Parity Checker Implementations in SpecC

Qiang Xie and Daniel Gajski

CECS Technical Report 02-06
January 27, 2002

Center for Embedded Computer Systems
Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425, USA
(949) 824-8059

{qxie,gajski}@ics.uci.edu

Parity Checker Implementations in SpecC

Qiang Xie and Daniel Gajski

CECS Technical Report 02-06
January 27, 2002

Center for Embedded Computer Systems
Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425,USA
(949) 824-8059

{qxie,gajski}@ics.uci.edu

Abstract

In this report we discuss an example where we synthesize a multiple implementations of a design with our RTL synthesis tool. We use different resource allocation combinations to obtain multiple implementations and perform synthesis on them. The initial part of this report introduces a Parity Checker, including its FSMD and implementation model. Then we further develop into different implementations of One's Counter. We do different combinations of allocation of different resources to the design and perform the synthesis on these implementations with our tool. We then analyze the performance of these implementations on the basis of synthesis results and show how the user has the choice to make the ultimate decision about the design with due considerations to all involved tradeoffs.

Contents

List of Figures

Parity Checker Implementations in SpecC

Qiang Xie and Daniel Gajski
Center for Embedded Computer Systems
Information and Computer Science
University of California, Irvine

Abstract

In this report we discuss an example where we synthesize a multiple implementations of a design with our RTL synthesis tool. We use different resource allocation combinations to obtain multiple implementations and perform synthesis on them. The initial part of this report introduces a Parity Checker, including its FSMD and implementation model. Then we further develop into different implementations of One's Counter. We do different combinations of allocation of different resources to the design and perform the synthesis on these implementations with our tool. We then analyze the performance of these implementations on the basis of synthesis results and show how the user has the choice to make the ultimate decision about the design with due considerations to all involved tradeoffs.

1. Introduction

The Parity Checker is used to check whether the number of '1' in a given data is odd or even. If it is odd, it will output '1', otherwise it should be '0'.

The Parity Checker incorporates 2 behaviors, One's Counter and Even Checker. One's Counter calculates the number of '1's in a data while Even Checker checks whether that number is odd or even. This setup is perfect to illustrate the working of a composition of FSMDs which are communicating with each other.

Figure 1 shows the block diagram of the Parity Checker. `Inport` is the input port for input data and `Outport` is the output port for output result. The port `Start` is a control port which is used to signal Parity Checker to start calculation and the port `Done` is a status port which is used to indicate the completion of the calculation. Inside the Parity Checker, `idata` is the data bus used to send data from Even Checker to One's Counter, and `iocount` is a 5 bit data bus used to send computed data from One's Counter to Even Checker. `Istart` is an internal control signal which is used to initiate the operation of One's Counter and `ack_istart` is an internal control signal from One's Counter to Even Checker. Signal `idone` is used to notify the Even Checker once the computations are over in One's Counter and `ack_idone` is an 0 signal from the Even Checker to the One's Counter. The Even Checker and One's Counter use a handshaking protocol to communicate with each other. As shown in Figure 1, we use two pairs of signals, namely, `istart` and `ack_istart`, `idone` and `ack_idone`, to implement the handshaking protocol described in the FSMD models in Section 2.

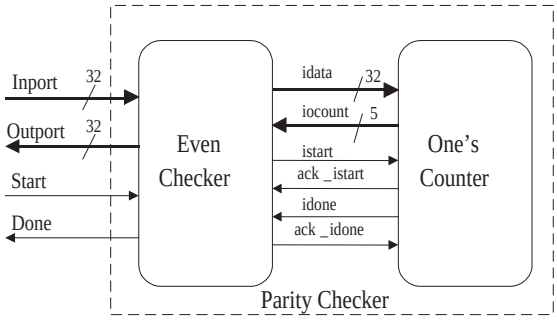


Figure 1. Block Diagram of Parity Checker

The rest of this report is organized as follows: Section 2 describes the FSMD models for both Even Checker and One's Counter. Section 3 gives an insight into the RTL implementations for these two components. Section 4 analyses the experimental results after performing the synthesis on different implementations of One's Counter using our RTL synthesis tool. Section 5 concludes this report with a brief summary and future works.

2. FSMD Models

A finite-state machine with datapath (FSMD) is one of the most popular design models in high-level synthesis and design of hardware systems. In this section, we will discuss the FSMD models for the Even Checker and One's Counter to illustrate their concomitant working as a Parity Checker.

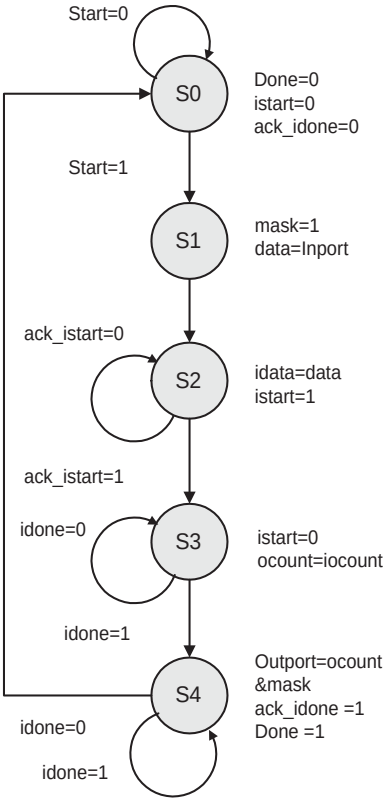


Figure 2. FSMD model for Even Checker

Figure 2 shows the FSMD of the Even Checker. The FSMD begins at state `S0` and evolves as follows:

- In the initial state `S0`, `Done`, `istart` and `ack_idone` are set to 0, and the state machine waits for port `Start` to become 1 before making transition to the next state, `S1`.
- In state `S1`, the variable `mask` is set to 1. Further data is read from the input port `Inport` and written to variable `data`. State machine make a transition to the next state, `S2`.
- In state `S2`, The `data` is sent to port `idata` and the signal `istart` is set to 1 which is received by One's Counter unit. One's Counter in turn sets the signal `ack_istart` to 1 in order to acknowledge the request to start computation. The state machine waits for port `ack_istart` to become 1 before making a transition to the next state, `S3`.
- In state `S3`, data is read from the port `iocount` and written to variable `ocount`, and the signal `istart` is reset to 0. The state machine then waits until `idone` is set to 1 before making transition to next state, `S4`.
- Finally, in state `S4`, the signals `ack_idone` and `Done` are set to 1. The Even Checker computes the logical 'AND' of `ocount` and `mask`, and the result is written to port `Outport`. The state machine then waits until it receives the signal `idone` reset to 0. After which it makes a transition to the initial state `S0`.

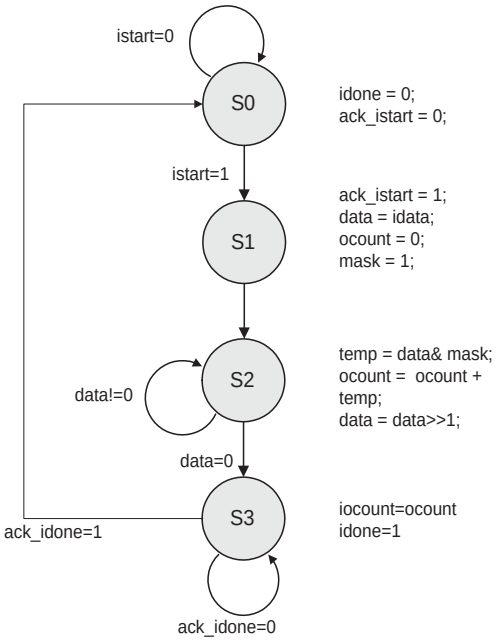
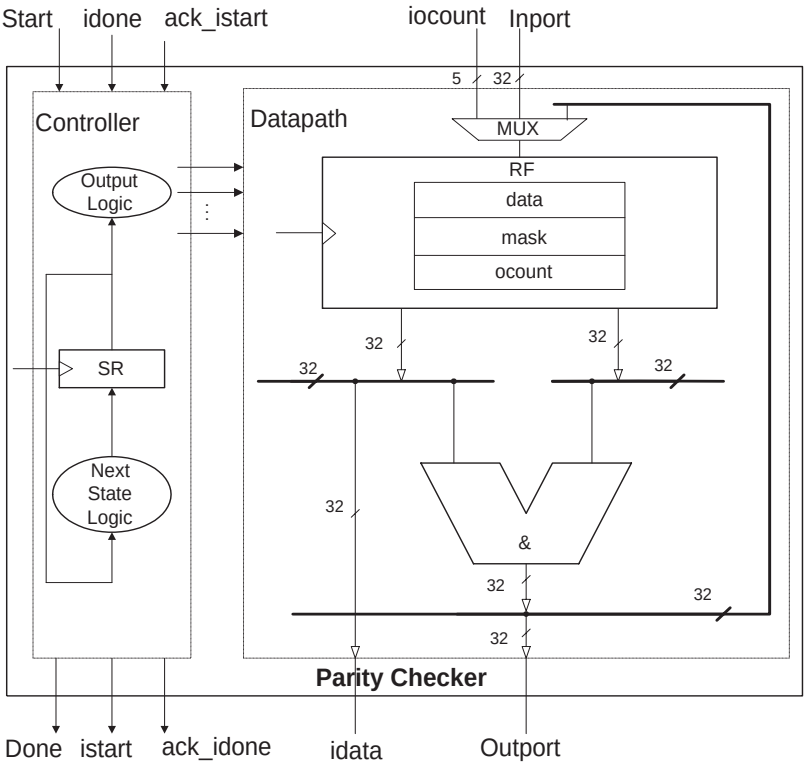


