# Lab report

Digital Design (EDA322)

*Group: Thu_PM 7*

Erik Thorsell
Robert Gustafsson

March 5, 2015

# Contents

# 1 Introduction

Understanding the complex architecture of a modern central processing unit can seem difficult, at best. This report aims to give a brief introduction to the functionality, development process and finished version of the ChAcc processor. The report covers the development of: the ALU and its parts, the design of the registers and memory components, the bus and the controller unit. The finished product was later tested extensively, programmed onto an FPGA (Field-Programmable Gate Array) and optimized. The last two sections covers this.

# 2 Method

The following sections will thoroughly describe the methodology used when implementing the ChAcc processor.

## 2.1 Arithmetic and Logic Unit (ALU)

The ALU (Arithmetic and Logic Unit) is one of the core components in a CPU (Central Processing Unit). As the name vouches the ALU is in control of the operations between operands. An ordinary ALU does arithmetic as well as logic operations, however the ChAcc (Chalmers Accumulator) processor comes with a slightly reduced set of instructions. Because of this the ChAcc ALU is limited to the operations: addition, subtraction as well as the logical operations nand (not and), not, as well as a comparison operation. One should also note that the operations are only supported for unsigned numbers.

The purpose of the laboration was to implement the ALU, mentioned above, in VHDL. Broken into several stages the first one was to implement an RCA (Ripple Carry Adder), composed of multiple full adders. Briefly, an RCA is a simple adder that can easily be scaled to handle input of various sizes. This is since an RCA is simply a chain of full adders that each takes three bits as input and returns the sum of them as well as a carry out. These inputs are the two bits that are to be added as well as a carry in. In our case the inputs to the ALU are composed of two eight bit, unsigned, numbers leaving us with total of eight full adders in the chain.

Using the data flow design for our full adders the logic was pretty straight forward. The truth table, given in the laboration description, for a full adder speaks for itself and after minimizing the table we moved onto the more interesting part of the laboration, the RCA.

The structural design style of VHDL lets one create instances of already

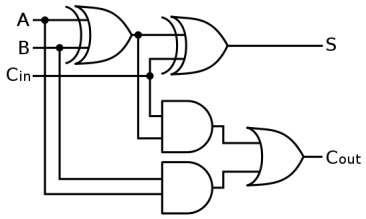


Figure 1: A full adder, described with logical gates. [1]

programmed components and was used to create the RCA. After creating eight instances of the full adder it was simply a matter of passing the right arguments to each of the full adders. The least significant bit of each input gets send to the first full adder, along with the carry in, which returns the least significant bit of the sum as well as a first carry out. The second to least significant bits of each input is then send to the second full adder along with the carry out from the first full adder. This is repeated eight times and the carry out of the eight full adder is the carry out of the RCA. The correctness of the RCA was easily verified by the use of a "do-file".

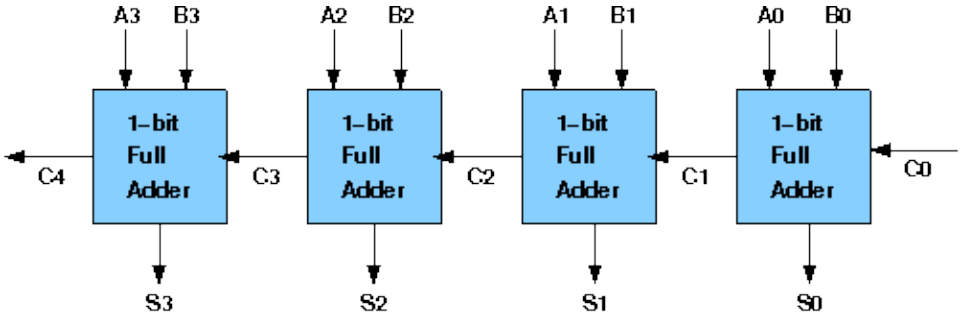


Figure 2: A ripple carry adder, composed out of 4 full adders. [2]

The comparison operation compares two operands and calculates whether or not they are equal. When this is done the corresponding flag (Equal (EQ) or Not Equal (NEQ)) is asserted. Implementing the component was straight forward and since the instructions specifies one must not use the behavioral design style we opted to use the data flow style. A *for .. generate* statement was used to improve readability as well as reduce the amount of code in the file. A bitwise comparison checks if the *n:th* bit of any input differs by the use of an xor gate. For every iteration of the loop, the output of the gate is stored in a temporary signal which is then also used in the loop by the use of an or operation which is done with the new result and the old.

When the loop is done the temporary signal is asserted to the EQ flag, and it is a simpel matter of inverting the signal to get the NEQ flag.

