XILINX®

A 32x16 Reconfigurable Correlator for the XC6200

XAPP 084 July 25, 1997 (Version 1.0)

Application Note by Ann Duncan

Summary

A correlator design for the XC6200 is discussed. Dynamic reconfiguration is exploited to rapidly reconfigure the hardwired match image template into the design.

Xilinx Family

- XC6200

Demonstrates

- Use of Fast Map Interface
- Dynamic reconfiguration at Logic Level
- Register Access through control store interface
- Codesign in C++
- CBUF timing strategy

Table of Contents

Introduction

Image correlation is performed by passing a template over an image and determining at each pixel position if a match has been found. The design described uses a 32x16-pixel match image and a 32x16-pixel mask image to construct the template. The match image is a small image to be located within a larger image. The mask image allows masking out of background regions in the match image. Only unmasked pixels will be correlated. If the number of matching pixels at any point in the image exceed a threshold, a Hit is detected.

The correlator is implemented as a pipelined design. Image data is fed in 16 rows at a time into a 32-stage pipeline. At each clock step the pixel data in each of the 512 registers is compared to the template. The match and mask images are hardwired into the logic. See Figure 1. The ability to dynamically reconfigure gates means that the image template can be rapidly reconfigured an unlimited number of times. Partial reconfiguration and fast reconfiguration times make the XC6200 ideal for this application, and by using XACT6000, reconfiguration data can be generated easily, thus minimizing development time even further.

An XC6200, resident on an XC6200DS, is programmed with a design to perform correlation on binary images. This note describes design flow showing how using the codesign approach, software and hardware tasks can be tackled simultaneously. Details are given of correlator development,

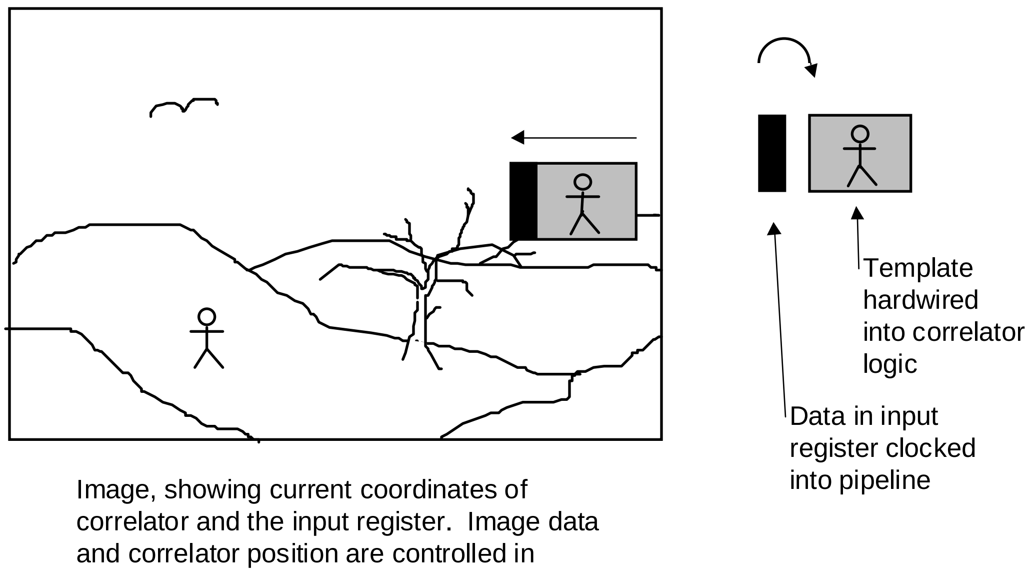

Template
hardwired
into correlator
logic
Data in input
register clocked
into pipeline
Image, showing current coordinates of
correlator and the input register. Image data
and correlator position are controlled in

Figure 1. Image being pushed into the correlator pipeline in rows of 16

modeling correlator functionality in software, logic simulation, and emulation (actual design running in FPGA). The standard XC6200DS libraries, linking user design elements to actual array resources, are presented. The logic architecture for an array of custom adders is described. ASIC designers will be familiar with this static ASIC design process; altering design for regularity and reconfiguration will be new. Finally, performance data is given for both image correlation and reconfiguration.

Design Flow

Design flow uses a codesign approach allowing software and hardware to be implemented in parallel.

In software:

- Write C++ program to read in the image, mask and match templates, and call a correlator.
- Create C++ classes for *Correlator*, *CorrelatorSoft*, *CorrelatorHard*, *CorrelatorSim* and *CorrelatorBoard*.
- Model correlator design in software using the class *CorrelatorSoft.*

In hardware:

- Capture the design as a schematic, in this case ViewDraw was used.

In software:

- Simulate and debug by defining a correlator of type *CorrelatorSim* and using a simulator, in this case ViewSim was used.

In hardware:

- Generate the Edif for the design.
- Use **XACTstepSeries6000** to optimally place and route the design.
- Generate the cal file, the symbol table and the RAL file.

In software:

- Redefine the correlator in the C++ program as type *CorrelatorBoard*.
- Run the program using the XC6200 hardware accelerated correlator.

See Figure 2.

