VHDL Implementation of the Baseline Centroid Finder Algorithm

Shweta Lolage, Kishma Meyers, Roberto Brown, Reginald Perry

Department of Electrical and Computer Engineering
FAMU-FSU College of Engineering
Florida A&M University
Florida State University

Horst Wahl
Department of Physics
College of Arts and Sciences
Florida State University

February 17, 2000
# TABLE OF CONTENTS

1. ACKNOWLEDGEMENTS .......................................................................................................................... 3  
2. INTRODUCTION ..................................................................................................................................... 3  
3. DECODER (OR STRIP READER) BLOCK ............................................................................................. 4  
4. BUFFER FILLER (OR CLUSTER FINDER) BLOCK ........................................................................... 5  
5. CALCULATOR BLOCK .......................................................................................................................... 6  
6. ALTERA FPLD IMPLEMENTATION OF THE BASELINE DESIGN .............................................. 9  
7. SUMMARY ........................................................................................................................................... 12  
8. REFERENCES ......................................................................................................................................... 12  
9. APPENDIX A ....................................................................................................................................... 13  
10. APPENDIX B ..................................................................................................................................... 30  
11. APPENDIX C ..................................................................................................................................... 63
1. Acknowledgements

The VHDL implementation of the Baseline Centroid Finder project is a joint effort between the Department of Physics at Florida State University and the Department of Electrical and Computer Engineering (ECE) at the joint Florida A&M University-Florida State University College of Engineering. The VHDL coding was accomplished by ECE MS graduate students Shweta Lolage, Kishma Meyers, and Roberto Brown under the direction of Professors Reginald J. Perry (ECE) and Horst Wahl (Physics).

2. Introduction

This report outlines the VHDL implementation of a baseline design for the Centroid Finder algorithm as outlined in [1] and [2]. It makes the following design assumptions:

1. Functional design only (i.e. timing constraints are not considered).
2. “Hit” Filter is not implemented.
3. Chip gain and offset logic is not implemented.
4. Use Altera FPLDs as the target technology.

The centroid finder design is divided into three major partitions for implementation. These partitions include 1) Decoder (or Strip Reader), 2) Buffer Filler (or Cluster Finder), and 3) Calculator (or Centroid Finder). A block diagram of the overall design is given in Figure 1 below.

![Figure 1: Block Diagram of Centroid Finder Design](image)

The Decoder (or Strip Reader) reads the 8-bit SMT data from the VTM and formats it for processing by the Buffer Filler. Two FIFO’s are used within the Decoder block to synchronize the input and output data streams. The Buffer Filler implements the five-strip centroid finder algorithm as given in [1]. Once it determines that a centroid should be calculated, it transfers the buffer information (i.e. D1-D5) to the Calculator which performs the calculation of the centroid using the algorithm also given in [1]. Simple handshaking between the blocks controls the flow of data through the design. The output
of this design is a two-bit datatype (axial, stereo, or Z) and a 13-bit pointer address to the centroid of the cluster.

In the following sections you will find the detailed description for each of the design partitions. A simulation of the design using the Altera MAXPLUS-II package is given in Section 6. All flow charts and state diagrams are provided in Appendix A while VHDL code for each partition is presented in Appendix B. Appendix C contains the Altera GDF files used to simulate the design.

3. Decoder (or Strip Reader) block

**Fig. 2: Block Diagram of the Decoder block**

**Description:**

An internal block diagram of the Decoder block is shown in Figure 2. A flow diagram of the design is given in Appendix A and the VHDL code for the Decoder is provided in Appendix B.

- **Input FIFO**

  The input FIFO is responsible for receiving all the data that the VTM is sending to the centroid finder. The depth of the FIFO is 256 words to hold the maximum event size while its width is set at 8 bits. The Altera LPM_FIFO_DC is used to implement this part of the design. This LPM has separate read and write clocks so that the FIFO can be simultaneously accessed. Default LPM_FIFO_DC settings were used for the centroid finder implementation. The "fifo_empty" output is used to indicate whether data is available within the FIFO for processing by the Buffer Filler.

- **Decoder**

  This unit is responsible for reading the information out of the input FIFO and processing it according to the algorithm given in [2]. The decoder will read and compare information such as
the sequencer and hdi and chip identification number to make sure that a valid frame exists before it continues with it's operation. Once a valid frame has been found, the decoder proceeds to select and store the required information into the 100x23 output FIFO.

- **Lpm ROM**

This 2048x9 memory device works in conjunction with the decoder by saving the chip and channel id for later checking to verify the validity of the channel. If the channel is recorded as “bad”, the decoder will ignore it’s data. Also, the LPM ROM contains channel 8-bit offset information which is used to correct the data of the “good” channels.

- **Output FIFO**

This FIFO uses the same LPM_FIFO_DC macro as the input FIFO, with the difference that it's 100 words deep by 23 bits wide. Also, it's write function is controlled by the decoder module while its read function is controlled by the centroid calculator module.

### 4. Buffer filler (or Cluster Finder) block

![Fig. 3 Schematic of the Buffer filler block](image)

**Description:**

In the buffer filler part of the design, there are three different tasks to be combined to have a smooth flow of the data from the input decoder (or strip reader) to the output calculator block. These are shown in Figure 3. See Appendix A for the state flow diagram.
The inputs and outputs from the buffer filler block are

Inputs:
- Busy
- Data_in
- Fifo_empty

Outputs:
- Data buffers D1, D2, D3, D4, D5
- Data_valid
- Read
- Data_type
- Pointer

The blocks

- Interfacing with the decoder (strip reader) to obtain valid 23-bit data words

This data is given by the decoder in the form of 23 bit data word stored in the decoder’s output FIFO. Each data word consists of the datatype, the new data signal, the end of FIFO (EOF) signal, the address of the chip and the channel from which data is obtained and finally the data value. This data is split up into its individual components in a VHDL process block.

- Filling the data buffers

A second VHDL process block is used to analyze the 23 bit data word. This process block implements the algorithm outlined in [1]. Actual calculation of the centroid is done in the “calculator” block. Please refer to Appendix A for a flowchart of this part of the design.

- Interfacing with the calculator block

This process block interfaces with the calculation block. When all the data buffers are full or when there is a change of datatype or EOF, a signal is sent to calculate the current centroid. A simple handshaking scheme is used to transfer data from the buffer filler to the calculator. When a “busy” signal from the calculator block has been received, the buffer filler prepares the output buffers for the next centroid calculation. This allows the “buffer filler” and “calculator” to run in parallel. Only when the calculator is not “busy” does the buffer filler transfer data to the calculator. Otherwise, it moves into a wait state for the calculator to become free again.

5. Calculator Block

An internal block diagram of the calculator block is given in Figure 4. This block accepts the data from the buffer filler and performs the arithmetic calculation of the centroid. It is compose of several modules including the controller, counter, register, adder, divider, and finder.
### Controller module

The purpose of the controller module is to turn on the register and counter blocks on when needed. The inputs of this module are clock, reset, data_valid and done. Data-valid comes from the buffer filler block and indicates that the inputs are valid. Done comes from the counter module and indicates that the calculation is complete and new input data can be read. The outputs of this module are register_enable, counter_enable and busy. Register_enable turns the register module on/off; counter_enable turns the counter module on/off and busy tells the previous block when the calculations are complete.

![Block Diagram of the Calculator Partition](image)

**Fig. 3: Block Diagram of the Calculator Partition.**
complete so new input data can be received. A state diagram in Appendix A describes the processes that occur within this module.

- **Counter module**

The purpose of the counter module is to send a signal to the controller module indicating when the calculation is complete so that new data can be taken from the previous block. It accomplishes this task by counting for a predetermined amount of clock cycles and then sending a signal to the controller module when the count is complete. The inputs of this module are `clock`, `reset`, and `enable`. `Enable` is a signal from the controller module that turns the counter on and off. The outputs of this module are `done`. `Done` tells the controller module when the count is complete. The VHDL code is located in Appendix B.

- **Register module**

The purpose of the register module is to take data from the previous block and stores it. The inputs of this module are `D1`, `D2`, `D3`, `D4`, `D5`, `pointer`, `data-type`, `clock`, `reset`, and `enable`. `Enable` is a signal from the controller module that turns the register on and off. The outputs of this module are `D1`, `D2`, `D3`, `D4`, `D5`, `pointer`, and `data-type`.

- **Adder module**

The purpose of the adder module is to take data from register module and create the sums $D1+D2+D3+D4+D5$ and $-D1+D3+2D4+3D5$, which are the denominator and numerator for the centroid of $D1$, $D2$, $D3$, $D4$ and $D5$, respectively. The centroid of a system is defined as the center of mass or the point in the diagram below where the seesaw’s support could be moved to attain equilibrium. In Figure 5 below, if $D2$ is viewed as the origin and $D1$, $D3$, $D4$ and $D5$ as point masses, the centroid of the system is $-D1+D3+2D4+3D5$.

$$\overline{x} = \frac{\sum_{i=1}^{n} m_i x_i}{\sum_{i=1}^{n} m_i}$$

The moment of the system is the mass times the distance from the origin: $D1$ (-1)+$D3$ (1)+$D4$ (2)+$D5$ (3). Thus, the moment of the system is $-D1+D3+2D4+3D5$.

