

Disclaimer

The document is written where each quoted question has answer below it. This is intended to help quicker and easier grading. It has been *best-effort*, but there is no guarantee, that the answer will only appear under that specific question. Please grade each question on all the content appeared in the report.

Introduction

MicroBlaze is a configurable 32-bit RISC soft processor core for Xilinx FPGAs, including the Spartan 7. It supports adjustable features, interfaces with various memory types and I/O peripherals. For week 2 design, we connect SPI and various GPIO modules to facilitate its communication with MAX3421E chip that reads USB devices. We also instantiate few system verilog module and IPs that enables the generation of VGA (therefore HDMI) signal. With this design, we could output the HDMI signal to a screen showing a ball and a background, and use keyboard to control the movement of the ball.

Module Description

See Lab 6 Module pdf instead

Microblaze System

Describe in Lab 6.1 how the I/O works.

In my case,

Input / Output	Connected Pins
16 switches	The GPIO pin of AXI GPIO (lower 16 bytes)
1 push button (as "add" operation)	The GPIO pin of AXI GPIO (17th bytes byte)
16 LEDs	The GPIO 2 pin of AXI GPIO

The AXI GPIO module is connected to AXI Interconnect module via AXI bus (communicated using AXI4 protocol). AXI Interconnect module will "connects one or more AXI memory-mapped Master devices to one or more memory-mapped Slave devices". That means, the input/output connected to GPIO module is memory mapped to Microblaze processor's memory space. In my case, the GPIO gets mapped to `0x40000000`, according to the GPIO manual, `GPIO_DATA` has an offset `0x0000` and `GPIO2_DATA` has `0x0008`. That means, the 16 switches and 1 push button value could be read from the `0x40000000 + 0x0000` with proper masking, and 16 LEDs could be set by writing to `0x40000000 + 0x0008`. We could write a for loop in the software part later to interact with these memory value later.

Describe in words how the MicroBlaze interacts with both the MAX3421E USB chip and the ball motion components.

Utilizing the SPI protocol, the MicroBlaze communicates with the MAX3421E USB chip via an instantiated AXI Quad SPI in the block design. The AXI Quad SPI module exposes four SPI pins (SS/MOSI/MISO/SCLK) used for SPI communication. It is connected to Microblaze using AXI4 bus, which make it a relay between the communication between MicroBlaze (master) and the MAX3421E USB chip (slave).

This chip polls the USB keyboard, storing results in its 256-byte double-buffered FIFO memory (64 bytes each for send and receive). In host mode, the chip's 32 registers can be accessed using SPI. Specifically, the SNDFIFO Register (0x02) lets us write, while the RCVFIFO Register (0x01) allows **reading from the chip's FIFO, enabling retrieval of keyboard inputs from the SNDFIFO register.**

The chip additionally provides 2 pins, the `usb_int` and `usb_rst`, these pins are connected to Microblaze using two AXI GPIO mentioned before. The `usb_int` is interrupt issued by USB chip to allow the handle of incoming USB data timely. The `usb_rst` are issued from Microblaze to reset the USB chip.

The block design also has a AXI Timer module. USB has many timeouts that require timekeeping in milliseconds. The timer allows the MicroBlaze core to keep track of these specific timing (via the interrupt)

CPU speed of 100MHz) have passedThe Microblaze will have software logic that extract the keycode from the above mentioned operation, and output the 8 keycodes (actually only 6 since that is the max number of keys the USB keyboard could support) to the GPIO and GPIO2 pin of the `gpio_usb_keycode` AXI GPIO module. The ball motion components has two internal registers `Ball_X_Motion` and `Ball_Y_Motion` (the Ball's velocity in X and Y direction). The Microblaze provide 6 keycodes through the GPIO module, but only the first keycode is used by the ball motion component. (the lower 8 bit of the first GPIO pin). For every `frame_clk` (60 Hz), it will update these two velocity according to this keycode with respect to the "wall", and then update its output `BallX` and `BallY` (the position of the ball).

