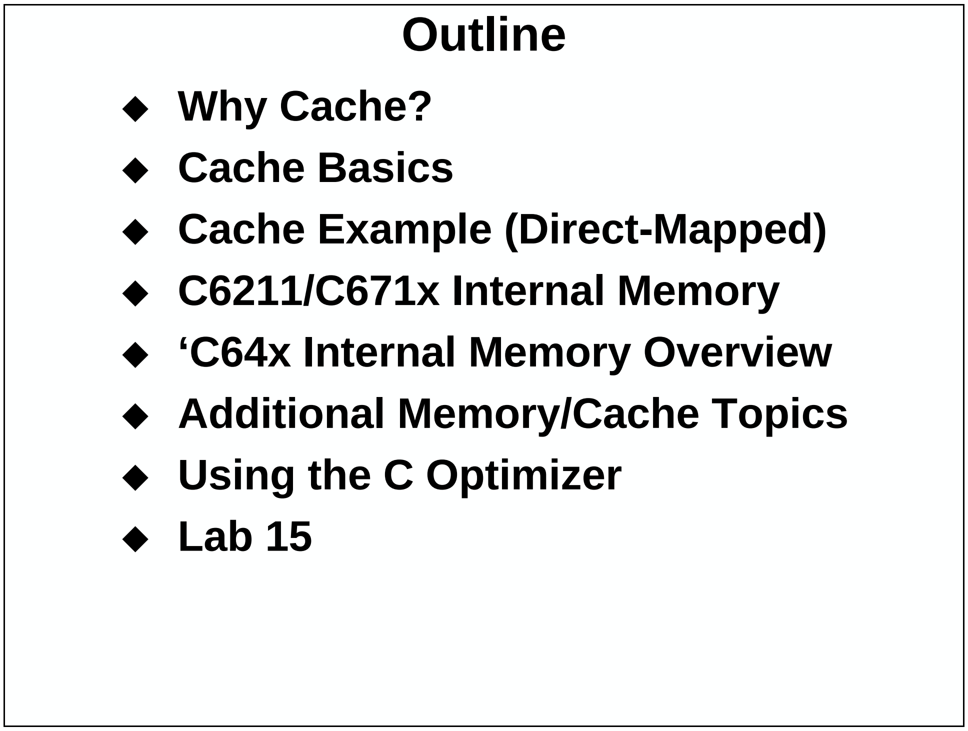
Internal Memory & Cache

Introduction

As the performance of DSPs increase, the ability to put large, fast memories on-chip decreases. Current silicon technology has the ability to dramatically increase the speed of DSP cores, but the speed of the memories needed to provide single-cycle access for date and instructions to these cores are limited in size. In order to keep DSP performance high while reducing cost, large, flat memory models are being abandoned in favor of caching architectures. Caching memory architectures allow small, fast memories to be used in conjunction with larger, slower memories and a cache controller that moves data and instructions closer to the core as they are needed. The ‘C6x1x devices provide a two-level cache architecture that is flexible and powerful. We'll look at how to configure the cache and use it effectively in a system.



Outline

- Why Cache?
- Cache Basics
- Cache Example (Direct-Mapped)
- C6211/C671x Internal Memory
- ‘C64x Internal Memory Overview
- Additional Memory/Cache Topics
- Using the C Optimizer
- Lab 15

Chapter Topics

Why Cache? *15-3*
Cache vs. RAM 15-5

Cache Fundamentals *15-7*

Direct-Mapped Cache *15-11*
Direct-Mapped Cache Example 15-12
Three Types of Misses 15-20

C6211/C671x Internal Memory *15-21*
L1 Data Cache (L1P) 15-22
L1 Data Cache (L1D) 15-25
L2 Memory 15-29
L2 Configuration 15-34

C64x Internal Memory Overview *15-36*

Additional Memory/Cache Topics *15-37*
'C64x Memory Banks 15-37
Cache Optimization 15-39
Data Cache Coherency 15-40
“Turn Off” the Cache (MAR) 15-49

Using the C Optimizer *15-52*
Compiler Build Options 15-52
Using Default Build Configurations (Release) 15-53
Optimizing C Performance (where to get help) 15-53

Lab15 – Working with Cache *15-54*

Lab 15 Procedure *15-55*
Move Buffers Off Chip and Turn on the L2 Cache 15-55
Use L2 Cache Effectively 15-59

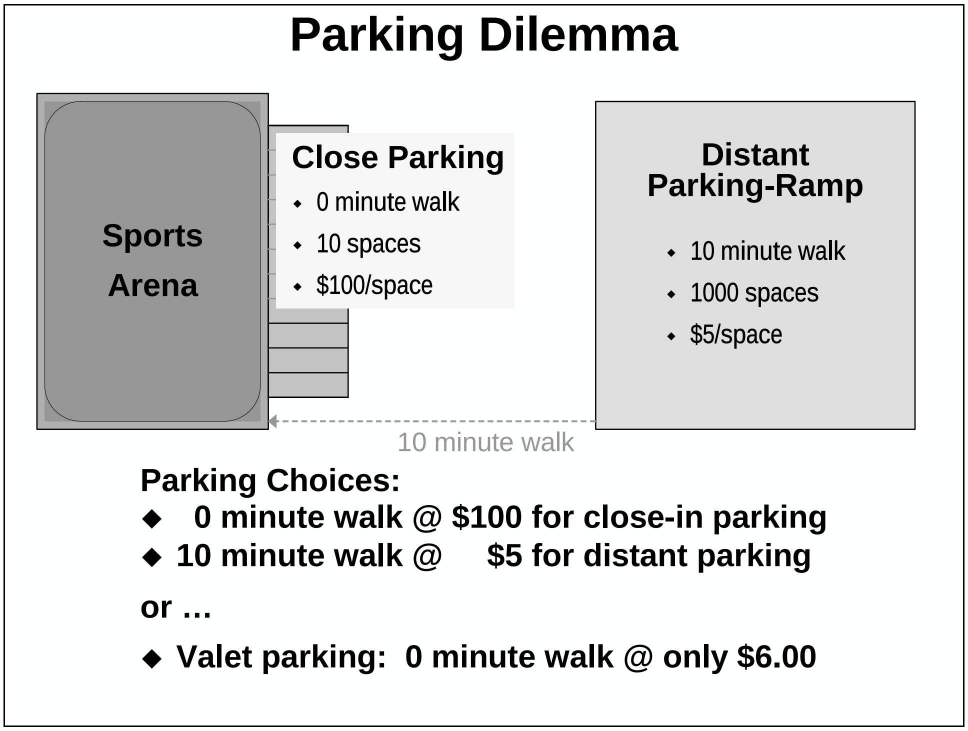
Lab15a – Using the C Compiler Optimizer *15-62*

Optional Topics *15-65*
‘0x Memory Summary 15-65
‘0x Data Memory – System Optimzation 15-66

Why Cache?

In order to understand why the C6000 family of DSPs uses cache, let's consider a common problem. Take, for example, the last time you went to a crowded event like the symphony, a sporting event, or the ballet, any kind of event where a lot of people want to get to one place at the same time. How do you handle parking? You can only have so many parking spots close to the event. Since there are only so many of them, they demand a high price. They offer close, fast access to the event, but they are expensive and limited.

Your other option is the parking garage. It has plenty of spaces and it's not very expensive, but it is a ten minute walk and you are all dressed up and running late. It's probably even raining. Don't you wish you had another choice for parking?



Parking Dilemma

Sports Arena

Close Parking
- 0 minute walk
- 10 spaces
- $100/space

Distant Parking-Ramp
- 10 minute walk
- 1000 spaces
- $5/space

10 minute walk

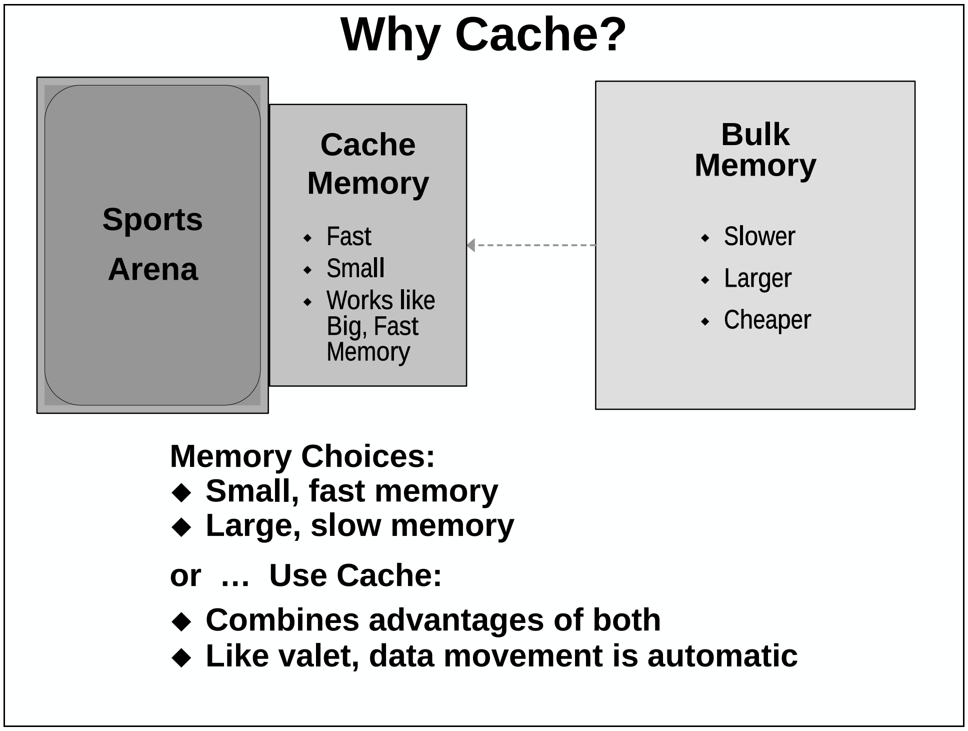
Parking Choices:
- ◆ **0 minute walk @ $100 for close-in parking**
- ◆ **10 minute walk @ $5 for distant parking**

or …

- ◆ **Valet parking: 0 minute walk @ only $6.00**

You do! A valet service gives the same access as the close parking for just a little more cost than the parking garage. So, you arrive on time (and dry) and you still have money left over to buy some goodies.

Cache is the valet service of DSPs. Memory that is close to the processor and fast can only be so big. You can attach plenty of external memory, but it is slower. Cache helps solve this problem by keeping what you need close to the processor. It makes the close parking spaces look like the big parking garage around the corner.



Why Cache?

Sports Arena

Cache Memory

- Fast
- Small
- Works like Big, Fast Memory

Bulk Memory

- Slower
- Larger
- Cheaper

Memory Choices:

- **Small, fast memory**
- **Large, slow memory**

or … Use Cache:

- **Combines advantages of both**
- **Like valet, data movement is automatic**

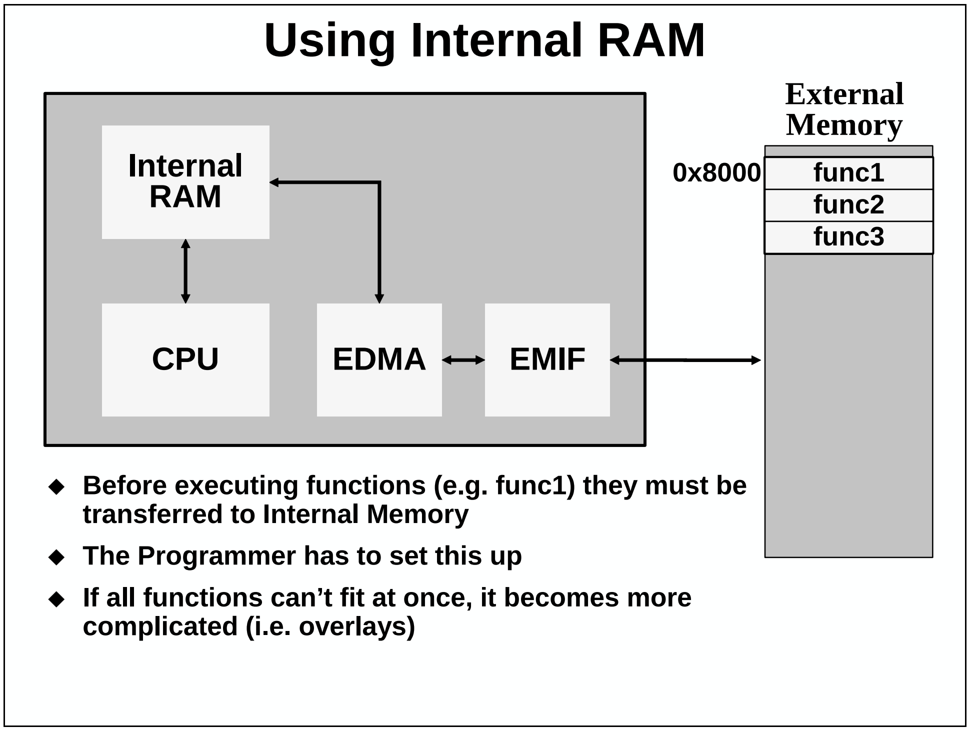
One of the often overlooked advantages of cache is that it is automatic. Data that is requested by the CPU is moved automatically from slower memories to faster memories where it can be accessed quickly.

Cache vs. RAM

Using Internal Program as RAM

DSPs achieve their highest performance when running code from on-chip program RAM. If your program will fit into the on-chip program RAM, use the DMA or the Boot-Loader to copy it there during system initialization. This method of using the DMA or a Boot-Loader is powerful, but it requires the system designer to set everything up manually.

If your entire system code cannot fit on chip but individual, critical routines will fit, place them into the on-chip program RAM as needed using the DMA. Again, this method is manual and can become complex very quickly as the system changes and new routines are added.



Using Internal RAM

- **Before executing functions (e.g. func1) they must be transferred to Internal Memory**
- **The Programmer has to set this up**
- **If all functions can't fit at once, it becomes more complicated (i.e. overlays)**

In the example above, the system has three functions (func1, func2, and func3) that will fit in the on-chip program memory located at 0x0. The system designer can set up a DMA transfer from 0x8000 to 0x0 for the length of all three functions. Then, when the functions are executed they will run from quick on-chip memory.

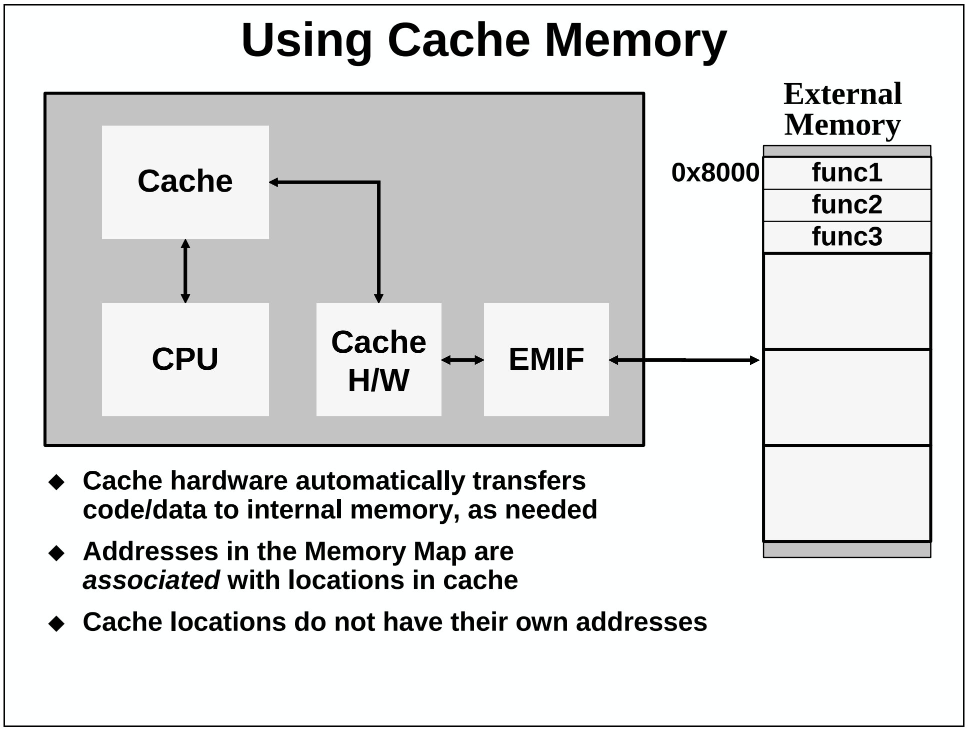
Unfortunately, the details of setting up the DMA-copy are left to the designer. Several of these details change every time the system/code is modified (i.e. addresses, section lengths, etc.).

Worse yet, if the code grows beyond the size of the on-chip program memory, the designer will have to make some tough choices about what to execute internally, and which to leave running from external memory. Either that, or implement a more complicated system which includes overlays.

Using Cache

The cache feature of the ‘C6000 allows the designer to store code in large off-chip memories, while executing code loops from fast on-chip memory … *automatically*.

That is, the cache moves burden of memory management from the designer to the cache controller – which is built into the device.



Using Cache Memory

- **Cache hardware automatically transfers code/data to internal memory, as needed**
- **Addresses in the Memory Map are *associated* with locations in cache**
- **Cache locations do not have their own addresses**

Notice that Cache, unlike the normal memory, does not have an address. The instructions that are stored in cache are *associated* with addresses in the memory map. Over the next few pages we further describe the term *associated* along with how cache works, in general.

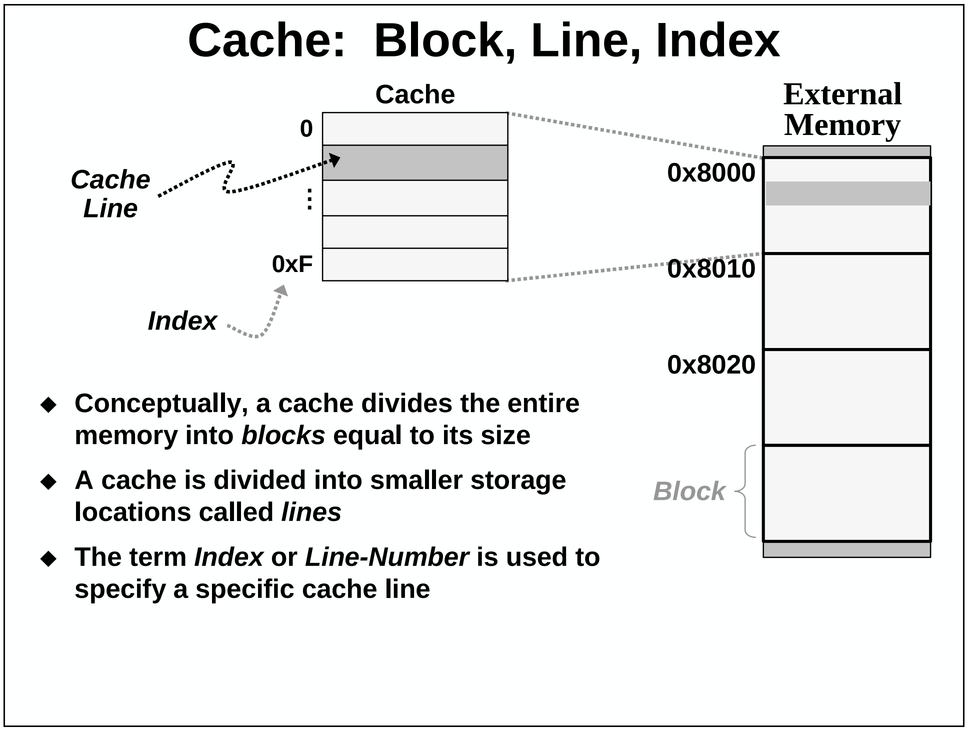
Cache Fundamentals

As stated earlier, locations in cache memory do not have their own addresses. These locations are associated with other memory locations. You may think of it like cache locations “shadowing” addressable memory locations (usually a larger, slower-access memory).

As part of its function, cache hardware and memory must have an organizational method to keep track of what addressable memory locations it contains.

Blocks, Lines, Index

One way to think about how a direct-mapped cache works is to think of the entire memory map as blocks. These **blocks** are the same size as the cache. The cache block is further broken into lines. A **line** is the smallest element (location) that can be specified in a cache. Finally, we number each line in the cache. This is often called an **index**, or more obviously, **line-number**.



Cache: Block, Line, Index

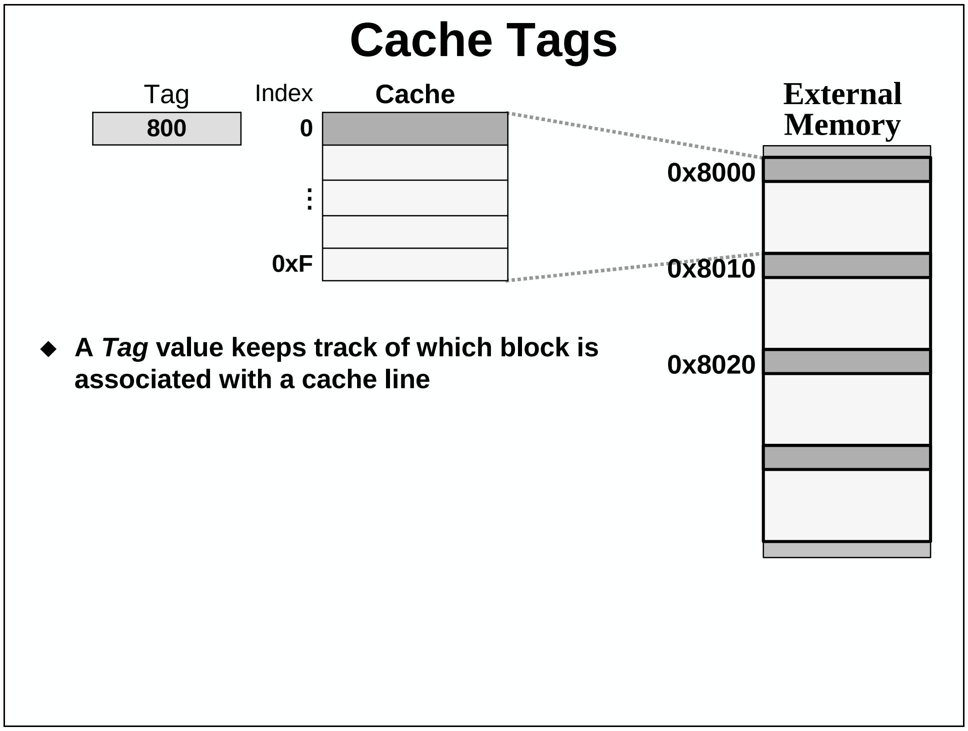
- **Conceptually, a cache divides the entire memory into *blocks* equal to its size**
- **A cache is divided into smaller storage locations called *lines***
- **The term *Index* or *Line-Number* is used to specify a specific cache line**

In the example above, the cache has 16 lines. Therefore, the entire memory map (or at least the part that can be cached) is broken up into 16 line blocks. The first line of each block is associated with the first line in cache; the second line of each block is associated with the second line of cache, continuing out to the 16th line. If the first line of cache is occupied by information from the first block and the DSP accesses the same line from the second block, the information in the cache will be overwritten because the two addresses reside at the same line.

Cache Tag

When values from memory are copied into a line or more of cache, how can we keep track of which block they are from?

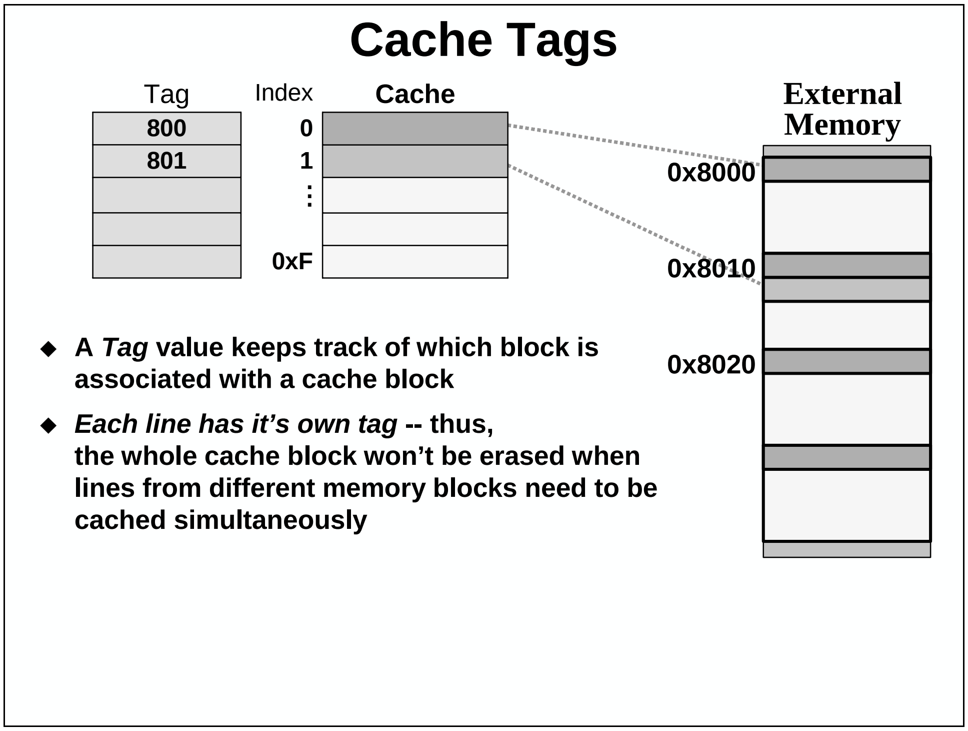
The cache controller uses the address of an instruction to decide which line in cache it is associated with, and which block it came from. This effectively breaks the address into two pieces, the *index* and the *tag*. The index determines which line of cache an instruction will reside at in cache (and the lower order bits of the address represent it). The **tag** is the higher order bits of the address, and it determines which block the cache line is associated with in the memory map.



Cache Tags

- **A *Tag* value keeps track of which block is associated with a cache line**

While a single tag will allow the cache to discern which block of memory is being “shadowed”, it requires all lines of the cache to be associated with the same block of memory. As caches become larger, as is the case with the C6000, you may want different lines to be associated with different blocks of memory. For this reason, each line has an associated tag.



