Digital Design Using Verilog and FPGAs
An Experiment Manual

Chirag Sangani
Abhishek Kasina

Contents

I Combinatorial and Sequential Circuits **1**

1 Seven-Segment Decoder **3**
1.1 Concept . . . 3
1.2 Implementation . . . 4
1.3 Decoder Design . . . 4
1.3.1 Logic Design . . . 4
1.3.2 Design using Verilog . . . 5

2 Digital Clock **7**
2.1 Concept . . . 7
2.2 Implementation . . . 7
2.3 Simple Digital Clock . . . 8
2.4 Digital Clock with LCD Display . . . 8

3 Memory and Loader **11**
3.1 Random Access Memory . . . 11
3.1.1 Introduction . . . 11
3.1.2 Implementation . . . 11
3.2 Loader . . . 12
3.2.1 Introduction . . . 12
3.2.2 Implementation . . . 12

II DLX Processor **13**

4 R-type Instructions **15**
4.1 Introduction . . . 15
4.2 Instruction Set Architecture . . . 15
4.2.1 Types of Instructions . . . 15
4.2.2 Instruction List . . . 16
4.3 Instruction Fetch . . . 16
4.4 Memory . . . 17
4.5 Register File . . . 17
4.5.1 Registers . . . 17
4.5.2 Register Busing . . . 18
4.5.3 Multiple Buses . . . 19
4.5.4 Register File Model . . . 19
4.6 Arithmetic Logic Unit . . . 20

4.7 Instruction Execution Datapath 21

5 I-type Instructions **23**
5.1 Introduction 23
5.2 Instructions 23
5.3 Immediate Value Datapath 24
5.4 Load-Store Datapath 24

6 J-type Instructions **27**
6.1 Introduction 27
6.2 Instructions 27
6.3 Instruction Execution Datapath 28

7 Peripheral Interface **29**
7.1 Introduction 29
7.2 Concept 29
7.3 Character LCD Module 30

Part I

Combinatorial and Sequential Circuits

Experiment 1

Seven-Segment Decoder

A seven-segment display is a form of an electronic display device for displaying decimal or hexadecimal numerals. Seven-segment displays are widely used in digital clocks, electronic meters and other electronic devices for displaying numerical information.

1.1 Concept

As indicated by the name, a seven-segment display is composed of seven display elements. Controllable individually, they combine visually to produce simplified representations of the numeric alphabet.

The seven segments are arranged as a rectangle, with two segments on each side, one on the top and bottom each, and the last one bisecting the rectangle horizontally at the middle.

The segments are referred to by the letters A to G as shown in the figure, where an optional DP decimal point (also sometimes referred to as the eight segment) is used for the display of non-integer numbers.

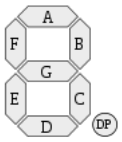

Figure 1.1: The individual segments of a seven-segment display.

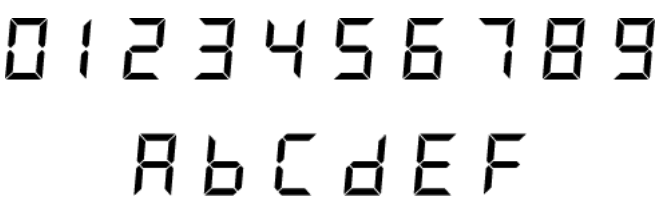

Figure 1.2: 7-segment display showing the 16 hexadecimal digits.

1.2 Implementation

Seven-segment displays may use a liquid crystal display (LCD) or arrays of light-emitting diodes (LEDs). In a simple LED package, which is the type of display that we will be working with, typically all of the cathodes (negative terminals) or all of the anodes (positive terminals) of the segments are connected together and brought out to a common pin; these devices are referred to as 'common cathode' or 'common anode' devices respectively. The other terminals are used to control the segments via a seven-segment decoder.

A seven-segment decoder receives a 4-bit number and displays the alphanumeric representation of that number (0, 1, 2, A, etc.) on a seven-segment display. The outputs of the decoder are labeled from A to G, which connect directly to the corresponding pins of the display. The decoder design depends on whether the display is a 'common anode' device or a 'common cathode' device.

1.3 Decoder Design

In this section, we will design a seven-segment decoder, assuming a 'common anode' device, and implement it on an FPGA using Verilog.

1.3.1 Logic Design

A segment on a 'common anode' device is visible when the input at its cathode is 0. Using this information, and knowing the inputs for which a segment must be visible, we can determine the truth table for a particular segment by assigning a value 0 to those inputs for which the segment must be visible. The final truth table is given below:

I_3	I_2	I_1	I_0	A	B	C	D	E	F	G
0	0	0	0	0	0	0	0	0	0	1
0	0	0	1	1	0	0	1	1	1	1
0	0	1	0	0	0	1	0	0	1	0
0	0	1	1	0	0	0	0	1	1	0
0	1	0	0	1	0	0	1	1	0	0
0	1	0	1	0	1	0	0	1	0	0
0	1	1	0	0	1	0	0	0	0	0
0	1	1	1	0	0	0	1	1	1	1
1	0	0	0	0	0	0	0	0	0	0
1	0	0	1	0	0	0	0	1	0	0
1	0	1	0	0	0	0	1	0	0	0
1	0	1	1	1	1	0	0	0	0	0
1	1	0	0	0	1	1	0	0	0	1
1	1	0	1	1	0	0	0	0	1	0
1	1	1	0	0	1	1	0	0	0	0
1	1	1	1	0	1	1	1	0	0	0

Table 1.1: Truth table of a seven segment decoder for a 'common anode' device

I_3, I_2, I_1 and I_0 denote the 4-bit binary input, with I_3 being the most significant bit.

