CSE141L: Building your own processor!

Hung-Wei Tseng
You will design and implement a microprocessor!
Goals

• Practice what you will learn in CSE141
• Extend what you will learn in CSE141
  • Understand deeply how a processor works
  • See architecture play itself out in a real design
• Learn Verilog
• Get experience working on a large-scale project
• Have fun~~~
You will implement a pipeline MIPS processor in Verilog using 5 weeks
  • It will be able to run simple but real programs compiled using gcc
  • It should be able to do simple I/O
Make it your own processor
  • We will give you some code pieces
  • You have design the rest
  • We will give you the specifications for some others
  • You will invent, design, and implement some of your own
Course format

- Five labs
  - Due on every Friday
- Zoom Lectures —
  - The same Zoom link to CSE141’s lecture
  - Please check the schedule on https://www.escalab.org/classes/cse141L-2020su/
  - Verilog coding
  - Discussing current or upcoming labs
  - Like group office hours
- Youtube: https://www.youtube.com/profusagi
- No final exam
Warning!

- The course is **a lot** of work
  - Don’t let the 2 units fool you
- Don’t fall behind
  - The labs build on each other
  - Hard to catch if you fall behind
- Don’t wait till the last minute
  - It’s called **hard**ware for a good reason
  - The tools are complicated, buggy in some sense...
  - Your code will be buggy, too...
Lab 1: Familiar with the tools

• Two tutorials
  • Building projects in Quartus
  • Entering and compiling Verilog
  • Simulation using ModelSim
  • Measuring the performance of your design
• Start now!
• Due: this Friday — 8/7
Lab 2: Datapath elements

- Implementing the datapath elements required for a subset of MIPS instructions
- We will give you the design and some other key components
- You will implement the design
- Due next Friday — 8/14
Lab 3: Lights of life...

- Add control path to Lab 2
- Test your simple processor
- Execute simple programs
- Due on 8/21
Lab 4: It lives!

- Add missing pieces of MIPS
- You know how to have a working processor!
- Due 8/28
Lab 5: Let it live better!

• Pipeline your processor
• Measure the performance
• Due 9/4
Lab 6: Make it awesome!

• Optional
• Due 9/4 as well
• You can implement any other fancy features in your processor to get an A+
  • Cache
  • Dual-core
  • Branch predictor
  • Speculation
  • Dynamic scheduling
  • and etc...
Lab space and software

- We will use Altera tools for development (Quartus II)
  - Verilog editing
  - Design analysis
- We will use ModemSim for simulation
  - Simulation
  - Debugging
- Tools are huge pains
- They are also available for free
Do the work

• Lab 1 should be done independently
• Lab 2–5 should be in group of 2
  • Choose your group carefully
  • You cannot merge groups
  • Splitting up is allowed (but not encouraged)
  • Schedule an interview with TA (or Hung-Wei) before Friday 5pm every week when you’re done with the lab assignment
• We will be interviewing with the whole group
• No written report
<table>
<thead>
<tr>
<th>Achievement</th>
<th>Final Grade</th>
<th>Due</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hidden Level (Lab 6)</td>
<td>A+</td>
<td>9/4/2020</td>
</tr>
<tr>
<td>Lab 5</td>
<td>A</td>
<td>9/4/2020</td>
</tr>
<tr>
<td>Lab 4</td>
<td>B+</td>
<td>8/28/2020</td>
</tr>
<tr>
<td>Lab 3</td>
<td>B-</td>
<td>8/21/2020</td>
</tr>
<tr>
<td>Lab 2</td>
<td>C</td>
<td>8/14/2020</td>
</tr>
<tr>
<td>Lab 1</td>
<td>D</td>
<td>8/7/2020</td>
</tr>
</tbody>
</table>
Instructor

• Instructor: Hung-Wei Tseng
  • Lectures: http://goo.gl/VSL97g
  • Lab hours: http://goo.gl/VSL97g
• Check the calendar on our website http://goo.gl/VSL97g
Tutors

• Boram Wong
  • Lab hours: TuWTh 7p - 9:20p on Zoom
  • E-mail: bjung @ ucsd.edu

