
DO NOT SHARE
SLIDES AND CLASS MATERIALS
ON ONLINE SITES
Course Hero

Advanced Logic Design
Verilog-HDL

Mingoo Seok

Columbia University

BV: Secs. 2.9, 2.10, 3.5, 4.6, 5.12-5.14, 6.4, 6.5, 7.1

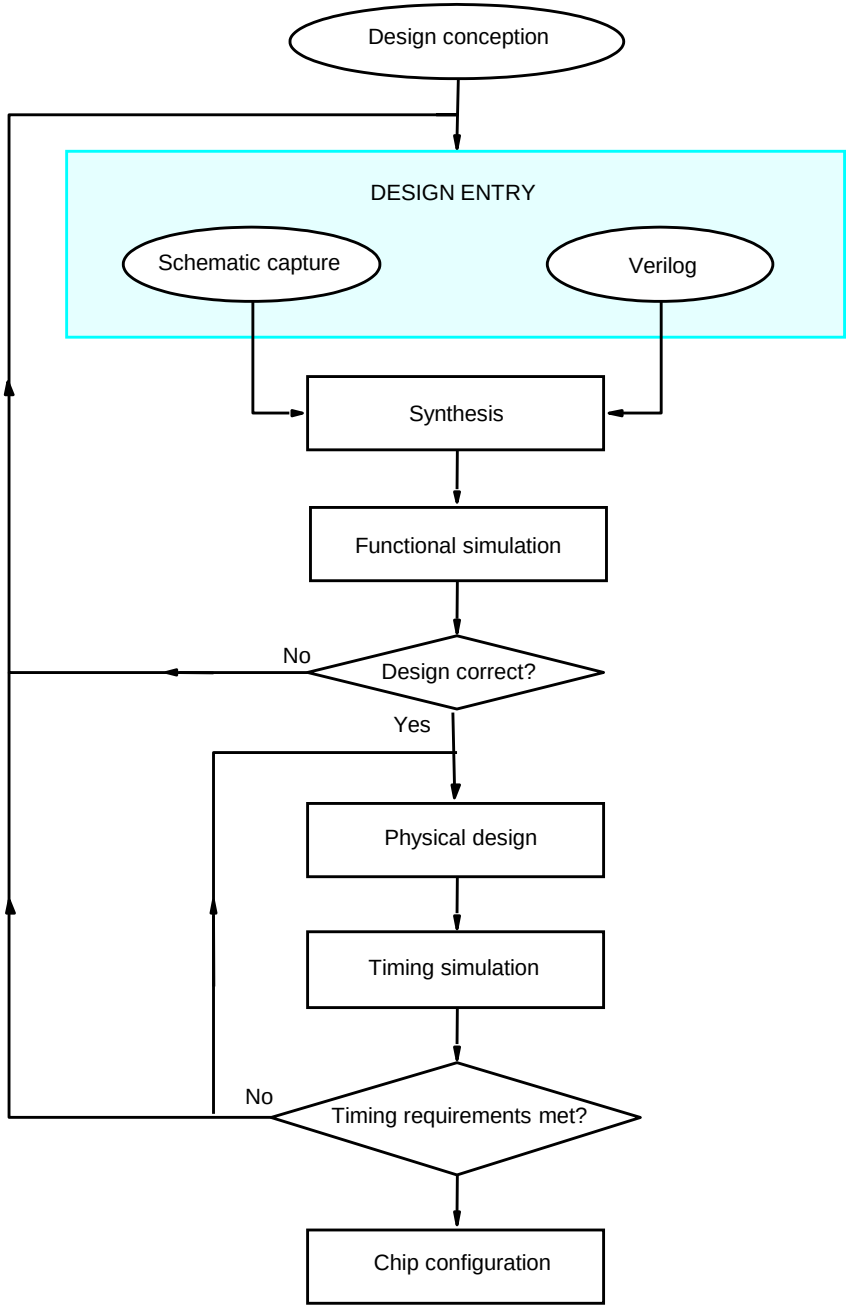
Outline

- L03 – Verilog introduction

- L04 – Sequential
- L03 – Verilog – Sequential Section

- L05 – FSM
- L03 – Verilog – FSM Section

- L06 – Computer Arithmetic
- L03 – Verilog – Computer Arithmetic



- Design entry
  - Schematic capture
  - HDL (Verilog)
- Logic synthesis
- Functional simulation
- Physical design
- Timing simulation
- Circuit implementation
  - Chip fabrication
  - Configuration

About Verilog

- 1980s: developed by Gateway Design Automation, later acquired by Cadence Design Systems
- 1990: put into the public domain
- 1995: IEEE standardized (1364-1995)
- 2001: updated (1364-2001)
- Include the features for (i) logic synthesis, (ii) verification and simulation; we will focus on synthesis

Verilog Variants

- SystemVerilog – Verilog added with features from C and C++
  - Popular for modeling & verification (not synthesis)
- Verilog AMS – a derivative of the Verilog that includes Analog and Mixed-Signal (AMS) extension; it contains both event-based simulator and continuous-time simulator, both of which are coupled
  - Only for modeling & verification

How Not to Write a Good Verilog Code

- A good general guideline is to assume that if the designer cannot directly determine what logic circuit is described by the code, then the CAD tools also are not likely to synthesize the circuit that the designer is trying to create
- Advised to adopt the same style of the code in the book before you invent your own

Signal

- Signal
  - Logic low (0), logic high (1)
  - Z: high-impedance
  - X: don’t care
- Vector variable (digital number)
  - [size][`base]constant
  - E.g., 4’b100: a binary number 0100 = $4_{(10)}$
  - E.g., 4’bx: an unknown 4-bit value xxxx
- Parameter
  - Pre-synthesis replacement
  - E.g., parameter n = 4;

Nets, Memory, Variables

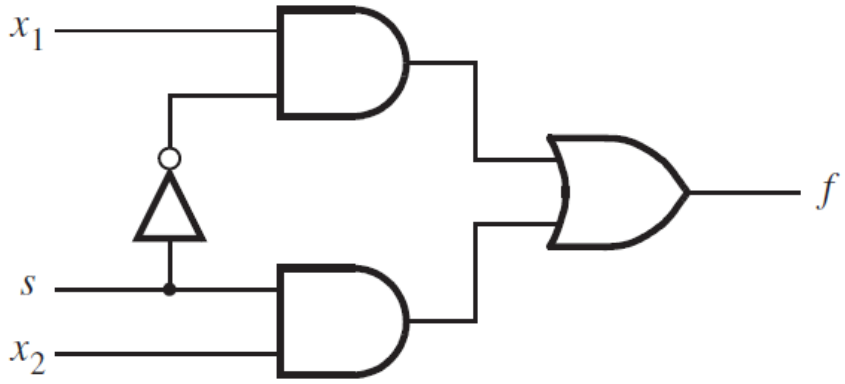
- Nets
  - Represent a node in a circuit
  - wire: connect one element to another
  - tri: a net that has tri-states (logic high, low, and high-Z)
- Memory
  - reg
  - However, it *doesn't* guarantee to become a storage gate such as latches or flip-flops, post-synthesis
  - Reg [7:0] R [3:0]; → four eight-bit variables
- Variable
  - reg, integer
  - Used to describe circuit's behavior, mostly in a testbench

Gate Level Primitives

Name	Description	Usage
and	$f = (a \cdot b \cdot \cdot)$	**and**(f, a, b,…)
nand	$f = \overline{(a \cdot b \cdot \cdot)}$	**nand**(f, a, b, …)
or	$f = (a + b + \cdot \cdot)$	**or** (f, a, b, …)
nor	$f = \overline{(a + b + \cdot \cdot)}$	**nor**(f, a, b, …)
xor	$f = (a \oplus b \oplus \cdot \cdot)$	**xor**(f, a, b, …)
xnor	$f = (a \odot b \odot \cdot \cdot)$	**xnor**(f, a, b, …)
not	$f = \overline{a}$	**not**(f, a)
buf	$f = a$	**buf** (f, a)
notif0	$f = (!e \ ? \overline{a} : \text{`}bz)$	**notif0**(f, a, e)
notif1	$f = (e \ ? \overline{a} : \text{`}bz)$	**notif1**(f, a, e)
bufif0	$f = (!e \ ? \ a : \text{`}bz)$	**bufif0**(f, a, e)
bufif1	$f = (e \ ? \ a : \text{`}bz)$	**bufif1**(f, a, e)

- Commonly-used logic gates

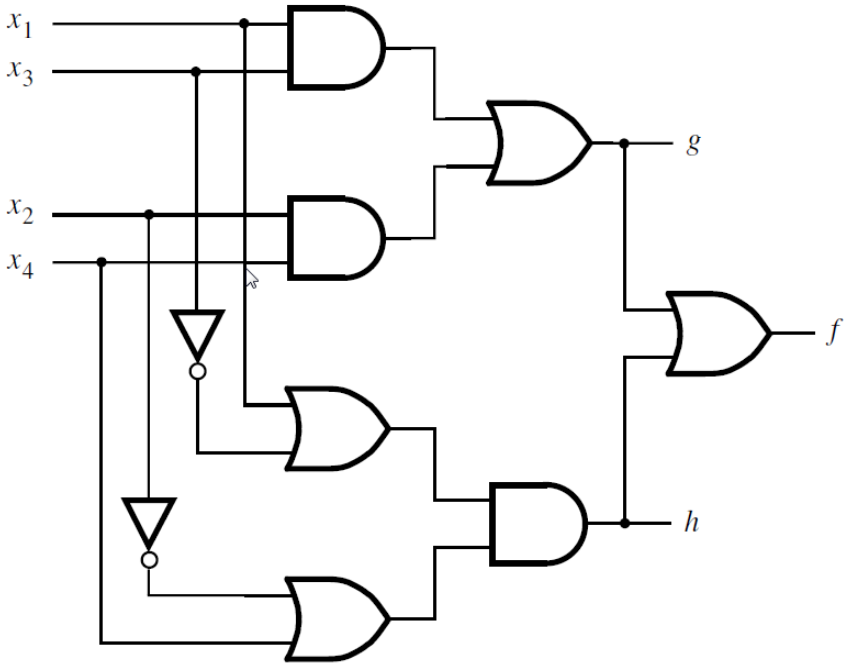
Module




```
module example1 (x1, x2, s, f);
    input x1, x2, s;
    output f;

    not (k, s);
    and (g, k, x1);
    and (h, s, x2);
    or (f, g, h);

endmodule
```

- A logic circuit is specified in the form of an *module*
- Structural presentation