From the above table, one can write expressions for A to G in terms of the inputs I_3, I_2, I_1 and I_0. The details of this are left to the reader as an exercise, and the final expressions are given below:

$A = \bar{I}_3.\bar{I}_2.\bar{I}_1.I_0 + \bar{I}_3.I_2.\bar{I}_1.\bar{I}_0 + I_3.\bar{I}_2.I_1.I_0 + I_3.I_2.\bar{I}_1.I_0$
$B = \bar{I}_3.I_2.\bar{I}_1.I_0 + \bar{I}_3.I_2.I_1.\bar{I}_0 + I_3.\bar{I}_2.I_1.I_0 + I_3.I_2.\bar{I}_1.\bar{I}_0 + I_3.I_2.I_1$
$C = \bar{I}_3.\bar{I}_2.I_1.\bar{I}_0 + I_3.I_2.\bar{I}_1.\bar{I}_0 + I_3.I_2.I_1$
$D = \bar{I}_3.\bar{I}_2.\bar{I}_1.I_0 + \bar{I}_3.I_2.\bar{I}_1.\bar{I}_0 + \bar{I}_3.I_2.I_1.I_0 + I_3.\bar{I}_2.I_1.\bar{I}_0 + I_3.I_2.I_1.I_0$
$E = \bar{I}_3.\bar{I}_2.\bar{I}_1.I_0 + \bar{I}_3.\bar{I}_2.I_1.I_0 + \bar{I}_3.I_2.\bar{I}_1 + \bar{I}_3.I_2.I_1.I_0 + I_3.\bar{I}_2.\bar{I}_1.I_0$
$F = \bar{I}_3.\bar{I}_2.\bar{I}_1.I_0 + \bar{I}_3.\bar{I}_2.I_1 + \bar{I}_3.I_2.I_1.I_0 + I_3.I_2.\bar{I}_1.I_0$
$G = \bar{I}_3.\bar{I}_2.\bar{I}_1 + \bar{I}_3.I_2.I_1.I_0 + I_3.I_2.\bar{I}_1.\bar{I}_0$

1.3.2 Design using Verilog

Writing Verilog code to implement the above design is trivial. An alternative and easier design can be achieved by the use of the `always` block. The `always` block can be used to model both sequential as well as combinatorial circuits. Details on how to use the `always` block have been provided in the next experiment.

The above implementation for a 'common anode' device can be easily modified for a common cathode device by inverting the outputs. The details are left to the reader as an exercise.

Experiment 2

Digital Clock

This chapter introduces the implementation of sequential logic on FPGAs. Sequential logic is a type of logic circuit whose output depends not only on the present input but also on the history of the input. This is in contrast to combinational logic, whose output is a function of, and only of, the present input. In other words, sequential logic has memory while combinational logic does not. We will design a simple clock module that gives the time as an output. We will also learn how to display the output on an LCD. The kit used for the latter purpose is a Xilinx Spartan-3E Starter Kit, and the development software used is the Xilinx ISE Design Suite 12.1.

2.1 Concept

We use counters to keep track of the time in a digital clock. A counter is a device which stores the number of times a particular event or process has occurred, often in relationship to a clock signal. On exceeding its maximum value, the counter reverts back to its original value.

Three counters will be used in the basic implementation - a `Seconds` counter, a `Minutes` counter and an `Hours` counter. The `Seconds` counter ideally receives a 1 Hz signal as input for its clock. When its value is 59, on receiving the next clock pulse, it resets itself to 0 and sends a clock signal to the `Minutes` counter. The functioning of the `Minutes` counter as well as the `Hours` counter is likewise similar, except the `Hours` counter resets on receiving a clock pulse when its value is 23 instead of 59.

2.2 Implementation

In this section, we will look at the implementation of a simple module that gives binary output. A more complicated design that utilizes an LCD to display the time is also given along with an explanation of the key differences in the next section.

Here, we introduce the `always` statement in Verilog. There are two separate ways of declaring a Verilog process. These are the `always` and the `initial` keywords. The `always` keyword indicates a free-running process. The `initial` keyword indicates a process executes exactly once. Both constructs begin execution at time 0, and both execute until the end of the block. Once an `always` block has reached its end, it is rescheduled (again). It is a common misconception to believe that an `initial` block will execute before an `always` block. In fact, it is better to think of the `initial` block as a special case of the `always` block, one which terminates after it completes for the first time.

`always` statements can be modified by a sensitivity list `@(...)`. Without this modification, the `always` block will run continuously. However, one can make its execution occur only at the change of certain or all variables, as well as dependent on an input clock signal by the use of the `posedge` command. A few examples are given below:

```
// Run continuously.
always

// Run when any variable changes its value.
always @(*)

// Run when the variables ‘a’ or ‘b’ change their value.
always @(a,b)

// Run when a positive edge is detected on CLK.
always @(posedge CLK)
```

2.3 Simple Digital Clock

In this example, we shall synthesize a `DigitalClock` module, which takes a 50 MHz clock input on the input wire `ClockCrystal` and gives the desired output on the output registers `Seconds[5:0]`, `Minutes[5:0]` and `Hours[4:0]`. Registers are used instead of wires as output, since wires are not capable of driving any load. This will become clearer as we model the clock.

Three concurrent `always` blocks can be used to update the output registers using non-blocking assignment operators (`<=`). The use of this operator ensures that the assignment operation executes only after the entire block has finished executing.

To supply a 1Hz signal to the concurrent `always` blocks, one can make use of a counter that receives a 50MHz clock from the onboard clock source. The `always` blocks should execute only when the counter reaches an appropriate value so that the total time taken to reach that value is one second, after which the counter should reset itself to 0.

