Generic Model Library for Rapidly Prototyping Real-Time Two-Dimensional Convolvers

D. J. Gibson, M. K. Teal,
S. Holloway & M. Winchester*
School of Design, Engineering & Computing, Department of Electronics,
Bournemouth University,
Fern Barrow, Poole, Dorset. UK
e-mail: gibsond@bournemouth.ac.uk
*British Aerospace Dynamic Division, Stevenage, Hertfordshire. UK

Abstract

The design and implementation of real-time image processing systems is difficult and often time consuming. This can lead to long 'time to market' delays which incur high cost penalties, something most engineers would rather avoid. This paper proposes the development of a library containing generic hardware models that allow rapid prototyping, verification and realisation of these systems. These models can be used by both algorithmic developers and hardware designers and will permit the rapid assessment of hardware realisations. This smoothes the design flow process and eliminates errors occurring during the high-level to hardware implementation transfer of the design. A library has been created containing generic components that can be used for the realisation of image convolvers. This paper documents the design of the basic blocks required for the implementation of these systems.

1. Introduction

The implementation of image processing systems is normally performed on very high speed processing units. However, before the image information is in a form that is acceptable for processing it will often require some form of low-level manipulation. These low-level or pixel-based operations tend to be realised in custom hardware as the algorithms are executed on all the image pixels, and as such are computationally intensive. One of the most widely used techniques for pixel-based manipulation is two-dimensional image convolution. This operation may be used to produce a wide variety of different results depending on the filter coefficients chosen [1]. Although an image convolver performs the same basic operation for each filter implementation, the design of its hardware is nontrivial. For example, a low-pass filter can be achieved using a simple averaging mask, whereas edge detection requires a more complex mask. Different masks require different architectures for the convolver realisation. This means that developers tend to design such hardware from scratch and rely on previous experience as a guide. This is often a very 'hit or miss' affair. Moreover, the development of these systems is undertaken at two levels in a normal design cycle. First, the system engineer will establish the required algorithm for the particular application. Often this is performed with very little regard for the actual hardware implementation that will be required. When the design is subsequently passed to the hardware developers, they have to try to interpret the algorithms as hardware. This is often difficult and close liaison between the two will be required until a compromise solution is met. This is a time consuming process and can lead to errors occurring between the specifications and final hardware.

The use of Hardware Descriptive Languages (HDL's) for modelling digital hardware has revolutionised design techniques. Over the last decade VHDL has become one of the most popular standards for digital design entry. It allows hardware to be modelled at various levels of abstraction: these models can then be simulated and synthesised to a particular technology. Several features of VHDL make it ideal for the realisation of a generic library.

Primarily, the language's architectural independence means the component models can have various different architectural realisations. This allows modelling at high-levels of abstraction, so that models can be created as functional 'black boxes'. Furthermore, VHDL permits models of scaleable architectures to be generated with the use of generic parameters [2].

In recent years, there has been a dramatic movement towards the production and use of standard VHDL models [3]. Due to the high throughput rates associated with the real-time operation of image processing systems, all the component models for these applications require various levels of pipelining. In addition, so that they can be easily incorporated into higher-level designs, a standard input/output (I/O) interface is required for each model [4]. For these reasons it is not possible to use 'off the shelf' component models.

2. The component model library

The components that are required can be ascertained by decomposing the specified algorithm into functional blocks. For each block a component will be desired with a standard interface. As the component model's internal structure may be subsequently modified with hardware details the interface should be in terms of hardware. For example, Figure 2-1 contains the interface requirements for an 8-bit adder. As can be seen the interface is defined in terms of the IEEE standard multi-valued logic system for VHDL model interoperability (std_logic_1164) [5]. Ports *A*, *B* and *Result* have been declared as type *std_logic_vector* and *Clk*, *Reset* and *Cout* as *std_logic*. Although the system engineers are only concerned with the development of the algorithms this style of interface will lead to early definition of the models implemented.

