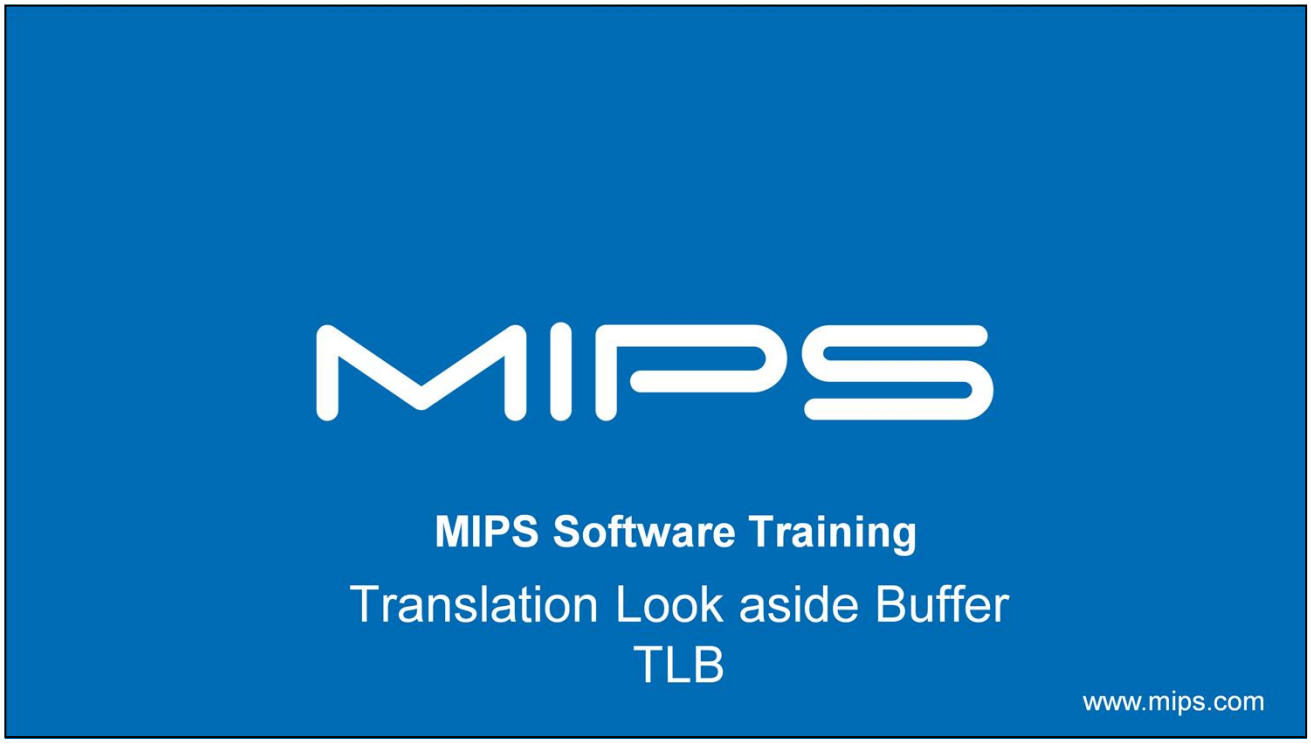
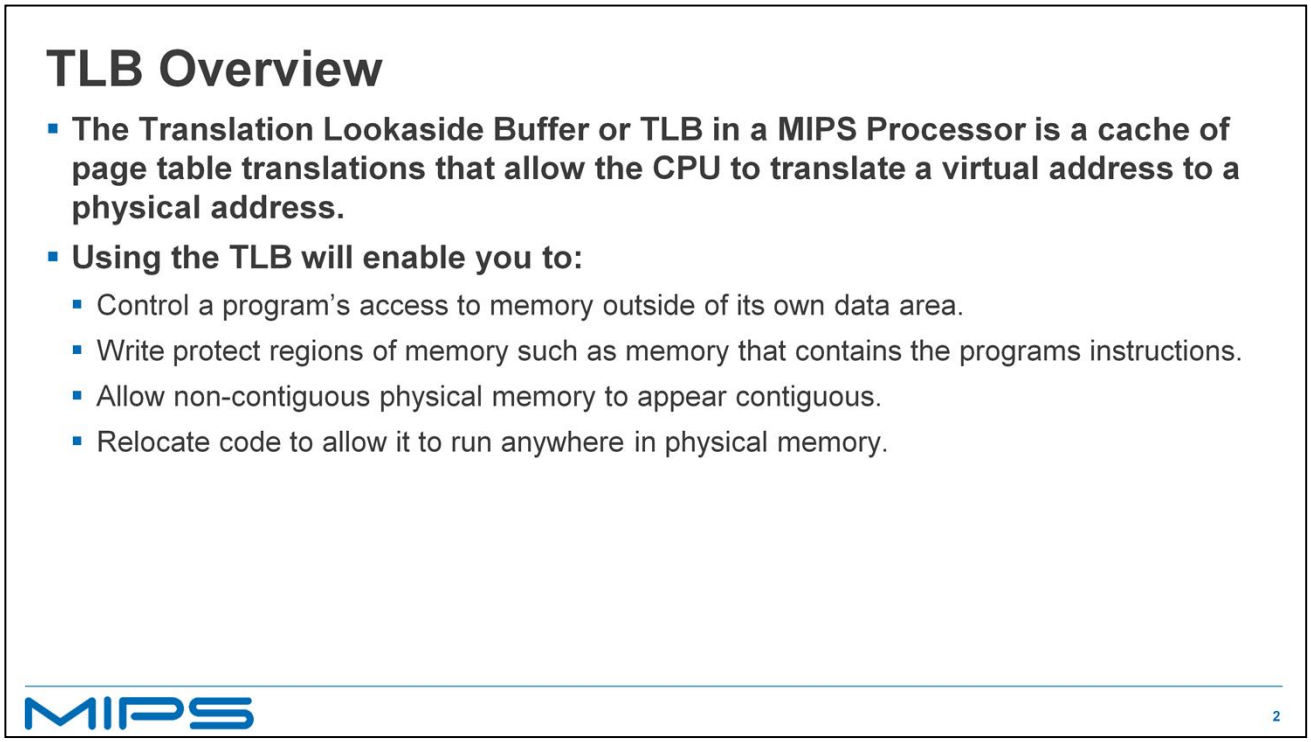


