Lab report

Digital Design (EDA322)

Group TUE_PM_8

Niklas Gustafsson

Oskar Lundström

March 1, 2017

Contents

1 Introduction

The purpose of this report is to describe the steps of building the ChAcc processor. Each section is intended to describe the most important parts of the work as well as the different challenges we faced.

In lab 2 we created an RCA and a CLA from full-adders. We then used this in combination with extra logic to create an ALU that can perform some basic operations such as addition, bitwise AND-ing and equality-testing.

In lab 3 we implemented a register and a memory with generic sizes. We also designed a bus using multiplexers instead of tri-state buffers. These components were then combined to create the datapath of the processor.

In lab 4 we engineered the controller of the processor. We started out with a state diagram describing our controller as a Mealy FSM and then realized it in VHDL with three processes. The datapath now had a functioning controller and we tested it with a provided testbench.

In lab 5 we designed a testbench for the processor. We were provided with a program and the expected outputs of some control signals. The testbench loads the program, executes it and compares the output with the expected values.

In lab 6 we synthesized our design of the processor. We encountered warnings from the synthesis tool which we had previously not seen. We corrected these errors and generated a bitsream which we loaded to the FPGA. We also made sure the program worked as expected.

In lab 7 we optimized our design after performance, area, power and efficiency and compared tradeoffs. We did this by using different optimization strategies.

Our main conclusions after completing all of the labs are that:

- You need to follow the specifications for control signals exactly, otherwise the instructions will either do nothing or do something completely unintended.
- Unlike when programming, there are more non-obvious things to take care of when describing hardware.
- When optimizing for performance the cost of increasing the clock frequency grows exponentially. Increasing the frequency by a few MHz is very costly in terms of power and area used.

2 Method

2.1 Arithmetic and Logic Unit (ALU)

When we built our ALU we began by constructing the most basic component, the full-adder. We started by minimizing the boolean expression for the sum and the carry out signals using a Karnaugh diagram. We came to the conclusion that a full-adder should be built in the way described in the picture below.

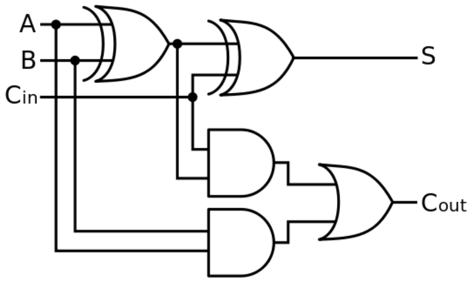

Figure 1: A full-adder.

You can also build a full-adder using two half-adders and some extra logic. We came to this conclusion by connecting the resulting signal of the first half-adder to the second one along with the carry in signal. After this we added the extra logic that was needed, which was an OR gate.

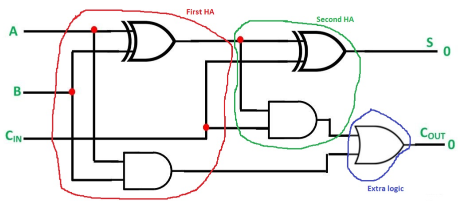

Figure 2: A full-adder made from two half-adders.

We created an 8-bit ripple carry adder (RCA) by connecting eight full-adders together according to the diagram below. The idea is to simply connect the carry out signal of the first full-adder to the carry in signal of the second one, and so on. The RCA was used in our original design of the ALU but later we replaced it with a carry lookahead adder (CLA).

The ALU includes a comparator unit which is used to check whether the operands are equal or not. You can implement it by using the structural or dataflow style of VHDL. We decided to do the dataflow implementation

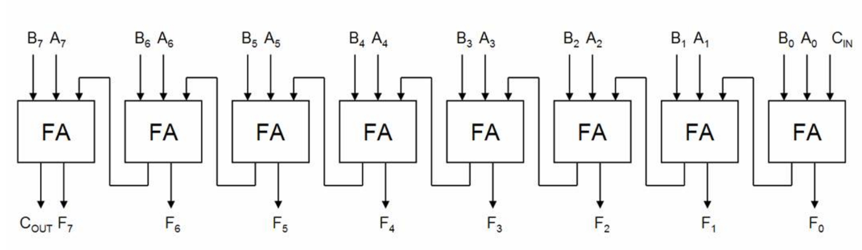

Figure 3: A ripple carry adder.

because we thought it would be easier and that it would result in a cleaner code. In order to check whether the operands are not equal we performed a bitwise XOR on the operands and OR:ed the results. This was used as the resulting NEQ signal. The EQ signal was simply an inversion of the NEQ signal.