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;

ENTITY adder IS
    GENERIC ( bus_width : INTEGER := 8;
              granularity : INTEGER := 2);
    PORT ( A : IN std_logic_vector ((bus_width - 1) DOWNTO 0);
           B : IN std_logic_vector ((bus_width - 1) DOWNTO 0);
           Clk, Reset : IN std_logic;
           Cout : OUT std_logic;
           Result : OUT std_logic_vector ((bus_width - 1) DOWNTO 0));
END adder;
```

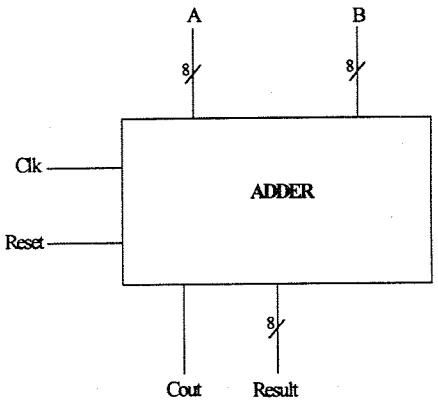

Figure 2-1 Example of component interface for an adder

All the components contained in the library are fully generic in size. This allows a single model to be used for the realisation of any bit width. This can be seen in Figure 2-1 where the width of the adder is controlled by the generic *bus_width*. Due to the high throughput nature of real-time applications it is difficult to predict the level of pipelining that a component will require. Moreover, the degree of pipelining required is partly dependent on the implementation technology chosen for the system. At the modelling stage these details may not be known. Thus, the 'granularity' or pipelining of each component is also controlled with generic constructs [6]. In addition, any algorithmic parameters that can be varied in the components will also be controlled with generic constructs.

The abstraction level that a VHDL component model is created on is dependent on the functionality of the component. For example, a storage device is easily constructed generically at a RT level. However, arithmetic components are first designed at a functional level and then have timing details added (behavioural level) before implementation details

are added later during the design cycle. This is shown more clearly in Figure 2-2 where, as an example, an adder at different stages of the design cycle is shown. As can be seen the adder starts off as a functional model and evolves into a full hardware model. First, at the algorithmic level a resolution of 8-bits is deemed to be sufficient to achieve the requirements. When it has been verified that the model meets the specifications the generic modelling of the components timing can be investigated. At the intermediate level a carry look-ahead architecture is added to the model, this can be achieved in one of two ways. Either incorporate a pre-designed ripple carry architecture from the library or modify the adder's existing architecture. Following the model's synthesis it is determined that the adder requires a *granularity* of four. Finally, during the layout it is discovered that the routing has incurred larger delays than expected and a *granularity* of two is required.

At any time during the design cycle the adder model should have the same functionality. During the high-level design each component model is regarded purely as a 'black box', thus the functionality of a model can be verified at any time during the design cycle. This is simply achieved by incorporating the component back into the overall system model [7].

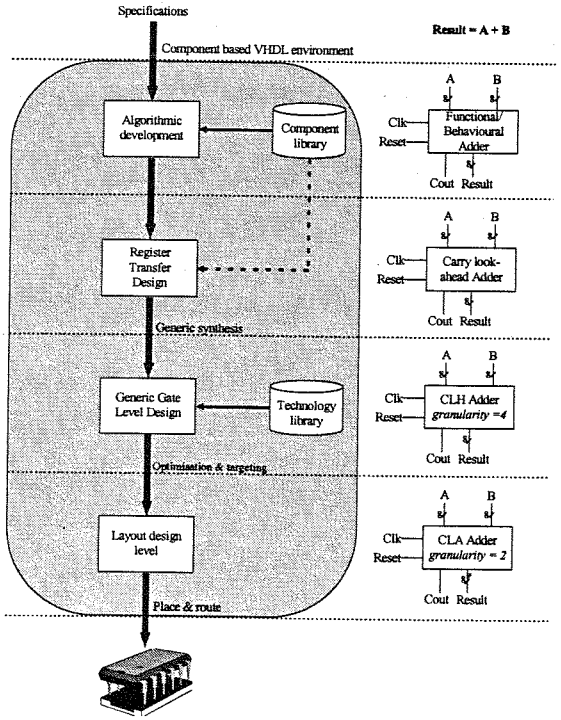

Figure 2-2 An example of an adder model throughout the rapid prototyping design cycle

3. Component library and template for image convolvers

Before the image convolver component library can be designed it is necessary to establish what components are required. This is achieved by decomposing the algorithms into functional blocks. The *centred, zero boundary superposition* convolution algorithm that has been applied to our rapid prototyping philosophy the algorithm is shown below in Equation 3-1 [1].

$$H(t_1,t_2) * I(m_1,m_2) = Q(j_1,j_2) = \frac{1}{N} \sum_{n1=0}^{L-1} \sum_{n2=0}^{L-1} T(n_1,n_2).I(j_1 - n_1 - Lc, j_2 - n_2 - Lc)$$

Equation 3-1

