EE 3613 – Fall Semester – AY 2024-2025

Project Report (teams of 2)

Instructions: convert this document to a PDF and include it in the zip file with the code. If you are working on your own, just fill in one row below.

Name	PID	% contribution to the project
Kavanaugh Frank	P101032426	50
Matthew Proboski	P101053601	50

If each team member did not contribute equally, explain the reasons below:

Statement of Originality

I, Kavanaugh Frank and Matthew Proboski (sign or type name here) **have neither received nor given unauthorized assistance during the completion of this work**.

THIS SECTION FOR OFFICE USE ONLY

Solution (1-2 pages):

Briefly describe your solutions to all the problems here. Discuss your methodology for each problem here and mention what worked and what didn't. **Include any waveforms from EPwave where necessary**.

EE 3613 Final Project Report: Matthew Proboski and Kavanaugh Frank

Problem 1

1-Bit ALU Module: Our implementation of the 1-bit ALU module can support five different operations (this includes AND, OR, add, subtract, and set-less-than) that is controlled by a 3-bit operation code. For the operations that involve arithmetic, propagate and generate signals were implemented in order to support carry-lookahead addition.
Overflow Detection Module: The unit for overflow detection tracks addition and subtraction operations by monitoring the most significant bits of both the operands and the result. This was implemented by using a case statement which checks for specific bit patterns that would tell if overflow occurs.
Carry Lookahead Adder (CLA) Module: The CLA module that we implemented supports carry lookahead logic for a 4-bit addition which in turn achieves faster computation when compared with a ripple carry approach. It calculates carry signals using generate and propagate terms at each bit position.
4-Bit ALU Module: Our 4-bit ALU module uses four 1-bit ALUs in addition with the CLA module to complete operations on 4-bit numbers. We completed the linkage between these components by connecting the generate and propagate signals from each of the 1-bit ALUs to the CLA.

Problem 2

16-Bit ALU Module: The 16-bit ALU module we created was made by connecting four 4-bit ALU modules together, with carry propagation being handled by the CLA module to achieve improved performance. We also implemented special handling for the SLT operation by using overflow detection.
32-Bit ALU Module: Our 32-bit ALU was built by connecting our two 16-bit ALU modules with correct carry propagation in between them. We made sure to keep overflow detection at the most significant bit and implement zero detection for our branch operations.
Sign Extension Module: In order to convert 16-bit immediate values to 32-bit values we implemented the sign extension unit by keeping and extending the most significant bit of the input properly.
32-Bit and 5-Bit Multiplexer Modules: Both multiplexers in our design use conditional operators to select between two inputs based on a control signal, which enables efficient data routing throughout our design.

Problem 3

Control Logic Module: The module for our control logic generates the right control signals based on the given instructions opcode. We implemented signal generation for many types of instructions including: R-type instructions, load/store operations, branches, and jumps.
ALUOpToALUControl Module: This module is used to translate the 2-bit ALUOp signal and function fields into a unique 3-bit ALU control signal, and supports all the required ALU operations.
Register File Module: Our register file implementation includes 32 registers with synchronous write operations and asynchronous read operations, and also includes special handling for the zero register.
Data Memory Module: Our data memory module we created includes a 128-word memory array with an synchronous write operation and an asynchronous read operation which is controlled from the MemRead and MemWrite signals.
Combined Control Module: To improve efficiency and simplicity, we combined the control logic and ALU control into one module while at the same time maintaining the same functionality as the separate modules.

Problem 4

Program Counter Module: Our implementation of the program counter as a register that updates its output value on the positive edge of the clock signal allows for simple sequential instruction execution.

PC Adder and BEQ Adder Modules: Both of these modules implement elementary addition for incrementing the program counter by 4 and calculating branch target addresses.
Left Shifter Modules: Our implementation has two left shifter modules: one that is used for jump instructions (26 to 28 bits) as well as another for branch instructions (32 to 32 bits). both keep proper word alignment by shifting left two positions.

Problem 5

In order to wire up all the components correctly, we took a look at a single-cycle MIPS CPU data flow diagram in great detail and slowly made all the connections needed. We then “programmed” the CPU by pre-loading a very basic instruction (ADD instruction) into the instruction memory. We then simply set the program counter to 0x0000 and tested our completed implementation in a testbench. This first test exposed a lot of simple mistakes in our CPU that needed to be addressed. These issues included, but are not limited to, incorrect inputs to Multiplexors, incorrect timing for the pcCLK and CLK, and incorrect formatting of the Program Counter and the Instructions in the Instruction Memory. After these were fixed, however, it worked as expected.

Problem 6

This part exposed many more complex problems that we made in our design. In this part we didn’t just need to simply add two numbers, but we needed to do it repeatedly by utilizing the Jump and Branch equal instruction. Also, setting the program counter to 0x1000 caused a lot of issues that took a lot of debugging to fix. One additional thing that we had to change was we only allocated room for a single data memory location, so instead of having a Data Memory array from 0 to 0x4000 we initialized the array from 0x4000 to 0x4001. This change was made for the convenience of faster run times. Another issue to be fixed was the calculation of the Jump and Branch address. Another error that we ran in to was the fact a Multiplexor was not created, and instead the Read Data from the Register files was directly wired to the ALU. This was solved after printing the Read Datas from the Register File along with the Jump and Branch Addresses, which were the calculations that were incorrect.