RISC-V

System-on-Chip Design

Harris, Stine, Thompson, & Harris

Chapter 4:

HDL Design Practices

Chapter 4 :: Topics

Hardware Description Language (HDL) Design Practices

4.1 HDL Design Methodology

4.2 Hardware Design with SystemVerilog

4.3 Wally Style Guidelines

HDL Design Methodology

Design Approach

- **Write specification before coding**
- **Block diagrams**
  - Start at top level
    - Sketch inputs, outputs, sub-blocks and their connections
  - Repeat recursively until done
- **Different designers own different blocks**
  - Rely on well-defined interfaces
- **Think of the hardware you want**
  - Write HDL idioms that generate that hardware
    - Use the synthesizable subset of the language
  - Don’t write like it’s a computer (software) program
  - Beware of bad examples on the Internet
  - Use the idioms from this chapter

Project Management Methods

Waterfall

- Traditional approach
- Divide project into tasks:
  - conception, design, build, test, deploy
- Review each step and freeze before next step
- Works well for industries like construction where steps are predictable
- Fails catastrophically on software and HDL because bugs aren't found until very late

Agile

- Iterative design process
- Frequent working prototypes
  - Start with key features
- Frequent customer feedback

Version Control

- **Essential for complex projects**
  - Provides backups
  - Allows one to roll back to the last working version
  - Allows teams to collaborate without overwriting
  - Allows experimenting in a branch before rolling into the main development
- **Git has taken over version control**

Verification

- **Verification is more time consuming and expensive than SoC design**
- **Must start with the goal of verification in mind**

- **Develop test cases before or concurrent with HDL**
  - So code can be tested as soon as it is written

- **Regression suite**
  - Repeatedly run on code each time it is modified
  - Longer regression run nightly
  - Early notification of trouble

- **Danger: don’t hack until the tests pass!**

Chapter 4: HDL Design Practices

Hardware Design With SystemVerilog

Hierarchical Design

- **Decompose complex systems hierarchically**
- **Upper levels are described structurally**
  - In terms of simpler modules
- **Bottom levels are described behaviorally**
  - Boolean equations, truth tables, FSMs, etc.

Combinational & Sequential Logic

- **Combinational Logic:**
  - Outputs depend on current inputs only
- **Sequential Logic:**
  - Outputs depend on current and previous inputs
  - Has memory
- **State:**
  - Minimum information about previous inputs needed to predict future outputs
- **Synchronous Sequential Logic:**
  - Every element is combinational logic or a flip-flop
  - All flip flops use the same clock (clk)
  - All feedback passes through a flip-flop

A large majority of digital design is synchronous sequential.

- Easy to design and verify

Chapter 4: HDL Design Practices

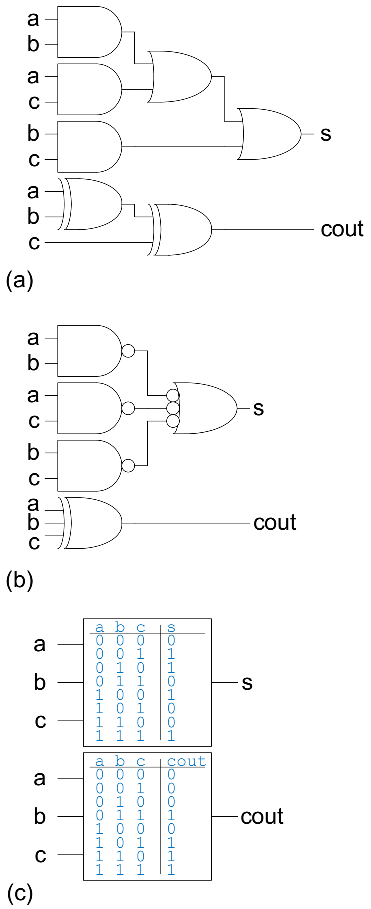
Combinational Logic

Modules

- **Module name**
- **Inputs & Outputs**
- **Behavioral or structural description**

Behavioral full adder example:

```
module fulladder(input  logic a, b, c,
                 output logic s, cout);

  assign s = a ^ b ^ c;
  assign cout = (a & b) | (a & c) | (b & c);
endmodule
```



(a)

(b)

a	b	c	s
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

a	b	c	cout
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

(c)

Operator Precedence

Only parenthesize when necessary or helpful

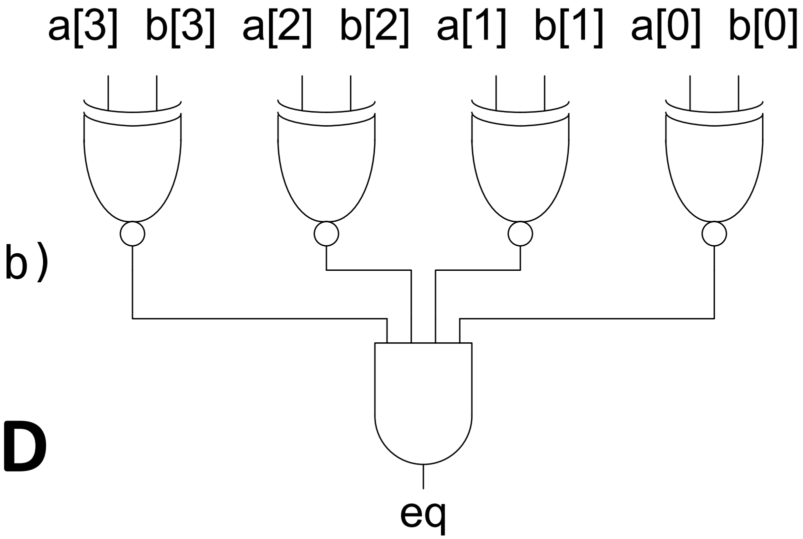
Precedence	Operator	Meaning
Highest	~	NOT
	*, /, %	Multiply, divide, modulo
	+, -	Add, subtract
	<<, >>	Logical left/right shift
	<<<, >>>	Arithmetic left/right shift
	<, <=, >, >=	Relative comparison
	==, !=	Equality comparison
	&, ~&	AND, NAND
	^, ~^	XOR, XNOR
	|, ~|	OR, NOR
Lowest	? :	Conditional (ternary)

Combinational Logic using `assign`

Express Boolean equations using `assign`

- ***Bitwise operators*** **take N bits in and out**
- ***Reduction operators*** **take N bits in, 1 bit out**

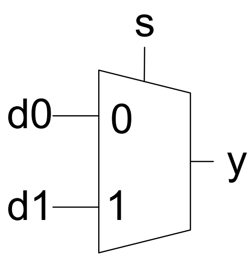
```
module eqcmp(input  logic [3:0] a, b,
             output logic       eq);

  assign eq = &(a ~^ b);
  // equivalent to assign eq = (a == b)
endmodule
```



- **Bitwise XNOR, reduction AND**

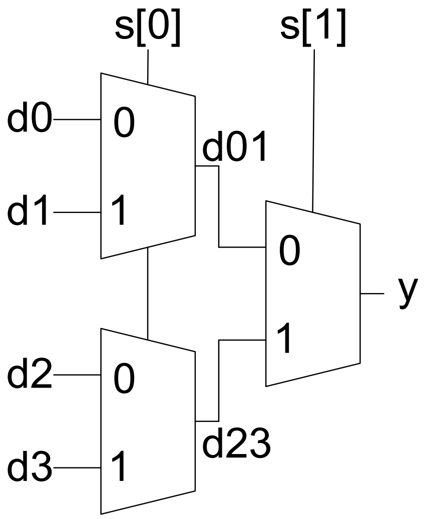
Multiplexer: Ternary Operator

Ternary operator ? : is more readable than gates



```
module mux2(input  logic [63:0] d0, d1,
            input  logic        s,
            output logic [63:0] y);

  assign y = s ? d1 : d0;

  /* alternative implementation using gates
  logic [63:0] s64;
  assign s64 = {64{s}}; // replicate s 64 times
  assign y = ~s64 & d0 | s64 & d1; */
endmodule
```