This equation can be decomposed into six components. A schematic diagram representing these components and their connectivity is shown in Figure 3-1. This section will detail the modelling of the component required for a convolver operation with power of two coefficients although this can be changed by using different architectural realisations.

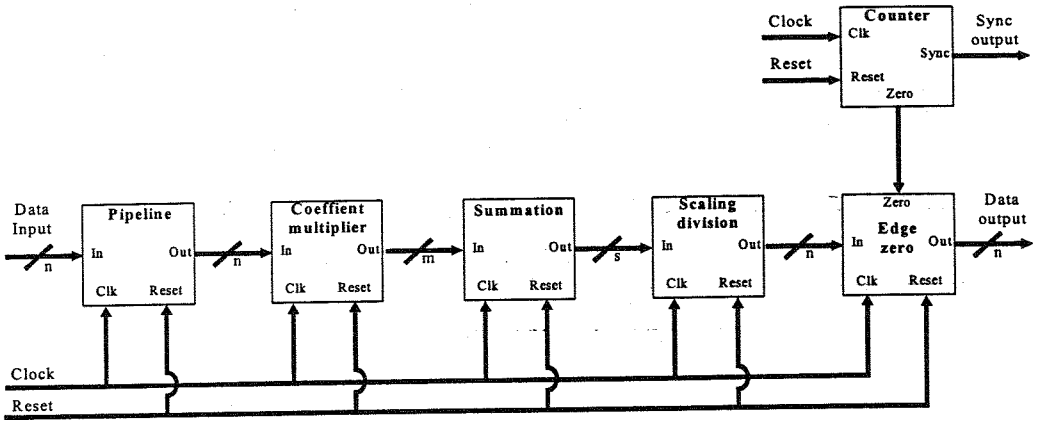

Figure 3-1 Schematic diagram representing the components required to implement an image convolver

3.1 Generic pipeline

The first component that is required in the convolver system is a pipeline. As previously stated, the input image is supplied in a raster scan form, pixel by pixel, line by line. It is necessary to have all the pixels available that correspond to the current mask position so that convolution can be performed. For example, if the mask size is 3×3 then the corresponding nine pixels of the image will be required. This can be achieved by constructing a shift register to hold the required lines of the image and supply the necessary pixels. Therefore, for each clock cycle a new pixel of image data will be shifted in and the contents of the registers will be shifted up one place. A schematic representation of a pipeline required for an image size of sixty-four pixels and a 3×3 convolution mask is shown in Figure 3-2.

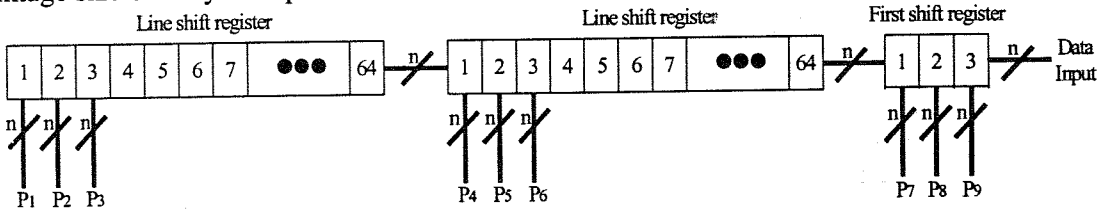

Figure 3-2 A block diagram of a pipeline for an image that is 64 pixels wide, being convolved with a 3×3 mask

The width of the input to the pipeline will be equal to a binary representation of the image grey scales and may be written as shown in Equation 3-2.

$$Width\ of\ pipeline\ input\ bus = \frac{Log\ (\text{No. of grey scales})}{Log2}$$

Equation 3-2

The width of the pipeline's output is a more complex problem. Generics within VHDL can not be used to define the number of outputs from an entity. (Note that the number of outputs is dependent on the convolution mask size.) However, generics can be used to define

a bus width. Therefore, the pipeline model should be constructed with one output bus that is made up of sub-buses. This is shown clearly in Figure 3-2, where P1 to P9 are the sub-buses each with width equal the binary grey scale of the image. These sub-buses make up the output bus from the pipeline. Thus, the width of the pipelines output bus is given below in Equation 3-3. For example, taking the system shown in Figure 3-2 and assuming that the image has 32 grey scales, then the output bus will be 45 bits wide.

$$Width\ of\ pipelines\ output\ bus = \left(\frac{Log(\text{No. of grey scales})}{Log2} \right) \times (Mask\ length)^2$$

Equation 3-3

3.2 Generic coefficient multiplier

The easiest multiplications to implement in hardware are those where the multiplier is a power of two. These may be realised as a binary left shift of the multiplicand. As many simple filter masks are made up of power of two numbers a multiplication *function* was written to implement this operation. A second *function* was written for the specific filter operator that the convolution system implements. This sub-program stores the filter mask as a constant and then calls the multiplication *function* dependent on the coefficient multiplication. For example, if the pixel value is '00110110' and the current coefficient value is 4, then the multiplication procedure will be called and the result will be '11011000'. Finally, a coefficient multiplication entity was designed that reads in the pixels that correspond to the filter mask every clock cycle. These pixel values are directly available from the output of the pipeline component. Concurrently, the filter mask function is called for all the coefficients in the mask. This procedure then shifts the pixels depending on the required coefficient multiplication. An example of the structure required to implement a typical power of two, filter mask is shown as a block diagram in Figure 3-3.

