

UT699 L1 Data and Instruction Cache Organization

Table 1: Cross Reference of Applicable Products

Product Name:	Manufacturer Part Number	SMD #	Device Type	Internal PIC Number:
UT699 32-bit Fault-Tolerant SPARC V8/LEON 3FT Processor	UT699	5962-08228	01, 02	WG07

* PIC = Product Identification Code

1 Introduction

Cache memory is an important element in microprocessors. In the UT699, each instruction and data access from external memory can take up to three clock cycles during random accesses and two clock cycles during burst instruction fetches. Accesses to cache memory in a processor such as the UT699 take only a single clock cycle. Microprocessor designers usually place cache memory on the same die as the central processing unit in order to achieve this fast access time.

During code execution, the processor fetches instructions and data from external memory and stores it into on-chip cache memory. Subsequent accesses to cached instructions or data will then take only a single CPU clock cycle per access. This results in higher system performance as a processor utilizing cache requires fewer clock cycles to execute code as the same processor without cache.

This application note explains the cache organization of the UT699, how the UT699 determines cache addresses, and the use of cache tags. Finally, Section 6 provides assembly code examples that the software programmer can utilize to access cache data and tags.

2 Cache Organization

The UT699 Leon 3FT microprocessor has 8kB of L1 instruction cache and 8kB of L1 data cache. Both cache units are organized as two-way, set associative, resulting in a logical configuration of 2x4kB for both instruction cache and data cache. The instruction cache is organized as 128 lines with 32 bytes per line for each set. The data cache is organized as 256 lines with 16 bytes per line for each set. In the event of a cache miss, i.e., a cache location does not contain valid data, the cache controller replaces an entire cache line using a least-recently used (LRU) replacement policy. The instruction and data cache are organized as shown in Tables 2 and 3.

Table 2: Instruction Cache Organization

Set	Line	Byte							
		31	30	...	4	3	2	1	0
0	0	31	30	...	4	3	2	1	0
0	...	31	30	...	4	3	2	1	0
0	127	31	30	...	4	3	2	1	0
1	0	31	30	...	4	3	2	1	0
1	...	31	30	...	4	3	2	1	0
1	127	31	30	...	4	3	2	1	0

Table 3: Data Cache Organization

Set	Line	Byte			
0	0	15	...	1	0
0	...	15	...	1	0
0	255	15	...	1	0
1	0	15	...	1	0
1	...	15	...	1	0
1	255	15	...	1	0

3 Cache Addresses

A unique address identifies each cache location. Accesses to either cache data or cache tags make use of these addresses. Section 5 explains the use of cache tags and their relationship to external addresses. Instruction and data cache addresses are word aligned. Instruction cache addresses range from 0000_{16} to $0FFC_{16}$ for set 0, and from 1000_{16} to $1FFC_{16}$ for set 1. Data cache addresses range from 0000_{16} to $0FFC_{16}$ for set 0, and from 1000_{16} to $1FFC_{16}$ for set 1. Since cache addresses are always aligned on 32-bit word boundaries, they must end in 00_{16} , 04_{16} , 08_{16} , or $0C_{16}$.

Table 4 shows an example of the addresses for the words in line 2 of set 1 of the instruction cache. For example, instruction cache address 1040_{16} is the address of word 0 of line 2 of set 1 of the instruction cache.

Table 4: Logical Representation of Instruction Cache Address

Cache Address	Set	Line	Word	“00” ¹
1040_{16}	x x x 1	0 0 0 0 0 1 0	0 0 0	0 0
1044_{16}	x x x 1	0 0 0 0 0 1 0	0 0 1	0 0
...	x x x 1	0 0 0 0 0 1 0	...	0 0
$105C_{16}$	x x x 1	0 0 0 0 0 1 0	1 1 1	0 0

Table 5 shows an example of the addresses for the words in line 2 of set 1 of the data cache.

