This is the cache section of the MIPS software training course

Cache Overview

- **Covered in this section**
  - Cache dimensions
  - Cache Lookup
  - Cache policy
  - Non-blocking loads
  - Cache initialization
  - Cache management
  - Virtual aliasing
  - Prefetch
  - Cache exceptions

MIPS

2

In this Cache section I will cover

+ The Dimensions of the cache and what make up the size of the cache
+ how things are looked up in the cache
+ what types of cache policy you can set
+ What non-blocking loads are
+ The initialization of the cache
+ cache management such as cache flushing
+ what virtual aliasing is and how to avoid it
+ what the prefetch instruction does
+ cache exceptions

Cache Overview

- **Covered in this section**
  - Cache dimensions
  - Cache Lookup
  - Cache policy
  - Non-blocking loads
  - Cache initialization

MIPS

3

In this Cache section I will cover

+ The Dimensions of the cache and what make up the size of the cache
+ how things are looked up in the cache
+ what types of cache policy you can set
+ What non-blocking loads are
+ The initialization of the cache

Cache Dimensions

- **Independent L1 Data and Instruction**
- **Four-way Set Associative**
- **Line Size: 32 bytes Lines Per Way:**
  - 64, 128, 256 or 512
  - Corresponding Cache size 8K, 16K, 32K or 64K
  - Equal to the number of Sets per way

Cache Size = associativity (4) * line size (32) * lines per way

MIPS

4

+ MIPS Cores have separate Instruction and data caches so that an instruction can be read and a load

or store done simultaneously.

+ Except for the 4KE all caches are four way Set associative. For the 4KE there is a choice from a direct mapped cache to a four way set associative cache. More ways of associating cache line with in a page to physical memory usually means less cache threshing.

+ Cache line size is fixed at 16 bytes for the 4KE and 32 bytes for the rest of the cores.

+ Lines per way will determine the cache size. These are a power of 2 starting from 64 up to 512.

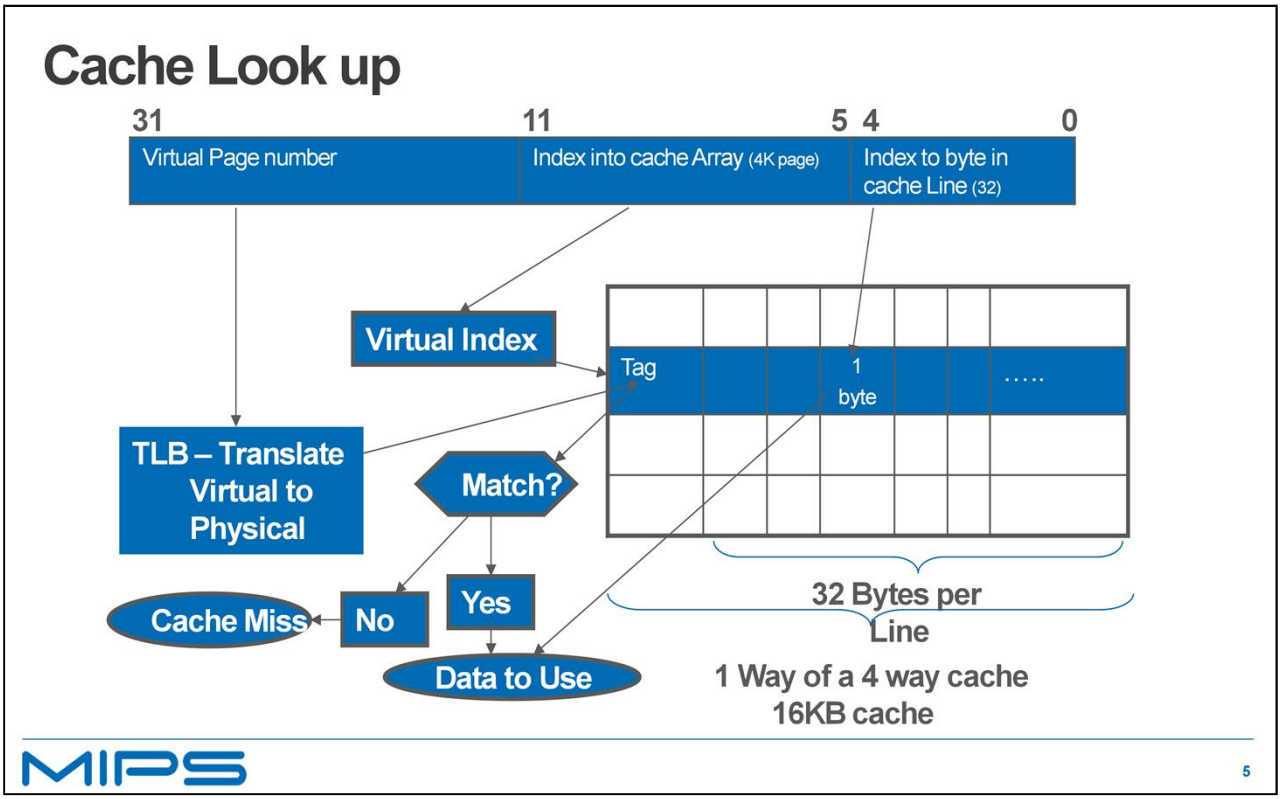
+ This corresponds to a cache size of eight to 64K.

+ Sets per way is a super set of a line where a cache line is the actual data and the whole set includes the cache tag and status bits.

+ The size of the cache is equal to the number of way times the line

size times the number of lines per way.

All cache choices regarding cache size and number of ways are options configured when the chip is built and are not changeable at run time.



This example will give you background information on how the cache works. It will show how data is retrieved from the cache. The example assumes a 4K page size and a cache with 4 ways for a total cache size of 16K.

Each virtual program address can be broken down into three parts

+ The virtual page number, the index into the cache array and the byte index into the cache line.

Caches are virtually indexed and physically tagged to allow them to be accessed in the same cycle as the address is translation.

+ This means that while the MMU is using the virtual page number to get the physical page number

+ The cache controller is using the index part to locate where in each way of the cache line might be stored.

+ So once the translation is made the controller can then compare

translated physical page address to the physical tags in each way to see if there is a match.

+ If there is a match we have a cache hit and the fetch, store or load is completed.

