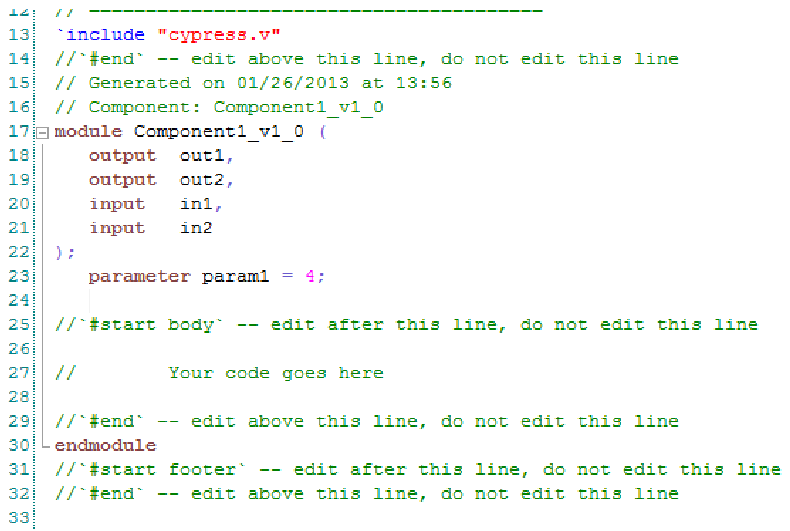
Just Enough Verilog for PSoC®

When creating a custom UDB-based component in PSoC® Creator™, if you start with the symbol wizard[1] and then generate the Verilog corresponding to that symbol, the code looks like:

Figure 1. Verilog Shell Generated by PSoC Creator



```
// ----------------------------------------
`include "cypress.v"
//`#end` -- edit above this line, do not edit this line
// Generated on 01/26/2013 at 13:56
// Component: Component1_v1_0
module Component1_v1_0 (
    output  out1,
    output  out2,
    input   in1,
    input   in2
);
    parameter param1 = 4;

//`#start body` -- edit after this line, do not edit this line

//        Your code goes here

//`#end` -- edit above this line, do not edit this line
endmodule
//`#start footer` -- edit after this line, do not edit this line
//`#end` -- edit above this line, do not edit this line
```

[1]*This is discussed in another KB Article titled 'Creating a Verilog-Based Component' available at www.cypress.com.*

Using this Verilog shell as a starting point, this document discusses important aspects of Verilog needed to understand and build meaningful designs in the PSoC UDBs. It is meant to be a handy supplement to the Warp Verilog Reference Guide and the Verilog textbook of your choice.

Only the elements of Verilog supported by the Warp synthesizer tool are discussed in this document. See the Warp Verilog Reference Guide in PSoC Creator **Help**>**Documentation** for more information.

Contents

Introduction

Verilog is a Hardware Description Language (HDL). To appreciate what this means, consider the 8-bit combinatorial multiplier in Code 1. The multiplier takes as input two 8-bit numbers A and B, multiplies them, and outputs the 16-bit result (Mult).

Code 1. Verilog is not C

```
module Multiplier (output [15:0] Mult,
                   input  [7:0]  A,
                   input  [7:0]  B    );
  assign Mult = A * B;
endmodule
```

Note Consumes 144 macrocells (75%) and 196 product terms (51%) in a 24-UDB PSoC device.

Code written in Verilog is synthesized (maps) to the UDB PLDs unless the UDB Datapaths/Status/Control blocks are explicitly instantiated in the Verilog. It does not run on the CPU.

C and Verilog

Now that you have been introduced to the fundamental difference between C (runs on a CPU) and Verilog (maps to logic), note that there are several similarities between the two:

- Language structure (file includes, variable declaration, code blocks, comments, semicolons to terminate statement, and more)
- If statement, case statement, bitwise and logical operators, and more
- while loop, for loop (these are not generally used because they are not synthesizable, and hence not discussed)

Table 1 shows some of the parallels between C and Verilog:

Table 1. Parallels between C and Verilog

Concept	C	Verilog
Operators (1)	Arithmetic Operators: *, +, -, /, % Shift Operators: <<, >> Relational Operators: <, >, <=, >= Equality Operators: ==, != Logical Operators: !, &&, || Conditional Operator: ?:	Same as C
Operators (2)	Has the boldfaced operators on the right	Bitwise Operators: **~**, **&**, **|**, **^**, ^~, ~^ Reduction Operators: &, |, ^, ^~, ~^, ~&, ~| Event or: or Concatenation: {}, {{}} These are explained with examples in section 2.4 of the *Warp Verilog Reference Guide*.

