ECE 385 – Digital Systems Laboratory

Lecture 14 – Experiment 8 (USB + VGA)

Zuofu Cheng, Volodymyr Kindratenko

Fall 2016

Link to Course Website

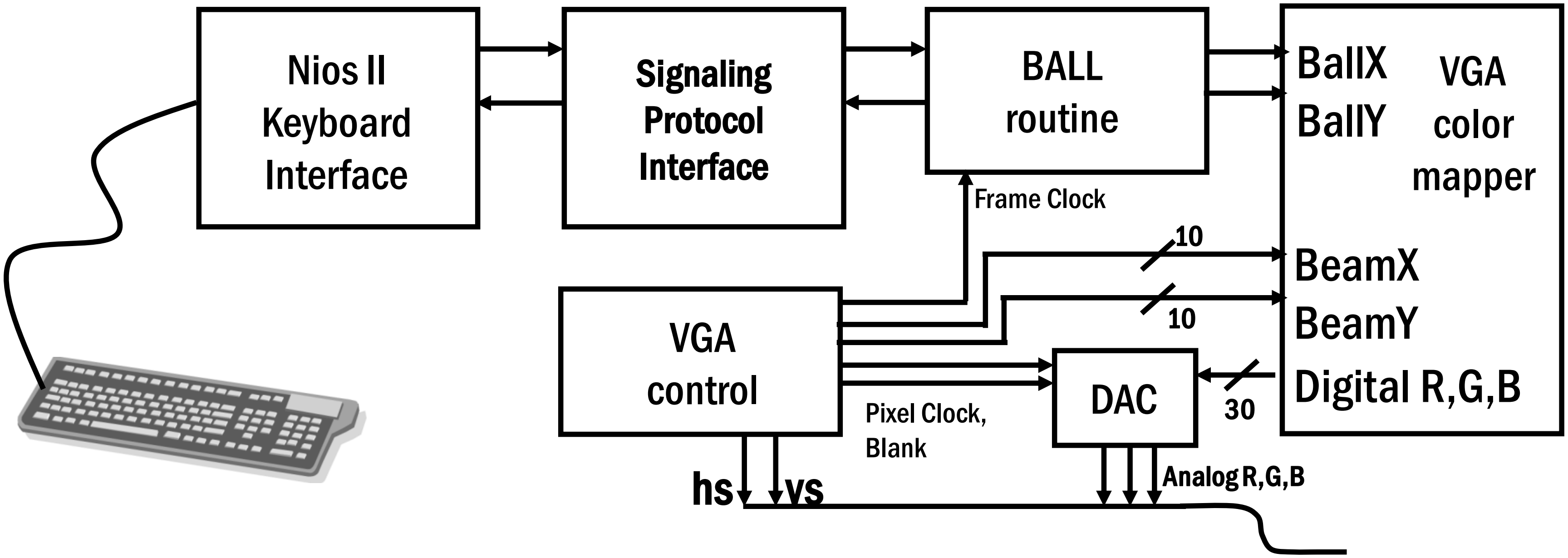
ECE ILLINOIS

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

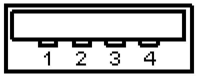
- Make code of last key pressed show on 7 segment display with an LED showing the correct direction. (1 point)
- Ball can move and bounce in both vertical and horizontal directions. If the ball can go through a wall the point will not be awarded. (1 point)
- Ball responds to keyboard commands. (1 point)
- 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. (1 point)
- Show valid SignalTap logic analyzer trace of USB transaction (read and write from USB)

Experiment 8 Hints

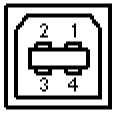
- Create the PIOs as recommended by the tutorial
  - Also create the JTAG UART
  - Lets you print to console using printf while debugging
- Fill in functions in io_handler.c (lower level) and usb.c (higher level)
  - These toggle the PIOs which you created to talk to the USB chip
- Create a synchronizer 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