Table 5: Logical Representation of Data Cache Address

Cache Address	Set	Line	Word	“00” ¹
1020_{16}	x x x 1	0 0 0 0 0 0 1 0	0 0	0 0
1024_{16}	x x x 1	0 0 0 0 0 0 1 0	0 1	0 0
1028_{16}	x x x 1	0 0 0 0 0 0 1 0	1 0	0 0
$102C_{16}$	x x x 1	0 0 0 0 0 0 1 0	1 1	0 0

Notes:

1. The two least-significant bits for both instruction and data cache addresses are always “00”, indicating word alignment.

4 Data Caching

The following section provides an example of how external data is stored in cache memory, demonstrates the case where two external addresses are mapped to the same cache location, and explains how cache sets are used.

Each set of the data cache contains 4096 bytes, or 1024 words, of cache memory that map to the entire 1GB external address space. Therefore, each individual cache location maps to 256k locations in external memory. Conversely, there are 256k locations of external address locations that map to a single location in cache memory. Now consider the case where two variables are written to external data, the first to address 40002000_{16} and the second to address 40003000_{16} . Both variables are aligned on a 4kB boundary, which is the size of each data cache set. Therefore, they necessarily map to the same cache location. Specifically, they both map to the data cache at address 0000_{16} . This is shown in Figure 1 below.

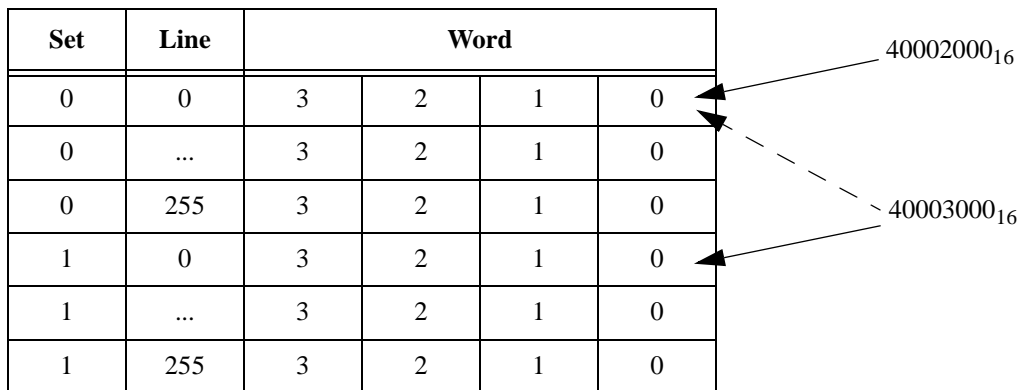


Figure 1. Example of External Data being Written to Cache

In this example, the first write to external address 40002000_{16} results in a write to cache location 0000_{16} , which is the first word of the first row of set 0. The valid bit for this cache location will be set, indicating that the cache location contains valid data. Valid bits are discussed further in Section 5. Next, data is written to external address 40003000_{16} . If the valid bit for cache location 0000_{16} were not set, this would result in a write to that cache location. However, since the valid bit is set, the write occurs to cache address 1000_{16} , which is the first word of the first row of set 1.

The cache controller uses a least-recently used (LRU) replacement policy. This means that a subsequent update to external memory on the same 4kB boundary results in a write to cache location 0000_{16} , assuming location 1000_{16} was the most recently accessed location. The cache data will be overwritten with the new data.

5 Cache Tags and Data

Each cache memory location has an associated cache data and a cache tag. The cache tag of a particular cache location contains information that identifies the address of the associated data in external memory. The cache data of a particular cache location contains the data corresponding to the data in external memory. Refer to the tag layouts in Figures 2 and 4. These figures show the fields of the instruction and data cache tags. The actual physical layout of the cache tags is explained in Section 2.6.3 of the *UT699 Functional Manual*. The ITAG and DTAG fields contain the most-significant 20 bits of the address of the data in external memory. The least-significant 12 address bits directly correspond to the cache address and are used to access cache tags and data using the load and store instructions `lda` and `sta`. The IVAL and DVAL fields identify whether or not the corresponding word in a cache line is valid. A '1' indicates that the word is valid, and accesses to the data or instruction at that address result in a valid cache hit. **Note:** The valid bits are shared with all cache tags for a given cache line. For more information on the instruction and data tag layouts, please refer to the *UT699 Functional Manual*.

