N-BODY GRAVITY SIMULATOR

Project Report

EC806 Digital Design using FPGAs

by

Omkar Kamath (201EC140)

Aniket Uppin (201EC169)

Bhimreddy B Y (201EC170)

Under the guidance of

Dr Sumam David

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING
NATIONAL INSTITUTE OF TECHNOLOGY KARNATAKA
SURATHKAL, MANGALORE - 575025

Table of Contents:

Introduction :

In the vast expanse of the cosmos, the force of gravity orchestrates the cosmic dance of celestial bodies, shaping the very fabric of the universe. Understanding and simulating the intricate interplay of gravitational forces has been a longstanding endeavour for scientists and researchers alike. The Gravity Simulators Project emerges as a groundbreaking initiative poised at the nexus of scientific exploration and technological innovation.

Simulating gravity between celestial bodies (stars, planets, asteroids, etc.) using a standard CPU is limited since it can only do sequential calculations, leading to very long simulation times as the number of bodies increases. As the acceleration computations are independent of each other, here we can see the possibility of using FPGAs as accelerator for computing accelerations. This project tries to parallelize the acceleration computations and achieve faster and accurate results which can be observed on a VGA monitor.

Aim and Objectives :

- To design the 2D N body gravity simulator and observe the interaction between N bodies on a VGA monitor.
- To use Cyclone V FPGA as accelerator for computing accelerations (Unleashing the Power of Parallel Processing).
- To design mouse based User Interface(UI) running on HPS (ARM Cortex A9) for increasing the interactiveness of the simulator.
- To observe the differences in the various performance metrics such as frame time with and without use of FPGA.

Newton's Law of Gravitation:

According to Newtonian gravitational theory force on one body because of other body is given by

$$|F_2| = |F_1| = \frac{Gm_1m_2}{r^2}$$

And acceleration on body1 because of body2 is given as

$$a_1 = \frac{F_1}{m_1}$$

$$a_1 = \frac{\frac{Gm_1m_2}{r^2}}{m_1} = \frac{Gm_2}{r^2}$$

Same with the acceleration on body2 because of body1

$$a_2 = \frac{F_1}{m_2} = \frac{Gm_1}{r^2}$$

As particles tends the come closer the r→0, which means the computed acceleration tends to be infinity theoretically, to avoid this condition, additional epsilon term is added as shown below

$$\frac{1}{r^2} \approx \frac{r}{\left(\sqrt{r^2+\epsilon^2}\right)^3}$$

Where ϵ^2 = 7.45058059692e-09

The value of the epsilon square is chosen based on the 9 digit precision of the 27 bit floating point number.

Now the accelerations will be

$$a_1 = \frac{Grm_2}{\left(\sqrt{r^2+\epsilon^2}\right)^3}$$

$$a_2 = \frac{Grm_1}{\left(\sqrt{r^2+\epsilon^2}\right)^3}$$

To compute the acceleration in vector form we also need to have the unit vector in both x and y directions.

$$\hat{x} = \frac{(x_1 - x_2)}{r} \quad \hat{y} = \frac{(y_1 - y_2)}{r}$$

Now

$$a_{1x} = \widehat{x}\frac{Grm_2}{\left(\sqrt{r^2+\epsilon^2}\right)^3} \quad a_{1y} = \widehat{y}\frac{Grm_2}{\left(\sqrt{r^2+\epsilon^2}\right)^3}$$

Since the displacement will be negative for body we have to include the negative sign

$$a_{2x} = -\widehat{x}\frac{Grm_1}{\left(\sqrt{r^2+\epsilon^2}\right)^3} \quad a_{2y} = -\widehat{y}\frac{Grm_1}{\left(\sqrt{r^2+\epsilon^2}\right)^3}$$

Since we have no way to perform a floating point divide quickly we are limited to 3 different operations in our calculation. Add/Subtract, Multiply and Inverse Square root (using the fast inverse square root algorithm). The following operations are used to calculate the final result in our accelerator. We first calculate the square of the radius using 2 adders to calculate the x and y displacement. Then we multiply the outputs with themselves and take their sum which is the radius squared.Finally accelerator on FPGA works on below equation based computations.

$$r^2 = \left(x_1 - x_x\right)^2 + \left(y_1 - y_2\right)^2$$

$$r = \frac{1}{\sqrt{r^2}}r^2 = \frac{1}{r}r^2$$

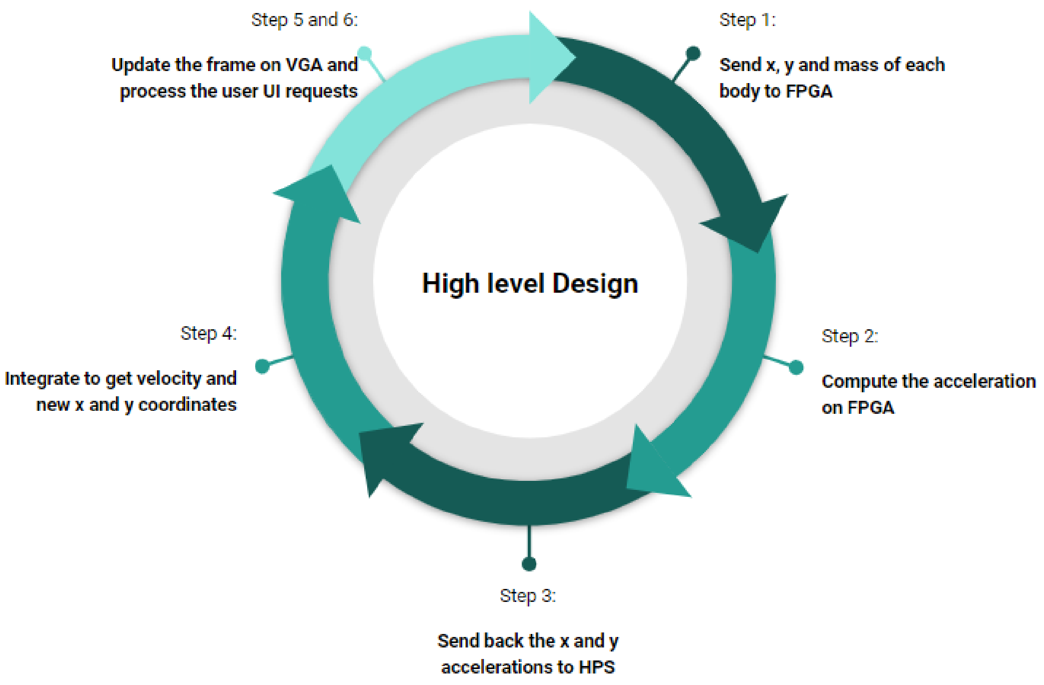
$$a_{1x} = \frac{1}{\sqrt{r^2}}\left(x_1 - x_2\right) \times Grm_2\left(\frac{1}{\sqrt{r^2+\epsilon^2}}\right)^3$$

Floating Point Representation:

- ➢ Floating point is a type of scientific notation that is used to represent non-integer numbers in computers. Normal IEEE single precision floating point has 3 parts.
    - 1 bit that represents the sign, with 0 being positive and 1 being negative.
    - 8 bits that are used to represent the exponent.
    - 23 bits that represent the mantissa or the precision of the location.
- ➢ The 27 bit floating point standard has
    - 1 sign bit.
    - 8 exponent bits.
    - 18 mantissa bits.

- ➢ This is the same as 32 bit but the mantissa is truncated by 5 bits. This is because we want to minimise the number of DSPs needed to create a floating point multiplier.

High level Design:


High level Design
Step 1:
Send x, y and mass of each body to FPGA
Step 2:
Compute the acceleration on FPGA
Step 3:
Send back the x and y accelerations to HPS
Step 4:
Integrate to get velocity and new x and y coordinates
Step 5 and 6:
Update the frame on VGA and process the user UI requests

I. Send the x, y positions and mass of each celestial body to FPGA.
II. Compute the accelerations using FPGA.
III. Send back the FPGA computed x and y accelerations to HPS.
IV. Use Euler's integration to get velocity and updated x and y positions.
V. Update the frame on the VGA screen.
VI. Process the user UI request and make corresponding changes on the VGA screen.
VII. while(1)

```
{
        Repeat steps 1 to 6
}
```

Hardware Design:

1) Floating point Adder:

A. The 2-stage pipelined adder/subtractor module performs the required operations on the before mentioned 27-bit floating point format numbers.
B. For both the operations, it normalises the smaller number i.e gets it to the same exponent as the greater number and appropriately shifts respective mantissas.
C. For addition operation, it performs addition of the new mantissas with 1 appended to the MSB side.
D. If the summation generates a final carry then we again normalise it by increasing the exponent and right shifting the obtained mantissa.
E. For subtraction operation, it performs the subtraction of the new mantissas with 1 appended to the MSB side and we check the resultant for any need of normalisation.
F. This module has been extensively tested using automated scripts and reference design written in python.

2) Floating point Multiplier:

A. Extract the sign,exponent and mantissa bits from the input 27 bit numbers.
B. Add the exponents of the two numbers and calculate the sign of the resultant.
C. Multiply the mantissa of the two numbers and truncate the product to 18 bits.
D. Normalise the product and re-adjust the exponent.
E. Check for the underflow condition

The Underflow condition provided is very crucial.IWhen underflow isn't checked,the module fails to handle the de-normal numbers. This leads to incorrect results.

3) Fast Inverse square root calculator:

A. Initial Guess:
    a. Start with the floating-point number you want to find the inverse square root for.
    b. Convert the floating-point number to a 32-bit integer.
B. Initial Estimate:
    a. Set the initial estimate for the inverse square root. A common choice is 0x5f3759df.
    b. This value is chosen to give a good initial guess and accelerate the convergence of the algorithm.

C. First Iteration:
   a. Perform a bitwise shift right by 1 on the initial estimate.
   b. Take the bitwise OR with 0x5f3759df and subtract it from the initial estimate, effectively dividing it by 2.

D. C code

```
float fastInverseSqrt(float x) {
    int i;
    float y = x;
    float threehalfs = 1.5f;

    // Initial guess (optional: perform more iterations for refinement)
    i = *(int*)&y;
    i = 0x5f3759df - (i >> 1);
    y = *(float*)&i;

    // Newton's method iteration (optional: perform more iterations for refinement)
    y = y * (threehalfs - (x * 0.5f * y * y));

    return y;
}
```

Fast Inverse square root is a 5 stage pipelined implementation which computes the inverse square root of any number.

