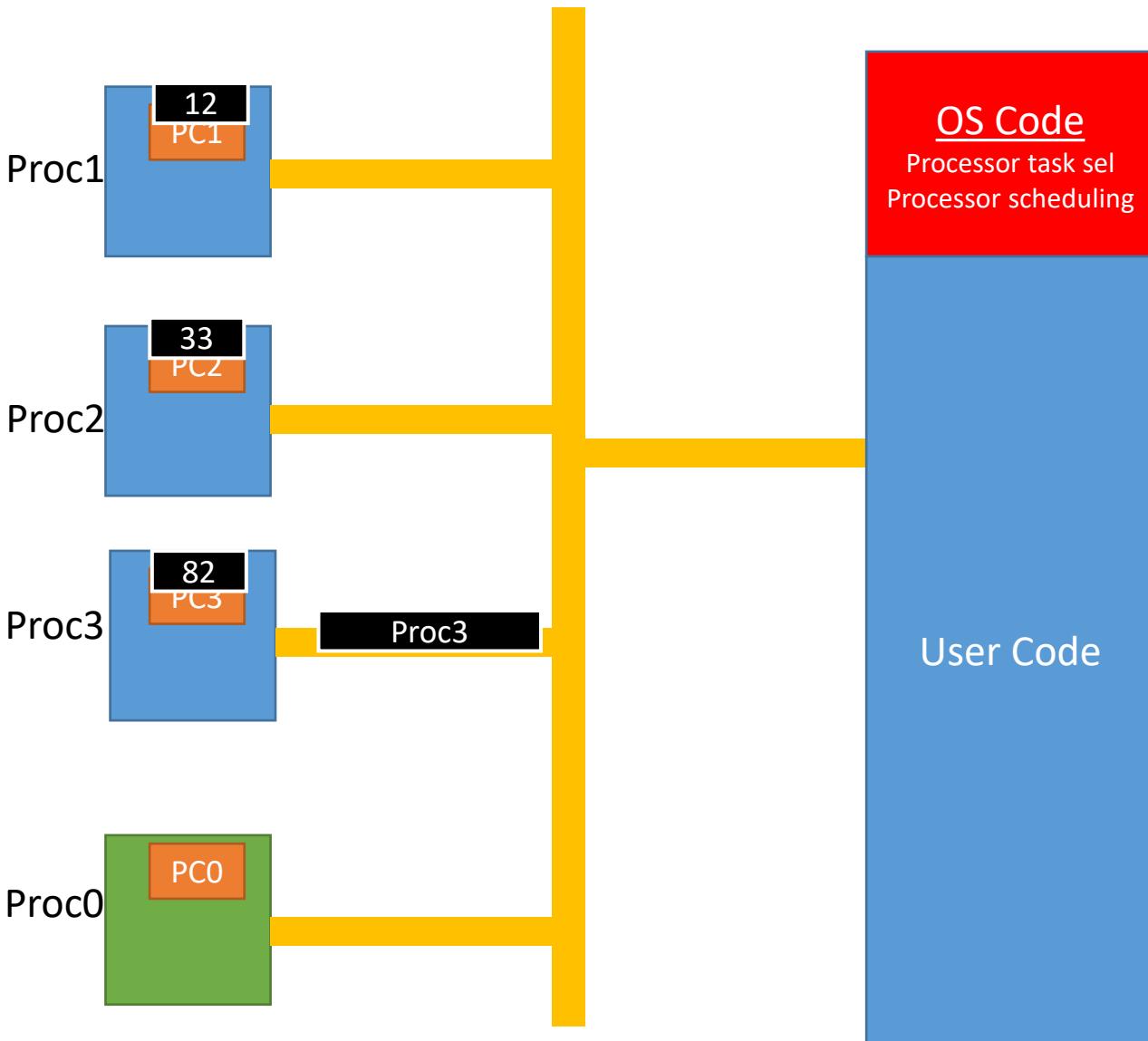
It also specifies other contents like Stack Pointer to ensure stacks for use during their respective function calls.

The OS containing processor also decides which processor will run next.

Such co-ordination is done by sending messages through the bus.

Note that there might be some common memory locations which may be used by more than 1 processor in parallel.

What is an SMP system?



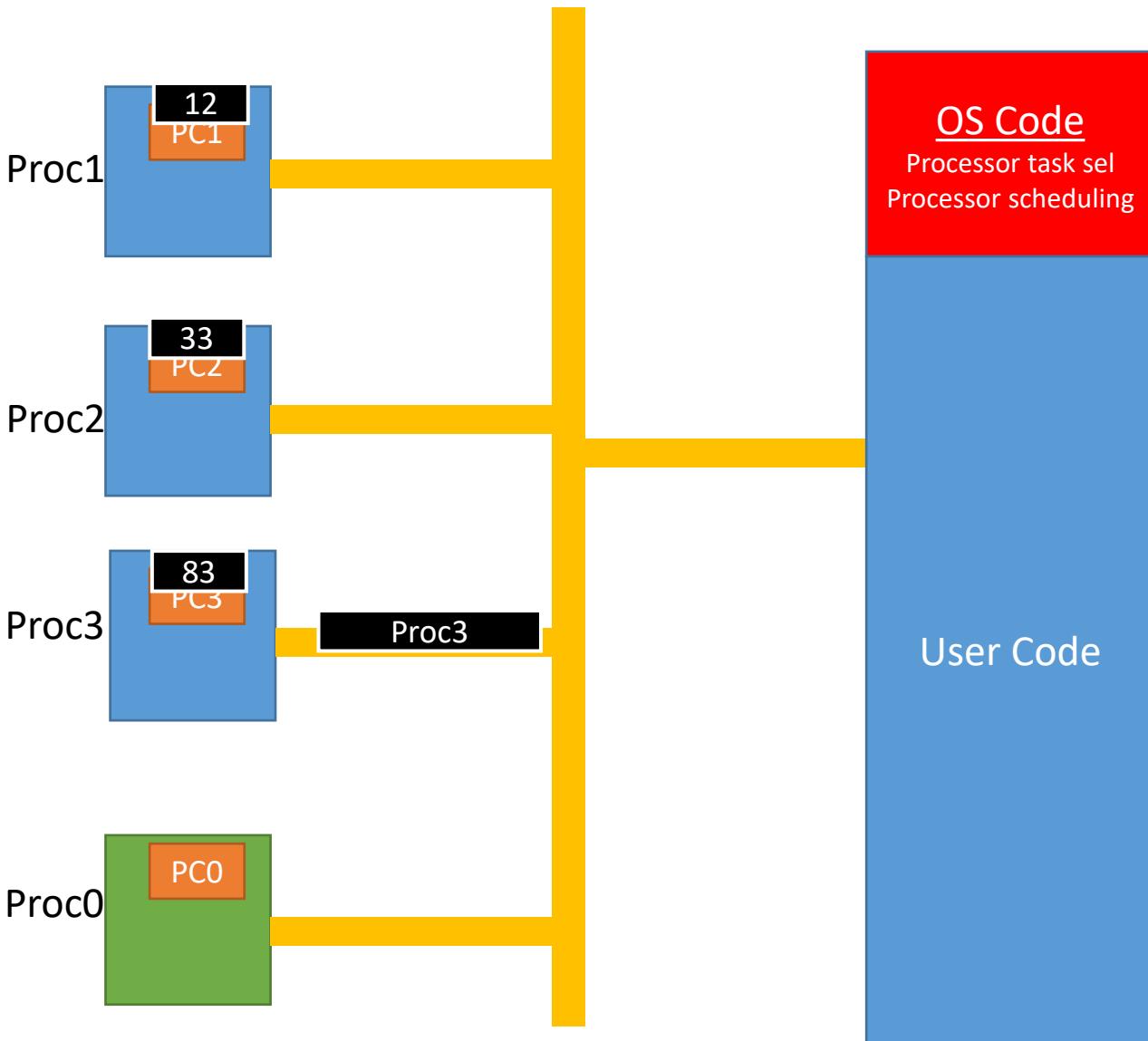
It also specifies other contents like Stack Pointer to ensure stacks for use during their respective function calls.

The OS containing processor also decides which processor will run next.

Such co-ordination is done by sending messages through the bus.

Note that there might be some common memory locations which may be used by more than 1 processor in parallel.

What is an SMP system?



It also specifies other contents like Stack Pointer to ensure stacks for use during their respective function calls.

The OS containing processor also decides which processor will run next.

Such co-ordination is done by sending messages through the bus.

Note that there might be some common memory locations which may be used by more than 1 processor in parallel.

Advantages of SMP system

- **Performance enhancement** - this is due to parallel processing
- **Availability** - If one processor is damaged, another one can take its place.
- **Incremental Growth** - additional processor -> performance enhancement
- **Scaling** - Different versions based on different number of processors

2. The Components of SMP systems

1. SMP System: Communication and Organization

- Processors can **signal each other** – with the help of bus
- Processors can **access** blocks of **memory** directly.
- Each processor also may have its **own private memory**.

2. SMP System Bus :

Characteristics of the bus that we will be using:

- **Time sharing:** Each processor is allowed its own time
- **Arbitration :** When a processor is allowed its own time, it will take entire control of the bus
- **Addressing :** The bus will carry control signals, data and addresses.

2. SMP System Bus organization Advantages

- **Simplicity**
- **Flexibility** - Easy to expand it or shrink it
- **Reliability** - if one processor fails, the system won't break down

2. SMP System Bus organization Disadvantages

- **Bus breakdown** – The entire system will collapse
- **Bus speed limit** – If high speed processors are used, there performance will be limited by the bus speed
- **Cache coherence** - There occurs issues with keeping the data in the caches consistent with the memory.

3. SMP System: Multiprocessor Operating System

Characteristics of the OS running in one of the processors:

- **Simultaneous concurrent processes:** Several processors running in parallel.
- **Scheduling:** Deciding which processor will run which process.
- **Synchronization:** Ensuring that no conflicts occur.
- **Memory management:** Memory allocation.
- **Reliability and Fault Tolerance** - Error handling is to be ensured

3. Ensuring consistency of data

Cache Coherence and the MESI protocol

Cache Coherence and the MESI protocol

- **Cache coherence** : the image in memory and the image in the cache are different.
- Two common **write policies** (to make the cache image same as the memory image):
 - **Write back:**
 - **Write through:**

Cache Coherence and the MESI protocol

Write back:

- Write operations are made only to the **cache**
- Main memory updated when a line is **evicted**.
- Or when the program is **terminated**

Write through:

- Write operations are made to the **cache** as well as the **main memory**

The **write back** operation may result in inconsistency.

But with **write through** inconsistency occurs as well as other caches may not be updated.

Which one should be used?

Basic approach:

- Unshared variables –
 - can be pushed into the memory later on
- Shared variables -
 - Needs to be written and read from memory.
 - Also one processor may send signals to communicate with another processor

A possible solutions for this:

Software Solution for dealing with Cache Coherence

- Compiler based coherence mechanism
- Code **analysis**
- Sees which variables are **unsafe(shared)** for caching - marks them
- Prevents the caching of the **marked** variable
- However this may result in inefficiencies as sometimes the shared variables may be **read only**
- More efficient approach - **mark the periods** when the variables are not read only, i.e **unsafe**.

The Hardware Approach - The MESI protocol

The caches of each processor are modified.

Each line in the cache has two memories for **two status bits**.

They depict 4 different states at a time:

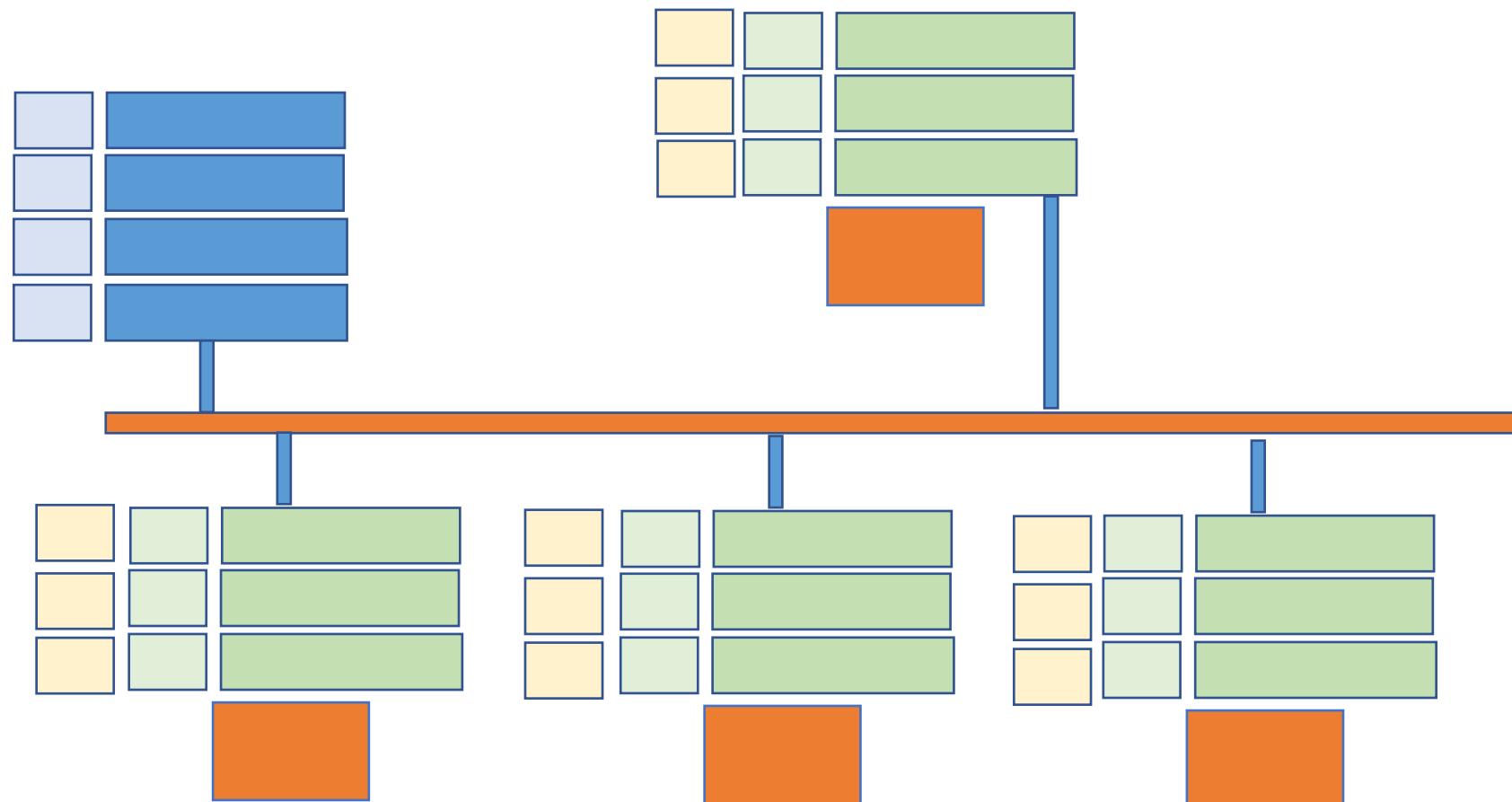
- **Modified** - This cache line has been modified (self modification)
- **Exclusive** - Only this cache has this line
- **Shared** - This line is in the main memory and may be present in another cache
- **Invalid** - This data is not valid

The Hardware Approach - The MESI protocol

MESI protocol and 4 scenarios:

1. READ HIT
2. READ MISS
3. WRITE HIT
4. WRITE MISS

Simulation Read Miss



Read Miss

Processor tries to read a data from an address. That data is missing in the cache/invalid

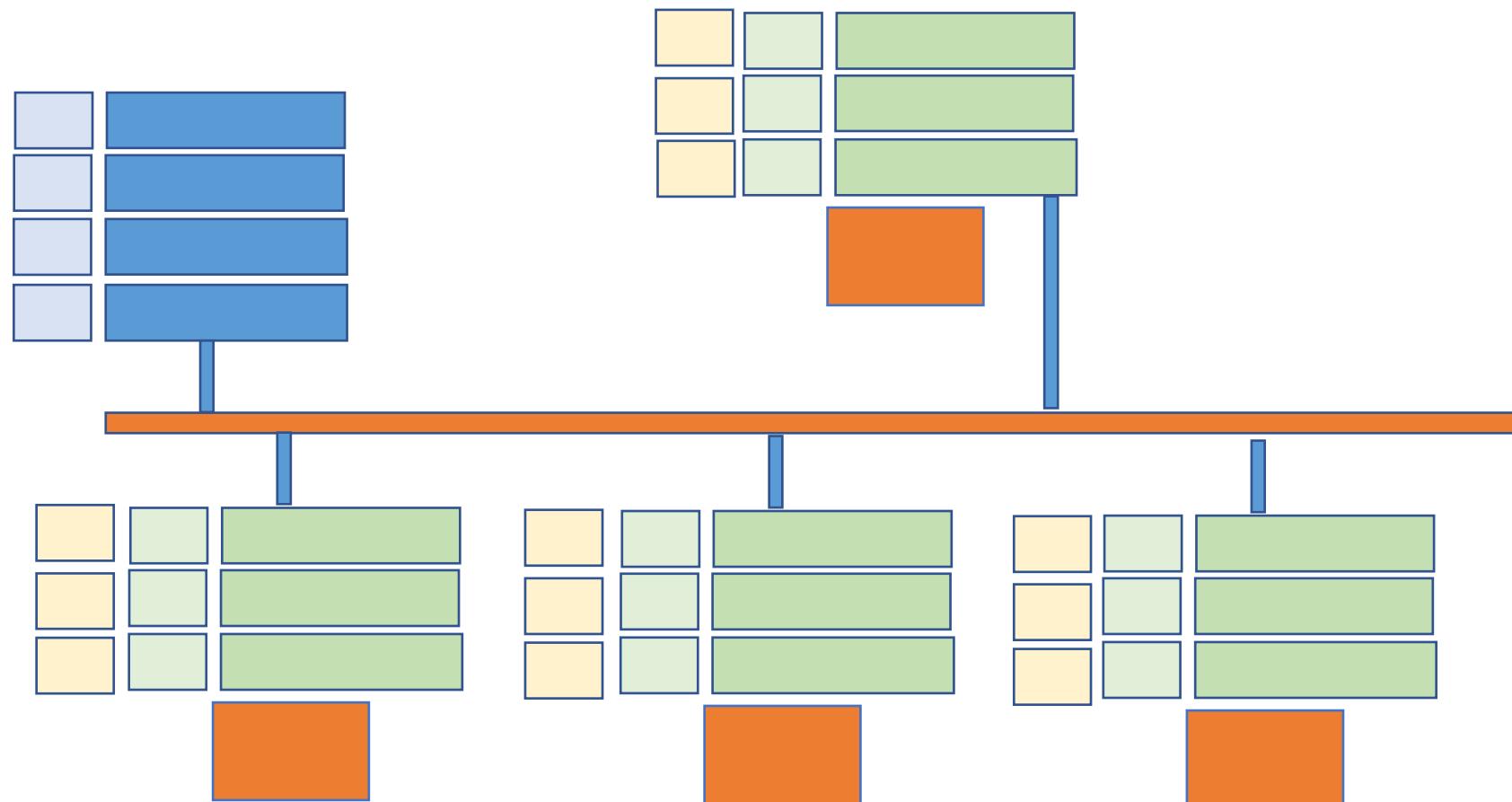
It broadcasts a signal regarding this
-> The other processors get to know about the read miss

Possible Outcomes:

1. If **another cache** has an **exclusive clean** copy. It returns a **signal** indicating that it has a clean copy. Then it (responding processor) changes the state from **exclusive to shared**. And then the initiating processor reads the line from the **main memory**. It also changes its line status to **shared**.

2. If **one or more caches** have a **clean copy** (in **shared** state). They signal that they have the line. The initiating processor reads the data from the memory and marks it as shared (it has been informed by the other two processors)

Simulation Read Miss



Read Miss

Processor tries to read a data from an address. That data is missing in the cache/invalid

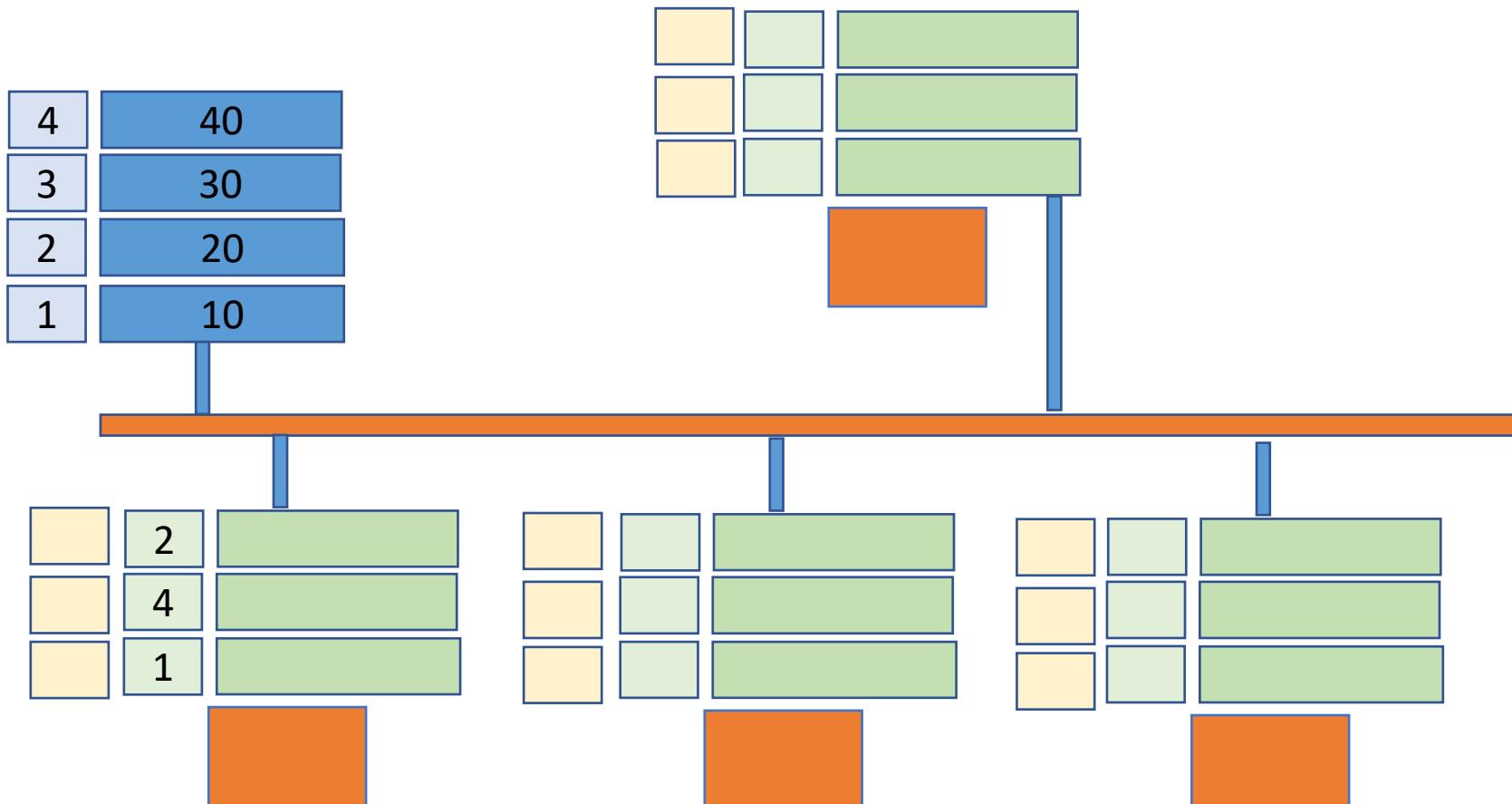
It broadcasts a signal regarding this ->
The other processors get to know about the read miss

Possible Outcomes:

3. If another cache has a **modified copy of the line**, then it **blocks the memory read** and provides the line to the initiating cache. It changes its line to shared. Initiating processor stores it as shared. Memory is updated.

4. If **no other processor** has the data, a simple memory read is done and it is marked as exclusive

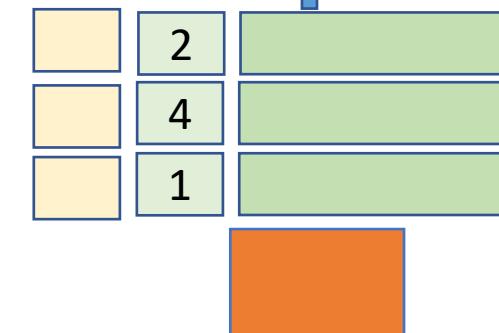
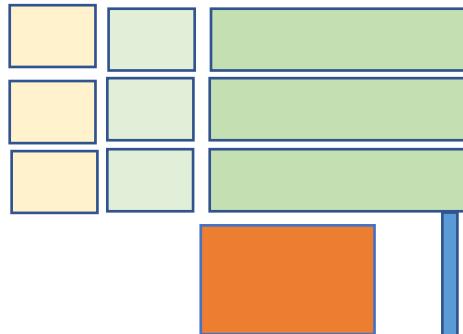
Simulation Read Miss



Wants to read from
Mem location 3

Simulation Read Miss

| | |
|---|----|
| 4 | 40 |
| 3 | 30 |
| 2 | 20 |
| 1 | 10 |

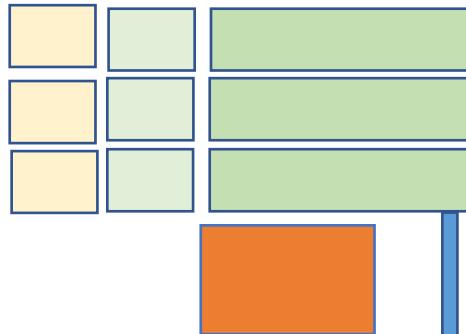


Fails as there is no
image
Of location 3

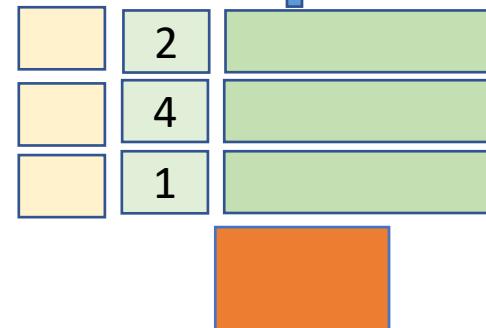
Wants to read from
Mem location 3

Simulation Read Miss

| | |
|---|----|
| 4 | 40 |
| 3 | 30 |
| 2 | 20 |
| 1 | 10 |

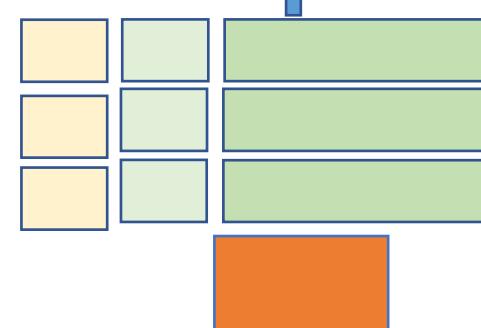
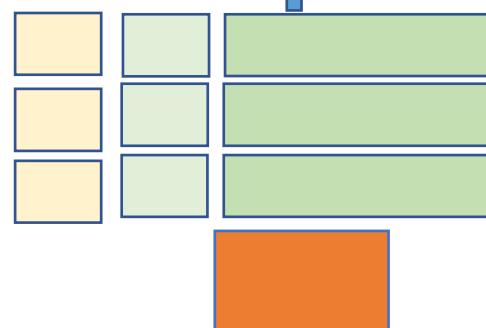


Fails as there is no
image
Of location 3

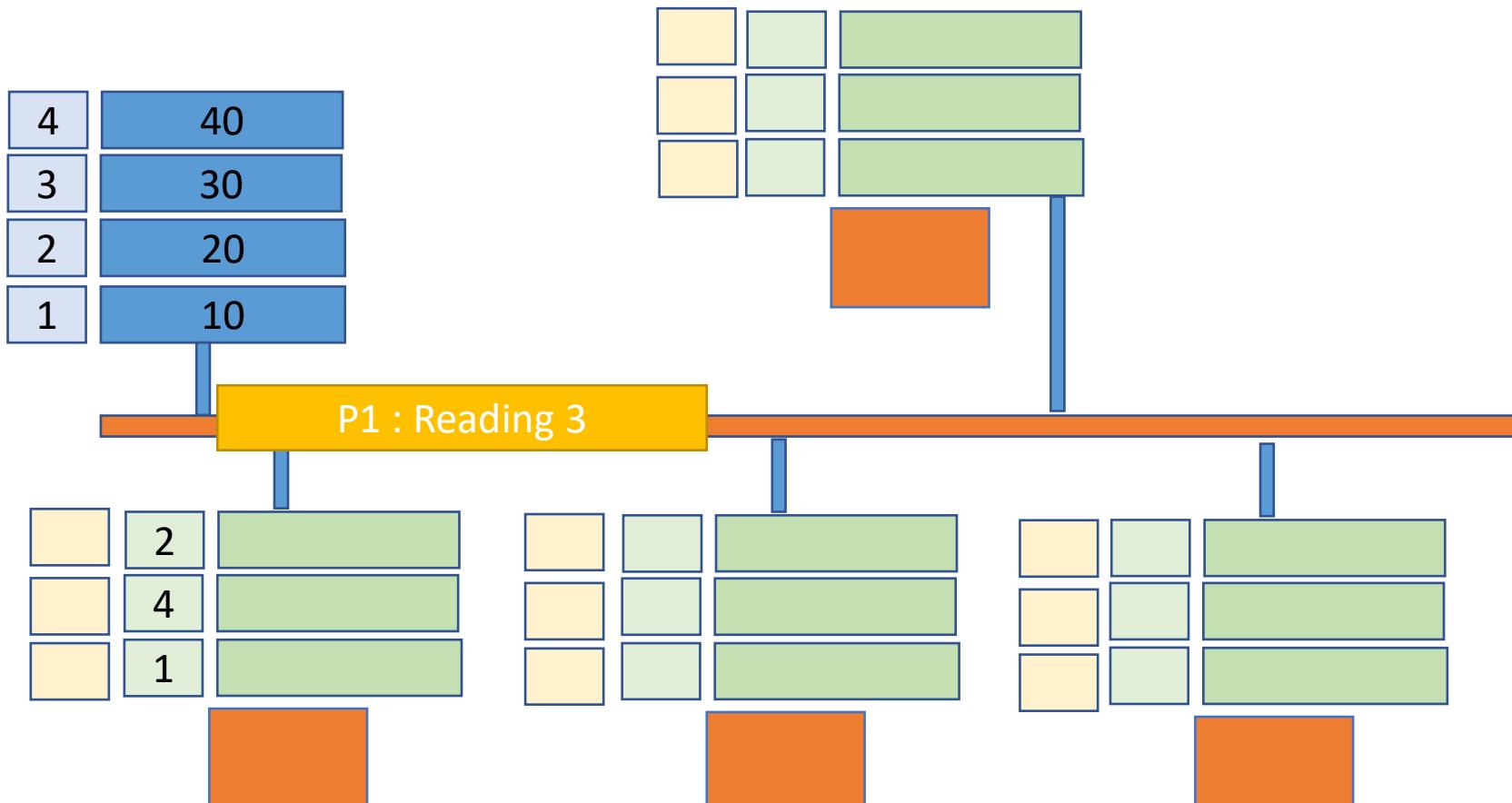


Note: If there was a
location 3 but in
invalid state, it
would still result in
read miss.

