Contents

1. INTRODUCTION AND OBJECTIVES

This report is a documentation of a game console project done for COE306. The documentation shall be used for future applications.

A simple game console which has the following properties:

1. Has the ability to interface the screen with VGA & HDMI (Only VGA is done)
2. Custom made Game Engine:
    a. Can render basic shapes (Done: rectangles, circles, lines, dots).
    b. Can render basic photo format (Not met).
    c. Can do Physics: forces, fraction, reflection for all shapes (Only reflection and force are DONE for rectangle-shaped objects).
    d. Can do memory management in the chaotic C language world (Dose does not have memory leaks) (DONE).
    e. Easy to program games with its API (for me DONE).
3. The user can switch games at runtime (NOT DONE).
4. Games can be stored in SD card (NOT DONE).
5. Wireless connected Controllers (Only wired connections for now).
6. Support floating points calculation (Only integers are allowed for now).

2. HARDWARE

Most of the hardware is for the VGA interface.

And VGA is all about the timing of VSYNC and HSYNC. This is the used configuration:

VGA Signal 640 x 480 @ 60 Hz Industry standard timing

General timing

Screen refresh rate	60 Hz
Vertical refresh	31.46875 kHz
Pixel freq.	25.175 MHz

Horizontal timing (line)

Polarity of horizontal sync pulse is negative.

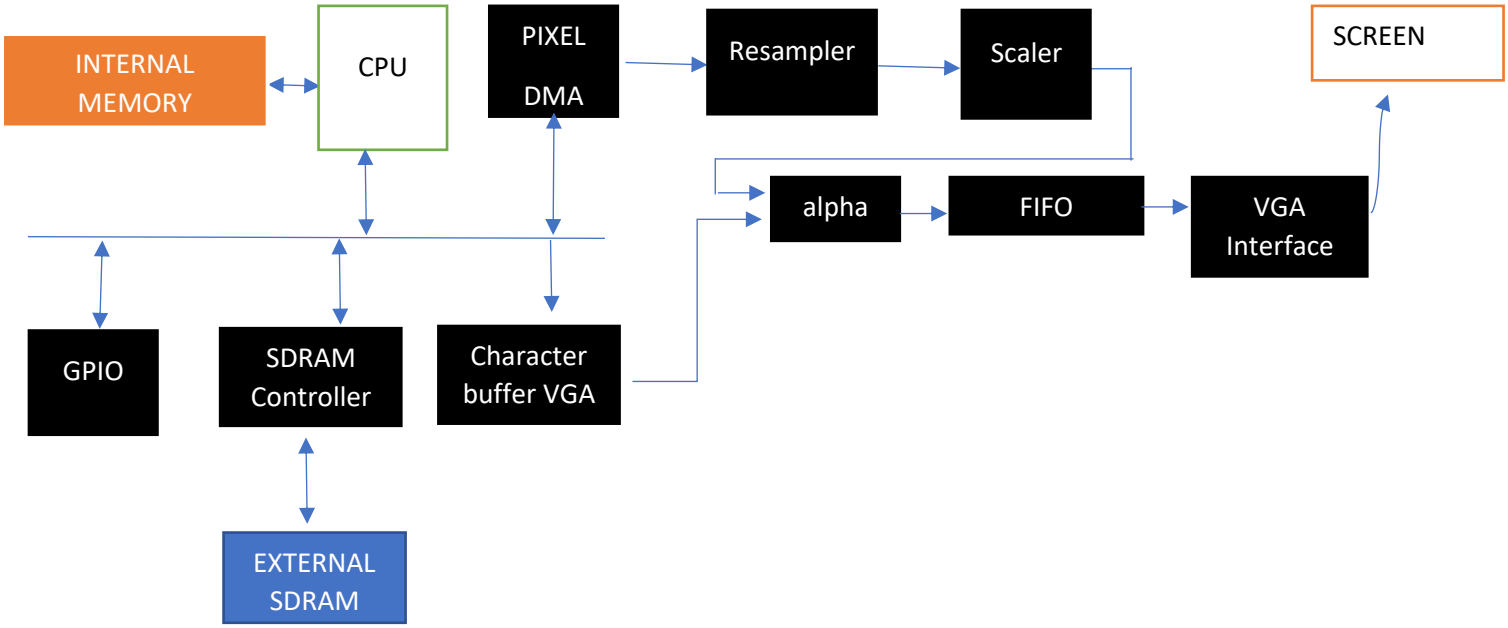
Scanline part	Pixels	Time [µs]
Visible area	640	25.422045680238
Front porch	16	0.63555114200596
Sync pulse	96	3.8133068520357
Back porch	48	1.9066534260179
Whole line	800	31.777557100298

Vertical timing (frame)

Polarity of vertical sync pulse is negative.

Frame part	Lines	Time [ms]
Visible area	480	15.253227408143
Front porch	10	0.31777557100298
Sync pulse	2	0.063555114200596
Back porch	33	1.0486593843098
Whole frame	525	16.683217477656

2.1 Platform



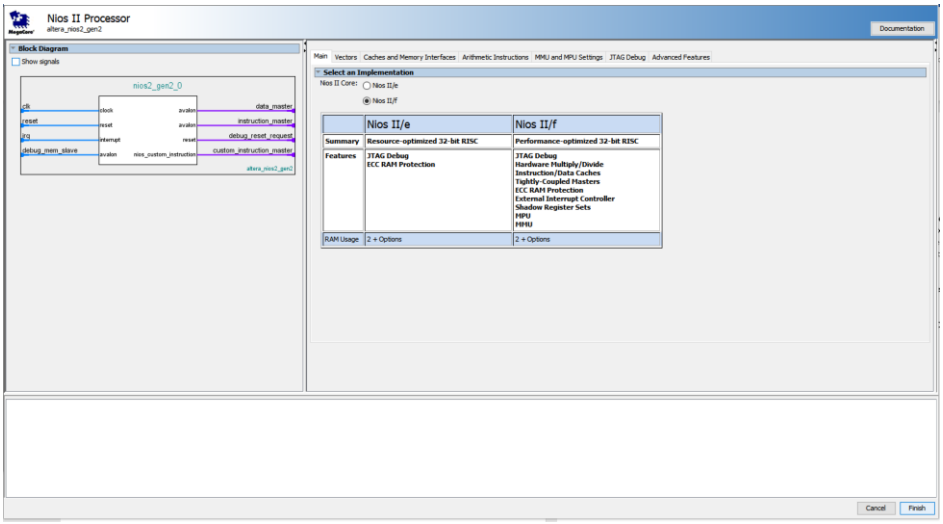
As illustrated in the picture, the current system has the following components:

1- **Nios II processor.**
2- **On-chip memory** for code (.txt segment).
3- **External SDRAM** for .heap, .stack segments and two screen buffers(double buffering).
4- **Pixel DMA** which reads 320x240 pixel each frame from the SDRAM (each pixel is 16-bit).
5- **Re-sampler** which takes 16-bit packed pixels from Pixel DMA and unpacks them to 30-bit.
6- **Scaler** which repeats each pixel 2 times and each row (240 pixels) 2 times to generate a 640X480 pixel image.
7- **Character buffer** which generates the needed pixel for ASCII codes.
8- **Alpha** which merges the result of the scaler(background) with an output of Character buffer(foreground) to get the final picture.
9- **FIFO** because the clock of VGA interface is 25MHZ while Nios is 50 MHZ.
10- **VGA interface**: this will generate the needed VGA signals which are VSYNC, HSYNC, RED, BLUE and GREEN.
11- **GPIO** for user input.
12- **Timer, jtag, systemid , clocks**.

