

Final Project Report: Chrome Dino

Cathy Wang and Samantha Whitt

Project Design

For our final project, we created a game modeled after Chrome's Dino Game where users, upon losing connection to the internet, can jump or duck the dinosaur on the left to avoid incoming obstacles such as cacti and pterodactyls. The user's score is determined by a timer that ends if the dinosaur collides with an obstacle, and a single high score reflects the highest score yet achieved with a maximum of 99,999. Visuals for the game are also minimal with a monochromatic grayscale theme.

Our minimum viable product thus stripped the game down to its core functionalities to include:

- A single dinosaur sprite that jumps in response to a single button press (up button)
- A basic background of a line
- A single cactus that periodically moves from right to left
- A running current score based on time
- A game over screen upon collision should the dinosaur not clear the incoming cactus

For additional features, we added:

- Reset game functionality (the ability for users to try again)
- Running high score/setting a new high score
- "Random" cactus obstacles appearing on the screen
- Ducking capability when the users hold down their button (down button)
- Background aesthetics with clouds and a textured floor
 - Clouds moved at a constant speed
 - Clouds were also colored a light gray in comparison to the rest of the black and white visuals
- Levels where each level would increase the speed of incoming cacti
- Dinosaur animations when running, ducking, jumping, or dying

Project Specifications

To incorporate the visual element, we used a 7" TFT LCD Monitor 480x640p that was connected to a Nexys A7 FPGA board via VGA cable. This board served as our processor, programmed in Verilog that combined a 5-stage multi-pipelined processor with a VGA Controller in the overarching Wrapper module.

To integrate our [code](#), we used GitHub where the "main" branch is the final branch, and under "bitstreams" is "final.bit," which is the final product bitstream. For organizational purposes within the code, all elements regarding the combination of these two were placed on the outermost file hierarchy, all processor modules were contained in the "processor" folder, all assets (i.e. static

pictures, MEM files) were contained in the “assets” folder, all bitstreams generated from multiple versions of our code were contained in the “bitstreams” folder, and all assembly code/compiler instructions were contained in the “ECE350F2020 Toolchain” folder.

Special diction throughout the report:

- screenEnd: high for one cycle between frames (signals when the display has finished loading pixels onto the screen)
- VGA Controller: overarching module that controls logic for what to output to the monitor based on game functionality

Inputs and Outputs

The four inputs to the Wrapper module are the FPGA board's up button, reset button, down button, and internal clock. The clock drives the processor and VGA Controller, the up button triggers jumping, the down button, while pressed, maintains ducking, and the reset button lets players play the game again. The outputs all relate to the VGA display: the horizontal and vertical sync signals and the red, green, and blue signal bits. These outputs control what is displayed on the VGA screen.

Processor Changes

Because the jumping mechanism moves through a preset series of y positions in response to a button push, we decided to have our processor drive the dinosaur's movement with register 1 representing its fixed x position and register 2 representing its variable y position (see Assembly Code for more information). Thus, we created two new instructions: one that branches if the IO port is high and one that stalls until screenEnd.

The first is called “bio,” which stands for **branch on I/O**. As a new instruction, its opcode is 11111 and of type jump immediate (JI), since it branches/jumps to the given target depending on a condition where if (I/O = 1) PC = T. This command triggers the processor's fast branching if the I/O port is high, or in other words, when the up button is pressed. Figure 1 demonstrates how it was added to the fast branching logic in code. In order for the processor to evaluate the condition, the up button was sent into it as an input, and because the processor clock was faster than human reaction times, the processor read that input directly, even though the fast branching happens in the decode stage.

```
assign doFastBranch = (fd_bne && ~(fb_by_a == fb_by_b)) //bne & rd != rs
| (fd_bex & (| fb_by_a)) //bex and rstatus != 0
| fd_j | fd_jal | fd_jr
| (fd_jio & io_jump);
```