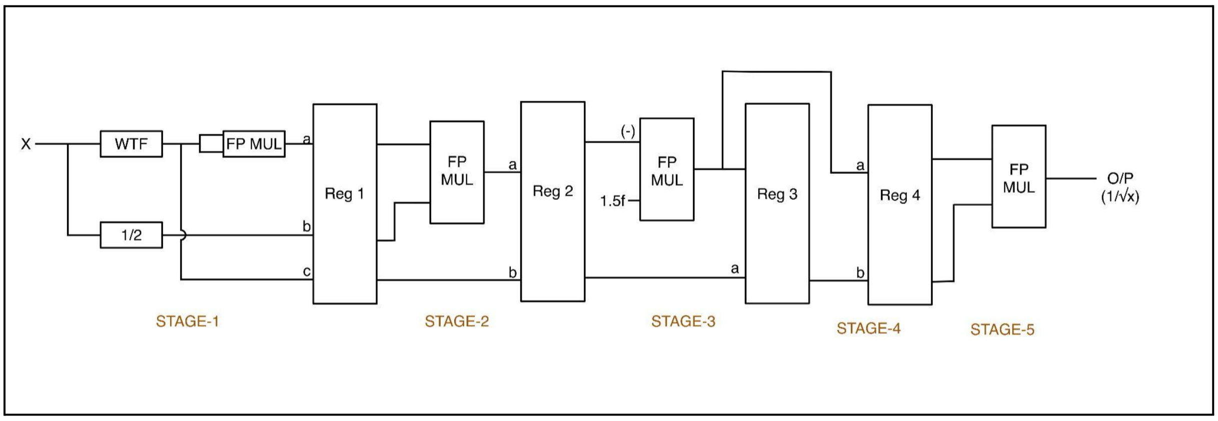


Fig 1: 5-stage pipelined structure of fast inverse square root

Accelerator Module:

- The accelerator module computes the acceleration on body1 because of body2.
- 18 stage pipelined architecture consisting of floating point adder, multiplier and inverse square root modules.

- It tries to compute acceleration on body1 as per equation below

$$a_{1x} = \frac{1}{\sqrt{r^2}}\left(x_1 - x_2\right) \times Grm_2\left(\frac{1}{\sqrt{r^2+\epsilon^2}}\right)^3$$

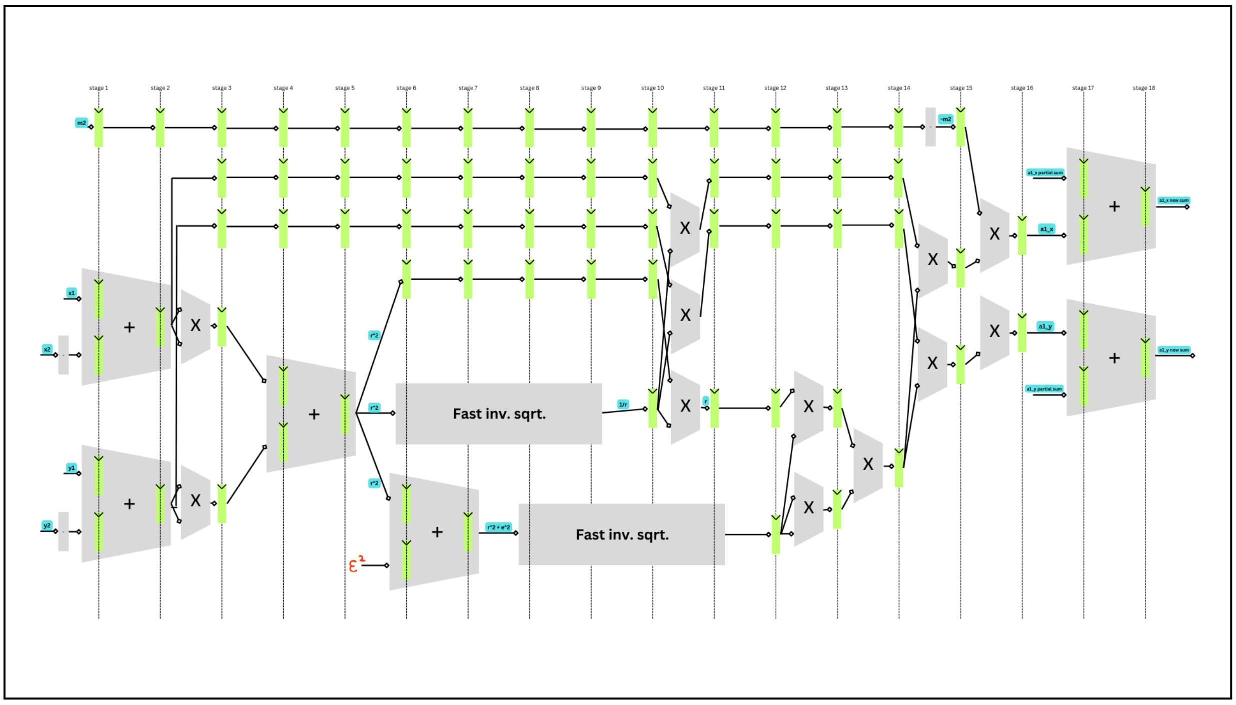


Fig 2: 18-stage pipelined structure of accelerator.

Neighbourhood Algorithm:

Neighbourhood algorithm is the way of providing celestial bodies to the accelerator to compute the accelerations(both x and y).

It consists of following parts:

1. Visitor centre m10k block memories.
2. Visitor centre controller.
3. N neighbourhoods each containing m10k memories and accelerator.
4. Neighbourhood controller.

1. Visitor centre m10k block memories:
    a. 3 m10k blocks each of depth 4096 words (1 word = 4 bytes) is used.
    b. First and second m10k for storing the x and y coordinates of the visitor body respectively.
    c. Third m10k for storing mass of the visitor body.

2. Visitor centre controller:

   a. It has two interfaces one to the HPS and other to the FPGA.
   b. The visitor centre controller sends the visitor body one by one to the N neighbourhoods (x, y and mass) present on FPGA.
   c. It also accepts the new x and y coordinates and mass of the body from the HPS.

3. Neighbourhoods m10k block memories:
   a. 4 m10k blocks each of depth 4096 words (1 word = 4 bytes) is used.
   b. First and second m10k for storing the x and y coordinates of the neighbourhood body respectively.
   c. Third and fourth m10k for storing partial accelerations of the neighbourhood body respectively.
   d. Each neighbourhood contains the accelerator unit which accelerates the acceleration computation.

4. Neighbourhood controller:
   a. It interfaces with the visitor centre controller (interfacing with FPGA).
   b. It generates the necessary control signals such as write address, read address, write enable signals for the neighbourhood m10k blocks.

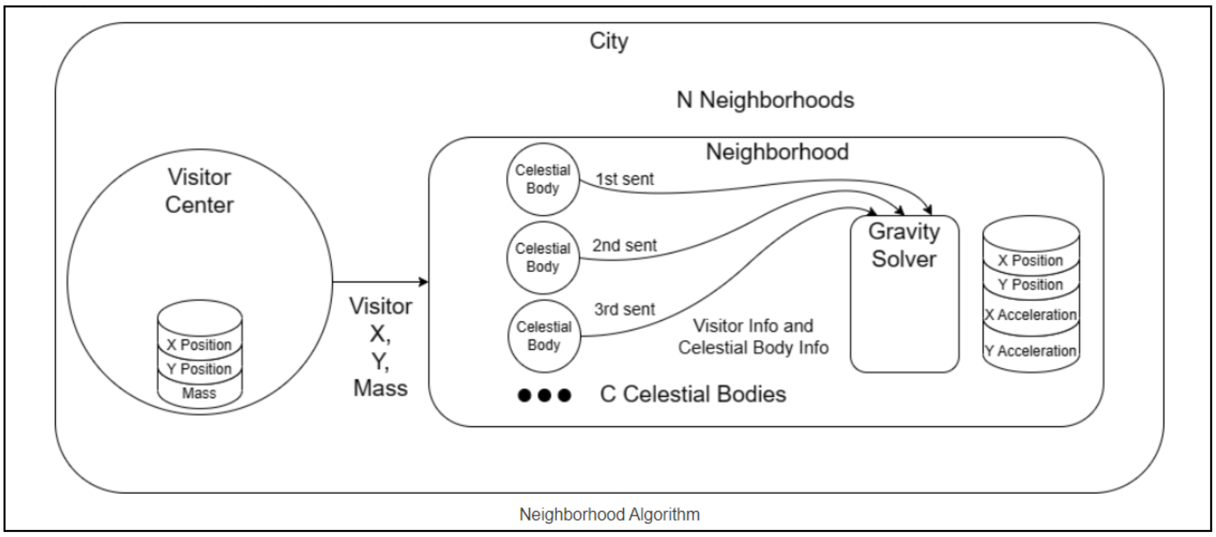


Fig 3: Block diagram for neighbourhood algorithm.

The algorithm has the x and y position and mass sent from the HPS to the FPGA and then sends back the x and y acceleration for each celestial body. We separate each solver into its own “neighbourhood” that contains “houses” that are celestial bodies. Each neighbourhood has m10k blocks that hold the data for the x and y position and x and y acceleration for each celestial body. The visitor centre holds a copy of the x and y positions and the mass of each celestial body.

The neighbourhoods separate the celestial bodies into groups as seen in the diagram below where C is the number of celestial bodies and N is the number of neighbourhoods based on how many gravity solvers can be created based on the available hardware. The C celestial bodies are then evenly split among each neighbourhood, with any spots not filled having a “fake” celestial body that is present in the M10k block but never read or written to by the HPS.

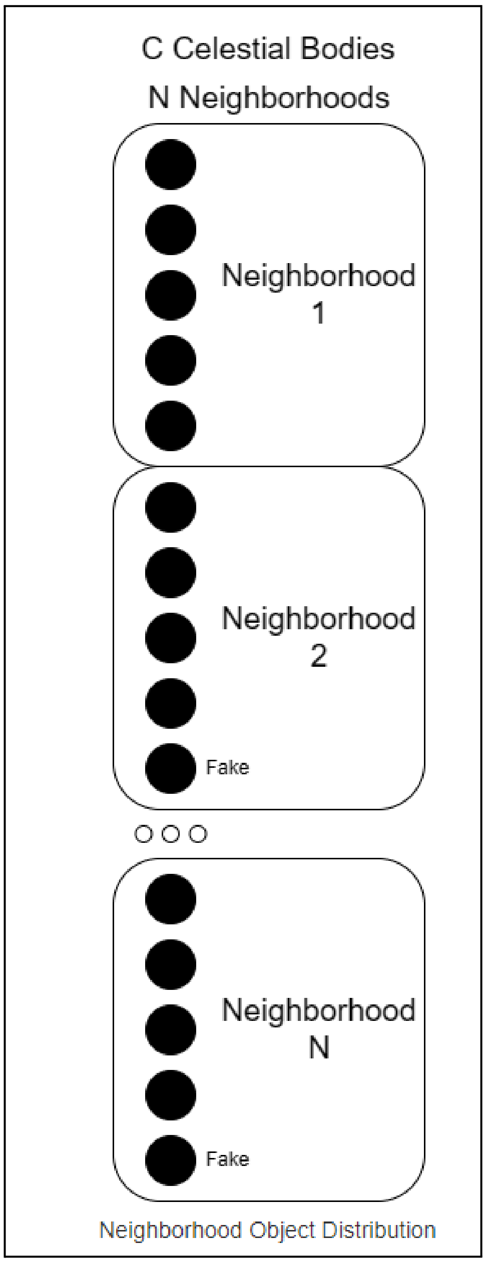


Fig 4: Celestial bodies distribution in neighbourhoods

With the structure set up and placed in a city with N neighbourhoods, to start a calculation cycle for a celestial body, the visitor centre sends over that celestial body as a “visitor” to all the neighbourhoods at the same time. The neighbourhoods then calculate the acceleration from gravity between the visitor and each celestial body, “house,” in the neighbourhood and adds the new acceleration to the sum of accelerations for that house. This is key since this allows for parallelization for calculating the interaction between one visitor and all houses - or 1 celestial body and all other celestial bodies. If the visitor is the same object as the celestial object being calculated against in the neighbourhood, the mass sent to the gravity solver for that calculation is sent as zero to avoid having infinite acceleration from one over zero distance. Once all the celestial bodies in each neighbourhood were compared to the visitor, we sent a next_visitor signal to the “visitor centre” to generate a new visitor.