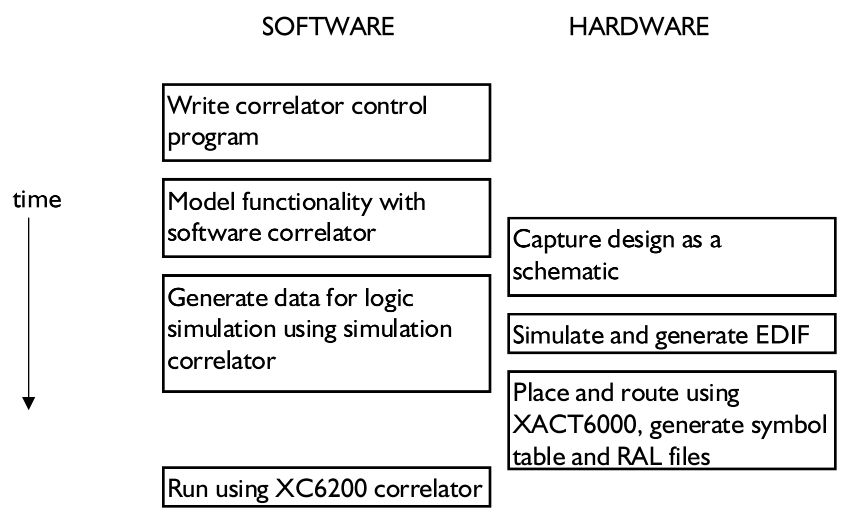

SOFTWARE
HARDWARE
time
Write correlator control program
Model functionality with software correlator
Generate data for logic simulation using simulation correlator
Run using XC6200 correlator
Capture design as a schematic
Simulate and generate EDIF
Place and route using XACT6000, generate symbol table and RAL files

Figure 2. Software and Hardware developed in parallel

Software Design

The Control Program

The C++ program controls both setup and operation.

Setup includes the following steps:

- Initialization of correlator design in one of software, simulation or hardware modes.
- Reading in the match and mask data from files.
- Reconfiguring correlator according to the match and mask images.
- Reading in the image to match against.

Correlator operation:

- Alternately, writing the image data to the input register in 16 pixel columns and checking for Hit
- Displaying crosshairs on the Hit location if Hit = 1.

The Correlator Classes

A correlator base class, *Correlator*, is defined as part of the C++ program. On it are defined arrays for the mask image and the match image; the dimensions of the search image and the threshold value. From this base class is derived the class *CorrelatorHard*. *CorrelatorHard* contains methods common to both simulation and utilization of the ASIC design e.g. a list of the blocks with reconfigurable gates (see section on Logic Design) and a function *build_correlator()* which generates the hierarchical path names for each of these reconfigurable gates. This design follows a recursive structural pattern through the hierarchy making the generation of these strings an elegant process. See Appendix B.

CorrelatorBoard and *CorrelatorSim* in turn derive from *CorrelatorHard*. *CorrelatorSim* performs simulation of a design by generating simulation files from image data. ViewSim was used in this instance. *CorrelatorBoard* uses the *AccessRegister* and *Pci6200* classes to interface to the Development System.

The AccessRegister Class

The *AccessRegister* classes *AccessRegisterSim* and *AccessRegisterBoard*, allow registers in the logic design to be treated as variables in the controlling C++ program. These classes are generic and can be used for variables in any design. *AccessRegisterSim* writes a command file for ViewSim. This file is imported by ViewSim and shows the operation of the correlator via simulation of the ViewLogic design. *AccessRegisterBoard* utilizes the symbol table data from *XACT6000* to evaluate the Map Register, column select and bus width for each variable in the design. For this design there are only two variables, the 16bit input register and the 1bit hit register. The 16bit Map Register selects the rows from which data will be passed on to the bus. Column select selects the column to or from which these bits are transferred. The image data is then written to the hardware correlator. See Appendix B and refer to the documentation on the *AccessRegister* classes for elaboration.

The Pci6200 Class

Below the *AccessRegister* classes in the software hierarchy lies the *Pci6200* function library. These functions perform the low-level interface to the XC6200 and XC6200DS. Included in the functionality of this low-level library is the ability to download a design onto the array and reprogramming it via the *FastMap* interface; set the Map Register, control the Global Clock, and Clear. For further information see the documentation for the *Pci6200* class. Figure 3 demonstrates the levels of each of these classes with respect to user software and the hardware.

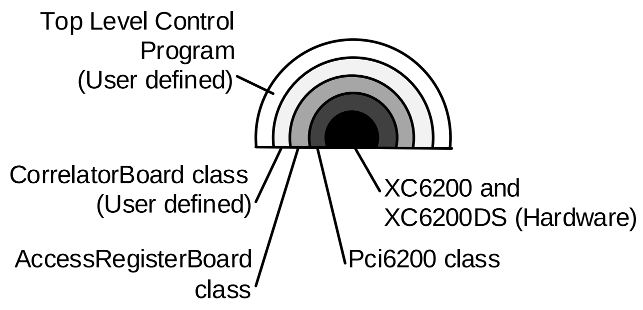

Top Level Control Program (User defined)
CorrelatorBoard class (User defined)
AccessRegisterBoard class
XC6200 and XC6200DS (Hardware)
Pci6200 class

Figure 3. An illustration of the XC6200DS software library hierarchy

The RAL Class

The address data pairs for reconfiguring individual function units are provided by a Ral Software Library function. This library extracts information from a special Ral file generated by XACT6000 for this design. Provided with the following parameters, a function call returns the address/data pairs required to reconfigure a specific function unit in the array: hierarchical instance name, hierarchical net names, and the XC6000 library function name for the new reconfiguration.

