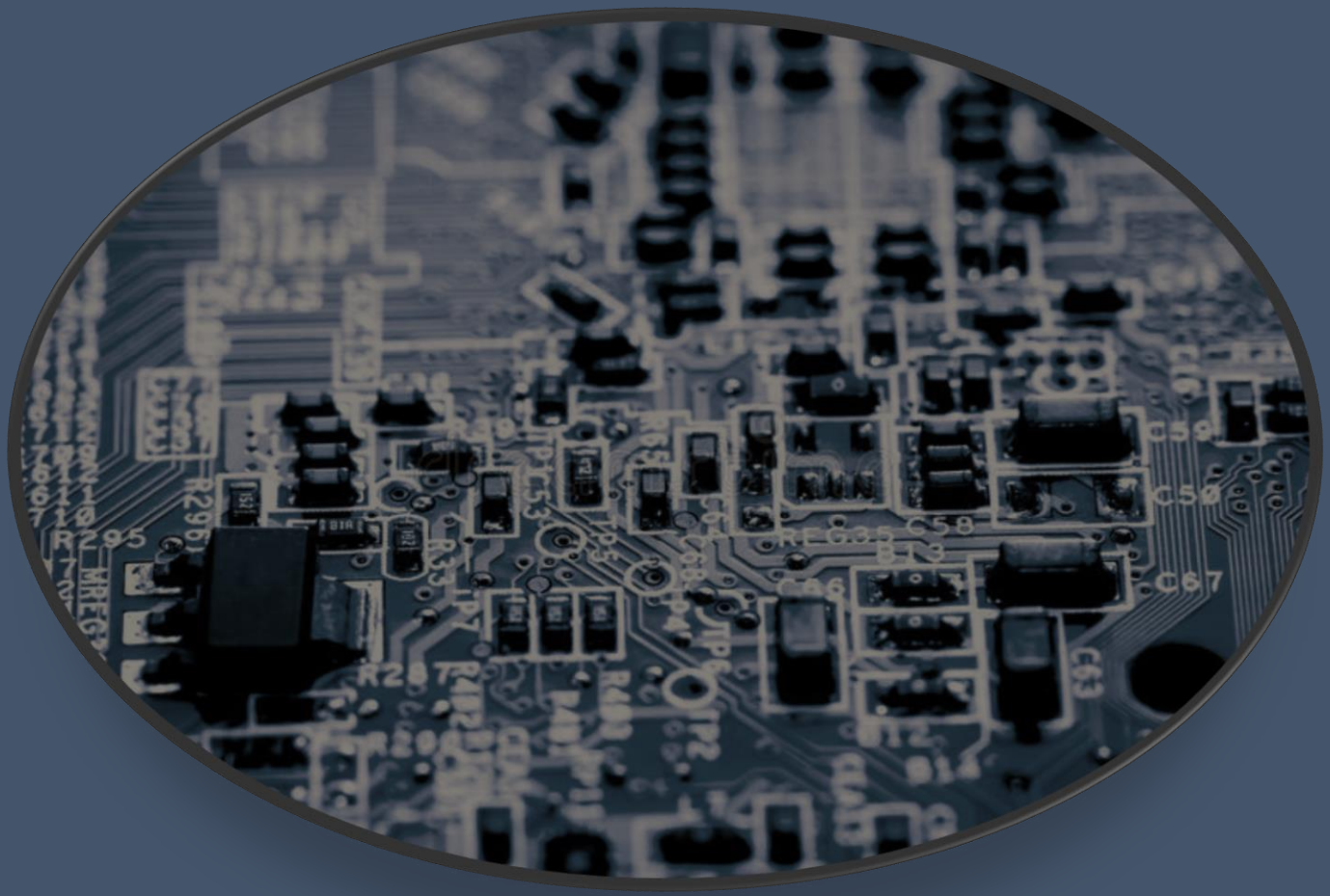
ITEC10261

Computer Technology & Maths

Computer Technology



Matthew Robinson

CONTENTS

Re-cap

Arithmetic

Logic

Computer Technology

Introduction

Routing Data

Storing Data

Routing and Storage

Architectures

Assembly Language

Transforming Data

CONTENTS

Re-cap

Arithmetic

Introduction to Number Systems

Definitions

A **number system** is used for representing numbers of a certain type.

Conversion Formula

$$value = \sum_{n=0}^{\infty} (weight_n \times digit_n)$$

Arithmetic

Decimal Number System

Definition

The **decimal number system** is a number system that:

- uses a notation in which each number is expressed in base 10;
- uses the number range 0-9; and
- lets each place value be a power of 10.

Place values

Information: Decimal place values

Position	5	4	3	2	1
Weight *(base 10)*	10000	1000	100	10	1

Example

Example: Determining decimal numbers

Determine the decimal number represented below.

Position	5	4	3	2	1
Weight *(base 10)*	10000	1000	100	10	1
Decimal Number	1	2	6	8	7

$$value = \sum_{n=0}^{\infty} (weight_n \times digit_n)$$

$$value = (10000 \times 1) + (1000 \times 2) + (100 \times 6) + (10 \times 8) + (7 \times 1)$$
$$value = 10000 + 2000 + 600 + 80 + 7$$
$$\boldsymbol{value = 12687}$$

Arithmetic

Binary Number System

Definition

The **binary number system** is a number system that:

- uses a notation in which each number is expressed in base 2;
- uses the number range 0-1; and
- lets each place value be a power of 2.

One binary character is represented by one (1) bit.

Place values

Information: Binary place values

Position	8	7	6	5	4	3	2	1
Weight *(base 2)*	128	64	32	16	8	4	2	1

The minimum decimal value for a binary number constructed using n bits is 0.
The maximum decimal value for a binary number constructed using n bits is calculated by $2^n - 1$.

An 8-bit binary number has:

- a minimum decimal value of 0, when the binary value is 00000000;
- a minimum non-zero decimal value of 1, when the binary value is 00000001; and
- a maximum decimal value of 255, when the binary value is 11111111.

Example

Example: Determining binary numbers

Determine the decimal number represented below.

Position	8	7	6	5	4	3	2	1
Weight *(base 2)*	128	64	32	16	8	4	2	1
Binary Number	0	1	1	0	1	1	0	1

$$value = \sum_{n=0}^{\infty} (weight_n \times digit_n)$$

$$value = (128 \times 0) + (64 \times 1) + (32 \times 1) + (16 \times 0) + (8 \times 1) + (4 \times 1) + (2 \times 0) + (1 \times 1)$$

$$value = 0 + 64 + 32 + 0 + 8 + 4 + 0 + 1$$

$$value = 64 + 32 + 8 + 4 + 1$$

$$\boldsymbol{value = 109}$$

Arithmetic

Hexadecimal Number System

Definition

The **hexadecimal number system**, or **hex**, is a number system that:

- uses a notation in which each number is expressed in base 16;
- uses the number range 0-9, A-F; and
- lets each place value be a power of 16.

One hexadecimal character is represented by a nibble (4 bits).

Place values

Information: Hexadecimal place values

Position	4	3	2	1
Weight *(base 16)*	4096	256	16	1

The minimum decimal value for a hexadecimal number constructed using n bits is 0.
The maximum decimal value for a hexadecimal number constructed using n bits is calculated by $16^n - 1$.

Example

Example: Determining hexadecimal numbers

Determine the hexadecimal number represented below.

Position	2	1
Weight *(base 16)*	16	1
Binary Number	4	1

$$value = \sum_{n=0}^{\infty} (weight_n \times digit_n)$$
$$value = (16 \times 4) + (1 \times 1)$$
$$value = 64 + 1$$
$$\boldsymbol{value = 65}$$

Arithmetic

Symbols

Information: Hexadecimal symbols

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

Converting decimal to hexadecimal

Division method

Process: Division method

1) Set the remainder equal to the decimal number.
2) Starting at the leftmost power and repeat until division leaves no remainder:
   - divide the remainder by the power;
   - if the result is 0, place the result under that place value and do not change the remainder; and
   - if the result is not 0, place the result under that place value and set the remainder r to the value left over after the division.
3) Write the answer.

Example: Division method

Convert 56_{10} to hexadecimal.

$r = 56$

4096	256	16	1

$\frac{56_{10}}{4096_{10}} = 0_{10}\ (r = 56)$

4096	256	16	1
0			

Set the remainder equal to the decimal number.

Starting at the leftmost power and repeat until division leaves no remainder:

- divide the remainder by the power;
- if the result is 0, place the result under that place value and do not change the remainder; and
- if the result is not 0, place the result under that place value and set the remainder r to the value left over after the division.

Arithmetic

$\frac{56_{10}}{256_{10}} = 0_{10}\ (r = 56)$

4096	256	16	1
0	0		

$\frac{56_{10}}{16_{10}} = 3_{10}\ (r = 8)$

4096	256	16	1
0	0	3	

$\frac{8_{10}}{1_{10}} = 8_{10}\ (r = 0)$

4096	256	16	1
0	0	3	8

$\mathbf{0038_{16}}$

Starting at the leftmost power and repeat until division leaves no remainder:

- divide the remainder by the power;
- if the result is 0, place the result under that place value and do not change the remainder; and
- if the result is not 0, place the result under that place value and set the remainder r to the value left over after the division.

Write the answer.

Binary method

Process: Binary method

1) Convert the decimal number to binary.
2) Split the binary number into octets (4 bits) and give each octet its own place values – if necessary zero's (0's) can be added before the most significant bit (MSB) if the number of characters in the binary number is not divisible by four.
3) Either:
   - compare each binary octet to the hexadecimal symbols table to find its hexadecimal equivalent; or
   - convert each binary octet to decimal and compare the decimal number to the hexadecimal symbols table to find its hexadecimal equivalent.
4) Write the answer.

Example: Binary method

Convert 56_{10} to hexadecimal.

128	64	32	16	8	4	2	1
0	0	1	1	1	0	0	0

8	4	2	1
0	0	1	1

8	4	2	1
1	0	0	0

Convert the decimal number to binary.

Split the binary number into octets (4 bits) and give each octet its own place values – if necessary zero's (0's) can be added before the most significant bit (MSB) if the number of characters in the binary number is not divisible by four.

Arithmetic

Decimal	Hexadecimal	Binary
3	3	11

$\therefore 0011_2 = 3_{16}$

Decimal	Hexadecimal	Binary
8	8	1000

$\therefore 1000_2 = 8_{16}$

Compare each binary octet to the hexadecimal symbols table to find its hexadecimal equivalent.

Either:

8	4	2	1
0	0	1	1

$0011_2 = 3_{10}$

Decimal	Hexadecimal	Binary
3	3	11

$\therefore 3_{10} = 3_{16}$

8	4	2	1
1	0	0	0

$1000_2 = 8_{10}$

Decimal	Hexadecimal	Binary
8	8	1000

$\therefore 8_{10} = 8_{16}$

Convert each binary octet to decimal and compare the decimal number to the hexadecimal symbols table to find its hexadecimal equivalent.

$\mathbf{0038_{16}}$

Write the answer.

Converting hexadecimal to binary

Process: Converting hexadecimal to binary

1) Split the hexadecimal number into their place values. If necessary, consult the Hexadecimal symbols table to convert symbols A-F.
2) Setup octets (4 bits) binary number place values.
3) Calculate the binary equivalent of each hexadecimal place value in the respective binary octet (4 bits).
4) Group the octets (4 bits) together.

Example: Converting hexadecimal to binary

Convert $3B_{16}$ to decimal.

16	1
3	B

$B_{16} = 11_{10}$

Split the hexadecimal number into their place values. If necessary, consult the Hexadecimal symbols table to convert symbols A-F.

16	1
3	11

8	4	2	1

8	4	2	1

Setup octets (4 bits) binary number place values.

8	4	2	1
0	0	1	1

8	4	2	1
1	0	1	1

Calculate the binary equivalent of each hexadecimal place value in the respective binary octet (4 bits).

$\mathbf{00111011_2}$

Group the octets (4 bits) together.

Arithmetic

Hexadecimal addition

Process: Hexadecimal addition

1) Convert each hexadecimal number to binary.

Process: Converting hexadecimal to binary

1) Split the hexadecimal number into their place values. If necessary, consult the Hexadecimal symbols table to convert symbols A-F.
2) Setup octets (4 bits) binary number place values.
3) Calculate the binary equivalent of each hexadecimal place value in the respective binary octet (4 bits).
4) Group the octets (4 bits) together.

2) Perform binary addition using the two binary numbers.

Process: Binary Addition

1) Add the first binary number and the second binary number together using the binary addition rules:
   - $0_2 + 0_2 = 00_2 = 0_{10}$
   - $0_2 + 1_2 = 01_2 = 1_{10}$
   - $1_2 + 0_2 = 01_2 = 1_{10}$
   - $1_2 + 1_2 = 10_2 = 2_{10}$
   - $1_2 + 1_2 + 1_2 = 11_2 = 3_{10}$
2) If the expression evaluates to an answer larger than one bit:
   - the first of the two bits is a carry bit;
   - the carry bit is moved to the next column; and
   - the addition in the next column must incorporate the carry bit.
3) If there is a carry bit present after performing addition on the most significant bit (MSB), there is an overflow error because there are not enough bits to represent all of the binary digits. This can be ignored when the binary numbers are represented using Two's complement.

3) Convert the result to decimal.

Example: Hexadecimal addition

Calculate the answer to the hexadecimal sum $0 \times 2A + 0 \times E8$. Assume that all numbers are signed, using Two's complement, and express the answer in decimal notation.

$2A$

16	1
2	A

$A_{16} = 10_{10}$

16	1
2	10

8	4	2	1
0	0	1	0

8	4	2	1
1	0	1	0

$2A_{16} = 00101010_2$

$E8$

16	1
E	8

$E_{16} = 14_{10}$

16	1
14	8

8	4	2	1
1	1	1	0

8	4	2	1
1	0	0	0

$E8_{16} = 11101000_2$

Convert each hexadecimal number to binary.

	128	64	32	16	8	4	2	1
$2A_{16}$	0	0	1	0	1	0	1	0
$E8_{16}$	1	1	1	0	1	0	0	0
SUM		0	0	1	0	0	1	0
CARRY	1	1		1				

Perform binary addition using the two binary numbers.

$0 \times 3a + 0 \times E8 = 18_2$

Convert the result to decimal.

Logic

Boolean Logic and Logic Gates

Boolean logic

Definition

Boolean algebra is a set of rules to describe certain propositions whose outcome could be either $True$ or $False$.

Postulates

Information: Boolean Postulates

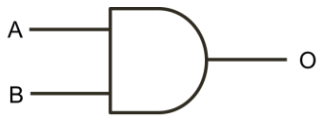
$$\begin{array}{cccccccccc} & & X & = & 0 & OR & X & = & 1 & \\ 0 & AND & 0 & = & 0 & & & & & \\ 1 & OR & 1 & = & 1 & & & & & \\ 0 & OR & 0 & = & 0 & & & & & \\ 1 & AND & 1 & = & 1 & & & & & \\ 1 & AND & 0 & = & 0 & & & & & \\ 1 & OR & 0 & = & 1 & , & 0 & OR & 1 & = & 0 \end{array}$$

Logic gates

Logic gates perform basic logical functions and are the fundamental building blocks of digital integrated circuits.

Fact File: AND gate

Notation	Circuit	Truth Table			Code Representation
		A	B	O	Python

$A \,.\, B$		0	0	0	if A and B:
					  print("true")
					else:
					  print("false")

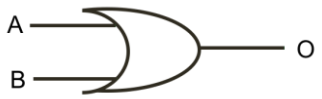
		0	1	0	
		1	0	0	
		1	1	1	
Explanation					**Java / C++**

An output is True if:
all inputs are True.
					if(A && B) {
					  // true
					} else {
					  // false
					}

Logic

Fact File: OR gate

Notation	Circuit	Truth Table			Code Representation
		A	B	O	Python

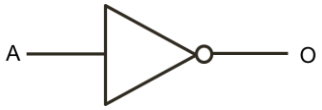
$A + B$		0	0	0	if A or B:
					  print("true")
					else:
					  print("false")

		0	1	1	
		1	0	1	
		1	1	0	
Explanation					**Java / C++**

An output is True if:
at least one input is True.
					if(A || B) {
					  // true
					} else {
					  // false
					}

Fact File: NOT gate

Notation	Circuit	Truth Table		Code Representation
		A	O	Python

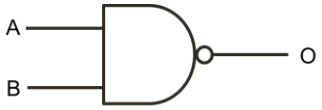
$\bar{A}$		0	1	if not A:
				  ...
				else:
				  ...

		1	0	
Explanation				**Java / C++**

An output is True if:
the input is False.
				if(!A) {
				  ...
				} else {
				  ...
				}

Fact File: NAND gate (NOT AND)

Notation	Circuit	Truth Table			Code Representation
		A	B	O	Python

$\overline{(A \,.\, B)}$		0	0	1	if not (A and B):
					  print("true")
					else:
					  print("false")

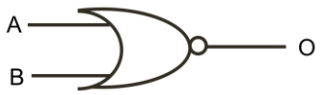
		0	1	1	
		1	0	1	
		1	1	0	
Explanation					**Java / C++**

An output is True if:
not all inputs are True.
					if(! (A && B)) {
					  // true
					} else {
					  // false
					}

Logic

Fact File: NOR gate (NOT OR)

Notation	Circuit	Truth Table			Code Representation
		A	B	O	Python

$\overline{(A + B)}$		0	0	1	if not (A or B):
					  print("true")
					else:
					  print("false")

		0	1	0	
		1	0	0	
		1	1	0	
Explanation					**Java / C++**

An output is True if:
no inputs are True.
					if(! (A || B)) {
					  // true
					} else {
					  // false
					}

Fact File: XOR gate (Exclusive OR)

Notation	Circuit	Truth Table			Code Representation
		A	B	O	Python

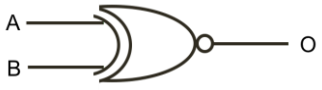
$A \oplus B$		0	0	0	if A != B:
					  print("true")
					else:
					  print("false")

		0	1	1	
		1	0	1	
		1	1	0	
Explanation					**Java / C++**

An output is True if:
only one input is True.
					if(A != B) {
					  // true
					} else {
					  // false
					}

Fact File: XNOR gate (Exclusive NOT OR)

Notation	Circuit	Truth Table			Code Representation
		A	B	O	Python

$\overline{A \oplus B}$		0	0	1	if not (A and B):
					  print("true")
					else:
					  print("false")

		0	1	1	
		1	0	1	
		1	1	0	
Explanation					**Java / C++**

An output is True if:
all inputs are True or all inputs are False.
					if(! (A && B)) {
					  // true
					} else {
					  // false
					}

When building circuits or evaluating Boolean expressions, the following order of precedence should be observed:

1) NOT;
2) AND; then
3) OR.

Logic

Truth tables

Definition

A **truth table** is a mathematical table used to compute the functional values of logical expressions on each of their functional arguments. Truth tables are composed of one column for each input variable and one final column for all of the possible results of the logical operation.

Deriving Boolean logic

Using truth tables to derive Boolean expressions and logic circuits

Process: Deriving Boolean expression	
Deriving Boolean Expressions	**Re-writing Using Boolean Notation**
1) Inspect each input where the output is True (1). 2) Construct a Boolean expression for the inputs on that row. 3) Repeat stages 1) and 2) for each row and separate the constructed Boolean expressions with ORs.	1) If the variable evaluates to 0, re-write the variable using the NOT notation. 2) If the variable evaluates to 1, do not modify the variable. 3) Replace other logic with the appropriate notation.

Example: Deriving Boolean expression

Truth Table				Derived Boolean Expression	Boolean Notation Expression
A	**B**	**C**	**O**		
0	0	0	0	$O =$ $A = 0\ AND\ B = 0\ AND\ C = 1$ OR $A = 0\ AND\ B = 1\ AND\ C = 0$ OR $A = 1\ AND\ B = 0\ AND\ C = 0$ OR $A = 1\ AND\ B = 1\ AND\ C = 1$	$O =$ $\bar{A} . \bar{B} . C + \bar{A} . B . \bar{C} + A . \bar{B} . \bar{C} + A . B . C$
0	0	1	1		
0	1	0	1		
0	1	1	0		
1	0	0	1		
1	0	1	0		
1	1	0	0		
1	1	1	1		

Logic

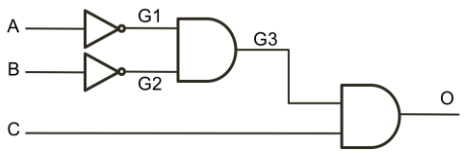
Using Logic circuit diagrams to derive truth tables and Boolean expressions

Process: Deriving truth table and Boolean expression

Deriving Truth Table	Deriving Boolean Expression	Re-writing Using Boolean Notation
1) Mark points on the diagram at output end of each logic gate. (e.g. G1, G2, G3 etc.) 2) Construct a truth table with columns for the inputs and the points marked on the diagram. 3) Populate the input columns with the possible permutations of input values (0 and 1). 4) Calculate the result of the input to each of the logic circuits and write the result in the columns for the points on the diagram – do this for each row. 5) Calculate the output based on the values calculated.	1) Inspect each input where the output is True (1). 2) Construct a Boolean expression for the inputs on that row. 3) Repeat stages 1) and 2) for each row and separate the constructed Boolean expressions with ORs.	1) If the variable evaluates to 0, re-write the variable using the NOT notation. 2) If the variable evaluates to 1, do not modify the variable. 3) Replace other logic with the appropriate notation.

Example: Deriving Truth Table and Boolean Expression

Logic Circuit Diagram



Truth Table

A	B	C	G1	G2	G3	O
0	0	0	1	1	1	0
0	0	1	1	1	1	1
0	1	0	1	0	0	0
0	1	1	1	0	0	0
1	0	0	0	1	0	0
1	0	1	0	1	0	0
1	1	0	0	0	0	0
1	1	1	0	0	0	0

Derived Boolean Expression

$$O = A = 0 \; AND \; B = 0 \; AND \; C = 1$$

Boolean Notation Expression

$$O = \bar{A} \,.\, \bar{B} \,.\, C$$

Calculating values

Example: Calculating values in Boolean expressions

Calculate the value of F in the Boolean expression $F = \overline{(X \,.\, Y + X \,.\, Z)}$, assuming $X = 1$, $Y = 0$, and $Z = 0$.

$F = \overline{(1 \,.\, 0 + 1 \,.\, 0)}$	Substitute in values
$F = \overline{(0 + 0)}$	$1 \,.\, 0 = 0$
$F = \overline{(0)}$	$0 + 0 = 0$
$\boldsymbol{F = 1}$	$\overline{(0)} = 1$

Logic

Equivalent Circuits

Definition

Equivalent circuits are ones where several statements have logical equivalence. This is a type of relationship where several circuits may be logically equivalent, in that they all have identical truth tables.

Example

Example: Two equivalent circuits

$X = A + B\,.\,C$					$Y = (A + B)\,.\,(A + C)$					
A	**B**	**C**	**G1** (B . C)	**X**	**A**	**B**	**C**	**G1** (A + B)	**G2** (A + C)	**Y**
0	0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	1	0	1	0
0	1	0	0	0	0	1	0	1	0	0
0	1	1	1	1	0	1	1	1	1	1
1	0	0	0	1	1	0	0	1	1	1
1	0	1	0	1	1	0	1	1	1	1
1	1	0	0	1	1	1	0	1	1	1
1	1	1	1	1	1	1	1	1	1	1

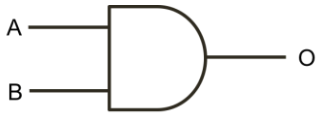
By abstracting the circuits to the inputs (A, B and C) and the output (X), it is shown that the circuits are equivalent.

NAND Gates

NAND gates are “universal gates”, meaning that they can be used to construct all other logic gates – this method is used to construct logic gates in most modern CPUs. This method of constructing logic gates is used because it is cheaper to construct CPUs using only one gate rather than many.

Fact File: NAND construction of AND gate

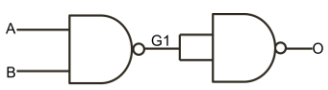
AND – Circuit



AND – Truth Table

A	B	O
0	0	0
0	1	0
1	0	0
1	1	1

NAND – Circuit



NAND – Truth Table

A	B	G1	O
0	0	1	0
0	1	1	0
1	0	1	0
1	1	0	1

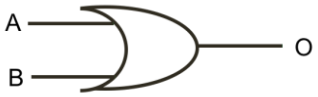
Boolean Expression

$$A\,.\,B = \overline{\overline{A}\,.\,\overline{B}}$$

Logic

Fact File: NAND construction of OR gate

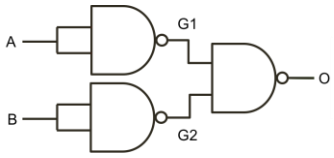
OR – Circuit



OR – Truth Table

A	B	O
0	0	0
0	1	1
1	0	1
1	1	1

NAND – Circuit



NAND – Truth Table

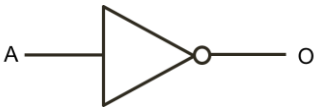
A	B	G1	G2	O
0	0	1	1	0
0	1	1	0	1
1	0	0	1	1
1	1	0	0	1

Boolean Expression

$$A + B = \overline{\overline{(A + B)}} = \overline{(\bar{A} \,.\, \bar{B})}$$

Fact File: NAND construction of NOT gate

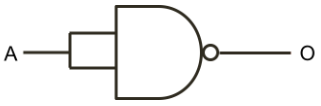
NOT – Circuit



NOT – Truth Table

A	O
0	1
1	0

NAND – Circuit



NAND – Truth Table

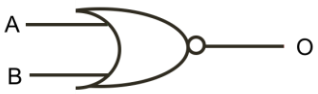
A	A	O
0	0	1
0	1	1
1	0	1
1	1	0

Boolean Expression

$$\bar{A} = \overline{(A \,.\, A)}$$

Fact File: NAND construction of NOR gate (NOT OR)

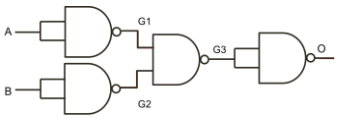
NOR – Circuit



NOR – Truth Table

A	B	O
0	0	1
0	1	0
1	0	0
1	1	0

NAND – Circuit



NAND – Truth Table

A	B	G1	G2	G3	O
0	0	1	1	0	1
0	1	1	0	1	0
1	0	0	1	1	0
1	1	0	0	1	0

Boolean Expression

$$\overline{(A + B)} = \overline{\overline{(A + B)}} = \overline{(\bar{A} \,.\, \bar{B})}$$

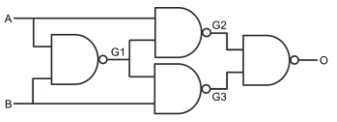
Fact File: NAND construction of XOR gate (Exclusive OR)

XOR – Circuit

XOR – Truth Table

A	B	O
0	0	0
0	1	1
1	0	1
1	1	0

NAND – Circuit



NAND – Truth Table

A	B	G1	G2	G3	O
0	0	1	1	1	0
0	1	1	1	0	1
1	0	1	0	1	1
1	1	0	1	1	0

Boolean Expression

$$A \oplus \mathrm{B} = \overline{((A \,.\, \overline{(A \,.\, B)}) \,.\, \overline{(B \,.\, \overline{(A \,.\, B)})})} = A \,.\, \bar{B} + \bar{A} \,.\, B$$

Logic

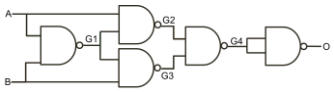
Fact File: NAND construction of XNOR gate (Exclusive NOT OR)

XNOR – Circuit

XNOR – Truth Table

A	B	O
0	0	1
0	1	0
1	0	0
1	1	1

NAND – Circuit



NAND – Truth Table

A	B	G1	G2	G3	G4	O
0	0	1	1	1	0	1
0	1	1	1	0	1	0
1	0	1	0	1	1	0
1	1	0	1	1	0	1

Boolean Expression

$$\overline{A \oplus B} = \overline{((A \,.\, \overline{(A \,.\, B)}) \,.\, \overline{(B \,.\, \overline{(A \,.\, B)})}} = \overline{A \,.\, \bar{B} + \bar{A} \,.\, B}$$

Computer Technology

Introduction

Fundamentals of Logic Gates

Definition

Logic gates can perform basic logical functions and are the fundamental building blocks of digital integrated circuits.

Uses

A logic gate can be used to:

- route data – data must be able to be passed between components in the computer system;
- store data – it is necessary to store data when it is not required by the CPU; and
- transform data – this includes arithmetic and logical operations on data.

Logic gates must be capable of controlling their uses. Control lines are used to choose:

- the mechanism to conduct the operation;
- what the inputs will be used; and
- where the output will be sent.

Logical Terminology

Logical True = 1 = High

Logical False = 0 = Low

Types of circuits

In a **combinatorial circuit**, the output always depends on the state of the current input(s). Therefore, these circuits have no "memory".

In a **sequential circuit**, the output can depend on the state of the current input(s) but also the "history" (past input values) of the circuit. Therefore, these circuits have a "memory".

Introduction

Introduction to CEDAR Logic

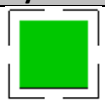
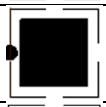
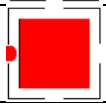
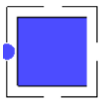
What is CEDAR Logic?

CEDAR Logic is an interactive digital logic simulator to be used for teaching of logic design or testing simple digital designs. It features both low-level logic gates as well as high-level components, including registers and a Z80 microprocessor emulator.

Output values

A LED can be used to show the value of an output.

Information: LED output values

Symbol	Output Value	Meaning
	No value	The LED has not been connected
	0	The current state of the circuit means that this output evaluates to 0
	1	The current state of the circuit means that this output evaluates to 1
	Undetermined	The logic of the circuit means that the output is not yet known *(common in sequential circuits where there is no "history")*
Light Blue	Conflicting Outputs	The outputs are clashing. *(in actual circuitry this would cause the circuit board to smoke)*

Encapsulation

Throughout this book the underlying circuitry and logic for any given component will be covered. After which, the encapsulated version of the component will be used in subsequent diagrams.

Once the mechanics of the components are understood, it is possible to build circuits using encapsulated components.

Many encapsulated components are included in CEDAR Logic and are a simulated version of the real circuit. Using these encapsulated components is beneficial to the economy of drawing the circuits and the understanding of the mechanics of a circuit.

Routing Data

Data Paths

Definition

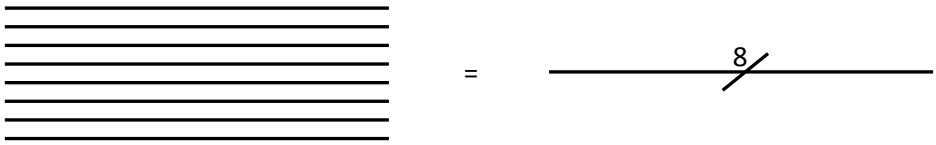
A **data path** is a set of functional units that carry out data processing operations.

Wires

Data paths consist of a single wire or multiple wires; many data paths consist of multiple wires as this allows more than one bit to be transmitted.

It is therefore necessary to distinguish multiple wires in the same group from multiple sources or destinations. In drawn diagrams, there is common shorthand for multiple wire in the same group.

Diagram: Multiple wires shorthand



However, in logic simulators, such as CEDAR Logic, it is necessary to make all of the wires explicit.

Routing Data

Multiplexers

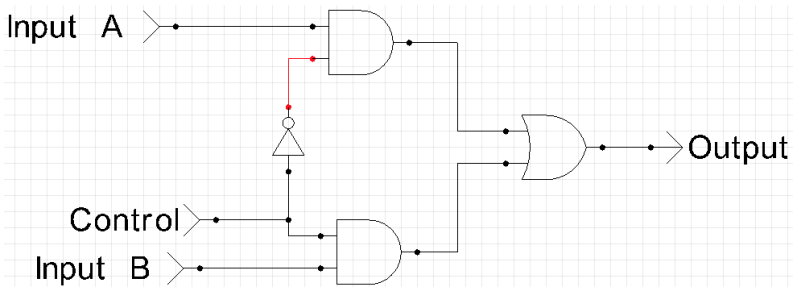
Definition

A **multiplexer** is a component that selects one of several input signals and forwards the selected input into a single line.

2-way Multiplexers

A 2-way multiplexer consists of two inputs, a control signal and an output.

Diagram: 2-way multiplexer



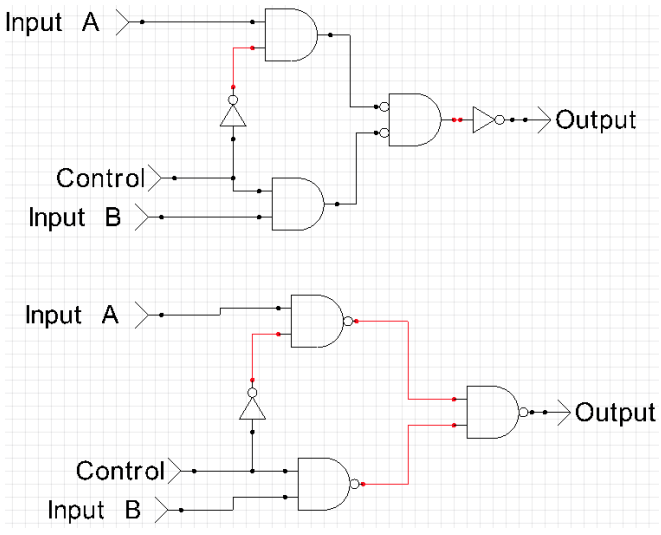
If input A will always toggle the output value. This is because of the inverted signal of the control signal and the AND gate.

If the control signal is low, input B will have no effect on the output value. If the control signal is high, input B will toggle the output value.

NAND gate implementation

A 2-way multiplexer can also be implemented using NAND gates.

Diagram: 2-way multiplexer using NAND gates



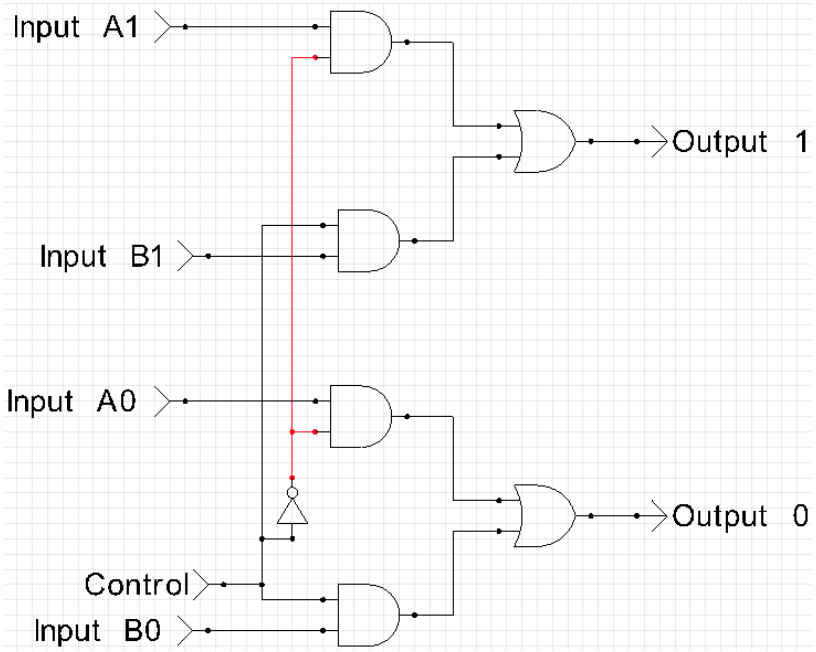
This allows for better optimisation as NAND gates are “universal gates”, meaning that they can be used to construct all other logic gates – this method is used to construct logic gates in most modern CPUs. This method of constructing logic gates is used because it is cheaper to construct CPUs using only one gate rather than many.

Routing Data

Multiple Multiplexer

A 2x2 multiplexer can be created by duplicating the data lines and making the control lines common.

Diagram: 2x2 multiplexer



A combination of inputs with various values for $Input\ A1$, $Input\ A0$, $Input\ B1$, $Input\ B0$ and $Control$ will allow either of the of the two possible one-bit outputs to be selected ($Output\ 1$, $Output\ 0$).

If the $Control$ is low:

- inputs from $Input\ A1$ will have a direct impact on $Output\ 1$; and
- inputs from $Input\ A0$ will have a direct impact on $Output\ 0$.

If the $Control$ is high:

- inputs from $Input\ B1$ will have a direct impact on $Output\ 1$; and
- inputs from $Input\ B0$ will have a direct impact on $Output\ 0$.

Truth Table: Some possible input/output combinations

INPUTS					OUTPUTS	
$Input\ A1$	$Input\ B1$	$Input\ A0$	$Input\ B0$	$Control$	$Output\ 1$	$Output\ 0$
0	0	0	0	0	0	0
0	0	0	1	0	0	0
0	0	1	0	0	0	1
0	1	0	0	0	0	0
1	0	0	0	0	1	0
0	0	0	0	1	0	0
0	0	0	1	1	0	1
0	0	1	0	1	0	0
0	1	0	0	1	1	0
1	0	0	0	1	0	0

There are $2^5 = 32$ possible inputs/outputs in total as some inputs may be high at the same time.

Routing Data

Three State Logic

Three State Logic

Use

Three state logic is used as an alternative to multiplexer.

In multiplexers, the inputs must be physically connected and this can become inconvenient when there are a large number of sources, particularly if they are physically separated. Three state logic overcomes this problem as all sources can be connected to a common wire, known as a bus.

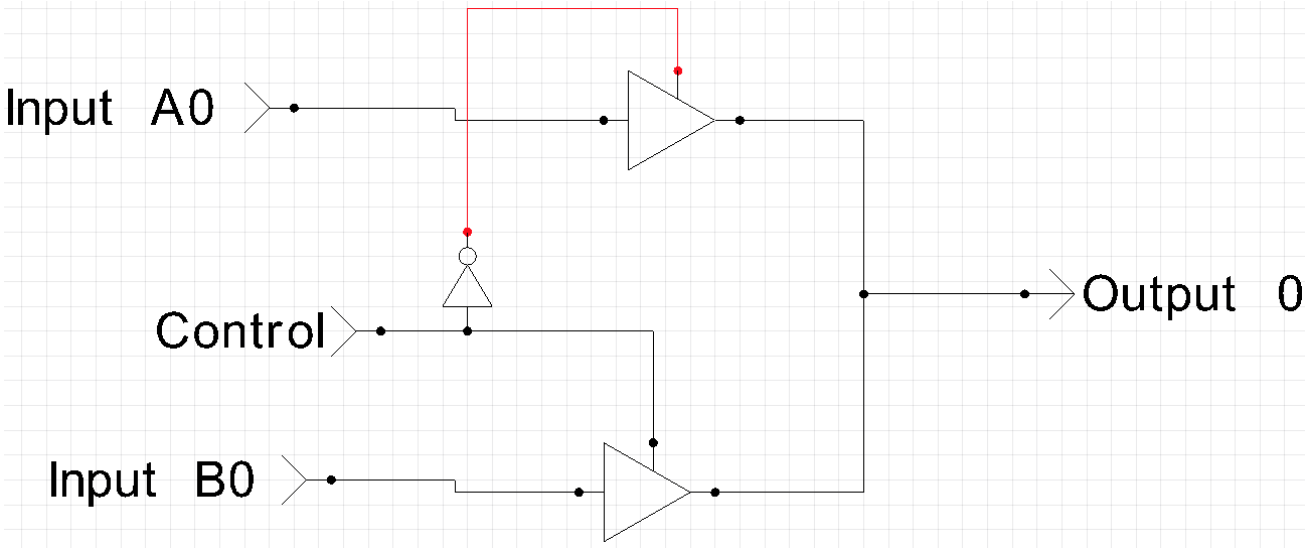
States

Three state logic has three possible states:

- high;
- low; or
- off.

Implementation

Diagram: Three state logic



If inputs A and B are both on, two currents are colliding and can cause the processor to break. The inverter is used to prevent this.

Storing Data

How is data stored?

Definition

Storing data involves mechanisms for remembering data that has been used before. Without a way to store this data, it makes moving data around a computer system useless.

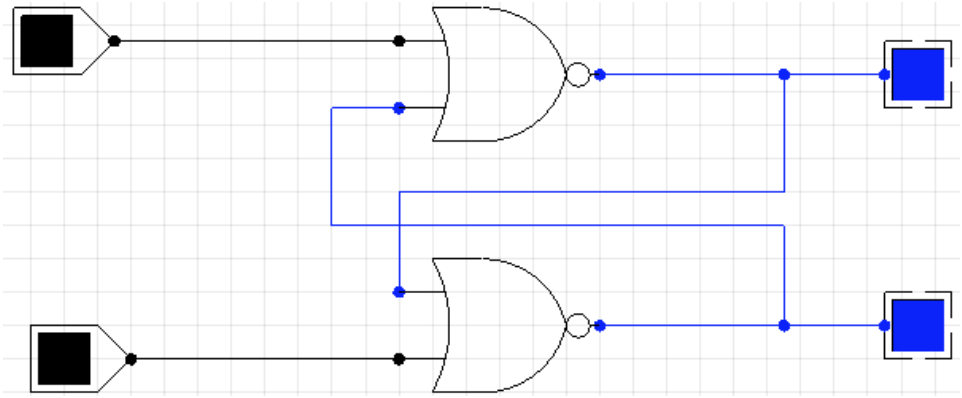
What is a flip flop?

A **flip flop**, or **latch**, is a circuit that has two stable states and can be used to store one bit of data.

Using NOR gates to construct a flip flop to store data

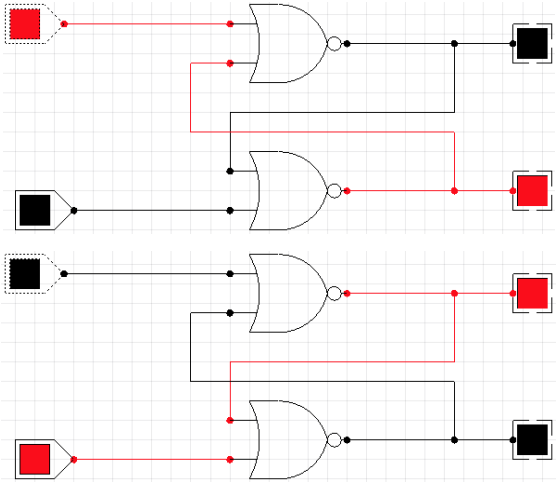
NOR gates can be used to construct a flip flop that is capable of storing data.

Diagram: NOR gate flip flop



The outputs are coloured blue as CEDAR Logic cannot determine the true output. This is because the inputs are having no impact on the output and no inputs have been set previously.

Diagram: Circuit behaviour



If the first input is high, the second output is high.

The second output will remain high even if the first input is transitioned to low.

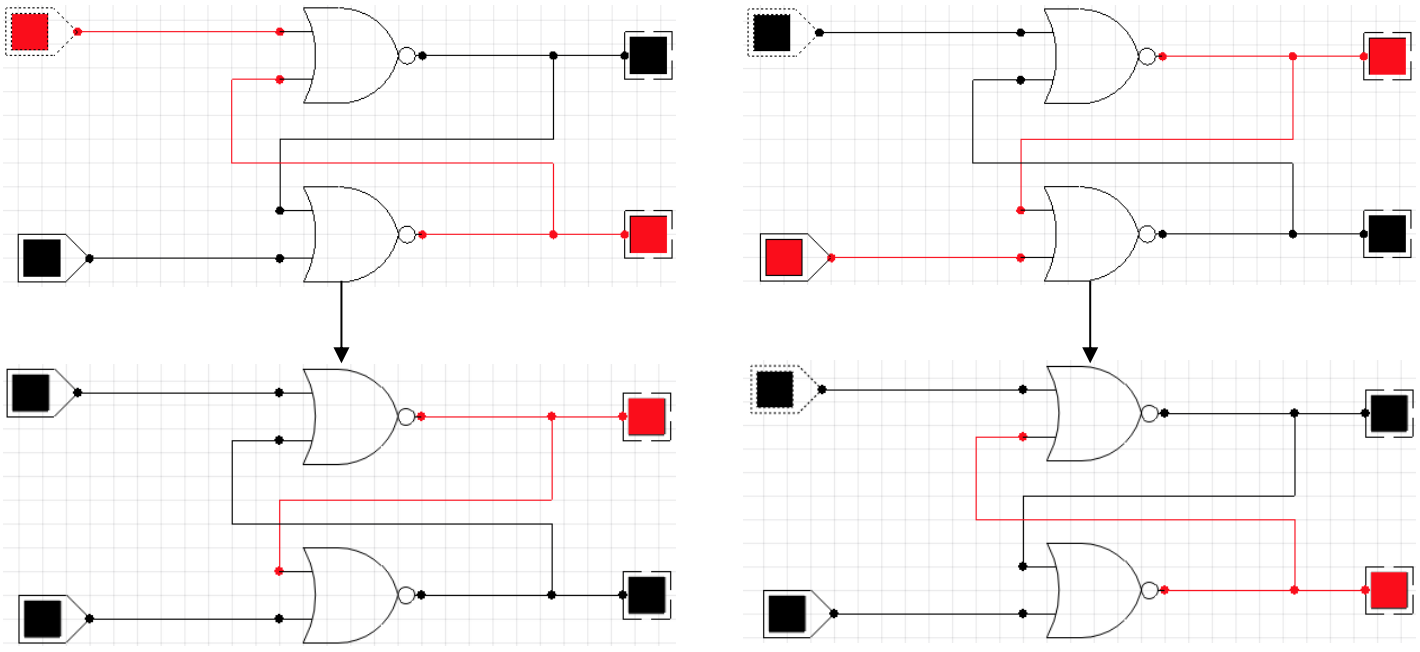
If the second input is high, the first output is high.

The first output will remain high even if the second input is transitioned to low.

Storing Data

The output in the “indeterminate” state depends on the history. This allows data to be “remembered”.

Diagram: History of states



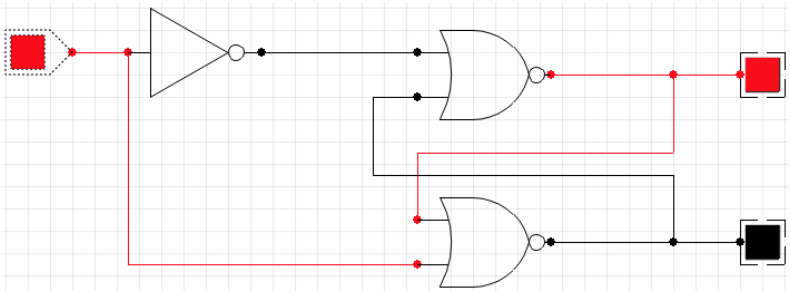
When setting the inputs to low, the output is “saved” from the previous state of the inputs.

Issues

The circuit remembers which input was high previously. In an ideal world, it would be better to just remember the value on a particular input.

In order to achieve this, an inverter could be used between the two inputs.

Diagram: Alternative circuit



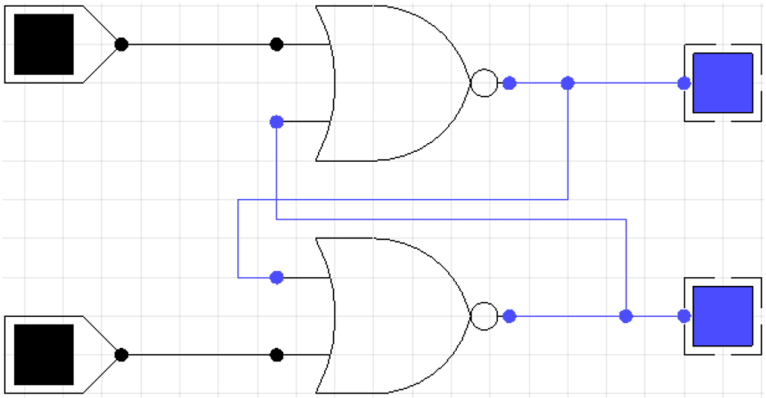
However, the circuit now has no “memory”.

