An Analysis of Verilog Software Design Techniques on Hardware Implementations

Justin D. Ehlert
Department of Computer Science
University of Wisconsin – Eau Claire
ehlertjd@uwec.edu

Andrew T. Phillips
Department of Computer Science
University of Wisconsin – Eau Claire
phillipa@uwec.edu

Abstract

Verilog is a hardware description language that allows hardware designers to use a software style approach, via a high level programming language, to construct realizable hardware implementations reflected in a blueprint called a "netlist." This paper presents a study of the various software coding styles and constructs available in Verilog and the effect of those design style choices on the resulting hardware implementations. Using a Verilog compiler and a netlist viewer, we show that different software design constructs often result in radically different implementations. In particular, we show that when using the data flow and behavioral Verilog coding styles, it is not uncommon for significant cost increases to occur as a result of some common software design approaches. A collection of such inefficient software design techniques will be described, and improved approaches will be demonstrated.

Introduction

Verilog [1,2] is a hardware description language that allows hardware designers to use a software approach (using high level programming constructs) to build realizable hardware implementations, typically on a field programmable gate array (FPGA). Since many different coding styles and constructs are available in Verilog, there are many differing hardware implementations that might result from these styles and constructs. This paper describes the results of our study on the effect of these various Verilog constructs on the corresponding hardware implementations. We found these results to be quite significant, and in some cases even surprising.

Verilog supports three main coding styles: structural, data flow, and behavioral. Structural style uses gate-level constructs (e.g. AND, OR, NOT) to directly represent logic gates in a manner that is both simple and straight-forward, but convenient only if the hardware designer already knows the desired implementation details beforehand. In contrast, data flow style uses a more "programming-like" approach to represent logic gates, thereby eliminating the need for gate level primitives and, more importantly, a priori knowledge of the target implementation that will result. But the data-flow style has its disadvantages as well; sequential circuits cannot be modeled, and designs based on conditional logic are cumbersome and tedious to implement. Finally, the behavioral style closely mimics high level software design by using procedural constructs (loops, conditional statements, etc.) to determine the actions to be performed as a result of changes in input signals, thereby permitting sequential circuit design while also eliminating the need to know beforehand how the design will be implemented. Of these three styles, the structural style most closely represents the actual hardware implementation that will result, while the behavioral style most closely represents the intended behavior, or high level design, of the circuit.

Our study evaluates various Verilog software design constructs using each of the three coding styles, and classifies these constructs according to the quality of the resulting hardware implementation. We found that structural style did indeed directly map to the hardware implementation, without any surprising or unexpected results. However, as we've noted above, the structural style forces the hardware designer to manually design "pre-build" all of the implementation detail, rather than permit the Verilog compiler to perform that task. For this reason, the structural style is not often preferred for even moderately complex designs. In contrast to the structural style, the results of both the data flow and behavioral styles surprised us at times. Among the most interesting findings are the alternative implementations that result from a variety of conditional statements including nested if/else and case statements. Common software design solutions that use if/else statements often result in inefficient multi-level, or cascaded, combinational logic including multiple two-input multiplexors, whereas case statements usually result in a single large multiplexer implementation without any cascading of combinational logic.

Because there are a variety of ways that software designers can use the Verilog HDL to implement combinational and sequential logic structures, each of these possible approaches may result in a different hardware realization, and in a corresponding cost for

that realization. In light of these styles, we wish to consider a variety of software design approaches and to select those that lead to the most cost efficient implementations. For that purpose we use two separate cost measures. The first measure is the sum of the total gate count and the total fan-in to those gates, excluding any contributions from inverters applied to input values (since most input signals are available within a logic design in both inverted and un-inverted forms already). This is an important and quite common metric because it measures the total amount of small scale integration (SSI) logic required to implement a design; smaller gate count / fan-in is better than larger. The second measure is the total delay through the circuit (again, discounting any delays from inverters applied to input values). Of course, shorter delay is better than longer delay.

In the remainder of this paper, we will illustrate the various coding styles and constructs, and demonstrate their effects on the resulting hardware implementation.

