Graduate thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

Maren Vorin Fossum

Custimisable waveform generator

Graduate thesis in Electronics System Design and Innovation
Supervisor: Per Gunnar Kjeldsberg
Co-supervisor: Mohammed Saifuddin
June 2022

NTNU
Norwegian University of
Science and Technology

Maren Vorin Fossum

Custimisable waveform generator

Graduate thesis in Electronics System Design and Innovation
Supervisor: Per Gunnar Kjeldsberg
Co-supervisor: Mohammed Saifuddin
June 2022

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

NTNU
Norwegian University of
Science and Technology

Task

Introduction:
Many applications require fine control of both on- and off-chip analog and/or digital buses to perform sophisticated functionalities. The high degree of complexity of these applications necessitates the controller to be able to execute tasks with clock-cycle precision. In addition, some sub-functionalities also require their own waveform generator that performs a specific task depending on the operation that is performed, triggered by the main controller. Furthermore, issues found after chip production adds significant cost if the wafer fabrication masks must be modified; and so, an easily programmable solution is desirable.

Goal:
The main goal of this project is to implement and synthesise low power, low cost (gate count) and highly customizable solutions that can generate the desired waveforms. The design should be easily integrated into future designs.

Scope of work:
This is a continuation of the project work from previous semester where the student investigated different architectures for waveform generators, including open source solutions, and then proposed an architecture for a waveform generator.

This scope entails building a synthesizable waveform generator in SystemVerilog, then performing power and gate count analysis of the implementation and test it out on FPGA to the extends that time allows. The student should consider developing a compiler, or modify an existing open source compiler e.g. GCC, LLVM compatible with the chosen architecture as a way of making the generator easily integrated into future designs. In addition, the student may also create a visualizer which can display the generated output waveforms without running RTL simulation.

Abstract

A waveform is a set of signals that helps drive and control different parts of digital designs. To create a waveform a waveform generator is needed. This generator can be implemented in many different ways depending on the needs of the digital design. However it is common that the waveform has little room for change once the design is made. In this work we aim to add customizability to the waveform generator. In this way future waveform generators do not need to be changed when designing new devices. This reasearch is therefore focused on small compact solutions that can be customized. In previous work some theoretical solutions were proposed, and will now be implemented.

The memory can be reprogrammed to achieve the customizability, but to save area there were proposed 5 solutions how to reduce the memory size needed. The first solution is to save all the bits of the waveform in the memory, the second solution is to save the position where each signal toggles, the third solution is to save which bit is changed, the fourth solution is to save the toggle position of which bit that changes and the last is a solution that saves waveform blocks.

The design that can sustain the fastest clock frequency is the solution that saves all the bits of the waveform in memory since no extra assembly is needed. The slowest solution is the change position solution. The design that uses the least amount of space in memory is the Waveform block solution, while the all bit solution uses the most space in memory. The solution with the least overall area is the waveform block solution while the solution with the most overall area is the toggle bit solution

Contents

List of Tables

List of Figures

Source code

Chapter 1

Introduction

1.1 Motivation

Electrical systems are becoming more and more essential in today's society. One of the directions that technology is developing is towards autonomous cars. One of the companies that contributes to this is Sony Semiconductor Solutions Europe. They collaborate with the key players in the automotive industry in Europe to define and implement the next generation image sensors. These sensors are used for safety systems as well as display based visual, and augmented reality functions in cars[SP21]. These sensors need to be small and effective to be profitable. Designing such electrical systems from scratch takes time and cost money, therefore it is desirable to have some general components that can be reused multiple time on different designs. One example of a general component that can be used in several systems is a waveform generator. A waveform generator can be used as a controller for the applications that need to do a specific task or initialize other modules in order. However, it is hard to design everything perfectly and issues found after chip production add significant costs if the wafer fabrication masks must be modified. It is therefore preferable if the waveform can be reprogrammed as long as this does not cause the are and power consumption to be increased to an unacceptable level.

A waveform generator is not a new concept. The waveform is a vital part of the design, often able to execute tasks with clock-cycle precision. A normal FPGA can be used as a programmable waveform generator, but the resulting area and power consumption of the design are not suitable for the small sensor systems that are in demand. Therefore it is desirable to create a small and practical customizable waveform generator that can be used by different types of ASICs.

1.2 Objective

The main goal of this project is to implement and synthesize a low power, low cost, and highly customizable solution that can generate the desired waveforms. The design should be easily integrated into future designs.

1.3 Main contributions

- Implemented 4 different waveform generators that can be customised

- Compared the resources and memory usage of each solution
- Designs with clock speed between 200MHz and 60MHz

1.4 Structure

In Chapter 2 some theory will be presented. This is to help understand the rest of the report. As this is a continuation from a pre-study, a short summary of the necessary results that is needed to understand this work is given in Section 3.2.Next in Chapter 4 the design of the implementation will be explained. Here each module of the design will first be presented before explaining how they are connected. The test plan that makes sure the design works will also be described here. Chapter 5 shows the result of the test described and some discussions of how the design compares to other solutions of waveform generators. At the end of the chapter there will also be presented some thoughts on future work Lastly in Chapter 6 the conclusion will be presented as well as the reference list.

Chapter 2

Theory

2.1 Waveform

In the context of digital circuits a waveform is a combination of several signals where each signal switch between the binary values of one and zero. Ideally the signals would use no time in switching value, but in reality there will always be some delay as shown in Figure 2.1. This delay comes from the analog component's transfer characteristics and can cause some differences between theoretical behaviour and the actual behaviour [HH07].

None of the signals in the waveform is exactly like each other, meaning all the signals will change at different times. However this does not mean that two signals can not change at the same time. The shape of the waveform is dependant on what the purpose of the waveform is. If a waveform is used as a means to control other part of the circuit, then there will be some constraints on how each of the signals can propagate compared to each other. One example is that a signal only can be high if another signal also is high.

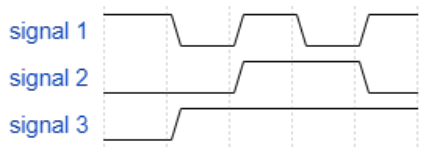


Figure 2.1: Example waveform with 3 signals

2.2 Memory

There exist two main types of memory components called Random access memory (RAM) and Read only memory(ROM). If the memory needs to be re-programmable only RAM and EEPROM is normally used. EEPROM is a form om memory component that originates from ROM, but is able to be re-programmed multiple times. However EEPROM can be quite slow so it is often better to use RAM since this is faster and cheaper[Tys09].

In addition to the main memory it is also normal to have several temporary storage of intermediate data in the form of registers. Memory is a important component in any design, And is normally one of the limiting factors when designing a embedded system designs[CJLZ11].

2.3 Generating designs

To generate electrical design it is necessary to describe the hardware and characteristics of the design. One language that is able to make such descriptions are Verilog and SystemVerilog. SystemVerilog is a hardware description language that is an extention from the Verilog language. Verilog was originaly primarly for designing and testing at the Register Transfer Level(RTL), while SystemVerilog added means for describing testbenches, defining functional coverage, and specifying assertions[CDHK10]. Verilog have a lot of similar syntaxes and semantics as C, but is more directed torwards hardware modeling[Moh12].

To make use of the descriptions given by the hardware description language a compiler is needed. A compilers job is to translate high-level language source code into machine-specific assembly code. Compilers uses a built in model of the target processor that captures the compiler-relevant machine resources, including the instruction set, register files and instruction scheduling constraints[LHC$^+$05].

It is also possible to use a compiler to translate the design descriptions into using resources on a target FPGA. One program that does this is Vivado. Vivado is a software designed by Xilinx that synthesise and analyse hardware description languages. The Vivado High-Level Synthesis compiler translates the hardware description language to a bit file that can be directly targeted into Xilinx devices. This means that it is no need to create a separate register transfer level code to integrate the code into the devise [Xil].

2.4 Area and energy usage in FPGA and ASICs

Power consumption is one of the most important limiting factor when deciding between using an FPGA or an ASIC. An ASIC will normally use a lot less power and is therefore more efficient for specialised design[AAE06][Hau98].

Measuring the actual area of an FPGA implementation is not easy as most design does not fully utilize all the logic on an FPGA. Instead the area is calculated by how many logic resources that is used[KR06].

In an ASIC it is possible to place each components very close to each other in a strategic way with a physical short critical path, but this is not possible in an FPGA. However most logic gates and flip flops in the FPGA is placed close together, only the clock crystal may be a bit further from what it would in an ASIC.

Chapter 3

Design constraints and results from pre-study

3.1 Design constraints

To achieve a customisable waveform generator it is fully possible to use an FPGA. However an FPGA is not normally a good solution for ASIC design. In this case the waveform generator should be compatible on an ASIC design, so purely FPGA solutions have not been researched. However some inspiration from different FPGA solutions were considered in the pre-study, but as they are not relevant to the final implementation.

The implementation will not be made physically instead an FPGA will be used to test the design. The FPGA that is available is the xc7z010iclg225-1L from the Zynq-7000 series. This has a Dual ARM Cortex-A9 MPCore with CoreSight and uses a 28nm technology. The max frequency of the clock is 766 MHz.

The data in the input and output can wary, but the amount of I/O signals remains the same. For this implementation the output amount is 8 signals. There is no restrictions for how many input signals there should be, but it is desirable to be able to give some instruction to the waveform generator. The instructions that can come is a jump instruction, reset instruction or a do nothing signal.

3.2 Pre-study summary

Memory is a large component in most digital designs, and if the waveform should be saved in the memory it should use as little space as possible. To reduce the space a waveform takes in memory, 5 solutions were proposed in a theoretical research paper[Fos21]. An example of a waveform is given in Figure 3.1. Here it shows a simple waveform with three signals called A, B and C over 5 clock periods.

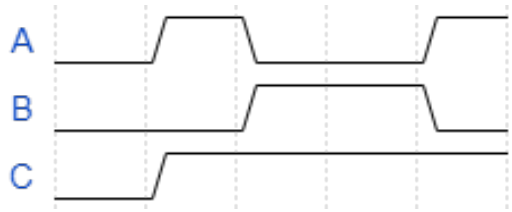


Figure 3.1: Simple waveform with three signal over 5 clock cycles

An overview of how the 5 different solutions would save the waveform from the example in Figure 3.1 is shown in Table 3.1

Table 3.1: The 5 solutions saving a simple waveform

(a) All bits solution

A B C
0 0 0
1 0 1
0 1 1
0 1 1
1 0 1

(b) Toggle solution

	pos.
A	2, 3, 5
B	3, 5
C	2

(c) Change bit solution

-
01, 11
01, 10
-
01, 10

(d) Change position solution

pos.	change bit
2	01, 11
3	01, 10
5	01, 10

(e) Wave block solution

addr.	A B C	repeat	inst.
0x0	0 0 0	1	0x0
0x1	1 0 1	1	0x1
0x2	0 1 1	2	0x2
		1	0x1

The first solution is showed in Table 3.1a. This solution involves saving all the bits of the waveform. This can take up a lot of space in memory if the waveform is very large, but it does not need any counters or extra logic. The second solution involves looking at each individual signals toggle position. This method is beneficial if there are few toggles, and all happens at different times. An example of this is the C signal that only toggles one time in the span of the five clock cycles. The third solution proposed were to save which bit change as shown in Table 3.1c. Here each signal gets a numerical value, in this case signal A got the value 1, signal B got the value 2 and signal 3 got the value 3. This method is beneficial if there only is one signal that change at a time and that the total amount of signals in the waveform is 2 to the power of N, where N can be any whole number. However in this solution there will be some space where none of the signals change. Therefore to reduce the size further a forth solution were proposed where the changed bit is saved with the position the change happens as shown in Table 3.1d. The last proposed solution to save space in memory were to separate the wave construction information with the operation instruction. The wave memory contains every possible combination of each signal that is used in the waveform. From the example waveform there only is three combination the waveform makes as shown on the left table in Table 3.1e. On the right is the operation instruction. This tells how the data from the wave memory should be combined to create the waveform from Figure 3.1.

3.3 Results from pre-study

In the pre-study some of the different solutions theoretical values were calculated. This results is shown in Table 3.2. To calculate this results the waveform shown in Figure 4.14 were used. These values are purely theoretical and will be compared to the actual results in Section 5.5.

Table 3.2: Comparison table from pre-study[Fos21]

method of saving	**size needed in memory for waveform in Figure 4.14**	**customizability**
all bits solution	1224 bits	49 write operations needed
toggle solution	400 bits	2 write operation needed
reduced toggle positions	704 bits	8 write operation needed
change bit solution	711 bits	49 write operation needed
change position solution	276 bits	6 write operation needed
wave block solution	288 bits	6 write operation needed

It is worth noting here that the reduced toggle position solution is a solution with similar structure as the toggle solution. The only difference is that it does not have a fixed amount of space for the amount of toggles in one counter period. This resulted in varied memory size for each signal in the solution. This needed extra control logic that knew exactly how many times the signals toggled. This extra logic that needed to be added or removed depending on the waveform removed the customizability part of the solution. Therefore this solution is not explored further as it does not achieve the minimum requirement of customizability, and preformed worse than the original method.

Chapter 4

Design

4.1 Overview

This design is based on the theories presented in the project preceding this thesis[Fos21] as described in Section 3.2. To make the waveform generator work it needs more than just a memory. An suggestion to the other different parts that are needed is shown in the block diagram in Figure 4.1.

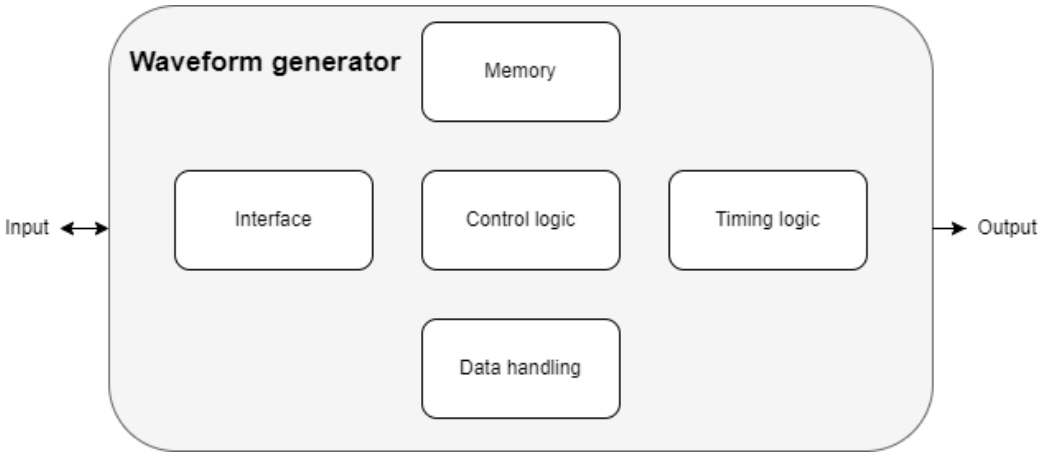


Figure 4.1: A block diagram of the design and its required parts

The design mainly consist of five parts that does different task. The main part is the memory, this is where all the information needed to generate the waveform is stored. The control logic is the second biggest part. This controls which part of the memory that needs to be read, the internal clock and enable signals, and what to do if something needs extra operations. The data handling part will read from memory and send relevant information to the correct modules for implementation. The timing logic is the last part before the waveform is sent out. This part ensures that the timing of the output is correct. The last part is the interface logic. This is not active during normal operation and is only active when the waveform needs to be altered.

