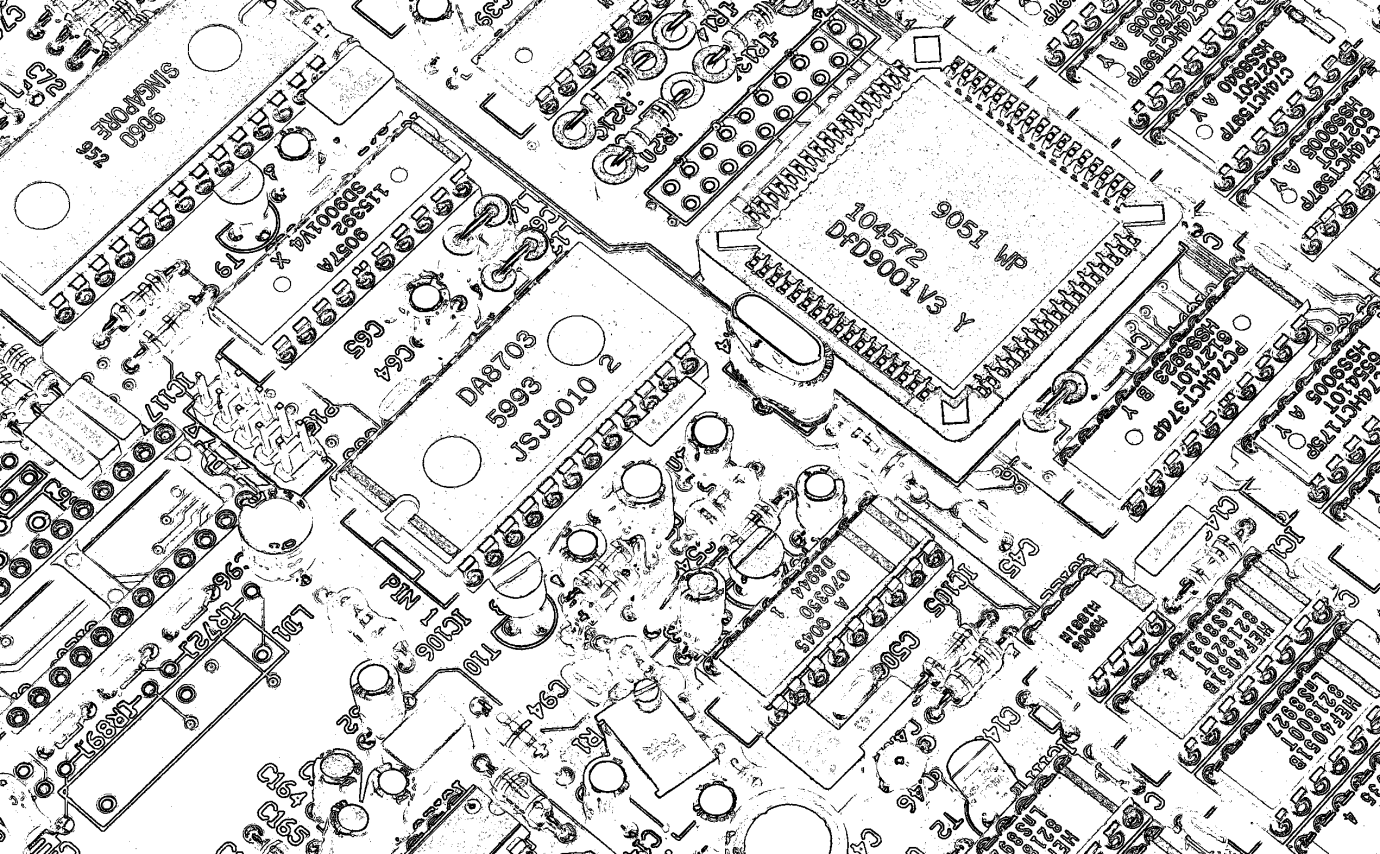
A Triangle Drawing Machine: The Design and Implementation of a Microcomputer, VGA Driver, Programming Language, Assembler & More…

Advanced Topics in Information Technology 6206ICT, Digital Systems

By Luke Lincoln, S2807774

Email: [luke.lincoln@griffithuni.edu.au](mailto:luke.lincoln@griffithuni.edu.au)

Contents

[Problem Statement 3](#_Toc371506122)

[Solution 3](#_Toc371506123)

[Digital Techniques 3](#_Toc371506124)

[VGA Driver & Frame Buffer 3](#_Toc371506125)

[Microcomputer 6](#_Toc371506126)

[Luke’s Assembly Language 4.0 (LAL4) 10](#_Toc371506127)

[The LAL4 Assembler & Interpreter 11](#_Toc371506128)

[Triangles, Drawing, and Assembly Techniques 15](#_Toc371506129)

[FPGA implementation 21](#_Toc371506130)

[The Image Indexer 21](file:///C:\Users\Lukes\Desktop\Dropbox\digital%20systems\ass%202\Report.docx#_Toc371506131)

[21](#_Toc371506132)

[The Counter used in the VGA driver 21](file:///C:\Users\Lukes\Desktop\Dropbox\digital%20systems\ass%202\Report.docx#_Toc371506133)

[VGA Signal Generator Module 22](file:///C:\Users\Lukes\Desktop\Dropbox\digital%20systems\ass%202\Report.docx#_Toc371506134)

[ALU Module Part 1 23](file:///C:\Users\Lukes\Desktop\Dropbox\digital%20systems\ass%202\Report.docx#_Toc371506135)

[23](#_Toc371506136)

[ALU Module Part 2 24](file:///C:\Users\Lukes\Desktop\Dropbox\digital%20systems\ass%202\Report.docx#_Toc371506137)

[ALU Module Part 3 25](file:///C:\Users\Lukes\Desktop\Dropbox\digital%20systems\ass%202\Report.docx#_Toc371506138)

[CPU & VGA Driver Part 1 26](file:///C:\Users\Lukes\Desktop\Dropbox\digital%20systems\ass%202\Report.docx#_Toc371506139)

[CPU & VGA Driver Part 2 27](file:///C:\Users\Lukes\Desktop\Dropbox\digital%20systems\ass%202\Report.docx#_Toc371506140)

[CPU & VGA Driver Part 3 28](file:///C:\Users\Lukes\Desktop\Dropbox\digital%20systems\ass%202\Report.docx#_Toc371506141)

[28](#_Toc371506142)

[Final Project Outcome 29](#_Toc371506143)

# Problem Statement

Build an electronic machine which can draw an arbitrary list of triangles to a television monitor using the Video Graphics Array (VGA) interface. This device must be implemented using a Field-programmable gate array (FPGA) using various digital techniques learned in the course.

# Solution

My solution to this problem was to design my own microcomputer and assembly language. The assembly language must be simple enough for a one semester project, but must also have the robustness and completeness required to implement a triangle drawing algorithm. The microcomputer must be fully able to run the assembly language, and be able to interface with memory and output in the form of visual information. Besides the microcomputer and assembly language, a VGA driver must also be implemented, the driver must be able to transmit the image given to it by the microcomputer, and use the correct timing and protocol required by VGA input. To accomplish this, a frame buffer device was also designed and built; the system was implemented using a hardware description language and works successfully.

# Digital Techniques

## VGA Driver & Frame Buffer

The VGA driver generates the necessary signals to operate a television or display which honours the VGA standard. The dimensions of the input image are 640 by 480; the driver uses a sequential method to colour in each pixel. The VGA display uses 5 input signals which must be generated by the VGA driver using the exact timing requirements. A high level explanation of the VGA driver and frame buffer is shown in figure 1. The VGA driver has 14 output logic lines, one for vertical synchronization (frame complete), one for horizontal synchronization (pixel row complete), the other twelve lines contain 3 buses which of four bits each, these 3 buses correspond to the red, green and blue colour signals. The screen refreshes every 16.7 milliseconds; therefore it has a refresh frequency of around 60 Hz. The clock which synchronizes the VGA system is a 25MHz square wave oscillator. The horizontal sync signal is pulsed before each row of colour is transmitted across the communication channel, and the vertical signal is pulsed before each frame of the video is sent. Both of these signals are active low, have a specific duration and both have a buffer time both before and after the signal begins and finishes. The pre sync signal buffers are called the front porch and back porch respectively. The timing requirements are listed in figure 2. I designed a module which takes a 25Mhz clock, and outputs the vertical and horizontal sync, as well as whether the colour data should be output (a signal called data out) and the x and y position for which the corresponding input RGB signals should be sent. This module is shown in figure 3. The mod-800 counter is used for the row signal generation and the mod-521 counter uses it’s RCO output as the clock input. The horizontal sync output and vertical sync output both go active high at the correct time range, the send data signal goes active high when the rgb data should be sent. The frame buffer memory is limited to a size which is less than the 640\*480\*4 bits which is required for a 640\*480 image, therefore an image of size 160\*120 is used for the frame buffer. To provide the correct indexing, the x and y coordinates are first divided by four (two right shifts) and the y coordinate is multiplied by 160 and added to the x coordinate, this is used as the address input for the frame buffer, the data is then selected to go to the output using a multiplexer. Figure 4 shows this module.