2.4 Digital Clock with LCD Display

This module shows considerable differences from the previous one due to the requirement of displaying the output on an industry standard HD44780 compatible LCD display.

A major difference in this example, apart from the obvious inclusion of components required to drive the LCD, is the splitting of the `Seconds`, `Minutes` and `Hours` counters into two counters each. Each counter in the units place works as a decimal counter, counting up to 9, after which it resets on the next clock pulse. Counters in the tens place resent on achieving a value of 5, 5 and 2 (with the units counter on 3) for seconds, minutes and hours respectively. This is because the LCD accepts only one character at a time, and converting a binary number into a two digit decimal number is complicated.

The six counters are updated using the same method of concurrent `always` blocks and a counter generating a 1Hz signal as in the previous example.

To display data on the LCD, one needs to first initialize the LCD on power-on. The LCD connects to the FPGA through a 4-bit data bus and three control bits. Since the data bus is also shared with the onboard StrataFlash, it is necessary to write a logical high to the SF_CE0 bit at all times while using the LCD. A schematic diagram is given in the user manual, which has been reproduced here for convenience:

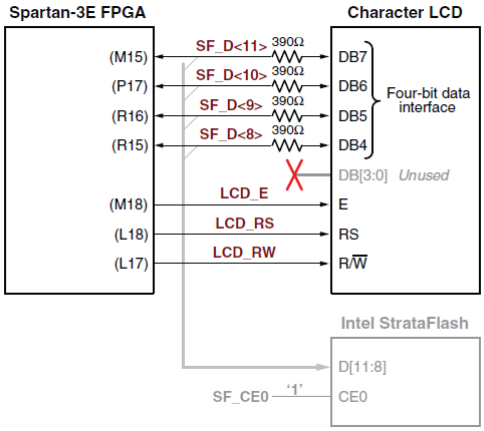

Figure 2.1: Character LCD Interface

A description of the various control and data bits used to control the LCD is given in the following table.

Signal Name	FPGA Pin	Function
SF_D<11>	M15	Data bit DB7
SF_D<10>	P17	Data bit DB6
SF_D<9>	R16	Data bit DB5
SF_D<8>	R15	Data bit DB4
LCD_E	M18	Read/Write Enable Pulse: 0:Disabled; 1:Read/Write operation enabled
LCD_RS	L18	Register Select: 0: Instruction register during write operations. Busy Flash during read operations; 1:Data for read or write operations
LCD_RW	L17	Read/Write Control: 0:WRITE, LCD accepts data; 1:READ, LCD presents data

Table 2.1: Character LCD Interface

The UCF constraints for the above pins are:

```
NET "LCD_E" LOC = "M18" | IOSTANDARD = LVCMOS33 | DRIVE = 4 | SLEW = SLOW ;
NET "LCD_RS" LOC = "L18" | IOSTANDARD = LVCMOS33 | DRIVE = 4 | SLEW = SLOW ;
NET "LCD_RW" LOC = "L17" | IOSTANDARD = LVCMOS33 | DRIVE = 4 | SLEW = SLOW ;
# The LCD four-bit data interface is shared with the StrataFlash.
NET "SF_D<8>" LOC = "R15" | IOSTANDARD = LVCMOS33 | DRIVE = 4 | SLEW = SLOW ;
NET "SF_D<9>" LOC = "R16" | IOSTANDARD = LVCMOS33 | DRIVE = 4 | SLEW = SLOW ;
NET "SF_D<10>" LOC = "P17" | IOSTANDARD = LVCMOS33 | DRIVE = 4 | SLEW = SLOW ;
NET "SF_D<11>" LOC = "M15" | IOSTANDARD = LVCMOS33 | DRIVE = 4 | SLEW = SLOW ;
```

At power-on, the LCD will first need to be initialized. The initialization procedure is given as follows:

- Wait 15 ms or longer, although the display is generally ready when the FPGA finishes configuration. The 15 ms interval is 750,000 clock cycles at 50 MHz.

- Write SF_D<11:8> = 0x3, pulse LCD_E High for 12 clock cycles.
- Wait 4.1 ms or longer, which is 205,000 clock cycles at 50 MHz.
- Write SF_D<11:8> = 0x3, pulse LCD_E High for 12 clock cycles.
- Wait 100 μs or longer, which is 5,000 clock cycles at 50 MHz.
- Write SF_D<11:8> = 0x3, pulse LCD_E High for 12 clock cycles.
- Wait 40 μs or longer, which is 2,000 clock cycles at 50 MHz.
- Write SF_D<11:8> = 0x2, pulse LCD_E High for 12 clock cycles.
- Wait 40 μs or longer, which is 2,000 clock cycles at 50 MHz.

Experiment 3

Memory and Loader

This experiment involves the modeling of a RAM (Random Access Memory) and a hardware module to read and load data into the RAM (hereby known as the loader). These two modules will be highly useful for the next part of this manual.

3.1 Random Access Memory

3.1.1 Introduction

A RAM is a large array of memory modules (generally registers) that can be addressed individually. It is a volatile form of storage. Although there are various models for different types of RAM, we shall be using a simplified model - that of an array of registers.

The RAM model we are interested in has 8-bit wide registers. Memory addressing is at the register-level. The memory address bus is 12-bit wide, so there are a total of $2^{12} - 1$ addressable registers in the module, taking the total capacity of the module to 4 KB.

