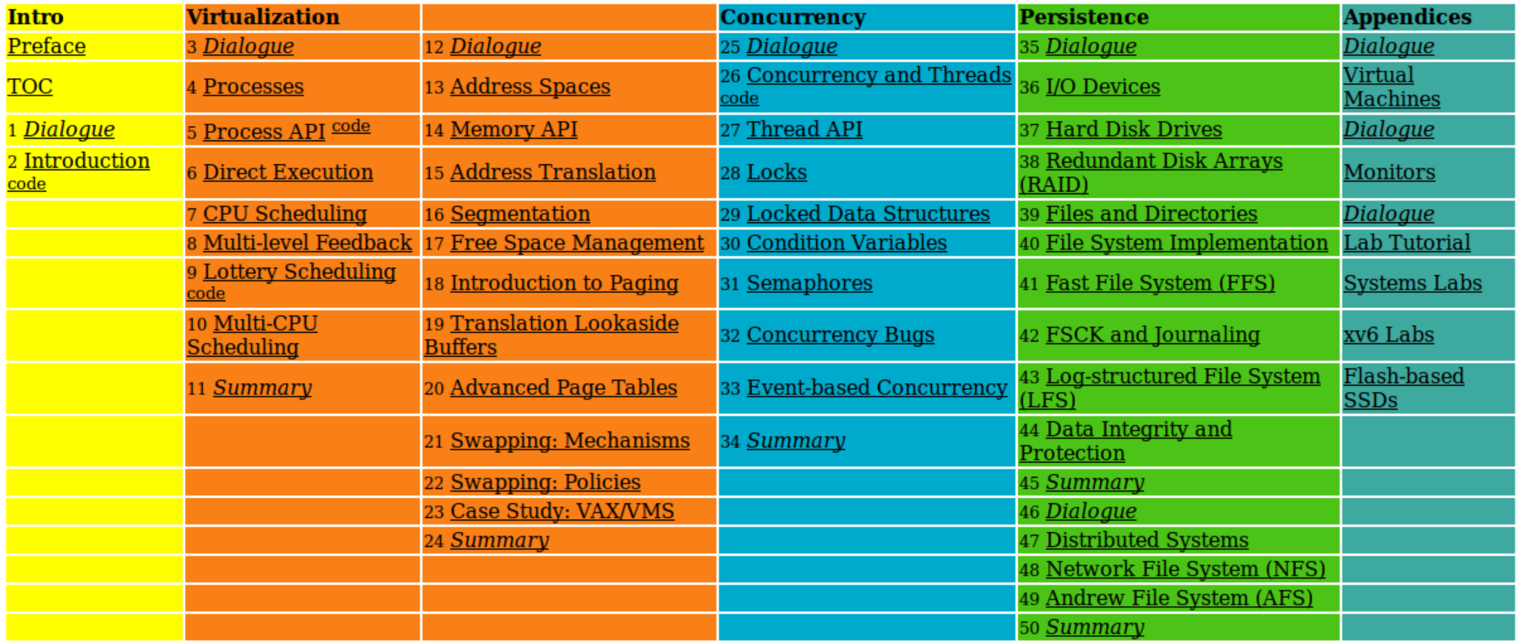
CSIS 429 Operating Systems

Lecture 8: Faster Translations

September 30th 2020

Textbook chapters

- Read “Intro to Paging” and “Translation LB”



