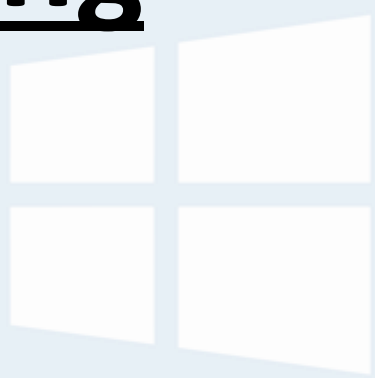
Operating Systems COMS(3010A) Paging

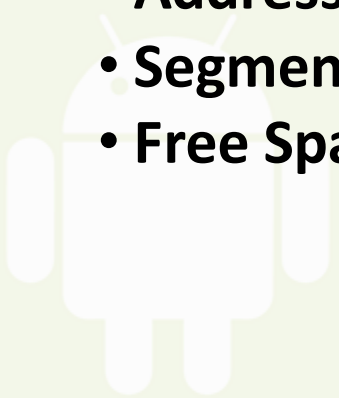
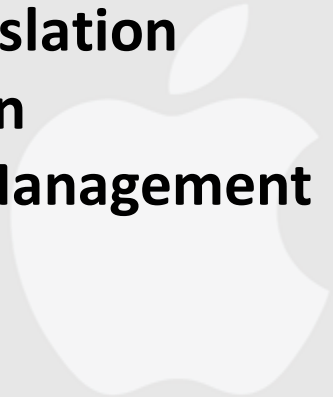



Branden Ingram

branden.ingram@wits.ac.za

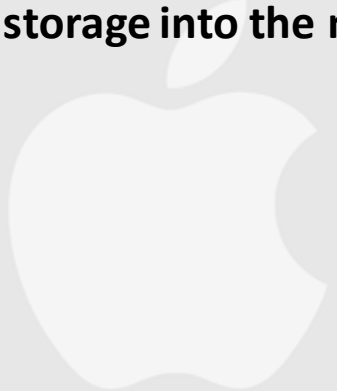
Recap

- Memory Virtualisation
- Address Translation
- Segmentation
- Free Space Management

Paging

• In Operating Systems, Paging is a storage mechanism used to retrieve processes from the secondary storage into the main memory in the form of pages.



Paging

- In Operating Systems, Paging is a storage mechanism used to retrieve processes from the secondary storage into the main memory in the form of pages.

- The main idea behind the paging is to divide each process(Address Space) into a fixed-sized unit called a page

Paging

- **In Operating Systems, Paging is a storage mechanism used to retrieve processes from the secondary storage into the main memory in the form of pages.**

- **The main idea behind the paging is to divide each process(Address Space) into a fixed-sized unit called a page**

  - Segmentation: variable size of logical segments(code, stack, heap, etc.)

Paging

- **In Operating Systems, Paging is a storage mechanism used to retrieve processes from the secondary storage into the main memory in the form of pages.**

- **The main idea behind the paging is to split each process(Address Space) into a fixed-sized unit called a page**

    - Segmentation: variable size of logical segments(code, stack, heap, etc.)

- **With paging, physical memory is also split into some number of pages called a page frame.**

Paging Translation ?

- **Page table** per process is needed **to translate** the virtual address to physical address.
  - Similar to the segment register

Segment Register

Segment	Base	Size
Code	256KB	4KB
Heap	260KB	60KB
Stack	128KB	64KB

Paging Translation ?

- **Page table** per process is needed **to translate** the virtual address to physical address.
  - Similar to the segment register

Segment Register

Segment	Base	Size
Code	256KB	4KB
Heap	260KB	60KB
Stack	128KB	64KB

Page Table

Page	Page Frame
0	3
1	7
2	5
3	2

Advantages of Paging

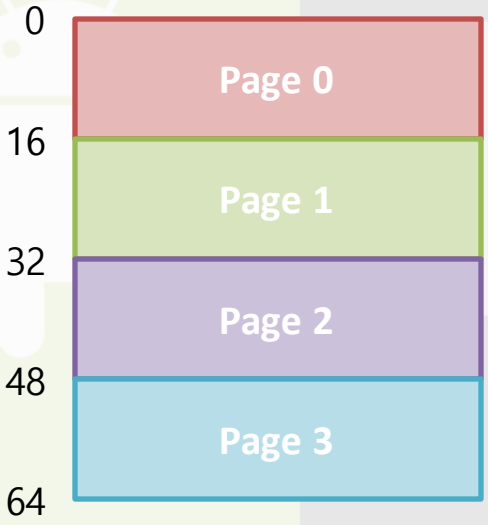
- **Flexibility:** Supporting the abstraction of address space effectively
  - Don’t need assumption how heap and stack grow and are used.

Advantages of Paging

- **Flexibility: Supporting the abstraction of address space effectively**
  - Don’t need assumption how heap and stack grow and are used.
- **Simplicity: ease of free-space management**
  - The page in address space and the page frame are the same size.
  - Easy to allocate and keep a free list

Paging : Example

- 128-byte physical memory with 16 bytes page frames
- 64-byte address space with 16 bytes pages

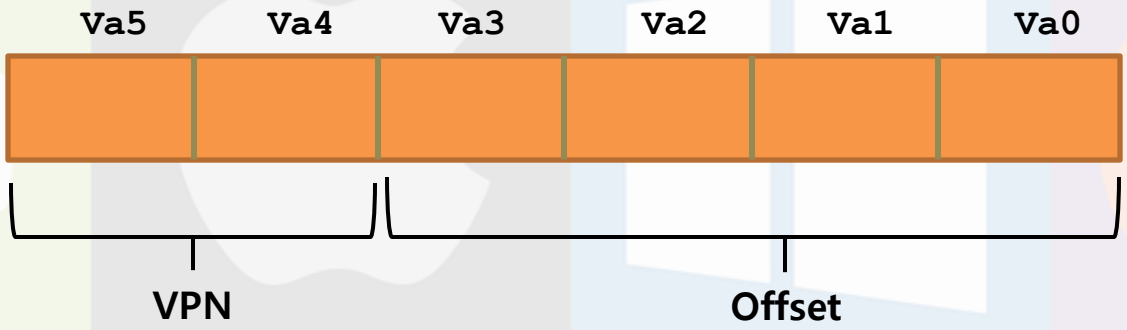


64 byte Address Space

64 byte Address Space placed in Physical Memory

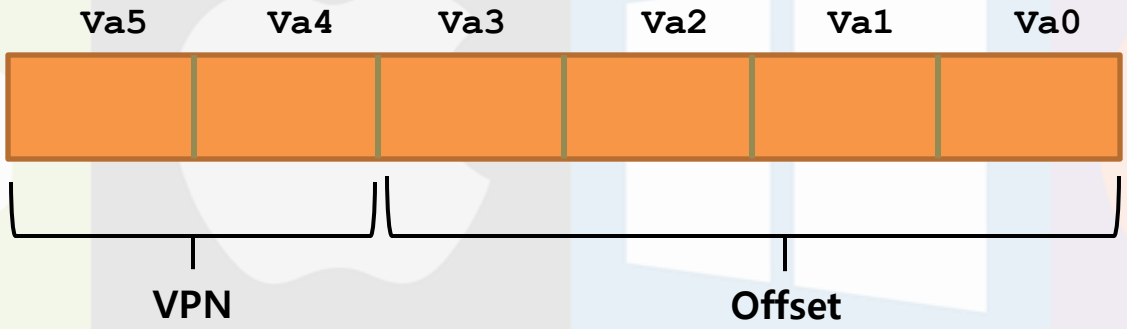
Address Translation : Example

- **Two components in the virtual address**
  - **VPN**: virtual page number
  - **Offset**: offset within the page

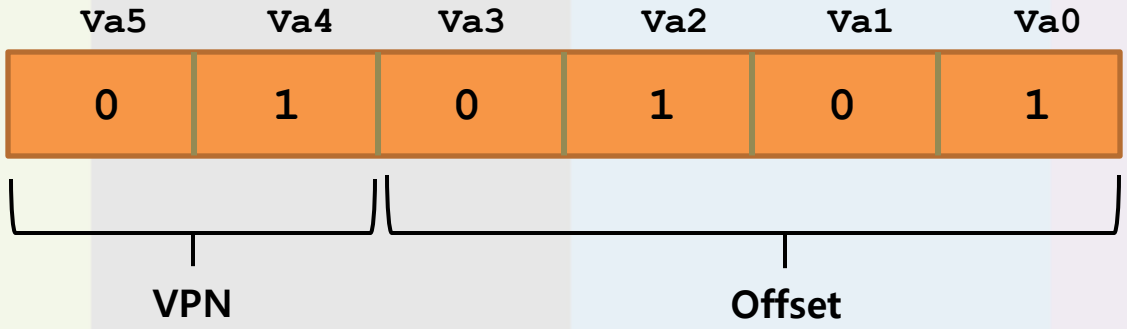


Address Translation : Example

- **Two components in the virtual address**
  - **VPN**: virtual page number
  - **Offset**: offset within the page

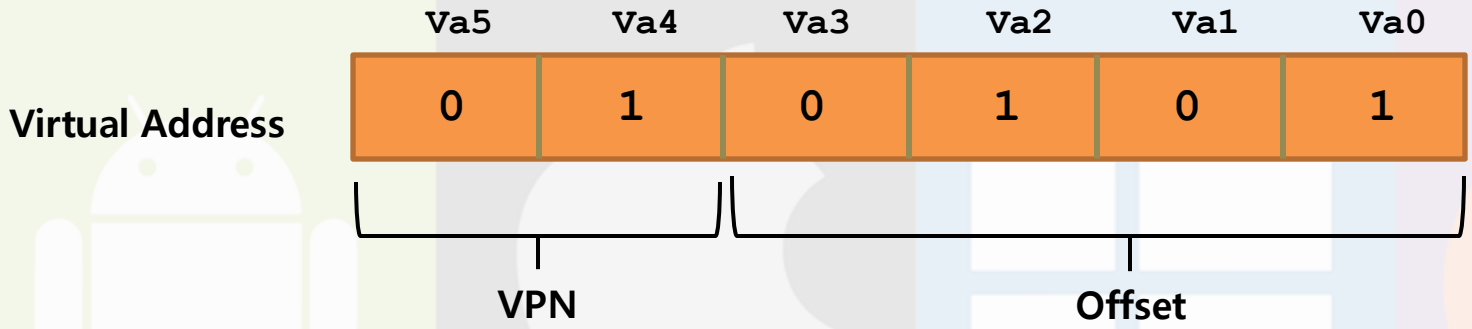


- Example: virtual address 21 in 64-byte address space



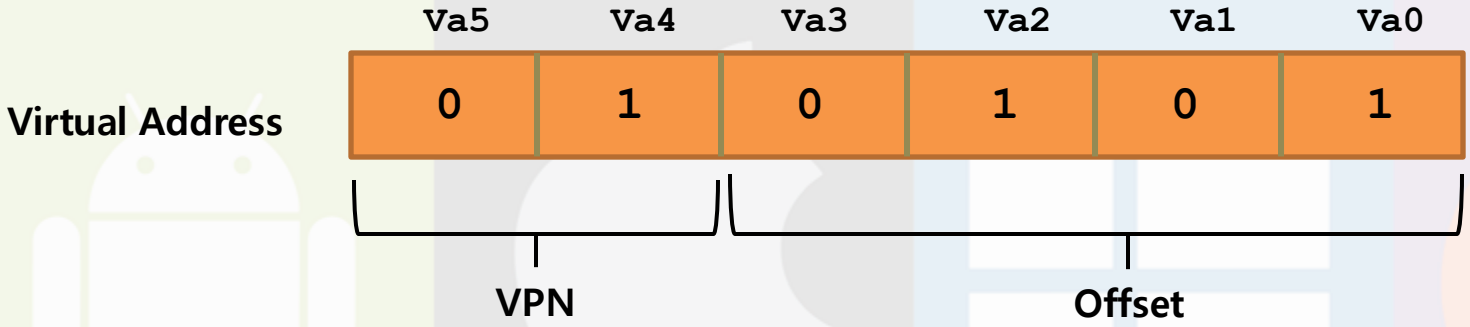
Address Translation : Example

- Example: virtual address 21 in 64-byte address space



Address Translation : Example

- Example: virtual address 21 in 64-byte address space



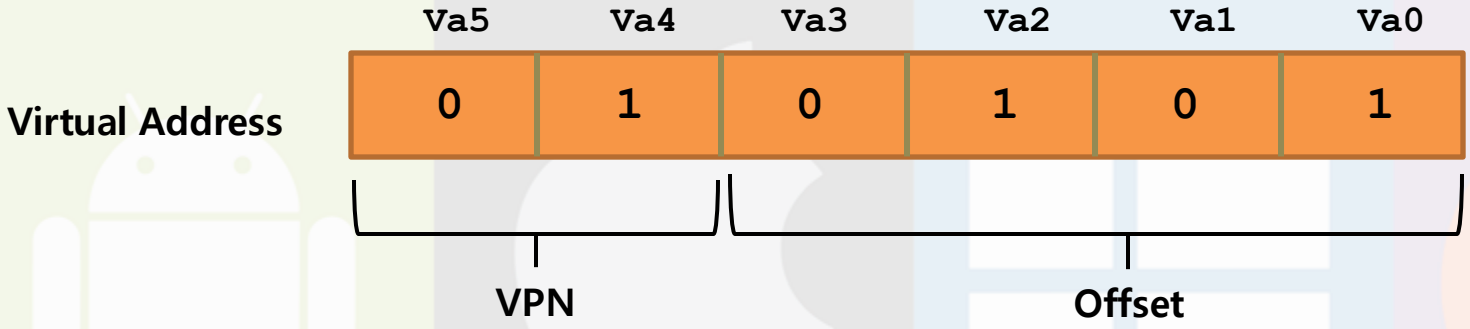
How do I know the Virtual Address is x bits ?

Address space= 64bytes

log_2(64) = 6

Address Translation : Example

- Example: virtual address 21 in 64-byte address space



How do I know the Virtual Address is x bits ?

Address space= 64bytes

log_2(64) = 6 (Virtual Address)

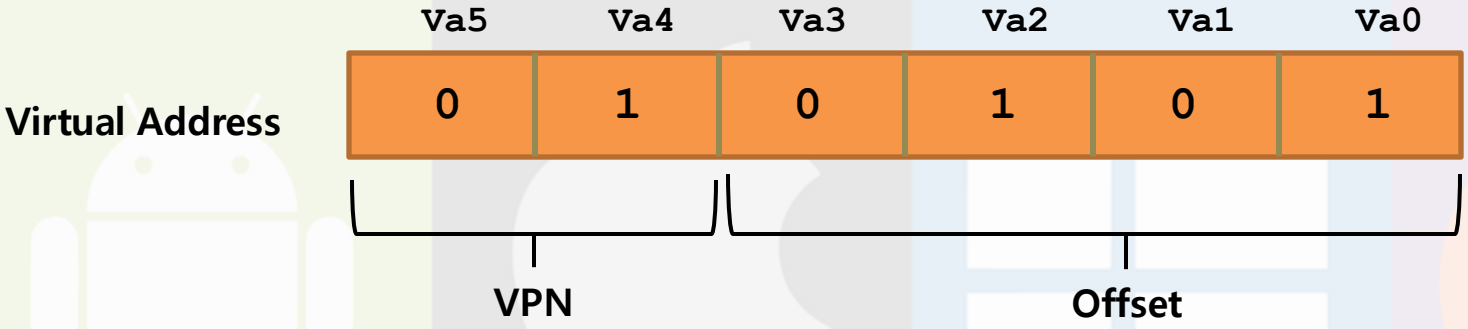
How do I know its 2 bits for VPN and 4 bits for Offset

Page Size = 16

log_2(16) = 4 (offset)

Address Translation : Example

- Example: virtual address 21 in 64-byte address space



How do I know the Virtual Address is 6 bits ?