**Fig. 5: Realization of centroid calculation**
The inputs of this module are \( D1, D2, D3, D4 \) and \( D5 \). The outputs of this module are \( \text{num} \) and \( \text{den} \) which are the numerator and denominator, respectively. The sums are computed by using the algorithm (Refer to Appendix A).

- **Divider**

The purpose of the divider module is to take \( \text{num} \) and \( \text{den} \) values from the adder module and calculate the 4-bit quotient of \( \text{num} \) divided by \( \text{den} \). The inputs of this module are \( \text{num} \) and \( \text{den} \). The output of this module is \( \text{quotient} \). One whole bit and 3 decimal bits represent the 4-bit quotient. The highest number that can be represented is 1111 (binary), \( 2^0 + 2^{-1} + 2^{-2} + 2^{-3} \), which is equal to 1.875. The division is computed by using the algorithm (Refer Appendix A).

- **Finder**

The purpose of the finder module is to use the algorithm below to find the 13-bit centroid strip number with quarter strip resolution. The strip number is a function of the pointer and the quotient of the centroid calculation. The pointer is used to indicate the address corresponding to the data in buffer D4, this pointer – 2 (decimal) are the address of D2. The first 3 bits of the quotient are added on to (pointer – 2 decimal) * 4 decimal. The last bit of the quotient is used for rounding. For the algorithm refer Appendix A.

### 6. Altera FPLD Implementation of the Baseline Design

The design was automatically fitted into an EPF10K50EQC208-1 using the default synthesis options of the version 9.4 of the MAXPLUS-II CAD software system. Default resource allocations as reported by the synthesis package are listed in Table I. The “EABs used” given in the Table are primarily employed to implement the internal FIFOs and the RAM which holds channel correction data.

<table>
<thead>
<tr>
<th>Item</th>
<th>Reported Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total logic cells used:</td>
<td>1433/2880 (49%)</td>
</tr>
<tr>
<td>Total embedded cells used:</td>
<td>40/160 (25%)</td>
</tr>
<tr>
<td>Total EABs used:</td>
<td>8/10 (80%)</td>
</tr>
<tr>
<td>Average fan-in:</td>
<td>3.24/4 (81%)</td>
</tr>
<tr>
<td>Total flipflops required:</td>
<td>481</td>
</tr>
</tbody>
</table>
Altera GDF files for each partition are given in Appendix C. A functional simulation of the design using the sample test data given in Table II is shown in Figure 6 below. This functional simulation assumes:

- 1 MHz Clock,
- No bad channels,
- Sequential channels,
- No offset correction per channel is needed.

### Table II
**Sample Input Data with Expected Results**

Note: Pointer MSB shown below does not include a 780H offset

<table>
<thead>
<tr>
<th>Channel</th>
<th>Data</th>
<th>D1</th>
<th>D2</th>
<th>D3</th>
<th>D4</th>
<th>D5</th>
<th>Centroid</th>
<th>Pointer</th>
<th>Pointer_MSB</th>
<th>Pointer_LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>12</td>
<td>0</td>
<td>10</td>
<td>12</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>14</td>
<td>10</td>
<td>12</td>
<td>14</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>12</td>
<td>10</td>
<td>12</td>
<td>14</td>
<td>12</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>10</td>
<td>12</td>
<td>14</td>
<td>12</td>
<td>10</td>
<td>1.00</td>
<td>3</td>
<td>2</td>
<td>&quot;00&quot;</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>10</td>
<td>12</td>
<td>14</td>
<td>10</td>
<td>8</td>
<td>0.89</td>
<td>4</td>
<td>3</td>
<td>&quot;11&quot;</td>
</tr>
<tr>
<td>6</td>
<td>10</td>
<td>10</td>
<td>8</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>7</td>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>10</td>
<td>8</td>
<td>10</td>
<td>2</td>
<td>0</td>
<td>7</td>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>10</td>
<td>8</td>
<td>10</td>
<td>2</td>
<td>2</td>
<td>0.31</td>
<td>7</td>
<td>5</td>
<td>&quot;01&quot;</td>
</tr>
<tr>
<td>9</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0A</td>
<td>0A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0A</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0B</td>
<td>0B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0B</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0C</td>
<td>0C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0C</td>
<td>6</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0D</td>
<td>0D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0D</td>
<td>6</td>
<td>4</td>
<td>6</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0E</td>
<td>0E</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0E</td>
<td>4</td>
<td>4</td>
<td>6</td>
<td>6</td>
<td>4</td>
<td>0</td>
<td>0E</td>
<td>0E</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0F</td>
<td>6</td>
<td>6</td>
<td>4</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>8</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>11</td>
<td>11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>8</td>
<td>6</td>
<td>8</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>12</td>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>10</td>
<td>8</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>13</td>
<td>13</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>12</td>
<td>8</td>
<td>10</td>
<td>12</td>
<td>0</td>
<td>0</td>
<td>14</td>
<td>14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>14</td>
<td>10</td>
<td>12</td>
<td>14</td>
<td>0</td>
<td>0</td>
<td>15</td>
<td>15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>12</td>
<td>10</td>
<td>12</td>
<td>14</td>
<td>12</td>
<td>0</td>
<td>15</td>
<td>15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>10</td>
<td>10</td>
<td>12</td>
<td>14</td>
<td>12</td>
<td>10</td>
<td>1.00</td>
<td>15</td>
<td>14</td>
<td>&quot;00&quot;</td>
</tr>
<tr>
<td>17</td>
<td>4</td>
<td>10</td>
<td>12</td>
<td>14</td>
<td>10</td>
<td>4</td>
<td>0.72</td>
<td>15</td>
<td>14</td>
<td>&quot;11&quot;</td>
</tr>
<tr>
<td>C0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The operation of the overall design can be examined from this simulated output. Initially, `output_fifo_empty` is high until the FIFO is loaded with valid data words by the Decoder block. The Buffer Filler block is waiting for this line to go low before requesting data from the FIFO. These requests are shown as pulses on the `output_fifo_rdreq` line. Since the buffer is initially loaded with zeros (due to the system wide reset), the design immediately requests that a centroid be calculated. This request is seen in the simulation as `data_valid` going low closely followed by `busy` going low. After `busy` goes low, the Buffer Filler begins to read additional data out of the output FIFO performing the algorithm as outlined in [1]. Once the Buffer Filler determines that a centroid needs to be calculated again, it brings `data_valid` low again requesting the Calculator to compute a centroid. In this case, we have valid data and it can be seen that the output vectors `opointer_msb` and `opointer_lsb` are set to 782H and 00H respectively. This matches the expected results given in Table II above. Recall, while the Calculator is computing the centroid, the Buffer Filler is configuring the data buffers for the next centroid calculation. However, the Buffer Filler must wait until the Calculator raises `busy` high before it can bring `data_valid` low again. From the simulation, we can see at approximately 110us, the Buffer Filler has read in a new data word and wants to calculate a new centroid. However, it must enter a wait condition until the Calculator module raises its `busy` line high. After this occurs at 120uS, the Buffer Filler immediately brings `data_valid` low again, initiating another centroid calculation. In this case, the new centroid is calculated to be 782H and 11H as expected. This process repeats until the output FIFO becomes empty.
The static timing analyzer predicts a maximum switching frequency of 32MHz for the design. It reports this performance is limited by the state vector used within the Buffer Filler module. It also reports that the LPM_ROM module used to store channel information would limit the performance of the design to 35MHz.

7. Summary

A baseline design for the Centroid Finder algorithm has been completed. This design was completed entirely in VHDL and simulated using the Altera MAXPLUS-II software package. The logic design fit easily within an Altera EPF10K50 using only 50% of the total logic cells. However, 80% of the 24K of embedded memory was also needed in the design implementation. This includes 18K for the channel correction parameters and 4.9K for the decoder FIFO’s. The functional (1 MHz) simulation results given an ideal input data set showed the design operated as expected. A static timing analysis predicted a maximum clock frequency of 32 MHz with the switching frequency being limited by the Buffer Filler module.

8. References

9. Appendix A
Flow Diagram For the Decoder Block

Chart 1

A

Initialize all register & default conditions

Reset

FIFO Empty

Yes

B

No

Input = Seq ID?