Table 1. Parallels between C and Verilog (continued)

Concept	C	Verilog

Comments	`// slash slash comment`
	`/* slash star comment */`
		Same as C

Compiler directives	`#define, #include, #ifdef,`
	`#ifndef, #else, #endif, #undef`
		`` `define, `include, `ifdef, ``
		`` `ifndef, `else, `endif, `undef, ``
		`` `elseif ``

Referring to a #defined constant	`#define CONST 5`
	`...`
	`a = CONST;`
		`` `define CONST 5 ``
		…
		`` assign a = `CONST; ``

Declaring vectors (arrays)	Declare 5-member array as:
	`uint8 var[5];`
		Declare 5-bit vector as:
		`reg  [4:0] a;`
		or:
		`wire [4:0] a;`

Block Delimiting	Braces { }	**begin** and **end** keywords

if-else	`if(condition_1)`
	`{`
	`  // do something`
	`}`
	`else`
	`{`
	`  // do something else`
	`}`
		`if(condition_1)`
		`begin`
		`  // do something`
		`end`
		`else`
		`begin`
		`  // do something else`
		`end`

Case (switch) statement	`switch(a)`
	`{`
	`  case 31:`
	`  //statements here`
	`  break;`
	
	`  case 0;`
	`  //statements here`
	`  break;`
	
	`  // other cases`
	
	`  default:`
	`  // statements here`
	`  break;`
	`}`
		`case(a)      // a is a 5-bit vector`
		`  10'd31:`
		`  begin`
		`  //statements here`
		`  end`
		
		`  10'd0:`
		`  begin`
		`  //statements here`
		`  end`
		
		`  //other cases`
		
		`  //good practice to have a`
		`default`
		`  default:`
		`  begin`
		`  //statements here`
		`  end`
		`endcase`

Modules

The module is the basic building block in Verilog. It is the metaphorical black box with inputs and outputs, analogous to a function in C.

In all cases where you describe modules, you are providing the template for the behavior of the module. This module (or function) can be later instantiated (or called) in other top level modules.

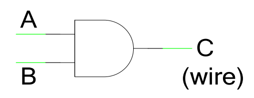
The module name for the Component should be the same as the file it is saved in. For example, the file name of the example in Figure 1 is Component1_v1_0.v

Data Types: Wire vs Reg

Keeping in mind that Verilog describes hardware, *signals* in a Verilog module are either of type 'wire' or type 'reg'.

Wire

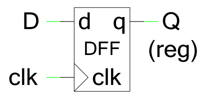
- Combinatorial signal – continuously driven, literally like a wire
- Assigned outside an **always** block with **assign** statement or inside **always** block (see the Assignments section)



an

Reg

- Synchronous signal – changes state only on a trigger event
- Can be assigned a value only inside an **always** block



There exists a third data type – a parameter, discussed in the Parameter section.

Registering Outputs

Signals listed in the module terminal list by default are of type wire. If the outputs of the module you are defining are synchronous (as they often are), you must change the terminal list shown in Figure 1 to be:

Code 2. Module Terminal List

```
module Component1_v1_0 (
    output reg out1,
    output reg out2,
    input   in1,
    input   in2
);
```

This is the only code that you have to write outside of the #start and #end comments in the Verilog file – so if you regenerate the Verilog, it has to be re-entered.

Declarations

Declare all other signals (besides for the ones in the module terminal list) after the #start body comment. This is similar to declaring variables in C.

Code 3. Examples of type declarations

```
//`#start body` -- edit after this line, do not edit this line

wire sig1, sig2;
wire [3:0] bus1;
reg cmp;
reg [2:0] addressSelect;
```

These signals are a single bit wide if the width is not specified, or buses (vectors) if a width is specified – as Code 3 shows.

Constants

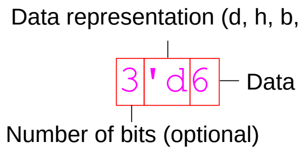
One source of confusion for first-time Verilog readers are statements similar to those in Code 4.

