CHULA ENGINEERING
Foundation toward Innovation

COMPUTER

Hardware Synthesis Laboratory

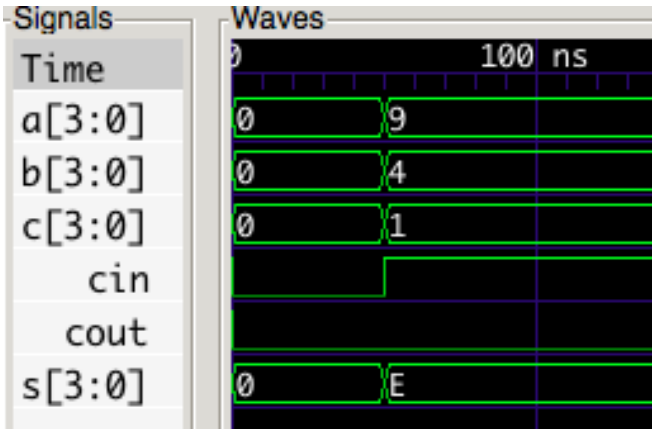
1st semester

Academic Year 2021

```
module fulladder (cout,s,a,b,cin);
    output cout;
    output s;
    input a,b,cin;

//assign {cout,s}=a+b+cin;
reg cout, s;
always @(a or b or cin)
begin
    {cout,s}=a+b+cin;
end

endmodule
```



Krerk Piromsopa, Ph. D.

Preface

Since the exploding complexity of digital electronic circuits in the 1970s, Hardware Descriptive Language (HDL) has evolved. With the rise of FPGA, Hardware Synthesis (together with HDL) has become a fundamental skill in Computer Engineering.

The department of Computer Engineering, Chulalongkorn University is among the first universities to teach VerilogHDL and Hardware Synthesis Lab. (We have done this for more than 20 years.) This course has evolved from an in-house FPGA board to the standardized developer boards. Generations of students have enjoyed (and weeped) for this class. The success of this class is evidenced by a number of students that have graduated and worked in semiconductor industries (Intel, Amd, IBM, etc.)

As a hardware design enthusiasm, I recreated this Lab book to ease the study of Hardware Descriptive Language (HDL) and Hardware Synthesis Lab. My personal belief is the more you understand hardware is the better you are as a programmer.

For those that may want to further their study in digital design, this is one chance for you to show your skill. Note that we have 2-3 persons per academic year that are natural at hardware descriptive language. Most (if not all) of them usually got a Ph.D. in Hardware-related fields and work in the industries. For those of you that may not be comfortable with hardware design, the minimum requirement for this class is to understand and use the digital tools.

Hopefully, students will learn something useful for their future from this class

Enjoy,

Krerk Piromsopa, Ph.D.

Associate Professor

Department of Computer Engineering

Chulalongkorn University

(Updated: August 9, 2021)

Table of Contents

Laboratory 1: Introduction to VerilogHDL and Digital Simulation

Objectives

1. Get students to familiar with the simulation tool (Vivado)
2. Demonstrate the basic of Verilog simulation and waveform output
3. Able to explain structural model and behavioral model
4. Able to explain the differences between blocking and non-blocking assignments

Background

In this lab, you will learn the fundamentals of VerilogHDL and Digital Simulation using the Vivado Design suite. Firstly, please download the free (webpack) version of Vivado Design Suite from Xilinx Web site[1]. The whole download is about 20GB. Alternatively, you may download it from the department server[2] . Should you have trouble finding a machine for installing the software, please contact the instructors. A (virtual) machine can be provided for you to remotely work with the tool. However, you may still have to install the Lab Edition to download the design to the FPGA board. For more information about the installation and Vivado IDE, please what the Xilinx tutorials' videos[3].

Please watch the demonstration video on how to use the simulation.