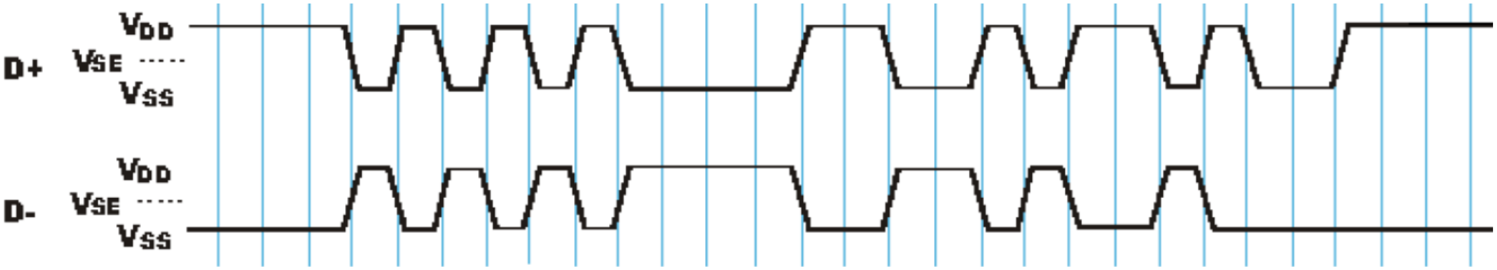
Type A



Type B

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

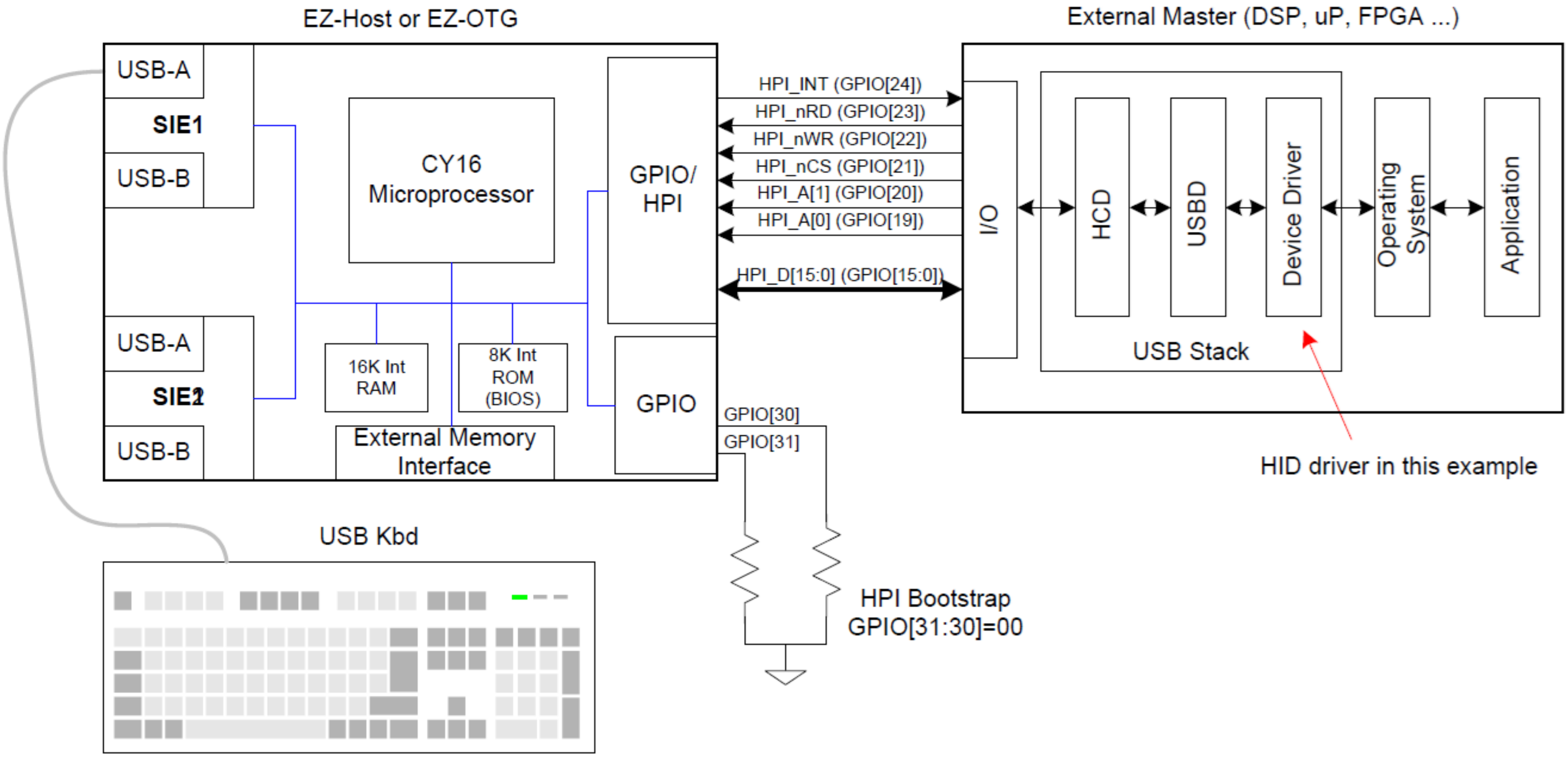
- Data transmit via the D+ and D- lines in a differential pattern
  - Physical layer



