Application Note

Accelerating CRCs on eSi-RISC with user-defined instructions

Introduction

This application note provides some practical examples of calculating a cyclic redundancy check (CRC) [1] on 16 and 32-bit versions of eSi-RISC, and looks at how the user-defined instruction extensions can provide a saving in power, computation cycles and reclaiming memory space.

The CRC is a common algorithm performed on microprocessors, but can be surprisingly compute intensive despite its apparent simplicity. Even on architectures well suited to bit manipulation and having extensive addressing modes such as eSi-RISC, there is still a high minimum cycle count. In this paper we examine computing a 16-bit CRC in its non-reflected form and a 32-bit CRC in its reflected form to cover the two typical versions encountered in practice. The reader is referred to [1] for a tutorial on the difference between reflected and non-reflected computation, but in simple terms it refers to whether a CRC operates on bits from LSB to MSB or vice-versa.

8-bit CRCs are still commonly used, for example in broadband ISDN [2] and robust compression for RTP, UDP and ESP IP headers [3], but in the majority of cases 16 and 32-bit CRCs are required to protect longer data payloads, and for this very reason they can end up consuming significant processor cycles. The chosen 16-bit CRC is commonly found protecting MAC frames, and the 32-bit CRC is the standard 4 octet IEEE 802.3 frame check sequence (FCS) used to protect an Ethernet MAC frame [4].

The eSi-RISC toolchain is based on the GNU suite and benefits from seamless integration within the Eclipse Integrated Development Environment (IDE). For clarity the algorithms are implemented in "C" using efficient loops and compiled for a Release build with –O2 optimization which gives a good balance between code size and execution speed. The program memory is arranged to fetch 32-bit instructions in a single cycle. The eSi-RISC compiler is very efficient at producing optimised code, often as good as hand crafted assembly language.

CRC-16

16-bit CRCs were common in older data link standards to protect frames of data sent through a serial port, and also for interfacing to floppy disk formats. These applications are still prevalent today in low complexity embedded applications but are starting to be replaced by IP encapsulation for more complex systems.

More recent common uses for 16-bit CRCs are in MAC layer error control for moderate length payloads. In particular DECT [5] and TETRA [6] both specify these at the MAC layer and are typical applications for embedded processors. Other low data rate wireless networking standards such as Bluetooth [7] and Zigbee [8] use a 16-bit checksum for the MAC payload frame check sum (FCS). Broadcast standards like Digital Radio Mondiale [9] have applied this checksum to their multiplex channels for error protection. Wired communication standards like USB 2.0 apply a 16-bit checksum over the data field prior to transmission.

We begin by looking at a variant of the CRC-16 CCITT algorithm, but the same principles apply to the many different implementations.

Although the CRC is often specified acting on individual bits of a data stream, in practice the algorithm is applied to an array of bytes. Where the number of bits is not a multiple of 8 then a few extra iterations at the end, operating on the excess bits, is still the most efficient method in software. This means that the algorithm can be accelerated by operating on 8-bits at a time. To achieve this requires building a table specifying the effect the polynomial would have on each of the 256 possible input bit combinations. The following code generates a table for the CRC-16 in unreflected form.

```c
uint16_t crc16Table[256];

// example unreflected form
void genCrc16Table(unsigned int numBits, uint16_t poly, uint16_t *table)
{
    uint16_t msb, symbol;
    unsigned int i, j, tableSize;

    tableSize = 1 << numBits;
    msb = 1 << 15;

    for(i = 0; i < tableSize; i++)
    {
        symbol = i << (16 - numBits);
        for(j = 0; j < numBits; j++)
        {
            if(symbol & msb)
                symbol = (symbol << 1) ^ poly;
            else
                symbol = symbol << 1;
        }
        table[i] = symbol;
    }
}

void main()
{
    genCrc16Table(8, 0x1021, crc16Table);     // CRC16-CCITT
    ...
}
```

Figure 1: Code for generating the unreflected CRC-16 look-up table

The resulting table is detailed in Appendix A for those interested readers. This table would typically be generated during an initialization phase of the microprocessor execution, in which case it occupies 512 bytes of available SRAM in the data space. Alternatively the table can be pre-generated and declared as *const* so it resides in the ROM space (FLASH memory for a standard FPGA development board). On a typical 16-bit processor this represents about 1% of the available data and program memory space.

The CRC iteration subroutine is given below. Note we have used unsigned int for efficient machine dependent variable size (16 and 32 bits respectively for a 16-bit and 32-bit processor architecture) and C99 [11] specific sizes where the word length is explicit. The crc16Table is a global const array in this example. The correct behaviour of a CRC is defined by its operation on the string "123456789". In this example the CRC result is 0x29B1 as expected.

```
uint16_t crc16(uint8_t *message, unsigned int N)
{
      uint16_t crc;

      crc = 0xffff;
      while(N--)
      {
            crc = crc16Table[((crc >> 8)^*message++) & 0xFF]^(crc << 8);
      }
      return crc;
}
```

Figure 2: Subroutine for calculating CRC-16 using the unreflected table method

On a 32-bit processor like the eSi-3200 the generated assembly code is as follows

```
0000014c <crc16>:
    14c:   3d68          mv     r10, r8
    14e:   e1ff 147f     l      r8, 65535
    152:   2496          bz     r9, 17e <crc16+0x32>
    154:   e000 1e92     l      r13, (gp+[48 <_interrupt10_vector>])
    158:   e00b 3448     sru    r11, r8, 8
    15c:   8e0a          lbu    r12, (r10+[0])
    15e:   e201 c5ff     and    r11, 0xff
    162:   e20b 3dcc     xor    r11, r11, r12
    166:   35a1          sl     r11, r11, 1
    168:   e00b 3e9b     add    r11, r13, r11
    16c:   958b          lh     r11, (r11+[0])
    16e:   3428          sl     r8, r8, 8
    170:   e208 3dc8     xor    r8, r11, r8
    174:   c4ff          add    r9, -1
    176:   e3ff c47f     and    r8, 0xffff
    17a:   c501          add    r10, 1
    17c:   2cee          bnz    r9, 158 <crc16+0xc>
    17e:   3882          ret
```