• Andrew Nyugen
  • Lab hours: F 2p-5p, Sa 12p - 2p, Su 11a-1p on Zoom
  • E-mail: ahn018 @ ucsd.edu
Course resources

• Course webpage:
  https://www.escalab.org/classes/cse141l-2020su/
• Canvas
  https://canvas.ucsd.edu/courses/17389
  we use Canvas to turn in reports, record grades.
• Discussion board:
  • Search before ask
  • https://www.piazza.com/ucsd/summer2020/cse141cse141l
Verilog

Prof. Usagi
Verilog

- Verilog is a Hardware Description Language (HDL)
  - Used to describe & model the operation of digital circuits.
  - Specify simulation procedure for the circuit and check its response — simulation requires a logic simulator.
  - Synthesis: transformation of the HDL description into a physical implementation (transistors, gates)
    - When a human does this, it is called logic design.
    - When a machine does this, it is called synthesis.
- In this class, we use Verilog to implement and verify your processor.
- C/Java like syntax
Data types in Verilog

- Bit vector is the only data type in Verilog
- A bit can be one of the following
  - 0: logic zero
  - 1: logic one
  - X: unknown logic value, don’t care
  - Z: high impedance, floating
- Bit vectors expressed in multiple ways
  - 4-bit binary: 4'b11_10 ( _ is just for readability)
  - 16-bit hex: 16'h034f
  - 32-bit decimal: 32'd270
## Operators

<table>
<thead>
<tr>
<th>Arithmetic</th>
<th>Logical</th>
<th>Bitwise</th>
<th>Relational</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>!</td>
<td>~</td>
<td>&gt;</td>
</tr>
<tr>
<td>-</td>
<td>&amp;&amp;</td>
<td>&amp;</td>
<td>&lt;</td>
</tr>
<tr>
<td>*</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>/</td>
<td>^</td>
<td>^</td>
<td>&lt;=</td>
</tr>
<tr>
<td>%</td>
<td>~^</td>
<td></td>
<td>==</td>
</tr>
<tr>
<td>**</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Don’t use**

<table>
<thead>
<tr>
<th>Concatenation</th>
<th>Replication</th>
</tr>
</thead>
<tbody>
<tr>
<td>{} (e.g., {1b'1,1b'0} is 2b'10)</td>
<td>{} (e.g., {4{1b'0}} is 4b'0)</td>
</tr>
</tbody>
</table>

Conditional: *condition ? value_if_true : value_if_false*
Wire and Reg

- wire is used to denote a hardware net — “continuously assigned” values and do not store
  - single wire
    ```
    wire my_wire;
    ```
  - array of wires
    ```
    wire[7:0] my_wire;
    ```
- reg is used for procedural assignments — values that store information until the next value assignment is made.
  - again, can either have a single reg or an array
    ```
    reg[7:0] result; // 8-bit reg
    ```
  - reg is not necessarily a hardware register
  - you may consider it as a variable in C
Revisit the 4-bit adder

- **A3** B3
- **A2** B2
- **A1** B1
- **A0** B0

**C3** **C2** **C1** **C0**

**O3** **O2** **O1** **O0**

**Full Adder Module**

**Half Adder Module**

reg, input

reg, output
### Half adder

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Out = A’B + AB’

Cout = AB

```verilog
module HA(input a, input b, output cout, output out);
assign out = (~a & b) | (a & ~b);
assign cout = a & b;
endmodule
```
Full adder

Out = A’BCin’ + AB’Cin’ + A’B’Cin + ABCin
Cout = ABCin’ + A’BCin + AB’Cin + ABCin

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

module FA( input a, input b, input cin, output cout, output out );
assign out = (~a&b&~cin)|(a&~b&~cin)|(~a&~b&cin)|(a&b&cin);
assign cout = (a&b&~cin)|(~a&b&cin)|(a&~b&cin)|(a&b&cin); endmodule
The Adder

wire [2:0] carries;
HA ha0(.a(A[0]), .b(B[0]), .out(O[0]), .cout(carries[0]));
FA fa1(.a(A[1]), .b(B[1]), .cin(carries[0]), .out(O[1]), .cout(carries[1]));
FA fa2(.a(A[2]), .b(B[2]), .cin(carries[1]), .out(O[2]), .cout(carries[2]));
FA fa3(.a(A[3]), .b(B[3]), .cin(carries[2]), .out(O[2]), .cout(cout));
endmodule

