Recall What You Will Do

- Design and implement a serial MIDI receiver
  - Hardware in an Altera Complex Programmable Logic Device (CPLD) MAX 7000S (part number EPM7064SLC44-10)
  - Using ALTERA Quartus II software tools for synthesis
  - Debug - functional simulation (wave forms)
  - Debug of board - logic analyzer

- Coding in Verilog
- Next we look at Verilog design issues
Verilog

- Hardware Description Language (HDL)
  - Verilog one of the main HDLs, VHDL is the other
- A way to describe digital **hardware**
  - Don’t think of Verilog as programming language
- Can describe hardware…
  - Structurally -- as network of interconnected components
  - Behaviorally – as logical functions, more abstract, but eventually gets mapped to some structure anyways
- Why do HDLs exist?
  - To manage complexity and facilitate design automation
  - So that designs can be simulated before being built
- Synthesizable and non-synthesizable parts
- Why does an HDL include parts that cannot be synthesized to hardware?
Translating Abstract Designs to Hardware

- Identify hardware functionality that is needed
- Divide and conquer
  - Break into smaller ‘black-boxes’ when complicated
  - Think also about performance – what you do in a clock period
- Focus on the heart of the problem first
- Stub-out all (or majority of) modules
  - List inputs, outputs
  - Write comments - how outputs can be generated from inputs
- Implement one by one
  - Control-first design is intuitive for ordering your work
    - FSMs, state-based outputs, output generation logic
    - Verification
- Instantiate and wire modules together in top module
Basics of Verilog
Basics

- **Constants**
  - Binary ’b // e.g. 3’b000
  - Octal ’o // rarely used
  - Decimal ’d
  - Hexadecimal ’h // e.g. 8’hFF

- **Example:**
  - V = 8’b1011, V = 00001011
  - V = ‘h12a how many bits in V?

- **Verilog predefined logic values**
  - 0 - represents number zero, logic zero, logical false
  - 1 - represents number one, logic one, logical true
  - x - represents an unknown logic value ; driven unknown (in sim)
  - z - represents high impedance logic value ; undriven
Basics

- **Operations**
  - Pretty much follow same syntax as C
  - \( X = a + b \); add
  - \( X = a - b \); subtract
  - \( X = a \times b \); multiply
  - \( X = a^{**}b \); power
  - \( X = a/b \); division
  - \( X = a \% b \); modulo

- **Arrays**
  - `reg[31:0] mema[0:4095];` is a typical memory, 4096 words of 32 bits
  - `wire[31:0] a;` declares a to be 32 wires. just 'a' is all 32 wires, 
    \( a[31] \) is the most significant bit \( a[0] \) is the least significant bit 
    \( a[15:8] \) is eight wires, bits 15 to 8 of 'a'
  - `wire[0:31] a; a[0]` is the most significant bit \( a[31] \) is the least significant bit
Relational Operators

- greater-than (>)
- less-than (<)
- greater-than-or-equal-to (>=)
- less-than-or-equal-to (<=)

Relational operators return logical 1 if expression is true, 0 if false

; let a = 4, b = 3, and...
; x = 4'b1010, y = 4'b1101, z = 4'b1xxx
- a <= b ; evaluates to logical zero
- a > b ; evaluates to logical one
- y >= x ; evaluates to logical 1
- y < z ; evaluates to x
Bitwise and Logic Manipulations

- \( c = a \& b; \) // bitwise AND
- \( c = a \mid b; \) // bitwise OR
- \( c = a \^ b; \) // bitwise XOR
- \( b = \& a; \) // AND of all bits of \( a \)
- \( b = \mid a; \) // OR of all bits of \( a \)
- \( b = ^a; \) // XOR of all bits of \( a \) (i.e. parity)
- \( c = a \&\& b \) // logical AND
- \( c = a \mid\mid b; \) // logical OR
- etc

```vhdl
reg [3:0] a = 4'b0101
reg [3:0] b = 4'b1000
reg [3:0] c;
c = a\mid b  // c = 4'b1101
c = a\mid\mid b  // c = 4'b0001
```
Miscellaneous Manipulations

