Signal Processing Applications using VHDL on Splash 2

Sea H. Choi, Nalini K. Ratha, Moon J. Chung, and Diane T. Rover[†]
Dept. of Computer Science
Michigan State University
East Lansing, MI 48824
[†] Dept. of Electrical Engineering

Abstract

This paper reports the use of VHDL in modeling signal processing algorithms on a reconfigurable parallel architecture. The algorithms are specified by describing the behavior of each processing element of the parallel system and then synthesized for the specific hardware. The advantages and limitations of using VHDL in such an environment are reported.

1 Introduction

Signal processing applications are computation intensive primarily because of the large amount of data to be handled in a very short time. Using single processor machines, we cannot get the desired performance. Parallel machines and application specific integrated circuits (ASICs) are used to obtain the desired speed. However, these options are costly. Furthermore, once the ASIC is built, it is very difficult to change the design.

Field Programmable Gate Arrays (FPGAs) have gained considerable attention recently because of their reconfigurability. FPGAs allow a new form of computing where the architecture of a computer may evolve over time, changing to fit the needs of each application it executes. With an FPGA-based machine, the architecture can be tailored to meet the desired performance for the given application.

The reconfigurable architecture we have adopted is Splash 2, developed and built by Supercomputing Research Center (SRC). Splash 2 is a special purpose attached parallel processor having processing elements (PEs) based on user programmable Xilinx 4010 FPGA chips. The Splash 2 system consists of a Sun SPARCstation as a host, an interface board, and Splash array boards ranging from one to sixteen boards. Each array board of Splash consists of 16 PEs having linear connections as well as a reconfigurable crossbar interconnection between PEs.

VHDL is used to model the computational algorithms as well as the architectures. Simulation is used to verify the parallel model. Then, from the VHDL description, the compiler re-targets the algorithms and architecture into FPGAs. Thus, the role of VHDL in this system is multifaceted: it is used as both the specification (modeling) language and the implementation language. It describes algorithms within user applications and the necessary hardware to realize them (including processing units, memories, and interconnections). To support our current work in image processing and to test the use of Splash 2 and VHDL on a widely used algorithm, we have implemented 1-D and 2-D convolution algorithms.

This paper is organized as follows. The Splash 2 architecture is introduced in the next section including the programming environment of the system. Section 3 covers the basic theory of convolution and the algorithm used. The VHDL implementation is covered in section 4. Experimental results are presented in section 5. Conclusions are given in section 6.

2 Splash 2

Splash 2 [1] is an attached special purpose parallel processor where each processing element is a user programmable FPGA chip. The architecture of Splash 2 can easily support parallel applications, such as systolic or data-parallel computations. Splash 2 is developed/modified from the

Splash 1 system [2] which consisted of a fixed size linear array of Xilinx 3090 FPGA chips. Splash 2 has several improvements over Splash 1. Splash 2 is based on newer hardware technology, the Xilinx XC4010 FPGA. A crossbar is added to connect PEs on a board in any pattern. The linear path was the only configuration in Splash 1. The programming environment centers on VHDL in place of SRC's Logic Description Generator (LDG) [3, 4, 5].

2.1 Splash 2 Architecture

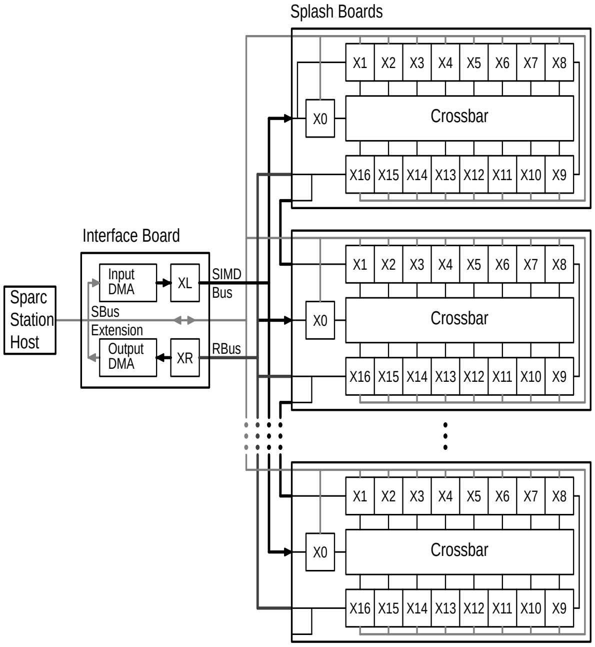

Figure 1: Splash 2 Architecture

The Splash 2 is attached to a Sun SPARCstation host. Figure 1 shows the Splash 2 architecture [6, 7]. The host is connected to the Splash 2 via an interface board. The host can read/write to memories on the Splash processing boards via this interface board.