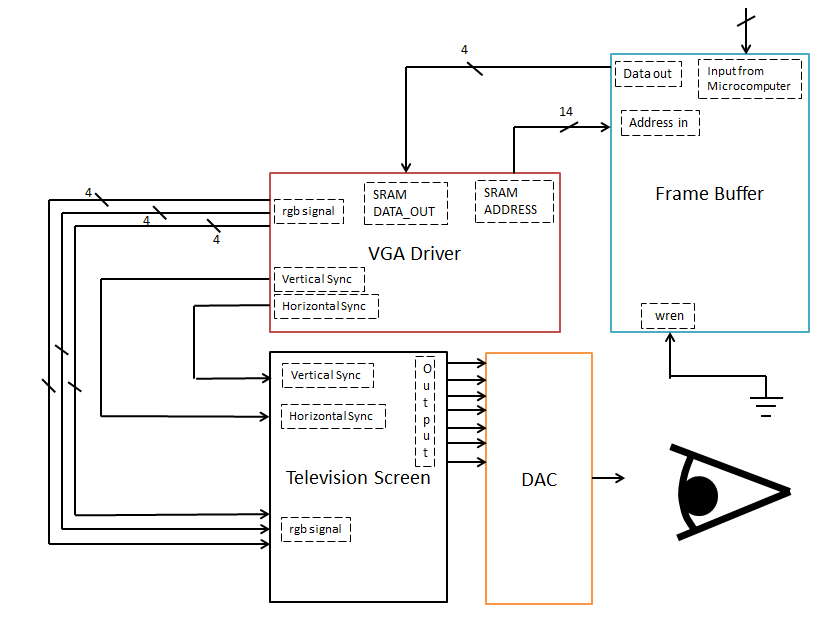


Figure 1 High level details of the vga driver and frame buffer

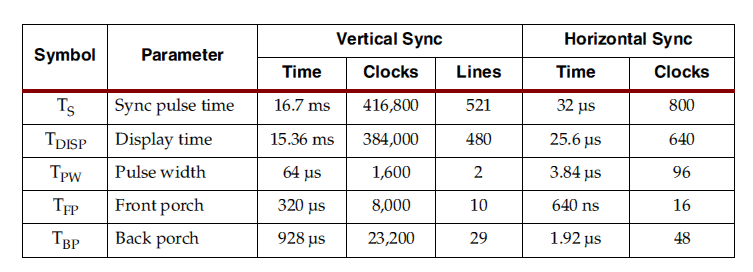


Figure 2 VGA synchronization timing table

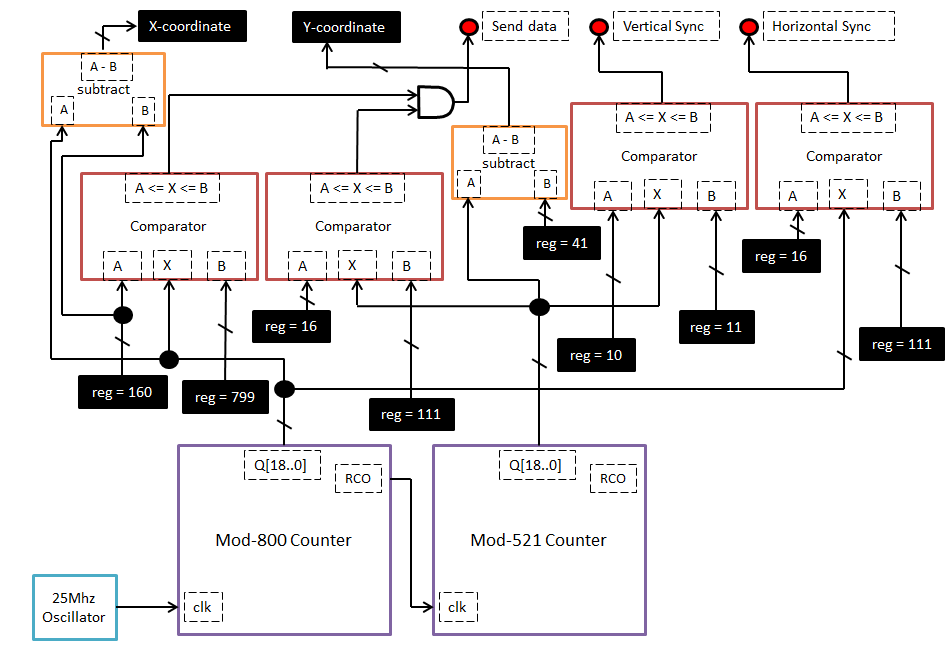


Figure 3 VGA DRIVER Module Design

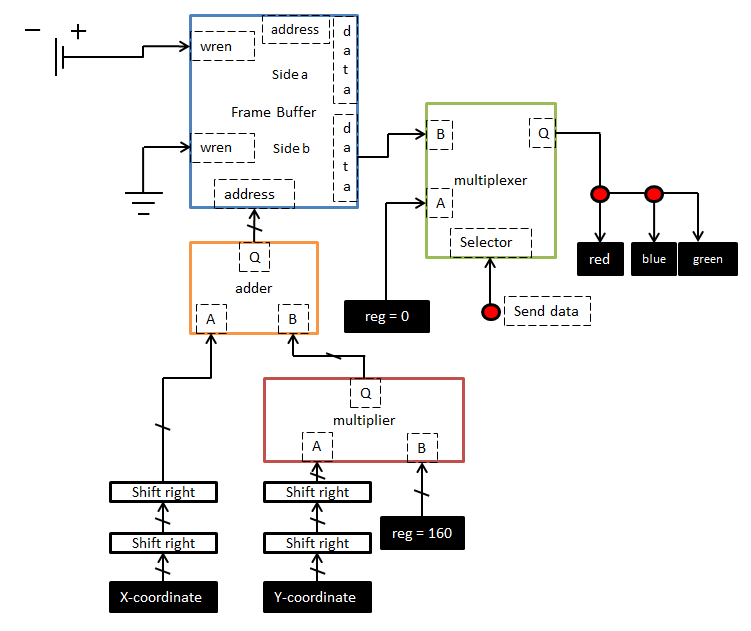


Figure 4 Frame Buffer To Display indexing

# Microcomputer

The microcomputer I designed includes a CPU for processing, SRAM containing ‘memory contents and machine code’ and a connection between the CPU and the frame buffer (for output). A diagram of the microcomputer is shown in figure 5. The program code is separated into a separate SRAM from the program memory, the CPU also controls the frame buffer. The program code has 40 bits per address, 32 bits to storage a number associated with each operation, 5-bits for the operation code and 3 bits are left blank. The CPU has several on board D-Flip Flop registers including: Accumulators A & B, the program counter PC and registers to store the current op-code and associated number. The CPU performs mathematical operations on numbers, reads and writes from memory and writes pixels to the frame buffer. There are two main sections: the ALU and the control unit. The ALU is responsible for performing mathematical operations and also modifying the program counter. The control unit controls the various signals for reading and writing to the memory units and frame buffer. Figure 6 shows a diagram of the high level operation of the proposed CPU. The registers in the CPU are processed and the output provides feedback to the new register values, the control unit receives input signals and sends output signals, the ALU processes all operations which do not require input or output. All of the operations the CPU can perform are listed in table 1. These operations are performed by either the ALU or the Control Unit, in figure 6, an encoder is used to select the correct output using a multiplexer. This provides the necessary feedback for the CPU registers. The CPU is a simple state machine, it first loads an operation then executes it, figure 7 shows this state diagram as well as the timing diagram which the CPU follows.