3. COMPONENTS DETAIL

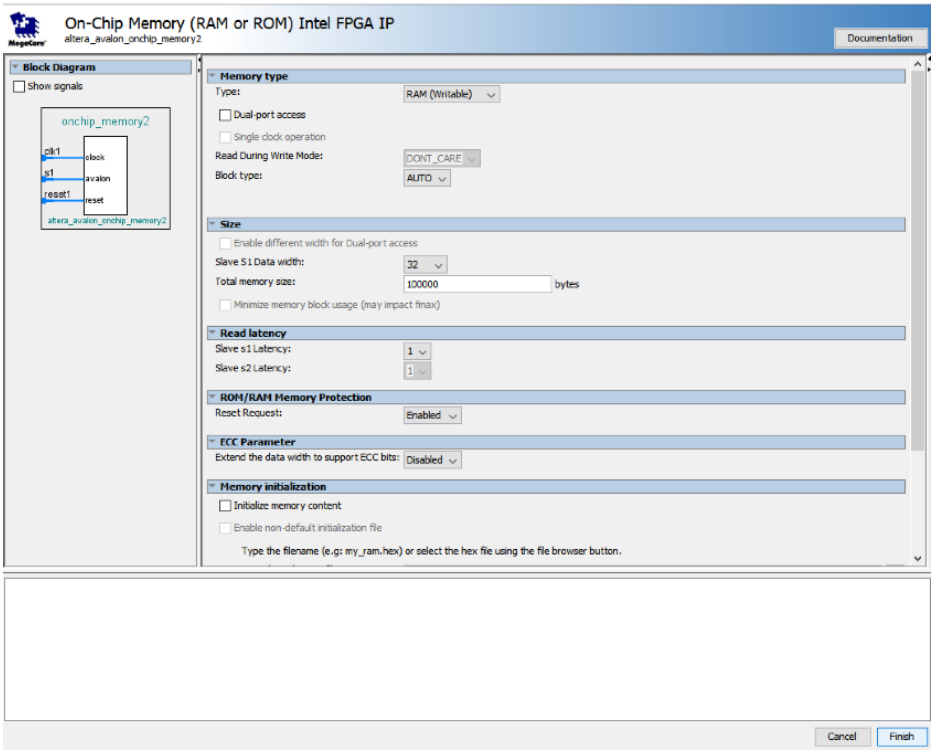
3.1. NIOS II processor:

This CPU will run the Game Engine



3.2. On-Chip memory

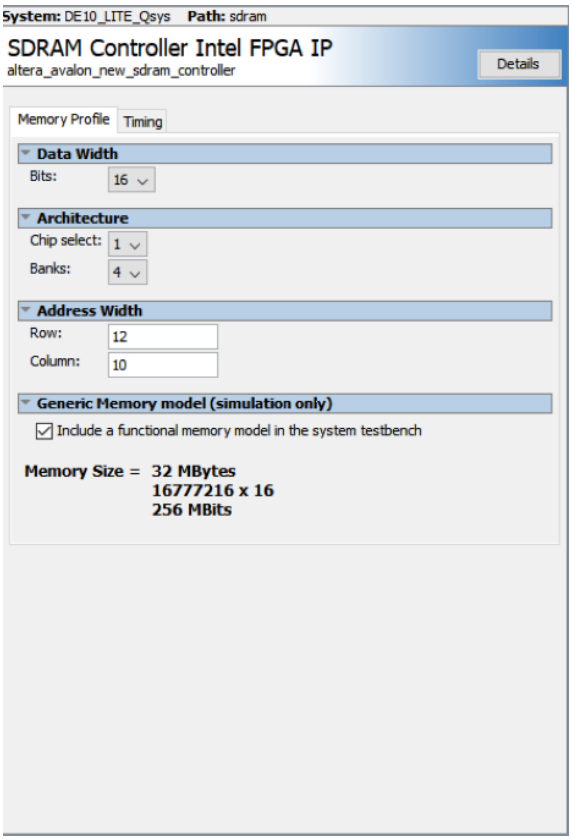
On-Chip memory will hold only the code's .txt section. The memory is not large enough to accommodate the entire program and 2 screen buffers since its resolution is 320X240 with 16-bit pixels.



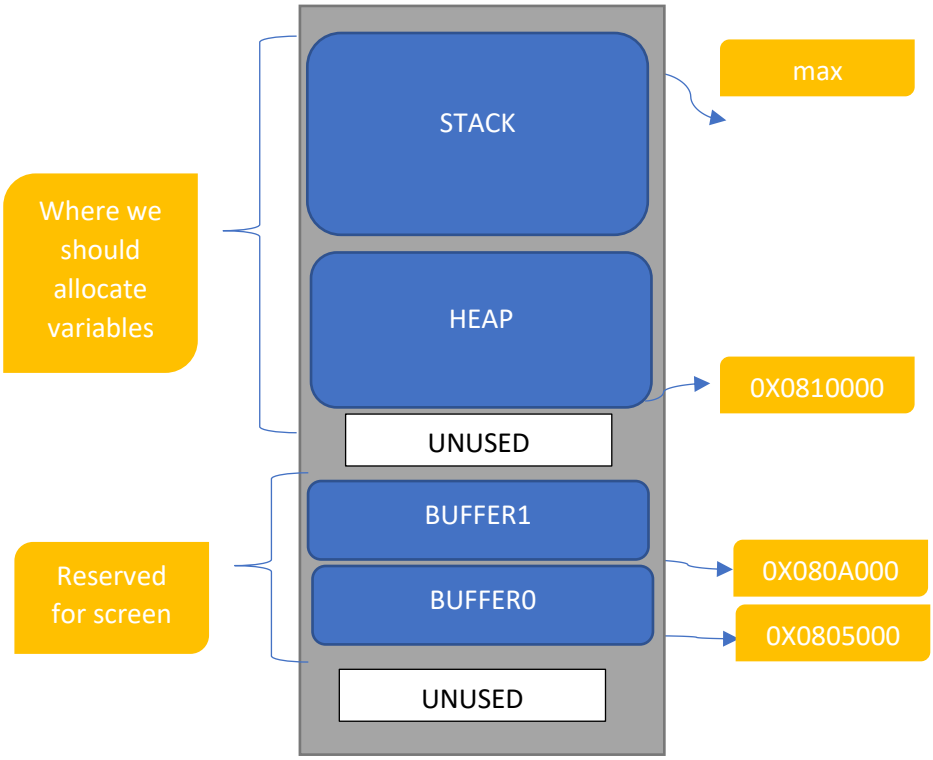
3.3. SDRAM Controller

The SDRAM controller allows the integration of external SDRAM inside the FPGA platform. The SDRAM holds the data segments like .heap, .stack and the two screen buffers.

The linker by default allocate memory for heap at the lowest address and stack at the highest address. But the lowest addresses overlaps with the screen buffers. The image below illustrates how to move the heap segment up above the screen buffers by modifying the linker script.



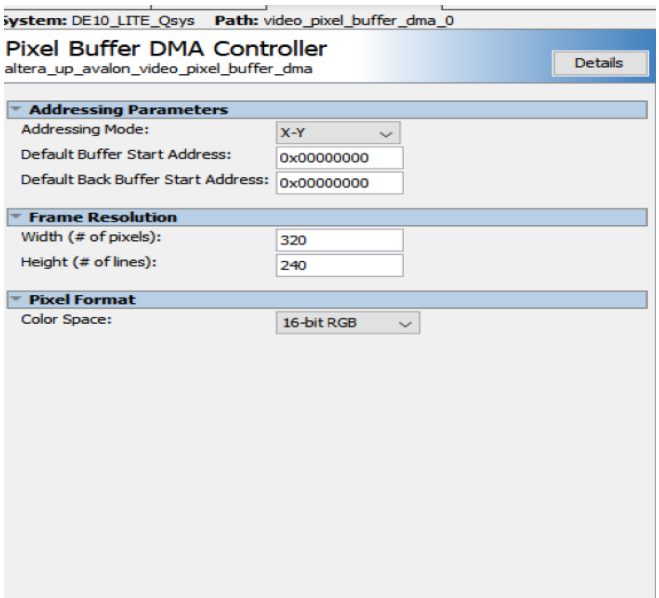

```
406  * This symbol controls where the start of the heap is.  If the stack is
407  * contiguous with the heap then the stack will contract as memory is
408  * allocated to the heap.
409  * Override this symbol to put the heap in a different memory.
410  */
411 PROVIDE( __alt_heap_start     = 0x0810000 );
412 PROVIDE( __alt_heap_limit     = 0x40386a0 );
413
```



3.4. Pixel DMA

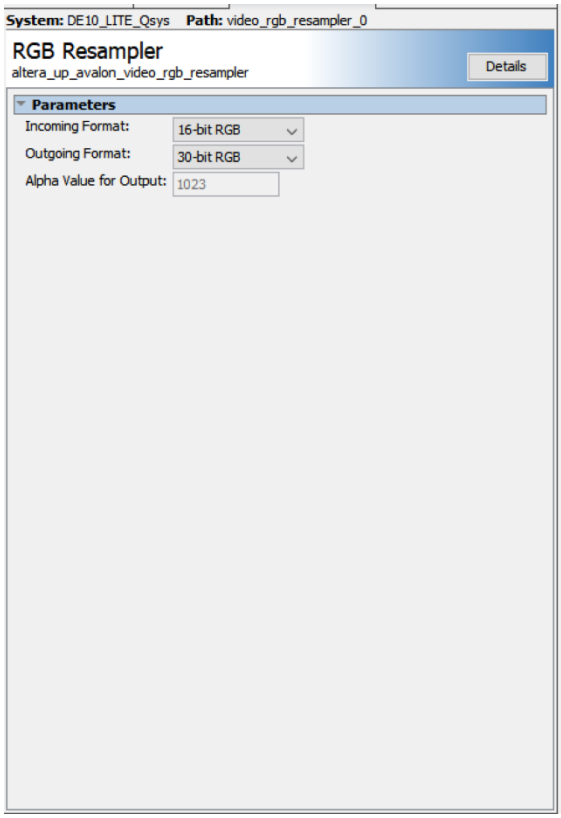
The buffering technique is used in the memory which utilizes the existing two buffers which will be swapped in each frame. The CPU will draw on buffer1 while the pixel DMA will fetch buffer0 and display it. When the CPU finishes drawing, the next frame on buffer1 will ask the DMA to swap buffers. The DMA will then display buffer1 while the CPU will draw the next frame on buffer0.