The state machine for the accelerator is shared between the visitor centre and the neighbourhoods. The state machine mirrors the HPS’ behaviour with its 5 states. The base state is reset where most of the addresses/indices and state variables are set to 0. Most of the time is spent in the wait state where the FPGA is waiting on the HPS for the trigger to move to the next state. Once the fill trigger is sent, write enable being high, the FPGA yields control of the M10k memory addressing and data to the HPS so that it can send position and mass data to the FPGA and fill the visitor and neighbour M10K blocks. After we exit the fill state, the FPGA enters the calc state where we do all the comparisons between the N-bodies and calculate the total acceleration for each one. When we are done, we move back to the wait state and send the done signal to the FPGA and wait for the trigger to move to the send state. Once the send signal is sent from the HPS, we let the HPS control the acceleration memory address and send back the acceleration values for each celestial body.

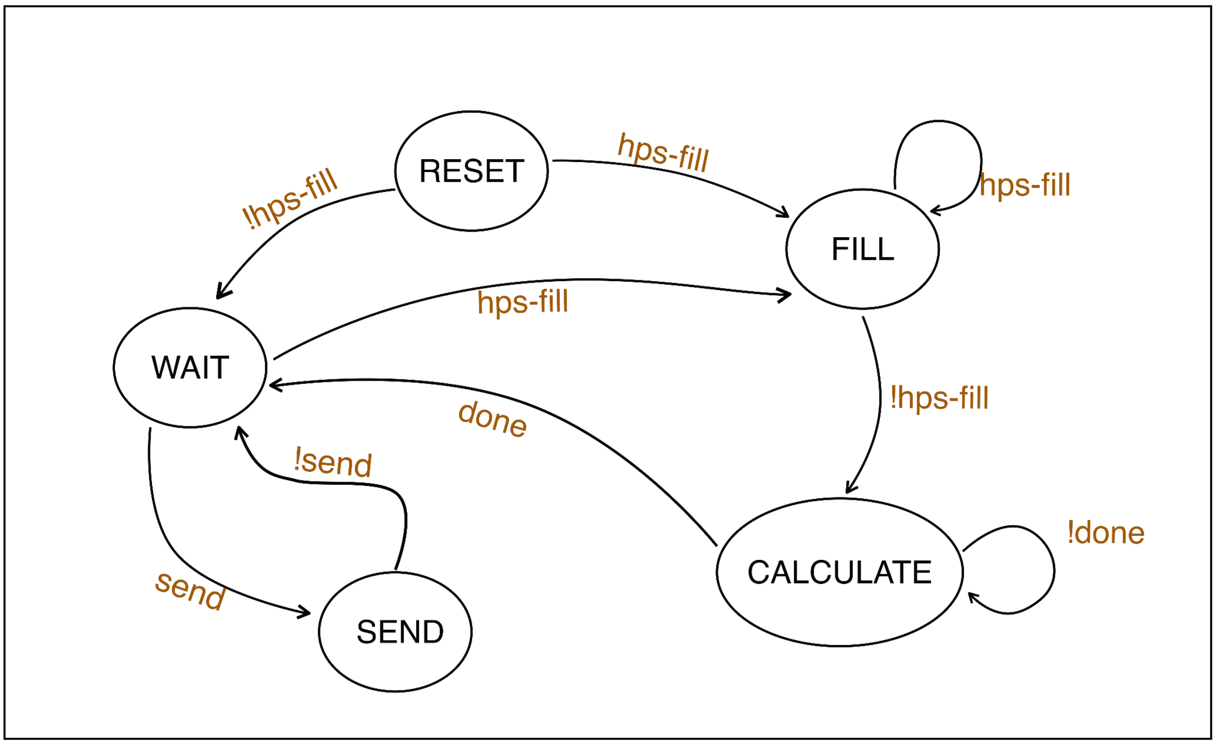
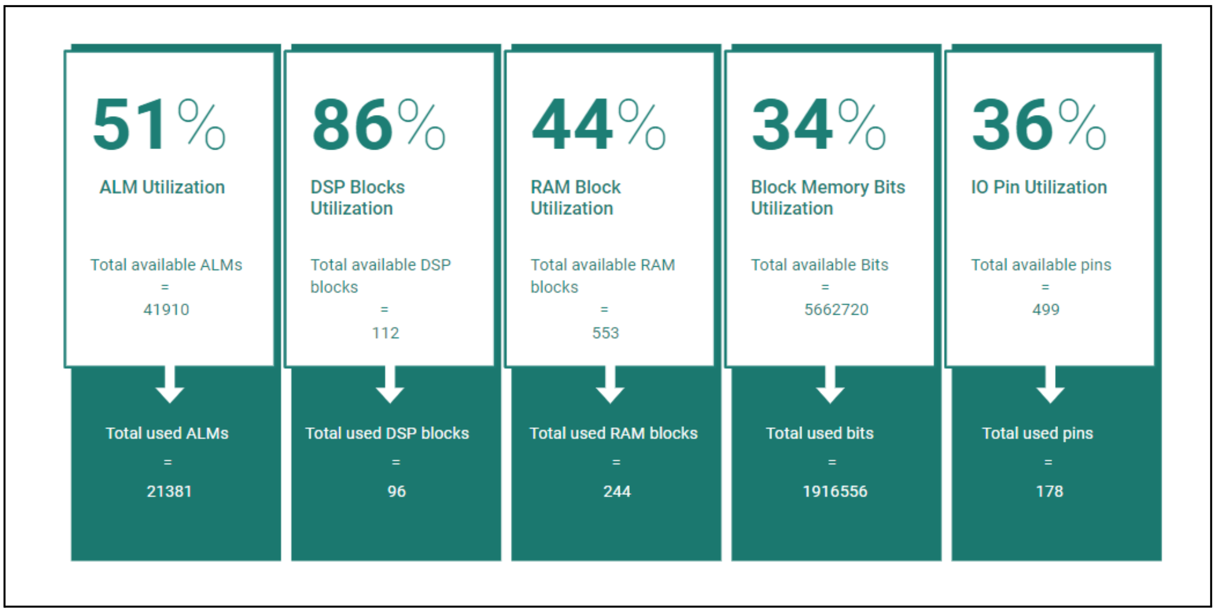


Fig 5: State diagram for HPS and FPGA behaviour.

Resource utilisation of FPGA:



51%
ALM Utilization
Total available ALMs = 41910
Total used ALMs = 21381

86%
DSP Blocks Utilization
Total available DSP blocks = 112
Total used DSP blocks = 96

44%
RAM Block Utilization
Total available RAM blocks = 553
Total used RAM blocks = 244

34%
Block Memory Bits Utilization
Total available Bits = 5662720
Total used bits = 1916556

36%
IO Pin Utilization
Total available pins = 499
Total used pins = 178

Others:

- ❖ PLL utilisation = 3/15 (20%)
- ❖ Total registers = 20800 (for FPGA configuration)

ALM / Adder	DSP blocks / Adder
393	0

ALM / Multiplier	DSP blocks / Multiplier
30	1

Total ALMs	Total DSP	Number of accelerators	ALM/design	DSP/design	# add/design	# mul/design	Total Throughput
21381	96	5	3684	18	8	18	1*5 = 5

Comments On Resource Utilisation:

- ❖ Since each accelerator is using 18 multipliers and each multiplier is inferring one DSP block on the FPGA, therefore each accelerator is using 18 DSP blocks configured to multiply two 27 bit numbers.

- ❖ The maximum number of accelerators which can be synthesised on the board are dependent on the number of DSP blocks (the limiting factor).

- ❖ Cyclone V FPGA (C6 grade) has 112 DSP blocks, hence we can run 5 accelerators at a time on FPGA. In that case number of DSP blocks inferred will be 18*5 = 90. But in the utilisation we are seeing 96 DSP blocks as utilisation.

- ❖ This is because we are doing 6 division operations for converting addresses from global to local domain (similar to address translation in cache memory organisation).

- ❖ We are using 3 PLLs and 1 DLL.
  - ➢ One PLL for 50 Mhz to 100 Mhz (clock for FPGA).
  - ➢ Second one for VGA, 100 Mhz to 25 Mhz. (640*480 resolution standard).
  - ➢ Third one to remove the skews in clock output ports ie, 100Mhz to 100 Mhz.

- ❖ Out of 553 M10k blocks we are using 244 blocks.

- ❖ ALM utilisation is 51% (significant utilisation), each adder and multiplier was using 393, 30 ALM blocks respectively.

- ❖ Total number of register utilisation is 20800 (each ALM contains 4 registers).

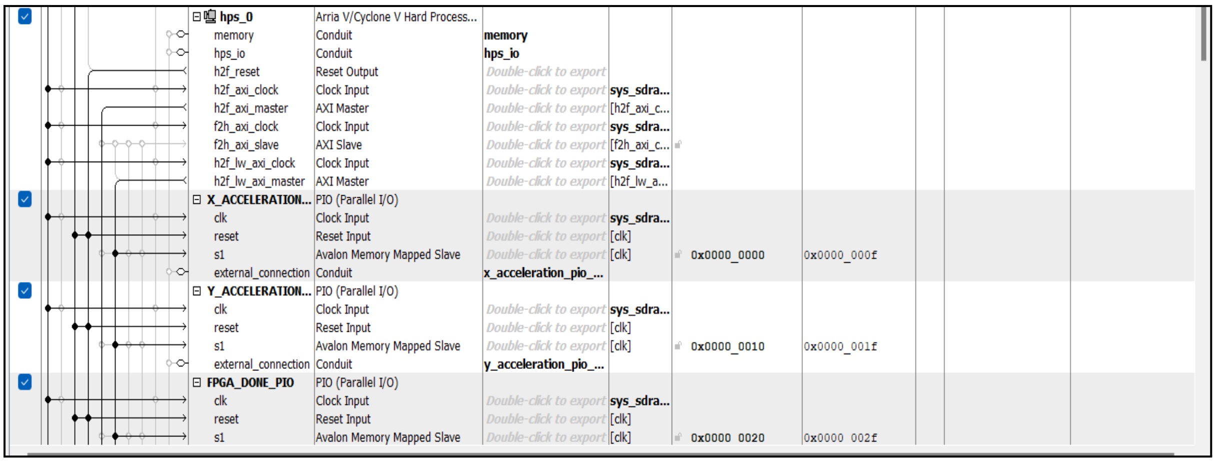
- ❖ The number of adders per accelerator is 8.

- ❖ Since in each clock cycle we are getting 1 acceleration (pipelined architecture) and we have 5 accelerators running parallely, so the throughput we get is 5 or IPC=5.

- ❖ IO pin utilisation is 178 out of 288 FPGA user IO pins.

