Lecture 3: Lab 1

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

Overview

- VGA
  - You get an X and Y and assign the color. The VGA controls your logic
- Debouncing efficiently:
  - Use a global slow-clock to enable a much smaller counter in each debouncer
- BRAMs
  - Dual-port: have one port for your next-state update and one port for drawing to VGA
  - You need two BRAMs (one for current state and one for next state)
- Hardware vs. Software
  - Think about how many adders you are creating
  - Hierarchical design! Hierarchical design!

Logistics

- Any questions?
  - Verilog? Tutorial? FSMs?
- Finished the tutorial yet?
  - Due Wednesday by midnight
  - All submissions are PDF format **email/URL to Joel**
    - See "Tao of EE183" on web
    - Tutorial is a mini-writeup
  - Pick up CDs to install at home
    - Do **NOT** install the MultiLINX software
    - **You must return the CDs (we have only 10 copies)**
    - License ID is on the Tao of EE183 webpage under Foundation

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 with blinking cursor
- Optional (encouraged and easy)
  - Speed-up button
  - Background image

EE183 Design Process

1. Understand the problem and choose a hierarchical decomposition
2. Design the **FSMs** (bubble diagrams) and **Data Path** (block diagram)
3. Code the FSMs in verilog
4. Simulate the FSMs
5. Connect the FSMs to the data path in a top-level verilog file
6. Run the static timing tool to check timing
7. Test it out, then repeat the above until it works

Understanding the Problem

- **Output: VGA**
  Need to display a grid of cells…
- **Input: Game pad**
  How does this integrate with the algorithm?
- **Control: Game of Life Algorithm**
  Easy in C, but how to make it run in hardware?

Output: VGA

- VGA monitors are raster devices
  - Data is sent sequentially for each pixel in conjunction with the sync signals
  - Sync signals determine the resolution (timing critical)
  - The image is 'painted' across and then down the screen
  - 6 bits of color: 2 for each of red, green, blue
- **`sync_gen50`** (50MHz) module
  - Generates the sync signals
  - Gives you an X and Y and you set the color
  - **It controls your logic and tells you what to draw when**

VGA Example: Tutorial

```
sync_gen50 syncVGA(     .clk(clock), .CounterX(XPos),
                        .CounterY(YPos), .Valid(Valid),
                        .vga_h_sync(vga_hsync),
                        .vga_v_sync(vga_vsync)
                   );

wire red0 = Valid &&    ((XPos <200 ) ||
                         (YPos > 80 ) );
wire red1 = Valid &&    ((XPos > 200) && (XPos  < 400) ||
                         (YPos > 80 ) && (YPos  < 160) );


wire green0 = Valid &&  ((XPos > 400) && (XPos  < 600) ||
                         (YPos > 160) && (YPos  < 240) );
wire green1 =  Valid && ((XPos > 600) && (XPos  < 800) ||
                         (YPos > 240) && (YPos  < 320) );

wire blue0 = Valid &&   ((XPos > 800) && (XPos  < 1000)||
                         (YPos > 320) && (YPos  < 400) );
wire blue1 = Valid &&   ((XPos > 1000)|| (YPos  > 400) );
```

Instantiate a copy of the **`syncVGA`** module for generating X, Y, and sync signals.

red0 will be on if X < 200 or Y > 80.

red1 will be on if X>200 and X<400 Or or Y>80 and Y<160.

And so on for green and blue.