If we were to do this in structural VHDL we would have to create a component which cheked if two certain bits were not equal. We would then have to create eight such components. The input as a whole is equal if all 8 bits are equal, otherwise not.

The operation A minus B is performed by converting B to its 2-complement and then adding it with A. By passing B through 8 XOR gates, each having SUB as its second input, we can form B:s 1-complement when subtracting. We connect SUB to carry-in to form the 2-complement.

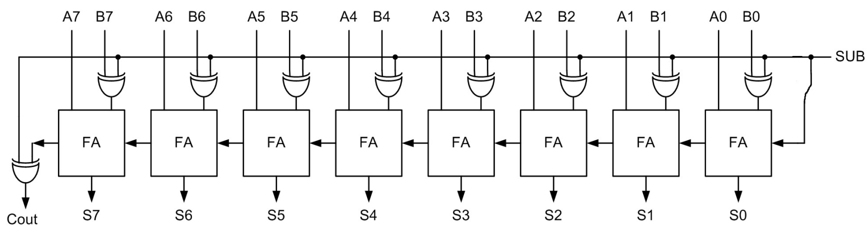

Figure 4: Logic for performing subtraction with an adder.

One thing that we learned was that in VHDL the entity names in a project have to be unique for all references to work properly.

We also learnt why the critical path for an n-bit RCA is $2n + 1$ gates. For every full adder the carry-out depth depends on the inputs (3 gates deep) and the carry-in (2+previous depth). The depth is the maximum of these. For the first the depth becomes 3, for the second 5 and so on. In other words, for the first adder the inputs are the fartherest away, but for the following it is the carry-in instead.

2.2 Top-level Design

Registers
We made a generic implementation of a register by letting a variable hold an integer representing the number of bits the register is able to hold. The vectors for in- and output to the register are adjusted after this variable. We implemented our register using behavioral VHDL, i.e using a process statement, since that seemed to result in the simplest and cleanest code. Another reason for using a process statement is that we know that assigning signals values in a process statement result in flip-flops when the code is synthesized.

Memory
We also made our implementation of the memory generic. We have two integer variables, one describing the number of address bits (*addr_width*) and one describing the number of bits each memory location can hold (*data_width*). The number of bits in the input signals for address and data are adjusted after these variables. The memory itself is an array of a type which we specified. The number of entries in this type of array is 2^{addr_width} and the size of each entry is *data_width*. The read operation is asynchronous and the write operation is synchronous.

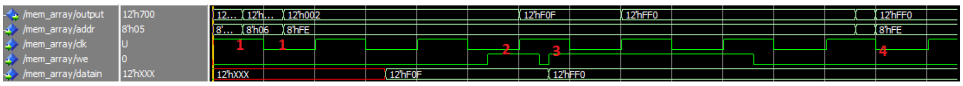

Figure 5: The operation of the memory.

In figure 5 interesting things happen at the red numbers.

1. The read is asynchronous, the output is changed directly after the address is changed.
2. The write is synchronous, the write does not happen until the positive edge.
3. When the input is changed, no write happens at the negative edge.
4. If we changed the address to something else, and then change back, the value previously written is still there.

Bus
The bus was implemented with a mux. The 4 indata signals each had 8 bits. Bit 0 from each goes to a 4 to 1 mux, bit 1 from each to another mux and so on, i.e. the signals were muxed bitwise. The muxes are inferred by

the tools since we used a *WITH SELECT* statement. The mux select signal was created from a one-hot representation of the 4 control signals. Based on which bit was set, the corresponding mux select signal was created. If more than 1 bit are set, the bus error signal is set.

We used a multiplexer for the bus since if 2 or more control signals are set, the bus will not take an undefined value, as opposed to a tri-state version. If 0 or more than 1 control signals are set, we put 0's on the bus since we will not read from it anyway in that case.

