Announcements

- Lab 2, Part 1 is up
  - Due Friday, Oct 17 at NOON
  - Sign up for the RSS feed to stay up-to-date with announce.
- Common question: “What percentage of grade is lab 1 worth?”
  - Evasive answer: Not a lot but… don’t fall behind now!
- Lab partner search
  - Lab 2 is individual but you should aim to find a partner by next Friday (Oct. 17)
    - E-mail sat with your partner and TEAM NAME
Responses to Xilinx

- “Why doesn’t this work?”
- “It can’t be this bad. Maybe I just need to reinstall!”
- “I hate you, Xilinx!”
- “Isn’t there an open-source alternative?”
- “I’m never going to get this to work”
- “OK, maybe I should just restart the project.”
- “It worked (somehow). I can handle this…”

Lab 2: Build your fetch Unit

- **Overview:**
  - Part A: Design the datapath for fetch unit
    - We give you a datapath schematic
    - Leaf modules in RTL style, datapath uses structural style
  - Part B: Implement fetch unit control and test functionality
Fetch Unit Overview

Role of Fetch Unit

- Supply instructions to execution unit
  - How to handle branches?
    - Don’t wait… predict! (p-bit)
      - What if we are wrong?
  - Service Inst-Mem operations
    - Load/Store instructions
Fetch Unit Interface

Fetch Unit Interface Diagram:
- **Dequeue**: Instruction(data, addr, valid)
- **Restart**: restart
- **Restart Addr**: restart_addr
- **Memory Req**: (load, store, addr, store_data)
- **Load Data, Load Valid**: load_data, load_valid

Advanced Hardware Design with Verilog

By Sat Garcia
Complete the quote

- “Good artists ___ copy _____.
  Great artists ___ steal______.”
  - Pablo Picasso
- The following slides are only slightly modified from those in the MIT 6.375 course

Designing a GCD Calculator

- Euclid’s Algorithm for GCD (in C):

```c
int GCD(int inA, int inB)
{
    int done = 0;
    int A = inA;
    int B = inB;
    while (!done )
    {
        if ( A < B ) // if A < B, swap values
            {
                swap = A;
                A = B;
                B = swap;
            }
        else if ( B != 0 ) // subtract as long as B isn't 0
            {A = A - B;
            else
                done = 1;
        }
    return A;
}
```

How do we implement this in hardware?

Adapted from Arvind and Asanovic's MIT 6.375 lecture
Take 1: Behavioral Verilog

```verilog
module gcdGCDUnit_behav(#( parameter W = 16 ) // parameterize for better reuse
  input [W-1:0] inA, inB,
  output [W-1:0] out
);
  reg [W-1:0] A, B, out, swap;
  integer  done;
always @(*)
  begin
    done = 0;
    A = inA; B = inB;
    while ( !done )
      begin
        if ( A < B )
          swap = A;
          A = B;
          B = swap;
        else if ( B != 0 )
          A = A - B;
        else
          done = 1;
      end
      out = A;
  end
endmodule
```

What’s wrong with this approach?

Doesn’t synthesize! (notice that data dependent loop?)

Making the code synthesizable

- Start with behavioral and find out what hardware constructs you’ll need
  - Registers (for state)
  - Functional units
    - Adders / Subtractors
    - Comparators
    - ALU’s
Identify the HW structures

module gcdGCDUnit_behav( parameter W = 16 )
{
  input [W-1:0] inA, inB,
  output [W-1:0] out
};
reg [W-1:0] A, B, out, swap;
integer done;
always @(*)
begingroup
  done = 0;
  A = inA; B = inB;
  while ( !done )
  begin
    if ( A < B )
      swap = A;
      A = B;
      B = swap;
    else if ( B != 0 )
      A = A - B;
    else
      done = 1;
  end
  out = A;
endgroup
endmodule