Each Splash 2 processing board has 16 processing elements, X_1 through X_{16}, and there is one special PE, X_0, which controls the data flow into the processor board. Each X_i is a PE built around a Xilinx 4010 FPGA chip. A crossbar connection can be programmed by X_0. The processing element organization is shown in Figure 2. Each PE has 512 KB of attached memory, which has a 16-bit word size. The host can also read/write this memory over the SBus.

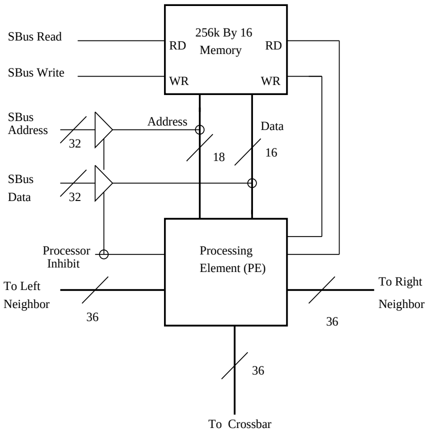

Figure 2: A Processing Element (PE) in Splash 2

Each PE can communicate using the SIMD Bus (left-right neighbor data paths) or the crossbar. Data can be broadcast using the crossbar. The Splash 2 architecture supports systolic and pipeline modes of computation, SIMD or data-parallel mode (PEs execute the same instructions on different data streams), or even MIMD mode (PEs execute different instructions on different data streams). The Splash 2 system can run at a maximum clock speed of 40 MHz where the maximum is limited by the FPGA technology. The actual operating speed is determined when the FPGA logic is synthesized.

2.2 Programming Splash 2

The programming environment for Splash 2 is based on VHDL [8, 9]. The behavioral description is analyzed, simulated, and synthesized onto Xilinx FPGAs. An application for Splash 2 is developed by writing its behavioral description in VHDL, and the description is iteratively refined and debugged with the Splash 2 system simulator. After the description is verified to be functionally correct by simulation, it is translated into a Xilinx net list form. The net list is then mapped onto the FPGA architecture by an automatic partition, placement and routing tool to form a loadable FPGA object module. A static timing analysis tool is then applied to the object module to determine the maximum operating speed. This process is shown in Figure 3.

To program Splash 2, we need to program each of the PEs, i.e. X_0 through X_{16}, and the host interface. The host interface is responsible for data transfers between host and the Splash 2 board. For this purpose, the system provides a C-language interface and C programs are written for the host's tasks.

```
1. Input: A 1-dimensional vector f(t), a mask vector g(t).
2. Output: A 1-dimensional result vector h(t).
3. Begin          Assume $k$ PEs are available.
                  The $i^{th}$ PE holds the g(i) mask value.
       On each PE:
                  Receive from left neighbor: signal value f(i), partial sum S(i-1).
                  Compute new partial sum S(i) = S(i-1) + f(i) * g(i).
                  Send signal value f(i) and partial sum S(i) to right neighbor.
   End.
```

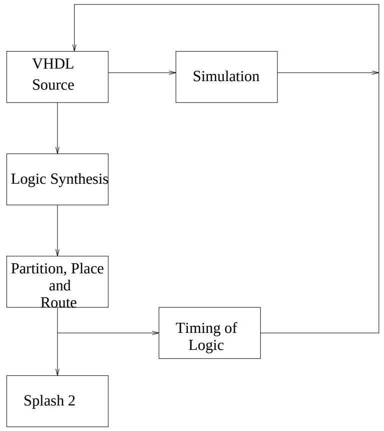

Figure 3: Splash 2 Programming Flow Diagram

3 Convolution

An important class of signal and image processing algorithms are based on the convolution of two signals. For analog one-dimensional signals, $f(t)$ and $g(t)$, their convolution $h(t)$ is defined as

$$h(t) = \int_{-\infty}^{+\infty} g(x) f(x-t) dx \qquad (1)$$

For discrete-time one-dimensional signals, this reduces to the following equation:

$$h(t) = \sum_{-\infty}^{+\infty} g(x) f(x-t) \qquad (2)$$

Further, if the signal $g(t)$ is a finite-time-duration signal (called the mask signal), then the summation range changes, so that

$$h(t) = \sum_{x=1}^{+k} g(x) f(x-t) \qquad (3)$$

where k is the mask size.