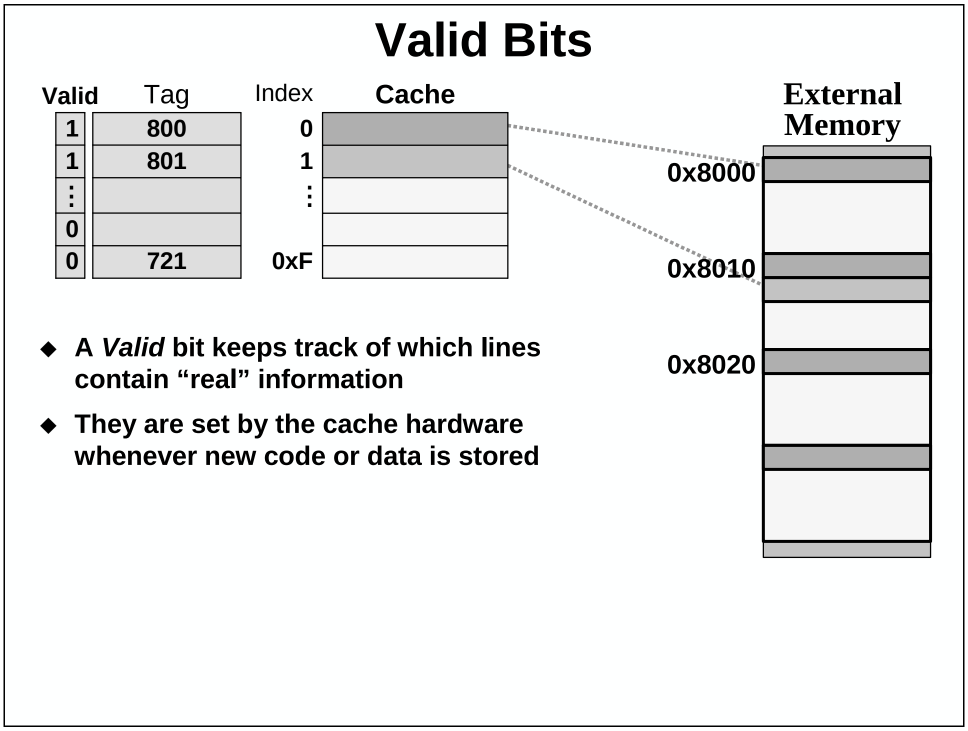
Cache Tags

- A *Tag* value keeps track of which block is associated with a cache block
- *Each line has it’s own tag* -- thus, the whole cache block won’t be erased when lines from different memory blocks need to be cached simultaneously

Valid Bits

Just because a cache can hold, say, 4K bytes, that doesn't mean that all of its lines will always have valid data. Caches provide a separate **valid bit** for each line. When data is brought into the cache, the valid bit is set.

When a CPU load instruction reads data from an address, the cache is examined to see if the valid, specified address exists in the cache. That is, at the index specified by the address, does the correct tag value exist and is it marked valid?



Valid Bits

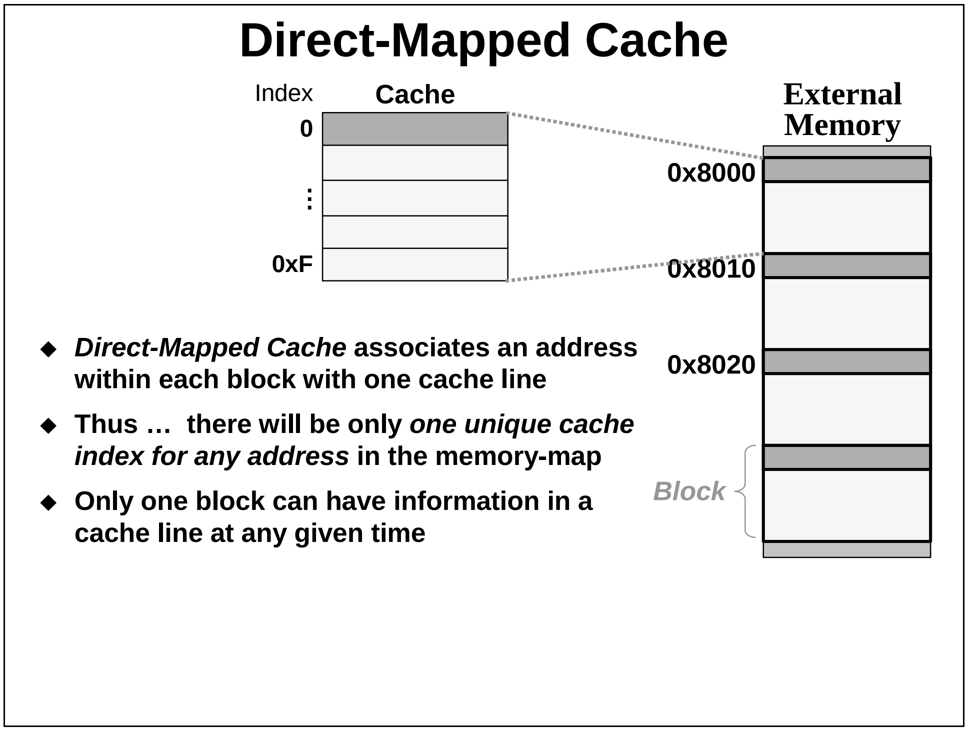
Valid	Tag	Index	Cache
1	800	0	
1	801	1	
⋮		⋮	
0			
0	721	0xF	

- A *Valid* bit keeps track of which lines contain "real" information
- They are set by the cache hardware whenever new code or data is stored

Note: Given a 4K byte cache, do the bits associated with the cache management (tag, valid, etc.) use up part of the 4K bytes? The answer is *No*. When a 4K byte cache is specified, we are indicating the amount of *usable* memory.

Direct-Mapped Cache

A Direct-Mapped cache is a type of cache that associates each one of its lines with a line from each of the blocks in the memory map. So, only one line of information from any given block can be live in cache at a given time.



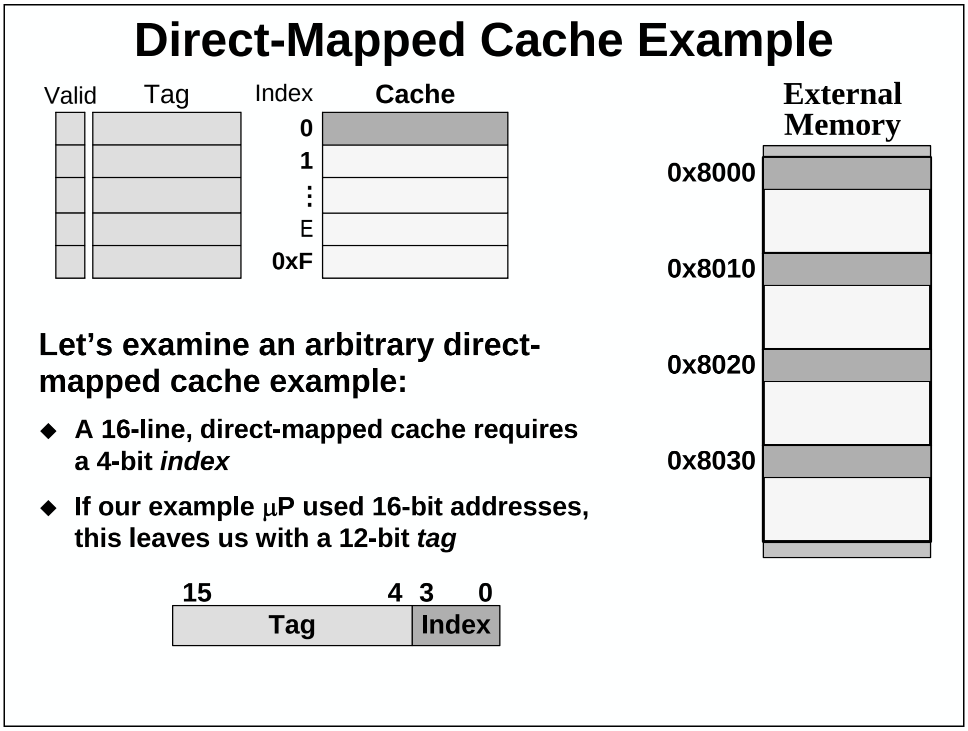
Direct-Mapped Cache

- ◆ ***Direct-Mapped Cache*** **associates an address within each block with one cache line**
- ◆ **Thus … there will be only *one unique cache index for any address* in the memory-map**
- ◆ **Only one block can have information in a cache line at any given time**

Another way to think about this is, “For any given memory location, it will map into one, and-only-one, line in the cache.”

Direct-Mapped Cache Example

In the example below, we have a 16-line cache. How many bits are needed to address 16 lines? The answer of course is four, so this is the number of bits that we have as the index. If we have 16-bit addresses, and the lowest 4-bits are used for the index, this leaves 12-bits for the tag. The tag is used to determine from which 16-line block of memory the index came.

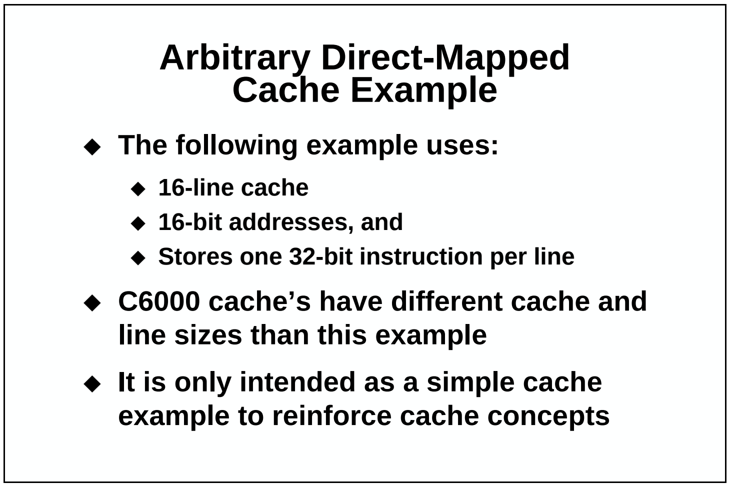


Direct-Mapped Cache Example

Let's examine an arbitrary direct-mapped cache example:

- **A 16-line, direct-mapped cache requires a 4-bit *index***
- **If our example μP used 16-bit addresses, this leaves us with a 12-bit *tag***

The best way to understand how a cache works is by studying an example. The example below illustrates how a direct-mapped cache with 16-bit addresses operates on a small piece of code. We will use this example to understand basic cache operation and define several terms that are applicable to caches.

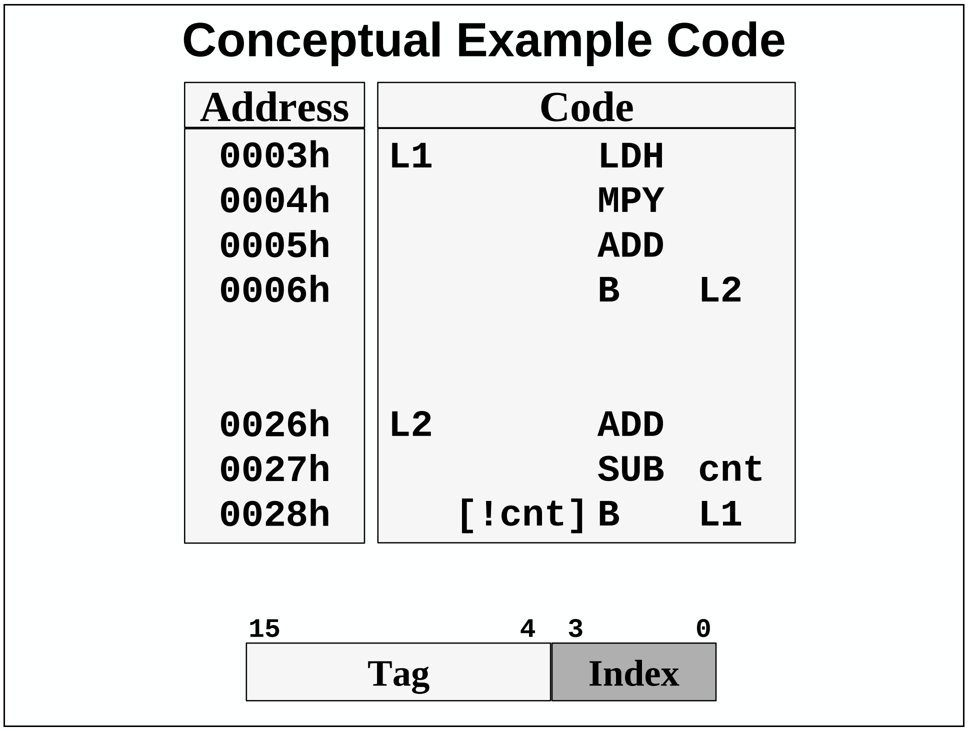


Arbitrary Direct-Mapped Cache Example

- **The following example uses:**
  - **16-line cache**
  - **16-bit addresses, and**
  - **Stores one 32-bit instruction per line**
- **C6000 cache's have different cache and line sizes than this example**
- **It is only intended as a simple cache example to reinforce cache concepts**

Note: The following cache example does not illustrate the exact operation of a 'C6000 cache. The example has been simplified to allow us to focus on the basic operation of a direct-mapped cache. The operation of a 'C6000 cache follows the same basic principles.

Example



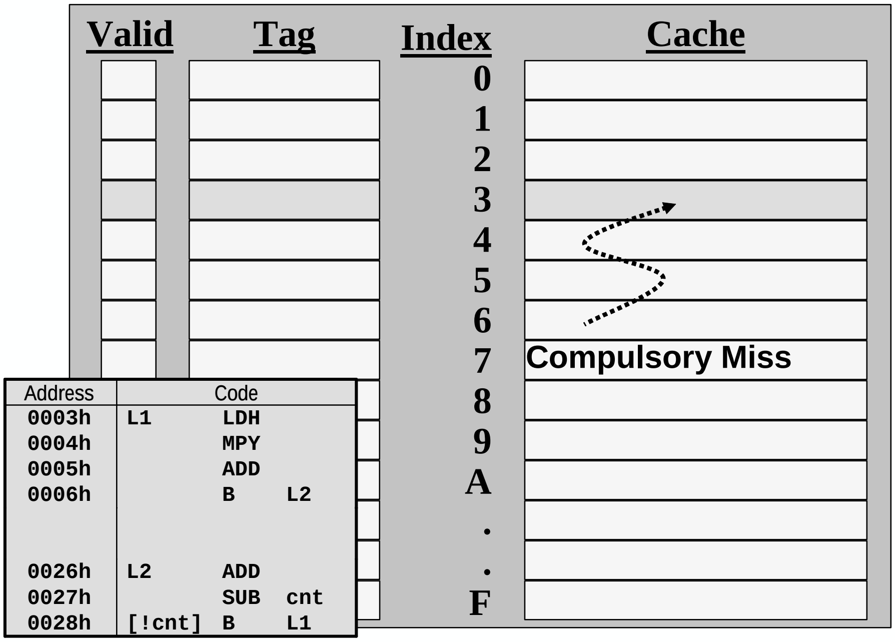
Conceptual Example Code

Address	Code
0003h	`L1          LDH`
0004h	`            MPY`
0005h	`            ADD`
0006h	`            B    L2`
0026h	`L2          ADD`
0027h	`            SUB  cnt`
0028h	`    [!cnt]  B    L1`

15 … 4	3 … 0
Tag	Index

The first time instructions are accessed the cache is cold. A cold cache doesn't have anything in it. When the DSP accesses the first instruction of our example code, the LDH, the cache controller uses the index, 3, to check the contents of the cache. The cache controller includes a valid bit for each line of cache. As you can see below, the valid bit for line 3 is not set. Therefore, the LDH instruction causes a *cache miss*. More specifically, this is called a *compulsory miss*. The instruction has to be fetched from memory at its address, 0x0003. This operation will cause a delay until the instruction is brought in from memory.

Direct Mapped Cache Example

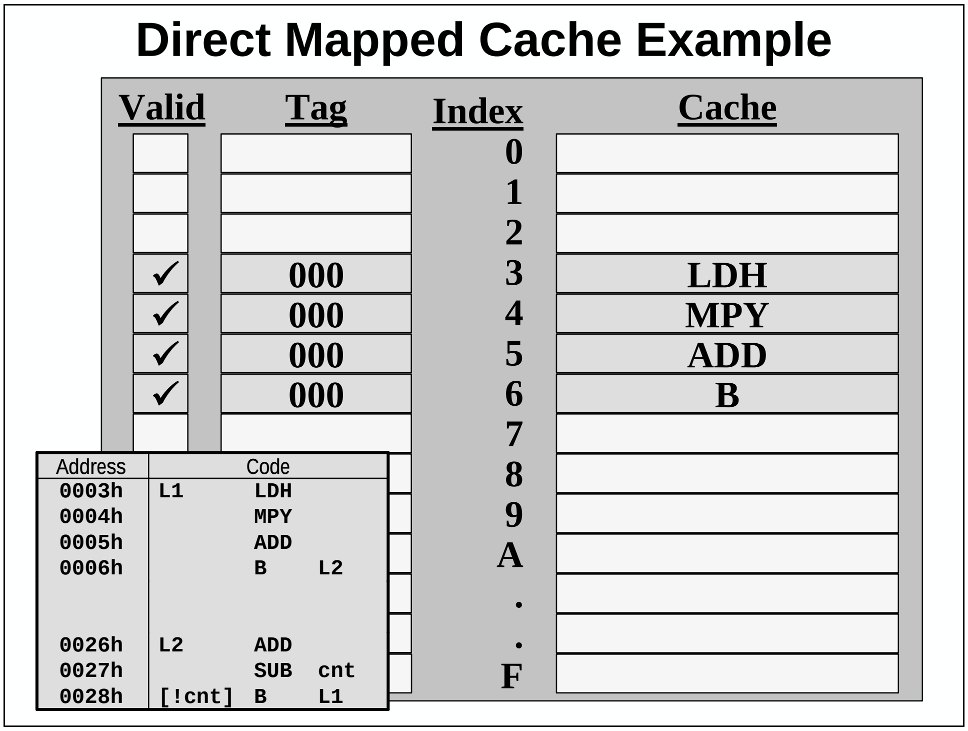


```
Address   Code
 0003h   L1      LDH
 0004h           MPY
 0005h           ADD
 0006h           B    L2


 0026h   L2      ADD
 0027h           SUB  cnt
 0028h   [!cnt]  B    L1
```

When the LDH instruction is brought in from memory, it is given to the core and added to the cache at the same time. This operation minimizes the delay to the core. When the instruction is added to the cache, it is added to the appropriate index line, the tag is updated, and the valid bit is set.

The following three instructions are added to the cache in the same manner. When they have all been accessed, the cache will look like this:



Direct Mapped Cache Example

Valid	Tag	Index	Cache
		0	
		1	
		2	
✓	000	3	LDH
✓	000	4	MPY
✓	000	5	ADD
✓	000	6	B
		7	
		8	
		9	
		A	
		.	
		.	
		F	

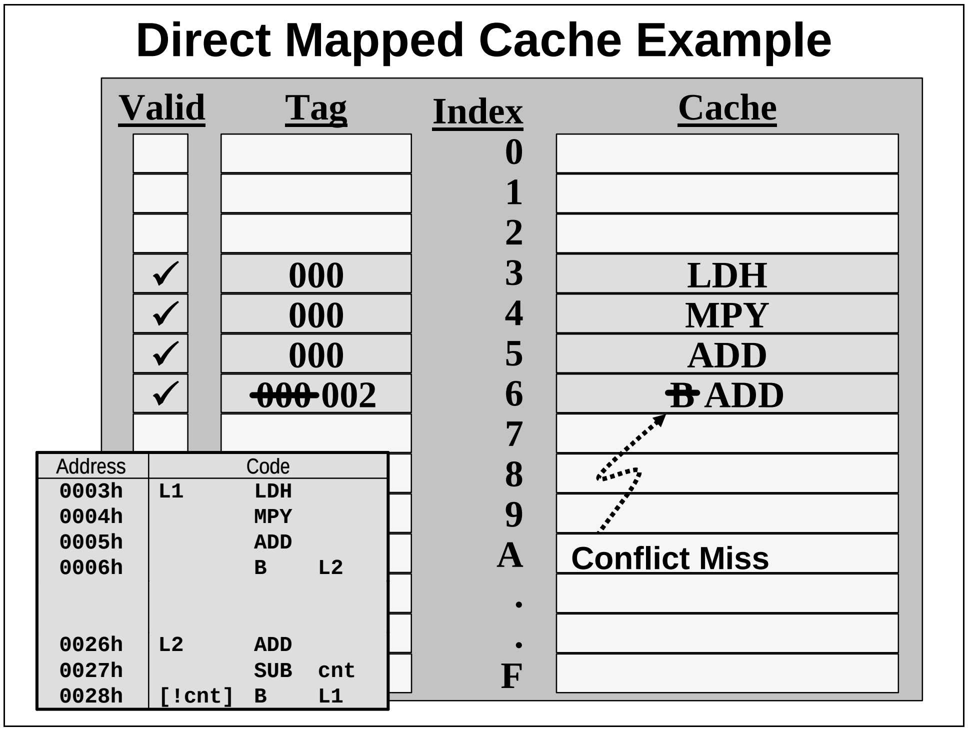
```
Address   Code
 0003h   L1      LDH
 0004h           MPY
 0005h           ADD
 0006h           B    L2


 0026h   L2      ADD
 0027h           SUB  cnt
 0028h   [!cnt]  B    L1
```

Notice that the branch instruction is the last instruction that was transferred by the cache controller. A branch by definition can take the DSP to a new location in memory. The branch in this case takes us to the label *tst*, which is located at 0x0026.

When the CPU fetches the ADD instruction, it checks the cache to see if it currently resides there. The cache controller checks the index, 6, and finds that there is something valid in cache at this index. Unfortunately, the tag is not correct, so the add instruction must be fetched from memory at its address.

Since this is a direct-mapped cache, the ADD instruction will overwrite whatever is in cache at its index. So, in our example, the ADD will overwrite the B instruction since they share the same index, 6.



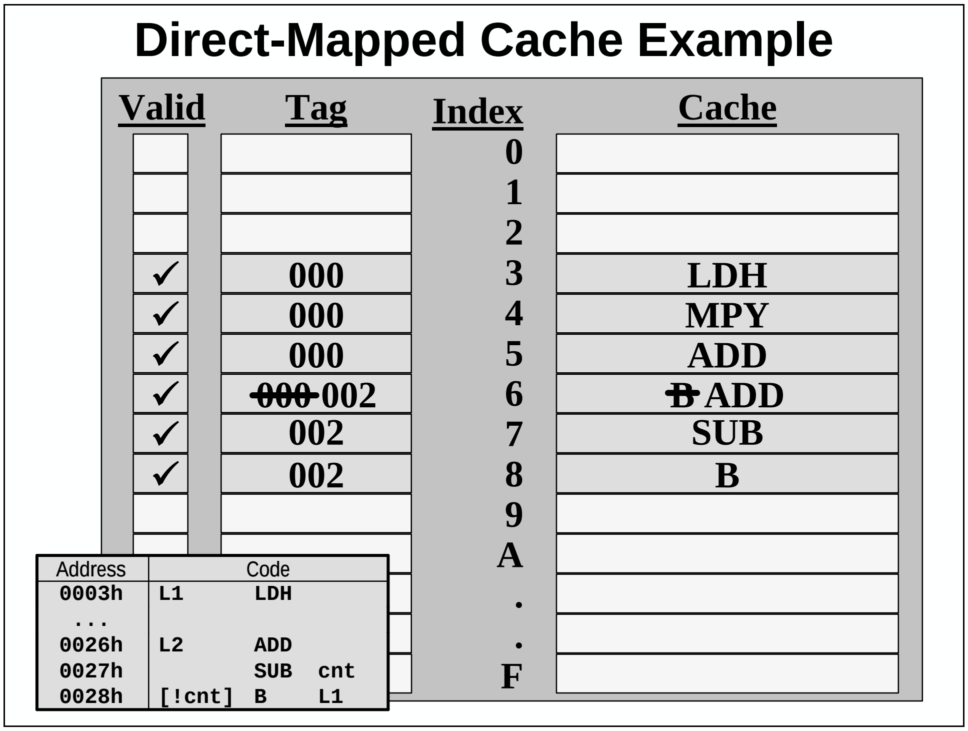
Direct Mapped Cache Example

Valid	Tag	Index	Cache
		0	
		1	
		2	
✓	000	3	LDH
✓	000	4	MPY
✓	000	5	ADD
✓	~~000~~ 002	6	~~B~~ ADD
		7	
		8	
		9	
		A	Conflict Miss
		.	
		.	
		F	

```
Address   Code
 0003h    L1      LDH
 0004h            MPY
 0005h            ADD
 0006h            B    L2


 0026h    L2      ADD
 0027h            SUB  cnt
 0028h    [!cnt]  B    L1
```

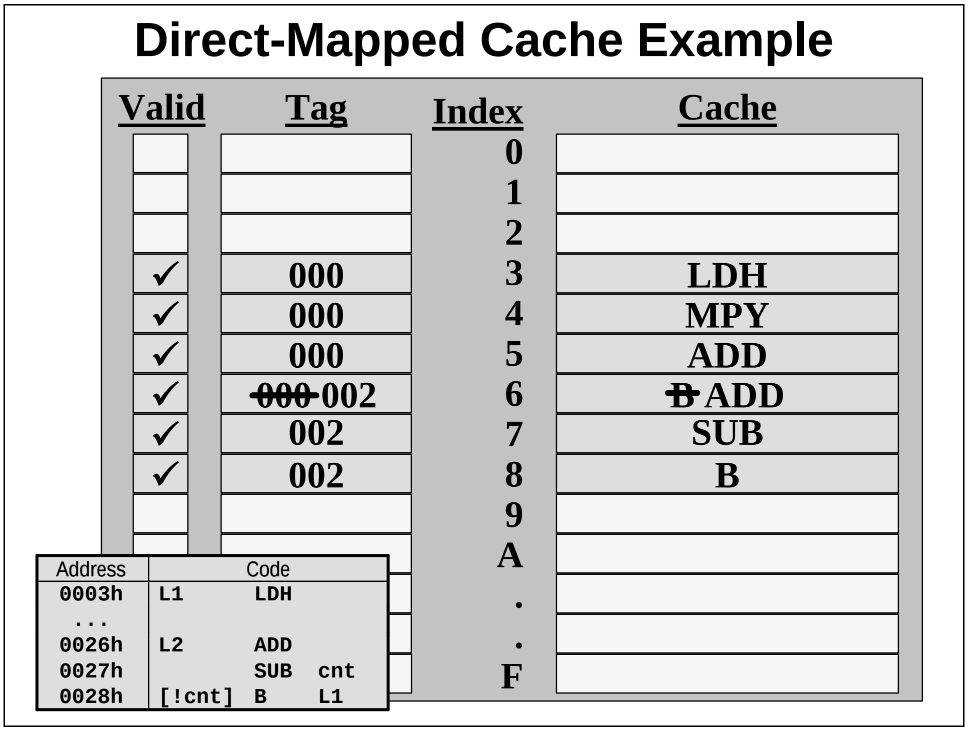
The DSP executes the instructions after the ADD, the SUB and the B. Since they are not valid in cache, they will cause cache misses.