Logic Description

Overview

The design comprises the following:

- 16bit input register
- 32x16bit correlator block programmed with mask and match
- Threshold block
- 1bit hit register
- CBUF clock: generates a clock pulse on each write to the input register

Figure 4 is a block diagram illustrating the structure of the correlator logic. The image data is written in columns of 16 pixels to the input register via the

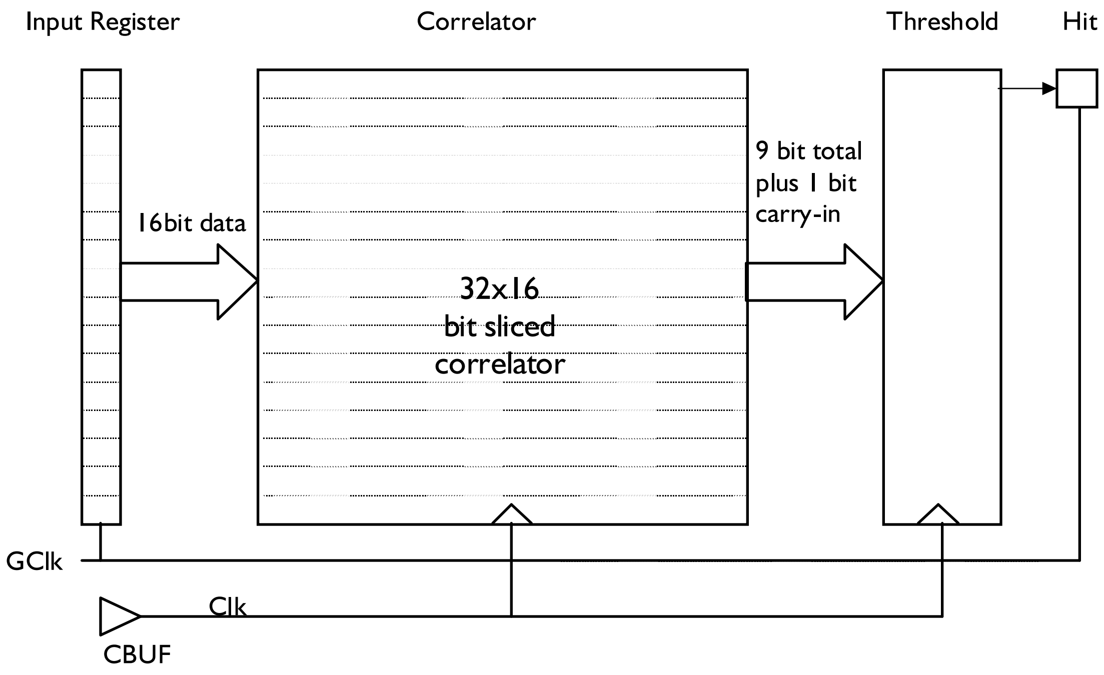


Figure 4. 32x16 pixel correlator design for XC6200

control store interface. (See Application Note XAPP063 for details [1].) On each write to this register, the CBUF generates a clock pulse clocking the design and shifting the columns of data into the pipeline. All pixels in the pipeline are summed in an array of custom adders.

Correlator Logic Details

The operation of the correlator is as follows. On each clock step, the 16bit column of data in the input register is shifted into the 32 bit long pipeline as the next column is written by the control program to the register. Each pixel is logically compared with the match and mask data (hardwired as gates) for its current position and the total number of matches evaluated.

Figure 5 shows a 3-pixel correlator. The gates inside each dotted box perform selection on the pixels depending on the match and mask data. All of these gates can be eliminated from the design if it is possible to reconfigure the gates within the three-bit adder, minimizing the adder logic according to the match and mask data for each pixel.

Figure 6 is the base configuration for the lowest level block of the design showing a reconfigurable 2-stage adder. The adder inputs come from a section of the pipeline. Operation is as follows. Pixel data is clocked through the shift register. Summing of the number of matched pixels takes place over two clock cycles. The output from ABXOR is delayed for one clock cycle and then operates as in a standard adder design when the three pixels being correlated lie in the registers. The adder sums the number of matched pixels, the sum has a one-clock cycle delay, the carry a delay of two. When placed and routed using XACT6000, this block occupies 6 function units, the design in Figure 5 would require 12.