+ If not it’s a miss and the processor will call on the bus interface unit to get the cache line from memory.

Cache Policy - Cache Coherency Attribute, CCA

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

- For TBL Systems
  - EntryLo 0 and 1 contains cache policy for the cached page

Config Register														
31	30 28	27 25	24	23	22	21	18	16	15	14 13	12 10	9 7	6 3	2 0
M	K23	KU	ISP	DSP	UDI	SB	MM	BM	BE	AT	AR	MT		K0

- For KSEG0 The CCA is set through the Config[K0] field

MIPS

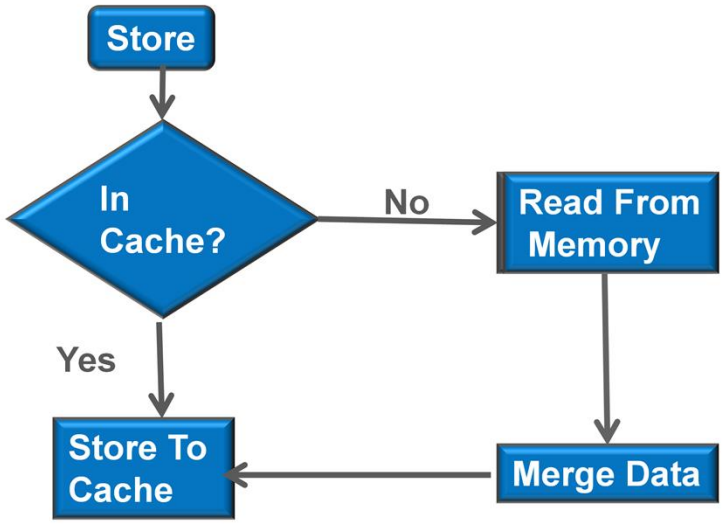
I am going to go into the Cache Policy

For regions mapped by the TLB (KUSEG KSEG/2/3) the CCA is determined by the TLB entry for the page that contains the memory access. This is set up by the C field in the Entry Lo registers when the TLB entry is written for each page. The CPU uses these bits to figure out what to do. The page can be marked un-cacheable which would cause the CPU to go directly to memory for the load or store or Instruction Fetch. If the page is cacheable these bits will set the cache policy which I will cover next. I am not going to go into TLB entries because they are covered in the TLB section of this course.

For KSEG0 the CCA is determine by the K0 field in the Config register.

Cache Policy

- Write-Back



MIPS

7

Another cache police you can set is called write back. For a write back cache

+ On a store

+ The cache is Searched to see if the target address is cache resident.

+ If it is resident, the cache contents are updated,

+ If the cache lookup misses the cache line is Read from memory

+ then the new data being stored is merged into the line and store into the cache

Important points here are:

Cache lines are always filed from memory on a cache miss

Main memory is never updated on a store

If there are no way entries free when a new line needs to be put into the cache, a line needs to be evicted from the cache to make room for the new line. The line to be evicted is selected on a **L**east **R**ecently **U**sed basis. If the evicted line is dirty, meaning it contains data not in memory, then the line is written back to memory.

Outstanding Data Cache Misses

- **Non-blocking D-cache misses**
  - In the case where instructions later in the instruction stream are not dependent on the load data, the core can continue executing instructions while the miss is being processed

MIPS

8

In most cases the data caches are non-blocking allowing for the instruction stream to continue executing until a instruction that is dependent on the missed data is executed. Then the processor will stall until that data word is loaded from memory.

+ The 4KE core is an exception to this. The data cache on the 4KE will block until the data word is loaded from memory. It will not continue to execute non data dependent instructions.

Some cores can have more than one miss outstanding at a time. The 24K 34K and 1004K can be configured at chip build time for four or eight outstanding data cache miss. The 74K is not configurable and has four.

Cache Instruction

- **Types of Cache Instruction operations**
  - Index operations
    - invalidate, write-back invalidate, load tag, store tag, store data, load way-select, store way-select
  - Hit operations address is specified
    - invalidate, hit write-back invalidate, hit write-back,
      - Address operations
    - fill, fetch & lock

Cache initialization

- **Before use, the cache must be initialized to a known state, all cache entries must be invalidated**
  - The following code example will initialize the cache.
    - Find the total number of cache sets
    - Loop through the cache sets and use the cache instruction it invalidate each cache set.

Cache initialization

Configuration Register 1, CP0#16-Sel1

Config1 Register															
31	30 25	24 22	21 19	18 16	15 13	12 10	9 7	6							0
M	MMU Size	IS	IL	IA	DS	DL	DA	C2	MD	PC	WR	CA	EP	FP	

IS	Icache sets per way (cache lines)
	= 0 64, = 1 128, = 2 256, = 3 512, = 4 1024, = 5 2048, = 6 4096
IL	Icache Line Size
	= 0 No ICache present; = 1 4 bytes = 2 8 bytes; … = 6 128 bytes
IA	Icache Associativity; = 0 Direct mapped
	= 1-7 2-way-- 8-way respectively
DS	Dcache sets per way (cache lines) (same encoding as IS field)
DL	Dcache line size (same encoding as IL field)
DA	Dcache Associativity (same encoding as IA field)

Cache initialization

- **What the code does:**
  - Finds out total number of set in the cache. This will give us the number of loops to go through to initialize the tags in each set of the cache.

Total set in cache = Ways (4) * Set per way (128)

MIPS

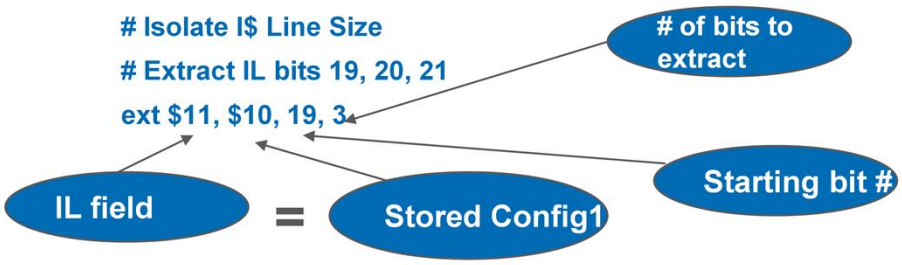
The code will use configuration register one to compute the total number of sets in the cache to loop through and initialize each sets cache tags.