[1] https://www.xilinx.com/products/design-tools/vivado.html
[2] https://mis.cp.eng.chula.ac.th/krerk/teaching/2018s2-HWSynLab/
Username: student, password: HWSynLab
[3] https://www.xilinx.com/products/design-tools/vivado.html#video

Exercises

1. Complete the following 1-bit full adder and a test bench to validate such design by simulating all possible inputs. Use the Vivado tool to simulate and validate the design.

```
module fullAdder(cout, s, a, b, cin);

output cout;
output s;
input a;
input b;
input cin;

reg cout, s;

always @(       )
begin

end

endmodule
```

```
`timescale 1ns/1ns

module tester;

      reg a,b,cin;
      wire cout,s;

      fullAdder a1(cout,s,a,b,cin);

initial
begin
      //$dumpfile("time.dump");
      //$dumpvars(2,a1);
      $monitor("time %t: {%b %b} <-
{%d %d %d}", $time,cout,s,a,b,cin);
      #0;
      a=0;
      b=0;
      cin=0;

      //…...
      $finish;
end
endmodule
```

2. What would happen if we replace the always block of the full adder in question 1 with the following module? Would it give the same result? Please run the test bench and provide your analysis.

```
module fullAdder(cout, s, a, b, cin);

output cout;
output s;
input a;
input b;
input cin;

assign {cout,s} = a + b + cin;

endmodule
```

3. Please modify the following latch to be a (positive edge triggering) flip flop with asynchronous reset. Please also modify the test bench to validate your design.

```
`timescale 1ns/1ns

module DFlipFlop(q,clock,nreset,d);

output q;
input clock,nreset,d;

reg q;

always @(clock)
begin
      if (nreset==1)
            q=d;
      else
            q=0;
end
endmodule
```

```
module testDFlipFlop();
reg clock, nreset, d;
DFlipFlop D1(q,clock,nreset,d);
always
      #10   clock=~clock;

initial
begin

//$dumpfile("testDFlipFlop.dump");
      //$dumpvars(1,D1);
      #0 d=0;
      clock=0;
      nreset=0;
      #50 nreset=1;
      #1000 $finish;
end
always
      #8 d=~d;
endmodule
```

4. What are the differences between the 2 provided designs? Please write a test bench to show your analysis.

```
module shiftA(q,clock,d);
output [1:0] q;
input clock,d;

reg [1:0] q;

always @(posedge clock)
begin
      q[0]=d;
      q[1]=q[0];
end
endmodule

module shiftB(q,clock,d);
output [1:0] q;
input clock,d;

reg [1:0] q;

always @(posedge clock)
begin
      q[0]<=d;
      q[1]<=q[0];
end
endmodule
```

5. Please answer the following questions and submit (in PDF format) to CourseVille on Friday before 23:59 (midnight).
    1. Please draw a schematic representing the logical blocks of both shiftA and shiftB in exercise 4.
    2. What is the difference between blocking and non-blocking assignments?
    3. Is it possible to apply parameters to the design in exercise 4 to create shiftRegister with any number of bits? If Yes, please explain how.

Laboratory 2: Time-Division Multiplexing and Clock Divider

Objectives

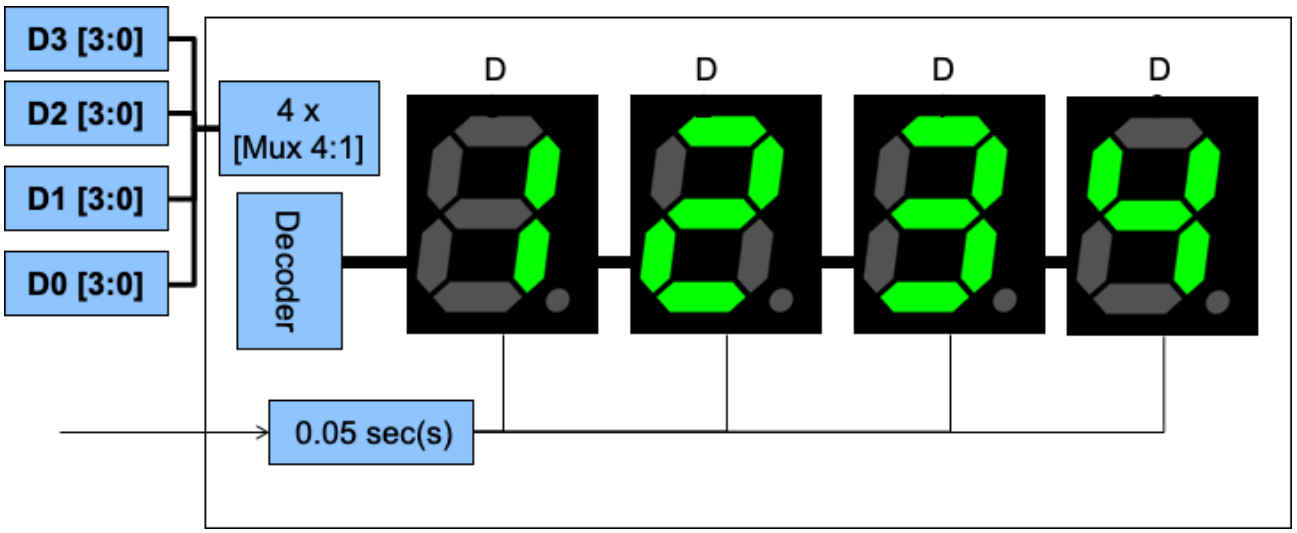
1. Get students to familiar with the synthesis tool (Vivado)
2. Synthesis FPGA
3. Able to explain time-division multiplexing
4. Able to design clock divider
5. Able to use a language template.