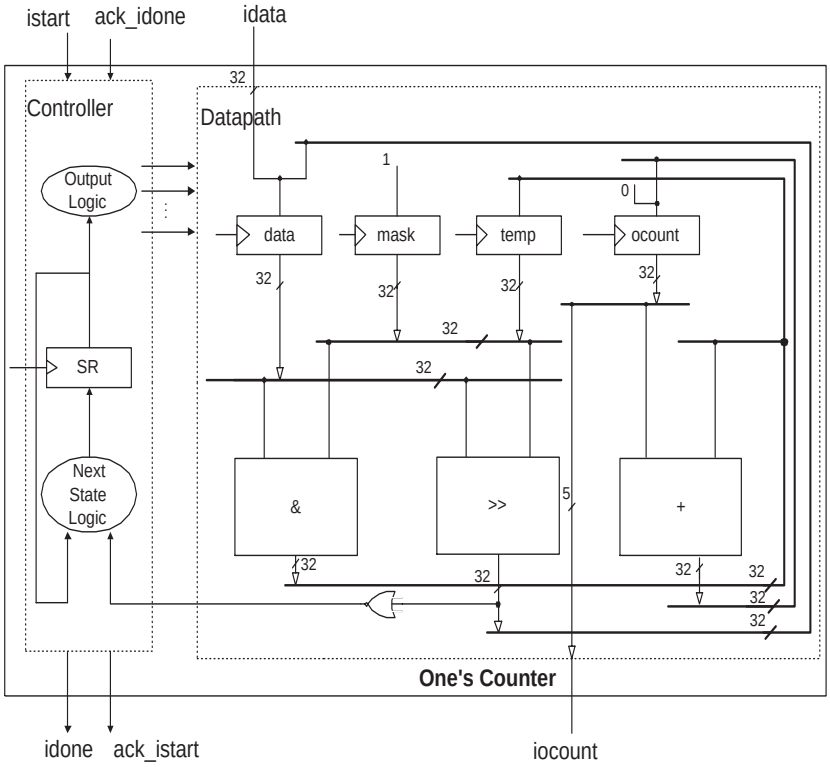
Figure 3. FSMD model for One's Counter

Figure 3 shows the FSMD model for the One's Counter. One's Counter starts at state `S0` and evolves as follows:

- In state `S0`, `idone`, `ack_istart` are set to 0, and the state machine waits until `istart` becomes 1 before making a transition to next state, `S1`.
- In state `S1`, `ack_istart` and `mask` are set to 1, `ocount` is set to 0, and the input data is written to `data` from port `idata`. Then the sate machine makes a transition to state `S2`.
- In `S2`, One's Counter complete the following arithmetic operations.
  - Computing the logical operation 'AND' of `data` and `mask` and writing the result to variable `temp`;
  - Computing the sum of `ocount` and `temp` and writing the result to `ocount`;
  - Shifting `data` right by a bit;



(a) RTL structure model for Even Checker



(b) Architecture model for One's Counter

Figure 4. RTL structure model for Parity Checker

– Comparing the resulting value of `data`, if `data` equals to 0, the state machine will make a transition to state `S3`.

- Finally, in state `S3`, the data `ocount` is written to port `iocount`, and `idone` is set to 1. Then the state machines waits until it receives the signal `ack_idone` set to 1 before making a transition to the initial state`S0`.

The Even Checker and One's Counter work in concurrent fashion and they follow a handshaking protocol for their inter-communication. With the aid of this protocol, it is possible to interleave their execution in multiple clock speeds.

3. Implementation

Now we examine the implementations of the Even Checker and One's Counter. In an RTL structure view, each implementation has a controller and a datapath. The datapath consists of sequential storage units and combinatorial units which are used for computation of the behavior. The controller dictates the operations of the datapath by assigning proper values to the control signals. The controller is a Finite State Machine(FSM), which is used to control the flow on the datapath and output the control signals. It can be divided into three parts, the `SR`(state register) which stores the state information, the `Next State Logic` which generates the next state information, and the `Output Logic` which controls the operation of the datapath.

Figure 4(a) explicates the implementation of the Even Checker. The inputs to the controller are the port `Start`, and the internal signal `idone`, `ack_istart` from One's Counter. The outputs of the controller are the output port `Done`, and `istart` and `ack_idone` which are sent to One's Counter. The datapath of Even Checker sends data to One's Counter and receives the result of computation from it, which it uses to check whether the result is even or odd. The datapath consists of a register file which is used to store variables `data`, `mask` and `ocount`, and an ALU unit for the logical 'AND' operation. The inputs to the datapath include the data port `Inport` and `iocount` from the One's Counter. The outputs of the datapath include port `Outport`, and `idata` which is sent to One's Counter.

Figure 4(b) explicates the implementation of One's Counter. The inputs for the controller are `istart` and `ack_idone` from Even Checker. The outputs of the controller are signals `idone` and `ack_istart` which are sent to Even Checker. The controller checks the status of the `data` in the datapath through a wire connection. The datapath performs all the arithmetic operations, such as addition, AND or comparison. It includes of 4 registers, 3 functional units, and 6 buses. One's Counter uses the registers to store the variables `data`, `mask`, `temp` and `ocount`. The functional units perform perform the AND, addition, comparison and shift operations separately. There is also a NOR gate which is used to check whether the `data` is 0. The input to this datapath is the `idata` from the Even Checker and the output is the `iocount` which in turn is sent to the Even Checker.

The above implementations are optimal in the sense they are designed with maximum performance. This although maximum the performance, the cost is very high due to so many resources used here. If we allocate less resources the cost may be lowered but the performance decreases. In the following section, we discuss different implementations of the Parity Checker with different resource allocations at their disposal. We use our RTL synthesis tool to synthesize various such implementations and make a comparative study of their performances which finally leads us to a good design decision.

4. Experimental Results

Our tool synthesize a design from a RTL behavior description in style 1 to style 4 [ZSY+00]. This tool performs four different tasks: scheduling, storage unit binding, functional unit binding and bus binding. The scheduling takes place first followed by the different binding. Here we use resource constraint binding algorithms in which the type and the number of of resources to be used can be specified. The tool synthesizes different implementations with varying resource allocation combinations. The central idea is that a user specifies the resources, such as register files, functional units and buses, the tool synthesizes the design into an implementation that makes complete utilization of these allocated resources and at the same time minimize the cost of the interconnections, such as multiplexer and bus driver. So with our tool user does a comparative performance analysis of different implementations according to the synthesis result and finally determines the most apt implementation with consideration to the cost-performance tradeoff.

Considering the two components in Parity Checker, the Even Checker is relatively simple and involves little exploration compared to that in the One's Counter. Therefore we concentrate on various possible implementations of the One's Counter.

The input to the tool is a behavior description of the One's Counter in RTL style 1(Appendix A.2.2). In the input source code, we explicitly define the FSMD states in a declaration and use a case statement in a `while()` loop to move from state to state. The behaviors in each state are triggered on a `clk` event, which is implemented as `wait(clk)` in the beginning of the loop. There are 4 variables (`data`, `temp`, `mask`, `ocount`) and 4 operations ($>>$, +, &, $>=$). We make allocations of different types or

Resource Unit	Operations	Delays(ns)
ALU	add, sub, negate, and, or, not	3.02
ALU (pipelined)	add, sub, negate, and, or, not	3.02 1.5
Shifter	shl, shr	2.25
Register File	storage unit	1.46
Register File(setup)	storage unit	1.20
Reg32	storage unit	0.75
Reg32(setup)	storage unit	0.59
Control Unit	control logic	1.4

Table 1. RTL components delays

number of register files, ALUs, and buses and obtain different implementations.

In order to synthesize this design, we create RTL models for different resources as shown in Appendix A.1. These RTL models include:

- Storage units:register, register file;
- Function units: ALU, Shifter;
- Interconnection: bus

The allocation of these resources is made from the component library. The delays of these RTL components are shown in Table 4.1.

We now discuss the performance of different implementations in detail.

4.1 Design 1: 1 ALU, 1 Shifter, 1 RF, 3 buses

This implementation has a register file(`RF`), 1 ALU, 1 Shifter and 3 buses. It utilizes the minimum number of resource and has the least cost among all the implementations.