On a sequential machine, convolution is easily implemented. However, a sequential computer may not be practical when convolution needs to be done in real-time for a large number of data points. For parallelizing a convolution computation, two approaches can be taken [10]: (i) data parallel computing and (ii) systolic computing. Data parallel computing uses a divide-and-conquer approach to deal with the large amount of data. Usually $f(t)$ has large number of data points (spread over a large time domain) compared to the mask signal $g(t)$. Hence, a set of processors can be used to compute on shorter segments of the data in parallel. This computational model assumes that each PE is powerful enough to carry out all computations and that signal values are already available at each PE. If this latter assumption is not the case, then the data path from host to PEs becomes a bottleneck in distributing data to the PEs. This problem can be overcome using a systolic approach. This requires only a single data path between the host and k PEs and that the PEs are powerful enough to perform the add and multiply operations.

A set of PEs are used as a linear array. The basic convolution algorithm translates into the above systolic algorithm.

The input is fed into the PE array at the left input of the first PE with the partial sum initialized to zero. At the output of the last PE, the final result is obtained.

It can be readily seen that the algorithm, after the initial pipeline latency, will output one result every clock cycle. The overall model has been schematically described in Figure 4. If the number of PEs available is smaller than the number of mask values, we need to map virtual PEs to the physical PEs using a fair policy. In that case, the PEs need wider communication paths and will require more cycles to produce results.

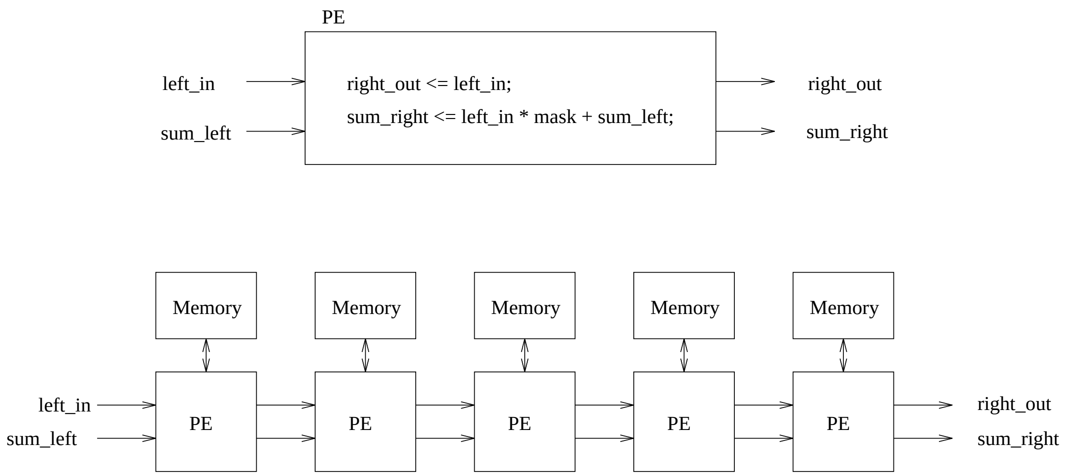

Figure 4: PE Behavior and Configuration for 1-D Convolution ($k = 5$)

4 VHDL Implementation

In this section, the implementation of 1-D convolution using VHDL is described. In our implementation of convolution, the PEs are configured as a linear array and a systolic method is used. In 1-D convolution, each signal value (ranging from 0 through 255 in the applications being targeted) is input into the left most PE at each clock cycle, and this input is multiplied by the mask value. For each signal value, its input value and the partial sum are passed to the next PE to the right for accumulation.

4.1 1-D Convolution

A signal value, `left_in`, is received at each clock cycle and is multiplied by the mask value, `mask`. The partial sum, `psum`, is also received at the same time and is added to this newly formed product. The signal value and the new partial sum, `sum`, are then passed to the right PE for accumulation. The VHDL program representing this PE behavior is shown below. This configuration is shown in Figure 4.

```
WAIT ON valid_clk;                      -- 1
psum := sum_left;                       -- 2
IF (left_in>0) AND (psum>=0) THEN       -- 3
 sum := mask * left_in + psum;          -- 4
ELSE
 sum := psum2;
END IF;
right_out <= left_in;   sum_right <= sum;
```

In the program, the calculation of the accumulation is shown at line 4. The synchronization with the clock is done by waiting for the clock transition (line 1).

