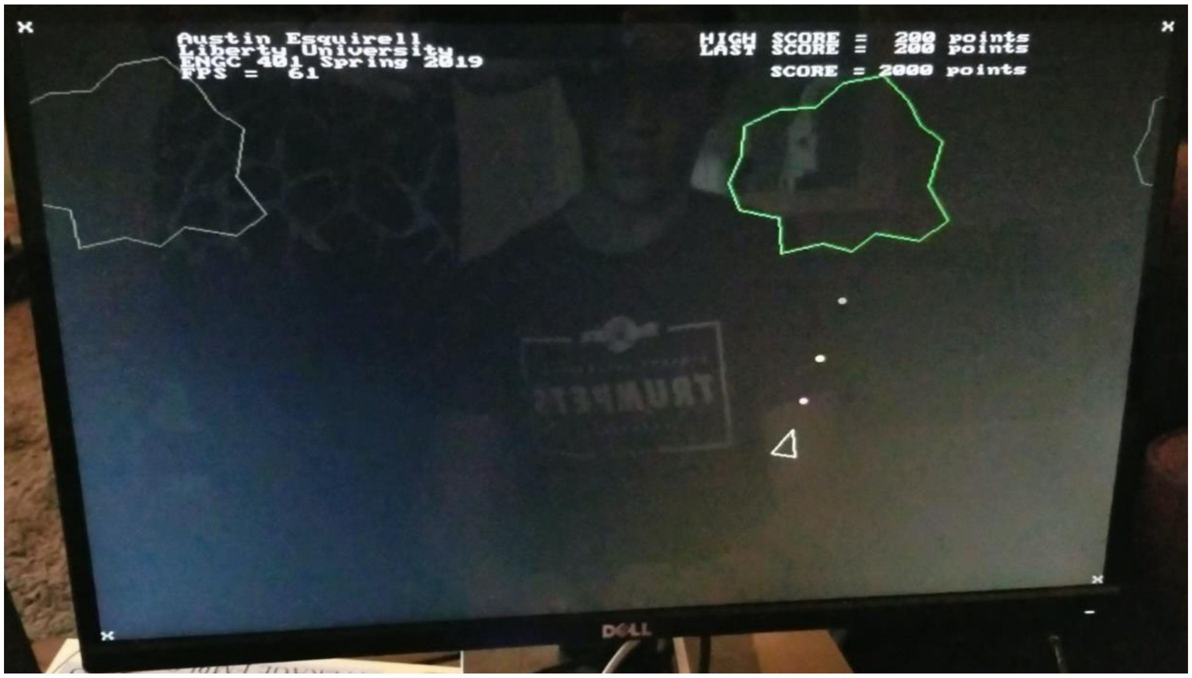
DE1-SOC ASTEROIDS GAME

USING ARM HPS/FPGA INTERFACES AND RANDOM NUMBER HARDWARE ACCELERATION



Introduction

“Asteroids” on the DE1-SoC is a 2D retro arcade-style game, utilizing only the hardware on the device as well as a custom system architecture for implementing the ARM HPS to FPGA interconnect fabric.

Creating a playable game using the Cyclone V FPGA development board is a comprehensive project that fully encompasses the ideas and concepts learned in the classroom. For example, configuring hardware as game controls, applying output display buffering techniques, and developing and utilizing IP cores all involved techniques learned throughout the course. Developing a video game allows for the culmination of many learning objectives in a single project that in the end is both challenging and satisfying.

The classic game “Asteroids” was specifically targeted for development due to its ability to implement and demonstrate a variety of random elements into the gameplay. This was important as the hardware focus of project was on random number generation. Asteroids allows for colors, positions, velocities, directions and graphic models to be randomized, illustrating the ability to quickly generate large amounts of uniformly distributed random numbers in hardware without bogging down the system. Overall, the hardware accelerated randomness adds life to the game without affecting its fundamental nature, playability or performance.

High Level Design

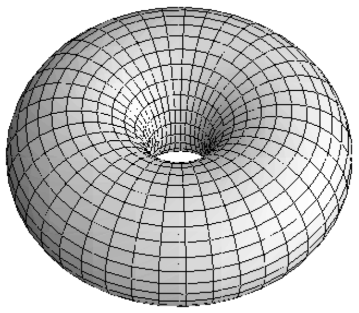
As previously mentioned, rationale for the Asteroids implementation on the HPS/FPGA system involved being able to implement a simple-enough game that could emphasize and showcase the usage of IP core hardware accelerators to enhance the game. In this case, the hardware was random number generation.

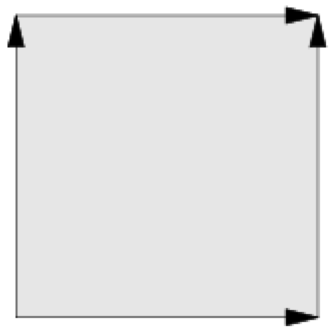
The basis of the “Asteroids” game is simple and similar to original Atari, Inc. design from 1979. It is a space-themed multidirectional shooter, in which the player controls a triangular ship with the ability to rotate left and right, apply a forward accelerating thrust, and shoot bullets forward. The player attempts to shoot floating asteroids to accumulate the largest number of points as possible. If the player collides with an asteroid, the player “dies”, and the game is reset. The game-space is two-dimensional, and incorporates a toroidal coordinate system, meaning that the edges of the screen are wrapped around to the opposing edges.

This game involves some mathematical background knowledge for the full understanding of the logic and design of the game. There are three mathematical topics to be discussed: kinematics, toroidal coordinate wrapping, and number range mapping.

Kinematics is the study of motion. There are some fundamental kinematic equations used in physics to describe object movement in multiple dimensions, and some of those equations are used in this project. The basic principles here to recognize are those relating position to velocity and acceleration. These, of course, are $V = \frac{P_2 - P_1}{t}$ and $A = \frac{V_2 - V_1}{t}$. In software, we can only plot arbitrary positions as pixels on a screen, which can be pretty lifeless. However, when we reference real time by using the the time between frames, and combining that with these equations, we can implement (or at least mimic) life-like kinematic motion.

Asteroids also utilizes a toroidal coordinate system for keeping objects within the screen-space. This is opposed to scrolling or having invisible walls at the edges of the screen. For those not familiar with the term, a torus is a geometric shape that is the result of wrapping around both pairs of opposite edges. This allows the torus to feature the characteristic of having equal coordinate spacing in both the X and Y direction when a 2D space is mapped to it.





torus

In the game, having this toroidal coordinate system is important to allowing our objects to generate and display correctly when approaching and crossing over the edges of the visible screen. First of all, the game's physics and kinematics are not altered when approaching screen edges (as they would be when approaching the poles of a spherical mapping, for example). Objects are rendered smoothly and consistently despite coordinate location. Secondly, this allows objects to wrap around to the other side of the screen when they cross the boundary of the opposite edge. The right and left sides of the screen are wrapped (X coordinates), as well as the top and bottom edges (Y coordinates). This type of coordinate system provides a unique playing experience for the user and programming challenge for the developer.

Finally, the last bit of background math to understand how to map different ranges of numbers to each other. This is necessary when using random numbers streamed in from the hardware, as the numbers are often in a single range that we need to apply to different situations in the program. The random numbers generated by the Altera Random Number Generator IP Core were very large, ranging on average from 8 to 10 decimal digits in length. This is great for variability, but not great for the number range we need to work with. In the software, we only ever need a maximum value of 0.5 from a random number. This problem can be solved by mapping number ranges, and it can be achieved through a simple equation: $Y = (X - A)/(B - A) * (D - C) + C$, where [A,B] is the range of the input and [C,D] is the desired output range. We can first scale the large random number to a known range. In this case we chose to take the first 5 bits of the number to give the range [0,31]. Then the equation $Y = (X_{5:0} - 0)/(31 - 0) * (0.5 - (-0.5)) + (-0.5)$ is used to map this to a random number in the range [-0.5,0.5] for example.

The logical structure of the application is similar to many video games created today. There are three main sections of the code: Outside of main() where global variables and functions are defined. This is pretty standard to any C program, so it will not be discussed in depth here.