Appendix B shows the output result in RTL style 4 after synthesizing this design. We can see that since there is limited resource available, the tool splits the original state `S1` in the input to 3 states, state `S1` and two extra states `X0` and `X1`. We use initial 'X' to represent the extra states generated by our tool. In state `S1`, the tool also splits it to 4 states, `S2`, `X2`, `X3` and `X4`. So the total number of states in the output FSMD increases from 4 to 9 with 5 extra states generated.

Figure 5 shows the RTL structure model of the One's Counter. The operations '+', '&' and comparison are mapped to the `ALU` and shift operation is mapped to `SHIFTER`. And the 4 variables `data`, `mask`, `temp` and `ocount` are mapped to the register file `RF`. Since we do not consider multiplexer and bus driver in our tool, they are not generated in our result. Following is a estimation of the performance of this implementation.

The clock cycle of this implementation can be determined as the maximum of the critical path candidates as follows:

- Delay of path p1, computing the next state of the FSM

$$\begin{aligned}\Delta(p1) &= delay(SR) + delay(OL) + delay(RF) \\ &+ delay(ALU) + delay(NL) + setup(SR) \\ &= 0.75 + 1.4 + 1.46 + 3.02 + 1.4 + 0.59 \\ &= 8.62ns\end{aligned}$$

- Delay of datapath, performing the arithmetic operations

$$\begin{aligned}\Delta(p2) &= delay(SR) + delay(OL) + delay(RF) \\ &delay(ALU) + setup(RF) \\ &= 0.75 + 1.4 + 1.46 + 3.02 + 1.20 \\ &= 7.83ns\end{aligned}$$

Here `delay(SR)` is the delay lapsed in reading state register `SR` which is the same as the `Reg32` in Table 4.1, `delay(OL)` is the delay of output logic which equals the delay of `Control Unit`, `delay(ALU)` is the delay of the ALU, `delay(RF)` is the delay of reading data from the register file `RF`, `delay(NL)` is the delay of the next state logic which equals to the delay of `Control Unit`, `setup(RF)` is the setup time of the register file `RF`, `setup(SR)` is the setup time of the `SR` register. Hence, the minimum clock cycle is:

$$Clock\_cycle = max(\Delta(p1), \Delta(p2)) = 8.62ns$$

As we mention above, the number of the states in the output is 9. And there are 4 states in the loop from state `S2` back to `S2`. Therefore, given an input data with 32 bits of '1', the One's Counter needs $4 \times 31 + 9 = 133$ cycles to finish the computation. The estimated execution time for this implementation is:

$$\begin{aligned}Execution\_time &= num\_cycles \times clock\_cycle \\ &= 133 \times 8.62 \\ &\approx 1.15\mu s\end{aligned}$$

4.2 Design 2: 1 ALU, 1 Shifter, 4 registers, 3 buses

This implementation uses a ALU, a Shifter, three buses and four registers to implement the One's Counter. These 4 registers are used as special storage units to store the variable `data`, `temp`, `mask` and `ocount` respectively.

Appendix C shows the output result in style 4 RTL after synthesizing this design. Comparing to the input, the original state `S2` is split up to 4 states, `S2`, `X0`, `X1` and `X2` due to only a `ALU` available in each control step. The total number of states in the output FSMD increases from 4 to 7 with 3 extra states generated.

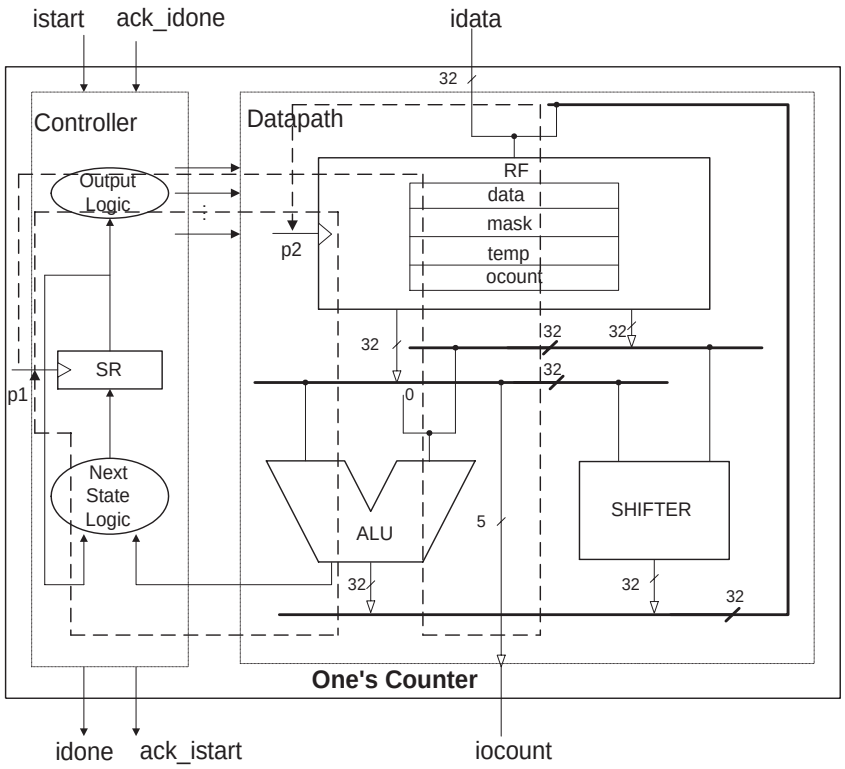


Figure 5. Design 1: 1 ALU, 1 Shifter, 1 RF, 3 buses

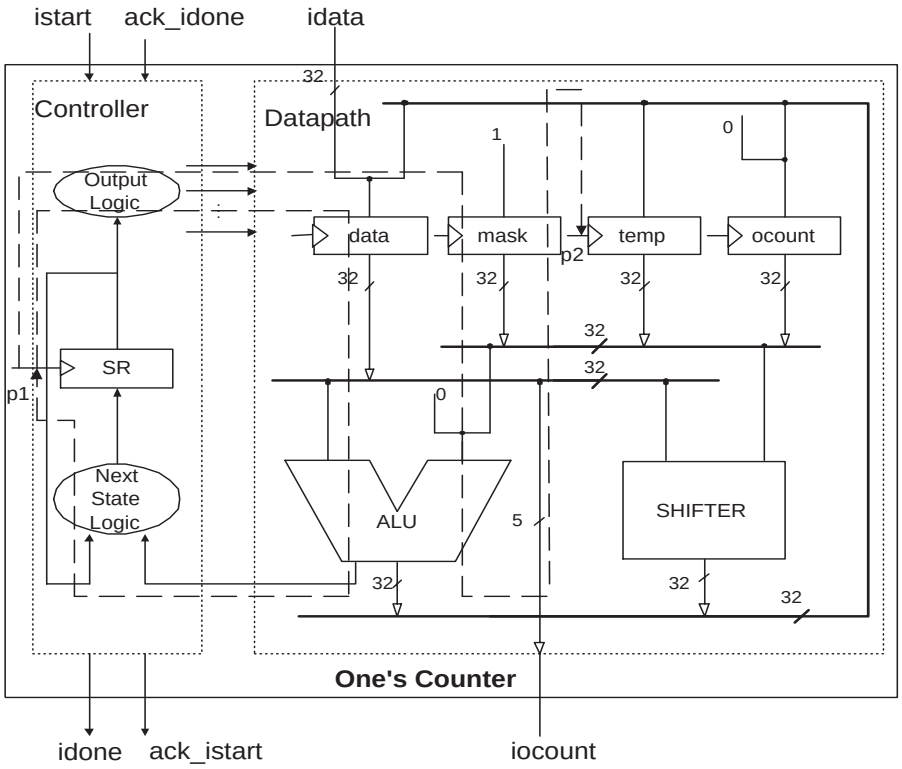


Figure 6. Design 2: 1 ALU, 1 Shifter, 4 registers, 3 buses

Here we observe a bug in the storage unit binding from the output result. Though the four variables are supposed to be mapped to the 4 allocated registers after binding, this mapping is not reflected in the output code. In the output result(Appendix C), the variables are still used in the FSMD description instead of using special registers to replace them.