Structural Presentation

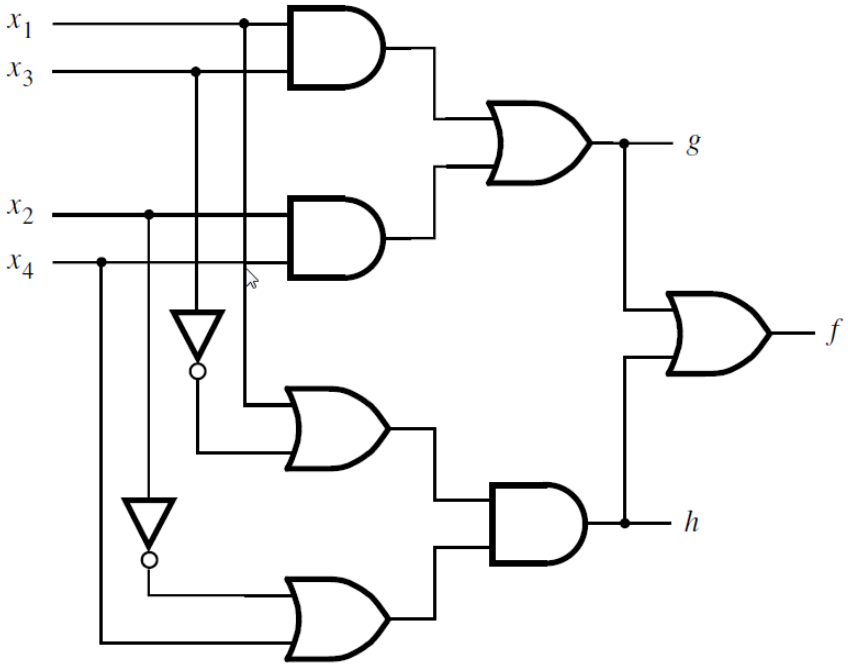



```
module example2 (x1, x2, x3, x4, f, g, h);
        input x1, x2, x3, x4;
        output f, g, h;

        and (z1, x1, x3);
        and (z2, x2, x4);
        or (g, z1, z2);
        or (z3, x1, ~x3);
        or (z4, ~x2, x4);
        and (h, z3, z4);
        or (f, g, h);

endmodule
```

Syntex

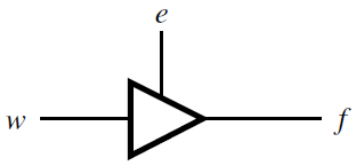
- Name of modules and signals:
  - Start w/ letter, only containing letter, number, underscore, and $
  - Case sensitive
- White space, TAB, indentation, and blank lines are ignored
- Commenting: use //

Behavior Presentation

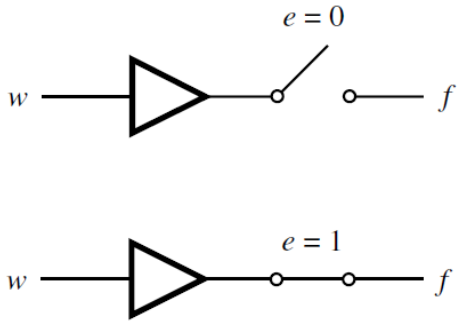



```
module example4 (x1, x2, x3, x4, f, g, h);
        input x1, x2, x3, x4;
        output f, g, h;

        assign g = (x1 & x3) | (x2 & x4);
        assign h = (x1 | ~x3) & (~x2 | x4);
        assign f = g | h;

endmodule
```

Tristate Buffer



(a) Symbol



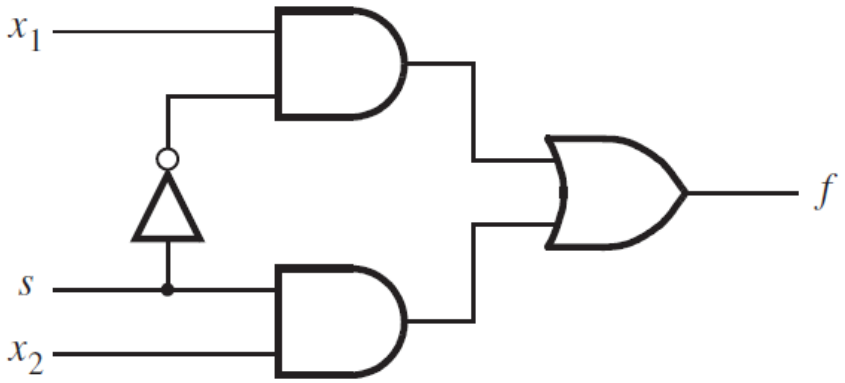
(b) Equivalent circuit

e	w	f
0	0	Z
0	1	Z
1	0	0
1	1	1

(c) Truth table

```
module trin (Y, E, F);
    input Y;
    input E;
    output wire F;

    assign F = E ? Y : 1'bz;

endmodule
```

Behavior Presentation: Procedure Statement




```
// Behavioral specification
module  example5 (x1, x2, s, f);
    input  x1, x2, s;
    output f;
    reg f;

    always @(x1 or x2 or s)
        if (s == 0)
            f = x1;
        else
            f = x2;

endmodule
```

- Always
- Sensitivity list
  - Always @*

Table 4.2. Verilog operators.

Operator type	Operator symbols	Operation performed	Number of operands
Bitwise	~	1's complement	1
	&	Bitwise AND	2
	|	Bitwise OR	2
	^	Bitwise XOR	2
	~^ or ^~	Bitwise XNOR	2
Logical	!	NOT	1
	&&	AND	2
	||	OR	2
Reduction	&	Reduction AND	1
	~&	Reduction NAND	1
	|	Reduction OR	1
	~|	Reduction NOR	1
	^	Reduction XOR	1
	~^ or ^~	Reduction XNOR	1

Arithmetic	+ − − * /	Addition Subtraction 2's complement Multiplication Division	2 2 1 2 2
Relational	>> < >= <=	Greater than Less than Greater than or equal to Less than or equal to	2 2 2 2
Equality	== !=	Logical equality Logical inequality	2 2
Shift	>> <<	Right shift Left shift	2 2
Concatenation	{,}	Concatenation	Any number
Replication	{{}}	Replication	Any number
Conditional	?:	Conditional	3

Conditional Operator

```
module mux2to1 (w0, w1, s, f);
    input w0, w1, s;
    output f;

    assign f = s ? w1 : w0;

endmodule
```

```
module mux2to1 (w0, w1, s, f);
    input w0, w1, s;
    output reg f;

    always @(w0, w1, s)
        f = s ? w1 : w0;

endmodule
```

- Conditional expression ? true_expression : false_expression

Nested Style

```
module mux4to1 (w0, w1, w2, w3, S, f);
    input w0, w1, w2, w3;
    input [1:0] S;
    output f;

    assign f = S[1] ? (S[0] ? w3 : w2) : (S[0] ? w1 : w0);

endmodule
```

If-Else Statement

```
module mux2to1 (w0, w1, s, f);
    input w0, w1, s;
    output reg f;

    always @(w0, w1, s)
        if (s==0)
            f = w0;
        else
            f = w1;

endmodule
```

```
module mux4to1 (W, S, f);
    input [0:3] W;
    input [1:0] S;
    output reg f;

    always @(W, S)
        if (S == 0)
            f = W[0];
        else if (S == 1)
            f = W[1];
        else if (S == 2)
            f = W[2];
        else if (S == 3)
            f = W[3];

endmodule
```

Case (Casex, Casez) Statement

```
module mux4to1 (W, S, f);
    input [0:3] W;
    input [1:0] S;
    output reg f;

    always @(W, S)
        case (S)
            0: f = W[0];
            1: f = W[1];
            2: f = W[2];
            3: f = W[3];
        endcase

endmodule
```

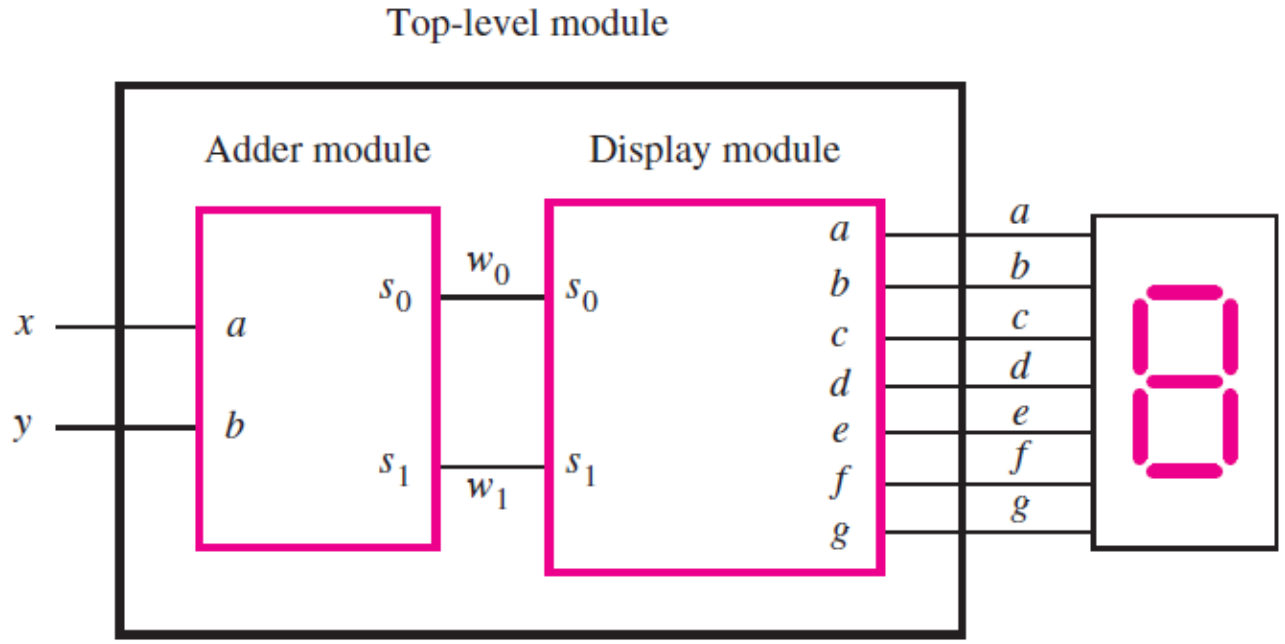
```
module priority (W, Y, z);
    input [3:0] W;
    output reg [1:0] Y;
    output reg z;

    always @(W)
    begin
        z = 1;
        casex (W)
            4'b1xxx: Y = 3;
            4'b01xx: Y = 2;
            4'b001x: Y = 1;
            4'b0001: Y = 0;
            default: begin
                        z = 0;
                        Y = 2'bx;
                     end
        endcase
    end

endmodule
```

