Implementation of complex operators and datastructures in FPGA

Frijns, R.M.W.

Award date:
2008

TU/e technische universiteit eindhoven

Faculty of Electrical Engineering
Section Design Technology For Electronic Systems (ICS/ES)
ICS-ES 875

Master's Thesis

IMPLEMENTATION OF COMPLEX OPERATORS AND DATASTRUCTURES IN FPGA.

R.M.W. Frijns

Coach: dr.ir. L. Józwiak
Date: June 2006

TU/e technische universiteit eindhoven

Faculty of Electrical Engineering
Section Design Technology For Electronic Systems (ICS/ES)
ICS-ES 876

Practical Training Report

IMPLEMENTATION OF COMPLEX OPERATORS AND DATASTRUCTURES IN FPGA.

R.M.W. Frijns

Supervisor: dr.ir. L. Józwiak
Date: June 2006

Abstract

Currently, the Position Sensitive Neutron Detector (PSND) of the Reactor Institute Delft is being re-implemented. Originating from 1996, its electronics became obsolete and its measurement results started to be less and less reliable. Therefore, it was necessary to develop and implement a new neutron detector, and it was decided to make a neutron detector with better functional and parametric characteristics using the modern FPGA technology to implement its main real-time signal processing part.
Position Sensitive Neutron Detection (PSND) is realized by a combination of several standard neutron detectors placed at different angles with respect to the source. Normally, only the number of incident neutrons is counted, but with PSND, the count is a function of the angle of impact. Electric charge generated at a neutron count in such a detector is distributed to two outputs. The closer the impact is to one of the outputs, the more charge is transferred to it, so the ratio of the charge on the two outputs is a measure for its position (in one dimension). This ratio is calculated using function with a division operator, which happens to be the most complex and time-consuming operator of the basic arithmetic operators.
The aim of the work reported here was to research some of the most complex operators and data structures required for the FPGA implementation of the signal processing part of the neutron detector regarding their effectiveness and efficiency, to implement some selected operators and data structures, and to validate their implementation.
This report describes the research, FPGA implementation, and validation of the selected operators and data structures for use in the new PSND, focusing mainly on implementation of the most complex division operation. Initially, a 65536x8 memory array has been realized for use as a lookup-table implementation of the division operation with 8 bit input/output operands. Although this is the fastest way to perform a division, it is the least scalable, and requires lots of memory for longer words. Therefore, an algorithmic implementation has been realized, trading the computation speed for scalability by application of a shift-and-subtract division algorithm. However, this introduced a large latency, so a literature search on fast division algorithms has been performed that resulted in a proposal to use a restoring division algorithm or the Goldschmidt algorithm. Also, our own binary search division algorithm has been proposed, exploiting the availability of hardware multiplier blocks to improve execution time with respect to the standard shift-and-subtract division method. To compare performance, a restoring division algorithm, binary search division algorithm and the Goldschmidt multiplicative division algorithm have been implemented. Comparison of the implemented designs shows that the Goldschmidt division algorithm performs best. It requires the least iterations, has the best overall latency, occupies the least area and scales logarithmically with input operand width. Although this algorithm imposes restrictions on the input format, even with some pre-shifting stage to overcome these restrictions, this algorithm outperforms the other implemented designs. The binary search division algorithm implementation performs worst, due to the complex estimate generation process. Implementing part of this function in a lookup-table should enhance its performance.
For now, the division function for the PSND is not implemented on the FPGA itself, but on a computer. Raw data is sent from the FPGA to the computer via an UART-bus, where the division itself is performed locally by a table-lookup. The main reason for the off-chip processing is the easier implementation, trading communication bandwidth for a faster implementation trajectory. When there is time left for improvements, the restoring-division algorithm will be implemented to perform the division on the FPGA itself. This algorithm is preferred over the Goldschmidt algorithm because it is directly implementable without setting inconvenient input data requirements.

Contents

1 Introduction

After 10 years of service, the Position Sensitive Neutron Detector (PSND) of the Reactor Institute Delft is being re-implemented. Its electronics have become obsolete and its measurement performance has detoriated, so a new neutron detector with better functional and parametric characteristics was needed. Instead of building a completely new detector, it was decided to re-implement the old detector, replacing its obsolete parts with new designs. One of the main updates was the detector's real-time signal processing part, which is now implemented using the modern FPGA-technology.

Most neutron detectors use detection methods based on scintillation. The impact of a neutron on a scintillation material generates photons by means of a fluorescence process. These photons strike a photocathode in a photomultiplier tube, generating free electrons which are accelerated by an electric field, which is excited by several metal plates at an increasingly positive potential (dynodes). Accelerated electrons striking these plates excite more free electrons, effectively increasing the number of electrons at every dynode. At the end of the photomultiplier tube, the stream of free electrons is accumulated at the anode, resulting in a current pulse at the arrival of a neutron.

Standard neutron detectors only count the number of incoming neutrons. By placing several of these detectors at different angles, position sensitive neutron detection can be realized. Neutrons are now counted at different angles, measuring the angle of impact and therefore the impact position on a 2 dimensional plane.

The charge generated at a neutron count event in such a detector is distributed to two outputs. The closer the impact is to one of the outputs, the more charge is transferred to it, so the ratio of the charge on the two outputs is a measure for its position (in one dimension). This ratio is calculated using a division function.

The division operator is the most complex of the four basic arithmetic operators { +, -, *, / }, requiring lots of computation time and/or hardware resources. Unlike multiplication, a division operation can not be decomposed into several parallel stages, since because of data dependencies a consecutive stage cannot start its computation before the previous stage is finished. Therefore, implementing such an operator will be a tradeoff between computation time and chip area.

The aim of the work reported here was to research some of the most complex operators and data structures required for the FPGA implementation of the signal processing part of the neutron detector regarding their effectiveness and efficiency, to implement some selected operators and data structures, and to validate their implementation.

The main goals are:

- Implementation of synthesizable datastructures for use in the PSND:
  - 65536x8 memory array for table-lookup division
  - Realization of a function using a division operator
  - Realization of different clock signals from one external clock
- Literature search on existing division algorithms
- Implementation of some of these algorithms
- Design and implementation of a division algorithm based on number search

The internship assignment is stated in more detail in chapter 2. In Chapter 3, the implementation of the datastructures is discussed, followed by an overview of different algorithms for fast division (CH4). The implementation of some of these algorithms is discussed in chapter 5, followed by a discussion on the results of these implementations in chapter 6. Conclusions are made in chapter 7.

2 Assignment

The internship assignment is generally to assist in the implementation and synthesis of some more complex algorithms and data structures to be used in the new position sensitive neutron detector of the Reactor Institute Delft. During the internship, the effort focused into application of the division operator, seeking balance between scalability, speed and area.
This chapter describes the assignments in more detail, categorizing them into the more general implementation of datastructures and the focus into implementation of division algorithms.

2.1 Implementation of synthesizable datastructures

65536x8 memory array
To determine the position of an incoming neutron, a function f(a,b) = a*8'b11111111/(a+b), with a,b,f 8,9 or 10 bits is used. Division is the most complex and time-consuming operator of the basic arithmetic operators, so it is favorable to implement such a function in a lookup-table. The input for the table is the concatenation of the inputs a and b, and its address space is the set of all possible combinations of its inputs a and b. The contents of an address is the result of the function for the corresponding (a,b) pair. Such a lookup-table implementation introduces a latency of just one clock cycle, but has a very poor scalability since the table size grows exponentially with the input width. In this particular case, when the inputs and output are limited to 8 bits, this function can be implemented in a 65536x8 table. Goal is to determine the possibilities for implementation of a 65536x8 synthesizable memory array on a Xilinx Virtex2/Spartan2 FPGA.

Function using a division operator
Although the abovementioned function in lookup-table implementation is the fastest possible realization, it is not scalable for inputs greater then 8 bits, since its address space, and therefore its size, is determined by the number of possible input combinations. Even with inputs of 9 bits wide, the function can not be implemented by table lookup, since the FPGAs that are used do not have enough memory for such an application. Therefore, speed has to be traded for area and scalability by implementing the division operation in some algorithmic manner. Goal is to implement and synthesize the function f(a,b) = a*8'b11111111/(a+b) using a simple division algorithm.

Multiple clock domains
Since parts of the neutron detector run at different speeds, multiple clock signals need to be created based on one external clock signal. The external clock can be selected to run at 25,30,33,40,45,50,55,60,62.5,66,67.5,70,77.5,80,83 or 90 MHz. This clock signal had to be converted to three systems clocks, running at 125, 250 and 20 MHz.

2.2 Implementation of division algorithms

Literature search and implementation of division algorithms
Since the standard shift-and-subtract division methods have a high latency, different approaches and algorithms for fast division have been developed. Goal is to present an overview of these different division algorithms and their properties, and to implement one or two of the more promising algorithms in FPGA.

Design and implementation of a search division algorithm

A division operation Q=N/D, can be approached as a search for a number in an ordered list. The result of a division, Q, is a number somewhere in that ordered list. An estimate for Q is generated, and by multiplication of this estimate with the divisor D, an estimate for the numerator is formed. Comparison of this estimate with the remainder of the division process yields information about the position of the number Q in the ordered list of possible outcomes.

Due to the limited availability of hardware resources, multiplication operations are often replaced by shift operations, limiting one of the operands to a power of 2. In this case, Q is the limited operand, resulting in the well-known shift-and subtract methods.

With the availability of hardware multiplier blocks in the Virtex2 FPGA, any integer/ fixed point operand is allowed in a multiplication, opening doors for possible improvements.

Goal is to apply an application of a search algorithm for division that exploits the availability of a hardware multiplier and implement this algorithm in FPGA.

3 Implementation of synthesizable datastructures

In this chapter, the implementation of the datastructures mentioned in 2.1 is discussed.
The chapter starts with several implementations of memory arrays, followed by an implementation of an algorithmic division function. Finally, the generation of multiple clock domains is discussed.
Implementations are designed with Xilinx ISE6.3, tested/simulated with Modelsim 5.8, and targeted for the Xilinx XC2v1500 Virtex2 FPGA [2,3]. Testbenches are generated with Visual Software Solution's HDL bencher 1.02.

3.1 Memory arrays

There are three general ways to implement a memory array on a Xilinx FPGA:

- Using slice Flip-flops/latches
- Using LUTs as distributed RAM
- Using dedicated on-chip block RAM

Using slice flip-flops for synthesis of memory is highly inefficient, since for each used flip-flop the corresponding 4 input LUT is unused and cannot be used for other functions, since its output is used as a memory source. This implementation is infeasible, since there are not enough slice flip-flops on the FPGA for an array of 65536x8.