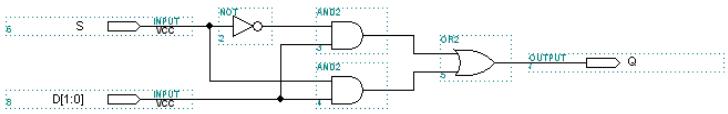
2x1 MUX

To illustrate these two cost metrics with an example, we show in Figure 1 an implementation of a 2x1 multiplexor (MUX) designed in Verilog using the structural coding style. In this style, the hardware designer must have a priori knowledge of the gate level implementation desired, and the result is always a direct mapping of software constructs to hardware gates. While this approach is simple and direct, it is not often used by hardware designers on larger and more complex designs because it replaces high level behavioral or functional design concepts with structure specific ones. In short, it requires design at a much lower level than is either necessary or desired. However, in simple examples such as this, it does permit us to compare various designs directly to known optimal ones.

The 2x1 MUX uses a 1 bit selector signal S to "select for output" exactly one of two possible inputs lines denoted D[1:0]. While the 2x1 MUX is particularly simple, it happens to be a fundamental building block for many designs to be discussed shortly.

Figure 1: Structural Style 2x1 MUX

```
module mux2x1(Q,D,S);
   input [1:0] D;
   input S;
   output Q;

   wire Sn;
   wire R0, R1;

   not(Sn,S);

   and(R1,S,D[1]);
   and(R0,Sn,D[0]);

   or(Q,R0,R1);
endmodule
```



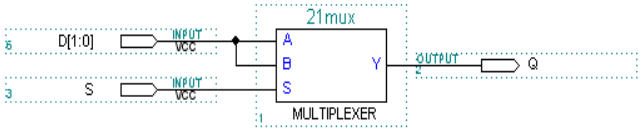
Cost = 9:2T

The gate count for the optimally implemented 2x1 MUX shown above is 3 + 6 = 9 gates with a total delay of 2T, which we will denote as 9:2T. We now compare three competing

software design styles, all in Verilog, for implementing this same 2x1 MUX. We do this as a means for illustrating some basic Verilog software design styles, even though the results in this simple case are indistinguishable. Figure 2 provides the most common data flow style solution to the 2x1 MUX design, while Figure 3 and Figure 4 are typical behavioral style approaches. We've also included Figure 5 and Figure 6 to round out the most likely approaches to 2x1 MUX design, even though we doubt these styles would be common for software developers.

Figure 2: Data Flow Style 2x1 MUX (Conditional Operator)

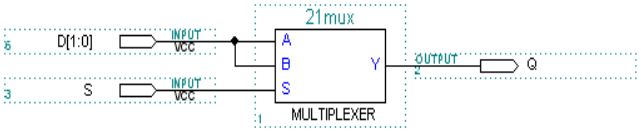
```
module mux2x1(Q,D,S);
   input [1:0] D;
   input S;
   output Q;

   assign Q = S ? D[1] : D[0];
endmodule
```



Cost = 9:2T

Figure 3: Behavioral Style 2x1 MUX (If/Else Statement)

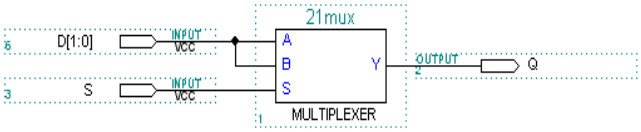
```
module mux2x1(Q,D,S);
   input [1:0] D;
   input S;
   output Q;
   reg Q;

   always @(S or D)
   begin
      if (S)
         Q = D[1];
      else
         Q = D[0];
   end
endmodule
```



Cost = 9:2T

Figure 4: Behavioral Style 2x1 MUX (Case Statement)

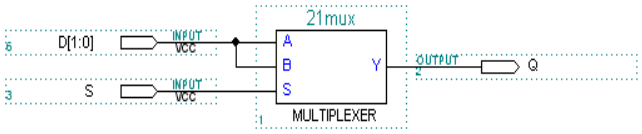
```
module mux2x1(Q,D,S);
   input [1:0] D;
   input S;
   output Q;
   reg Q;

   always @(S or D)
   begin
      case (S)
      1'b0 : Q = D[0];
      1'b1 : Q = D[1];
      endcase
   end
endmodule
```



Cost = 9:2T

Figure 5: Behavioral Style 2x1 MUX (If Statement)