4.2 Splash 2 Implementation

The 1-D convolution program is ported to the Splash simulator in order to simulate and synthesize for the Splash architecture and for Xilinx FPGAs. The entity description [4] of each PE is shown in Figure 5. The signals are self-explanatory and are shown schematically in Figure 2.

The behavioral code for 1-D convolution using VHDL could not be directly synthesized into a Xilinx FPGA chip since an FPGA consists of a finite, and limited, number of logic devices. The logic synthesized directly from the VHDL description includes 8-bit multiplication, which consumes too many gates to fit in the Xilinx chip. To reduce the logic requirements, the multiplication operation is converted into a table look-up operation using the memory of the PEs. The VHDL code segment for the Splash PE is shown in Figure 6. A diagram of the PE input/output data paths is shown in Figure 7.

Each PE is synchronized by the rising edge of the clock, `XP_Clk` (line 1). The effect of lines 4 and 5 is shown in the comment of line 3. However, a memory look-up is used in place of multiplication. Since each signal is 8 bits wide, there are 256 possible distinct values for each data point. The 8-bit input is shown at line 4. It is used as an address into the 256-word look-up table, which holds the results of the 256 values times the mask value (a single constant for that PE). We can pre-calculate the 256 multiplications and load these values into the PE memory. The loading of these values is done from

```
entity Xilinx_Processing_Element is
  Generic(
    BD_ID               : Integer := 0;         --  Splash Board ID
    PE_ID               : Integer := 0);        --  Processing Element ID
  Port (
    XP_Left             : inout DataPath;       --  Left Data Bus
    XP_Right            : inout DataPath;       --  Right Data Bus
    XP_Xbar             : inout DataPath;       --  Crossbar Data Bus
    XP_Xbar_EN_L : out Bit_Vector(4 downto 0);  --  Crossbar Enable (low-true)
    XP_Clk              : in    Bit;            --  Splash System Clock
    XP_Int              : out   Bit;            --  Interrupt Signal
    XP_Mem_A            : inout MemAddr;        --  Splash Memory Address Bus
    XP_Mem_D            : inout MemData;        --  Splash Memory Data Bus
    XP_Mem_RD_L         : inout RBit3;          --  Memory Read Signal
    XP_Mem_WR_L         : inout RBit3;          --  Memory Write Signal
    XP_Mem_Disable      : in    Bit;            --  Splash Memory Disable
    XP_Broadcast        : in    Bit;            --  Broadcast Signal
    XP_Reset            : in    Bit;            --  Reset Signal
    XP_HS0              : inout RBit3;          --  Handshake Signal Zero
    XP_HS1              : in    Bit;            --  Handshake Signal One
    XP_GOR_Result       : inout RBit3;          --  Global OR Result Signal
    XP_GOR_Valid        : inout RBit3;          --  Global OR Valid Signal
    XP_LED              : out   Bit);           --  LED Signal
end Xilinx_Processing_Element;
```

Figure 5: PE Entity Description

the host of the Splash system before the convolution execution starts. The signal value is used as the look-up table address.

Once the Address is set to a certain value (line 4), the value in the memory location pointed to by Address is loaded into a variable Data at the next clock cycle. Thus the multiplication result is available at the next clock cycle in Data (line 5). One buffer is used fo rstoring a temporary result (psum1 at line 2) since Data is available one clock cycle later. Before the partial sum is calculated, we need to convert the signal of the bit-vector type into integer type. This is done by using a function bvtoi. Similarly itobv is used to convert an integer to a bit-vector. The input signal value and partial sum are packed and passed to the right neighbor (lines 6 and 7).

Once the PE program is complete, we configure the Splash system. This is done by programming the VHDL top model. A portion of this code is shown in Figure 8. We use a predefined interface board configuration. By assigning the generic constants such as input.dat, we can tailor the system (line 1). Line 2 specifies one Splash board. A specific interconnection between PEs is obtained by loading the initial configuration from a file, here named xcrossbar (line 3). Each PE is configured by using a predesigned component. Line 4 marks the section of code for PE 1, line 5 for PE 2, and the rest of PEs are defined similarly.