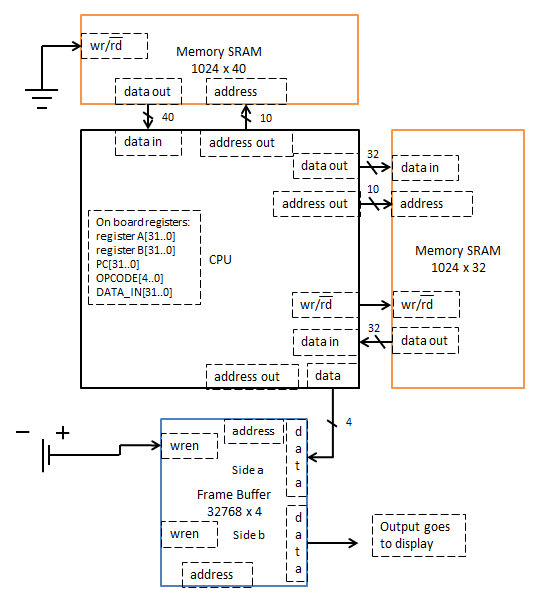


Figure 5 High level microcomputer diagram

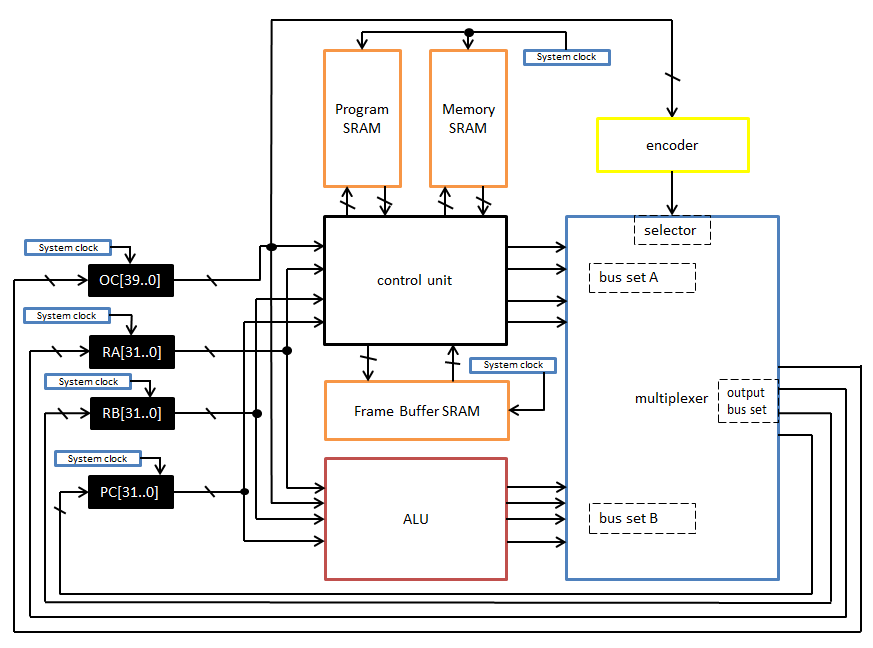


Figure 6 INside the cpu

Table 1 CPU Operations

|  |  |  |  |
| --- | --- | --- | --- |
| op-code | function | type | binary oc |
| 0 | Comment | Do Nothing | 00000 |
| 1 | Function | Do Nothing | 00001 |
| 2 | lda | Control Unit | 00010 |
| 3 | ldb | Control Unit | 00011 |
| 4 | ldafb | Control Unit | 00100 |
| 5 | sta | Control Unit | 00101 |
| 6 | stb | Control Unit | 00110 |
| 7 | staib | Control Unit | 00111 |
| 8 | setim | Control Unit | 01000 |
| 9 | bisa | ALU | 01001 |
| 10 | goto | ALU | 01010 |
| 11 | goa | ALU | 01011 |
| 12 | ifa | ALU | 01100 |
| 13 | ifan | ALU | 01101 |
| 14 | altb | ALU | 01110 |
| 15 | aeqb | ALU | 01111 |
| 16 | agtb | ALU | 10000 |
| 17 | add | ALU | 10001 |
| 18 | inca | ALU | 10010 |
| 19 | deca | ALU | 10011 |
| 20 | sub | ALU | 10100 |
| 21 | or | ALU | 10101 |
| 22 | and | ALU | 10110 |
| 23 | xor | ALU | 10111 |
| 24 | not | ALU | 11000 |
| 25 | shla | ALU | 11001 |
| 26 | shra | ALU | 11010 |
| 27 | end | Do Nothing | 11011 |

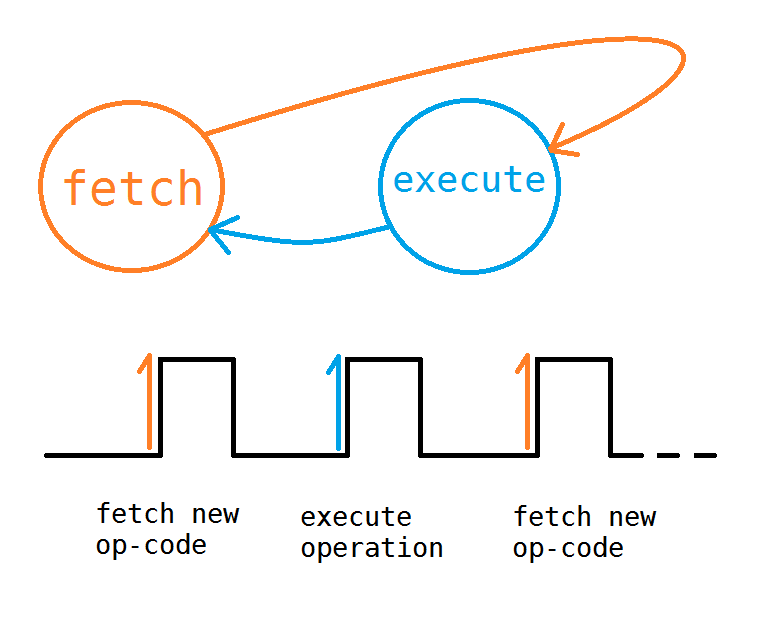


Figure 7 CPU STATE machine and timing diagram

# Luke’s Assembly Language 4.0 (LAL4)

LAL4 is the 4th assembly language designed for this project, it’s specifications are listed in below. The name column is the assembly defined names for writing in assembly, descriptions and machine codes are provided for each code. The x symbol represents the raw address or number associated with each code, the vx allows the programmer to load a static number into a register, this is not an actual operation performed by the CPU, instead the assembler converts this to an address where the static number can be stored. When jumping to different locations in the memory, the PC must be altered, the n symbol allows the programmer to specify where the PC should be set, the func word allows the programmer to specify a location within the program, the name following this is the one used in the naming convention.

Lukes Assembly Language 4.0

Memory:

Flip-Flops in CPU: a, b

SRAM: X \* 16 bit memory (X is user defined)

codes and operations

No. Operations : 28

OpCode: 5 bits XXXXX

Numerics 32-bits

Total CodeLen: 40 bits : [5-bit op-code | 32-bit num | 3-bit empty section]

(39-35) (34-3) (2-0)

OPCODE Name Usage description

Others:

00000 # #blah blah comment

00001 func func name function naming

11011 end end stop execution

Control/Memory:

00010 lda lda x load memory[x] into a

lda vx load value x into a

00011 ldb ldb x load memory[x] into b

ldb vx load value b into b

00100 ldafb ldafb load memory[b] into a

00101 sta sta x store a in memory[x]

00110 stb stb x store b in memory[x]

00111 staib staib store a in memory[b]

01000 setim setim store a in image.data[b]

ALU/PC:

01001 bisa bisa b = a

01010 goto goto n set PC to n

01011 goa goa set PC to a

01100 ifa ifa n if a != 0, set PC to n

01101 ifan ifan n if a is 0, set PC to n

01110 altb altb if a < b -> a = 1, else a = 0

01111 aeqb aeqb if a == b -> a = 1, else a = 0

10000 agtb agtb if a > b -> a = 1, else a = 0

10001 add add a = a+b

10010 inca inca a = a+1

10011 deca deca a = a-1

10100 sub sub a = a-b

10101 or or a = a|b

10110 and and a = a&b

10111 xor xor a = a^b

11000 not not a = !a

11001 shla shla x shift a left by x bits

11010 shra shra x shift a right by x bits

# The LAL4 Assembler & Interpreter

The LAL4 assembler and interpreter (LAL4AI) performs several tasks, it converts any LAL4 assembly program into machine code which can be run on the my microcomputer, it also converts the static functions and variables into a memory initialization file which is loaded into the main memory on start up. The interpreter can be used to simulate LAL4 programs, and even provides an output image so the programmer can see what the television screen should look like.

The LAL4AI is written in the python programming language, the entire program is shown below. Not shown is the bitmap object which simply communicates with a C program which reads and writes images to the hard drive.

class Assembler:

def \_\_init\_\_(self, finame):

fi = open(finame, 'r')

self.lines = fi.readlines()

fi.close()

self.programCounter = 0

self.memory = []

self.im = BMP()

self.im.setupFile(160, 120)

self.memoryAmount = 1000

self.stackMemCount = self.memoryAmount - 1

self.ra = 0

self.rb = 0

self.end = False

for i in range(self.memoryAmount):

self.memory.append(0)

count = 0

lut = [0, 1024, 512, 341, 256, 204, 170, 146, 128, 113, 102, 93, 85, 78, 73, 68, 64, 60, 56, 53, 51, 48, 46, 44, 42, 40, 39, 37, 36, 35, 34, 33, 32, 31, 30, 29, 28, 27, 26, 26, 25, 24, 24, 23, 23, 22, 22, 21, 21, 20, 20, 20, 19, 19, 18, 18, 18, 17, 17, 17, 17, 16, 16, 16, 16, 15, 15, 15, 15, 14, 14, 14, 14, 14, 13, 13, 13, 13, 13, 12, 12, 12, 12, 12, 12, 12, 11, 11, 11, 11, 11, 11, 11 , 11, 10, 10, 10, 10, 10, 10, 10, 10, 10, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2 , 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2 , 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 , 1]

for i in range(300, 941):

self.memory[i] = lut[count]

count += 1

self.setupStackMemory()

self.functionNames = []

self.functionLines = []

self.setupFunctionNames()

self.genMachineCode()

#self.printInfo()

def setupFunctionNames(self):

for i in range(len(self.lines)):

st = self.lines[i]

codes = st.split(' ')

if(codes[0] == 'func'):

self.functionNames.append(codes[1])

self.functionLines.append(int(i))

for i in range(len(self.lines)):

st = self.lines[i]

codes = st.split(' ')

if(codes[0] == 'goto' or codes[0] == 'ifa' or codes[0] == 'ifan'):

self.lines[i] = codes[0] + ' ' + str(self.findFunctionLine(codes[1], i))

if(codes[0] == 'lda' or codes[0] == 'ldb'):

if(codes[1][0] == 'n'):

funcName = codes[1]

funcName = funcName[1:]

val = int(self.findFunctionLine(funcName, 2))

naddr = self.addStackMem(val)

self.lines[i] = str(codes[0]) + ' ' + str(naddr)

def findFunctionLine(self, name, xval):

for i in range(len(self.functionNames)):

if(name == self.functionNames[i]):

return int(self.functionLines[i])

return int(xval)

def printInfo(self):

print self.lines

print 'Program Counter: ', self.programCounter

print 'functions:'

for i in range(len(self.functionNames)):

print self.functionNames[i], ' on line: ', self.functionLines[i]

print 'memory:'