For Loop Statement

```
module dec2to4 (W, En, Y);
    input [1:0] W;
    input En;
    output reg [0:3] Y;
    integer k;

    always @(W, En)
        for (k = 0; k <= 3; k = k+1)
        if ((W == k) && (En == 1))
            Y[k] = 1;
        else
            Y[k] = 0;

endmodule
```

- Unlike for-loops in high-level programming languages, the Verilog loop ***does not*** specify changes that take place in time through successive loop iterations
- Instead, during each iteration, it specifies different subcircuits

Hierarchical Code



- Top-level module contains two sub-modules, Adder and Display

```
module adder_display (x, y, a, b, c, d, e, f, g);
    input x, y;
    output a, b, c, d, e, f, g;
    wire w1, w0;

    adder U1 (x, y, w1, w0);
    display U2 (w1, w0, a, b, c, d, e, f, g);

endmodule
```

```
// An adder module
module  adder (a, b, s1, s0);
    input a, b;
    output s1, s0;

    assign s1 = a & b;
    assign s0 = a ^ b;

endmodule
```

```
// A module for driving a 7-segment display
module display (s1, s0, a, b, c, d, e, f, g);
    input s1, s0;
    output a, b, c, d, e, f, g;

    assign a = ~s0;
    assign b = 1;
    assign c = ~s1;
    assign d = ~s0;
    assign e = ~s0;
    assign f = ~s1 & ~s0;
    assign g = s1 & ~s0;

endmodule
```

Generate Construct

```
module addern (carryin, X, Y, S, carryout);
    parameter n=32;
    input carryin;
    input [n-1:0] X, Y;
    output [n-1:0] S;
    output carryout;
    wire [n:0] C;

    genvar k;
    assign C[0] = carryin;
    assign carryout = C[n];
    generate
        for (k = 0; k < n; k = k+1)
        begin: fulladd_stage
            wire z1, z2, z3; //wires within full-adder
            xor (S[k], X[k], Y[k], C[k]);
            and (z1, X[k], Y[k]);
            and (z2, X[k], C[k]);
            and (z3, Y[k], C[k]);
            or (C[k+1], z1, z2, z3);
        end
    endgenerate

endmodule
```

- Used to create multiple instances of subcircuits
- genvar for iteration variable; no integer

For vs. Generate For

- A *for-loop in an 'always'* block can do many of the same things a *'generate' for-loop* can do. One major difference is that you can't instantiate blocks in a for loop. Only the *'generate' for loop* allows you to control instantiation.
- Generate is more synthesis-friendly, in my opinion
- Another way is to write a Perl/ Python code to generate a Verilog code, which I recommend!

```
module  mux16to1 (W, S16, f);
   input  [0:15] W;
   input  [3:0] S16;
   output  reg  f;

   always @(W, S16)
      case (S16[3:2])
         0: mux4to1 (W[0:3], S16[1:0], f);
         1: mux4to1 (W[4:7], S16[1:0], f);
         2: mux4to1 (W[8:11], S16[1:0], f);
         3: mux4to1 (W[12:15], S16[1:0], f);
      endcase

   // Task that specifies a 4-to-1 multiplexer
   task  mux4to1;
      input  [0:3] X;
      input  [1:0] S4;
      output  reg  g;

      case (S4)
         0: g = X[0];
         1: g = X[1];
         2: g = X[2];
         3: g = X[3];
      endcase
   endtask

endmodule
```

Task

- To avoid replicating specific subroutines
- Must be in the module that calls it
- Task input, output are not real ports; may disappear after logic synthesis

```
module mux16to1 (W, S16, f);
   input [0:15] W;
   input [3:0] S16;
   output reg f;

   // Function that specifies a 4-to-1 multiplexer
   function mux4to1;
      input [0:3] X;
      input [1:0] S4;

      case (S4)
         0: mux4to1 = X[0];
         1: mux4to1 = X[1];
         2: mux4to1 = X[2];
         3: mux4to1 = X[3];
      endcase
   endfunction

   always @(W, S16)
      case (S16[3:2])
         0: f = mux4to1 (W[0:3], S16[1:0]);
         1: f = mux4to1 (W[4:7], S16[1:0]);
         2: f = mux4to1 (W[8:11], S16[1:0]);
         3: f = mux4to1 (W[12:15], S16[1:0]);
      endcase

endmodule
```

Function

- Similar to the task
- Can be outside of a module
- Typically defined before use
- A function can invoke another function but not a task
- A task can invoke another function and task
- Tip: better stick to either of function or task in a code

Per-Instance Parameter Assignment

```
module adder_hier (A, B, C, D, S, T, overflow);
    input [15:0] A, B;
    input [7:0] C, D;
    output [16:0] S;
    output [8:0] T;
    output overflow;

    wire o1, o2;  // used for the overflow signals

    addern U1 (1'b0, A, B, S[15:0], S[16], o1);
    defparam U1.n = 16;
    addern U2 (1'b0, C, D, T[7:0], T[8], o2);
    defparam U2.n = 8;

    assign overflow = o1 | o2;

endmodule
```

```
module addern (carryin, X, Y, S, carryout, overflow);
    parameter  n = 32;
    input carryin;
    input [n-1:0] X, Y;
    output reg [n-1:0] S;
    output reg carryout, overflow;

    always @(X, Y, carryin)
    begin
        {carryout, S} = X + Y + carryin;
        overflow = (X[n-1] & Y[n-1] & ~S[n-1]) | (~X[n-1] & ~Y[n-1] & S[n-1]);
    end

endmodule
```

Per-Instance Parameter Assignment

```
module adder_hier (A, B, C, D, S, T, overflow);
    input [15:0] A, B;
    input [7:0] C, D;
    output [16:0] S;
    output [8:0] T;
    output overflow;

    wire o1, o2;  // used for the overflow signals

    addern  #(.n(16)) U1 (1'b0, A, B, S[15:0], S[16], o1);
    addern  #(.n(8)) U2 (1'b0, C, D, T[7:0], T[8], o2);

    assign overflow = o1 | o2;

endmodule
```

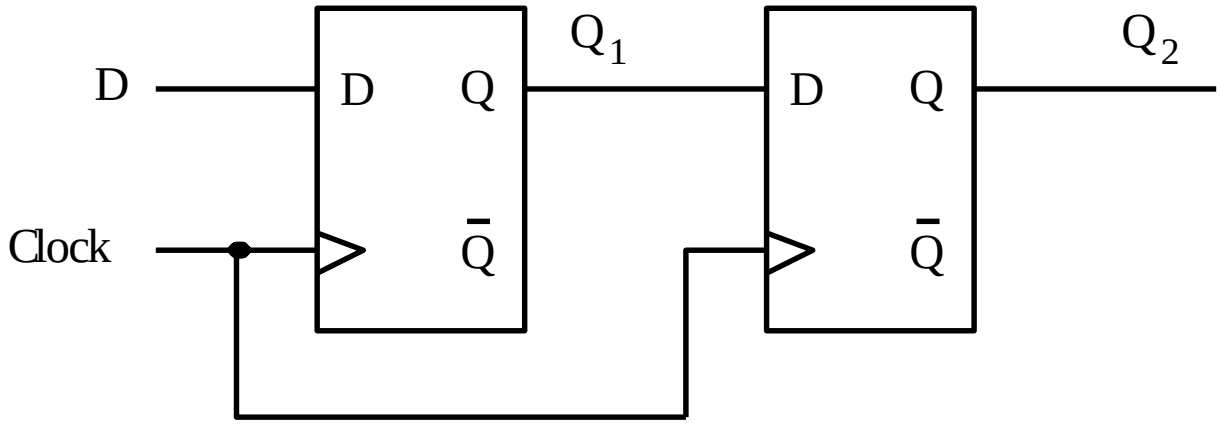
Sequential Logic

Sequencing Elements

```
module D_latch (D, Clk, Q);
    input D, Clk;
    output reg Q;

    always @(D, Clk)
        if (Clk)
            Q = D;

endmodule
```

```
module flipflop (D, Clock, Q);
    input D, Clock;
    output reg Q;

    always @(posedge Clock)
        Q = D;

endmodule
```

Blocking and Non-Blocking Assignment



Blocking and Non-Blocking Assignment

```
module example5_3 (D, Clock, Q1, Q2);
    input D, Clock;
    output reg Q1, Q2;

    always @(posedge Clock)
    begin
        Q1 = D;
        Q2 = Q1;
    end

endmodule
```

Incorrect code

```
module example5_4 (D, Clock, Q1, Q2);
    input D, Clock;
    output reg Q1, Q2;

    always @(posedge Clock)
    begin
        Q1 <= D;
        Q2 <= Q1;
    end

endmodule
```

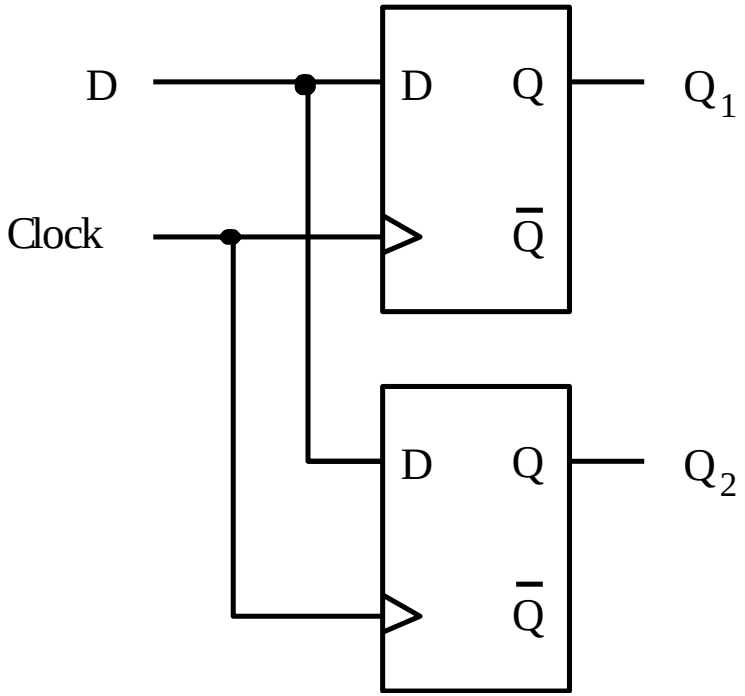
Correct

Blocking and Non-Blocking Assignment

- Blocking assignment (=)
  - Blocking assignment executes "in series" because a blocking assignment blocks execution of the next statement until it completes. Therefore the results of the next statement may depend on the first one being completed.
- Non-blocking (parallel) assignment (<=)
  - Non-blocking assignment executes in parallel because it describes assignments that all occur at the same time. The result of a statement on the 2nd line will not depend on the results of the statement on the 1st line. Instead, the 2nd line will execute as if the first line had not happened yet.

