Lab Manual-2 for CE/CZ3001 Advanced Computer Architecture

Register file and Single cycle datapath for Register type instruction

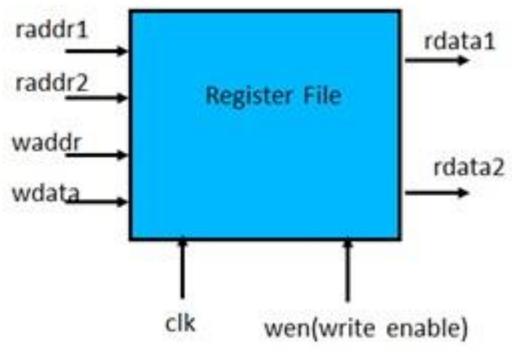
SCHOOL OF COMPUTER SCIENCE AND ENGINEERING
NANYANG TECHNOLOGICAL UNIVERSITY

LAB: 2

The register file (RF) in the CPU is used to hold operands and addresses for one or more clock cycles. In this lab you will analyse the implementation of the RF. Besides, you also need to find the area and access time of RF of different sizes (number of registers) and for various bit-widths of registers. You are provided with parametrizble Verilog code where the bit-width and the number of registers can be changed. You can specify the bit-width and the number of registers in the Verilog code of the RF and find the change in LUT consumption and access speed from the synthesis results. In the second part of lab 2, you need to build a single cycle datapath for Register-type instruction. You are also given with the Verilog code for the control logic. You need to notice how the code of register file with the code for control logic and the ALU (of lab1) are integrated by module instantiation to form the datapath. You are required to understand the given Verilog code completely.

I. PARAMETRIZABLE IMPLEMENTATION OF RF

You are provided with the Verilog file written in behavioural approach for implementing the parametrizable RF where the bit-widths and number of registers can be changed to find the hardware and access time corresponding to different bit-widths and address-widths. The block diagram of the register file are given in Fig.1



Note that address width depends on the number of registers in the RF. If the address width is "n' then number of registers in the register file is 2^n. The following are the characteristics of the behavioural RF. The RF is positive edge triggered; which means that the data available at the write port (wdata) is written into the selected register in the register file on the positive edge of clock if the write enable (wen) signal is high. The register in which the wdata could be written is selected by a decoder according to the write address (waddr).The read ports are combinational circuits. Data stored the registers specified by raddr1 and raddr2 are obtained as radata1 and rdata2, respectively. All registers are reset to 'zero' when the reset signal is **high**.

1. The Verilog file 'regfile.v' (provided) is a parametrizable behavioural code.
2. The 'define.v' file is used to define DSIZE (bit-width of the register) = 4, NREG (number of registers) =4, ASIZE (address size)=2. The testbench 'regfiletest.v' (provided) helps to test the

behavioural implementation of RF. The test-bench uses file operation (an input file is used to read data therefrom and an output file to write data).

3. Verify the operation by simulation.
4. Synthesize the given behavioural code and plot the ‘number of slices’ used vs ‘No. of registers’ and ‘the minimum clock cycle period’ in ns vs ‘No. of registers’ (change the parameter for No. of registers {NREG=4, 8, 16, 32, 64}). Here we can keep the bit-width of each register (DSIZE parameter) to be constant (for example DSIZE=64). Note that when we change number of registers (NREG) the address parameter ASIZE also needs to be changed since NREG = 2^(ASIZE).
5. Please note that we are taking the minimum clock period for the delay and not the maximum combinational delay.

Table 1: LUT consumption and delay for behavioural register file vs bit-width

No. of Registers(NREG)	Bit-width of the register (DSIZE)	No of register slices	No of LUT slices	Minimum clock Period
4	64			
8	64			
16	64			
32	64			
64	64			

6. Synthesize the given behavioural code and plot the number of slices used vs bit-width of the register (DSIZE) and delay in ns vs bit-width of the register (change the parameter for bit-width of the registers {DSIZE=4, 8, 16, 32, 64}). Here we can keep the number of registers (NREG parameter) to be constant (for example NREG=32) and thus ASIZE will be a constant =5.

Table 2: LUT consumption and delay for behavioural register file Vs Number of registers

Bit-width of the register (DSIZE)	No. of registers (NREG)	No of register slices	No of LUT slices	minimum clock period in ns
4	32			
8	32			
16	32			
32	32			
64	32			

EVALUATION -I

1) Plot the graph, area in LUT slices (vs) No. of registers (NREG) as well as delay (vs) No. of registers (NREG) for the register file module for NREG =4, 8, 16, 32, 64. Set DSIZE (bit-width of each register) =64.
2) Plot the graph, area in LUT slices (vs) bit-width of register (DSIZE) as well as delay (vs) bit-width of register (DSIZE) for the register file module for DSIZE =4, 8, 16, 32, 64. Set NREG (number of registers) =32.