Background

TDM

To drive a seven-segment display (whether it is common anode or common cathode), each digit would require 9 wires (a to g, dot, common ground or common vcc). With several digits of seven-segment display, the number of wires would intuitively multiplied. For example, 4 digits of seven-segment displays may require up to 33 wires (a to g and dot for each digit with a sharing common wire). It is not practical to have so many wires. To share (reduce) physical wires, Time-Division Multiplexor is introduced.

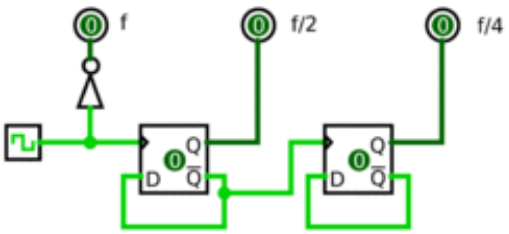
With time-division multiplexing, we can share 8 wires (a to g and dot) among the displays. Only a digit will be active at a time. If the segments turn on and off at the appropriate rate (I.e. 15 frames per second or more), the observer would see it as if all segments are on at the same time.--hence the term time-division multiplexing. This way, four digits of seven-segment displays can be connected with only 12 wires (8 from a to g and dot + 4 for activating digits) .



For more details about time-division multiplexing, please watch the demonstration video.

Clock Division

There are several ways to divide high frequency clocks into slower clocks. A simple solution is to cascade D flip flops (or even T flip flops) together by feeding ~Q0 to D0 and feed Q0 as a clock for D1 (and so on). Nonetheless, this is just one implementation of the clock division. You may use a counter to set and clear a bit as a clock division as well.



Language Templates

Vivado IDE tool comes bundled with language templates. Language templates are basically code snippets for HDL. You may access the language templates from ***menu > Tools > Language Templates***. A language template that might be useful for this lab is 7-segment encoding.

Exercises

1. Use your knowledge from clock division and time-division multiplexor to display a 4-digit hexadecimal number (0x1234) to the seven-segment display of the BASYS 3 board. Your design should be modularized (You can save the component for reuse later). There should be at least 3 modules: clock divider, hex (or bcd) to 7-segment encoder and 7-segment TDM.

2. Please answer the following questions and submit (in PDF format) to CourseVille on Friday before 23:59 (midnight).
    a. Is the 4-digit seven-segment display on the BASYS 3 board a common anode for common cathode? Please explain.
    b. From the wiring of the board, which logic do you have to assign to the 7-segment pins (a to g and dot) to turn the LED on.
    c. Given that the clock of the BASYS3 is around 10ns, how many bits do you have to divide the clock with to get the appropriate clock for the TDM. Please provide your analysis (calculation).

    Hint

    1. Use the BASYS 3 XDC[4] file as a base constraint file.
    2. Read the datasheet to determine the interconnection in the board.
    3. Use the language templates for 7-segment encoder

[4] https://mis.cp.eng.chula.ac.th/krerk/teaching/2018s2-HaWSynLab/downloads/Basys-3-Master.xdc

Laboratory 3: Counter and Switch (Debounce)

