Group assignment 3 - ARK

Martin Bjerregaard Jepsen - 251190
Rasmus Wriedt Larsen - 070290
Jonas Brunsgaard - 141185

October 25, 2012

G3 Implementing a simple microarchitecture

Introduction

In this assignment the team has been building the circuits for a micro processor. It is our perception, that this report and the enclosed zip archive answers all the questions asked in the third group assignment of the course. Thus we find the assignment to be fully answered.

Basics

Our microarchitecture is an implementation of a single-cycle processor as described in COD and illustrated on page 329. Because the memory for data and instruction is shared, we have to use 2 clock cycles for the `lw` and `sw` instructions – we've already used the memory to fetch the instruction.

This is implemented by having a boolean, `Run Instr`, indicating if we in the current clock cycle should run instructions or access memory for a `lw`/`sw`. We keep this value in a register, and update it on every clock cycle, only deasserting it when either of the control lines `MemRead` or `MemWrite` are asserted.

We also store values for the things needed by the lw/sw instructions:

sw/lw: The address (ALU Result) – "lw/sw Addr"

sw: Data (Reg B) – "sw Data"

lw: register write – "lw rW"

sw/lw: if we're currently doing sw or lw – "is sw"

If `Run Instr` is deasserted, we take the appropriate action depending on `is sw`, and issue a NOP to the rest of the hardware – so it's very important that the control lines `MemRead` and `MemWrite` are deasserted when a NOP occur, else the hardware would be stuck in the "don't-run-instructions-this-clock-cycle" state.

Implementing I-instructions

I-instructions were implemented by adding extra control logic that asserted control lines such that the immediate values were fed to the ALU.

Implementing Branch on not equal

The `beq` instruction was implemented as specified in COD. To implement the `bne` instruction, an additional control line called `ZeroInvert` was added. Asserting this control line will invert the Zero result given by the ALU, thus enabling us to reuse the `beq` functionality to implement `bne`.

Implementing jumps

Basic jumps are implemented as specified in COD on page 328. To implement the `jal` instruction a mux was added to the register file write address input. This mux selects $31 if the `jump` flag is asserted. A mux was also added to choose `PC` + 4 as the write data instead of the ALU result if the `jump` flag is asserted. The `jr` instruction was implemented by adding a `RegtoPC` flag to the ALU control. This was necessary as the instruction is an R-format instruction, and thus cannot be processed in the main control. When the `RegtoPC` flag is asserted, a mux chooses the register A value as the next PC value.

Testing the implementation

To test our processor we wrote up some Assembly test files using the MIPS instructions that we have implemented.[1] Afterwards we constructed a circuit[2] in Logisim converting the Assembly instructions to byte code.

The bytecode was then loaded into the PC ram and executed in Logisim by enable ticks. Afterwards we compared the values in the registers with the excepted values. If the register held the expected values we concluded that the instruction had been implemented correctly. All passed tests can be seen in the 'tests/testresults.txt' file.

[1] These files can be found in the directory 'tests' in the zip package

[2] The circuit can be found in the file 'asm_to_ram.circ' which is part of the zip package