II. Simple Datapath and Control

In this section you will use the ALU (64 bits) from the first lab and the register file (parameter NREG (number of registers) =32, DSIZE (bit-width of each register) =64, and ASIZE(address size to access 32 registers) =5) which you have analysed earlier in this lab to make a simple Datapath and control unit of a processor. The Verilog code for control unit which generates the control signals for the operation of RF and ALU is also provided (‘control.v’). Instead of directly feeding data inputs to the ALU or RF, you need to do it by an instruction and according to the signal generated by the control unit to perform a sequence of operations. Please note that you have 64-bit data, and a 32-bit instruction words.

The ADD, SUB, AND, XOR, and ORR, instructions are of R-format instructions with addressing mode register addressing (R- format)

R format ALU instructions:
Meaning: [Rd] 🡐 [Rn] (OP) [Rm]: the registers Rm and Rn are the source registers, and the result is stored in the destination register Rd. Instruction structure: <operation> <Rd>, <Rn>, <Rm>
The bit assignments to different fields of R-format are shown below.

opcode	Rm	shamt	Rn	Rd
31 21	20 16	15 10	9 5	4

Examples:

1. ADD X5, X4, X3 (meaning: X5 🡐 (X4 + X3). The machine format is given below.

00000000000	00011	000000	00100	00101
31 21	20 16	15 10	9 5	4

2. XOR X8, X9, X10 (meaning: X8🡐 X9 (XOR) X10):

00000000011	01010	000000	01001	01000
31 21	20 16	15 10	9 5	4

Please note that as per define.v, the opcodes for ADD, SUB, AND, XOR and ORR are ‘00000000000’, ‘000000000001’, ‘000000000010’, ‘00000000011’and ‘000000000100’ respectively. A block diagram of the expected design is shown in Fig. 1.

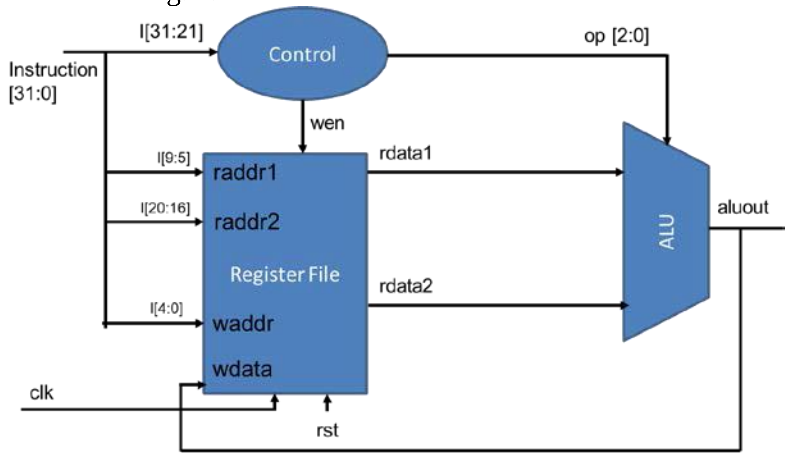


Figure 1: Datapath and Control

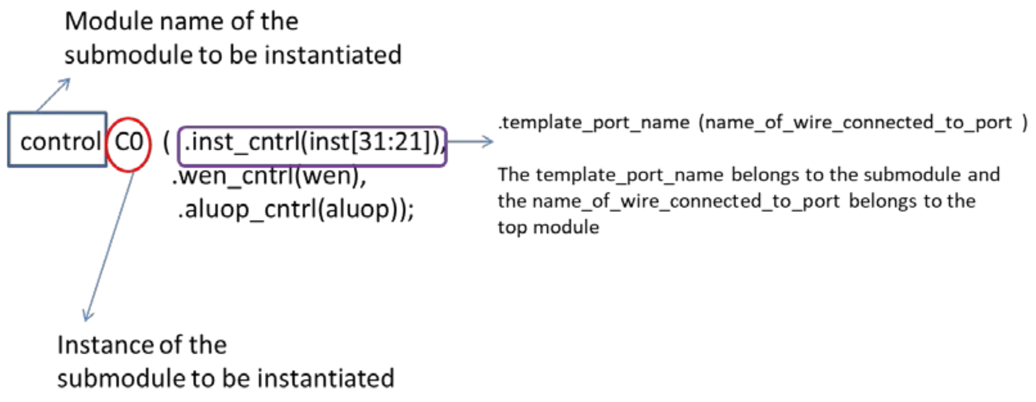
The control unit provides the control words for RF (write enable ‘wen’ signal), and ALU (operation ‘op[2:0]’ signal, for selecting the operation in ALU). RFs are initialized to zero. We need to load data into the registers of the RF before we do any operations. Since you have not yet implemented memory operations, we need to load few values into the registerfile by hardcoding as done in regfile.v shown below.

