## Introduction

In this chapter the memory options of the C6000 will be considered. By far, the easiest – and highest performance – option is to place everything in on-chip memory. In systems where this is possible, it is the best choice. To place code and initialize data in internal RAM in a production system, refer to the chapters on booting and DMA usage. Most systems will have more code and data than the internal memory can hold. As such, placing everything off-chip is another option, and can be implemented easily, but most users will find the performance degradation to be significant. As such, the ability to enable caching to accelerate the use of off-chip resources will be desirable. For optimal performance, some systems may benefit from a mix of on-chip memory and cache. Fine tuning of code for use with the cache can also improve performance, and assure reliability in complex systems. Each of these constructs will be considered in this chapter.

## Objectives

At the conclusion of this module, you should be able to:

- Set up a system to use internal memory directly
- Configure a system that uses external memory
- Employ the C6000 caches to improve external memory performance
- Optimize a given system by using a balance of cache vs internal RAM
- Modify C source code to work optimally and safely with the cache

## Module Topics

- **Cache Options**
- **Use Internal RAM**
- **Use External Memory**
- **Enable Cache**
- **IRAM and Cache**
- **Tuning C Source Code For Caching**
- **Lab 10: Cache Usage**
  - A. Tune Code and Enable Cache
  - B. Experiment With Larger Buffer Sizes
Use Internal RAM

Option 1 : Use Internal Memory

- When possible, place all code and data into internal RAM
  - Select all internal memory to be mapped as RAM
  - Add IRAM(s) to memory map
  - Route code/data to IRAM(s)

- Ideal choice for initial code development
  - Defines optimal performance possible
  - Avoids all concerns of using external memory
  - Fast and easy to do – just download and run from CCS

- In production systems
  - Add a ROM type resource externally to hold code and initial data
  - Use DMA (or CPU xfer) to copy runtime code/data to internal RAM
  - Boot routines available on most TI DSPs

- Limited range
  - Usually not enough IRAM for a complete system
  - Often need to add external memory and route resources there

C6000 Internal Memory Topology

- Level 1 – or “L1” – RAM
  - Highest performance of any memory in a C6000 system
  - Two banks are provided L1P (for program) and L1D (for data)
  - Single cycle memory with wide bus widths to the CPU

- Level 2 – or “L2” – RAM
  - Second best performance in system, can approach single cycle in bursts
  - Holds both code and data
  - Usually larger than L1 resources
  - Wide bus widths to CPU - via L1 controllers
Use Internal RAM

**Configure IRAM via GCONF**

To obtain maximum IRAM, zero the internal caches, which share this memory.

**Define IRAM Usage via GCONF**

BIOS System Integration - Cache Options
Define IRAM Usage via GCONF

Here, L1D is used for the most critical storage, and all else is routed to L2 "IRAM". A variety of options can be quickly tested, and the best kept in the final revision.

Sample of C6000 On-Chip Memory Options

<table>
<thead>
<tr>
<th>Device</th>
<th>CPU</th>
<th>L1P</th>
<th>L1D</th>
<th>L2</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>C6416T</td>
<td>64</td>
<td>16</td>
<td>16</td>
<td>1024</td>
<td>250</td>
</tr>
<tr>
<td>C6455</td>
<td>64+</td>
<td>32</td>
<td>32</td>
<td>2048</td>
<td>300</td>
</tr>
<tr>
<td>DM6437</td>
<td>64+</td>
<td>32</td>
<td>80</td>
<td>128</td>
<td>30</td>
</tr>
<tr>
<td>C6727</td>
<td>67+</td>
<td>32</td>
<td>0</td>
<td>256</td>
<td>23</td>
</tr>
<tr>
<td>C6747</td>
<td>67++</td>
<td>32</td>
<td>32</td>
<td>* 256</td>
<td>15</td>
</tr>
</tbody>
</table>

Notes:
- Memory sizes are in KB
- Prices are approximate, @ 100pc volume
- 6747 also has 128KB of L3 IRAM
Use External Memory

Option 2 : Use External Memory

- For larger systems, place code and data into external memory
  - Define available external memories
  - Route code/data to external memories