#for i in range(self.memoryAmount):

for i in range(90):

print '\t', i, '\t', self.memory[i]

def setMem(self, addr, val):

self.memory[addr] = int(val)

def getMemory(self, addr):

return int(self.memory[addr])

def addStackMem(self, val):

self.memory[self.stackMemCount] = val

r = self.stackMemCount

self.stackMemCount -= 1

return r

def chomp(self, strin):

return strin.rstrip()

def setupStackMemory(self):

for i in range(len(self.lines)):

st = self.chomp(self.lines[i])

self.lines[i] = st

codes = st.split(' ')

if(codes[0] == 'lda' or codes[0] == 'ldb'):

if(codes[1][0] == 'v'):

lnch = codes[1]

lnch = lnch[1:]

val = int(lnch)

naddr = self.addStackMem(val)

self.lines[i] = str(codes[0]) + ' ' + str(naddr)

def run(self):

while(1):

self.iterateProgram()

if(self.end == True):

return

def iterateProgram(self):

st = self.lines[self.programCounter]

incit = True

codes = st.split(' ')

if(codes[0] == 'lda'):

self.ra = self.getMemory(int(codes[1]))

elif(codes[0] == 'ldb'):

self.rb = self.getMemory(int(codes[1]))

elif(codes[0] == 'ldafb'):

self.ra = self.getMemory(int(self.rb))

elif(codes[0] == 'bisa'):

self.rb = int(self.ra)

elif(codes[0] == 'sta'):

self.setMem(int(codes[1]), self.ra)

elif(codes[0] == 'stb'):

self.setMem(int(codes[1]), self.rb)

elif(codes[0] == 'staib'):

self.setMem(self.rb, self.ra)

elif(codes[0] == 'setim'):

self.im.data[self.rb] = self.ra

#self.im.saveFile('out.bmp')

elif(codes[0] == 'goto'):

self.programCounter = int(codes[1])

incit = False

elif(codes[0] == 'goa'):

self.programCounter = int(self.ra)

incit = False

elif(codes[0] == 'ifa'):

if(self.ra != 0):

self.programCounter = int(codes[1])

incit = False

elif(codes[0] == 'ifan'):

if(self.ra == 0):

self.programCounter = int(codes[1])

incit = False

elif(codes[0] == 'altb'):

if(self.ra < self.rb):

self.ra = 1

else:

self.ra = 0

elif(codes[0] == 'aeqb'):

if(self.ra == self.rb):

self.ra = 1

else:

self.ra = 0

elif(codes[0] == 'agtb'):

if(self.ra > self.rb):

self.ra = 1

else:

self.ra = 0

elif(codes[0] == 'add'):

self.ra = self.ra + self.rb

elif(codes[0] == 'inca'):

self.ra = self.ra + 1

elif(codes[0] == 'deca'):

self.ra = self.ra - 1

elif(codes[0] == 'sub'):

self.ra = self.ra - self.rb

elif(codes[0] == 'or'):

self.ra = self.ra | self.rb

elif(codes[0] == 'and'):

self.ra = self.ra & self.rb

elif(codes[0] == 'xor'):

self.ra = self.ra ^ self.rb

elif(codes[0] == 'not'):

self.ra = ~self.ra

elif(codes[0] == 'shla'):

self.ra <<= int(codes[1])

elif(codes[0] == 'shra'):

self.ra >>= int(codes[1])

elif(codes[0] == 'end'):

self.end = True

if(incit == True):

self.programCounter += 1

def printProgram(self):

for i in self.lines:

print i

def printMC(self):

for i in self.machineCode:

print i

def genMachineCode(self):

self.machineCode = []

for i in self.lines:

codes = i.split(' ')

if(codes[0][0] == '#'):

self.machineCode.append(getBin(0, 5) + zeroStrs(35))

elif(codes[0] == 'func'):

self.machineCode.append(getBin(1, 5) + zeroStrs(35))

elif(codes[0] == 'lda'):

self.machineCode.append(getBin(2,5) + getBin(codes[1], 32) + zeroStrs(3))

elif(codes[0] == 'ldb'):

self.machineCode.append(getBin(3,5) + getBin(codes[1], 32) + zeroStrs(3))

elif(codes[0] == 'ldafb'):

self.machineCode.append(getBin(4, 5) + zeroStrs(35))

elif(codes[0] == 'sta'):

self.machineCode.append(getBin(5, 5) + getBin(codes[1], 32) + zeroStrs(3))

elif(codes[0] == 'stb'):

self.machineCode.append(getBin(6, 5) + getBin(codes[1], 32) + zeroStrs(3))

elif(codes[0] == 'staib'):

self.machineCode.append(getBin(7, 5) + zeroStrs(35))

elif(codes[0] == 'setim'):

self.machineCode.append(getBin(8, 5) + zeroStrs(35))

elif(codes[0] == 'bisa'):

self.machineCode.append(getBin(9, 5) + zeroStrs(35))

elif(codes[0] == 'goto'):

self.machineCode.append(getBin(10, 5) + getBin(codes[1], 32) + zeroStrs(3))

elif(codes[0] == 'goa'):

self.machineCode.append(getBin(11, 5) + zeroStrs(35))

elif(codes[0] == 'ifa'):

self.machineCode.append(getBin(12, 5) + getBin(codes[1], 32) + zeroStrs(3))

elif(codes[0] == 'ifan'):

self.machineCode.append(getBin(13, 5) + getBin(codes[1], 32) + zeroStrs(3))

elif(codes[0] == 'altb'):

self.machineCode.append(getBin(14, 5) + zeroStrs(35))

elif(codes[0] == 'aeqb'):

self.machineCode.append(getBin(15, 5) + zeroStrs(35))

elif(codes[0] == 'agtb'):

self.machineCode.append(getBin(16, 5) + zeroStrs(35))

elif(codes[0] == 'add'):

self.machineCode.append(getBin(17, 5) + zeroStrs(35))

elif(codes[0] == 'inca'):

self.machineCode.append(getBin(18, 5) + zeroStrs(35))

elif(codes[0] == 'deca'):

self.machineCode.append(getBin(19, 5) + zeroStrs(35))

elif(codes[0] == 'sub'):

self.machineCode.append(getBin(20, 5) + zeroStrs(35))

elif(codes[0] == 'or'):

self.machineCode.append(getBin(21, 5) + zeroStrs(35))

elif(codes[0] == 'and'):

self.machineCode.append(getBin(22, 5) + zeroStrs(35))

elif(codes[0] == 'xor'):

self.machineCode.append(getBin(23, 5) + zeroStrs(35))

elif(codes[0] == 'not'):

self.machineCode.append(getBin(24, 5) + zeroStrs(35))

elif(codes[0] == 'shla'):

self.machineCode.append(getBin(25, 5) + getBin(codes[1], 32) + zeroStrs(3))

elif(codes[0] == 'shra'):

self.machineCode.append(getBin(26, 5) + getBin(codes[1], 32) + zeroStrs(3))

elif(codes[0] == 'end'):

self.machineCode.append(getBin(27, 5) + zeroStrs(35))

else:

self.machineCode.append(getBin(0, 5) + zeroStrs(35))

def saveMachineCode(self, fname):

fi = open(fname + '\_program.mif', 'w')

count = 0

fi.write('\n\nDEPTH = 1024;\n')

fi.write('WIDTH = 40;\n')

fi.write('ADDRESS\_RADIX = HEX;\n')

fi.write('DATA\_RADIX = BIN;\n')

fi.write('CONTENT\nBEGIN\n\n')

for i in self.machineCode:

fi.write(getNormalHex(count) + ' : ' + i + ';\n')

count += 1

fi.write('END;')

fi.close()

def saveProgramMemory(self, fname):

fi = open(fname + '\_memory.mif', 'w')

count = 0

fi.write('\n\nDEPTH = 1024;\n')

fi.write('WIDTH = 32;\n')

fi.write('ADDRESS\_RADIX = HEX;\n')

fi.write('DATA\_RADIX = BIN;\n')

fi.write('CONTENT\nBEGIN\n\n')

for i in self.memory:

fi.write(getNormalHex(count) + ' : ' + getBin(i, 32) + ';\n')

count += 1

fi.write('END;')

fi.close()

assemblerA = Assembler('triangleDrawerOther')

assemblerA.saveMachineCode('lineDrawer')

assemblerA.saveProgramMemory('lineDrawer')

print 'Saved Machine Code'

print 'running program::>>'

assemblerA.run()

assemblerA.printProgram()

assemblerA.printMC()

assemblerA.printInfo()

assemblerA.im.saveFile('out.bmp')

The LAL4AI sets up a look up table which is used as a function generator to generate the reciprocal of a function F(x) = pow(x, -1). It also sets up all the memory locations where static numbers are required; it does this similarly to the malloc functions present in the C programming language. Both the machine code and the initial main memory are saved in a special file format called a memory initialization file. The programmer can use the interpreter to print out the entire main memory contents and step through each operation.

# Triangles, Drawing, and Assembly Techniques

Drawing triangles requires algorithms and concepts from computer graphics, the triangle has 3 points A, B and C, the algorithm which is used in this project is the most basic, it samples the points from A to B, then draws lines from these points to C. To perform these operations, the principle of functional programming is used, where functions can perform independent operations on input and provide output, the technique used by the assembler is to specify specific areas of memory for which each function can use. A total of 13 functions were generated for use in the triangle drawing program. These functions are mapped to specific areas of memory for which their input and output can be stored. Notice all functions have one common element, they all contain a variable input called nextLocation, this specifies where the PC should return to after the function is complete. The rest of the program can then collect the output values from their memory location for further processing.