Code 4. Example of Constant Syntax

```
wire [3:0] var1 = 4'd6;
wire var2 = 1'b1;
```

This is similar to declaring and initializing variables in C. Figure 2 explains what the constant means:

Figure 2. Explanation of Sized Constants



- <Number of bits> (optional)
    - Number of bits (not digits) used to represent the data
- '<Data Representation>
    - The base of the data field. Can be decimal (d), binary (b), hexadecimal (h), octal (o); is case insensitive
- <Data>
  - A decimal base number is composed of a sequence of 0 through 9 digits.
  - A binary base number is composed of a sequence of x (don't care), z (high impedance), 0 and 1.
  - A hexadecimal base number is composed of a sequence of x, z, 0 through 9 digits and A through F characters.

Always Construct

This statement is used to model a block of activity repeated on a set of conditions. This set of conditions is called the sensitivity list. In Warp, an **always** statement must have a sensitivity list.

Code 5. Examples of **always** Construct

```
// 1. Asynchronous trigger
always @ (x or y)
begin
// If x or y change, execute statements
end

// 2. Synchronous trigger
always @ (posedge clock)
begin
// At every rising edge of 'clock', execute statements
end
```

Sensitivity List

The sensitivity list determines when the `always` block is executed. With reference to Code 5, it is the expression after the @, in parentheses.

1. The first `always` block has an asynchronous trigger – the `always` block is executed any time x or y change state.
2. The second `always` block has an synchronous trigger – the `always` block is executed on each rising edge of the clock.

The sensitivity list can contain only asynchronous triggers or only synchronous triggers, but not both. For example, the sensitivity list cannot contain `always @ (x or posedge clock)`.

There also exists a `negedge` keyword. However, because of the architecture of the PSoC, only `posedge` should be used. Timing and synchronization failures are likely if `negedge` is used. If it is essential that something occur on the positive and negative edge of a clock, use the rising edge of a clock twice the frequency to trigger the circuitry.

Assignments

Continuous Assignment

When modeling combinatorial logic outside an `always` statement, assignments to wires are made using the `assign` statement:

Code 6. Example of Continuous Assignment

```
// z and p were declared as wires
assign z = x & y;   // The right hand side of this statement can be a
assign p = 1'b1;    // wire, reg, a constant, or a combination of the 3
```

Procedural Assignment

When modeling combinatorial or sequential logic inside an `always` block, assignments are of two types:

a. Blocking assignments – using the "=" operator; are executed one after the other (*in a serial fashion*)
b. Non-blocking assignments – using the "<=" (looks like an arrow) operator, are executed *in parallel*.

To understand this concept a little better, consider the examples in Code 7.

Code 7. Examples of Blocking and Non-Blocking Assignments

```
// initial values: a=1, b=2, c=3; all three are registers
// clk is a clock in the system

// blocking assignment
always @ (posedge clk)
begin
  a = b ;
  c = a ;
end /* a=2, b=2, c=2 */

// non-blocking assignment
always @ (posedge clk)
begin
  a <= b ;
  c <= a ;
end /* a=2, b=2, c=1 */
```

Keep in mind the following rules:

- When modeling sequential logic such as state machines, use non-blocking assignments.
- When modeling combinatorial logic with an `always` block, use blocking assignments. For example, if you try to model `y=(a&b)|(c&d)` with Code 8, y is assigned a value according to the old contents of tmp1 and tmp2, not from the current pass of the always block.

Code 8. Modeling Combinatorial Logic - Non-Blocking Assignments Gives Unexpected Results

```
wire y;
wire a, b, c, d;
reg y, tmp1, tmp2;
always @(a or b or c or d)
begin
  tmp1 <= a & b;
  tmp2 <= c & d;
  y <= tmp1 | tmp2;
end
```

- When modeling both sequential and combinatorial logic within the same `always` block, use non-blocking assignments.*
- Do not mix blocking and non-blocking assignments in the same `always` block (See Code 9).

Code 9. What Not to Do – Mixing Blocking and Non-Blocking Assignments in the Same `always` Block

```
//  illegal - mixing blocking an non-blocking assignments
always @ (posedge clk)
begin
  a <= b + c ; // non-blocking assignment to a
  b = d ;      // blocking assignment to b
end
```

- Do not make assignments to the same variable from more than one `always` block. For more information, refer to Cliff Cummings' paper: *Nonblocking Assignments in Verilog Synthesis, Coding Styles That Kill!*