5 Experimental Results

Splash simulation indicates the correct behavior of the model, as determined by analyzing Figure 9, the simulation waveform. The synthesized result achieves a clock rate of 18.5 MHz for the 1-D convolution as shown in Figure 10.

The timing result shows that different operations can run at a various maximum clock rates. For example in Figure 10, several operations can run at the maximum clock rate of 18.5 MHz, while other operations can run at the maximum rate of 40 MHz. The 18.5 MHz clock rate becomes the maximum operating speed. Figure 11 shows a different timing profile obtained by using a different PE implementation. Specifically, multiplication was performed as a series of additions. For nonnegative mask values, the signal value is repeatedly added a num-

```
WAIT until XP_Clk’event AND XP_Clk = ’1’;               -- 1
-- ...
psum := psum1;
psum1 := bvtoi(left_in(23 downto 8));                   -- 2
--   sum := mask*bvtoi(left_in(7 downto 0))+psum;       -- 3
Address(7 downto 0) <= left_in(7 downto 0);             -- 4
sum := bvtoi(Data)+psum;                                -- 5
right_out(7 downto 0) <= left_in(7 downto 0);           -- 6
right_out(23 downto 8) <= itobv(sum,16);                -- 7
```

Figure 6: PE Behavioral Description

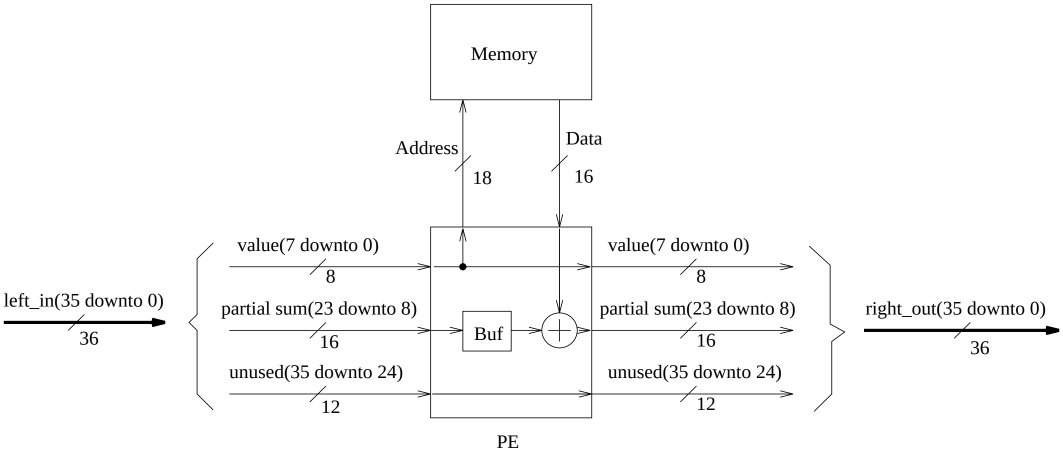

Figure 7: Splash PE Input/Output Data Paths

ber of times equal to the mask value. Using this method, the performance decreases to a maximum clock rate of 9.5 MHz, as shown in Figure 11.

6 Conclusion

VHDL is shown to be powerful enough to model parallel processing algorithms and their timing behavior. VHDL is especially suitable for modeling the behavior of a processing element, which is then synthesized to specific hardware. Using VHDL, we can also simulate parallel algorithms and measure the performance of parallel architectures. We better understand the behavior of parallel algorithms after implementing them in VHDL, simulating them, and analyzing their performance, especially the timing behavior. Different implementation techniques can lead to different synthesis results and hence different performance although the simulation results remain the same.

We are in the process of developing of a 2-D convolution algorithm on Splash 2 to be used on digital images.

Acknowledgment

This research was supported by a grant from the Supercomputing Research Center.

References

[1] J. M. Arnold, D. A. Buell, and E. G. Davis, “Splash 2,” in *Proc. 4th Annual ACM Symp. on Parallel Algorithms and Architectures*, pp. 316–322, 1992.