Adapted from Arvind and Asanovic's MIT 6.375 lecture

Next step: define module ports

Adapted from Arvind and Asanovic's MIT 6.375 lecture
Implementing the modules

- Two step process:
  1. Define datapath
  2. Define control/control path

Adapted from Arvind and Asanovic’s MIT 6.375 lecture

Developing the datapath

Also need a couple MUXs

Adapted from Arvind and Asanovic’s MIT 6.375 lecture
Adding control

```
while (!done)
    begin
        if (A < B)
            swap = A;
            A = B;
            B = swap;
        else if (B != 0)
            A = A - B;
        else
            done = 1;
    end

Y = A;
```

Datapath module

```
module gcdDatapath#( parameter W = 16 )
(
    input  clk,
    // Data signals
    input  [W-1:0] operands_bits_A,
    input  [W-1:0] operands_bits_B,
    output [W-1:0] result_bits_data,
    // Control signals (ctrl->dpath)
    input  A_en,
    input  B_en,
    input  [1:0] A_mux_sel,
    input  B_mux_sel,
    // Control signals (dpath->ctrl)
    output  B_zero,
    output  A_lt_B
);
```

Adapted from Arvind and Asanovic's MIT 6.375 lecture
Implementing datapath module

```
wire [W-1:0] B;
wire [W-1:0] sub_out;
wire [W-1:0] A_mux_out;
3inMUX#(W) A_mux
  (#.in0 (operands_bits_A),
   .in1 (B),
   .in2 (sub_out),
   .sel (A_mux_sel),
   .out (A_mux_out));
wire [W-1:0] A;
ED_FF#(W) A_ff // D flip flop
  (#.clk (clk),
   .en_p (A_en),
   .d_p (A_mux_out),
   .q_np (A));
wire [W-1:0] B_mux_out;
2inMUX#(W) B_mux
  (#.in0 (operands_bits_B),
   .in1 (A),
   .sel (B_mux_sel),
   .out (B_mux_out));
ED_FF#(W) B_ff
  (#.clk (clk),
   .en_p (B_en),
   .d_p (B_mux_out),
   .q_np (B));
2inEQ#(W) B_EQ_0
  (#.in0(B),.in1(W'd0),.out(B_zero));
LessThan#(W) lt
  (#.in0(A),.in0(B),
   .out(A_lt_B));
Subtractor#(W) sub
  (#.in0(A),.in1(B),.out(sub_out));
assign result_bits_data = A;

Remember: Functionality only in "leaf" modules!
```

State machine for control

```
reset

WAIT
  Wait for new inputs

input_available
CALC
  Swapping and subtracting
  (B = 0)

DONE
  Wait for result to be grabbed

result_taken
```

Adapted from Arvind and Asanovic's MIT 6.375 lecture
Implementing control module

module gcdControlUnit
{
    input clk,
    input reset,

    // Data signals
    input input_available,
    input result_rdy,
    output result_taken,

    // Control signals (ctrl->dpath)
    output A_en,
    output B_en,
    output [1:0] A_mux_sel,
    output [1:0] B_mux_sel,

    // Control signals (dpath->ctrl)
    input B_zero,
    input A_lt_B
};

State update logic

- Remember: keep state update, next state calculation, and output logic separated.

localparam WAIT = 2’d0; // local params are scoped constants
localparam CALC = 2’d1;
localparam DONE = 2’d2;

reg [1:0] state_next;
wire [1:0] state;