Intro	**Virtualization**		**Concurrency**	**Persistence**	**Appendices**
Preface	3 *Dialogue*	12 *Dialogue*	25 *Dialogue*	35 *Dialogue*	*Dialogue*
TOC	4 Processes	13 Address Spaces	26 Concurrency and Threads code	36 I/O Devices	Virtual Machines
1 *Dialogue*	5 Process API code	14 Memory API	27 Thread API	37 Hard Disk Drives	*Dialogue*
2 Introduction code	6 Direct Execution	15 Address Translation	28 Locks	38 Redundant Disk Arrays (RAID)	Monitors
	7 CPU Scheduling	16 Segmentation	29 Locked Data Structures	39 Files and Directories	*Dialogue*
	8 Multi-level Feedback	17 Free Space Management	30 Condition Variables	40 File System Implementation	Lab Tutorial
	9 Lottery Scheduling code	18 Introduction to Paging	31 Semaphores	41 Fast File System (FFS)	Systems Labs
	10 Multi-CPU Scheduling	19 Translation Lookaside Buffers	32 Concurrency Bugs	42 FSCK and Journaling	xv6 Labs
	11 *Summary*	20 Advanced Page Tables	33 Event-based Concurrency	43 Log-structured File System (LFS)	Flash-based SSDs
		21 Swapping: Mechanisms	34 *Summary*	44 Data Integrity and Protection	
		22 Swapping: Policies		45 *Summary*	
		23 Case Study: VAX/VMS		46 *Dialogue*	
		24 *Summary*		47 Distributed Systems	
				48 Network File System (NFS)	
				49 Andrew File System (AFS)	
				50 *Summary*	

Address Translation with Paging

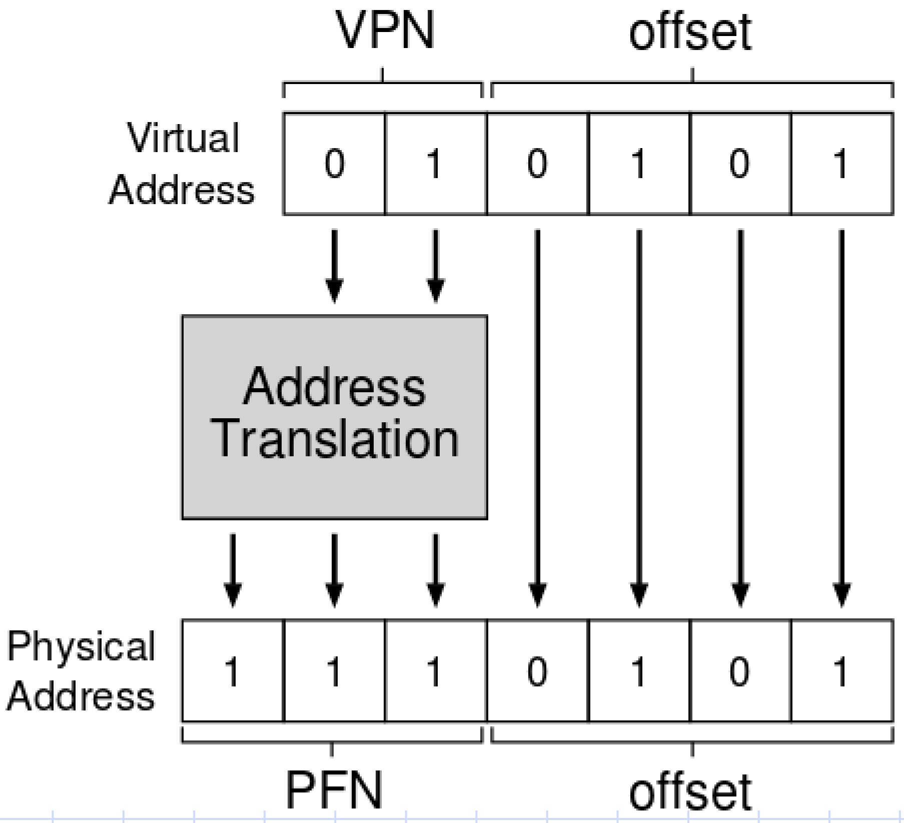
Example: Instruction to load data from memory to register

```
movl <virt-addr>, %eax
```

64 byte virtual address space

page size is 16 bytes

128 byte physical address space



What about instructions?