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

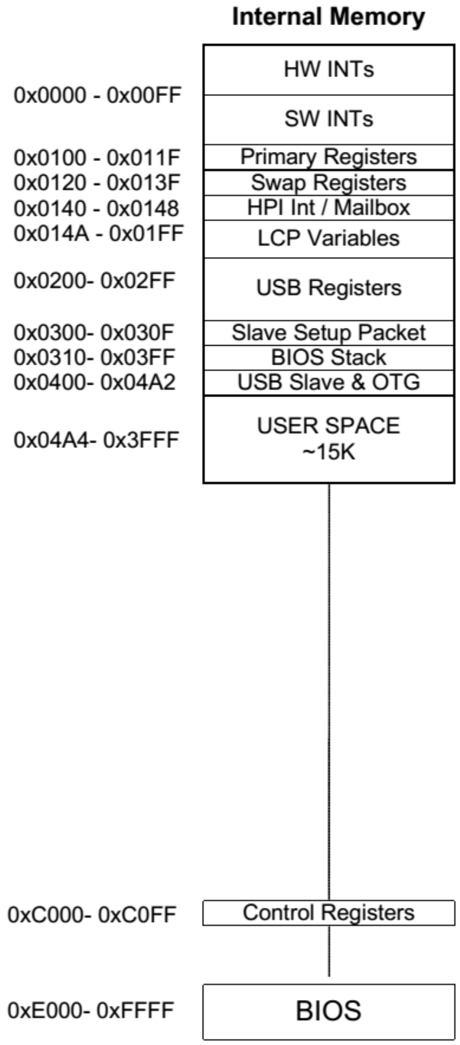
USB Controller: Cypress EZ-OTG

- 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 transactions are done from host to endpoint
- Different kinds of transactions 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, etc…
- Memory space of EZ-OTG is shown on the right
- We’ll read/write into “USER SPACE” and interact with software built into the EZ-OTG

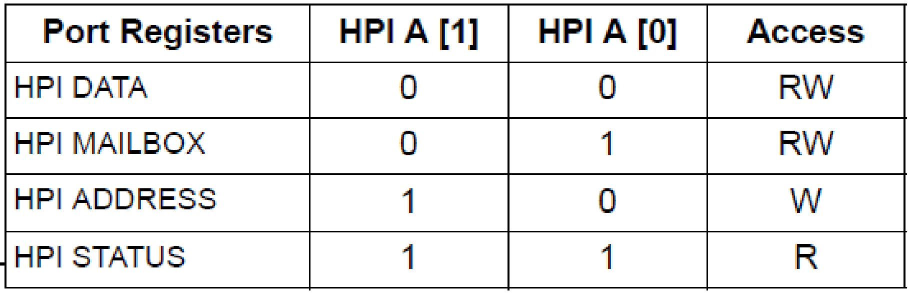


Internal Memory

0x0000 - 0x00FF	HW INTs SW INTs
0x0100 - 0x011F	Primary Registers
0x0120 - 0x013F	Swap Registers
0x0140 - 0x0148	HPI Int / Mailbox
0x014A - 0x01FF	LCP Variables
0x0200- 0x02FF	USB Registers
0x0300- 0x030F	Slave Setup Packet
0x0310- 0x03FF	BIOS Stack
0x0400- 0x04A2	USB Slave & OTG
0x04A4- 0x3FFF	USER SPACE ~15K
0xC000- 0xC0FF	Control Registers
0xE000- 0xFFFF	BIOS