Next there is an initializing section during main() that creates and configures important game data, such as graphic model structures, player data, etc. This section is also where we can map physical addresses to virtual addresses for the purposes of accessing hardware on the FGPA through Avalon memory-mapped registers connected to the HPS-to-FPGA lightweight AXI bridge. When components such as the lightweight AXI bridge, pixel buffers, and character buffers are mapped, we can define the variables needed for accessing the pushbuttons, buffer control registers, and FIFO memory (mapped to the random number generator).

The third section is the game loop, or as many game engines call the update function. Most of the game logic occurs here, as this section defines what happens each frame. This is obviously where game objects are updated, pixel drawing routines are called, collision detection happens, and much more. The setup logic and variables and functions defined in the previous sections are called here to make the game come to life.

Program/Hardware Design

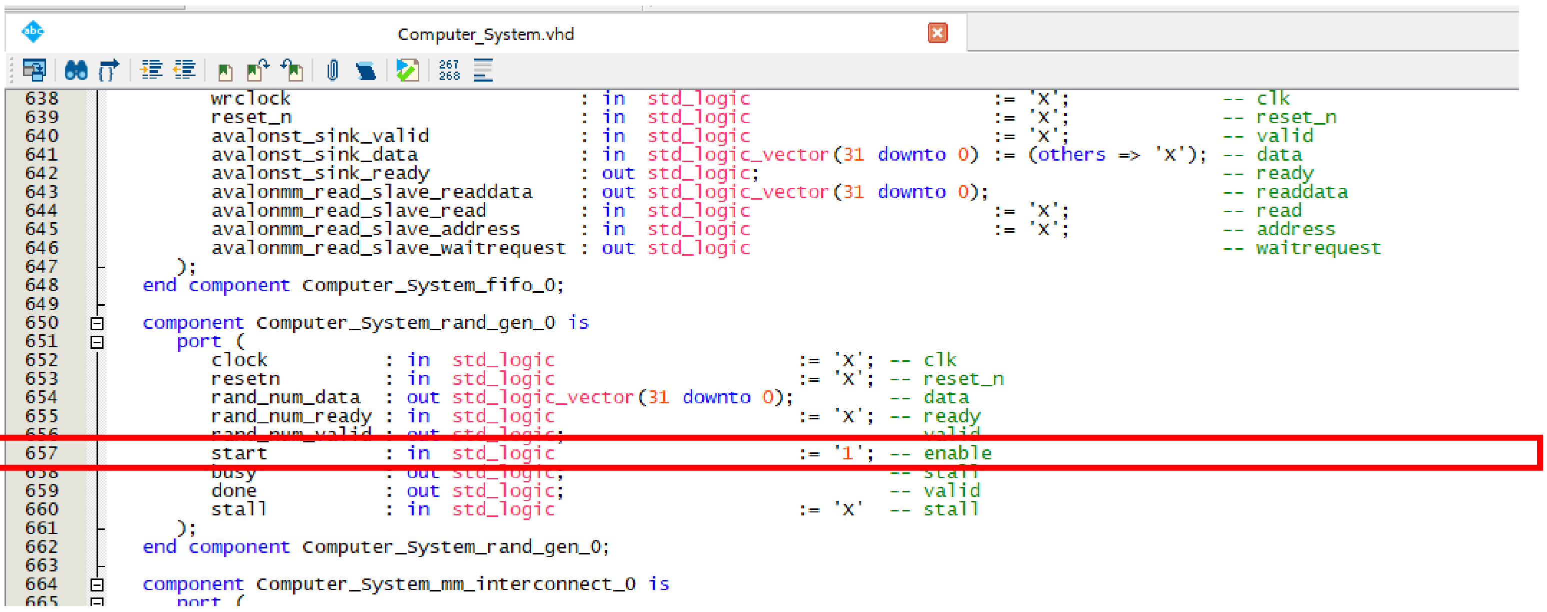
Programming the game logic was a fun challenge with a bit of a learning curve. It required different forms of thinking from how I would normally go about writing code, based on the things I have been taught in class. For example, one of the important concepts in this type of game design is the idea of a separate game space and screen space. The game space is where all of the logic and computations happen, while the screen space is the literal space on the screen. The “background” workings in the game space drive the contents of the screen space. As another example, the position of objects in the game space are represented by a single point, while the screen space objects are represented by fully drawn models. We perform calculations using the points in the game space, and those results are translated to the screen space. This kind of thinking was initially tricky to wrap my head around, but it made sense after some playing around with the code.

Dynamically instantiating, keeping track of, and removing objects from the game on command without causing memory leaks was also something I struggled with a bit during the development process. This game requires the system to keep track of a large number of objects at once, instantiating new objects and deleting old ones where appropriate. To help with some of this memory allocation, I ended up implementing a “vector in C”, which allowed me to use the data type “vector” which had similar functionality to the vector type in C++. This vector implementation allowed me to effectively and dynamically add and delete game objects quickly. The vector class performed the background computations for allocation, reallocating, and freeing memory.

In terms of the hardware development, implementing Altera’s Random Number Generator and Avalon FIFO Memory involved additions to the computer system in the Platform Designer, as well as some slight VHDL code modification. Both the RNG and FIFO Memory need to be added to QSYS/Platform Designer, and they are connect as one might expect. They both need to connect to the same system clock and reset, and then the random number data from the Avalon Streaming Source connects to the Avalon Streaming Sink on the FIFO Memory. From there, the memory mapped slave on the FIFO memory needs to connect to the HPS to FPGA lightweight AXI master bus to allow us to assign a virtual address to this memory in software. The conduit port on the RNG also needs to be exported.

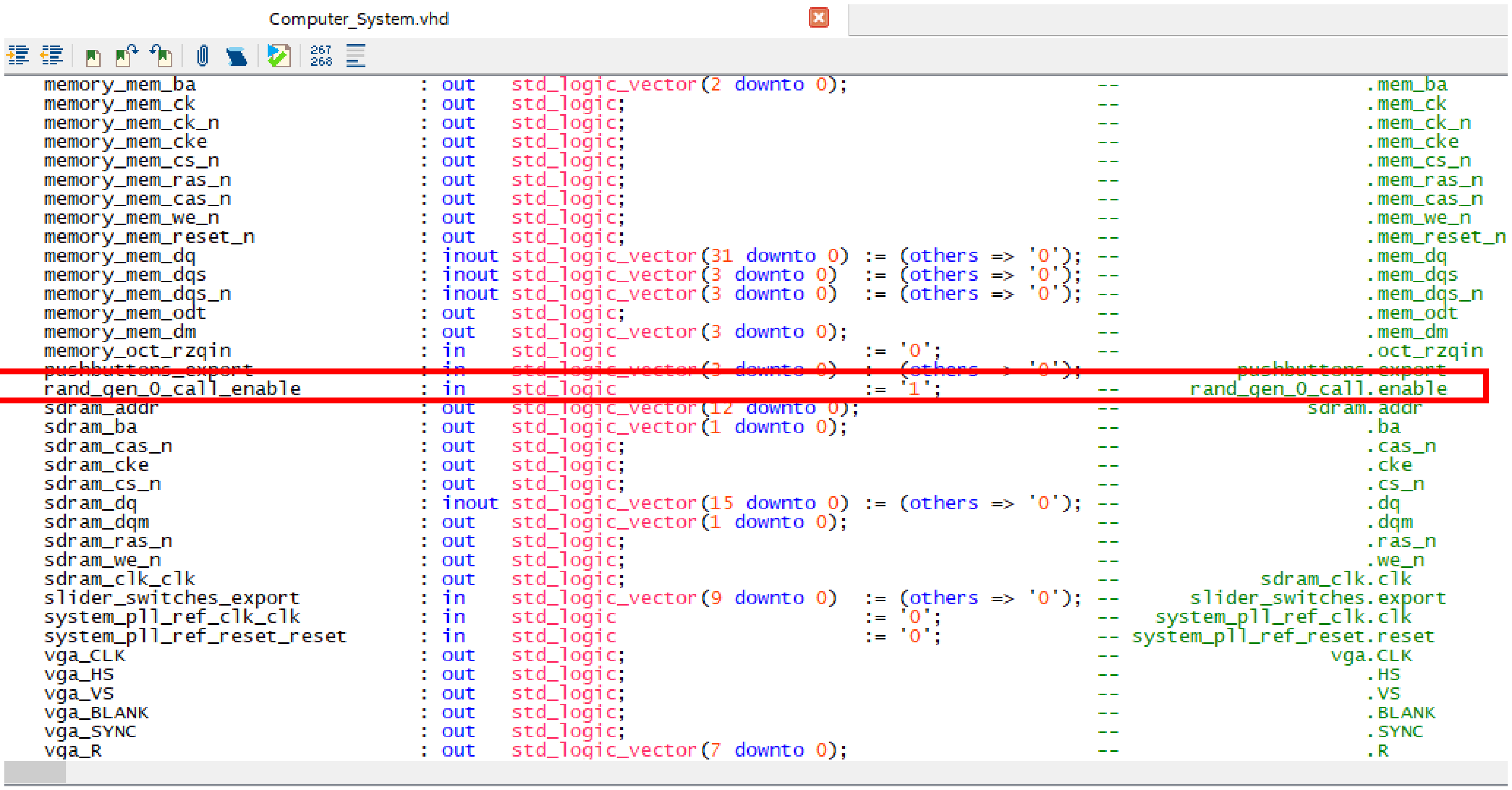
It was the conduit connection on the random number generator that caused the biggest problem. The idea of connecting the random number generator to a memory block that had an input streaming sink and an output memory mapped register made sense, but I did not originally notice the conduit connection on the RNG was for enabling or starting the random number generation on the device. The simple failure to recognize the conduit port on the RNG was why I did not have these components working for the original demonstration of the project. However, now with the proper understanding of the Avalon components, we find that the designed system functions exceptionally.