Describe in detail the VGA operation, and how your Ball, Color Mapper, and the VGA controller modules interact.

The window size we will use is 640x480. To generate the VGA signal, we send our pixel data from the top-left pixel, row by row, to the bottom-right pixel. The VGA specification requires Horizontal Blanking Interval and Vertical Blanking Interval that used to separate the rows and frames, and Video display enable signal that indicates whether the current pixel is displayed (this should be low during either blanking interval).

The VGA controller internally keeps two values `hpxels` and `vlines` that records the x and y value of current pixel we should output. The VGA controller receives `pixel_clk` as the clock for its operation. Since we want to run our screen at 60Hz, and there are 800x525 number of pixels (the blanking interval is added), we should output pixel at a rate roughly $60 \times 800 \times 525 = 25\text{MHz}$, which is generated using the clocking wizard.

During every `pixel_clk`, the VGA controller moves forwards current pixel coordinate, wrapping around if needed (modulo 800 / 525). It generates (lowered them) horizontal pulse (When `x = 656~752`) and vertical pulse (When `y = 490~491`), and `vde` signal when the pixel coordinate is out of the bound of `640x480`. It expose its `hpxels` to `vlines` to other components as well.

The Ball uses a `frame_clk` of (60 Hz), since it's the framerate we want. Coincidentally, since the vertical sync pause are lowered only per `800x525` pixel, the vertical sync actually lower at rate of `60Hz`. Since it's active low, we take the inverse of and that means `frame_clk = ~vsync`. The operation of the ball component is detailed at previous section.

The color mapper will take the current balls position and ball size (generated by Ball component), and the current pixel we are drawing (given VGA controller). If the current pixel we are drawing is inside the ball (a distance check basically), it will output a yellowish color. Otherwise it will output a gradient depending on the pixel's position. The color is represented by 4-4-4 RGB value.

Describe the VGA-HDMI IP, how does HDMI differ from VGA, how are they similar?

The VGA-HDMI IP takes the color output from the color mapper, and the vertical / horizontal sync pulse from the VGA controller, and various clock signal from the clock wizard, and it output the TMDS (Transition Minimized Differential Signaling) signal that requires for the HDMI transfer. (More description about specific pin in Lab 6 Module)

In the context of the HDMI, the data signal is 3-bit wide, and

- **Channel 0:** Carries data for the Blue color and, depending on the pixel format, may also carry a bit of audio data.
- **Channel 1:** Carries data for the Green color, vertical & horizontal sync signals, and, depending on the pixel format, may also include some audio data.
- **Channel 2:** Carries data for the Red color and, depending on the pixel format, may also include a bit of audio data.
- **Clock Channel:** Helps in the synchronization of data channels.

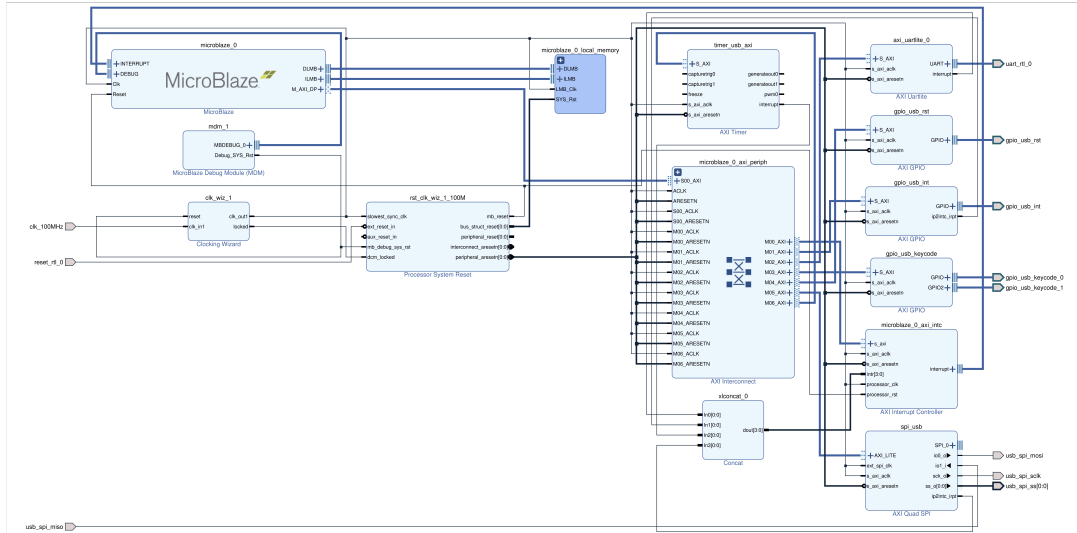
Therefore the IP outputs 4 pins (Channel [3:0] P/N, Clock P/N).