4.1.1 Connection

Each module need to be connected to each other. A block diagram of how the parts are connected is shown in Figure 4.2

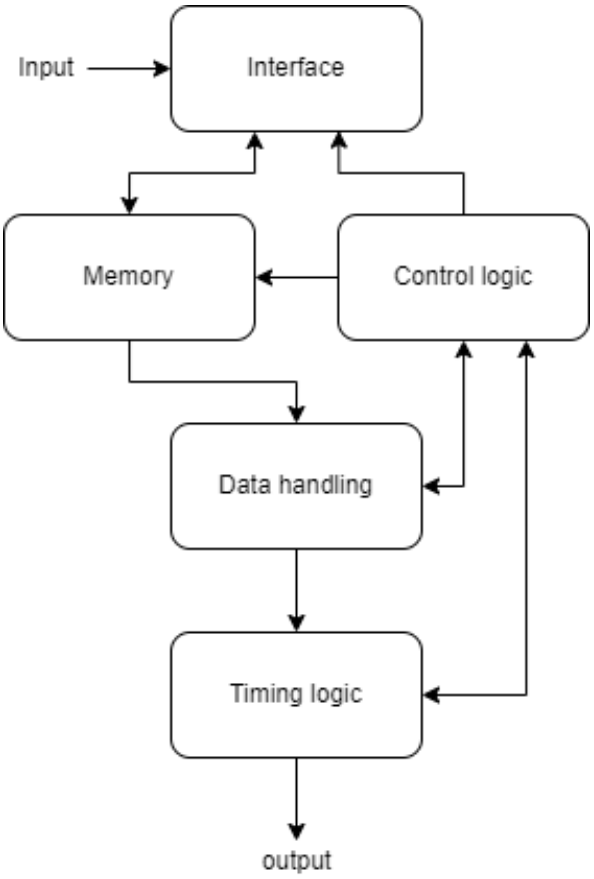


Figure 4.2: Block diagram of how each module is connected

The interface part will often be in sleep mode, so the other four are responsible for generating the waveforms. As seen in the block diagram the different parts communicate in a circular pattern. If the data handling read from the data in memory that it needs to to jump to a new address, the inner circle will be used. The control logic is then responsible for getting a new address for the memory, and ensure that the design is not stuck in an eternal loop without sending data to the output. Eventually a instruction containing data that should be sent to the output should be read and then data handle will send this data to the timing logic. As there must be a new entry to the timing module before one clock cycle have passed, a signal will be sent to the control logic that it should start fetching the next instruction before the previous is finished.

The different solutions all have some difference when connecting each module, but the functionality of the modules remain mostly the same. In the next section each module will therefore be presented, and a greater detail of how the modules are connected in the different solutions will be introduced later in Section 4.3.

4.2 Module description

Inside the main five design parts there are several modules that helps preform the tasks. Here the different modules will be explained with some of the code of the implementation.

4.2.1 Interface

The interface will mainly be in sleep mode in normal operation. It is therefore preferred to have a small and simple design that does not have a large impact on the overall design. The interface for this design is therefore modelled after a SPI interface. To limit the number of wires, the interface will look for special commands like write or read. It will then wait for the next change to know the address followed by data. The specific commands that are accepted are shown in Table 4.1

Table 4.1: expected commands from interface

Command	Explanation
00	start writing at available address
01	start writing at this address
10	write this data
11	read data

The interface consist of 10 bits where the first two bits needs to be a command from the table. The remaining 8 bits is the ones that contains the data. The code can be seen in Listing B.1.

4.2.2 Memory

The memory section consist of the information on how the waveform should be generated. As discussed in the project preceding this thesis this can be done in 5 different ways[Fos21]. Depending on the methods, the size of the memory also varies. However, there are some similarities between all the possibilities. The saved data needs to have some form of identification that difference each type of data. This identification is therefore always the first 4 bits in each address. The biggest difference between the solutions are where each information is stored. In the waveform building block solution it is needed two different memory locations, while in the others one big memory is required instead. To implement this a large register is needed, the code is given in Appendix A.

4.2.3 Control Logic

The control logic is the most complex part and consist of 4 smaller modules.
init module:
The task of the init module is to control the start of the possess. The most essential part of this module is that it makes sure all the other modules is reset before the process is beginning. The reset signal is active until the clock signal has stabilised after booting up.
wait module: The task of the wait module is to do nothing for 16 clock cycles. This can be interrupted by the interface if needed. The purpose of the module is to shut down the other processes while there is a long period where the waveform does not change. The output of the waveform generator needs to remain the same, and no new data should be fetched in the control logic, however the process needs to start again after the timing period has ended. The number of clock cycles to wait is decided basted on the test waveform that will be described in later sections.
next address module:
The next address modules code is shown in Listing B.2 and its expected behaviour is shown in Figure 4.3.

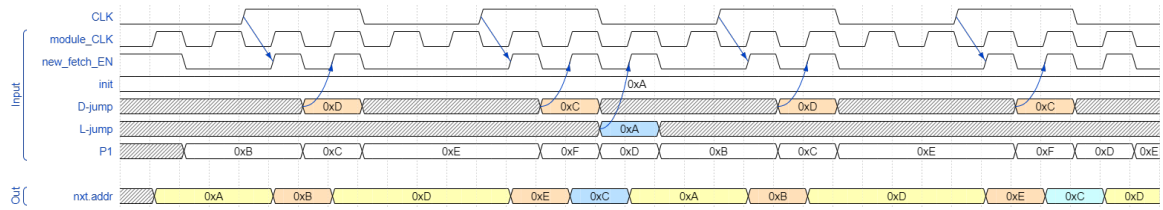


Figure 4.3: A waveform of how the next address module handle the expected inputs

Here the out value describes the address of the memory where the data lies. The different colours describe what type of data that lies in the address. The yellow colour symbolise a normal wave that needs to be sent to the Delay module. The red colour symbolises that the data at that location contains a D-Jump command. The blue colours describe that a L-Jump command is at that location and needs to be send to the Loop module. The darker blue symbolises that the loop is not finished and therefore will send a jump command back to the start. The lighter blue describes that the loop have reached its final iteration and that the following command should exit the loop. The next address module obviously has no information about the colours, as this is decoded in the Decoding module, but is added to visualise why the inputs change.

The top signal in Figure 4.3 symbolise the output clock of the entire design. The next address module needs to have the next address ready before one clock cycle have passed. The second signal is the module clock. This goes four times faster than the output clock. The next_fetch_EN signal is a synchronous enable signal. This signal tells the module when it is ready to start outputting a new address. The output will only change when both module_CLK and next_fetch_EN is high. This enable signal is controlled by the enable module.

Enable module:

The enable signal that is in the next address is controlled by the data handling module, the Loop module and the Delay module. The data handling module will set the signal high shortly after sending the D-Jump command like shown with the arrows in Figure 4.3. Similarly the Loop module will also set the signal high right after sending the address it need to jump to. The rest of signal tops is controlled by the Delay module. The Delay module uses the output clock signal to determine when the signal should be set, so the tops is always a little after the clock change. Since there are three sources that can effect the signal, an module were necessary to ensure that there was no errors accruing if more than one of the sources activated the enable at the same time. A snippet of the code is shown in Figure 4.4.

```
module enableCtrl (
    input logic delay_fetch,
    input logic loop_fetch,
    input logic jump_fetch,
    output logic enable);

    always_comb
        enable = delay_fetch | jump_fetch | loop_fetch;
endmodule
```

Figure 4.4: Code for implementing the enable module

Program counter:

The program counter module work in combination with the next address module to update the address. The program counter will always increase the current address by one, however

this address will only be used if there are no jump instructions. If a jump occurs the address in the program counter will also be updated and the module will starting to increase the address by one from the jump address. The implementation code for this module is shown in Figure 4.5.

```
module programCounter (
    input logic rst_all,
    input logic fast_clk,
    input logic[7:0] current_address,
    output logic[7:0] P1);

    always @ (posedge fast_clk)
        if(rst_all)
            P1 <= 1'b0;
        else
            P1 <= current_address + 1'b1;
endmodule
```

Figure 4.5: Code for program counter

4.2.4 Data handling

The data handling module gets an address from next address module and reads the data in the memory location corresponding to the address. As stated earlier the first 4 bits decide where to send data. A list of the different data types is given in Table 4.2 and Table 4.3. The reason that there are two different data type sets are that the memory size varies between the solution. In model B there is only expected to be 4 bits containing data for the waveform, while model A expects that there are 8 bits available. However, the address is expressed with 8 bits, so in model B the data handling module wait on sending the data until the next input comes. If the corresponding part 2 does not come, the module will act as if it never received the part 1 command.

Table 4.2: Data types model A

Bits	Data type	Where it should go
0000	no data at given address	Next address
0001	Do nothing	Wait module
0010	Normal wave	Delay module
0011	Reset all modules	Init module
0100	Loop start	Loop module
0101	Loop end	Loop module
0110	D-jump inside loop	Next address module
0111	D-jump outside of loop	Next address module
1000	not in use/error	Interface
1001	not in use/error	Interface
1010	Normal inverted wave	Delay module
1011	Irregular output change	Delay module
1100	inverted loop start	Loop module
1101	Inverted loop end	Loop module
1110	Inverted D-jump inside loop	Next address module
1111	Inverted D-jump outside of loop	Next address module

As seen from this table the most significant bit is simply telling if the data should be inverted or not. If the data is inverted then all the bits in the data will change. It is also possible to only use the first half of the table if the inverting option is not wanted, and thereby be able to reduce the data type bits from 4 to 3.

Table 4.3: Data types model B

Bits	Data type	Where it should go
0000	no data at given address	Next address
0001	multiple commands for wave	Wave assembly
0010	Normal wave	Wave assembly
0011	Do nothing	Wait module
0100	Loop start part 1	wait
0101	Loop start part 2	Loop module
0110	Loop end part 1	wait
0111	Loop end part 2	Loop module
1000	D-jump inside loop part 1	wait
1001	D-jump inside loop part 2	Next address module
1010	D-jump outside loop part 1	wait
1011	D-jump outside loop part 2	Next address module
1100	Irregular output change part 1	wait
1101	Irregular output change part 2	Delay module
1110	not in use/error	interface
1111	not in use/error	interface

Depending on what the first 4 bits are, the remaining 8 bits can contain different information. In a "normal wave type" the 4 bits following the data type bits will contain the repeat count, while the remaining bits either contains the desired output waveform or the address it is saved at. The multiple commands for wave command is special for the change bit solution and enables multiple signals to change at the same time. This commands store the data and waits for the next normal wave command, before combining them so the change bits change at the same time. The irregular output change is a command that changes the read out value of the output without affecting the output. This is used when looping needs to reset the value to a reference point for the solutions that depends on the previous output to determine the next output. In the "loop start type" the 8 bits are the number of times the loop should repeat, while the bits in the "L-jump type" contains the address of the start of the loop. L-jump is just a normal jump commands that comes when looping is used. The D-Jump command is simmilar in that it is just a normal jump command, but originates from a planned jump in the instruction data. The data in the "D-jump types" contain the address of where the jump should go. The difference between the two is if the D-jump should break out of the loop or not, but the data is not affected by this.
Currently with this setup it is not possible to utilise nested loops. However it is possible to implement by taking advantage of the unused data types, but this will not be implemented in this design.

4.2.5 Timing

The main task of the timing part is to control the output delay and repeat amount. The output delay is the amount of clock cycles the waveform stays unchanged, while the repeat

amount is the number of times a waveform pattern should repeat. The respective functions are split into two modules called the delay module and the loop module. To be able to handle the timing, a counter is implemented. The counter is present in both timing modules and consist of an adder and a register. The delay module will only need a small counter of 16 for the test waveform that would be described in Section 4.4, so this is used as a limit. The loop count will on the other hand need to be larger as one pattern often is reused multiple times. As the max amount of bits that are available in the data is 8, the counter max count is 4096.

Both counters have an input for reset and increase. The reset is a synchronous signal that is controlled by the delay and loop module and is activated when the counter have reach the repeat count from the data. The reset signal will reset the count to zero, this will also happen if the count goes over the max count. The increase signal is a signal that tells the counter when it should increase the count. For the delay module this is on the positive edge of the output clock. The increase signal for the loop counter is when a loop end type command is received in the data handling module.

Loop module:

The microarcitecture of the loop module is given in Figure 4.6.

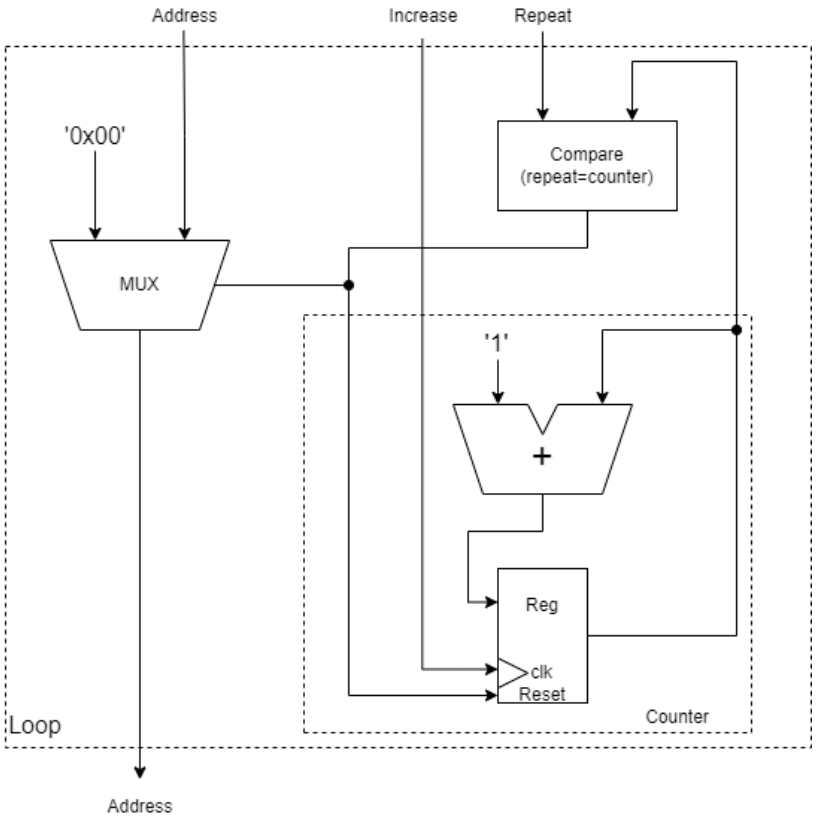


Figure 4.6: Loop architecture

The loop module have tree input that it receives at different times. The first input is the repeat count that comes when a loop start type command is read in data handling. The next input is received when a loop stop type command is read from memory. The input consist of a 8 bit address and the logic signal called increase that is sent into the counter. If the repeat value is unequal the counter value, then the address is sent to the next address module. If they

are equal, then the address 0x00 is sent to symbolise that no jump is necessary. Whenever a address or a zero-address is sent, an fetch enable signal is also sent to the enable module.

