ECE 385 – Digital Systems Laboratory

Lecture 14 – Experiment 8 (USB + VGA)

Zuofu Cheng

Spring 2018

Link to Course Website

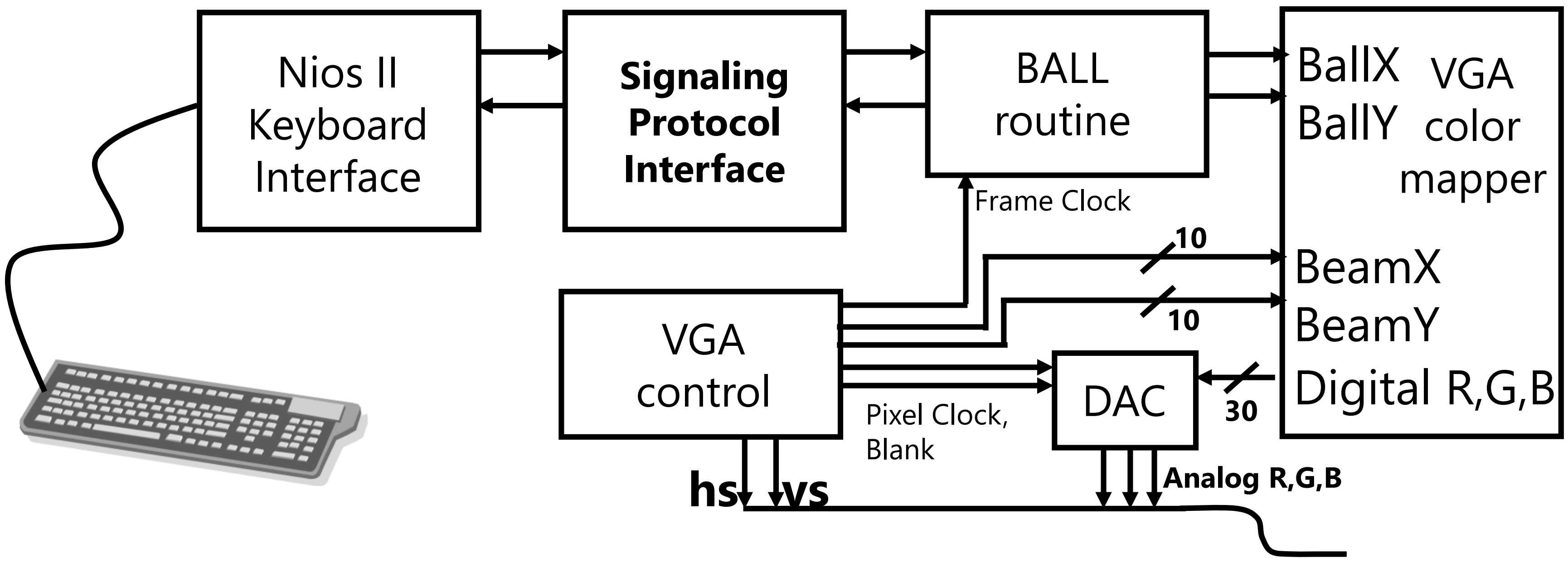
ECE ILLINOIS

ILLINOIS

Experiment 8 Goals

- Create low-level interface between NIOS II and USB chip (CY7C67200 “EZ-OTG”)
- Connect USB keyboard to “USB Host” port on DE2-115 and be able to enumerate & read key-codes
- Display bouncing ball using VGA controller on monitor (connect to VGA port)
- Use key-codes to control bouncing ball

Experiment 8 Overall Block Diagram



Ball routine: partially given

Color mapper: given

VGA controller: given

Experiment 8 Demo Points

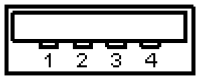
- Make code of last key pressed show on 7 segment display with an LED showing the correct direction. (1 point)
- Ball can move and bounce in both vertical and horizontal directions. If the ball can go through a wall the point will not be awarded. (1 point)
- Ball responds to keyboard commands (2 points) **or** show valid SignalTap logic analyzer trace of USB transaction (read and write from USB) for 1 point partial credit. (2 points total)
- Keyboard input is read on the keypress, not the release. The key should not continue to be read after the release. While the key is depressed, the ball may either continually react to the key or ignore it. The ball should not move diagonally or glitch. (1 point)

Experiment 8 Hints

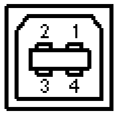
- Create the PIOs as recommended by the tutorial
  - Also create the JTAG UART
  - Lets you print to console using printf while debugging
  - May need to include stdlib.h/stdio.h
- Fill in functions in io_handler.c (lower level) and usb.c (higher level)
  - These toggle the PIOs which you created to talk to the USB chip
- Create a synchronizer circuit using hpi_io_intf.sv (this makes sure there synchronous->asynchronous transitions between the clocked NIOS II and un-clocked EZ-OTG do not cause problems)
  - Synchronizer circuit can just be a flip flop on each line

Experiment 8: USB Protocol

- The Universal Serial Bus (USB) standard defines the connection and communication protocols between computers and electronic devices
- Also provides power supply
- Compatible with a wide variety of devices
- USB Port has 4 pins