Yes

Set -> State to HDI & New data flag
Clear -> Stereo Flag
Save -> Sequencer ID Value

C
Chart 2

1. **Input = H D I ID?**
   - No: Go to B
   - Yes: Go to State is set to Chip ID & H D I ID is saved

2. **Most Significant Bit = 1?**
   - No: Go to B
   - Yes: Go to Save CHIP ID & set State to Zero. Check for firs t stereo chip.

3. **First Stereo Chip?**
   - No: Go to D
   - Yes: Set Stereo and New Data Flag

---

**Legend**
- C
- D
- B
- No
- Yes
Chart 3

D

ZERO

Input is all zeros?

A

CHANNEL

Set State to Channel

E

Input = C0?

Yes

Write flag is set & the 23 bit-word is sent to the fifo. State set to Seq.

B

No

MSB of input = 1

Yes

Save Chip ID and set state to ZERO

No

First Stereo Chip

G

Channel ok?

No

First Stereo Chip

Yes

Set Stereo and New Data Flag

D
Chart 4

Write enable goes high, and the data is sent to the FIFO in 23-bit long format. New Data Flag is cleared and State is set to Channel.

State is set back to Channel.
Flow Diagram for the Buffer Filler

Chart 1

1. **SRESET**
   - Initialize all registers
   - Reset

2. **SFIRSTREAD**
   - D1, D2, D4, D5 ← 0
   - D3 ← Datain
   - Dtype ← Type
   - pointer ← address + 1
   - Store the Previous State
   - Branch

3. **SLOAD**
   - Load data into Type, New data, Eof, Datain, and Address
   - Branch

4. **SCONTINUE**
   - Branch

5. **SINIT_READ**
   - RFID Empty?
   - No
   - Store the Previous State
   - Read RFID
   - Branch

6. **SFIFOREAD**
   - RFID Empty or EOF?
   - No
   - Store the Previous State
   - Branch

7. **SREAD**
   - RFID Empty or EOF?
   - Yes
   - Store the Previous State
   - Branch
   - No
   - Store the Previous State
   - Branch

8. **SFCONTINUE**
   - Branch

Diagram notes:
- Arrows indicate the flow of execution.
- D1, D2, D4, D5, D3, Datain, Type, New data, Eof, and Address are variables used in the flowchart.
- The diagram illustrates the process of initializing registers, checking RFID status, and loading data into memory.
Chart 2

SCHEDULE_TYPE

Type = Datatype?  OR  No New Data

Yes

Hold_address ← address_pointer
Mux_address ← hold_address(2 downto 0)

Mux_address

0 1 2 3 4

M2  M3  M4

Store the Previous State

S
Chart 3

M0

Is Datain > D3 ?

No

D1 ← D1
D2 ← D2
D3 ← D3
D4 ← Datain
D5 ← 0
Pointer ← Pointer

Yes

D1 ← D2
D2 ← D3
D3 ← Datain
D4 ← 0
D5 ← 0
Pointer ← Address + 1

C

M1

Is Datain > D3 ?

No

D1 ← D1
D2 ← D2
D3 ← D3
D4 ← D4
D5 ← Datain
Pointer ← Pointer

Yes

D1 ← D3
D2 ← D4
D3 ← Datain
D4 ← 0
D5 ← 0
Pointer ← Address + 1

C
Chart 4

**SCALCULATE**

1. Calculate ← 0
   Read ← 1
   Dtype ← Datatype
   Point ← Pointer

2. **Valid = 1?**
   - **Yes**
   - **No**
     - **SWAIT**
       - Continue = 0?
         - **Yes**
         - **SCALCULATE**
           - Calculate ← 1

3. Depending on the state before the Calculation, the flow of the program will be decided.

- Read
- check_type
- smux2
- smux3
- smux4

4. **D1, D2, D4, D5 ← 0**
   D3 ← Data
   Dtype ← Type
   Pointer ← Address + 1
Chart 6

D₁ <--- 0
D₂ <--- 0
D₃ <--- Data_in
D₄ <--- 0
D₅ <--- 0
Pointer <--- Address + 1
Chart 7

Process block for the interface of the control block with calculator block

**CALCSRESET**
- Initialize all registers
- Reset

**CALCSINIT**
- Continue ----> 0
- Data_valid ----> 1
- Calculate ?
  - No
  - Yes

**CALCS1**
- Data_valid ----> 1

**CALCS2**
- Continue <--- 0
- Busy ?
  - Yes
  - No
- Data_valid ----> 0

**CALCS3**
- Busy ?
  - No
  - Yes
- Continue <--- 1

Note: The continue signal is internal to the block, this signal is represented active high
Chart 8

CALS4

Busy?

Yes

No

Data_valid <= 1

I
State Diagram for the Control Module

State S0:
- Data_valid = 1
- R_enable = 1
- C_enable = 1
- Busy = 1
- Count > x
- Wait x clock cycles
- Turn busy signal on
- Enable registers

State S1:
- Data_valid = 0
- R_enable = 0
- C_enable = 0
- Busy = 0
- Count < x
- Wait x clock cycles
- Turn busy signal off
- Disable registers

State S2:
- Data_valid = 1
- R_enable = 1
- C_enable = 1
- Busy = 1
- Count < x
- Wait x clock cycles
- Turn busy signal on
- Enable registers

State S3:
- Data_valid = 0
- R_enable = 0
- C_enable = 0
- Busy = 0
- Count > x
- Wait x clock cycles
- Turn busy signal off
- Disable registers

State S4:
- Data_valid = 1
- R_enable = 1
- C_enable = 1
- Busy = 1
- Count < x
- Wait x clock cycles
- Turn busy signal on
- Enable registers

State S5:
- Data_valid = 0
- R_enable = 0
- C_enable = 0
- Busy = 0
- Count > x
- Wait x clock cycles
- Turn busy signal off
- Disable registers

State S6:
- Data_valid = 1
- R_enable = 1
- C_enable = 1
- Busy = 1
- Count < x
- Turn counter on
- Wait for counter to finish
- Done = 1

State S7:
- Data_valid = 0
- R_enable = 0
- C_enable = 0
- Busy = 0
- Count > x
- Turn counter off
- Wait for counter to finish
- Done = 0

State S8:
- Data_valid = 1
- R_enable = 1
- C_enable = 1
- Busy = 1
- Count < x
- Turn counter off
- Wait for counter to finish
- Done = 1

State S9:
- Data_valid = 0
- R_enable = 0
- C_enable = 0
- Busy = 0
- Count > x
- Turn counter on
- Wait for counter to finish
- Done = 0

State S10:
- Data_valid = 1
- R_enable = 1
- C_enable = 1
- Busy = 1
- Count < x
- Turn counter off
- Wait for counter to finish
- Done = 1

State S11:
- Data_valid = 0
- R_enable = 0
- C_enable = 0
- Busy = 0
- Count > x
- Turn counter on
- Wait for counter to finish
- Done = 0

State S11:
- Data_valid = 1
- R_enable = 1
- C_enable = 1
- Busy = 1
- Count < x
- Turn counter off
- Wait for counter to finish
- Done = 1

State S11:
- Data_valid = 0
- R_enable = 0
- C_enable = 0
- Busy = 0
- Count > x
- Turn counter on
- Wait for counter to finish
- Done = 0
Algorithm for Centroid Calculation

\[ \begin{align*}
D_1 + D_2 & \rightarrow 8\text{-bit adder} \\
D_3 & \rightarrow 8\text{-bit adder} \\
D_4 & \rightarrow 8\text{-bit adder} \\
D_5 & \rightarrow 8\text{-bit adder} \\
\end{align*} \]

\[ \begin{align*}
D_1 + D_2 & = D_1 + D_2 \\
D_3 & = D_9 \\
D_4 & = D_9 \\
D_5 & = D_9 \\
\end{align*} \]

\[ \begin{align*}
D_1 + D_2 & \rightarrow 8\text{-bit adder} \\
D_3 & \rightarrow 8\text{-bit adder} \\
D_4 & \rightarrow 8\text{-bit adder} \\
D_5 & \rightarrow 8\text{-bit adder} \\
\end{align*} \]

\[ \begin{align*}
D_1 + D_2 & = D_1 + D_2 \\
D_3 & = D_9 \\
D_4 & = D_9 \\
D_5 & = D_9 \\
\end{align*} \]

\[ \begin{align*}
D_1 + D_2 & \rightarrow 8\text{-bit adder} \\
D_3 & \rightarrow 8\text{-bit adder} \\
D_4 & \rightarrow 8\text{-bit adder} \\
D_5 & \rightarrow 8\text{-bit adder} \\
\end{align*} \]

\[ \begin{align*}
D_1 + D_2 & = D_1 + D_2 \\
D_3 & = D_9 \\
D_4 & = D_9 \\
D_5 & = D_9 \\
\end{align*} \]

