Design Studies – Part 3

- VGA display buffer
- Simple video game

Jon Turner
VGA Display

- VGA interface designed to drive CRT display
  - electron beam sweeps across screen in successive rows
  - driven by vertical and horizontal sync signals
  - pixel data displayed during display period of each sweep
    - designed for 640x480 display at pixel clock period of 40 ns
  - modern digital displays emulate CRT behavior
Design Notes

- Proto board memory only large enough for 320x240 pixels
  - so, each "memory pixel" used for 2x2 block of "screen pixels"
- Separate vertical and horizontal states for sync, back porch, display region, front porch driven by counters tick, line
- Use synchronous block RAM with separate data in and out
State Diagrams

\( hState \)
(note 50 MHz clock)

- \( \text{syncState} \)
  - \( \text{frontPorch} \)
    - \( \text{tick} = hFpWidth - 1 \)
      - \( \text{tick} < 0 \), \( \text{line} = \text{line} + 1 \)
    - \( \text{tick} = hDispWidth - 1 \)
      - \( \text{tick} < 0 \)
- \( \text{backPorch} \)
  - \( \text{tick} = hSyncWidth - 1 \)
    - \( \text{tick} < 0 \)
  - \( \text{tick} = hBpWidth - 1 \)
    - \( \text{tick} < 0 \)

\( vState \)
(updated at end of horizontal lines)

- \( \text{syncState} \)
  - \( \text{frontPorch} \)
    - \( \text{line} = vFpWidth - 1 \)
      - \( \text{line} < 0 \)
    - \( \text{line} = vDispWidth - 1 \)
      - \( \text{line} < 0 \)
  - \( \text{backPorch} \)
    - \( \text{line} = vSyncWidth - 1 \)
      - \( \text{line} < 0 \)
    - \( \text{line} = vBpWidth - 1 \)
      - \( \text{line} < 0 \)
  - \( \text{displayRegion} \)
entity vgaDisplay is port (
  clk, reset: in std_logic;
  en, rw: in std_logic;
  addr: in dbAddr; data: inout pixel;
  -- video outputs
  hSync, vSync: out std_logic;
  dispPix: out pixel);
end vgaDisplay;
architecture al of vgaDisplay is

type dbType is array(0 to 240*320) of pixel;
signal dBuf: dbType;
signal dispAddr: dbAddr;

-- signals and constants for generating video timing
begin
  type stateType is (syncState, frontPorch, displayRegion, backPorch);
signal hState, vState: stateType;

  -- horizontal clock tick counter, vertical line counter
  signal tick: unsigned(10 downto 0);
signal line:   unsigned( 9 downto 0);

  -- Constants defining horizontal timing, in 50 MHz clock ticks
  constant hSyncWidth: hIndex := to_unsigned(192,tick'length);

  -- Constants defining vertical timing, in horizontal lines
  constant vsyncWidth: vIndex := to_unsigned(2,line'length);
begin
  -- display buffer process - dual port memory
  process (clk) begin
    if rising_edge(clk) then
      data <= (others => 'Z');
      if en = '1' then
        if rw = '0' then
          dBuf(int(addr)) <= data;
        else
          data <= dBuf(int(addr));
        end if;
      end if;
      dispPix <= dBuf(int(dispAddr));
    end if;
  end process;
  dispPix <= dispData when vState = displayRegion
    and hState = displayRegion
    else (others => '0');
-- generate display timing signals and display address
process (clk) begin
  if rising_edge(clk) then
    if reset = '1' then
      vState <= syncState; hState <= syncState;
      tick <= (others => '0'); line <= (others => '0');
      dispAddr <= (others => '0');
    else
      -- generate horizontal timing signals
      tick <= tick + 1; -- increment by default case hState is
      when syncState =>
        if tick = hSyncWidth-1 then
          hState <= backPorch; tick <= (others => '0'); end if;
        when backPorch => ...
      when displayRegion =>
        if tick = hDispWidth-1 then
          hState <= frontPorch; tick <= (others => '0'); end if;
        if vState = displayRegion then
          if tick(1 downto 0) = "11" then dispAddr <= dispAddr+1;
          end if;
          if tick = hDispWidth-1 and line(0) = '0' then
            dispAddr<=dispAddr-to_unsigned(319,dispAddr'length);
          end if;
        end if;
    end if;
  end if;
end process;
when frontPorch =>
  if tick = hFieldWidth-1 then
    hState <= syncState; tick <= (others => '0');
   end if;
end when;