3.1.1 Configuring FPGA LUTs as distributed RAM

With the Xilinx Virtex2/Spartan2 slice architecture, the 4-input LUTs can be configured as small 16x1 RAM modules. They can be used directly by instantiating their primitives, inferred indirectly by using an appropriate Verilog code style (language template), or can be generated by the CoreGen IP-core generator tool in ISE. Table 3.1 shows the distributed selectRAM primitives.

Table 3.1 : *Distributed SelecRAM primitives.*

Primitive				**type**
1 bit wide	*2 bit wide*	*4 bit wide*	*8 bit wide*	
RAM16X1S	RAM16X2S	RAM16X4S	RAM16X8S	single port
RAM32X1S	RAM32X2S	RAM32X4S	RAM32X8S	single port
RAM64X1S	RAM64X2S			single port
RAM128X1S				single port
RAM16X1D				dual port
RAM32X1D				dual port
RAM64X1D				dual port

The following Verilog template can be used to infer a single port distributed RAM with an asynchronous read:

```
parameter RAM_WIDTH = <ram_width>;
parameter RAM_ADDR_BITS = <ram_addr_bits>;
reg [RAM_WIDTH-1:0] <ram_name> [(2**RAM_ADDR_BITS)-1:0];
wire [RAM_WIDTH-1:0] <output_data>;

<reg_or_wire> [RAM_ADDR_BITS-1:0] <address>;
<reg_or_wire> [RAM_WIDTH-1:0] <input_data>;

always @(posedge <clock>)
    if (<write_enable>)
      <ram_name>[<address>] <= <input_data>;
assign <output_data> = <ram_name>[<address>];
```

Although this type of memory synthesis is more efficient than using flip-flops/Latches, it still is not scalable for deeper memories. Large memory arrays will consume too many resources on the FPGA.
Due to the limited number of LUT's on the FPGA, synthesis of a 65536x8 memory array on the XC2V1500 (or XC2V3000) with LUTs is infeasible. With all LUTs configured as distributed RAM only 240 kbits of RAM are realized, while 512 kbits are required.

3.1.2 Using dedicated FPGA block RAM

The virtex2/Spartan2 FPGAs also provide blocks of dedicated on-chip RAM. These blocks can be used independently or can be concatenated to form a larger RAM. Block RAM can be used by instantiating the RAMB16 primitives or by using the appropriate Verilog code.
The following Verilog template can be used to instantiate a single port block RAM in no-change mode:

```
parameter RAM_WIDTH = <ram_width>;
parameter RAM_ADDR_BITS = <ram_addr_bits>;

reg [RAM_WIDTH-1:0] <ram_name> [(2**RAM_ADDR_BITS)-1:0];
reg [RAM_WIDTH-1:0] <output_data>;

<reg_or_wire> [RAM_ADDR_BITS-1:0] <address>;
<reg_or_wire> [RAM_WIDTH-1:0] <input_data>;

always @(posedge <clock>)
  if (<ram_enable>)
    if (<write_enable>)
      <ram_name>[<address>] <= <input_data>;
    else
      <output_data> <= <ram_name>[<address>];
```

The following example module generates a synthesizable 65536X8 memory array:

```
module nochange1 (clk, we, en, addr, di, do);
 input   clk,we,en;
 input   [15:0] addr;
 input   [7:0]  di;
 output [7:0]  do;

 reg [7:0] RAM [65535:0];
 reg [7:0] do;

 always @(posedge clk)
 begin
  if (en)
    begin
     if (we)
       RAM[addr] <= di;
     else
       do <= RAM[addr];
    end
 end
endmodule
```

The maximum available block-RAM size is 864 Kbits on the XC2V1500 and 1,728 Kbits on the XC2V3000, so this implementation is feasible and efficient for synthesis of a 65536x8 memory array.

3.2 Function using a division operator

The two simplest methods to implement a division function, besides a ROM-table, are the shift-and-subtract method and the repeated subtraction method [4]. With the latter method, the divisor is repeatedly subtracted from the dividend, until the remainder is smaller then the divisor, and the number of performed subtractions is the resulting quotient. This requires little hardware, but may require many iterations.
The shift-and-subtract method (see CH4), is the well-known paper-and-pencil division method. The divisor is right-shifted, compared to the remainder, and subtracted if smaller then the remainder. The quotient is formed by left-shifting it each iteration, and adding '1' when the divisor is smaller then the current remainder. The hardware for this method is more complicated due to the need for control and the increase in operations, but the number of iterations is reduced considerately with respect to the repeated subtraction method.
Therefore, this method is chosen for implementation of the division function.
Figure 3.1 shows the structural components of an implementation of the shift-and-subtract division method.

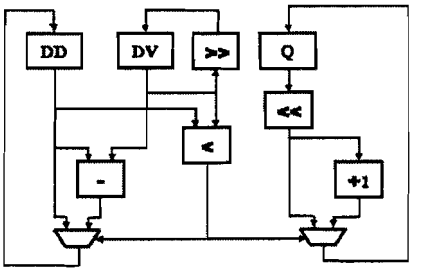

Figure 3.1: Structural components of a shift-and-subtract division algorithm implementation

The dividend/remainder (DD) is compared to the divisor (DV). The result of this comparison controls whether or not a subtraction is performed, and whether or not a '1' is added to the quotient register. At the end of each iteration, the divisor is right-shifted.
The division function is implemented by a behavioral Verilog description integrated in a Finite State Machine (appendix A) for control and timing purposes. The state transition diagram for this FSM is shown in figure 3.2.

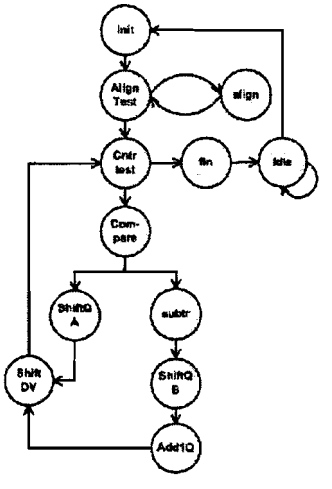

Figure 3.2: State transition diagram of the FSM-implementation of a division function realized by a shift-and-subtract division algorithm.

After a "reset" or "newdata" signal, the FSM is in its *init* state, where its registers are initialized (the result register Q is cleared, inputs are latched in, and working registers are filled with the proper values). After initialization, the system is in the *aligntest* state, where the divisor register is checked for alignment. If the register is not aligned, i.e. its msb is not the leftmost bit of the register, a shift-left operation is performed in the *align* state, and a counter is updated to keep track of the number of applied alignment steps to control the number of algorithm iterations in the later states. After the shift operation, the system returns to the *aligntest* state, and alignment is checked again. This process continues until the leftmost bit of the divisor register is a one.
When the divisor register is aligned, the *countertest* state is reached, where the abovementioned counter's value is checked. When the counter is greater then zero, an algorithm iteration is performed. First, in the *compare* state, the divisor is compared to the current remainder (where the first remainder is the dividend). If the divisor is smaller then or equal to the remainder, it is subtracted from the remainder in the *subtract* state, and the quotient register is updated in the *shiftQ_B* and *addtoQ* states.
If the divisor is greater then the current remainder, the quotient register is updated in the *shiftQ_A* state. After updating the quotient register, the divisor is right-shifted one step for the next iteration and the program counter is decreased in the *shiftDV* state.
When the program counter reaches zero, the division algorithm is finished and outputs are latched in the *finished* state. The system stays in an *idle* state until a "reset" or "newdata" signal arrives at its inputs.

3.3 Multiple clock domains

For the realization of multiple clock signals from one external clock, the Virtex2 Digital Clock Manager's (DCM's) can be used [2,3]. These dedicated on-chip clock control blocks can be used for clock synthesis, synchronization, de-skew and phase control. A DCM-primitive can be instantiated directly as a sub module using Verilog language templates. By connecting the appropriate I/O ports and setting DCM parameters, different clock domains can be created.
A DCM primitive has 9 clock output ports:

- 4 phase shifted clock signal outputs (CLK0,CLK90,CLK180,CLK270 for 0, 90,180 and 270 degree phase shifting)
- 2 double-frequency outputs (CLK2X and CLK2X180)
- 1 frequency divided output (CLKDV)
- 2 frequency synthesis outputs (CLKFX and CLKFX180)

The important parameters are:

- *CLKDV_DIVIDE* : This parameter sets the frequency division constant for the CLKDV output. Allowed values are 1.5,2.0,2.5,3.0,3.5,4.0,4.5,5.0,5.5,6.0,6.5,7.0,7.5,8.0,9.0,10.0, 11.0,12.0,13.0,14.0,15.0 or 16.0
- *CLKFX_DIVIDE and CLKFX_MULTIPLY* : These parameters set the multiply/divide factors for the CLKFX output, where CLKFX=CLKin*M/D. Allowed are integers from 1 to 32 for D, and integers from 2 to 32 for M.
- *CLK_FEEDBACK* : This parameter sets the clock feedback, used for stabilization and synchronization of clock signals. Allowed values are NONE, 1X or 2X.
- *DFS_FREQUENCY_MODE* : This parameter sets the frequency mode. There are two frequency modes, HIGH and LOW, each with its own minimum and maximum frequency ranges for the different output ports.

In this particular case, goal is to create 125, 250 and 20 MHz clock signals from a 25,30,33,40,45,50,55,60,62.5,66,67.5,70,77.5,80,83 or 90 MHz external clock. This is realized by using two different DCM's. The first generates the 20Mhz clock signal by selecting the external clock to run at 50 MHz and setting its CLKDV_DIVIDE at 2.5. The CLKFX is set to 2.5 to

generate a clock of 125 MHz. This signal is input to the second DCM, whose CLK0 and CLK2X outputs generate the 125 MHz and 250Mhz clock signals. The port connections for two DCM's are shown in figure 3.3.

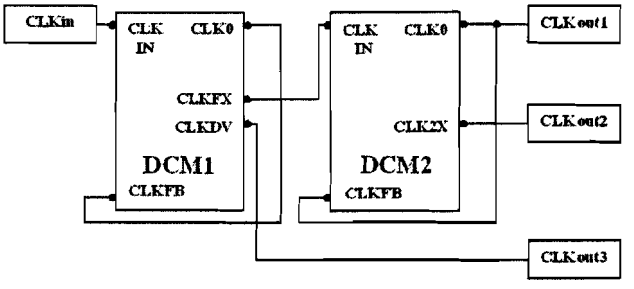

Figure 3.3: Using two Digital Clock Managers to create 20,125, and 250 MHz clock signals from one external 50Mhz clock.