Cache initialization

- Get the dimensions of the ICache

```
# Get ICache line size from the Config1 CP0 register
mfc0    $10, $16, 1

# Isolate I$ Line Size
# Extract IL bits 19, 20, 21
ext $11, $10, 19, 3
```




```
# If line size is 0 we have no cache, Skip ahead if no I$
 beq    $11, $0, 10f
 nop
```

MIPS

13

The code uses the move from coprocessor 0 instruction to get the value of the config1 register into the general purpose Ten

+ the extract instruction is used to extract the Instruction cache line size. It uses the config one register value that was saved in general purpose register ten starting at bit nineteen and extracts 3 bits to the least significant bits of register eleven.

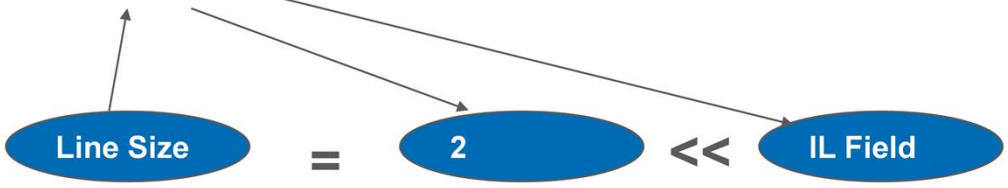
+ it is then tested to see if it is 0 and if it is it means there is no Instruction cache so it will branch ahead and not initialize it.

Cache initialization

- Compute the actual Line size

```
# 2 IL
# For example a 0x4 in $11 = 32 byte line size
# 0x2 << 4 = 0x20 (32 decimal)

li    $14, 2
sllv  $11, $14, $11                # I$ line size in $11
```



MIPS

Now the code decodes the line size to get the actual number of bytes in a line.

It does this by shifting 2 to the left by the encoded line size value.

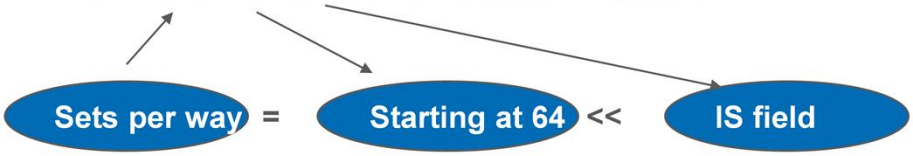
Cache initialization

- Get the sets per way (IS)

```
# Isolate I$ sets (number of lines, and tags per way)
# Extract IS bits 22, 23, 24
ext   $12, $10, 22, 3

# Compute sets per way (IS = 0 = 64, 1 = 128 .....)
# power of 2 starting with 64 or 64 x (2 to the IS power)

li      $14, 64
sllv    $12, $14, $12        # I$ Sets per way in $12
```



MIPS

15

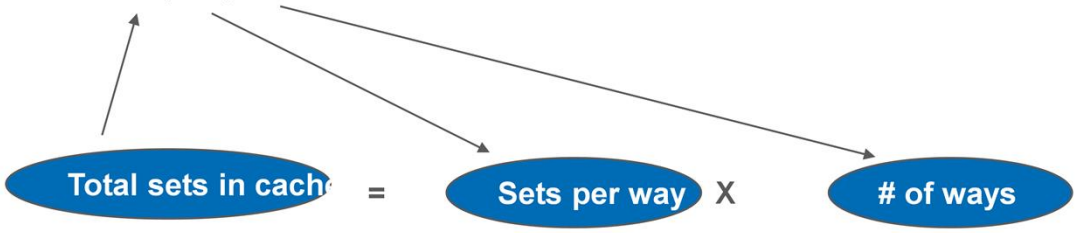
Now the code extracts the number of sets per way from the value read from the config one register that was stored in general purpose register ten it does this by using the extract instruction.

+ The extracted value is converted to the actual number of set per way by shifting sixty four left by the extracted value.

Cache initialization

- Get the number of ways (IA)

```
# Isolate the IA bits:
# Extract IA bits 16, 17, 18
ext     $13, $10, 16, 3
add     $13, 1              # 0 = Direct mapped (1 way)
# Note: $12 contains number of sets per way
mul     $12, $12, $13       # Total number of sets in cache
```



MIPS

16

The number of ways is extracted from the value config one register value using the extract instruction.

+ The code then adds one to the value to get the actual number of ways.

+ Now the sets per way are multiplied by the number of ways to get the total number of sets in the cache which will become the number of loops to be preformed to initialize the cache.

Cache initialization

 # Set $14 to a virtual address

- Address will be translated to a physical address.
- Since the address 0x8000 0000 is in kseg0 the cpu will ignore the top bit so virtual 0x8000 0000 will become physical address 0x0000 0000.
- Since the cache is physically indexed, the first time through the loop the cache instruction will write the tag to way 0 index line 0.

 # The lui instruction will load 0x8000 into the upper 16 bits

 # and clear the lower 16 bits of the register

lui $14, 0x8000

MIPS

17

The general purpose register fourteen will be used as the index into the cache.

+ The trick is this needs to be a virtual address.

+ The virtual address that is the index into set 0 of the cache is eight million hex. This works out because this virtual address is in KSEG zero. If you recall kseg0 is in the virtual range of eight million hex up to A million hex minus one which directly maps to physical address zero through two million hex minus one. So the cpu translates the virtual address eight million hex to physical address zero.

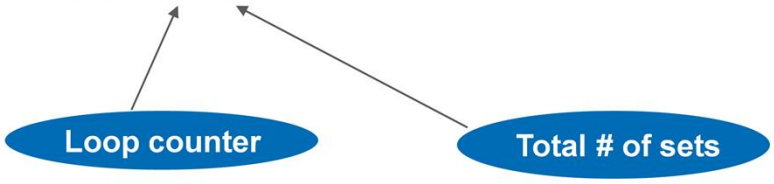
+ Physical address zero will always index to set zero of the cache.

+ The code loads register fourteen by using the load upper immediate instruction will eight thousand hex. This instruction sets the upper bits to eight thousand hex and clears the lower 16 bits.