Each pixel in the buffers is 16-bit wide. In pixel DMA there are two registers to specify the starting address of each buffer which is set to 0X0805000 and 0X080A000 by code.



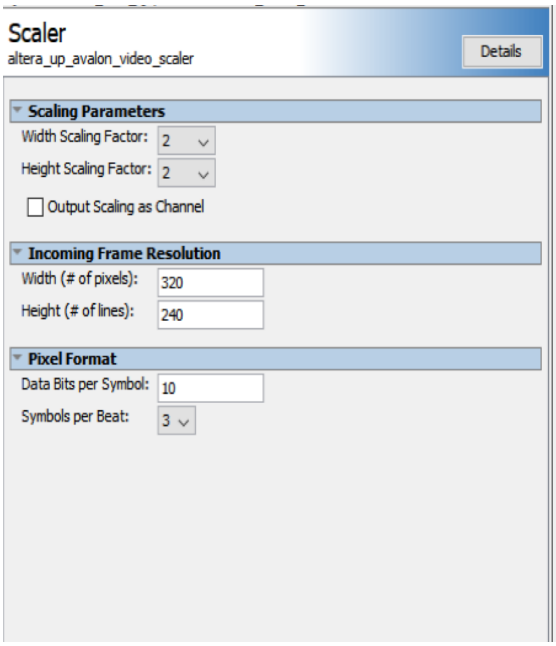
3.5. RGB Re-sampler

The standard pixel used in VGA is not 16-bit but 30-bit, each color is 10-bits. Therefore, to extend the 16-bits we got from the DMA, RGB Re-sampler must be used



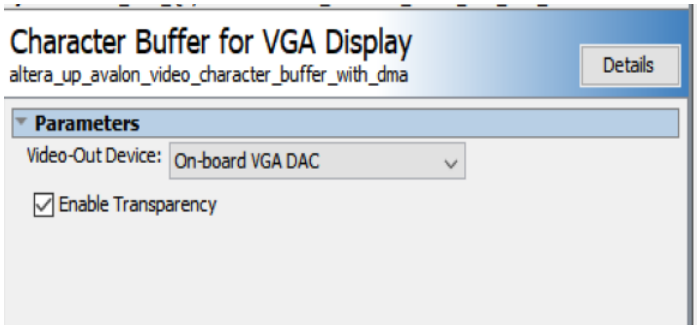
3.6. Scaler

The resolution used in the project is 320X240 pixel which is not supported by VGA standards as shown below. Therefore, a scaler was used to repeat the same pixel 4 times, 2 on the x-axis and 2 on the Y-axis. Now we have 640X480 resolution.



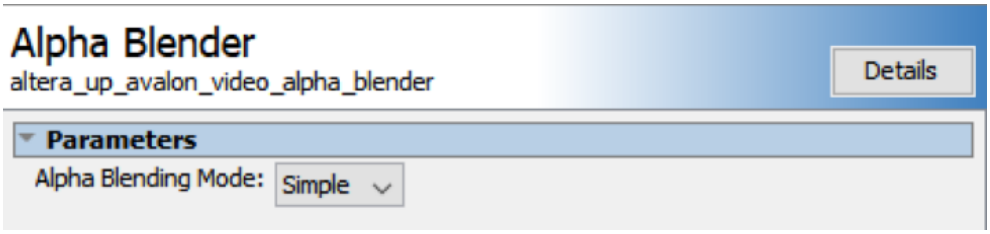
3.7. Character buffer

The character buffer component will get ASCII characters and their location on the screen on its data line from the CPU and store them in its own On-Chip RAM for them to be written out in their proper place. Each character is already stored in the corresponding cell inside the On-Chip RAM and there is an internal DMA which fetches the frame and passes it to the next component one character at a time.



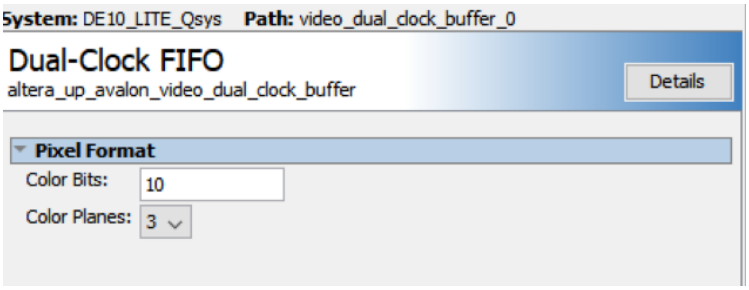
3.8. Alpha

This component will merge both pixels from the Re-sampler and the one from the character buffer into one pixel. The one from the re-sampler is the background and the one from the character is foreground. This new pixel is the final pixel which will be sent to the screen.



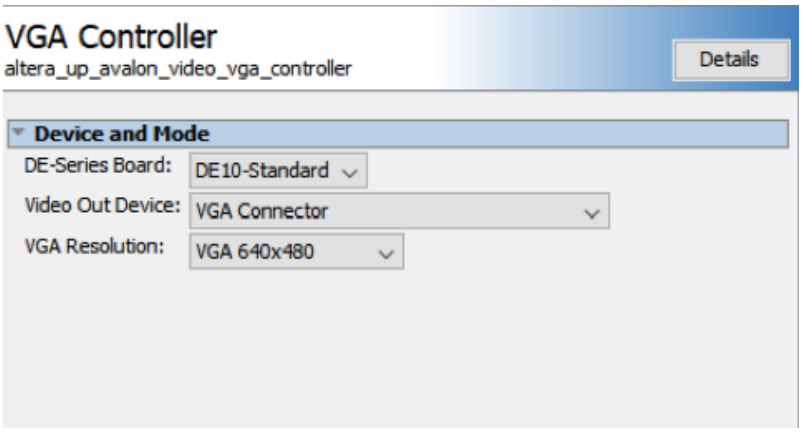
3.9. FIFO

The 640X480 resolution is used to meet the VGA standard of 25MHZ clock. But since the FPGA uses 50MHZ clock, FIFO is implemented which takes input in every 50MHZ and gives output every 35Mhz.



3.10. VGA Controller

According to the VGA specification for the 640X480 we need to produce a VSYNC signal every 64us and VHSYNC every 3.5us. These signals will be generated by the VGA controller. The VGA uses analog voltage for colors. The VGA reads the pixel from the FIFO every 25Mhz and generates the needed voltage using DAC.