In our old example - instruction to load data from memory to register

```
movl 21, %a
```

We did not account for time taken to read the instruction which is at some virtual address which has to be converted to a physical address – by accessing a Page Table!!

Can all this work?   Yes, but we need some help from H/w

Example of a loop

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

The last statement gets converted to:

```
1024 movl $0x0,(%edi,%eax,4)
1028 incl %eax
1032 cmpl $0x03e8,%eax
1036 jne  0x1024
```

edi has the base address of “array”

And eax is index I

~ edi [eax * 4]

Memory access in loop

The `array[i] = 0;` statement becomes:

```
1024 movl $0x0,(%edi,%eax,4)
1028 incl %eax
1032 cmpl $0x03e8,%eax
1036 jne  0x1024
```

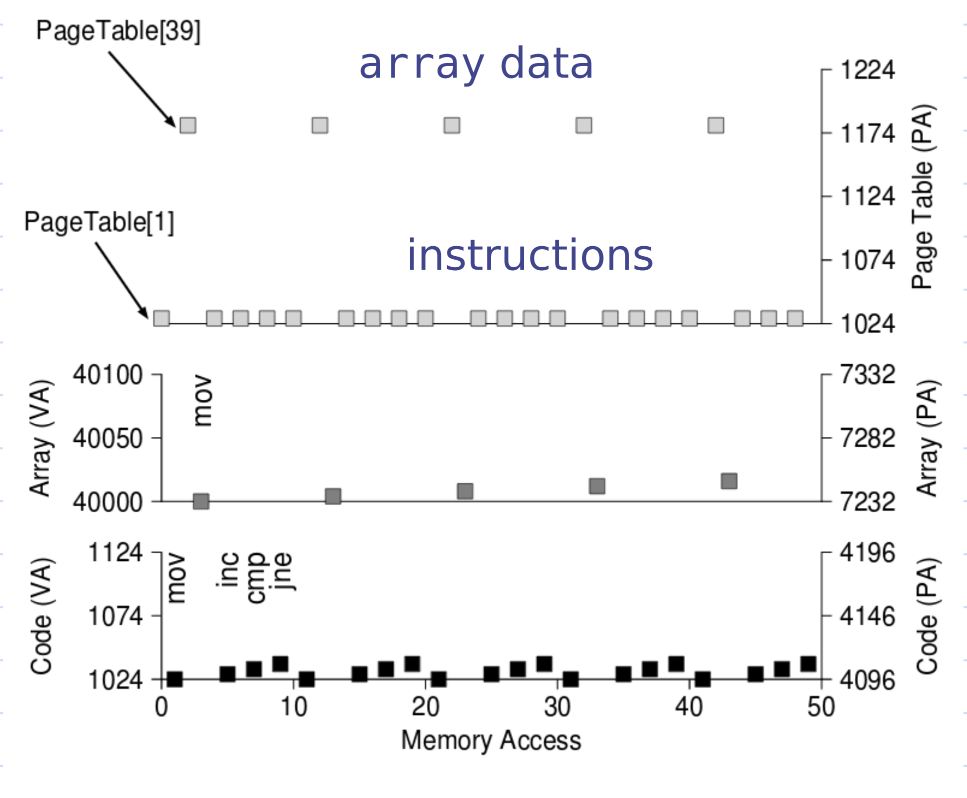


Figure 18.7: **A Virtual (And Physical) Memory Trace**

The Crux of the Problem

How can we speed up Address Translation using Page Tables?

We want to avoid all those memory references.

What hardware support is required?