- \( c = \{a,b\} \);  \\
  // concatenation
- \( c[7:0] = a[8:1] \);  \\
  // select group of bits
- \( d = \{d[6:0],d[7]\} \)  \\
- \( d = \{d[7:1],d[0]\} \)
Combination vs Sequential
Wires and Regs

- Wires are connections (including some combinational logic)
- Regs are combinational or sequential logic, depending on use
  - A reg is a variable, capable of holding state
  - Reg is only *sometimes* a register
- Either can be used on RHS of blocking or nonblocking assignment

```vhdl
assign wire = {wire|reg}  // Wire as combinational signal

always @(posedge clk)  // Reg as sequential element
  reg <= {wire|reg}  // Non-blocking assignment

always @(*)  // Reg as combinational element
  reg = {wire|reg}  // Blocking assignment
```

Make sure you understand these well
Blocking and Non-blocking Assignment

- `begin` and `end` keywords are used to group together assignments under a single condition
- Can be omitted if there is only one assignment under condition

```verilog
always @(posedge clk) begin
  reg <= {wire|reg};   //Non-blocking assignment
  reg <= {wire|reg};   //updates are simultaneous
end

always @(*) begin
  reg = {wire|reg};    //blocking assignment
  reg = {wire|reg};    //updates not simultaneous
end
```

Make sure you understand these well
Sensitivity List

- Every always block has a sensitivity list
- This list describes trigger events that will cause the logic inside the block to be executed
- For this project, suffices to only use ‘*’ and ‘posedge clk'

```verilog
always @({sensitivity list})
always @(posedge clk) //Do this only at clock edge
    reg <= {wire|reg} //Reg as sequential element
        (Non-blocking assignment)
always @(*) //Reg as combinational element
    reg = {wire|reg} (blocking assignment)
```
Using an Always Block

- Sensitivity list indicates when a value may change
- What happens the rest of the time?
  - Value stays unchanged (i.e. holds its state)
- Only regs can hold state
- So, LHS is always block must be reg
- always @(*) often implies combinational logic, but since it uses an always block, regs still must be used on LHS
Using Wires and Regs

wire w;
assign w = a;
assign w = a&b;

always @(posedge clk)
  w = a  // illegal

always @(a)
  w = a  // illegal

reg r;

always @(posedge clk)
  r <= a

always @(*)
  r = a & b

always @(*)
  r <= a & b  //??
Non-Synthesizable Verilog

- Delays
  \[\text{foo} = \#10 \text{ 8'hFF} \; // \text{set value in 10 time units}\]

- Clock
  \[
  \text{always}
  \#2 \text{ clk} = \sim\text{clk}; \; //\text{toggle every 2 time units}
  \]

- Initialize values
  \[
  \text{initial begin}
  \text{reset} = 1'b0;
  \end
  \]

- Simulation outputs
  \[
  \text{initial begin}
  // \text{print verilog change dump}
  // \text{can open vcd file in waveform viewer}
  \$\text{dumpfile( "dump.vcd" );}
  \end
  \]
Uninitialized Values

- Unassigned wire has initial value of z (high impedance)
- Uninitialized reg has a value of x (unknown logic state)
- What are values of r1 and r2 in the example below?

```verilog
module foo;
    reg    r1, r2;
    wire       w;

    always
        #2 r1 = ~r1;
    always
        #2 r2 = ~r2;

    initial begin
        $monitor("r1=%b r2=%b w=%b",r1,r2,w); //print
        r1 = 0;
        #10;
        $finish;
    end
endmodule
```
Use of Modules
Modules

- Declaration is a definition of module
- Instantiation is a use of module
- Connect modules together using wires
- Note that Verilog is flexible about how to write this, so you may see different formats with equivalent meaning

```verilog
module fa( s,cout,a,b,cin);
    output   s,cout;
    input    a,b,cin;
    wire     a,b,cin,s,cout;
    wire     s1,c1,c2;

    ha HA1(s1,c1,a,b);
    ha HA2(s,c2,s1,cin);
    assign cout = c1 | c2;
endmodule
```
Hierarchy of Modules