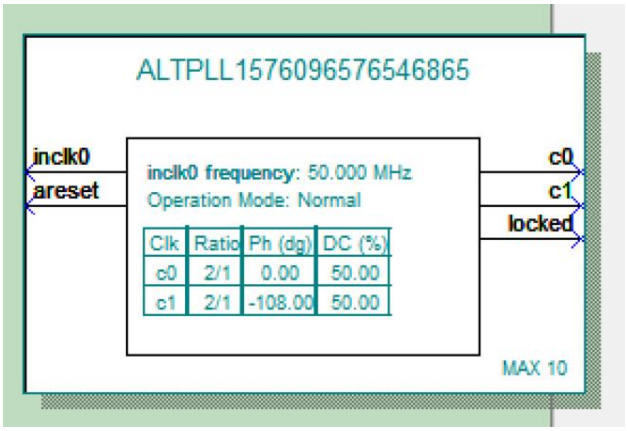


3.11. Clocks

There are three components which are needed for a different clock:

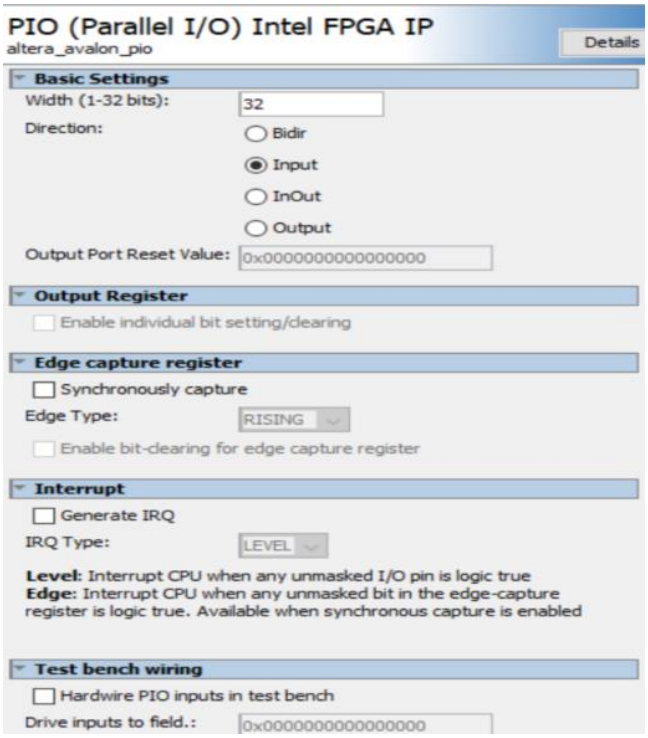
1) 25Mhz: VGA controller and the FIFO
2) 50Mhz but shifted -108 degrees
3) SDRAM controller

These special clocks can be generated using the PLL component.

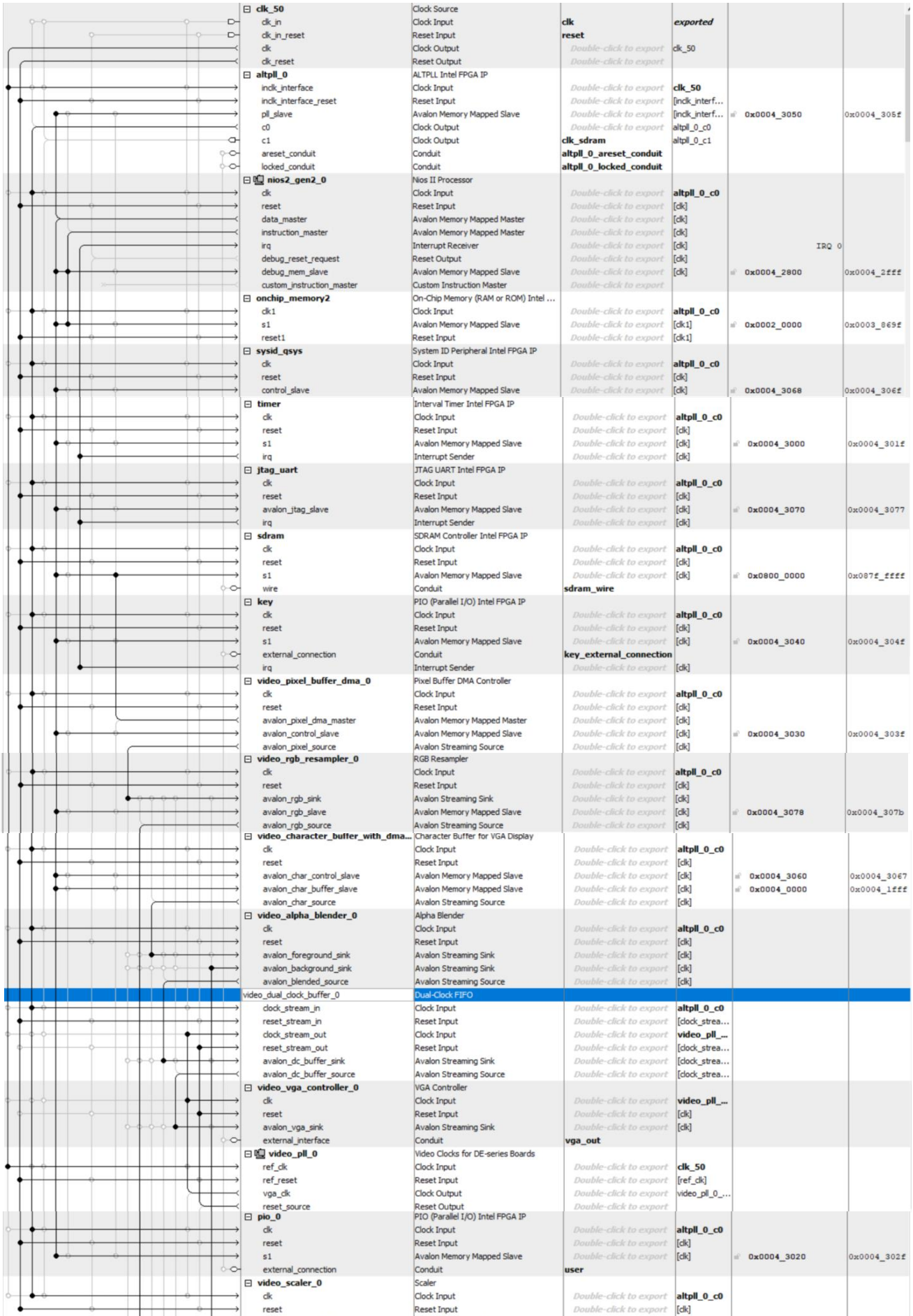


3.12. GPIO

The GPIO is used to make the player able to interact with the game console.



4. PLATFORM DESIGNER(Qsys)



5. CODE

The functional programming methodology is used where every action is a function and since there isn't sufficient space for code as it is stored in On-Chip memory which is quite limited. The code can be divided into the following structures:

a- **VGA driver**: To color screen pixels according to the shape GUI.h.

b- **Math:** Mainly vector manipulation math.h.

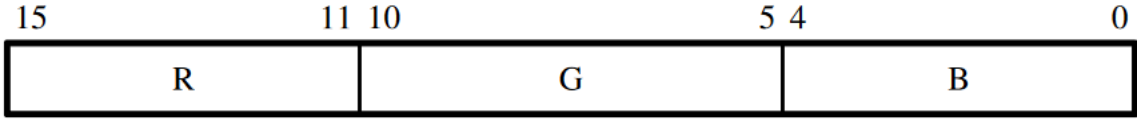
c- **Game Engine:** For objects management, tracking and movement, physics engine game_play.h.

d- **Main function**: To launch the game main.c.

e- **Game examples:** ping_pong.h, pixel_art.h and ball_and_blocks.h.

5.1. VGA driver (GUI.h)

To draw a pixel, we need to set the corresponding address for that pixel to the wanted color. Each pixel is 16-bit short where bits 0-4 are for the blue color, bits 5-10 are for the green color and bits 11-15 are for the red color as shown in the figure below.



In our configuration, we used 320 columns and 240 rows starting from the buffer's base address (buffer0, buffer1). To calculate the address of a pixel (row:10, col:6) the following formula was used:

pixel_address = buffer_base_address + WIDTH*y + x

pixel_address = buffer_base_address + 320 * 10 + 6 ;

***pixel_address = color(red);**

Code snippets of the most important functions from gui.h:

```
short color(char red , char green , char blue){
    return (0x0000 | (red & RED_MASK) << RED_OFFSET | (green & GREEN_MASK) << GREEN_OFFSET | (blue & BLUE_MASK) << BLUE_OFFSET);
}
```

Given the intensity of each color ➔ return the packed color pixel

```
void   set_pixel(unsigned short y , unsigned short x , short value , char backBuffer){
    if(y > ROWNUM){
        //printf("Out of range row %d  with color %d\n" , y , value);
        return ;
    }
    if(x > COLNUM){
            //printf("Out of range column %d  with color %d\n" , x , value);
            return;
    }
    volatile short * address ;
    if(backBuffer)
        address = (short*)pixel->back_buffer_start_address;

    else
        address = (short*)pixel->buffer_start_address;

    address += COLNUM * y ;
    IOWR_16DIRECT(address, x << 1, value);
}
```

Arguments:

a. Pixel position (y,x).

b. Color value.

c. Which buffer to write to (buffer0 , buffer1).