Memory Function Mapper:

divider\_Function (24,x) (25,y) (26,ans) (23,nextLocation)

getLUTVal (29,denominator) (28,output) (27,nextline)

multiplier (30,x) (31,y) (32,looper) (33,ans) (34,nextLocation)

absoluteVal (35,inp) (36,ans) (37,nextLocation)

maximumlVal (38,a) (39,b) (40,ans) (41,nextLocation)

divider\_Function2 (42,x) (43,y) (44,ans) (45,nextLocation)

setPixelF (46,x) (47,y) (48,dval) (49,nextLocation)

logicalEnd (50:x) (51:xd) (52:y) (53:yd) (54:l1) (55:l2) (56:l3) (57:ans) (58:nextLocation)

lineDrawerLoopA (73:ox) (74:oy) (75:x) (76:y) (77:incx) (78:incy) (79:l1)

(80:l2) (81:l3) (82:xd) (83:yd) (84:x1) (85:y1) (86:nextLocation)

lineDrawerFunctionPart1 (59:x1) (60:y1) (61:x2) (62:y2) (63:ox) (64:oy)

(xd:65) (yd:66) (absxd:67) (absyd:68) (maxd:69) (incx:70) (incy:71)

(72:nextlocation)

screenClear (yax:87) (88:nextLocation)

DrawTriangle (89:x1) (90:y1) (91:x2) (92:y2) (93:x3) (94:y3) (95:ox) (96:oy)

(xd:97) (yd:98) (absxd:99) (absyd:100) (maxd:101) (incx:102) (incy:103)

(104:nextlocation)

DrawTriangleLoop (105:ox) (106:oy) (107:x) (108:y) (109:incx) (110:incy) (111:l1)

(112:l2) (113:l3) (114:xd) (115:yd) (116:x1) (117:y1) (118:x3) (119:y3) (120:nextLocation)

The high level program is shown in below on the left, it simply sets the screen to black, then draws three triangles and begins again. The other two columns show the first half of the triangle drawing program

#DrawTriangle

func DrawTriangle

#input x1, y1, x2, y2, x3, y3

#uses [89 - 104]

#89:x1, 90:y1, 91:x2, 92:y2, 93:x3, 94:y3, 95:ox, 96:oy,

#xd:97, yd:98, absxd:99, absyd:100, maxd:101,

#incx:102, incy:103, 104:nextlocation

lda 89

sta 95

lda 90

sta 96

lda 91

ldb 89

sub

sta 97

lda 92

ldb 90

sub

sta 98

#now get abs of xd and yd and put it in abs(x/y)d

lda 97

sta 35

lda nafter1stABSinTriDrawer

sta 37

goto absoluteVal

func after1stABSinTriDrawer

lda 36

sta 99

lda 98

sta 35

lda nafter2ndABSinTriDrawer

sta 37

goto absoluteVal

func after2ndABSinTriDrawer

lda 36

sta 100

#calculate max(absxd, absyd)

lda 99

ldb 100

sta 38

stb 39

lda ninTriDrawerFunctionafterMaxfunc

sta 41

goto maximumVal

func inTriDrawerFunctionafterMaxfunc

lda 40

sta 101

#if maxd == 0 return

lda 101

ldb v0

aeqb

ifa exitTriDrawerP1

#shift xd left by 10

lda 97

shla 10

sta 97

lda 98

shla 10

sta 98

#do the first division function calculating incx

lda 97

sta 42

ldb 101

stb 43

lda nafterCalcIncXinTriFunc

sta 45

goto divider\_Function2

func afterCalcIncXinTriFunc

lda 44

shra 10

sta 102

#do the second division function calculating incx

lda 98

sta 42

ldb 101

stb 43

lda nafterCalcIncYInTriDrawA

sta 45

goto divider\_Function2

func afterCalcIncYInTriDrawA

lda 44

shra 10

sta 103

lda 97

shra 10

sta 97

lda 98

shra 10

sta 98

lda v0

sta 89

sta 90

#now pass to the looper function

lda nexitTriDrawerP1

sta 120

lda 95

sta 105

lda 96

sta 106

lda 102

sta 109

lda 103

sta 110

lda 97

sta 114

lda 98

sta 115

lda 89

ldb 90

sta 116

stb 117

lda 93

ldb 94

sta 118

stb 119

goto DrawTriangleLoop

func exitTriDrawerP1

lda 104

goa

end

func restart

lda v0

sta 48

lda nafterTheScCLR

sta 88

goto screenClear

func afterTheScCLR

lda v255

sta 48

lda nafter2ndTriangleDrawingIsComplete

sta 104

lda v58

ldb v29

sta 89

stb 90

lda v71

ldb v57

sta 91

stb 92

lda v120

ldb v67

sta 93

stb 94

goto DrawTriangle

func after2ndTriangleDrawingIsComplete

lda v66

sta 48

lda nafter3TriangleDrawingIsComplete

sta 104

lda v81

ldb v31

sta 89

stb 90

lda v84

ldb v55

sta 91

stb 92

lda v124

ldb v59

sta 93

stb 94

goto DrawTriangle

func after3TriangleDrawingIsComplete

lda v138

sta 48

lda nafter4TriangleDrawingIsComplete

sta 104

lda v14

ldb v60

sta 89

stb 90

lda v14

ldb v96

sta 91

stb 92

lda v141

ldb v98

sta 93

stb 94

goto DrawTriangle

func after4TriangleDrawingIsComplete

goto restart

end

The program below in the left column allows the 2nd half of the triangle drawing program. The right column shows the code to clear the screen.

#TriangleDrawerLoopA

func DrawTriangleLoop

#uses [105 - 120]

#input x1, y1, ox, oy, incx, incy, xd, yd

#105:ox, 106:oy, 107:x, 108:y, 109:incx, 110:incy, 111:l1, 112:l2, 113:l3, 114:xd,

#115:yd, 116:x1, 117:y1, 118:x3, 119:y3, 120:nextLocation

func TriloopRestartlineDrawerLoopA

lda 116

shra 10

sta 107

lda 117

shra 10

sta 108

lda 107

ldb 105

add

sta 107

lda 108

ldb 106

add

sta 108

lda nAfterLineDrawinTriDrawerLoopAR

sta 72

lda 107

sta 59

lda 108

sta 60

lda 118

sta 61

lda 119

sta 62

goto lineDrawerFunctionPart1

func AfterLineDrawinTriDrawerLoopAR

lda 116

ldb 109

add

sta 116

lda 117

ldb 110

add

sta 117

lda nafterLogicalEndinTriDrawerLoopA

sta 58

lda 107

ldb 105

sub

sta 50

lda 114

sta 51

lda 108

ldb 106

sub

sta 52

lda 115

sta 53

goto logicalEnd

func afterLogicalEndinTriDrawerLoopA

lda 57

ifa exitTriDrawerLoopA

goto TriloopRestartlineDrawerLoopA

func exitTriDrawerLoopA

lda 120

goa

end

#screen clear function

func screenClear

#yax:87, nextLocation:88

lda v0

sta 87

func screenClearLineLooper

lda v0

sta 59

lda v160

sta 61

lda 87

sta 60

sta 62

lda nafterLineDrawinClearScreenLooperddf

sta 72

goto lineDrawerFunctionPart1

func afterLineDrawinClearScreenLooperddf

lda 87

inca

sta 87

lda 87

ldb v120

altb

ifa screenClearLineLooper

lda 88

goa

end

The first two columns below show the first half of the line drawing program, the right column shows the second half

#lineDrawerLoopA

func lineDrawerLoopA

#uses []

#input x1, y1, ox, oy, incx, incy, xd, yd

#73:ox, 74:oy, 75:x, 76:y, 77:incx, 78:incy, 79:l1, 80:l2, 81:l3, 82:xd,

#83:yd, 84:x1, 85:y1, 86:nextLocation

func loopRestartlineDrawerLoopA

lda 84

shra 10

sta 75

lda 85

shra 10

sta 76

lda 75

ldb 73

add

sta 75

lda 76

ldb 74

add

sta 76

lda nAfterPixelSetinlineDrawerLoopA

sta 49

lda 75

sta 46

lda 76

sta 47

goto setPixelF

func AfterPixelSetinlineDrawerLoopA

lda 84

ldb 77

add

sta 84

lda 85

ldb 78

add

sta 85

lda nafterLogicalEndinDrawerLoopA

sta 58

lda 75

ldb 73

sub

sta 50

lda 82

sta 51

lda 76

ldb 74

sub

sta 52

lda 83

sta 53

goto logicalEnd

func afterLogicalEndinDrawerLoopA

lda 57

ifa exitDrawerLoopA

goto loopRestartlineDrawerLoopA

func exitDrawerLoopA

lda 86

goa

end

#do the first division function calculating incx

lda 65

sta 42

ldb 69

stb 43

lda nafterCalcIncX

sta 45

goto divider\_Function2

func afterCalcIncX

lda 44

shra 10

sta 70

#do the second division function calculating incx

lda 66

sta 42

ldb 69

stb 43

lda nafterCalcIncY

sta 45

goto divider\_Function2

func afterCalcIncY

lda 44

shra 10

sta 71

lda 65

shra 10

sta 65

lda 66

shra 10

sta 66

lda v0

sta 59

sta 60

#now pass to the looper function

lda nexitLineDrawerP1

sta 86

lda 63

sta 73

lda 64

sta 74

lda 70

sta 77

lda 71

sta 78

lda 65

sta 82

lda 66

sta 83

lda 59

ldb 60

sta 84

stb 85

goto lineDrawerLoopA

func exitLineDrawerP1

lda 72

goa

end

#lineDrawerFunctionPart1

func lineDrawerFunctionPart1

#input x1, y1, x2, y2

#uses [59 - 72]

#59:x1, 60:y1, 61:x2, 62:y2, 63:ox, 64:oy,

#xd:65, yd:66, absxd:67, absyd:68, maxd:69,

#incx:70, incy:71, 72:nextlocation

lda 59

sta 63

lda 60

sta 64

lda 61

ldb 59

sub

sta 65

lda 62

ldb 60

sub

sta 66

#now get abs of xd and yd and put it in abs(x/y)d

lda 65

sta 35

lda nafter1stABSinlineDrawerFunctionPart1

sta 37

goto absoluteVal

func after1stABSinlineDrawerFunctionPart1

lda 36

sta 67

lda 66

sta 35

lda nafter2ndABSinlineDrawerFunctionPart1

sta 37

goto absoluteVal

func after2ndABSinlineDrawerFunctionPart1

lda 36

sta 68

#calculate max(absxd, absyd)

lda 67

ldb 68

sta 38

stb 39

lda ninDrawerFunctionPart1afterMaxfunc

sta 41

goto maximumVal

func inDrawerFunctionPart1afterMaxfunc

lda 40

sta 69

#if maxd == 0 return

lda 69

ldb v0

aeqb

ifa exitLineDrawerP1

#shift xd left by 10

lda 65

shla 10

sta 65

lda 66

shla 10

sta 66

The line drawer and triangle drawing functions use a special logical function to calculate when the line drawing should end (otherwise the line would continue indefinitely). This function is shown in the left column. Two division functions are used, one outputs a floating point number and another outputs an integer. Both the division functions use a look up tables as the reciprocal function generator. The function to retrieve a lookup table output is shown in the bottom box.