3.1.2 Implementation

In Verilog, a register array can be instantiated as follows:

```
reg [7:0] RAM [0:4095];
```

We require that our RAM module gives us the value of any *one* register asynchronously. To do so, the RAM requires an input for selecting the memory address, as well as an output for returning the data. Also, to write data synchronously into the RAM, it requires an input bus and a clock signal. For writing, it uses the same memory address bus as the output. However, to control writing, a "Write Enable" Signal needs to be sent to the RAM via an input wire. The final module looks like this:

```
module Memory(
        input wire [7:0] IN,
        output wire [7:0] OUT,
        input wire [11:0] ADR,
        input wire WE,
        input wire CLK
    );
```

```
reg[7:0] RAM [0:4095];

always @(posedge CLK)
begin
        if(WE == 1)
                RAM[ADR] = IN;
end

assign OUT = RAM[ADR];

endmodule
```

3.2 Loader

3.2.1 Introduction

The Loader is a hardware module used to store data into the RAM manually. This module will be of use in the later part of this manual while designing processors.

The Loader is operated by the four slide-buttons found on the Xilinx Spartan-3E Starter Kit. These buttons are labeled as `SW[0]`, `SW[1]`, `SW[2]` and `SW[3]`. The entire Loader module should operate only when the `SW[0]` switch is high. The LCD is used to display the current address at which the Loader is pointing towards, the data stored in the current address, as well as the data stored in the last 3 addresses. `SW[1]` changes the address pointer - the direction (up/down) is determined by `SW[2]`. Finally, `SW[3]` increases the value stored at the current address.

3.2.2 Implementation

Like the LCD, the RAM is a synchronous module. It can be controlled in a fashion similar to the LCD - by the use of `case` statement.

The modeling of the other hardware is simple. Registers can be used as buffers that store the value of the address pointer and the values of the RAM memory cells. The state of the switches can be checked periodically every 1 second by means of an `always` block. After every such check, the state of the RAM as well as the buffers can be updated. The LCD can be updated periodically by means of another concurrent `always` block.

The UCF constraints for the four slide switches are given below:

```
NET "SW<0>" LOC = "L13" | IOSTANDARD = LVTTL | PULLUP ;
NET "SW<1>" LOC = "L14" | IOSTANDARD = LVTTL | PULLUP ;
NET "SW<2>" LOC = "H18" | IOSTANDARD = LVTTL | PULLUP ;
NET "SW<3>" LOC = "N17" | IOSTANDARD = LVTTL | PULLUP ;
```

Part II

DLX Processor

Experiment 4

R-type Instructions

4.1 Introduction

This part is dedicated to the design of a processor based on a simplified version of the DLX architecture. The DLX is a RISC processor architecture designed by John L. Hennessy and David A. Patterson, the principal designers of the MIPS and the Berkeley RISC designs (respectively), the two benchmark examples of RISC design. Intended primarily for teaching purposes, the DLX design is widely used in university-level computer architecture courses.

4.2 Instruction Set Architecture

Following the Princeton architecture (section 4.4), instructions are stored in their own separate memory. In the DLX architecture, they are fetched, stored and executed one at a time.

Simple instructions as in a RISC architecture provide the function code (also known as the opcode), addresses of the source registers, addresses of the destination registers as well as the "Write Enable" command. Instruction execution is sequential, and the clock for this process is obtained from the system clock.

4.2.1 Types of Instructions

DLX instructions can be broken down into three types, *R-type*, *I-type* and *J-type*. *R-type* instructions are pure register instructions, with three register references contained in the 32-bit word. *I-type* instructions specify two registers, and use 16 bits to hold an *immediate* value. Finally *J-type* instructions are jumps, containing a 26-bit address.

Opcodes are 6 bits long, for a total of 64 possible basic instructions. 5 bits are needed to select one of 32 registers. In the case of R-type instructions this means that only 21 bits of the 32-bit word are used, which allows the remaining bits to be used as "extended instructions". This allows the DLX to support more than 64 instructions, as long as those instructions work purely on registers. This is useful for things like FPU support. However, in this simplified version, we will be omitting FPU support.

For the purpose of this manual, the following table formally defines the structures of the *R-type*, *I-type* and *J-type* instructions.

Format	Bits					
	31:26	25:21	20:16	15:11	10:6	5:0
R-type	0x0	RS1	RS2	RD	*unused*	*opcode*
I-type	*opcode*	RS1	RD	*immediate*		
J-type	*opcode*	*value*				

Table 4.1: Types of Instructions

4.2.2 Instruction List

The following table lists the instructions that are to be implemented in this experiment. Note that the operations for each opcode are specified using C syntax and operators.

Instruction	Description	Format	Opcode	Operation (C-style coding)
ADD	add	R	0x20	RD=RS1+RS2
AND	and	R	0x24	RD=RS1&RS2
OR	or	R	0x25	RD=RS1|RS2
SEQ	set if equal	R	0x28	RD=(RS1==RS2?1:0)
SGE	set if greater than or equal	R	0x2D	RD=(RS1>=RS2?1:0)
SGT	set if greater than	R	0x2B	RD=(RS1>RS2?1:0)
SLE	set if less than or equal	R	0x2C	RD=(RS1<=RS2?1:0)
SLL	shift left logical	R	0x04	RD=RS1<<(RS2%8)
SLT	set if less than	R	0x2A	RD=(RS1<RS2?1:0)
SNE	set if not equal	R	0x29	RD=(RS1!=RS2?1:0)
SRA	shift right arithmetic	R	0x07	as SRL
SRL	shift right logical	R	0x06	RD=RS1>>(RS2%8)
SUB	subtract	R	0x22	RD=RS1-RS2
XOR	exclusive or	R	0x26	RD=RS1∧RS2

Table 4.2: R-type instructions

Note that SRA and SRAI are arithmetic right shifts. This means that, instead of shifting in zeros from the left, the sign bit of the operand is duplicated. SRL and SRA perform identically if RS1 is positive. If RS1 is negative (RS1[31] == 1), 1's are shifted in from the left for SRA and SRAI.

