Notes

Implementation of RISC-V in SME

Daniel Ramyar

January 27, 2020

Contents

Chapter 1

Introduction

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 [2].

Chapter sections are subject to change in name and order

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 is 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 would 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 undefined 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.

rewrite this section.

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 6 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.

remember to tell why this is later

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.

remember to tell why this is later

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 permutations 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 using our basic operators AND, OR and NOT.

give a definition of disjunctive normal form

To do this we first identify all true output 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

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

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 table 2.4 with the corresponding operators which we have defined throughout this section. The filled table can be found in table 2.8.

rewrite this section.

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 the last section 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 use this notation since it is 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. 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 low 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 though always be 1 output which is the result of all the subsequent input.

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.

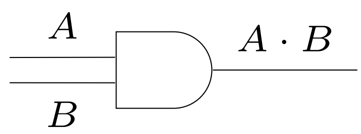

Figure 2.1: Illustration of the AND gate where A and B are the input and $A \cdot B$ is the output.

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.

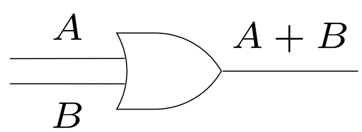

Figure 2.2: Illustration of the OR gate where A and B are the input and $A + B$ is the output.

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.

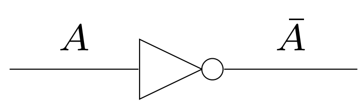

Figure 2.3: Illustration of the NOT gate where A and B are the input and $A + B$ is the output.

(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.

maybe talk about that nor and nand gates are universal

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.

finish this section

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 can 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.

We had 3 input in the previous example, 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 to output 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 have 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 a previous section 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 next section we will se how this is used to generate the gate-level implementation.

2.2.4 Programmable logic array

Chapter 3

Synchronous Message Exchange

3.1 Communicating Sequential Processes

Rewrite this section

add examples of a process and channels

The problem with multiprocessor workloads is the sharing of memory. This creates a whole slew of problems. There are many different processes going on at once all having access to the same memory. Unless you got superpowers it is very hard to determine where in the program something goes wrong. It all boils down to the non-determinism.

For example if you are going to print multiple strings using multiple threads you don't know which string i going to be printed first it's gonna depend on the operating system not on anything in your code. That can create race conditions (meaning the behaviour in your code is dependent on the timing of different threads) which can cause unpredictable behaviour and therefore bugs which is undesirable.

This has been tried to been solved with mutexes or locks but this also have its downside inform of deadlocks where multiple processes are waiting for each other and because these processes are non-deterministic it is very hard to reproduce errors in your code which in turn makes it hard to debug and therefore hard to make reliable software.

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

The other primitive is channels which is the way the processes communicate which each other. You can pass whatever you want through these channels and once you pass a value you loose access to it.

There is a lot of ways the processes and channels can be arranged the most simple one can be found in figure 3.1 which illustrates process 1 which passes a value onto a channel which process 2 takes as input. Some different configuations can be found in figures 3.2-3.4

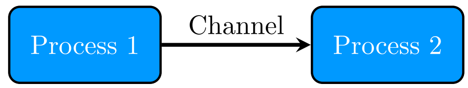

Figure 3.1: CSP one to one

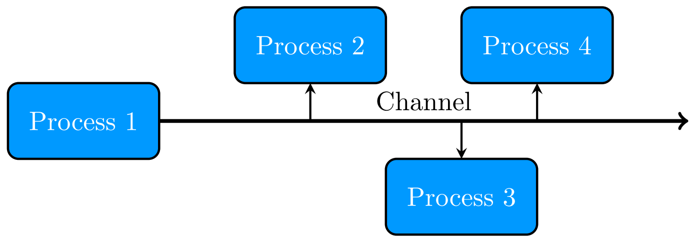

Figure 3.2: CSP one to many

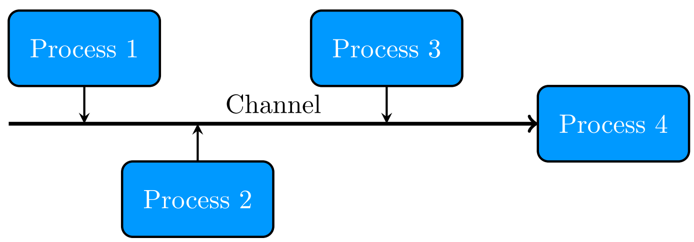

Figure 3.3: CSP many to one

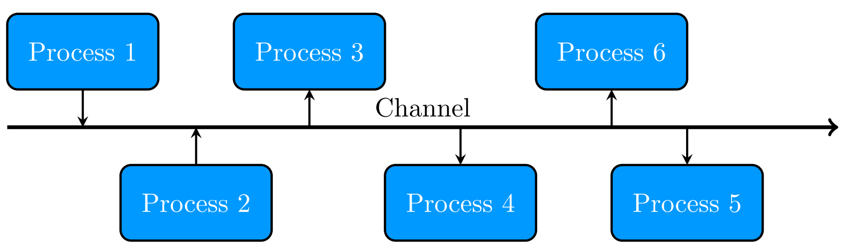

Figure 3.4: CSP many to many

3.2 Synchronous Message Exchange

Vinter and Skovhede [3] Vinter and Skovhede [4]