#grab the divisor from LUT

func getLUTVal

#memory used [27 - 29]

# 29: denomenator, 28: outputvalue, 27: next line

#get memory location of memLocation, store it in output value

lda 29

ldb v300

add

bisa

ldafb

sta 28

lda 27

goa

end

func divider\_Function2

#(x / y)

# uses memory [42 - 45]

# 42:x, 43:y, 44:ans, 45:next location

lda 42

shla 10

sta 42

ldb 43

stb 29

lda naftergetLUTinDividerF2

sta 27

goto getLUTVal

func aftergetLUTinDividerF2

lda 28

sta 31

lda 42

sta 30

lda naftermultiplierDividerFuncF2

sta 34

goto multiplier

func aftermultiplierDividerFuncF2

lda 33

shra 10

sta 44

lda 45

goa

end

#division function

func divider\_Function

#(x / y)

# uses memory [23 - 26]

# 24 : x, 25 : y, 26 : ans, 23 : next location

lda 24

shla 10

sta 24

ldb 25

stb 29

lda naftergetLUTinDivider

sta 27

goto getLUTVal

func aftergetLUTinDivider

lda 28

sta 31

lda 24

sta 30

lda naftermultiplierDividerFunc

sta 34

goto multiplier

func aftermultiplierDividerFunc

lda 33

shra 20

sta 26

lda 23

goa

end

#logicalEndFunction

func logicalEnd

#uses memory [50 - 54]

INPUT is x, xd, y, y2

#50:x, 51:xd, 52:y, 53:yd, 54:l1, 55:l2, 56:l3, 57:ans, 58:nextLocation

lda 51

ldb 50

sub

sta 35

lda nafterABS1LogicalEndFunc

sta 37

goto absoluteVal

func afterABS1LogicalEndFunc

lda 36

sta 54

lda 53

ldb 52

sub

sta 35

lda nafterABS2LogicalEndFunc

sta 37

goto absoluteVal

func afterABS2LogicalEndFunc

lda 36

sta 55

lda 54

ldb 55

add

ldb v4

altb

sta 57

lda 58

goa

end

The last functions required are the multiplier function, the absolute function and the division function, these are shown below:

func maximumVal

# returns max(a,b)

#uses memory [38 - ]

# 38:a, 39:b, 40:ans, 41:nextLocation

lda 38

ldb 39

agtb

ifa amorethanb

stb 40

lda 41

goa

func amorethanb

lda 38

sta 40

lda 41

goa

end

func absoluteVal

#uses memory [35 - 37]

#35:inp, 36:ans, 37:nextLocation

lda 35

ldb v0

altb

ifa continueWithAbsValFunc

lda 35

sta 36

lda 37

goa

func continueWithAbsValFunc

lda 35

not

inca

sta 36

lda 37

goa

end

func multiplier

#uses memory [30 - 34]

#30:x, 31:y, 32:looper, 33:ans, 34:nextLocation

lda v0

sta 32

sta 33

ldb 31

aeqb

ifan loop\_in\_multiplier

goto end\_multiplier

func loop\_in\_multiplier

#add em

lda 33

ldb 30

add

sta 33

#increment loop i

lda 32

inca

sta 32

ldb 31

altb

ifa loop\_in\_multiplier

func end\_multiplier

lda 34

goa

end

A total of 637 lines of assembly are required to draw three triangles, refresh the frame buffer and repeat the process.

# FPGA implementation

The DE1 board was used to implement these designs. Using the AHDL hardware description language and the Quartus II integrated development environment, these designs were implemented with success. The AHDL modules used are provided below, beginning with the ones for the VGA driver.

## The Image Indexer

subdesign GetImageIndex(

xPoint[9..0] :INPUT;

yPoint[9..0] :INPUT;

outIndex[14..0] :OUTPUT;

)

variable

xAx[14..0] : NODE;

yAx[14..0] : NODE;

yAxAddSh5[14..0] : NODE;

yAxAddSh7[14..0] : NODE;

yadd[14..0] : NODE;

begin

xAx[14..0] = (GND,GND, GND,GND,GND, GND, GND, xPoint[9..2]);

yAx[14..0] = (GND,GND, GND,GND,GND, GND, GND, yPoint[9..2]);

yAxAddSh5[14..0] = (yAx[9..0], GND, GND, GND, GND, GND);

yAxAddSh7[14..0] = (yAx[7..0], GND, GND, GND, GND, GND, GND, GND);

yadd[14..0] = yAxAddSh5[14..0] + yAxAddSh7[14..0];

outIndex[14..0] = xAx[14..0] + yadd[14..0];

end;

# 

## The Counter used in the VGA driver

SUBDESIGN AVGACOUNTER(

clk\_in :INPUT;

reset :INPUT;

set\_maximum :INPUT;

max\_num[18..0] :INPUT;

count\_out[18..0] :OUTPUT;

)

VARIABLE

ffs[18..0] : JKFF;

BEGIN

ffs[].clk = clk\_in;

IF reset == B"1" THEN

ffs[].j = (0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0);

ffs[].k = (1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1);

ELSIF set\_maximum == B"1" THEN

IF ffs[].q >= max\_num[] THEN

ffs[].j = (0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0);

ffs[].k = (1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1);

ELSE

ffs[].j = ffs[].q + 1;

ffs[].k = !(ffs[].q + 1);

END IF;

ELSE

ffs[].j = ffs[].q + 1;

ffs[].k = !(ffs[].q + 1);

END IF;

count\_out[] = ffs[].q;

END;

## VGA Signal Generator Module

INCLUDE "AVGACOUNTER.inc";

SUBDESIGN AVGASYNCMOD(

clk25\_in : INPUT;

x\_pos[18..0] : OUTPUT;

y\_pos[18..0] : OUTPUT;

av\_sync, ah\_sync :OUTPUT;

send\_data :OUTPUT;

)

VARIABLE

syncCounterV :AVGACOUNTER;

syncCounterH :AVGACOUNTER;

data\_out :NODE;

BEGIN

IF syncCounterH.count\_out[] == 639 THEN

syncCounterV.clk\_in = B"1";

ELSE

syncCounterV.clk\_in = B"0";

END IF;

syncCounterV.reset = B"0";

syncCounterV.max\_num[] = 520;

syncCounterV.set\_maximum = B"1";

syncCounterH.clk\_in = clk25\_in;

syncCounterH.reset = B"0";

syncCounterH.max\_num[] = 799;

syncCounterH.set\_maximum = B"1";

data\_out = (syncCounterV.count\_out[] >= 41 AND syncCounterV.count\_out[] < 521 AND syncCounterH.count\_out[] >= 160 AND syncCounterH.count\_out[] < 800);

av\_sync = (syncCounterV.count\_out[] >= 10 AND syncCounterV.count\_out[] < 12);

ah\_sync = (syncCounterH.count\_out[] >= 16 AND syncCounterH.count\_out[] < 112);

send\_data = data\_out;

IF data\_out == B"1" THEN

x\_pos[] = syncCounterH.count\_out[] - 160;

y\_pos[] = syncCounterV.count\_out[] - 41;

ELSE

x\_pos[] = 0;

y\_pos[] = 0;

END IF;

END;

# 

## ALU Module Part 1

subdesign ALU\_PC(

OPCODE[5..0] :INPUT;

INVAL[31..0] :INPUT;

RA\_IN[31..0] :INPUT;

RB\_IN[31..0] :INPUT;

PC\_IN[31..0] :INPUT;

RA\_OUT[31..0] :OUTPUT;

RB\_OUT[31..0] :OUTPUT;

PC\_OUT[31..0] :OUTPUT;

)

begin

if OPCODE[] == 9 then -- bisa

RB\_OUT[31..0] = RA\_IN[31..0];

RA\_OUT[31..0] = RA\_IN[31..0];

PC\_OUT[31..0] = PC\_IN[31..0] + 1;

elsif OPCODE[] == 10 then --goto

PC\_OUT[31..0] = INVAL[31..0];

RA\_OUT[31..0] = RA\_IN[31..0];

RB\_OUT[31..0] = RB\_IN[31..0];

elsif OPCODE[] == 11 then --goa

PC\_OUT[31..0] = RA\_IN[31..0];

RA\_OUT[31..0] = RA\_IN[31..0];

RB\_OUT[31..0] = RB\_IN[31..0];

elsif OPCODE[] == 12 and RA\_IN[31..0] != 0 then --ifa

PC\_OUT[31..0] = INVAL[31..0];

RA\_OUT[31..0] = RA\_IN[31..0];

RB\_OUT[31..0] = RB\_IN[31..0];

elsif OPCODE[] == 13 and RA\_IN[31..0] == 0 then -- ifan

PC\_OUT[31..0] = INVAL[31..0];