This training session will cover the Translation Look aside Buffer or TLB

I will cover:

What It is
How to initialize it
And How TLB exceptions are handled



First let me explain what we are dealing with:

In the MIPS architecture, operating systems such as Linux use a section of memory called kuseg or Kernel / User segment for user processes.

The addresses in this segment are called virtual address because they do not directly address physical memory.

Instead, these virtual address need to be mapped to physical address.

When a process requests access to memory, the TLB translates the virtual address to a physical address where that data is actually stored.

Using the TLB enables you to do several things:

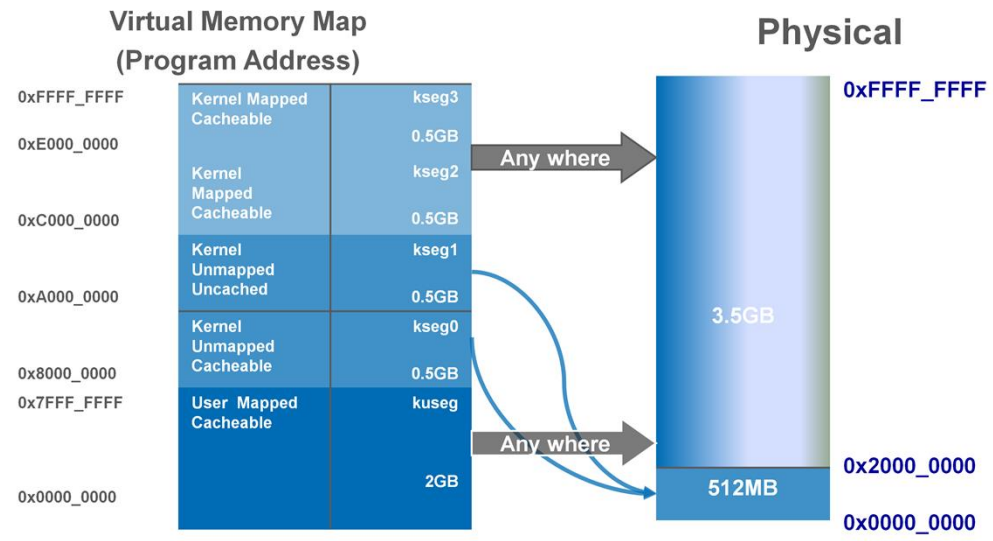
For the security minded the TLB allows the OS to control a process's memory accesses to its own data areas and not be able to read or write over another processes memory.

The TLB can also prevent a process from overwriting its own instructions.

It Allows non-contiguous physical memory to appear contiguous to a process.

A process can be linked to run at a specific address but be loaded and run from anywhere in physical memory.

Memory Map With TLB MIPS32



MIPS

3

There are different memory mappings depending on the state of the system and if you have a TLB or Fixed mapping MMU. I will cover all of the different mappings in this section. We going to start with the most common map, the map of Virtual memory after the boot process has completed.

The virtual memory map is broken down into segments. These segments differ in three characteristics:

Whether access to an address is mapped; that is, the address is virtual and is passed through the translation lookaside buffer (TLB) to translate the virtual address into a

physical address

Whether an address can be accessed when the CPU is operating in user mode or in kernel mode

Whether access to an address is cached; that is, looked up in the primary and secondary caches before it is sent to main memory

+ Here is a picture of the Virtual Memory map.

The first thing to note is all program address for both Kernel or user modes are virtual address. Also all address are in hexadecimal.

+On a MIPS32 Core the memory map covers a 4 gigabyte range of Physical memory.

