Hardware Design Tips

EE 361  
University of Hawaii

Outline

• Verilog: some subleties  
• Simulators  
• Test Benching  
• Implementing the MIPS  
  – Actually a simplified 16 bit version
Monday

- Overview of verilog, again
- Implementing combinational circuits using verilog
  - Rules to avoid problems

Verilog

- Basic module
- Combinational subcircuits
- Sequential subcircuits (on wed)
Verilog: Basic Module

module circuitName(y1,y2,clock,reset,select,x1,x2,x3)
output y1,y2;
input clock, reset;
input select, x1, x2, x3;
wire h1,h2,h3;
reg k1,k2;
// Instantiations
Circuit circuit1(h1,h2,clock,h3,5,k1,k2);
endmodule

Combinational Circuits

• Continuous assign
• Procedural always
  – Rules
  – Examples of errors
  – More rules
Verilog: Combinational Subcircuits

Continuous assign

```
assign y = x1 + x2 + 3;
```

Formula (no begin-end blocks, if-else, case statements, or anything else)

For all input values, there should be an output value

Verilog: Combinational Subcircuits

Procedural always

```
always @(x1 or x2 or … or xk)
A description of how to compute outputs from the inputs
```

Example:
```
always @(x1 or x2) y = x1 + x2 + 3;
```

Sensitivity List
Rule of thumb: List of all inputs

From this description you should be able to write a truth table for the circuit

Be sure the table is complete, i.e., it covers all possible inputs OR For all input values, there should be an output value
Example: missing an input in the sensitivity list

```verilog
// 2:1 multiplexer
module mux(y, sel, a, b)
  input  a, b, sel;
  output y;
  always (a or b)
    begin
      if (sel == 0) y = a;
      else y = b;
    end
endmodule
```

Case of missing an input ("sel") in the sensitivity list.

Example: outputs are not defined for all inputs

```verilog
// 2:1 multiplexer
module mux(y, sel, a, b)
  input  a, b, sel;
  output y;
  always (a or b or sel)
    begin
      if (sel == 0) y = a;
      else y = b;
    end
endmodule
```

Case of not updating y for all inputs

Possible hardware implementation

It’s a transparent D latch
Example

```
always @(x1 or x2 or s)
begin
if (s == 1) h = 0;
else h = 1;
case(h)
  0: y = x1|x2; // AND the inputs
  1: y = x1&x2; // OR the inputs
endcase
end
```

Computation proceeds downwards (just like C language)

Variables
inputs: x1, x2, s
output: y
intermediate value: h

Next, we'll present some rules to use procedural always to model combinational circuits

<table>
<thead>
<tr>
<th>Input</th>
<th>x1</th>
<th>x2</th>
<th>Output</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

Truth table for circuit

Rules for procedural-always to model combinational circuits

Assignments

```
y = x1 + x2 + 3;
```

Blocking assignment (we'll discuss blocking and nonblocking assignments shortly)

Note that the left hand side is always an output or intermediate variable
Rules for procedural-always to model combinational circuits

Update variables at most once

always @ (x1 or x2)
begin
    y = x1 + x2;
    if (y > 4) y = x1;
    else y = x2;
end

y could be updated more than once BUT outputs shouldn't change twice when inputs change

always @ (x1 or x2)
begin
    r = x1 + x2;
    if (r > 4) y = x1;
    else y = x2;
end

We introduced a new reg variable r.
Now “r” and “y” are updated at most once.

Rules for procedural-always to model combinational circuits

always @ (x1 or x2)
begin
    blocking assignments (e.g., y = x1+x2)
    if-else case statements
end

module ....
always @ (x1 or x2)
begin
    Circuit circ1(y,x1,x2);
end
endmodule

module Circuit(h,g1,g2)
output h;
input g1, g2
assign h=g1+g2;
endmodule

This won't work. A module is not a C function.
Some rules to design combinational circuits with “always”

- Sensitivity list should have all inputs
- Outputs should have values for all possible inputs
- Variables that are updated (left side of blocking assignments) should be a register variable
- Update each variable at most once per change in input.
- Module instantiations are not C function calls
  - Use blocking assignments, case, if-else, maybe others such as “for” but be careful.