Delay module:

The wave data for the delay module is similarly structured as the loop module as seen in Figure 4.7.

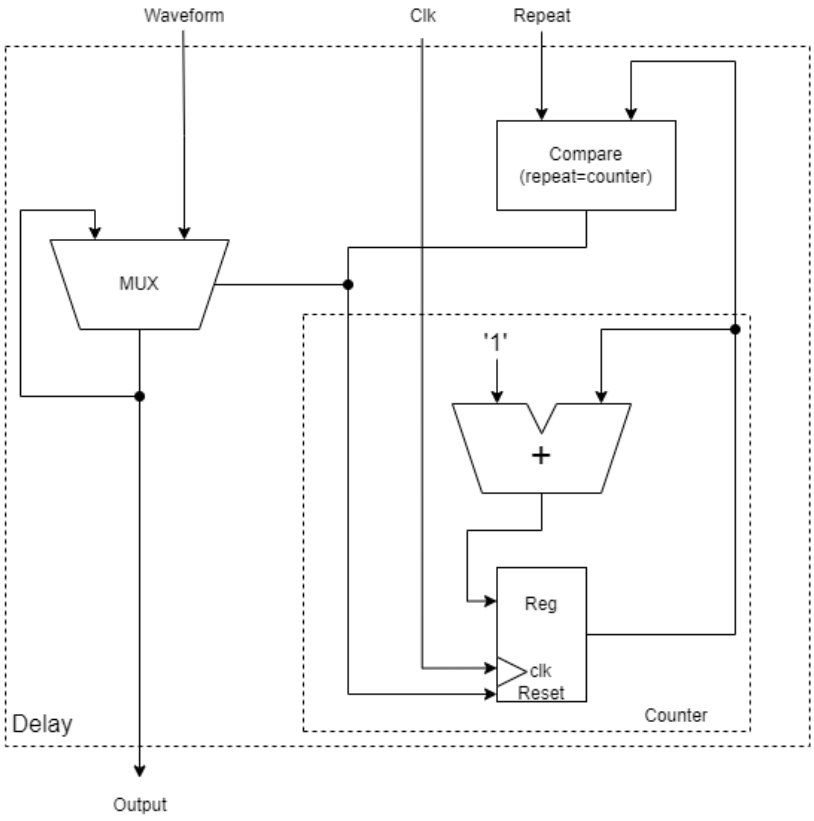


Figure 4.7: Delay architecture

The delay module also have three inputs and one output. The inputs are the waveform, repeat count and a logic signal that tells the counter when it should increase. As mentioned earlier this is the positive edge of the output clock that controls the delay counter. However unlike the loop module, all the inputs here will arrive at the same time. If the repeat count is equal to the counter value then the waveform input will be sent to the output, else the output remains unchanged. When a waveform have been passed trough to the output a fetch enable signal will be sent to the enable module so that a new waveform eventually comes to the input.

In the change bit solution there will also be an additional module called wave assembly that will supplement the delay module like how the counter does. This module takes in the data of which bit that need to change and output the 8 bit waveform that otherwise would come directly from the memory. If the first bit in the 4 bit data signal is high the module will wait on next input as this means that there are multiple bits that need to change at the same clock period.

4.3 Full design

4.3.1 Difference between solutions

A more detailed diagram is needed for all the solutions that will be tested. The architecture of the saving all bits solution is shown in Figure 4.8.

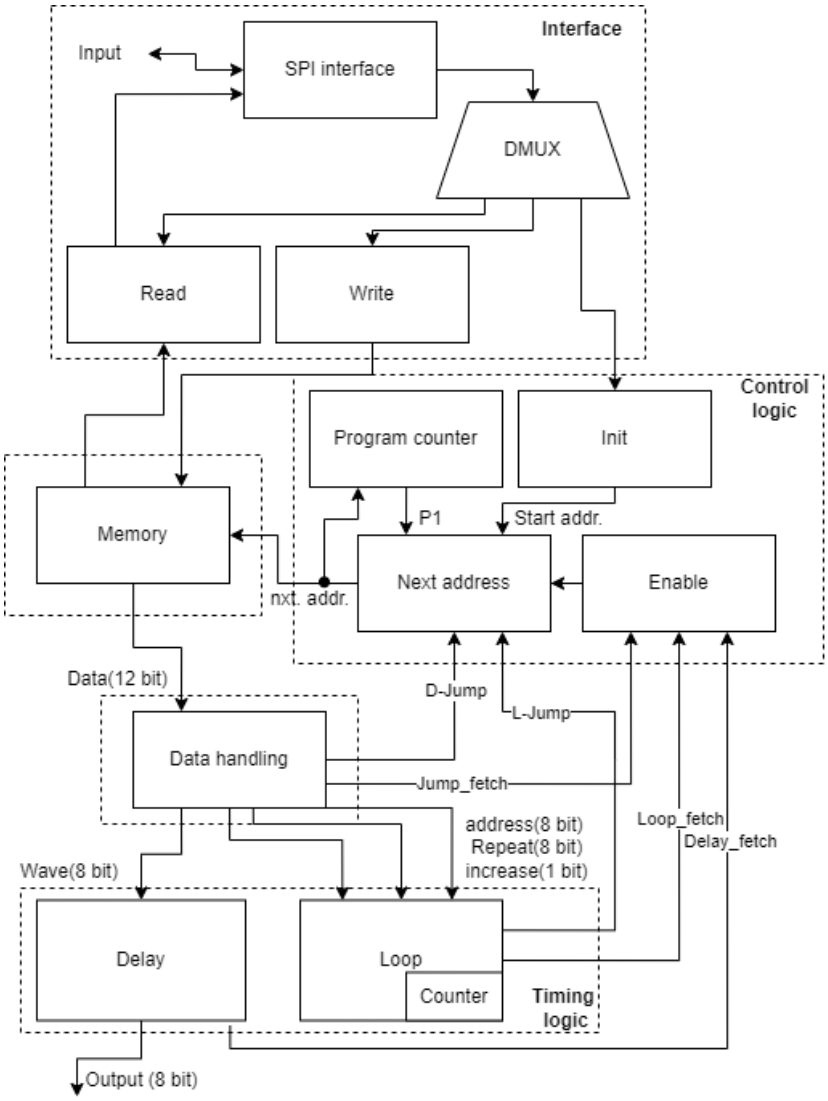


Figure 4.8: Architecture of saving all bits solution

Since all the bits are saved, it means that each memory address is corresponding to one clock cycle. This means that the delay module does not need a counter as all the repeat counter would be 1. The data size in memory would only need 12 bits per address, where the first 4 bits are the data type, and the next 8 bit contain the waveform. This architecture is the basis for the next solutions.

One such solution is the saving of the toggle position. This have an architect like shown in Figure 4.9.

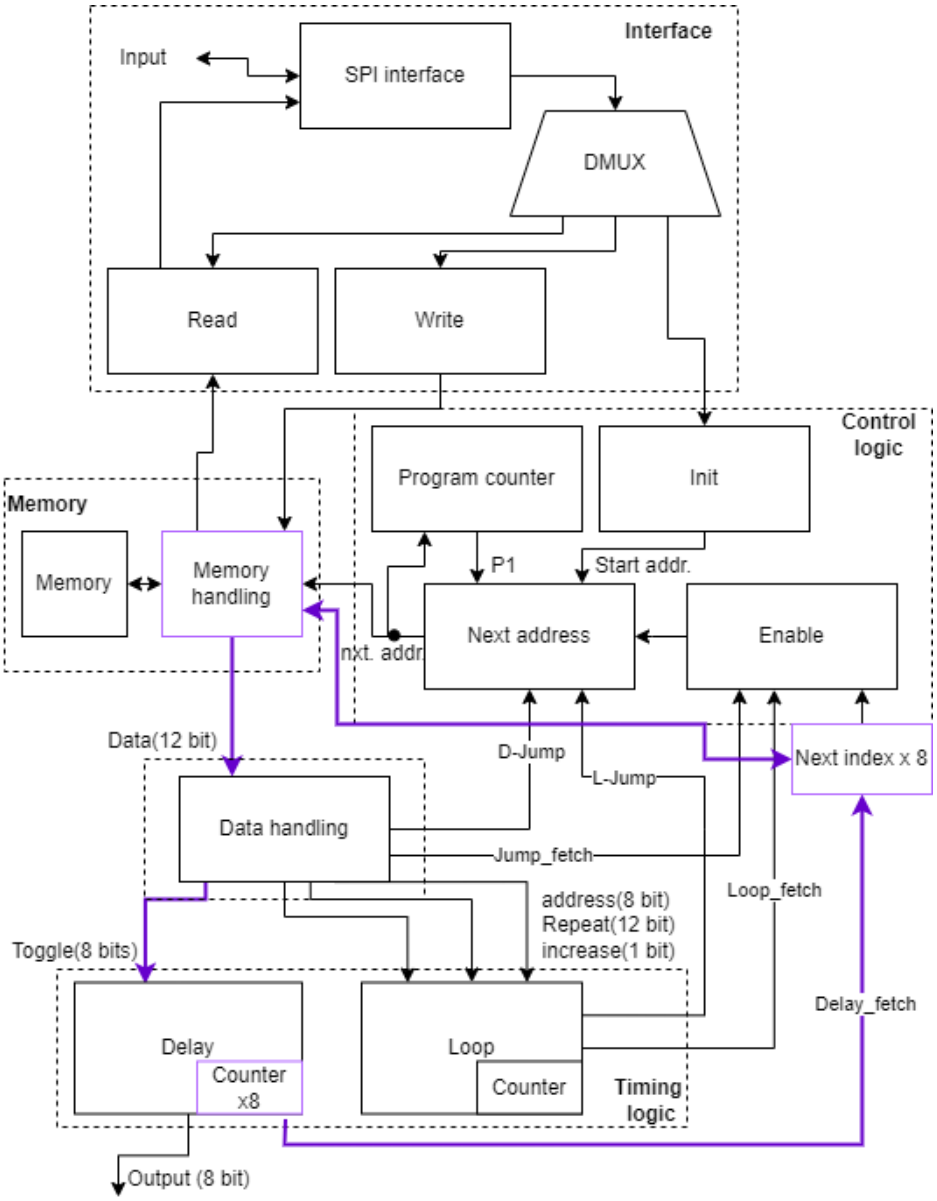


Figure 4.9: Architecture of saving toggle position solution

Here it is important to mention that the thick lines are busses that contain 8 different signals that change independently of each other. In this solution a memory handling module is introduced. This is to handle the reading of the different addresses at the same time. Another addition is the module called "Next index". This module is responsible for changing the index of the signal that is sent to the data handling. In this way it works similar to the next address module, with how it waits on a enable signal and is responsible of sending the output to the memory handling. This module also have a program counter, but do not have any jump addresses coming into it. When the indexing have reach its end, the memory handling will send a signal back to the next index module, so that the program counter in the module can be reset.

An architect of how the change bit solution is shown in Figure 4.10.

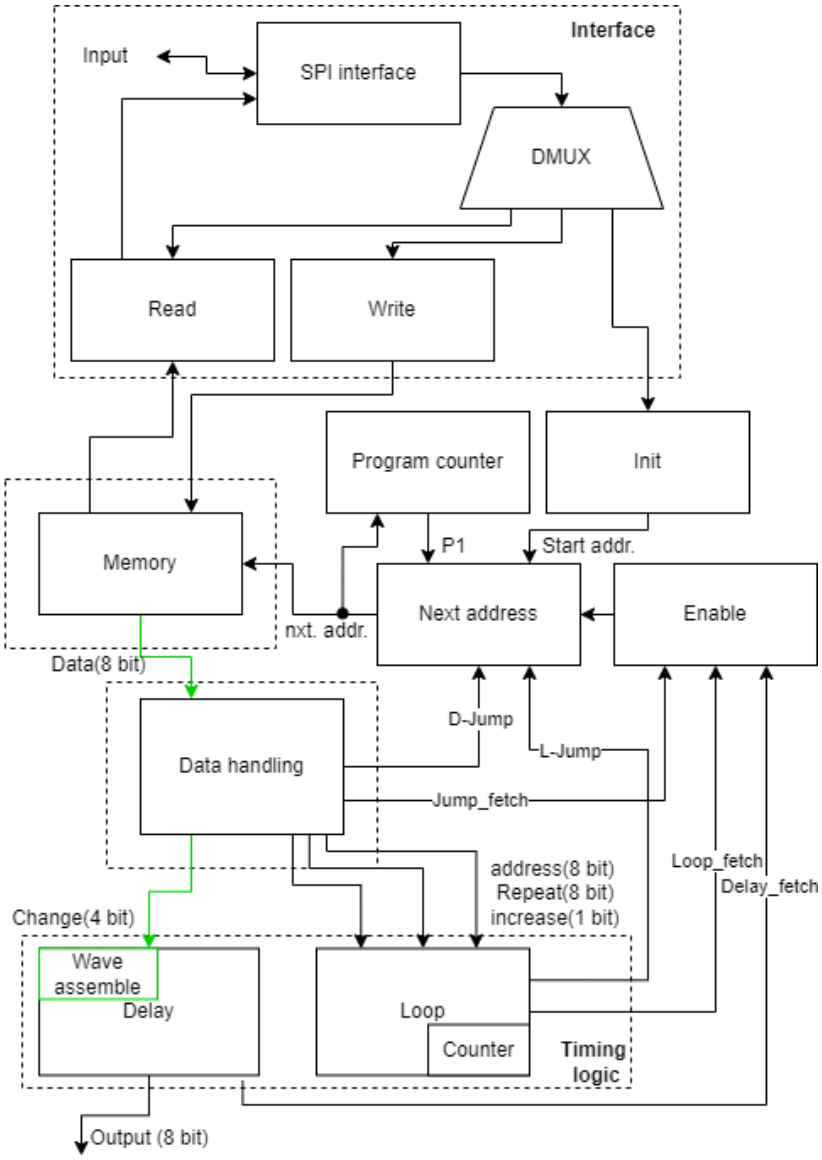


Figure 4.10: Architecture of saving changed bit solution

This solution is similarly structured as the saving of all bits solution. The difference lies in memory size and an extra wave assembly module. The memory only save 8 bits in opposition to the 12 bits that were used when saving all the bits in the waveform. However this difference also change instruction mode to type B.