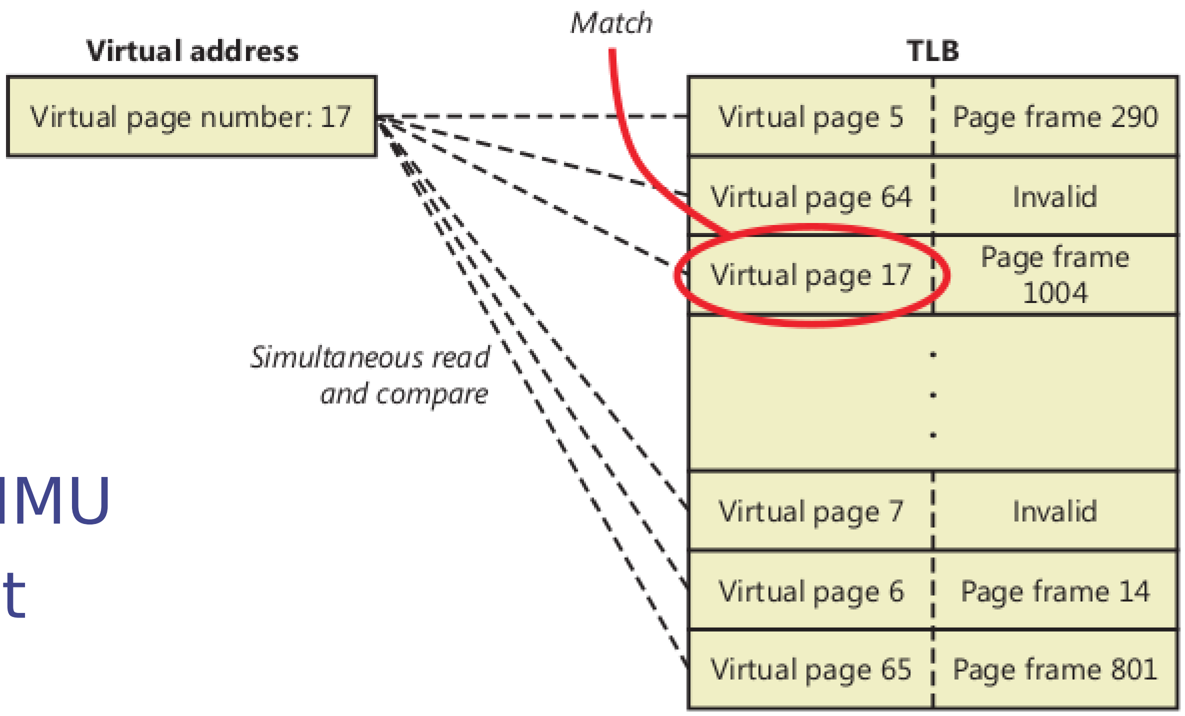
What will the OS need to do?

The Answer: Translation Lookaside Buffer

To speed up Address Translation using Page Tables we will use a Translation Lookaside Buffer - TLB

The TLB is part of the MMU

It is a special cache that allows fast searches



Translation Lookaside Buffer

A Translation Lookaside Buffer speeds up Address Translation

- part of the MMU
- special cache that allows fast searches

For each virtual memory reference, the CPU first checks the TLB to see if the desired translation is there; if so → **TLB hit**! → the translation is done – in 1 CPU clock cycle!

No need to access the page table to get the physical address!

Translation Lookaside Buffer

For each virtual memory reference, the CPU first checks the TLB to see if the desired translation is there; if so → **TLB hit**! → the translation is done – in 1 CPU clock cycle!

No need to access the page table to get the physical address!

If the virtual to physical address translation is not in the TLB → **TLB miss**. OS will have to access the page table and bring the entry into the TLB so that it is available. Retry translation → TLB hit this time.

Translation Lookaside Buffer

Every virtual memory reference will involve the TLB to speed address translations. A TLB miss is expensive → shows up as slow execution time.

We can see speed-ups due to TLB hit rate in our programs – mainly in executing loops.

