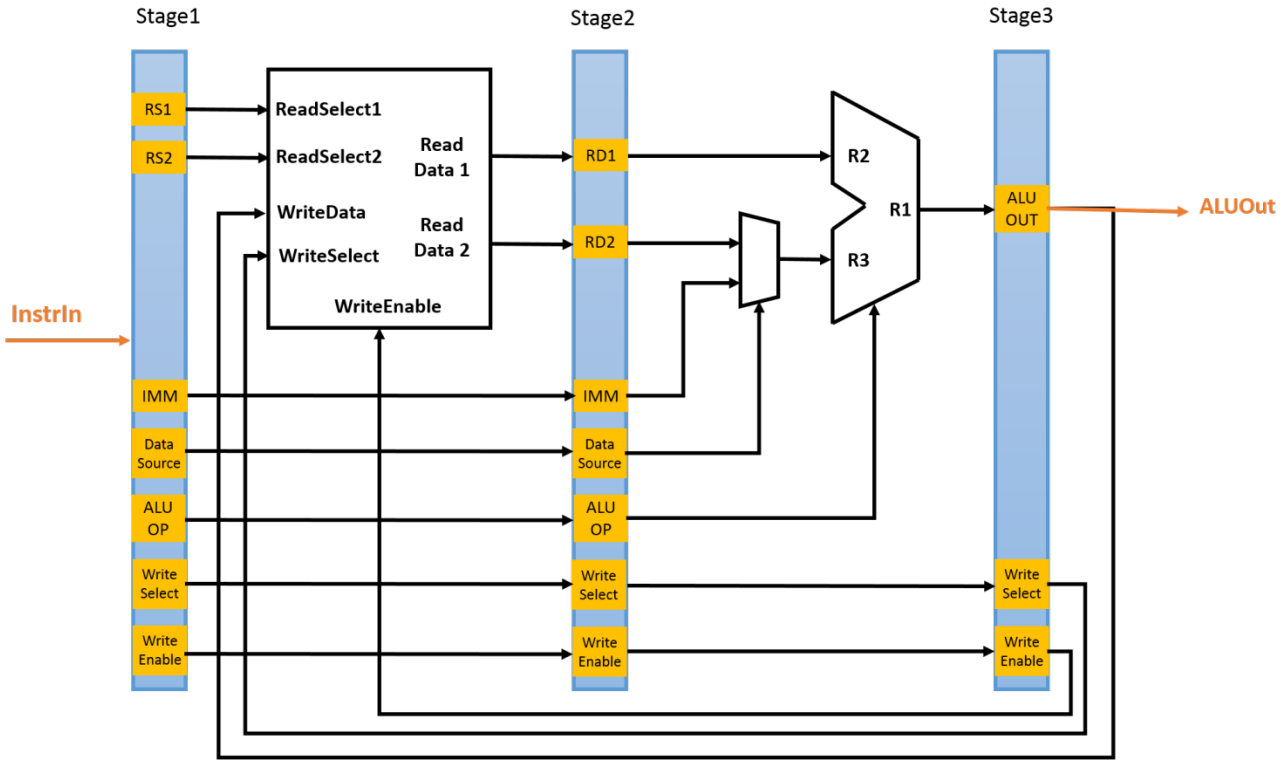
Lab 5 – EC413 Computer Organization

Overview

The purpose of Lab 5 is to gain additional experience with digital design and with how computers work by building, debugging, and testing a 4-stage 32-bit pipelined datapath (shown in this figure).



Registers and buses

A good practice is to name your registers with a prefix telling which pipeline stage they belong to: S1_RS1, S1_RS2, S1_IMM, S1_ALUOP, S1_WS, S1_WE, S2_RD1, S2_RD2, etc. You should take a similar approach to labeling your buses. The table below shows a number of the internal buses and their widths. You can extrapolate the rest by following the paths. Interfaces to your top module are shown in yellow and are also shown in the first part of the table below. Lines which are originating in the pipeline stages are shown in blue and are also shown in the second part of the following table. Not shown are reset, clock, and logic to route the instruction from the testbench to the Stage1 registers.

