Notes

Implementation of RISC-V in SME

Daniel Ramyar

February 29, 2020

Contents

Chapter 1

Introduction

Add motivation behind whole project

Why RISK-V?, why RISCK-V in physics?, why SME?

Chapter 2

Logic Design

This chapter aims to introduce the reader to the basics of logic design, which will be imperative to the understanding the subsequent chapters. The general structure of this chapter will be based on Appendix A in [3].

We will begin in Section 2.1 by introducing the fundamental algebra and the physical building blocks, used to implement the algebra, such as the OR gate.

Hereafter we will be using these building blocks to design and create the core components used in the RISC-V architecture such as the decoder and multiplexer in section 2.2.

2.1 Boolean algebra

The fundamental tool used in logic design is a branch of mathematical logic called Boolean algebra. Compared to elementary algebra, where we deal with variables which represents some real or complex number, in Boolean algebra the variables are viewed as statements or propositions, which are either *true* or *false*.

In addition to the variables in elementary algebra we also had a means of manipulating them. These manipulations are called operations which operates on the variables (operands) where the basic operators of algebra consists of $+$, $-$, $\times$ and $\div$.

In Boolean algebra we have a distinction between operators which work on one operand and the ones that work on to two operands. These are called unary and binary operators respectively. We will go through a description of these in the following section.

2.1.1 Unary operators

With a single binary operand p we have 2 possible input *true* and $false$. All output combinations are summarized in Table 2.1. Each numbered column here represents an unnamed operator. We will go ahead and describe one of these in the following. The rest can referred to in Appendix A.

make ven diagrams to show the operator function

p	1	2	3	4
$true$	$true$	$true$	$false$	$false$
$false$	$true$	$false$	$true$	$false$

Table 2.1: Logic table of possible unary operators. Each numbered column represents an unnamed operator.

Logical complement

For our first basic Boolean operator we have the logical complement operator, which is represented by NOT, !, $\neg$ or $\bar{x}$ in various literature and commonly referred to as the negation operator.

The negation operator inverts an operand such that $\neg true = false$ and $\neg false = true$. Using a table we can neatly represent the complete function of the negation operator. These tables are called *logic tables*.

A logic table has been created for the negation operator as can be seen in Table 2.2. The first column represents our proposition and all its possible arguments $true$ and $false$. The second column is then the negated proposition.

p	$\neg p$
$true$	$false$
$false$	$true$

Table 2.2: Logic table of the negation operator where the proposition p, which is either true or false, can be found in the first column. In the second column we find $\neg p$, which is read as NOT p, and its return values.

Summary

We can now go ahead and fill the numbered columns Table 2.1 with the corresponding operators, which we have defined throughout this section and Appendix A. The filled table can be found in Table 2.3.

p	$T(p)$	$I(p)$	$\neg p$	$F(p)$
$true$	$true$	$true$	$false$	$false$
$false$	$true$	$false$	$true$	$false$

Table 2.3: Logic table of possible unary operators where p is our proposition. Column 2-5 shows the output of the corresponding operator.

2.1.2 Binary operators and disjunctive normal form

With two binary operands, p and q, there exist four possible combinations between their respectable values namely $(true, true)$, $(true, false)$, $(false, true)$, $(false, false)$.

Compared to the previous section we now have 4 possible input values for our yet unnamed operators $X(p, q)$. There exist 16 unique sets of outputs and therefore 16 possible operators. An example of a set of outputs could be

$$X(p, q) = \{true, false, false, false\} \tag{2.1}$$

where $(p, q) = \{(true, true), (true, false), (false, true), (false, false)\}$ is the set of possible inputs.

All output sets are summarized in Table 2.4 where each numbered column represents an unnamed operator.

We will in this section start by defining the basic operators from which we will derive the rest. For brevity we will only go through the 3 most commonly used operators, the rest can be referred to in Appendix B.

The choice of basic operators is arbitrary but I have chosen the operators for which it is the easiest to derive all other operators, since there exists a method to convert any truth table into a Boolean expression using these which we will get into later.

p	q	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
t	t	t	t	t	t	f	t	f	f	t	t	f	t	f	f	f	f
t	f	t	t	t	f	t	t	t	f	f	f	t	f	t	f	f	f
f	t	t	t	f	t	t	f	t	t	f	t	f	f	f	t	f	f
f	f	t	f	t	t	t	f	f	t	t	f	t	f	f	f	t	f

Table 2.4: Logic table of possible binary operators where $t = true$ and $f = false$. Each numbered column represents an unnamed operator.

Logical conjunction

The logical conjunction operator is represented by $\wedge$ in mathematics; AND, &, && in computer science and a $\cdot$ in electronic engineering and commonly referred to as the AND operator or the logical product. The AND operator only results in a true value if both of the operands are true.

Using Table 2.4 we see that the set of outputs which corresponds to this definition is column 12 and is summarized in Table 2.5.

Here we have the propositions p and q in the first two columns and all possible permutations between them in the following rows. The last column then shows the resulting value after doing the AND operation between p and q.

p	q	$p \wedge q$
$true$	$true$	$true$
$true$	$false$	$false$
$false$	$true$	$false$
$false$	$false$	$false$

Table 2.5: Logic table of the AND operator where p is the first proposition and q is the second. All possible permutations are then specified in each row for each proposition. The third column then shows the resulting value of the AND operation between p and q.

Logical disjunction

The logical disjunction operator is represented by $\vee$ in mathematics; OR, |, || in computer science and a + in electronic engineering and commonly referred to as the OR operator or the logical sum. The OR operator results in a true value if one or more of the operands are true.

Using Table 2.4 we see that the set of outputs which corresponds to this definition is column 2 and is summarized in Table 2.6.

Here we have the propositions p and q in the first two columns and all possible permutations between them in the following rows. The last column then shows the resulting value after doing the OR operation between p and q.

p	q	$p \vee q$
$true$	$true$	$true$
$true$	$false$	$true$
$false$	$true$	$true$
$false$	$false$	$false$

Table 2.6: Logic table of the OR operator where p is the first proposition and q is the second. All possible permutations are then specified in each row for each proposition. The third column then shows the resulting value of the OR operation between p and q.

We choose AND, OR and NOT to form our basic or primitive operators from which we will derive all remaining operators.

Exclusive disjunction and disjunctive normal form

The exclusive disjunction is represented by $\veebar$ in mathematics or XOR, ^ in computer science and commonly referred to as the XOR or exclusive OR operator. The XOR operator results in a true value only if the operands differ.

Using Table 2.4 we see that the set of outputs which corresponds to this definition is column 7 and is summarized in Table 2.7.

Here we have the propositions p and q in the first two columns and all possible permuta-

tions between them in the following rows. The last column then shows the resulting value after doing the XOR operation between p and q.

p	q	$p \veebar q$
$true$	$true$	$false$
$true$	$false$	$true$
$false$	$true$	$true$
$false$	$false$	$false$

Table 2.7: Logic table of the XOR operator where p is the first proposition and q is the second. All possible permutations are then specified in each row for each proposition. The third column then shows the resulting value of the XOR operation between p and q.

We can define this operator in *disjunctive normal form* (DNF) or *Sum of products form*, using our basic operators AND, OR and NOT. A logic equation (see section 2.1.3) is said to be in DNF, when it consists of disjunctions between one or more conjunctions, where each of the propositions can be complemented.

To write our operator in DNF, we first identify all true outputs in 2.7 namely row 3 and 4. We then take a look at the corresponding input values

$$(p, q) = (true, false) \quad \text{and} \quad (p, q) = (false, true) \tag{2.2}$$

and applying the NOT operator on all the false values. We now have the two tuples

$$(p, \neg q) = (true, \neg false) \quad \text{and} \quad (\neg p, q) = (\neg false, true). \tag{2.3}$$

Hereafter we apply the AND operator between the values in each tuple of input such that

$$p \wedge \neg q = true \wedge \neg false \quad \text{and} \quad \neg p \wedge q = \neg false \wedge true. \tag{2.4}$$

Lastly we apply the OR operators between each tuple and we have the final expression for XOR in terms of the basic operators

$$p \veebar q = (p \wedge \neg q) \vee (\neg p \wedge q). \tag{2.5}$$

The procedure is summarized as follows

1. Find all output values, which are true.
2. Negate all false input for corresponding true output value.
3. Apply AND operator between each value in each input tuple.
4. Lastly apply OR operator between each input tuple.

Using this procedure any logic table can be expressed as a Boolean expression and will be used extensively throughout this thesis.

Summary

We can now go ahead and fill the numbered columns in Table 2.4 with the corresponding operators, which we have defined throughout this section and Appendix B. The filled table can be found in Table 2.8.

p	q	$\top$	$\vee$	$\leftarrow$	$\rightarrow$	$\uparrow$	$P(p,q)$	$\veebar$	$\neg P(p,q)$	$\leftrightarrow$	$Q(p,q)$	$\neg Q(p,q)$	$\wedge$	$\not\rightarrow$	$\not\leftarrow$	$\downarrow$	$\bot$
t	t	t	t	t	t	f	t	f	f	t	t	f	t	f	f	f	f
t	f	t	t	t	f	t	t	t	f	f	f	t	f	t	f	f	f
f	t	t	t	f	t	t	f	t	t	f	t	f	f	f	t	f	f
f	f	t	f	t	t	t	f	f	t	t	f	t	f	f	f	t	f

Table 2.8: Logic table of binary operators where $t = true$ and $f = false$.

2.1.3 Boolean equations

In Section 2.1.2, we saw that it was possible to describe any logic table in terms of the AND, OR and Negation operators. An example of this could be the following

$$p \veebar q = (p \wedge \neg q) \vee (\neg p \wedge q) \tag{2.6}$$

where p and q was our propositions. Expression 2.6 is an example of a *Boolean equation*.

Like ordinary algebra, Boolean equations satisfy many of the same basic laws of algebra as summarized in Table 2.9. Here we see that the laws are exactly equivalent to the version we see with ordinary addition and multiplication, hence the names logical sum $\vee$ and logical product $\wedge$.

Using these laws we can drastically simplify complex expressions which we will use later to greatly reduce the complexity of logic units.

Say we have

$$C = A \cdot \bar{B} \cdot \bar{S} + A \cdot B \cdot \bar{S} + \bar{A} \cdot B \cdot S + A \cdot B \cdot S \tag{2.7}$$

where A, B, C and S are Boolean variables. Notice that $\cdot = \wedge$ and $+ = \vee$, we change to this notation since I find it much easier to discern the individual terms. Now we can use the distributivity law we found in Table 2.9 to pull $A \cdot \bar{S}$ and $B \cdot S$ outside the parentheses

$$C = (\bar{B} + B) \cdot A \cdot \bar{S} + (\bar{A} + A) \cdot B \cdot S. \tag{2.8}$$

Lastly we use the complement law in Table 2.10 ($\bar{B} + B = 1$ and $\bar{A} + A = 1$) and the identity law in Table 2.9 ($1 \cdot A \cdot \bar{S} = A \cdot \bar{S}$ and $1 \cdot B \cdot S = B \cdot S$) to simplify such that we have

$$C = A \cdot \bar{S} + B \cdot S. \tag{2.9}$$

Notice that we went from using 11 operations in (2.7) to 3 in (2.9) by using the Boolean laws to manipulate the equations, this reduces the complexity of an eventual implementation of the logic equation. Incidentally (2.7) is an example of a multiplexer which we will get into later.

Law	Law of $\vee$	law of $\wedge$
Commutativity	$p \vee q = q \vee p$	$p \wedge q = q \wedge p$
Associativity	$p \vee (q \vee r) = (p \vee q) \vee r$	$p \wedge (q \wedge r) = (p \wedge q) \wedge r$
Distributivity	$p \wedge (q \vee r) = (p \wedge q) \vee (p \wedge r)$	
Identity	$p \vee 0 = p$	$p \wedge 1 = p$
Zero law		$p \wedge 0 = 0$

Table 2.9: Basic Boolean laws. These laws satisfy both Boolean and ordinary algebra.

Law	Law of $\vee$	law of $\wedge$
Distributivity		$p \vee (q \wedge r) = (p \vee q) \wedge (p \vee r)$
One law	$p \vee 1 = 1$	
Idempotence law	$p \vee p = p$	$p \wedge p = p$
Absorption law	$x \vee (x \wedge y) = x$	$x \wedge (x \vee y) = x$
Complement law	$p \vee \neg p = 1$	$p \wedge \neg p = 0$
De Morgan Laws	$\neg p \vee \neg q = \neg(p \wedge q)$	$\neg p \wedge \neg q = \neg(p \vee q)$

Table 2.10: Basic Boolean laws. These laws do not have an equivalent in ordinary algebra.

2.1.4 Gates

In this and following sections the physical abstractions to the propositions *true* and $false$ will be represented by a voltage either being high or low. When the voltage is high we say that the signal is *asserted* and represented by 1 and when the voltage low it is *deasserted* and represented by 0.

We will use 3 fundamental physical components, *gates*, to implement logic tables or Boolean equations and each of these is represented by a symbol which we will go through in the following.

It should be noted that multiple input are possible with the AND and OR gates since they are both commutative and associative. There will always be 1 output, which is the result of all the subsequent inputs e.g. $A + B + C = D$ here the three input A, B, C would go into a single OR gate and return a single output D.

AND Gate

The AND gate is the physical implementation of logic Table 2.5 we defined earlier. It is illustrated by the symbol found in Figure 2.1.

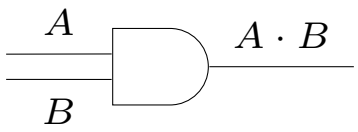


Figure 2.1: Illustration of the AND gate where A and B are the input and $A \cdot B$ is the output.

OR Gate

The OR gate is the physical implementation of logic Table 2.6 we defined earlier. It is illustrated by the symbol found in Figure 2.2.

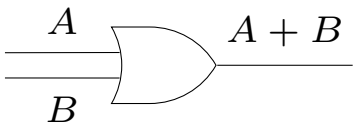


Figure 2.2: Illustration of the OR gate where A and B are the input and $A + B$ is the output.

NOT Gate

The NOT gate or inverter is the physical implementation of logic table 2.2 we defined earlier. It is illustrated by the symbol found in Figure 2.3. Usually the inverter is not drawn explicitly, but rather a "bubble" is drawn at the input or output of the respective gate, as shown in figure 2.4.

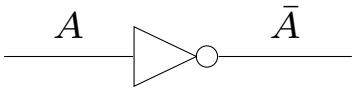


Figure 2.3: Illustration of the NOT gate where A and B are the input and $A + B$ is the output.

2.2 Combinational logic

When we design logic units which contain no memory i.e always return the same output given same input, we deal with *combinational logic*. In this section we will go through the essential combinational logic units that will be used throughout this thesis.

(a) (b)

Figure 2.4: (a) illustrates the inverter explicitly drawn before the input to the AND gate. (b) shows the inverter illustrated as a bubble before the input to the AND gate.

2.2.1 Decoder

The first combinational logic unit we will take a look at will be the *decoder*. Its function is to select one of multiple outputs to assert. This selection is determined by the inputs.

Say that we have 3 inputs i.e 3 bits of information. There are 8 possible configurations of these 3 bits ($2^3 = 8$) and for each configuration we assign one output to be asserted.

In Table 2.11 we have for each configuration asserted one output. Notice that we have used the binary representation of a decimal number to determine which output should be asserted for given input configuration. For example the binary representation for the decimal number 5 is 101, so when the input is $In2 = 1$, $In1 = 0$ and $In0 = 1$ output 5 is asserted.

It should be noted that the choice of which output that should get asserted for given input is arbitrary and up to the logic designer to decide, though each input configuration must only assert one unique output.

In this example we had 3 input, but we can generalize the decoder such that for n input, where $n > 0$, we have 2^n output. Only one output is asserted per input configuration.

Input			**Output**							
In2	In1	In0	Out7	Out6	Out5	Out4	Out3	Out2	Out1	Out0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

Table 2.11: Logic Table of a 3 input decoder where the binary representation of the input determines which output gets asserted. For example when In2=1, In1=0, In0=1 output 5 will get asserted as the binary representation for 5 is 101.

2.2.2 Multiplexer

When we will later deal with larger systems consisting of multiple logic units, we will need a way to select from which unit we want the output to go further up the chain. This select

unit is known as a *multiplexer* or *mux*. Its function is to select one of multiple input and output the selected input unchanged.

In Table 2.12 we have constructed a multiplexer with three input one of which is the control signal S. If the control signal is asserted $S = 1$ the output will have the value of B and if deasserted $S = 0$ it will output the value of A.

In this example we only had two input, but the multiplexer can be made such that it can select between arbitrary many input though this requires an increase in control signals. For n control signals we are able to select between 2^n input, where $n > 0$.

A	**B**	**S**	**C**
0	0	0	0
0	1	0	0
1	0	0	1
1	1	0	1
0	0	1	0
0	1	1	1
1	0	1	0
1	1	1	1

Table 2.12: Logic Table of a multiplexer.

2.2.3 Two-level logic

We saw in Section 2.1.2 that it was possible to express any logic table into a logic equation expressed as a sum of one or more products, also known as *disjunctive normal form* or *Sum of Products*. As we will see shortly this type of logic expression can be implemented using only two levels of logic, one layer consisting only of AND gates and one only of OR Gates, where negations are only applied to individual variables.

In this and next section we will see an example how one would implement various logic units, such as the multiplexer, going from logic table to the sum of products logic equation and lastly generating a gate-level implementation.

Going ahead we will implement the two input multiplexer starting by writing the logic table found in 2.12 in sum of products form. Using the approach mentioned in 2.1.2 we end up with the logic equation for the multiplexer

$$C = A \cdot \bar{B} \cdot \bar{S} + A \cdot B \cdot \bar{S} + \bar{A} \cdot B \cdot S + A \cdot B \cdot S. \tag{2.10}$$

We already saw how one could drastically simplify this expression in Section 2.1.3, such that we end up with

$$C = A \cdot \bar{S} + B \cdot S. \tag{2.11}$$

Now we have the simplified two-level representation for the two input multiplexer, in the next section we will se how this is used to generate the gate-level implementation.

2.2.4 Programmable logic array

A common two-level logic device used to implement logic equations is the *programmable logic array* or PLA for short. The PLA consists of two lines per input, one unaltered and one complemented (negated input), which are then connected to a plane of AND gates. The connections between the inputs and AND plane of course depends on the logic equation, which is to be implemented. Hereafter the outputs of the AND plane connects to the OR plane and again the connections depends on the logic equation.