Address space= 64bytes

$\log_2(64) = 6$ (Virtual Address)

How do I know its 2 bits for VPN and 4 bits for Offset

Page Size = 16bytes

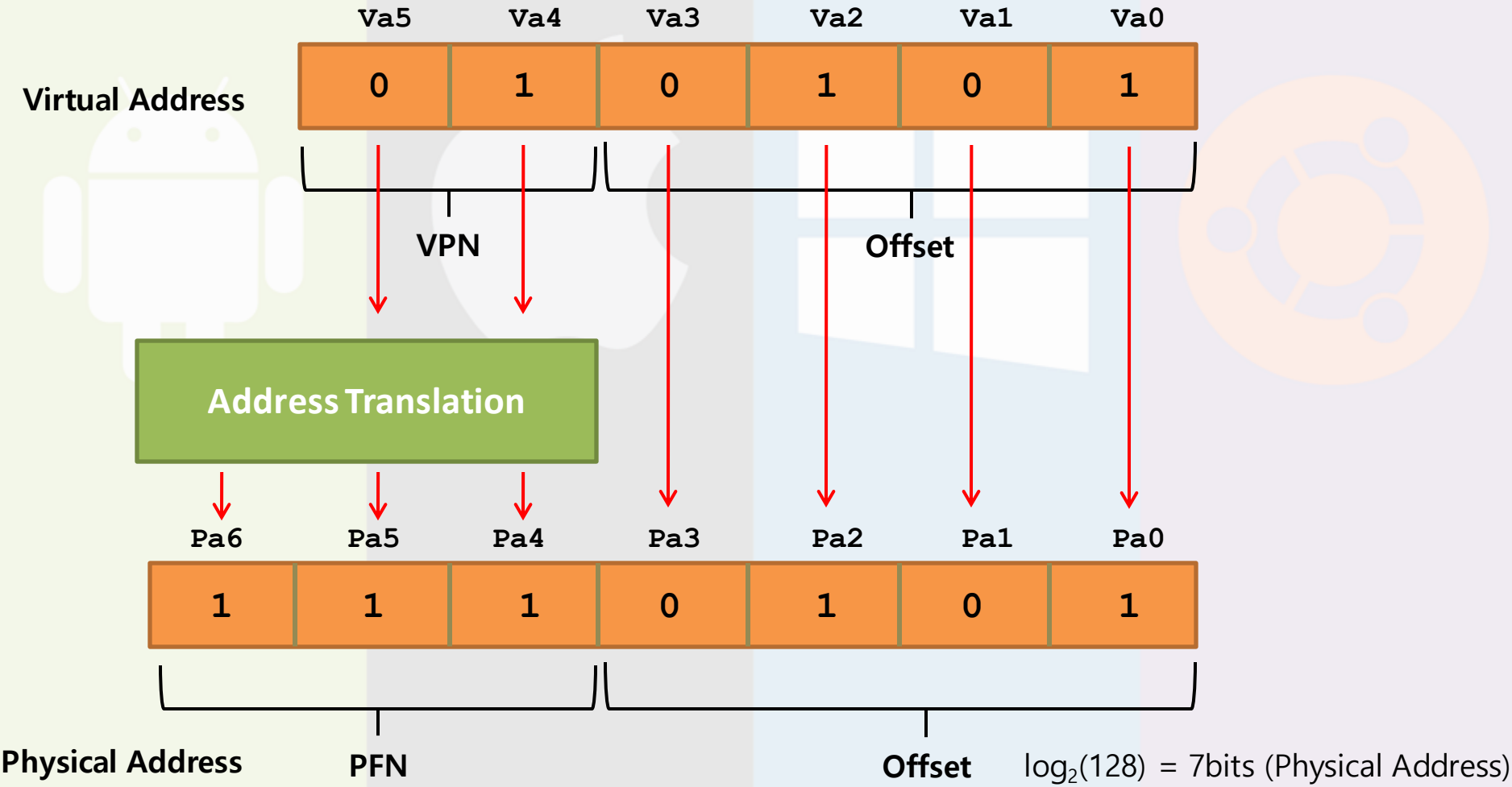
$\log_2(16) = 4$ (offset)

How do I know 21 = 010101

$21_{10} = 10101_2(5\text{bits}) = 010101_2(6\text{bits})$

Address Translation : Example

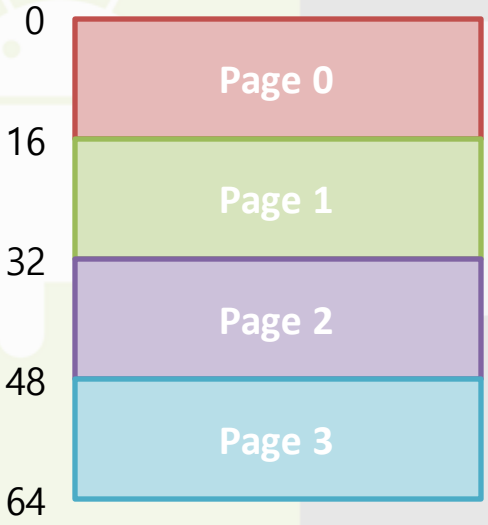
- Example: virtual address 21 in 64-byte address space



$\log_2(128) = 7$bits (Physical Address)

Paging : Example

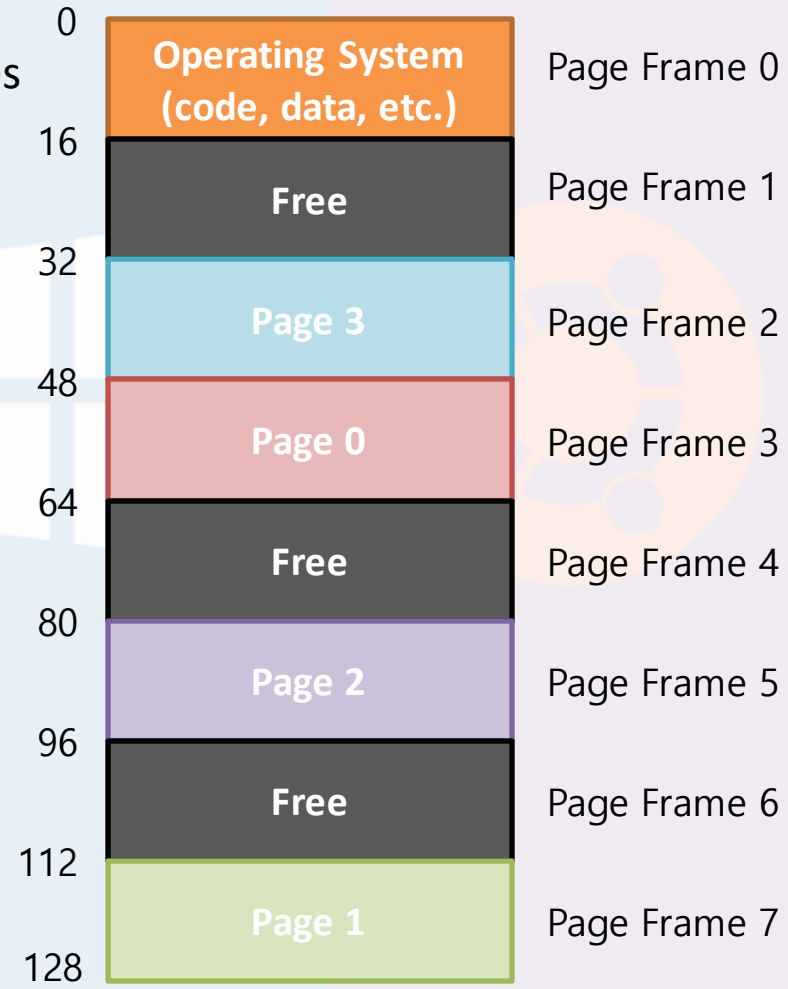
- 128-byte physical memory with 16 bytes page frames
- 64-byte address space with 16 bytes pages



64 byte Address Space

VPN = 01

PFN = 3bits



64 byte Address Space placed in Physical Memory

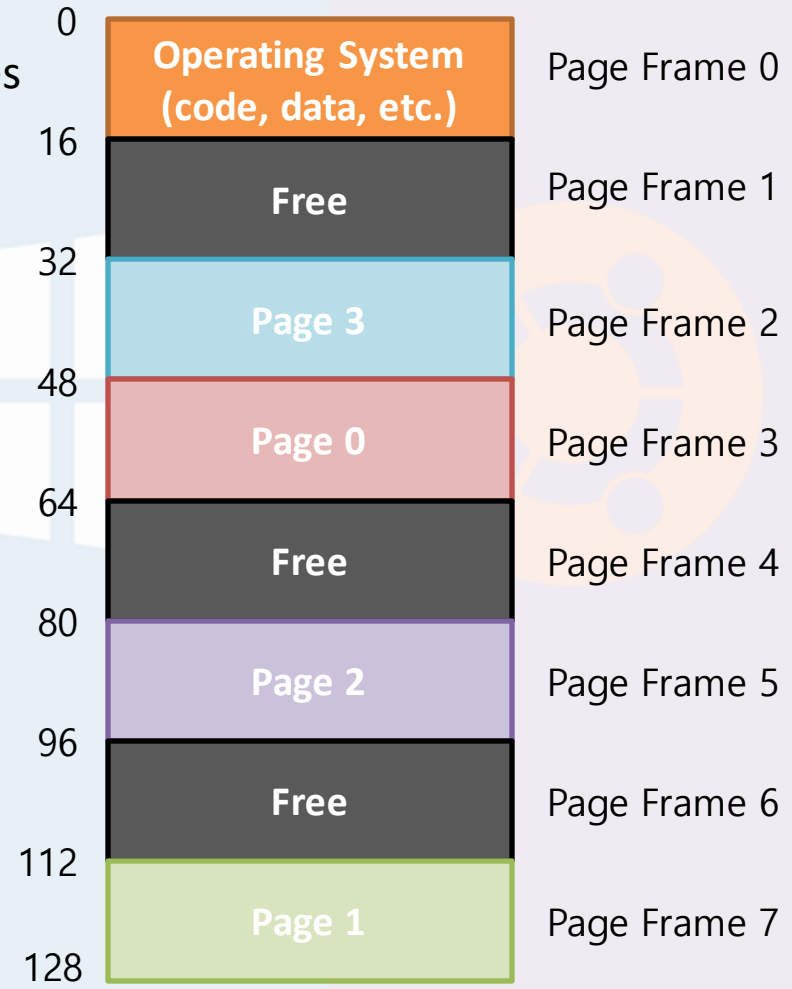
Paging : Example

- 128-byte physical memory with 16 bytes page frames
- 64-byte address space with 16 bytes pages

64 byte Address Space

VPN = 01

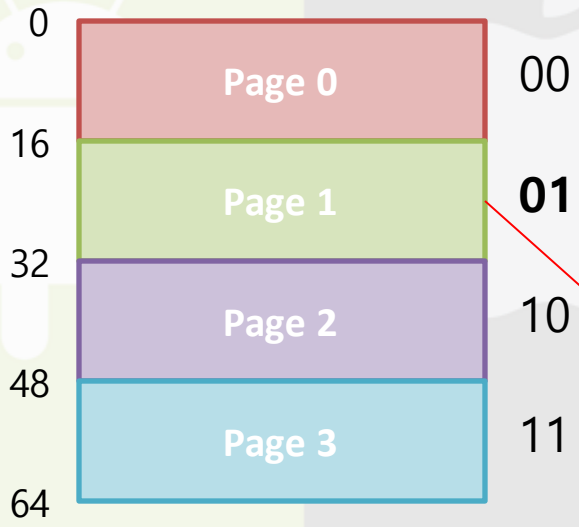
PFN = 3bits



64 byte Address Space placed in Physical Memory

Paging : Example

- 128-byte physical memory with 16 bytes page frames
- 64-byte address space with 16 bytes pages



64 byte Address Space

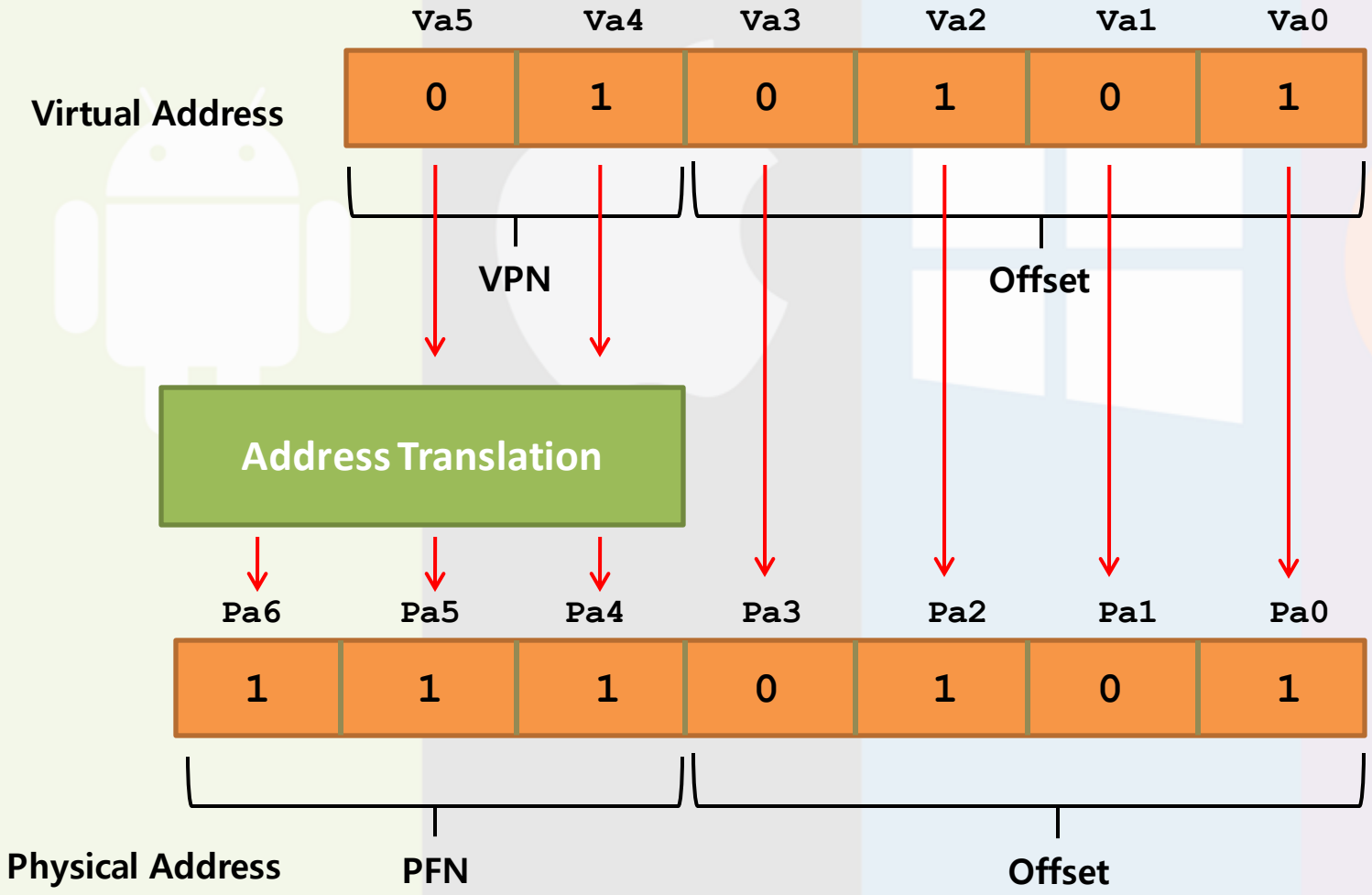
VPN = 01

PFN = 3bits

64 byte Address Space placed in Physical Memory

Address Translation : Example

- Example: virtual address 21 in 64-byte address space



Paging Translation ?

- **Page table** per process is needed **to translate** the virtual address to physical address.
  - Similar to the segment register

Page Table

Page	Page Frame
0	3
1	7
2	5
3	2

Ok well where are these tables stored?