Figure 6 shows this three-pixel adder. As discussed, correlation, or matching against different pixel patterns, is achieved by making each gate of this adder reconfigurable, basing the new gate on the mask and match data. For example, if the pixel on the input was masked and the pixel in the first flip-flop matched with 1, the gate ABXOR would be configured to be a buffer. If the pixel in the first flip-flop matched against 0, ABXOR would be configured to an inverter. Similarly, gates and their net connections are evaluated for CMUX and SUMXOR. Say for example the template for these 3 pixels is as follows [*, 1, 0] where the * signifies that the first pixel is masked out. The pixel on Din is masked, and the

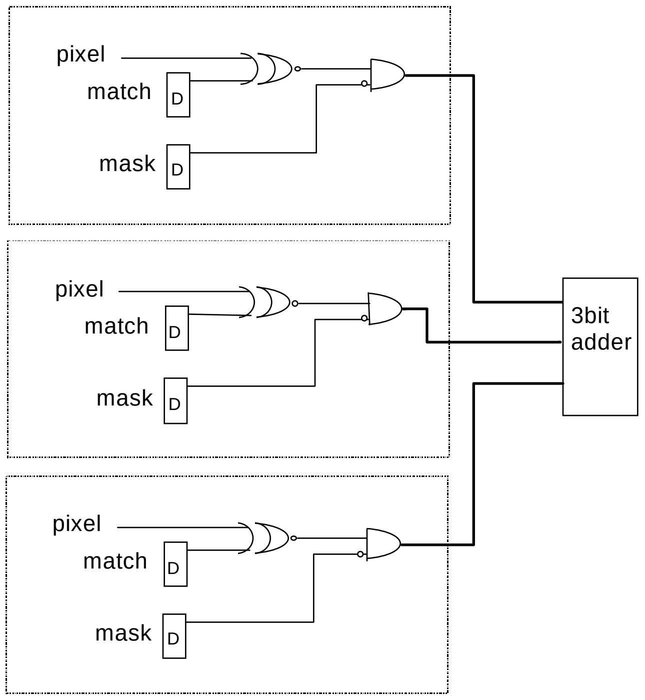


Figure 5. 3-pixel correlator <u>without</u> reconfiguration

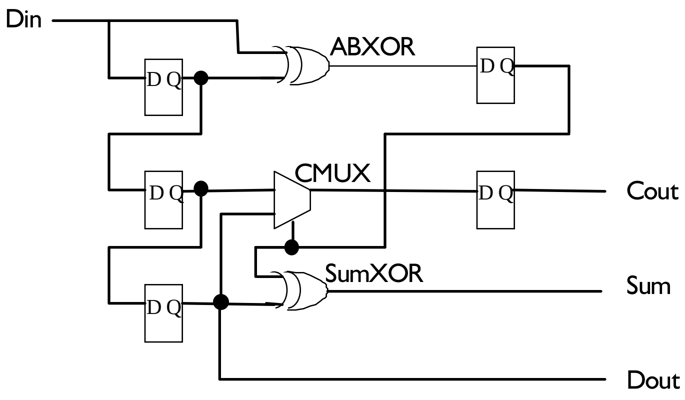


Figure 6. A 3-pixel correlator. Lowest level block of correlator showing 3 pixel shift register and 3 reconfigurable gates

pixel in the top flip-flop is matched with 1. One clock cycle later, all pixels will have shifted one step along the pipeline. Therefore, at T+1, the pixel in the middle flip-flop matches with 1 and the pixel in the bottom flip-flop matches with 0. During reconfiguration, the CMUX is replaced by a MUX with an inverter on the second input and the SumXOR with an XNOR gate.

See Appendix A for all possible reconfigurations of mask and match combinations.

The correlator is constructed hierarchically from these low-level blocks. Two blocks are chained together to make a 6bit shift register. See Figure 7. The sum and carry bits from these two blocks are then summed to give a matched pixel total for the 6bits. As a space saving trick here, an extra pixel is taken as a carry in, making a seven-pixel correlator block. The carry in pixel must be delayed by one clock cycle to synchronize timing with the sum output from the 3-pixel correlator to which it is being added. The gate which takes the carry in is reconfigurable, another ABXOR but with only one of the inputs being a pixel value the other is assumed always to have a mask value of 0 and a match of 1.

The sum of the 511-pixel correlator and the 512th pixel are fed into the Threshold block.

In total, if comparing the reconfigurable design against the alternative shown in Figure 3, a total of 2048 function units are saved. The final design uses 2343 function units in all.

Threshold Block Logic

The threshold value determines the quality of the match of the image with the template. Logically, the threshold block compares the number of matches with a preset threshold *t*, and if it is greater, sets the *Hit* flag. For a perfect match the threshold value would be 32x16=512. In practice, the threshold block operates by adding the total number of matched pixels plus the extra carry in pixel to a constant value and generating an overflow when the threshold is exceeded. This means programming the threshold block with a constant value, 512-*t*, where *t* is the threshold value. This is done by performing a single threshold register write before correlation of the image begins. The overflow or carry appears in the single bit *Hit* register.