Direct-Mapped Cache Example

Valid	Tag	Index	Cache
		0	
		1	
		2	
✓	000	3	LDH
✓	000	4	MPY
✓	000	5	ADD
✓	~~000~~ 002	6	~~B~~ ADD
✓	002	7	SUB
✓	002	8	B
		9	
		A	
		.	
		.	
		F	

```
Address   Code
0003h     L1      LDH
 ...
0026h     L2      ADD
0027h             SUB  cnt
0028h     [!cnt]  B    L1
```



Direct-Mapped Cache Example

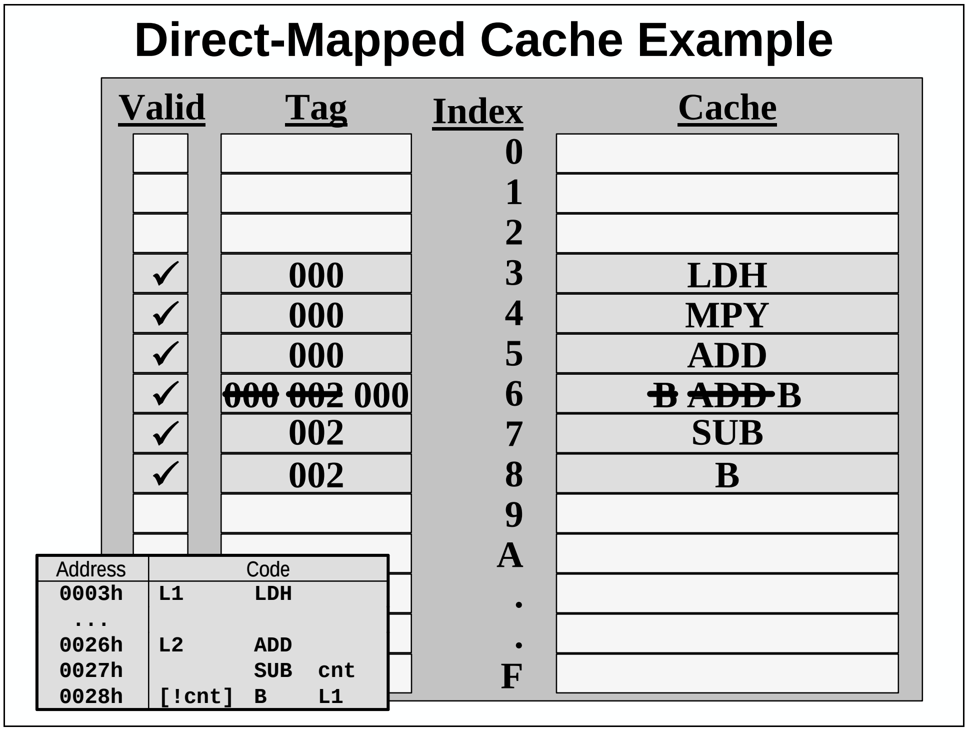
Valid	Tag	Index	Cache
		0	
		1	
		2	
✓	000	3	LDH
✓	000	4	MPY
✓	000	5	ADD
✓	~~000~~ 002	6	~~B~~ ADD
✓	002	7	SUB
✓	002	8	B
		9	
		A	
		.	
		.	
		F	

```
Address   Code
0003h     L1      LDH
 ...
0026h     L2      ADD
0027h             SUB  cnt
0028h     [!cnt]  B    L1
```

When the branch executes, it will take the DSP to a new location in memory. The branch in this case takes the DSP to the address of the symbol *lbl*, which is 0x0003. This is the address of the original LDH instruction from above.

When the DSP accesses the LDH instruction this time, it is found to be in cache. Therefore, it is given to the core without accessing memory, which removes any memory delays. This operation is called a *cache hit.*

A few observations can be made at this point. Instructions are added to cache only by accessing them. If they are only used once, the cache does not offer any benefit. However, it doesn't cause any additional delays. This type of cache has the biggest benefit for looped code, or code that is accessed over and over again. Fortunately, this is the most common type of code in DSP programming.



Direct-Mapped Cache Example

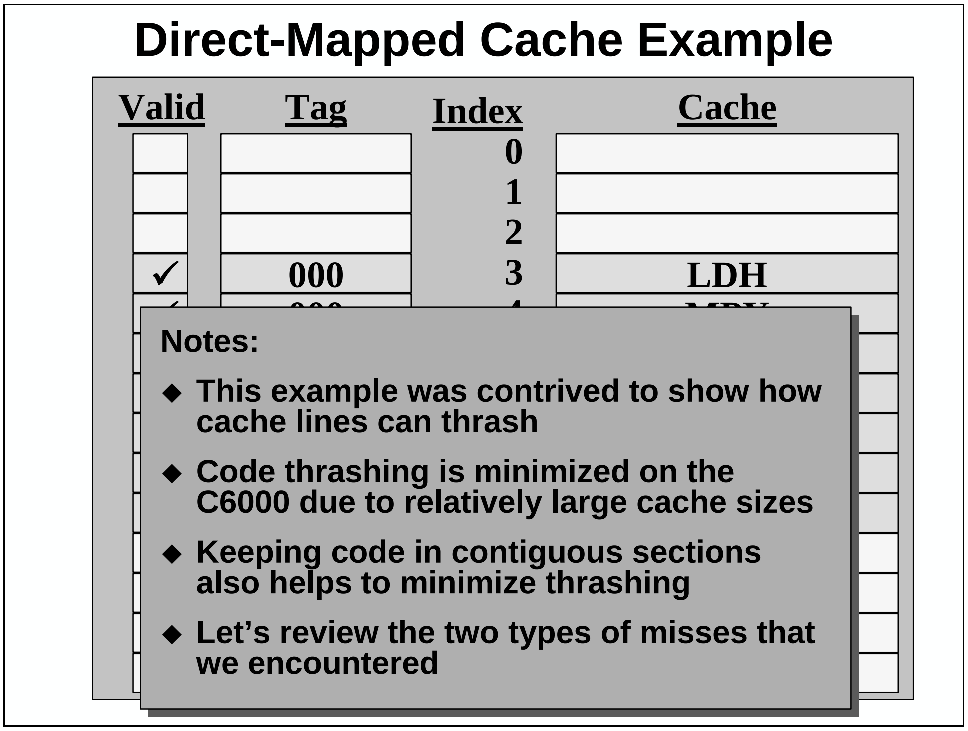
Valid	Tag	Index	Cache
		0	
		1	
		2	
✓	000	3	LDH
✓	000	4	MPY
✓	000	5	ADD
✓	~~000~~ ~~002~~ 000	6	~~B~~ ~~ADD~~ B
✓	002	7	SUB
✓	002	8	B
		9	
		A	
		.	
		.	
		F	

```
Address    Code
 0003h   L1      LDH
  ...
 0026h   L2      ADD
 0027h           SUB  cnt
 0028h   [!cnt]  B    L1
```

Notice also what seems to be happening at line 6. Each time the code runs, line 6 is overwritten twice. This behavior is called thrashing the cache. The cache misses that occur when you are thrashing the cache are called conflict misses. Why is it happening? Is it reducing the performance of the code?

Thrashing occurs when multiple elements that are executed at the same time live at the same line in the cache. Since it causes more memory accesses, it dramatically reduces the performance of the code. How can we remove thrashing from our code?

The thrashing problem is caused by the fact that the ADD and the B share the same index in memory. If they had different indexes, they would not thrash the cache. So, a simple fix to this problem is to make sure that the second piece of code (ADD, SUB, and B) doesn't share any indexes with the first chunk of code. A simple fix is to move the second chunk down by one line so that its indexes start at 7 instead of 6.



Direct-Mapped Cache Example

Valid	Tag	Index	Cache
		0	
		1	
		2	
✓	000	3	LDH
✓	000	4	MPY

Notes:

- **This example was contrived to show how cache lines can thrash**
- **Code thrashing is minimized on the C6000 due to relatively large cache sizes**
- **Keeping code in contiguous sections also helps to minimize thrashing**
- **Let's review the two types of misses that we encountered**

This relocation can be done several different ways. The simplest is probably to make the two sections contiguous in memory. Code that is contiguous and smaller than the size of the cache will not thrash because none of the indexes will overlap. Since code is placed in the same memory section a lot of the time, it will not thrash. Given the possibility of thrashing, caution should be exercised when creating different code sections in a cache based system.

Three Types of Misses

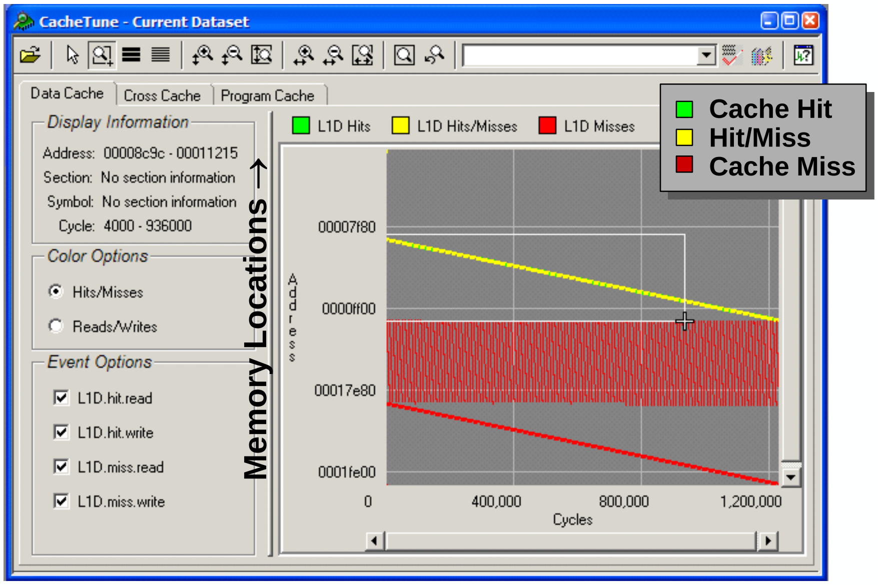
The types of misses that a cache encounters can be summarized into three different types.

Types of Misses

- **Compulsory**
  - Miss when first accessing an new address
- **Conflict**
  - Line is evicted upon access of an address whose index is already cached
  - Solutions:
    - Change memory layout
    - Allow more lines for each index
- **Capacity** (we didn't see this in our example)
  - Line is evicted before it can be re-used because capacity of the cache is exhausted
  - Solution: Increase cache size

The CacheTune tool withing CCS helps visualize different types of cache misses.

CacheTune



CacheTune - Current Dataset

Data Cache | Cross Cache | Program Cache

Display Information

Address: 00008c9c - 00011215
Section: No section information
Symbol: No section information
Cycle: 4000 - 936000

Color Options

● Hits/Misses
○ Reads/Writes

Event Options

☑ L1D.hit.read
☑ L1D.hit.write
☑ L1D.miss.read
☑ L1D.miss.write

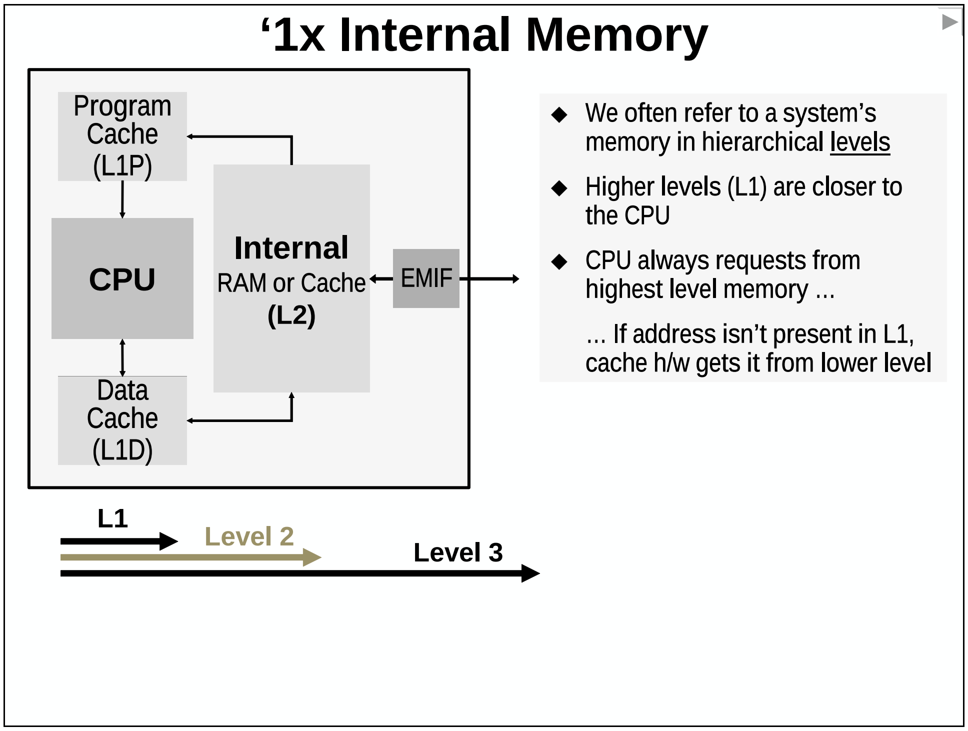
🟩 Cache Hit
🟨 Hit/Miss
🟥 Cache Miss

Memory Locations →

Time (# of instructions executed) →

C6211/C671x Internal Memory

As mentioned earlier in the workshop, the C6211/C671x devices provide three chunks of internal memory. Level 1 memories (being closest to the CPU) are provided as cache for both program (L1P) and data (L1D), respectively.



‘1x Internal Memory

- We often refer to a system’s memory in hierarchical levels
- Higher levels (L1) are closer to the CPU
- CPU always requests from highest level memory ...

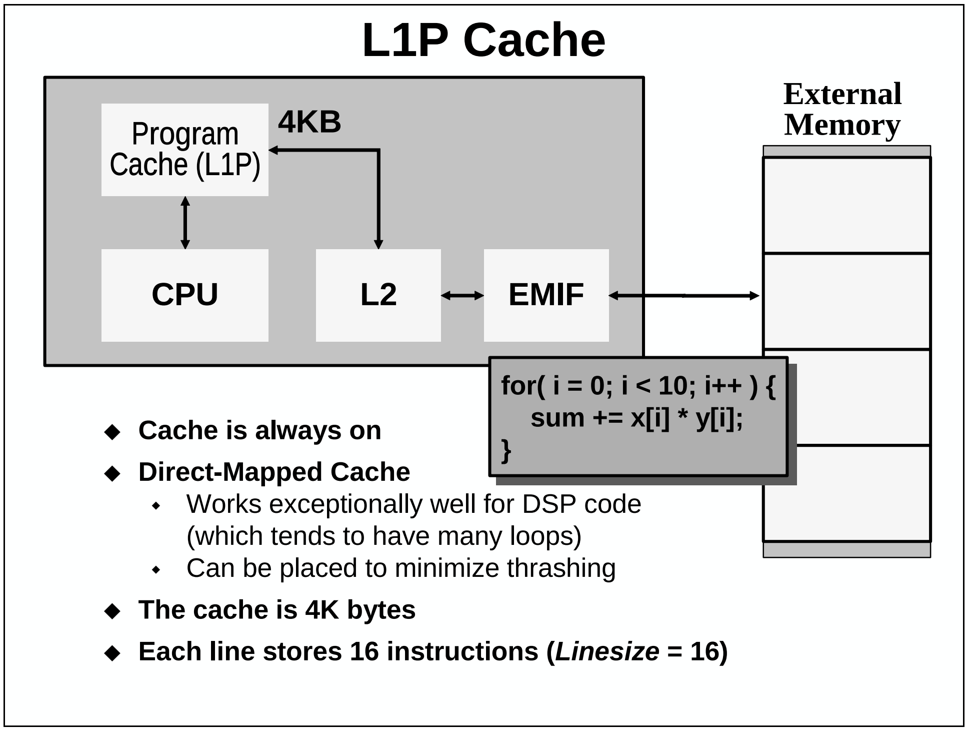
  ... If address isn’t present in L1, cache h/w gets it from lower level

The third memory chunk is called L2 memory. The processor will look for an address in L1 memories first; if not found L2 memory is examined next. L2 memory may be addressable RAM or cache – its configurability will be discussed shortly.

Finally, on these DSPs, all external memory is considered Level three memory since it is the third location examined in the memory access hierarchy. Of course, this makes sense since external accesses are slower than internal accesses.

L1 Data Cache (L1P)

The C6211/C671x devices have a direct-mapped internal program cache called L1P which is 4K bytes large. The L1 caches is always enabled.



L1P Cache

Program Cache (L1P)

4KB

CPU

L2

EMIF

External Memory

```
for( i = 0; i < 10; i++ ) {
    sum += x[i] * y[i];
}
```

- **Cache is always on**
- **Direct-Mapped Cache**
    - Works exceptionally well for DSP code (which tends to have many loops)
    - Can be placed to minimize thrashing
- **The cache is 4K bytes**
- **Each line stores 16 instructions (*Linesize* = 16)**

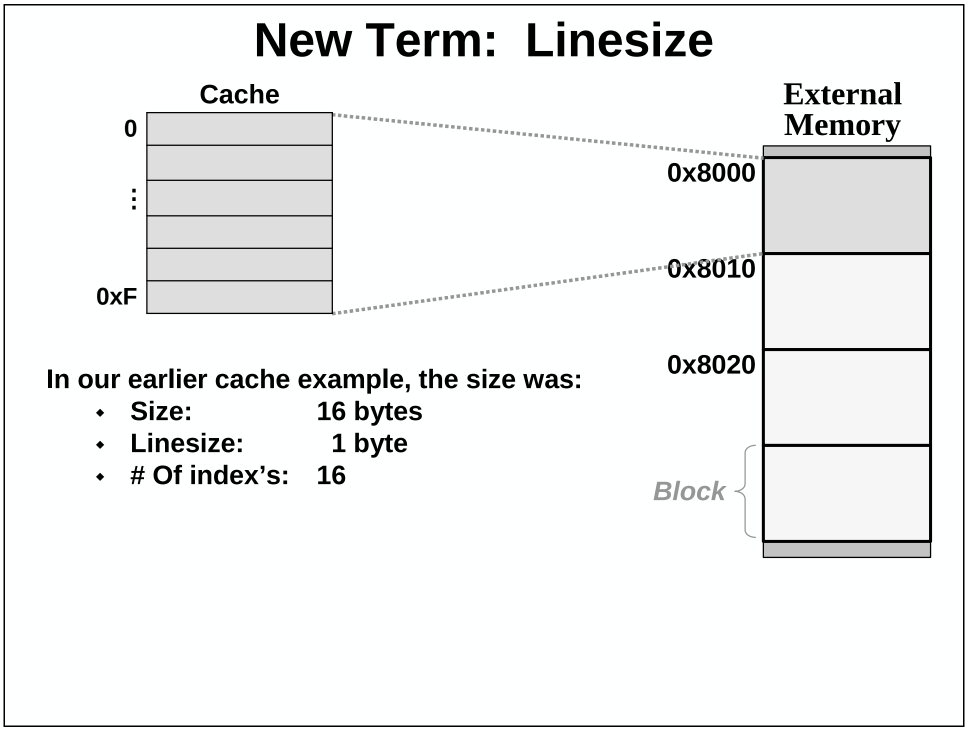
L1P has 4KB of cache broken into cache lines that store 16 instructions. So, the *linesize* of the L1P is 16 instructions. What do we mean by *linesize* …

Cache Term: Linesize

Our earlier direct-mapped cache example only stored one instruction per line; conversely the C6711 L1P cache line can hold 16 instructions. In essence, **linesize** specifies the number of addressable memory locations per line of cache.

Increasing the linesize does not change the basic concepts of cache. The cache is still organized with: blocks, lines, tags, and valid-bits. And cache accesses still result in hits and misses. What changes, though, is how much information is brought into cache when a miss occurs.

Let’s look at a simple linesize comparison. In this case, let’s look at a line that caches one byte of external memory …

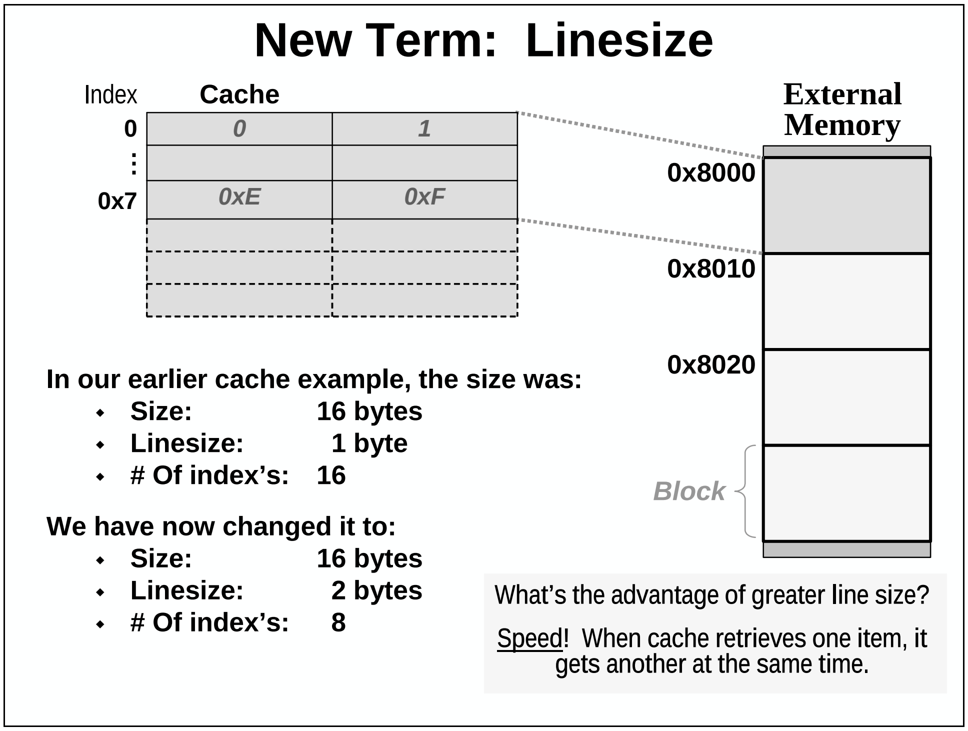


New Term: Linesize

In our earlier cache example, the size was:

- Size: 16 bytes
- Linesize: 1 byte
- # Of index’s: 16

Versus a linesize of two bytes of external memory:



New Term: Linesize

In our earlier cache example, the size was:

- **Size: 16 bytes**
- **Linesize: 1 byte**
- **# Of index's: 16**

We have now changed it to:

- **Size: 16 bytes**
- **Linesize: 2 bytes**
- **# Of index's: 8**

What's the advantage of greater line size?

Speed! When cache retrieves one item, it gets another at the same time.

Notice that the block size is consistent in both examples. Of course, when the linesize is doubled, then number of indexes is cut in half.

Increasing the linesize often may increase the performance of a system. If you are accessing information sequentially (especially common when accessing code and arrays), while the first access to a line may take the extra time required to access the addressable memory, each subsequent access to the cache line will occur at the fast cache speeds.

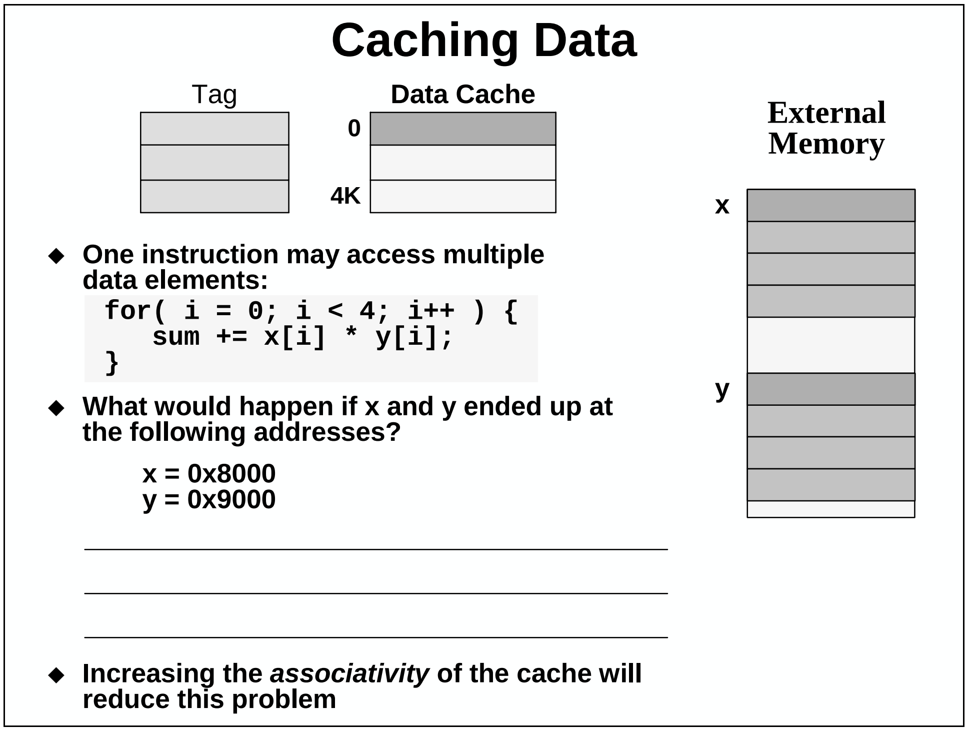
Coming back to the L1P, when a miss occurs, not only do you get one 32-bit instruction, but the cache also brings in the next 15 instructions. Thus, if your code execute sequentially, on the first pass through your code loops, you will only receive one delay every 16 instructions rather than a delay for every instruction.