- Module instances can instantiate other modules within them
- Ultimately, the design is an instance of some top level module
- Top level modules can usually be inferred automatically from the Verilog code
  - It is the one that instantiates other modules and is not instantiated by any modules (except for a test bench)

```verilog
module fa( s,cout,a,b,cin);
  output   s,cout;
  input    a,b,cin;
  wire     a,b,cin,s,cout;
  wire     s1,c1,c2;

  ha HA1(s1,c1,a,b);
  ha HA2(s,c2,s1,cin);
  assign cout = c1 | c2;
endmodule
```
Module Hierarchy

- How many half adders in top module?

```verilog
module top();
    wire [2:0] a;
    wire [2:0] b;
    wire [1:0] c;
    wire [3:0] s;
    fa fa0(s[0],c[0],a[0],b[0],1' b0);
    fa fa1(s[1],c[1],a[1],b[1],c[0]);
    fa fa2(s[2],s[3],a[2],b[2],c[1]);
endmodule

module fa( s,cout,a,b,cin);
    output s,cout;
    input a,b,cin;
    wire a,b,cin,s,cout;
    wire s1,c1,c2;

    ha HA1(s1,c1,a,b);
    ha HA2(s,c2,s1,cin);
    assign cout = c1 | c2;
endmodule
```
Wires and Regs in Module Ports

- **In module declaration**
  - Input ports must be wires
  - Output ports can be regs or wires

  ```
  module foo(  i, o);
  input  i;
  output o;
  wire i;
  {wire|reg} o;
  //something here
  endmodule
  ```

- **In module instantiation**
  - Wires and regs can both connect to input ports
  - Output ports must connect to wires

  ```
  wire b;
  {wire|reg} a;
  foo fooinst (a,b);
  //something here
  ```
module adder( input [31:0] a,     // a input
input [31:0] b,     // b input
input        cin,   // carry-in
output [31:0] sum,   // sum output
output        cout);  // carry-out

assign {cout, sum} = a + b + cin;
endmodule // adder

module fa( s,cout,a,b,cin);

output   s,cout;
input   a,b,cin;
wire   a,b,cin,s,cout;
wire   s1,c1,c2;

ha HA1(s1,c1,a,b);
ha HA2(s,c2,s1,cin);
assign cout = c1 | c2;
endmodule
Which of the following are legal? reasonable?

module ha0( s,c,a,b );
  output s,c;
  input a,b;
  wire a,b;
  wire s,c;
  assign s = a ^ b;
  assign c = a & b;
endmodule

module ha1( s,c,a,b );
  output s,c;
  input a,b;
  wire a,b;
  reg s,c;
  always @(*) begin
    s = a ^ b;
    c = a & b;
  end
endmodule

module ha2( s,c,a,b );
  output s,c;
  input a,b;
  wire a,b;
  reg s,c;
  always @(*) begin
    s <= a ^ b;
    c <= a & b;
  end
endmodule

module ha3( s,c,a,b );
  output s,c;
  input a,b;
  wire a,b;
  wire s,c;
  always @(*) begin
    s = a ^ b;
    c = a & b;
  end
endmodule
Implicit and Explicit port connections

module top();
    wire [1:0] a;
    wire [1:0] b;
    wire      c;
    wire [2:0] s;
    fa fa0(
        .s( s[0]),
        .cout( c   ),
        .a( a[0]),
        .b( b[0]),
        .cin( 1'b0   )    );
    fa fa1(
        s[1],
        s[2],
        a[1],
        b[1],
        c    );
endmodule

module fa( s,cout,a,b,cin);
    output   s,cout;
    input    a,b,cin;
    wire     a,b,cin,s,cout;
    wire     s1,c1,c2;
    ha HA1(s1,c1,a,b);
    ha HA2(s,c2,s1,cin);
    assign cout = c1 | c2;
endmodule
Examples
Blocking vs non-blocking assignment