Figure 1. Fast branch logic with I/O input in Verilog

The second is called “wait” with an opcode of 11100 and of type zero (Z). More information on the new Z-type instruction is found under Assembly Code. Its functionality makes the processor

stall until it detects screenEnd, which we used when updating the dinosaur's y position for jumping. In this way, each update of the jump would be seen by the players. Because the processor already included logic for stalling, the *wait* instruction was added as an or-condition. For implementing when to stop stalling though, the processor used the positive edge of screenEnd that was detected when the D flip-flop holding screenEnd switched from 0 to 1 upon an incoming screenEnd (see Figure 2). This was to ensure that successive wait instructions won't rush through while screenEnd is still high.

```

wire screen_end_hold;
dffe_ref SCREENEND(screen_end_hold, screen_end, clock, 1'b1, reset);

assign stall =
pw_stall |
(
    (dx_opcode == 5'b01000) //load
    & ((rs == dx_rd) //fd_rsb = rs
    || rt == dx_rd //fd_rsb = rt
    && fd_opcode != 5'b00111)) //opcode = store
    | (fd_opcode == 5'b11100 &
    (
        screen_end == 0 |
        (
            screen_end & screen_end_hold
        )
    )
); //if opcode is stall and screen_end is low... stop stalling when screen end is high

```

Figure 2. Stall logic with screenEnd logic in Verilog

Visual Elements

The left image in Figure 7 depicts the Moore FSM which shows the concept for what the dinosaur would look like on the VGA display at different states. When the down button is pressed, the dinosaur alternates between two ducking positions to have an animated run. When the up button is pressed, the dinosaur changes to a jumping position throughout the entire jump. When the dinosaur is neither ducking nor jumping but the game is still going, the dinosaur alternates between two running frames. And when the dinosaur collides with a cactus, it changes to the game over position.

One thing to note is that all the arrows pointing to a certain state have the same combinations of inputs, so this was simplified to a Mealy machine in which the states were broken up into running, ducking, jumping, and the game over states, as depicted in the image on the right of Figure 7. During the running or ducking, the frame we use for animation changes according to the score's least significant bit, making the dinosaur run at the same speed at which the score increases and also making sure that it's always running with the left foot following the right foot and the right foot following the left. Because the inputs are already saved in D flip-flops, no additional logic was needed to save the states in new state D flip-flops because the input D flip-flops could be used to keep track of state. Because of this, the output is just combinational logic used to figure out the frame number read from RAM (Figure 3).

```
// update dinosaur position
assign dino_frame[0] = (dino_y != 275 | curr_score == 0) ? 0 : curr_score[0];
assign dino_frame[1] = (dino_y != 275 | curr_score == 0) ? 0 : ((down & curr_score[0]) | (~down & ~curr_score[0]));
assign dino_frame[2] = (dino_y != 275 | curr_score == 0) ? 0 : (down & ~curr_score[0]);

assign dino_frame_addr = game_over ? 3'd5 : dino_frame;
```

Figure 3. Dinosaur animation following FSM

The running score was kept in a 17 bit register to hold a max score of 99,999. To have the score to also update with the rest of the screen's objects but at a slower pace than screenEnd, we updated the score (+1) on a new screenEnd clock divider that counts to 15, or for 16 screenEnds. As for visualization, the score was broken per digit in order to correctly choose which number from the numbers' RAM. Breaking it into its digits involves non-blocking statements to first copy the score, mod it, and then divide until you get all 5 separate digits as independent frame addresses. The high score worked similarly with its own floating register, and upon game over, was set to the current score and current score frame addresses if greater than the existing high score.

For the cacti obstacles coming towards the dino, we used behavioral verilog and had a fixed y position and a decreasing x position. When the x position reached 10 or when the user pressed reset, the cacti would reset to its original position on the right (550) to simulate scrolling, as shown by Figure 4. Furthermore at every level, or when the current score reached a multiple of 100, the obstacle speed increased, which was dictated by an increasing velocity that decremented the x position on screenEnd.

<pre>// change velocity assign velocity = curr_score / 100 + 2;</pre>	<pre>reg[3:0] screenEndDivider = 0; assign scoreClock = &screenEndDivider;</pre>
---	--


```
assign cacti_update = (cacti_x < 10) ? 550 : cacti_x-1;
always @(posedge screenEnd or posedge reset) begin
  screenEndDivider <= screenEndDivider + 1; // screen divider clock
  if (reset) begin
    cacti_x <= 550;
  end
  else begin
    if (~game_over)begin
      cacti_x <= cacti_update;
    end
  end
end
```

Figure 4. Score clock divider and cacti scrolling code

To simulate random cacti heights, we used the current score % 3 to choose one of the three cacti of various sizes. This provided a random effect because the score was unpredictable at the time the cactus reached the left side of the screen before resetting, especially when the cactus speed would change per level.

