

Application Note

Implementing square roots on eSi-RISC

Introduction

This application note provides some practical examples of calculating integer and fractional square roots on 32-bit versions of eSi-RISC, and looks at how the user-defined multi-cycle instruction extensions can provide a saving in power and computation cycles.

The square root 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. Whilst it would be possible to include a floating point library emulation this takes up a lot of code space and is also very cycle intensive. In this paper we examine computing an integer or fractional square root using the simple binary restoring algorithm in reference [1]. This method calculates a N-bit integer square root in N/2 iterations. The objective is to minimise the number of cycles required per iteration to get an efficient implementation. Another method involving Newton Raphson iterations is also commonly found, but has the disadvantage of a division at each iteration, although the number of iterations is less. In some applications Newton Raphson is used to find $1/\sqrt{x}$ and then a final multiply by x produces the desired result. This method avoids the division per iteration, but still requires three multiplications and fractional accuracy.

The square root finds application in many areas of statistics and digital signal processing. When converting from power to amplitude and in normalising data sets it can represent a significant burden on cycle count and hence influences both clock speed and power consumption. Implementing the square root in hardware as a user-defined instruction can provide significant speed-up and power saving for a small increase in gate count.

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 eSi-3200 is configured with 32-bit wide instruction memory, so that 32-bit instructions can be fetched in a single cycle. The eSi-RISC compiler is very efficient at producing optimised code, often as good as hand crafted assembly language. In addition the instruction set is optimised so that many instructions are encoded in 16-bits leading to a high code density.

Binary restoring algorithm

The algorithm described in reference [1] performs a square root by finding the root a bit at a time. The implementation is quite similar to integer division, having a trial root, comparison and restoring step. Full details are in references [1] and [2] and shall not be repeated here.

The *isqrt* subroutine is shown in Figure 1 below. When *isqrt()* is called with N=16 it performs a 32-bit integer square root with a 16-bit integer result. For $16 < N \leq 30$ there are an additional N-16 fractional bits in the result. For instance calling *isqrt(x,30)* computes a result with 16 integer and 14 fractional bits. As an example, to calculate $\sqrt{2.0}$ we call *isqrt(2,30)* and the result is 23170. To convert this to a fraction divide by 2^{14} ; $(23170/2^{14}) = 1.4141845703125$.

Subsequent analysis uses N=16 for an integer square root result.

```
unsigned int isqrt(unsigned int x, unsigned int N)
{
    unsigned int q, trial, rem;

    // initialise
    q = 0;
    rem = 0;

    do {
        // shift top 2 bits of x into LSBs of remainder
        rem = (rem << 2) | (x >> 30);
        x <<= 2;

        trial = ((q << 2) | 1); // trial root
        q <<= 1;
        if(rem >= trial)
        {
            rem = rem - trial; // restore remainder
            q |= 1;
        }
    } while(--N);

    return q;
}
```

Figure 1: Code for binary restoring square root

The first 17 roots are shown below

```
Input: 0, output 0
Input: 1, output 1
Input: 2, output 1
Input: 3, output 1
Input: 4, output 2
Input: 5, output 2
Input: 6, output 2
Input: 7, output 2
Input: 8, output 2
Input: 9, output 3
Input: 10, output 3
Input: 11, output 3
Input: 12, output 3
Input: 13, output 3
Input: 14, output 3
Input: 15, output 3
Input: 16, output 4
```

Figure 2: First 17 square roots

Clearly the routine generates the equivalent of $\text{floor}(\text{sqrt}(\text{float}(x)))$. The generated assembly code produced by the eSi-RISC compiler is shown below

```

00000196 <isqrt>:
  196:  1600      l    r12, 0
  198:  1580      l    r11, 0
  19a:  e40d 344e  sru  r13, r8, 30
  19e:  e00a 35a2  sl   r10, r11, 2
  1a2:  3622      sl   r12, r12, 2
  1a4:  e00c 3ecc  or   r12, r13, r12
  1a8:  cd01      or   r10, 0x1
  1aa:  3e7a      cmp  r12, r10
  1ac:  35a1      sl   r11, r11, 1
  1ae:  3422      sl   r8, r8, 2
  1b0:  c4ff      add  r9, -1
  1b2:  0483      blu  1b8 <isqrt+0x22>
  1b4:  3e2a      sub  r12, r12, r10
  1b6:  cd81      or   r11, 0x1
  1b8:  2cf1      bnz  r9, 19a <isqrt+0x4>
  1ba:  3c6b      mv   r8, r11
  1bc:  3882      ret

```

Figure 3: eSi-3200 assembly listing for isqrt

The main loop is highlighted in red and consists of 12 to 13 cycles per iteration or between 192 and 224 cycles per root, plus a few cycles overhead on entry and exit. Using the eSi-RISC profile capability we can generate the square root of all the squares of 0...65535 and determine the average execution time in cycles. This turns out to be 220 cycles, in close agreement with the expected value. Further optimization is possible using optional instruction set features such as the hardware loop instruction, which reduces the average execution time to 190 cycles.