Operation:

- Will write the given color value to the corresponding address (y,x) on the selected buffer.
- IOWR_16DIRECT is like *(address) = x but will bypass caches and write directly to the address.
- The first if statement is to check if the given address is within the range: x < 320, Y < 240 to prevent unwanted errors.

```
void set_box(unsigned short y1 ,unsigned short y2 ,unsigned short x1 ,unsigned short x2 , short value , char backBuffer){
    for(int i = y1 ; i < y2 ; i++)
        for(int k = x1 ; k < x2 ; k++){
            if(x2 > COLNUM &&  y2 < ROWNUM){
                x2 = COLNUM;
                continue;
            }
            if(y2 > ROWNUM){
                    y2 = ROWNUM;
                    continue;
                }
            set_pixel( i , k , value ,backBuffer);

        }

        //alt_up_pixel_buffer_dma_draw_box(pixel, x1, y1, x2, y2, value, 0);
}
```

Arguments:

- Rectangle corners (y0 , x0) to (y1 , x1) where x0 < x1 and y0 < y1.
- Color value.
- Which buffer to write to (buffer0, buffer1).

Operation:

- Will color the pixel contained in the range of the two corners to make a rectangle.

- Also, the if statement checks if the rectangle is inside the screen, if not, just ignore that part and save time.

```
void set_box_center(unsigned short y ,unsigned short x ,unsigned short h ,unsigned short w , short value , char backBuffer){
    short y_start , y_end , x_start , x_end , y_dist , x_dist;
    y_dist = h/2;
    x_dist = w/2;
    y_start = y - y_dist;
    y_end = y + y_dist;

    x_start = x - x_dist;
    x_end = x + x_dist;
    if(y_start  < 0)
            y_start = 0;
    if(x_start < 0)
        x_start = 0;
    set_box(y_start , y_end ,x_start , x_end , value , backBuffer);


}
```

This function is quite useful as it is a wrapper on top of set_box to draw rectangles using the center point, width and height.

Arguments:

- Rectangle center (y0,x0), its width and height.
- Color value.
- Which buffer to write to (buffer0,buffer1).

Operation:

- Will color the pixel contained in the range of the rectangle with center (x,y) and width w, height h.

```
void display(){
    alt_up_pixel_buffer_dma_swap_buffers(pixel);

    while(alt_up_pixel_buffer_dma_check_swap_buffers_status(pixel));
    //while(alt_up_pixel_buffer_dma_check_swap_buffers_status(pixel));

    //set_box(0 , ROWNUM , 0 , COLNUM , 0);

}
```

The above function is called for each frame by the Game Engine to ask for buffer swap(note: a pixel is a global variable).

```
void init(){
    char_buf = alt_up_char_buffer_open_dev("/dev/video_character_buffer_with_dma_0_avalon_char_buffer_slave");
    alt_up_char_buffer_clear(char_buf);
    pixel =  alt_up_pixel_buffer_dma_open_dev("/dev/video_pixel_buffer_dma_0") ;

    alt_up_pixel_buffer_dma_change_back_buffer_address(pixel ,0x080a0000 );
    set_box(0 , ROWNUM , 0 , COLNUM , 0,1 );
    alt_up_pixel_buffer_dma_swap_buffers(pixel);
    while(alt_up_pixel_buffer_dma_check_swap_buffers_status(pixel));

    alt_up_pixel_buffer_dma_change_back_buffer_address(pixel,0x08050000);
    set_box(0 , ROWNUM , 0 , COLNUM , 0,1);
    while(alt_up_pixel_buffer_dma_check_swap_buffers_status(pixel));

}
```

The above function is called at the begging of main() to set up the address of the buffer (0x080a0000, 0x080a0000) and to initialize the memory with 0s.

Summary of gui.h:

```
void set_circle_center (unsigned short y, unsigned short x, unsigned short radius, short color , char backBuffer)
```
draw a circle with center (y,x) and radius color = color , buffer to write to -> backbuffer

```
void my_memcpy(short * base ,short * buffer , unsigned int len )
copy len bytes from buffer to base

void set_row(unsigned short y , short value , char backBuffer)
y -> row number , value -> color , backbuffer -> (0 = front buffer , 1 = back buffer)

void set_col(unsigned short x , short value){
x -> col number , value -> color , backbuffer -> (0 = front buffer , 1 = back buffer)
```

5.2. Math (math.h)

Note: PROGRAMMER SHOULD NOT USE THIS HEADER FILE. THIS FILE IS MEANT TO BE USED INTERNALLY BY THE GAME ENGINE.

The header file *math.h* contains basic vector mathematical operations like adding two vectors, multiplying by a scaler, dot product, cross product, reflection vector and rejection vector.

Vector's (y,x) values can be used to represent location or force direction:

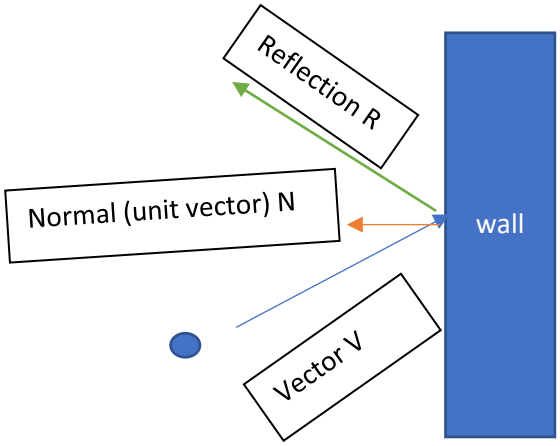
```
struct Vector{
      short x ;
      short y;
};
```

The reflection vector formula calculates the resulting vector after reflecting one vector on a plane:

R = V – 2(Projection of V on N)

```
struct Vector * vector_reflect(struct Vector * v1 , struct Vector *
normal , struct Vector * res){
	struct Vector tmp;
	vector_projection(v1 , normal , &tmp);
	vector_mul(&tmp , 2);
	vector_sub(v1 , &tmp , res);
	//vector_mul(res , -1);
	return res;

}
```



Summary of Math.h:

```
struct Vector *  vector_copy(struct Vector * v1 , struct Vector * v2){
	v1->x = v2->x;
	v1->y = v2->y;
	return v1;
}
```

Copy v2 to v1

```
struct Vector * vector_add(struct Vector *v1 , struct Vector *v2){
	v1->x += v2->x;
	v1->y += v2->y;
	return v1;
}
```

V1 = v1 + v2

```
struct Vector * vector_mul(struct Vector* v1 , short x_scale ){
	v1->x *= x_scale;
	v1->y *= x_scale;
	return v1;

}
```

V1 = x_scale * v1

```
struct Vector * vector_sub(struct Vector * v1 , struct Vector * v2 , struct
Vector * res ){
	res->x = v1->x - v2->x;
	res->y = v1->y - v2->y;
	return res;
}
```

Res = v1 – v2

```
struct Vector * vector_projection(struct Vector * v1 , struct Vector * v2 ,
struct Vector * res){
	vector_copy(res , v2);
	int nem = vector_dot(v1 , v2);
```

```
	int dem = vector_dot(v2 , v2);
	vector_mul(res , nem/dem);
	return res;
}
Res = Projection of v1 on v2
int vector_dot(struct Vector * v1 , struct Vector * v2){
	return v1->x * v2->x + v1->y * v2->y;
}
Dot product of v1 . v2
struct Vector * vector_rejection(struct Vector * v1 , struct Vector * v2 ,
struct Vector * res){
	struct Vector tmp;
	return vector_sub( v1,vector_projection(v1 , v2 ,&tmp)  , res);
}
Res = v1 - reflection(v1,v2)
void print_vector_info(struct Vector * v){
	printf("x = %d , y = %d" , v->x , v->y);
}

For debug purposes .
```

Game Engine (game_play.h) :

This is the largest and most important file. Here we define the Game Engine API to be used by the game programmer.