Using this logic we can go ahead and implement 2.11. Looking at the equation we see that we need to perform two AND operations and one OR operation. We therefore need two AND gates and one OR gate. Since the PLA has lines for both input and inverted input we only need to connect the correct lines to the corresponding gates. This is done in figure 2.5a where the black dots show which lines are connected to which gate. When designing larger logic circuits it is more common to omit drawing all gates explicitly, which is illustrated in 2.5b.

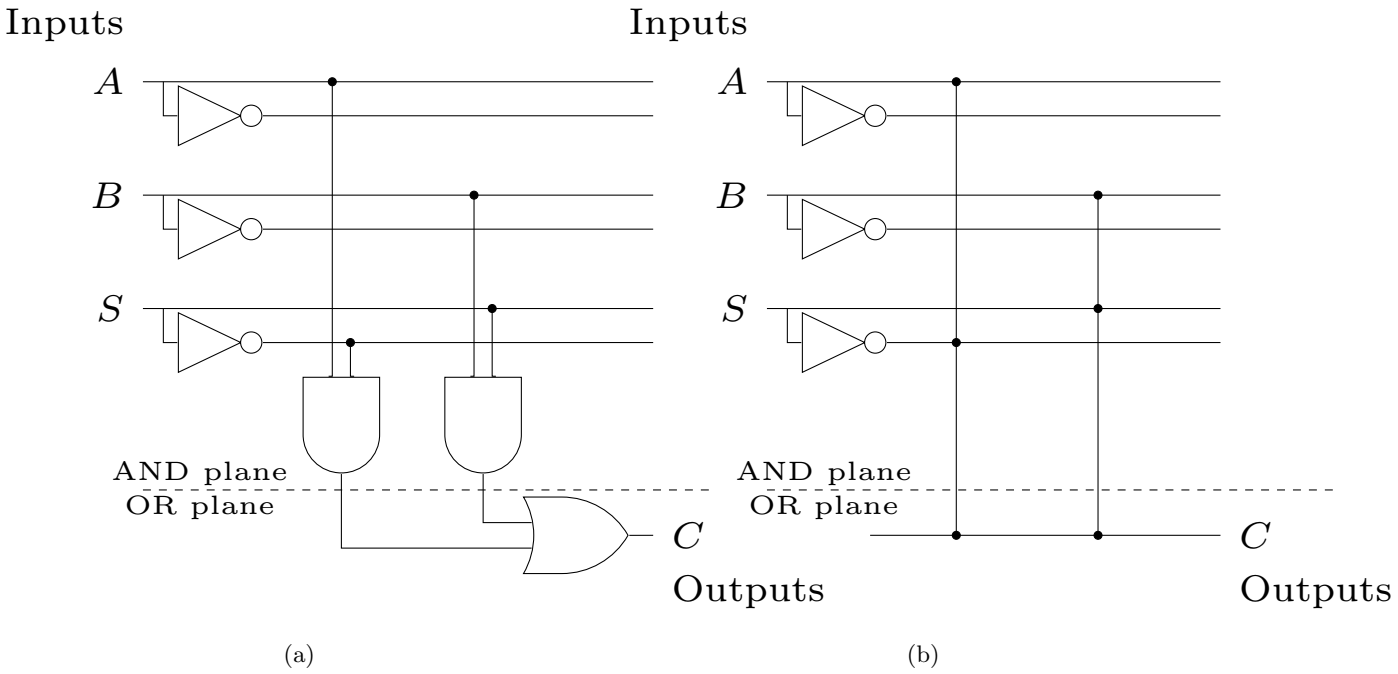


Figure 2.5: Both of these circuits show the PLA implementation of the logic equation we saw in 2.11. (a) Illustrates the multiplexer with explicitly drawn gates where the black dots indicates a connection. (b) Illustrates the multiplexer with implicitly drawn gates where each vertical line in the AND plane represents a connection to an AND gate and each horizontal line in the OR plane represents a connection to an OR gate. As before the black dots shows which lines are connected.

Chapter 3

Synchronous Message Exchange

In the previous chapter we went through the theory behind logic design, all the way from the introduction of Boolean algebra to gate-level implementation of logic units. In this chapter we will use those ideas to implement logic units in synchronous message exchange (SME). This will serve as a means to introduce SME to the reader and further cement former logic design ideas.

Starting out we will go through a short introduction behind the inspiration for SME, Communicating Sequential Processes. Hereafter we will go through the theory and syntax of SME with an emphasis on real examples, as I believe this is the most efficient manner of introducing SME to the reader.

3.1 Communicating Sequential Processes

When working with multiprocessor workloads, you will quickly realize the inconvenience of memory sharing. The non-determinism of having multiple processes reading and writing the same memory often results in unexpected behavior.

A classic example of non-determinism would be the act of printing out the numbers one through ten, to your console using more than one thread. If the aforementioned code is executed multiple times, you will notice that the order of numbers would vary between the runs. This is due to the scheduler of the operating system, which we do not have control over. That can create race conditions (meaning the behavior in your code is dependent on the timing of different threads), which can cause unpredictable behavior and therefore bugs, which is undesirable.

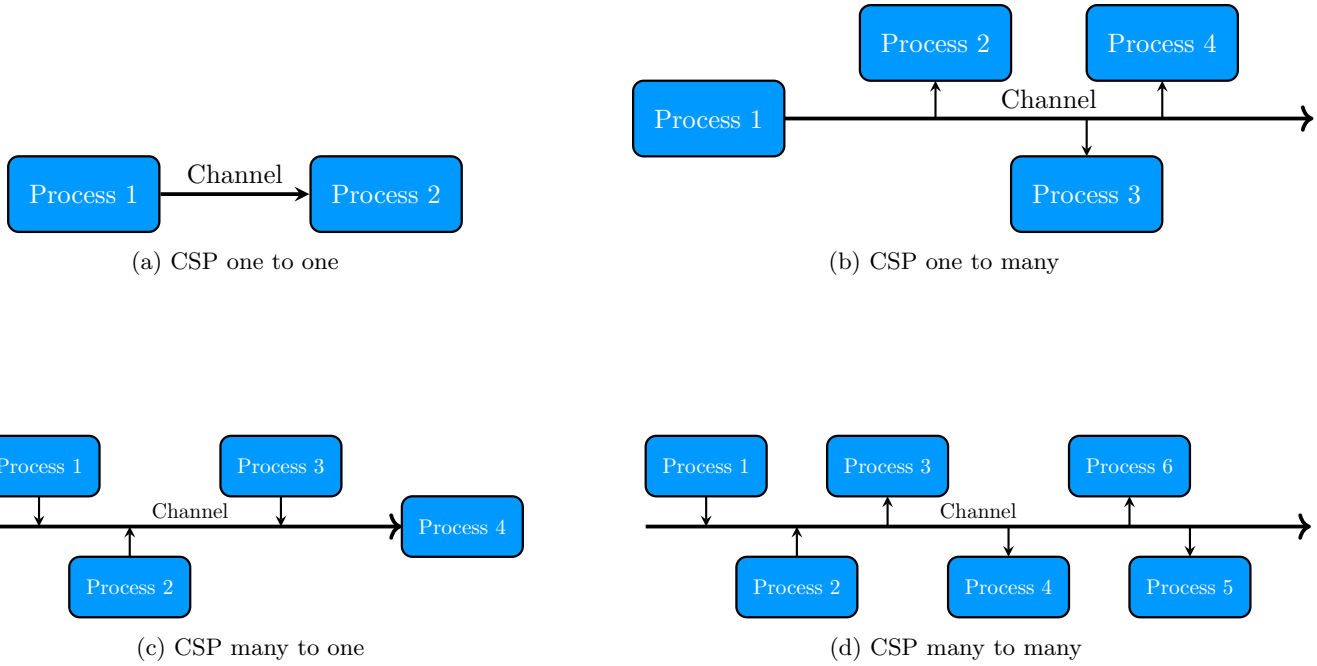
Various attempts have been made to solve this problem, such as the introduction of mutexes or locks. Though it does not solve our problem completely as these "solutions" introduce deadlocks, which is a state, where multiple processes are waiting for each other and the program stalls indefinitely. These deadlocks might not happen every run and thus

introduces another layer of difficulty, as error reproducibility is essential for code debugging and therefore makes it hard to make reliable software.

Communicating Sequential Processes (CSP) is an algebra first proposed by Hoare [1] to solve exactly these issues. CSP is build on two very basic primitives, first is the process (which should not be confused with operating system processes). A process could be an ordered sequence of operations. These processes do not share any memory, therefore one process cannot access a specific value in another process (which solves the problems we had with shared memory).

The other primitive is channels, which is the way the processes communicate with each other. You can pass whatever you want through these channels, but once you pass the value, the process lose access to it. There are a lot of ways the processes and channels can be arranged. The most simple can be found in figure 3.1a, which illustrates process 1 passing a value onto a channel, which process 2 takes as input. Some different configurations can be found in figures 3.1b-3.1d.

Using these primitives as the programming paradigm, multiprocess workloads can be designed without the shared memory problems mentioned earlier. The asynchronous nature of CSP will though be a problem for hardware models, as we will see shortly and is the fundamental inspiration behind Synchronous Message Exchange.



(a) CSP one to one

(b) CSP one to many

(c) CSP many to one

(d) CSP many to many

Figure 3.1: Figures illustrating various CSP process configurations.

3.2 Synchronous Message Exchange

The idea of using CSP for hardware modeling was first introduced in Rehr et al. [4] and later further developed in [5], where two students successfully implemented a vector processor using PyCSP (a CSP library for python). They however found a number of shortcomings of using PyCSP for hardware modeling. The need for global synchronicity, when simulating hardware models using CSP, meant that additional channels were needed to simulate clock progress and emulation of latches among others. This caused an explosion of required channels and processes, which is an unnecessary increase in complexity.

They did however find that building isolated processes and then connecting them via channels proved to be a powerful approach for building larger hardware models. As in the unit testing method one can develop and test individual processes and then connecting them together and form larger hardware models.

With this in mind a more suitable message-passing framework for hardware modeling was developed, with the following requirements:

- Globally synchronous
- Broadcasting channels
- Hidden clock
- Shared nothing
- Implicit latches

All these observations laid the foundation behind Synchronous Message Exchange (SME), Vinter and Skovhede [7]. Since then the SME framework has been implemented in the .NET framework, Skovhede and Vinter [6], which is the version of SME that will be used throughout this thesis.

3.2.1 The hidden clock

To ensure predictability for hardware models some form of global coordination is needed. If processes could read and write signals at any time there is no way of knowing if subsequent processes would use old or new data. This would lead to undesirable behavior.

Therefore a clocking methodology is introduced to establish coordination between processes. In SME a hidden clock exist, which synchronizes all processes. The clock can be thought of as a function cycling between high and low, as shown in Figure 3.2.

In SME, Clocked processes are then activated (meaning the procedure defined inside the process gets executed) on rising clock edges. Un-clocked SME processes on the other hand first activates once its input has been written to by other (clocked or un-clocked) processes.

A single clock cycle is then defined to start at every rising edge and end at the next one. In a clock cycle the following then happens: A clocked process gets activated on a rising clock edge, which outputs a value to some bus that is connected to an un-clocked process, this will then activate the process. This whole procedure would then be repeated in the following cycle. This ensures global coordination between processes.

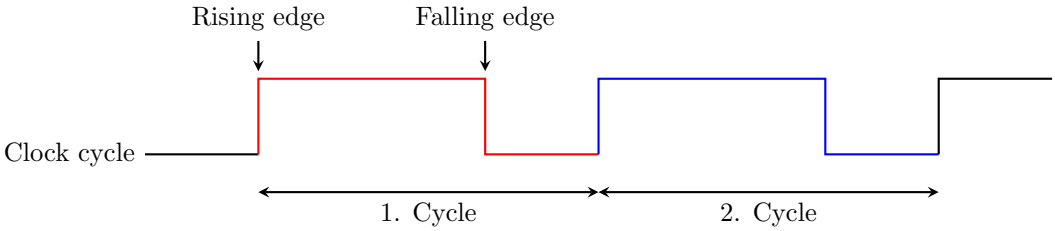


Figure 3.2: Illustration of 2 clock cycles where the first clock cycle is shown in red and the second clock cycle in blue.

3.2.2 Broadcasting and buses

In CSP processes communicate with each other using channels that make use of the rendezvous protocol, meaning that data first gets transmitted through the channel once the destination says it is ready to receive.

SME on the other hand uses a broadcasting model, meaning that data instead gets transmitted to SME busses, that any process can read from. In other words a process can broadcast its output to multiple processes through a single bus. Using CSP, multiple channels would have to be used to emulate a single bus in SME. When data is written to a bus it will first be available in the following clock cycle. Furthermore a bus is able to contain multiple data types and values that can be accessed by a process.

We will not dwell further into the theory behind SME in this thesis. Instead we will be developing actual units in SME, that will be needed when we later implement the RISC-V architecture. The SME syntax will be introduced as we go through the examples and is intended to be a from-scratch introduction for the uninitiated reader.

3.2.3 SME setup and structure

We will in this section go through the code structure I have decided to use for the development of all logic units. Much thought has been given to this and from previous experience doing code projects I would say it is good practice to think about the general structure of the project for ease of development and future extensions.

A modular design is essential, as many logic units is going to be needed when implementing the RISC-V architecture. This also allows for easy debugging and testing, as individual units

can be addressed before integrating them into a larger system. Each logic unit will also be separated into two files. One which contains all the buses and one which contains the function of the unit. The thought behind this is to help compartmentalize bus declarations from unit function and hopefully ease development.

In the following it is a prerequisite that .NET Core SDK[1] is installed on their respective systems to be able to run the code. All following code examples can be found by clicking here or if reading printed version the full link can be found in this footnote[2].

Project structure example

We are going to build an AND gate for this example. This will be the only section were the complete code is shown, as this section is meant to serve as a "quick start guide" for readers beginning their journey into SME. Subsequent sections will only show relevant parts of the code.

For this quick example we are going to start a project in a folder called `.../ANDGate/`, which is where all shown files are going to be placed. To do this we are going to open up a terminal window and navigate to the desired folder destination (for example the Desktop) and invoke the command:

```
$ dotnet new console -n ANDGate
```

where "`ANDGate`" can be changed to a name of choice.

Next navigate into the folder

```
$ cd ANDGate
```

and add the necessary SME libraries

```
$ dotnet add package SME --version=0.4.0-beta
$ dotnet add package SME.GraphViz --version=0.4.0-beta
$ dotnet add package SME.Tracer --version=0.4.0-beta
$ dotnet add package SME.VHDL --version=0.4.0-beta
```

There should now exist 3 items inside the `ANDGate` folder: `Program.cs`, `ANDGate.csproj` and a folder called `obj`. If all went well we should now be able to begin with our project.

I always find drawing a quick flowchart before implementing code gives a nice overview the project. So in the spirit of this I've drawn out a quick sketch of the project in Figure 3.3, which shows the two processes we are going to create, namely the SME simulator and the AND gate process.

[1] The SDK can be found here https://dotnet.microsoft.com/download

[2] https://github.com/DanielRamyar/Master_Thesis/tree/master/SME_Implementations

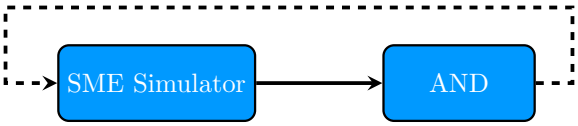


Figure 3.3: Flowchart of AND gate SME project. Here the rectangles represent individual processes, which lies in a file of its own. The solid arrow represents a bus and dashed arrow a bus going to a clocked process.

Opening up the `Program.cs` file, we are going to add a couple of lines, as shown in Listing 3.1. `Program.cs` contains the `Main()` method of the project and is the entry point of any C# program. To use the SME library we import it with the "`using SME`" command as shown in the second line.

Within the `Main()` method, the first function we meet is the using function, this is just to ensure the resources are properly cleaned up after the simulation.

Hereafter we see the simulation object, which as the name implies, is responsible for the simulation of the logic unit. We then pass the simulated input to the `ANDgate` object, which will then perform the AND operation (it is not necessary to pass the simulation object itself to the `ANDGate`, but it is shown here to not cause confusion about how the simulation passes input values to the AND process). We will see how these objects are defined shortly.

Lastly we can configure the simulation with the sim object, which uses fluent syntax. In this case we configured it to build a CSV file, which shows what each bus contained at every clock cycle. We build a graph, which outputs a `.dot` file that shows how all the processes are connected and we transpile the code to VHDL. It should be noted that `Run()` should always be the last method called.

The file which contains the `Main()` will always be named `program.cs` in this thesis, but the naming is irrelevant.

```
1  using System;
2  using SME;
3
4  namespace ANDGate {
5      class Program {
6          static void Main(string[] args) {
7              using(var sim = new Simulation()) {
8
9                  var simulator = new ANDGateSimulator();
10                 var ANDcalculator = new ANDGate(simulator.input);
11
12                 sim
13                     .BuildCSVFile()
14                     .BuildGraph()
15                     .BuildVHDL()
16                     .Run();
17             }
18         }
19     }
20 }
```

Listing 3.1: The `Program.cs` file, which contains the `Main()` method for the project.

We are going to need a way to test our AND gate, this is what the simulation file is for and is shown in Listing 3.2. Remember to name your `namespace` the same as in the project file. The first two definitions inside the simulation process create or load the two buses for the test. Notice that the gate output is labeled as an input bus and vice versa for the gate input. This is because the simulation is a process in and of itself and therefore needs to output the test values to a bus, which the `ANDGate` process takes in as input.