Like the real game, the processor game doesn't start until the player makes their first jump (Figure 5). Only then will the cacti scroll, the score increase, and so on.

```

dffe_ref STARTGAME(game_on, up, clk, ~game_on, reset);

if (~game_over & game_on) begin
    // update score & cactus values for gameplay
end

```

Figure 5. Start Game Code

The game ends when the dino and cactus collide. This collision is when a pixel is occupied by both the cactus and the dinosaur, and this collision is stored in a flip flop so the game stays in a game over state until reset (Figure 6). Game over is then used as a signal to freeze the register file in the processor--which holds the trex at whatever position it died on--the cloud and cactus position, and current score. The high score is also updated and the dinosaur image is changed to reflect its defeat.

```

dffe_ref COLLISION(
    game_over,
    ((dinoSquare & dino_data)
     & (cactiSquare & cacti_data)),
    clk,
    ~game_over,
    reset
);

```

Figure 6. Game Over Code

Diagrams

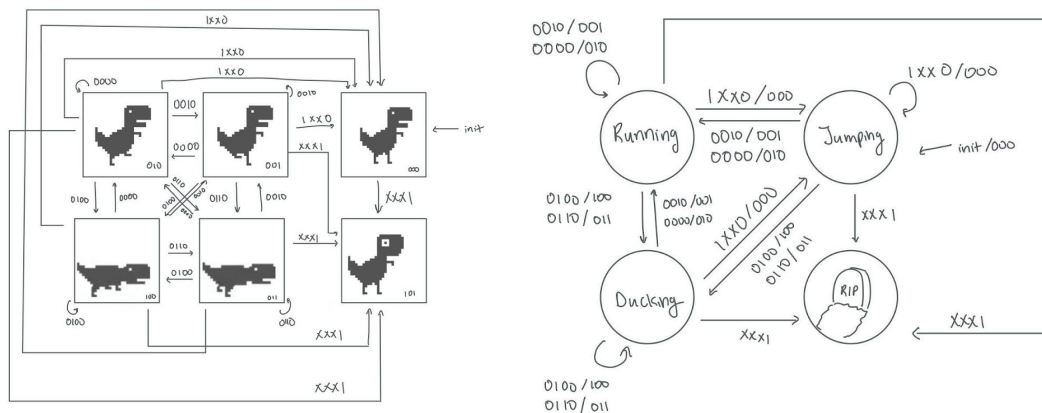


Figure 7. Dinosaur animation frames as a Moore FSM (left) and Mealy FSM (right)