Both VGA and HDMI are standards for transmitting video signals from a source to a display, with each having a few connector variants. However, while VGA is limited to video transmission, HDMI can convey both video and audio. Additionally, HDMI offers support for higher resolutions than VGA, thanks to its use of TMDS technology.

From what we seen from the IP, the most significant difference between HDMI and VGA is the use of TMDS signal, it could support higher data rates. It utilizes differential signaling, transmitting signals over two conductors with opposite voltages. This offers noise immunity by rejecting external interferences and lowers electromagnetic emissions for reduced EMI. TMDS also minimizes signal transitions to cut down on EMI and maintain a balanced signal, facilitating clock recovery and minimizing data errors. It also uses a dedicated clock channel for data synchronization instead of using hsync and vsync. Because better noise immunity and less data errors, TMDS could clock higher that transfer more data at higher rate.

Note, there are also a `pix_clk_locked`, since this IP requires a 125 MHz clock, when the input clock arrives, the clock wizard usually takes some time (some cycles) before it could generate and stabilize the output 125MHz clock. The clock wizard uses `locked` signal to notify this IP when to start processing this HDMI signal, so that proper HDMI output could be given.

Top Level Block Diagram



Software

Written description of the SPI protocol and how it operates in the context of the MAX3421E.

The Serial Peripheral Interface (SPI) protocol is a synchronous serial communication method, commonly employed to connect microcontrollers to peripherals. Within the context of the MAX3421E USB chip:

1. **Communication Mechanism:** The MicroBlaze interfaces with the MAX3421E USB chip through the SPI protocol using an AXI Quad SPI block design.
2. **Pins and Functionality:**
 - **CS/SS (Chip/Slave Select):** Active low signal that indicates to the MAX3421E that it should be ready for communication.
 - **MOSI (Master-Out-Slave-In):** This pin sends data from the MicroBlaze (master) to the MAX3421E (slave).
 - **MISO (Master-In-Slave-Out):** Through this pin, the MicroBlaze receives data from the MAX3421E.
 - **SCLK (Serial Clock):** Synchronizing clock signal generated by the master.
3. **Data Transfer:**