Now everything within the `Run()` method is what is going to be simulated and each time we want a new clock cycle to occur, we use the line `await ClockAsync()`. In a simulation process any .NET library is allowed and is not going to get transpiled to a VHDL file. Therefore we can print the output of the AND gate to console to see whether or not it works correctly. We do this by transmitting data via the input bus, which contains two signals, `in1` and `in2`. We then set the two signals to false wait a clock cycle and print the output. Since this is a fairly small system we can test for all input combinations and look at the outputs to see if the gate is behaving correctly.

```
1  using System;
2  using SME;
3
4  namespace ANDGate {
5      public class ANDGateSimulator : SimulationProcess {
6          [InputBus]
7          public readonly GateOutput output = Scope.CreateOrLoadBus<GateOutput>();
8
9          [OutputBus]
10         public readonly GateInputs input = Scope.CreateOrLoadBus<GateInputs>();
11
12         public async override System.Threading.Tasks.Task Run() {
13             Console.WriteLine("Starting test!\n");
14             await ClockAsync();
15
16             input.in_1 = false;
17             input.in_2 = false;
18
19             await ClockAsync();
20             Console.WriteLine($"Gate input: {input.in_1} (Input 1) - {input.in_2} (Input
                   2)\n");
21             Console.WriteLine($"AND gate output: {output.out_AND}");
22             ... (Abbreviated the code here for space concerns)
23             Console.WriteLine("Done testing!");
24         }
25     }
26 }
```

Listing 3.2: The simulator file, which specifies how the simulation is run. Most lines in the run method are concatenated brievity.

Hereafter we are going to make our bus declarations in a new file called buses.cs. Looking at Listing 3.3, we see that the bus declarations are made using the interface command, which inherits its fields and methods from the `IBus` interface. The name for the bus is free of choice and in this case are called `GateInputs` and `GateOuputs` respectively.

A bus can contain multiple signals of various types. Since the AND gate evaluates binary values, the `bool` type has been chosen for the input and output signals. You may have noticed that all the signals have the `InitialValue` attribute. What this does is that once the bus is created it will contain an initial value. If this is not done you have to be careful not letting clocked processes read the bus before any value has been given to it, as this will crash the program.

Then we have the `TopLevelInputBus` and `TopLevelOuputBus` attributes, they tell SME which buses goes in and out of the hardware implementation. Remember that this attribute is only given to buses which are interfacing with some outside process, in this case the simulation process. Internal buses should not be given this attribute. By an internal bus we mean a bus which is connecting processes inside the hardware implementation (see Figure 3.6, here the internal buses goes from the NOT gates to the AND gates).

```
1  using System;
2  using SME;
3
4  namespace ANDGate {
5      [TopLevelInputBus]
6      public interface GateInputs : IBus {
7          [InitialValue]
8          bool in_1 {get; set;}
9          [InitialValue]
10         bool in_2 {get; set;}
11     }
12
13     [TopLevelOutputBus]
14     public interface GateOutput : IBus {
15         [InitialValue]
16         bool out_AND {get; set;}
17     }
18 }
```

Listing 3.3: The file where all bus declarations are made.

Lastly we are going to define our AND gate in a process class. We create a file called `ANDGate.cs`, where our process is going to lie. We see how the process is defined in Listing 3.4. In line 5 we define the class `ANDGate` and since the AND gate is only going to execute once per cycle, we are going to inherit from the `SimpleProcess` class.

We are then loading our output bus in lines 6-7 and declaring our input bus in lines 9-10. Then I have added a constructor, which checks whether or not the object passed from the simulator object (remember that we passed the simulator inputs in the project file) contains

any values.

Finally we have the `OnTick` method, which contains the logic that has to be performed. In line 16 we perform the logical AND operation between the two input signals and then pass it on to the output. The `OnTick` method makes sure that the code within runs exactly once per cycle.

```
1  using System;
2  using SME;
3
4  namespace ANDGate {
5      public class ANDGate : SimpleProcess {
6          [OutputBus]
7          public readonly GateOutput output = Scope.CreateOrLoadBus<GateOutput>();
8
9          [InputBus]
10         private readonly GateInputs m_input;
11         public ANDGate(GateInputs input) {
12             m_input = input ?? throw new ArgumentNullException(nameof(input));
13         }
14
15         protected override void OnTick() {
16             output.out_AND = m_input.in_1 && m_input.in_2;
17         }
18     }
19 }
```

Listing 3.4: This is where we define the function of the AND gate process.

Finally run the project by returning to the terminal. Navigating to the directory where the project is placed, simply type following command

```
1  $ dotnet run
```

this will output 3 files to a folder named output placed in the same directory as the project. These files consist of `network.dot`, `trace.csv` and a `VHDL` folder. The `network.dot` file is a graphical representation of our model, which we have shown in Figure 3.4. Next we have the `trace.csv` file, which shows what each signal in all buses contained at each clock cycle. This is a very helpful tool when debugging, as problems can be identified quickly if wrong values are spotted throughout the simulation.

Lastly we transpiled the VHDL files to the `VHDL` folder, which we can verify using any VHDL simulator (`GHDL` is recommended). The folder contains a `makefile`, which automates this process, so you simply navigate to the `VHDL` folder in your terminal and run the command

```
1  $ make
```

assuming that you have already installed GHDL on your system.

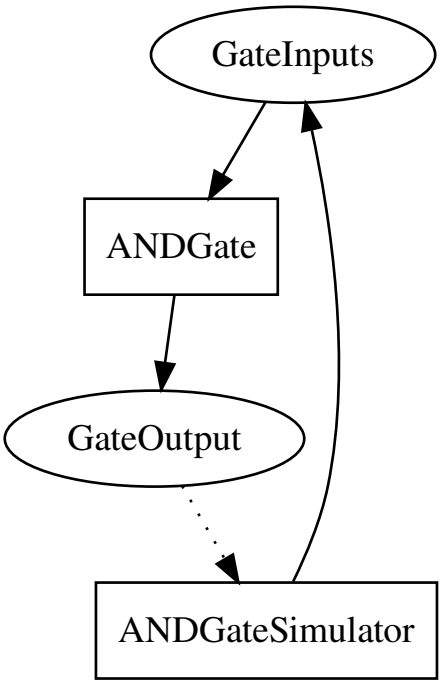


Figure 3.4: Graphical representation of the AND gate generated by the outputted network.dot file. Here rectangles represent processes and ellipse buses. The arrows show the direction of the data flow, where a dotted line indicates data flow to a clocked process, which means that the process is activated on a rising clock edge.

3.2.4 The Decoder

In this section we will start by implementing the 2-bit decoder (see 2.2.1) and then look at the n-bit implementation.

First we set up the logic table, as shown in Table 3.1.

Input		**Output**			
In1	In0	Out3	Out2	Out1	Out0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

Table 3.1: Logic Table of a 2 input decoder, where the binary representation of the input determines which output gets asserted. For example when In1=1, In0=0 output 1 will get asserted as the binary representation for 2 is 10.

We then derive the logic equation from the table:

$$Out0 = \overline{In1} \cdot \overline{In0}, \quad Out1 = \overline{In1} \cdot In0, \quad Out2 = In1 \cdot \overline{In0}, \quad Out3 = In1 \cdot In0 \tag{3.1}$$

We notice from Eq. 3.1 that we need two NOT gates (one each input), four AND gates and four outputs. Since we only have 1 minterm (a term which results in true) per output

no OR gates are necessary. We can then create a circuit diagram of the decoder, as shown in Figure 3.5 and use it as our design guide when we implement it in SME.

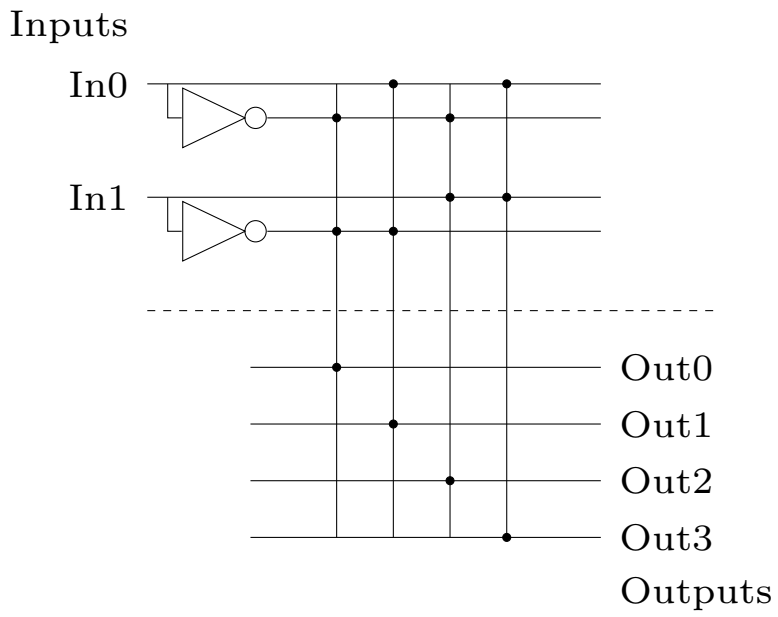


Figure 3.5: Schematic of the 2-bit decoder SME project.

Starting with the processes, we see that we need two NOT processes. Hereafter we know that each vertical line goes to an AND gate and since there are four, we create four AND processes. Moving on to the buses we see that there are two input buses, two negated input buses and four output buses. Wiring the buses to the processes correctly we end up with the SME implementation as shown in Figure 3.6. The code can be found following the link in this footnote[3]

The N bit Decoder

The N bit decoder SME implementation is not trivial and the solution here is inspired from Carl Johnson's master thesis [2]. Since SME needs everything to be known at compile time, we need a way of auto-generating code depending on how many bits we want. This is achieved by using C# templates. We notice that the number of NOT gates needed is always the same as the number of inputs, so for an N bit decoder we need N NOT gates. Knowing this we can use a for loop in the code generator to generate N NOT gate processes and since each NOT gate only has one corresponding input, we can add the input and output buses to each NOT gate process in each loop iteration.

Next we notice that the number of AND gates scales as 2^N. Doing the same as for the

[3] https://github.com/DanielRamyar/Master_Thesis/tree/master/SME_Implementations/Decoder_2_Bit_new_implementation

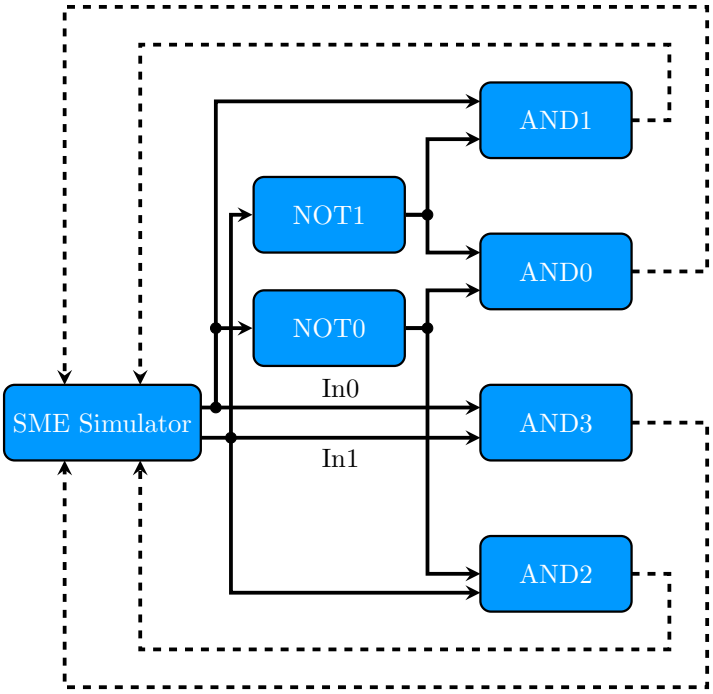


Figure 3.6: Flowchart of decoder SME project. Here the rectangles represent individual processes. The solid arrow represents a bus and the dashed arrow a bus going to a clocked process. Solid dots show extension of the same bus and any crossing lines are not connected.

NOT gates, we can use a loop to generate each AND gate process. However it is not trivial adding the correct input buses to each AND gate process. The way this was solved was to add a nested loop, with indices `i` for the outer loop and `j` for the inner loop. `i` is responsible for creating the correct number of AND gate processes and `j` is responsible for the correct number of input buses added to these processes. So for N = 2, we have `i` = 2^2 = 4 AND gates and `j`=2 inputs to these.

Lastly we know that each AND gate should have the correct minterm as input e.g. AND1 in 3.6 has `In0` and NOT(`In1`) as input, which produces the correct output. To do this, we use the conditional operator with the condition `((i ≫ j & 1) == 1)`, this will ensure that all AND gate processes gets the correct inputs.

A similar approach was used to generate the bus and simulator files and can be found in this footnote[4]. To change the number of input bits, simply go into all `.tt` files and change the `n` variable at the top of the file to the desired number of bits and run the code generation.

3.2.5 The Multiplexer

In the section we will implement the 2-bit multiplexer (see 2.2.2) in SME. First we setup the logic table, as shown in Table 3.2.

[4] https://github.com/DanielRamyar/Master_Thesis/tree/master/SME_Implementations/Decoder_n_Bit

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

Table 3.2: Logic Table of a 2 input multiplexer, where A and B are the input bits, S is the select bit, which chooses whether input A or B should be outputted unchanged and C is the output.

Hereafter we derive the logic equation from the table:

$$C = A \cdot \overline{B} \cdot \overline{S} + A \cdot B \cdot \overline{S} + \overline{A} \cdot B \cdot S + A \cdot B \cdot S \tag{3.2}$$

this can be reduced to (see 2.1.3)

$$C = A \cdot \bar{S} + B \cdot S. \tag{3.3}$$

We see from Eq. 3.3 that we need a single NOT gate, two AND gates and a single OR gate. We can then create a circuit diagram of the decoder, as shown in Figure 3.7 and use it as our design guide when we implement it in SME.

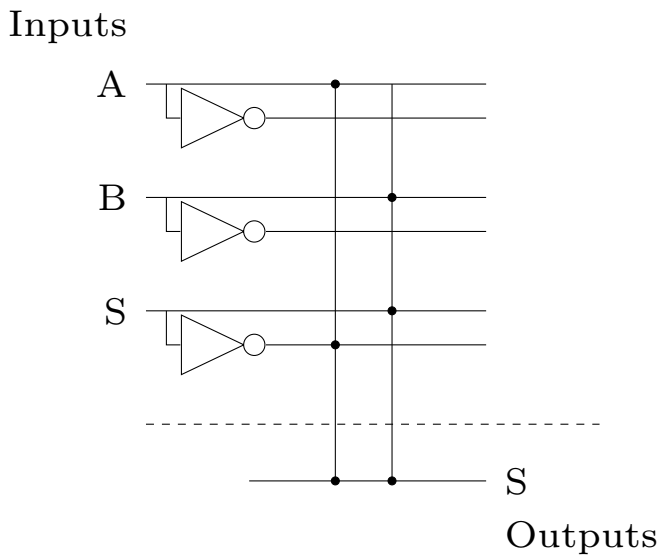


Figure 3.7: Schematic of the 2-bit multiplexer SME project.

Starting with the processes, we notice that we only need one NOT process. Hereafter we know that each vertical line goes to an AND gate and since there are two, we create two

AND processes. Lastly we know that each horizontal line in the OR plane corresponds to an OR gate, since we only have one we create one OR process. Moving on to the buses we see that there are 3 input buses, where we only use the negated signal from one. Next we have two output buses from the AND gates and a single output bus from the OR gate, that is 7 buses in total we have to create. Wiring the buses to the processes correctly we end up with the SME implementation as shown in Figure 3.8. The code can be found following the link in this footnote [5]

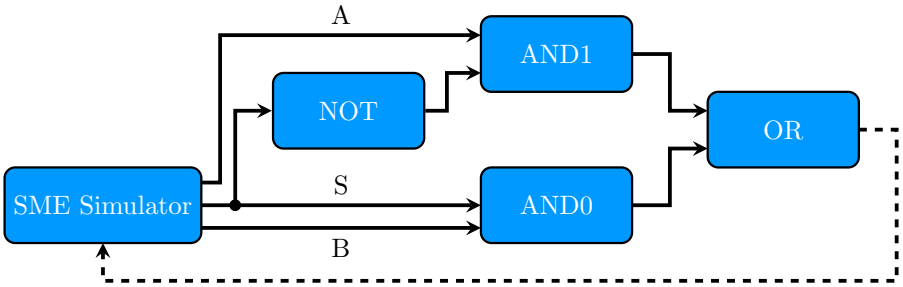


Figure 3.8: Flowchart of multiplexer SME project. Here the rectangles represent individual processes. The solid arrow represents a bus and the dashed arrow a bus going to a clocked process. Solid dots show extension of the same bus and any crossing lines are not connected.

3.2.6 Full Adder

No general purpose CPU would be any good if it could not add two numbers together. This is why we will go through a gate level implementation of an adder unit in SME. Doing this will also give the reader a good intuition on how the arithmetic logic unit (ALU) works, when we later move away from this approach and let the C# compiler do the work for us and the gate level implementation is "hidden".

As always we start by writing up a logic table with the correct function, which has been done in Table 3.3. The idea is that by chaining multiple 2-bit full adders, we will be able to create an n-bit adder. Therefore each unit contains a carry input and a carry output. The rest is just a matter of asserting the sum output depending on the sum of bits in the inputs. For example in row 3 (not counting the label row) in Table 3.3, we see that only input B is asserted, then it follow from addition $A + B + CarryIn = 1$ hence the sum has to be asserted. If the result exceeds 1-bit e.g $1 + 1 = 10$ in binary, the CarryOut signal is asserted and sum output deasserted. Following this approach the whole table can be filled.

Now deriving the logic equations from the table we get the following for the CarryOut signal:

$$CarryOut = B \cdot CarryIn + A \cdot CarryIn + A \cdot B + A \cdot B \cdot CarryIn \tag{3.4}$$

[5] https://github.com/DanielRamyar/Master_Thesis/tree/master/SME_Implementations/Multiplexer_2_Input

A	B	CarryIn	CarryOut	SUM
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Table 3.3: Logic Table of a 2 input multiplexer, where A and B are the input bits, S is the select bit, which chooses whether input A or B should be outputted unchanged and C is the output.

Notice that the negated input has been omitted. This can be done as adding these just results in the last term, so removing these doesn't change the output. Simplifying the expression

$$CarryOut = B \cdot CarryIn + A \cdot CarryIn + A \cdot B(1 + CarryIn), \tag{3.5}$$

and using that $(1 + CarryIn) = 1$ we get

$$CarryOut = B \cdot CarryIn + A \cdot CarryIn + A \cdot B. \tag{3.6}$$

For the SUM signal we get

$$SUM = \bar{A} \cdot \bar{B} \cdot CarryIn + \bar{A} \cdot B \cdot \overline{CarryIn} + A \cdot \bar{B} \cdot \overline{CarryIn} + A \cdot B \cdot CarryIn. \tag{3.7}$$

From Eq. 3.6 and 3.7, we see that we will need three NOT gates (one for each input), seven AND gates and two OR gates. From this we can then create the circuit diagram for the full adder, which is shown in Figure 3.9. Hereafter we can use the diagram as our guideline for our SME project.

