Digital Design Lab Report

Klas Mannberg, Rickard Bemm
klaman-8@student.ltu.se, ricbem-8@student.ltu.se

Luleå university of technology
971 87 Luleå, Sweden

May 27, 2019

1 Introduction

In order to create a fully functioning processor, we must start at the bottom and work our way up in stages, designing and testing each component before the next one is added. This way ensures us each component is a solid building block for which the next component relies on and allows us to expand the project with minimal revision.

From a single component capable of adding two singular bits we have developed a full 32-bit subtraction and addition unit which in itself is just one part of an even bigger component ALU, which in turn is just a singular component of our finished MIPS. The report follows the stages of this process in order to achieve the end result of a fully implemented general purpose processor.

The process of constructing a component can be summarized to the following 3 steps:

1. **Design**
   We start by figuring out what component we need, and how its behaviour will be in the form of a schematic. Depending on the component, it itself usually has several subcomponents that need to be considered which is why it is important to design and implement a component fully before starting the next. The schematic will be relevant for the rest of the project hence the names of components, signals and ports need to be consistent and relevant to prevent uncertainty for future additions. The final step is usually using the declared in- and outputs and connecting it to the rest of the design.

2. **Implementation**
   Our designs are implemented as code written in VHDL. We use Vivado by Xilinx when coding as an interpreter and compiler. Beyond syntax help, Vivado also organizes the components and various tests which is invaluable when so many parts need to be considered. Behaviour and in/outputs have been planned during the design phase and are coded accordingly.

3. **Testing**
   After a component has been designed and implemented we run several tests on it using the behavioural simulation tool in Vivado. The test is very versatile and can be coded to send simulated signals to our inputs which allows us to observe the behaviour of each component. This allows us to make revisions based on the results and has been proven to be very effective for us when troubleshooting.

2 Logic design, simulation, and validation in VHDL

Simulation view, figure 1, from Vivado, consists of one 4bit input `X` signal and five output signals such that: *max* is HIGH when input `X` is 9 else LOW, *min* is HIGH when input `X` is 0 else LOW, **even** is the inverted value from first bit of input `X`, *lo3* is HIGH when input `X` is greater or equal to 0 and less than 3 else LOW, and *noBCD* is HIGH when input `X` is greater than 9 and less than or equal to $F_{16}(= 15_{10})$ else LOW.

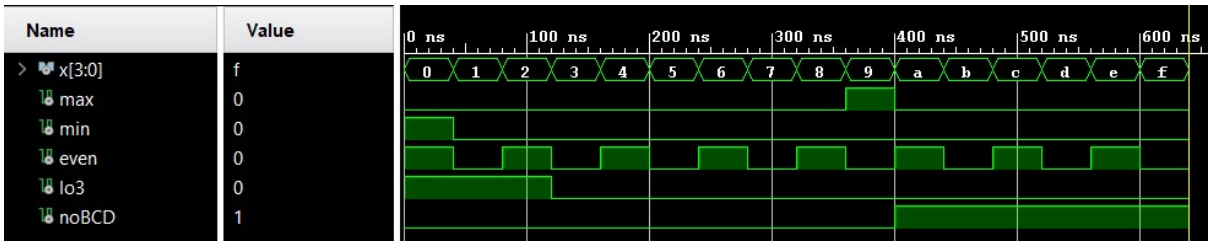

Figure 1: Simulation snapshot from Vivado of *bcdcheck2.vhd*.

The output signal *hieq3* was implemented such as

$$hieq3 = \overline{lo3}\,\overline{noBCD}. \tag{1}$$

As mentioned earlier *lo3* is HIGH when input `X` is greater or equal to 0 and less than 3 else LOW and *noBCD* is HIGH when input `X` is greater than 9 and less than or equal to $F_{16}(= 15_{10})$ else LOW. Therefore the eq 1 is correct.