HPS ⇔ FPGA Communication:

The communication between the FPGA and HPS (Hard Processor System) on the Cyclone V SoC is typically achieved through various interfaces, one of which is the Parallel I/O (PIO) port and the Advanced eXtensible Interface (AXI) bus. Parallel ports (PIO ports) instantiated in Qsys are defined as output if they communicate data from the HPS to the FPGA, and as input if, then communicate data from the FPGA to the HPS. PIO ports can be instantiated on the light-weight AXI bus, or on the full AXI bus. Looping the PIO write-reads in C to perform a few thousand PIO write/reads as fast as possible suggests that the AXI bus can do about 3.1 million write-read loopbacks/second and the light-weight bus can do about half that. A simple increment loop is at least 25 times faster, so most of the time in each loop iteration is spent waiting for the bus transactions.



⊟ **hps_0**	Arria V/Cyclone V Hard Process...				
memory	Conduit	**memory**			
hps_io	Conduit	**hps_io**			
h2f_reset	Reset Output	*Double-click to export*			
h2f_axi_clock	Clock Input	*Double-click to export*	**sys_sdra...**		
h2f_axi_master	AXI Master	*Double-click to export*	[h2f_axi_c...		
f2h_axi_clock	Clock Input	*Double-click to export*	**sys_sdra...**		
f2h_axi_slave	AXI Slave	*Double-click to export*	[f2h_axi_c...		
h2f_lw_axi_clock	Clock Input	*Double-click to export*	**sys_sdra...**		
h2f_lw_axi_master	AXI Master	*Double-click to export*	[h2f_lw_a...		
⊟ **X_ACCELERATION...**	PIO (Parallel I/O)				
clk	Clock Input	*Double-click to export*	**sys_sdra...**		
reset	Reset Input	*Double-click to export*	[clk]		
s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0000_0000	0x0000_000f
external_connection	Conduit	**x_acceleration_pio_...**			
⊟ **Y_ACCELERATION...**	PIO (Parallel I/O)				
clk	Clock Input	*Double-click to export*	**sys_sdra...**		
reset	Reset Input	*Double-click to export*	[clk]		
s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0000_0010	0x0000_001f
external_connection	Conduit	**y_acceleration_pio_...**			
⊟ **FPGA_DONE_PIO**	PIO (Parallel I/O)				
clk	Clock Input	*Double-click to export*	**sys_sdra...**		
reset	Reset Input	*Double-click to export*	[clk]		
s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0000_0020	0x0000_002f

Fig 6: Qsys file showing PIO port connections

The parallel input/output (PIO) core with Avalon interface provides a memory-mapped interface between an Avalon memory-mapped slave port and general purpose I/O ports. The I/O ports connect either to on-chip user logic, or to I/O pins that connect to devices external to the FPGA. Each PIO core can provide up to 32 I/O ports. An intelligent host such as a microprocessor controls the PIO ports by reading and writing the register-mapped Avalon-MM interface. Under control of the host, the PIO core captures data on its inputs and drives data to its outputs. When the PIO ports are connected directly to I/O pins, the host can tristate the pins by writing control registers in the PIO core.

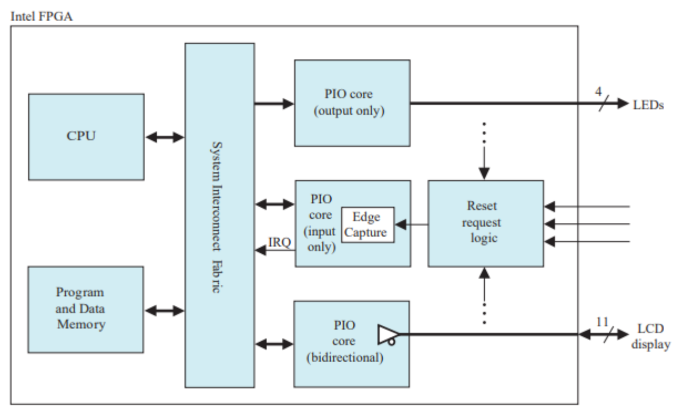
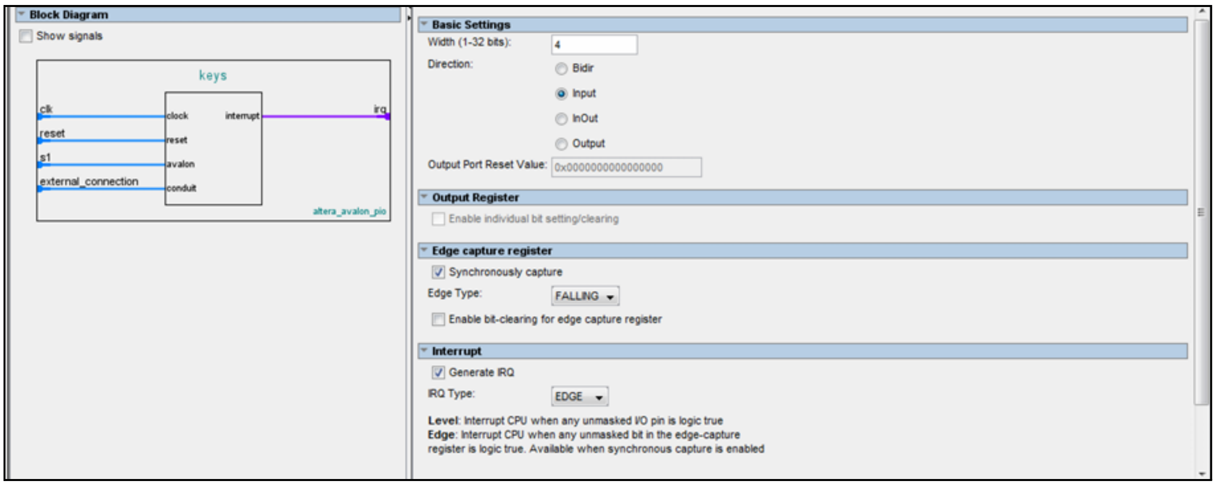


Fig 7: System Using Multiple PIO Cores

Polling is a technique to monitor an I/O port and to trigger an appropriate action when a change is detected. The general idea is that the processor periodically reads the I/O port and determines if there is a change. If there is a change, the processor will execute the subset of code to deal specifically with this change. Otherwise, the processor will continue with normal operations. The change is the edge transition of the external signals which can be configured in Qsys.For example, if we want to detect the falling edge of the pushbuttons (KEYs) in the HPS, we can configure the PIO core as shown in Fig. 8. Note that we can also configure the PIO core so that it will generate an interrupt request when the change occurs.



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

Fig 8: PIO Core configuration.

From FPGA to HPS:

1. X_ACCELERATION_PIO: Receives the computed X acceleration by FPGA and sends it to HPS.
2. Y_ACCELERATION_PIO: Receives the computed Y acceleration by FPGA and sends it to HPS.
3. FPGA_DONE_PIO: Sends the done signal from FPGA to HPS.

From HPS to FPGA:

1. X_PIO: X position from HPS to FPGA.
2. Y_PIO: Y position from HPS to FPGA.
3. RESET_PIO: active low reset from HPS to FPGA.
4. OBJECT_COUNT_PIO: number of objects from HPS to FPGA.
5. INDEX_OUT_PIO: global address sent from HPS to FPGA.

6. MASS_PIO: mass of the object which is being sent from HPS to FPGA.
7. WRITE_ENABLE_PIO: acts as hps_fill signal (refer to the state diagram).
8. SEND_ENABLE_PIO: acts as a send signal (refer to the state diagram).

In our design we used AXI light weight 32 bit bus (HPS master and FPGA slave) for exchanging information between FPGA and HPS. AXI bus for writing the frame information in pixel buffer and character buffer. We used SDRAM as the pixel buffer instead of on chip pixel memory. This is because the on chip pixel memory is 256 KB and our VGA resolution was 640*480 and each pixel was represented using 16 bits (5 bits red, 6 bits green, 5 bits blue). So the required memory for storing one frame data is 640*480*2 = 614.4 KB, but we have only 256KB of pixel on chip memory.

One solution was to store 320*240 (half the original resolution) frame data and use scaler to double the resolution but while doubling it duplicates the both x and y pixels, this might create excess number of particles on the screen because of duplication. To avoid this we are using 64 MB (32M *16) SDRAM as pixel memory.

VGA:

Hardware Design:

The Hardware part of the VGA involved developing the Video-Output System that consists of Video IP cores(hardware controllers) together with SDRAM and the On-chip memory that facilitate the display of Video data.The hardware controllers make use of the streaming protocol to transfer the pixel values.These hardware controllers and the memory controllers for the SDRAM and the On-chip memory, were designed using Qsys (platform designer). We developed two Qsys designs, one for the VGA subsystem and a top level Qsys file, the Computer System which includes the processor, the memory controllers and the VGA subsystem module.