Starting with the processes, we create 3 NOT gate processes. Hereafter we count seven vertical lines in the AND plane of the schematic, which means we have to make seven AND gate processes. Lastly we see that there are two horizontal lines in OR plane, so we create two OR gate processes.

Moving on to the buses we see that we have 3 input buses and 3 negated input buses. Here we really see the power of SME, since I only have to create a bus once from which all processes can read data from, because of broadcasting (this means i don't have to make a separate copy of the same bus every time a process needs access to it i.e. the bus gets "broadcasted" to all processes). All AND gate processes have an output, so we create 7 output buses for these. Lastly we create the two output buses for the OR gate processes. That is 15 buses in total.

Wiring the corresponding buses to the correct processes we end up with the SME imple-

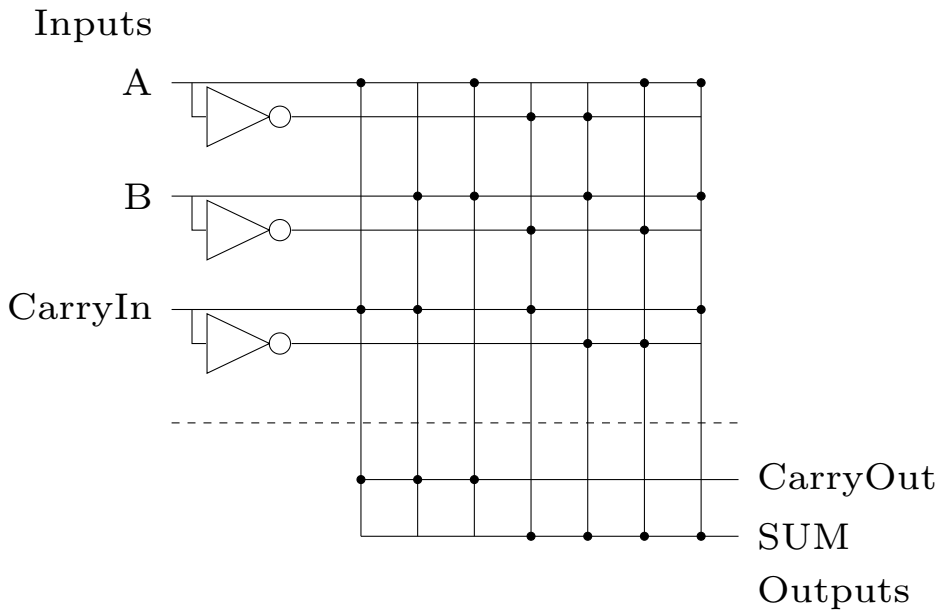


Figure 3.9: Schematic of the 2-bit multiplexer SME project.

mentation as shown in Figure 3.10. Now this implementation quickly became very complex compared to previous units and this is only for the 2-bit full adder! Realizing that it would be a very ambitious project to do a full gate level implementation of the RISC-V processor, we will move on to another approach discussed in the next section.

The code can be found following the link in this footnote[6]

3.2.7 Arithmetic Logic Unit

We will in this section move away from the gate level implementation of various units. Instead we will fully utilize the power of C# and let the compiler do most of the work for us. What is meant by this is for example rather than writing a gate level implementation of 64-bit adder, we would instead write a process, which just adds two numbers together using the C# syntax, which simplifies things a whole lot.

The arithmetic logic unit (ALU) is where all operations happens in the RISC-V architecture. In the following we will go ahead and implement a simple ALU, which can do 4 operations. These operations are addition, subtraction, AND and OR. This will all be contained within the ALU process.

Drawing up a flowchart for the ALU SME project, as we have done in Figure 3.11, we see that the ALU takes 3 input, two lines with data to operate on and one line for selecting the desired operation. Now inside the ALU process a `switch` statement has been made (see Listing 3.5), which uses an operation code coming from the `OperationCode` bus to determine,

[6] https://github.com/DanielRamyar/Master_Thesis/tree/master/SME_Implementations/FullAdder_2_Bit

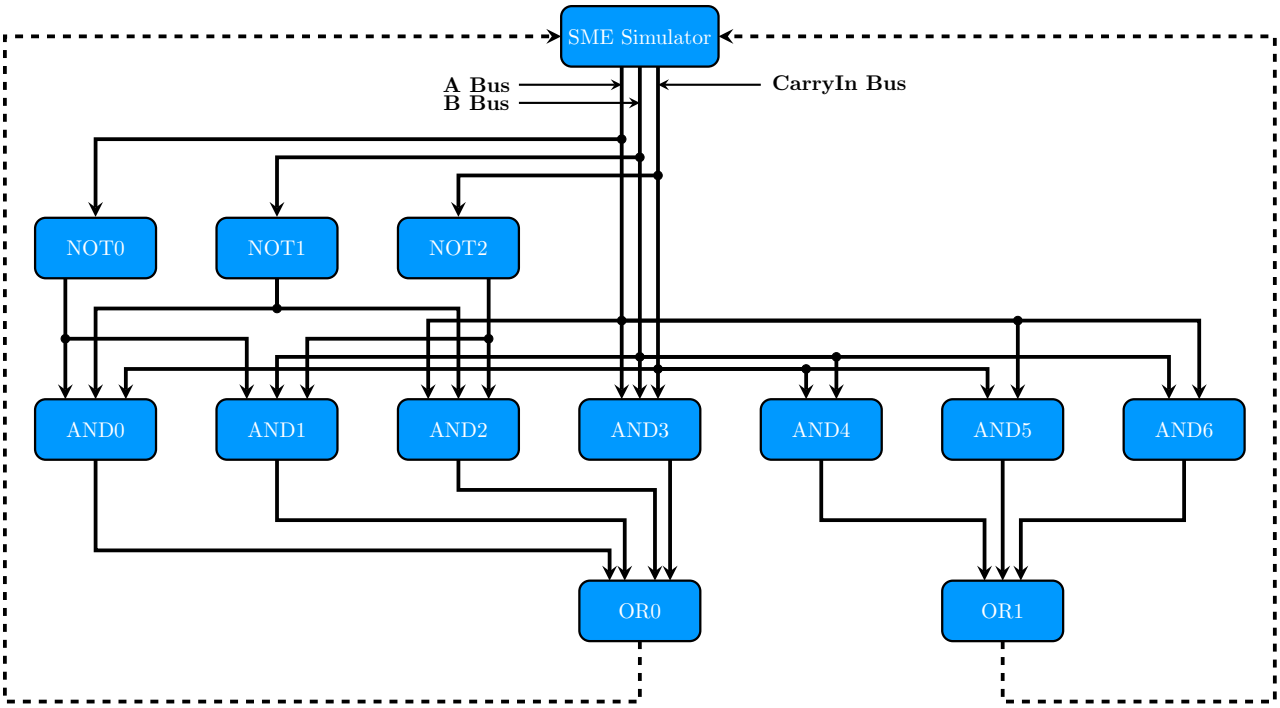


Figure 3.10: Flowchart of full adder SME project. Here the rectangles represent individual processes. The solid arrow represents a bus and the dashed arrow a bus going to a clocked process. Solid dots show extension of the same bus and any crossing lines are not connected.

which operation it has to do on our data. Hereafter the ALU outputs the result for us to read. Now if this had to be implemented using gate level approach, as we did before, it would have quickly been an overwhelming project and this is exactly why using a high level language to do the hard work for us is a powerful approach.

The code for this project can be found following the link in this footnote[7]

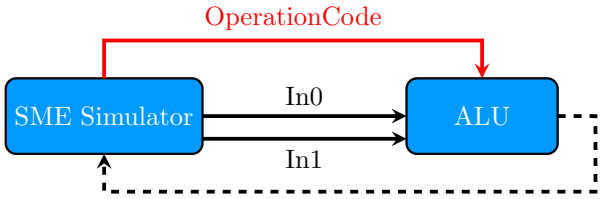


Figure 3.11: Flowchart of ALU SME project. Here the rectangles represent individual processes. The solid arrow represents a bus and the dashed arrow a bus going to a clocked process.

[7] https://github.com/DanielRamyar/Master_Thesis/tree/master/SME_Implementations/ALU_1_Bit

```
protected override void OnTick() {
    switch (m_OperationCode.Value) {
        case 0:
            output.Value = m_A.Value & m_B.Value;
            break;
        case 1:
            output.Value = m_A.Value | m_B.Value;
            break;
        case 2:
            output.Value = m_A.Value + m_B.Value;
            break;
        case 3:
            output.Value = m_A.Value - m_B.Value;
            break;
}
```

Listing 3.5: The Ontick() method for the ALU process. Here we use a `switch` statement, controlled by the `OperationCode` bus, to determine which operation the ALU should perform.

Chapter 4

Introduction to RISC-V instructions

To understand the design decisions made, building the RISC-V CPU in Chapter 5, we will go through the fundamentals of the machine language behind the RISC-V architecture.

To this end we will go through a top down approach going from assembly code written by the programmer all the way down to the machine code representation in the hardware implementation. We will therefore start out by introducing the RISC-V assembly code in section 4.1.

Hereafter we will talk about the operands of the RISC-V CPU, which unlike high-level programming languages are confined in a small directory called the *register*.

Then we will go through the base 2 numeral system to make meaning of the following section, which will go through how the assembly language, also known as instructions, is represented in the CPU itself. This chapter is based on chapter 2 in [3], which can be referred to if more detailed explanations are needed.

4.1 RISC-V Assembly

Like every computer, we need a way of telling it what to do. This is done via the RISC-V assembly language, which is a human readable abstraction of the RISC-V instruction set. There exists multiple instruction standards like `RV32I` and `RV64I` for the RISC-V architecture and extensions to these[1]. This project will implement the `RV64I` instructions. For writing small RISC-V Assembler programs I can recommend the RISC-V Assembly simulator *Jupiter*[2].

For the assembler to know where to start reading instructions from we use:

```
1 .global __start
2
3 __start:
```

[1]The exect specifications can be found here https://riscv.org/specifications/
[2]https://github.com/andrescv/Jupiter

where all instructions are added after the `__start:` command.

We will in this example start by adding two numbers together. Keep in mind however that this is not executable by the Jupiter simulator, as crucial instructions have been omitted to ease readability for the reader. An executable version of the example will be shown when the necessary concepts have been introduced.

To find the sum between two numbers, we use the `add` instruction:

```
1 .global __start
2
3 __start:
4     add a, b, c
```

here we tell the CPU to find the sum between the two varibles, b and c, and save the result in variable a. It is worth noting that the RISC-V assembly language instructions always have exactly 3 operands and always performs a single operation.

This may seem to restrictive, because how would the then calculate the following equation:

$$d = a + b + c \tag{4.1}$$

The answer is to split the equation up in small bites. To add more instructions we simply add a new line, since only one instruction is allowed per line. The indent is just there for readability purposes and is not necessary.

Having that in mind we therefore split Eq. 4.1 the following way:

```
1 .global __start
2
3 __start:
4     add d, a, b # Find the sum between a and b and put the result in variable d
5     add d, d, c # Take the previous result which now lies in d = a + b and add c to it.
6                 # Then save the result to the same varible
```

We see that we had to use two instructions in order to solve Eq. 4.1. Everything right of the Hashtags are the way we add comments to code, which the computer ignores.

In the previous examples, we made use of a single instruction, the `add` instruction. There are however 49 base instructions in the RISC-V instruction set, a subset of these is shown in Table 4.1. The rest can be referred to in the RISC-V reference card in the appendix.

Name	Assembly	Description (in C)
Add	add rd, rs1, rs2	rd = rs1 + rs2
Subtract	sub rd, rs1, rs2	rd = rs1 − rs2
AND	and rd, rs1, rs2	rd = rs1 & rs2
OR	or rd, rs1, rs2	rd = rs1 | rs2
XOR	xor rd, rs1, rs2	rd = rs1 ^ rs2

Table 4.1: This table is a subset of the RISC-V assembly language arithmetic instructions. For the complete base instruction set, please refer to the RISC-V reference card in the appendix.

4.2 Register

In a high-level programming language we do not think much, when we declare a variable. For the most part the memory management is hidden for the end user and is taken care of by the compiler. However we do not have this convenience in assembly, where the operands of instructions are much more restricted.

In the RISC-V architecture we have a special location reserved for the operands build directly into the hardware implementation, which is known as the *register*. It comprises of 32 storage locations for operands, each of which is 64 bits long (this depends on the architecture and can vary in size). These locations are accessed by the instructions, using the numbers 0 through 31 with an x in front e.g. using our addition instruction we type:

```
1 add x18, x12, x13
```

which finds the sum between values stored in register 12 and 13 and stores the result in register 18.

We do in principle have full control over what we store in these registers, but we will follow the conventions specified in the RISC-V spec sheet, which can be found here[3] and is summarized in Table 4.2. Notice that each register also has an ABI (application binary interface) calling convention, so we could also type

```
1 add s2, a2, a3
```

to access the exact same registers as in the previous example.

Register	ABI Name	Description
x0	zero	Hard-wired zero
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5	t0	Temporary/alternate link register
x6-7	t1-2	Temporaries
x8	s0/fp	Saved register/frame pointer
x9	s1	Saved register
x10-11	a0-1	Function arguments/return values
x12-17	a2-7	Function arguments
x18-27	s2-11	Saved registers
x28-31	t3-6	Temporaries

Table 4.2: This table shows the register naming conventions RISC-V architecture. Both the register number and ABI name can be used to access a register.

[3] https://riscv.org/specifications/

4.3 Data transfer instructions

So far our operands in Section 4.1 and 4.2 have been restricted to the register. This approach will however quickly run into its limitations, as only a small amount of data can be stored here. To support larger data sizes and complex data structures, we therefore need a larger storage location for our data, which we call the *memory*.

To access the memory we make use of the *data transfer instructions*. A subset of these are outlined in Table 4.3.

Name	Assembly	Description (in C)
Load Byte	lb rd, rs1, imm	rd = M[rs1+imm][0:7]
Load Half	lh rd, rs1, imm	rd = M[rs1+imm][0:15]
Load Word	lw rd, rs1, imm	rd = M[rs1+imm][0:31]
Load Doubleword	ld rd, rs1, imm	rd = M[rs1+imm][0:63]
Store Byte	sb rs1, rs2, imm	M[rs1+imm][0:7] = rs2[0:7]
Store Half	sh rs1, rs2, imm	M[rs1+imm][0:15] = rs2[0:15]
Store Word	sw rs1, rs2, imm	M[rs1+imm][0:31] = rs2[0:31]
Store Doubleword	sd rs1, rs2, imm	M[rs1+imm][0:63] = rs2[0:63]

Table 4.3: This table is a subset of the RISC-V assembly language data transfer instructions. For the complete base instruction set, please refer to the RISC-V reference card behind the appendix.

The memory can be thought of as a big array, where each element is 8 bits or a byte long. In RISC-V the memory is little-endian addressed. This means that if we had to store the doubleword value 8444628 in memory, which is equal to 10000000 11011010 11010100 in binary, it would get stored such that rightmost 8 bits would be the *base address* and the subsequent 8 bit chunks offset with respect to the base address. This is illustrated in Figure 4.1.

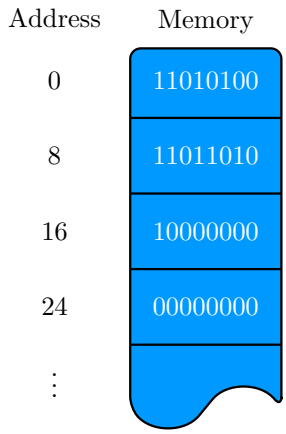


Figure 4.1: This figure illustrates the memory, with the data contents inside the blue squares. They each contain 1 byte or 8 bits of data. To the left we see the address of each block, which is offset by 8 due to the number of bits they contain.

Assuming that we in a high-level programming language have associated an array of doublewords, meaning each element in the array is 64 bit, with the variable `A`. We now want to calculate

$$A[2] = x + A[4] \tag{4.2}$$

where x is stored in register x12 and suppose that the base address of `A` is saved in register x9. Using the load doubleword instruction, we copy the fourth element into the register 6

```
1 ld x6, x9, 256 # The temporary register x6 gets fourth element in array A.
```

here the first argument tells the CPU where to save the loaded data, in this case register 6. The second argument tells the instruction, where in memory the array is located. Finally the last argument specifies the offset, so if each doubleword is 64 bits long (each element in A is 64 bits) and we want to access element four in A, we need an offset of 256 ($64 \cdot 4 = 256$).

We then want to do the sum and save it back to the second element in A:

```
1 add x6, x6, x12 # x + A[4]
2 sd x9, x6, 128  # Store sum in the second element of A
```

After finding the sum we save the value back into the second element of A using the `sd` instruction. Here the first argument takes the base address and third argument the offset. The second argument then contains the data to be stored.

4.4 Immediate instructions

Up until now we have had to load constants from memory (assuming the values got loaded into memory on startup) to the register everytime we wanted to do an arithmetic operation and what do we do if the memory do not contain the required data?

As operations with constants are the most frequently used in most programs, dedicated instructions have been added for this purpose. With these the constant is typed into the immediate field `imm` and is added directly into the instruction itself.

Using the add immediate instruction `addi` allows us to add data directly into the register and skip the step of loading constants from memory entirely, like so:

```
1 addi x13, x0, -5 # x13 = 0 - 5
```

notice that we made use of the zero register, which is hard-wired to zero, to add the number -5 to register 13. A subset of the immediate instructions has been outlined in Table 4.4. It should be noted that any instruction with an `imm` parameter when refering to the full instruction set behind the appendix, a numerical integer can be inserted just as shown.

Name	Assembly	Description (in C)
Add Immediate	addi rd, rs1, imm	rd = rs1 + rs2
AND Immediate	andi rd, rs1, imm	rd = rs1 & rs2
OR Immediate	ori rd, rs1, imm	rd = rs1 | rs2
XOR Immediate	xori rd, rs1, imm	rd = rs1 ^ rs2

Table 4.4: This table is a subset of the RISC-V assembly language arithmetic immediate instructions. For the complete base instruction set, please refer to the RISC-V reference card in the appendix.