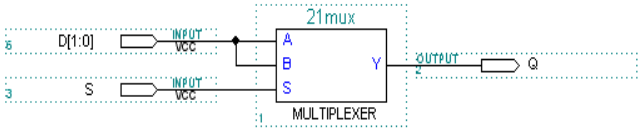
```
module mux2x1(Q,D,S);
   input [1:0] D;
   input S;
   output Q;
   reg Q;

   always @(S or D)
   begin
      Q = D[0];
      if (S)
         Q = D[1];
   end
endmodule
```



Cost = 9:2T

Figure 6: Behavioral Style 2x1 MUX (Defaulted Case Statement)

```
module mux2x1b3(Q,D,S);
   input [1:0] D;
   input S;
   output Q;
   reg Q;

   always @(S or D)
   begin
      case (S)
      1'b0 : Q = D[0];
      default : Q = D[1];
      endcase
   end
endmodule
```



Cost = 9:2T

Note that regardless of the software design style selected above, the resulting implementation for the 2x1 MUX is precisely the same. This is typically true only for the most simple of designs. While other possible data flow and behavioral approaches are possible here as well, the results are typically no different.

4x1 MUX

We now examine a more interesting case, the 4x1 multiplexor. In the 4x1 MUX, we wish to use a pair of input bits S[1:0] to select one of four possible outputs D[3:0]. Here is the simple pseudocode representing the basic idea (note that this example is not intended to represent a good software approach to implementing this structure):

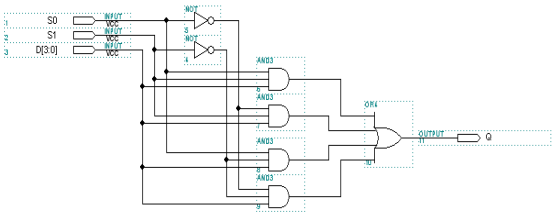
```
if (S[1:0] == 00) {
   Q = D[0];
} else if (S[1:0] == 01) {
   Q = D[1];
} else if (S[1:0] == 10) {
   Q = D[2];
} else {
   Q = D[3];
}
```

In general, a 2^Nx1 MUX uses an N bit selector input S[N-1:0] to select one of 2^N inputs for output on a 1 bit line. Of course multiplexors of various sizes exist, and this example can be easily generalized to any of those other sizes.

As we demonstrated for the 2x1 MUX case, there are a number of ways that a software designer can potentially implement this logic structure. Figure 7 shows a cost optimal implementation of the 4x1 MUX done in Verilog using the structural coding style.

Figure 7: Structural Style 4x1 MUX

```
module mux4x1(Q,D,S);
   input [3:0] D;
   input [1:0] S;
   output Q;

   wire [1:0] Sn;
   wire [3:0] R;

   not(Sn[0],S[0]);
   not(Sn[1],S[1]);

   and(R[3],S[1],S[0],D[3]);
   and(R[2],S[1],Sn[0],D[2]);
   and(R[1],Sn[1],S[0],D[1]);
   and(R[0],Sn[1],Sn[0],D[0]);

   or(Q,R[0],R[1],R[2],R[3]);
endmodule
```



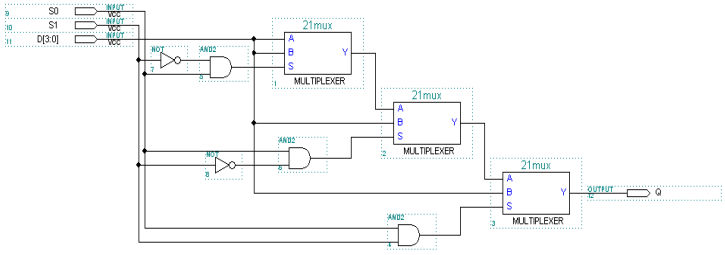
Cost = 21:2T

The cost of this implementation is clearly 21:2T.

We continue now by comparing two very similar behavioral approaches for designing the 4x1 MUX. While both use a nested if/else structure, one uses Boolean logic while the other uses bitwise operators in the conditional tests. We claim that both of these approaches are reasonably likely choices for classically trained software developers. We also note that they both result in the same implementation with cost of 30:7T, among the highest cost hardware implementations of the 4x1 MUX designs we will discuss!