InstrIn	32 bit	Instruction Input.
ALUOut/WriteData	32 bit	ALU output.
Reset	1 bit	Resets all sequential logic to zero.
Clk	1 bit	Clock for all sequential logic (not shown in diagram).
ReadSelect1	5 bit	Register select for first read register.
ReadSelect2	5 bit	Register select for second read register.
WriteSelect	5 bit	Register select for input register.
Imm	16 bit	Immediate data.
DataSrc	1 bit	Data source selection for second operand.
ALUOp	3 bit	ALU operation as in Lab 4.
WriteEnable	1 bit	Register file write enable.
ReadData1	32 bit	Register output 1
ReadData2	32 bit	Register output 2

Operation

In a pipeline running in steady state, components in each pipeline stage are active all the time, each executing a different instruction. For example, at the same time: one instruction is driving the S1 register inputs, a second instruction is driving the S2 inputs (with some signals going through the register file), a third instruction is driving the S3 inputs (with some signals going through the MUX and the ALU), and a fourth instruction driving the input ports of the register file. When the clock edge rises, all of these signals are latched simultaneously and each instruction either completes or starts driving the next stage.

Instruction Format

All instructions are 32-bit, and the format is the following.

Type	**31**	**format (bits)**			**0**
R	opcode(6)	r1(5)	r2(5)	r3(5)	
I	opcode(6)	r1(5)	r2(5)	imm(16)	

For this lab, you will only be implementing R-Type and I-Type instructions. Also, for simplicity, the interpretation of the opcode field will not be the same as MIPS, it will instead have the following format:

- The most significant bit (31) is unused.
- The next most significant bit of the opcode (30) determines whether the instruction is Logical/Arithmetic (1) or other (0). This will always be set for this lab.
- The third bit of the opcode (29) determines whether the instruction uses an immediate operand (1) or not (0). That is, if this bit is set, the instruction follows the I-type template.
- The least significant 3 bits of the opcode field (28-26) are fed into the ALU's ALUOp input.
- The ALU has the following functions:

AOp2	**AOp1**	**AOp0**	**Output**	**Function Name**
0	0	0	R1 = R2	MOV
0	0	1	R1 = ~R2	NOT
0	1	0	R1 = R2 + R3	ADD
0	1	1	R1 = R2 - R3	SUB
1	0	0	R1 = R2 | R3	OR
1	0	1	R1 = R2 & R3	AND
1	1	0	R1 = 1 if R2 < R3, else 0	SLT (signed)

Requirements

1. Behavioral Verilog OK for this lab.
2. You are free to use the provided modules:
    a. DFF
    b. nbit_reg
    c. nbit_mux
    d. nbit_demux
    e. nbit_register_file
3. Your design must be hierarchical (i.e., using blocks which you can reuse).
4. Your timing diagrams should include several instruction sequences with representative examples and corner cases for each function.

Recommended Order of Work (including prelab)

1. Paper test case. Create a sequence of four instructions (or use the one given below). Trace execution of each instruction individually, clock by clock. Then trace them through together. This should be done in enough detail so that you can use this paper test to verify your final design. A really good way to do this is with a spreadsheet.

2. Create testbench for the register file and test. This should include exercising read/write and enable.
3. Link register file with input and output registers (see #1) and test. Note that by doing so you are testing both Stage2 and Stage4 of the pipeline.
4. Retest ALU from Lab 5.
5. Add MUX, input, and output registers. Test the resulting ALU assembly (Stage3).
6. Assemble and test individual instructions. You can initialize the register file with MOV instructions.
7. Create small programs for which you can verify correctness of results.

Data Hazards and Forwarding

Pipelining a datapath can cause it to execute programs incorrectly. In particular, when an instruction writes to a register and one of the next instructions reads the same register, those later instructions will be reading old (stale) data. The solution involves methods called *data forwarding* which we cover later in the course. For now, don't worry about these cases.

Deliverables and Grading

For the report, please write up a formal report with introduction and subsections describing your design and how you tested it.

Style

An important component of your grade will be style. For example:

- Avoid unnecessary copy-pasting.
- Keep your top level module as simple as possible.
- Comment your code
- Describe inputs and outputs.
- A name given to a signal should explain its function: e.g. clocks should be labeled as Clk and not C or CL.

Grading guidelines

- Prelab [10 points]
- Demonstrate complete system functionality [40 points]
- Report
    - Description of how you tested the register file with waveforms [10 points]
    - Description of how you tested the ALU with waveforms [10 points]
    - Description of how you tested the datapath and timing diagrams [30 points]
    - Report formatting, code style [20 points]

Sample instruction sequence

```
InstrIn = 32'b011010_00001_00001_0000000000001010;   // I type, ADD r1 with 0000000A =>   r1 = 0000000A
InstrIn = 32'b011100_00010_00010_0000000000000010;    // I type, OR r2 with 00000002  =>   r2 = 00000002
InstrIn= 32'b010010_00011_00001_00010_00000000000;  // R type, ADD r1(0000000A) with r2(00000002) =>  r3 =
0000000C
InstrIn= 32'b010011_00100_00001_00010_00000000000;   // R type, SUB r1(0000000A) with r2(00000002) =>  r4 =
00000008
```