4.3 Instruction Fetch

On every positive edge on the clock input, which signifies the beginning of a new instruction cycle, the instruction is first read from memory. The address of the instruction is provided by the *Program Counter* (PC). After the fetch, the PC is incremented by the size of the instruction, and the instruction is stored in the *Instruction Register* (IR). Subsequently, the instruction in the IR is executed.

In the 32-bit DLX architecture, all instructions are 32-bit, or 4 bytes wide. Thus the PC needs to be incremented by four after each instruction fetch. The size of the PC is decided by the size of the address space of the program memory. In DLX, the address space is 32-bits wide. However, since each instruction is 4 bytes wide, the address of each instruction will necessarily have 0 in the last two bits. Thus, we need only a 30-bit wide register for the PC. Note that the 30-bit PC needs to be incremented by 1, and not 4, after each instruction fetch.

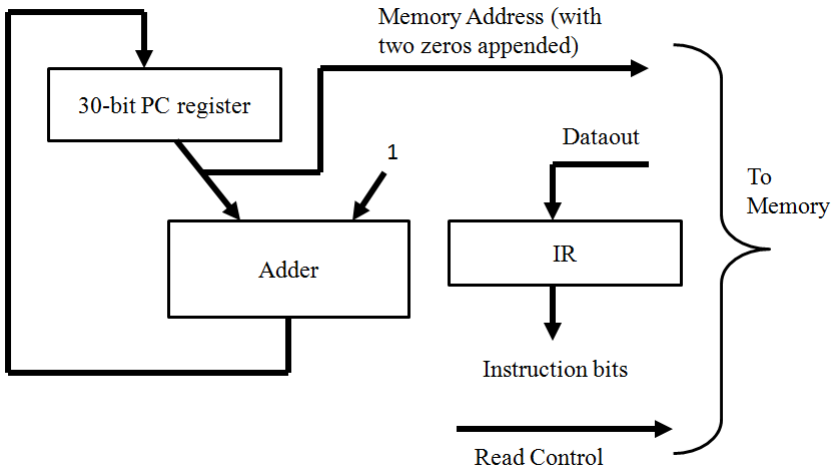

Figure 4.1: Instruction Fetch

4.4 Memory

Memories in a processor are of two types - program memory and data memory. Program memory is where the instructions are stored, and is read during Instruction Fetch. On the other hand, the data memory is used for storing data processed by the program. Data memory is accessed by the `LW` and `SW` commands, which are *I-type* instructions (see Experiment 5).

Depending on the architecture, the program memory and the data memory can be either separate (Princeton architecture) or they can be the same (Harvard architecture). In this manual, we shall be following the Princeton architecture of separate memories.

In this experiment, we shall only require a program memory since *R-type* instructions do not affect the data memory.

4.5 Register File

In this experiment, we shall implement the execution datapaths for the *R-type* instructions. To do so, we need to model the register file and the module that performs the calculations that are required by the instructions - the Arithmetic Logic Unit.

As mentioned before, the register file contains 32 registers labeled `R0`, `R1`,..., `R31`; each register being 32-bits wide. Out of these, `R0` and `R31` are special registers. `R0` always reads as 0; any data written to it is discarded. `R31` is a special register used for the `JAL` and `JALR` instructions. The remaining registers are general purpose, and are primarily used for executing instructions.

Before we model a register file, we shall first describe a register in greater detail.

4.5.1 Registers

A n-bit register is essentially a collection of n D-type flip flops as shown below:

The function of a register can be concisely described by its Verilog implementation:

```
module Register #(parameter n = 32) (
        input wire wr,
```

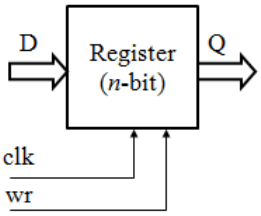

Figure 4.2: Register schematic

```
        input wire clk,
        input wire [n-1,0] D,
        output reg [n-1,0] Q
);

always @(posedge clk)
        if (wr == 1)
                Q = D;

endmodule
```

4.5.2 Register Busing

For our purpose, we require multiple registers to be connected to the same input and output buses (A bus is a collection of wires representing a multiple bit connection, also called as a vector).

Here is an example of two registers connected to the same bus:

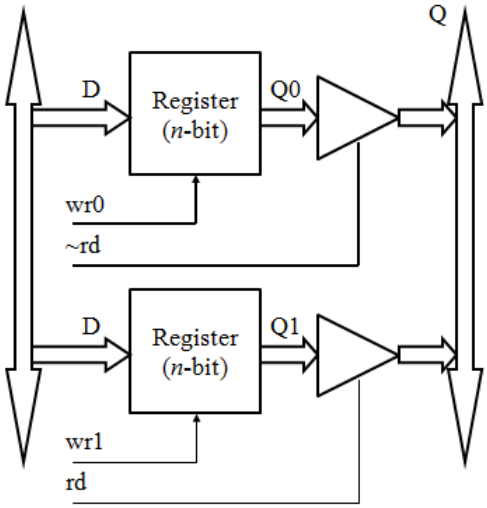

Figure 4.3: Register busing

These registers can be implemented in Verilog in the following manner:

```
always @(posedge clk)
begin
        if (wr0 == 1)
                Q0 = D;
        if (wr1 == 1)
                Q1 = D;
end

always @(rd, Q0, Q1)
begin
        if (rd == 0)
                Q = Q0;
        else
                Q = Q1;
end
```

4.5.3 Multiple Buses

Using the above method, we can add multiple buses to our Register File:

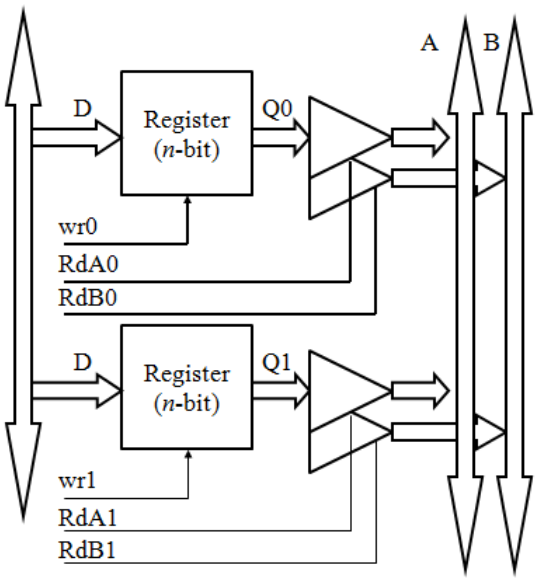

Figure 4.4: Multiple Buses

4.5.4 Register File Model

A schematic showing the various input and output ports of the required register file is give below:

`RD` is the selection vector for the destination register, and `D` is the data to be stored in the destination vector. `RdA` and `RdB` are the selection vectors for the registers that are to appear on buses `A` and `B` respectively. `WE` is the "Write Enable" bit - changes are allowed in the contents

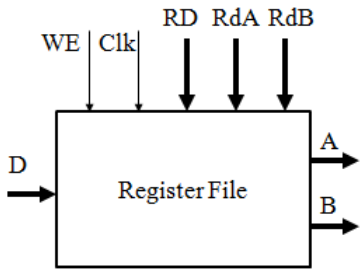

Figure 4.5: Register File schematic

of the file only when this bit is set high. `Clk` is the clock input for the synchronous operation of the register file.

The code for the register file is left to the reader as an exercise. Please note that the file that your model should follow the required specification for the DLX processor, that is, the `R0` register should always read as 0, irrespective of the data written into it.

Various alternative methods can be used to implement the above file. These methods may be easier to code as well as be more efficient. The reader is encouraged to experiment with such alternative methods.

4.6 Arithmetic Logic Unit

The Arithmetic Logic Unit (ALU) is a combinatorial module that performs the integer and logical calculations required by the instruction set. Its inputs are a set of operands and a function code (supplied by the control unit), and it gives the result as an output.

Generally, the design of an efficient ALU is very complicated. To keep the experiment simple, we shall forgo the requirement of performance and design a simple ALU that meets the minimum specifications of the instruction set.

A very simple ALU that can perform the `ADD` and `OR` instructions is given below:

```
module ALU(
        input wire [31:0] RS1,
        input wire [31:0] RS2,
        input wire [5:0] Opcode,
        output reg [31:0] RD
);

always @(*)
        case (Opcode)
                6'h20: RD = RS1 + RS2;
                6'h25: RD = RS1 | RS2;
                default:;
        endcase

endmodule
```

4.7 Instruction Execution Datapath

A simplified schematic of the execution datapath of a *R-type* instruction is given below.

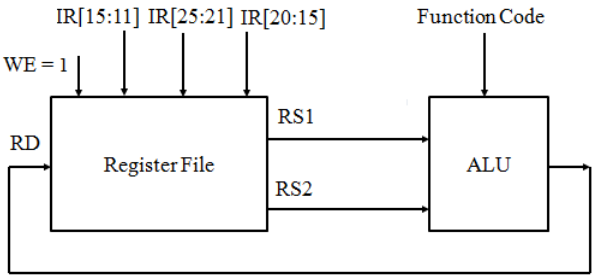

Figure 4.6: R-type execution datapath

The input to this module is the instruction register `IR` and a clock. Output ports consist of the outputs of the ALU and the Register File, that is, `RS1`, `RS2` and `RD`.

This datapath module, used in conjuction with the Intruction Fetch module, completes the *R-type* processor, which is the aim of this experiment. Instructions in the Program Memory can be loaded by the Loader module described in Experiment 3.

Experiment 5

I-type Instructions

5.1 Introduction

This experiment adds the execution datapaths for *I-type* instructions in addition to the *R-type* as described in the previous experiment.

5.2 Instructions

I-type instructions are those that involve registers, an immediate value supplied within the instruction, or the data memory. A list of *I-type* instructions is given below.

Instruction	Description	Format	Opcode	Operation (C-style coding)
ADDI	add immediate	I	0x08	RD=RS1+extend(immediate)
ANDI	and immediate	I	0x0C	RD=RS1&extend(immediate)
LHI	load high bits	I	0x0F	RD=immediate<<16
LW	load word	I	0x23	RD=MEM[RS1+extend(immediate)]
ORI	or immediate	I	0x0D	RD=RS1|extend(immediate)
SEQI	set if equal immediate	I	0x18	RD=(RS1==extend(immediate)?1:0)
SGEI	set if greater than or equal immediate	I	0x1D	RD=(RS1>=extend(immediate)?1:0)
SGTI	set if greater than immediate	I	0x1B	RD=(RS1>=extend(immediate)?1:0)
SLEI	set if less than or equal immediate	I	0x1C	RD=(RS1<=extend(immediate)?1:0)
SLLI	shift left logical immediate	I	0x14	RD=RS1<<(extend(immediate)%8)
SLTI	set if less than immediate	I	0x1A	RD=(RS1<extend(immediate)?1:0)
SNEI	set if not equal immediate	I	0x19	RD!=(extend(immediate)?1:0)
SRAI	shift right arithmetic immediate	I	0x17	as SRLI
SRLI	shift right logical immediate	I	0x16	RD=RS1>>(extend(immediate)%8)
SUBI	subtract immediate	I	0x0A	RD=RS1-extend(immediate)
SW	store word	I	0x2B	MEM[RS1+extend(immediate)]=RS2
XORI	exclusive or immediate	I	0x0E	RD=RS1^extend(immediate)

Table 5.1: I-type instructions

A few notes:

- The SW instruction actually uses RS2 for a source register (the value going to memory comes from RS2).