4.5 Numeral system of a computer

In everyday life we are used to using the decimal numeral system, meaning that ten base digits are used to represent a quantity, such as 6 apples. However the decimal numeral system was not chosen to represent numbers in a computer. Here we live in a world were everything consists of signals being high and low voltage or transistors being on or off. Therefore a much more inherent counting system can be chosen to represent numbers in this world, namely the binary numeral system.

The binary digit or bit is the fundamental building block of any computer. A bit requires only two "symbols" to be represented. A high/low signal, on/off transistor, true/false statements or of course, but not limited to, the digits 0 and 1.

In this thesis we label the digits such that the rightmost digit in a number is the zeroth digit and then increasing the index going left. To avoid confusion we will always label which numeral system we are working with e.g. $10_{\texttt{two}}$ or $10_{\texttt{ten}}$.

Since humans do not usually think in binary, we need a way of converting decimal numbers to binary numbers. We know that any base representation of a number can be expressed in base 10 using[4] (note that the following derivation only works for positive/unsigned numbers)

$$N_b = \sum_{i=0}^{n} d_i b^i \tag{4.3}$$

where N_b is the number to be converted to base 10, b is the target base and d_i is the i'th digit in N_b, which satisfies $0 \leq d_k < b$. We can for example expand the number 1101

$$1101_{\texttt{two}} = (1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0)_{\texttt{ten}} = 13_{\texttt{ten}} \tag{4.4}$$

Now Eq. 4.3 is invertible by using the remainder theorem, which states that there exists a unique choice of quotient q and remainder r such that

$$n = qb + r \quad \text{and} \quad 0 \leq r < b \tag{4.5}$$

[4]The following derivation is inspired from
https://math.stackexchange.com/questions/1359770/why-does-the-division-algorithm-work-for-converting-between-number-bases

For example

$$13_{\,\texttt{ten}} = (6 \cdot 2 + 1)_{\,\texttt{ten}} \tag{4.6}$$

Rewriting Eq. 4.3 to

$$N_b = N'_b \cdot b + d_0 \tag{4.7}$$

where

$$N'_b = \sum_{i=0}^{n-1} d_{i+1} b^i \tag{4.8}$$

We can see that d_0 corresponds to the remainder r and N'_b to the quotient q in 4.5 . Dividing by the base b to get the remainder

$$\frac{N_b}{b} = N'_b + \frac{d_0}{b} \tag{4.9}$$

which we know corresponds to the leftmost digit in the target repressentation (see Eq. 4.4). The same process can be done with the quotient N'_b to gain the remaining digits. For example inverting $13_{\,\texttt{ten}}$ back to binary

$$\frac{13_{\,\texttt{ten}}}{2_{\,\texttt{ten}}} = 6_{\,\texttt{ten}} + \frac{1_{\,\texttt{ten}}}{2_{\,\texttt{ten}}} \tag{4.10}$$

we have the remainder 1, which is the leftmost digit and quotient 6. Reapeating with the quotient

$$\frac{6_{\,\texttt{ten}}}{2_{\,\texttt{ten}}} = (3 \cdot 2)_{\,\texttt{ten}} + 0_{\,\texttt{ten}} \tag{4.11}$$

we have the remainder 0 and quotient 3. Reapeating again

$$\frac{3_{\,\texttt{ten}}}{2_{\,\texttt{ten}}} = 1_{\,\texttt{ten}} + \frac{1_{\,\texttt{ten}}}{2_{\,\texttt{ten}}} \tag{4.12}$$

we have the remainder 1 and quotient 1. Reapeating again

$$\frac{1_{\,\texttt{ten}}}{2_{\,\texttt{ten}}} = 0_{\,\texttt{ten}} + \frac{1_{\,\texttt{ten}}}{2_{\,\texttt{ten}}} \tag{4.13}$$

we have the remainder 1 and quotient 0 and we are done. Arranging the remainders in the right order we have

$$13_{\,\texttt{ten}} = 1101_{\,\texttt{two}} \tag{4.14}$$

We have matched unsigned decimal numbers 0 through 15 with the corresponding unsigned binary value in Table 4.5.

Decimal	Binary	Decimal	Binary
0	0	8	1000
1	1	9	1001
2	10	10	1010
3	11	11	1011
4	100	12	1100
5	101	13	1101
6	110	14	1110
7	111	15	1111

Table 4.5: Table of unsigned decimal numbers and their unsigned binary representation.

4.5.1 Signed and Unsigned numbers

No computer would be any good for scientific purposes if we are not able to work with negative numbers. Therefore we need a way of representing them in binary. The standard implementation is the *two's complement* representation. Here leading zeros means a positive number and leading ones means a negative number. Using 4 bits we are able to represent numbers -8 to 7 and have been outlined in Table 4.6.

We observe that

$$x + \bar{x} = -1 \tag{4.15}$$

where x is a decimal and $\bar{x}$ its complement e.g. $0001_{\texttt{two}} + 1110_{\texttt{two}} = 1111_{\texttt{two}}$. Rearranging Eq. 4.15

$$\bar{x} + 1 = -x \tag{4.16}$$

Therefore to negate any number you simply invert all bits and add one e.g. going from -7 to 7

$$-7_{\texttt{ten}} = 1001_{\texttt{two}} \xrightarrow{\text{Invert Bits}} 0110_{\texttt{two}} \xrightarrow{+1} 0111_{\texttt{two}} = 7_{\texttt{ten}} \tag{4.17}$$

Notice however that -8 do not have any complement using only 4 bits. In the two's complement representation we will always end up with an extra negative number, if not careful this may cause problems and should be known.

Subtraction of two's complement numbers is very simple. You just add them together and throw away the last bit, which exceeds the size of the bit field. In other words if we got two 4 bit numbers we want to find the sum of and we still have a carry after the 4'th bit, which is supposed to be in the 5'th bit place, we throw away the carry. For example subtracting five with two

$$\begin{array}{r} {\scriptstyle 1\ 1\ \ \ \ } \\ 0\,1\,0\,1_{\texttt{two}} \\ +\,1\,1\,1\,0_{\texttt{two}} \\ \hline 1\,0\,0\,1\,1_{\texttt{two}} \end{array}$$

and throwing away the carry we get $0011_{\texttt{two}} = 3_{\texttt{ten}}$

Decimal	Binary	Decimal	Binary
0	0000	-1	1111
1	0001	-2	1110
2	0010	-3	1101
3	0011	-4	1100
4	0100	-5	1011
5	0101	-6	1010
6	0110	-7	1001
7	0111	-8	1000

Table 4.6: Table of unsigned decimal numbers and their unsigned binary representation.

4.6 Instruction representation in binary

All base instructions in the RISC-V architecture implemented in this project, `RV64I`, are 32 bits long, meaning 32 binary digits are used to represent an instruction. This is what all assembly code gets compiled into and is known as *machine code*. Going back to our `add` instruction

```
add x18, x12, x13
```

in machine code the instruction would look like

```
0000000 01101 01100 000    10010 0110011
funct7  x13   x12   funct3 x18   add
```

where first (remember we count from right) 7 bits tell the CPU what instruction it should perform, in this case `add`. The next 5 bits tells which register the CPU should save the result to, in this case `x18`. Bits 12-14 and 25-31 again tell the instruction type in conjunction with the first 7 bits. Bits 15-19 and 20-24 tell which register the first operand and second operand is located respectively, in this case register `x12` and `x13`. A subset of instruction formats is shown in Table 4.7. For a more comprehensive table please refer to the RISC-V instruction set sheet behind the appendix.

31 25	24 20	19 15	14 12	11 7	6 0	
funct7	rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]		rs1	funct3	rd	opcode	I-type
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type

Table 4.7: A subset of machine code instruction formats, where the `opcode`, `funct3` and `funct7` fields are responsible for instruction identification. The `rd`, `rs1`, and `rs2` fields are operand fields and is responsible for the register destination and register source for the first and second operands respectively. The immediate fields `imm` are where the constants are stored, where the I-type instruction stores the whole constant in bits 20-31 and S-type splits the constant into bits 7-11 and 25-31.

4.6.1 Hexadecimal

Now working with 32 bit strings can get quite bothersome. A more compact representation is possible using hexadecimal or base 16. Here a single hexadecimal digit corresponds to 4 binary digits. So we only need 8 hexadecimal digits instead of 32 bits to represent an instruction. All hexadecimal digits and their corresponding binary values are shown in Table 4.8.

Hexadecimal	Binary	Hexadecimal	Binary	Hexadecimal	Binary	Hexadecimal	Binary
$0_{\texttt{hex}}$	$0000_{\texttt{two}}$	$4_{\texttt{hex}}$	$0100_{\texttt{two}}$	$8_{\texttt{hex}}$	$1000_{\texttt{two}}$	$c_{\texttt{hex}}$	$1100_{\texttt{two}}$
$1_{\texttt{hex}}$	$0001_{\texttt{two}}$	$5_{\texttt{hex}}$	$0101_{\texttt{two}}$	$9_{\texttt{hex}}$	$1001_{\texttt{two}}$	$d_{\texttt{hex}}$	$1101_{\texttt{two}}$
$2_{\texttt{hex}}$	$0010_{\texttt{two}}$	$6_{\texttt{hex}}$	$0110_{\texttt{two}}$	$a_{\texttt{hex}}$	$1010_{\texttt{two}}$	$e_{\texttt{hex}}$	$1110_{\texttt{two}}$
$3_{\texttt{hex}}$	$0011_{\texttt{two}}$	$7_{\texttt{hex}}$	$0111_{\texttt{two}}$	$b_{\texttt{hex}}$	$1011_{\texttt{two}}$	$f_{\texttt{hex}}$	$1111_{\texttt{two}}$

Table 4.8: Table of hexadecimal numbers and their corresponding binary digits.

As an example we can convert the add instruction machine code shown in Section 4.6 to hexadecimal. Using the lookup table shown in Table 4.8, we can easily convert the machine code to hex and back, as illustrated in Figure 4.2. In many programming languages hexadecimal numbers are usually preceded by a `0x` such that the hex value shown in Table 4.2 would look like `0x00d60d33`.

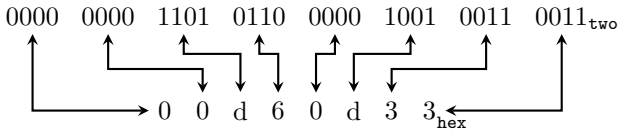


Figure 4.2: Using Table 4.8 we can convert the 32 binary string to hexadecimal and vice versa.

4.7 Operators

In Section 4.6 we introduced the machine code representation for instructions. We saw that within the 32-bit strings there existed various fields responsible for instruction identification, target registers and constant fields. In this section we take a look at some useful operations, outlined in Table 4.9, to address strings of bits or even individual bits within an instruction.

4.7.1 Shifts

To move bits within a string of bits left or right, we make use of the shift operations. There exist two types of shifts, arithmetic and logical. The logical shift moves strings of bits in a given direction e.g. we have the 16 bits

$$0000\ \ 0000\ \ 1101\ \ 0110$$

Decimal	C operators	RISC-V Instruction
Logical left shift	<<	`sll`, `slli`
Logical right shift	>>	`srl`, `srli`
Arithmetic right shift	>>	`sra`, `srai`
Bitwise AND	&	`and`, `andi`
Bitwise OR	|	`or`, `ori`
Bitwise XOR	^	`xor`, `xori`
Bitwise NOT	~	`xori`

Table 4.9: Table of logical and bitwise operators in C and corresponding RISC-V Instruction.

logical shifting left by 9 bits

$$\not{1} \leftarrow \overbrace{1010}\ \overbrace{1100}\ \overbrace{0000}\ \overbrace{0000} \leftarrow 0_{\texttt{two}}$$

logical shifting right by 9 bits

$$0 \rightarrow \overbrace{0000}\ \overbrace{0000}\ \overbrace{0101}\ \overbrace{0110} \rightarrow \not{0}_{\texttt{two}}$$

When shifting left the bit at the end, bit number 15, gets discarded and at the opposite end, bit number 0, a zero gets inserted. The exact opposite happens when logical shifting right. Notice that the logical shift does not preserve sign, this is what the arithmetic shift is for. Again with the same string of bits

$$0000\ \ 0000\ \ 1101\ \ \ 0110$$

we perform an arithmetic shift to the left by 9 bits

$$\not{1} \leftarrow \overbrace{1010}\ \overbrace{1100}\ \overbrace{0000}\ \overbrace{0000} \leftarrow 0_{\texttt{two}}$$

then performing a arithmetic shift to the right by 9 bits

$$1 \rightarrow \overbrace{1111}\ \overbrace{1111}\ \overbrace{1101}\ \overbrace{0110} \rightarrow \not{0}_{\texttt{two}}$$

When shifting left the bit at the end, bit number 15, gets discarded and at the opposite end, bit number 0, a zero gets inserted. However going right something different happens this time, as the arithmetic shifts preserves sign, so the inserted bit depends on the end bit (bit number 15 here). If the end bit is 1, ones will get inserted and if its 0, zeros will get inserted when shifting right.

We see that the arithmetic shift to the left is exactly the same as the logical left shift, therefore only logical left instruction is needed. A beneficial shortcut to know about is shifting bits in the left direction `i` times multiplies the number by $2^{\texttt{i}}$ times. Shifting right therefore is the opposite and divides the number by $2^{\texttt{i}}$.

4.7.2 Bitwise logical operations

Just like the logical operators `AND` and `OR`, which compares two values e.g $7_{\texttt{ten}}$ `AND` $4_{\texttt{ten}}$ = `FALSE`. We have bitwise versions of the logical operators `AND` and `OR`. The bitwise operators work a little different than their logical versions. They compare every bit in a string of bits, so using the operator on the decimals seven and four as before would result in

$$\begin{array}{r} 0111_{\texttt{two}} \\ \&\,0100_{\texttt{two}} \\ \hline 0100_{\texttt{two}} \end{array}$$

You have to be careful using the bitwise logical operators, as they give unintuitive results when used with decimals as you can see $7_{\texttt{ten}}$ `&` $4_{\texttt{ten}} = 4_{\texttt{ten}}$.

This operator is very useful when used in conjunction with the logical right shift operator for isolating fields in various strings of bits, for example an instruction. Say we wanted to isolate the register field in the `add` instruction,

```
0000000 01101 01100 000    10010 0110011
funct7  rs2   rs1   funct3 rd    add
```

you simply shift the instruction to the right by 15 bits, such that bits numbers 0-4 now contains register 1 field bits.

```
00000000000000000000001101 01100 # shifted right by 15 bits
```

Hereafter you perform a bitwise `AND` instruction with string of bits that contains ones only at position 0-4

```
   00000000000000000000001101 01100
 & 00000000000000000000000000 11111
-----------------------------------
   00000000000000000000000000 01100
```

everything but the first 5 bits gets canceled out and we end up with the contents of the rs1 field. Our 32-bits now contain whole slew of zeros left of the fourth bit. To place string of bits back into the 32-bits we can use the bitwise `OR` operator for example

```
   00000000000000000000000000 01100
 | 00000001010110000000000000 00000
-----------------------------------
   00000001010110000000000000 01100
```

Lastly we have the bitwise `XOR` and `NOT` operators. Since `XOR` can be used to perform a `NOT` operation, only the `XOR` operator has a dedicated instruction in the RISC-V architecture. To emulate the bitwise `NOT` operation, which inverts every bit, using the `XOR` operator, you simply compare is to -1, for example using our result from the `OR` operation

```
   00000000101011000000000000 01100
 ^ 11111111111111111111111111 11111
------------------------------------
   11111111010100111111111111 10011
```

4.8 Branching Instructions

To produce advanced programs that do more than arithmetic calculations, *branching* instructions are supported by the basic RISC-V instruction set. Branching instructions allows the processor to execute instructions not located immediately after the current one based on some condition.

For example to move 5 instructions back if values stored in registers 17 and 18 are equal the `beq` instruction would be used. In assembly this would be written like so

```
1 ...
2 beq x17, x18, -5
```

In higher level programming languages the abstractions for the branching instructions are commonly represented by `if/else` statements or `for/while` loops. In an `if/else` statement the branching instruction would be used to go to a specific place in the program if some condition is met. In a `for/while` loop branching instructions would be used to jump back and repeat the same part of a program until the condition is met. A subset of the branching instructions are outlined in Table 4.10.

Name	Assembly	Description (in C)
Branch If Equal	beq rs1, rs2, imm	if(rs1 == rs2) PC += imm
Branch Not Equal	bne rs1, rs2, imm	if(rs1 != rs2) PC += imm
Branch Less Than	blt rs1, rs2, imm	if(rs1 < rs2) PC += imm
Branch Greater Than Or Equal	bgeu rs1, rs2, imm	if(rs1 $\geq$ rs2) PC += imm

Table 4.10: This table is a subset of the RISC-V assembly language branching instructions. For the complete base instruction set, please refer to the RISC-V reference card in the appendix.

4.9 Jumping Instructions

When creating larger programs often same the procedure is used multiple times. An example of this could be calculating $\sin(x)$ multiple places in the code. Instead of rewriting the

algorithm for calculating $\sin(x)$ every time it is used, a function can be written once, which we call instead every time it is needed.

In assembly jumping instructions are used for this purpose. Two jumping instructions are supported in the basic instruction set, `jump and link` and `jump and link register`. They have been outlined in Table 4.11.

Assuming that the $\sin(x)$ is implemented at some address, jumping to the procedure would be done with a `jump and link` instruction like so

```
1 ...
2 jal x1, sinxprocedureaddress
```

here the address for the next instruction after the current is saved in register `x1`. To return from the $\sin(x)$ procedure the `jump and link register` instruction is used as so

```
1 ...
2 jalr x0, x0, x1
```

taking advantage of the hard-wired zero in register 0.

Name	Assembly	Description (in C)
Jump and link	jal rd, imm	rd = PC + 4; PC += imm
Jump and link register	jalr rd, rs1, imm	rd = PC + 4; PC = rs1 + imm

Table 4.11: This table shows the RISC-V assembly language jumping instructions. For the complete base instruction set, please refer to the RISC-V reference card in the appendix.

4.10 Compiling simple C code to assembly

To close this chapter off an example of a simple `C` programs compilation into assembly will be shown.

For this example a loop that increments the value, stored in variable `a` is constructed

```
1 int a = 0;
2 for (int i=0; i<10; i++) {
3     a+=i;
4 }
```

Using any RISC-V compiler the `C` program can get compiled into its assembly equivalent. It will however look quite different from the code about to be shown depending on the compiler, but the train of thought should still be the same.