A direct mapped cache is very effective for program code where a sequence of instructions is executed one after the other. This effect is maximized for looped code, where the same instructions are executed over and over again. So a direct-mapped cache works well when a single element (instruction) is being accessed at a given time and the next element is contiguous in memory.

Will a direct mapped cache work well for data?

L1 Data Cache (L1D)

The aspects that make a direct-mapped cache effective for code make it less useful for data. For example, the CPU only accesses one instruction at a time, but one instruction may access several pieces of data. Unlike code, these data elements may or may not be contiguous. If we consider a simple sum of products, the buffers themselves may be contiguous, but the individual elements are probably not. In order to avoid organizing the data so that each element is contiguous, which is difficult and confusing, a different kind of cache is needed.



Caching Data

- One instruction may access multiple data elements:

```
for( i = 0; i < 4; i++ ) {
   sum += x[i] * y[i];
}
```

- What would happen if x and y ended up at the following addresses?

  x = 0x8000
  y = 0x9000

____

____

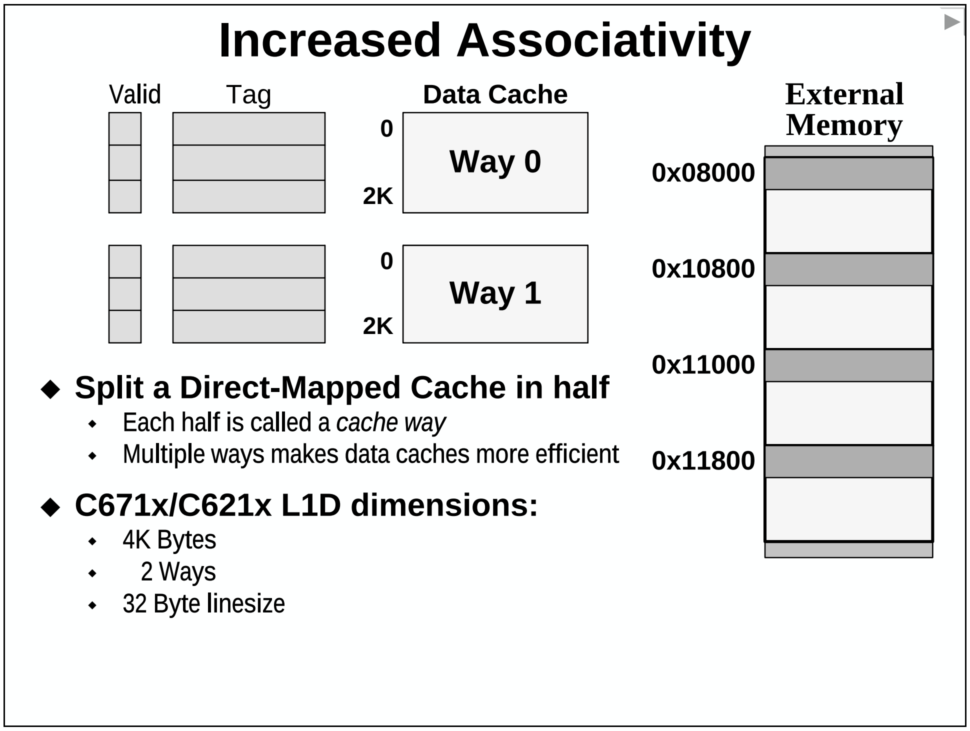
____

- Increasing the *associativity* of the cache will reduce this problem

If the addresses of X and Y both began at the start of a cache *block*, then they would end up overwriting each other in the cache, which is called *thrashing*. x_0 would go into index 0, and then y_0 would overwrite it. x_1 would be placed in index 1, and then y_1 would overwrite it. And so on.

A Way Better Cache

Since multiple data elements may be accessed by one instruction, the associativity of the cache needs to be increased. Increasing the associativity allows items from the same line of multiple blocks to live in cache at the same time. Splitting the cache in half doubles the associativity of a cache. Take the L1P as an example of a single, 4Kbyte direct-mapped cache. Splitting it in half yields two blocks of 2Kbytes each – which is how the L1D cache is configured. These two blocks are called cache **ways**. Each way has half the number of lines of the original block, but each way can store the associated line from a block. So, two cache ways means that the same line from memory can be stored in each of the two cache ways.

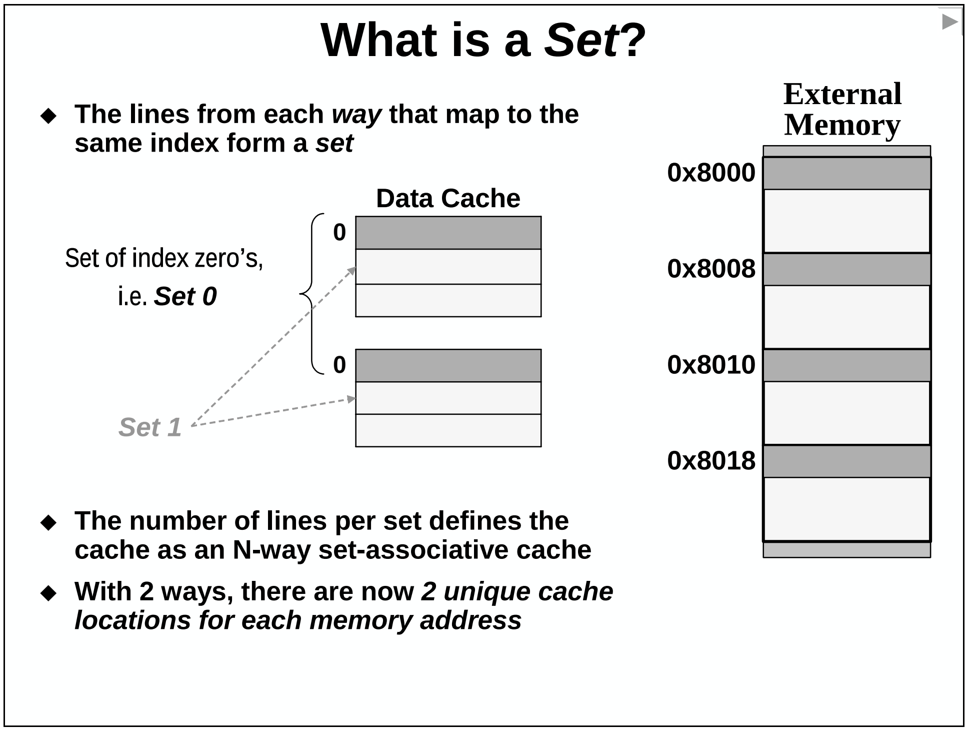


Increased Associativity

- **Split a Direct-Mapped Cache in half**
  - Each half is called a *cache way*
  - Multiple ways makes data caches more efficient
- **C671x/C621x L1D dimensions:**
  - 4K Bytes
  - 2 Ways
  - 32 Byte linesize

Cache Sets

All of the lines from the different cache ways that store the same line from memory form a **set**. For example, in a 2-way cache, the first line from each way stores the first line from each of the N blocks in memory. These two lines form a set, which is the group of lines that store the same indexes from memory. This type of cache is called a set associative-cache. So, if you have 2 cache ways, you have a 2-way set-associative cache.



What is a *Set*?

- **The lines from each *way* that map to the same index form a *set***
- **The number of lines per set defines the cache as an N-way set-associative cache**
- **With 2 ways, there are now *2 unique cache locations for each memory address***

Another way to look at this is from the address point of view. In a direct-mapped cache, each index only appears once. In an N-way set-associative cache, each index appears N times. So, N items from the same index (with the same lower address bits) can reside in the cache at the same time. In reality, a direct-mapped cache can be thought of as a 1-way set-associative cache.

Advantage of Multiple Cache Sets

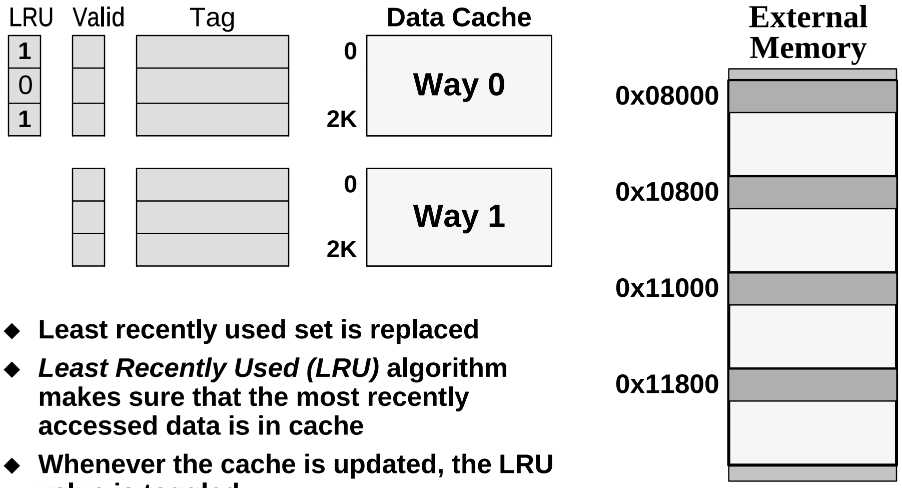
The main reason to increase the associativity of the cache, which increases the complexity of the cache controller, is to decrease the burden on the user. Without associativity, the user has to make sure that the data elements that are being accessed are contiguous. Otherwise, the cache would thrash. Consider the sum of produces example below. If the x[] and y[] arrays start at the beginning of two different blocks in memory, then each instruction will thrash. First, x[i] is brought into the cache with index 0. Then, y[i] is brought in with the same index, forcing x[i] to be overwritten. If x[] is every used again, this would dramatically decrease the performance of the cache.

Take the same example as shown with two cache ways. Now, x[i] and y[i] each have their own location in the cache, and the thrashing is eliminated. The programmer does not have to worry about where the data elements ended up in their system because the associativity allows more flexibility.

Replacing a Set (LRU)

What happens in our 2-way cache when both lines of a set have valid data and a new value with the same index (i.e. line number) needs to be cached?

What Set to Replace?



- Least recently used set is replaced
- *Least Recently Used (LRU)* algorithm makes sure that the most recently accessed data is in cache
- Whenever the cache is updated, the LRU value is toggled

The cache controller uses a **Least Recently Used** (LRU) algorithm to decide which cache way line to overwrite when a cache miss occurs. With this algorithm, the most recently *accessed* data is always stays in the cache. Note that this may or may not be the "oldest" item in the cache, rather the most recently “used”. In a 2-way set-associative cache, this algorithm can be implemented with a bit per line. The LRU algorithm maximizes the effect of temporal locality, which caches depend upon to maximize performance.

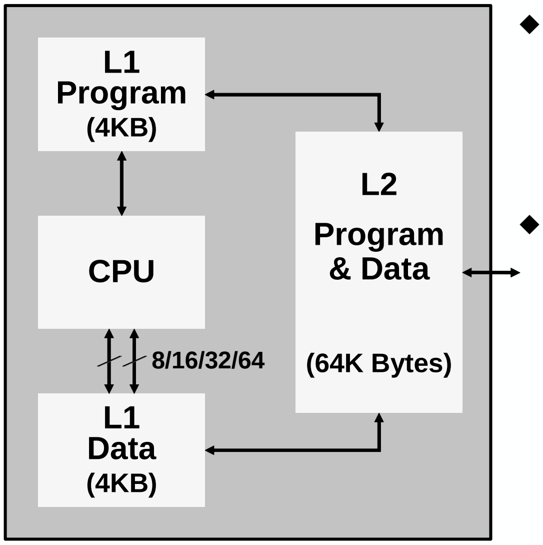
L1 Data (L1D)Cache Summary

The L1D is a 2-way set-associative data cache. On the C671x devices, it is 4K bytes large with a 32-byte linesize.

L2 Memory

The Level 2 memory (L2) is a middle hierarchical layer that helps the cache controller keep the items that the CPU will need next closer to the L1 memories. It is significantly larger (64Kbytes vs. 4Kbytes on the C6711) to help store larger arrays/functions and keep them closer to the CPU. It is a unified memory, meaning that it can store both code and data.

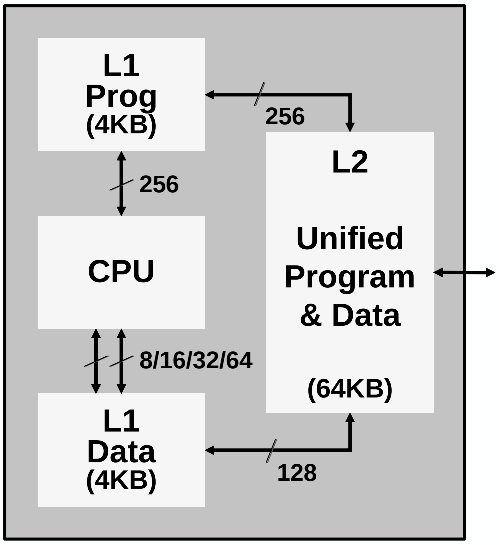
'11 / '12 Internal Memory



- **Level 1 Caches**
  - **Single-cycle access**
  - **Always enabled**
  - **L2 accessed on miss**
- **Level 2 Memory**
  - **Unified: Prog or Data**
  - **L2 → L1D delivers 32-bytes in 4 cycles**
  - **L2 → L1P delivers 16 instr's in 5 cycles**
  - **Configure L2 as cache or addressable RAM**

(C6713: L2 memory is 256K bytes)

'11/'12 Internal Memory -- Details



Level 1 Program
- **Always cache**
- **1 way cache (Direct-Mapped)**
- **Zero wait-state**
- **Line size: 512 bits (or 16 instr)**

Level 1 Data
- **Always cache**
- **2 way cache**
- **Zero wait-state**
- **Line size: 256 bits**

Level 2
- **Unified (prog or data)**
- **RAM or cache**
- **1-4 way cache**
- **32 data bytes in 4 cycles**
- **16 instr. in 5 cycles**
- **Line Size: 1024 bits (or 128 bytes)**

(C6713: L2 memory is 256K bytes)

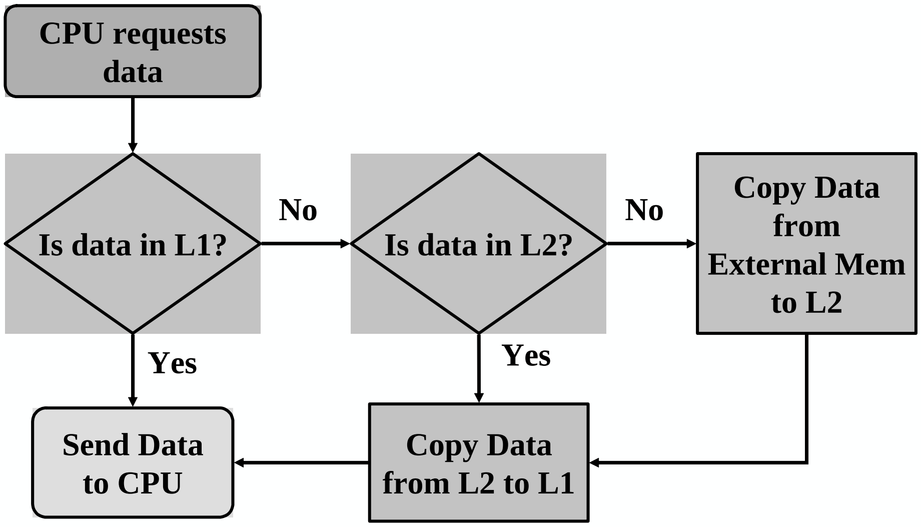
Memory Hierarchies further explained

The 'C6x11 uses a Memory Hierarchy to maximize the effectiveness of its on and off chip memories. This hierarchy uses small, fast memories close to core for performance and large, slow memories off-chip for storage. The cache controller is optimized to keep instructions and data that the core needs in the faster memories automatically with minimal effect on the system design. Large off-chip memories can be used to store large buffers without having to pay for larger memories on-chip, which can be expensive.

- **A Memory Hierarchy organizes memory into different levels**
  - **Higher Levels are closer to the CPU**
  - **Lower Levels are further away**
- **CPU requests are sent from higher levels to lower levels**
- **The higher levels are designed to keep information that the CPU needs based on:**
  - **Temporal Locality – most recently accessed**
  - **Spatial Locality – closest in memory**
- **Middle levels can buffer between small-fast memory and large-slow memory**

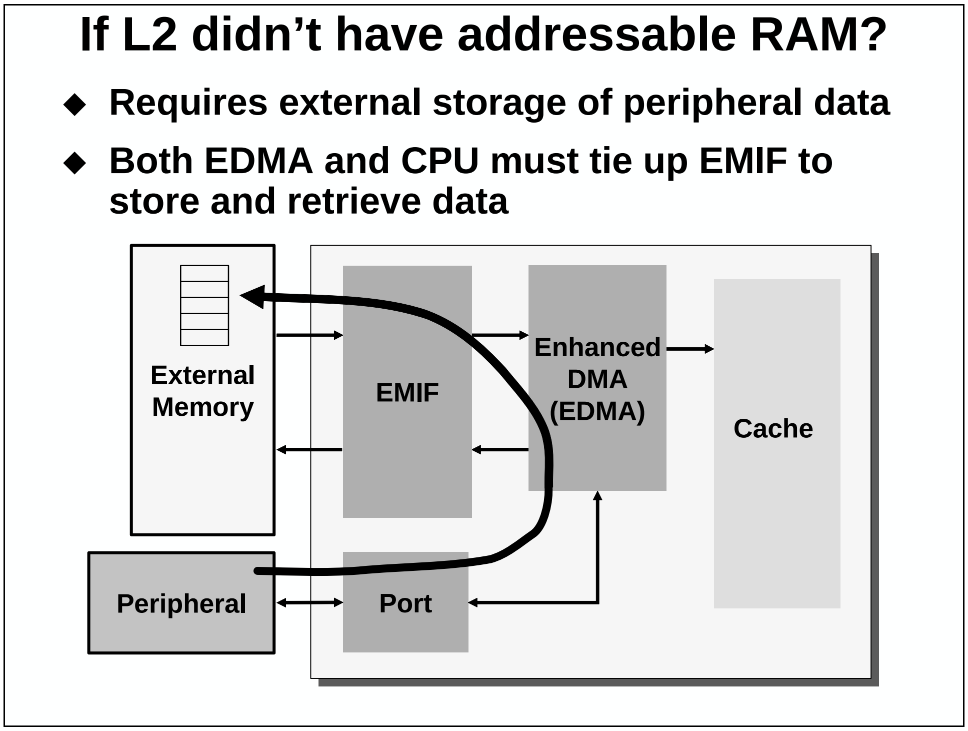
The L1P and L1D are the 'C6x11's highest order memories in the hierarchy. As you move further away from these memories, performance decreases. CPU requests are first sent to these fast memories, then to slower memories lower in the hierarchy. The highest orders are designed to store the information that the CPU needs based on temporal and spatial locality. Intermediate levels can be inserted between the highest order (L1P and L1D) and the lowest order (external memory) to serve as a larger buffer that further increases performance of the memory system. Again, L2 is a middle hierarchical layer that helps the cache controller keep the items that the CPU will need next closer to the L1 memories.

Here is a simple flow chart of the decision process that the cache controller uses to fulfill CPU requests.


CPU requests data
Is data in L1?
No
Is data in L2?
No
Copy Data from External Mem to L2
Yes
Yes
Send Data to CPU
Copy Data from L2 to L1

Why both RAM and Cache in L2?

Why would a designer choose to configure the L2 memory as RAM instead of cache? Consider a system that uses the EDMA to transfer data from a serial port. If there is no internal memory, this data has to be written into external memory. Then, when the CPU accesses the data, it will be brought in to L2 (and L1) by the cache controller. Does this seem inefficient?

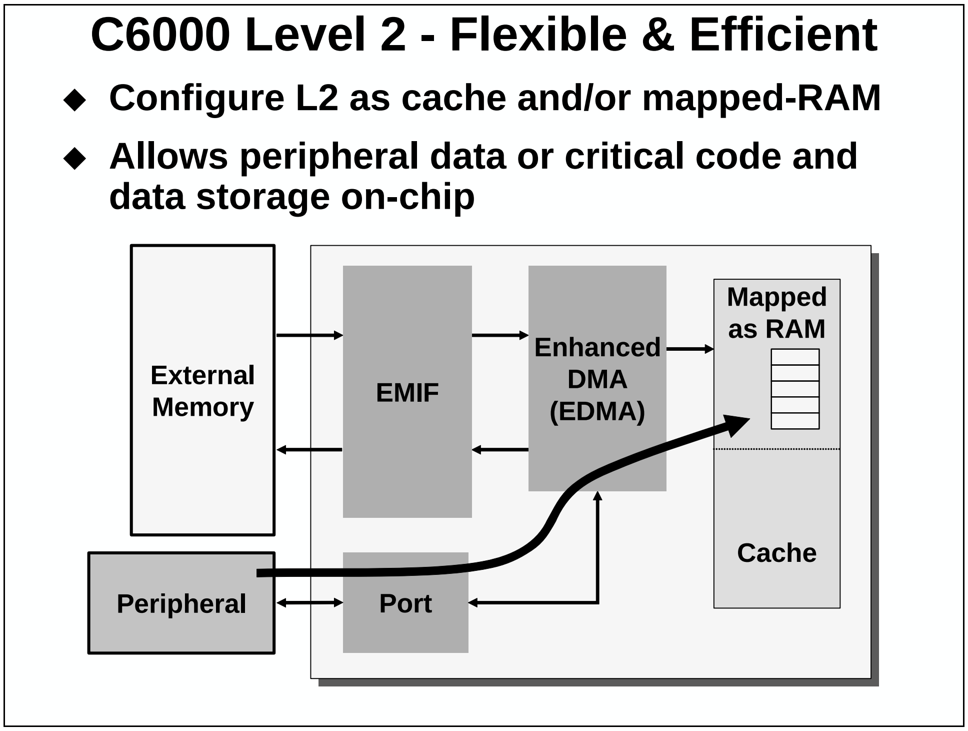


If L2 didn’t have addressable RAM?

- **Requires external storage of peripheral data**
- **Both EDMA and CPU must tie up EMIF to store and retrieve data**

If you use the DMA to read from on-chip peripherals – such as the McBSP – you might prefer to use part of the L2 memory as memory-mapped RAM. This setup allows you to store incoming data on-chip, rather than having to move it to off-chip, cache it on-chip, and then move it back off-chip to send it out to the external world.

The configurability of the L2 memory as RAM or cache allows designers to maximize the efficiency of their system.

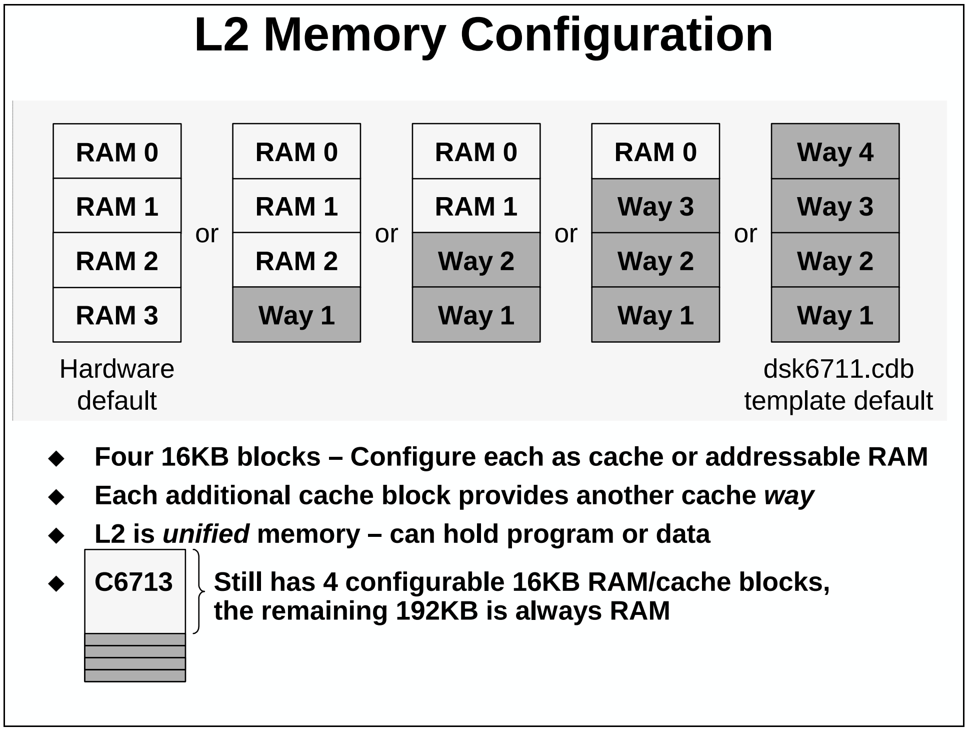


C6000 Level 2 - Flexible & Efficient

- Configure L2 as cache and/or mapped-RAM
- Allows peripheral data or critical code and data storage on-chip

L2 Configuration

The L2 memory is configurable to allow for a mix of RAM blocks and cache ways. The 64KB is divided into four chunks, each of which can either be RAM memory or a cache way. This allows the designer to set some on-chip memory aside for dedicated buffers, and to use the other memory as cache ways.