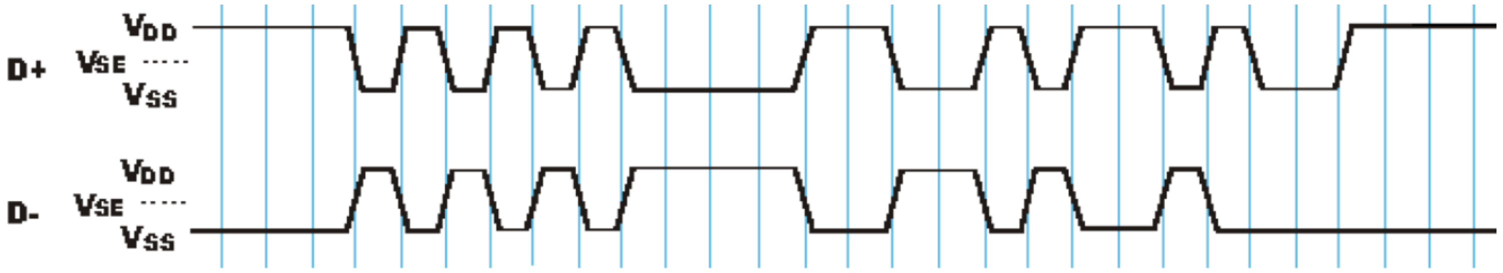
Type A



Type B

Pin	Name
1	VDD (5V)
2	D-
3	D+
4	GND

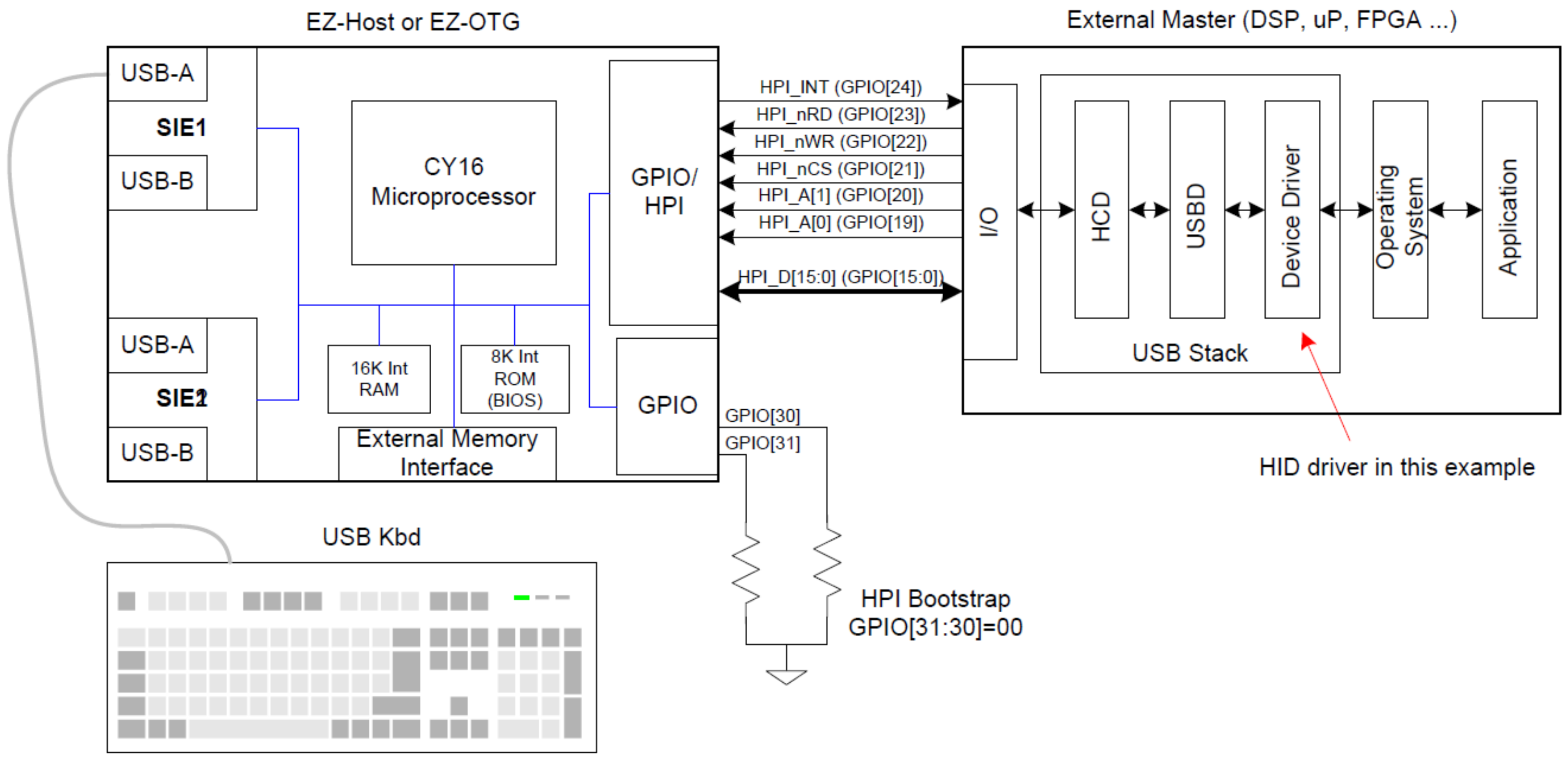
- Data transmit via the D+ and D- lines in a differential pattern
  - Physical layer



- Data are always polled by the host. Device cannot initiate transmission.

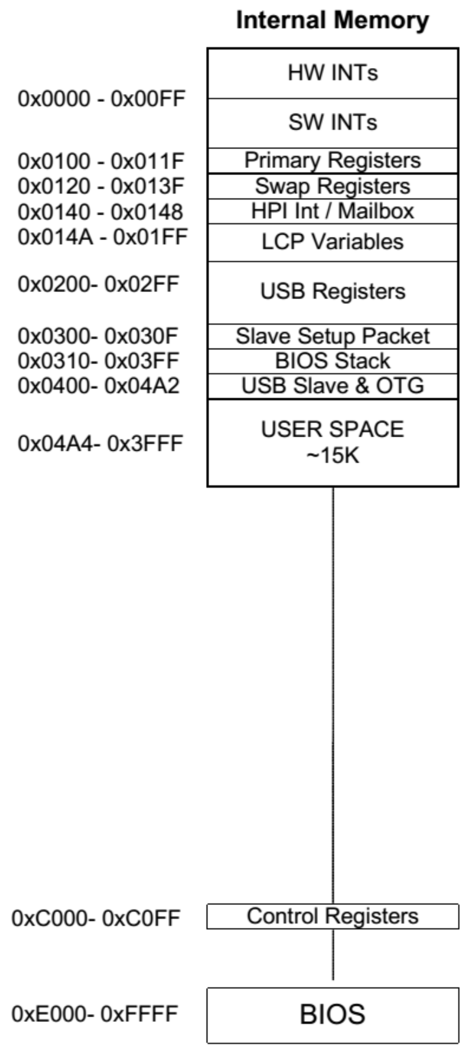
USB Controller: Cypress EZ-OTG

- DE2-115 board comes with the Cypress EZ-OTG (CY7C67200) USB controller
  - Can make DE2-115 act as a host or a USB device
  - For this lab, we will use USB host mode (single device – USB keyboard)



USB Controller: Cypress EZ-OTG

- Rule #1 of USB: All transfers are initiated by the host (in this case, this is the EZ-OTG)
- Rule #2 Each device can have multiple endpoints, all transfers are done from host to endpoint
- Different kinds of transfers are possible, but all transactions must follow above rules
- We program USB by programming memory of the EZ-OTG through the parallel interface (HPI)
- EZ-OTG has built in software & 16-bit CPU to do low level packet assembly, queueing, CRC etc…
- Memory space of EZ-OTG is shown on the right
- We'll read/write into "USER SPACE" where the 16-bit CPU will dump the contents of USB packets