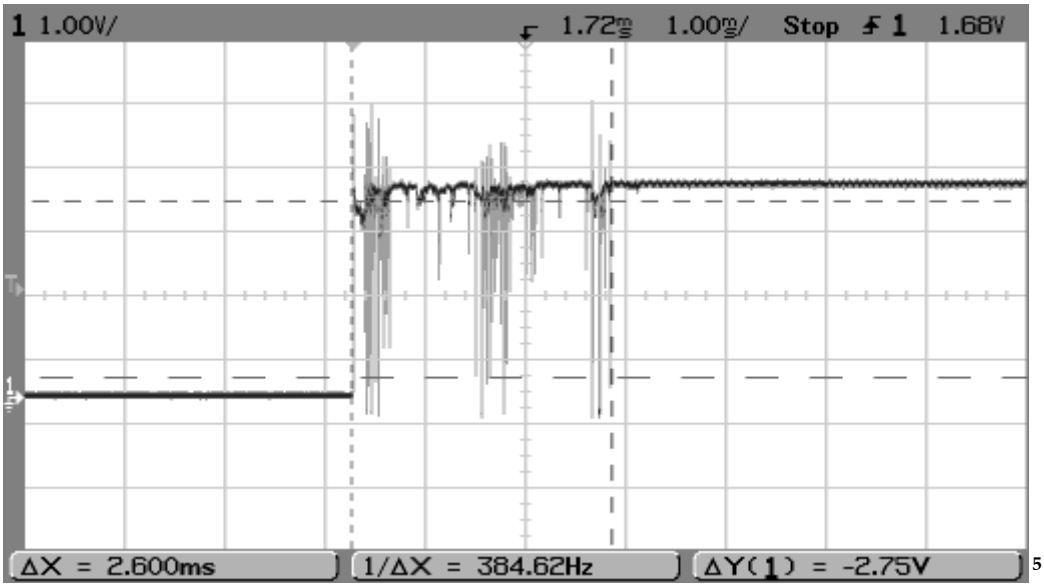
Objectives

1. Synthesis FPGA
2. Able to design debounce switch and input
3. Able to design up and down Counter

Background

Switch and Bounce

Mechanic switches and relays have a common issue called contact bounce (aka. chatter). Switch and relay contacts are usually made of metals. When the contacts strike together, their momentum and elasticity act together to cause them to bounce apart one or more times before making steady contact. (Imagine a ball falling on a fall, it would bounce several times before coming to a complete stop.)

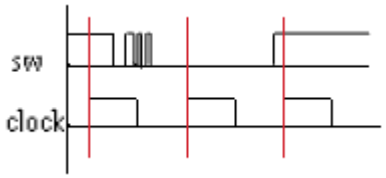


[5]

[5] Image taken from https://upload.wikimedia.org/wikipedia/commons/thumb/a/ac/Bouncy_Switch.png/400px-Bouncy_Switch.png

There are several ways to debounce. Debouncing methods include using capacitor, SR Latch, or Low-pass filtered schmitt trigger. However, those methods usually required special hardware.

To debounce without using special hardware, we can use software methods by resampling for input several times.



Please also note that there is a metastable issue. To avoid this, it is generally advised that two D flip flops be placed between the input and the digital circuit.

Exercises

1. Create an up/down 1-digit BCD counter with 4-bit outputs (DCBA) and 1 overflow output (cout), 1 borrow (bout) and 6 inputs (up, down, set9, set0,clock). Write a simulator to show that the counter functions correctly.

2. Create a single pulser with one input, clock, and one output. Write a simulator to show that the single pulser works correctly.

3. Use 4 counters from exercise 1 to create 4 digits BCD counters. Connect all displays to 4 digits seven-segment displays. (Use the display components from Laboratory II.) Use BTNU for set9 (set the number to 9999). Use BTNC for reset (set the number to 0000). Use SW0 for countdown by 1. Use SW1 for count up by 1. Use SW2 for countdown by 10. Use SW3 for count up by 10. Use SW4 for countdown by 100. Use SW5 for count up by 100. Use SW6 for countdown by 1000. Use SW7 for count up by 1000. If the number is at 0000, a countdown would not decrease the number. If the number is at 9999, a count up would not increase the number. Do not worry about the bounce at the moment. We will fix it in the next exercise.

4. Correct the bounce in exercise 2 by implementing a debounce component for each input.

5. Please answer the following questions and submit (in PDF format) to CourseVille on Friday before 23:59 (midnight).
    a. From the circuit diagram, the BTNx is active High or active Low? Please provide your analysis.
    b. What is a bounce? How do you programmatically debounce the input? Please provide your analysis.
    c. Please show your method for implementing a single pulser. (e.g. draw a state diagram, or verilogHDL code)

Laboratory 4: Memory

Objectives

1. Able to implement memory in HDL
2. Able to instantiate internal FPGA memory

Background

We would be soon implementing our first processor in the next Lab!. Now, you should have an understanding of how to implement a FSM with Verilog. What you are missing is how to implement a memory model on Verilog such that you can use it on your very first processor. We will be looking at read-only-memory (ROM), random-access-memory (RAM), and first-in-first-out (FIFO)

ROM

The first kind of memory you are going to implement is read-only-memory (ROM). So far, you have been using only Verilog for synthesizing registers (D-Flip Flop). However, it's costly to implement memory using purely registers. Field-programmable gate array (FPGA) manufacturers often include blocks of memory inside the FPGA such that you can use.