Mux4: Structural Implementation



```
// 4:1 mux
module mux4 #(parameter WIDTH = 64) (
  input  logic [WIDTH-1:0] d0, d1, d2, d3,
  input  logic [1:0]  s,
  output logic [WIDTH-1:0] y);

  logic [WIDTH-1:0] d01, d23;

  // structural implementation using 2:1 muxes
  mux2 #(WIDTH) lsbmux(d0, d1, s[0], d01);
  mux2 #(WIDTH) msbmux(d2, d3, s[0], d23);
  mux2 #(WIDTH) finalmux(d01, d23, s[1], y);

endmodule
```

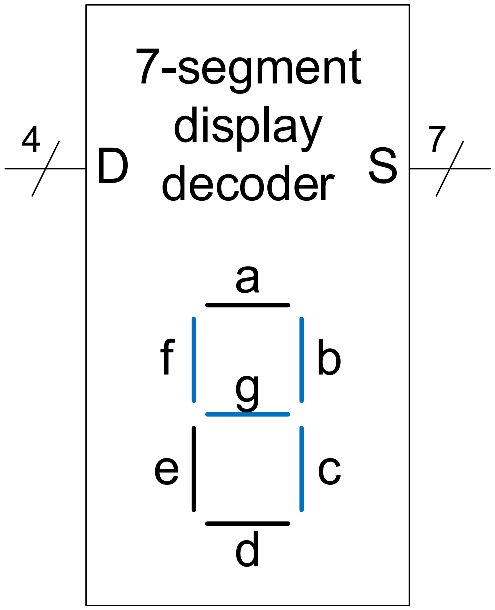
Mux4: Other Implementations

```
module mux4 ...

  /* alternative behavioral implementation using assign statement
  assign y = s[1] ? (s[0] ? d3 : d2) : (s[0] ? d1 : d0); */

  /* alternative behavioral implementation using case statement
  always_comb
    case(s)
      0: y = d0;
      1: y = d1;
      2: y = d2;
      3: y = d3;
    endcase */

  /* alternative behavioral implementation using if statements
  always_comb
    if      (s == 2'b00) y = d0;
    else if (s == 2'b01) y = d1;
    else if (s == 2'b11) y = d2;
    else                 y = d3; */
endmodule
```

Combinational Logic with Always

case statements for truth tables

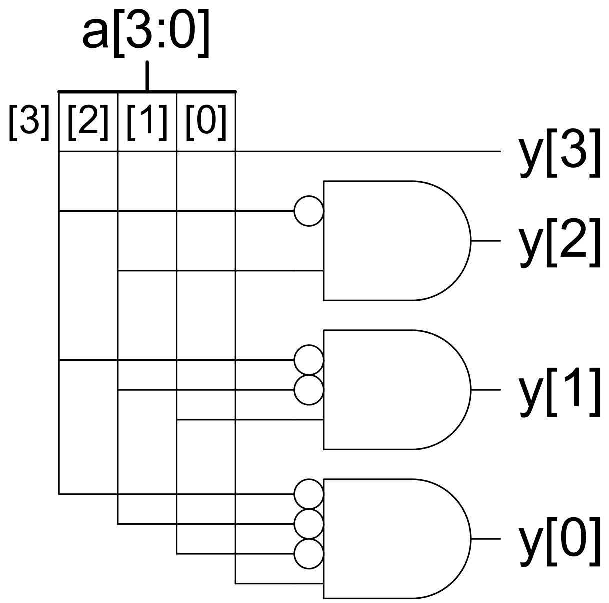
```
module sevenseg(input  logic [3:0] digit,
                output logic [6:0] segments);

  always_comb
    case(digit)
                          // abc_defg
      0:        segments = 7'b111_1110;
      1:        segments = 7'b011_0000;
      2:        segments = 7'b110_1101;
      3:        segments = 7'b111_1001;
      4:        segments = 7'b011_0011;
      5:        segments = 7'b101_1011;
      6:        segments = 7'b101_1111;
      7:        segments = 7'b111_0000;
      8:        segments = 7'b111_1111;
      9:        segments = 7'b111_0011;
      default: segments = 7'b000_0000;
    endcase
endmodule
```



Combinational Logic with Always

casez statements for truth tables with don’t care

```
module priorityckt(input  logic [3:0] a,
                   output logic [3:0] y);

  // casez implementation
  always_comb
    casez(a)
      4'b1???: y = 4'b1000;
      4'b01??: y = 4'b0100;
      4'b001?: y = 4'b0010;
      4'b0001: y = 4'b0001;
      default: y = 4'b0000;
    endcase

  /* if implementation:
  always_comb
    if      (a[3]) y = 4'b1000;
    else if (a[2]) y = 4'b0100;
    else if (a[1]) y = 4'b0010;
    else if (a[0]) y = 4'b0001;
    else           y = 4'b0000; */
endmodule
```



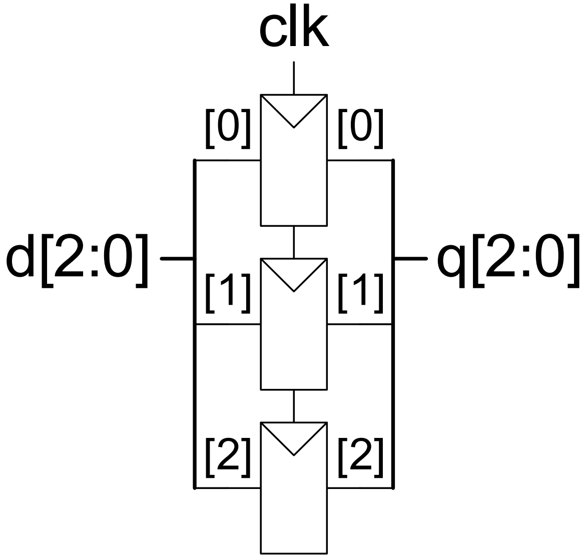
Chapter 4: HDL Design Practices

Sequential Logic: Elements

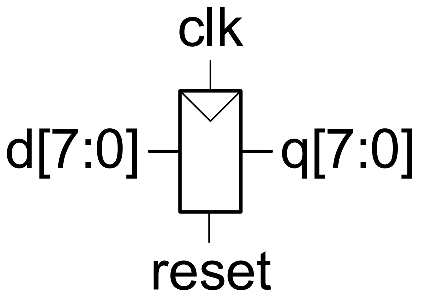
Flip-Flops

On rising edge of clock, copy D to Q

```
module flop #(parameter WIDTH = 3) (
  input  logic             clk,
  input  logic [WIDTH–1:0] d,
  output logic [WIDTH–1:0] q);

  always_ff @(posedge clk)
    q <= #1 d;
endmodule
```