Cache initialization

```
# Clear TagLo (28)/TagHi registers (29) (invalidates entry)
# written to the cache tag by the cache instruction
mtc0    $0, $28   # TagLo
mtc0    $0, $29   # TagHi 74K only (loads 64 bits at a time)

# Set up $15 as loop counter
move    $15, $12
```



MIPS

18

Clearing the tag registers does two important things it sets the Physical Tag address called PTagLo to 0 this insures the upper physical address bits are zeroed out. It also clears the valid bit for the set which insures that the set is free and may be filled as needed.

+ the code uses the Move to Co processor zero instruction to move the general purpose register zero, which always contains a zero, to the tag registers. You need to consult your programmers manual for the core you are using to see if the TagHi register also needs to be cleared.

+ the code is almost ready to start the loop through the cache. This move instruction puts

+ the total number of set that the code computed

+ into register 15 which will be decremented each time through the loop.

Cache initialization - Cache Instruction

- Cache Instruction Format:

```
CACHE  op, offset(base)
```

- Cache Op encoding
  - Bits 0 and 1 define which cache the operation will be performed on, which in this case is 00 or Primary Instruction cache
  - Bits 2,3 and 4 define cache operation, which in this case is 010 or Index Store Tag
  - Op field = 0x8 (010 00)

MIPS

19

Here is the format for the cache instruction.

+ The op field is 5 bits it contains cache type and the cache the operation will be performed.

+ Bits 0 and 1 encode the cache type: 00 for the Level one instruction cache, 01 the level one data cache, 10 the level 3 cache and 11 for the level 2 cache. Our MIPS core do not support the level 3 cache and the level 1 and 2 caches are optional.

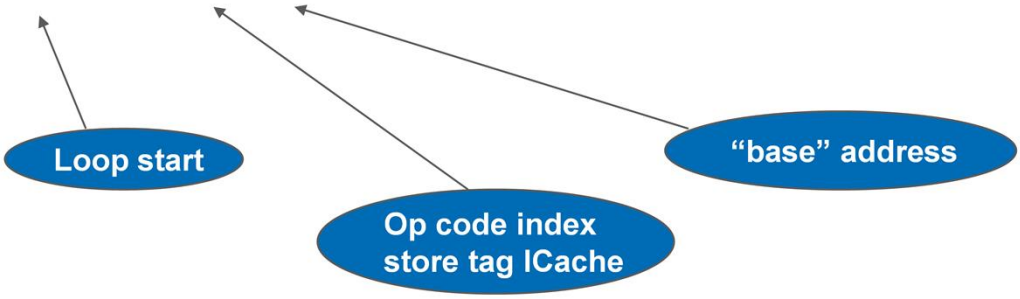
+ Bits two, three and four encode the cache operation.

+ For the upcoming example I will be using the Index Store tag operation on the Level 1 instruction cache so the op field is coded with a eight. The first two bits are 00 for the level one instruction cache and the operation code for Index Store tag is encoded as 010 in bits two, three and four.

Cache initialization

- Loop to initialize tag entries
  - Use cache operation to invalidate entry

```
1:    cache   0x8, 0($14)
```



MIPS

20

Here is the actual cache instruction code. To review

+ the cache instruction takes two arguments

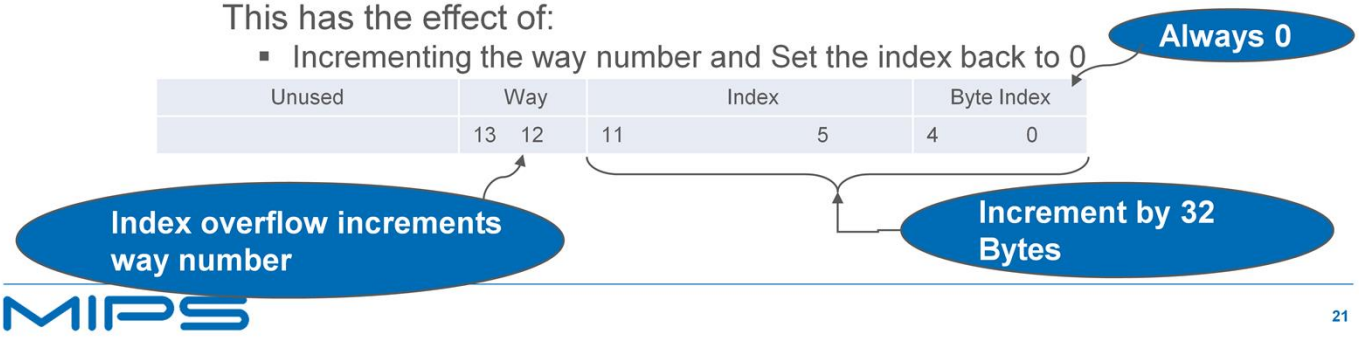
+ the base address which we set initially to 8 million hex to address set zero of the cache.

+ and the operation code which in this case is index store tag to the Instruction cache. This will move the tag data at is in CP0 TagLo and TagHi registers into the cache line that’s index by the instruction.

+ the assemble line is tagged with a one in column zero to give the code a place to branch back to and start the loop again.

Cache initialization

- How the code increments through cache ways
  - Finds out the size of each line. We will use this to increment the virtual address, “base” register, to the next cache index.
    - The “base” register will be incremented by a line size with each iteration of the loop. This will be done for the total number of sets (lines) in the cache which, assuming a 16K, 4 way cache would be 512.
    - That will cause the index to overflow its 7 bits when it reaches 128. This has the effect of:
      - Incrementing the way number and Set the index back to 0