The third task of the laboration was composed of writing the implementation for the subtraction, the not and nand operations, as well as an *isOutZero* signal. Of course one also has to be able to choose between the operations, this ability was implemented using a 4-to-1 multiplexer.

The subtraction operation was simply a matter of performing an addition with one of the operands *two complements representation*. This is achieved by inverting the operand and then add 1 to it. An xor with eight ones with one of the operands as well as adding a carry in to the RCA input solves this. The nand operation is self explained and is achieved by inverting an and with the two operands. The not operation returns the first operand ($ALU\_inA$) inverted. *isOutZero* is done by performing a bitwise or of the output from the multiplexer.

### 2.2 Top-level Design

Implementing the top-level design of the ChAcc processor included the implementation of storage components, such as memory, registers and the bus, as well as initializing the memory components, and connecting all the data path components together.

Firstly we created a generic register with asynchronous reset making use of the behavioral design in VHDL. A simple process, triggered by either the clock or the reset signal, firstly checks if the register is to be reseted or not. The register will be reset if and only if the reset signal is 0. However if the reset signal is 1 the process checks if the clock signal was caused by a rising edge and if that is the case the register will be written to if, and only if, the registers load signal is 1.

```
PROCESS(ARESETN, CLK)
    BEGIN
        IF ARESETN = '0' THEN
            output <= (OTHERS => '0');
        ELSIF rising_edge(CLK) THEN
            IF loadEnable = '1' THEN
                output <= input;
            END IF;
        END IF;
END PROCESS;
```

A register holds only one signal at the time so for larger portions of data the ChAcc has two memories - one for the instructions and one for the data. ChAcc has two memories instead of one since it follows the Harvard

architecture; however making use of generics - as with the register - we only had to implement one, and when implementing the top-level design we instantiated the two memories with generic maps and different init files. The picture below shows read/write operations to a memory.

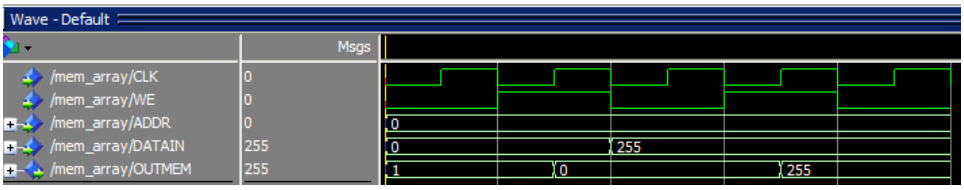


Figure 3: Read and write to a memory.

The processor bus was implemented using a 4 to 1 multiplexer along with some extra logic to incorporate the four control signals into one bit. We minimized the expression with the four signals and opted to default to the EXTDATA signal. Two or gates were needed to get the desired functionality.

## 2.3 Controller

While designing the top level of the ChAcc processor we were provided with a *mock controller*, that acted purely as a place holder. However when that assignment was done we were to implement our very own controller. The controller is used to pass the correct instructions to the correct components of the ChAcc processor at the right time. In order to accomplish this we implemented a Mealy machine, divided into three processes. A Mealy machine is an FSM (Finite-State Machine) whose output values depends both on the current state and the current input of the finite state machine. A complete diagram of the FSM, as well as code examples for the three processes, can be found in the appendix.

The ChAcc processor's specification document provided describes the working of the controller, and the document also specifies which signals are to be set and reset during which operations. This information is crucial in order to implement the controller and was of great use to us.

When the controller was implemented we were provided with a test bench to be run in the simulation software. The test bench tested a couple of operations e.g. adding, subtracting, reading and writing from memory, etc. Unfortunately our processor did not pass the test on our first attempt and we spend many hours trying to figure out why. The problem was finally solved by rewriting our FSM. Our first implementation made use of some logical minimization in order decide what state to enter, however the software seemed not to approve of this and our final implementation uses a simple case statement instead.

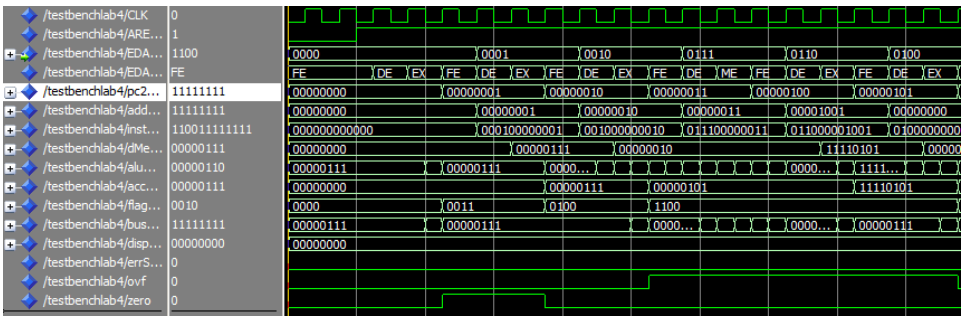


Figure 4: A run of the test bench provided.