All DCM inputs and outputs are buffered with an IBUFG input buffer and BUFG output buffers.

4 Binary division algorithms

In this chapter, an overview of binary division algorithms is given. In general, there are two classes of binary division methods: digit recurrence algorithms, and multiplicative algorithms [5,6]. These are discussed in the first sections. After the classic algorithms, a division algorithm based on binary number search is discussed. The chapter is concluded with a motivation for the selection of division algorithms for implementation.

4.1 Sequential division

Sequential division is the most basic and well-known division algorithm. It consists of a series of shift and subtract operations, yielding one quotient digit each iteration. In general, given a dividend X and a divisor D, the quotient Q is given by $X = Q \cdot D + R$, with $R<D$ a possible non-zero remainder. Since a division is often preceded by a multiplication, the dividend X is assumed a double-length register, while all other registers are assumed single-length.

4.1.1 Restoring division

For restoring division, the recurrence relation is given by Eq.4.1.

$$r_i = 2 \cdot r_{i-1} - q_i \cdot D \tag{4.1}$$

The digit set for the quotient q_i is $\{0,1\}$. With each iteration, the previous remainder is left-shifted one position and the divisor is subtracted from this remainder. If the result of this subtraction is positive (or no overflow occurred), the result is kept and bit q_i of the quotient is set to 1. If an overflow occurred, q_i is set to zero, and the previous remainder is restored from memory or by adding D. Figure 4.1 shows the transfer function for the remainder of the restoring division.

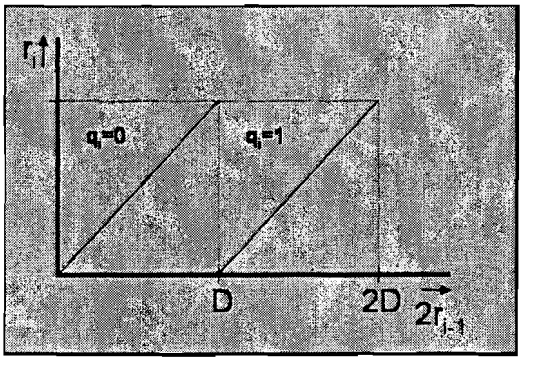

Figure 4.1: Robertson diagram for the restoring division algorithm. When $2r_{i-1}$ is greater then divisor D, digit q_i of the quotient is set to 1, else q_i is set to 0 and the previous remainder is restored.

The quotient digit selection (QDS) is given by equation 4.2.

$$q_i = \begin{cases} 1 \;\; if \;\; 2r_{i-1} \geq D \\ 0 \;\; if - 2r_{i-1} < D \end{cases} \tag{4.2}$$

For a register-width of n bits, this algorithm requires n shifts, n subtractions and on average n/2 restoring steps.

4.1.2 Non-restoring division

Non-restoring division has the same recurrence relation as restoring division (Eq.4.1), but the digit set for q_i is now {-1, 1). Again, the divisor D is subtracted from the shifted previous remainder, and if the result has no overflow q_i is set to 1. If an overflow occurred, no restoring action is taken, but q_i is set to -1 and instead of a next-step subtraction, an addition is performed in the next iteration.
Fig.4.2 shows the transfer function for the remainder for non-restoring division.

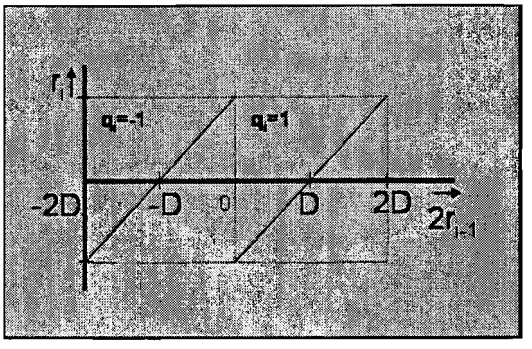

Figure 4.2: Robertson diagram for the non-restoring division algorithm. Instead of setting $q_i = 0$ and restoring the remainder when overflow occurs, the quotient digit is set to -1, postponing correction by adding D in the next iteration.

As shown in the diagram, the QDS is given by Eq.4.3.

$$q_i = \begin{cases} 1 \;\; if \;\; 2r_{i-1} \geq 0 \\ -1 \;\; if - 2r_{i-1} < 0 \end{cases} \qquad (4.3)$$

With restoring division, if a quotient digit is set to 0, the remainder $2r_{i-1}$ is restored, and in the next iteration it is shifted and D is subtracted again, yielding $r_i = 4r_{i-1}-D$.
With non-restoring division, the negative remainder is kept, while in the next iteration it is shifted and D is added, yielding $r_i=2(2r_{i-1}-D)+D = 4r_{i-1}-D$. So even though non-restoring division does not correct overflow, it yields exactly the same remainder each iteration.
The correction for the occurrence of overflow during the execution of the algorithm is performed afterwards on the complete quotient register, transforming the digit set from {-1,1} back to {0,1} by masking the negative terms, taking their two's complement and add it to the mask of the positive terms, yielding the quotient in two's complement form.

For a register-width of n bits, this algorithm requires exactly n shifts and n subtractions.

4.2 SRT division

The SRT algorithm (Sweeny, Robertson and Tocher) was developed as a speedup of the sequential division algorithm. In the SRT algorithm, a redundant digit set is used to introduce a selectable digit for which no addition or subtraction is needed. Further optimization is obtained by conditioning the divisor to force quotient digits into this redundant selection region as much as possible.

Since this algorithm is another form of a shift and subtract method, its recurrence relation is again given by Eq.4.1. The digit set for the quotient is now {-1,0,1} and the QDS is given by equation 4.4.

$$q_i = \begin{cases} 1 \ \ if \ \ 2r_{i-1} \geq D \\ 0 \ \ if \ \ -D \leq 2r_{i-1} < D \\ -1 \ \ if -2r_{i-1} < -D \end{cases} \qquad (4.4)$$

This digit set is redundant, providing means for selection of a digit for which no add or subtract operation is needed. However, to determine the quotient digits, a full comparison of the remainder to +/- D is needed. This comparison step can be sped up by requiring a normalized divisor, $0.5 \leq |D| < 1$, leading to the QDS of equation 4.5.

$$q_i = \begin{cases} 1 \ \ if \ 2r_{i-1} \geq 0.5 \\ 0 \ \ if \ -\frac{1}{2} \leq 2r_{i-1} < 0.5 \\ -1 \ \ if \ 2r_{i-1} < -0.5 \end{cases} \qquad (4.5)$$

Now the remainder is compared to +/- 0.5, (1.1 and 0.1 in two's complement representation) reducing the fan-in of the comparison circuit from an upper rounded $^2\log(D)$ to only 2. Figure 4.3 shows the transfer function for the remainder for SRT division.

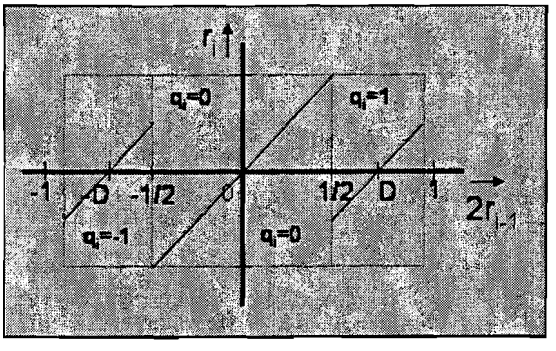

Figure 4.3: Robertson diagram for SRT division. Using a redundant digit set creates a region in the transfer function where no add/subtract operation is needed (q = 0).

Statistical research has shown that if $0.6 \leq D \leq 0.75$ the total required number of operations is minimal. Therefore improvement of speed is achieved by further normalizing D into this optimal region. This can be done by changing the comparison constant k (which normally is 0.5), or by subtracting multiples (like 2D or D/2) of D each iteration, avoiding long sequences of -1's or 1's by forcing as much digits to 0 as possible.

4.3 High-radix division

With high-radix division, the radix of the division process is increased from 2 to $\beta=2^m$, setting m quotient digits each iteration. The recurrence relation is given by equation 4.6.

$$r_i = \beta \cdot r_{i-1} - q_i \cdot D \qquad (4.6)$$

Because quotient digit selection with high radix division is more complicated, a redundant digit set $\{-\alpha,-(\alpha-1),...,-1,0,1,...,(\alpha-1),\alpha\}$ is used. The quotient digits are chosen such that $|r_i| \leq k|D|$, so $\alpha \geq k(\beta-1)$, where k is a measure for the redundancy. This redundancy results in freedom of choice for the quotient digits, so suitable comparison constants can be chosen to ensure fast digit selection. Figure 4.4 shows the remainder transfer function for high-radix SRT division.

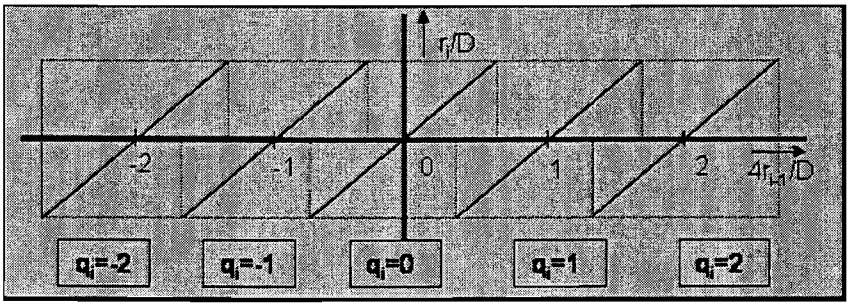

Figure 4.4: Robertson plot for high-radix division. The overlap in the transfer function is due to a redundant digit set, providing means for efficient quotient selection methods.

By rewriting Eq.2.6 ,the partial remainder $P = \beta r_{i-1} = r_i + q \cdot D$ is obtained. Since the maximum partial remainder for which q can be selected depends on the maximal allowed r_i, the bounds of the partial remainder are dependent on the divisor. Therefore:

$$\begin{cases} Pmax = (k+q) \cdot D \\ Pmin = (-k+q) \cdot D \end{cases} \qquad (4.7)$$

This relation between remainder and divisor is shown in the P-D plot of figure 4.5.