Parameters

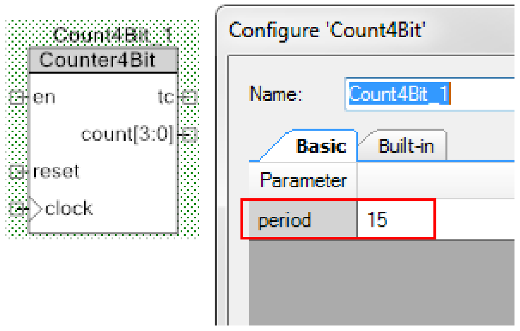
Parameters are constants which cannot be modified during run time. Based on their scope, constants in a design can either be:

a. Design wide:
- Use the `` `define `` syntax
- Examples could be to define design-wide constants like TRUE and FALSE

b. Local to the module, but passed to it:
- Use the `parameter` keyword - see Figure 3
- The parameter value is assigned at *compile-time*, that is, it does change at run-time.
- See Figure 3. Note that the value in code (on the right) is overwritten at compile-time based on the customizer values.
- Used as a method to configure a Component
- Parameters can be passed to the Component either through the Component customizer (as shown in Figure 3) or in Verilog – this is discussed in the Instantiation section.

The example shown in Figure 3 is a custom Component taken from [AN82250](AN82250) - *PSoC® 3 and PSoC 5LP Implementing Programmable Logic Designs with Verilog*; 'period' is the configurable counter period. The full Verilog code for the counter is provided in Appendix A.

Figure 3. Example of Component Parameters



Name: Count4Bit_1

Parameter	
period	15

```
module Count4Bit_v1_20 (
    output reg [3:0] count,
    output reg tc,
    input   clock,
    input   en,
    input   reset
);

    /* period overwritten by the customizer input */
    parameter period = 2;
```

c. Local to the module:
- Use the `localparam` keyword
- Ensures that the parameter cannot be modified by or interfere with other modules.
- Example could be to define sensible names for various states of a state machine.

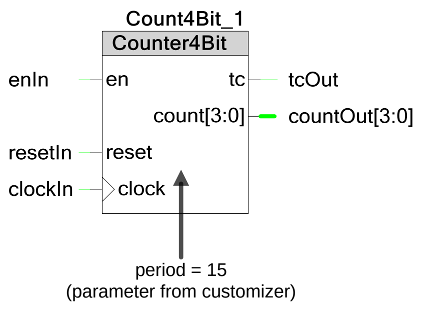
Code 10. Example of localparams

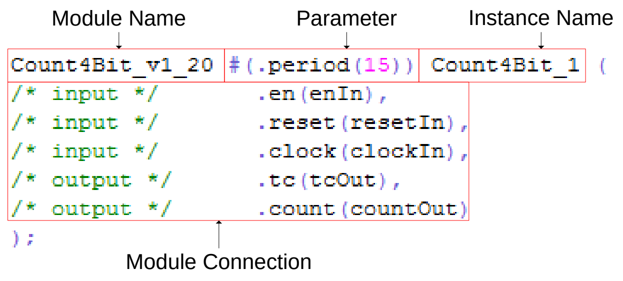
```
localparam START   = 3'd0;        /* detect, restart = 0, 1 */
localparam STATE_1 = 3'd1;        /* detect, restart = 0, 0 */
localparam STATE_2 = 3'd2;        /* detect, restart = 0, 0 */
localparam STATE_3 = 3'd3;        /* detect, restart = 0, 0 */
localparam STATE_4 = 3'd4;        /* detect, restart = 0, 0 */
localparam DETECT  = 3'd5;        /* detect, restart = 1, 0 */
```

Instantiation

You can use another prewritten Component or module directly in your Verilog code by instantiating it. Placing, connecting, and configuring a UDB Component on a schematic is equivalent to instantiating a UDB Component/Module in Verilog Code. For example, the code snippet and schematic in Figure 4 are equivalent.