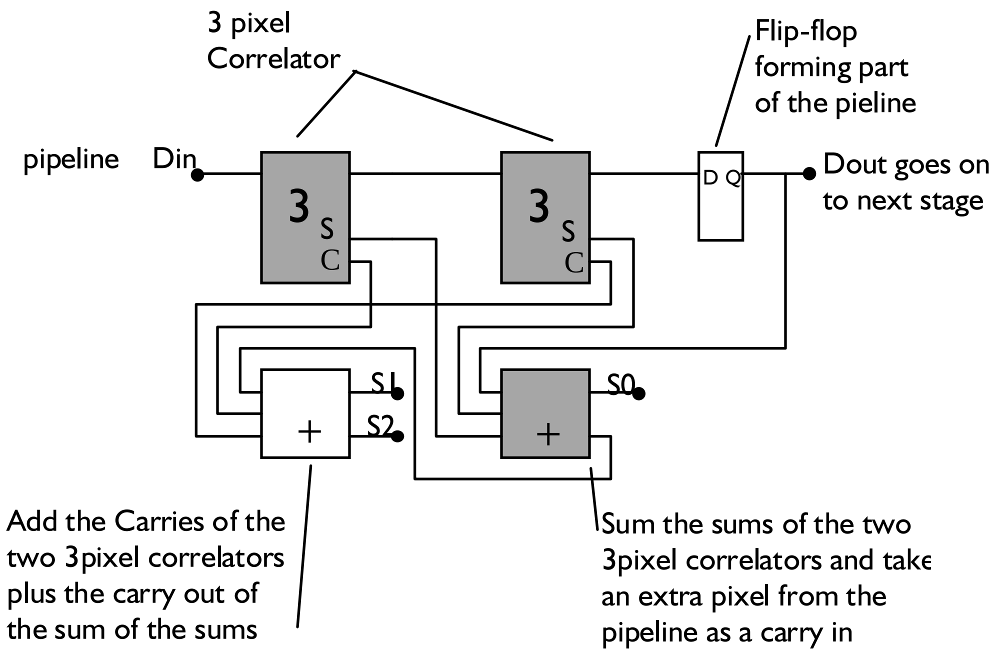

3 pixel Correlator
Flip-flop forming part of the pieline
pipeline
Din
3 S C
3 S C
D Q
Dout goes on to next stage
S1
S2
S0
+
+
Add the Carries of the two 3pixel correlators plus the carry out of the sum of the sums
Sum the sums of the two 3pixel correlators and take an extra pixel from the pipeline as a carry in

Figure 7. A 7-pixel correlator constructed from two 3-pixel correlators in series. Shaded blocks contain reconfigurable gates

Two of these 7 pixel correlators are combined in parallel, their totals summed, plus an extra pixel on the carry in, and delayed by two; this make a 15 pixel correlator (see Figure 8). Two 15 pixel correlators are combined in series with an extra carry in delayed by three clock cycles to make a 31 bit correlator; and so on, until a 511 pixel correlator is constructed, leaving one pixel untested.

Figure 9 shows how the reconfigurable gates map to the pixels in the mask and match template images.

XACT6000 Instantiation

Knowing the regular structure of XC6200 (refer to the XC6200 Datasheet [2]), it is easy to exploit this in structured logic design. XACT6000 performs automatic placement and routing. It can be forced to place and route according to user preference by attaching constraints during the schematic capture phase. This feature is exploited fully in the development of this design. The advantages XACT6000 offers, of maintaining the hierarchy in the physical Instantiation of a design, are shown clearly in Figure 10. This figures shows the XACT6000 layout and clearly maps the structure of the logic, Figures 4, 6, 7 and 8 and the hierarchical breakdown of the image in the same way show logic mapping directly into hardware.

Placement Constraints
RLOC constraints are attached to the instances in the low-level blocks forcing them to be placed in rectangles which can then be tiled together.

Transform Constraints
The way that blocks in this design connect together can be specified by the use of transforms. In the layout of the 15 pixel correlator (Figure 8), for example, it is sensible to place two 7 pixel correlators back to back with the adders in between them. The REF0 constraint is attached to one of the blocks, forcing a reflection in the Y-axis of the default layout for the block.

Flatten Constraint
The FLATTEN constraint forces all gates within an instance to be placed as gates rather than as a single block i.e. the hierarchical structure of the block is removed. This is particularly useful when mapping gates and flip-flops into single cells or when space is at a premium and placing blocks would cause redundancy.

Routing Constraints
Routing is deferred to a higher level with the attachment of the RTDEFER constraint to a schematic sheet. Optimal routing for this design results when deferring low level routing to the 15-pixel correlator block level. This block is routed limiting routing resources to local and length four routing. Subsequent levels in the hierarchy have routing deferred to the top level. All resources are made available for top level routing.