MIPS

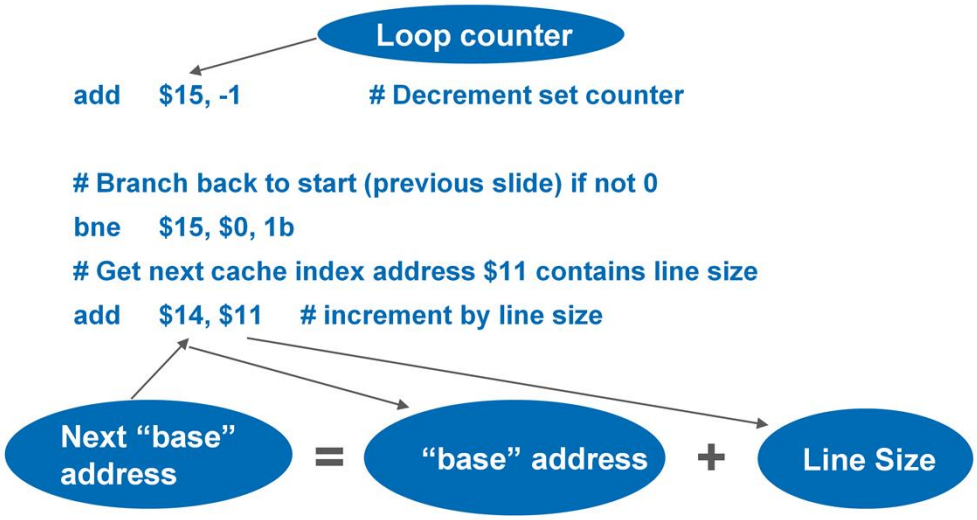
21

I want to explain one slightly tricky thing that will be happening in the code. The index type of operation can be used to address a byte in the cache in a specific way of the cache. This is done by breaking down the virtual address argument stored in the bass register of the cache instruction into several fields. The size of the index field will vary according to the size of a cache way. The larger the way the larger the index needs to be. In this example the combined byte and page index is 12 bits because each way of the cache is four K. The way number is always the next two bits following the index.

+ in this example the code does not explicitly set the way bits. Instead it just increments the virtual address by the cache lines size so the next time through the loop the cache instruction will initialize the next set in the cache.

+ Eventually this increment has the effect of setting the cache to index 0 of the next way in the cache because it over flows into the way bits.

Cache initialization



MIPS

Now all the code needs to do is loop maintenance.

+ first decrement the loop counter

+ then test it to see if its gotten to zero and if it has not branch back to label one

+ the branch delay slot which always gets executed is used to increment the virtual address to the next set in the cache.

Cache Management

- **Cache maintenance DMA**
  - Before a DMA out of memory:
    - If the data cache is in write back mode, some of the correct data may still be held in the D-cache but not written back to main memory.
    - Before the DMA device starts reading data out of memory, any data for that range of locations in the D-cache must be written back (flushed out) to memory.
  - DMA into memory:
    - If a device is loading data into memory, any cached entries of the memory locations must be invalidated to prevent the CPU reading stale data from the cache.

MIPS 23

Now lets talk about cache maintenance.

When working with cached data buffers where DMA is involved there are some necessary steps your code needs to take to insure data integrity.

+ If you are starting a DMA operation to DMA data out of memory

+ and the data is stored in cached write back pages some of the data may not be coherent with main memory.

+ So before the DMA starts the data must be forced back to main memory using a cache instruction to flush it from the cache to main memory.

+ If you are starting a DMA operation to DMA into a cached area of main memory

+ You will need to invalidate those addresses in the cache. If you don't and the cache could hold valid data for the buffer range. The upshot would be that the new data that the DMA operation brought into main

memory will not be brought into the cache.

Cache Management

- **I Cache and D cache Coherency**
  - When writing instructions, inserting software breakpoints and self-modifying code to cached memory instructions must be written back from the cache to memory and the corresponding memory location held in the Icache must be invalidated.
    - This is done using the SYNCI offset(base) instruction
      - The 16-bit offset is sign-extended and added to the contents of the base register to form an effective address. The effective address is used to address the cache line in all caches which may need to be synchronized with the write of the new instructions. The operation occurs only on the cache line which may contain the effective address. One SYNCI instruction is required for every cache line that was written.
    - Or mips_sync_icache macro can be used from C code. (covered in later slides)