Figure 3: Assembly language listing for eSi-3200 implementation of CRC-16

Using the eSi-RISC profile capability we can generate the CRC for a message of 1024 bytes. This takes 16389 cycles, corresponding to 16 cycles per byte plus a one-time overhead of 7 cycles. The main iteration loop is highlighted in red above. Clearly the one line of C code hides many operations;

- two shifts
- a logical "and"
- two logical "exclusive or"
- a memory content fetch
- a pointer increment
- a table look-up

It is now instructive to see if this loop can be accelerated by a user-defined instruction. On eSi-RISC user-defined instructions are implemented in small hardware accelerators attached to the main processor and able to access the register file for up to 2 input arguments and 1 output argument. For simple logical operations like a CRC these correspond to very efficient hardware taking only a few gates and able to offload look-up tables that would otherwise occupy valuable RAM into logic.

EnSilica Limited, The Barn, Waterloo Road
Wokingham RG40 3BY, United Kingdom
Tel: +44 (0)118 3217 310
info@ensilica.com
3 of 17
© 2010 EnSilica Ltd, All Rights Reserved
Version 1.1
11/01/2010

The user-defined instructions have a "C" subroutine call prototype for simple integration into the code. The actual assembly language instruction is inlined after the arguments are resolved to avoid the overhead of an actual function call. For example the next listing shows how one might implement the CRC calling a user-defined instruction within the while loop.

```c
uint16_t crc16(uint8_t *message, unsigned int N)
{
    uint16_t crc;

    crc = 0xffff;
    while(N--)
    {
        crc = user0(crc, *message++);
    }
    return crc;
}
```

Figure 4: Subroutine for generating CRC-16 with user-defined instruction user0

The generated assembly code is given below, and the user-defined instruction is clearly visible.

```
0000014c <crc16>:
    14c:    3d68            mv      r10, r8
    14e:    e1ff 147f       l       r8, 65535
    152:    2489            bz      r9, 164 <crc16+0x18>
    154:    c4ff            add     r9, -1
    156:    8d8a            lbu     r11, (r10+[0])
    158:    c501            add     r10, 1
    15a:    e0e8 3c0b       user0   r8, r8, r11
    15e:    e3ff c47f       and     r8, 0xffff
    162:    2cf9            bnz     r9, 154 <crc16+0x8>
    164:    3882            ret
```

Figure 5: Assembly language listing for eSi-3200 implementation of CRC-16 with user-defined instruction

This code executes in 8196 cycles corresponding to 8 clock cycles per byte. Most of the clock cycles here are related to updating the pointer address and handling the while loop control variable. Implementing the CRC in user logic corresponds to a 2x speed-up which can be a significant saving for some applications.

Next we turn our attention to how the same code maps to a 16-bit processor like the eSi-1600. The generated assembly code for the same C routine of Figure 2 is given below

```
000000a2 <crc16>:
      a2:   3d68          mv    r10, r8
      a4:   e092 1694     l     r13, 18708
      a8:   147f          l     r8, -1
      aa:   2490          bz    r9, ca <crc16+0x28>
      ac:   8d8a          lbu   r11, (r10+[0])
      ae:   e00c 3448     sru   r12, r8, 8
      b2:   e20b 3e4b     xor   r11, r12, r11
      b6:   35a1          sl    r11, r11, 1
      b8:   e00b 3e9b     add   r11, r13, r11
      bc:   958b          lh    r11, (r11+[0])
      be:   3428          sl    r8, r8, 8
      c0:   c4ff          add   r9, -1
      c2:   e208 3dc8     xor   r8, r11, r8
      c6:   c501          add   r10, 1
      c8:   2cf2          bnz   r9, ac <crc16+0xa>
      ca:   3882          ret
```

Figure 6: Assembly language listing for eSi-1600 implementation of CRC-16

This code executes in 13317 cycles corresponding to 13 clock cycles per byte. The reason for the improved cycle count on a 16-bit machine is simply that the 32-bit machine has to mask off the lower 16-bits of the CRC, whereas a natural 16-bit machine doesn't need to.

The corresponding assembly listing with the user-defined instruction implemented is shown below

```
000000a2 <crc16>:
      a2:   157f          l     r10, -1
      a4:   2487          bz    r9, b2 <crc16+0x10>
      a6:   c4ff          add   r9, -1
      a8:   8d88          lbu   r11, (r8+[0])
      aa:   c401          add   r8, 1
      ac:   e0ea 3d0b     user0 r10, r10, r11
      b0:   2cfb          bnz   r9, a6 <crc16+0x4>
      b2:   3c6a          mv    r8, r10
      b4:   3882          ret
```

Figure 7: Assembly language listing for eSi-1600 implementation of CRC-16 with user-defined instruction

This optimised code runs in 7172 cycles corresponding to 7 clock cycles per byte, and benefits from the same cycle saving from avoiding a mask operation. We conclude that there's no loss in efficiency from running a 16-bit CRC on a 16 or 32-bit machine. User-defined instructions can give a 2x improvement in both architectures.

CRC-32

Next we look at performing the IEEE 802.3 MAC FCS [4] in both architectures. This checksum uses a reflected version of the table to simplify the main iteration loop. The table is efficiently generated as follows, and listed in Appendix A for the interested reader.