Datapath
The last step was to connect all components of the datapath as described by figure 6. In the datapath entity we had signals with the same names as on that figure (made it easier to code if the names were the same). We then instatiated all components and connected them according to the figure. Some components, such as the registers, have different datawidths in different places. Since they were generic it was easy to have multiple versions.

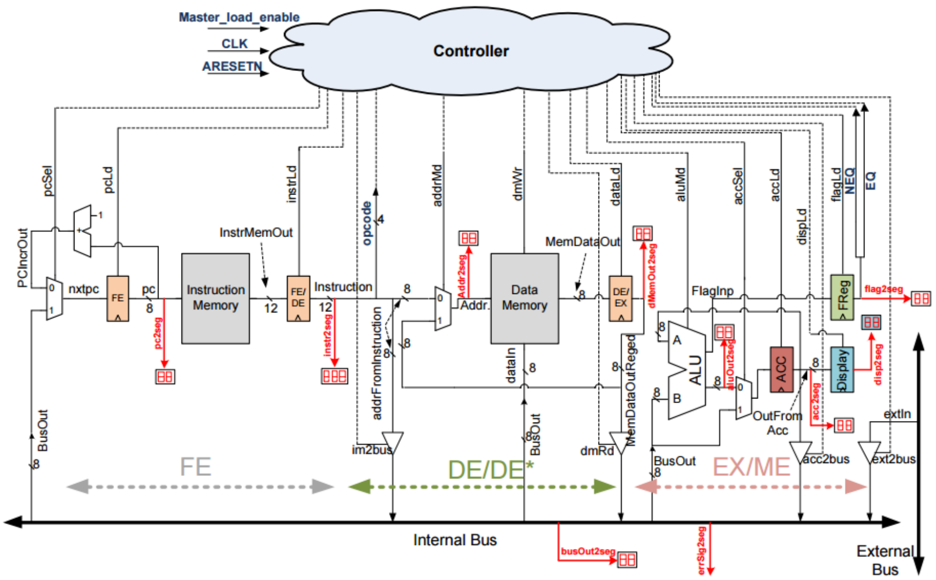

Figure 6: The datapath of the processor.

2.3 Controller

The FSM

We chose to design our controller as a Mealy FSM. We did this in order to reduce the amount of states (fewer flip-flops). If we would've chosen a Moore design we would roughly need one state for each combination of opcode and stage, although we would be able to minimize it to some extent. Fewer states means that we only have to keep track of five states, namely the stages (FE,DE,DE*,EX,ME), and hence the resulting VHDL code as well as the circuit itself becomes simpler. The statements for assigning the output signals a value are rather long. This would however be the case no matter which design we chose. If we had chosen a Moore design each row in these statements would represent a state and hence the code would be equally long.

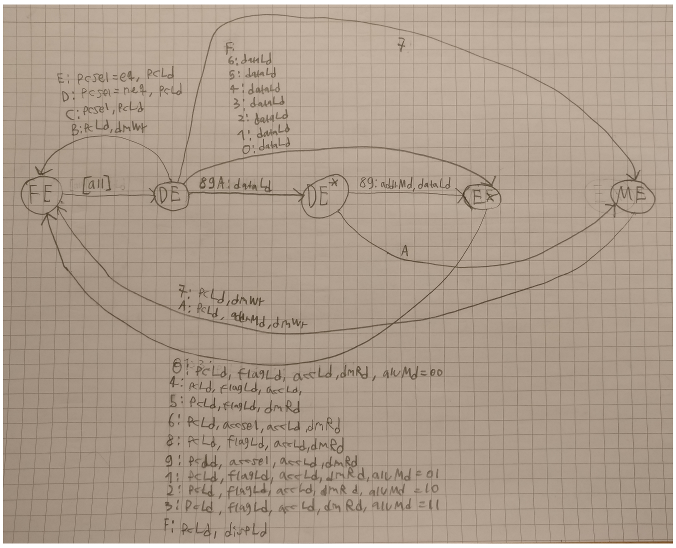

Figure 7: The state diagram.

Testing

When we first ran the testbench we noticed a problem. The contents of the data memory did not match the specifications of what it should contain after running the testbench. We started troubleshooting and found three problems.