VGA as a Raster Device

- Given an X and Y
- You turn on the colors to make a picture

- Your logic waits for the right X and Y and then turns on the appropriate colors

  What are the "right" colors? Well, how about black when the game of life cell is not alive and red when it is alive? You just need to index the cells based on X and Y. (I.e., divide the screen up into a grid.)
- Watch timing!
  - Your whole design must meet 50MHz for the **`sync_gen50`** module

Input: Game pad

- Inputs MUXed
  - B/C and A/Start button
  - How do you use both? (See 121 notes)
- Synchronizing
  - We'll have a guest lecture on this, but basically if a FF goes metastable it can propagate through your system. More likely with higher clocks.
  - Run all inputs through 2 FFs to try and make sure they are synced to the clock
- Debouncing
  - Mechanical buttons physically bounce
  - Need to ignore all spurious inputs after the first edge

Input: Debouncing

- What's wrong with the debouncer I gave you?
  - Uses a 20-bit delay counter to wait for the switch to settle (~20 ms delay)
  - So what? It seems to work.
  - Well, what if you have six inputs? You need six debouncers = **six 20-bit counters = 120 FFs.**
- How can we fix this?
  - Create a global 'slow clock' which can then be used as an **enable** to smaller counters. Why an enable?
  - If we have a 1ms 'slow clock' **enable**, then we only need to count to 20 in each debouncer (5 FFs)
  - **You must implement your own debouncer and fix this for lab 1!**
  - What else can you use the slow clock for?
  - This is the **only** gating of the clock you are allowed

Input: Be Smart

- Two main modes:
  - Edit
  - Run
- Think about hierarchical FSMs (design & debugging!)
  - What should the top level FSM do? (enable Edit/Run?)
  - What should the lower-level FSMs be? (Edit/Run?)
  - Can the enables/resets help? (Think about the tutorial!)

Lab #1: Conway's Game of Life

- The Rules (i.e., the algorithm)
  - For a space that is 'populated':
    - Each cell with one or no neighbors dies, as if by loneliness.
    - Each cell with four or more neighbors dies, as if by overpopulation.
    - Each cell with two or three neighbors survives.
  - For a space that is 'empty' or 'unpopulated'
    - Each cell with three neighbors becomes populated.
- To simplify things, game board wraps around
- Example:
  - http://www.bitstorm.org/gameoflife/

Implementation

- Need to store the current state
  - 64x64 grid: some sort of RAM
- Update: need to calculate the next state for each cell
  - FSM to do the calculations by iterating over the cells and looking at the neighbors
  - Where do we store the results?
- Need 2 RAMs
  - One for current state
  - One for next state
  - Calculate next state from current state and write it into next state RAM

RAMs

- CoreGen Block RAMs (BRAMs)
  - Ten 4096 bit dual-port synchronous RAMs on the chip. (We get these for free!*)
  - Features
    - Can be configured as any width/depth. What do we want? (64x64 grid, on/off = ?)
    - Synchronous. **Takes 1 cycle to get the results!**
    - Dual-port: can read/write two locations at once!
    - *Routing delay vs. using Distributed RAM
    - Simulation issues

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)
- But, we need two of them
  - One to store current state and one to write the next state into
  - Need data from all 8 neighbors but rows above have already been processed

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
- Speed
  - Is this inefficient?
  - Each full update is ~ 20ns*8*64^2 = 0.6ms
  - So 1500 updates a second
  - How fast is a human? How fast is the VGA!?

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?
One for each neighbor? Or re-use one?

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 do you change the x and y for each cell?

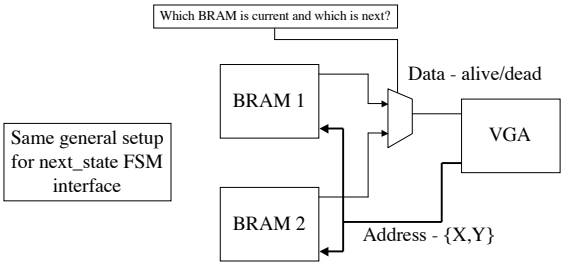
How do we display?

- 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
- 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)

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.

Double Buffering



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?

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

Lecture 3 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
  - Finish up the tutorial so you can work on the prelab for lab 1
  - **Prelab must contain all your FSM/block diagrams**

EE183 Lecture 3 - Slide 25