Wants to read from
Mem location 3



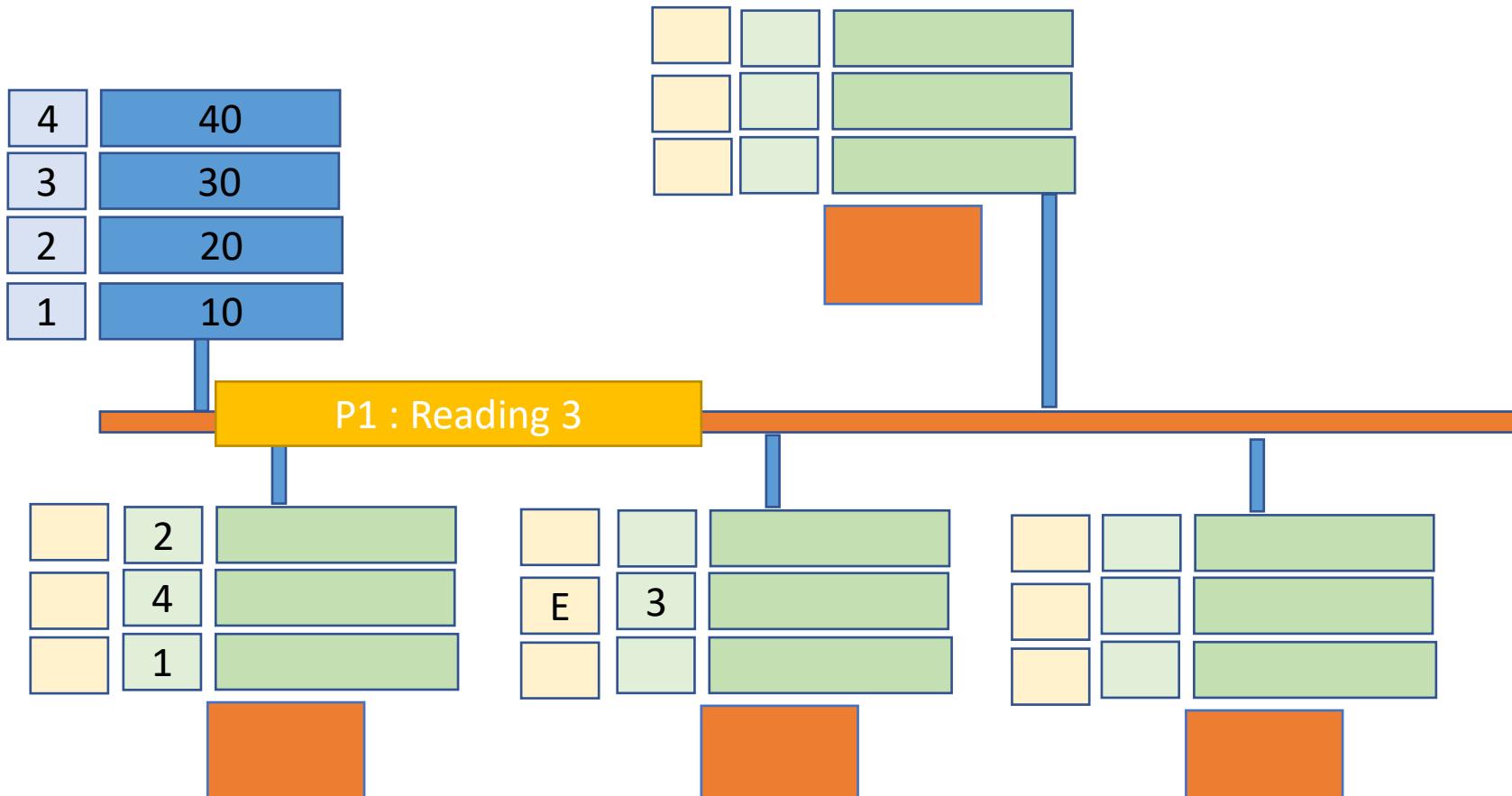
Simulation Read Miss



Wants to read from
Mem location 3

Simulation Read Miss

CASE 1:
Another processor has
The data in Exclusive state.

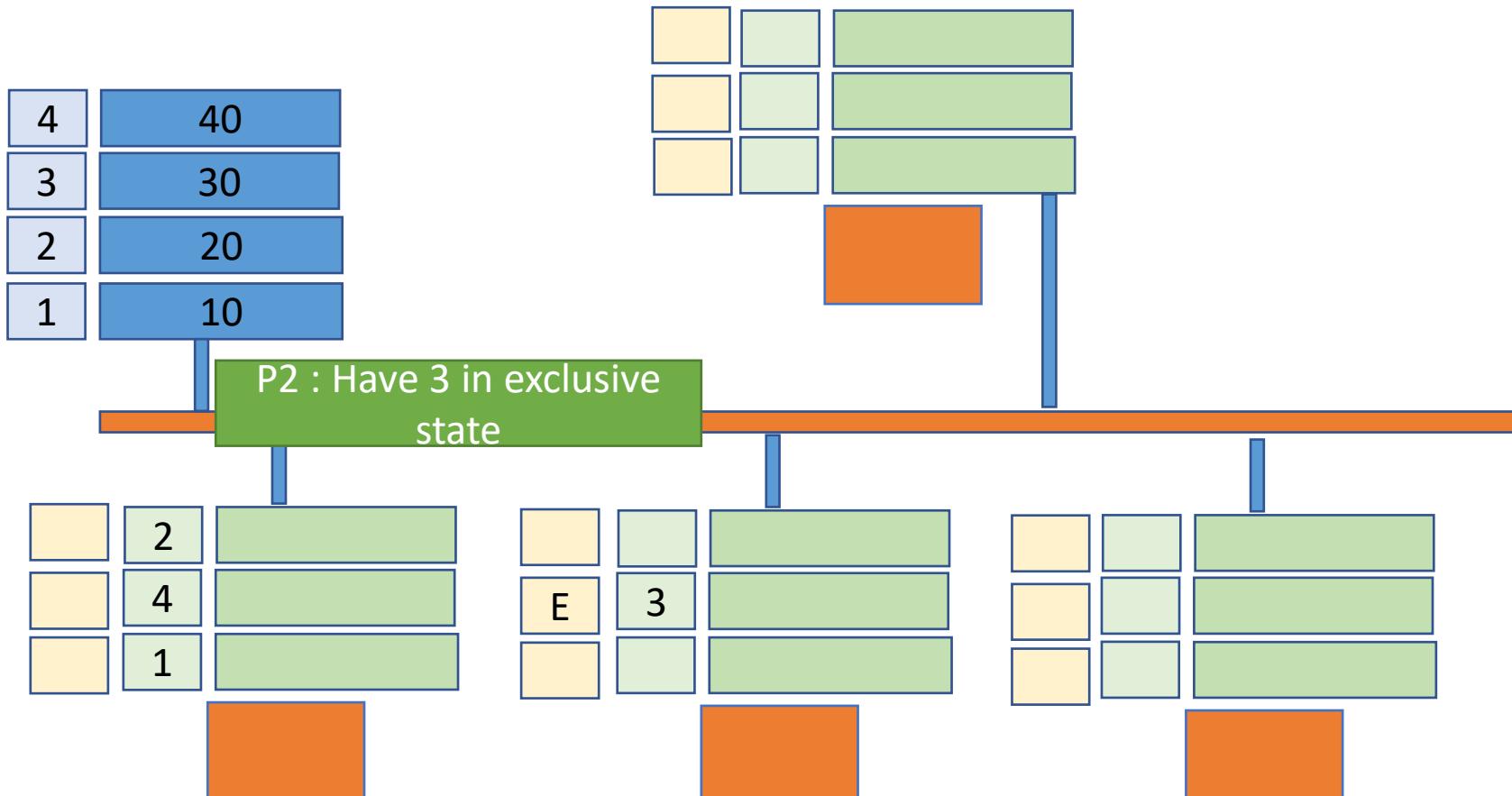


Wants to read from
Mem location 3

Simulation Read Miss

CASE 1:

Another processor has
The data in Exclusive state.

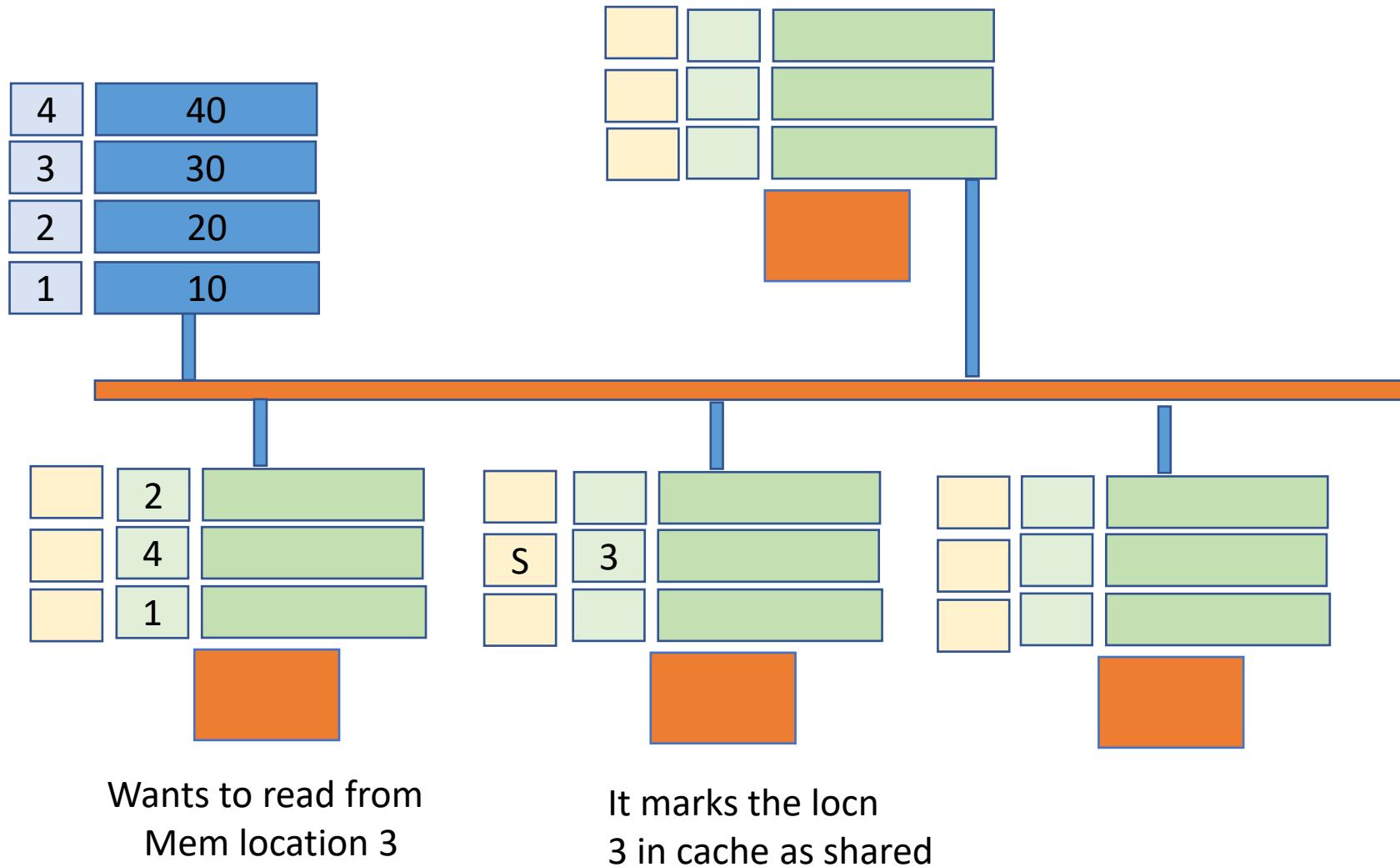


Wants to read from
Mem location 3

Simulation Read Miss

CASE 1:

Another processor has
The data in Exclusive state.

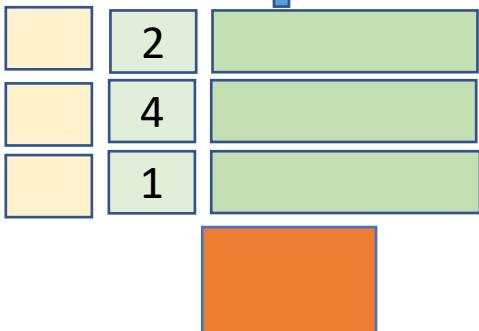
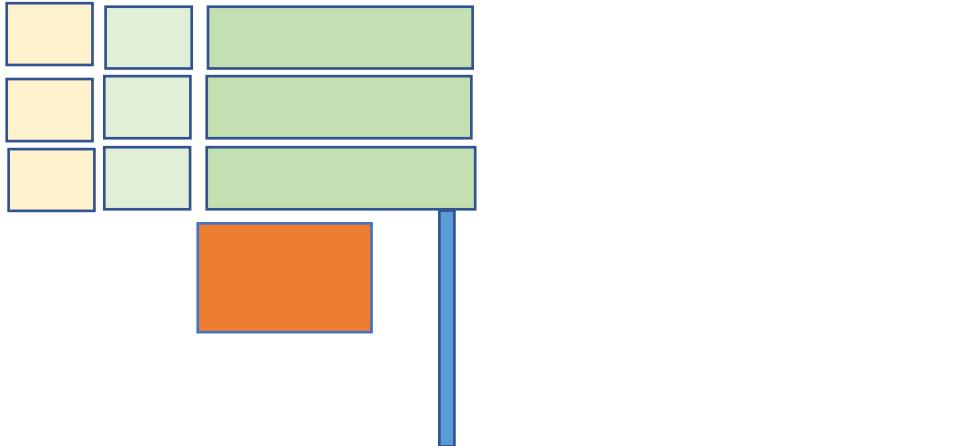


Simulation Read Miss

CASE 1:

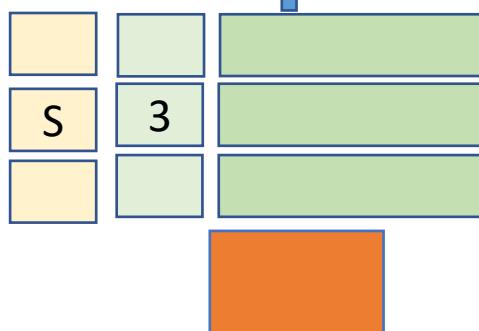
Another processor has
The data in Exclusive state.

| | |
|---|----|
| 4 | 40 |
| 3 | 30 |
| 2 | 20 |
| 1 | 10 |

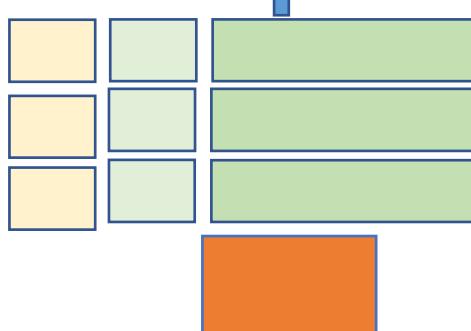


Also, P1 now knows,
That it will be reading
3 but the value will
Be shared

Wants to read from
Mem location 3



It marks the locn
3 in cache as shared

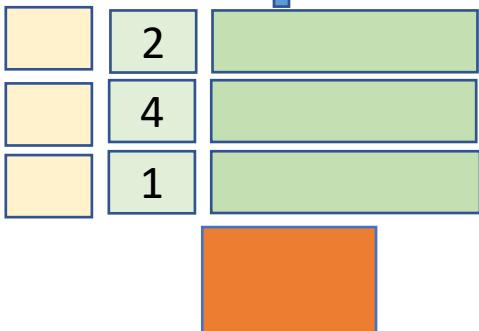
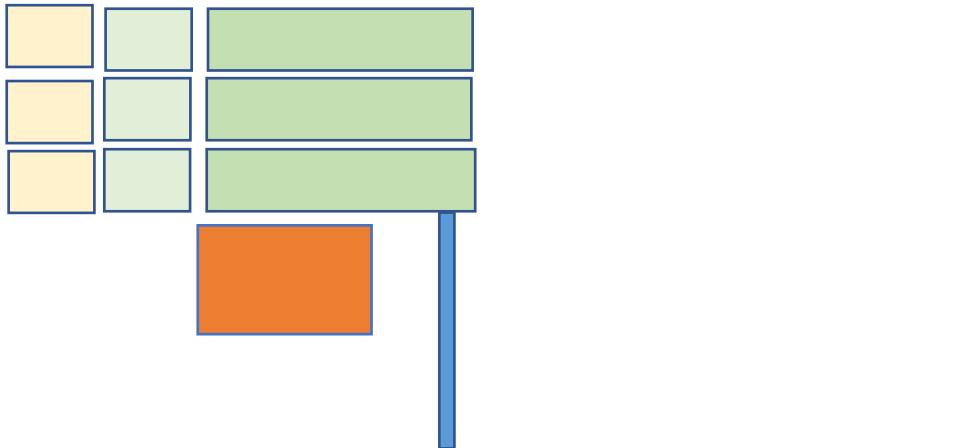


Simulation Read Miss

CASE 1:

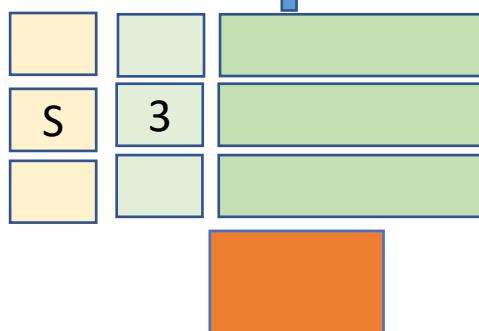
Another processor has
The data in Exclusive state.

| | |
|---|----|
| 4 | 40 |
| 3 | 30 |
| 2 | 20 |
| 1 | 10 |

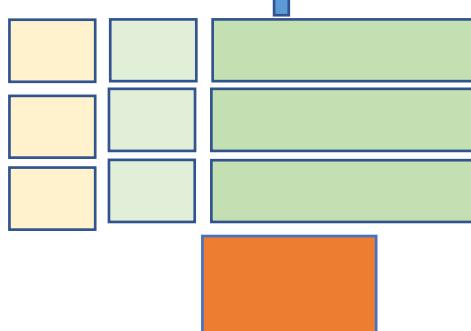


Also, P1 now knows,
That it will be reading
3 but the value will
Be shared

Wants to read from
Mem location 3



It marks the locn
3 in cache as shared

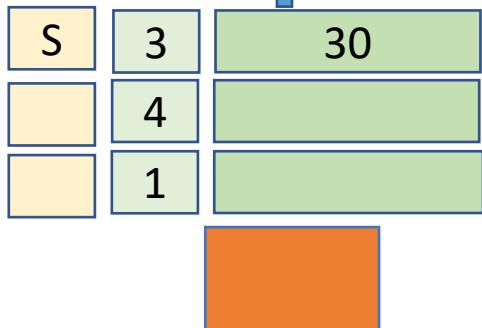
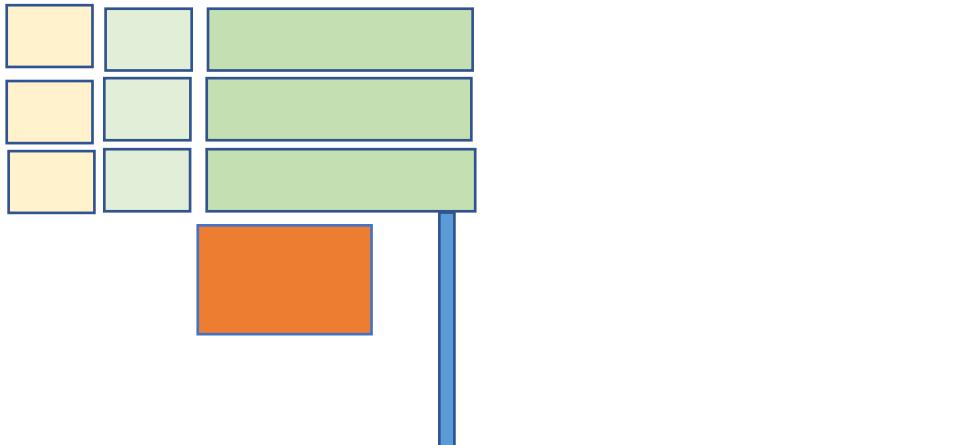


Simulation Read Miss

CASE 1:

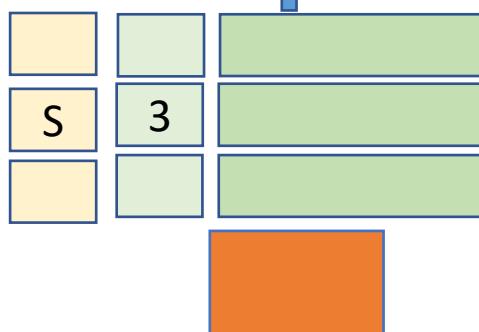
Another processor has
The data in Exclusive state.

| | |
|---|----|
| 4 | 40 |
| 3 | 30 |
| 2 | 20 |
| 1 | 10 |

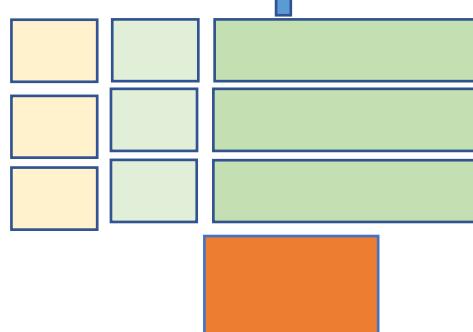


Also, P1 now knows,
That it will be reading
3 but the value will
Be shared

Wants to read from
Mem location 3



It marks the locn
3 in cache as shared



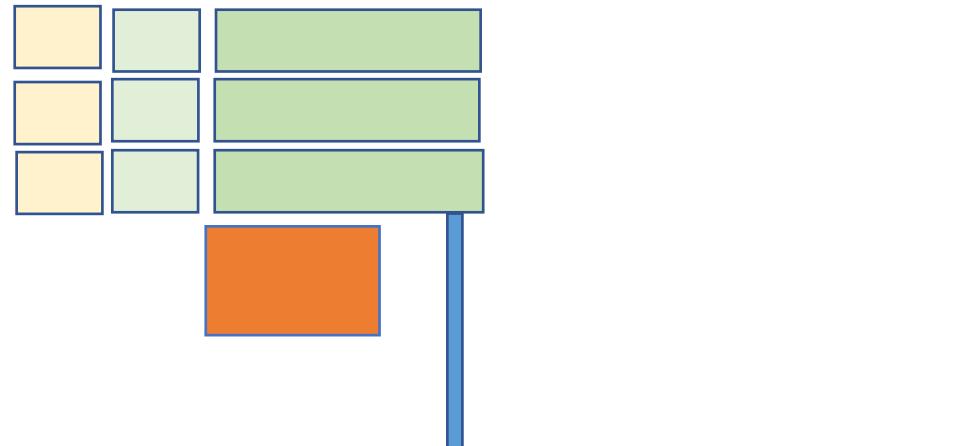
Simulation Read Miss

CASE 1:

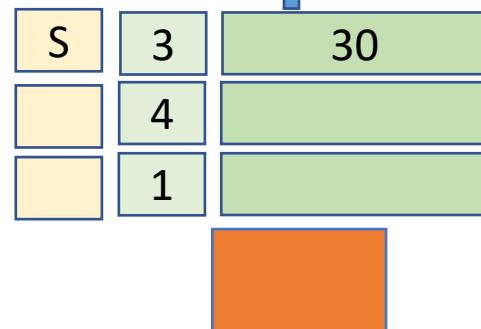
Another processor has
The data in Exclusive state.

Note that , the mem locn 2 is no longer in the cache. If it was in a valid state (M/S/E), It would be written back Into the memory.

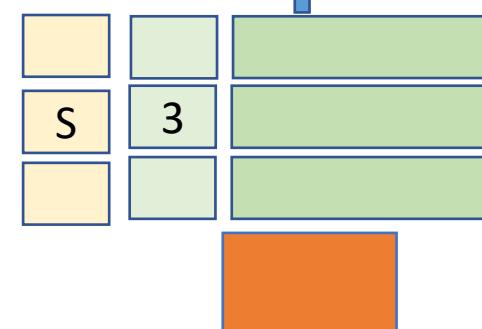
| | |
|---|----|
| 4 | 40 |
| 3 | 30 |
| 2 | 20 |
| 1 | 10 |



Also, P1 now knows,
That it will be reading
3 but the value will
Be shared



Wants to read from
Mem location 3

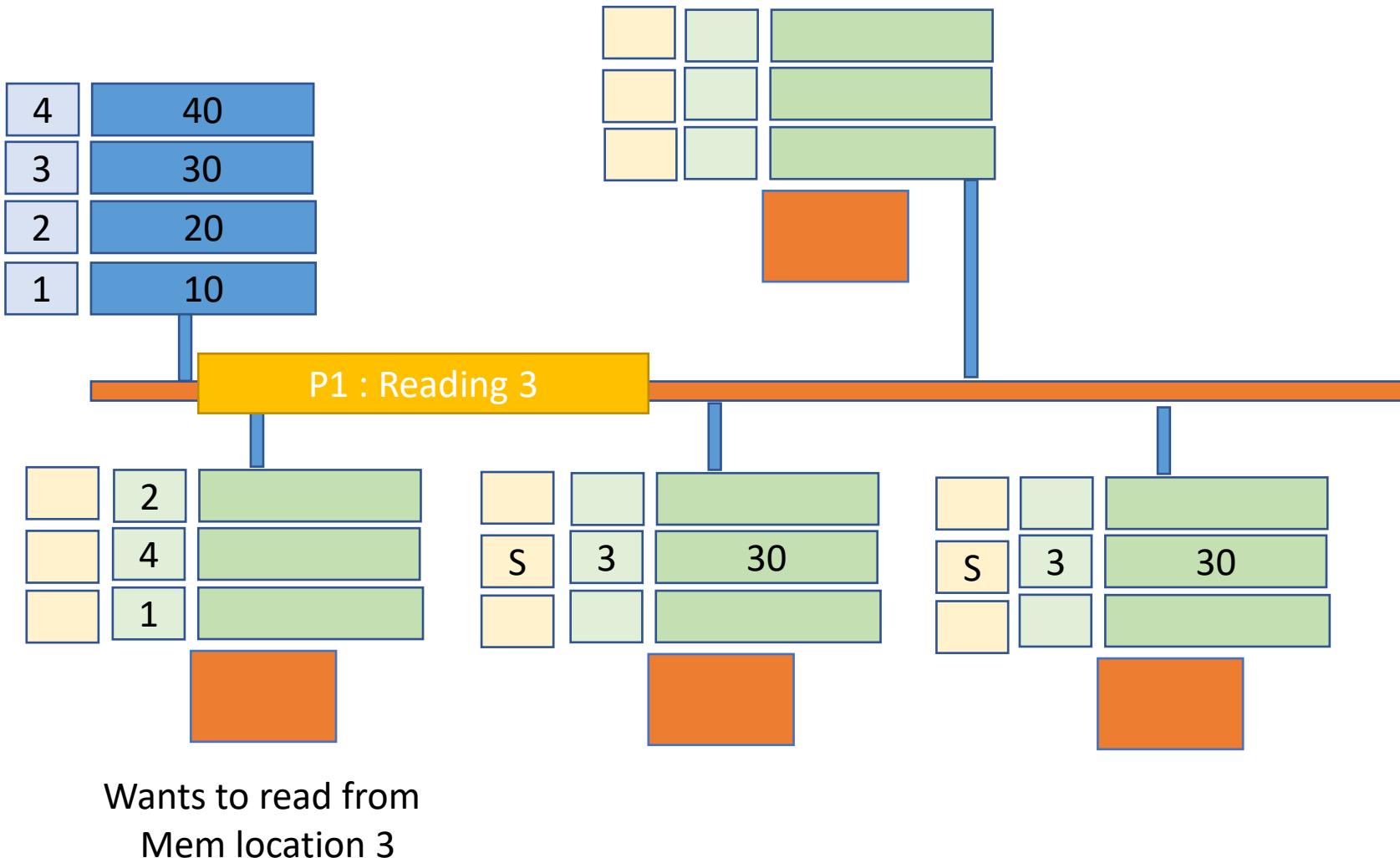


It marks the locn
3 in cache as shared

Simulation Read Miss

CASE 2:

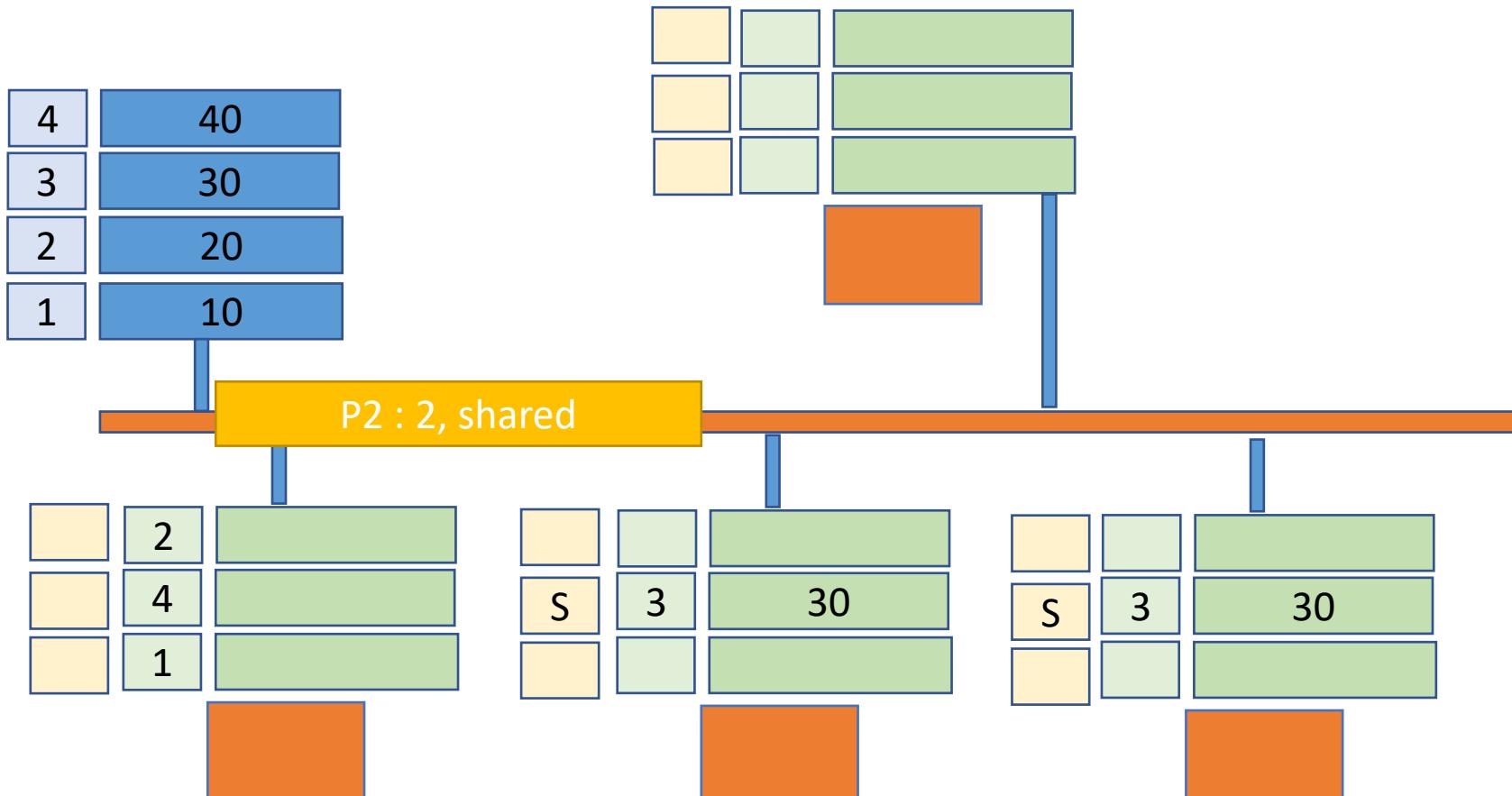
2 Other processors have
The data in shared state.