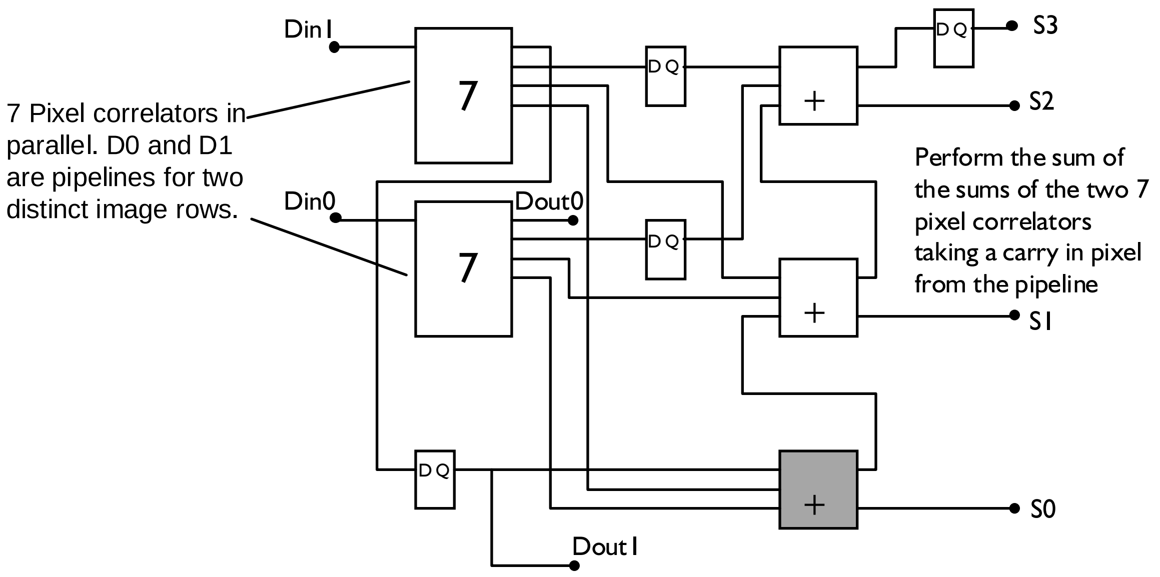

Din1
7 Pixel correlators in parallel. D0 and D1 are pipelines for two distinct image rows.
Din0
Dout0
7
7
D Q
D Q
D Q
D Q
+
+
+
S3
S2
S1
S0
Perform the sum of the sums of the two 7 pixel correlators taking a carry in pixel from the pipeline
Dout1

Figure 8. A 15-pixel correlator constructed from two 7-pixel correlators

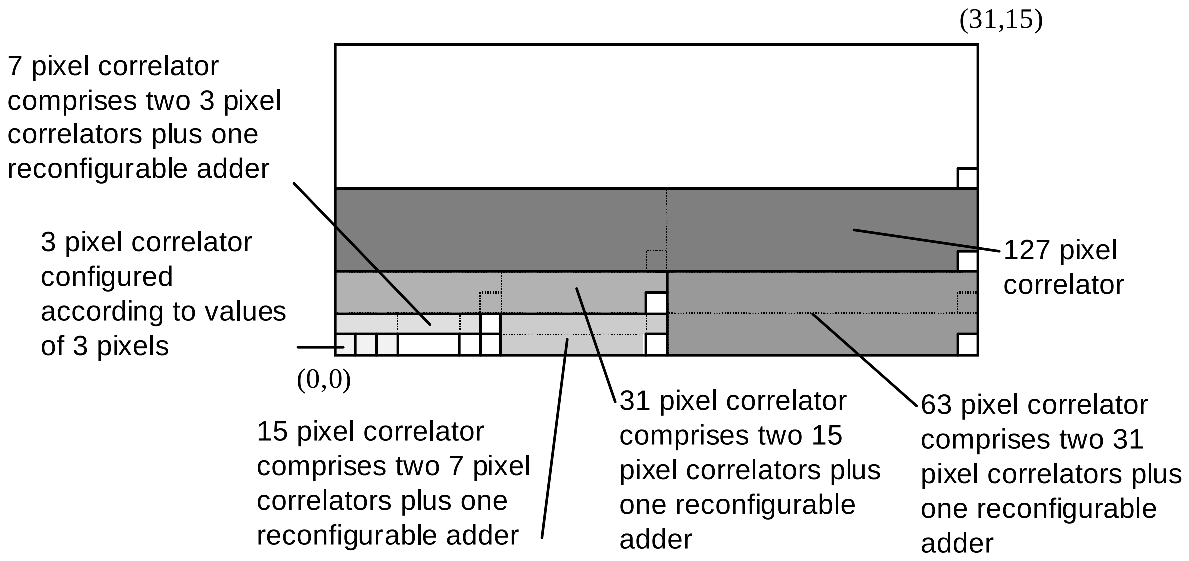

(31,15)
7 pixel correlator comprises two 3 pixel correlators plus one reconfigurable adder
3 pixel correlator configured according to values of 3 pixels
127 pixel correlator
(0,0)
15 pixel correlator comprises two 7 pixel correlators plus one reconfigurable adder
31 pixel correlator comprises two 15 pixel correlators plus one reconfigurable adder
63 pixel correlator comprises two 31 pixel correlators plus one reconfigurable adder

Figure 9. How template pixel values correspond to reconfigurable gates

Performance Measures

Speed of Operation

There are three possible modes of operation.

1. XC6216 as part of a purpose built hardware system for image processing. XC6216 would reside on a board with frame grabber and control logic. Data input would be via the array pins. According to timing analysis of the design using XACT6000, the maximum clock speed to the design is 50MHz if data is fed direct to IOBs. This would enable correlation of a 512x512 image in 5.2ms; a maximum frame rate of 190Hz.

2. A system where data is fed to the XC6216 design via the microprocessor interface, from local memory. Speeds of 20-25MHz would be attainable giving a worst case correlation time for a 512x512 image of 13ms. This equates to a frame rate of 76Hz.

3. As part of a development system where image data is stored on the PC. Timing is dependent on PCI bus performance and the control software.

Actual values measured from the software control program give correlation times of 0.69s using the XC6200. The software control and PCI interface increase the delay from the calculated value. This compares with a value of 38s for a software correlator running on a 133MHz pentium. To improve speed, *Hit* could be wired to an interrupt signal and image data could be stored in the on-board RAM.

Speed of Initialization

In performing reconfiguration, time taken is dependent on the method of reconfiguration. In the most simple method, also the most time consuming, each address/data pair is written individually from the control program. It is also possible to perform batch reconfiguration.

