Virtual Prototyping for Dynamically Reconfigurable Architectures using Dynamic Generic Mapping

D. J. Gibson, M. Vasilko & D. Long
School of Design, Engineering & Computing,
Department of Electronics,
Bournemouth University,
Fern Barrow, Poole, Dorset. UK
gibsond@bournemouth.ac.uk

Abstract

This paper presents a virtual prototyping methodology for Dynamically Reconfigurable (DR) FPGAs. The methodology is based around a library of VHDL image processing components and allows the rapid prototyping and algorithmic development of low-level image processing systems. For the effective modelling of dynamically reconfigurable designs a new technique named, Dynamic Generic Mapping is introduced. This method allows efficient representation of dynamic reconfiguration without needing any additional components to model the reconfiguration process. This gives the designer more flexibility in modelling dynamic configurations than other methodologies. Models created using this technique can then be simulated and targeted to a specific technology using the same code. This technique is demonstrated through the realisation of modules for a motion tracking system targeted to a DR environment, RIFLE-62.

1. Introduction

The recent introduction of FPGAs that can be dynamically reconfigured has pre-empted research into various applications. However, very little CAD tool support exists for these devices at present [1],[2]. Therefore, designers are currently forced to use conventional tools and work around the dynamic reconfiguration manually. A number of signal processing applications have been targeted to DR architectures using existing tools [3],[4]. These have been designed at a low-level with the partitioning into consecutive dynamic configuration performed manually. However, the more complex the system the less obvious it becomes where and how the reconfiguration should take place. This paper proposes the use of a modified rapid prototyping methodology for the realisation of low-level image processing systems on DR architectures. The methodology promotes early characterisation of the design and speeds up the lengthy algorithmic development of image processing systems. Moreover, it can be used with existing CAD tools. This methodology has been used in conjunction with a prototyping environment for Dynamically Reconfigurable Logic (DRL), to develop a real-time movement detection system.

2. RIFLE-62: DRL Prototyping Environment

RIFLE-62 is an FPGA based prototyping board that has been developed for the rapid prototyping of dynamically reconfigurable logic. The board has a flexible architecture based around the Xilinx XC6200 family of dynamically reconfigurable FPGAs. In addition, the board has two other FPGAs (XC4013E and XC3100A), static and dynamic memory, a 32-bit data bus, a 24-bit address bus, control signals, dual clocks and four dedicated interfaces [5]. A block diagram of the RIFLE-62 architecture is shown in Figure 1.

This board provides a flexible platform for the prototyping of dynamically reconfigurable applications. With RIFLE-62 it is possible to experiment with different reconfiguration strategies, such as various sizes of temporarily stored dynamic configurations or different configuration control schemes.

The fast configuration cache allows partial reconfiguration of the XC6200 at full speed. The reconfiguration overhead can be minimised to permit realistic estimation of system performance. The three FPGA architectures on the board suit the requirements of most logic circuits, whilst the large memory storage provides sufficient capacity and throughput for the implementation of real-time image processing tasks.

