Institutionen för systemteknik  
Department of Electrical Engineering

Examensarbete

Hardware bidirectional real time motion estimator  
on a Xilinx Virtex II Pro FPGA

Master thesis
Rashid Iqbal
LiTH-ISY-EX--06/3758--SE
APRIL 26, 2006

TEKNISKA HÖGSKOLAN  
LINKÖPINGS UNIVERSITET

Department of Electrical Engineering  
Linköping University
S-581 83 Linköping, Sweden

Linköpings tekniska högskola  
Institutionen för systemteknik
581 83 Linköping
Hardware bidirectional real time motion estimator on a Xilinx Virtex II Pro FPGA
Master thesis in Division of Electronic Systems at Linköping Institute of Technology

by

Rashid Iqbal
LiTH-ISY-EX--06/3758--SE

Supervisor: Prof. Dr.-Ing. Rolf Ernst (Institute of Computer & Communication Network Engineering, Technical University Braunschweig Germany)
Examiner: Assist. Prof. Dr. Per Löwenborg Linköping, April 26, 2006
Abstract
This thesis describes the implementation of a real-time, full search, 16x16 bidirectional motion estimation at 24 frames per second with the record performance of 155 Gop/s (1538 ops/pixel) at a high clock rate of 125 MHz. The core of bidirectional motion estimation uses close to 100% FPGA resources with 7 Gbit/s bandwidth to external memory. The architecture allows extremely controlled, macro level floor-planning with parameterized block size, image size, placement coordinates and data words length. The FPGA chip is part of the board that was developed at the Institute of Computer & Communication Networking Engineering, Technical University Braunschweig Germany, in collaboration with Grass Valley Germany in the FlexFilm research project. The goal of the project was to develop hardware and programming methodologies for real-time digital film image processing. Motion estimation core uses FlexWAFe reconfigurable architecture where FPGAs are configured using macro components that consist of weakly programmable address generation units and data stream processing units. Bidirectional motion estimation uses two cores of motion estimation engine (MeEngine) forming main data processing unit for backward and forward motion vectors. The building block of the core of motion estimation is an RPM-macro which represents one processing element and performs 10-bit difference, a comparison, and 19-bit accumulation on the input pixel streams. In order to maximize the throughput between elements, the processing element is replicated and precisely placed side-by-side by using four hierarchal levels, where each level is a very compact entity with its own local control and placement methodology. The achieved speed was further improved by regularly inserting pipeline stages in the processing chain.

Keywords
Bidirectional motion estimation, FPGA, block matching, sum of absolute differences, systolic array, SARow, PE2X8, MeProC, 125 MHz, Virtex II Pro, Relationally placed macro, CLB, slice, tristate buffers, comparator, pipelining, search upper, search lower, VHDL, Xilinx, MeEngine, LMC, CMC.
I dedicate this thesis to my dear & loving wife zaibee
vii
Preface

Abstract
This text is the report of the master thesis which is done at the Institute of Computer and Communication Network Engineering (IDA), Technical University of Braunschweig. In this memorandum the theoretical and practical work during the 6 month period is documented. The background idea when writing this report was the reusability of this work in further developments.

Chapter 1 of this report offers an introduction to the digital cinema video and associated challenges for real time operations on very high resolution images, description and motivation involved in this thesis work.

Chapter 2 provides description of full search block matching algorithm using sum of absolute differences (SAD) as cost function.

Chapter 3 starts by explaining the basic data processing architecture and its individual components. It also gives description of the data and control flow within the systolic array, search upper and lower division of the search area, problems at boundaries of the image and their solutions. It then extends the discussion to the entire FPGA by providing brief explanation of the different components within each motion estimation engine and its interface to external frame buffers.

Instead of providing only the final picture Chapter 4 gives an in depth information about the entire evolution process, where a myriad of architectural modifications were made before reaching the final architecture. Justification for each and every modification is also provided for better understanding.

Severe area constraints on the hardware used for bidirectional motion estimation resulted in severe timing problems, explanation of these timing problems and their remedy is discussed in Chapter 5.
Chapter 6 gives full description of all the hierarchal stages used. A detail explanation about each and every component, part of the final architecture is provided. In addition to the local control within each stage, different blocks of the global control are also described.

The placement of different components on the FPGA adopts a very flexible methodology. Chapter 7 focuses on implementation of different schemes part of different hierarchal stages.

Chapter 8 states the importance of the different components used for providing proper memory interface to the motion estimation hardware. It also gives a description of the weakly programmable memory interface components and how special accesses of memory required by motion estimation processor are accomplished.

Chapter 9 concludes the report.
Acknowledgments

First of all, I am thankful to Prof. Dr. Rolf Ernst, director Institute of Computer and Communication Network Engineering (IDA), for accepting me as 'thesis' student in his research group. The provided infrastructure and development tools are certainly key points towards my successful work.

Many thanks to Amilcar do Carmo Lucas. He was my adviser at IDA during my thesis work and he always provided me great help and guidance not only in my academic work but with my residential and other settlement issues. I learnt a lot from his experience with reconfigurable devices.

I would also like to thank all the members of IDA specially to the thesis and 'Hiwi' students for providing me very friendly company during my six months stay.

I acquired great deal of knowledge from Dr. Per Löwenborg during my 'Mixed-signal processing project' at Linköping University. He is also my supervisor for thesis at Linköping University for which he deserves special thanks from me.

And off course very special thanks to my wife, Zaibee, for without her I would be wondering around without master degree, as just after 10 days of our marriage I was attending my first lecture at Linköping University. I give full credit to her as her company fills my life with deep sense of joy and satisfaction.

Rashid Iqbal
Braunschweig,
Dec 28, 2005.
# Table of contents

**Preface** ........................................................................................................................................... viii

1. Introduction ........................................................................................................................................... 1
   1.1 Digital cinema ................................................................................................................................. 1
   1.2 Challenges for real time operations on digital cinema video .................................................... 2
   1.3 Description of the Thesis ............................................................................................................... 2

2. Motion Estimation Algorithm .............................................................................................................. 5
   2.1 Overview ........................................................................................................................................ 5
   2.2 Full search block matching algorithm ....................................................................................... 5
   2.3 Bidirectional motion estimation .................................................................................................. 8

3. Data processing & system architecture .............................................................................................. 9
   3.1 Basic processing architecture ...................................................................................................... 9
   3.2 Data flow ...................................................................................................................................... 10
   3.3 Pixel access pattern ...................................................................................................................... 11
   3.4 Search area division into 'Upper' and 'Lower' ............................................................................ 12
   3.5 Boundary problems and solutions .............................................................................................. 13
   3.6 System architecture ...................................................................................................................... 16
      3.6.1 Structure for bidirectional motion estimation ...................................................................... 17
      3.6.2 Inclusion of motion compensator ....................................................................................... 18

4. Data Path Evolution ............................................................................................................................. 21
   4.1 Overview ...................................................................................................................................... 21
   4.2 Placement on FPGA ...................................................................................................................... 21
   4.3 Xilinx Virtex II Pro architecture (brief explanation) .................................................................. 21
   4.4 RPM methodology ....................................................................................................................... 23
   4.5 Design methodology ..................................................................................................................... 24
   4.6 Basic Processing Element ............................................................................................................ 25
      4.6.1 New multiplexer architecture ........................................................................................... 31
   4.7 PE2X8 (group of 16 processing elements) ................................................................................... 32
   4.8 Systolic Array Row (SARow) ....................................................................................................... 36
   4.9 Motion estimation processor (MeProC) ........................................................................................ 37
5. Further improvements in data path design .......................................................... 39
   5.1 Overview ................................................................................................. 39
   5.2 Need for pipelining in the path of comparator ........................................... 39
   5.3 Long straight horizontal tristate buses ...................................................... 41
   5.4 LUT approach for shift register implementation ...................................... 44
   5.5 rowmux implementation ........................................................................... 46
   5.6 Need for extra pipelining and extra slice .................................................... 47
   5.7 Search upper and lower IO register placement ......................................... 48
   5.8 Reverse placement for odd rows .............................................................. 49
   5.9 upper_lower_sel flag placement ............................................................... 51

6. Detail Explanation including Controller ......................................................... 53
   6.1 Overview ................................................................................................. 53
   6.2 Basic processing element .......................................................... 54
   6.3 PE2X8 ..................................................................................................... 57
       6.3.1 Cascaded 16 processing elements ...................................................... 58
       6.3.2 Tristate buffers .............................................................................. 59
       6.3.3 Pipelining .......................................................................................... 59
       6.3.4 Local control ..................................................................................... 59
   6.4 SARow ..................................................................................................... 60
       6.4.1 Combination of PE2X8 components .................................................. 60
       6.4.2 Pipelining .......................................................................................... 61
       6.4.3 Local control ..................................................................................... 61
   6.5 MeProC ................................................................................................. 63
       6.5.1 Motion estimation Data Path (MeDP) .................................................. 64
           6.5.1.1 Data Path Systolic Array (DP_SA) .............................................. 66
           6.5.1.2 DP_SA_OP ( Systolic Array Output ) ......................................... 66
           6.5.1.3 DP_C ( Comparator ) ................................................................. 67
           6.5.1.4 DP_MV (Motion Vectors) .......................................................... 67
       6.5.2 MeGC (Motion Estimation Global Controller) ...................................... 68
           6.5.2.1 ME Starter (MS) ....................................................................... 69
           6.5.2.2 Processing Element Accumulator Initializer & Error Enable ........ 70
           6.5.2.3 Processing Element Search Upper/Lower Stream selection .......... 70
           6.5.2.4 Processing Element Motion Vector Counters (PE_MVC) .......... 71
           6.5.2.5 Top left Accumulator Initializer & Data Redirector .................. 71
           6.5.2.6 Boundary Detector (BD) .............................................................. 72
           6.5.2.7 Read Stream (RS) .................................................................... 72
           6.5.2.8 Reference Stream Reader (R_RS) ................................................ 73
           6.5.2.9 Search Upper Stream Reader (R_SU) .......................................... 73
           6.5.2.10 Search Lower Stream Reader (R_SL) ......................................... 73
6.5.2.11 Pixel Stream Disabler (R_SD) ................................................................. 73

7. Flexible placement over FPGA ........................................................................ 75
  7.1 Overview ......................................................................................................... 75
  7.2 Placement methodology for PE2X8 ............................................................... 75
     7.2.1 Placement starting with even horizontal location ............................... 76
           7.2.1.1 Placement of first 8 PEs ............................................................. 76
           7.2.1.2 Placement of second 8 PEs ......................................................... 77
     7.2.2 Placement starting with an odd horizontal location ............................. 77
          7.2.2.1 Placement of first 8 PEs ............................................................. 78
          7.2.2.2 Placement of second 8 PEs ......................................................... 79
  7.2.3 Middle region .......................................................................................... 78
     7.2.3.1 First 8 reg placement ...................................................................... 78
     7.2.3.2 Second 8 reg Placement .................................................................. 79
     7.2.3.3 Error multiplexer (error_mux) ....................................................... 79
     7.2.3.4 Tristate buffers for error_mux (buf_for_error_mux) .................... 80
  7.2.4 Reverse placement for odd rows .............................................................. 80
  7.3 SARow placement ....................................................................................... 81
     7.3.1 Placement starting with Even location ................................................. 82
           7.3.1.1 Forward Placement .................................................................. 82
           7.3.1.2 Reverse Placement .................................................................. 84
     7.3.2 Placement starting with an odd horizontal location ......................... 86
           7.3.2.1 Forward Placement .................................................................. 86
           7.3.2.2 Reverse placement .................................................................. 87
  7.4 Placement scheme for MeProC ................................................................. 88
  7.4.1 Four Row architecture placement ......................................................... 89
  7.4.2 Eight rows architecture placement ....................................................... 91

8. Memory Interface ......................................................................................... 93
  8.1 Overview ...................................................................................................... 93
  8.2 Local Memory Controller .......................................................................... 93
  8.3 Generator .................................................................................................... 95
  8.4 Local Address Controller ........................................................................... 97
  8.5 Algorithm Controller .................................................................................. 99
  8.6 Global Base Stepper .................................................................................. 100
  8.7 Building blocks of C2S, APT & S2C ......................................................... 101
  8.8 Final Chip ................................................................................................... 102
  8.9 Verificaiton plan ......................................................................................... 104
     8.9.1 Verification of MeProC .................................................................... 105
           8.9.1.1 Matlab Scripts ....................................................................... 105
           8.9.1.2 VHDL Testbench .................................................................... 106
     8.9.2 Verification of MeProC with Memory Interface ................................. 106
1. Introduction

1.1 High resolution digital cinema

Digital cinema refers to the use of digital technology, digital video or high definition TV, to make, distribute and project motion pictures. Special camcorders are used to shoot the movie as digital files on tape, hard disk or other electronic storage device rather than on film. The final movie can be distributed electronically and projected using a digital projector instead of a conventional film projector [1].

HDTV technology is providing a link which is used to help make 'Electronic Cinema' a reality. High definition simply means ‘more than standard definition’. The highest resolution SDTV format is PAL, with 576 lines. Thus, almost any video with a frame size greater than 576 lines tall is some type of HD. HD video is generally either 1920 x 1080 or 1280 x 720, with a 16:9 aspect ratio.

The definition of digital cinema is still evolving and moving towards resolution of 2048 x 2048 and 4096 x 4096. Intel co-founder Gordon Moore made an audacious prediction about computing decades ago: the number of transistors of an integrated circuit doubles at regular intervals. The modern interpretation of Moore’s law is that computing power, at a given price point, doubles about every 18 months. Consequently a system three years ago was a quarter of the speed, and a system four and half years ago was one eighth of the speed. Similarly a pixel size of 1920 x 1080 (at 60 fields per second) is only 6.5 times greater than the NTSC SD standard of 720x480 (60i). With the trend of 2k x 2k and 4k x 4k we can expect drastic increase in these formats over the next years.

High quality on screen picture performance is the ultimate technical goal of Digital Cinema and these quality requirements are much higher than they are for consumer HD, and Hollywood likes it this way- ‘they want theatrical experience to have an advantage over the home experience that goes beyond popcorn’. Video formats are either 8 bit per pixel or 10 bit per pixel. With 10 bit per pixel we have more grey levels as compared to 8 bit, which results in extra detail in video and hence improved quality.
1.2 Challenges for real time operations on digital cinema video

Different real time operations might be performed on cinema video, these operations may include, colour correction, compression, noise reduction etc. Real time operations are easier to accomplish with SDTV as compared to HDTV or Digital Cinema. Digital cinema use images with very large resolution in the range of 2k x 2k or 4k x 4k. For example, at resolutions of 2k x 2k (2048 x 2048) pixels per frame at 30 bit/pixel and 24 frames/s results in an image size of 15M bytes and a data rate of 360M bytes per second. Processing at these rates is not easy for normal sequential computing machines like processors. Instead dedicated hardware with huge computing resources is required. ASIC is also not a feasible solution because of low volume market of digital cinema production.