$$T = \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

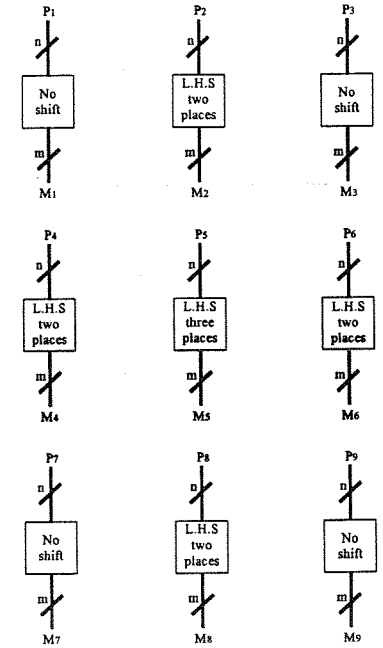

Figure 3-3 A block diagram of the coefficient multipliers for a low pass filter mask

The input bus to the coefficient multiplier entity is the same width as the output bus of the pipeline as given by Equation 3-3. As with the pipeline's output bus, the input bus to the coefficient multiplier is made up of sub-buses. Form the example shown in Figure 3-3 the input bus is made up of nine sub-buses (P_1 to P_9), all 'n' bits wide. In addition, the output bus from the coefficient multiplier is one bus made up of sub-buses. Note that the number of sub-buses on the input and output buses is equal to the mask length squared. However, the width of the sub-buses is not the same.

From Figure 3-3 it can be seen that the outputs from each multiplier should be different widths. For example, the output from the centre multiplication will be three bits wider than the input, whereas the input and outputs for the corner multipliers will be the same. To eliminate this problem all the output sub-buses are made the same width. That is, each sub-bus will have a width equal to the result from the largest multiplication. Therefore, the overall width of the output bus from the coefficient multiplier is:

$$Width\ of\ coefficent\ multipliers\ output\ bus = \left(\frac{Log((\text{No. of grey scales}) \times (\text{Largest coefficient}))}{Log2} \right) \times (Mask\ length)^2$$

Equation 3-4

3.3 Generic summation

The generic summation entity adds together the results from the coefficient, pixel multiplication's. In other words the summer must add together the sub-buses of the output bus from the coefficient multiplier. Thus, if the convolution mask is 3×3 the summation must add the nine multiplication results. This is achieved by adding together the first two values and then adding the result to the next value, until all the results have been summed. However, as the speed constraints of the system are tight, a full pipelined adder structure is required so that the timing is not violated. The use of a pipelined adder will have a latency attached to it but will allow a much higher data throughput to be obtained [8]. Figure 3-4 shows a schematic representation of a fully pipelined adder that will sum nine numbers.

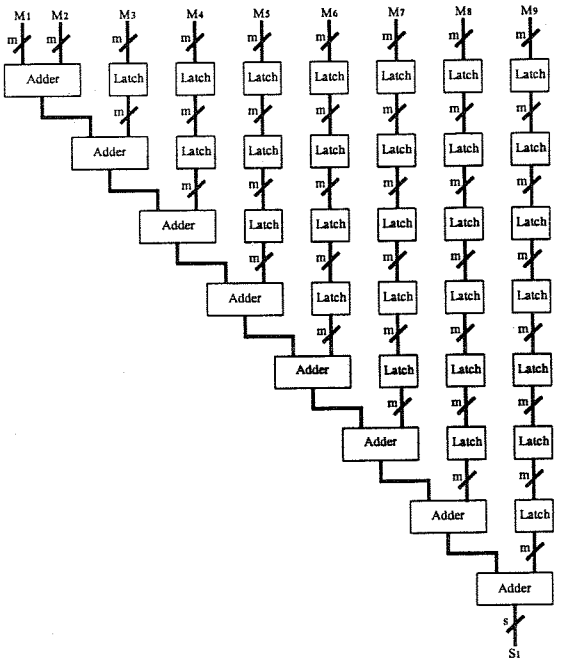

Figure 3-4 Schematic diagram of a fully pipelined generic adder for summing the nine values from a 3×3 mask

The input bus to the adder is the same width as the output bus from the coefficient multiplier entity as given by Equation 3-4. The coefficient multiplier's output sub-buses connect directly to the input sub-buses of the summation entity. The output bus of the summation entity will have a width of:

$$Width\ of\ summation\ output\ bus = \left(\frac{Log((\text{No. of grey scales}) \times (\text{Coefficient range}) \times (\text{Mask lenght})^2)}{Log2} \right)$$

Equation 3-5

Note that if Equation 3-5 is calculated and a whole number is not obtained the bus width will be equal to this value rounded up to the next whole number.

3.4 Generic scaling division

As the image convolution model has been designed for power of two filter operators, the scaling division is implemented as binary right hand shift. For example, if the result from the summation is '001101110100', and the scaling division is sixteen, this may be implemented by a four place right hand shift. Thus, the result from the division will be '00110111'. A diagram representing this scaling division is shown in Figure 3-5.

Figure 3-5 Diagram showing the implementation of divide by sixteen normalisation factor