Simulation Read Miss

CASE 2:

2 Other processors have
The data in shared state.

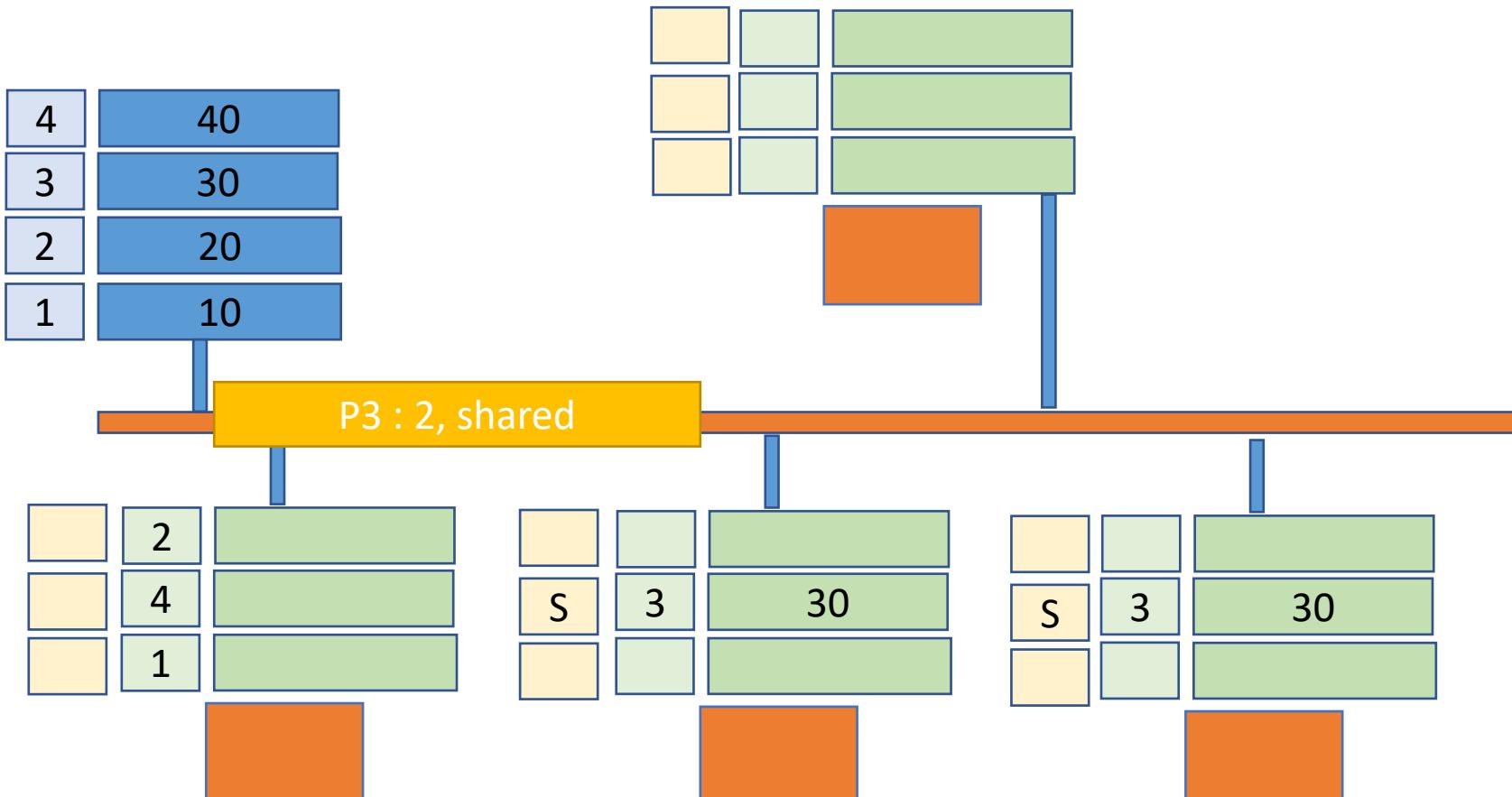


Wants to read from
Mem location 3

Simulation Read Miss

CASE 2:

2 Other processors have
The data in shared state.

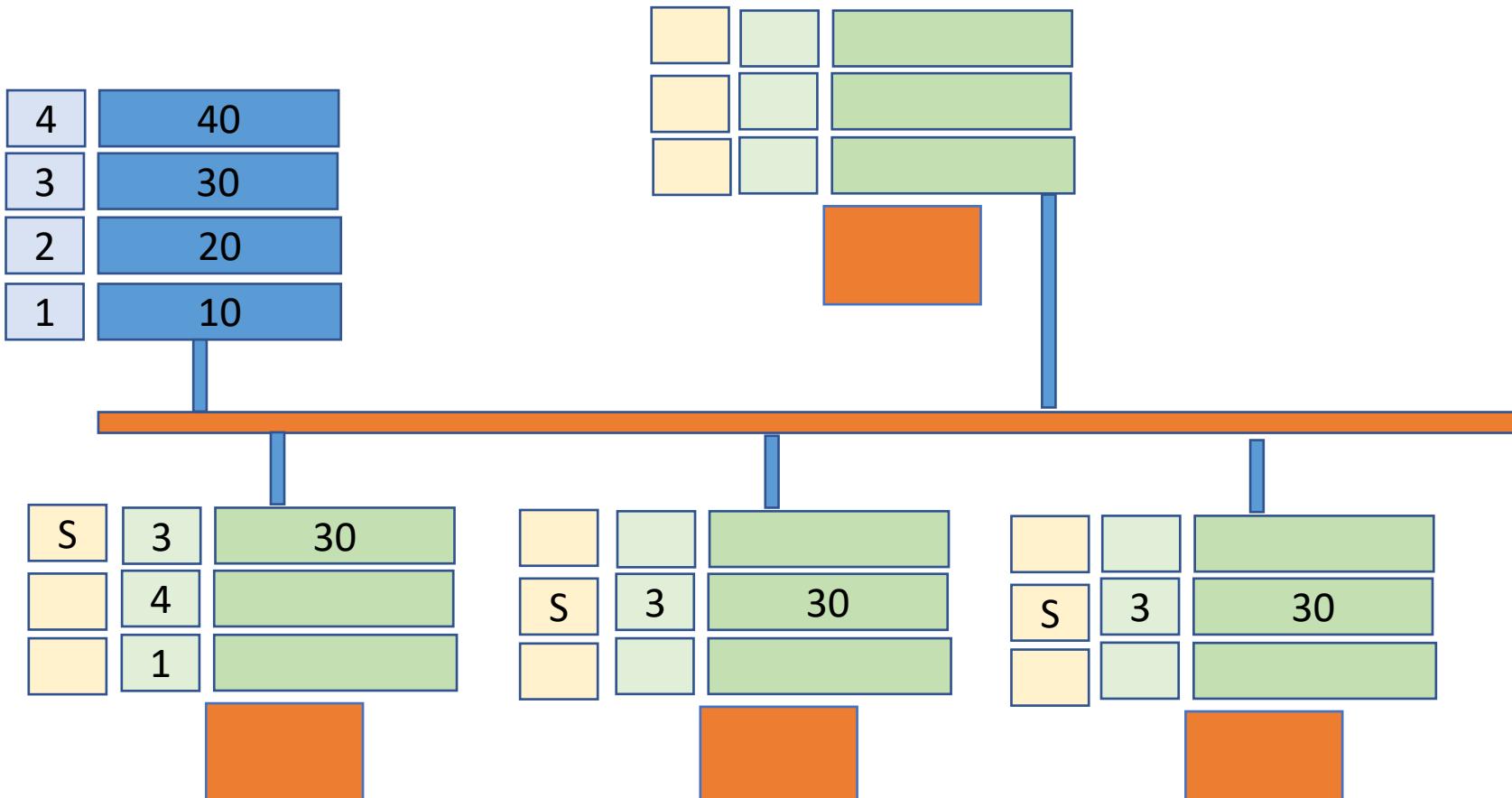


Wants to read from
Mem location 3

Simulation Read Miss

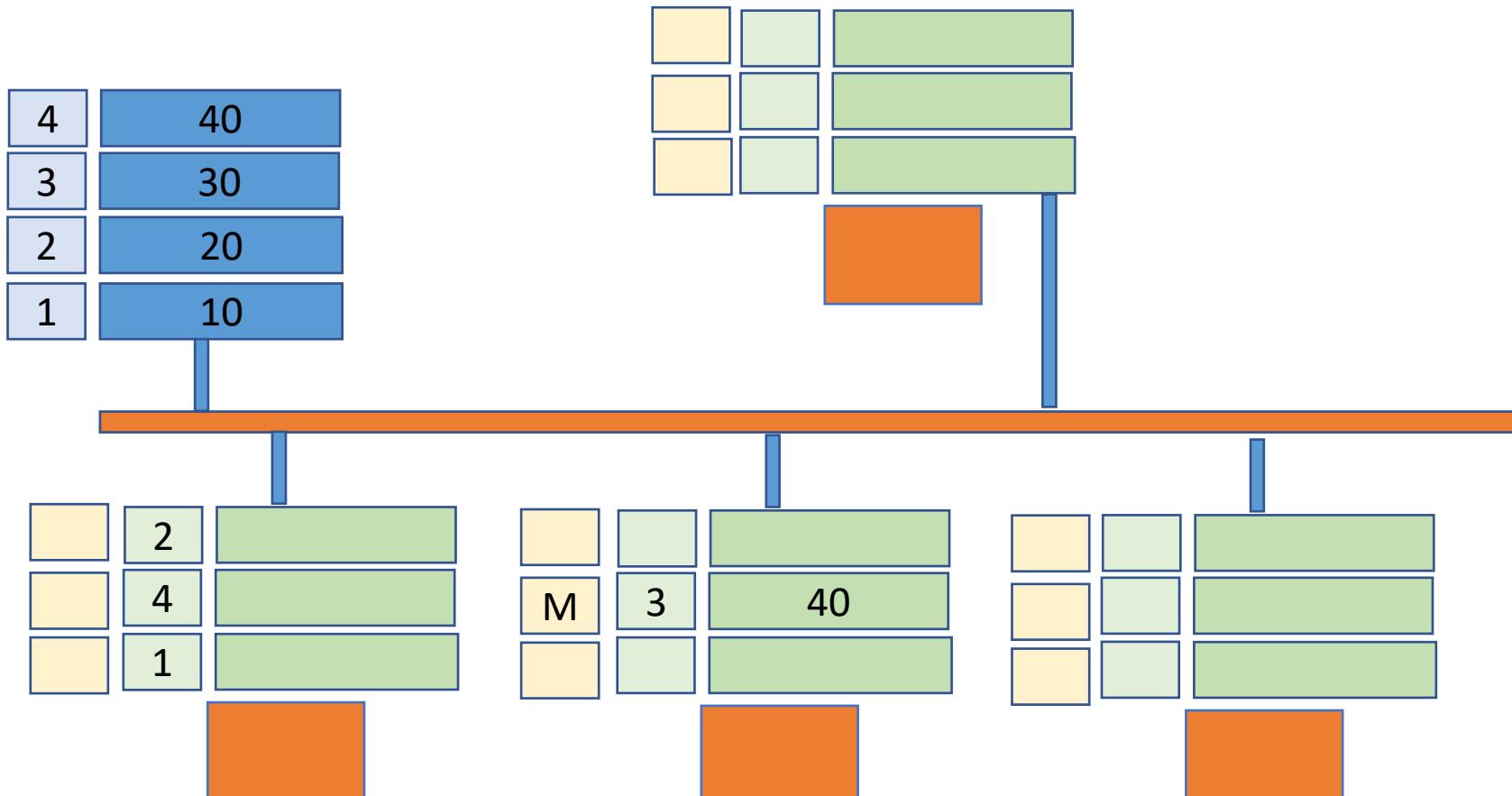
CASE 2:

2 Other processors have
The data in shared state.



Simulation Read Miss

CASE 3:
Another processor has the
Data in **Modified** state

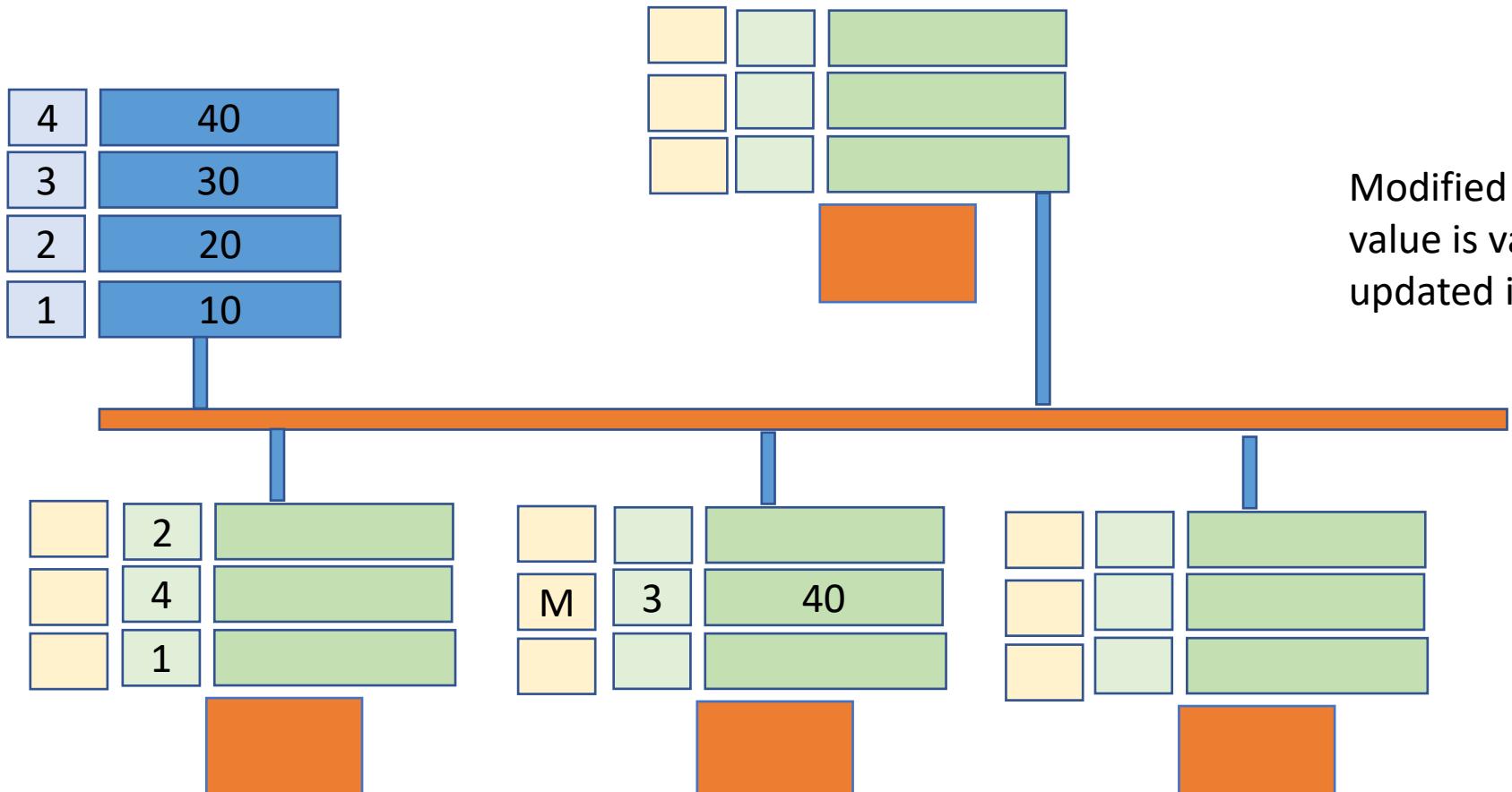


Wants to read from
Mem location 3

Simulation Read Miss

CASE 3:

Another processor has the Data in **Modified** state



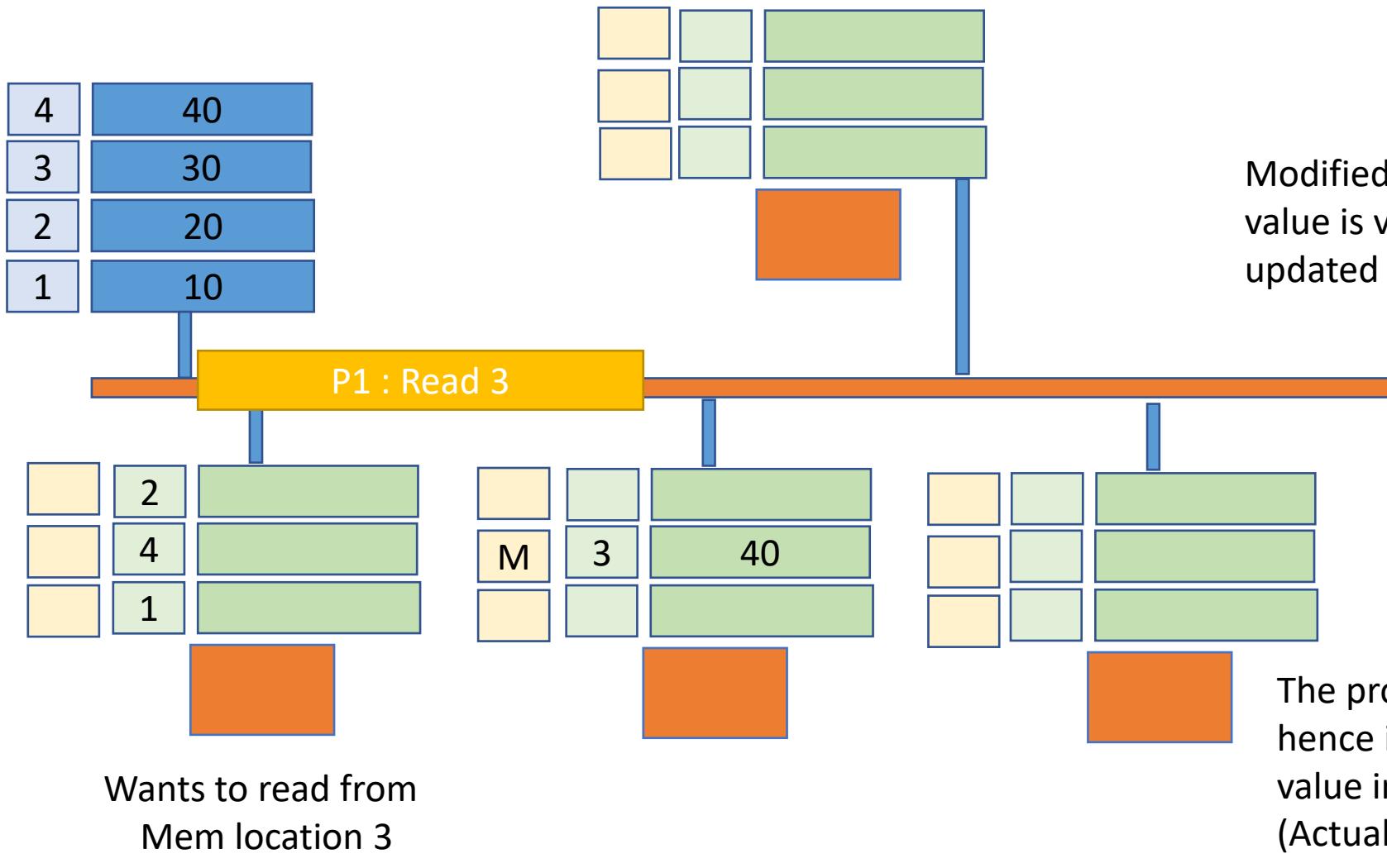
Wants to read from
Mem location 3

Modified means that, this value is valid, and it will be updated into the memory

Simulation Read Miss

CASE 3:

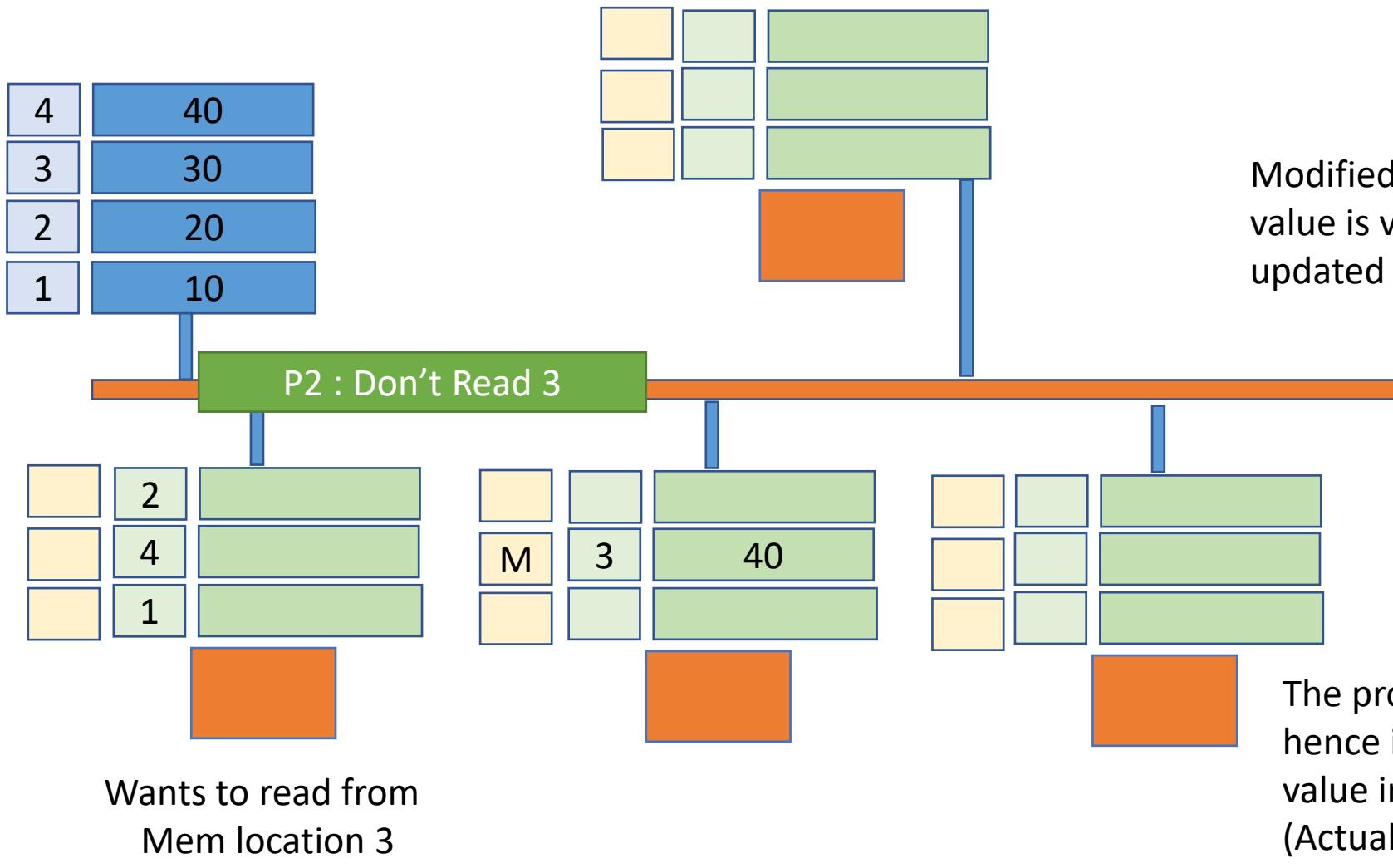
Another processor has the Data in **Modified** state



Simulation Read Miss

CASE 3:

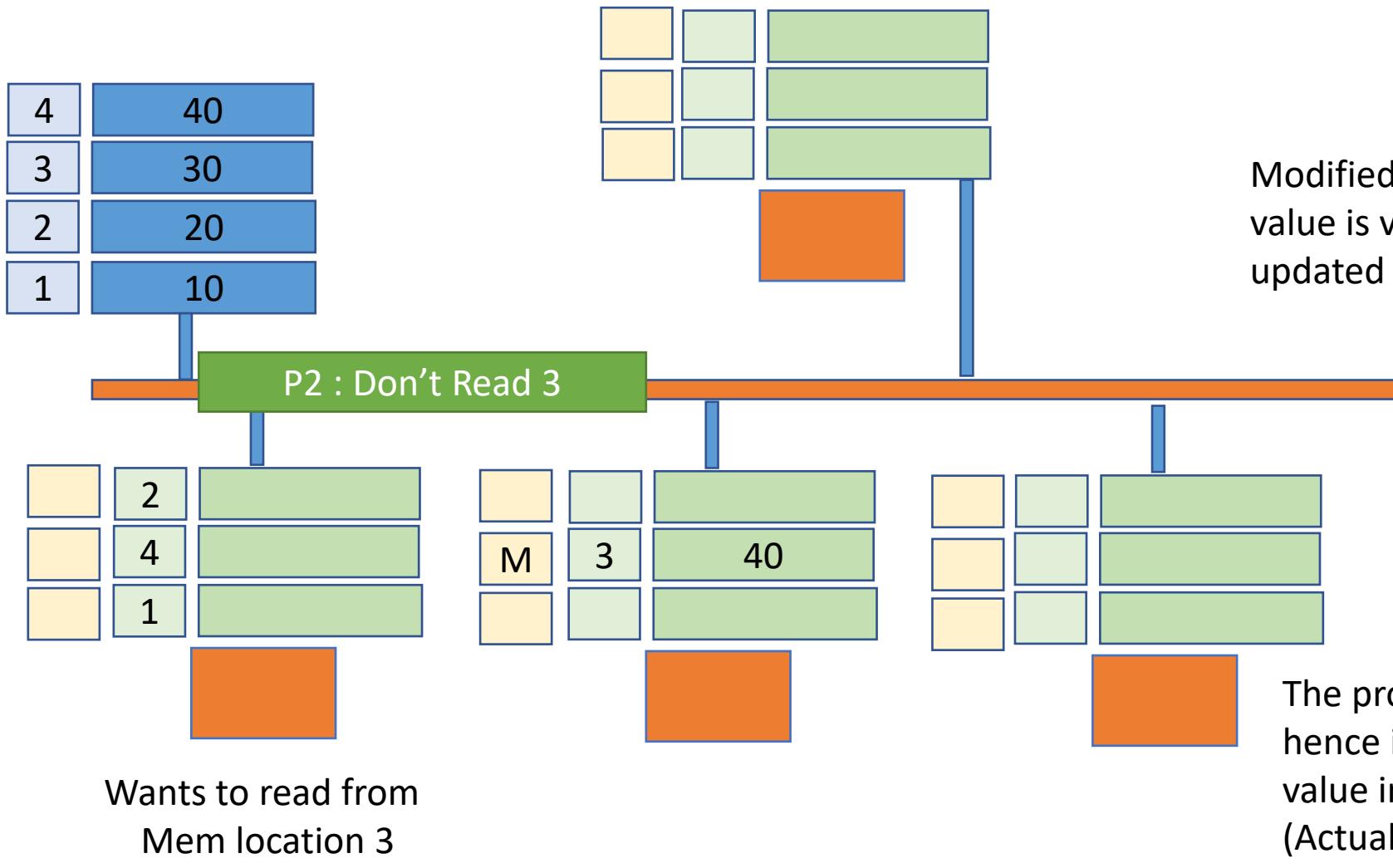
Another processor has the Data in **Modified** state



Simulation Read Miss

CASE 3:

Another processor has the Data in **Modified** state



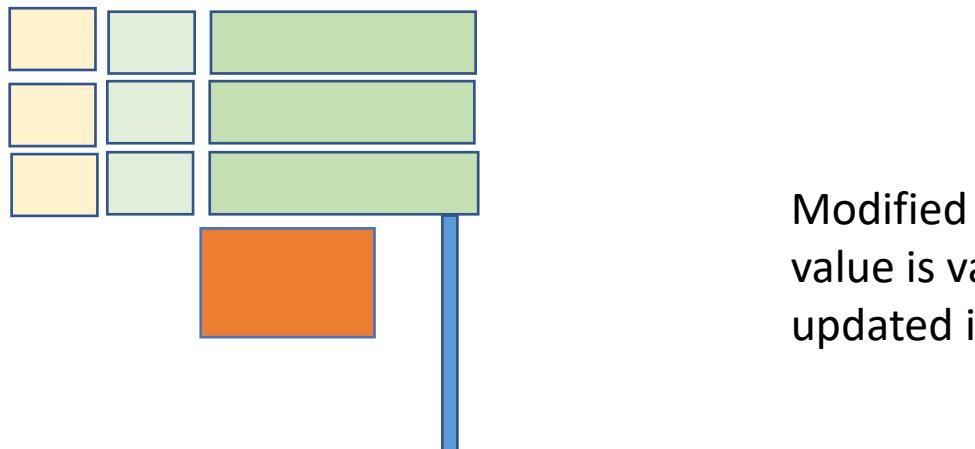
Simulation Read Miss

CASE 3:

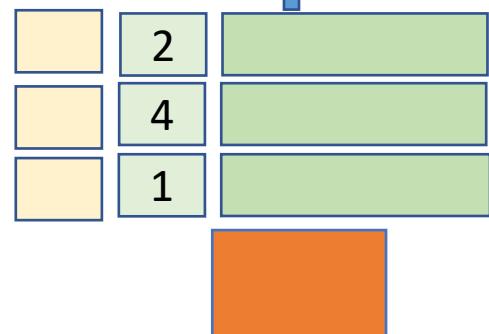
Another processor has the Data in **Modified** state

The memory is Updated with the modified value

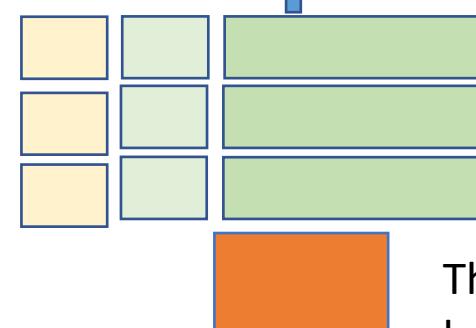
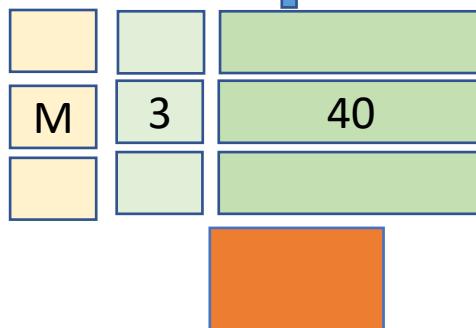
| | |
|---|----|
| 4 | 40 |
| 3 | 40 |
| 2 | 20 |
| 1 | 10 |



Modified means that, this value is valid, and it will be updated into the memory



Wants to read from
Mem location 3



The processor is just lazy and hence it has not yet placed the value into the memory.
(Actually this is because of the protocol)

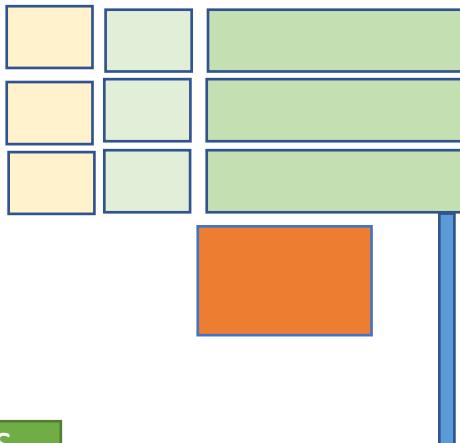
Simulation Read Miss

CASE 3:

Another processor has the Data in **Modified** state

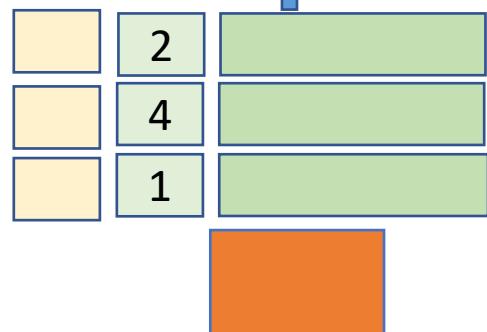
The memory is Updated with the modified value

| | |
|---|----|
| 4 | 40 |
| 3 | 40 |
| 2 | 20 |
| 1 | 10 |

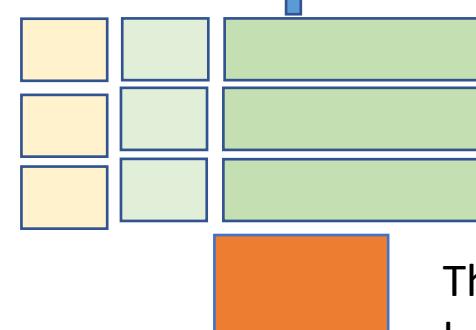
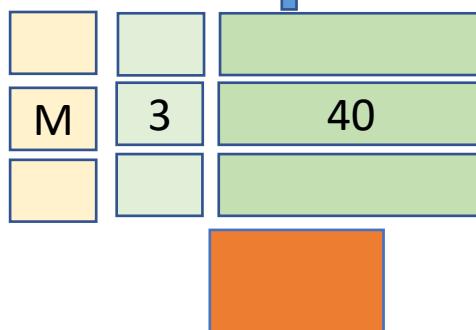


Modified means that, this value is valid, and it will be updated into the memory

P2 : The actual value is
40 for 3



Wants to read from
Mem location 3



The processor is just lazy and hence it has not yet placed the value into the memory.
(Actually this is because of the protocol)

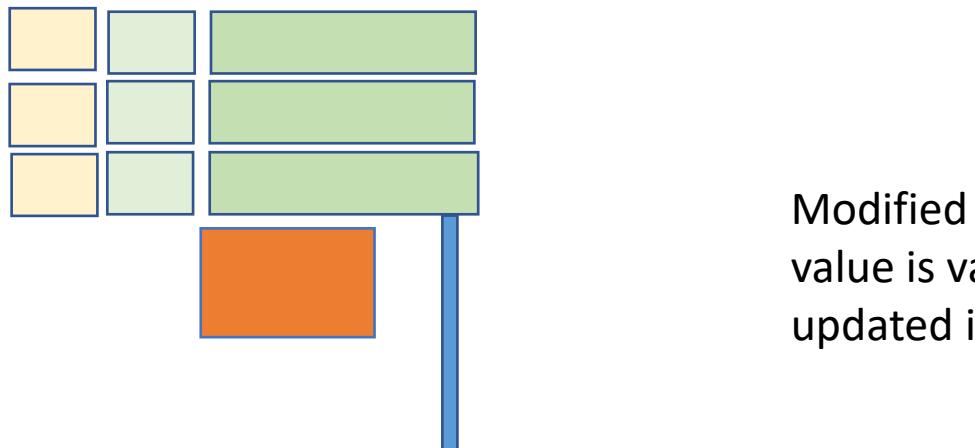
Simulation Read Miss

CASE 3:

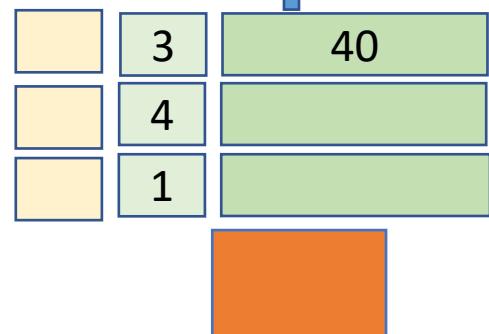
Another processor has the Data in **Modified** state

The memory is Updated with the Updated below

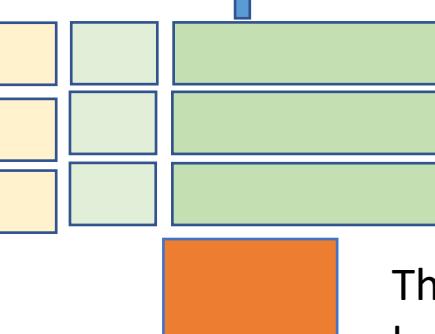
| | |
|---|----|
| 4 | 40 |
| 3 | 40 |
| 2 | 20 |
| 1 | 10 |



Modified means that, this value is valid, and it will be updated into the memory



Wants to read from
Mem location 3



The processor is just lazy and hence it has not yet placed the value into the memory.
(Actually this is because of the protocol)

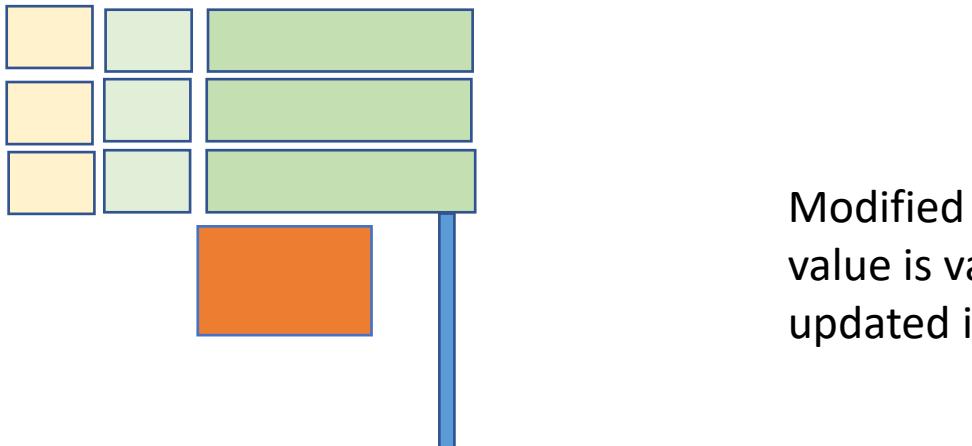
Simulation Read Miss

CASE 3:

Another processor has the Data in **Modified** state

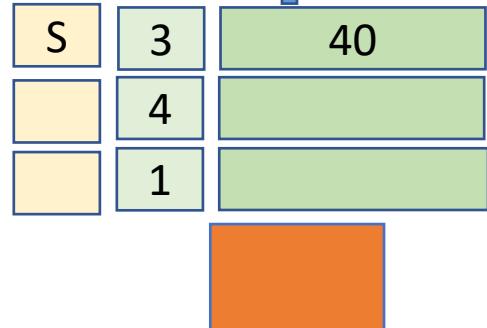
The memory is Updated with the Updated below

| | |
|---|----|
| 4 | 40 |
| 3 | 40 |
| 2 | 20 |
| 1 | 10 |

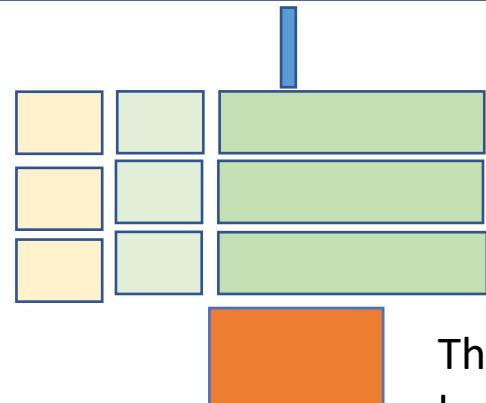
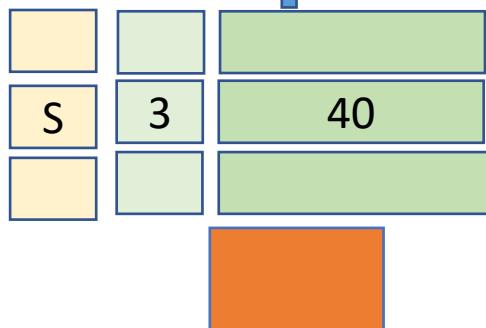


Modified means that, this value is valid, and it will be updated into the memory

Both of them are marked As shared

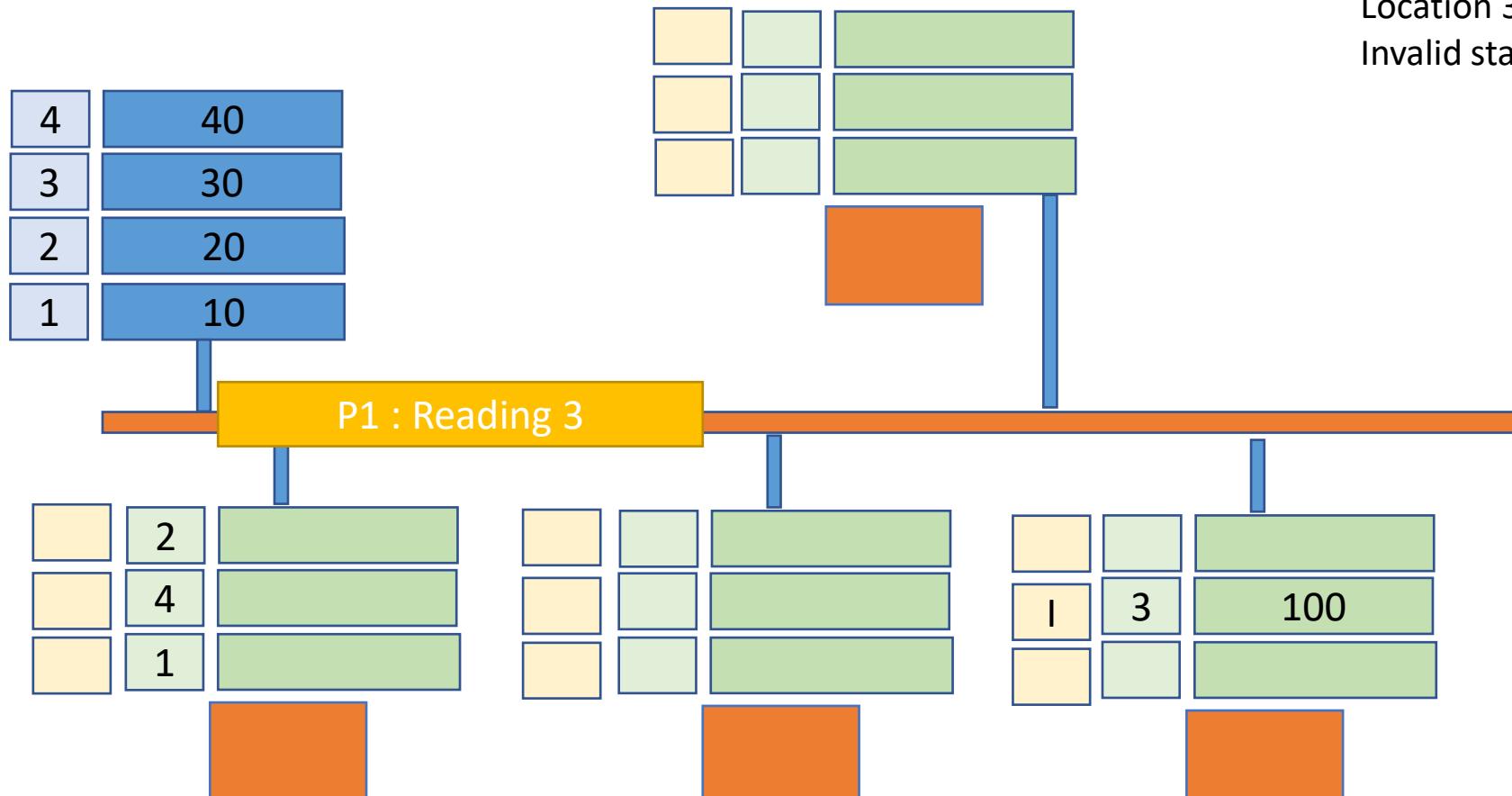


Wants to read from
Mem location 3



The processor is just lazy and hence it has not yet placed the value into the memory.
(Actually this is because of the protocol)

Simulation Read Miss

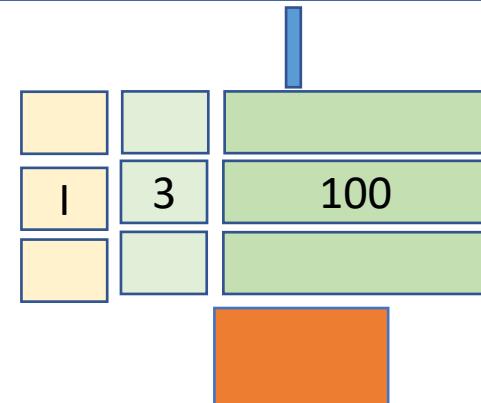
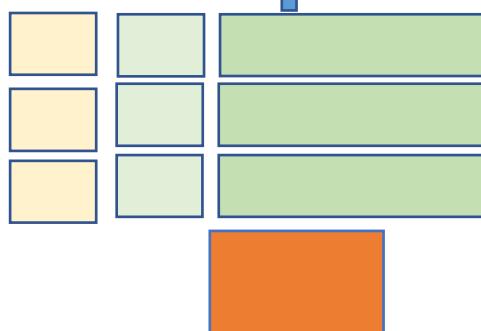
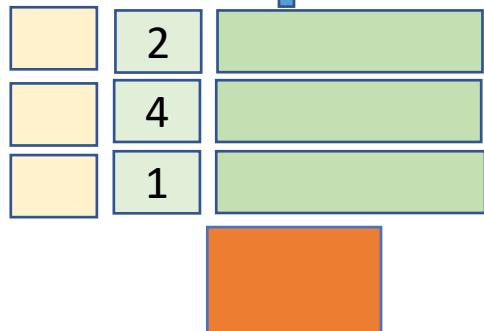
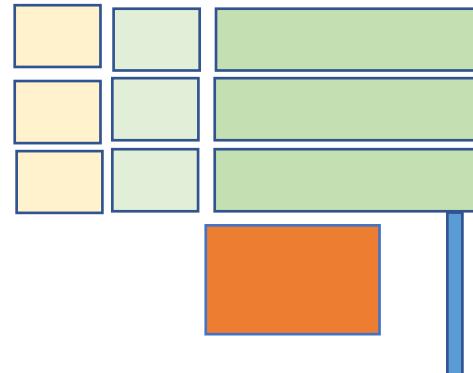


CASE 4:

No other processor
Has the data of the mem
Location 3, or have it in
Invalid state.

Simulation Read Miss

| | |
|---|----|
| 4 | 40 |
| 3 | 30 |
| 2 | 20 |
| 1 | 10 |



Wants to read from
Mem location 3

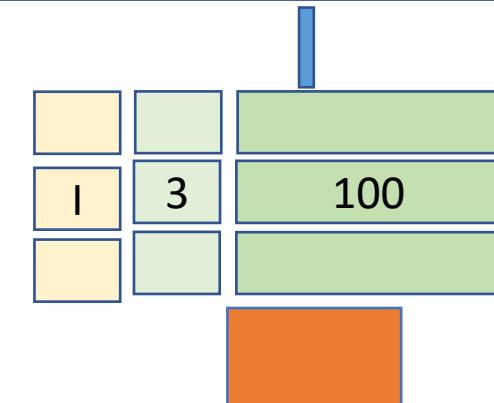
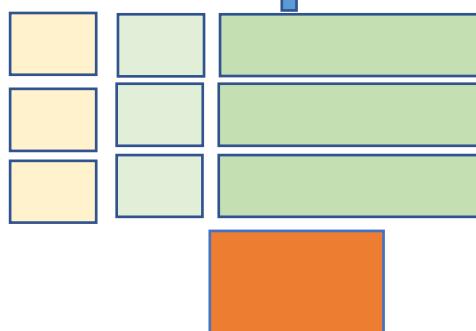
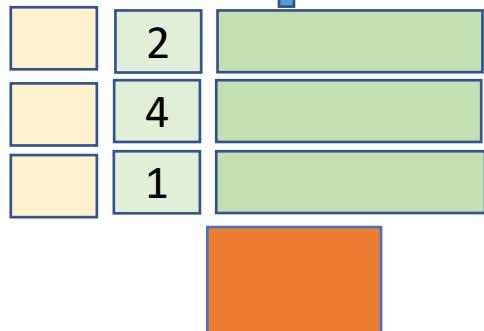
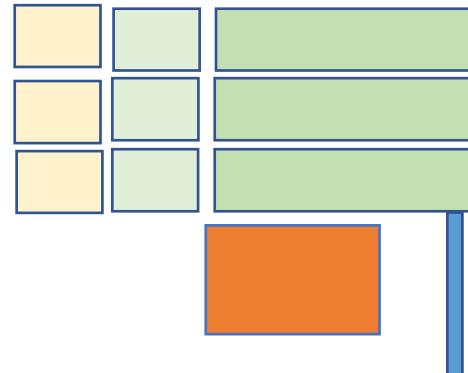
CASE 4:

No other processor
Has the data of the mem
Location 3, or have it in
Invalid state.

No response obtained,
From other processors

Simulation Read Miss

| | |
|---|----|
| 4 | 40 |
| 3 | 30 |
| 2 | 20 |
| 1 | 10 |



Wants to read from
Mem location 3

CASE 4:

No other processor
Has the data of the mem
Location 3, or have it in
Invalid state.

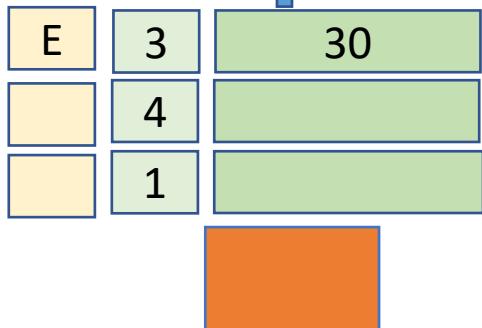
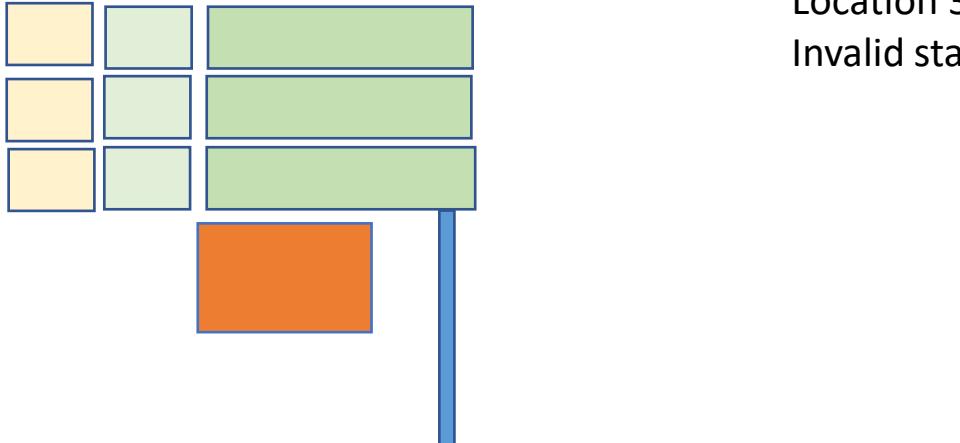
No response obtained,
From other processors

Simulation Read Miss

CASE 4:

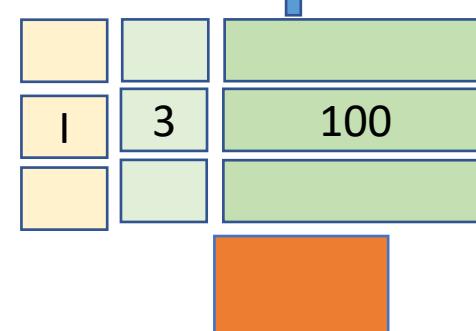
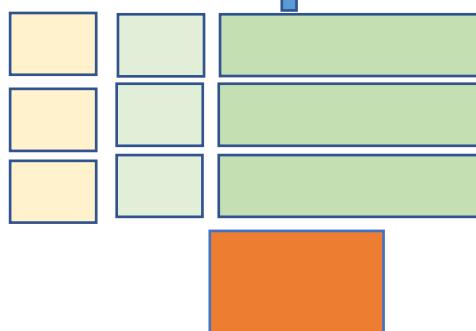
No other processor
Has the data of the mem
Location 3, or have it in
Invalid state.

| | |
|---|----|
| 4 | 40 |
| 3 | 30 |
| 2 | 20 |
| 1 | 10 |

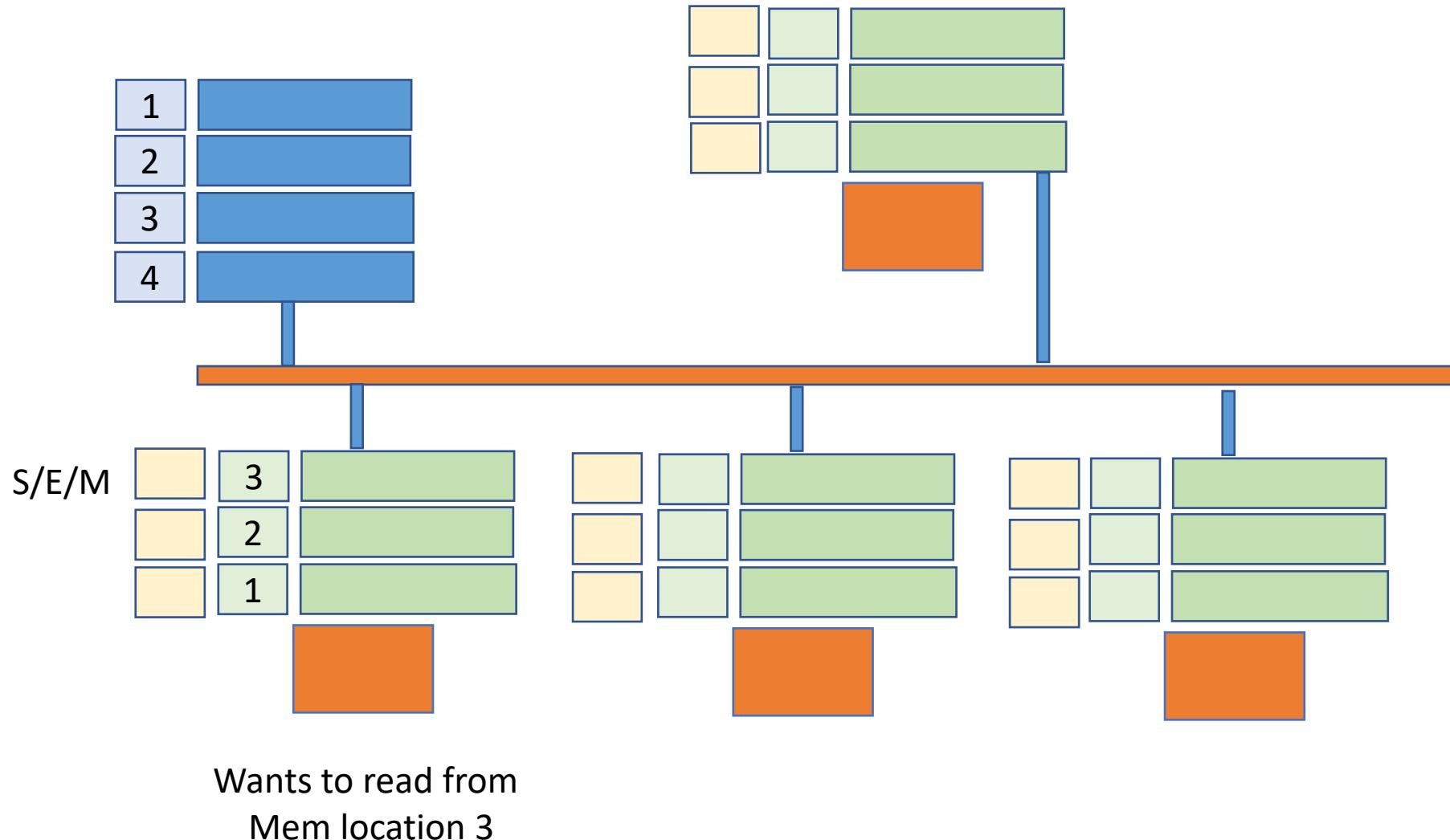


The processor simply
Reads the data and
Marks it as exclusive

Wants to read from
Mem location 3



Simulation Read Hit

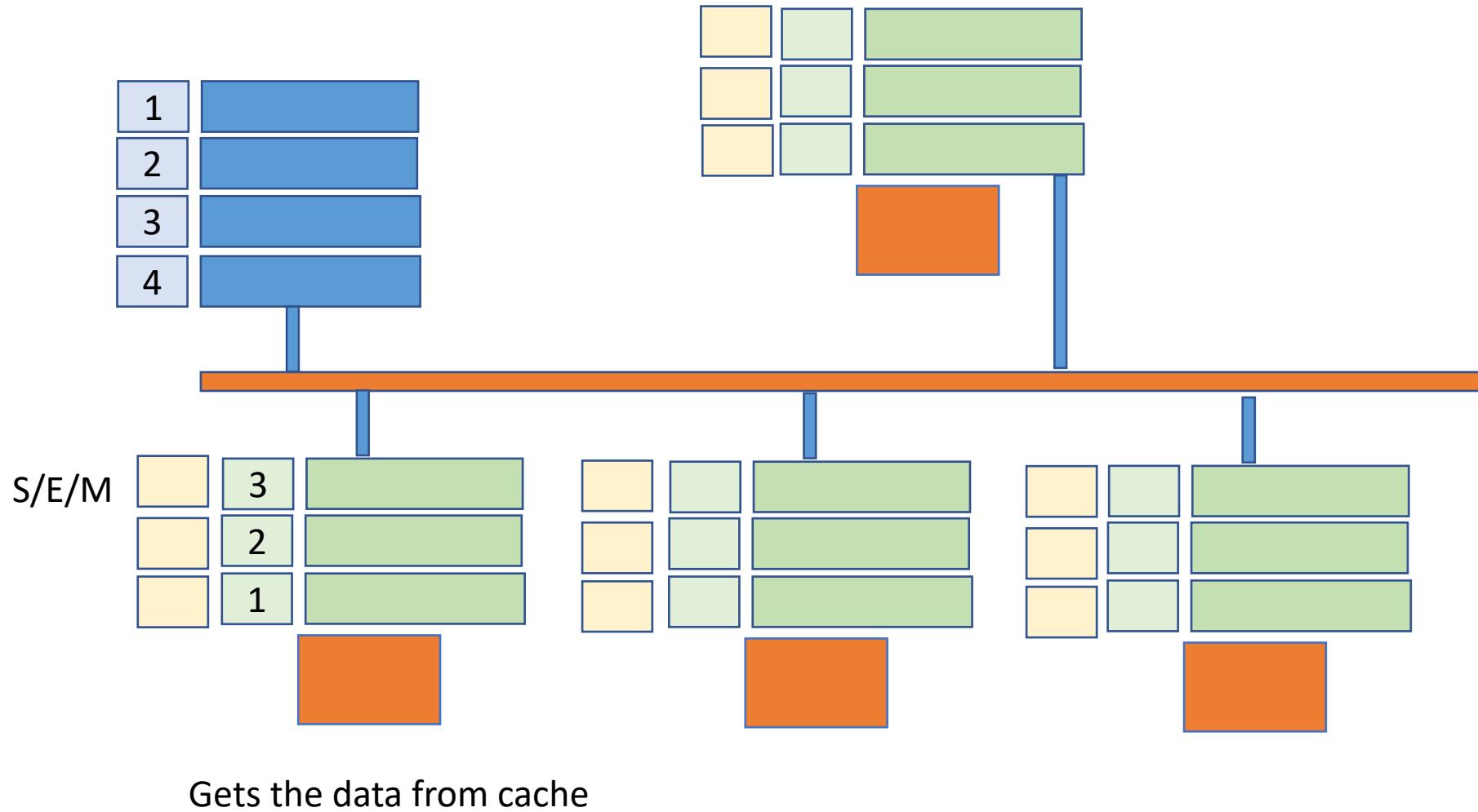


Read Hit:

Processor gets what it wanted to read in the cache. (in a valid state obviously)

No change of state occurs here for that line

Simulation Read Hit

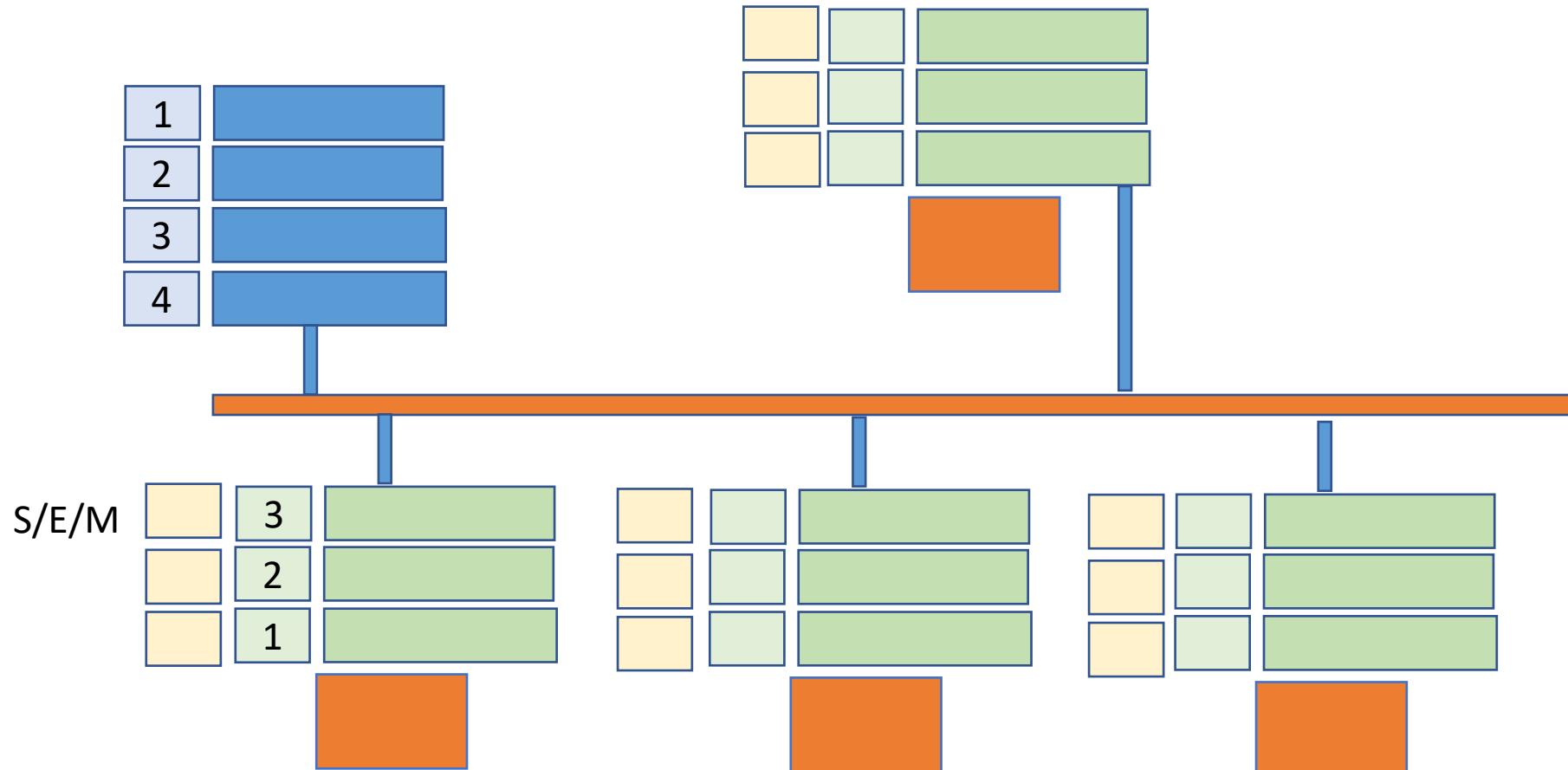


Read Hit:

Processor gets what it wanted to read in the cache. (in a valid state obviously)

No change of state occurs here for that line

Simulation Read Hit



Read Hit:

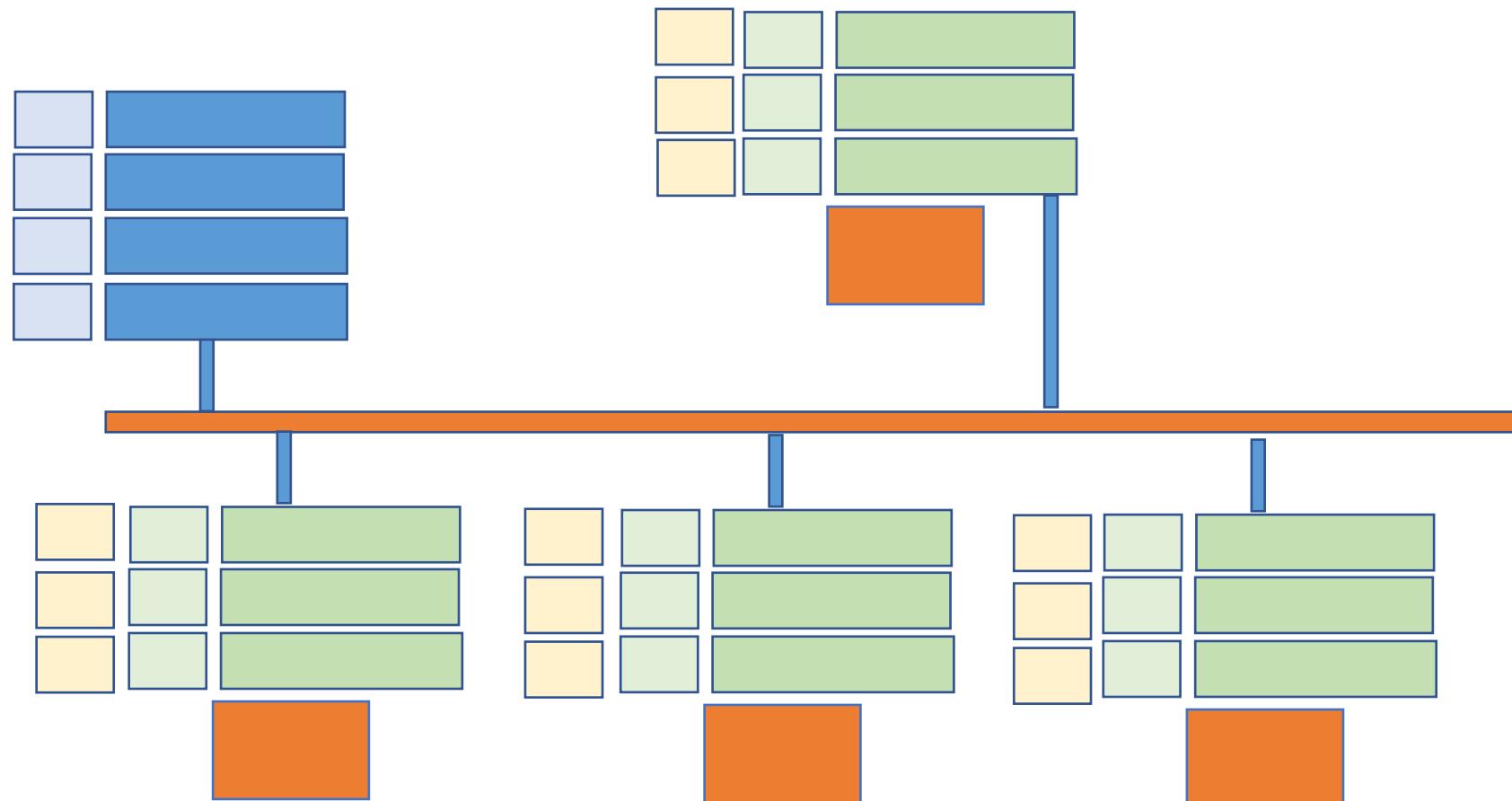
Processor gets what it wanted to read in the cache. (in a valid state obviously)

No change of state occurs here for that line

Gets the data from cache

Note :Having an invalid data is equivalent to having no data

Simulation Write Miss



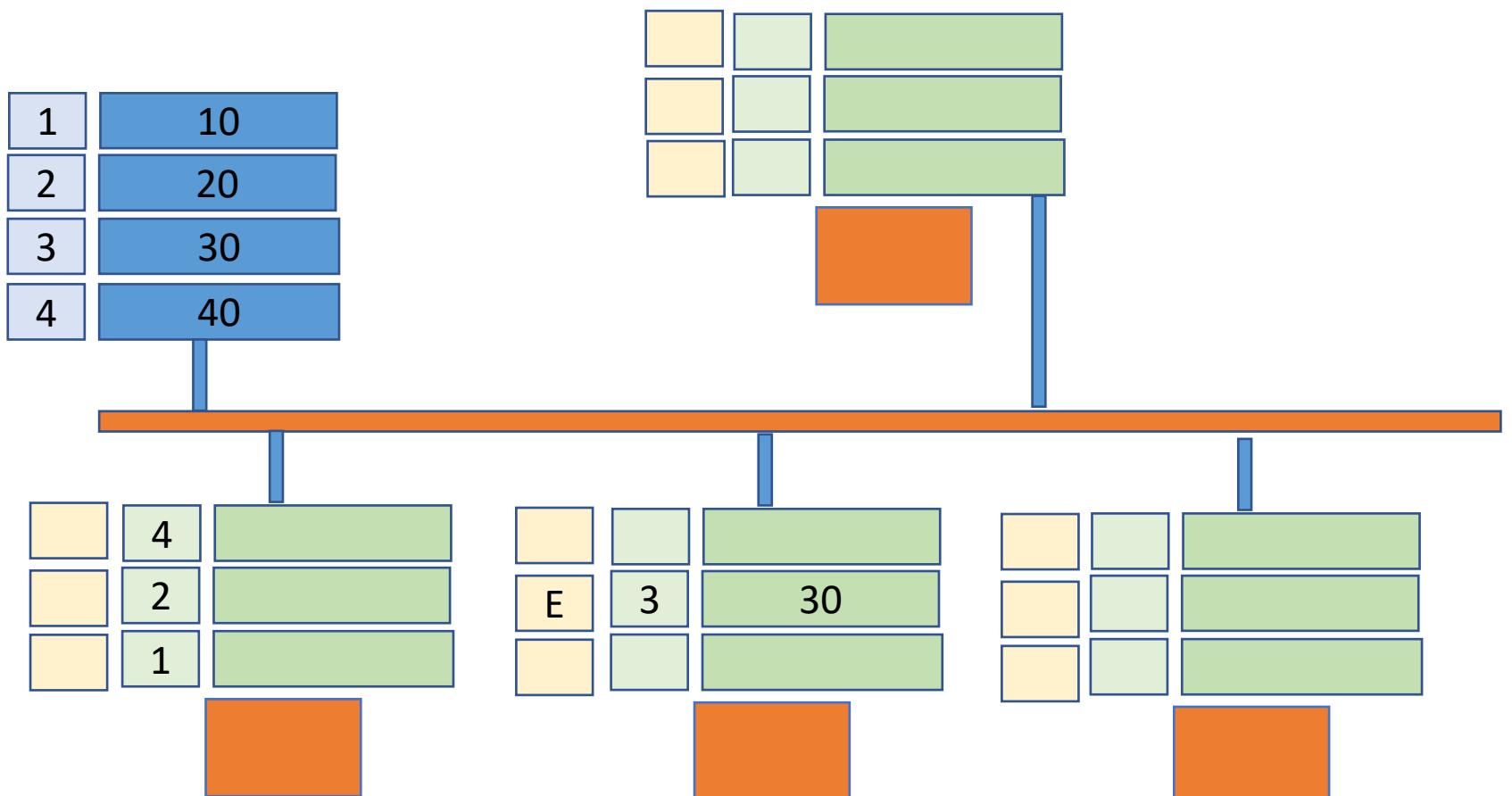
Write Miss:

The processor wanted to write a value in the cache. But the location for writing is not available or is invalid. Hence a signal is sent to the memory.

The signal is called Read with the Intent to Modify Line (RWITM).

The line is brought into the cache. Then the writing is done. Afterwards the line is marked as modified

Simulation Write Miss

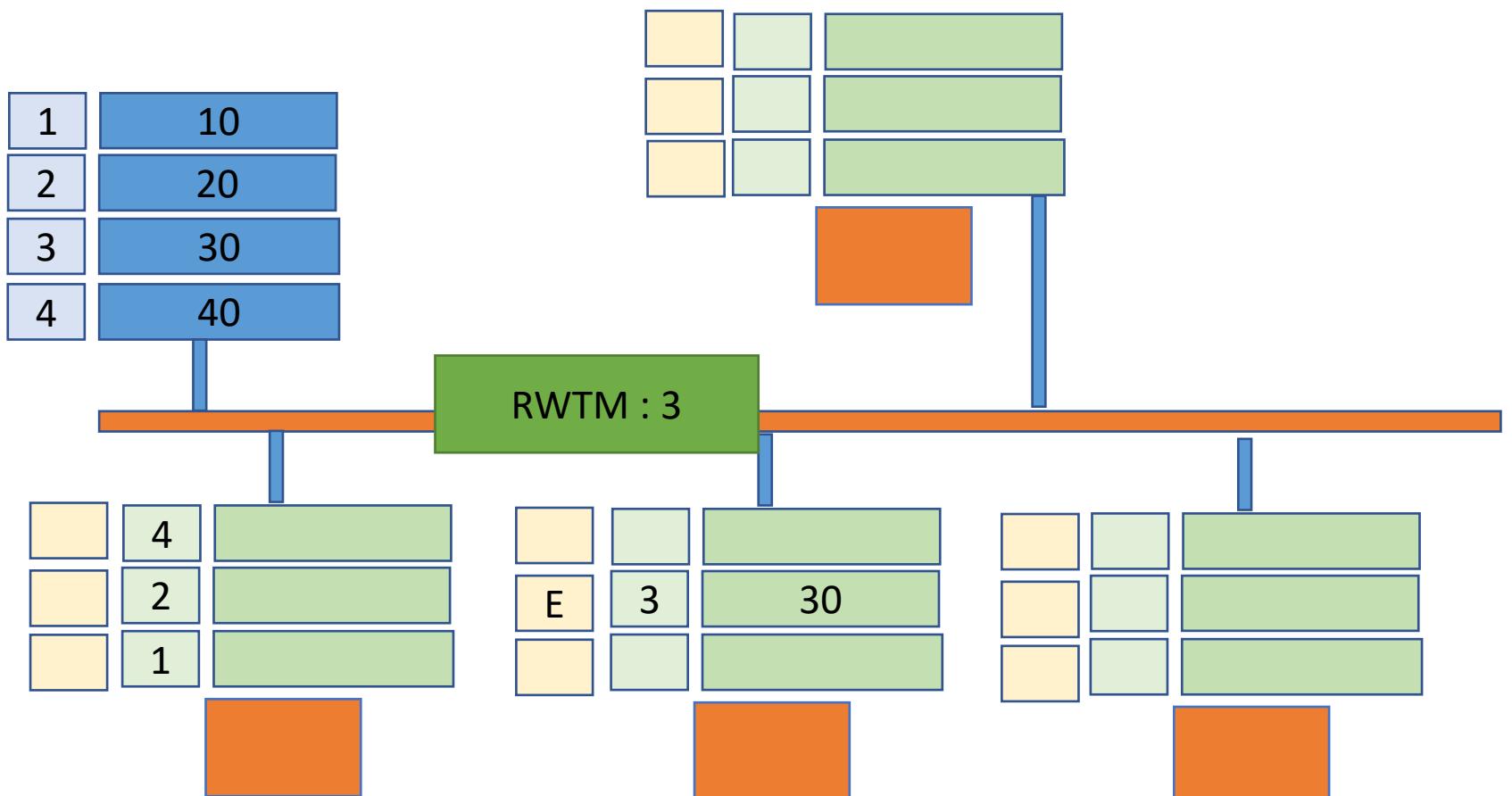


Wants to write to mem locn 3

Case 1:

Another processor has the data in **Exclusive** state

Simulation Write Miss

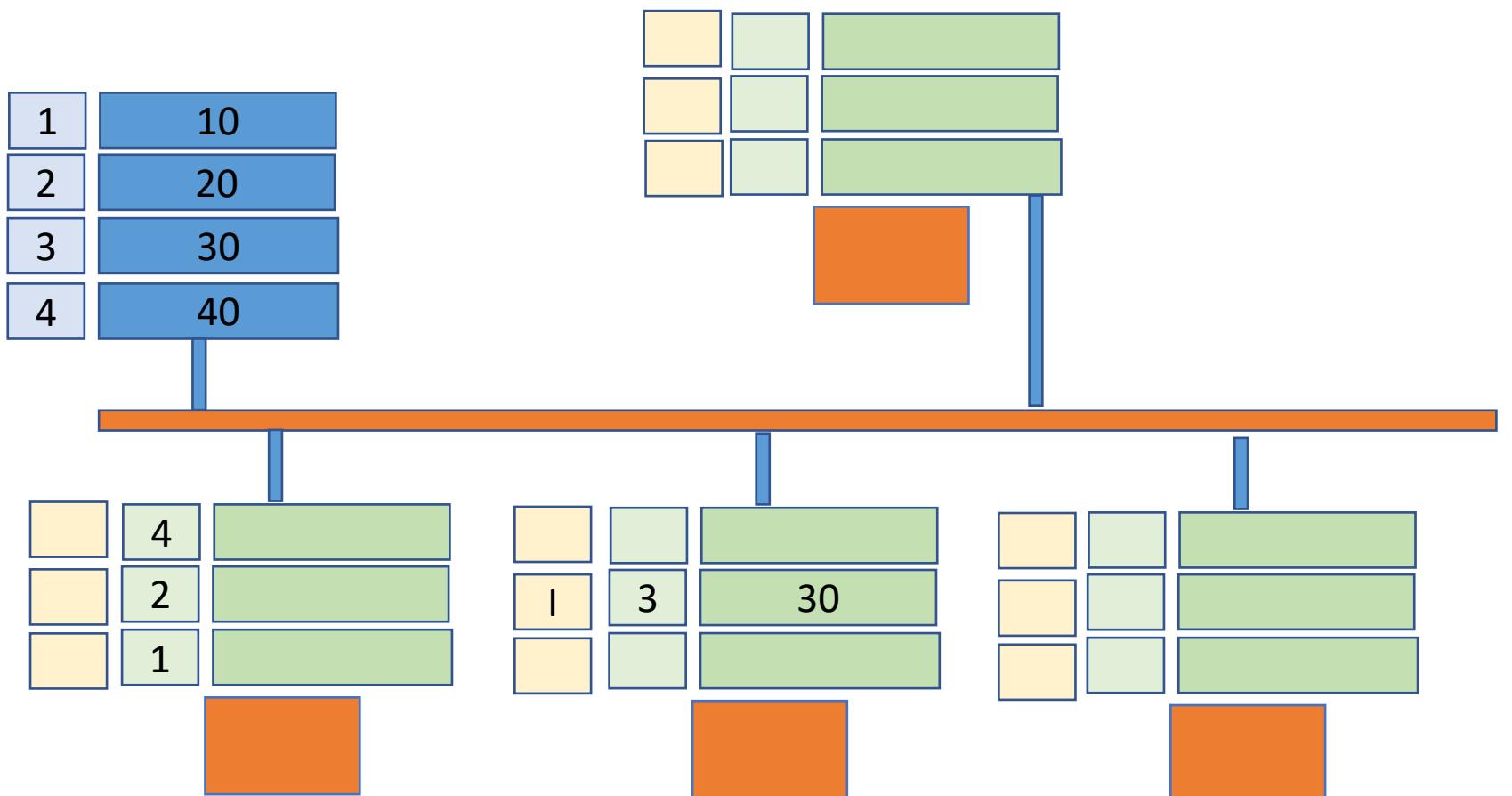


Wants to write to mem locn 3

Case 1:

Another processor has the data in Exclusive state

Simulation Write Miss



Wants to write to mem locn 3

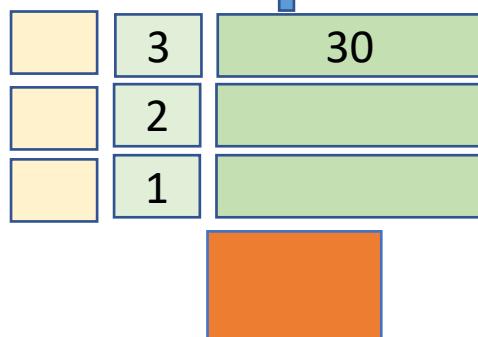
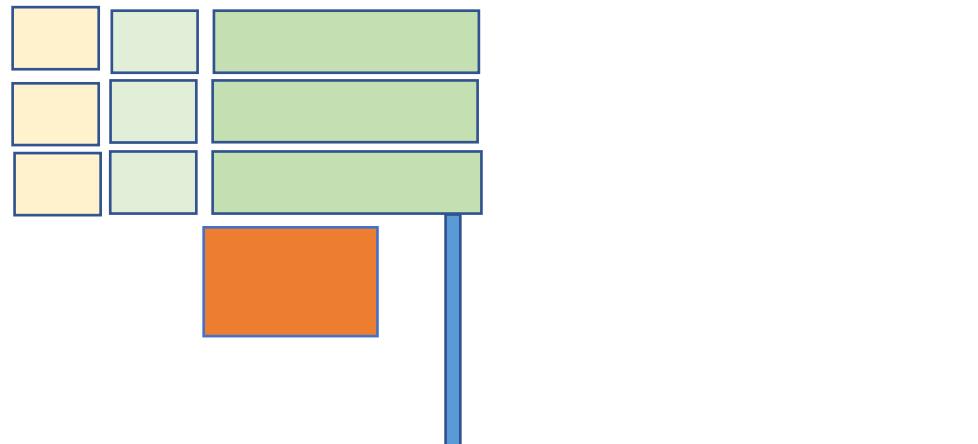
Invalidates is data

Case 1:

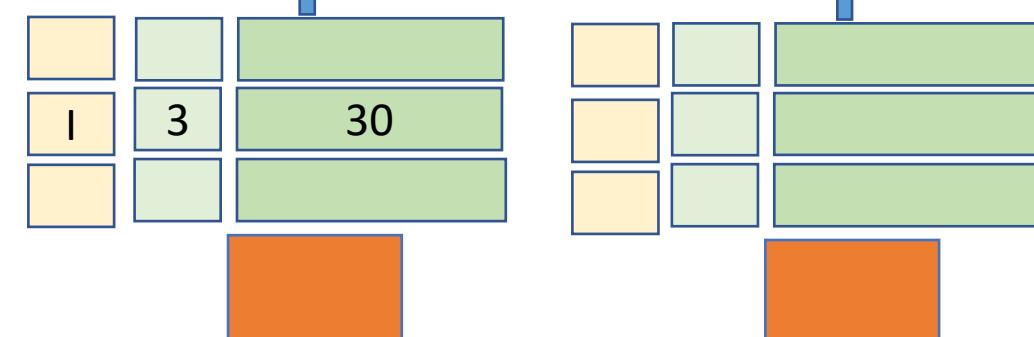
Another processor has the data in Exclusive state

Simulation Write Miss

| | |
|---|----|
| 1 | 10 |
| 2 | 20 |
| 3 | 30 |
| 4 | 40 |



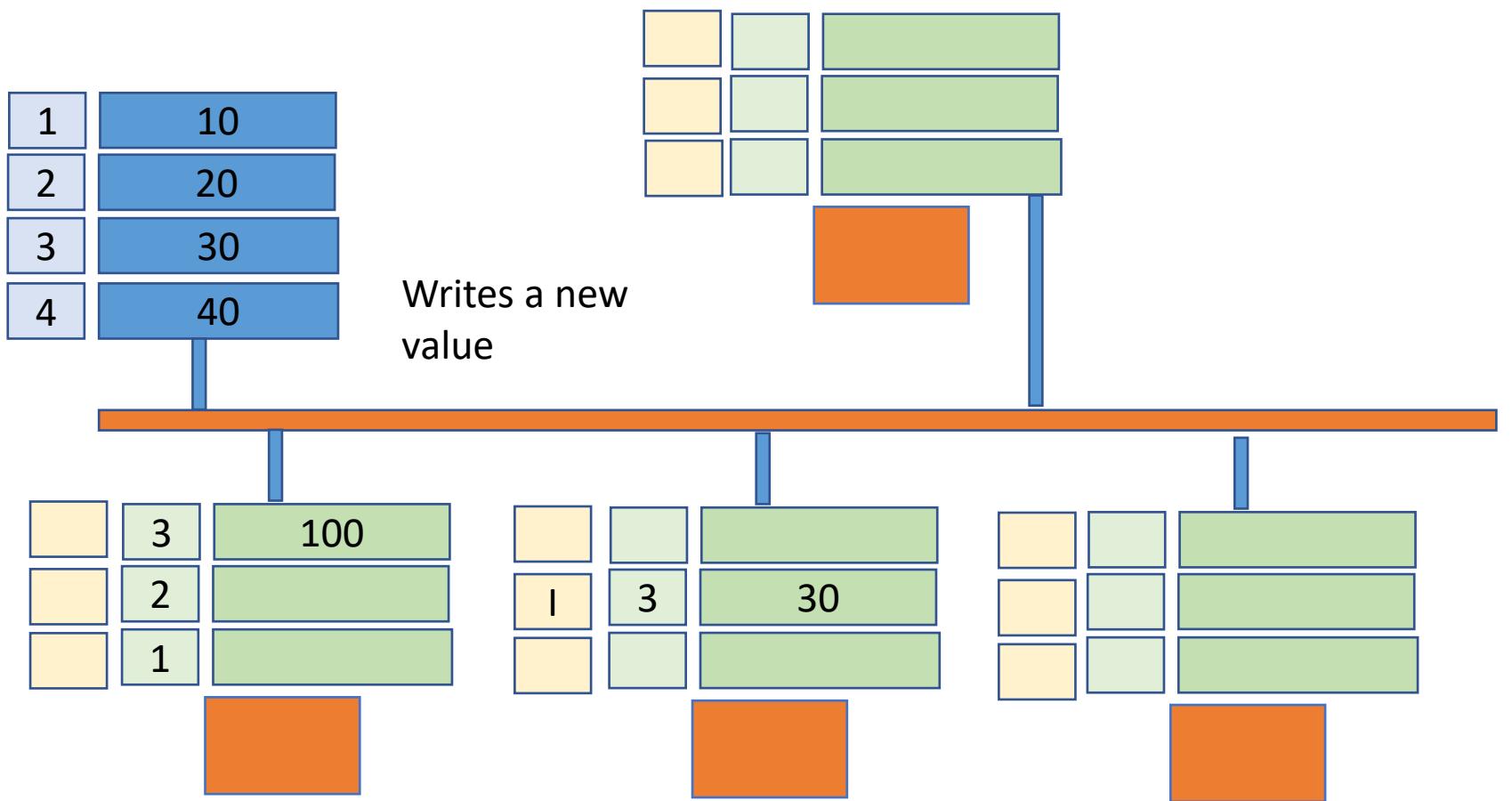
Wants to write to mem locn 3
Reads data from mem locn 3



Invalidates is data

Case 1:
Another processor has the data in Exclusive state

Simulation Write Miss



Wants to write to mem locn 3
Reads data from mem locn 3

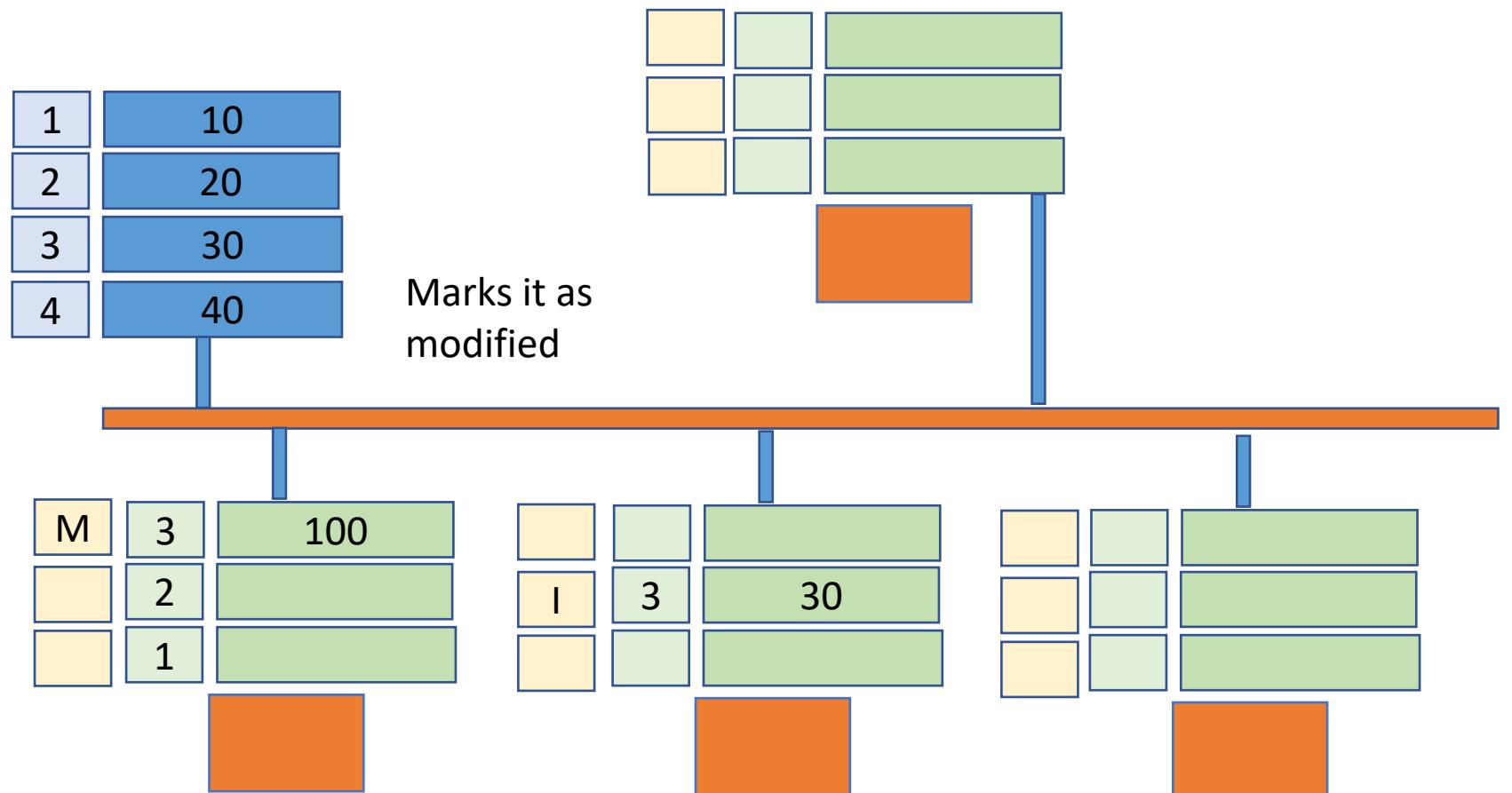
Invalidates its data

Case 1:
Another processor has
the data in Exclusive
state

Simulation Write Miss

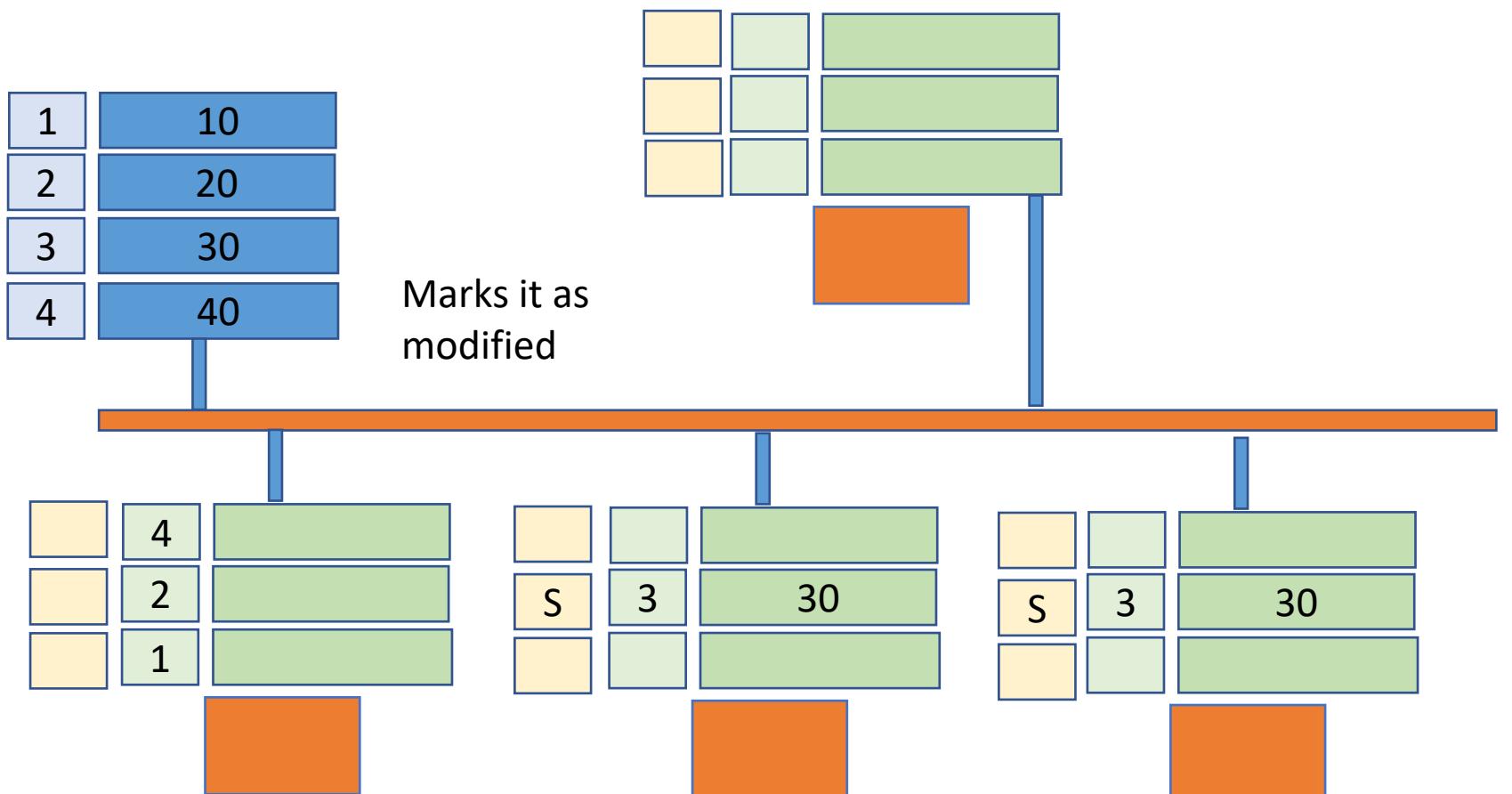
Case 1:

Another processor has
the data in Exclusive
state



Wants to write to mem locn 3
Reads data from mem locn 3

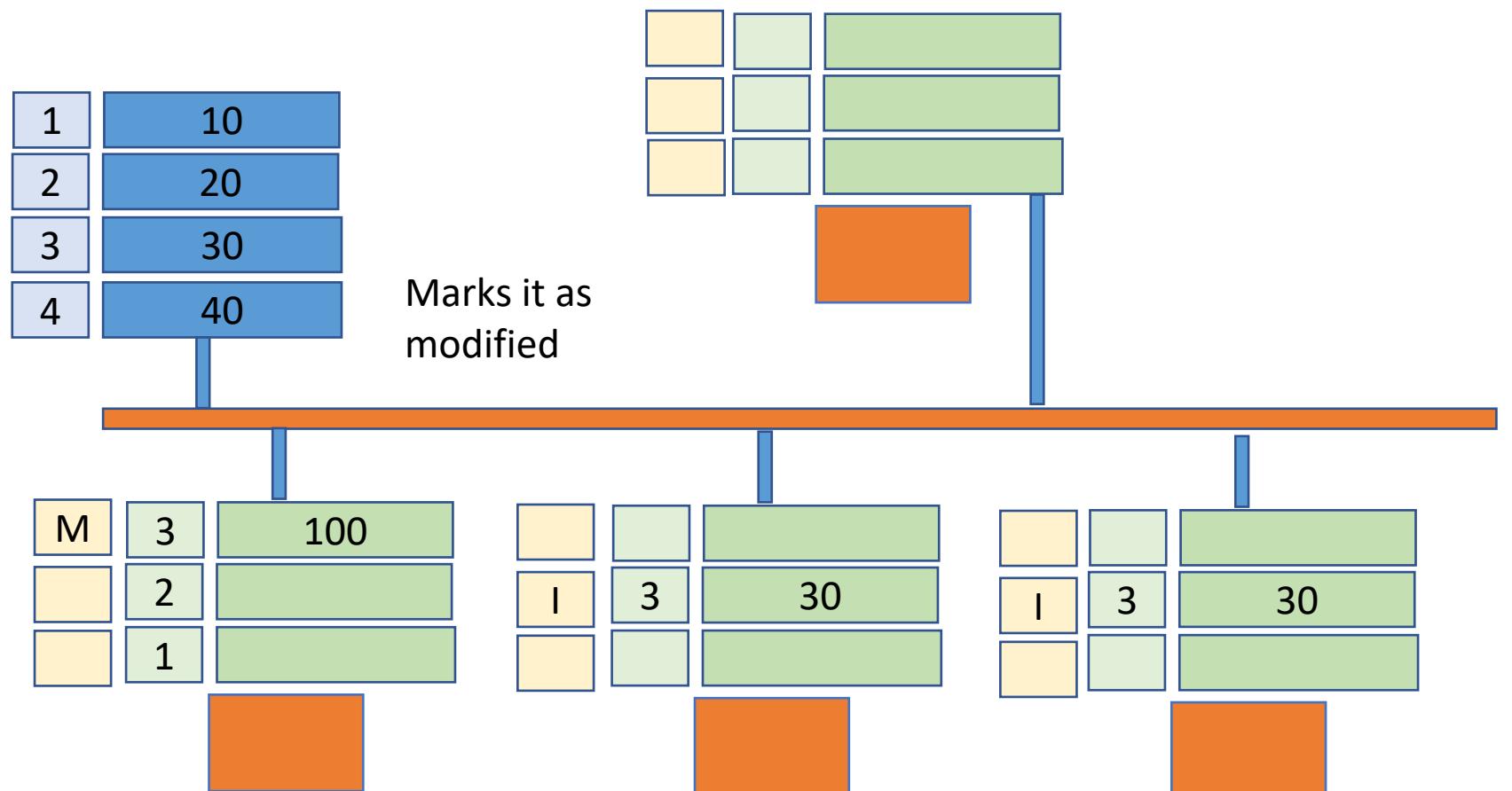
Simulation Write Miss



Case 2:

This is similar to case 1. Here the processors having the shared state will become invalid and the processor 1 will read data from location 3 and mark it as modified.