Flexibility is another important factor associated with a system performing real time operations on digital cinema video. The hardware system should be flexible enough to adapt to any of already present formats and also to any new format that might be developed in future.

Reconfigurable computers using FPGAs provide a very flexible platform which offers a compromise between the performance advantages of fixed-functionality hardware and the flexibility of software-programmable DSPs. FPGAs have made video Processing market more competitive as industry is getting new versions of FPGAs with more DSP resources and embedded-processing capabilities. In addition FPGA allow extreme level of parallelism for operations and this capability can exploit the inherent parallelism in video/image processing algorithms.

1.3 Description of the Thesis

Institute of Computer Communication & Network Engineering (IDA), TU Braunschweig is working on FlexFilm Project, a cooperation between two universities and an industry partner. System under development is a multi-board, extendible FPGA based platform having the capability to perform complex real time video processing algorithms on high resolution digital cinema video. This processing is achieved with huge communication and storage bandwidth and enormous computational power.
One of such PC extension boards is shown in Figure 1.1. It contains four Xilinx Virtex XC2VP50-6 devices containing huge computational resources. Each FPGA has 4G bit DDR-SRAM based external memory space, 2G bit on one side of FPGA while 2G bit on the other side. The memory banks operate at 125MHz, giving rise to data rates of 7 Gbit/s. The board also contains PCI communication network in order to have communication either with host or to other extension boards.

To test the system architecture developed, a complex noise reduction algorithm is implemented at 24 frames per second. This algorithm makes use of motion estimation, motion compensation and discrete wavelet transformation between consecutive images. The algorithm starts by performing bidirectional motion estimation on previous and next image, the result of the motion estimation is used
to produce a motion-compensated image. Then it performs a Haar filter between this image and current image. The two resulting images are then transformed into the 5/3 wavelet space, filtered with user selectable parameters, transformed back to the normal space and filtered with the inverse Haar filter.

The algorithm is divided into the three FlexWAFE FPGAs on the flexfilm board. The fourth FPGA is used for IO communication using PCI Express network. FlexWAFE0 is reserved for implementing bidirectional motion estimation as shown in Figure 1.1.

This thesis describes the implementation of real-time, full search, 16x16 bidirectional motion estimation at 24 frames per second with record performance of 155 Gop/s (1538 op/pixel) at high clock rate of 125MHz. The core of bidirectional motion estimation uses close to 100% FPGA resources with 7 Gbit/s bandwidth to external memory. The architecture allows extremely controlled, macro level floor-planning with parametrized block size, image size, placement coordinates and data words length. Summary of the important characteristics of the implemented bidirectional motion estimator is shown in Listing 1.1.

- bidirectional ME with block size 16x16
- searches -8/+7 vector interval
- 24 fps @ 2048x2048, 10bpp (125 MHz)
- 1024 net add/sub operations/pixel
- 514 net comparations operations/pixel
- 155 net Goperations/s

Listing 1.1. Summary of the computational power associated with motion estimator.
2. Motion Estimation Algorithm

2.1 Overview

Generally, the purpose of a motion estimator is to find the direction and amount of movement between two consecutive images or two consecutive frames of video. In the field of image/video coding motion estimation is applied for the elimination of the temporal redundancy of video material and is therefore a central part of the video coding standards ISO/IEC MPEG-1, MPEG-2, and MPEG-4 as well as the ITU-T H.261 and H.263 recommendations. The FlexFilm project uses motion estimation to produce magnitude and direction-motion vectors for motion compensator which will then use these vectors in order to produce an image with reduced noise.

A wide variety of ME algorithms exist, offering trade-offs between speed, complexity and quality of motion vectors obtained. There are two main techniques of motion estimation: pel-recursive algorithm (PRA) and block-matching algorithm (BMA). PRAs are iterative refining of motion estimation for individual pels by gradient methods. BMAs assume that all the pels within a block have the same motion activity. BMAs estimate motion on the basis of rectangular blocks and produce one motion vector for each block. PRAs involve more computational complexity and less regularity, so they are difficult to realize in hardware.

2.2 Full search block matching algorithm

In general, BMAs are more suitable for a simple hardware realization because of their regularity and simplicity.
Figure 2.1. Reference block and its corresponding search area in the search image.

Figure 2.1 explains block matching algorithm where the reference image is divided into smaller blocks each of size NxN. Each of these blocks is then searched out over a reduced area in the search image. Best match is determined by finding an optimum value of the cost function in the search area. Block matching algorithms can differ in term of search criteria selection and cost function selection.

In the full search algorithm, the motion vectors are calculated for all possible blocks of search image. This algorithm is highly computational intensive and is also known as exhaustive search or brute-force search and it finds the absolute minima of the search function. Therefore it does not suffer from the local minima problem that other motion estimation algorithms have. Example of such algorithms are the three step search, four step search and diamond search. For example consider Figure 2.2.

A block size of NxN is being searched in a rectangular area of size Q x Q where Q=N+2p-1, p being the maximum displacement of the reference block in four spatial directions. So the cost function will be evaluated \( N^2 \times (2p)^2 \) times in entire search area for one reference block.
Different error measures can be used for motion estimation. The two mostly used cost functions are Mean Square Error (MSE) and Sum of Absolute Differences (SAD). Simulations show that both of these measures give very similar results. Square calculation in MSE is expensive to implement in hardware in terms of area, instead SAD is very simple to implement [3]. This function is shown in listing 2.1

\[
SAD = \sum_{m=1}^{M} \sum_{n=1}^{N} | x(m, n) - x(m+i, n+j) |
\]

**Listing 2.1**

where \( X \) is the reference image and \( X_A \) is the search image.

The above equation can be decomposed into loops as:
The above loops can be executed either in a sequential or in parallel way. The approach taken up in this thesis will be explained in the next chapter. A serial approach comes with considerable latencies and with reduced bandwidth while a parallel approach has enormous advantages in terms of low latencies and high throughput with considerable increase in bandwidth. A parallel approach also needs huge hardware resources for implementation.

2.2 Bidirectional motion estimation

Bidirectional ME produces motion vectors in two directions. The two directions represent one reference image, to be searched in forward and backward image as shown in following Figure 2.3.

![Bidirectional motion estimation diagram](image)

Figure 2.3 Concept of bidirectional motion estimation.
3.0 Data processing & system architecture

3.1 Basic processing architecture

In this thesis both i and j loops mentioned in listing 2.2 are performed in parallel and therefore it is necessary to have NxN processing elements and as a result motion vectors are computed inNxN operation cycles [3]. The selected block size of 16x16 pixels is according to H.261 and MPEG-2 standard however the developed VHDL code is also fully parametrizable and flexible enough for other block sizes as well.

With block size of 16x16 pixels data is processed in the form of 256 Processing Elements (PEs), all of them working together to give motion vectors in the range of -8/+7. Each processing element independently performs SAD operation i.e. subtraction, absolute and accumulate on the two input pixels coming from the two images.

![Diagram of basic operations performed within each PE.](image)

Figure 3.1. Basic operations performed within each PE.

Interconnection of 256 PEs is accomplished in such a manner that reference pixel stream flows through all the processing elements in the form of shift registers. Certain control
signals used within each PE also flow in similar fashion to reference stream. This information flow gives rise to a systolic architecture (SA) structure. In addition to the shifted control flow each PE within the systolic array needs separate control which comes from a central Global Control (GC). A comparator module then performs required comparisons on all the accumulated SAD vectors to find the absolute minimum error and the corresponding motion vectors.

3.2 Data flow

In order to have an in depth understanding of the data flow and resultant data processing architecture we start with a simple example where we have two images, one is called search image and other one called reference image. A block of size 8x8 pixels is extracted from the reference image. Assume the block is extracted from middle of the image where corresponding search area contains all valid pixel locations. The reference block has to be searched out in 'search image', in a square area of width and height to be 15x15 pixels i.e. range of motion vectors is -4/+3.
3.3 Pixel access pattern

We start with a very simple methodology for the sake of explanation, where the very first block within search area to be compared with reference block is shown in Figure 3.3 with '0s'. Once this comparison is finished the next block to be fetched from search is shown in Figure 3.4 with '1s'. Similarly for the next area we need to move down by one row. In this way eight vertical blocks are completed. Next block to be fetched for comparison is obtained by shifting to the right by one pixel and again starting from the top row. In this way we have 64 blocks in the search area to be compared with the reference block.

![Figure 3.3. First block access.](image)

![Figure 3.4. Second block access.](image)

The above explanation represents a sequential approach where the same pixel is fetched several times for different blocks, however a different methodology was chosen for this thesis that makes use of the SA-architecture where data fetching, processing and motion vector calculation is highly parallel without any halt/wait cycle with tremendous increase in bandwidth.

Both the reference block and search block are being read column-wise where the reference stream is fed to the first processing element. So the idea here is that when complete reference block (64 pixels) is read (at the end of 64 cycles) processing element PE1 should give SAD corresponding to the block of the search area shown in Figure 3.3 and then on the next clock cycle PE2 should give a valid SAD corresponding to Figure 3.4. At the same time PE1 has started taking the next
reference block. On the next cycle PE3 will give the desired SAD and similarly after further 64 cycles we get SAD vector from PE64 and at this time we have calculated all the SADs required to find the motion vector for the first reference block (after 2x64 cycles). On the next cycle we will get SAD from PE1 again but this time for the next reference block. This example can easily be extended to a reference block size of 16x16 pixels where 256 processing elements will act together and valid motion vectors for the first search block will be obtained after the end of 512 clock cycles.

Figure 3.5. PEs arranged in a systolic array taking reference data.

3.4 Search area division into 'upper' and 'lower'

The methodology explained above suggests that different pixels of the search area are fed to different processing elements at the same time with shifted values of the reference data. Because of this independent access mechanism it is better to divide the search area into two parts, 'Search upper area' and 'search lower area'. For this particular example, search upper (SU) consists of upper 8x15 pixels of the search area while search lower consists of the lower 7x15 pixels of the search area. These two search parts are being read simultaneously column-wise and are being given to the desired processing elements. It can be be observed that processing elements 1,
9, 17, …. 57 always take data from search upper. PE2 takes the first 7 elements of
the column from search upper and one element from search lower. Similarly PE 8,
16, …. 64 always take the first pixel of each column from search upper and rest of
the pixels from search lower. This also suggests that except the top row of PEs,
each processing element will be fed with both search upper pixel stream and search
lower stream and inside each PE there should be a multiplexer to select either of
the two streams and the control signal for the multiplexer should be generated from
the global controller.

3.5 Boundary problems and solutions

Data fetching and its processing is normal when the search area contains no invalid
pixel locations. However if the search area extends beyond boundaries as shown in
Figure 3.6, then data fetching and its processing is different which needs
modifications in the data processing architectures. Separate blocks for boundary
detection and corresponding control generation are required. Special control is thus
needed for all the blocks which are present at the boundaries of the image.

In order to explain the change in data flow and the resulting architectures at the
boundaries we consider an example where again the reference block size is
assumed to be 8x8 pixels and the entire search image contains 3 horizontal and 3
vertical blocks of reference size. The dotted boundary is shown around the search
image in order to show a part of the search area which extends outside the image
for the reference blocks at the boundaries. We start with the top-left corner of the
image where we start fetching Blk0 of the reference image and feed this to the
processing elements. According to normal data flow procedure we should start
feeding the reference pixel stream to the first PE but for this block which is present
at the top left boundary we can’t do this as processing element # 1 (PE1) is
supposed to get data from a location which actually does not exist on search image.
For this special case we can see that only PE#37…40, 45… 48, 53.. 56, 61…64
(shown in Figure 3.5 ) can get pixels from the valid locations of the search image.
Figure 3.6. Example of an image containing 9 reference blocks of size 8x8.

So the solution is that PE33 is fed with the reference data for the first time. Of course PE33 also needs pixels from invalid locations but we can achieve the desired results by not start reading for search upper area until the first reference pixel enters PE37. Another important consideration is that we can not start reading from first pixel (top left) of reference block, instead we should start reading from 5th column of reference block (Figure 3.5) that is we only need to read the last 32 elements of the top left reference block.

An important and very interesting aspect of this algorithm is that memory access for pixels is very continuous and as soon as we complete reading the first reference block (Blk0), the next reference block starts immediately by feeding it to the first processing element (PE1). Reading of the search upper and lower areas are then
exactly aligned with the current and next reference block. So we do not need to make a big step backward for reading the search area whenever the next reference block comes in. This is the beauty of this algorithm and because of this factor we cannot feed PE33 the entire reference block.

After 32 cycles, PE33 will get all the last 32 pixels from the Blk0. Reading search upper is not continuous and a column of only four pixels is being read from search area for search upper. As soon as 32 cycles are complete PE33 should switch to PE32 for getting the reference pixel stream instead of getting it externally as was the case for the first 32 cycles. Another interesting thing here is that, as we start feeding reference data to PE33, the same data is also given to PE1 although it is useless for PE1 as it is not given any search area pixel but this is important because as 32 cycles are complete and PE33 switches to take reference input from PE32, it will again keep on getting the same part of the reference block for the next 32 cycles. In this way we have compared the second part (last 32 pixels) of Blk0 initially to the first part of the area contained in the search image and then to the second part of the search area, in this way we are able to get desired motion vectors.

Next reference block (Blk1) is being fed to the first processing element (PE1) as soon as the second half of the first reference block finishes. This is the time when PE1 will start calculating SAD for the next reference block and this is also the time when controller will start reading SAD one by one starting from PE1 to give them to comparator for calculating the Motion Vectors for the first reference block. Off course from PE1 to PE36, data will be erroneous and correct motion vectors can only be 0,1,2, and 3 for both x and y.

Valid vectors for first reference block will be available after 32+64 cycles and then onward motion vectors will be available for every other reference block present in reference image.
In order to have correct motion vectors at correct time 'motion vector counters' (MVC) should start with appropriate values.

After finishing reading Blk1, we start reading the third reference block (Blk2) which is again present at the boundary but this time it is the right boundary. Once we finish reading all 64 pixels of the third reference block (Blk2) we immediately start reading Blk3 (fourth block) which is present at the left boundary again. So as soon as we starting feeding Blk3 to PE1, we are in fact reading search area which
corresponds to Blk2 on the search area. Reading this search area is important because still some processing elements are calculating SAD vector for Blk2. As we finish reading the first 32 pixels of Blk3, we start fetching the search area in the second row of the image. This in fact does not create any problem because PE33 is getting the correct reference block for matching with correct search area part. An important thing here is that once we again reach the left boundary and start accessing memory for search area, this time we will get valid pixels for all search upper pixels so proper control should be initiated for this. This process keeps on happening until we reach Blk6 which is the bottom boundary of the image. This time not all the pixels for Search Lower are available, so controller should generate proper signals for this part.