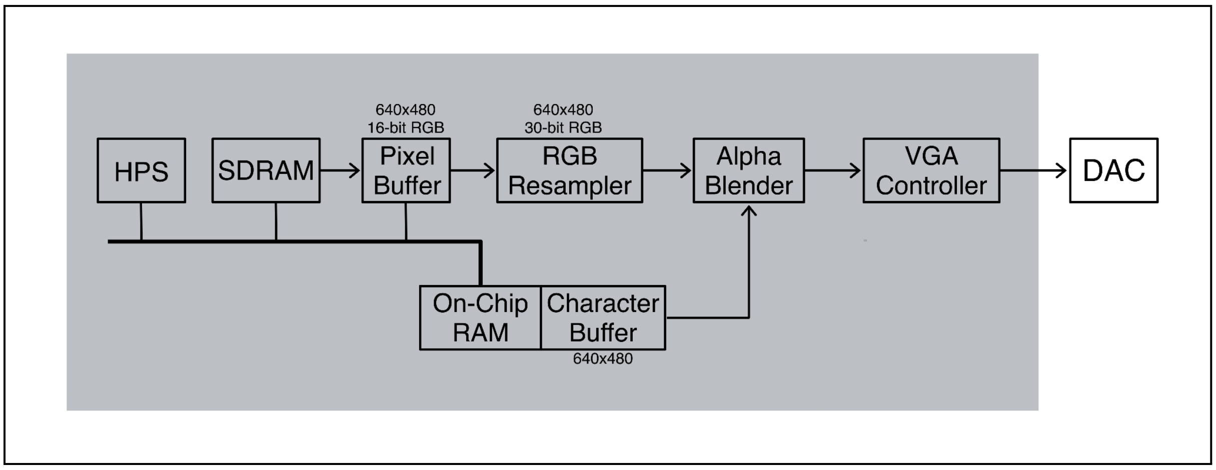
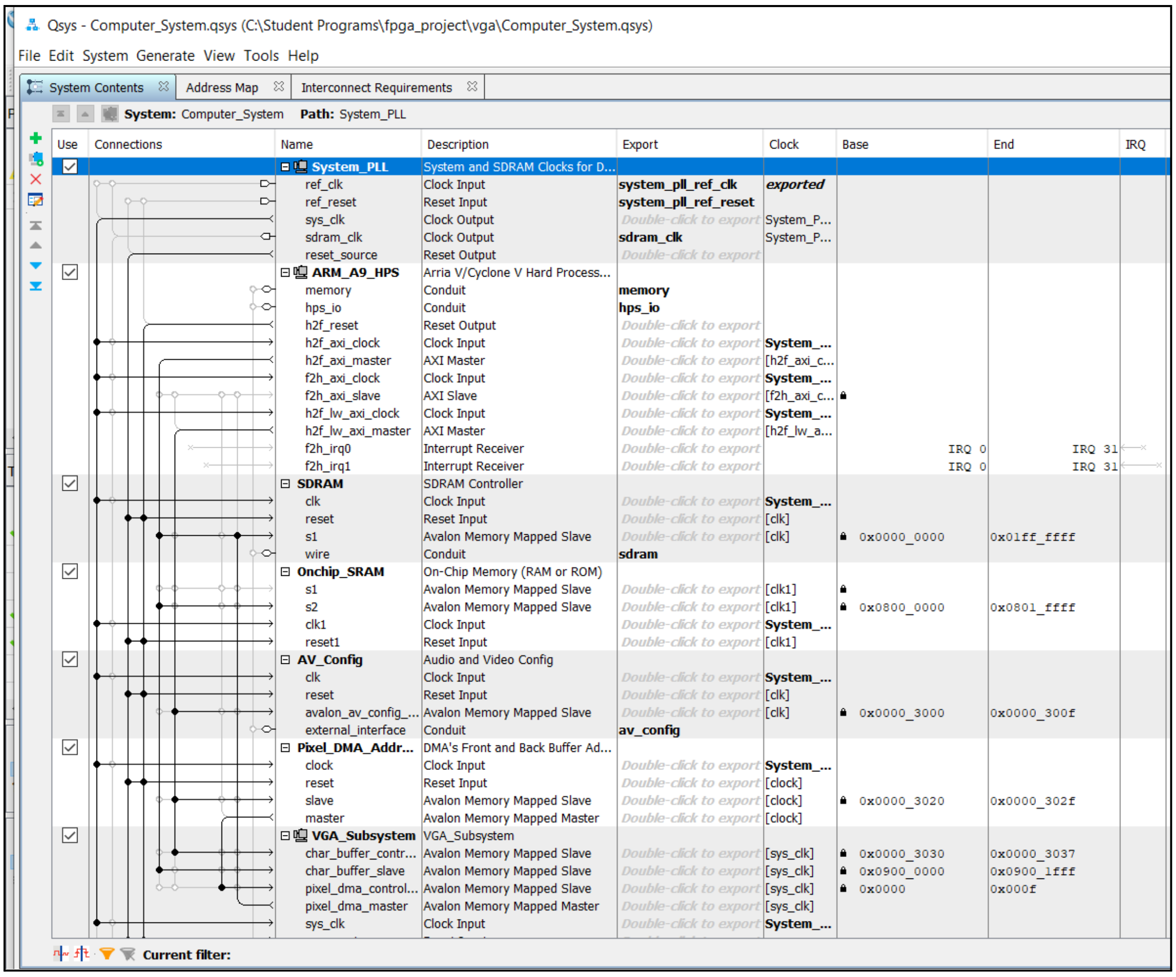


Fig 9: Component IP’s of the VGA subsystem.

The above-mentioned graphic depicts the IP cores and their connections that make up the VGA subsystem.The SDRAM writes the pixel values of each frame into the pixel buffer (of size 640x480 pixels), and similarly the On-chip RAM stores the characters (each measuring 8x8 pixels) to be displayed in each frame and writes the character data into the Character buffer. The Resampler converts the 16-bit RGB pixel value to 30-bit RGB pixel value and feeds the data into Alpha-Blender that combines the foreground Character data and the background pixel data into an image frame and hands it off to the VGA Controller. The DAC then sequentially displays each image frame on the monitor after receiving the frame data from the Controller and the timing signals it generates.

The top level Qsys file is shown in the figure below. It depicts the data transfer between the HPS, SDRAM, On-chip RAM, and the VGA Subsystem. The System PLL generates an input reference clock of 50MHz frequency. The VGA Controller demands to be operated on a 25MHz clock. The Dual-Clock FIFO helps transfer data stream between two clock domains.The data is buffered in a FIFO memory at input clock frequency. Then, the data is read out of the FIFO at the output clock frequency and streamed out of the core.


Qsys - Computer_System.qsys (C:\Student Programs\fpga_project\vga\Computer_System.qsys)

File Edit System Generate View Tools Help

System Contents | Address Map | Interconnect Requirements

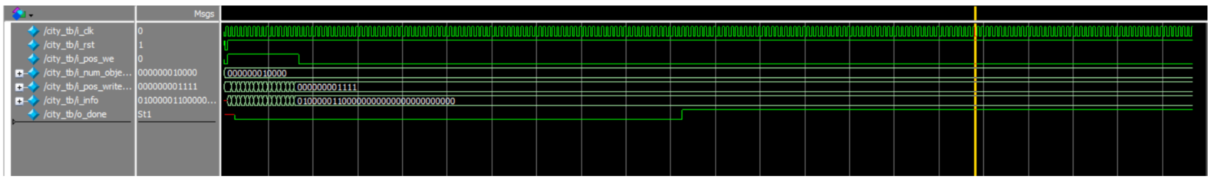
System: Computer_System **Path:** System_PLL

Use	Connections	Name	Description	Export	Clock	Base	End	IRQ
✓		**System_PLL**	System and SDRAM Clocks for D...					
		ref_clk	Clock Input	**system_pll_ref_clk**	*exported*			
		ref_reset	Reset Input	**system_pll_ref_reset**				
		sys_clk	Clock Output	*Double-click to export*	System_P...			
		sdram_clk	Clock Output	**sdram_clk**	System_P...			
		reset_source	Reset Output	*Double-click to export*				
✓		**ARM_A9_HPS**	Arria V/Cyclone V Hard Process...					
		memory	Conduit	**memory**				
		hps_io	Conduit	**hps_io**				
		h2f_reset	Reset Output	*Double-click to export*				
		h2f_axi_clock	Clock Input	*Double-click to export*	**System_...**			
		h2f_axi_master	AXI Master	*Double-click to export*	[h2f_axi_c...			
		f2h_axi_clock	Clock Input	*Double-click to export*	**System_...**			
		f2h_axi_slave	AXI Slave	*Double-click to export*	[f2h_axi_c...	🔒		
		h2f_lw_axi_clock	Clock Input	*Double-click to export*	**System_...**			
		h2f_lw_axi_master	AXI Master	*Double-click to export*	[h2f_lw_a...			
		f2h_irq0	Interrupt Receiver	*Double-click to export*		IRQ 0	IRQ 31	
		f2h_irq1	Interrupt Receiver	*Double-click to export*		IRQ 0	IRQ 31	
✓		**SDRAM**	SDRAM Controller					
		clk	Clock Input	*Double-click to export*	**System_...**			
		reset	Reset Input	*Double-click to export*	[clk]			
		s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	🔒 0x0000_0000	0x01ff_ffff	
		wire	Conduit	**sdram**				
✓		**Onchip_SRAM**	On-Chip Memory (RAM or ROM)					
		s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk1]	🔒		
		s2	Avalon Memory Mapped Slave	*Double-click to export*	[clk1]	🔒 0x0800_0000	0x0801_ffff	
		clk1	Clock Input	*Double-click to export*	**System_...**			
		reset1	Reset Input	*Double-click to export*	[clk1]			
✓		**AV_Config**	Audio and Video Config					
		clk	Clock Input	*Double-click to export*	**System_...**			
		reset	Reset Input	*Double-click to export*	[clk]			
		avalon_av_config_...	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	🔒 0x0000_3000	0x0000_300f	
		external_interface	Conduit	**av_config**				
✓		**Pixel_DMA_Addr...**	DMA's Front and Back Buffer Ad...					
		clock	Clock Input	*Double-click to export*	**System_...**			
		reset	Reset Input	*Double-click to export*	[clock]			
		slave	Avalon Memory Mapped Slave	*Double-click to export*	[clock]	🔒 0x0000_3020	0x0000_302f	
		master	Avalon Memory Mapped Master	*Double-click to export*	[clock]			
✓		**VGA_Subsystem**	VGA_Subsystem					
		char_buffer_contr...	Avalon Memory Mapped Slave	*Double-click to export*	[sys_clk]	🔒 0x0000_3030	0x0000_3037	
		char_buffer_slave	Avalon Memory Mapped Slave	*Double-click to export*	[sys_clk]	🔒 0x0900_0000	0x0900_1fff	
		pixel_dma_control...	Avalon Memory Mapped Slave	*Double-click to export*	[sys_clk]	🔒 0x0000	0x000f	
		pixel_dma_master	Avalon Memory Mapped Master	*Double-click to export*	[sys_clk]			
		sys_clk	Clock Input	*Double-click to export*	**System_...**			

Current filter:

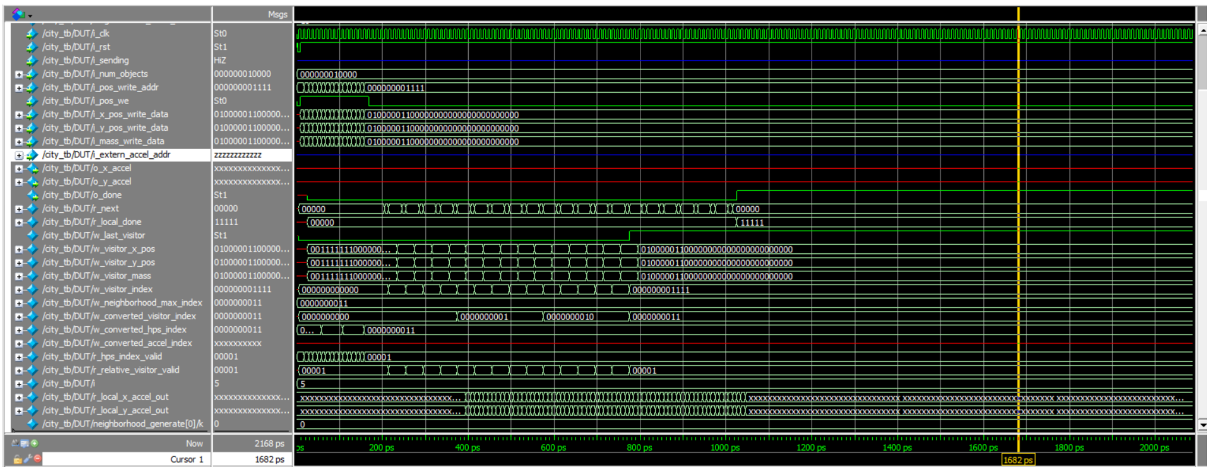
Fig 10: Top-level Qsys file.