Simulation Write Miss



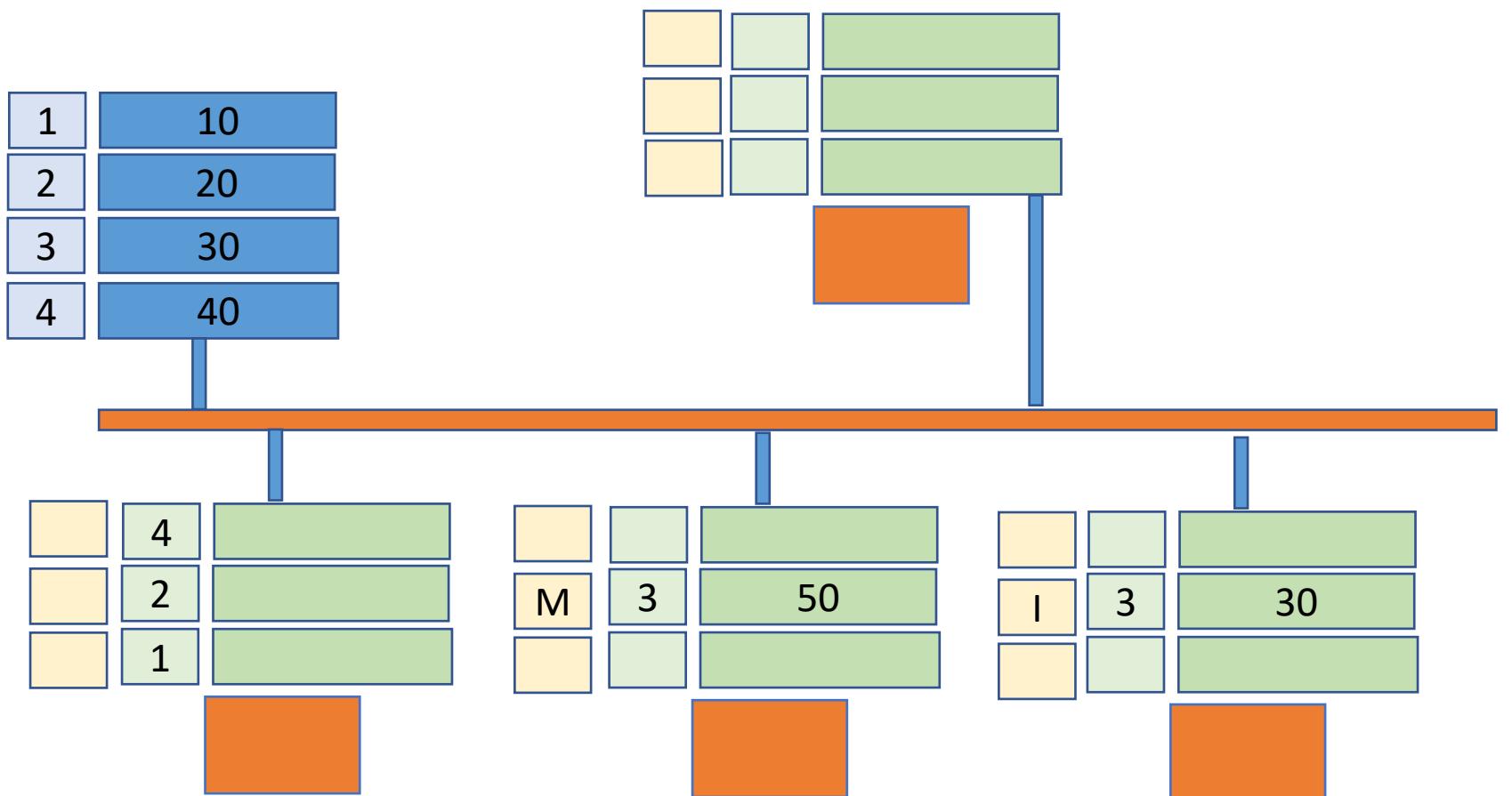
Case 2:

This is similar to case 1. Here the processors having the shared state will become invalid and the processor 1 will read data from location 3 and mark it as modified.

Wants to write to mem locn 3
Reads data from mem locn 3

Invalidates is data

Simulation Write Miss

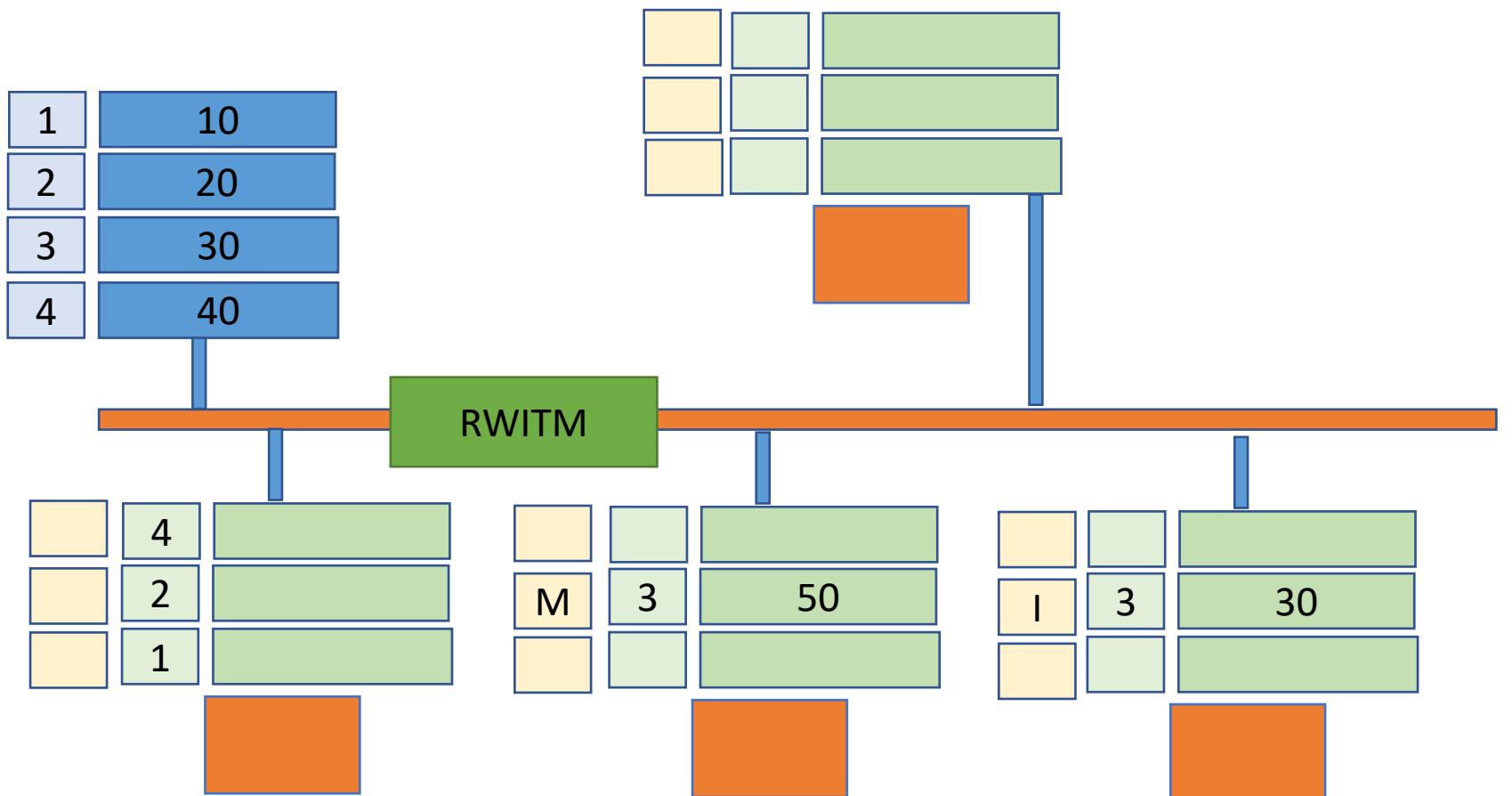


Case 3:

Another processor has the data in modified state.

Wants to write to mem locn 3

Simulation Write Miss



Wants to write to mem locn 3

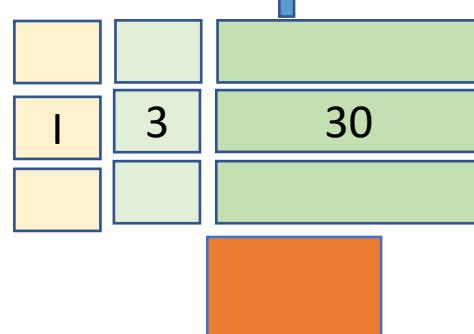
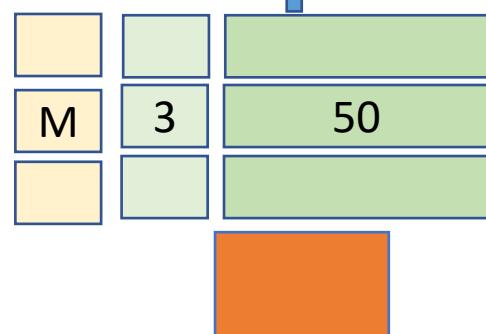
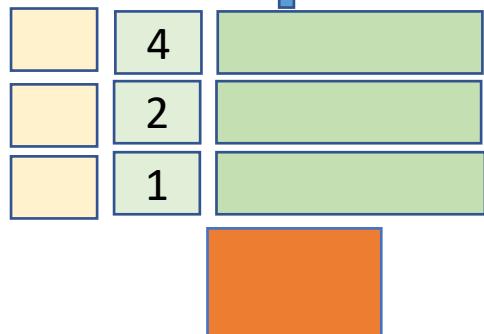
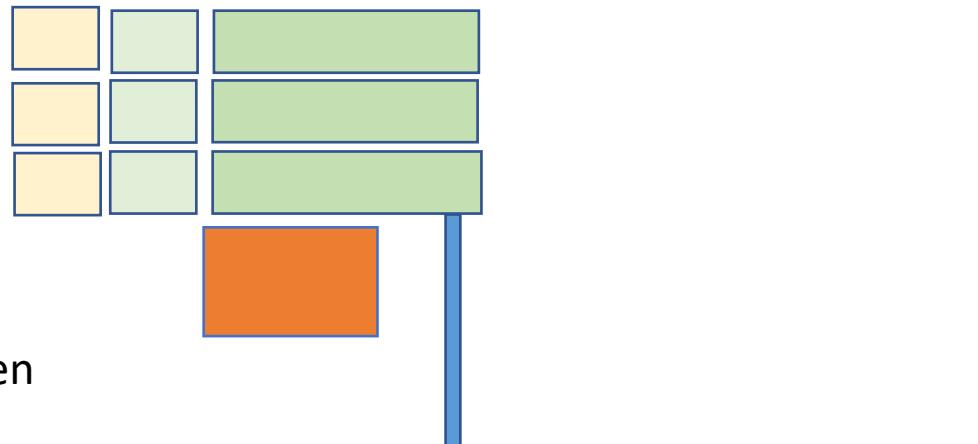
Case 3:

Another processor has the data in modified state.

Simulation Write Miss

| | |
|---|----|
| 1 | 10 |
| 2 | 20 |
| 3 | 50 |
| 4 | 40 |

The modified
Value is written
to location 3



Wants to write to mem locn 3

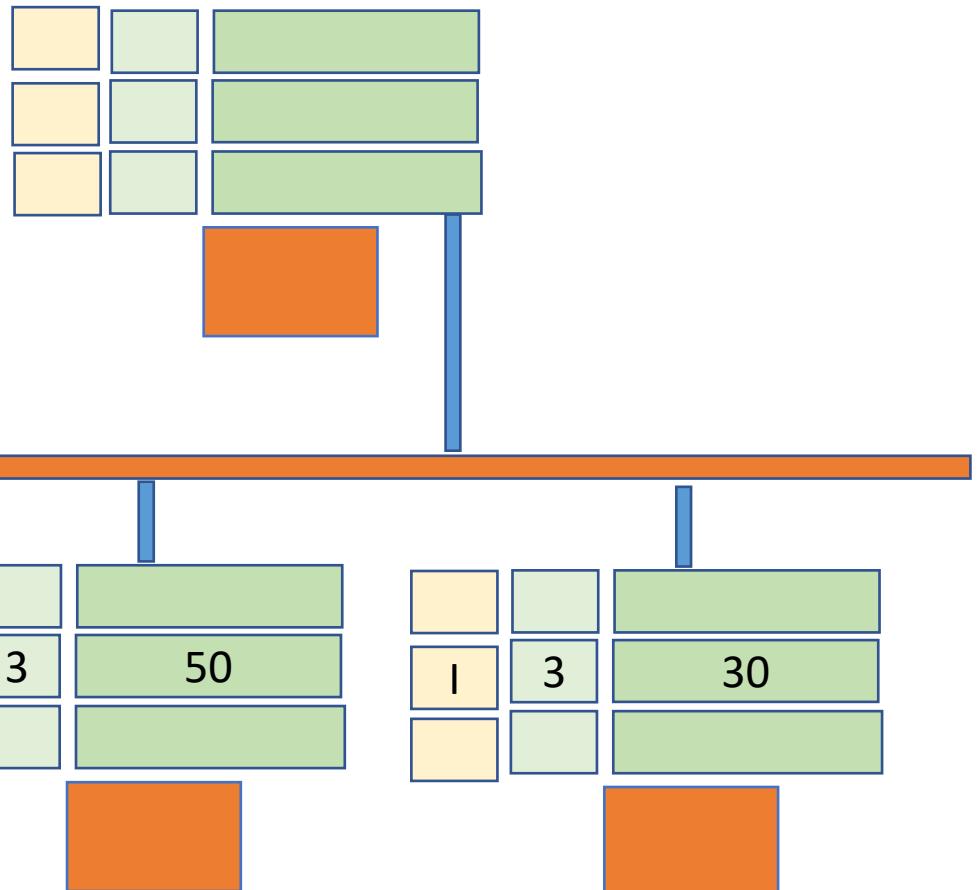
Case 3:

Another processor has
the data in modified
state.

Simulation Write Miss

| | |
|---|----|
| 1 | 10 |
| 2 | 20 |
| 3 | 50 |
| 4 | 40 |

The value is
invalidated in
P2



Case 3:

Another processor has
the data in modified
state.

Wants to write to mem locn 3

Simulation Write Miss

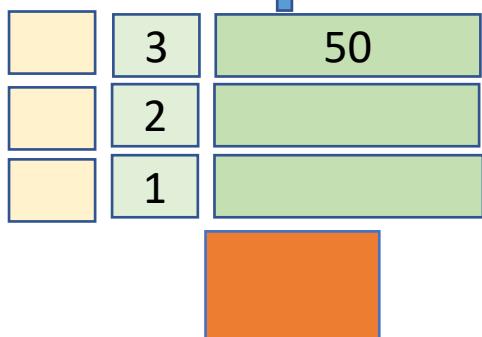
Case 3:

Another processor has the data in modified state.

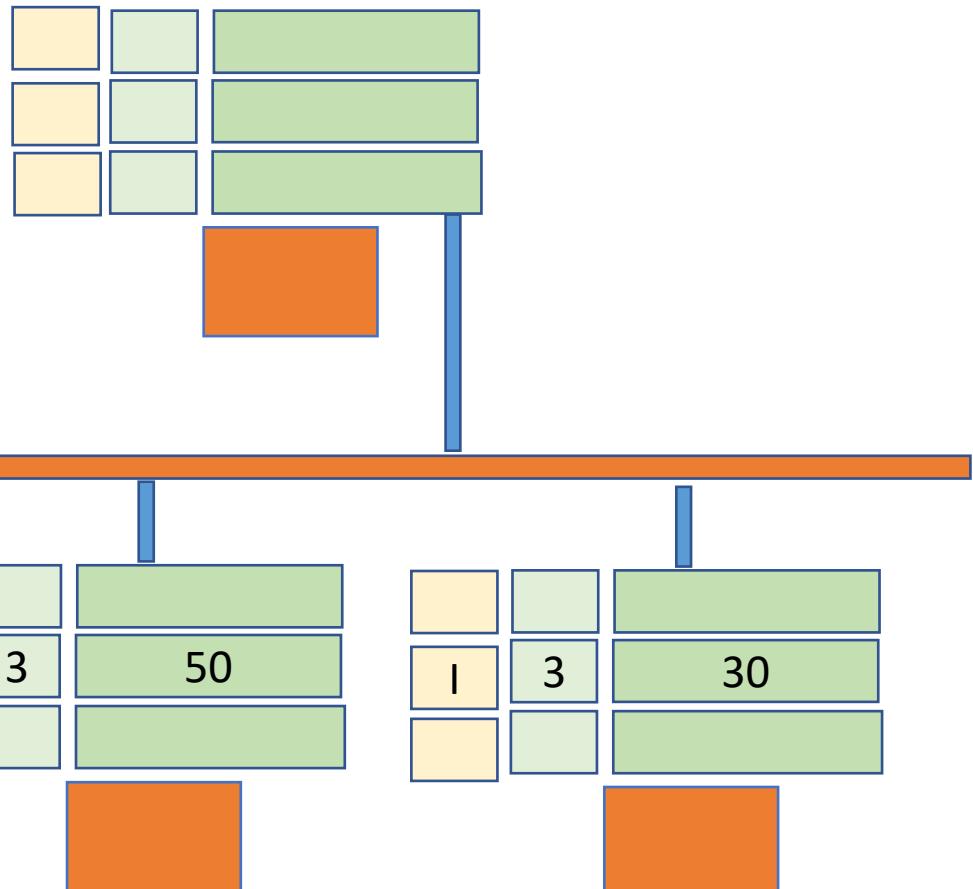
| | |
|---|----|
| 1 | 10 |
| 2 | 20 |
| 3 | 50 |
| 4 | 40 |

P1 :
The
Value in 3rd

location is
read



Wants to write to mem locn 3



Simulation Write Miss

Case 3:

Another processor has the data in modified state.

| | |
|---|----|
| 1 | 10 |
| 2 | 20 |
| 3 | 50 |
| 4 | 40 |

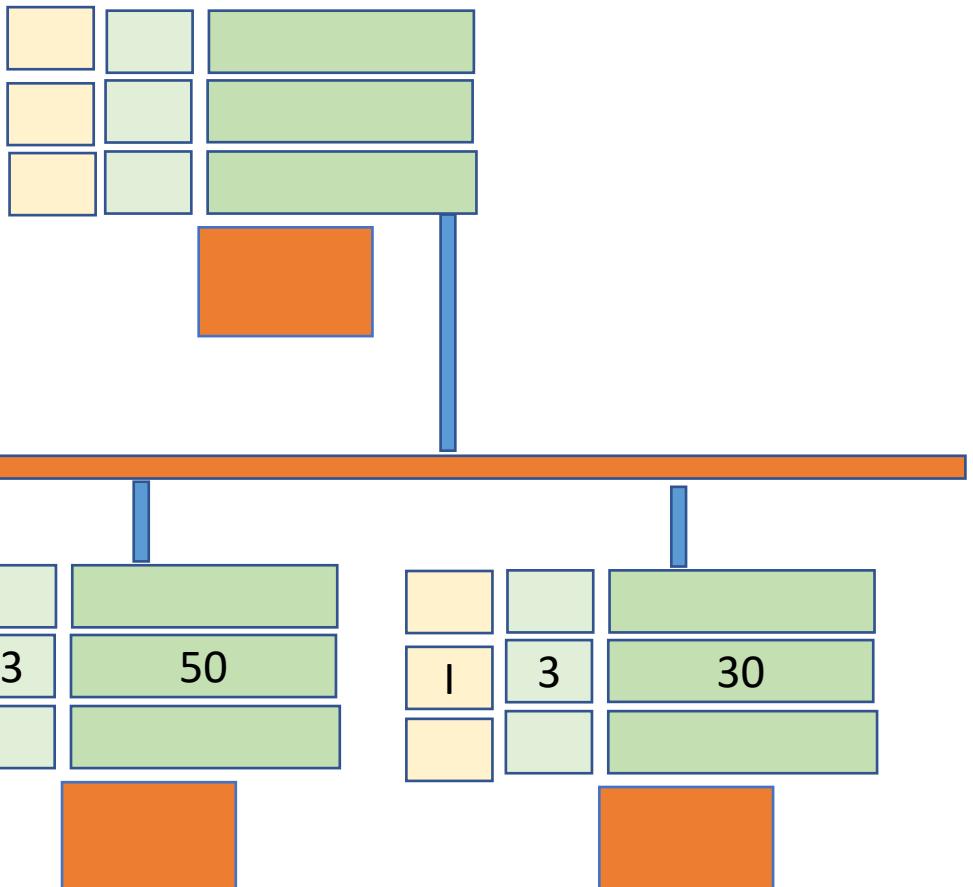
P1 :
The
Value in 3rd
location is
read

| | | |
|---|---|-----|
| M | 3 | 150 |
| 2 | | |
| 1 | | |

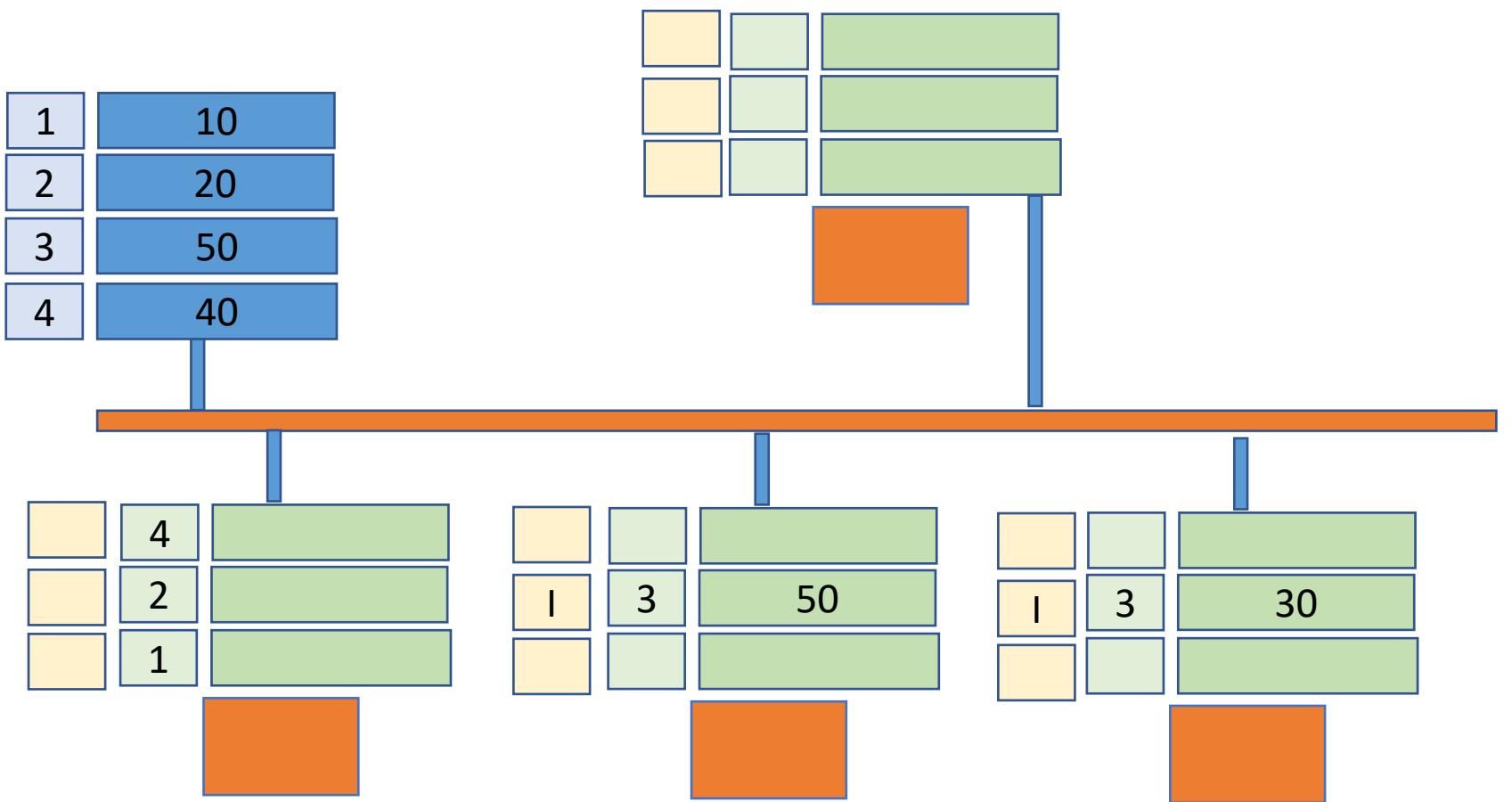
It is changed
and marked
as modified

Wants to write to mem locn 3

The value is
invalidated in
P2



Simulation Write Miss

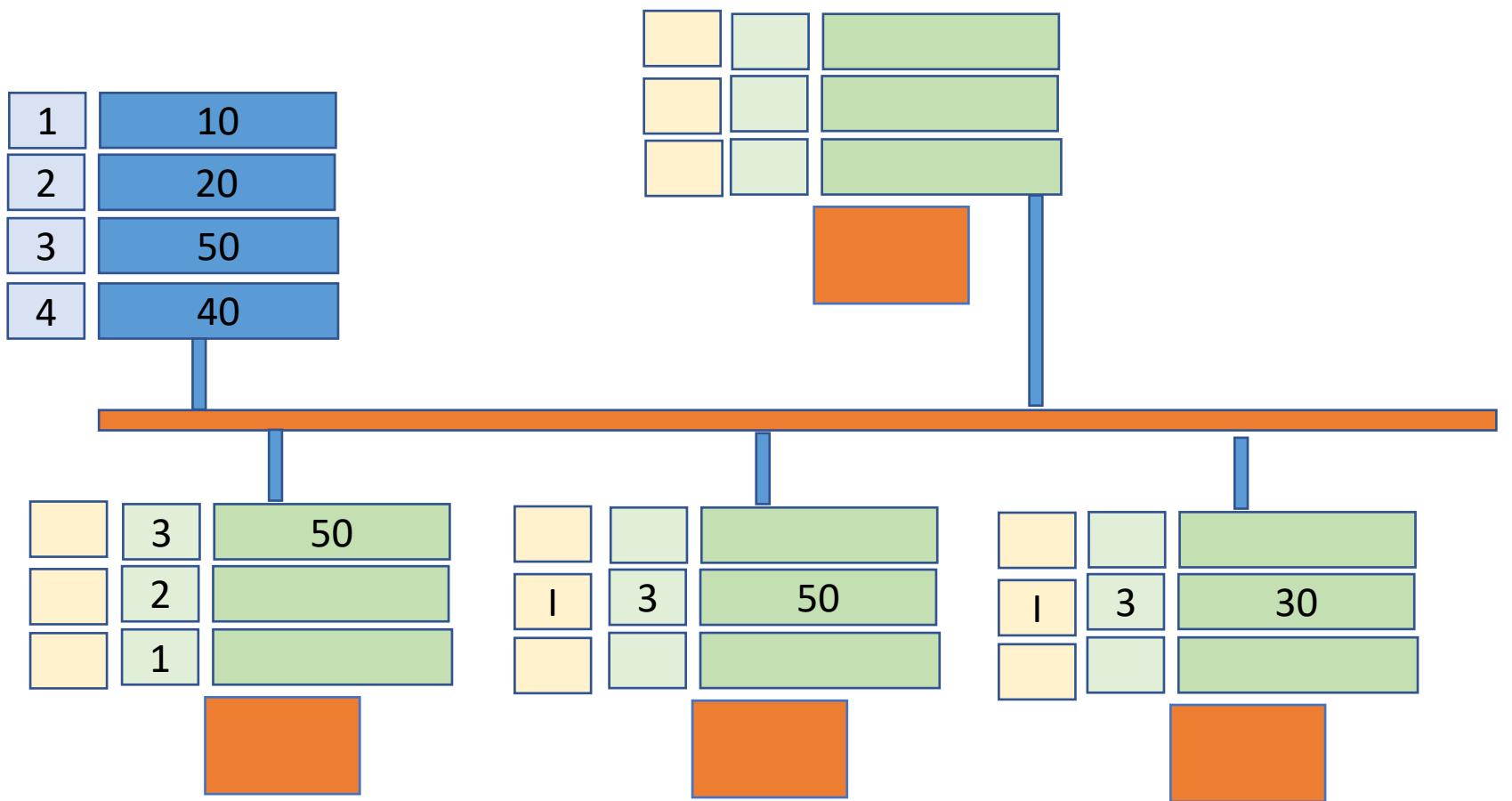


Wants to write to mem locn 3

Case 4:

No other processor has related data, or has invalid data

Simulation Write Miss

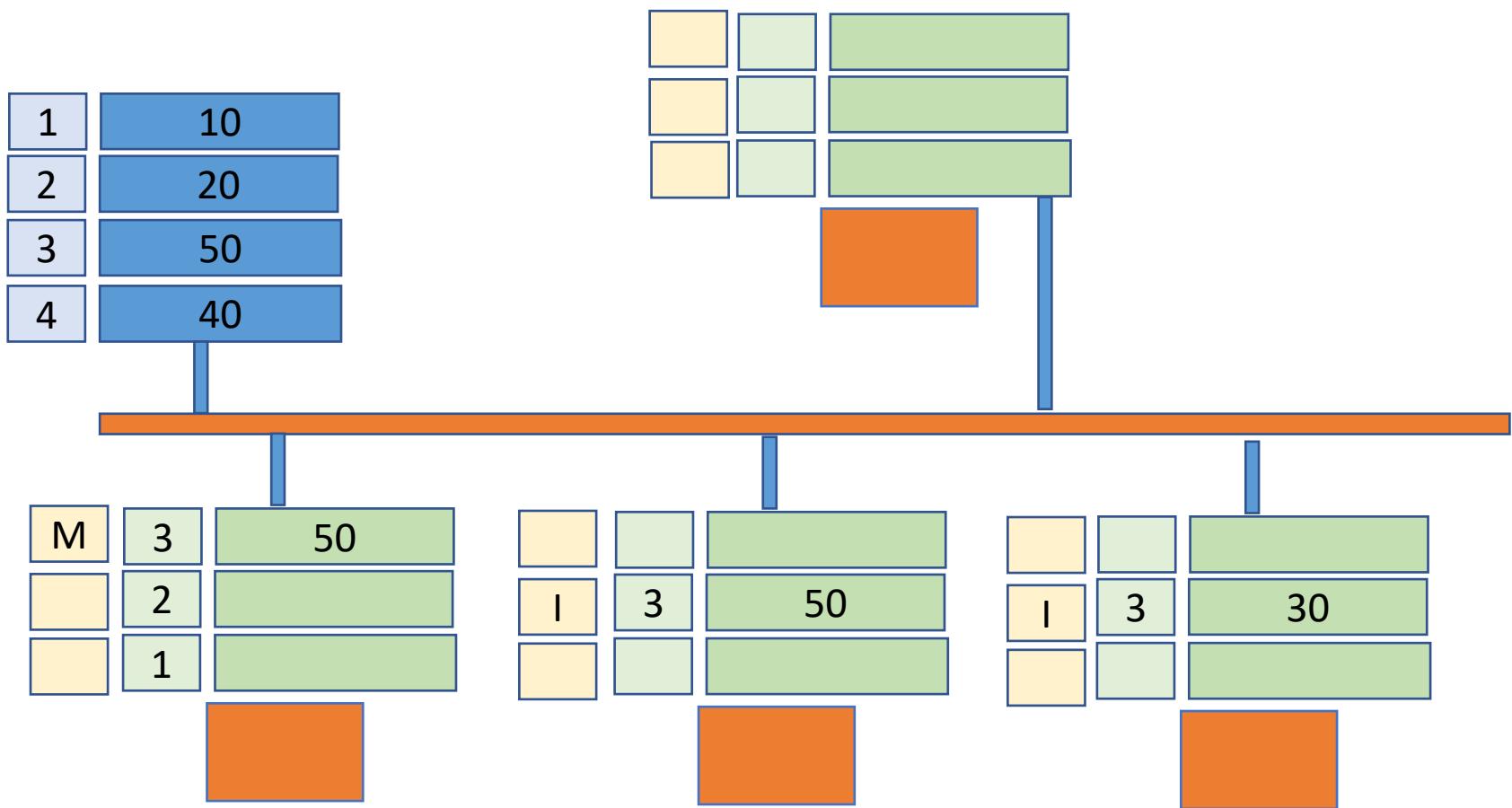


Wants to write to mem locn 3

Case 4:

No other processor has related data, or has invalid data

Simulation Write Miss



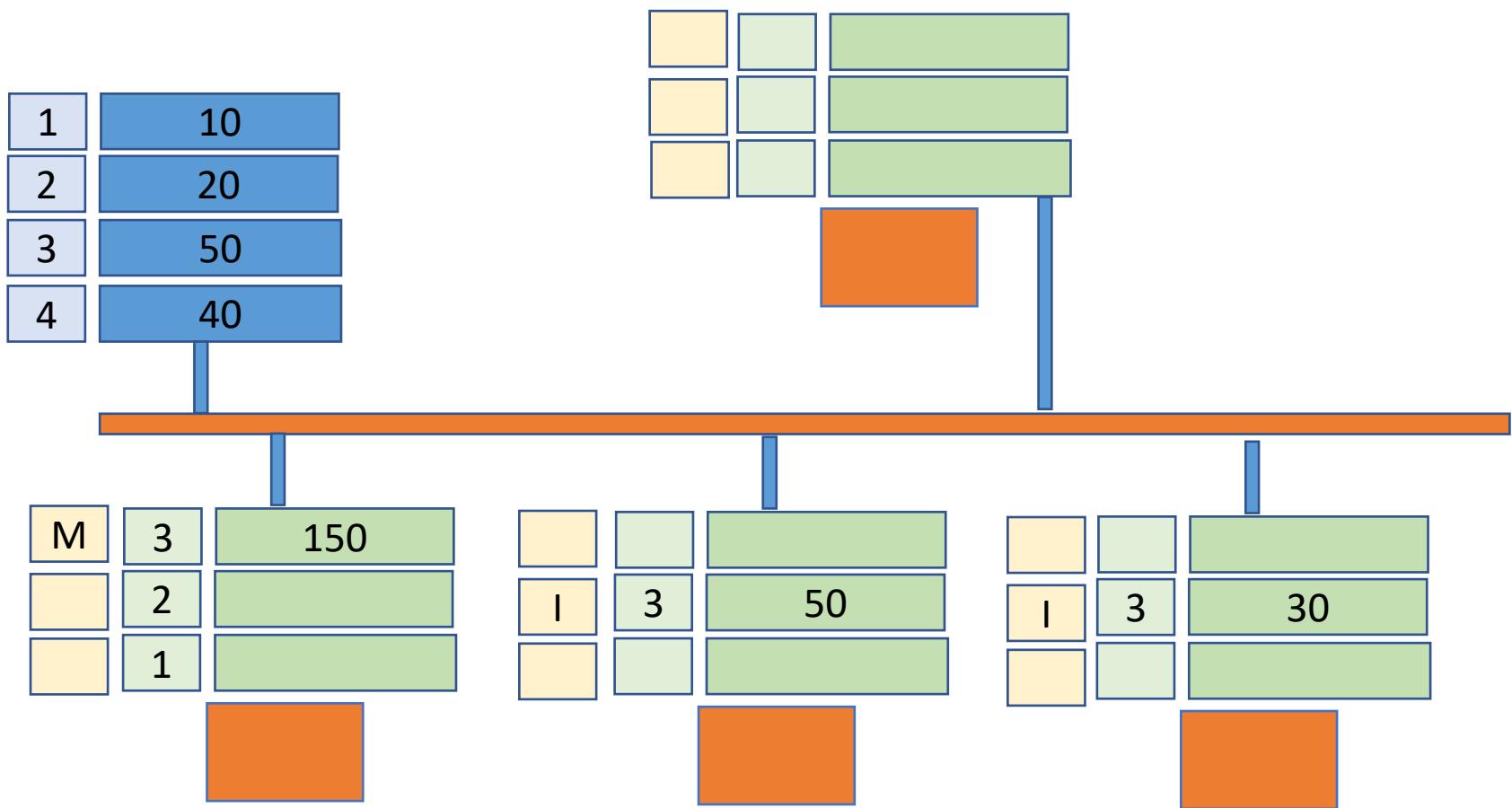
Case 4:

No other processor has related data, or has invalid data

Just reads in the value and
Changes it. Then marks it as
modified

Wants to write to mem locn 3

Simulation Write Miss



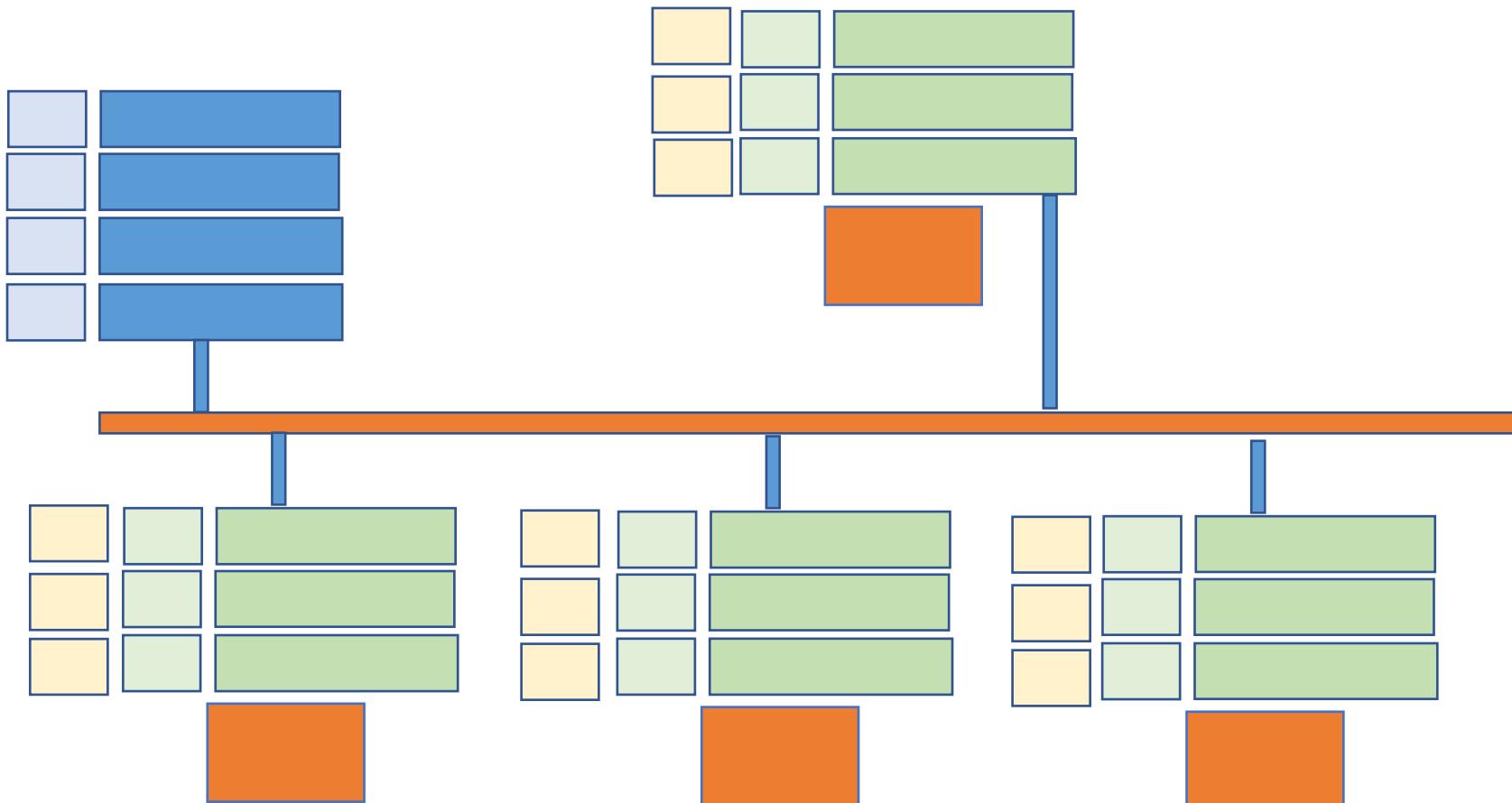
Case 4:

No other processor has related data, or has invalid data

Just reads in the value and
Changes it. Then marks it as
modified

Wants to write to mem locn 3

Simulation Write Hit



Write Hit:

The processor is able to write to the location in cache where it wanted to write. (the data there is valid, otherwise it would be a write miss)

Shared -

The shared line is also present in other processors. So a signal is sent by the processor. The other processors mark their data lines as invalid. And the initiating processor marks it as modified

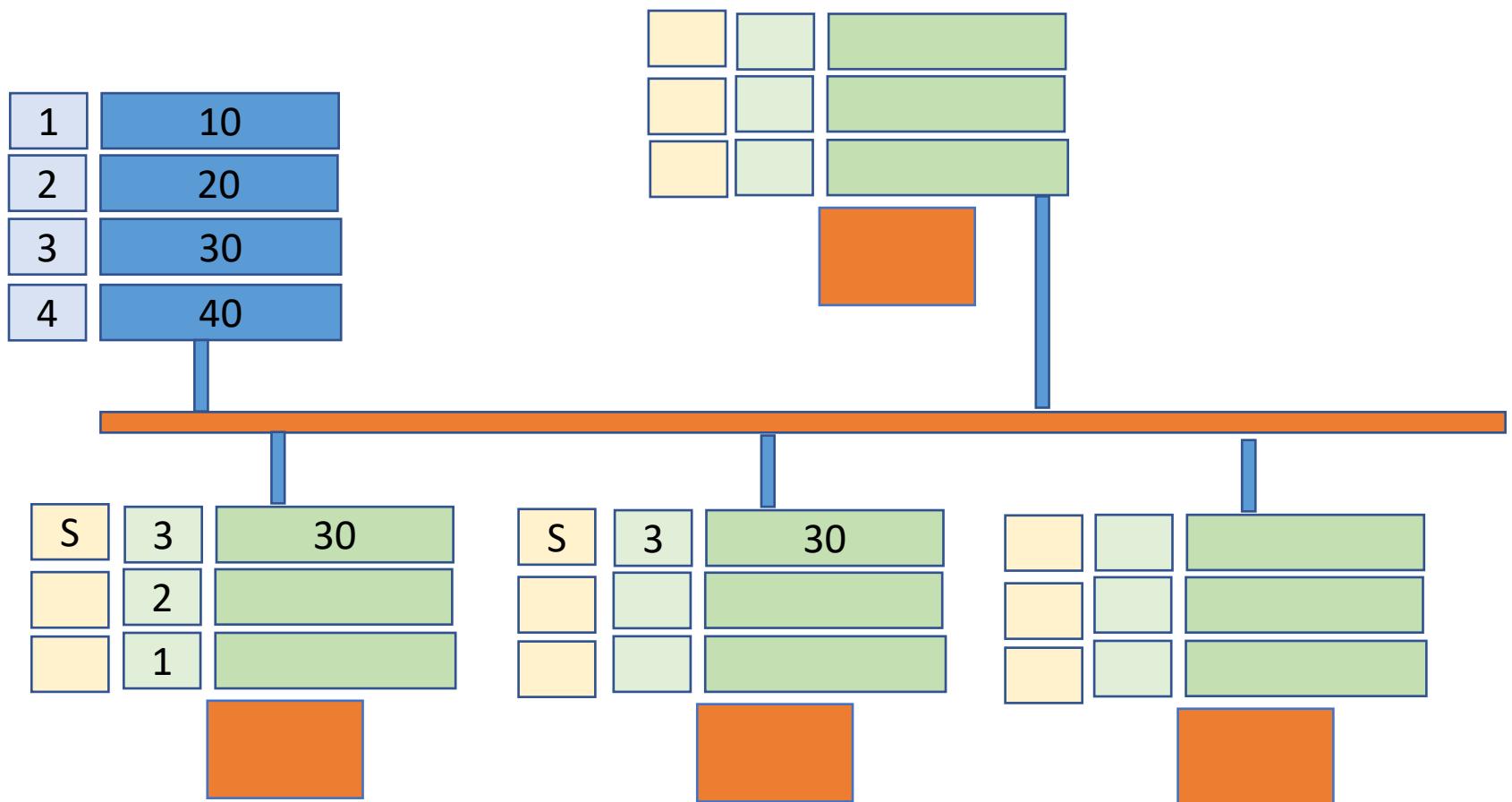
Exclusive -

Processor simply updates its own cache and marks it as modified

Modified - .

Processor simply updates its own cache and marks it as modified. The previously modified value is placed into the memory.

Simulation Write Hit

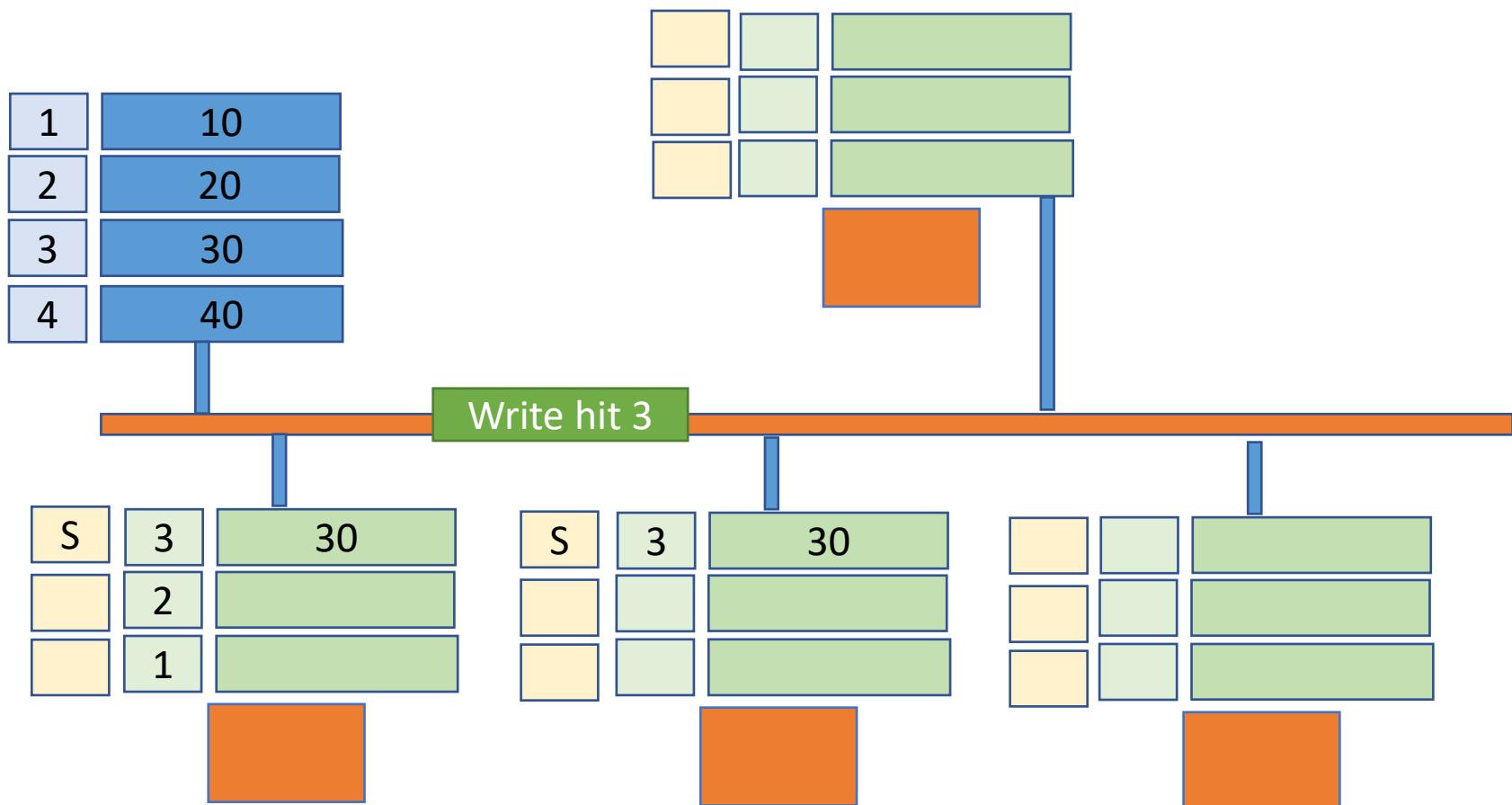


Wants to write to locn 3

Write Hit:

Case 1:
P1 has the data in shared state.

Simulation Write Hit

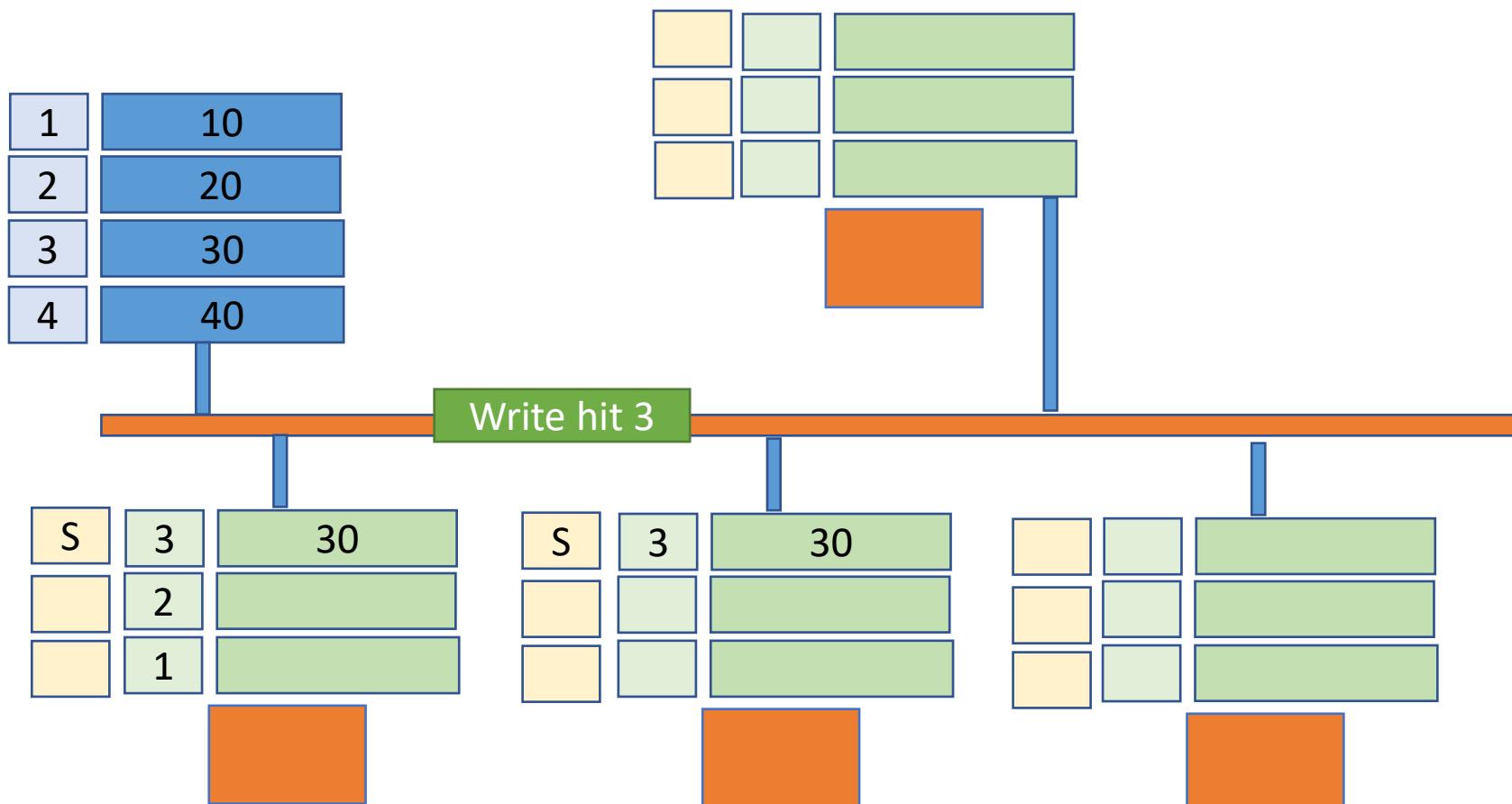


Wants to write to locn 3

Write Hit:

Case 1:
P1 has the data in shared state.

Simulation Write Hit



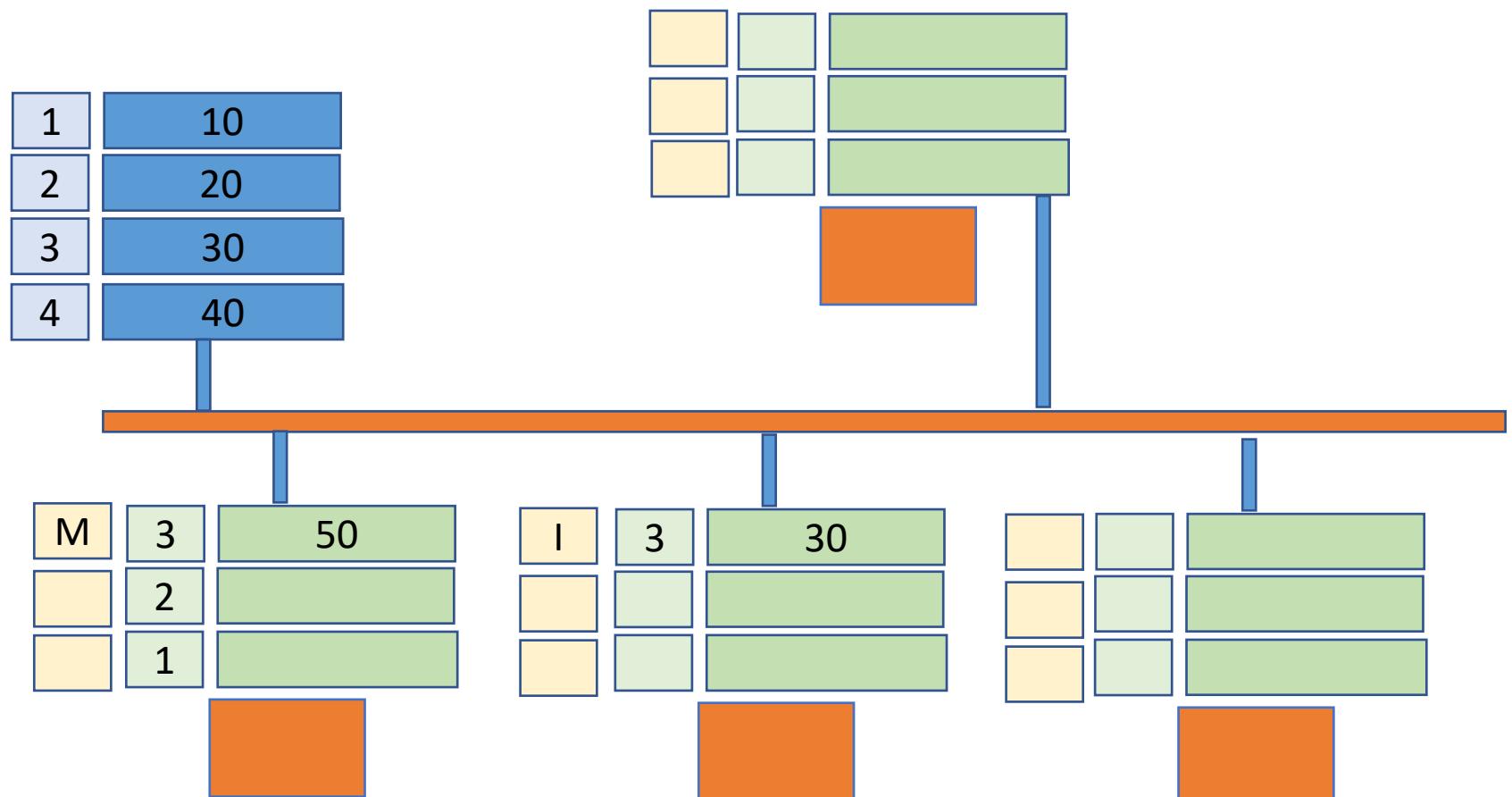
Wants to write to locn 3

It marks its location 3 as invalid

Write Hit:

Case 1:
P1 has the data in shared state.

Simulation Write Hit



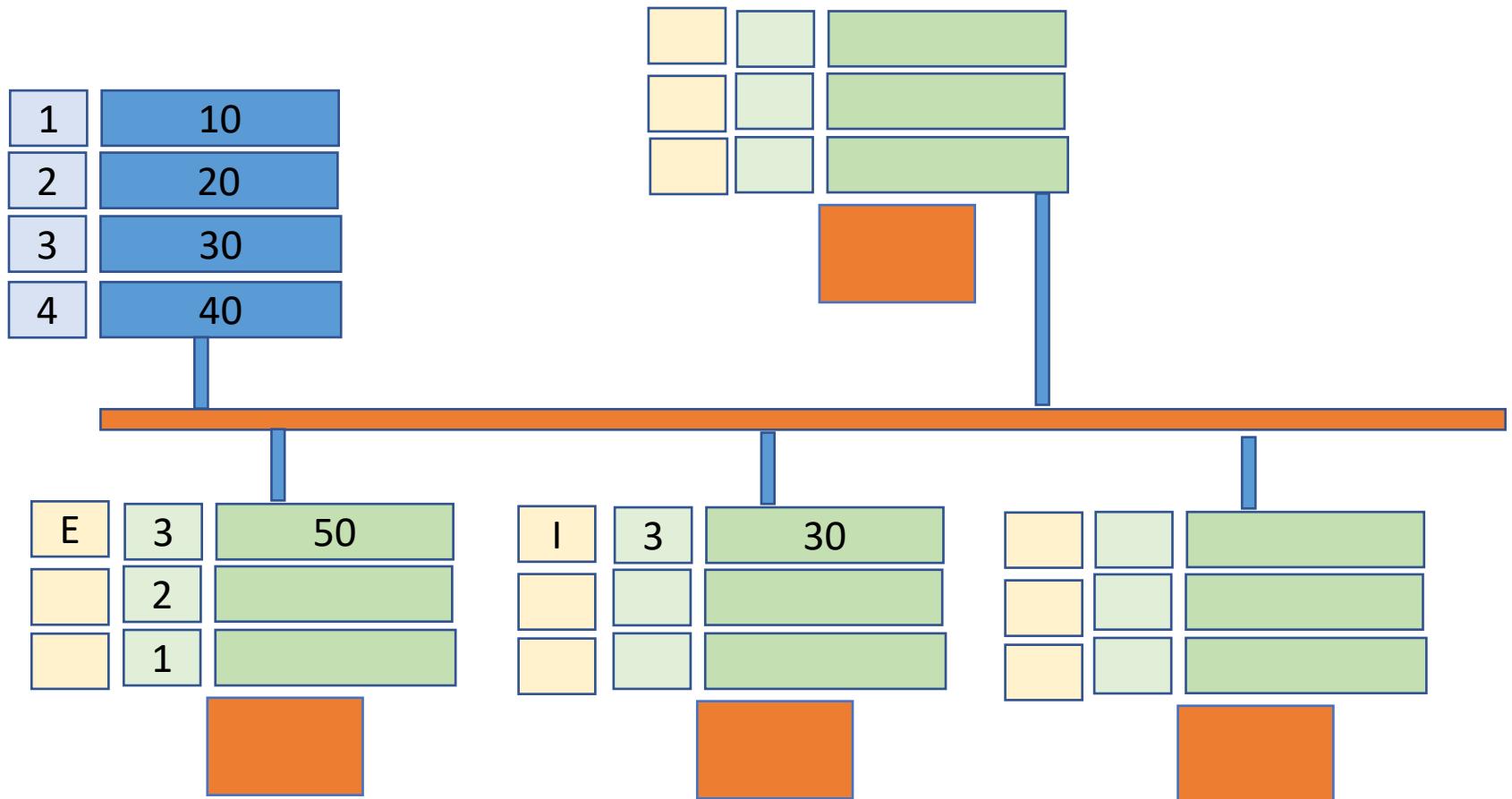
Wants to write to locn 3
Writes the new value, marks
As modified

It marks its location 3 as invalid

Write Hit:

Case 1:
P1 has the data in shared state.