Wednesday

- Sequential circuits with verilog
- Electronic Design Automation (EDA)
Sequential Circuits

- Procedural always
- Examples
- Nonblocking and blocking assignments

**Rules for procedural-always to model sequential circuits**

```
x1 -> state
x2 -> state
x3 -> state
```

```
clock -> y1 = state
clock -> y2
```

we’ll assume this

**Example: D flip flop**
```
always @(posedge clock)
q <= d;
```

**Example: T flip flop**
```
always @(posedge clock)
if (t == 1) q <= ~q;
```

state vars are reg vars.

nonblocking assignments
Nonblocking Assignments

All flip flops get updated together on a positive clock edge.

```
always @ (posedge clock)
begin
  A <= 0;
  B <= A;
  C <= B;
end
```

All nonblocking assignments are updated together on the positive edge of the clock.

Example

```
begin  A <= 0;
       B <= A;
       C <= B;
end
begin  A = 0;
       B = A;
       C = B;
end
```

Suppose initially (A,B,C) = (1,1,1)

(A,B,C) = (0,1,1)  (A,B,C) = (0,0,0)
Example: 2-bit counter

```verilog
counter2(q, clock, s, d)
out [1:0] q; // 2-bit output
in clock;
s [1:0] s; // Select input
in [1:0] d; // Parallel load input
reg [1:0] q; // This is our state variable
always @(posedge clock)
begin
  case (s)
    0: q <= 0;
    1: q <= q + 1; // Counting up. Note that the count wraps around
       // when it goes past the value 3
    2: q <= q - 1; // Counting down. Also has wrap around
    3: q <= d;   // Parallel load
  endcase // Actually, the begin-end is unnecessary
end
endmodule
```

Example: Lights

```verilog
Lights(y, clock, s, d)
out [3:0] y; // 4-bit output
in clock;
s [1:0] s; // Select input
in [1:0] d; // Parallel load input
reg [1:0] q; // This is our state variable
always @(posedge clock)
begin
  case (s)
    0: q <= 0;
    1: q <= q + 1; // Counting up. Note that the count wraps around
       // when it goes past the value 3
    2: q <= q - 1; // Counting down. Also has wrap around
    3: q <= q;   // Hold
  endcase // Actually, the begin-end is unnecessary
end
// Continued
```
Example:
Lights

// Continued
always @(q)
case (q)
  0: y=4'b1000;
  1: y=4'b0100;
  2: y=4'b0010;
  3: y=4'b0001;
endcase
endmodule

Electronic Design Automation

- Simulator
- EDA process
- Test bench
Simulators, e.g., veriwell and Modelsim

Simulator will simulate what a circuit will do over time.

Time is divided into time units (fictitious) but you can think of them as some small time duration, e.g., 0.1ns

A variable keeps track of the "current time" (e.g., $time)
   Initially, "current time" = 0 (or 1)

Update variable values at time 1 based upon values at time 0
Update variable values at time 2 based upon values at time 1
and so on.

Example

\[
\begin{align*}
\text{\$time = 0} & : 0 & \quad 0 & \quad 1 & \quad 1 \\
\text{\$time = 1} & : 0 & \quad 1 & \quad 1 & \quad 1 \\
\text{\$time = 2} & : 0 & \quad 1 & \quad 0 & \quad 0
\end{align*}
\]
Simulator

verify that your circuit works
(debugging)

Test Bench

inputs to excite the circuit
outputs to observe behavior
module testbench;

reg clock;      // Clock signal
reg [1:0] A;    // Inputs A and B to excite
reg [2:0] B;
wire [2:0] C;   // Output C to observe

icChip cl(A,B,C,clock); // Instantiation of the circuit to test

init clock = 0; // Clock generation, with clock period = 2 time units
always #1 clock = ~clock;

init // Changing inputs to excite icChip cl
begin:
    // Input values for testing
    A = 0;
    B = 0;
    #2 A = 1; // After 2 time units, A is changed to 1.
    #1 B = 3; // After another time unit, B changes to 3.
    #2 $stop; // After 2 more time units, the simulation stops
end

init // Display of outputs
begin:
    $display("A B   C   clock time"); // Displayed once at $time = 0
    $monitor("%d %d %d %d %d", A,B,C,clock,$time); // Displayed whenever variable changes
    // Note that $monitor can occur at most once in verilog code
    // while $display can occur many times.
end

endmodule

Friday

- Building single cycle MIPS – naive way
- MIPS-L: 16 bit version of MIPS
- Build it in stages
  - MIPS-L0: executes only R-type instructions
    - Tips on testing and debugging
  - MIPS-L1, L2, L3
Building Single Cycle MIPS

How NOT to build a single cycle MIPS in verilog.

