**CS211 Lab-8**

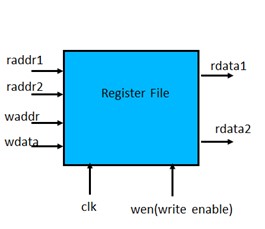
Faculty: Dr. Sharad Sinha

TA: Pavitra Bhade & Prachi Kashikar

In this lab you will analyse the implementation of the Register File (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 parametrizable 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. 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 lab7) are integrated by module instantiation. You are required to understand the given Verilog code completely.

# PARAMETRIZABLE IMPLEMENTATION OF RF

You are provided with the behavioral design in Verilog 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 is given in Fig.1



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. Data stored in the registers specified by raddr1 and raddr2 are obtained as radata1 and rdata2, respectively. All registers are reset to 0x0000 when the reset signal is **high**.

1. The Verilog file ‘regfile.v’ (provided) is a parametrizable behavioral 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=32).

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

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

1. Synthesize the given behavioral 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 used | No of LUT slices used | 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) =32.

1. 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.

# SIMPLE 32 BIT DATAPATH AND CONTROL

In this section you will use the ALU (32 bits) from the lab-7 and the register file (parameter NREG (number of registers) =32, DSIZE (bit-width of each register) =32, and ASIZE(address size) =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. As in the first two assignments, here also you will have 32-bit data, and a 32-bit instruction words.

The ADD, SUB, AND, XOR, COM, MULT and ADDI instructions have two different addressing modes called as register addressing (R- format) and immediate addressing (I-format).

R format ALU instructions:

Meaning: $rd  $rs (OP) $rt : the registers $rs and $rt are the source registers, and the result is stored in the destination register $rd. Instruction structure: <operation> <rd>, <rs>, <rt>

In case of MULT instruction only the lower 32-bits are stored at the destination. The bit assignments to different fields of R-format are shown below.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| opcode | Rs | Rt | Rd | shamt | fn | |
| 31 26 25 21 20 16 15 11 10 6 5    Examples:  1. ADD $5, $4, $3 (meaning: $5  ($4 + $3). The machine format is given below. | | | | | | 0 |
| 000000 | 00100 | 00011 | 00101 | 00000 | 000000 |  |
| 31 26 25 21 20 16 15 11 10 6 5    2. XOR $8, $9, $10 (meaning: $8 $9 (XOR) $10): | | | | | | 0 |
| 000011 | 01001 | 01010 | 01000 | 00000 | 000000 |  |

31 26 25 21 20 16 15 11 10 6 5 0

I format ALU instuction:

Meaning: meaning: $rt $rs + sign-extended(immediate): add the content of register $rs with the signextended form of the content of “immediate” field, and store the result in the destination register $rt. The ADDI instruction has a two address (one source and one destination) and one immediate format. The bit assignments to different fields of I-format are shown below.

|  |  |  |  |
| --- | --- | --- | --- |
| opcode | Rs | Rt | imm |

31 26 25 21 20 16 15 0

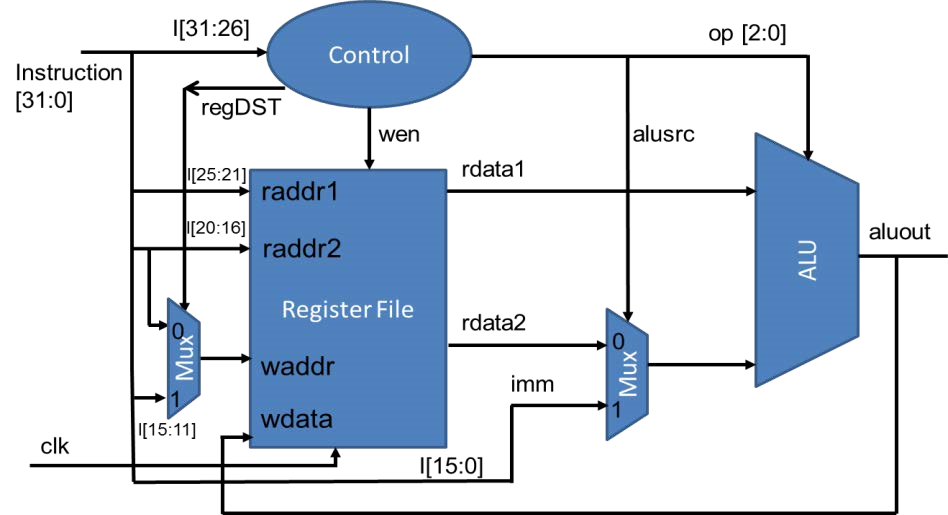
Example:

1. ADDI $5, $4, 3 (meaning: $5  ($4 + 3). Here ‘3’ is an immediate value. The machine format is given below.

|  |  |  |  |
| --- | --- | --- | --- |
| 000110 | 00100 | 00101 | 0000 0000 0000 0011 |

31 26 25 21 20 16 15 0 A block diagram of the

expected design is shown in Fig. 1. The control unit provides the control words for RF (write enable ‘wen’ signal), select signal for the waddr multiplexer( ‘regDST’), ALU (operation ‘op[2:0]’ signal) and select signal for the multiplexer (‘alusrc’). In lecture we have seen two level of control used in MIPS processor. But for simplicity, in the lab we will be using only one control unit. Hence, we will not be using the function field.



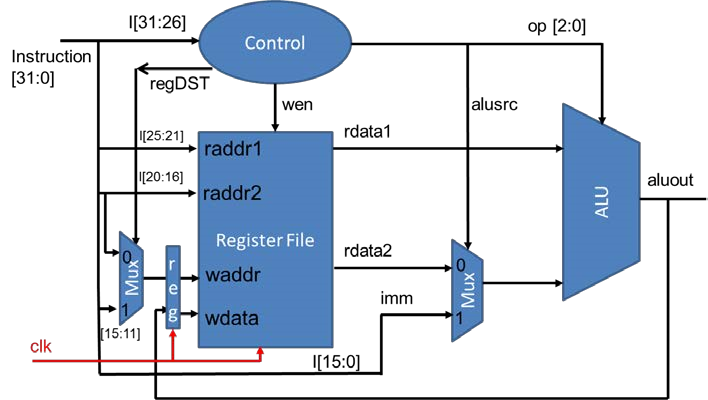
**Figure 1: 32-bit Datapath and Control**

RF is 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] <=5;//hardcoding few values into register file for initialization*

*regdata[2] <=2;*”

Two multiplexers (Mux) are required, one for selecting the write address for RF and another for selecting the second input to ALU. Control signal named ‘reg DST’ and ‘alusrc’ are used respectively by the Mux for selecting the operand. The detailed diagram is shown in Fig. 2. It can be noticed that the output ‘aluout’ is fed as wdata input to the RF input. This can create combinational loops. To eliminate combinational loops, a delay register ‘reg’ (given in ‘delay\_reg.v’) is inserted before wdata and waddr as shown in Fig. 2. This will enable to do the writeback at the beginning of the next clock cycle.