> *“regdata[1] <=2;//hardcoding few values into register file for initialization*
> *regdata[2] <=3;”*

In order to put together the datapath and control we need to connect the RF, ALU and the control unit. The main “datapath.v” is provided to you. In the given Verilog file you need to instantiate “regfile.v”, “control.v”, are “alu.v” and connect together. Modules are instantiated inside other modules, and each instantiation creates a unique object from the template. The instantiated module’s ports must be matched to those defined in the template. This is specified: (i) by name, using a dot(.) “ .template_port_name (name_of_wire_connected_to_port)” or

(ii) by position, placing the ports in exactly the same positions in the port lists of both the template and the instance.

In Fig. 2, an example of module instantiation is given by connecting the submodule “control.v” to the main module/top module “datapath.v”


Module name of the submodule to be instantiated
.template_port_name (name_of_wire_connected_to_port)
The template_port_name belongs to the submodule and the name_of_wire_connected_to_port belongs to the top module
Instance of the submodule to be instantiated

```
control C0 ( .inst_cntrl(inst[31:21]),
             .wen_cntrl(wen),
             .aluop_cntrl(aluop));
```

Figure 2: Module instantiation

The rest of the module instantiations are done in a similar way as shown in Fig. 3.

```
control C0
(.inst_cntrl(inst[31:21]),.wen_cntrl(wen),.aluop_cntrl(aluop));


regfile  RF0 (.clk(clk), .rst(rst), .wen(wen), .raddr1(inst[9:5]),
.raddr2(inst[20:16]), .waddr(inst[4:0]), .wdata(aluout),
.rdata1(rdata1), .rdata2(rdata2));


alu  ALU0 ( .a(rdata1), .b(rdata2), .op(aluop), .out(aluout));
```

Figure 3: Module instantiation of all the four submodules

Note that the opcode encoding is given in the define.v file, and should not be modified. NREG (number of registers) =32, DSIZE (bit-width of each register) =64 and ASIZE (address size) =5.

1. Test the datapath implementation by generating a test bench. Test bench can be generated in the ‘simulation mode’ by right clicking the top module and adding ‘New source’ to be ‘Verilog test fixture’. Name the Verilog test bench and choose the corresponding top module to generate the test bench. Once the test bench is generated, you need to check the bit-width of the inputs and outputs, add the clock and the test inputs as shown below.

   a. Check the bit-width of input ‘inst’, as well as the bit-width of output ‘aluout’. Make their bit-width in accordance with ‘define.v’ file (ISIZE=32, DSIZE=64 and ASIZE=5).
   b. Inserting CLK signal: Insert ‘**always #15 clk = ~clk;**’ before the ‘initial’ statement.
   c. Add the following test vectors on the portion “//Add the stimulus”
      #10 rst =1;
      #50 rst=0;
      #50;// Wait 50 ns for global reset to finish

      inst=32'b00000000000001000000000000100011;// ADD X3, X1, X4
      // ISIZE =32, DSIZE=64 bit and Write and read ADDR= 5 bit.
      in this example opcode =0, Rn=1,Rm=4 and Rd=3. The ‘shamt’ filed is made zero.

      #100; inst=32'b00000000001000100000000000100100;//SUB X4,X1,X2

   d. More 32 bit instructions can be added to fully test the datapath below the earlier instructions.
      #100;inst= 32'h00040023; // ADD X3, X1, X4 (already written above-shown in hexformat)
      #100;inst= 32'h00220024;// SUB X4, X1, X2 (already written above-shown in hexformat)
      #100 inst=32'h00440027; //AND X7, X1, X4
      #100 inst=32'h006200E8; //XOR X8, X7, X2

You can use https://personal.ntu.edu.sg/smitha/OPCoder/OPCoder/converter.html
to convert the ARM instructions to machine code in binary or hexadecimal number system.

EVALUATION II

1) Complete the Verilog code for the full datapath which includes (RF, control and ALU) and verify its operation.

ADDITIONAL INFORMATION

Note that as we keep on increasing the number of registers and the bit width of the operands, the LUTs consumed by the architecture keeps on growing. This creates a problem for hardware engineer to incorporate higher bit width as well as complex architectures on to FPGA. The designers of FPGA’s used dedicated circuits called as “hard macros”, which can be used for the implementation of arithmetic circuits in less area. One such is “DSP 48” that we saw in lab 1.
Consider the case of Register file. If we keep on increasing the DSIZE and NREG the LUT size becomes large. This can be handled by using a hard macro called a “BRAM”. Note that hard-macros cannot be optimized further by the synthesis tools.