Lab Assignment #2

EE-126/COMP-46: Computer Engineering w/lab
Professor: Mark Hempstead **TA:** Parnian Mokri, Jessica Nordlund
Tufts University, Fall 2022

Due as per the class calendar, via 'provide'

The ultimate goal of this project is to design a pipelined version of a ARM processor that can detect control and data hazards. The functionality of the processor and its components should match the descriptions in the textbook unless otherwise noted. All work must be your own; copying of code will result in a zero for the project and a report to the administration.

Overview

List of assignments

- Lab 0: Set up modelsim, simulate an AND gate with 2 inputs and 1 output
- Lab 1: Basic Processor Components and Testbenches
- **Lab 2: Remaining Processor Components including ALU, Memories, and control logic**
- Lab 3: Implementation LEGv8: a single cycle processor that executes a subset of ARM v8-64bit ISA
- Lab 4: Pipelined processor with no hardware hazard detection
- Lab 5: Overcoming data-hazards using forwaring and stalling
- Lab 6: Overcoming control hazards by resolving conditional/unconditional branches in ID and using flushing
- Lab 7: Advanced Topics: open-ended team project (groups up to 2 people)

Lab Submission

Please submit your VHDL files *and* a PDF report via 'provide' command on the EE/CS machines. Please follow the announcement on Canvas about Provide to submit your labs and pay attention to messages you get when you try to provide.
VHDL Files: Submit the VHDL source files **(*.vhd)**[1] and any dependencies thereof. Use the entity descriptions provided at the end of this document.[2] These descriptions can also be found in assignment2.zip.
Scripts/Makefile/README: The course staff needs to know how to compile your code and run your testbenches. Please include a README and/or runscripts/Makefile. In addition, do no change the entitity definitions or it will not run with our tests
Report: Submit your report as a PDF**(*.pdf)**. Demonstrate the functionality of your code by providing waveforms as detailed in the Deliverables Section. Label/annotate important signals and events in your waveforms and then provide a brief description of what is happening.

Lab2 Objectives

- Understand the functionality of the provided DMEM (dmem.vhd) component and complete it
- Write a testbench for DMEM
- Implement ADD (as an Adder), ALU Control, ALU, CPU Control, Instruction Memory, Registers in VHDL
  - *HINT*: Use dmem.vhd as a starting point for IMEM and Registers

[1]Do NOT submit your entire Modelsim project (including but not limited to ***.mpf** and files in **work/**)
[2]Submissions that fail to to follow any of these directions may be penalized at the discretion of the grader. If you have questions, contact the TA (Parnian Mokri: parnian.mokri@tufts.edu).