RA\_OUT[31..0] = RA\_IN[31..0];

RB\_OUT[31..0] = RB\_IN[31..0];

elsif OPCODE[] == 14 then --altb

RB\_OUT[31..0] = RB\_IN[31..0];

PC\_OUT[31..0] = PC\_IN[31..0] + 1;

if RA\_IN[31..0] < RB\_IN[31..0] then

RA\_OUT[31..0] = 1;

else

RA\_OUT[31..0] = 0;

end if;

elsif OPCODE[] == 15 then -- aeqb

RB\_OUT[31..0] = RB\_IN[31..0];

PC\_OUT[31..0] = PC\_IN[31..0] + 1;

if RA\_IN[31..0] == RB\_IN[31..0] then

RA\_OUT[31..0] = 1;

else

RA\_OUT[31..0] = 0;

end if;

elsif OPCODE[] == 16 then --agtb

PC\_OUT[31..0] = PC\_IN[31..0] + 1;

RB\_OUT[31..0] = RB\_IN[31..0];

if RA\_IN[31..0] > RB\_IN[31..0] then

RA\_OUT[31..0] = 1;

else

RA\_OUT[31..0] = 0;

end if;

elsif OPCODE[] == 17 then --add

PC\_OUT[31..0] = PC\_IN[31..0] + 1;

RB\_OUT[31..0] = RB\_IN[31..0];

RA\_OUT[31..0] = RA\_IN[31..0] + RB\_IN[31..0];

elsif OPCODE[] == 18 then --inca

PC\_OUT[31..0] = PC\_IN[31..0] + 1;

RA\_OUT[31..0] = RA\_IN[31..0] + 1;

RB\_OUT[31..0] = RB\_IN[31..0];

elsif OPCODE[] == 19 then --deca

PC\_OUT[31..0] = PC\_IN[31..0] + 1;

RA\_OUT[31..0] = RA\_IN[31..0] - 1;

RB\_OUT[31..0] = RB\_IN[31..0];

elsif OPCODE[] == 20 then --sub

PC\_OUT[31..0] = PC\_IN[31..0] + 1;

RA\_OUT[31..0] = RA\_IN[31..0] - RB\_IN[31..0];

RB\_OUT[31..0] = RB\_IN[31..0];

elsif OPCODE[] == 21 then --or

PC\_OUT[31..0] = PC\_IN[31..0] + 1;

RA\_OUT[31..0] = RA\_IN[31..0] # RB\_IN[31..0];

RB\_OUT[31..0] = RB\_IN[31..0];

## ALU Module Part 2

elsif OPCODE[] == 22 then -- and

PC\_OUT[31..0] = PC\_IN[31..0] + 1;

RA\_OUT[31..0] = RA\_IN[31..0] & RB\_IN[31..0];

RB\_OUT[31..0] = RB\_IN[31..0];

elsif OPCODE[] == 23 then --xor

PC\_OUT[31..0] = PC\_IN[31..0] + 1;

RA\_OUT[31..0] = RA\_IN[31..0] $ RB\_IN[31..0];

RB\_OUT[31..0] = RB\_IN[31..0];

elsif OPCODE[] == 24 then --not

PC\_OUT[31..0] = PC\_IN[31..0] + 1;

RA\_OUT[31..0] = RA\_IN[31..0] $ (GND,GND,GND,GND,GND,GND,GND,GND,GND,GND,GND,GND,GND,GND,GND,GND,GND,GND,GND,GND,GND,GND,GND,GND,GND,GND,GND,GND,GND,GND,GND,GND);

RB\_OUT[31..0] = RB\_IN[31..0];

elsif OPCODE[] == 25 then --shla

PC\_OUT[31..0] = PC\_IN[31..0] + 1;

RB\_OUT[31..0] = RB\_IN[31..0];

if INVAL[31..0] == 0 then

RA\_OUT[31..0] = RA\_IN[31..0];

elsif INVAL[31..0] == 1 then

RA\_OUT[31..0] = (RA\_IN[30..0], GND);

elsif INVAL[31..0] == 2 then

RA\_OUT[31..0] = (RA\_IN[29..0], GND, GND);

elsif INVAL[31..0] == 3 then

RA\_OUT[31..0] = (RA\_IN[28..0], GND, GND, GND);

elsif INVAL[31..0] == 4 then

RA\_OUT[31..0] = (RA\_IN[27..0], GND, GND, GND, GND);

elsif INVAL[31..0] == 5 then

RA\_OUT[31..0] = (RA\_IN[26..0], GND, GND, GND, GND, GND);

elsif INVAL[31..0] == 6 then

RA\_OUT[31..0] = (RA\_IN[25..0], GND, GND, GND, GND, GND, GND);

elsif INVAL[31..0] == 7 then

RA\_OUT[31..0] = (RA\_IN[24..0], GND, GND, GND, GND, GND, GND, GND);

elsif INVAL[31..0] == 8 then

RA\_OUT[31..0] = (RA\_IN[23..0], GND, GND, GND, GND, GND, GND, GND, GND);

elsif INVAL[31..0] == 9 then

RA\_OUT[31..0] = (RA\_IN[22..0], GND, GND, GND, GND, GND, GND, GND, GND, GND);

elsif INVAL[31..0] == 10 then

RA\_OUT[31..0] = (RA\_IN[21..0], GND, GND, GND, GND, GND, GND, GND, GND, GND, GND);

elsif INVAL[31..0] == 11 then

RA\_OUT[31..0] = (RA\_IN[20..0], GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND);

elsif INVAL[31..0] == 12 then

RA\_OUT[31..0] = (RA\_IN[19..0], GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND);

elsif INVAL[31..0] == 13 then

RA\_OUT[31..0] = (RA\_IN[18..0], GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND);

elsif INVAL[31..0] == 14 then

RA\_OUT[31..0] = (RA\_IN[17..0], GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND);

elsif INVAL[31..0] == 15 then

RA\_OUT[31..0] = (RA\_IN[16..0], GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND);

elsif INVAL[31..0] == 16 then

RA\_OUT[31..0] = (RA\_IN[15..0], GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND);

elsif INVAL[31..0] == 17 then

RA\_OUT[31..0] = (RA\_IN[14..0], GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND);

elsif INVAL[31..0] == 18 then

RA\_OUT[31..0] = (RA\_IN[13..0], GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND);

elsif INVAL[31..0] == 19 then

RA\_OUT[31..0] = (RA\_IN[12..0], GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND);

elsif INVAL[31..0] == 20 then

RA\_OUT[31..0] = (RA\_IN[11..0], GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND);

elsif INVAL[31..0] == 21 then

RA\_OUT[31..0] = (RA\_IN[10..0], GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND);

end if;

## ALU Module Part 3

elsif OPCODE[] == 26 then --shra

RB\_OUT[31..0] = RB\_IN[31..0];

PC\_OUT[31..0] = PC\_IN[31..0] + 1;

if INVAL[31..0] == 0 then

RA\_OUT[31..0] = RA\_IN[31..0];

elsif INVAL[31..0] == 1 then

RA\_OUT[31..0] = (GND, RA\_IN[31..1]);

elsif INVAL[31..0] == 2 then

RA\_OUT[31..0] = (GND, GND, RA\_IN[31..2]);

elsif INVAL[31..0] == 3 then

RA\_OUT[31..0] = (GND, GND, GND, RA\_IN[31..3]);

elsif INVAL[31..0] == 4 then

RA\_OUT[31..0] = (GND, GND, GND, GND, RA\_IN[31..4]);

elsif INVAL[31..0] == 5 then

RA\_OUT[31..0] = (GND, GND, GND, GND, GND, RA\_IN[31..5]);

elsif INVAL[31..0] == 6 then

RA\_OUT[31..0] = (GND, GND, GND, GND, GND, GND, RA\_IN[31..6]);

elsif INVAL[31..0] == 7 then

RA\_OUT[31..0] = (GND, GND, GND, GND, GND, GND, GND, RA\_IN[31..7]);

elsif INVAL[31..0] == 8 then

RA\_OUT[31..0] = (GND, GND, GND, GND, GND, GND, GND, GND, RA\_IN[31..8]);

elsif INVAL[31..0] == 9 then

RA\_OUT[31..0] = (GND, GND, GND, GND, GND, GND, GND, GND, GND, RA\_IN[31..9]);

elsif INVAL[31..0] == 10 then

RA\_OUT[31..0] = (GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, RA\_IN[31..10]);

elsif INVAL[31..0] == 11 then

RA\_OUT[31..0] = (GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, RA\_IN[31..11]);

elsif INVAL[31..0] == 12 then

RA\_OUT[31..0] = (GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, RA\_IN[31..12]);

elsif INVAL[31..0] == 13 then

RA\_OUT[31..0] = (GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, RA\_IN[31..13]);

elsif INVAL[31..0] == 14 then

RA\_OUT[31..0] = (GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, RA\_IN[31..14]);

elsif INVAL[31..0] == 15 then

RA\_OUT[31..0] = (GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, RA\_IN[31..15]);

elsif INVAL[31..0] == 16 then

RA\_OUT[31..0] = (GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, RA\_IN[31..16]);

elsif INVAL[31..0] == 17 then

RA\_OUT[31..0] = (GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, RA\_IN[31..17]);

elsif INVAL[31..0] == 18 then

RA\_OUT[31..0] = (GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, RA\_IN[31..18]);

elsif INVAL[31..0] == 19 then

RA\_OUT[31..0] = (GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, RA\_IN[31..19]);

elsif INVAL[31..0] == 20 then

RA\_OUT[31..0] = (GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, RA\_IN[31..20]);

elsif INVAL[31..0] == 21 then

RA\_OUT[31..0] = (GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, GND, RA\_IN[31..21]);

end if;

else

RA\_OUT[31..0] = RA\_IN[31..0];