Figure 8: Behavioral Style 4x1 MUX (Boolean Logic Nested If/Else)

```
module mux4x1(Q,D,S);
   input [3:0] D;
   input [1:0] S;
   output Q;
   reg Q;

   always @(S or D)
   begin
      if (S[1] && S[0])
         Q = D[3];
      else if (S[1] && !S[0])
         Q = D[2];
      else if (!S[1] && S[0])
         Q = D[1];
      else
         Q = D[0];
   end
endmodule
```

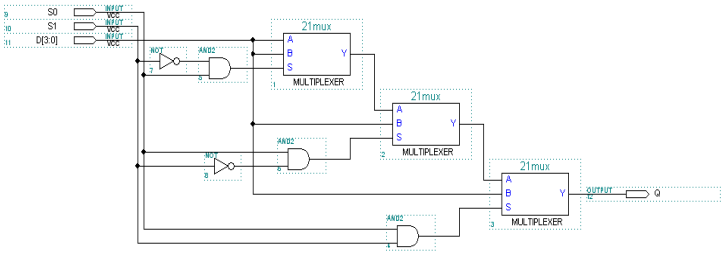


Cost = 30:7T

It is clear from both Figure 8 and Figure 9 that, regardless of whether the conditions are determined by Boolean algebra or by bitwise operations, the resulting design involves primitive gates that dictate the select functions of each 2x1 MUX, where one MUX is associated with each conditional test.

Figure 9: Behavioral Style 4x1 MUX (Bitwise Logic Nested If/Else)

```
module mux4x1(Q,D,S);
   input [3:0] D;
   input [1:0] S;
   output Q;
   reg Q;

   always @(S or D)
   begin
      if (S[1] & S[0])
         Q = D[3];
      else if (S[1] & ~S[0])
         Q = D[2];
      else if (~S[1] & S[0])
         Q = D[1];
      else
         Q = D[0];
   end
endmodule
```

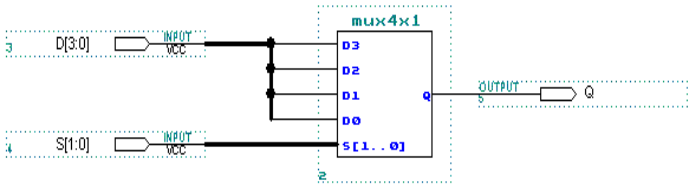


Cost = 30:7T

Figure 10 shows the same approach one last time, but now directly using a binary encoding of the selectors. This simple change results in the optimal 21:2T cost implementation, the same as the structural style approach.

Figure 10: Behavioral Style 4x1 MUX (Binary Encoded Nested If/Else)

```
module mux4x1(Q,D,S);
   input [3:0] D;
   input [1:0] S;
   output Q;
   reg Q;

   always @(S or D)
   begin
      if (S == 2'b11)
         Q = D[3];
      else if (S == 2'b10)
         Q = D[2];
      else if (S == 2'b01)
         Q = D[1];
      else
         Q = D[0];
   end
endmodule
```

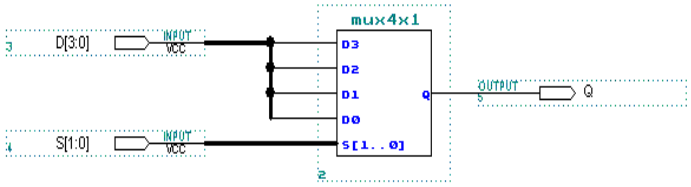


Cost = 21:2T

The key property of this software design style (the property that makes it the most cost efficient) is that the conditions tested use *constant values* that, when taken as a whole, cover the range of all possible 2 bit inputs of S. Of course, this approach should motivate the use of a case statement, and Figure 11 below indeed shows that the case statement is equivalent to this.