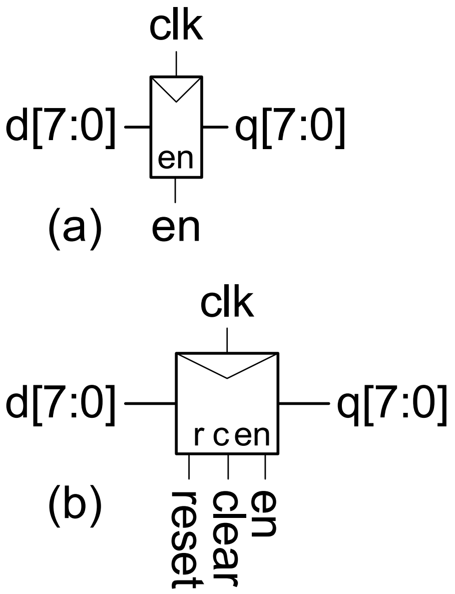
Resettable Flip-Flops



```
module flopr #(parameter WIDTH = 8) (
  input  logic             clk, reset,
  input  logic [WIDTH–1:0] d,
  output logic [WIDTH–1:0] q);

  always_ff @(posedge clk, posedge reset) // asynchronous reset
    if (reset) q <= #1 0;
    else       q <= #1 d;
endmodule

module flopc #(parameter WIDTH = 8) (
  input  logic             clk, clear,
  input  logic [WIDTH–1:0] d,
  output logic [WIDTH–1:0] q);

  always_ff @(posedge clk)
    if (clear) q <= #1 0; // synchronous clear
    else       q <= #1 d;
endmodule
```

Enabled Flip-Flops

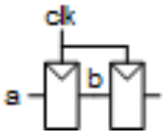
```
module flopen #(parameter WIDTH = 8) (
  input  logic             clk, en,
  input  logic [WIDTH-1:0] d,
  output logic [WIDTH-1:0] q);

  always_ff @(posedge clk)
    if (en) q <= #1 d;
endmodule

module flopenrc #(parameter WIDTH = 8) (
  input  logic             clk, reset, clear, en,
  input  logic [WIDTH-1:0] d,
  output logic [WIDTH-1:0] q);

  always_ff @(posedge clk, posedge reset)
    if (reset)   q <= #1 0;
    else if (en)
      if (clear) q <= #1 0;
      else       q <= #1 d;
endmodule
```



Blocking vs. Nonblocking Assginments

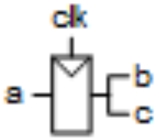
Nonblocking

```
always_ff @(posedge clk) begin
    b <= a;
    c <= b;
end
```



Blocking

```
always_ff @(posedge clk) begin
    b = a;
    c = b;
end
```

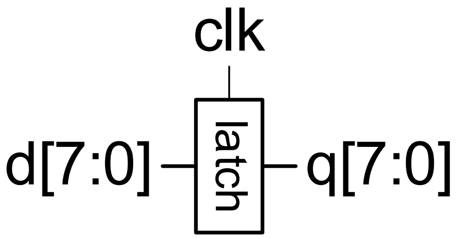


- **Nonblocking** are appropriate for **sequential logic**
- **Blocking** are more efficient for **combinational logic**

Latches

When clk = 1, copy d to q

```
module latch #(parameter WIDTH = 8) (
  input  logic             clk,
  input  logic [WIDTH-1:0] d,
  output logic [WIDTH-1:0] q);

  always_latch
    if (clk) q <= #1 d;
endmodule
```

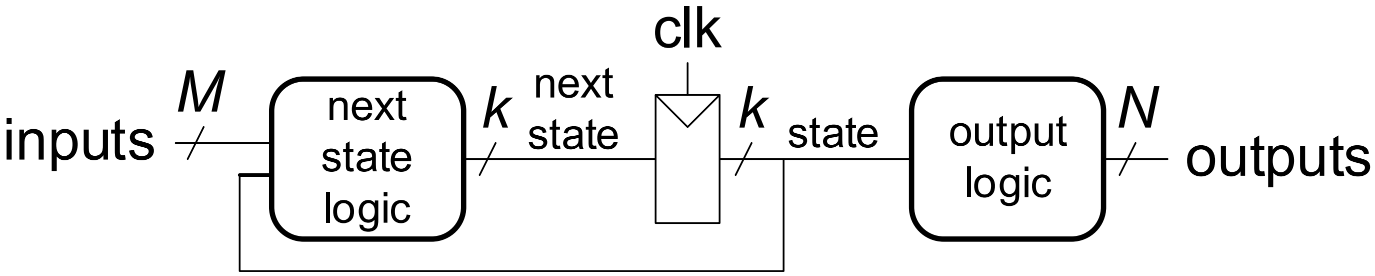


Chapter 4: HDL Design Practices

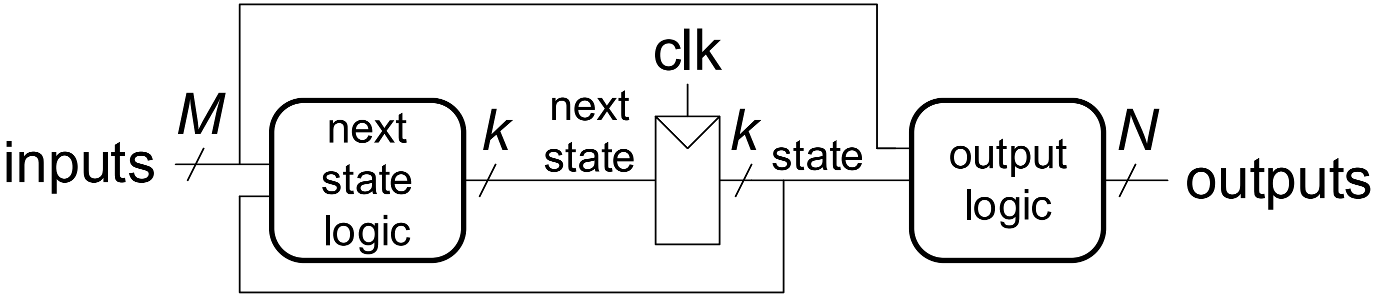
Sequential Logic: FSMs

Finite State Machines

- **Moore:** **Output** depends on **current state**
- **Mealy:** **Output** depends on **current state & inputs**