3.6 System architecture

Now after describing the algorithm and related data flow and its basic processing architecture this is the proper time to explain the system architecture before further going into detailed analysis of the individual components of the system.

The motion estimator is called 'Motion Estimation Engine (MeEngine)'. It consists of two blocks, one is the main 'Motion Estimation Processor (MeProC). This component contains a systolic array of processing elements and necessary global control.

MeProC needs column wise data from external memory which is not a regular way of accessing memory. Therefore another component called 'Motion Estimation Data Transformer (MeDataTF)' is required. In order to access external memory (DDR SDRAM) at a high bandwidth a scheduling memory controller (CMC) is developed at IDA. Thus MeDataTF has interface to CMC on one side while on other side it has an interface to MeProC.

As MeProC needs three pixel streams, we need three modules of MeDataTF for reference, search upper and search lower pixel streams. In addition to MeDataTF and CMC, MeEngine needs another module for receiving external frames of video and writing them to external memory (Frame Buffers), this component is called 'Motion Estimation Data Writer (MeDataWR)'. Thus, the complete block diagram of MeEngine is shown in Figure 3.7.
3.6.1 Structure for bidirectional motion estimation

For Bidirectional motion estimation two MeEngines (MeEngine0 and MeEngine1) are used. The two engines access memory and generate motion vectors independently.
Figure 3.8 shows the required frame buffer access structure for the motion estimator. As can be seen, three images are accessed simultaneously, one image as reference (n-2), and two images are backward and forward search area (n-3 and n-1). The two search areas are read twice with different addresses. Besides that, the current incoming image (n) needs to be buffered. Each of the two MeEngines contain its own frame buffer to store four full-size images of up to 4k x 4k accessed via its respective CMC0 or CMC1. Each of the CMCs writes one stream to memory and reads three streams. For ease of implementation each pixel is stored using 16 bits. This translates to 1.5 Gbit/write and 4.1 Gbit/second read bandwidth to off-chip SDRAM amounting to a total of 6.1 Gbit/second that is below the maximum practical bandwidth of 7 Gbit/second [6].

3.6.2 Inclusion of motion compensation block

Implementation of this hardware on one FPGA chip with the desired frequency of 125MHz remained the biggest challenge throughout this thesis. However the approach taken has produced highly optimized and perfectly placed MeProC macros, which managed to place not only both MeEngines but it also produced enough room on the FPGA to place 'Motion Compensator' block on the same FPGA. So the Figure 3.9 shows all the hardware blocks that are part of FlexWAFE0 chip. This Figure also shows one RGB to luminance conversion for motion estimator. Motion vectors generated by MeEngine0 and MeEngine1 are used by the motion compensator to generate a motion compensated image which is transmitted out of the chip.
Figure 3.9. All the components contained in FlexWAFE 0 FPGA.
Data Path Evolution

4.1 Overview

In this chapter data path of motion estimator will be explained. Rather than giving only the final structure, a full evolution of the final structure will be described because starting from the basic architecture it was a continual exploration process where a myriad of architectural modifications were made and were tested against desired area and performance measures. Attaining performance of 125MHz at very reduced rectangular area over the FPGA die remained the biggest challenge throughout the evolution process.

4.2 Placement on FPGA

FPGA placement of all the data-path/control units is also a very important concern while designing data path because performance is a function of the placement of design unit. Design and placement schemes went side by side during the entire evolution course of action. Whenever an entity was designed, it was synthesized and implemented to see performance and area figures using Xilinx tools. This is important because relative location of different units decides interconnects length which plays an important role in performance measures. So before going into a detailed discussion about the design, place and route of different units at different hierarchical levels, its better to know certain important characteristics of the target device which is XC2V50-6 Xilinx FPGA.

4.3 Xilinx Vertix II Pro architecture (brief explanation)

The basic logic building block of a Virtex FPGA is a CLB and basic building block of CLB is the slice. A CLB contains four slices organized as two cells. A slice includes a 4-input function generator, carry logic and a storage element. The
The functional generator is implemented as a 4-input look-up table (LUT) and can implement any 4-input logic function. The two look-up tables in one slice can be combined together to create a 16x2 bit or 32x1-bit synchronous RAM or 16x1-bit dual-port synchronous RAM. A Virtex LUT can also implement a 16-bit shift register with significantly reduced area, when compared to a discrete chain of registers solution.

Dedicated carry logic provides fast arithmetic carry capability for high-speed arithmetic functions. The Virtex supports two separate carry chains, one per CLB. The height of the carry chains is two bits per CLB. The arithmetic logic includes an XOR gate that allows a 1-bit full adder to be implemented within a Logic Cell (LC) [2].

Each Virtex CLB contains two 3-state drivers (BUFTs) that can drive on-chip buses. Each Virtex BUFT has an independent 3-state control pin and an independent input pin. Virtex FPGAs incorporate several large block RAM memories which are organized in columns. These RAMs provide dual ports with independent control signals for each port.

In addition to general purpose routing, Virtex FPGA also provide dedicated routing. Dedicated routing resources are provided for two classes of signal. a) Horizontal routing resources are provided for on-chip tristate buses. Four bus lines are provided per CLB row, permitting multiple buses within a row. b) two dedicated nets per CLB propagate carry signals vertically to the adjacent CLB.

XC2VP50 has 53136 logic cells (23616 slices), 11808 tristate buffers with two Power PC processors [2]. Total number of horizontal and vertical CLBs are shown in Figure 4.1.
Figure 4.1. The XC2VP50 structure.

4.4 RPM methodology

The algorithm provides a very regular structure from the implementation point of view as it has repetitions of basic processing units. So the idea here is that it would be very helpful to create a Xilinx Relationally Placed Macro (RPM) for the basic processing unit and then use these macros for all the PEs and integrate them on FPGA chip during Map, Place & Route process of FPGA design flow. This scheme is highly advantageous, as on one hand it saves considerable amount of effort for the tool and on other hand it saves lot of effort of the designer to optimize overall design. RPM methodology also places PEs of the systolic array in such a nice and controlled way that the designer does not really require to control individual components within a PE. Controlled placement of these modules is done according
to the signal flow in the data-path which further minimizes interconnects length and thus boosting performance considerably. The Xilinx Floor-planner is a GUI-based tool that allows one to view and make these RPMs through the MacroBuilder capability.

4.5 Design methodology

The very basic architecture of data path of motion estimator is the one shown in Figure 3.2 where all 256 processing elements are interconnected in a cascade fashion and data/control information flows from one PE to the next PE thus covering the entire systolic-array. The same Figure (3.2) is again shown here (Figure 4.2) while giving some more design details.

![Figure 4.2. Basic processing architecture.](image-url)

24
It is clear from the architecture that an arbiter is needed to provide proper SAD vector to the comparator at proper instant of time. As an implementation strategy we could either make use of tristate buffers with one data bus or we could make use of a huge 256:1 19bit multiplexer. The former choice seems to be ideal as it will provide simple hardware but each FPGA CLB only contains 2 tristate buffers so the space allocated for each MeProC does not have enough tristate buffers. Implementation of a huge 256:1 19bit multiplexer alone is also problematic because it will take plenty of on chip space and secondly it will slow down the design considerably. Thus implementation of the arbiter plays an important part in moulding the outline of the overall architecture. The problem is solved at different hierarchical levels as a combination of multiplexer and tristate solutions. These hierarchical stages are explained below. Another justification of these stages is the ease to have a flexible position of different modules over the die and yet attain desired performance.

The different hierarchical stages are as follows:

1. Basic Processing Element(PE)
2. PE2X8
3. Systolic Array Row (SARow)
4. MeProC (Motion Estimation Processor)

Designing at each of the above hierarchical step and then corresponding changes in the 256:1 (19 bit) multiplexer architecture will be explained in the following section. However a detail explanation of each and every component/signal will be provided in chapter 6.

4.6 Basic processing element

As discussed earlier the basic processing unit provides subtraction, absolute, and accumulation
operation on the input pixel streams. Figure 4.4 shows the structural block diagram. Pixel streams may arrive simultaneously from two parts of the search area, either from search upper or search lower area thereby an arbiter is needed at the input of PE to select either of the two streams depending on the particular PE. Control signals for the multiplexer comes from a central controller which is integrated with the MeProC module.

As shown the unit contains three computing entities: subtractor, absolute, and accumulator. The computing entities can be reduced to two by replacing 'absolute' determining unit with some logic around accumulator. This provides considerable reduction in area. Intelligent use of the most significant bit of the subtractor output is made for this purpose. The resultant architecture is shown in Figure 4.5.

We started our design of PE with 8 bits per pixel for each of the input pixel streams. The Output error of each PE is selected to consist of 16 bits keeping in view to provide enough precision in quantization and precision issues.
For 8 bits per pixel (BPP), the maximum difference is $2^{BPP} - 1$. Therefore:

$$
\max \left( \sum_{0}^{255} \left| X - X_A \right| \right) = \sum_{0}^{255} \max( |X - X_A| ) = \sum_{0}^{255} 2^{BPP} - 1 = 256 \times (2^{BPP} - 1) = 65280
$$

$$
65280 < 2^k - 1
$$

The creation of Xilinx RPM from the hardware of Figure 4.4 is the next step. The two main functions of the MacroBuilder are to create an RPM from a file and then use this RPM as a black box in a Xilinx Project. The RPM can be created either after NGDBuild or after PAR (Place And Route). The result of the RPM creation is an NGC file which is then instantiated as a 'black box' in the VHDL code. RPM is being created by manual placement of different components. The resulting RPM is shown in Figure 4.6.
Figure 4.6. RPM with 8 bits for input word length and 16 bit accumulator

The corresponding hardware elements are also shown in Figure 4.6. The first slice column contains flip flops for the reference pixel register. It also contains the input multiplexer. Some control flip flops are also part of the column of first slice. The second slice column contains the accumulator part.

Figure 4.6 shows that we were able to contain the entire hardware of a PE in a rectangular area of 5 vertical and 1 horizontal CLBs. In the first look one may notice, why RPM placement is not constrained to four vertical CLBs. Restricting the hardware to four vertical CLBs is not possible because of the following reasons:

1. Cutting one big carry chain is certainly not possible as it has to grow vertically.
2. Some flip flops were automatically forced to upper part of the slice and they use some routing from lower part of the slice. Hence space for one flip flop needs to be left otherwise it will not be possible for the tool to route the design at later stages, when all the RPMs are connected together.

There are also some other issues with RPM creation. Sometimes the placement order of flip-flop is very strange, for example, even if eight flip-flops are placed
vertically starting with bit0 at the lowest position and bit7 at upper most position, the tool adopts a different placement order of putting even bit numbers to the upper portion of the slice and odd numbered flip-flops to the lower part of the slice.

RPM of Figure 4.6 shows several empty resources. Leaving the resources empty does not seem to be a good choice as RPM will be repeated 256 times resulting in wastage of lot of important FPGA resources. An increase in the size of input pixel streams and output streams could be a good choice as 5 CLBs will be fully packed and increasing the length of pixel streams and accumulated errors will further increase precision of the data with less quantization noise. Now the length of the input stream is fixed to 10 and the length of the output error is fixed to be 19 bits.

As is explained in the previous chapter the 256:1 Multiplexer has to be taken in consideration during the design of individual units. Output error has 19 bits, feeding these entire 19 bits to a central multiplexer is not a good choice. We have used a combination of tristate Buffers and multiplexers. Each CLB of the Vertix FPGA has two tristate buffers which means that each RPM has 10 tri-state buffers so the idea here is to use these tristate buffers for the 10 least significant bits of error signal and use a multiplexer for the remaining 9 bits. Thus 9 bit multiplexer is also included as part of the RPM as shown in Figure 4.7. So the structure of the processing element is shown in Figure 4.8 (a) with corresponding generated RPM showing different parts of the design in Figure 4.8 (b).
Figure 4.8. a. Complete schematic diagram of PE and corresponding RPM.

Figure 4.8. b. Detail of different components within RPM.
An important point is that tristate buffers are not part of RPM. As the tools do not maintain proper relative location of tristate buffers after RPM creation, the same CLB tristate buffers will be used for the same 10 bits of error but these will not be part of the RPM. Instead they will be placed later on at a higher hierarchical level i.e. in module PE2X8.

4.6.1 New multiplexer architecture

After making several design changes to the basic processing unit PE, the new multiplexer architecture is being sketched in the Figure 4.9. This architecture is a combination of tristate buffers and multiplexers. By making the multiplexer a part of the RPM we have generated a compact structure for PE which then could be used to produce a rectangular structure for entire systolic array. An additional advantage of cascaded multiplexers is that they make the control easier.

Figure 4.9 Tristate buffers for 10 least significant bits of the error while other bits flow in the form of cascaded in the form of cascaded multiplexers.
As the target FPGA has 70 horizontal CLBs, which means 64 processing elements can occupy 64 horizontal CLBs in one row, i.e., one row will occupy a rectangular area of 64x5 CLBs. Similarly all 256 processing elements can be adjusted in four rows covering a rectangular area of 64x20 CLBs. The remaining 6x20 (CLB) area to the right most side could be used for placing multiplexers, comparator and other control logic. The overall architecture is shown in Figure 4.9.

4.7 PE2X8 (Group of 16 processing elements)

Looking at the architecture of Figure 4.9 it’s clear that such a long cascade of multiplexers is certainly not feasible from performance point of view. The performance measure is 20MHz if we only consider the signal flow through all 256 multiplexers. Boosting performance from 20MHz to 125MHz requires considerable further modifications in the

Figure 4.10. Insertion of pipelining after every eighth PE.
data path. Pipelining should be inserted regularly after certain number of RPMs. Pipelining after 32 PEs give speed of 70MHz. Pipelining after 16 Processing Elements give speed measure of 100MHz. Pipelining after 8 processing elements give rise to 136MHz, which is acceptable as we have constraint of 125MHz.

As is clear from Figure 4.10, such pipelining will create large latencies at the output which is not a good choice. As for some error lines we are using tristate buffers without any pipelining so mutual latencies can also be problematic. In order to maintain minimum latencies and yet achieve 125 MHz another architectural modification is inevitable as shown in Figure 4.11. It shows a group formation of 16 Processing Elements. The output of eight PE in each group of 8 is being registered and then fed to a 2:1 9 bit multiplexer.

Figure 4.11 The PE2X8 architecture

so there will be four such multiplexers per row for each of the four rows. Here we are facing two issues.

1) For each of the four rows, routing of the outputs of the four multiplexers to
another stage of 4:1 multiplexer is a big issue as it leads to very long lines. These long lines will no longer be straight as they will enter into different switching matrices before reaching the final multiplexer.

2) For each of the four PE2X8 modules in one row, placement of the extra two 9-bit registers and one 2:1 9 bit multiplexer is also an issue. Their placement could be very critical as 64 PEs are already capturing 64 horizontal CLBs in one row. This hardware should also not be placed away from the concerned group of 16 PEs, otherwise it could severely deteriorate performance.

