

Analysis of C6xxx Memory Interface And Math Capabilities

DSP Systems, Inc
September 2002

BACKGROUND

Instruction Access

The C6x family of processors fetch instructions in “fetch packets” of 32 bytes each. This means that each time the processor fetches an instruction it must read 32 bytes of memory.

The C671x can fetch an entire 256-bit fetch packed from internal L2 SRAM in a four cycles. If the fetch packet is in external RAM, the processor is stalled until the 64 bytes (L1 cache line) are read from the external interface. Theoretically, this takes:

$$6 + (16 \times 4)2 = 134 \text{ CPU cycles if the external RAM is 4 ECLK cycles}$$

This represents six cycles to setup the external fetch that is done by a DMA controller. Then the external accesses of 16 32-bit words times four ECLK cycles (CPU/2) each. If SBSRAM is used the packet fetch would take about 80 cycles. I should note that it could take substantially longer if L2 is used as cache. This is due to the cache line allocation and coherency check plus possible need to write back the newly allocated cache line to external RAM.

^This translates to about three to six million program fetches per second

The C671x and C641x processors do not have direct CPU access to external program memory, and all fetches are done using DMA. This is not true of the C6701 or C6201.

Data Access

The C6x family of processors fetches data, generally, in 4 byte quantities. Thus each time the processor needs data it must “stall” or wait for four bytes to be read. The C671x can fetch data from internal SRAM in four cycles. If the CPU needs data that is not in the L1 cache or internal SRAM is will stall until the data is delivered from the external interface.

If caching is enabled, the fetch will be a complete cache line (32 Bytes for L1 miss or 128 bytes for L2 miss) but the necessary data will be delivered to the CPU early in the line fetch. Thus the CPU only stalls for one word or so, but the bus is blocked for the time it takes to read either 32 or 128 bytes.

If caching is disabled the controller reads only the necessary bytes (usually four). This will theoretically, stall the CPU for 16 cycles if the external RAM is four cycle RAM.

This means that all accesses to any external device (either I/O or RAM) by the CPU will take 16 cycles each. This translates to a CPU transfer rate of about 60 Mbytes/sec.

Unfortunately, the theoretical minimum is not even close compared to the time it actually takes for an external memory read on the C671x or C641x processor. This will be discussed in detail later in this note.

DMA Transfers

Large DMA transfers moving data between external devices and internal RAM will take at best

1 CPU + 4 EMIF cycles (17 CPU cycles)

This equals 40 ns on the C67xx and 28 ns on the C64xx. Large external-to-external DMA moves will take 8 EMIF cycles at best. The C67xx allows EMIF = CPU/2 while the C64xx allows the EMIF = CPU/4 (or /6). This translates to transfer rates as shown below:

	<u>C67</u>	<u>C64</u>	<u>C64 (64-bit)</u>
Internal to external	100 MB/s	140 MB/s	280 MB/s
External to external	56 MB/s	74 MB/s	148 MB/s

ANALYSIS

The Problem with Cache

When a region of external memory is cached, a cache miss to a location in that region is very expensive. If L2 cache is all SRAM and there is an L1 miss, the controller fetches an entire L1 cache line. If L2 cache is enable and a cache miss occurs the controller fetches an entire L2 cache line. In either case the external memory interface is occupied for a minimum of 80 CPU cycles to over 150 CPU cycles or worse if the line is dirty. Clearly we should turn off as much caching as possible. This means the L2 should be all SRAM. The C671x and C641x processors do not allow L1 cache freeze but external memory caching can be disabled using the MAR registers. Power-up default is no external caching.

Real Cache Issues

The C671x and C641x processors do not have direct CPU access to external memory so all external accesses are done using EDMA transfers. This means that access to a single external register is inefficient as the CPU has to signal the EDMA controller, which in turn schedules the read or write, and returns the data.