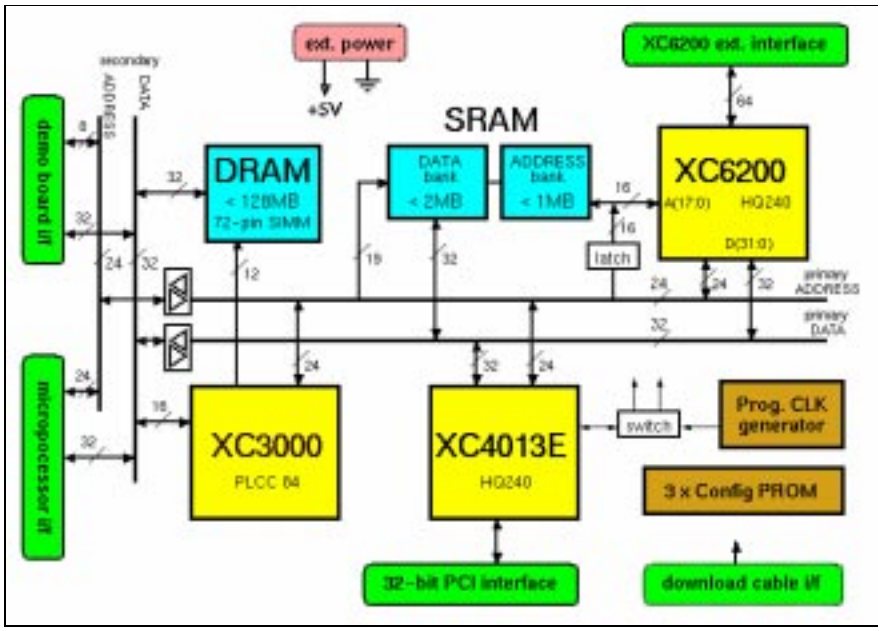

Figure 1 Block Diagram of RIFLE-62 (DRL) Prototyping Board

3. Dynamically Reconfigurable Real-Time Motion Detection

To demonstrate the design flow and the capabilities of RIFLE-62, a real-time motion detection system has been modelled and targeted to this environment. This application operates on an input data-stream from a video-camera, processes the data on the RIFLE-62 board and outputs the results to a video monitor. The system uses an Audio/Video interface board developed for the RIFLE-62. This unit has a multi-standard video processor that digitises an input signal and a video RAMDAC capable of translating a digital input to an analogue output.

The motion detection system consists of three functions: median filtering, edge detection and frame differencing. These were selected to represent three different categories of low-level image processing operation, each with different requirements on processing speed, memory and configurable logic area. A schematic diagram of the motion detection system is shown in Figure 2.

The median operation is a non-linear filter that is ideal for removing impulse noise from images. The algorithm replaces the grey scale value of each pixel with the median value of the neighbouring pixels. These values are taken from an odd sized window around the current pixel. This filter offers good low-pass performance whilst preserving the sharpness of edges within the image [6]. The hardware realisation of a median filter is challenging as it requires a rank ordering unit to sort the pixel values. Sorting nine pixel values requires a total of thirty-six compare functions. Therefore, a fast heavily pipelined design is required.

The edge detection is realised as two linear convolutions using the Sobel operators. This highlights changes in grey scale values in the vertical and horizontal directions and can be used to perform estimation of an edge's magnitude and phase angle [7]. The Sobel operators are attractive for a hardware realisation as the mask coefficients are all powers of two. This eliminates the need for hardware multipliers as binary shifts will suffice. As the two convolutions are performed concurrently the immediate implication is that a sequential structure could be utilised with resource sharing by reconfiguration.

Finally, frame differencing is performed between consecutive images to locate moving objects. Although the computational requirement for this unit is fairly trivial the storage requirement is high as an entire image must be stored. Therefore, fast access time memory is required together with an effective memory management strategy.

Partial dynamic reconfiguration is used to dynamically allocate operations within the required area of the XC6200. The design is heavily pipelined in order to provide the required throughput for real-time operation. A partial reconfiguration technique known as *pipeline morphing* could be exploited to minimise the total configuration overhead [8].

