1 Design Procedure

In this lab we have decided to make an original side-scroller game, on the DE1-SoC board! The game is called ***Dyno's Adventure***. Dyno's Adventure's main character is Dyno, a brave knight who lives in a very dangerous world. The game has two screens - the menu screen and the game screen. In the menu screen the user can enable/disable sprite animations and enable/disable sound effects before starting the game. Sound effects occur on level-up, on collisions, and when the player is walking (footsteps). In the game screen is where the actual game happens. The user is responsible for controlling a character on the screen by pressing the down arrow key (to duck) and the space bar (to jump), and the goal of the player is to live as long as possible dodging different obstacles. The game gets faster and harder, and more monsters appear after every level. The user knows they have passed a level when they hear a level-up sound effect. The score increases over time too. There are also two cheat-codes. One is "peace" to disable collisions, and one is "chill" to set the difficulty to that of level 0. Before getting to the complete and detailed top-level diagram of our implementation of the game, we will start with high-level abstractions. First is a high-level abstraction of the game, and a functional diagram with a brief explanation of how the big components work:

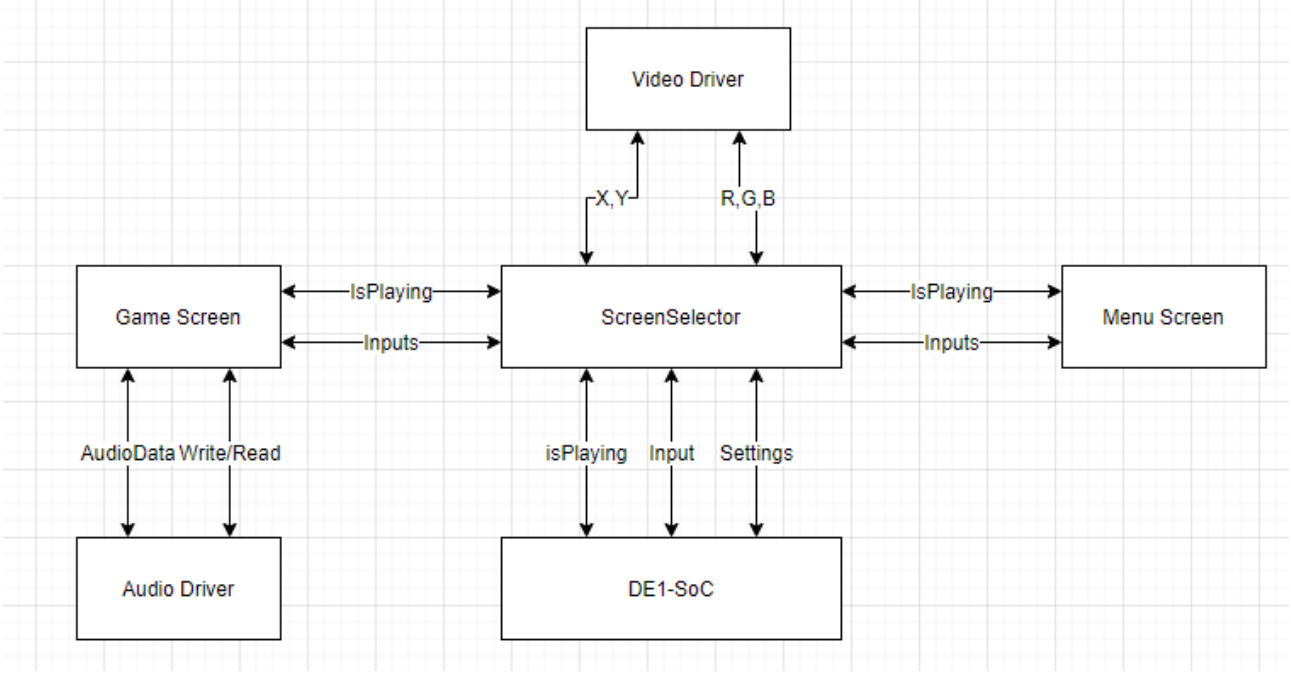
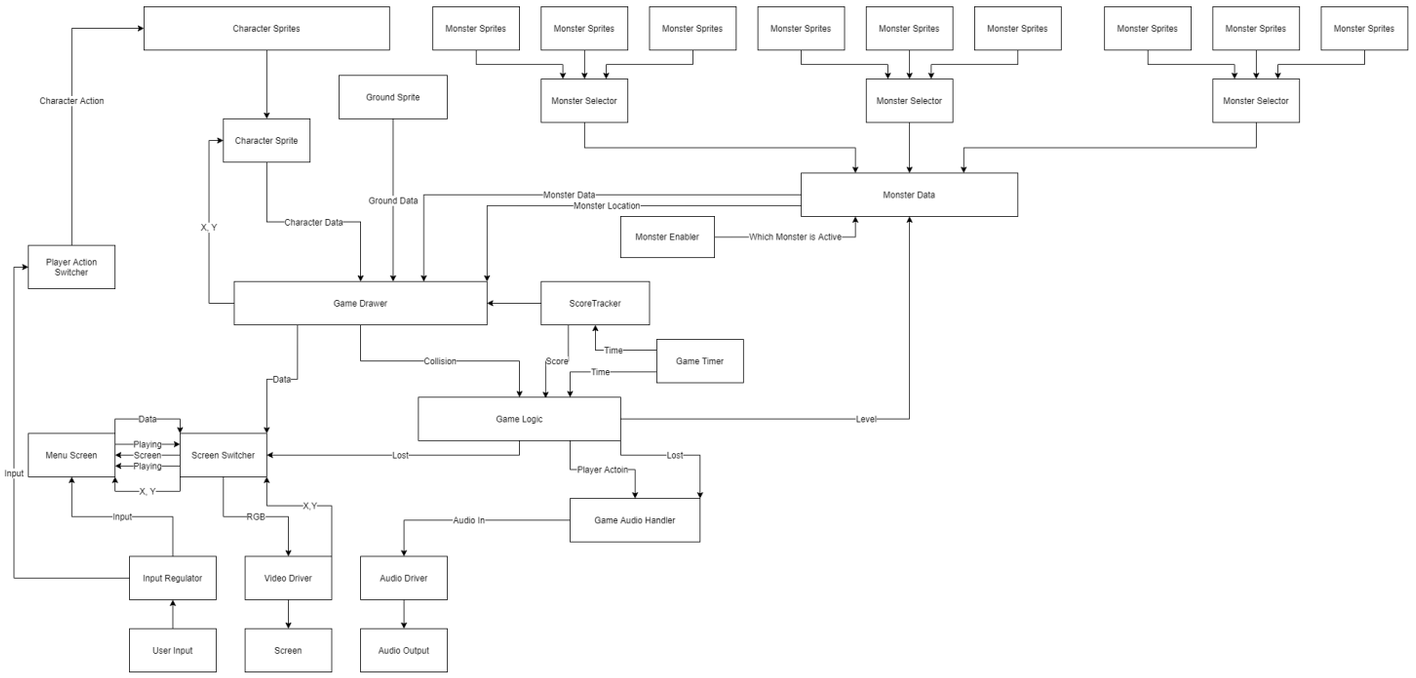
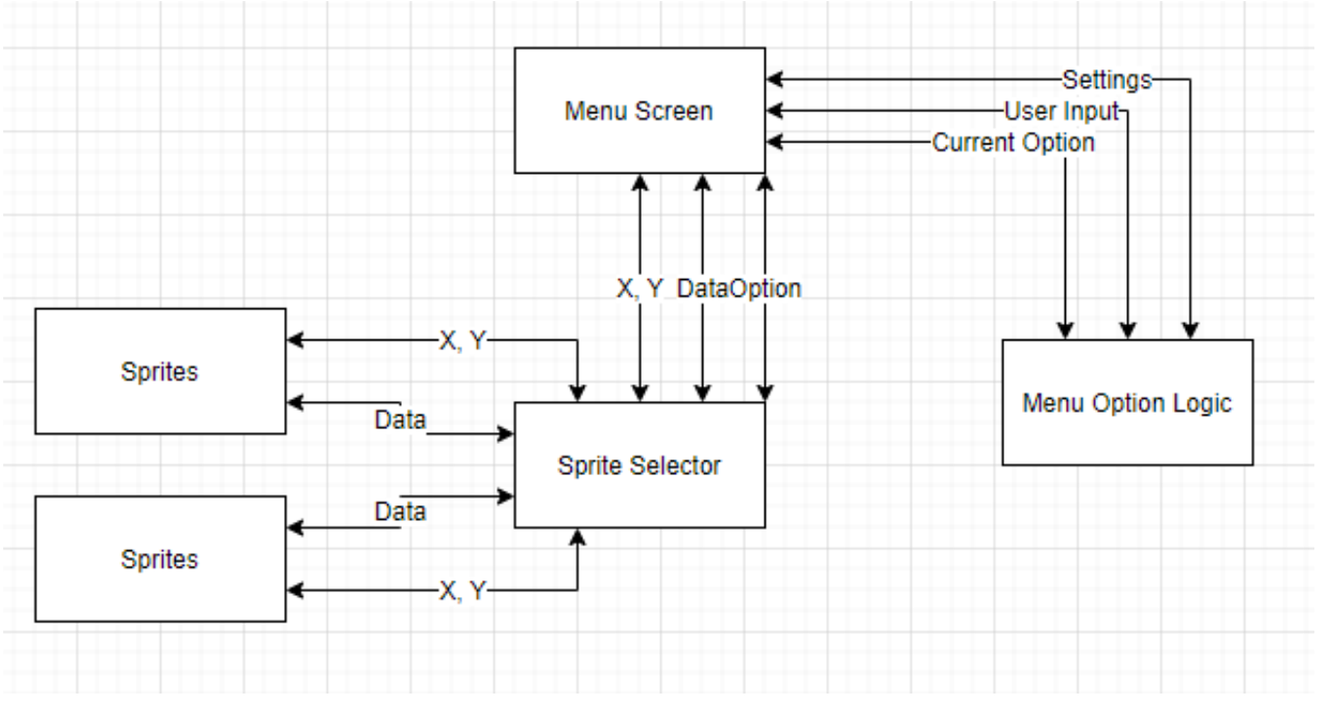


Figure 1: High-level Abstraction Diagram for the Game



Functional Diagram

In order to achieve the screen switching and object drawing logic, we used the given video driver, and a global screen identifier signal to decide which output data is shown on the VGA display. The game elements and the menu both know to ignore input when inactive. When a respective screen is active, all its elements respond to inputs and update their behavior over time. The down key arrow, up key arrow, and enter key are used to navigate the menu. Once the user selects the option to start the game, the menu stops ignoring input until the user goes back to the menu screen by pressing 'm' after they have lost.

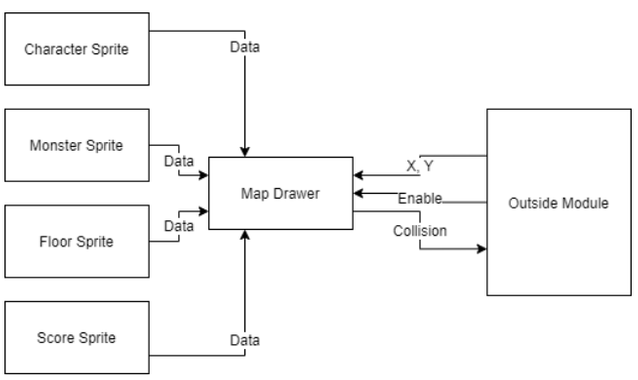


Block Diagram for the Menu

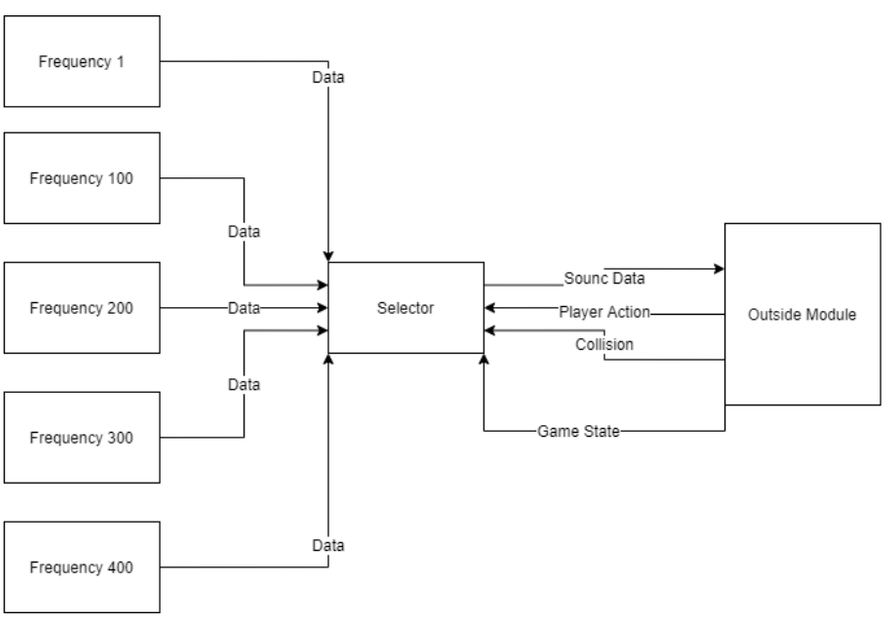
The game screen uses similar logic for game progression handling: It separates the game state with visual representation, the game progresses only depends on the user input and which

screen is active, the visual output is simply another module that gives the user and idea of the current game state. In addition to the video driver, the game makes use of audio. Similar to how we handle the drawing in menu, the main game module will export useful information such as main character's movement status, player's input, collision. Sound settings are provided by the menu settings. The map-drawing module will read in monster pixel data for every monster, character data, and ground data as drawing information. During the game, for each given pixel provided by the video drivers, *map_drawer* will check the state of each monster and of the main-character dyno, finding the pixel value at that point, and then determine if there is any collision, and what color shall be shown on the screen.

The audio module is also responsible of taking information about the state of the user to provide effects. The sound module needs to know about collisions to produce the game lost effect, when the user is walking to produce footsteps, and when the user advances a new level to produce a level-up sound effect. At each given moment only one sound will be played, and each sound-effect might make use of various square-wave signals in sequence. The block diagrams for sound module and game drawer module are shown below. The notation "Outside Module" refers to signals coming from the external world as seen from the module in question.

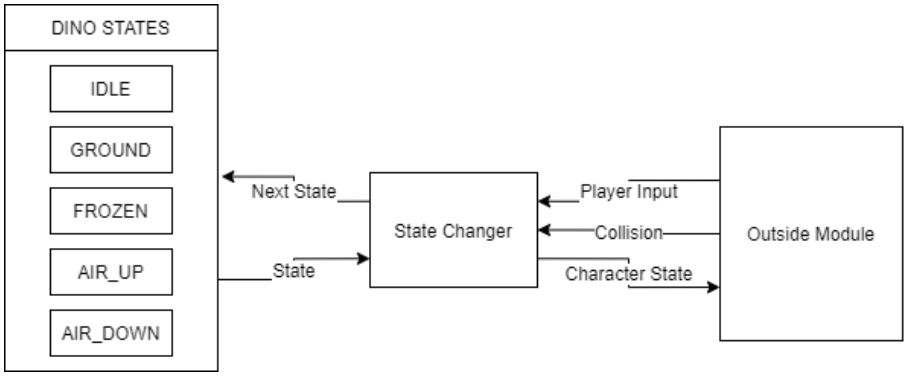


Block Diagram for Drawer Module

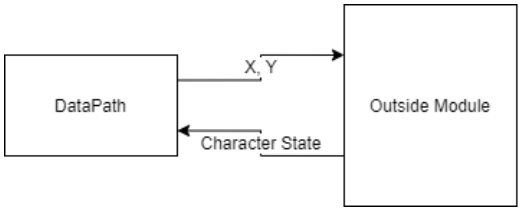


Block Diagram for Sound Module

In order to achieve the player logic, there are few states that a player could achieve. They are jumping, idle, moving, ducking, and mid-air. Those state transition are being handled in the player control module. The module will output flags to its data path module that will update the player's x coordinate, as well as the corresponding player sprite. The block diagram for the player module is shown below:

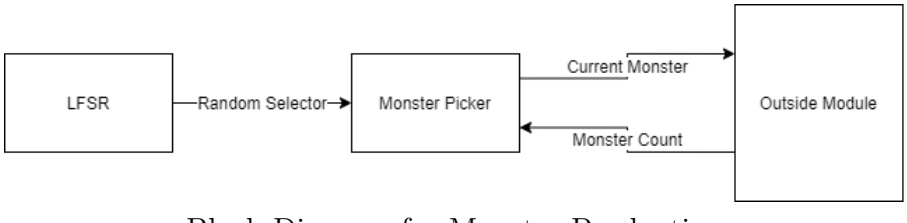


Block Diagram for Player Control Module



Block Diagram for Player Data path Module

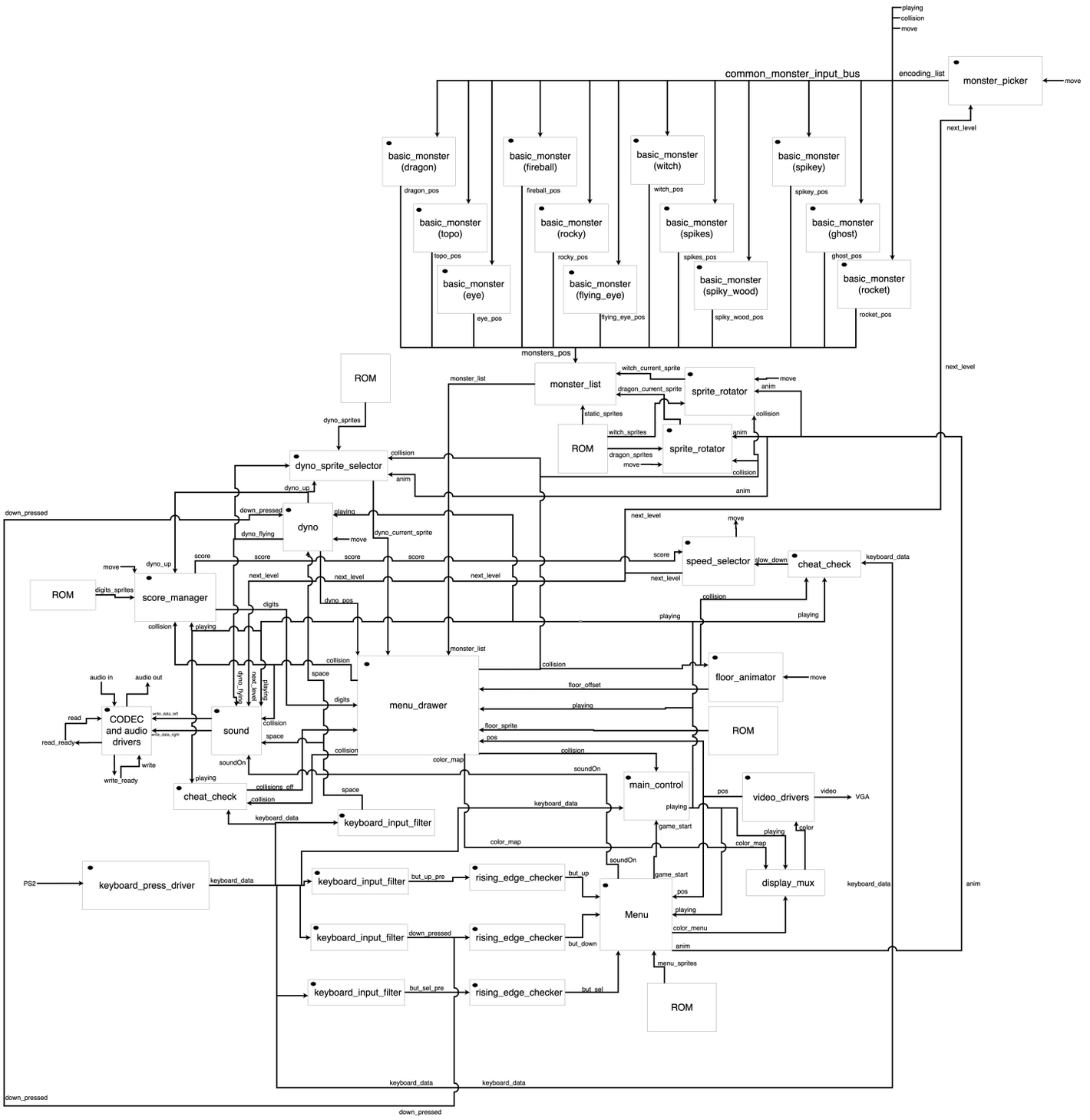
As contrast to the player module, there is the monster module. The monster module is responsible for picking a monster and updates its location throughout the game-play. The number of monster spawned should increase as the game gets more and more difficult. After picking a monster, the data will them be used by the drawer and then put them up on the screen. Below is the block level diagram for monster production:



Block Diagram for Monster Production

1.1 Top-Level Diagram

The following is the top-level block diagram of the implementation of our game on the DE1_SoC board:



Top-level diagram for Dyno's Adventure

The following is the legend for signals that were logically grouped into buses to avoid clutter in the diagram:

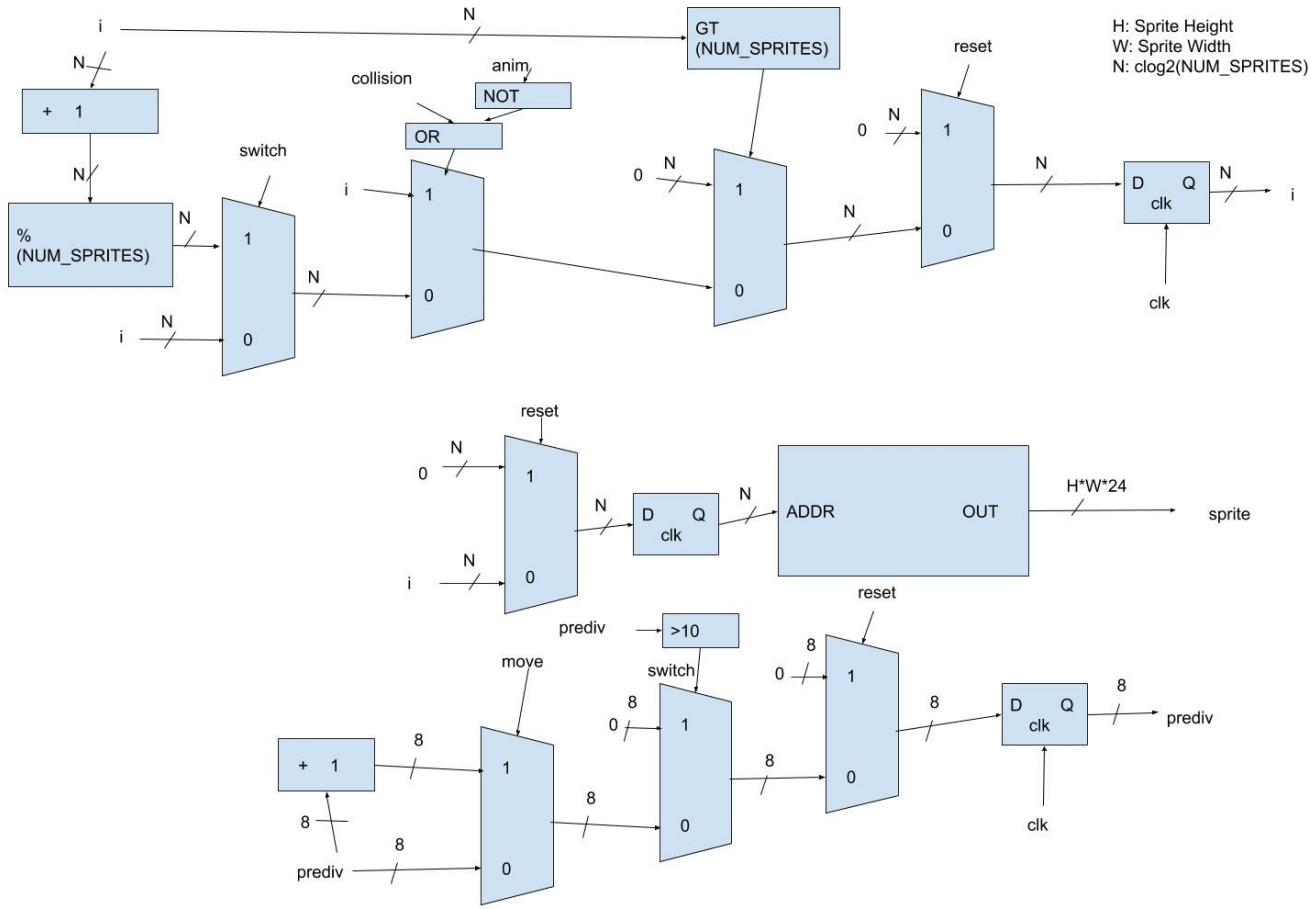
1. The black dot at the top-left of many modules indicates those modules take the clock (CLOCK_50) and reset (~KEY[0]) as inputs.
2. common_monster_input_bus: Groups *playing, collision, move, and encoding_list.*
3. keyboard_data: Groups *valid, outCode, makeBreak.*
4. menu_sprites: Groups sprites coming from pointer_mem, option_mem, on_mem, off_mem, back_mem, and menu_mem.

5. color_menu: Groups *r_menu, g_menu, b_menu.*
6. color_map: Groups *r_map, g_map, b_map.*
7. color: Groups *r, g, b.*
8. VGA: Groups VGA signals to the external world.
9. audio_out: Groups audio signals to the external world.
10. audio_in: Groups input audio data from the external world (unused, but required by the CODEC's interface).
11. digit_sprites: Groups digitX with X from 0 to 9 (inclusive).
12. digits: Same as digit_sprite_list.
13. pos: (x, y) provided by the video drivers.
14. dyno_pos: Groups *dyno_x* and *dyno_y.*
15. dyno_sprites: Groups *dyno_running_spriteX*, *dyno_flying_sprite*, *dyno_duckingX_sprite.* With X being 1 and 2.
16. witch_sprites: Groups *witch_sprite0, witch_sprite1.*
17. <monster_name>_pos: Groups <monster_name>_x and <monster_name>_y.
18. monsters_pos: Groups all monster positions.

1.2 Individual Modules

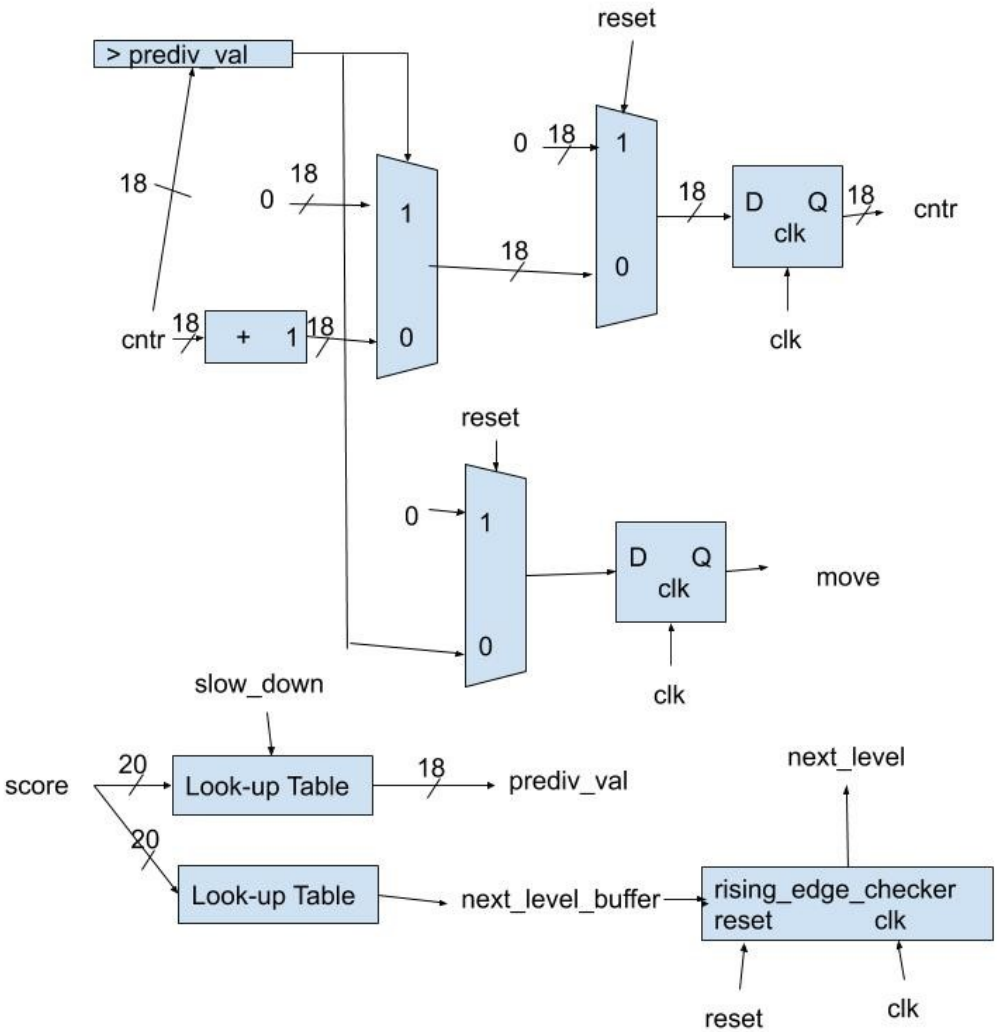
1.2.1 sprite_rotator

This module is responsible for rotating the incoming sprite. The module sprite_rotator changes the sprite of a monster that has more than 1 sprite. For example, the dragon monster has 3 sprites flapping its wings in different positions. The sprite is held constant when a collision occurs, or if animations are turned off (*collision* is HIGH, or *anim* is LOW). Otherwise, a counter and a pre-divider are to select a sprite from the ROM. The following is an implementation of this module:



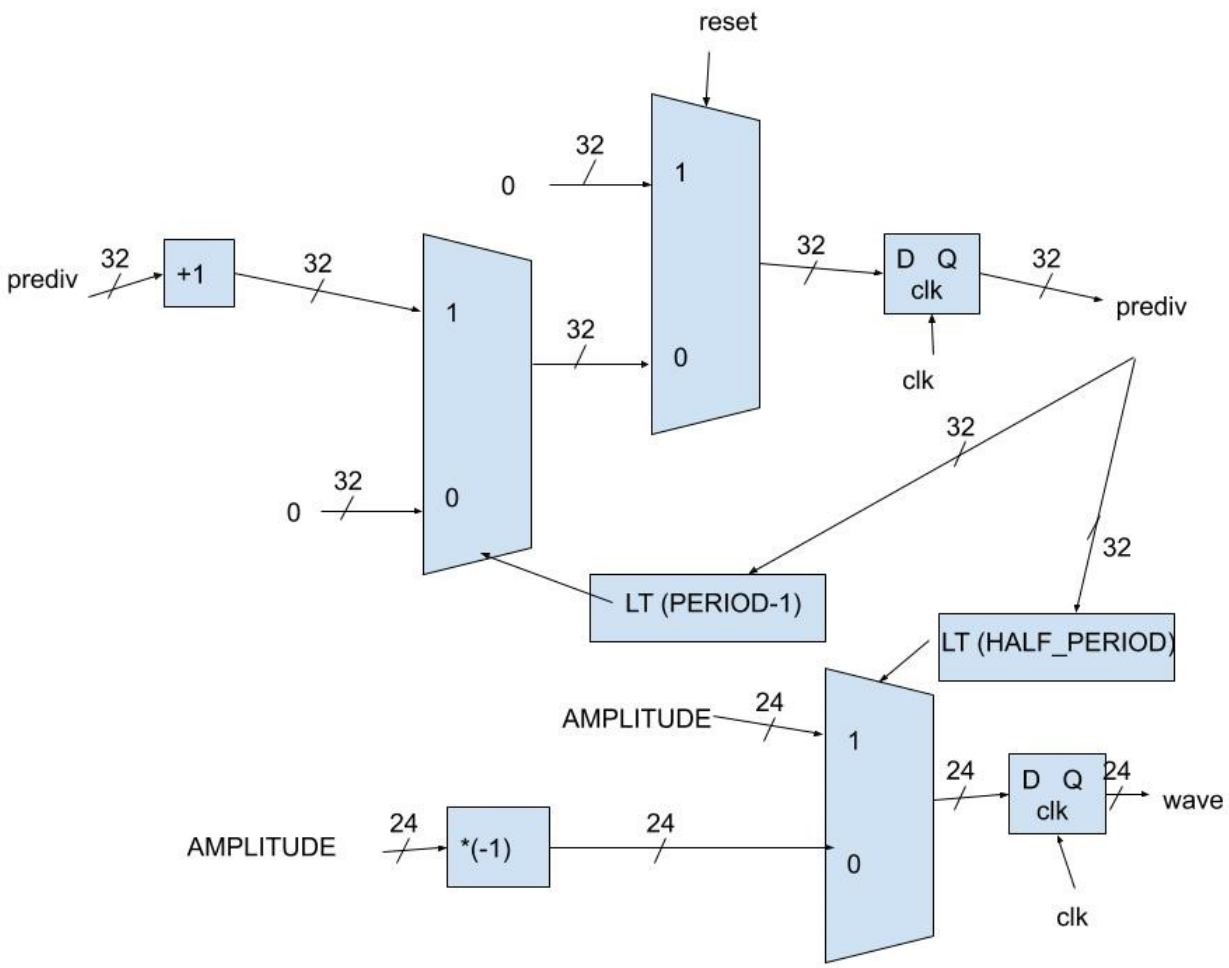
1.2.2 speed_selector

This module is responsible for control the game speed through generating clock signals at different period. It will get faster as the score of the player gets higher and higher. This is done by producing the *move* signal that synchronizes the game to a speed much lower than CLOCK_50 and whose period depends on the score. If the *slow_down* signal is HIGH, it is because the *chill* cheat-code is enabled, which sets the speed equal to that of level 0.



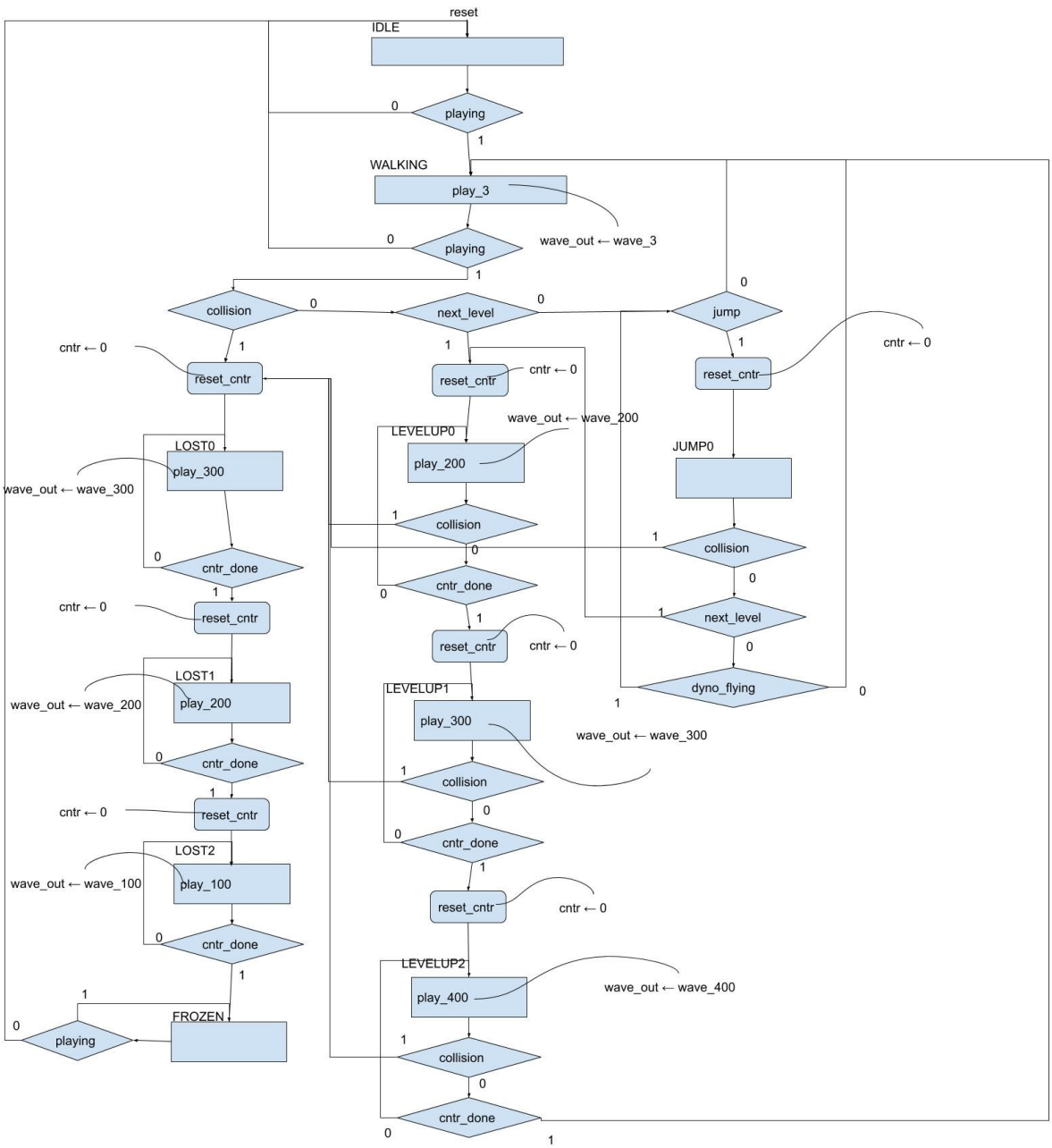
1.2.3 square_wave_sound_gen

This module is responsible for generating sound that will be passed into the sound module. A square-wave generator is produced by a parameter AMPLITUDE and a desired frequency. Using the AMPLITUDE and the desired frequency we can calculate the period: $PERIOD = \frac{50{,}000{,}000}{FREQ}$ and $HALF\_PERIOD = \frac{PERIOD}{2}$. A counter can be used to produce a certain frequency, and a comparator can be used to decide when the output wave should be AMPLITUDE or -AMPLITUDE. The following is a possible implementation of this module (LT stands for Less-Than):