\[ \begin{align*}
D_1 + D_2 & \rightarrow 8\text{-bit adder} \\
D_3 & \rightarrow 8\text{-bit adder} \\
D_4 & \rightarrow 8\text{-bit adder} \\
D_5 & \rightarrow 8\text{-bit adder} \\
\end{align*} \]

\[ \begin{align*}
D_1 + D_2 & = D_1 + D_2 \\
D_3 & = D_9 \\
D_4 & = D_9 \\
D_5 & = D_9 \\
\end{align*} \]

\[ \begin{align*}
D_1 + D_2 & \rightarrow 8\text{-bit adder} \\
D_3 & \rightarrow 8\text{-bit adder} \\
D_4 & \rightarrow 8\text{-bit adder} \\
D_5 & \rightarrow 8\text{-bit adder} \\
\end{align*} \]

\[ \begin{align*}
D_1 + D_2 & = D_1 + D_2 \\
D_3 & = D_9 \\
D_4 & = D_9 \\
D_5 & = D_9 \\
\end{align*} \]

\[ \begin{align*}
D_1 + D_2 & \rightarrow 8\text{-bit adder} \\
D_3 & \rightarrow 8\text{-bit adder} \\
D_4 & \rightarrow 8\text{-bit adder} \\
D_5 & \rightarrow 8\text{-bit adder} \\
\end{align*} \]

\[ \begin{align*}
D_1 + D_2 & = D_1 + D_2 \\
D_3 & = D_9 \\
D_4 & = D_9 \\
D_5 & = D_9 \\
\end{align*} \]

\[ \begin{align*}
D_1 + D_2 & \rightarrow 8\text{-bit adder} \\
D_3 & \rightarrow 8\text{-bit adder} \\
D_4 & \rightarrow 8\text{-bit adder} \\
D_5 & \rightarrow 8\text{-bit adder} \\
\end{align*} \]

\[ \begin{align*}
D_1 + D_2 & = D_1 + D_2 \\
D_3 & = D_9 \\
D_4 & = D_9 \\
D_5 & = D_9 \\
\end{align*} \]
Algorithm for Divider module

Complement of Denominator.000

Q3\[ CO \quad 14\text{-bit adder} \quad A \quad B \]

Q2\[ CO \quad 14\text{-bit adder} \quad A \quad B \]

Q1\[ CO \quad 14\text{-bit adder} \quad A \quad B \]

Q0\[ CO \quad 11\text{-bit adder} \quad A \quad B \]

1 & cden[13..1]

11 & cden[13..2]

111 & cden[13..3]

Numerator.00

= quotient
Algorithm for Finder module

Pointer 7FD(hex) + 1(bin)

11-bit adder

11

X4

13-bit adder

Q0
Q1
Q2
Q3

13-bit pointer
10. Appendix B
The code for the decoder module

-- Roberto Brown
-- FAMU-FSU COE
-- Version 1.0
-- 1-25-00

library altera;
use altera.maxplus2.all;
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

-----------------------------------------------------------------------------
--    This block defines the inputs and outputs of the counter              
-----------------------------------------------------------------------------
entity c_decoder_perry is
port ( reset, clock, input_fifo_empty : in  std_logic;
data_in   : in unsigned (8 downto 0);   -- Data offset and data_good are
stream:  in unsigned (7 downto 0);
oread,owrite   : out std_logic;
ostate    :out unsigned(3 downto 0);
rom_address    : out unsigned(10 downto 0);
output   : out unsigned (22 downto 0)
);
end entity c_decoder_perry;

----------------------------Architecture Beginning---------------------------

architecture logic_design of c_decoder_perry is

---------------------Signal and constant Declaration--------------------------
signal  neof,peof: std_logic;
signal  n_new_data_flag,p_new_data_flag: std_logic;
signal  nseq_id,pseq_id,nhdi_id,phdi_id: unsigned(7 downto 0);
signal  noutput,poutput    : unsigned (22 downto 0);
signal  nchip_id,pchip_id  : unsigned (3 downto 0);
signal  nchan_id,pchan_id  : unsigned (6 downto 0) ;
signal  ndata_type, pdata_type: unsigned(1 downto 0);
signal  nstate, pstate   : unsigned(3 downto 0);
signal  nrom_address, prom_address: unsigned(10 downto 0);
constant seq_id  : unsigned (7 downto 0)  := "10101010";  -- Assume sequence
    ID of AA
constant hdi_id  : unsigned (7 downto 0)  := "01110111";  -- Assume Head ID of
77
constant first_stereo_id :unsigned(7 downto 0) := "11111111" ; -- Assume first
stereo chip id of FF
constant zero8  : unsigned (7 downto 0)  := "00000000";
constant zero7  : unsigned (6 downto 0)  := "00000000";
constant zero4  : unsigned (3 downto 0)  := "0000";
constant C0     : unsigned (7 downto 0)  := "11000000";
constant zero2  : unsigned (1 downto 0)  := "00";
constant initial_state   : unsigned (3 downto 0)  := "0000";
constant seq_state   : unsigned (3 downto 0)  := "0001";
constant hdi_state   : unsigned (3 downto 0)  := "0010";-- State constants
constant chip_state : unsigned (3 downto 0) := "0011";
constant zero_state : unsigned (3 downto 0) := "0100";
constant chan_state : unsigned (3 downto 0) := "0101";
constant value_state : unsigned (3 downto 0) := "0111";
constant read_state : unsigned (3 downto 0) := "1000";
constant write_state : unsigned (3 downto 0) := "1001";
constant zero23 : unsigned (22 downto 0) := "00000000000000000000000000";
constant high : std_logic := '1';
constant low : std_logic := '0';
constant default_data_type : unsigned(1 downto 0) := "01";
constant stereo_data_type : unsigned(1 downto 0) := "11";

-- THESE CONSTANT DEFINE THE STATE OF THE DECODER --
type mystates is (seq,hdi,chip,zero,chan,value,write_fifo,read_fifo);
signal ns,ps,nrs,prs : mystates;

-----------------------------Block Begins-----------------

begin

process(pstate,ps,prs,poutput,pchip_id,pchan_id,p_new_data_flag,pdata_type,peof
,vseq_id,phdi_id,data_in)
--- VARIABLE DECLARATION ---
variable voutput : unsigned (22 downto 0);
variable vnew_data_flag,veof : std_logic;
variable vchip_id : unsigned(3 downto 0);
variable vchan_id : unsigned(6 downto 0);
variable vseq_id,vhdi_id,vdata_offset : unsigned(7 downto 0);
variable vdata_type : unsigned(1 downto 0);
variable next_state,return_state : mystates;
variable vstate : unsigned(3 downto 0);

begin

--
-- Default variable values
--
vstate := pstate;
vnew_data_flag := p_new_data_flag;
next_state := ps;
voutput := poutput;
veof := peof;
vdata_type := pdata_type;
vchip_id := pchip_id;
vchan_id := pchan_id;
vseq_id := pseq_id;
vhdi_id := phdi_id;
return_state := prs;
vdata_offset := data_in(7 downto 0);
nrom_address <= pchip_id&pchan_id;

---------------------
----CASE STATEMENT ----
---------------------
-- The following case statment will look at the incoming stream of data and
will proceed to
-- compare and check for the validity of the same. For each case different
actions will occur.
-- Note1: Every state requires to have a read fifo before it can proceed to check the information.
-- Note2: After the channel state and 11-bit word composed of the chip and channel id are sent to
-- to an lpm rom, which will send a 9-bit word back into the decoder. From
-- this 9 bit word we
-- will look at the MSB and if it is "0" then we know that the channel is ok.
-- Note3: Only the channel and Value state will have the command to enable
writing to the FIFO.