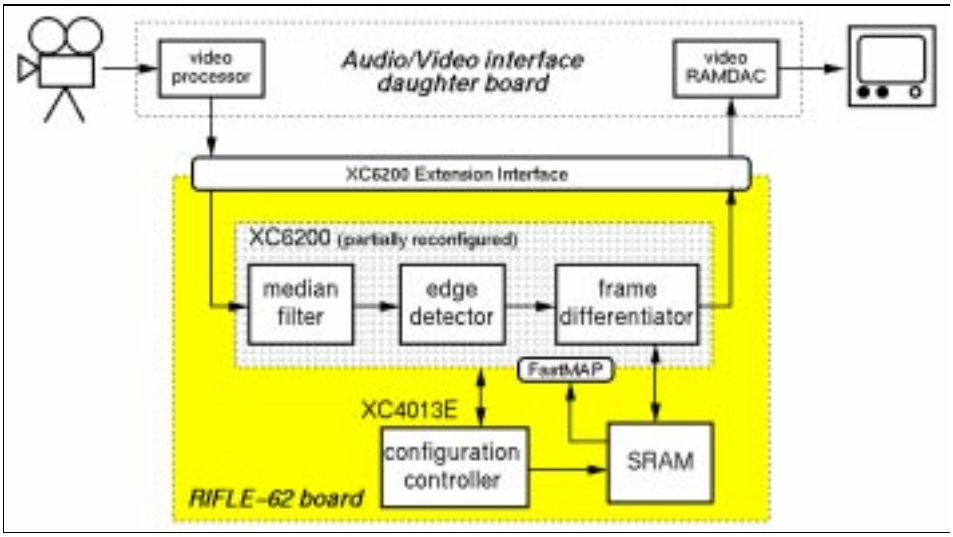

Figure 2 Schematic diagram of the Motion Detection System on RIFLE-62

4. Virtual Prototyping Methodology

Using this presented virtual prototyping methodology, high-level design is performed using a pre-designed VHDL component library [9]. This library contains models of basic arithmetic functions as well as specific image processing components. All the components are fully generic in terms of algorithmic implementation, size and degree of pipelining. The library allows designers to develop the required algorithms from the system specifications. This is achieved by creating an abstract 'virtual prototype' of the system from the components contained in the library [9]. As far as the designer is concerned the components are functional 'black boxes'. These boxes are used to construct and verify the hardware implementation of a desired algorithm.

Having established the algorithmic realisation, the designer should next consider whether any dynamic reconfiguration is required within the system. This includes not only areas where it is possible to dynamically reconfigure the design, but also which areas can afford to be reconfigured. Using the presented methodology this is achieved by generating sub-components for elements that are to be reconfigured. These should be parameterised with generics in terms of not only performance, but also reconfiguration. For example, Figure 3 shows an entity declaration of a coefficient multiplier module that will be dynamically reconfigured.

```
Library IEEE;
USE IEEE.std_logic_1164.ALL;
USE WORK.convolverpack.BusSize;

ENTITY Coefficient_Multiplier IS
 GENERIC ( GreyScale : INTEGER := 256;              -- No. of grey scales per pixel
           CoefficientRange : INTEGER := 5;         -- Value of largest coefficient size
           Coeff_1 : INTEGER := 3;                  -- Value of coefficient one
           Coeff_2 : INTEGER := 5;                  -- Value of coefficient two
           Coefficient_1_2 : BOOLEAN := FALSE;      -- Coefficient 1 or coefficient 2
 PORT (PixelIP : IN std_logic_vector(((BusSize(GreyScale))-1) DOWNTO 0);  -- Pixel input to coefficient multiplier
       Clk : IN std_logic;                  -- Pipeline clock
       Rst : IN std_logic;                  -- Pipeline Reset
       ResultOP : OUT std_logic_vector(((BusSize(GreyScale*CoefficientRange))-1) DOWNTO 0)); -- Resulting output
END Coefficient_Multiplier;
```

Figure 3 Entity for a Coefficient Multiplier using a Generic Reconfiguration strategy

In this example the reconfiguration of the coefficient multiplier is dependent upon the generic parameter *Coefficient_1_2*. This generic has been given a data type of Boolean. When the generic has a value of *false* the unit will perform multiplication between the input pixel and the first coefficient (*coeff_1*). Similarly when the generic has a value of *true* the second coefficient (*coeff_2*) is used. This reconfiguration generic allows simulation of the module by effectively switching between two realisations. This method offers a more flexible mechanism for controlling the configurations than Dynamic Circuit Switching (DCS) [1], as no extra components are required in the design to model the reconfiguration. Therefore, it is possible to simulate a design and directly synthesise partial configurations from the same code. In addition, the reconfiguration generics can be used for more than just switching circuits. For example, a counter's maximum value could be controlled by a generic and the realisation reconfigured depending upon this value. In addition, the use of generics removes the configuration control scheme from the design under test and allows it to be considered at the next level of design hierarchy.

A generic map in VHDL is a construct that allows the communication of static values to a specific instance of a component. These values are evaluated at elaboration of the component and are therefore globally static. The nature of dynamic reconfiguration requires a method for modifying the internal parameters of a component from an external source. As the reconfiguration is controlled by generic values a strategy is required for the dynamic modification of these values. This is achieved using the presented technique named, *Dynamic Generic Mapping*, which allows generic values for a specific instance of a component to be changed at any point in simulation time.