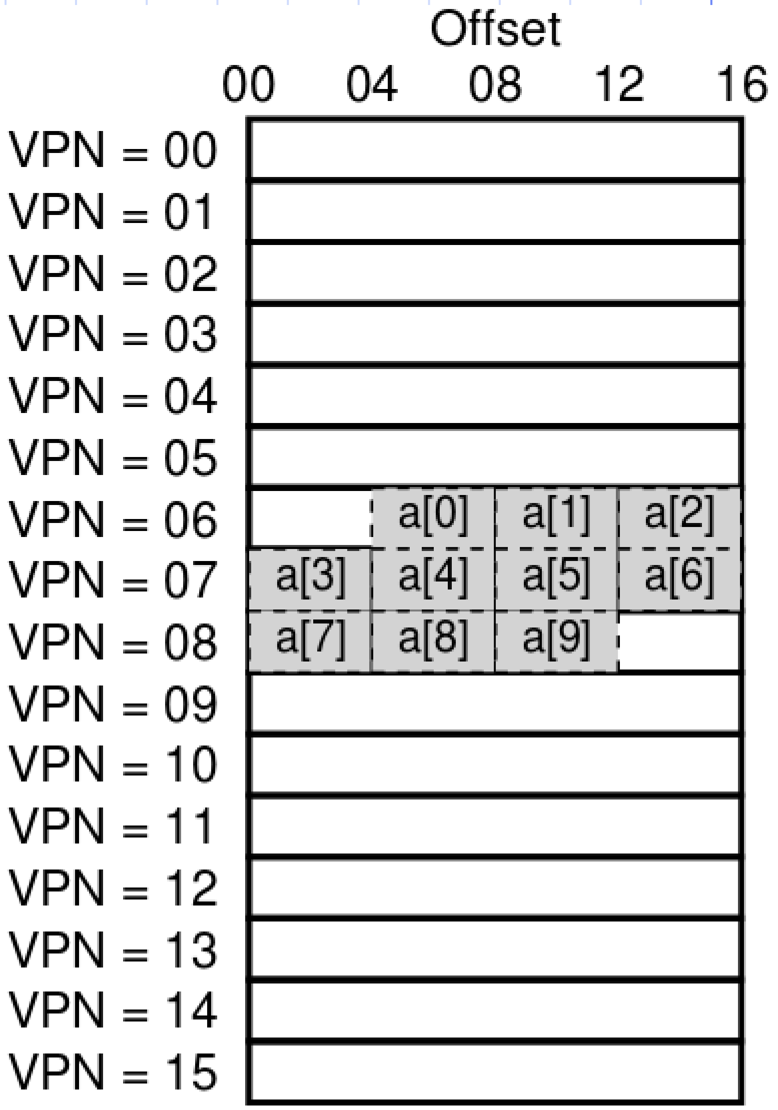
Why loops? Because repeatedly accessing the same or nearby data can increase the TLB hit rate if we do it right.

Array Access Example

Small 8-bit virtual address space divided into 16 pages.

16-byte pages → 4-bit offset

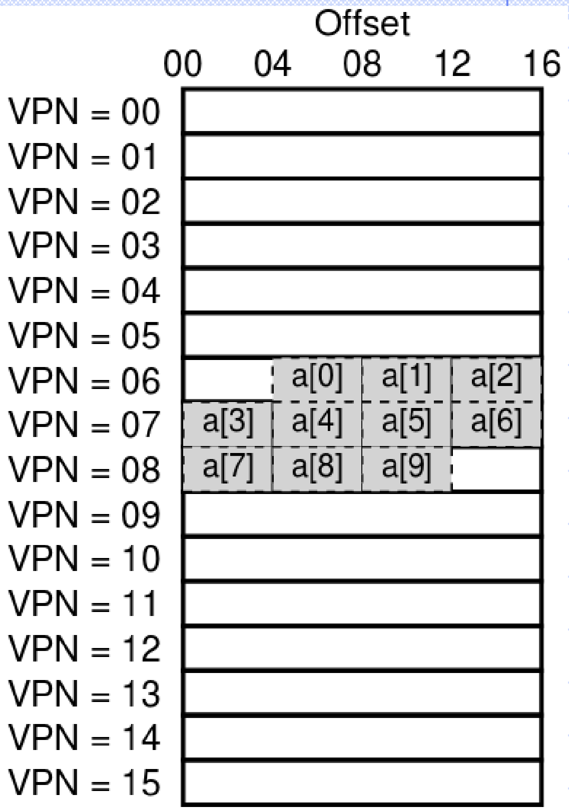
An array of 10 4-byte ints in memory starting at virtual address 100