Consequently, the two 9-bit registers, one 2:1 9 bit multiplexer and their control is placed in between two groups of eight processing elements which seems to be an ideal location as far as short interconnects are concerned. The resulting picture is shown in Figure 4.12.

![Figure 4.12](image)

Figure 4.12 Floorplanner snapshot of PE2X8 module with registers and multiplexer in the middle region.

From Figure 4.12 we observe that one extra 1x5 CLBs is inserted, hence, solving issue 1. These 5 vertical CLBs also contain 10 more tristate buffers which can be used to solve issue 2 stated above as tristate buffers can provide long straight horizontal buses with less timing issues and this will also eliminate the need for large multiplexers. The resulting new structure is shown in Figure 4.13 and its corresponding implementation is shown on floorplanner with Figure 4.14.
Figure 4.13. Insertion of tristate buffers after *error_mux*.

Figure 4.14. Corresponding floorplanner snapshot with tristate buffers in the middle region.
4.8) Systolic array row (SARow)

The next higher hierarchical level is the conception of one row of the systolic array by merging four PE2X8 modules together. Each PE2X8 module will consume a rectangular area of 17x5 CLBs. As cited previously, we have to put 64 processing elements in one row, which implies that four PE2X8 modules will be located in one row and subsequently these four will consume 68 horizontal CLBs in one row. Thus this one row will produce an error signal of 19 bits. Because the particular PE2X8 architecture pipelines nine most significant bits of the error one, clock cycle delay needs to be inserted in the path of the least significant ten bits of the error. Hence, a ten bit register is being placed. The new architecture is described in Figure 4.15 where SARow is shown consisting of for PE2X8 components.

![Figure 4.15](image)

Figure 4.15 Simplified architecture of SARow.

As a result of these modifications, the huge multiplexer 256:1 (19-bit) was reduced to 4:1 (19-bit) multiplexer by cascading four rows of processing elements. The structure is shown in Figure 4.16. This particular structure assumes 256 processing elements which are arranged in four rows where each row contains 64 Pes, but this may not be the case always. Sometimes there may be the need for a special structure where height is more than the width for the overall systolic array. For example if we require 8 rows with each row containing 32 PEs. This require the design of multiplexer with parametrized number of inputs where size of the multiplexer will grow with increasing number of rows.
Another scheme could be adopted by using the priority encoder shown in Figure 4.16, where each row uses one 2:1 (19-bit) multiplexer to select either the 'error' from current row or from the previous row. By making this multiplexer part of SARow we have made a very compact design for each SARow. Now we can generate as many rows as we want without worrying about complexity when designing the multiplexer.

4.9) Motion estimation processor (MeProC)

All 256 processing elements are arranged as a combination of 64x4 processing
elements. The module MeProC instantiates an SARow module four times thus generating four rows, each containing four PE2X8 modules. Four row multiplexers and comparator logic can be placed in a rectangular area of 2x20 CLBs to the right-most side of the FPGA as rest of the area has been taken up by four SARow modules. Placement of this logic at the end seems to be a better choice because we are expecting long straight horizontal lines of the error signal for each row. The resulting implementation from floorplanner window is shown in Figure 4.17.

Figure 4.17. Floorplanner snapshot of four SARow with multiplexer logic at the right most CLB

Hence the entire systolic array structure fits into a rectangular architecture of 70x20 CLBs. Proper registering of the signals is provided at input and output. As we know that there are two types of control logic, one is local and distributed within the SA and the other is generated by a global controller. Control elements are not constrained manually as they take only a small region around the rectangular region shown in Figure 4.17.
5. Further improvements in data path design

5.1 Overview

Bidirectional motion estimation contains two *MeEngines* and each *MeEngine* contains *MeDataTF* (motion estimation data transformer), *MeDataWr* (motion estimation data writer) and *CMC* (central memory controller) components for memory interface. Hence, enough space should be reserved on the target device for these memory interface components which results in very strict area constraints on each of *MeProC* modules which makes it difficult for the design to work at 125MHz.

Architectural evolution for the Figure 4.17 was carried out with the only assumption that the area contained by the MeProC should be as small and as regular as possible to accommodate other memory interface modules. CMCs and LMCs placement was not considered for the architecture of Figure 4.17. However, the two CMC modules will be placed towards the left and and right side of the chip. The rectangular area required by the two CMCs needs at least one vertical slice to both sides of the chip which means, that MeProC has to be placed in a rectangular area of 69x20 CLBs i.e. horizontal length is reduced by one CLB. Consequently, area constraints have gone harder and created several timing problems. Hence, the architecture of Figure 4.17 and its corresponding placement on FPGA was modified until we reached the target frequency of 125 MHz. The following section contains description of the different timing problems faced and their solutions.

5.2 Need for pipelining in comparator path

The critical path is shown in Figure 5.1. This path has a delay of 15 ns which is much more than the required 8 ns (125 MHz). The ten least significant bits of error signals for each row has a longer combinational path. As a result the critical path extends from the first row to the comparator covering all the priority encoders shown with red line. The comparator is part of the critical path. As the comparator
contain heavy combinational logic, it is obvious that pipelining at the input of comparator may solve the issues for us. This technique saves 4 ns (as shown in Figure 5.2) but still we have a margin of 3 ns. Accordingly, further considerations were made in order to get an estimate of delays caused by different components including long interconnects.

Figure 5.1. Critical path of MeProC.

Figure 5.2. Insertion of pipelining into comparator path.
5.3 Long straight horizontal error buses

The error signal coming from each SARow contains tristate buses. These tristate buses form an important part of the critical path. During the design and placement of tristate buffers it was assumed that, according to Xilinx documentation, long straight horizontal buses would be generated. If these buses are straight then they should not create any timing problem which does not seem to be the case. So there was a need to see whether the tool has really generated long horizontal straight lines for error signals or not. An observation in FPGA Editor showed that this is not the case. Instead the tristate buses enter the switching matrices at a number of locations.

Documentation of the Xilinx FPGA informs that tristate buffers are placed in an alternate way. Consider the following tristate routing as described in the Xilinx manual.

![Dedicated horizontal routing for tristate buffers.](image)

According to this routing methodology, we have to use alternative placement for tristate buffers on alternate RPMs. Explanation of this scheme is clear from Figure 5.4.

This alternate placement was made for all the processing elements in the design but it was noticed that even this scheme does not produce long straight horizontal buses and instead lines enter switching matrices regularly creating larger delays. However if we do not follow the alternate policy, we obtain much longer and straight horizontal buses except for a few locations in the middle of SARow. Later
on it was observed that Xilinx has itself placed these tristate buffers on alternate positions for every alternate CLB. The Xilinx documentation of Figure 5.3 is wrong and has lead us into error because we don't need to manually lock them to alternate positions.

Figure 5.4 Explanation of alternative buffers adopted according to Figure 5.3

Figure 4.17 shows that $row\_mux$ (multiplexer to select error of each $SARow$) for each row was placed to the right most end of each row. The rationale behind this scheme was that right-most end of the straight tristate buffer will be connected to the row_mux giving us lowest delay. However this is not the scenario when we concentrated on the long nets in the FPGA Editor window. Due to very strange routing methodology adopted by the Xilinx tool, a straight line is created but $row\_mux$ is connected to the middle of a long tristate line. Interconnect from the middle of the line enters different switching matrices before entering into row_mux which considerably deteriorates performance and creates very long delays. This routing methodology forced us to bring all row multiplexers and comparators to the middle of a row.

The space of one CLB was then created for $row\_mux$ and the comparator in the middle of the SARow. But still we are left with 10 ns delay (100 MHz) and we
have to modify the data path to a get performance of 125 MHz. The next modification is to bring the register which is present before the comparator to the input of all row multiplexers. This concept is very easy to think but its very difficult to implement as we have only one vertical CLB left for placing all the row multiplexers, comparator, two counters (purpose will be explained in next chapter), 19-bit output SAD vector (sad_reg) and now 19-bit registers (row_regs) for each row which alone will take total 19x4 flip-flops. This is certainly not an easy solution to implement as the comparator, row_reg, row_mux, and sad_regs can not be moved away from each other. These modules shown in Figure 5.5 should be placed closer to each other in a rectangular area of 1x20 CLBs (for all SARows) to get shorter interconnects and this is the new task from hereafter.

Figure 5.5. Structure of MeProC showing only row_regs, row_mux, comparator, and sad_regs
5.4 LUT approach for shift register implementation

Each row (i.e one SARow) takes 1x5 CLBs. These five CLBs have two columns of slices. Each column contains 10 slices. Each slice contains two flip-flops and two LUTs. Thus, one column of slices will enclose 20 flip-flops and 20 LUTs.

As the 19-bit error signal for each row comprises two error streams, i.e., the least significant 10 bits (LSB_error) and remaining 9 bits (MSB_error). The two streams come from two different logic paths and as stated in previous chapters, MSB_error has one extra register placed in its path as compared to LSB_error. Moving the pipeline register from the input of the comparator to the input of each row_mux results in one extra register to be placed in the path of MSB_error (MSB_error_reg) and two extra registers to be placed in the path of LSB_error (LSB_error_reg) in order to equalize the delay from the two paths. This results in 29 registers and all these registers should be placed in one slice column which only contains 20 registers.

Figure 5.6 shows details on the block diagram and the corresponding empty floorplanner snapshot showing only the logic where row_regs and row_mux will be placed.