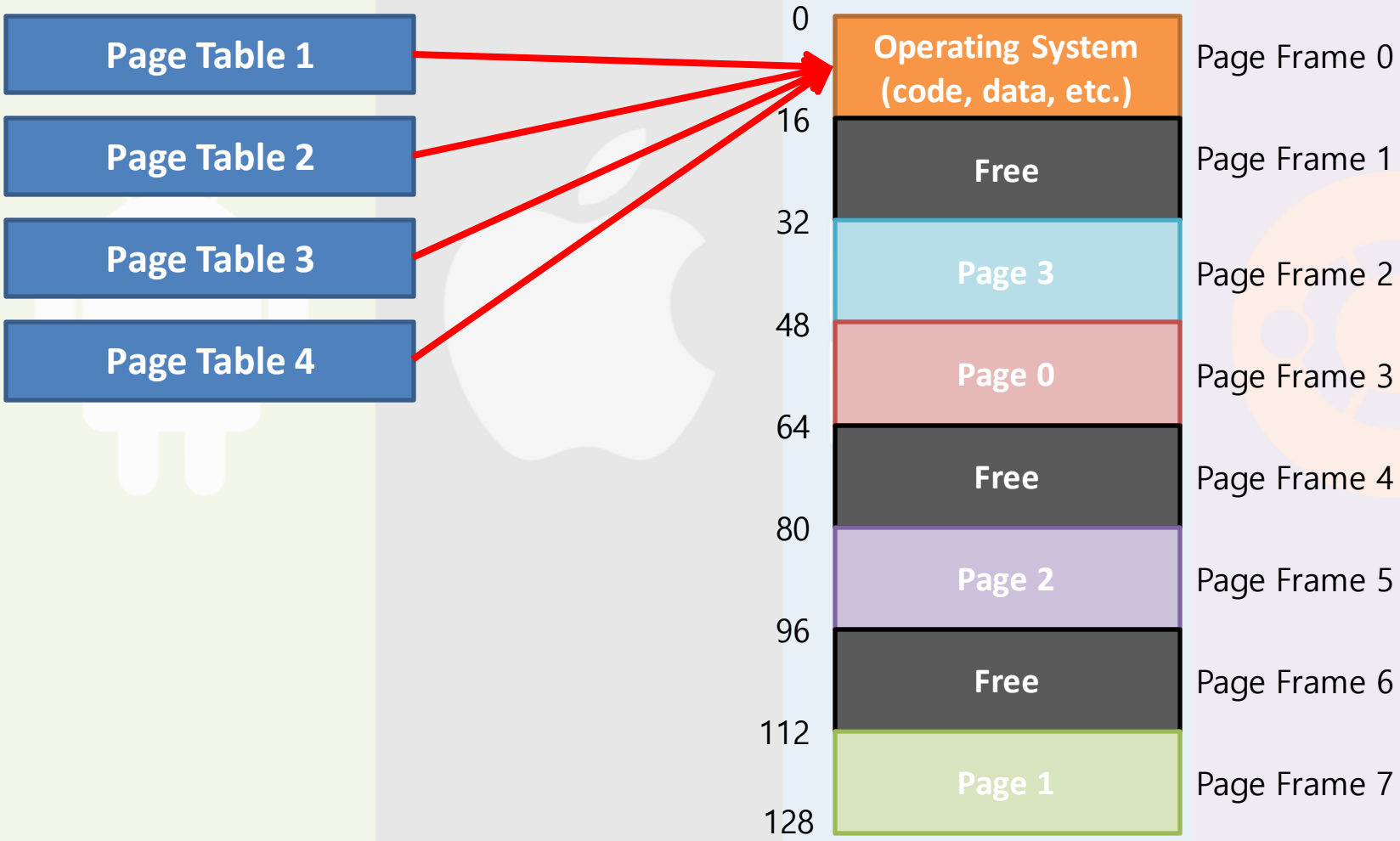
- Page tables can get large quickly
  - 32 bit address space with 4KB pages, 20bits for the VPN
    - 4MB = 2^{20} entries x 4 bytes per page table entry
- This is too large to store entries in dedicated CPU register

Ok well where are these tables stored?

- Page tables can get large quickly
  - 32 bit address space with 4KB pages, 20bits for the VPN
    - 4MB = 2^{20} entries x 4 bytes per page table entry

- And Therefore all of that is stored in memory

Page Table in Kernel Memory



64 byte Address Space placed in Physical Memory

What is in the Page Table?

- The page table is just a data structure that is used to map the virtual address to the physical address

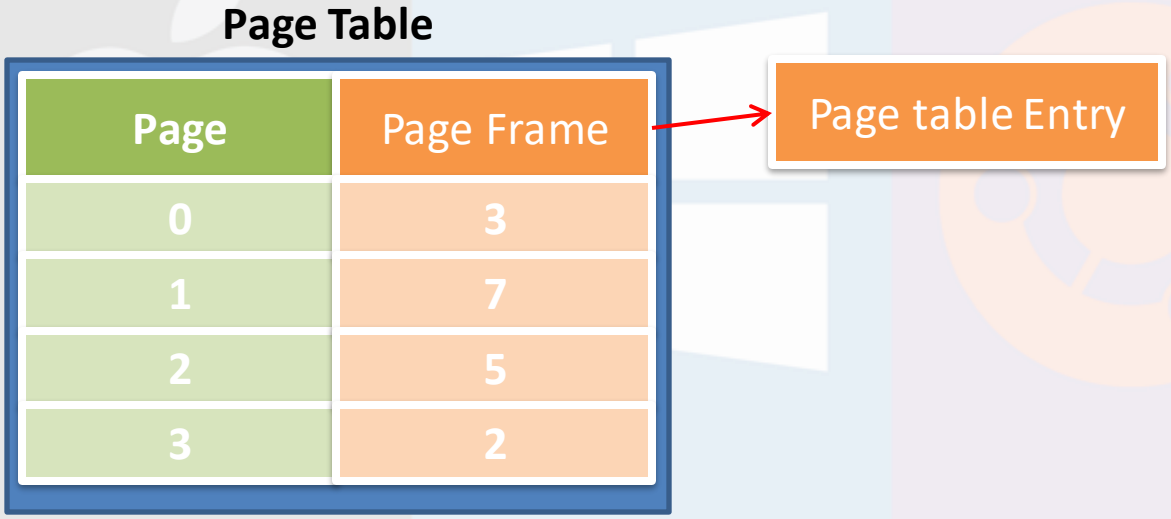
Page Table

Page	Page Frame
0	3
1	7
2	5
3	2

- In its simplest form it is just an array

What is in the Page Table?

- The page table is just a data structure that is used to map the virtual address to the physical address



- In its simplest form it is just an array
- The OS indexes the array by VPN and looks up the page table entry

Page Table Entry : Example x86

- P: present bit
- R/W: read/write bit
- U/S: supervisor
- A: accessed bit
- D: dirty bit
- PFN: the page frame number

Page Table Entry : Example x86

- **Valid Bit(P)**: Indicating whether the particular translation is valid.

- **Protection Bit(R/W)**: Indicating whether the page could be read from, written to, or executed from

- **Present Bit(P)**: Indicating whether this page is in physical memory or on disk(swapped out)

- **Dirty Bit(D)**: Indicating whether the page has been modified since it was brought into memory

- **Reference Bit(Accessed Bit -A):** Indicating that a page has been accessed

What is the issue with Paging?

What is the issue with Paging?

- **Size – Page tables and entries take up a lot of space in memory**

What is the issue with Paging?

- Size – Page tables and entries take up a lot of space in memory
- Slow – 2 references per memory references

Paging is Slow?

- **We need to first reference the correct page table (1 reference)**
  - To find a location of the desired PTE, the **starting location** of the page table is **needed**
- **Then we can reference the correct page table entry (1 reference)**

Paging is Slow?

- **We need to first reference the correct page table (1 reference)**
  - To find a location of the desired PTE, the **starting location** of the page table is **needed**
- **Then we can reference the correct page table entry (1 reference)**
- **Therefore, for every memory reference, paging requires the OS to perform one extra memory reference**

Accessing Memory with Paging

```
1    // Extract the VPN from the virtual address
2   VPN = (VirtualAddress & VPN_MASK) >> SHIFT

3    // Form the address of the page-table entry (PTE)
4   PTEAddr = PTBR + (VPN * sizeof(PTE))

5    // Check if process can access the page
6    if (PTE.Valid == False)
7          RaiseException(SEGMENTATION_FAULT)

8    else if (CanAccess(PTE.ProtectBits) == False)
9          RaiseException(PROTECTION_FAULT)

10   else
11         // Access is OK: form physical address and fetch it
12         offset = VirtualAddress & OFFSET_MASK
13         PhysAddr = (PTE.PFN << PFN_SHIFT) | offset
14         Register = AccessMemory(PhysAddr)
```

Accessing Memory with Paging

Example : A Simple Memory Access

```
int array[1000];
...
for(i = 0; i < 1000; i++)
        array[i] = 0;
```

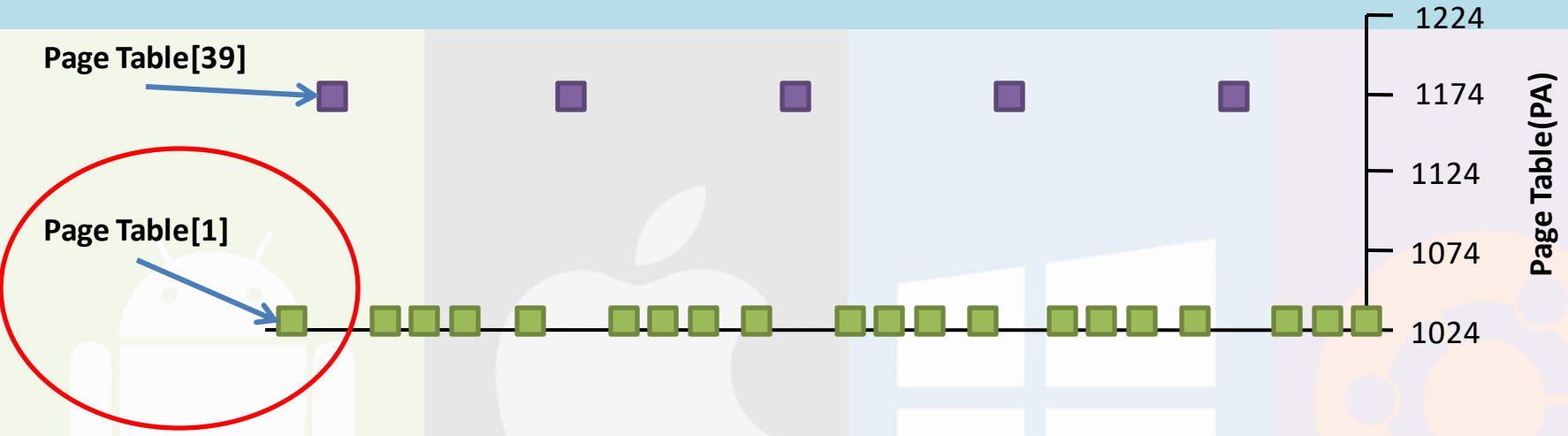
Compile and Execute

```
prompt> gcc –o array array.c –Wall -o
prompt> ./array
```

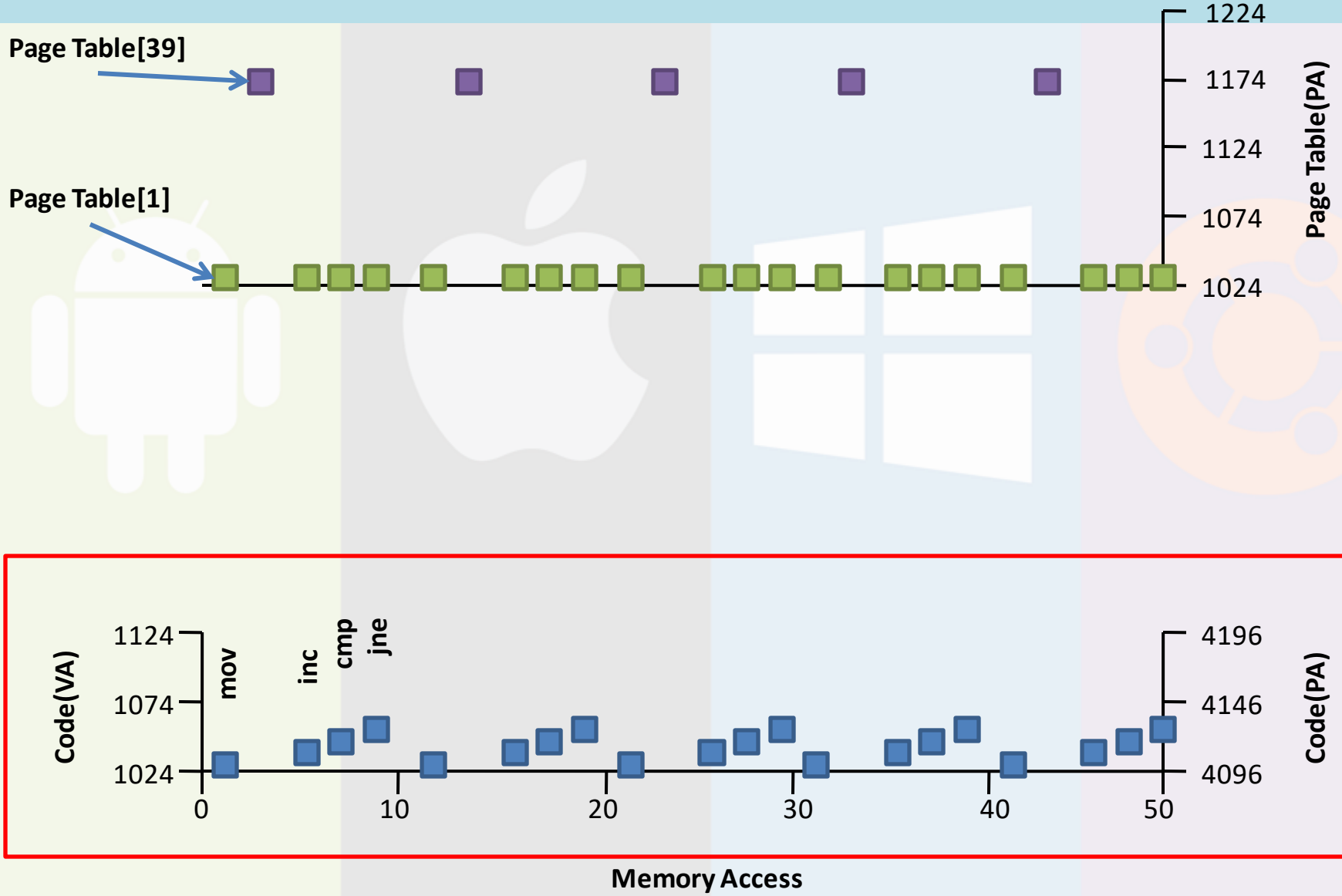
Resulting Assembly code

```
0x1024 : movl $0x0, (%edi,%eax,4)
0x1028 : incl %eax
0x102c : cmpl $0x03e8,%eax
0x1030 : jne 0x1024
```