If the toggle position and the change bit solution is combined it would look like the architecture shown in Figure 4.11.

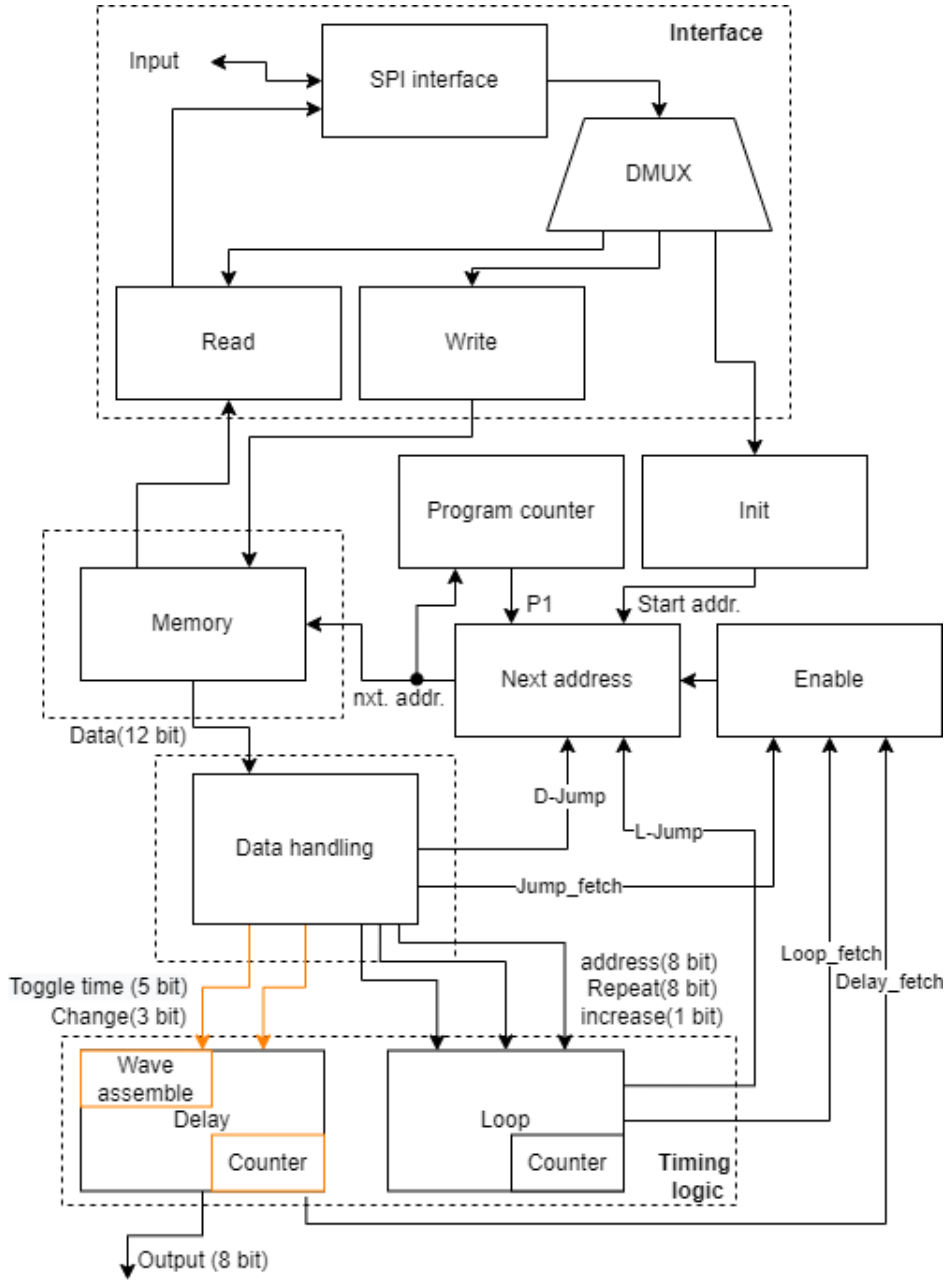


Figure 4.11: Architecture of saving changed bit at toggle positions solution

This solution is very similar to the change bit architecture, however there now is a need for a counter in the delay module. The data size will also change back to 12 bits, and thereby the instruction type switch back to type A. Another difference is that there now is two output from data handling that goes into the delay module. The fist input contains the change bit information, while the other contains the delay value from the toggle position. Here the wave assembly module will never wait before changing the bit, instead if more than 1 signals needs to change at the same time the toggle position on the first change will be zero so that the next change comes immediately after. This means that the register value inside the delay module may change multiple times before one clock cycle have passed and the output updated. This could be a potential problem if a lot of signals need to change at the same time, but the only way to fix this is to increase the internal clock speed to be at least 8 times faster than the

output clock speed and that is not realistic with the current design.

The last possible solution is the waveform block solution. This have an architect as shown in Figure 4.12.

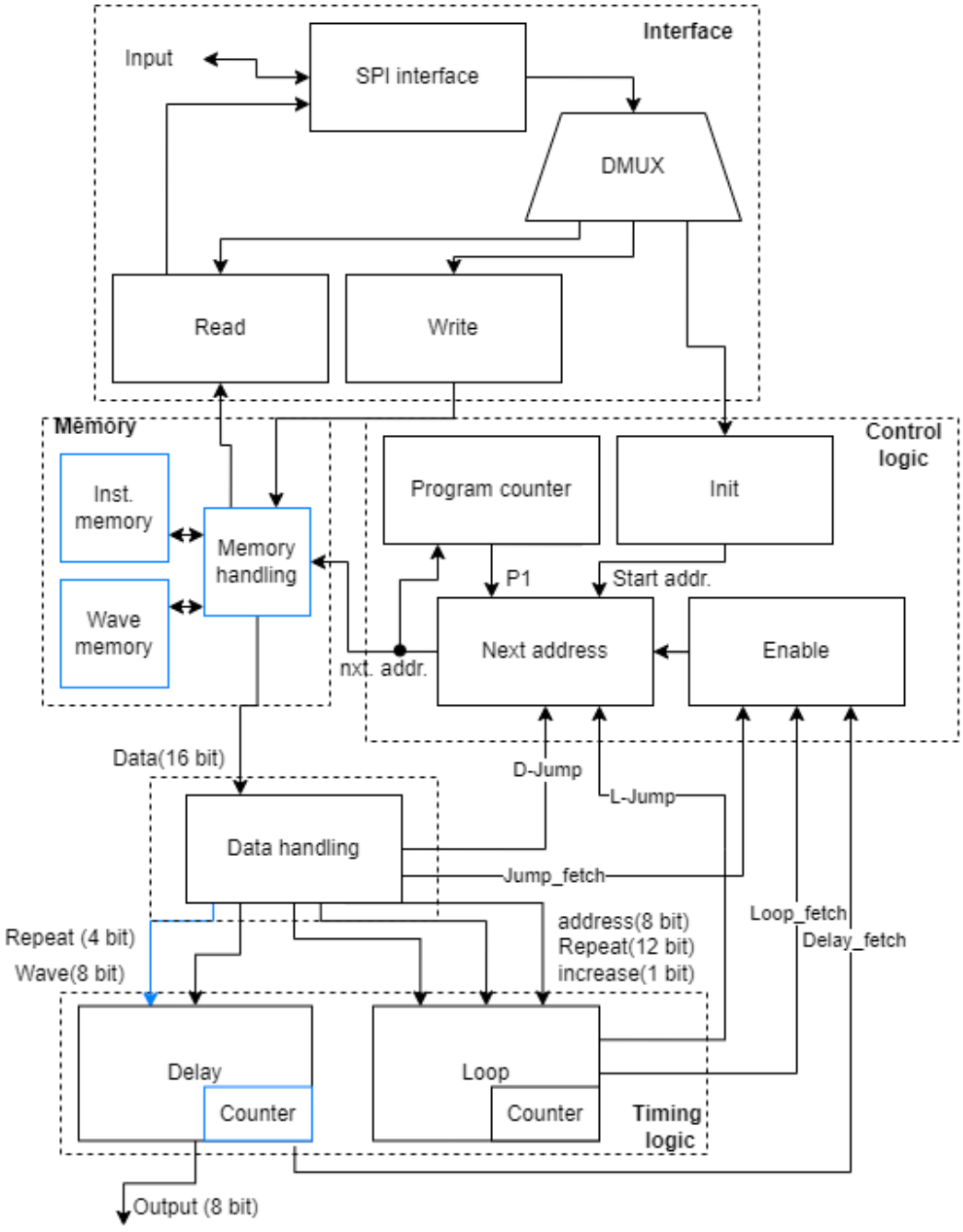


Figure 4.12: Architecture of saving waves in blocks solution

The biggest difference between this and the other architects are the memory handling module that combines instructions and wave info. 16 bits are entering the data handling that uses instruction type A. The delay module does also here take in two different inputs about the wave and the repeat counter.

4.3.2 Comparison

A summary of the difference from each design is given in Table 4.4.

Table 4.4: A summary of the difference in solutions

Solutions	memory size per address	memory handling	Delay counter	Wave assembly	control logic
all bits of waveform	12	No extra	none	No need	Common
toggle positions	12	Handling index input	8 at 6 bit	Timing constrain	extra enable module
changed bit	8	No extra	none	Transform wave	common
changed bit at toggle position	12	extra arithmetic unit	1 at 4 bit	Transform wave	common
wave block	16	Combines instructions and wave	1 at 4 bit	No need	common

4.4 Test plan

A test plan is needed to ensure that the design works. In this section the testing of each module and their connections will be described in greater detail.

4.4.1 Read and Write module

To test the write module the read module needs to be working. The test script will send in inputs that simulate normal input in the design. These inputs will consist of both writing at a specific address and writing where there is no data followed by reading the memory and passing it to the interface. The check is successfully when the output from the interface is the same as what were sent in.

An error handling test will also be preformed. This test includes sending in wrong commands and see if this affects anything followed by a test where the instruction tells the module to write outside the allowed memory addresses. The test is successful if nothing happens in the memory and a correct input is achievable after sending in the wrong input.

The last test is that the read and write module goes into sleep mode in normal operation. This is tested by seeing that the clock input into these modules remains unchanging.

4.4.2 Init module

In this module it is important to test that every other module gets a reset signal at startup. The start address of where the instruction memory is, will also be tested to be correct after reset. Different start address will be tested to ensure that the start address can be anything.

4.4.3 Next address module

The most important functionality test of this module is that the inputs arrive and stays for a correct amount of time for the output to notice them. However the output depends on the different inputs so a event handling is sat in place. A D-jump command should never arrive at the same time as a L-jump command. The only way this happens is if the new_fetch_enable signal rise when it should not. When this happens a priority will take place. Since D-jump

have a higher priority this address is the one that is sent at the output of the next address module. When this happens the L-jump command will be treated as it never arrived.

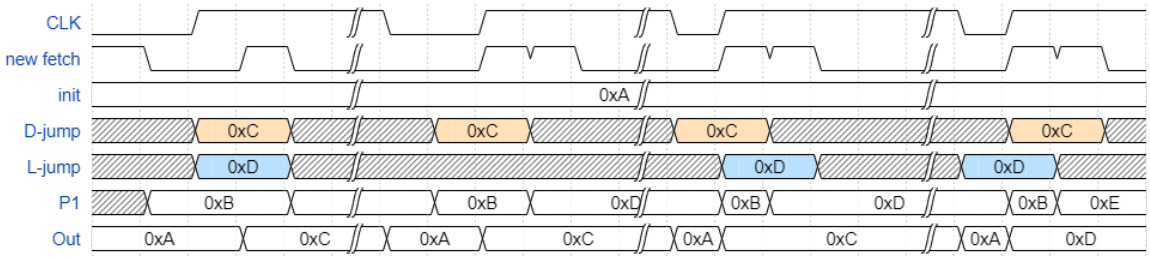


Figure 4.13: A waveform of how the next address module handle the unexpected inputs

The waveform in Figure 4.13 shows the expected output when multiple inputs are waiting.

The program counter will also be tested with the next address module to see that the address always are the output plus one one clock cycle after the output have changed.

4.4.4 Enable module

Test that the output always rise when one of the three signals are high. There will also be a test to see if any of the three signals are high at the same time as this should not happen unless some of the signals is not set low again after they were set high.

4.4.5 Data handling

In the data handling module it is important that every data type is handled correctly. Especially the wait command with the type B command set is tested that the output does not change until the second part comes. Another important thing to check is that the fetch enable signal is activated when a D-jump command or a wait command comes. It is also important that this signal is lowered again before the next high comes.

The test will also check that the input is split into the correct output. The wave info and the repeat count is split into different wires, so there will be checks that these values have not been switched or been changed from the input value.

The last check for this module involves the loop signal "increase". This signal should be created in the data handling module and should only be high when a loop stop signal is at the input.

4.4.6 Delay

Testing of the counter involve seeing that the counter value reset to zero if the count reaches the max value. The reset in itself will also be tested to be able to reset the count to zero even if the count haven't reached max value.

The delay module is suppose to start fetching the next wave as soon as the output is changed. This means that a test to see that the register values that contain the current output wave and possibly the repeat count does not change even when new input arrives. The fetching of new information will also be tested. Here the test entails not fetching new data until the current input is at the output.

4.4.7 Loop

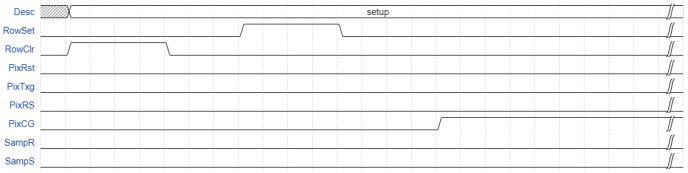
The loop gets the inputs at different times, so a test involving haven gotten the repeat count before the address is preformed. There is also a test to check that the loop count only increase at the loop end command. Other than that the loop is testes by looking at the total functionality of the overall design test.

4.4.8 Overall design test

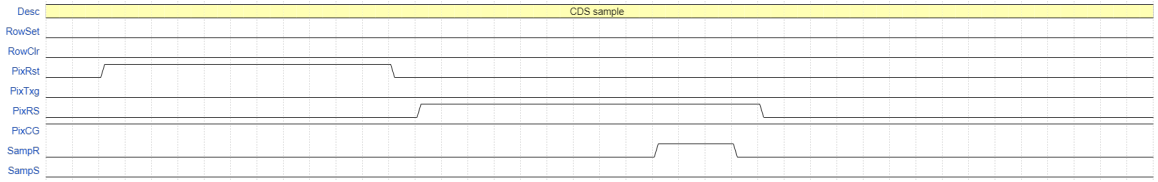
the last tests are for when all the modules are connected between each other. The main functional test is to compare the output of the waveform generator to the expected output. This test will check that the values are correct as well as how long the output stays.