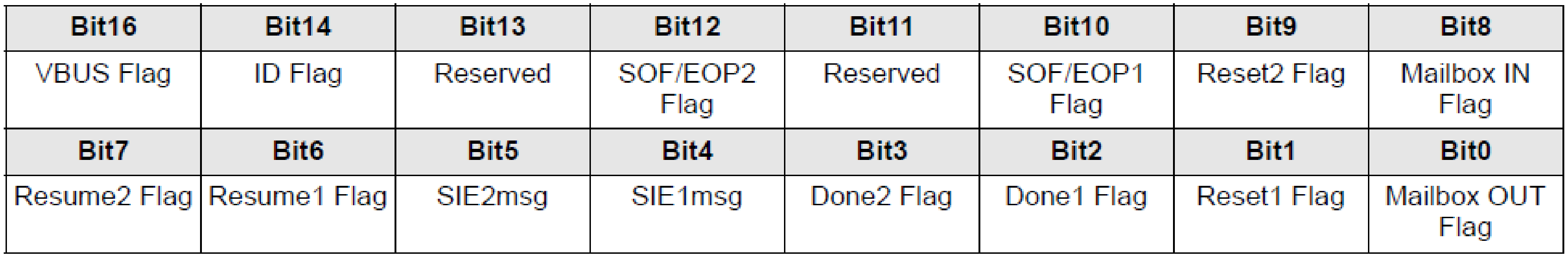
USB Controller: Cypress EZ-OTG

- 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 ←

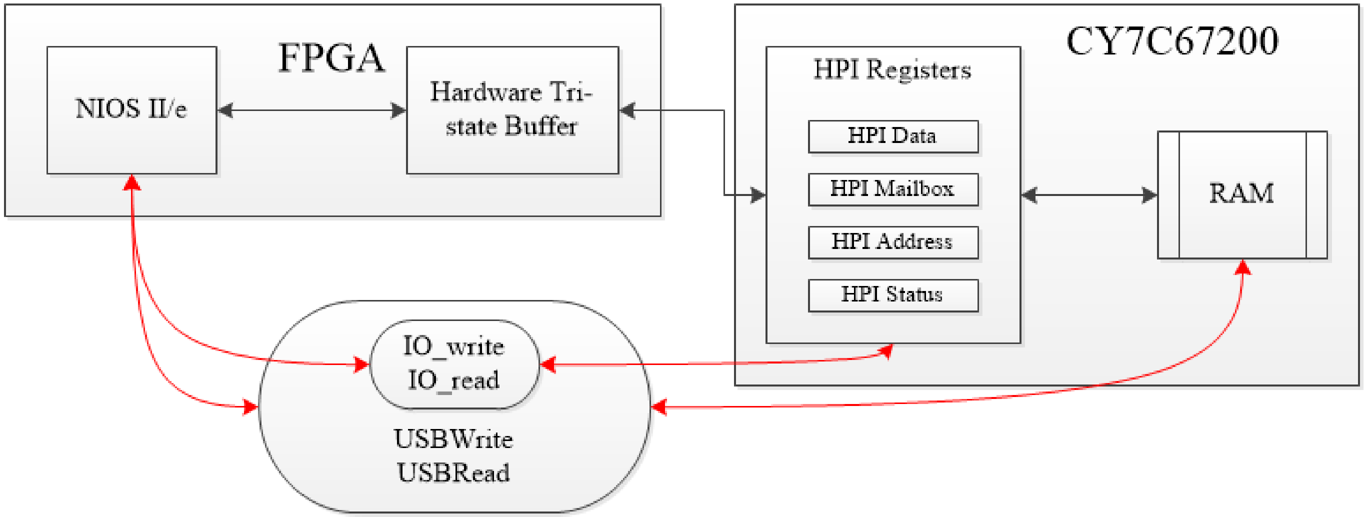


Bit16	Bit14	Bit13	Bit12	Bit11	Bit10	Bit9	Bit8
VBUS Flag	ID Flag	Reserved	SOF/EOP2 Flag	Reserved	SOF/EOP1 Flag	Reset2 Flag	Mailbox IN Flag
Bit7	**Bit6**	**Bit5**	**Bit4**	**Bit3**	**Bit2**	**Bit1**	**Bit0**
Resume2 Flag	Resume1 Flag	SIE2msg	SIE1msg	Done2 Flag	Done1 Flag	Reset1 Flag	Mailbox OUT Flag