## 2.4 Processor's Test bench

In order to test the correctness of our implementation of the ChAcc processor we created a test bench. We were with provided two files to initialize our memories with, as well as five files to compare the output of our ChAcc processor with.

The test bench is divided into two parts. A reading part and a comparison part. For each "comparison file" we created one reading process and one comparison process. The reading process simply reads each line of the provided file and transforms the given row into a standard logic vector which can be compared to the output of the ChAcc processor.

```
readAcc: PROCESS
    VARIABLE accLine : LINE;
    VARIABLE accData : BIT_VECTOR(7 DOWNTO 0);
BEGIN
    FOR i IN 1 TO 30 LOOP
        WAIT UNTIL (ARESETN = '1' AND rising_edge(CLK);
            READLINE(accFile, AccLine);
            READ(accLine, accData);
            WAIT UNTIL (acc2seg'ACTIVE);
                accSignal <= TO_STDLOGICVECTOR(accData);
    END LOOP;
    WAIT;
END PROCESS;
```

The comparison process compares the output from the read file with the signal from the ChAcc processor. If they differ an error has occurred and we report this to the user, the test bench also terminates due to the severity of the error.

In order to terminate the test bench after a successful run, the test bench is programmed to end when the display register outputs the decimal number 144, a simple if statement checks the disp2seg signal and if the signal equals 144 the test is declared successful.

The files with data differs in length so if no errors occur throughout the comparison process for one process we simply wait for the other processes to continue and for disp2seg to reach decimal 144.

```
verifyAcc : PROCESS(CLK)
BEGIN
    IF (ARESETN = '1' AND falling_edge(CLK)) THEN
        IF (acc2seg /= accSignal) THEN
            REPORT "Acc ERROR"
            SEVERITY ERROR;
        END IF;
    END IF;
END PROCESS
```

### 2.5 ChAcc on Nexys 3 board

After completion of the test bench we synthesized our implementation of the ChAcc processor with the intention to program it onto an FPGA. During synthesization the software used (XILINX ISE Design Suite) reported a couple of warnings, mostly regarding problems with latches in our code. This was fixed by making sure all possible states were covered for all signals.

When the synthesization completed without warnings we programmed our bit-file onto the FPGA and started to make our way through the switches and jumpers on the FPGA. Unfortunately we got very inconsistent behavior from the board and this was finally resolved by switching the board.

With a fresh board and a minor adjustment to the code everything worked as expected. We now perform a logical and with every control signal and the master_load_enable signal in order to sync the control signals with the signal. The Fibonacci sequence appeared in hex-decimal on the two right-most seven-segment displays and continued incrementing until it reached E9 (decimal 233). The next value of the Fibonacci sequence - decimal 377 - is larger than what we can represent with our 8-bit values.

The FPGA can show different values on the four seven-segment displays. For the two left displays these are: the lower eight bits of the instruction, the output from the data memory, the contents of the address register, and

the content of the accumulator. The two right displays shows either the flags and opcode, the program counter, the bus data, or the contents of the display register. The display register is the output wanted when looking for the Fibonnaci sequence. However, the program counter is also quite interesting since the program is written as a loop. When stepping through the program the program counter increments from 0 to 6 and then goes back to 0.

## 2.6 Performance, Area and Power Analysis

Optimizing our implementation of the ChAcc processor was done using the same software as was used in the sixth task. Briefly, what was done, was a three step optimization each focusing on one specific property of the implementation. When estimating the over all performance of the processor a formula was used provided by the examinator.

$$\text{Final Efficiency} = \frac{\text{Clock Frequency(MHz)}}{\text{Power(Watt)*Area(Slices)}}$$

**Performance**
Firstly we optimized for performance. The only property of interest to us was to increase the frequency of the processor's clock as much as possible. The following settings were used that differs from the default settings under the Synthesize - XST Process Properties:

- Optimization Goal - Speed
- Optimization Effort - High
- Generate RTL Schematic - No
- Case - Upper
- FSM Encoding Algorithm - Speed1

Starting with the suggested "7ns" we lowered the constraint value until we eventually got errors telling us that our processor did not meet the given timing constraints. The final constraint we entered was 4.95ns which yielded a frequency of 202 MHz.