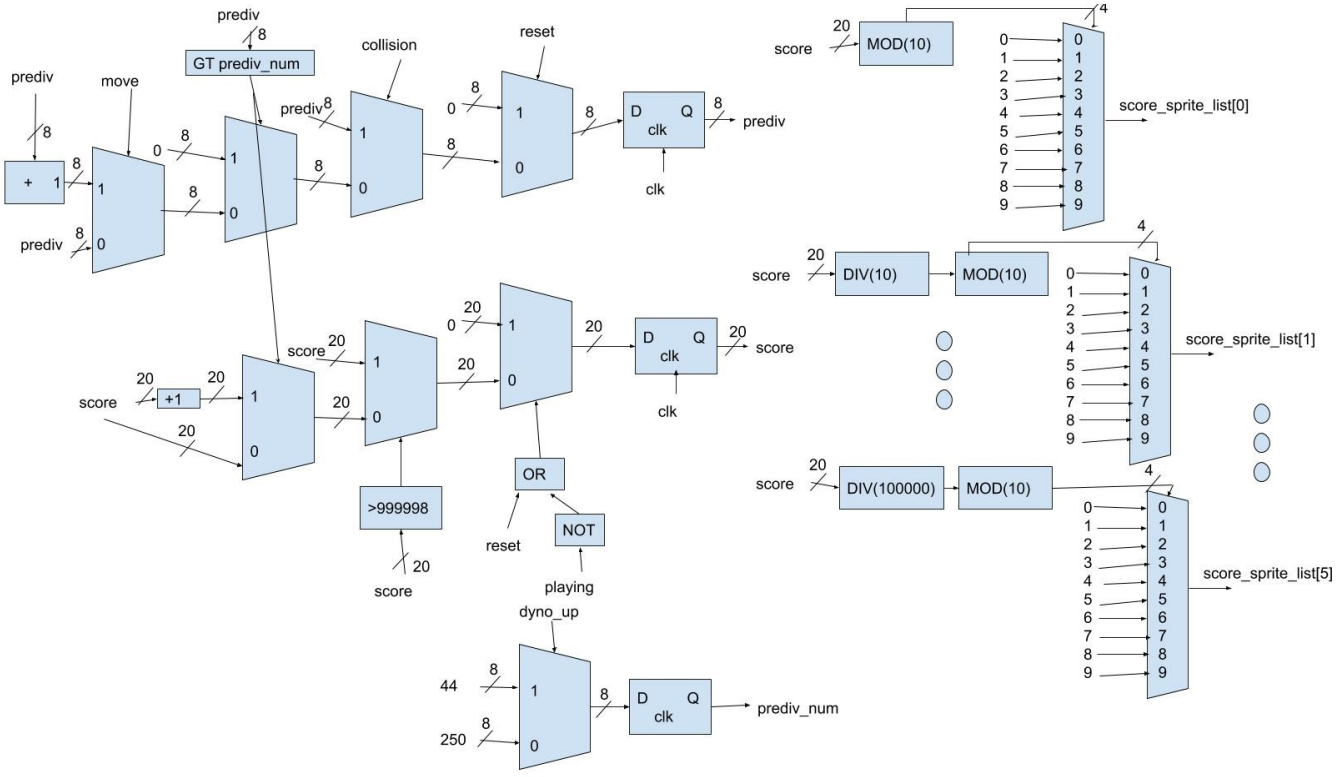
1.2.4 sound

This is the sound module that is responsible for handing of sound data to the audio driver during the game. It can also be enabled or disabled through the menu. Given *soundOn*, a simple multiplexer inside the datapath decides if it should output sound or not. The implemented sound effects are (i) footsteps (3Hz), (ii) level up (sequence of increasing frequency from 200Hz to 400Hz), and (iii) losing (sequence of decreasing frequency from 300Hz to 100Hz). The ASMD chart is shown next:



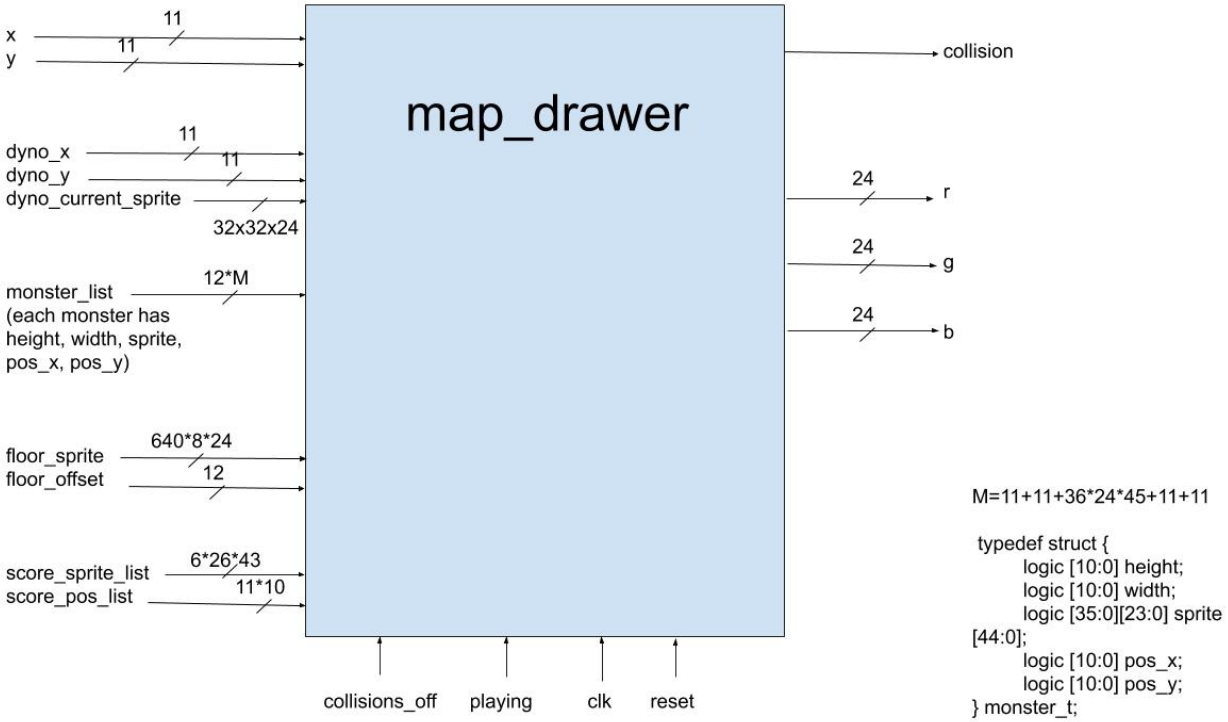
1.2.5 score_manager

This module is responsible for calculating the score of the user. It uses the *move* signal from the speed_selector module in order to make the score increase with the speed/difficulty of the game. It also produces a list of sprites that map_drawer uses to print the score on the top-right section of the screen.



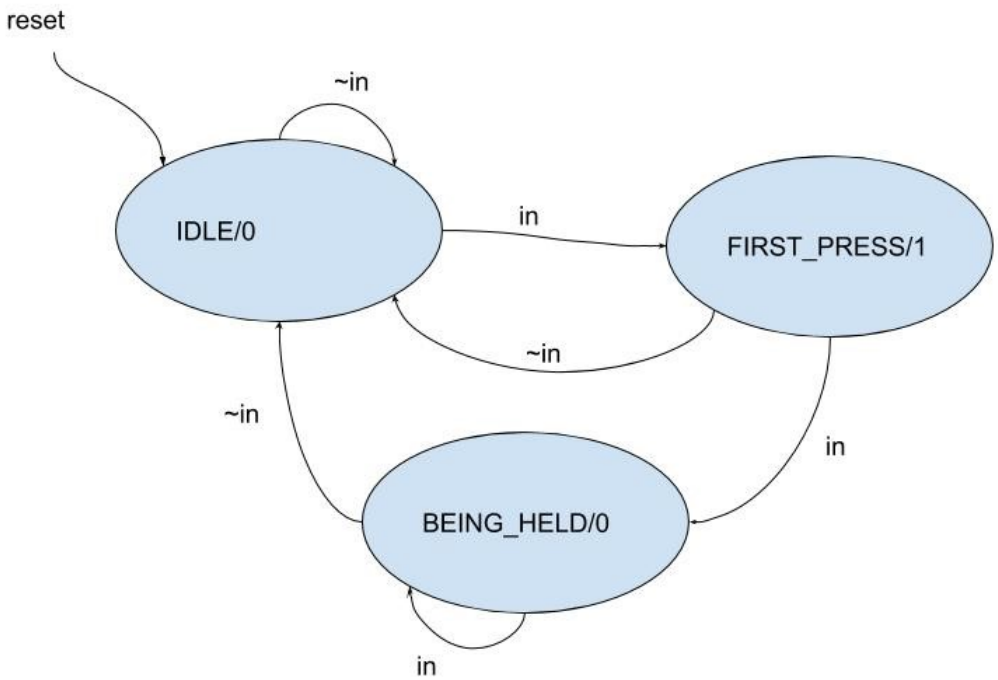
1.2.6 map_drawer

This module has the responsibility of drawing each sprite on the map, and in the process, it checks whether a monter's pixel is overlapping with dyno's pixel to conclude whether there has been a collision. The image is producing by checking the values of x and y produced by the video-driver, and supplying the correct r, g, b triplet. The r, g, b triplet is produced by looking at each sprite (monster, dyno, floor, and score) along with their position and dimensions, checking whether the pixel lies in their area, and checking if the sprite has a pixel in that location. To differentiate between sprite pixels and void space in a sprite, a special value of 0xFF is used for r, g, and b. Thus if a sprite needs to have a black value, it must use 0xFF-1 or any other value, which to the eye doesn't make any difference between real black. A triplet where r, g, and b are all 0xFF is reserved for a void space, which allows map_drawer to detect collisions with great resolution for a very small cost in logic.



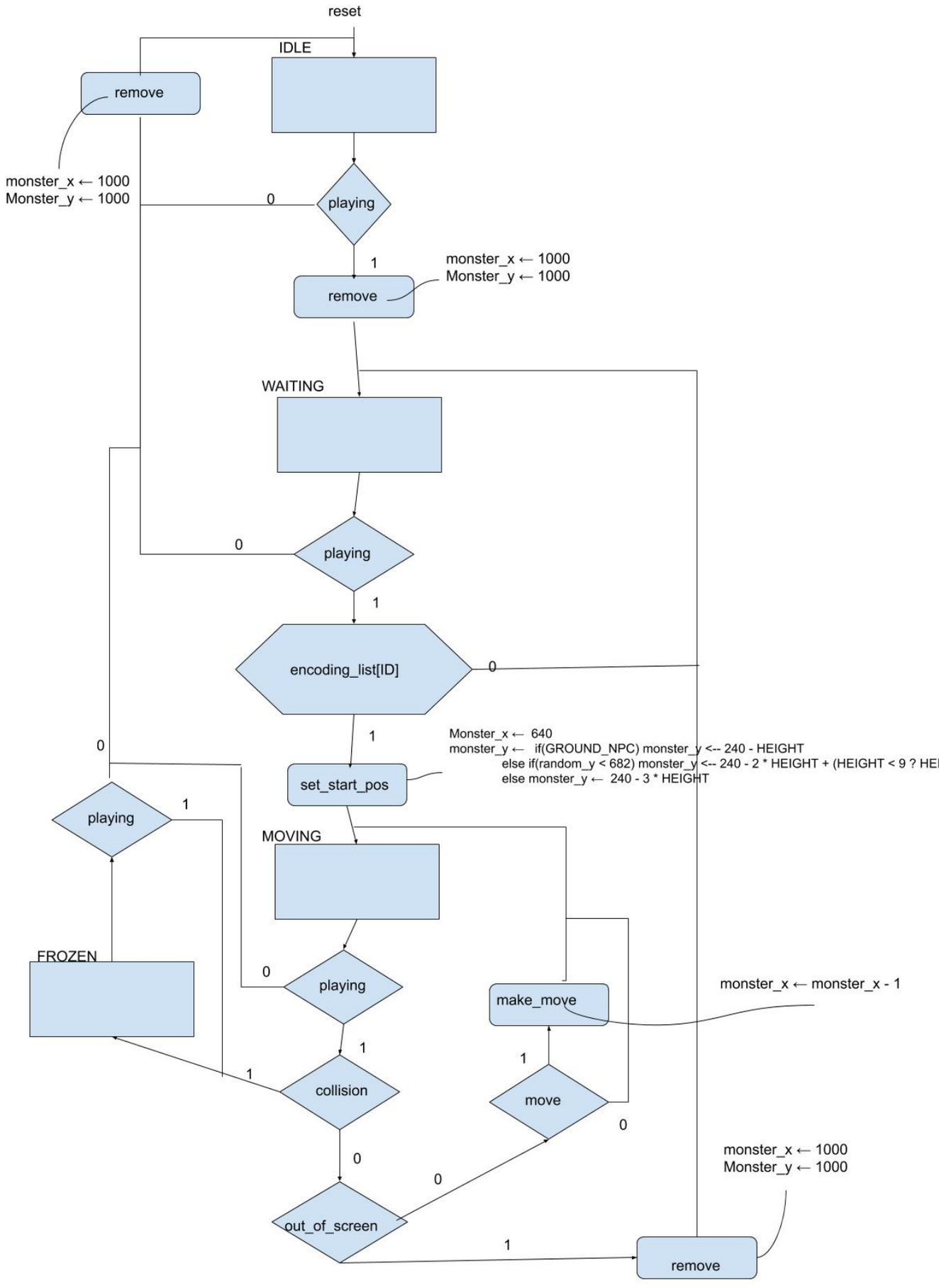
1.2.7 rising_edge_checker

This module is to regulate inputs from mostly user to make sure that each button press only triggers the event once. This is used by speed_selector to check when a new level is reached, and by the top-level DE1_SoC to process key up, down, and the enter key.



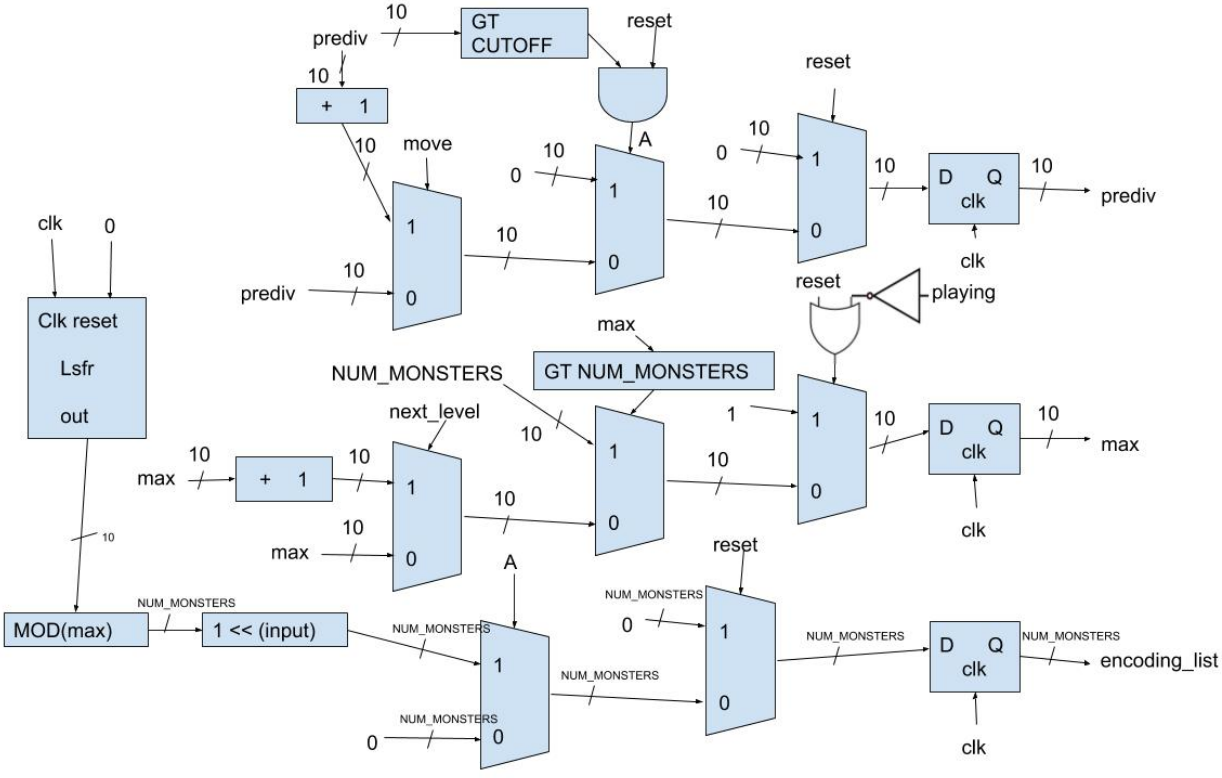
1.2.8 basic_monster

The monster controller is responsible for representing different states that a monster can have. They are IDLE (waiting for the game to start), WAITING (waiting to go into screen), MOVING (on-screen), and FROZEN (after a collision). The basic_monster module can be used with any sprite to provide a monster to the game. The behavior of this module is captured in the ASMD chart below. Some important characteristics are that a monster is always on the map, but is only on a visible area when its ID is pulsed HIGH by monster_picker on encoding_list[ID]. There is a parameter GROUND_NPC (active-high) which indicates if the monster shall be on the ground or it is airborne.


reset
IDLE
remove
monster_x ← 1000
Monster_y ← 1000
0
playing
1
monster_x ← 1000
Monster_y ← 1000
remove
WAITING
0
playing
1
encoding_list[ID]
0
1
Monster_x ← 640
monster_y ← if(GROUND_NPC) monster_y <-- 240 - HEIGHT
else if(random_y < 682) monster_y <-- 240 - 2 * HEIGHT + (HEIGHT < 9 ? HEI
else monster_y ← 240 - 3 * HEIGHT
0
set_start_pos
playing
1
MOVING
FROZEN
0
playing
make_move
monster_x ← monster_x - 1
1
1
collision
1
move
0
0
0
monster_x ← 1000
Monster_y ← 1000
out_of_screen
1
remove

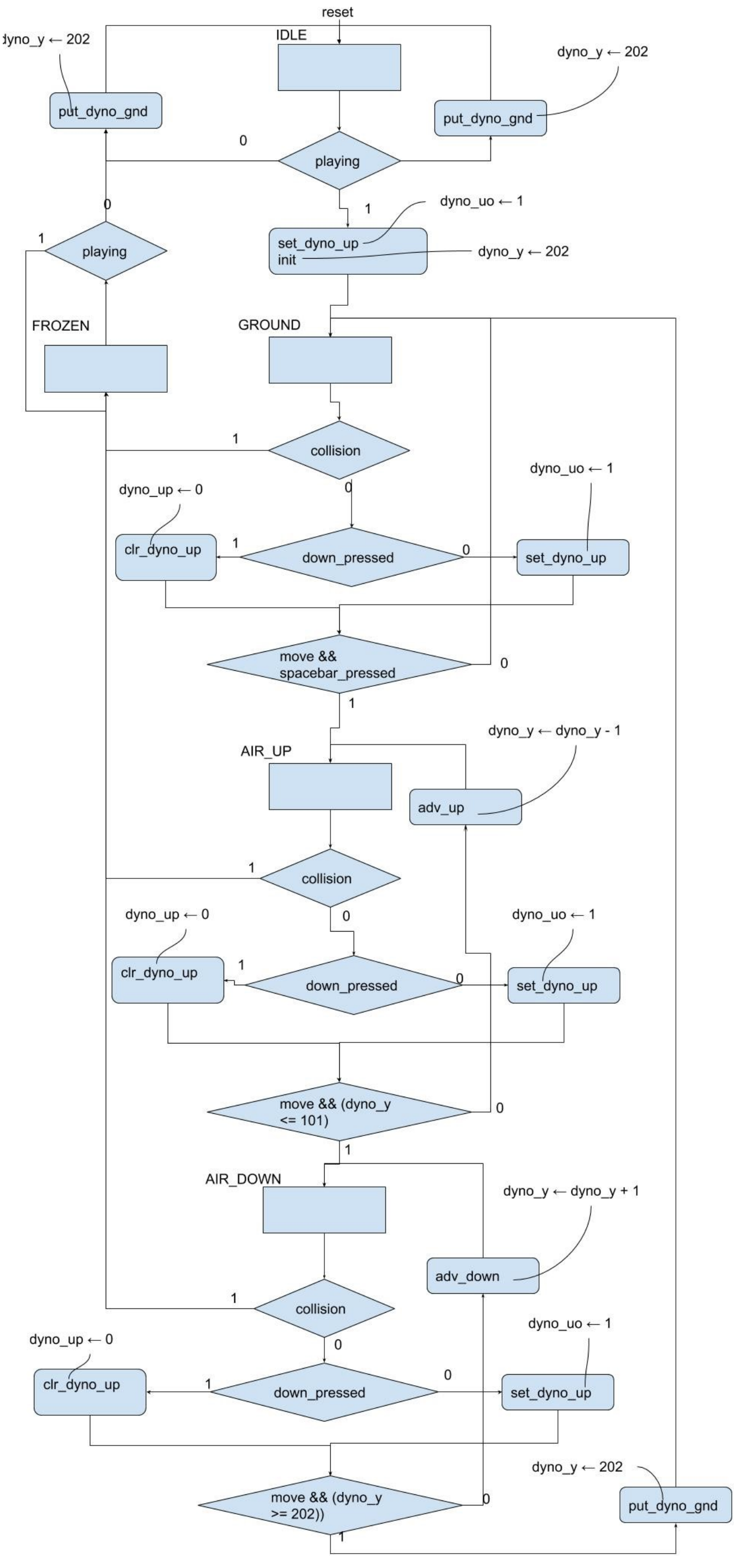
1.2.9 monster_picker

This module uses a random number generator to achieve generating monster onto a map randomly. As the level increases, the number of possible monsters on the board also increases. Technically speaking, the goal of this module is to pulse HIGH for one cycle one of the bits in encoding_list. First, a pre-divider is used to generate a signal that indicates when a new monster should be picked. This pre-divider increases on move, thus the speed of the module increases as move increases. The pre-divider is checked against a fixed value CUTOFF. When this value is met, the pre-divider is reset and a monster is thrown into the map by a pulse on a bit inside encoding_list. The value max determines which monsters are available in the current level. When the next_level signal is asserted (this is converted to a pulse by rising_edge_checker), the value of max is increased by one. Then this module can only pick a monster whose ID is lower than max. An LFSR is used to decide which monster is picked next.