5. Dynamic Generic Mapping

In the presented methodology the *Dynamic Generic Maps* (DGM) have been implemented manually using a combination of **ASSERT** statements and (QuickHDL) simulator commands. Figure 4 shows an example of a Dynamic Generic Map.

```
ARCHITECTURE reconfigurable OF convolve IS
 TYPE b2s_type IS ARRAY (BOOLEAN) OF STRING(1 TO 5);
 CONSTANT Boo2Strg : b2s_type := (TRUE =>  "TRUE ",
                                  FALSE => "FALSE");
 SIGNAL reconfigure : BOOLEAN;

 COMPONENT Coefficient_Multiplier
  GENERIC ( GreyScale : INTEGER := 256;          -- No. of grey scales per pixel
            CoefficientRange : INTEGER := 5;     -- Value of largest coefficient size
            Coeff_1 : INTEGER := 3;              -- Value of coefficient one
            Coeff_2 : INTEGER := 5;              -- Value of coefficient two
            Coefficient_1_2 : BOOLEAN := FALSE); -- Coefficient 1 or coefficient 2
  PORT (PixelIP : IN std_logic_vector(((BusSize(GreyScale))-1) DOWNTO 0); -- Pixel input to coefficient multiplier
        Clk : IN std_logic;              -- Pipeline clock
        Rst : IN std_logic;              -- Pipeline Reset
        ResultOP : OUT std_logic_vector(((BusSize(GreyScale*CoefficientRange))-1) DOWNTO 0)); -- Resulting output
 END COMPONENT;
BEGIN

 CM1 : Coefficient_Multiplier
  GENERIC MAP (Greyscale => 256,
               CoefficientRange => 5,
               Coeff_1 => 2,
               Coeff_2 => 5,
               Coefficient_1_2 => reconfigure)
  PORT MAP (PixelIP => Data_IN,
            Clk => Clk,
            Rst => Rst,
            ResultOP => Data_OUT);

  ASSERT NOT (reconfigure'EVENT)
  REPORT "Update the value of the generic COEFFICIENT_1_2 to " & Boo2Strg(reconfigure)
  SEVERITY NOTE;

   •
   •
   •
END reconfigurable;
```

Figure 4 Example of manual realisation of a Dynamic Generic Map (DGM)

In this example the reconfiguration generic, *Coefficient_1_2* is assigned to the value of a control signal, *reconfigure*. When an event occurs on this signal the assert statement is activated and a severity level note is issued to the simulator. This note requests that the value of the reconfiguration generic is updated. With the simulator set to break upon a severity note the simulation is halted when the signal *reconfigure* is updated. Then a simulator command can be issued to modify the generic value. This method allows the dynamic updating of the reconfiguration generics and hence the modelling of dynamic reconfiguration. It should be noted that upon initialisation the generic *Coefficient_1_2* is assigned the default value of the *reconfigure* signal's data type (*false*). Therefore, this value should be equivalent to the first required configuration. Having established the DGM the architecture can then model the reconfiguration strategy and any additional functionality of the module. Hence, it is possible to establish estimates of the reconfiguration overheads such as resource utilisation, I/O requirements and the allocation of memory. The component *Coefficient_Multiplier* can then be synthesised for every value of the reconfiguration generic. As synthesis tools consider the generic values to be globally static, all statements that dependent on other values of the reconfiguration generics will be removed by optimisation. This allows estimation of the hardware realisation for different configurations.

6. Realisation Estimates

The *Dynamic Generic Mapping* technique has been used in the virtual prototyping of the motion detection system outlined in Section 3. This has allowed the assessment of various reconfigurations and the hardware overheads. For example, the edge detector possesses a threshold unit that determines if the convolved pixel value is representative of an edge in the resulting image. This is realised as a unit that compares the current convoloution output with a given threshold. When this value is greater than the threshold it is considered to be an edge and the output pixel is set to the maximum grey scale value. In all other cases the pixel is set to the minimum grey scale value. This results in a black and white image from the edge detector with the edges highlighted in white. For accurate results from a real-world scene it is desirable to have between 9-11% of pixels in the resulting image being representative of edges. This can be achieved using an adaptive algorithm that counts the number of edge pixels identified in the previous image [6]. If the total number of edges is less than 9% the threshold value is decremented by one. If the value is greater than 11% the threshold is incremented by one. A pseudo code representation of this adaptive algorithm is shown in Figure 5.

```
Threshold := Thresholding value

BEGIN
        Count = No. of edge pixels in the previous image
        IF Count <= 9% of total image size THEN
                Threshold = Threshold - 1
        ELSE IF Count >= 11% of total image size THEN
                Threshold = Threshold + 1
        ELSE
                Threshold = Threshold
        END IF
        Threshold next image
END
```