BCDCheck1 is only using Boolean and-or expressions to calculate the results of each output signal. However *BCDCheck2* is using a 4-bit vector compared to four 1-bit input signals in *BCDCheck1*, the RTL schematic show that it is using four arithmetic units and two ROM's to calculate its output signals.

To implement the first function

$$F = B\overline{C}\,\overline{D} + \overline{A}\,\overline{D} + A\overline{C} + \overline{A}C. \tag{2}$$

To simplify the expression an inverted karnaugh map was used and we got

$$\overline{F} = \overline{A}\,\overline{C}D + AC \iff F = \overline{\overline{A}\,\overline{B}D + AC} \tag{3}$$

which we then implemented using a PLDcell by letting the first AND gate be `A and C and '1'` while the second AND gate being `A' and C' and D` with the inverted signal being true. The second function

$$G = \overline{A}\,\overline{B}\,\overline{C} + ABCDE \tag{4}$$

was supported by first using a PLDcell to check `A and B and C or '0' and '0' and '0'` in order to only be true when A and B and C are true, then a second PLDcell was added to check `(A and B and C) and D and E or A' and B' and C'`.

In VHDL the order of assignments does not matter, unless in a process then the assignments are sequential. This has been verified in Lab 2, part 1c.

3 The 4-bit arithmetic logic unit

When performing unsigned addition with an adder component the overflow cannot be used to detect errors nor overflow, the carry does indicate overflow and is not indicating any errors. All answers are mathematically correct when overflows are counted with when operating with unsigned numbers and incorrect output when operating with signed numbers. When performing an unsigned operation with the arithmetic component the carryout signal indicates overflow and the overflow nor the carryout signal are indicating on any errors.

The ALU was designed and implemented using 32 individual full bit adders in parallel where overflow signals being the carry input for the next, with the last two being used to detect overflow for the full 32-bit operation. Subtraction is supported after we add an XOR gate between each `B` input and our subtraction signal. The ALU also contains a 4-1 MUX to control what operation we get.

The SLT operation could theoretically already be supported here if two numbers subtracted equals a positive or negative number we can conclude if the number is smaller or greater or even equal if the result is zero.

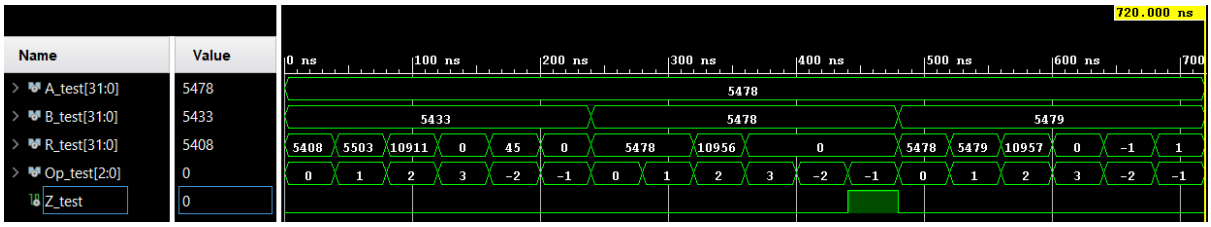

Figure 2: Simulation of ALU where A and B are 32-bit input signals, R is the result of the selected operation and Op is a 3-bit selector.

4 Program counter for MIPS processor

The program counter begins counting at 0, since the initial value stored in the register is 0, and increments it's output by 4 at the rising edge of each clock cycle. The output of program counter then becomes the memory address that executes the next instruction from our program memory, which is placed in a list with each instruction being an offset by 4 from the next. This allows us to add new instructions to our program memory and as long as the clock keeps cycling, they will be executed. A clock cycle is simply when the clock signal is toggled on or off once.

5 MIPS architecture supporting R and I-type instructions

The first iteration of a MIPS processor was created in lab 4 with support for AND, OR, ADDI, ADD, SUB, SLT and SLTI instructions. All of these instruction requires an OP-code and function code, except I-type instructions, which the control unit are using to decode and set its outputs.