CASE ps IS
  -- Sequencer Stage --
  WHEN seq => --Check for Sequencer
    vstate := seq_state;
    -- variable used to check state status of program
    next_state := read_fifo;
    if(stream = seq_id) then --when stream equals a preset value
      return_state := hdi;   -- we proceed to save the sequencer
      vseq_id := seq_id;   -- and proceed to HDI state.
      vdata_type := default_data_type;
    vnew_data_flag := high;
    else
      return_state := seq;
    -- Is value does not match, then we stay in seq
    end if;
  -- HDI Stage --
  WHEN hdi => --Check for hdi
    vstate := hdi_state;
    -- variable used to check state status of program
    next_state := read_fifo;
    if(stream = hdi_id) then
      --when stream equals a preset HDI value,
      return_state := chip;  -- it is saved and state is set to chip.
      vhdi_id  := hdi_id;
    else
      return_state := seq;
    -- If does not match HDI value return to
    end if;      -- sequencer state
  -- CHIP Stage
  WHEN chip =>
    vstate := chip_state;
    -- variable used to check state status of program
    next_state := read_fifo;
    if (stream(7) = high) then
      vchip_id := stream(3 downto 0);
      return_state := zero;
      if(stream = first_stereo_id) then
        vnew_data_flag := '1';
        vdata_type := stereo_data_type;
      end if;
    else
      return_state := seq;
    end if;
  -- ZERO Stage --
  WHEN zero =>
    vstate := zero_state;
    -- variable used to check state status of program
    next_state := read_fifo;
    if (stream = zero7) then
      return_state := chan;
else
  return_state := seq;
end if;

-- CHANNEL Stage --
WHEN chan => -- The channel ID is read from stream
  vstate := chan_state;
  -- variable used to check state status of program
  next_state := read_fifo;
  return_state := seq;
  veof := '0';
  if (stream = C0) then
    veof := '1';
    voutput(19) := '1'; -- set peof to 1 when eof
    next_state := write_fifo;
    return_state := seq;
  elsif (stream(7) = '1') then
    -- This means this is a new chip ID
    next_state := chip; -- Check for a new stereo ch
  else
    next_state := read_fifo;
    vchan_id := stream(6 downto 0);
    return_state := value;
end if;

-- VALUE Stage --
WHEN value =>
  vstate := value_state;
  -- variable used to check state status of program
  next_state := read_fifo;
  return_state := chan;
  if(data_in(8) = '0') then
    -- This information is obtained from a ROM macro
    voutput(22 downto 21) := pdata_type;
    voutput(20) := p_new_data_flag;
    voutput(19) := peof;
    voutput(18 downto 11) := stream-vdata_offset;
      -- subtract offset from data
    voutput(10 downto 7) := pchip_id;
    voutput(6 downto 0) := pchan_id;
    vnew_data_flag := low;
    next_state := write_fifo;
  else
    next_state := chan;
    -- this channel is bad read the next one
  end if;

-- write_fifo Stage --
WHEN write_fifo =>
  vstate := write_state;
  -- variable used to check state status of program
  next_state := return_state;

when read_fifo =>
  vstate := read_state;
  -- variable used to check state status of program
  next_state := return_state;

-- OTHERS --
WHEN OTHERS =>
  next_state := seq;
  return_state := seq;

END CASE;
-- set the next state of signal to their respective variable signals
nstate <= vstate;
n_new_data_flag <= vnew_data_flag;
nrs <= return_state;
nstate <= next_state;
noutput <= voutput;
n.eof <= veof;
nrs <= return_state;
n_new_data_flag <= vnew_data_flag;
noutput <= voutput;
n.eof <= veof;
nstate <= vstate;
n_new_data_flag <= vnew_data_flag;
nrs <= return_state;
nstate <= next_state;
noutput <= voutput;
n.eof <= veof;
end process;

-- The following two-process blocks work in concurrently with the above block.
One
-- block is regarded as the read FIFO process block and the other is regarded
as
-- the write FIFO process block

-- Read fifo block
process(ps,input_fifo_empty)
  -- This process is used to read the input fifo
variable vread: std_logic;
begin
  vread := '0';
  if(input_fifo_empty = '1') then
    vread := '0';
    if(input_fifo_empty = '1') then
      vread := '0';
  end if;
end process;

-- Write and read fifo process
  case ps is
    when read_fifo|write_fifo=>
      vread := '1';
      vread := '0';
    when others =>
      vread := '0';
  end case;
end if;
end if;
end process;

-- Write and read fifo process
  case ps is
    when read_fifo|write_fifo=>
      vread := '1';
      vread := '0';
    when others =>
      vread := '0';
  end case;
end process;

-- This process is used to write to the output fifo
variable vwrite: std_logic;
begin
  vwrite := '0';
case ps is
  when write_fifo=> -- to write to the output fifo the 23-bit word
    vwrite := '1';
  when others =>
    vwrite := '0';
end case;
end if;
end if;
end process;

-------------------------------------------------------------------------------
begin
if (reset = '0') then --Sets all outputs and counters to zero
  pstate <= initial_state;
pseq_id <= zero8;
prom_address <= zero4&zero7;
elsif (clock'event and clock = '1') then
  pstate <= nstate;
pseq_id <= nseq_id;
prom_address <= nrom_address;
end if;
end process reg; --End of register block

--End of architecture block
The code for the buffer filler module

--- File : A_filler_calc.vhd
--- Description : VHDL code to implement Filling up of buffers for Centriod Finder
--- Created on : 01/14/2000 ( Shweta Lolage)
--- Modified on : 01/21/2000 ( Shweta Lolage)
--- Modified on : 01/22/2000 ( Shweta Lolage)
--- Modified on : 01/26/2000 ( Shweta Lolage)

-- Normal initialization stuff
library altera;
use altera.maxplus2.all;
library ieee;
use ieee.std_logic_arith.all;
use ieee.std_logic_1164.all;

-- Entity Declaration
entity a_filler_calc is
  port(data_in : in unsigned (22 downto 0); --data from the fifo
       clock,reset,fifo_empty,busy : in std_logic;
       read,data_valid : out std_logic; --output signals
       d1,d2,d3,d4,d5 : out unsigned (7 downto 0); --buffer outputs
       datatype : out std_logic_vector (1 downto 0); --datatype of centriod to calculator
       pointer : out unsigned (10 downto 0) --pointer for the calculator);
end entity a_filler_calc;

-- Architecture Body

architecture behavior of a_filler_calc is

--define major states for the main process and the calculation interface
type states is (sreset,sinit_read,sfiforead,sload,sfirstread,scontinue,sread,
scheck_type,smux0,smux1,smux2,smux3,smux4,scalculate,swait,scalccont);
type calcstates is (calcsreset,calcsinit,calcs1,calcs2,calcs3,calcs4);

--define local states for the mux
constant s0 : unsigned (2 downto 0) :="000";
constant s1 : unsigned (2 downto 0) :="001";
constant s2 : unsigned (2 downto 0) :="010";
constant s3 : unsigned (2 downto 0) :="011";
constant s4 : unsigned (2 downto 0) :="100";

--define signals for each part of the data from the decoder
signal ndatain,pdatain : unsigned (7 downto 0); --the data from the channel
signal ntype,ptype : unsigned (1 downto 0); --the datatype of channel data
signal nnew_data,pnew_data,neof_flag,peof_flag : std_logic; --some signals
signal naddress, paddress : unsigned (10 downto 0);
--- the combined chip and the channel address

signal ndatatype,pdatatype : unsigned (1 downto 0);

--- define fifo_types (Are needed to be defined)
constant fifo : unsigned (1 downto 0) := "00";

--- define fifo_states (Are needed to be defined)
signal ps,ns : states;
signal ps1,ns1 : calcstates;
signal pstate,nstate : states;
signal pstatel,nstatel : states;

--- signals used for data registers
signal nd1,nd2,nd3,nd4,nd5 : unsigned (7 downto 0);
signal pd1,pd2,pd3,pd4,pd5 : unsigned (7 downto 0);

--- these constants are used with the data registers
constant zero_reg : unsigned (7 downto 0) := "00000000";

--- signals used for output fifo buffer
signal nread,nbusy : std_logic;
signal pread,pbusy : std_logic;

--- signals used for the inputs to the block
signal nempty,nvalid : std_logic;
signal pempty,pvalid : std_logic;

--- signal used for the internal signals
signal ncalc,ncont : std_logic;
signal pcalc,pcont : std_logic;

--- counter signals and constants
signal ppointer,npointer : unsigned (10 downto 0);

--- signal used for data compare of d3 and d5 to din
signal pdata5,ndata5,pdata3,ndata3 : std_logic;

--- define local states
constant l0 : std_logic := '0';
constant l1 : std_logic := '1';

--- internal signals
signal ndtype,pdtype : std_logic_vector (1 downto 0);
signal npoint,ppoint : unsigned (10 downto 0);

begin

--- to register the input signals
nempty <= fifo_empty; -- from the output FIFO of the decoder
nbusy <= not (busy); -- from the calculator block

--- process to take in the data from the fifo
process (data_in,ptype,pnew_data,peof_flag,paddress,pdatain,pread)
begin

if (pread = '1') then
    ntype <= data_in (22 downto 21);
    nnew_data <= data_in (20);
    neof_flag <= data_in (19);
    ndatain <= data_in (18 downto 11);
    naddress <= data_in (10 downto 0);
else
    ntype <= ptype;
    nnew_data <= pnew_data;
    neof_flag <= peof_flag;
    ndatain <= pdatain;
    naddress <= paddress;
end if;
end process;

--main control process for the buffer filler

process (ps,pstate,pstatel,pdatain,paddress,pdatatype,pnew_data,peof_flag,
pd1,pd2,pd3,pd4,pd5,pcalc,pread,ppointer,pempty,pvalid,
pcont,ptype,ntype,pdata3,pdata5)

-- variables for defining the mux address from the address and the pointer
variable hold_address : unsigned (10 downto 0);
variable mux_address : unsigned (2 downto 0);

begin

-- defining the default outputs
ncalc <= pcalc;
read <= pread;
npointer <= ppointer;