Figure 5 Pseudo code representation of adaptive threshold algorithm

This adaptive algorithm will maintain an edge count of between 9-11% in the resulting image irrespective of changes in ambient lighting. Due to the adaptive nature of this algorithm it is suited to a realisation using dynamic reconfiguration. From the component library available a basic threshold unit exists. The component declaration for this unit is shown in Figure 6. This component purely performs the thresholding operation between the pixel value and the threshold, which is represented by the generic *ThresholdVal*. Therefore, in the realisation of a reconfigurable adaptive threshold module this sub-component is required and *ThresholdVal* becomes the reconfiguration generic. When the value is changed using the DGM it represents a new configuration with a different threshold value. Figure 6 shows the VHDL model of the adaptive threshold unit. In this example the threshold component is instantiated and the reconfiguration generic is assigned to the value of the signal *Adapt_thres*. The value of this signal is modified as represented by the pseudo code (shown in Figure 5), at the start of each new image frame. Note that this value is also an output from the module via the port *Reconfigure*. A change in this value represents a new configuration and all that is necessary externally is an address decoder to enable the new configuration to be read from memory. On the RIFLE-62 the address decoder could be implemented easily in the XC4013E device.

```
Library IEEE;
USE IEEE.std_logic_1164.ALL;
USE WORK.convolverpack.BusSize;

ENTITY Threshold_dyn_config IS
  GENERIC ( GreyScale : INTEGER := 256;          -- No. of grey scales in image
            ImageSize : INTEGER := 256;          -- Size of image (ImageSize x ImageSize)
            MaskSize  : INTEGER := 3;            -- Size of convolver mask (Masksize x MaskSize)
            CoefficentRange : INTEGER := 2;      -- Value of largest coefficent size
            ThresholdVal : INTEGER := 12);       -- images threhold value
  PORT ( ThresIP : IN std_logic_vector (((BusSize(((GreyScale*CoefficentRange)-1)*(MaskSize**2)))) DOWNTO 0);     -- Data input
         ThresClk : IN std_logic;
         ThresRst : IN std_logic;
         ThresEdgeIP : IN std_logic;                -- Input from edge counter entity
         Frame : IN std_logic;
         Reconfiguration : OUT INTEGER RANGE 0 TO (greyscale-1);
         ThresOP : OUT std_logic_vector (((BusSize(GreyScale))-1) DOWNTO 0));   -- Data output
END Threshold_dyn_config;

ARCHITECTURE Behave_adapt OF Threshold_dyn_config IS
  SIGNAL Adapt_thres : INTEGER RANGE 0 TO (greyscale-1);
  SIGNAL Edge_Cnt : INTEGER RANGE 0 TO ((Imagesize*Imagesize)-1);
  SIGNAL Cnt_rst : std_logic;
  SIGNAL Output : std_logic_vector (((BusSize(GreyScale))-1) DOWNTO 0);

  COMPONENT Threshold
    GENERIC ( GreyScale : INTEGER := 256;          -- No. of grey scales in image
              ImageSize : INTEGER := 256;          -- Size of image (ImageSize x ImageSize)
              MaskSize  : INTEGER := 3;            -- Size of convolver mask (Masksize x MaskSize)
              CoefficentRange : INTEGER := 2;      -- Value of largest coefficent
              ThresholdVal : INTEGER := 12);       -- images threhold value
    PORT ( ThresIP : IN std_logic_vector (((BusSize(((GreyScale*CoefficentRange)-1)*(MaskSize**2)))) DOWNTO 0); -- Data input to threshold component
           ThresClk : IN std_logic;
           ThresRst : IN std_logic;
           ThresEdgeIP : IN std_logic;  -- Input from edge counter entity
           Frame : IN std_logic;
           ThresOP : OUT std_logic_vector (((BusSize(GreyScale))-1) DOWNTO 0)); -- Data output from threshold component
  END COMPONENT;

BEGIN

Reconfiguration <=Adapt_thres;
ThresOP <= Output;

  TH1 : Threshold
    GENERIC MAP (GreyScale => 256,
                 Imagesize => 256,
                 MaskSize => 3,
                 CoefficentRange => 2,
                 ThresholdVal => Adapt_thres)
    PORT MAP (ThresIP => ThresIP,
              ThresClk => ThresClk,
              ThresRst => ThresRst,
              ThresEdgeIP => ThresEdgeIP,
              Frame => Frame,
              ThresOP => Output);

ASSERT NOT (Adapt_thres'EVENT)
    REPORT "Update the value of the generic ThresholdVal in the threshold component to equal the ADAPT_THRES signal"
    SEVERITY NOTE;

  Edge_counter : PROCESS(ThresClk, ThresRst, Cnt_rst, Output, Edge_Cnt)   -- Process modelling the edge pixel counter
  BEGIN
    IF ThresRst = '1' OR Cnt_rst = '1' THEN
      Edge_cnt <= 0;
    ELSIF ThresClk'EVENT AND ThresClk = '1' THEN
      IF Output(0) = '1' THEN
          Edge_cnt <= Edge_cnt +1;
      END IF;
    END IF;
END PROCESS Edge_counter;

Change_Threshold : PROCESS (Cnt_Rst, Edge_Cnt)                          -- Process modelling the changing of the threshold value
  BEGIN
  IF Cnt_Rst'EVENT AND Cnt_Rst = '1' THEN
     IF Edge_Cnt >= (((Imagesize*Imagesize)/100)*11) THEN
          Adapt_thres <= Adapt_thres+1;`
     ELSIF Edge_Cnt <= (((Imagesize*Imagesize)/100)*9)  THEN
          Adapt_thres <= Adapt_thres-1;
     ELSE
          Adapt_thres <= Adapt_thres;
     END IF;
   END IF;
  END PROCESS Change_threshold;

  Reset_counter : Process (ThresRst, Frame, Cnt_rst)                    -- Process modelling the counter reset controller
  BEGIN
    IF ThresRst = '1' OR Cnt_rst = '1' THEN
      Cnt_rst <= '0';
    ELSIF Frame'EVENT AND Frame = '1' THEN
      Cnt_rst <= '1';
    END IF;
  END PROCESS Reset_counter;

END Behave_adapt;
```