The cache data fields are represented in Figures 3 and 5. These are 32-bit fields that contain the same data as the referenced address in external memory when the cache is valid, i.e., the valid bit is set for that cache location.

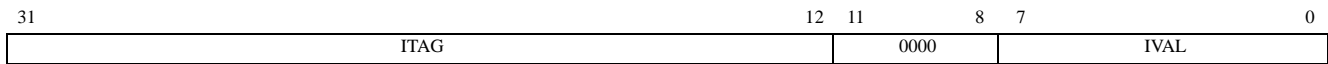


Figure 2. Instruction Cache Tag Layout

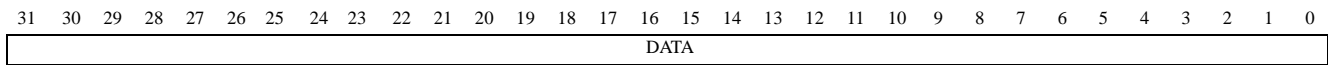


Figure 3. Instruction Cache Data Layout

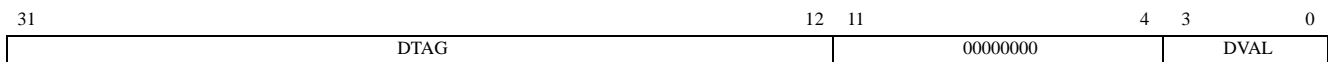


Figure 4. Data Cache Tag Layout

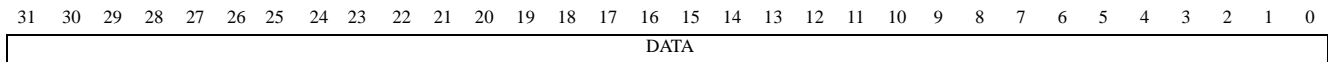
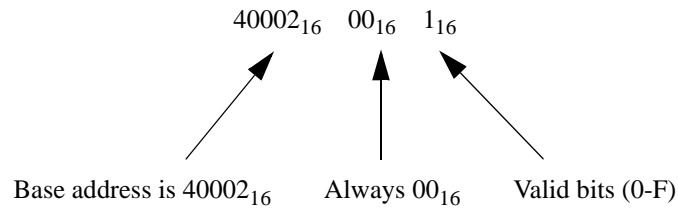


Figure 5. Data Cache Data Layout

Consider the previous example in Figure 1 of a write to external memory at address 40002000_{16} . It is assumed that prior to the write, the entire cache line does not contain valid data, i.e., the DVAL field is 0000_2 . Following the write to external memory, the data cache tag contains the following information (in hexadecimal):



To reconstruct the external address from the cache tag, the DTAG or ITAG field is concatenated with the cache address. In this example, the DTAG field is 40002_{16} and the cache address is 0000_{16} . Therefore, the referenced external address is:

$$40002xxx_{16} + 0000_{16} = 40002000_{16}$$

Following this write, the data cache at location 0000_{16} contains the same data as external address 40002000_{16} .

6 Accessing Cache Memory Using Alternate Space Identifier (ASI) Instructions

Accesses to the cache tags and cache data are handled automatically by the LEON 3FT core. However, they can be accessed using `lda` and `sta` instructions. These commands are similar to the load and store instructions `ld` and `st`, except that they access memory in an alternate memory space using an alternate space identifier (ASI). The following table shows the ASI usage for the UT699 microprocessor.