-- generate vertical timing signals at end of line
line <= line + 1;
case vState is
  when syncState =>
    dispAddr <= (others => '0');
    if line = vFieldWidth-1 then
      vState <= backPorch; line <= (others=>'0') end if;
  when backPorch => ..
  when displayRegion => ..
  when frontPorch => ..
  end case:
end if;
end if;
end when;
end if;
end if;
end process;

hSync <= '0' when hState = syncState else '1';
vSync <= '0' when vState = syncState else '1';
end al;
Simulation – Read/Write
Vertical Timing
Horizontal Timing

- Line numbers
- Horizontal states, sync pulse
- Display blanked outside display region
- Display pixels incremented
Display Address Updating

- **Even line**
- **Odd line**
- **19200 = (120/2) * (640/2)**
- **Repeating display buffer addresses**
- **Display buffer address advances by 320**
Exercises

1. Modify the VHDL for the vgaDisplay component to produce even lower resolution images, specifically 160 pixels per line and 120 lines. How much memory is needed for the display buffer in this case?

2. The vgaDisplay component uses a dual-port memory. Consider an alternate design using a single port memory. What affect would this have on the video read-out process? Suggest a way that you could design a system with a single port memory that still allows the client and video controller to access the display buffer without conflicts. Hint: recall that the display address is only incremented on every fourth clock tick.
Minesweeper Game

- Hardware version of classic PC game
  - VGA display show game state
  - knob selects square
  - buttons used to flag square or step on it
  - mines placed randomly, density controlled by switches

- Representing game state
  - \textit{isMined}(x,y) if square (x,y) has a mine
  - \textit{isCovered}(x,y) if square (x,y) is covered
  - \textit{isFlagged}(x,y) if square (x,y) is marked with a flag
  - \textit{mineCount}(x,y) is # of mines on neighbors of (x,y)
Things to do

- **Initialization**
  - clear mine counts
  - place mines randomly on board, cover them, clear flags
  - for each mine, increment counts of neighboring squares
  - each of these steps involves iterating through all squares
    - when incrementing mine counts, also iterate thru neighbors

- **Game play**
  - set/clear flag on current square – changes $is\text{Flagged}(x,y)$
  - step on current square – clear $is\text{Covered}(x,y)$ and end game if square contains a mine
  - for uncovered squares with no neighboring mines, uncover neighbors

- **When game over, uncover all squares**
State Diagram

- **newGame=0/**
  - **isCovered(x,y)<=0**
    - advance x,y
  - **isCovered(x,y)<=0**
    - advance x,y

- **clearCounts**
  - **not done/**
    - **mineCount(x,y)<=0**
    - advance x,y
  - **done/**
    - reset x,y
  - **newGame=1/**

- **playTime**
  - **stepOnIt=1** and
    - **isMined(x,y)=1/**
    - advance x,y
  - **flagIt=1/**
    - **isFlagged(x,y)<=1**
    - advance x,y
  - **true/**
    - if **isCovered(x,y)=0**
    - **numMines(x,y)=0**
    - **isCovered(x,y,nbor)<=0**
    - advance x,y,nbor
  - **done/**
    - reset x,y,nbor

- **countMines**
  - **true/**
    - if **isMined(x,y)=1**
    - **mineCount of nbor**
    - advance x,y,nbor

- **setMines**
  - **not done/**
    - **isCovered(x,y)<=1**
    - **isFlagged(x,y)<=0**
    - advance x,y,nbor
entity mineSweeper is port (  
  clk, reset: in std_logic;  
  xIn, yIn: in nibble;  
  newGame, markIt, stepOnIt: in std_logic;  
  level: in std_logic_vector(2 downto 0);  
  hSync, vSync: out std_logic; dispVal: out pixel);  
end mineSweeper;  
architecture al of mineSweeper is  
-- state of game controller  
type stateType is (clearCounts, setMines, countMines, playTime, gameOver);  
signal state: stateType;  
-- arrays of bits that define state of squares on game board  
-- range extends beyond gameboard to eliminate boundary cases  
type boardBits is array(0 to 15) of std_logic_vector(0 to 11);  
signal isMined: boardBits := (others => (0 to 11 => '0'));  
signal isCovered: boardBits := (others => (0 to 11 => '1'));  
signal isFlagged: boardBits := (others => (0 to 11 => '0'));  
-- mineCount(x)(y)=# of mines in squares y-2 to y in column x  
type countColumn is array(0 to 11) of unsigned(2 downto 0);  
type countArray is array(0 to 15) of countColumn;  
signal mineCount: countArray := (others => (0 to 11 => o"0"));
Main Process