The conduit was actually enabled in the system by modifying the VHDL code that instantiated the IP cores in the overall computer system. See the two screen shots below. Modifying the VHDL code was not difficult, but necessary for the proper functioning of the system in total. Other than that, the rest of the new hardware design was able to be implemented using the Platform Designer, and screenshots of the new system can be found below.



```
      wrclock                         : in  std_logic                     := 'X';             -- clk
      reset_n                         : in  std_logic                     := 'X';             -- reset_n
      avalonst_sink_valid             : in  std_logic                     := 'X';             -- valid
      avalonst_sink_data              : in  std_logic_vector(31 downto 0) := (others => 'X'); -- data
      avalonst_sink_ready             : out std_logic;                                        -- ready
      avalonmm_read_slave_readdata    : out std_logic_vector(31 downto 0);                    -- readdata
      avalonmm_read_slave_read        : in  std_logic                     := 'X';             -- read
      avalonmm_read_slave_address     : in  std_logic                     := 'X';             -- address
      avalonmm_read_slave_waitrequest : out std_logic                                         -- waitrequest
   );
end component Computer_System_fifo_0;

component Computer_System_rand_gen_0 is
   port (
      clock          : in  std_logic                     := 'X'; -- clk
      resetn         : in  std_logic                     := 'X'; -- reset_n
      rand_num_data  : out std_logic_vector(31 downto 0);        -- data
      rand_num_ready : in  std_logic                     := 'X'; -- ready
      rand_num_valid : out std_logic;                            -- valid
      start          : in  std_logic                     := '1'; -- enable
      busy           : out std_logic;                            -- stall
      done           : out std_logic;                            -- valid
      stall          : in  std_logic                     := 'X'  -- stall
   );
end component Computer_System_rand_gen_0;

component Computer_System_mm_interconnect_0 is
   port (
```

Enable Random Number Generator in VHDL


Computer_System.vhd

```
    memory_mem_ba                  : out   std_logic_vector(2 downto 0);                         --               .mem_ba
    memory_mem_ck                  : out   std_logic;                                            --               .mem_ck
    memory_mem_ck_n                : out   std_logic;                                            --               .mem_ck_n
    memory_mem_cke                 : out   std_logic;                                            --               .mem_cke
    memory_mem_cs_n                : out   std_logic;                                            --               .mem_cs_n
    memory_mem_ras_n               : out   std_logic;                                            --               .mem_ras_n
    memory_mem_cas_n               : out   std_logic;                                            --               .mem_cas_n
    memory_mem_we_n                : out   std_logic;                                            --               .mem_we_n
    memory_mem_reset_n             : out   std_logic;                                            --               .mem_reset_n
    memory_mem_dq                  : inout std_logic_vector(31 downto 0) := (others => '0'); --               .mem_dq
    memory_mem_dqs                 : inout std_logic_vector(3 downto 0)  := (others => '0'); --               .mem_dqs
    memory_mem_dqs_n               : inout std_logic_vector(3 downto 0)  := (others => '0'); --               .mem_dqs_n
    memory_mem_odt                 : out   std_logic;                                            --               .mem_odt
    memory_mem_dm                  : out   std_logic_vector(3 downto 0);                         --               .mem_dm
    memory_oct_rzqin               : in    std_logic                     := '0';                 --               .oct_rzqin
    pushbuttons_export             : in    std_logic_vector(3 downto 0)  := (others => '0'); --     pushbuttons.export
    rand_gen_0_call_enable         : in    std_logic                     := '1';                 --     rand_gen_0_call.enable
    sdram_addr                     : out   std_logic_vector(12 downto 0);                        --          sdram.addr
    sdram_ba                       : out   std_logic_vector(1 downto 0);                         --               .ba
    sdram_cas_n                    : out   std_logic;                                            --               .cas_n
    sdram_cke                      : out   std_logic;                                            --               .cke
    sdram_cs_n                     : out   std_logic;                                            --               .cs_n
    sdram_dq                       : inout std_logic_vector(15 downto 0) := (others => '0'); --               .dq
    sdram_dqm                      : out   std_logic_vector(1 downto 0);                         --               .dqm
    sdram_ras_n                    : out   std_logic;                                            --               .ras_n
    sdram_we_n                     : out   std_logic;                                            --               .we_n
    sdram_clk_clk                  : out   std_logic;                                            --      sdram_clk.clk
    slider_switches_export         : in    std_logic_vector(9 downto 0)  := (others => '0'); --   slider_switches.export
    system_pll_ref_clk_clk         : in    std_logic                     := '0';                 --   system_pll_ref_clk.clk
    system_pll_ref_reset_reset     : in    std_logic                     := '0';                 -- system_pll_ref_reset.reset
    vga_CLK                        : out   std_logic;                                            --            vga.CLK
    vga_HS                         : out   std_logic;                                            --               .HS
    vga_VS                         : out   std_logic;                                            --               .VS
    vga_BLANK                      : out   std_logic;                                            --               .BLANK
    vga_SYNC                       : out   std_logic;                                            --               .SYNC
    vga_R                          : out   std_logic_vector(7 downto 0);                         --               .R
```