Blocking and Non-Blocking Assignment

```
module example5_3 (D, Clock, Q1, Q2);
    input D, Clock;
    output reg Q1, Q2;

    always @(posedge Clock)
    begin
        Q1 = D;
        Q2 = Q1;
    end

endmodule
```



Incorrect code

Blocking and Non-Blocking Assignment

```
module example5_4 (D, Clock, Q1, Q2);
    input D, Clock;
    output reg Q1, Q2;

    always @(posedge Clock)
    begin
        Q1 <= #2 D;
        Q2 <= #2 Q1;
    end

endmodule
```

- A better practice is to add an artificial delay
- #2: Q1 is assigned to D after 2 unit time after the clock edge
- Note: this artificial delay is ignored in logic synthesis

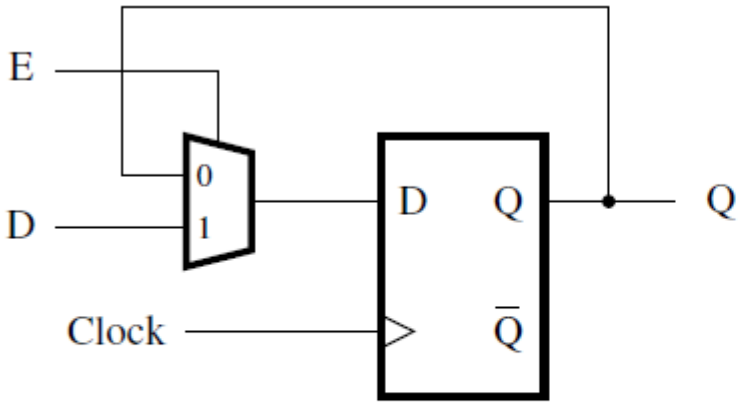
Asynchronous Reset

```
module flipflop (D, Clock, Resetn, Q);
    input D, Clock, Resetn;
    output reg Q;

    always @(negedge Resetn, posedge Clock)
        if (!Resetn)
            Q <= 0;
        else
            Q <= D;

endmodule
```

Synchronous Reset

```
module flipflop (D, Clock, Resetn, Q);
    input D, Clock, Resetn;
    output reg Q;

    always @(posedge Clock)
        if (!Resetn)
            Q <= 0;
        else
            Q <= D;

endmodule
```

Sequential Element with a Combinational Circuit



↑ *Can you correct it?*

```
module muxdff (D0, D1, Sel, Clock, Q);
    input D0, D1, Sel, Clock;
    output reg Q;

    wire D;
    assign D = Sel ? D1 : D0;

    always @(posedge Clock)
        Q <= D;

endmodule
```

4-b Shift Register

```
module shift4 (R, L, w, Clock, Q);
    input [3:0] R;
    input L, w, Clock;
    output [3:0] Q;
    wire [3:0] Q;

    muxdff Stage3 (w, R[3], L, Clock, Q[3]);
    muxdff Stage2 (Q[3], R[2], L, Clock, Q[2]);
    muxdff Stage1 (Q[2], R[1], L, Clock, Q[1]);
    muxdff Stage0 (Q[1], R[0], L, Clock, Q[0]);

endmodule
```

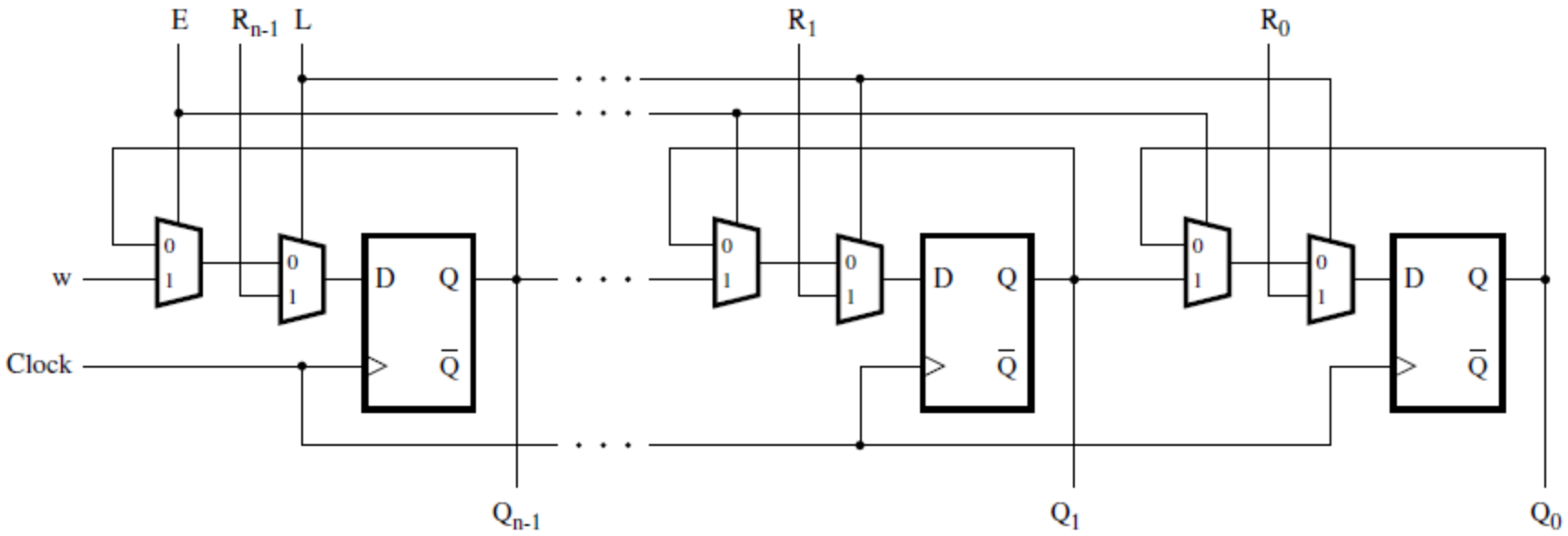
4-b Shift Register

```
module shift4 (R, L, w, Clock, Q);
    input [3:0] R;
    input L, w, Clock;
    output reg [3:0] Q;

    always @(posedge Clock)
        if (L)
            Q <= R;
        else
        begin
            Q[0] <= Q[1];
            Q[1] <= Q[2];
            Q[2] <= Q[3];
            Q[3] <= w;
        end
endmodule
```

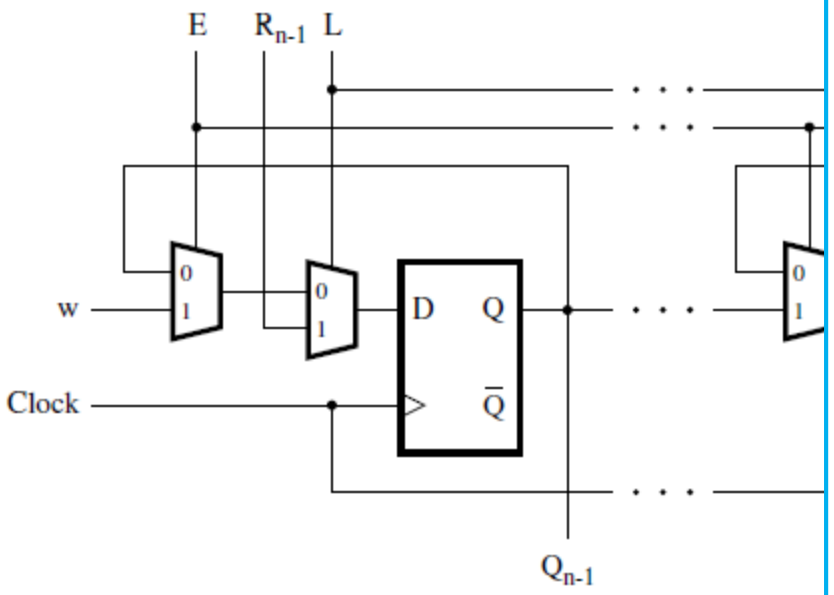
N-bit Shift Register

```
module shiftn (R, L, w, Clock, Q);
    parameter n = 16;
    input [n-1:0] R;
    input L, w, Clock;
    output reg [n-1:0] Q;
    integer k;

    always @(posedge Clock)
        if (L)
          Q <= R;
        else
        begin
          for (k = 0; k < n-1; k = k+1)
              Q[k] <= Q[k+1];
              Q[n-1] <= w;
        end

endmodule
```

Shift Register with Parallel Load and Enable



Shift Register with Parallel Load and Enable



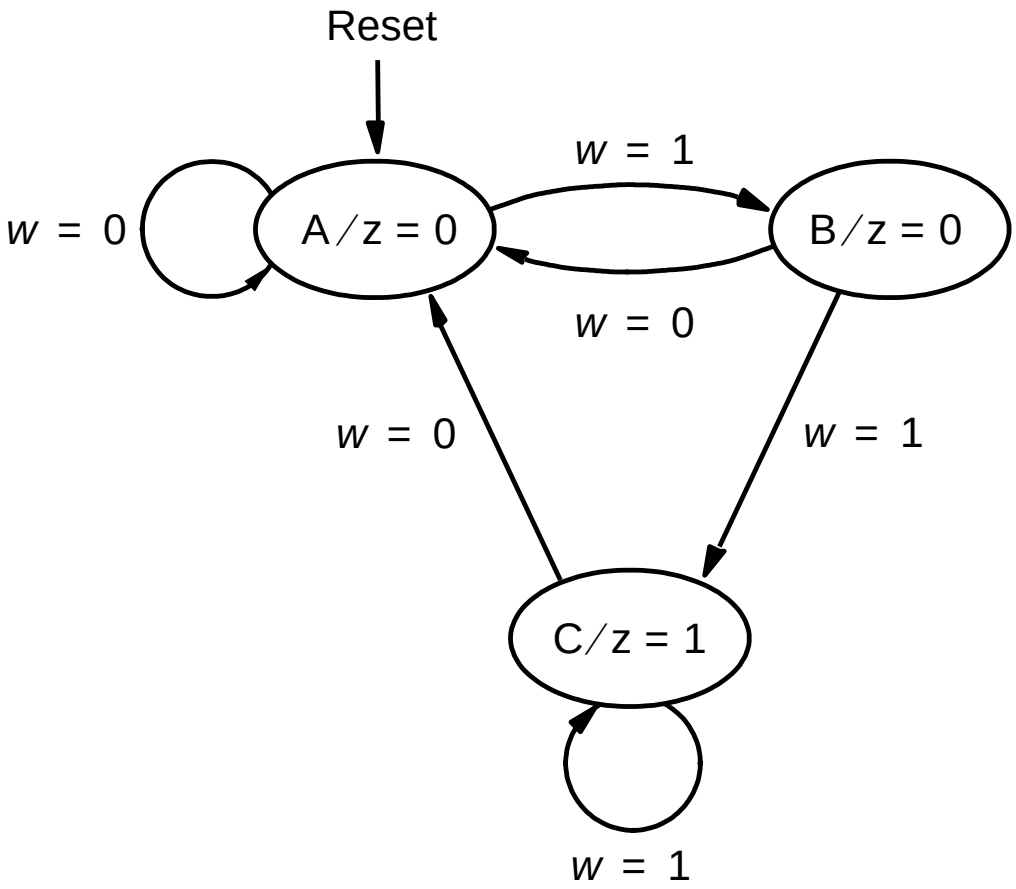

```
module shiftrne (R, L, E, w, Clock, Q);
    parameter n = 4;
    input [n-1:0] R;
    input L, E, w, Clock;
    output reg [n-1:0] Q;
    integer k;

    always @(posedge Clock)
    begin
        if (L)
            Q <= R;
        else if (E)
        begin
        Q[n-1] <= w;
        for (k = n-2; k >= 0; k = k-1)
            Q[k] <= Q[k+1];
        end
    end
endmodule
```