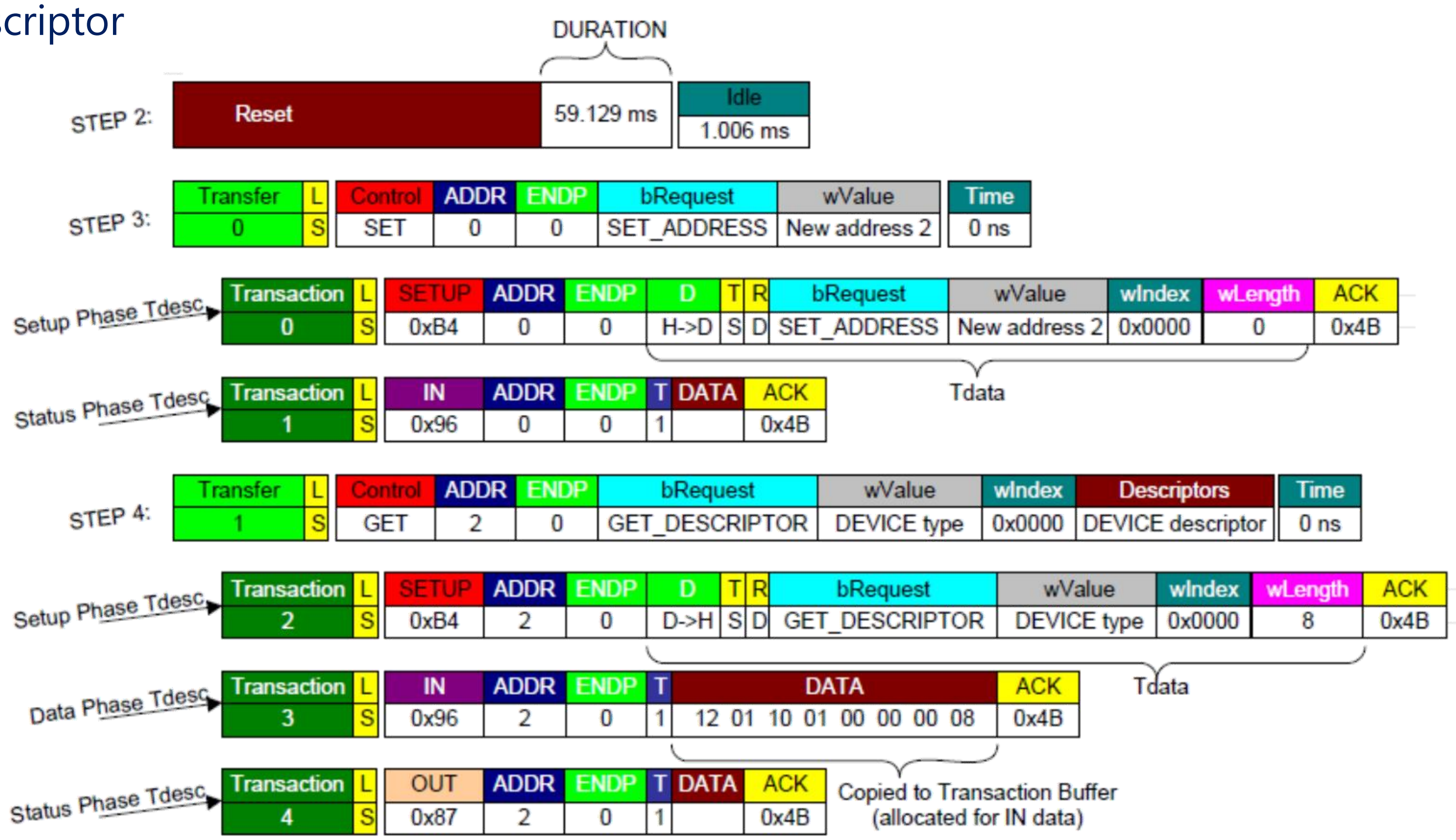


USB Transactions

- Each **USB transfer** is made up of one or more **USB transactions**
  - Four types: control, bulk, isochronous, and interrupt
- Each transaction is then made up of 2-3 **USB packets**
  - First packet is the direction token sent by the Host Controller, which can be: Setup, IN, or OUT
  - Second packet is the data packet, sent by either the Host or the Device, depending on direction
  - Third packet is the ACK/Handshake
  - Each packet has data, as well as CRC and synchronization
- EZ-OTG de-deserializes USB (physical layer) and creates/checks CRC using onboard CPU and places packet data in the RAM
- Provided USB driver (usb.c) interacts with packet data, using the EZ-OTG chip as an intermediary

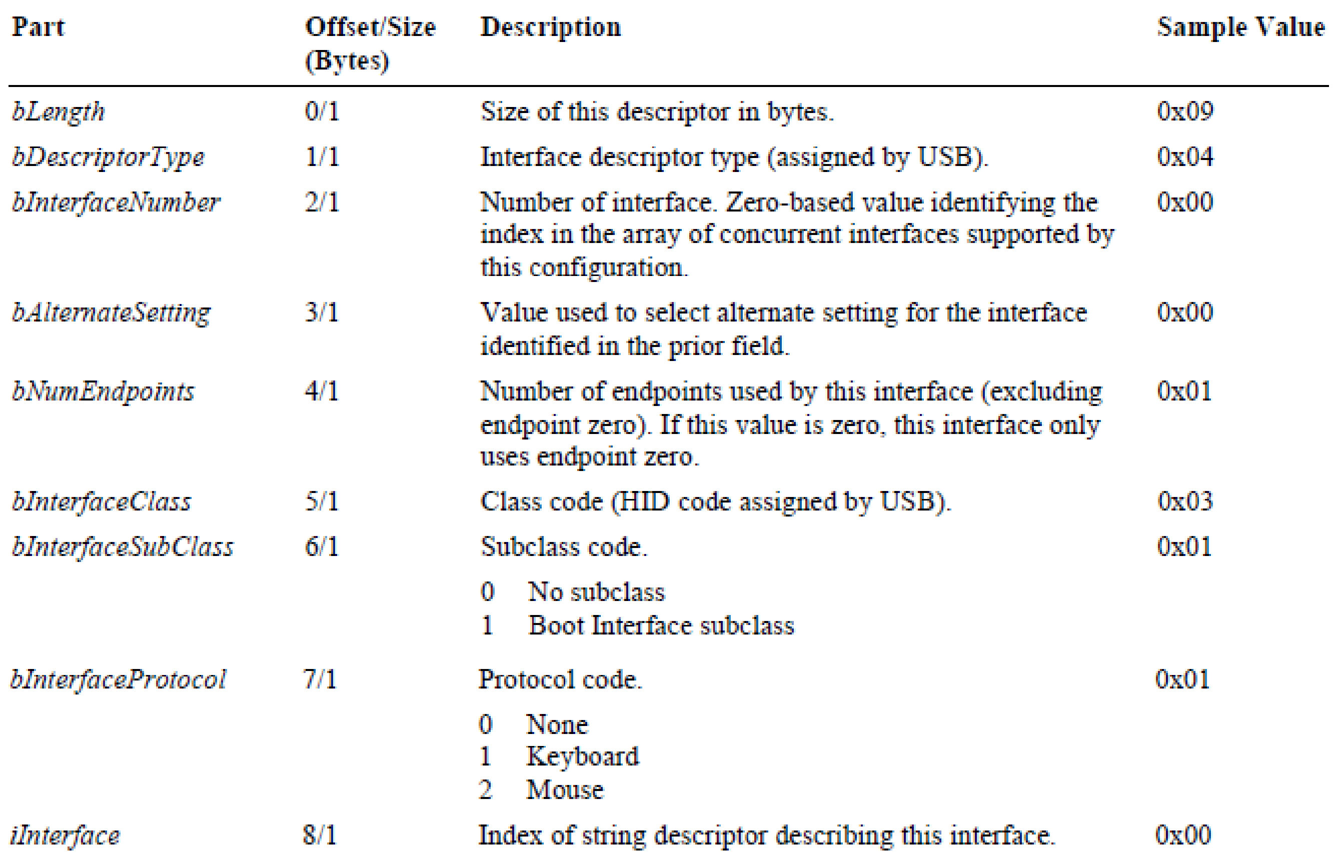
USB Enumeration

- Before a USB device may be used, it needs to be enumerated
- Enumeration consists of 4 steps:
  - Initialization the EZ-OTG chip
  - Performing USB reset
  - Assigning USB address to device
  - Getting device descriptor
- This is done by usb.c


DURATION
STEP 2: Reset 59.129 ms Idle 1.006 ms
STEP 3: Transfer 0 L S Control SET ADDR 0 ENDP 0 bRequest SET_ADDRESS wValue New address 2 Time 0 ns
Setup Phase Tdesc Transaction 0 L S SETUP 0xB4 ADDR 0 ENDP 0 D H->D T S R D bRequest SET_ADDRESS wValue New address 2 wIndex 0x0000 wLength 0 ACK 0x4B
Tdata
Status Phase Tdesc Transaction 1 L S IN 0x96 ADDR 0 ENDP 0 T 1 DATA ACK 0x4B
STEP 4: Transfer 1 L S Control GET ADDR 2 ENDP 0 bRequest GET_DESCRIPTOR wValue DEVICE type wIndex 0x0000 Descriptors DEVICE descriptor Time 0 ns
Setup Phase Tdesc Transaction 2 L S SETUP 0xB4 ADDR 2 ENDP 0 D D->H T S R D bRequest GET_DESCRIPTOR wValue DEVICE type wIndex 0x0000 wLength 8 ACK 0x4B
Tdata
Data Phase Tdesc Transaction 3 L S IN 0x96 ADDR 2 ENDP 0 T 1 DATA 12 01 10 01 00 00 00 08 ACK 0x4B
Status Phase Tdesc Transaction 4 L S OUT 0x87 ADDR 2 ENDP 0 T 1 DATA ACK 0x4B
Copied to Transaction Buffer (allocated for IN data)

USB Device Descriptor

- Device descriptor tells host what kind of device is connected
- Example device descriptor for USB HID (Human Interface Device)



Part	Offset/Size (Bytes)	Description	Sample Value
bLength	0/1	Size of this descriptor in bytes.	0x09
bDescriptorType	1/1	Interface descriptor type (assigned by USB).	0x04
bInterfaceNumber	2/1	Number of interface. Zero-based value identifying the index in the array of concurrent interfaces supported by this configuration.	0x00
bAlternateSetting	3/1	Value used to select alternate setting for the interface identified in the prior field.	0x00
bNumEndpoints	4/1	Number of endpoints used by this interface (excluding endpoint zero). If this value is zero, this interface only uses endpoint zero.	0x01
bInterfaceClass	5/1	Class code (HID code assigned by USB).	0x03
bInterfaceSubClass	6/1	Subclass code. 0 No subclass 1 Boot Interface subclass	0x01
bInterfaceProtocol	7/1	Protocol code. 0 None 1 Keyboard 2 Mouse	0x01
iInterface	8/1	Index of string descriptor describing this interface.	0x00