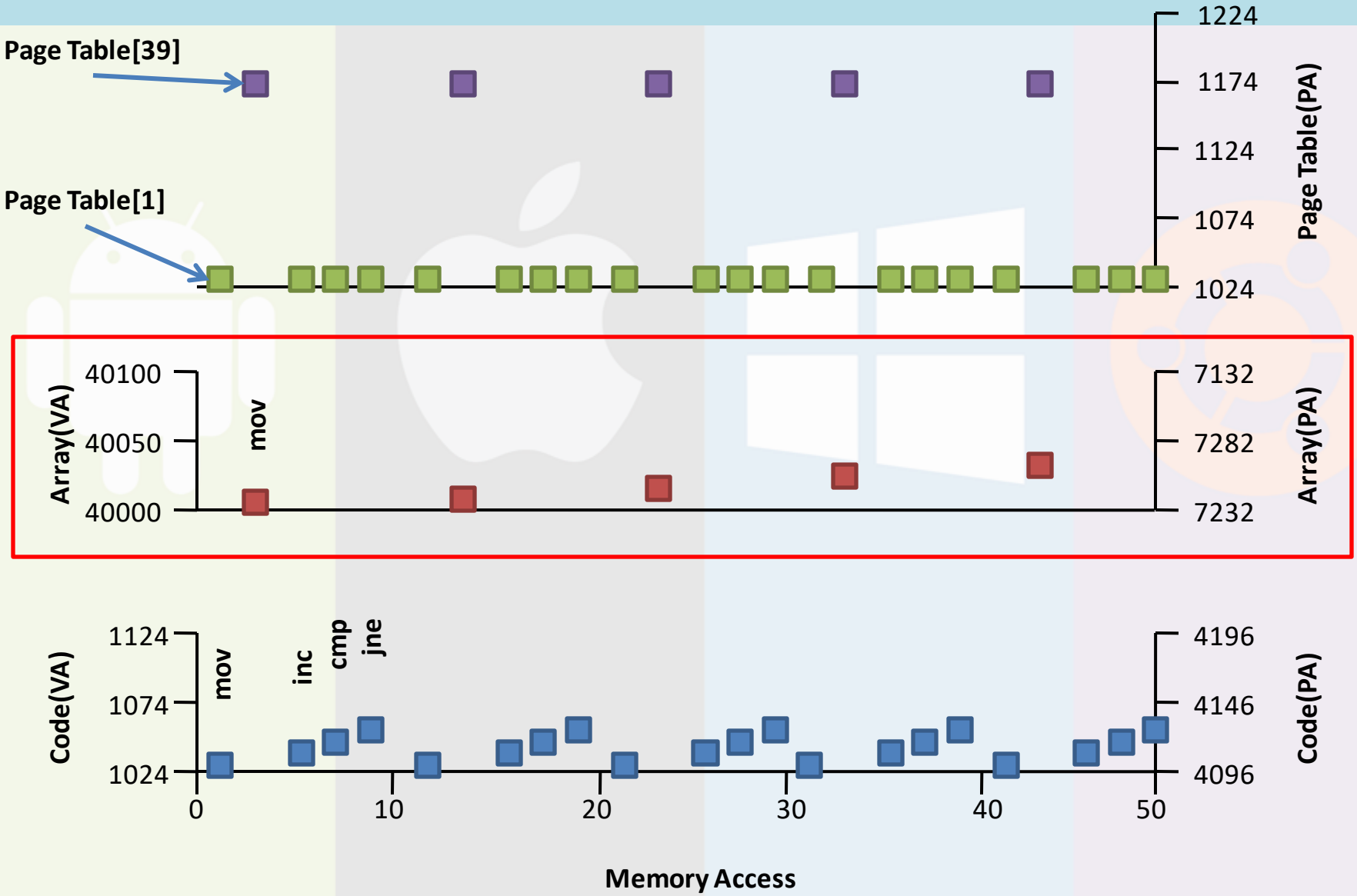
A Virtual(And Physical) Memory Trace



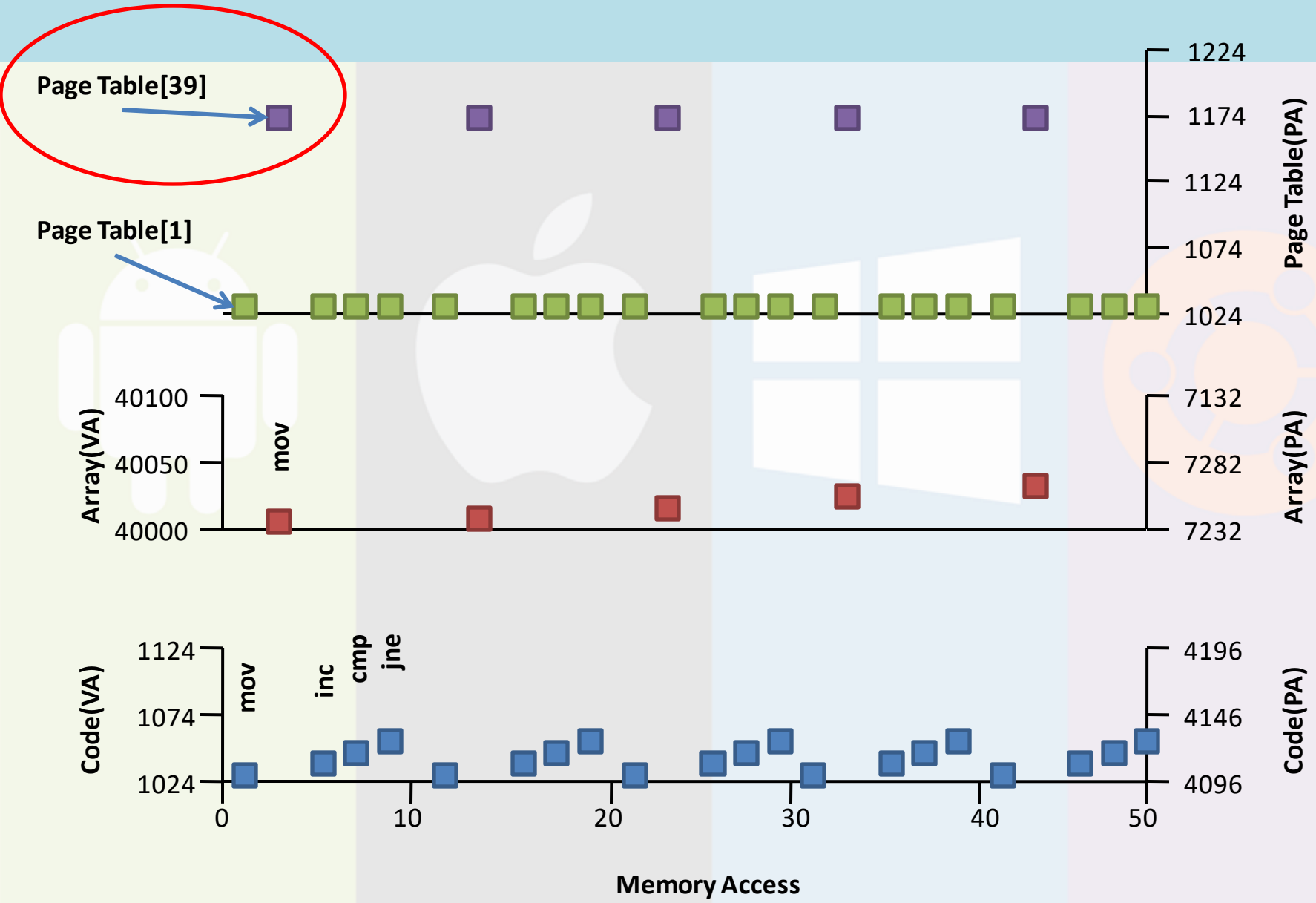
A Virtual(And Physical) Memory Trace



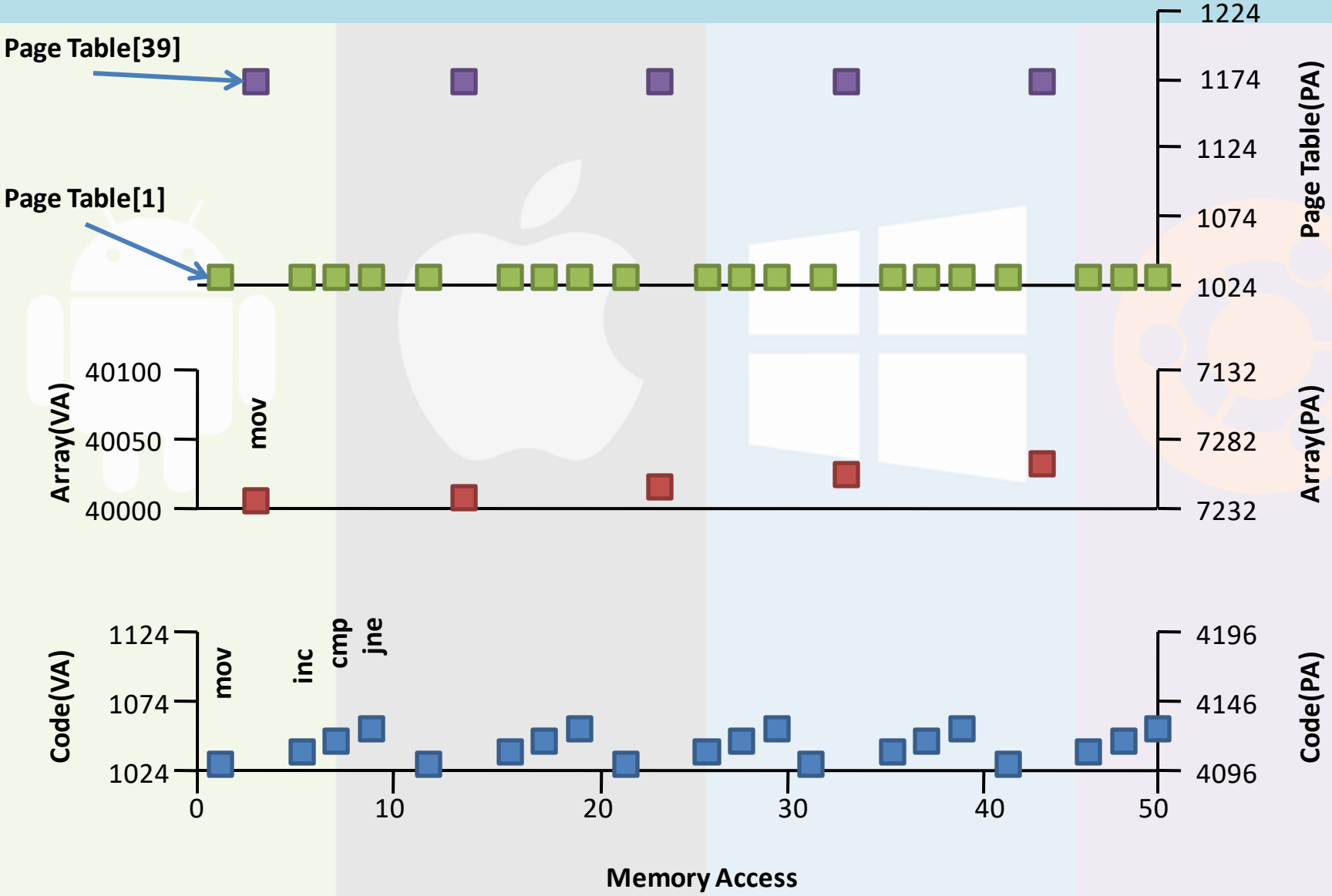
A Virtual(And Physical) Memory Trace



A Virtual(And Physical) Memory Trace



A Virtual(And Physical) Memory Trace



Paging is Slow?

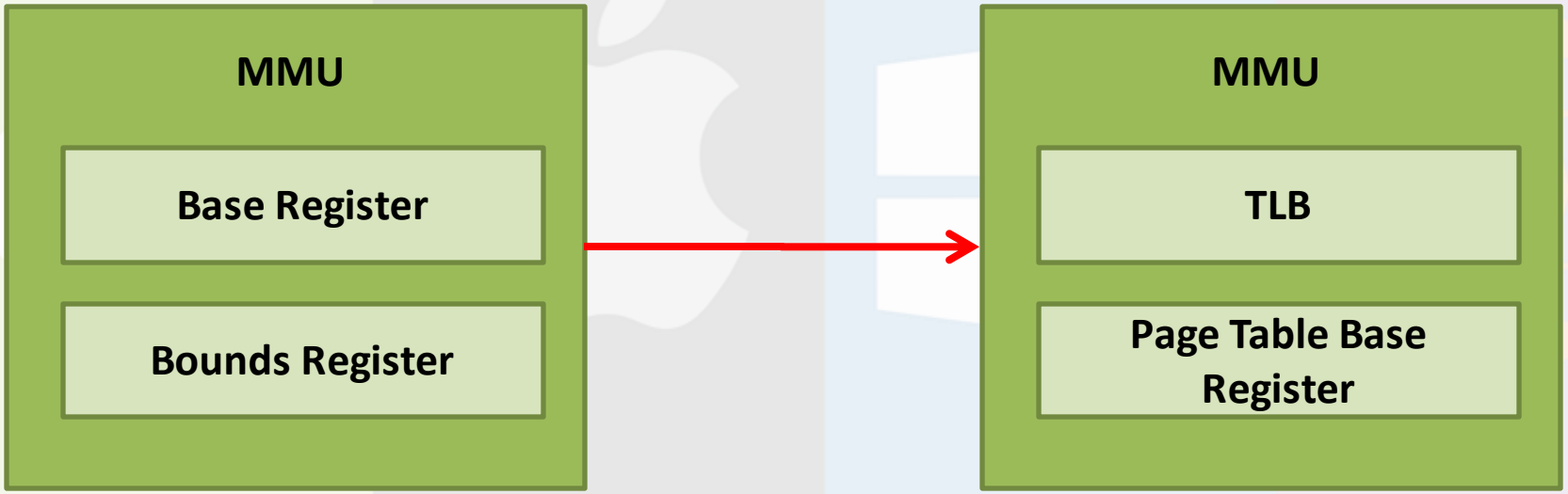
- **We need to first reference the correct page table (1 reference)**
  - To find a location of the desired PTE, the **starting location** of the page table is **needed**
- **Then we can reference the correct page table entry (1 reference)**
- **Therefore, for every memory reference, paging requires the OS to perform one extra memory reference**
- **How can we improve this access speed**

Translation Lookaside Buffers

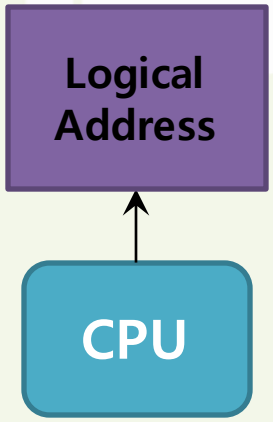
- Part of the chip's memory-management unit(MMU).

Translation Lookaside Buffers

- Part of the chip's memory-management unit(MMU).