Total time taken to change the mask and match image for this design is given by the following equation:

$$\boxed{T = G \times 2\,Nclk \times {}^{1}\!/_{Fclk}}$$

Where T is the total reconfiguration time, G is the number of gates, $Nclk$ is the number of Clock cycles taken for a write of one address/data pair and $Fclk$ is the frequency of the clock.

$$\boxed{T = 512 \times 2 \times 5 \times {}^{1}\!/_{33\times 10^{6}} = 0.155ms}$$

Performance with multiple templates

Given that correlator reconfiguration with a 32x16 mask and match template requires less than 0.2ms and correlation over a 512x512 image takes approximately 30ms, reconfiguration time between templates can be regarded as insignificant.

Summary

For the problems of image correlation: large amounts of data, intensive computation and the desire, in most applications, to frequently change mask and match images two advantages of XC6200 are clear. It has both the flexibility of rapid partial reconfiguration for changing between image templates, times of less than 0.2ms calculated; with high density on this kind of structured, heavily pipelined application.

Durations of 30ms are obtainable for full correlation of images of dimension 512x512 pixels.

References

[1] XAPP063: "Interfacing the XC6200 to a Microprocessor", Bill Wilkie, 1996 Xilinx Scotland.
[2] "XC6200 Reconfigurable Programmable Logic Family" Datasheet, v1.10, 1997 Xilinx Scotland.

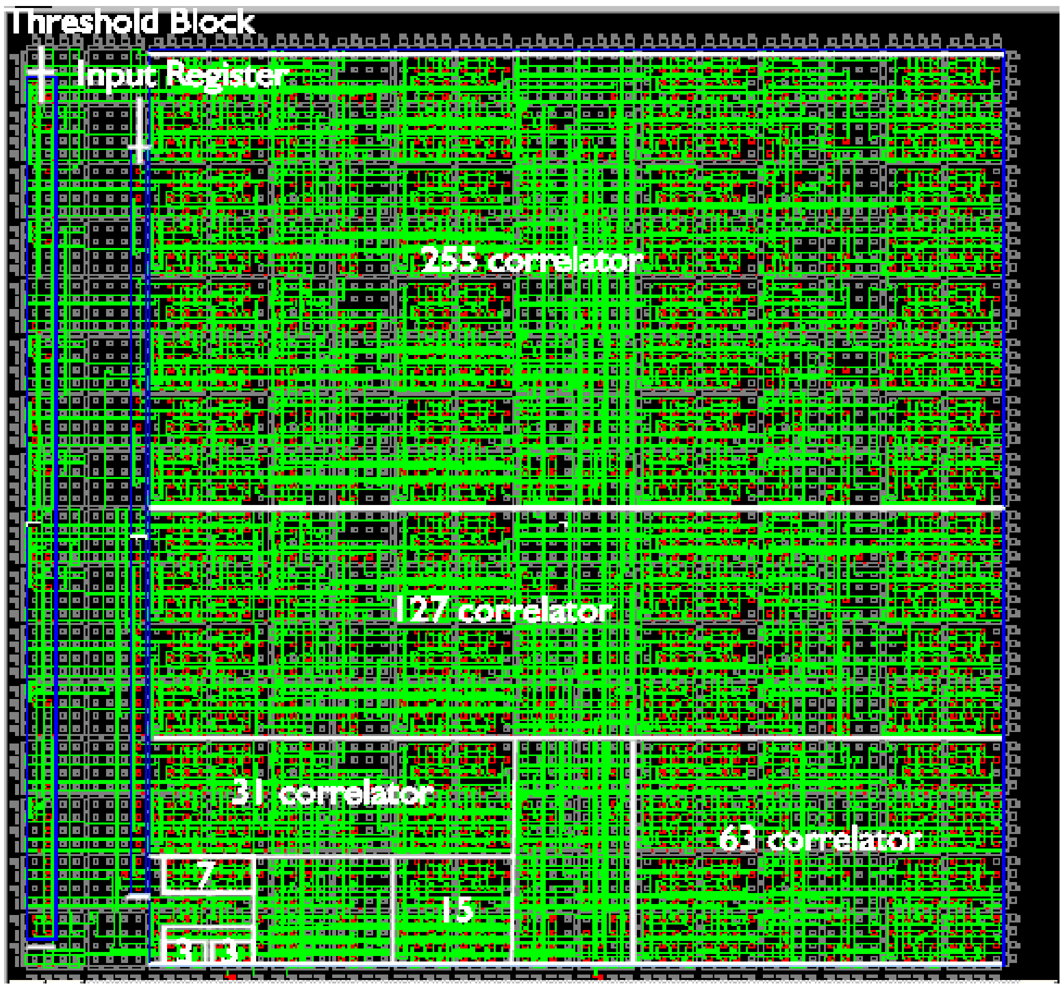
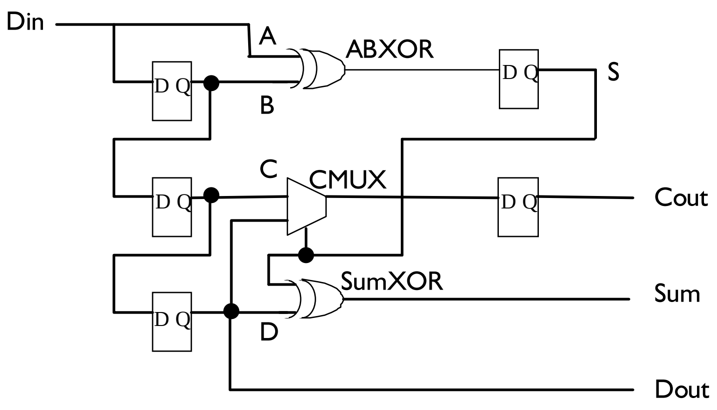