USB Controller: Cypress EZ-OTG

- EZ-OTG has 4 HPI registers for commands and configurations
  - Read and write on the registers to control EZ-OTG by specifying the address
  - Functionality of each bit field in each register is defined in the data sheet

Port Registers	HPI A [1]	HPI A [0]	Access
HPI DATA	0	0	RW
HPI MAILBOX	0	1	RW
HPI ADDRESS	1	0	W
HPI STATUS	1	1	R

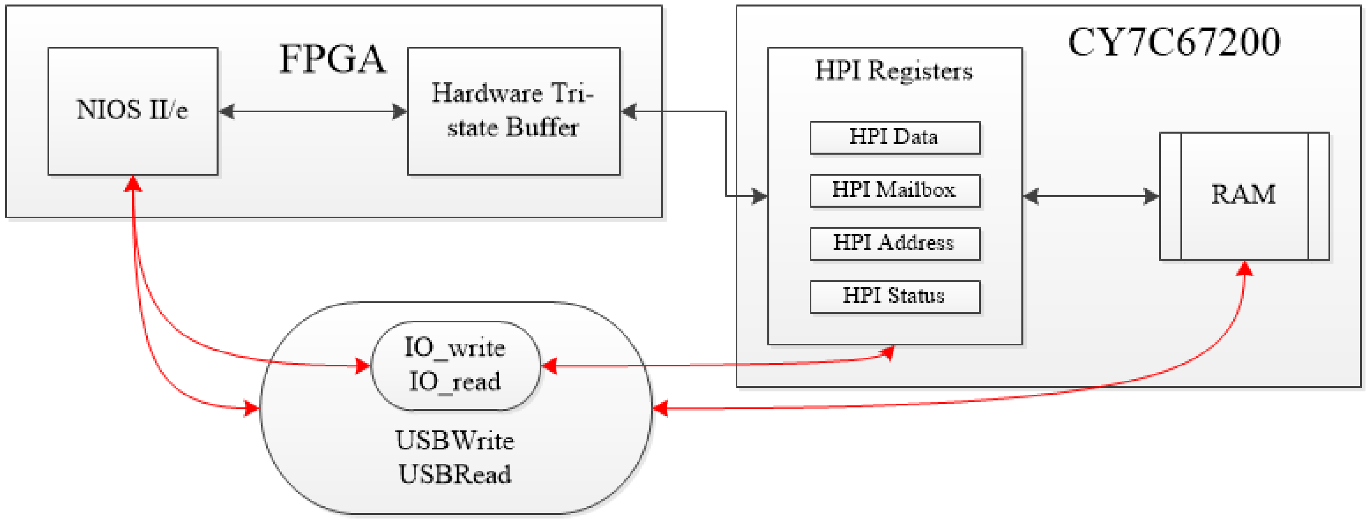
HPI Status ←

Bit16	Bit14	Bit13	Bit12	Bit11	Bit10	Bit9	Bit8
VBUS Flag	ID Flag	Reserved	SOF/EOP2 Flag	Reserved	SOF/EOP1 Flag	Reset2 Flag	Mailbox IN Flag
Bit7	**Bit6**	**Bit5**	**Bit4**	**Bit3**	**Bit2**	**Bit1**	**Bit0**
Resume2 Flag	Resume1 Flag	SIE2msg	SIE1msg	Done2 Flag	Done1 Flag	Reset1 Flag	Mailbox OUT Flag

- EZ-OTG also has a RAM, a ROM (BIOS), and a 16-bit RISC processor (CY16) which can be programmed (we'll use the CY16's built in software for host)
  - Many commands can be made by writing the command into the RAM and then activate through the HPI registers
- Note 4 register "window" into bigger EZ-OTG memory space

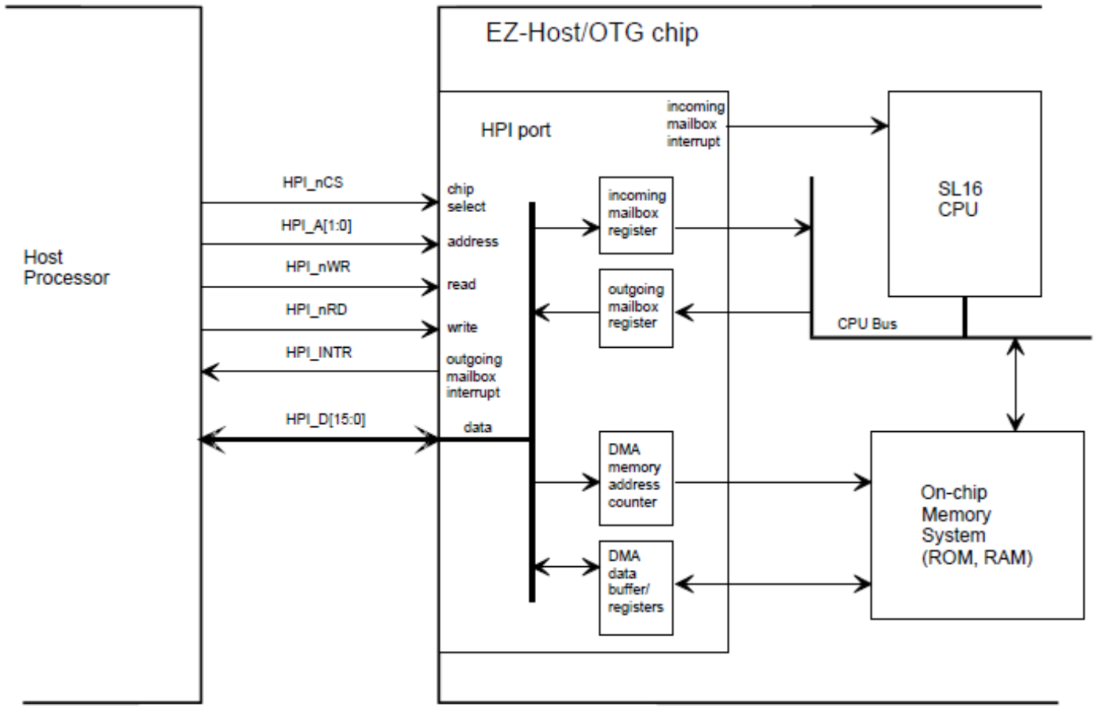
Communicating with the EZ-OTG

- HPI has only 4 hardware addresses (ADDR[1], ADDR[0]) to reduce pin-count
- `IO_read()/IO_write()` reads/writes into these 4 addresses
  - interfaces with PIO, just like we did in Lab 7
- `USBRead()/USBWrite()` reads/writes into full EZ-OTG memory space
  - Muliple calls to `IO_read()/IO_write()`



IO Read/Write Operation

- IO Read/Write functions interact directly with PIO (and additional hardware in FPGA, as needed)
- Simplest approach: drive hardware pins directly using PIOs
  - hpi_io_intf.sv acts as a buffer, adding one cycle of delay on the output
  - This ensures there are no glitches from synchronous FPGA logic to asynchronous HPI interface