Fortunately, Xilinx provides a design element SRL16 which could do a lot for us. SRL16 is a shift register look up table (LUT) which allows us to implement a shift register of size up to 16 using only one LUT in a slice.
The inputs A3, A2, A1, and A0 select the output length of the shift register. The following listing can be used to infer this component into VHDL code.

```vhdl
component SRL16
    -- synthesis translate_off
    generic (
        INIT: bit_value:= X"0001"
    );
    -- synthesis translate_on
    port ( Q : out STD_ULOGIC;
        A0 : in STD_ULOGIC;
        A1 : in STD_ULOGIC;
        A2 : in STD_ULOGIC;
        A3 : in STD_ULOGIC;
        CLK : in STD_ULOGIC;
        D : in STD_ULOGIC);
end component SRL16;

Because of the use of SRL16 to implement shift registers of LSB_error, we can now implement the desired logic in one column. The idea is to use one slice very
efficiently such that each slice should implement two-bit shift register, one 2:1 multiplexer (part of Figure 5.7. Architecture of 'shifmuxreg' and use of SRL16 component.

row_mux) and one bit register for MSB_error_reg. In this way 10 slices will enclose all the desired hardware. One slice is shown in Figure 5.7. For this purpose, a utility shifmux_reg has been created and is duplicated 10 times to enclose all the desired hardware.

5.5 Implementation of row_mux

Module 'shifmuxreg' also contains one 2:1 1-bit mux. Since row_mux is a 2:1 19 bit mux, it needs 19 such small units. Unfortunately, we don’t have enough LUTs in one column to accommodate a 19-bit multiplexer. So we have no other choice but to use the second column of the CLB to implement the remaining 9 bits. This will slightly complicate the routing resulting in longer interconnects but there is no other possibility of placement.
5.6 Need for extra pipelining and extra slice

As is clear from Figure 5.7 the rectangular area of 1x20 CLBs is not enough to place the comparator and sad_regs. One more vertical slice is needed because the work that has already been done so far is on its extreme limits as far as placement is concerned. A good thing from the CMC placement point of view is that we could shift the two CMCs in such way that we can get one vertical slice on either side of the motion estimation block. For upper motion estimation block. CMC will take one vertical slice from left and for lower ME it will take one from the right-most side. After getting an extra vertical slice we have the luxury to constrain the comparator just adjacent to the last row multiplexer. Every thing is very nicely aligned and placed but still the tool gives an error of not meeting the constraint of 8 ns, instead it is reporting a time period of 8.1 ns. Still we have to perform some architectural modification to get rid of the 0.1 ns timing issue. We have one fortunate point here that there is space left for one vertical 19-bit register before the comparator. We can use this space to create an extra pipelining which gives us 7.9 ns. Hence, the performance Figure of 125MHz for each of the motion estimation block is reached successfully at the end.

Figure 5.8. Placement of extra pipelining and comparator in the last SARow.
5.7 Search upper and lower IO register placement

Each processing element in the systolic array of 256 PEs needs search upper and search lower pixel streams. So far a 10 bit input register for both search upper and search lower stream are part of MeProC and then this stream is provided to all the PEs. This distribution to all the PEs results in very long and irregular interconnects which gives a timing problem of 12 ns. Different modifications were made in the architecture in order to get desired frequency figure.
1) Initially, `search_upper_reg` and `search_lower_reg` were not constrained to any location. However, if we place these registers in the middle of the rectangular area (where they could use the empty slots) then we have the possibility of solving the timing issues. Unfortunately, this did not work.

2) Use separate `search_upper_reg` and `search_lower_reg` for each of the four SARows. Unfortunately, this again did not work and still we found problems of 12ns.

3) Place both the registers in the middle of each row so that we may have short interconnects but still these interconnects are not short enough and give timing problems.

4) Take these registers part of the PE2X8 module but do not restrict their placement. This technique worked and now we have `search_upper_reg` and `search_lower_reg` as part of each PE2X8 module. This solved the timing problem.

### 5.8 Reverse placement for odd rows

As is mentioned in chapter 2 and chapter 4, data and control information flows from first processing element to the last processing element in a systolic-array of 256 elements such that all the PEs process the data together. The four SARows are placed on top of each other where each SARow module is further divided into four PE2X8 modules, i.e., from PE2X8_0 to PE2X8_3.

In the single SARow we place first PE2X8 component to the left-most side and last PE2X8 is placed to the right-most side. So signals from the last PE2X8 module travel to the first PE2X8 module in the second row (row1) which is present to the left most side as shown in Figure 5.10. These long interconnect are problematic from the timing point of view. Therefore, placement methodology adopts an automated procedure to place PE2X8 modules of the odd rows in reverse order starting with the first PE2X8 module to the right-most side as shown in Figure 5.10.
Within each PE2X8 module the first element is placed to the left-most side and information flows from left to right as shown in Figure 5.11.a. Signals from last PE2X8 module in SARow0 travel to the left most PE of the first PE2X8 in the second row. Even this long interconnect is problematic therefore we have to reverse the PE placement order even within each PE2X8. The resulting structure is also shown in Figure 5.11. These modifications ensure no timing issue.
5.9 Placement of upper_lower_sel_flag

Each processing element (PE) receives search upper and search lower pixel streams. Within each PE, a multiplexer is required to select the correct stream depending upon the location of PEs within the systolic array. For any particular size of the reference block, a certain group of PEs always need the same control signal. This signal comes from a flag (upper_lower_sel_flag) present in the global controller. Feeding several processing elements present in different SARows with one flag produces long interconnects which cause timing problems.

This problem is solved by duplicating the flag to the concerned PE. Hence the same control signal is generated but each processing elements also contain one flip flop to store it before providing this to the multiplexer for proper control. This flag is in fact part of the RPM of the processing element.
6. Detail Explanation Including Controller

6.1 Overview

MeEngine contains a highly optimized block of MeProC which implements not only the systolic array but also the complex control associated with it. As is described in chapter 5, the entire hardware is composed of four hierarchical levels using bottom-top methodology. Each level is associated with its data path and its local control in addition to a global control. Generation of local control in modules is important as it provides a very compact design which is highly beneficial from performance point of view because it has local interconnects. Previous chapters provide information about the evolution of the data path design.

![Figure 6.1 Picture showing different hierarchical levels.](image)

and only a brief explanation of major components in different modules and their placement on the FPGA. Four hierarchical levels are shown in Figure 6.1, where
each block receives certain control and data information from the top level module while only MeProC has interface to the outside world for input and output. In addition to the data and control information, placement and data generics also flow from top to bottom components.

### 6.2 Basic processing element (PE)

The PE is the building block of the systolic array. It performs three operations on the input pixel stream. Operations include subtraction, absolute and accumulation. As is explained in previous chapters, the final structure of the PE has evolved as a result of different architectural modifications. One of the important modifications in term of area is not to use a dedicated absolute unit. Instead logic around the accumulator is developed to provide absolute of the subtraction result. A schematic of the processing element is shown in Figure 6.3.

The first operation to be performed on the inputs is 'Subtraction'. Each PE, depending upon its location in the systolic array, needs either search upper stream or search lower stream. Thus an input multiplexer is needed which is controlled by 'i_up_down' signal. This signal is provided by the global controller to each processing element and prior to feeding this signal to the multiplexer it is registered in order to solve the timing problems.

The input data is signed and has size of 10 bits. The result of the subtraction operation is 11 bit wide which demands 1-bit sign extension (SE) for the inputs. The output of the subtracter is given to the adder/subtracter module of the accumulator. The accumulator has a size of 19 bits which makes it necessary to perform another sign extension after the input subtracter.
The most significant bit of the subtracter result determines the type of operation performed by the adder/subtractor module. Subtraction is performed when input to the subtracter has a negative result. Otherwise addition is performed. Thus, accumulation and absolute operations are performed at the same time without the need for a separate absolute unit.

Initialization of the accumulator register is very important whenever a new reference block arrives at the input of the processing element. We have used a scheme where the absolute result of the first pixels of the new block is stored in the accumulator without involving the previous accumulator value. For this purpose an accumulator initializer signal ($i_{\text{init\_acc}}$) along with a multiplexer is used. This multiplexer is present at one of the inputs of the adder/subtractor module. Thus, when an active $i_{\text{init\_acc}}$ (active low) is received, zeros are selected from the multiplexer such that operation '0 – error' is performed if MSB (most significant bit) of the input_error is '1'. Otherwise the operation '0 + error' is evaluated.

The ten least significant bits of the accumulator error are taken out as 'o_error_tbuf' and connected to the tristate buffers which are part of next higher level module. Nine least significant bits of the 'error' signal are given to the multiplexer whose
function is to pass either the error signal coming from the previously cascaded PE or the error from the current PE. To control this arbitration, the signal $i_{err\_reg}$ is provided by the global controller and flows through all the PEs in a cascaded fashion. This signal is in fact a 'valid' signal indicating the availability of the valid error from each PE. This control signal is also used to trigger the tristate buffers at the proper time in the $PE2X8$ module.

Figure 6.3. Block diagram of a PE.

The reference pixel stream is registered in each processing element and it uses an 'enable signal ($i_{strobe}$) which ensures the flow of only valid reference pixels through the systolic array.
Thus we have two control signals and one data signal which are registered in each processing element and flow in the form of shift registers from PE to PE.

6.3 PE2X8 (Two groups of eight (2x8) processing elements)

The next higher level is the PE2X8, its symbol with input output signals is shown in Figure 6.4. Structural diagram of the PE2X8 module is shown in Figure 6.5. Is clear from the Figure that the PE2X8 consists of four parts:

1. Cascaded 16 processing elements
2. Tristate buffers
3. Pipelining
4. Local control

Figure 6.4. Symbol of PE2X8.
6.3.1 Cascaded 16 processing elements.

This module contains 16 processing elements, which are divided into two groups, each containing 8 PEs. The reason for this special architecture is explained in previous chapters. This division into 8 PEs allows proper pipelining after eighth PE. The eight PEs within one group are connected in a cascaded fashion where the reference pixel stream and necessary control information flows from the first PE to the last PE in the group and then it is fed to the second group and after this information has reached the last PE in second group. It is then ready to pass to the next stage. Registers for search upper and search lower streams are brought to the PE2X8 module to get rid of timing problems associated with one global register feeding data to all PEs.

Figure 6.5. Schematic diagram of the PE2X8.
6.3.2 Tristate buffers

Tristate buffers for each processing element are not part of the RPM developed from the basic processing element, as the tool does not maintain correct relative position of different TBUFs (tristate buffers) after RPM generation. Therefore, the TBUFs are integrated with the RPM at this level. 'o_error_tbuf' signal from each PE is fed to 10 tristate buffers. The error enable signal i_err_reg within each PE is used to change the state of the corresponding 10 tristate buffers thus emitting the right error at the right time to the tristate bus. The tristate buffers are locked to specific positions on FPGA within each RPM.

6.3.3 Pipelining

The nine least significant bits of the error signal called o_error_mux within each PE flow through eight cascaded multiplexers. This requires pipelining after eighth PE in each group in the form of two registers called first8_reg and second8_reg. This ensures that the delay of the path is less than 8 ns. In order to generate motion vectors after every N x N cycles, the 'comparator' needs an error signal sequentially starting from the first PE to the last PE in the systolic-array. Thereby, there is a need for a multiplexer error_mux present in each PE2X8 module. The use of another set of 9 tri-state buffers is made at the output of error_mux. In this way we are generating more tristate buses, making the total number equal to 19.

6.3.4 Local control

Within each PE every component/signal is well controlled as described in the previous section. In the PE2X8 module we have to provide necessary control to one multiplexer (error_mux) and 9 tristate buffers (tbufs_for_error_mux). One option is to generate control from the global controller but it will result in extra cost in term of long slow nets. Hence we have to generate control locally for these two types of elements.

The multiplexer needs toggling control such that when the control signal is 'zero' the multiplexer will pass the error from any of the PE within the first group, and when it is 'one' selection will be made for the second group of 8 PEs. 'error_mux_sel_flag' is used for controlling the multiplexer and this flag should be
set and reset at the right time in order to ensure correct data flow. As we discussed in the previous section, within each PE we have an 'error valid' signal, i.e., \textit{i}\_\textit{err\_reg} flowing through every PE in the form of the shift register. This signal is also used to provide correct arbitration for \textit{error\_mux}. Arrival of this \textit{i}\_\textit{err\_reg} (which is active low) at the first PE of the first group is an indication that selection for the first group should be made. Similarly, arrival of this signal at the second group provides correct selection for the second group. These two signals are therefore used for setting \textit{error\_mux\_sel\_flag}. \textit{tbufs\_for\_error\_mux} are also controlled in the same fashion by flag \textit{error\_mux\_buff\_flag}.

6.4 SARow (Systolic array row)

Figure 6.6 shows the symbol of SARow with input and output signals. A structural block diagram of the module SARow is shown in Figure 6.7 which suggests that there are three main parts of the SARow.

1. Combination of PE2X8 components
2. Pipelining
3. Local Control

6.4.1 Combination of PE2X8 components

SARow forms one row of the systolic array to be placed on chip. Depending upon the overall desired shape of the placement over the chip, it contains certain number of PE2X8 components. In case of 256 PEs to be placed in four rows, we need four PE2X8 modules to be part of every SARow. All the PE2X8 components are connected to ensure correct flow of the reference pixel stream and associated control information from all the PE within SARow.
Figure 6.6. Symbol of SARow.

The SARow contains desired number of PE2X8 modules. If the overall structure contains 256 PEs with four rows then each SARow will contain four PE2X8 modules. Each PE2X8 is fed with search upper and lower stream and corresponding 'strobe' signals indicating the validity of these streams. In addition, each PE2X8 also needs 16 signals for each PE for proper upper or lower stream. We also see another input signal \textit{i\_error\_mux} to each PE2X8 module. This signal is fed to the first PE of each group within each PE2X8, which does not take cascaded multiplexer output from the previous PE. This signal can also be initialized within each PE2X8.

6.4.2 Pipelining

Each PE2X8 provides the two streams of error, one is \textit{o\_error\_tbuf} (10 bits) and other is \textit{o\_error\_mux} (9 bits). Each of the streams originates from PEs in two
different paths. The path of \textit{o\_error\_mux} contains one extra register \textit{(first8\_reg/second8\_reg)}. Therefore, the two streams can not be concatenated without providing one clock cycle delay to the \textit{o\_error\_tbuf} stream. Thereby the need for one register in its path. In addition to this register we have to provide another register to get rid of timing problems as discussed in previous chapters.

6.4.2 Local control

Because of the use of the priority encoding scheme described in chapter 4, each SARow contains one multiplexer to select either the error signal from the current row or the one from the previous row. Implementation of pipelining and the row multiplexer is made using \textit{shifmuxreg} module described in chapter 5.