Figure 11: Behavioral Style 4x1 MUX (Case Statement)

```
module mux4x1(Q,D,S);
   input [3:0] D;
   input [1:0] S;
   output Q;
   reg Q;

   always @(S or D)
   begin
      case(S)
      2'b11 :   Q = D[3];
      2'b10 :   Q = D[2];
      2'b01 :   Q = D[1];
      default : Q = D[0];
      endcase
   end
endmodule
```



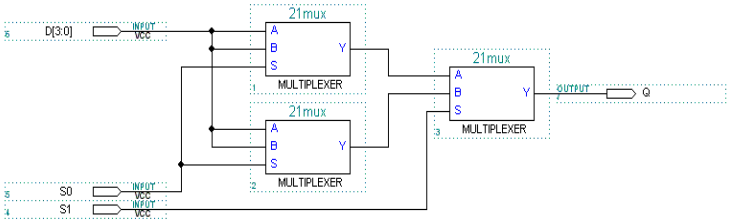
Cost = 21:2T

The use of the `default` label really has no effect here so long as all of the possible 2 bit select values are explicitly listed; that is, the `default` case label could have been replaced by the constant `2'b11` with the same result.

There are a number of more complex alternative approaches that, while unlikely choices for software design styles, are enlightening when compared with the resulting implementations. The first such approach, shown in Figure 12, is to treat the 2 bit selector as two separate 1 bit selectors using structured and nested conditional statements. While we doubt any software developers would follow this cryptic approach, we do note that it is more cost efficient (at 27:4T) than the cascaded MUX designs since it involves the use of only three 2x1 MUXs (in a tree like structure), as opposed to the four cascaded MUXs shown in Figure 8 and Figure 9 above.

Figure 12: Behavioral Style 4x1 MUX (Nested Structured Conditionals)

```
module mux4x1(Q,D,S);
   input [3:0] D;
   input [1:0] S;
   output Q;
   reg Q;

   always @(S or D)
   begin
      if (S[1] == 0)
         if (S[0] == 0)
            Q = D[0];
         else
            Q = D[1];
      else
         if (S[0] == 0)
            Q = D[2];
         else
            Q = D[3];
   end
endmodule
```



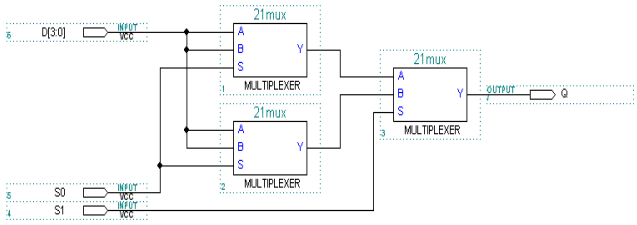
Cost = 27:4T

Logically, this same idea can also be implemented using a nested conditional operator, an approach which is not at all likely to be used by software engineers (even though it is fairly common among hardware engineers). Nevertheless, the results of this style are shown in Figure 13 and are exactly the same as in Figure 12 above. While software

designers are often loath to use the conditional operator, hardware designers know that it *always* results in a 2x1 MUX in the implementation, as can be seen from Figure 13. Hence, it is a fairly common HDL coding construct.

Figure 13: Behavioral Style 4x1 MUX (Nested Conditional Operators)

```
module mux4x1(Q,D,S);
   input [3:0] D;
   input [1:0] S;
   output Q;
   reg Q;

   always @(S or D)
   begin
      Q = S[1] ? (S[0] ? D[3] : D[2])
               : (S[0] ? D[1] : D[0]);
   end
endmodule
```



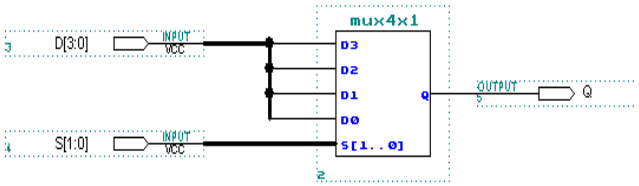
Cost = 27:4T

It is important to remember that the cost efficiency of these approaches is not simply a matter of the choice of a case statement versus nested conditional statements or conditional operators. As we stated earlier, the optimal implementation is produced when the conditions to be tested can be clearly mapped to *constants* that can be used in case labels, and when the conditions tested cover the range of constant values that could occur.

As a final remark, we note by way of Figure 14 below that it also is possible to generate the optimal 21:2T cost approach using the data flow style. While we don't recommend this approach due its lack of readability, it does provide proof that the three distinct Verilog coding styles are each capable of producing the identical results, even if it requires some clever and cryptic coding.