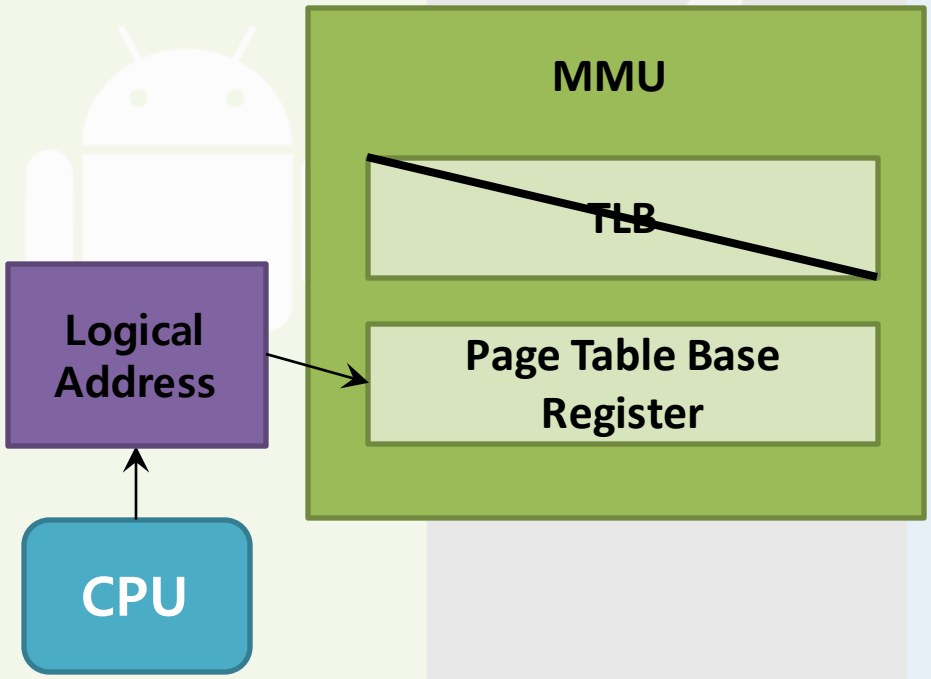


Translation Lookaside Buffers



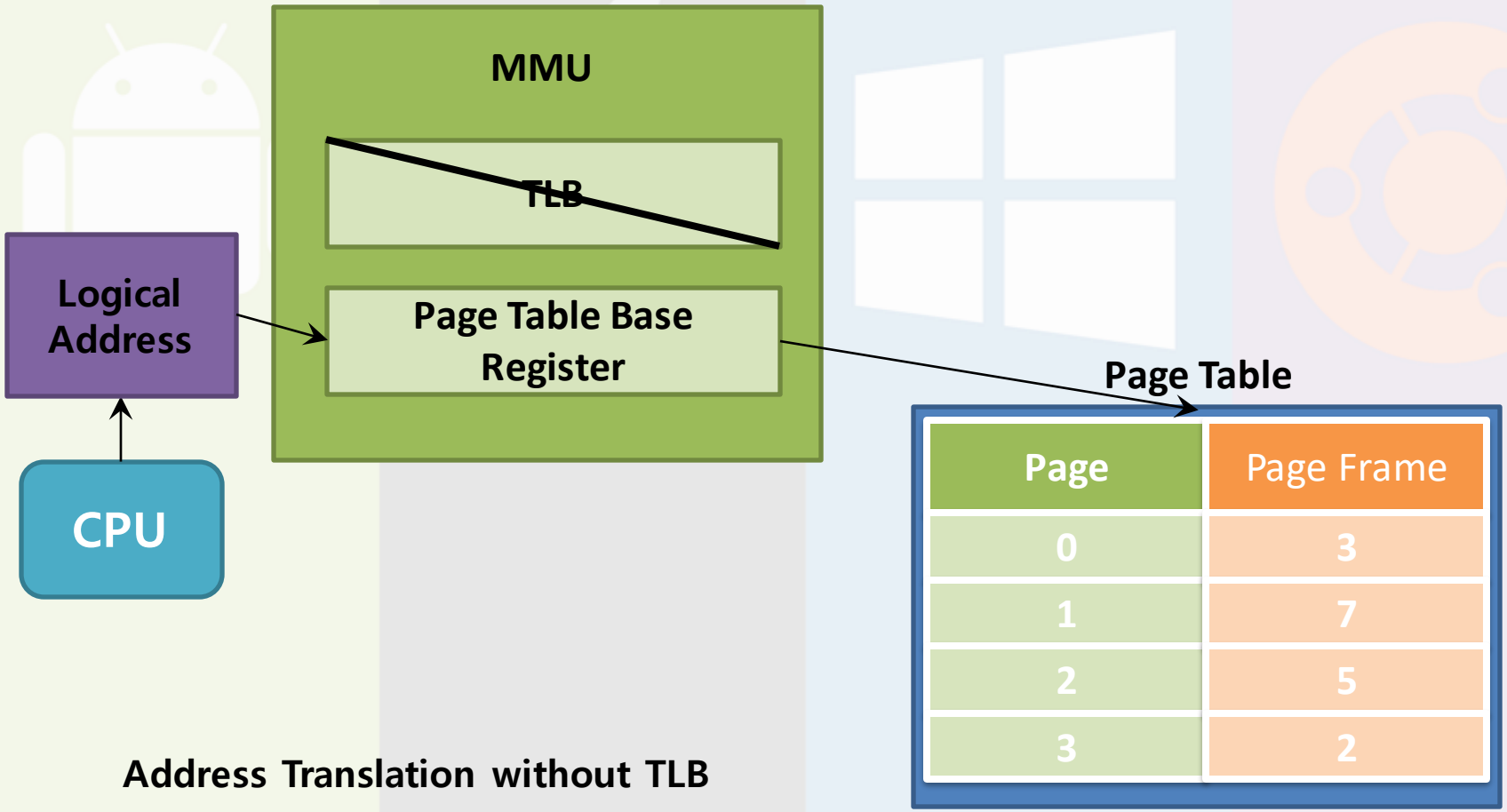
Address Translation without TLB

Translation Lookaside Buffers



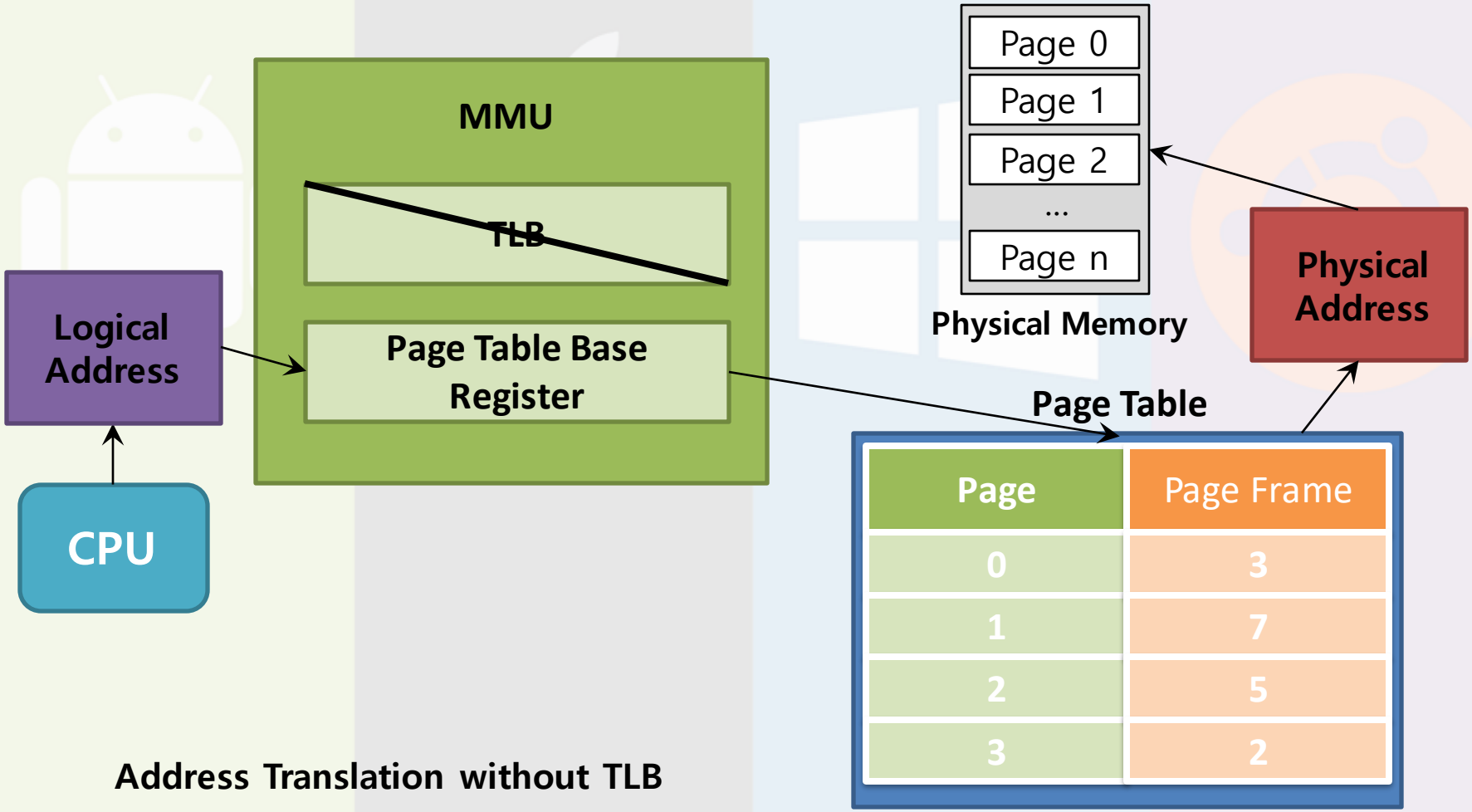
Address Translation without TLB

Translation Lookaside Buffers



Address Translation without TLB

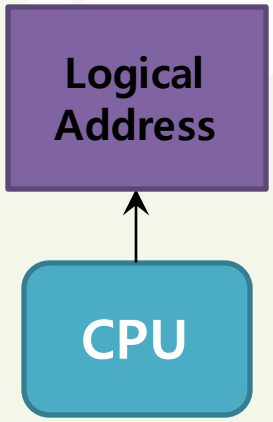
Translation Lookaside Buffers



Address Translation without TLB

Translation Lookaside Buffers

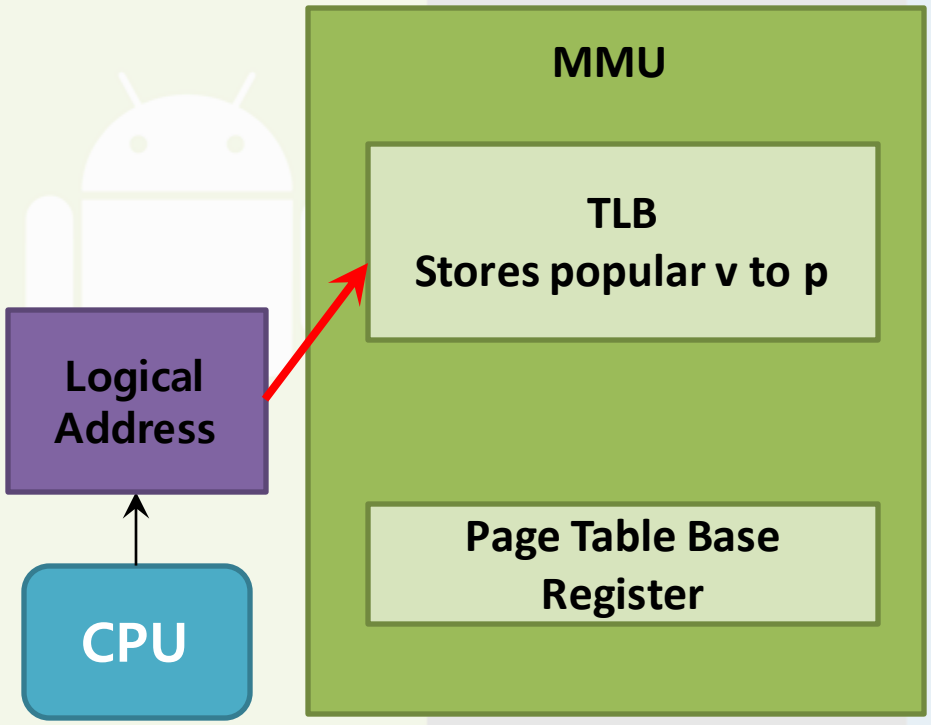
- The TLB is a hardware cache of popular virtual-to-physical address translation



Address Translation with TLB

Translation Lookaside Buffers

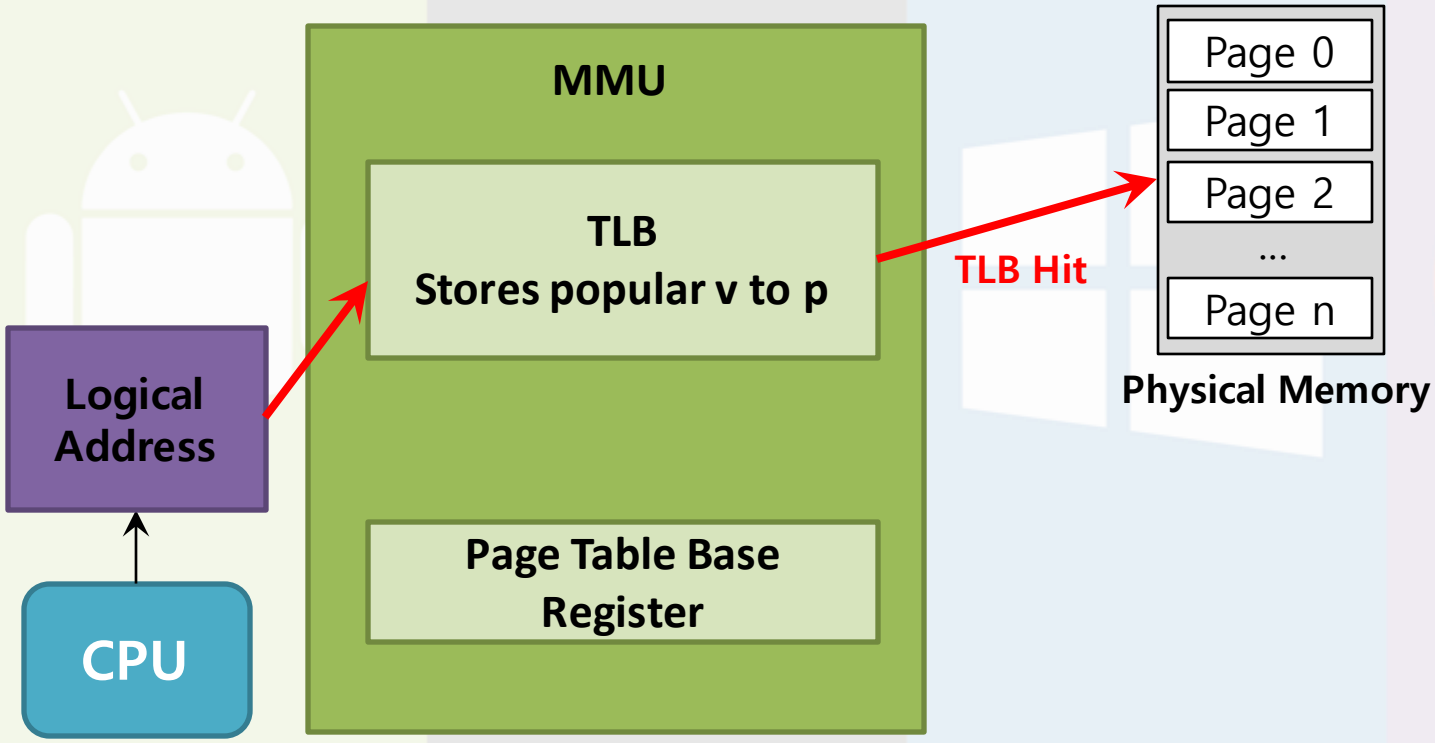
- The TLB is a hardware cache of popular virtual-to-physical address translation



Address Translation with TLB

Translation Lookaside Buffers

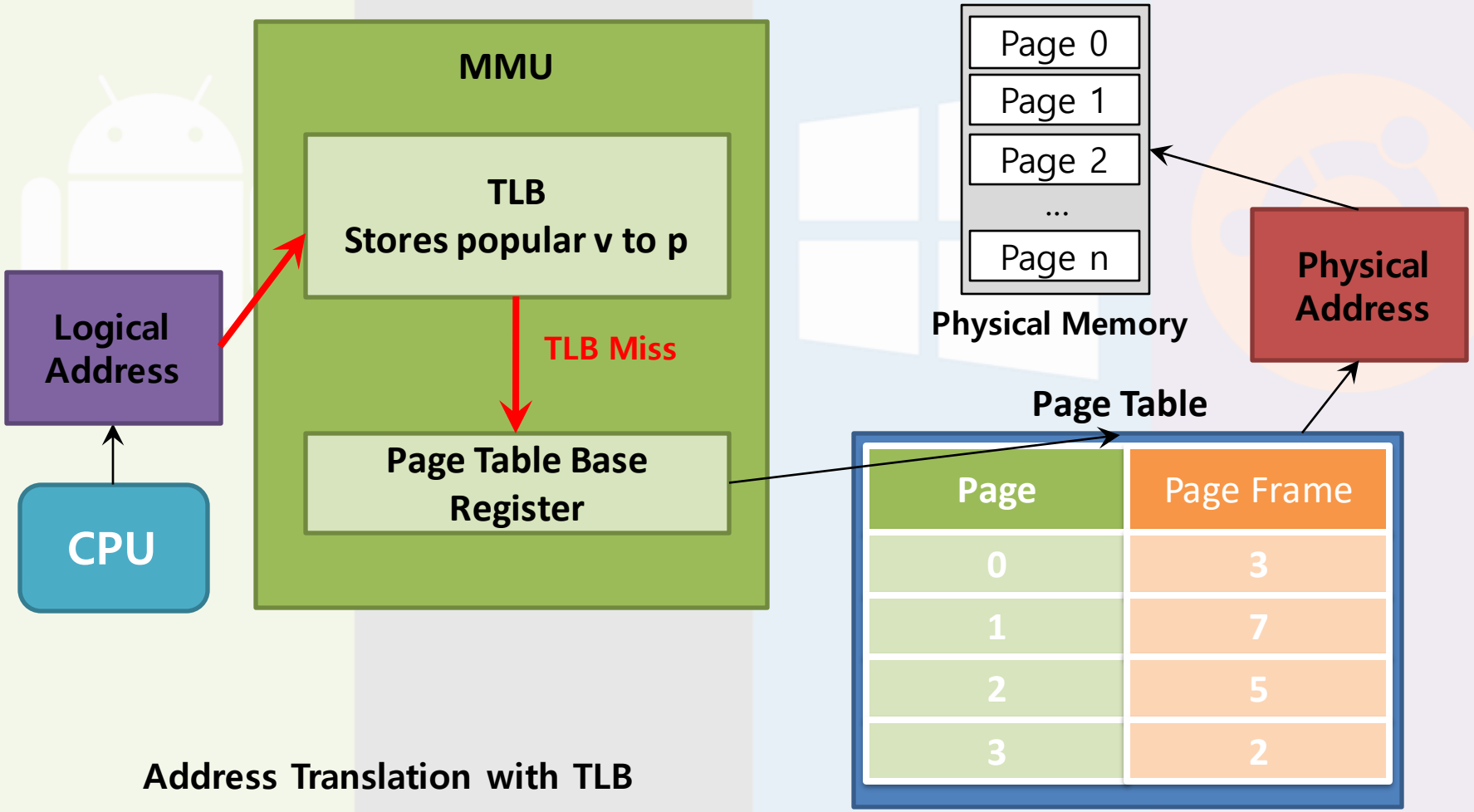
- The TLB is a hardware cache of popular virtual-to-physical address translation



Address Translation with TLB

Translation Lookaside Buffers

- The TLB is a hardware cache of popular virtual-to-physical address translation



Address Translation with TLB

TLB ALgortihm

```
1: VPN = (VirtualAddress & VPN_MASK ) >> SHIFT
2: (Success , TlbEntry) = TLB_Lookup(VPN)
3:    if(Success == Ture){ // TLB Hit
4:        if(CanAccess(TlbEntry.ProtectBit) == True ){
5:            offset = VirtualAddress & OFFSET_MASK
6:            PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
7:            AccessMemory( PhysAddr )
8:        }else RaiseException(PROTECTION_ERROR)
```

- (1) extract the virtual page number(VPN).
- (2) check if the TLB holds the translation for this VPN.
- (5-8) extract the page frame number from the relevant TLB entry, and form the desired physical address and access memory.