All game engine logic is manipulating three important data types:

GameObject , MapObject , Controller

Data Types:

1) GameObject the most important struct which represent all and any game element defined by the programmer , definition:

```
struct GameObject{
	char * name;
	struct GameObject * next;
	void (*HitHandler)( struct MapObject * ,struct GameObject * , struct
GameObject *);

	enum SHAPE shape;

	enum HitResponse hit_response;

	struct Vector position;
	struct Vector position_buffer0;
	struct Vector position_buffer1;
	struct Vector force;
	struct Vector force_next;
	short width_raduis;

	short hight;
	short color;
	short id ;
	char buffer_id ;

	char isDynamic;
	char isHitHandler;
};
```

5.3. Struct GameObject

Name	Type	Information
name	char *	Object name useful when we search for an object by name
next	struct GameObject *	Internal linked list
HitHandler (function pointer)	void (*funct)(struct MapObject * ,struct GameObject * , struct GameObject *);	function pointer, when an object collides with this object HitHandler will be invoked HitHandler()
shape	enum SHAPE	Possible values are : NONE , CIRCLE , SEQUARE , LINE , PIONT
hit_response	enum HitResponse	The action is taken by the game engine when another object collides with this one. possible values: STATIC, DYNAMIC, GHOST , IGNORE
position	struct Vector	object position y,x
position_buffer0	struct Vector	object current position on buffer0
position_buffer1	struct Vector	object current position on buffer1
force	struct Vector	Force to be applied on the object by the physics engine
force_next	struct Vector	no use for now
width_raduis	short	If this is a shape = rectangle then this value represents the width. if this is a shape = circle then this value represents the radius

hight	**short**	the height of the object
color;	**short**	object color
id	**short**	object id VERY USEFUL setup automatically by the MapObject
buffer_id	**char**	Which buffer to draw on buffer0, buffer1 set up Automatically by draw_game_object()
buffer_id	**char**	Which buffer to draw on buffer0, buffer1 set up Automatically by draw_game_object()
isDynamic	**char**	if 0 don't apply physics engine on this object
isHitHandler	**char**	If **1** then invoke HitHandler() when collision happen

Usage example :

```
// Draw a rectangle at position (10,10) with height = 20 , width = 30 , color red

      struct  GameObject * rect;
      rect = (struct GameObject *)malloc(sizeof(struct GameObject));
      fill_game_object(rect , "rect" , SEQUARE,20,30 , 10,10,color(-1,0,0));
      draw_game_object(rect);
```

function related to GameObject:

void copy_game_object(**struct** GameObject * dist , **struct** GameObject * src)
copy the members of src to dist

void fill_game_object(**struct** GameObject * obj , **char** * name ,**enum** SHAPE shape, **short** hight , **short** width_raduis , **short** x , **short** y , **short** color)
fill obj content with the given parameters

void delete_game_object(**struct** GameObject * obj)
delete obj from the heap using free function to prevent memory leaks

void print_info(**struct** GameObject * obj)
print obj info for debugging purposes

void draw_sequare(**struct** GameObject * obj)
draw obj of type *SEQUARE* on the buffer uses set_box_centre from gui.h

void hide_game_object(**struct** GameObject * obj)
draw obj on buffer but with the same color as background to hide it

void draw_game_object(**struct** GameObject * obj)
general drawing function works for shape *SEQUARE , CIRCLE*

```
int check_collide_sequare(struct GameObject * o1 ,struct GameObject * o2 )
check if two Gameobject collided with each other used by physics engine
```

5.4. MapObject

This data structure represents the game world. The programmer should add all GameObjects used in the game to the global MapObject called World. The MapObject works like a container and organizer for all GameObject inside the game. The Game Engine will loop through all GameObjects inside the global MapObject called World and apply a physics engine on them then render them on the screen.

```
struct MapObject {
	short id;
	char * name;
	short o_counter;
	short color;
	struct Sequare dim;
	char stop_time ;
	struct GameObject * first;
	 struct GameObject * current;

}
```

Struct MapObject		
Name	Type	Information
id;	**short**	Map id
name;	**char** *	Map name
o_counter	**short**	Number of objects contained within the map
color;	**short**	Map background color
dim	**struct** Sequare	Map dimension two struct Vector represent the corners
stop_time ;	**char**	If this **1** then stop the physics engine
first;	**struct** GameObject *	First object pointer used to loop through all GameObjects inside the map O(n)
current;	**struct** GameObject *	The last added GameObject used to add GameObject at the tail of the linked list O(1)

Usage example:

```
// add two GameObject to the global map World

struct  GameObject * ball;
struct  GameObject * player;
player = (struct GameObject *)malloc(sizeof(struct GameObject));
ball = (struct GameObject *)malloc(sizeof(struct GameObject));


fill_game_object(ball , "main_ball" , SEQUARE, 2 , 2 , 0,0 ,-1 );
ball->isDynamic = 1;
ball->hit_response = DYNAMIC;
ball->force.x = 1;
ball->force.y = -1;

fill_game_object(player , "player" , SEQUARE,  6 , 21 , 0,0,-1);
player->isDynamic = 1;
player->hit_response = STATIC;
player->force.x = 0;
player->force.y = 0;
player->color = color(23,46,184);

game_play_init()

map_add(World , player);
map_add(World , ball);

while(){
      game_engin();

}
// now the game Engine by itself will render and animate both
objects without the interference of the programmer.

function related to MapObject:
void game_play_init()
initialize the global MapObject World and the global Controller
player1
void fill_map_object(struct MapObject * obj , char * name ,short id
,short y1 , short x1 ,short y2 , short x2 ,short color)
used to fill obj with the given parameters
void map_add(struct MapObject * map , struct GameObject * obj)
add GameObject to the map utilizing the internal linked list
capability of GameObjects

void map_empty(struct MapObject * map )
```

```
loop through all GameObjects inside map and call
delete_game_object() on them to free the heap
void map_remove(struct MapObject * map , struct GameObject * obj)
find obj inside map and remove it using delete_game_object()
struct GameObject * map_find_by_name(struct MapObject * map ,  char
* name)
very usefull function to search and return the first occurrence of
GameObject with the given name inside the give map
int within_map(struct  GameObject * obj , struct MapObject * m)
this function used by the Game Engine to check if obj is still
within map dimensions after moving it.
```

5.5. Controller

The simplest data type used by the Game Engine to capture user input in each frame. There is a global Controller called player1 and the main function will pass this global variable to the update function of the game as a parameter. The programmer should check the passed Controller variable in each frame to respond to user input.

```
struct Controller{
      char isActive; // in case of multiple players we need to know how
many controllers is active

      char up;     // user pressed up button
      char down;   // user pressed down button
      char left;   // user pressed left button
      char right; // user pressed right button
      char A;      // user pressed A button
      char B;      // user pressed B button

};
```

Usage Example:

```
void update_block_and_ball(struct Controller * input){
// retrieve objects from map using their names

struct GameObject * ball = map_find_by_name(World , "main_ball");
struct GameObject * player = map_find_by_name(World , "player");

// check user input and move player accordingly
if(input->right)
        player->force.x = 2;
else if(input->left)
        player->force.x = -2;
else
        player->force.x = 0;



}

function related to Controller:
void game_play_init()
initialize the global MapObject World and the global Controller player1
void wait_for_input()
```

```
loop until user press any button
void get_player_input(struct Controller * cnt)
fill cnt with the current user input using Polling
```

5.6. Game Engine

By default, the game engine will work on the World global MapObject.

```
void game_engin(){
	get_player_input(player1);
	if(!World->stop_time)
		physics_engine();

	render();
	display();
}
```

The above function should be called inside the while loop of the main function

Each time this function is invoked:

1- Store user input inside the global Controller variable player1
2- If not stop_time apply the physics engine on all GameObejct inside World.
3- Loop through all GameObjects inside World and draw them on the screen.
4- After drawing the GameObjects ask the DMA pixel buffer for buffer swap to display the newly drawn frame on the screen.

Core function documentation:

```
void get_player_input(struct Controller * cnt){
	int x = ~(IORD_ALTERA_AVALON_PIO_DATA(PIO_0_BASE));

	// x = GPIO data register content
	cnt->left  = (x&1);
	cnt->right = ((x>>1)&1);
	cnt->down  = ((x>>4)&1);
	cnt->up    = ((x>>2)&1);
	cnt->A = ((x>>3)&1);
	cnt->B = ((x>>5)&1);
   }

void render(){
// loop through World and draw each object .
// this function will invoke draw_game_object which will invoke the
suitable shape drawing function from gui.h
for(struct GameObject * obj = World->first ; obj != 0 ; obj = obj-
>next){
		draw_game_object(obj);
	}

   }
```

@Note: to move and animate gameObject just delete them from their old position and redraw them in a new position. This is how animation works.

The following points describe how the game engine draws rectangles:

1- First of all, the game engine checks which buffer are we going to draw on using buffer_id.

2- If we are going to draw on buffer0, then at first we remove the old GameObject from the buffer by drawing the GameObject with the same color as the background (black) using its position_buffer0 then draw the object on buffer0 using position.

3- If we are going to draw on buffer1, then at first we remove the old GameObject from the buffer by drawing the GameObject with the same color as the background (black) using its position_buffer1 then draw the object on buffer1 using position.