- Verify functionality of each component via simulation in Questasim

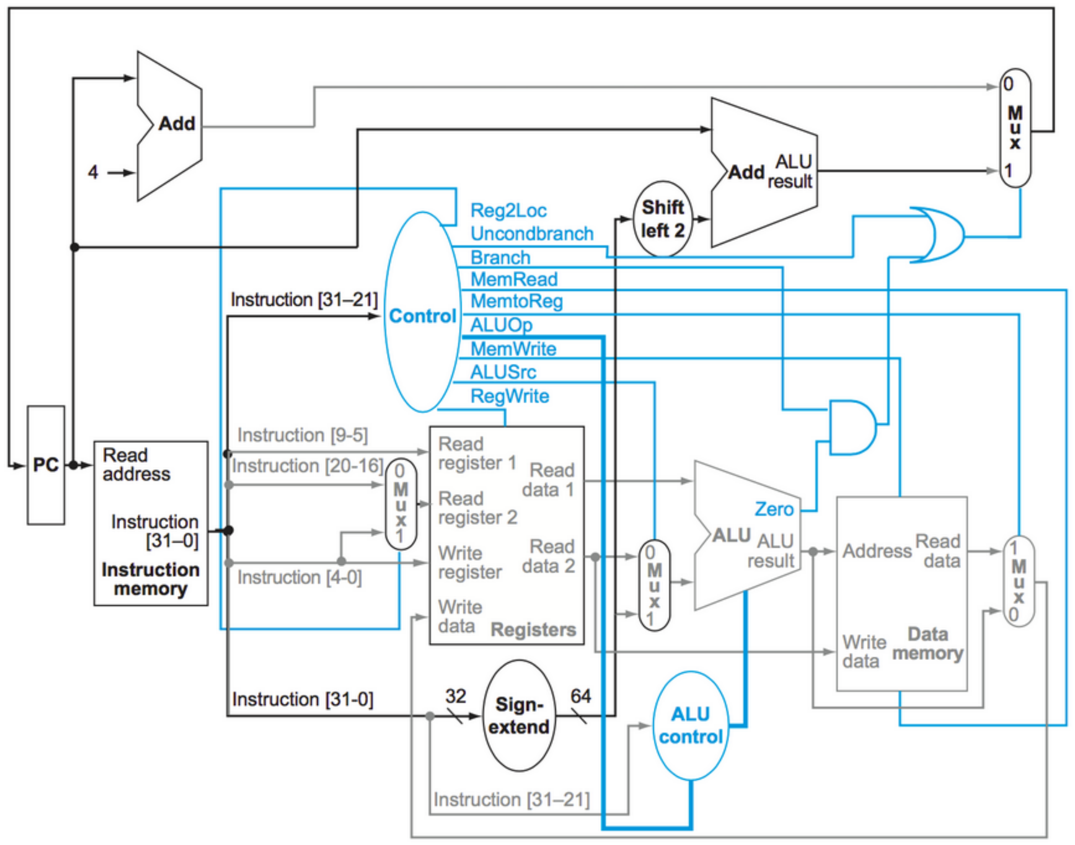
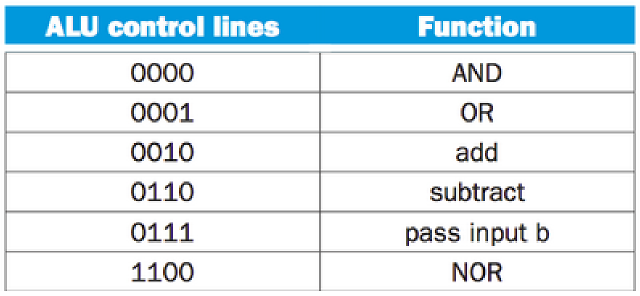


Figure 1: **Components of ARM processor for Lab 2.** NOTE: The DMEM component will be given to you, but you must submit a testbench for it. This is Figure 4.23, page 282, in the textbook.



ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	pass input b
1100	NOR

Figure 2: ALU control truth table in Section 4.4

Deliverables

VHDL Files:

1. ADD, ALU Control, ALU, CPU Control, Instruction Memory, Registers. [3]
2. ALU control implements the control logic described in the textbook Section 4.4 and in Figure 2.
3. CPU control implements the control logic defined in the textbook Section 4.4 and in Figure 4.18 of your textbook.

[3]You do *not* need to submit your testbenches for these components, but making them is *highly* recommended to facilitate simulation, debugging, and waveform generation.

4. The Adder, adds two 64 bit numbers and it needs to have structural implementation
5. ALU needds to be using the Adder you designed See Figure 1 for the detailed diagram.
6. Compeleted DMEM.vhd, and a testbench for DMEM that writes to multiple adresses and then reads out the stored data.