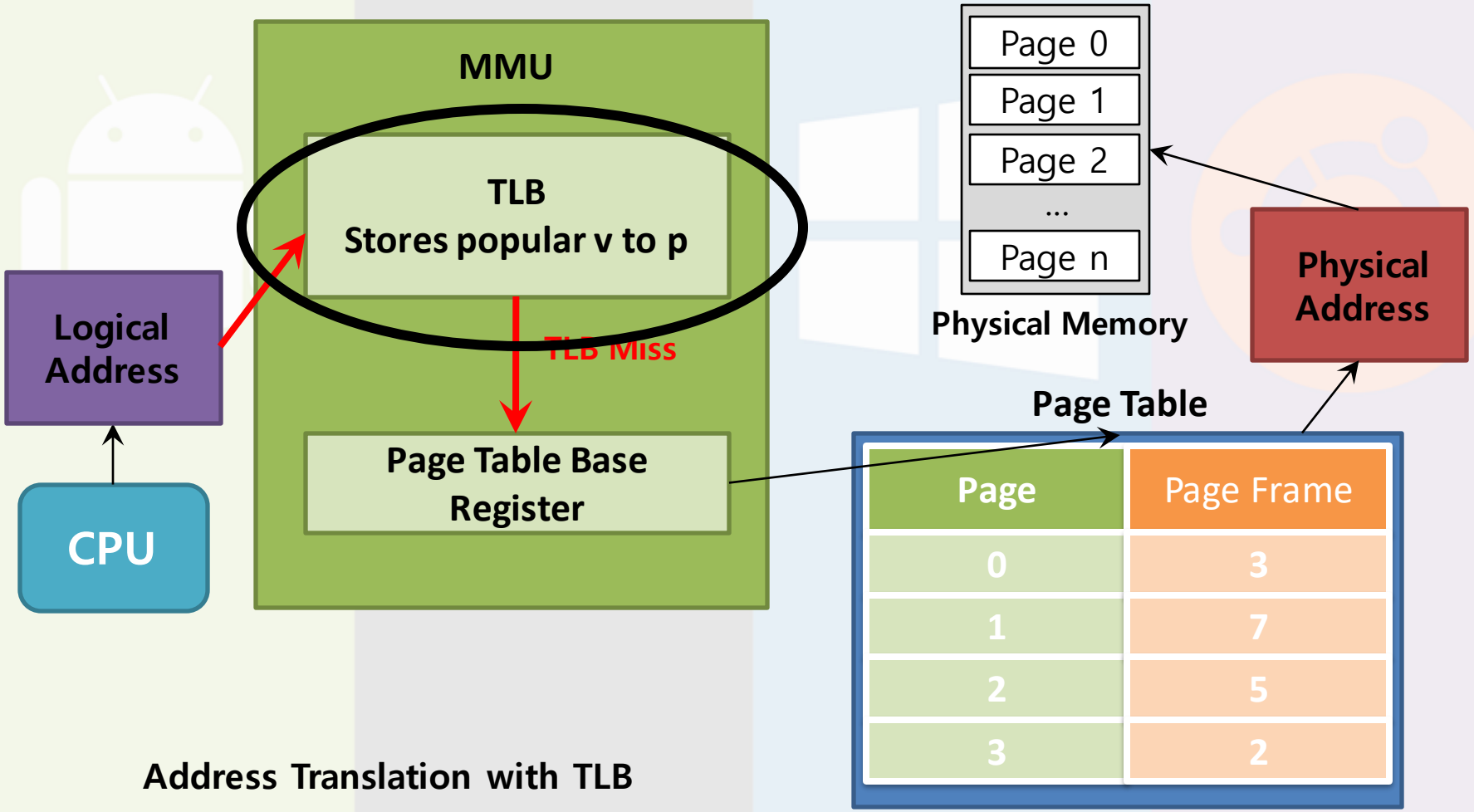
TLB ALgortihm

```
11:     }else{ //TLB Miss
12:          PTEAddr = PTBR + (VPN * sizeof(PTE))
13:          PTE = AccessMemory(PTEAddr)
14:          if(PTE.VALID == FALSE){
15:               RaiseException(SEGMENTATION_FAULT)
16:          else if (CanAccess(PTE.ProtectBits) == False){
17:               RaiseException(PROTECTION_FAULT)
18:          }else{
19:               TLB_Insert( VPN , PTE.PFN , PTE.ProtectBits)
20:               RetryInstruction()
21:           }
22:}
```

- (12-13) The hardware accesses the page table to find the translation.
- (19) updates the TLB with the translation.

A TLB Entry

- The TLB is a hardware cache of popular virtual-to-physical address translation



Address Translation with TLB

A TLB Entry

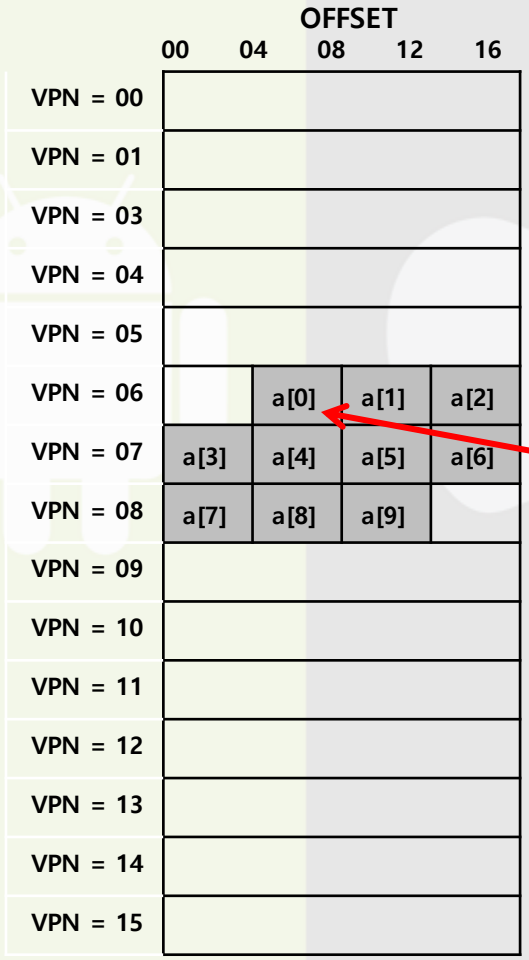
- **TLB is managed by Full Associative method.**
  - A typical TLB might have 32, 64, or 128 entries.
  - Hardware search the entire TLB in parallel to find the desired translation.
  - Other bits: valid bits , protection bits, address-space identifier, dirty bit

Typical TLB entry look like this

Example : Accessing an Array

```
0:      int sum = 0 ;
1:      for( i=0; i<10; i++){
2:              sum+=a[i];
3:      }
```

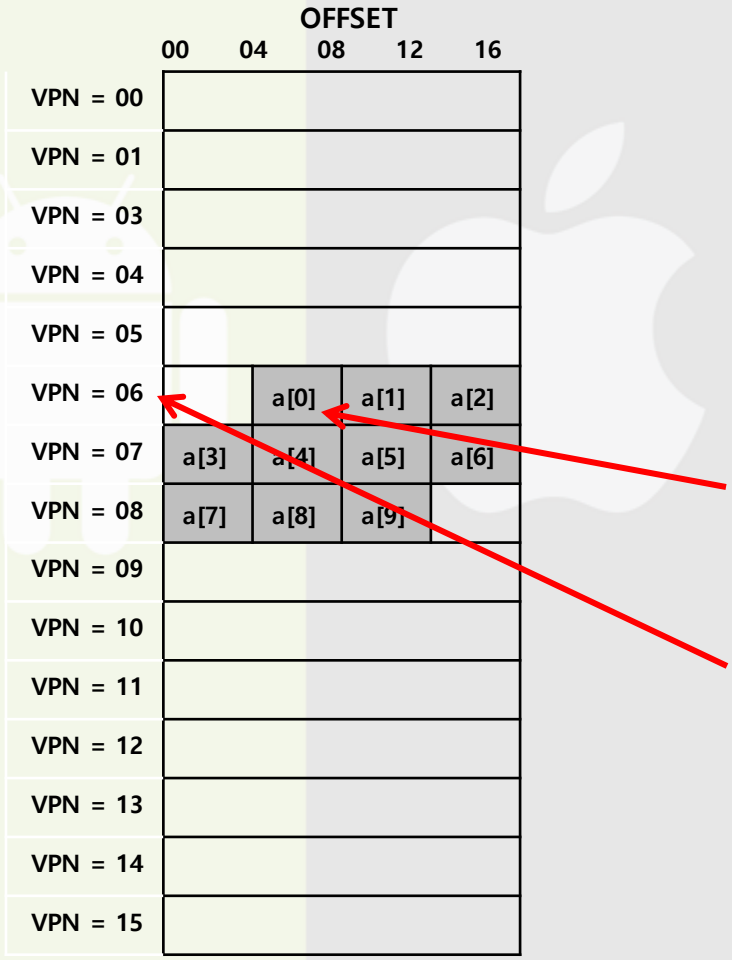
Example : Accessing an Array




```
0:      int sum = 0 ;
1:      for( i=0; i<10; i++){
2:              sum+=a[i];
3:      }
```

- When the first array element (a[0]) is accessed, the CPU will see a load to virtual address 100

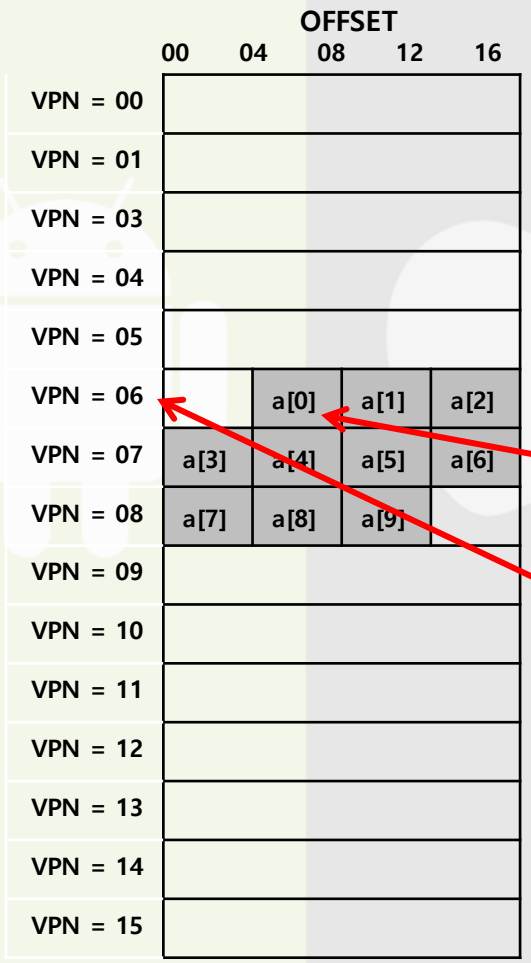
Example : Accessing an Array




```
0:      int sum = 0 ;
1:      for( i=0; i<10; i++){
2:              sum+=a[i];
3:      }
```

- When the first array element (a[0]) is accessed, the CPU will see a load to virtual address 100
- The hardware extracts the VPN from this (VPN=06), and uses that to check the TLB for a valid translation.

Example : Accessing an Array

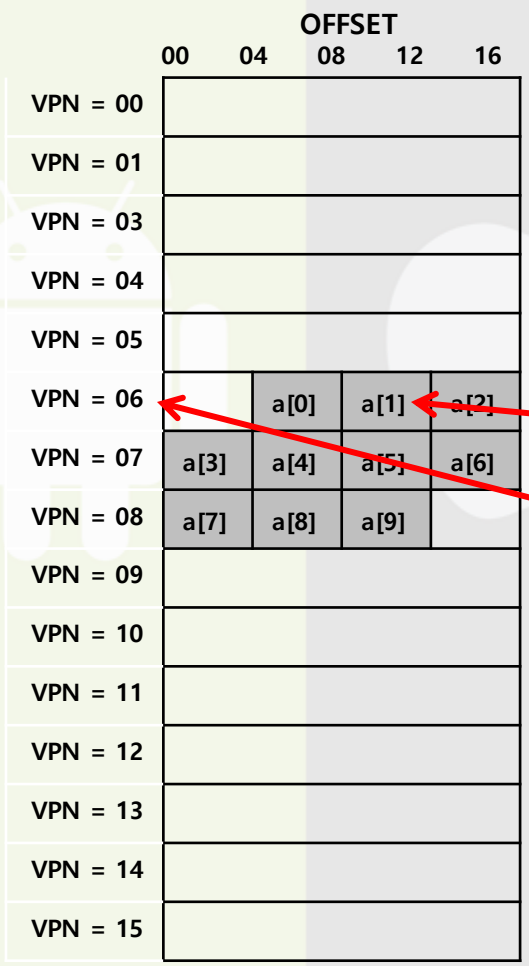


TLB

VPN	PFN	other bits
06	03	-

- When the first array element (a[0]) is accessed, the CPU will see a load to virtual address 100
- The hardware extracts the VPN from this (VPN=06), and uses that to check the TLB for a valid translation.
- Assuming this is the first time the program accesses the array, the result will be a TLB miss.
- Add new TLB Entry

Example : Accessing an Array

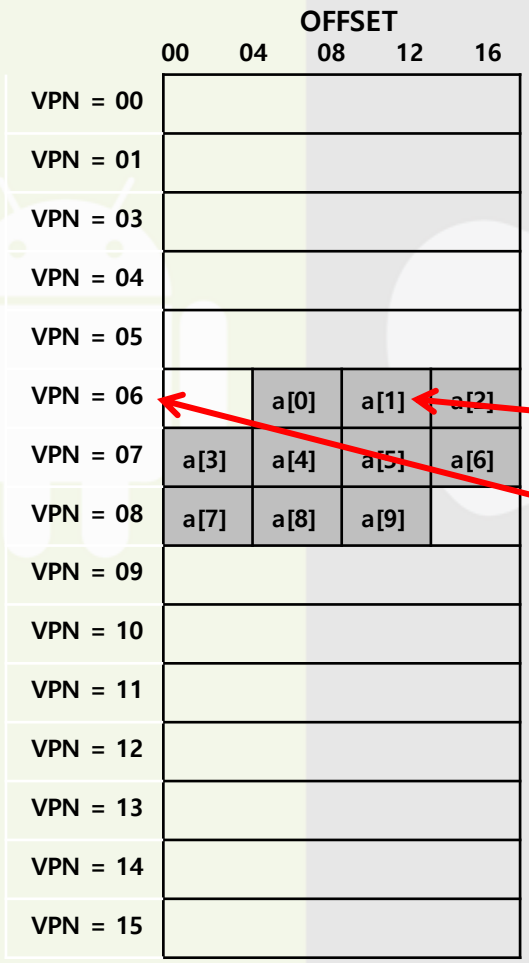


TLB

VPN	PFN	other bits
06	03	-

- The next access is to a[1]
- The hardware extracts the VPN from this (VPN=06), and uses that to check the TLB for a valid translation.