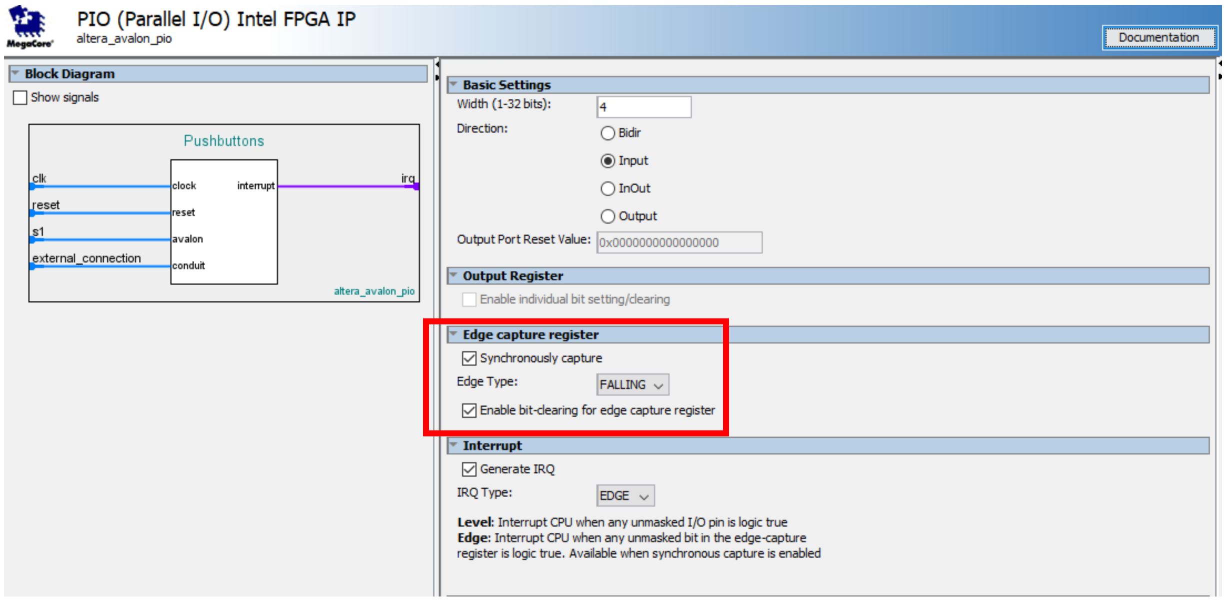
Enable Random Number Generator in VHDL

There was also hardware that was not new to this class, but modified for the purposes of this project. The pushbuttons and VGA components were used here, and modified in the Platform Designer to meet our needs. Changes to the pushbuttons involve changes to the edgecapture register to capture the falling edge of the pushbutton, instead of the rising edge. This translates to detecting the release of the button instead of the initial push. This is the functionality we wanted in the game. One bullets is able to be fired upon each button release. This change can be seen below.

The VGA components also were modified from our usage of them in the previous labs. The previous labs implemented a 320x240 screen resolution with 16-bit colors. In software, this meant writing to the screen required 9 bits for the x value and 8 bits for the y value. Also, the pixel addresses in the buffer were shifted one bit to the left due to the utilization of 16-bit color. The new computer system, however, uses a 640x480 resolution with 8-bit colors. This means the x value requires 10 bits and the y value requires 9 bits, and the addresses do not need to be shifted due to the usage of 8-bit colors, as per Intel's specifications.

Utilization of these components requires knowledge on a hardware and software level, much of which were taught through different Intel/Altera labs demonstrated throughout the semester.

Pushbutton Register configuration



PIO (Parallel I/O) Intel FPGA IP
altera_avalon_pio

Block Diagram
☐ Show signals

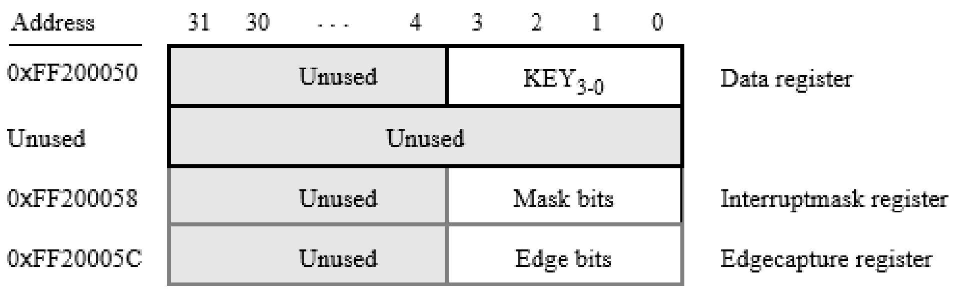
Basic Settings
Width (1-32 bits): 4
Direction: Bidir [✓ Input] InOut Output
Output Port Reset Value: 0x0000000000000000

Output Register
☐ Enable individual bit setting/clearing

Edge capture register
☑ Synchronously capture
Edge Type: FALLING
☑ Enable bit-clearing for edge capture register

Interrupt
☑ Generate IRQ
IRQ Type: EDGE

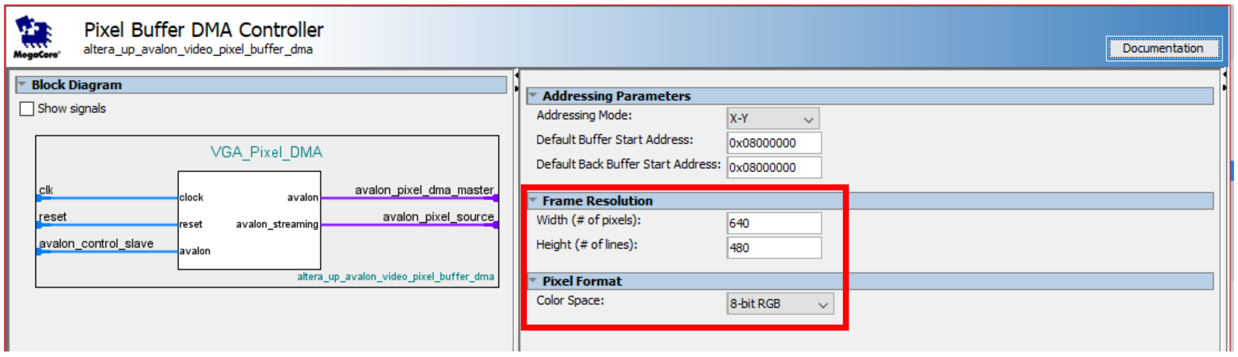
Level: Interrupt CPU when any unmasked I/O pin is logic true
Edge: Interrupt CPU when any unmasked bit in the edge-capture register is logic true. Available when synchronous capture is enabled



Address	31 30 . . . 4	3 2 1 0	
0xFF200050	Unused	KEY$_{3\text{-}0}$	Data register
Unused	Unused		
0xFF200058	Unused	Mask bits	Interruptmask register
0xFF20005C	Unused	Edge bits	Edgecapture register

Pushbutton register

640x480 screen resolution configuration



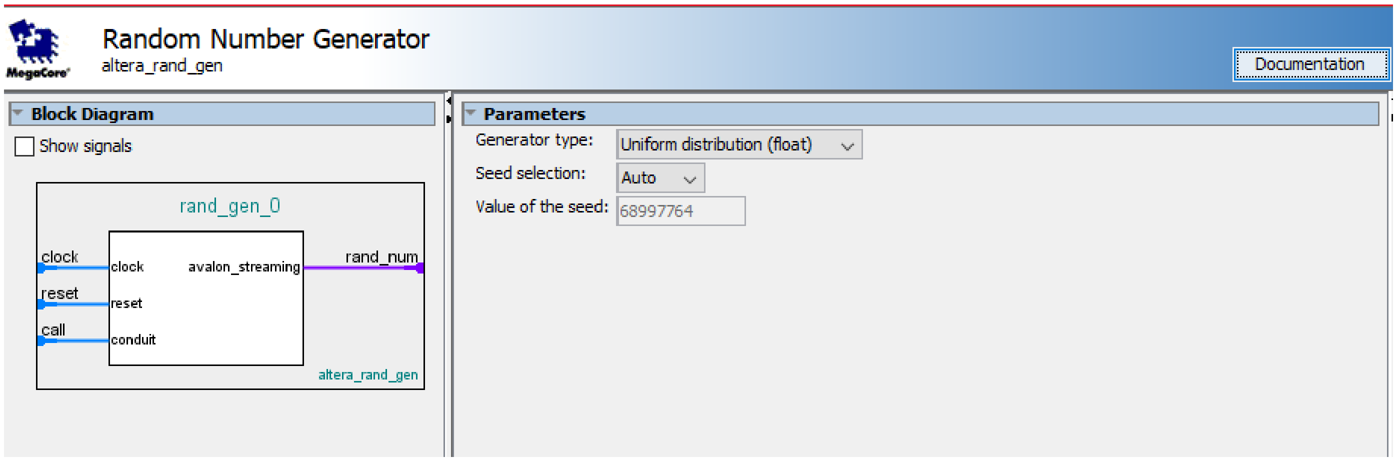
Pixel Buffer DMA Controller
altera_up_avalon_video_pixel_buffer_dma

Addressing Parameters
Addressing Mode: X-Y
Default Buffer Start Address: 0x08000000
Default Back Buffer Start Address: 0x08000000

