HUNT ENGINEERING
Chestnut Court, Burton Row,
Brent Knoll, Somerset, TA9 4BP, UK
Tel: (+44) (0)1278 760188,
Fax: (+44) (0)1278 760199,
Email: sales@hunteng.demon.co.uk
URL: http://www.hunteng.co.uk

Data unpacking techniques for C6000 systems

Introduction

In real C6000 systems such as those being built with HERON modules, it is necessary to connect the processors to "real world" data via devices like A/Ds and D/As. The processing power of the C6000 makes it sensible to process multiple channels of data per processor, which leads to a need for multi-channel I/O devices such as the HEGD9 and HEGD11 A/D modules. The data from the channels is then interleaved into a single communication stream – which brings additional challenges to the processor.
This paper looks at various ways that the C6000 can deal with this data.

Why use interleaving?

It does not make sense to use the C6000 processor core to poll for I/O data and copy the I/O data, a DMA engine should really be used for this task, freeing the processor core for processing of the data. With an I/O device like the HEGD11, which has 12 channels of A/D, it might seem sensible to provide the data to the processor through 12 separate FIFOs each of which is read using a DMA. In fact the C62xx processor core has only 4 DMA engines, so this would not be possible. In a real system DMAs would also be needed for other communications – to and from other processors, I/O nodes, or the host machine. It is also quite likely that a user may wish to claim a DMA for their own use. Thus it is not sensible to use separate DMAs and FIFOs. Feeding all channels of the data through a single FIFO, which can be read using a single DMA is the only way forward. I/O nodes like the GDIO modules use a single FIFO to place all channels of data, along with a marker that identifies which channel the data is associated with.

Basic C code

The original examples that HUNT ENGINEERING provided for the HEGD9 was a general purpose C routine, that took each data item from the HEGD9, and tested the channel marker. The data could then be added to the data array for the channel indicated by the channel marker. It was necessary to make this test for every data item, as the HEGD9 had a mode where all of the channels of the module could be clocked independently. This allowed the received data stream to contain data from the active channel in any order and any combination.
Testing of the original C routine showed that functionally it was working, but the execution on a group of 512 data samples was taking 380us on a 200Mhz C6201! This works out that we are able to unpack (but not perform any processing !) at a sample rate of 512 per 380us which is 1.35Mhz.
This immediately made us search for other methods of unpacking…..

Code optimisation

The first thing that was tried was to use different programming styles in the C code, using if, if then else, or switch statements made only minor differences to the time.
Using the optimiser options of the C compiler however gave astonishing results. The same C code that was taking 380us actually takes 24us when compiled with the "–o3" option. While this is an impressive improvement this only equates to a maximum sample unpacking rate of 21Mhz. Given that the HEGD9 is capable of generating 56Msps this result is still not good.

Channel ordering

The time taken by the bulk of the code is due to the need to make a decision about each individual data item in the data stream. If the data is guaranteed to be in a fixed order, even if we have to search for the first channel in the data we have received, we can remove the need for a decision on every data point.
Using an I/O module that guarantees the data ordering allows us to simply search for the first channel and then assume that every nth data item will be for that channel.
Understandably this makes sorting the data significantly easier. C code can be written to perform the sorting of such an array, but it is hard to control other system parameters such as interruptability.
Once the data is in a known order however there are some other techniques that can be used….

DMA sorting

The C6000 DMA engine can be used for sorting data. This is actually aimed at telephony packets such as those that can be received via the McBSP of the C6000, but it can equally as well be applied to sorting data stored in memory. The DMA engine has the concept of transferring a data packet, as a series of "frames". The Source and destination pointers can be incremented by a programmable number of data items, and the data items can be byte, half word and long. At the end of a frame there is a different "increment" that can be applied to the source, and this is how the DMA can be used to sort.

A source array that has data from several channels can be stepped through, incrementing each time by the spacing between the channels. Thus the data from the first channel can be read from the array, in order. At the end of the frame the source increment can be set to step back to the first item from the second channel. Using a new frame for each channel the data can be completely sorted in one DMA operation.

When we try this on our 512 data points from an HEGD9, we measure a time to sort of 16us. At first this looks good, but when you calculate how many cycles per data point this is, it works out at about 6.5. This seems a strange result, but it can be partially explained by the fact that the DMA subsystem can make only one memory access per cycle. This means that reading a value and writing a value can be no better than 2 cycles. The remainder of the cycles are present because the internal pipelining of the memory accesses is not efficient when making one read – one write – one read etc. In fact it is surprising to know that making the DMA to or from external memory can achieve the 2 cycles per data point. This means that sorting from external to internal memory or vice versa is more efficient than sorting totally in internal memory! This could be used to our advantage if we need to move the data onto or off of the chip.

There is another issue -- the data is simply sorted by the DMA, and in the case of our HEGD9 data, there are still channel identifier bits in the data. To remove these bits by masking, with a C routine takes 61us with no optimisation, but 1.8us with –o3.

Sorting with a Linear Assembly function.

It is quite easy with the Linear Assembler for the C6000 to write an assembler function to perform the sorting. In this assembler we can perform a load and a store in each processor cycle. The processor can actually perform more instructions in a cycle than is needed by the simple copying, so it is possible to utilise some of the other execution units to do the bit-mask of the data, and perhaps even to perform some scaling, sign extension or other constant operation. The penalty for these operations is limited to an extra pipeline delay in the sorting – there is no loss of bandwidth.

Our GD9 example of such a function takes 7us to sort the same 512 data items, which is actually less performance than the predicted cycle per sample, but when we use the –mtw option to build the code this time collapses to between 3.5 and 4us. The variation can be explained by the fact that the HERON-API is being used to capture the next data buffer in parallel with this sorting operation. The predicted cycle per sample operation would actually take 2.56us, but the code must be made interruptible in order to allow the HERON-API to continue to handle the I/O.

Using the linear assembler function in this way gives us an effective sorting rate of 128 Million data points per second – a much more respectable rate!

Conclusions

The first thing that becomes clear is that a multi-channel I/O board should guarantee the ordering of data unless there is a very good reason not to.

It is interesting that the best solution at first sight (the DMA) does not come out to be the best one, especially given the sometimes bizarre behaviour of the DMA (such as copying to and from internal being slower than to or from external memory).

The optimisation settings of the C compiler makes a vast difference, and from this it can be deduced that the highest optimisation setting should always be used, unless there is good reason not to.

For the linear assembler the –mtw option is very important, but as this option tells the assembler that there are no bad aliases to the same memory address, the code must not contain such aliases.

Summary results

C sorting	Optimised C sorting	DMA sorting	Linear assembler sorting	Linear assembler sorting with -mtw
148	9	6.5	2.73	1.36

Number of processor cycles per data item sorted, actually measured while I/O in progress so interrupts and I/O DMAs are in progress.