Example : Accessing an Array

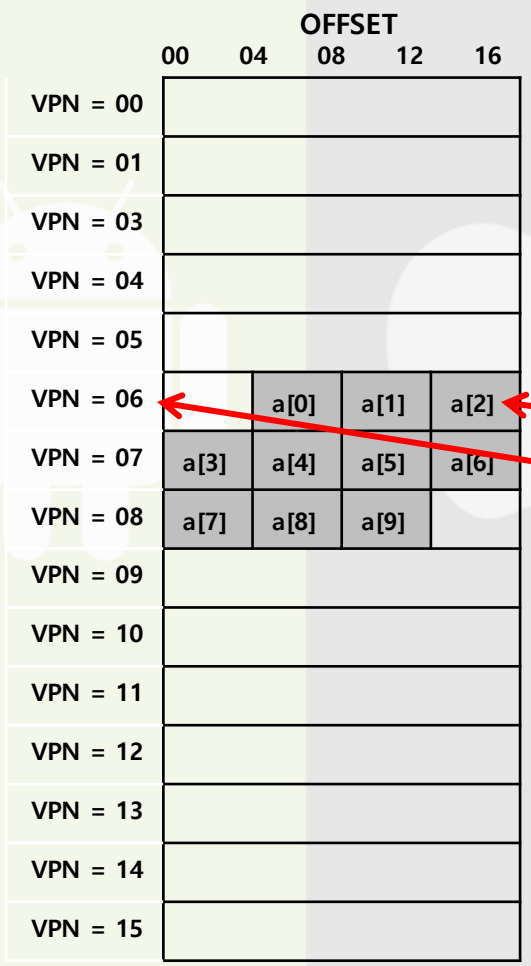


TLB

VPN	PFN	other bits
06	03	-

- The next access is to a[1]
- The hardware extracts the VPN from this (VPN=06), and uses that to check the TLB for a valid translation.
- Some good news here: a TLB hit! Because the second element of the array is packed next to the first, it lives on the same page which had already been loaded into the TLB

Example : Accessing an Array

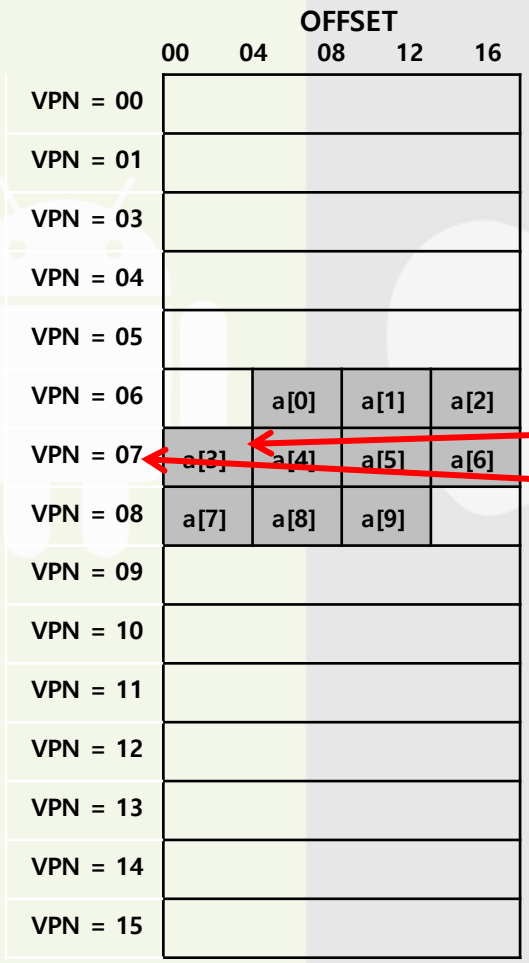


TLB

VPN	PFN	other bits
06	03	-

- The next access is to a[2]
- (VPN=06)
- TLB hit!

Example : Accessing an Array

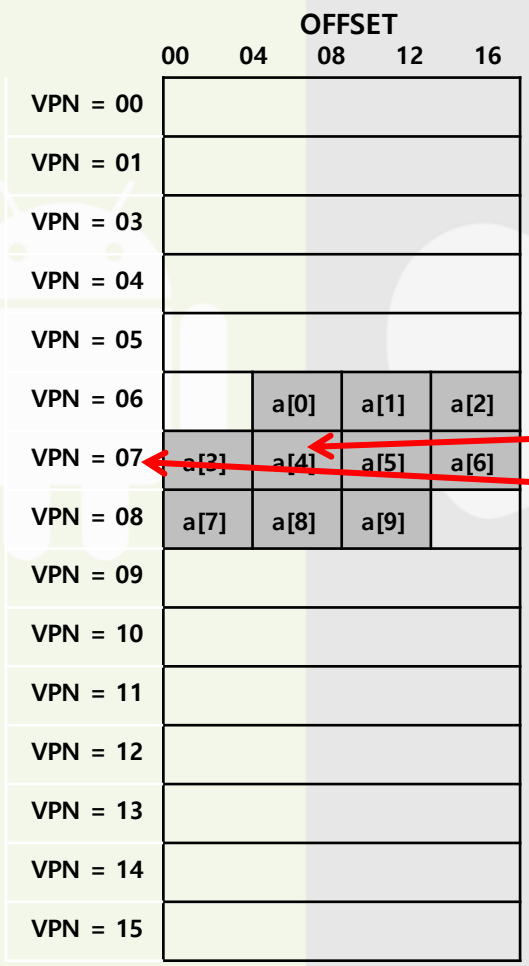


TLB

VPN	PFN	other bits
06	03	-
07	08	-

- The next access is to a[3]
- (VPN=07)
- TLB miss!
- Load translation into TLB

Example : Accessing an Array

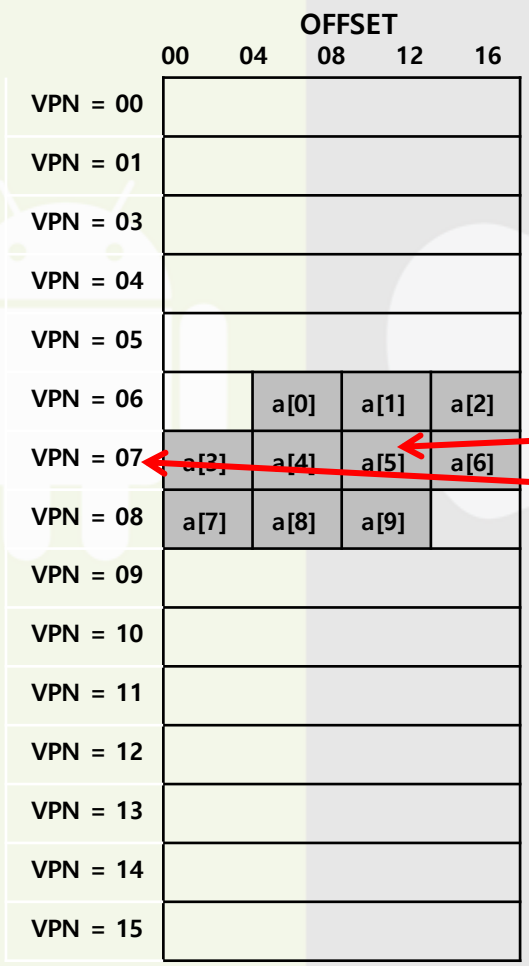


TLB

VPN	PFN	other bits
06	03	-
07	08	-

- The next access is to a[4]
- (VPN=07)
- TLB hit!

Example : Accessing an Array

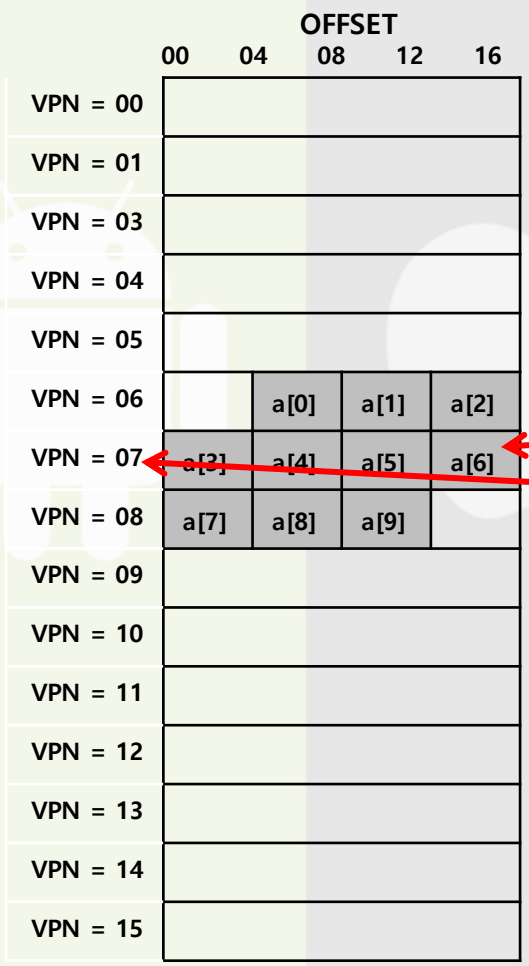


TLB

VPN	PFN	other bits
06	03	-
07	08	-

- The next access is to a[5]
- (VPN=07)
- TLB hit!

Example : Accessing an Array

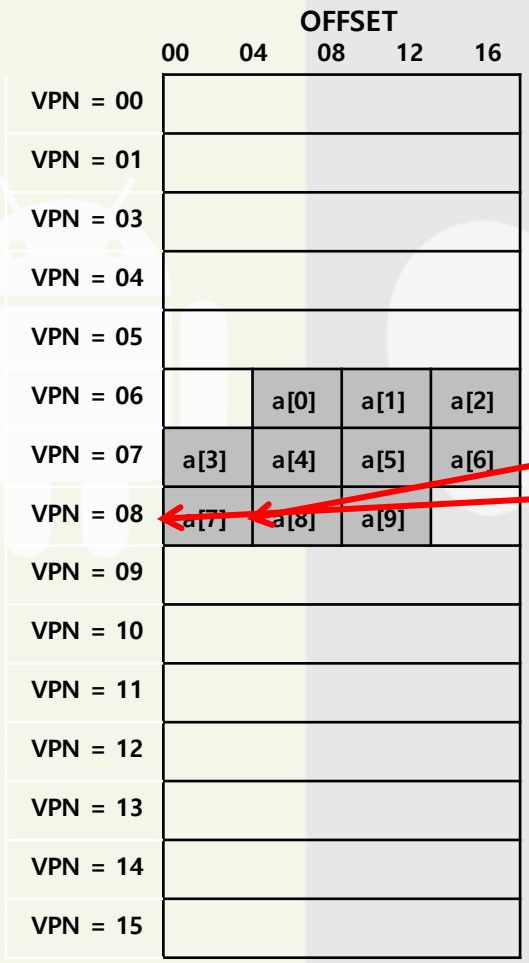


TLB

VPN	PFN	other bits
06	03	-
07	08	-

- The next access is to a[6]
- (VPN=07)
- TLB hit!

Example : Accessing an Array

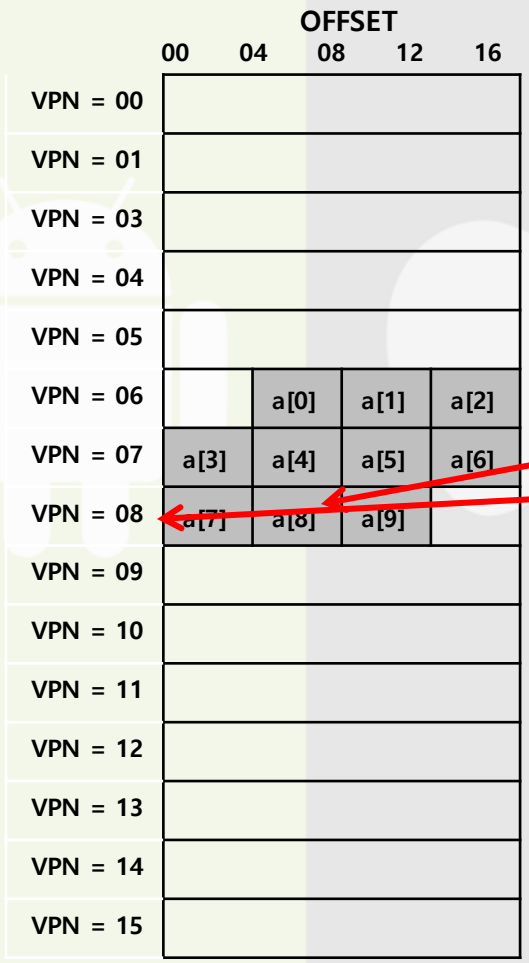


TLB

VPN	PFN	other bits
06	03	-
07	08	-
08	02	-

- The next access is to a[7]
- (VPN=08)
- TLB miss!
- Load translation into TLB

Example : Accessing an Array

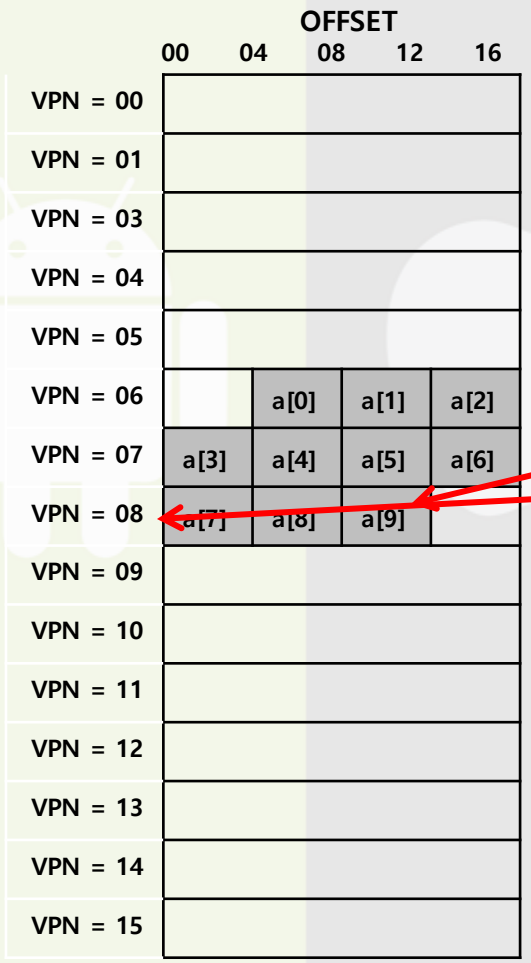


TLB

VPN	PFN	other bits
06	03	-
07	08	-
08	02	-

- The next access is to a[8]
- (VPN=08)
- TLB hit!

Example : Accessing an Array

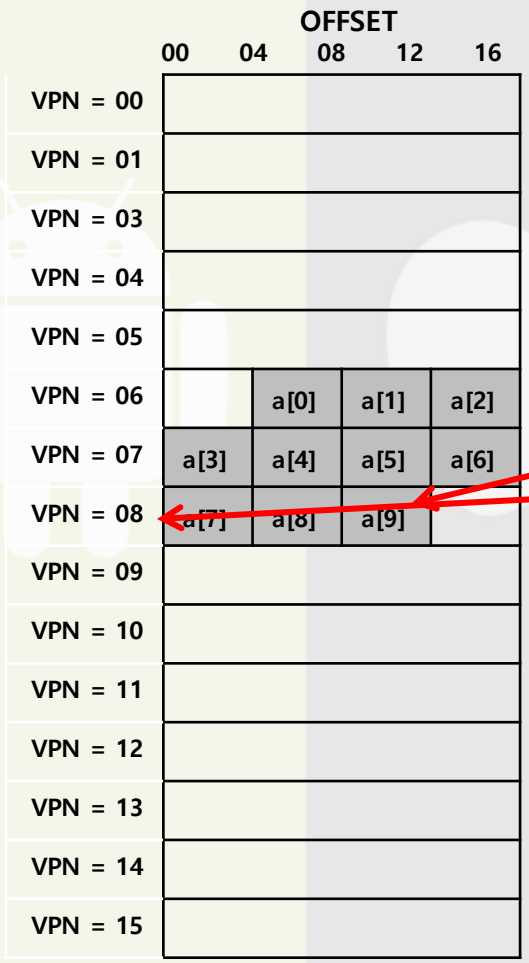


TLB

VPN	PFN	other bits
06	03	-
07	08	-
08	02	-

- The next access is to a[9]
- (VPN=08)
- TLB hit!

Example : Accessing an Array



TLB

VPN	PFN	other bits
06	03	-
07	08	-
08	02	-

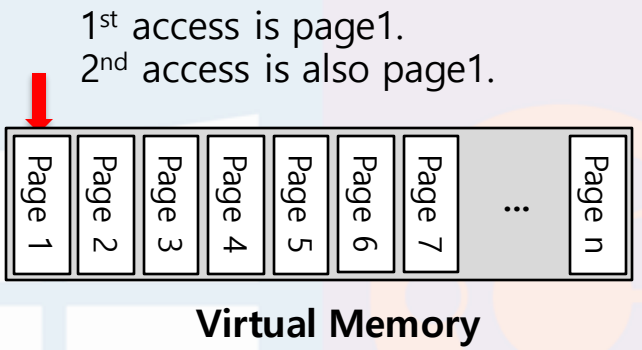
- The next access is to a[9]
- (VPN=08)
- TLB hit!

The TLB improves performance due to spatial locality

3 misses and 7 hits.
Thus TLB hit rate is 70%.

