Measuring Springboard Throughput

Application Note AN-19	Revision 1.00	Document No. 80-0159-00

Table of Contents

List of Acronyms	
DRAM	Dynamic RAM (Random Access Memory)
MHz	Megahertz. 1,000,000 cycles per second.
ms, µs, ns	Time measurements: millisecond (1/1,000), microsecond (1/1,000,000) and nanosecond (1/1,000,000,000).

1. How Fast Is Springboard?

The *theoretical* maximum transfer rate is a function of the bus width and clock speed. If bus cycles are placed back to back, the Motorola CPU will complete a single bus cycle (e.g. executing a `NOP` from Springboard memory) in 4 clock cycles. All data transfers are done in 16 bit increments. Therefore a 33MHz system will transfer 33/4 million words per second resulting in 8.25Mwords/second, or 16.5Mbytes/second.

The *practical* limits are lower. Here is a summary of cautions when measuring data throughput on the Springboard bus:

- Since there is no instruction cache or separate memory bus, memory that an application is executing from shares the bus with Springboard. Instruction fetches will take bus time along with transfers to a Springboard device.
- Wait states on chip selects for internal memory and Springboard will influence transfer rates.
- There are two Springboard chip selects. Wait states for these two chip selects must be programmed as a pair. The slowest access time between the two must be used.
- When a module is inserted the Springboard slot is configured with the maximum number of wait states[1]. A ROM token (access time) is later retrieved from the Springboard module ROM header and the Springboard access time is reprogrammed. A handheld may have a minimum wait state requirement of greater than 0, even if the ROM token specifies 0ns access time explicitly.
- The CPU has a pre-fetch capability which allows it to complete computational tasks while fetching the next instruction.

1 The DragonBall EZ process at 16.58MHz has a maximum of 6 60ns wait states. The DragonBall VZ processor at 33Mhz has a maximum of 13 30ns wait states.

Here is a listing of sample benchmarks using the optimized code developed at the end of this application note. It is provided for reference purposes only; developers should conduct tests appropriate to their product and can then make design decisions based on those practical results.

	120ns Springboard Access Time	**0ns Springboard Access Time (Requested)**
Visor/Visor Deluxe	~3.2Mbytes/s	~3.5Mbytes/s
Visor Prism	~4.0Mbytes/s	~5.2Mbytes/s
Visor Platinum	~4.0Mbytes/s	~5.2Mbytes/s
Visor Edge	~4.6Mbytes/s	~6.4Mbytes/s

Visor/Visor Deluxe uses a 16.58MHz CPU. All other handhelds listed use a 33MHz CPU. The test demonstrates an optimized series of word reads, word writes and pointer increment in assembly language. Read and write timing is symmetrical. The table above shows that zero access time is requested, however, the CPU/OS will always impose a minimum period.

2. Benchmark in C

To begin, let's take a look at benchmarking in C since the target application is also likely to be written in this language. Here's the code fragment of a routine that writes data to the Springboard slot.

```
iterations = 0;
ticks1 = TimGetTicks();
while (true)
        {
        for (i = 0; i < 0x5000; i++)
```

```
                    {
                    // Write to Springboard slot pointed to by begin.
                    *(volatile UInt16 *)begin = 0x5a;
                    iterations++;
                    }

            ticks2 = TimGetTicks();
            if ((ticks2 - ticks1) >= sysTicksPerSecond)
                    {
                    break;
                    }
            }
    ips = sizeof(*begin) * iterations * (ticks2 - ticks1) / sysTicksPerSecond;
```

A Visor Prism (33MHz, VZ processor) with a module (Diagnostic Module) requesting 120ns access time will generate the following results:

• Read Throughput	1.17Mbytes/s
• Write Throughput	1.085Mbytes/s

Let’s take a look at the code this C routine produces. This code is a tight loop that writes out words to the Springboard slot and checks to see if one second has elapsed. The code is optimized such that the time is checked only once every 0x5000 (hexadecimal value; equivalent to 20,480 decimal) write operations to reduce overhead. This core loop of 0x5000 iterations is shown in the disassembly.

```
+$005C  00093FDE   _TimGetTicks                     ; $100A4506   | 4E4F A0F7
+$0060  00093FE2   MOVE.L    D0,D3                                | 2600
+$0062  00093FE4   CLR.W     D1                                   | 4241
+$0064  00093FE6   MOVE.W    #$005A,(A2)            ; 'Z.'        | 34BC 005A
+$0068  00093FEA   ADDQ.L    #$01,D4                              | 5284
+$006A  00093FEC   ADDQ.W    #$01,D1                              | 5241
+$006C  00093FEE   CMPI.W    #$4FFF,D1              ; '.O'        | 0C41 4FFF
+$0070  00093FF2   BLS.S     PrvStartTest+$0064     ; 00093FE6    | 63F2
+$0072  00093FF4   _TimGetTicks                     ; $100A4506   | 4E4F A0F7
```

The `MOVE.W #$005a,(a2)` instruction moves one word of data to the Springboard slot. However, the next four instructions is logic for the loop and result calculation. Most of the processing time is spent in the overhead code (4 of 5 instructions).

3. Instruction Execution Relative To Bus and Clock Cycles

Motorola documents execution times for each instruction. From the previous disassembly, most of the time is spent in the following loop:

```
MOVE.W    #$005A,(A2)              ; 'Z.'        | 34BC 005A    // write word         12
ADDQ.L    #$01,D4                                | 5284         // iterations++       8
ADDQ.W    #$01,D1                                | 5241         // i++                4
CMPI.W    #$4FFF,D1                ; '.O'        | 0C41 4FFF    // loop test          8
BLS.S     PrvStartTest+$0064       ; 00093FE6    | 63F2         //                    10
                                                                                      ---
                                                                      (clock cycles)  42
```

The number of clock cycles for each instruction is noted on the far left of the code.

The Motorola documentation indicates that a bus cycle takes four clock cycles. For example, fetching an instruction word from memory or writing a word to memory takes four clock cycles. Combining an instruction fetch with it’s execution produces the expected execution times above. Let’s make sure that makes sense...

On a 33Mhz system a clock cycle is 30ns. The DragonBall VZ processor specifications indicate that the setup time (address valid to chip select asserted) takes about a clock cycle (20ns). The chip select pulse width is two clock cycles plus wait states (60ns + wait states). The data hold and time for signals to de-assert prior to the next bus cycle is conceivably an additional clock cycle. That totals four clock cycles for each bus operation so the timing seems viable.

Our expected execution time for the above loop is 42 clock cycles, or ~794Kwords/second (1/(30ns * 42 cycles)). That gives us a reasonable maximum throughput of ~1.6Mbytes/second for a 33MHz system. That's not including loop overhead (~54/second), interrupt processing (e.g. timer ticks) and wait states on both internal memory and Springboard.

Let's see if we can tighten our benchmark and anticipated results.

4. Test One: Tighten C Loop/Minimize Overhead In Measurement

As a test let's try modifying the benchmark code such that we write 1,000,000 words and simply measure the elapsed ticks (1/100th second intervals in this case). Let's also tighten the loop as well to see if we can get closer to the theoretical throughput we calculated:

This C code:

```
ticks1 = TimGetTicks();
count = 1000000;
while (--count)
       *(volatile UInt16 *)begin = 0x5a;
ips = TimGetTicks()-ticks1;
```

Produces this assembly:

```
+$005A  0005D4F0   _TimGetTicks                        ; $100A4506   | 4E4F A0F7
+$005E  0005D4F4   MOVE.L    D0,D3                                   | 2600
+$0060  0005D4F6   MOVE.L    #$000F423F,D0             ; '..B?'      | 203C 000F 423F
+$0066  0005D4FC   MOVEQ.L   #-$18,D5                                | 7AE8
+$0068  0005D4FE   ADD.L     A6,D5                                   | DA8E
+$006A  0005D500   MOVE.W    #$005A,(A2)               ; 'Z.'        | 34BC 005A
+$006E  0005D504   SUBQ.L    #$01,D0                                 | 5380
+$0070  0005D506   BNE.S     PrvStartTest+$006A        ; 0005D500    | 66F8
+$0072  0005D508   _TimGetTicks                        ; $100A4506   | 4E4F A0F7
```

On a Visor Prism (33MHz) the result is an elapsed time of 126 ticks, or 1.26 seconds. That means that 1,000,000 words were written in 1.26 seconds. That's 2,000,000 bytes / 1.26 seconds = ~1,587,000 bytes/second which is much closer to our calculated maximum based on the previous logic. Let's see what else we can do...