Storing Data

Using NAND gates construct a flip flop to store data

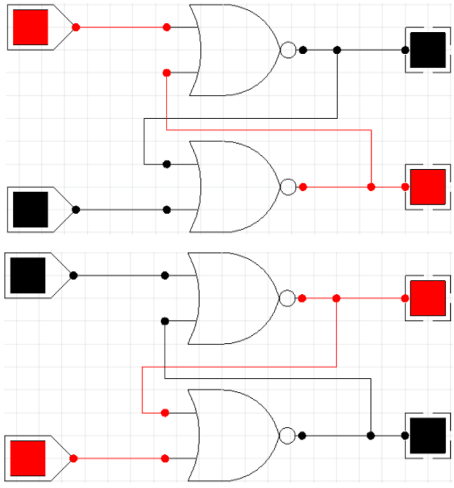
NAND gates can be used to construct a flip flop that is capable of storing data.

Diagram: NAND gate flip flop



The outputs are coloured blue as CEDAR Logic cannot determine the true output. This is because the inputs are having no impact on the output and no inputs have been set previously.

Diagram: Circuit behaviour



If the first input is high, the second output is high.

The second output will remain high even if the first input is transitioned to low.

If the second input is high, the first output is high.

The first output will remain high even if the second input is transitioned to low.

This allows for better optimisation as NAND gates are “universal gates”, meaning that they can be used to construct all other logic gates – this method is used to construct logic gates in most modern CPUs. This method of constructing logic gates is used because it is cheaper to construct CPUs using only one gate rather than many.

Storing Data

Timing

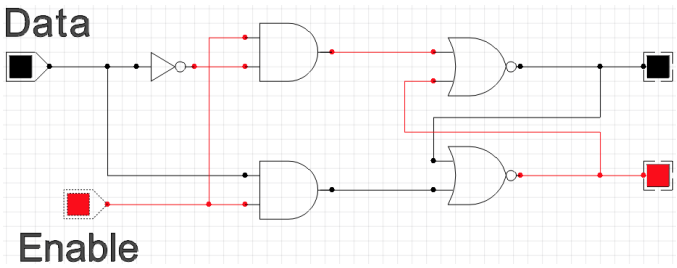
Why is timing required?

It is important to be able to specify a time when the data should be “remembered”.

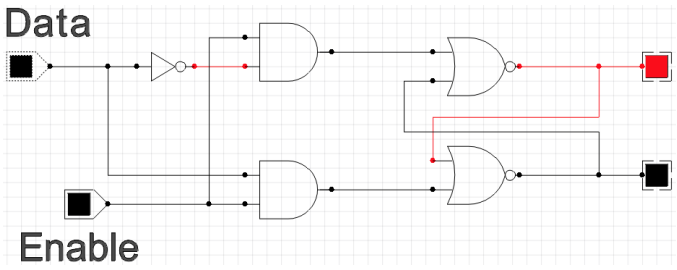
How does timing work?

Another input is required called an “enable”. This is used to determine when the circuit will remember data and when it will accept new data.

Diagram: Using “enable” input



When the “enable” input is high, the state of the “data” input will be “remembered”.



When the “enable” input is low, the state of the “data” input will have no impact on the output.

Instead, the state of the “data” input when the “enable” input was last high is “remembered”.

This allows a decision to be made on when the circuit will accept a change in data.

Issues

However, it is possible that data will “leak” through the system when the “enable” input is high. This would occur if the “data” input is changed more than once during the period of time that the “enable” input is high.

This issue could be addressed by making the time that the “enable” input is high for a very short period of time by constructing a circuit that gives a very short pulse. difficult to do and to be. However, this is difficult to engineer as it is not certain that the time the “enable” input is high is long enough for the data that should be “remembered” is accepted and not too long that any subsequent data “leaks” through the system.

A working solution would be to use a master-slave configuration.

Storing Data

Master-Slave

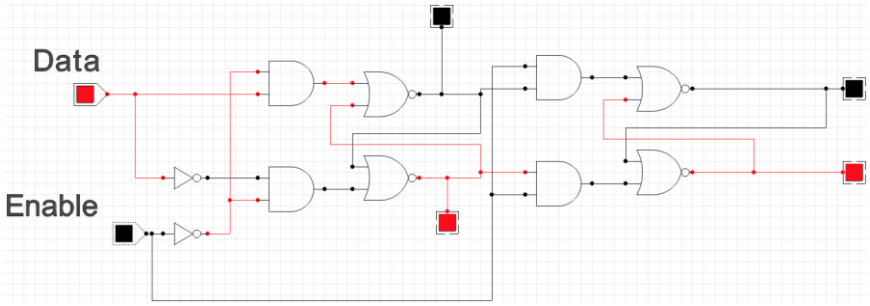
This method “remembers” the state of the “data” input when the “enable” input makes a transmission from high to low.

This is achieved by adding a second (slave) flip flop. The arrangement is such that:

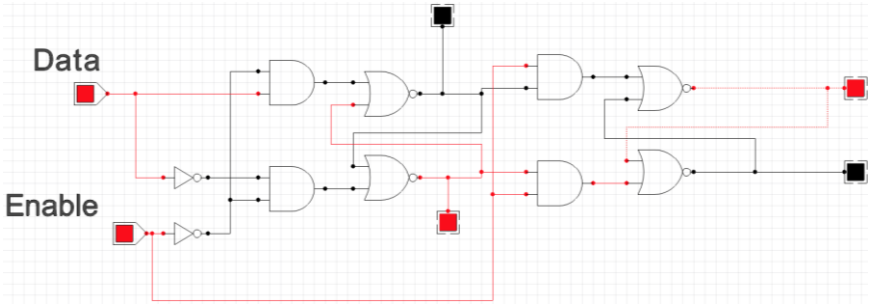
- the slave is enabled when the master is disabled; and
- the slave is disabled when the master is enabled.

This means that the value of the master is transferred to the slave only when the master makes an enabled-disabled transition.

Diagram: Master-slave



When master enable is low, outputs are unknown (blue) as it hasn't got anything to "remember". At this time, data will not impact the circuit.



When master enable is high, the master will "remember" the data but the slave will not react to the actions of the master. The slave is remaining in its state.

When enabling the slave, the data is received from the master.

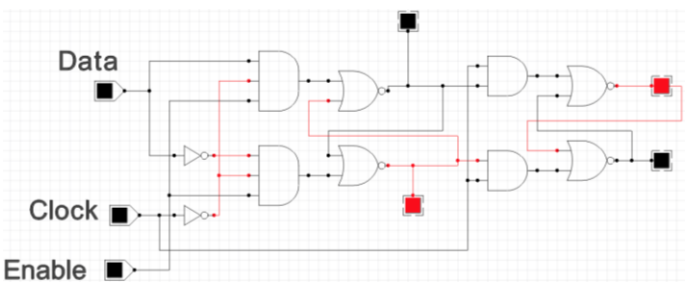
The inverter near enable ensures that the master is low when the slave is high and the master is high when the slave is low. This means that the data is transferred from the master to the slave at the "transition" and no data is "leaked".

Using clock signal

When the “enable” input is wired as such, it is often called a “clock”. The clock is common to other components of the computer system, such as the CPU.

A second “enable” input can be added in order to allow some memory devices to only respond on some clock cycles but not others.

Diagram: Master-slave using clock signal

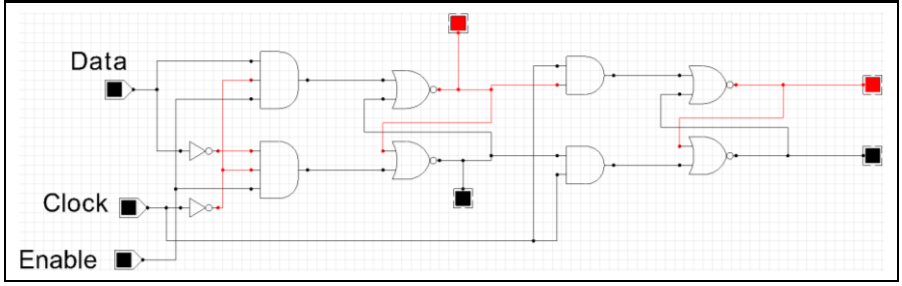


Storing Data

However, there is a defect with this circuit as:

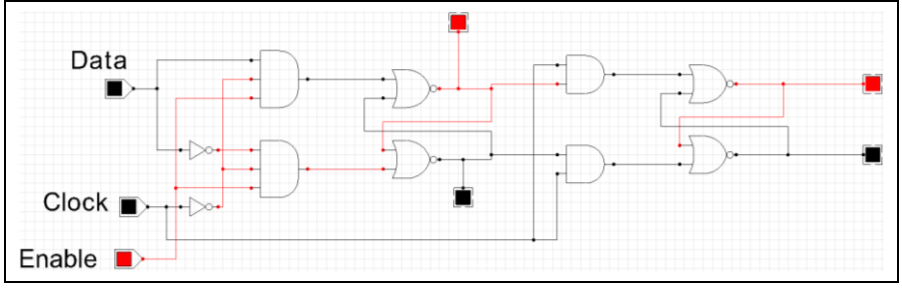
- when the clock is low, the transition of enable from high to low has no impact on the state of the slave;

Diagram: Defect



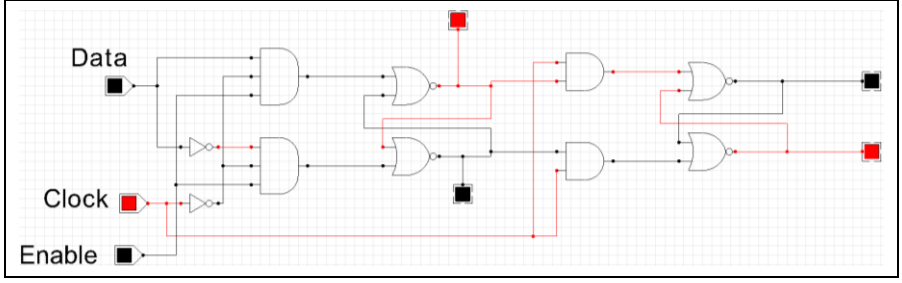
- when the enable makes a transition from low to high, the state of the "data" input is transferred to the master; and

Diagram: Defect



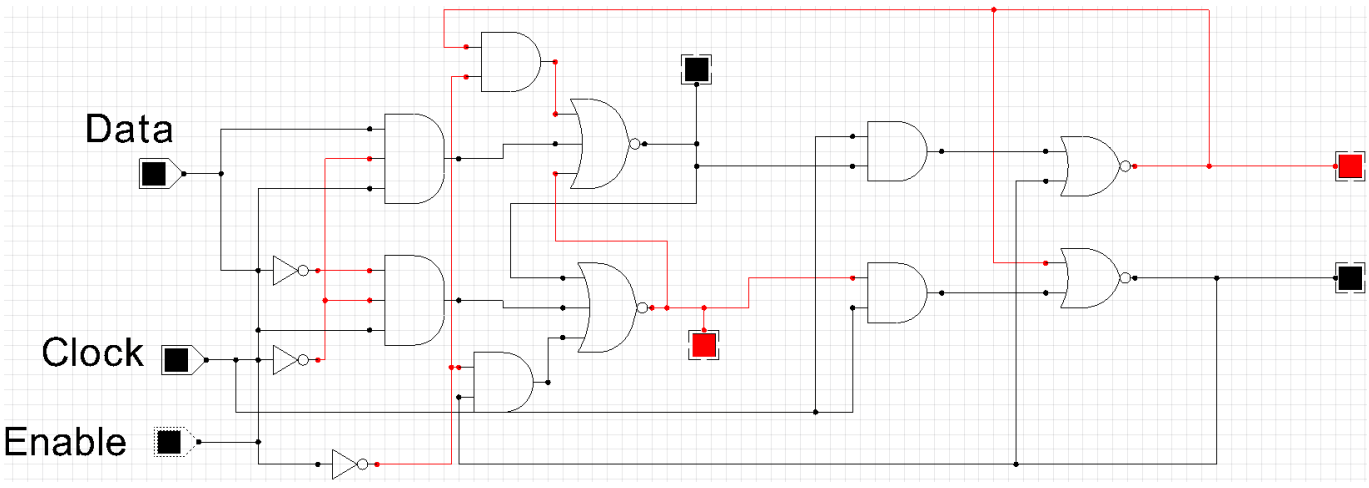
- when the clock makes a transition from low to high, the state of the "data" input is transferred to the slave, despite the fact that the enable is low.

Diagram: Defect



The circuit can be fixed by routing the output of the slave back into the master.

Diagram: Fixed master-slave using clock signal

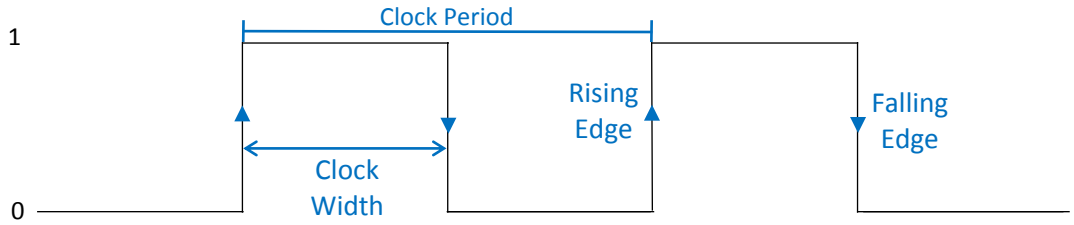


In this fixed circuit, the master tracks the slave when the enable is low.

Storing Data

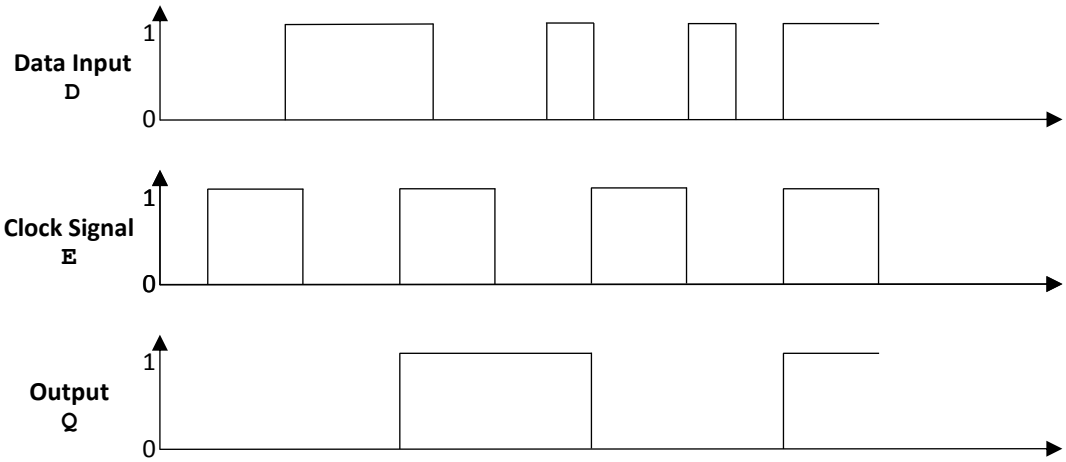
A clock signal is a series of signals representing a 0 or a 1.

Diagram: Clock signal



The clock signal is used to determine when data in the circuit should be changed.

Diagram: Clock signal



The output of an edge-triggered flip flop shows that:

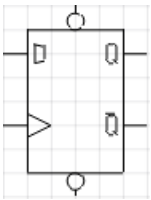
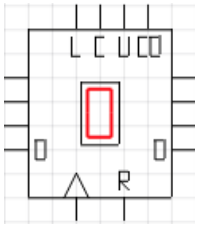
- when the clock signal is at a rising edge, the data input can change the output; and
- when the clock signal is not at a rising edge, the output remains constant.

Storing Data

CEDAR Logic Circuit Components

Components

Information: Encapsulated components

<table>
<tr><th>Component</th><th>Diagram</th><th colspan="2">Input/Output</th></tr>
<tr><td rowspan="5">One-bit flip-flop</td><td rowspan="5"> 
</td><th>Symbol</th><th>Meaning</th></tr>
<tr><td>Arrow</td><td>Clock
(timed input, edge triggered)</td></tr>
<tr><td>D</td><td>Data input</td></tr>
<tr><td>Q / $\bar{Q}$</td><td>Data output</td></tr>
<tr><td>Circle & line</td><td>SET and RESET inputs
(circle shows inverted inputs)</td></tr>
<tr><td rowspan="7">Four-bit flip-flop</td><td rowspan="7"> 
</td><th>Symbol</th><th>Meaning</th></tr>
<tr><td>Arrow</td><td>Clock
(timed input, edge triggered)</td></tr>
<tr><td>D</td><td>Four data inputs</td></tr>
<tr><td>R</td><td>RESET
(common to all)</td></tr>
<tr><td>L</td><td>LOAD
(overall output)</td></tr>
<tr><td>C</td><td>COUNT
(gates for arithmetic)</td></tr>
<tr><td>U</td><td>UP
(instructs to count up or down)</td></tr>
<tr><td>CO</td><td>CARRY OUT</td></tr>
</table>

Routing and Storage

Buses

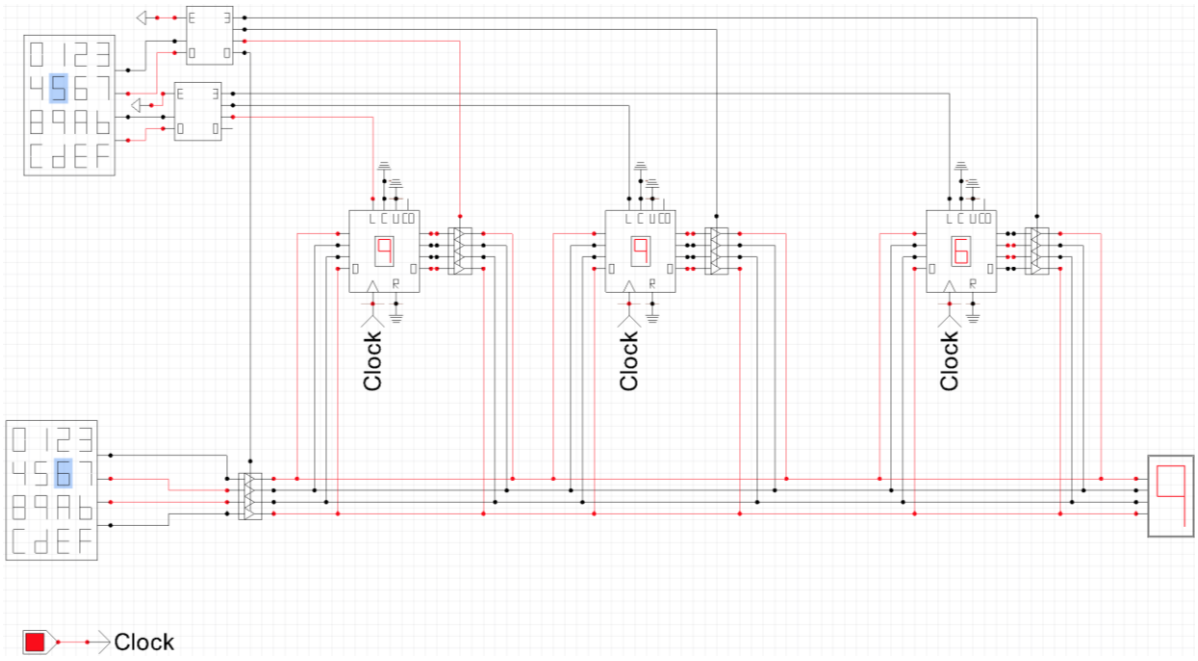
Definition

A **bus** is a communication system that transfers data between components inside a computer system, or between computer systems.

Combining routing and storage

When combining routing and storage together, the concept of a bus is reached.

Diagram: A bus



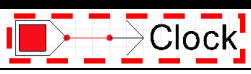
This circuit is capable of moving data along the four wires, known as a bus.

The circuit is comprised of:

- clock – allows the registers to share a common clock;
- bottom keypad – allows data to be entered;
- top keypad – used to define the source and destination of the data;
- registers – used to store one bit of data; and
- decoders – used to decide where data will flow.
- tristate buffers – ensures that only one set of data is on the bus at any given time.

Clock

Diagram Part: Clock



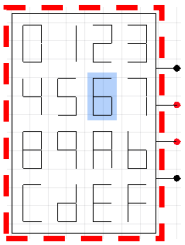
Data can be moved from any register, or keypad, to any other register in one clock cycle. Data will only flow throughout the circuit when the clock makes a transition from low to high.

Routing and Storage

Bottom Keypad

The bottom keypad allows data to be entered.

Diagram Part: Bottom keypad (encapsulated)



The keypad allows selection of hexadecimal symbols which will be converted to a four-bit binary number.

Each bit will be put on a line of the bus.

Example: Input of 6

The hexadecimal symbol 6 corresponds a binary value.

8	4	2	1
0	1	1	0

Therefore, when the bottom keypad has an input of 6, the binary digits 0110 are places on the bus lines.

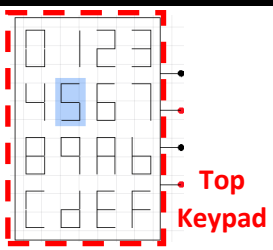
Guide: Hexadecimal conversion

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

Top keypad

The top keypad is used to define the source and destination of the data.

Diagram Part: Top keypad (encapsulated)



The keypad allows selection of hexadecimal symbols in the same way as the bottom keypad.

The four-bit output determines the source and destination of the data:

- the first two bits are used for the source of the data; and
- the second two bits used for the destination of the data.

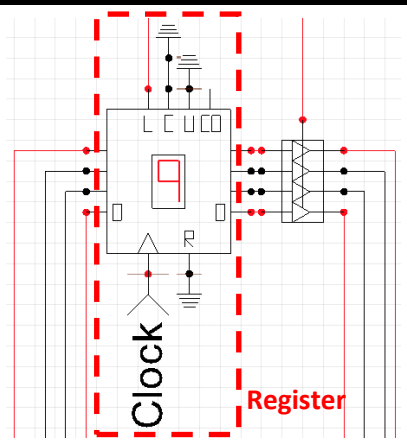
Routing and Storage

Registers

A **register** is a group of concatenated flip flops.

In this circuit, there are three registers.

Diagram Part: A register (encapsulated)

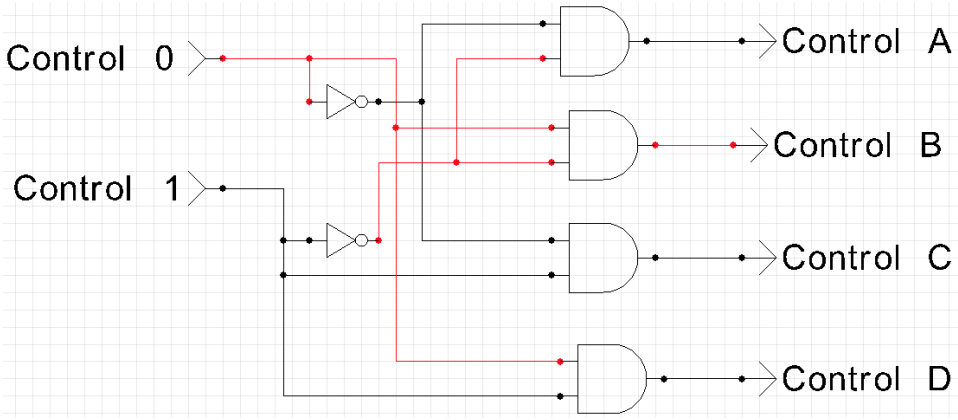


Symbol	Meaning
Arrow	Clock *(timed input, edge triggered)*
D	Four data inputs
R	RESET *(common to all)*
L	LOAD *(overall output)*
C	COUNT *(gates for arithmetic)*
U	UP *(instructs to count up or down)*
CO	CARRY OUT

Decoders

A **decoder**, or **demultiplexer**, is a combinatorial circuit that takes a two-bit input and translates this into a one-bit output.

Diagram: Decoder (full circuitry)



A combination of inputs with various values for $Control\ 0$ and $Control\ 1$ will allow any of the four possible one-bit outputs to be selected ($Control\ A$, $Control\ B$, $Control\ C$, $Control\ D$).

Truth Table: All possible input/output combinations

INPUTS		OUTPUTS			
$Control\ 0$	$Control\ 1$	$Control\ A$	$Control\ B$	$Control\ C$	$Control\ D$
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

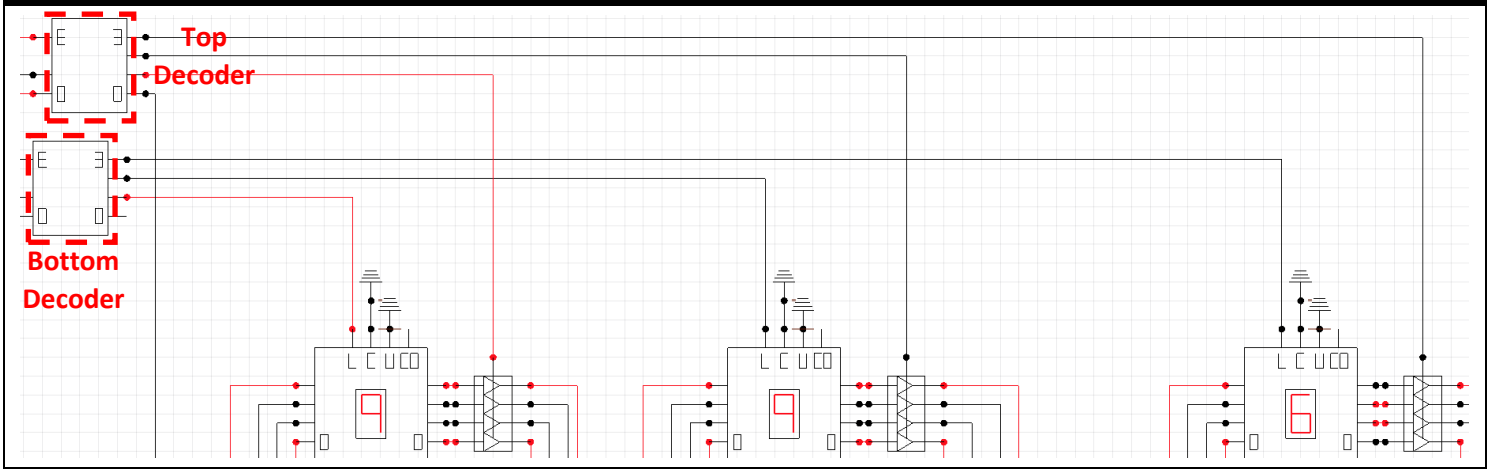
There are $2^2 = 4$ possible inputs/outputs in total as some inputs may be high at the same time.

Routing and Storage

CEDAR Logic provides an encapsulated decoder component which has an added enable line E that enables the decoder.

In this circuit, there are two decoders: the top decoder; and the bottom decoder.

Diagram Part: Decoders (encapsulated)

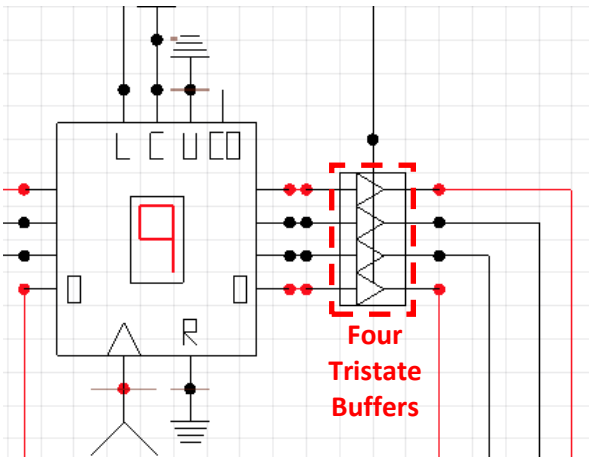


The top decoder is connected to the tristate buffers to determine where the data will come from.
The bottom decoder is connected to the LOAD inputs on the registers and determines which register will receive data.

Tristate buffers

In this circuit, tristate buffers are used to ensure that only one set of data is on the bus at any given time.

Diagram Part: Tristate buffer



If the top decoder makes the $Enable$ line for the four tristate buffers high, the data from the register will be put on the bus.
If the top decoder makes the $Enable$ line for the four tristate buffers low, the data from the register will not be put on the bus.

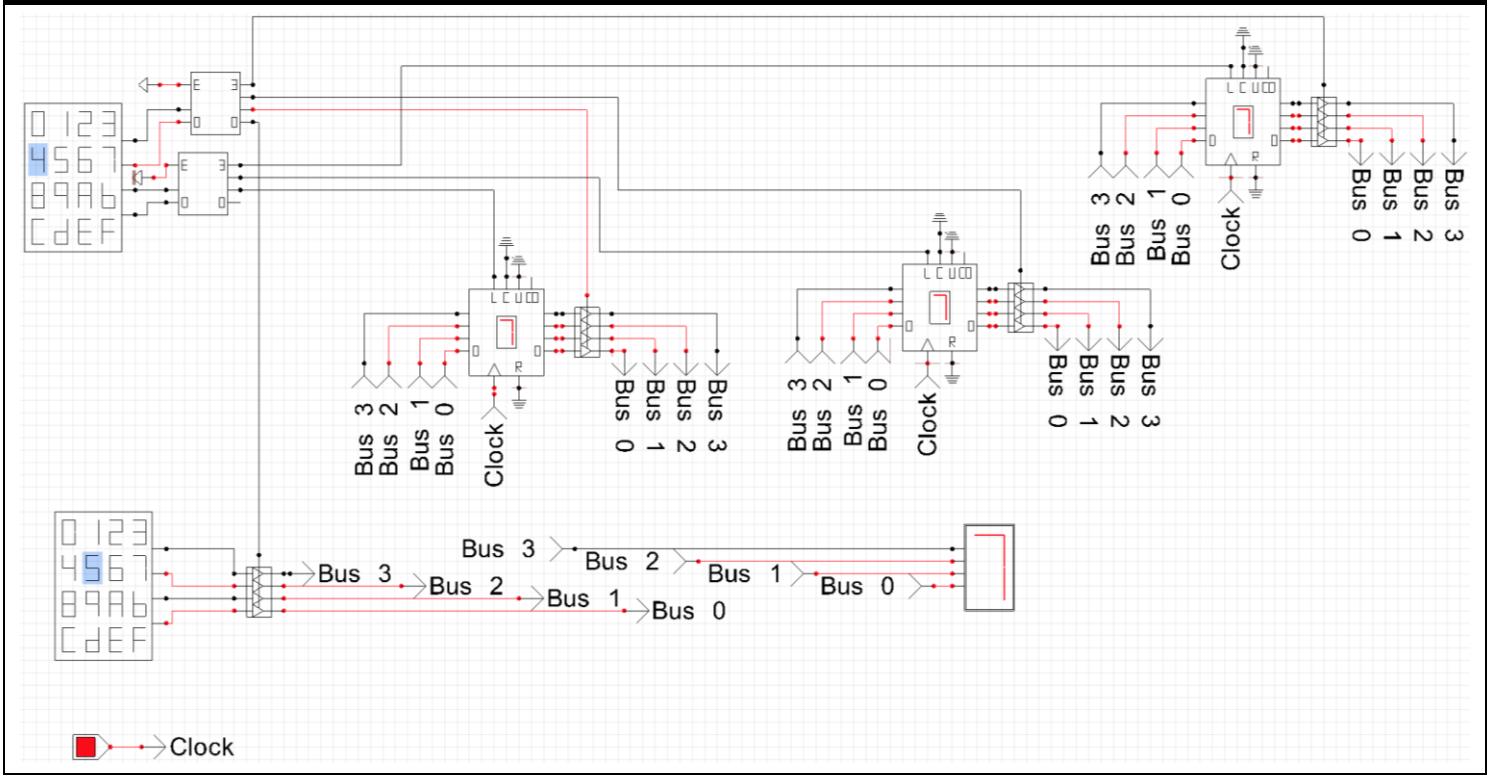
This prevents the value from two registers being put on the bus at the same time, this is important as it prevents conflicting outputs.

Routing and Storage

Alternative bus diagram

The previous diagram of a bus shows how the wires are actually connected. However, for practicality, it is often easier to build and understand the bus circuit diagram using connection labels in CEDAR Logic.

Diagram: A bus



The connection labels are directly mapped to one another based on their exact name.

Routing and Storage

Moving data around the bus

Using the top keypad, it is possible to control the flow of data around the bus.

Information: Top keypad inputs

INPUT			ACTION PERFORMED
Hexadecimal	Denary	Binary	
0	0	0000	• No action to registers. • Bottom keypad value appears on output.
1	1	0001	• Keypad value is copied to register 1.
2	2	0010	• Keypad value is copied to register 2.
3	3	0011	• Keypad value is copied to register 3.
4	4	0100	• Bottom keypad is disabled. • Enables output from register 1. ○ The data from register 1 is put on the bus.
5	5	0101	• Bottom keypad is disabled. • Enables output from register 1 and enables LOAD on register 1. ○ The data from register 1 is copied to register 1.
6	6	0110	• Bottom keypad is disabled. • Enables output from register 1 and enables LOAD on register 2. ○ The data from register 1 is copied to register 2.
7	7	0111	• Bottom keypad is disabled. • Enables output from register 1 and enables LOAD on register 3. ○ The data from register 1 is copied to register 3.
8	8	0100	• Bottom keypad is disabled. • Enables output from register 2. ○ The data from register 2 is put on the bus.
9	9	0101	• Bottom keypad is disabled. • Enables output from register 2 and enables LOAD on register 1. ○ The data from register 2 is copied to register 1.
A	10	1010	• Bottom keypad is disabled. • Enables output from register 2 and enables LOAD on register 2. ○ The data from register 2 is copied to register 2.
B	11	1011	• Bottom keypad is disabled. • Enables output from register 2 and enables LOAD on register 3. ○ The data from register 2 is copied to register 3.
C	12	1100	• Bottom keypad is disabled. • Enables output from register 3. ○ The data from register 3 is put on the bus.
D	13	1101	• Bottom keypad is disabled. • Enables output from register 3 and enables LOAD on register 1. ○ The data from register 3 is copied to register 1.
E	14	1110	• Bottom keypad is disabled. • Enables output from register 3 and enables LOAD on register 2. ○ The data from register 3 is copied to register 2.
F	15	1111	• Bottom keypad is disabled. • Enables output from register 3 and enables LOAD on register 3. ○ The data from register 3 is copied to register 3.

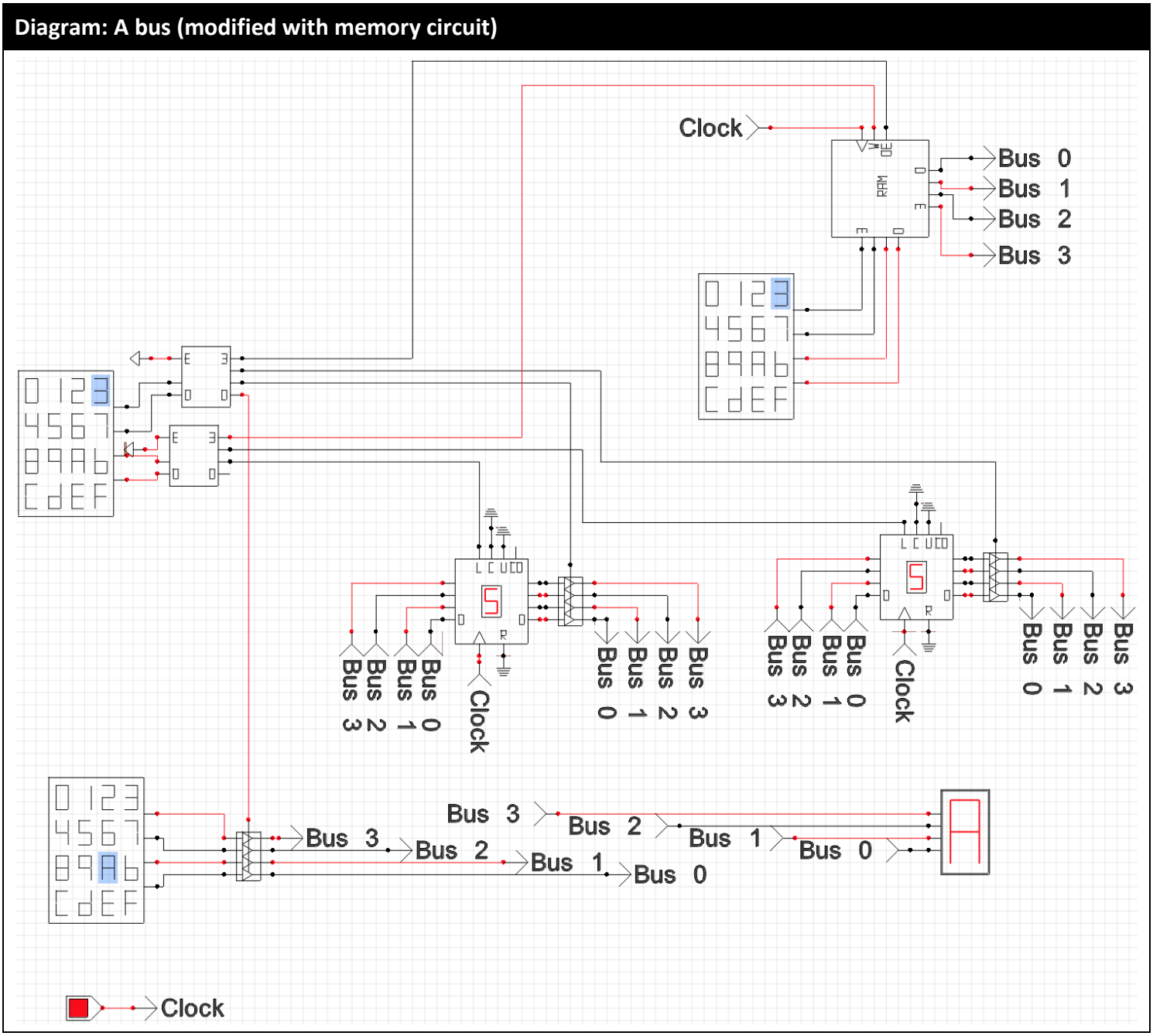
These actions will take place when the clock makes a transition from low to high.

Routing and Storage

Using memory circuits

It is possible to replace a register with a memory circuit (RAM).

Diagram: A bus (modified with memory circuit)



The memory circuit (RAM) has 16 addressable locations (0-15).

The memory circuit (RAM) can be accessed in the same way as register 3 was accessed in the previous circuit. Subsequently. The memory locations (0-15) are accessed using the keypad attached to the address lines on the memory circuit (RAM).

Routing and Storage

Overview of buses

This layout is not meant to be fully representative of actual memory circuits. It is broadly similar to static RAM however, dynamic RAM is different as it uses multiplexers to select rows and columns.

In addition, in actual memory circuits, input and output are often connected together.

Overall, this bus circuitry is flexible. However, it can become cumbersome if there are many registers.

Routing and Storage

Memory

Definition

Memory is any physical device capable of storing information temporarily or permanently. For example, Random Access Memory (RAM), is a volatile memory that stores information on an integrated circuit used by the operating system, software, and hardware.

How is data stored in memory?

Memory consists of a large number of registers.

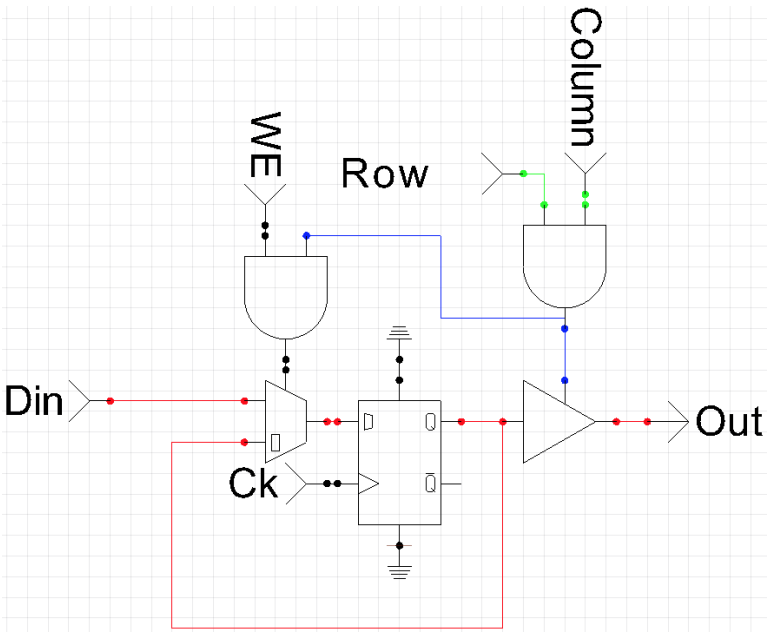
Memory:

- uses the same control bits, known as “address”, for output and input (i.e. reading and writing);
- may connect input and output together, allowing direct connection to a bus; and
- has overall control of reading and writing via:
    - output enable (OE); and
    - write enable (WE).

Memory cells

A memory cell allows storage of one-bit of data.

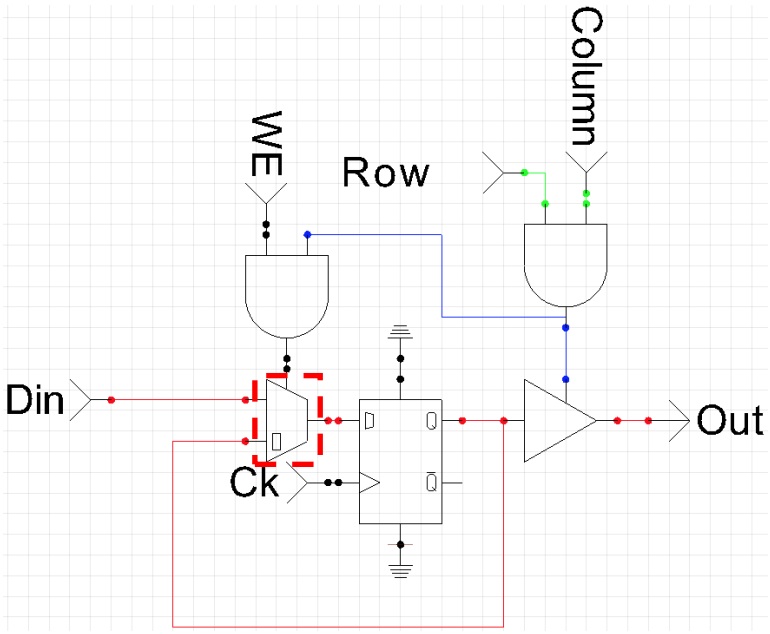
Diagram: A memory cell



Routing and Storage

Multiplexers in memory cells

Diagram Part: Multiplexer

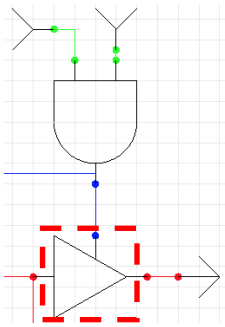


Din passes data to the multiplexer which selects between the data that is currently stored in the memory cell or new data coming in to the memory cell via Din. The selected data will then be stored when the clock makes a transition from low to high.

Tristate buffer in memory cells

The tristate buffer is activated when $Row\,.\,Column = True$.

Diagram Part: Tristate buffer



The output of the flip flop will go to the tristate buffer and will only be output if the tristate buffer has been activated.

When memory cells are arranged in grids, the row and column of the memory cell will be used to determine which memory cell should be accessed. Therefore, if the row and column values are both $True$ for a given memory cell it is that memory cell whose value should be output.

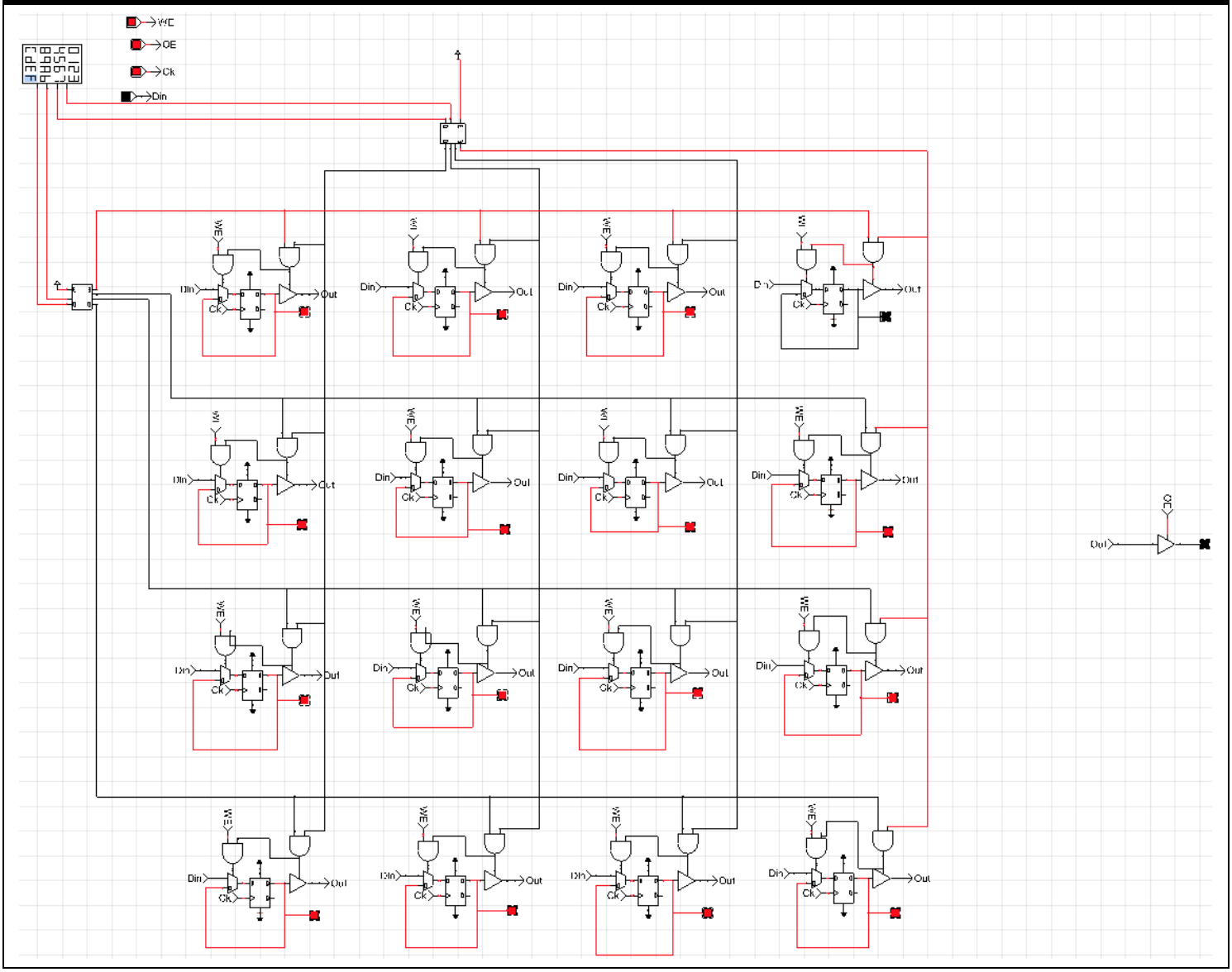
The tristate buffer only needs to be on if:

- the output enable (OE) is $True$ as data will be read from the memory cell; or
- the write enable (WE) is $True$ as data will be written to the memory cell.

Memory grids

Memory cells are arranged in memory grids.