L2 Memory Configuration

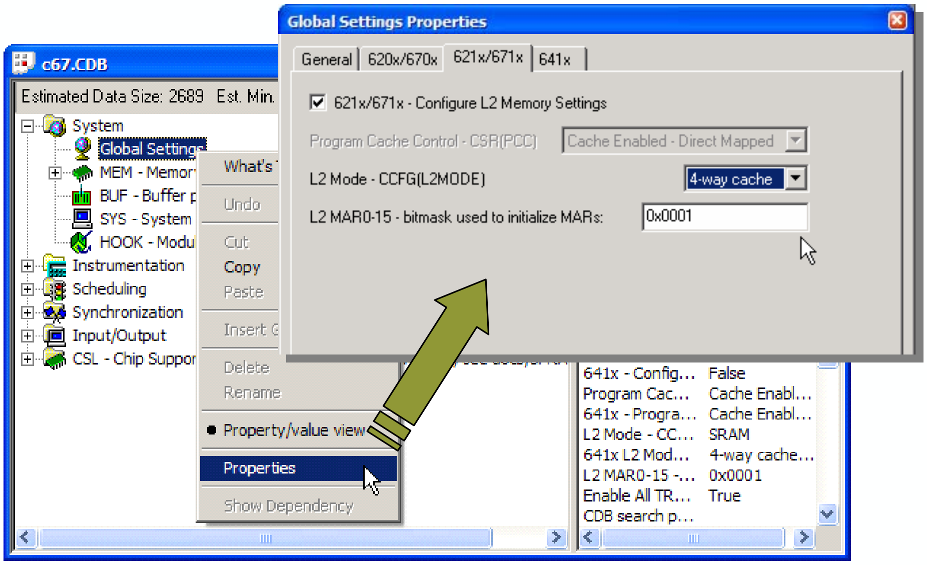
RAM 0		RAM 0		RAM 0		RAM 0		Way 4
RAM 1	or	RAM 1	or	RAM 1	or	Way 3	or	Way 3
RAM 2		RAM 2		Way 2		Way 2		Way 2
RAM 3		Way 1		Way 1		Way 1		Way 1
Hardware default								dsk6711.cdb template default

- **Four 16KB blocks – Configure each as cache or addressable RAM**
- **Each additional cache block provides another cache *way***
- **L2 is *unified* memory – can hold program or data**
- **C6713** **Still has 4 configurable 16KB RAM/cache blocks, the remaining 192KB is always RAM**

The L2 can be changed during run time. So, a designer could choose to change a RAM block to cache or vice versa. Before making a switch from RAM to cache, the user should make sure the any information needed by the system that is currently in the RAM block is copied somewhere else. This copy can be done with the DMA to minimize the overhead on the CPU. Before switching a cache way to RAM, the cache should be free of any dirty data. Dirty data is data that has been written by the CPU but may not have been copied out to memory.

The L2 can be configured at initialization using the configuration tool.

Configuring L2 Cache with CDB



☑ 621x/671x - Configure L2 Memory Settings

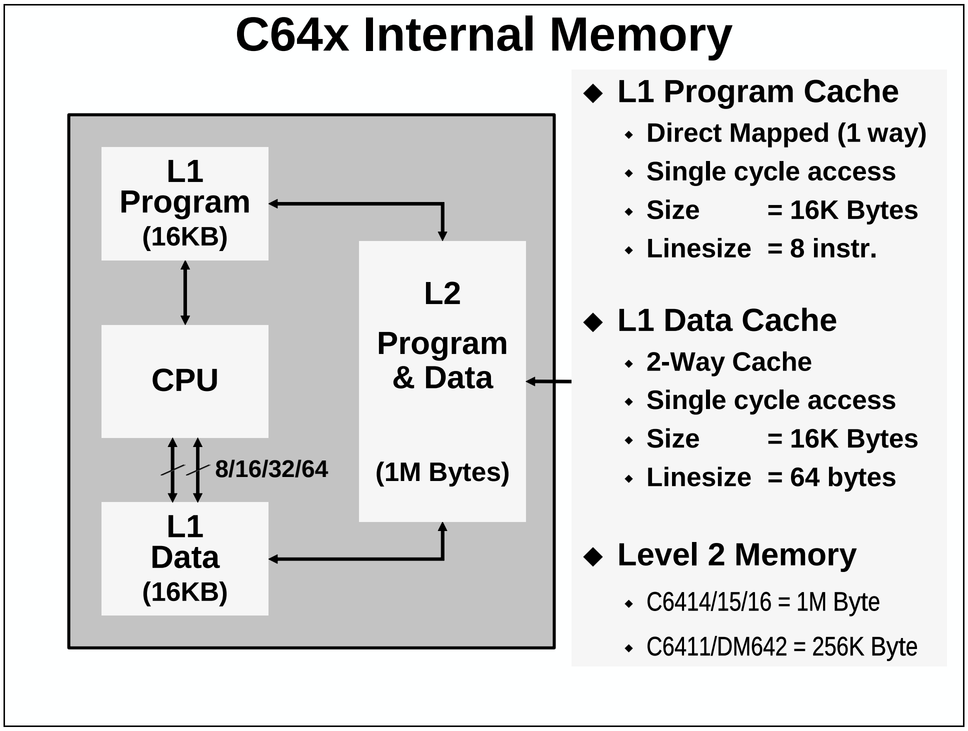
Program Cache Control - CSR(PCC): Cache Enabled - Direct Mapped

L2 Mode - CCFG(L2MODE): 4-way cache

L2 MAR0-15 - bitmask used to initialize MARs: 0x0001

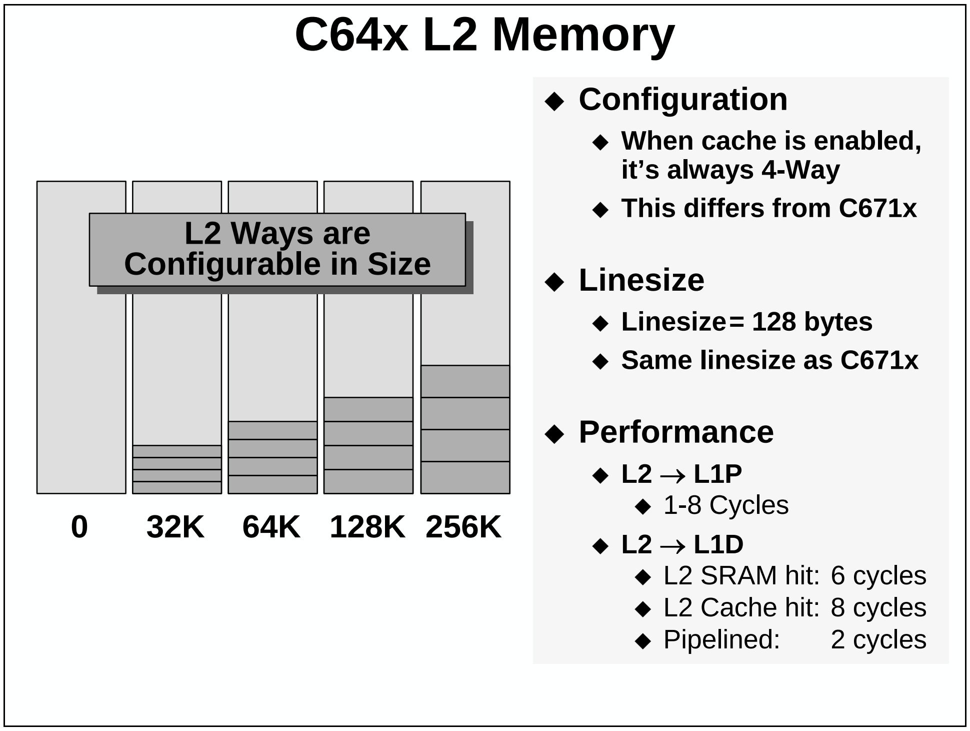
641x - Config…	False
Program Cac…	Cache Enabl…
641x - Progra…	Cache Enabl…
L2 Mode - CC…	SRAM
641x L2 Mod…	4-way cache…
L2 MAR0-15 -…	0x0001
Enable All TR…	True
CDB search p…	

C64x Internal Memory Overview



C64x Internal Memory

- **L1 Program Cache**
  - **Direct Mapped (1 way)**
  - **Single cycle access**
  - **Size = 16K Bytes**
  - **Linesize = 8 instr.**
- **L1 Data Cache**
  - **2-Way Cache**
  - **Single cycle access**
  - **Size = 16K Bytes**
  - **Linesize = 64 bytes**
- **Level 2 Memory**
  - C6414/15/16 = 1M Byte
  - C6411/DM642 = 256K Byte



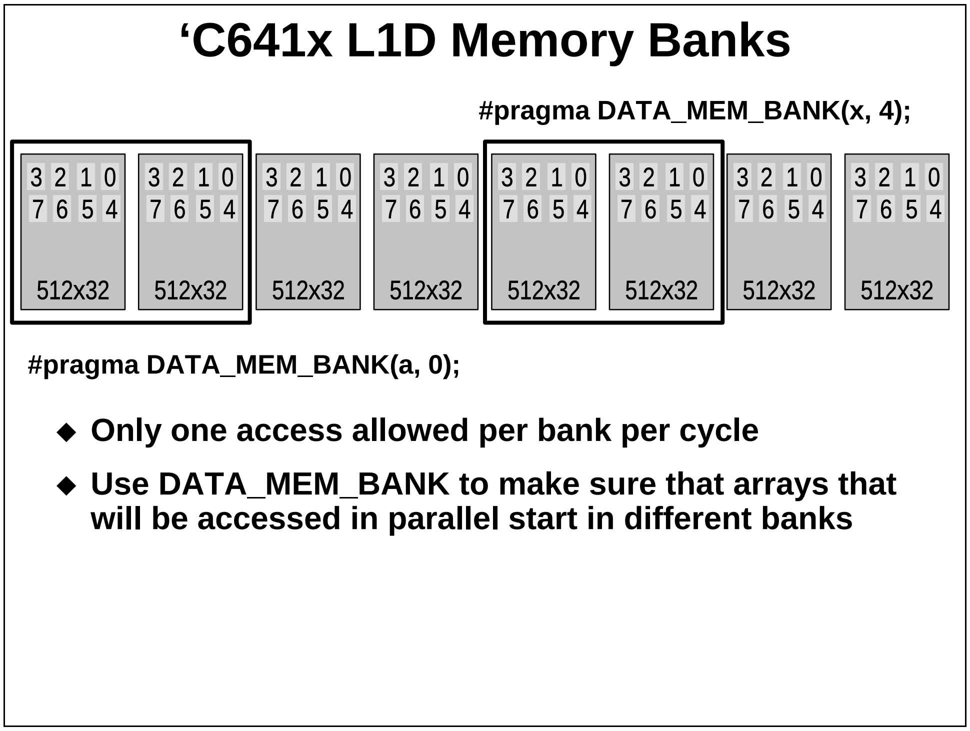
C64x L2 Memory

- **Configuration**
  - **When cache is enabled, it's always 4-Way**
  - **This differs from C671x**
- **Linesize**
  - **Linesize= 128 bytes**
  - **Same linesize as C671x**
- **Performance**
  - **L2 → L1P**
    - 1-8 Cycles
  - **L2 → L1D**
    - L2 SRAM hit: 6 cycles
    - L2 Cache hit: 8 cycles
    - Pipelined: 2 cycles

Additional Memory/Cache Topics

'C64x Memory Banks

The 'C64x also uses a memory banking scheme to organize L1. Each bank is 32 bits wide, containing four byte addresses. Eight banks are interleaved so that the addresses move from 1 bank to the next. The basic rule is that you can access each bank once per cycle, but if you try to access a bank twice in a given cycle you will encounter a memory bank stall. So, when creating arrays that you plan to access with parallel load instructions, you need to make sure that the arrays start in different banks. The DATA_MEM_BANK() pragma helps you create the arrays so that they start in different memory banks.



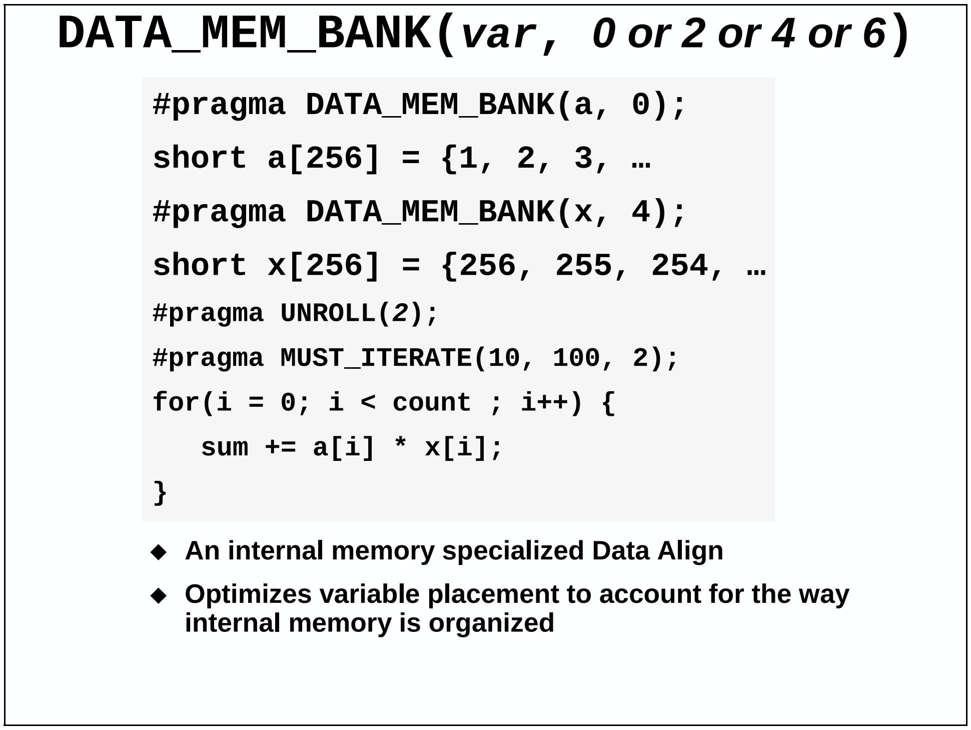
‘C641x L1D Memory Banks

#pragma DATA_MEM_BANK(x, 4);

#pragma DATA_MEM_BANK(a, 0);

- Only one access allowed per bank per cycle
- Use DATA_MEM_BANK to make sure that arrays that will be accessed in parallel start in different banks

Sometimes variables need to be aligned to account for the way that memory is organized. The DATA_MEM_BANK is a specialized data align type #pragma that does exactly this.



DATA_MEM_BANK(*var, 0 or 2 or 4 or 6*)

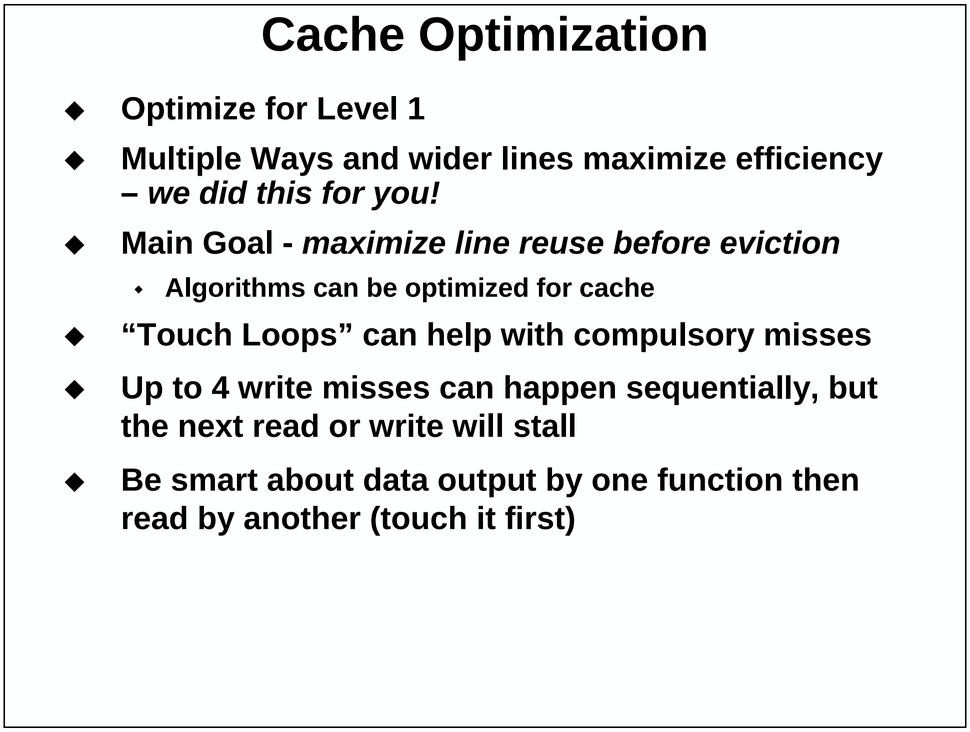
```
#pragma DATA_MEM_BANK(a, 0);
short a[256] = {1, 2, 3, …
#pragma DATA_MEM_BANK(x, 4);
short x[256] = {256, 255, 254, …
#pragma UNROLL(2);
#pragma MUST_ITERATE(10, 100, 2);
for(i = 0; i < count ; i++) {
   sum += a[i] * x[i];
}
```

- **An internal memory specialized Data Align**
- **Optimizes variable placement to account for the way internal memory is organized**

Unlike some of the other pragma’s discussed in this chapter, the DATA_ALIGN pragma does not have to be used directly before the definition of the variable it aligns. Most users, though, prefer to keep them together to ease in code maintenance.

Cache Optimization

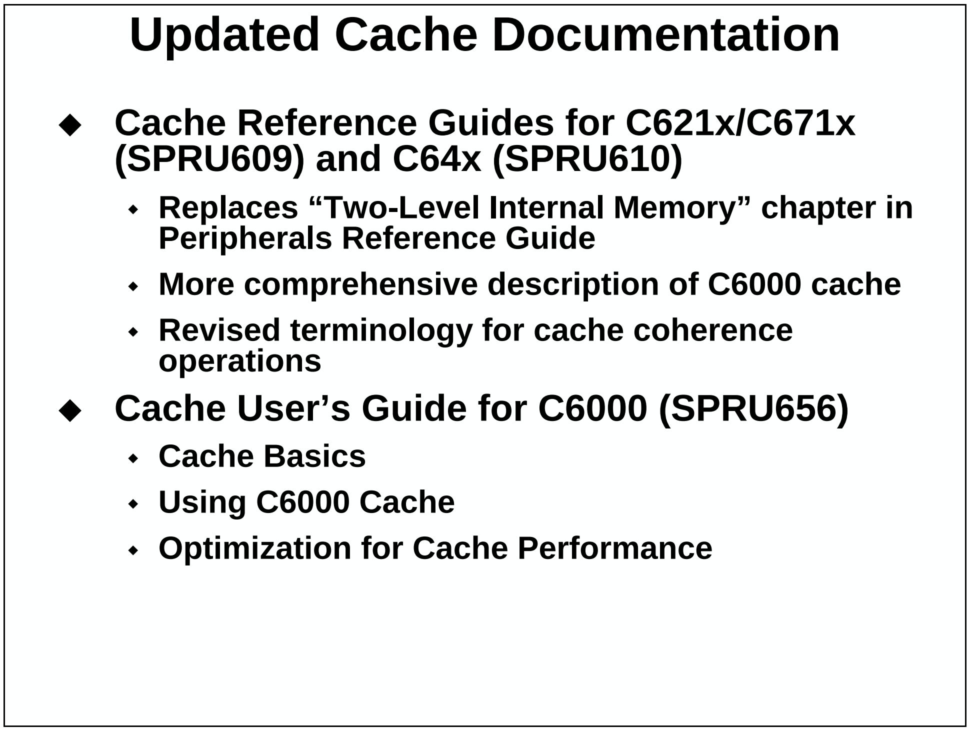
Here are some great ideas for how to optimize cache.



Cache Optimization

- Optimize for Level 1
- Multiple Ways and wider lines maximize efficiency – *we did this for you!*
- Main Goal - *maximize line reuse before eviction*
  - Algorithms can be optimized for cache
- "Touch Loops" can help with compulsory misses
- Up to 4 write misses can happen sequentially, but the next read or write will stall
- Be smart about data output by one function then read by another (touch it first)

Each one of these subjects deserves to be treated with enough material to fill a chapter in a book. In fact, a book has been written to cover these subjects.



Updated Cache Documentation

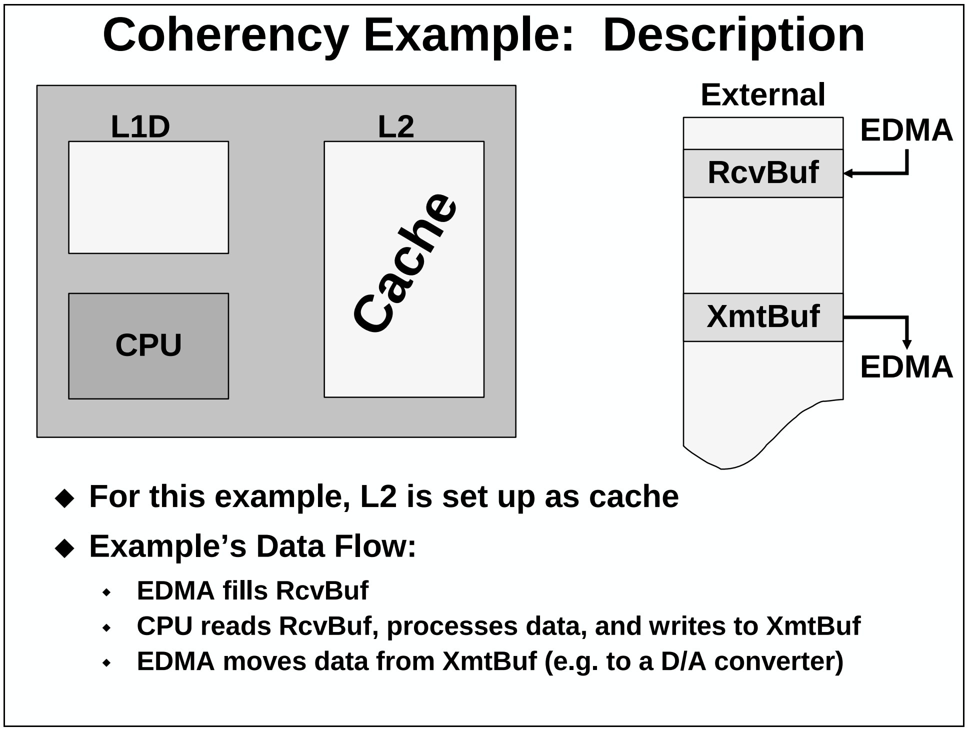
- Cache Reference Guides for C621x/C671x (SPRU609) and C64x (SPRU610)
  - Replaces "Two-Level Internal Memory" chapter in Peripherals Reference Guide
  - More comprehensive description of C6000 cache
  - Revised terminology for cache coherence operations
- Cache User's Guide for C6000 (SPRU656)
  - Cache Basics
  - Using C6000 Cache
  - Optimization for Cache Performance

Data Cache Coherency

One issue that can arise with caching architectures is called coherency. The basic idea behind coherency is that the information in the cache should be the same as the information that is stored at the memory address for that information. As long as the CPU is the only piece of the system that modifies information, and the system does not use self-modifying code, coherency will always be maintained. Ignoring the self-modifying code issue, is there anything else in the system that modifies memory?

Example Problem

Let's look at an example that will highlight coherency issues and provide some solutions.



Coherency Example: Description

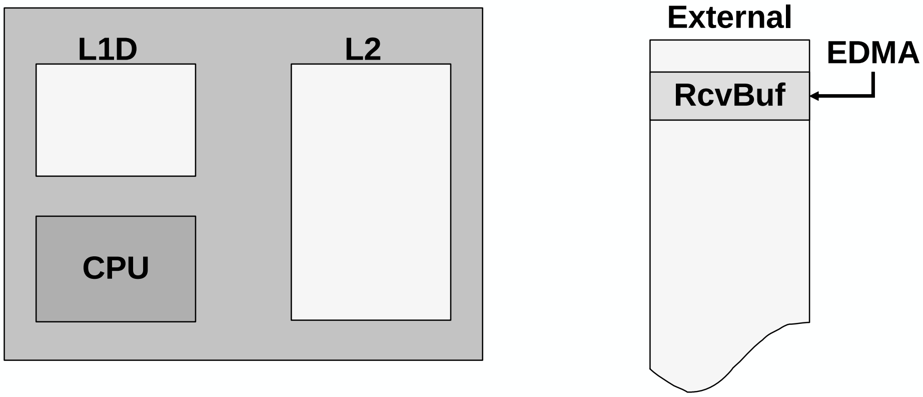
- **For this example, L2 is set up as cache**
- **Example's Data Flow:**
  - **EDMA fills RcvBuf**
  - **CPU reads RcvBuf, processes data, and writes to XmtBuf**
  - **EDMA moves data from XmtBuf (e.g. to a D/A converter)**

In this example, the coherency between the L1, L2, and external memories is considered. This example only deals with data.

An important consideration in 'C6x11 based systems is the effect of the EDMA. The EDMA can modify (read/write) information. The CPU does not know about the EDMA modifying memory locations. The CPU and the DMA can be viewed as two co-processors (which is what they really are) that are aware of each other, but don't know exactly what the other is doing.

Look at the diagram below. This system is supposed to receive buffers from the EDMA, process them, and send them out via the EDMA. When the EDMA finishes receiving a buffer, it interrupts the CPU to transfer ownership of the buffer from the EDMA to the CPU.