Simulation Results:



```
/city_tb/i_clk          0
/city_tb/i_rst          1
/city_tb/i_pos_we       0
/city_tb/i_num_obje...  000000010000
/city_tb/i_pos_write... 000000001111
/city_tb/i_info         01000001100000...
/city_tb/o_done         St1
```

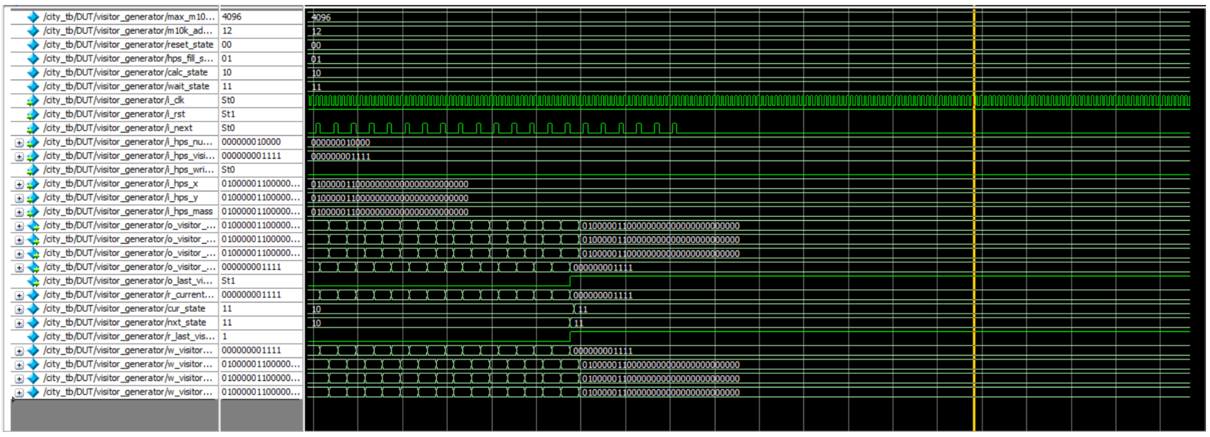
Fig 11: Top level simulation result.



```
                                                  Msgs
/city_tb/DUT/i_clk                                St0
/city_tb/DUT/i_rst                                St1
/city_tb/DUT/i_sending                            HiZ
/city_tb/DUT/i_num_objects                        000000010000
/city_tb/DUT/i_pos_write_addr                     000000001111
/city_tb/DUT/i_pos_we                             St0
/city_tb/DUT/i_x_pos_write_data                   01000001100000...
/city_tb/DUT/i_y_pos_write_data                   01000001100000...
/city_tb/DUT/i_mass_write_data                    01000001100000...
/city_tb/DUT/i_extern_accel_addr                  zzzzzzzzzzzz
/city_tb/DUT/o_x_accel                            xxxxxxxxxxxxxx...
/city_tb/DUT/o_y_accel                            xxxxxxxxxxxxxx...
/city_tb/DUT/o_done                               St1
/city_tb/DUT/r_next                               00000
/city_tb/DUT/r_local_done                         11111
/city_tb/DUT/w_last_visitor                       St1
/city_tb/DUT/w_visitor_x_pos                      01000001100000...
/city_tb/DUT/w_visitor_y_pos                      01000001100000...
/city_tb/DUT/w_visitor_mass                       01000001100000...
/city_tb/DUT/w_visitor_index                      000000001111
/city_tb/DUT/w_neighborhood_max_index             0000000011
/city_tb/DUT/w_converted_visitor_index            0000000011
/city_tb/DUT/w_converted_hps_index                0000000011
/city_tb/DUT/w_converted_accel_index              xxxxxxxxxx
/city_tb/DUT/r_hps_index_valid                    00001
/city_tb/DUT/r_relative_visitor_valid             00001
/city_tb/DUT/i                                    5
/city_tb/DUT/r_local_x_accel_out                  xxxxxxxxxxxxxx...
/city_tb/DUT/r_local_y_accel_out                  xxxxxxxxxxxxxx...
/city_tb/DUT/neighborhood_generate[0]/k           0
                                          Now     2168 ps
                                     Cursor 1     1682 ps
```

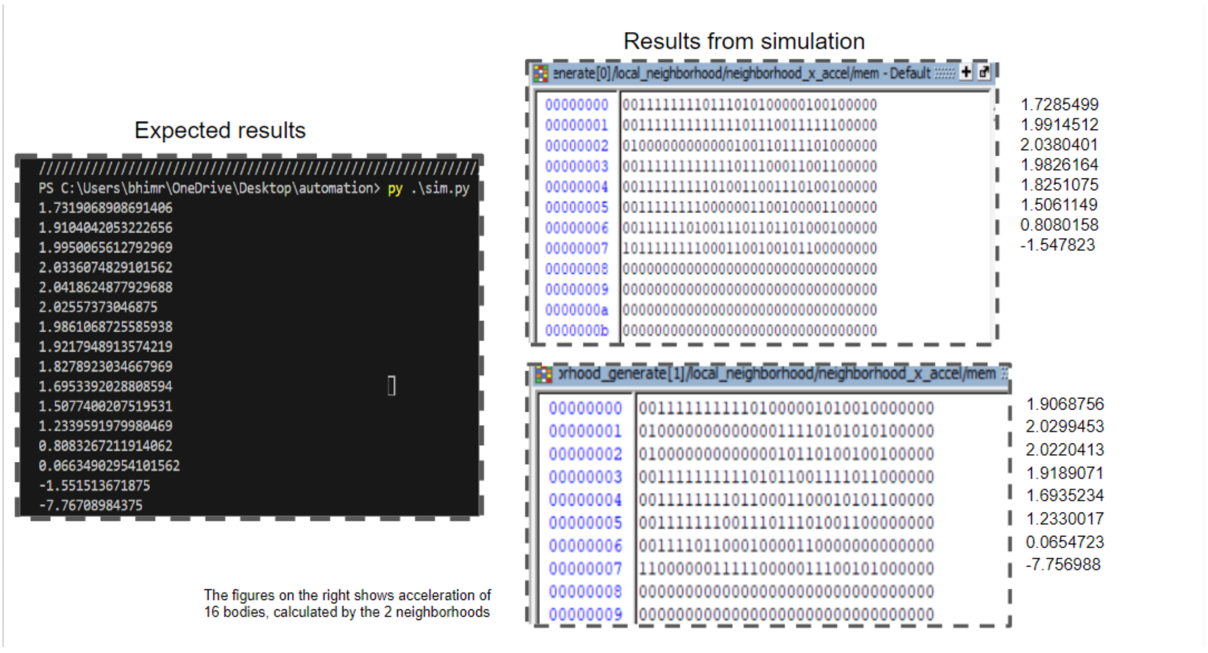
Fig 12: Waveforms of one neighbourhood controller.

(the unknown values in figure 12 are because we are using testbench as HPS and some of the signals which are not useful in verification we are not sending it to FPGA).



```
/city_tb/DUT/visitor_generator/max_m10...     4096
/city_tb/DUT/visitor_generator/m10k_ad...     12
/city_tb/DUT/visitor_generator/reset_state    00
/city_tb/DUT/visitor_generator/hps_fill_s...  01
/city_tb/DUT/visitor_generator/calc_state     10
/city_tb/DUT/visitor_generator/wait_state     11
/city_tb/DUT/visitor_generator/i_clk          St0
/city_tb/DUT/visitor_generator/i_rst          St1
/city_tb/DUT/visitor_generator/i_next         St0
/city_tb/DUT/visitor_generator/i_hps_nu...    000000010000
/city_tb/DUT/visitor_generator/i_hps_visi...  000000001111
/city_tb/DUT/visitor_generator/i_hps_wri...   St0
/city_tb/DUT/visitor_generator/i_hps_x        01000001100000...
/city_tb/DUT/visitor_generator/i_hps_y        01000001100000...
/city_tb/DUT/visitor_generator/i_hps_mass     01000001100000...
/city_tb/DUT/visitor_generator/o_visitor_...  01000001100000...
/city_tb/DUT/visitor_generator/o_visitor_...  01000001100000...
/city_tb/DUT/visitor_generator/o_visitor_...  01000001100000...
/city_tb/DUT/visitor_generator/o_visitor_...  000000001111
/city_tb/DUT/visitor_generator/o_last_vi...   St1
/city_tb/DUT/visitor_generator/r_current...   000000001111
/city_tb/DUT/visitor_generator/cur_state      11
/city_tb/DUT/visitor_generator/nxt_state      11
/city_tb/DUT/visitor_generator/r_last_vis...  1
/city_tb/DUT/visitor_generator/w_visitor...   000000001111
/city_tb/DUT/visitor_generator/w_visitor...   01000001100000...
/city_tb/DUT/visitor_generator/w_visitor...   01000001100000...
/city_tb/DUT/visitor_generator/w_visitor...   01000001100000...
```

Fig 13: Waveforms of visitor centre controller.



```
////////////////////////////////////////////////////////////////
PS C:\Users\bhimr\OneDrive\Desktop\automation> py .\sim.py
1.7319068908691406
1.9104042053222656
1.9950065612792969
2.0336074829101562
2.0418624877929688
2.02557373046875
1.9861068725585938
1.9217948913574219
1.8278923034667969
1.6953392028808594
1.5077400207519531
1.2339591979980469
0.8083267211914062
0.06634902954101562
-1.551513671875
-7.76708984375
```

```
enerate[0]/local_neighborhood/neighborhood_x_accel/mem - Default
00000000  00111111101110101000001001000000   1.7285499
00000001  00111111111111101110011111100000   1.9914512
00000002  01000000000000100110111101000000   2.0380401
00000003  00111111111111101110001100110000   1.9826164
00000004  00111111110100110011101001000000   1.8251075
00000005  00111111100000011001000011000000   1.5061149
00000006  00111111010011101101101000100000   0.8080158
00000007  10111111110001100100101100000000   -1.547823
00000008  00000000000000000000000000000000
00000009  00000000000000000000000000000000
0000000a  00000000000000000000000000000000
0000000b  00000000000000000000000000000000
```

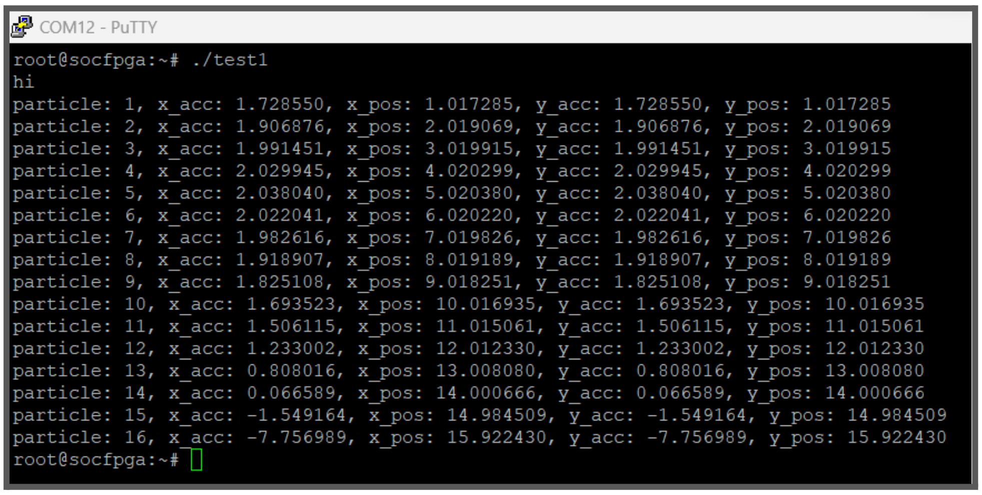
```
orhood_generate[1]/local_neighborhood/neighborhood_x_accel/mem
00000000  00111111111101000001010010000000   1.9068756
00000001  01000000000000011110101010100000   2.0299453
00000002  01000000000000010110100100100000   2.0220413
00000003  00111111111101011001111011000000   1.9189071
00000004  00111111110110001100010101100000   1.6935234
00000005  00111111100111011101001100000000   1.2330017
00000006  00111101100010000110000000000000   0.0654723
00000007  11000000111110000011100101000000   -7.756988
00000008  00000000000000000000000000000000
00000009  00000000000000000000000000000000
```