Every R- and I-type instruction has an operation code which is the first six bits of the instruction. The control unit takes these six bits and controls various MUX components and

control signals. The control signals change the behaviour of some components and the MUX controls changes the path within our MIPS, this way the opcode can control what type of operation will happen.

Beside *Reset* and *Clk* signals, the implementation of R and I-type instructions required the control unit to output six other signals in order to control the data flow in the MIPS:

1. **RegDestination** is set to 0 when an I-type instruction is executed else if R-type the signal becomes 1. As the I-type instruction only contain two 5-bit addresses (see figure3), one save a word to and one to read a word from, and an immediate to operate with, the RegDestination signal choose between bits 20-16 or 15-11 in the instruction. This operation takes place in a 2 to 1 multiplexer before the third register address input.

2. **WriteEnable** is set to 1 when an instruction is about to write to the register. Which for the currently implemented instructions, always is 1 since the result of the executed instruction is stored in the register.

3. **ALUSource** is a selector signal for a multiplexer that selects either a word from the register when executing an R-type instruction or the immediate from an I-type instruction. The output of the multiplexer is received by the ALU.

4. **ALUControl** is a 3-bit control signal that selects an operation for the ALU to perform on the given inputs.

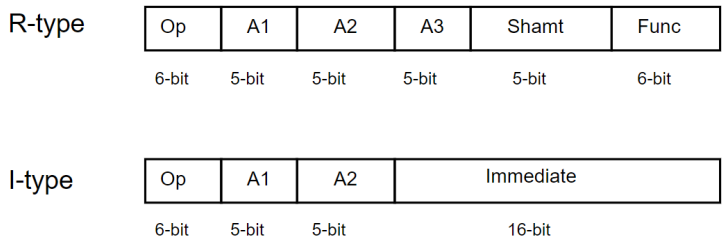

Figure 3: Different parts of the 3 supported instruction types.

The register file is an array of 32 registers, where each register can contain any 32-bit value. These registers will often either contain a result from an executed instruction or a value from which an operation will work. To store values we can use the ADDI instruction or any R-type instruction.

6 MIPS architecture supporting R, I, and J-type instructions

For our improved MIPS we have added support for load word, store word, branch if equal and jump instructions. To support load and store we must implement a new component Data-Memory.

Data memory is a 32-bit register array which operates with four inputs (*Adress*, *Clk*, *DataIn*, and *MemoryWE*) and one output (*DataOut*), see figure 5. Whenever a load instruction is executed the *MemToReg* MUX will be set to 1 and the signal from data memory will be selected, else if a save instruction is executed the *MemToReg* signal is a do not care since no data will be written to the register file.

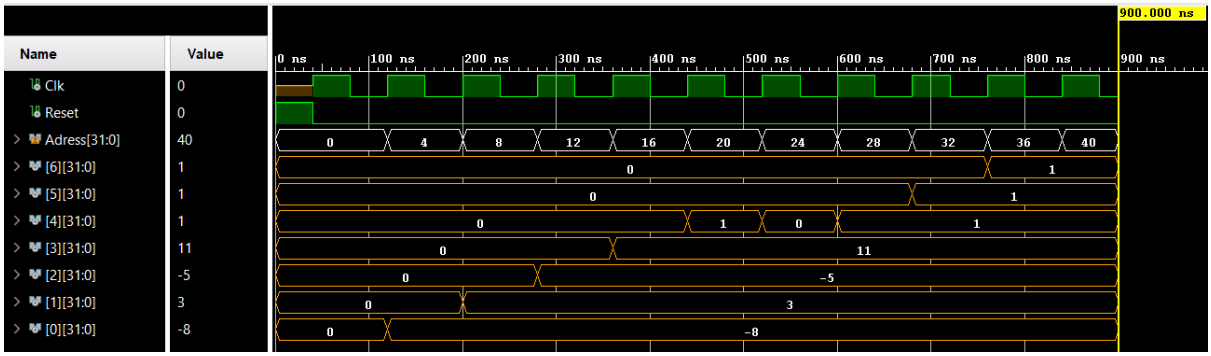

Figure 4: Simulation of provided instructions in *ProgramMemory.vhd* from lab 4 instructions. The white signal is the input address for instruction memory and the orange signals are the stored word in register address 0 to 6.