Figure 6 depicts the RTL structure model for this implementation. Since the numbers of functional units and buses are the same as in Design 4.1, the critical path is almost the same. Again, there are two candidates for the critical path:

- Delay of path p1, computing the next state of the FSM

$$\begin{aligned}\Delta(p1) &= delay(SR) + delay(OL) + delay(Reg32) \\ &+ delay(ALU) + delay(NL) + setup(SR) \\ &= 0.75 + 1.4 + 0.75 + 3.02 + 1.4 + 0.59 \\ &= 7.91ns\end{aligned}$$

- Delay of datapath, performing the arithmetic operations

$$\begin{aligned}\Delta(p2) &= delay(SR) + delay(OL) + delay(Reg32) \\ &+ delay(ALU) + setup(Reg32) \\ &= 0.75 + 1.4 + 0.75 + 3.02 + 0.59 \\ &= 6.51ns\end{aligned}$$

Hence, the minimum clock cycle is:

$$Clock\_cycle = max(\Delta(p1), \Delta(p2)) = 7.91ns$$

The number of the states in the output FSMD is 7. Also there are 4 states in the loop from state `S2` back to `S2`. Given an input data with 32 bits of '1', the One's Counter needs $4 \times 31 + 7 = 131$ cycles to finish the computation. The estimated execution time for this implementation is:

$$\begin{aligned}Execution\_time &= num\_cycles \times clock\_cycle \\ &= 131 \times 7.91 \\ &\approx 1.04\mu s\end{aligned}$$

which is only about 9.5% faster than Design 4.1. We see that there is not much improvement compared to Design 4.1. The only difference from the previous implementation is that special registers are used in place of the register file. To further improve the performance, we need to allocate more functional units and buses to the design so that different operations in datapath can run in parallel in the same control step.

4.3 Design 3: 2 ALUs, 1 Shifter, 4 registers, 5 buses

We allocate two ALUs, a Shifter, four registers and five buses in this implementation. So that the ALU and the Shifter can run in parallel thereby reducing the number of execution cycles required for the computation to take place.

Appendix D shows the output result in style 4 RTL after performing synthesis on it. There is only one extra state `X0` split up from the original state `S2`. The total number of states in the output FSMD is 5. And the variables are still not mapped as depicted in Design 4.2.

Figure 7 shows the RTL structure model for this implementation. There are two critical path candidates:

- Delay of path p1, computing the next state of the FSM

$$\begin{aligned}\Delta(p1) &= delay(SR) + delay(OL) + delay(Reg32) \\ &+ delay(ALU) + delay(NL) + setup(SR) \\ &= 0.75 + 1.4 + 0.75 + 3.02 + 1.4 + 0.59 \\ &= 7.91ns\end{aligned}$$

- Delay of datapath, performing the arithmetic operations

$$\begin{aligned}\Delta(p2) &= delay(SR) + delay(OL) + delay(Reg32) \\ &+ delay(ALU) + setup(Reg32) \\ &= 0.75 + 1.4 + 0.75 + 3.02 + 0.59 \\ &= 6.51ns\end{aligned}$$

Hence, the minimum clock cycle is:

$$Clock\_cycle = max(\Delta(p1), \Delta(p2)) = 7.91ns$$

The number of states in the output FSMD is 5 and there are 2 states in the loop from state `S2` back to `S2`. The total cycles need to complete the computation for input data with 32 bits of '1' is $31 * 2 + 5 = 67$, which is a drastic improvement compared to that for Design 4.1 and Design 4.2. The estimated execution time here is:

$$\begin{aligned}Execution\_time &= num\_cycles \times clock\_cycle \\ &= 67 \times 7.91 \\ &\approx 0.53\mu s\end{aligned}$$

which shows a significantly decrease compared to that in Design 4.1 and Design 4.2. We observe that the greater the number of resources at our disposal, the more efficient performance we obtain.

However, this implementation also reflects some limitations. Though we allocate 5 buses to the design, only four are put to use. Also the binding of storage units is not reflected in the output.

4.4 Design 4: 1 pipelined ALU, 1 pipelined Shifter, 1 RF, 3 buses

In this design we attempt a pipelined implementation with a limited number of resources for further improvement in the performance. We allocate a pipelined ALU and a pipelined Shifter, other resources being the same as in Design 4.1.

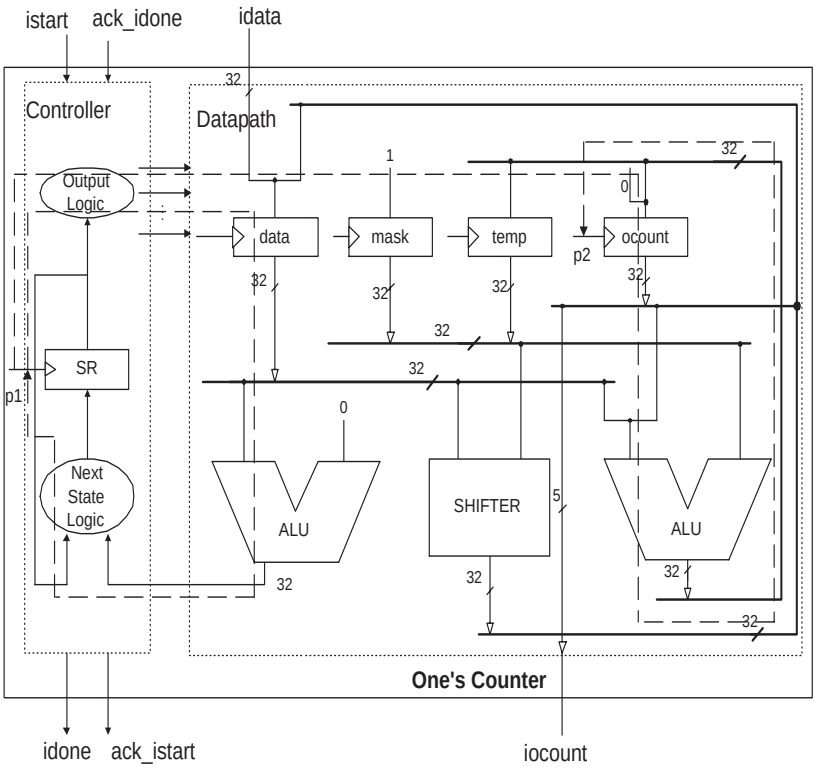


Figure 7. Design 3: 2 ALUs, 1 Shifter, 4 registers, 5 buses

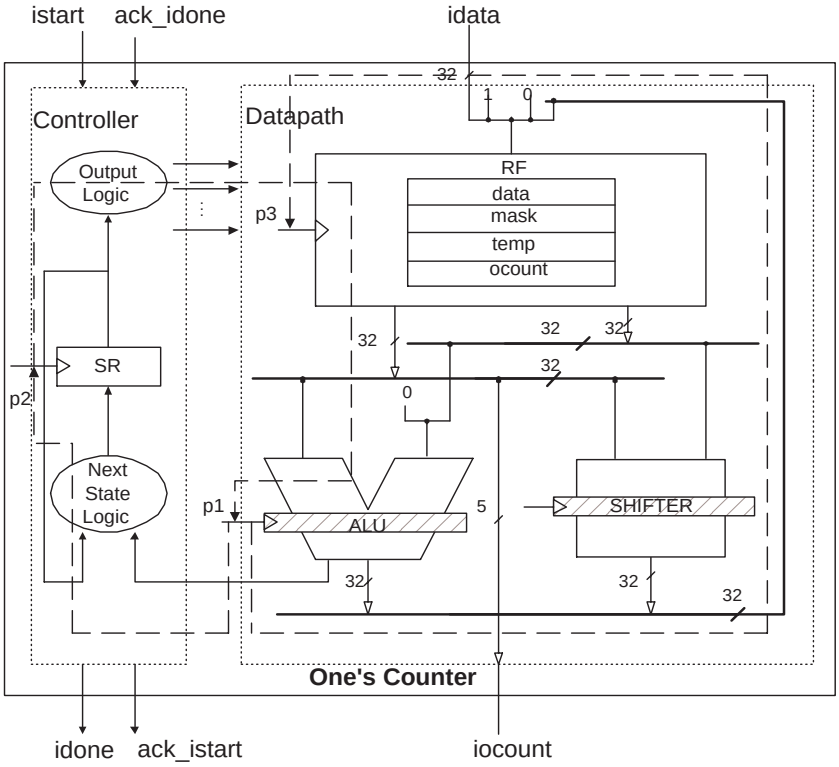


Figure 8. Design 4: 1 pipelined ALU, 1 pipelined Shifter, 1 RF, 3 buses

Appendix E shows output result after synthesis has been performed on this design. To enable pipelining, the tool splits the original states into more states comparing to that in design 4.1. First, the tool splits the original state `S1` to 3 states, state `S1` and two extra states `X0` and `X1`. Then for original state `S2`, the tool splits it to 5 states, `S2`, `X2`, `X3`, `X4` and `S5`. The total number of states in the output FSMD increases to 10 with 6 extra states generated.