The figures on the right shows acceleration of
16 bodies, calculated by the 2 neighborhoods

Fig 14: Automated verification results

In-circuit Verification:

- ☐ First, send initial X and Y positions and mass of each celestial body from HPS to FPGA.
- ☐ Second, receive the computed acceleration values from the FPGA.
- ☐ Third, integrate the acceleration to get new X, Y velocity and positions.
- ☐ Finally, send the new X and Y positions to HPS.
- ☐ Each iteration of all the above four steps create one frame on the display and our aim is to achieve maximum FPS by parallelizing computation as much as possible.



```
root@socfpga:~# ./test1
hi
particle: 1, x_acc: 1.728550, x_pos: 1.017285, y_acc: 1.728550, y_pos: 1.017285
particle: 2, x_acc: 1.906876, x_pos: 2.019069, y_acc: 1.906876, y_pos: 2.019069
particle: 3, x_acc: 1.991451, x_pos: 3.019915, y_acc: 1.991451, y_pos: 3.019915
particle: 4, x_acc: 2.029945, x_pos: 4.020299, y_acc: 2.029945, y_pos: 4.020299
particle: 5, x_acc: 2.038040, x_pos: 5.020380, y_acc: 2.038040, y_pos: 5.020380
particle: 6, x_acc: 2.022041, x_pos: 6.020220, y_acc: 2.022041, y_pos: 6.020220
particle: 7, x_acc: 1.982616, x_pos: 7.019826, y_acc: 1.982616, y_pos: 7.019826
particle: 8, x_acc: 1.918907, x_pos: 8.019189, y_acc: 1.918907, y_pos: 8.019189
particle: 9, x_acc: 1.825108, x_pos: 9.018251, y_acc: 1.825108, y_pos: 9.018251
particle: 10, x_acc: 1.693523, x_pos: 10.016935, y_acc: 1.693523, y_pos: 10.016935
particle: 11, x_acc: 1.506115, x_pos: 11.015061, y_acc: 1.506115, y_pos: 11.015061
particle: 12, x_acc: 1.233002, x_pos: 12.012330, y_acc: 1.233002, y_pos: 12.012330
particle: 13, x_acc: 0.808016, x_pos: 13.008080, y_acc: 0.808016, y_pos: 13.008080
particle: 14, x_acc: 0.066589, x_pos: 14.000666, y_acc: 0.066589, y_pos: 14.000666
particle: 15, x_acc: -1.549164, x_pos: 14.984509, y_acc: -1.549164, y_pos: 14.984509
particle: 16, x_acc: -7.756989, x_pos: 15.922430, y_acc: -7.756989, y_pos: 15.922430
root@socfpga:~#
```

Fig 15: In circuit verification results

Software Design:

The software for the project had a baseline structure of sending information to the FPGA, waiting for the calculations to finish up and be done and then reading the new data back to be displayed on the monitor. The first initial sending of information was the current position of every point. These values would be stored inside of M10K blocks in the HPS to be used later. New accelerations for every point were calculated with respect to every other point, and once these calculations were done the FPGA would signal the HPSof its completion and then proceed to send all of the values back, update the position based on the time step and then display the results.

We started with the basic mapping of each body on the screen with no user interface, each particle is represented as '+' on VGA and while writing the new frame the old location is erased and the new location is painted with the corresponding colour of the particle. Each frame instead of writing entire frame data, only changes from the previous frame are written, reducing the SDRAM writing speed bottleneck.

Once we got the results on VGA, we started to add the buttons on the screen. We decided that the UI will be mouse enabled and we studied the professor Bruce Land (Cornell university) mouse integration to FPGA project and integrated the USB mouse to the FPGA. Adding buttons was incremental and we started with a reset and play/pause button. At last now we have 10 buttons and functionality of each is listed below.

Reset : Resets the entire simulation, makes it start running from the beginning.

Play/Pause: Enables users to play and pause the simulation.

+ : Increases the time step of integration which makes particles move quickly.

- : decreases the time step of integration which makes particles move slowly.

Add particle : Adds the new particle of entered information (mass, x and y position and velocity) on the screen.

FPGA : When pressed, computes the acceleration on the FPGA.

HPS : When preseed, computes the acceleration on the HPS.

Maps: As of now we have four datasets of particles and with this button users can choose the different datasets.

Bound: If enabled particles will get bounced back from the screen edges, so that we can make them always present on the screen, else particles will go out of frame.

Trace: If pressed displays the path taken by the particle instead of just the particle.

Each button is a square box and an alphabet is written inside them of corresponding functionality.

Next we created the gravity simulator on the HPS system to be able to test the rest of the HPS features. We began with defining what each celestial body needed for its calculations on the HPS and FPGA and how we would get the data into the HPS. After creating a test map with a hundred points and being able to load it into the HPS, we worked on creating a simulator that was O(N^2) and saved the accelerations for both of the celestial bodies being looked at. It also used the fast-inverse square root algorithm when calculating the forces between two different celestial bodies. The integrator was then written for the HPS simulator and a mini-demo was made to see how the HPS simulated the celestial bodies and the cycle required for drawing, simulating, and erasing celestial bodies. When drawing a celestial body, we decided to erase the exact location where it was before by colouring it in with black which matched the background since this allowed for it to be faster instead of erasing the entire screen each time we wanted to update the screen. We also checked to see if the celestial body was on the screen before drawing it since drawing it off screen would take up time and not produce any result.

On the screen we are also displaying the number of bodies, time step and frame time, these information can be used for debugging purposes. The entire Code is compiled using makefile which we wrote and for cross compiling we used the SOC-EDS tool from Quartus.

One trickier part was the computed acceleration must be read as an unsigned number not as the floating number, so in HPS we need to convert this Unsigned number to floating point for integrating. But our choice of 32 bit representation of floating point (27 bit floating point and 5’b0) helped us and the conversion process became extremely easy, just changing the datatype by typecasting it to float and dereferencing the unsigned value as floating value.

Linux OS handles memory mapped peripherals of HPS as files. Wireless mouse of 10000 DPI, 40*40 pixel array and reporting frequency of 120 Hz reports change in x and y positions 120 times in one second. The change in x and y positions are calculated using a cross correlation algorithm in which it subtracts the previous image from the shifted version of the current image until the subtraction yields minimum value. The shifts in the current image yields the change in x and y positions. The mouse we are using takes 170000 images per second and reports the change in x and y positions 120 times in one second. Each time the USB transceiver sends 3 bytes of data. First and second byte corresponds to change in x and y positions respectively and third byte corresponds to clicks on the mouse. Third byte 1,2 and 4 corresponds to left, right and middle clicks respectively.

Results:

☐ Acceleration computation time

# of particles	By FPGA(μs)	By HPS(μs)
30	30	65
1009	2950	78500
1681	7180	233800
2200	11670	411250

☐ Frame time

# of particles	By FPGA(μs)	By HPS(μs)
30	1470 (680 FPS)	1490 (671 FPS)
1009	8200 (121 FPS)	82000 (12 FPS)
1681	15200 (65 FPS)	240000 (4 FPS)
2200	21800 (45 FPS)	419000 (2 FPS)

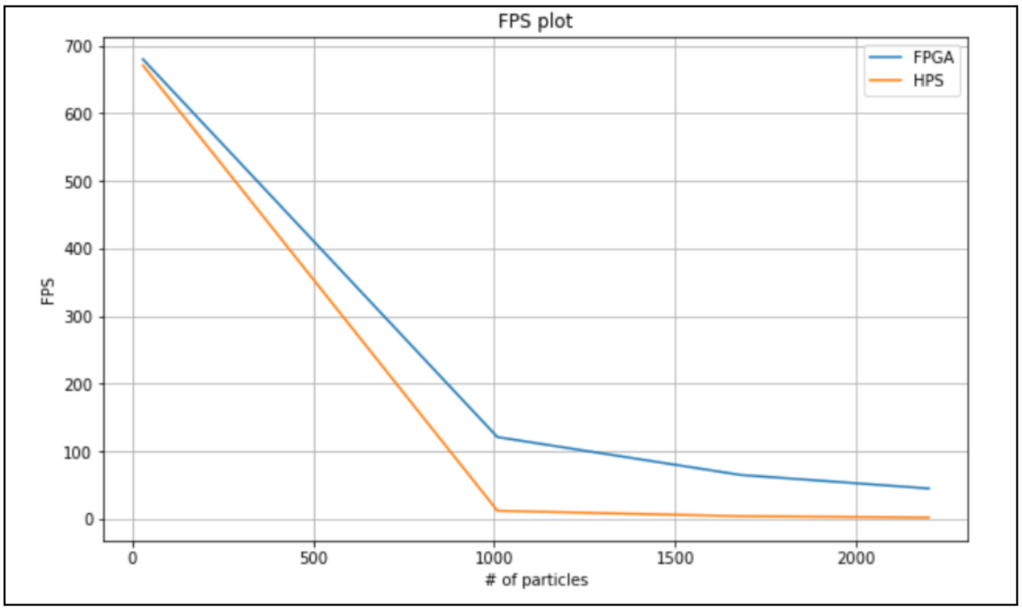


Fig 16: Frame rate vs. No. of bodies

As is evident from the graph we can see a massive improvement in the frame time on the FPGA vs the ARM. Each of these times were based on the maximum frame calculation from the ARM/FPGA in each frame cycle.

Conclusions:

We can see that the one frame acceleration computation time (from table 1) diverges when we have a larger number of bodies. Since the acceleration computations are independent of each other we made the decision to parallelize these calculations.

After parallelising using FPGA we can see the huge improvement in the frame time.

- For 2200 particles FPS on HPS is just 2 but on FPGA it is 45, and it is nearly 22.5 times improvement of each frame computation time.
- For fewer particles (around less than 100) the power of the FPGA cannot be explained much because even though HPS is computing sequentially it is operating at 925 Mhz but our FPGA is operating at 100 Mhz.
- Since both HPS and FPGA are 28 nm node and operating frequency of HPS is 925 Mhz and FPGA is 100 Mhz, with the same node the power dissipation decreases as operating frequency decreases. Hence FPGA consumes lesser power.

“Finally we were successful in unleashing the power of the parallel processing by the FPGA”

Challenges Faced:

1. **Network Setup:** Setting up a connection between our host computer and the DE-10 board was a very challenging task. After struggling for a few days, we came up with 2 methods for establishing the connection.
    a. First Method:
        i. Establishing the internet by setting up a bridge connection between the WIFI port on the host and the Ethernet port on the board.
        ii. Unfortunately, this led to many failed attempts.
        iii. This method didn’t work, as the internet on the host computer is disconnected after the bridge is established.
    b. Second Method:
        i. Assigned Static IP address and a DNS address to the Ethernet port

through which the board was connected.

ii. Eventually, this method turned out to be successful.

iii. We were able to ping to other networks.