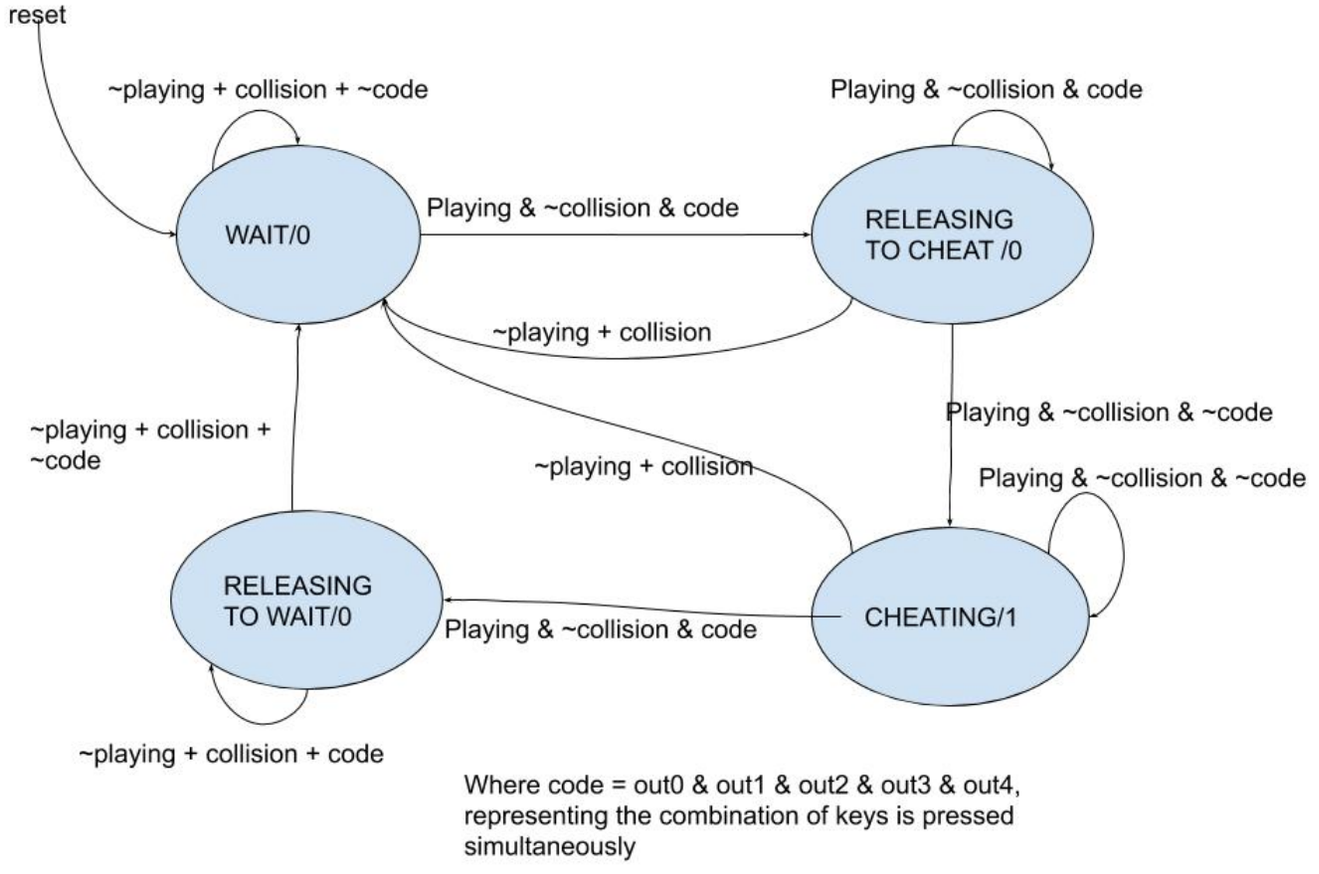
1.2.10 dyno

Dyno is the name of the character in the game. This module manages manages Dyno, based on user input. Dyno can be walking or in the middle of a jump. Furthermore, it can also be ducking or standing, independent of whether it is also walking or in the middle of a jump. The following ASMD chart describes dyno.


reset
IDLE
dyno_y ← 202
put_dyno_gnd
dyno_y ← 202
put_dyno_gnd
0
playing
0
1
dyno_uo ← 1
1
playing
set_dyno_up
init
dyno_y ← 202
FROZEN
GROUND
1
collision
dyno_uo ← 1
dyno_up ← 0
0
clr_dyno_up
1
down_pressed
0
set_dyno_up
move &&
spacebar_pressed
0
1
dyno_y ← dyno_y - 1
AIR_UP
adv_up
1
collision
dyno_up ← 0
dyno_uo ← 1
clr_dyno_up
1
down_pressed
0
set_dyno_up
move && (dyno_y
<= 101)
0
1
AIR_DOWN
dyno_y ← dyno_y + 1
adv_down
1
collision
dyno_uo ← 1
dyno_up ← 0
0
clr_dyno_up
1
down_pressed
0
set_dyno_up
dyno_y ← 202
move && (dyno_y
>= 202))
0
put_dyno_gnd
1

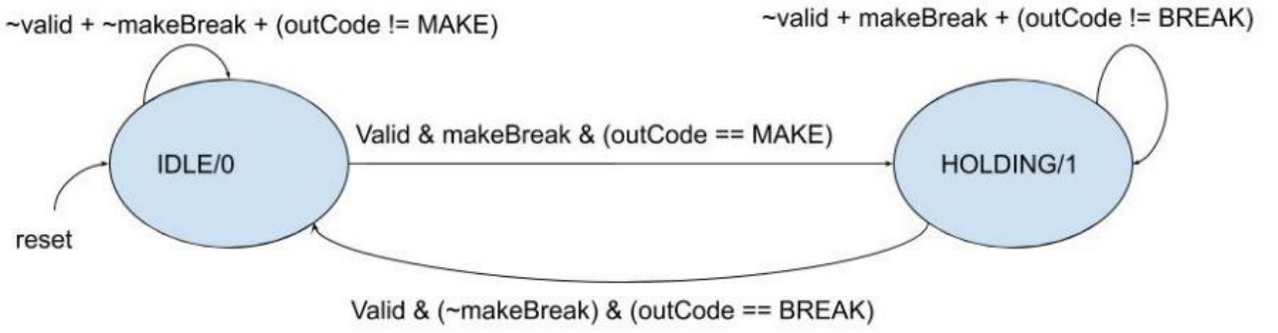
1.2.11 cheat_check

The check check module is there to verify if user have entered the cheat code. If so, then the user can cheat in game. A cheat-code is given by 5 parameters that select the keys that must be pressed for a cheat-code. A cheat-code therefore must be 5 letters or less, and all letters must be pressed at the same time. To de-activate the cheat-code, the user must follow the same procedure for activating a code. A cheat-code only counts for one game, so if the user activates a cheat-code, loses, and re-starts the game, the cheat-code will not be active unless the user explicitly enters the cheat-code again. The cheat-code should only be entered when playing, otherwise it will be ignored (i.e.: if the cheat-code is entered when the menu is displayed, it will be ignored). The implemented cheat-codes in the game are "peace" which de-activates collisions, and "chill" which slows downs the game to the lowest speed level. The module can be captured by the following FSM:



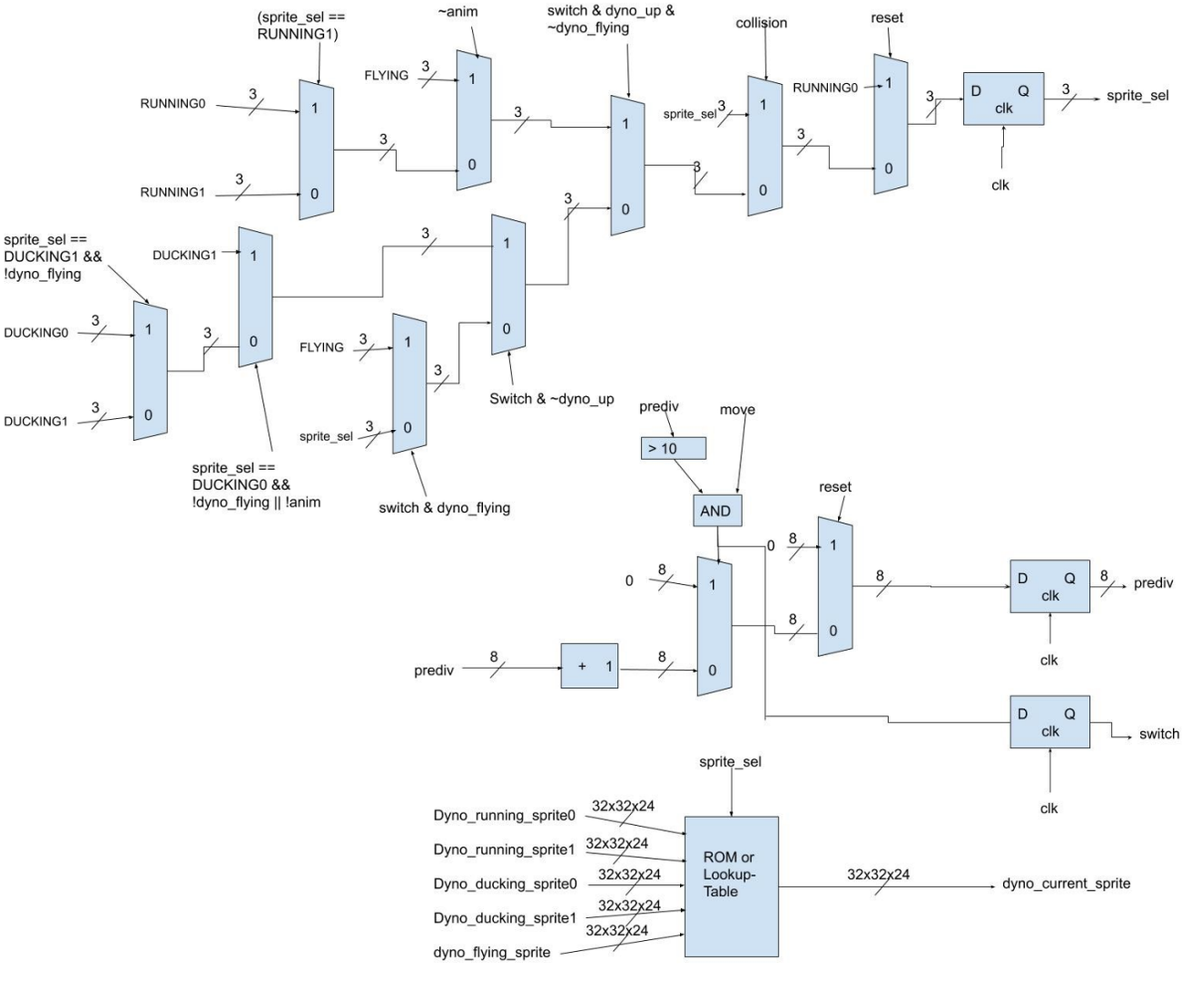
1.2.12 keyboard_input_filter

The keyboard_input_filter wraps the output from keyboard_press_driver with an FSM so that the output is TRUE if and only if a specific key is being PRESSED and FALSE if the key is not being pressed. Two parameters are used to specify which key, these are BREAK for the break-code, and MAKE for the make-code.



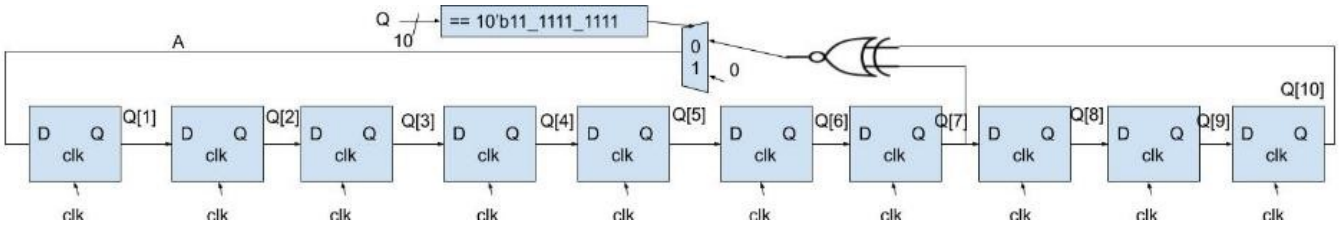
1.2.13 dyno_sprite_selector

This module is responsible for selecting different sprites for drawing dyno based on character's current state. Dyno can be running or jumping, and independent of whether it is running or jumping it can be ducking. The following is an implementation of this module, the sprite for when animations are off are re-used from the regular sprites, and the sprite for jumping whether running or ducking are also re-used from the regular running/ducking sprites. The difference is that the sprite is static when jumping (since dyno is not making any footsteps).



1.2.14 lfsr10

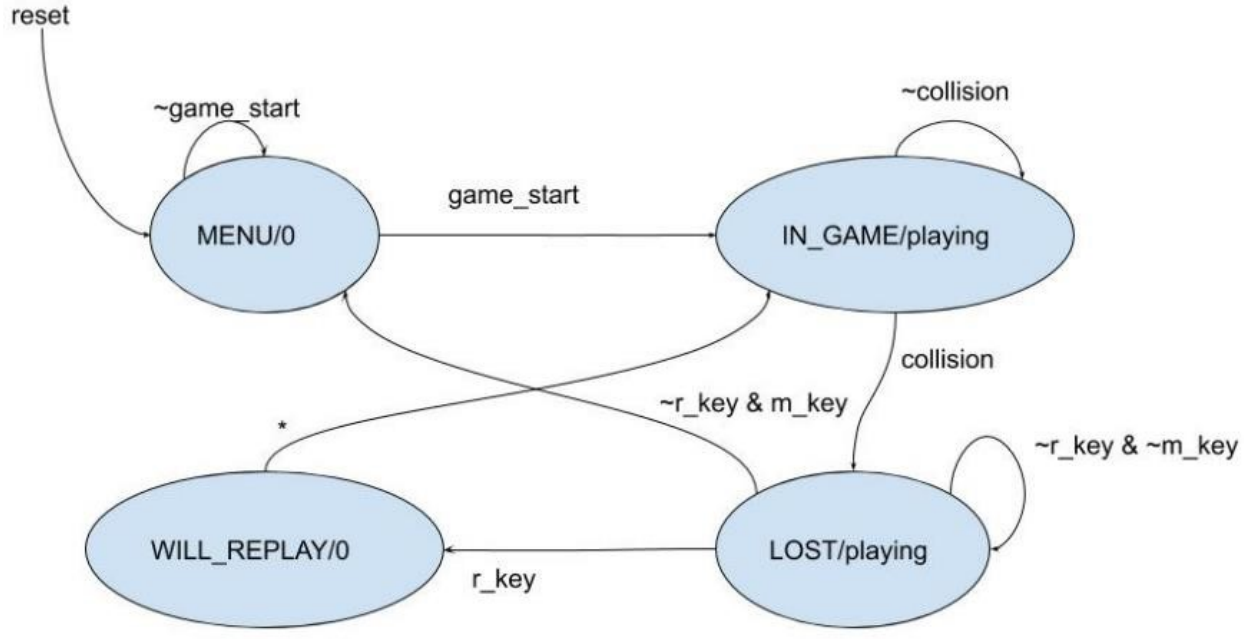
This module is responsible for generating pseudo-random numbers using a linear-feedback shift-register. This is used to decide the height at which a monster implemented with basic_monster appears on the screen, and also which monster appears next. Notice that this module can have its reset connected to LOW, thus the initial value inside the module is unknown, which increases the perceived randomness by producing an incredibly high possibility of games (Each of the 12 monsters using basic_monster has a 10-bit LSFR, and monster_picker has another 10-bit LSFR, all of which start without being synchronized). However, the LSFR can get stuck if all values were initially 1, since the module lacks a reset. To fix this issue, a multiplexer is used to insert a 0 when an LFSR is stuck on that state.



1.2.15 main_control

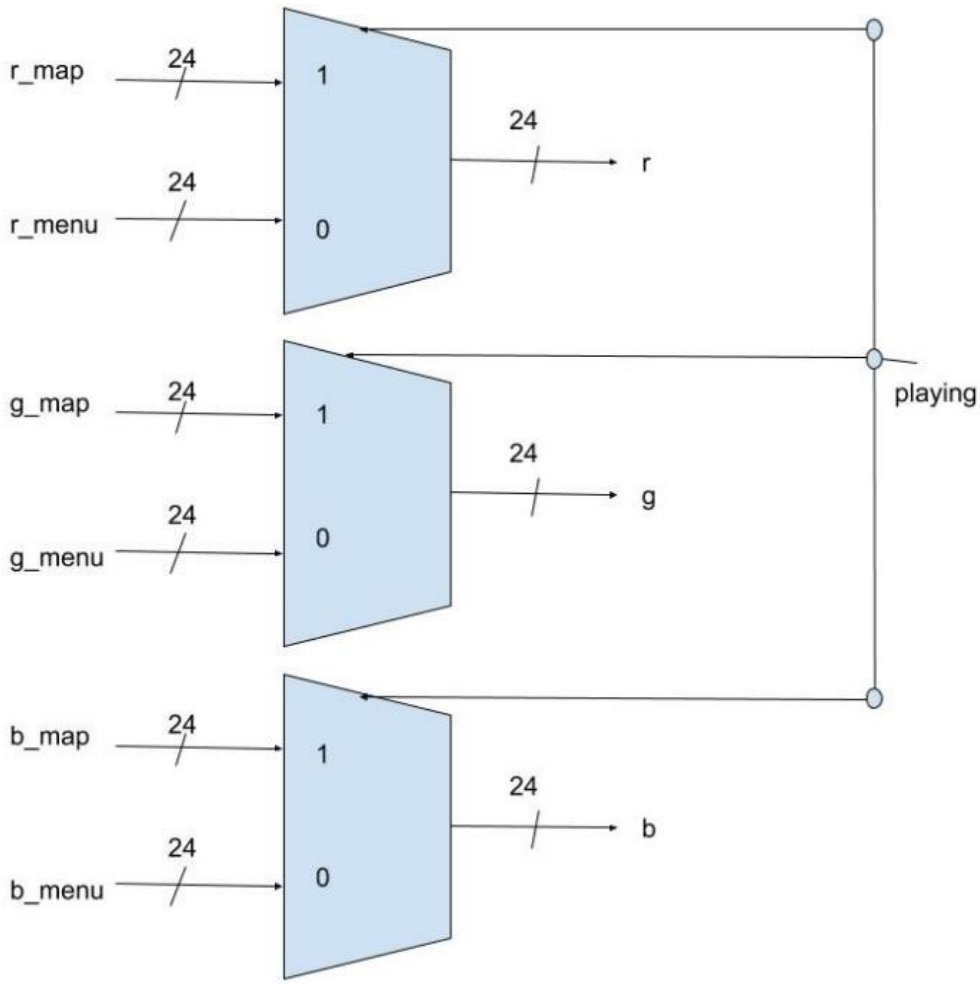
This module is responsible for telling the system what screen the user is on. User can press 'r' to restart the game, 'm' to go back to menu and stop the game when user is lost.

At reset, the output from menu is always shown, until the game is started by a positive pulse on game_start, produced by menu, which makes the module assert playing. When the game is being played, the module waits until a collision occurs, detected by the collision signal. The module then waits for user input. If the user presses the 'r' key, the game is re-played by producing a negative pulse on playing. If the user instead presses the 'm' key, the menu is shown, produced by de-asserting *playing*. The module display_mux uses *playing* to route the output from *menu* or *map_drawer* to the video drivers.