[2] M. B. Gokhale, B. Holmes, A. Kosper, D. Kunze, D. Lopresti, S. P. Lucas, R. G. Minnich, and P. Olsen, “SPLASH: A Reconfigurable Linear Logic Array,” in *Proceedings of the International Conference on Parallel Processing*, August 1990.

```
configuration TOP of Splash_System is
-- ...
  use entity Interface.Interface_Board(Structure)   -- 1
    Generic Map (Input_file1  => "input.dat", Output_file1 => "result.dat",
                 File_Type => Hex,  Clock_Freq => 20);
-- ...
  use entity S2Board.Splash2_Boards(Structure)      -- 2
    Generic Map (Number_Of_Boards => 1);
-- ...
  use entity S2Board.Splash_Crossbar(Behavior)      -- 3
    Generic Map (Config_File => "xcrossbar");
-- ...
for XPARTS (1)   -- PE 1                            -- 4
  for all : Xilinx_Processing_part
     use entity work.Xilinx_Processing_Part(conv_1d);
  end for;
  for all : Memory_Part
     use entity S2Board.Memory_Part(Dynamic)
       Generic Map (Load_File => "lookup01.dat");
  end for;
end for;
for XPARTS (2)   -- PE 2                             -- 5
  for all : Xilinx_Processing_part
     use entity work.Xilinx_Processing_Part(conv_1d);
  end for;
  for all : Memory_Part
     use entity S2Board.Memory_Part(Dynamic)
       Generic Map (Load_File  => "lookup02.dat");
  end for;
end for;
-- ...
```

Figure 8: Top Module Configuration Description

[3] J. M. Arnold, "The Splash 2 Software Environment," in *Proc. of IEEE Workshop on FPGAs as Custom Computing Machines*, April 1993.

[4] J. M. Arnold and D. A. Buell, "VHDL Programming on Splash 2," in *Proc. of the International Workshop on Field-Programmable Logic*, 1993.

[5] M. B. Gokhale and R. G. Minnich, "FPGA Computing in a Data Parallel C, SRC-TR-93-097," tech. rep., Supercomputing Research Center, Institute for Defense Analyses, April 1993.

[6] J. M. Arnold and M. A. McGarry, "SPLASH 2 Programmer's Manual, SRC-TR-93-107," tech. rep., Supercomputing Research Center, Institute for Defense Analyses, September 1993.

[7] D. A. Buell, "A SPLASH 2 Tutorial, SRC-TR-92-087," tech. rep., Supercomputing Research Center, Institute for Defense Analyses, December 1992.

[8] *IEEE Standard VHDL Language Reference Manual.* IEEE Std 1076-1987, New York, NY, 1988.

[9] *VHDL Compiler Reference Manual.* Synopsys Inc., Mt. View, CA, 1992.

[10] P. E. Danielsson, *Serial Parallel Convolvers*, ch. 2, pp. 31–71. Systolic Signal Processing System, E. E. Swartzlander, (Ed.), Marcel Dekker, Inc., New York, 1987.