begin
  if rising_edge(clk) then
    if reset = '1' then
      randBits <= x"357d"; -- initial pseudo-random value
      state <= clearCounts; x1 <= x"0"; y1 <= x"0"; nbor <= x"0"
    elsif newGame = '1' then
      state <= clearCounts; x1 <= x"0"; y1 <= x"0"; nbor <= x"0"
    else
      case state is
        when clearCounts =>
          mineCount(int(x1))(int(y1)) <= o"0";
          if x1 /= x"f" then x1 <= x1 + 1;
          elsif y1 /= x"b" then x1 <= x"0"; y1 <= y1 + 1;
          else
            x1 <= x"1"; y1 <= x"1"; state <= setMines;
          end if;
        when setMines =>
          -- place mines at random and "cover" them
          randBits <= random(randBits);
          if randBits < ("0" & level & x"000") then
            setMine(x1,y1,1');
          else setMine(x1,y1,0');
          end if;
      end case;
  end if;
end
setCover(x1,y1,'1'); setFlag(x1,y1,'0');
-- move onto next square
advance(x1,y1);
if lastSquare(x1,y1) then state <= countMines; end if;
when countMines =>
  addToMineCount(x1,y1,nbor);
  advance(x1,y1,nbor);
  if lastSquare(x1,y1,nbor) then state <= playtime; end if;
when playTime =>
  if markIt = '1' then
    -- mark/unmark current cell if it's covered
    if covered(x,y) then
      if flagged(x,y) then setFlag(x,y,'0');
        else setFlag(x,y,'1');
      end if; end if;
    elsif stepOnIt = '1' then
      if covered(x,y) then
        if mined(x,y) then state <= gameOver;
          else setCover(x,y,'0');
        end if; end if; end if;
    clearNeighbors(x1,y1,nbor); advance(x1,y1,nbor);
  when gameOver =>
  setCover(x1,y1,'0'); advance(x1,y1);
  when others =>
end case; end if; end if; end process;
Sample Support Functions/Procedures

-- Return true if square contains a mine, else false
begin impure function mined(x,y: nibble) return boolean is begin
  if isMined(int(x))(int(y)) = '1' then return true;
  else return false; end if; end;

-- Return the number of mines contained in neighboring squares
impure function numMines(x,y: nibble) return unsigned is begin
  report "x,y range violation";
  return mineCount(int(x))(int(y)); end;

-- Return true if x,y is the last square on the board.
function lastSquare(x,y: nibble) return boolean is begin
  if x = x"e" and y = x"a" then return true; ... end;

-- Advance x and y to the next square on board. Wrap around at end.
procedure advance(signal x, y: inout nibble) is begin
  if x /= x"e" then x := x + 1;
  elseif y /= x"a" then x := x"1"; y := y + 1;
  else x := x"1"; y := x"1"; end if; end;

-- Return next pseudo-random value following r
function random(r: word) return word is begin
  return (r(5) xor r(3) xor r(2) xor r(0)) & r(15 downto 1); end;
Simulation Overview
Setting Mines
Counting Mines

- Mines in columns 0-2
- Mine on square 4,10
- Incrementing counts at neighbors of square 5,10
- Mine counts in column 1
Uncovering Safe Squares

uncovering mines after stepping on 1,3

mines in columns 0..6

covering of columns 0..6
Exercises

1. Draw a diagram showing the locations of all mines in columns 1-6 of the example on the previous slide. Identify the squares that should be cleared when the player "steps on" square 1,3. Compare this to the result on the previous slide.

2. Write an implementation of the procedure `advance(x,y,neighbor)` used by the `mineSweeper` component.

3. Write an implementation of the procedure `clearNeighbors(x,y,neighbor)` used by the `mineSweeper` component.

4. How many passes are needed, in the worst-case, to clear all the squares on the board that can be cleared after a player steps on a square? Hint: start by thinking about a board with no mines; how many passes are needed if the player first steps on the square in the bottom right corner (largest x value, largest y value)?
Displaying Game State

- Pre-defined tiles for squares on board
  - covered and un-flagged, covered and flagged
  - uncovered with 0, 1, 2,..., 8 neighbors
  - uncovered with bomb

- Periodically copy tiles to vga display buffer, based on board state
  - 20x20 pixel tiles on 320x240 display
  - 10x14 tiles leaves 20 pixel border on all sides
  - store tile patterns in separate pattern memory
    - 12 patterns, 400 pixels, 3 bits per pixel gives 14,400 bits
  - copy all 140 tiles in fraction of second
    - allow 4 ticks to copy one pixel, so at least 32 ns per tile
    - copying one tile every ms updates screen in 140 ms
Copying Patterns

-- process to iterate through cells for copying to display
-- advance (x2,y2) through range of values periodically
process(clk) begin
    if rising_edge(clk) then
        if reset = '1' then
            timer <= (others => '0');
            x2 <= x"1"'; y2 <= x"1";
        else
            if timer = (timer'range => '0') then
                advance(x2,y2);
                end if;
            timer <= timer + 1;
        end if;
    end if;
end process;