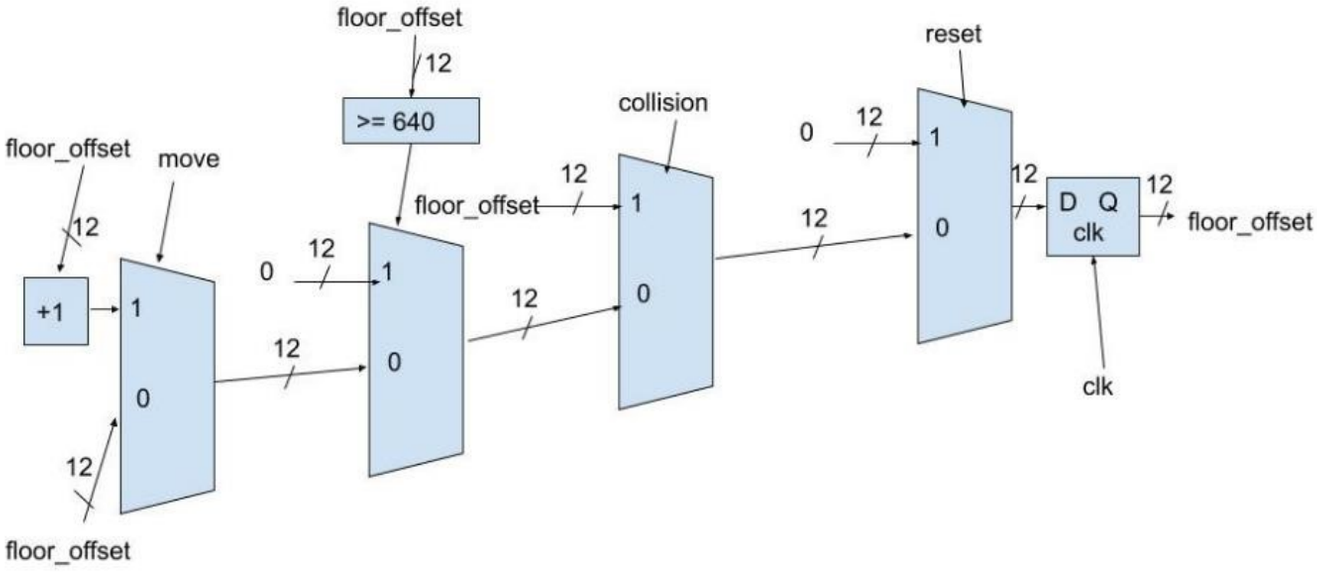
1.2.16 display_mux

This module is responsible for choosing the correct rgb output from screen inputs, based on the current screen the player is on.



1.2.17 floor_animator

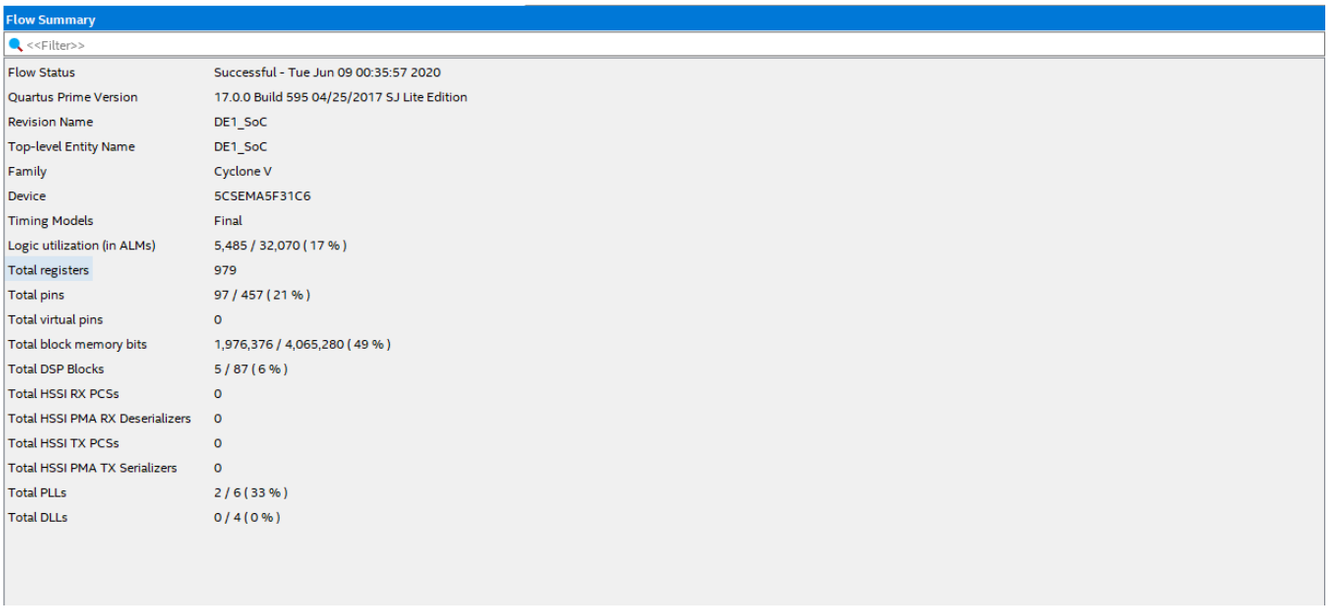
This module is responsible for animating the floor so that the floor seems like it's moving. This is produced by a counter to make an offset based on the *move* signal. Then *map_drawer* takes care of using this *floor_offset* as a horizontal shift applied to the *floor_sprite*.


floor_offset
12
>= 640
floor_offset
move
12
+1
1
0
12
floor_offset
0
12
1
0
12
floor_offset
12
collision
1
0
12
0
12
reset
1
0
12
D Q
clk
12
floor_offset
clk

2 Results

Overall, the lab is very rewarding because it provides an opportunity to combine all the skills that we have learned in the course and assemble them into a one, big project. Making a game on the DE1-SoC board requires a strong designing skill that assembles every logical elements in the game together, while making the game overall structurally clear. We have to come up with efficient strategy that manages and displays all objects on the screen, and handles all the edge cases for interactions between objects (such as collision, player's action, difficulty, and monster spawning).

The implementation of our system was found to meet and exceed our expectations, and each of the module was tested to meet the design requirements. As a result, we were able to create *dyno's adventure* game, a side-scrolling game where the main character (dyno) is a brave knight avoiding obstacles in a dangerous world. The game features a menu where the user can navigate with the keyboard to enable/disable animations and/or sound. The user can duck and avoid obstacles using the down arrow key and the spacebar. The user can also re-play or go back to the menu after they have lost by pressing 'r' or 'm' (respectively). The game features sound effects for dying, walking, and leveling-up. At each level, a new monster is discovered, and the game gets a bit faster. The monsters come in a random order, and at random heights if they are flying monsters. In total, there are 12 different monsters, half of which are ground characters and half of which are flying characters. The score grows with time and at the pace of the game. The score is shown on the top-right. The game features animations using different sprites for dyno (running/jumping while standing/ducking), the dragon (flapping its wings), and the flying witch (and her magic broom). Finally, the game features cheat-codes. To activate a cheat-code, the user must enter a keyword during the game. For the keyword to count, all letters should be pressed at the same time. The cheat-code "peace" turns off collisions, and the cheat-code "chill" slows down the game to its lowest level. First, the flow-summary for the top-level module:



Flow Summary

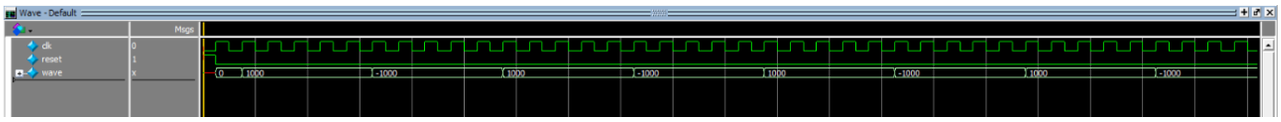
<<Filter>>

Flow Status	Successful - Tue Jun 09 00:35:57 2020
Quartus Prime Version	17.0.0 Build 595 04/25/2017 SJ Lite Edition
Revision Name	DE1_SoC
Top-level Entity Name	DE1_SoC
Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	Final
Logic utilization (in ALMs)	5,485 / 32,070 (17 %)
Total registers	979
Total pins	97 / 457 (21 %)
Total virtual pins	0
Total block memory bits	1,976,376 / 4,065,280 (49 %)
Total DSP Blocks	5 / 87 (6 %)
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	2 / 6 (33 %)
Total DLLs	0 / 4 (0 %)

Below are the simulations for this lab:

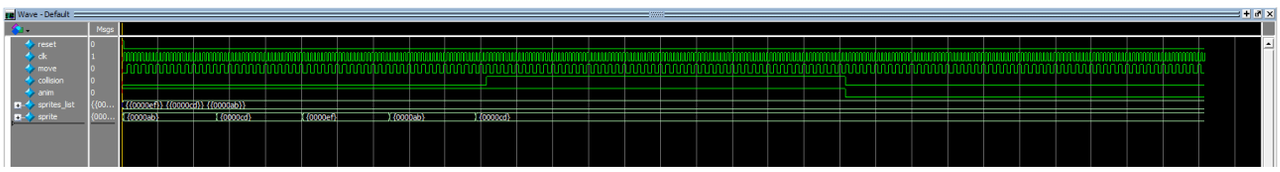
2.1 square_wave_sound_gen

- A square wave with 10 cycles of period is shown and an amplitude of 1000.



2.2 sprite_rotator

1. The sprites are 0xAB, 0xCD, 0xEF.
2. Collisions haven't occurred, animations are ON, and the signal *move* alternates. The sprites rotate (0xAB, 0xCD, 0xEF, 0xAB, etc). When the *move* signal is asserted.
3. Then there is a collision (*collision* is HIGH) and the sprites stop rotating.
4. The *collision* is de-asserted, but *anim* is LOW and thus the sprites don't rotate.

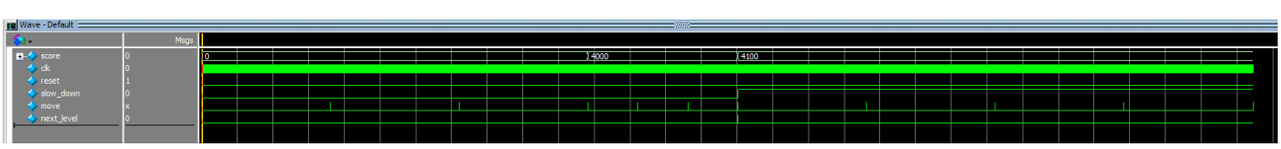


2.3 speed_selector

- The console verifies that the correct time (based on the clock period) occurs at different levels, and then it also checks that the *slow_down* cheat-code translates into the speed of the first level.
- Looking at the waveform, one can also see visually that it starts slow (less frequency for move) when the score is zero. When the score increases, the speed of the game increases (closer pulses for move). Finally, when *slow_down* is asserted, the speed becomes again the speed of the first level.

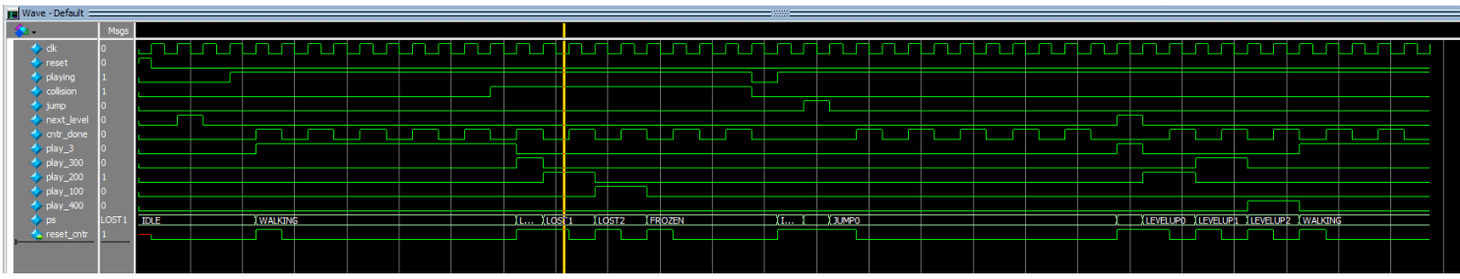
```
# Success: DeltaT =      180289
# Success: DeltaT =      180289
# Success: DeltaT =      180289
# Success: DeltaT =       70101
# Success: DeltaT =       70101
# Success: DeltaT =       70101
# Success: DeltaT =      180289
# Success: DeltaT =      180289
# Success: DeltaT =      180289
# ** Note: $stop    : ../../speed_selector.sv(138)
```

Console for speed_selector



2.4 sound_controller

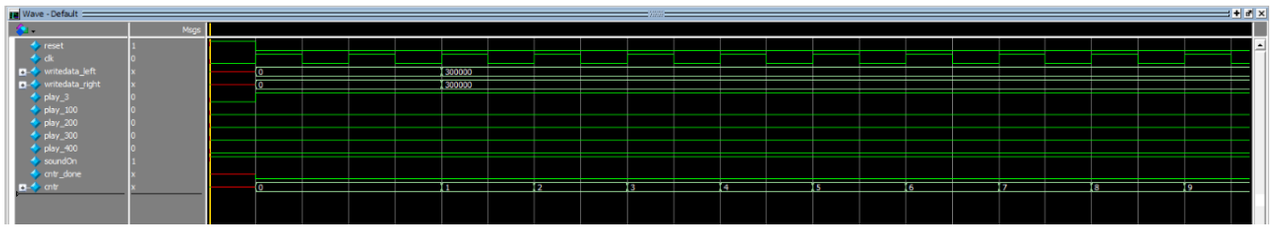
1. The game is not being played, thus no sound is played on IDLE.
2. Then the game starts and *play_3* produces footstep sounds.
3. A collision occurs, and the collision sound happens and the LOST status leads to FROZEN.
4. The game is re-started by de-asserting and re-asserting the *playing* signal.
5. A jump occurs which produces no sound.
6. The next-level is obtained producing the *next_level* sound is played during the *LEVELUP* status.
7. More footsteps at the end.
8. Notice that any time a transition occurs, *reset_cntr* is pulsed HIGH. This allows sound-effects to last full time, no matter when in the game these occur.

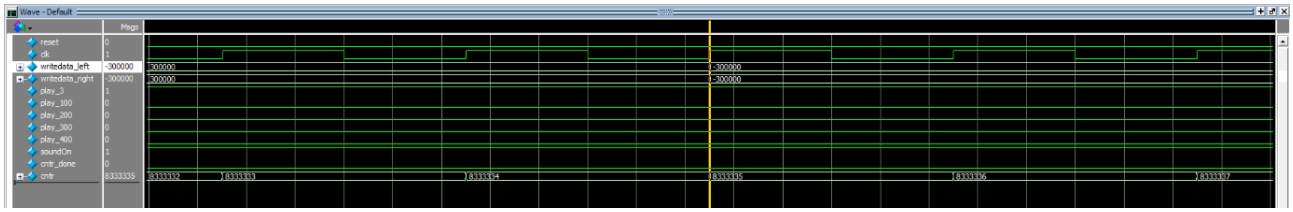


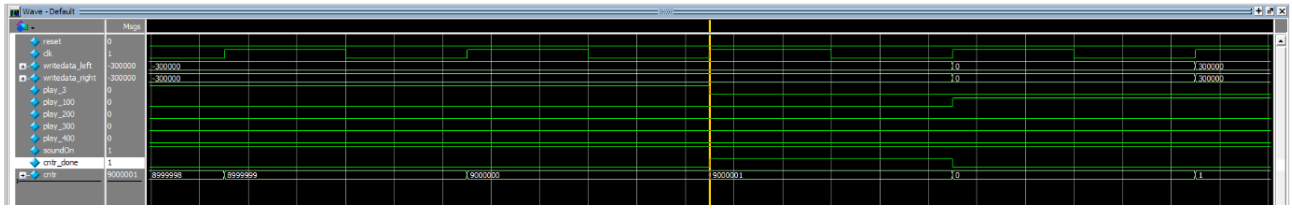
2.5 sound_datapath

Inside Sound_Simul.sv, I have provided three simulation profiles for testing sound, which regulate the length of a sound effect. When uploading to the board, the profile sets 50 million cycles per sound-effect stage which equals 1 second, and most effects have three stages. For this testbench, I am using SOUND_SIMUL_0 which means each stage takes 9 million cycles which allows my computer to run the simulation without crashing, but shows all the information we need:

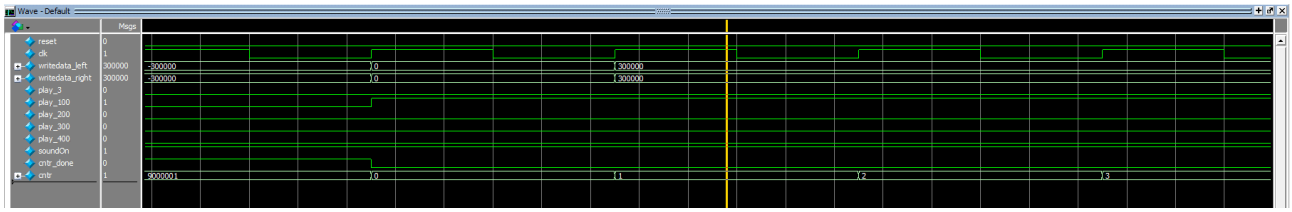
1. First three images show:
   - *play_3* is asserted at $cntr = 0$. The sound is output with a positive amplitude of $300,000$ at $cntr = 1$, and changes to negative at $cntr = 8333350$. The period is $2 \cdot cntr$ (the behavior of *wave_sound_gen* is verified to produce square waves) and thus we have $\left((2 \cdot (8,333,350)) \cdot \frac{1s}{50Mcycles}\right) = 3Hz$
   - This is repeated until cntr=PREDIV_VAL.

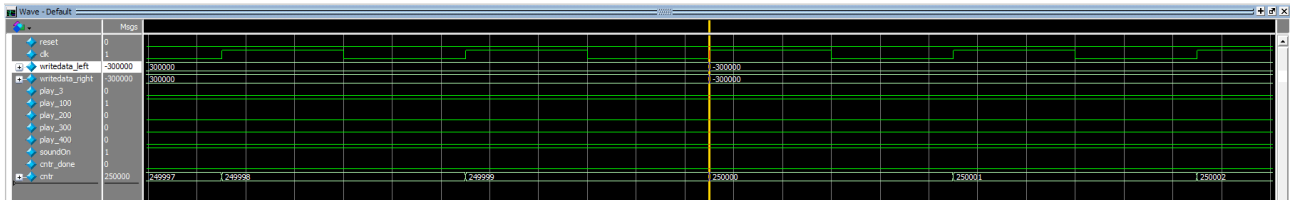


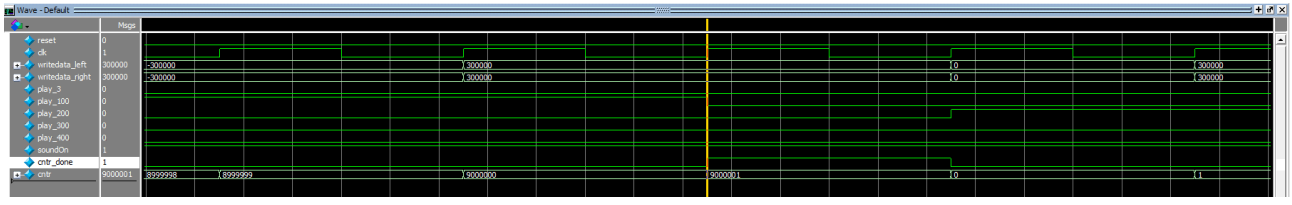




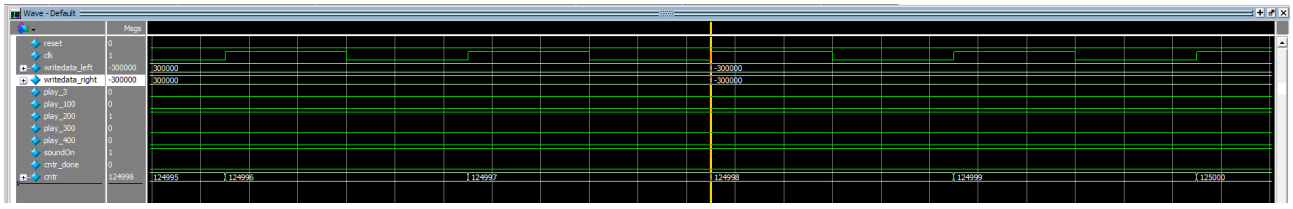
2. The same process is followed for play_100. The frequency is (simplifying the formula): $\frac{25M}{250{,}000} = 100Hz$.