- Essential for systems with environments larger than available internal memory
  - Allows systems with size range from Megs to Gigs
  - Often realized when a build fails for exceeding internal memory range
  - Avoids all concerns of using external memory
  - Fast and easy to do – just download and run from CCS

- Reduced performance
  - Off chip memory has wait states
  - Lots of setup and routing time to get data on chip
  - Competition for off-chip bus : data, program, DMA, ...
  - Increased power consumption

C6000 Memory Topology

- External memory interface has narrower bus widths
- CPU access to external memory costs many cycles
- Exact cycle count varies greatly depending on state of the system at the time
Use External Memory

Define External Memory via GCONF

Define External Usage via GCONF
Enable Cache

**Option 3 : Use Cache & External Memory**

- Improves performance in code loops or re-used data values
  - First access to external resource is ‘normal’
  - Subsequent accesses are from on-chip caches with:
    - Much higher speed
    - Lower power
    - Reduced external bus contention

- Not helpful for non-looping code or 1x used data
  - Cache holds recent data/code for re-use
  - Without looping or re-access, cache cannot provide a benefit

- Not for use with ‘devices’
  - Inhibits re-reads from ADCs and writes to DACs
  - Must be careful when CPU and DMA are active in the same RAMs

- Enabling the cache:
  - Select maximum amounts of internal memory to be mapped as cache
  - Remove IRAM(s) from memory map
  - Route code/data to off-chip (or possible remaining onchip) resources
  - Map off-chip memory as cachable

---

**C6000 Memory Topology**

- Caches automatically collect data and code brought in from EMIF
- If requested again, caches provide the information, saving many cycles over repeated EMIF activity

- Writes to external memory are also cached to reduce cycles and free EMIF for other usage
- Writeback occurs when a cache needs to mirror new addresses
- Write buffers on EMIF reduce need for waiting by CPU for writes
Enable Cache

Configure Cache via GCONF

For best cache results, maximize the internal cache sizes.

Memory Attribute Registers: MARs

- 256 MAR bits define cache-ability of 4G of addresses as 16MB groups
- Many 16MB areas not used by chip or present on given board
- Example: Usable 6437 EMIF addresses at right
- EVM6437 memory is:
  - 128MB of DDR2 starting at 0x8000 0000
  - FLASH, NAND Flash, or SRAM (selected via jumpers) in CS2_ space at 0x4200 0000
- Note: with the C64+ program memory is always cached regardless of MAR settings

<table>
<thead>
<tr>
<th>Start Address</th>
<th>End Address</th>
<th>Size</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x4200 0000</td>
<td>0x42FF FFFF</td>
<td>16MB</td>
<td>CS2_</td>
</tr>
<tr>
<td>0x4400 0000</td>
<td>0x44FF FFFF</td>
<td>16MB</td>
<td>CS3_</td>
</tr>
<tr>
<td>0x4600 0000</td>
<td>0x46FF FFFF</td>
<td>16MB</td>
<td>CS4_</td>
</tr>
<tr>
<td>0x4800 0000</td>
<td>0x48FF FFFF</td>
<td>16MB</td>
<td>CS5_</td>
</tr>
<tr>
<td>0x8000 0000</td>
<td>0x8FFF FFFF</td>
<td>256MB</td>
<td>DDR2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MAR</th>
<th>MAR Address</th>
<th>EMIF Address Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>66</td>
<td>0x0184 8108</td>
<td>4200 0000 – 42FF FFFF</td>
</tr>
<tr>
<td>128</td>
<td>0x0184 8200</td>
<td>8000 0000 – 80FF FFFF</td>
</tr>
<tr>
<td>129</td>
<td>0x0184 8204</td>
<td>8100 0000 – 81FF FFFF</td>
</tr>
<tr>
<td>130</td>
<td>0x0184 8208</td>
<td>8200 0000 – 82FF FFFF</td>
</tr>
<tr>
<td>131</td>
<td>0x0184 820C</td>
<td>8300 0000 – 83FF FFFF</td>
</tr>
<tr>
<td>132</td>
<td>0x0184 8210</td>
<td>8400 0000 – 84FF FFFF</td>
</tr>
<tr>
<td>133</td>
<td>0x0184 8214</td>
<td>8500 0000 – 85FF FFFF</td>
</tr>
<tr>
<td>134</td>
<td>0x0184 8218</td>
<td>8600 0000 – 86FF FFFF</td>
</tr>
<tr>
<td>135</td>
<td>0x0184 821C</td>
<td>8700 0000 – 87FF FFFF</td>
</tr>
</tbody>
</table>
Configure MAR via GCONF