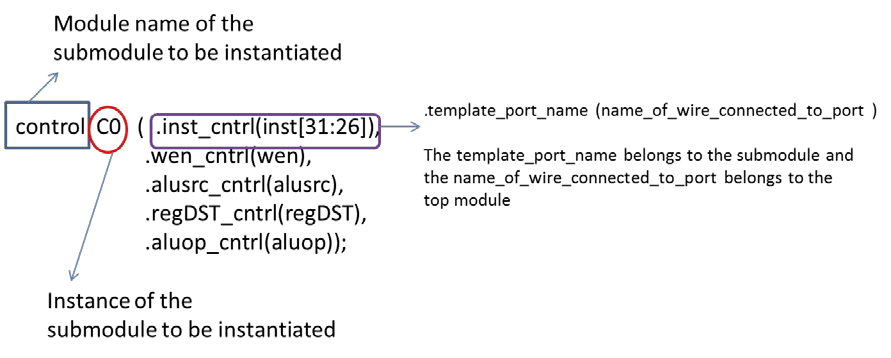


**Figure 2: Final 32-bit Datapath and Control**

In order to put together the 32 bit datapath and control we need to connect the 32 bit register file, 32-bit ALU and the control unit. As mentioned earlier a delay register is also needed to eliminate combinational loops. The main “datapath.v” is provided to you. The Verilog file also has the multiplexer defined. In the given Verilog file you need to see how “regfile.v”, “control.v”, “alu.v” and “delay\_reg.v” are instantiated and connected together. Modules are instantiated inside other modules, and each instantiation creates a unique object of the module. The instantiated module’s ports must be matched to those defined in the template. This is specified:

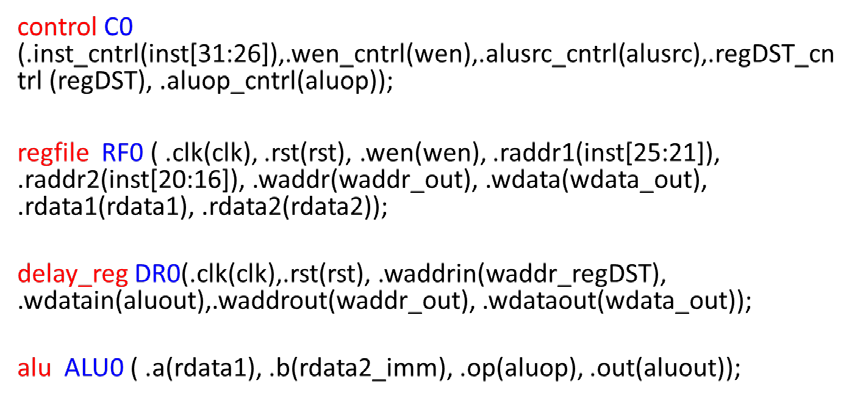
1. by name, using a dot(.) “ .template\_port\_name (name\_of\_wire\_connected\_to\_port )” or
2. by position, placing the ports in exactly the same positions in the port lists of both the template and the instance.

In Fig. 3, an example of module instantiation is given by connecting the submodule “control.v” to the main module/top module “datapath.v”



**Figure 3: Module instantiation**

The rest of the module instantiations are done in a similar way as shown in Fig. 4.



**Figure 4: 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) =32 and ASIZE (address size) =5.

1. Test the 32 bit 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.

* 1. 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=32 and ASIZE=5).
  2. Inserting CLK signal: Insert ‘**always #15 clk = ~clk;**’ before the ‘initial’ statement.
  3. 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'b00000000010000010011100000000000;// ADD$7,$2,$1 // ISIZE =32, DSIZE=32 bit and Write and read ADDR= 5 bit. in this example opcode =0, $rs= $2, $rt=$1 and $rd=$7. The ‘shamt’ and

‘fn’ filed are made zero.

* 1. More 32 bit instructions can be added to fully test the datapath below the earlier instructions.

#100;inst=32'h00413800; // ADD $7,$2,$1(already written above-shown in hexformat)

#100;inst=32'h04411800; // SUB $3,$2,$1

#100;inst=32'h08E13000; // AND $6,$7,$1 #100;inst=32'h18E40001; // ADDI $4,$7,1

#100;inst=32'h0CE12800; // XOR $5,$7,$1;

#100;inst=32'h10414800; // COM $9,$2,$1

#100;inst=32'h14415000; // MUL $10,$2,$1

You can use <http://www.ntu.edu.sg/home/smitha/OPCoder/OPCoder/converter.html>to convert the MIPS instructions to 32 bit-machine code in hexadecimal number system. Please do note to make corresponding changes to your opcode part of webpage according to the opcode that you have used in ‘define.v’ file.

# EVALUATION II

1) Complete the Verilog design for the full datapath which includes (RF (32 bit), control and ALU (32 bit) 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 7.

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.