Figure 10. Physical layout of the correlator design in XACT6000

Appendix A

Gate reconfigurations depending on match and mask values

Gate	Condition	Reconfiguration	Net connections
ABXOR	Match A = Match B	XOR2	A, B
	Match A != Match B	XNOR2	A, B
	Match A = 1, B Masked	BUF	A
	Match A = 0, B Masked	INV	A
	Match B = 1, A Masked	BUF	B
	Match B = 0, A Masked	INV	B
	A, B Masked	GND	
RCMUX	Match C, Match D = 1	M2_1	C, D, S
	Match C = 0, Match D = 1	M2_1B1	Cinv, D, S
	Match C = 1, Match D = 0	M2_1B1B	C, Dinv, S
	Match C, Match D = 0	M2_1B2	Cinv, Dinv, S
	Match C = 1, D Masked	AND2B1	C, S
	Match D = 1, C Masked	AND	S0, D1
	Match C = 0, D Masked	AND2B2	Cinv, Sinv
	Match D = 0, C Masked	AND2B1	S, Dinv
	C, D Masked	GND	
SUMXOR	Match D = 1	XOR2	D, S
	Match D = 0	XNOR2	D, S
	D Masked	BUF	S



Appendix B

```
int image_correlate(Image *image)
// Locate match and mask template in image using defined correlator

{
    CorrelatorBoard  cor;   //Initialise a correlator, current definition
          //sets up XC6200 design and registers
    //CorrelatorSim  cor;
    //CorrelatorSoft cor;

    unsigned thresh =239;   //Set a threshold value for an acceptable
          //match

    cor.rows=image->_rows;     //Initialise image dimensions
    cor.cols=image->_cols;

    cor.set_match_and_mask(); //Reconfigures custom adders according to
          //template

    cor.set_threshold(thresh);//Set acceptance threshold

    cor.correlate(image);   //correlate the image with the correlator
          //template

    return 0;

}  /* end image_correlate() */

/************************************************************************/
// The hardware correlation classes overide some of the methods of
// this base class.
class Correlator{
public:
   char match_image[32][16],mask_image[32][16];
   int rows,cols;
   int threshold;

   void set_match_and_mask();       //Set up arrays mask_image and match_image
   void set_threshold(int t){threshold=t;}
   Bool correlate(char **image);
};

/************************************************************************/
// This abstract class contains common methods and data structures
// for the hardware simulation and implementation classes.
class CorrelatorHard: public Correlator {
public:
  CorrelatorRes resources;   // Access to named resources in the schematic
  CList<RCAdder *,RCAdder *> full_adders;
  CList<RCOneInAdder *,RCOneInAdder *> cin_adders;

  void build_correlator();
  void set_threshold(int t)
  {
     Correlator::set_threshold(t);
     // The hardware comparator works by generating a carry in
     // a 9 bit addition rather than doing a subtraction.
     // (val>=t) iff (val+512-t>511)
     resources._threshold->set_map_reg();
     (resources._threshold)->write((unsigned long) 512-t);
  } // set_threshold
  void set_match_and_mask()
  {
    POSITION p;
    // Set up arrays in Correlator structure.
    Correlator::set_match_and_mask();

    // set_match_and_mask refer to arrays in correlator structure.
    // for each full adder reconfigure gates according to mask and
    // match data
        for(p=full_adders.GetHeadPosition();
        p!=NULL;full_adders.GetNext(p))
           (full_adders.GetAt(p))->set_match_and_mask(this);
    // for each carry-in adder reconfigure gates according to mask and
    // match data
    for(p=cin_adders.GetHeadPosition();
        p!=NULL;cin_adders.GetNext(p))
           (cin_adders.GetAt(p))->set_match_and_mask(this);
  }
  void clock(int cycles);
  Bool correlate(Image*);

  virtual RCAdder *new_RCAdder(int ix,int iy,CString name)=0;
  virtual RCOneInAdder *new_RCOneInAdder(int ix,int iy,CString name)=0;
};
// Implements the hardware correlator on the board.
class CorrelatorBoard: public CorrelatorHard{
public:
   CorrelatorBoard(){
       build_correlator();
       resources.initialiseBoard();

       //set up access to array elements, using symbol table data
         resources._data= new AccessRegisterBoard("DATA",&resources);
       resources._threshold= new AccessRegisterBoard("THRESH",&resources);
       resources._hit=    new AccessRegisterBoard("HIT",&resources);
       resources._clr=        new AccessRegisterBoard("CLR",&resources);
   }

   RCAdder *new_RCAdder(int ix,int iy,CString name)
     {return new RCAdderBoard(ix,iy,name);}
   RCOneInAdder *new_RCOneInAdder(int ix,int iy,CString name)
     {return new RCOneInAdderBoard(ix,iy,name);}
};

/**************************************************************************/
```

Limitations And Restrictions