EnSilica Limited, The Barn, Waterloo Road
Wokingham RG40 3BY, United Kingdom
Tel: +44 (0)118 3217 310
info@ensilica.com

© 2010 EnSilica Ltd, All Rights Reserved
Version 1.1
11/01/2010

5 of 17

```
uint32_t crc32Table[256];

// example reflected form - poly should be reflected too
void genCrc32Table(unsigned int numBits, uint32_t poly, uint32_t *table)
{
      uint32_t symbol;
      unsigned int i, j, tableSize;

      tableSize = 1 << numBits;

      for(i = 0; i < tableSize; i++)
      {
            symbol = i;
            for(j = 0; j < numBits; j++ )
            {
                  if(symbol & 1)
                        symbol = (symbol >> 1) ^ poly;
                  else
                        symbol = symbol >> 1;
            }
            table[i] = symbol;
      }
}

void main()
{
      genCrc32Table(8, 0xEDB88320, crc32Table); // CRC32-IEEE802.3
      ...
}
```

Figure 8: Subroutine to generate reflected CRC-32 look-up table

This table occupies 1K bytes, which is quite a significant waste of RAM available to the microprocessor just to allow efficient CRC calculation.

The reflected CRC calculation is actually slightly simpler than the non-reflected case, since it avoids one of the shift operations. Correct operation is verified by operating on the message string "123456789" to give the expected result 0xCBF43926. The listing is given below for the most efficient implementation.

```
// reflected form - use reflected table too !!
uint32_t crc32(uint8_t *message, unsigned int N)
{
      uint32_t crc;

      crc = 0xffffffff;
      while(N--)
      {
            crc = crc32Table[(crc ^ *message++) & 0xFF] ^ (crc >> 8);
      }
      return ~crc;
}
```

Figure 9: Subroutine for calculating CRC-32 using the reflected table method

The eSi-3200 assembly language listing corresponding to this is given below

```
00000180 <crc32>:
    180:   157f        l      r10, -1
    182:   e000 1e86   l      r13, (gp+[18 <_debug_vector>])
    186:   1580        l      r11, 0
    188:   2493        bz     r9, 1ae <crc32+0x2e>
    18a:   8e08        lbu    r12, (r8+[0])
    18c:   e00b 3548   sru    r11, r10, 8
    190:   e20a 3d4c   xor    r10, r10, r12
    194:   e201 c57f   and    r10, 0xff
    198:   3522        sl     r10, r10, 2
    19a:   e00a 3e9a   add    r10, r13, r10
    19e:   a50a        lw     r10, (r10+[0])
    1a0:   c4ff        add    r9, -1
    1a2:   e20a 3d4b   xor    r10, r10, r11
    1a6:   c401        add    r8, 1
    1a8:   2cf1        bnz    r9, 18a <crc32+0xa>
    1aa:   e041 3d8a   not    r11, r10
    1ae:   3c6b        mv     r8, r11
    1b0:   3882        ret
```

Figure 10: Assembly language listing for eSi-3200 implementation of CRC-32

This executed in 14343 cycles, corresponding to 14 cycles per byte. This is slightly more efficient than the 16-bit non-reflected CRC.

We now implement the main CRC calculation as a user-defined instruction in the same manner as the 16-bit CRC. For this example the 32-bit CRC is performed in user instruction #1.

```c
uint32_t crc32(uint8_t *message, unsigned int N)
{
    uint32_t crc;

    crc = 0xffffffff;
    while(N--)
    {
        crc = user1(crc, *message++);
    }
    return ~crc;
}
```

Figure 11: Subroutine for generating CRC-32 with user-defined instruction user1

The corresponding assembly language code generated by the compiler is given below

```
00000166 <crc32>:
    166:   157f        l      r10, -1
    168:   1580        l      r11, 0
    16a:   2489        bz     r9, 17c <crc32+0x16>
    16c:   c4ff        add    r9, -1
    16e:   8d88        lbu    r11, (r8+[0])
    170:   c401        add    r8, 1
    172:   e0ea 3d0b   user1  r10, r10, r11
    176:   2cfb        bnz    r9, 16c <crc32+0x6>
    178:   e041 3d8a   not    r11, r10
    17c:   3c6b        mv     r8, r11
    17e:   3882        ret
```

Figure 12: Assembly language listing for eSi-3200 implementation of CRC-32 with user-defined instructions

EnSilica Limited, The Barn, Waterloo Road
Wokingham RG40 3BY, United Kingdom
Tel: +44 (0)118 3217 310
info@ensilica.com   7 of 17

© 2010 EnSilica Ltd, All Rights Reserved
Version 1.1
11/01/2010

This now executes in 7175 cycles equivalent to 7 cycles per byte. The speed-up that can be achieved is 2x and the 1K byte look-up table can be implemented efficiently in ROM within the user-defined function logic.

Finally we consider how a 16-bit processor executes a 32-bit CRC. Clearly there is now an architectural miss-match because the registers in this microprocessor are only 16-bits wide and the calculations are on 32-bit variables. Nevertheless it is a basic requirement for a 16-bit machine to handle wider data widths efficiently.

```
000000cc <crc32>:
      cc:   3d69          mv    r10, r9
      ce:   1600          l     r12, 0
      d0:   1480          l     r9, 0
      d2:   2521          bz    r10, 114 <crc32+0x48>
      d4:   167f          l     r12, -1
      d6:   14ff          l     r9, -1
      d8:   e096 1788     l     r15, 19208
      dc:   8e88          lbu   r13, (r8+[0])
      de:   e00b 3648     sru   r11, r12, 8
      e2:   e20c 3e4d     xor   r12, r12, r13
      e6:   e201 c67f     and   r12, 0xff
      ea:   3622          sl    r12, r12, 2
      ec:   e00c 3f9c     add   r12, r15, r12
      f0:   969c          lh    r13, (r12+[1])
      f2:   e00e 34a8     sl    r14, r9, 8
      f6:   960c          lh    r12, (r12+[0])
      f8:   e00b 3f4b     or    r11, r14, r11
      fc:   34c8          sru   r9, r9, 8
      fe:   c57f          add   r10, -1
     100:   e20c 3e4b     xor   r12, r12, r11
     104:   e209 3ec9     xor   r9, r13, r9
     108:   c401          add   r8, 1
     10a:   2d69          bnz   r10, dc <crc32+0x10>
     10c:   e041 3e0c     not   r12, r12
     110:   e041 3c89     not   r9, r9
     114:   3c6c          mv    r8, r12
     116:   3882          ret
```

Figure 13: Assembly language listing of eSi-1600 implementation of CRC-32

This code runs in 18443 cycles, corresponding to 18 cycles per byte, or 0.78x the speed achieved by the 32-bit processor.