**Area**
When optimizing for area we wanted our processor to use as few slices as possible. The performance optimization used 67 slices but changing the processor properties to optimize for area made the processor use only 54 slices instead. The following settings were used that differs from the default settings under the Synthesize - XST Process Properties:

- Optimization Goal - Area
- Optimization Effort - High
- Generate RTL Schematic - No
- Case - Upper
- FSM Encoding Algorithm - Compact

By doing this the power consumption dropped from 0.249 W to 0.118 W. However, the clock speed was significantly slower: 105 MHz.

**Power**
Aiming to optimize for power efficiency we went through all the settings we had previously gone through and ticked every box with "power efficiency" next to it. We also had in mind that a fast processor would be a processor in need of much power why we opted to choose "slower" settings as well. The following settings were used that differs from the default settings under the Synthesize - XST Process Properties:

- Optimization Goal - Area
- Optimization Effort - High
- Power Reduction
- Generate RTL Schematic - No
- Case - Upper

When done we got a power consumption of 0.117 W, 0.001 W less than for the area optimized implementation. However, we did get a clock frequency of 118 MHz and this implementation used only 52 slices. In other words, this was by far our best implementation yet in terms of over all efficiency.

**Optimized**
With our acquired knowledge we tried to optimize our processor to reach as high of an overall score as possible. Since our power optimization had yielded the best overall score yet we used it as a base and a lab instructor showed us some additional settings that could be used. These were the setting used that differs from the settings used for power optimization from the Implement Design Process Properties:

- Placer Extra Effort - Normal
- Power Reduction - Extra Effort
- Extra Effort (Highest PAR level only) - Normal

- Power Reduction

When these settings had been changed we tried to increase the clock frequency while managing to keep our area small and our power consumption low. By lowering the value of the timing constraint we did eventually reach a clock frequency of 126 MHz and a total score of 19.6 which we found satisfying and concluded our quest to be over. Worth noticing is that the area decreased even more and our final implementation used only 51 slices.

| Metric | | | | |
|---|---|---|---|---|
| Design | Perf. (MHz) | Area (Slices) | Power (W) | Overall |
| Perform.-opt. | 202 | 67 | 0.249 | 12.1 |
| Area-opt. | 105 | 54 | 0.118 | 16.6 |
| Power-opt. | 118 | 52 | 0.117 | 19.4 |
| Efficiency-opt. | 126 | 51 | 0.126 | 19.7 |

# 3 Analysis

With all labs completed it is - first of all - very interesting to see what we have created. At first it seemed like a far fetched goal to implement our very own control processing unit, but after five weeks we had a fully working unit programmed onto an actual silicon chip and after one more week we have even been able to optimize it a bit. Arithmetic and logical operations works flawlessly, we have implemented working memory components such as registers and memory arrays, a bus has been implemented to send data around the data path and everything is controlled by a controller. Last but not least, everything has been tested thoroughly and been proved to work!

During the creation of our ChAcc processor many questions arose, some which was eventually answered and some that until this day remains unanswered. For instance there are many VHDL specific questions regarding the syntax which seems unintuitive and right out weird, but those are probably better to ventilate in another forum.

The most difficult laboration was by far the forth one. As has already been mentioned we had major difficulties completing the forth lab and we spent many hours trying to solve the problem with our FSM. The problem remains unresolved until this day and concerned how we choose state depending on the operational code. We had minimized the operational code with Karnaugh diagrams but this implementation did not work, at all. During a compare instruction the FSM entered a state that it was not supposed to be able to enter. Several lab assistants, and even the examinator himself, spend additional time helping us but we all reached the same conclusion: our code should have worked.

However, this was eventually solved and the fifth, sixth and seventh lab was completed without further - weird - issues. There were, of course, problems that appeared during the labs but none that was not resolved by logical reasoning.

Finally, the labs have taught us a lot and even though we have barely scratched the surface of VHDL we have gained a good foundation to keep building from.