Frame Resolution
Width (# of pixels): 640
Height (# of lines): 480

Pixel Format
Color Space: 8-bit RGB

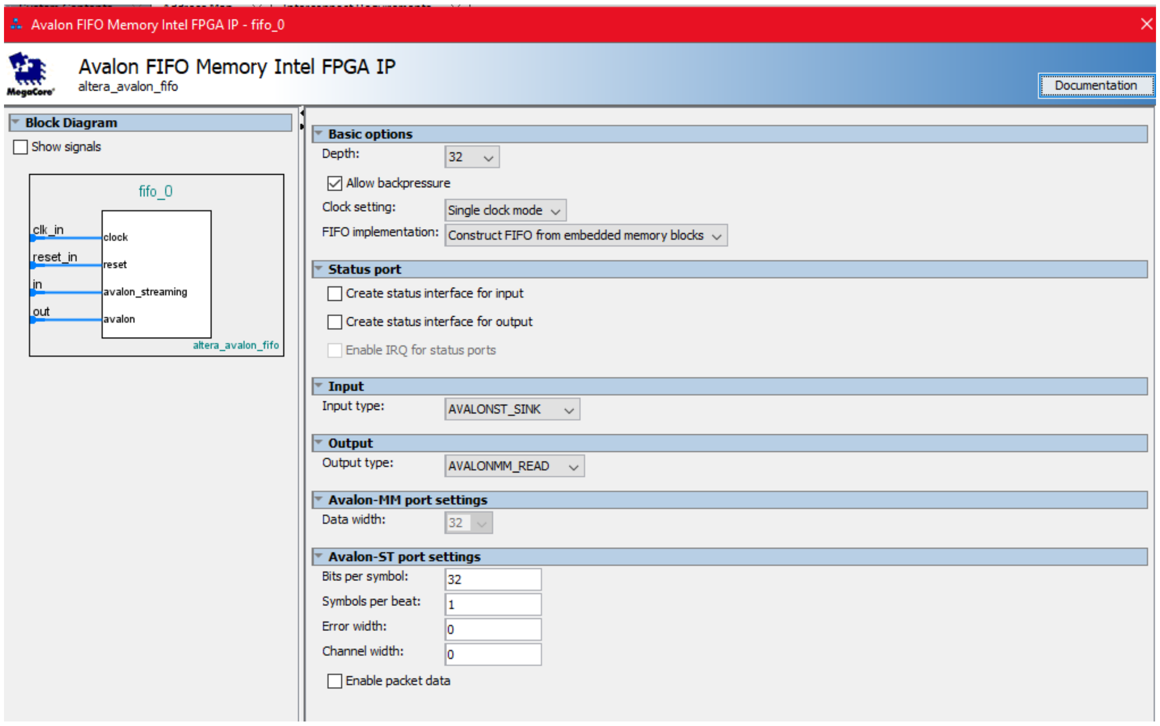
Random Number Generator configuration



Random Number Generator
altera_rand_gen

Parameters
Generator type: Uniform distribution (float)
Seed selection: Auto
Value of the seed: 68997764

Avalon FIFO Memory Core configuration



Basic options
Depth: 32
[✓] Allow backpressure
Clock setting: Single clock mode
FIFO implementation: Construct FIFO from embedded memory blocks

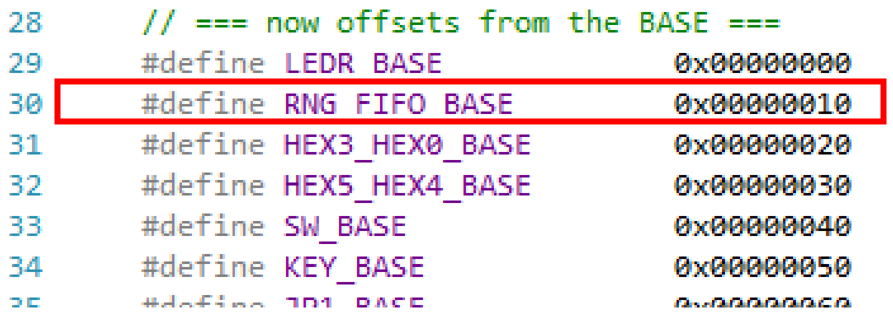
Status port
[ ] Create status interface for input
[ ] Create status interface for output
[ ] Enable IRQ for status ports

Input
Input type: AVALONST_SINK

Output
Output type: AVALONMM_READ

Avalon-MM port settings
Data width: 32

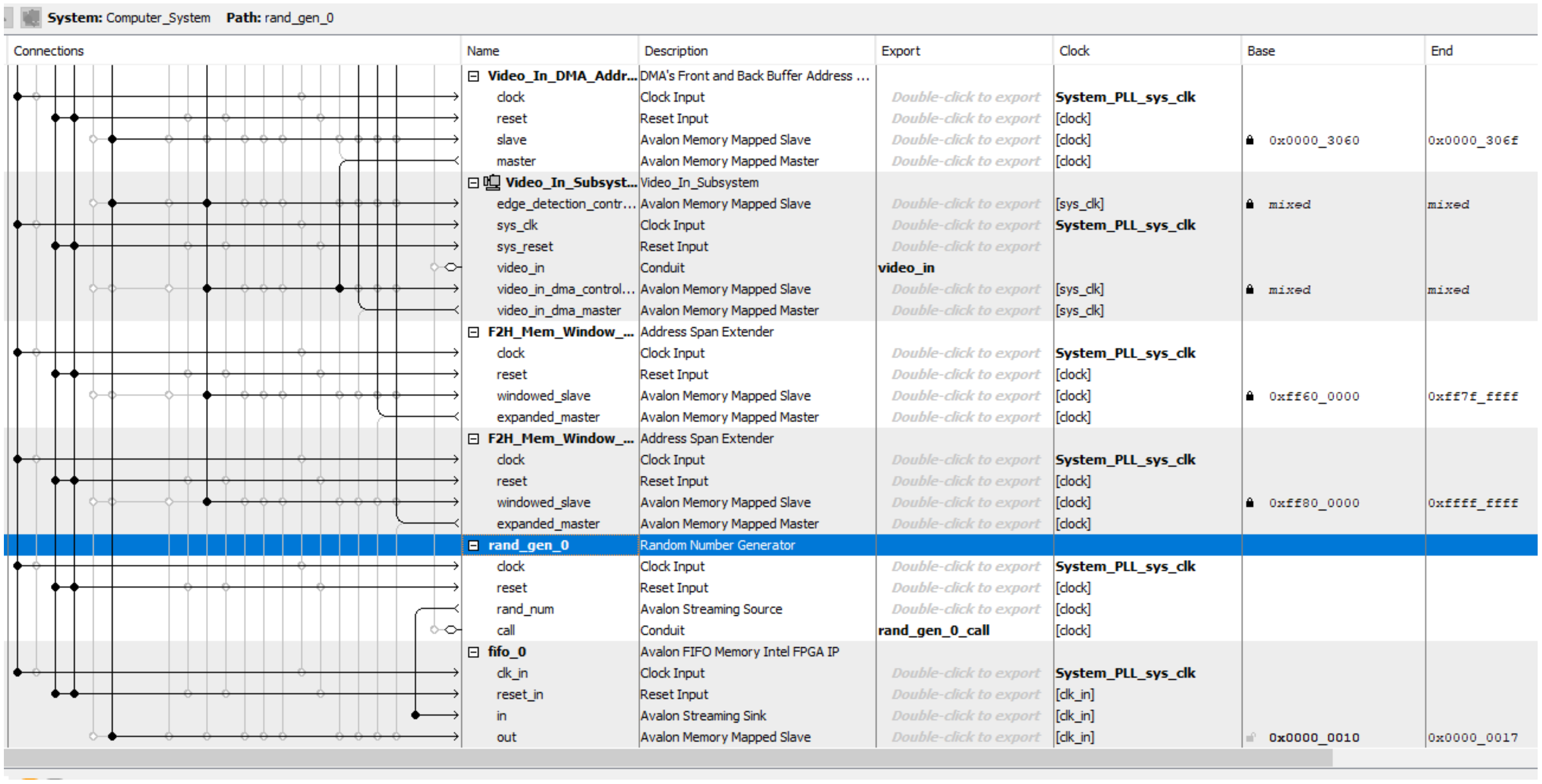
Avalon-ST port settings
Bits per symbol: 32
Symbols per beat: 1
Error width: 0
Channel width: 0
[ ] Enable packet data



```
// === now offsets from the BASE ===
#define LEDR_BASE               0x00000000
#define RNG_FIFO_BASE           0x00000010
#define HEX3_HEX0_BASE          0x00000020
#define HEX5_HEX4_BASE          0x00000030
#define SW_BASE                 0x00000040
#define KEY_BASE                0x00000050
```

New address needed to be added to address header file. This address was provided in Qsys/ Platform Designer. It allows us to map the physical address of the Avalon FIFO Memory to a virtual address we can use in the software.