Typical ROM instantiation looks like the following

```
module rom case(
      (* synthesis , rom block = ”ROM CELLXYZ01” *)
      output reg [3:0] z ,
      input wire [2:0] a); // address- 8 deep memory


always@* begin // @(a)
      case (a)
      3’b000: z = 4’b1011;
      3’b001: z = 4’b0001;
      3’b100: z = 4’b0011;
      3’b110: z = 4’b0010;
      3’b111: z = 4’b1110;
```

```
      default : z =4'b0000 ;
      endcase
end
endmodule // rom case
```

The code above would generate ROM with 8 addresses, each address is 4 bits.

You might notice the synthesis suggestion keyword (* synthesis, rom_block = "ROM_CELLXYZ01" *). This tells the synthesis tools to try to use the dedicated ROM inside the FPGA instead of implementing it as a block of registers. The keywords may differ from one FPGA vendor from another. Note that most synthesis tools nowadays are smart enough to detect the access pattern that you can remove that out.

Second, the ROM in is actually asynchronous. You could make it a synchronous ROM by adding a clock, i.e.

```
module rom case(
      (* synthesis , rom block = "ROM CELLXYZ01" *) input clk ,
      output reg [3:0] z ,
      input wire [2:0] a); // address- 8 deep memory
always@(posedge clk)
      begin
      case (a)
      3'b000: z = 4'b1011;
      3'b001: z = 4'b0001;
      3'b100: z = 4'b0011;
      3'b110: z = 4'b0010;
      3'b111: z = 4'b1110;
      default : z =4'b0000 ;
      endcase
end
endmodule // rom case
```

Having the data inside your program is extremely inconvenient especially if you have a large file set of data. Verilog allows you to “read” data from a file.

```
// Verilog-2001 style
// ROM module using two dimensional arrays with
// memory defined in text file with $readmemb or $readmemh
// NOTE: This style can lead to simulation/synthesis mismatch
//       if the content of data file changes after synthesis
module rom_2dimarray_initial_readmem (
  output wire [3:0] z,
  input  wire [2:0] a);
  // declares a memory rom of 8 4-bit registers.
  //The indices are 0 to 7
  (* synthesis, rom_block = "ROM_CELL XYZ01" *)
  reg    [3:0] rom[0:7];
  // NOTE:  To infer combinational logic instead of a ROM, use
  // (* synthesis, logic_block *)
  initial \$readmemb("rom.data", rom);
  assign z = rom[a];
endmodule
```

The rom.data would look like this

```
1011 // addr=0
1000 // addr=1
0000 // addr=2
1000 // addr=3
0010 // addr=4
0101 // addr=5
1111 // addr=6
1001 // addr=7
```

RAM

Another useful primitive for the FPGA is random-access-memory (RAM). Again, as in the ROM case, we could have implemented RAM as a set of registers, but it's expensive and costly to do so. Typical FPGAs have dedicated areas for RAMs, (BlockRAM for Xilinx, Memory Block for Altera, etc.) The following code will generate RAM with 128x8 bits.

```
module SinglePortRAM (
inout wire [7:0] d, // Data In and Out
input wire [6:0] addr , // Address
input wire oe , // Output Enable
input wire clk , we) ;
(* synthesis , ram block *)

reg [7:0] mem [127:0];

always @(posedge clk)
if(we)
      mem[addr] <= d;

assign d = oe ? mem[addr] : 8'bZ;
endmodule
```

You may see the inout port in the example. The idea is that the port can be used as both input and output (at different times). To read, you have to assign Z to the wire before reading the data. To write, just connect the register to the wire. This line "assign d = oe ? mem[addr] : 8'bZ; " explains such a connection.