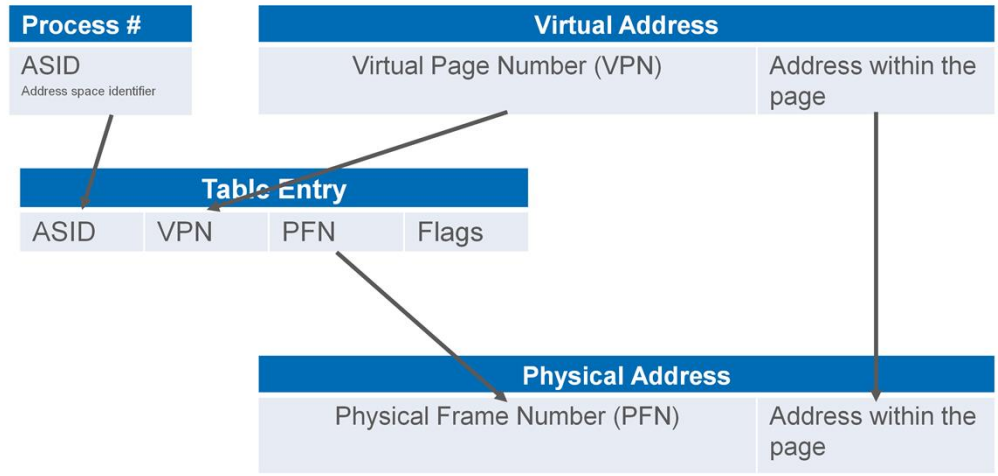
+The lower segment of memory, kuseg, covers a 2 gigabyte virtual address range starting at 0. It can be cached and mapped using the TLB to anywhere in physical memory. This segment can be accessed in Kernel or user mode.

+The next two sections kseg0 and kseg1 are designed to be use for the OS code and data. These segments can only be accessed in Kernel mode. If the processor is in User Mode any accesses to the kernel segment it will cause an address error exception. Both Kseg0 and Kseg1 sections are directly translated to the same lower 512 megabytes of physical memory. For example address 80 million and A0 million both are directly mapped to physical address 0. The difference between the two is, Kseg0 addresses are cacheable and can be used once the cache has been initialized. Kseg1 address are not cached and are used at boot time and for memory mapped I/O.

+The next two segments cover the virtual address range that runs contiguously from C0 million to FF FF FF FF hex and is accessible only in Kernel mode. The region is divided into two halves called kseg2 and kseg3, kseg2 was designed to be accessible to programs running in supervisor mode but supervisor mode is seldom, if ever, used. An OS such as Linux concatenates these segments together and refers to it as kernel high memory.

Simple TLB Overview

- **Simple Translation Diagram:**



MIPS

4

Now we will go through a simple TLB translation.

When an address is accessed by a process the TLB uses two things to find the correct internal table entry, The Virtual address itself and the process's Address space identifier or ASID.

The ASID will cause the look up to be valid for entries that match only the current process's entries.

The virtual address is broken into two pieces the Virtual Page number and address with in the page.

This breakdown allows the TLB to map large areas of memory with just one entry.

A typical page size is 4K.

This means that the lower 12 bits of the address will be the index within a page and the upper 20 bits will be the virtual page number or VPN.

The VPN is used to select the TLB entry with the correct Virtual Page Number.

The TLB will then use that entry's Physical Page number or PFN combined with the lower index bits of the virtual address to translate to the complete physical address.

If the TLB lookup fails - the CPU will signal an exception.

The exception handler is then responsible for looking up the virtual address translation in the process's page table and replacing an entry in the TLB with it.

Then the CPU will again execute the instruction that caused the exception and search the TLB once more for the correct entry.

TLB Entry Detail

Entry	Description	CP0 Register
ASID	Address Space (process) Identifier	EntryHi
VPN2	Even Virtual Page Number	EntryHi
R	memory region type (64 bit mode xuseg or xkseg)	EntryHi
EHINV	TLBWI invalidate enable	EntryHi
CMASK	Compressed Page mask to determine page size	Page Mask
PFN	Physical Page Frame # translation even address	EntryLo0/1
G	Global permissions (disable ASID matching)	EntryLo0/1
C	Coherency attribute of the page	EntryLo0/1
D	Dirty Bit (software controlled if cleared stores to page will cause an exception)	EntryLo0/1
V	Indicates that the TLB entry is valid	EntryLo0/1
RI	If set loads from page will cause an Read Inhibit exception	EntryLo0/1
XI	If set instruction fetches from page will cause an execute Inhibit exception	EntryLo0/1

MIPS

5

Now for a more detailed view of the actual TLB.

You cannot actually read or write whole TLB entries.

There are CoProcessor zero registers designed to hold the data to be read or written to the TLB.

These CoProcessor zero registers will be covered in more detail later in this section.

This table shows which register contains the piece of date that will go into making the TLB entry.

I will explain each entry the makes up the TLB.

ASID is a specific identifier that is assigned by the OS to a process.

The address space identifier (ASID) helps to reduce the frequency of TLB

flushing because it allows the mapping of up to 256 different address spaces simultaneously, without requiring the TLB to be cleared on a context switch. When the G bit is not set the TLB will concatenate the ASID and the virtual Page number to find the correct entry. The ASID is stored in the EntryHi register

VPN2 is the Virtual Page number to be looked up. Each entry contains two translations one for the even numbered page and one for the odd numbered page. The low order bit of the VPN is not used to find the entry but is used to determine which of the two translations - contained in the entry - is used. The VPN2 is stored in the EntryHi register

R Region bit tells you the region type of the access when an exception occurs.