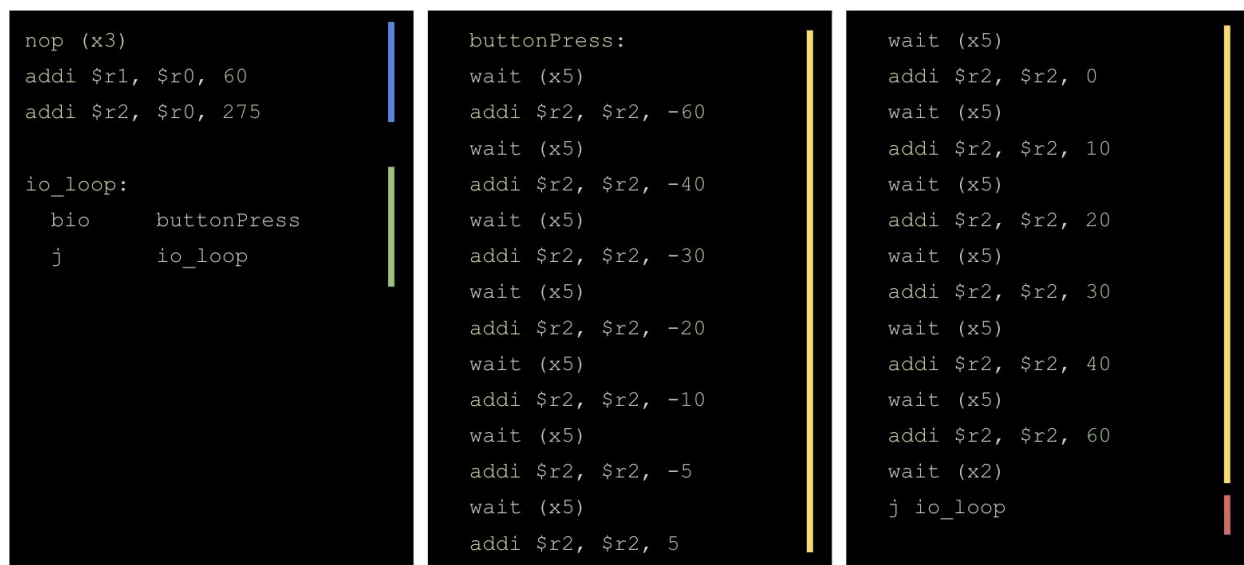
Assembly Code

For our processor game instructions, we handled the dinosaur's position and changed it accordingly to have it jump appropriately, as demonstrated by Figure 8 and will refer to code components according to the color bars.

First, the section with the blue bar sets up the game. We start with nops to make sure we aren't bypassing old values right after a reset, and then we set register 1 to hold the dinosaur's x position and register 2 to hold the dinosaur's initial y position.

The green section is an infinite loop that reads the button IO port. If the button is pressed, the code branches to the yellow section, which changes the dinosaur's y position to look like it's jumping. It "jumps" by decreasing the value almost exponentially, and then increasing it back by the same intervals to give a bell-like jump where the beginning has a fast uptake and a slow airtime at the top. We decrement the values first because the VGA screen goes from 0 to 480 top to bottom.

Finally when the y register's value reaches its initial number, the line with the orange bar jumps back to the infinite loop, where it waits again for a button press. This looping helps to prevent users from being able to jump again when the dinosaur is already in mid-air.



```
nop (x3)
addi $r1, $r0, 60
addi $r2, $r0, 275

io_loop:
    bio    buttonPress
    j      io_loop

buttonPress:
    wait (x5)
    addi $r2, $r2, -60
    wait (x5)
    addi $r2, $r2, -40
    wait (x5)
    addi $r2, $r2, -30
    wait (x5)
    addi $r2, $r2, -20
    wait (x5)
    addi $r2, $r2, -10
    wait (x5)
    addi $r2, $r2, -5
    wait (x5)
    addi $r2, $r2, 5

    wait (x5)
    addi $r2, $r2, 0
    wait (x5)
    addi $r2, $r2, 10
    wait (x5)
    addi $r2, $r2, 20
    wait (x5)
    addi $r2, $r2, 30
    wait (x5)
    addi $r2, $r2, 40
    wait (x5)
    addi $r2, $r2, 60
    wait (x2)
    j io_loop
```

Figure 8. MIPS game instructions

In order to add the two new instructions, we needed to establish them in the compiler to create the game memfile. Because *bio* jumped to a specific location, it was a jump immediate type (J), but *wait* failed was unlike any other instruction, as its functionality was to merely indicate a stall. If we added any other filler information, it would possibly be parsed incorrectly and affect the register file or jump somewhere unwanted. For this, we created a new instruction type called zero, or "Z" for short that had all zeros except for the opcode. In the parsing code of the assembler, it was added as a new case so that none of the other instruction types would trigger and cause errors mentioned above. See Figure 9 below for the specific code on how we added the new instruction types.


```

instruction_t(OPCODE_JIO, OPCODE_ALU_DEFAULT, "bio", J),
instruction_t(OPCODE_WAIT, OPCODE_ALU_DEFAULT, "wait", Z),

struct type_z
{
    unsigned zeros : JMP_ADDR_BITS + REG_BITS;
    unsigned opcode : OPCODE_BITS;
};

union inst
{
    type_i itype;
    type_r rtype;
    type_ji jitype;
    type_jii jiitype;
    type_z ztype;
    unsigned int bits;
};

```

Figure 9. Adding instructions to the header for the assembly compiler

Challenges

Towards the beginning of the project, we noticed that the same bitstream would act differently on different FPGA boards of the same model. After testing with our TA and each of our boards, we went through trial-and-error testing to find some constants that made all the boards behave similarly. Although we never discovered what exactly caused this to occur, our gradual testing led to consistencies that eventually worked for all 100T boards.