Chapter 4

Introduction to RISC-V instructions

Chapter sections are subject to change in name and order

This chapter aims to introduce the reader to the basics of machine language. Based on chapter 2 in [2]

4.1 RISC-V Assembly

4.2 Operands

4.2.1 Register

4.2.2 Memory Format

4.2.3 Const vs imm

4.3 Numeral system of a computer

4.3.1 base 2

4.3.2 signed unsigned

4.4 Instruction representation in binary

4.5 Operators

Chapter 5

The RISC-V processor

This chapter aims to introduce the reader to the basics of machine language. Based on chapter 4 in [2]

Chapter sections are subject to change in name and order

5.1 Single Cycle RISC-V Units

make a better title

5.1.1 Program Counter

5.1.2 Instruction Memory

5.1.3 incrementor?

figure out a name for this subsection

5.1.4 Register

5.1.5 Arithmetic Logic Unit (ALU)

5.1.6 Immediate generator

5.1.7 Data Memory

Need to figure out more sections to explain whole datapath

5.2 Designing the Control

5.3 Single Cycle RISC-V datapath

5.4 Improving the datapath

figure out better naming for sections

5.4.1 RV64I Base Instructions Support

5.4.2 Supporting R-Format

5.4.3 Supporting I-Format

5.4.4 Supporting S-Format

5.4.5 Supporting B-Format

5.4.6 Supporting U-Format

5.4.7 Supporting J-Format

5.5 Debugging the instructions

5.5.1 Writing assembly to test instructions

5.5.2 Writing simple C code to run on RISC-V

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

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

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

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

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 $\nrightarrow$ 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	$p \nrightarrow q$
$true$	$true$	$false$
$true$	$false$	$true$
$false$	$true$	$false$
$false$	$false$	$false$

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

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

using the procedure mentioned in chapter 2.1.

Converse nonimplication

Converse nonimplication is represented by $\nleftarrow$ 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	$p \nleftarrow q$
$true$	$true$	$false$
$true$	$false$	$false$
$false$	$true$	$true$
$false$	$false$	$false$

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

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

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 $\ll$ 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 $\gg$ rs2
Shift Right Arithmetic[†]	R	0110011	101	0100000	sra rd, rs1, rs2	rd = rs1 $\gg$ 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 $\ll$ rs2
Shift Right Logical Word	R	0111011	101	0000000	srlw rd, rs1, rs2	rd = rs1 $\gg$ rs2
Shift Right Arithmetic Word[†]	R	0111011	101	0100000	sraw rd, rs1, rs2	rd = rs1 $\gg$ rs2
Add Immediate	I	0010011	000		addi rd, rs1, imm	rd = rs1 + imm
AND Immediate	I	0010011	111		and rd, rs1, imm	rd = rs1 & imm
OR Immediate	I	0010011	110		or rd, rs1, imm	rd = rs1 | imm
XOR Immediate	I	0010011	100		xor rd, rs1, imm	rd = rs1 ^ imm
Shift Left Logical Immediate	I	0010011	001	0000000	slli rd, rs1, shamt	rd = rs1 $\ll$ shamt
Shift Right Logical Immediate	I	0010011	101	0000000	srli rd, rs1, shamt	rd = rs1 $\gg$ shamt
Shift Right Arithmetic Immediate[†]	I	0010011	101	0100000	srai rd, rs1, shamt	rd = rs1 $\gg$ 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 $\ll$ shamt
Shift Right Logical Immediate Word	I	0011011	101	0000000	srliw rd, rs1, shamt	rd = rs1 $\gg$ shamt
Shift Right Arithmetic Imm Word[†]	I	0011011	101	0100000	sraiw rd, rs1, shamt	rd = rs1 $\gg$ 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 $\geq$ 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 $\geq$ rs2) PC += imm
Load Upper Immediate	U	0110111			lui rd, imm	rd = imm $\ll$ 12
Add Upper Immediate To PC	U	0010111			auipc rd, imm	rd = PC + (imm $\ll$ 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] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design RISC-V Edition: The Hardware Software/Interface*. Morgan Kaufmann, 2017. ISBN 9780128122754.

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

[4] B. Vinter and K. Skovhede. Bus centric synchronous message exchange for hardware designs. 2015.