Using an assembly routine we tested several external I/O operations:

Case 1

```
STW  .D2T2      B4, *B14(4)    (read / read)
STW  .D2T2      B2, *B14(44)
```

Case 2

```
STW  .D2T2      B4, *B14(4)    (read / write)
LDW  .D2T2      *B14(44), B2
```

Case 3

```
LDW  .D2T2      *B14(4), B2    (write / write)
LDW  .D2T2      *B14(44), B4
```

We used an oscilloscope to measured the bus timing between operations for each case and found the following:

Read	324 ns each	(48 cycles)
Read / Write	410 ns for both	(62 cycles)

Write	119 ns each	(18 cycles)
-------	-------------	-------------

The same code on a 200 MHz C6201 took:

Read	130 ns	(26 cycles)
Read/Write	160 ns	(32 cycles)
Write	65 ns	(13 cycles)

The C6711 and C6713 can do non-DMA I/O at a rate of about 12 Mbytes/s for reads and 32 Mbytes/s for writes.

These results can be verified using the C program at the end of the note and hand optimizing the assembly output. In most cases, the C code is optimized sufficiently to use it as is. All benchmarks were performed using no BIOS or chip support libraries. The executable code was under 7Kbytes. These results can also be verified using the Altera part and signal tap.

Interrupts

Even a simple ISR will require several reads and a write to be of much use. Using the C671x timing above this routine would take over 700ns to complete. Add to that the overhead of saving and restoring registers the ISR could take over 1us.

TRADEOFF

C641x

The C64xx has 16 Kbytes of L1P and L1D cache compared to 4 Kbytes each on the C67. The C64 also has 1 Mbytes of L2 RAM compared to 64 Kbytes on the C67. Clearly these larger internal RAM blocks will allow the CPU to use internal program and data fetches to a greater extent. It is clear that we must minimize the amount of external program fetches. Even using SBSRAM the penalty for external program fetches is huge.

When data must be fetched from external memory the "pipelining" ability of the C64xx will make retrieval a bit faster. Further, the C64xx has a 64-bit wide EMIF allowing twice as much data or code to be fetched per I/O operation. An external read on the C64 takes:

The C64xx has a 600 MHz CPU clock and programmable ECLK for external I/O. This means that it will execute instructions from internal RAM at over twice rate of the C671x. The C641x also supports a variety of 64 bit Instructions and has a large number of 64-bit registers.

Floating Point Math

We have run several floating point benchmarks to test execution speeds between a C6711 and C6201. The C6201 floating point math was done using the rts.lib that implements multiplication, division, addition, and subtraction using C coded functions; specifically, `_fltif`, `_fixif`, `_mpyf`, `_divf`, `_addf`, `_subf`. No care at all was taken to optimize these library functions. I do not think double precision floating point is available.

We recorded the number of timer clocks it took to do the four test loops 1000 times each. The results shown below are the actual number of timer ticks per loop.

	<u>C711 (dsk)</u>	<u>C6201 (200 MHz)</u>
Multiply	5,259	27,752
Add	5,258	30,001
Divide	47,025	44,752
Subtract	5,259	37,001
Cos		44,757

It might be possible to reduce this difference by writing or buying optimized floating point routines for the integer processor.

C6701 and C6201

The C6701 will out perform the C6711 and C6713 by a factor of two when doing external I/O operations. The floating point performance seems to scale with clock speed.

CONCLUSION

The DI DSP programmers must reevaluate their approach. You must write code to minimize size and maximize the use of internal RAM. All pipelined processors are slow to fetch data from external mapped I/O space, but it is essential to minimize program fetches from external RAM.

Using DSP Bios and the chip support libraries bloats the code and adds unnecessary complexity.

Addendum

The concepts outlined above are true for all modern cached, pipelined processors including Intel, PowerPC, SHARC, and probably any other processor you can name. CPU instruction execution rates are not what drive this task, it is input and output data rates we need to be concerned with. The challenge for us it to write smart code and do as much processing as possible in the FPGA. This would include FPGA based I/O state machines.