Lecture 4: More Lab 1

David Black-Schaffer
davidbbs@stanford.edu
EE183 Spring 2003

Overview

- Hardware vs. Software
  - Think about how many adders you are creating
  - Hierarchical design is good for both **scalability** and **debugging**!
- Output to VGA
  - Double-buffering via dual BRAMs
  - Take the MSBs of the VGA position to index into BRAMs ***(understand this!)***
- FSMs
  - Inferred latches: bad; Knowing your hardware: good

Logistics

- Any questions?
  - Lab 1 requirements clear?
  - We'll be talking about RAM, FSMs, and VGA today
- Finished the tutorial yet?
  - Due **TONIGHT** by midnight
    - email URL for your PDF to Joel
    - See "Tao of EE183" on web
    - Tutorial is a mini-writeup (not a lot of text)
- Start on pre-lab for lab 1
  - Complete Design section
  - FSMs and block diagram
  - The more you do here the easier lab 1 will be…

Lab 1 Requirements

- Implement a VGA Game of Life
  - 64x64 wrap-around grid
  - Update speed 2-4Hz
  - Gamepad editor for initial state and edit/run mode select with a blinking cursor
- Optional (encouraged and easy)
  - Speed-up button
  - Background image

Hierarchical Design

- Two main modes:
  - Edit
  - Run
- Two main modules?
- Master Control module?
- Think about hierarchical FSMs for both design & debugging
  - What should the top level FSM do?
  - What should the lower-level FSMs be?
  - Can the enables/resets help?
  - Do you need other signals? (done? slow clock?)

BRAM stores game state

- Each BRAM is 4Kbits
  - So we can have a 64x64x1 game board
  - Use CoreGen and create it as 4Kx1 and use the **concatenation of the row and column as the index:** {X,Y} where X, Y are 6-bit (0..64)
  - Is this clear?
- But, we need two of them
  - One to store current state
    and one to write the next state into.

Reading from a RAM

- How do you read from a RAM?

```
case(state)
   `READ_UP:
        bram_address = {X,Y-1};
        bram_enable = 1;
        next_state = GET_UP;
        neighbor_counter_enable = 0;
   `GET_UP:
        bram_address = 12'b0; //WHY!?
        bram_enable = 0;
        next_state = READ_LEFT;
        neighbor_counter_enable = bram_out;
   `READ_LEFT:
        ...
```

Address in here

Data out
One (?) cycle later!

Next State FSM

- Think about the algorithm for one cell
  - How would you do this in C?
    Iterate over every cell and look at the neighbors.
  - Remember this is in hardware!
  - Speed vs. area is always the tradeoff

How many adders? Does it matter?

C Algorithm

```
for x = 0..63 {
  for y = 0..63 {
  neighbors = 0;
  // Look at all the neighbors
  if (current_state[x-1][y])
    neighbors++;
  if (current_state[x-1][y-1])
    neighbors++;
  if (current_state[x][y-1])
    neighbors++;
  ...

  if (current_state[x][y] == 0)
    if (neighbors == 2) || (neighbors = 3)
      next_state[x][y] = 1;
  else if (neighbors == 3)
    next_state[x][y] = 1;
  else
    next_state[x][y] = 0;

  }
}
```

What is a for loop in hardware?
A counter with a comparator for the end value?
But it will be enabled by a FSM!

How many adders do you have here?
Two for each neighbor? Or re-use two?

How do you store the number of neighbors?
A counter? Controlled by the FSM?

How do you swap the next_state and current_state after each iteration? MUX?

Verilog State Machine(s)

```
always @(current_state_q or ??)
   begin
        case(current_state_q)
        `CHECK_LEFT: begin
                next_state_d = ??
             end
        `CHECK_UPLEFT: begin
                next_state_d = ??
             end
        `CHECK_UP: begin
                next_state_d = ??
             end
        `CHECK_UP_RIGHT: begin
                next_state_d = ??
             end
        .
        .
        .

        `UPDATE_CELL: begin
                next_state_d = ??
             end
endcase
```

How do you get the current_state[x-1][y]?

How do you keep track of the neighbor count?

How does your memory access fit in here?

How do you change the x and y for each cell?

Double Buffering

- We need to have the VGA logic switch between the two BRAMs depending on which one is the current complete game state.
  - Standard technique in many areas but specifically graphics.
  - current_bram vs. next_bram

  - **Make this a self-contained module so you only have one current output and the VGA doesn't have to know which BRAM to use. Standard OOP practice.**

  (what will deal with the switching?)

How do we display?

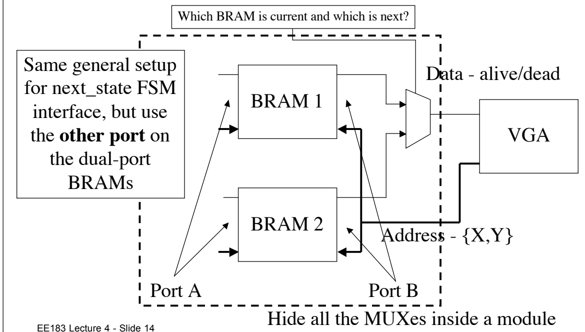
- When it is refreshing, the VGA needs a data element every clock (remember it gives you the X and Y and you give it the color - it always needs some color)
  - Whenever the VGA is valid we **MUST** provide it with a color
  - We could coordinate the game state updates to occur around the VGA screen refreshes but this is a pain.