In the following the first line adds variable `a` to register 6 with value 0, the second line add the index `i` to register 7 with value 0 and constant `c=10` is added to register 28.

Line 4 finds the sum between `a` and `i`, which gets stored back into register 6 and line 5 increment `i` by 1.

Lastly to emulate the `for` loop a branch if less than, `blt`, instruction has been added to go back to line 4 as long as `i<10`.

```
1 ori x6, x0, 0   # a=0
2 ori x7, x0, 0   # i=0
3 ori x28, x0, 10 # c=10 (the value i has to be less than in the for loop)
4 add x6, x6, x7  # a=a+i
5 addi x7, x7, 1  # i=i+1
6 blt x7, x28, -2 # if(i < 10) go two instructions back
```

Chapter 5

The RISC-V processor

We are finally at a point where we can start building the RISC-V CPU in SME. At the start of this project I followed the approach described in chapter 4 of the RISC-V book [3], but when a functioning CPU was made I quickly came to the realization that the version described in the book was quite inadequate. It only supported a handful instructions, which is not enough to run any but the simplest of programs.

Therefore I had to sit down and design a version with the requirement of supporting all 49 basic RISC-V instructions `RV64I`. This design of course took foundation in the book version, with improvements from the lessons learned from the first implementation.

Throughout this chapter we will go through the new design, which can be found here[1].

5.1 Single Cycle RISC-V Units

In this section we will explain the function of each of the 12 unique units, used in the implementation of the CPU. This design is single cycle, meaning that only one instruction is executed per clock cycle.

5.1.1 Program Counter

The *program counter*, PC for short, keeps track of where in the program we are located and is a fairly simple unit. It can we thought of as a single register, which holds the address of the current instruction.

To implement the PC unit we create a clocked SME process. It needs to be clocked, meaning the unit will activate on a rising clock edge, as it will part of a closed loop circuit, when we later connect the units. If no unit is clocked in a closed loop, there is no way of knowing, where to begin sending signals. Therefore the PC was chosen to be clocked, as it seemed like the most logical place to start the signal propagation.

[1] https://github.com/DanielRamyar/Master_Thesis/tree/master/SME_Implementations/SingleCycleRISCV2

The PC process contains a single `ulong` integer, which will hold the instruction address. The input bus contains the address of the next instruction and the output bus the current. You may ask how the process outputs the current address, when it reads the next address first. Remember that the process is clocked, so when it reads the next address input, it will actually contain the address calculated in the previous clock cycle, as the bus hasn't been updated yet. This would then be the correct address in the current clock cycle.

The PC process has been illustrated in Figure 5.1 and a code segment is shown in Listing 5.1.

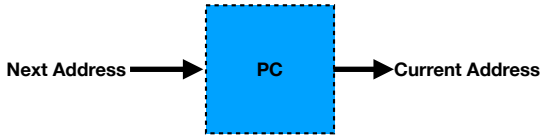


Figure 5.1: Illustration of the clocked `program counter` process having the next address as input and current address as output. The dashed square indicates a clocked process.

```
1  ...
2      ulong address_hold;
3
4      protected override void OnTick() {
5          address_hold = m_Input.Address;
6          Output.Address = address_hold;
7      }
```

Listing 5.1: A slice of the PC unit SME code, which contains variable that holds the input address. On every cycle edge it then holds and outputs the current address.

5.1.2 Instruction Memory

The *instruction memory*, IM for short, contains the program to be run on the CPU. To implement the instruction memory we create a SME process. It contains a byte array with the instructions to be run. A byte array was chosen to respect the conventions discussed in 4.3. This also has the added benefit of having a built-in C# function to read a binary file, which automatically puts the instructions contained within the file, in the correct array format.

The instructions can also be hand written when declaring the array. Since an instruction is 32-bits, 4 elements in the array are needed to form an instruction, where index 0-3 contains the first instruction.

The instruction memory process has a single input from the program counter, which it uses access the correct instructions. In each cycle we first check whether the address given lies within the instruction array range, if not we shut down the CPU. We then construct an instruction, using methods discussed in 4.7, to a temporary variable.

The register fields are then sliced out of this variable and put in the corresponding output buses, `Read RS1, RS2` and `Write RS`. The full instruction is also outputted to its own bus, `Instruction`. Lastly we tell the simulation process to keep the CPU running by asserting the `CPU` bus. The instruction memory process has been illustrated in Figure 5.2 and a code segment is shown Listing 5.2 explaining parts of the code.

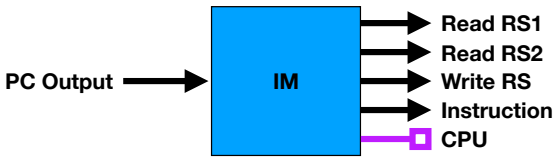


Figure 5.2: Illustration of the `instruction memory` process, taking the PC output as input and outputting to the 5 buses, `Read RS1 and RS2`, `Write RS`, `Instruction` and `CPU`.

```
1  ...
2      private readonly byte[] Instruction_Memory =
           System.IO.File.ReadAllBytes("/Users/danielramyar/Downloads/fibo.bin");
3
4      protected override void OnTick() {
5          ulong temp_address = m_input.Address;
6          uint temp_instruction;
7
8          if (temp_address >= 0 && temp_address < (uint)Instruction_Memory.Length) {
9              temp_instruction = 0u | (uint)Instruction_Memory[temp_address]     << 24
10                                   | (uint)Instruction_Memory[temp_address + 1] << 16
11                                   | (uint)Instruction_Memory[temp_address + 2] << 8
12                                   | (uint)Instruction_Memory[temp_address + 3];
13
14             m_Instruction.Current = temp_instruction;
15             m_read_1.Address = (uint)temp_instruction >> 15 & (uint)31;
16             m_read_2.Address = (uint)temp_instruction >> 20 & (uint)31;
17             m_write.Address  = (uint)temp_instruction >> 7  & (uint)31;
18
19             m_CPU.Running = true; // Keep CPU running
20         }
21         else {
22             temp_instruction = 0u; // No Instruction
23             ...  // Same as in the if statement
24             m_CPU.Running = false; // Turn of cpu no more instructions
25     }
```

Listing 5.2: A slice of the Instruction Memory unit SME code. It contains a single byte array, which holds all the instructions to be run. First we check whether the given address to be accessed lies within instruction array, if not we shut down the CPU. We then use the address to access the correct array elements and create a temporary variable, which contains the instruction, as shown in lines 9-12. Hereafter we slice out the fields in the instruction and place the values in the correct busses. Lastly we tell the simulator to keep the CPU running using the CPU bus.

5.1.3 Next instruction Unit

To calculate the address of the next instruction in the queue I have created a unit called `Next`. This unit is very simple, as its only function is to increment the value found in the PC output bus by 4. We increment by 4, since each instruction is 32 bits long and since our instructions are contained in a byte array, we need to move 4 bytes every time we want to access the following instruction. For example if we are placed at index 0, we would have to go to index 4 to access the next instruction (index 0-3 contains instruction 1 and index 4-7 the next).

The next instruction process has the the output from the program counter as input and sends the incremented address to the `Next Output` bus. The `Next` process is illustrated in 5.3 and a code segment is shown in Listing 5.3.

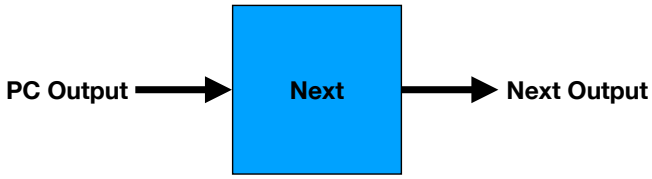


Figure 5.3: Illustration of the `next` process, taking the `PC output` as input and outputs the next instruction address to the `next output` bus.

```
1  ...
2      ulong temp;
3
4      protected override void OnTick() {
5          temp = m_Input.Address + 4;
6          Output.Address = temp;
7      }
```

Listing 5.3: A slice of the `Next` process SME code. Here we declare a temporary varible, which contains the program counter output. We increment the temporary variable by four and place it in the output bus.

5.1.4 Register

The `register`, or RS for short, is a small data storage location for operands of instructions among others (see Table 4.2). It can be accessed by specifying a register address in an instruction.

Since R-type instructions use two operands to execute an arithmetic operation, the register has two input lines carrying the address to the specified registers and two output lines carrying the data in the read registers.

Furthermore the result needs to get stored, therefore we need two additional lines one carrying the result of an operation and the other an address for the storage location. To

avoid any unintentional write backs to the register, we introduce a write control line, which is asserted if write back is intended.

In SME the register is declared as a process containing a signed 64-bit array with 32 entries. To ensure the latest data is always read, we write to the register as the first thing. Since register 0 is hard-wired to zero according to the conventions, we make sure that the write back address lies within the range 1-31 and that the write control is asserted. Lastly we check if the read addresses lies within the range of the register and then output the data in the read registers to the corresponding buses. The `register` process is illustrated in Figure 5.4 and a code segment shown in Listing 5.4.

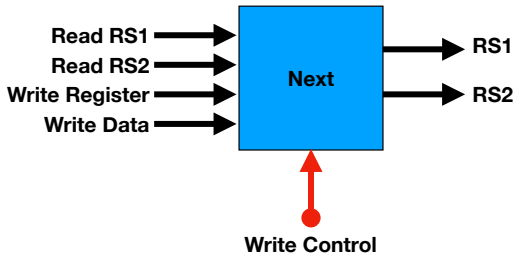


Figure 5.4: Illustration of the `register` process. It gets its information on which register addresses to read from `Read RS1, RS2` buses. To write data the register uses `write register` bus to know which register to write to, the `write data` bus contains the data to be written and the `write control` bus authorize whether the register should write or not.

```
1  ...
2      private readonly long[] m_register = {0, 0, 0, 0, 0, 0, 0, 0,
3                                            0, 0, 0, 0, 0, 0, 0, 0,
4                                            0, 0, 0, 0, 0, 0, 0, 0,
5                                            0, 0, 0, 0, 0, 0, 0, 0,};
6
7      protected override void OnTick() {
8          if (m_write_control.Enable == true && m_write.Address > 0 && m_write.Address < 32) {
9              m_register[m_write.Address] = m_write_data.Data;
10         }
11         if (m_read_1.Address >= 0 && m_read_1.Address < 32) {
12             output_1.Data = m_register[m_read_1.Address];
13         }
14         if (m_read_2.Address >= 0 && m_read_2.Address < 32) {
15             output_2.Data = m_register[m_read_2.Address];
16         }
17     }
```

Listing 5.4: A slice of the `register` process SME code. The register is declared as a signed 64-bit array with 32 elements. In every clock cycle the process checks if the write control signal is asserted. If it is, we check if the address to be written lies within the register range and is above 0, as this register is reserved to be hard-wired zero. Lastly the process reads the two specified registers and if they lie within the register range, they get outputted to the corresponding buses.

5.1.5 Arithmetic Logic Unit (ALU)

The `Arithmetic Logic Unit`, or ALU for short, is the computational powerhouse of the CPU. It is here all arithmetic, logical, shift and bitwise operations are computed. The ALU has two inputs that consists of the operands of an operation. The output is then the result. To determine which operation to perform we additionally have a control line with the operation code or opcode.

In the SME implementation all the operations are enclosed in a `switch` statement, that is controlled by the opcode to determine which operation to execute. The ALU is illustrated in Figure 5.5 and a code segment is shown in Listing 5.5.

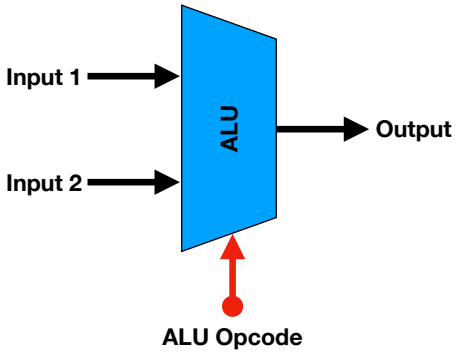


Figure 5.5: Illustration of the `ALU` process. The two input contains the operands of the operation to be executed and the output is the result. The ALU opcode (in red) controls which operations the ALU will perform.

```
...
    protected override void OnTick() {
        switch (m_ALUOP.Value) {
            case 0:
                output.Value = m_ALU_In_1.Data + m_ALU_In_2.Data;   // ADD
                break;
            case 1:
                output.Value = m_ALU_In_1.Data - m_ALU_In_2.Data;   // SUB
                break;
            case 2:
                output.Value = m_ALU_In_1.Data & m_ALU_In_2.Data;   // AND
                break;
            case 3:
                output.Value = m_ALU_In_1.Data | m_ALU_In_2.Data;   // OR
                break;
            case 4:
                ...
        }
    }
```

Listing 5.5: A slice of the `ALU` process SME code. The ALU consists of a large `switch` statement, with cases for each operation to be performed, which is controlled by the ALU operation code (`opcode`).

5.1.6 Immediate generator

The `Immediate Generator`, or IMMGEN for short, extracts immediate fields from instructions. This is a pretty large unit since it needs to know where the immediate field is located in every instruction. To make matters worse some immediate fields in the instructions are totally scrambled, so we have to be really careful on how we reconstruct them.

To start we need to be able to identify which type of instruction we are dealing with to know where the immediate field is located. Luckily almost all types of instructions have a unique operation code (for example the I-type has opcode 19), we therefore use a `switch` statement in the SME code with the opcode in the instruction as a branching condition to identify the instruction type.

Some instructions of same type, only use the lower 6 bits in the immediate field, such as the `slli` instruction. To accommodate this fact we make use of another `switch` statement, within the specific case of the first `switch` statement, that uses the `funct3` field as branching condition, since this is the unique identifier within the same type of instruction (for example `opcode` 19, `funct3` 1 is the `slli` instruction).

When we know where the immediate field is located it is time to extract it. This is done by shifting the instruction in question to the right, such that the first bit of the immediate field is located at bit index 0 of the 32-bit instruction. Hereafter we do a bitwise AND operation to remove all but the necessary bits (see Section 4.7). It should be noted that in cases, where the immediate field only is 12-bits long we need to perform a shift to the left and then right to retain the sign bit, as 12-bit numbers are not supported by `C#` and therefore does not happen automatically, when cast to `long` type.

If we deal with `B-` and `J-type` instructions, we need to be really careful when we reconstruct the immediate field, as the immediate field bits do not lie in a trivial manner. An important thing to note is that in these types of instructions the first bit is not supplied and therefore should not get shifted all the way to the 0'th bit index but should get shifted to the 1'st bit index. This firstly ensures that we only jump by an even number of bytes, as this is where the base index is for the instructions and secondly it effectively doubles the range we can jump instructions.

When the immediate field is constructed it simply gets outputted as a signed 64-bit number. The immediate generator is illustrated in Figure 5.6 and a code segment is shown in Listing 5.6.

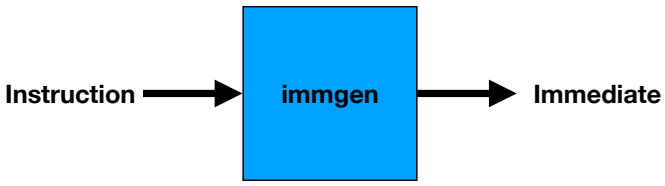


Figure 5.6: Illustration of the `immediate generation` process, taking an instruction as input. It then extracts the immediate field, which it outputs.

```
1  ...
2      protected override void OnTick() {
3          uint Opcode = m_instruction.Current & (uint)0x7F;
4          uint funct3 = m_instruction.Current >> 12 & (uint)0x7;
5
6          switch(Opcode) {
7              ...
8              case 19:                                          // I-format
9                  switch (funct3) {
10                     ...
11                     case 5:
12                         temp1 = m_instruction.Current >> 20 & (uint)0x3F;
13                         temp0 = (long)temp1;
14                         output.Immediate = temp0;
15                         break;
16                     default:
17                         temp1 = m_instruction.Current >> 20 & (uint)0xFFF;
18                         short temp5 = (short)((short)(temp1 << 4) >> 4);
19                         temp0 = (long)temp5;
20                         output.Immediate = temp0;
21                         break;
22                 }
23             break;
24             case 27:                                          // I-format Word
25                 ....
26         }
27     }
```

Listing 5.6: A slice of the `IMMGEN` process SME code. We first extract the opcode and funct3 fields from the instruction and put them in a variable. We then use a `switch` statement and the opcode to determine what type of instruction we are dealing with. Since some instructions of same type like `slli` only use the lower 6 bits in the immediate field (shamt field), we need another switch statement to tell these apart using the funct3 field. Lastly we construct the immediate and output it . Note that in line 18 we make use of a little hack to retain the sign bit of a 12 bit number since it is not supported in `C#`

5.1.7 Data Memory

The `Data Memory`, or DM for short, is the main data storage unit of the CPU, as it can contain much more data than the register. The data memory is very similar to the instruction memory, as it is also little-endian addressed and is constructed from a byte array. The processor can either read or write from the data memory unit once per clock cycle.

Because of this fact the data memory needs control signals, which manages read and write access. To support data of varying size an additional control signal has been added to distinguish between double word, word, short and byte access. Since unsigned versions of the `load` instructions also needs to be supported, this functionality is added to the size control signal.

As the data memory communicates with the register, which is big-endian addressed and every element is 64-bit a translation needs to happen between these formats.

In SME a process is created that contains a byte array. To support 64-bit numbers the array has to be larger than 8 bytes if a single number has to be stored. Otherwise the sizing is irrelevant if made big enough and is divisible by eight. In this implementation the array is 2000 bytes long.

To distinguish between the read and write states an `if/else` statement is used. For the branching condition, two control signals are used to determine whether a read or write should be performed. When reading or writing the size of the data needs to be known, such that a correct translation between big- and little-endian formats can take place. To this end a `switch` statement is used, with cases for each data size scenario. Furthermore cases for handling unsigned reads has been added.

The read and write address gets calculated by the ALU and the data is supplied by the second register output line. If the read control is enabled the data memory will output the specified data. The data memory is illustrated in Figure 5.7 and a code segment is shown in Listing 5.7.