Diagram: A memory grid

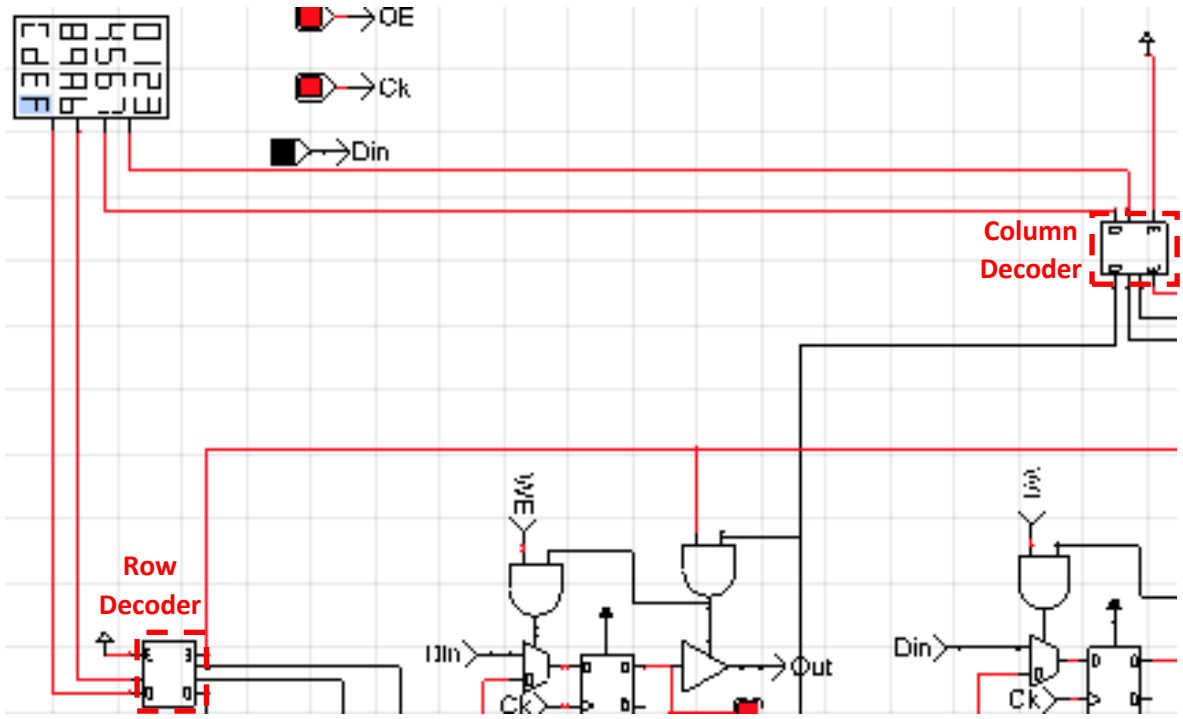


The memory cells form part of a larger two-dimensional (2D) array. This is because the silicon in memory chips “prints” them in 2D.

Routing and Storage

Decoders in memory grids

Diagram part: Decoders



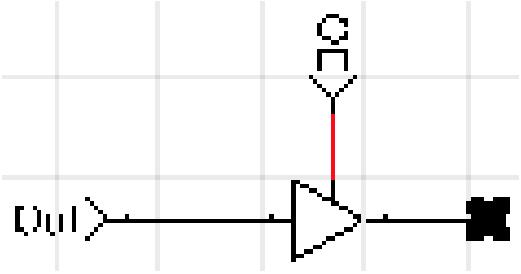
The column decoder takes the first two bits from the numpad input and sets the $column$ input on the selected memory cell to high.

The row decoder takes the last two bits from the numpad input and sets the row input on the selected memory cell to high.

"Ultimate output" in memory grids

The "ultimate output" uses a tristate buffer.

Diagram Part: "Ultimate output"



Each memory cell's Out is connected to this tristate buffer and the data from the selected memory cell will be output here if the output enable (OE) is high.

Architectures

Introduction to Architectures

Harvard Architecture

Two separate systems are used to be the controller and the controlled system:

- controller – the system performs a "fetch" cycle and loads the contents from memory into the Instruction Register (IR), this controls the control system;
- controlled system – the system performs an "execute" cycle using the data in the Instruction Register (IR) that determines what actions will take place, this gives feedback to the controller to allow response to events.

The Harvard architecture circuit is comprised of an address bus and a data bus. These buses act independently of one another.

The Harvard architecture is often used in the real world for "machine within a machine" subsystems and microprograms to execute complex instructions, where the computer system has a fixed program and therefore there is no concern regarding the loading process of loading instructions into the controller and therefore, there will less communication issues.

Von Neumann Architecture

One system is used that alternates between being the controller and the controlled system on each clock cycle:

- acting as the controller – the system performs a "fetch" cycle and loads the contents from memory into the Instruction Register (IR);
- acting as the controlled system – the system performs an "execute" cycle using the data in the Instruction Register (IR) that determines what actions will take place.

Evaluation: Von Neumann architecture

Advantages	Disadvantages
Cheaper to manufacture.	**Nominally runs at half the speed of the Harvard Architecture** as a bus is shared for data and instructions, known as the "Von Neumann bottleneck".

Due to the "Von Neumann bottleneck", processor designers have implemented solutions (such as pipelining, extra buses and special purpose memories) in order to improve the speed of execution. However, the standard Von Neumann architecture is still logically consistent with modern day computers.

Architectures

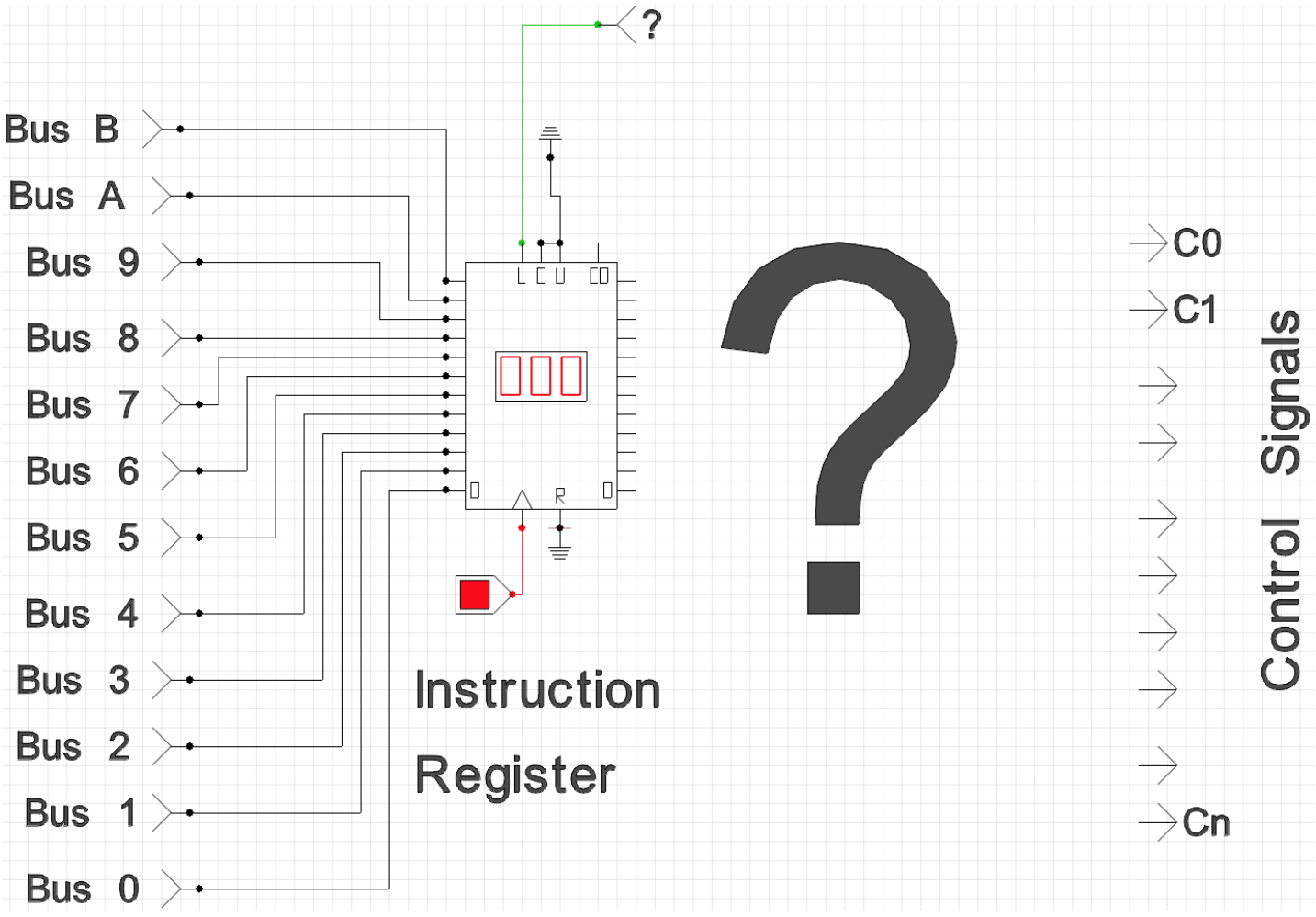
Comparison

Comparison: Architectures		
	Harvard	**Von Neumann**
Memory	Data and instructions are stored on separate memory.	Data and instructions are both stored on the same memory.
System Bus	There are two system buses that connect to the memory, one bus for data and one bus for instructions, from the CPU.	The system bus is connected to the CPU and memory.
Cycle Speed	The CPU will spend less time idle while waiting for data and instructions as they are not limited by the speed of each other as they travel along different buses.	The CPU may be idle while waiting for data and instructions as they travel at different rates along the same bus.
Identification	Data and instructions are easy to distinguish as the travel along different buses, making corruption less likely.	Data and instructions are difficult to distinguish as they travel along the same bus, allowing corruption.

The common problem

There is a common problem for both Harvard and Von Neumann architectures: “how are instructions decoded?”.

Diagram: Decoding instructions



Control signals must be sent out from the Instruction Register (IR) to components such as multiplexers, three state buffers, LOAD inputs on other circuits and arithmetic control lines. These signals determine how data is routed and the circuits that should be enabled.

As instructions can vary in their size, it is necessary to have a general strategy, that contains the correct links between the Instruction Register (IR) and the control signals, to process all instructions of varied length. The solution will successfully decode the instruction from a small number of bits, allowing a reasonable set of useful operations, into a large number of bits that are used and sent as control signals.

Architectures

Strategy #1

Each output bit from the Instruction Register (IR) could be mapped to a control signal.

Evaluation: Strategy #1	
Advantages	**Disadvantages**
	The instruction word would be too long as real world systems are far more complex. The instruction word is the set of bits that defines the memory location for an instruction; the $instruction\ register\ (IR)$ is loaded with the instruction word that is routed into the $decoder$.
	Some combinations of bits could damage the computer system's hardware through three state output clashes.
	Many combinations of bits would not perform any useful operation.

Strategy #2

For each instruction, defined as an input bit pattern, determine the output control bits required. This would allow a circuit to be designed that can produce the correct control signals from the instruction in the Instruction Register (IR).

Logical operations conducted using methods such as Boolean logic equations and Karnaugh maps could be used to determine the minimal implementation of the required logic.

Evaluation: Strategy #2	
Advantages	**Disadvantages**
	It is not possible to predict how much hardware is required until the design is complete.
	Some combinations of bits could damage the computer system's hardware through three state output clashes.
	Small changes to the requirements for the system can product relatively large changes to the implementation.

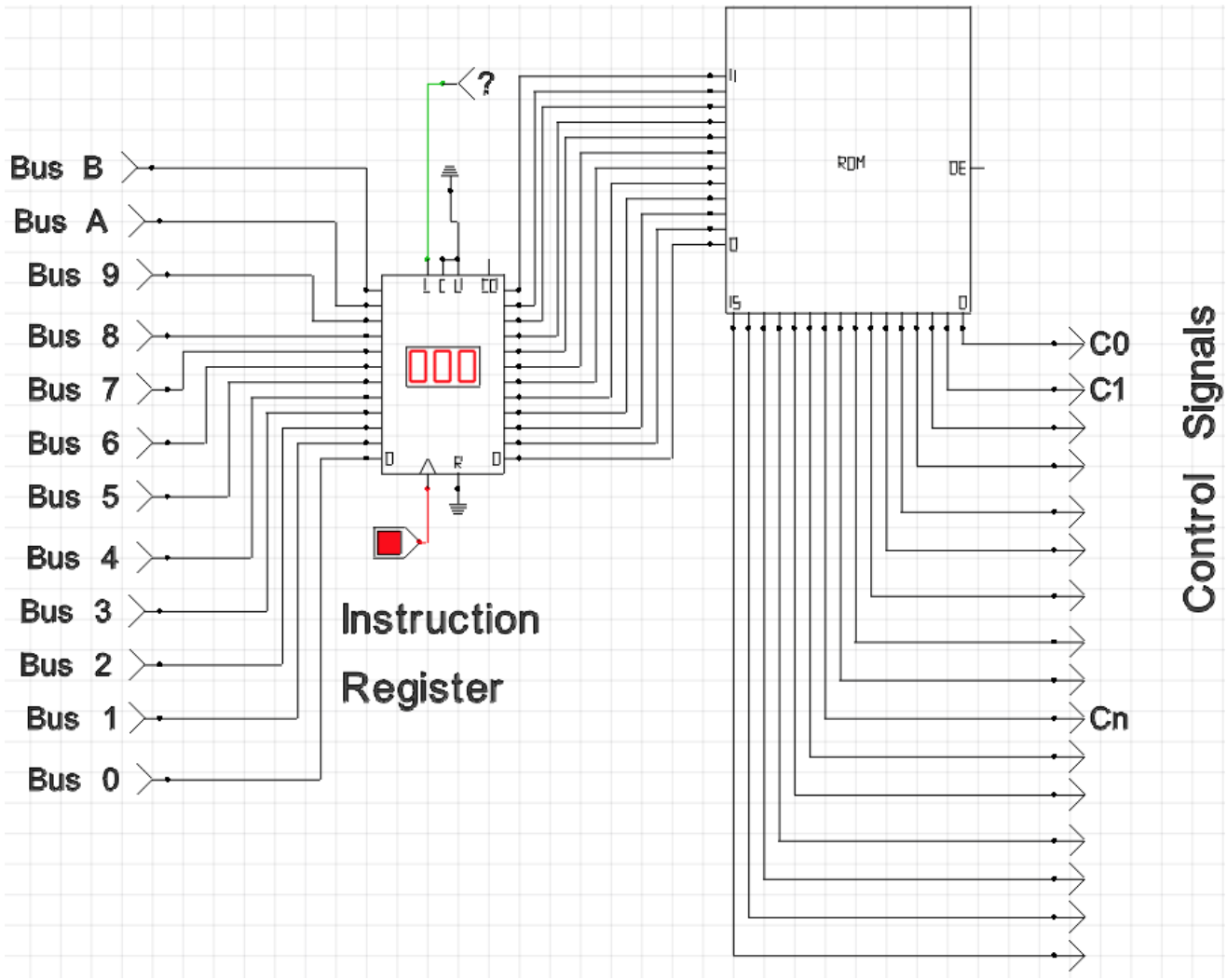
As a result, this strategy has rarely been used for around forty years.

Strategy #3

The entire circuitry can be "wrapped up" into a single memory device. This is a viable solution as a memory device can provide any desired output for each possible input by changing the contents of the memory.

Architectures

Diagram: Strategy #3



The output bits from the Instruction Register (IR) are connected to the input for the memory (ROM).
The output bits from the memory (ROM) are connected to the control signal lines.

Evaluation: Strategy #3

Advantages	Disadvantages
The design of the circuitry does not need to be changed if the requirements for the system change as a new desired output can be made by changing the contents of memory.	**The memory becomes too large** for a large number of inputs.
Maximum flexibility as the only limit to the amount of possible input and output combinations is the number of address bits.	**May represent overkill** in many situations as not all combinations are required.

Strategy #4

This strategy uses programmable logic.

Architectures

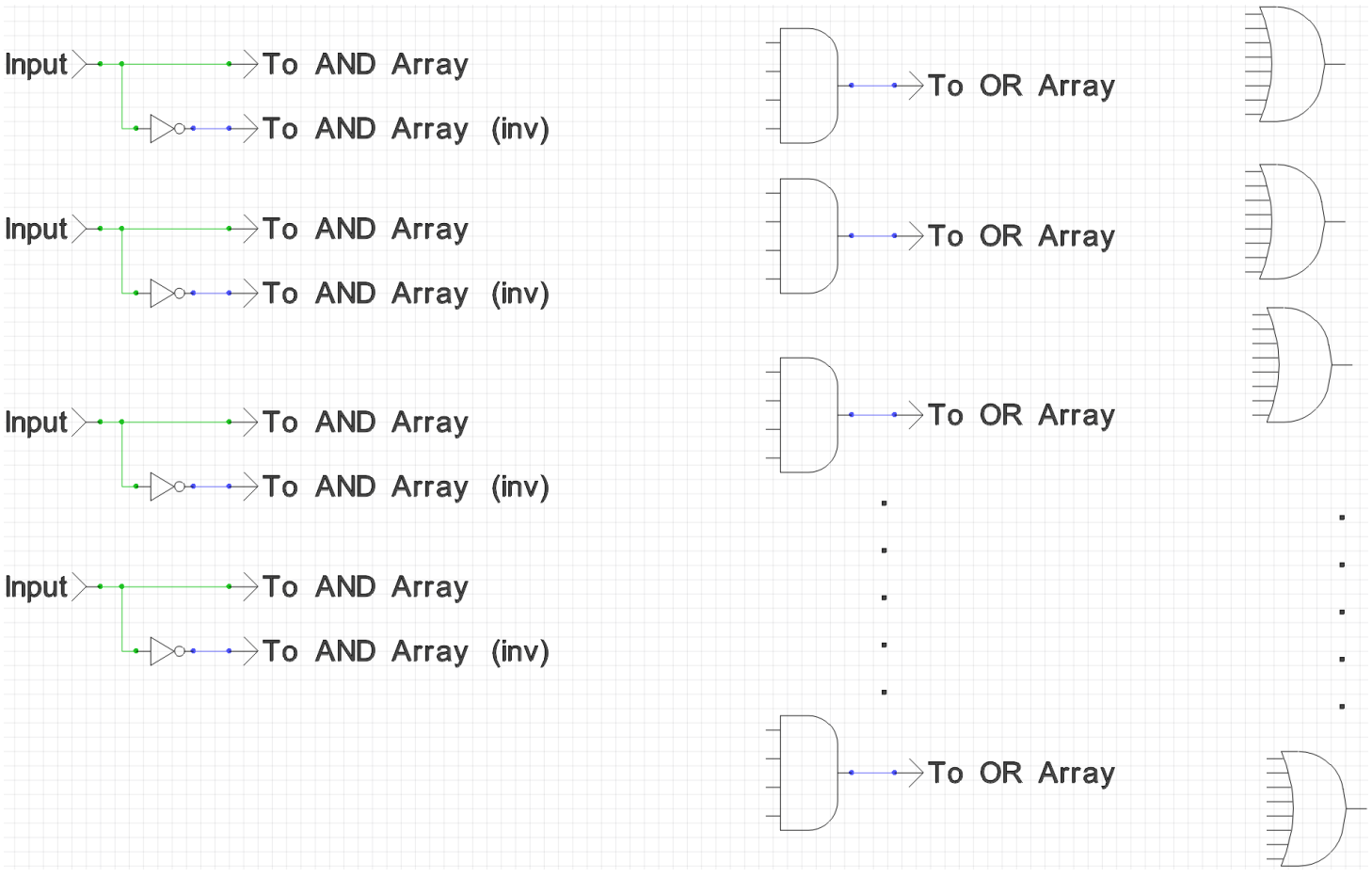
Programmable Logic

How does programmable logic work?

Take the outputs from the Instruction Register (IR) and:

- generate an inverse;
- feed the inverse into a layer of AND gates; and
- feed the output from the AND gates into a layer of OR gates.

Diagram: Programmable logic



Any logical equation can be expressed in this way. This is achieved by constructing a chip that consists of a layer of NOT gates, a layer of AND gates and then a layer of OR gates. The connections made between these layers determines the output and any connections can be made in order to achieve the desired output.

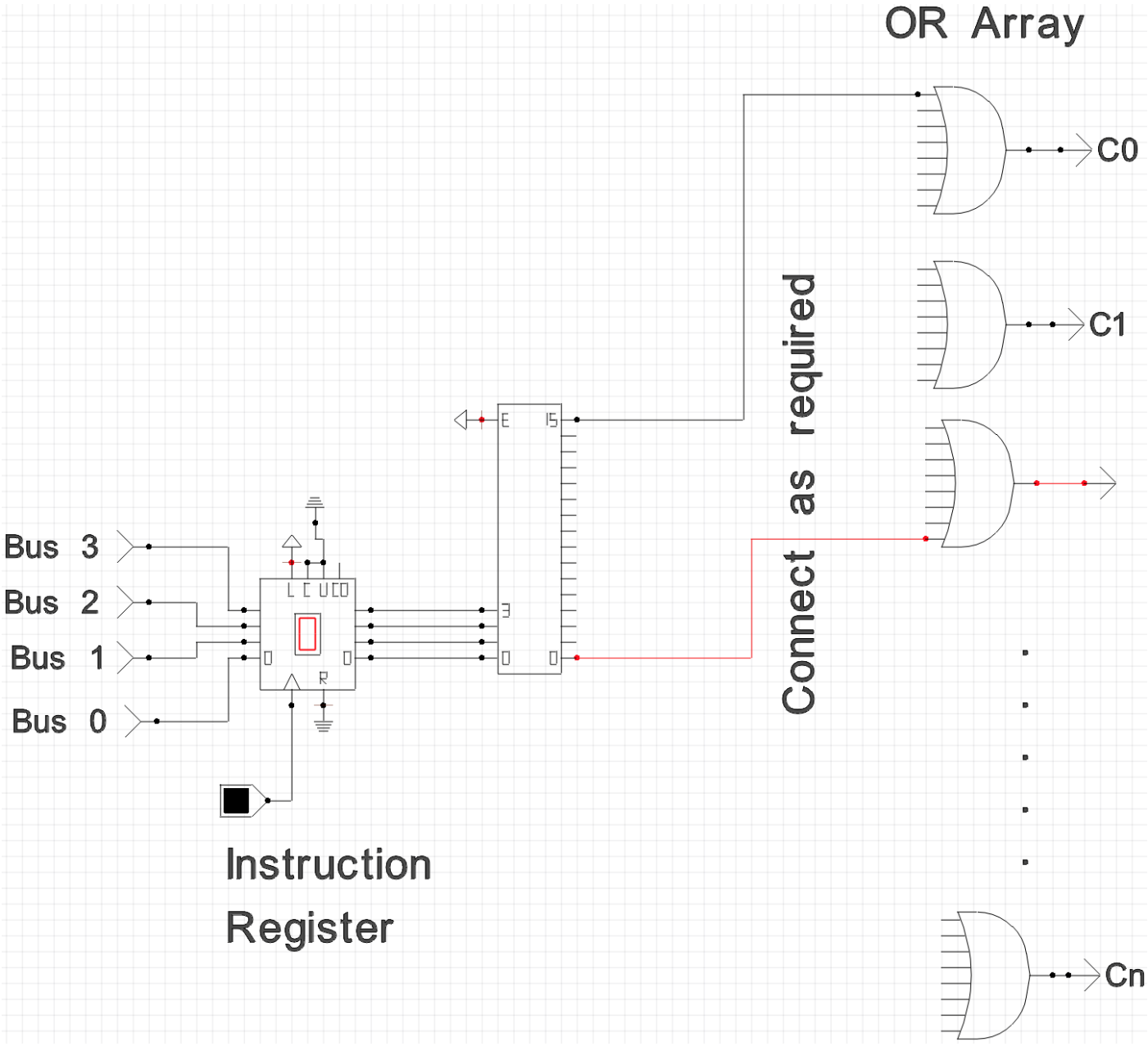
Evaluation: Programmable logic

Advantages	Disadvantages
Maximum flexibility as it can be designed so that more gates are present if they are required for a given system.	**Some problems may have a smaller solution** as particular programs may not utilise all of the combinations that this circuitry is capable of producing.
Minimal general solution as there are no smaller alternative ways in which all instructions can be decoded.	

Variant of programmable logic

In order to represent programmable logic in CEDAR logic, a variant of this architecture will be used.

Diagram: Programmable logic (variant)



In this variant, the inverters and AND gates have been replaced with a decoder. The decoder generates the same output as the inverters and AND gates in the original circuit. This is because the internal components of a decoder consists of a layer of inverters and AND gates.

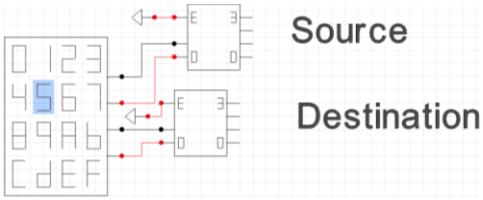
The instructions are broken down into 4 bit fields and therefore, there are only 16 instructions available.

Architectures

Bit fields

In order to simplify the program, instructions are split into fields. This also provides a solution to not having a large enough decoder.

Diagram: Instruction broken down using decoders

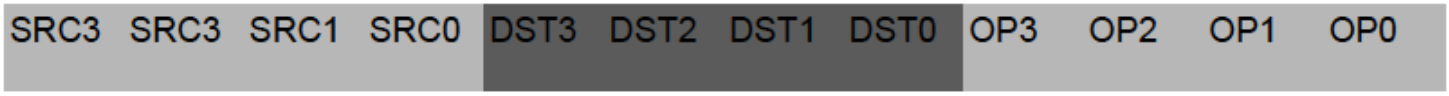


The four-bit output determines the source and destination of the data:

- the first two bits are used for the source of the data; and
- the second two bits used for the destination of the data.

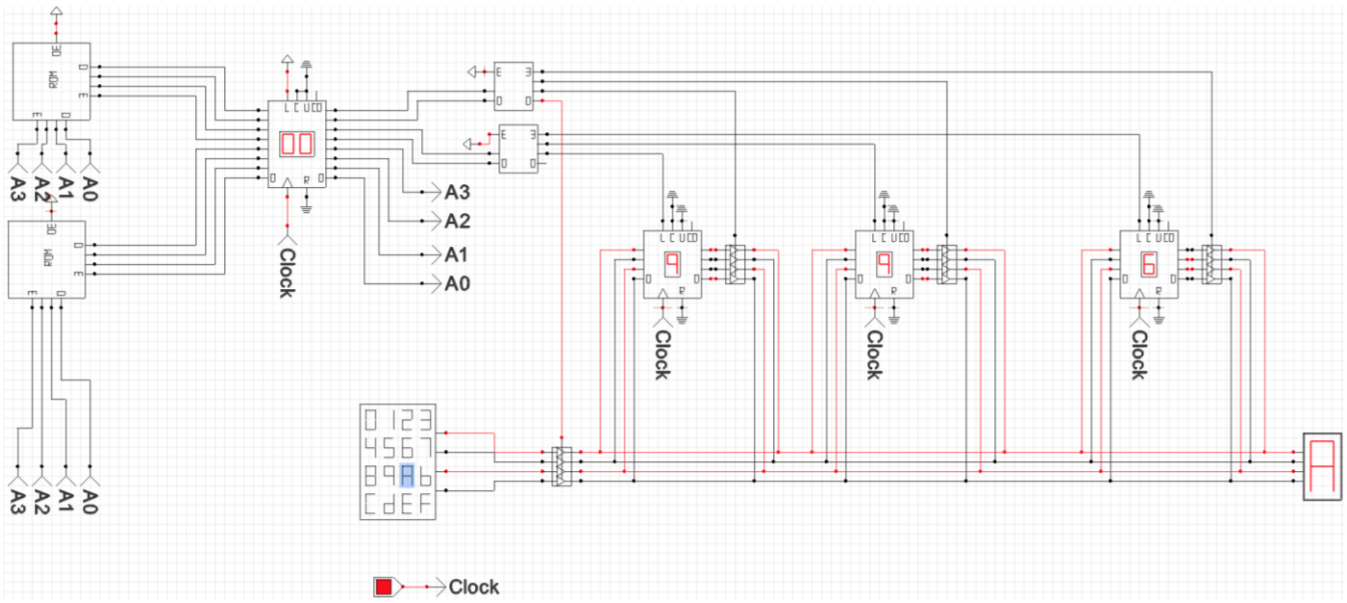
Typically, the instruction is broken down into several fields as this makes decoding much easier and creates a structure for the instruction set that makes programming more straightforward. This is because many instructions in assembly language are a combination of other instructions. By breaking down the instruction into separate parts, it is possible to create these combinations that allows fewer instructions to be necessary in assembly language. As a result, programmers only have to be aware of fewer instructions; for example, 16 instructions are present in this example however without bit fields there would be 64 instructions.

Diagram: Bit fields



Programmable logic in Harvard architecture

Diagram: Harvard architecture

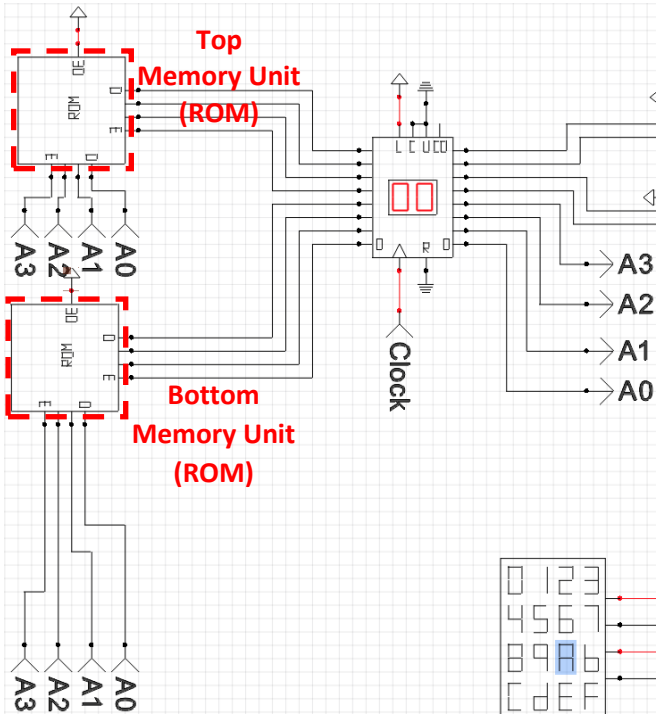


Architectures

The Harvard architecture circuit is comprised of an address bus and a data bus. These buses act independently of one another.

Interacting with the Harvard architecture

Diagram Part: Memory units (ROM)



The bottom memory unit (ROM) determines the next instruction to be fetched from the top memory unit (ROM), acting as a pointer to the next memory location.

The top memory unit (ROM) contains the instruction. This instruction is encoded using hexadecimal, in the same way as the keypad inputs.

The instructions stored in ROM are encoded using hexadecimal symbols which will be converted to a four-bit binary number.

Each bit will be put on a line of the bus.

Example: Input of 6

The hexadecimal symbol 6 corresponds a binary value.

8	4	2	1
0	1	1	0

Therefore, when the memory unit (ROM) has a value of 6, the binary digits 0110 are places on the bus lines..

Guide: Hexadecimal conversion

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

Architectures

Example: Interacting with the Harvard architecture

Insert data into the memory circuits such that:

- **on the first clock cycle, the keypad contents loads into the leftmost register;**
- **on the second clock cycle, the contents of the leftmost register are moved to the middle register;**
- **on the third clock cycle, the keypad contents loads into the leftmost register again;**
- **on the fourth clock cycle, the contents of the middle register are moved to the rightmost register; and**
- **then the sequence should then return to the first clock cycle and repeat from there.**

Bottom memory unit (ROM) contents

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0x0X	1	2	3	0	0	0	0	0	0	0	0	0	0	0	0	0

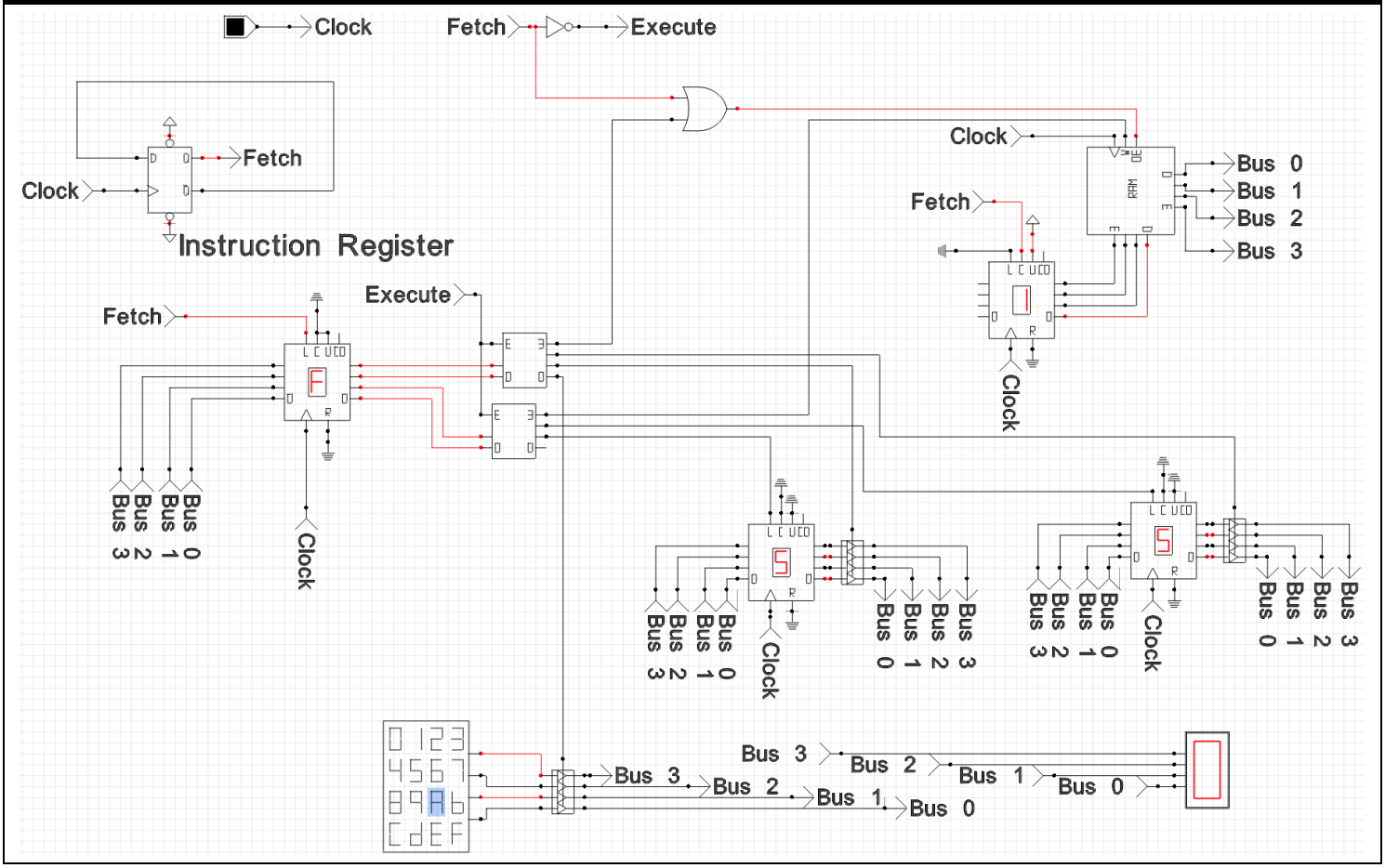
Top memory unit (ROM) contents

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0x0X	1	6	1	B	0	0	0	0	0	0	0	0	0	0	0	0

Architectures

Programmable logic in Von Neumann architecture

Diagram: Von Neumann architecture



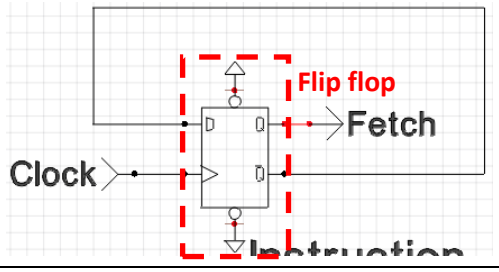
There are two completely different circuits for control.

During the $fetch$ cycle, only the circuits that are required for fetching data or instructions from memory are active. The contents of the $instruction\ register$ (IR) on the next clock cycle will determine what happens next, as such which circuits will be active. Data or instructions are loaded into the $instruction\ register$ (IR) at the end of the clock cycle.

Flip flop

The flip flop drives the fetch-execute cycle.

Diagram Part: Flip flop



On a $fetch$ cycle:

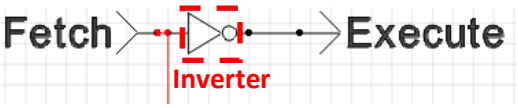
- the decoders that decode the contents of the $instruction\ register$ (IR) are disabled by the execute signal;
- the fetch signal is used to control everything.

Architectures

Inverter

The alternation between the $fetch$ and $execute$ signals is achieved by using an inverter.

Diagram Part: Flip flop



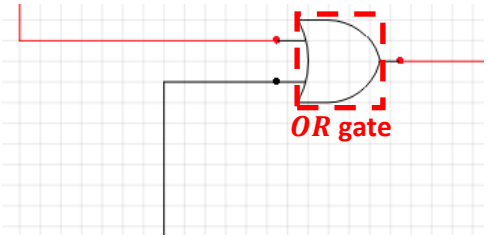
This means that:

- if the $fetch$ signal is high, the $execute$ signal is low; and
- if the $fetch$ signal is low, the $execute$ signal is high.

OR gate

The OR gate controls the output of the memory.

Diagram Part: OR gate



On a $fetch$ cycle:

- the data from the memory is transferred to the $instruction\ register\ (IR)$;
- the address for the memory is supplied by the $program\ counter\ (PC)$; and
- the contents of the $program\ counter\ (PC)$ will be loaded into the $instruction\ register\ (IR)$.

Security

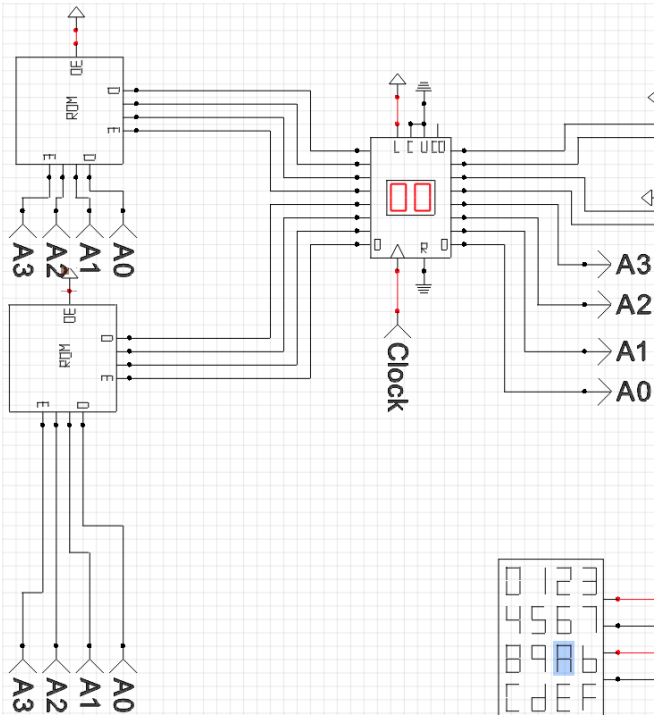
The Von Neumann architecture has security flaws as data and instructions are stored in the same place. This means that any program can write to areas in memory used by itself but also, areas in memory used by other programs. This is a security flaw as it potentially allows malicious programs to corrupt or cause intentional harm to a computer system.

As a result, there are measures in place to help prevent malicious programs from accessing the areas in memory used by the operating system. The memory protection system can be used so that when an exception is made by the operating system, it enters a higher processor state in which it accesses an area of memory that is not accessible by user programs. In addition, potential ways of corrupting the operating system by malicious users is reduced by implementing hardware changes.

Architectures

Interacting with the 8-bit Von Neumann architecture

Diagram Part: Memory units (ROM)



The memory unit (RAM) contains the instruction. This instruction is encoded using hexadecimal, in the same way as the keypad inputs.

The instructions stored in RAM are encoded using hexadecimal symbols which will be converted to a four-bit binary number.

Each bit will be put on a line of the bus.

Example: Input of 6

The hexadecimal symbol 6 corresponds a binary value.

8	4	2	1
0	1	1	0

Therefore, when the memory unit (RAM) has a value of 6, the binary digits 0110 are places on the bus lines.

Guide: Hexadecimal conversion

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

Architectures

Example: Interacting with the 8-bit Von Neumann architecture

Insert data into the memory circuit such that:

- **on the first cycle the keypad contents loads into the leftmost register.**
- **on the second cycle the contents of the leftmost register are moved to the rightmost register.**
- **then the sequence should then repeat.**

Memory unit (RAM) contents

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0x0X	1	6	1	6	1	6	1	6	1	6	1	6	1	6	1	6

The instructions in the memory unit (RAM) as currently accessed in sequence and therefore, it is necessary to repeat the instructions as there is currently no ability to "jump" to an instruction in memory.

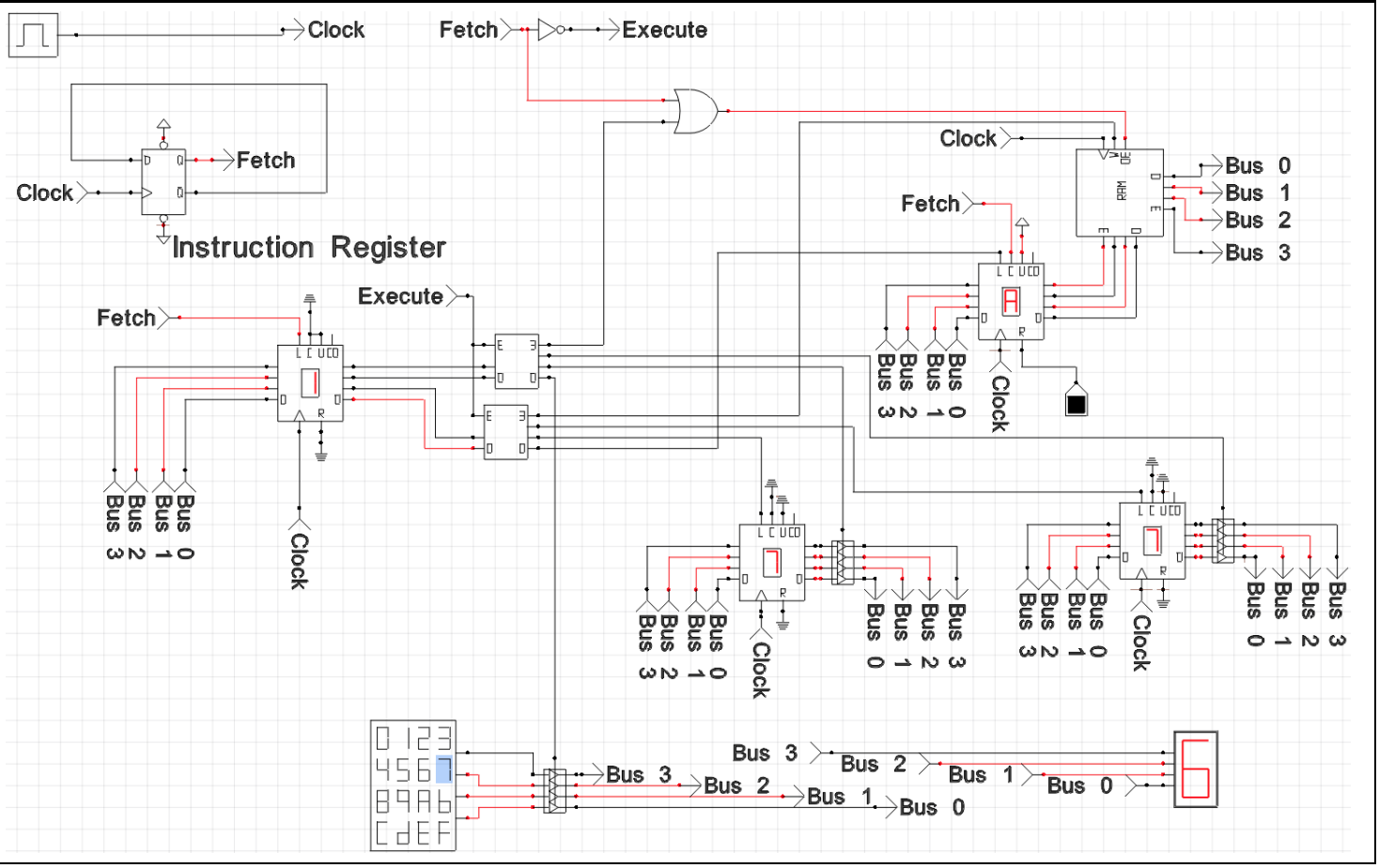
Adding "jump" capabilities to the 8-bit Von Neumann architecture

This circuit has defects as each instruction simply follows on from its predecessor. In the real world, many programs are not likely to always run in sequence and therefore the circuit requires a "jump" instruction as in the Harvard architecture.

In order to solve this issue, the program counter inputs can be connected to the bus. The program counter could then be loaded from a register or from memory.

In order to allow "jump" instructions to be processed, a connection must be added on to the reset (R) input of the $program\ counter$ (PC) so that its value can be changed reset and reloaded based on the value of another register.

Diagram: 8-bit Von Neumann architecture with "jump" capabilities



Architectures

The $program\ counter\ (PC)$ can now be loaded during the $execute$ cycle. This now changes the next instruction to be executed, a "jump" instruction.

Interacting with the 8-bit Von Neumann architecture with "jump" capabilities

The instructions no longer have to be repeated in the memory unit (RAM).

Example: Interacting with the 8-bit Von Neumann architecture with "jump" capabilities

Insert data into the memory circuit such that:

- **on the first cycle the keypad contents loads into the leftmost register.**
- **on the second cycle the contents of the leftmost register are moved to the rightmost register.**
- **then the sequence should then repeat.**

Memory unit (RAM) contents

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0x0X	1	6	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Instead the clock's transition from low to high can be used to trigger the reset (R) on the memory unit (RAM) so that its contents are removed and the "jump" instruction can be loaded into the $program\ counter\ (PC)$ from the $instruction\ register\ (IR)$.

Increasing the word length

When loading from memory, the jump destination will be the location immediately following the jump instruction. This is reasonably practical but, currently, it is not possible to use the memory for data as the only connection to the address lines is from the $program\ counter\ (PC)$, that stores the next instruction to be executed. There is no easy solution to this problem using 4-bits and therefore it is necessary to increase the word length.