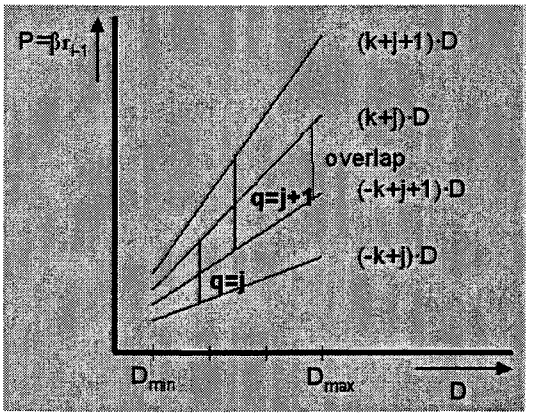

Figure 4.5: P-D plot for two consecutive values of q. The overlap region provides freedom of choice for the comparison constant.

The value of P in the overlap region is the comparison constant to distinguish between $q = j$ and $q = j+1$. If $(k+j) \cdot Dmin \geq c \geq (-k+j+1) \cdot Dmax$, P can be a single value for the whole region of D (horizontal line in the PD-plot), else P will be a staircase function for which the stepping

points determine the number of bits needed to represent P. The maximum step width and step height are given by equations 4.8 and 4.9.

$$\Delta X = D2 - D1 = P \cdot \frac{2k-1}{j(j+1)+k(1-k)} \qquad (4.8)$$

$$\Delta Y = (2k-1) \cdot D \qquad (4.9)$$

The high-radix division algorithm reduces the number of iterations from n to n/m, at the cost of more complex quotient digit selection. Per iteration at most one shift and one add/subtract operation are required.

4.4 Division by multiplication

Multiplicative division algorithms do not compute the quotient directly, but use successive approximations to converge to the quotient. Normally, such algorithms only yield a quotient, but with an additional step the final remainder can be computed, if needed. Computations include several multiplications each iteration, so these methods are only applicable when fast multipliers are available.

4.4.1 Newton-Raphson algorithm

The Newton-Raphson method uses Newton's algorithm to iteratively find the inverse of the divisor, and multiplies this inverse with the dividend to obtain the quotient. In general, a Newton-Raphson algorithm iteratively finds the roots of a function f(x) with equation 4.10.

$$x_{i+1} = x_i + \frac{f(x_i)}{f'(x_i)} \qquad (4.10)$$

In fact, this is an approximation of the function itself by its tangent at x_i. The function $f(x) = \frac{1}{x} - D$ has a zero at x = 1/D, and its derivative is $f'(x) = -\frac{1}{x^2}$. With Eq.4.10, this results in equation 4.11.

$$x_{i+1} = x_i(2 - D \cdot x_i) \qquad (4.11)$$

With first approximation x_0=1, this recurrent equation converges quadratically to 1/D. After multiplication with the dividend, Q is obtained.

With registers of n bits, this algorithm needs $^2\log(n)$ iterations with two dependent multiplications and one subtraction each, and one final multiplication to get the quotient.

4.4.2 Goldschmidt algorithm

The Goldschmidt algorithm multiplies both numerator and denominator with the same factor each iteration, converging the denominator to 1, and the nominator to the quotient. It is derived from the Newton-Raphson algorithm.
To obtain a quotient A/B, the iterative nominator and denominator are defined by equation 4.12.

$$\begin{cases} N_i = A \cdot x_i \\ D_i = B \cdot x_i \end{cases} \qquad (4.12)$$

Rewriting Eq.4.11 to $x_{i+1} = x_i \cdot F_i$, choosing $F_i = 2 - D_i$, and multiplying with A and B yields equation 4.13.

$$\begin{cases} N_{i+1} = N_i \cdot F_i \\ D_{i+1} = D_i \cdot F_i \end{cases} \tag{4.13}$$

x_i converges to 1/B, so D converges to 1 and N converges to the quotient A/B.

As with the Newton-Raphson method, the Goldschmidt algorithm requires a subtraction and two multiplications per iteration, and converges in $^2\log(n)$ steps, but now the multiplications are independent and can be executed in parallel.

4.5 Division with a binary search algorithm

Division, in general, is basically a search for a number in some bounded interval. Given a dividend X and divisor D, the goal of a division is to find Q=X/D, which will be, assuming only fixed-point integer numbers, at least somewhere in the interval [0, X].
Usually an estimate for Q is set or computed, and multiplied with the divisor to allow comparison with the original dividend (or remainder). After comparison the interval for Q is reduced, since now it is known whether the actual Q is smaller or bigger than the estimated Q. Iteration continues until the range for Q is within desired precision.
In most applications however, due to the cost in area, multiplications are avoided or one of their operands is restricted to a power of two, so it can be replaced by a shift operation.
Shift-and-subtract division methods apply a form of a search algorithm, with abovementioned limitations to the estimate for Q. Each iteration, the divisor is multiplied with a power-of-two-estimate of Q (starting with the largest possible fitting in the quotient register) and compared to the current remainder. Depending on the result of the comparison of the estimate and the remainder, the correct half of the interval in which Q will be is chosen by setting the appropriate bit in the quotient to 0 or 1. With a dividend of n_X significant bits and a divisor of n_D significant bits, n_X-n_D+1 subtracts and shifts are required to reduce the range of Q to within 1 bit precision. Since Q has at most n_X-n_D+1 significant bits, its initial range would be $[0,2^{nx-nd+1}-1]$. Every step a quotient digit is set, effectively halving the range, but always requiring the fixed number of iterations n_X-n_D+1.
The availability of a fast hardware multiplier removes the power-of-two restriction on the multiplication operands, offering more freedom of choice for the estimate for Q. Since now the estimate can be any binary number, a more efficient binary search method can be used. With a binary search, the estimate for the quotient is exactly the middle of the interval in which Q will be, so each next-step interval is half the size of the previous one.

4.5.1 Basic search method

In the case of a division, dividend X and divisor D are known, and quotient Q is to be searched. The initial interval of possible values for Q is determined by the msb's of X and D. In general, for a binary number with n significant bits, its maximal value is 2^n-1 and its minimal value is 2^{n-1}. So given the significant bits of the dividend and divisor, n_x and n_d, Q is approximately bounded to the interval $[2^{nx-nd-1},2^{nx-nd+1}]$.
Based on this interval, the quotient is estimated, and a feedback on the quality of this estimate provides the means to improve a next-round estimate. The actual quotient is not known, so the estimate $\hat{Q}$ can not be compared to it directly to provide feedback. However, since the divisor D is fixed and the dividend/remainder X is proportional to Q, an estimate $\hat{X}$ can be used for the comparison. Therefore, a remainder estimate $\hat{X}$ is generated by multiplying $\hat{Q}$ with D.

The general approach is to choose some estimate $\hat{Q}$ based on the current interval, and multiply this $\hat{Q}$ with D to obtain $\hat{X}$, which will be compared to the actual X. Depending on this comparison, the estimate $\hat{Q}$ will be either the new upper bound or the new lower bound for Q, setting the next-step interval. Based on this new interval, a new estimate can be calculated, and so on.
The most obvious estimate for Q is half of the interval, resulting in a binary search. With each iteration, the range for Q is halved, as is the case with shift and subtract methods, so this approach would require, including 1 extra iteration due to estimate round-offs and a starting interval that is not necessarily a power of 2, at most n_x-n_D+2 iterations to find Q. However, especially in the later iterations when the range is small, there is a non-zero chance that an estimate $\hat{Q}$ is exactly equal to Q, resulting in instant full-precision convergence.
In some cases, when the quotient is in the lower or higher region of the interval, computing an estimate based on ¼ or ¾ of the interval instead of ½ could result in faster convergence. Or when a new upper bound is established, it is best to find a new lower bound as close to this upper bound as possible. Using the ¾ estimate in such a case could reduce the interval for Q much faster. However, since information about the location of Q in the interval is not known a-priori, this would only work for some distinct cases, reducing performance in all other cases. Best overall results are obtained with half-range estimates.

4.5.2 Subtraction of the dividend estimate

With the basic binary search method, the quality of the estimate does not speed up or slow the reduction of the interval for Q. When, for instance, the estimate $\hat{Q}$ is just one lsb smaller then the actual value Q, the interval is still only halved, and the next estimate will again be half-range, which will be a worse estimate then the previous one.
However, for quotient estimates that are under the actual value of Q, subtraction of the dividend estimate $\hat{X}$ results in a remaining dividend X' whose value depends on how close $\hat{Q}$ was to the actual Q. If the estimate was close, using the msb of X' for the new bounds on Q could result in a much smaller interval then half the previous range (and consequently a much better next estimate for Q). In the worst case, using the msb of X' results in an interval equal to the previous. This is show in figure 4.6, which show a logarithmic plot of the quotient interval for subsequent iterations.

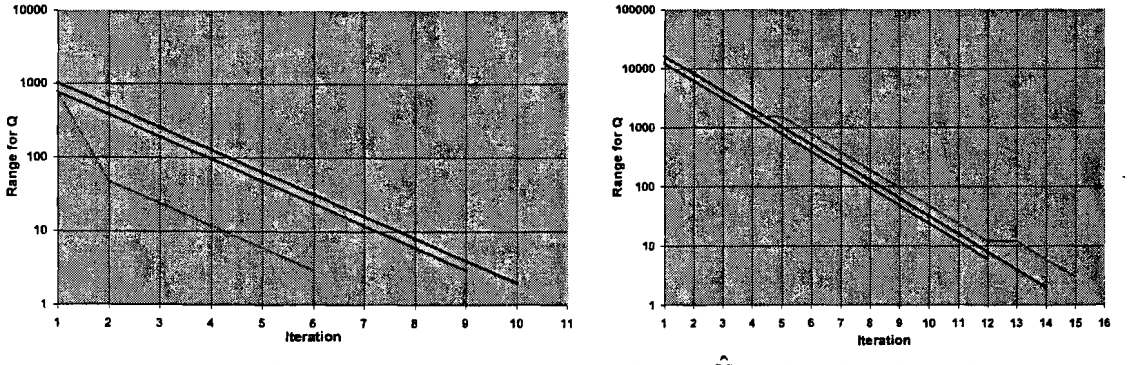

Figure 4.6: *Left:* Subtraction of a dividend estimate $\hat{X}$ smaller then the real dividend X (green line) can result in faster convergence then without subtraction (red line). *Right:* In worst-case, a dividend estimate subtraction leads to a quotient interval equal to the previous.