MAR66, 128-135 turned 'on'

BCACHE API

IRAM modes and MAR can be set in code via BCACHE API

- In projects where GCONF is not being used
- To allow active run-time reconfiguration option

**Cache Size Management**

- `BCACHE_getSize(*size)`
  - rtn sizes of all caches
- `BCACHE_setSize(*size)`
  - set sizes of all caches

**MAR Bit Management**

- `marVal = BCACHE_getMar(base)`
  - rtn mar val for given address
- `BCACHE_setMar(base, length, 0/1)`
  - set mars stated address range
**IRAM and Cache**

**Option 4: IRAM & Cache Ext’l Memory**

- **Let some IRAM be Cache to improve external memory performance**
  - First access to external resource is “normal”
  - Subsequent access from on-chip caches – better speed, power, EMIF loading
- **Keep some IRAM as normal addressed internal memory**
  - Most critical data buffers (optimal performance in key code)
  - Target for DMA arrays routed to/from peripherals (2x EMIF savings)
- **Internal program RAM**
  - Must be initialized via DMA or CPU before it can be used
  - Provides optimal code performance
- **Setting the internal memory properties:**
  - Select desired amounts of internal memory to be mapped as cache
  - Define remainder as IRAM(s) in memory map
  - Route code/data to desired on and off chip memories
  - Map off-chip memory as cachable
- **To determine optimal settings**
  - Profile and/or use STS on various settings to see which is best
  - Late stage tuning process when almost all coding is completed

**Select Desired IRAM Configuration**

- Define desired amount of IRAM to be cache (GCONF or BCACHE)
- Balance of available IRAM is “normal” internal mapped-address RAM
- Any IRAM beyond cache limits are always address mapped RAM
- Single cycle access to L1 memories
- L2 access time can be as fast as single cycle
- Regardless of size, L2 cache is always 4 way associative

<table>
<thead>
<tr>
<th>Device</th>
<th>CPU</th>
<th>L1P</th>
<th>L1D</th>
<th>L2</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>C6416T</td>
<td>64</td>
<td>16</td>
<td>16</td>
<td>1024</td>
<td>250</td>
</tr>
<tr>
<td>C6455</td>
<td>64+</td>
<td>32</td>
<td>32</td>
<td>2048</td>
<td>300</td>
</tr>
<tr>
<td>DM6437</td>
<td>64+</td>
<td>32</td>
<td>80</td>
<td>128</td>
<td>30</td>
</tr>
<tr>
<td>C6727</td>
<td>67+</td>
<td>32</td>
<td>0</td>
<td>256</td>
<td>23</td>
</tr>
<tr>
<td>C6747</td>
<td>67++</td>
<td>32</td>
<td>32</td>
<td>*256</td>
<td>15</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>#</th>
<th>L1(kB)</th>
<th>L2 (kB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>32</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>64</td>
</tr>
<tr>
<td>3</td>
<td>16</td>
<td>128</td>
</tr>
<tr>
<td>4</td>
<td>32</td>
<td>256</td>
</tr>
</tbody>
</table>

**Notes:**

- Memory sizes are in KB
- Prices are approximate, @ 100pc qty
- 6747 also has 128KB of L3 IRAM
**Set Cache Size via GCONF or BCACHE**

### Global Settings Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>IRAP Control</td>
<td>Configure Memory Cache Settings</td>
</tr>
<tr>
<td>IRAP L1 PS Mode</td>
<td>16k</td>
</tr>
<tr>
<td>IRAP L1 DTL Mode</td>
<td>4k</td>
</tr>
<tr>
<td>IRAP L2 STG Mode</td>
<td>8k</td>
</tr>
<tr>
<td>MAR 8-31 Mask</td>
<td>0x00000000</td>
</tr>
<tr>
<td>MAR 32-63 Mask</td>
<td>0x00000000</td>
</tr>
<tr>
<td>MAR 64-95 Mask</td>
<td>0x00000000</td>
</tr>
<tr>
<td>MAR 96-127 Mask</td>
<td>0x00000000</td>
</tr>
<tr>
<td>MAR 128-151 Mask</td>
<td>0x00000000</td>
</tr>
<tr>
<td>MAR 152-175 Mask</td>
<td>0x00000000</td>
</tr>
<tr>
<td>MAR 176-207 Mask</td>
<td>0x00000000</td>
</tr>
</tbody>
</table>