- `LHI` is used to load the upper bits of a 32-bit constant. To load the 32-bit constant 0x12345678 into `R1`, execute the following instructions:

```
LHI     R1,#0x1234
ORI     R1,R1,#0x5678
```

5.3 Immediate Value Datapath

For *I-type* instructions that are immediate value versions of their *R-type* counterparts, only a small change in the original execution datapath for *R-type* is necessary to accomodate I-type:

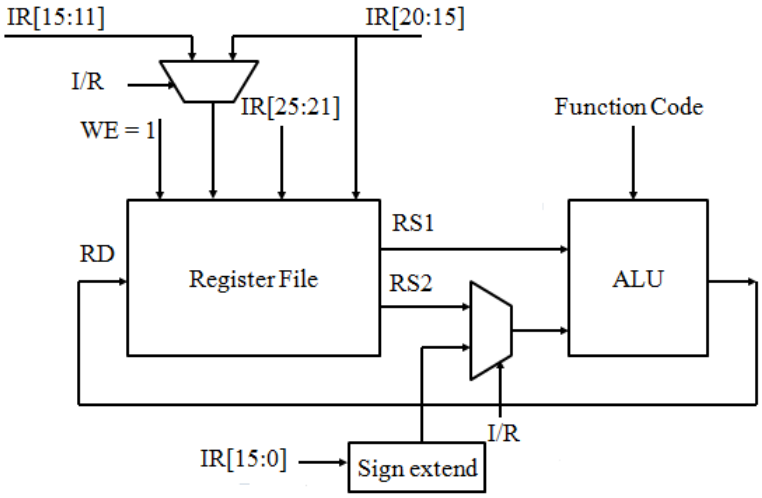

Figure 5.1: R-type and I-type execution datapath

The "Sign Extend" module converts a 16-bit number into a 32-bit number without changing its value or sign.

5.4 Load-Store Datapath

This datapath allows access to the data memory via the `LW` and the `SW` commands.

Note that the input for `RS2` selection and `RD` selection are the same in *I-type*, though the value of the `RS2` register is ignored. This fact can be exploited to use the `RS2` register as an input to the memory. Although we are supplying the address of the source register in the value of `RD`, it is truly `RS2` which supplies the data to the memory. And by ensuring that the output of the ALU goes into the address of the memory and the new memory output is fed back into `RD`, as obvious from the figure, we can avoid overwriting the value of the original source register.

This completes the implementation of the *I-type* instructions execution datapath. The working of the processor can be verified by adding another Loader module for the data memory (in addition to the Loader for the program memory).

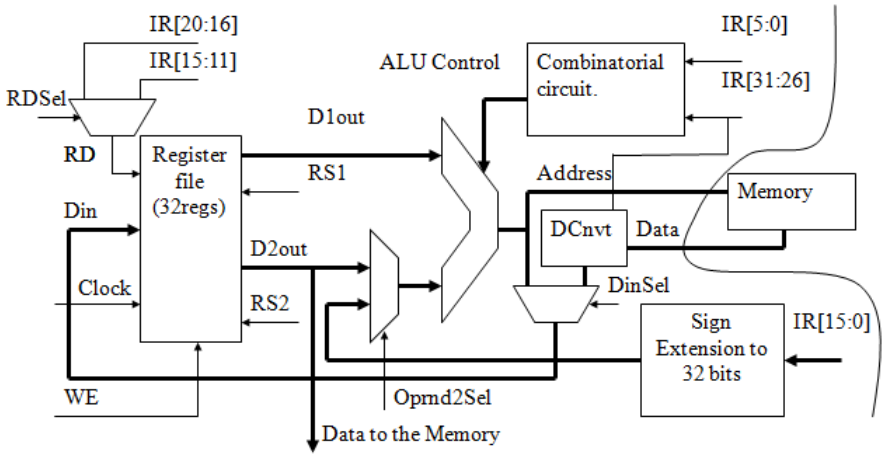

Figure 5.2: R-type and I-type execution datapath with Load-Store

Experiment 6

J-type Instructions

This experiment introduces the implementation of the last type of instructions in the DLX instruction set architecture - *J-type*, or *jump* instructions.

6.1 Introduction

J-type instructions are those which modify the `Program Counter` (`PC`). They may do so explicitly or only when a certain condition is satisfied.

From the point of view of computing, *J-type* instructions are very important. They allow for the execution of conditional statements as well as implicit repetition of intstructions. These are important qualities for the processor to be Turing-capable.

6.2 Instructions

The following table lists all the *J-type* instructions.

Instruction	Description	Format	Opcode	Operation (C-style coding)
`BEQZ`	branch if equal to zero	J	0x04	`PC+=(RS1==0?extend(immediate):0)`
`BNEZ`	branch if not equal to zero	J	0x05	`PC+=(RS1!=0?extend(immediate):0)`
`J`	jump	J	0x02	`PC+=extend(value)`
`JAL`	jump and link	J	0x03	`R31=PC+4;PC+=extend(value)`
`JALR`	jump and link register	J	0x13	`R31=PC+4;PC=RS1`
`JR`	jump register	J	0x12	`PC=RS1`

Table 6.1: J-type instructions

A few points to be noted:

- The `BEQZ` and `BNEZ` instructions require three inputs instead of the maximum two supported by our current implementation of the ALU - `PC`, `RS1` as well as the immediate value.
- The `JAL` and the `JALR` instructions store the value of the next immediate instructions in case of no jump before jumping to another instruction.