Subtraction of the dividend estimate exploits the quality of a quotient estimate to improve subsequent quotient estimates, i.e. passing information about the actual location of Q in the total interval.

4.5.3 Algorithm summary

The algorithm for division by binary search can be summarized as follows:

1. While (N>D):
2. Determine msb's of X and D and use these with general bounds $Q \in \left[2^{n_X - n_D - 1}, 2^{n_X - n_D + 1}\right]$ to get new lower and upper bound on Q.
3. Compute half-range quotient estimate $\hat{Q} = \frac{1}{2}(Q_{\min} + Q_{\max})$
4. Compute dividend estimate $\hat{N} = \hat{Q} * D$.
5. Compare $\hat{N}$ with N.

6a. If $\hat{N} > N \rightarrow Q_{\max} = \hat{Q}$ and go to step 2.

6b. If $\hat{N} \leq N \rightarrow N = N - \hat{N}$, $Q = Q + \hat{Q}$ and go to step 1.

4.6 Selection of algorithms for implementation

As mentioned in chapter 2, the fastest way to perform a division operation is by table-lookup. The latency is always only one clock pulse, but the table size, and therefore the amount of chip area, grows exponentially with the input data width.
Implementation of a shift-and-subtract division function (Chapter 3.2) offers more scalability to the input data width, but introduces a huge latency.
The division algorithms presented in sections 4.1.2 - 4.5 aim to reduce this latency. A selection has to be made among those algorithms, choosing those candidates that are most promising with respect to complexity and expected latency for implementation. After implementation, the speed, latency and chip area of the designs will be compared (Chapter 5.4).
The (potential) clock speed of an implementation is for a great deal limited by the amount of combinational logic between two consecutive flip-flops or latches. This, in turn, is determined by the complexity of the partial function of the algorithm that is realized by that logic.
Latency is the total time between the intake of new inputs and the presentation of the full-precision result. It is predominantly determined by the algorithm execution time, i.e. its clock speed multiplied with the number of required iterations.
Figure 4.7 shows the number of algorithm iterations as a function of the number of input operand bits for the division algorithms discussed in the previous sections [5].

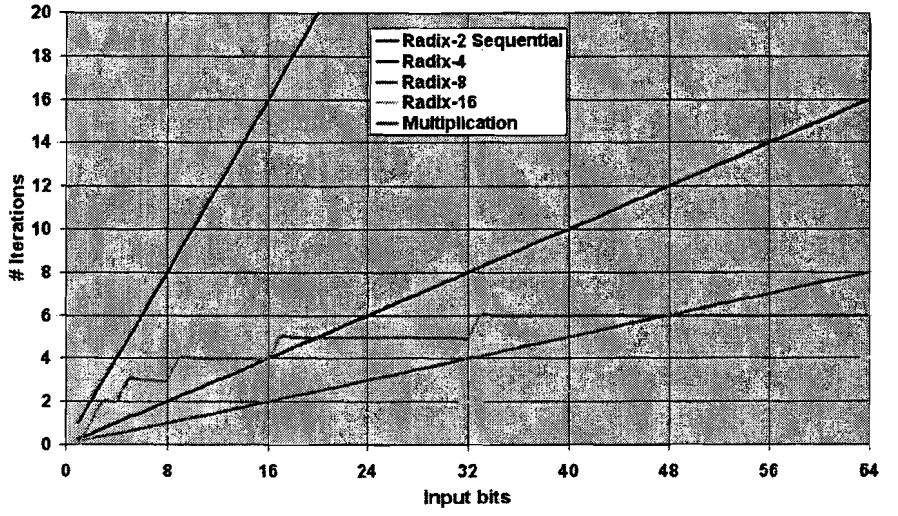

Figure 4.7: Number of required iterations as a function of the number of input bits for sequential division (blue line), high-radix division (red, orange and purple line) and multiplicative division (green line). The line for binary search division is missing, because this algorithm does not have a fixed number of iterations.

The graph shows that all subtractive methods scale linearly with the number of input operand bits. Higher radix algorithms set more digits per iterations, so these require less iterations at the cost of more complex digit selection logic. The binary search division method is not shown in the graph, since this method does not have a fixed number of iterations. With this method, the number of iterations is dependent on the input data width as well as the data itself, requiring roughly the same or less iterations then radix 2 subtractive division. Multiplicative division scales logarithmically with the number input bits, but these algorithms impose extra requirements on the input format and may require pre-scaling.

To obtain a relatively complete comparison in the limited amount of time available, one algorithm based on each of the three division principles is chosen for implementation. Since it is the standard and most basic method with the lowest complexity, the radix-2 *restoring division* algorithm is chosen to represent the subtractive methods. Latency will be very poor, since many iterations are required, but clock speed is expected to be high because of the low-complexity. The *binary search method* is implemented to aim at a divider with low or reasonable complexity and an overall performance that is competitive or better then with the subtractive methods. The latency is expected to be better, since the algorithm requires equal or less iterations. Clock speed is expected to be about the same as for restoring division.
For the multiplicative algorithms, *Goldschmidt* division is chosen for implementation over Newton-Raphson division because its two multiplications can be performed in parallel (both algorithms require the same hardware). Latency is expected to be very low, and since the FPGA on which it is implemented contains multiple dedicated hardware multipliers, the use of multiplications should not force clock speed down considerately.

4.7 Recent Developments

Recently, a hybrid division algorithm [7] has been developed, combining digit recurrence methods and multiplicative methods by employing Prescaling, Series expansion and Taylor expansion (PST).
Many digit recurrence algorithms, like SRT division, require a table lookup to obtain partial quotients. For higher radix division, this lookup can require lots of memory, while still converging linearly to the required precision. Using a prescaling method to obtain a scaling factor from a lookup table can reduce the amount of memory required for a division operation, but the table size still scales exponentially with the required precision of each partial quotient.
Multiplicative algorithms on the other hand, converge to the result starting from a rough estimation, requiring little memory. However, these methods have a large computational load, leading to a large chip area and high power consumption.
The PST algorithm combines prescaling, series expansion and zero-order Taylor expansion to reduce memory requirement and to boost overall performance.

A N-bit division operation with the PST algorithm requires, just like the Newton-Raphson and Goldschmidt algorithms, fixed point fractions as input format, a divisor between 0.5 and 1, and a dividend smaller then the divisor.

Starting with the prescaling step, dividend A and divisor B are scaled by a factor E_0, which is an estimate for the reciprocal of B. The scaling factor is obtained by taking the reciprocal of the up-rounded (starting from bit-position M+2) divisor B by a table lookup, and truncating it at the M+1 bit:

$$E_0 \cong \frac{1}{B} = trunc\left(\frac{1}{B_{[M+2]}}\right)_{M+1} = 1.e_1^0 e_2^0 ... e_{M+1}^0 \qquad (4.14)$$

Both divisor and dividend are multiplied with this factor, bringing the dividend A_1 close to the quotient, and the divisor B_1 close to 1:

$$\begin{aligned} A_1 &= A \cdot E_0 = 0.a_1^1 a_2^1 ... a_{N+M+1}^1 \\ B_1 &= B \cdot E_0 = 0.11...1 b_{M+1}^1 b_{M+2}^1 ... b_{N+M+1}^1 \end{aligned} \qquad (4.15)$$

Secondly, by series expansion, the reciprocal of B_1 is arithmetically estimated and truncated at the 2M bit by inverting B_1.

$$E_1 = trunc\left(\overline{B}_1\right)_{2M} = 1.00...0 \overline{b}_{M=1}^1 \overline{b}_{M+2}^1 ... \overline{b}_{2M}^1 \qquad (4.16)$$

The effort put in each the first two steps is a tradeoff between memory size and computational load. The total effort put in those two steps determines the precision of each partial quotient, hence the algorithm latency.
In the last step, partial quotient $\tilde{Q}_j$ and partial remainder R_j (with first remainder $R_0 = A_1$) are iteratively calculated until the required precision is achieved:

$$\tilde{Q}_j = trunc\left(trunc\left(R_{j-1}\right)_{2M} \cdot E_1\right)_{2M} = 0.\tilde{q}_1^j \tilde{q}_2^j ... \tilde{q}_{2M}^j \qquad (4.17)$$

$$R_j = 2^{2M-2} \cdot \left(R_{j-1} - B_1 \cdot \tilde{Q}_j\right) \qquad (4.18)$$

The quotient is the sum of the intermediate partial quotients:

$$Q_j = Q_{j-1} + 2^{-(2M-2)\cdot(j-1)} \cdot \tilde{Q}_j \qquad (4.19)$$

Each new remainder in Eq. 4.18 has 2M-2 leading zero, so the total number of iterations J is given by:

$$J = \left\lceil \frac{N}{2M-2} \right\rceil \qquad (4.20)$$

The final quotient Q is obtained by removing Q_t, which is the tail of Q_j beyond bit position N:

$$Q = trunc\left(Q_j\right)_N \qquad (4.21)$$

The final remainder is recovered by adding the product of Q_t and B_1:

$$R = R_j + Q_t \cdot B_1 \qquad (4.22)$$

If this remainder is greater than or equal to B_1, both remainder and quotient need to be corrected by increasing the lsb of Q:

$$\begin{cases} Q = Q + 2^{-N} \\ R = R - B_1 \end{cases} \qquad (4.23)$$

5 Implementation of division algorithms

In this chapter, the implementation of the selected division algorithms is discussed. First, the restoring division algorithm is discussed, followed by two implementations of the binary search divider. The chapter is concluded by an implementation of the Goldschmidt division algorithm. Verilog modules realizing the functions are included on CD-ROM. Example of some constructs are given in appendix A, general Verilog constructs and synthesis with ISE software can be found in [8,9,10,11].

5.1 Restoring division

Restoring division is the most basic of the shift-and-subtract division algorithms discussed in the previous chapter. The divisor is subtracted from a left-shifted remainder, and by evaluating the sign bit of the subtraction, the decision is made to keep the result of the subtraction or to restore the previous value of the shifted remainder. At the start of an iteration, the quotient register is left-shifted, and based on the keep/restore decision, the lsb of this register is set to '1' or '0'. Figure 5.1 shows an implementation of this algorithm.