EHINV If the TLBINV instruction is supported (indicated by the IE bit in the CP0 Config 4 register) and this bit is set the TLBWI instruction acts as a TLB invalidate operation, setting the hardware valid bit associated

with the TLB entry to the invalid state. NOTE: the PageMask and EntryLo0/EntryLo1 registers do not need to be valid when the TLBWI is executed.

CMASK determines the size of the page that is mapped by this entry. CMASK is stored in the Page Mask Register

PFN is the physical address translation for the page PFN is stored in the EntryLo0 and 1 registers

G is the Global Permissions bit. If this bit is set the address space identifier is ignored and all processes can use this entry.

This is used by an OS and for memory that will be shared by all processes.

For TLB read or write operations this bit is stored in the EntryLo0 and 1 registers.

The C bit is the cache ability attribute of the page. This will determine if the translated page address will be looked up in the cache or directly use to access memory. The C bit stored in the EntryLo0 and 1 registers

The D bit is the “Dirty” or Write-enable Bit. If this bit is set, stores to the page are permitted. If the bit is cleared, stores to the page cause a TLB Modified exception. An OS can use this bit to determine if a page has been written and thus is dirty. The OS insures the bit is cleared when it writes the entry to the

TLB. When the OS is trying to decide if the page needs to be written to a backing store it will check this bit and avoid an unnecessary write if the bit is cleared indicating the page is not dirty. Note if the OS uses this method then when a TLB Modified exception occurs and it sets the D bit in the TLB entry, it also needs to set the D bit in the process's page table entry in memory, other wise if the page gets overwritten by a tlbwr instruction the information will be lost. The D bit stored in the EntryLo0 and 1 registers.

The V bit will determine if the entry is valid and usable. If the V bit of both dual-entries is not set the dual-entry is not valid and will not be used. This bit can be used to make just one of a pair of pages valid. So if one entry is set but the other is not any accesses to that page causes a TLB Invalid exception. This bit should be cleared at boot time during the TLB initialization. The V bit stored in the EntryLo0 and 1 registers.

The RI bit is the read inhibit bit. If set loads from page will cause an Read Inhibit exception.

The XI bit is the Execute Inhibit bit. If set instruction fetches from page will cause an execute Inhibit exception.

EntryHi Register 32bit mode

Entry Hi Register					
31	13	12	8	7	0
VPN2		0		ASID	

- *VPN2, Virtual Page Number.*
- ASID Address Space Identifier
- Written by a TLBR instruction
- Read by a TLBW or TLBWR

Bit assignments assume a 4k page size

MIPS

6

I'll now go into details on the registers involved with the TLB functionality starting with the EntryHi register.

The EntryHi register is Register 10 of Coprocessor zero .

This slide assumes a 4K page size.

The register contains two elements:

VPN2 and

The ASID

The *ASID* field is written by software usually during a context switch.

EntryHi is filled in when you are reading a TLB entry using the TLB Read instruction.

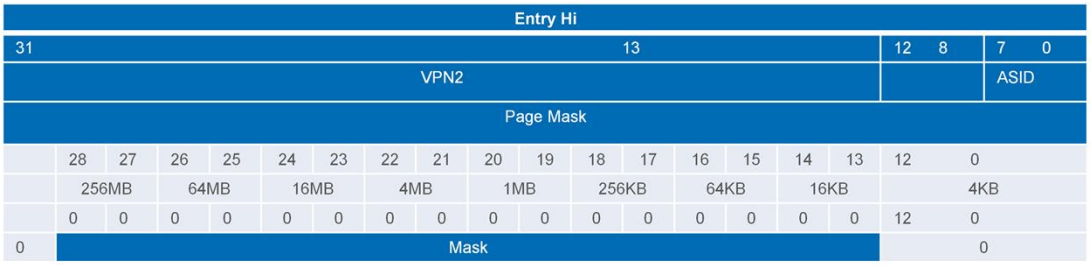
Because the *ASID* field is overwritten by a TLBR instruction, software must

save and restore the value of *ASID* around use of the TLBR.

EntryHi is read into the TLB entry using the TLB Write or TLB Write Random instructions.

Page Mask Register 32 bit mode

- **The Mask field is aligned within the register to the virtual part of the address bits.**



- Written by a TLBR instruction
- Read by a TLBW or TLBWR

MIPS 7

I have overlaid the Entry Hi register with the page mask register so you can see the effect the page Mask register has on the VPN2 address.

The Page Mask register allows TLB entries to map pages larger than 4K.

The 1 bits in the page mask have the effect of causing the corresponding bit of the virtual address to be ignored when matching the TLB entry.

Instead the bit is carried unchanged to the resulting physical address, effectively increasing the page size by a power of 4 for each mask bit up to 256MBs.

Bits must be filled in right to left and always in pairs

NOTE the odd/even page selection bit also moves 2 bits with each mask position.