5. Test Two: Tighter Measurement and Assembly Code

Let's take advantage of the built in timer in the DragonBall CPU. A prescaler is used such that a register is incremented every 6 clock cycles or 0.18μs (6 x 30ns on a 33MHz system). Let's change the prescaler to 0 so that the resolution is down to 30ns (on a 33MHz system). Let's also move to assembly language so that we can control the exact instructions that we're measuring.

Measure a single MOVE.W:

Here's a test that obtains the timer information, writes a word of data and reads the timer again.

```
MOVE.W (a0), d6       // Overhead to get timer. Expecting 8 cycles.
MOVE.W #$1234, (a1)   // Actual work. Expecting 12 cycles (not including wait states)
MOVE.W (a0), d7       // Overhead to get timer. Expecting 8 cycles.
```

This test took 28 clock cycles. That makes sense. We're expecting some additional delay due to wait states, however, any efficiencies gained by pre-fetches and sections of code executed before the register value is retrieved probably cancel out the wait state delay.

Let's try to minimize the overhead in this test and measure 128 consecutive MOVE.W instructions. The results are 2455-2462 clock cycles. The throughput is: 2455-2462 clock cycles = 19.18-19.23 clock cycles/word. Using the slower time of 19.23 clock cycles/word, that's 1.7M words/s or 3.4M bytes/s.

This method approaches a write speed of 3.4M bytes/second on a Visor Prism running at 33Mhz with a Springboard module set to 120ns access time.

6. Test Three: Optimize Assembly and Verify Results

It's clear that the throughput results are directly related to the efficiency of the code and that instruction fetches has considerable bearing on measuring Springboard throughput. To optimize the assembly code, we'll use the MOVEM.W instruction which can move words between multiple registers and memory in a single instruction. The advantage is that an instruction doesn't have to be fetched for each word move.

To verify the results we obtain from the DragonBall timer, we'll make use of the Springboard Reset* signal (asterisk denotes as active low signal). We'll assert the signal just prior to the test code and de-assert it immediately after. With this technique, we can then monitor the signal on an oscilloscope and compare the results to our timer based calculation.