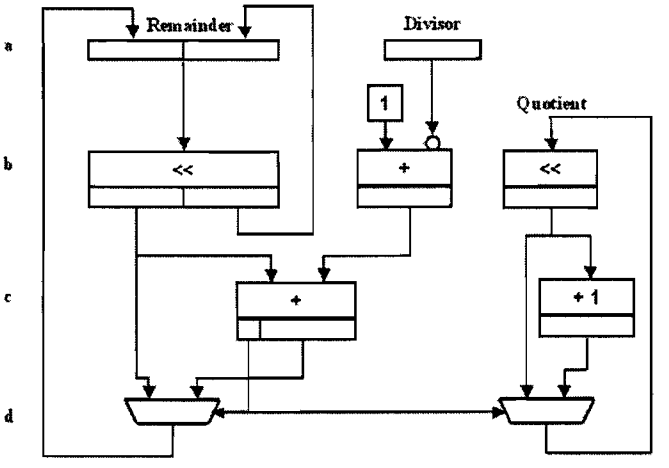

Figure 5.1: Implementation of the restoring division algorithm. Based on the sign bit of the addition of the remainder and the two's complement of the divisor, the result is kept or the previous remainder is restored.

The implementation is divided into four stages. In the first stage *(a)*, new data is latched into the remainder and divisor registers and for both a sign bit is added. The remainder and quotient registers are then left-shifted *(b)*, while the divisor is two's complemented. During the third stage *(c)*, the shifted remainder is added to the complemented divisor. Since the remainder register is double the length of the divisor register, a full-length addition (by concatenating zeros to the divisor) would always yield the same lower half result, so only the upper half of the shifted remainder register(including the sign bit) is used while the lower half of the shiftregister can be directly used to set the next-round lower half of the remainder.
In the last stage *(d)*, the sign bit of the addition controls how the upper half of the remainder is set for the next iteration. If the sign bit is zero, the result of the addition is used, else the upper half of the shifted remainder is used. The sign bit also controls the setting of the quotient bit. If the sign bit was zero, the lsb of the quotient register is set to one, else it is set to zero.
To provide timing and control, an FSM implementation is used. Figure 5.2 shows the state diagram for this FSM.

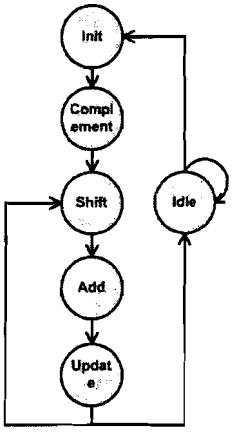

Figure 5.2: State diagram for FSM realizing a restoring division algorithm implementation.

In the *init* state, the quotient register is cleared, the program counter is reset, a new dividend and divisor are latched in, and for both a sign bit is added. In the *complement* state, which also is an initialization state, the divisor with its sign bit is two's complemented.
The algorithm starts in the *shift* state, where the remainder and the quotient registers are left-shifted and the program counter is increased by one. The significant part of the shifted remainder and the complement of the divisor are added in the add state. In the update state, the msb of the addition is checked and based on its value, the remainder is updated. If this msb is '0', the remainder is updated with the result of the addition and the lsb of the quotient is set to '1'. If the msb is '1', the remainder is updated with the result of the shift operation and the lsb of the quotient is set to '0'. Checking the value of the program counter determines whether the next state is the shift state or the idle state.

5.2 Binary search division

Chapter 4.5.3 summarizes the algorithm for binary search division. The basic approach is to generate a quotient estimate based on the msb position of remainder and divisor, multiply it with the dividend to form a comparable remainder estimate, and obtain tighter bounds on the quotient by comparing the remainder estimate with the actual remainder.
First of all, the msb's of N and D should be determined. This is realized by applying a priority encoder (appendix A) with a casex statement in Verilog. With subtraction of the msb's of N and D, a relative msb is obtained, which is used for addressing a ROM-table coded with the minimum and maximum possible values for Q as a function of this relative msb (appendix A).
The estimate for Q is either calculated by adding the minimum and maximum, or the minimum and the estimate of the previous iteration, and right-shifting the result of this addition.
The generation of an estimate for Q is shown in figure 5.3.

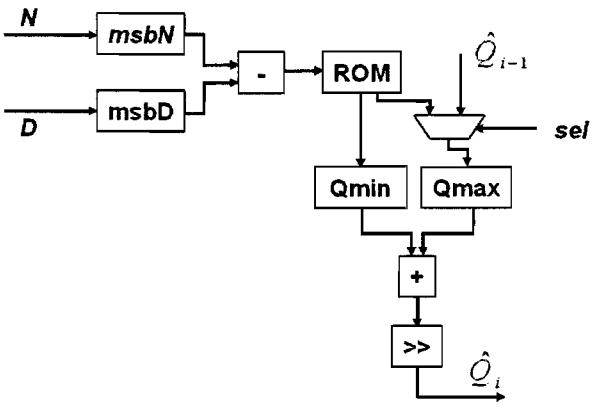

Figure 5.3: Implementation of an estimate generator of the binary search division algorithm. Based on the msb positions of the remainder and dividend, and the result of the comparison of the previous remainder and dividend estimate, an estimate for Q is generated.

Multiplication of the estimate and the dividend is realized by inferring a hardware multiplier. Comparison of the dividend estimate with the actual dividend is realized by inferring a less-then-or-equal comparator using an if-statement. The control signal produced by the comparator controls the conditional subtraction of N and $\hat{N}$, as well as the method to generate an estimate for Q and the conditional addition of the estimate to the total result. This is shown in figure 5.4.

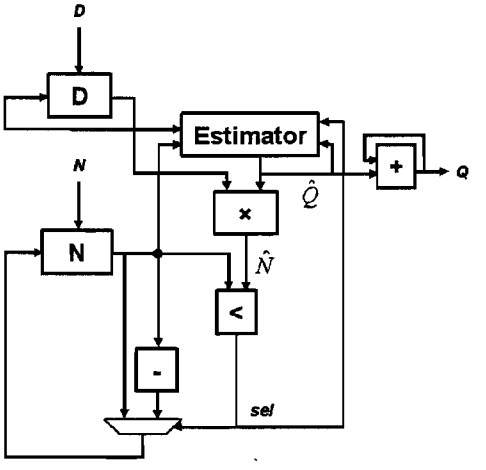

Figure 5.4: Hardware realization of the binary search division algorithm.

5.2.1 FSM implementation

The first implementation of the binary search divider is realized using a behavioral Verilog description, integrating functionality, timing and control in one FSM. The state transitions for this FSM are shown in figure 5.5.

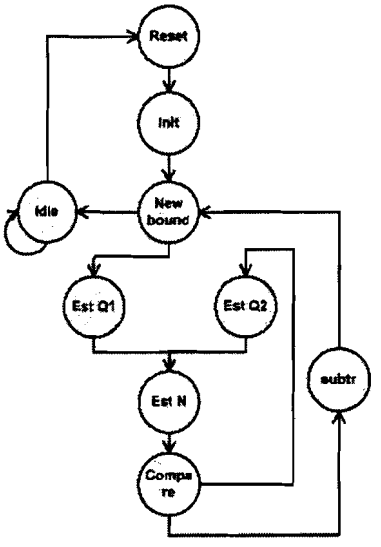

Figure 5.5: State transition diagram for the FSM-implementation of a binary search divider.

In the *reset* state, registers are cleared. After a reset, the new dividend and divisor are latched in during the *init* state. After initialization, the actual division algorithm starts. First, new bounds for the quotient are calculated in the *newbounds* state. The calculation of the msb position of N and D as well as the ROM-table lookup are performed at every level change of N,D and the relative msb. In the *newbounds* state the relative msb position is calculated by subtracting the msb positions of N and D, setting the bounds on Q by triggering the ROM-lookup. In the *estQ1* state, a quotient estimate is generated by adding the ROM-table outputs and shifting them right one position, i.e. taking the middle of the interval.
In the *estQ2* state, the quotient estimate is also generated based on the middle of the interval, but in this state, the interval is bounded by the minimum obtained from the ROM-lookup and the previous quotient estimate.
In the *estN* state, an estimate for the remainder N is calculated by multiplication of the estimate for Q with the divisor. This estimate is compared to the actual remainder in the *compare* state.
If the estimate is smaller then or equal to N, the next state will be the *subtr* state. Here, the estimate is subtracted from N, and the estimate for Q is added to the result register.
If the estimate is greater than N, the next state will be the *estQ2* state, and the result register is unchanged.
Iteration continues until the remainder is smaller then the divisor, and depending on the particular state path followed, 4 or 5 stages are required per iteration.

5.2.2 Token ring implementation

Since the FSM implementation yields an unsatisfactory maximum clock speed, a second implementation of the binary search divider is realized by partitioning functionality into small blocks. Each block is a registered module with one specific task and an extra input and output to pass a control token, resulting in a cyclic pipeline (Appendix A).
The modules each perform their specific task when they are triggered by the token, produce an output token when finished, and keep their output constant until a new input token is received.
Figure 5.6 shows a block scheme for this implementation.

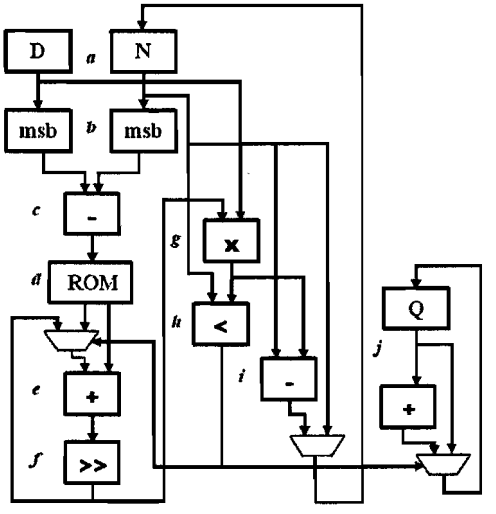

Figure 5.6: Implementation of the search divider, using a token ring for timing and control.

At the first clock pulse, new input data is latched into the numerator and denominator registers N and D *(a)*. In the next time step, the msb position of these registers are calculated *(b)*, which are then subtracted to obtain the relative msb position *(c)*.
This relative msb is used to calculate bounds on the quotient by a ROM-lookup *(d)*.
An estimate for Q is generated in step *(e)* and *(f)*. In step *(e)* the minimum and maximum for Q (based on either the ROM or the ROM and the previous estimate) are added, and shifted right in step *(f)*. The estimate is multiplied with the divisor to obtain a dividend/remainder estimate *(g)*. In step (h), comparison of the dividend/remainder estimate and the current remainder is performed, based on which a control signal for steps *(e),(i)* and *(j)* is generated. Depending on the control signal, the estimate is subtracted from the remainder *(i)* and the quotient estimate is accumulated to the total division result *(j)*. Steps *(i)* and *(j)* are performed in parallel since their dataflow is not dependent on each other. So in total, 9 stages per iteration are required.