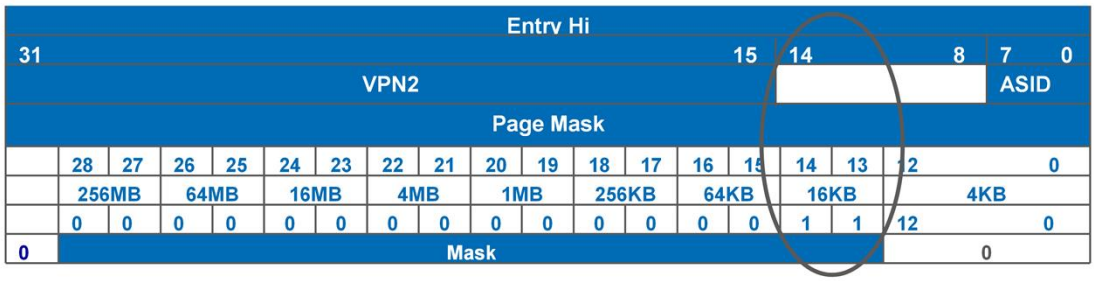
The Page Mask Register is filled in when you are reading a TLB entry using the TLB Read instruction.

The Page Mask Register is read into the TLB entry using the TLB Write or TLB Write Random instructions

This slide shows the default values for a 4K Page

Page Mask Register for 16KB Page 32 bit mode

- Setting bits 13 and 14 change the page size to 16KB.



This slide, shows that setting bits 13 and 14 will cause the page size to increase to 16K

Entry Lo 0 and 1 Registers 32 bit mode

Entry Lo 0/1						
31 26	25 6	5 3	2	1	0
0	PFN	C	D	V	G

- **EntryLo 0 and 1 contains** (assuming 4K page):
  - PFN Upper 20 bits of the physical page number
  - C cache control bits
  - D Read only bit
  - V Valid bit
  - G Global Bit
- **Written by a TLBR instruction**
- **Read by a TLBW or TLBWR**

MIPS

9

EntryLo 0 and 1 contains the translation information.

It contains the Physical Frame number
The cache control bits
The Dirty bit or write control
The valid bit
And the Global bit

Index Register

- **Index – This register is used write or read a particular TLB index (*tlbwi/tlbr*) or to return an index from a TLB probe (tlbp) for a virtual address. If bit 31 is set TLB probe did not find an entry.**

Index Register (CP0 Register 0, Select 0)			
31		9	0
P	0		Index

  - **If the P bit is set the previous TLBProbe (TLBP) instruction failed to find a match in the TLB.**

MIPS

10

The index register is used by the TLB Write Index or TLB Read instructions to determine the TLB entry to be written or read.

It is also used by the TLB probe instruction to determine if an entry exists in the TLB.

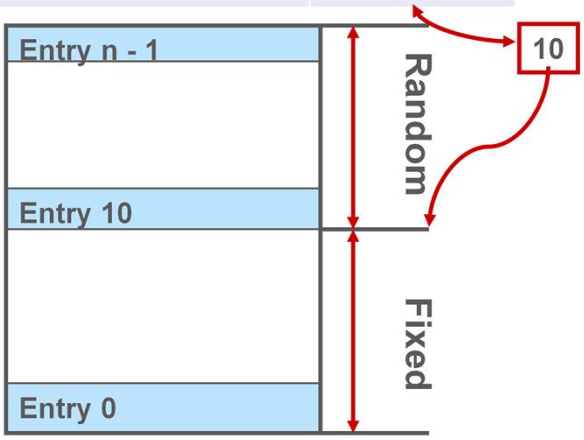
If the probe was successful bit 31 of this register will be cleared.

Wired Register

release <= 5

Wired Register (CP0 Register 6, Select 0)			
31	6	5	0
0		Wired	

- **Wired – determines number of entries that cannot be replaced by TLB Write Random (*tlbwr*).**



MIPS

11

The Wired register is a read/write register that specifies the number of the wired entries in the TLB

For example if you wrote 10 to the Wired Register TLB entries 0 through 9 would be unchangeable by a TLB Write Random instruction

To write to TLB entries that are in the wired range you must use the TLB Write Index instruction.

Random Register

Random Register (CP0 Register 1, Select 0)		
	5	0
		Random

- **Random Register is a read only used by TLB Write Random (*tlbwr*) instruction to select a entry to write (or over write).**
  - The upper bound is set by the total number of TLB entries minus 1.

MIPS

12

The Random register is read only register that is used to determine the TLB index when you use the TLB Write Random instruction. It is that is decremented by one almost every clock, wrapping after the value in the Wired register is reached.

To enhance the level of randomness and reduce the possibility of a live lock condition, a Linear Feedback Shift Register is used to introduce a pseudo-random perturbation into the decrement.

The processor initializes the Random register to the number TLB entries -1 on a Reset exception and when the wired register is written.

This register is only valid with the TLB.