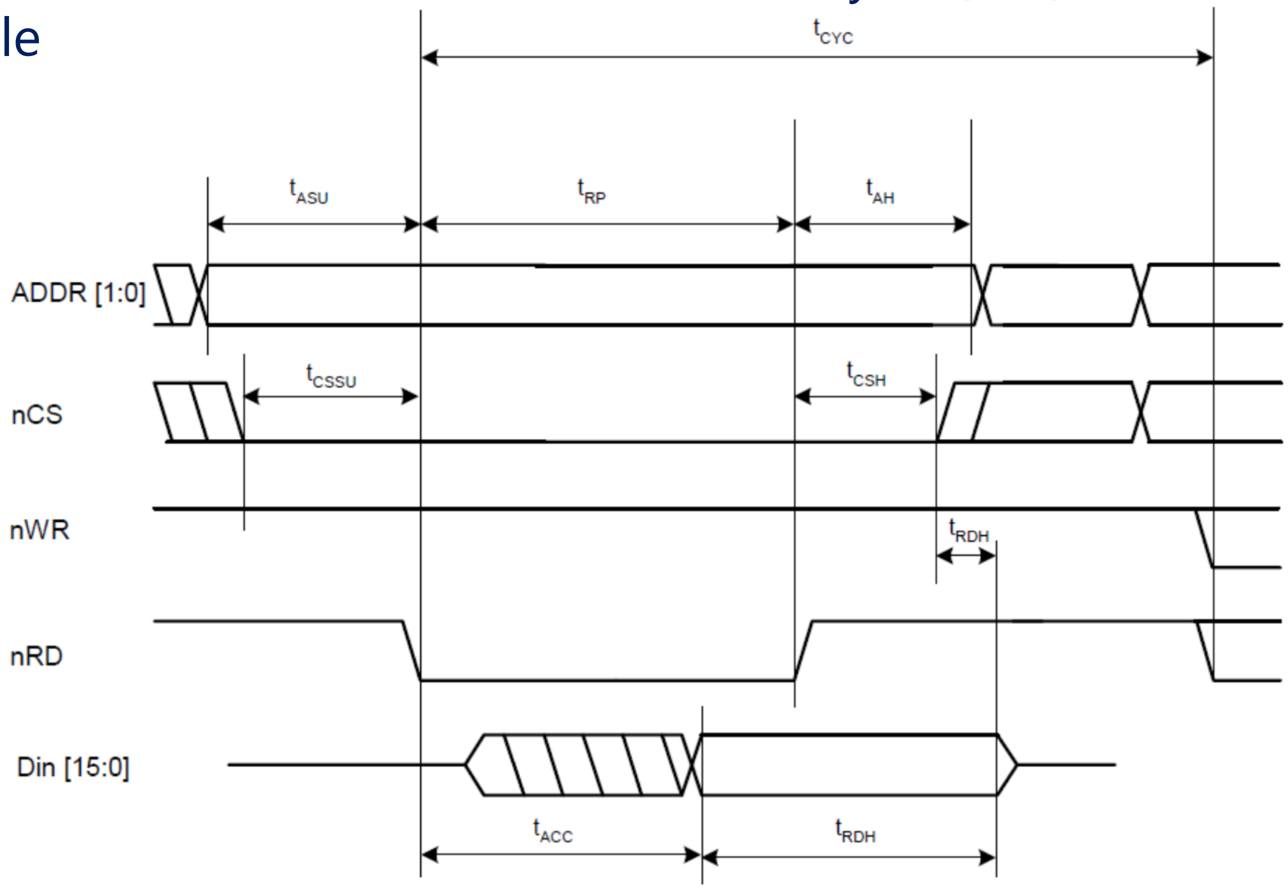

EZ-Host/OTG chip
Host Processor
HPI port
incoming mailbox interrupt
SL16 CPU
HPI_nCS
HPI_A[1:0]
HPI_nWR
HPI_nRD
HPI_INTR
HPI_D[15:0]
chip select
address
read
write
outgoing mailbox interrupt
data
incoming mailbox register
outgoing mailbox register
CPU Bus
DMA memory address counter
DMA data buffer/ registers
On-chip Memory System (ROM, RAM)

Understanding hpi_io_intf.sv

```
module hpi_io_intf (
input[1:0]  from_sw_address,
output[15:0] from_sw_data_in,
input[15:0] from_sw_data_out,
input from_sw_r, from_sw_w, from_sw_cs,
inout[15:0] OTG_DATA,
output[1:0] OTG_ADDR,
output OTG_RD_N, OTG_WR_N, OTG_CS_N, OTG_RST_N,
input OTG_INT, Clk, Reset);
```

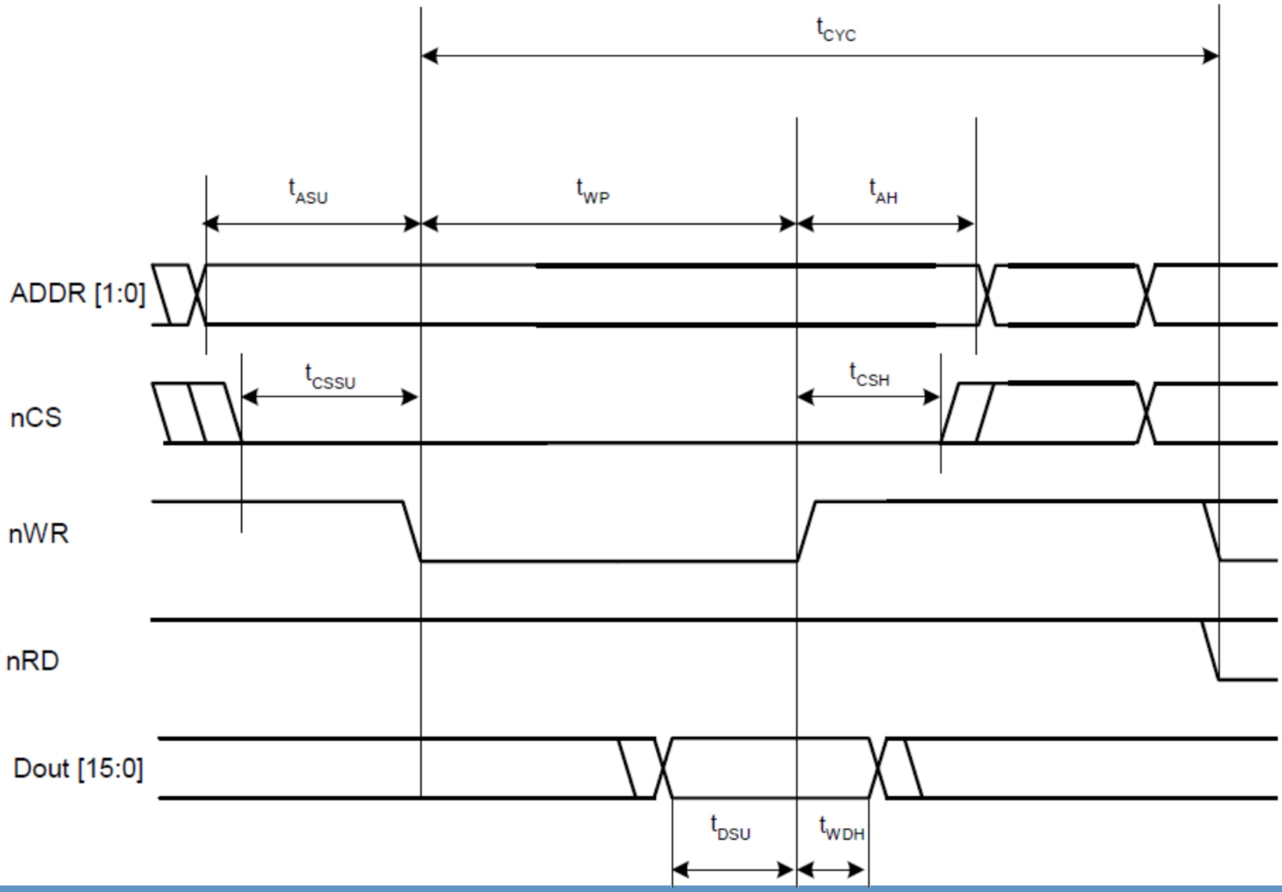
IO Read Timing via HPI

- IO Read timing is similar to SRAM from Lab 6
- Note: like SRAM, DATA pins are bidirectional (will need some type of tristate buffer, either in/out or use old tristate.SV and control both ports via PIOs)
- Access time is much faster than CPU clock cycle (6ns) so reads can happen in one cycle



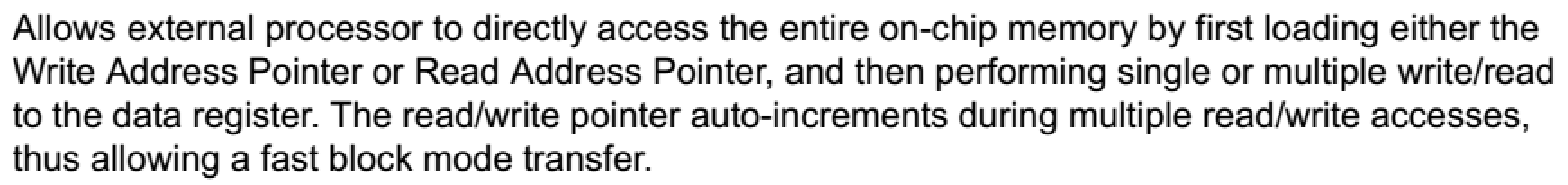
IO Write Timing via HPI

- Write timing is similar to asynchronous SRAM as well
- Note, need to toggle WR (write enable) through PIOs
- CPU is not able to operate at full HPI speed, irrelevant because keyboard has low data rate (might consider making a faster interface if using other USB profiles)