LFSR

```
module lfsr (R, L, Clock, Q);
    input [0:2] R;
    input L, Clock;
    output reg [0:2] Q;

    always @(posedge Clock)
        if (L)
            Q <= R;
        else
            Q <= {Q[2], Q[0] ^ Q[2], Q[1]};

endmodule
```

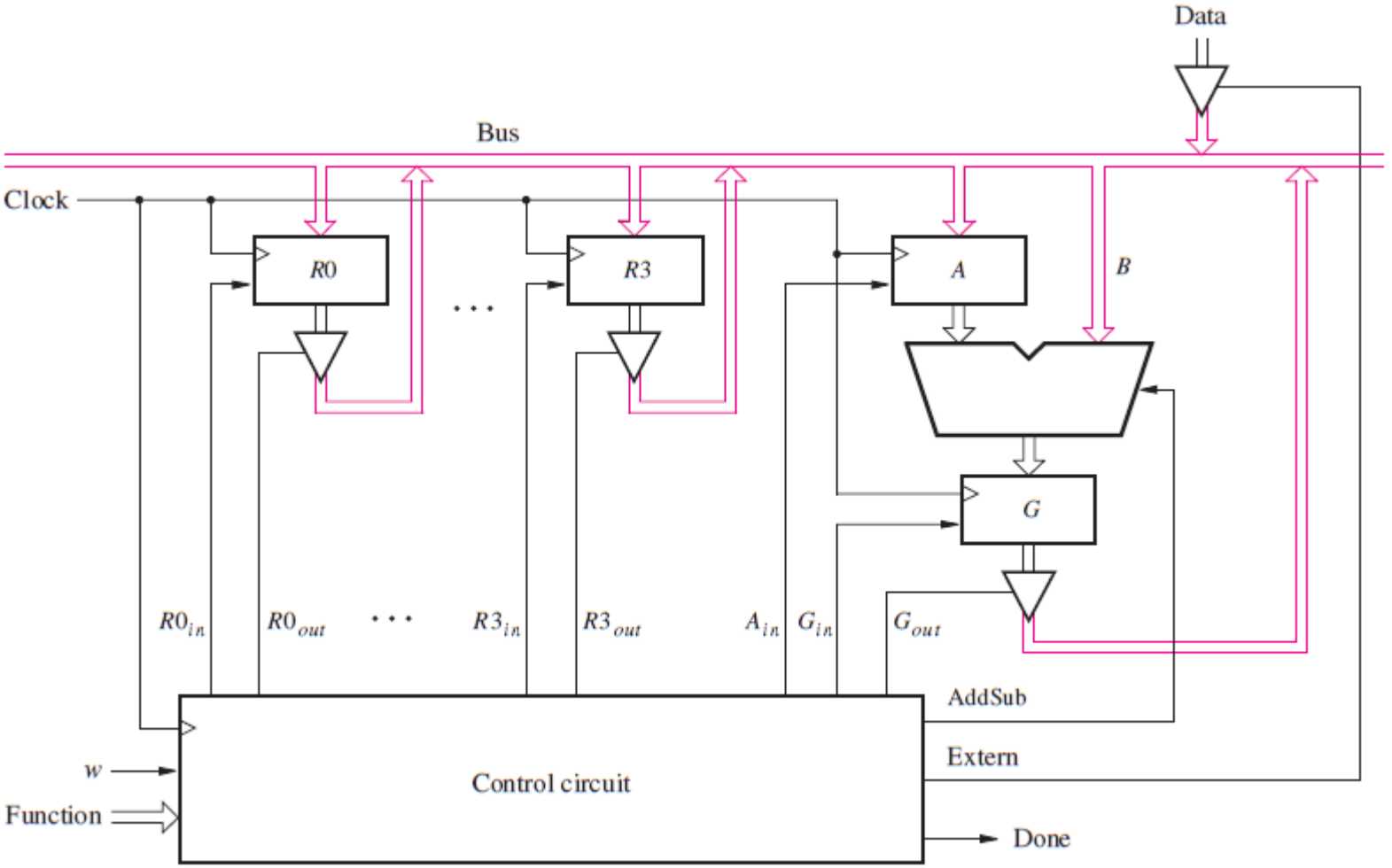
FSM

FSM

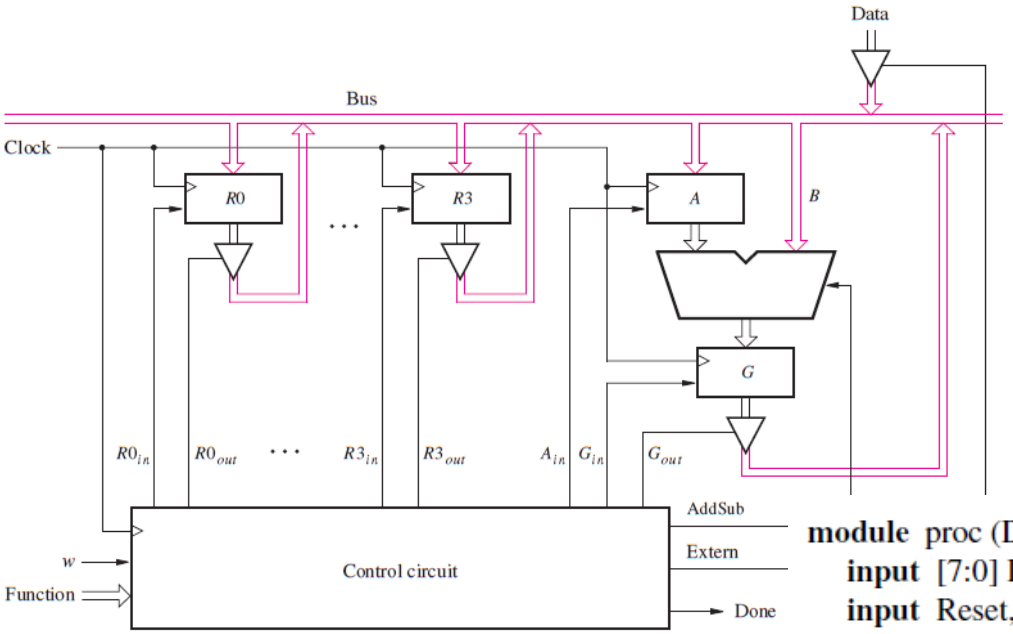



```
module simple (Clock, Resetn, w, z);
    input Clock, Resetn, w;
    output z;
    reg [2:1] y, Y;
    parameter [2:1] A = 2'b00, B = 2'b01, C = 2'b10;

    // Define the next state combinational circuit
    always @(w, y)
        case (y)
            A: if (w)   Y = B;
               else     Y = A;
            B: if (w)   Y = C;
               else     Y = A;
            C: if (w)   Y = C;
               else     Y = A;
            default:    Y = 2'bxx;
        endcase

    // Define the sequential block
    always @(negedge Resetn, posedge Clock)
        if (Resetn = = 0) y <= A;
        else y <= Y;

    // Define output
    assign z = (y = = C);

endmodule
```