Control of the row multiplexer is generated locally from \textit{line\_mux\_sel\_flag} which is set and reset by the same error enable signal \textit{(i\_err\_reg)} flowing through every PE in cascaded fashion. As soon as the \textit{i\_err\_reg} signal enters into the first PE of the first PE2X8 in each SARow control of the row multiplexer is shifted for the current row and as soon as this signal reaches the last PE of the last PE2X8 in each row, control of the row multiplexer is shifted to the previous SARow. Because of extra pipelining in the path of error in each SARow, an extra delay should be also be provided in the path of control signal belonging to to the row multiplexer.
6.5 MeProC (Motion estimation processor)

Motion estimation processor (MeProC) contains the entire systolic array of processing elements with special placement constraints to lock all the components on specific FPGA positions. In
addition to the systolic array which forms a major part of the data path, MeProC contains global control for the modules present at different hierarchical levels. Figure 6.8 shows the structure of the MeProC. MeProC contains two main blocks:

1. MeDP (Motion estimation data path)
2. MeGC (Motion estimation global control)

### 6.5.1 Motion estimation data path (MeDP)

The following Figure 6.9 shows the schematic of the entire data path of MeProC.
Figure 6.9. Data path schematic.
The data path is further divided into following blocks

1. Systolic array (DP_SA) of 256 processing elements
2. Systolic array Output interface (DP_SA_OP)
3. Comparator (DP_C)
4. Motion vectors (DP_MV)

### 6.5.1.1 Data path systolic array (DP_SA)

The systolic-array possessed desired number of SARow modules. Depending the upon overall placement structure it may contain either four or eight rows. These SARows are connected such that the reference stream and control information flows from the first row to the last row. Each SARow outputs an error which flows through all the multiplexers until it reaches the comparator.

### 6.5.1.2 DP_SA_OP (Systolic array output)

The last SARow outputs four signals. 1) o_error_of_line 2) o_err_reg 3) o_init_acc 4) o_ref_data.

1. The signal o_error_of_line is the error of all the SARows. Before feeding this signal to the comparator we need to use a register in order to avoid the timing problem described in chapter 5. As a result of this the path of the error originating from the PE to the comparator has a delay of three clock cycles.

2. The error enable signal i_err_reg within each PE indicate the validity of the error outputted by the processing element. Starting from the first PE, this signal reaches the last PE in 256 cycles. In This way the particular signal can also be used to show that all the accumulated errors are being read by the comparator and now valid motion vectors are available from the SAD and motion vector registers. The error has a latency of two extra cycles as compared to the o_err_reg. Therefore there is a need for delaying this signal by two cycles.

3. o_ref_data & o_init_acc are also emitted from the last SARow. They could be taken out for debugging purposes in order to check the correct data flow through all the PEs or to cascade to other motion estimation block to increase the search area.
6.5.1.3 DP_C (Comparator)

This block consists of a comparator and an accumulator register (\textit{min\_error}). These two together form an entity which is used to accumulate SAD vectors and necessary control generation for outputting motion vectors. The \textit{Comparator} performs comparison on the input signals. One is the \textit{error\_of\_all\_rows} and the other is the accumulated \textit{min\_error}. The output of the \textit{comparator} is an enable signal \textit{comp\_enab} for register \textit{min\_error}. This control is also used to provide a proper enable signal for SAD\_regs and MV\_regs.

\textit{min\_error} stores error\_of\_all\_rows whenever it receives an active enable signal from the comparator. Initialization of the \textit{min\_error} with maximum values is important whenever a new reference block enters the systolic array.

6.5.1.4 DP_MV (Motion vectors)

Three types of motion vectors are generated by the MeProC.

1. MV\_SAD (SAD motion vector)
2. MV\_X (horizontal motion vector)
3. MV\_Y (vertical motion vector)

MV\_SAD provides the minimum error for the desired comparison between the reference block and corresponding search area. It receives error\_of\_all\_rows at its input and the enable signal, that is taken from the comparator.

MV\_X and MV\_Y are the corresponding direction vector in x and y direction respectively. These vectors are obtained by two pipeline stages for the corresponding motion vector counters (MVC). The purpose for pipelining is to synchronize these with MV\_SAD vector which is already pipelined.

Because of the severe area constraints, MV\_X, MV\_Y and associated shift registers are implemented using SRL16 components and are locked to particular locations on FPGA.
6.5.2 MeGC (Motion estimation global controller)

Global controller provides control to the entire systolic array and is further divided into sub blocks which are described below.

1. ME Starter (MS)
2. PE_Accumulator Initializer (PE_AI)
3. PE_Error Enabler (PE_EE)
4. PE_Upper_lower Controller (PE_ULC)
5. Motion Vector Counters (MVC)
6. Top Left Boundary Accumulator Initializer (TL_AI)
7. Top left Boundary pixel Redirector (TL_PR)
8. Boundary Detector (BD)
9. Reference Stream Reader (R_RS)
10. Search Upper Stream Reader (R_SU)
11. Search Lower Stream Reader (R_SL)
12. Pixel Stream Disabler (R_SD)
6.5.2.1 ME Starter (MS)

Processing of the very first frame starts with an external signal $i_{\text{init me}}$ after that processing of every next frame is initiated by a locally generated signal $\text{init me}_{\text{local}}$. The input signal $i_{\text{init me}}$ is registered twice in order to detect its positive edge which is important for certain components.

$\text{init me}_{\text{local}}$ signal is generated from reference block counter. When $\text{bd ref block counter}$ reaches $\text{image size}$, one frame is completed and counter has to start again from zero, but before reaching zero it stays at value of $\text{image size} + 1$.

---

Figure 6.10. Global controller.
for one clock cycle during which locally generated signal is zero indicating the start of new frame.

6.5.2.2 Processing element accumulator initializer & error enable (PE_AI & PE_EE)

Each processing element requires one control signal (init_acc) for initialization of the accumulator whenever processing of the new reference block starts. Use of this control signal within each PE was described in its respective section.

The error enable signal (err_reg) is another important signal used within each processing element and flows through all the PEs in cascaded fashion. This signal is also generated in the global controller and is fed to the systolic array.

Timing of both err_reg and init_acc is the same and hence they are generated in the same way. The use of motion vector counters is made for this purpose. These signals are generated whenever a new reference block starts, i.e., start of both motion vector counters.

6.5.2.3 Processing element search upper/lower stream selection PE_ULC

Within each processing element there is an input multiplexer which selects either the search upper pixel or search lower pixel stream. As was mentioned in chapter 3, there is a regularity for generation of this signal. For example in the Figure 3.5 PE1, PE9, ..., PE57 always need search upper pixel stream while PE2, PE10, .... PE58 need the first seven pixels of the column from search upper and remaining pixels from search lower area. Similarly PE8, PE16, ..., PE64 need the first pixel from search upper while rest of the pixels from lower area. This way each PE in each row in the Figure 3.5 always needs the same signal, hence for a reference block of 8x8 we generate eight signals and these signals are fed to the entire systolic array. For a reference block size of 16x16, 16 such pulses (upper_lower_sel) are generated and fed to the entire systolic array.

These signals are generated with the help of a combination of one counter, i.e., counter_upper and a set of shifted pulses. The signal strobe_reg acts as an enable signal for the counter which counts from zero to maximum value of
REF_HEIGHT-1. When counter reaches the final value, a pulse is generated. The pulse acts as upper_lower_sel for all the processing elements in the last row (PE8, PE16, ...., PE64). This ‘pulse’ is then fed to certain combination of shift registers to act as upper_lower_sel for all the other processing elements present in the remaining rows.

6.5.2.4 Processing element motion vector counters (PE_MVC)

Motion vectors are generated by two counter horiz_counter & vert_counter. These counters are also used by the global controller to generate some other required control signals thereby avoiding the use of similar other counters.

horiz_counter provides motion vector in x direction. Its value ranges from -REF_HEIGHT to REF_HEIGHT-1. At the top left boundary it starts with zero because we only take the second half of the first reference block. It increments when the vertical counter reaches its end value.

vert_counter counter provides a motion vectors in y direction. Its value ranges from -(REF_HEIGHT) to REF_HEIGHT-1. It always starts with -REF_HEIGHT and it increments with the reference strobe signal.

6.5.2.5 Top left accumulator initializer & data redirector TL_AI & TL_DR

Each Processing Element needs an accumulator initializer signal whenever a new reference block enters into the systolic array. As described in chapter 3, for top-left boundary, the pixel stream is fed to PE129 (for 16x16 block) and PE33 for (8x8 block) so during this phase all the input pixel stream and control signals are directed to PE129 instead of PE1. For this purpose, a redirector signal is needed indicating the top-left part of the image (ref_sel). As we start feeding pixel streams to PE129, this PE should also get an initializing signal for the accumulators. Hence, a separate signal is generated for this purpose and is only fed at the top-left boundary.

ref_sel (active high) is initialized before the start of a new reference image. It is enabled as soon as we have the positive edge on start_me_reg. This is the time when we have the first reference pixel on the data bus. ref_sel remains '1' until the
start of the second reference block.

The signal `init_acc_topleft` is generated when the positive edge is detected at `i_strobe` signal which requires the `i_strobe_reg` signal to remain '1' thereafter during one entire image (after the first positive edge).

### 6.5.2.6 Boundary detector (BD)

Information about top-left, top and bottom boundary is important, as it is used to generate certain control signals. The boundary detector consists of one main counter `bd_ref_block_counter` and two flags `top_boundary` & `bottom_boundary`.

The counter `bd_ref_block_counter` starts from zero and goes to the total number of reference blocks in the reference image (size of each reference block is `REF_HEIGHT x REF_HEIGHT`). It increments when both motion vector counters reach their maximum value. This counter stays at its maximum value i.e. `image_size+1` for only one cycle, the time enough to initialize MeProC to start searching a new image.

The flag `top_boundary` indicates that MeDP is processing the top boundary of the images. Thus `top_boundary` stays high as long as `bd_ref_block_counter` is less than `NR_IMAGE_HORIZ_BLOCKS` (as counter starts with zero). The timing of low transition on this signal is also important as we have to ‘lower’ the signal only when we start fetching the second row of reference blocks in search image.

Flag `bottom_boundary` indicates that MeDP is in the processing of fetching the lower boundary of the images. Logic is very simple as when `top_boundary` is high, `bottom_boundary` should be initialized and it becomes 'high' when the last row of the search image is being fetched and processed. Here the timing for a low to high transition is very important.

### 6.5.2.7 Read stream (RS)

This block is dedicated for providing proper reading interface with the LMC. Read enable signals are thus generated for the desired stream. In response to the enable signals, LMC provides data and its associated strobe signals.
6.5.2.8 Reference stream reader (R_RS)

$o_{\text{ref\_data\_enable}}$ is generated whenever $start\_me$ is '1' and $disable\_data\_fetching$ is '0'. In response to this, $ref\_data$ is received along with its strobe signal. $i_{\text{ref\_data}}$ is registered with strobe signal. $ref\_data$ is then provided to the systolic array. $strobe\_reg$ is also used by the global controller to generate other important control signals described above.

6.5.2.9 Search upper stream reader (R_SU)

Generation of $o_{\text{search\_upper\_enable}}$ signal is different for top boundary and for rest of the image. For top boundary, search area extends to invalid pixel locations thereby disabling the signal. For this purpose use of $vert\_counter$ is made to generate the desired sequence at the top boundary. The strobe and data signals are fed to each PE2X8, as associated registers are present in each PE2X8.

6.5.2.10 Search lower stream reader (R_SL)

Generation of $o_{\text{search\_lower\_enable}}$ signal is different for bottom boundary and for rest of the image. For bottom boundary, the search area extends to invalid pixel locations, so $o_{\text{search\_lower\_enable}}$ should not be generated for those locations. $search\_lower\_counter$ is used to generate the enable signal for bottom boundary.

6.5.2.11 pixel stream disabler (R_SD)

The image areas corresponding to the reference, search upper and search lower streams finish at different instants of time. The reference stream is finished first, then search upper and then search lower. As soon as these streams are finished for each frame, the corresponding enable signals should also be stopped. For this purpose the block of $R\_SD$ is used which generates 'disable' signals for each stream which are then used by the $RS$ block.
7. Flexible placement over FPGA

7.1 Overview

The systolic-array forms a major part of the data path (MeDP) of MeEngine as far as placement is concerned. Flexible placement of MeDP is a requirement of the thesis. Use of relative-lock constraints is made in order to flexibly place various components over the chip.

Placement is flexible in terms of:

1. Desired number of rows of PEs in systolic array
2. Desired size of reference block
3. Desired size of the image to be processed
4. Desired rectangular/Square placement shape of MeProC
5. Starting RPM/TBUFs with either odd location or even horizontal location
6. Alternate reverse placement for odd rows to meet timing specifications

In this way only few placement variables at MeProC level are used to change the entire placement structure. This chapter will explain entire automated placement methodology which is adopted at each hierarchical level to place different resources on desired sites of the target device.

Components are constrained to particular sites using two types of locking commands: a) Logic is constrained within certain areas by mentioning the rectangular area using LOC commands b) components related to particular block at the same hierarchical level are placed relative to each other using Relative LOC (RLOC) constraints, and then RLOC_ORIGIN is used to provide bottom left-most location for the block at next higher hierarchical level.

7.2 Placement methodology for PE2X8

As we know that the RPM generated for one processing element is enclosed within one horizontal CLB and five vertical CLBs (2x10 Slices). Module PE2X8 contains 16 RPMs.
corresponding to 16 processing elements connected and placed in special fashion described in previous chapters. The ten least significant bits of the error signal coming out from each PE are connected to the 10 tristate buffers, which are locked to the tristate buffers captured by each RPM. All RPMs have the same height so their placement varies only in horizontal direction.

Each PE2X module receives its bottom-left position from SARow module. This position is described in terms of $rloc_{origin\_x}$ (giving horizontal coordinate) and $rloc_{origin\_y}$ (describing vertical coordinate). The module adopts different placement methodologies for even and odd starting location. The following section describes the details.

### 7.2.1 Placement starting with even horizontal location

We now describe placement schemes for RPMs and tristate buffers.

#### 7.2.1.1 Placement of first 8 PEs

One PE and its 10 tristate buffers are constrained using the same coordinates. First of all, placement is carried out for the first 8 RPMs. With the height of 5 CLBs, $y$ coordinates for each RPM varies from $rloc_{origin\_y}$ to $rloc_{origin\_y} + 9$ while $x$ coordinates for each RPM takes two values such that the first RPM starts with $rloc_{origin\_x}$ and 8th RPM starts at $rloc_{origin\_x} + 14$. 
PE placement:

\[
X = rloc\_origin\_x + (i \times 2) \quad \text{(for } i=0 \text{ to } 7) \\
Y = rloc\_origin\_y
\]

Each set of 10 tristate buffers is also placed with following \textit{RLOC ORIGIN} coordinates

\[
rloc\_origin\_x + (i \times 2) \quad \text{(for } i=0 \text{ to } 7) \\
rloc\_origin\_y
\]

### 7.2.1.2 Placement of second 8 PEs

After placing the first eight PEs with the order described above, a jump of 1 CLB is made to start placing the next set of eight PEs. This 'mid-region' is required as we have to provide enough space for additional hardware pertaining to PE2X8 module. The second set of eight RPMs is placed in the same manner but a jump of '2' is required for each horizontal location as shown in Figure 7.1.

PE placement:

\[
X = rloc\_origin\_x + 2 + (i \times 2) \quad \text{(for } i = 8 \text{ to } 15) \\
Y = rloc\_origin\_y
\]

Tristate Buffer Placement:

\[