1. We found that our EQ and NEQ signals to the controller weren't assigned at all. We fixed this but the problem remained.

2. We continued to troubleshoot the controller by examining the waveform diagram of the controller and datapath when running the test-

bench. While doing this we noticed that after about 500 ns the opcodes didn't match the instructions that should be executed. It turned out that the *dmRd* signal didn't have the correct value when the AND instruction was executed. In the processors specification document this signal was a "don't care" for this opcode when it really should've been set to 1, so we changed it. A few days after writing the code we checked the processors specification document and noticed that this had been changed there as well. We assume that this simply was a mistake.

3. The instruction for the jump statement was placed at memory location 256 (index starts from 0). This meant that when we jumped to memory location 255 a *NOOP* instruction was executed, the PC overflowed and then the program started from the beginning. We corrected this by moving the contents of memory location 256 to memory location 255. After this the testbench ran successfully.

In figure 8 we see the waveform diagram of instructions 1 to 3 in the test program being executed. If you compare the diagram with the processor's specification document it's clear that the signals have the correct value at the correct time. You can also see that it's first after the fetch stage is completed that the opcode changes. This is because the fetch stage has to be completed before the opcode can be sent from the FE/DE register to the controller.

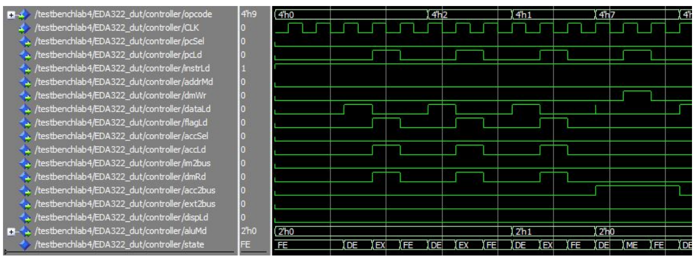

Figure 8: The waveform diagram for executing instructions 1 to 3 in the test program. We selected these instructions since they use most of the datapath and control signals.

Note that when the opcode changes from 1 to 7 in figure 8 a spike occurs for the signal dataLd. This is an effect of our Mealy design. The output signal only depends on state and input (opcode). Since the stage is DE and the opcode is 1 for a brief moment the *dataLd* signal becomes 1. When the opcode changes to 7 *dataLd* becomes 0. This is not a problem since the spike occurs after the positive edge of the clock. Since everything is synchronous the control signals have corrected themselves before the next positive edge, meaning that no errors will be made.

2.4 Processor's Testbench

The testbench is used to verify the processor's correct operation. By simulating inputs to the processor and examining the outputs you can determine whether an operation was carried out in a correct fashion or not. In practice this is done by writing an assembler program which you load into the memory of the processor. The verification is done by comparing the actual outputs with the expected outputs while running the program on the processor. If any of the outputs take an unexpected value an error has been found.

When designing the testbench we initially thought that there were huge issues with the flag register. We got no output to the *flag2seg* signal which indicated that something was wrong with the flag register. This seemed a little bit odd however, since the flag register had worked just fine previously. When checking the VHDL code for the processor it turned out that we'd forgotten to connect the *flag2seg* signal to the flag register. After connecting this our testbench ran successfully.

How the testbench works

- Generating input. The input to the processor are the control signlas (*externalIn, clk, master_load_enable* and *aresetn*) and the instruction and data memory files. The control signals are assigned values, some periodically, in the architecture of the testbench. The memory content is read when the memory entities are instantiated. Based on the memory content different output is expected, hence we consider those files an input from the testbench.
- Reading expected outputs. The expected output values are read from text files into five different arrays, one for each output signal to check. The reading is done inside a process in the testbench architecture. For each file, while it has not reached the end, we read the current line. That line is parsed as an 8-bit vector and is stored in the array at the current index. The index is incremented and the procedure is repeated.
- Comparing outputs and expected outputs. The output signals of interest have one process each, sensitive to the signal. When that signal changes the process is triggered and its value is compared to its expected value in the array by using asserts. If they are equal, the index for that array is incremented. Otherwise the testbench stops and reports the error. The output signals that are checked are *DataMemOut, flag2seg, pc2seg, acc2seg* and *disp2seg*.