# A Appendix

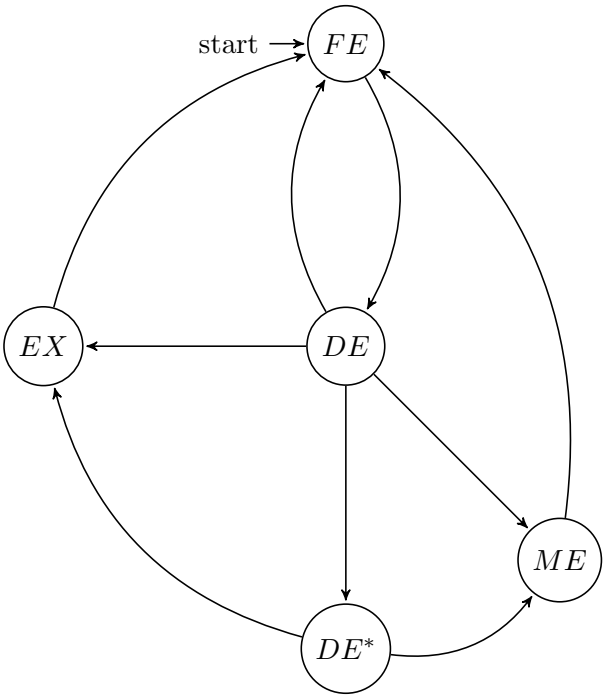


Figure 5: Mealy Finite-State Machine.

```
PROCESS(CLK, ARESETN)
BEGIN
IF ARESETN = '0' THEN
    current_state <= FE;
ELSE
    IF (rising_edge(CLK) THEN
        IF master_load_enable = '1' THEN
            current_state <= next_state;
        END IF;
    END IF;
END IF;
END PROCESS
```

*The first third of the FSM.*

| Mealy FSM | | |
|---|---|---|
| State | Opcode | Signals |
| FE | 000X, 0101, 0110, 100X, 1111 | instrLd |
| | 0010 | instrLd, aluMd(0) |
| | 0011 | instrLd, aluMd(1) |
| | 0100 | instrLd, aluMd(1), aluMd(0) |
| | 0111, 1010 | instrLd, acc2bus |
| | 1011 | instrLd, ext2bus |
| | 1100 - 1110 | instrLd, im2bus |
| DE | 000X, 0101, 0110, 1001 | instrLd, dataLd |
| | 0010 | instrLd, dataLd, aluMd(0) |
| | 0011 | instrLd, dataLd, aluMd(1) |
| | 0100 | instrLd, dataLd, aluMd(1), aluMd(0) |
| | 0111 | instrLd, acc2bus |
| | 1010 | instrLd, dataLd, acc2bus |
| | 1011 | pcLd, instrLd, dmWr, ext2bus |
| | 110X, 1110 | pcSel*, pcLd, instrLd, im2bus |
| | 1111 | instrLd |
| DE* | 100X | instrLd, addrMd, dataLd |
| | 1010 | instrLd, acc2bus |
| EX | 000X | pcLd, instrLd, flagLd, accLd, dmRd |
| | 0010 | pcLd, instrLd, flagLd, accLd, dmRd, aluMd(0) |
| | 0011 | pcLd, instrLd, flagLd, accLd, dmRd, aluMd(1) |
| | 0100 | pcLd, instrLd, flagLd, accLd, aluMd(1), aluMd(0) |
| | 0101 | pcLd, instrLd, flagLd, dmRd |
| | 0110 | pcLd, instrLd, accSel, accLd, dmRd |
| | 1000 | pcLd, flagLd, accLd, dmRd |
| | 1001 | pcLd, flagLd, accSel, accLd, dmRd |
| | 1111 | pcLd, instrLd, dispLd |
| ME | 0111 | pcLd, instrLd, dmWr, acc2bus |
| | 1010 | pcLd, instrLd, addrMd, dmWr, acc2bus |

*The complete table of states and corresponding signals.*

```
PROCESS(current_state, opcode)
BEGIN
CASE current_state IS
    WHEN FE =>
        next_state <= DE;
    WHEN DE =>
        CASE opcode IS
            WHEN "0000" => next_state <= EX;
            WHEN "...." => next_state <= ..;
        END CASE;
    WHEN DES =>
        CASE opcode IS
            ...
        END CASE;
    WHEN ...
END CASE;
END PROCESS
```

*The second part of the FSM.*

```
PROCESS(current_state, opcode)
BEGIN
CASE opcode IS
    WHEN "0000" =>
        CASE current_state IS
            WHEN FE =>
              v_control <= (2 => '1', others => '0');
            WHEN DE =>
              v_control <= (2 | 5 => '1',
                            others => '0');
            WHEN EX =>
              v_control <= (1 | 2 | 6 | 8 | 10 => '1',
                            others => '0');
        END CASE;
    WHEN .... =>
END CASE;
END PROCESS
```

*The third part of the FSM.*

## References

[1] Saif Imtiaz Circuit,
*Full Adder Circuit.*
http://implement-logic.blogspot.se/2011/08/full-adder-circuit_16.html,
2011,
Visited: 2015-03-05.

[2] Virtual Labs,
*Design of Ripple Carry Adders.*
http://deploy.virtual-labs.ac.in/labs/cse10/rca_design.html,
2010,
Visited: 2015-03-05.