-- nmul <= pmul;
ndatatype <= pdatatype;

nstatel <= pstatel;
nstate <= pstate;
ns <= ps;

npoint <= ppointer;
ndtype <= pdtype;

nd1 <= pd1;
nd2 <= pd2;
nd3 <= pd3;
nd4 <= pd4;
nd5 <= pd5;

-- the control block defining the buffer filling

case ps is
    when sreset =>  -- default state
        ns <= sinit_read;
        ncalc <= '0';
nread <= '0';
ndatatype <= "00";
npointer <= "00000000000";

when sinit_read => -- to check for the fifo_empty
  if ( pempty = '1') then
    ns <= sinit_read;
  else
    ns <= sfiforead;
    nstate1 <= ps;
  end if;

when sfiforead => -- to read data from the fifo
  nread <= '1';
  ncalc <= '0';
  ns <= sload;

when sload => -- load the data from the fifo
  nread <= '0';
  ncalc <= '0';

  case pstatel is
    when sinit_read =>
      ns <= sfistread;
    when scontinue =>
      ns <= sread;
    when others =>
      ns <= sinit_read;
  end case;

when sfistread => -- when the first data is read from the fifo
  nd1 <= "00000000";
  nd2 <= "00000000";
  nd3 <= pdatain;
  nd4 <= "00000000";
  nd5 <= "00000000";

  ndatatype <= ptype;
  npointer <= paddress + 1;
  ns <= scontinue;

when scontinue => -- continue reading from the fifo for the next data
ns <= sfiforead;
nstate1 <= ps;

when sread => --check for any flags
    if (pempty = '1' or peof_flag = '1') then
        ns <= scalculate;
nstate <= ps;
    else
        ns <= scheck_type;
    end if;
end when;

when scheck_type => -- check for the datatype or for new_data
    if (ndatatype = pdatatype and pnew_data = '0') then
        hold_address := paddress - ppointer;  -- determine the
        mux_address := hold_address (2 downto 0);
        case mux_address is  -- go to the right routine for
            storing the data values
            when s0 =>
                ns <= smux0;
            when s1 =>
                ns <= smux1;
            when s2 =>
                ns <= smux2;
            when s3 =>
                ns <= smux3;
            when s4 =>
                ns <= smux4;
            when others =>
                ns <= scheck_type;
            end case;
        else
            ns <= scalculate;
nstate <= ps;
        end if;
when smux0 => -- this the second word stored
    
    ns <= scontinue;
    
    if (pdata3 = '1') then
        nd1 <= pd2;
        nd2 <= pd3;
        nd3 <= pdatain;
        nd4 <= "00000000";
        nd5 <= "00000000";
        npointer <= paddress +1;
    else
        nd1 <= pd1;
        nd2 <= pd2;
        nd3 <= pd3;
        nd4 <= pdatain;
        nd5 <= "00000000";
        npointer <= ppointer;
    end if;

when smux1 => -- this the third word stored

    ns <= scontinue;
    
    if (pdata3 = '1') then
        nd1 <= pd3;
        nd2 <= pd4;
        nd3 <= pdatain;
        nd4 <= "00000000";
        nd5 <= "00000000";
        npointer <= paddress +1;
    else
        nd1 <= pd1;
        nd2 <= pd2;
        nd3 <= pd3;
        nd4 <= pd4;
        nd5 <= pdatain;
        npointer <= ppointer;
    end if;

when smux2 => --after calculation this the fourth word to be stored

    nd1 <= pd1;
nd2 <= pd2;
nd3 <= pd3;
nd4 <= pd4;
nd5 <= pd5;
ns <= scalculate;
nstate <= ps;

when smux3 =>  -- after calculation again this the fifth word stored

    nd1 <= pd1;
    nd2 <= pd2;
    nd3 <= pd3;
    nd4 <= pd4;
    nd5 <= pd5;
    ns <= scalculate;
    nstate <= ps;

when smux4 => -- again doing calculation and then storing this word

    nd1 <= pd1;
    nd2 <= pd2;
    nd3 <= pd3;
    nd4 <= pd4;
    nd5 <= pd5;
    ns <= scalculate;
    nstate <= ps;

-- calculation routine called

when scalculate =>

    ncalc <= '1';
    nread <= '0';

    ndtype <= conv_std_logic_vector (pdatatype,2);
    npoint <= ppointer;

    if (pvalid = '1') then
        ns <= scalculate;
    else
        ns <= swait;
    end if;
when wait =>

    if (pcont = '0') then
        ns <= swait;

FAMU-FSU College of Engineering
Department of Electrical and Computer Engineering
else
  ns <= scalccont;
end if;

when scalccont =>

ncalc <= '0';

case pstate is

when sread =>
  ns <= sinit_read;
when scheck_type =>
  ns <= scontinue;
    nd1 <= "00000000";
    nd2 <= "00000000";
    nd3 <= pdatain;
    nd4 <= "00000000";
    nd5 <= "00000000";
    ndatatype <= ptype;
    npointer <= paddress + 1;
when smux2 =>
  ns <= scontinue;
    if (pdata5 = '1') then
      nd1 <= pd4;
      nd2 <= pd5;
      nd3 <= pdatain;
      nd4 <= "00000000";
      nd5 <= "00000000";
      npointer <= paddress + 1;
    else
      nd1 <= pd1;
      nd2 <= pd2;
      nd3 <= pd3;
      nd4 <= pd5;
      nd5 <= pdatain;
      npointer <= paddress - 1;
    end if;
when smux3 =>
  ns <= scontinue;
    if (pdata5 = '1') then
      nd1 <= pd5;
      nd2 <= "00000000";
      nd3 <= pd3;
      nd4 <= pd5;
      nd5 <= pdatain;
      npointer <= paddress - 1;
    end if;

FAMU-FSU College of Engineering
Department of Electrical and Computer Engineering
FAMU-FSU College of Engineering
Department of Electrical and Computer Engineering

```vhdl
nd3 <= pdatain;
nd4 <= "00000000";
nd5 <= "00000000";

npointer <= paddress +1;
else
    nd1 <= pd1;
    nd2 <= pd2;
    nd3 <= pd3;
    nd4 <= pd5;
    nd5 <= pdatain;

    npointer <= paddress - 1;
end if;
when smux4 =>
    ns <= scontinue;
    nd1 <= "00000000";
    nd2 <= "00000000";
    nd3 <= pdatain;
    nd4 <= "00000000";
    nd5 <= "00000000";

    npointer <= paddress +1;
when others =>
    ns <= sinit_read;
end case;
-- end of calculation routine
when others =>
    ns <= sinit_read;
end case;
end process;

-- process to compare the pd3 with the input data
process(pd3,pdatain)
begin
    if (pdatain >= pd3) then
        ndata3 <= '1';
    else
```
ndata3 <= '0';
end if;
end process;

-- process to compare the pd5 with the input data
process(pd5,pdatain)
begin
if (pdatain >= pd5) then
    ndata5 <= '1';
else
    ndata5 <= '0';
end if;
end process;

-- process to register the states
process(ns,nstate,nstate1,clock,reset)
begin
if(reset = '0') then
    ps <= sreset;
    pstate <= sreset;
    pstate1 <= sreset;
elsif(clock'event and clock = '1') then
    ps <= ns;
    pstate <= nstate;
    pstate1 <= nstate1;
end if;
end process;

-- process to register the input signals
process(ndatain,naddress,ntype,nnew_data,neof_flag,ncont,nempty,clock,reset)
begin
if(reset = '0') then
    pdatain <= "00000000";
    paddress <= "00000000000";
    ptype <= "00";
    pnew_data <= '0';
    peof_flag <= '0';
    pcont <= '0';
    pempty <= '0';
elsif \( \text{clock}'\text{event and clock} = '1' \) then

\[
\begin{align*}
\text{pdatain} & \leq \text{nadatain}; \\
\text{paddress} & \leq \text{naddress}; \\
\text{ptype} & \leq \text{ntype}; \\
\text{pnew_data} & \leq \text{nnew_data}; \\
\text{peof\_flag} & \leq \text{neof\_flag}; \\
\text{pcont} & \leq \text{ncont}; \\
\text{pempty} & \leq \text{nempty};
\end{align*}
\]

end if;

end process;

--process to register the output signals
process( \text{nread}, \text{ncalc}, \text{clock}, \text{reset})
begin

if (\text{reset} = '0') then

\[
\begin{align*}
\text{pread} & \leq '0'; \\
\text{pcalc} & \leq '0';
\end{align*}
\]

elsif (\text{clock}'\text{event and clock} = '1') then

\[
\begin{align*}
\text{pread} & \leq \text{nread}; \\
\text{pcalc} & \leq \text{ncalc};
\end{align*}
\]

end if;

end process;