The divider input bus is the same width as the output bus from the summation entity Equation 3-5. However, after the division has been performed the result will be the same width as the original input to the system. Therefore, the width of the output bus from the division entity is:

$$Width\ of\ division\ output\ bus = \frac{Log\ (\text{No. of grey scales})}{Log2}$$

Equation 3-6

3.5 Counter

The convolution model requires three counter circuits that have been included within one counter entity. The first counter is needed to synchronise the start of each new output image. This counter should count each pixel of the image and set a control signal at the start of each new image. Thus, this counter should have counter states:

After a reset:

$$Synchronisation\ counter\ states = \text{Pipeline delay latency}$$

Thereafter:

$$Synchronisation\ counter\ states = (\text{Image length})^2$$

Equation 3-7

In addition, two more counters are needed to keep track of the current co-ordinate position within the image being processed. This information is necessary so that the edges of the image can be zeroed. Hence, when the current position within the image is an edge a control signal is made active. These two counters both have counter states:

$$Co - ordinate\ counter\ states = \text{Image length}$$

Equation 3-8

3.6 Edge zero

The edge zero entity is connected directly to the output from the division entity. In addition, the co-ordinate control output from the counter entity is connected to this entity. The edge zero outputs a zero if the co-ordinate output signal is made active and latches out the current data input if the signal is not active.

4. Conclusions

If the convolver component models are structured carefully and any algorithmic parameters are quantified using generics, then algorithmic changes to the model may be implemented simply by modifying the generic values. This means that such changes can be realised in a matter of minutes where they would have taken days using a traditional design technique. Once the model has been altered it is possible to assess the impact of the changes on the system performance and architecture using simulation and synthesis. Thus, designers will instantly be able assess if the modification has had the desired effect, and alternatives can be quickly and easily tried. The synthesis of multiple architecture models enables the impact on the systems realisation to be verified. Without the generic component library it is difficult to implement algorithmic changes upon a system. The designer may have to rework large parts of the design just to assess the architectural impact of the alterations. However, using the generic library changes may be made instantly and the new architecture synthesised very quickly.

References

[1] Gonzalez, R. C., and P. Wintz, *Digital Image Processing,* Addison-Wesley, 1987.

[2] Shewring, I. W., M. A. Wahab 'An Integrated Approach to the Design and Implementation of Image Filters.' *Proceedings IEE 15th SARAGA Colloquium on Digital and Analogue Filters and Filtering Systems,* 1995.

[3] Hurat, S., 'Standards for Interoperability and Portability.' *Proceedings of the International User's Forum, VIUF-Fall'94: Component Modelling,* 1994.

[4] Pridmore, J., 'The standard virtual interface - An interoperability approach' *The RASSP Digest,* Vol. 2, 4th. Qtr. 1995.

[5] IEEE Std 1164-1993, IEEE Standard Multivalued Logic System for VHDL Model Interoperability (Std_logic_1164).

[6] Hein, C., 'Exploiting VHDL design in RASSP.' *Proceeding of the VIUF Conference, Component modeling.* Fall 1994.

[7] Hein, C., T. Carpenter, A. Gadient, R. Harr, P. Kalutkiewicz & V. Madissetti, *RASSP VHDL Modeling Terminology and Taxonomy.* RASSP Taxonomy Working Group (RTWG), 1996.

[8] Dadda, L., & V. Piuri, 'Pipelined Adders.' IEEE Transactions on Computers, Vol. 45, No. 3, 1996.