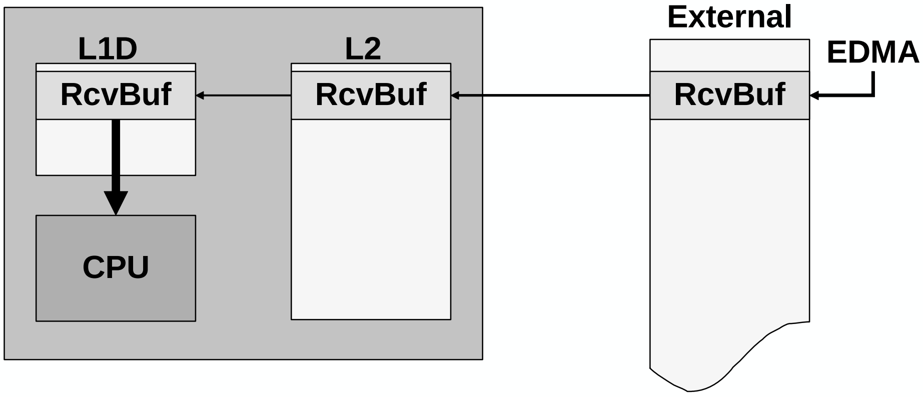
EDMA Writes Buffer



- Buffer (in external memory) written by the EDMA

In order to process the buffers, the CPU first has to read them. The first time the buffer is accessed, it is not in either of the caches, L1 or L2. When the buffer is read, the data is brought in to both of the caches. At this point, all three of the buffers (L1, L2, and External) are coherent.

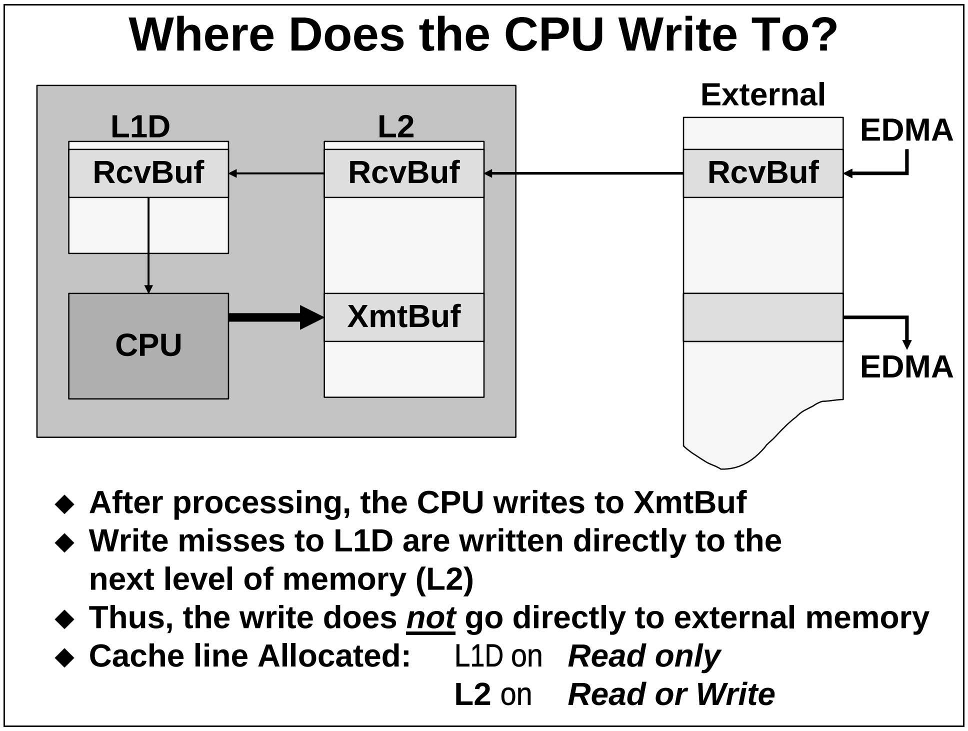
CPU Reading Buffers



- CPU reads the buffer for processing
- This read causes a cache miss in L1D and L2
- <u>RcvBuf is added</u> to <u>both</u> caches
  - Space is allocated in each cache
  - RcvBuf data is copied to both caches

When the CPU is finished processing the buffer, it writes the results to a transmit buffer. This buffer is located out in external memory. When the buffer is written, since it does not currently reside in L1D, a write miss occurs. This write miss causes the transmit buffer to be written to the next lower level of memory, L2 in this case. The reason for this is that L1D does NOT allocate space for write misses. Usually DSPs do a lot more reading than they do writing, so the effect of this is to allow more read misses to live in cache.

The net effect is that the transmit buffer gets written to L2.

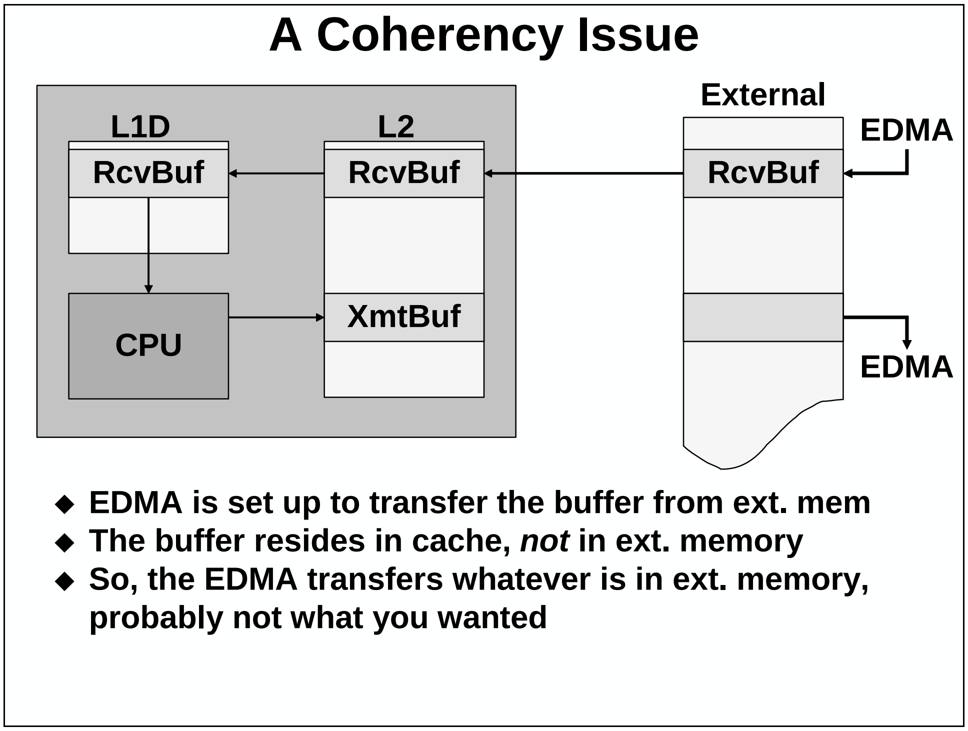


Where Does the CPU Write To?

- **After processing, the CPU writes to XmtBuf**
- **Write misses to L1D are written directly to the next level of memory (L2)**
- **Thus, the write does *not* go directly to external memory**
- **Cache line Allocated:** L1D on ***Read only***
  L2 on ***Read or Write***

Remember that the EDMA is going to be used to send the buffer out to the real world. So, where does it start reading the buffer from? That's right, external memory. Don't forget that caches do not have addresses. The EDMA requires an address for the source and destination of the transfer. The EDMA can't transfer from cache, so the buffer has to get from cache to external memory at the correct time.

Since the cached value which was written by the CPU is different from the value stored in external memory, the cache is said to be incoherent.



A Coherency Issue

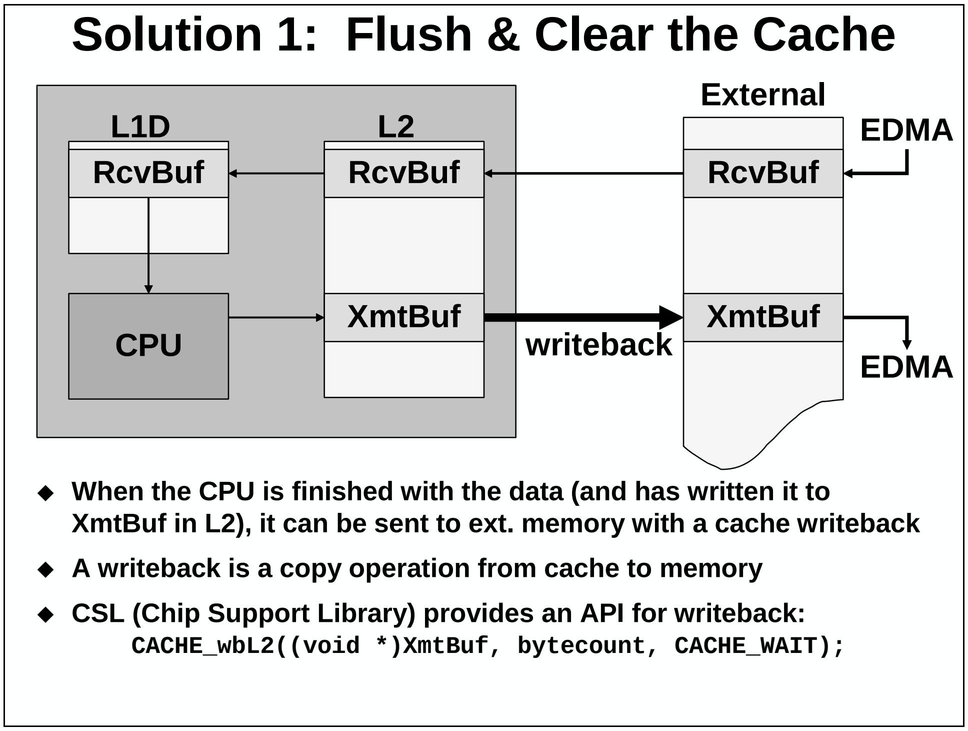
- **EDMA is set up to transfer the buffer from ext. mem**
- **The buffer resides in cache, *not* in ext. memory**
- **So, the EDMA transfers whatever is in ext. memory, probably not what you wanted**

If coherency is not maintained (by sending the new cache values out to external memory), then the EDMA will send whatever is at the address that it was told to use. The best case is that this memory has been initialized with something that won't cause the system to break. The worst case is that the EDMA sends garbage data that may disrupt the rest of the system. Either way, the system is not doing what we wanted it to do.

Solution 1: Using Cache Flush & Clean

A solution to this problem is to tell the cache controller to send out anything that it has stored at the address of the transmit buffer. This can be done with a cache writeback operation. A cache writeback sends anything that is in cache out to its address in external memory. Does a writeback need to send all of the data? No, it only needs to send the information that has been modified by the CPU, which is referred to as dirty. In the case of the transmit buffer, all of the information was written by the CPU, so it is all dirty and it will all be sent to external memory by a writeback.

So, when the CPU is finished with the data, performing a writeback of the entire buffer will force the information out to its real address so that the EDMA can read it. Another way to think of a writeback is a copy of dirty data from cache to its memory location.

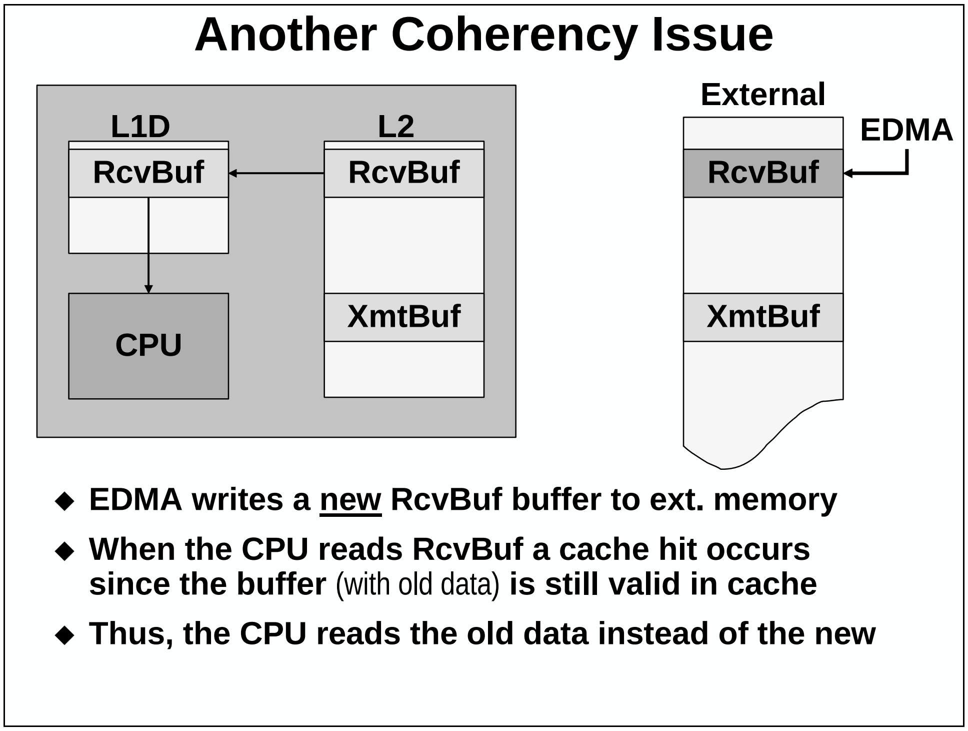


Solution 1: Flush & Clear the Cache

- When the CPU is finished with the data (and has written it to XmtBuf in L2), it can be sent to ext. memory with a cache writeback
- A writeback is a copy operation from cache to memory
- CSL (Chip Support Library) provides an API for writeback:

```
    CACHE_wbL2((void *)XmtBuf, bytecount, CACHE_WAIT);
```

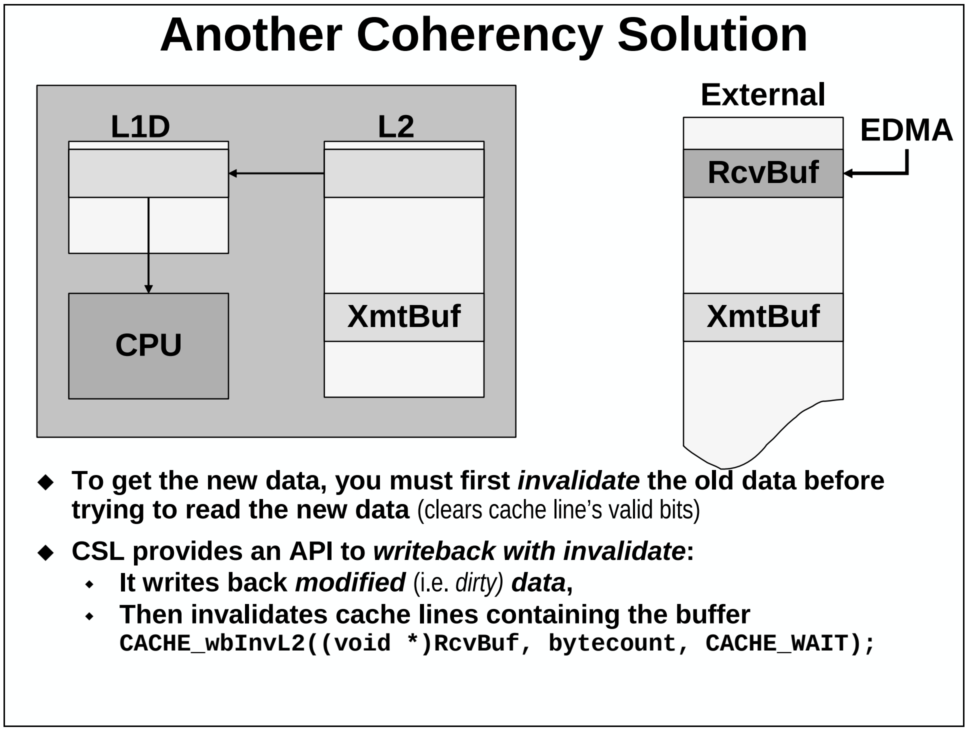
Now that we know how to get the transmit buffers to their memory addresses to solve the coherency issue, let's consider another case on the read side. What happens if the EDMA writes new data to the receive buffer. The CPU needs to process this new data and send it out, just like before. However, this situation is different because the addresses for the receive buffer are already in the cache. So, when the CPU reads the buffer, it will read the cached values (i.e. the old values) and not the new values that the EDMA just wrote.



Another Coherency Issue

- **EDMA writes a <u>new</u> RcvBuf buffer to ext. memory**
- **When the CPU reads RcvBuf a cache hit occurs since the buffer** (with old data) **is still valid in cache**
- **Thus, the CPU reads the old data instead of the new**

In order to solve this problem, we need to force the CPU to read the external memory instead of the cache. This can be done with a cache invalidate. An *invalidate* invalidates all of the lines by setting the valid bit of each line of cache to 0 or false.



Another Coherency Solution

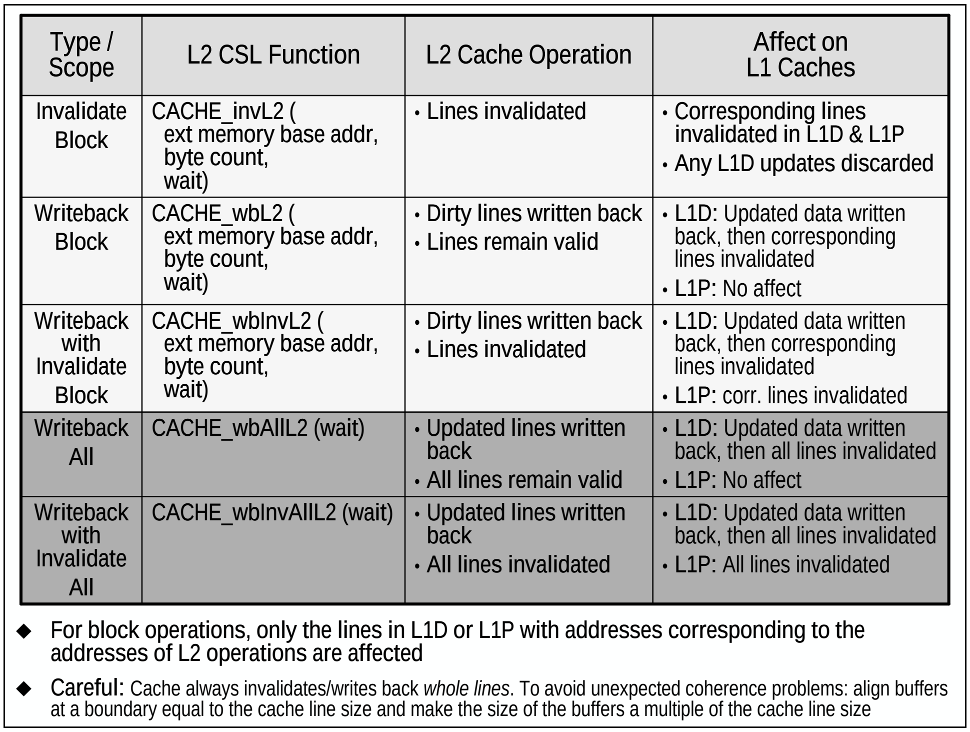
- **To get the new data, you must first *invalidate* the old data before trying to read the new data** (clears cache line's valid bits)
- **CSL provides an API to *writeback with invalidate*:**
  - **It writes back *modified*** (i.e. *dirty)* ***data*,**
  - **Then invalidates cache lines containing the buffer**

```
CACHE_wbInvL2((void *)RcvBuf, bytecount, CACHE_WAIT);
```

The C621x/C671x processors only have a writeback-invalidate operation on L2. They cannot do an invalidate by itself. A couple of things need to be considered before performing the cache writeback-invalidate. Since the writeback-invalidate performs a writeback of the data on L2, any modified or dirty data will be sent out to external memory. So, the writeback-invalidate must be done while the CPU owns the buffer. Otherwise, the old modified values could overwrite the new values from the EDMA. Also, a writeback-invalidate should only be performed after the CPU has finished modifying the buffer. If the writeback-invalidate is performed before the CPU is finished with the data, it will be brought back in, negating the effect of the writeback-invalidate.

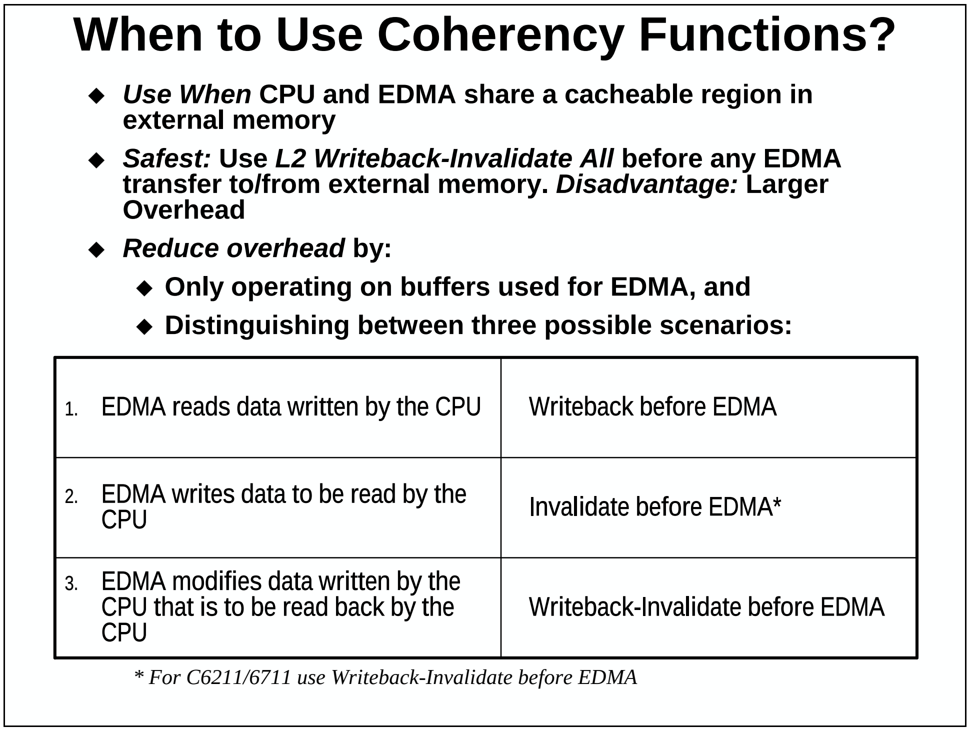
Cache Coherency Summary

The tables below list the different situations that may cause coherency issues and their possible solutions:



Type / Scope	L2 CSL Function	L2 Cache Operation	Affect on L1 Caches
Invalidate Block	CACHE_invL2 (ext memory base addr, byte count, wait)	• Lines invalidated	• Corresponding lines invalidated in L1D & L1P • Any L1D updates discarded
Writeback Block	CACHE_wbL2 (ext memory base addr, byte count, wait)	• Dirty lines written back • Lines remain valid	• L1D: Updated data written back, then corresponding lines invalidated • L1P: No affect
Writeback with Invalidate Block	CACHE_wbInvL2 (ext memory base addr, byte count, wait)	• Dirty lines written back • Lines invalidated	• L1D: Updated data written back, then corresponding lines invalidated • L1P: corr. lines invalidated
Writeback All	CACHE_wbAllL2 (wait)	• Updated lines written back • All lines remain valid	• L1D: Updated data written back, then all lines invalidated • L1P: No affect
Writeback with Invalidate All	CACHE_wbInvAllL2 (wait)	• Updated lines written back • All lines invalidated	• L1D: Updated data written back, then all lines invalidated • L1P: All lines invalidated

- For block operations, only the lines in L1D or L1P with addresses corresponding to the addresses of L2 operations are affected
- Careful: Cache always invalidates/writes back *whole lines*. To avoid unexpected coherence problems: align buffers at a boundary equal to the cache line size and make the size of the buffers a multiple of the cache line size



When to Use Coherency Functions?

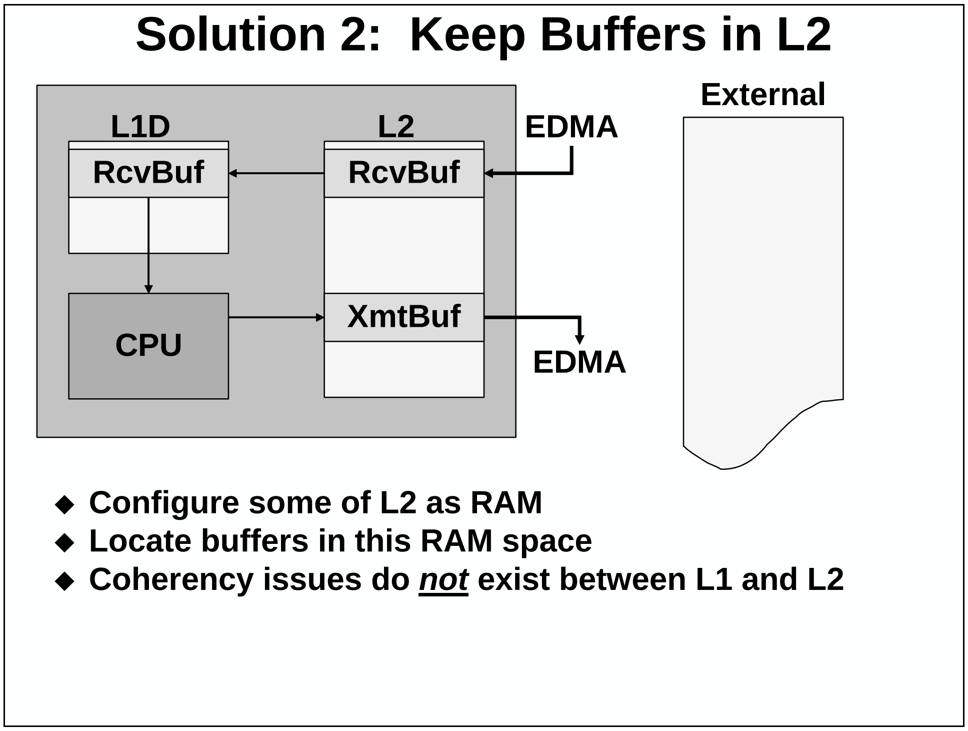
- ***Use When*** **CPU and EDMA share a cacheable region in external memory**
- ***Safest:*** **Use** ***L2 Writeback-Invalidate All*** **before any EDMA transfer to/from external memory.** ***Disadvantage:*** **Larger Overhead**
- ***Reduce overhead*** **by:**
  - **Only operating on buffers used for EDMA, and**
  - **Distinguishing between three possible scenarios:**

1. EDMA reads data written by the CPU	Writeback before EDMA
2. EDMA writes data to be read by the CPU	Invalidate before EDMA*
3. EDMA modifies data written by the CPU that is to be read back by the CPU	Writeback-Invalidate before EDMA

** For C6211/6711 use Writeback-Invalidate before EDMA*

Solution 2: Use L2 Memory

A second solution to the coherency issues is to let the device handle them for you. Start by linking the buffers into addressable L2 memory rather than external memory. The EDMA can then transfer in and out of these buffers without any coherency issues. What about coherency issues between L1 and L2? The cache controller handles all coherency issues between L1 and L2.



