Register File (of eight 16-bit registers)

DDCO Assignment 2a

October 3, 2017

In a modern computer, it typically takes hundreds of clock cycles to fetch data from main memory to the microprocessor (even cache memory reads take a few clock cycles). So if the ALU in the microprocessor operated directly on data from memory, it would spend most of its time waiting. To achieve high performance, microprocessors incorporate a *register file*. The register file, a key component of microprocessors, is a bank of registers used as a temporary high-speed storage into which data is fetched from memory. The register file then provides inputs to the ALU. As you may recall, in assignment 1 you constructed an ALU that received two 16-bit inputs and produced one 16-bit output.

1 Register File Implementation

So in this assignment, your task is to construct a register file, from which two 16-bit values can be read, and to which one 16-bit value written, every clock cycle. Therefore, the register file requires two read ports and one write port. The register file should contain eight (16-bit) registers. Each of the read and write ports need to be able address any of the eight registers. Register address varies from 0 to 7, hence each of the two read and one write addresses need to be three bits wide.

Also, register 0 should always be zero[1]. Specifically, when register address 0 is supplied on any of the read ports, the corresponding output should be zero, and when register address 0 is supplied on the write port, the corresponding input data should be ignored (not written to any register).

The inputs to the register file are clk, reset, wr, rd_addr_a[2:0], rd_addr_b[2:0], wr_addr[2:0], d_in[15:0], and its outputs are d_out_a[15:0], d_out_b[15:0]. rd_addr_a and d_out_a form read port a just as rd_addr_b and d_out_b form read port b. wr, wr_addr and d_in form the write port.

Read operation: For port a, the address of the register to be read is applied to rd_addr_a and at the next positive clock edge, the contents of

[1]Processor architectures such as MIPS, and the legendary CDC 6600, had this feature of a register 0 permanently tied to 0. Doing so can optimize some computation, such as not initializing a register to zero before using it, zero being somewhat frequently required.

the corresponding register should be available on d_out_a. Port b should operate in the same manner.

Write operation: When wr is high the value applied to d_in should, at the next clock positive clock edge, be stored in the register whose address is specified by wr_adder. When wr is low, inputs d_in and wr_adder should be ignored.

So in this manner the register file should be able to perform two read operations and one write operation (if wr is high) *every clock cycle.*

You can construct the registers using the dfrl flip flop (positive edge triggered D flip flop with reset and load enable) supplied in lib.v. All dfrl clock and reset inputs should be connected to the global clock and reset inputs supplied to the reg_file module.

2 Integrate with ALU

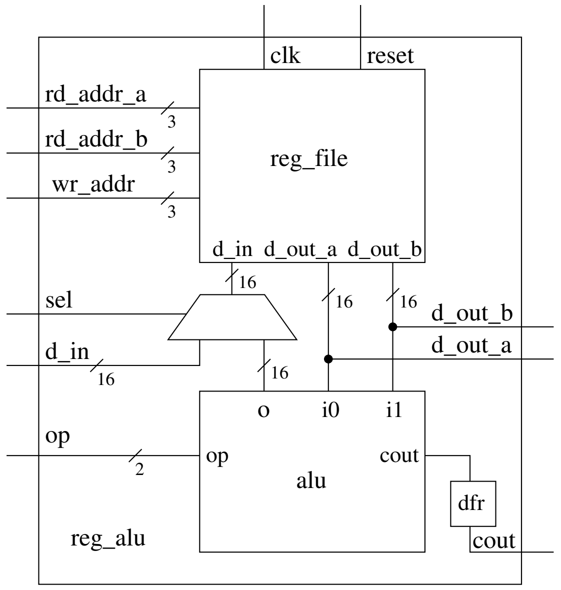

Figure 1: Integration of alu and reg_file to form reg_alu.

The two read outputs and the write input of the of the register file implemented above need to be connected respectively to the two inputs and one output of the ALU implemented in the previous assignment. Also, the register file read and write ports would need "outside" connection as well so that operands can be written in the register file and the results read out. The register file thus needs to be integrated with the ALU, in a straightforward manner, as shown in figure 1.

Inside the reg_alu module one simply instantiates the alu and reg_file modules and also the 2:1 16-bit mux to mediate the writes into the register file, and wires up the above three instantiations as shown in the figure. The mux is required to select whether data to be written to the reg_file (d_in) comes from the alu or “outside”, the sel control input to the mux being used to make the selection. Also, the cout alu output is registered in a flip-flop[2]. All the inputs and outputs of reg_file are inputs and outputs of the reg_alu module (d_in reg_alu input being routed via mux to d_in reg_file input, all other inputs and outputs are connected directly). In addition reg_alu has as inputs the mux sel described above and the op input to the alu. Finally reg_alu also has as an additional output the registered cout.

The reg_file, reg_alu and other related modules need to be submitted in the reg_alu.v file.

3 Design and Simulation

You can use the alu.v and augmented lib.v supplied for basic components. The reg_file module can be tested using the supplied tb_reg_file.v with the commands:

```
iverilog -o tb_reg_file lib.v alu.v reg_alu.v tb_reg_file.v
vvp tb_reg_file
```

The reg_alu module can be tested using the supplied tb_reg_alu.v with the commands:

```
iverilog -o tb_reg_alu lib.v alu.v reg_alu.v tb_reg_alu.v
vvp tb_reg_alu
```

Note that above reg_alu.v is used in both cases (it is okay for reg_alu.v to contain both reg_file and reg_alu even if you are testing just one of the two).

The general approach and the tools used remain the same as last assignment.

[2] In a microprocessor, an instruction may make use of the cout output of the previous instruction. So it is sufficient to store cout for only one clock cycle due to which no load enable to flip flop is required, so the dfr component in lib.v can be used (dfr, though not shown in figure, has to be supplied with clk and reset as well).