MIPS

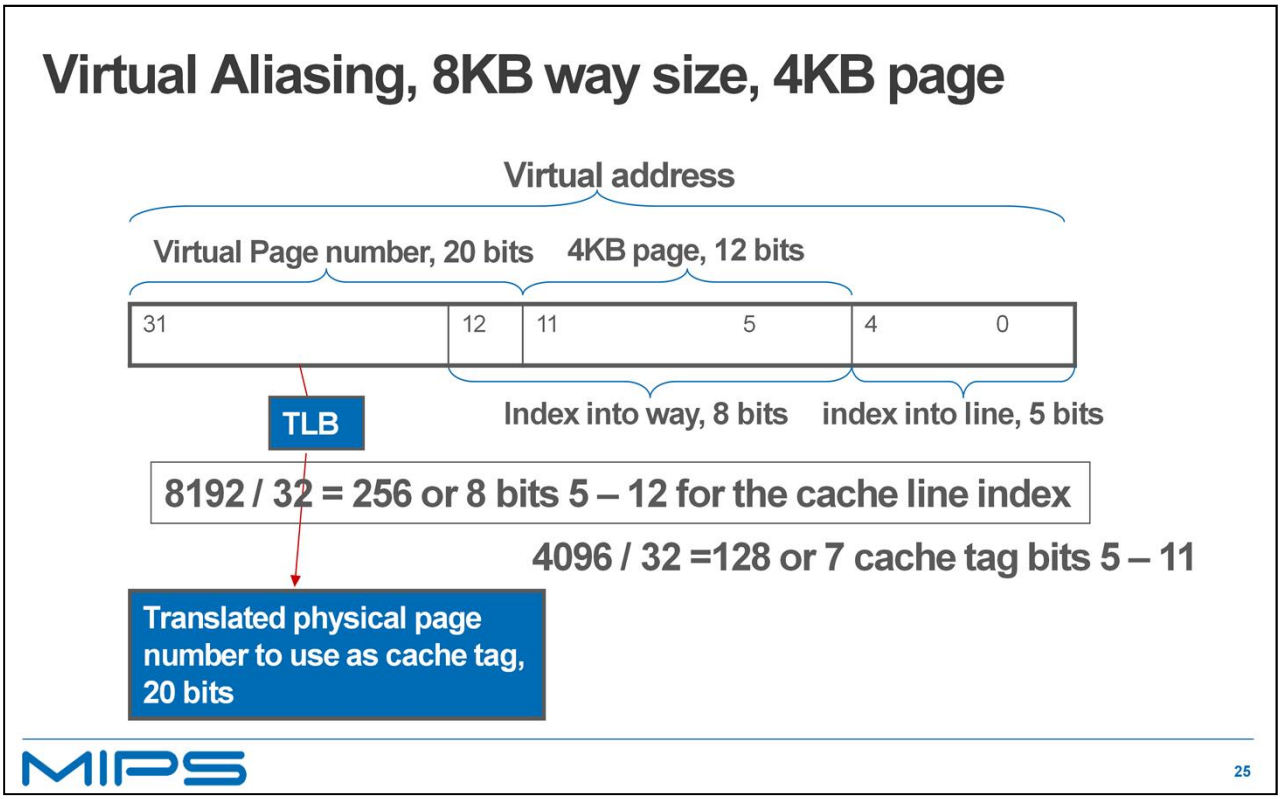
24

Another cases you need to worry about cache coherency is

+ when you write instructions usually done by a loader program or an OS, when you insert a break point usually done by a monitor program controlled by a debugger or when you code is self modifying which is done by many of the Java accelerators. In these cases the code written to the cache using store instructions will write to the data cache but the not the instruction cache. The instruction cache may already contain valid instructions for the memory you modified so it would not fetch new instructions. First the data cache and the main memory must be made coherent and then the area in the instruction cache that contains old instruction must be marked invalid.

+ There is one instruction that handles this for you called the SyncI instruction. This instruction takes the address of the instruction that you are changing and flushes the data in the cache line that contains the instruction out to main memory and checks to see that that same line in the instruction cache has been invalidated so the next time the instruction is fetched it will miss in the instruction cache and the cache line will be brought in from main memory.

+ There is also a macro provided with most tool chains call mips_sync_icache that can be used from C code.



Virtual Aliasing – happens when the number of sets in a way of the cache is larger than the number of sets needed for a size of a memory page.

For example Lets assume a thirty two K cache, with four ways and a thirty two byte line size. That would mean each way was eight K of the cache.

+ To index an eight K way it takes two hundred fifty six indexes given the thirty two byte line size.

That's eighty one thousand ninety two divided by thirty two bytes per line which equals two hundred and fifty six or eight bits, bits five through twelve for the cache line index.

*+ To index a four K page it takes one hundred and twenty eight cache lines

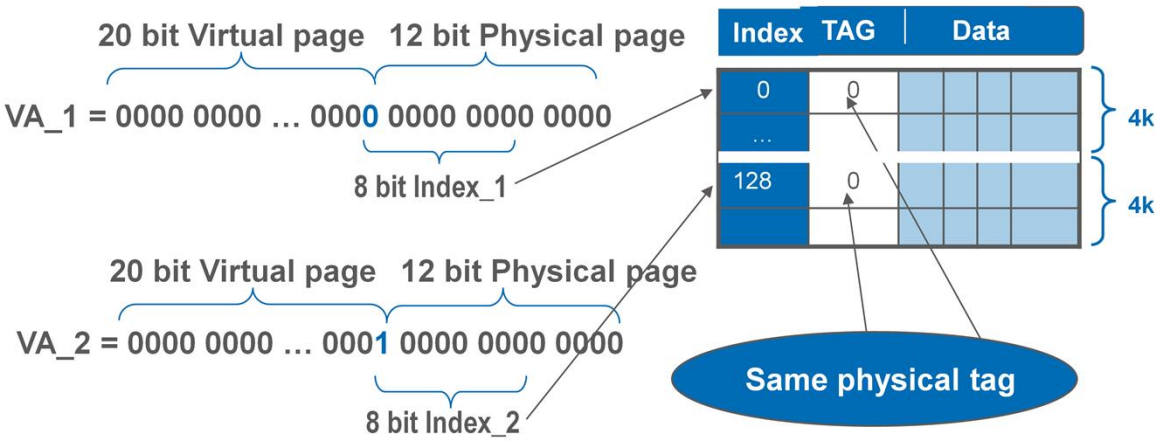
That's four thousand ninety six divided by thirty two bytes per line which equals one hundred and twenty eight cache lines or seven bits, bits five through eleven for the index into the page.

+ Leaving bits twelve through thirty-one for the physical tag

This causes the tag and the index to overlap at bit twelve.

Virtual Address Aliasing

Assume VA_1 and VA_2 translate to the same Physical Page (0) so they will both have the same physical cache tag



This overlap can mean that two different virtual addresses, mapped to the same physical location in memory can have two different entries into the cache. So the cache can end up with two different data values for the same physical memory.

+ Using the same cache dimensions I just showed you, one cache way would look like this. Each way could fit two four K pages.

For this example the first virtual address is address 0 and it translates to physical address 0.

+ It will look like this. As you can see the virtual address is made up of twelve bits to index into the page and 20 bits of virtual address to be translated by the TLB to a physical address. In this case page translated to is physical address 0.

+ the cache index, which I explained previously is eight bits, bits five through twelve, which for this address indexes to index zero of the cache.

+ Now lets look at a second virtual address. The 20 bits of virtual address to translate by the TLB also is translated to physical address zero.

+ the cache index of eight bits, bits five through twelve, which for this address will index to one hundred eight

+ both indexes end up with the same physical tag

Virtual Address Aliasing

- **Our Cores provide an optional hardware mechanism to prevent virtual address aliasing in 32KB and 64KB caches. Down side is a slightly slower clock rate.**
- **A few software solutions:**
  - Page size >= way size
  - The overlapping bits should be mapped straight through (VA = PA)
  - Align shareable data on a boundary greater than or equal to the largest way size
  - Flush and Invalidate shareable data entries at context switch

MIPS

27

The easiest way to avoid Aliasing is to create your core using the optional Hardware Anti aliasing option. There may be a small difference in the clock rate you are able to achieve using this option . To see if your core was built with this option you can check bit sixteen of the CP zero Config7 register.

+ If you don't have that hardware option the next easiest thing would be to make sure your page size is at least as large as the cache's way size.

+ another way is always map the overlapping bit straight through. In the example we just went through you would avoid creating the VA_2 address.

+ In most cases aliasing happens when you are sharing data between processes so another way to avoid them is to align shared data buffers virtual address on a way boundary. For example if your way size was eight K then align your buffers on a 8K boundary.

+ You can also Flush and invalidate sharable buffers at a contest switch.

By the way you should not have to worry about the instruction cache since it is read only. Also Versions of Linux from version 2.6.18 and upward have been designed to avoid data cache aliasing.