2.5 ChAcc on Nexys 3 board

Synthesis vs simulation
We encountered three problems in particular:

- In the process that assigned the next state we did not have *else* clauses and *when others* statements. We thought we didn't need this since we had exhautsed all possibilites. When synthesizing, latches were created because we did not have these statements and hence we added them.
- When synthesizing we got warnings saying that we didn't have all signals used in the process in the sensitivity list. This warning did not occur when simulating so it did not affect the correctness during simulation. This will not change the behaviour of the program logically but we added it to avoid glitches.
- When we ran the program on the FPGA we observed the PC values and opcode values. They did not match the program. It appeared that the values jumped more than one step each cycle. The lab assistant suggested that we should make sure that the load control signals were set only if *master_load_enable* was set. This fixed the problem. If, for instance, *pcLd* does not depend on *master_load_enable* PC will be incremented at the pace of the main clock. But the stage would not change if *master_load_enable* was 0. This is of course unwanted beaviour. If the stage would change independently of *master_load_enable* the hardware would function correctly, but we would not be able to toggle the progress manually and observe changes ourselves.

When we simulated our processor all of our tests were completed successfully. We didn't get any indications that we should make any of the changes mentioned in bullets 1 and 2. It was when we synthesized our design that we got warnings about the sensitivity list and not exhausting all possibilities. Since it's hardware we're dealing with we have to make sure that the VHDL we write is possible to synthesize into well functioning hardware. If the synthesis tool ask us to make certain changes it's a good idea to follow its advice.

Verifying the correctness
We began the verifictation by observing the PC value. Since it should go from 0 to 6 it should be easy to detect an error and follow the progress. When observing it we saw that it jumped between 0 and 6 directly, and sometimes to 1 and 2, indicating that something was wrong. This lead us to the conclusions above. After fixing it, we observed PC again and this time it went from 0 all the way to 6 and then started over again from 0, as we

expected. Next we observed the opcode. The opcode was also easy to follow since we could compare it to the *inst_mem.mif* file. The opcode sequence was correct as well. Finally we observed the *Display* sequence. It correctly showed the Fibonacci sequence. Table 1 shows the sequence for all three seven-segment displays. This was the same sequence we saw in Modelsim.

pc	0, 1, 2, 3, 4, 5, 6, 0,...
opcode	6, 2, F, 7, 1 , 7, C, 6,...
display	0, 2, 3, 5, 8, D, 15, 22, 37, 59, 90

Table 1: These are the sequences of the seven segment displays on the FPGA. The values are hexadecimal.

2.6 Performance, Area and Power Analysis

We started out by creating a Xilinx project and added all the necessary VHDL files. We specified certain deisgn goals for each metric that we wanted to optimize our design for. Although we created custom strategies for each version they all had the process properties in table 3 and the results in table 7 in the appendix in common. We evaluated the designs by checking the clock period in the static timing report, the number of slices used in the map report and the power usage by using the XPower Analyzer tool. Our general strategy was to iteratively change the process properties, evaluate the specific metric, and check for improvements.

For each optimization we did there's a table in the appendix showing the data below.

- Process properties.
- Results.
- Modules ranked by power dissipation.

Performance-optimized
When optimizing for performance our goal was to get as high clock frequency as possible. Our general strategy when trying to accomplish this was to allow the tool to use all the power and area that it needed. We also used a one-hot representation in our FSM encoding to get as simple logic as possible and hence a shorter critical path. See table 4 for process properties and table 8 for results, both in the appendix.

To achieve these results we used the timing constraint 5.3 ns. We began with 7 ns as the timing constraint and gradually lowered it. Even if we violated it, we continued to lower and obtained good results. Our current timing constraint is violated but it still produces the best result.

Area-optimized
When optimizing for area our goal was to use as few slices as possible. We accomplished this by having loose timing constraints and settings that would result in few gates. See table 5 for process properties and table 9 for results, both in the appendix.

Power-optimized
When optimizing for power we generally used the same settings as for area. We thought that a smaller circuit would require less power (fewer LUTs to power, and shorter wires which reduces capacitance). See table 6 for process properties and table 10 for results, both in the appendix.