If we now want to implement the CRC in user-defined logic we come across a problem. The user-defined instructions operate on registers having the native architecture bit-width, in this case 16-bits, but the CRC is 32-bits wide. Clearly we can no longer pass the crc variable to the user-defined instruction. This is an example where the previous algorithm can't be implemented in user logic without restructuring the meaning of the instruction, but fortunately eSi-RISC supports this. In this case we simply use two user-defined control and status registers (CSR) to hold the 32-bit CRC and the user instruction just passes one message byte. The user66 instruction that only takes a single argument and returns void is applicable to this example. This technique is described in the code listing below but further details are beyond the discussion in this application note.

```
uint32_t crc32(uint8_t *message, unsigned int N)
{
      uint32_t crc;

      user_wcsr(0, 0, 0xffff);
      user_wcsr(0, 1, 0xffff);
      while(N--)
      {
            user66(*message++);
      }
      crc = (user_rcsr(0, 1) << 16) | user_rcsr(0, 0);
      return crc;
}
```

Figure 14: Using user-defined control and status registers to allow 32-bit CRC calculation on eSi-1600.

Further efficiencies discussion using CRC-16 example

The eSi-RISC processor has some optional instructions and addressing modes that can have a positive impact on reducing processor clock cycles or improving code density. One of these is a hardware loop instruction. In the earlier sections it was noted that the overhead in a user-defined instruction implemented in a loop is predominately related to the loop management. Enabling the hardware loop instruction by passing "–mloop-enabled" to the toolchain slightly changes the compiled output as follows

```
000000a2 <crc16>:
      a2:   157f        l      r10, -1
      a4:   2488        bz     r9, b4 <crc16+0x12>
      a6:   c4ff        add    r9, -1
      a8:   8d88        lbu    r11, (r8+[0])
      aa:   c401        add    r8, 1
      ac:   e0ea 3d0b   user0  r10, r10, r11
      b0:   efff 04fc   loop   r9, a8 <crc16+0x6>
      b4:   3c6a        mv     r8, r10
      b6:   3882        ret
```

Figure 15: Assembly language listing of eSi-1600 implementation of CRC-16 with user-defined instruction and hardware loop

Now the loop executes in 5 cycles per byte instead of the previous 7. It is also possible to fold the address increment into the load instruction by passing "–mupdate-addr-enabled" to the toolchain. This causes the processor to be configured with a second write port to the register file so that two results can be written each clock cycle (e.g for store instructions that load a value and store the updated address). Combining this with a small amount of loop unrolling can give further savings. The compiler can be instructed to do this automatically for statically defined loop iterations by passing –funroll-loops to the compiler, or it can be manually applied. Consider re-writing the code to unroll 4 iterations of the loop and taking care of remaining bytes as follows.

EnSilica Limited, The Barn, Waterloo Road
Wokingham RG40 3BY, United Kingdom
Tel: +44 (0)118 3217 310
info@ensilica.com

© 2010 EnSilica Ltd, All Rights Reserved
Version 1.1
11/01/2010

9 of 17

```
uint16_t crc16(uint8_t *message, unsigned int N)
{
     uint16_t crc;

     crc = 0xffff;
     while(N >= 4)
     {
          crc = user0(crc, *message++);
          crc = user0(crc, *message++);
          crc = user0(crc, *message++);
          crc = user0(crc, *message++);
          N -= 4;
     }
     while(N--)
     {
          crc = user0(crc, *message++);
     }
     return crc;
}
```

Figure 16: When the loop management overhead is significant, some loop unrolling can provide benefits

Now the loop overhead is reduced and the relevant part of the assembly related to the while loop is listed below. This re-write benefits from 15 cycles per 4 bytes, equivalent to 3.75 cycles per byte.

```
     ...
     122:   8e9a        lbu    r13, (r10+[1])
     124:   8e0a        lbu    r12, (r10+[0])
     126:   e0ec 3c0c   user0  r12, r8, r12
     12a:   e0ec 3e0d   user0  r12, r12, r13
     12e:   8eaa        lbu    r13, (r10+[2])
     130:   e0ec 3e0d   user0  r12, r12, r13
     134:   8eba        lbu    r13, (r10+[3])
     136:   c504        add    r10, 4
     138:   e0e8 3e0d   user0  r8, r12, r13
     13c:   efff 05f3   loop   r11, 122 <crc16+0x12>
     ...
```

Figure 17: Assembly listing for eSi-1600 with loop unrolling by 4

The ultimate speed-up for eSi-1600 can be achieved by taking advantage of further architectural features of eSi-RISC, notably user-defined control and status registers (CSRs) available in 16 banks of 32 registers each. These are designed to enhance the capability of user-defined instructions further. For instance the CRC can be stored in one of these registers, and a user-defined instruction can now be used to pass two 16-bit operands, since it doesn't have to pass the crc variable each time, and we are assuming that the hardware is redefined to operate on 32-bits at a time. Some house keeping is required to handle the final byte boundaries, but this further optimization together with a loop unroll of 2 can process 8 bytes in 8 cycles, equivalent to 1 cycle per byte.

This section should have provided you with an insight into the power granted by judicious use of user-defined instructions, and the trade-offs available should be weighed against the speed-up required for your application.

Implementing user-defined instructions in the Simulator and Hardware
In order to employ user-defined instructions when executing an application on the instruction set simulator, a shared library must be loaded in to the simulator which implements the instructions.

EnSilica Limited, The Barn, Waterloo Road
Wokingham RG40 3BY, United Kingdom
Tel: +44 (0)118 3217 310
info@ensilica.com   10 of 17

© 2010 EnSilica Ltd, All Rights Reserved
Version 1.1
11/01/2010

This library can implement any functionality for the instruction it chooses, perhaps even by connecting to real hardware.

