An Efficient Method for Complex Digital Systems Design Using Verilog

Wayne Lu
Department of Electrical Engineering and Computer Science
University of Portland
Portland, OR 97203

Abstract

Verilog is a very popular hardware description language for ASIC (Application Specific Integrated Circuit) design. Students can learn Verilog programming in various courses. Although Verilog is easy to learn due to its similarity to the C language, however, it still will take some time before a student can write efficient synthesizable Verilog code for designing complex digital systems. This paper presents a quick method to implement complicated digital systems before students have reached such a proficiency level of using Verilog. The method is to design a target digital system using an extremely easy to learn ABEL language as the building blocks and then translate the ABEL programs to the Verilog syntax. The system operations of the converted Verilog programs can be verified by programming a CPLD (complex programmable logic device) or FPGA (field programmable gate array) device.

Introduction

Many digital system designers use Verilog to implement digital ASIC designs. Students can learn Verilog programming in various courses. Although Verilog is easy to learn due to its similarity to the C language, however, it still will take some time before a student can write efficient synthesizable Verilog code for designing complex digital systems. Often times, there is a need to quickly implement a complicated digital system using a CPLD (complex programmable logic device) or FPGA (field programmable gate array). Verilog certainly will be the choice of language for those who have already learned the language, but how about students without the proficiency of using Verilog? To address this need, this Paper presents a quick way to implement complicated digital systems using Verilog. The strategy is to spend a few hours learning the extremely easy ABEL language and then translate it to the Verilog syntax. Since ABEL statements are closely related to logic gates and flip-flops, the translated Verilog code will be completely synthesizable.

A Sample ABEL Program

The ABEL language is so easy to learn that students can comfortably learn ABEL programming within a few hours [1]. ABEL defines a digital design through a module and is very efficient for a single module design targeted for a single PLD regardless of the design complexity. The module can be as simple as a single gate or as complicated as a microcomputer [2]. The ABEL syntax is simple enough to be outlined in the following short paragraph.

Each design begins with the **module** keyword followed by a module name. A **title** statement uses a pair of single quotes to provide documentation information. Afterwards, each ABEL statement is terminated by a semicolon. A **device** statement specifies the target device used by the design. The **pin** statements define a digital circuit's input and output signal names. The **istype** keyword in a pin statement defines an output to be a combinational **('com'**) or registered **('reg'**) signal. A single line comment is indicated by a double quote. Any intermediate labels can be defined without first declaring the labels once the signals or variables involved have been previously defined. The logic equations for the output signals are defined in the **equations** section. A state machine is defined by a **state_diagram** statement which specifies the next state transitions using **if-then-else** statements based on the current state and current input values. A **GOTO** statement specifies a forced next state transition. The state machine outputs can be embedded in the state transition statements or specified in the "equations" section. The **end** module_name statement ends the module.

For example the ABEL program, doors.abl (Listing 1), implements a state machine and combinational logic for a 4-story elevator controller. The state machine consists of four D flip-flops for simulating the elevator doors' closing and opening movements. The combinational logic generates arrival signals to another state machine to trigger state transitions.

Converting ABEL to Verilog

Verilog also defines a digital circuit in module. The complexity of a module can also be as simple as a gate or as complicated as a microcomputer [3]. But the main difference is that Verilog is very efficient for multiple-module hierarchical top-down designs. Each module can be individually designed, tested, and then integrated with other modules to form a more complicated module. Those more complicated modules can be integrated to form an even more complicated module and on and on. Eventually the entire system consists of only a few complicated modules in a style very similar to the main() of a C program consisting of a few subroutine calls. It is this hierarchical design capability that provides Verilog the flexibility and efficiency so crucially demanded in ASIC designs.

Due to the similarity, an ABEL program can be easily converted to Verilog syntax [3], [4]. The conversion process involves very straightforward editing steps as detailed below.

1. Add a port list including all input and output signals after the module name, delete title and device statements, make module and endmodule keywords lower case letters.

2. Add the **input** and **output** keywords for input and output signals. Delete the active-low signal polarity prefix (!) and pin declarations. Change the ABEL comment indicator to //.

3. Declare **wire** data type variables for intermediate equations.

4. Convert all intermediate and equations statements to **assign** statements. Change ! (not) to ~, # (OR) to |. Add a ~ (not) before active-low input variables.