--process to register the internal signals
process( \text{ndatatype}, \text{npointer}, \text{ndata3}, \text{ndata5}, \text{ndtype}, \text{npoint}, \text{clock}, \text{reset})
begin

if (\text{reset} = '0') then

\[
\begin{align*}
\text{pdatatype} & \leq "00"; \\
\text{ppointer} & \leq "00000000000"; \\
\text{ppoint} & \leq "00000000000"; \\
\text{pdtype} & \leq "00"; \\
\text{pdata3} & \leq '0'; \\
\text{pdata5} & \leq '0';
\end{align*}
\]

elsif (\text{clock}'\text{event and clock} = '1') then

\[
\begin{align*}
\text{pdatatype} & \leq \text{ndatatype}; \\
\text{ppointer} & \leq \text{npointer}; \\
\text{ppoint} & \leq \text{npoint}; \\
\text{pdtype} & \leq \text{ndtype}; \\
\text{pdata3} & \leq \text{ndata3}; \\
\text{pdata5} & \leq \text{ndata5};
\end{align*}
\]

end if;
end process;
--process to register the internal signals
process( nd1,nd2,nd3,nd4,nd5,clock,reset)
begin
if(reset = '0') then
    pd1 <= "00000000";
pd2 <= "00000000";
pd3 <= "00000000";
pd4 <= "00000000";
pd5 <= "00000000";
elsif(clock'event and clock = '1') then
    pd1 <= nd1;
pd2 <= nd2;
pd3 <= nd3;
pd4 <= nd4;
pd5 <= nd5;
end if;
end process;

--this process block is for the interfacing
--between the control block and the calculating block
process(pbusy,pvalid,pcont,pcalc,ps1)
begin
    --default states
    ns1 <= ps1;
nvalid <= pvalid;
    case ps1 is
    when calcsreset =>
        ns1 <= calcsinit;
    when calcsinit => --waiting for the calculation signal
        nvalid <= '0';
        ncont <= '0';
        if (pcalc = '1') then
            ns1 <= calcs1;
        else
            ns1 <= calcsinit;
        end if;
    when calcs1 =>        -- waiting for the busy(high) signal
        -- to put data_valid(low) signal
        ns1 <= calcs2;
nvalid <= '0';

when calcs2 =>
  ncont <= '0';
  if (pbusy = '1') then
    ns1 <= calcs2;
  else
    nvalid <= '1';
    ns1 <= calcs3;
  end if;

when calcs3 =>  -- waiting for the busy (low) signal
  -- to give a continue to the main
  process
    if (pbusy = '0') then
      ncont <= '0';
      ns1 <= calcs3;
    elsif (pbusy = '1') then
      ncont <= '1';
      ns1 <= calcs4;
    end if;
when calcs4 =>  -- waiting for the busy (high) signal
  -- to put the data_valid (high)
  process
    if (pbusy = '0') then
      nvalid <= '0';
      ns1 <= calcsinit;
    else
      ns1 <= calcs4;
    end if;
when others =>
  ns1 <= calcsreset;
  end case;
end process;

-- to register the signals
process (clock, reset, nvalid, nbusy)
begin
  if (reset = '0') then
    pvalid <= '0';
    pbussy <= '0';
  elsif (clock'event and clock = '1') then
pvalid <= nvalid;
pbusy <= nbusy;
end if;
end process;

-- to register the states
process(clock,reset,ns1)
begin
  if (reset = '0') then
    psl <= calcsreset;
  elsif (clock'event and clock = '1') then
    psl <= ns1;
  end if;
end process;

-- to assign the outputs
read <= pread;
data_valid <= not pvalid;
d1 <= pd1;
d2 <= pd2;
d3 <= pd3;
d4 <= pd4;
d5 <= pd5;
pointer <= ppoint;
datatype <= pdtype;
end architecture behaviour;
The code for the controller module.

--this program implements a controller which turns the register
--and counter modules on when they are needed.

library altera;
use altera.maxplus2.all;
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
entity controller is
    port(clock, reset, data_valid, done : in std_logic;
    busy, counter_enable, register_enable : out std_logic
    );
end entity controller;
architecture behavior of controller is

type mystates is (s0, s1, s2, s3, s4, s5, s6, s7, s8, s9, s10, s11);
signal ns, ps : mystates;
signal pstate, nstate : unsigned(7 downto 0);
signal psbusy, nsbusy : std_logic;
signal pscounter, nscounter : std_logic;
signal psregister, nsregister : std_logic;

--this constant indicates the wait time between indicated states
--it has a value between 0 and 255 clock cycles
constant x : unsigned(7 downto 0) := "00000001";
begin
    process(ps, pstate, psbusy, pscounter, psregister, data_valid, done) 
    --this process sets up the states
    begin
        ns <= ps;
        nsbusy <= psbusy;
        nscounter <= pscounter;
        nsregister <= psregister;
        case ps is
        --wait for data valid to go low
            when s0 =>
                if (data_valid = '0') then
                    ns <= s1;
                else
                    ns <= s0;
                end if;
            --wait for data valid to go low
            when s1 =>
                if (data_valid = '0') then
                    ns <= s2;
                else
                    ns <= s0;
                end if;

            --enable the registers
            when s2, s3, s4, s5, s6, s7, s8, s9, s10, s11 =>
            end case;
        end process;
end architecture behavior;
when s2 =>
  ns <= s3;
  nsregister <= '0';
--waits x clock cycles before proceeding
when s3 =>
  if (pstate > x) then
    ns <= s4;
  else
    ns <= s3;
  end if;
--disable the registers
when s4 =>
  ns <= s5;
  nsregister <= '1';
--turn the busy signal on
when s5 =>
  ns <= s6;
  nsbusy <= '0';
--turn the counter on
when s6 =>
  ns <= s7;
  nscounter <= '0';
--wait for the counter to get finished
when s7 =>
  if (done = '0') then
    ns <= s8;
  else
    ns <= s7;
  end if;
--turn the counter off
when s8 =>
  ns <= s9;
  nscounter <= '1';
--waits x clock cycles before proceeding
when s9 =>
  if (pstate > x) then
    ns <= s10;
  else
    ns <= s9;
  end if;
--set busy signal, calculation is complete
when s10 =>
  ns <= s11;
  nsbusy <= '1';
--waits x clock cycles before proceeding
when s11 =>
  if (pstate > x) then
    ns <= s0;
  else
    ns <= s11;
  end if;
when others =>
  ns <= s0;
end case;
end process;

--this process sets up the registers for the controller
process(clock, reset, ps)
begin
  if (reset = '0') then
    ps <= s0;
end
elsif (clock'event and clock = '1') then
    ps <= ns;
end if;
end process;

--this process sets up the counter the wait period
process(clock, reset, pstate)
variable cnt : unsigned(7 downto 0);
begin
  --nstate <= pstate;
  cnt := pstate;
  if (ps=s3 or ps=s9 or ps=s11) then
      cnt := cnt + 1;
  else
      cnt := "00000000";
  end if;
  nstate <= cnt;
end process;

--this process sets up the registers for the counter
--the register and counter enable lines and the busy signal
process(clock, reset, nstate, nsbusy, nsregister, nscounter)
begin
  if (reset = '0') then
    pstate <= "00000000";
    psbusy <= '1';
    psregister <= '1';
    pscounter <= '1';
  elsif (clock'event and clock = '1') then
    pstate <= nstate;
    psbusy <= nsbusy;
    psregister <= nsregister;
    pscounter <= nscounter;
  end if;
end process;

--assign a value to the output signals
busy <= psbusy;
counter_enable <= pscounter;
register_enable <= psregister;
end architecture behavior;
The code for the counter module.