TLB Initialization

- **The following code initializes the TLB must be initialized before access to any mapped address.**
  - CP0 Config1 register (16 select 1) contains the size of the TLB in the MMU size field

Config1 Register (CP0 Register 16, Select 1)

31	30 25	24												0
M	MMU Size	IS	IL	IA	DS	DL	DA	C2	MD	PC	WR	CA	EP	FP

```
# Read the CP0 Config1 register
mfc0        $10,    $16,    1

# Extract MMU Size bits 25, 26, 27, 28, 29, 30
ext         $11,    $10,    25,     6       # Highest number TLB entry
```

MIPS

13

I am going to show you how to initialize the TLB. You should initialize the TLB as part of the boot process before you enable it.

First we need to get the size of the TLB.

The TLB size along with other information is stored in the read only Coprocessor 0 register called config1.

You can read this register by using the move from Coprocessor 0 instruction

Here I am going to move into General Purpose Register 10 the Config1 register which is Coprocessor zero register 16 select 1

Next I will extract the MMU size field by using the extract instruction

From general purpose register 10 where I just stored the config1 register

To general purpose register 11.

Starting from bit 25

For 6 bits

The size value extracted is the total number of TLB entries -1 since the first entry is entry 0.

TLB Initialization

```
# clear all entry registers
mtc0  $0, $2   # C0_ENTRYLO0
mtc0  $0, $3   # C0_ENTRYLO1
mtc0  $0, $5   # C0_PAGEMASK  (4K)
mtc0  $0, $6   # C0_WIRED
mtc0  $0, $10 # C0_ENTRYHI
```


Now to clear all entry registers so we will initialize their TLB entire fields to 0

To do this we use the same move to Coprocessor zero instruction using the general purpose register 0 which always contains a 0 value and move to the corresponding Coprocessor 0 register

TLB Initialization Loop

```
1       # Loop start
        # use $11 as index (decremented each loop)
        mtc0    $11,    $0              # C0_INDEX
        ehb                             # Clear hazard barrier
        tlbwi                           # Write indexed entry
        bne     $11,    $0,     1b      # Loop until index = 0
        add     $11,    -1              # subtract 1 from index
```

MIPS

15

We will now use a loop to initialize each TLB entry.

The 1 in the left column is the label of the start of the loop and the point we will loop back to.

Remember we stored the highest numbered TLB entry in general purpose register 11.

We will use it here to decrement through the TLB entries from the highest to the lowest and use it to program the TLB entry index.

I use the move to Coprocessor 0 instruction to copy the contents of General purpose register 11 to Co Processor 0 register which is the index register.

You need to make sure all the writes have been completed to coprocessor 0 before we write the TLB entry.

I do this by using the ehb instruction.

Now I use the TLB Write Index Instruction to write the TLB entry

Next is the branch instruction

Here I compare the TLB index value that is in the general purpose register 11

with 0

and if they are not equal the code branches back to label 1

The last instruction is in the branch delay slot and will always be executed

I use the add instruction to increment the index value in the general purpose register 11 by a -1

TLB Using Page Tables

- Systems that need to map more memory pages than will fit into the TLB can use an in-memory Page Table.
- There is a scheme in the MIPS architecture that makes the Page Table look up and writing the correct entries into the TLB easy and efficient.

MIPS

16

For many systems the amount of entries in the TLB will not be enough to cover the range of address translations needed.

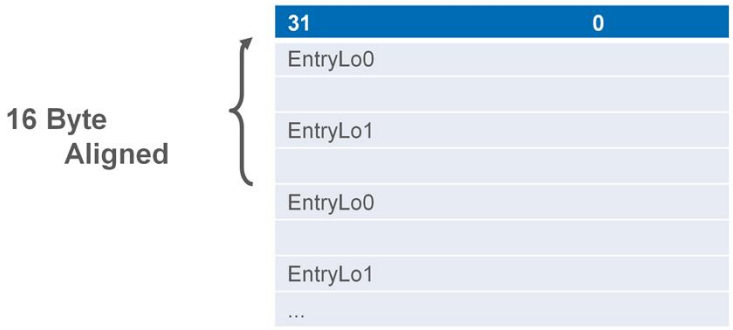
In this case the TLB will just serve similar to a data cache providing most translation very quickly and have to call on help to fill in translations it doesn't currently have.

Such systems will have in memory page tables for the OS and each process that is currently running.

If the TLB does not find a translation in its entries it will cause a TLB exception. MIPS has a way of making a TLB exception easy and fast.

Page Table layout

- **The page table consists of a linear array of structures that contains the paired values of the EntryLo 0 and EntryLo 1 registers, aligned to a 16 byte boundary.**



MIPS

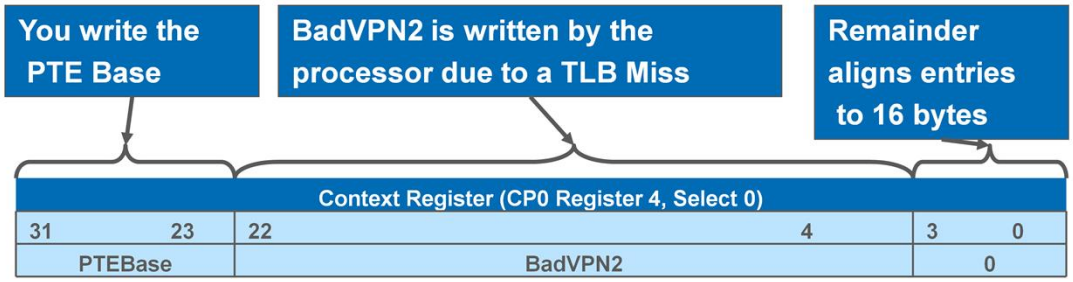
17

In the MIPS scheme the in memory page table is a linear array of EntryLo 0 and 1 entries. There is a 8 byte alignment between the values in each pair to make it compatible with a MIPS64 processor that has a 64 bit address space.