There also is 3 smaller test that checks the connection between each module. The first test is that the output of one module match is corresponding input on the other modules, with the exception of the delay modules output that is the output of the generator. The second test are that the next address module wait when it needs to. This happens when a wave is unchanged at the output. This is important so that the generator don't read several instructions while the output clock still in the same clock periods. The second test ensures that the looping in the output corresponds with the looping count in memory. This test involves that there is a looping start and a looping end command, as well as that the addresses outside the loop also eventually is read. The last test involves the operation with and without activation the interface. Activating the interface should stop the output from changing, and likewise the output should change even if the interface never were interacted with.

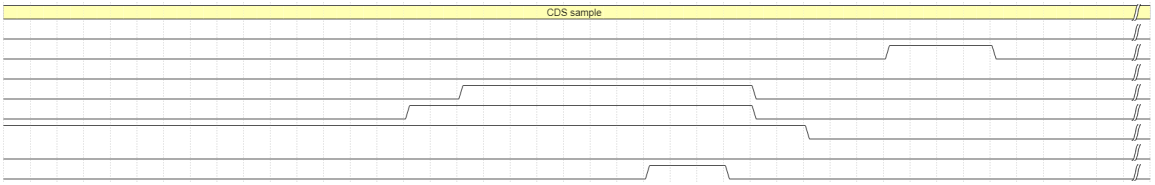
All the overall tests were preformed with the waveform shown in Figure 4.14. This is a short waveform example in itself, however in the test the sample part were repeated 5 times before the shutter part were generated. This result in an waveform of 489 clock cycles instead of the original 153 clock cycles.



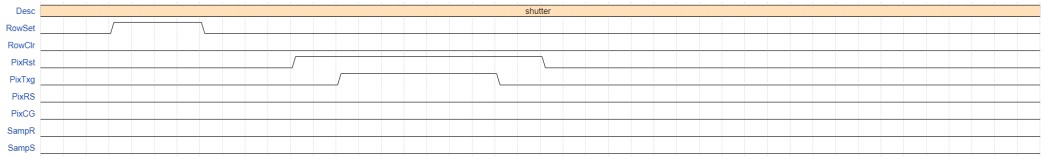
(a) Waveform part 1, the setup



(b) Waveform part 2, the sample



(c) Waveform part 3, the sample



(d) Waveform part 4, the shutter

Figure 4.14: Waveform used in functionality test

The setup part of the corresponding memory for each design when this waveform is used is given in Table 4.5. The whole memory can be found in Appendix A. Here it is worth mentioning that the four most significant bits represent the data type information. Here 0x2 represent a normal wave, meaning this is the waveform information. In the case of the toggle, change bit and wave block solutions there are data at the following addresses, but those addresses are used to represent the rest of the waveform beside the setup part. At address 0x00 there is always a no data data type. This is to make sure the output is not affected at startup as the modules still is being reset. This is normally not needed as the output module should output 0x00 until the reset is done, but are there for extra safety.

Table 4.5: memory structure for the different solutions for saving the setup waveform

address	all bit	toggle	change bit	change position	wave block inst.	waveform blocks
0x00	0x000	0x0000	0x00	0x000	0x000	0x00
0x01	0x200	0x29D0	0x30	0x310	0x210	0x40
0x02	0x240	0x2260	0x26	0x226	0x242	0x80
0x03	0x240	0x200	0x30	0x266	0x230	0x04
0x04	0x240	0x200	0x30	0x297	0x241	-
0x05	0x240	0x200	0x30	0x2D7	0x240	-
0x06	0x200	0x202	0x26	0x202	0x293	-
0x07	0x200	0x200	0x30	0x390	-	-
0x08	0x200	0x200	0x30	-	-	-
0x09	0x280	-	0x27	-	-	-
0x0A	0x280	-	0x30	-	-	-
0x0B	0x280	-	0x30	-	-	-
0x0C	0x280	-	0x30	-	-	-
0x0D	0x200	-	0x27	-	-	-
0x0E	0x200	-	0x30	-	-	-
0x0F	0x200	-	0x30	-	-	-
0x10	0x200	-	0x30	-	-	-
0x11	0x204	-	0x22	-	-	-
0x12	0x204	-	0x30	-	-	-
0x13	0x204	-	0x30	-	-	-
0x14	0x204	-	0x30	-	-	-
0x15	0x204	-	0x30	-	-	-
0x16	0x204	-	0x30	-	-	-
0x17	0x204	-	0x30	-	-	-
0x18	0x204	-	0x30	-	-	-
0x19	0x204	-	0x30	-	-	-

4.4.9 Customizability

All the solutions can be reprogrammed to customize the waveform, however how easy it is to do so also need testing. To test customizability the waveform in Figure 4.14d will be changed into the waveform shown in Figure 4.15.

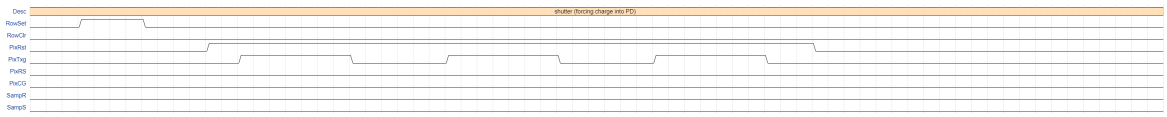


Figure 4.15: An alternative for waveform part 4, the shutter

The main alteration consist of toggling the PixTxg signal three times instead of just one. This cause the total amount of clock periods to increase. Two test will therefore be preformed. One is to add this waveform in the memory without removing the previous alteration, while the other test involves completely replacing the waveform. the main focus on both test is how much extra work this process will take.

Chapter 5

Result & Discussion

5.1 Functionality

All modules works as expected, but when the full test of functionality were preformed on each of the design a varied results can be seen on the toggle solution. The toggle solution is able too work fine when each signal switch an equal amount of times, but this was not the case in the test waveform. The toggle solution uses four bit to state when the signal should change position. However as the waveform last more than 16 clock cycles, the waveform counter restart its count to 1 after it reaches 15. If the four bits in memory equals to zero the signal should go without change for 15 clock cycles. For example the PixCG signal in Figure 4.14a does not change in the first 15 clock cycles, but change in the in the 17th clock cycle, so this is represented by the value 0x02. However the signal RowClr changes twice in the first 15 clock cycles(at position 2 and 6), but none in the clock cycles after 15. This is represented by the value 0x260. As the setup waveform ends before clock cycle 31 comes, the end zero is not needed, but this is not the case for the sample and shutter waveform. To keep consistency the end zero is therefore needed, but this means that the signal is described with 4 more bit in memory compared to the other signals.

This is a problem as the memory needs to be a fixed size. The problem can be solved by adding extra zeros to the other signals, but this takes of extra space in memory. It also creates problems with the timing logic for each of the signals, as some of the signals then have 15 more clock cycles worth of data than the others. This problems also appear when the rest of the waveform is saved, as the sample waveform needs 28 or 32 bits in memory and shutter waveform needs 12 or 16 bits in memory for each signal. To keep consistency for the entire test each address in memory only contain up to 15 clock cycles for the waveform. With this change all the addresses in memory will contains 12 bits, but this also put a new restrain on the available waveform possibilities. Since there only is 8 bits available to describe the toggle position within the 15 clock period, each signal can only toggle twice in this period. If more toggling is needed, the entire memory needs to be increased.

Another problem with this solution is that the waveforms does not contain clock cycles that can be divided by 15. The setup waveform shown in Figure 4.14a contains 25 clock cycles. This means that there is a remainder of 10 clock cycles after the first 15. The sample waveform have an remainder of 9 clock cycles, the original shutter waveform have an remainder of 13 clock cycles and the alternative shutter waveform also have an remainder of 10 clock cycles. To fix this the unused data type commands and the do nothing command from Table 4.2 can be used. This puts another constraint on the possible waveform generation as the waveform

must have an remainder of 0, 8, 10 or 13 clock periods after dividing by 15 with this suggestion.

The toggle, change bit and change position solutions also have some issues in functionallity when it comes to repeating each waveform. This is because all the data in memory relies on the previous output of the waveform generator. For example the setup waveform assumes the output were 0x00 at the start, while the sample waveform assumes the previous output data was 0x04 at the start of the waveform. However when the sample waveform is finished it ends at a value of 0x00. This means that before looping the output value needs to be reset to the start value when looping. This is done by adding an extra instruction in memory of an irregular output change. This uses up one clock cycle of the fast clock, but as long as a normal wave instruction follow it is no problem.

5.2 Area

The utilization of the FPGA that the implementation is using is given in Table 5.1. Here it is seen that the all bit solution uses the least of the resources, however the memory utilization is not included.

Table 5.1: Comparison of the utilization of the different solutions

Solution	Slice LUTs	Slice register	Slice	Lut as logic	FF
All bit	41	57	20	41	51
Change bit	62	78	28	62	53
Change position	67	91	31	67	58
Wave block	44	79	22	44	64

Memory usage can be compared by measuring how many bits each solution needs to generate the example waveform that were described in subsection 4.4.8. This value can also be compared to the theoretical worst case scenario. This comparison can be seen in Table 5.2

Table 5.2: Memory usage comparison

Solution	Total bits used	Whereas contain wave info	Worst case scenario	Saved space compared to all bit solution
All bit	1860	1216	1860	-
Change bit	1296	612	9768	30.32%
Change position	444	264	7344	76.13%
Wave block	484	328	3680	73.98%

From the comparison table in Table 5.2 we can see that the most efficient solution is the change position solution with the wave block solution not far behind. The change position solution and the wave block solution is able to save over 70% of the space compared to the all bit solution.

The worst case scenario is a theoretical waveform with the most ill suiting waveform for each of the solution. To compare the value to the results from the test waveform some restriction were set on this worst case waveform. First is that it is assumed that there are 8 signals like in the test waveform. The waveform is also structured similar to test waveform with an equal amount of clock cycles and needs to be split into three parts. The difference lies in how each

signal change.

It is possible to calculate a worst case for the toggle position by using the suggested fixes, even though the solution currently is not working. In the worst case scenario for the toggle solution each signal will change each clock cycle. To accommodate this the bitsize of each instruction in memory needs to change. In the test waveform each signal only toggled twice in each period of 15 clock cycles, but now they will toggle 15 times. This increases the bits for each instruction, B_{toggle} to be $4*15+4 = 64$ bits. The equation for the worst case, W_{toggle} is therefore given in Equation 5.1.

$$W_{toggle} = (I + P_1 * A_{signal} + I + P_2 * A_{signal} + I + P_3 * A_{signal}) * B_{toggle} \tag{5.1}$$

Here I is the non-wave instruction like loop start and loop end. This takes up one line in the memory. A_{signal} is the amount of signals in the waveform. P is the amount of periods of 15 clock cycles in each waveform. To replicate the test waveform the first waveform is split into 2, the second into 6 and the last into 3.

For the worst case for the change bit solution and the change position solution every signal will change at the same at each clock cycle. This gives the equations as shown in Equation 5.2 and Equation 5.3. Here CL is the clock periods in each waveform. As stated those need to be the same as the test waveform so $CL_1 = 25$, $CL_2 = 84$ and $CL_3 = 43$.

$$W_{changeBit} = (I + CL_1 + I + CL_2 + I + CL_3) * B_{ChangeBit} \tag{5.2}$$

$$W_{ChangePos} = (I + CL_1 * M_{signal} + I + CL_2 * M_{signal} + I + CL_3 * M_{signal}) * B_{ChangePos} \tag{5.3}$$

Do note that there is a difference between the change solutions in that since the change position solution uses data types model A(from Table 4.2) while the change bit solution uses the data type model B(from Table 4.3), the change position solution have the option to invert the normal wave. Here the inverted wave is interpreted as change every signal except the one given in memory. This efficiently means that the memory at max need to instruct four signals to change at the same time instead of all eight, so M_{signal} equals to 4. The bits needed for each instruction is 8 for the change bit solution and 12 for the change position solution, so $B_{ChangeBit} = 8$ and $B_{ChangePos} = 12$.

The worst case for the wave block solution is that there is never a repeat of the previous combination of which signal is high at the same time and at least one signal will change each clock cycle. Since there is a new combination every clock cycles, the amount of combinations, A_C is the same as amount of clock cycles meaning it is 152. To find the total used space the wave memory takes up the amount of combination needs to be multiplied with the space each combination takes. In this solution this will always be the number of signals in the waveform, A_{signal}.

$$W_{WaveBlock} = A_C * A_{signal} + (I + CL_1 + I + CL_2 + I + CL_3) * B_{WaveBlock} \tag{5.4}$$

The all bit solution does not have a worst case as regardless of how the waveform is formed, the size in memory is unaffected. Naturally this means that the all bit solution uses the smallest area in the worst case, but the wave block solution is not that far behind. The worst case of the change bit solution and the change position solution is well over 3 times worse

than the all bit solutions worst case. This may come from that multiple signals change at the same time.

The method that preforms the overall worst with considering to area is the change bit solution. This is probably because of all the empty spaces in memory. In the test waveform from Figure 4.14 there is a total of 112 clock cycles where the instruction from memory tells the generator to do nothing. This means that in total 896 bits of memory is used up to tell the generator that there will be no change in the output for that clock period. In comparison the change position solution only need a max of $152 * 4 = 608$ bits extra to add the information of the change position.

5.3 Timing

The design uses two clocks, but the speed of each clock is different in all of the solution. The main clock that controls the output speed should be around 200MHz, but currently it is only the all bit solution that achieve this. The clock speed is decided by the critical path. The critical path for the all bit solution goes from memory to the data handling module and have a slack of 0.182ns.

The fast clock in the change bit design have a clock frequency of 300MHz. As the fast clock needs to be four times faster than the main clock, the resulting clock speed becomes only 75MHz. The critical path here goes from data handle to the enable module. The reason this takes longer time than from the all bit solution is that the enable module waits on the enable signal that comes from the delay module and the loop module. As there is more logic in the delay module, it uses a longer path. The same critical path appears in the change position solution, but here the fast clock and main clock needs to be 280MHz and 70MHz respectively. Since the critical path were the same, it can indicate that the main problem lies in the assembly module.

The wave block solution have the worst clock frequency of them all. The clock frequency is only at 60MHz. This is due to the enable module waits on the enable signal from the delay module and the loop module before passing it to the next address module in less than one clock cycle. The other solution does not try to this, however there becomes a logic error in the design if this is not done. Theoretical this should not be needed, but currently cannot be fixed.

5.4 Customizability

All the solutions have a good grade of customizability as it only involves writing to memory. So the baseline to determine which solution have the best customizability lies in the number of write operations that is needed.

In the all bit solution adding the alternative waveform takes 71 write operation. You need one jump instruction before the original waveform and 70 extra to add the new waveform. If the waveform should replace the waveform, you need 48 write operation as the waveform is similar in the first 22 clock cycles. This assumes that there is no other wave information after the shutter waveform. If there were two extra write operation would be needed to place the

jump instructions. The change bit solution uses the exact same amount of write operations as it is structured quite similar to the all bit solution.