Data Prefetch

- **Data Prefetch**
  - The PREF instruction advises the CPU that you would like it to load data into the data cache if it has the opportunity to do so. In other words, it may or may not happen depending on activity on the data bus.
  - While your intention may be to increase the performance of your program that may not be the case.
    - Prefetching data may force other frequently used data out of the cache and refilling the cache when that data is needed again. You may therefore see no change, or even a decrease in performance.

Now lets discuss Perfetching, Data Prefetching is like what some fast food restaurants do. They cook food ahead of time anticipating the demand so they can shorten the customer wait time. Of course the amount of food the can cook ahead will depend on grill capacity and other needs for the grill. Such is the same for your data needs. You can use the prefetch instruction or the macro function provided in cpu.h, to get data into the cache ahead of time, anticipating the code's need for the data. For example when you are doing a memory copy from one buffer to another. The prefetch instruction tells the cpu to get the data into the cache if it does not interfere with other data bus operations.

+ Prefetching can be tricky because while you may think it should increase performance it might not. It will depend on your cache resource needs.

+ For example some times you can have a lot of frequently used data and if your code does a lot of prefetching it can force this data out of the cache. So you may increase the performance of a memory copy but decrease the performance of something else. The frequently used date could also force your pre-fetched data out of the cache before you actually use it. As a side note, this could be a good sign you need a larger cache.

Data Prefetch

- **Data Prefetch**
  - For prefetching to be effective, the prefetch must be done far enough in advance of use, so the data will be in the cache when needed.
  - For example, if you are using prefetching in a loop, you should start prefetching data in advance of entering the loop and continue to prefetch data several loops before it will be used. How far in advance depends on the speed of the bus, memory, and the number of instructions in the loop.
    - NOTE: When used in conjunction with DMA operations, you should take care to only prefetch a buffer where the DMA operation has been completed. Don't prefetch beyond the end of the buffer.

MIPS

29

Here's a few simple rules to follow:

+ prefetching to be effective needs to be done far enough ahead of time so that the data is in the cache when need.

+ For example in a copy loop you may need to be prefetching data several loop integrations before it's needed.

+ one caution be careful when you are using data the is being put into memory by a DMA device. In your effort to prefetch data enough ahead of its use the code could get ahead of the DMA operation and the data pre-fetched would not be fetching the new data.

Data Prefetch Example Code

- **memcopy example**
  - The following code example is meant to show you how to start going about finding the most optimal prefetching required.
  - The example is a simplified version of memcopy.
    - Assumptions have been made so the code will give you a clear focus on prefetching. It assumes the following:
      - Code is aligned to a cache line boundary and a cache line is 32 bytes.
      - There is > 512 bytes to copy
    - Arguments to assembly function are:

      *int input, (register a0)

      *int output, (register a1)

      int Bytes_to_copy (register a2)

      Int Cycles_taken (register v1)

MIPS

I going to show you an example of prefetching in a memory copy to give you an idea of what you may need to do.

+ the code is simplified to highlight the prefetching being done.

+ If this were a actual memory copy the code would need to have some logic to align the copy on a cache line boundary.

+ also, it would decide what the minimum number of bytes would be to make the prefetching useful.

+ here are the arguments used in the example.

Data Prefetch Example Code

- **The performance counter register will be used to measure cycle performance.**
- **The idea is to experiment with the code by trial and error to get the best performance number for the copy.**
  - Why experiment? Isn't there a formula that could be used?
    - There are so many variables like memory speed, bus speed. CPU speed, fill store buffers, number of out standing loads, number of outstanding stores, L2 cache refill etc… It just easier to do a simple experiment!

MIPS

31

I have also include, how to use a performance counter register to measure how many cycles the copy takes so you can see how well the code is functioning.

+ For your system you will need to experiment to see what gives you the best results.

+ There are just too many variables to be able to calculate how pre-fetching is going to effect the code.

Data Prefetch Example Code

Performance Counters Register 25	
Select	**Register**
0	Counter 0 Control
1	Counter 0 Count
2	Counter 1 Control
3	Counter 1 Count

```
li      t1, 2           # count cycles for kernel mode
mtc0    t1, C0_PERFCNT, 0
ehb
mtc0    $0, C0_PERFCNT, 1   # clear the counter
```

MIPS

32

Here is an example of pre-fetching.

First I going to use an internal counter register to determine how long the copy took so I can judge the performance of the pre-fetching.

Each of our cores contains at least two performance counters.

+ The performance counters are located at co processor 0 register 25.

+ Each counter is broken down into two CP0 registers, the even select register is the control register and the odd select register is the actual count register. For example lets say the core we are using has two performance counters Co processor 0 register 25 select 0 is the control register for counter 0 and select 1 is the counter for counter 0. the second counter's control register would be co processor register 25 select 2 and its counter would be select 3.

+ in the code you need to tell the CPU what you want to count. I want to count cycles. Writing a 2 to the performance counter zero's control register programs it to count cycles. I do this by loading a 2 into register t1 and the using the move to co processor zero instruction move it to co

processor 0 register 25 select 0.

+ I use the EHB instruction to insure the register is written before I go on

+ Then I clear the count in the performance counter by move register 0 to co processor 0 register 25 select 1.

Data Prefetch Example Code

```
 addu v1, a0, a2        # count a2 plus address of input buffer

#   prefetch ahead 512 bytes (0x0 to 0x200)

    pref 0, 0x0(a0)      # prefetch the first source line
    pref 30, 0x0(a1)     # prefetch the first destination line

    pref 0, 0x20(a0)     # prefetch the next source line
    pref 30, 0x20(a1)    # prefetch the next destination line
    ……                   #.. Continue to 0x1e0
1: # branch back here to do more Prefetching
    pref 0, 0x200(a0)    # prefetch the next source line
    pref 30, 0x200(a1)   # prefetch the next destination line
```

MIPS

33

For this code I decided to try prefetching five hundred and twelve bytes ahead and see what numbers I get.

+ I have paired the input and out put prefetches.

The input prefetch will prefetch a cache line into memory. The output prefetch will prepare the cache for writing an entire line, without the overhead involved in filling the line from memory.

+ I am going to continue with prefetch instruction until I have given enough to fetch five hundred and twelve bytes.