5. Declare a state machine's state, next state, and output variables as **reg** data type.

6. Prefix a **parameter** keyword to state definitions and change ^b to 7'b, 6'b, 4'b, 3'b etc. depending on the number of bits defined for a value. Change a set definition into individual **assign** statements for each variable defined in the set.

7. Change each state-diagram statement into three **always** statements. The first **always** statement updates the next state variable based on the state transitions through a **case** statement with the state variables and state machine inputs included in the sensitivity list after the **always** keyword. Remove the ABEL state keywords and replace the ABEL then keywords by the next state name. The second **always** statement updates the state variable to the next state at each clock tick. The Verilog **case** statement has the **default** statement to capture all unspecified state transitions. Therefore, all the ABEL GOTO statements in a state machine can be replaced by a single Verilog default statement. Add ';' to the end of every state machine statement. The third **always** statement defines the output signals based on the current state included in the sensitivity list. If the state variables are used as outputs directly, the third **always** statement Listing 2. More ABEL to Verilog conversion examples for a 4-story elevator controller can be found from the author's website at www.egr.up.edu/contribu/lu.

It will take just a little time to convert an ABEL program into Verilog syntax. Note that the ABLE variable names are kept as uppercase in the converted Verilog program to illustrate the conversion process. Normally, variable names in Verilog design modules are defined in lowercase letters.

From Simple Modules To A Complicated Digital System

A complicated digital system can be efficiently designed by dividing a complex system into simpler subsystems, the so-called top-down design methodology. Each subsystem performs a specific task much like the subroutine does in a high-level language program. After each subsystem is developed and tested, they can be connected together in a bottom-up order to form the system. ABEL is very efficient for developing such subsystem modules and Verilog is extremely efficient for connecting modules into a system. By converting the debugged ABEL subsystem modules into Verilog syntax, a top Verilog module can easily integrate all the modules into a system. The top module is mainly to instantiate the subsystem modules in a straightforward net-list format. For example, the following top module, controller.v, instantiates seven modules designed for a 4-story elevator control system. These eight Verilog modules are then synthesized into a complete system by a synthesis tool such as Xilinx ISE and implemented by an XC9572 CPLD. The complete ABEL and Verilog programs can be downloaded from the author's web site at www.egr.up.edu/contribu/lu.

Conclusion

This paper presents a very efficient method to quickly implement a complicated digital system using Verilog. Although this method uses only part of the Verilog language constructs, however, it provides the designers an insight on the top-down design and bottom-up implementation methodology. In Verilog, combinational circuits can also be represented by behavioral modeling (always statements) to create a more abstract description of the circuit. Therefore, a digital system can be designed in a style much like that of high-level programming languages. Readers are encouraged to take a Verilog-related design course to fully utilize the capability of this popular hardware description language.

References

1. John F. Wakerly, "ADigital Design Principles and Practices", Third Edition Updated, Prentice-Hall, 2001.

2. Dave Van d. Bout, "The Practical Xilinx Designer Lab Book", Prentice-Hall, 1998.

3. Michael D. Ciletti, "Modeling, Synthesis, and Rapid Prototyping with the VERILOG HDL", Prentice-Hall, 1999.

4. Samir Palnitkar, "Verilog HDL", 2nd Edition, Prentice-Hall PTR, 2003.

Biographical Information

Wayne Lu received the phD degree in Electrical Engineering from the University of Oklahoma in 1989. He has been with the University of Portland since 1988 and currently is an Associate Professor of Electrical Engineering. Dr. Lu's primary research interests are ASIC design & prototyping, embedded systems, and computer vision.