RD_FF state_ff // flip flop with reset
{
    .clk (clk),
    .reset_p (reset),
    .d_p (state_next),
    .q_np (state)
};
Output signals logic

```verilog
reg [6:0] cs;
always @(*) begin
  // Default control signals
  A_mux_sel = A_MUX_SEL_X;
  A_en = 1'b0;
  B_mux_sel = B_MUX_SEL_X;
  B_en = 1'b0;
  input_available = 1'b0;
  result_rdy = 1'b0;
  case ( state )
    WAIT : begin
      A_mux_sel = A_MUX_SEL_IN;
      A_en = 1'b1;
      B_mux_sel = B_MUX_SEL_IN;
      B_en = 1'b1;
      input_available = 1'b1;
    end
    CALC : begin
      if ( A_lt_B )
        A_mux_sel = A_MUX_SEL_B;
        A_en = 1'b1;
        B_mux_sel = B_MUX_SEL_A;
        B_en = 1'b1;
      else if ( !B_zero )
        A_mux_sel = A_MUX_SEL_SUB;
        A_en = 1'b1;
    end
    DONE : begin
      result_rdy = 1'b1;
    end
  endcase
end
```

Next state logic

```verilog
always @(*) begin
  // Default is to stay in the same state
  state_next = state;
  case ( state )
    WAIT : begin
      if ( input_available )
        state_next = CALC;
    end
    CALC : begin
      if ( B_zero )
        state_next = DONE;
    end
    DONE : begin
      if ( result_taken )
        state_next = WAIT;
    endcase
end
```

Adapted from Arvind and Asanovic's MIT 6.375 lecture
Fibonacci Generator

- Goal: Design a fibonacci generator using Verilog
- \( F(n) = \)
  - 0 if \( n = 0 \)
  - 1 if \( n = 1 \)
  - \( F(n-1) + F(n-2) \) if \( n > 1 \)
- Before we start we need to know exactly what we are implementing

Fibonacci Module

- Reset = 1
  - Start at beginning
- Enable = 0
  - Stop and keep last number on output
- Enable = 1
  - Generate next number with each clk cycle
- Reset has precedence over enable
Developing an FSM for FibGen

- S0: “reset” state
  - Outputs “0”
- S1: first fib number
  - Outputs “1”
- S2: next fib num
  - Outputs new fib num
- S3: hold
  - Outputs last fib num calculated

Module interface and setup

module FibGen(input clk, rst, enb, output [16:0] out);

// states
parameter S0 = 2’b00,
    S1 = 2’b01,
    S2 = 2’b10,
    S3 = 2’b11;

// next state variables (combinatorial)
reg [15:0] next_reg_0;
reg [15:0] next_reg_1;
reg [15:0] next_fib;

// state variables (should become registers)
reg [15:0] reg_0 = 16’d0; // two fib nums ago
reg [15:0] reg_1 = 16’d1; // last fib num
reg [15:0] fib = 16’d0; // current fib num

reg [1:0] State;
reg [1:0] next_state;
Next state logic

always @ (*)
begin : next_state_logic
  next_state = State; // default is to keep current state
  next_fib = fib; // default keep current fib
  next_reg_0 = reg_0;
  next_reg_1 = reg_1;
  case( State )
  S0:
    begin
      if( enb == 1 )
        next_state = S1;
      else
        next_state = S0;
      next_fib = reg_0;
    end
  S1:
    begin
      if( enb == 1 )
        next_state = S2;
      else
        next_state = S1;
      next_fib = reg_1;
    end
  S2:
    begin
      if( enb == 1 )
        next_state = S2;
      else
        next_state = S3;
      next_fib = reg_0 + reg_1;
      next_reg_0 = reg_1;
      next_reg_1 = next_fib;
    end
  S3:
    begin
      if( enb == 1 )
        next_state = S2;
      else
        next_state = S3;
      next_fib = fib;
    end
  default: next_state = State;
endcase
end

Inferring HW structures

- What HW would our next state logic map into?
  - Lots of MUXes (based on if/case statements)
State assignment and output

assign out = fib;

always @ (negedge rst or posedge clk)
begin : state_assignment
  if (rst)
  begin
    reg_0 <= 16'd0;
    reg_1 <= 16'd1;
    fib <= 16'd0;
    State <= S0;
  end

  else
  begin
    State <= next_state;
    fib <= next_fib;
    reg_0 <= next_reg_0;
    reg_1 <= next_reg_1;
  end
end