Figure 4. Instantiation and Schematic - Similarities

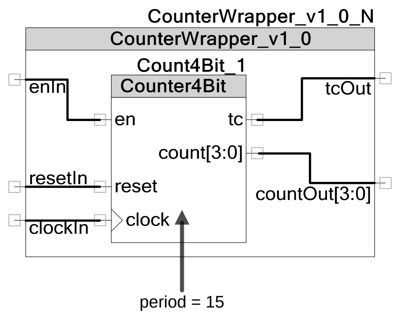


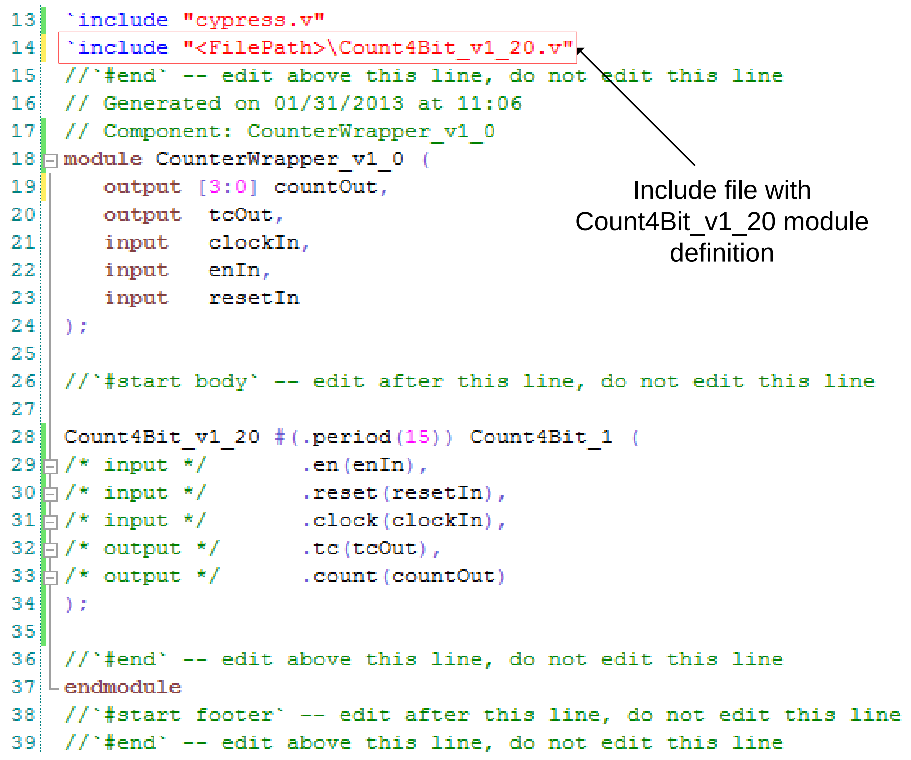


```
Count4Bit_v1_20 #(.period(15)) Count4Bit_1 (
/* input */          .en(enIn),
/* input */          .reset(resetIn),
/* input */          .clock(clockIn),
/* output */         .tc(tcOut),
/* output */         .count(countOut)
);
```

If you were to create a wrapper Component for the 4-bit counter, the Verilog code would look similar to the code in Figure 5. Note that to instantiate another Component or pre-written module in your Verilog code, you have to include the corresponding Verilog file (line 14 in Figure 5) and then instantiate it (lines 28-33 in Figure 5).

Figure 5. Counter Wrapper Example for Instantiation – Verilog Code with Equivalent Component Symbol





```
13 `include "cypress.v"
14 `include "<FilePath>\Count4Bit_v1_20.v"
15 //`#end` -- edit above this line, do not edit this line
16 // Generated on 01/31/2013 at 11:06
17 // Component: CounterWrapper_v1_0
18 module CounterWrapper_v1_0 (
19     output [3:0] countOut,
20     output  tcOut,
21     input   clockIn,
22     input   enIn,
23     input   resetIn
24 );
25 
26 //`#start body` -- edit after this line, do not edit this line
27 
28 Count4Bit_v1_20 #(.period(15)) Count4Bit_1 (
29 /* input */          .en(enIn),
30 /* input */          .reset(resetIn),
31 /* input */          .clock(clockIn),
32 /* output */         .tc(tcOut),
33 /* output */         .count(countOut)
34 );
35 
36 //`#end` -- edit above this line, do not edit this line
37 endmodule
38 //`#start footer` -- edit after this line, do not edit this line
39 //`#end` -- edit above this line, do not edit this line
```

<FilePath> in Figure 5 is the location on the disk which contains Count4Bit_v1_20.v.

Cypress modules such as the Datapath, Control, Status, UDB Clock Enable blocks are automatically included by including `"cypress.v"` at the top of your Verilog code.

Verilog code maps to the Datapath, Control/Status, UDB clock enable blocks only if these are explicitly instantiated. All the rest of the Verilog code maps to the PLDs.

The instantiation syntax can be broken out as:
<Module Name> #(Parameters) <Instance Name> (Module Connection);

- **Module Name**
  Is the name of the top level module in the included file.

- **Parameters** (see Parameters section)
  Parameters are passed to the instantiated module by their name in the module definition – for example SignalWidth is a parameter for the Debouncer (Figure 3). The syntax to do this is:
  #(.<param1>(value1), .<param2>(value2), ….)

  In Figure 5, period (param1) is passed a value of 15 (value1).