Connecting ports by name yields clearer and less buggy code.

module FA(input a, input b, input cin, output cout, output out);
assign out = (~a&b&~cin)|(a&~b&~cin)|(~a&~b&cin)|(a&b&cin);
assign cout = (a&b&~cin)|(~a&b&cin)|(a&~b&cin)|(a&b&cin);
endmodule

module HA(input a, input b, output cout, output out);
assign out = (~a & b)|(a & ~b);
assign cout = a&b;
endmodule
Always block — combinational logic

- Executes when the condition in the sensitivity list occurs

```verilog
module FA( input a,
           input b,
           input cin,
           output cout,
           output out );

assign out = (~a&b&~cin)|(a&~b&~cin)|(~a&~b&cin)|(a&b&cin);
assign cout = (a&b&~cin)|(~a&b&cin)|(a&~b&cin)|(a&b&cin);;
endmodule
```

Always block — sequential logic

• Executes when the condition in the sensitivity list occurs

```vhdl
always@(posedge clk)// the following block only triggered by a positive clock
begin
...
...
end
```
Blocking and non-blocking

- Inside an always block, = is a blocking assignment
  - assignment happens immediately and affect the subsequent statements in the always block
- <= is a non-blocking assignment
  - All the assignments happens at the end of the block

Assignment rules:
- The left hand side, LHS, must be a reg.
- The right hand side, RHS, may be a wire, a reg, a constant, or expressions with operators using one or more wires, regs, and constants.

Initially, \(a = 2, b = 3\)

```verilog
reg a[3:0];
reg b[3:0];
reg c[3:0];
always @(posedge clock)
begin
  a <= b;
  c <= a;
end
Afterwards: \(a = 3\) and \(c = 2\)
```

Afterwards: \(a = 3\) and \(c = 3\)
"Always blocks" permit more advanced sequential idioms

module mux4( input a,b,c,d, 
    input [1:0] sel, 
    output out );

reg out;
always @( *) 
begin 
    if ( sel == 2'd0 )
        out = a;
    else if ( sel == 2'd1 )
        out = b;
    else if ( sel == 2'd2 )
        out = c;
    else if ( sel == 2'd3 )
        out = d;
    else
        out = 1’bx;
end
endmodule

module mux4( input a,b,c,d, 
    input [1:0] sel, 
    output out );

reg out;
always @( *) 
begin 
    case ( sel )
    2’d0 : out = a;
    2’d1 : out = b;
    2’d2 : out = c;
    2’d3 : out = d;
    default : out = 1’bx;
endcase
end
endmodule

Courtesy of Arvind http://csg.csail.mit.edu/6.375/
Initial block

- Executes only once in beginning of the code

```plaintext
initial
begin
...
...
end
```
Testing the adder!

`timescale 1ns/1ns // Add this to the top of your file to set time scale
module testbench();
reg [3:0] A, B;
reg C0;
wire [3:0] S;
wire C4;
adder uut (.B(B), .A(A), .sum(S), .cout(C4)); // instantiate adder

initial
begin
A = 4'd0; B = 4'd0; C0 = 1'b0;
#50 A = 4'd3; B = 4'd4; // wait 50 ns before next assignment
#50 A = 4'b0001; B = 4'b0010; // don't use #n outside of testbenches
end
endmodule
Parameterize your module

module adder #(parameter WIDTH=32)(
  input[WIDTH-1:0]  A,
  input[WIDTH-1:0]  B,
  output[WIDTH-1:0] O,
  output  cout);
endmodule
Resources

• Check out MIT’s 6.375 course webpage http://csg.csail.mit.edu/6.375/
  • thanks to Asanovic & Arvind for slides

• Tips for using Altera tools https://sites.google.com/a/eng.ucsd.edu/using-the-altera-tools/
  • Thanks to Steven Swanson and other CSE141L winter 2012 staffs