### Cache Size Management

typedef struct

BCACHE_Size {

BCACHE_L1_Size l1psize;
BCACHE_L1_Size l1dsize;
BCACHE_L2_Size l2size;
} BCACHE_Size;

<table>
<thead>
<tr>
<th>#</th>
<th>L1 (kB)</th>
<th>L2 (kB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>32</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>64</td>
</tr>
<tr>
<td>3</td>
<td>16</td>
<td>128</td>
</tr>
<tr>
<td>4</td>
<td>32</td>
<td>256</td>
</tr>
</tbody>
</table>

**C64x+ L1D Memory Banks**

- Only one L1D access per bank per cycle
- Use DATA_MEM_BANK pragma to begin paired arrays in different banks
- Note: sequential data are not down a bank, instead they are along a horizontal line across across banks, then onto the next horizontal line
- Only even banks (0, 2, 4, 6) can be specified

```c
#pragma DATA_MEM_BANK(a, 4);
short a[256];

#pragma DATA_MEM_BANK(x, 0);
short x[256];
for(i = 0; i < count; i++) {
    sum += a[i] * x[i];
}
```
5: Tune Code for Cache Optimization

- Align key code and data for maximal cache usage
  - Match code/data to fit cache lines fully – align to 128 bytes
- Clear caches when CPU and DMA are both active in a given memory
  - Keep cache from presenting out-of-date values to CPU or DMA
- Size and align cache usage where CPU and DMA are both active
  - Avoid risk of having neighboring data affected by cache clearing operations
- Freeze cache to maintain contents
  - Lock in desired cache contents to maintain performance
  - Ignore new collecting until cache is 'thawed' for reuse

There are many ways in which caching can lead to data errors, however a few simple techniques provide the 'cure' for all these problems

---

Cache Coherency

Example of read coherency problem:
1. DMA collects Buf A
2. CPU reads Buf A, buffer is copied to Cache; DMA collects Buf B
3. CPU reads Buf B, buffer is copied to Cache; DMA collects Buf C over “A”
4. CPU reads Buf C… but Cache sees “A” addresses, provides “A” data – error!
5. Solution: Invalidate Cache range before reading new buffer

Write coherency example:
1. CPU writes Buf A. Cache holds written data
2. DMA reads non-updated data from external memory – error!
3. Solution: Writeback Cache range after writing new buffer

Program coherency:
1. Host processor puts new code into external RAM
2. Solution: Invalidate Program Cache before running new code

Note: there are NO coherency issues between L1 and L2!
Managing Cache Coherency

<table>
<thead>
<tr>
<th>Invalidate</th>
<th>BCACHE_inv(blockPtr, byteCnt, wait)</th>
<th>BCACHE_invL1pAll()</th>
</tr>
</thead>
<tbody>
<tr>
<td>Writeback</td>
<td>BCACHE_wb(blockPtr, byteCnt, wait)</td>
<td>BCACHE_wbAll()</td>
</tr>
<tr>
<td>Invalidate &amp; Writeback</td>
<td>BCACHE_wbInv(blockPtr, byteCnt, wait)</td>
<td>BCACHE_wbInvAll()</td>
</tr>
<tr>
<td>Sync to Cache</td>
<td>BCACHE_wait()</td>
<td></td>
</tr>
</tbody>
</table>

- blockPtr : start address of range to be invalidated
- byteCnt : number of bytes to be invalidated
- Wait : 1 = wait until operation is completed

Coherence Side Effect – False Addresses

- False Address: ‘neighbor’ data in the cache but outside the buffer range
- Reading data from the buffer re-reads entire line
  - If ‘neighbor’ data changed externally before CPU was done using prior state, old data will be lost/corrupted as new data replaces it
- Writing data to buffer will cause entire line to be written to external memory
  - External neighbor memory could be overwritten with old data