- Initial value of $a=1$ and $b=2$
- What are the values of $a$ and $b$ on future clock cycles?

```verilog
always @ (posedge clock) begin
    a = b; // blocking assignment
    b = a;
end

always @ (posedge clock) begin
    a <= b; // nonblocking assignment
    b <= a;
end
```
Blocking vs. Non-blocking

initial begin
  a = #10 1'b1 ; assigns 1 to a at time 10
  b = #20 1'b0 ; assigns 0 to b at time 30
  c = #40 1'b1 ; assigns 1 to c at time 70
end

initial begin
  a <= #10 1'b1 ; assigns 1 to a at time 10
  b <= #20 1'b0 ; assigns 0 to b at time 20
  c <= #40 1'b1 ; assigns 1 to c at time 40
end
Example - Shift-Register in Verilog

Incorrect implementation
always @(posedge clk) begin
  shift_reg[1] = shift_reg[2];
  shift_reg[0] = shift_reg[1];
end

- ‘=’ : Blocking Assignment
- Value in shift_reg[3] will be assigned to shift_reg[0] directly

Correct implementation
always @(posedge clk) begin
  shift_reg[1] <= shift_reg[2];
  shift_reg[0] <= shift_reg[1];
end

- ‘<=’ : Non-Blocking Assignment
- Updating will happen after capturing all right-side register values
Coding Style

- Many considerations like the quality of expected/resulting synthesis but also ease of debugging
- **A good (and required for lab) convention is to separate combinational and sequential blocks entirely**
  - No combinational code in the sequential block!
  - Sequential block has only assignments to registers on clock edges or resets!
    - e.g.,
      - state <= state_nxt
      - signal <= signal_nxt
  - This keeps your code easy to read and debug and avoids subtle flaws
Combinational and Sequential Blocks

- Combinational
  - Generate signals inside a clock period
  - E.g., the next state that you will assign to a register
  - Either wires
    ```
    assign foo_next = bar;
    ```
  - Or regs
    ```
    always @(*)
    foo_next = bar;
    ```

- Sequential
  - Latch signal values on clock edges
    ```
    always @(posedge clk)
    foo <= foo_next;
    ```
Mealy vs. Moore State Machines

- **Mealy** - “event driven”
  - Next-state and Output depend on both current state and input

- **Moore** - “state driven”
  - Next-state depends on both current state and input
  - Output depends **only** on current state
module fsm_moore(  output [1:0] out,
    input clk,
    input s1,
    input s2,
    input rst_n );
reg [1:0]  state, state_nxt;
assign out = state;
always @(posedge clk) begin
    if (!rst_n) state <= 2'b00;
    else state <= state_nxt;
end
always @(*) begin
    case(state)
    2'b00: begin
        if (s1 == 1'b1) state_nxt = 2'b01;
        else state_nxt = 2'b00;
    end
    2'b01: begin
        if (s2 == 1'b1) state_nxt = 2'b10;
        else state_nxt = 2'b01;
    end
    2'b10:  state_nxt = 2'b00;
    default: state_nxt = 2'b00;
    endcase
end
endmodule
Moore State Machine

Quartus: Tools → Netlist Viewer → RTL viewer
Moore State Machine

Quartus: Tools → Netlist Viewer → State machine viewer
Moore State Machine

Quartus: simulation results
Verilog Simulation

- Testbenches (usually)
  - Verilog code with non-synthesizable constructs to exercise your logic
  - Written as module with no inputs
  - Instantiates a top level design, and applies inputs at different times

```verilog
module tb;
    reg        a,b,ci;
    wire       s,co;

    fa inst ( .sum(s),
              .cout(co),
              .ina(a),
              .inb(b),
              .cin(ci) );

    initial begin
        $monitor("time=",$time,"%b %b %b %b %b",a,b,ci,s,co);
        a=0;  b=0;  ci=0;
        #1;
        a=0;  b=1;  ci=0;
        #1;
        a=0;  b=1;  ci=0;
        $finish;
    end
endmodule
```
Summary

- Partition into modules and plan out design before implementing
- Stub out all inputs and outputs and comment

- **Separate combinational blocks from sequential block**
  - FSM is implemented in combinational block
    - Next state is calculated in combinational block
    - Output is calculated in combinational block
  - Sequential block contains simple latching assignments

- Make sure you use non-blocking statements in sequential and blocking in combinational blocks

- Use intuitive names (signal, signal_nxt) and follow convention
- Test each module independently, and then test entire design

- **Remember this is hardware not software**