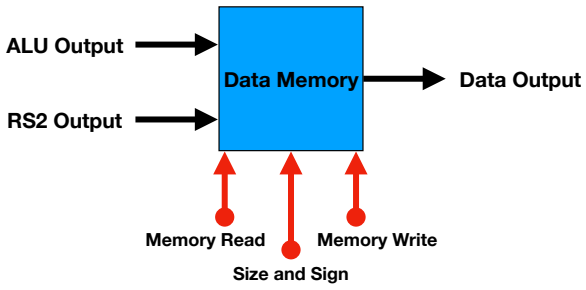


Figure 5.7: Illustration of the `Data Memory` process. The target address is calculated in the ALU and data to be written outputted from register 2, which the unit takes as input. Controls signals are marked in red and consists of `Memory Read`, `Memory Write` and `Size and Sign`. It then outputs the read data or 0 if both `Memory Read` and `Memory Write` are deasserted.

```
...
    byte[] Data_Memory = new byte[2000];

    protected override void OnTick() {
        if (m_MemRead.Enable) {
            ...
            switch (m_SizeAndSign.Value) {
                ...
                case 1: // load word
                    temp0 =
                    Data_Memory[m_Address.Value] | Data_Memory[m_Address.Value + 1] << 8
                                                 | Data_Memory[m_Address.Value + 2] << 16
                                               | (sbyte)Data_Memory[m_Address.Value + 3] << 24;
                    break;
                case 2: // load short
                    ...
            }
            output.Data = temp0;
        }
        else if (m_MemWrite.Enable) {
            switch (m_SizeAndSign.Value) {
                ...
                case 2:
                    Data_Memory[m_Address.Value]     = (byte)(m_Data_input.Data      &
                        0xFF);
                    Data_Memory[m_Address.Value + 1] = (byte)(m_Data_input.Data >> 8 &
                        0xFF);
                    break;
                case 3:
                    Data_Memory[m_Address.Value] = (byte)(m_Data_input.Data & 0xFF);
                    break;
            }
        }
        else {
            output.Data = 0;
        }
    }
```

Listing 5.7: A slice of the `Data Memory` process SME code. Similar to the instruction memory, the data memory consists of a byte array, which we made 2000 elements long in this case. As there can only be read or written to the data memory once per clock cycle, we simply construct an `if/else` statement, which uses control signals to determine the procedure to be done. Both the read and write procedures needs to know the size and sign of the data to be loaded or written. Therefore a `switch` statement has been added to both outcomes, which uses `SizeAndSign` control signal to choose between the cases. When reading the data from memory we need to remember that it is little-endian addressed so before we output the value the bits need to be shuffled around in the correct order, so the correct value is added to the register, as that is big-endian. When writing the data the opposite need to happen, so a big-endian value need to be converted to a little-endian one. If both read and write control signals are deasserted I choose to output 0.

5.1.8 Go to Unit

To support the RISC-V branching and jump instructions a special unit called `Go To` was created. The purpose of this unit is to do the logical comparison between the specified registers. For example if the `branch if equal` (`BEQ`) instruction is given, the unit will check if the values in the specified registers are equal and outputs logical `true` or `false` depending on the result.

In the SME implementation a simple `switch` statement is used. Each case represent the comparison to be done by a given instruction, where the case branching is controlled by a control line. The result is then outputted. The Go To unit is illustrated in Figure 5.8 and a code segment is shown in Listing 5.8.

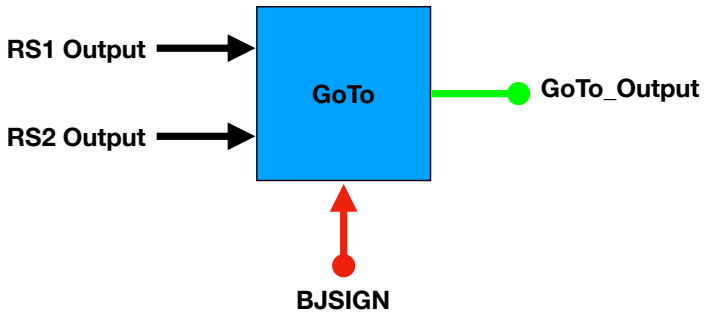


Figure 5.8: Illustration of the `Go To` process. As input it uses the two output lines from the register. It has one control line in red and one output line in green.

```
1  ...
2      protected override void OnTick() {
3          switch (m_BJSIGN.Value) {
4              case 0:
5                  output.Value = (m_RS1.Data == m_RS2.Data) ? true:false; // BEQ
6                  break;
7              case 1:
8                  output.Value = (m_RS1.Data != m_RS2.Data) ? true:false; // BNE
9                  break;
10             case 2:
11                 ...
12         }
13     }
```

Listing 5.8: A slice of the `Go To` process SME code. The unit consists of a `switch` statement, where each case does the comparison relevandt to the branching instruction. The conditional operator `?` is used for the comparison (the syntax goes as follows `condition ? If true return this : If false return this`).

5.1.9 Multiplexer

To select between multiple lines we make use of a `multiplexer`, or mux. It is a fairly simple process that only consists of a `switch` statement, where each case corresponds to an input to output. The cases then gets selected depending on the control line. The multiplexer is illustrated in Figure 5.9 and a code segment is shown in Listing 5.9.

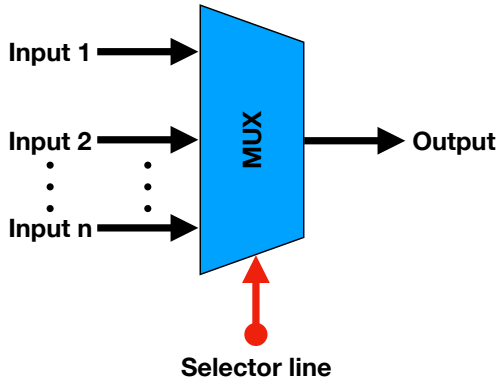


Figure 5.9: Illustration of the `Multiplexer` process. An arbitrary amount of input can be given, where the output i selected by the red control line.

```
1  ...
2      protected override void OnTick() {
3          switch (SelectBus) {
4              case 0:
5                  Mux_output = input1;
6                  break;
7              case 1:
8                  Mux_output = input2;
9                  break;
10             case 2:
11                 Mux_output = input3;
12                 break;
13         }
14     }
```

Listing 5.9: An example of a `Multiplexer` process SME code. The whole multiplexer simply consists of a `switch` statement that selects an input to output dependent on `SelectBus`.

5.1.10 Write Back Unit

When wire-ring the processor there is going to form a second closed loop between the register and data memory. As with the closed loop between the program counter and Next unit, there has to be a clocked process somewhere in the loop. For this reason a clocked write back process is introduced. It works exactly the same as the program counter, which read the contents of a bus that contained data from the previous clock cycle and got outputted

unchanged.

It may seem that this could cause problems, but as the data output from this unit gets written to the register as the first thing, instructions will never access old data. The `Write Back` unit is illustrated in Figure 5.10 and a code segment is shown in Listing 5.10.

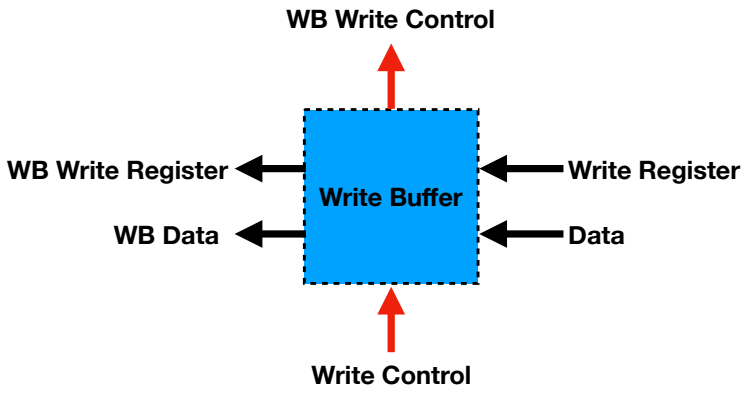


Figure 5.10: Illustration of the `Write Back` process. It has 3 input `Data`, `Write Register` and `Write control`. These input gets outputted unchanged to their respective output buses. The dashed square indicates a clocked process.

```
1  ...
2      long WB_Data_Hold;
3      uint WB_RegisterWrite_Hold;
4      bool WB_WriteControl_Hold;
5
6      protected override void OnTick() {
7          WB_Data_Hold = m_write_data.Data;
8          WB_RegisterWrite_Hold = m_write_register.Address;
9          WB_WriteControl_Hold = m_write_control.Enable;
10
11         m_WB_Data.Data = WB_Data_Hold;
12         m_WB_WriteRegister.Address = WB_RegisterWrite_Hold;
13         m_WB_WriteControl.Enable = WB_WriteControl_Hold;
14     }
```

Listing 5.10: A slice of the `Write Back` process SME code. It contains 3 variables, which will hold the incomming values, which gets outputted unchanged in the same clock cycle.

5.1.11 AND gate unit

To make use of a trick inspired from the RISC-V implementation in chapter 4 of [3] an AND gate unit has been implemented. The purpose of this unit is to check whether branching

or jumping instructions are given. If such instructions are taking place a control signal is asserted. The AND gate only returns true if the branching condition from the `Go To` gate returns true. This result can now be feed to a multiplexer which chooses whether to run next instruction or jump in the instruction memory. The `AND` unit is illustrated in Figure 5.11 and a code segment is shown in Listing 5.11.

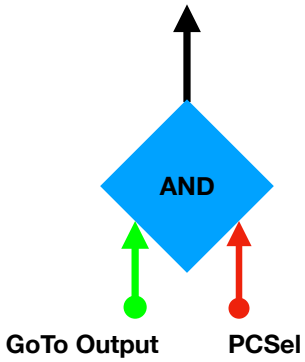


Figure 5.11: Illustration of the `AND gate` process. It has two input one from the `Go To` unit and one from the `control unit` named `PCSel`. The output is then the result of the logical AND operation between the two input.

```
1  ...
2      protected override void OnTick() {
3          Output.Value = m_Input_1.Value && m_Input_2.Value;
4      }
```

Listing 5.11: A slice of the `AND gate` process SME code. It simply calculates the logical AND of two input and outputs the result.

5.1.12 Control

The final unit that is to be shown is the `Control` unit. This is quite a substantial unit, as its function is to correctly set the values for the control lines for all 49 instructions. What the values look like will be discussed in Section 5.2. For now the SME implementation will be covered.

This unit uses a similar solution for instruction detection, as the `Immediate Generator` (see Section 5.1.6). To identify the current instruction the `opcode`, `funct3` and `funct7` fields are extracted from the 32-bit instruction input and put in their respective variables. All 3 fields are used this time, as every instruction needs to be identified to correctly set the control lines.

Multiple `switch` statements are layered for instruction detection. The first `switch` uses the `opcode` to determine the type of instruction that is currently running. Say a R-type instruction has been identified. Most control lines can be set here, as only the control line for the ALU is different between the R-type instructions. To set the last control line a second

`switch` statement is used with the `funct3` field as branching condition to identify which R-type instruction is given. Since some R-type instructions has the same `funct3` code, a third `switch` statement is required to differentiate these.

When an instruction as been identified the control lines are set to the correct values and outputted. The `Control` unit is illustrated in Figure 5.12 and a code segment is shown in Listing 5.12.

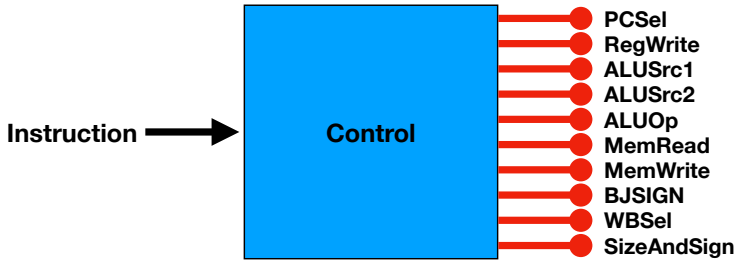


Figure 5.12: Illustration of the `Control` process. It takes a 32-bit instruction as input and sets the control lines for the respective instruction.

```
...
    protected override void OnTick() {
        uint Opcode = m_instruction.Current       & (uint)0x7F;
        uint funct3 = m_instruction.Current >> 12 & (uint)0x7;
        uint funct7 = m_instruction.Current >> 25 & (uint)0x7F;
        switch (Opcode) {
            ...
            case 51:                              // R-format
                PCSel.Value = false;
                RegWrite.Enable = true;
                ALUSrc1.Enable = false;
                ALUSrc2.Enable = false;
                BJSIGN.Value = 0;
                SizeAndSign.Value = 0;
                MemWrite.Enable = false;
                MemRead.Enable = false;
                WBSel.Value = 0;

                switch (funct3) {
                    case 5:
                        switch (funct7) {
                            case 0:                           // ADD
                            ALUOP.Value = 0;
                            break;
                        case 32:                          // Subtract
                            ALUOP.Value = 1;
                            break;
                        }
                        break;
                    ...
            ...
        }
    }
```

Listing 5.12: A slice of the `Control` process SME code. First 3 varibles are declared for the extracted `opcode`, `funct3` and `func7` fields. They are then used in the following `switch` statements to correctly identify the given instruction. Finally the control lines are set with the values for the respective instruction.

5.2 Single Cycle RISC-V datapaths

5.2.1 RV64I Base Instructions Support

5.2.2 Supporting R-Format

5.2.3 Supporting I-Format

5.2.4 Supporting S-Format

5.2.5 Supporting B-Format

5.2.6 Supporting U-Format

5.2.7 Supporting J-Format

5.3 Debugging the instructions

5.3.1 Writing assembly to test instructions

5.3.2 Writing simple C code to run on RISC-V

Chapter 6

Conclusion and future work

Pipeline the processor, Add necessary elements to run linux on it, add an fft instruction

Appendix A

Unary Operators

Logical identity

Hereafter we have the logical identity operator which we will represent as the function $I(x)$. The logical identity operator takes an argument and returns it as is.

A logic table for the identity operator has been created and can be found in table A.1. In the first column we find our preposition p and its arguments. In the second column we find the return values of the identity operator with the prepositions as the argument $I(p)$.

p	$I(p)$
$true$	$true$
$false$	$false$

Table A.1: Logic table of the identity operator where the proposition p, which is either true or false, can be found in the first column. In the second column we find $I(p)$, which is the identity operator with p as its argument, and its return values.

Logical true

Next we have logical true which we will represent as the function $T(x)$. Logical true takes an argument and always returns true.

A logic table for the true operator has been created and can be found in table A.2. In the first column we find our preposition p and its arguments. In the second column we find the return values of the true operator with the prepositions as the argument $T(p)$.

p	$T(p)$
$true$	$true$
$false$	$true$

Table A.2: Logic table of the true operator where the proposition p, which is either true or false, can be found in the first column. In the second column we find $T(p)$, which is the true operator with p as its argument, and its return values.

Logical false

Lastly we have logical false which we will represent as the function $F(x)$. Logical false takes an argument and always return false.

A logic table for the false operator has been created and can be found in table A.3. In the first column we find our preposition p and its arguments. In the second column we find the return values of the false operator with the prepositions as the argument $F(p)$.

p	$F(p)$
$true$	$false$
$false$	$false$

Table A.3: Logic table of the false operator where the proposition p, which is either true or false, can be found in the first column. In the second column we find $F(p)$, which is the false operator with p as its argument, and its return values.

Appendix B

Binary Operators

Joint denial

Joint denial is represented by $\downarrow$ in mathematics or NOR in computer science and commonly referred to as the NOR operator. The NOR operator results in a true value only if both operands are false.

Using table 2.4 we see that the set of outputs which corresponds to this definition is column 15 and is summarized in table B.1.

Here we have the propositions p and q in the first two columns and all possible permutations between them in the following rows. The last column then shows the resulting value after doing the NOR operation between p and q.

p	q	$p \downarrow q$
$true$	$true$	$false$
$true$	$false$	$false$
$false$	$true$	$false$
$false$	$false$	$true$

Table B.1: Logic table of the NOR operator where p is the first proposition and q is the second. All possible permutations are then specified in each row for each proposition. The third column then shows the resulting value of the NOR operation between p and q.

In disjunctive normal form the NOR operator can be expressed in the following form

$$p \downarrow q = (\neg p \wedge \neg q) \tag{B.1}$$

using the procedure mentioned in chapter 2.1.

Alternative denial

Alternative denial is represented by $\uparrow$ in mathematics or NAND in computer science and commonly referred to as the NAND operator. The NAND operator results in a true value

only if one or more of the operands are false.

Using table 2.4 we see that the set of outputs which corresponds to this definition is column 5 and is summarized in table B.2.

Here we have the propositions p and q in the first two columns and all possible permutations between them in the following rows. The last column then shows the resulting value after doing the NAND operation between p and q.

p	q	$p \uparrow q$
$true$	$true$	$false$
$true$	$false$	$true$
$false$	$true$	$true$
$false$	$false$	$true$

Table B.2: Logic table of the NAND operator where p is the first proposition and q is the second. All possible permutations are then specified in each row for each proposition. The third column then shows the resulting value of the NAND operation between p and q.

In disjunctive normal form the NAND operator can be expressed in the following form

$$p \uparrow q = (p \wedge \neg q) \vee (\neg p \wedge q) \vee (\neg p \wedge \neg q) \tag{B.2}$$

using the procedure mentioned in chapter 2.1.

Logical biconditional

The logical biconditional is represented by $\leftrightarrow$ in mathematics or XNOR in computer science and commonly referred to as the exclusive NOR operator. The XNOR operator results in a true value only if both operands are either true or false.

Using table 2.4 we see that the set of outputs which corresponds to this definition is column 9 and is summarized in table B.3.

Here we have the propositions p and q in the first two columns and all possible permutations between them in the following rows. The last column then shows the resulting value after doing the XNOR operation between p and q.

p	q	$p \leftrightarrow q$
$true$	$true$	$true$
$true$	$false$	$false$
$false$	$true$	$false$
$false$	$false$	$true$

Table B.3: Logic table of the XNOR operator where p is the first proposition and q is the second. All possible permutations are then specified in each row for each proposition. The third column then shows the resulting value of the XNOR operation between p and q.

In disjunctive normal form the XNOR operator can be expressed in the following form

$$p \leftrightarrow q = (p \wedge q) \vee (\neg p \wedge \neg q) \tag{B.3}$$

using the procedure mentioned in chapter 2.1.

Tautology

The tautology operator is represented by $\top$ in mathematics which always returns a true value.