Array Access in Loop

C program to access the array:

```
int a[10];
…
int sum = 0;
for (i = 0; i < 10; i++)
    sum += a[i];
```



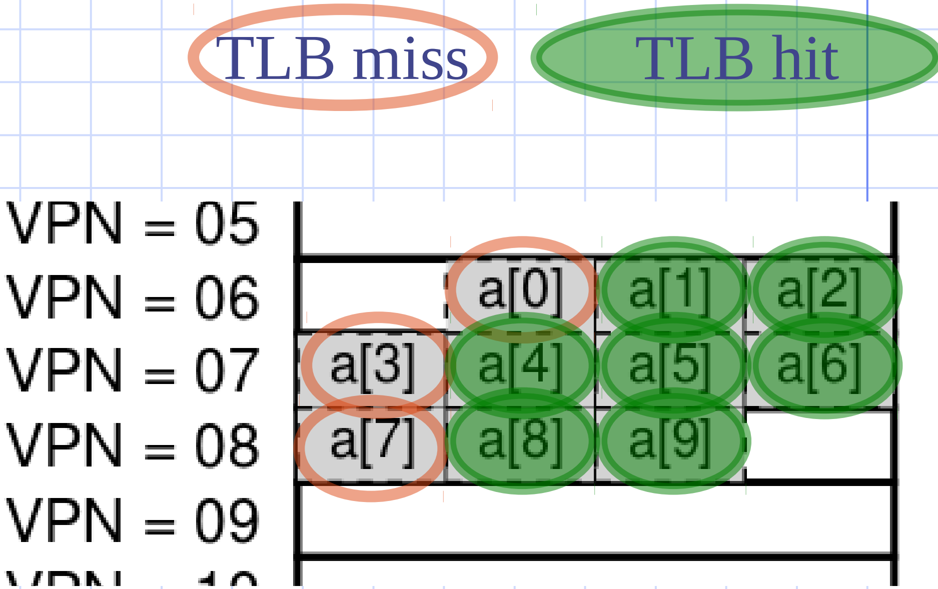
Array Access in Loop

C program to access the array:

```
int a[10];

…

int sum = 0;
for (i = 0; i < 10; i++)
    sum += a[i];
```



Arrays have spatial locality

Loops have temporal locality

Handling TLB misses

How should TLB misses be handled? OS or CPU?

Intel approach: Complex Instruction Set Computer – CISC

Hardware handles TLB miss – uses the Page Table Base Register to load entry from page table and update TLB

cf. Software-managed TLB: TLB miss → CPU raises an exception → Kernel trap handler updates TLB.

Reduced Instruction Set Computers – RISC (e.g. ARM)

TLB entries

A "fully-associative" TLB cache may have 32, 64, or 128 entries, each of which will have the form:

VPN	PFN	V P D A

Valid bit, Protection bits, Dirty bit, ASID

VPNs are specific to a process.

What happens during a context switch?

- TLB flush => remove all entries in TLB

- enable sharing TLBs across context switches with ASID – address space ID

Handling TLB entry replacement

TLBs are small – will fill up when a process has accessed a few pages. When it’s full, what do we do?

Get rid of entries? Which entry should be taken out so a new one can be brought in?

Goal should be: minimize miss rate or maximize hit rate

TLB entry replacement policy

Replacement policy goal should be to minimize miss rate or maximize hit rate.

One common approach: get rid of the <u>least recently used</u> (LRU) entry

Note: LRU != LIFO

LRU can be bad if conditions are just so.

