What You Will Do In Lab 4

- 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 (waveforms)
  - 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 = ‘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 * b ;` multiply
  - `X = a**b ;` power
  - `X = a/b ;` division
  - `X = a % b ;` modulo

### Arrays
- `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
- `reg[31:0] mema[0:4095];` is a typical memory, 4096 words of 32 bits.
  - `mema[0]` is the zeroth 32-bit word from memory.
  - `mema[0][31]` is the MSB of the zeroth 32-bit word.
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...
- ; c = 4'b0010, d = 4'b1101, e = 4'b1xxx
  - a <= b ; evaluates to logical 0
  - a > b ; evaluates to logical 1
  - d >= c ; evaluates to logical 1
  - d < e ; evaluates to x
  - c < e ; evaluates to logical 1
Bitwise and Logic Manipulations

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

```
reg [3:0] a = 4'b0101
reg [3:0] b = 4'b1000
reg [3:0] c;
```
Miscellaneous Manipulations

- \( c[3:0] = a[4:1]; \) // select group of bits

- \( c[3:0] = \{a[3:1], a[0]\} \) // concatenation that does nothing

- \( c[3:0] = \{a[2:0], a[3]\} \) // concatenation that reorders

- \( c = \{a[1:0], a[1:0]\}; \)
### Miscellaneous Manipulations

- **c[3:0] = a[4:1];**  // select group of bits
  
  ![Diagram](a4 a3 a2 a1 a0)

- **c[3:0] = {a[3:1],a[0]}**  // concatenation that does nothing
  
  ![Diagram](a3 a2 a1 a0)

- **c[3:0] = {a[2:0],a[3]}**  // concatenation that reorders
  
  ![Diagram](a2 a1 a0 a3)

- **c = {a[1:0],a[1:0]};**
  
  ![Diagram](a1 a0 a1 a0)
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 <= {wire|reg}       (Non-blocking assignment)

always @(*)
  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  //Non-blocking assignment
  reg <= {wire|reg};        //updates are simultaneous
  reg <= {wire|reg};
end

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

Make sure you understand these well
Sensitivity List

- Every always block has a sensitivity list
- This list describes trigger events that cause logic inside block to be executed
  - What happens in block when not triggered?
    - Values remain unchanged (i.e. hold state)
    - Only regs can hold state, so only regs can be on LHS in always block
    - But regs are combinational if they never have to hold state
- 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)
```
Ordering of statements should not matter

- In non-blocking assignments, ordering doesn’t matter because all updates are synchronous (on posedge clk)

```verilog
always @(posedge clk) begin
    a <= b; // nonblocking
    b <= a;
end
```

- In blocking assignments, ordering doesn’t matter because the statements are being evaluated all the time

```verilog
always @(*) begin
    a = d & b; // blocking
    b = c & e;
end
```

- If keeping combinational and sequential logic separate, order of statements won’t matter. Order of statements can matter if you follow bad practices
Using Wires and Regs

wire w1, w2, w3, w4;

assign w1 = a;

assign w2 = a & b;

always @(posedge clk)
  w3 = a

always @(*)
  w4 = a

reg r1, r2, r3;

always @(posedge clk)
  r1 <= a

always @(*)
  r2 = a & b

always @(*)
  r3 <= a & b
Using Wires and Regs

```verilog
wire w1, w2, w3, w4;

assign w1 = a;
assign w2 = a & b;

always @(posedge clk)
  w3 = a

always @(*)
  w4 = a & b
```

```verilog
reg r1, r2, r3;

always @(posedge clk)
  r1 = a

always @(*)
  r2 = a & b

always @(*)
  r3 <= a & b
```
Non-Synthesizable Verilog (don’t use in lab)

- Delays
  
  ```verilog
  foo = #10 8’hFF // set value in 10 time units
  ```

- Clock
  
  ```verilog
  always
  #2 clk = ~clk; // toggle every 2 time units
  ```

- Initialize values
  
  ```verilog
  initial begin
  reset = 1'b0;
  end
  ```

- Simulation outputs
  
  ```verilog
  initial begin
  // print verilog change dump
  // can open vcd file in waveform viewer
  $dumpfile( "dump.vcd" );
  end
  ```

Instead, use external clock in lab

Instead, use reset input to initialize

instead, waveform viewer and logic analyzer
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;

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

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

- Modules break up design and allow for replication of common blocks in a large design.
- Declaration is a definition of module.
- Creates a part that can be used to build bigger modules.
- Each module can contain sequential and combinational logic, and can contain other modules.

```verbatim
module ha( s,c,a,b );
    output s,c;
    input a,b;
    wire a,b,s,c;
    assign s = a ^ b;
    assign c = a & b;
endmodule
```
Modules

- Instantiation creates a named instance of that module in top level design or declaration of another module
- Connect instantiated 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 that gets placed onto chip is an instance of some top level module
  - the inputs and outputs of the module become the pins of the chip
- Top level modules can usually be inferred automatically from the Verilog code
  - top level module is the one that instantiates other modules and is not instantiated by any other modules

```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 ha( s, c, a, b );
  output s, c;
  input a, b;
  wire a, b, s, c;
  assign s = a ^ b;
  assign c = a & b;
endmodule
```
Module Hierarchy

- How many half adders in foo module?

```verilog
module foo();
    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
```
### Modules

```verilog
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

- Explicit port connections give the name of both the port and the wire connecting to them.
- Implicit port connections give only the name of the wire and use order to infer which port they connect to.
- Explicit is more verbose and can be useful for avoiding ordering bugs.
- Either will be fine for lab.

```verilog
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(    s[1]),
        .s[2],
        .a[1],
        .b[1],
        .c    );
endmodule
```

```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
```
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
```
Example - Shift-Register in Verilog

Incorrect implementation

```verilog
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

```verilog
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
Combinational and Sequential Blocks

- Combinational
  - Signals that are changing all the time 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;
    ```
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**
- Sequential block has **only** assignments to registers based on next states from combinational logic or initial values when reset goes low (see example below)
  - This keeps your code easy to read and debug and avoids subtle flaws

```verilog
always @(posedge clk) begin
    if (!rst_n) begin
        // reset things here
        state <= 0;
        foo   <= 1;
    end
    else begin
        // next state from comb.
        state <= state_nxt;
        foo   <= foo_nxt;
    end
end
```
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
Moore State Machine

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
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
endmodule

Note the use of else conditions and default cases in FSM definition. Without this, the synthesis tools may infer extra state bits to hold state when nothing is assigned to state_nxt.
Moore State Machine

Quartus: simulation results
Verilog Simulation

- Testbenches (usually — but not in this lab)
  - 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=0;  ci=1;
        #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