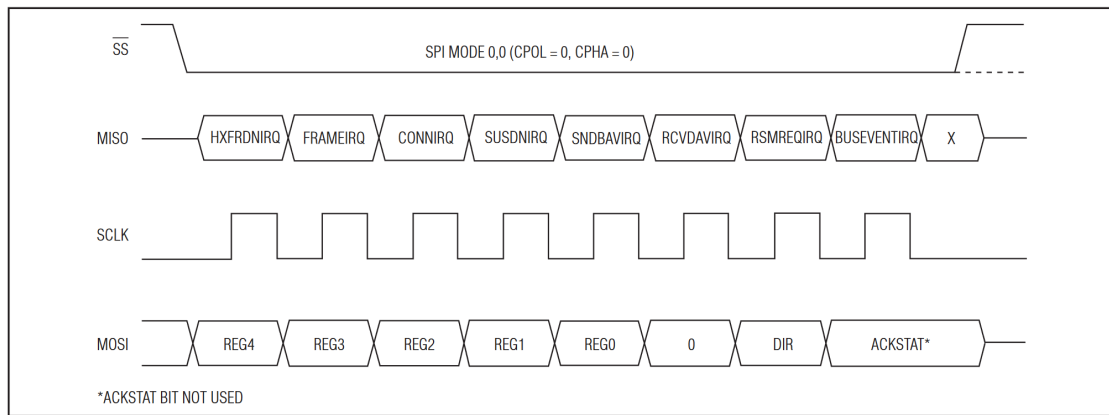


Figure 18. SPI Port in Full-Duplex Mode (Host Mode)

- The data transfer initiates when the master lowers the SS pin.
- Command byte sequence consists of a 5-bit register address, a 0 bit (indicating read=0/write=1 operation). The ACKSTAT bit is not utilized and can be set to 0.
- After sending the command, the master raises the SS pin. When reading, the master fetches subsequent bytes via MISO. When writing, data is sent through MOSI. (Between each byte the SS needs to be lowered first, then raised at end of the byte)
- For the MAX3421E, reading/writing *multiple bytes* for RCVFIFO/SNDFIFO Registers will read/write the internal chip FIFO buffer byte by byte.

The SPI protocol facilitates a synchronized data exchange between the MicroBlaze and the MAX3421E USB chip, enabling the latter to communicate USB device states and information to the former.

One of the INMB questions asks about the blinker code, but you must also describe your accumulator.

```
volatile uint32_t* sw_gpio_data = (uint32_t*) (XPAR_AXI_GPIO_0_BASEADDR +
0x0008);
uint32_t sum = 0;
uint32_t stop_rapid_fire = 0;

while (1) {
    volatile uint32_t accumulate_button = (*sw_gpio_data) & 0xFFFF0000;

    if (accumulate_button && !stop_rapid_fire) {
        sum += ((*sw_gpio_data) & 0x0000FFFF);
        if (sum > 65535) {
            sum -= 65536;
            printf("Overflow!\n");
        }
        stop_rapid_fire = 1;
    }
    if (!accumulate_button) stop_rapid_fire = 0;
    *led_gpio_data = sum;
}
```

The `sw_gpio_data` is the data from the current switch input (at GPIO 2), we start our accumulator `sum` with 0. In the loop, we check if the `accumulator` button is pressed (it's the 17 bit of the `sw_gpio_data` declared in the top level sv file). If the `accumulate_button` is pressed, we add the switch data to the sum, which is (the lower 16 bit of the `sw_gpio_data`). If we find that the sum is 16-bit overflowed, we mimic the overflow behavior (subtract 65536) and print to notify this. We then assign the sum to the `*led_gpio_data` (at GPIO) which will turn corresponding LED on. There are some logic involves with `stop_rapid_fire`, this is to prevent the `accumulate` button constantly trigger the summation. It will only allow the summation once per press.

Describe the purpose of each function you filled in in the C code (you do not need to describe the functions you did not modify).

There are four functions

```
void MAXreg_wr(BYTE reg, BYTE val)
BYTE* MAXbytes_wr(BYTE reg, BYTE nbytes, BYTE* data)
BYTE MAXreg_rd(BYTE reg)
BYTE* MAXbytes_rd(BYTE reg, BYTE nbytes, BYTE* data)
```

All four function access the register with address `reg` of `MAX3421E` via SPI protocol.

The first function write one byte `val` to the register with address `reg` of `MAX3421E`. The second function write `nsize` bytes from `data` to the register with address `reg`, it returns a pointer to a memory position after last written. The third function read one byte from the register `reg` and return that byte. The fourth function reads `nbytes` from the register `reg` and store the them starting at `data` pointer, it returns a pointer to a memory position after last written

INMB Questions

You should do some research and figure out what are some primary differences between the various presets which are available.