Report:
Include brief descriptions and annotated waveforms that demonstrate the functionality of the following:

- ALU
  - 'and' and 'or' operations (display values in binary)
  - 'add' and 'subtract' operations (display values in decimal or hex)
  - functionality of 'zero' and 'overflow' output flags (show at least one case where each is '1')
- Registers
  - Initialize \$X9=0 \$X10=1 \$X11=2 \$X12=4 \$X13=8 \$X14=16 \$X15=32 and the saved registers: \$X19=8 \$X20=0 \$X21=2 \$X22=4 \$X23=16 \$X24=32 \$X25=64 \$X26=128 \$X27=128
    - *Hint*: look at the *if(first)* code block from dmem.vhd for an example of initialization
    - *Hint*: DEBUG_TMP_REGS should be 0x0000000000000000 -0x0000000000000001-0x0000000000000002-0x0000000000000004 and so on
  - Write to and read from some of the registers. Label where reads/writes occur!
  - Attempt to write register XZR and show that such writes fail (Reg XZR) is always 0)
- IMEM (using your testbench)
- DMEM (using your testbench)
  - Include annotations and a brief description of each read/write event

Entity Descriptions **(provided in assignment2.zip)**

Note that some of these entities have output signals beginning with "DEBUG". These signals will be used for testing purposes and do not impact the functionality of the components, but they need to be hooked up properly or the tests will fail. The comments around the DEBUG signals will tell you how they should be connected. In these comments, the '&' character means concatenation and is also the concatenation operator in VHDL. The provided dmem.vhd file shows an example of properly connecting internal signals to the DEBUG ports.
The size of imem an dmem will change for future labs. Also, note that the dmem we are giving you is not complete. You need to follow the instructions in the file and complete it before simulating it.

ADD

```
entity ADD is
-- Adds two signed 64-bit inputs
-- output = in1 + in2
port(
     in0    : in  STD_LOGIC_VECTOR(63 downto 0);
     in1    : in  STD_LOGIC_VECTOR(63 downto 0);
     output : out STD_LOGIC_VECTOR(63 downto 0)
);
end ADD;
```

ALU Control

```
entity ALUControl is
-- Functionality should match truth table shown in Figure 4.13 in the textbook.
-- Check table on page2 of Green Card.pdf on canvas. Pay attention to opcode of operations and type of operations.
-- If an operation doesn't use ALU, you don't need to check for its case in the ALU control implemenetation.
--  To ensure proper functionality, you must implement the "don't-care" values in the funct field,
-- for example when ALUOp = '00", Operation must be "0010" regardless of what Funct is.
port(
     ALUOp     : in  STD_LOGIC_VECTOR(1 downto 0);
     Opcode    : in  STD_LOGIC_VECTOR(10 downto 0);
     Operation : out STD_LOGIC_VECTOR(3 downto 0)
    );
end ALUControl;
```

ALU

```
entity ALU is
-- Implement: AND, OR, ADD (signed), SUBTRACT (signed)
--    as described in Section 4.4 in the textbook.
-- The functionality of each instruction can be found on the 'ARM Reference Data' sheet at the
--    front of the textbook (or the Green Card pdf on Canvas).
port(
     in0       : in     STD_LOGIC_VECTOR(63 downto 0);
     in1       : in     STD_LOGIC_VECTOR(63 downto 0);
     operation : in     STD_LOGIC_VECTOR(3 downto 0);
     result    : buffer STD_LOGIC_VECTOR(63 downto 0);
     zero      : buffer STD_LOGIC;
     overflow  : buffer STD_LOGIC
    );
end ALU;
```

CPU Control

```
entity CPUControl is
-- Functionality should match the truth table shown in Figure 4.22 of the textbook, inlcuding the
--    output 'X' values.
-- The truth table in Figure 4.22 omits the unconditional branch instruction:
--    UBranch = '1'
--    MemWrite = RegWrite = '0'
--    all other outputs = 'X'
port(Opcode   : in  STD_LOGIC_VECTOR(10 downto 0);
     Reg2Loc  : out STD_LOGIC;
     CBranch  : out STD_LOGIC;  --conditional
     MemRead  : out STD_LOGIC;
     MemtoReg : out STD_LOGIC;
     MemWrite : out STD_LOGIC;
     ALUSrc   : out STD_LOGIC;
     RegWrite : out STD_LOGIC;
     UBranch  : out STD_LOGIC; -- This is unconditional
     ALUOp    : out STD_LOGIC_VECTOR(1 downto 0)
);
end CPUControl;
```