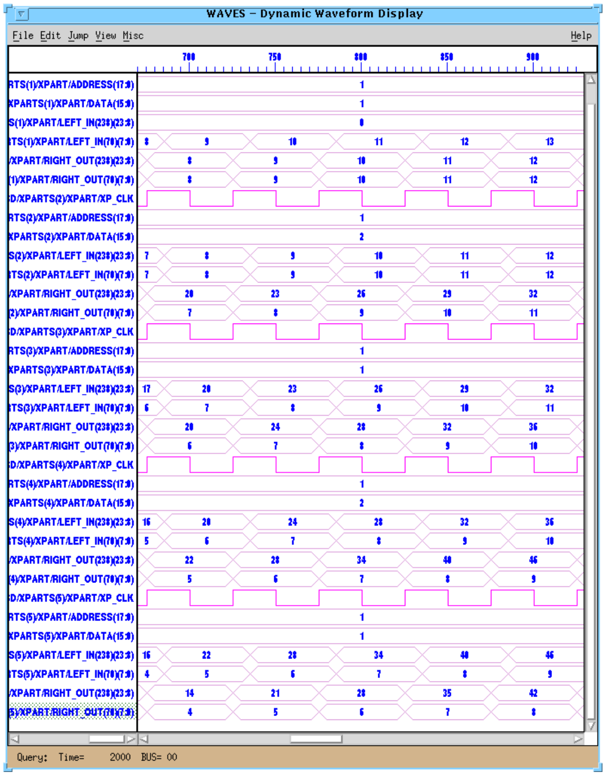
WAVES – Dynamic Waveform Display
File Edit Jump View Misc
Help
700
750
800
850
900
RTS(1)/XPART/ADDRESS(17:0) 1
XPARTS(1)/XPART/DATA(15:0) 1
S(1)/XPART/LEFT_IN(238)(23:0) 0
RTS(1)/XPART/LEFT_IN(70)(7:0) 8 9 10 11 12 13
/XPART/RIGHT_OUT(238)(23:0) 8 9 10 11 12
(1)/XPART/RIGHT_OUT(70)(7:0) 8 9 10 11 12
D/XPARTS(2)/XPART/XP_CLK
RTS(2)/XPART/ADDRESS(17:0) 1
XPARTS(2)/XPART/DATA(15:0) 2
S(2)/XPART/LEFT_IN(238)(23:0) 7 8 9 10 11 12
RTS(2)/XPART/LEFT_IN(70)(7:0) 7 8 9 10 11 12
/XPART/RIGHT_OUT(238)(23:0) 20 23 26 29 32
(2)/XPART/RIGHT_OUT(70)(7:0) 7 8 9 10 11
D/XPARTS(3)/XPART/XP_CLK
RTS(3)/XPART/ADDRESS(17:0) 1
XPARTS(3)/XPART/DATA(15:0) 1
S(3)/XPART/LEFT_IN(238)(23:0) 17 20 23 26 29 32
RTS(3)/XPART/LEFT_IN(70)(7:0) 6 7 8 9 10 11
/XPART/RIGHT_OUT(238)(23:0) 20 24 28 32 36
(3)/XPART/RIGHT_OUT(70)(7:0) 6 7 8 9 10
D/XPARTS(4)/XPART/XP_CLK
RTS(4)/XPART/ADDRESS(17:0) 1
XPARTS(4)/XPART/DATA(15:0) 2
S(4)/XPART/LEFT_IN(238)(23:0) 16 20 24 28 32 36
RTS(4)/XPART/LEFT_IN(70)(7:0) 5 6 7 8 9 10
/XPART/RIGHT_OUT(238)(23:0) 22 28 34 40 46
(4)/XPART/RIGHT_OUT(70)(7:0) 5 6 7 8 9
D/XPARTS(5)/XPART/XP_CLK
RTS(5)/XPART/ADDRESS(17:0) 1
XPARTS(5)/XPART/DATA(15:0) 1
S(5)/XPART/LEFT_IN(238)(23:0) 16 22 28 34 40 46
RTS(5)/XPART/LEFT_IN(70)(7:0) 4 5 6 7 8 9
/XPART/RIGHT_OUT(238)(23:0) 14 21 28 35 42
(5)/XPART/RIGHT_OUT(70)(7:0) 4 5 6 7 8
Query: Time= 2000 BUS= 00

Figure 9: Simulation Result for 1-D Convolution

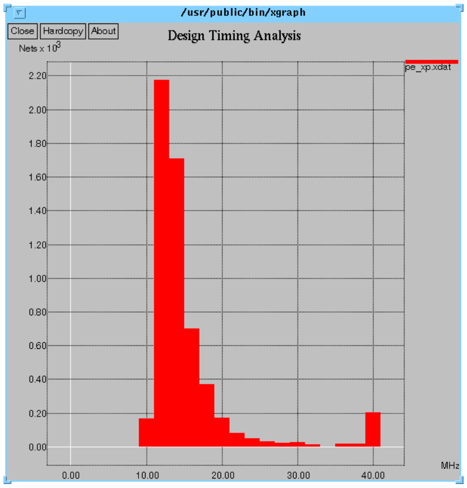

Figure 11: Timing Result using Loop Statement

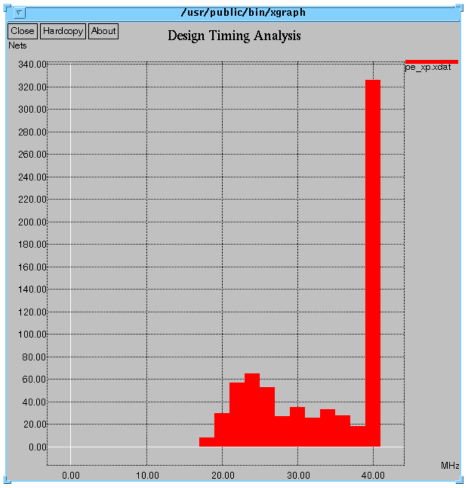

Figure 10: Timing Result using Memory Lookup