6.3 Instruction Execution Datapath

The following figure gives a brief idea about the implementation of *J-type* instructions in harmony with the previous implementations.

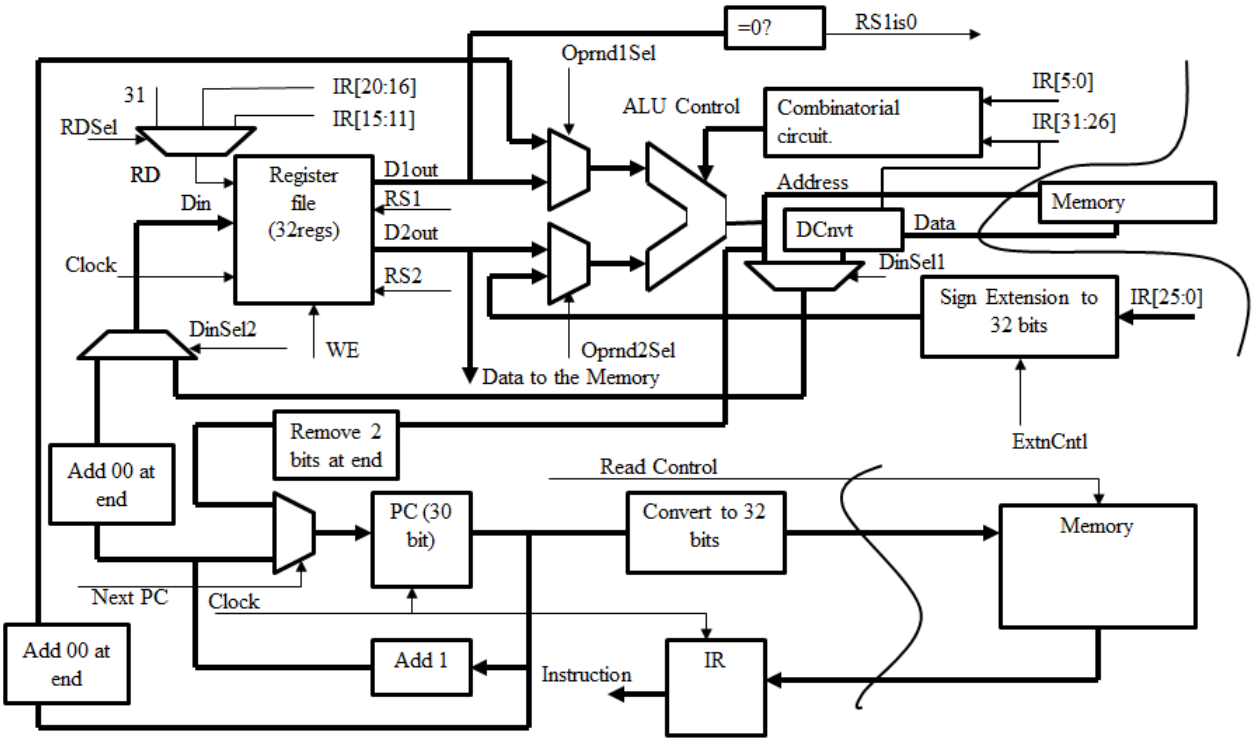

Figure 6.1: J-type instructions

The figure is quite self-explanatory. Note that the `RS1 is 0` pin is actually an input for the ALU, although not shown.

The successful implementation of this module can be tested by using a Loader module for the data memory and using jumps to skip a `SW` instruction.

Experiment 7

Peripheral Interface

7.1 Introduction

Peripheral interfacing involves the communication between the processor and other devices, such as the LCD module, PS/2 port, VGA port, etc. There are various methods to achieve this - we shall be using the method of mapping the peripherals to the data memory.

7.2 Concept

Peripherals can be controlled and communicated with by a collection of control, data and address pins. By permanently mapping these pins to certain addresses in the data memory, the processor can communicate with the peripheral by means of simple `LW` and `SW` commands.

An example of such a mapping is given below for a row of 8 LEDs.

```
module DATAMEM(
        input wire [3:0] ADR,
        input wire [7:0] IN,
        output wire [7:0] OUT,
        input wire WE,
        input wire CLK,
        output wire [7:0] LED_OUT
);

reg [7:0] RAM [0:15];

always @(posedge CLK)
        if (WE == 1)
                RAM[ADR] <= IN;

assign LED_OUT = RAM[1];

endmodule
```

Any data written into the address 1 will be immediately reflected onto the row of LEDs. Using this method, one can also read data from peripherals by reading from wires posing as registers in the address spectrum.

7.3 Character LCD Module

As an example of peripheral interfacing, we shall communicate with the character LCD available on the Xilinx Spartan-3E Starter kit.

Writing data onto the LCD interface as per the above method is trivial, and will not be explained here. You may manually strobe the *LCD_E* signal. To generate the required delays, you can use the following commands (assuming a 50MHz clock):

```
ADDI, R2, R2, #0x8000
BNEZ, R2, #0xFFF8
```

The first command increments the value of `R2` register by 0x8000 every time. The second command ensures that the first command is run until the `R2` register overflows and returns to a value of 0. The delay generated by these commands is approximately 5.25 ms, which is sufficient for most purposes while driving the LCD.

Using the above methods, you are to display on the LCD:

1. The string "Hello World!"
2. The sequence of Fibonacci numbers displaying one number at a time for one second.