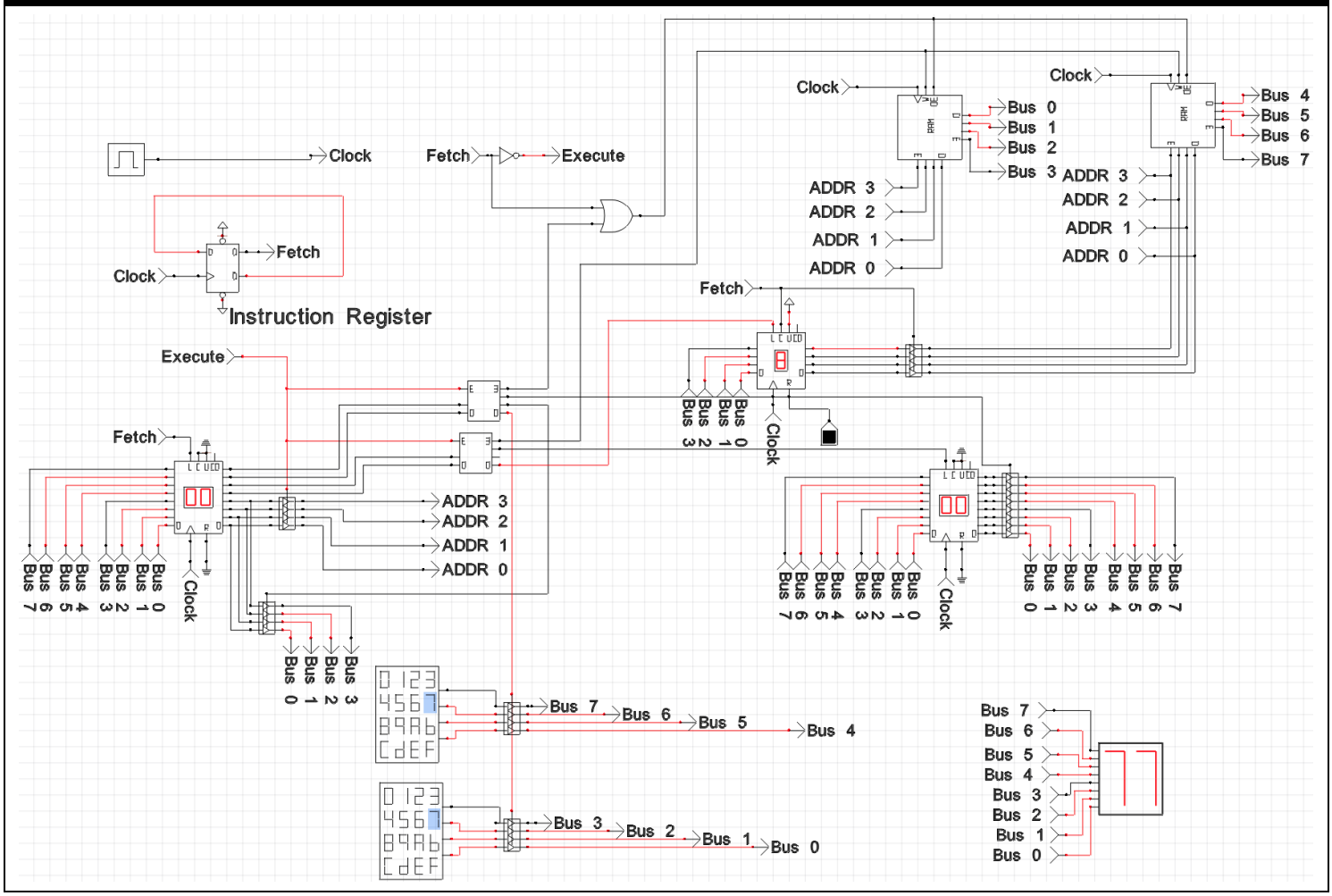
If the word length is increased to 8-bits, while maintaining a 4-bit address, then it is possible to encode an address into the instruction.

As a result, this needs to be multiplexed with the $program\ counter\ (PC)$ into the memory address. This:

- introduces the address bus; and
- provides a route for the data onto the data bus.

Architectures

Diagram: 8-bit Von Neumann architecture



It can now be decided whether to use the address in the instruction or the address from the $program\ counter\ (PC)$:

- on a $fetch$ cycle, the $program\ counter\ (PC)$ is enabled allowing the instruction from the $program\ counter\ (PC)$ to be used; and
- on an $execute$ cycle, the address coming from the $instruction\ register\ (IR)$ is enabled allowing the instruction from the $instruction\ register\ (IR)$ to be used.

This address is loaded on to the $address\ bus$, that is separate from the $data\ bus$.

In the 8-bit Von Neumann architecture:

- the top 4 bits of the instruction are the opcode, that are used to determine the type of instruction will be executed and what hardware will be used in the execution; and
- the bottom 4 bits of the instruction are address or data.

In the opcode:

- the top 2 bits specify the source; and
- the bottom 2 bits specify the destination.

Architectures

Diagram: 8-bit Von Neumann architecture instruction

SRC SRC DST DST A/D A/D A/D A/D

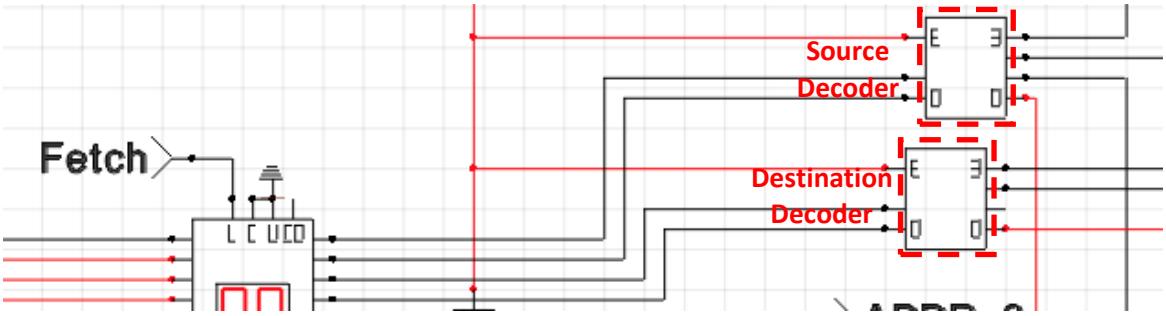
KEY	
SRC	Source
DST	Destination
A/D	Address or Data

Information: 8-bit Von Neumann architecture instruction set

Source		Destination	
00	Keypad	00	Program Counter
01	Instruction Register	01	No Operation
10	Data Register	10	Data Register
11	Memory	11	Memory

The source and destination instructions in the opcode are decoded independently by two decoders.

Diagram Part: Decoders



The 2 bits used to specify the source and the 2 bits used to specify the destination are converted into one of four options.

Assembly Language

Introduction to Assembly Language

Definition

Assembly language is a low-level language that uses mnemonics to refer to instructions for the CPU to process. It requires an assembler to convert the assembly instructions into machine code.

Why is assembly language required?

A problem arose in the late 1940's and the early 1950's where computer systems can process instructions automatically but require programs to be written using machine code (0's and 1's). This is a tedious and difficult process and therefore would prevent further development of software for computer systems.

Assembly language solves this issue by allowing programs to be represented in a descriptive format in which the purpose of the program can be identified and provides an abstraction as programmers no longer need to be concerned with the machine code implementation of a given program. In addition, translation from assembly language to machine code can be achieved easily using an assembler.

The nature of assembly language

The code in assembly language has a one-to-one relationship with the corresponding machine code, this means that each assembly language instruction directly maps to a machine code instruction. Whereas, the code in assembly language has a many-to-one relationship with the corresponding high-level language code.

Overtime, assembly language has been adapted to make it more intuitive to use however it retains its original premises of direct mapping between the assembly language instructions and the machine code instructions.

While processors differ in their instruction set, which contains the instructions available that can be decoded and executed, most real world assemblers use similar codes for similar instructions.

Assembly Language

Instructions

Moving data

Move instructions are used to transfer data from a source to a destination.

To move data, the opcode mnemonic `Move` is typically used.

Syntax: Move instruction

```
Move <src>, <dst>
```

KEY	
<src>	Source placeholder
<dst>	Destination placeholder

This can be used to move data from a keypad to a data register.

Example: Move instruction

```
Move Kpd, D
```

KEY	
Kpd	Keypad
D	Data register

Where memory is involved, the address would be specified numerically. Where the instruction register is a source then we specify the data numerically, this needs to be differentiated from an address and therefore modes of addressing are required.

Modes of addressing

Modes of addressing are used to define how machine language instructions in a given architecture identify the operand(s) of each instruction. These are defined for a given instruction set architecture.

Information: Typical modes of addressing

Mode of Addressing		Operand Identification	Notation	Example
Immediate Addressing		The operand is the actual value to be operated on.	The value is preceded with a hashtag (#) and is present in the source.	`Move #4, D` Load the data register with the value 4.
Direct Addressing	**Data from memory**	The operand holds the memory address of the value to be operated on.	The memory address is present in the source.	`Move 4, D` Load the data register with the contents of memory location 4.
	Data to memory	The operand holds the memory address of the value to be operated on.	The memory address is present in the destination.	`Move D, 4` Load the memory location 4 with the contents of the data register.

Other combinations of notation would not make sense and would therefore form an invalid instruction.

Assembly Language

Jump instructions

Jump instructions, or **branch instructions**, are used to change the sequence of instructions that are being executed.

To jump, the opcode mnemonics `jmp` or `bra` are typically used.

In many machines, the mnemonics `jmp` and `bra` coexist where:

- `jmp` is used to "jump" the long way; and
- `bra` is used to "jump" the short way.

Syntax: Jump instruction

```
Jmp <option>
```

KEY	
<option>	Placeholder

Modes of addressing are also used in jump instructions.

Modes of addressing

Information: Typical modes of addressing

Mode of Addressing		Operand Identification	Notation	Example
Indirect Addressing	Register	The operand is an intermediate location, typically a register, which holds the address of the data or instruction to be operated on.	The register is present as the option.	`Jmp D` Jump to the location addressed by the contents of the data register.
	Memory	The operand is an intermediate location, typically a register, which holds the address of the data or instruction to be operated on.	The memory location is present as the option enclosed by curly brackets.	`Jmp (4)` Jump to the location addressed by the contents of the memory location 4.
Direct Addressing		The operand holds the memory address of the value to be operated on.	The memory location is present as the option.	`Jmp 4` Jump to memory location 4.

Transforming Data

Introduction to Transforming Data

Why is transforming data necessary?

It is necessary to transform data from one format to another to allow instructions, such as arithmetic options like ADD, to be carried out on the data.

Data representation in a computer system

If there are some binary bits inside a register or a memory location, they could be a:

- number;
- instruction; or
- character.

If the binary bits are a number, they could be encoded in several different ways.

The representation of a binary number is distinguished by:

- the “significance” of each bit; and
- the method used for encoding negative numbers or the methods used for floating point numbers that allow the “significance” of each bit to be adjusted dynamically.

Transforming Data

Unsigned Binary Numbers

Representation

The representation of an unsigned binary number is distinguished by using the binary place value system in which:

- each bit represents a power of 2;
- each power is a positive integer;
- the rightmost bit will be the lowest power; and
- there is an integer format which follows $2^7 2^6 2^5 2^4 2^3 2^2 2^1 2^0$ (8 bit).

Information: Binary place values for an 8-bit binary number

Position	8	7	6	5	4	3	2	1
Weight *(base 2)*	128	64	32	16	8	4	2	1

The minimum decimal value for an unsigned binary number constructed using n bits is 0.
The minimum non-zero decimal value for an unsigned binary number constructed using n bits is 1.
The maximum decimal value for an unsigned binary number constructed using n bits is calculated by $2^n - 1$.

An 8-bit unsigned binary number has:

- a minimum decimal value of 0, when the binary value is 00000000;
- a minimum non-zero decimal value of 1, when the binary value is 00000001; and
- a maximum decimal value of 255, when the binary value is 11111111.

Converting binary to decimal

Example: Determining binary numbers

Determine the decimal number represented below.

Position	8	7	6	5	4	3	2	1
Weight *(base 2)*	128	64	32	16	8	4	2	1
Binary Number	0	1	1	0	1	1	0	1

$$value = \sum_{n=0}^{\infty} (weight_n \times digit_n)$$

$$value = (128 \times 0) + (64 \times 1) + (32 \times 1) + (16 \times 0) + (8 \times 1) + (4 \times 1) + (2 \times 0) + (1 \times 1)$$

$$value = 0 + 64 + 32 + 0 + 8 + 4 + 0 + 1$$

$$value = 64 + 32 + 8 + 4 + 1$$

$$\boldsymbol{value = 109}$$

Transforming Data

Converting decimal to binary

Division method

Process: Division method

1) Repeat until division evaluates to 0:
   - divide the decimal number by 2 (as binary is base 2); and
   - if there is a fraction left over, such as 0.5, set this as the remainder r.
2) Construct the binary number using the remainder r values in reverse order.

Example: Division method

Convert 155_{10} to binary.

$\frac{155_{10}}{2_{10}} = 77_{10}\ (r = 1)$
$\frac{77_{10}}{2_{10}} = 38_{10}\ (r = 1)$
$\frac{38_{10}}{2_{10}} = 19_{10}\ (r = 0)$
$\frac{19_{10}}{2_{10}} = 9_{10}\ (r = 1)$
$\frac{9_{10}}{2_{10}} = 4_{10}\ (r = 1)$
$\frac{4_{10}}{2_{10}} = 2_{10}\ (r = 0)$
$\frac{2_{10}}{2_{10}} = 1_{10}\ (r = 0)$
$\frac{1_{10}}{2_{10}} = 0_{10}\ (r = 1)$

Repeat until division evaluates to 0:
- divide the decimal number by 2 (as binary is base 2); and
- if there is a fraction left over, such as 0.5, set this as the remainder r.

$\mathbf{10011011_2}$

Construct the binary number using the remainder r values in reverse order.

Subtraction method

Process: Subtraction method

1) Start at the most significant bit (MSB) and repeat until the least significant bit (LSB) is reached:
   - subtract the place value from the decimal number;
   - if the result is positive, place a one (1) under that place value and perform the subtraction operation; and
   - if the result is negative, place a zero (0) under that place value and do not perform the subtraction operation.
2) Write the answer.

Transforming Data

Example: Subtraction method

Convert 155_{10} to binary.

128	64	32	16	8	4	2	1

$155 - 128 = 27$

128	64	32	16	8	4	2	1
1							

$27 - 64 = -37$

128	64	32	16	8	4	2	1
1	0						

$27 - 32 = -5$

128	64	32	16	8	4	2	1
1	0	0					

$27 - 16 = 11$

128	64	32	16	8	4	2	1
1	0	0	1				

$11 - 8 = 3$

128	64	32	16	8	4	2	1
1	0	0	1	1			

$3 - 4 = -1$

128	64	32	16	8	4	2	1
1	0	0	1	1	0		

$3 - 2 = 1$

128	64	32	16	8	4	2	1
1	0	0	1	1	0	1	

$1 - 1 = 0$

128	64	32	16	8	4	2	1
1	0	0	1	1	0	1	1

$\mathbf{10011011_2}$

Start at the most significant bit (MSB) and repeat until the least significant bit (LSB) is reached:

- subtract the place value from the decimal number;
- if the result is positive, place a one (1) under that place value and perform the subtraction operation; and
- if the result is negative, place a zero (0) under that place value and do not perform the subtraction operation.

Write the answer.

Transforming Data

Unsigned Fractional Binary Numbers

Representation

The representation of an unsigned fractional binary number can also be distinguished by using the binary place value system in which:

- each bit represents a power of 2;
- each power is a negative integer;
- the rightmost bit will be the lowest power; and
- there is an integer format which follows $2^{-1}2^{-2}2^{-3}2^{-4}2^{-5}2^{-6}2^{-7}2^{-8}$ (8 bit).

Information: Binary place values for an 8-bit binary number

Position	8	7	6	5	4	3	2	1
Weight *(base 2)*	$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{8}$	$\frac{1}{16}$	$\frac{1}{32}$	$\frac{1}{64}$	$\frac{1}{128}$	$\frac{1}{256}$

The minimum decimal value for an unsigned fractional binary number constructed using n bits is 0.
The minimum non-zero decimal value for an unsigned fractional binary number constructed using n bits is calculated by $\frac{1}{2^n}$.
The maximum decimal value for an unsigned fractional binary number constructed using n bits is calculated by $1 - \frac{1}{2^n}$.

An 8-bit unsigned fractional binary number has:

- a minimum decimal value of 0, when the binary value is 00000000;
- a minimum non-zero decimal value of $\frac{1}{256}$, when the binary value is 00000001; and
- a maximum decimal value of $1 - \frac{1}{256}$, when the binary value is 11111111.

This format is useful if the numbers that are to be represented will always be between 0 and 1.

Fixed point binary numbers

Adding a “binary point” in the word allows an unsigned fractional number to be expressed as a fixed point binary number.

Converting decimal to fixed point binary

Process: Decimal to fixed point binary conversion

1) Convert the denary number to binary.
2) If the number is negative, convert it using Two’s Complement.
3) Set the place values appropriately according to the rules:
    - the integer place values must be 2^n;
    - the decimal place values must be 2^{-n}.
4) Place a decimal point between the integer place values and the decimal place values.
5) Write the correct binary digits for each place value to create the fixed point binary number.

Transforming Data

Example: Decimal to fixed point binary conversion

Convert the real number -5.5 into its fixed point binary equivalent.

	-8	4	2	1	1/2	1/4	1/8	1/16
5.5	0	1	0	1	1	0	0	0

Convert the denary number to binary.

	-8	4	2	1	1/2	1/4	1/8	1/16
5.5	0	1	0	1	1	0	0	0
Flip Bits	1	0	1	0	0	1	1	1
Add one	1	0	1	0	1	0	0	0
CARRY					1	1	1	1

Convert 5.5 to -5.5 using Two's Complement.

1010.1000

Write the answer.

Converting fixed point binary to decimal

Process: Fixed point binary to decimal conversion

1) Place a decimal point between the integer place values and the decimal place values.
2) Convert the binary number to denary.

Example: Fixed point binary to decimal conversion

Convert the unsigned fixed point binary number 1010.11 into its denary equivalent.

	8	4	2	1	1/2	1/4	1/8	1/16
5.5	1	0	1	0	1	1	0	0

Place a decimal point between the integer place values and the decimal place values.

$8 \times 1 \quad = 8$
$4 \times 0 \quad = 0$
$2 \times 1 \quad = 2$
$1 \times 0 \quad = 0$
$\frac{1}{2} \times 1 \quad = 0.5$
$\frac{1}{4} \times 1 \quad = 0.25$
$\frac{1}{8} \times 0 \quad = 0$
$\frac{1}{16} \times 0 \quad = 0$
$8 + 0 + 2 + 0 + 0.5 + 0.25 + 0 + 0 = 10.75$

Convert the binary number into denary.

10.75

Write the answer.

Transforming Data

Sign and Magnitude

Definition

Sign and magnitude is a way of representing binary numbers where the most significant bit (MSB) stores the sign: 0 is positive; and 1 is negative.

Converting from decimal to sign and magnitude

Example: Decimal to sign and magnitude conversion

Convert 88 to a sign and magnitude binary number.

MSB	64	32	16	8	4	2	1
0	1	0	1	1	0	0	0

$64 \times 1 = 64$
$32 \times 0 = 0$
$16 \times 1 = 16$
$8 \times 1 = 8$
$4 \times 0 = 0$
$2 \times 0 = 0$
$1 \times 0 = 0$
$64 + 16 + 8 = 88$

MSB is zero, so the number is positive.
Therefore, the number being represented is 88.

Example: Decimal to sign and magnitude conversion

Convert -88 to a sign and magnitude binary number.

MSB	64	32	16	8	4	2	1
1	1	0	1	1	0	0	0

$64 \times 1 = 64$
$32 \times 0 = 0$
$16 \times 1 = 16$
$8 \times 1 = 8$
$4 \times 0 = 0$
$2 \times 0 = 0$
$1 \times 0 = 0$
$64 + 16 + 8 = 88$

MSB is one, so the number is negative.
Therefore, the number being represented is -88.

Transforming Data

Converting from sign and magnitude to decimal

Example: Sign and magnitude to decimal conversion

Convert the sign and magnitude binary number 00000011 into denary.

MSB	64	32	16	8	4	2	1
0	0	0	0	0	0	1	1

$64 \times 0 = 0$
$32 \times 0 = 0$
$16 \times 0 = 0$
$8 \times 1 = 0$
$4 \times 0 = 0$
$2 \times 1 = 2$
$1 \times 1 = 1$
$2 + 1 = 3$

MSB is zero, so the number is positive.
Therefore, the number being represented is 3.

Example: Sign and magnitude to decimal conversion

Convert the sign and magnitude binary number 10000011 into denary.

MSB	64	32	16	8	4	2	1
1	0	0	0	0	0	1	1

$64 \times 0 = 0$
$32 \times 0 = 0$
$16 \times 0 = 0$
$8 \times 1 = 0$
$4 \times 0 = 0$
$2 \times 1 = 2$
$1 \times 1 = 1$
$2 + 1 = 3$

MSB is one, so the number is negative.
Therefore, the number being represented is -3.

Issues

Representation of binary numbers using sign and magnitude yields an issue where there are two possible ways to represent zero (0):

- $00000000 = +0$; and
- $10000000 = -1$.

This is an issue because binary numbers should only have one representation so that their interpretation is uniform across varied processor architectures.

Transforming Data

Binary Addition

Rules of binary addition

The rules of binary addition define the sum and carry of a binary addition operation.

Information: Rules of binary addition

Operation		Result	Sum (S)	Carry (C)
$0+0$	$=$	0	0	NO
$0+1$	$=$	1	1	NO
$1+0$	$=$	1	1	NO
$1+1$	$=$	10	1	NO
$1+1+1$	$=$	11	0	YES

A carry bit is present where the result of the operation cannot be represented using one bit.

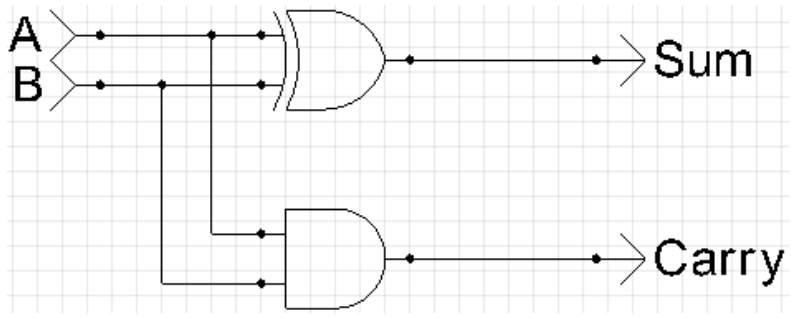
Half adder

Definition

A **half adder** is a circuit that can take an input of two bits (input A and input B), perform addition on the two input bits and give a two-bit output (sum S and a carry bit C).

Circuitry

Diagram: Half adder



TRUTH TABLE			
A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Only one exclusive OR (XOR) gate is used as implementations that use multiple exclusive OR (XOR) gates are less efficient as there is a longer path for the data to travel along.

A half adder cannot use a carry bit from a previous addition operation and therefore is unable to perform n-bit number additions.

Transforming Data

Expressions

The half adder circuit can be evaluated to give expressions in terms of S and C.

- $S = A \oplus B$
- $C = A \,.\, B$

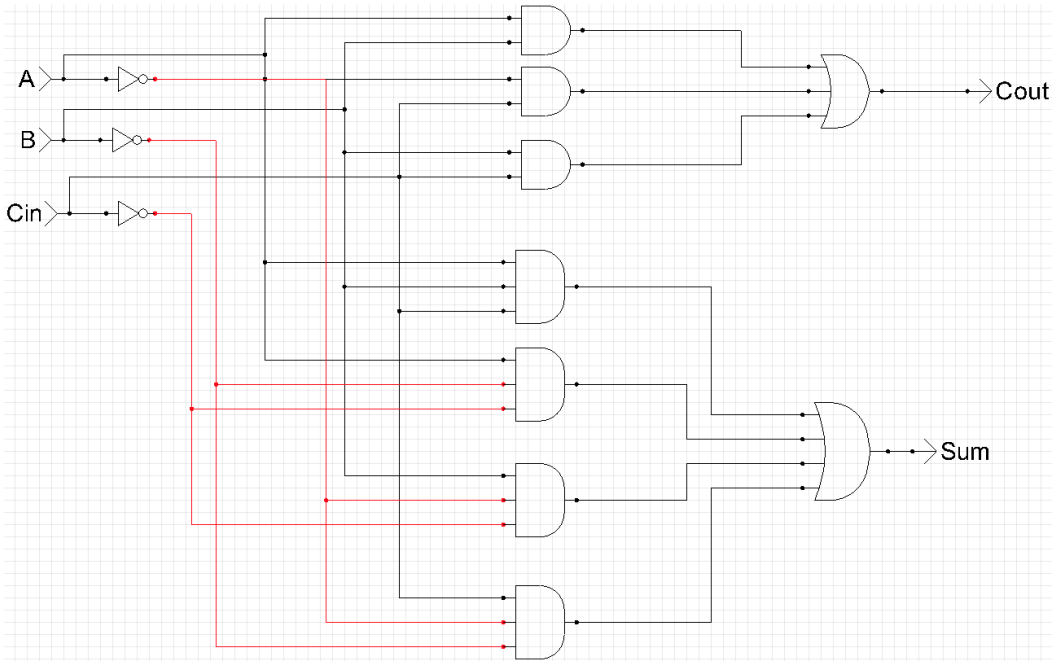
Full adder

Definition

A **full adder** is a circuit which can take an input of three bits (input A, input B and a carry bit C_{IN}), perform addition on the three bits and give a two-bit output (sum S and a carry bit C_{OUT}).

Circuitry

Diagram: Full adder (using "general logic")



TRUTH TABLE				
A	B	C_{IN}	S	C_{OUT}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

The sum S is $True$ if one or all of the inputs are $True$.

The carry bit C_{OUT} is $True$ if any two of the inputs are $True$. As a result, there are three AND gates that are connected:

- $A \,.\, B$;
- $B \,.\, C_{IN}$; and
- $A \,.\, C_{IN}$

such that if any of these AND gates evaluate to $True$ then C_{OUT} will also be $True$ as the result of the 3-input OR gate will be $True$.

Transforming Data

Expressions

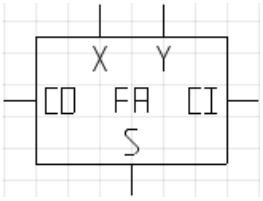
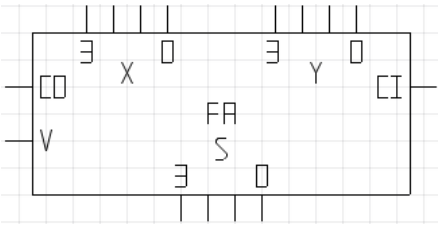
The full adder circuit can be evaluated to give expressions in terms of S and C_{OUT}.

- $S = A \oplus B \oplus C_{IN}$
- $C = (A \,.\, B) + (C_{IN} \,.\, (A \oplus B))$

Encapsulated components

CEDAR Logic provides a 1-bit full adder component and a 4-bit full adder component.

Information: Encapsulated components

Component	Diagram	Input / Output	
1-bit Full Adder		**Symbol**	**Meaning**
		X	First value
		Y	Second value
		CI	Carry in
		CO	Carry out
		S	Sum
4-bit Full Adder		**Symbol**	**Meaning**
		X	First value
		Y	Second value
		CI	Carry in
		CO	Carry out
		S	Sum
		V	Overflow output

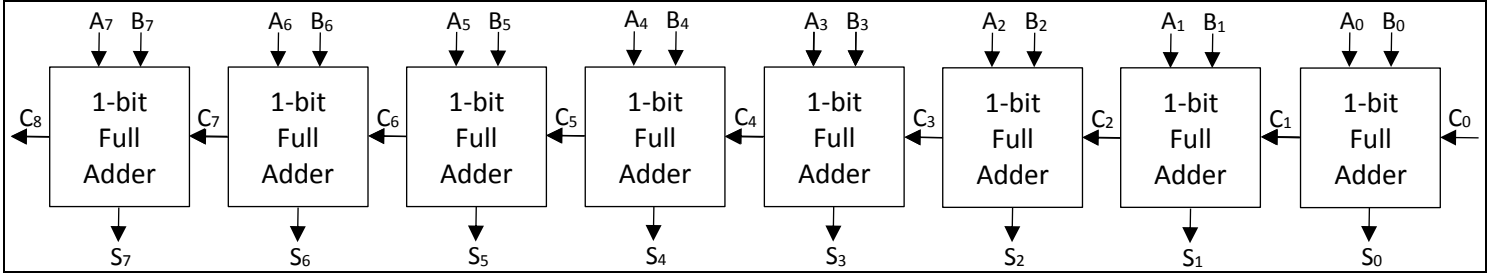
Transforming Data

Concatenating Full Adders

Abstraction

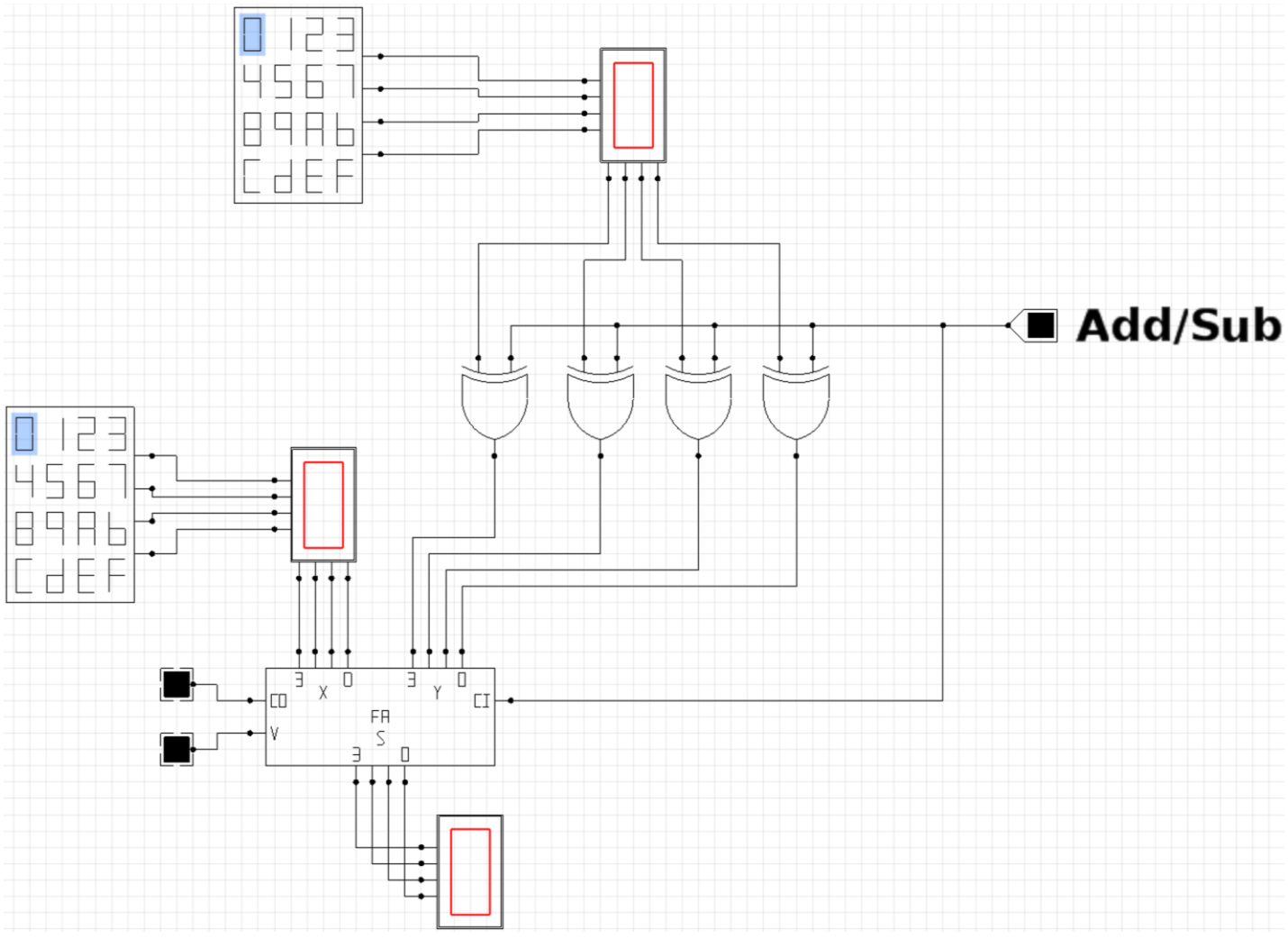
A full adder can use a carry bit (C) from a previous addition operation and therefore is able to perform n-bit number additions. In order to do this, multiple full adders must be concatenated together by connecting the carry out (C_{OUT}) from one full adder circuit to the carry in (C_{IN}) on a subsequent full adder circuit.

Diagram: 8-bit full adder abstraction



4-bit full adder

Diagram: 4-bit full adder



Transforming Data

Add/Sub can be used to set the circuit to perform addition or subtraction:

- if Add/Sub is low, addition is to be performed; and
- if Add/Sub is $high$, subtraction is to be performed.

This is possible as when Add/Sub is $high$, the input bits are inverted and the carry in (C_{IN}) is set to $high$ – this converts the number to Two's complement. XOR gates are used to perform the invert as the truth table for the XOR gate has the correct characteristics for this purpose.

Truth Table: XOR gate

A	B	O
0	0	0
0	1	1
1	0	1
1	1	0

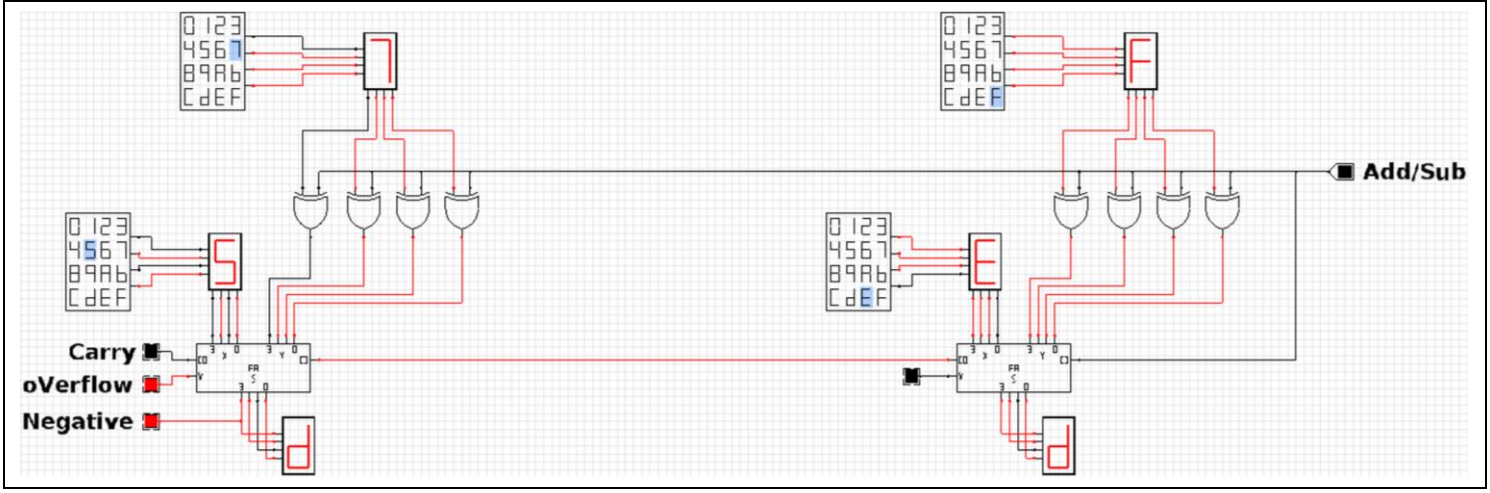
Each bit from the number acts as an input for input $\boldsymbol{A}$ and Add/Sub acts as input $\boldsymbol{B}$:

- if the bit from the number is 0 and Add/Sub is 0, then addition is being performed and therefore the output bit should be 0;
- if the bit from the number is 0 and Add/Sub is 1, then subtraction is being performed and therefore the output bit should be 1;
- if the bit from the number is 1 and Add/Sub is 0, then addition is being performed and therefore the output bit should be 1; and
- if the bit from the number is 1 and Add/Sub is 1, then subtraction is being performed and therefore the output bit should be 0.

8-bit full adder

The 8-bit full adder consists of two 4-bit full adders concatenated together.

Diagram: 8-bit full adder



The carry out (C_{OUT}) from the right 4-bit full adder is connected to the carry in (C_{IN}) on the left 4-bit full-adder.

Arithmetic can be performed where the bottom number is to be added to or subtracted from the top number.

Transforming Data

Test Data: 8-bit full adder								
A	**Op**	**B**	**Result**	**Carry**	**Negative**	**oVerflow**	**Zero**	**Decimal Result**
43	+	27	6A	NO	NO	NO	NO	70
43	-	27	1C	NO	YES	NO	NO	28
43	+	D9	1C	YES	NO	NO	NO	284
27	+	D9	00	YES	NO	NO	YES	256
5E	+	7F	DD	NO	YES	YES	NO	221
5E	-	7F	DF	NO	YES	NO	NO	-33
5E	+	80	DE	NO	YES	NO	NO	-222

$27 + D9$ is not possible as there is a carry out (C_{OUT}) meaning that a full adder larger than 8 bits is required to perform this operation.

$5E + 80$ is not possible as 222 is outside of the range that 8-bit hexadecimal is able to represent as 8-bit hexadecimal can only represent values up to 128.

Issues

However, there is a delay present and with each full adder circuit added to the concatenation, the delay will become greater. The original “general logic” system could be used to generate an adder of any length by using three layers of gates and therefore only have three layers of delay. However, there is technical difficult with this approach as some gates may become very wide, as they would have many inputs, and therefore would be slower. This approach is also physically difficult to construct. Instead, to avoid having a slow circuit, look ahead carry is used.

Transforming Data

Lookahead Carry

Definition

A **carry-lookahead adder (CLA)**, or **fast adder**, is a type of adder that improves speed by reducing the amount of time required to determine carry bits.

How it works

This works using the logic that a carry out (C_{OUT}) will only be generated if $C = (A \,.\, B) + (C_{IN} \,.\, (A \oplus B))$ evaluates to $True$.

This can be demonstrated by looking at the binary addition rules.

Information: Rules of binary addition

Operation		Result	Sum (S)	Carry (C)
$0 + 0$	=	0	0	NO
$0 + 1$	=	1	1	NO
$1 + 0$	=	1	1	NO
$1 + 1$	=	10	1	YES
$1 + 1 + 1$	=	11	0	YES

This shows that a carry bit (C) is only present if three 1's are present in the operation. This information can be used to identify where a full adder will generate a carry out (C_{OUT}).

Truth Table: Full adder

A	B	C_{IN}	S	C_{OUT}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

The truth table for the full adder shows that the full adder circuit will only generate a carry out (C_{OUT}) if:

- $A = 0, B = 1$ and $C_{IN} = 1$;
- $A = 1, B = 0$ and $C_{IN} = 1$;
- $A = 1, B = 1$ and $C_{IN} = 0$; or
- $A = 1, B = 1$ and $C_{IN} = 1$.

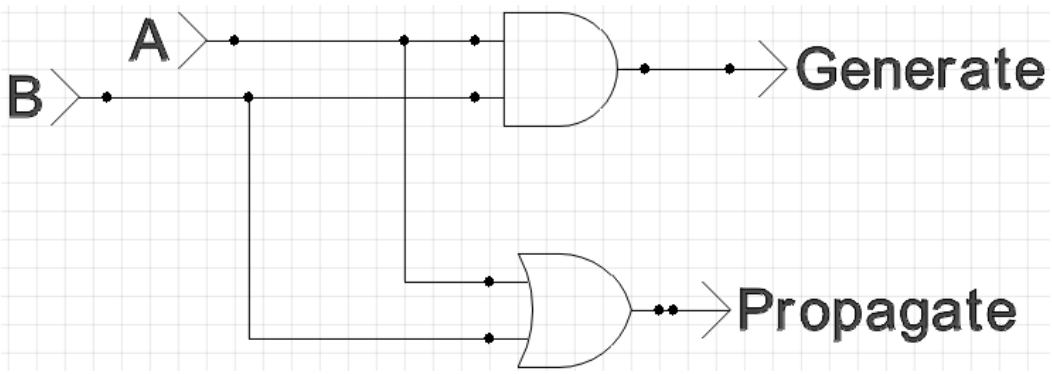
From this, it can be deduced that:

- if both inputs (A and B) are 0, there will never be a carry out (C_{OUT});
- if either input (A or B) is 1, there will only be a carry out (C_{OUT}) if there was a carry in (C_{IN}); and
- if both inputs (A and B) are 1, there will always be a carry out (C_{OUT}).

This logic allows the lookahead carry circuit to "cheaply" generate a signal for the word. At each bit position in the adder circuit, a carry generate and a carry propagate is created.

Transforming Data

Diagram: Lookahead carry



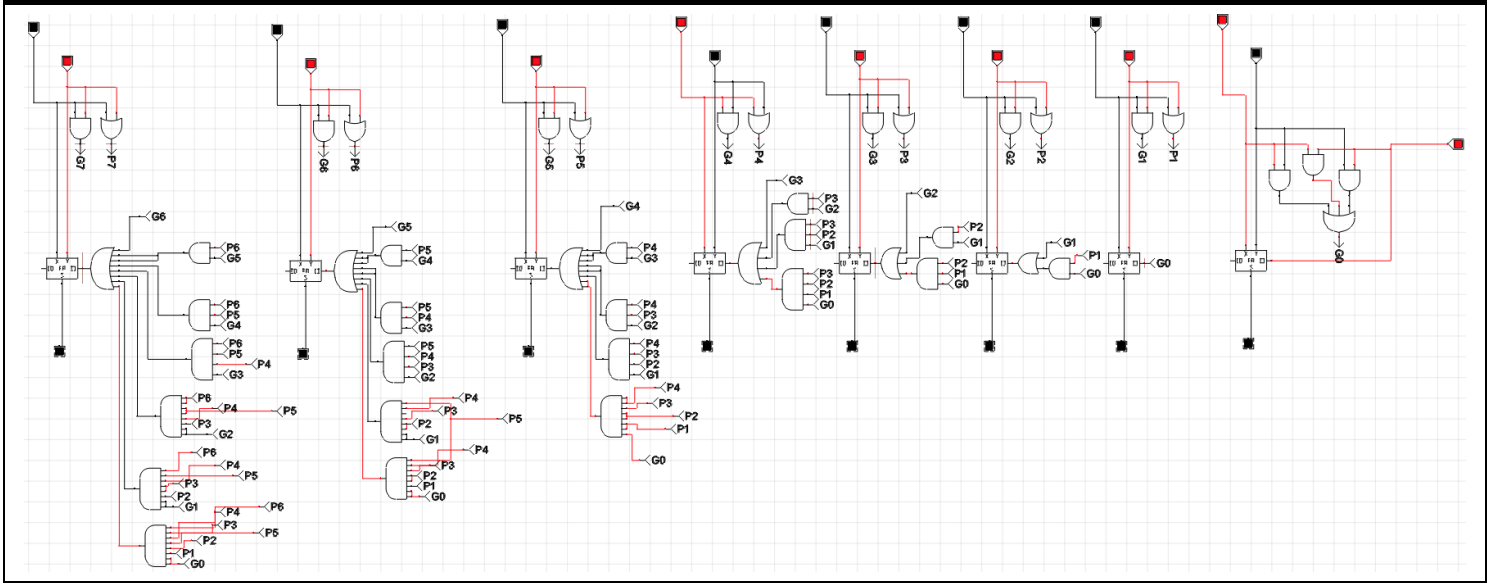
A $generate$ signal is generated if it is certain that a carry out (C_{OUT}) will be generated. This occurs when both inputs (A and B) are 0 or both inputs (A and B) are 1.

A $propagate$ signal is generated if it cannot be certain whether a carry out (C_{OUT}) will be generated. This occurs when either input (A or B) is 1.

Concatenating adders with lookahead carry

When concatenating 1-bit adders with lookahead carry implemented, each 1-bit adder will have a circuit that is capable of creating a carry generate and a carry propagate.

Diagram: 8-bit carry-lookahead adder (CLA)



Moderately sized adders can use these signals to produce all the required carry bits. While, longer adders can adapt a hierarchical approach.

Transforming Data

Binary Subtraction

How subtraction works

A circuit can perform binary subtraction by adding a negative binary number. This works as adding a negative number yields the same result as subtracting the positive equivalent of the same number:

$$3 - 2 \equiv 3 + -2.$$

Process: Binary subtraction using arithmetic

1) Convert the denary number to binary.
2) For each place value, starting at the least significant bit (LSB), subtract the second binary number from the first binary number using the binary addition rules:
   - $0 - 0 = 0$;
   - $1 - 0 = 1$; and
   - $1 - 1 = 0$.
2) If there is an subtraction operation where $0 - 1$ occurs, borrow two from the column to the left:
   - cross out the one (1) in the column to the left;
   - place two ones (1s) above the current column; and
   - perform the subtraction operation.
3) If there is an subtraction operation where $0 - 1$ occurs, and there is a zero in the column to the left:
   - continue to the left until there is a column with a one (1);
   - cross out the one (1) in that column;
   - place two ones (1s) above the column to the right:
     - if the column to the right is the current column, perform the subtraction operation;
     - if the column to the right is not the current column, cross out one of the borrowed ones (1s) and place two ones (1s) above the column to the right – continue this process until the current column is reached and perform the subtraction operation.
4) Write the answer.

Example: Binary subtraction using arithmetic

Perform $16 - 4$ in binary.

	128	64	32	16	8	4	2	1
16	0	0	0	1	0	0	0	0
4	0	0	0	0	0	1	0	0

Convert the denary number to binary.

	128	64	32	16	8	4	2	1
16	0	0	0	1	0	0	0	0
4	0	0	0	0	0	1	0	0
SUM							0	0

Perform the subtraction operation.

Transforming Data

BORROW					~~1~~ 1	1 1		
	128	**64**	**32**	**16**	**8**	**4**	**2**	**1**
16	0	0	0	~~1~~	0	0	0	0
4	0	0	0	0	0	1	0	0
SUM	0	0	0	0	1	1	0	0

If there is a subtraction operation where
$0 - 1$ occurs, and there is a zero in the column to the left:

- continue to the left until there is a column with a one (1);
- cross out the one (1) in that column;
- place two ones (1s) above the column to the right:
  - if the column to the right is the current column, perform the subtraction operation;
  - if the column to the right is not the current column, cross out one of the borrowed ones (1s) and place two ones (1s) above the column to the – continue this process until the current column is reached and perform the subtraction operation.

00001100

Write the answer.

However, on approximately 50% of binary subtraction operations, there will be a carry from the top of the addition.

Information: Adding 3-bit binary numbers

A / B	**000**	**001**	**010**	**011**	**100**	**101**	**110**	**111**
000	000	001	010	011	100	101	110	111
001	001	010	011	100	101	110	111	000
010	010	011	100	101	110	111	000	001
011	011	100	101	110	111	000	001	010
100	100	101	110	111	000	001	010	011
101	101	110	111	000	001	010	011	100
110	110	111	000	001	010	011	100	101
111	111	000	001	010	011	100	101	110

Transforming Data

The binary numbers coloured in red are the incorrect sum for each given $A - B$ operation. This is because there is a carry bit present at the most significant bit (MSB). This bit cannot be represented as there is a finite space available in the registers and this bit cannot be stored and is discarded, therefore an overflow error has occurred as, for example:

$$110 + 010 \neq 000 \; (3 + 2 \neq 0).$$

In order to determine a new method of dealing with these additions, we can inspect the results of some of the sums.

Information: Results of sample sums

N	$-N$
001	111
010	110
011	101

In order to convert N to $-N$, the following rules are observed:

1) invert all bits in the binary number; then
2) add one to the binary number.

Therefore, it can be identified that:

- any number starting with a 0 is a positive number; and
- any number starting with a 1 is a negative number.

This method of representing binary numbers is known as Two's complement.

Two's complement

Definition

Two's complement is a way of representing binary numbers where the most significant bit (MSB) is always negative.

Representation

The representation of a Two's complement binary number can also be distinguished by using the binary place value system in which:

- each bit represents a power of 2;
- the most significant bit (MSB) has a negative integer power while all other powers are positive integers;
- the rightmost bit will be the lowest power; and
- there is an integer format which follows $-(2^7)2^6 2^5 2^4 2^3 2^2 2^1 2^0$ (8 bit).

Information: Binary place values for an 8-bit Two's complement binary number

Position	8	7	6	5	4	3	2	1
Weight *(base 2)*	-128	64	32	16	8	4	2	1

The minimum positive decimal value for a Two's complement binary number constructed using n bits is 0.
The minimum non-zero decimal value for a Two's complement binary number constructed using n bits is calculated by 1.
The maximum decimal value for a Two's complement binary number constructed using n bits is calculated by $2^{n-1} - 1$.

Transforming Data

An 8-bit Two's complement binary number has:

- a minimum positive decimal value of 0, when the binary value is 00000000;
- a minimum non-zero decimal value of -128, when the binary value is 10000000; and
- a maximum decimal value of 127, when the binary value is 01111111.

The minimum non-zero decimal value and the maximum decimal value are not symmetrical as there has to be a combination available for zero (0). This removes one option of a combination of binary bits from the positive values but not rather than the negative values as all negative numbers are non-zero.