Block RAM

As you can see, we can write a HDL code to generate registers, and we can potentially implement a memory using it. However, a FPGA has a small number of these CLB, and it is a bit overkill since these logic can do much greater things than being just memory. So, most FPGA vendors have specialized memory units that we can use on these FPGAs. Each vendor has a different name, but for Xilinx, we call it Block RAMs. Typically, each bRAMs

has a size of 10-20Kbit depending on the FPGA, and each FPGA may have from ten to thousands of these bRAMs.

There are several ways to initiate bRAMs on Xilinx, but in general Xilinx Synthesis step can recognize if you are going to generate a bRAMs from a pattern following pattern

```
parameter RAM WIDTH = <ram width >;

parameter RAM ADDR BITS = <ram addr bits >;

reg [RAMWIDTH-1:0] <ram name> [(2**RAMADDRBITS)- 1:0]; reg [RAM WIDTH-1:0] <output
data >;

<reg or wire> [RAMADDRBITS-1:0] <address>;

<reg or wire> [RAMWIDTH-1:0] <input data >; always @(posedge <clock>)

if (<ram enable>) begin

if (<write enable>) begin

<ram name>[<address >] <= <input data >; <output data> <= <input data >;

end else

<output data> <= <ram name>[<address >]; end
```

Note that bRAMs is not the only type of construct that the Synthesizer can recognize. There are many other types of construct. Some of these are even more complicated. So most FPGA vendors has a so called language template. For Xilinx, you can find these out in the menu by going to Tools > Language Templates. For memory, it is in Verilog > Synthesis Constructs > Coding Examples > RAM > BlockRAM. We recommend you explore these constructs.

Most FPGAs have what is called Distributed RAM. This essentially uses the logic gate and registers to implement the memory. Usually, Distributed RAM is faster than bRAMs, but it is smaller. 1

Note that there is also another method for instantiating the bRAMs. This is through the use of Block Design. We recommend you take a look at this document https://www.xilinx.com/support/documentation/university/Vivado-Teaching/Digital-Design/2014x/docs-pdf/Vivado_tutorial.pdf for more information about how to use IP Integrator and Block Design.

Exercises

1. Design and build a circuit to work as a Stack (LIFO). The user can use two push buttons in order to PUSH (BTNU) or POP (BTNC). Use 8 switches on the board as a value. When a user hits a PUSH button, it will store the value from the switches to the stack. When the user hits the POP switch, it will display the value from the top of stack in the two hex displays on the left. The other hex displays are used to display the number of elements currently in the stack. The stack can keep up to 256 elements. If the stack is full, hitting the PUSH button should not do anything.

   We recommend using a PUSH button as a reset (BTND).

2. Read an input from 5 binary switches. (You are welcome to use any switch. If you have no preference, use SW[4..0].) This should give you a number ranging from 0 to 31. Use distributed memory or bRAMs as the ROM for converting binary to 2 BCD for displaying on a seven-segment display. Use 5 bit binary as an address. The output can be either 2x4 BCD for applying to BCDtoSevenSegment (as used in the previous lab) or 2x8 for feeding directly to a seven-segment display.

   Note: You may want to initialize the memory from data. Please see the language template for more information

3. Use Block Design to create a simple calculator 4-bit calculator. You will assign 4 switches as the 4-bits input for A and another 4 switches for B. You will assign 4 push buttons to do 4 different operations.
   a. When BTNU is pushed, you will display the result of A+B in base 10 using the three 7-segment displays.
   b. When BTNL is pushed, you will display the result of A-B in base 10 using the three 7-segment displays.
   c. When BTND is pushed, you will display the result of A*B in base 10 using the three 7-segment displays.
   d. When BTNR is pushed, you will display the result of A/B in base 10 using the three 7-segment displays.

4. Please answer the following questions and submit (in PDF format) to CourseVille on Friday before 23:59 (midnight).
    a. Explain your ROM for mapping 5-bit binary to 2-digit BCDs (or 2x8 bits seven segment displays depending on your design in Exercise.2).

Laboratory 5: Simple CPU and Memory Mapped I/O

Objectives

1. Able to implement Simple CPU in VerilogHDL
2. Able to implement the memory mapped I/O

Background

Memory Mapped I/O and Port-Mapped I/O

Memory Mapped I/O and Port-Mapped I/O are two different methods for implementing input/output between CPU and peripheral devices in a computer.

Memory Mapped I/O uses the same address to address both memory and I/O devices. The memory and registers of devices are associated with physical addresses. Each I/O device monitors the CPU's address bus and responds to any CPU access of an address assigned to that device, connecting the data bus to the desired device's hardware register. This type of I/O allows software to interact with I/O directly using standard load and store instructions.