Listing 1 doors.abl

```
module doors
title 'Door opening and closing control for a 4-story elevator'
U4 device 'p20v8';

"Input pins
CLK                                     pin 1;
DIRA, DIRB, DOOR                        pin 6,5,2;
MOTIONU, MOTIOND                        pin 3,4;
DOC, CNT2, CNT1, CNT0   pin 11,7,8,9;

"Output pins
LED1, LED2, LED3, LED4   pin 19,20,21,22 istype 'reg';
ARR0, ARR1, ARR2, ARR3 pin 15,16,17,18 istype 'com';

"Intermedia equations
CLOSING = DOC & !DOOR & !MOTIONU & !MOTIOND;
OPENING = DOC &  DOOR & !MOTIONU & !MOTIOND;

"State definitions
CLOSE = ^b1111;  OPEN1 = ^b0111;  OPEN2 = ^b0011;
OPEN3 = ^b0001;  OPEN   = ^b0000;

"Intermediate equations
UP01 = !DOC & !DOOR &  MOTIONU & !MOTIOND & !DIRB & !DIRA;
UP12 = !DOC & !DOOR &  MOTIONU & !MOTIOND & !DIRB &  DIRA;
UP23 = !DOC & !DOOR &  MOTIONU & !MOTIOND &  DIRB & !DIRA;
DN10 = !DOC & !DOOR & !MOTIONU &  MOTIOND & !DIRB &  DIRA;
DN21 = !DOC & !DOOR & !MOTIONU &  MOTIOND &  DIRB & !DIRA;
DN32 = !DOC & !DOOR & !MOTIONU &  MOTIOND &  DIRB &  DIRA;

"Doors movement and status display, LED1 at the bottom, LED4 at the top
CONTROL = [LED1, LED2, LED3, LED4];

equations
ARR0 =  DN10 & CNT2 & CNT1 & CNT0;
ARR1 = (DN21 # UP01) & CNT2 & CNT1 & CNT0;
ARR2 = (DN32 # UP12) & CNT2 & CNT1 & CNT0;
ARR3 =  UP23 & CNT2 & CNT1 & CNT0;
CONTROL.C = CLK;

"Doors control state machine description
state_diagram CONTROL
        state CLOSE:
            if (OPENING) then OPEN1
            else CLOSE;
        state OPEN1:
            if (OPENING) then OPEN2
            else CLOSE
        state OPEN2:
            if (OPENING) then OPEN3
            else OPEN1
        state OPEN3:
            if (OPENING) then OPEN
            else OPEN2
        state OPEN:
            if (CLOSING) then OPEN3
            else OPEN;
end doors
```

Listing 2 doors.v

```
module doors(CLK,DIRA,DIRB,DOOR,MOTIONU,MOTIOND,DOC,CNT2,CNT1,CNT0,
       LED1,LED2,LED3,LED4,ARR0,ARR1,ARR2,ARR3);

input  CLK,DIRA,DIRB,DOOR,MOTIONU,MOTIOND,DOC,CNT2,CNT1,CNT0;
output LED1,LED2,LED3,LED4,ARR0,ARR1,ARR2,ARR3;
wire   CLOSING,OPENING,OPENDOOR,CLOSEDOO;
wire   UP01,UP12,UP23,DN10,DN21,DN32;
reg[3:0] CONTROL,NEXTCONTROL;

//Intermediate equations
assign CLOSING = DOC & ~DOOR & ~MOTIONU & ~MOTIOND;
assign OPENING = DOC &  DOOR & ~MOTIONU & ~MOTIOND;
assign OPENDOOR = ~DOC &  DOOR  & ~MOTIONU & ~MOTIOND;
  assign CLOSEDOOR = ~DOOR & ~MOTIONU & ~MOTIOND;

  //State definitions
  parameter CLOSE = 4'b1111;
  parameter OPEN1 = 4'b0111;
  parameter OPEN2 = 4'b0011;
  parameter OPEN3 = 4'b0001;
  parameter OPEN  = 4'b0000;

  //Intermediate equations
  assign UP01 = ~DOC & ~DOOR &  MOTIONU & ~MOTIOND & ~DIRB & ~DIRA;
  assign UP12 = ~DOC & ~DOOR &  MOTIONU & ~MOTIOND & ~DIRB &  DIRA;
  assign UP23 = ~DOC & ~DOOR &  MOTIONU & ~MOTIOND &  DIRB & ~DIRA;
  assign DN10 = ~DOC & ~DOOR & ~MOTIONU &  MOTIOND & ~DIRB &  DIRA;
  assign DN21 = ~DOC & ~DOOR & ~MOTIONU &  MOTIOND &  DIRB & ~DIRA;
  assign DN32 = ~DOC & ~DOOR & ~MOTIONU &  MOTIOND &  DIRB &  DIRA;

  //CONTROL = [LED1, LED2, LED3, LED4] for door movement and status;
  assign LED1 =  CONTROL[3], LED2=CONTROL[2], LED3=CONTROL[1], LED4=CONTROL[0];
  assign ARR0 =  DN10 & CNT2 & CNT1 & CNT0;
  assign ARR1 = (DN21 | UP01) & CNT2 & CNT1 & CNT0;
  assign ARR2 = (DN32 | UP12) & CNT2 & CNT1 & CNT0;
  assign ARR3 =  UP23 & CNT2 & CNT1 & CNT0;

  always @(posedge CLK)
         CONTROL = NEXTCONTROL;

  always @(CONTROL or OPENING or CLOSING)
    begin
     case(CONTROL)
          CLOSE:
                 if (OPENING) NEXTCONTROL = OPEN1;
                 else NEXTCONTROL = CLOSE;
          OPEN1:
                 if (OPENING) NEXTCONTROL = OPEN2;
                 else  NEXTCONTROL = CLOSE;
         OPEN2:
                 if (OPENING) NEXTCONTROL = OPEN3;
                 else  NEXTCONTROL = OPEN1;
```

```
            OPEN3:
                        if (OPENING) NEXTCONTROL = OPEN;
                        else  NEXTCONTROL = OPEN2;
            OPEN:
                        if (CLOSING) NEXTCONTROL = OPEN3;
                        else NEXTCONTROL = OPEN;
        default: NEXTCONTROL = CLOSE;
       endcase
  end
  endmodule
```