5.3 Goldschmidt division

The Goldschmidt division algorithm requires and addition and two multiplication operations. These multiplications can be done in parallel or sequentially. Since the FPGA has many hardware multipliers available, the parallel implementation is realized.
Figure 5.7 shows a parallel Goldschmidt divider.

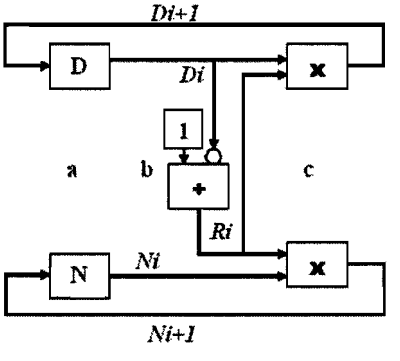

Figure 5.7: Parallel implementation of the Goldschmidt division algorithm.

The implementation is divided into three different stages. In the first stage *(a)*, new data is latched into the N and D registers. Then the multiplication factor Ri is computed by two's complementing the current contents of the D register *(b)*. In the third stage, this factor is multiplied with N and D to compute the next-round N and D *(c)*.
If the divisor is normalized to be within the range $0<D<1$ and the dividend is smaller then the divisor, the N register will converge quadratically to the quotient of the division (and D will converge to 1).
The Goldschmidt divisor is implemented by partitioning the structure in functional Verilog modules and providing timing and control by a token ring, similar to that of the divider in 5.2.2.

6 Results and discussion

In this chapter the properties of the implemented designs are discussed and compared. The chapter is concluded with an overall discussion of the implementations.

6.1 Properties of implemented designs

In this section the tested performance of the implemented designs are discussed. The results discussed in this chapter are based on the division of a 32 or 16 bits wide dividend by a 16 bit divisor, resulting in a 16 bit wide quotient and remainder.

6.1.1 Speed and area

The speed of a design can be defined as the total time needed to perform a pull-precision division, starting at the arrival of new data and ending at the presentation of the final result. It is determined by the maximum clock speed and the total number of required clock cycles.
The FPGA device utilization is used as a measure for required area. The target FPGA for the implementations is the Xilinx XC2V1500, which has 7680 slices with two LUTs and FFs each, and 48 hardware multiplier blocks on its die.

The implementation of the restoring division algorithm is based on a FSM (CH5.1). The division process itself is preceded by two initialization states, where the new data is latched in and prepared for processing. After initialization, the FSM goes through the shift, add and update states repeatedly, requiring 3 clock cycles per iteration. Since one bit per iteration is set, 16 iteration are required, so together with the initialization, the total required clock cycles for division is 50.
The maximum clock frequency as stated by the Xilinx ISE synthesis tool is 249MHz, but post-place and route (PPAR) simulation shows a maximum of 200MHz, above which several bit-errors in the simulation occur. This is probably due to setup and hold time violations, which can be caused by switching delays of routed signals and/or high pin-to-register-time. With a frequency of 200MHz, the total required division time is 250ns.
The required FPGA-resources for implementation of restoring division are 92 slices, 117 slice flip-flops and 171 four-input LUTs.

The FSM implementation of the binary search division requires two initialization steps to pre-process the data. Each algorithm iteration requires, depending on the previous iteration, 3 to 5 clock cycles. The number of required iterations is data-dependent, and can be just one in some distinct cases, as well as seventeen in worst-case.
The maximum PPAR clock frequency is 100MHz, so taking the worst-case and best-case sequence, the total required division time is 70-870ns.
The FSM implementation requires 275 slices, 191 slice FFs and 504 LUTs.

The token ring implementation has no initial delay. It requires 9 clockcycles per iteration at a maximum PPAR clock frequency of 175MHz. With these settings, the total division time will be between 52ns and 875 ns, requiring 197 slices, 197 FFs and 357 LUTs.

The implementation of the Goldschmidt algorithm does not require initialization steps. Per iteration, 3 clock cycles are required. With the maximum PPAR clock frequency at 200MHz, the total required division time is 60s, with a device utilization of 29 slices, 51 FFs and 48 LUTs.
However, the algorithm requires fixed-point fractional inputs, with the restrictions that the divisor is between 0.5 and 1, and that the dividend is smaller then the divisor. This will result in a quotient between 0 and 1, which is consistent with the fixed-point fractional representation. Therefore, to process integers and/or floating point mantissas, some pre-scaling is needed to meet this requirement. This can be realized by shifting dividend and divisor, and after finishing the

division, shifting back the quotient. For a 16 bit dividend and divisor, at most 15 shifts are needed (since the dividend and divisor shifting can be done in parallel) to scale the inputs, and at most another 15 shifts are needed to rescale the quotient. Assuming this can be processed at 200MHz, this would require 150ns extra, so the total division time would be 210ns in that case.

6.1.2 Comparison

The properties of the implemented division algorithms are summarized in table 5.1.

Table 5.1: Properties of implemented division algorithms.

	Shift/Subtract	**Binary Search**		**Multiplicative**
Area	*Restoring*	*FSM*	*Token ring*	*Goldschmidt*
# Slices	92	275	197	29
# Slice Flipflops	117	191	197	51
# LUTs	171	504	357	48
MULT18x18s	0	1	1	2
Speed				
Max. Clock	249 MHz	130 MHz	185 MHz	250 MHz
Max PPAR Clock	200 MHz	100 MHz	175 Mhz	200 MHz
# clocks/iteration	3	3 – 5	9	3
# Setup-delay	2	2	0	0 / 30
# Iterations	16	< +/- 16	< +/- 16	4
Total division time	250 ns	70ns - 870ns	52ns - 875ns	60ns / 210ns

The restoring division algorithm is essentially a more efficient implementation of the algorithm used in the division function in chapter 3.2. By left-shifting the remainder instead of right-shifting the divisor, the alignment procedure can be omitted. Better timing and control results in a more efficient state machine, only requiring 3 clock cycles per iteration instead of 6. The more efficient implementation results in a double maximum PPAR clock speed. By using a two's complement addition instead of a subtraction, the comparison step is also improved, since now only one bit is to be checked instead of the whole register[6].
However, the division function of 3.2 potentially requires less iterations, since the alignment procedures provides information about how many iterations exactly are required (the number of alignment shifts is the number of required iterations, while the restoring division algorithm always iterates as many times as its quotient register is wide).
In total, the restoring division implementation is a significant improvement on the earlier implemented division function, reducing latency from approximately 340ns for a division of a 16 bits dividend by an 8 bits divisor to 250ns for a division with twice as wide operands.

The binary search algorithm has the advantage of less required iterations over the restoring algorithm, at the cost of a multiplication instead of a shift operation during execution. Since hardware multipliers are available on the FPGA, this should not induce a significant increase in required chip area.
First, a FSM implementation with synchronous and asynchronous computation steps was realized, requiring 3-5 clock cycles per algorithm iteration, at a maximum PPAR clock speed of 100MHz. Because of this low clock speed, a token ring implementation with partitioned functionality was realized. With this implementation, 9 clock cycles per iteration are required, but the clock speed can now be increased to 175MHz.
Both implementations require approx. 50ns-875ns for a division, but the region [400,875] ns is more likely then the [0,400] ns region, so overall performance is worse compared to restoring division. However, the token ring implementation claims less chip area while requiring roughly the same time for an addition compared to the FSM implementation.
The ill performance of both binary search division implementations is caused by the more complex estimate generation process. With restoring division, the estimates are simply shifted versions of the divisor, but with binary search division, the estimates are the result of multiple calculation steps.

The Goldschmidt algorithm proves to be the fastest algorithm. Its clock speed of 200MHz and 3 clock cycles per iteration equal that of restoring division, but the Goldschmidt divider requires only 4 iterations instead of 16. With a latency of only 60ns for a 16 bit division, this implementation is the fastest by far, however, it requires normalized inputs. Even with a pre-shifter to allow non-normalized inputs, this implementation requires worst-case only 210ns vs. 250 ns of the restoring divider and 50-875ns for the binary search divider.

Figure 5.8 shows the routed layout of the implemented designs.

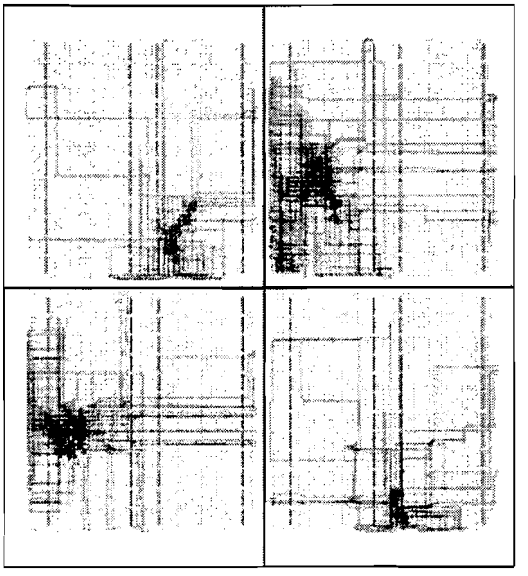

Figure 5.8: Routed layout of the implemented dividers on the FPGA. *Top-left:* restoring divider, *top-right:* binary search divider (FSM), *bottom-left:* binary search divider (token ring), *bottom-right:* Goldschmidt divider.

The figure shows the connections of the logic blocks, switch boxes and other resources on the layout of the FPGA, revealing the true required chip area of the implemented designs for this particular FPGA/synthesis tool combination.

Taking the restoring divider as a reference, the binary search divider requires about three times more chip area. As shown in table 5.1, especially the number of required lookup tables is significant compared to the other implementations. This is probably caused by the complexity of the estimate generation process (requiring two priority encoders, a subtracter, a ROM-table, an adder, a shifter and some registers).
The Goldschmidt divider however, requires about ½ to ⅓ of the area compared to the restoring divider. Even though this algorithm needs two multiplications, the increase in area is small, due to the availability of sufficient hardware multiplier blocks (if the multiplication had to be implemented using LUTs, the required area would be significantly greater). The multiplier blocks are shown in figure 5.8 as the rectangles in the vertical columns.