1. Build all the components in verilog.
2. Test/debug each component
3. Put everything together into the single cycle MIPS
4. Simulate --> syntax errors
5. Fix syntax errors, and then simulate --> ‘xxx’
7. Hmmm. Must be the simulator. Reset PC.
   Simulate again.
8. Problem too hard -- it’s a complicated computer
   after all

*First three steps are okay. But need improvement after that.*

MIPS-L

• To experience building a moderately large circuit, you will build a MIPS computer
  – Homework 10A and 10B
• MIPS-L: 16 bit version of MIPS
  – Description
  – Simplified version MIPS-L0
    • Only R-type arithmetic instructions
    • Modified register file: RegFileL0
MIPS-L Description

- Instructions (and integer data) are 16 bits long
  - Word = 16 bits
- Addresses are 16 bits
- Eight general purpose registers
  - $0-$7
  - $0$ is always equal to 0
- Memory is byte-addressable and big Endian
  - Addresses of words are divisible by 2

MIPS-L Register Convention

<table>
<thead>
<tr>
<th>Name</th>
<th>Register Number</th>
<th>Usage</th>
<th>Preserved on call?</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>0</td>
<td>the constant 0</td>
<td>n.a.</td>
</tr>
<tr>
<td>$v0-$v1</td>
<td>1-2</td>
<td>values for results and</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td>expression evaluation</td>
<td></td>
</tr>
<tr>
<td>$t0-$t2</td>
<td>3-5</td>
<td>temporaries</td>
<td>No</td>
</tr>
<tr>
<td>$sp</td>
<td>6</td>
<td>stack pointer</td>
<td>Yes</td>
</tr>
<tr>
<td>$ra</td>
<td>7</td>
<td>return address</td>
<td>No</td>
</tr>
</tbody>
</table>
MIPS-L Instruction Formats

<table>
<thead>
<tr>
<th>Name</th>
<th>Fields</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Field Size</td>
<td>3 bits 3 bits 3 bits 4 bits</td>
<td>ALL MIPS-L instructions 16 bits</td>
</tr>
<tr>
<td>R-format</td>
<td>op rs rt rd funct</td>
<td>Arithmetic instruction format</td>
</tr>
<tr>
<td>I-format</td>
<td>op rs rt rd</td>
<td>Address/ Immediate format</td>
</tr>
<tr>
<td>J-format</td>
<td>op target address</td>
<td>Jump instruction format</td>
</tr>
</tbody>
</table>

MIPS-L Machine Instructions

<table>
<thead>
<tr>
<th>Name</th>
<th>Format</th>
<th>Example</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>R</td>
<td>0 2 3 1 0</td>
<td>add $1,$2,$3</td>
</tr>
<tr>
<td>sub</td>
<td>R</td>
<td>0 2 3 1 1</td>
<td>sub $1,$2,$3</td>
</tr>
<tr>
<td>and</td>
<td>R</td>
<td>0 2 3 1 2</td>
<td>and $1,$2,$3</td>
</tr>
<tr>
<td>or</td>
<td>R</td>
<td>0 2 3 1 3</td>
<td>or $1,$2,$3</td>
</tr>
<tr>
<td>slt</td>
<td>R</td>
<td>0 2 3 1 4</td>
<td>slt $1,$2,$3</td>
</tr>
<tr>
<td>jr</td>
<td>R</td>
<td>0 7 0 0 8</td>
<td>jr $7</td>
</tr>
<tr>
<td>lw</td>
<td>I</td>
<td>4 2 1 100</td>
<td>lw $1,100($2)</td>
</tr>
<tr>
<td>sw</td>
<td>I</td>
<td>5 2 1 100</td>
<td>sw $1,100($2)</td>
</tr>
<tr>
<td>beq</td>
<td>I</td>
<td>6 1 2</td>
<td>beq $1,$1,100</td>
</tr>
<tr>
<td>addi</td>
<td>I</td>
<td>7 2 1 100</td>
<td>addi $1,$2,100</td>
</tr>
<tr>
<td>j</td>
<td>J</td>
<td>2 5000</td>
<td>j 10000</td>
</tr>
<tr>
<td>jal</td>
<td>J</td>
<td>3 5000</td>
<td>jal 10000</td>
</tr>
</tbody>
</table>
Single Cycle MIPS-L

This thing's got a cycle in it. Not good. Build it in stages.