Figure 8 depicts the RTL structure model for this implementation. Since a different combination of units is being used in the design, the critical path candidates change significantly compared to the earlier implementations. There are three candidates for the critical path here:

- Delay of path p1, from the state register(`SR`) in the controller to the pipelined ALU in the datapath:

$$\begin{aligned}\Delta(p1) &= delay(SR) + delay(OL) + delay(RF) \\ &\quad + pipe(ALU) \\ &= 0.75 + 1.4 + 1.46 + 1.5 \\ &= 5.11ns\end{aligned}$$

- Delay of path p2, from `ALU` to `SR` to finish the computation for the next state of FSM:

$$\begin{aligned}\Delta(p2) &= pipe(ALU) + delay(NL) + setup(SR) \\ &= 1.5 + 1.4 + 0.59 \\ &= 3.49ns\end{aligned}$$

- Delay of path p3, from `ALU` to `RF` finish the computation of the arithmetic operations:

$$\begin{aligned}\Delta(p3) &= pipe(ALU) + + setup(RF) \\ &= 1.5 + 1.2 \\ &= 2.7ns\end{aligned}$$

`Pipe(ALU)` is the delay of the pipelined `ALU` and is a half of a normal ALU delay as in Table 4.1. Since p1 has the largest delay among all the three candidates, the minimum clock cycle is:

$$Clock\_cycle = max(\Delta(p1), \Delta(p2), \Delta(p3)) = 5.11ns$$

It is much faster than the earlier implementations. However, there is a payoff involved as we split the FSMD into further more states in order to incorporate pipelining in the design. The number of states in the output FSMD of One's Counter blows up to 10 and the total execution cycles for an input data with 32 bits of '1' 1 is $31 * 5 + 10 = 165$. Hence, the estimated execution time is:

$$\begin{aligned}Execution\_time &= num\_cycles \times clock\_cycle \\ &= 165 \times 5.11 \\ &\approx 0.84\mu s\end{aligned}$$

1 of the growth in the number of states, the current design is still 27while the cost is more or less same for both of them. Similarly the idea of pipelined implementations can be further extended to other units such as storage units to obtain substantial improvement in the design performance.

4.5 Design 5: 2 pipelined ALU, 1 pipelined Shifter, 4 registers, 5 buses

In this design we employ larger number of resources with pipelined functional units. We allocate two pipelined ALUs and one pipelined Shifter, four special registers, and five buses. This combination subsumes both the Design 4.3 and Design 4.4.

Appendix F shows the output result in style 4 RTL after perform synthesis on this design. There are only 3 extra states (`X0`, `X1` and `X2`) generated from the original state `S2`. The total number of states in the output FSMD is 7. And the variables are not mapped to registers as depicted in Design 4.2.

Figure 9 shows the RTL structure model for this implementation. Again, there are three candidates for the critical path in this design:

- Delay of path p1, from the state register(`SR`) in the controller to the pipelined `ALU` in the datapath:

$$\begin{aligned}\Delta(p1) &= delay(SR) + delay(OL) + delay(Reg32) \\ &\quad + pipe(ALU) \\ &= 0.75 + 1.4 + 0.75 + 1.5 \\ &= 4.4ns\end{aligned}$$

- Delay of path p2, from pipelined `ALU` to `SR` to finish the computation of the next state of FSM:

$$\begin{aligned}\Delta(p2) &= pipe(ALU) + delay(NL) + setup(SR) \\ &= 1.5 + 1.4 + 0.75 \\ &= 3.65ns\end{aligned}$$

- Delay of path p3, from pipelined `ALU` to register to finish the computation of the arithmetic operations:

$$\begin{aligned}\Delta(p3) &= pipe(ALU) + setup(Reg32) \\ &= 1.5 + 0.59 \\ &= 2.09ns\end{aligned}$$

Hence, the minimum clock cycle is:

$$Clock\_cycle = max(\Delta(p1), \Delta(p2), \Delta(p3)) = 4.4ns$$

The number of states in the FSMD of One's Counter is 7(4 states in the loop from state `S2` to `S2`) and the total execution cycles for an input data with 32 bits of '1' is $31 * 4 + 7 = 131$. Hence, the estimated execution time is:

$$\begin{aligned}Execution\_time &= num\_cycles \times clock\_cycle \\ &= 131 \times 4.4 \\ &\approx 0.58\mu s\end{aligned}$$

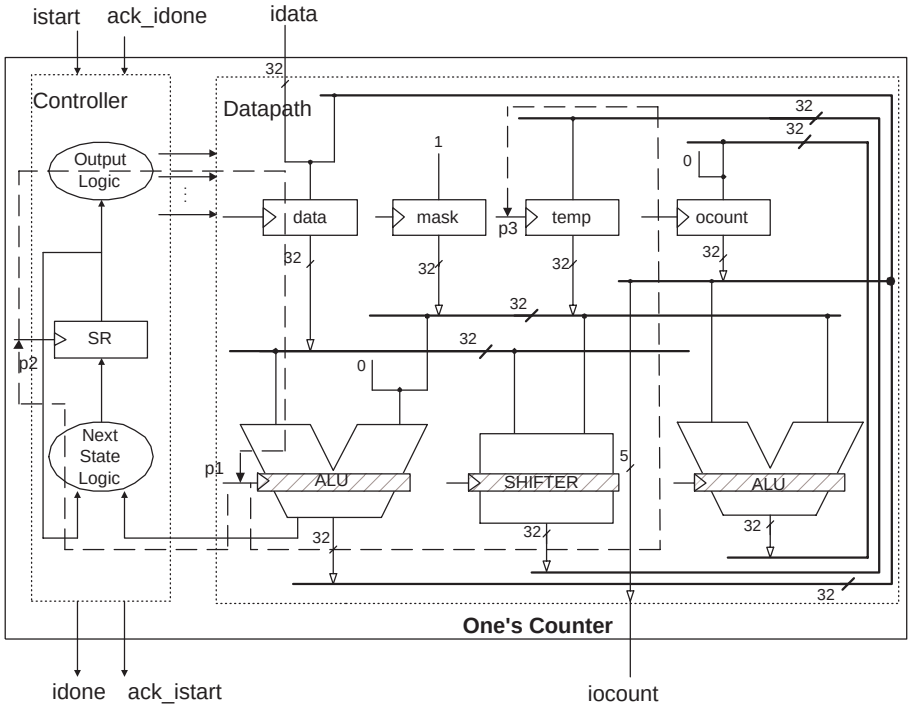


Figure 9. Design 5: 2 pipelined ALU, 1 pipelined Shifter, 4 registers, 5 buses

Design	Clock Cycles(ns)	Number of States in FSMD	Execution Cycles	Execution Time(μs)
Design 1	14.32	9	133	1.15
Design 2	13.61	7	131	1.04
Design 3	12.81	5	67	0.53
Design 4	8.36	10	165	0.84
Design 5	7.65	7	131	0.58

Table 2. Experimental Result for One's Counter(input data is 32 bit of '1')

The execution time is faster then that of Design 4.4 but not as fast as that of Design 4.3. It is due to the fact that the FSMD has to be split up into more states in order to allow pipelined implementation. Therefore, the total number of execution cycles of Design 4.5 is much more than that of Design 4.3 thereby making its execution sluggish.

Table 4.5 shows a summary of the above different implementations. Design 4.1 has the lowest cost but with the penalty of slowest execution. Design 4.3 has the fastest performance but with a high cost. These results clearly demonstrate that our tool synthesizes different implementations of a design, with different resources allocations. Hence gives the user the 1 to determine the final implementation on the basis of performance-cost tradeoff after estimating the performance of different implementations.

Also, we notice that there are two ways to improve the design performance, increasing the number of resources used in the design or introducing pipelined units in the Design. Employing more resources in the design can reduce the number of states in the FSMD of the behavior with little change in the critical path. Introduction of pipelined units in the design causes a drastic reduction in clock cycle but at the same time the FSMD of the behavior has to split up to generate more states and the total number of execution cycles increases, which leads to a poorer performance.

5. Conclusion and Future Works

We demonstrated the different implementations of the Parity Checker, mainly the One's Counter with different allocation of resources from the component library. Our RTL synthesis tool was used to do comparative analysis of the performance of these different implementations. These allow the end user to decide upon the final implementation which strikes out an optimal balance between the cost and the performance.

However, there are still some impending modifications in the tool. Firstly, the result of bus binding is not optimal, the resources are not fully utilized in bus binding as can be observed in Design 4.3, where out of five allocated buses only four are mapped. Secondly, when we allocate registers for all the variables in the FSMD, the binding result doesn't reflect the binding of the registers. Finally, our approach introduces storage units like register file and memory in the component library mapping of whose ports is not supported in the current binding algorithm. The future extension to our work is proposing a binding algorithm which considers the mapping of ports of the storage units. We expect to release the improvised version in the future.

References

[ZSY+00] P. Zhang, D. Shin, H. Yu, Q. Xie, and D. Gajski. SpecC RTL Design Methodology. Technical Report ICS-TR-00-44, University of California, Irvine, December 2000.