The change position solution and the wave block solution needs 11 write operations to add the alternative waveform and only 5 if it replaces the original. This is by far the easiest and fastest way of customising the waveform. However it is worth mentioning that the wave block solution would potentially need more write operations if the signal combination contained patterns that does not appear in the earlier waveform.

5.5 Results compared to theoretical values from pre-study

Table 5.3: Comparison of results from pre-study and actual results

method of saving	**Theoretical size needed in memory**	**Actual size needed in memory**	**difference**
all bits solution	1224 bits	1860 bits	636 bits
toggle solution	400 bits	-	-
change bit solution	711 bits	1256 bits	545 bits
change position solution	276 bits	444 bits	168 bits
wave block solution	288 bits	484 bits	196 bits

A comparison between the theoretical values and the actual values is given in Table 5.3. Here it is shown that the theoretical value always is smaller than the actual value. This is because the theoretical value did not consider that the type of data also would take up space in memory. In the implementation there is always 4 bits in the start of each address that tells the data handle which type of data it is.

If the values in the theoretical calculations is compared to only the bits used to contain wave info we get the table shown in Table 5.4.

Table 5.4: Comparison of results from pre-study and actual results of purely wave data

method of saving	**Theoretical size needed in memory**	**Size of purely wave data in memory**	**difference**
all bits solution	1224 bits	1216 bits	8 bit
change bit solution	711 bits	612 bits	99 bits
change position solution	276 bits	264 bits	12 bits
wave block solution	288 bits	328 bits	40 bits

The values does still not completely match for any of the solutions, but are much closer. This can have something to do with an error in calculation. It seems from the results that the theoretical calculations have added an extra clock cycle that should not have been there. The change bit solution is quite far off ass Well. This comes from that the theoretical calculation added 2 extra bits in memory for each instruction. this were to be able to handle two signals changing at the same time, but this were changed to simply use the multiple command for wave when this was needed.

Chapter 6

Future work & Conclusion

6.1 Future work

As mentioned the toggle solution is currently not working as it should. It still have several issues that cause it to output the wrong waveform unless the waveform is in a very specific pattern. This should be fixed and explored further in future work to make the comparison complete.

Some research into the wave block solution is also needed as the current maximum clock speed is fairly low and should be able to be increased. The worst case scenario also remains to be tested and not just calculated theoretically.

For future work it is necessary to test each solution physically on an FPGA. To make the design compatible and easy integrated with future solutions a compiler should also be developed. Here a visualiser would be helpful as this could easier display the current waveform and thereby customise the waveform better.

6.2 Conclusion

Four of the five proposed solution have been implemented and is working. However not one of the solutions have been physically tested on an FPGA. From the simulated results we see that the all bit solution have the fewest LUTs and Flip Flops of only 41 and 51 respectively. However it uses the largest space in memory for the test waveform. The change position solution uses the least amount of space in memory for the test waveform and achieve a saved space of 76.13% compared to the all bit solution, but have the second largest worst case of 7344 bits in memory. On the other hand the wave block solution uses the second smallest space in memory for the test waveform which also saved over 70% of the space used by the waveform in memory compared to the all bit solution. In addition it also second best in the worst case scenario with only needing 3680 bits in memory. Therefore when looking at area the wave block solution is the best alternative, but it have the worst clock speed of only 60MHz. All the solutions have high degree of customizability, but the change position solution and the wave block solution have the best customizability. Overall the best solution is the change position solution, but it have a bad worst case. The wave block solution is not far behind but have a bad clock speed that needs to be improved in future work.

Appendix A

memory for the difference solutions

A.1 All bit solution

Listing A.1: all bit solution

```
1  module memory(
2      input logic fast_clk,
3      input logic wr,
4      input logic rd,
5      input logic [11:0] wr_data,
6      input logic [7:0] addr,
7      input logic [7:0] wr_addr,
8      output logic [11:0] rd_data,
9      output logic [11:0] data
10     );
11
12     integer i;
13      reg [11:0] memory_all_bit [254:0];
14
15     initial begin
16         i <= 0;
17         //saved
18         memory_all_bit[0] <= 12'h200;
19         memory_all_bit[1] <= 12'h200;
20         //setup
21         memory_all_bit[2] <= 12'h200;
22         memory_all_bit[3] <= 12'h240;
23         memory_all_bit[4] <= 12'h240;
24         memory_all_bit[5] <= 12'h240;
25         memory_all_bit[6] <= 12'h240;
26         memory_all_bit[7] <= 12'h200;
27         memory_all_bit[8] <= 12'h200;
28         memory_all_bit[9] <= 12'h200;
29         memory_all_bit[10] <= 12'h280;
30         memory_all_bit[11] <= 12'h280;
31         memory_all_bit[12] <= 12'h280;
32         memory_all_bit[13] <= 12'h280;
33         memory_all_bit[14] <= 12'h200;
34         memory_all_bit[15] <= 12'h200;
35         memory_all_bit[16] <= 12'h200;
36         memory_all_bit[17] <= 12'h200;
37         memory_all_bit[18] <= 12'h204;
38         memory_all_bit[19] <= 12'h204;
39         memory_all_bit[20] <= 12'h204;
40         memory_all_bit[21] <= 12'h204;
41         memory_all_bit[22] <= 12'h204;
42         memory_all_bit[23] <= 12'h204;
43         memory_all_bit[24] <= 12'h204;
44         memory_all_bit[25] <= 12'h204;
```

```
45          memory_all_bit[26] <= 12'h204;
46
47          memory_all_bit[27] <= 12'h403;  //loop start, repeat=3
48          //sample
49          memory_all_bit[28] <= 12'h204;
50          memory_all_bit[29] <= 12'h204;
51          memory_all_bit[30] <= 12'h224;
52          memory_all_bit[31] <= 12'h224;
53          memory_all_bit[32] <= 12'h224;
54          memory_all_bit[33] <= 12'h224;
55          memory_all_bit[34] <= 12'h224;
56          memory_all_bit[35] <= 12'h224;
57          memory_all_bit[36] <= 12'h224;
58          memory_all_bit[37] <= 12'h224;
59          memory_all_bit[38] <= 12'h224;
60          memory_all_bit[39] <= 12'h224;
61          memory_all_bit[40] <= 12'h224;
62          memory_all_bit[41] <= 12'h204;
63          memory_all_bit[42] <= 12'h20C;
64          memory_all_bit[43] <= 12'h20C;
65          memory_all_bit[44] <= 12'h20C;
66          memory_all_bit[45] <= 12'h20C;
67          memory_all_bit[46] <= 12'h20C;
68          memory_all_bit[47] <= 12'h20C;
69          memory_all_bit[48] <= 12'h20C;
70          memory_all_bit[49] <= 12'h20C;
71          memory_all_bit[50] <= 12'h20C;
72          memory_all_bit[51] <= 12'h20E;
73          memory_all_bit[52] <= 12'h20E;
74          memory_all_bit[53] <= 12'h20E;
75          memory_all_bit[54] <= 12'h20C;
76          memory_all_bit[55] <= 12'h204;
77          memory_all_bit[56] <= 12'h204;
78          memory_all_bit[57] <= 12'h204;
79          memory_all_bit[58] <= 12'h204;
80          memory_all_bit[59] <= 12'h204;
81          memory_all_bit[60] <= 12'h204;
82          memory_all_bit[61] <= 12'h204;
83          memory_all_bit[62] <= 12'h204;
84          memory_all_bit[63] <= 12'h204;
85          memory_all_bit[64] <= 12'h204;
86          memory_all_bit[65] <= 12'h204;
87          memory_all_bit[66] <= 12'h204;
88          memory_all_bit[67] <= 12'h204;
89          memory_all_bit[68] <= 12'h204;
90          memory_all_bit[69] <= 12'h204;
91          memory_all_bit[70] <= 12'h204;
92          memory_all_bit[71] <= 12'h204;
93          memory_all_bit[72] <= 12'h204;
94          memory_all_bit[73] <= 12'h204;
95          memory_all_bit[74] <= 12'h204;
96          memory_all_bit[75] <= 12'h204;
97          memory_all_bit[76] <= 12'h204;
98          memory_all_bit[77] <= 12'h204;
99          memory_all_bit[78] <= 12'h204;
100         memory_all_bit[79] <= 12'h204;
101         memory_all_bit[80] <= 12'h204;
102         memory_all_bit[81] <= 12'h204;
103         memory_all_bit[82] <= 12'h204;
104         memory_all_bit[83] <= 12'h204;
105         memory_all_bit[84] <= 12'h204;
106         memory_all_bit[85] <= 12'h20C;
107         memory_all_bit[86] <= 12'h20C;
108         memory_all_bit[87] <= 12'h21C;
109         memory_all_bit[88] <= 12'h21C;
110         memory_all_bit[89] <= 12'h21C;
111         memory_all_bit[90] <= 12'h21C;
112         memory_all_bit[91] <= 12'h21C;
113         memory_all_bit[92] <= 12'h21C;
114         memory_all_bit[93] <= 12'h21C;
```

```
115         memory_all_bit[94] <= 12'h21D;
116         memory_all_bit[95] <= 12'h21D;
117         memory_all_bit[96] <= 12'h21D;
118         memory_all_bit[97] <= 12'h21C;
119         memory_all_bit[98] <= 12'h204;
120         memory_all_bit[99] <= 12'h204;
121         memory_all_bit[100] <= 12'h200;
122         memory_all_bit[101] <= 12'h200;
123         memory_all_bit[102] <= 12'h200;
124         memory_all_bit[103] <= 12'h240;
125         memory_all_bit[104] <= 12'h240;
126         memory_all_bit[105] <= 12'h240;
127         memory_all_bit[106] <= 12'h240;
128         memory_all_bit[107] <= 12'h200;
129         memory_all_bit[108] <= 12'h200;
130         memory_all_bit[109] <= 12'h200;
131         memory_all_bit[110] <= 12'h200;
132         memory_all_bit[111] <= 12'h200;
133
134         memory_all_bit[112] <= 12'h51C; //loop stop, return to 28
135         //shutter
136         memory_all_bit[113] <= 12'h200;
137         memory_all_bit[114] <= 12'h200;
138         memory_all_bit[115] <= 12'h200;
139         memory_all_bit[116] <= 12'h280;
140         memory_all_bit[117] <= 12'h280;
141         memory_all_bit[118] <= 12'h280;
142         memory_all_bit[119] <= 12'h280;
143         memory_all_bit[120] <= 12'h200;
144         memory_all_bit[121] <= 12'h200;
145         memory_all_bit[122] <= 12'h200;
146         memory_all_bit[123] <= 12'h200;
147         memory_all_bit[124] <= 12'h220;
148         memory_all_bit[125] <= 12'h220;
149         memory_all_bit[126] <= 12'h230;
150         memory_all_bit[127] <= 12'h230;
151         memory_all_bit[128] <= 12'h230;
152         memory_all_bit[129] <= 12'h230;
153         memory_all_bit[130] <= 12'h230;
154         memory_all_bit[131] <= 12'h230;
155         memory_all_bit[132] <= 12'h230;
156         memory_all_bit[133] <= 12'h220;
157         memory_all_bit[134] <= 12'h220;
158         memory_all_bit[135] <= 12'h200;
159         memory_all_bit[136] <= 12'h200;
160         memory_all_bit[137] <= 12'h200;
161         memory_all_bit[138] <= 12'h200;
162         memory_all_bit[139] <= 12'h200;
163         memory_all_bit[140] <= 12'h200;
164         memory_all_bit[141] <= 12'h200;
165         memory_all_bit[142] <= 12'h200;
166         memory_all_bit[143] <= 12'h200;
167         memory_all_bit[144] <= 12'h200;
168         memory_all_bit[145] <= 12'h200;
169         memory_all_bit[146] <= 12'h200;
170         memory_all_bit[147] <= 12'h200;
171         memory_all_bit[148] <= 12'h200;
172         memory_all_bit[149] <= 12'h200;
173         memory_all_bit[150] <= 12'h200;
174         memory_all_bit[151] <= 12'h200;
175         memory_all_bit[152] <= 12'h200;
176         memory_all_bit[153] <= 12'h200;
177         memory_all_bit[154] <= 12'h200;
178         memory_all_bit[155] <= 12'h200;
179
180     end
181
182     always @ (posedge fast_clk) begin
183         data <= memory_all_bit[addr];
184
```

```
185         if(wr)
186             memory_all_bit[wr_addr] <= wr_data;
187         if(rd) begin
188             rd_data <= memory_all_bit[i];
189             if(i==254)
190                 i = 0;
191             else
192                 i = i + 1;
193         end
194     end
195 endmodule
```