A Simple Processor: uArch

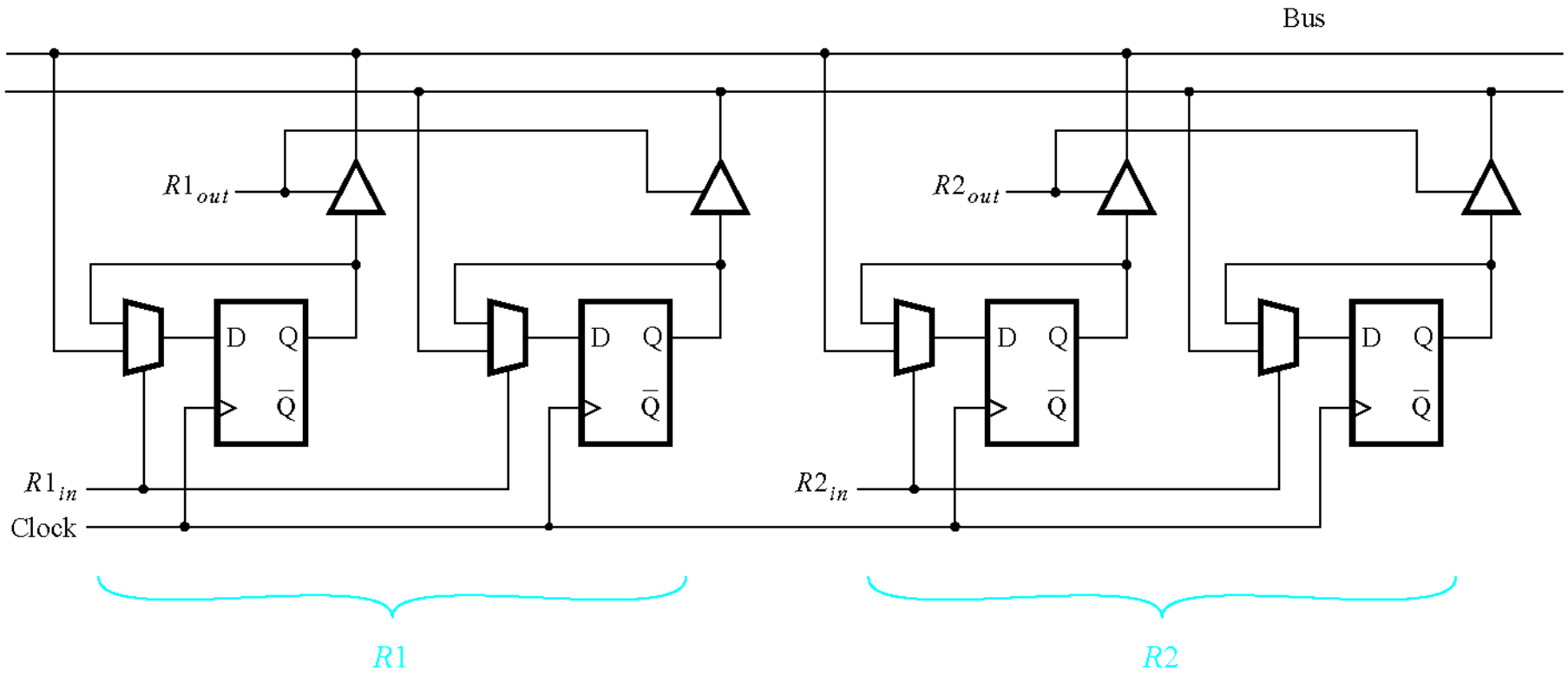


A Simple Processor: uArch




```
module proc (Data, Reset, w, Clock, F, Rx, Ry, Done, BusWires);
    input [7:0] Data;
    input Reset, w, Clock;
    input [1:0] F, Rx, Ry;
    output wire [7:0] BusWires;
    output Done;
    reg [0:3] Rin, Rout;
    reg [7:0] Sum;
    wire Clear, AddSub, Extern, Ain, Gin, Gout, FRin;
    wire [1:0] Count;
    wire [0:3] T, I, Xreg, Y;
    wire [7:0] R0, R1, R2, R3, A, G;
    wire [1:6] Func, FuncReg;
    integer k;
```

A Simple Processor: Bus

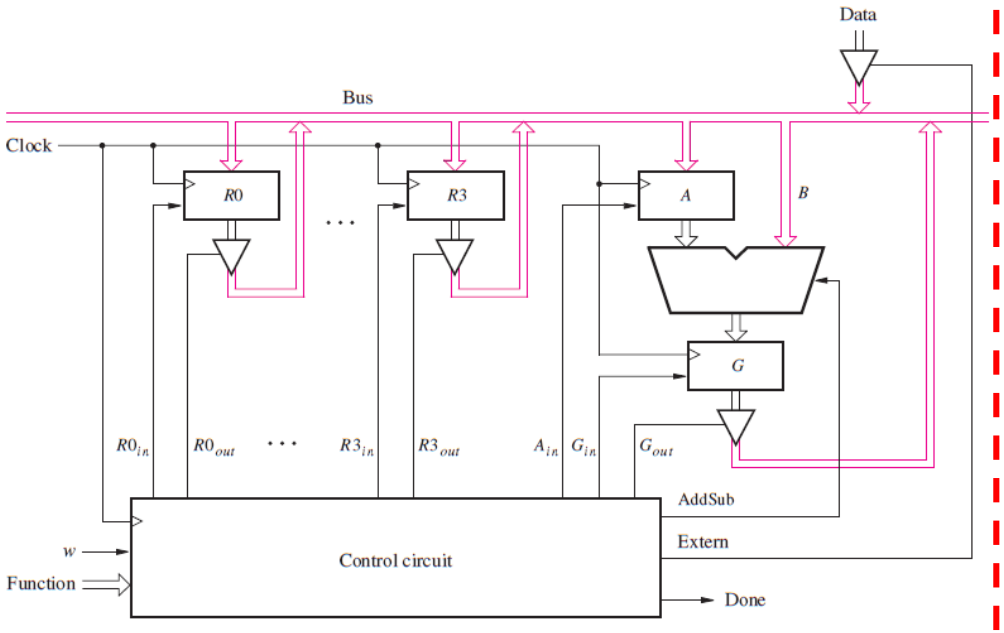


A Simple Processor: Bus

```
module trin (Y, E, F);
    parameter n = 8;
    input [n-1:0] Y;
    input E;
    output wire [n-1:0] F;

    assign F = E ? Y : 'bz;

endmodule
```

```
module  regn (R, L, Clock, Q);
    parameter n = 8;
    input  [n – 1:0] R;
    input  L, Clock;
    output  reg  [n – 1:0] Q;

    always @(posedge Clock)
        if (L)
            Q < = R;

endmodule
```

```
module proc (Data, Reset, w, Clock, F, Rx, Ry, Done, BusWires);
    input [7:0] Data;
    input Reset, w, Clock;
    input [1:0] F, Rx, Ry;
    output wire [7:0] BusWires;
    output Done;
    reg [0:3] Rin, Rout;
    reg [7:0] Sum;
    wire Clear, AddSub, Extern, Ain, Gin, Gout, ERin;
```




```
    dec2to4 decX (FuncReg[3:4], 1'b1, Xreg);
    dec2to4 decY (FuncReg[5:6], 1'b1, Y);

    assign Extern = I[0] & T[1];
    assign Done = ((I[0] | I[1]) & T[1]) | ((I[2] | I[3]) & T[3]);
    assign Ain = (I[2] | I[3]) & T[1];
    assign Gin = (I[2] | I[3]) & T[2];
    assign Gout = (I[2] | I[3]) & T[3];
    assign AddSub = I[3];
```

. . . continued in Part *b*.

```
    // RegCntl
    always @(I, T, Xreg, Y)
        for (k = 0; k < 4; k = k+1)
        begin
            Rin[k] = ((I[0] | I[1]) & T[1] & Xreg[k]) |
                ((I[2] | I[3]) & T[3] & Xreg[k]);
            Rout[k] = (I[1] & T[1] & Y[k]) | ((I[2] | I[3]) &
                ((T[1] & Xreg[k]) | (T[2] & Y[k])));
        end

    trin tri_ext (Data, Extern, BusWires);
    regn reg_0 (BusWires, Rin[0], Clock, R0);
    regn reg_1 (BusWires, Rin[1], Clock, R1);
    regn reg_2 (BusWires, Rin[2], Clock, R2);
    regn reg_3 (BusWires, Rin[3], Clock, R3);

    trin tri_0 (R0, Rout[0], BusWires);
    trin tri_1 (R1, Rout[1], BusWires);
    trin tri_2 (R2, Rout[2], BusWires);
    trin tri_3 (R3, Rout[3], BusWires);
    regn reg_A (BusWires, Ain, Clock, A);

    // alu
    always @(AddSub, A, BusWires)
        if (!AddSub)
            Sum = A + BusWires;
        else
            Sum = A – BusWires;

    regn reg_G (Sum, Gin, Clock, G);
    trin tri_G (G, Gout, BusWires);

endmodule
```

A Simple Processor: ISA

Operation	Function performed
Load *Rx*, *Data*	$Rx \leftarrow Data$
Move *Rx*, *Ry*	$Rx \leftarrow [Ry]$
Add *Rx*, *Ry*	$Rx \leftarrow [Rx] + [Ry]$
Sub *Rx*, *Ry*	$Rx \leftarrow [Rx] - [Ry]$

A Simple Processor: Function Decoder & Register

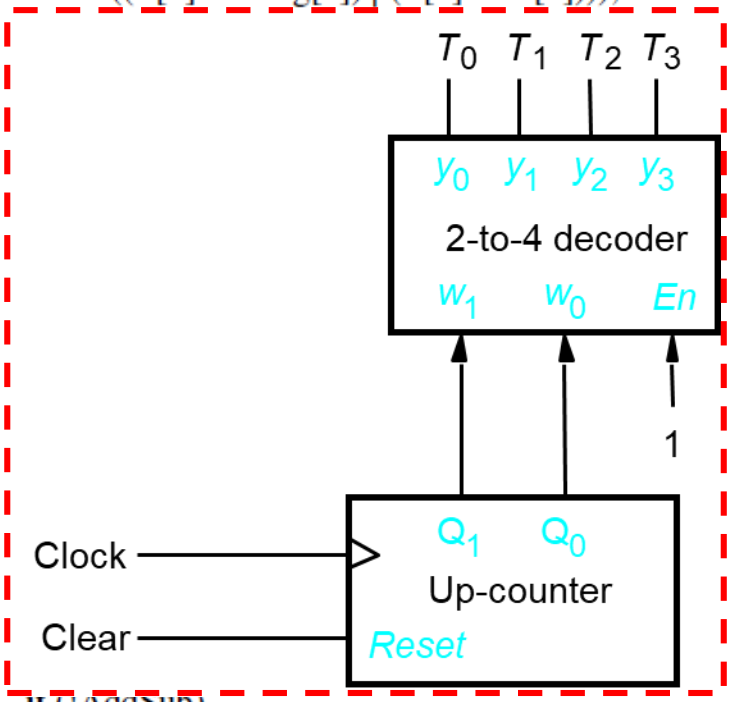
	T_1	T_2	T_3
(Load): I_0	$Extern, R_{in} = X,$ $Done$		
(Move): I_1	$R_{in} = X, R_{out} = Y,$ $Done$		
(Add): I_2	$R_{out} = X, A_{in}$	$R_{out} = Y, G_{in},$ $AddSub = 0$	$G_{out}, R_{in} = X,$ $Done$
(Sub): I_3	$R_{out} = X, A_{in}$	$R_{out} = Y, G_{in},$ $AddSub = 1$	$G_{out}, R_{in} = X,$ $Done$

```
module proc (Data, Reset, w, Clock, F, Rx, Ry, Done, BusWires);
    input [7:0] Data;
    input Reset, w, Clock;
    input [1:0] F, Rx, Ry;
    output wire [7:0] BusWires;
    output Done;
    reg [0:3] Rin, Rout;
    reg [7:0] Sum;
    wire Clear, AddSub, Extern, Ain, Gin, Gout, FRin;
    wire [1:0] Count;
    wire [0:3] T, I, Xreg, Y;
    wire [7:0] R0, R1, R2, R3, A, G;
    wire [1:6] Func, FuncReg;
    integer k;

    upcount counter (Clear, Clock, Count);
    dec2to4 decT (Count, 1'b1, T);

    assign Clear = Reset | Done | (~w & T[0]);
    assign Func = {F, Rx, Ry};
    assign FRin = w & T[0];

    regn functionreg (Func, FR
        defparam functionreg.n
    dec2to4 decI (FuncReg[1:2
    dec2to4 decX (FuncReg[3:
    dec2to4 decY (FuncReg[5:

    assign Extern = I[0] & T[1
    assign Done = ((I[0] | I[1]
    assign Ain = (I[2] | I[3]) &
    assign Gin = (I[2] | I[3]) &
    assign Gout = (I[2] | I[3]) & T[3];
    assign AddSub = I[3];
```