Using table 2.4 we see that the set of outputs which corresponds to this definition is column 1 and is summarized in table B.4.

Here we have the propositions p and q in the first two columns and all possible permutations between them in the following rows. The last column then shows the resulting value after doing the tautology operation between p and q.

p	q	$p \top q$
$true$	$true$	$true$
$true$	$false$	$true$
$false$	$true$	$true$
$false$	$false$	$true$

Table B.4: Logic table of the tautology operator where p is the first proposition and q is the second. All possible permutations are then specified in each row for each proposition. The third column then shows the resulting value of the tautology operation between p and q.

In disjunctive normal form the tautology operator can be expressed in the following form

$$p \top q = (p \wedge q) \vee (p \wedge \neg q) \vee (\neg p \wedge q) \vee (\neg p \wedge \neg q) \tag{B.4}$$

using the procedure mentioned in chapter 2.1.

Contradiction

The contradiction operator is represented by $\perp$ in mathematics which always returns a false value.

Using table 2.4 we see that the set of outputs which corresponds to this definition is column 16 and is summarized in table B.5.

Here we have the propositions p and q in the first two columns and all possible permutations between them in the following rows. The last column then shows the resulting value after doing the contradiction operator between p and q.

In disjunctive normal form the contradiction operator can be expressed in the following

p	q	$p \perp q$
$true$	$true$	$false$
$true$	$false$	$false$
$false$	$true$	$false$
$false$	$false$	$false$

Table B.5: Logic table of the contradiction operator where p is the first proposition and q is the second. All possible permutations are then specified in each row for each proposition. The third column then shows the resulting value of the contradiction operation between p and q.

form

$$p \perp q = p \wedge \neg p \tag{B.5}$$

Proposition P

We will define the operator Proposition P which results in a true value only if the first operand p is true.

Using table 2.4 we see that the set of outputs which corresponds to this definition is column 6 and is summarized in table B.6.

Here we have the propositions p and q in the first two columns and all possible permutations between them in the following rows. The last column then shows the resulting value after doing the proposition P between p and q.

p	q	$P(p,q)$
$true$	$true$	$true$
$true$	$false$	$true$
$false$	$true$	$false$
$false$	$false$	$false$

Table B.6: Logic table of the proposition P operator where p is the first proposition and q is the second. All possible permutations are then specified in each row for each proposition. The third column then shows the resulting value of the proposition P operation between p and q.

In disjunctive normal form the proposition P can be expressed in the following form

$$P(p,q) = (p \wedge q) \vee (p \wedge \neg q) \tag{B.6}$$

using the procedure mentioned in chapter 2.1.

Proposition Q

We will define the operator Proposition Q which results in a true value only if the second operand q is true.

Using table 2.4 we see that the set of outputs which corresponds to this definition is column 10 and is summarized in table B.7.

Here we have the propositions p and q in the first two columns and all possible permutations between them in the following rows. The last column then shows the resulting value after doing the proposition Q between p and q.

p	q	$Q(p,q)$
$true$	$true$	$true$
$true$	$false$	$false$
$false$	$true$	$true$
$false$	$false$	$false$

Table B.7: Logic table of the proposition P operator where p is the first proposition and q is the second. All possible permutations are then specified in each row for each proposition. The third column then shows the resulting value of the proposition P operation between p and q.

In disjunctive normal form the proposition Q can be expressed in the following form

$$P(p,q) = (p \wedge q) \vee (\neg p \wedge q) \tag{B.7}$$

using the procedure mentioned in chapter 2.1.

Negated P

We will define the operator negated P which results in a true value only if the first operand p is false.

Using table 2.4 we see that the set of outputs which corresponds to this definition is column 8 and is summarized in table B.8.

Here we have the propositions p and q in the first two columns and all possible permutations between them in the following rows. The last column then shows the resulting value after doing the negated P between p and q.

p	q	$\neg P(p,q)$
$true$	$true$	$false$
$true$	$false$	$false$
$false$	$true$	$true$
$false$	$false$	$true$

Table B.8: Logic table of the negated P operator where p is the first proposition and q is the second. All possible permutations are then specified in each row for each proposition. The third column then shows the resulting value of the negated P operation between p and q.

In disjunctive normal form the negated P can be expressed in the following form

$$\neg P(p,q) = (\neg p \wedge q) \vee (\neg p \wedge \neg q) \tag{B.8}$$

using the procedure mentioned in chapter 2.1.

Negated Q

We will define the operator negated Q which results in a true value only if the second operand q is false.

Using table 2.4 we see that the set of outputs which corresponds to this definition is column 11 and is summarized in table B.9.

Here we have the propositions p and q in the first two columns and all possible permutations between them in the following rows. The last column then shows the resulting value after doing the negated Q between p and q.

p	q	$\neg Q(p,q)$
$true$	$true$	$false$
$true$	$false$	$true$
$false$	$true$	$false$
$false$	$false$	$true$

Table B.9: Logic table of the negated Q operator where p is the first proposition and q is the second. All possible permutations are then specified in each row for each proposition. The third column then shows the resulting value of the negated operation between p and q.

In disjunctive normal form the negated Q can be expressed in the following form

$$\neg Q(p,q) = (p \wedge \neg q) \vee (\neg p \wedge \neg q) \tag{B.9}$$

using the procedure mentioned in chapter 2.1.

Material implication

Material implication is represented by $\rightarrow$ in mathematics. The material implication operator results in a false value only if the first operand p is true and second operand q is false.

Using table 2.4 we see that the set of outputs which corresponds to this definition is column 4 and is summarized in table B.10.

Here we have the propositions p and q in the first two columns and all possible permutations between them in the following rows. The last column then shows the resulting value after doing the material implication operation between p and q.

In disjunctive normal form the material implication operator can be expressed in the following form

$$p \rightarrow q = (p \wedge q) \vee (\neg p \wedge q) \vee (\neg p \wedge \neg q) \tag{B.10}$$

using the procedure mentioned in chapter 2.1.

p	q	$p \rightarrow q$
$true$	$true$	$true$
$true$	$false$	$false$
$false$	$true$	$true$
$false$	$false$	$true$

Table B.10: Logic table of the material implication operator where p is the first proposition and q is the second. All possible permutations are then specified in each row for each proposition. The third column then shows the resulting value of the material implication operation between p and q.

Converse implication

Converse implication is represented by $\leftarrow$ in mathematics. The converse implication operator results in a false value only if the first operand p is true and the second q is true.

Using table 2.4 we see that the set of outputs which corresponds to this definition is column 3 and is summarized in table B.11.

Here we have the propositions p and q in the first two columns and all possible permutations between them in the following rows. The last column then shows the resulting value after doing the converse implication operation between p and q.

p	q	$p \leftarrow q$
$true$	$true$	$true$
$true$	$false$	$true$
$false$	$true$	$false$
$false$	$false$	$true$

Table B.11: Logic table of the converse implication operator where p is the first proposition and q is the second. All possible permutations are then specified in each row for each proposition. The third column then shows the resulting value of the converse implication operation between p and q.

In disjunctive normal form the converse implication operator can be expressed in the following form

$$p \leftarrow q = (p \wedge q) \vee (p \wedge \neg q) \vee (\neg p \wedge \neg q) \tag{B.11}$$

using the procedure mentioned in chapter 2.1.

Material nonimplication

Material nonimplication is represented by $\not\rightarrow$ in mathematics. The material nonimplication operator results in a true value only if the first operand p is true and the second operand q is false.

Using table 2.4 we see that the set of outputs which corresponds to this definition is column 13 and is summarized in table B.12.

Here we have the propositions p and q in the first two columns and all possible permutations between them in the following rows. The last column then shows the resulting value after doing the material nonimplication operation between p and q.

p	q	$p \not\rightarrow q$
$true$	$true$	$false$
$true$	$false$	$true$
$false$	$true$	$false$
$false$	$false$	$false$

Table B.12: Logic table of the material nonimplication operator where p is the first proposition and q is the second. All possible permutations are then specified in each row for each proposition. The third column then shows the resulting value of the material nonimplication operation between p and q.

In disjunctive normal form the material nonimplication operator can be expressed in the following form

$$p \rightarrow q = p \wedge \neg q \tag{B.12}$$

using the procedure mentioned in chapter 2.1.

Converse nonimplication

Converse nonimplication is represented by $\not\leftarrow$ in mathematics. The converse nonimplication operator results in a true value only if the first operand p is false and the second operand q is true.

Using table 2.4 we see that the set of outputs which corresponds to this definition is column 14 and is summarized in table B.13.

Here we have the propositions p and q in the first two columns and all possible permutations between them in the following rows. The last column then shows the resulting value after doing the converse nonimplication operation between p and q.

p	q	$p \not\leftarrow q$
$true$	$true$	$false$
$true$	$false$	$false$
$false$	$true$	$true$
$false$	$false$	$false$

Table B.13: Logic table of the converse nonimplication operator where p is the first proposition and q is the second. All possible permutations are then specified in each row for each proposition. The third column then shows the resulting value of the converse nonimplication operation between p and q.

In disjunctive normal form the converse nonimplication operator can be expressed in the following form

$$p \leftarrow q = \neg p \wedge q \tag{B.13}$$

using the procedure mentioned in chapter 2.1.

Risc V Reference Card

Instruction Formats

31 25	24 20	19 15	14 12	11 7	6 0	
funct7	rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]		rs1	funct3	rd	opcode	I-type
imm[11:6]	imm[5:0]	rs1	funct3	rd	opcode	I-type*
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]				rd	opcode	U-type
imm[20|10:1|11|19:12]				rd	opcode	J-type

* This is a special case of the RV64I I-type format used by slli, srli and srai instructions where the lower 6 bits in the immediate are used to determine the shift amount (shamt). If slliw, srliw and sraiw are used it should generate an error if imm[6] $\neq 0$

RV64I Base Instructions

Name	Fmt	Opcode	Funct3	Funct7/ imm[11:5]	Assembly	Description (in C)
Add	R	0110011	000	0000000	add rd, rs1, rs2	rd = rs1 + rs2
Subtract	R	0110011	000	0100000	sub rd, rs1, rs2	rd = rs1 − rs2
AND	R	0110011	111	0000000	and rd, rs1, rs2	rd = rs1 & rs2
OR	R	0110011	110	0000000	or rd, rs1, rs2	rd = rs1 | rs2
XOR	R	0110011	100	0000000	xor rd, rs1, rs2	rd = rs1 ^ rs2
Shift Left Logical	R	0110011	001	0000000	sll rd, rs1, rs2	rd = rs1 ≪ rs2
Set Less Than	R	0110011	010	0000000	slt rd, rs1, rs2	rd = (rs1 < rs2)?1:0
Set Less Than (U)*	R	0110011	011	0000000	sltu rd, rs1, rs2	rd = (rs1 < rs2)?1:0
Shift Right Logical	R	0110011	101	0000000	srl rd, rs1, rs2	rd = rs1 ≫ rs2
Shift Right Arithmetic†	R	0110011	101	0100000	sra rd, rs1, rs2	rd = rs1 ≫ rs2
Add Word	R	0111011	000	0000000	addw rd, rs1, rs2	rd = rs1 + rs2
Subtract Word	R	0111011	000	0100000	subw rd, rs1, rs2	rd = rs1 − rs2
Shift Left Logical Word	R	0111011	001	0000000	sllw rd, rs1, rs2	rd = rs1 ≪ rs2
Shift Right Logical Word	R	0111011	101	0000000	srlw rd, rs1, rs2	rd = rs1 ≫ rs2
Shift Right Arithmetic Word†	R	0111011	101	0100000	sraw rd, rs1, rs2	rd = rs1 ≫ rs2
Add Immediate	I	0010011	000		addi rd, rs1, imm	rd = rs1 + imm
AND Immediate	I	0010011	111		andi rd, rs1, imm	rd = rs1 & imm
OR Immediate	I	0010011	110		ori rd, rs1, imm	rd = rs1 | imm
XOR Immediate	I	0010011	100		xori rd, rs1, imm	rd = rs1 ^ imm
Shift Left Logical Immediate	I	0010011	001	0000000	slli rd, rs1, shamt	rd = rs1 ≪ shamt
Shift Right Logical Immediate	I	0010011	101	0000000	srli rd, rs1, shamt	rd = rs1 ≫ shamt
Shift Right Arithmetic Immediate†	I	0010011	101	0100000	srai rd, rs1, shamt	rd = rs1 ≫ shamt
Set Less Than Immediate	I	0010011	010		slti rd, rs1, imm	rd = (rs1 < imm)?1:0
Set Less Than Immediate (U)*	I	0010011	011		sltiu rd, rs1, imm	rd = (rs1 < imm)?1:0
Add Immediate Word	I	0011011	000		addiw rd, rs1, imm	rd = rs1 + imm
Shift Left Logical Immediate Word	I	0011011	001	0000000	slliw rd, rs1, shamt	rd = rs1 ≪ shamt
Shift Right Logical Immediate Word	I	0011011	101	0000000	srliw rd, rs1, shamt	rd = rs1 ≫ shamt
Shift Right Arithmetic Imm Word†	I	0011011	101	0100000	sraiw rd, rs1, shamt	rd = rs1 ≫ shamt
Load Byte	I	0000011	000		lb rd, rs1, imm	rd = M[rs1+imm][0:7]
Load Half	I	0000011	001		lh rd, rs1, imm	rd = M[rs1+imm][0:15]
Load Word	I	0000011	010		lw rd, rs1, imm	rd = M[rs1+imm][0:31]
Load Doubleword	I	0000011	011		ld rd, rs1, imm	rd = M[rs1+imm][0:63]
Load Byte (U)*	I	0000011	100		lbu rd, rs1, imm	rd = M[rs1+imm][0:7]
Load Half (U)*	I	0000011	101		lhu rd, rs1, imm	rd = M[rs1+imm][0:15]
Load Word (U)*	I	0000011	110		lwu rd, rs1, imm	rd = M[rs1+imm][0:31]
Store Byte	S	0100011	000		sb rs1, rs2, imm	M[rs1+imm][0:7] = rs2[0:7]
Store Half	S	0100011	001		sh rs1, rs2, imm	M[rs1+imm][0:15] = rs2[0:15]
Store Word	S	0100011	010		sw rs1, rs2, imm	M[rs1+imm][0:31] = rs2[0:31]
Store Doubleword	S	0100011	011		sd rs1, rs2, imm	M[rs1+imm][0:63] = rs2[0:63]
Branch If Equal	B	1100011	000		beq rs1, rs2, imm	if(rs1 == rs2) PC += imm
Branch Not Equal	B	1100011	001		bne rs1, rs2, imm	if(rs1 != rs2) PC += imm
Branch Less Than	B	1100011	100		blt rs1, rs2, imm	if(rs1 < rs2) PC += imm
Branch Greater Than Or Equal	B	1100011	101		bge rs1, rs2, imm	if(rs1 ≥ rs2) PC += imm
Branch Less Than (U)*	B	1100011	110		bltu rs1, rs2, imm	if(rs1 < rs2) PC += imm
Branch Greater Than Or Equal (U)*	B	1100011	111		bgeu rs1, rs2, imm	if(rs1 ≥ rs2) PC += imm
Load Upper Immediate	U	0110111			lui rd, imm	rd = imm ≪ 12
Add Upper Immediate To PC	U	0010111			auipc rd, imm	rd = PC + (imm ≪ 12)
Jump And Link	J	1101111			jal rd, imm	rd = PC + 4; PC += imm
Jump And Link Register	I	1100111	000		jalr rd, rs1, imm	rd = PC + 4; PC = rs1 + imm

*Assumes values are unsigned integers and zero extends † Fills in with sign bit during right shift and msb (most significant bit) extends

RV64M Standard Extension Instructions

Name	Fmt	Opcode	Funct3	Funct7	Assembly	Description (in C)
Multiply	R	0110011	000	0000001	mul rd, rs1, rs2	rd = (rs1 · rs2)[63:0]
Multiply Upper Half	R	0110011	001	0000001	mulh rd, rs1, rs2	rd = (rs1 · rs2)[127:64]
Multiply Upper Half Sign/Unsigned†	R	0110011	010	0000001	mulhsu rd, rs1, rs2	rd = (rs1 · rs2)[127:64]
Multiply Upper Half (U)*	R	0110011	011	0000001	mulhu rd, rs1, rs2	rd = (rs1 · rs2)[127:64]
Divide	R	0110011	100	0000001	div rd, rs1, rs2	rd = rs1 / rs2
Divide (U)*	R	0110011	101	0000001	divu rd, rs1, rs2	rd = rs1 / rs2
Remainder	R	0110011	110	0000001	rem rd, rs1, rs2	rd = rs1 % rs2
Remainder (U)*	R	0110011	111	0000001	remu rd, rs1, rs2	rd = rs1 % rs2
Multiply Word	R	0111011	000	0000001	mulw rd, rs1, rs2	rd = (rs1 · rs2)[63:0]
Divide Word	R	0111011	100	0000001	divw rd, rs1, rs2	rd = rs1 / rs2
Divide Word (U)*	R	0111011	101	0000001	divuw rd, rs1, rs2	rd = rs1 / rs2
Remainder Word	R	0111011	110	0000001	remw rd, rs1, rs2	rd = rs1 % rs2
Remainder Word (U)*	R	0111011	111	0000001	remuw rd, rs1, rs2	rd = rs1 % rs2

*Assumes values are unsigned integers and zero extends † Multiply with one operand signed and the other unsigned

Bibliography

[1] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978. ISSN 1557-7317.

[2] C.-J. Johnson. Implementing a mips processor using sme. Master's thesis, University of Copenhagen, Niels Bohr Institute, 2017.

[3] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design RISC-V Edition: The Hardware Software/Interface*. Morgan Kaufmann, 2017. ISBN 9780128122754.

[4] M. Rehr, K. Skovhede, and B. Vinter. Bpu simulator. *Communicating Process Architectures*, pages 233–248, 2013.

[5] E. Skaarup and A. Frisch. Generation of fpga hardware specifications from pycsp networks. Master's thesis, University of Copenhagen, Niels Bohr Institute, 2014.

[6] K. Skovhede and B. Vinter. Building hardware from c# models. In *FSP 2016; Third International Workshop on FPGAs for Software Programmers*, pages 1–9. VDE, 2016.

[7] B. Vinter and K. Skovhede. Synchronous message exchange for hardware designs. *Communicating Process Architectures*, pages 201–212, 2014.