A. Parity Checker in RTL style 1

A.1 RTL component Library

A.1.1 ALU

```
bit[31:0] alu(bit[31:0] a, bit[31:0] b, int ctrl)
{
    note  alu.library = "1";
    note  alu.a="data";
    note  alu.b="data";
    note  alu.sum="data";
    note  alu.ctrl="control";

    note  alu.type="rca";
    note  alu.width="472";
    note  alu.height="920";
    note  alu.cost="100";
    note  alu.pipelined = "1";
    note  alu.delay="2";
    note  alu.bits="32";
    note  alu.operation="+,-,<,<=,>,>=,!=,==,&,+:,-:,+=,-=";
    bit[31:0] sum;

    switch(ctrl)   {
        case 000b:  // +
            sum = a+b;
            break;
        case 001b:  // -
            sum = a-b;
            break;
        case 010b:  // <
            sum = (a<b)? 0x0001:0x0000;
            break;
        case 011b:  // <=
            sum = (a<=b)? 0x0001:0x0000;
            break;
        case 100b:  // >
            sum = (a>b)? 0x0001:0x0000;
            break;
        case 101b:  // >=
            sum = (a>=b)? 0x0001:0x0000;
            break;
        case 110b:  // !=
            sum = (a!=b)? 0x0001:0x0000;
            break;
        case 111b:  // ==
            sum = (a==b)? 0x0001:0x0000;
            break;
        case 1000b:     // &
            sum = a&b;
            break;
    }
    return sum;
```

```
}
```

A.1.2 Shifter

```
bit[31:0] shift(bit[31:0] si, bit[31:0] amount, int ctrl)
{
    note shift.library = "1";
    note shift.si = "data";
    note shift.amount = "data";
    note shift.so = "data";
    note shift.ctrl = "control";

    note shift.type = "shifta";
    note shift.width = "272";
    note shift.height = "420";
    note shift.cost = "60";
    note shift.pipelined = "1";
    note shift.delay = "2";
    note shift.bits = "32";
    note shift.operation = ">>,<<";

    bit[31:0] so;
    switch(ctrl) {
      case 0b:
        so = si >> amount;
        break;
      case 1b:
        so = si << amount;
        break;
    }
    return so;
}
```

A.1.3 Register File

```
void RF(event clk, bit[0:0] rst, bit[31:0] inp,
    bit[1:0] raA, bit[1:0] raB, bit[0:0] reA, bit[0:0] reB,
    bit[1:0] wa, bit[0:0] we, bit[31:0] outA, bit[31:0] outB)
{
    note RF.library = "1";
    note RF.type = "RF";
    note RF.size = "4";
    note RF.width = "272";
    note RF.height = "420";
    note RF.cost = "60";
    note RF.pipelined = "0";
    note RF.delay = "0";
    note RF.num_inports= "1";
    note RF.num_outports = "2";
    note RF.bits = "32";
}
```

A.1.4 Register

```
void Reg32(event clk, bit[0:0] rst, bit[31:0] input,
    bit[0:0] read,
    bit[0:0] write, bit[31:0] output)
```

```
{
    note Reg32.library = "1";
    note Reg32.type = "reg";
    note Reg32.size = "1";
    note Reg32.width = "72";
    note Reg32.height = "100";
    note Reg32.cost = "10";
    note Reg32.pipelined = "0";
    note Reg32.delay = "0";
    note Reg32.num_rports= "1";
    note Reg32.num_wports = "1";
    note Reg32.bits = "32";
}
```

A.1.5 Bus

```
void bus(bit[31:0] outp, bit[31:0] inp)
{
    note bus.library = "1";
    note bus.type = "bus";
    note bus.width = "1";
    note bus.height = "1";
    note bus.cost = "60";
    note bus.delay = "0";
    note bus.bits = "32";
}
```

A.2 Parity Checker

```
/*****************************************************************************
*  Title: parity.sc
*  Author: Dongwan Shin
*  Date: 12/03/2000
*  Description: top behavior for parity checker
*****************************************************************************/

import "ones";
import "even";

behavior parity(in event clk1, in event clk2, in unsigned bit[0:0] rst,
    in unsigned bit[31:0] Inport, out unsigned bit[31:0] Outport,
    in unsigned bit[0:0] Start, out unsigned bit[0:0] Done)
{
    unsigned bit[31:0] data;
    unsigned bit[4:0] ocount;
    unsigned bit[0:0] istart, idone;
    unsigned bit[0:0] ack_istart, ack_idone;

    even U00(clk1, rst, Inport, Outport, Start, Done, data, ocount, istart,
        idone, ack_istart, ack_idone);
    ones U01(clk2, rst, data, ocount, istart, idone, ack_istart, ack_idone);

    void main (void)
    {
        par {
            U00.main();
```

```
            U01.main();
        }
    }
};
```

A.2.1 Even Checker

```
/*****************************************************************************
*  Title: even.sc
*  Author: Qiang Xie
*  Date: 10/01/2001
*  Description: Behavioral RTL model for even parity checker
*****************************************************************************/
import "io";
behavior even(in event clk, in unsigned bit[0:0] rst,
    in unsigned bit[31:0] Inport, out unsigned bit[31:0] Outport,
    in unsigned bit[0:0] Start, out unsigned bit[0:0] Done,
    out unsigned bit[31:0] idata, in unsigned bit[4:0] iocount,
    out unsigned bit[0:0] istart, in unsigned bit[0:0] idone,
    in unsigned bit[0:0] ack_istart, out unsigned bit[0:0] ack_idone)
{
    void main(void) {
        unsigned bit[31:0] ocount;
        unsigned bit[31:0] data, mask;
        enum state { S0, S1, S2, S3, S4 } state;

        state = S0;

        while (1) {
            wait(clk);
            if (rst == 1b) {
                state = S0;
            }
            switch (state) {
                case S0:
                    Done = 0b;
                    istart = 0b;
                    ack_idone = 0b;
                    if (Start == 1b)
                        state = S1;
                    else
                        state = S0;
                    break;
                case S1:
                    mask = 0x0001;
                    data = Inport;
                        state = S2;
                    break;
                case S2:
                    idata = data;
                    istart = 1b;
                    if (ack_istart == 1b)
                        state = S3;
                    else
                        state = S2;
```

```
                    break;
                case S3:
                    istart = 0;
                    ocount = iocount;
                    if (idone == 1)
                        state = S4;
                    else
                        state = S3;
                    break;
                case S4:
                    Outport = ocount & mask;    // even parity checker
                    ack_idone = 1;
                    Done = 1;
                    if (idone == 0)
                        state = S0;
                    else
                        state = S4;
                    break;
            }
        }
    }
};
```

A.2.2 One's Counter

```
/*****************************************************************************
*  Title: ones.sc
*  Author: Qiang Xie
*  Date: 02/11/2002
*  Description: Behavioral RTL model for Ones'Counter
*****************************************************************************/
import "lib";
behavior ones(in event clk, in unsigned bit [0:0] rst, in unsigned bit[31:0] idata,
    out unsigned bit[4:0] iocount, in unsigned bit[0:0] istart,
    out unsigned bit[0:0] idone,
    out unsigned bit[0:0] ack_istart, in unsigned bit[0:0] ack_idone)
{
    note ones.scheduled = "0";
    note ones.fubind = "0";
    note ones.regbind = "0";
    note ones.busbind = "0";

    note ones.clk = "clk";
    note ones.rst = "rst";
    note ones.idata = "data";
    note ones.iocount = "data";
    note ones.istart = "ctrl";
    note ones.idone = "ctrl";
    note ones.ack_istart = "ctrl";
    note ones.ack_idone = "ctrl";

    void main(void) {
        unsigned bit[31:0] data;
        unsigned bit[31:0] ocount;
        unsigned bit[31:0] mask;
```

```
        unsigned bit[31:0] temp;

        enum state { S0, S1, S2, S3 } state;

        state = S0;

        while (1) {
            wait(clk);
            if (rst) {
                iocount = 0x0000;
                state = S0;
            }
            switch (state) {
                case S0 :
                    idone = 0;
                    ack_istart = 0;
                    if (istart != 0)
                        state = S1;
                    else
                        state = S0;
                    break;
                case S1:
                        ack_istart = 1;
                    data = idata;
                    ocount = 0;
                    mask = 1;
                    state = S2;
                    break;
                case S2:
                    temp = data & mask;
                    ocount = ocount + temp;
                    data = data >> mask;

                    if (data == 0)
                        state = S3;
                    else
                        state = S2;
                    break;
                case S3:
                    iocount = ocount;
                    idone = 1;
                    if (ack_idone == 1b)
                        state = S0;
                    else
                        state = S3;
                    break;
            }
        }
    }
};
```

A.3 Test Bench

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

```
*  Title: tb.sc
*  Author: Dongwan Shin
*  Date: 12/03/2000
*  Description: testbench for partiy checker
*************************************************************************/

import "io";
import "clkgen";
import "ones";
import "parity";

behavior Main
{
    unsigned bit[31:0] inport, outport;
    unsigned bit[0:0] rst;
    event clk1, clk2;
    unsigned bit[0:0] start, done;

    clkgen U00(clk1, 5);
    clkgen U01(clk2, 4);
    IO U02(clk1, rst, inport, outport, start, done);
    parity U03(clk1, clk2, rst, inport, outport, start, done);

    int main (void)
    {
        par {
            U00.main();
            U01.main();
            U02.main();
            U03.main();
        }
        return 0;
    }
};
```