X = rloc\_origin\_x + 2 + (i \times 2) \quad \text{(for } i = 8 \text{ to } 15) \\
Y = rloc\_origin\_y
\]

### 7.2.2 Placement start with an odd horizontal location

Each CLB contains tristate buffer parts of even slices. The methodology described above places the tristate buffers and RPMs to the same horizontal coordinates. However when RPMs are locked to odd slices, the x-coordinate for tristate buffers should differ by '1'.

PE placement:

\[
X = rloc\_origin\_x + (i \times 2) \quad \text{(for } i=0 \text{ to } 7) \\
Y = rloc\_origin\_y
\]
Set of 10 tristate buffers is locked to a different location

\[
X = rloc\_origin\_x + (i*2) + 1 \quad (\text{for } i=0 \text{ to } 7) \\
Y = rloc\_origin\_y
\]

To make it generalized, module PE2X8 receives a generic ‘bufloc’ from SARow. When this signal is ‘0’ it indicates that \( rloc\_x\_origin \) is an even location while value ‘1’ informs that \( rloc\_x\_origin \) is an odd location. Now the placement variables look like the following.

PE placement:

\[
X = rloc\_origin\_x + (i*2) \quad (\text{for } i=0 \text{ to } 7) \\
Y = rloc\_origin\_y
\]

Tristate Buffer Placement:

\[
X = rloc\_origin\_x + (i*2) + (1*bufloc) \quad (\text{for } i=0 \text{ to } 7) \\
Y = rloc\_origin\_y
\]

7.2.3. Middle region

The middle region contains two 9-bit registers \( \text{first8}\_\text{reg} \) and \( \text{second8}\_\text{reg}, \) \( \text{error}\_\text{mux}, \) and tristate buffers. Only the necessary components are locked the rest of the logic is left for the tool to place automatically (also shown in Figure 7.2).

7.2.3.1 \textit{first8}\_\textit{reg} placement

The nine most significant bits of the error signal coming out of the last PE in the group of first eight PEs are given to the register called \( \text{first8}\_\text{reg}. \) This register is locked to a rectangular area containing one horizontal slice and 5 vertical slices. Bottom-left coordinates for the rectangular area are given below:

\[
X = rloc\_origin\_x + 16 \\
Y = rloc\_origin\_y + 5
\]

while top-right coordinates are given as under:
X = rloc_origin_x + 16  
Y = rloc_origin_y + 9  

7.2.3.2 second8_reg Placement

Similar to first8_reg, 9 bit error signal coming out from the last PE in the second group of 8 PEs is constrained to another rectangular region with following coordinates for top-right and bottom-left corners. (also shown in Figure 7.2)

Bottom-left:

\[
X = rloc_origin_x + 17  
Y = rloc_origin_y + 5
\]

Top-Right:

\[
X = rloc_origin_x + 17  
Y = rloc_origin_y + 9
\]

7.2.3.3 Error multiplexer (error_mux)

error_mux is also locked to a rectangular region shown in Figure 7.2. Coordinates are given below.

Bottom-left corner:

\[
X = rloc_origin_x + 16  
Y = rloc_origin_y
\]

Top-right corner:

\[
X = rloc_origin_x + 16  
Y = rloc_origin_y + 4
\]
7.2.3.4 Tristate buffers for error_mux (tbuf_for_error_mux)

tbuf_for_error_mux is a set of 9 tristate buffers placed after error_mux for the 9 most significant bits of the error signal. It is also placed in the 'mid-region'. So the coordinates for the RLOC_ORIGIN for the set are given below.

\[
X = \text{rloc\_origin\_x} + 16 + (1 \times \text{bufloc}) \\
Y = \text{rloc\_origin\_y}
\]

7.2.4 Reverse placement for odd rows

When the module PE2X8 is part of an odd row then RPMs within PE2X8 are also reversed. As a result of this, the first RPM of the first group of eight is placed at the right most horizontal position which is \( \text{rloc\_origin\_x} + 32 \). The variable \( \text{rloc\_origin\_x\_new} \) contains this reverse value

\[
\text{rloc\_origin\_x\_new} = \text{rloc\_origin\_x} + 32
\]

The value of ‘32’ comes from the fact that in Figure 7.1, the last PE of the second set is placed at location X32 and X33. Hence the total 34 horizontal locations are needed to place one PE2X8 module. Therefore following coordinates are used

RPM Placement:

\[
\begin{align*}
X &= \text{rloc\_origin\_x\_new} - (i \times 2) \quad \text{(for } i = 0 \text{ to } 7) \\
Y &= \text{rloc\_origin\_y} \\
X &= \text{rloc\_origin\_x\_new} - 2 - (i \times 2) \quad \text{(for } i = 8 \text{ to } 15)
\end{align*}
\]
\[ Y = rloc\_origin\_y \]

Buffer’s Placement:

\[
\begin{align*}
X &= rloc\_origin\_x\_new-(i*2)+(1*bufloc) \quad \text{(for } i=0 \text{ to } 7) \\
Y &= rloc\_origin\_y \\
X &= rloc\_origin\_x\_new-2-(i*2)+(1*bufloc) \quad \text{(for } i=8 \text{ to } 15) \\
Y &= rloc\_origin\_y
\end{align*}
\]

Hence, generics needed by module PE2X8 are summarized below:

- \( rloc\_origin\_x \): Bottom Left X coordinate
- \( rloc\_origin\_y \): Bottom Left Y coordinate
- \( bufloc \)
  - 0 -> PE2X8 starting with even horizontal location
  - \( (rloc\_origin\_x \text{ is Even}) \)
  - 1 -> PE2X8 starting with odd horizontal location \( (rloc\_origin\_x \text{ is odd}) \)
- \( reverse \)
  - 1 -> PE2X8 is part of Odd Row
  - 0 -> PE2X8 is part of Even Row

### 7.3 SARow placement

SARow places all the PE2X8 modules in a constrained manner. Flexible placement methodology is achieved by allowing different structures for odd row placement, even row placement, forward placement, reverse placement for more than 2 PE2X8 modules, reverse placement for less than 2 PE2X8 modules, odd starting location, and even starting location. Different placement schemes will be described in this section.
7.3.1 Placement starting with even location

For this case \( rloc\_origin\_x \) coordinate for SARow placement is an even number. We will consider both the possibilities, either it is an even row (forward placement) or it is an odd row (reverse placement).

7.3.1.1 Forward placement

Forward placement is performed for even rows, i.e., row0 and row2. During forward placement PE2X8 modules are placed in an increasing order from left to right on chip. For even rows, RPMs within each PE2X8 are also placed in forward way. Each SARow can accommodate a maximum of four PE2X8 modules. As end row logic (row multiplexers, comparator, motion vector registers, pipeline-stages) is placed in the middle of the SARow, different placement schemes exist for the first two PE2X8 modules and the last two modules.

Placement of first two PE2X8

The first two PE2X8 modules are placed to the left side of 'mid-row-logic' with following parameters.
Each PE2X8 module covers 34 horizontal slices hence every PE2X8 module (except the first one) will start with an offset which is a multiple of 34.

Horizontal location for the first two PE2X8 modules starts with an even x-coordinate thus $bufloc$ should remain '0' for the first two PE2X8 modules. Meanwhile, the variable 'reverse' is also 'false' indicating even row.

**Placement of last two PE2X8 modules**

For some particular structures, $SARow$ may contain only two PE2X8 modules. However, for a structure containing four PE2X8 modules (64 PEs) in one SARow, placement of the last two PE2X8 modules start with an offset of '3'. Reason is that 'mid-row-logic' covers a rectangular area of three horizontal slices and 20 vertical slices. As this offset has an odd value, the last two PE2X8 modules will start with an odd horizontal slice-location. Therefore the structure should be flexible enough to accommodate all these possibilities.

The following parameters are provided to the last two PE2X8 modules (if applicable)

\[
\begin{align*}
Y & => rloc\_origin\_y + 3 \\
x & => rloc\_origin\_x + (i*34) \\
reverse & => false \\
bufloc & => 1
\end{align*}
\]
During the reverse placement corresponding to odd rows (row1, row3), the first two PE2X8 modules will be placed to the right side of the 'mid-row-logic'. This is true if SARow contains more than two PE2X8 modules. However, if the final structure contains only two PE2X8 modules (corresponding to eight row structures) then we have to provide enough provision in the placement scheme to place the first two PE2X8 modules to the left side of the 'mid-row-logic'. For this purpose, another parameter is defined called 'columns_pe2x8_array'. Under this scenario, 'middle-row-logic' will sustain its position as it was for four modules.

The parameter 'columns_pe2x8_array' provides information of how many PE2X8 modules the structure contains in one row. This generic element comes from MeProC module. With the help of this parameter, we are able to know in advance, if we have to place the first two modules to the right or left side of the middle region shown in Figure 7.5.
Parameter \( rloc\_origin\_x\_reverse \) is very important for reverse placement. This provides x-coordinates for placing the right-most PE2X8 in an odd row. This parameter is initialized with \( rloc\_origin\_x \) plus an offset value. The offset value is calculated as a sum of two values.

\[
rloc\_origin\_x\_reverse = rloc\_origin\_x + (\text{no}\_\text{of}\_2\times8\_\text{pe}-1) \times 34 + \text{reverse}\_\text{midx}\_\text{jump}
\]

The first component of the offset provides the desired number of 'multiples of 34' corresponding to the number of PE2X8 modules. SARow also receives an additional parameter \( \text{reverse}\_\text{midx}\_\text{jump} \) from MeProC module. This parameter is very important as it decides the 'jump' that reverse position has to make for the two types of structures, one is 'four row' architecture and the other one is 'eight row' architecture. In case of the 'four row' architecture this parameter should be zero because the reverse placement has to start from the left side of the 'middle-row-logic', hence there is no need to make a 'jump'. However if we have 'four row' architecture then each row contains more than two PE2X8 modules then reverse x position has to make a jump of '3'.

Figure 7.5. Placement scheme where SARow contains only two PE2X8 modules.
For eight row architecture the first two PE2X8 modules in reverse placement will receive the following parameters.

\[
\begin{align*}
    rloc\_origin\_y & \Rightarrow rloc\_origin\_y, \\
    rloc\_origin\_x & \Rightarrow (rloc\_origin\_x\_reverse -(i*34)), \\
    reverse & \Rightarrow true, \\
    bufloc & \Rightarrow 0
\end{align*}
\]

However, if we have more than two PE2X8 modules then we will place them to the right side of the ‘mid_region’ with the following parameters.

\[
\begin{align*}
    rloc\_origin\_y & \Rightarrow rloc\_origin\_y, \\
    rloc\_origin\_x & \Rightarrow (rloc\_origin\_x\_reverse -(i*34)), \\
    reverse & \Rightarrow true, \\
    bufloc & \Rightarrow 1
\end{align*}
\]

**7.3.2 Placement starts with an odd horizontal location**

If the first RPM starts with an odd value of \( rloc\_origin\_x \) then we have to adopt a totally opposite methodology from the above one.

**7.3.2.1 Forward placement**

If it is an even row then there are two further cases, whether PE2X8 modules are the first two or the last two because each case has a different type of starting x-coordinate.

Second two PE2X8 modules:

As the modules will start with an even location, \( bufloc \) should be ‘0’ and other parameters are given below:
rloc_origin_y => rloc_origin_y
rloc_origin_x => (rloc_origin_x+(i*34)+3)
reverse => false
bufloc => 0

The First two PE2X8 modules start with an odd location therefore they will be provided with following values:

rloc_origin_y => rloc_origin_y
rloc_origin_x => (rloc_origin_x+(i*34))
reverse => false
bufloc => 1

7.3.2.2 Reverse placement

The reverse placement is again slightly complicated because it has to include several possibilities. Again things are simpler for the second two PE2X8 modules (if applicable) with the following parameters

rloc_origin_y => rloc_origin_y
rloc_origin_x => (rloc_origin_x_reverse -(i*34)-3)
reverse => true
bufloc => 1

with the first two PE2X8 modules we have to sort out whether the total number of PE2X8 in the row are less than three or greater than two. Depending upon the information from 'columns_pe2x8_array' the following parameters are provided to the PE2X8 module.

case: columns_pe2x8_array < 3

rloc_origin_y => rloc_origin_y,
rloc_origin_x => (rloc_origin_x_reverse -(i*34)),
reverse => true,
bufloc => 1
case: columns_pe2x8_array > 2

rloc_origin_y => rloc_origin_y,

rloc_origin_x => (rloc_origin_x_reverse -(i*34)),

reverse => true,

bufloc => 0,

7.4 Placement scheme for MeProC

This module provides a a correct placement methodology for all the desired number of SARows. MeProC needs the following placement generics:

1) rloc_origin_y
2) rloc_origin_x
3) no_of_lines
4) REF_HEIGHT

The reference block has a square size. Therefore parameter ‘REF_HEIGHT’ is enough to provide the desired width and height for the reference block.

no_of_lines informs MeProC about the desired number of rows for REF_HEIGHT x REF_HEIGHT PEs to be placed on the chip for the entire systolic array. rloc_origin_x and rloc_origin_y are the coordinates of the bottom-left corner of the very first row. Location coordinates for the rest of the SARows are calculated from these two values.

The parameter no_of_lines is very important, as it decides how many SARow modules will be placed which then translates to the number of PE2X8 modules part of each SARow. So it decides how many processing elements (RPMs) will be placed in each row. This thesis work describes both eight row placement methodology and four row methodology.

The parameter columns_2x8pe_array present in MeProC and needed by SARow is computed as:

\[
\text{columns}_\text{2x8pe}_\text{array} = \frac{(\text{REF}_\text{HEIGHT}*\text{REF}_\text{HEIGHT})}{(\text{no}_\text{of}_\text{lines}^* \text{REF}_\text{HEIGHT})}.
\]

88
where \( PES\_IN\_2X8PE = 16 \)

\[
\text{columns}_{2\times8pe\_array} = \frac{16 \times 16}{4} = 4.
\]

With the above parameters we have four \( PE2X8 \) modules in SAsync and thus 64 processing elements (RPMs) in each SAsync and total of four rows will contain 256 processing elements.

Following parameters are passed to the SAsync module.

\[ \text{REF\_HEIGHT}=16 \]
\[ \text{no\_of\_lines}=4 \]

\[ \text{columns}_{2\times8pe\_array} = (16 \times 16) / 4 \times 16 = 4. \]

7.4.1 Four row architecture placement

Suppose we have the following parameters then the overall structure should look like Figure 7.6.

Figure 7.6. MeProC contains four SARows.
\[
\begin{align*}
\text{rloc\_origin\_y} & \rightarrow (\text{rloc\_origin\_y} - (i \times 10)) \quad \text{(for i=0 to no\_of\_lines-1)} \\
\text{rloc\_origin\_x} & \rightarrow \text{rloc\_origin\_x} \\
\text{row} & \rightarrow i \\
\text{reverse\_midx\_jump} & \rightarrow 3
\end{align*}
\]

Figure 7.7. MeProC contains eight SARows.
Since each RPM has a height of 10 slices, the next row will be placed at 10 slices below the first row. row parameter gives information to the SARow modules whether it is an even row or an odd row. For four row architecture, rloc_origin_x_reverse parameter present in each SARow has to make a jump of ‘3’. Therefore the proper value is passed.

7.4.2 Eight rows architecture placement

For this architecture we again consider 256 processing elements, i.e., REF_HEIGHT=16. This time the parameter no_of_lines = 8 thereby giving us the architecture of Figure 7.7.

where

columns_2x8pe_array = (16 * 16) / 8 * 16 = 2

This means that we have two PE2X8 modules in one SARow and that each SARow contains 32 Processing elements and again in total we have 32x8 = 256 processing elements. The following parameters are passed to the SARow modules:

\[
\begin{align*}
    rloc\_origin\_y & \quad \Rightarrow (rloc\_origin\_y - (i*10)) \\
    rloc\_origin\_x & \quad \Rightarrow rloc\_origin\_x \\
    row & \quad \Rightarrow i \\
    reverse\_midx\_jump & \quad \Rightarrow 0
\end{align*}
\]
8. Memory Interface

8.1 Overview

The developed motion estimation processor (MeProC) is part of the FlexWAFE architecture [6], which also contains weakly programmed data communicators (MeDataTF) together with algorithm controller (AC) and central memory controller (CMC).

MeProC needs image data ordered in regular but complex patterns which rises the need for data transformers (MeDataTF). In FlexWAFE architecture data is transferred, reorganized and stored using local memory controllers (LMCs).

Figure 8.1. Structure diagram of MeEngine.

8.2 Local memory controller

There are two different kinds of LMCs depending upon the direction of data flow, either the data stream is read from the external memory (frame buffers) through the
CMC and fed to the MeProC or the stream is read from the receiver port and is written to frame buffers through CMC. The former one is referred to as C2S (CMC to Stream) while the later one is called S2C (Stream to CMC). Each MeProC needs three C2S modules for reference, search upper and search lower stream. In addition to C2S each MeProC needs one S2C in order to write external images into the frame buffers through CMC [5].

Data is provided by the CMC in bursts only. That is why complex address patterns that require single pixels are very inefficient in respect to the memory bandwidth. Hence C2S module in every data transformer (MEDataTF) serves to work with bursts and writes it into FIFO, while the next module called Address Pattern Transformer (APT) can then perform calculations that are more complicated, like zig zag and the specific transformation that we need for MeProC.

![Figure 8.2. Further division of MeDataTF.](image)
8.3 Generator

In order to generate arbitrary address patterns at a sufficient speed the basic building block of LMC is a component called 'generator'. Its structure shown in Figure 8.3.

Generator consists of three connected steppers. Each of the 'steppers' implements a small, fast and flexible counter. The parameter set of the counter includes the starting point, the value that is added in every step and the upper limit.

Base-stepper provides the 'starting' parameter while limit-stepper provides the end parameter for 'addr-stepper'. Once the parameters are set, base-stepper and limit-stepper starts with base-start and limit-start values. These values are fed to the addr-stepper, which generates values from base_start to limit_start with increment of addr_delta. Once it finishes this counting it produces 'done' signal to the base and limits stepper which provide next count parameters for addr-stepper.
*addrs-stepper* will keep on counting until it reaches the final parameters of *base-stepper* and limit-stepper. Further explanation is given below in a very simple example.