- 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

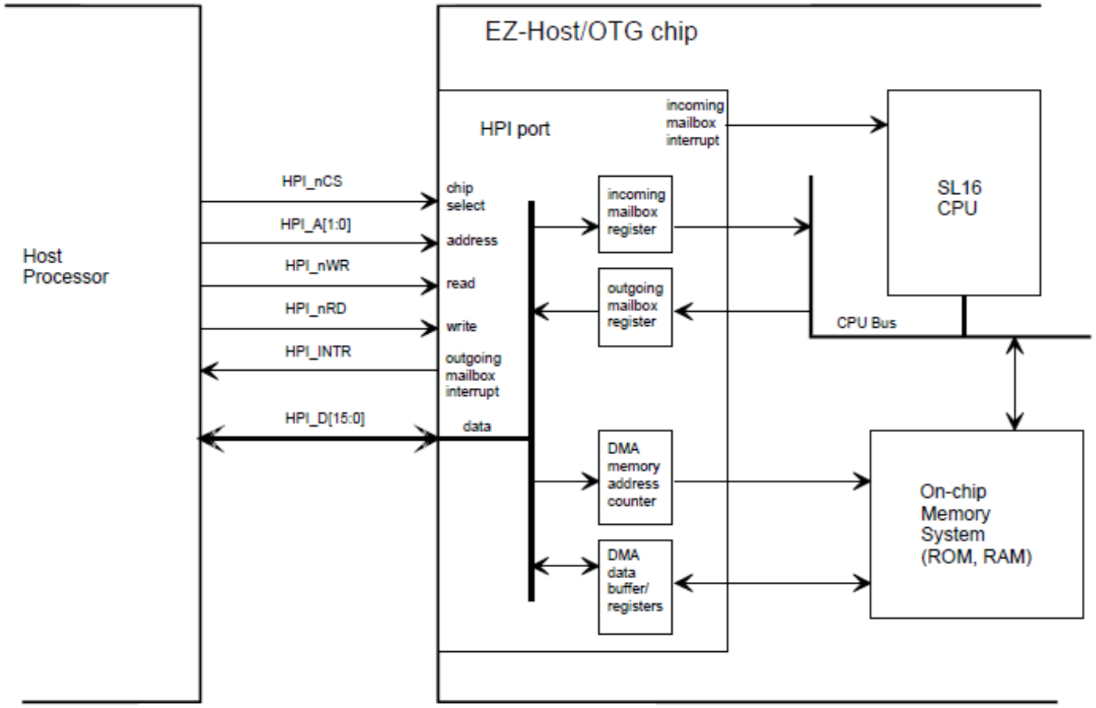
Communicating with the EZ-OTG

- 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
  - Multiple 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



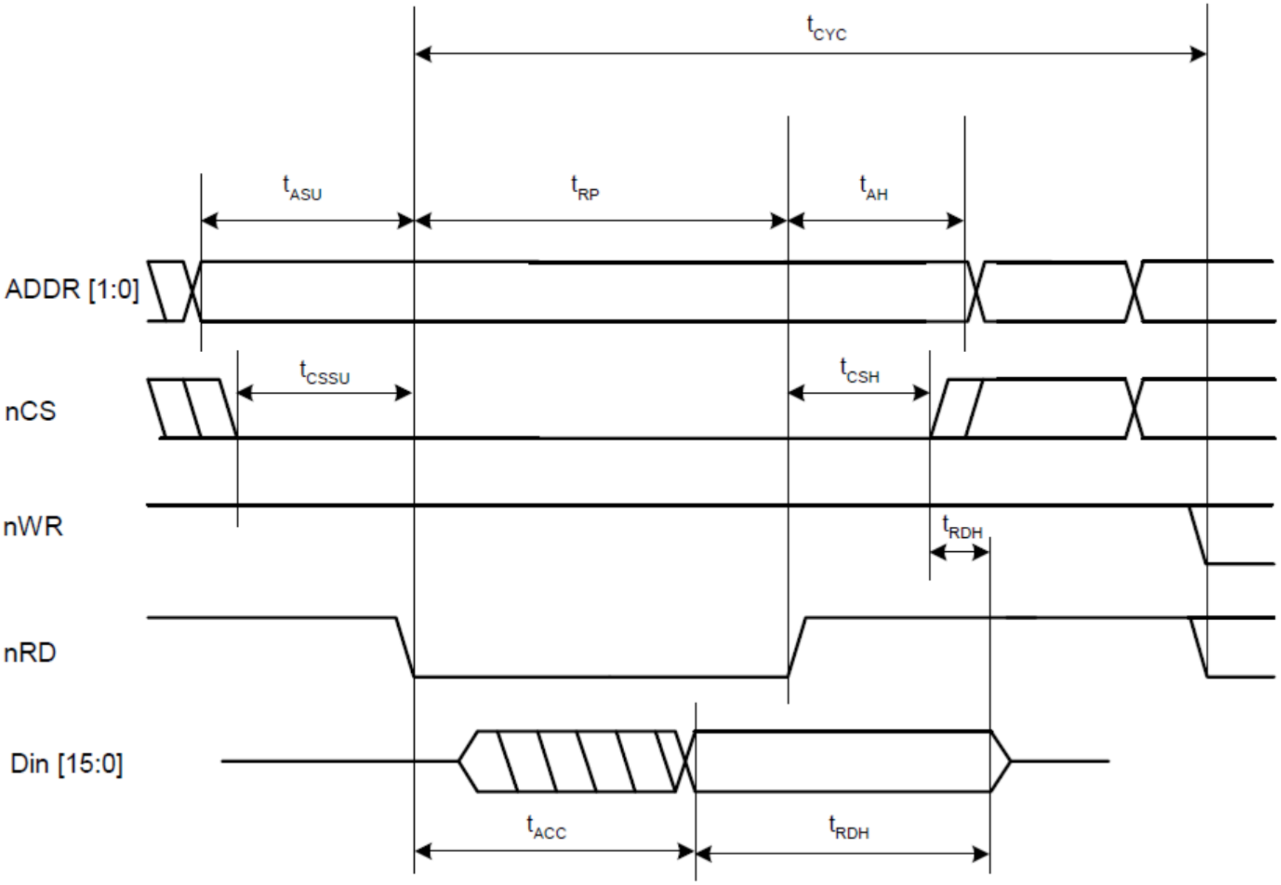
Host Processor
EZ-Host/OTG chip
HPI port
HPI_nCS
HPI_A[1:0]
HPI_nWR
HPI_nRD
HPI_INTR
HPI_D[15:0]
chip select
address
read
write
outgoing mailbox interrupt
data
incoming mailbox interrupt
incoming mailbox register
outgoing mailbox register
DMA memory address counter
DMA data buffer/ registers
SL16 CPU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