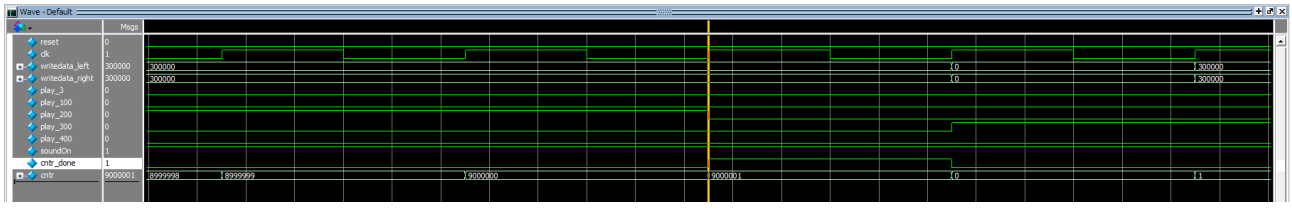




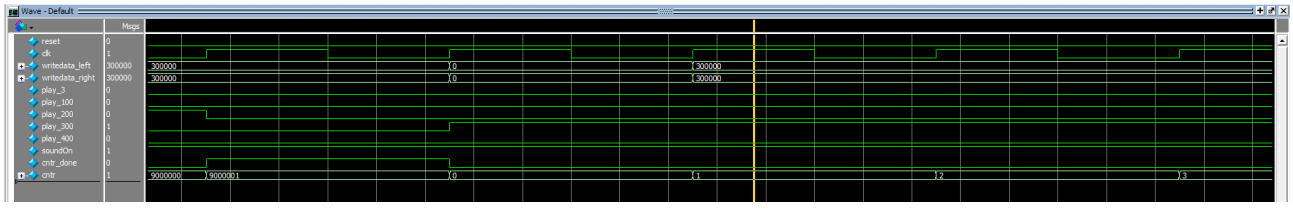


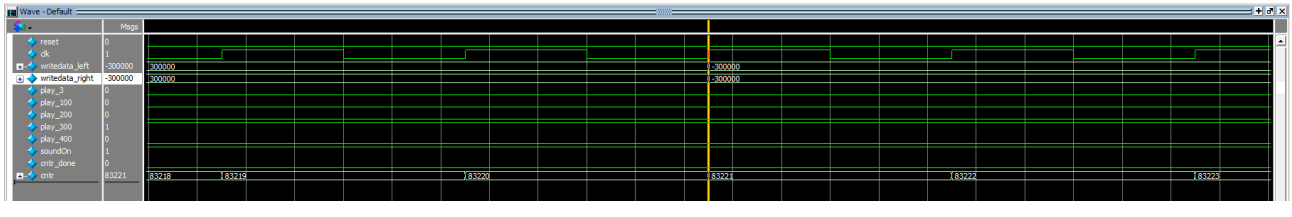
3. For *play_200:* $\frac{25M}{124998} = 200Hz$.

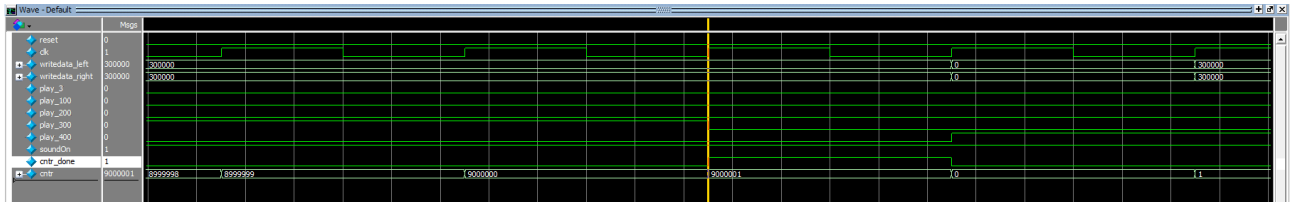




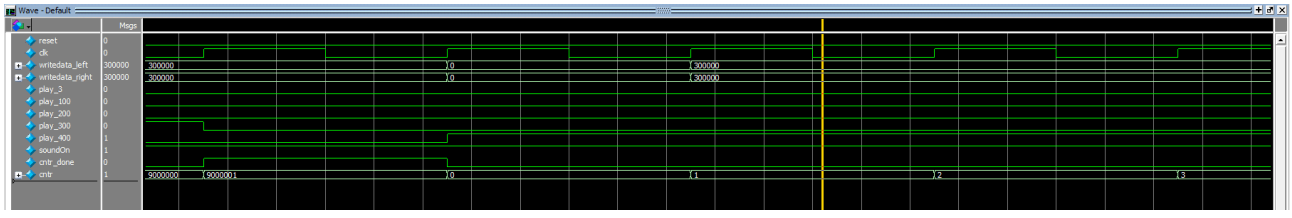
4. For *play_300:* $\frac{25M}{83221} = 300Hz$.

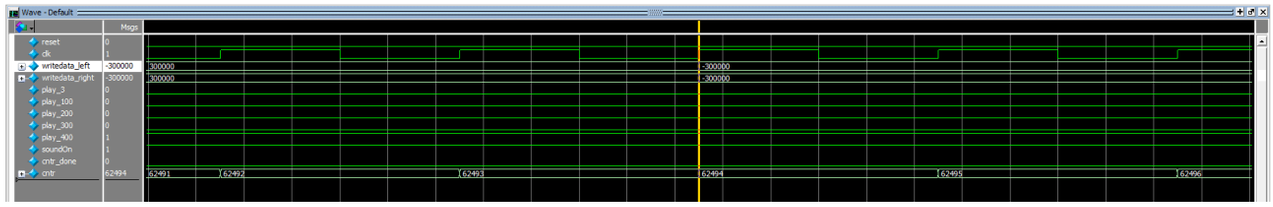


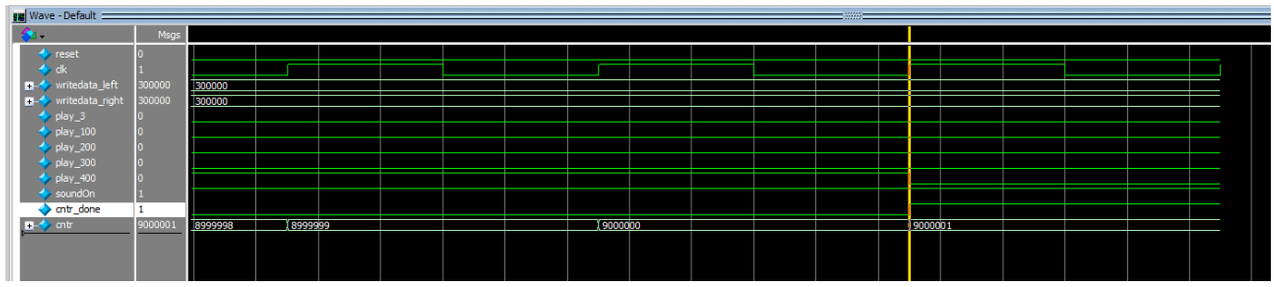




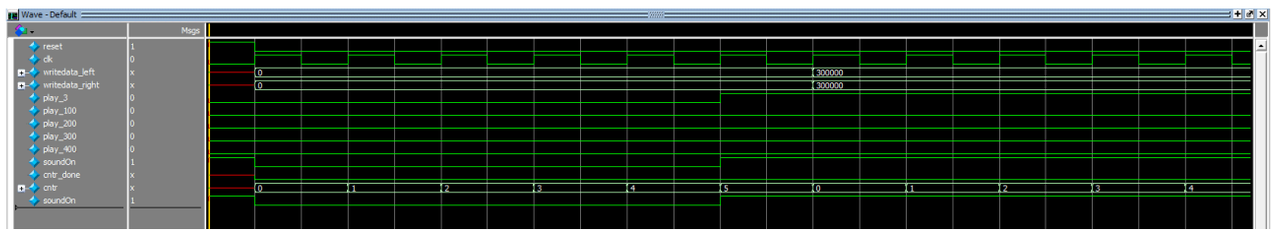
5. For *play_400:* $\frac{25M}{62494} = 300Hz$.







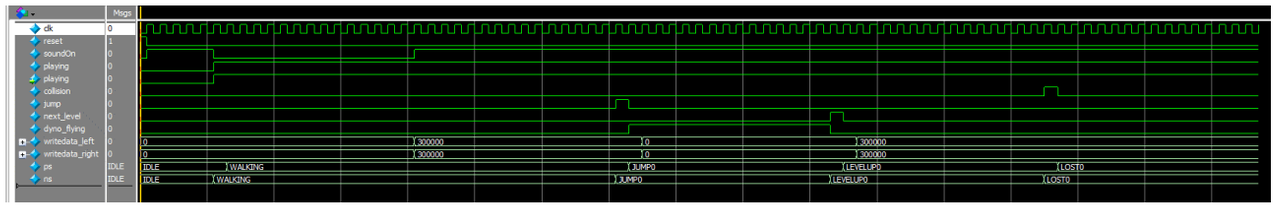
6. Finally, the mute feature works: Sound doesn't play until soundOn is HIGH.



2.6 sound

First, SOUND_SIMUL_1 is used to verify the logic. This means each sound stage takes only 10 cycles.

1. At first, the game hasn't started *playing* is LOW, thus no sound is output.
2. Then the game starts playing is asserted, but *soundOn* is FALSE. Thus no sound is output.
3. Next, *soundOn* is asserted and the WALKING footsteps can be heard.
4. *Jump* is pulsed HIGH and *dyno_flying* is asserted and JUMP override the footsteps and no sound is output
5. Then *next_level* is pulsed HIGH and the LEVELUP sound starts.
6. Finally, a collision occurs (*collision* is asserted) and the LOST sound effect begins.



7. Now the same logic is repeated, but using SOUND_SIMUL_2 which uses 9 million cycles per stage, to show that all stages of a sound effect are being played and to verify their relative frequencies.

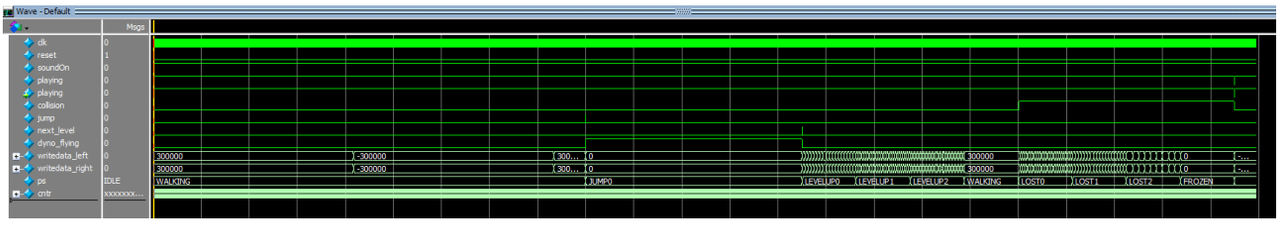
8. First, the user is walking and we can see a very low frequency for the footsteps.

9. Then, *jump* is pulsed and the user jumped, we can see the JUMPX sequence where no sound is played (since there should be no footsteps in the air).

10. Now, *next_level* is pulsed HIGH and the LEVELUPX sequence is played, where each stage has higher frequency.

11. Next, *collision* gets asserted causing the LOSTX sequence where each stage has lower frequency, the module becomes FROZEN.

12. Finally, *playing* is pulsed LOW and *collision* is de-asserted and the module goes back to WALKING (see cursor information).



2.7 score_manager

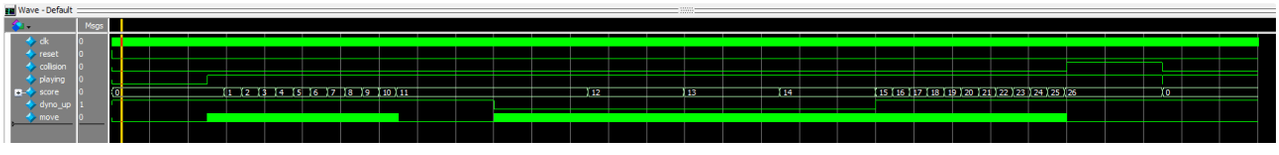
1. First, *playing* is low, and no points are awarded if the game is not being played.

2. *playing* is asserted, the user starts getting points. Notice *dyno_up* is also HIGH

3. *move* is de-asserted, thus no points are awarded (this allows the game to speed up the score proportional to the speed of the game). Then *move* is asserted again.

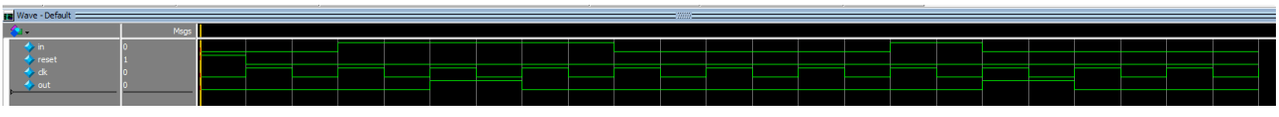
4. When *dyno_up* is de-asserted, the user gets points more slowly.

5. Finally, *collision* being asserted caused the score to freeze. A negative pulse on the signal *playing* causes the game to be re-started and thus the score becomes 0 again.



2.8 rising_edge_checker

1. Every sequence of TRUEs for the signal in are translated into a single pulse for the signal out.



rising_edge_checker testbench

2.9 basic_monster_controller

1. The monster IDLEs for a bit which causes it to get removed (*remove* goes HIGH) out of screen

2. The *playing* signal is asserted and the game starts. The monsters goes WAITING.

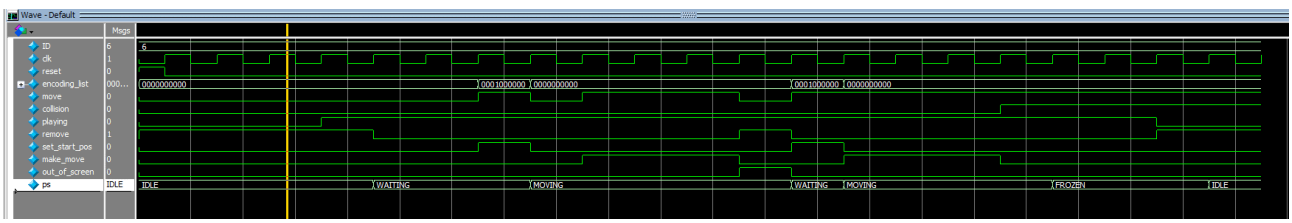
3. Once the ID of this monster is pulsed in *encoding_list*, the *set_start_pos* is pulsed and the monster starts MOVING with the *make_move* signal.

4. The monster goes out of screen, *remove* is pulsed and the monster goes WAITING.

5. Signal *playing* is pulsed and the monster goes back the screen when *set_start_pos* is pulsed and then the monster goes to state MOVING.

6. When *collision* gets asserted at the end, the monster goes FROZEN and stops making moves.

7. Finally, once the *playing* signal is de-asserted, the monster goes IDLE, waiting for the next game to start.



2.10 basic_monster_datapath

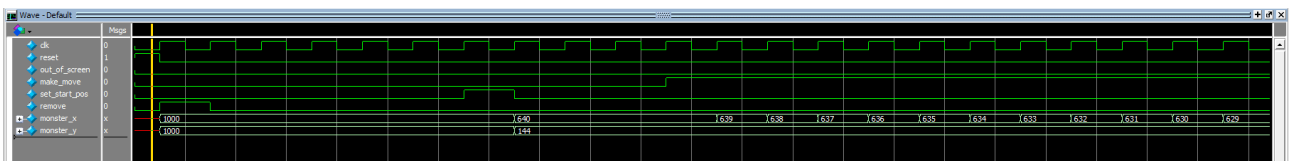
1. The first image shows a pulse on *set_start_pos* that puts the monster on screen. Then *make_move* is asserted and the monster moves to the left of the screen (*monster_x* changes).

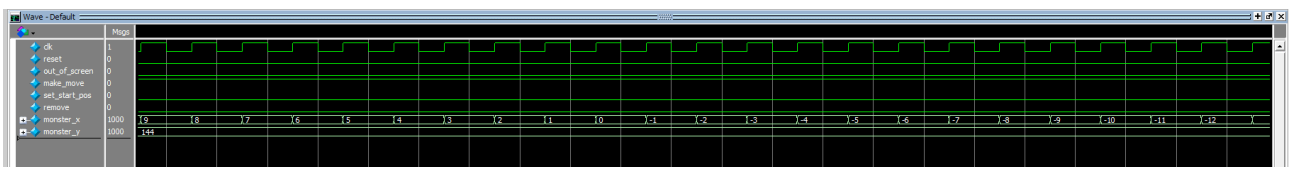
2. The monster goes out of screen on the second image.

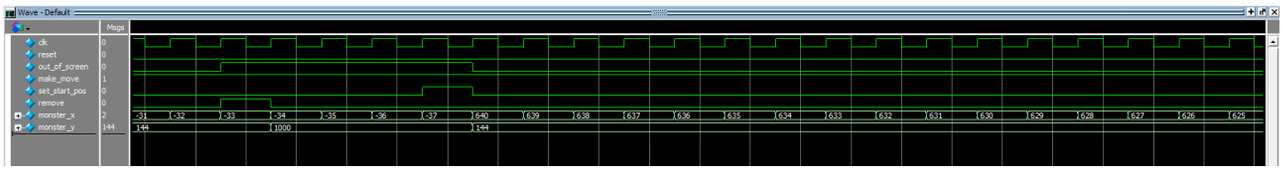
3. The third image shows the monster is fully out of screen and the *remove* signal removes the monster and it gets ready to re-appear.

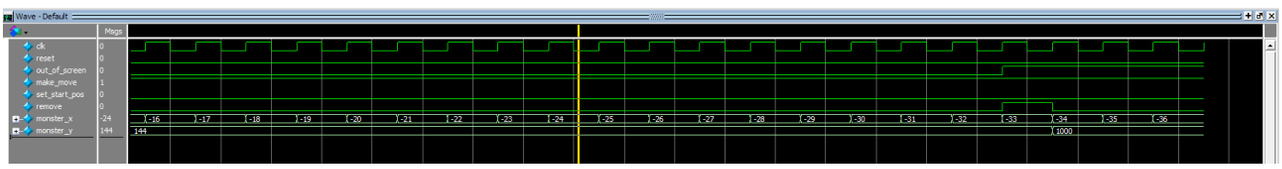
4. *set_start_pos* is asserted and the monster comes back to screen.

5. In the last image, the monster again went out of screen, *remove* is pulsed, and the monster gets ready to re-appear.









2.11 basic_monster

1. The monster first IDLEs.

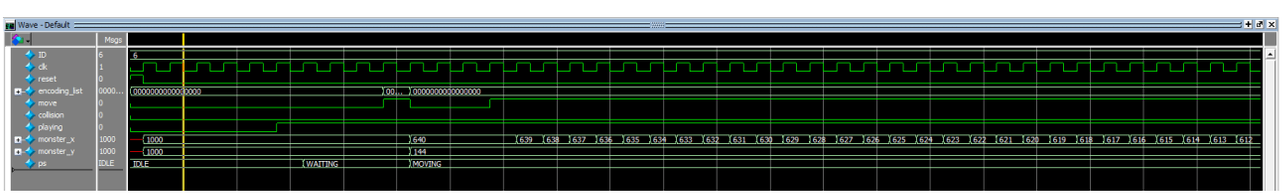
2. *playing* is asserted and the monster gets initialized since the *encoding_list* outputs a high pulse on index 6 (the ID of this monster), and starts moving from right to left on the screen.