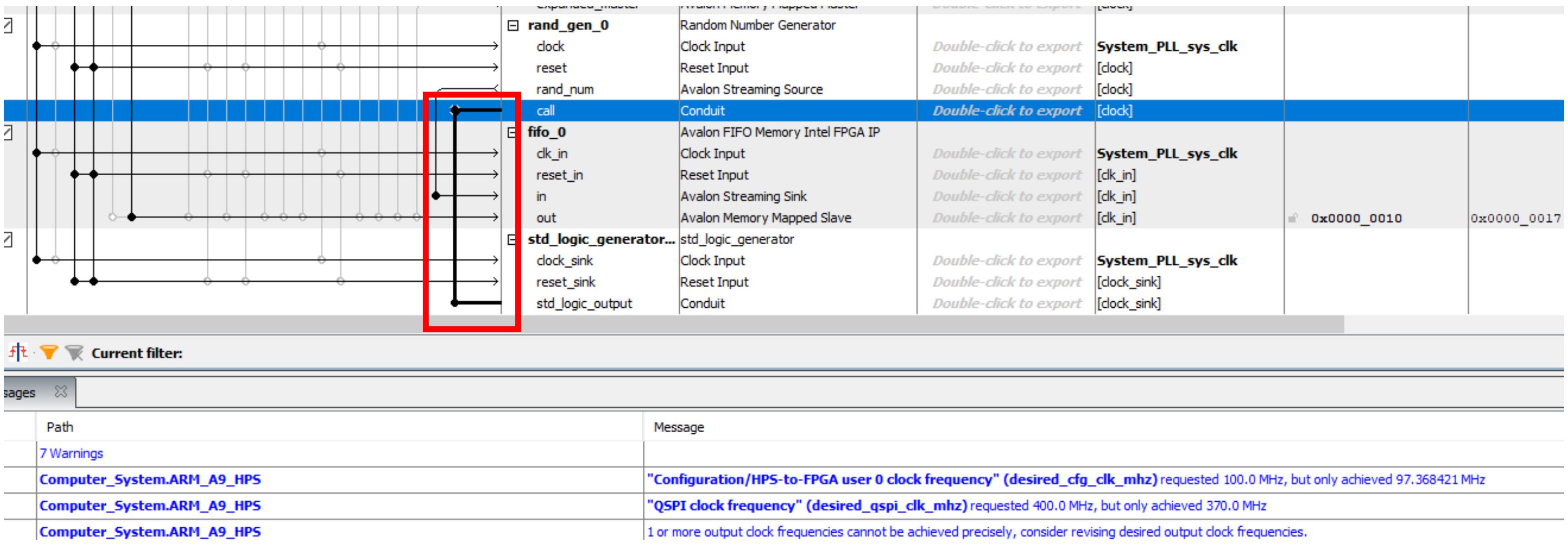
Platform Designer connections


System: Computer_System **Path:** rand_gen_0

Name	Description	Export	Clock	Base	End
Video_In_DMA_Addr...	DMA's Front and Back Buffer Address ...				
clock	Clock Input	Double-click to export	**System_PLL_sys_clk**		
reset	Reset Input	Double-click to export	[clock]		
slave	Avalon Memory Mapped Slave	Double-click to export	[clock]	0x0000_3060	0x0000_306f
master	Avalon Memory Mapped Master	Double-click to export	[clock]		
Video_In_Subsyst...	Video_In_Subsystem				
edge_detection_contr...	Avalon Memory Mapped Slave	Double-click to export	[sys_clk]	mixed	mixed
sys_clk	Clock Input	Double-click to export	**System_PLL_sys_clk**		
sys_reset	Reset Input	Double-click to export			
video_in	Conduit	**video_in**			
video_in_dma_control...	Avalon Memory Mapped Slave	Double-click to export	[sys_clk]	mixed	mixed
video_in_dma_master	Avalon Memory Mapped Master	Double-click to export	[sys_clk]		
F2H_Mem_Window_...	Address Span Extender				
clock	Clock Input	Double-click to export	**System_PLL_sys_clk**		
reset	Reset Input	Double-click to export	[clock]		
windowed_slave	Avalon Memory Mapped Slave	Double-click to export	[clock]	0xff60_0000	0xff7f_ffff
expanded_master	Avalon Memory Mapped Master	Double-click to export	[clock]		
F2H_Mem_Window_...	Address Span Extender				
clock	Clock Input	Double-click to export	**System_PLL_sys_clk**		
reset	Reset Input	Double-click to export	[clock]		
windowed_slave	Avalon Memory Mapped Slave	Double-click to export	[clock]	0xff80_0000	0xffff_ffff
expanded_master	Avalon Memory Mapped Master	Double-click to export	[clock]		
rand_gen_0	Random Number Generator				
clock	Clock Input	Double-click to export	**System_PLL_sys_clk**		
reset	Reset Input	Double-click to export	[clock]		
rand_num	Avalon Streaming Source	Double-click to export	[clock]		
call	Conduit	**rand_gen_0_call**	[clock]		
fifo_0	Avalon FIFO Memory Intel FPGA IP				
clk_in	Clock Input	Double-click to export	**System_PLL_sys_clk**		
reset_in	Reset Input	Double-click to export	[clk_in]		
in	Avalon Streaming Sink	Double-click to export	[clk_in]		
out	Avalon Memory Mapped Slave	Double-click to export	[clk_in]	**0x0000_0010**	0x0000_0017

What did not work – New IP Core to enable RNG

One of the things that I tried that did not work, was actually making my own IP core to enable the conduit signal in the random number generator, to prevent having to change the VHDL code manually each time it is re-generated in the Platform Designer. Most of my research in this project went into learning about Avalon-ST and Avalon-MM components, which is how I was able to design the Random Number Generator/ FIFO memory combination that I and others in our ENGC401 class used for random number generation. I did not spend as much time learning about designing conduit signals that interface well together. Below are screen shots of what I attempted to do with the new IP components, which was send a high signal to the enable conduit signal of the Random Number Generator.