Figure 14: Data Flow Style

```
module mux4x1(Q,D,S);
   input [3:0] D;
   input [1:0] S;
   output Q;

   assign Q = (S[0] & S[1] & D[3]) |
              (S[0] & ~S[1] & D[2]) |
              (~S[0] & S[1] & D[1]) |
              (~S[0] & ~S[1] & D[0]);
endmodule
```



Cost = 21:2T

Traditional Software Techniques

Logic design in an HDL like Verilog is not like traditional software design. As one simple example of this, the attempt to represent a MUX (of any size) by the data flow style logic

```
assign Q = D[S];
```

or by the behavioral style logic

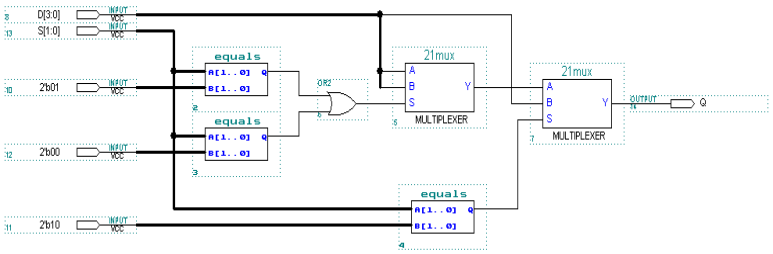
```
always @(S or D)
begin
   Q = D[S];
end
```

is not permitted. While this is a perfectly reasonable software design representation of a MUX, the non-constant index into D, represented by S in this case, is not permitted. For vectored signals such as D, the index is called a *bit select* (for a single bit of D) or part *select* (for multiple bits of D), and this index must be constant. In short, D is not an array but rather a multi-bit signal (bus) that requires a constant selector in order for the logic connections to be determined at compile time.

Similarly, consider the example in Figure 15 of a 4x1 MUX in which two of the select cases are identical (S = `2'b00` and S = `2'b01` both select D[1]). Software developers are likely to handle the two identical cases by combining the logical conditions into a single Boolean expression as shown.

Figure 15: 4x1 MUX with Combined Conditional Cases

```
module mux4x1(Q,D,S);
   input [3:0] D;
   input [1:0] S;
   output Q;
   reg Q;

   always @(S or D)
   begin
      if (S == 2'b00 ||
          S == 2'b01)
         Q = D[1];
      else if (S == 2'b10)
         Q = D[2];
      else
         Q = D[3];
   end
endmodule
```

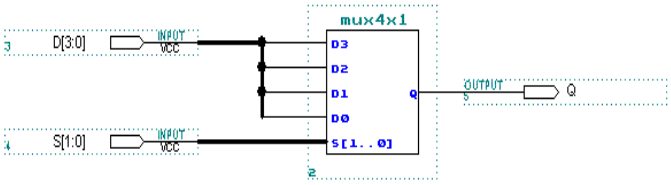


Cost = 46:7T

Unfortunately, the resulting implementation is very inefficient with a cost of 46:7T, assuming that the 2 bit "equals" comparators can be efficiently implemented in 9:2T using primitive gates including xor. On the other hand, by avoiding compound Boolean conditions and either testing each case separately or by permitting the final fall through else clause to handle the combined cases, the result is a 4x1 MUX with cost 21:2T and two ports wired together (ports D1 and D0 in Figure 16). The lesson to be learned here is that one pays for compact Boolean logic in software with additional hardware logic as well as delays in the implementation. That is, in most instances the extra software effort required to itemize the cases will often result in more efficient hardware.

Figure 16: Duplicated Conditional Cases

```
module mux4x1(Q,D,S);
   input [3:0] D;
   input [1:0] S;
   output Q;
   reg Q;

   always @(S or D)
   begin
      if (S == 2'b11)
         Q = D[3];
      else if (S == 2'b10)
         Q = D[2];
      else // fall thru cases
         Q = D[1];
   end
endmodule
```