This method is just used to clear the buffer because by clearing the buffer then drawing the frame from scratch, the screen flickers when the whole buffer is cleared, thus making the game unplayable. Therefore, this method to used to clear the GameObject in each scene then redraw them in their new position to have animation effects.

```
void draw_sequare(struct GameObject * obj){
      if(obj->buffer_id == 0){
      // delete the GameObject from its old position
      set_box_center(obj->position_buffer0.y ,obj-
      >position_buffer0.x , obj->hight , obj->width_raduis ,
      color(0,0,0),1);

      obj->position_buffer0.x = obj->position.x ;
      obj->position_buffer0.y = obj->position.y ;

      obj->buffer_id = 1;


      }
      else if(obj->buffer_id == 1){
      // delete the GameObject from its old position

      set_box_center(obj->position_buffer1.y ,obj-
      >position_buffer1.x , obj->hight , obj->width_raduis ,
      color(0,0,0),1);

      obj->position_buffer1.x = obj->position.x ;
      obj->position_buffer1.y = obj->position.y ;

      obj->buffer_id = 0;
```

```
      }
// redraw the GameObject in its new postion
set_box_center(obj->position.y ,obj->position.x , obj->hight , obj-
>width_raduis , obj->color,1);

    }
```

5.7. Physics Engine

This is a very simple engine to apply forces and reflection logic.

```
void physics_engine(){
    struct Vector tmp , old_position ;
    int state;
    // loop through all GameObject inside the World MapObject
    for(struct GameObject * obj = World->first ; obj != 0 ; obj = obj->next){

        // *************** APPLAY FORCE *****************//
        // *************** CHECK & HANDLE COLLISION******************//
        // if Not GameObject isDynamic this mean dont apply physics on it
        if(obj->isDynamic){
            // store the current position just in case we need to return it
            vector_copy(&old_position , &obj->position);
            // apply force and move object
            move_by(obj , &obj->force);
            // check if object outside map if so apply reflection for example the ball should not go outside screen
            state = within_map(obj , World);
            // apply reflection force if object gose outside screen
            if(state){
                vector_copy(&obj->position , &old_position);
                if(state == -1 || state == 1)
                    obj->force.x *=-1;
                if(state == -2 || state == 2)
                    obj->force.y *=-1;

                move_by(obj , &obj->force);
            }
            // if object still within screen then check for collision with other objects
            else if(obj->force.x != 0 || obj->force.y != 0){
                // loop through World again and check for collision between obj and other GameObjects inside the MapObject
                for(struct GameObject * obj2 = World->first ; obj2 != 0 ; obj2 = obj2->next){
                    // dont check collision between the object and itself :(
                    if(obj2->id == obj->id)
                      continue;
                   // dont check collision if the other object is set to IGNORE collision
                   if(obj2->hit_response != IGNORE){
                       // check collision
                       state = check_collide_sequare(obj ,obj2);
                       // if collision happen
                       if(state == -1){
                           // check if object is set to GHOST then dont calculate the reflection vector because its GHOST
                           if(obj2->hit_response == STATIC ||obj2->hit_response == DYNAMIC  ){
                           // calculate the reflection vector
                           vector_copy(&obj->position , &old_position);
                           // get the normal unit vector this value depend on which side obj1 hit obj2
                           get_unit_noraml(obj , obj2 , &tmp);
                           // write the reflection vector in obj->force
                           vector_reflect(&obj->force , &tmp , &obj->force);


                           }
                           // if object wants to do some action for example if collision delete object or Game Over message and so on
                           if(obj2->isHitHandler)
                               obj2->HitHandler(World , obj2 , obj);
                       }
                   }

                }
            }


        }
    }
```

Core functions used by physics engine:

```
int check_collide_sequare(struct GameObject * o1 ,struct GameObject * o2 ){
    if(o1->position.x < (o2->position.x + o2->width_raduis/2) && o1->position.x > (o2->position.x - o2->width_raduis/2)){
        //printf("\nObject %s share x with %s " , o1->name , o2->name);
        if(o1->position.y < (o2->position.y + o2->hight/2) && o1->position.y > (o2->position.y - o2->hight/2))
            return -1;
    }
    return 0;
}
```

The above function checks if a collision happens using the idea that two objects cannot occupy the same space. If this happens, then we know that there is a collision. This function is - as its name suggests - only works for rectangular shapes.

Return value:

1. **-1** if collision is detected.
2. **0** if there is no collision.

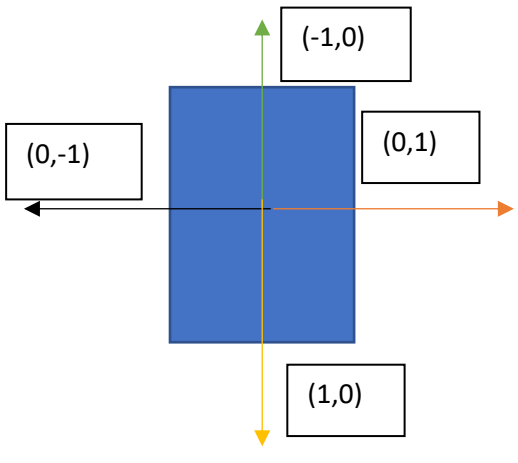
```
void get_unit_noraml(struct GameObject * o1 , struct GameObject * o2 , struct Vector * n){
    short upper_x = (o2->position.x + o2->width_raduis/2);
    short lower_x = (o2->position.x - o2->width_raduis/2);


    if(o1->shape == SEQUARE)
        if(o1->position.x <upper_x  && o1->position.x > lower_x ){
            if(o1->force.y > 0){
                n->x = 0 ;
                n->y = 1;
                return;
            }
            else{
                n->x = 0 ;
                n->y = -1;
                return;
            }


        }

        else
            if(o1->force.x >= 0){
                n->x = -1 ;
                n->y = 0;
                return;
            }
            else if(o1->force.x <= 0){
                n->x = 1 ;
                n->y = 0;
                return;
            }
}
```

This function will fill Vector n with the suitable unit normal vector depending on which side the o1 hits o2.



5.8. Game Code

Each game should implement two functions. The first function is called once at the begging of the main() to populate the World with GameObjects. The second function is called in each frame to do game logic and should be inside the main's while loop.

5.8.1. Game Example: Ping Pong

```
void setup_ping_pong(){
struct  GameObject * player_1;
struct  GameObject * player_2;
struct  GameObject * ball;
player_1 = (struct GameObject *)malloc(sizeof(struct GameObject));
player_2 = (struct GameObject *)malloc(sizeof(struct GameObject));
ball = (struct GameObject *)malloc(sizeof(struct GameObject));

fill_game_object(ball , "main_ball" , SEQUARE, 2 , 2 , 0,0 ,-1 );
ball->isDynamic = 1;
ball->hit_response = DYNAMIC;
ball->force.x = 1;
ball->force.y = -1;
fill_game_object(player_1 , "player_one" , SEQUARE,  21 , 6 , 0,0,-
1);
fill_game_object(player_2 , "player_2" , SEQUARE,  21 , 6 , 0,0,-1);


player_1->isDynamic = 1;
player_1->hit_response = STATIC;
player_1->force.x = 0;
player_1->force.y = 0;
player_2->isDynamic = 1;
player_2->hit_response = STATIC;
player_2->force.x = 0;
player_2->force.y = 0;
set_player_1(player_1);
set_player_2(player_2);
center(ball);

map_add(World , player_2);
map_add(World , player_1);
map_add(World , ball);
}

set_player_1(struct GameObject * x){
      x->position.x = 10;
      x->position.y = ROWNUM/2;

};
set_player_2(struct GameObject * x){
      x->position.x = 300;
      x->position.y = ROWNUM/2;
};

void center(struct GameObject * v){
```

```
	v->position.x = COLNUM / 2;
	v->position.y = ROWNUM / 2;
}



void update_ping_pong(struct Controller * input){
struct GameObject * ball = map_find_by_name(World , "main_ball");
struct GameObject * player_1 = map_find_by_name(World ,
"player_one");
struct GameObject * player_2 = map_find_by_name(World , "player_2");
// get player_1 , player_2 and ball pointers from World
// get user input and change player_1 and player_2 accordingly
if(input->up)
	player_1->force.y = 2;
else if(input->down)
	player_1->force.y = -2;
else
	player_1->force.y = 0;
if(input->A)
	player_2->force.y = 2;
else if(input->B)
	player_2->force.y = -2;
else
	player_2->force.y = 0;
// if ball is behind player_1 then print "Player_2 WON" on the
screen
if(ball->position.x < player_1->position.x - player_1->width_raduis)
{
alt_up_char_buffer_string(char_buf, "Player_2 WON",35, 30);
// if player_2 won then stop the physics engine to stop all
animations
World->stop_time = 1;
// print on the screen Press any button to restart :) .
alt_up_char_buffer_string(char_buf, "Press any button to restart
:)",30, 50);
// wait for user input
wait_for_input();
//game restart logic
//if user press any button remove all messages from the screen
alt_up_char_buffer_clear(char_buf);
//reset ball force to start new game
ball->force.x = 1;
ball->force.y = -1;
//change ball position to the center of the screen
center(ball);
//set players in their positions to start new game
set_player_1(player_1);
set_player_2(player_2);
//restart physics engine to start the game
World->stop_time = 0;
}
// if ball is behind player_2 then print "Player_1 WON" on the
screen
```

```
else if(ball->position.x > player_2->position.x + player_2-
>width_raduis){
alt_up_char_buffer_string(char_buf, "Player_1 WON",35, 30);
World->stop_time = 1;
alt_up_char_buffer_string(char_buf, "Press any button to restart
:)",30, 50);
wait_for_input();
alt_up_char_buffer_clear(char_buf);
ball->force.x = 1;
ball->force.y = -1;
center(ball);
set_player_1(player_1);
set_player_2(player_2);
World->stop_time = 0;
      }
}
```

```
#include <stdio.h>
#include <stdlib.h>
#include "system.h"
#include "alt_types.h"
#include "gui.h"
#include "game_play.h"
#include <stdlib.h>
#include "altera_up_avalon_video_character_buffer_with_dma.h"
#include "ball_and_block.h"
#include "ping_pong.h"
#include "pixel_art.h"

int main()
{
      volatile int  frame_id = 0;
      init();
      game_play_init();
      setup_ping_pong();

      while(1){
      //see how simple the Engine API just add GameObjects to World
and the engine will take care of the reset you just need to focus on
your game logic
            update_ping_pong(player1);
            game_engin();
            frame_id++;
      };
    return 0;
}
```