Converting decimal to Two's complement

Process: Converting decimal to Two's complement

1) Convert the decimal number to binary.
2) Convert to One's Complement by flipping the bits.
3) Add one.

Example: Converting decimal to Two's complement

Convert -57 to a two's complement binary number.

-128	64	32	16	8	4	2	1
0	0	1	1	1	0	0	1

Convert the decimal number to binary.

-128	64	32	16	8	4	2	1
1	1	0	0	0	1	1	0

Convert to One's Complement by flipping the bits.

-128	64	32	16	8	4	2	1
1	1	0	0	0	1	1	1

Add one.

Binary subtraction using Two's complement

Process: Binary subtraction using Two's complement

1) Convert both decimal numbers to binary.
2) Convert the second binary number into a negative binary number using Two's Complement.
3) Add the first binary number and the converted second binary number together using the binary addition rules:
   - $0 + 0 = 0$;
   - $1 + 0 = 1$;
   - $1 + 1 = 10$; and
   - $1 + 1 + 1 = 11$.
4) If the expression evaluates to an answer larger than one bit:
   - the first of the two bits is a carry bit;
   - the carry bit is moved to the next column; and
   - the addition in the next column must incorporate the carry bit.
5) If there is a carry bit present after performing addition on the most significant bit (MSB), there is an overflow error because there are not enough bits to represent all of the binary digits.
6) Write the answer.

Transforming Data

Example: Binary subtraction using Two's complement

Perform $85 - 16$ in binary.

	-128	64	32	16	8	4	2	1
85	0	1	0	1	0	1	0	1
16	0	0	0	1	0	0	0	0

Convert both decimal numbers to binary.

	-128	64	32	16	8	4	2	1
16	0	0	0	1	0	0	0	0
Flip Bits	1	1	1	0	1	1	1	1
Add one	1	1	1	1	0	0	0	0
CARRY				1	1	1	1	

Convert the second binary number into a negative binary number using Two's Complement.

	-128	64	32	16	8	4	2	1
86	0	1	0	1	0	1	0	1
-16	1	1	1	1	0	0	0	0
SUM	0	1	0	0	0	1	0	1
CARRY	1	1	1					

Add the first binary number and the converted second binary number together using the binary addition rules.

1 | 01000101

Write the answer.

There is an overflow error.

Issues with Two's complement

Information: Adding 3-bit binary numbers

A / B	000	001	010	011	100	101	110	111
000	000	001	010	011	100	101	110	111
001	001	010	011	100	101	110	111	000
010	010	011	100	101	110	111	000	001
011	011	100	101	110	111	000	001	010
100	100	101	110	111	000	001	010	011
101	101	110	111	000	001	010	011	100
110	110	111	000	001	010	011	100	101
111	111	000	001	010	011	100	101	110

While there are fewer incorrect answers as some of the binary numbers are now correct, coloured in green, there are still some incorrect answers, coloured in red. However, we are now able to handle negative numbers and perform subtraction.

Transforming Data

Signed fractional Two's complement

Definition

Signed fractional Two's complement is a way of representing fractional binary numbers where the most significant bit (MSB) is always negative.

Representation

The representation of a signed fractional Two's complement binary number can also be distinguished by using the binary place value system in which:

- each bit represents a power of 2;
- the most significant bit (MSB) has a negative integer power while all other powers are positive integers;
- the rightmost bit will be the lowest power; and
- there is an integer format which follows $-(2^0)2^{-1}2^{-2}2^{-3}2^{-4}2^{-5}2^{-6}2^{-7}$ (8 bit).

Information: Binary place values for an 8-bit signed fractional Two's complement binary number

Position	8	7	6	5	4	3	2	1
Weight *(base 2)*	-128	64	32	16	8	4	2	1

The minimum positive decimal value for a signed fractional Two's complement binary number constructed using n bits is 0.
The minimum non-zero decimal value for a signed fractional Two's complement binary number constructed using n bits is -1.
The maximum decimal value for a signed fractional Two's complement binary number constructed using n bits is calculated by $1 - \frac{1}{2^{n-1}-1}$.

An 8-bit signed fractional Two's complement binary number has:

- a minimum positive decimal value of 0, when the binary value is 00000000;
- a minimum non-zero decimal value of -1, when the binary value is 10000000; and
- a maximum decimal value of $1 - \frac{1}{128}$, when the binary value is 01111111.

The minimum non-zero decimal value and the maximum decimal value are not symmetrical as there has to be a combination available for zero (0). This removes one option of a combination of binary bits from the positive values but not rather than the negative values as all negative numbers are non-zero.

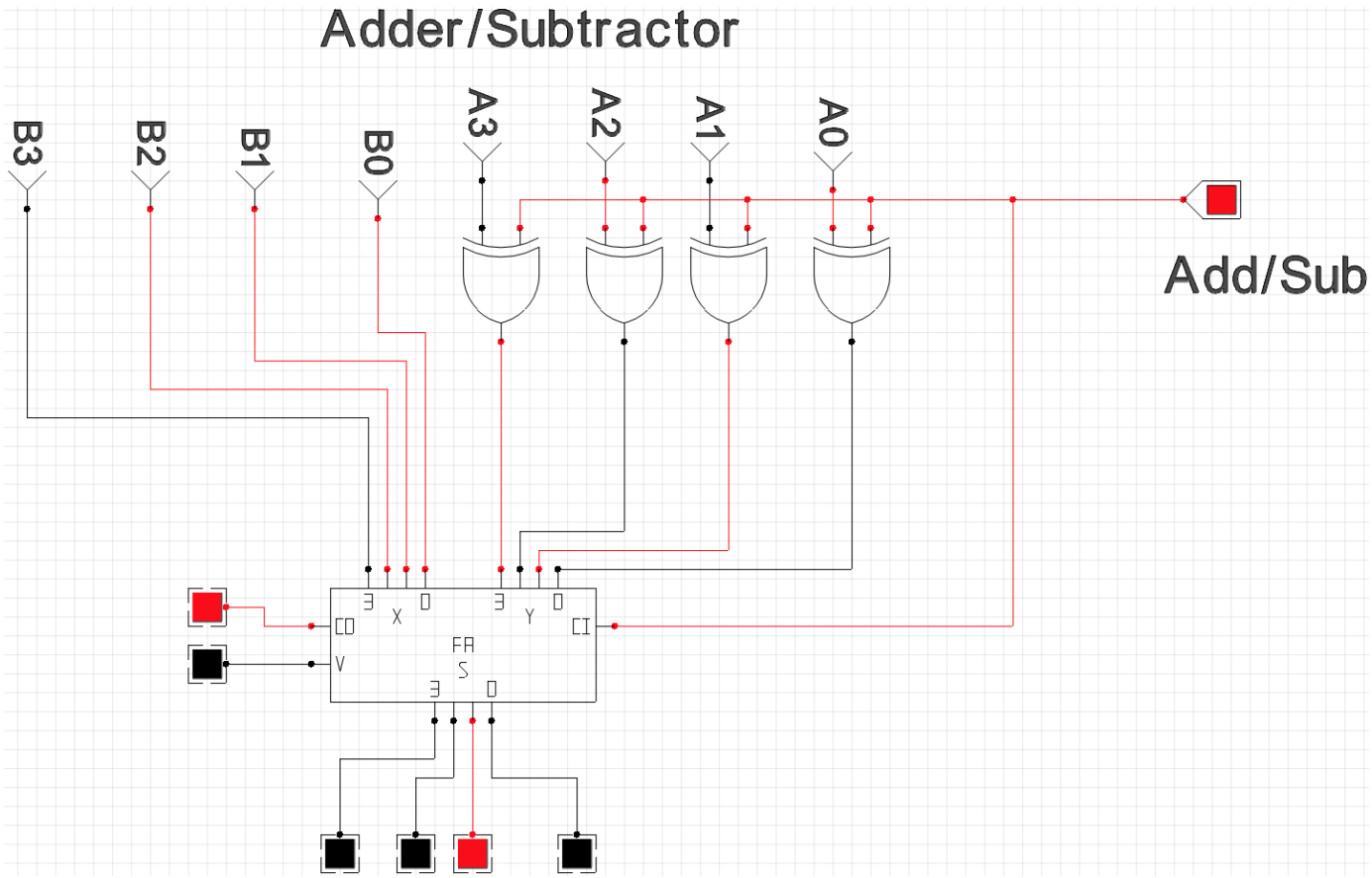
This representation uses fixed point and therefore the binary point can be placed anywhere in the word if required. The signed fractional Two's complement place values will be present to the right of the fixed binary point and fixed point arithmetic can be performed.

Transforming Data

Full adder with subtraction capabilities

When a given number arrives as a positive, it must be converted to a negative number. The negative number is represented using Two's complement by inverting all of the bits and adding one.

Diagram: 4-bit full adder with subtraction capabilities



This is achieved using a 4-bit full adder and a set of exclusive OR (XOR) gates.

The exclusive OR (XOR) gates are used as a controllable invert that can be activated when a positive binary number is to be converted to a negative binary number.

Truth Table: XOR gates

Add/Sub	$A0$ / $A1$ / $A2$ / $A3$	Output
0	0	0
0	1	1
1	0	1
1	1	0

By inspecting the truth table, it can be seen that:

- if the Add/Sub input is low, the bit from the number will be left unchanged; and
- if the Add/Sub input is $high$, the bit from the number will be inverted.

When the Add/Sub input is $high$, the carry in (C_{IN}) on the full adder (FA) will be enabled in order to add one (1) to the binary number - in accordance with Two's complement.

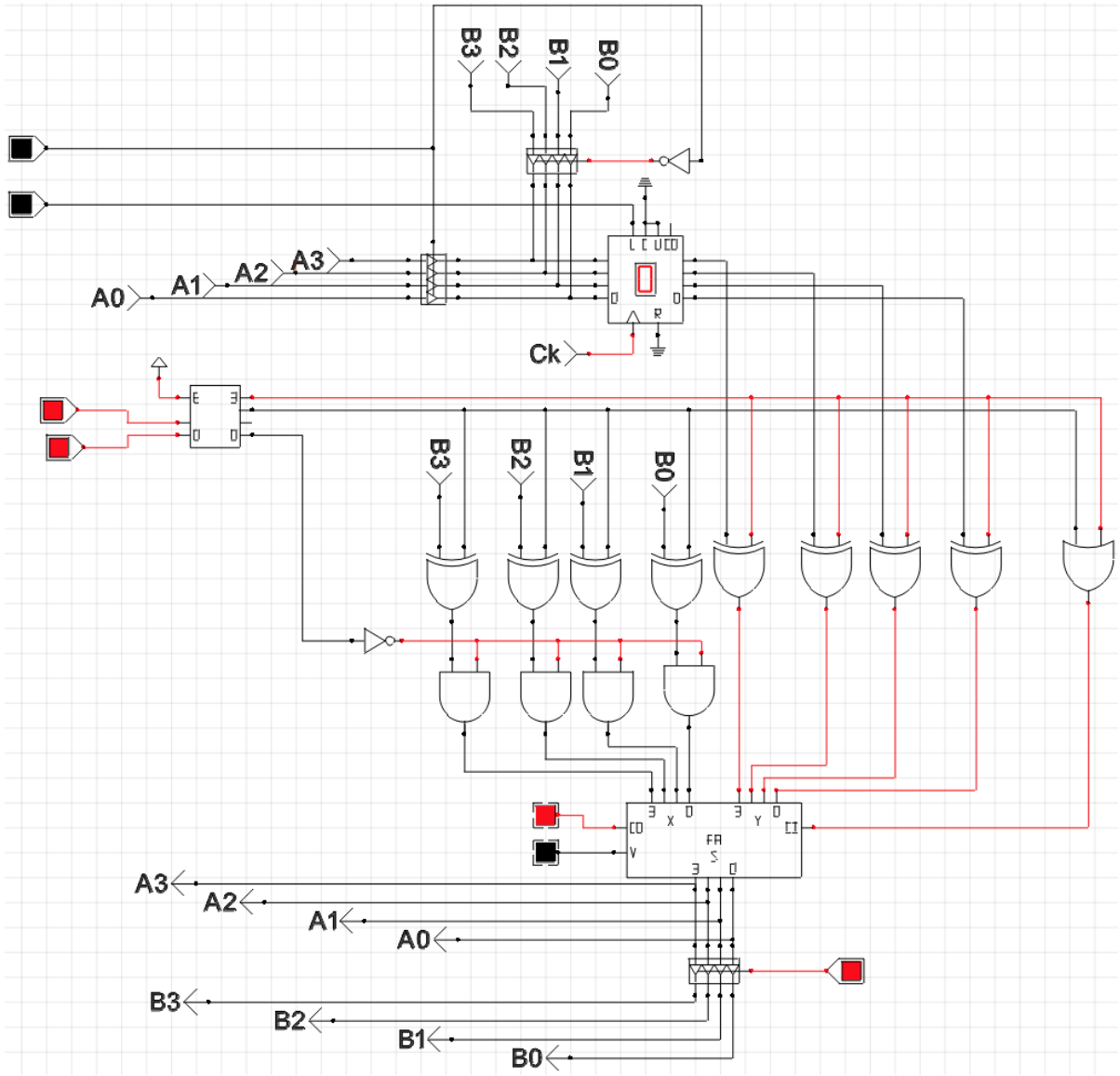
Accumulator

Addition and Subtraction Capabilities

Circuit

An accumulator with addition and subtraction capabilities is able to perform a range of addition and subtraction operations and store data in a register.

Diagram: Accumulator with addition and subtraction capabilities



Wires denoted using A are concerned with the accumulator and wires denoted using B are concerned with the bus.

Exclusive OR (XOR) and AND gates are used to provide control operations. AND gates are used to kill inputs (set to low).

This accumulator circuit is capable of performing the operations:

- $A = A - B$;
- $A = B - A$;
- $A = A + B$;
- $B = A$; and
- $A = B$.

In a real computer system, the switches on the accumulator circuit will be driven by the contents of the $instruction\ register$ (IR).

Accumulator

Conditionals

Definitions

Conditionals are statements included in programming languages that perform different computations or actions depending on whether a programmer-specified Boolean condition evaluates to True or False.

Jump instructions

Jump instructions are required when the flow of a computer program no longer follows the sequential flow and rather moves to a different location as a result of instructions such as `if` statements and `while` loops.

When a condition, such as `if (x > y)`, evaluates to True, the program must execute the code specified to be executed when the condition evaluates to true. When a condition evaluates to False, the program must execute the code after the current block of code.

Jump operations require changing the contents of the $program\ counter\ (PC)$. A jump instruction is sent as a control from the result of an instruction, such as an arithmetic operation. The instruction is a "test" as it determines if the jump instruction is required.

The information required to perform a jump instruction includes:

- the variables;
- the destination of the jump; and
- the condition.

Unfortunately, all of the information required to perform a jump instruction cannot be encoded into a single instruction. In order to overcome this, condition codes are used.

Condition codes

Condition codes are a group of bits indicating the condition of something inside a computer, often used to decide which instructions the computer will subsequently execute.

These are sometimes called status bits and the names for these bits can be used interchangeably. However, depending on the design of a processor, condition codes and status bits may have different but similar meanings; condition codes are usually a subset of status bits.

Typically, four flip flops can be used such that they are loaded with values after each operation:

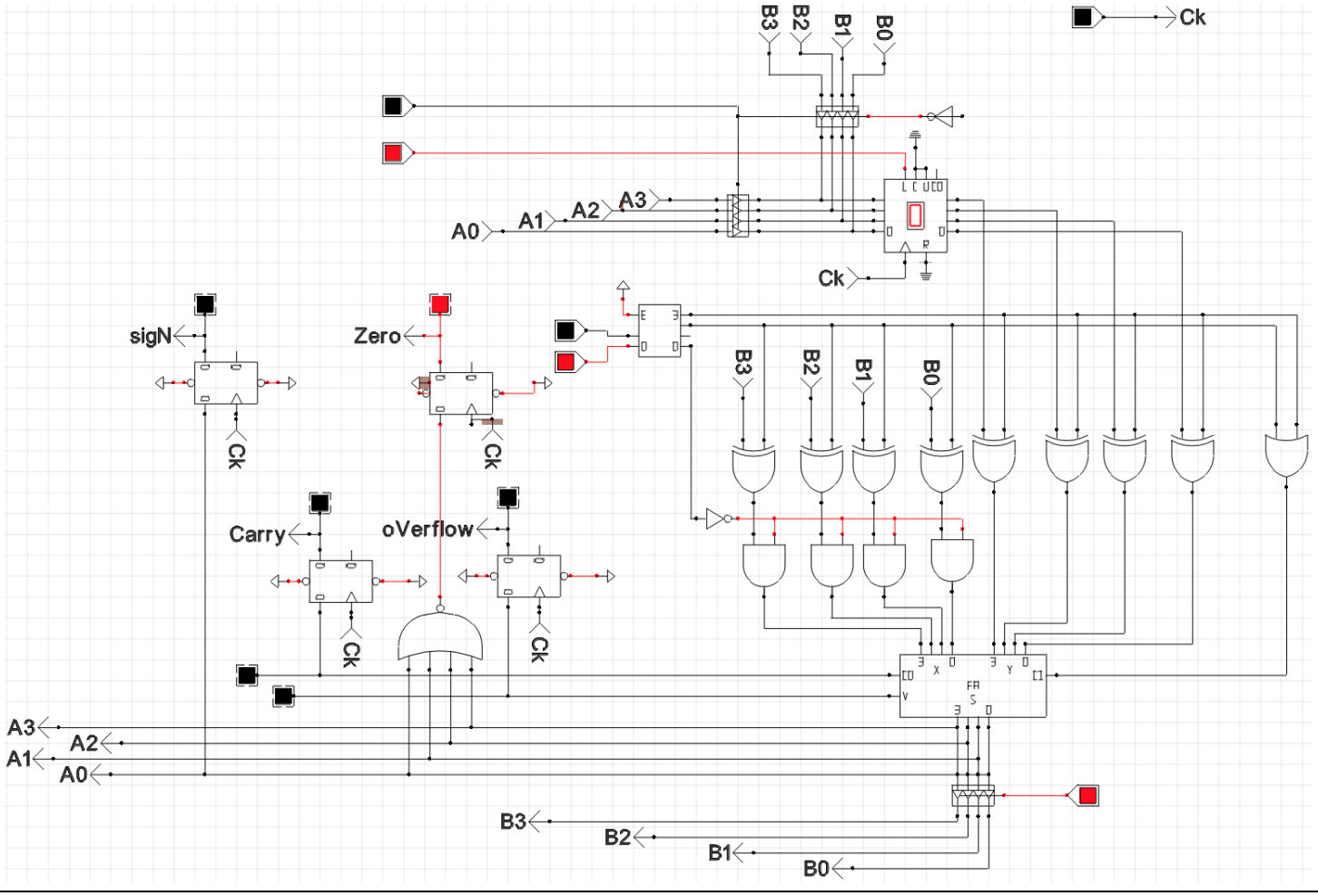
- carry – this records the carry out (C_{OUT}) from the topmost bit of an adder – this is an overflow for unsigned binary numbers;
- overflow – this is used to detect if the sign of the result is correct, it helps to determine whether the number can be represented using the bits available – for example, if an addition operation takes place on two positive numbers and the result is negative, it is known that the sign is incorrect;
- sign (N) – this records the top bit of the result and is usually denoted as N for negative; and
- zero – this is a test to check if the result is zero – for example, if $X = 5$ and $Y = 5$ where the operation is $X - Y$ then the test for zero would be True.

Accumulator

Accumulator with condition codes

Condition codes can be implemented into the accumulator by adding the four flip flops.

Diagram: Accumulator with condition codes



The data inputs for the four flip flops are connected such that:

- $Zero$ data input – is connected to the ;
- $oVerflow$ data input – is connected to the ;
- $Carry$ data input – is connected to the carry out (C_{OUT}) from the full adder; and
- $sign$ data input – is connected to the .

Using condition codes requires a sequence of two instructions.

- The first instruction performs an arithmetic operation, usually a subtract operation. After which, the status of the result is recorded in the condition flip flops – this will affect the subsequent actions of the control unit.
- The second instruction performs the jump conditional jump based on the status recorded. The condition flip flops are available to the control unit and can the control unit is used to ensure that the components in the processor are setup correctly depending on whether the jump instruction is to be executed or not. This usually required a logical combination of the bits.

For example, after a subtract operation, $A - B$, the condition $A < B$ is True if $\overline{\oplus V}$. If the overflow bit is high, then the sign is exactly oppositely incorrect to the correct sign – this can be rectified as the sign is not arbitrarily incorrect.

Designing and Implementing the Processor

Design Choices

Decision factors

When making the design choices, the capabilities of CEDAR Logic have been taken into account and therefore are generally based on what is conveniently achievable within the scope of CEDAR Logic's capabilities.

Word length

Word length refers to the number of bits processed by a computer system's processor in one clock cycle. Data bus size, instruction size, address size are usually multiples of the word size.

The word length for this processor will be 16 bits

A 16-bit word length is typical of minicomputers from the 1970s and the first IBM PCs. A larger word length would require more wiring in CEDAR Logic and would distract from the mechanics of the processor.

While some processors split the word, in this processor one word will be used and no subdivisions will be used.

Address length

Address length refers to the number of bits used to contain the address or data for an instruction.

The address length for this processor will be 12 bits

The address length is four bits smaller than the word length. This allows the instruction and a 4-bit opcode to be used in one clock cycle and therefore simplifies the design of the processor as all instructions can be executed in a single cycle and decoding is made easier.

Other architectural features

One circuit for each component	–	While many real processors generally have duplicated circuits in order to increase performance – for example, while this processor will only have one accumulator, real processors may have eight data registers.
Simple timing cycle	–	The timing cycle will alternate between fetch and execute. The fetch instruction will take one clock cycle to complete and the execute instruction will also take one clock cycle to complete.
No complicated instructions	–	For example, there will be no multiple or divide instructions as these can be achieved using software where a combination of other instructions will be used.

Designing and Implementing the Processor

Instruction Set

Standard instruction set

Some instructions require an address within the instruction.

Diagram: Instruction set with address

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
I	I	I	I	A	A	A	A	A	A	A	A	A	A	A	A

In this instruction set:

- four bits are used for the opcode (I) in order to represent the instruction; and
- 12 bits are used for the operand (A) in order to represent an address (A).

Whereas, some instructions require literal data within the instruction.

Diagram: Instruction set with data

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
I	I	I	I	D	D	D	D	D	D	D	D	D	D	D	D

In this instruction set:

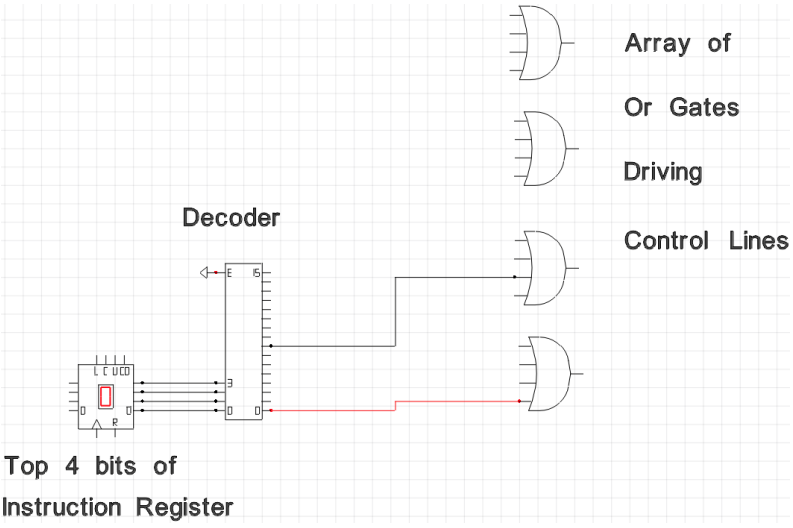
- four bits are used for the opcode (I) in order to represent the instruction; and
- 12 bits are used for the operand (D) in order to represent data (D).

These instruction set formats are typically used for the following instructions:

- moving data from memory to the accumulator;
- moving literal data into the accumulator;
- moving data from arithmetic operations, such as addition and subtraction operations, to the accumulator so that the result of the operation is stored in the accumulator;
- moving data from the accumulator to memory; and
- jump instructions.

Decoding these instructions can be achieved using a modified general logical strategy.

Diagram: Instruction set decoder



Designing and Implementing the Processor

The top four bits of the $instruction\ register$ (IR) are fed into the decoder that activates one of the 16 output lines depending on the combination of the four input bits. This assumes that every control signal is active when $high$, whereas in a real processor control signals are usually active when low. The control for every enable or signal are connected using an OR gate. This means that if a control signal needs to be high, all that is required is a $high$ signal from the decoder to the OR gate. By choosing which connections are made between the decoder and the OR gates, the instruction set can be implemented.

Alternative instruction sets

Not every instruction will require all 12 bits of the operand for the address (A) or data (D) and therefore it is useful to have some instructions where only a smaller portion of the word is used to represent the operand. As a result, an alternative instruction set may be used for some instructions.

Diagram: Alternative instruction set with address

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	I	I	I	I	A	A	A	A	A	A	A	A

In this alternative instruction set:

- eight bits are used for the opcode (I) in order to represent the instruction; and
- eight bits are used for the operand (A) in order to represent an address (A).

Diagram: Alternative instruction set with data

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	I	I	I	I	D	D	D	D	D	D	D	D

In this alternative instruction set:

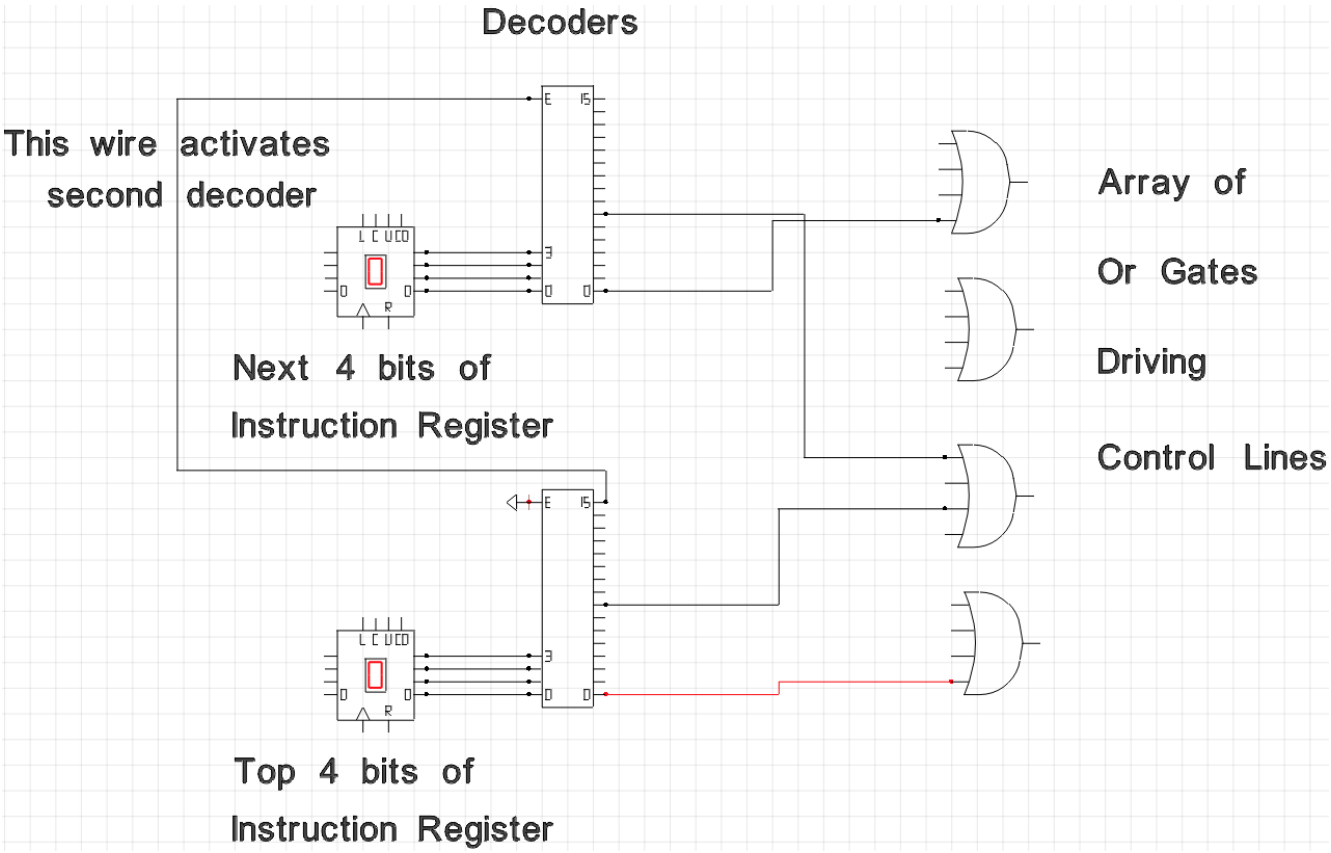
- eight bits are used for the opcode (I) in order to represent the instruction; and
- eight bits are used for the operand (D) in order to represent data (D).

In order to use this alternative instruction set, an instruction from the first instruction set must be used in order to activate the alternative instruction set.

Decoding these instructions can be achieved using a modified general logical strategy and “chaining” the decoders together.

Designing and Implementing the Processor

Diagram: Instruction set decoder with alternative instruction set implemented



Activating the top decoder in order to use the alternative instruction set can be achieved by setting the top four bits of the instruction register to all one's (1111) so that output 15 is $high$ and therefore makes the enable (E) $high$ on the top decoder.

This method of implementing an alternative instruction set can be repeated to create another instruction set for instructions that require a larger opcode and do not require such a large operand.

Diagram: Alternative instruction set with data

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	I	I	I	I	D	D	D	D

In this alternative instruction set:

- 12 bits are used for the opcode (I) in order to represent the instruction; and
- four bits are used for the operand (D) in order to represent data (D).

Diagram: Alternative instruction set with a full address

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	1	1	1	I	I	I	I

In this alternative instruction set:

- 16 bits are used for the opcode (I) in order to represent the instruction; and
- there is no operand.

This uses the same decoding strategy as seen with the previous alternative instruction set. This is similar to mechanisms used by real processors, however, they may be more complicated.

Designing and Implementing the Processor

Implementing the conditional jump

There are 15 available instructions when using the alternative instruction set with a full address. These can be used for condition codes.

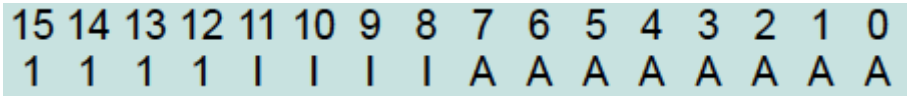
Condition codes have many choices and therefore each of the 15 available instructions can be allocated to actions that need to be taken based on the status of condition codes.

Around seven or eight instructions that require the alternative instruction set with a full address can be identified however, there are ten possible conditions:

- GT;
- LT;
- GE;
- LE;
- EQ;
- V;
- C;
- $\bar{V}$; and
- $\bar{C}$.

These conditions will occupy many of the available opcodes and instructions must be left available for expandability when adding more instructions in the future. Therefore, it is necessary to use some of the instructions from the 8-bit alternative instruction set.

Diagram: Alternative instruction set with address



The address (A) will be implemented as an offset, also known as a relative address. An offset denotes the number of address locations added to or subtracted from a base address, or current address, in order to get to a specific absolute address.

Designing and Implementing the Processor

Strategy for the Design in CEDAR Logic

Pages

Due to the large size of the circuit, multiple pages will be used in order to keep the content of each page down to manageable proportions. Each page will contain a major component and some components may have to be split across multiple pages if they become too large.

A “front panel” page will be used to display the contents of the registers and buses conveniently.

See **Appendix A** for the full final revision of the processor design

Labels

There will be extensive use of labels in order to connect components across pages.

Control signals will be on one page and therefore will no longer be connected with “wires”, instead labels will be used.

See **Appendix B** for the full final revision of the processor instruction set

See **Appendix C** for the full list of control signal labels

Designing and Implementing the Processor

Page 1 – Front Panel

Contents

This page is designed to show an overview of the processor.

The front panel displays the contents of the:

- registers;
- data bus; and
- address bus.

Diagram

Diagram: Front panel
See page 101.

How it works

The XOR gate incorporating the different clock signals allows multiple switches across multiple pages to be used to manually clock the processor.

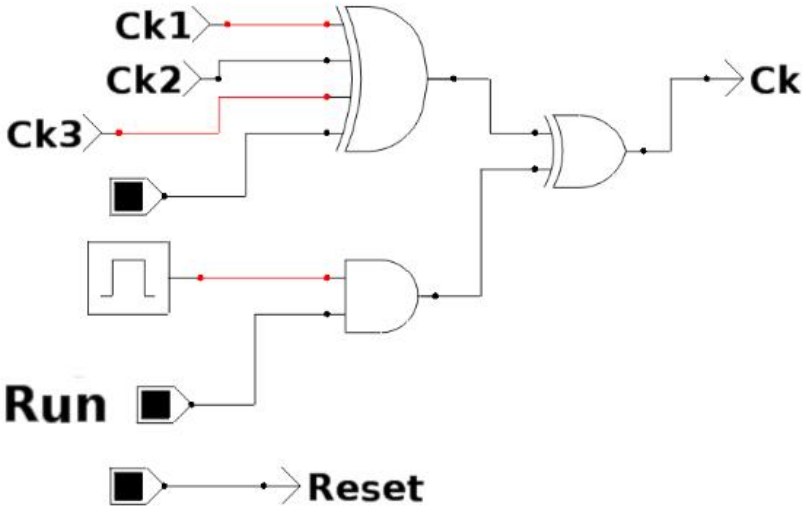
The clock cycles of the processor can be controlled by:

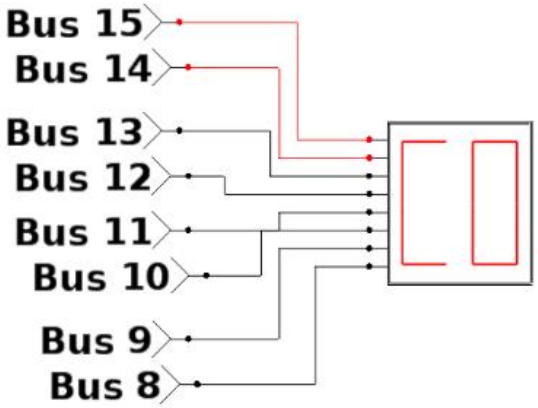
- manually controlling the input to the clock signal XOR gate; or
- setting Run to $high$ so that the oscillator can control the clock speed and the processor will run continually.

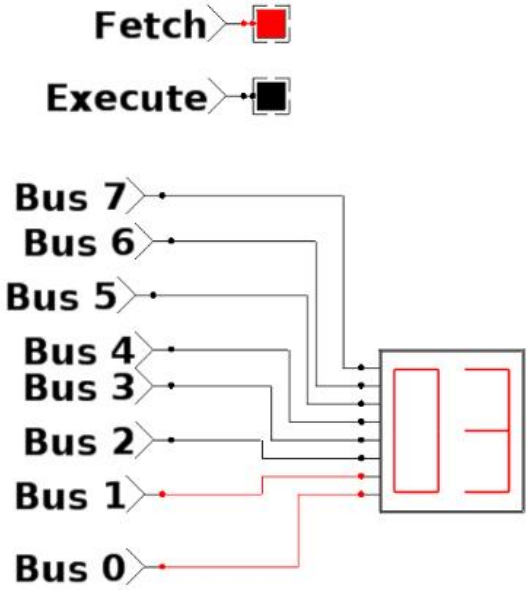
The frequency of the oscillator can be inspected and changed by double-clicking and if the circuit malfunctions, lower the frequency as the circuit may be running too quickly.

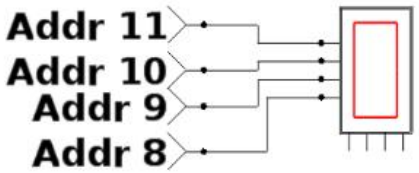
The values in the processor can be reverted to their default by setting $Reset$ to $high$ and clocking the circuit a few times.

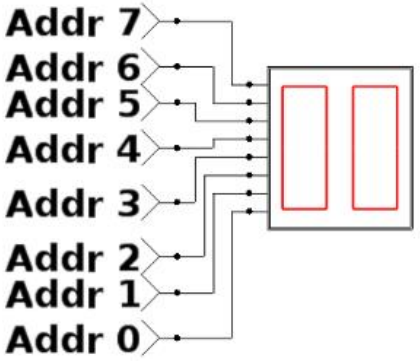
Designing and Implementing the Processor











Designing and Implementing the Processor

Page 2 – Memory

Contents

This page contains the memory (RAM) for the processor.

Diagram

Diagram: Memory
See page 103.

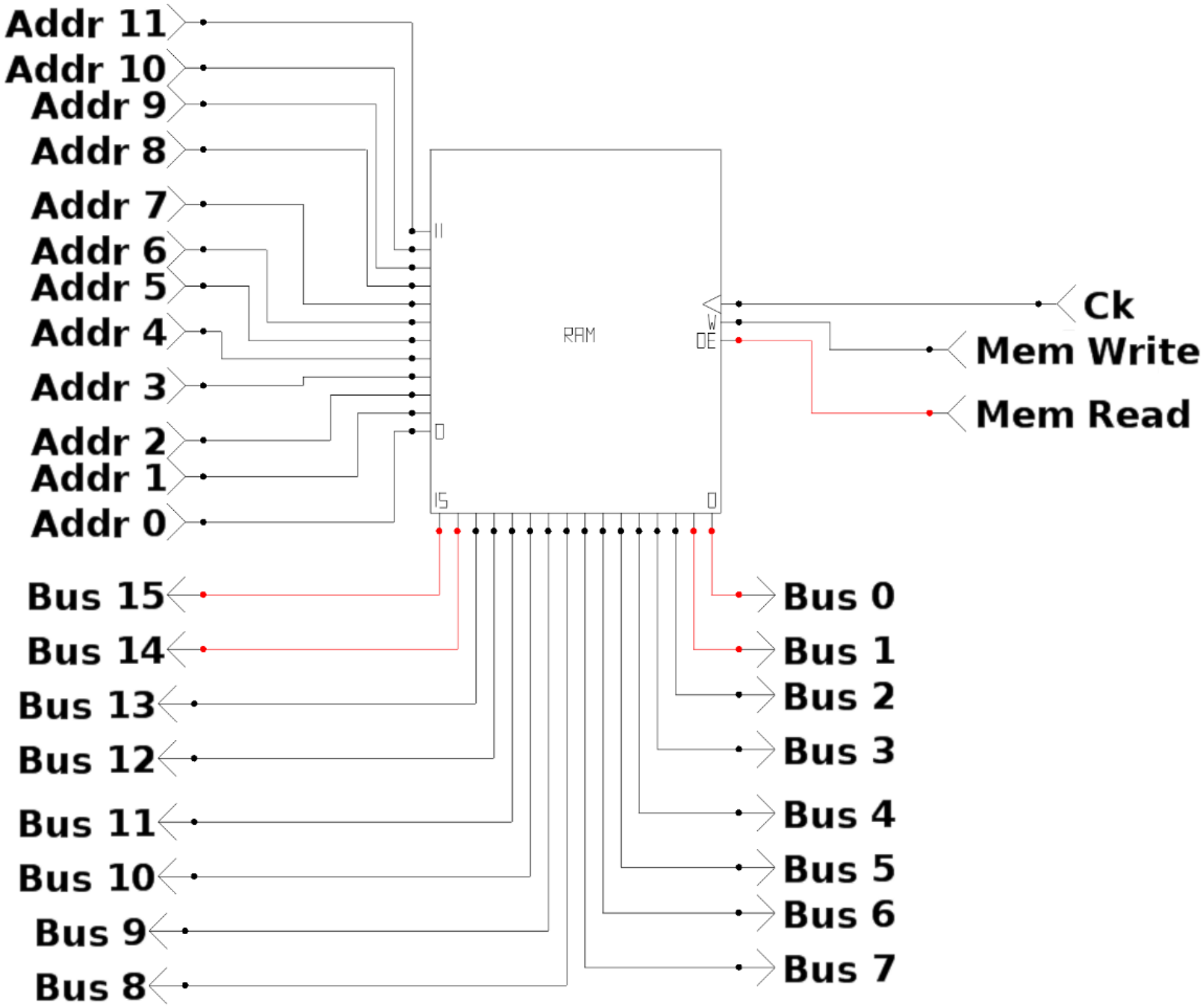
How it works

Each memory location can contain 16 bits of data.

The contents of the memory can be inspected and changed by double-clicking and the memory display can be seen on other pages.

This is important as all parts of the computer system, including peripherals, communicate with memory.

Designing and Implementing the Processor



Designing and Implementing the Processor

Page 3 – Instruction Register and Initial Decode

Contents

This page shows the $instruction\ register\ (IR)$ and how its output is used to select an instruction. This page also shows the connections between the operand (address/data field) of the instruction and the data bus, address bus and program counter input bus. Note the sign extension hardware connecting to the data bus.

Diagram

Diagram: Instruction register and initial decode

See page 104.

How it works

An instruction is fetched from memory and passed into the decoder that inspects the sequence of zeros and ones to determine which output line should be $high$ and therefore which instruction should be active.

There are multiple decoders that can be used for different instruction sets:

- first decoder – selects instructions that use the instruction set with a 4-bit opcode and a 12-bit operand;
- second decoder – selects instructions that use the instruction set with a 8-bit opcode and a 8-bit operand;
- third decoder – selects instructions that use the instruction set with a 12-bit opcode and a 4-bit operand; and
- fourth decoder – selects instructions that use the instruction set with a 12-bit opcode and no operand.

These decoders can be activated by setting an octet (4-bit chunk) of the instruction register to ones (1111) so that output 15 is $high$ and therefore makes the enable (E) $high$ on the subsequent decoder.

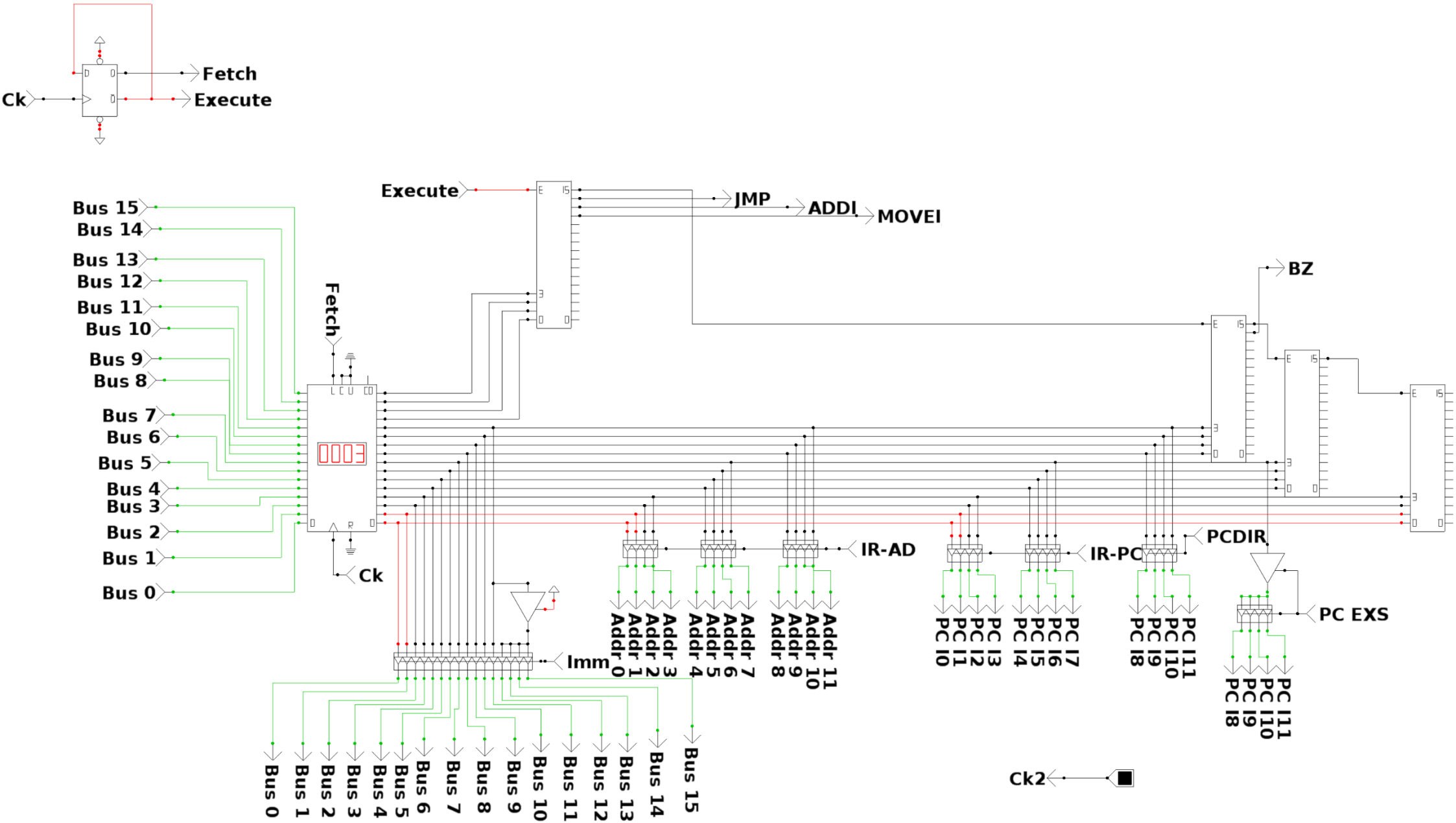
Truth Table: Activating the decoders

Instruction Register Contents		Activated Decoder
Binary Value	Hexadecimal Value	
0000000000000000	0000	First Decoder
1111000000000000	F000	Second Decoder
0000111100000000	0F00	Third Decoder
0000000011110000	00F0	Fourth Decoder

NOTE: The binary and hexadecimal values for zero can be replaced with any other combination of zeros and ones apart from another combination in which an octet (4-bit chunk) is all ones.

Each output from the decoders is attached to a labelled connection representing an instruction; these connections are visible again on page 4.

Designing and Implementing the Processor



Designing and Implementing the Processor

Page 4 – Instruction Decode

Contents

This page shows the decoding process for instructions generated by the initial decode process.

Diagram

Diagram: Instruction decode
See page 106.