This means the each pair is 16 byte aligned.

Context Register

- Context Register becomes the index into a Page Table aligned at a 4MB boundary (0x400000)



MIPS

18

The Context register is organized in such a way that the operating system can directly reference an 8-byte page table entry or PTE in memory. This index will point to the EntryLO 0 entry of the EntryLO 0 and 1 pair in the page table.

On a context switch you will need to write the base address of the process's Page Table into the Context Register.

The Page Table Base must be aligned to a 4 megabyte boundary which means only the top 9 most significant are needed.

When a TLB exception happens the processor fills in the virtual page number that it needs translated so the register will contain the index to a 16 byte aligned EntryLO 0 and 1 pair.

TLB Exception Handler

```
        .set noreorder
TLBmiss32:
        mfc0    k1,     $4      # Get Context register
        lw      k0,     0(k1)   # Load Entrylo0 into K0
        lw      k1,     8(k1)   # Load Entrylo1 into K1
        mtc0    k0,     $2      # Move k0 to CP0 EntryLo0
        mtc0    k1,     $3      # Move k1 to CP0 EntryLo1
        ehb                     # Clear hazard barrier
        tlbwr                   # Write it to random TLB entry
        eret                    # Return from exception
        .set reorder
```

Remember: On a TLB refill exception, **EntryHi** is set with the VPN and ASID that could not be translated so you don't need to fill it in.

MIPS

19

Here is an example of a TLB exception handler.

First you need to use an assembler directive to insure that the assembler does not reorder the code when it tries to optimize. You do this by the directive .set noreorder.

Next we label the function TLBmiss32

We now copy the context register which holds the page table index into a kernel register.

As a side note: Kernel registers are set aside by the MIPS API for use by the kernel. These are usually used for exceptions because they don't have to be saved. There are two registers reserved as Kernel general purpose registers $26 designated here as k0 and $27 designated here as k1.

The copy is done using the move from coprocessor 0 instruction to kernel register k1 from Coprocessor register 4 the Context register.

Now we use the index to load the page table's EntryLo 0 value into the other Kernel register.

We us the Load word instruction to load kernel register k0 using the index stored in kernel register k1.

Next we will use the index to load the page table's EntryLo1 value into a kernel register

We us the Load word instruction to load kernel register k1 using the index stored in kernel register k1 with an address offset of 8.

Now that we have the values to write to the TLB we need to move them into the appropriate Coprocessor 0 registers.

We use the move to coprocessor 0 instruction to move the EntryLo 0 value stored in the kernel register k0 to coprocessor 0 – register 2 which is the EntryLo 0 register.

Then we again use the move to coprocessor 0 instruction to move the EntryLo1 value stored in the kernel register k1 to the coprocessor register 3 which is the EntryLo 1 register.

Now to insure all the values are written properly we need to clear the instruction hazard barrier by using the ehb instruction.

We can now write the new entry to the TLB using the TLB Write Random instruction.

Last we use the return from exception instruction to go back to normal processing

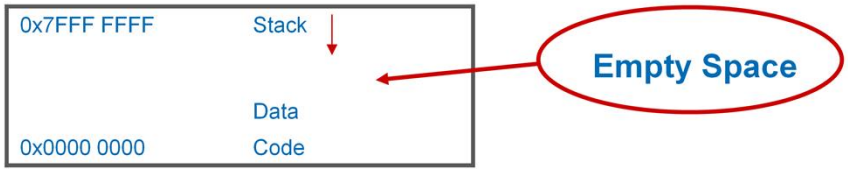
.set reorder tells the assembler that it can continue optimization of the code from this point.

Since the TLB refill exception already filled in the virtual address and the address space identifier with the address that missed in the TLB we only need 8

instructions to do the refill.

Using Page Tables in Virtual Memory

- **To conserve space in memory each process' page table should be located in virtual memory in the Kernel-mapped KSEG2 region.**
  - Most processes are filled at the bottom of their address space with code and data and at the top with their stack with a lot on nothing in the middle which will never be referenced.



MIPS

20

Page tables are usually only as big as they need to be to cover memory that the program actually uses.

Most processes are filled up from the bottom with code and data and down from the top with their stack. There is a big hole in the middle that's empty and will never be referenced.

To make the system think there are no holes and still be able to index into the table as if it covered the whole 4 gigabytes of address space, the process's page tables use virtual memory mapped in the OS page table.

As part of a context switch, your OS needs to write the process's ASID value to the EntryHi register which and the process's page table base to the context register..