Here's the in-line assembly code:

```
asm volatile ("
      | Store A5 and A6 on the stack, since GCC doesn't like us to say
      | that those are trashed (it can't handle saving/restoring them).
      | For reference: A5 is the globals pointer (used to access global
      | data), A6 is the frame pointer (used by convention to access
      | locals / parameters).
      MOVE.L  %%a5, -(%%a7)
      MOVE.L  %%a6, -(%%a7)

      | Store initial timer value on the stack so we don't waste a
      | register...
      MOVE.W  0xF608.w, -(%%a7)


      | 16 instruction pairs moving 13 dwords each = 832 bytes
      |
      | IMPORTANT: In reality, we would probably be copying to a fixed
      | address and _wouldn't_ need to increment the destination.  This
      | would save us 2 instruction reads per block, plus the LEA at the end!
      MOVEM.L (%0)+, %%a2-%%a6/%%d0-%%d7
      MOVEM.L %%a2-%%a6/%%d0-%%d7, (%1)

      MOVEM.L (%0)+, %%a2-%%a6/%%d0-%%d7
      MOVEM.L %%a2-%%a6/%%d0-%%d7, 52(%1)

      MOVEM.L (%0)+, %%a2-%%a6/%%d0-%%d7
      MOVEM.L %%a2-%%a6/%%d0-%%d7, 104(%1)

      MOVEM.L (%0)+, %%a2-%%a6/%%d0-%%d7
      MOVEM.L %%a2-%%a6/%%d0-%%d7, 156(%1)

      MOVEM.L (%0)+, %%a2-%%a6/%%d0-%%d7
      MOVEM.L %%a2-%%a6/%%d0-%%d7, 208(%1)

      MOVEM.L (%0)+, %%a2-%%a6/%%d0-%%d7
      MOVEM.L %%a2-%%a6/%%d0-%%d7, 260(%1)
```

```
        MOVEM.L (%0)+, %%a2-%%a6/%%d0-%%d7
        MOVEM.L %%a2-%%a6/%%d0-%%d7, 312(%1)

        MOVEM.L (%0)+, %%a2-%%a6/%%d0-%%d7
        MOVEM.L %%a2-%%a6/%%d0-%%d7, 364(%1)

        MOVEM.L (%0)+, %%a2-%%a6/%%d0-%%d7
        MOVEM.L %%a2-%%a6/%%d0-%%d7, 416(%1)

        MOVEM.L (%0)+, %%a2-%%a6/%%d0-%%d7
        MOVEM.L %%a2-%%a6/%%d0-%%d7, 468(%1)

        MOVEM.L (%0)+, %%a2-%%a6/%%d0-%%d7
        MOVEM.L %%a2-%%a6/%%d0-%%d7, 520(%1)

        MOVEM.L (%0)+, %%a2-%%a6/%%d0-%%d7
        MOVEM.L %%a2-%%a6/%%d0-%%d7, 572(%1)

        MOVEM.L (%0)+, %%a2-%%a6/%%d0-%%d7
        MOVEM.L %%a2-%%a6/%%d0-%%d7, 624(%1)

        MOVEM.L (%0)+, %%a2-%%a6/%%d0-%%d7
        MOVEM.L %%a2-%%a6/%%d0-%%d7, 676(%1)

        MOVEM.L (%0)+, %%a2-%%a6/%%d0-%%d7
        MOVEM.L %%a2-%%a6/%%d0-%%d7, 728(%1)

        MOVEM.L (%0)+, %%a2-%%a6/%%d0-%%d7
        MOVEM.L %%a2-%%a6/%%d0-%%d7, 780(%1)

        | Get the new timer value.  We'll just use D0
        MOVE.W  0xF608.w, %%d0

        | Compare the two measurements.  Note: if new is < old, we
        | must have got a 16-bit wraparound.  To fix this, we do
        | 32-bit math, adding 0x10000 if we detect a wrap.
        CLR.L   %%d1
        CLR.L   %%d2
        MOVE.W  (%%a7)+, %%d1
        MOVE.W  %%d0, %%d2

        CMP.W   %%d1, %%d2
        BHI            noWrap

        ADD.L   #0x10000, %%d2

noWrap:
        SUB.L   %%d1, %%d2

        | Store the result in srcP
        MOVE.L  %%d2, %1

        | Restore A5 and A6
        MOVE.L  (%%a7)+, %%a6
        MOVE.L  (%%a7)+, %%a5

        " :
  "=a" (srcP), "=a" (dstP) :
  "0"  (srcP), "1"  (dstP) :
                 "a2", "a3", "a4",
  "d0", "d1", "d2", "d3", "d4", "d5", "d6", "d7",
  "cc", "memory" );
```

The results that we obtain from this measurement are in the table located at the beginning of this application note. The Reset* signal verifies that the measurements are accurate with an error of approximately 5%. This is acceptable due to the slight delay (instructions executing) between obtaining the timer and toggling the Reset* line.

7. Source Documentation

"MC68000 Assembly Language and Systems Programming" by William Ford and William Topp. ISBN 0-669-16085-7

"C: A Reference Manual" by Samuel P. Harbison and Guy L. Steele Jr. ISBN 0-13-326232-4

Websites related to this subject include:

Motorola website: www.motorola.com
("Motorola EC000 (SCM68000) Core Processor User's Manual", "MC68VZ328 Integrated Processor User's Manual', "MC68EZ328 Integrated Processor User's Manual")

Handspring Website: www.handspring.com/developers/sw_dev.jhtml
(Code Samples section, GameSamples download)

Aaron Ardiri's Website: www.ardiri.com
(Burning and Cube3D projects demonstrate in-line assembly in both GNU and CodeWarrior)

Handspring GNU Tools Documentation:

If you've installed the Handspring GNU Tools using the default directory structure, here's where some information is regarding this topic:

C:\handspring\prc-tools\PalmDev\doc\index.html (Click on the "gcc" link)
In the section "Extensions to the C language family"

- "Assembler Instructions with C Operands" is helpful, in addition to the sample code
- "Constraints for asm operands" is helpful when review the sample code.

8. History

Date	Revision #	Description of changes
15 May 2001	1.00	Initial release.