A picture with dimension 6x6 is shown in Figure 8.4. The desired reading order is column wise (as required by *MeProC*) which is also shown in Figure 8.4 with correct timing order. The following parameters are selected in order to generate the desired address pattern shown in Figure 8.4.

\[
\begin{align*}
\text{base}_\text{start} &= 0, \quad \text{base}\_\delta = 1, \quad \text{base}\_\text{end} = 30 \\
\text{limit}_\text{start} &= 5, \quad \text{limit}\_\delta = 1, \quad \text{limit}\_\text{end} = 35 \\
\text{addr}\_\delta &= 6
\end{align*}
\]
We can notice from the above example that base_delta and limit_delta are always equal and base_end and limit_end occurs at the same time.

**8.4 Local address controller**

Hence by providing different sets of these parameters to the generation unit we can produce different addressing patterns. By programming these parameters during run time week programmability can be introduced. This week programmability is controlled by a central algorithm controller, AC. The AC stores all the possible sets of parameters in its memory. Having one memory for all the LMCs is not a feasible
solution (from a timing point of view) because we have shared buses for address and data. A solution is to have the address parameters kept in a small local parameter memory and also use a local control address unit, this module is called local address controller (LAC) as shown in Figure 8.5.

Hence LAC (local address controller) together with 'generator' form a flexible and weakly programmable address generator unit.
8.5 Algorithm controller

Parameters are transmitted sequentially in advance over the low bandwidth control bus from the algorithm controller to the LAC unit where each parameter register has its own unique address. In effect this local memory provides a shadow register for the working parameters that currently control the address generator. Once address generation for one set of parameter is finished, the local address controller sends an acknowledgment signal to the AC which feeds the next set of parameters to the address generator. In this way, the AG can work uninterrupted while AC can control and synchronize multiple address generation and operation sequences.

As shown in Figure 8.6 AC consists of a micro code memory and a simple micro code sequencer implemented as a FSM (finite state machine). Simple sequencings need a fixed execution time. Synchronization is used to account for the buffered DDR-SDRAM memory access. The AC reacts to LMCs via point to point connections and controls them through the shared parameter bus by programming their local shadow registers.

Figure 8.6. Algorithm controller.
8.6 Global base stepper

In order to access the DDR SDRAM at a high bandwidth a scheduling memory controller (CMC) was developed [6]. Forward MeProC and backward MeProC use CMC0 and CMC1 respectively to access the two external frame buffers. Each of these buffers is capable of storing four images of size 4k x 4k with each pixel consisting of 16 bits. The two MeProCs operate on three images simultaneously. One image is the reference image and two search images with one forward and one backward image. Besides this the current incoming image also needs to be stored. These buffers rotate their position in the frame buffer thereby allowing real time processing of the data (as shown previously in Figure 3.8).

Figure 8.7. Use of global base stepper.

The three images are placed on different parts of the frame buffer. Thus a global offsets are also needed in addition to the local offsets generated by 'generator' module. Hence the address generated in C2S and S2C part of LMCs needs modification in terms of an extra stepper global-addrs-stepper and an adder. Final CMC address also appends some additional zeros at least significant and most
significant locations.

8.7 Building blocks of C2S, APT, & S2C

Major building blocks of C2S, S2C and APT modules are shown in Figure 8.8, 8.9, 8.10.

C2S consists of following parts
1. LAC
2. Generator (base stepper, limit-stepper, addr-stepper)
3. Global Base Stepper
4. FIFO

LMC APT contains following components
1. 'ingress' generator
2. 'egress' generator
3. Block RAM

![Figure 8.8 C2S.](image1)
![Figure 8.9. APT.](image2)
S2C has the following major elements.

1. LAC
2. Generator (base stepper, limit-steppe, addrs-stepper)
3. Global Base Stepper
4. FIFO

8.8 Final chip

All the elements which are part of the FlexWAFE0 chip are shown in chapter 3. In chapter 5 we presented snap of the chip showing only two MeProC covering a rectangular structure over the chip. Two CMC modules are also shown and rest of the space was reserved for other LMC components and IO interface modules as shown in Figure 8.11. During the efforts to constrain CMCs, AC, Rx and Tx components, it was noticed that rectangular structure may not be suitable for the desired placement as it does not provide enough required space for all the components. Instead a square structure will be more beneficial. Flexibility of the placement was then used to generate a square structure having eight SARows, where each SARow contains 32 PEs as shown in Figure 8.12. Because of this particular structure we have now enough room for placing motion compensator and its memory interface components.
Figure 8.11. Rectangular structure for MeProC.
8.9 Verification Plan

Verification of motion estimation is carried out at two levels:

a) Verification of MeProC only
b) Verification of MeProC with Memory Interface
Because of the special way of accessing memory for MeProC, the first level is the proper check of the hardware by providing ideal pixel streams in the desired form and then verify the functionality of the MeProC at different abstraction levels. The second level of verification provides correct functionality along with proper memory interface by including Local Memory Controllers and Central Memory Controller. At this level we are fully confident that MeProC will provide correct motion vectors if we feed it with correct order of pixel data. Following is a brief explanation of each of these verification schemes.

8.9.1 Verification of MeProC only

An automated verification of MeProC is needed covering images of different sizes and having different size of the reference blocks. For this purpose, an automated environment was developed as a combination of MATLAB Scripts and VHDL test-benches. Basic rational behind the adopted methodology is to generate a random search and reference image such that we already know the relative motion between the two images in terms of motion vectors. Based on this information pixel streams for search upper, lower and reference are generated which are then read by the VHDL test-benches and are fed to the MeProC during Modelsim simulation. As a result of the simulation motion vectors are stored in files which are then compared to the original motion vectors to validate the hardware.

The following section describes very briefly the different scripts developed.

8.9.1.1 Matlab scripts

1. Random motion vectors generation
   Matlab Script gen_xyvectors.m provides randomly generated motion vectors for the desired image sizes. Within the script proper validation is also handled.

2. Random reference and search image generation.
   Matlab script build_me_image_sets.m takes motion vectors as input and builds a search image. This search image is further used to extract the reference image depending upon motion vectors and search upper and lower pixel streams are also produced.

3. Verification of motion vectors generated by MeProC
   Reference motion vectors and the vectors generated by the hardware are compared in verify_motionvectors.m.
8.9.1.2 VHDL testbench

Image data set created by Matlab scripts are read by the module *dpu_image_file_input16* which reads *.pgm* files. This module has the necessary generics to allow reading different sizes of multi-frame images with different structures. It models a real time interface by allowing proper 'read enable' and 'strobe' signals for the data. Each of the input pixel streams needs one such module. In addition to 'pixel-stream reading', writing of the motion vectors generated, in the form of image files accomplished with the use of *dpu_image_file_output16* files which are similar in construction to *dpu_image_file_input16* except the difference in direction of data flow as, this file takes motion vectors and stores them in *.pnm* files which are then read by Matlab scripts for the validity. These files and DUT (MeProC) is instantiated inside script *motion_estimation_tb.vhd* to form the top level testbench.

8.9.2 Verification of MeProC with memory interface

After verification of the MeProC, the next step is to include CMC and all the required LMCs. For this purpose we need to provide correct data for the local programme memories of LMCs. These parameters are described in a human
readable and easily editable XML file and transformed into VHDL using a XSLT script. This VHDL file contains the memory contents of the AC. The AC state machine will read it and send the parameters to the desired LACs that will in turn control the AGs.

Verification of the MeProC with LMCs starts by using a part of the 4k x 4k image. The smaller image takes out only 64x48 pixels from the 4k x 4k image. Once we have correctly programmed the parameters and received correct pixel order out of LMCs, we can extend verification to bigger images. Each frame buffer is divided into four buffers to store four frames of video. Hence each LMC needs to be programmed for four buffer locations. This corresponds to four main parameter sets for each of the LMC. Further one more parameter set is required to restart the address generation for each LMC without interruption in address generation. Thus for S2C and reference C2S for each MeProC we need five sets of parameters. These parameters include base_start, base_delta, base_end, addr_delta, limit_start, global_base_start, and global_base_end and instruction.

The 'instruction' contains control information for the address generators like enable, stop and reset signals for the address generators. Information of 'global_base_delta' is also embedded into the instruction as it has smaller possible delta values. Programming of S2C and reference C2S is the same with the different values of global_base_start and global_base_end because of using different buffers at the same time i.e. rotating buffer mechanism.

Search upper C2S (SU_C2S) and search lower C2S (SL_C2S) contain more sets of parameters because of different access schemes at top and bottom boundaries of the image. SU_C2S needs two different parameters sets, one for the top of the image and the other for the rest of the image in each buffer. Similarly SL_C2S needs two parameter sets, one for bottom part of the image and the other one for rest of the image for corresponding to each buffer. Hence total they need nine parameter sets including one for resetting back to the first position. In case of SU_C2S and SL_C2S global_base_delta is '2' because of the need to access a shorter region of the buffer.

In addition to C2S programming for each of the incoming pixel stream to MeProC, APT modules are also needed to be programmed in order to get correct transformation of the data. Both ingress and egress modules should be programmed. Their programming is simpler as they only need to access a smaller
on chip block RAM with simple address generation.

The following example shows the parameter set for search upper LMC, in order to access only top 8 rows of the image corresponding to top boundary of the image. Figure 8.14 shows the offsets generated by LMC C2S, corresponding CMC address and the contents of the image.

Figure 8.14. Example to access top part of the image for search upper.

C2S module will produce data in the order shown in Figure 8.14. This data is then fed to the APT module. Both 'ingress' and 'egress' of APT are also programmed in correct way such that we get the required form of data shown in Figure 8.15.
Figure 8.15. a. Pixel sequence at the output of C2S module.

Parameter set for 'ingress' port of search upper APT

*base_start ms = 0*

*base_delta ms = 0*

*base_end ms = 0*
addr_delta  ms = 1  
limit_start ms  =  63  
limit_delta ms =  0  

Parameter set for 'egress' port of search upper APT  

base_start    = 0  
base_delta   = 1  
base_end     = 15  
addr_delta   = 16  
limit_start   = 112  
limit_delta  = 1
9. Conclusion

Bidirectional motion estimation is performed using an exhaustive search and block matching SAD algorithm, on luminance channel of the video frames. Motion estimation core uses FlexWAFE reconfigurable architecture where FPGAs are configured using macro components that consist of weakly programmable local memory address generator (LMC) that supports sophisticated memory pattern transformation and data stream processing units (DPU). FlexWAFE architecture for bidirectional motion estimation uses two cores of motion estimation engine (MeEngine) forming the main data processing unit (DPU) for backward and forward motion vectors. Memory interface is provided to the two MeEngines by using one central scheduling SDRAM controller and eight local memory controllers.

In order to facilitate controlled and parametrized placement, each MeEngine is decomposed into three stages. MeProC maintains flexible placement of desired number of SARows and global control for entire data flow in the systolic array. The building block of data path MeDP is a RPM-macro which represents one processing element and performs 10-bit difference, a comparison and 19-bit accumulation. A combination of 16 RPM-macros in specific order gives rise to PE2X8 component and desired number of PE2X8 components are integrated to produce one SARow. Four SARows are then cascaded together with separate placement procedures for even and odd numbered row. Each level produces a very compact entity with its local control and its own placement methodology. Therefore bidirectional motion estimator processes 24 frames per second, where each frame may have a size of up to 2048 x 2048 pixels with 10 bits per pixel. This results in the enormous computational power of 155 Gop/s.

All the components are modelled with VHDL and synthesized and implemented using Xilinx ISE 7.1. All the modules have been simulated with Modelsim 7.1b and verified for correct functionality and desired speed after the place-&-route
level. MeProC with four SARows for 256 PEs is successfully tested against the performance figure of 125 MHz. However the structure with eight SARows was not tested for desired speed because of the 'timing constraints' of the thesis work itself with time period of 'six months'. Very high resource utilization, closer to 90% of the chip resources, makes it difficult to optimize the blocks with Xilinx Floorplanner therefore use of Xilinx PlanAhead is highly recommended for future placement work with developed bidirectional motion estimation.
# Table of abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AC</td>
<td>Algorithm controllers.</td>
</tr>
<tr>
<td>AG</td>
<td>Address generator.</td>
</tr>
<tr>
<td>BUFT</td>
<td>Tristate buffers.</td>
</tr>
<tr>
<td>BMA</td>
<td>Block matching algorithm.</td>
</tr>
<tr>
<td>BRA</td>
<td>Block recursive algorithm.</td>
</tr>
<tr>
<td>CLB</td>
<td>Configurable logic block.</td>
</tr>
<tr>
<td>CMC</td>
<td>Central memory controller.</td>
</tr>
<tr>
<td>DSP</td>
<td>Digital signal processors.</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field programmable gate arrays.</td>
</tr>
<tr>
<td>FlexWAFE</td>
<td>Flexible weakly programmable advance film engine.</td>
</tr>
<tr>
<td>HDTV</td>
<td>High definition television.</td>
</tr>
<tr>
<td>IDA</td>
<td>Institute of Computer Communication &amp; Network Engineering.</td>
</tr>
<tr>
<td>LMC</td>
<td>Locally memory controller.</td>
</tr>
<tr>
<td>LSB</td>
<td>Least significant bit.</td>
</tr>
<tr>
<td>LUT</td>
<td>Lookup table.</td>
</tr>
<tr>
<td>LAC</td>
<td>Local address controller.</td>
</tr>
<tr>
<td>MeEngine</td>
<td>Motion estimation engine.</td>
</tr>
<tr>
<td>MSE</td>
<td>Mean square error.</td>
</tr>
<tr>
<td>MSB</td>
<td>Most significant bit.</td>
</tr>
<tr>
<td>MeProC</td>
<td>Motion estimation processor.</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Definition</td>
</tr>
<tr>
<td>--------------</td>
<td>------------</td>
</tr>
<tr>
<td>MeDP</td>
<td>Motion estimation data path.</td>
</tr>
<tr>
<td>MeGC</td>
<td>Motion estimation global controller.</td>
</tr>
<tr>
<td>MeDataTF</td>
<td>Motion estimation data transformer.</td>
</tr>
<tr>
<td>MeDataWr</td>
<td>Motion estimation data writer.</td>
</tr>
<tr>
<td>NTSC</td>
<td>National television systems committee. A television broadcast standard mostly used in United States, Canada and Japan.</td>
</tr>
<tr>
<td>PAR</td>
<td>Place &amp; Route.</td>
</tr>
<tr>
<td>PEL</td>
<td>Pixel element.</td>
</tr>
<tr>
<td>PE</td>
<td>Processing element.</td>
</tr>
<tr>
<td>PEs</td>
<td>Processing elements.</td>
</tr>
<tr>
<td>PAL</td>
<td>Phase alternated line. A television broadcast standard used mostly in Europe.</td>
</tr>
<tr>
<td>PE2X8</td>
<td>Group of 16 processing elements.</td>
</tr>
<tr>
<td>RPM</td>
<td>Relationally placed macro.</td>
</tr>
<tr>
<td>SAD</td>
<td>Sum of added differences.</td>
</tr>
<tr>
<td>SA</td>
<td>Systolic array.</td>
</tr>
<tr>
<td>SU</td>
<td>Search upper.</td>
</tr>
<tr>
<td>SL</td>
<td>Search lower.</td>
</tr>
<tr>
<td>SDTV</td>
<td>Standard definition television</td>
</tr>
<tr>
<td>SARow</td>
<td>Systolic array row.</td>
</tr>
<tr>
<td>5/3 Wavelet</td>
<td>The integer Wavelet coefficient set ((-\frac{1}{2}, 1, -\frac{1}{2})) and ((-\frac{1}{2}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, -\frac{1}{8})) for high-pass and low-pass analysis respectively.</td>
</tr>
</tbody>
</table>
Bibliography


Detta dokument hålls tillgängligt på Internet – eller dess framtidiga ersättare – under en längre tid från publiceringsdatum under förutsättning att inga extra-ordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnt som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/