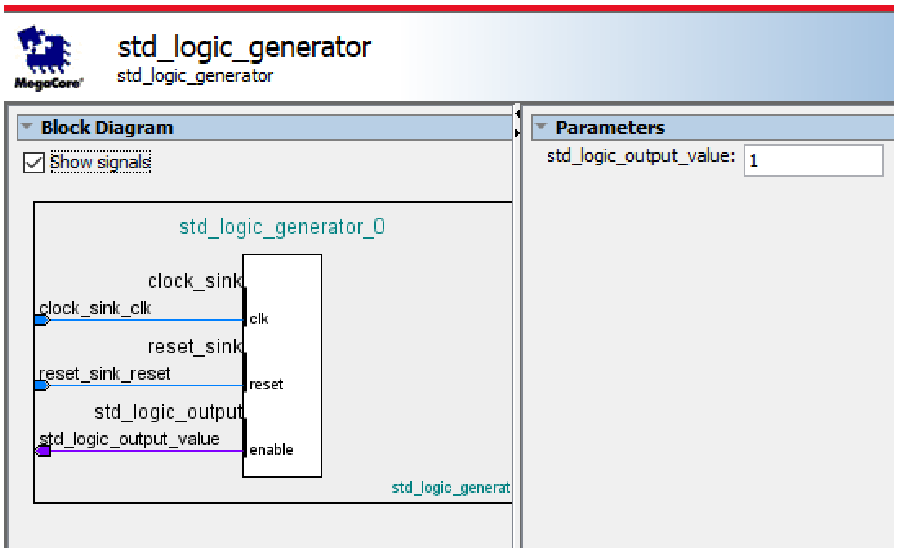


rand_gen_0	Random Number Generator				
clock	Clock Input	Double-click to export	System_PLL_sys_clk		
reset	Reset Input	Double-click to export	[clock]		
rand_num	Avalon Streaming Source	Double-click to export	[clock]		
call	Conduit	Double-click to export	[clock]		
fifo_0	Avalon FIFO Memory Intel FPGA IP				
clk_in	Clock Input	Double-click to export	System_PLL_sys_clk		
reset_in	Reset Input	Double-click to export	[clk_in]		
in	Avalon Streaming Sink	Double-click to export	[clk_in]		
out	Avalon Memory Mapped Slave	Double-click to export	[clk_in]	0x0000_0010	0x0000_0017
std_logic_generator...	std_logic_generator				
clock_sink	Clock Input	Double-click to export	System_PLL_sys_clk		
reset_sink	Reset Input	Double-click to export	[clock_sink]		
std_logic_output	Conduit	Double-click to export	[clock_sink]		

Current filter:

Path	Message
7 Warnings	
Computer_System.ARM_A9_HPS	"Configuration/HPS-to-FPGA user 0 clock frequency" (desired_cfg_clk_mhz) requested 100.0 MHz, but only achieved 97.368421 MHz
Computer_System.ARM_A9_HPS	"QSPI clock frequency" (desired_qspi_clk_mhz) requested 400.0 MHz, but only achieved 370.0 MHz
Computer_System.ARM_A9_HPS	1 or more output clock frequencies cannot be achieved precisely, consider revising desired output clock frequencies.

Attempting to interface between the two conduit signals directly within the Platform Designer

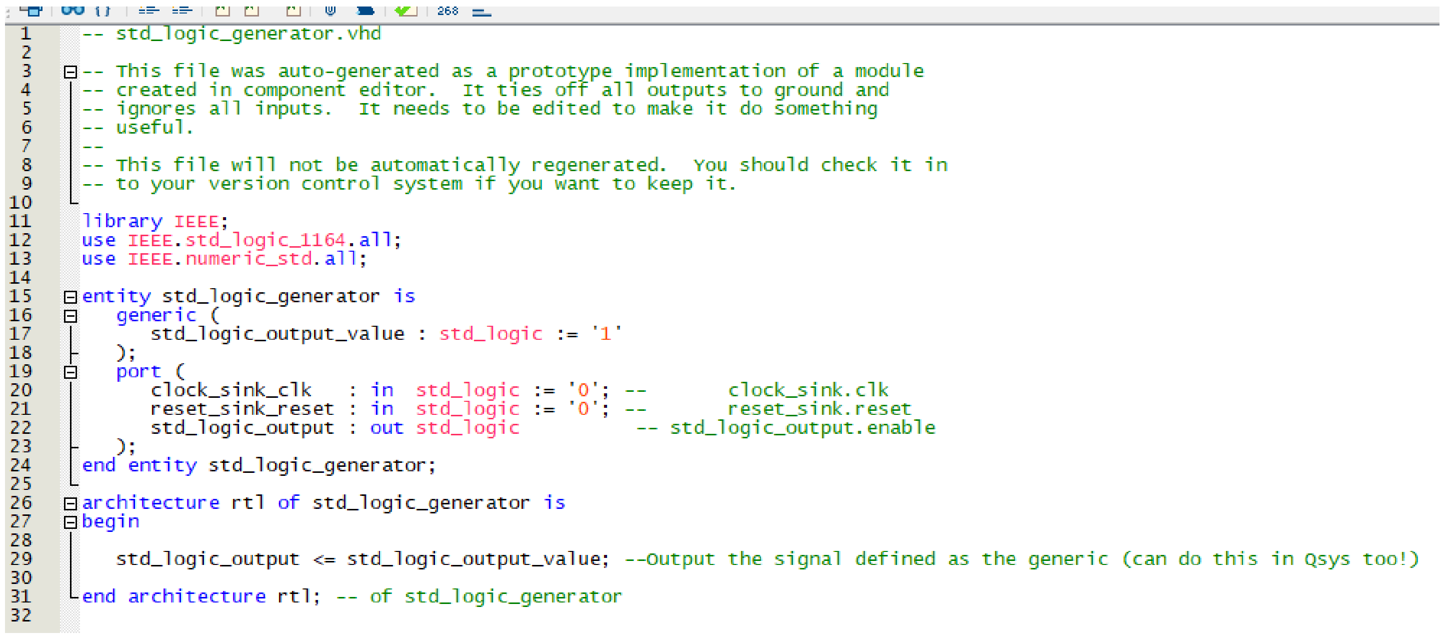


std_logic_generator
std_logic_generator

Block Diagram
☑ Show signals

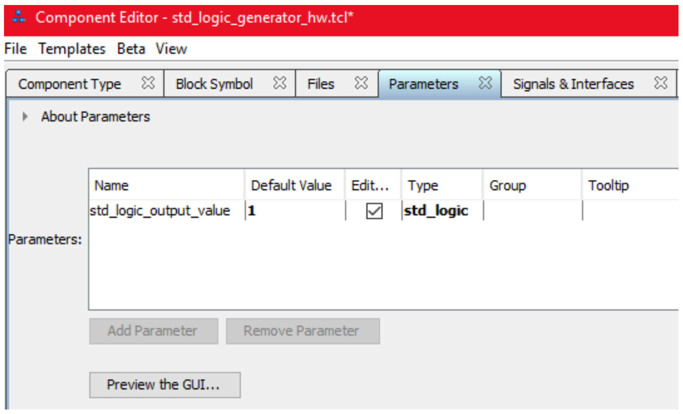
Parameters
std_logic_output_value: 1

The IP instantiation for the new component



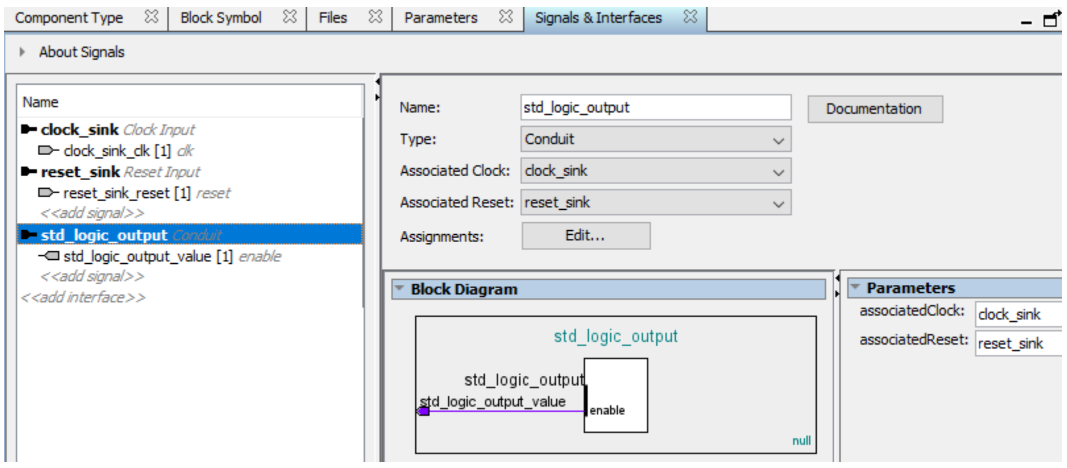
```vhdl
-- std_logic_generator.vhd

-- This file was auto-generated as a prototype implementation of a module
-- created in component editor.  It ties off all outputs to ground and
-- ignores all inputs.  It needs to be edited to make it do something
-- useful.
--
-- This file will not be automatically regenerated.  You should check it in
-- to your version control system if you want to keep it.

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity std_logic_generator is
	generic (
		std_logic_output_value : std_logic := '1'
	);
	port (
		clock_sink_clk    : in  std_logic := '0'; --       clock_sink.clk
		reset_sink_reset  : in  std_logic := '0'; --       reset_sink.reset
		std_logic_output  : out std_logic         -- std_logic_output.enable
	);
end entity std_logic_generator;

architecture rtl of std_logic_generator is
begin

	std_logic_output <= std_logic_output_value; --Output the signal defined as the generic (can do this in Qsys too!)

end architecture rtl; -- of std_logic_generator

```

New component VHDL code (Not included in project, as it was not used in the end result)



Name	Default Value	Edit...	Type	Group	Tooltip
std_logic_output_value	1	☑	std_logic		

Creating and editing of the new component. By including the VHDL file, the goal was to allow the ability to change the output of the STD_LOGIC output conduit signal.



