TDT4255 Assignment 3

Knut Halvor Skrede & Ole Magnus Ruud

November 13, 2010

Abstract/Summary

The task was to implement a simple multicycle CPU. A top level architecture was proposed in the assignment text, we chose to implement the CPU in this way. We could use the ALU we made from the previous assignment, but we chose to use the one included in the support files. The CPU should load instructions into memory and execute them. A program counter will be used to walk through the instructions. A BNZ (branch if not zero) instruction can also be used, this instruction will load the program counter with a given address if the zero flag in the status register is not set. Additionally the CPU should implement a LDI (load immediate) instruction that loads the the register with a value. The cpu should also implement the ALU functionality.

We chose to use the instruction word layout proposed in the assignment text.

The control unit implements a simple fetch-execute state machine.

The assignment was also to write a simple program to run on the CPU, we chose to implement fibonacci.

Introduction

The task of this assignment was to expand the design from the previous assignment with pipelining and a data storage memory, and make it work on Nalle. To do this, it was necessary to add registers between each pipelining stage containing all information and control signals needed for later stages. It was also necessary to take dependancy hazards into account by writing back results to where it is needed in the earlier pipelining stages.

We used the suggested solution as a starting point.

Suggested Solution

Design

Instruction format

As our instruction format we decided to use a slightly modified version of the suggested format from assignment 2, shown in figure 1.

- Opcode: used to determine the control signals in the cpu given different instructions.

Name:	Opcode	unused	funct	Imm	Rd	Ra	Rb
Bits:	31-29	28-24	23-20	19-12	11-8	7-4	3-0

Table 1: Instruction set format

- Funct: used for choosing the ALU instruction.
- Immediate: used for Load Immediate, load the register with the given value and for branch instruction to set the new value in the programme counter
- Rd : number of the register which will be written
- Ra : number of register A
- Rb : number of register B

Name	Opcode	Comment
ALU_INST	001	Writes alu result to register
BNZ	011	Loads pc register if zero flag is set
LDI	010	Loads register with value
LOAD	110	Writes register Ra to memory at address given by immediate
STORE	100	Loads register Rd with memory at address given by immediate

Table 2: Instruction set for the ALU, not including the alu functionality

Name	Funct	Comment
MOVE	0000	sets output of ALU out to input A
AND	0001	Bitwise AND
ADD	0111	Arithmetic ADD

Table 3: Function set for the ALU

Implemented functions

Instruction flow

We will here describe the basic flow of operations for an instruction through our processor architecture stage by stage without hazards handling.

Instruction fetch Instructions starts out by being fetched based on the program counter from the instruction memory to the IF/ID register. This is the complete basic operation of the instruction fetch stage.

Decode In the decode stage the most basic action is the forwarding of the Rd, Immediate and the function field to the ID/EX register. The Ra and Rb fields are sent to the regfile, and the corresponding outputs are written to the ID/EX A and B field. The immediate field is also connected to the data memory read address, as this value needs to be read on a rising edge because of the memory implementation.

Based on the opcode of the instruction, control has to write several control signals to the ID/EX register. These are divided into control signals for the Execute stage and control signals for the writeback stage.

The execute control signals are memory write enable, which is set for memory write instructions, and status write enable, which is set for ALU instructions.

The writeback control signals are source select and register write enable. The source select is the control signal for the register input selector, which is set to alu output for alu operations, memory output for load instructions and immediate for load immediate. The register write enable is set for all instructions resulting in a register write.

Execute In the execute stage the Rd, immediate and writeback control signal fields are forwarded from ID/EX to EX/WB.

The ALU combinatorially computes an output value based on the A, B and function fields of the ID/EX register, and stores it in an ALU field in the EX/WB register. The output from the ALU status is sent to the Status register along with the status register write signal from the ID/EX register which decides if the register is written.

The immediate value from the Decode stage has now been caught by the Data memory, and the value stored at this address is written to the memory field of the EX/WB register. The immediate, A and memory write enable fields of the ID/EX register are sent to the write address, write data and write enable inputs of the data memory, respectively. If the write enable is asserted, the data is written.

Writeback In the writeback stage the register file is written if the register write enable signal of the EX/WB register is asserted. The value

written is selected with a register input mux based on the source select signal in the EX/WB register with all the data fields of EX/WB as inputs. The register destination is decided by the propagated Rd field of the EX/WB register.

Hazards handling

A big part of this assignment was the Hazards handling

Approach

Our approach to the problem in this excercise was to construct all the modules described in the architecture diagram, starting with the simplest. This included the writeback muxes between the register file and the id/ex pipeline register, and the writeback muxes between the id/ex pipeline register and the ALU and data memory as we felt confident we were going to get the hazards handling working. Some of the modules like the program counter and status register we reused from the previous assignments.

When the muxes were finished, we had a good indication on what control signals would be needed for all the pipelining stages, and so we could implement the pipelining registers with all of these control signals.