The three presets are (the difference are bolded)

- Microcontroller preset: Microcontroller preset suitable for microcontroller designs. **Area optimized, with no caches and debug enabled.**
- Real-time preset: Real-time preset geared towards real-time control. **Performance optimized, small caches and debug enabled, most execution units.**
- Application preset: Application preset design for high performance applications. **Performance optimized, large caches and debug enabled, and all execution units including floating-point.**

The other options are:

- Minimum Area: The smallest possible MicroBlaze core. **No caches or debug.**
- Maximum Performance: Maximum possible performance. **Large caches and debug, as well as all execution units.**
- Maximum Frequency: Maximum achievable frequency. **Small caches and no debug, with few execution units.**

- Linux with MMU: Settings suitable to get high performance when running Linux with MMU. **Memory Management enabled, large caches and debug, and all execution units.**
- Low-end Linux with MMU: Settings corresponding to the MicroBlaze Embedded Reference System. Provides suitable settings for Linux development on low-end systems. **Memory Management enabled, small caches and debug.**
- Typical: Settings giving a reasonable compromise between performance, area, and frequency. Suitable for standalone programs, and low-overhead kernels. **Caches and debug enabled.**
- Frequency Optimized: Designed to provide all MicroBlaze features, **including MMU, while still achieving high frequency** by utilizing the frequency optimized 8-stage pipeline.

Note the bus connections coming from the MicroBlaze; is it a Von Neumann, “pure Harvard”, or “modified Harvard” machine and why?

MicroBlaze has separate instruction (ILMB - Instruction Local Memory Bus) and data (DLMB - Data Local Memory Bus) interfaces, it can also access memory outside of these local interfaces through its M_AXI_DP (AXI memory interfaces) connections, which can be used for both instruction and data access, making it more like a Modified Harvard architecture.

What does the “asynchronous” in UART refer to regarding the data transmission method?
What are some advantages and disadvantages of an asynchronous protocol vs. a synchronous protocol?

The term "asynchronous" in UART refers to the fact that the communication does not rely on an external clock signal to synchronize the transmission of data between two devices.

Asynchronous Communication: Advantages (1) No external clock needed. (2) Generally simpler in structure. Disadvantages: (1) Baud rate agreement is essential. (You need to know it) (2) Potentially more error-prone.

Synchronous Communication Advantages (1) Higher data efficiency. (2) Can achieve higher speeds. (3) Less error-prone. Disadvantages (1) Requires external clock. (2) Typically more complex.

You should have learned about interrupts in ECE 220, and it is obvious why interrupts are useful for inputs. However, even devices which transmit data benefit from interrupts; explain why. Hint: the UART takes a long time (relative to the CPU) to transmit a single byte.

UART takes a long time to transmit a single byte. Without interrupts, the CPU would have to constantly poll the UART to check whether it's ready to send the next byte. This is known as polling and it's highly inefficient because the CPU wastes a lot of cycles checking the status of the UART. With interrupts, the CPU can perform other tasks and only attend to the UART when it's actually ready, i.e., when the UART raises an interrupt.

Why are the UART and LED peripherals only connected to the data bus?

The processor has instruction and data buses. UART and LED peripherals are memory mapped to the data bus because they handle data transfer and I/O operations, not instruction fetches. The instruction bus is specifically for fetching instructions for the CPU to execute, while the data bus handles data read and write operations. Connecting UART and LED to the instruction bus is

invalid and will cause error.

You must be able to explain what each line of this (very short) program does to your TA. Specifically, you must be able to explain what the volatile keyword does (line 18), and how the set and clear functions work by working out an example on paper (lines 30 and 33).