A.3.1 Input/Output

```
/*************************************************************************
*  Title: io.sc
*  Author: Dongwan Shin
*  Date: 11/15/2000
*  Description: input/output for testbench
*************************************************************************/

// I/O for testbench
#include <stdio.h>
#include <stdlib.h>

behavior IO(in event clk, out unsigned bit[0:0] rst,
    out unsigned bit[31:0] Inport, out unsigned bit[31:0] Outport,
    out unsigned bit[0:0] Start, in unsigned bit[0:0] Done)
{
    void main(void) {
        char buf[16];
```

```
        rst = 1b;
        Start = 0b;
        wait(clk);
        wait(clk);

        rst = 0b;       // deassign reset

        while (1) {
            printf("Input␣for␣one's␣counter:␣");
            gets(buf);
            Inport = atoi(buf);

            Start = 1b;
            wait(clk);
            while (Done != 1b) {
                wait(clk);
            }
            printf("parity␣checker␣output(%s)␣=␣%u\n", buf,
                (unsigned int)Outport);
            Start = 0b;
            waitfor(100);
        }
    }
};
```

A.3.2 Clock Generator

```
/*****************************************************************************
*  Title: clkgen.sc
*  Author: Dongwan Shin
*  Date: 11/15/2000
*  Description: clock generator
*****************************************************************************/

behavior clkgen(out event clk, in int clk_period)
{
    void main(void) {
        while (1) {
            waitfor(clk_period);
            notify(clk);
        }
    }
};
```

B. Output: One's Counter in Style 4 - Design 1

```
/*****************************************************
* SpecC code generated by 'genc'
* Date: Thu Mar 28 12:53:15 2002
* User: qxie
*****************************************************/
import "lib";
behavior ones(in event clk, in bit[0:0] rst, in bit[31:0] Inport, out bit[31:0] Outport, in bi
{

    note ones.scheduled = "1";
    note ones.fubind = "1";
    note ones.regbind = "1";
    note ones.busbind = "1";
    note ones.Inport = "data";
    note ones.Outport = "data";
    note ones.clk = "clk";
    note ones.done = "ctrl";
    note ones.rst = "rst";
    note ones.start = "ctrl";

    bit[31:0] shift0(bit[31:0] si, bit[31:0] amount, int ctrl)
    {
        return shift(si, amount, ctrl);
    }

    bit[31:0] alu0(bit[31:0] a, bit[31:0] b, int ctrl)
    {
        return alu(a, b, ctrl);
    }

    void main(void)
    {
        bit[31:0] Data;
        bit[31:0] Mask;
        bit[31:0] Ocount;
        bit[31:0] RF0[4];
        bit[31:0] Temp;
        bit[0:0] _ctrl_;
        bit[31:0] bus0;
        bit[31:0] bus1;
        bit[31:0] bus2;
        enum state { S0, S1, S2, S3, X0, X1, X2, X3, X4 } state;
        while (1)
        {
            wait(clk);
            if (rst)
            {
                state = S0;
            }
            switch (state)
            {
```

```
case S0 :
{
    done = 0;
    if (start!=0)
    {
        state = S1;
    }
    else
    {
        state = S0;
    }
    break;
}
case S1 :
{
    RF0[0] = Inport;
    state = X0;
    break;
}
case X0 :
{
    RF0[1] = 0;
    state = X1;
    break;
}
case X1 :
{
    RF0[2] = 1;
    state = S2;
    break;
}
case S2 :
{
    bus0 = RF0[0];
    bus1 = RF0[2];
    bus2 = alu0(bus0, bus1, 8);
    RF0[3] = bus2;
    state = X2;
    break;
}
case X2 :
{
    bus0 = RF0[1];
    bus1 = RF0[3];
    bus2 = alu0(bus0, bus1, 0);
    RF0[1] = bus2;
    state = X3;
    break;
}
case X3 :
{
    bus0 = RF0[0];
    bus1 = RF0[2];
```

```
                    bus2 = shift0(bus0, bus1, 0);
                    RF0[0] = bus2;
                    state = X4;
                    break;
                }
                case X4 :
                {
                    bus0 = RF0[0];
                    if (alu0(bus0, 0, 7))
                    {
                        state = S3;
                    }
                    else
                    {
                        state = S2;
                    }
                    break;
                }
                case S3 :
                {
                    done = 1;
                    bus1 = RF0[1];
                    Outport = bus1;
                    state = S0;
                    break;
                }
            }
        }
    }

};
```

C. Output: One's Counter in Style 4 - Design 2

```
/*******************************************************
* SpecC code generated by 'genc'
* Date: Tue Feb 12 23:23:01 2002
* User: qxie
*******************************************************/
import "lib";



behavior ones(in event clk, in unsigned bit[0:0] rst, in unsigned bit[31:0] idata,
        out unsigned bit[31:0] iocount, in unsigned bit[0:0] start,
        out unsigned bit[0:0] done, out unsigned bit[0:0] ack_istart,
        in unsigned bit[0:0] ack_idone)
{

    note ones.scheduled = "1";
    note ones.fubind = "1";
    note ones.regbind = "1";
    note ones.busbind = "1";
    note ones.ack_idone = "ctrl";
    note ones.ack_istart = "ctrl";
    note ones.clk = "clk";
    note ones.done = "ctrl";
    note ones.idata = "data";
    note ones.iocount = "data";
    note ones.rst = "rst";
    note ones.start = "ctrl";


    bit[31:0] shift0(bit[31:0] si, bit[31:0] amount, int ctrl)
    {
        return shift(si, amount, ctrl);
    }

    bit[31:0] alu0(bit[31:0] a, bit[31:0] b, int ctrl)
    {
        return alu(a, b, ctrl);
    }

    void main(void)
    {
        bit[31:0] Reg320;
        bit[31:0] Reg321;
        bit[31:0] Reg322;
        bit[31:0] Reg323;
        bit[0:0] _ctrl_;
        unsigned bit[31:0] data;
        unsigned bit[31:0] mask;
        unsigned bit[31:0] ocount;
        unsigned bit[31:0] temp;
        bit[31:0] bus0;
```

```
bit[31:0] bus1;
bit[31:0] bus2;
enum state { S0, S1, S2, S3, X0, X1, X2 } state;
while (1)
{
    wait(clk);
    if (rst)
    {
        state = S0;
    }
    switch (state)
    {
        case S0 :
        {
            ack_istart = 0;
            done = 0;
            if (start!=0)
            {
                state = S1;
            }
            else
            {
                state = S0;
            }
            break;
        }
        case S1 :
        {
            mask = 1;
            ocount = 0;
            data = idata;
            ack_istart = 1;
            state = S2;
            break;
        }
        case S2 :
        {
            bus0 = data;
            bus1 = mask;
            bus2 = alu0(bus0, bus1, 8);
            temp = bus2;
            state = X0;
            break;
        }
        case X0 :
        {
            bus0 = ocount;
            bus1 = temp;
            bus2 = alu0(bus0, bus1, 0);
            ocount = bus2;
            state = X1;
            break;
        }
```

```
                    case X1 :
                    {
                        bus0 = data;
                        bus1 = mask;
                        bus2 = shift0(bus0, bus1, 0);
                        data = bus2;
                        state = X2;
                        break;
                    }
                    case X2 :
                    {
                        bus0 = data;
                        if (alu0(bus0, 0, 7))
                        {
                            state = S3;
                        }
                        else
                        {
                            state = S2;
                        }
                        break;
                    }
                    case S3 :
                    {
                        done = 1;
                        bus0 = ocount;
                        iocount = bus0;
                        if (ack_idone==1)
                        {
                            state = S0;
                        }
                        else
                        {
                            state = S3;
                        }
                        break;
                    }
                }
            }
        }
};
```