RB\_OUT[31..0] = RB\_IN[31..0];

PC\_OUT[31..0] = PC\_IN[31..0] + 1;

end if;

end;

## CPU & VGA Driver Part 1

Include "AVGASYNCMOD.inc";

Include "GetImageIndex.inc";

Include "myRam.inc";

Include "ONEHZCLOCK.inc";

Include "TheProgram.inc";

Include "TheMemory.inc";

Include "ALU\_PC.inc";

subdesign MyProjA(

CLOCK\_50 :INPUT;

VGA\_HS, VGA\_VS :OUTPUT;

VGA\_R[3..0] :OUTPUT;

VGA\_G[3..0] :OUTPUT;

VGA\_B[3..0] :OUTPUT;

LEDR[1..0] :OUTPUT;

)

variable

VgaController :AVGASYNCMOD;

jkff25 :JKFF;

ImageIndexer :GetImageIndex;

imageMemory : myRam;

keyPressed : NODE;

PC[31..0] : DFF;

RA[31..0] : DFF;

RB[31..0] : DFF;

DATAIN[31..0] :DFF;

OPCODE[5..0] :DFF;

State[1..0] :DFF;

mem : TheProgram;

prog : TheMemory;

alu : ALU\_PC;

ohc : ONEHZCLOCK;

begin

defaults

mem.address[9..0] = 0;

mem.data[31..0] = 0;

mem.wren = GND;

imageMemory.address\_a[14..0] = 0;

imageMemory.wren\_a = GND;

imageMemory.data\_a[3..0] = 0;

end defaults;

--cpu clock, only for this

ohc.CLKIN50 = CLOCK\_50;

--

LEDR[1..0] = State[1..0].q;

--Preliminary Setup:>

imageMemory.inclock = CLOCK\_50;

keyPressed = ohc.OUTCLK;

State[1..0].clk = keyPressed;

OPCODE[5..0].clk = keyPressed;

DATAIN[31..0].clk = keyPressed;

PC[31..0].clk = keyPressed;

RA[31..0].clk = keyPressed;

RB[31..0].clk = keyPressed;

prog.wren = GND;

prog.data[39..0] = 0;

## CPU & VGA Driver Part 2

prog.clock = CLOCK\_50;

prog.address[9..0] = PC[9..0].q;

mem.clock = CLOCK\_50;

alu.OPCODE[5..0] = OPCODE[5..0].q;

alu.INVAL[31..0] = DATAIN[31..0].q;

alu.RA\_IN[31..0] = RA[31..0].q;

alu.RB\_IN[31..0] = RB[31..0].q;

alu.PC\_IN[31..0] = PC[31..0].q;

--

--actual program execution

if State[1..0] == 0 then

RA[31..0].d = RA[31..0].q;

RB[31..0].d = RB[31..0].q;

PC[31..0].d = PC[31..0].q;

DATAIN[31..0].d = prog.q[34..3];

OPCODE[5..0].d = (GND, prog.q[39..35]);

State[1..0].d = 1;

elsif State[1..0] == 1 then

OPCODE[5..0].d = OPCODE[5..0].q;

DATAIN[31..0].d = DATAIN[31..0].q;

if OPCODE[5..0] == 27 then

State[1..0].d = 2;

RA[31..0].d = RA[31..0].q;

RB[31..0].d = RB[31..0].q;

PC[31..0].d = PC[31..0].q;

elsif OPCODE[5..0] >= 0 and OPCODE[5..0] < 9 then

--a control / memory operation

State[1..0].d = 0;

PC[31..0].d = PC[31..0].q + 1;

if OPCODE[5..0] == 0 or OPCODE[5..0] == 1 then

RA[31..0].d = RA[31..0].q;

RB[31..0].d = RB[31..0].q;

elsif OPCODE[5..0] == 2 then --lda

mem.address[9..0] = DATAIN[9..0].q;

RA[31..0].d = mem.q[31..0];

RB[31..0].d = RB[31..0].q;

elsif OPCODE[5..0] == 3 then --ldb

mem.address[9..0] = DATAIN[9..0].q;

RA[31..0].d = RA[31..0].q;

RB[31..0].d = mem.q[31..0];

elsif OPCODE[5..0] == 4 then --ldafb

mem.address[9..0] = RB[9..0].q;

RA[31..0].d = mem.q[31..0];

RB[31..0].d = RB[31..0].q;

elsif OPCODE[5..0] == 5 then --sta

mem.wren = VCC;

mem.address[9..0] = DATAIN[9..0].q;

mem.data[31..0] = RA[31..0].q;

RA[31..0].d = RA[31..0].q;

RB[31..0].d = RB[31..0].q;

elsif OPCODE[5..0] == 6 then --stb

mem.wren = VCC;

mem.address[9..0] = DATAIN[9..0].q;

mem.data[31..0] = RB[31..0].q;

RA[31..0].d = RA[31..0].q;

RB[31..0].d = RB[31..0].q;

elsif OPCODE[5..0] == 7 then --staib

mem.wren = VCC;

mem.address[9..0] = RB[9..0].q;

mem.data[31..0] = RA[31..0].q;

RA[31..0].d = RA[31..0].q;

RB[31..0].d = RB[31..0].q;

# 

## CPU & VGA Driver Part 3

elsif OPCODE[5..0] == 8 then --setim

RA[31..0].d = RA[31..0].q;

RB[31..0].d = RB[31..0].q;

imageMemory.wren\_a = VCC;

imageMemory.address\_a[14..0] = RB[14..0].q;

imageMemory.data\_a[3..0] = RA[3..0].q;

--other memory things

end if;

else

--an ALU or PC operation

State[1..0].d = 0;

RA[31..0].d = alu.RA\_OUT[31..0];

RB[31..0].d = alu.RB\_OUT[31..0];

PC[31..0].d = alu.PC\_OUT[31..0];

end if;

else

RA[31..0].d = RA[31..0].q;

RB[31..0].d = RB[31..0].q;

PC[31..0].d = PC[31..0].q;

DATAIN[31..0].d = DATAIN[31..0].q;

OPCODE[5..0].d = OPCODE[5..0].q;

State[1..0].d = 0;

end if;

--end program execution

--VGA DRIVER

jkff25.j = VCC;

jkff25.k = VCC;

jkff25.clk = CLOCK\_50;

VgaController.clk25\_in = jkff25.q;

VGA\_HS = !VgaController.ah\_sync;

VGA\_VS = !VgaController.av\_sync;

--sending data:

ImageIndexer.xPoint[9..0] = VgaController.x\_pos[9..0];

ImageIndexer.yPoint[9..0] = VgaController.y\_pos[9..0];

--end sending data

--image memory indexing:

imageMemory.address\_b[14..0] = imageIndexer.outIndex[14..0];

imageMemory.outclock = !CLOCK\_50;

imageMemory.wren\_b = GND;

imageMemory.data\_b[3..0] = 0;

IF VgaController.send\_data == VCC THEN

VGA\_R[3..0] = 0;

VGA\_G[3..0] = imageMemory.q\_b[3..0];

VGA\_B[3..0] = imageMemory.q\_b[3..0];

ELSE

VGA\_R[3..0] = 0;

VGA\_G[3..0] = 0;

VGA\_B[3..0] = 0;

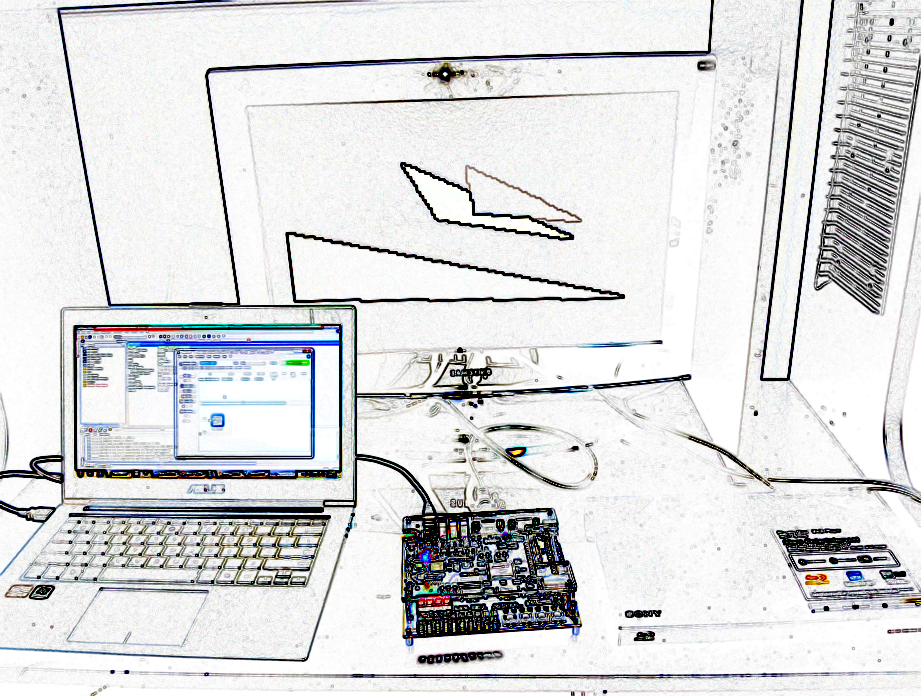
END IF;

--END VGA Driver

end;

# Final Project Outcome

The project works successfully, below are some pictures showing it in use.





YouTube videos are can also be views in these links: <http://www.youtube.com/watch?v=taQ5eouGpJs>, <http://www.youtube.com/watch?v=v-oTsPKXnSg>