- Instead use a **dual port memory** and use one port for the game state updates and one for display.
  - Dual port memories are already implemented in BRAMs so we might as well use it.
  - Port A is for updating the game state (FSM control)
  - Port B is for the VGA color output (VGA control)

Key VGA Points

- Two concepts here:
  - Double Buffering: draw the next state into one memory while displaying from the first, then switch
  - Dual-port RAM: use one port for the VGA to draw from and the other for the next_state FSM to update the game state
- These work together to make this fast and easy

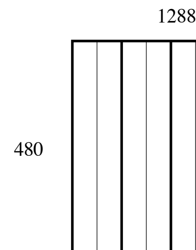
Double Buffering & Dual Ports



How do we actually generate the image?

- Split the screen up into a 64x64 grid
  - Fine if it is in top left corner
  - Can you use a divider?
    How do you divide by a power of 2?
- Could directly draw to the screen
- Could use TCGROM
  - Maybe change one of the characters to a solid block and then an outline block so there is a grid? Cheap ways to make a grid?

VGA



- We want 64x64
- No dividers
- Okay if it doesn't fill the screen

Bit 0: divide in half
Bit 1: divide in quarters
Bit 2: divide in eights
…

Optional Fun Things

- Display the number of iterations
- Clear-screen button
- Random start state button
  - LFSR seeded by free-running counter
- Fast-forward
- Interesting initial state
  - Use Memutils.zip to load a state into the BRAM via a .coe file
- Background image
  - Simply use a third BRAM and display it when the cell is off

FSM Review

A Finite State Machine is simply a **state register** that holds the current state and some **combinational logic** which calculates the next state and outputs based on the current state and the inputs.

- A feedback system which updates on each clock

What is it really?

- A bunch of MUXes which select the next state & outputs based on the current state (FFs) & inputs.
- Keep this in mind when writing your verilog.

Inferred Latch Review

An inferred latch occurs when **you fail to define all the outputs** for a given path through your logic and the tools instead try to remember the previous output to make up for this.

- All outputs have to be defined at all times in real hardware

What is it really?

- A mux where you haven't defined one of the inputs so it remembers the last value and loops it around.
- Make sure you know and state what all outputs should be at all times.


```
always @(current_state_q or button)
   begin
         case(current_state_q)
         `STOP: begin
                  go = 1'b0;
                  if (button) begin
                           next_state_d = `GO;
                           end
                  else     begin
                           next_state_d = `STOP;
                           end
                  end
         `GO: begin
                  go = 1'b1;
                  if (button) begin
                           next_state_d = `STOP;
                           end
                  else     begin
                           next_state_d = `GO;
                           end
                  end
         default: begin
                  go = 1'b0;
                  next_state_d = `STOP;
                  end
```

The next state must **ALWAYS** be defined.

Any output must be defined in **EVERY** state.

If any states are ever not used they **MUST** be included in a `default` statement.

an `else` for every `if`
a `default` for every `case`
every output must be defined in every state

Inferred Latches

- So why are they bad, really?
  - Hard to debug (hidden states in your logic)
    - You intend a MUX and get a MUX with memory
    - You intend a certain number of states but end up with states with memories (i.e., sub-states)
    - You weren't really thinking things all the way through or you wouldn't have had them!
  - Hard to optimize speed (hidden states in your logic)
    - What is your critical path? Depends on how many levels of logic between FFs.

Timing

- What limits our speed?
- RTL design - Register Transfer Logic
  - Speed limited by the time it takes to get from one register (flip flop) to another
  - State machines and pipeline data stages
- What gets in the way?
  - Combinational logic delay
  - Routing delay
  - FF setup and hold time requirements
  - Clock skew and delay

Combination Logic Delay

- Time to get through gates
  - The more gates (more complicated logic) the slower
  - One-hot encoding
- Well, sort of:
  - FPGA doesn't have "gates" we have those 4-input LUTs (look-up-tables)
  - All functions of 4 inputs or less are the same speed
  - >4 inputs and we have to hook up multiple CLBs (routing delay)

Routing Delay

- This can be (IS!) the real killer
- FPGA wiring slow compared to ASIC
  - Lots of switches to go through
  - Wires don't go exactly where you want
- Routing can be ~50% of your total delay
- Manual placement? NP-hard problem?
- Hope the tools work well!

Lecture 4 Key Points

- VGA gives you an X and Y coordinate and you set the color.
- Double buffering: calculate from one memory into the other
- Calculte-next-state FSM looks at all the neighbors to figure out alive/dead

- Logistics
  - **Prelab must contain all your FSM description (pseudo-verilog) and block diagrams** (don't waste your time drawing pictures in Word)