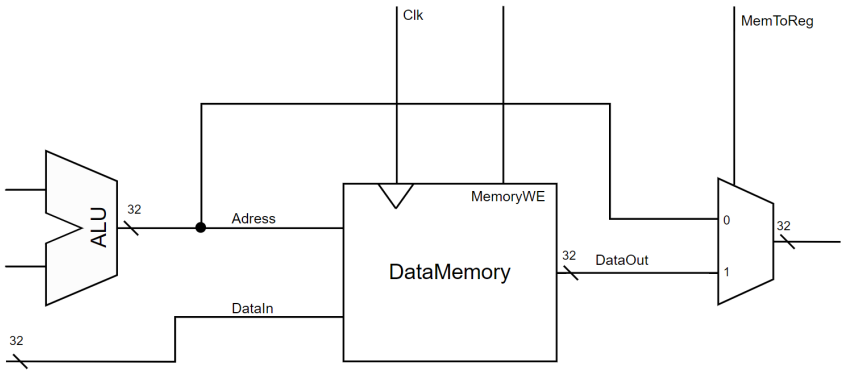

Figure 5: Datapath for data memory component.

A load word instruction consists of a 6-bit op-code, 5-bit register address to which will store the data read from memory, 5-bit memory address to read data from and a 16-bit immediate offset at the end of instruction. The last two parts are summarized and are sent to the address input of the data memory component.

A save word instruction is similar to the load word instruction described previously. The main difference between a load- and store instruction is the second part of instruction, instead of a 5-bit address to load a word from data memory the 5-bit address is used to address a word in the register file.

To support load and store word instructions we need to sign extended the immediate signal to match with the 32-bit input signal to the ALU component. With the signal *MemtoReg* controlling a new multiplexer just after the data memory we can select our *DataOut* as our registry input by setting *MemToReg* to 1. Using *MemtoReg* with *RegDestination* and *WriteEnable* in the register file, we can now read from data memory into the registry file. To store a word into data memory we only need to change our control signals *WriteEnable* and *MemoryWE* to store the constant registry output into the data memory.

To allow for division using our MIPS we must come up with an algorithm using only instructions we have implemented. This was achieved using machine and assembly code with our new supported instructions beq and jump. Beq allows us to branch on condition while jump

lets us move to target instruction. Using jump we could create a loop, and with our previous instructions to add and subtract we could now divide two numbers N and D by subtracting repeatedly.

The division by subtraction algorithm we came up with explained using assembly inspired pseudo code:

```
while:
if D > N branch to done:
Q = Q + 1:
R = R - D:
jump to while:
done:
```

Where Q is the resulting quotient and R is the remainder.

7 Conclusion

Starting out using Vivado was confusing at first when a lot of new concepts are introduced like test bench files and file hierarchy, but after working with Vivado throughout the entire project a new way of thinking develops and the software becomes logical and self-explanatory. Learning VHDL was easier, some new concepts such as behaviour and ports took a bit of learning but in general, a lot of concepts were familiar from previous programming experience. We learned about the entire design process of components and ways to optimize them which is important for the project in the long run. We also got introduced to coding in assembly and although we did not use it much it felt natural after a short while. A lot of useful techniques were learned overtime like karnaugh simplification, implementing and optimizing functions, handling and converting binary both signed and unsigned, creating and reading truth tables are just a few that were used frequently during our labs.

During the process of figuring out our division algorithm, we were met with new challenges like assembly and machine. This part of the project ended up taking the longest time due to mistrust in previous components, which ended up in us wasting time revising. But in turn, we learned a lot from our mistakes, including how important it is having a good design from the start and not taking alluring shortcuts when implementing a component.