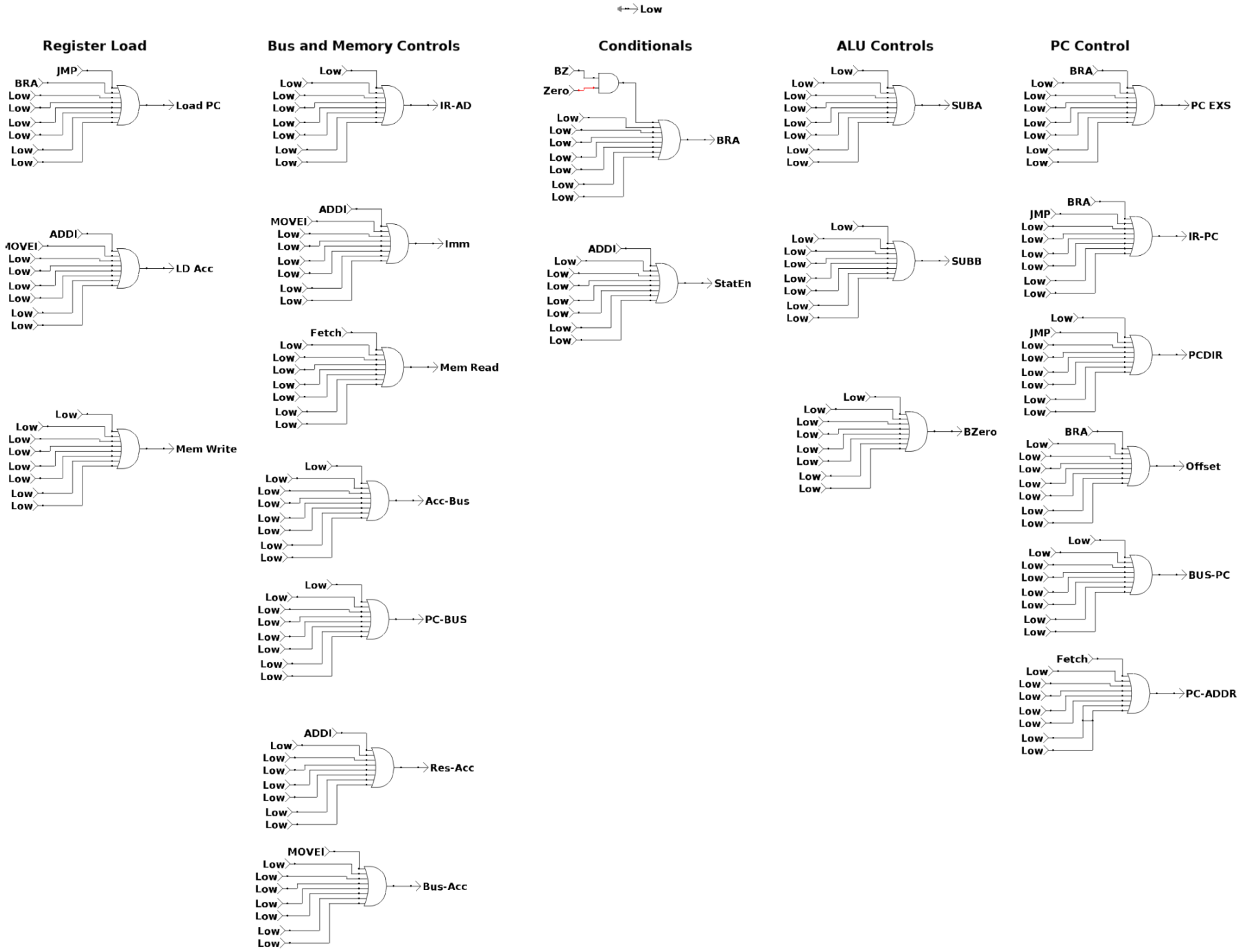
How it works

Each of the OR gates, apart from those in the $Conditionals$ column, drive one control signal.

For particular instructions, defined by a sequence of ones and zeros, different control signals are activated. This allows components inside the processor to be enabled and disabled in accordance with the requirements of a given instruction.

Many of the inputs to the OR gates have been set to Low; the connection Low is wired up to a ground component. This allows for expansion in the future as instructions can be implemented when added in place of the Low inputs.

Designing and Implementing the Processor



Designing and Implementing the Processor

Page 5 – Accumulator (arithmetic unit)

Contents

This page shows the accumulator in which some processes are concerned with performing arithmetic using full adder circuits.

Diagram

Diagram: Accumulator
See page 109.

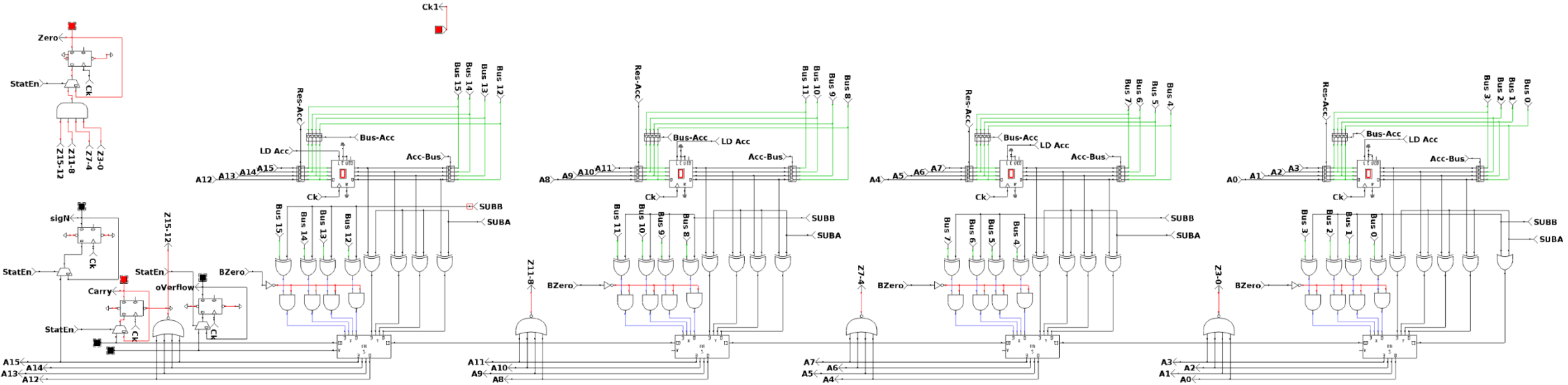
How it works

The circuit uses 16 bits and makes use of four 4-bit full adders concatenated together.

Input data for the full adder may:

- come into the register from the data bus;
- go out of the register on to the data bus; or
- be identical to the result of a previous operation – the bus labelled $A15$ to $A0$ is the result from the full adder, the control bits make use of a multiplexer so that they are only loaded when demanded to be so by the $StatEn$ control.

Designing and Implementing the Processor



Designing and Implementing the Processor

Page 6 – Program Counter

Contents

This page shows the program counter.

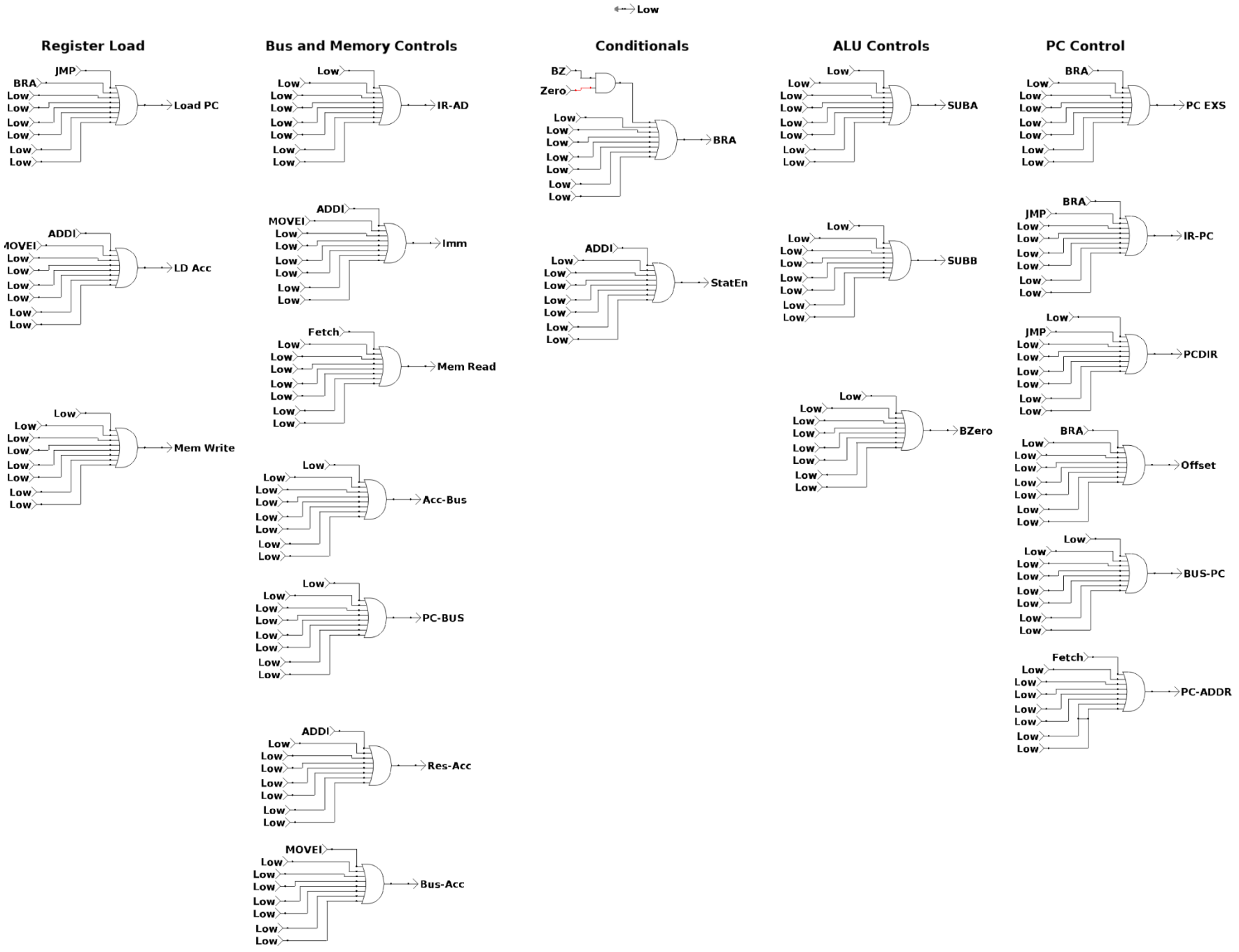
Diagram

Diagram: Program counter
See page 111.

How it works

Note that the input is via an adder so we can do relative branches as described in the lecture. Note that the sign extension circuitry for this is on page 3 – bottom right. Input to this circuit is via a special bus PC I (11-0), allowing more direct access from the instruction register. This will avoid conflicts in some later.

Designing and Implementing the Processor



Designing and Implementing the Processor

Address Register

Array constructs in high level languages

Array constructs in high level programming languages can be accessed using syntax such as `a[i]` where `a` is the name of the array and `i` is the index of the element in the array.

This location is specified in the instruction and therefore will be specified in somewhere in the code. However, this address cannot be easily modified according to the data. As a result, another mechanism is required as self-modifying code is dangerous.

Addressing the issue using the address register

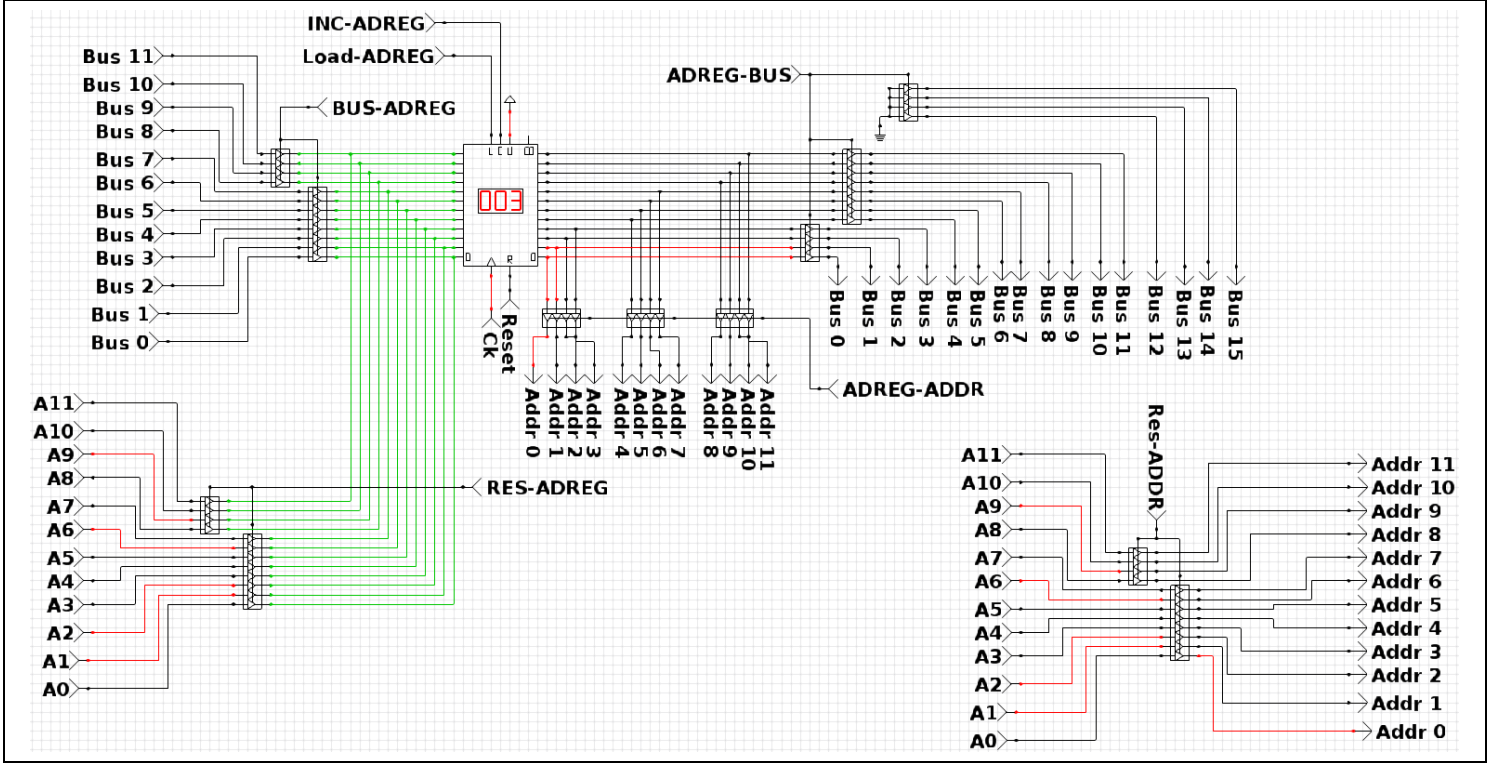
Definition

The **address register** stores either the memory address from which data will be fetched from the CPU, or the address to which data will be sent and stored. This means that the address register holds the memory location of data that needs to be accessed.

Implementation (added to page 7 of the processor)

An auto-increment is implemented to assist array access by “stepping through” the elements in the array.

Diagram: Address register



The address register is a 12-bit register as the processor has a 12-bit address.

Designing and Implementing the Processor

This register is connected in a similar manner to the accumulator. The address register is connected to the:

- address bus – to allow the register to directly address memory and select memory locations;
- data bus – to allow data to be taken from the bus into the address register or for data from the address register to be put on the bus; and
- accumulator – to allow arithmetic to be performed for array index calculations and allow results from the accumulator to be loaded into the address register.

This means that the address register can:

- perform calculations where the source memory location is used for the sum and destination;
- load and store to/from memory; and
- use a memory location as an address (address source).

The address register can be incremented or decremented. However, in this implementation, the up (U) signal on the address register is high and therefore, the address register can only be incremented.

Typical array operations can be performed using the address register.

Information: Array operations using address register

Accessing an element	Sequential access
Accessing a specific element, such as `a[i]`, can be achieved by: • loading the base address (location of the first element of the array) unto the address register; • loading the index (`i`) into the accumulator (data register); • adding the contents of the address register to the contents of the accumulator; • sending the result of the addition calculation to the address register; and • using the address register to access memory.	Sequential access, or repeated access, can be achieved by using the auto-increment facility to access the next element in the array.

Designing and Implementing the Processor

Functions in high level languages

When calling a function in a high level language, a jump is performed to the code at the start of the function.

A return address is the location just after the function call. This information must be stored so that, once the function has been completed, the program can be resumed from its previous location.

Storing the return address could be achieved by introducing a new register for this purpose. However, this is likely to cause issues as functions can be "nested", where functions are called within other functions, and therefore more than one return address would need to be stored. As a result, the stack is used.

Addressing the issue using a stack pointer

Definition

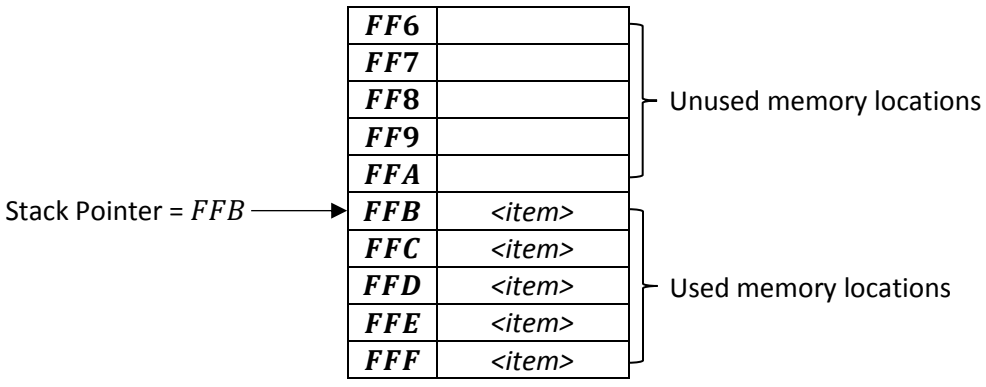
A **stack** is an abstract, linear, static, contiguous and mutable "Last In, First Out" (LIFO) data structure, where the last item added is the first item to be accessed.

- Abstract – Created by a programmer and is a logical description of how the data is viewed and the operations that can be performed on the data. This makes use of encapsulation as the details of implementation are hidden from the user.
- Linear – The elements form a sequence.
- Static – Memory for the stack is allocated from the heap when a program is compiled, therefore it is fixed in size and cannot change size or space taken in memory during program execution.
- Contiguous – Stored in memory addresses one after the other.
- Mutable – Data within the stack can be edited, deleted or moved after they have been defined.

How data is stored

The stack is stored in memory and it is common to have the data for the stack out of the way from other data stored in memory.

Diagram: Abstract visualisation of a stack in memory



placeholders are written in italics and enclosed by angle brackets (< >).

The stack pointer is a register and stores the location of the top of the stack. In this implementation, the stack makes use of the last memory location (FFF) as the starting point for the stack and then works backwards through the memory locations with subsequent items. This means that the stack "grows" from high memory address towards low memory addresses.

Designing and Implementing the Processor

Stack operations

Two main operations can be performed on a stack:

- pre-decrement (push) – decrements the stack pointer and adds the new item to the location where the stack pointer is pointing; and
- post-increment (pop) – increments the stack pointer so that the item on the top of the stack is effectively removed.

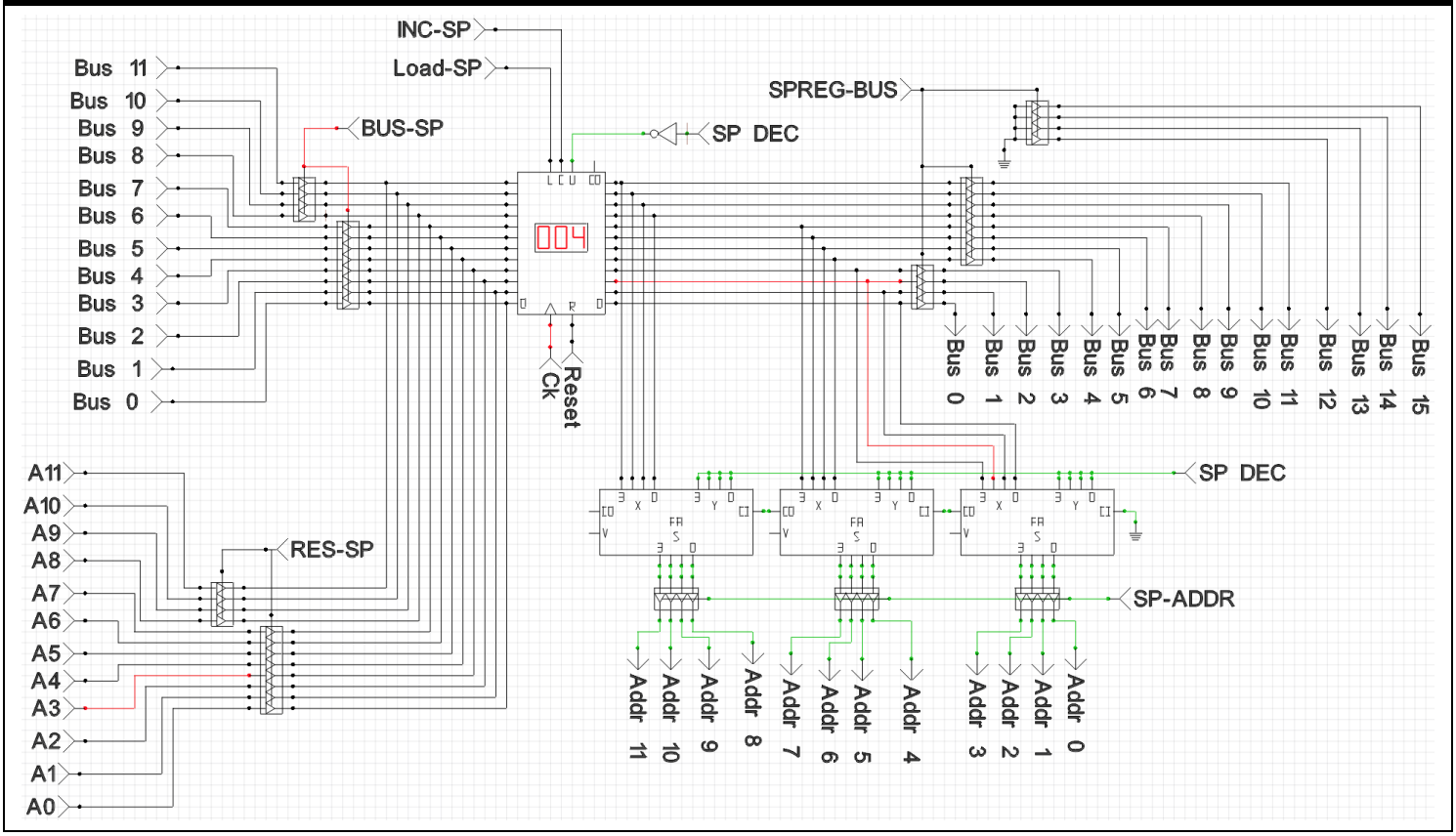
Using these operations, it is possible to store return addresses in successive memory locations and access them in the correct order:

- when adding a return address, a pre-decrement is performed and the contents from that location is stored in the $program\ counter\ (PC)$; and
- when removing a return address, a post-increment is performed and the contents are from that location are read.

Implementation (added to page 8 of the processor)

The stack pointer is added to the address register using three 4-bit full adders.

Diagram: Stack pointer



The stack pointer uses 12 bits as the processor has a 12-bit address.

Incrementing and decrementing the stack pointer is achieved using $SP\ DEC$:

- pre-decrement (push) – $SP\ DEC = 0$ such that the input is 0; and
- post-increment (pop) – $SP\ DEC = 1$ such that the input is -1.

The full adders are used to ensure that the address that goes out when doing a pre-decrement is decremented using $SP\ SEC$.

Typical function operations can be performed using the stack pointer.

Designing and Implementing the Processor

Information: Function operations using stack pointer

Calling a function	Returning from a function
A function call can be achieved by using a Jump SubRoutine (`JSR`) instruction. This will: • set $SP\ DEC$ and $SP - ADDR$ to high; • perform a jump; and • store the contents of the $program\ counter\ (PC)$ on to the stack by pre-decrementing the stack pointer. With a suitably wired stack pointer register, where a full adder is included in the path to the address bus, the processor is able to perform a function call in one clock cycle.	Returning from a function involves retrieving the previous program counter (PC) contents from the stack by post-incrementing the stack pointer.

Designing and Implementing the Processor

Executing Instructions

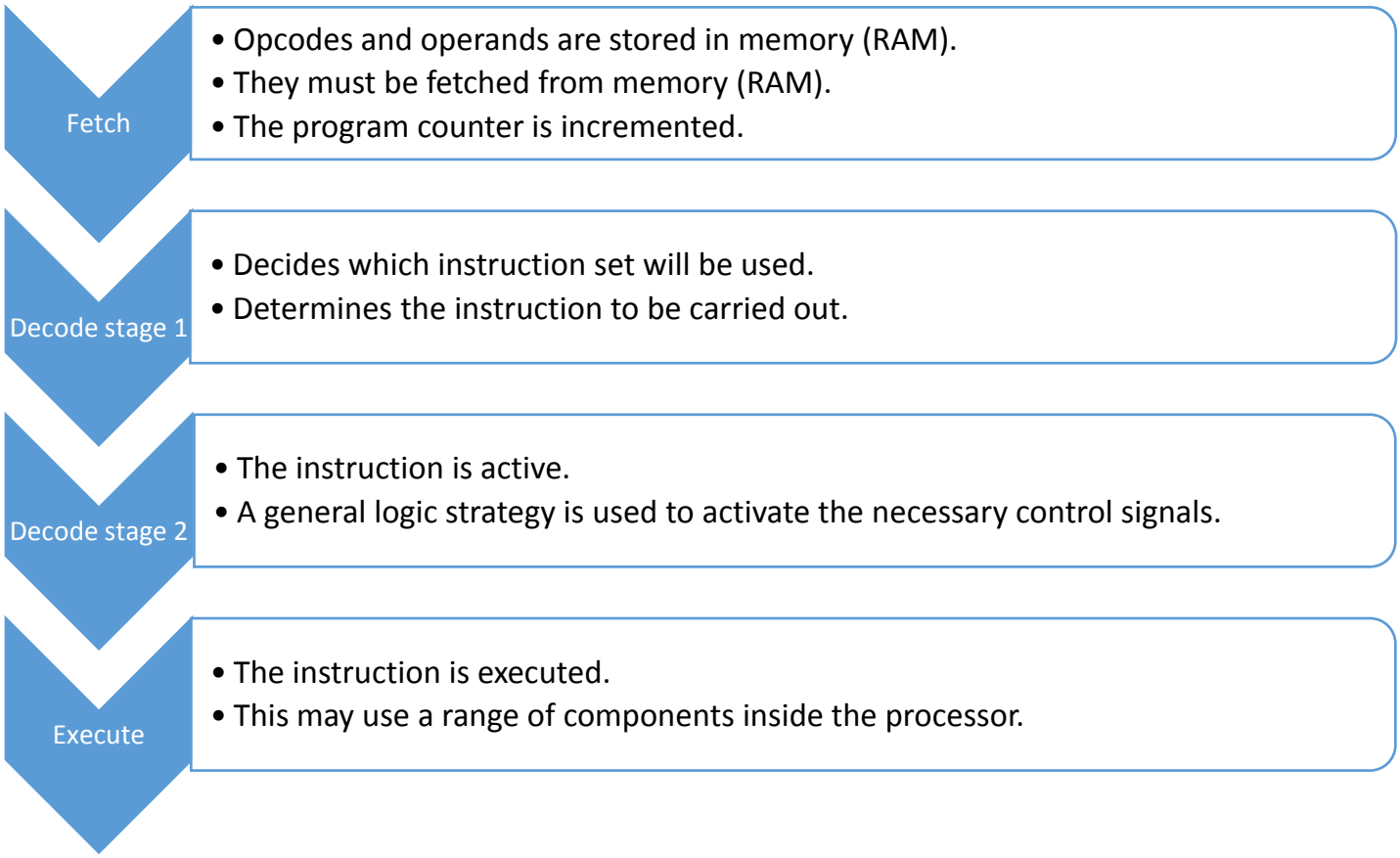
Clock cycle

Definition

Clock cycle is the speed of a computer system's processor and is determined by the clock cycle, which is the amount of time between two pulses of an oscillator.

Process

Diagram: Clock cycle process

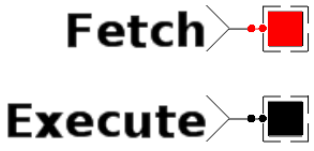
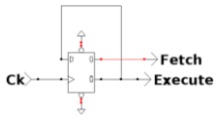
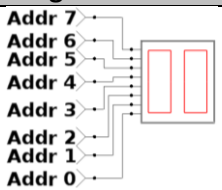
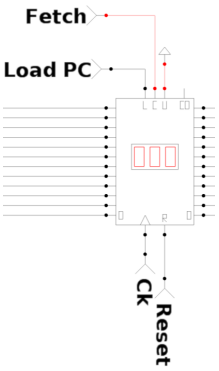
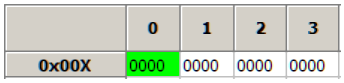
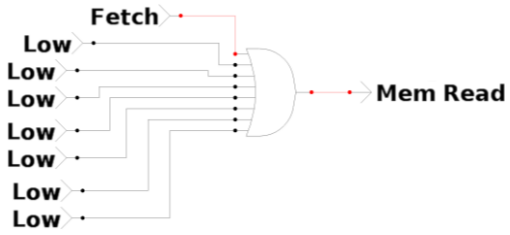
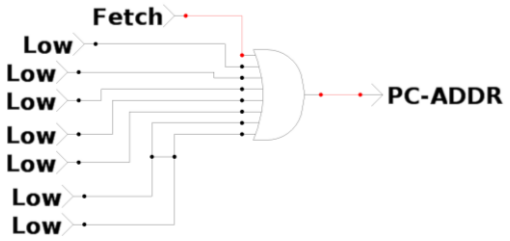


Designing and Implementing the Processor

Clocking the circuit

When clocking the circuit, the next instruction, denoted by the value of the program counter (PC), will be fetched from memory (RAM). $Fetch$ will be activated for one clock cycle.

Diagram: $Fetch$ clock cycle

Clock Cycle State	Program Counter	Memory (RAM) State	Active Control Signals
 *(page 1)*  *(page 3)*	 *(page 1)*  *(page 6)*	 *(page 2)*	 *(page 4)*  *(page 4)*

During the first $fetch$ clock cycle:

- the $Fetch$ control signal is $high$;
- the $program\ counter\ (PC)$ has a hexadecimal value of 00;
- the memory location $0x000$ is being accessed; and
- the control signals $Mem\ Read$ and $PC - ADDR$ are $high$.

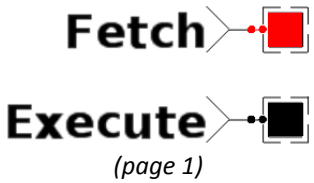
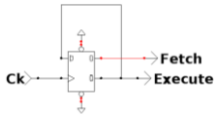
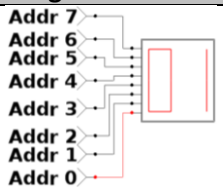
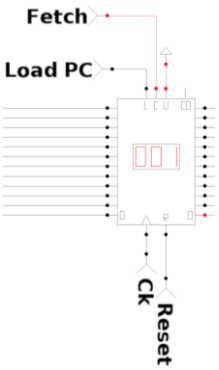
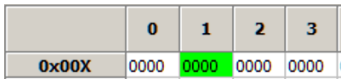
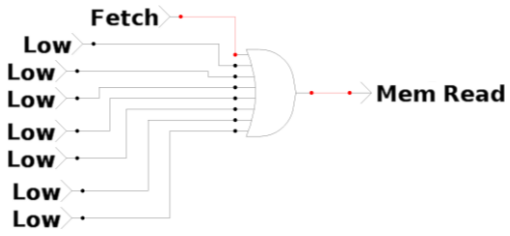
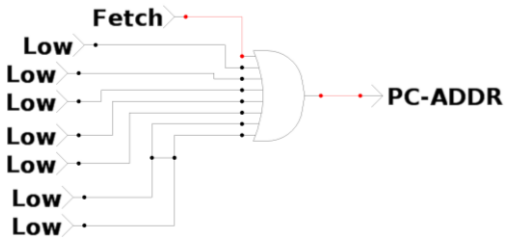
The $Fetch$ control signal activates the control signals $Mem\ Read$ and $PC - ADDR$:

- $Mem\ Read$ allows the instruction to be read from memory (RAM); and
- $PC - ADDR$ allows the current value of the program counter to be put on the address bus as this value is the memory address from which the instruction should be read.

On subsequent $fetch$ clock cycles, the value of the $program\ counter\ (PC)$ will be incremented sequentially, unless a jump instruction has been performed. This means that different memory locations will be accessed.

Designing and Implementing the Processor

Diagram: Subsequent $Fetch$ clock cycle

Clock Cycle State	Program Counter	Memory (RAM) State	Active Control Signals
 *(page 1)*  *(page 3)*	 *(page 1)*  *(page 6)*	 *(page 2)*	 *(page 4)*  *(page 4)*

The diagram shows an example of a subsequent $fetch$ clock cycle, where no jump instruction has taken place, in which:

- the $program\ counter\ (PC)$ has been incremented by one, from the hexadecimal value 00 to 01;
- and the memory location currently being accessed has also been incremented by one, from $0x000$ to $0x001$.

Writing programs in memory

After the $fetch$ clock cycle completes, $Execute$ will be activated for one clock cycle. This allows the instruction to be carried out.

The instructions for a given program are written in the memory (RAM). Each instruction is broken into two parts, the opcode and the operand. These instructions are described in the processor's instruction sets and the hexadecimal values from the instruction sets are used in memory (RAM).

Instruction Set: Instructions with 4-bit opcode and 12-bit operand

Binary Value	Hexadecimal Value	Operation	Remarks
1111 $XXXX\ XXXX\ XXXX$	**F**XXX	See below	Enables other formats
1110 $AAAA\ AAAA\ AAAA$	**E**AAA	JMP	Unconditional Jump
1101 $DDDD\ DDDD\ DDDD$	**D**DDD	$ADDI$	ADD immediate data $\text{Acc} = \text{Acc} + DDD$
1100 $DDDD\ DDDD\ DDDD$	**C**DDD	$MOVEI$	MOVE immediate data $\text{Acc} = DDD$
			X – Placeholder A – Address D – Data

Designing and Implementing the Processor

Instruction Set: Instructions with 8-bit opcode and 8-bit operand

Binary Value	Hexadecimal Value	Operation	Remarks
1111 1111 $XXXX\ XXXX$	**FF**XX	See below	Enables other formats
1111 1110 $RRRR\ RRRR$	**FE**RR	BZ	Branch if zero (Relative)

X – Placeholder
R – Relative Address

Example: Program written in memory

Write a program using the processor's instruction set such that it:

- **loads the accumulator with the value 3;**
- **decrements the accumulator; and**
- **go back to the first instruction.**

The program should continuously cycle through these instructions.

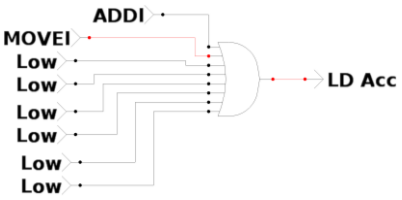
In order to load the accumulator with a value, the $MOVEI$ operation must be used. The hexadecimal value for this operation is **C**DDD.

Loading the accumulator with the hexadecimal value 3 will use the instruction with hexadecimal value $C003$ and this will be put into memory location $0x000$.

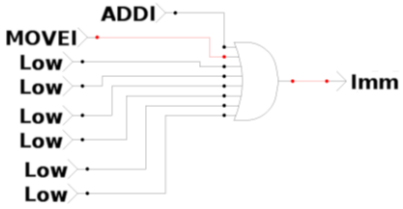
	0	1	2	3
0x00X	C003	0000	0000	0000

(page 2)

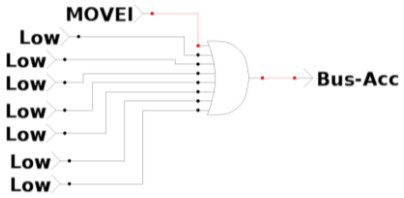
While this instruction is being executed, the control signals $LD\ Acc$, Imm and $Bus - Acc$ are activated.



(page 4)

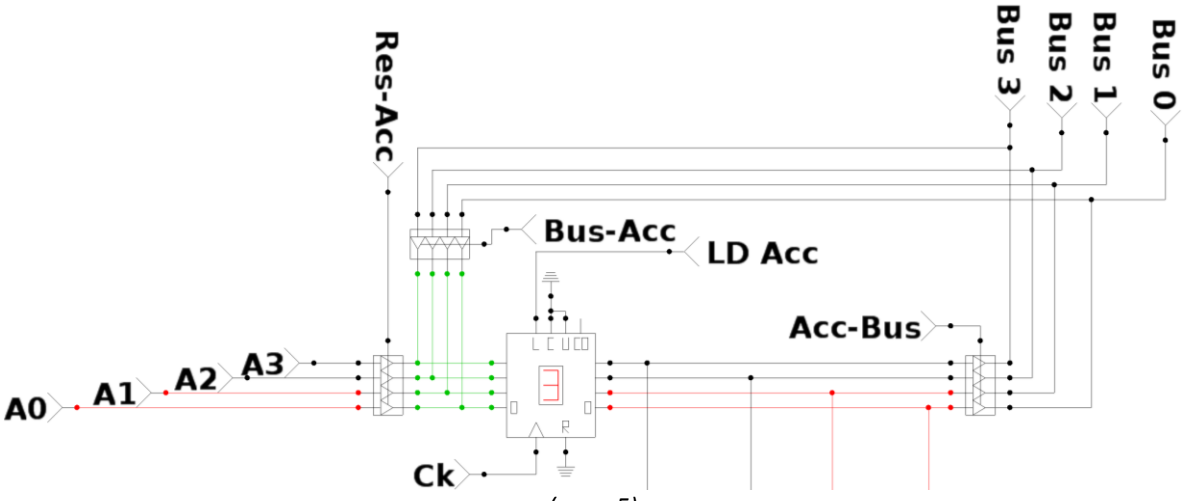


(page 4)



(page 4)

As a result of this instruction, the hexadecimal value 3 can be seen in the accumulator.



(page 5)

Load the accumulator with the hexadecimal value 3.

Designing and Implementing the Processor

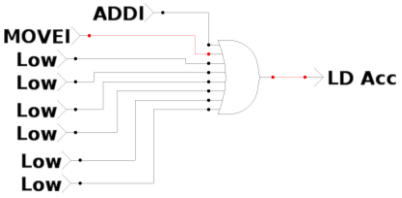
In order to increment or decrement the accumulator, the $ADDI$ instruction must be used. The hexadecimal value for this operation is $\mathbf{D}DDD$.

Decrementing the accumulator will require a subtraction operation. For this to be achieved, an addition of the negative number -1 must be performed such that $accumulator\ contents = accumulator\ contents - 1$. As a result, this operaion will use the instruction with hexadecimal value $DFFF$ and this will be put into memory location $0x001$.

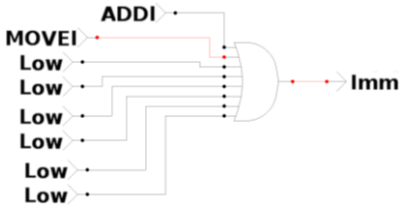
	0	1	2	3
0x00X	C003	DFFF	0000	0000

(page 2)

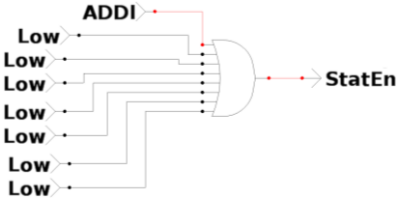
While this instruction is being executed, the control signals $LD\ Acc$, Imm, $StatEn$ and $Res - Acc$ are activated.



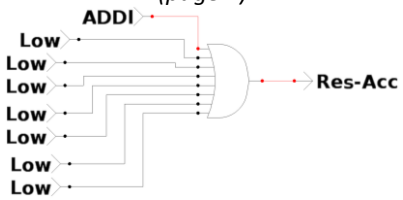
(page 4)



(page 4)



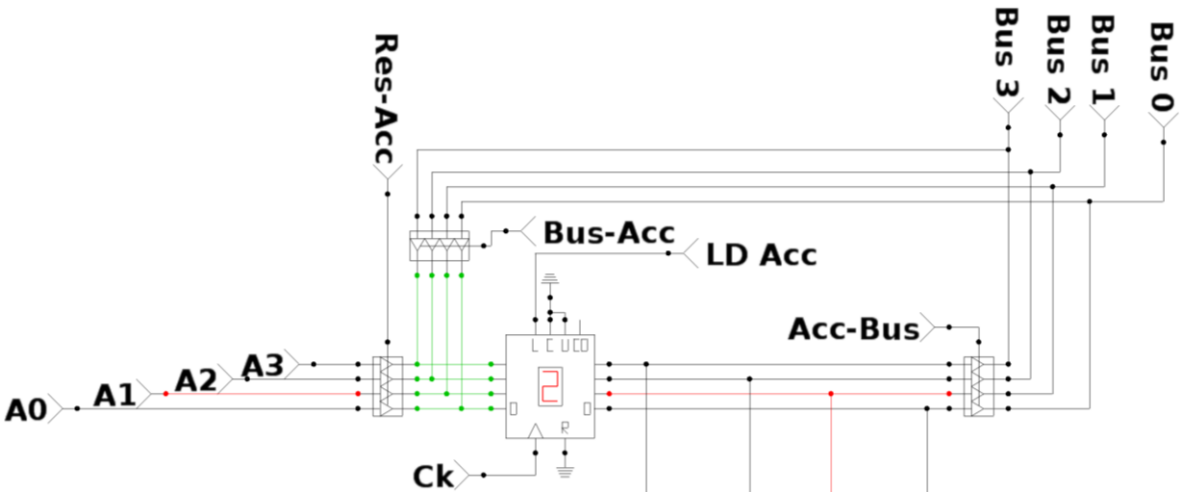
(page 4)



(page 4)

Decrement the accumulator.

As a result of this instruction, the value in the accumulator has been decremented.



(page 5)

Designing and Implementing the Processor

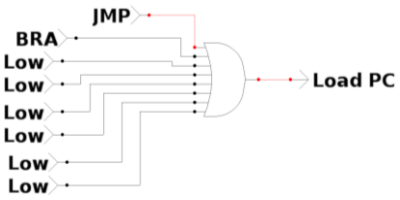
In order to go back to the first instruction an unconditional jump must be performed.

An unconditional jump back to zero will use the operation with hexadecimal value $E000$ and this will be put into memory location $0x002$.

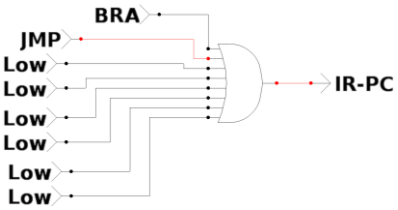
	0	1	2	3
0x00X	C003	DFFF	E000	0000

(page 2)

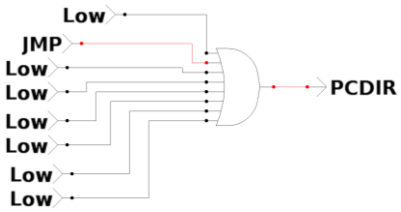
While this instruction is being executed, the control signals $LD\ Acc$, Imm, $StatEn$ and $Res - Acc$ are activated.



(page 4)



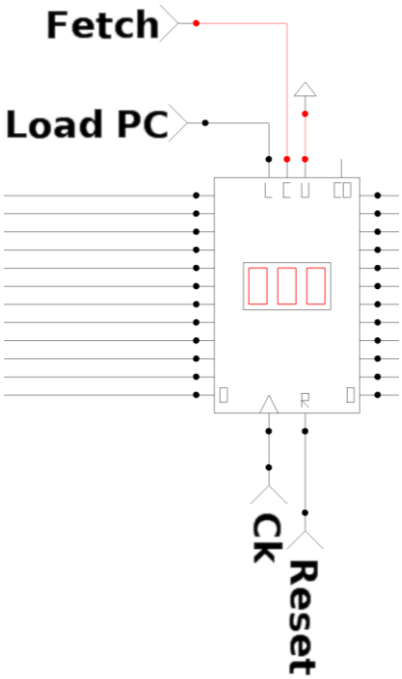
(page 4)



(page 4)

As a result of this instruction, the $program\ counter$ (PC) is set to 000 and the memory location accessed on the next clock cycle is $0x000$.

Go back to the first instruction.



(page 6)

Final program

	0	1	2	3
0x00X	C003	DFFF	E000	0000

(page 2)

Designing and Implementing the Processor

Example: Program written in memory

Write a program using the processor's instruction set such that it:

- **loads the accumulator with the value 3;**
- **decrements the accumulator until its value is 0.**

The program should continuously cycle through these instructions.

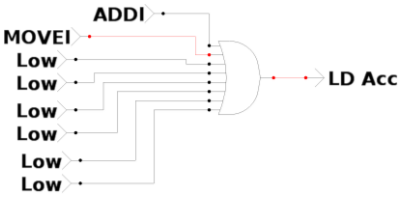
In order to load the accumulator with a value, the $MOVEI$ instruction must be used. The hexadecimal value for this operation is $\mathbf{C}DDD$.

Loading the accumulator with the hexadecimal value 3 will use the instruction with hexadecimal value $C003$ and this will be put into memory location $0x000$.

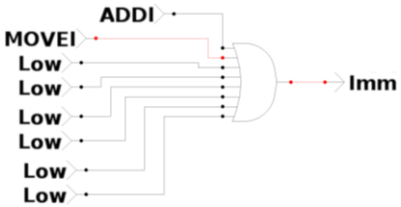
	0	1	2	3	4
0x00X	C003	0000	0000	0000	0000

(page 2)

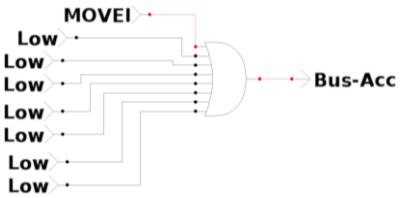
While this instruction is being executed, the control signals $LD\ Acc$, Imm and $Bus - Acc$ are activated.



(page 4)



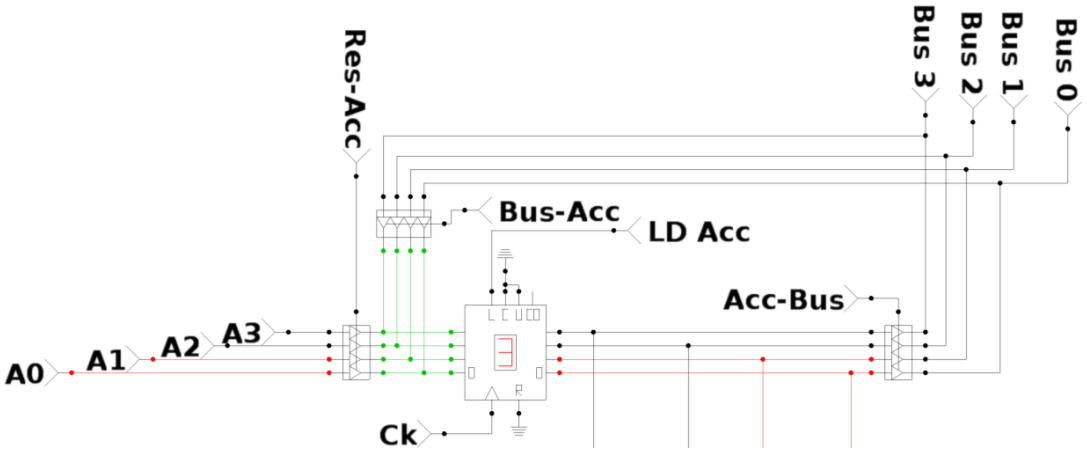
(page 4)



(page 4)

Load the accumulator with the value 3.

As a result of this instruction, the hexadecimal value 3 can be seen in the accumulator.



(page 5)

In order to increment or decrement the accumulator, the $ADDI$ instruction must be used. The hexadecimal value for this operation is $\mathbf{D}DDD$.

Decrementing the accumulator will require a subtraction operation. For this to be achieved, an addition of the negative number -1 must be performed such that $accumulator\ contents = accumulator\ contents - 1$. As a result, this operaion will use the instruction with hexadecimal value $DFFF$ and this will be put into memory location $0x001$.

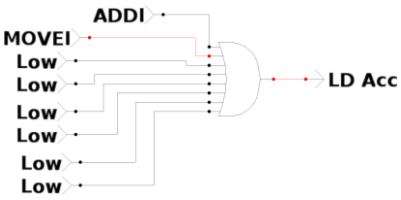
Decrement the accumulator until its value is 0.

	0	1	2	3	4
0x00X	C003	DFFF	0000	0000	0000

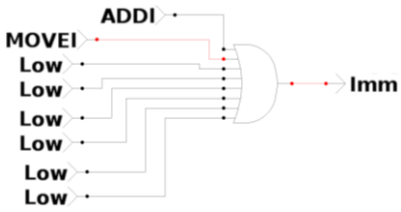
(page 2)

Designing and Implementing the Processor

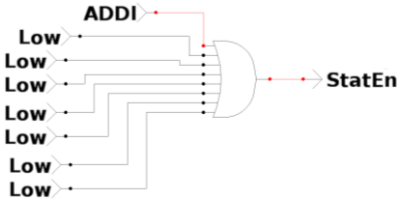
While this instruction is being executed, the control signals $LD\ Acc$, Imm, $StatEn$ and $Res - Acc$ are activated.