Table 6: ASI Usage

ASI	Usage
01 ₁₆	Forced cache miss
02 ₁₆	System (cache control) registers
08 ₁₆ , 09 ₁₆ , 0A ₁₆ , 0B ₁₆	Normal instruction and data access
0C ₁₆	Instruction cache tags
0D ₁₆	Instruction cache data
0E ₁₆	Data cache tags
0F ₁₆	Data cache data
10 ₁₆	Flush entire instruction cache
11 ₁₆	Flush entire data cache

For example, to access the data cache tag and cache data at a particular cache location, the programmer must use ASI 0E₁₆ and 0F₁₆ with an `lda` and `sta` instruction using inline assembly code. The most efficient way to access memory in the standard or an alternate memory space is to create inline assembly procedures called as C routines. The following four functions show examples of inline assembly code. The first two show how to perform a store and load operation in standard memory space.

```
inline void storemem(int addr, int val)
{
    asm volatile (" st %0, [%1] "           // store val to addr
                 :                          // output
                 : "r" (val), "r" (addr)    // inputs
                 );
}

inline int loadmem(int addr)
{
    int tmp;                               // used for returned value
    asm volatile (" ld [%1], %0 "          // load tmp from addr
                 : "=r" (tmp)             // output
                 : "r" (addr)             // input
                 );
    return tmp;
}
```

The next two functions are used to read the values of the data cache tags and data cache data in alternate spaces $0E_{16}$ and $0F_{16}$, respectively.

```
inline int loadmem_asi_0e(int addr)
{
    int tmp; // used for returned value
    asm volatile (" lda [%1] 0x0e, %0 " // load tmp from addr at ASI 0x0e
                  : "=r" (tmp) // output
                  : "r" (addr) // input
                  );
    return tmp;
}
```

```
inline int loadmem_asi_0f(int addr)
{
    int tmp; // used for returned value
    asm volatile (" lda [%1] 0x0f, %0 " // load tmp from addr at ASI 0x0f
                  : "=r" (tmp) // output
                  : "r" (addr) // input
                  );
    return tmp;
}
```

We can now make use of our inline assembly routines using a C function call. An example is the following write to data memory at locations 40002000_{16} and 40003000_{16} in standard memory space using the following C code:

```
storemem(0x40002000, 0x55555555);
storemem(0x40003000, 0xaaaaaaaa);
```

In the event of a flushed data cache line or a cleared valid bit at cache location 0000_{16} , both physical memory locations would map to data cache address 0000_{16} , i.e., word 0 of line 0 of set 0. However, this example shows that the first store operation results in the writing of data 55555555_{16} to set 0, with the second store operation writing data to set 1. The C functions and resultant returned values are shown below:

```
dcache_data = loadmem_asi_0f(0x0000);
dcache_tag = loadmem_asi_0e(0x0000);
```

The instruction passes the 12-bit cache address as a function parameter. The first function returns the cache data 55555555_{16} . The second function returns the data cache tag 40002001_{16} . The five most-significant hex digits of the tag indicate the upper 20 address bits of the data stored in physical memory space. The least-significant hex digit corresponds to the valid bits for the cache line. In this example, the value of '1' in the least-significant digit indicates that word 0 is now valid as a result of the first storemem operation.

The first storemem operation resulted in an update of the data cache memory at location 0000_{16} . Therefore, any access to data in physical memory at an address with the same three least-significant hex digits results in either a replacement at cache location 0000_{16} , or the data being written to set 1 at address 1000_{16} . Since only cache location 0000_{16} has been updated, the

data will be written to set 1. To illustrate this, the data cache data and data cache tag at cache location 1000_{16} are accessed using the following instructions:

```
dcache_data = loadmem_asi_0f(0x1000);  
dcache_tag = loadmem_asi_0e(0x1000);
```

These instructions return values of $AAAAAAAA_{16}$ and 40003001_{16} for the data cache data and tag, respectively, showing that the data was stored in set 1. As before, the '1' in the least-significant digit of the data cache tag indicates that the first word in the cache line is valid.

Note: Diagnostic accesses to instruction cache ($ASI\ 0C_{16}$ and $0D_{16}$) fail unless the instruction cache is disabled in the cache control register.

7 Conclusion

Updates and accesses to data and instruction cache during the execution of application code are automatically handled by the LEON 3FT processor core logic. However, the contents of the cache tags and data are readily available with memory accesses using alternate space identifier (ASI) instructions. This can be particularly useful during code debug when confirmation of cache accesses is required or to compare performance in a system where cache could be either enabled or disabled.