Solution 2: Keep Buffers in L2

- **Configure some of L2 as RAM**
- **Locate buffers in this RAM space**
- **Coherency issues do *not* exist between L1 and L2**

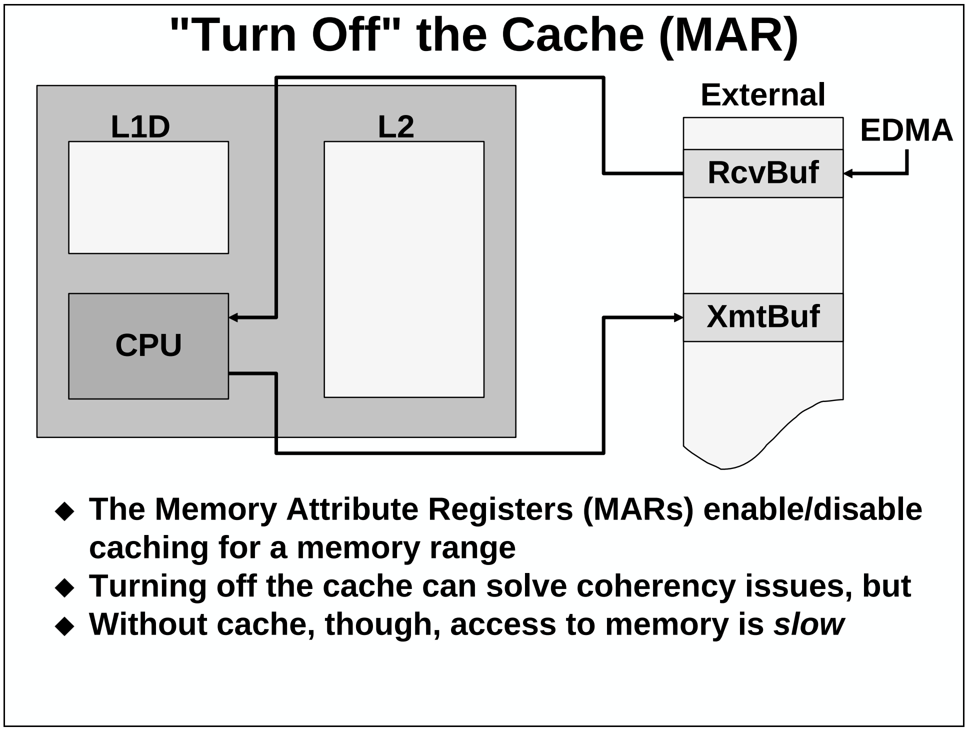
This solution may be the simplest and best for the designer. It is a powerful solution, especially when considering that the EDMA could be transferring from another peripheral, the McBSP. In this case, it is best to have the EDMA transfer to on-chip buffers so that they don't have to be brought back in again by the cache controller as we discussed earlier. Add this to the fact that all coherency issues are taken care of for you, and this makes for a powerful, efficient solution

“Turn Off” the Cache (MAR)

As stated earlier in the chapter, the L1 cache cannot be *turned-off*. While this is true, alternatively a region of memory can be made *non-cacheable*. A memory access that must go all the way to the original memory location is called a *long-distance* access.

Using the Memory Attribute Registers (MAR), one can force the CPU to do a *long-distance* access to memory every time a read or write is performed. The L1 and/or L2 cache is not used for these long-distance accesses.

Why would you want to prevent some memory addresses from being cached? Often there are values found in off-chip, memory-mapped registers that must be read anew each time they are accessed. One example of this might be a system that references a hardware status register found in a field programmable gate array (FPGA). Another example where this might be useful is a FIFO out in external memory, where the same memory address is read repeatedly, but a different value is accessed for each read.

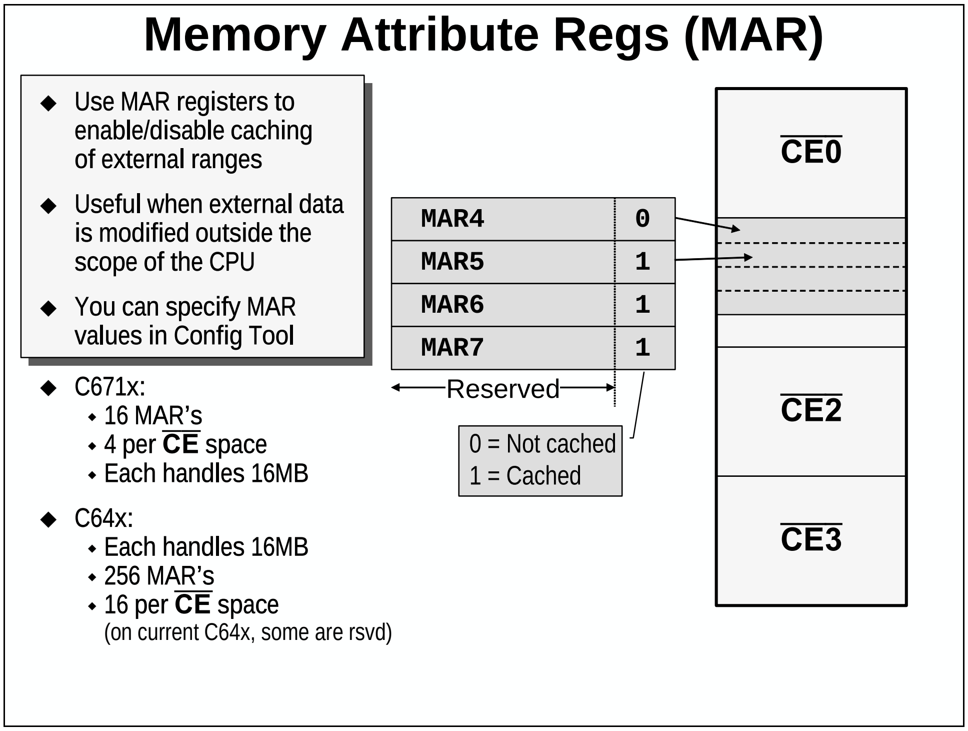


"Turn Off" the Cache (MAR)

- ◆ **The Memory Attribute Registers (MARs) enable/disable caching for a memory range**
- ◆ **Turning off the cache can solve coherency issues, but**
- ◆ **Without cache, though, access to memory is *slow***

While MAR’s may also provide a solution to coherency issues, this is not a recommended solution because long-distance accesses can be extremely slow. If accesses infrequently, this decreased speed may not be an issue, but if used for real-time data acceses the decreased performance may keep the system from operating correctly anyway, coherency issues or not.

The Memory Attribute Registers allow the designer to turn cacheability on and off for a given address range. Each MAR controls the cacheablity of 16MB of external memory.



Memory Attribute Regs (MAR)

- Use MAR registers to enable/disable caching of external ranges
- Useful when external data is modified outside the scope of the CPU
- You can specify MAR values in Config Tool
- C671x:
  - 16 MAR's
  - 4 per $\overline{\textbf{CE}}$ space
  - Each handles 16MB
- C64x:
  - Each handles 16MB
  - 256 MAR's
  - 16 per $\overline{\textbf{CE}}$ space
    (on current C64x, some are rsvd)

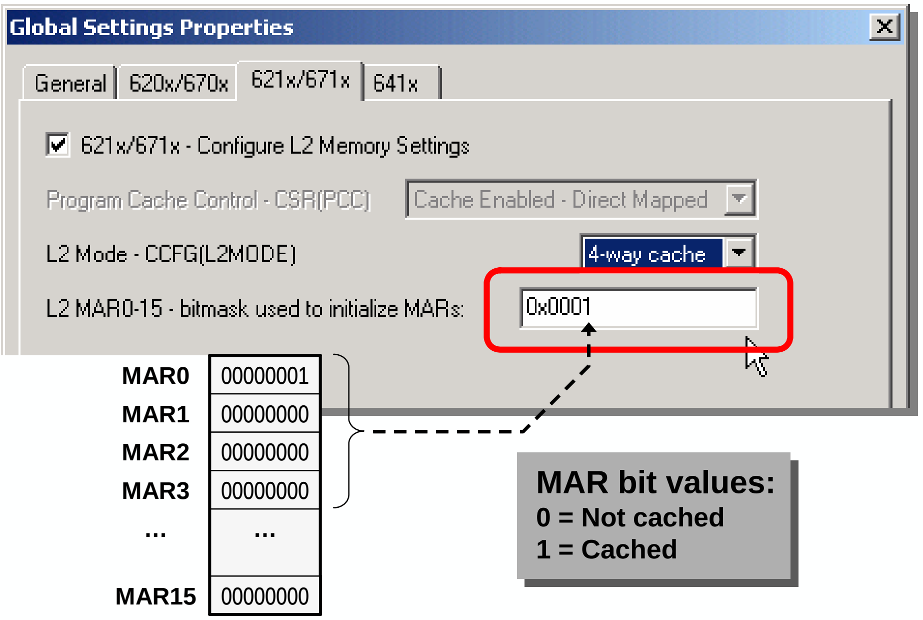
MAR4	0
MAR5	1
MAR6	1
MAR7	1

Reserved

0 = Not cached
1 = Cached

These registers can be used to control the caching of different ranges by setting the appropriate bit to 1 for cache enabled and 0 for cache disabled. These registers can also be setup using the configuration tool.

Setting MARs in CDB (C67x)



☑ 621x/671x - Configure L2 Memory Settings

Program Cache Control - CSR(PCC): Cache Enabled - Direct Mapped

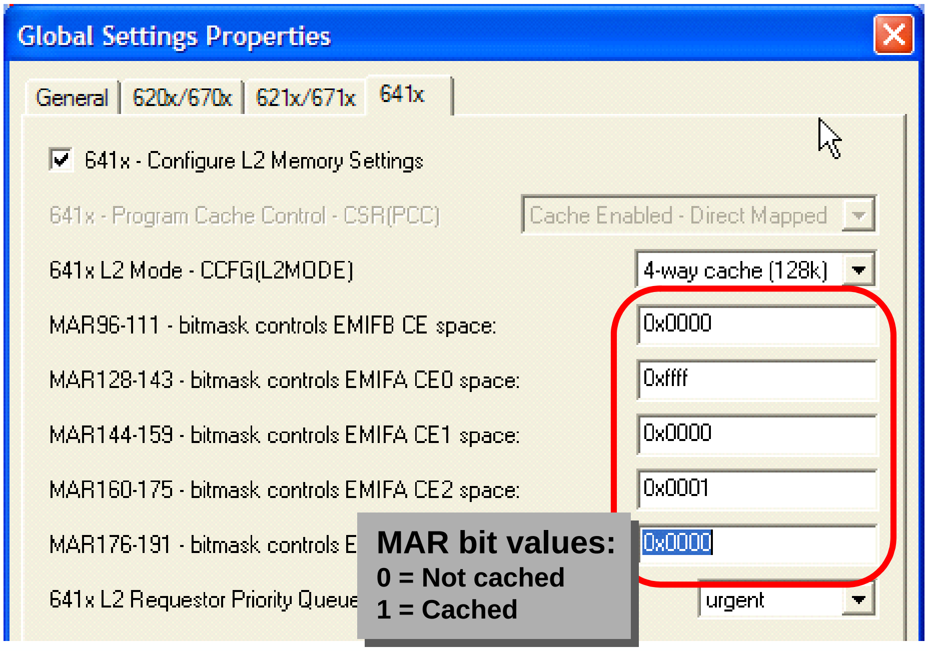
L2 Mode - CCFG(L2MODE): 4-way cache

L2 MAR0-15 - bitmask used to initialize MARs: 0x0001

MAR0	00000001
MAR1	00000000
MAR2	00000000
MAR3	00000000
...	...
MAR15	00000000

MAR bit values:
0 = Not cached
1 = Cached

Setting MARs in CDB (C64x)



☑ 641x - Configure L2 Memory Settings

641x - Program Cache Control - CSR(PCC): Cache Enabled - Direct Mapped

641x L2 Mode - CCFG(L2MODE): 4-way cache (128k)

MAR96-111 - bitmask controls EMIFB CE space: 0x0000

MAR128-143 - bitmask controls EMIFA CE0 space: 0xffff

MAR144-159 - bitmask controls EMIFA CE1 space: 0x0000

MAR160-175 - bitmask controls EMIFA CE2 space: 0x0001

MAR176-191 - bitmask controls E: 0x0000

641x L2 Requestor Priority Queue: urgent

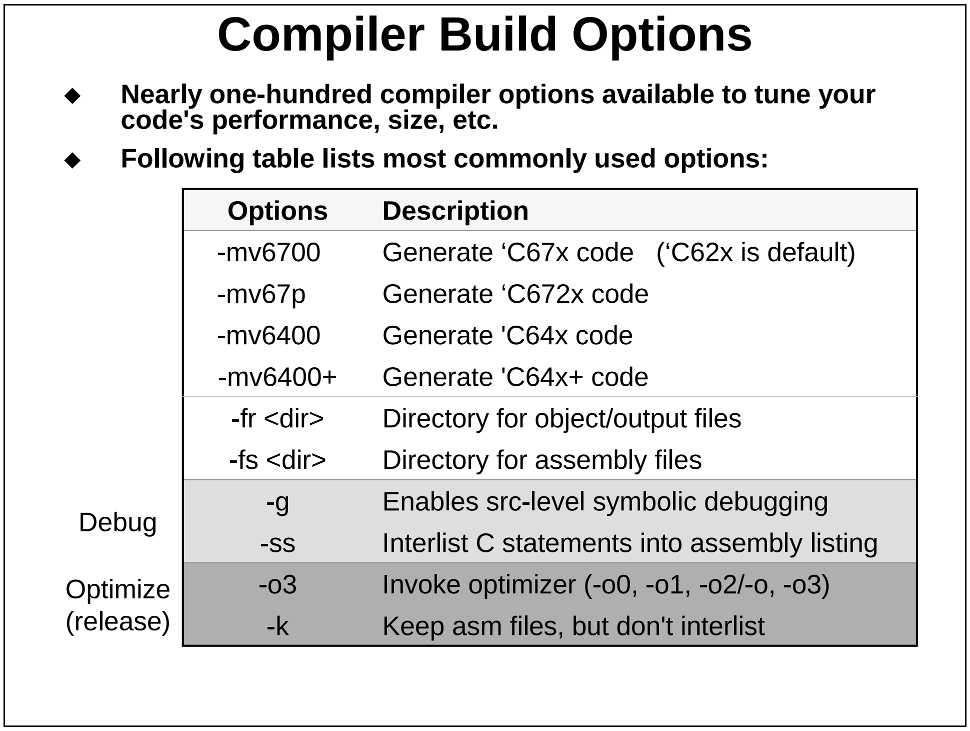
MAR bit values:
0 = Not cached
1 = Cached

Using the C Optimizer

One way to quickly optimize your code is to use the Release configuration. So far in the workshop, we haven’t talked much about optimizations. A full optimizations class (OP6000) is available to take if you desire. For this workshop, we’ll just hit the “dummy mode” button to turn ON the optimizer. There are several ways to do this – by using Build Options and going to the Compiler Tab and turning on –o3. Or, just click on the Release Build Configuration that is already set up for you (as we discuss below).

In the lab, we’ll use the Release configuration and do some benchmarking on code speed and size.

Compiler Build Options



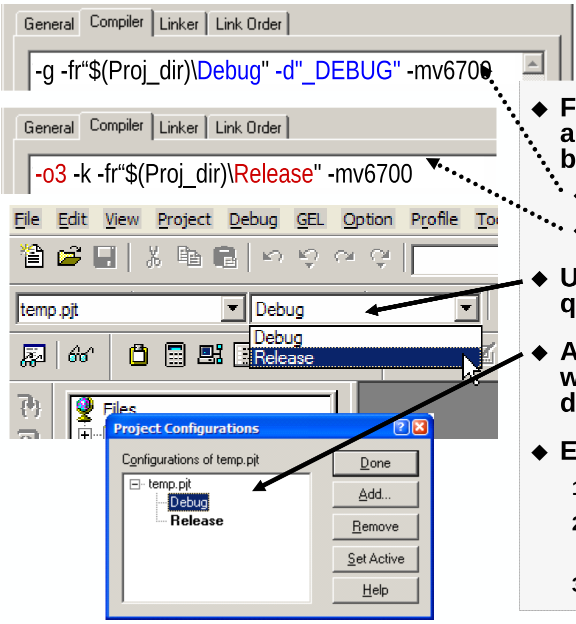
Compiler Build Options

- **Nearly one-hundred compiler options available to tune your code's performance, size, etc.**
- **Following table lists most commonly used options:**

	Options	**Description**
	-mv6700	Generate ‘C67x code (‘C62x is default)
	-mv67p	Generate ‘C672x code
	-mv6400	Generate 'C64x code
	-mv6400+	Generate 'C64x+ code
	-fr <dir>	Directory for object/output files
	-fs <dir>	Directory for assembly files
Debug	-g	Enables src-level symbolic debugging
	-ss	Interlist C statements into assembly listing
Optimize (release)	-o3	Invoke optimizer (-o0, -o1, -o2/-o, -o3)
	-k	Keep asm files, but don't interlist

Using Default Build Configurations (Release)

Default Build Configurations



-g -fr“$(Proj_dir)\Debug" -d"_DEBUG" -mv6700

-o3 -k -fr“$(Proj_dir)\Release" -mv6700

Project Configurations

Configurations of temp.pjt

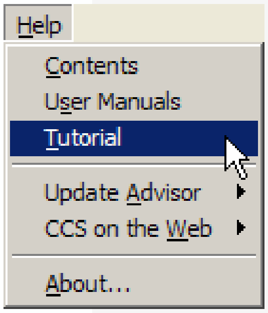
- temp.pjt
  - Debug
  - Release

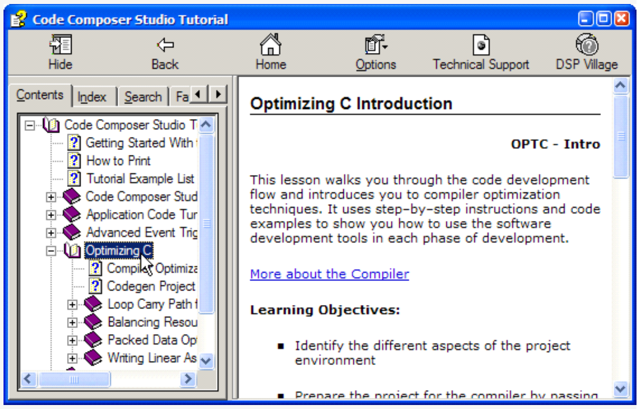
- For new projects, CCS automatically creates two build configurations:
  - Debug (unoptimized)
  - Release (optimized)
- Use the drop-down to quickly select build config.
- Add/Remove build config's with *Project Configurations* dialog (on project menus)
- Edit a configuration:
  1. Set it active
  2. Modify build options (shown next)
  3. Save project

Optimizing C Performance (where to get help)

Optimizing C Performance

- **Compiler Tutorial** (in CCS Help & SPRU425a.pdf)





Optimizing C Introduction

OPTC - Intro

This lesson walks you through the code development flow and introduces you to compiler optimization techniques. It uses step-by-step instructions and code examples to show you how to use the software development tools in each phase of development.

More about the Compiler

Learning Objectives:

- Identify the different aspects of the project environment
- Prepare the project for the compiler by passing

- **C6000 Programmer’s Guide** (SPRU198)

  Chapter 4: *“Optimizing C Code”*

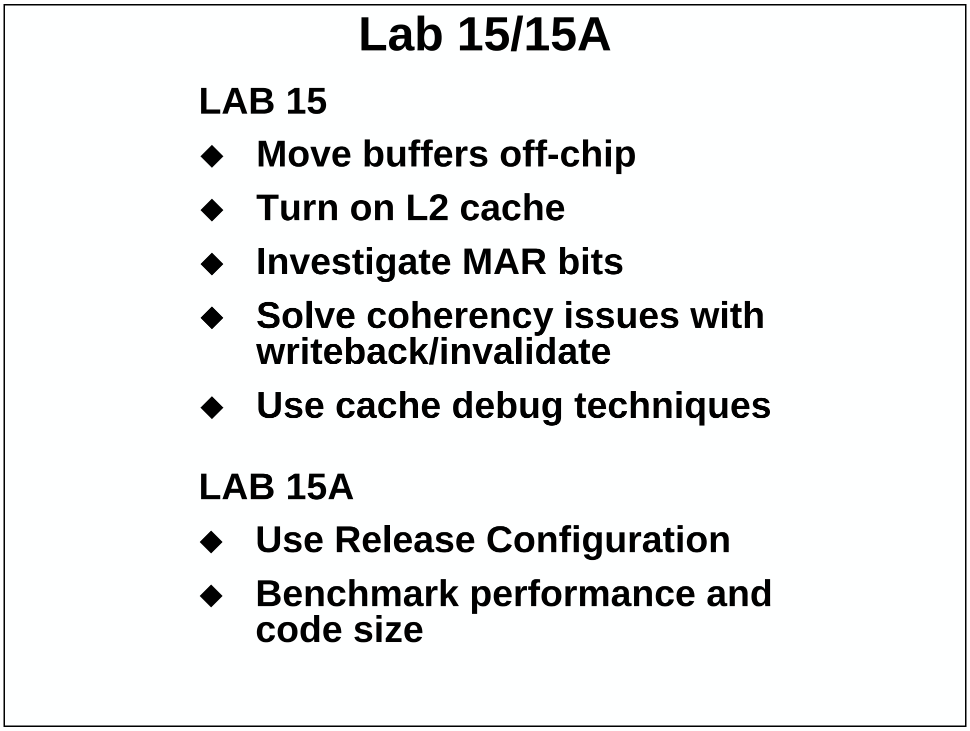
- **C6000 Optimizing C Compiler UG** (SPRU187)

Lab15 – Working with Cache

In lab12 we utilized streams and drivers to interface our application to the hardware of the DSK. The driver we used from the DDK just happens to have *cache coherency built into it.* (Investigating this feature is left for a home exercise). If we used lab12 to investigate cache problems, we wouldn't have any. So we'll reload lab11 (pre-streams and pre-driver) to understand the concepts needed to work with the 6x cache. Besides that, it's just possible that you might not use a cache-coherent driver in your system. ☺

We're going to use the L2 Cache on the 'C6416 and the 'C6713 instead of using all of it as internal SRAM. This will allow us to see how to create a system that uses cache effectively. The general process will be:

- Start from a working lab 11 code base
- Use the .CDB file to move the buffers off-chip and turn the L2 cache on
- Use the MAR bits to make the external memory region uncacheable
- Use CSL cache calls to make the system work with L2 cache and cacheable external memory
- Use a nice debugger trick to view the values stored in cache vs. what is in external memory



Lab 15/15A

LAB 15

- ◆ **Move buffers off-chip**
- ◆ **Turn on L2 cache**
- ◆ **Investigate MAR bits**
- ◆ **Solve coherency issues with writeback/invalidate**
- ◆ **Use cache debug techniques**

LAB 15A

- ◆ **Use Release Configuration**
- ◆ **Benchmark performance and code size**

Lab 15 Procedure

In this lab, we’re going to move the buffers off-chip and turn on the L2 cache. We'll change several cache settings to see what their effect is on the system.

Copy Files and Rename the Project

1. **Copy Lab11 folder to the audioapp folder**

   In the **`c:\iw6000\labs`** folder, delete the \audioapp folder. Right-click on your lab11 solution and select copy. Move your mouse to an open spot in the \labs folder, right click and choose paste. You will now have a “copy of” the lab11 folder. Rename the folder to audioapp. You now have your lab11 code as a base for beginning this lab.

2. **Reset the DSK, start CCS and open audioapp.pjt**

Move Buffers Off Chip and Turn on the L2 Cache

3. **Use the Configuration Tool to move the buffers to the off-chip SDRAM**

   Open your .cdb file and navigate to the Memory Manager. Open its *properties* view and select the *Compiler Sections* tab. Move the .bss and .far sections from ISRAM (or IRAM) to SDRAM. Click OK.

64

4. **Change the properties of the ISRAM segment**

   In order to turn on some of the L2 cache, we need to decrease the amount that is dedicated to SRAM. Open the properties for the ISRAM segment. Change the *len* property to **`0x000C0000`**. This will leave us space for 256KB of cache. Click OK.

5. **Turn on the L2 cache**

   Open the *Global Settings* properties box. Choose the C641x tab. Check the *641x-Configure L2 Memory Settings* checkbox. Change the L2 mode to ”4-way cache (256k)”.

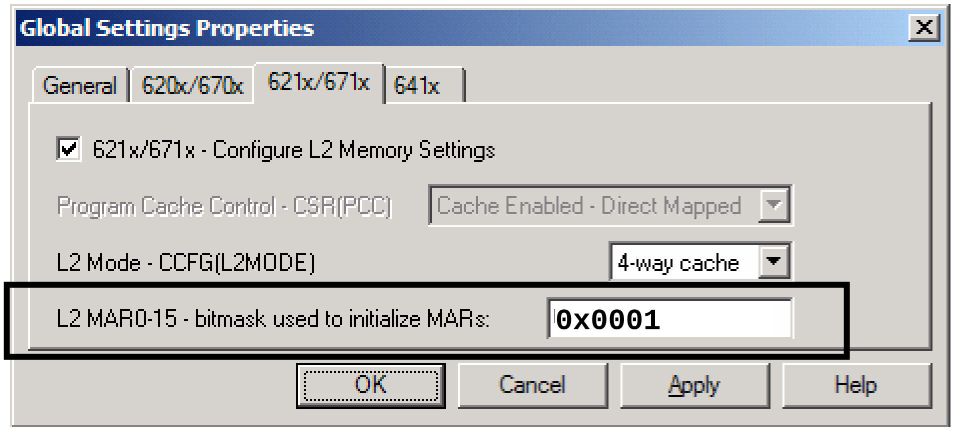
6. **Modify the MAR bits**

   Change the MAR value for the EMIFA CE0 space from 0x0000, to 0x0001. This change will make the SDRAM region cacheable. Click OK.