-- copy pattern for x2,y2 to the position on the display
cp: copyPattern port map(clk,reset,startCopy,highlight, pat,x2,y2,busy,hSync,vSync,dispVal);
startCopy <= '1' when timer = (timer'range => '0') else '0';
highlight <= '1' when x2 = x and y2 = y else '0';
pat <= pattern(x2,y2);
Pattern Function

-- Return the appropriate pattern number for square x,y
impure function pattern(x,y: nibble) return nibble is begin
  if (not covered(x,y)) and (not mined(x,y)) then
    return "0" & std_logic_vector(numMines(x,y));
  elsif covered(x,y) and (not flagged(x,y)) then
    return x"9";
  elsif covered(x,y) and flagged(x,y) then
    return x"a";
  elsif (not covered(x,y)) and mined(x,y) then
    return x"b";
  else
    return x"0";
  end if;
end;
entity copyPattern is port(
  clk, reset : in std_logic;
  -- client side interface
  start, highlight: in std_logic;
  pattern: in nibble;
  x, y: in nibble;
  busy: out std_logic;

  -- interface to external display
  hSync, vSync: out std_logic; dispVal: out pixel);
end copyPattern;
architecture a1 of copyPattern is
component vgaDisplay ...end component;
subtype patRow is std_logic_vector(3*20-1 downto 0);
type patArray is array(0 to 12*20-1) of patRow;
constant patMem: patArray := (
  o"2222222222222222",
  o"2222222222222222", ... );
begin
  curPix <= patPix when hiLite = '0' else not patPix;
  vga: vgaDisplay port map(clk,reset,en,rw,dispAddr,curPix,
      hSync,vSync,dispVal);
process (clk) ... begin
  if rising_edge(clk) then
    en <= '0'; rw <= '0'; -- default values
    if reset = '1' then tick <= (others => '1'); hiLite <= '0';
    else
      tick <= tick + 1; -- increment by default
      if start = '1' and tick = (tick'range => '1') then
        initAddr(x, y, pattern, dispAddr, patAddr, patOffset);
        hiLite <= highlight;
        elsif tick < to_unsigned(4*400,11) then
          -- each step involves copying a pixel from the pattern to
          -- the display buffer; we allow four clock ticks per step
          if tick(1 downto 0) = "00" then
            -- first read from pattern memory
            patFix <= unsigned(patMem(int(patAddr))(int(patOffset) downto int(patOffset-2)));
            en <= '1'; -- write to display buffer during next tick
          elsif tick(1 downto 0) = "11" then
            advanceAddr(dispAddr, patAddr, patOffset);
          end if;
        end if;
        else -- returns circuit to "ready" state
          tick <= (others => '1');
        end if; end if; end if; end process;
    busy <= '1' when reset='0' and tick /= (tick'range=>'1') else '0';
  end all;
-- Initialize address signals used to access the pattern memory
-- and the display buffer

procedure initAdr(x,y,pObj; in nibble; signal dAdr: out dbAdr;
signal pAdr, pOffset: out byte) is
variable row, col: unsigned(2*dAdr'length-1 downto 0);
begin
-- first display address of square x,y is 20*320*y + 20*x
-- display row has 320 pixels and pattern extends over 20 rows
row := to_unsigned(20*320,dAdr'length)
  * pad(unsigned(y),dAdr'length);
col := to_unsigned(20,dAdr'length) * pad(unsigned(x),dAdr'length);
dAdr <= row(dAdr'high downto 0) + col(dAdr'high downto 0);
pAdr <= pad(sl(20,8) * pad(pattern,8),8);
pOffset <= sl(59,8);
end;

-- Advance address signals used for pattern memory, display buffer

procedure advanceAdr(signal dAdr: inout dbAdr;
signal pAdr, pOffset: inout byte) is
begin
  if pOffset = 2 then -- reached end of pattern row
    pAdr <= pAdr + 1; pOffset <= sl(59,8);
dAdr <= dAdr + to_unsigned(320-19,dAdr'length);
else -- continue along the row
  pOffset <= pOffset - 3; dAdr <= dAdr + 1;
end if; end;
Copying Pattern

- Copying pattern 5 to square 5,6
- First display address at 6*20*320 + 5*20
- Pattern 5 starts at row 5*20 in pattern memory
Exercises

1. Write an implementation of the \( \text{pad}(x, \text{length}) \) function used by the \text{copyPattern} component to produce a “padded” version of a \text{std_logic_vector} \( x \), that has a specified number of bits.

2. If the timer register in the \text{mineSweeper} component that is used to control the copying of patterns has 16 bits, how long does it take to fully update the display buffer? Approximately how many times per second does the \text{vgaDisplay} interface read-out process skip a pixel, because of a write from the \text{copyPattern} block? How does this compare to the total number of times the read-out process attempts to read a pixel from the display buffer.