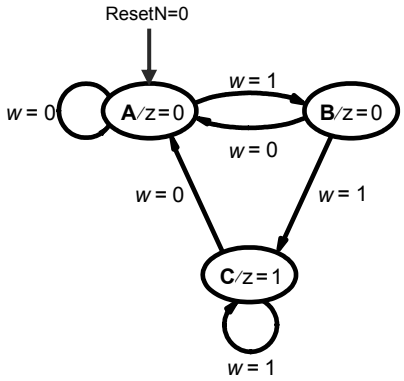


Cost = 21:2T

Applications to Finite State Machine Design

The importance of the results is easily demonstrated by a practical example involving finite state machine design, a very common technique for designing the control of sequential circuits. Most sequential circuits are designed and implemented using finite state machines, employing either the Moore or the Mealy model [3]. In Moore machines, the output is governed solely by the current state of the machine, whereas Mealy machine output is a combination of both the current state and the current input, thus producing a level-sensitive output. As a simple example, Figure 17 presents a finite state diagram representing a Moore style model that outputs z=1 only when two or more consecutive inputs w are 1.

Figure 17: Moore FSM for Detecting Two Consecutive 1's



The Verilog for a Moore machine is remarkably simple, and uses many of the design principles we have investigated above. In general, a Moore machine can be structured in Verilog to have three sections: one for defining the state transitions (based on both the current state and the current input value), one for defining the outputs (associated with each state, but independent of the inputs), and one for the actual transition that occurs only on a clock edge. The first two sections are entirely combinational is nature, and often use only the design principles we have discussed previously. The third section

involves sequential logic since the transitions must occur on clock edges, but the logic involved is very simple. Figure 18 illustrates this in Verilog for the Moore machine "two 1's" problem:

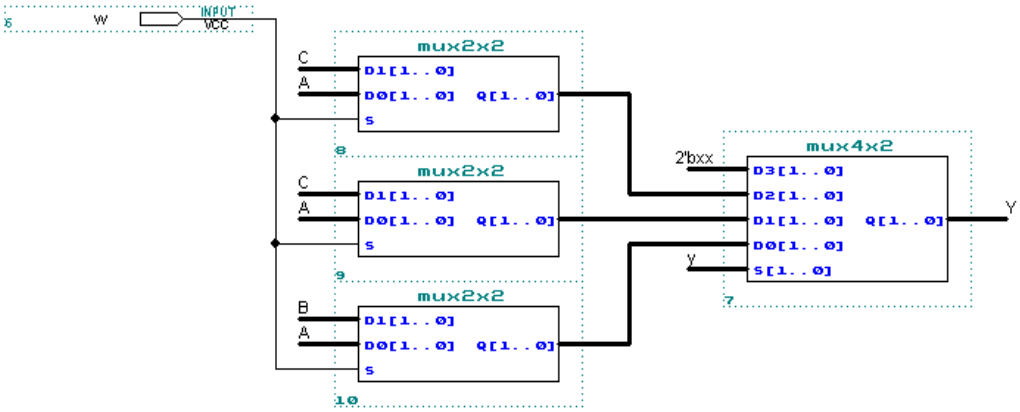
Figure 18: Two 1's Moore Style Finite State Machine

```
module MooreFSM(z,w,ResetN,Clock);
   input w, ResetN, Clock;
   output z;
   reg [1:0] y, Y;

   // the state assignment
   parameter A = 2'b00, B = 2'b01, C = 2'b10;

   // define the FSM combinational logic
   // y represents the current state
   // Y represents the next state
   // w is the input that controls the next state selection
   always @(w or y)
      case (y)
      A: Y <= (w ? B: A);
      B: Y <= (w ? C: A);
      C: Y <= (w ? C: A);
      default: Y <= 2'bxx;
      endcase

   // define the combinational output logic
   assign z = (y == C);

   // define the sequential state change logic
   always @(negedge ResetN or posedge Clock)
      if(!ResetN)
         y <= A;
      else
         y <= Y;
endmodule
```

The input values to this module are w, ResetN, and Clock; the output value is z. It is customary to use lowercase y to define the current state and uppercase Y as a wire to define the next state to enter on a clock edge.