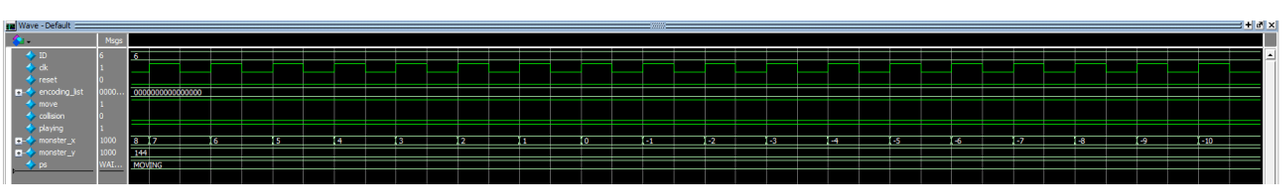
3. In the second image, after the monster has moved all the way to the left, it starts going out of screen.

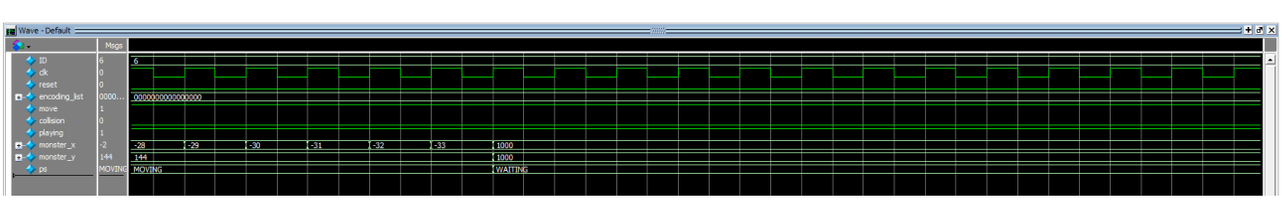
4. In the third image, the monster has gone full way out of the screen, and gets ready to come out again when the encoding list has the appropriate ID.

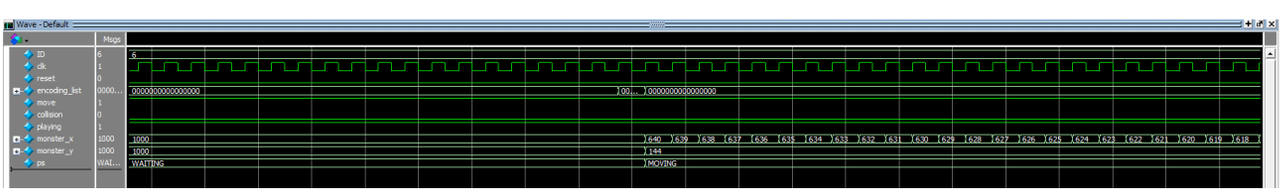
5. Fourth and fifth images release the monster again, and then show a *collision* (the monster goes to state FROZEN).

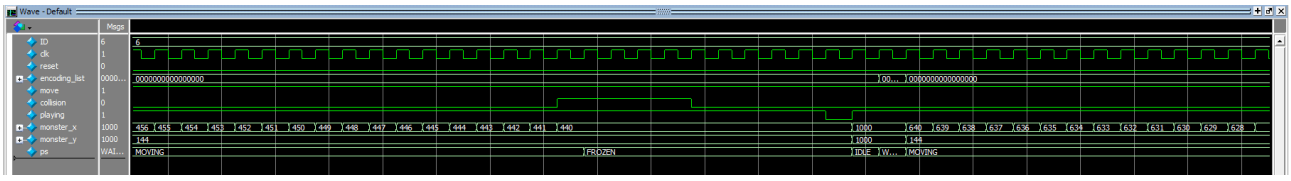
6. In the last image, a negative pulse on playing re-starts the monster (ex: the user went pressed 'r' after losing, or went to the menu and came back to play).

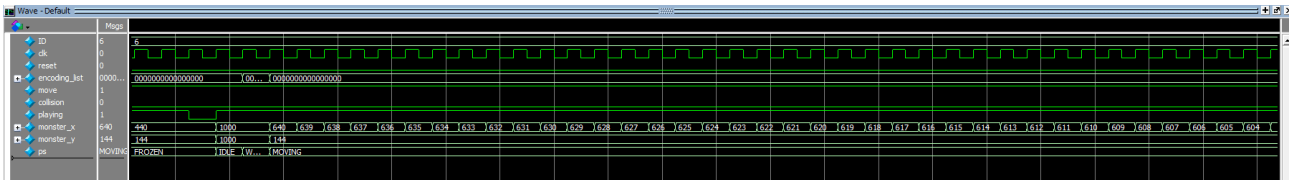






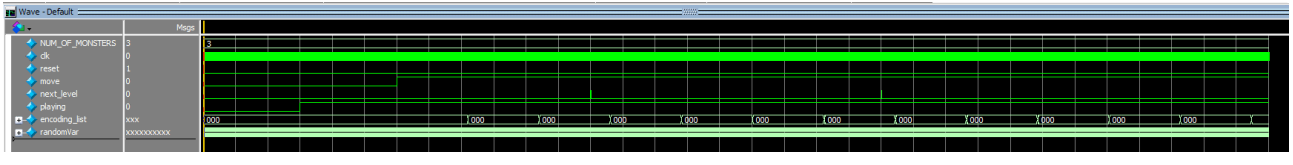




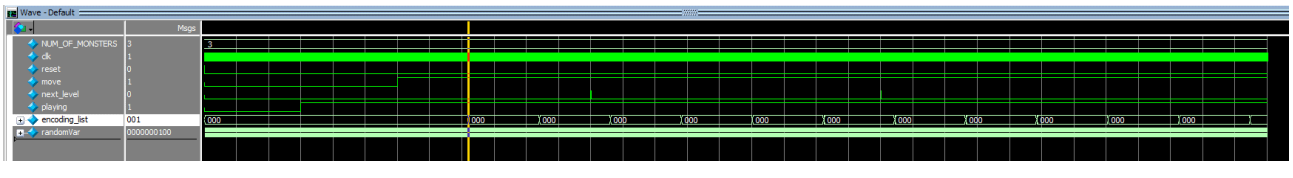


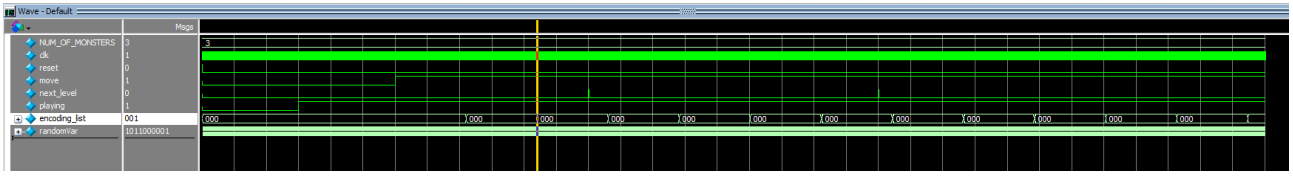
2.12 monster_picker

1. At the beginning playing is off, and no monsters should go to the battlefield.
2. Then *playing* is asserted, but *move* is not. This causes no monsters to go to the battlefield.
3. Monsters start coming out when *move* is asserted.

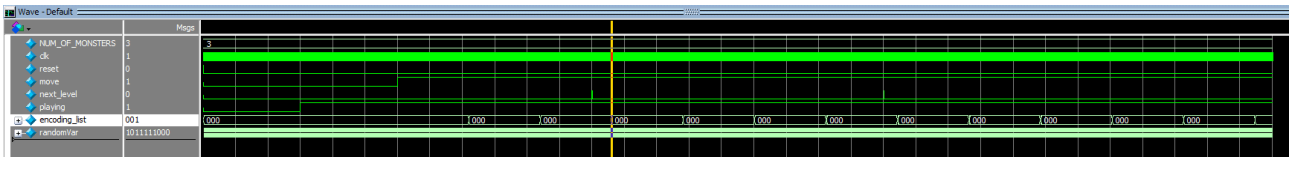


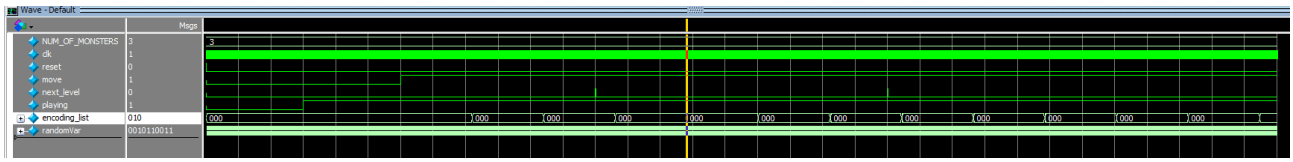
4. The first monster has ID 0 (the lowest bit of *encoding_list* is pulsed HIGH for a cycle). At level 0, this is the only possible monster. To observe the ID, see the value of *encoding_list* on the left side (cursor information).

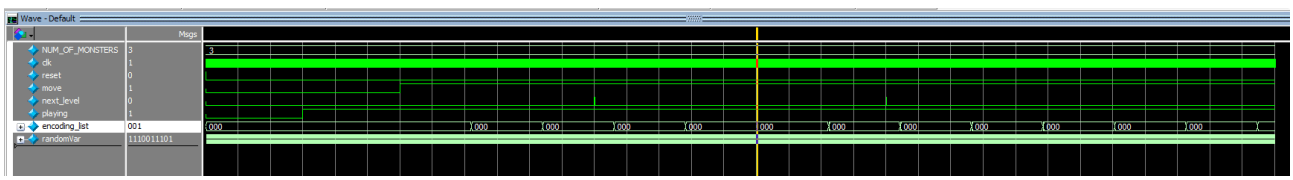


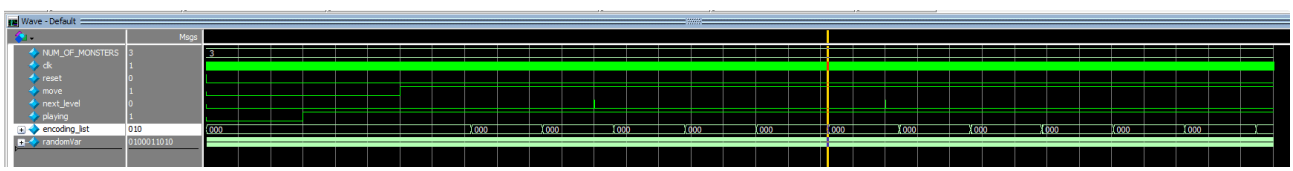


5. Then *next_level* is pulsed HIGH. Now, on level 1, we can have ID 0 or ID 1.

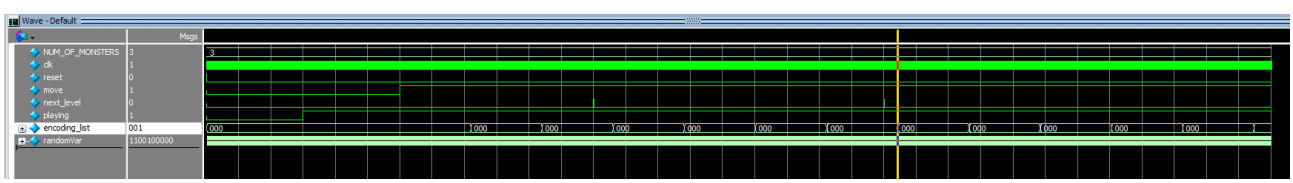


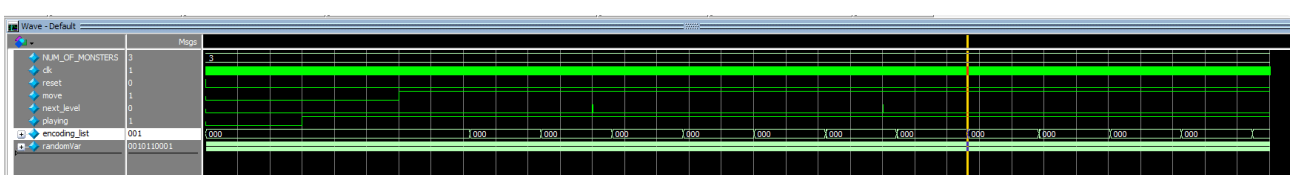


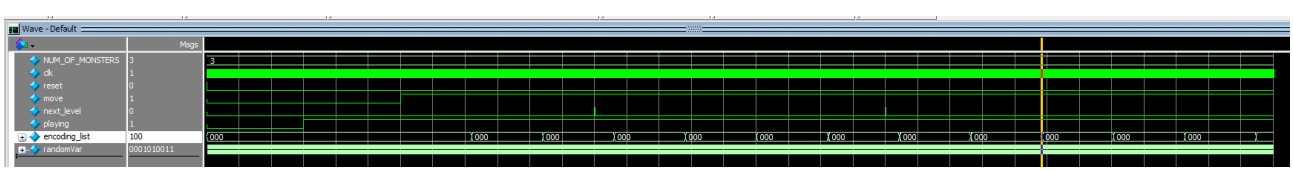


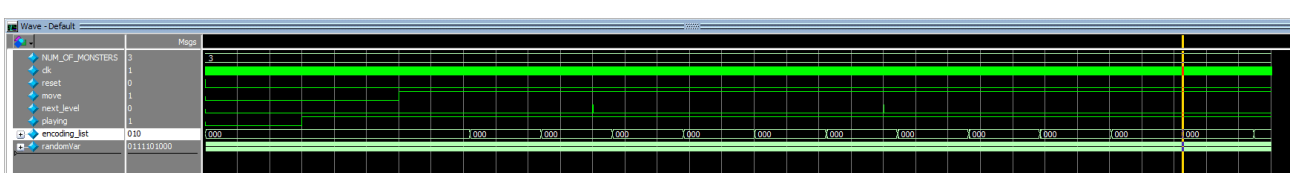


6. Last, *next_level* is pulsed HIGH and now we have monsters with ID 0, 1, and 2.









2.13 dyno_controller

1. *playing* is asserted which causes dyno to get initialized (*init* is pulsed HIGH) and dyno is put to GROUND (*put_dyno_gnd*).

2. *set_dyno_up* indicates dyno is standing up.

3. Then *down_pressed* causes a negative pulse on *set_dyno_up* and a high pulse on *clr_dyno_up*, indicating the user ducked momentarily.

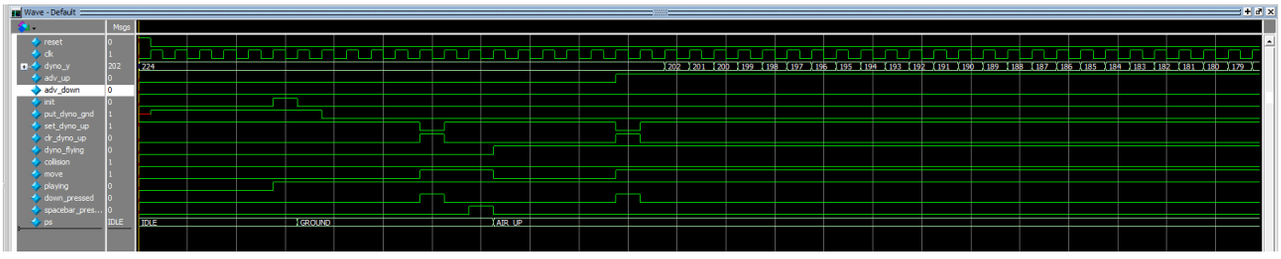
4. Then space_bar is pressed and *dyno_flying* is asserted. Dyno goes to AIR_UP state.

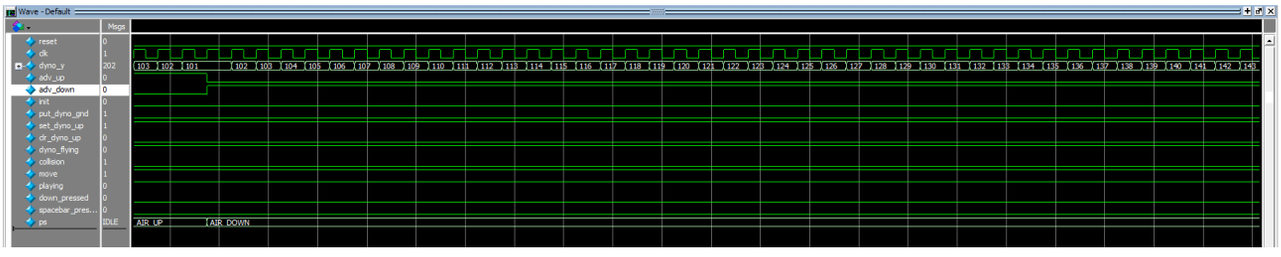
5. It is possible to duck in the air, so *down_pressed* at around causes dyno to duck while on air. Also, the position *dyno_y* starts changing in the direction such that the character will move vertically and up.

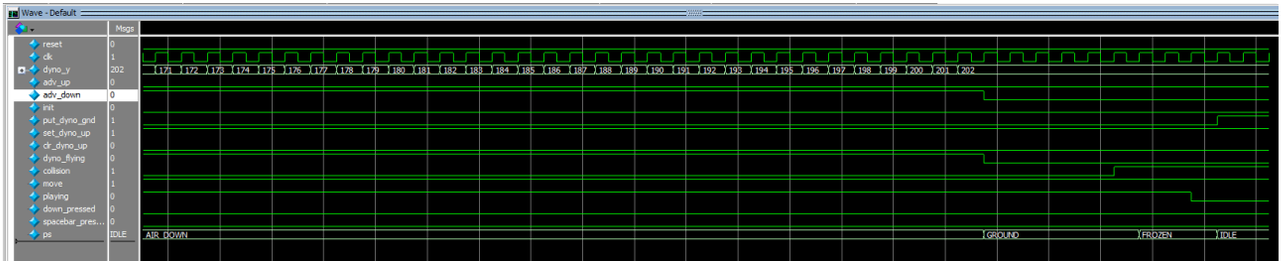
6. In the second image, dyno is falling (AIR_DOWN).

7. Finally, dyno goes back to GROUND on the last image.

8. A collision is simulated and dyno goes to the FROZEN state.







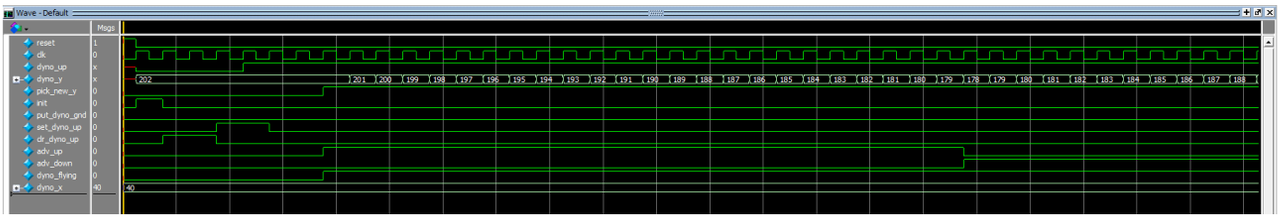
2.14 dyno_datapath

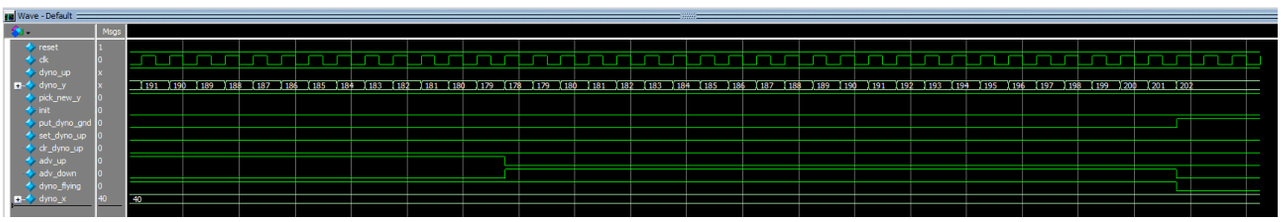
1. First, dyno is initialized in the field due to *init*, with coordinates x=40, y=202. Then dyno stands up due to the initial *set_dyno_up* which sets *dyno_up*.