Listing A.1: all bit solution

A.2 change bit solution

Listing A.2: change bit solution

```
1  module memory(
2      input logic fast_clk,
3      input logic wr,
4      input logic rd,
5      input logic [7:0] wr_data,
6      input logic [7:0] addr,
7      input logic [7:0] wr_addr,
8      output logic [7:0] rd_data,
9      output logic [7:0] data
10     );
11
12     integer i;
13      reg [7:0] memory_all_bit [254:0];
14
15     initial begin
16         i <= 0;
17         //saved
18         memory_all_bit[0] <= 8'h43; //loop start part 1
19         memory_all_bit[1] <= 8'h50; //loop start part 2
20         //setup
21         memory_all_bit[2] <= 8'h30;
22         memory_all_bit[3] <= 8'h26;
23         memory_all_bit[4] <= 8'h30;
24         memory_all_bit[5] <= 8'h30;
25         memory_all_bit[6] <= 8'h30;
26         memory_all_bit[7] <= 8'h26;
27         memory_all_bit[8] <= 8'h30;
28         memory_all_bit[9] <= 8'h30;
29         memory_all_bit[10] <= 8'h27;
30         memory_all_bit[11] <= 8'h30;
31         memory_all_bit[12] <= 8'h30;
32         memory_all_bit[13] <= 8'h30;
33         memory_all_bit[14] <= 8'h27;
34         memory_all_bit[15] <= 8'h30;
35         memory_all_bit[16] <= 8'h30;
36         memory_all_bit[17] <= 8'h30;
37         memory_all_bit[18] <= 8'h22;
38         memory_all_bit[19] <= 8'h30;
39         memory_all_bit[20] <= 8'h30;
40         memory_all_bit[21] <= 8'h30;
41         memory_all_bit[22] <= 8'h30;
42         memory_all_bit[23] <= 8'h30;
43         memory_all_bit[24] <= 8'h30;
44         memory_all_bit[25] <= 8'h30;
45         memory_all_bit[26] <= 8'h30;
46
47
```

```
48          memory_all_bit[27] <= 8'h43; //loop start part 1
49          memory_all_bit[28] <= 8'h50; //loop start part 2
50
51          //sample
52          memory_all_bit[29] <= 8'h30;
53          memory_all_bit[30] <= 8'h30;
54          memory_all_bit[31] <= 8'h22;
55          memory_all_bit[32] <= 8'h30;
56          memory_all_bit[33] <= 8'h30;
57          memory_all_bit[34] <= 8'h30;
58          memory_all_bit[35] <= 8'h30;
59          memory_all_bit[36] <= 8'h30;
60          memory_all_bit[37] <= 8'h30;
61          memory_all_bit[38] <= 8'h30;
62          memory_all_bit[39] <= 8'h30;
63          memory_all_bit[40] <= 8'h30;
64          memory_all_bit[41] <= 8'h30;
65          memory_all_bit[42] <= 8'h30;
66          memory_all_bit[43] <= 8'h22;
67          memory_all_bit[44] <= 8'h24;
68          memory_all_bit[45] <= 8'h30;
69          memory_all_bit[46] <= 8'h30;
70          memory_all_bit[47] <= 8'h30;
71          memory_all_bit[48] <= 8'h30;
72          memory_all_bit[49] <= 8'h30;
73          memory_all_bit[50] <= 8'h30;
74          memory_all_bit[51] <= 8'h30;
75          memory_all_bit[52] <= 8'h30;
76          memory_all_bit[53] <= 8'h30;
77          memory_all_bit[54] <= 8'h26;
78          memory_all_bit[55] <= 8'h30;
79          memory_all_bit[56] <= 8'h30;
80          memory_all_bit[57] <= 8'h26;
81          memory_all_bit[58] <= 8'h24;
82          memory_all_bit[59] <= 8'h30;
83          memory_all_bit[60] <= 8'h30;
84          memory_all_bit[61] <= 8'h30;
85          memory_all_bit[62] <= 8'h30;
86          memory_all_bit[63] <= 8'h30;
87          memory_all_bit[64] <= 8'h30;
88          memory_all_bit[65] <= 8'h30;
89          memory_all_bit[66] <= 8'h30;
90          memory_all_bit[67] <= 8'h30;
91          memory_all_bit[68] <= 8'h30;
92          memory_all_bit[69] <= 8'h30;
93          memory_all_bit[70] <= 8'h30;
94          memory_all_bit[71] <= 8'h30;
95          memory_all_bit[72] <= 8'h30;
96          memory_all_bit[73] <= 8'h30;
97          memory_all_bit[74] <= 8'h30;
98          memory_all_bit[75] <= 8'h30;
99          memory_all_bit[76] <= 8'h30;
100         memory_all_bit[77] <= 8'h30;
101         memory_all_bit[78] <= 8'h30;
102         memory_all_bit[79] <= 8'h30;
103         memory_all_bit[80] <= 8'h30;
104         memory_all_bit[81] <= 8'h30;
105         memory_all_bit[82] <= 8'h30;
106         memory_all_bit[83] <= 8'h30;
107         memory_all_bit[84] <= 8'h30;
108         memory_all_bit[85] <= 8'h30;
109         memory_all_bit[86] <= 8'h30;
110         memory_all_bit[87] <= 8'h30;
111         memory_all_bit[88] <= 8'h24;
112         memory_all_bit[89] <= 8'h30;
113         memory_all_bit[90] <= 8'h23;
114         memory_all_bit[91] <= 8'h30;
115         memory_all_bit[92] <= 8'h30;
116         memory_all_bit[93] <= 8'h30;
117         memory_all_bit[94] <= 8'h30;
```

```
118         memory_all_bit[95] <= 8'h30;
119         memory_all_bit[96] <= 8'h30;
120         memory_all_bit[97] <= 8'h27;
121         memory_all_bit[98] <= 8'h30;
122         memory_all_bit[99] <= 8'h30;
123         memory_all_bit[100] <= 8'h27;
124         memory_all_bit[101] <= 8'h13;
125         memory_all_bit[102] <= 8'h24;
126         memory_all_bit[103] <= 8'h30;
127         memory_all_bit[104] <= 8'h25;
128         memory_all_bit[105] <= 8'h30;
129         memory_all_bit[106] <= 8'h30;
130         memory_all_bit[107] <= 8'h21;
131         memory_all_bit[108] <= 8'h30;
132         memory_all_bit[109] <= 8'h30;
133         memory_all_bit[110] <= 8'h30;
134         memory_all_bit[111] <= 8'h21;
135         memory_all_bit[112] <= 8'h30;
136         memory_all_bit[113] <= 8'h30;
137         memory_all_bit[114] <= 8'h30;
138         memory_all_bit[115] <= 8'h30;
139
140         memory_all_bit[116] <= 8'h25;  //correction to enable loop
141         memory_all_bit[117] <= 8'h69;  //loop end part 1, addr 0x29
142         memory_all_bit[118] <= 8'h72;  //loop end part 2, addr 0x29
143         memory_all_bit[119] <= 8'h25;  //correction to go out of loop
144
145         //shutter
146         memory_all_bit[120] <= 8'h30;
147         memory_all_bit[121] <= 8'h30;
148         memory_all_bit[122] <= 8'h30;
149         memory_all_bit[123] <= 8'h20;
150         memory_all_bit[124] <= 8'h30;
151         memory_all_bit[125] <= 8'h30;
152         memory_all_bit[126] <= 8'h30;
153         memory_all_bit[127] <= 8'h20;
154         memory_all_bit[128] <= 8'h30;
155         memory_all_bit[129] <= 8'h30;
156         memory_all_bit[130] <= 8'h30;
157         memory_all_bit[131] <= 8'h22;
158         memory_all_bit[132] <= 8'h30;
159         memory_all_bit[133] <= 8'h23;
160         memory_all_bit[134] <= 8'h30;
161         memory_all_bit[135] <= 8'h30;
162         memory_all_bit[136] <= 8'h30;
163         memory_all_bit[137] <= 8'h30;
164         memory_all_bit[138] <= 8'h30;
165         memory_all_bit[139] <= 8'h30;
166         memory_all_bit[140] <= 8'h23;
167         memory_all_bit[141] <= 8'h30;
168         memory_all_bit[142] <= 8'h22;
169         memory_all_bit[143] <= 8'h30;
170         memory_all_bit[144] <= 8'h30;
171         memory_all_bit[145] <= 8'h30;
172         memory_all_bit[146] <= 8'h30;
173         memory_all_bit[147] <= 8'h30;
174         memory_all_bit[148] <= 8'h30;
175         memory_all_bit[149] <= 8'h30;
176         memory_all_bit[150] <= 8'h30;
177         memory_all_bit[151] <= 8'h30;
178         memory_all_bit[152] <= 8'h30;
179         memory_all_bit[153] <= 8'h30;
180         memory_all_bit[154] <= 8'h30;
181         memory_all_bit[155] <= 8'h30;
182         memory_all_bit[156] <= 8'h30;
183         memory_all_bit[157] <= 8'h30;
184         memory_all_bit[158] <= 8'h30;
185         memory_all_bit[159] <= 8'h30;
186         memory_all_bit[160] <= 8'h30;
187         memory_all_bit[161] <= 8'h30;
```

```
188         memory_all_bit[162] <= 8'h30;
189
190     end
191
192     always @ (posedge fast_clk) begin
193         data <= memory_all_bit[addr];
194
195         /*if(wr)
196             memory_all_bit[wr_addr] <= wr_data;
197         if(rd) begin
198             rd_data <= memory_all_bit[i];
199             if(i==254)
200                 i = 0;
201             else
202                 i = i + 1;
203         end */
204     end
205 endmodule
```

Listing A.2: change bit solution

A.3 change position

Listing A.3: change position solution

```
1  module memory(
2      input logic fast_clk,
3      input logic wr,
4      input logic rd,
5      input logic [11:0] wr_data,
6      input logic [7:0] addr,
7      input logic [7:0] wr_addr,
8      output logic [11:0] rd_data,
9      output logic [11:0] data
10     );
11
12     integer i;
13      reg [11:0] memory_change_pos [254:0];
14
15     initial begin
16         i <= 0;
17         //saved
18         memory_change_pos[0] <= 12'h100;
19         memory_change_pos[1] <= 12'h100;
20         //setup
21         memory_change_pos[2] <= 12'h316;
22         memory_change_pos[3] <= 12'h256;
23         memory_change_pos[4] <= 12'h287;
24         memory_change_pos[5] <= 12'h2C7;
25         memory_change_pos[6] <= 12'h202;
26         memory_change_pos[7] <= 12'h880; //end comand for waveform, resets counter
27
28         memory_change_pos[9] <= 12'h403;    //loop start, repeat= 3
29         //sample
30         memory_change_pos[10] <= 12'h225;
31         memory_change_pos[10] <= 12'h2D5;
32         memory_change_pos[10] <= 12'h2E3;
33         memory_change_pos[10] <= 12'h271;
34         memory_change_pos[10] <= 12'h2A1;
35         memory_change_pos[10] <= 12'h2B3;
36         memory_change_pos[10] <= 12'h100; //do nothing 16 clock periods
37         memory_change_pos[10] <= 12'h2D3;
38         memory_change_pos[10] <= 12'h2F4;
39         memory_change_pos[10] <= 12'h260;
40         memory_change_pos[10] <= 12'h290;
```

```
41          memory_change_pos[10] <= 12'h3A4;
42          memory_change_pos[10] <= 12'h2A3;
43          memory_change_pos[10] <= 12'h2C2;
44          memory_change_pos[10] <= 12'h2F6;
45          memory_change_pos[10] <= 12'h236;
46          memory_change_pos[10] <= 12'h870; //end command for waveform, resets counter
47
48          memory_change_pos[10] <= 12'hB02; //resets waveform for looping
49          memory_change_pos[10] <= 12'h50A; //loop end
50          memory_change_pos[10] <= 12'hB02; // cancel reset of waveform for looping
51          //shutter
52          memory_change_pos[10] <= 12'h237;
53          memory_change_pos[10] <= 12'h277;
54          memory_change_pos[10] <= 12'h2B5;
55          memory_change_pos[10] <= 12'h2D4;
56          memory_change_pos[10] <= 12'h244;
57          memory_change_pos[10] <= 12'h265;
58          memory_change_pos[10] <= 12'h100; //do nothing 16 clock cycles
59          memory_change_pos[10] <= 12'h840; //end command for waveform, resets counter
60
61      end
62
63      always @ (posedge fast_clk) begin
64          data <= memory_change_pos[addr];
65
66          if(wr)
67              memory_change_pos[wr_addr] <= wr_data;
68          if(rd) begin
69              rd_data <= memory_change_pos[i];
70              if(i==254)
71                  i = 0;
72              else
73                  i = i + 1;
74          end
75      end
76  endmodule
```

Listing A.3: change position solution

A.4 wave block solution

Listing A.4: wave block solution

```
1  module memory(
2      input logic fast_clk,
3      input logic wr,
4      input logic rd,
5      input logic [11:0] wr_data,
6      input logic [7:0] addr,
7      input logic [7:0] wr_addr,
8      output logic [11:0] rd_data,
9      output logic [11:0] data,
10     output logic [7:0] wave_data
11     );
12
13     integer i;
14      reg [11:0] InstMemory [254:0];
15      logic[3:0] wave_addr;
16
17      memory_wave memory_wave1(.fast_clk(fast_clk), .addr(wave_addr),
            .wave_data(wave_data));
18
19     initial begin
20
21         InstMemory[0] <= 12'h210;
```

```
22          InstMemory[1] <= 12'h403;       //Loop start
23          //setup
24          InstMemory[2] <= 12'h210;       //normal
25          InstMemory[3] <= 12'h242;       //normal
26          InstMemory[4] <= 12'h230;       //normal
27          InstMemory[5] <= 12'h241;       //normal
28          InstMemory[6] <= 12'h240;       //normal
29          InstMemory[7] <= 12'h293;       //normal
30
31          InstMemory[8] <= 12'h403;       //Loop start
32          //sample
33          InstMemory[9] <= 12'h223;
34          InstMemory[9] <= 12'h2B4;
35          InstMemory[9] <= 12'h213;
36          InstMemory[9] <= 12'h295;
37          InstMemory[9] <= 12'h236;
38          InstMemory[9] <= 12'h215;
39          InstMemory[9] <= 12'h2F3;
40          InstMemory[9] <= 12'h2F3;
41          InstMemory[9] <= 12'h225;
42          InstMemory[9] <= 12'h277;
43          InstMemory[9] <= 12'h238;
44          InstMemory[9] <= 12'h217;
45          InstMemory[9] <= 12'h223;
46          InstMemory[9] <= 12'h230;
47          InstMemory[9] <= 12'h242;
48          InstMemory[9] <= 12'h250;
49
50          InstMemory[9] <= 12'h509;  //loop end
51          //shutter
52          InstMemory[9] <= 12'h230;
53          InstMemory[9] <= 12'h241;
54          InstMemory[9] <= 12'h240;
55          InstMemory[9] <= 12'h229;
56          InstMemory[9] <= 12'h27A;
57          InstMemory[9] <= 12'h229;
58          InstMemory[9] <= 12'h2F0;
59          InstMemory[9] <= 12'h260;
60
61
62
63      end
64
65      always @ (posedge fast_clk) begin
66          data <= InstMemory[addr];
67          wave_addr <= InstMemory[addr][3:0];
68
69          if(wr)
70              memory_all_bit[wr_addr] <= wr_data;
71          if(rd) begin
72              rd_data <= memory_all_bit[i];
73              if(i==254)
74                  i = 0;
75              else
76                  i = i + 1;
77          end
78      end
79  endmodule
80
81
82  module memory_wave (
83      input logic fast_clk,
84      input logic[3:0] addr,
85      output logic[7:0] wave_data);
86
87      reg [7:0] WaveMemory [10:0];
88
89      initial begin
90          WaveMemory[0] <= 8'h00;
91          WaveMemory[1] <= 8'h80;
```

```
92              WaveMemory[2] <= 8'h40;
93              WaveMemory[3] <= 8'h04;
94              WaveMemory[4] <= 8'h24;
95              WaveMemory[5] <= 8'h0C;
96              WaveMemory[6] <= 8'h0E;
97              WaveMemory[7] <= 8'h1C;
98              WaveMemory[8] <= 8'h1D;
99              WaveMemory[9] <= 8'h20;
100             WaveMemory[10] <= 8'h30;
101
102         end
103
104         always @ (posedge fast_clk) begin
105
106             wave_data <= WaveMemory[addr];
107         end
108
109 endmodule
```