The following is JUST A SUGGESTION

MIPS-L0

- Something even simpler: MIPS-L0
- Only executes R format arithmetic instructions
- Register File RegFileL0
  - Register $5 = 1 always
  - Register $4 = 2 always
- ALUControl: Same as MIPS-L
MIPS-L0

- Describe MIPS-L0
- Example Program Memory
  - Example testbench
  - General testbench of simple combination circuits
- Building Single Cycle MIPS-L0
  - Pre MIPS-L0, yet even simpler
  - Debugging tips
- Building Single Cycle MIPS-L0

Everything is 16 bits except reset and clock
MIPS-L0

Example Instruction Memory

// Instruction memory
// Program has only 8 instructions
module IM(addr, dout)
input [15:0] addr;
dout [15:0] dout;
reg [15:0] dout;

always @(addr[3:1])
case (addr[3:1])
  0: dout = {3’d0,3’d4,3’d5,3’d3,4’d0} // add $3,$4,$5
  1: dout = // sub $2,$4,$5
  2: dout = // slt $1,$4,$5
  3: dout = // and $1,$3,$5
  4: dout = // slt $1,$4,$3
  5: dout = // sub $1,$3,$5
  6: dout = // add $3,$0,$2
  7: dout = // add $3,$5,$1
endcase
endmodule
Example Testbench

// Testbench for program memory
module testbench_memory
reg [15:0] addr;
wire [15:0] dout;

IM pmem(addr, dout); // Instantiation of memory

initial begin // Drive the memory
addr = 0;
#2 addr = 2;
#2 addr = 4;
......
#2 addr = 14;
#2 $stop;
end

initial begin // Display the outputs
$monitor("time = %d, addr=%d, instr=(%d,%d,%d,%d,%d)",time,addr,dout[15:13]...$end
endmodule

Simple Testbenches

• Declare reg variables to drive inputs
• Declare wire variables to tap into outputs and connect circuits
• Clock generator signal (if necessary)
• Set up circuit with instantiations and possible connections
• Initial procedure to change input signals over time (use delays # and $stop)
• Initial procedure to output results
Building MIPS-L0

- Build the components and test
  - Instruction memory
  - ALU
  - ALU Control
  - Register file RegFileL0
- Build and test a Pre-MIPS-L0 (see next slide)
- Build and test a MIPS-L0 (finally!)
Pre MIPS-L0

• What do we gain by removing the cycle?
  – If there’s a bug, we can find it by tracing backwards

• Testbench
  – Has an instantiation of MIPS-LO and program memory
  – Try different programs for testing

Pre MIPS-L0

• Rules of thumb for debugging
  – Determine a set of input signals to test whether the circuit works or not
  – Input signals will vary over time
  – Determine where to probe signals
    • At outputs
    • At intermediate points
  – Determine by hand what signals to expect
Pre MIPS-L0

• Rules of thumb for debugging
  – Create test bench that
    • generates the appropriate input signals over time
    • outputs the signals you want
  – Run the test bench and see if the circuit works
  – If not, put probe signals upstream to determine where the problem is
Pre MIPS-L0

• Rules of thumb for debugging
  – Change input signals to REALLY verify that the circuit will work under all conditions
  • Example: change program

MIPS-L0

• After verifying correctness of Pre MIPS-L0, build and test MIPS-L0
• This is Homework 10A
MIPS-L

- Build the MIPS-L in stages
- Stage 1 MIPS-L1
  - Include addi and j
  - Use ordinary register file
  - Include Control circuit
- Stage 2 MIPS-L2:
  - include beq
  - Modify ALU for zero output

MIPS-L

- Stage 3 MIPS-L3 (final)
  - Include lw and sw
  - Have the RAM store 128 words.
  - RAM is implemented like a register file
    - Have the program read and write to RAM and check if it does so properly
    - Verification can be done by checking inputs and outputs
  - This is Homework 10B
Testbenching

- Your testbench should include
  - Your MIPS-Lx
  - Program memory
    - Don’t be afraid to try different programs for testing
- Last slide unless there’s more time
Single Cycle MIPS

Check controller outputs

EE 361 Fall 2003
University of Hawaii

57

Single Cycle MIPS

Check register file outputs.
You must initialize register values somehow

EE 361 Fall 2003
University of Hawaii

58
Single Cycle MIPS

Complete datapath but not writing to anything

Program just has R-type and j instructions
Single Cycle MIPS