+ at the point I will be looping back to, I will prefetch the next lines to always keep ahead five hundred and twelve bytes.

Info only:

pref 0, = load

pref 30, = prepare for store

Data Prefetch Example Code

```
2:  # Branch to here when no more Prefetching should be done
# Load Data from input

    lw t0, 0x0(a0)
    lw t1, 0x4(a0)
    lw t2, 0x8(a0)
    lw t3, 0xc(a0)
    lw t4, 0x10(a0)
    lw t5, 0x14(a0)
    lw t6, 0x18(a0)
    lw t7, 0x1c(a0)

# Increment base address of input by cache line size

    add a0, a0, 0x20
```

MIPS

I have a second branch point I will branch back to when I have prefetched to the end of the buffer.

+ Next I will load a whole cache line from the input buffer

+ and then increment the base address by a line size.

Data Prefetch Example Code

```
# Store Data to output

    sw t0, 0x0(a1)
    sw t1, 0x4(a1)
    sw t2, 0x8(a1)
    sw t3, 0xc(a1)
    sw t4, 0x10(a1)
    sw t5, 0x14(a1)
    sw t6, 0x18(a1)
    sw t7, 0x1c(a1)

    addiu v0, a0, 0x200  # calculate fetch ahead
```

MIPS

35

Now store a cache lines worth of data to the destination address

+ increment the next address to fetch ahead

Data Prefetch Example Code

```
# prefetch more, Fetched ahead enough?
    bltu v0, v1, 1b
# Increment base address of output by cache line size (BDS)
    add a1, a1, 0x20

# Done copying?
    bne a0, v1, 2b
    nop

    mfc0 v1, C0_PERFCNT, 1  # counter to v1 to return to caller
    jr     ra
    nop
```

MIPS

36

+ The code checks to see if we have prefetched the entire buffer yet and if we have not branch back to the prefetch branch point.

+ In the Branch delay slot, that will always get executed no matter if the branch is taken or not, the code increments the output buffer pointer by a cache line size.

+ The code will reach this point once it has fetched all it can but it still needs to copy the last final five hundred and twelve bytes. It checks to see if there is more to copy and will branch back to the non fetch point if there is.

+ when the copy is done the code reads the performance counter to get the number of cycles it took to do the copy and puts it into the return register. Then returns to the calling function.

You can experiment with the code by changing the number of prefetches done ahead to see what gives you the best numbers on your system.

Cache Exceptions

- **If your caches have error detection logic they can communicate with the Core and generate Cache exceptions.**
  - If your system uses Parity the errors are usually serious errors that cannot be corrected. They are warnings that the data is unreliable and allows a somewhat controlled shutdown instead of random failures.
  - If your system uses ECC hardware it will correct single bit errors, and generate a parity error (Cache exception) for multiple bit errors.

MIPS

37

Lets talk about cache exceptions. The cache can generate exceptions. Cache exceptions will depend on your hardware. If your cache is designed to report errors, it can do so by causing a cache exception. Cache error detection is implementation dependent.

+ If your cache has parity detection, it will generate a cache exception when it gets a parity error. In general a parity error in the cache is fairly serious. One problem is the exception is imprecise, which means it may be difficult to determine which instruction the data belongs to or in the case of the instruction cache which instruction coming up has a problem.

+ If your cache has error correcting hardware the hardware will correct the first error without generating an exception. It will generate an exception if there uncorrectable errors.

Cache Exceptions

- **Cache exceptions have their own vector that is located in the non-cached KSEG1 segment.**
  - 0xBFC0 0300 on boot (BEV =1)
  - 0xA000 0100 on boot (BEV =0)
- **Cache Error detection logic uses the CP0 CacheErr register (27) to communicate information on the error**

MIPS

A cache exception always vectors to a non cached virtual address in KSGE1.

+ on boot this address in BF C0 03 00 hex.

+ Once the boot code finishes initialization, it should set the boot exception bit that which changes the cache exception vector to A0 00 01 00 hex.

+ There is a separate status register for cache errors called the Cache Error register. This is Co processor register zero register 27.

Cache Exceptions

Cache Error Register CP0 #27											
31	30	29	28	27	26	25	24	23	22	21 20	19 0
ER	EC	ED	ET	ES	EE	EB	EF	SP	EW	Way	Index

ER	Error Reference type. 0=Instruction; 1=Data
EC	Cache level at which the error was detected. 0=Primary; 1=Non-primary
ED	Error Data. Indicates a data RAM error.
ET	Error Tag. Indicates a tag RAM error.
ES	Error source. Indicates whether error was caused by internal processor or external snoop request. 0 = Internal; 1 = External
EE	Error external. Indicates whether a bus parity error was detected.
	Not supported – always reads 0
EB	Error both. Indicates that a data cache error occurred in addition to an instruction cache error In the case of an additional data cache error, the remainder of the bits in this register are set according to the instruction cache error.
EF	Error Fatal. Indicates that a fatal cache error has occurred.
SP	Scratchpad. Indicates a Scratchpad RAM error
EW	Error Way. Indicates a way selection RAM error.
Way	Way. Specifies the cache way error was detected.
Index	Specifies the cache index of the double word in which the error was detected.

MIPS

39

This chart breaks down the cache error register. One thing to keep in mind is that this is somewhat implementation dependent. The implementer of your chip will make the decision to include cache error logic or not. That person will also determine which parts of the cache error register to support.

Cache Exceptions

- **ErrCtl Register CP0 #26:**
  - PE bit is the parity enable bit
  - FE or First error to indicate this is the first unhandled cache error
  - SE or Second error to indicate a second error was detected before the first was handled

This is another coprocessor zero register, number 26 the Error control register that plays the part in controlling the cache parity and of telling you if the Core detected another cache error while the exception code was in the process of handling a cache error.

+ Your code can use the PE bit to determine if the core supports cache parity. If the code writes a one to the PE bit and reads a 0 it means that there is no parity support in the cache. This bit is set to zero when the core boots. Once the cache has been initialized the code should write a one to this bit to enable cache parity

+ The FE bit is set to indicate the first instance of a cache exception.

+ and the SE bit is set when the core detects a second error before the first error has been handled.

There are other control bits in this register that can be used for cache testing. Check the Software Users manual for more information.