- False Address problems can be avoided by aligning the start and end of buffers on cache line boundaries
  - Align memory on 128 byte boundaries
  - Allocate memory in multiples of 128 bytes

```c
#define BUF 128
#pragma DATA_ALIGN (in,BUF)
short in[2][20*BUF];
```
Cache Freeze (C64x+)

- Freezing cache prevents data that is currently cached from being evicted
- Cache Freeze
  - Responds to read and write hits normally
  - No updating of cache on miss
  - Freeze supported on C64x+ L2/L1P/L1D
- Commonly used with Interrupt Service Routines so that one-use code does not replace realtime algo code
- Other cache modes: Normal, Bypass

**Cache Mode Management**

```
Mode = BCACHE_getMode(level) // return state of specified cache
oldMode = BCACHE_setMode(level, mode) // set state of specified cache
```

```
typedef enum {
    BCACHE_L1D,
    BCACHE_L1P,
    BCACHE_L2
} BCACHE_Level;
```

```
typedef enum {
    BCACHE_NORMAL,
    BCACHE_FREEZE,
    BCACHE_BYPASS
} BCACHE_Mode;
```

**BCACHE-Based Cache Setup Example**

This BCACHE example shows how to put the EVM 6437 in the default power-up mode.
(Note: code such as this will required for stand-alone bootup where CCS GEL files are not present)

```c
#include "myWorkcfg.h" // most BIOS headers provided by config tool
#include <bcache.h> // headers for DSP/BIOS Cache functions
#define DDR2BASE   0x80000000; // size of DDR2 area on DM6437 EVM
#define DDR2SZ        0x07D00000; // size of external memory

setCache() {
    struct BCACHE_Size cachesize; // L1 and L2 cache size struct
    cachesize.l1dsize = BCACHE_L1_32K; // L1D cache size 32k bytes
    cachesize.l1psize = BCACHE_L1_32K; // L1P cache size 32k bytes
    cachesize.l2size = BCACHE_L2_0K; // L2 cache size ZERO bytes
    BCACHE_setSize(&cacheSize); // set the cache sizes

    BCACHE_setMode(BCACHE_L1D, BCACHE_NORMAL); // set L1D cache mode to normal
    BCACHE_setMode(BCACHE_L1P, BCACHE_NORMAL); // set L1P cache mode to normal
    BCACHE_setMode(BCACHE_L2,   BCACHE_NORMAL); // set L2 cache mode to normal

    BCACHE_inv(DDR2BASE, DDR2SZ, TRUE); // invalidate DDR2 cache region
    BCACHE_setMar(DDR2BASE,DDR2SZ,1); // set DDR2 to be cacheable
}
```
C64+ Cache Controller Review

- Select how much IRAM and Cache is needed
- Enable caching via MARs
- Align to 128
- Allocate in multiples of 128

- Invalidate cache before reads from memory under external control
- Writeback cache after writing to RAM under external control
Lab 10: Cache Usage

A. Tune Code and Enable Cache

1. Open CCS and load your most recent prior solution project myWork.pjt
2. Add to proc.h a # define for symbol BUF of 128
   in proc.c:
3. Add a pragma to make the out buffers of the same linker type as the in buffers
4. Align the in and out buffers
5. Declare the out buffers to be of size 2*BUF
6. Declare the in buffers to be of size 2*BUF+BUF (2nd BUF for HIST)
7. Modify the two SIO_create's and all 6 SIO_issue's to use a size arg of 4*BUF
8. In the for loop that primes the history buffer reverse the locations of pIn and pPriorIn.
9. Build, load, and test the code. Note the CPU load in debug and release with the filter running and bypassed. How do these numbers compare to when buffers were on-chip and off-chip with cache disabled?

B. Experiment With Larger Buffer Sizes

in proc.h:

1. Add a new define of symbol N, set to 1
2. Add a define of symbol BUFF set to 128
3. Modify the define of BUF to be BUFF*N
   in proc.c
4. Modify the data alignments to be of size BUFF
5. Build, load, and test the code. Verify that with N=1 the results are the same as that of part A
7. Try some other values of N to see how performance is affected

Question: was internal memory the fastest version of all? Why?