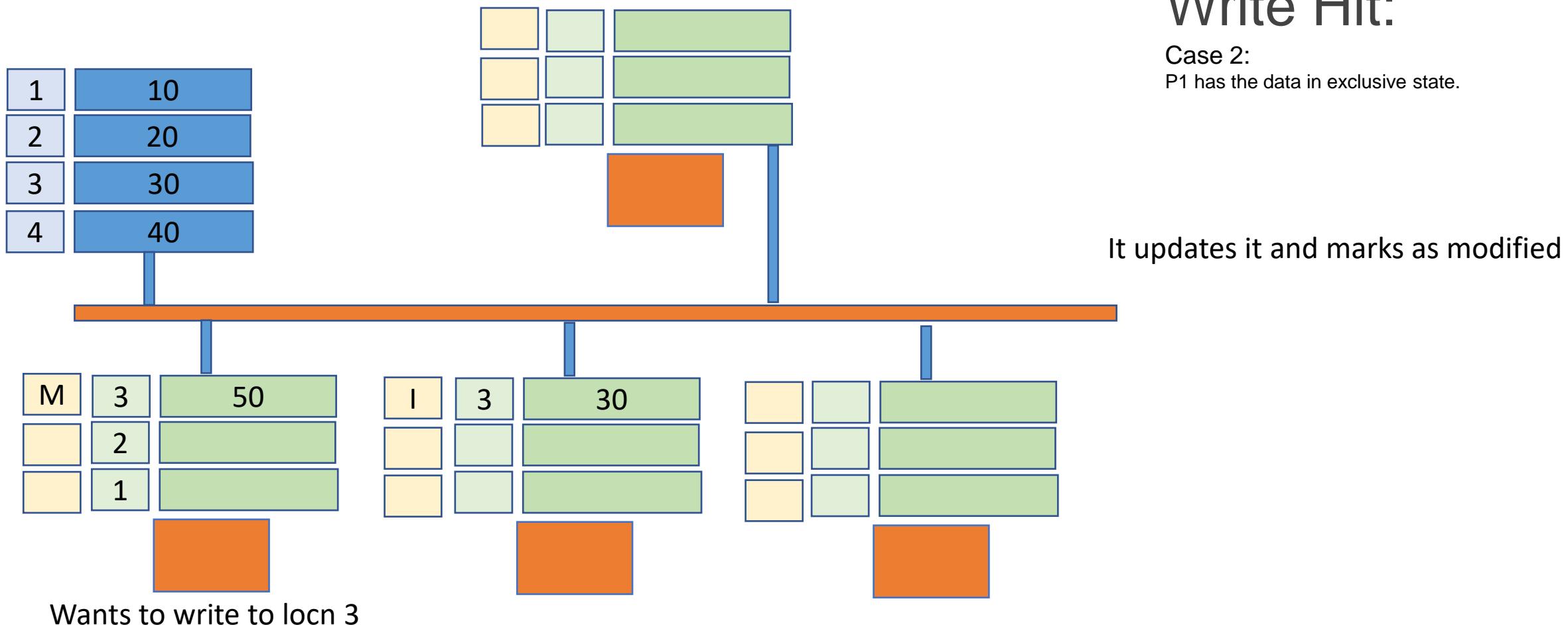
Simulation Write Hit



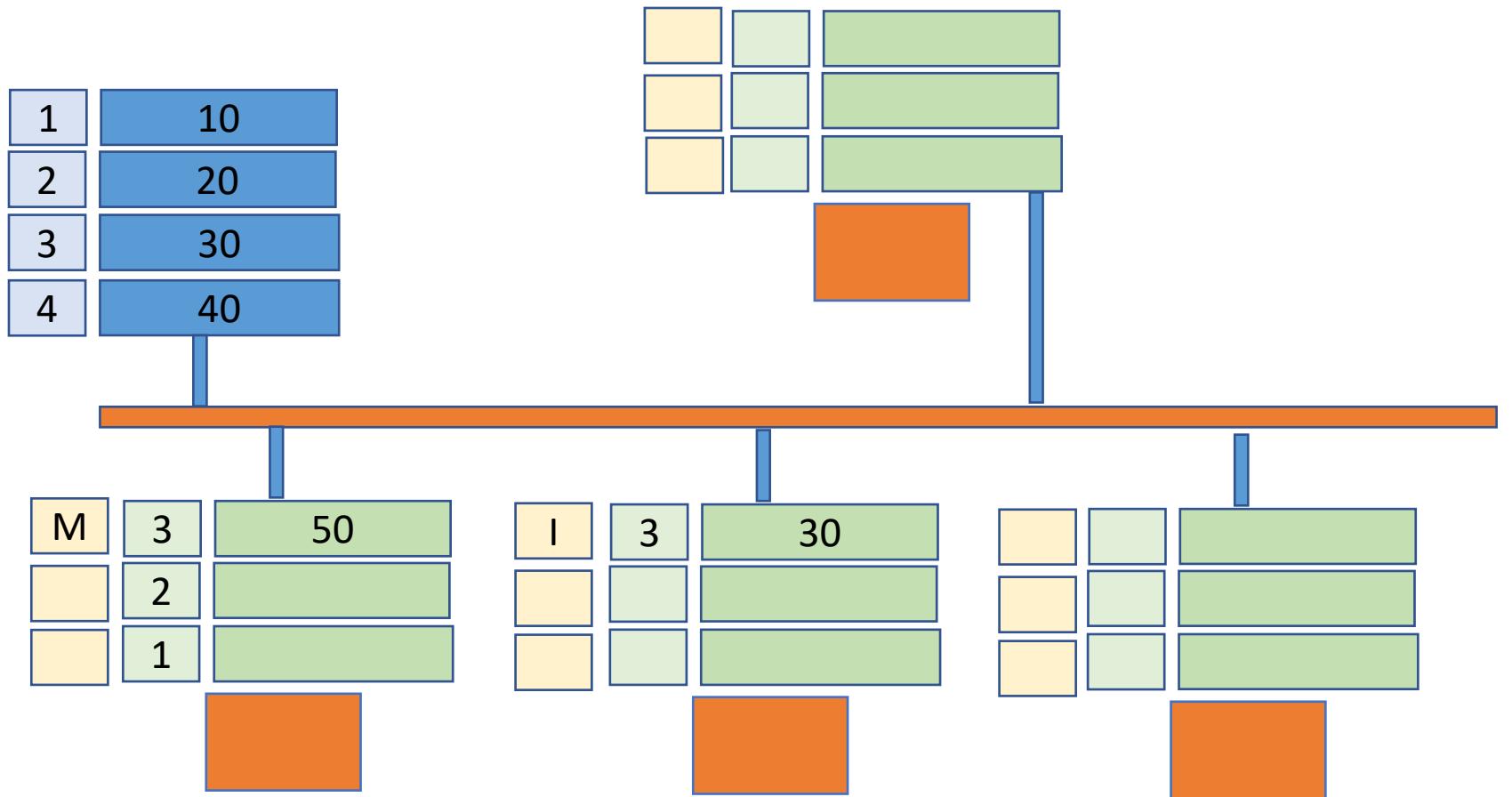
Write Hit:

Case 2:
P1 has the data in exclusive state.

Simulation Write Hit



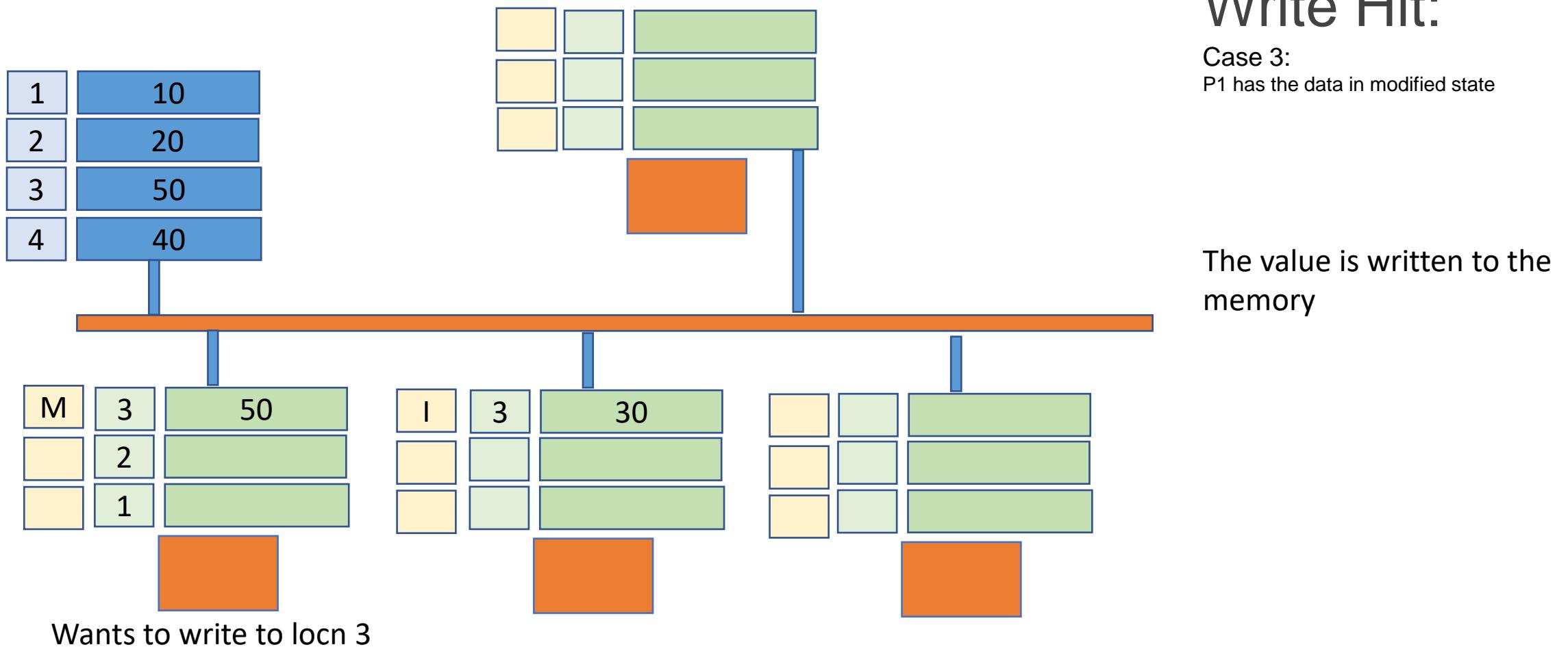
Simulation Write Hit



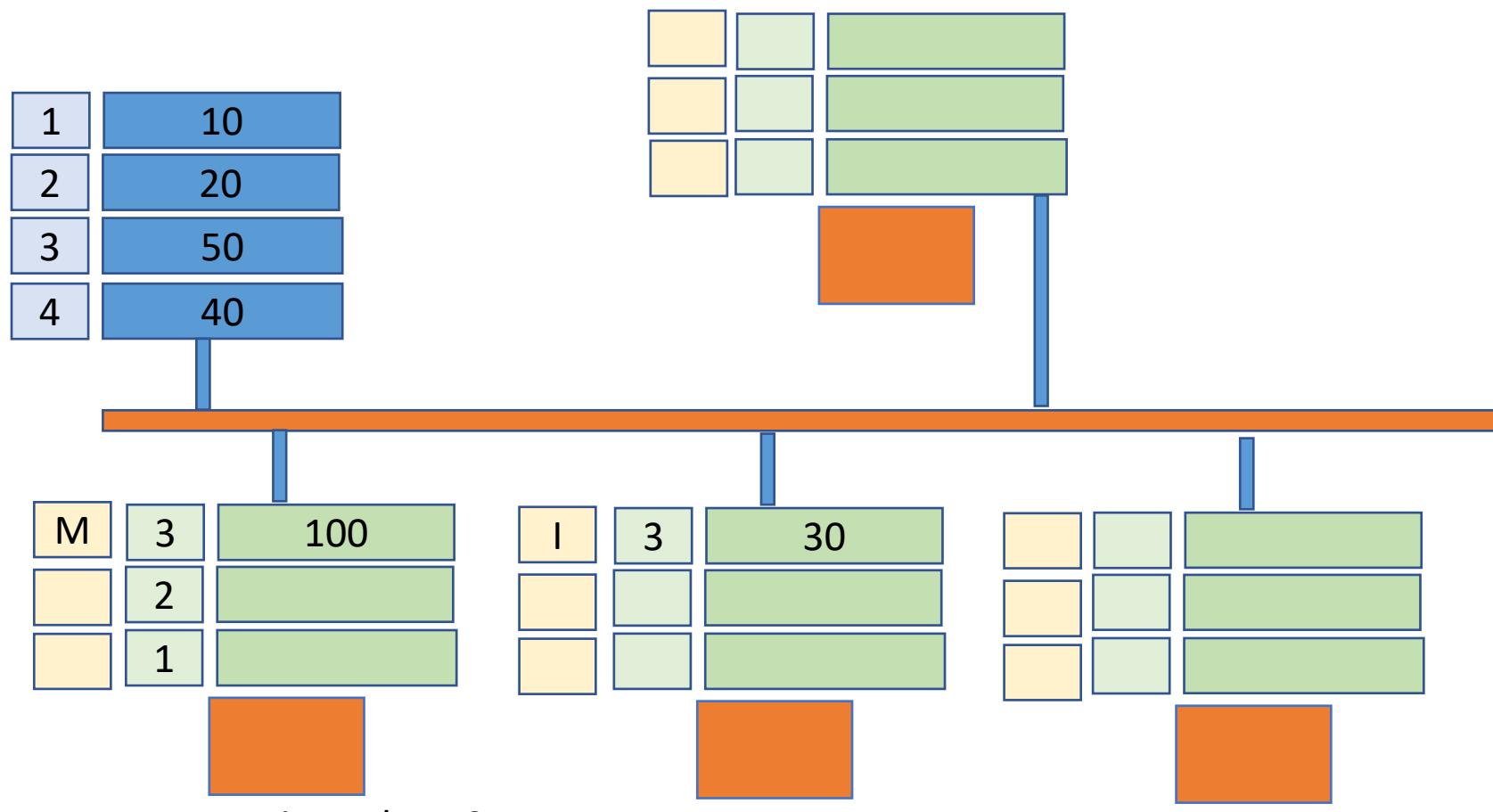
Write Hit:

Case 3:
P1 has the data in modified state

Simulation Write Hit



Simulation Write Hit



Write Hit:

Case 3:
P1 has the data in modified state

The new value is written to the Cache, still kept as modified.

Multi-Threading

Some approaches for increasing speed:

- Pipeline - we have seen
- Multiple parallel instruction pipelines
- Changing instruction order

Another approach :

Using Threads

What is a thread and what is a process?

Process

- Instance of a program running in computer
- Resources - PC, data (memory, Register), stack
- Scheduling and switching

Thread

- dispatchable part of a process, own stack
- Thread can also be scheduled and switched

Just for now, think of threads as programs or parts of a program that can be executed **parallelly**.

4 principal approaches multithreading

Interleaved multithreading:

- Processor deals with two or more thread contexts at a time.
- For some clock cycles a thread is executed.
- For other clock cycles , another thread is executed.
- If a thread becomes blocked - due to i/o , it is swapped with another processor.

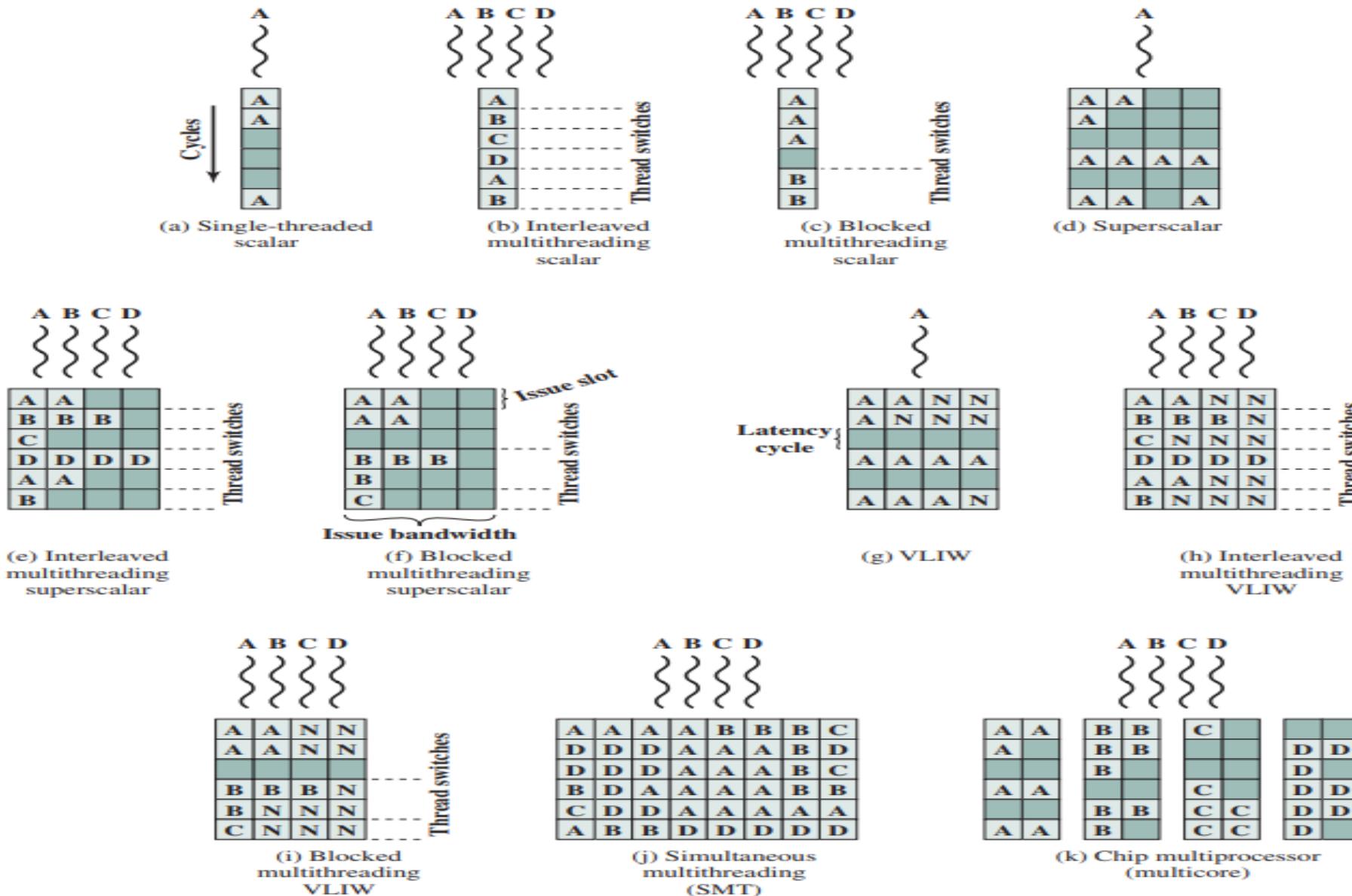
Blocked multithreading:

- Keep on executing a thread.
- If any delay causing event occurs (for example i/o).
- Only then switch to another thread.

4 principal approaches for multithreading

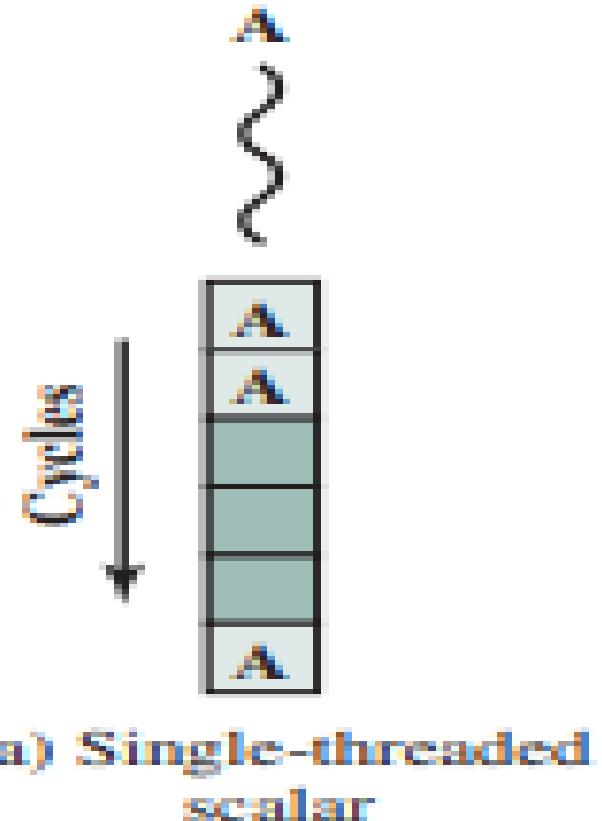
- **Simultaneous multithreading:**
 - Superscalar processor - has multiple execution units.
 - Each executable unit executes a thread.
- **Chip multiprocessing:**
 - Multiple cores are there in the processor.
 - They can execute one or more threads.

Some instance of multithreading



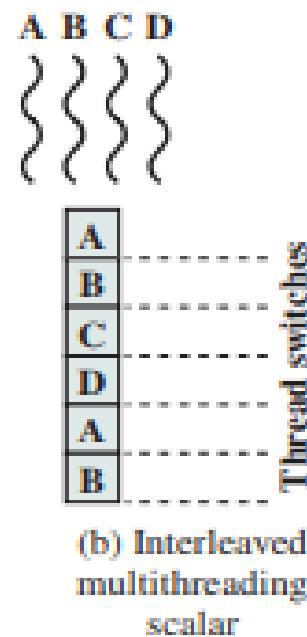
a. Single threaded scalar

- No multithreading , one thread is finished, and then another thread is loaded.



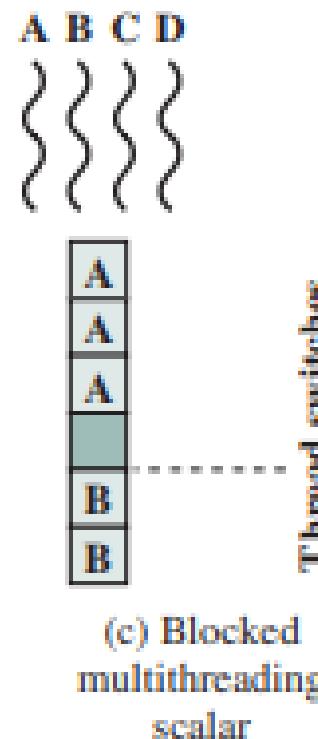
b. Interleaved Multithreading

- Alternately, for some clock cycles one thread is run, and then another thread is run. In a circular manner.



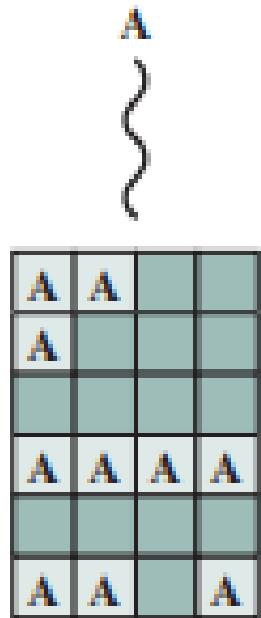
c. Blocked multithreading scalar

A thread is executed until a latency event occurs. (For example i/o). Upon encountering a latency event, the thread is switched.



d. Superscalar

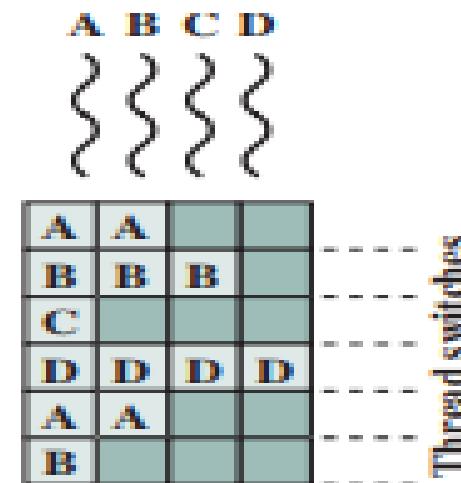
- A superscalar approach is shown with no multithreading.
- The gaps in the figure indicate some latency event / problems in instruction issue due to dependency.



(d) Superscalar

e. Interleaved multithreading superscalar

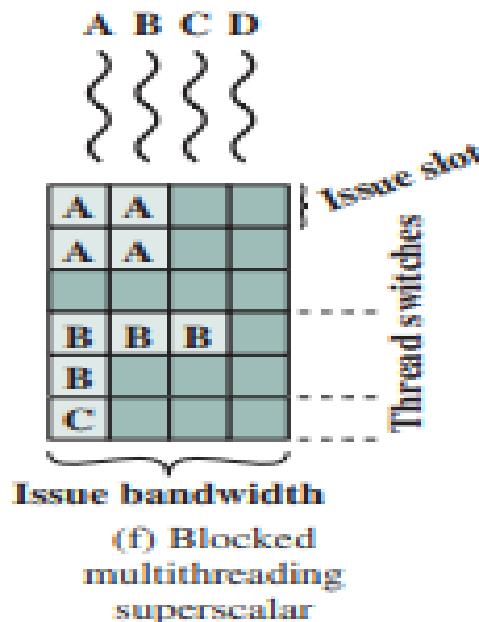
- As many instructions as possible are issued from a thread.
- The instruction issuing may be limited due to data dependency.



(e) Interleaved
multithreading
superscalar

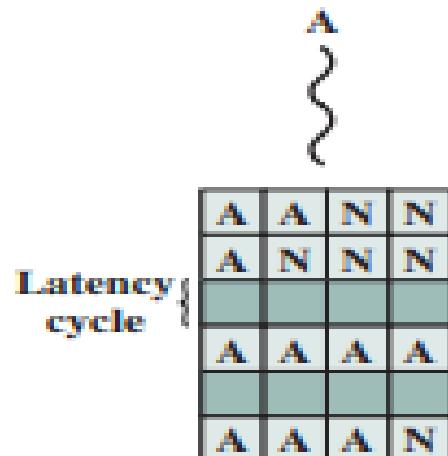
f. Blocked multithreading superscalar

- Instructions from one thread are issued until it becomes blocked by a latency event.
- After becoming blocked, a switch occurs.
- Also switch may occur when a thread is finished



g. Very long instruction word

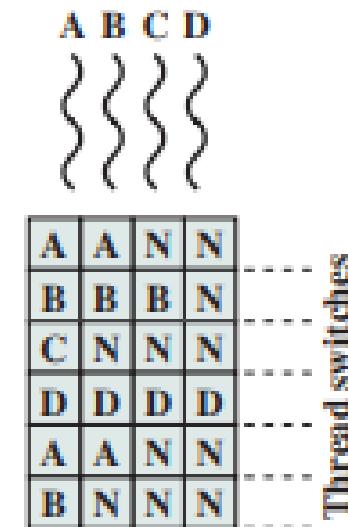
- Here one instruction consists of multiple instructions. (this is done by the compiler).
- And the instructions are variable length instructions.
- Hence. Multiple sub instructions are sent into multiple pipelines at once.
- Since the instructions are variable in length, No ops are used.



(g) VLIW

h. Interleaved multithreading VLIW

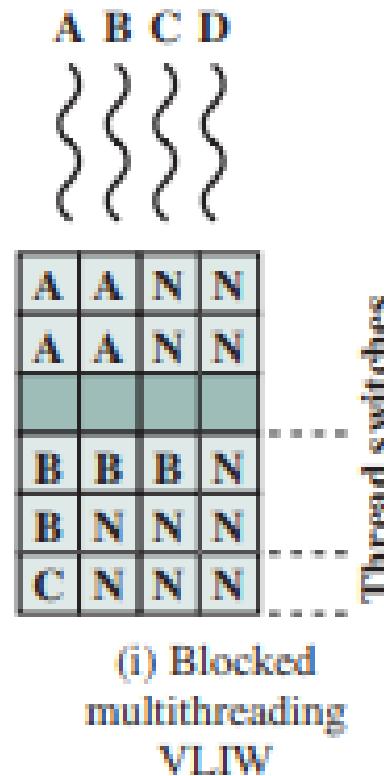
- Same as before, just the threads are switched



(h) Interleaved
multithreading
VLIW

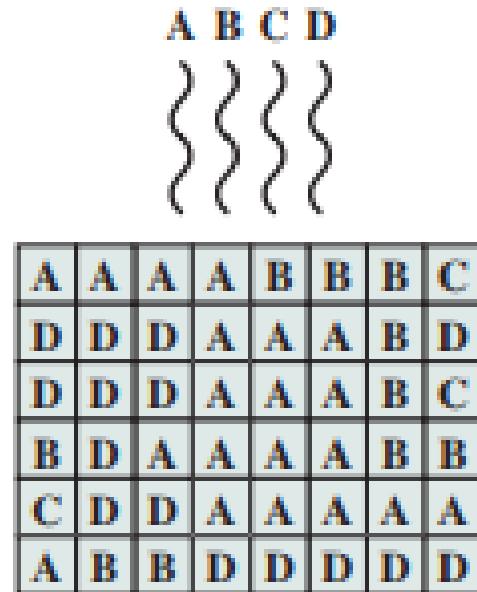
i. Blocked multithreading VLIW

- Execute sub instructions together until blocked by some latency event.



j. Simultaneous multithreading (SMT)

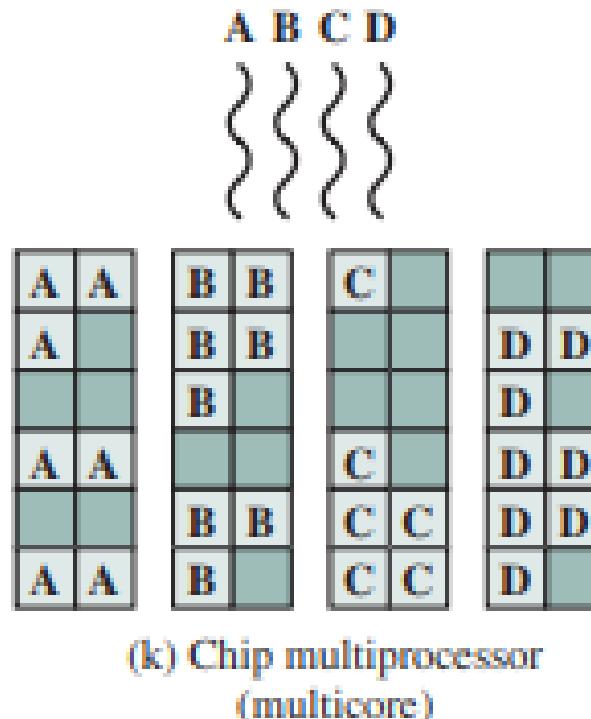
- As long as no dependency is found, keep on inserting instructions horizontally into processing units. Upon finding dependency or latency event, switch.



(j) Simultaneous
multithreading
(SMT)

k. Chip multiprocessor (multicore)

- Here a chip with 4 processors are shown.
- Each processor is superscalar (i.e has two execution units).



The End