Listing 3 controller.v

```
module controller(CLK,OCALL,CCALL,CALL0,CALL1,CALL2,CALL3,ECALL0,ECALL1,ECALL2,ECALL3,RESET,
                W0,W1,W2,W3,E0,E1,E2,E3, DLED1,DLED2,DLED3,DLED4,
                LED0, LED1, LED2, LED3,LED4, LED5, LED6, LED7,
                SEGA,SEGB,SEGC,SEGD,SEGE,SEGF,SEGG,UPARROW,DNARROW,MIDARROW);

input  CLK,OCALL,CCALL,CALL0,CALL1,CALL2,CALL3,ECALL0,ECALL1,ECALL2,ECALL3,RESET;
output W0,W1,W2,W3,E0,E1,E2,E3, DLED1,DLED2,DLED3,DLED4;
output LED0, LED1, LED2, LED3,LED4, LED5, LED6, LED7;
output SEGA,SEGB,SEGC,SEGD,SEGE,SEGF,SEGG, UPARROW,DNARROW,MIDARROW;

wire   W0_BAR,W1_BAR,W2_BAR,W3_BAR,E0_BAR,E1_BAR,E2_BAR,E3_BAR;
wire   L0CALL,L1CALL,L2CALL,L3CALL,ARR0,ARR1,ARR2,ARR3;
wire   CT3,DOOR,DIRA,DIRB,MOTIONU,MOTIOND,DOC,CNT2,CNT1,CNT0;

//Instantiate modules and their interconnections
wallbtns  U1(OCALL,CALL0,CALL1,CALL2,CALL3,DOOR,DIRA,DIRB,
                  W0,W0_BAR,W1,W1_BAR,W2,W3_BAR,W3,W4_BAR);

ebuttons U2(OCALL,DOOR,DIRA,DIRB,ECALL0,ECALL1,ECALL2,ECALL3,
                  E0, E0_BAR, E1, E1_BAR,E2, E2_BAR,E3, E3_BAR);

mainctl U3(CLK,L0CALL,L1CALL,L2CALL,L3CALL,ARR0,ARR1,ARR2,ARR3,CCALL,OCALL,RESET,
                 CT3,DOOR,DIRA,DIRB,MOTIONU,MOTIOND,DOC,CNT2,CNT1,CNT0);

doors U4(CLK,DIRA,DIRB,DOOR,MOTIONU,MOTIOND,DOC,CNT2,CNT1,CNT0,
              DLED1,DLED2,DLED3,DLED4,ARR0,ARR1,ARR2,ARR3);

etravel U5(MOTIONU, MOTIOND,CNT2, CNT1, CNT0,
               LED0, LED1, LED2, LED3,LED4, LED5, LED6, LED7);

display U6(DIRA,DIRB,MOTIONU,MOTIOND,CNT0,
                SEGA,SEGB,SEGC,SEGD,SEGE,SEGF,SEGG);

arrow U7(W0,W1,W2,W3,E0,E1,E2,E3,MOTIONU,MOTIOND,
              UPARROW,DNARROW,MIDARROW,L0CALL,L1CALL,L2CALL,L3CALL);

Endmodule
```