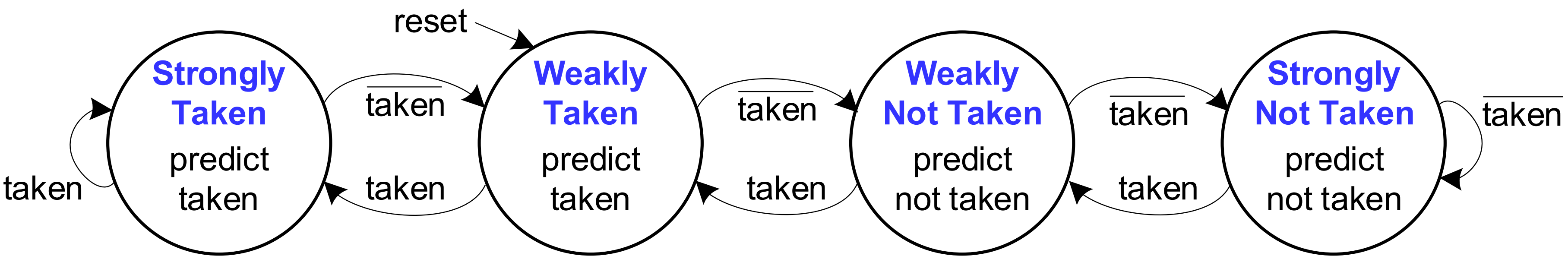


Moore



Mealy

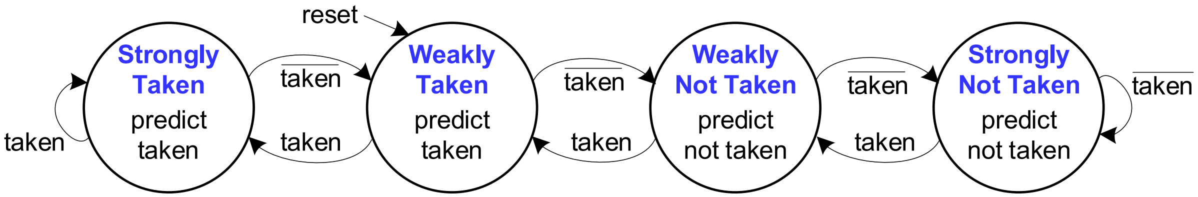
Moore Example: 2-bit branch predictor

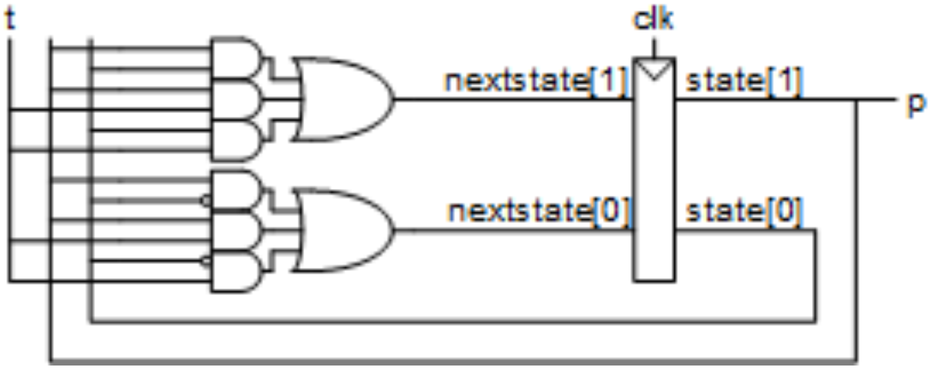


Moore Example: 2-bit branch predictor

```
module twobitbranchpredictor(input  logic clk, reset,
                             input  logic t,
                             output logic p);

  typedef enum logic [1:0] {SNT, WNT, WT, ST} statetype;

  statetype state, nextstate;

  // state register
  always_ff @(posedge clk, posedge reset)
    if (reset) state <= WT;
    else       state <= nextstate;

  // next state logic
  always_comb
    case (state)
      SNT: if (t) nextstate = WNT;
           else   nextstate = SNT;
      WNT: if (t) nextstate = WT;
           else   nextstate = SNT;
      WT:  if (t) nextstate = ST;
           else   nextstate = WNT;
      ST:  if (t) nextstate = ST;
           else   nextstate = WT;
      default:    nextstate = WT;
    endcase

  // output logic
  assign p = (state == WT) | (state == ST);
endmodule
```

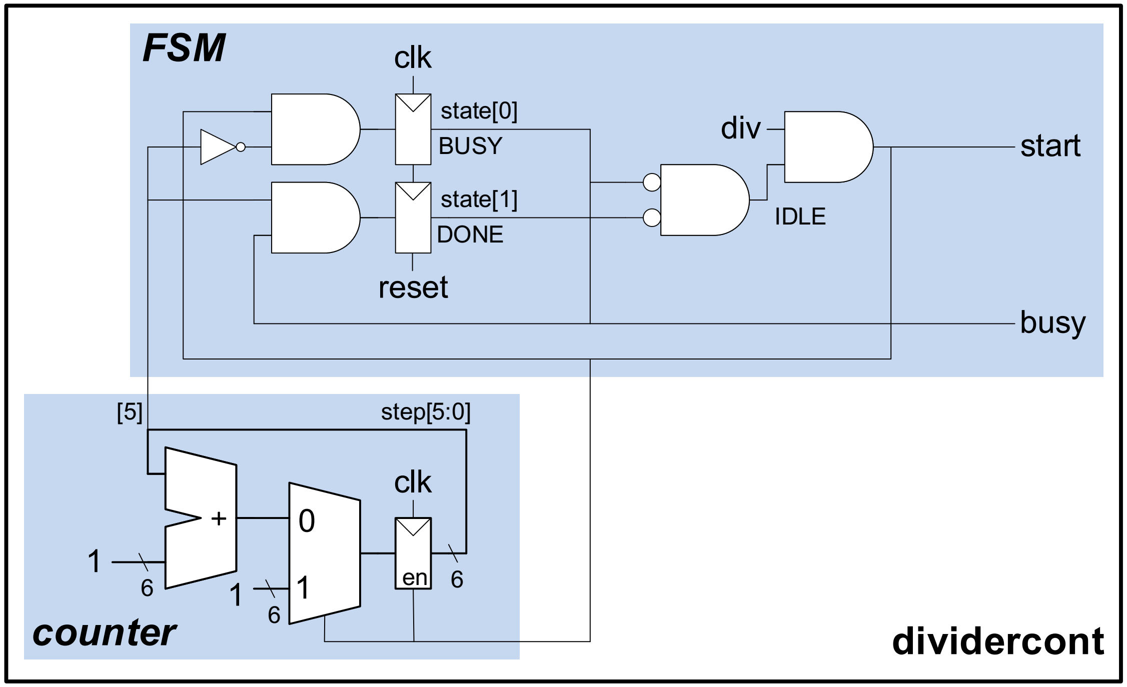
State	Encoding S[1:0]
SNT	00
WNT	01
WT	10
ST	11





Mealy Example: Divider Controller

```
module dividercont (input  logic clk, reset,
                    input  logic div,
                    output logic start, busy);

  typedef enum logic [1:0] {IDLE, BUSY, DONE} statetype;
  statetype state;
  logic [5:0] step;

  always_ff @(posedge clk)
    if (reset) state <= IDLE;
    else case (state)
      IDLE: if (div) begin
              state <= BUSY;
              step  <= 1;
            end
      BUSY: begin
              if (step[5]) state <= DONE;
              step <= step + 1;
            end
      DONE: state <= IDLE;
    endcase

  assign start = div & (state == IDLE);
  assign busy = start | (state == BUSY);
endmodule
```

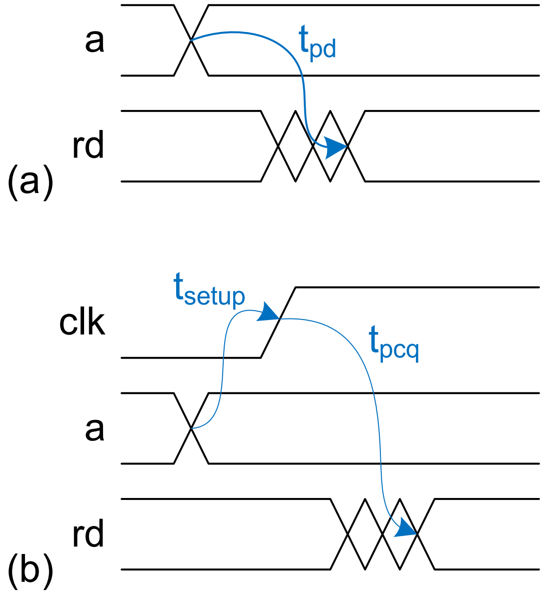


Chapter 4: HDL Design Practices

Sequential Logic: Memories

Memories

- 2^M words x N bits
- Read and write ports
- Optional byte and chip enables
- Synchronous or asynchronous read
- Synchronous write