Another challenge we faced was our clock timing. In our final project, we had a score clock, our processor clock, and our screenEnd clock -- each at different speeds. The score clock was something that we made by dividing the screenEnd clock such that each for every 16 screenEnds (when the monitor is finished loading all the pixels), the score clock would go high. Because of the many moving components and RAMs in our VGA Controller, making a small mistake of not assigning a bit of a wire or not having the correct length for a register/wire would drastically slow down all clocking. Because testbenches weren't long enough to see effects and Verilog did not register these as syntax errors, we had to debug ourselves by tediously combing through the code and backtracking when we saw the slowed result after generating the bitstream. Although this process seems like an easy fix, it wasn't until the last week that we realized these small errors were the ones that caused the timing to slow down, as before then, no one could guess the reason why. After that discovery, we re-structured our code to have all declarations at the top with comments blocking off parts of code like updating values and outputting to the VGA for better readability.

Testing

Modifying the testbench given for the processor checkpoint let us test the new processor changes. For testing the branch if I/O (bio) instruction, we ran the processor on instructions that had an infinite loop that only included this branch instruction and a branch that changed a register and then jumped back to this loop. Looking at the GTK wave showed that integrating this new instruction with existing fast branch and bypassing logic worked properly. The wait instruction was also tested with the test bench, but we had to modify screenEnd to be on a faster clock than the VGA Timing Generator provides so that the test may run without taking a

long time. Testing this is how we found out we had to have the wait instruction stop stalling on the positive edge rather when screenEnd is high so that consecutive wait instructions won't go through during one screenEnd. Testing also confirmed that the instructions before the wait instruction would finish passing through the processor and that the wait instruction doesn't freeze the entire processor.

For VGA Controller changes, we edited the test bench to visualize the registers and wires throughout the process. However, running the test bench at the fastest clock and extending the simulation time still didn't last long enough to reach a screenEnd, so we weren't able to use this to test our software. Extending the simulation time beyond this meant that the compiling and running the test bench wouldn't be done in over two hours, so we switched to trial and error for testing software components because the time between each screenEnd is too long to test by waveform.

We additionally tested small logic based components that didn't rely on visualization in an outside module with new testbenches. Examples of things tested this way was getting the digits of our score, generating random numbers, and getting the dinosaur FSM outputs.

Improvements

Additional features we wanted to incorporate include adding pterodactyls, multiple clouds, different jump heights, and more colors to simulate background, midground, and foreground. However in order to achieve these, the first improvement would be cleaner code. As mentioned in our Challenges and Testing sections, many problems arose from minor mistakes that would cause multiple setbacks. The next step would be to utilize local parameters rather than hard coding constants (ex. VGA display locations, memfile widths, etc.), establishing a system for muxing versus if statements with behavioral verilog, and simplifying multiple RAMs. Once we have a stronger foundation that's easier to troubleshoot, we'd then move on to implementing those new features. Specifically, we wanted to use our processor more, like moving the obstacles logic there or adding different loops for different jump heights.