Instruction Memory (read only)

```
entity IMEM is
-- The instruction memory is a byte addressable, little-endian, read-only memory
-- Reads occur continuously
-- HINT: Use the provided dmem.vhd as a starting point
generic(NUM_BYTES : integer := 128);
-- NUM_BYTES is the number of bytes in the memory (small to save computation resources)
port(
     Address  : in  STD_LOGIC_VECTOR(63 downto 0); -- Address to read from
     ReadData : out STD_LOGIC_VECTOR(31 downto 0)
);
end IMEM;
```

Registers

```
entity registers is
-- This component is described in the textbook, starting on section 4.3
-- The indices of each of the registers can be found on the LEGv8 Green Card
-- Keep in mind that register 31 (XZR) has a constant value of 0 and cannot be overwritten
-- This should only write on the negative edge of Clock when RegWrite is asserted.
-- Reads should be purely combinatorial, i.e. they don't depend on Clock
-- HINT: Use the provided dmem.vhd as a starting point
port(RR1      : in  STD_LOGIC_VECTOR (4 downto 0);
     RR2      : in  STD_LOGIC_VECTOR (4 downto 0);
     WR       : in  STD_LOGIC_VECTOR (4 downto 0);
     WD       : in  STD_LOGIC_VECTOR (63 downto 0);
     RegWrite : in  STD_LOGIC;
     Clock    : in  STD_LOGIC;
     RD1      : out STD_LOGIC_VECTOR (63 downto 0);
     RD2      : out STD_LOGIC_VECTOR (63 downto 0);
     --Probe ports used for testing.
     -- Notice the width of the port means that you are
     --      reading only part of the register file.
     -- This is only for debugging
     -- You are debugging a sebset of registers here
     -- Temp registers: $X9 & $X10 & X11 & X12
     -- 4 refers to number of registers you are debugging
     DEBUG_TMP_REGS : out STD_LOGIC_VECTOR(64*4 - 1 downto 0);
     -- Saved Registers X19 & $X20 & X21 & X22
     DEBUG_SAVED_REGS : out STD_LOGIC_VECTOR(64*4 - 1 downto 0)
);
end registers;
```

Data Memory Implementation(provided in assignment2.zip)

Note: You should *not* modify this file except for the initialization of **dmemBytes** inside the *if(first)* block. DMEM that you get on zip file is partially initialized, add enough byte to it so that the total size is 1KB. You should follow the pattern and complete it.
Don't copy this file from the pdf, copy it from the zip file.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL; -- STD_LOGIC and STD_LOGIC_VECTOR
use IEEE.numeric_std.ALL; -- to_integer and unsigned

entity DMEM is
-- The data memory is a byte addressble, little-endian, read/write memory with a single address port
-- It may not read and write at the same time
generic(NUM_BYTES : integer := 64);
-- NUM_BYTES is the number of bytes in the memory (small to save computation resources)
port(
WriteData          : in  STD_LOGIC_VECTOR(63 downto 0); -- Input data
Address            : in  STD_LOGIC_VECTOR(63 downto 0); -- Read/Write address
MemRead            : in  STD_LOGIC; -- Indicates a read operation
MemWrite           : in  STD_LOGIC; -- Indicates a write operation
Clock              : in  STD_LOGIC; -- Writes are triggered by a rising edge
ReadData           : out STD_LOGIC_VECTOR(63 downto 0); -- Output data
--Probe ports used for testing
DEBUG_MEM_CONTENTS : out STD_LOGIC_VECTOR(64*4 - 1 downto 0)
);
end DMEM;
```

Tips and tricks

1. We are providing an example of half adders. This will show how to wire different components please refer to the zip file.
2. Design the Adder, don't use "+" operation.
3. XZR The register X31 is hardwired to the constant 0
4. In ARM, the opcode field has different length, you can use don't care to make your design easier.
5. Adding debugging signals to memories are encouraged
6. Using Hex to initialize memories is acceptable.
7. Path to questasim license: /usr/cots/mentor/modeltech/modelsim.ini