So, the Goldschmidt division algorithm shows to be the most efficient algorithm, introducing the least latency, requiring the least chip area, and promising the best scalability for division with wider operands.

6.2 Discussion

The fastest implementation of a division function is a table lookup. The inputs a and b are concatenated and used as input for the table. The address space of the table is coded with all possible (a,b) pairs, the contents of the addresses are coded with f(a,b) for that particular input pair. A lookup-table implementation introduces a latency of just one clock cycle, but requires a table-size that grows exponentially with the input data width. A lookup-table for f(a,b) with a,b 8 bits wide requires a table of 65536x8. This can not be implemented by using FPGA LUTs, since the amount of memory generated in this way is insufficient and inefficient. A memory array for 8 bit input operands using dedicated block RAM has been implemented. However, scaling up to 9 bit inputs proved to be infeasible.

Trading speed for scalability, an algorithmic division function has been implemented, using a shift-and-subtract division algorithm. The algorithm is based on the well-known paper-and-pencil method of shifting the divisor from left to right of the dividend, and subtracting it when it is smaller then the remainder. Such an implementation scales linearly with the input data width, but requires a number of iterations equal to the input data width. The realized implementation runs at 100MHz, performing the function f(a,b) in approximately 350ns. Although such an implementation scales linearly with the input width, so does its latency.

To look for algorithms with lower latency, a literature search on fast division algorithms has been performed. In general, there are two classes of division algorithms: digit recurrence algorithms such as the shift-and-subtract algorithms, which iteratively produce one or more quotient digits, and multiplicative division algorithms, which produce a successively more accurate approximation for the quotient. Improvements in latency for the digit recurrence algorithms are made by adding redundant digit sets, reducing the number of operations, or increasing the radix of the process. However, these algorithms still scale linearly with input operand width. Multiplicative division algorithms set quotient digits at a quadratic rate, requiring little iterations. These algorithms scale logarithmically with input width, but impose restrictions on the input format, so extra pre-shifters are needed to provide compatibility of format.

Division is basically a search operation, where the quotient is to be searched in an ordered list of possible results. Traditionally, due to the cost in area, the required multiplication step in the process is replaced by a shift operation, restricting one of the operands to a power of two (shift-and-subtract algorithms). A binary search division algorithm has been designed to exploit the availability of fast hardware multipliers on the target FPGA. Based on the msb of dividend and divisor, an interval of possible results is calculated. The middle of this interval is taken as a quotient estimate, multiplied with the divisor to form a remainder estimate, and compared to the actual remainder. The result of this comparison narrows the interval and controls a conditional subtraction of the estimate and remainder. With this algorithm, the number of iterations is data-dependent, varying from one iteration in best case to the number of input bits plus two in worst case.

One algorithm of each class has been implemented in FPGA, and compared on basis of a 16 bit division. The restoring division algorithm runs at 200MHz, requires 16 iterations of 3 clock cycles, and has a total latency of 250ns. Its chip area is taken as reference in comparison with the other algorithms.
The binary search division is implemented in two forms. The first is a finite state machine implementation, running at 100MHz, with 3-5 clock cycles per iteration. The second implementation uses the passing of tokens for timing and control, and runs at 175MHz, requiring 9 clock cycles per iteration. Both implementations have a total latency of 50-875ns, depending on the data, and require approximately 3 times the area of the restoring divider.

The Goldschmidt division algorithm is implemented in a parallel form, performing its two multiplications in the same clock cycle. The implementation runs at 200MHz, requires 4 iterations of 3 clock cycles, and has a total latency of 60ns and takes about a third of the area of the restoring divider. However, its input format is restricted to special-case fixed point fractions, but even with a pre-shifting stage to improve format compatibility, its latency would be approximately 210 ns.

The Goldschmidt algorithm proves to yield the best results in speed, area and definitely scalability. Only drawback is its input format restriction, but this can be overcome by a pre-shifting stage. The binary search division implementation performs worst, both regarding speed and area. This is due to the more complex estimate generation process compared to standard shift-and-subtract method. Improvement could be achieved by implementing more steps into a table-lookup.

With the first completed PSND re-implementation, the division operation is performed on a computer instead of on the FPGA. The raw data is sent from FPGA to a computer using an UART communication channel. The division is then performed locally by a lookup-table. This off-chip division has been chosen because of its ease of implementation, trading communication bandwidth for a shorter implementation trajectory. When there is time left for improvements, the restoring-division algorithm will be implemented to perform the division on the FPGA itself. This algorithm is preferred over the Goldschmidt algorithm because it is directly implementable without setting inconvenient input data requirements.

7 Conclusion

The aim of the work reported here was to research some of the most complex operators and data structures required for the FPGA implementation of the signal processing part of the neutron detector regarding their effectiveness and efficiency, to implement some selected operators and data structures, and to validate their implementation.

During the internship, the focus shifted towards implementation of the most complex division operation. Goal was to find division algorithms that had better speed/area/scalability properties than the initial implementations (lookup-table and pencil-and-paper division method).

For an 8 bit division operation, a lookup-table implementation has the lowest latency, at a heavy cost in memory and therefore chip area. The table size scales exponentially with input operand width, so operands greater then 8 bits are infeasible on the target FPGA.

Aiming at better scalability and lower latency, several division algorithms have been researched. Division algorithms, in general, can be categorized in digit recurrence algorithms and multiplicative algorithms. With the first category, the quotient digits are iteratively set, converging linearly to the required precision. Algorithms of the second category use fast multiplications to produce successively more accurate approximations of the quotient, converging quadratically to the required precision.

Together with our own proposed division algorithm, one algorithm from each category has been implemented in FPGA. A restoring division algorithm, our own binary search division algorithm and the Goldschmidt algorithm have been realized and tested. Comparison of these implementations shows that for wider operand division, the Goldschmidt division algorithm is most suited when hardware multipliers are available. It requires the least iterations, has the best overall latency, occupies the least area and scales logarithmically with input operand width. Binary search division performs worst, due to a complex estimation process, but performance is expected to improve if part of this process is done by table-lookup. The restoring division algorithm is the most simple and straightforward. It is easy to implement, but has a high latency.

Currently, because of the ease of implementation, the division operation is performed by table-lookup on a computer, instead of on the FPGA itself. When there is time left for improvements, the restoring division algorithm will be used on the FPGA to perform the division. This algorithm is preferred over the Goldschmidt algorithm, because it does not impose input format requirements on its operands, as is the case with Goldschmidt division.

With the realization of a clock managing unit, a 65536x8 memory array, an algorithmic division function and several implementations for latency-improvement of this division function including an implementation of a binary search divider, the assignment can be considered successfully and fully completed.

References

[1] P.Klimczak, *Ontwerp en implementatie van de positie gevoelige neutronen detector,* Technische Universiteit Eindhoven, 2006.

[2] Xilinx Inc, *Virtex-II Complete Data Sheet*, www.xilinx.com, 2005.

[3] Xilinx Inc, *Virtex-II Platform FPGA user guide*, www.xilinx.com, 2005.

[4] J.A.N.Lee, *Binary division*, http://courses.cs.vt.edu/~cs1104/BuildingBlocks/divide.010.html, 2000.

[5] I.Koren, *Computer Arithmetic Algorithms*, Prentice Hall 1993.

[6] D.Patterson, J.Hennessy, *Computer Organization & Design*, Morgan Kaufmann 1997, pp. 210-335.

[7] J.H.Liu, M.L.Chang, C.K.Cheng, *An Iterative Algorithm for FPGAs*, FPGA '06, pp. 83-89, February 2006.

[8] Xilinx Inc, ISE 6 software manuals, www.xilinx.com, 2004.

[9] D.E.Thomas, P.R.Moorby, *The Verilog hardware description language*, Kluwer Academic, 1996.

[10] M.D.Ciletti, *Modeling, synthesis and rapid prototyping with the Verilog HDL*, Prentice Hall, 1999.

[11] D.R.Smith, P.D.Franzon, *Verilog styles for synthesis of digital systems*, Prentice Hall, 2000.

Appendix A Examples of used Verilog code

State machine declaration:
A state machine can be inferred using the following Verilog coding style:

```
// Standard State Machine Deaclaration
parameter msbstate = 2; // msbstate=f(number of states)
parameter [msbstate:0] state0=0, state1=1, ...;
reg [msbstate:0]   cs;  // current state
reg [msbstate:0]   ns;  // next state

// State transitions
always@(posedge CLK or posedge RST)
begin
        if (RST) cs <= state0;
        else     cs <= ns;
end

// Next-state function
always@(cs)
case(cs)
        state0:                         ns <= state1;
        state1:  if(condition)          ns <= state2;
                 else if (condition)    ns <= state1;
                 else                   ns <= state0;
        state2:  ...
endcase

// Output function
always@(posedge CLK)
case(cs)
        state0:  begin
                         statement 1;
                 end
        state1:  begin
                         if(condition)      statement 2;
                         else               statement 3;
                 end
endcase
```

State machines are used for timing and control of functionality in the divider function, the restoring divider and binary search divider.

Priority encoder:
To determine the msb-position of a register, a casex statement can be used:

```
always@(regA)
begin
        casex(reg A)
                4'b1xxx: msbA <= 16;  // use don't cares to set priorities on position of 1's
                4'b01xx: msbA <= 15;
                   ...
                4'b0001: msbA <= 1 ;
        endcase
end
```

ROM-table:
A ROM-table can be inferred using a case statement with the same output register for every possible input. By using a relative msb position as input, such a table can be used to look up bounds on a quotient given the msb position of dividend and remainder.

```
always@(regA)  // ROM can be triggered synchronously or asynchronously
begin
        case(reg A)  // register A is set as input address for ROM
                4'b0000: regB <= value1;  // one or multiple outputs can be looked up
                4'b0001: regB <= value2;
                    ...
                4'b1111: regB <= value16;
        endcase
end
```

Token ring control:
The passing of tokens through submodules can be used for timing and control of applications. A case statement is used to handle the possible occurrence of an input token. If no token is present, the output is kept constant. The input token is fed to a flip-flop, which produces an output token the next clock cycle.

```
always@(posedge CLK)
begin
        case(input-token)
                1'b1: begin   // apply function if token on input
                        output <= function(inputs);
                      end
                1'b0: begin   // else keep output constant
                        output <= output;
                      end
        endcase
end

always @(posedge CLK)
begin
        output-token <=  input-token;  // use flipflop to pass token
end
```