Pictures

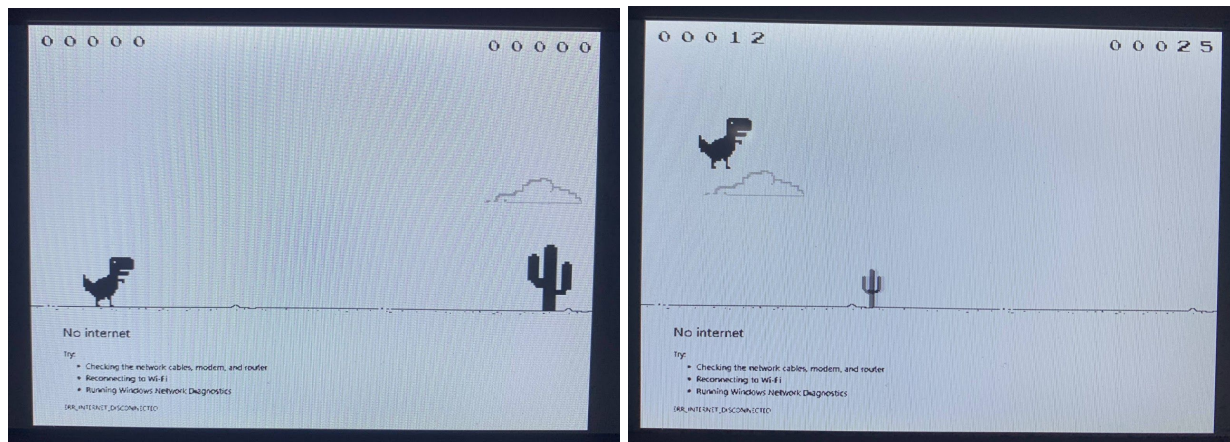


Figure 10. Game play starting and jumping

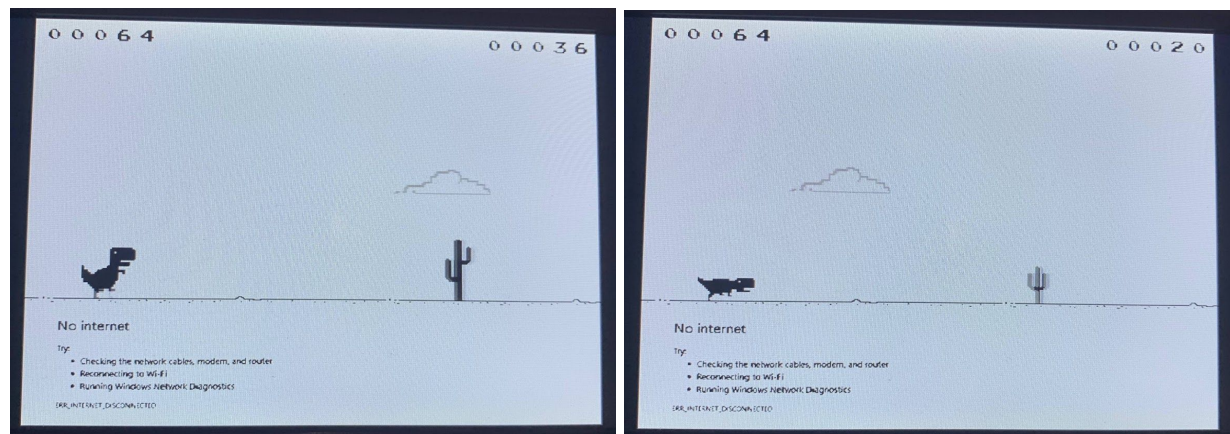


Figure 11. Animated dinosaur running (upright and ducking)

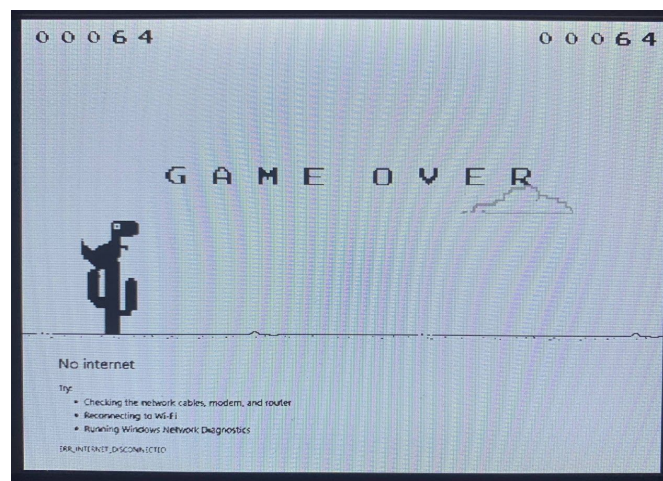


Figure 12. Game over screen with new high score with frozen dinosaur and cloud positions