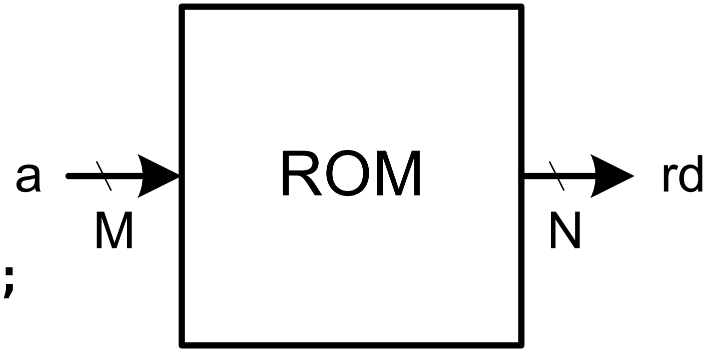


Asynchronous Read

Synchronous Read

ROM: Read Only Memory

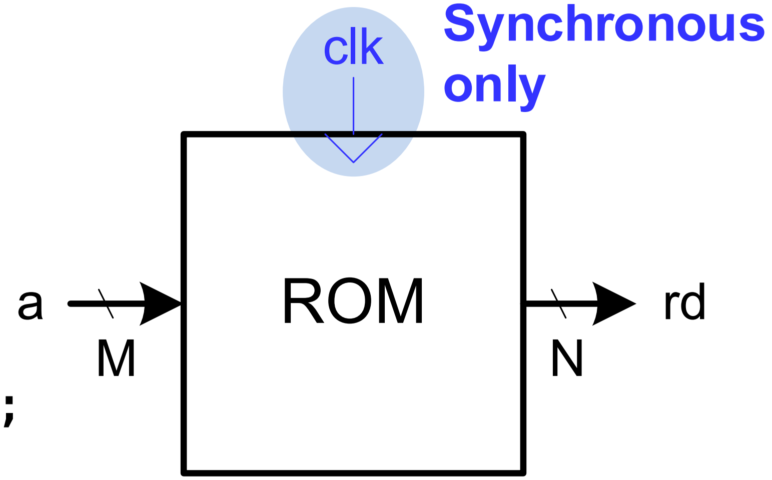
```
module rom1p1r #(parameter M=3, N=16) (
  input  logic [M–1:0] a,
  output logic [N–1:0] rd);

  logic [N–1:0] mem[2**M];

  initial $readmemh("romfile.dat", mem);

  assign rd = mem[a]; // asynchronous read
endmodule
```



```
romfile.dat
0000
FFFF
A5A5
0001
DEAD
BEEF
ABCD
0123
```

ROM: Read Only Memory

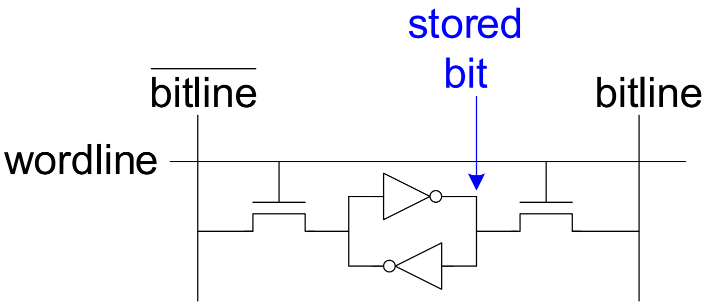
```
module rom1p1r #(parameter M=3, N=16) (
  input  logic [M–1:0] a,
  output logic [N–1:0] rd);

  logic [N–1:0] mem[2**M];

  initial $readmemh("romfile.dat", mem);

  always_ff @(posedge clk)
    rd <= mem[a]; // synchronous read
endmodule
```



Synchronous only

```
romfile.dat
0000
FFFF
A5A5
0001
DEAD
BEEF
ABCD
0123
```

Static Random Access Memory (SRAM)



Read:

Precharge bitline & bitlineb high

Raise wordline

Either bitline or bitlineb will drop

Write:

Drive bitline to write value and bitlineb to its complement

Raise wordline

Bitlines will overpower inverters and store into cell

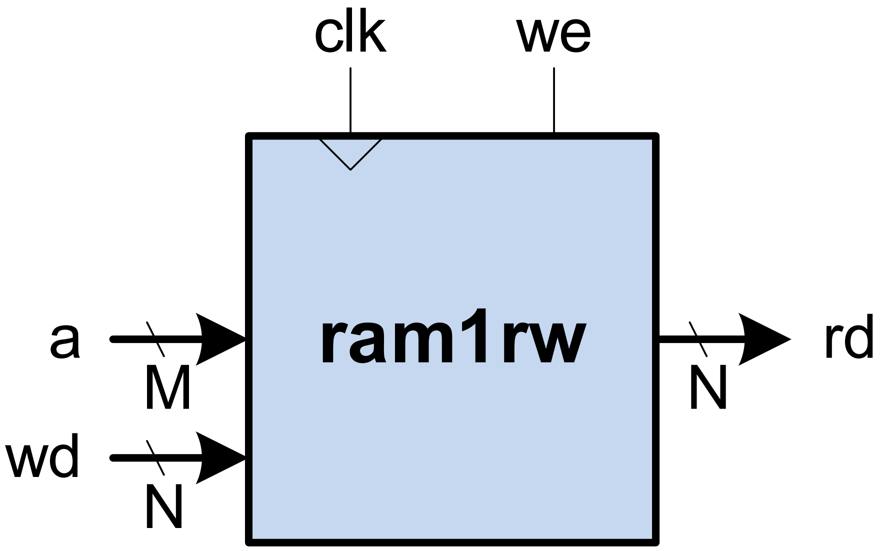
Large SRAMs use SRAM bit cells

- Along with decoders, bitline circuits

Small SRAMs use flip-flops or latches

Single-port SRAM

```
module ram1p1rw_async #(parameter M = 6, N = 32) (
  input  logic         clk,
  input  logic         we,
  input  logic [M-1:0] a,
  input  logic [N-1:0] wd,
  output logic [N-1:0] rd);

  logic [N-1:0] mem[2**M];

  assign rd = mem[a];      // asynchronous read
  always_ff @(posedge clk)
    if (we) mem[a] <= wd;
dndmodule

module ram1p1rw #(parameter M = 6, N = 32) (
  input  logic         clk,
  input  logic         we,
  input  logic [M-1:0] a,
  input  logic [N-1:0] wd,
  output logic [N-1:0] rd);

  logic [N-1:0] mem[2**M];

  always_ff @(posedge clk)  begin
    rd <= mem[a];           // synchronous read
    if (we) mem[a] <= wd;
  end
endmodule
```



SRAM with Byte Enable

Only write byte when its we = 1

```
module ram1p1rwb #(parameter M = 6, N = 32) (
  input  logic           clk,
  input  logic [N/8-1:0] we,
  input  logic [M-1:0]   a,
  input  logic [N-1:0]   wd,
  output logic [N-1:0]   rd);

  logic [N-1:0] mem[2**M];
  genvar i;

 always_ff @(posedge clk)
   rd <= mem[a];

  for(i=0; i<M/8; i++) // byte writes
    always_ff @(posedge clk)
      if (we[i])
        mem[a][i*8+7:i*8] <= wd[i*8+7:i*8];
endmodule
```