It is first necessary to build the shared library that implements the instructions. This library must be built using the host's toolchain (i.e. Cygwin GCC), which is not part of the eSi-RISC development tools installation. It can be installed using the Cygwin installer. The following template shows how a userX() call is mapped to the resolving function "user_insn()" in the shared library.

```c
#include <stdint.h>
#include "crc16Table.h"
#include "crc32Table.h"

/* Execute a user-defined instruction. Which operands are valid will
depend upon the opcode. */
unsigned user_insn(void *config, unsigned opcode, unsigned operand_a,
unsigned operand_b, unsigned *unsupported, unsigned *cycles)
{
    switch (opcode)
    {
    case 0:
        *cycles = 1;
        *unsupported = 0;
        return crc16Table[((operand_a >> 8) ^ operand_b) & 0xFF] ^
(operand_a << 8);
    case 1:
        *cycles = 1;
        *unsupported = 0;
        return crc32Table[(operand_a ^ operand_b) & 0xFF] ^
(operand_a >> 8);
    default:
        *unsupported = 1;
        return 0;
    }
}
```

Figure 18: Shared library implementation of the CRC-16 and CRC-32 user-defined instructions

The hardware implementation of a user-defined instruction is written in a standard HDL language like Verilog or VHDL and connected to the eSi-RISC processor "User Interface". Appendix B provides a listing of the file crc.v implementing the hardware CRC instructions through a parameterised interface. This implementation uses 8 repeated shifts and conditional xor rather than a table based approach and leads to highly efficient and compact logic. Also in that appendix is the esi_user.v listing that interfaces the hardware implementation of user0 and user1 to the eSi-RISC processor.

Conclusion

The CRC is a ubiquitous algorithm finding application in a number of established standards, mostly around MAC layer error protection. The eSi-RISC architecture is well suited to performing this function efficiently using its standard instruction set. To achieve higher performance the eSi-RISC user-defined instructions were explored as a simple expansive mechanism for offloading computation and large 1K byte look-up tables. The application code is documented in this paper for reference, together with the actual generated assembly. The results are summarised below
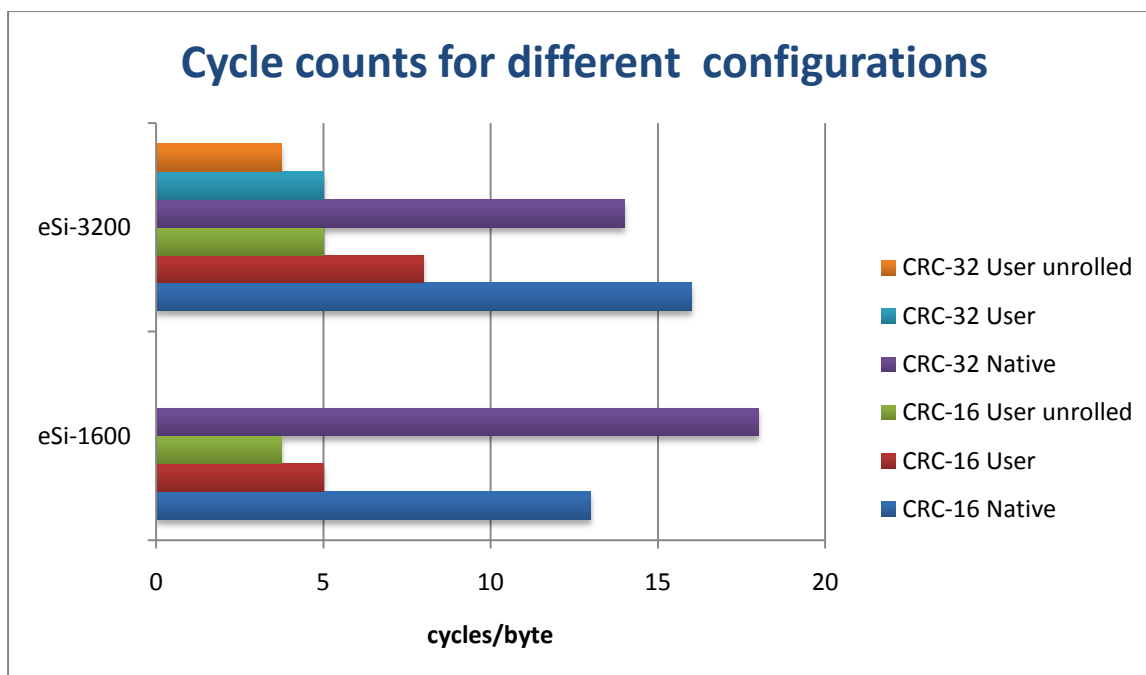
Figure 19: Cycle counts per byte

A potential speed up between 2x and 4x are available from simple application of user-defined instructions acting on a byte at a time, some architectural features and loop-unrolling. In addition we showed how the CRC-16 can be accelerated to 1 and 0.5 cycles/byte on eSi-1600 and eSi-3200 respectively, equivalent to a speed-up of 32x to 40x, by operating on multiple bytes in each user-defined instruction. The clear message is that executing a subroutine in fewer cycles leads to a lower power consumption, or the ability to trade those cycles for performance in another part of the system. Moving data tables from RAM into ROM or logic also saves power and frees up valuable memory resource for the processor.

About EnSilica

EnSilica is an established company with many years experience providing high quality IC design services to customers undertaking FPGA and ASIC designs. We have an impressive record of success working across many market segments with particular expertise in multimedia and communication applications. Our customers range from start-ups to blue-chip companies. EnSilica can provide the full range of front-end IC design services, from System Level Design, RTL coding and Verification through to either a FPGA device or the physical design interface (synthesis, STA and DFT) for ASIC designs. EnSilica also offer a portfolio of IP, including a highly configurable 16/32 bit embedded processor called eSi-RISC and the eSi-Comms range of communications IP.