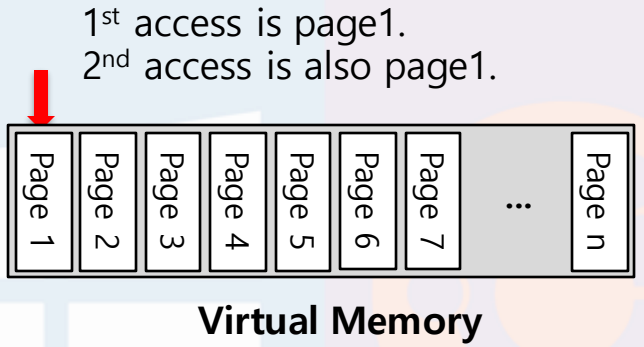
Locality

- **Temporal Locality**
  - An instruction or data item that has been recently accessed will likely be re-accessed soon in the future

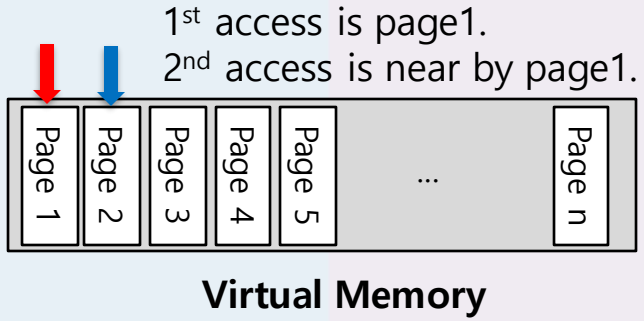


Locality

- **Temporal Locality**
  - An instruction or data item that has been recently accessed will likely be re-accessed soon in the future



- **Spatial Locality**
  - If a program accesses memory at address x, it will likely soon access memory near x



Who handles the TLB Miss?

- Well there are 2 approaches

Who handles the TLB Miss?

- Hardware handle the TLB miss entirely on CISC

Who handles the TLB Miss?

- **Hardware handle the TLB miss entirely on CISC**
  - The hardware has to know exactly where the page tables are located in memory.
  - The hardware would “walk” the page table, find the correct page-table entry and extract the desired translation, update and retry instruction.
  - **hardware-managed TLB.**

Who handles the TLB Miss?

- **Hardware handle the TLB miss entirely on CISC**
  - The hardware has to know exactly where the page tables are located in memory.
  - The hardware would “walk” the page table, find the correct page-table entry and extract the desired translation, update and retry instruction.
  - **hardware-managed TLB.**

- **RISC have what is known as a software-managed TLB**

Who handles the TLB Miss?

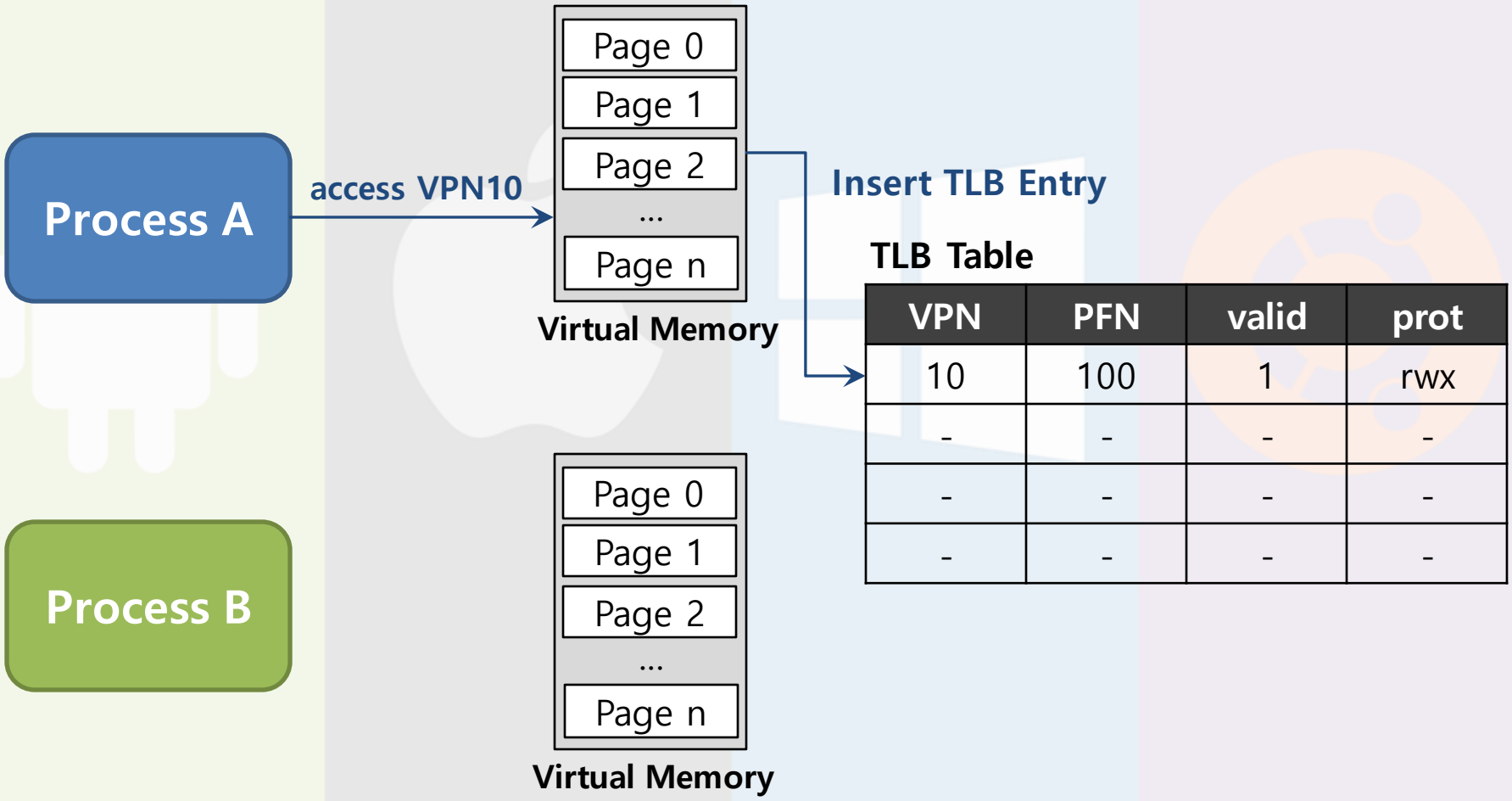
- **Hardware handle the TLB miss entirely on CISC**
  - The hardware has to know exactly where the page tables are located in memory.
  - The hardware would “walk” the page table, find the correct page-table entry and extract the desired translation, update and retry instruction.
  - **hardware-managed TLB**
  - Complex Instruction Set Computing

- **RISC have what is known as a software-managed TLB**
  - On a TLB miss, the hardware raises exception(trap handler).
  - **Trap handler is code** within the OS that is written with the express purpose of handling TLB miss.
  - Reduced instruction set computer

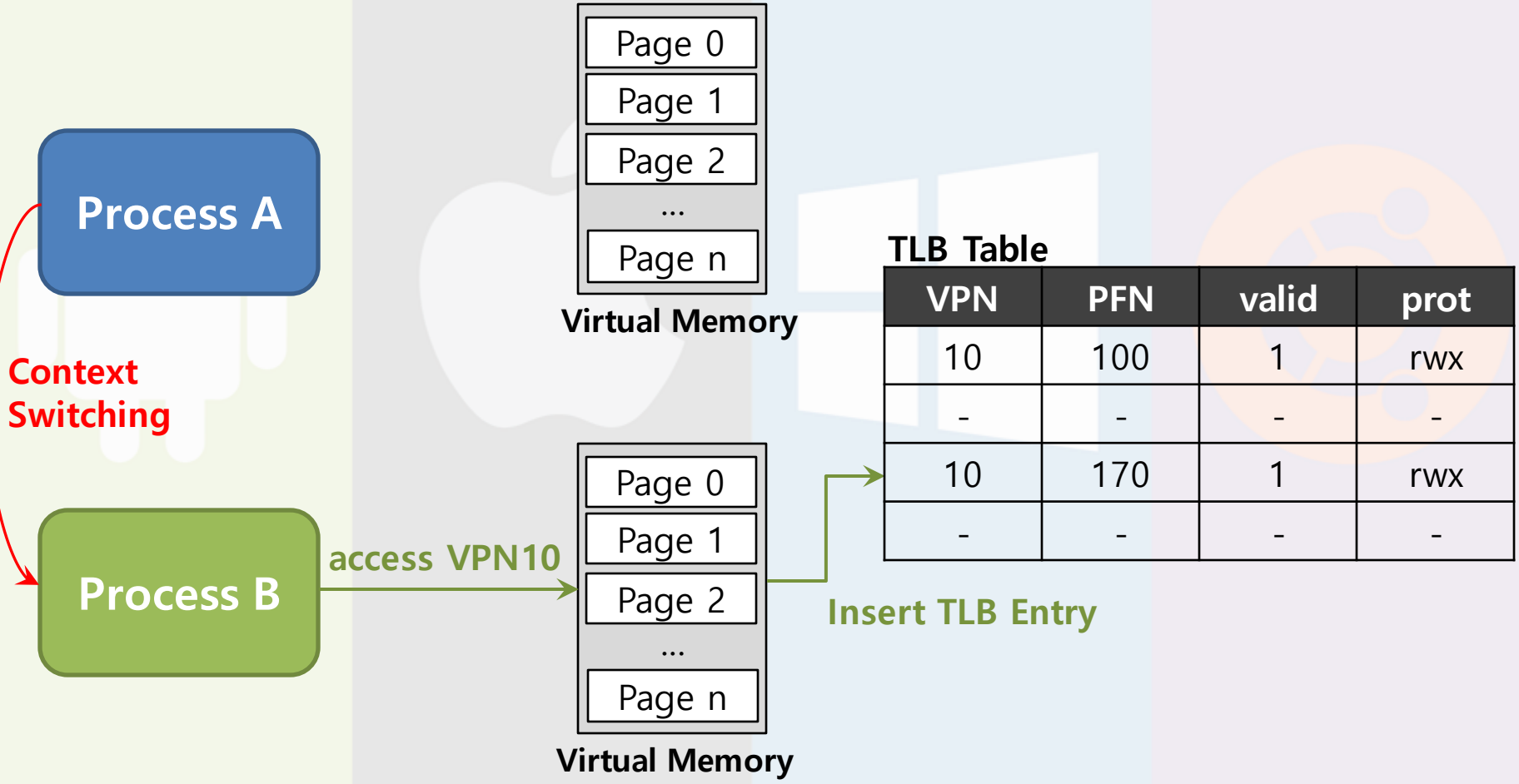
Ok so job Done?

- The TLB in its current form solves all our problems

TLB Issue : Context Switching



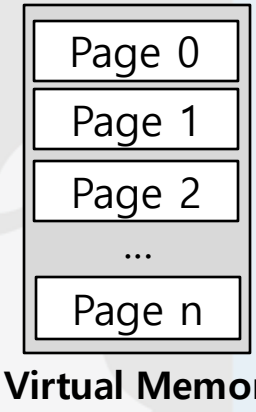
TLB Issue : Context Switching



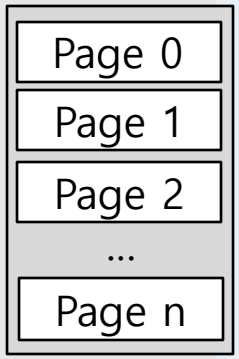
TLB Issue : Context Switching

Process A

Process B



Virtual Memory



Virtual Memory

TLB Table

VPN	PFN	valid	prot
10	100	1	rwx
-	-	-	-
10	170	1	rwx
-	-	-	-

Can't Distinguish which entry is meant for which process

TLB Issue : Page Sharing

- **Two processes share a page.**
  - Process 1 is sharing physical page 101 with Process2
  - P1 maps this page into the 10th page of its address space
  - P2 maps this page to the 50th page of its address space

VPN	PFN	valid	prot
10	101	1	rwx
-	-	-	-
50	101	1	rwx
-	-	-	-

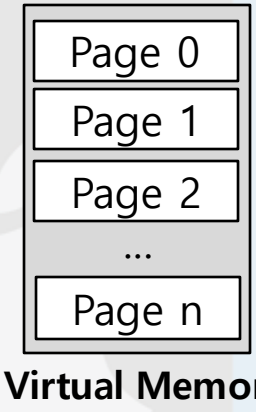
Sharing of pages is useful as it reduces the number of physical pages in use.

TLB Issue : Context Switching - Solution

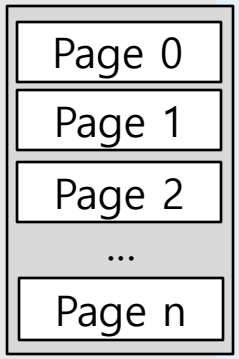
- Provide an address space identifier(ASID) field in the TLB

Process A

Process B



Virtual Memory



Virtual Memory

TLB Table

VPN	PFN	valid	prot	ASID
10	100	1	rwx	1
-	-	-	-	-
10	170	1	rwx	2
-	-	-	-	-

What happens when we want to add a new entry

- Problem – We need to add a new entry to the TLB
  - Well where do we put it and if needed which entry do we replace

TLB Issue : Replacement Policy

- In order to add new entries a replacement policy is used
- The general idea with these policies is that you want to minimize the miss rate and hence maximise the hit rate


TLB Issue : Replacement Policy

- **In order to add new entries a replacement policy is used**
- **The general idea with these policies is that you want to minimize the miss rate and hence maximise the hit rate**

- LIFO
- FIFO
- Random
- LRU

TLB Issue : Replacement Policy - Solution

- LRU (LEAST RECENTLY USED)
  - Evict an entry that has not recently been used.
  - Take advantage of locality in the memory-reference stream

TLB Issue : Replacement Policy - Solution

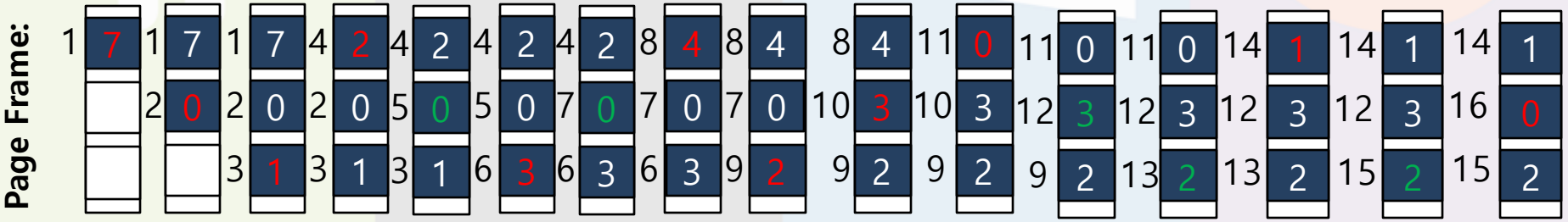
- **LRU (LEAST RECENTLY USED)**
  - **Evict an entry that has not recently been used.**
  - **Take advantage of locality in the memory-reference stream**
    - Add a register to every page frame - contain the last time that the page in that frame was accessed
    - Use a "logical clock" that advance by 1 tick each time a memory reference is made.
    - Each time a page is referenced, update its register

TLB Issue : Replacement Policy - Solution

- **LRU (LEAST RECENTLY USED)**
  - Evict an entry that has not recently been used.
  - Take advantage of locality in the memory-reference stream

Page Reference Row

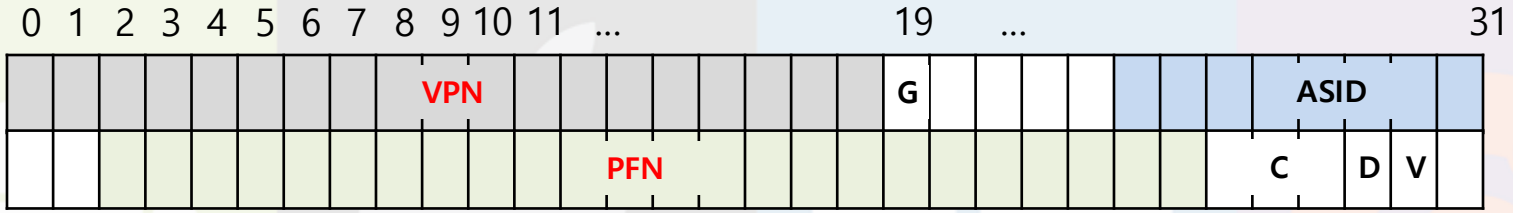
7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0



Total 11 TLB miss

A Real TLB Entry

All 64 bits of this TLB entry(example of MIPS R4000)



Flag	Content
19-bit VPN	The rest reserved for the kernel.
24-bit PFN	Systems can support with up to 64GB of main memory($2^{24} * 4KB$ pages)
Global bit(G)	Used for pages that are globally-shared among processes.
ASID	OS can use to distinguish between address spaces.
Coherence bit(C)	determine how a page is cached by the hardware.
Dirty bit(D)	marking when the page has been written.
Valid bit(V)	tells the hardware if there is a valid translation present in the entry.