5.8.2. Game Example: Pixel Art

```
short ink[3] ;
char ink_sq = 0;
void add_ink(short x , short y){
      //this function will spawn a GameObject in position x , y
      struct  GameObject * block;
      short block_h = 2;
      short block_w = 2;
      short c = ink[ink_sq % 3] ;
      // create GameObject
      block = (struct GameObject *)malloc(sizeof(struct
GameObject));
      fill_game_object(block , "ink" , SEQUARE, block_h , block_w ,y
,x ,c);

      block->isDynamic = 0; // No need for physics
      block->hit_response = GHOST;
      // add GameObject to World
      map_add(World , block);

};
void setup_pixel_art(){
            // create object called pen
            struct  GameObject * pen;
            pen = (struct GameObject *)malloc(sizeof(struct
GameObject));
            fill_game_object(pen , "pen" , SEQUARE, 2 ,2 , 0,0 ,-1 );
            pen->isDynamic = 1;//this pen will move around needs
physics
            map_add(World , pen); // pen to World
            center(pen); // change position to the center of the
screen
             // initialize the ink array with 3 colors red green blue
             ink[0] = color(-1,0,0) ;
             ink[1] = color(0,0,-1);
             ink[2] = color(0,-1,0);
}
void update_pixel_art(struct Controller * input){
// get pen pointer from World
      struct GameObject * pen = map_find_by_name(World , "pen");
// move pen according to user input
      if(input->right)
            pen->force.x = 1;
      else if(input->left)
            pen->force.x = -1;
      else if(input->up)
            pen->force.y = -1;
      else if(input->down)
            pen->force.y = 1;
      else{
            pen->force.x = 0;
            pen->force.y = 0;
      }
// if user press A then color the pixel at pen locaton
      if(input->A)
```

```
            add_ink(pen->position.y ,pen->position.x );
// if user press B change the color
      if(input->B)
      ink_sq++;
}
```

```
#include <stdio.h>
#include <stdlib.h>
#include "system.h"
#include "alt_types.h"
#include "gui.h"
#include "game_play.h"
#include <stdlib.h>
#include "altera_up_avalon_video_character_buffer_with_dma.h"
#include "ball_and_block.h"
#include "ping_pong.h"
#include "pixel_art.h"

int main()
{
      volatile int  frame_id = 0;
      init();
      game_play_init();
      setup_ pixel_art();

      while(1){
      //see how simple the Engine API just add GameObjects to World
and the engine will take care of the reset you just need to focus on
your game logic
            update_ pixel_art(player1);
            game_engin();
            frame_id++;
      };
    return 0;
}
```

5.8.3. Game Example: Ball and blocks

```
int level = 0;
void level_builder(){
      if(level == 0)
            for(int k = 0 ; k < 2 ; k++)
                  for(int  i = 0 ; i <10 ; i++)
                        add_block(k ,i);
}
 // THIS IS THE ONLY NEW THING
// this is HitHandler when the ball collide with a block this
function will be invoked with World , obj2 , obj1

void HitHandler_block(struct MapObject * map ,struct  GameObject *
block , struct  GameObject * projectil ){
      // after collision delete the block from screen buffer remove
it from map then free its memory

      hide_game_object(block);
      map_remove(map , block);
      delete_game_object(block);
      // after collision delete the block from screen buffer remove
it from map then free its memory

      // if there is no more object with the name block then the
player wins the game
      if(!map_find_by_name(World,"block")){
            alt_up_char_buffer_string(char_buf, "YOU WON ",35, 30);
            World->stop_time = 1;
      }
}
void setup_block_and_ball(){
      struct  GameObject * ball;
      struct  GameObject * player;
      player = (struct GameObject *)malloc(sizeof(struct
GameObject));
      ball = (struct GameObject *)malloc(sizeof(struct GameObject));


      fill_game_object(ball , "main_ball" , SEQUARE, 2 , 2 , 0,0 ,-1
);
      ball->isDynamic = 1;
      ball->hit_response = DYNAMIC;
      ball->force.x = 1;
      ball->force.y = -1;
      fill_game_object(player , "player" , SEQUARE,  6 , 21 , 0,0,-
1);
      map_add(World , player);
      map_add(World , ball);

      player->isDynamic = 1;
      player->hit_response = STATIC;
      player->force.x = 0;
      player->force.y = 0;
      player->color = color(23,46,184);
```

```
	level_builder();

	center(ball);
	set_player(player);
}

void update_block_and_ball(struct Controller * input){
	struct GameObject * ball = map_find_by_name(World ,
"main_ball");
	struct GameObject * player = map_find_by_name(World ,
"player");


	if(input->right)
		player->force.x = 2;
	else if(input->left)
		player->force.x = -2;
	else
		player->force.x = 0;

	if(ball->position.y > player->position.y + player->hight){
	alt_up_char_buffer_string(char_buf, "GAME OVER ",35, 30);
	World->stop_time = 1;

	alt_up_char_buffer_string(char_buf, "Press any button to
restart :)",30, 50);

	wait_for_input();
	alt_up_char_buffer_clear(char_buf);
	map_empty(World);
	setup_block_and_ball();
	World->stop_time = 0;


	}
}

void add_block(short x , short y){
	struct  GameObject * block;
	short block_h = 8;
	short block_w = 19;
	block = (struct GameObject *)malloc(sizeof(struct
GameObject));
	fill_game_object(block , "block" , SEQUARE, block_h , block_w
, 50 + (block_w +1)* y ,40 +(block_h+1)* x ,-1);
	block->isDynamic = 0;
	block->hit_response = STATIC;
	block->isHitHandler = 1;
	block->HitHandler = HitHandler_block;
	map_add(World , block);



};
```

```
#include <stdio.h>
#include <stdlib.h>
#include "system.h"
#include "alt_types.h"
#include "gui.h"
#include "game_play.h"
#include <stdlib.h>
#include "altera_up_avalon_video_character_buffer_with_dma.h"
#include "ball_and_block.h"
#include "ping_pong.h"
#include "pixel_art.h"

int main()
{
      volatile int  frame_id = 0;
      init();
      game_play_init();
      setup_block_and_ball()
      while(1){
      //see how simple the Engine API just add GameObjects to World
and the engine will take care of the reset you just need to focus on
your game logic
            update_block_and_ball(player1)
            game_engin();
            frame_id++;
      };
    return 0;
}
```

In summary, to program a game that represents each game element as a GameObject you just need to add these elements to the global MapObject called World. This is the only thing outside the game's logic that a programmer should do, the rest depends on game logic.

6. SCOPE AND FUTURE UPDATES

- Add HDMI interface instead of the old VGA
- Add floating-point unit to the processor to do fancy physics
- The game code should be loaded at runtime instead of compile-time
- Move the code's .txt section from the internal memory to the external one
- Add sensors like accelerometer to the player's Controller to make games more enjoyable
- Improve the physics engine to support other shapes like circles
- Improve the physics engine to handle more forces like gravity, fraction etc.
- Improve the renderer to support images like bitmap
- Implementation support for 3D