References

1. http://www.ross.net/crc/download/crc_v3.txt
2. ITU-T I.432.1 B-ISDN user-network interface – Physical layer specification: General characteristics 02/99
3. RFC3095 RObust Header Compression (ROHC)
4. IEEE 802.3-2005 Part 3: Carrier sense multiple access with collision detection (CSMA/CD) access method and physical layer specifications.
5. EN 300 175-3 v2.2.1 (2008-11) Digital Enhanced Cordless Telecommunications (DECT); Common Interface (CI); Part 3: Medium Access Control (MAC) layer
6. EN 300 392-2 v3.2.1 (2007-09) Terrestrial Trunked Radio (TETRA); Voice plus Data (V+D); Part 2: Air Interface (AI)

EnSilica Limited, The Barn, Waterloo Road
Wokingham RG40 3BY, United Kingdom
Tel: +44 (0)118 3217 310
info@ensilica.com

© 2010 EnSilica Ltd, All Rights Reserved
Version 1.1
11/01/2010

12 of 17

7. IEEE Std 802.15.1-2005 Wireless Medium Access control (MAC) and Physical Layer (PHY) Specifications for Wireless Personal Area Networks (WPANs)

8. IEEE Std 802.15.4-2006 Wireless Medium access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (WPANs)

9. ES 201 980 v2.2.1 (2005-08) Digital Radio Mondiale (DRM); System Specification

10. Universal Serial Bus Specification Revision 2.0 April 27, 2000.

11. ISO/IEC 9899:201x August 11, 2008 Programming languages - C

EnSilica Limited, The Barn, Waterloo Road
Wokingham RG40 3BY, United Kingdom
Tel: +44 (0)118 3217 310
info@ensilica.com   13 of 17

© 2010 EnSilica Ltd, All Rights Reserved
Version 1.1
11/01/2010

Appendix A

```c
const uint16_t crc16Table[256] = {
    0x0000, 0x1021, 0x2042, 0x3063, 0x4084, 0x50a5, 0x60c6, 0x70e7,
    0x8108, 0x9129, 0xa14a, 0xb16b, 0xc18c, 0xd1ad, 0xe1ce, 0xf1ef,
    0x1231, 0x0210, 0x3273, 0x2252, 0x52b5, 0x4294, 0x72f7, 0x62d6,
    0x9339, 0x8318, 0xb37b, 0xa35a, 0xd3bd, 0xc39c, 0xf3ff, 0xe3de,
    0x2462, 0x3443, 0x0420, 0x1401, 0x64e6, 0x74c7, 0x44a4, 0x5485,
    0xa56a, 0xb54b, 0x8528, 0x9509, 0xe5ee, 0xf5cf, 0xc5ac, 0xd58d,
    0x3653, 0x2672, 0x1611, 0x0630, 0x76d7, 0x66f6, 0x5695, 0x46b4,
    0xb75b, 0xa77a, 0x9719, 0x8738, 0xf7df, 0xe7fe, 0xd79d, 0xc7bc,
    0x48c4, 0x58e5, 0x6886, 0x78a7, 0x0840, 0x1861, 0x2802, 0x3823,
    0xc9cc, 0xd9ed, 0xe98e, 0xf9af, 0x8948, 0x9969, 0xa90a, 0xb92b,
    0x5af5, 0x4ad4, 0x7ab7, 0x6a96, 0x1a71, 0x0a50, 0x3a33, 0x2a12,
    0xdbfd, 0xcbdc, 0xfbbf, 0xeb9e, 0x9b79, 0x8b58, 0xbb3b, 0xab1a,
    0x6ca6, 0x7c87, 0x4ce4, 0x5cc5, 0x2c22, 0x3c03, 0x0c60, 0x1c41,
    0xedae, 0xfd8f, 0xcdec, 0xddcd, 0xad2a, 0xbd0b, 0x8d68, 0x9d49,
    0x7e97, 0x6eb6, 0x5ed5, 0x4ef4, 0x3e13, 0x2e32, 0x1e51, 0x0e70,
    0xff9f, 0xefbe, 0xdfdd, 0xcffc, 0xbf1b, 0xaf3a, 0x9f59, 0x8f78,
    0x9188, 0x81a9, 0xb1ca, 0xa1eb, 0xd10c, 0xc12d, 0xf14e, 0xe16f,
    0x1080, 0x00a1, 0x30c2, 0x20e3, 0x5004, 0x4025, 0x7046, 0x6067,
    0x83b9, 0x9398, 0xa3fb, 0xb3da, 0xc33d, 0xd31c, 0xe37f, 0xf35e,
    0x02b1, 0x1290, 0x22f3, 0x32d2, 0x4235, 0x5214, 0x6277, 0x7256,
    0xb5ea, 0xa5cb, 0x95a8, 0x8589, 0xf56e, 0xe54f, 0xd52c, 0xc50d,
    0x34e2, 0x24c3, 0x14a0, 0x0481, 0x7466, 0x6447, 0x5424, 0x4405,
    0xa7db, 0xb7fa, 0x8799, 0x97b8, 0xe75f, 0xf77e, 0xc71d, 0xd73c,
    0x26d3, 0x36f2, 0x0691, 0x16b0, 0x6657, 0x7676, 0x4615, 0x5634,
    0xd94c, 0xc96d, 0xf90e, 0xe92f, 0x99c8, 0x89e9, 0xb98a, 0xa9ab,
    0x5844, 0x4865, 0x7806, 0x6827, 0x18c0, 0x08e1, 0x3882, 0x28a3,
    0xcb7d, 0xdb5c, 0xeb3f, 0xfb1e, 0x8bf9, 0x9bd8, 0xabbb, 0xbb9a,
    0x4a75, 0x5a54, 0x6a37, 0x7a16, 0x0af1, 0x1ad0, 0x2ab3, 0x3a92,
    0xfd2e, 0xed0f, 0xdd6c, 0xcd4d, 0xbdaa, 0xad8b, 0x9de8, 0x8dc9,
    0x7c26, 0x6c07, 0x5c64, 0x4c45, 0x3ca2, 0x2c83, 0x1ce0, 0x0cc1,
    0xef1f, 0xff3e, 0xcf5d, 0xdf7c, 0xaf9b, 0xbfba, 0x8fd9, 0x9ff8,
    0x6e17, 0x7e36, 0x4e55, 0x5e74, 0x2e93, 0x3eb2, 0x0ed1, 0x1ef0};
```