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 square root these correspond to very efficient hardware taking only a few logic gates. 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. In addition user-defined instructions can take multiple clock cycles to compute their result. For example the next listing shows how one might implement the square root calling a user-defined instruction that takes 18 cycles to return a result; 1 to register the inputs, 16 for iterations and 1 to capture the output.

```

unsigned int isqrt(unsigned int x, unsigned int N)
{
    return user2(x, N);
}

```

Figure 4: Subroutine for generating square root with user-defined instruction user2

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

```

00000196 <isqrt>:
  196:  e0e8 3c29  user2 r8, r8, r9
  19a:  3882      ret

```

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

This code executes in 18 cycles as expected, and represents a 12x speed-up over the software implementation. 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. 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.

```

/* 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 2:
            *cycles = operand_b + 2;
            *unsupported = 0;
            return isqrt(operand_a, operand_b);
        default:
            *unsupported = 1;
            return 0;
    }
}

```

Figure 6: Shared library implementation of the square root 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 A provides a listing of the file sqrt.v implementing the hardware instructions. Also in that appendix is the esi_user.v listing that interfaces the hardware implementation of user2 to the eSi-RISC processor.

Conclusion

The square root is a ubiquitous algorithm finding application in statistics and digital signal processing. 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. The application code is documented in this paper for reference, together with the actual generated assembly. A 12x cycle count speed up is achieved for a small addition of hardware.

Configuration	Cycles	Speed-up
Basic	220	1.0x
With hardware loop	190	1.16x
With user-defined instruction	18	12.2x

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.

Many simple algorithms can be accelerated with the addition of small hardware modules. By adding these in as custom instructions the module becomes tightly coupled with the processor and has a bit exact C code simulation model. Examples of suitable algorithms include

- CRC
- IP checksum calculation
- Max/Min search
- Sorting
- AES decoder
- Cordic
- Data scrambler
- Convolutional encoder
- Viterbi and Turbo decoder
- Sin/cosine generation
- Galois field multiplier
- G.711 u/A-law conversion
- MP3 sample dequantization
- Y'CrCb to R'G'B' conversion

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. Computer Arithmetic: Algorithms and Hardware Designs, Oxford University Press, New York, 2101, B Parhami.
2. http://en.wikipedia.org/wiki/Methods_of_computing_square_roots

Appendix A

```

module sqrt #(parameter WIDTH = 32, CY_WIDTH = 6)
(
    input clk,                // Clock
    input reset_n,           // Reset, active-low
    input enable,            // Set high to start a new calculation
    input [WIDTH-1:0] radicand, // Value to find square root of
    input [CY_WIDTH-1:0] N,   // Iteration count
    output reg [WIDTH-1:0] root, // Square root of radicand
    output waitrequest       // Wait for iterations to finish
);
    reg [CY_WIDTH-1:0] cycles; // How many iterations remain
    reg [2*WIDTH-1:0] remainder;
    wire [WIDTH+1:0] remainder_hi, restore, trial;
    wire [WIDTH-3:0] remainder_lo;
    wire start, loop, do_restore;

    assign remainder_hi = remainder[2*WIDTH-1:WIDTH-2];
    assign remainder_lo = remainder[WIDTH-3:0];
    assign trial = {root, 2'b01};
    assign restore = remainder_hi - trial;
    assign start = enable & !loop;
    assign loop = cycles > {CY_WIDTH{1'b0}};
    assign waitrequest = enable | loop;
    assign do_restore = remainder_hi >= trial;

    always@(posedge clk or negedge reset_n)
    begin: CYCLE_COUNT
        if(~reset_n)
            cycles <= {CY_WIDTH{1'b0}};
        else if(start)
            cycles <= N;
        else if(loop)
            cycles <= cycles - 1'b1;
    end

    always@(posedge clk or negedge reset_n)
    begin: REMAINDER_ITERATION
        if(~reset_n)
            remainder <= {2*WIDTH{1'b0}};
        else if(start)
            remainder <= {{WIDTH{1'b0}}, radicand};
        else if(loop)
            begin
                if(do_restore)
                    remainder <= {restore[WIDTH-1:0], remainder_lo, 2'b00};
                else
                    remainder <= {remainder_hi[WIDTH-1:0], remainder_lo, 2'b00};
            end
    end

    always@(posedge clk or negedge reset_n)
    begin: ROOT_ITERATION
        if(~reset_n)
            root <= {WIDTH{1'b0}};
        else if(start)
            root <= {WIDTH{1'b0}};
        else if(loop)
            begin
                if(do_restore)
                    root <= {root[WIDTH-2:0], 1'b1};
                else
                    root <= {root[WIDTH-2:0], 1'b0};
            end
    end
endmodule

```

Figure 7: Listing of sqrt.v Verilog module

```

`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
);
    localparam CY_WIDTH = 6;
    localparam WIDTH = 32;

    wire enable;

    assign enable = user_opcode == `CPU_USER_OPCODE_WIDTH'd2;
    assign user_opcode_unsupported = user_opcode != `CPU_USER_OPCODE_WIDTH'd2;
    assign user_condition_met = `FALSE;
    assign user_csr_read_data = `CPU_WORD_WIDTH'd0;
    assign user_csr_unsupported = `TRUE;

    //-----
    // user2
    //-----

    sqrt #(
        .WIDTH                (WIDTH),
        .CY_WIDTH              (CY_WIDTH)
    ) user2 (
        .clk                    (clk),
        .reset_n                (reset_n),
        .enable                  (enable),
        .radicand                (user_operand_a),
        .N                       (user_operand_b[CY_WIDTH-1:0]),
        .root                    (user_result),
        .waitrequest             (user_stall)
    );
endmodule

```

Figure 8: Listing of esi_user.v Verilog interface to the CPU.