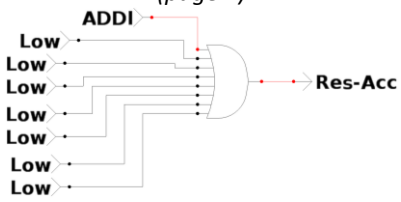
(page 4)



(page 4)

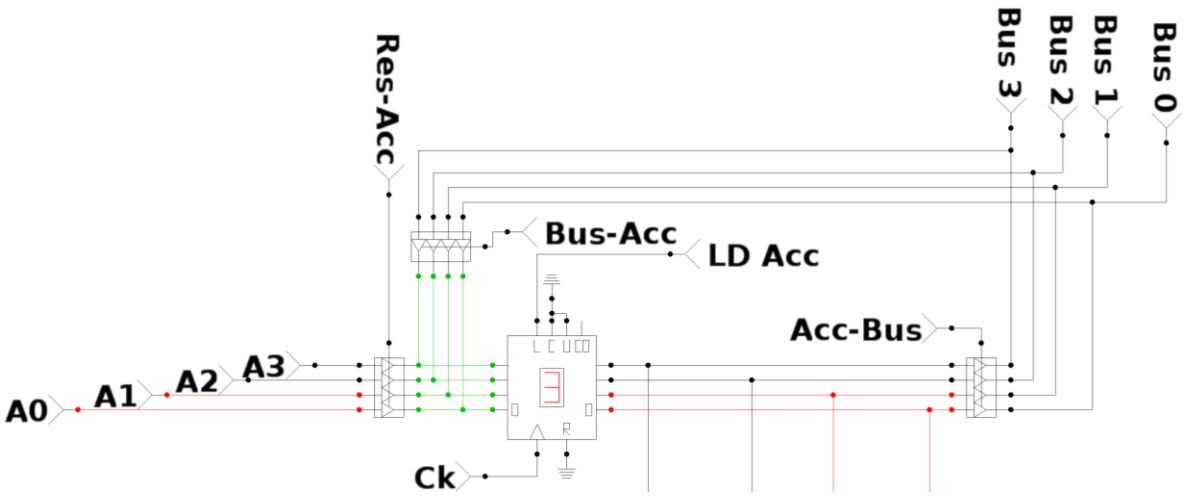


(page 4)

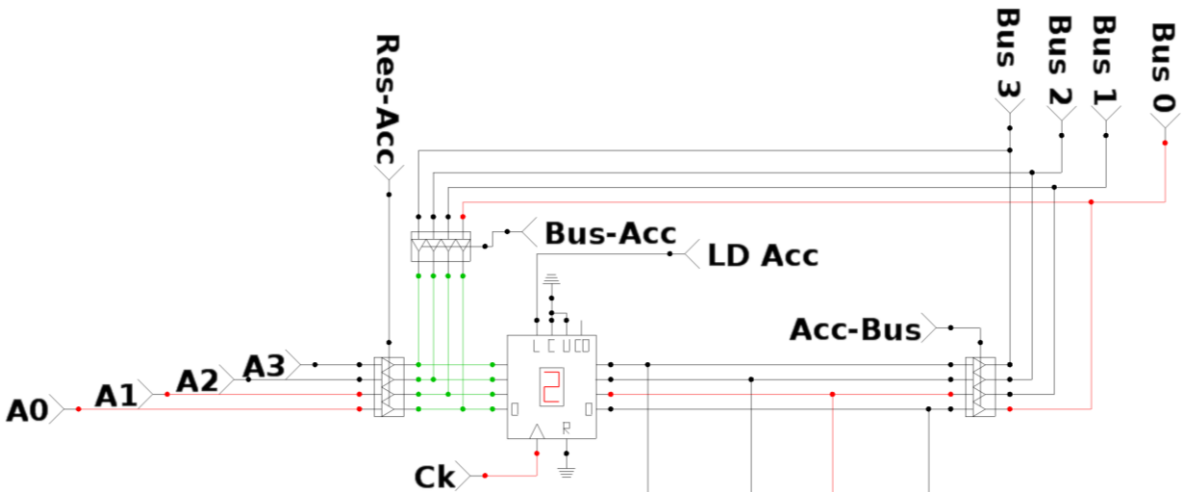


(page 4)

As a result of this instruction, the value in the accumulator has been decremented.



(page 5)



(page 5)

In order to continue decrementing the accumulator until its value is 0, a series of branch instructions must be implemented so that the program will:

- perform the decrement instruction repeatedly until the accumulator has a value of 0; and
- load the accumulator with the value 3 once the accumulator has value a 0.

Decrement the accumulator until its value is 0 (continued).

Designing and Implementing the Processor

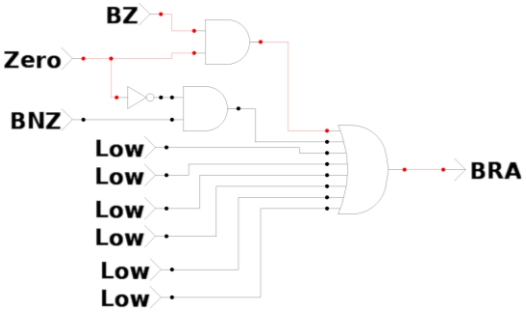
In order to perform a branch if the contents of the accumulator is 0, the BZ instruction must be used. The hexadecimal value for this instruction is $\mathbf{FE}RR$.

The branch instruction responsible for returning to the first instruction, which is responsible for loading the value 3 into the accumulator, will be present in memory location $0x002$. The BZ instruction uses a relative address (RR) as the operand and the jump from memory location $0x002$ to $0x004$ is 1 – this is because the program counter (PC) points to the next instruction, which would be located at $0x003$, and a further increment of 1 is required to reach $0x004$. As a result, this operaion will use the instruction with hexadecimal value $FE01$ and this will be put into memory location $0x002$.

	0	1	2	3	4
0x00X	C003	DFFF	FE01	0000	0000

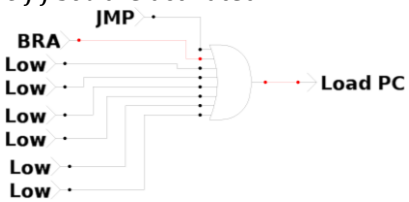
(page 2)

While this instruction is being executed, the control signal BRA is activated.

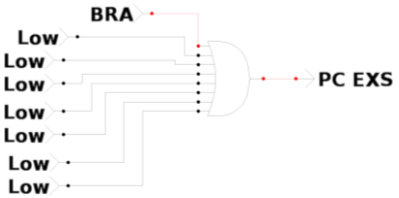


(page 4)

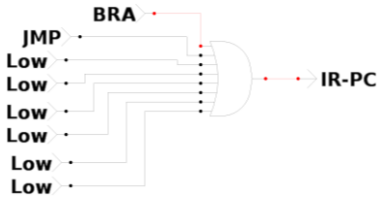
As a result of the control signal BRA being active, the control signals $Load\ PC$, $PC\ EXS$, $IR - PC$ and $Offset$ are activated.



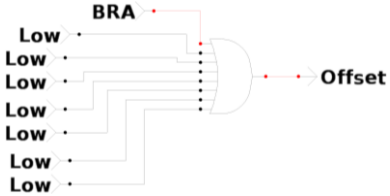
(page 4)



(page 4)



(page 4)



(page 4)

In order to perform a branch back to the second instruction, which is responsible for decrementing the accumumator, the $JUMP$ instruction must be used. The hexadecimal value for this instruction is $\mathbf{E}AAA$.

Performing an unconditional jump back to the instruction located in the memory location $0x001$ will use the instruction with hexadecimal value $E001$ and this will be put into memory location $0x003$. The placement of this instruction in memory relative to the other instructions in memory is important as it must be after the BZ instruction, so that it is not executed when the contents of the accumulator is 0 as placing it before the BZ instruction would not give the unconditional jump opportunity to execute.

	0	1	2	3	4
0x00X	C003	DFFF	FE01	E001	0000

(page 2)

Decrement the accumulator until its value is 0 (continued).

Designing and Implementing the Processor

While this instruction is being executed, the control signals $Load\ PC$, $IR - PC$ and $PCDIR$ are activated.

(page 4) *(page 4)* *(page 4)*

If the contents of the accumulator is not 0, the $FE01$ instruction will not be executed and the $E001$ instruction will be executed. As a result of this instruction, the program will branch to the instruction $DFFF$ after the accumulator has been decremented such that it will be decremented until it reaches the value of 0.

Once the $E001$ instruction has been executed, the program will branch back to the $DFFF$ instruction.

	0	1	2	3	4
0x00X	C003	DFFF	FE01	E001	0000

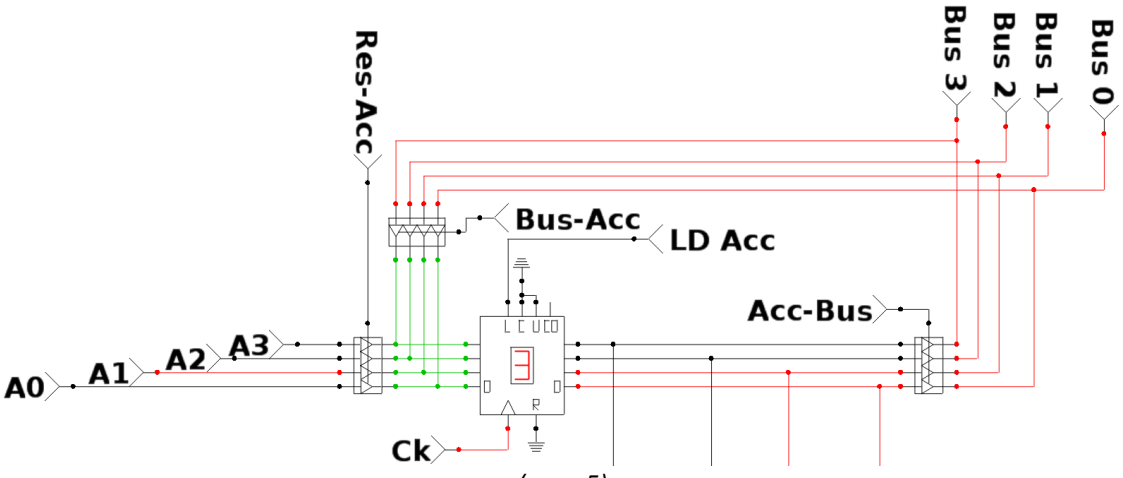
(page 2)

	0	1	2	3	4
0x00X	C003	DFFF	FE01	E001	0000

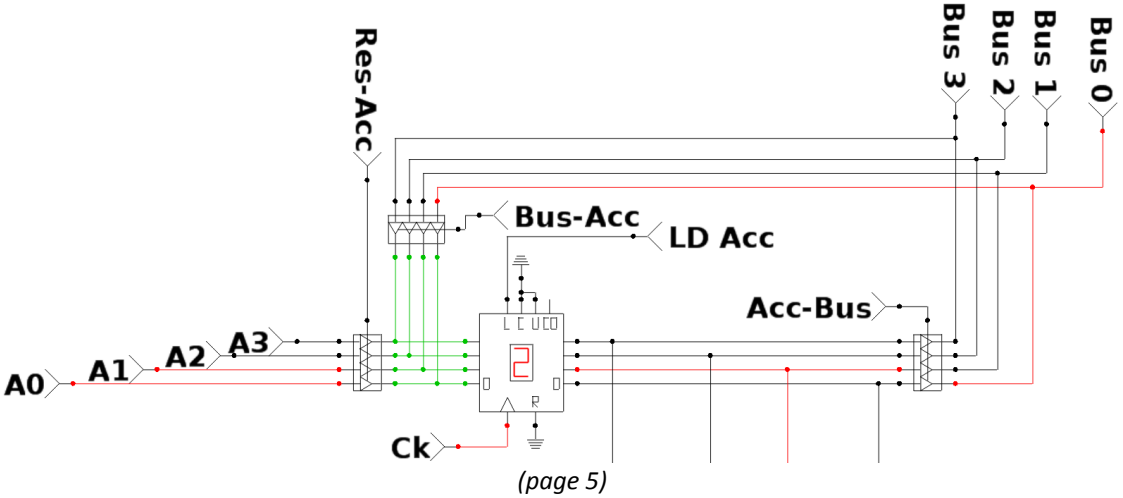
(page 2)

This will continue until the value of the accumulator is 0.

Decrement the accumulator until its value is 0 (continued).

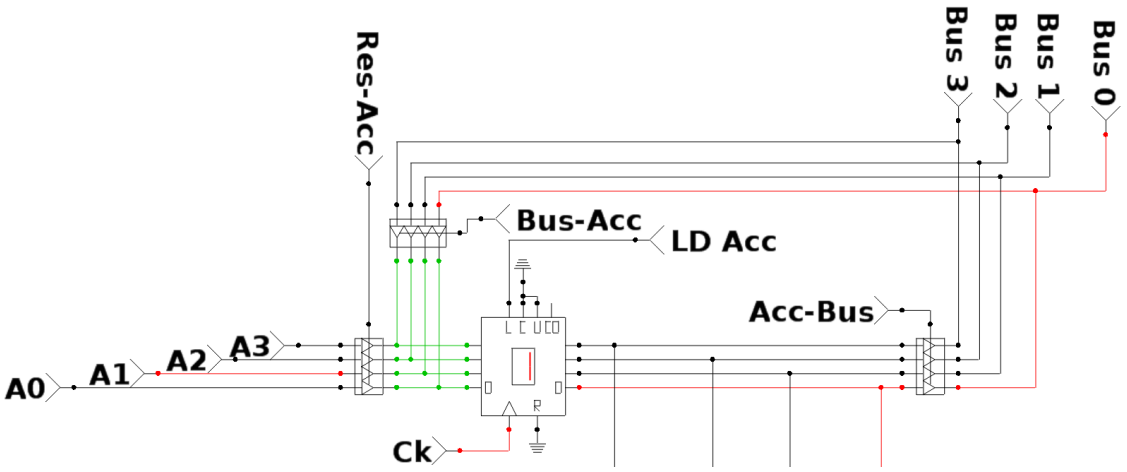


(page 5)

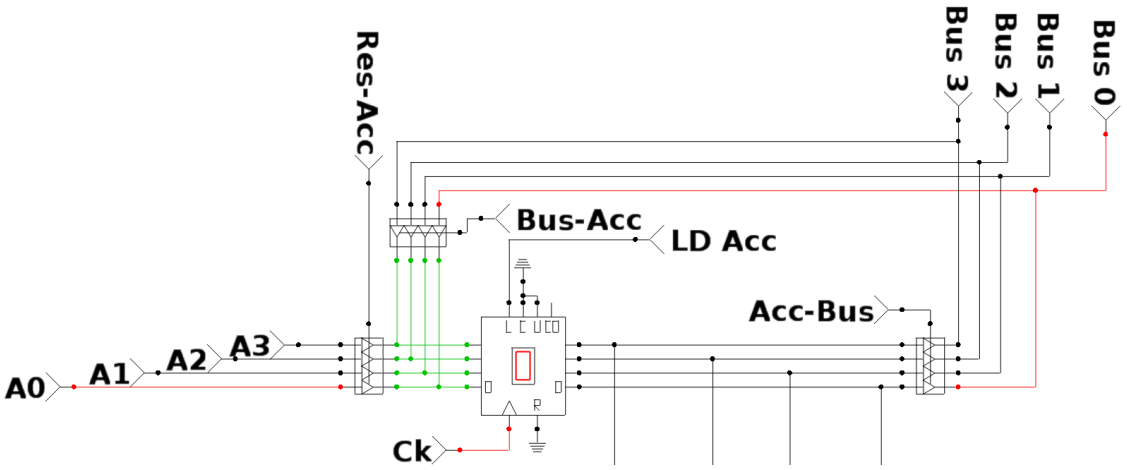


(page 5)

Designing and Implementing the Processor



(page 5)



(page 5)

Decrement the accumulator until its value is 0 (continued).

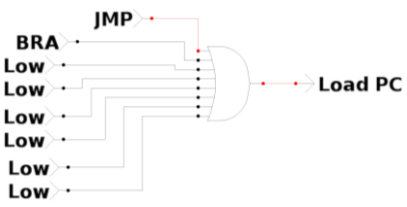
In order to perform a branch back to the second instruction, which is responsible for decrementing the accumumator, the $JUMP$ instruction must be used. The hexadecimal value for this instruction is $\mathbf{E}AAA$.

Performing an unconditional jump back to the instruction located in the memory location $0x001$ will use the instruction with hexadecimal value $E001$ and this will be put into memory location $0x003$. The placement of this instruction in memory relative to the other instructions in memory is important as it must be after the BZ instruction, so that it is not executed when the contents of the accumulator is 0 as placing it before the BZ instruction would not give the unconditional jump opportunity to execute.

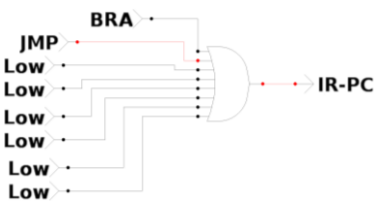
	0	1	2	3	4
0x00X	C003	DFFF	FE01	E001	0000

(page 2)

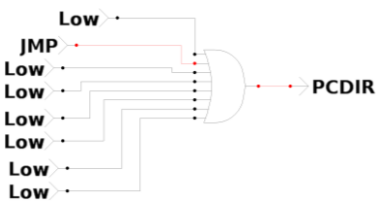
While this instruction is being executed, the control signals $Load\ PC$, $IR - PC$ and $PCDIR$ are activated.



(page 4)



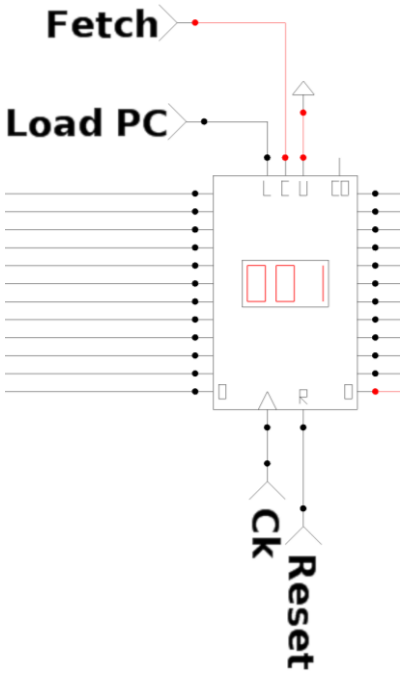
(page 4)



(page 4)

As a result of this instruction, the $program\ counter$ (PC) is set to 001 and the memory location accessed on the next clock cycle is $0x001$.

Designing and Implementing the Processor



(page 6)

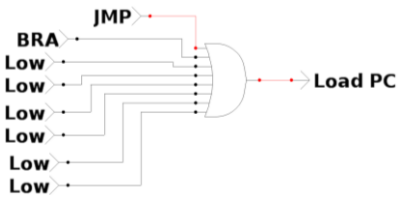
In order to perform a branch back to the first instruction, which is responsible for loading the accumulator with the value 3, the $JUMP$ instruction must be used. The hexadecimal value for this instruction is $\mathbf{E}AAA$.

Performing an unconditional jump back to the instruction located in the memory location $0x000$ will use the instruction with hexadecimal value $E000$ and this will be put into memory location $0x004$. The placement of this instruction in memory relative to the other instructions in memory is important as it must be after the previous $JUMP$ instruction, so that the BZ instruction branches to the correct memory location. The BZ instruction uses the relative jump of 1; this means that it will branch to the instruction two locations after itself.

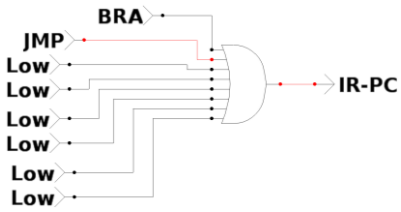
	0	1	2	3	4
0x00X	C003	DFFF	FE01	E001	E000

(page 2)

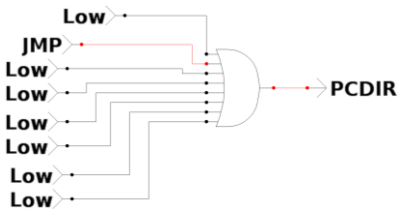
While this instruction is being executed, the control signals $LD\ Acc$, Imm, $StatEn$ and $Res - Acc$ are activated.



(page 4)



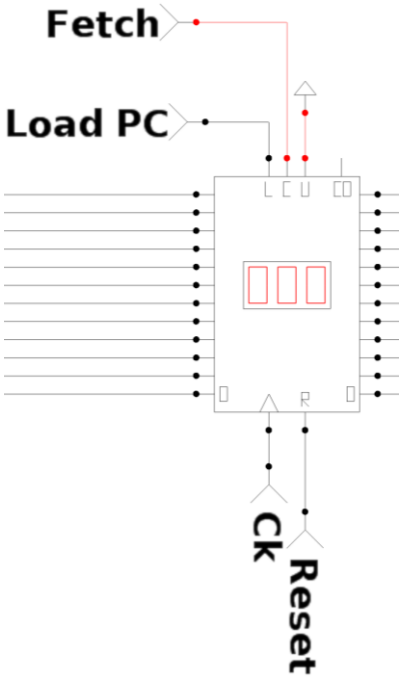
(page 4)



(page 4)

As a result of this instruction, the $program\ counter$ (PC) is set to 000 and the memory location accessed on the next clock cycle is $0x000$.

Decrement the accumulator until its value is 0 (continued).



(page 6)

Decrement the accumulator until its value is 0 (continued).

Final program

	0	1	2	3	4
0x00X	C003	DFFF	FE01	E001	E000

(page 2)

Designing and Implementing the Processor

Implementing an instruction

There are currently many available instructions in the processor's instruction sets as there are "free" spaces available for instructions on the decoders in the processor.

Process: Implementing an instruction

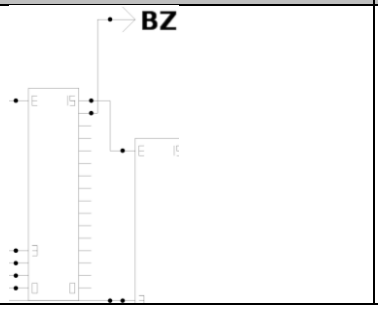
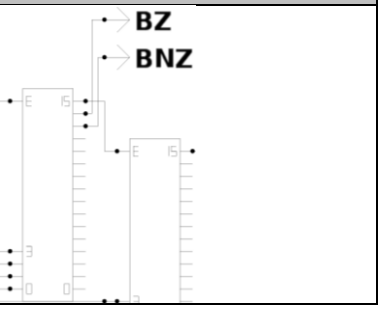
1) Add the control signal for the instruction as a label to the appropriate decoder (page 3).
2) Create the required logic for the instruction's control signal to be activated.
3) Using the general logic strategy, add the logic for the instruction's control signal to the necessary control signals on the instruction decode circuits (page 4).

Branch Not Zero (BNZ)

Branch Not Zero (BNZ) will perform a conditional jump if the content of the accumulator is any non-zero value.

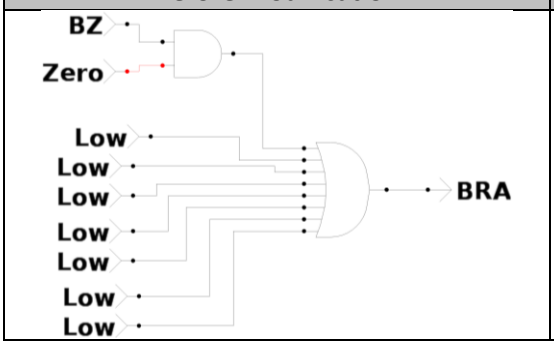
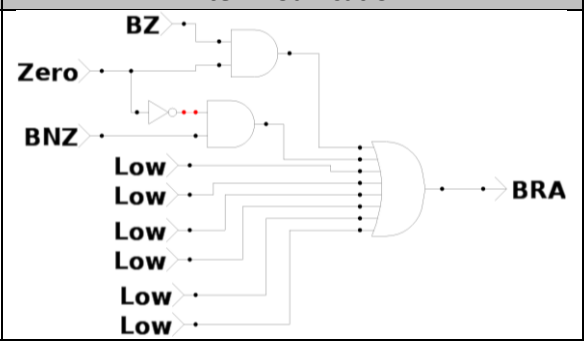
Example: Implementing the BNZ instruction

A label can be attached to an unused output on the second decoder (page 3). The second decoder is used as the instruction BNZ will make use of the instruction set with a 8-bit opcode and a 8-bit operand.

Before Modification	After Modification
	

Add the control signal for the instruction as a label to the appropriate decoder (page 3).

The BRA control signal must be active when BNZ is $high$ and $Zero$ is low. This logic can be achieved using the logic $BRA = BNZ \,.\, \overline{Zero}$.

Before Modification	After Modification
	

Create the required logic for the instruction's control signal to be activated.

Using the general logic strategy, add the logic for the instruction's control signal to the necessary control signals on the instruction decode circuits (page 4).

Designing and Implementing the Processor

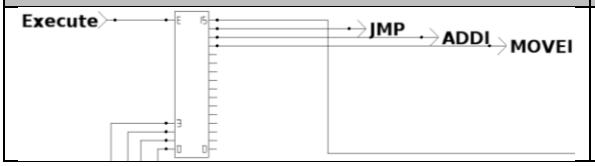
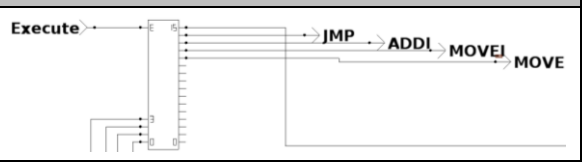
Move ($MOVE$)

Move ($MOVE$) will load the accumulator with the value of a given memory location.

Example: Implementing the $MOVE$ instruction

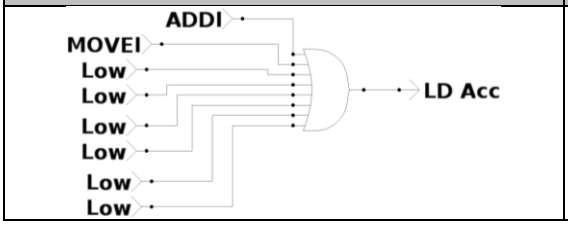
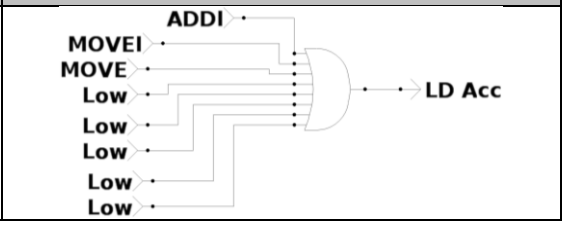
A label can be attached to an unused output on the first decoder (page 3). The first decoder is used as the instruction $MOVE$ will make use of the instruction set with a 4-bit opcode and a 12-bit operand.

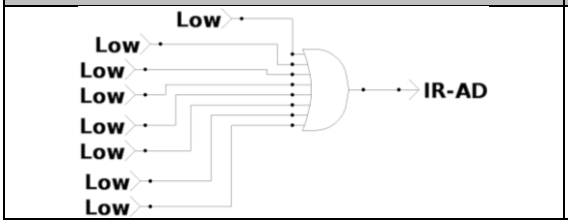
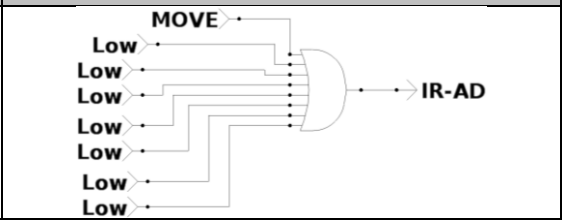
Add the control signal for the instruction as a label to the appropriate decoder (page 3).

Before Modification	After Modification
	

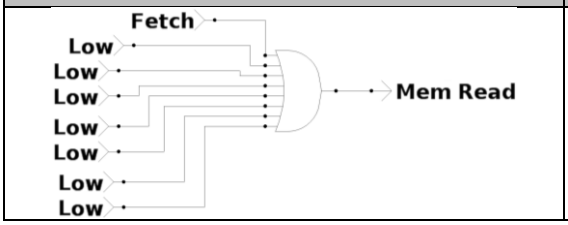
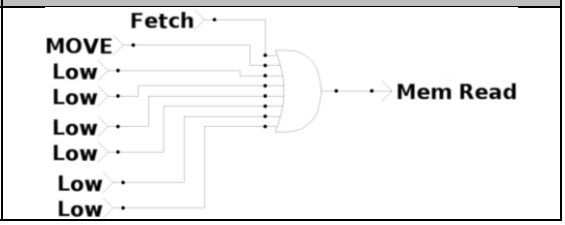
Create the required logic for the instruction's control signal to be activated.

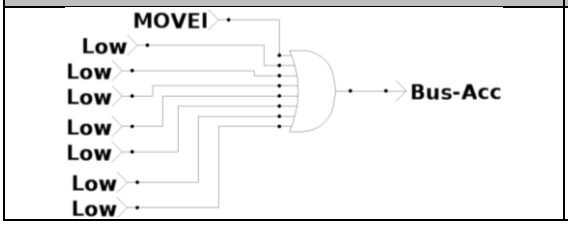
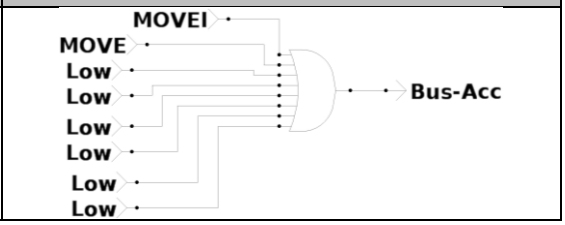
The $LD\ Acc$, $IR-AD$, $Mem\ Read$ and $Bus-Acc$ control signals must be active when $MOVE$ is $high$.

Before Modification	After Modification
	

Before Modification	After Modification
	

Using the general logic strategy, add the logic for the instruction's control signal to the necessary control signals on the instruction decode circuits (page 4).

Before Modification	After Modification
	

Before Modification	After Modification
	

Designing and Implementing the Processor

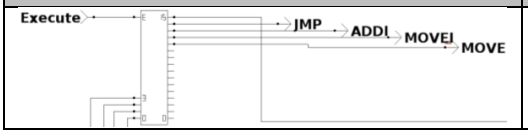
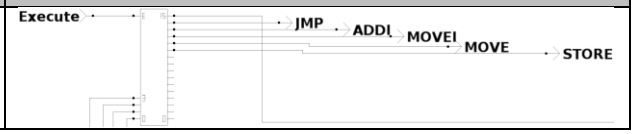
Store ($STORE$)

Store ($STORE$) will store the contents of the accumulator in a given memory location.

Example: Implementing the $STORE$ instruction

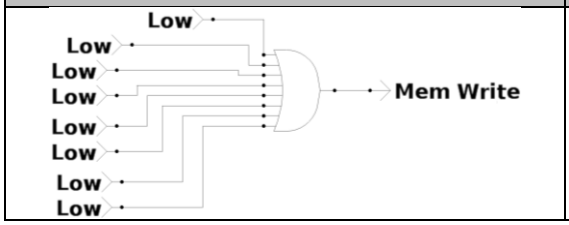
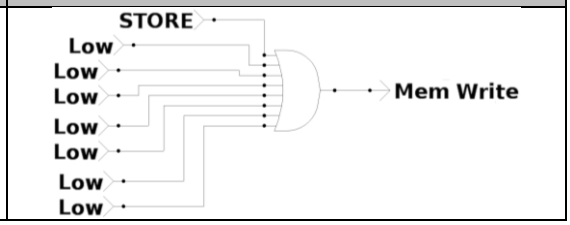
A label can be attached to an unused output on the first decoder (page 3). The first decoder is used as the instruction $STORE$ will make use of the instruction set with a 4-bit opcode and a 12-bit operand.

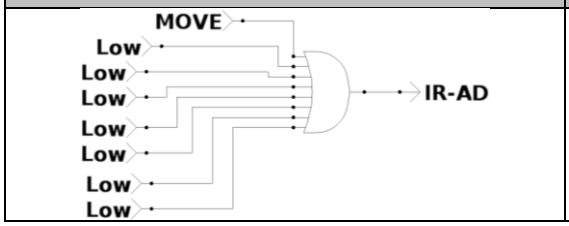
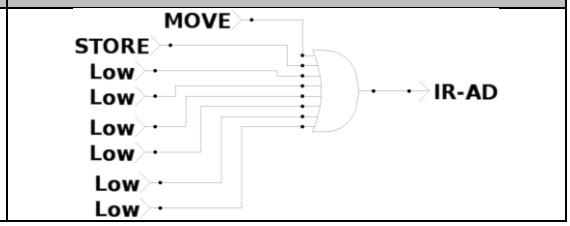
Add the control signal for the instruction as a label to the appropriate decoder (page 3).

Before Modification	After Modification
	

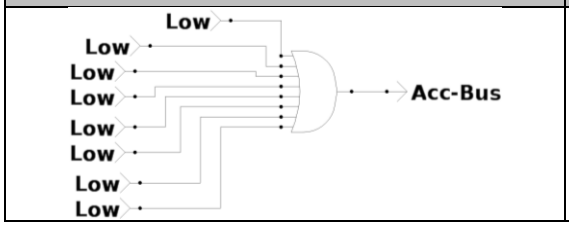
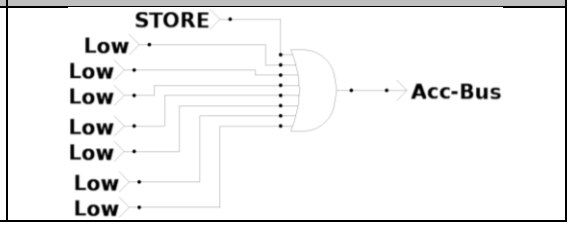
Create the required logic for the instruction's control signal to be activated.

The $Mem\ Write$, $IR - AD$ and $Acc - Bus$ control signals must be active when $STORE$ is $high$.

Before Modification	After Modification
	

Before Modification	After Modification
	

Using the general logic strategy, add the logic for the instruction's control signal to the necessary control signals on the instruction decode circuits (page 4).

Before Modification	After Modification
	

Designing and Implementing the Processor

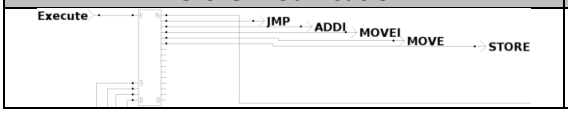
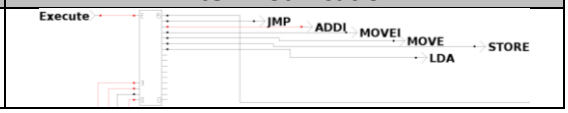
Load (LDA)

Load (LDA) will load an immediate value into the address register.

Example: Implementing the LDA instruction

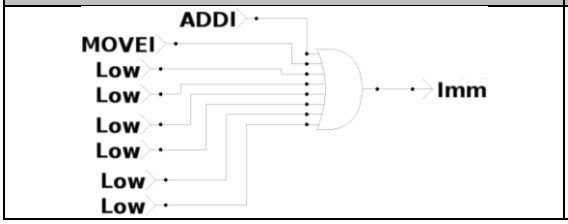
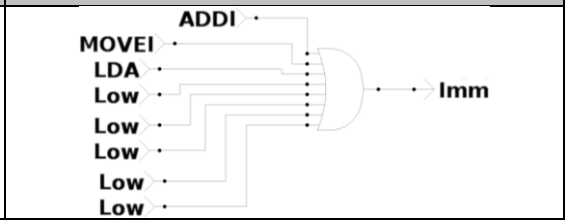
A label can be attached to an unused output on the first decoder (page 3). The first decoder is used as the instruction LDA will make use of the instruction set with a 4-bit opcode and a 12-bit operand.

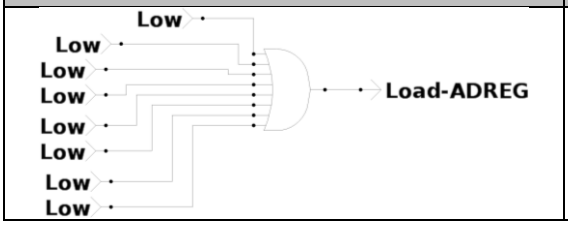
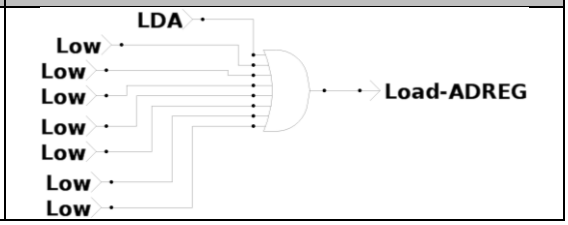
Add the control signal for the instruction as a label to the appropriate decoder (page 3).

Before Modification	After Modification
	

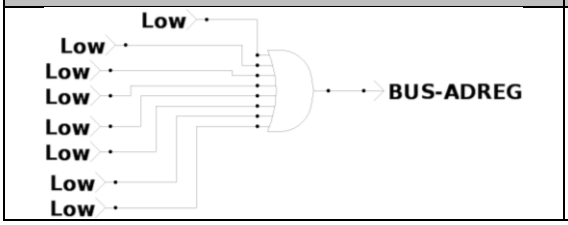
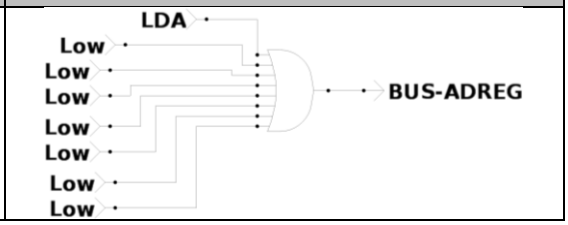
The $Imm, Load - ADREG$ and $BUS - ADREG$ control signals must be active when LDA is $high$.

Create the required logic for the instruction's control signal to be activated.

Before Modification	After Modification
	

Before Modification	After Modification
	

Using the general logic strategy, add the logic for the instruction's control signal to the necessary control signals on the instruction decode circuits (page 4).

Before Modification	After Modification
	

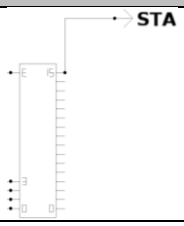
Designing and Implementing the Processor

Store at address (STA)

Store at address (STA) will store the contents of the accumulator at a memory location specified by the contents.

Example: Implementing the STA instruction

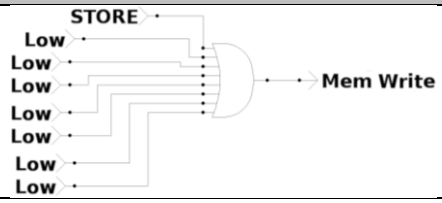
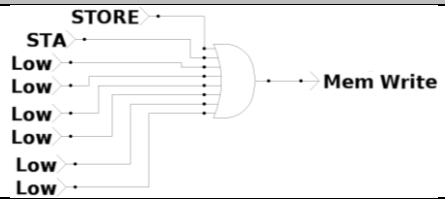
A label can be attached to an unused output on the last decoder (page 3). The last decoder is used as the instruction STA will make use of the instruction set with a 16-bit opcode and no operand.

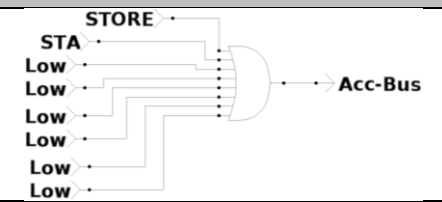
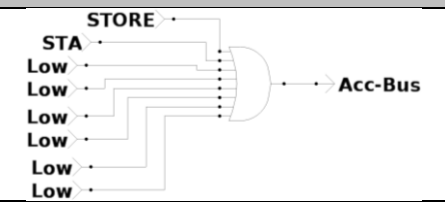
Before Modification	After Modification
	

Add the control signal for the instruction as a label to the appropriate decoder (page 3).

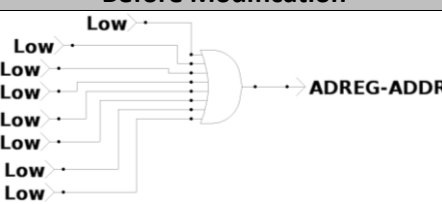
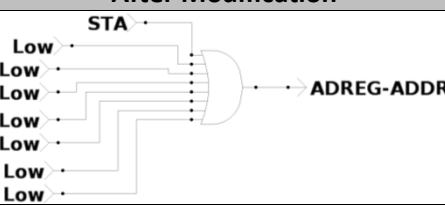
The $Mem\ Write$, $Acc - Bus$ and $ADRED - ADDR$ control signals must be active when STA is $high$.

Create the required logic for the instruction's control signal to be activated.

Before Modification	After Modification
	

Before Modification	After Modification
	

Using the general logic strategy, add the logic for the instruction's control signal to the necessary control signals on the instruction decode circuits (page 4).

Before Modification	After Modification
	

Designing and Implementing the Processor

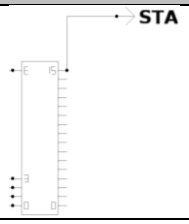
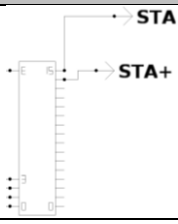
Store at address with auto-increment ($STA +$)

Store at address with auto-increment ($STA +$) will store the contents of the accumulator at a memory location specified by the contents.

Example: Implementing the $STA +$ instruction

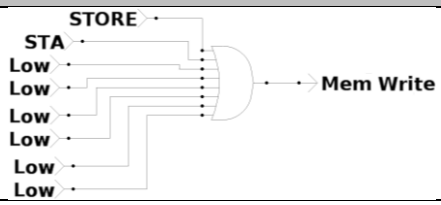
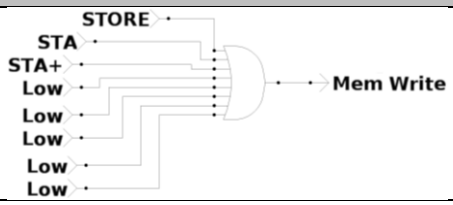
A label can be attached to an unused output on the last decoder (page 3). The last decoder is used as the instruction $STA +$ will make use of the instruction set with a 16-bit opcode and no operand.

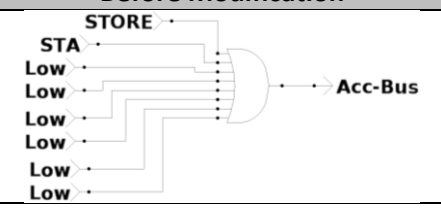
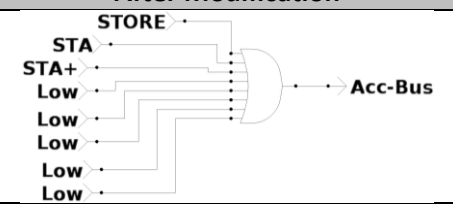
Add the control signal for the instruction as a label to the appropriate decoder (page 3).

Before Modification	After Modification
	

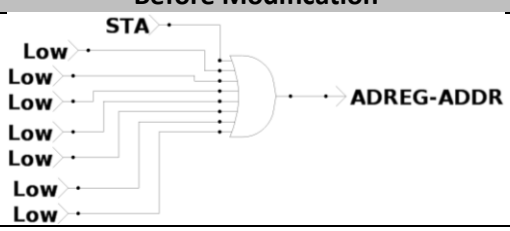
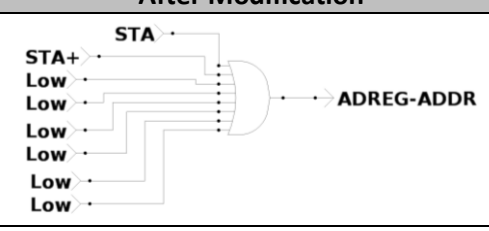
The $Mem\ Write$, $Acc - Bus$, $ADRED - ADDR$ and $INC - ADREG$ control signals must be active when $STA +$ is $high$.

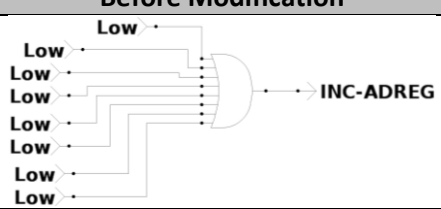
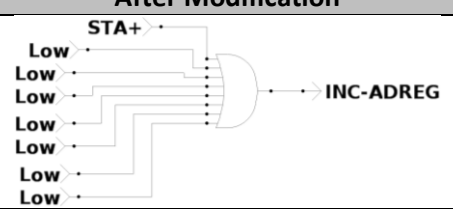
Create the required logic for the instruction's control signal to be activated.

Before Modification	After Modification
	

Before Modification	After Modification
	

Using the general logic strategy, add the logic for the instruction's control signal to the necessary control signals on the instruction decode circuits (page 4).

Before Modification	After Modification
	

Before Modification	After Modification
	

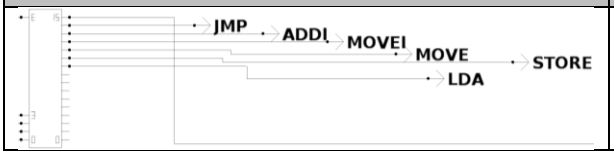
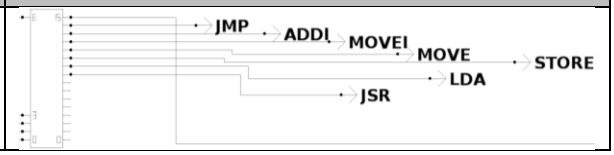
Designing and Implementing the Processor

Jump SubRoutine (JSR)

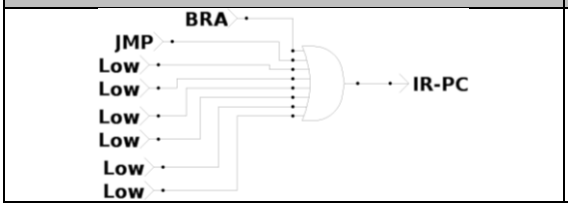
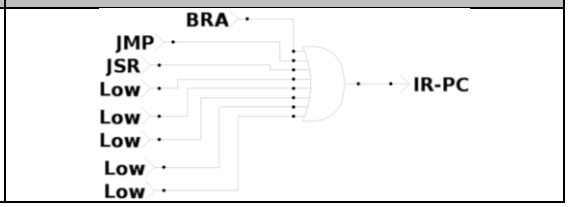
Jump SubRoutine (JSR) will load the program counter from a 12-bit immediate value in the instruction, pre-decrement the program counter and use the decremented value of the program counter as a memory address to store the current value of the program counter.