Figure 20: Listing of generated unreflected CRC-16 table

```c
const uint32_t crc32Table[256] = {
    0x00000000, 0x77073096, 0xee0e612c, 0x990951ba, 0x076dc419, 0x706af48f, 0xe963a535, 0x9e6495a3,
    0x0edb8832, 0x79dcb8a4, 0xe0d5e91e, 0x97d2d988, 0x09b64c2b, 0x7eb17cbd, 0xe7b82d07, 0x90bf1d91,
    0x1db71064, 0x6ab020f2, 0xf3b97148, 0x84be41de, 0x1adad47d, 0x6ddde4eb, 0xf4d4b551, 0x83d385c7,
    0x136c9856, 0x646ba8c0, 0xfd62f97a, 0x8a65c9ec, 0x14015c4f, 0x63066cd9, 0xfa0f3d63, 0x8d080df5,
    0x3b6e20c8, 0x4c69105e, 0xd56041e4, 0xa2677172, 0x3c03e4d1, 0x4b04d447, 0xd20d85fd, 0xa50ab56b,
    0x35b5a8fa, 0x42b2986c, 0xdbbbc9d6, 0xacbcf940, 0x32d86ce3, 0x45df5c75, 0xdcd60dcf, 0xabd13d59,
    0x26d930ac, 0x51de003a, 0xc8d75180, 0xbfd06116, 0x21b4f4b5, 0x56b3c423, 0xcfba9599, 0xb8bda50f,
    0x2802b89e, 0x5f058808, 0xc60cd9b2, 0xb10be924, 0x2f6f7c87, 0x58684c11, 0xc1611dab, 0xb6662d3d,
    0x76dc4190, 0x01db7106, 0x98d220bc, 0xefd5102a, 0x71b18589, 0x06b6b51f, 0x9fbfe4a5, 0xe8b8d433,
    0x7807c9a2, 0x0f00f934, 0x9609a88e, 0xe10e9818, 0x7f6a0dbb, 0x086d3d2d, 0x91646c97, 0xe6635c01,
    0x6b6b51f4, 0x1c6c6162, 0x856530d8, 0xf262004e, 0x6c0695ed, 0x1b01a57b, 0x8208f4c1, 0xf50fc457,
    0x65b0d9c6, 0x12b7e950, 0x8bbeb8ea, 0xfcb9887c, 0x62dd1ddf, 0x15da2d49, 0x8cd37cf3, 0xfbd44c65,
    0x4db26158, 0x3ab551ce, 0xa3bc0074, 0xd4bb30e2, 0x4adfa541, 0x3dd895d7, 0xa4d1c46d, 0xd3d6f4fb,
    0x4369e96a, 0x346ed9fc, 0xad678846, 0xda60b8d0, 0x44042d73, 0x33031de5, 0xaa0a4c5f, 0xdd0d7cc9,
    0x5005713c, 0x270241aa, 0xbe0b1010, 0xc90c2086, 0x5768b525, 0x206f85b3, 0xb966d409, 0xce61e49f,
    0x5edef90e, 0x29d9c998, 0xb0d09822, 0xc7d7a8b4, 0x59b33d17, 0x2eb40d81, 0xb7bd5c3b, 0xc0ba6cad,
    0xedb88320, 0x9abfb3b6, 0x03b6e20c, 0x74b1d29a, 0xead54739, 0x9dd277af, 0x04db2615, 0x73dc1683,
    0xe3630b12, 0x94643b84, 0x0d6d6a3e, 0x7a6a5aa8, 0xe40ecf0b, 0x9309ff9d, 0x0a00ae27, 0x7d079eb1,
    0xf00f9344, 0x8708a3d2, 0x1e01f268, 0x6906c2fe, 0xf762575d, 0x806567cb, 0x196c3671, 0x6e6b06e7,
    0xfed41b76, 0x89d32be0, 0x10da7a5a, 0x67dd4acc, 0xf9b9df6f, 0x8ebeeff9, 0x17b7be43, 0x60b08ed5,
    0xd6d6a3e8, 0xa1d1937e, 0x38d8c2c4, 0x4fdff252, 0xd1bb67f1, 0xa6bc5767, 0x3fb506dd, 0x48b2364b,
    0xd80d2bda, 0xaf0a1b4c, 0x36034af6, 0x41047a60, 0xdf60efc3, 0xa867df55, 0x316e8eef, 0x4669be79,
    0xcb61b38c, 0xbc66831a, 0x256fd2a0, 0x5268e236, 0xcc0c7795, 0xbb0b4703, 0x220216b9, 0x5505262f,
    0xc5ba3bbe, 0xb2bd0b28, 0x2bb45a92, 0x5cb36a04, 0xc2d7ffa7, 0xb5d0cf31, 0x2cd99e8b, 0x5bdeae1d,
    0x9b64c2b0, 0xec63f226, 0x756aa39c, 0x026d930a, 0x9c0906a9, 0xeb0e363f, 0x72076785, 0x05005713,
    0x95bf4a82, 0xe2b87a14, 0x7bb12bae, 0x0cb61b38, 0x92d28e9b, 0xe5d5be0d, 0x7cdcefb7, 0x0bdbdf21,
    0x86d3d2d4, 0xf1d4e242, 0x68ddb3f8, 0x1fda836e, 0x81be16cd, 0xf6b9265b, 0x6fb077e1, 0x18b74777,
    0x88085ae6, 0xff0f6a70, 0x66063bca, 0x11010b5c, 0x8f659eff, 0xf862ae69, 0x616bffd3, 0x166ccf45,
    0xa00ae278, 0xd70dd2ee, 0x4e048354, 0x3903b3c2, 0xa7672661, 0xd06016f7, 0x4969474d, 0x3e6e77db,
    0xaed16a4a, 0xd9d65adc, 0x40df0b66, 0x37d83bf0, 0xa9bcae53, 0xdebb9ec5, 0x47b2cf7f, 0x30b5ffe9,
    0xbdbdf21c, 0xcabac28a, 0x53b39330, 0x24b4a3a6, 0xbad03605, 0xcdd70693, 0x54de5729, 0x23d967bf,
    0xb3667a2e, 0xc4614ab8, 0x5d681b02, 0x2a6f2b94, 0xb40bbe37, 0xc30c8ea1, 0x5a05df1b, 0x2d02ef8d};
```