2. The signals *adv_up* and *dyno_flying* cause dyno to fly and its y position *dyno_y* changes accordingly.

3. Dyno then starts going back to ground after *adv_up* is de-asserted.

4. Finally, *put_dyno_gnd* is asserted and dyno falls back to ground and *dyno_flying* is de-asserted.





2.15 dyno

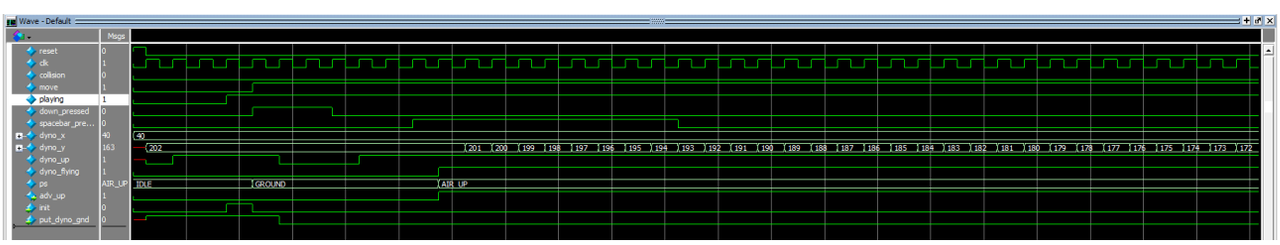
1. First, the *playing* signal is asserted and dyno is initialized (*init*) and put to ground (*put_to_gnd*).

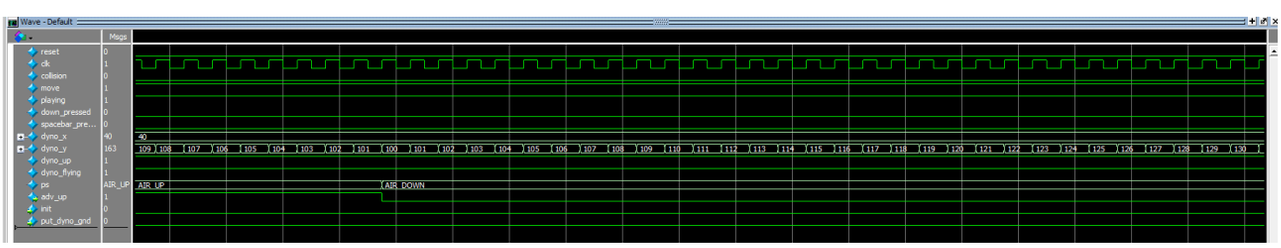
2. The user presses down key and dyno ducks down (*down_pressed* and causes *dyno_up* to be de-asserted). Then *spacebar_pressed* is asserted and dyno jumps.

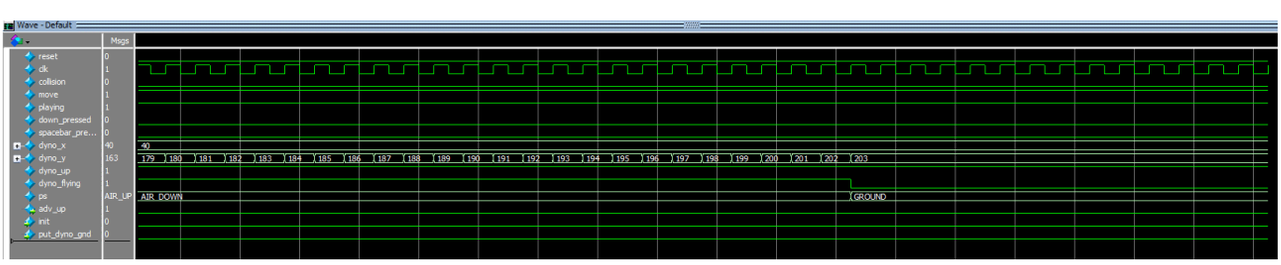
3. Dyno goes to the state AIR_UP and its position *dyno_y* starts changing.

4. In the second image, dyno goes from AIR_UP to AIR_DOWN and starts falling down (see*dyno_y*).

5. In the third image, dyno has fell back to the ground (GROUND state).

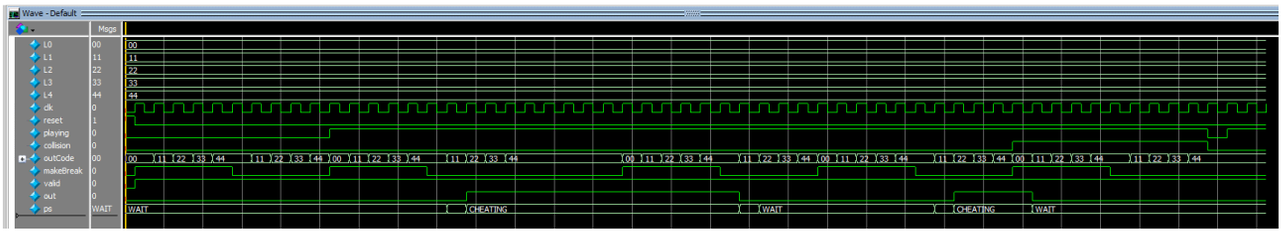






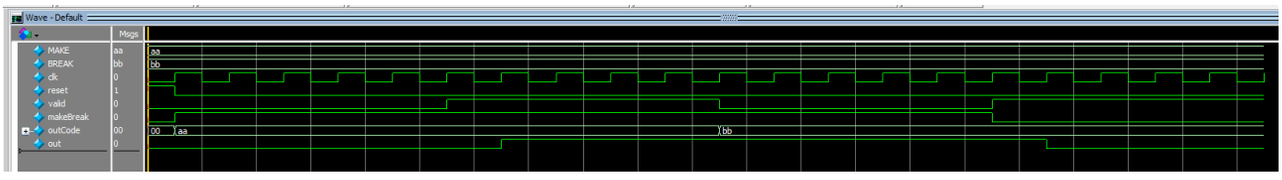
2.16 cheat_check

1. The cheat-code in this testbench is 0x00, 0x11, 0x22, 0x33, 0x44.
2. It is entered at first, but *playing* is LOW and the cheat-code is ignored.
3. It is entered again when *playing* is HIGH, and the cheat-code succeeds (*out* is asserted).
4. It is entered again, which causes the cheat-code to be de-activated (goes into WAIT).
5. It is entered again, which succeeds. But then *collision* is asserted, and the cheat-code is disabled (goes into WAIT).
6. It is entered one last time after the collision, but it is ignored due to the collision.



2.17 keyboard_input_filter

1. The key that corresponds to 0xaa is pressed, but *valid* is FALSE and the input is ignored.
2. The signal *valid* becomes TRUE, and *makeBreak* is TRUE (make). Then the input is not ignored and *out* becomes HIGH.
3. The signal *valid* becomes FALSE, and the outCode is ignored.
4. The signal *valid* becomes TRUE, and *makeBreak* is FALSE (a break). Thus *out* gets de-asserted since the key is un-pressed (around 1650 ps).



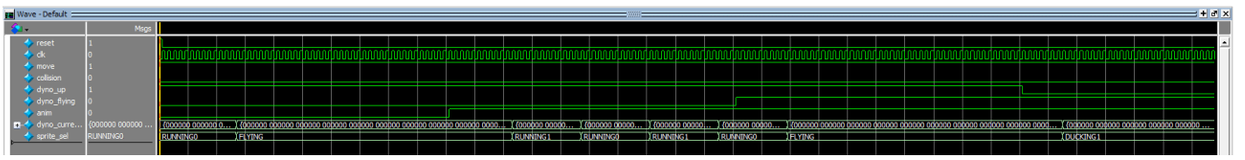
2.18 dyno_sprite_selector

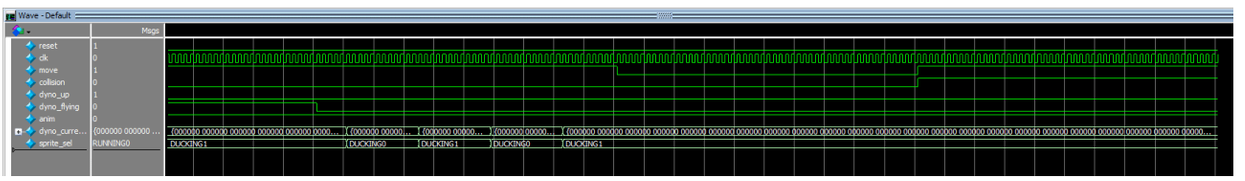
1. *dyno_up* is TRUE and *dyno_flying* is FALSE. Thus the sprite RUNNING0 is correct in the first cycle after reset.
2. The user chose *anim* to be OFF while on the menu, so when the game starts the sprite is FLYING (this sprite is re-used for jumping and for disabled animations).
3. Then *anim* is asserted and *sprite_sel* starts alternating between RUNNING1 and RUNNING0, effectively simulating footsteps.
4. Now *dyno_flying* is asserted around 1050ps, which causes *dyno* to fly around 1050ps. Thus *sprite_sel* is FLYING.
5. Mid-flight, dyno decides to duck, and the sprite is correctly DUCKING1.

6. In the second image, *dyno_flying* is de-asserted and dyno is now on the ground. Thus *sprite_sel* alternates between DUCKING0, and DUCKING1 to simulate footsteps while ducking.

7. The signal move is de-asserted and dyno should stop moving, which is why *sprite_sel* stops changing.

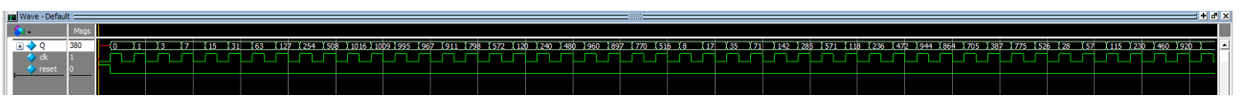
8. Finally, move is asserted but *collision* is asserted too, the player has lost the game. The game is frozen and the last sprite is preserved.





2.19 lfsr10

1. Shows some values of the linear feedback shift register.



2.20 main_control

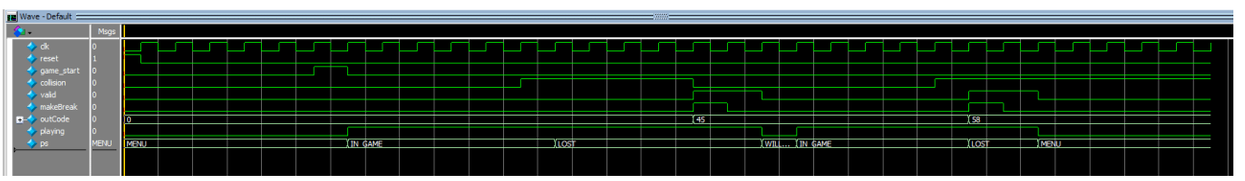
1. The signal *game_start* is pulsed HIGH and the game starts (state becomes IN_GAME and *playing* is asserted).

2. The user lost since *collision* is asserted at around 1150 ps. State becomes LOST.

3. The user presses key 'r' and the game starts again, so playing is pulsed LOW for a cycle and then asserted HIGH. The state becomes IN_GAME after WILL_REPLAY.

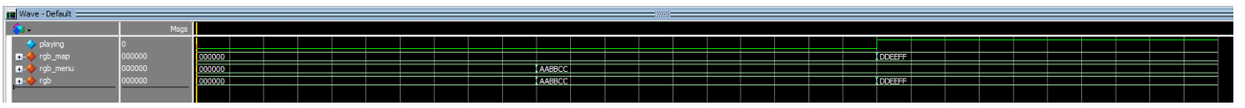
4. Again, the user lost due to a collision.

5. The user pressed 'm' and the game goes to state MENU. The signal *playing* is de-asserted.



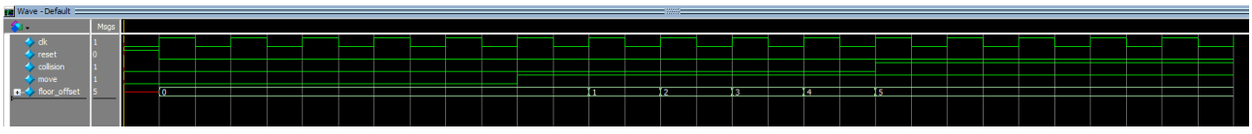
2.21 display_mux

1. The rgb triplet of colors follows *rgb_menu* when playing is LOW. Otherwise, it follows *rgb_map*, as shown below:



2.22 floor_animator

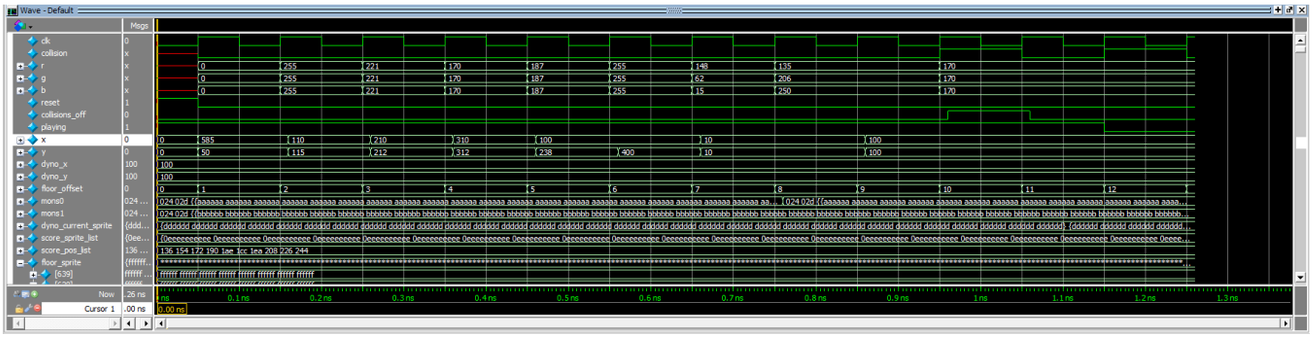
1. First, *move* is not asserted, thus *floor_offset* doesn't change.
2. Then, *move* is asserted, and *floor_offset* starts increasing.
3. Finally, a collision occurs (*collision* is asserted), and the game is frozen. Therefore *floor_offset* stops changing.



2.23 map_drawer

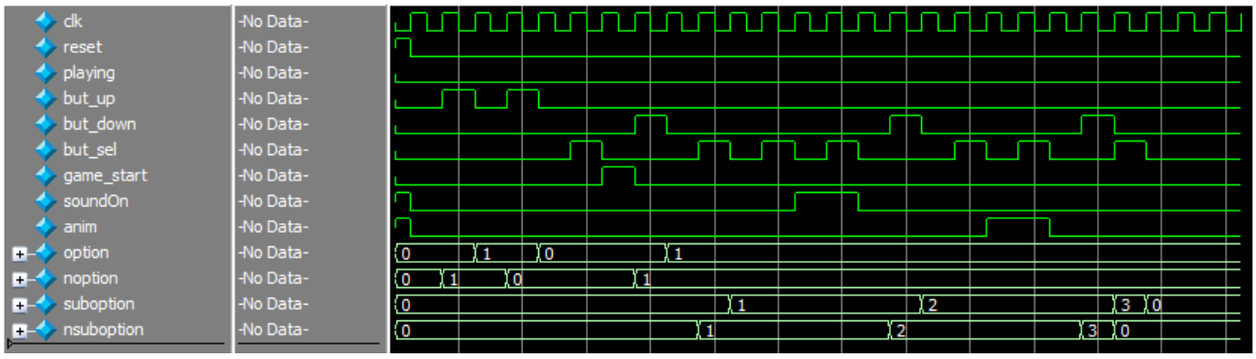
1. There are a lot of signals in the window. However, the only output signals in this module are *collision* and *r, g, b*. Everything else is an input, and is therefore simulated.
2. First, (x, y) are situated within the score area, and it is checked that the module produces the color of the score. This is 255, 255, 255 (white)
3. Then (x, y) are situated within the position of the main character, and it is checked the module produces the r, g, b triplet that we put to simulate dyno. This is 0x8DD. This is 221 in decimal.
4. There are two monsters in the playfield. The simulated monsters are: Monster0 has r, g, b all 8'hAA, and Monster1 has 8'hBB. Thus we set (x, y) to overlap with the monsters, and we verify the color outputs. This is 170 and 187 in decimal.
5. We check the color of the floor, which in our simulation should be 8'hFF for each of r, g, and b.
6. We check the color of the ground below the floor. This is brown $(r, g, b) = (148, 62, 15)$.
7. Now it is time to test the logic. We overlap monster0 and dyno. The collision is not detected until it is time to draw an overlapping pixel. In the simulation, this occurs one cycle after we have $(x, y) = (100, 100)$. The signal *collision* gets asserted.
8. Then *collisions_off* is asserted and therefore *collision* is de-asserted. Even with the collision since these were disabled.
9. Similarly, *collisions_off* is de-asserted. But *playing* is de-asserted and the player must be on the menu. Thus *collision* remains de-asserted.

```
# Score color is correct at t=                160
# Dyno color is correct at t=                260
# Monster #0 color is correct at t=                360
# Monster #1 color is correct at t=                460
# Floor color is correct at t=                560
# Ground color is correct at t=                660
# Background color is correct at t=                760
# Will detect a collision:
# Collision occurred! at                960
# Success, collisions disabled
# Now on menu. The only thing to check here is collisions are off: Correct
```



2.24 menu

1. Button up is pressed to make sure it moves the menu. See that *option* increases.
2. Button down is pressed to make sure it also move the menu. See that *option* goes back to zero.
3. Button select is pressed on option 1 to make sure that *game_start* functions. Notice that *game_start* has a positive pulse.
4. Option is being moved to the setting option through *but_down*.
5. Button select is pressed to make sure it enters the sub option menu. See now that the *suboption* signal responds to the buttons while the regular option doesn't.
6. Now navigate through the options and toggle sound, and animation in order to make sure that pressing select button on the respected option changes them,
7. Finally, use *but_down* to move to exit menu and then return to the parent menu with the final *but_sel*.



3 Experience Report

3.1 Partner Work Summary

We have divided the work by modules. Since Andrew Shi joined in the middle of the project and the game already has its basic structure constructed, Andrew Shi is responsible for working on a separate start screen module while Rafael keeps on working the main game module. Using modularity, the only thing that we have to do is to make sure that we keep all the interfaces for each circuit clear before implementing them. This helps getting our work more organized and the system to work. For sharing code, we used GitHub with which we didn't encounter problems. We also used overleaf to make a shared document that we could both edit in LaTeX. Overall, we both believe that we communicated well and gained valuable experience for working in group for this project. This success can be evidenced in the fact that our design works up-to the specifications that we decided. We are proud to have produced Dyno's Adventure.

3.2 Difficulties

The most frustrating part for both of us was the compilation time penalty gained when the sprite gets bigger and bigger. For small sprites that only involves hundreds of pixels (such as dyno), the compilation time is acceptable. However, when the size gets increasingly bigger, it takes a long time to finish compiling a module since a sprite has so many wires and registers. In addition, since our project was highly graphical, it was necessary to frequently test on the board. We needed to see what the changes looked like and the simulated waveforms were always used and were always useful to provide correctness, but we still needed to check the aesthetics. One nice found was Andrew Shi's Python script (based off the original script provided by the staff) that transforms any given image into Intel hex format. Which allows the game to support the really large sprites used in the menu.

Additionally, we spent a significant amount of time before realizing the video driver pads the lower 2 bits for each color provided to the drivers. This initially caused the colors to be inaccurate. Later, this issue was resolved by making small modifications in the video drivers that prevent the loss of color. Now, the colors are much more accurate.

3.3 Tips and Feedback

Since this is the last lab, we wouldn't have tips for future labs. However, one very important tip for the future in any other class is to at least look at the internals of the provided code. First, they can help you have an idea of what's going on and understand the interface better. Second, it is possible that you might spot a bug like we did with the video drivers.

We have very positive feedback about the lab. We had an opportunity to come up with a system of our own and make it alive in just two weeks and a half on the DE1_SoC board. We were able to showcase everything we have learned during the past two quarters, and the experience of working in groups is invaluable.