Figure 6 VHDL model of the adaptive threshold unit using Dynamic Generic Mapping

	Library Threshold (non adaptive)	Adaptive threshold (Non-reconfigurable)	Dynamically Reconfigurable Adaptive threshold
XC6264 Primitive Cells	**25**	**248**	**91**
I/O Ports	**25**	**25**	**33**
External configuration cache Memory	-	-	**64×32 Bits**

Table 1 Threshold units hardware realisation

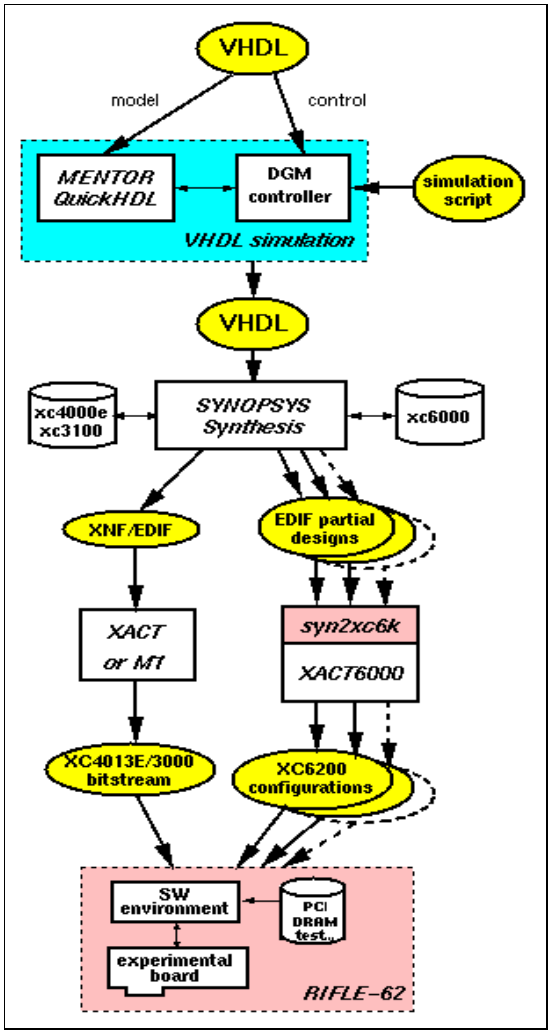

Figure 7 Virtual Prototyping Design Flow

To assess the hardware realisation of the threshold module, synthesis was performed using the prototyping design flow, shown in Figure 7. For comparison purposes synthesis was undertaken on the non adaptive threshold model from the image processing library [9], a dynamic reconfigurable adaptive threshold unit and a realisation of the adaptive threshold module without using reconfiguration. These results are shown in Table 1.