. . . continued in Part *b*.

```
module upcount (Clear, Clock, Q);
    input Clear, Clock;
    output reg [1:0] Q;

    always @(posedge Clock)
        if (Clear)
            Q <= 0;
        else
            Q <= Q + 1;

endmodule
```

```
// RegCntl
always @(I, T, Xreg, Y)
    for (k = 0; k < 4; k = k+1)
    begin
        Rin[k] = ((I[0] | I[1]) & T[1] & Xreg[k]) |
            ((I[2] | I[3]) & T[3] & Xreg[k]);
        Rout[k] = (I[1] & T[1] & Y[k]) | ((I[2] | I[3]) &
            ((T[1] & Xreg[k]) | (T[2] & Y[k])));
```

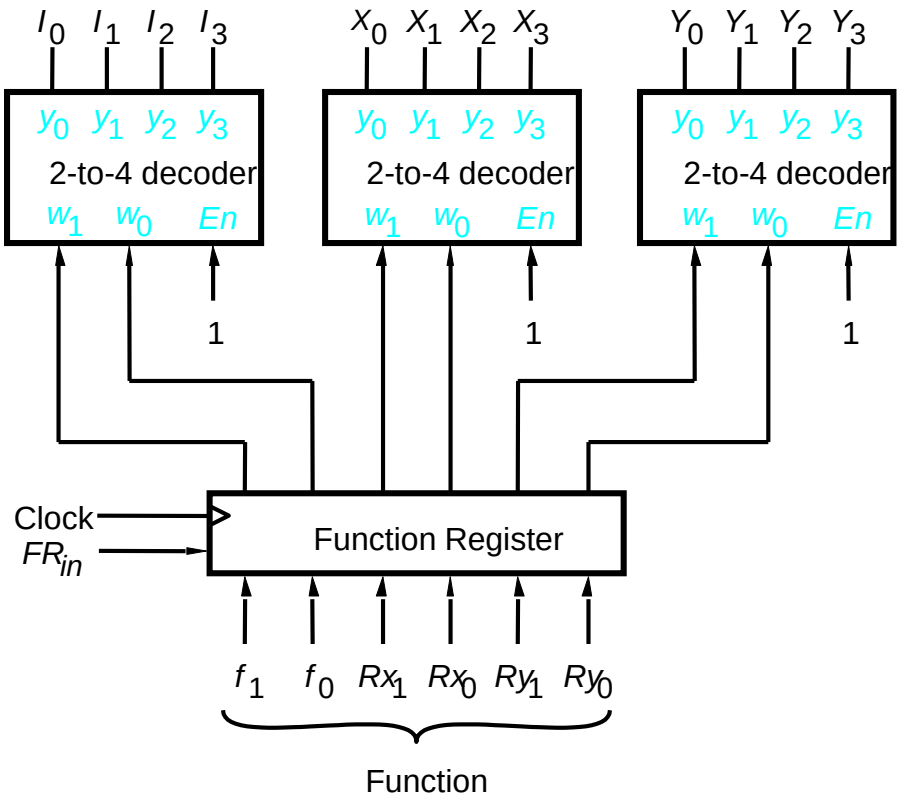



```
    if (!AddSub)
        Sum = A + BusWires;
    else
        Sum = A – BusWires;

regn reg_G (Sum, Gin, Clock, G);
trin tri_G (G, Gout, BusWires);

endmodule
```

```
module proc (Data, Reset, w, Clock, F, Rx, Ry, Done, BusWires);
    input [7:0] Data;
    input Reset, w, Clock;
    input [1:0] F, Rx, Ry;
    output wire [7:0] BusWires;
    output Done;
    reg [0:3] Rin, Rout;
    reg [7:0] Sum;
    wire Clear, AddSub, Extern, Ain, Gin, Gout, FRin;
    wire [1:0] Count;
    wire [0:3] T, I, Xreg, Y;
    wire [7:0] R0, R1, R2, R3, A, G;
    wire [1:6] Func, FuncReg;
    integer k;

    upcount counter (Clear, Clock, Count);
    dec2to4 decT (Count, 1'b1, T);

    assign Clear = Reset | Done | (~w & T[0]);
    assign Func = {F, Rx, Ry};
    assign FRin = w & T[0];

    regn functionreg (Func, FRin, Clock, FuncReg);
        defparam functionreg.n = 6;
    dec2to4 decI (FuncReg[1:2], 1'b1, I);
    dec2to4 decX (FuncReg[3:4], 1'b1, Xreg);
    dec2to4 decY (FuncReg[5:6], 1'b1, Y);

    assign Extern = I[0] & T[1];
    assign Done = ((I[0] | I[1]) & T[1]) | ((I[2] | I[3]) & T[3]);
    assign Ain = (I[2] | I[3]) & T[1];
    assign Gin = (I[2] | I[3]) & T[2];
    assign Gout = (I[2] | I[3]) & T[3];
    assign AddSub = I[3];
```

. . . continued in Part *b*.

```
    // RegCntl
    always @(I, T, Xreg, Y)
        for (k = 0; k < 4; k = k+1)
        begin
            Rin[k] = ((I[0] | I[1]) & T[1] & Xreg[k]) |
                ((I[2] | I[3]) & T[3] & Xreg[k]);
            Rout[k] = (I[1] & T[1] & Y[k]) | ((I[2] | I[3]) &
                ((T[1] & Xreg[k]) | (T[2] & Y[k])));
```




```
    regn reg_G (Sum, Gin, Clock, G);
    trin tri_G (G, Gout, BusWires);

endmodule
```

```
module proc (Data, Reset, w, Clock, F, Rx, Ry, Done, BusWires);
    input [7:0] Data;
    input Reset, w, Clock;
    input [1:0] F, Rx, Ry;
    output wire [7:0] BusWires;
    output Done;
    reg [0:3] Rin, Rout;
    reg [7:0] Sum;
    wire Clear, AddSub, Extern, Ain, Gin, Gout, FRin;
```

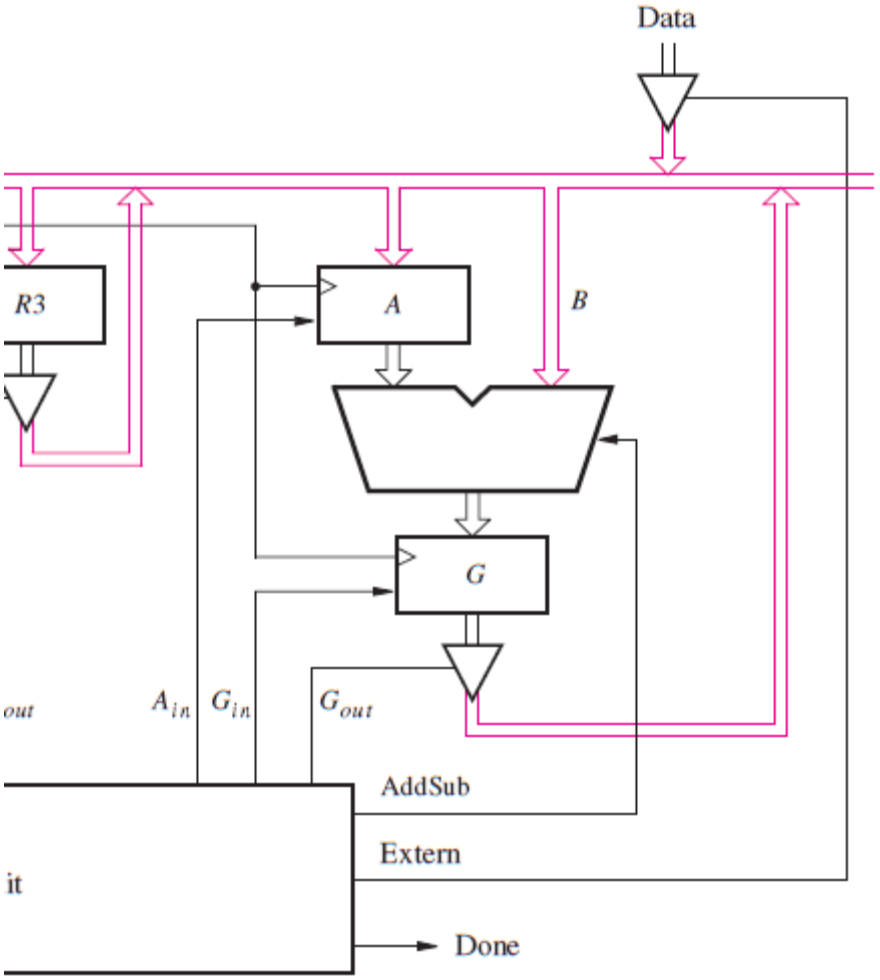
	T_1	T_2	T_3
(Load): I_0	*Extern*, $R_{in} = X$, *Done*		
(Move): I_1	$R_{in} = X$, $R_{out} = Y$, *Done*		
(Add): I_2	$R_{out} = X$, A_{in}	$R_{out} = Y$, G_{in}, $AddSub = 0$	G_{out}, $R_{in} = X$, *Done*
(Sub): I_3	$R_{out} = X$, A_{in}	$R_{out} = Y$, G_{in}, $AddSub = 1$	G_{out}, $R_{in} = X$, *Done*

```
    assign Func = {F, Rx, Ry};
    assign FRin = w & T[0];

    regn functionreg (Func, FRin, Clock, FuncReg);
        defparam functionreg.n = 6;
    dec2to4 decI (FuncReg[1:2], 1'b1, I);
    dec2to4 decX (FuncReg[3:4], 1'b1, Xreg);
    dec2to4 decY (FuncReg[5:6], 1'b1, Y);

    assign Extern = I[0] & T[1];
    assign Done = ((I[0] | I[1]) & T[1]) | ((I[2] | I[3]) & T[3]);
    assign Ain = (I[2] | I[3]) & T[1];
    assign Gin = (I[2] | I[3]) & T[2];
    assign Gout = (I[2] | I[3]) & T[3];
    assign AddSub = I[3];
```