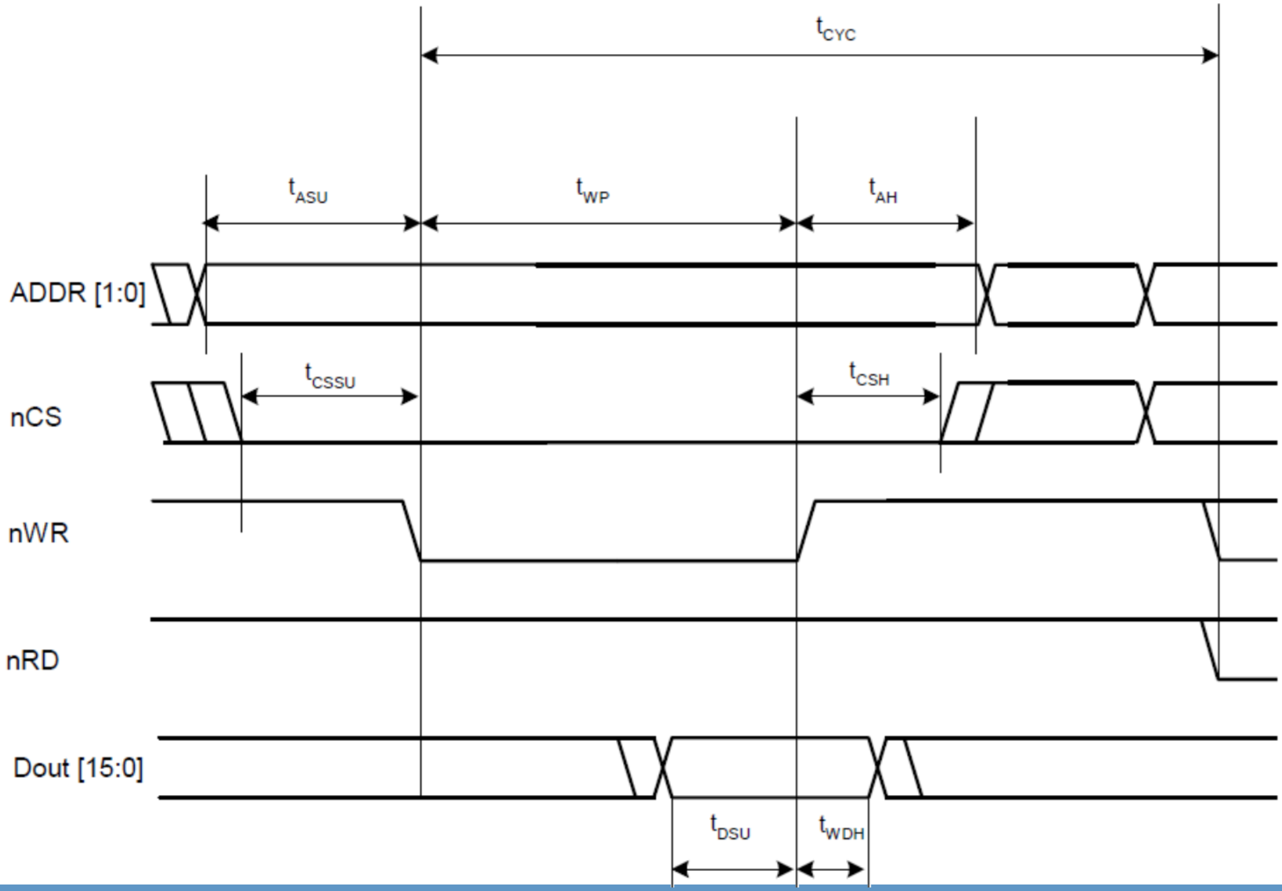
IO Read Timing via HPI

- 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



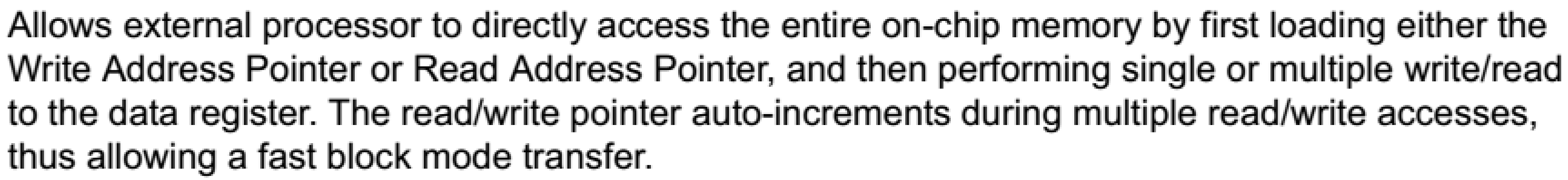
IO Write Timing via HPI

- 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

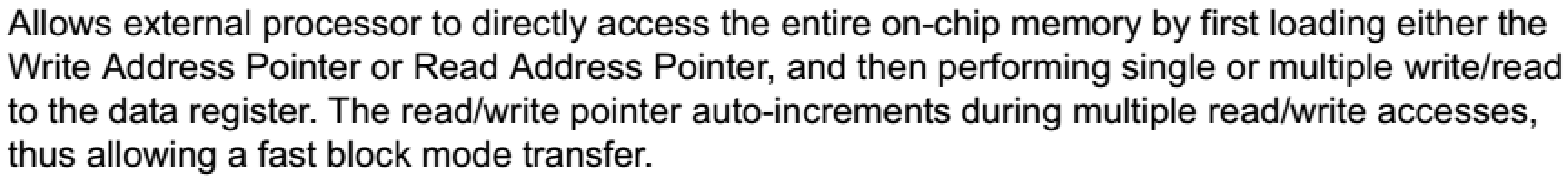
- USB read operation is defined as follows (from BIOS manual under HPI)
  - First set address register (0b10) to address (in EZ-OTG memory space)
  - Read data out of data register (0b11)
- Address may auto increment (therefore, you will sometimes see multiple `IO_read()` following a `USBRead()`
- Not necessary to give EZ-OTG command through mailbox (it assumes you want to read memory location if you follow procedure)



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 assumes you want to read memory location if you follow procedure)