Figure 21: Listing of generated reflected CRC-32 table

Appendix B

```verilog
module crc #(parameter CRC_WIDTH=5'd16, POLY=16'h0121, REFLECT=1'b0)
(
    input [CRC_WIDTH-1:0] crc_in,
    input [7:0] message,
    output [CRC_WIDTH-1:0] crc_out,
    output waitrequest
);
    //------------------------------------------------------------------------
    // iteration function for a single bit
    //------------------------------------------------------------------------

    function [CRC_WIDTH-1:0] rem(input [CRC_WIDTH-1:0] a, input b);
        reg carry;
        begin
            rem = REFLECT ? a >> 1 : a << 1;
            carry = REFLECT ? a[0] : a[CRC_WIDTH-1];
            if(carry ^ b)
                rem = rem ^ POLY;
        end
    endfunction

    //------------------------------------------------------------------------
    // reflect a byte
    //------------------------------------------------------------------------

    function [7:0] refl(input [7:0] a);
        integer i;
        begin
            for(i=0;i<8;i=i+1)
                refl[i] = a[7-i];
        end
    endfunction

    //------------------------------------------------------------------------
    // connectivity
    //------------------------------------------------------------------------

    wire [CRC_WIDTH-1:0] r[0:7];
    wire [7:0] m;

    //------------------------------------------------------------------------
    // iterate over a byte
    //------------------------------------------------------------------------

    assign m = REFLECT ? refl(message) : message;
    assign r[0] = rem(crc_in, m[7]);
    assign r[1] = rem(r[0], m[6]);
    assign r[2] = rem(r[1], m[5]);
    assign r[3] = rem(r[2], m[4]);
    assign r[4] = rem(r[3], m[3]);
    assign r[5] = rem(r[4], m[2]);
    assign r[6] = rem(r[5], m[1]);
    assign r[7] = rem(r[6], m[0]);

    assign crc_out = r[7];
    assign waitrequest = 1'b0; // not used - crc module is single cycle

endmodule
```

Figure 22: Listing of crc.v Verilog module

```verilog
`include "esi_user_include.v"

module esi_user
(
    input clk,
    input reset_n,

    // user instruction interface
    input [`CPU_USER_OPCODE_RNG] user_opcode,
    input [`CPU_WORD_RNG] user_operand_a,
    input [`CPU_WORD_RNG] user_operand_b,
    output reg [`CPU_WORD_RNG] user_result,
    output user_condition_met,
    output reg user_stall,
    output user_opcode_unsupported,

    // user csr interface
    input [`CPU_CSR_RNG] user_csr_bank,
    input [`CPU_CSR_RNG] user_csr_csr,
    input user_csr_write,
    input user_csr_read,
    input [`CPU_WORD_RNG] user_csr_write_data,
    output [`CPU_WORD_RNG] user_csr_read_data,
    output [`CPU_WORD_RNG] user_csr_unsupported
);

    //-------------------------------------------------------------------------
    // connectivity
    //-------------------------------------------------------------------------

    wire [15:0] crc16_out;
    wire [31:0] crc32_out;
    wire [1:0] waitrequest;

    //-------------------------------------------------------------------------
    // output mux
    //-------------------------------------------------------------------------

    always@(*)
    begin: RESULT_MUX
        case(user_opcode) // synthesis parallel_case
            0: user_result = crc16_out;
            1: user_result = crc32_out;
            default: user_result = `CPU_WORD_WIDTH'd0;
        endcase
    end

    always@(*)
    begin: STALL_MUX
        case(user_opcode) // synthesis parallel_case
            0: user_stall = waitrequest[0];
            1: user_stall = waitrequest[1];
            default: user_stall = `FALSE;
        endcase
    end

    assign user_opcode_unsupported = user_opcode > `CPU_USER_OPCODE_WIDTH'd1;
    assign user_condition_met = `FALSE;
    assign user_csr_read_data = `CPU_WORD_WIDTH'd0;
    assign user_csr_unsupported = `TRUE;
```

```verilog
   //------------------------------------------------------------------------
   // user0
   //------------------------------------------------------------------------

   crc #(
       .CRC_WIDTH(5'd16),
       .POLY(16'h1021),
       .REFLECT(1'b0)
   ) u_user0 (
       .crc_in(user_operand_a[15:0]),
       .message(user_operand_b[7:0]),
       .crc_out(crc16_out),
       .waitrequest(waitrequest[0])
   );

   //------------------------------------------------------------------------
   // user1 only valid for CPU_WORD_WIDTH > 16
   //------------------------------------------------------------------------

   crc #(
       .CRC_WIDTH(6'd32),
       .POLY(32'hEDB88320),    // reflected polynomial
       .REFLECT(1'b1)
   ) u_user1 (
       .crc_in(user_operand_a[31:0]),
       .message(user_operand_b[7:0]),
       .crc_out(crc32_out),
       .waitrequest(waitrequest[1])
   );

 endmodule
```

Figure 23: Listing of esi_user.v Verilog interface to the CPU.