. . . continued in Part *b*.

```
    // RegCntl
    always @(I, T, Xreg, Y)
        for (k = 0; k < 4; k = k+1)
        begin
            Rin[k] = ((I[0] | I[1]) & T[1] & Xreg[k]) |
                ((I[2] | I[3]) & T[3] & Xreg[k]);
            Rout[k] = (I[1] & T[1] & Y[k]) | ((I[2] | I[3]) &
                ((T[1] & Xreg[k]) | (T[2] & Y[k])));
        end

    trin tri_ext (Data, Extern, BusWires);
    regn reg_0 (BusWires, Rin[0], Clock, R0);
    regn reg_1 (BusWires, Rin[1], Clock, R1);
    regn reg_2 (BusWires, Rin[2], Clock, R2);
    regn reg_3 (BusWires, Rin[3], Clock, R3);

    trin tri_0 (R0, Rout[0], BusWires);
    trin tri_1 (R1, Rout[1], BusWires);
    trin tri_2 (R2, Rout[2], BusWires);
    trin tri_3 (R3, Rout[3], BusWires);
    regn reg_A (BusWires, Ain, Clock, A);

    // alu
    always @(AddSub, A, BusWires)
        if (!AddSub)
            Sum = A + BusWires;
        else
            Sum = A – BusWires;

    regn reg_G (Sum, Gin, Clock, G);
    trin tri_G (G, Gout, BusWires);

endmodule
```

A Simple Processor: ALU




```
// RegCntl
always @(I, T, Xreg, Y)
    for (k = 0; k < 4; k = k+1)
    begin
        Rin[k] = ((I[0] | I[1]) & T[1] & Xreg[k]) |
            ((I[2] | I[3]) & T[3] & Xreg[k]);
        Rout[k] = (I[1] & T[1] & Y[k]) | ((I[2] | I[3]) &
            ((T[1] & Xreg[k]) | (T[2] & Y[k])));
    end

trin tri_ext (Data, Extern, BusWires);
regn  reg_0 (BusWires, Rin[0], Clock, R0);
regn  reg_1 (BusWires, Rin[1], Clock, R1);
regn  reg_2 (BusWires, Rin[2], Clock, R2);
regn  reg_3 (BusWires, Rin[3], Clock, R3);

trin  tri_0 (R0, Rout[0], BusWires);
trin  tri_1 (R1, Rout[1], BusWires);
trin  tri_2 (R2, Rout[2], BusWires);
trin  tri_3 (R3, Rout[3], BusWires);
regn  reg_A (BusWires, Ain, Clock, A);

// alu
always @(AddSub, A, BusWires)
    if (!AddSub)
        Sum = A + BusWires;
    else
        Sum = A – BusWires;

regn  reg_G (Sum, Gin, Clock, G);
trin  tri_G (G, Gout, BusWires);

endmodule
```

```
module proc (Data, Reset, w, Clock, F, Rx, Ry, Done, BusWires);
    input [7:0] Data;
    input Reset, w, Clock;
    input [1:0] F, Rx, Ry;
    output wire [7:0] BusWires;
    output Done;
    reg [0:3] Rin, Rout;
    reg [7:0] Sum;
    wire Clear, AddSub, Extern, Ain, Gin, Gout, FRin;
    wire [1:0] Count;
    wire [0:3] T, I, Xreg, Y;
    wire [7:0] R0, R1, R2, R3, A, G;
    wire [1:6] Func, FuncReg;
    integer k;

    upcount counter (Clear, Clock, Count);
    dec2to4 decT (Count, 1'b1, T);

    assign Clear = Reset | Done | (~w & T[0]);
    assign Func = {F, Rx, Ry};
    assign FRin = w & T[0];

    regn functionreg (Func, FRin, Clock, FuncReg);
        defparam functionreg.n = 6;
    dec2to4 decI (FuncReg[1:2], 1'b1, I);
    dec2to4 decX (FuncReg[3:4], 1'b1, Xreg);
    dec2to4 decY (FuncReg[5:6], 1'b1, Y);

    assign Extern = I[0] & T[1];
    assign Done = ((I[0] | I[1]) & T[1]) | ((I[2] | I[3]) & T[3]);
    assign Ain = (I[2] | I[3]) & T[1];
    assign Gin = (I[2] | I[3]) & T[2];
    assign Gout = (I[2] | I[3]) & T[3];
    assign AddSub = I[3];
```

. . . continued in Part *b*.

```
    // RegCntl
    always @(I, T, Xreg, Y)
        for (k = 0; k < 4; k = k+1)
        begin
            Rin[k] = ((I[0] | I[1]) & T[1] & Xreg[k]) |
                ((I[2] | I[3]) & T[3] & Xreg[k]);
            Rout[k] = (I[1] & T[1] & Y[k]) | ((I[2] | I[3]) &
                ((T[1] & Xreg[k]) | (T[2] & Y[k])));
        end

    trin tri_ext (Data, Extern, BusWires);
    regn reg_0 (BusWires, Rin[0], Clock, R0);
    regn reg_1 (BusWires, Rin[1], Clock, R1);
    regn reg_2 (BusWires, Rin[2], Clock, R2);
    regn reg_3 (BusWires, Rin[3], Clock, R3);

    trin tri_0 (R0, Rout[0], BusWires);
    trin tri_1 (R1, Rout[1], BusWires);
    trin tri_2 (R2, Rout[2], BusWires);
    trin tri_3 (R3, Rout[3], BusWires);
    regn reg_A (BusWires, Ain, Clock, A);

    // alu
    always @(AddSub, A, BusWires)
        if (!AddSub)
            Sum = A + BusWires;
        else
            Sum = A – BusWires;

    regn reg_G (Sum, Gin, Clock, G);
    trin tri_G (G, Gout, BusWires);

endmodule
```

Computer Arithmetic and Other Discussions

Arithmetic

```
module addern (carryin, X, Y, S);
        parameter n = 32;
        input carryin;
        input [n-1:0] X, Y;
        output reg [n-1:0] S;

        always @(X, Y, carryin)
                S = X + Y + carryin;

endmodule
```

```
module adder4 (carryin, x3, x2, x1, x0, y3, y2, y1, y0, s3, s2,
    input carryin, x3, x2, x1, x0, y3, y2, y1, y0;
    output s3, s2, s1, s0, carryout;

    fulladd stage0 (carryin, x0, y0, s0, c1);
    fulladd stage1 (c1, x1, y1, s1, c2);
    fulladd stage2 (c2, x2, y2, s2, c3);
    fulladd stage3 (c3, x3, y3, s3, carryout);

endmodule

module fulladd (Cin, x, y, s, Cout);
    input Cin, x, y;
    output s, Cout;

    assign s = x ^ y ^ Cin,
    assign Cout = (x & y) | (x & Cin) | (y & Cin);

endmodule
```

Fixed-Point Arithmetic

```
module fixedtest();
    reg signed [7:0] a;
    reg signed [7:0] b;
    reg signed [7:0] c;
    reg signed [15:0] ab;  // large enough for product

    localparam sf = 2.0**-4.0;  // Q4.4 scaling factor is 2^-4

    initial begin
        $display("Fixed Point Examples by TimeToExplore.net.");

        a = 8'b0011_1010;  // 3.6250
        b = 8'b0100_0001;  // 4.0625
        c = a + b;         // 0111.1011 = 7.6875
        $display("%f + %f = %f", $itor(a)*sf, $itor(b)*sf, $itor(c)*sf);

        a = 8'b0011_1010;  // 3.6250
        b = 8'b1110_1000;  // -1.5000
        c = a + b;         // 0010.0010 = 2.1250
        $display("%f + %f = %f", $itor(a)*sf, $itor(b)*sf, $itor(c)*sf);

        a = 8'b0011_0100;  // 3.2500
        b = 8'b0010_0001;  // 2.0625
        ab = a * b;        // 00000110.10110100 = 6.703125
        c = ab[11:4];      // take middle 8 bits: 0110.1011 = 6.6875
        $display("%f x %f = %f", $itor(a)*sf, $itor(b)*sf, $itor(c)*sf);

        a = 8'b0111_1000;  // 7.5000
        b = 8'b0000_1000;  // 0.5000
        ab = a * b;        // 00000011.11000000 = 3.7500
        c = ab[11:4];      // take middle 8 bits: 0011.1100 = 3.7500
        $display("%f x %f = %f", $itor(a)*sf, $itor(b)*sf, $itor(c)*sf);
    end
endmodule
```

- Use ‘_’ for decimal point location, which will be ignored in the simulator and logic synthesis tool
- The $itor function converts an integer into a real number we can display.
- We divide our results by 2^4 to account for the lower four bits being fractional. For example, 00101000 would normally be interpreted as 40 (32+8), but we want it to be 2.5 (2 + 1/2). If we divide 40 by 2^4 we get the desired 2.5.

https://timetoexplore.net/blog/fixed-point-numbers-in-verilog

Common Mistakes in Verilog Coding (BV pp.728-731)

```
always @(w0, w1, s)
   if ( s == 1 )
      f = w1;
   f = w0;
```

```
always @(w0, w1, s)
begin
   if ( s = 1 )
      f = w1;
   f = w0;
end
```

```
always @(x)
begin
   s = x ^ y;
   c = x & y;
end
```

- Missing Begin-End
- Missing semicolon
- Missing {}: {3{A},2{B}} instead of {{3{a}},{2{B}}}
- Accidental assignment
- Incomplete sensitive list: sometimes okay in synthesis but not simulation

Common Mistakes in Verilog Coding (BV pp.728-731)

```
bit_count cbits (T, C);
    defparam bit_count.n = 8, bit_count.logn = 3;
```

```
bit_count cbits (T, C);
    defparam cbits.n = 8, cbits.logn = 3;
```

- Variables vesus nets: only nets can serve as the targets of continuous assignment statement; variable assigned values inside an always block have to be of type reg or integer;
- Assignments in multiple always blocks: a given variable should never be assigned a value in more than one always block
- Blocking vs. non-blocking: in always blocks, combinational circuits should use blocking (=); sequential should use non-blocking (<=);
- Module instantiation: defparam must reference the instance name of a module, not the subcircuit's module name

Common Mistakes in Verilog Coding (BV pp.728-731)

```
always @(LA)
  if (LA == 1)
    EA = 1;
```

```
always @(LA)
  if (LA == 1)
    EA = 1;
  else
    EA = 0;
```

```
always @(W)
  case (W)
    2'b01:  EA = 1;
    2'b10:  EB = 1;
  endcase
```

```
always @(W)
begin
  EA = 0; EB = 0;
  case (W)
    2'b01:  EA = 1;
    2'b10:  EB = 1;
  endcase
end
```

- Implied memory
  - The left of each pair creates a latch (memory) in the logic synthesis (e.g., EA, EB)