D. Output: One's Counter in Style 4 - Design 3

```
/*****************************************************
* SpecC code generated by 'genc'
* Date: Tue Feb 12 23:29:28 2002
* User: qxie
*****************************************************/
import "lib";



behavior ones(in event clk, in unsigned bit[0:0] rst, in unsigned bit[31:0] idata,
        out unsigned bit[31:0] iocount, in unsigned bit[0:0] start,
        out unsigned bit[0:0] done, out unsigned bit[0:0] ack_istart,
        in unsigned bit[0:0] ack_idone)
{

    note ones.scheduled = "1";
    note ones.fubind = "1";
    note ones.regbind = "1";
    note ones.busbind = "1";
    note ones.ack_idone = "ctrl";
    note ones.ack_istart = "ctrl";
    note ones.clk = "clk";
    note ones.done = "ctrl";
    note ones.idata = "data";
    note ones.iocount = "data";
    note ones.rst = "rst";
    note ones.start = "ctrl";


    bit[31:0] shift0(bit[31:0] si, bit[31:0] amount, int ctrl)
    {
        return shift(si, amount, ctrl);
    }

    bit[31:0] alu0(bit[31:0] a, bit[31:0] b, int ctrl)
    {
        return alu(a, b, ctrl);
    }

    bit[31:0] alu1(bit[31:0] a, bit[31:0] b, int ctrl)
    {
        return alu(a, b, ctrl);
    }

    void main(void)
    {
        bit[31:0] Reg320;
        bit[31:0] Reg321;
        bit[31:0] Reg322;
        bit[31:0] Reg323;
        bit[0:0] _ctrl_;
```

```
unsigned bit[31:0] data;
unsigned bit[31:0] mask;
unsigned bit[31:0] ocount;
unsigned bit[31:0] temp;
bit[31:0] bus0;
bit[31:0] bus1;
bit[31:0] bus2;
bit[31:0] bus3;
bit[31:0] bus4;
enum state { S0, S1, S2, S3, X0 } state;
while (1)
{
    wait(clk);
    if (rst)
    {
        state = S0;
    }
    switch (state)
    {
        case S0 :
        {
            ack_istart = 0;
            done = 0;
            if (start!=0)
            {
                state = S1;
            }
            else
            {
                state = S0;
            }
            break;
        }
        case S1 :
        {
            mask = 1;
            ocount = 0;
            data = idata;
            ack_istart = 1;
            state = S2;
            break;
        }
        case S2 :
        {
            bus0 = data;
            bus1 = mask;
            bus3 = alu0(bus0, bus1, 8);
            temp = bus3;
            bus2 = shift0(bus0, bus1, 0);
            data = bus2;
            state = X0;
            break;
        }
```

```
                    case X0 :
                    {
                        bus2 = ocount;
                        bus1 = temp;
                        bus3 = alu0(bus2, bus1, 0);
                        ocount = bus3;
                        bus0 = data;
                        if (alu1(bus0, 0, 7))
                        {
                            state = S3;
                        }
                        else
                        {
                            state = S2;
                        }
                        break;
                    }
                    case S3 :
                    {
                        done = 1;
                        bus2 = ocount;
                        iocount = bus2;
                        if (ack_idone==1)
                        {
                            state = S0;
                        }
                        else
                        {
                            state = S3;
                        }
                        break;
                    }
                }
            }
        }
};
```

E. Output: One's Counter in Style 4 - Design 4

```
/*******************************************************
* SpecC code generated by 'genc'
* Date: Thu Feb 21 22:11:32 2002
* User: qxie
*******************************************************/
import "lib";



behavior ones(in event clk, in unsigned bit[0:0] rst, in unsigned bit[31:0] idata,
        out unsigned bit[31:0] iocount, in unsigned bit[0:0] start,
        out unsigned bit[0:0] done, out unsigned bit[0:0] ack_istart,
        in unsigned bit[0:0] ack_idone)
{

    note ones.scheduled = "1";
    note ones.fubind = "1";
    note ones.regbind = "1";
    note ones.busbind = "1";
    note ones.ack_idone = "ctrl";
    note ones.ack_istart = "ctrl";
    note ones.clk = "clk";
    note ones.done = "ctrl";
    note ones.idata = "data";
    note ones.iocount = "data";
    note ones.rst = "rst";
    note ones.start = "ctrl";


    bit[31:0] shift0(bit[31:0] si, bit[31:0] amount, int ctrl)
    {
        return shift(si, amount, ctrl);
    }

    bit[31:0] alu0(bit[31:0] a, bit[31:0] b, int ctrl)
    {
        return alu(a, b, ctrl);
    }

    void main(void)
    {
        bit[31:0] RF0[4];
        bit[0:0] _ctrl_;
        unsigned bit[31:0] data;
        unsigned bit[31:0] mask;
        unsigned bit[31:0] ocount;
        unsigned bit[31:0] temp;
        bit[31:0] bus0;
        bit[31:0] bus1;
        bit[31:0] bus2;
        enum state { S0, S1, S2, S3, X0, X1, X2, X3, X4, X5 } state;
```

```
while (1)
{
    wait(clk);
    if (rst)
    {
        state = S0;
    }
    switch (state)
    {
        case S0 :
        {
            ack_istart = 0;
            done = 0;
            if (start!=0)
            {
                state = S1;
            }
            else
            {
                state = S0;
            }
            break;
        }
        case S1 :
        {
            ack_istart = 1;
            RF0[0] = idata;
            state = X0;
            break;
        }
        case X0 :
        {
            RF0[1] = 0;
            state = X1;
            break;
        }
        case X1 :
        {
            RF0[2] = 1;
            state = S2;
            break;
        }
        case S2 :
        {
            bus0 = RF0[0];
            bus1 = RF0[2];
            bus2 = alu0(bus0, bus1, 8);
            RF0[3] = bus2;
            state = X2;
            break;
        }
        case X2 :
        {
```

```
                    bus0 = RF0[0];
                    bus1 = RF0[2];
                    bus2 = shift0(bus0, bus1, 0);
                    RF0[0] = bus2;
                    state = X3;
                    break;
                }
                case X3 :
                {
                    bus0 = RF0[1];
                    bus1 = RF0[3];
                    bus2 = alu0(bus0, bus1, 0);
                    RF0[1] = bus2;
                    state = X4;
                    break;
                }
                case X4 :
                {
                    bus0 = RF0[0];
                    _ctrl_ = alu0(bus0, 0, 7);
                    state = X5;
                    break;
                }
                case X5 :
                {
                    if (_ctrl_)
                    {
                        state = S3;
                    }
                    else
                    {
                        state = S2;
                    }
                    break;
                }
                case S3 :
                {
                    done = 1;
                    if (ack_idone==1)
                    {
                        state = S0;
                    }
                    else
                    {
                        state = S3;
                    }
                    bus0 = RF0[1];
                    iocount = bus0;
                    break;
                }
            }
        }
    }
```

```
};
```

F. Output: One's Counter in Style 4 - Design 5

```
/*****************************************************
* SpecC code generated by 'genc'
* Date: Thu Feb 21 22:19:05 2002
* User: qxie
*****************************************************/
import "lib";



behavior ones(in event clk, in unsigned bit[0:0] rst, in unsigned bit[31:0] idata,
        out unsigned bit[31:0] iocount, in unsigned bit[0:0] start,
        out unsigned bit[0:0] done, out unsigned bit[0:0] ack_istart,
        in unsigned bit[0:0] ack_idone)
{

    note ones.scheduled = "1";
    note ones.fubind = "1";
    note ones.regbind = "1";
    note ones.busbind = "1";
    note ones.ack_idone = "ctrl";
    note ones.ack_istart = "ctrl";
    note ones.clk = "clk";
    note ones.done = "ctrl";
    note ones.idata = "data";
    note ones.iocount = "data";
    note ones.rst = "rst";
    note ones.start = "ctrl";


    bit[31:0] shift0(bit[31:0] si, bit[31:0] amount, int ctrl)
    {
        return shift(si, amount, ctrl);
    }

    bit[31:0] alu0(bit[31:0] a, bit[31:0] b, int ctrl)
    {
        return alu(a, b, ctrl);
    }

    bit[31:0] alu1(bit[31:0] a, bit[31:0] b, int ctrl)
    {
        return alu(a, b, ctrl);
    }

    void main(void)
    {
        bit[31:0] Reg320;
        bit[31:0] Reg321;
        bit[31:0] Reg322;
        bit[31:0] Reg323;
        bit[0:0] _ctrl_;
```

```
unsigned bit[31:0] data;
unsigned bit[31:0] mask;
unsigned bit[31:0] ocount;
unsigned bit[31:0] temp;
bit[31:0] bus0;
bit[31:0] bus1;
bit[31:0] bus2;
bit[31:0] bus3;
bit[31:0] bus4;
enum state { S0, S1, S2, S3, X0, X1, X2 } state;
while (1)
{
    wait(clk);
    if (rst)
    {
        state = S0;
    }
    switch (state)
    {
        case S0 :
        {
            ack_istart = 0;
            done = 0;
            if (start!=0)
            {
                state = S1;
            }
            else
            {
                state = S0;
            }
            break;
        }
        case S1 :
        {
            ack_istart = 1;
            ocount = 0;
            mask = 1;
            data = idata;
            state = S2;
            break;
        }
        case S2 :
        {
            bus0 = data;
            bus1 = mask;
            bus2 = alu0(bus0, bus1, 8);
            temp = bus2;
            bus3 = shift0(bus0, bus1, 0);
            data = bus3;
            state = X0;
            break;
        }
```

```
                case X0 :
                {
                    state = X1;
                    break;
                }
                case X1 :
                {
                    bus0 = data;
                    _ctrl_ = alu0(bus0, 0, 7);
                    bus1 = ocount;
                    bus2 = temp;
                    bus4 = alu1(bus1, bus2, 0);
                    ocount = bus4;
                    state = X2;
                    break;
                }
                case X2 :
                {
                    if (_ctrl_)
                    {
                        state = S3;
                    }
                    else
                    {
                        state = S2;
                    }
                    break;
                }
                case S3 :
                {
                    done = 1;
                    if (ack_idone==1)
                    {
                        state = S0;
                    }
                    else
                    {
                        state = S3;
                    }
                    bus1 = ocount;
                    iocount = bus1;
                    break;
                }
            }
        }
    }
};
```