UNIVERSITY OF CALIFORNIA AT BERKELEY
COLLEGE OF ENGINEERING
DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Fall 2003 Project
Checkpoint 2, Video Encoder

1. Motivation

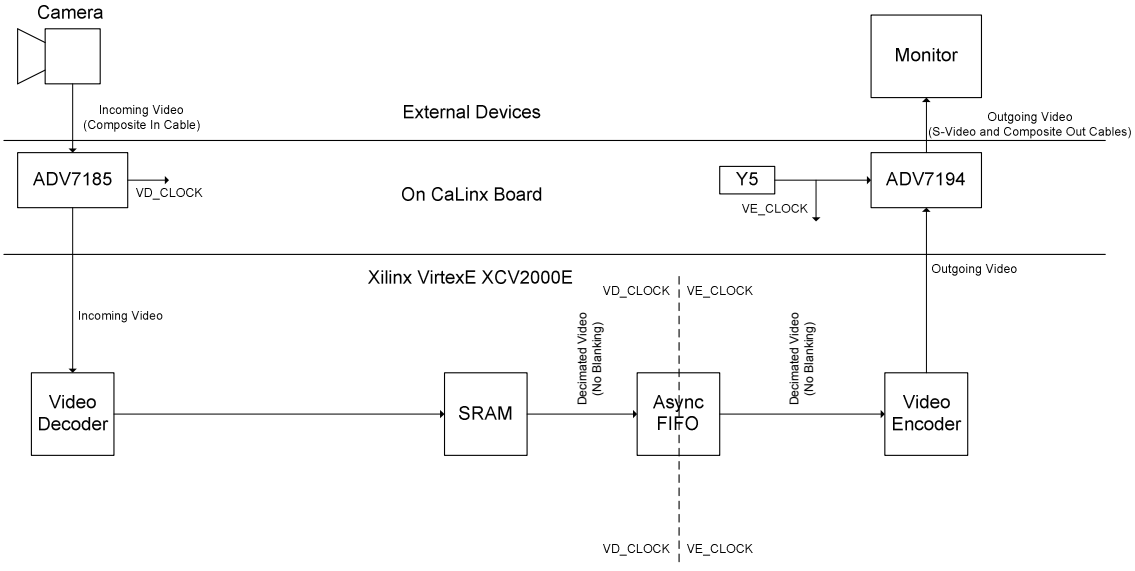
This is the second checkpoint for the Fall 2003 project.

- You will familiarize yourself with digital video encoding and decoding standards
- You will familiarize yourself with NTSC standard
- You will become familiar with Xilinx's internal BlockRam
- You will use an asynchronous FIFO

2. Objective

Use an asynchronous FIFO and a small piece of SRAM to take a single line of video from the ADV7185 decoder and display it repeatedly on the monitor using the ADV7194.

3. Methodology



Given Modules:

- SRAM module, "**linememory**" which can hold a single line of 320 pixels in full color
- An asynchronous FIFO "**async_fifo**" with 32bit I/O for use in crossing clock domains
- A Video Decoder "**vid_dec**" module which will decode video from the ADV7185 and output valid 32bit blocks of data and separate sync signals.

- A skeleton **FPGA_TOP**, you must add to this module
- A skeleton Video Encoder, "**vid_enc**" which will initialize the ADV7194, you must add the main functionality

Using these modules you will continuously read the first line of active video from the camera into the line memory. You will also continuously read data from the line memory into the aynchronous FIFO which will make it possible to cross from the VD_CLOCK domain to the VE_CLOCK domain. Your video encoder must read data from this asynchronous FIFO and display it on EVERY line of the outgoing. Thus the first line of video from the camera will be replicated to every line of video on the monitor. This should result in recognizable vertical bars on the monitor which should change as you wave in front of the camera.

3.1 Video Encoder

Most of the work for this checkpoint will be in creating the video encoder module. The ports you will be concerned with are:

```
output [9:0]  P
output        PAL_NTSC;
output        RST_OUT_;
output        HSYNC_;
output        VSYNC_;
output        BLANK_;
output        SCRESET;
input         CLK;
```

CLK will be VE_CLOCK will be the 27MHz clock from the encoder, which will be provided to you to work with. P is the 10bits output on which you have to send your video and control data in the upper 8. It doesn't matter what you send in the lowest 2 bits. Ignore them.