  If a parameter value is not explicitly passed, it remains at the value it was initialized to in the module definition.

- **Instance Name**
  Is the name you choose to give the instantiated Component (Count4Bit_1 in above example)

- **Module Connection**
  Describes the connection between the signals listed in the module instantiation statement and the ports in the module definition. The syntax is similar to that for parameters:
  (.port1(signal1), .port2(signal2),…);
  Where the port1, port2, and so on are the names in the module definition, and signal1, signal2, and so on are the names of signal in the top level module.

  In Figure 5, en, reset, clock, tc, count (port1, port2, port3, port4, port5) are connected to enIn, resetIn, clockIn, tcOut, countOut (signal1, signal2, signal3, signal4, signal5). Note that input ports must be connected, while output ports may be left unconnected.

A Datapath instance (Code 11) is only an instantiation. The blue text (parameters) is automatically generated based on the Datapath Configuration Tool settings. The rest of the statements in the list are module connections, and allow you to connect the signals you want to interface with the Datapath.

Code 11. Instance of a Datapath (taken from the B_PWM Component)

Module Name

```
cy_psoc3_dp8 #(.cy_dpconfig_a (
```

Parameters

```
{
    `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
    `CS_SHFT_OP_PASS, `CS_A0_SRC___D0, `CS_A1_SRC_NONE,
    `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
    `CS_CMP_SEL_CFGA, /* CS_REG0 Comment:Preload Period (A0 <= D0) */
    `CS_ALU_OP__DEC, `CS_SRCA_A0, `CS_SRCB_D0,
    `CS_SHFT_OP_PASS, `CS_A0_SRC__ALU, `CS_A1_SRC_NONE,
    `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
    `CS_CMP_SEL_CFGA, /* CS_REG1 Comment:Dec A0  ( A0 <= A0 - 1 ) */
    `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
    `CS_SHFT_OP_PASS, `CS_A0_SRC__ALU, `CS_A1_SRC_NONE,
    `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
    `CS_CMP_SEL_CFGA, /* CS_REG2 Comment:Idle */
    `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
    `CS_SHFT_OP_PASS, `CS_A0_SRC__ALU, `CS_A1_SRC_NONE,
    `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
    `CS_CMP_SEL_CFGA, /* CS_REG3 Comment:Idle */
    `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
    `CS_SHFT_OP_PASS, `CS_A0_SRC__ALU, `CS_A1_SRC_NONE,
    `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
    `CS_CMP_SEL_CFGA, /* CS_REG4 Comment:Idle */
    `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
    `CS_SHFT_OP_PASS, `CS_A0_SRC__ALU, `CS_A1_SRC_NONE,
    `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
    `CS_CMP_SEL_CFGA, /* CS_REG5 Comment:Idle */
    `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
    `CS_SHFT_OP_PASS, `CS_A0_SRC__ALU, `CS_A1_SRC_NONE,
    `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
    `CS_CMP_SEL_CFGA, /* CS_REG6 Comment:Idle */
    `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
    `CS_SHFT_OP_PASS, `CS_A0_SRC__ALU, `CS_A1_SRC_NONE,
    `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
    `CS_CMP_SEL_CFGA, /* CS_REG7 Comment:Idle  */
      8'hFF, 8'h00, /* SC_REG4       Comment: */
      8'hFF, 8'hFF, /* SC_REG5       Comment: */
    `SC_CMPB_A0_D1, `SC_CMPA_A0_D1, `SC_CI_B_ARITH,
    `SC_CI_A_ARITH, `SC_C1_MASK_DSBL, `SC_C0_MASK_DSBL,
    `SC_A_MASK_DSBL, `SC_DEF_SI_0, `SC_SI_B_DEFSI,
    `SC_SI_A_DEFSI, /* SC_REG6 Comment: */
    `SC_A0_SRC_ACC, `SC_SHIFT_SL, 1'b0,
    1'b0, `SC_FIFO1_BUS, `SC_FIFO0_BUS,
    `SC_MSB_DSBL, `SC_MSB_BIT0, `SC_MSB_NOCHN,
    `SC_FB_NOCHN, `SC_CMP1_NOCHN,
    `SC_CMP0_NOCHN, /* SC_REG7 Comment: */
     10'h0, `SC_FIFO_CLK__DP,`SC_FIFO_CAP_AX,
    `SC_FIFO__EDGE,`SC_FIFO__SYNC,`SC_EXTCRC_DSBL,
    `SC_WRK16CAT_DSBL /* SC_REG8 Comment */
}))
```