Name
- clock_sink *Clock Input*
  - clock_sink_clk [1] *clk*
- reset_sink *Reset Input*
  - reset_sink_reset [1] *reset*
  - *<<add signal>>*
- std_logic_output *Conduit*
  - std_logic_output_value [1] *enable*
  - *<<add signal>>*
- *<<add interface>>*

Name: std_logic_output
Type: Conduit
Associated Clock: clock_sink
Associated Reset: reset_sink
Assignments: Edit...

associatedClock: clock_sink
associatedReset: reset_sink

In the given time, we were not able to get this component working to generate the enable conduit signal necessary to start the Random Number Generator, so this component was not included in the final design. In the end, this did not affect the overall end result of the project, as the main goal was to enable the Random Number Generator to output values to the FIFO memory. We were still able to achieve this end result though directly editing the VHDL code as discussed above; we simply were not able to achieve it through the creation of a new IP Core component.

Results of Design

The end result of this project is a fully functional and satisfying game based on the classic Asteroids. During game execution, there is no hesitation from the system, and performance is exceptional. The game runs at a full 60 frames per second at all times, which is both desired and expected from our display hardware that runs at 60Hz. The smooth game performance can be attributed to the use of the vertical sync (VSync) functionality of the pixel buffer controller, which synchronizes frame updates with the display hardware, and also to the efficient game logic by javidx9 that this project was based off of. This includes using the elapsed time between frame updates as a factor when performing kinematic equation calculations for object movement. I also believe I was also efficient in developing the game methods in the C language for the DE1-SoC.

While the game itself runs smoothly and without errors, there is some screen flicker when running the game. This is most likely due to the use of single, rather than double buffering. If double buffering was used, there would be a more instantaneous update to the pixel buffer, which reduces the chances of making changes to the buffer while it is being read by the display. With a large amount of operations taking place every frame, it is possible that there is some overlap of buffer reading and writing.

The new IP cores introduced in this project also function satisfactorily. The random number generator produces and streams numbers to the FIFO memory quickly and with a uniform distribution. Concurrently, the FIFO memory mapped register effortlessly manages the fast incoming data stream and produces outputs in the data register without introducing any additional lag or delay. This system is quick enough to handle our need for multiple random numbers per frame.

The random numbers are used for 1. Asteroid Colors, 2. Bullet Colors, 3. Asteroid Velocity, 4. Asteroid Direction, 5. Asteroid WireFrame Graphics. The position, velocity, direction and graphic randomization only happen occasionally, but asteroid and bullet color randomization happens every frame for each instance of the objects. This shows off the ability to stream large amounts of random numbers into the system without slowing down the system performance. In the software implementation, we would come to a point sooner where the need for this many random numbers would slow down the system. But because this random number generation system is hardware accelerated, we have access to a consistent stream of random numbers that we can access at will without slowing down the system. Colors of every asteroid and every bullet on screen change randomly per frame, based on the numbers received from the IP cores, demonstrating high throughput of the random number stream.

Despite the slight flickering issue discussed earlier, the overall system is accurate, functional, and satisfactorily meets the requirements we set out to achieve. The game is fully featured, meaning the player can win (by beating the high score), lose (by hitting an asteroid), is met with a challenge, and is able to have a different play experience each time.

Conclusion

As a whole, this implementation of Asteroids is quite satisfactory and exceeds the expectations we originally had. Initially, our goal was to create a playable version of the game that involved randomization of the asteroid direction and graphic shape. We met both of those expectations, and also was able to incorporate randomization of colors as well. We initially thought latency between the LW AXI bridge would cause a problem with accessing random numbers quickly in the game, but because the RNG and FIFO memory had the ability to output numbers at an exceedingly high throughput, we were able to utilize these components more fully and add extra “randomness” to the game to add more life to the game.

If this project were to be repeated with the knowledge we currently hold, the most significant change to the system would do with the screen buffering functionality. As previously mentioned, there is slight flickering during the game. It does not affect the playability of the game, and does not relate to framerate, but it is still noticeable. Incorporating a double buffering solution would most likely be preferable to reduce this issue. Although a considerable amount of time was spend during this project to try to add double buffering functionality, we could not get past a certain bug that only successfully buffered the top half of the screen. The bottom half of the screen would always output the contents of both screen buffers, which ruins the effect that double buffering has on reducing flickering and screen tearing issues. We think this has to do with memory allocation; if the same amount of memory for the 320x240 resolution system was used to implement a double buffer at 640x480, then there may only be half the memory necessary for implementing the screen buffers. We did not find the final solution to this issue, as double buffering was not demonstrated in the project, but this is work that could be done in the future for those looking to develop games on the De1-SoC in 640x480.

This project would not have been successful if it were not for Altera, javidx9 on YouTube, Cornell University, and eddmann.com. These sources provided hardware and software aids that were influential in the successful functionality of this project. First of all, YouTube channel javidx9 provided the high-level concepts for the game logic used in this project. His video on the Asteroids game helped me properly reason my way though the game logic. Next, the vector implementation was taken from an article from eddmann.com. This article provided the code and a nice explanation for how vector functionality could be implemented in the C language. The Random Number Generator and FIFO memory IP cores were created by Altera, and the IP Cores themselves come bundled with Quartus 18.0. Finally, some of the graphics primitives/ drawing routines for different shapes used were from Cornell University. Documentation for all of these references can be found in the Appendix.

Appendix

The following program files included in the project file are as follows:

- asteroids.c – This is my main project source code. It contains all of the game logic and runs on a modified version of the DE1-SoC computer system detailed below. Compile with "`gcc asteroids.c -o asteroids -O3 -lm`"
- vector.h – This is the vector implementation in C that I utilized in the asteroids.c file. It is a dependency of asteroids.c, and simply needs to be in the same folder when compiling the game.
- address_map_arm_br14.h – This is the address map header that also needs to be in the same folder as the asteroids.c file upon game compilation. It simply contains the definitions for physical addresses on the DE1-SoC that we use to map to virtual addresses to be used in the software.

The final project folder also contains the folder for the modified computer system that is needed to run the software on the DE1-SoC. Without this system, the random number generation components will not be present, and the game will not work. The system is a modified version of the one used in Lab 15. The system folder is called "DE1_SoC_Computer_chopped_1", or the included .qar file can be used, which is called "DE1_SoC_Computer.qar".

- There is also a shortcut to the YouTube video demonstration of the project. (Link: https://youtu.be/QTQDB-6qthQ)
  - The video demonstrates how the game is played, and how randomness is achieved in 1. Asteroid Colors, 2. Bullet Colors, 3. Asteroid Velocity, 4. Asteroid Direction, 5. Asteroid WireFrame Graphics

Links to code/ designs borrowed from others:

- Graphics Primitives (From Cornell University): http://people.ece.cornell.edu/land/courses/ece5760/DE1_SOC/HPS_peripherials/univ_pgm_computer.index.html
- High Level Code Design (From javidx9): https://www.youtube.com/watch?v=QgDR8LrRZhk

Links to Datasheets for Altera IP Cores (also included as PDF's):

- Random Number Generator IP Core User Guide: https://www.intel.com/content/www/us/en/programmable/documentation/dmi1455632999173.html#dmi1455633326157
- Avalon On-Chip FIFO Memory Core User Guide: https://www.intel.com/content/www/us/en/programmable/documentation/sfo1400787952932.html#iga1401396003807