2. **Underflow problem**: We had 4 different datasets each consisting of a different number of celestial bodies. The dataset with the maximum number of bodies, didn't simulate as expected. So this led to sudden disappearing of particles from the frame.

   We tried to debug this issue for many days, and finally we began checking the real time position,velocity and the acceleration of random particles. We noticed that the acceleration values blew up to large values after a few iterations. This was due to the observed fact that a large dataset had celestial bodies very close to each other,due to this they were experiencing large accelerations and these values accumulated for a few iterations. So we found the issue was in **mishandling of the subnormal floating point numbers**. Then we fixed this issue by considering the underflow condition in our floating point modules and fixed this issue.

3. **Simultaneous operation of HPS and FPGA**: Initially we had linux desktop kernel image loaded into SD card. Bootloader reads the kernel image from the SD card and loads the Ubuntu version of linux into 1 GB DDR3 SDRAM of HPS. Once the OS gets loaded we can view the terminal interface of the OS in the putty terminal. But we observed that whenever we program the FPGA the HPS was getting stuck and we found out that the desktop version has a soc_system.rbf file which tries to initialise and configure the FPGA peripherals. This was creating conflict, for example both HPS and FPGA trying to access the ethernet PHY at the same time. The solution was to use the console version of linux kernel image or convert the .sof file to .rbf file using programming file conversion tool in quartus and place it in the partition where kernel image is present and configure the FPGA by parallel configuration with FPGA as slave and HPS as master. We opted for the console version of linux kernel image.

4. **Blank signal of VGA controller:** When we were trying to display a static image on the VGA we faced this issue. Each line of video begins with an active video region, in which RGB values are output for each pixel in the line. The active region is followed by a blanking region, in which black pixels are transmitted. In the middle of the blanking interval, a horizontal sync pulse is transmitted. The blanking interval before the sync pulse is known as the "front porch", and the blanking interval after the sync pulse is known as the "back porch". Note that the dedicated horizontal sync signal output from the FPGA directly to the VGA connector must be delayed by two clock cycles relative to the composite sync signal passed to the DAC chip, to account for the pipeline delay of the DAC. After one frame data is written on screen the light pulse needs to travel from bottom right to top left and in this period there must be no RGB values which acts as input to VGA DAC. This period is known as the vertical sync

period. To prevent values being written into DAC registers there is a signal called VGA_BALNK_N signal.When this line is pulled to low logic it forces zero to be written into these DAC RGB registers. Initially we were neglecting this signal and forcing it to be always zero, hence we were unable to get an image on the screen as DAC registers were always zero. We studied the DAC datasheet and figured out the solution.

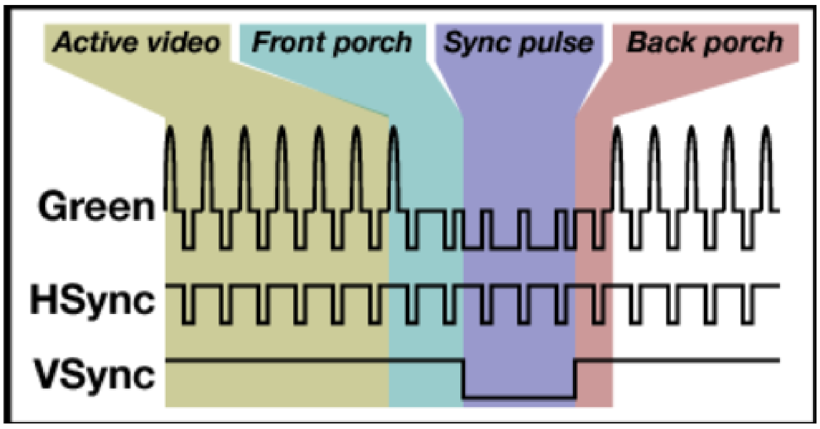


Fig 17: VGA signals

Video Output Level	IOG (mA)	$\overline{\text{IOG}}$ (mA)	IOR/IOB (mA)	$\overline{\text{IOR}}/\overline{\text{IOB}}$ (mA)	$\overline{\text{SYNC}}$	$\overline{\text{BLANK}}$	DAC Input Data
White Level	26.0	0	18.67	0	1	1	0x3FFH
Video	Video + 7.2	18.67 – Video	Video	18.67 – Video	1	1	Data
Video to $\overline{\text{BLANK}}$	Video	18.67 – Video	Video	18.67 – Video	0	1	Data
Black Level	7.2	18.67	0	18.67	1	1	0x000H
Black to $\overline{\text{BLANK}}$	0	18.67	0	18.67	0	1	0x000H
$\overline{\text{BLANK}}$ Level	7.2	18.67	0	18.67	1	0	0xXXXH (don't care)
$\overline{\text{SYNC}}$ Level	0	18.67	0	18.67	0	0	0xXXXH (don't care)

Fig 18: Typical Video Output Truth Table (RSET = 530 Ω, RLOAD = 37.5 Ω)

5. **Blocking mouse polling:** we wrote basic mouse integration code to read the mouse data and we observed that when there was an event on mouse, only at that time our UI C code was used to run else simulations were paused. This was because the simple read operation of the mouse event was blocking read means it will block the processor execution until there is an event on the mouse. To resolve this we appended the NON_BLOCK to the set of flags and everything started working properly.

6. **Huge compilation time:** The full compilation time was around 30 minutes, so each time we debug some part of the verilog code we used to wait for 30 minutes which was sometimes more annoying.

Setup Image:

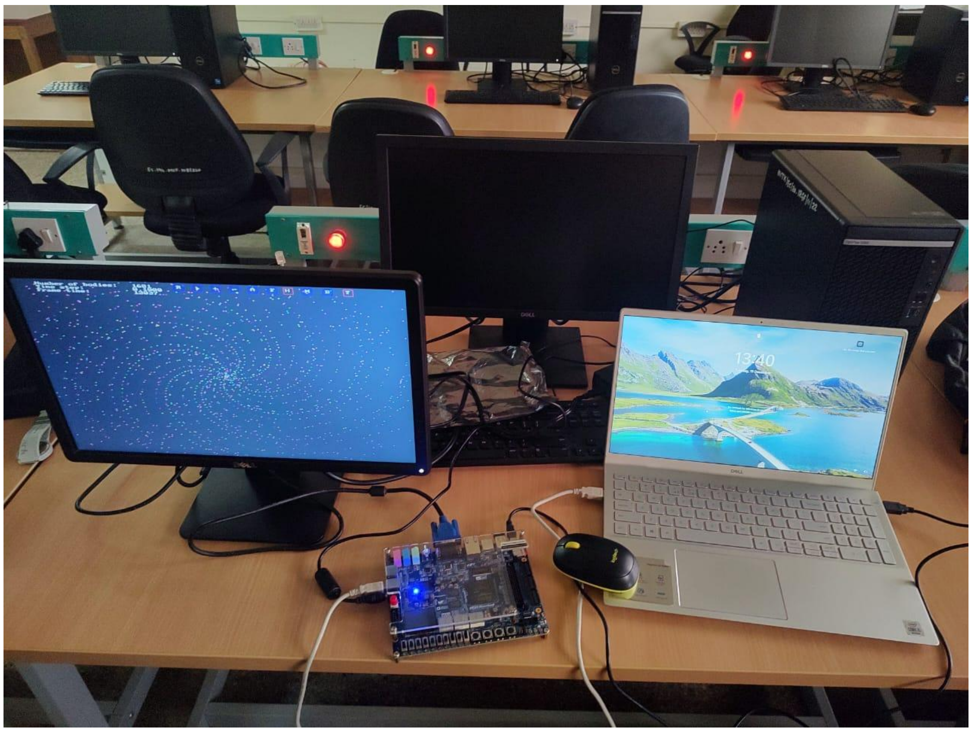


Fig 18: Full setup image

Peripherals used:

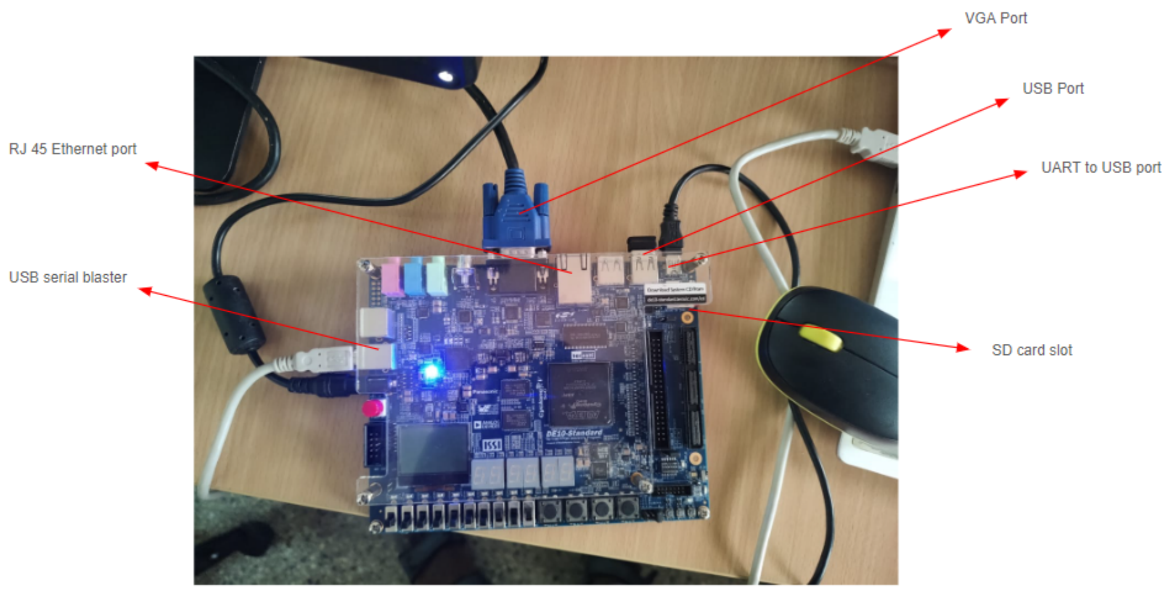


Fig 19: Peripherals used in project

Limitations:

1. Minimum number of bodies in each Neighborhood must be at least 3, this because of the final 2 cycle pipelined adder which takes two clock cycles to update results before reading it.

2. Lesser accuracy, since we are using 27 bit floating point for minimising the resource utilisation.

3. Mouse polling rate depends on frame time, on HPS where we observed 2 FPS the mouse was very less interactive since it was getting polled for only 2 times in one second. This can be resolved using the multithreading concept.

Reference[s]:

1. GRAPE-6: Massively-Parallel Special-Purpose Computer for Astrophysical Particle Simulations. (publication in the Astronomical Society of Japan).

2. ECE-5760 the Cornell university project called "Gravitational Simulators".

3. Particle Box online simulator (http://particlesandbox.com/)

4. J. Barnes and P. Hut. 1986. A hierarchical O(N log N) force-calculation algorithm. Nature 324 (Dec. 1986), 446–449.

Github repo link:
https://github.com/omikami747/grav_sim.git

Demo video link:
https://drive.google.com/drive/folders/1Ml40HDuvfR9mu9PndPdcZwAu4PXh0R9n?usp=sharing

(" Demo video contains a very clear intuitive explanation of our 2D N body gravity Simulator Project").