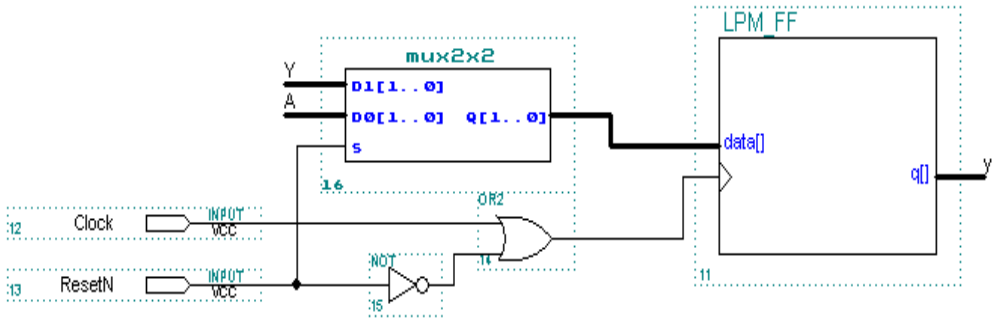
By applying our best results from the earlier analysis of various software design styles for implementing multiplexers, we can easily estimate the cost of this machine. In the first behavioral style block involving the state transitions, the four value case statement will infer a 4x2 MUX (cost = 21:2T per bit), since all cases are enumerated by two bit constants. The three conditional statements that are inputs to that 4x2 MUX will result in three (possibly even two since two cases are identical) 2x2 MUXs (cost for each = 9:2T per bit). Hence, we can infer the hardware implementation shown in Figure 19. In our solution, we have assumed that the two identical MUXs will be duplicated simply to make the point that, in general, a four state Moore machine design would have the format exactly as shown.

Figure 19: FSM Combinational Logic for the State Transition Behavioral Block



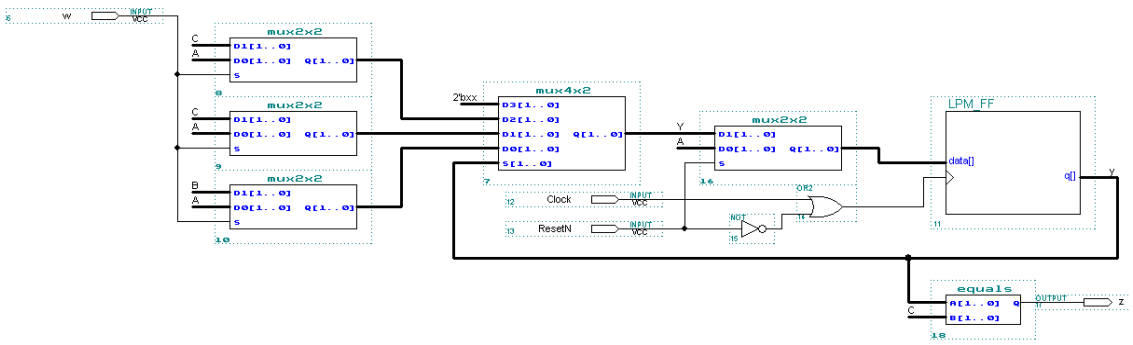
Similar to the behavioral style combinational logic for computing the next state, the sequential logic for actually making that transition on the next clock edge also involves a MUX. In this case, the result is always a 2x2 MUX (dependent only on the number of states in the FSM, not on the problem itself) with cost 9:2T per bit. The output of that MUX sets a D flip-flop of the appropriate size, where the clock signal of the flip-flop is driven by both the Clock and ResetN signals.
Figure 20 illustrates this.

Figure 20: Sequential Logic for State Transitions



Finally, adding in the equals comparator logic (cost = 9:2T) for the combinational output logic results in a final design shown in Figure 21.

Figure 21: Final Moore Style Two 1's Design



The final gate/fan-in cost (discounting delay since this is a sequential circuit) of this design is therefore 125. Design of finite state machines using the Mealy style model is not substantially different.

In conclusion, as a result of our analysis of simpler designs, we are able to accurately estimate and optimize the cost of hardware implementations that result from carefully chosen software design constructs. An understanding of the effects of software style can therefore make a significant difference in the final cost of a digital system implementation.

References

1. Bhasker, J. (1999). *A Verilog HDL Primer*. Star Galaxy Publishing.
2. Palnitkar, S. (1996). *Verilog HDL: A Guide to Digital Design and Synthesis.* SunSoft Publishing.
3. Hopcroft, J.E., and Ullman, J.D. (1979). *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley Publishing.

Acknowledgements

The authors would like to acknowledge the Office of Research and Sponsored Programs at UW – Eau Claire for their sponsorship of this research project.