The non-reconfigurable realisation requires 248 primitive cells and 25 ports. Whereas the dynamic implementation utilises 91 primitive cells and 33 ports. The dynamic system saves 157 cells in the reconfigurable FPGA as logic is not required to modify the threshold value. However, the logic overhead required for the instantiated threshold unit and the part of the algorithm that determines the new threshold value remains constant. This design also requires an extra 8-bit port, 64×32 bits of memory and an address decoder, (not shown in Table 1). It should also be noted that the memory requirement for the partial configurations is dependent upon the placing of the design within the device. This example has demonstrated that the use of dynamic reconfiguration has shifted the resource utilisation from the XC6264 to other resources on the RIFLE-62 board.

7. Conclusions

In this paper it has been shown how *Dynamic Generic Mapping* can be used in the virtual prototyping of DR systems. This technique allows reconfigurable components to be modelled and then integrated within larger systems. Using generics to parameterise reconfigurable elements gives a number of advantages: Firstly, the dynamic model can be kept completely separate from the configuration control scheme and no extra components are required to model the reconfiguration. Secondly, generics offer greater flexibility in the parameterisation and modelling of reconfiguration. Finally, the Dynamic Generic Map (DGM) allows various configuration strategies to be evaluated and the hardware requirements for each established.

After generation of the reconfigurable components the DGM permits the simulation of a system containing any number of dynamic elements. The same model can then be synthesised for all the possible values of the

reconfiguration generics. As these values are globally static during synthesis any elements that are dependent upon other values of the reconfiguration generic will be optimised. This will generate the circuits required for each partial configuration.

As further work it is intended to automate the implementation of the DGMs. This could be achieved by extending VHDL, adding two extra constructs to the language. These being a *Dynamic Generic List* and a *Dynamic Generic Map*. The introduction of these new clauses will allow the modelling of dynamically reconfigurable logic, whilst leaving the standard constructs in place for ordinary model parameterisation.

Acknowledgements

The authors would to thank the Xilinx Incorporated for the beta release of XACT6000 and the Synopsys libraries.

References

[1] Stockwood, J., and P. Lysaght, 'A Simulation Tool for Dynamically Reconfigurable Field Programmable Gate Arrays.' *ASIC '95*, Texas, Sept. 1995.

[2] Luk, W., N. Shirazi & P. Y. K. Chung, 'Compilation Tools for the Run-Time Reconfigurable Design.' *FCCM97*, K. L. Pocek & J. Arnold (eds.), IEEE Computer Society Press, 1997.

[3] Brebner, G., & J. Gray 'Use of Reconfigurablity in Variable-length Code Detection at Video Rates.' *Field Programmable Loic and Applications*, W. Moore & W. Luk (eds.), LNCS 975, Springer. 1995.

[4] Lysaght, P., & H. P. Dick, 'Implementation of Adaptive Signal Processing Architectures Based on Dynamically Reconfigurable FPGAs.' Proceedings of *EUSIPCO-94*, Edinburgh, Scotland, Vol III.

[5] Vasilko, M., & D. Long, 'RIFLE-62: A Flexible Environment for Prototyping Dynamically Reconfigurable Systems.' *9th IEEE International Workshop on Rapid System Prototyping*, 1998.

[6] Gonzalez, R., & R. Woods *Digital Image Processing.* Addison-Wesley. 1992

[7] Pratt, W., *Digital Image Processing (Second Edition).* John Wiley & Sons, New York, 1991.

[8] Luk, W., N. Shirazi, S. R. Guo & P. Cheung, 'Pipeline Morphing and Virtual Pipelines.' Field Programmable Logic and Applications (FPL'97 Proceedings), LNCS 1304. Springer-Verlag, 1997.

[9] Gibson, D. J., M. K. Teal, D. Ait-Boudaoud & M. Winchester 'A New Methodology and Generic Model Library for the Rapid Prototyping of Real-Time Image Processing Systems.' *VIUF Rapid Systems Prototyping with VHDL Conference*, Fall 1997.