SRAM with Chip & Byte Enable

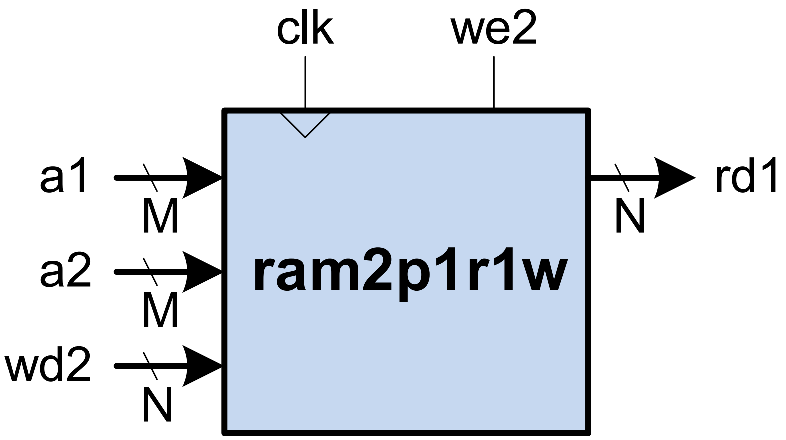
Disable write & retain old output when ce = 0

```
module ram1p1rwbe #(parameter M = 6, N = 32) (
  input  logic           clk,
  input  logic           ce,
  input  logic [N/8-1:0] we,
  input  logic [M-1:0]   a,
  input  logic [N-1:0]   wd,
  output logic [N-1:0]   rd);

  logic [N-1:0] mem[2**M];
  genvar i;

  always_ff @(posedge clk)
    if (ce) rd <= mem[a];

  for(i=0; i<N/8; i++) // byte writes
    always_ff @(posedge clk)
      if (ce & we[i])
         mem[a][i*8+7:i*8] <= wd[i*8+7:i*8];
    end
endmodule
```

Dual-Ported SRAM

2 ports: 1 read and 1 write port

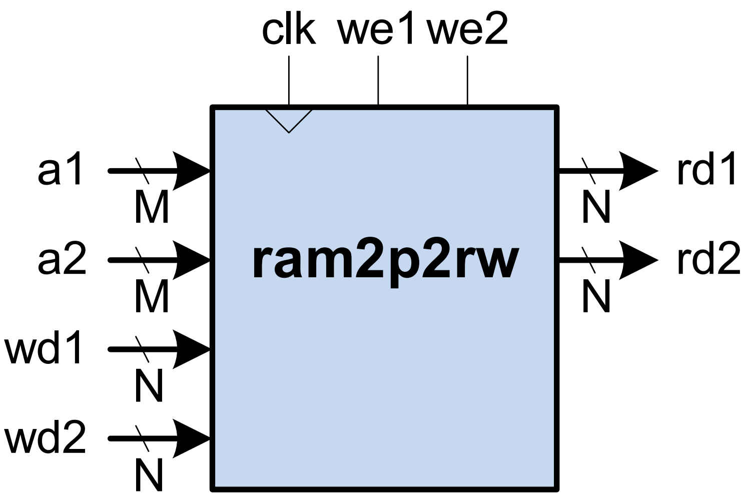
```
module ram2p1r1w #(parameter M = 6, N = 32) (
  input  logic        clk,
  input  logic        we2,
  input  logic [M–1:0] a1, a2,
  input  logic [N–1:0] wd2,
  output logic [N–1:0] rd1);

  logic [N–1:0] mem[2**M];

  always_ff @(posedge clk)  begin
    rd1 <= mem[a1];
    if (we2) mem[a2] <= wd2;
  end
endmodule
```



Dual-Ported SRAM

2 ports: 2 read/write ports

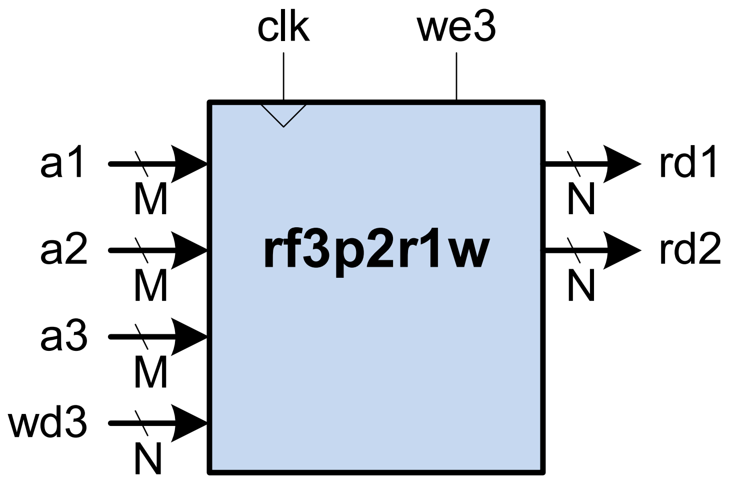
```
module ram2p2rw #(parameter M = 6, N = 32) (
  input  logic          clk,
  input  logic          we1, we2,
  input  logic [M–1:0] a1, a2,
  input  logic [N–1:0] wd1, wd2,
  output logic [N–1:0] rd1, rd2);

  logic [N–1:0] mem[2**M];

  always_ff @(posedge clk)  begin
    rd1 <= mem[a1];
    rd2 <= mem[a2];
    if (we1) mem[a1] <= wd1;
    if (we2) mem[a2] <= wd2;
  end
endmodule
```



Triple-Ported Register File

3 ports: 2 read ports, 1 write port

```
module rf3p2r1w #(parameter M = 5, N = 32) (
  input  logic        clk,
  input  logic        we3,
  input  logic [M-1:0] a1, a2, a3,
  input  logic [N-1:0] wd3,
  output logic [N-1:0] rd1, rd2);

  logic [N-1:0] mem[2**M];

  // asynchronous read on ports 1, 2
  assign rd1 = mem[a1];
  assign rd2 = mem[a2];

  // synchronous write on port 3
  always_ff @(posedge clk)
    if (we3) mem[a3] <= wd3;
endmodule
```



Chapter 4: HDL Design Practices

Verilog Syntax

Constants

```
N'Bvalue
```

- **N** = **size** in bits
- **B** = **base** (b binary, d decimal, h hexadecimal)

Number	Bits	Base	Decimal Value	Stored
3'b101	3	2	5	101
8'b1010_1011	8	2	171	10101011
8'b11	8	2	3	00000011
3'd6	3	10	6	110
8'hAB	8	16	171	10101011

Bit Swizzling

```
// Create 64-bit bus of all 0 or all 1 depending on s
logic [63:0] s64;
assign s64 = {64{s}};

// Create 8-bit bus of all x or all 1s
logic [7:0] data0, data1;
assign data0 = 'x;     // data0 = xxxxxxxx
assign data1 = '1;     // data1 = 11111111

// Join together two 32-bit values to form a 64-bit value
logic [31:0] upper, lower;
logic [63:0] entire;
assign entire = {upper, lower};

// Form bus containing constants and parts of other busses
// y = a[2] a[2] b[2] b[1] b[0] a[1] a[0] 1 0 1
logic [2:0] a, b;
logic [9:0] y;
assign y = {{2{a[2]}}, b, a[1:0], 3'b101};

// Add 2 32-bit numbers: produce 32-bit sum, 1-bit carry out
logic [31:0] a, b, sum;
logic        cout;
assign {cout, sum} = a + b;
```

Gating one bit with many

```
logic         reset;
logic [N-1:0] a, y1, y2;

// equivalent methods
assign y1 = reset ? '0 : a;
assign y2 = {N{~reset}} & a;
```