This volatile tells the compiler that the variable can change at any time without any action being taken by the code the compiler finds. It's typically used for memory-mapped peripheral registers to prevent the compiler from optimizing out repeated accesses to that address. In this context, it's ensuring that reads/writes to the LED's GPIO register actually happen and are not optimized away.

```
*led_gpio_data |= 0x00000001;
```

- This line turns ON the least significant bit (LSB) of the LED GPIO register. It's using the bitwise OR operation to set the LSB.
- Example:

```
LED GPIO register before: 0100 0000 0010 0000 0000 0000 0000 0000 (assuming
all LEDs are off)
OR operation with          : 0000 0000 0000 0000 0000 0000 0000 0001
Result                     : 0100 0000 0010 0000 0000 0000 0000 0001 (turns on
the LED associated with the LSB)
```

```
*led_gpio_data &= ~0x00000001;
```

- This line turns OFF the least significant bit of the LED GPIO register. It's using the bitwise AND operation combined with bitwise NOT operation to clear the LSB.
- Example:

```
LED GPIO register before: 0010 0010 0000 0000 0000 0000 0000 0001 (assuming
the LED associated with the LSB is on)
AND operation with        : 1111 1111 1111 1111 1111 1111 1111 1110 (this is
the bitwise NOT of 0x00000001)
Result                    : 0010 0010 0000 0000 0000 0000 0000 0000 (turns off
the LED associated with the LSB)
```

Look at the various segments (text, data, bss), what does each segment mean? What kind of code elements are stored in each segment? Also note the size of the executable in bytes. Remember that we configured 32Kbytes of on-chip memory to use as our program memory. Why does the provided code, which does very little, take up so much program memory? Hint: try commenting out some lines of code and see how the size changes

The **text segment** is where the executable code of the program resides, holding the instructions meant for CPU execution. The **data segment** is dedicated to storing initialized global and static variables. Within the data segment, there's a specific sub-segment called the **BSS segment** (Block Started by Symbol), which is designed for uninitialized global and static variables. These uninitialized variables are automatically set to zero. The provided code is big because we call some library functions like `init_program`, `printf`, `sleep`, and the implementation of these function is

actually really big (there are also static variable declared in each file), and these library function is the reason why it takes so much program memory. (Removing these function, say `printf()` could reduce the memory taken significantly)

Make sure you understand the register map on page 10. If the base address is 0x40000000, how would you access GPIO2_DATA (for example?).

You should access `0x40000000 + 0x0008` for GPIO2_DATA, the general formula is that `memory_address = base_address + register_offset`

Table 2-4: Registers

Address Space Offset ⁽³⁾	Register Name	Access Type	Default Value	Description
0x0000	GPIO_DATA	R/W	0x0	Channel 1 AXI GPIO Data Register.
0x0004	GPIO_TRI	R/W	0x0	Channel 1 AXI GPIO 3-state Control Register.
0x0008	GPIO2_DATA	R/W	0x0	Channel 2 AXI GPIO Data Register.
0x000C	GPIO2_TRI	R/W	0x0	Channel 2 AXI GPIO 3-state Control.
0x011C	GIER ⁽¹⁾	R/W	0x0	Global Interrupt Enable Register.
0x0128	IP IER ⁽¹⁾	R/W	0x0	IP Interrupt Enable Register (IP IER).
0x0120	IP ISR ⁽¹⁾	R/TOW ⁽²⁾	0x0	IP Interrupt Status Register.

Design Resources and Statistics

LUT	2787
DSP	9
Memory (BRAM)	8
Flip-Flop	2592
Latches*	0
Frequency (MHz)	117.6
Static Power (W)	0.075
Dynamic Power (W)	0.384
Total Power (W)	0.459

Conclusion

With this design, we could output the HDMI signal to a screen showing a ball and a background, and use keyboard to control the movement of the ball. All my design works (yay)

There are few places where it's indeed a bit unclear or misleading. It's hard to find where to open the "Address Map", the tutorial needs to be more detailed on that. Also, the italicized question should to be put together, so I could make sure I indeed answered all the questions. Also, the Lecture 17 still has the wrong function, we should use `xSpi_SetSlaveSelect` instead of `xSpi_SetSlaveSelectReg`. Also, the lecture should mention that the `SetSlaveSelectReg` is active low (where `xSpi_SetSlaveSelect` is not)