The final and most difficult component we implemented was the control module setting all the control signals. When we thought we had it right everything was sewed together and we could test it with a testbench from the previous assignment. After some debugging in Modelsim we had a working architecture.

Implementation

Our implementation consists of the following components ordered by their appearance in the pipelining stages:

1. Program counter.
2. Instruction memory.
3. IF/ID pipeline register.
4. Control unit.
5. Register file.
6. Register to ID/EX muxes.

7. ID/EX pipeline register.
8. ID/EX to ALU and data memory muxes.
9. ALU.
10. Status Register
11. Data memory.
12. EX/WB pipeline register.
13. Register input mux.

We will now describe the components.

Program Counter

The program counter is the same one used in the previous assignment. It is a counter with an initial value of 0 after reset, with a built in mux choosing between a provided immediate value, and the incrementation of the current value. It also takes in a write enable that is not needed in the current architecture, but that would be if we decided to expand the instruction set in the future.

Instruction memory

The instruction memory is an instantiation of the provided memory component taking in output from the program counter and providing the instruction at the provided address.

IF/ID pipeline register

Since the instruction is stored on the output register of the instruction memory, the IF/ID pipeline register component is actually just an abstraction of this, sending most of its input straight through, and only registering the valid bit, provided by the control module.

Control Unit

The control unit is the most complex component in our design, as all the control signals in the decode stage is set here, as well as the future control signals for the execute and writeback stage that must be written to the

ID/EX pipeline register. The component is purely combinatorial, with outputs based on the current instruction in the IF/ID register and some fields in the other pipeline registers to handle hazards as described in the design section.

Register file

The register file is an instantiation of the delivered register file, with 2 combinatorial read adresses with corresponding outputs, and one sequential write address and data port.

Register to ID/EX muxes

Based on two control signals from the control unit, this double mux combinatorially decides the input of the two operand fields of the ID/EX register. It can either be from the register or the writeback signal from the writeback stage.

ID/EX pipeline register

This is a simple sequentially written register. The contents are described in the design section.

ID/EX to ALU and data memory muxes

This component works the same way as the Register to ID/EX mux, but the source of the mux control signals is the ex field of the ID/EX register, and the ouput is connected to the ALU and Data memory data input.

ALU

The alu is the simple handed out component.

Status Register

We use our Status Register implementation from the previous assignment, but have built in the writeback mux to be controlled by the write status register signal. When the status write signal is asserted, the mux will also choose the current status signal from the ALU, even before it is written to the register.

Data Memory

As with the Instruction memory, this is an instantiation of the handed out memory module, only used for data. This implies an 8-bit width. Because it needs the read address on the rising edge, we had to connect the read address port to the immediate value of the if/id register.

EX/WB pipeline register

As with ID/EX this is just a simple register. Fields are described in the design section.

Register input mux

Our last component is the muxer choosing what value of the EX/WB register to send to the register file.

Name	Opcode	Comment
ALU_INST	000	Writes alu result to register
BNZ	001	Loads pc register if zero flag is set
LDI	010	Loads register with value

Table 4: Instruction set for the cpu, not including the alu functionality

Testing

For testing we wrote the following programs:

Instruction flush test

This program tests if the instruction after a branch is successfully flushed. The program segment loads two values, 246 and 1. It then adds 1 to 246 and branches back to the add instruction until it reaches 255 and then overflows to 0. After the overflow add the branch should not be taken and the next instruction should be executed. If it fails, the register holding 1 will be loaded with 10 and the program will net perform as expected.

Datadependency test

This program test if the dataforwarding is successfull.

Line Nr	Opcode	funct	Imm	Rd	Ra	Rb
0	LDI		246	1		
1	LDI		1	2		
2	ALU_INST	ADD		1	2	1
3	BNZ		2			
4	LDI		10	2		

Table 5: Program to test branching

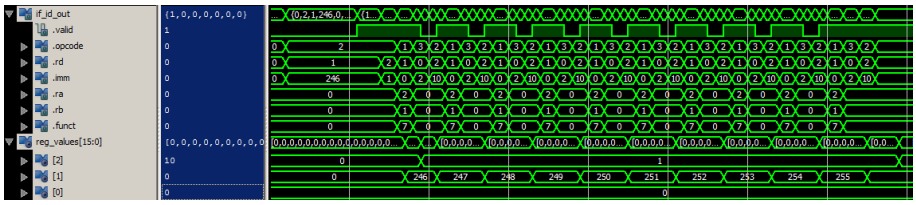

Figure 1: This figure shows the waveform from the testbench, most of the register signals has been removed, this is because they are not beeing used.

Status register test

This program test if the statusregister is successfully forwarded. It first loads 254 to register 1, 2 to register 2 and 1 to register 3. It then adds 254 and 1 and writes it to register 4, the result should because 255. It then adds 2 to 254, this operations should overflow and result in zero, resulting in the following branch instruction not to be taken. The instructions following this test should count upwards in the power of 2 until it overflows and stops. This test is located in file 'cpu_testbench3.vhd'.