Listing A.4: wave block solution

Appendix B

Common modules

B.1 interface

Listing B.1: interface module

```
1  module interfaceA(
2      input logic fast_clk,
3      input logic [1:0] command,
4      input logic [7:0] data_in,
5      input logic [11:0] rd_data,
6      output logic [7:0] addr,
7      output logic wr,
8      output logic [11:0] wr_data,
9      output logic rd,
10     output logic [7:0] data_out
11     );
12
13     always_comb begin
14         data_out <= rd_data;
15     end
16
17     always@(posedge fast_clk) begin
18         if (command == 2'b00) begin      //write at availible address
19
20         end
21         else if (command == 2'b01) begin //write at this address
22             addr <= data_in;
23             wr_data <= 12'h000;
24             wr <= 1'b0;
25             rd <= 1'b0;
26         end
27         else if (command == 2'b10) begin //write this
28             wr_data <= data_in;
29             addr <= addr;
30             wr <= 1'b1;
31             rd <= 1'b0;
32         end
33         else if (command == 2'b11) begin //read
34             rd <= 1'b1;
35             wr <= 1'b0;
36             wr_data <= 12'h000;
37             addr <= addr;
38         end
39         else begin //error
40             rd <= 1'b0;
41             wr <= 1'b0;
42             addr <= 8'h00;
43             wr_data <= 12'h000;
44         end
```

```
45      end
46
47  endmodule
```

Listing B.1: interface module

B.2 next address module

Listing B.2: next address module

```
1  module nextAddress (
2      input logic rst_all,
3      input logic fast_clk,
4      input logic[7:0] D_jump,
5      input logic[7:0] L_jump,
6      input logic fetchEnable,
7      output logic[7:0] nxt_addr);
8
9      logic[7:0] P1;
10     reg [7:0] nxt_addr_temp;
11
12     programCounter programCounter1 (.rst_all(rst_all), .fast_clk(fast_clk),
           .current_address(nxt_addr), .P1(P1));
13
14     always @ (posedge fetchEnable) begin
15         if (rst_all) begin
16             nxt_addr_temp <= 1'b0;
17         end else begin
18             if (D_jump != 1'b0)
19                 nxt_addr_temp <= D_jump;
20             else if (L_jump != 1'b0)
21                 nxt_addr_temp <= L_jump;
22             else if (P1 != 1'b0)
23                 nxt_addr_temp <= P1;
24             else
25                 nxt_addr_temp <= 1'b0;
26         end
27     end
28
29     always @ (posedge fast_clk) begin
30         if(¬rst_all) begin
31             if(fetchEnable)
32                 nxt_addr <= nxt_addr_temp;
33             else
34                 nxt_addr <= nxt_addr;
35          end else
36             nxt_addr <= 1'b0;
37     end
38 endmodule
```

Listing B.2: next address module

B.3 data handle module

Listing B.3: data handle module

```
1  module data_handle(
2      input logic rst,
3      input logic[11:0] data,
4      input logic fast_clk,
```

```
5       output logic[7:0] wave,
6       output logic wait_enable,
7       output logic rst_command,
8       output logic[7:0] loop_addr,
9       output logic[7:0] loop_repeat,
10      output logic increase,
11      output logic jump_fetch,
12      output logic[7:0] D_jump
13      );
14
15      reg [3:0] state;
16
17      always_comb begin
18          if(rst)
19              state = 4'h0;
20          else
21              state = data[11:8];
22      end
23
24      always@(posedge fast_clk) begin
25          case (state)
26          4'h0: begin
27              wave = 8'h00;
28              wait_enable = 1'b0;
29              rst_command = 1'b0;
30              loop_addr = 8'h00;
31              loop_repeat = 8'h00;
32              increase = 1'b0;
33              jump_fetch = 1'b0;
34              D_jump = 8'h00;
35              end
36          4'h1: begin
37              wave = wave;
38              wait_enable = 1'b1;
39              rst_command = 1'b0;
40              loop_addr = loop_addr;
41              loop_repeat = loop_repeat;
42              increase = increase;
43              jump_fetch = jump_fetch;
44              D_jump = D_jump;
45              end
46          4'h2: begin
47              wave = data[7:0];
48              wait_enable = 1'b0;
49              rst_command = 1'b0;
50              loop_addr = loop_addr;
51              loop_repeat = loop_repeat;
52              increase = 1'b0;
53              jump_fetch = 1'b0;
54              D_jump = D_jump;
55              end
56          4'h3: begin
57              wave = wave;
58              wait_enable = wait_enable;
59              rst_command = 1'b1;
60              loop_addr = loop_addr;
61              loop_repeat = loop_repeat;
62              increase = increase;
63              jump_fetch = jump_fetch;
64              D_jump = D_jump;
65              end
66          4'h4: begin
67              loop_repeat = data[7:0];
68              wave = wave;
69              wait_enable = 1'b0;
70              rst_command = 1'b0;
71              loop_addr = loop_addr;
72              increase = 1'b0;
73              jump_fetch = 1'b0;
74              D_jump = D_jump;
```

```
75              end
76          4'h5: begin
77              loop_addr = data[7:0];
78              wave = wave;
79              wait_enable = 1'b0;
80              rst_command = 1'b0;
81              loop_repeat = loop_repeat;
82              increase = 1'b1;
83              jump_fetch = 1'b0;
84              D_jump = D_jump;
85              end
86           4'h6: begin
87              wave = wave;
88              wait_enable = 1'b0;
89              rst_command = 1'b0;
90              loop_addr = loop_addr;
91              loop_repeat = loop_repeat;
92              increase = 1'b0;
93              jump_fetch = 1'b1;
94              D_jump = data[7:0];
95              end
96           4'h7: begin
97              wave = wave;
98              wait_enable = 1'b0;
99              rst_command = 1'b0;
100             loop_addr = 8'h00;
101             loop_repeat = 8'h00;
102             increase = 1'b0;
103             jump_fetch = 1'b1;
104             D_jump = data[7:0];
105             end
106          default: begin
107             wave = data[7:0];
108             wait_enable = 1'b0;
109             rst_command = 1'b0;
110             loop_addr = 8'h00;
111             loop_repeat = 1'b0;
112             increase = 1'b0;
113             jump_fetch = 1'b0;
114             D_jump = data[7:0];
115             end
116         endcase
117     end
118 endmodule
```

Listing B.3: data handle module

B.4 Loop module

Listing B.4: loop module

```
1  module loop(
2      input logic fast_clk,
3      input logic[7:0] loop_addr,
4      input logic[7:0] loop_repeat,
5      input logic increase,
6      input logic rst,
7      output logic loop_fetch,
8      output logic[7:0] L_jump);
9
10     reg [1:0] state;
11     reg [7:0] count;
12
13     always @(fast_clk) begin
14         if(rst)
```

```
15              state <= 2'b00;
16          else begin
17          case(state)
18          2'b00:
19              state <= 2'b01;
20          2'b01: begin
21              if(increase) begin
22                  if(count < loop_repeat)
23                      state <= 2'b10;
24                  else
25                      state <= 2'b11;
26              end else
27                  state <= 2'b01;
28              end
29          2'b10:
30              state <= 2'b01;
31          2'b11:
32              state <= 2'b01;
33          endcase
34          end
35      end
36
37      always @(state) begin
38          case(state)
39          2'b00: begin
40              count <= 8'h01;
41              L_jump <= 8'h00;
42              loop_fetch <= 1'b0;
43              end
44          2'b01: begin
45              count <= count;
46              L_jump <= 8'h00;
47              loop_fetch <= 1'b0;
48              end
49          2'b10: begin
50              count <= count + 1'b1;
51              L_jump <= loop_addr;
52              loop_fetch <= 1'b1;
53              end
54          2'b11: begin
55              count <= 8'h01;
56              L_jump <= 8'h00;
57              loop_fetch <= 1'b1;
58              end
59          endcase
60      end
61
62  endmodule
```

Listing B.4: loop module

B.5 Delay module

Listing B.5: delay module

```
1  module delay(
2      input logic [7:0] wave,
3      input logic fast_clk,
4      input logic main_clk,
5      input logic clk_cycle,
6      input logic rst,
7      output logic [8:0] out_wave,
8      output logic delay_fetch
9      );
10
```

```
11      reg [1:0] state;
12
13      always@(state) begin
14          case (state)
15              2'b00: begin
16                 out_wave = 8'h00;
17                 delay_fetch = 1'b0;
18                 end
19              2'b01: begin
20                 out_wave = wave;
21                 delay_fetch = 1'b1;
22                 end
23              2'b10: begin
24                 out_wave = out_wave;
25                 delay_fetch = 1'b0;
26                 end
27              default: begin
28                 out_wave = 8'h00;
29                 delay_fetch = 1'b0;
30                 end
31          endcase
32      end
33
34      always @ (posedge fast_clk) begin
35          if (rst)
36              state <= 2'b00;
37          else
38          case (state)
39             2'b00:
40                state <= 2'b01;
41             2'b01:
42                state <= 2'b10;
43             2'b10:
44                if (clk_cycle==0)
45                   state <= 2'b10;
46                else
47                   state <= 2'b01;
48          endcase
49          end
50  endmodule
```

Listing B.5: delay module

Bibliography

[AAE06] Amara Amara, Frédéric Amiel, and Thomas Ea. *FPGA vs. ASIC for low power applications.* Microelectronics Journal, 21 rue d'Assas, 75006 Paris, France, 2006.

[CDHK10] Eduard Cerny, Surrendra Dudani, John Havlicek, and Dmitry Korchemny. *The Power of Assertions in SystemVerilog.* Springer, Boston, MA, 2010.

[CJLZ11] Jason Cong, Wei Jiang, Bin Liu, and Yi Zou. *Automatic memory partitioning and scheduling for throughput and power optimization.* ACM Transactions on Design Automation of Electronic Systems, California, Los Angeles, 2011.

[Fos21] Maren Vorin Fossum. *Custimisable waveform generator.* NTNU, Trondheim, Norway, 2021.

[Hau98] Scott Hauck. The roles of fpgas in reprogrammable systems. In *Proceedings of IEEE Volume 86*, pages 615 – 638, Evanston, IL, USA, 1998. IEEE Computer Society.

[HH07] David Money Harris and Sarah L. Harris. *Digital Design and Computer Architecture, ARM edition.* Morgan Kaufmann Publishers In, 2007.

[KR06] Ian Kuon and Jonathan Rose. *Measuring the Gap Between FPGAs and ASICs.* IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Toronto, Canada, 2006.

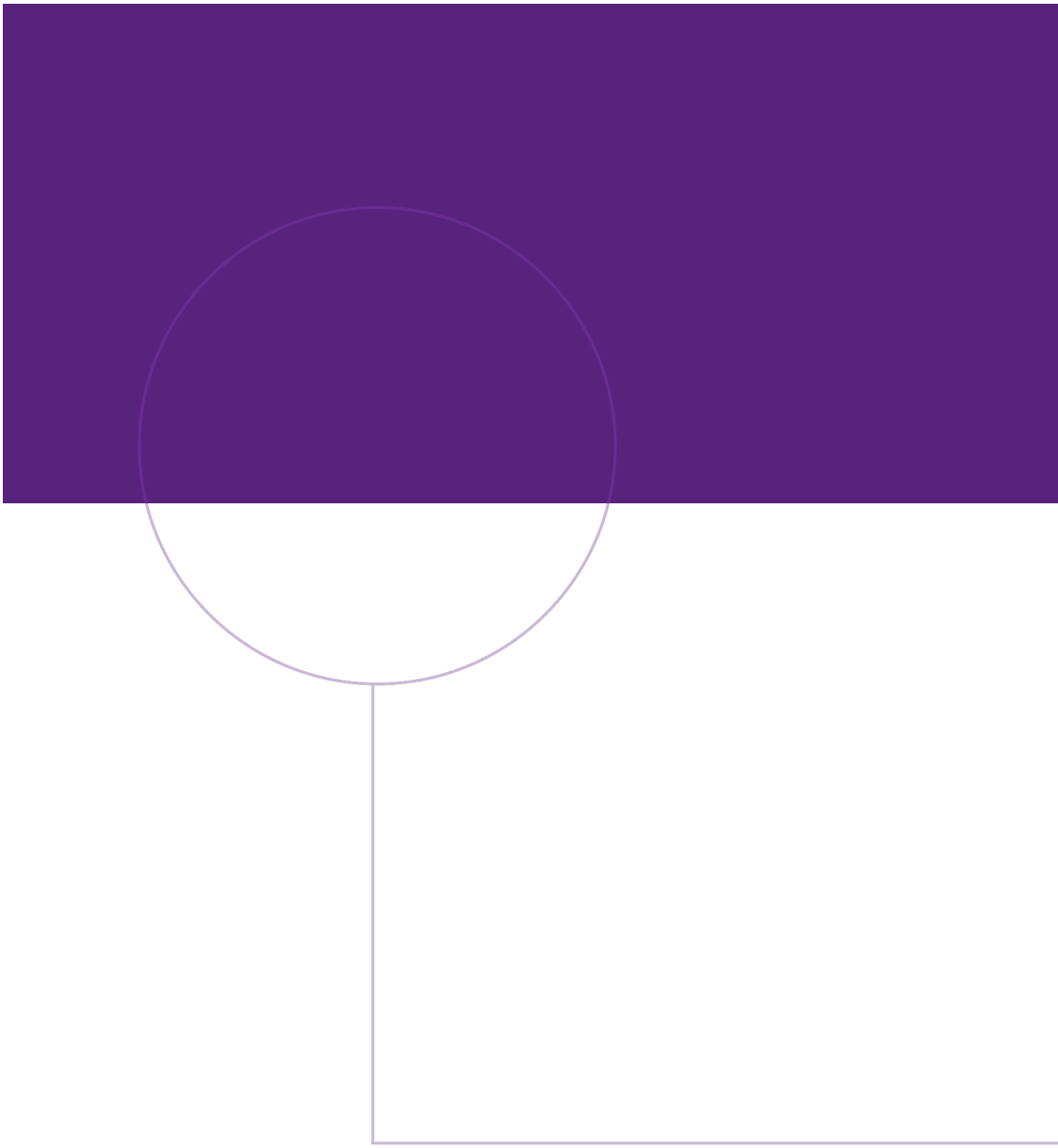
[LHC+05] R. Leupers, M. Hohenauer, J. Ceng, H. Scharwaechter, H. Meyr, G. Ascheid, and G. Braun. *Retargetable compilers and architecture exploration for embedded processors.* IEE Proc.-Comput. Digit. Tech., Vol. 152, CA, USA, 2005.

[Moh12] Khaled Salah Mohamed. *IP Cores Design from Specifications to Production. Modeling, Verification, Optimization, and Protection.* Springer, Cham, 2012.

[SP21] Mohammed Saifuddin and Jenny Picalausa. Imaging and sensing technology group direction. From Sony official webcite and conversations with engenering employers, August 2021. https://www.sony-semicon.co.jp/e/technology/direction/.

[Tys09] Jeff Tyson. *How ROM works.* UNIKOM, Perpustakaan, Indonesia, 2009.

[Xil] Inc Xilinx. Xilinx accelerates productivity for zynq-7000 all programmable socs with the vivado design suite 2014.3, sdk, and new ultrafast embedded design methodology guide. From Xilinx official. https://www.design-reuse.com/news/35626/xilinx-zynq-7000-vivado-design-suite-2014-3.html.



NTNU

Norwegian University of
Science and Technology