TBL Miss During a TLB Refill

- **Refill within a Refill**
  - A TLB refill exception handler may itself get a TLB refill exception because the kseg2 mapping for the page table isn't in the TLB.
  - A nested TLB refill exception does not use the TLB exception vector. Instead the processor uses the general exception vector.
    - When a nested exception happens the EPC (restart location) does not change. Then when the eret is done processing returns to the instruction that caused the TLB exception. This will cause a TLB exception again but this time the virtual address to the process's page table will be resolved.
    - See next slide for flow chart.

MIPS

21

Since the process's page table is in OS virtual memory - access to it can also cause a TLB miss exception.

To distinguish this nested TLB exception from a primary TLB exception the CPU uses the General exception vector instead of the TLB exception vector.

When the General exception vector is used the OS knows to use its own page table to fill in the TLB entry.

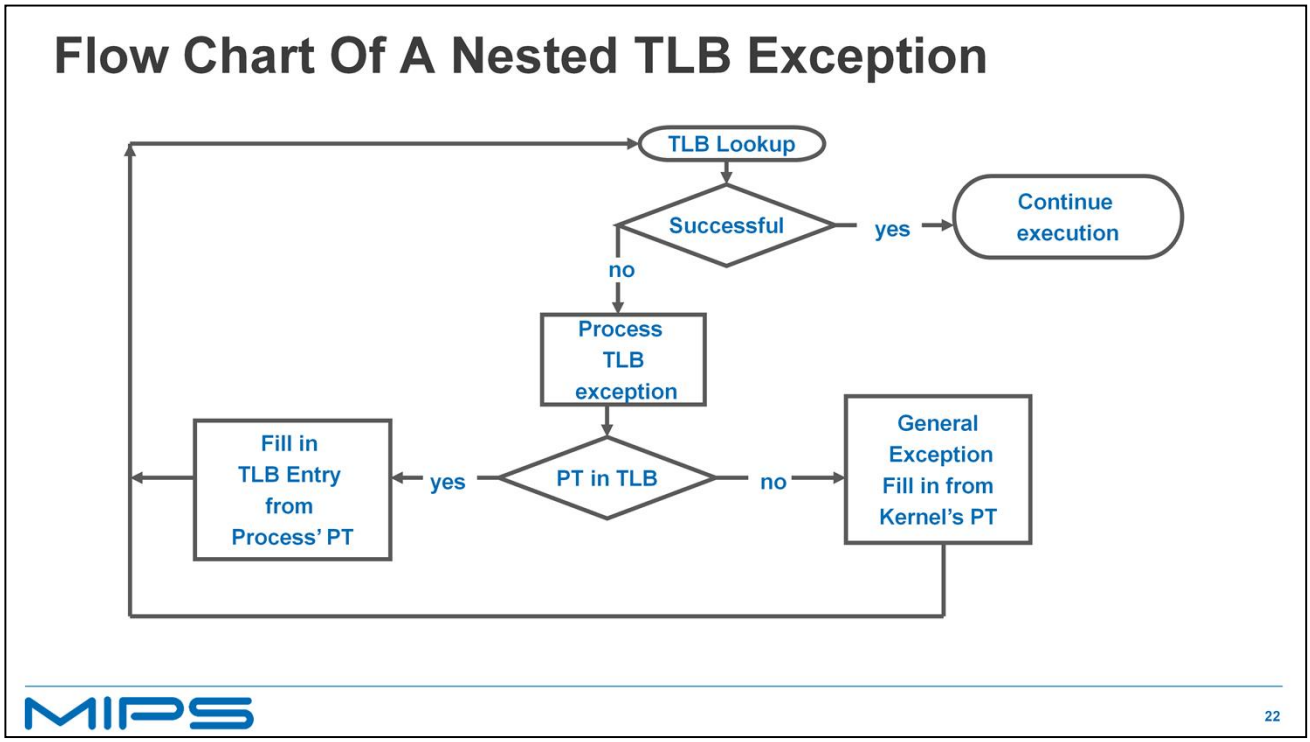
The OS page table is located in kseg0 which is not TLB mapped so no TLB miss can happen on an access to it.

The CPU knows the second TLB exception should be nested because the EXL bit in the status register was set at the time of the exception.

The EXL bit being set triggers the CPU to not change the Error PC register. The EPC is the Program Counter of the instruction that got the exception and where execution will start when exception processing has completed.

Since this does not get changed and it is still pointing to the instruction that caused the first TLB exception when the nested exception returns the CPU will get another TLB exception.

This time the process's page table entry will be mapped into the TLB so that it can be used to fill the TLB entry.



Let's go through a flow chart of nested TLB miss exception handling.

First the CPU tries to do a TLB Lookup and it checks to see if it was successful

There is a TLB miss so the processor generates a TLB exception.

The exception handler tries to look up the entry in the process's page table and we get another TLB miss

This time we start execution from the General exception handler. This handler will fill the process's page table miss from the OS Page table.

The CPU returns back to user mode executing the process's code.

The lookup is tried again.

And fails.

Another TLB exception is generated but this time the entry is found in the process’s page table.

The TLB is filled from the process’s page table entry.

And returns to back to user mode.

The TLB look up happens again

And this time is successful.