Unlike the Memory Mapped I/O, Port-mapped I/O uses special CPU instructions for performing I/O, such as the inp and outp. The benefit of Port-mapped I/O is the separation of address spaces between I/O and memory.

To connect a device to either memory mapped I/O or port-mapped I/O, a decoder must be added to the address. When an address is matched, accessing (reading and writing) to the data bus will be relayed to a device. In certain cases (eg. microcontroller), control registers (and buffers) will be added to related addresses in order to act as an interface between processor and devices.

(For more information, see the video clips.)

Exercises

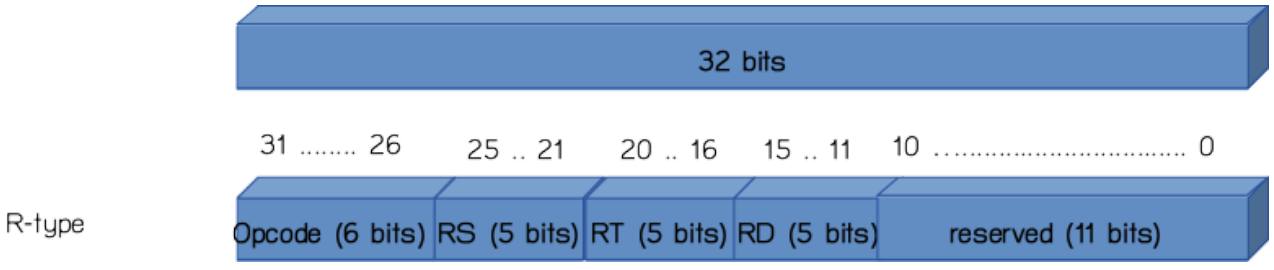
Please use nanoLADA CPU code[6] as a base.

1. Based on the given ALU, extend the ALU to support the following operations.

ALU OP	Operations
000	{Cout,S}=A+B+Cin;
001	{Cout,S}=A-B;
010	S=A | B; Cout=0; (or)
011	S=A & B; Cout=0; (and)
100	S=A ^ B; Cout=0; (xor)
101	S=-A; Cout=0; (2's complement)
110	S=~A; Cout=0; (not)
111	S=~B; Cout=0; (not)

Use the simulator to validate your ALU.

2. . Extend the R-type instruction format to support the following operations.



[6] Available at
https://www.cp.eng.chula.ac.th/~krerk/books/Computer%20Architecture/nanoLADA/

Original instruction

	Opcode (6 bits)	RS (5 bits)	RT (5 bits)	RD (5 bits)	Reserved (11bits)
ADD rd, rs, rt R[rd] ← R[rs] + R[rt];	000001				

New instructions.

	Opcode (6 bits)	RS (5 bits)	RT (5 bits)	RD (5 bits)	ALU Operation (11 bits)
ADD rd, rs, rt R[rd] ← R[rs] + R[rt];	000001				00000000_000
SUB rd, rs, rt R[rd] ← R[rs] - R[rt];	000001				00000000_001
OR rd, rs, rt R[rd] ← R[rs] | R[rt];	000001				00000000_010
AND rd, rs, rt R[rd] ← R[rs] & R[rt];	000001				00000000_011
XOR rd, rs, rt R[rd] ← R[rs] ^ R[rt];	000001				00000000_100
COM rd, rs,xx R[rd] ←	000001				00000000_101

-R[rt];					
NOT rd, rs, xx R[rd] ← ~R[rs]	000001				00000000_110

Use the simulator to validate your extended CPU.

3. Use the knowledge from memory-mapped I/O to map the following devices to associated addresses. (You have to add a few registers and buffers to the associated address.) Note that you have to take away memory from those addresses.

Address	Device	Note
0xFFF0	4-bit register for driving Seven-segment display digit 0.	Use last 4 bits
0xFFF4	4-bit register for driving Seven-segment display digit 1.	Use last 4 bits
0xFFF8	4-bit register for driving Seven-segment display digit 2.	Use last 4 bits
0xFFFC	4-bit register for driving Seven-segment display digit 3.	Use last 4 bits
0xFFE0	4-bit buffer for reading Switch [3..0] (A)	Use last 4 bits
0xFFE4	4-bit buffer for reading Switch [7..4] (B)	Use last 4 bits
0xFFE8	4-bit buffer for reading Switch [11..8] (Op)	Use last 4 bits

Write a simple software to your ROM to repeatedly read from switches. Display the value of switch [3..0] to seven-segment display digit 0. Display the value of switch [7..4] to seven-segment display digit 1. For seven segment display digits [3..2], show the result from the following operations.