Name	Description
P	10 bits for 8-bit parallel ITU 601/656 stream
PAL_NTSC	keep it to 0 for NTSC transmissions
RST_OUT_	use it to RESET the encoder
HSYNC_	not needed for our mode of operation. Tie to logic 1
VSYNC_	not needed for our mode of operation. Tie to logic 1
BLANK_	not needed for our mode of operation. Tie to logic 1
SCRESET	not needed. Tie to 0

3.2 Video Memory

You are provided a file called **linememory.v** which instantiates a module called **linememory**. This is a piece of SRAM. You will need to instantiate this memory and use it to store the first active line of video coming from the **vid_dec** module.

linememory has the following IO ports:

```
input   [8:0]   addra;
input   [8:0]   addrb;
input           clka;
input           clkb;
input   [7:0]   dina;
output  [7:0]   doutb;
input           wea;
```

addra: is a 9 bit address bus for the write port. Values should range from 0-319. This will correspond to 320 pixels. (Remember each 8 bit value contains 1 pixel)
dina: is a 8bit data bus. This is the data to write to addra. This should come from the video decoder.
wea: is the write enable for the write port. When this is active (high) data will be written to the line memory.
addrb: is an 9 bit address bus for the read port. Values should range from 0-319. This will correspond to

320 pixels. (Remember each 8 bit value contains 1 pixel)

dina: is a 8bit data bus. This is the data read from addrb. This should go to the asynchronous FIFO. NOTE: THE RAM HAS SYNCHRONOUS READ! This means that if the address is valid on clock cycle 1, the data will appear on clock cycle 2. Remember to account for this delay. Almost all memory is like this.

clka, clkb: these are the clocks. The SRAM on the VirtexE part is dual ported, the a and b ports are separate and can be used simultaneously. In fact they can be clocked separately which is the reason for the separate clocks. In this lab clka and clkb MUST BE THE SAME!

Keep in mind that this **linememory** module will soon be replaced with your SDRAM module from checkpoint 1. Why do we need the SDRAM? Why can't we just use the simpler SRAM on the Xilinx chip? (Hint: There are 160 block RAMs each of which can store 4096bits of data)

3.3 Asynchronous FIFO

The asynchronous FIFO is very similar to the synchronous FIFOs of Checkpoint1. The main difference is that the read and write clocks are now separate.
Reads and writes to this FIFO are completely independent. The clocks do not even need to be at the same frequency (though the data rates in and out can't be very different). This means that we can read and write when we're ready to. It also means that you must be VERY careful not to fill or empty the FIFO. A full or empty FIFO will almost always result in lost video data and major headaches. Why?

3.4 Video Output

Once the I2C data is sent, you can start sending the video date to the encoder. See the class information and the data sheets (page 22) and the VideoNutshell.doc for specific details on the 8-bit parallel bit stream to send.

The circuit has two main states – one when it is sending the I2C data, and one when transmitting video data. While transmitting video, we have to send a new byte every clock cycle, so it would be convenient to just have a couple of counters (HCOUNT and VCOUNT for example) running and let various operations like sending EAV, blanking data, SAV, video data take place depending on the value of the counter. Since counters are expensive, think of a way to share your video counters with the I2C logic (we've pretty much given this to you)

Whenever GRST (switch 1) is pressed, restart from the beginning, resending the I2C data and then start transmitting video data.

Notes

- It's not necessary to start transmitting your EAV immediately after you finish transmitting the I2C sequence. The video encoder will wait until it gets an EAV (ff, 00, 00, code) before it begins decoding your data. Just make sure you have your sequence exactly the way it should be. Once you begin sending, you have to maintain the sequence of EAV, HB, SAV, Video *exactly* with the correct number of cycles.
- If you get a shaky picture, then you probably have too few or too many cycles somewhere (wrong number of pixels per line, wrong number of lines per field, etc). Check your state machine/counter logic. An easy way to check you are doing the right thing is to write the output of your 656/601 stream to a file and actually count pixels and check the sync codes.
- If you are getting weird colors, you probably are not reading the memory in the correct order (i.e., Cb first, Y next, Cr, and then Y).
- Wrong values of Y,Cb and Cr could also result in the monitor losing sync! Therefore, if you want to play around with values start with black (0x80 for Chroma and 0x10 for Luma)
- We will always use 0x80 for chroma to keep things in black and white

4 Acknowledgements

Original Lab by Greg Gibeling
Based on labs by Prof.John Wawryznek and N. Vinay Krishnan
Video Expertise – Tom Oberheim

Checkpoint 2 Check-offs

Name: _____

Name:_____

Lab Section: _____

1. Answers to memory and FIFO questions _____ (25%)

2. Testbench and Simulation (VideoEncoder) _____ (25%)

3. Demo _____ (50%)

Total _____ (%)