--hire C:\My Documents\Thesis\Version 1.0\Programs\Counter.vhd

--hire Author: Kishma A. Meyers
--hire Date: January 22, 2000
--hire Created on: January 20, 2000 by Kishma A. Meyers

--this program implements a counter

library altera;
use altera.maxplus2.all;
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

--define input and output pins
entity counter is
port(clock, reset, enable : in std_logic;
        done : out std_logic);
end entity counter;

--define architecture
architecture behavior of counter is

--define states for counter
signal pstate, nstate : unsigned(7 downto 0);

--define states for the done output
signal psdone, nsdone : std_logic;

--this constant specifies how many clock cycles to wait
constant x : unsigned(7 downto 0) := "00000100";
begin

--this process sets up the states for the counter
process(pstate, enable)
variable cnt : unsigned(7 downto 0);
begin

  nsdone <= psdone;
  cnt := pstate;
  if (enable = '1') then
    cnt := pstate;
  else
    cnt := cnt + 1;
  end if;
  nstate <= cnt;

--this sets up the done state
--if count is greater than x, done goes low
  if (pstate > x) then
    nsdone <= '0';
  else
    nsdone <= '1';
  end if;
end process;

--this process sets up the registers for the counter/done state
process(clock, reset, enable, nstate, nsdone)
begin

if (reset = '0' or enable = '1') then
    pstate <= "00000000";
    psdone <= '0';
elsif (clock'event and clock = '1') then
    pstate <= nstate;
    psdone <= nsdone;
end if;
end process;

--assign an a value to the output
done <= not(psdone);

end architecture behavior;
The code for the register module

```vhdl
-- this program implements a register

library altera;
use altera.maxplus2.all;
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity registr is
  port(clock, reset, enable : in std_logic;
       id1, id2, id3, id4, id5 : in unsigned(7 downto 0);
       ipointer : in unsigned(10 downto 0);
       idatatype : in unsigned(1 downto 0);
       d1, d2, d3, d4, d5 : out unsigned(7 downto 0);
       pointer : out unsigned(10 downto 0);
       datatype : out unsigned(1 downto 0));
end entity registr;

architecture behavior of registr is

  signal pstate_a, nstate_a : unsigned (7 downto 0);  --d1
  signal pstate_b, nstate_b : unsigned (7 downto 0);  --d2
  signal pstate_c, nstate_c : unsigned (7 downto 0);  --d3
  signal pstate_d, nstate_d : unsigned (7 downto 0);  --d4
  signal pstate_e, nstate_e : unsigned (7 downto 0);  --d5
  signal pstate_f, nstate_f : unsigned (1 downto 0);  --idatatype
  signal pstate_g, nstate_g : unsigned (10 downto 0); --pointer

begin

  --this process sets up the inputs to the register
  process(id1, id2, id3, id4, id5, idatatype, ipointer)
  begin
    --set default values
    nstate_a <= pstate_a;
    nstate_b <= pstate_b;
    nstate_c <= pstate_c;
    nstate_d <= pstate_d;
    nstate_e <= pstate_e;
    nstate_f <= pstate_f;
    nstate_g <= pstate_g;

    --if the registers are enabled, then
    if (enable = '0') then
      nstate_a <= id1;
      nstate_b <= id2;
      nstate_c <= id3;
      nstate_d <= id4;
      nstate_e <= id5;
      nstate_f <= idatatype;
      nstate_g <= ipointer;
  end process;

```
else
end if;

--this process sets up the registers
process(reset, clock,nstate_a, nstate_b, nstate_c, nstate_d, nstate_e, nstate_f, nstate_g)
begin
if(reset = '0') then
pstate_a <= "00000000";
pstate_b <= "00000000";
pstate_c <= "00000000";
pstate_d <= "00000000";
pstate_e <= "00000000";
pstate_f <= "00";
pstate_g <= "00000000000";
elsif (clock'event and clock = '1') then
pstate_a <= nstate_a;
pstate_b <= nstate_b;
pstate_c <= nstate_c;
pstate_d <= nstate_d;
pstate_e <= nstate_e;
pstate_f <= nstate_f;
pstate_g <= nstate_g;
end if;
end process;

--this process assigns a value to the "clocked" outputs
process(pstate_a, pstate_b, pstate_c, pstate_d, pstate_e, pstate_f, pstate_g)
begin
d1 <= pstate_a;
d2 <= pstate_b;
d3 <= pstate_c;
d4 <= pstate_d;
d5 <= pstate_e;
datatype <= pstate_f;
pointer <= pstate_g;
end process;
end architecture;
The code for the adder module

--./-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.
--./ C:\My Documents\Thesis\Version 1.0\Programs\Counter.vhd /
--./ Version : 1.0 /
--./ Author : Kishma A. Meyers /
--./ Date : January 22, 2000 /
--./ Created on : December 26, 1999 by Kishma A. Meyers /
--./-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.

--the purpose of this program is to implement the adder
--that generates the numerator and denominator values for
--the divider.

library altera;
use altera.maxplus2.all;
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

--define input and output pins
entity adder is
  port(d1, d2, d3, d4, d5 : in unsigned(7 downto 0);
      num, den : out std_logic_vector(10 downto 0)
  );
end entity adder;

--define architecture
architecture behavior of adder is
  begin
    process(d1, d2, d3, d4, d5)
    variable sum1 : unsigned(8 downto 0);
    variable sum2 : unsigned(7 downto 0);
    variable sum3 : unsigned(8 downto 0);
    variable sum4 : unsigned(9 downto 0);
    variable sum5 : unsigned(9 downto 0);
    variable suma : unsigned(10 downto 0);
    variable sumb : unsigned(10 downto 0);
    variable sumc : unsigned(10 downto 0);
    begin
      --implementation of first 8-bit adder, 9-bit result
      sum1 := d1+d2;
      
      --implementation of second 8-bit adder, 8-bit result
      sum2 := d3-d1;
      
      --implementation of third 8-bit adder, 9-bit result
      sum3 := d4+d5;
      
      --implementation of 9-bit adder, 10-bit result
      sum4 := sum3 + sum1;
      
      --implementation of multiplier, 10-bit result
      sum5 := sum3(8 downto 0) & '0';
      
      --implementation of first 10-bit adder, 11-bit result
      suma := d5 + sum5;
    end process;
  end architecture;
--implementation of second 10-bit adder, 11-bit result
sumb := d3 + sum4;

--implementation of third 10-bit adder, 11-bit result
sumc := sum2 + suma;

den <= conv_std_logic_vector(sumb,11);
num <= conv_std_logic_vector(sumc,11);

end process;
end architecture behavior;
The code for the divider module

library altera;
use altera.maxplus2.all;
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

--this program implements a 4-bit divider.
--define input and output pins
entity divider is
port(num, den :in unsigned(10 downto 0);
    quotient : out unsigned(3 downto 0)
    --dummy_1, dummy_2 : out unsigned(10 downto 0);
    );
end entity divider;
architecture behavior of divider is
--2-bit extension of the numerator
signal extnum : unsigned(13 downto 0);
--2-bit extension of the denominator
signal extden : unsigned(13 downto 0);
--the following are signals that pass the sum
--from process to process
signal sum_a : unsigned(14 downto 0);
signal sum_b : unsigned(14 downto 0);
signal sum_c : unsigned(14 downto 0);
signal sum_d : unsigned(14 downto 0);
begin
    --add three zeroes at the LSB
    extnum <= num & "000";

    --this process inverts the bits of the denominator
    process(den)
    variable i : integer;
    variable notden : unsigned(10 downto 0);
    begin
        for i in 0 to 10 loop
            notden(i) := not(den(i));
        end loop;
        --and adds one to the inverted denominator to
        --form the two's complement, add two zeroes at LSB
        extden <= (notden + 1) & "000";
    end process;

process(extnum, extden)

variable sum : unsigned(14 downto 0);
begin
sum := extnum + extden(13 downto 0);
quotient(3) <= sum(14);
if (sum(14) = '0') then
    sum := '0' & extnum;
else
    sum := '0' & sum(13 downto 0);
end if;
sum_a <= sum;
end process;

process(extden, sum_a)
variable sum : unsigned(14 downto 0);
begin
sum := sum_a + ('1' & extden(13 downto 1));
quotient(2) <= sum(14);
if (sum(14) = '0') then
    sum := sum_a;
else
    sum := '0' & sum(13 downto 0);
end if;
sum_b <= sum;
end process;

process(extden, sum_b)
variable sum : unsigned(14 downto 0);
begin
sum := sum_b + ('11' & extden(13 downto 2));
quotient(1) <= sum(14);
if (sum(14) = '0') then
    sum := sum_b;
else
    sum := '0' & sum(13 downto 0);
end if;
sum_c <= sum;
end process;

process(extden, sum_c)
variable sum : unsigned(14 downto 0);
begin
sum := sum_c + ('111' & extden(13 downto 3));
quotient(0) <= sum(14);
if (sum(14) = '0') then
    sum := sum_c;
else
    sum := '0' & sum(13 downto 0);
end if;
sum_d <= sum;
end process;

end architecture;
The code for the finder module

```
library altera;
use altera.maxplus2.all;
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
entity finder is
  port(pointer : in unsigned(10 downto 0);
       q : in unsigned(3 downto 0);
       strip_pointer : out unsigned(13 downto 0));
end entity finder;

architecture behavior of finder is
  --this signal represents (pointer - decimal 2) * 4
  signal pointer2_x_4 : unsigned(12 downto 0);
  signal quotient : unsigned(3 downto 0);
  begin
    pointer2_x_4 <= (pointer + "11111111110") & "00";
    quotient <= q(3 downto 1) + ("00" & q(0));
    strip_pointer <= pointer2_x_4 + ("0000000000" & quotient);
  end architecture;
```
11. APPENDIX C

Altera GDF Schematic of Decoder Partition

Altera GDF Schematic of Buffer Filler Partition
Altera GDF Schematic of Calculator Partition