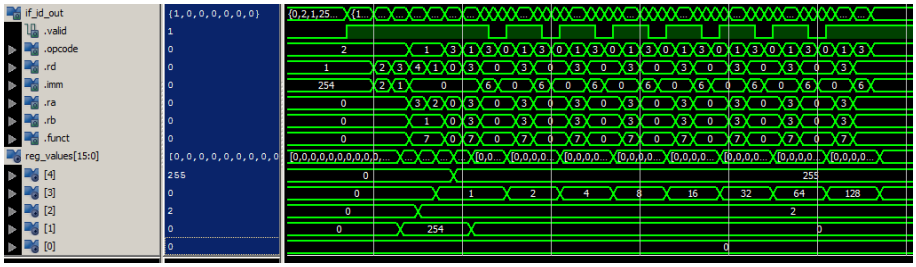

Figure 2: This figure shows the waveform from the testbench, most of the register signals has been removed, this is because they are not beeing used.

Line Nr	Opcode	funct	Imm	Rd	Ra	Rb
0	LDI		18	1		
1	LDI		2	2		
Test 1: Dataforwarding from writeback- to execute-stage (one-stage forwarding)						
2	ALU_INST	ADD		1	2	1
3	ALU_INST	ADD		1	2	1
4	ALU_INST	ADD		1	2	1
Test 2: Dataforwarding from writeback- to decode-stage (two-stage forwarding)						
5	ALU_INST	ADD		1	2	1
6	ALU_INST	ADD		1	2	2
7	ALU_INST	ADD		1	2	1

Table 6: Program to test data forwarding

Line Nr	Opcode	funct	Imm	Rd	Ra	Rb
0	LDI		254	1		
1	LDI		2	2		
2	LDI		1	3		
3	ALU_INST	ADD		4	3	1
4	ALU_INST	ADD		1	2	1
5	BNZ		0			
6	ALU_INST	ADD		3	3	3
7	BNZ		6			

Table 7: Program to test data forwarding

Synthesis

From the synthesis report we see that

1. The regfile muxer inferred 8 1-bit multiplexers, this was a little strange as we expected 1 8-bit multiplexer. This does not matter to the functionality though.

2. The sr module inferred one D-type flip flop, this is exatly what we wanted.

3. The pc module inferred one 8-bit counter, this is also exactly what we wanted.

4. The control module inferred 1 D-type flip flop, this is used to hold the current state (fetch or execute), the rest is combinatorial, so this is

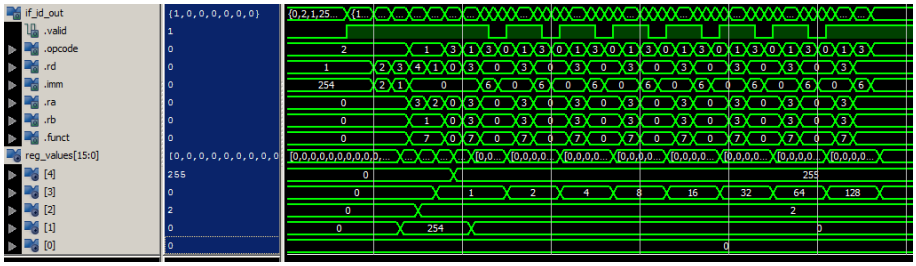

Figure 3: This figure shows the waveform from the testbench, most of the register signals has been removed, this is because they are not beeing used.

also what we wanted.

5. the toplevel and cpu modules did not infer anything, which was expected as they are simply mapping signals between the other modules.

The estimated maximum clock frequency was 61.747MHz.
Selected Device : v1000efg860-6
Usage of selected device:

Number of Slices:	252 out of 12288	2%
Number of Slice Flip Flops:	205 out of 24576	0%
Number of 4 input LUTs:	454 out of 24576	1%
Number of bonded IOBs:	54 out of 664	8%
Number of BRAMs:	2 out of 16	12%
Number of GCLKs:	2 out of 4	50%

We got no errors from the place and route report, all constraints met, and all signals routed.

Result

Everything seemed to work after a couple of days work, both simulation and testing on nalle.

However, we had previously tried some different solutions which did not work, and we realized that we did not understand why they did not work. We first tried to write to the registers on the rising edge of the clock, but this did not work. We then used the write enable signal as the clock on the registers and everything worked great. However, we changed this later, when we discovered what we did wrong earlier, which was having registers

on the output of the control unit. After that we removed the registers on the control unit and set the registers to be written on the rising edge of the clock, and everything worked the way we intended it to.

Evaluation

This assignment was very well defined compared to the previous. There was alot less guesswork involved during the design process. We found this to be more educational as we could focus on the tasks, in contrast to making tasks up.

Conclusion

It was easy to map everything together and create most of the components. The difficult part of this assignment was to figure out the timing and syncronization, what signals to register and when they where ready to be read. All in all this was a fun and educational exercise.

References

[1] http://en.wikipedia.org/wiki/Mealy_machine