Switches [10..8] (Op)	**Seven segment display [3..2]**
000	Show A+B
001	Show A-B
010	Show A | B
011	Show A&B
100	Show A^B
101	Show ~A (not A)
110	Show -A
111	Show -B

Demonstrate your design in our BASYS 3 FPGA board.

Laboratory 6: VGA and UART

(This lab is partly taken from Pitchaya Sitti-Amorn, Ph.D.)

Objectives

1. Understand the asynchronous serial communication
2. Understand how to use FPGA to interface with external devices

Background

One use of the FPGA is to interface with external devices that require precise timing. In this lab, we will be looking at two devices: VGA (Display), and a serial communication (UART).

VGA

VGA protocol is designed when computer monitors still used cathode ray tube (CRT) devices. In those times, most of the controls are actually analog. While FPGA cannot output the analog signal to control the VGA port directly, it can be used with a resistor ladder to create a simple digital to analog signal (DAC) (See VGA Port section of the BASYS3 reference[7]) .

And due to the analog design of the VGA signal, it will also require precise timing. In this lab, you will be interfacing with the VGA and make some adjustments from the given code. You can also get code and more information from embedded thoughts[8].

Note. Please make sure that you understand H-Sync, V-Sync, and related signals. You may find it in a quiz.

UART

Universal Asynchronous Receiver/transmitter or UART is a computer protocol that enables data transfer between two devices. UARTs are commonly used with the electrical

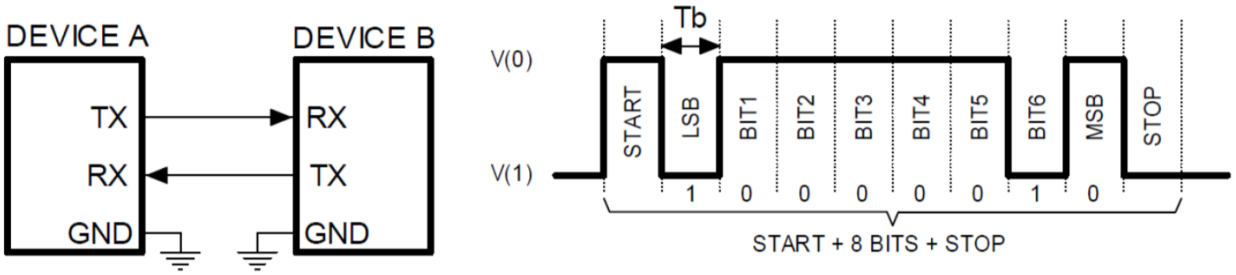
[7] https://reference.digilentinc.com/basys3/refmanual

[8] https://embeddedthoughts.com/2016/07/29/driving-a-vga-monitor-using-an-fpga/

layer standard such as TIA, RS-232, RS-422 or RS-485. The board you have in the lab has a UART port through the USB.

The minimal communication requirement by UART uses only two wires, TX and RX. Figure 1 shows a typical UART communication.

In order for a device to send the data out, both devices must use the same clock rate (baud), data bit size, parity and stop bits.
http://www.unm.edu/~zbaker/ece238/slides/UART.pdf provides good details on the protocol.



In order to communicate with the UART on the Basys 3 board, you will need to install some serial communication software such as Putty, Tera Term, etc. You can also find more information about the USB-UART Bridge on the Basys 3 in the reference manual.

Exercises

1. VGA: Either modify the example code or rewrite your own code so that the board will display gradients between two colors set by the switches. The gradient can be changed back and forth between horizontal to vertical via a push-button.

2. UART: You will implement a simple program that receives UART inputs, add one to each input character, then returns to the UART. If you test this with a console. For example, if you type "***abcde***" in the console, you should receive "***bcdef***".

> You may choose any baud rate you would like to use. I recommend testing with 9600bps or 115200bps. You may also want to test the loopback, i.e. wiring TX and RX together.

Note: You might need a terminal software to connect your computer to BASYS3. On Linux (and Mac OS X), try minicom or screen. On Windows, try RealTerm[9], TerraTerm, or PuTTY.

[9] https://sourceforge.net/projects/realterm/