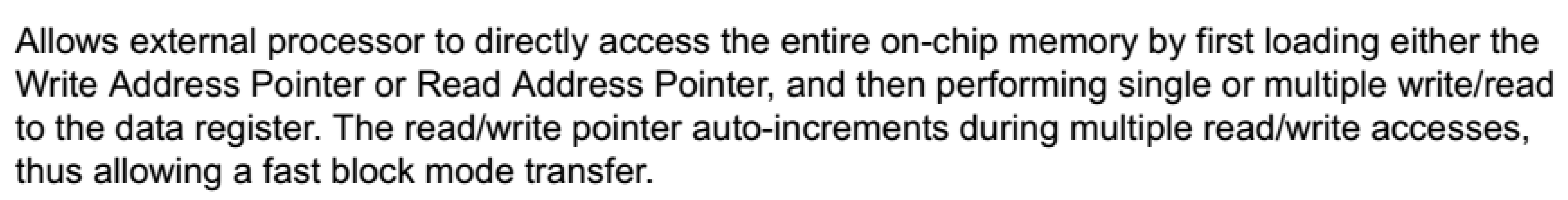
USB Read Operation

- USB read operation is defined as follows (from BIOS manual under HPI)
  - First set address register (0b10) to address (in EZ-OTG memory space)
  - Read data out of data register (0b00)
- Address may auto increment (therefore, you will sometimes see multiple `IO_read()` following a `USBRead()`
- Not necessary to give EZ-OTG command through mailbox (it


Allows external processor to directly access the entire on-chip memory by first loading either the Write Address Pointer or Read Address Pointer, and then performing single or multiple write/read to the data register. The read/write pointer auto-increments during multiple read/write accesses, thus allowing a fast block mode transfer.

USB Write Operation

- USB write operation is defined as follows (from BIOS manual under HPI)
  - First set address register (0b10) to address (in EZ-OTG memory space)
  - Set data register (0b11) with data to be written
- Address may auto increment (therefore, you will sometimes see multiple `IO_write()` following a `USBWrite()`
- Not necessary to give EZ-OTG command through mailbox (it

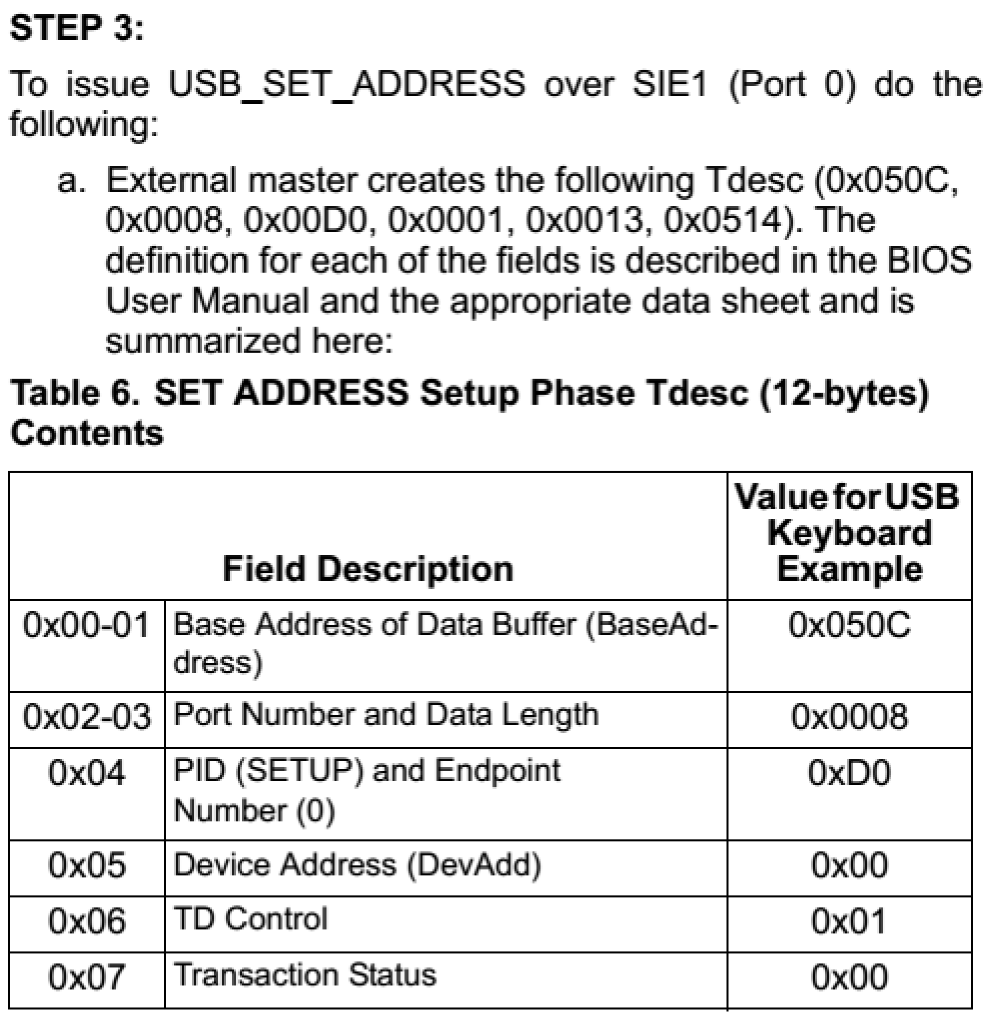


Allows external processor to directly access the entire on-chip memory by first loading either the Write Address Pointer or Read Address Pointer, and then performing single or multiple write/read to the data register. The read/write pointer auto-increments during multiple read/write accesses, thus allowing a fast block mode transfer.

How The Software Uses USBRead/Write

```
void UsbSetAddress()
{
...
IO_write(HPI_DATA,0x050C);
IO_write(HPI_DATA,0x0008);

IO_write(HPI_DATA,0x00D0);
IO_write(HPI_DATA,0x0001);
IO_write(HPI_DATA,0x0013);
IO_write(HPI_DATA,0x0514);
…
UsbWrite(HUSB_SIE1_pCurrentTDPtr,0x0500);
```



STEP 3:

To issue USB_SET_ADDRESS over SIE1 (Port 0) do the following:

a. External master creates the following Tdesc (0x050C, 0x0008, 0x00D0, 0x0001, 0x0013, 0x0514). The definition for each of the fields is described in the BIOS User Manual and the appropriate data sheet and is summarized here:

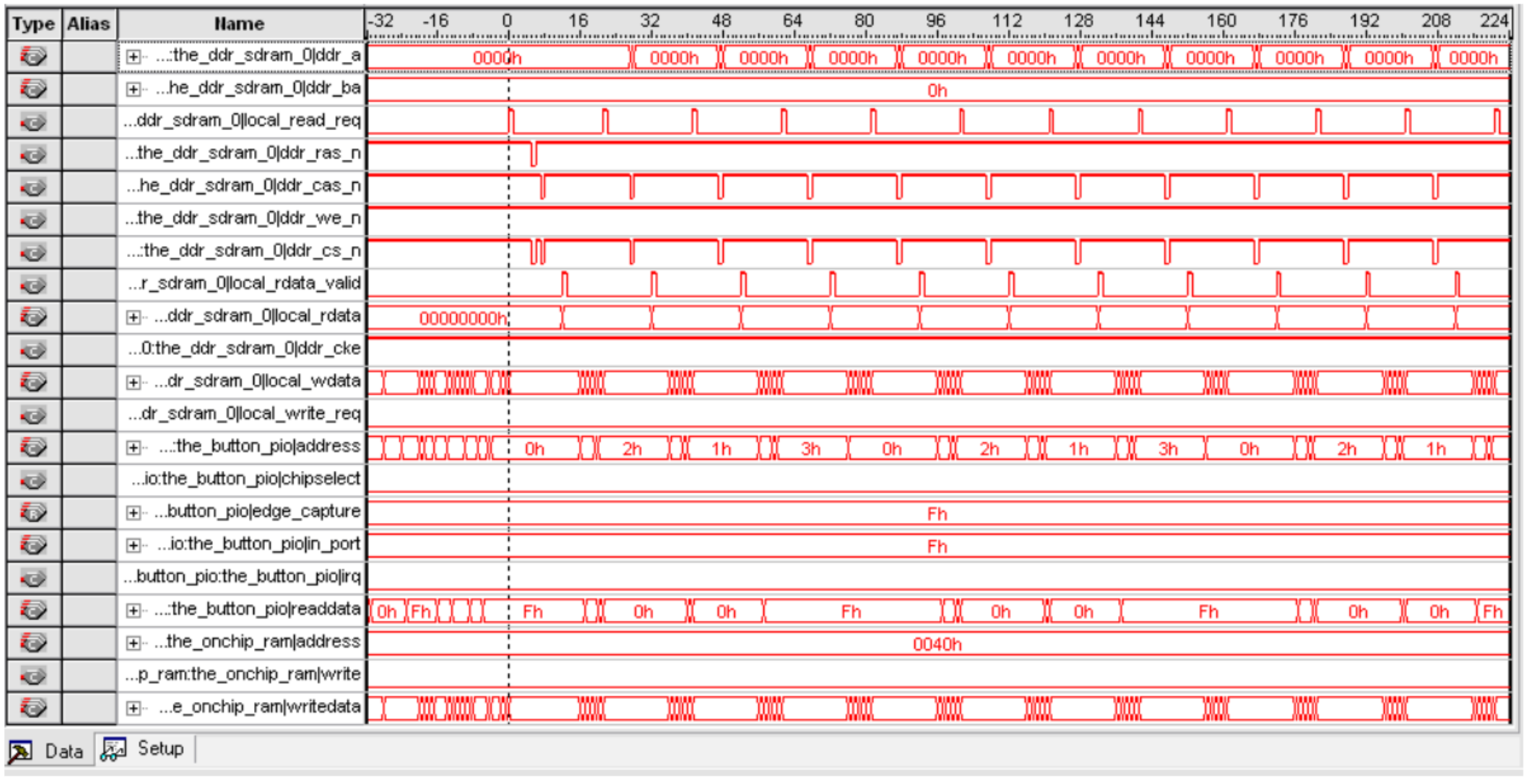
Table 6. SET ADDRESS Setup Phase Tdesc (12-bytes) Contents

Field Description		Value for USB Keyboard Example
0x00-01	Base Address of Data Buffer (BaseAd-dress)	0x050C
0x02-03	Port Number and Data Length	0x0008
0x04	PID (SETUP) and Endpoint Number (0)	0xD0
0x05	Device Address (DevAdd)	0x00
0x06	TD Control	0x01
0x07	Transaction Status	0x00

Note: this is already in provided code!

Using SignalTap II

- Simulation is useful for designs which are entirely on-chip
- Some external devices will have simulation models (SDRAM, for example)
- Most external devices will not – need to debug in-circuit!

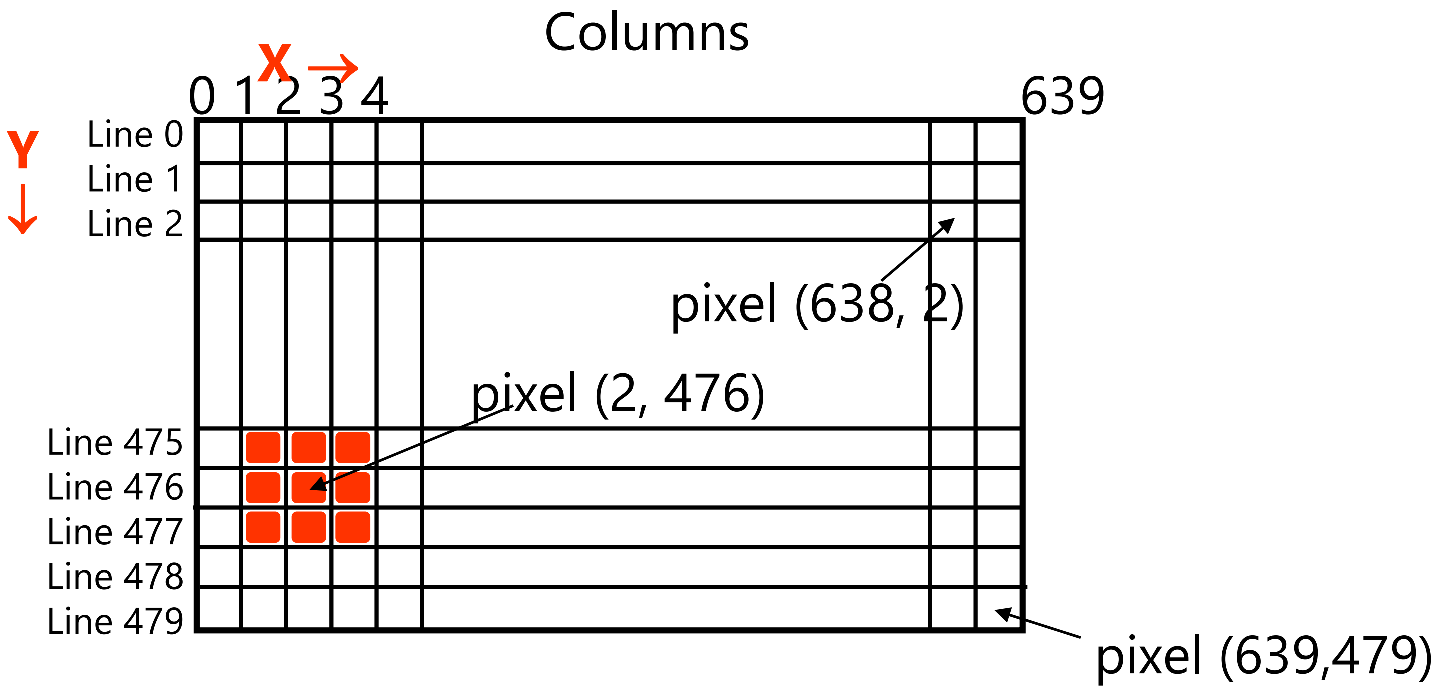

Type
Alias
Name
-32
-16
0
16
32
48
64
80
96
112
128
144
160
176
192
208
224
...:the_ddr_sdram_0|ddr_a
...he_ddr_sdram_0|ddr_ba
...ddr_sdram_0|local_read_req
...the_ddr_sdram_0|ddr_ras_n
...he_ddr_sdram_0|ddr_cas_n
...the_ddr_sdram_0|ddr_we_n
...:the_ddr_sdram_0|ddr_cs_n
...r_sdram_0|local_rdata_valid
...ddr_sdram_0|local_rdata
...0:the_ddr_sdram_0|ddr_cke
...dr_sdram_0|local_wdata
...dr_sdram_0|local_write_req
...:the_button_pio|address
...io:the_button_pio|chipselect
...button_pio|edge_capture
...io:the_button_pio|in_port
...button_pio:the_button_pio|irq
...:the_button_pio|readdata
...the_onchip_ram|address
...p_ram:the_onchip_ram|write
...e_onchip_ram|writedata
0000h
0h
00000000h
2h
1h
3h
Fh
0040h
Data
Setup

Using SignalTap II - Continued

- General overview for using SignalTap
  - Define size of logic analyzer to instantiate (remember that SignalTap uses FPGA resources – LUTs and BRAMs)
  - Define signals to watch (node picker)
  - Define clock(s)
  - Determine and set-up trigger conditions – this tells SignalTap when to start recording (note that trigger can happen at the beginning of, middle, or end of sequence)
  - Run SignalTap – invoke trigger condition on hardware (pressing a button, starting the device, starting the Nios II software, etc)
  - SignalTap will trigger and display waveform for watched signals
- Note that this represents what is actually happening on the board – not a simulation!