Instance Name

```
killmodecounterdp (
/*  input                  */  .clk(ClockOutFromEnBlock),
/*  input    [02:00]       */  .cs_addr({2'b0,km_run}),
/*  input                  */  .route_si(1'b0),
/*  input                  */  .route_ci(1'b0),
/*  input                  */  .f0_load(1'b0),
/*  input                  */  .f1_load(1'b0),
/*  input                  */  .d0_load(1'b0),
/*  input                  */  .d1_load(1'b0),
/*  output                 */  .ce0(),
/*  output                 */  .cl0(),
/*  output                 */  .z0(km_tc),              /* Terminal Count (A0 == 0)  */
/*  output                 */  .ff0(),
/*  output                 */  .ce1(),
/*  output                 */  .cl1(),
/*  output                 */  .z1(),
/*  output                 */  .ff1(),
/*  output                 */  .ov_msb(),
/*  output                 */  .co_msb(),
/*  output                 */  .cmsb(),
/*  output                 */  .so(),
/*  output                 */  .f0_bus_stat(),
/*  output                 */  .f0_blk_stat(),
/*  output                 */  .f1_bus_stat(),
/*  output                 */  .f1_blk_stat()
);
```

Only the clock and the CFGRAM address are connected to varying signals

Module Connection

Guidelines

- All if-then-else statements should have a final else clause.
- Case statements should be fully defined, that is, circuit behavior for all possibilities of inputs must be defined. Additionally, it is good practice to add the `default` statement.

If an if-then-else statement does not have the final else clause, a latch will be created. Similarly, case statements that are not fully defined could also produce latches. Avoid latches in PSoC 3 and PSoC 5LP designs. The results can be unexpected and synchronization to the rest of the design becomes an issue.

- Code all intentional priority encoders using if-else-if statements.
- Do not use non-constants as indices for any array (Code 12). The logic produced is very large.

Code 12. Do Not Use Non-Constants as Array Indices

```
always @(posedge clock)
begin
  rx_cnt_minus1 <= rx_cnt - 1;   // rx_cnt and rx_cnt_minus1 are 4-bit vectors
  if (rx_cnt > 0 && rx_cnt < 9)
  begin
    rx_reg[rx_cnt_minus1] <= rx_d2; //rx_reg is a vector, and rx_d2 is single-bit
  end
end
```

- Use the Datapath for arithmetic operations on variables. (See Table 2)

Table 2. Comparison of PLD and Datapath Resource Usage for Arithmetic Operations

Function	**Resource consumption in PLDs only**		**Resource consumption in datapaths only**	
	PLDs	**% Used**	**Datapath**	**% Used**
ADD8	5	10.4%	1	4.2%
SUB8	5	10.4%	1	4.2%
CMP8	3	6.3%	1	4.2%
SHIFT8	3	6.3%	1	4.2%

- Always prefer synchronous reset/preset over asynchronous. In any situation where you have a choice, use synchronous signals as a good design practice.
- Use synchronous reset/preset if the affected component has a continuously running clock.
- Limit the use of reset and preset to a choice of either (not both) for any single flip-flop.

Asynchronous presets and resets can be applied at any time. They can introduce timing problems into a circuit. Reserve Asynchronous presets and resets for purposes such as power-up conditioning or in the case where the Component clock is not running and the Component needs to be reset. This might be the case for certain state machines. PSoC 3 and PSoC 5LP architecture has a single signal that you can use as either a reset or a preset.

- All clocks passed to a Component should be phase aligned with BUS_CLOCK. This is ensured by adding a Sync Component.
- Do not modify a clock input with combinatorial logic (such as gating a clock).

Since the CPU and DMA may be in a different clock domain from the Component clock, all clocks used in a design must be synchronized with BUS_CLOCK to prevent problems caused by metastability or clock domain crossing. For a detailed discussion of timing-related issues and how to solve them, see *AN81623 – Digital Design Best Practices*.

Generate Construct

A `generate` block is used to conditionally generate code within a module based on a parameter (constant) passed to the module at compile-time. Read section 2.9.6 of the Warp Verilog Reference Guide for more information. Appendix B contains an example of the `generate` statement.