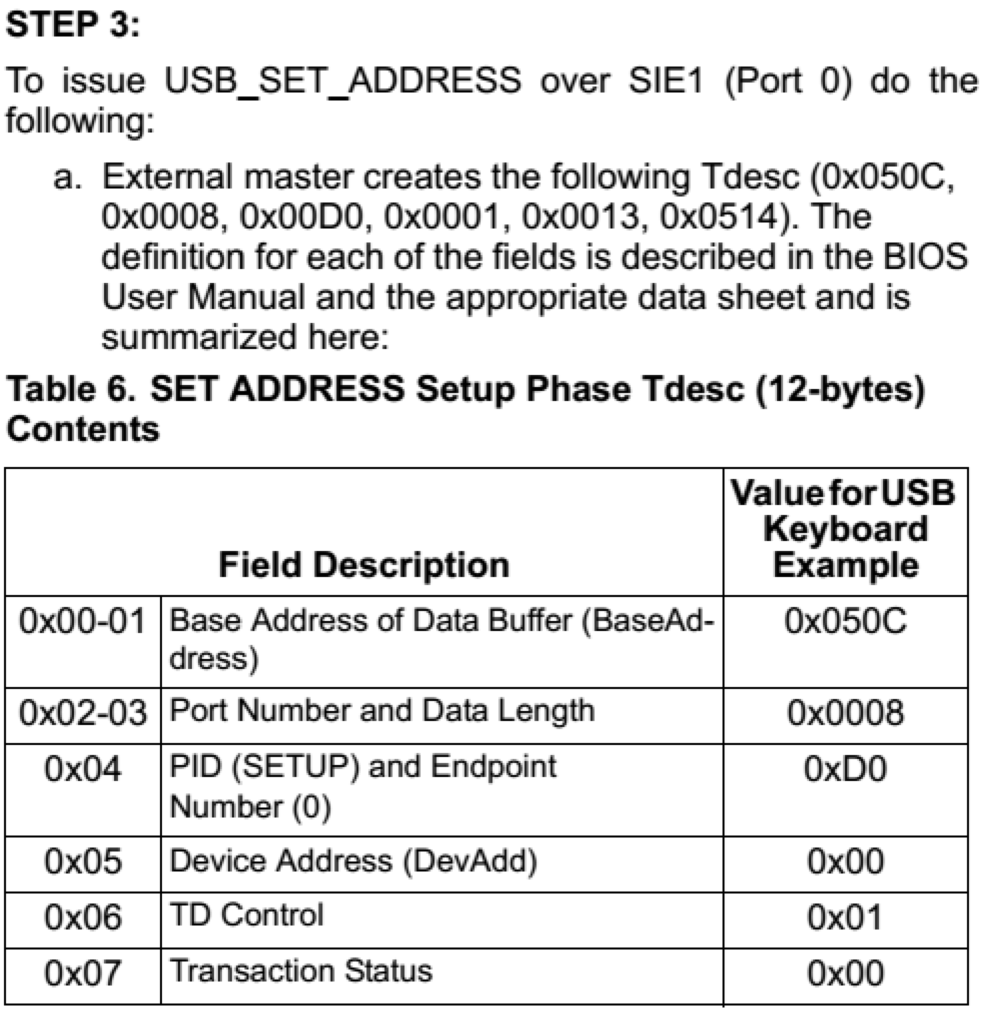


Allows external processor to directly access the entire on-chip memory by first loading either the Write Address Pointer or Read Address Pointer, and then performing single or multiple write/read to the data register. The read/write pointer auto-increments during multiple read/write accesses, thus allowing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

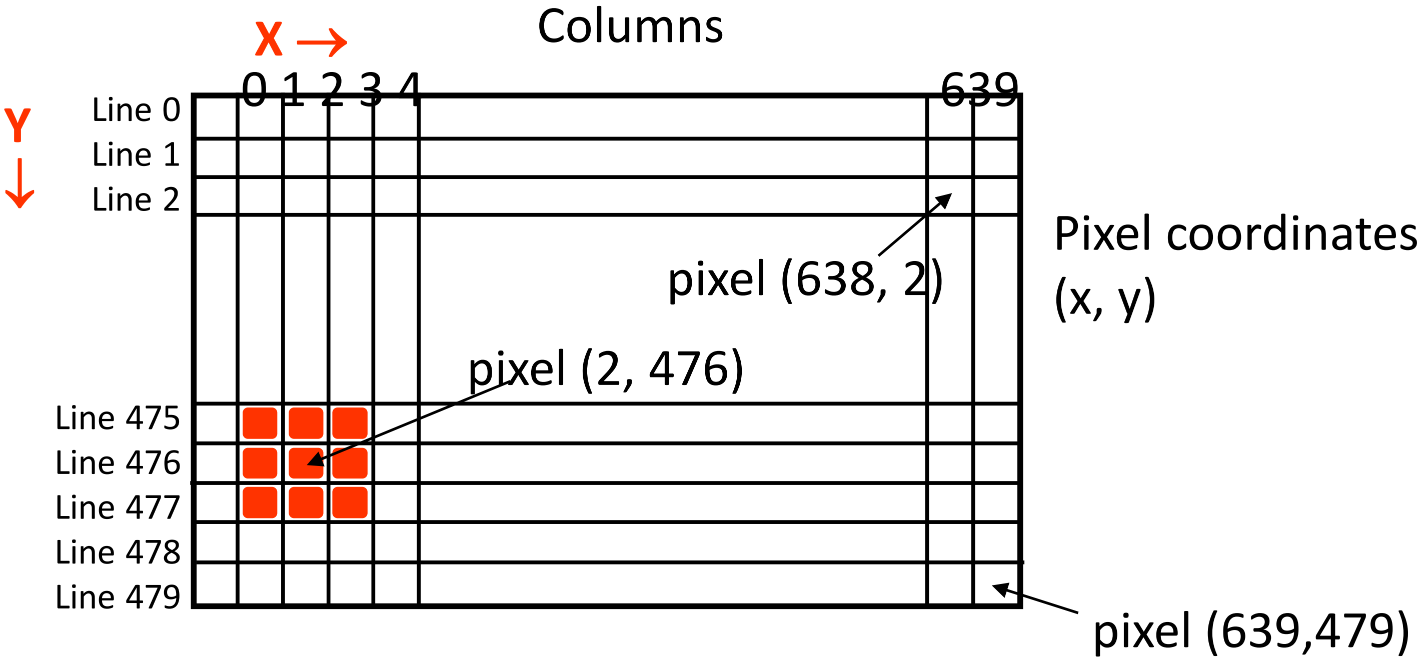
Table 6. SET ADDRESS Setup Phase Tdesc (12-bytes) Contents

Field Description		Value for USB Keyboard Example
0x00-01	Base Address of Data Buffer (BaseAddress)	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!

VGA Monitor Operation

- 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
- A Shape can be defined by specifying a boundary around a center. In the previous example, the center is (2, 476) and the box is defined *Center ± Size*. For Size=1 all pixels in the box satisfy: **(X≥2-1) AND (X≤2+1) AND (Y≥476-1) AND (Y≤476+1)**



Pixel coordinates are: (x, y)