Alternative: random replacement!

TLB entry example

- MIPS R4000 – RISC CPU
- 32-bit address space
- 4 KB pages → 19/20-bit VPN, 12 bit offset, 24-bit PFN
- Uses software-managed TLB approach.

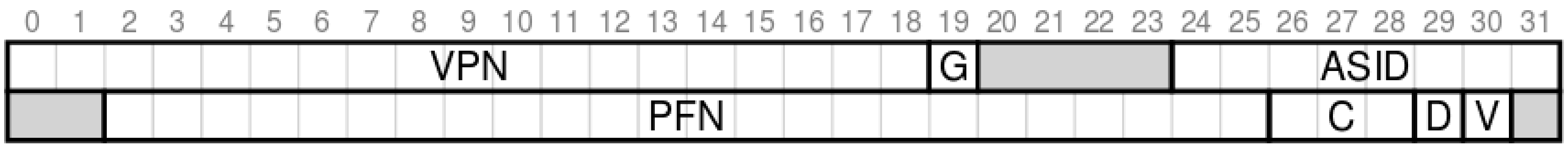


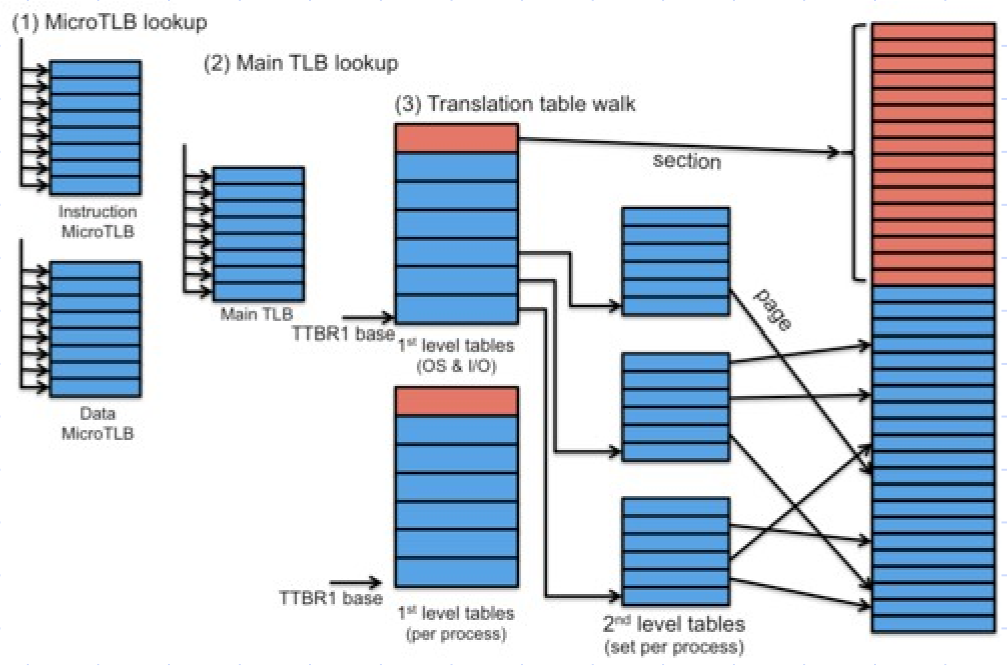
Figure 19.4: **A MIPS TLB Entry**

Modern TLBs

ARM – RISC CPU with 64-bit address space

Two levels of TLBs:

- Fast 32-entry fully-associative Micro TLBs for Data and Instruction
- Slower back-up Main TLB with 8 fully associative plus 64 set-associative (variable # of clock cycles for search)


(1) MicroTLB lookup
(2) Main TLB lookup
(3) Translation table walk
section
Instruction MicroTLB
Main TLB
page
TTBR1 base
1st level tables (OS & I/O)
Data MicroTLB
TTBR1 base
1st level tables (per process)
2nd level tables (set per process)