From the results table we can see that the power-optimized version uses fewer slices than the area-optimized version. We believe the reason we got a smaller area when we optimized for power is that the tool realises it can save power by reducing area in some parts of the design.

Efficiency-optimized

To create a design which was more efficient overall[1], we lowered the timing constraint for the power optimized design from 10 ns (as it had been in both the area and power optimized versions) to 9 ns. This improved all three metrics. However, lowering it to 8.5 ns made all three metrics worse so we stopped there. We did not change any process properties since we believed the current settings were good for both power and area. Any change to improve performance would be very slight, and worsen the two other metrics significantly. We obtained the results showed in table 11 in the appendix.

It is interesting to note that the efficiency-optimized version is better than, or equally good, as the power-optimized in all three metrics. The performance improvement was expected since we set a tighter constraint. The total on-chip power is only 0.01 mW lower which is within the margin of error. However, the dynamic power is actually 0.001 W higher, so it appears it consumes more power after all.

Overview of the results

We did four versions, each optimized for a certain metric. We set properties we thought would improve the version in the current metric. However, in the end the efficiency-optimized is the best in the metrics area, power and efficiency. The performance-optimized was best in performance though. All details are in the table below.

Metric	**Performace**	**Area**	**Power**	**Overall efficiency**
Performance	188.218	275	0.00481	142.90
Area	119.290	189	0.00189	333.95
Power	118.119	179	0.00154	428.50
Efficiency	124.950	179	0.00153	456.24

Table 2: Overview of the result in the four metrics for the four designs. Each row is a design optimized for a certain metric. The units are MHz, number of slices and watts.

[1] According to the formula given in the lab PM.

3 Analysis

After completing lab 2 to 5 [2] we have a processor which can perform some basic operations. The processor consists of an ALU, registers, memories, muxes, a bus and a controller. Through careful testing and debugging we have verified that each component works as intended, both individually and together. We used waveforms to test the components and a testbench we designed ourselves to test the processor as a whole.

We've made two projects for each of our labs, one using an ALU based on an RCA and one based on a CLA. Although we haven't been able to notice any difference in performance (which we didn't expect to be able to) it's been interesting to see both implementations and how they accomplish the same task. Since the critical path of the CLA is shorter the clock frequency can be increased and the performance enhanced.

When testing our controller in lab 4 we had some issues. In order to solve this we needed to use waveforms to check what instructions were being executed as well as if the correct state and control signals were set. By examining the waveforms we were able to track the error to a certain control signal which wasn't set when it should've been. This gave us valuable experience in how to use simulations and waveforms to debug errors.

It was a good learning experience to design the controller as an FSM. Previously we had only designed smaller circuits for the purpose of learning how to do it, and not for use in practice. Designing an FSM for use in practice gave use a better understanding of the characteristics of a Mealy FSM compared to a Moore FSM (number of states and timing).

The hardware and the testbench have been described in VHDL. We have gained experience in this and also seen the difference between programming and describing hardware in code. Throughout this lab series we have kept in mind to write the VHDL in such a way that no unwnated hardware (e.g. latches) is synthesized. This was new to us compared to programming, where you can write more freely.

We realize that if the goal was to design an ASIC much more work would be needed. For example we would need to describe the placement of the components on the chip. And if we were to make some errors (which we have done) it would cost both money and time.

[2]The analysis doesn't include lab 1, 6 and 7 acoording to the writing guidelines.

A Appendix

A.1 Process properties for the three different versions

Process property	Setting	Comment
Optimization effort	High	We wanted the tool to optimize our design as much as possible.
Keep hierarchy	False	Allows the tool to flatten the design in order to optimize it.
Allow logic optimization across hierarchy	True	We want to allow the tool to optimize. our design the way it wants to.

Table 3: Design goals common for all versions.

Process property	Setting	Comment
Optimization goal	Speed	Self-explanatory.
Power reduction	Method 1	We sacrifice power for performance.
FSM encoding	One-hot	Simpler logic means a shorter critical path. We sacrifice area for speed by using more flip-flops. For one-hot the state is already decoded from the flip-flops so no extra logic for that.
FSM style	LUT	We tried to set this both to LUT and BRAM but the result was the same.
LUT combining	Off	We don't want the tool to combine LUTs in order to save area since this could reduce the speed.
Combinatorial logic optimization	Off	We used the default value, in hindsight we probably should've tried changing it.
Global optimization	Speed	Self-explanatory.
Equivalent register removal	Off	If there are registers used for the same task it. might take more space but that's ok if the result is higher speed.