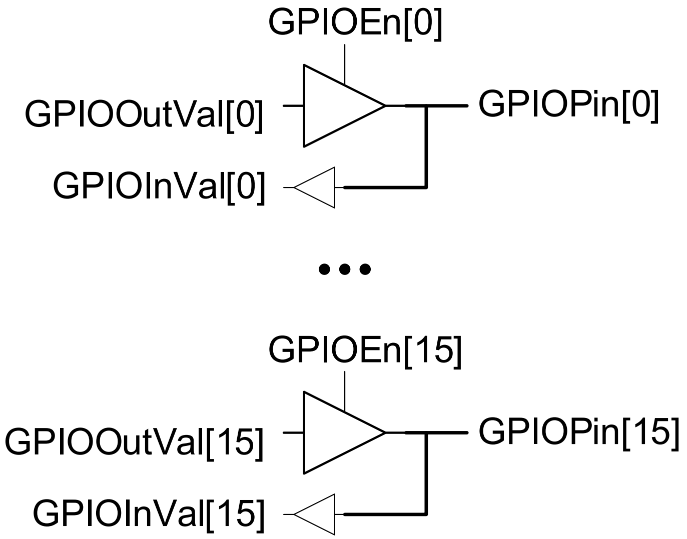
Delays

- **Handy to track causality in simulation**
  - Especially on flip-flop outputs
- **Necessary to create clocks in testbench**
- **Ignored for synthesis**

```
module fulladder(input  logic a, b, c,
                 output logic s, cout);

  assign #3 s = a ^ b ^ c;
  assign #2 cout = (a & b) | (a & c) | (b & c);
endmodule
```

Tristate and Bidirectional I/O

z indicates floating (undriven) value



```
module tristate #(parameter WIDTH=32) (
  input  logic [WIDTH-1:0] a,
  input  logic             en,
  output tri   [WIDTH-1:0] y);

  assign y = en ? a : 'z;
endmodule


module gpio #(parameter WIDTH=16) (
  input  logic [WIDTH-1:0] GPIOOutVal, GPIOEn,
  output logic [WIDTH-1:0] GPIOInVal,
  inout  tri   [WIDTH-1:0] GPIOPin);

  assign GPIOInVal = GPIOPin;
  tristate #(1) ts[WIDTH-1:0](GPIOOutVal, GPIOEn, GPIOPin);
endmodule
```

Port Connections

- `.sig` connects signal with same name
- `.sig(sig2)` connects `sig2` to `sig` port
- `.*` connects all ports
  - **Avoid** because it makes code hard to trace

```
module reg4(input  logic       clk,
            input  logic       reset,
            input  logic [3:0] x,
            output logic [3:0] y);

  flopr #(1) f0(clk, reset, x[0], y[0]);
  flopr #(1) f1(.clk(clk),
                .reset(reset),
                .d(x[1]),
                .q(y[1]));
  flopr #(1) f2(.clk, .reset, .d(x[2]), .q(y[2]));
  flopr #(1) f3(.d(x[3]), .q(y[3]), .*);
endmodule
```

Assertions

Check properties during simulation

```
always_comb
  assert (`U_SUPPORTED | (`S_SUPPORTED == 0))
    else $error ("S supported only if U is too");
```

`$warning` is also available

Chapter 4: HDL Design Practices

Configurable Hardware

`define

Place configuration values in a .vh header file

- `include ‘wally-config.vh’ in each .sv file

```
`define XLEN 32
`define PPN_BITS (`XLEN==32 ? 22 : 44)
```

Configurable Widths

Use {size{val}} to create configurable busses

```
`include “wally-config.vh”

// Create XLEN-bit bus of all 0 or all 1 depending on s
logic [`XLEN-1:0] sx;
assign sx = {`XLEN{s}};

// Sign extend D[31:20] to `XLEN bits
logic [`XLEN-1:0] extend;
assign extend = {{(`XLEN-12){Instr [31]}}, Instr[31:20]};

// LT in the bottom bit and WIDTH-1 0s in the upper bits
logic [`XLEN-1:0] SLT;
assign SLT = {{(`XLEN-1){1'b0}}, LT};
```

Conditional Hardware

Turn off signals based on configuration

- **Example:** `WriteMSTATUSHM` = 0 when `` `XLEN `` is not 32

```
`include “wally-config.vh”
assign WriteMSTATUSHM = CSRMWriteM & (CSRAdrM == MSTATUSH) &
                        (`XLEN==32);
```

Instantiate module based on configuration

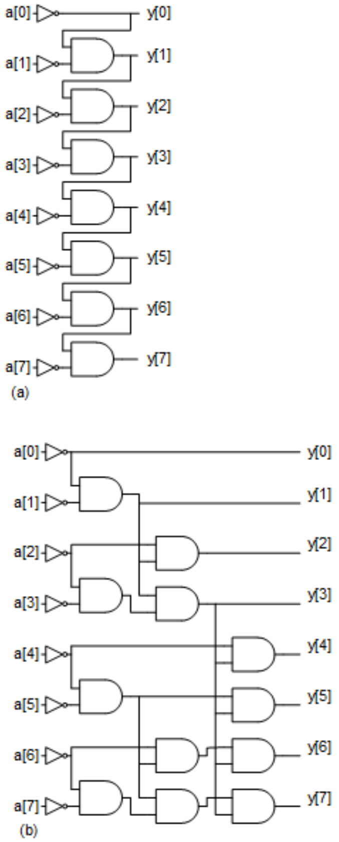
- **Example:** Make `MDU` when `` `M_SUPPORTED ``
- Otherwise tie outputs to 0

```
logic [`XLEN-1:0] MulDivResultW;
logic             DivBusyE;
if (`M_SUPPORTED) begin: mdugen
  mdu mdu(.clk, .reset, .ForwardedSrcAE, .ForwardedSrcBE,
    .Funct3E, .Funct3M, .MulDivE, .W64E,
    .MulDivResultW, .DivBusyE, .StallM, .StallW, .FlushM, .FlushW);
end else begin // no M instructions supported
  assign MulDivResultW = 0;
  assign DivBusyE = 0;
end
```

Generate Loops

Loop to describe each bit

```
module prioritythermometer #(parameter N = 8) (
  input  logic  [N–1:0] a,
  output logic  [N–1:0] y);

  // create thermometer code mask
  genvar i;
  assign y[0] = ~a[0];
  for (i=1; i<N; i++) begin: therm
    assign y[i] = y[i–1] & ~a[i];
  end
endmodule
```



Arrays of Modules

How do **signals** connect to an **array**?

```
module tlbram #(parameter TLB_ENTRIES = 8) (
  input  logic                     clk, reset,
  input  logic [`XLEN-1:0]         PTE,
  input  logic [TLB_ENTRIES-1:0]    Matches, WriteEnables,
  output logic [`PPN_BITS-1:0]     PPN,
  output logic [7:0]               PTEAccessBits,
  output logic [TLB_ENTRIES-1:0]    PTE_Gs
);

  logic [`PPN_BITS+9:0] RamRead[TLB_ENTRIES];
  logic [`PPN_BITS+9:0] PageTableEntry;

  // RAM implemented with array of flops and AND/OR read logic
  tlbramline #(`PPN_BITS+10) tlbramline[TLB_ENTRIES-1:0]
     (.clk, .reset, .re(Matches), .we(WriteEnables),
      .d(PTE[`PPN_BITS+9:0]), .q(RamRead), .PTE_G(PTE_Gs));
  or_rows #(TLB_ENTRIES, `PPN_BITS+10) PTEOr(RamRead, PageTableEntry);

  // Rename the bits read from the TLB RAM
  assign PTEAccessBits = PageTableEntry[7:0];
  assign PPN = PageTableEntry[`PPN_BITS+9:10];
endmodule


module tlbramline #(parameter WIDTH = 22)
  (input  logic               clk, reset,
   input  logic               re, we,
   input  logic [WIDTH-1:0] d,
   output logic [WIDTH-1:0] q,
   output logic               PTE_G);

   logic [WIDTH-1:0] line;

   flopenr #(WIDTH) pteflop(clk, reset, we, d, line);
   assign q = re ? line : 0;
   assign PTE_G = line[5]; // send global bit to CAM as part of ASID matching
endmodule
```

```
module or_rows #(parameter ROWS = 8, COLS=2) (
  input  var logic [COLS-1:0] a[ROWS],
  output logic [COLS-1:0] y);

  genvar row;
  if(ROWS == 1)
    assign y = a[0];
  else begin
    logic [COLS-1:0] mid[ROWS-1:1];
    assign mid[1] = a[0] | a[1];
    for (row=2; row < ROWS; row++)
      assign mid[row] = mid[row-1] | a[row];
    assign y = mid[ROWS-1];
  end
endmodule
```