Final Words

The appendices contain examples to illustrate the concepts taught in this guide. However, these examples are not exhaustive. Since state machines are particularly important, Cypress recommends reading the Verilog code for the SeqDetect Component from AN82250. *AN82156 - PSoC 3 and PSoC 5LP® - Designing PSoC Creator Components with UDB Datapaths* contains good examples on Datapath-based Component development. For Verilog syntax-related needs, use the Warp Verilog Reference Guide or a Verilog textbook of choice. For more guidelines and Verilog best practices, see section 11.6 of the *Component Author Guide* in PSoC Creator *Help>Documentation*. Finally, there is nothing like learning by doing. So go ahead and start building your own Verilog designs in PSoC!

Appendix A: 4-bit Counter Custom Component

```verilog
//`#start header` -- edit after this line, do not edit this line
// ============================================================
// Count4Bit_v1_20.v (from AN82250)
//
// Parameter:
// uint8 period - 4-bit period value
//
// Inputs:
// en    - Hardware enable. If low, outputs are still active but the component does not change states.
// reset - Active high, synchronous reset
// clock - Operating frequency of component
//
// Outputs:
// tc    - Synchronous terminal count. Goes high for 1 clock cycle when count value equals period value.
// count - 4-bit current count value
// ============================================================
`include "cypress.v"

`ifdef Count4Bit_v1_20_V_ALREADY_INCLUDED
`else
`define Count4Bit_v1_20_V_ALREADY_INCLUDED
//`#end` -- edit above this line, do not edit this line

// Component: Count4Bit_v1_20
module Count4Bit_v1_20 (
  output reg [3:0] count,
  output reg tc,
  input   clock,
  input   en,
  input   reset
);
  parameter period = 0;

  //`#start body` -- edit after this line, do not edit this line
  always @ (posedge clock)
  begin
    if(reset)                        /* Initialize the counter */
    begin
      count <= 4'b0000;
      tc <= 1'b0;
    end
    else                             /* counter is not in reset */
    begin
      if(en)                         /* start counting */
      begin
        if(count == period)          /* reached terminal count */
        begin
          tc <= 1'b1;
          count <= 4'b0000;
        end
        else                         /* count is less than the period, so count up */
        begin
          count <= count + 1;
          tc <= 1'b0;
        end
      end
      else                           /* enable is 0 - so preserve the output states */
      begin
        count <= count;
        tc <= tc;
      end
    end
  end
//`#end` -- edit above this line, do not edit this line
endmodule
//`#start footer` -- edit after this line, do not edit this line
`endif /* Count4Bit_v1_20_V_ALREADY_INCLUDED */
//`#end` -- edit above this line, do not edit this line
```

Appendix B: EdgeDetect Cypress Catalog Component

```
/*******************************************************************************
* File Name: EdgeDetect_v1_0.v
* Version `$CY_MAJOR_VERSION`.`$CY_MINOR_VERSION`
*
* Description:
* The Edge Detector component samples the connected signal and produces a pulse when the selected edge
* occurs. This file describes the component functionality in Verilog.
********************************************************************************
* I*O Signals:
********************************************************************************
*   Name        Direction       Description
*   det         output          Detected edge
*   clock       input           Sampling clock
*   d           input           Signal to sample for edge
********************************************************************************
`include "cypress.v"

`ifdef EdgeDetect_v1_0_V_ALREADY_INCLUDED
`else
`define EdgeDetect_v1_0_V_ALREADY_INCLUDED

module EdgeDetect_v1_0 (
    det,    /* Detected edge */
    clock,  /* Sampling clock */
    d       /* Signal to sample for edge */
);
    /***************************************************************************
    *             Parameters
    ***************************************************************************/
    parameter [1:0] EdgeType = 0;

    /***************************************************************************
    *            Interface Definition
    ***************************************************************************/
    output wire det;
    input wire clock;
    input wire d;

    /* Edge Types */
    localparam [1:0] EDGE_DETECT_EDGETYPE_RISING = 2'd0;
    localparam [1:0] EDGE_DETECT_EDGETYPE_FALLING  = 2'd1;
    localparam [1:0] EDGE_DETECT_EDGETYPE_EITHER = 2'd2;

    reg last;

    always @(posedge clock)
    begin
        last <= d;
    end

    generate
    if (EdgeType == EDGE_DETECT_EDGETYPE_RISING)
    begin
        assign det = (~last & d);
    end
    else if (EdgeType == EDGE_DETECT_EDGETYPE_FALLING)
    begin
        assign det = (last & ~d);
    end
    else if (EdgeType == EDGE_DETECT_EDGETYPE_EITHER)
    begin
        assign det = (last ^ d);
    end
    endgenerate

endmodule

`endif /* EdgeDetect_v1_0_V_ALREADY_INCLUDED */
```