67

7. Change the Memory Attribute Register Settings (MAR bits)

In audioapp.cdb, under System, right-click on *Global Settings* and select *Properties*. Select the "621x/671x" tab. Verify that the setting highlighted below is set to 0x0001. This enables the L2 cache.



☑ 621x/671x - Configure L2 Memory Settings

Program Cache Control - CSR(PCC): Cache Enabled - Direct Mapped

L2 Mode - CCFG(L2MODE): 4-way cache

L2 MAR0-15 - bitmask used to initialize MARs: 0x0001

The value for the MAR bits in the .cdb file allocates 1 bit for each of the MAR registers, and each register corresponds to a given memory region. The value of the bit in the ith position determines the cacheability of that region. For example, a 1 in the 0th position makes the MAR 0th region (from 0x80000000 to 0x80FFFFFF) cacheable, and the other regions uncacheable.

8. Build the program, Reload the program, and Run to main()

9. Run and Listen

What is the system doing now? Probably not what you want to hear. Move on to the next step to figure out what is going on. Halt the CPU.

Debugging Cache

This section will describe a nice little debugger trick that we can use to figure out what is going on with the cache in our system. In order to use this trick, we need three things:

- The external memory range needs to use aliased addressing. This means that we can use two different addresses (an alias) to access the same memory location. We also need for these two addresses to be in two different MAR regions. We will set one region to be cacheable and the other to be uncacheable. The SDRAM on the DSK has aliased addresses.
- If we are using the memory mapping feature of Code Composer Studio, we need to make sure that there is a memory range created for each one of the memory region addresses from the previous requirement.
- Two memory windows open at each of the memory ranges. Depending on how we set the MAR bits above, one will show the value currently stored in cache, and the other will show the actual value stored at the memory address (in the SDRAM).

Note: The debugger always shows values from the CPU's point of view. So, when we use a memory window to view an address, we are seeing what the CPU sees. In other words, if an address is currently cached, we will see the value in cache and NOT the value in external memory. The trick above tells the CPU that one of the memory aliases is not cacheable (the one with the MAR bit set to 0), therefore it will go out to the external memory and show us what is stored there. With two memory windows, we can see both. A note within a note, we shouldn't edit the values using the memory windows at this point since we could easily corrupt the data.

10. Open the startup GEL file

CCS has a setting to tell it what memory looks like. We can use this feature to detect accesses to invalid memory addresses. Up to this point, this has all been set up for us by the startup GEL file. To add a memory range to the debugger, we will need to modify this file.

Open the GEL file located in the GEL files folder in the project view. This is the pane that lists all of the files in your project. The file should be called DSK6416.gel or DSK6713.gel.

11. **Add a GEL_MapAdd() function call for the new memory region**

    Find the following line of code in the setup_memory_map( ) function of the GEL file:

    ```
    GEL_MapAdd(0x80000000,0,0x01000000,1,1); // 16MB SDRAM…
    ```

    This function adds a 16MB region at location 0x80000000. This represents the SDRAM on the DSK.

12. **Copy and paste this line. Change the address of the copied text to start at location 0x81000000.**

    This is an aliased address for the SDRAM which happens to fall in the second MAR region. The MAR bit for this region is currently disabled by the configuration tool.

    Save the changes to the GEL file and close the file.

13. **Reload the GEL file**

    Reload the GEL file that we just modified by right-clicking on it in the project view and selecting **reload**.

14. **Apply the changes to CCS using the GEL menu**

    We have now made the necessary changes to the CCS memory map, but they have not been applied yet. Use the following menu command to apply the changes:

    ```
    GEL → Memory Map → SetMemoryMap
    ```

15. **Open a memory window to view the cached values (L2)**

    Use the following command to open a memory window:

    View → Memory

    Inside the box, change the **address** to **gBufXmtLPing**. You can also change the **title** to something more meaningful like Cache or L2 if you'd like. Click OK.

16. **Open a memory window to view the non-cached values (the SDRAM)**

    Open another memory window to view the same address that was opened up by the previous command, but change the second hex digit from 0 to 1. For example, if **gBufRcvLPing** resides at 0x80000000, we would change the address in this watch window to 0x81000000. You can also change the title of this memory window to something like SDRAM if you'd like.

    The L2 memory window will use the CPU to display addresses in the 0x80000000 to 0x80FFFFFF range, which is marked as cacheable. Therefore, we will see values which are currently stored in cache if they are valid in cache. The second window will use the CPU to show addresses in the 0x81000000 to 0x81FFFFFF range that is marked as uncacheable. So, the CPU will go out to the external memory and show us what is stored there. This allows us to see the values in the cache and the values currently stored at the actual address.

17. Use the memory windows to observe the system

Using this new visualization capability, step through code, especially the initialization code that writes 0's to the transmit buffers in main(). Specifically, try setting a breakpoint in the for loop. Are the 0's being written into cache or into the SDRAM? Where does the EDMA transfer the values from? You should be able to see that once the addresses are allocated in cache, the CPU is no longer accessing the SDRAM even though the values in the SDRAM (the correct values) are changing (or should be changing).

All of this was to show that this system is not working because the CPU is accessing the data in cache (over and over again) instead of accessing the real values out in external memory.

Use L2 Cache Effectively

18. Align buffers on a cache line size boundary in main.c

We need to make sure that the buffers that we access occupy a cache line by themselves. This will maximize the efficiency of the cache when using clean and flush calls later. We can do this by aligning the buffers on a 128 byte boundary, which is the size of an L2 line. The line of code shows how we can use a C pragma statement to do this for the receive ping buffer:

```
#pragma DATA_ALIGN(gBufRcvLPing, 128);
```

Make sure to add a pragma for each of the data buffers. Above the 8 lines declaring the buffers in main.c, add 8 of these #pragma statements – one for each buffer as shown below:

```
#pragma DATA_ALIGN(gBufRcvLPing, 128);
#pragma DATA_ALIGN(gBufRcvRPing, 128);
#pragma DATA_ALIGN(gBufRcvLPong, 128);
#pragma DATA_ALIGN(gBufRcvRPong, 128);
#pragma DATA_ALIGN(gBufXmtLPing, 128);
#pragma DATA_ALIGN(gBufXmtRPing, 128);
#pragma DATA_ALIGN(gBufXmtLPong, 128);
#pragma DATA_ALIGN(gBufXmtRPong, 128);
short gBufRcvLPing[BUFFSIZE];
short gBufRcvRPing[BUFFSIZE];
short gBufRcvLPong[BUFFSIZE];
short gBufRcvRPong[BUFFSIZE];
short gBufXmtLPing[BUFFSIZE];
short gBufXmtRPing[BUFFSIZE];
short gBufXmtLPong[BUFFSIZE];
short gBufXmtRPong[BUFFSIZE];
```

19. Add a call to CACHE_invL2() for the input buffers

64

The invalidate operation is necessary to invalidate the addresses for the processed buffer in L2. If the addresses are NOT invalidated, the CPU will read the values from cache the next time it wants to read the buffer. Unfortunately, these values will be incorrect as they will be the OLD data, not the new data that has been written to the buffers in external memory by the EDMA.

Between the 1st and 2nd closing braces "}" of **processBuffer()**, add the following code:

```
CACHE_invL2(sourceL, BUFFSIZE * 2, CACHE_NOWAIT);
CACHE_invL2(sourceR, BUFFSIZE * 2, CACHE_NOWAIT);
```

67

Add a call to CACHE_wbInvL2() for the input buffers

In the processBuffer() function, after you have processed an input buffer, call the CSL writeback/invalidate API to invalidate the addresses in L2. Make sure to do this for both the ping and pong receive buffers. Make sure that the invalidate will happen for both the FIR filter and the copy routines for both channels.

The writeback/invalidate operation is necessary to invalidate the addresses for the processed buffer in L2. If the addresses are NOT invalidated, the CPU will read the values from cache the next time it wants to read the buffer. Unfortunately, these values will be incorrect as they will be the OLD data, not the new data that has been written to the buffers in external memory by the EDMA.

Between the 1st and 2nd closing braces “}” of **processBuffer()**, add the following code:

```
CACHE_wbInvL2(sourceL, BUFFSIZE * 2, CACHE_NOWAIT);
CACHE_wbInvL2(sourceR, BUFFSIZE * 2, CACHE_NOWAIT);
```

20. **Add a call to CACHE_wbL2() for the output buffers**

The writeback is necessary to force the values that are written to L2 by the CPU to the external memory. Since L2 is a write allocate cache, it will allocate a location in cache for writes. When the FIR filter (or the copy) writes their values, these get written to the L2 cache, NOT the external memory. So, to get the values from L2 to external memory so that the EDMA can transfer the new data (the correct data), we need to "writeback" it from L2. Notice that it is not necessary to do an invalidate, as this would just force a new allocation at the next write miss (since we had invalidated the address). It is best to leave these addresses in cache and simply writeback the new data before it is needed in external memory.

Right after the cache invalidate commands you used in the previous step, write the following write back commands:

```
CACHE_wbL2(destL, BUFFSIZE * 2, CACHE_NOWAIT);
CACHE_wbL2(destR, BUFFSIZE * 2, CACHE_NOWAIT);
```

21. **Add a call to CACHE_wbL2() after the initialization of the transmit buffers**

Find the place in main() where we are initializing the output buffers to 0. Add the following code to writeback the zeroes from cache to the SDRAM where the EDMA will start transferring:

```
CACHE_wbL2(gBufXmtLPing, BUFFSIZE * 2, CACHE_NOWAIT);
CACHE_wbL2(gBufXmtLPong, BUFFSIZE * 2, CACHE_NOWAIT);
CACHE_wbL2(gBufXmtRPing, BUFFSIZE * 2, CACHE_NOWAIT);
CACHE_wbL2(gBufXmtRPong, BUFFSIZE * 2, CACHE_NOWAIT);
```

22. **Add a #include statement for csl_cache.h**

We need this for the CACHE_invL2(), CACHE_wbL2(), CACHE_wbInvL2() and other definitions that are used by these calls.

Build and the Run program

23. Build the Program, Load and Run.

24. Verify Operation

This time, the application should work perfectly from cache. Use the memory windows from earlier to observe the clean and flush operations in action. Try to understand how they "fixed" the system.

25. Copy project to preserve your solution.

Using Windows Explorer, copy the contents of:

c:\iw6000\labs\audioapp\*.* TO c:\iw6000\labs\lab15

Lab15a – Using the C Compiler Optimizer

Unfortunately there isn't a whole lot of room for optimization in this software since it's a pretty small project. The two processes using the most cycles are the FIR filter (brought in as a library, so optimization won't affect it) and the sine wave generator. We'll take some measurements of our existing system to see what the changes will be from our, present un-optimized state to an optimized one.

26. Add Statistics APIs to benchmark SINE_add().

Open main.c and find the 2 SINE_add calls in **processBuffer()**. Add the following statement *before* the first SINE_add:

```
STS_set(&sineAddTime,CLK_gethtime());
```

After the second SINE_add, add the following:

```
STS_delta(&sineAddTime,CLK_gethtime());
```

27. Add a Statistics Object to track the benchmark of SINE_add().

Open your .cdb file. Select:

Click the + next to *Instrumentation*. Right click on *STS-Statistics Object Manager* and nsert an STS object named sineAddTime. Open its properties and change the *Unit Type* to High Resolution time based. Click OK and close/save your cdb.

28. Build/load/run your code.

29. Make sure DIP switches are depressed and look at the CPU load graph.

Make sure DIP switches 0 and 1 are depressed – running the sine wave generator and the FIR filter. Open the *CPU load graph*, clear the peak and write your CPU load in the table below (under Not Optimized). For reference, our results are shown in parentheses.

30. Use Statistics View to check the benchmark for sineAddTime.

Open the *BIOS Statistics View*, right-click in it and select clear. Write the max sineAddTime in the table below (under Not Optimized).

31. Find the length of the .text (code) section in the .map file.

Open `audioapp.map` in the `\audioapp\debug\` folder. Find the length of the .text section and write it below (under Not Optimized).

	Not Optimized	Optimized
CPU Load(%)	______(9.46)	______(8.26)
sineAddTime(inst)	______(546984)	______(418552)
.text length	______(6400)	______(6400)

32. Turn on the Optimizer

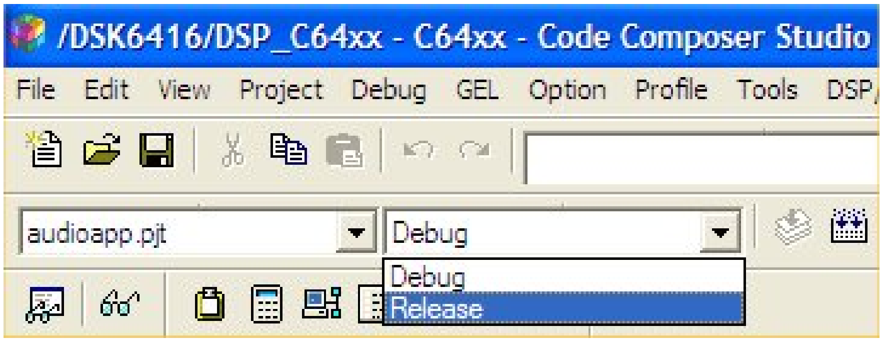
Now that we have a baseline, let’s run the optimizer. First we’ll have to copy some settings. Select:

Project → Build Options → Preprocessor Category

Under *Include Search Path*, copy the entire list of paths. Click Cancel.

33. Choose the Release Build Configuration

Select the Release configuration as show below:


/DSK6416/DSP_C64xx - C64xx - Code Composer Studio
File Edit View Project Debug GEL Option Profile Tools DSP
audioapp.pjt
Debug
Debug
Release

After selecting the Release Build Configuration, open the Project Build Options and note the optimization selections made on the *Basic* page. Click on the *Preprocessor* Category and paste your Include Search path. Add CHIP_6416 to the *Pre-Define Symbol*. Click OK.

34. **Rebuild/load/run and re-do steps 29-31 and add your results to the table.**

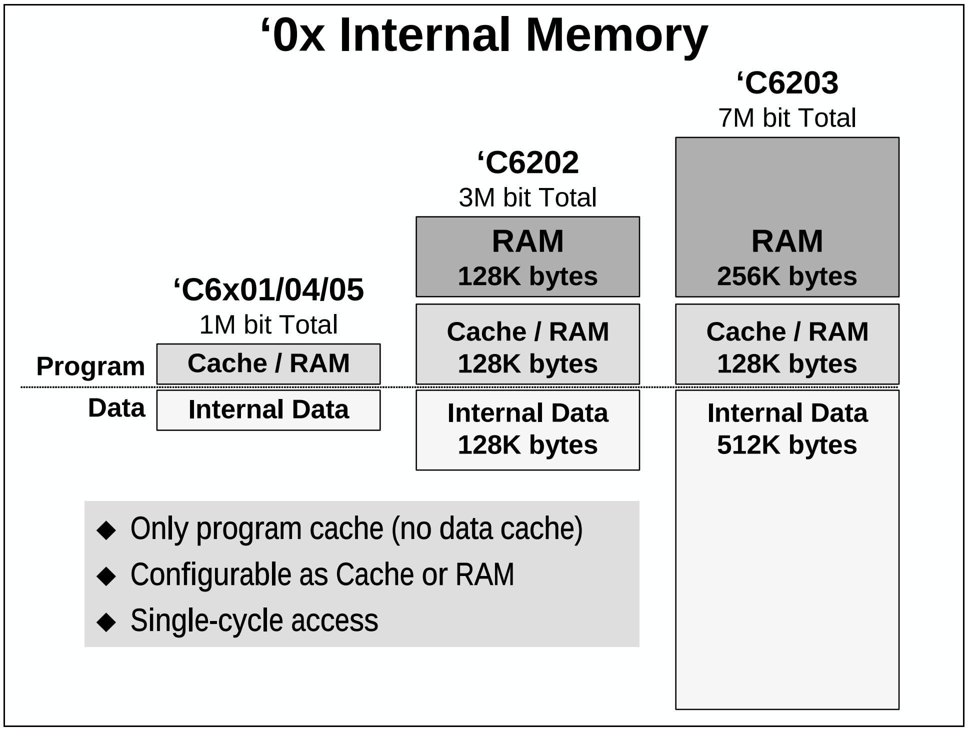
35. **Conclusion**

    We saw the CPU load drop by about 13% and the sineAddTime reduced by about 23%. We didn’t see the code length change at all. Certainly these weren’t significant gains, but well worth the tiny effort. More complex code would likely benefit to a much greater degree.

You’re done

Optional Topics

‘0x Memory Summary



‘0x Internal Memory

‘C6203
7M bit Total

‘C6202
3M bit Total

RAM
256K bytes

RAM
128K bytes

‘C6x01/04/05
1M bit Total

Program

Cache / RAM

Cache / RAM
128K bytes

Cache / RAM
128K bytes

Data

Internal Data

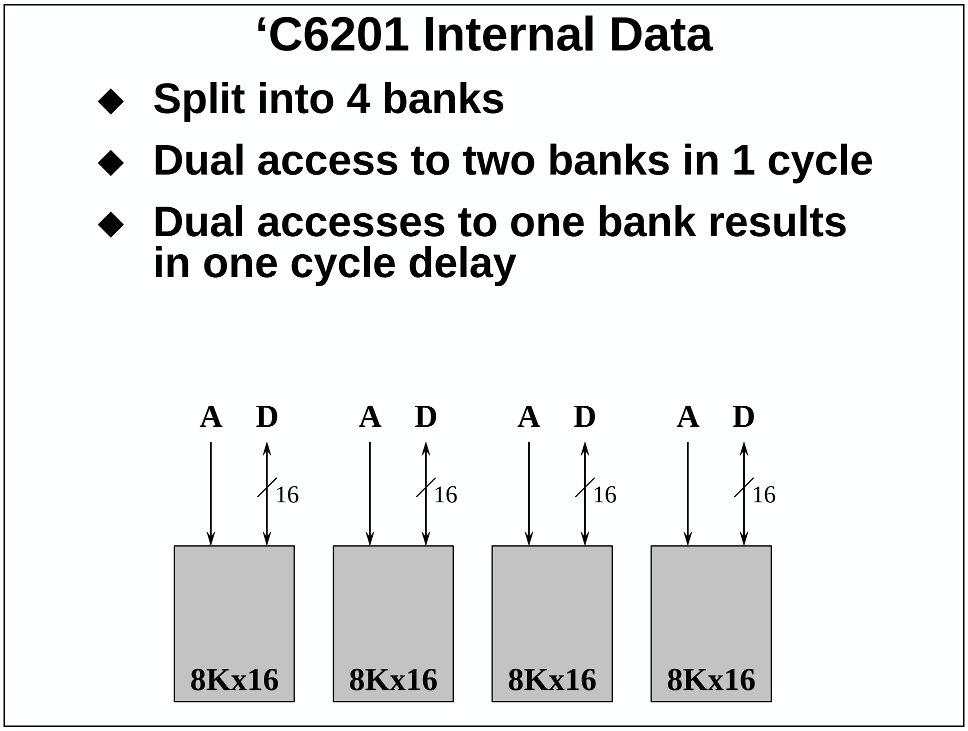
Internal Data
128K bytes

Internal Data
512K bytes

- Only program cache (no data cache)
- Configurable as Cache or RAM
- Single-cycle access

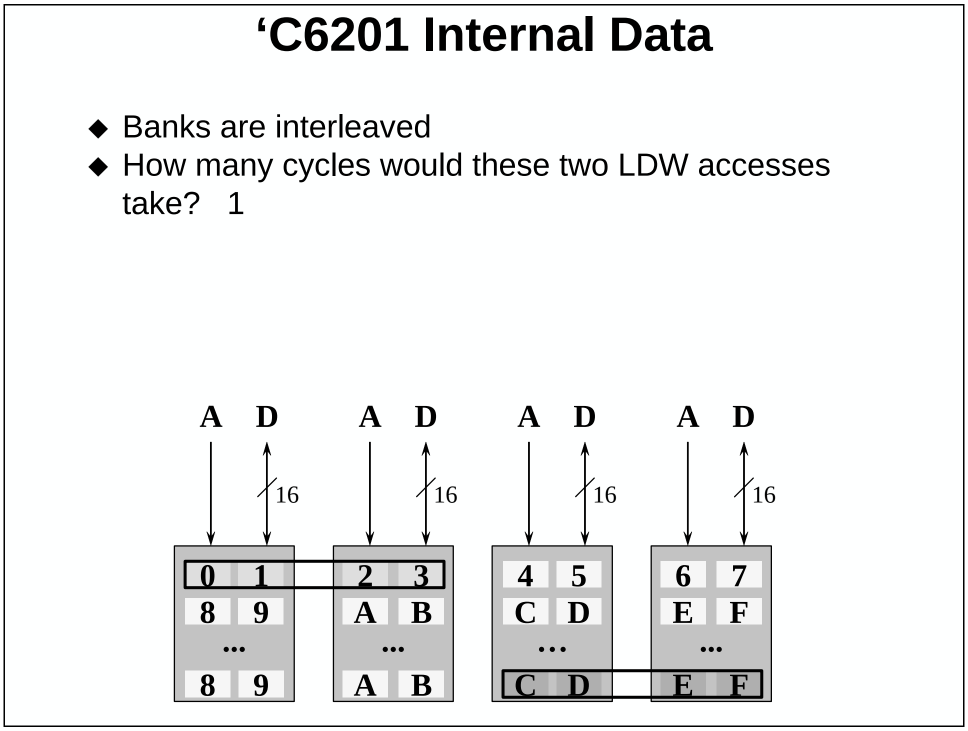
‘0x Data Memory – System Optimzation

Basic Memory Layout



‘C6201 Internal Data

- Split into 4 banks
- Dual access to two banks in 1 cycle
- Dual accesses to one bank results in one cycle delay

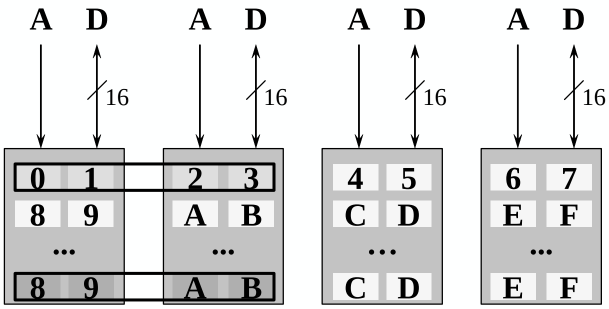


‘C6201 Internal Data

- Banks are interleaved
- How many cycles would these two LDW accesses take?   1

'C6201 Internal Data

- Now, how many cycles would it take for these two LDW's? 2

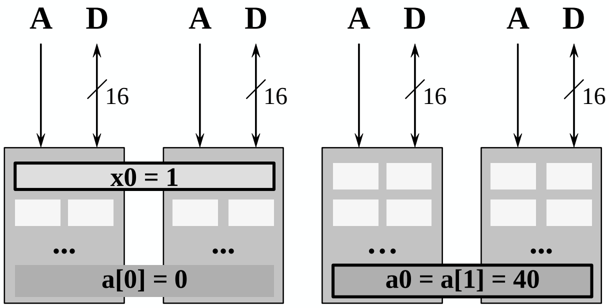


Improving Performance

Solution 1: Offset Arrays

- Offset accesses

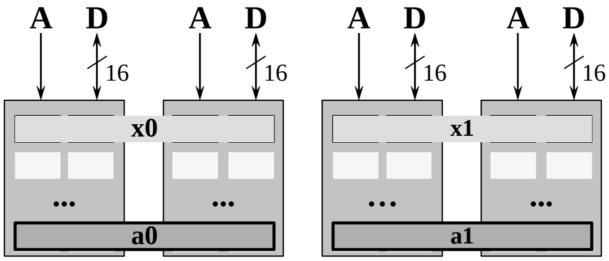
```
#pragma DATA_ALIGN(x, 8);
#pragma DATA_ALIGN(a, 8);
int x[40] = {1, 2, 3, … };
int a[41] = {0, 40, 39, 38, … };
int *xp = &x[0];
int *ap = &a[1];
```



Solution 2: Unroll Loop

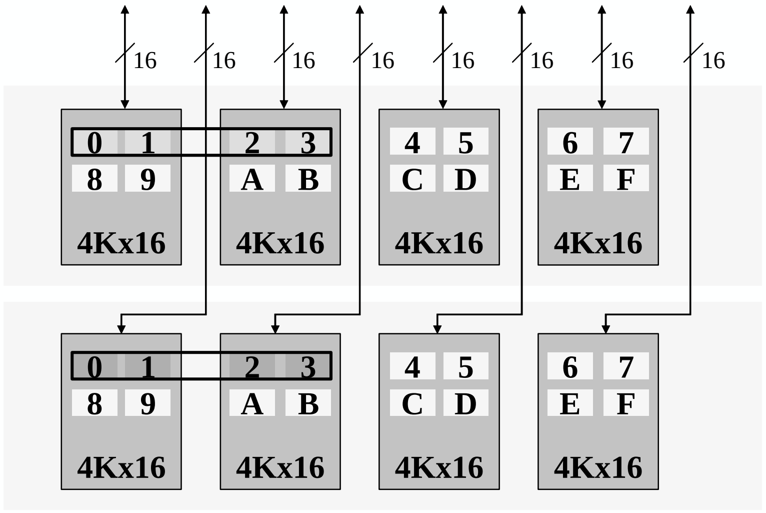
- Offset accesses
- Unroll the loop:
  Read two values from each array in parallel,
  then perform two calculations

```
        LDW    from x0
||      LDW    from x1
        LDW    from a0
||      LDW    from a1
        ADD    x0, a0
||      ADD    x1, a1
```



Aren't There Two Blocks?

Two Blocks of Memory (4 banks each)



- **Why use offset-arrays or loop-unrolling if there's two blocks?**

This allows the DMA unrestricted access to internal memory

The diagram above shows the configuration for the C6201. The C6701 is similar, but each of its banks are 2Kx32 in size. This gives it the same total number of bytes, but allows the C6701 the ability to access two LDDW loads in parallel.