Chapter 4: HDL Design Practices

Testbenches

Self-checking Testbench

- Instantiate device under test (DUT)
- Generate clock and reset inputs
- Read a set of *testvectors* (inputs and expected outputs) from a file
- Apply the inputs on successive clock cycles and check the outputs
- Log any mismatches

fulladder testbench

```
module testbench();
  logic        clk, reset;
  logic        a, b, c, s, cout, sexpected, coutexpected;
  logic [31:0] vectornum, errors;
  logic [4:0]  testvectors[10000:0];

  // instantiate device under test
  fulladder dut(a, b, c, s, cout);

  // generate clock
  always
    begin
      clk = 1; #5; clk = 0; #5;
    end

  // at start of test, load vectors and pulse reset
  initial
    begin
      $readmemb("fulladder.tv", testvectors);
      vectornum = 0; errors = 0;
      reset = 1; #22; reset = 0;
    end

  // apply test vectors on rising edge of clk
  always @(posedge clk)
    begin
      #1; {a, b, c, coutexpected, sexpected} = testvectors[vectornum];
    end
```

fulladder.tv

```
// abc_cout s
000_00
001_01
010_01
011_10
100_01
101_10
110_10
111_11
```

fulladder testbench (continued)

```
// check results on falling edge of clk
  always @(negedge clk)
    if (~reset) begin // skip during reset
      if (s !== sexpected | cout !== coutexpected) begin  // check result
        $display("Error: inputs = %b", {a, b, c});
        $display("  outputs cout s = %b%b (%b%b expected)",cout, s, coutexpected, sexpected);
        errors = errors + 1;
      end
      vectornum = vectornum + 1;
      if (testvectors[vectornum] === 'bx) begin
        $display("%d tests completed with %d errors",
                      vectornum, errors);
        $stop;
      end
    end
endmodule
```

Use === or !== for Signals that could be x or z

Chapter 4: HDL Design Practices

Wally

Style

Guidelines

Signals

- Use **flatcase** for `clk` and `reset`.
- Use **UpperCamelCase** for signals.
- Use **suffixes** on all signals associated with pipeline stages.
  - F, D, E, M, W
- Use **prefixes** to distinguish related signals from different sources
  - IEUResultM, MDUResultW, SCResultW

- **Declare all internal signals**, including 1-bit signals.
- Only use (bidirectional) **inout pins at the top-le**vel chip module.
  - Define them as `inout tri`.
- Declare all other signals as **`logic`**.
- Declare busses as **little-endian**
  - bus[31:0], not bus[0:31]
- **Avoid passing arrays through ports** if possible, but declare the port as var logic if necessary.
  - `input var logic [7:0] PMPCFG_ARRAY_REGW[`PMP_ENTRIES-1:0]`

Port Connections

- **Connect ports by order for simple generic modules** such as flip-flops and multiplexers.
- Guideline 3.12 **Connect ports by name** for other modules.
- Guideline 3.13 For non-generic modules used only once, **use the same name** inside and outside of the module.
- Guideline 3.14 **Avoid .*** notation except in wrapper modules
- Guideline 3.15 Pack **multiple port connections on one line** of the instantiation.

Example:

```
mux2 #(`XLEN) ieuresultmux(ALUResultE, AltResultE, ALUResultSrcE,
                            IEUResultE);
mmu #(.TLB_ENTRIES(`DTLB_ENTRIES), .IMMU(0)) dmmu(
   .clk, .reset, .SATP_REGW, .STATUS_MXR, .STATUS_SUM,
   .STATUS_MPRV, .STATUS_MPP,
   .PrivilegeModeW, .DisableTranslation(SelHPTW),
   .PAdr(PreLSUPAdrM),
   .VAdr(IEUAdrM),
   .PMPCFG_ARRAY_REGW, .PMPADDR_ARRAY_REGW);
```

Spacing and Indentation

- **Indent by two spaces** for each level of nesting.
- **Indent using spaces** instead of tabs.
- **Align signal declarations** within a block.
- Leave a **space around operators** and **after if/case/for** statements.
- **Avoid space before** semicolons, commas, closing parentheses, and open parentheses in module instantiations, and inside bus width declarations.
- **Only use begin and end** when more than one statement is required in a block.
- Pack the **begin statements on the same line** where feasible.
- Limit lines to a maximum of **95 characters**
- Use **blank lines sparingly** to logically separate portions of code.
- Use **banner comments sparingly** to separate major portions of code in a complex module when using hierarchy isn't a good alternative.

Synchronous Design

Use synchronous sequential design:

- Every component should be: a positive edge-triggered flip-flop, a standard memory, or combinational logic.
- All components share common clock called clk.
- Avoid latches, negative edge-triggered flip-flops, and asynchronous components.
- Exceptions should only occur when essential (e.g., at an interface with an asynchronous peripheral)

Instantiate memories and flip-flops using modules:

- Facilitates identifying # of state bits
- Provides a consistent built-in delay for simulation
- Enables swapping out synchronous/asynchronous reset, scan, or low-power features
- **Exception:** FSM state registers use enumerated state types

Modules

- Each module in its own file; **file name = module name**
- Module **name** immediately folllowed by “(“
- Input and output ports **start on same line** as module name - unless parameterized
- **For parameterized modules:**
  - “(“ and parameters go on same line as module declaration.
  - Input and output ports go on subsequent lines, indented by two spaces
  - Place the closing); on the same line with the last port declaration.
- **One input or output keyword on a single line** in module declaration; ports with the same polarity (input or output) and width are declared on the same line
- **Align text horizontally**

System

- Use **behavioral** coding style for **leaf** cells including generic blocks and control blocks.
- Use **structural** coding style for **datapath** blocks.
- Use **hierarchy** so that modules remain small and easy to understand.
- Avoid programming constructs that obscure the hardware being implied.

Configurable Hardware

- Define **generic configurable** modules where appropriate.
- Use **`` `define ``** to define **global parameters** in a `wally-config.vh` file included by all modules.
- Use **`#parameter`** to define **parameters for individual modules.**
- **Module interfaces should remain the same** whether or not a feature is enabled.
- **Omit** the optional **`generate/endgenerate`** keywords around conditional generates and generate loops.
- When a **generate block** contains instance or signal declarations, **name the block with a colon** after the begin keyword.

Comments

- **Place a banner comment at the beginning of each file.**
  - Filename, authors name and email, date. This gives you copyright.
  - Purpose of module
  - License (Wally uses SolderPad license, an Apache derivative).
- **Use comments to explain the big picture rather than restate what the code is doing.**

Bad Example:

```
assign ab = ~a; // flip the bits
assign minusa = ab + 1; // add one
```

Good example:

```
// Take the 2’s complement to negate a
assign minusa = ~a + 1;
```

About these Notes

RISC-V System-on-Chip Design Lecture Notes