Table 4: Design goals for the performance-optimized version.

Process property	**Setting**	**Comment**
Optimization goal	Area	Self-explanatory.
Power reduction	Method 1	We let the chip consume more power if it can reduce the area.
FSM encoding	Gray	Gray encoding usually has simpler next state logic than binary, which reduces area.
FSM style	LUT	We tried to set this both to LUT and BRAM but the result was the same.
LUT combining	Area	Less area needed if LUTs are combined.
Combinatorial logic optimization	On	Used to reduce the area used by combinatorial logic.
Global optimization	Area	Self-explanatory.
Equivalent register removal	On	Remove registers with the same task.

Table 5: Design goals for the area-optimized version.

Process property	**Setting**	**Comment**
Optimization goal	Area	We could only pick between speed and area. Speed definitly consumes more power while area probably reduces power. since the circuit will be smaller.
Power reduction	Method 2	Method 2 reduces power at the cost of speed and area.
FSM encoding	Gray	If the states go in sequence only one bit will flip each time for gray-encoding. This is what matters since it is the bit-flips that consume the most power.
FSM style	LUT	We tried to set this both to LUT and BRAM but the result was the same.
LUT combining	Area	Less area (and hopefully power) needed if LUTs are combined.
Combinatorial logic optimization	On	Used to reduce the area (and hopefully power) used by combinatorial logic.
Global optimization	Power	Self-explanatory.
Equivalent register removal	On	Remove registers with the same task.
Power reduction (Map)	Extra Effort	Self-explanatory.

Table 6: Design goals for the power-optimized version.

A.2 Results

Most power hungry module	Processor entity
Most area-costly module	Processor entity
Leakage power	0.022 W

Table 7: Results which were the same in all three cases.

Metric	Result
Performance	Clockperiod: 5.313 ns, critical path: FE/DE register to FReg.
Area	275 slices, 63 slice registers and 212 LUT slices.
Power	Total: 0.00481 W, dynamic power: 0.216 W.

Table 8: Results obtained when optimizing for performance.

Metric	Result
Performance	Clockperiod: 8.383 ns, critical path: FE/DE register to DE/EX register.
Area	189 slices, 39 slice registers and 150 LUT slices.
Power	Total: 0.00189 W, dynamic power: 0.160 W.

Table 9: Results obtained when optimizing for area.

Metric	Result
Performance	Clockperiod: 8.466 ns, critical path: FE/DE register to ACC register.
Area	179 slices, 39 slice registers and 140 LUT slices.
Power	Total: 0.00154 W, dynamic power: 0.159 W.

Table 10: Results obtained when optimizing for power.

Metric	Result
Performance	Clockperiod: 8.003 ns, critical path: FE/DE register to ACC register.
Area	179 slices, 39 slice registers and 140 LUT slices.
Power	Total: 0.00153 W, dynamic power: 0.160 W.

Table 11: Results obtained when optimizing for efficiency.

A.3 Modules ranked by power dissipation

Module	Power dissipation
ALU	0.000011 W
FE/DE register	0.00011 W
Controller	0.0001 W
Bus	0.00005 W
Data memory	0.00003 W
Instruction memory	0.00001 W

Table 12: The processor's modules ranked by power dissipation when optimizing for performance. Does not include modules with 0 W power dissipation.

Module	**Power dissipation**
ALU	0.00015 W
Controller	0.00013 W
Data memory	0.00002 W
Instruction memory	0.00001 W
ACC register	0.00001 W

Table 13: The processor's modules ranked by power dissipation when optimizing for area. Does not include modules with 0 W power dissipation.

Module	**Power dissipation**
Controller	0.00008 W
Data memory	0.00002 W
Instruction memory	0.00001 W

Table 14: The processor's modules ranked by power dissipation when optimizing for power. Does not include modules with 0 W power dissipation.