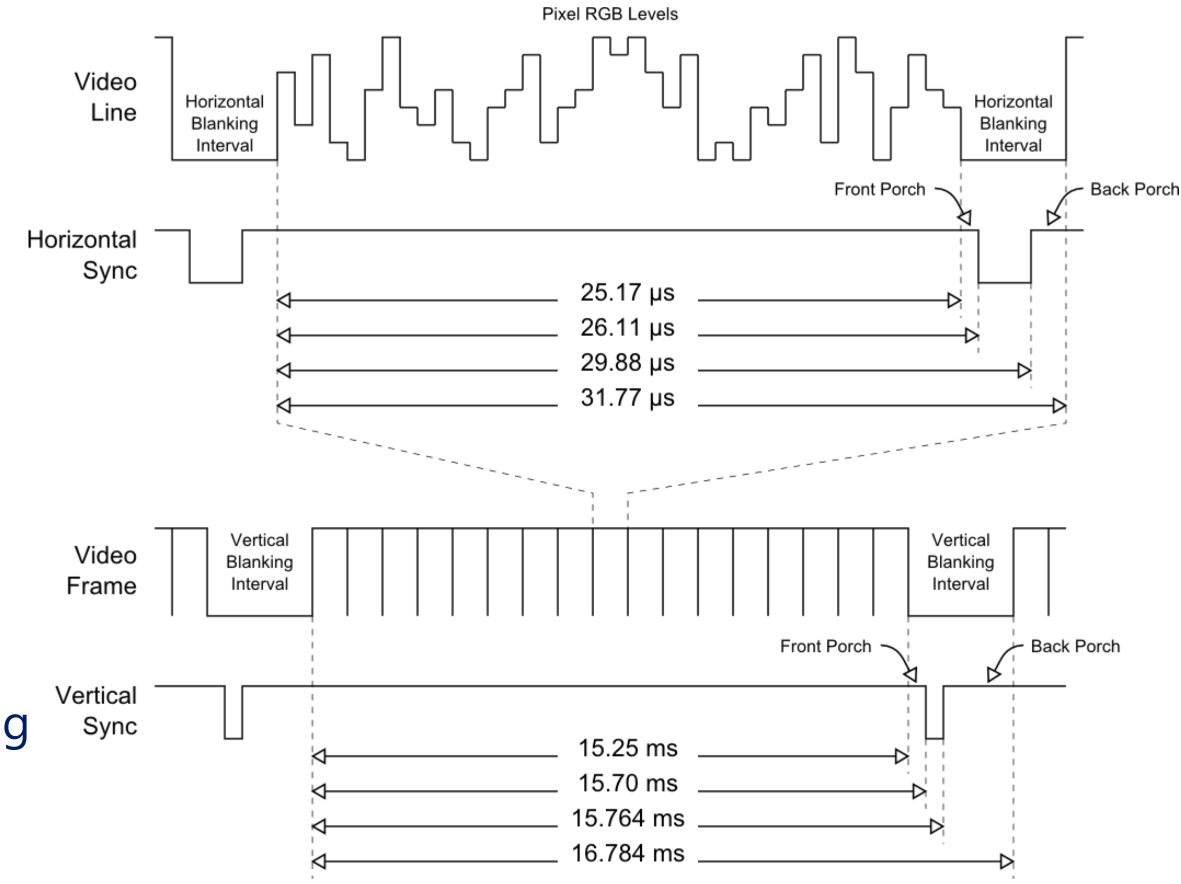
VGA Monitor Operation

- VGA (Video Graphics Array) Standard
  - The screen is organized as a matrix of pixels
    - 640 horizontal pixels x 480 vertical lines
  - An Electron Beam "paints" each pixel from left to right in each row, and each row from top to bottom



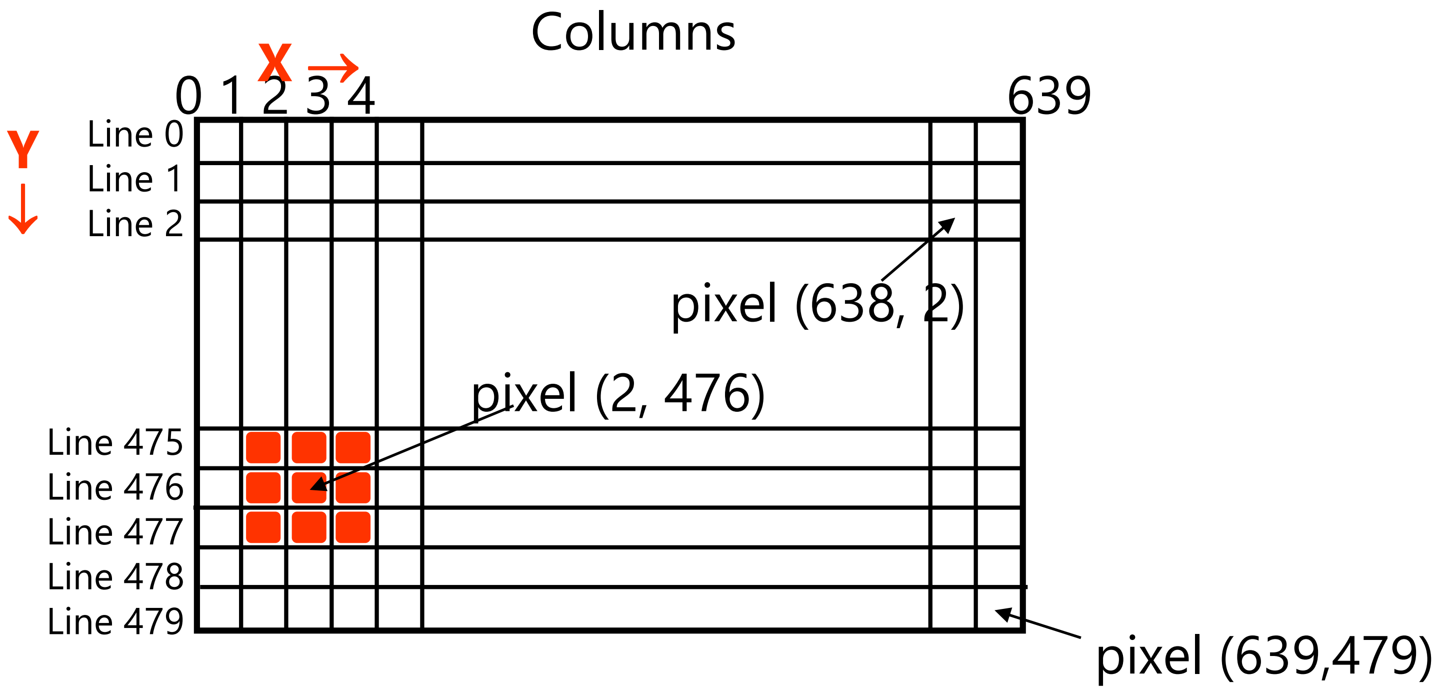
VGA Timing (continued)

- Screen refresh rate = 60 Hz
  - Note: this doesn't mean your game must run at 60 Hz
  - But you must generate VGA signal 60 times a second!
  - One frame = 16.67 ms
- Overall pixel frequency = 25.175 MHz
- Can approximate by using 25.000 MHz (50 MHz / 2 using flip flop)
  - Makes frame time longer, now 16.784 ms
- Note: VGA communicates via analog voltages (DE2-115 has DAC to generate these)
- Easy to create clock divider by 2 (how?)



VGA Monitor Operation

- In lab 8, we used a simple color mapper combined with the VGA controller to draw simple shapes
- Color mapper needs to have as inputs the horizontal and vertical position counters, and maps output color either to foreground color (e.g. red) or background color (e.g. white)



Columns
X →
0 1 2 3 4
639
Y ↓
Line 0
Line 1
Line 2
pixel (638, 2)
pixel (2, 476)
Line 475
Line 476
Line 477
Line 478
Line 479
pixel (639,479)