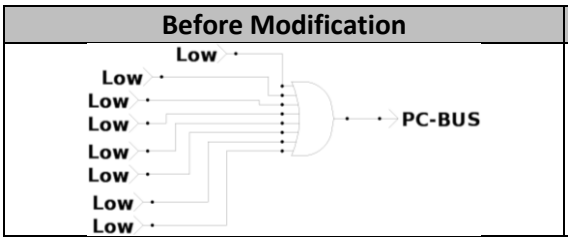
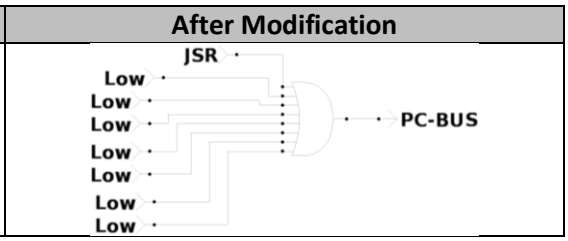
Example: Implementing the JSR instruction

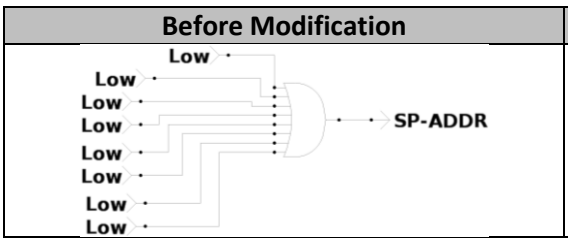
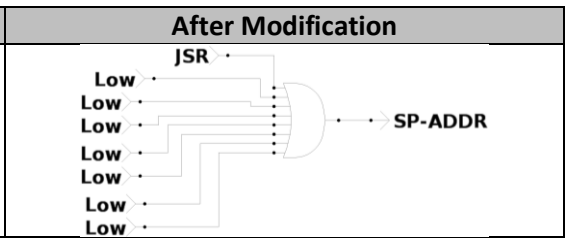
A label can be attached to an unused output on the first decoder (page 3). The first decoder is used as the instruction JSR will make use of the instruction set with a 4-bit opcode and a 12-bit operand.

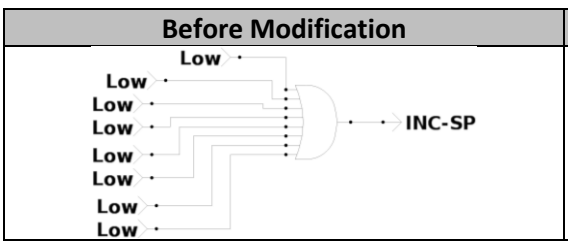
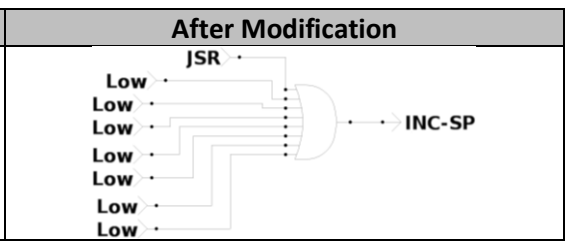
Before Modification	After Modification
	

The $IR - PC$, $PC - BUS$, $SP - ADDR$, $INC - SP$, $SP - DEC$, $PCDIR$, $Mem\ Write$ and $Load\ PC$ control signals must be active when JSR is $high$.

Before Modification	After Modification
	

Before Modification	After Modification
	

Before Modification	After Modification
	

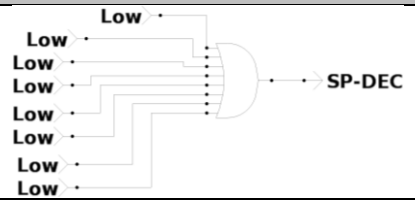
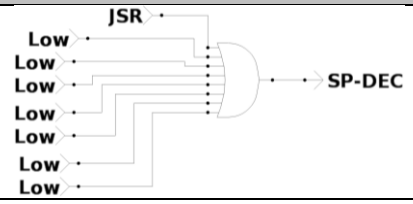
Before Modification	After Modification
	

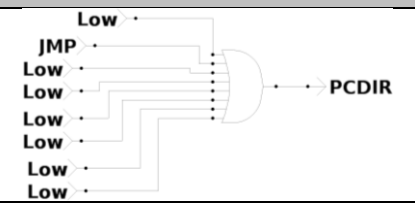
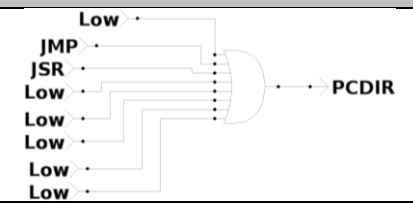
Add the control signal for the instruction as a label to the appropriate decoder (page 3).

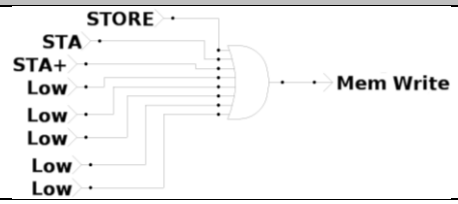
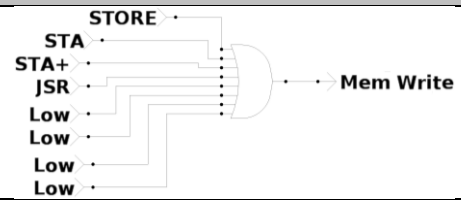
Create the required logic for the instruction's control signal to be activated.

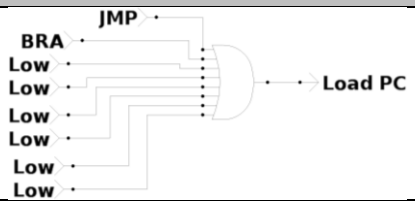
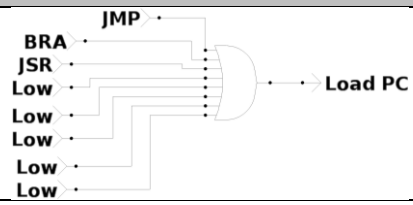
Using the general logic strategy, add the logic for the instruction's control signal to the necessary control signals on the instruction decode circuits (page 4).

Designing and Implementing the Processor

Before Modification	After Modification
	

Before Modification	After Modification
	

Before Modification	After Modification
	

Before Modification	After Modification
	

Using the general logic strategy, add the logic for the instruction's control signal to the necessary control signals on the instruction decode circuits (page 4).

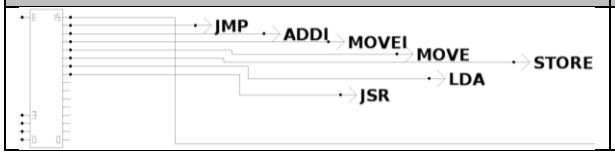
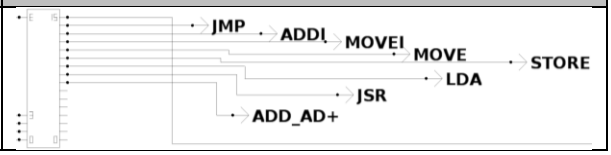
Designing and Implementing the Processor

Add address register with auto-increment (ADD_AD+)

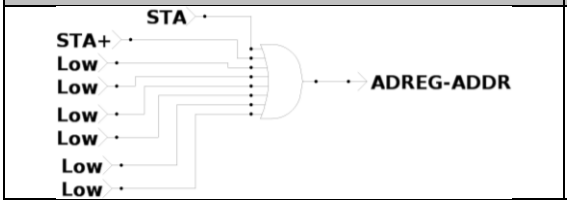
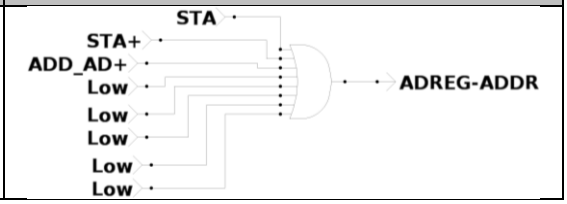
Add address register with auto-increment (ADD_AD+) will add to the accumulator a value taken from a memory location that is specified by the address held in the address register and increments the value in the address register.

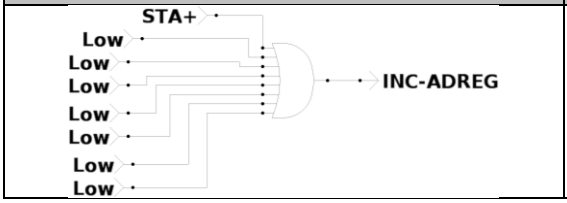
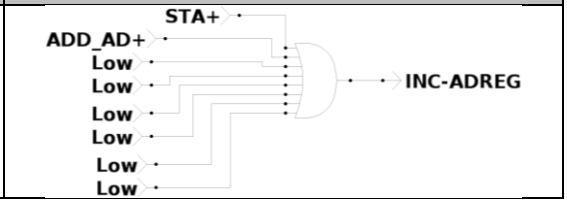
Example: Implementing the ADD_AD+ instruction

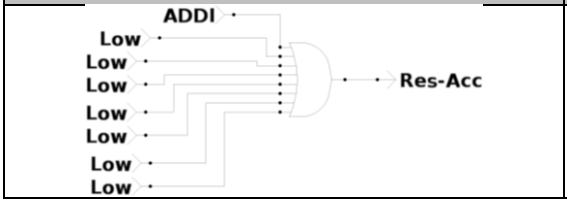
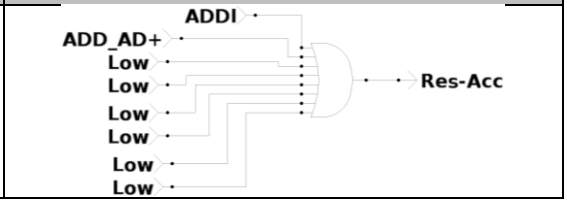
A label can be attached to an unused output on the last decoder (page 3). The last decoder is used as the instruction $STA+$ will make use of the instruction set with a 16-bit opcode and no operand.

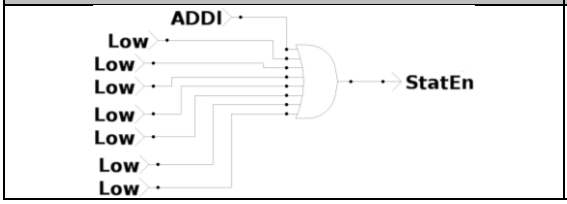
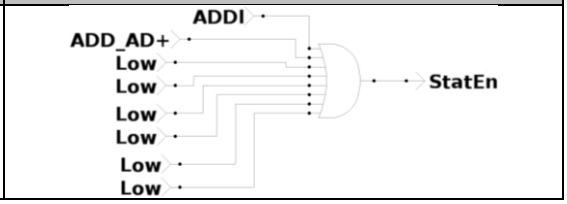
Before Modification	After Modification
	

The $ADREG-ADDR$, $INC-ADREG$, $Res-Acc$, $StatEn$, $LD\ Acc$ and $Mem\ Read$ control signals must be active when ADD_AD+ is $high$.

Before Modification	After Modification
	

Before Modification	After Modification
	

Before Modification	After Modification
	

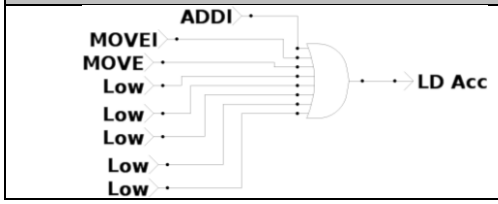
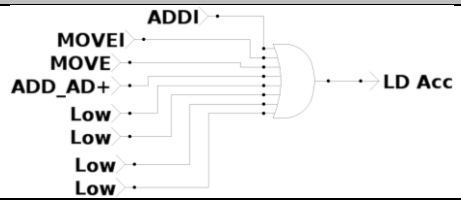
Before Modification	After Modification
	

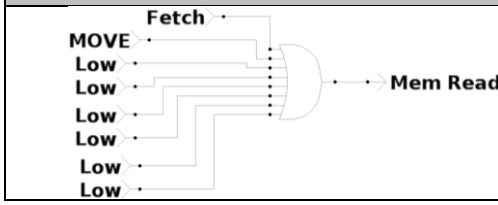
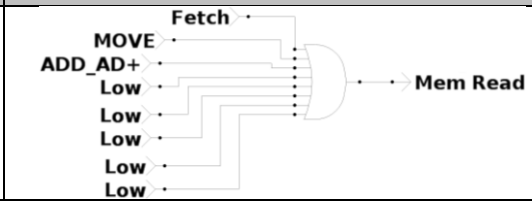
Add the control signal for the instruction as a label to the appropriate decoder (page 3).

Create the required logic for the instruction's control signal to be activated.

Using the general logic strategy, add the logic for the instruction's control signal to the necessary control signals on the instruction decode circuits (page 4).

Designing and Implementing the Processor

Before Modification	After Modification
	

Before Modification	After Modification
	

Using the general logic strategy, add the logic for the instruction's control signal to the necessary control signals on the instruction decode circuits (page 4).

Designing and Implementing the Processor

Updating the instruction set

The instruction set for the processor can now be updated to reflect the additions of the newly implemented instructions.

Instruction Set: Instructions with 4-bit opcode and 12-bit operand

Binary Value	Hexadecimal Value	Operation	Remarks
1111 *XXXX XXXX XXXX*	**F***XXX*	See below	Enables other formats.
1110 *AAAA AAAA AAAA*	**E***AAA*	*JMP*	Unconditional jump.
1101 *DDDD DDDD DDDD*	**D***DDD*	*ADDI*	ADD immediate data. $\text{Acc} = \text{Acc} + DDD$
1100 *DDDD DDDD DDDD*	**C***DDD*	*MOVEI*	MOVE immediate data $\text{Acc} = DDD$
1011 *AAAA AAAA AAAA*	**B***AAA*	*MOVE*	MOVE from location. $Acc = contents\ of\ AAA$
1010 *AAAA AAAA AAAA*	**A***AAA*	*STORE*	MOVE to location. $AAA = Acc$
1001 *DDDD DDDD DDDD*	**9***DDD*	*LDA*	Load an immediate value into the address register
1000 *DDDD DDDD DDDD*	**8**DDD	*JSR*	Jump SubRoutine. Load the program counter from a 12-bit immediate value in the instruction, pre-decrement the program counter and use the decremented value of the program counter as a memory address to store the current value of the program counter.
0111 *DDDD DDDD DDDD*	**7***DDD*	*ADD_AD +*	Add address register with auto-increment. Add to the accumulator a value taken from a memory location that is specified by the address held in the address register and increments the value in the address register.

X – Placeholder
A – Address
D – Data

Instruction Set: Instructions with 8-bit opcode and 8-bit operand

Binary Value	Hexadecimal Value	Operation	Remarks
1111 1111 *XXXX XXXX*	**FF***XX*	See below	Enables other formats.
1111 1110 *RRRR RRRR*	**FE***RR*	*BZ*	Branch if zero (relative).
1111 1101 *RRRR RRRR*	**FD***RR*	*BNZ*	Branch if not zero (relative).

X – Placeholder
R – Relative Address

Instruction Set: Instructions with a 16-bit opcode and no operand

Binary Value	Hexadecimal Value	Operation	Remarks
1111 1111 1111 1111	**FFFF**	*STA*	Stores the contents of the accumulator at a memory location specified by the address register.
1111 1111 1111 1110	**FFFE**	*STA +*	Stores the contents of the accumulator at a memory location specified by the address register and then increments the address register.

Designing and Implementing the Processor

Executing Instructions with Updated Instruction Set

Using the BNZ instruction

Example: Program written in memory

Write a program using the processor's instruction set such that it:

- **loads the accumulator with the value 3;**
- **decrements the accumulator until its value is 0.**

The program should continuously cycle through these instructions.

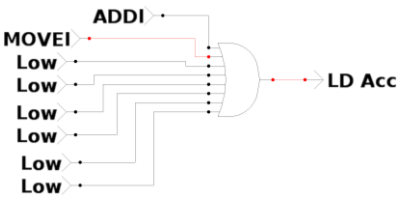
In order to load the accumulator with a value, the $MOVEI$ instruction must be used. The hexadecimal value for this operation is $\mathbf{C}DDD$.

Loading the accumulator with the hexadecimal value 3 will use the instruction with hexadecimal value $C003$ and this will be put into memory location $0x000$.

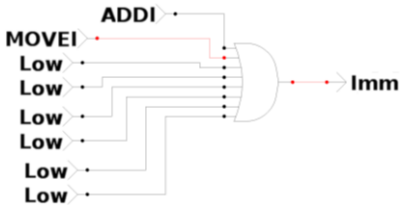
	0	1	2	3	4
0x00X	C003	0000	0000	0000	0000

(page 2)

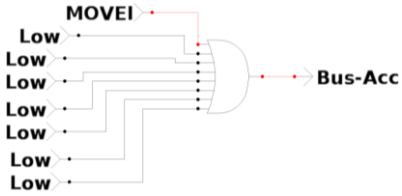
While this instruction is being executed, the control signals $LD\ Acc$, Imm and $Bus - Acc$ are activated.



(page 4)



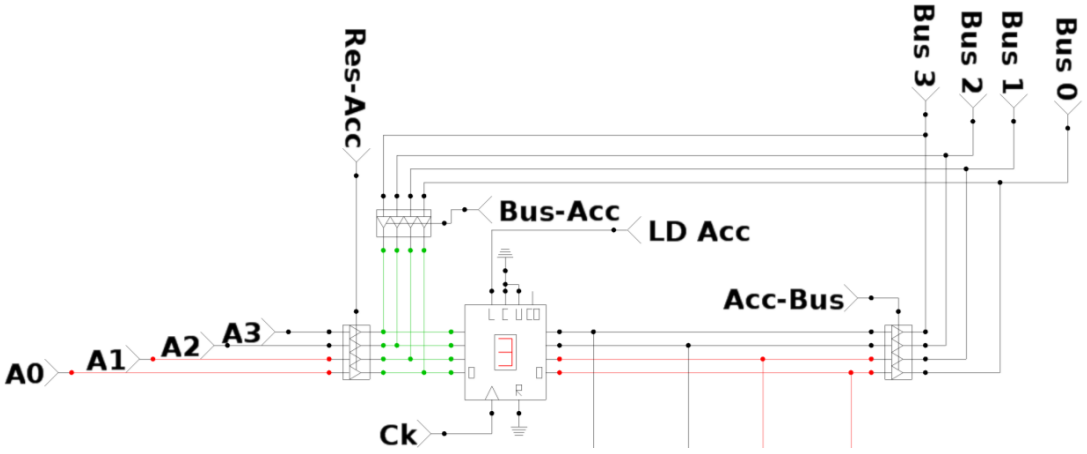
(page 4)



(page 4)

Load the accumulator with the value 3.

As a result of this instruction, the hexadecimal value 3 can be seen in the accumulator.



(page 5)

In order to increment or decrement the accumulator, the $ADDI$ instruction must be used. The hexadecimal value for this operation is $\mathbf{D}DDD$.

Decrementing the accumulator will require a subtraction operation. For this to be achieved, an addition of the negative number -1 must be performed such that $accumulator\ contents = accumulator\ contents - 1$. As a result, this operaion will use the instruction with hexadecimal value $DFFF$ and this will be put into memory location $0x001$.

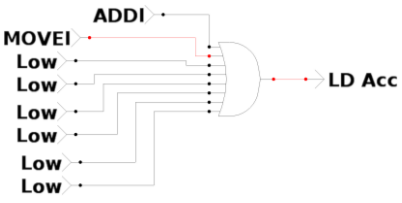
Decrement the accumulator until its value is 0.

Designing and Implementing the Processor

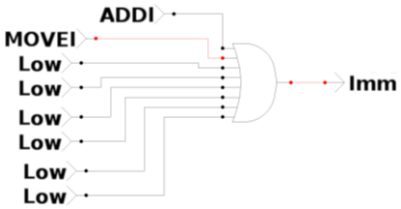
	0	1	2	3	4
0x00X	C003	DFFF	0000	0000	0000

(page 2)

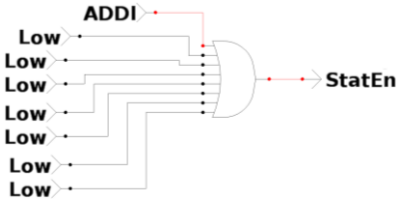
While this instruction is being executed, the control signals $LD\ Acc$, Imm, $StatEn$ and $Res - Acc$ are activated.



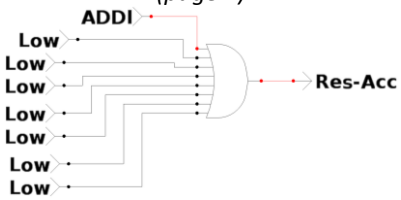
(page 4)



(page 4)

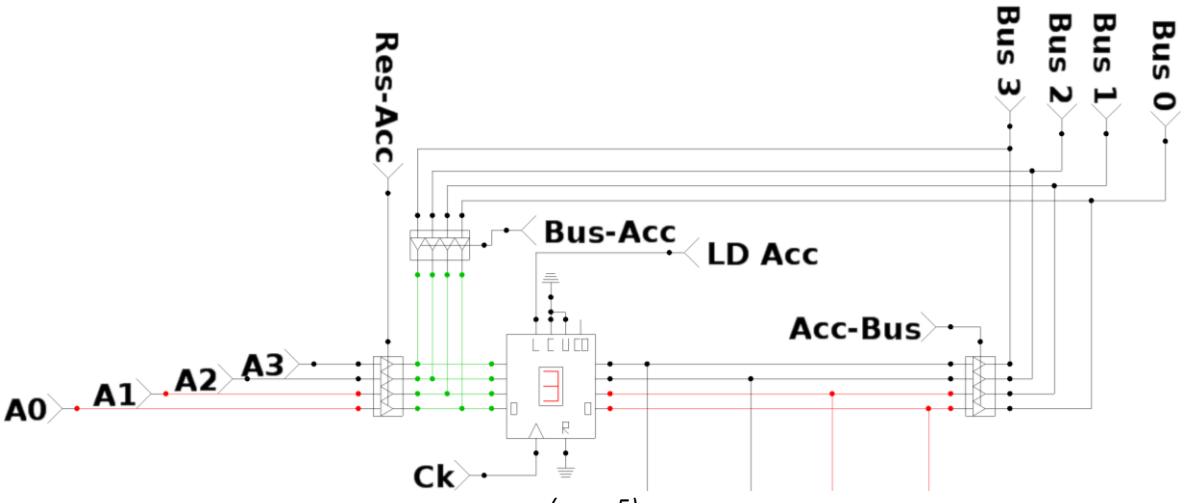


(page 4)

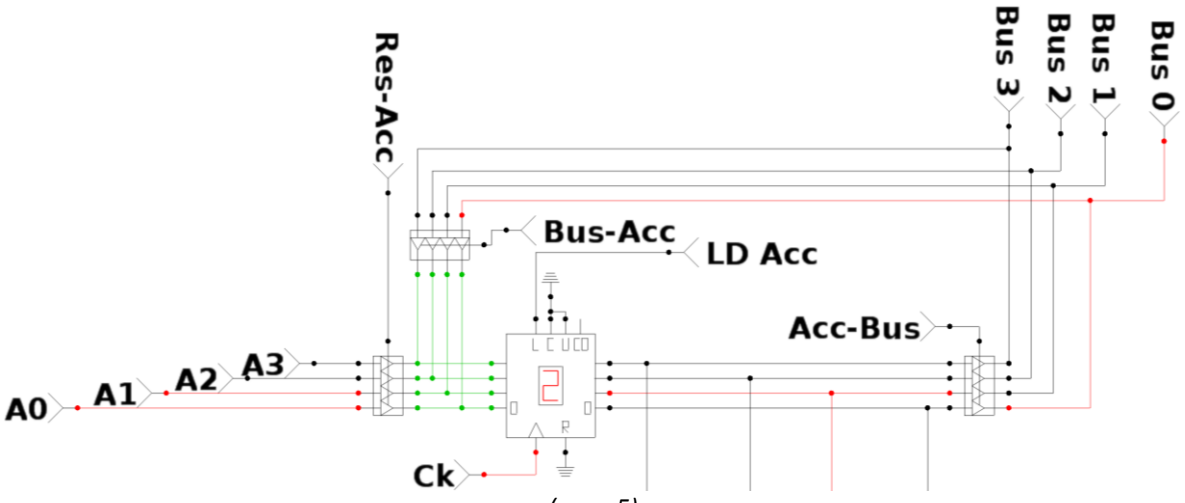


(page 4)

As a result of this instruction, the value in the accumulator has been decremented.



(page 5)



(page 5)

Decrement the accumulator until its value is 0 (continued).

Designing and Implementing the Processor

In order to continue decrementing the accumulator until its value is 0, a series of branch instructions must be implemented so that the program will:

- perform the decrement instruction repeatedly until the accumulator has a value of 0; and
- load the accumulator with the value 3 once the accumulator has value a 0.

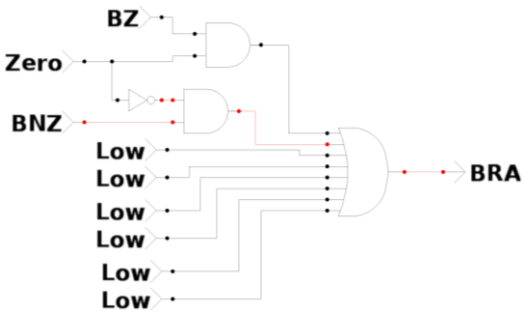
In order to perform a branch if the contents of the accumulator is not 0, the BNZ instruction must be used. The hexadecimal value for this instruction is $\mathbf{FD}RR$.

The branch instruction responsible for returning to the second instruction, which is responsible for decrementing the accumulator, will be present in memory location $0x002$. The BNZ instruction uses a relative address (RR) as the operand and the jump from memory location $0x002$ to $0x001$ is -1 – this is because the program counter (PC) points to the next instruction, which would be located at $0x003$, and a further decrement of 1 is required to reach $0x001$. As a result, this operaion will use the instruction with hexadecimal value $FDFE$ and this will be put into memory location $0x002$.

	0	1	2	3
0x00X	C003	DFFF	FDFE	0000

(page 2)

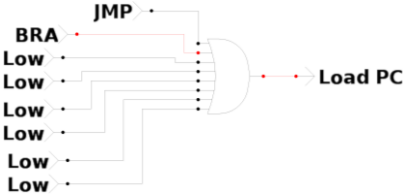
While this instruction is being executed, the control signal BRA is activated.



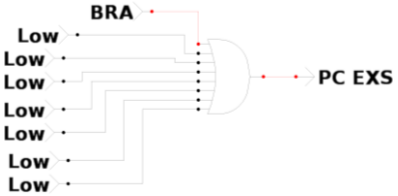
(page 4)

Decrement the accumulator until its value is 0 (continued).

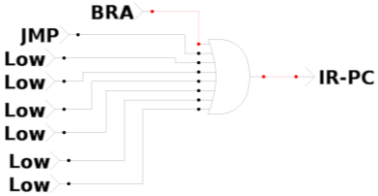
As a result of the control signal BRA being active, the control signals $Load\ PC$, $PC\ EXS$, $IR - PC$ and $Offset$ are activated.



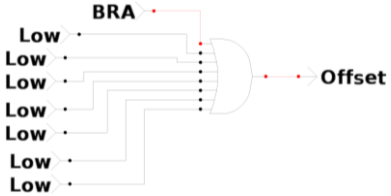
(page 4)



(page 4)



(page 4)



(page 4)

In order to perform a branch back to the first instruction, which is responsible for loading the accumulator with the value 3, the $JUMP$ instruction must be used. The hexadecimal value for this instruction is $\mathbf{E}AAA$.

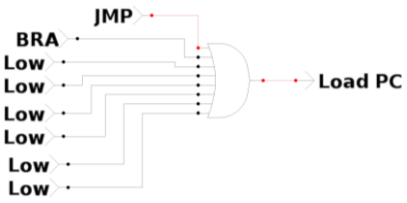
Performing an unconditional jump back to the instruction located in the memory location $0x000$ will use the instruction with hexadecimal value $E000$ and this will be put into memory location $0x003$. The placement of this instruction in memory relative to the other instructions in memory is important as it

must be after the BNZ instruction, so that it is not executed when the contents of the accumulator is not 0 as placing it before the BNZ instruction would not give the BNZ instruction opportunity to execute.

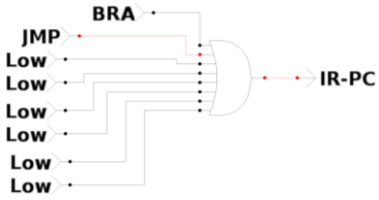
	0	1	2	3
0x00X	C003	DFFF	FDFE	E000

(page 2)

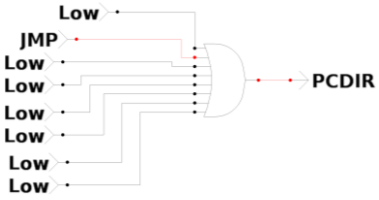
While this instruction is being executed, the control signals $Load\ PC$, $IR - PC$ and $PCDIR$ are activated.



(page 4)



(page 4)



(page 4)

If the contents of the accumulator is not 0, the $FDFE$ instruction will be executed. As a result of this instruction, the program will branch to the instruction $DFFF$ after the accumulator has been decremented such that it will be decremented until it reaches the value of 0.

Once the $EDFE$ instruction has been executed, the program will branch back to the $DFFF$ instruction.

	0	1	2	3
0x00X	C003	DFFF	FDFE	E000

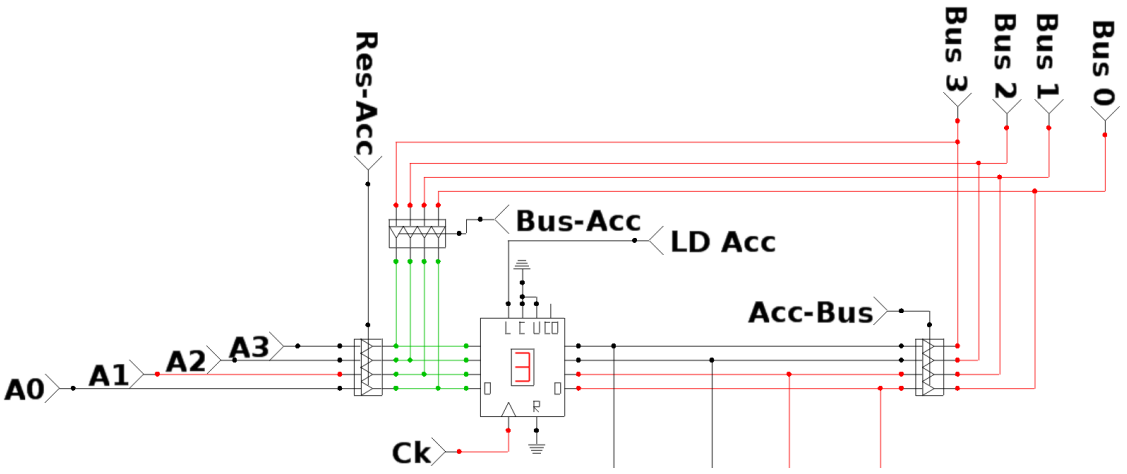
(page 2)

	0	1	2	3
0x00X	C003	DFFF	FDFE	E000

(page 2)

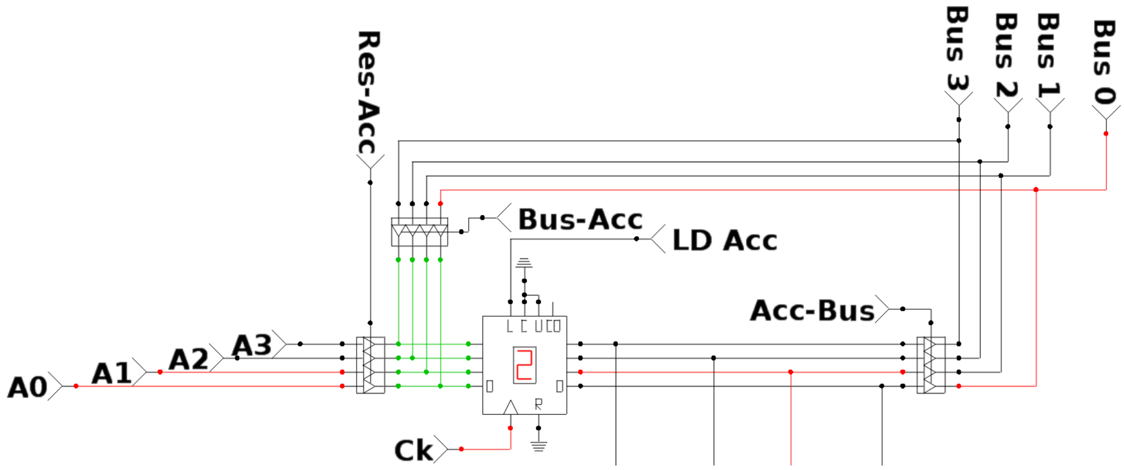
This will continue until the value of the accumulator is 0.

Decrement the accumulator until its value is 0 (continued).

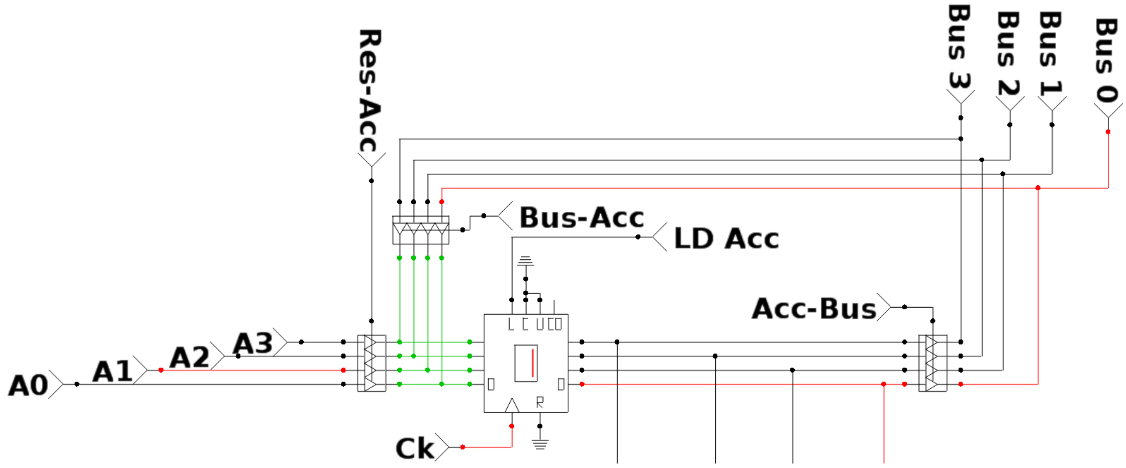


(page 5)

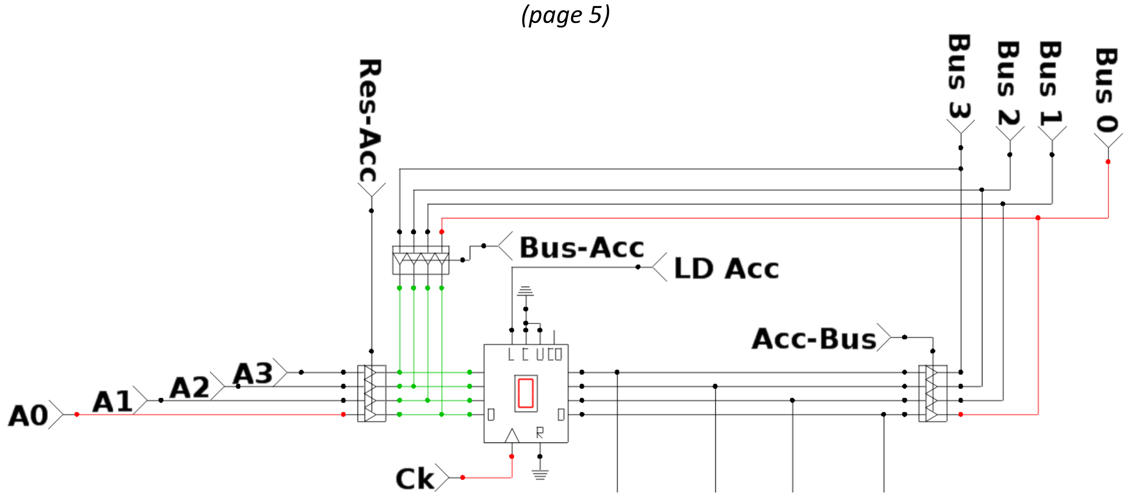
Designing and Implementing the Processor



(page 5)



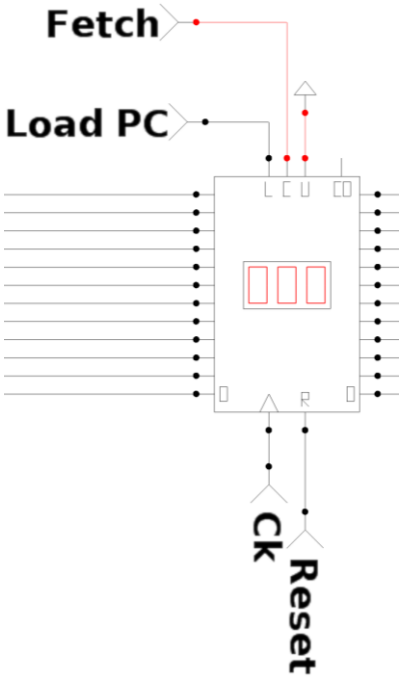
(page 5)



(page 5)

Decrement the accumulator until its value is 0 (continued).

Once the value of the accumulator is 0, the BNZ instruction will not be exeuted and the $JUMP$ instruction will be executed. As a result, the program counter (PC) is set to 000 and the memory location access on the next clock cycle is $0x000$.



(page 6)

Decrement the accumulator until its value is 0 (continued).

Final program

	0	1	2	3
0x00X	C003	DFFF	FDFE	E000

(page 2)

Designing and Implementing the Processor

Using the $MOVE$ and $STORE$ instructions

Example: Program written in memory

Write a program using the processor's instruction set such that it:

- **loads the accumulator with the hexadecimal value 32 from a memory location; and**
- **stores the contents of the accumulator in a memory location.**

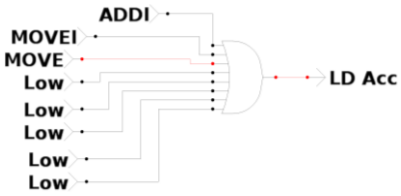
In order to load the accumulator with a value from a memory location, the $MOVE$ operation must be used. The hexadecimal value for this instruction is $\mathbf{B}AAA$.

Loading the accumulator with a hexadecimal value from a memory location will require the instruction to be stored in one memory location and the data to be stored in another location; this program will store the instruction at $0x000$ and the data at $0x001$. As a result, this will use the instruction with hexadecimal value $B001$ and this will be put into memory location $0x000$, and the data 0032 will be put into memory location $0x001$.

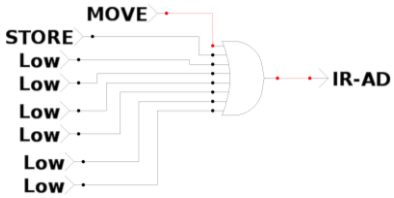
	0	1	2
0x00X	B001	0032	0000

(page 2)

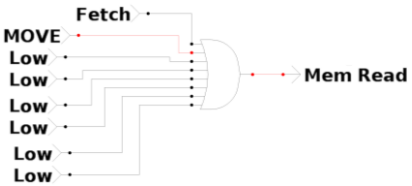
When this instruction is being executed, the control signals $LD\ Acc$, $IR - AD$ and $Mem\ Read$ are activated.



(page 4)

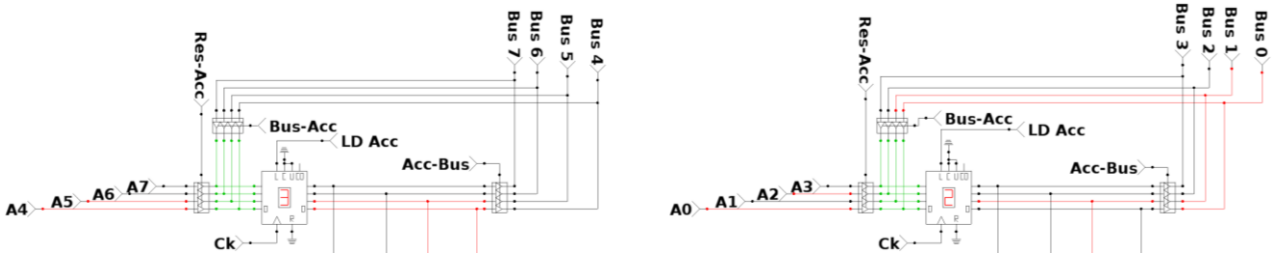


(page 4)



(page 4)

As a result of this instruction, the hexadecimal value 0032 can be seen in the accumulator.



(page 5)

In order to store the contents of the accumulator in a memory location, the $STORE$ operation must be used. The hexadecimal value for this instruction is $\mathbf{A}DDD$.

Storing the contents of the accumulator in the memory location $0x003$ will use the instruction with hexadecimal value $A003$ and this will be put into memory location $0x002$.

	0	1	2
0x00X	B001	0032	A003

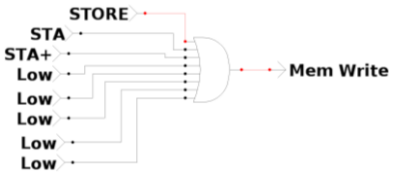
(page 2)

When this instruction is being executed, the control signals $Mem\ Write$, $IR - AD$ and $Acc - Bus$ are activated.

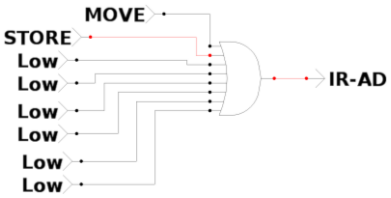
Load the accumulator with the hexadecimal value 32 from a memory location.

Store the contents of the accumulator in a memory location.

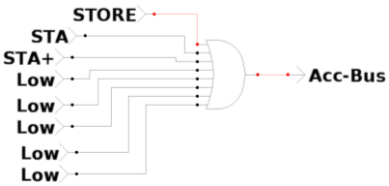
Designing and Implementing the Processor



(page 4)



(page 4)



(page 4)

As a result of this instruction, the hexadecimal value 0032 can be seen in the memory location $0x004$.

	0	1	2	3
0x00X	B001	0032	A003	0032

(page 2)

Store the contents of the accumulator in a memory location (continued).

Final program

	0	1	2
0x00X	B001	0032	A003

(page 2)

The 68000 Processor

Introduction to the 68000 processor

What is the 68000 processor?

The **Motorola 68000 series** is a family of 32-bit CISC microprocessors.

During the 1980s and early 1990s, they were popular in personal computers and workstations and they were most well known as the processors powering the:

- early Apple Macintosh;
- Commodore Amiga;
- the Sinclair QL;
- Atari ST; and
- the Sega Genesis (Mega Drive).

Although no modern desktop computers are based on processors in the 68000 series, derivative processors are still widely used in embedded systems. However, there are similarities to modern processors disregarding the extensions in modern processors added to improve speed and performance.

The 68000 reflects the standard practices of design from its time. It contains an internal Harvard architecture that executes each complicated instruction as a kind of subroutine of simpler internal instructions. This processor is similar to the aforementioned CEDAR Logic processors, however it is not as limited as it is able to perform instructions that take multiple clock cycles to complete and aims to make each instruction do as much as possible as to not perform fetch cycles as frequently.

CISC vs RISC

Complex Instruction Set Computer (CISC) processors have many instructions available.
Reduced Instruction Set Computer (RISC) processors have few instructions available.

CISC	RISC
Focused on hardware, more processor circuitry.	**Focused on software**, more programming code.
Complex hardware as more circuitry is required to create less logic gates in the CPU.	**Simple hardware** as less circuitry is required to create less logic gates in the CPU.
Larger physical size because there is more circuitry.	**Smaller physical size** because there is less circuitry.
Simple software code requiring more RAM as fewer operations must be completed to perform an action that may only take multiple operations in a RISC processor and less RAM is required to store these operations.	**Complex software code requiring more RAM** as multiple operations must be completed to perform an action that may only take one operation in a CISC processor and more RAM is required to store these operations.
Some operations require multiple fetch-decode-execute cycles to complete as the instructions are complex..	**Operations complete in one fetch-decode-execute cycle** as the instructions are simple.
Similar to high level languages as its instruction set is a similar size to the one in a high level language, making it less similar to assembly language; fewer lines of code must be used to represent one line of code in a high level language.	**Dissimilar to high level languages** as its instruction set is much smaller than the one in a high level language, making it more similar to assembly language; more lines of code must be used to represent one line of code in a high level language.
Complier has less work to do as there are less lines of code.	**Complier has more work to do** as there are more lines of code.
Instruction format can be variable.	**Instruction format is fixed**.
Cannot make use of pipelining because the instruction format is unknown so it is not possible to know how many cycles an instruction will take to complete.	**Can make use of pipelining** because the instruction format is predetermined so it is possible to know how many cycles an instruction will take to complete.
Higher heat production and energy consumption because they are larger and therefore do not last as long on a battery charge and are likely to require a heat sink and a fan to cool them.	**Lower heat production and energy consumption** because they are smaller and therefore can last longer on a battery charge and do not always require a heat sink and a fan to cool them.

The 68000 Processor

Characteristics of the 68000 processor

Architecture

The 68000 architecture is comprised of:

- eight 32-bit data registers (accumulators) labelled $D0$ to $D7$;
- eight 32-bit address registers labelled $A0$ to $A7$, in which $A7$ is used as a stack pointer;
- a 24-bit address;
- a 16-bit data bus;
- 16-bit instructions however, if operands are too large to fit in one instruction set, they may be distributed across subsequent words.

The arithmetic logic unit (ALU) inside the 68000 is capable of addition and subtraction operands. Multiply and divide operands are implemented using multi-cycle instructions, in which a succession of addition or subtraction is performed.

Data sizes

In the CEDAR Logic processor, the wordlength was fixed to 16-bit data with 12-bit address. While the 68000 processor, has a range of data sizes available:

- byte – 8-bit;
- word – 16-bit; and
- longword – 32-bit.

The 68000 Processor

Introduction to Assembly Language

What is assembly language?

Assembly language is a low-level language that uses mnemonics to refer to instructions for the CPU to process. It requires an assembler to convert the assembly instructions into machine code.

The code in assembly language has a:

- one-to-one correspondence with the corresponding machine code; and
- many-to-one correspondence with the corresponding high-level language code.

This means that a prediction can be made of the exact machine code. This can be achieved as assemblers take a literal and clearly expressed program and translate this into machine code (0's and 1's), whereas compliers and interpreters for high level languages take an expression of the program and create a version in machine code (0's and 1's) that will work on the processor.

Nature of assembly language

Different assembly languages

Although there is always a one to one correspondence between assembly language and machien code, it is possible to have different assembly languages for the same processor. For example, a 68000 program written for an x86 architecture may differ from one written for an UNIX architecuture, such as different order of source and destination or different directives.

This means that it is necessary to de-couple the processor from the assembler.

The processor vs the assembler

The processor is a piece of physical hardware. The processor instruction set is defined by binary bits, known as machine code, that is raw code that the processor can execute directly.

The assembler is a piece of software that takes a program written in assembly language, a human readable form, and translates it into machine code.

Evaluation

Advantages	Disadvantages
Same efficiency of execution as machine code as it uses a one-to-one translator, known as an assembler.	**Not portable** because the code has been compiled according to a specific processor's instruction set / architecture.
Ability to precisely control the actions of the processor because the instructions have a one-to-one relationship with machine code instructions.	

The 68000 Processor

Assembly Language Instructions

Instruction contents

A assembly language instruction can contain:

- mnemonics – these specifiy instructions, such as MOVE and ADD;
- directives – these specify data or provide the assembler with other information, for example not everything that is going to the processor will be an instruction; and
- identifiers – these are words with a meaning and will be specified using an instruction or directive and:
  - enable memory locations or constants to be referred to by name; and
  - have a one to one correspondence between name and number.

Instruction format

Each line in an assembly lanuguage program usually specifies one instruction or directive. The meaning of each instructions elements are controlled by their position on the line.

Syntax: Line format

```
Label_field:     opcode_field     first_operand_field,second_operand_field
```

- The seperation is achieved by spaces and/or tabs.
- The `Label_field` must be at the start of the line and will be used as an identifier to specify the address of the instruction. The colon (`:`) is required by some assembers.
- The `opcode_field` cannot be at the start of the line.
- No seperation is required between the `first_operand_field` and the `second_operand_field`.

Unlike the CEDAR Logic processor, the 68000 processor has multiple registers. This means that instructions must be formatted such that the source location is on the left and the destination location is on the right.

Syntax: Standard instruction format

Two operands	One operand	No operand
`OPCODE SRC, DST`	`OPCODE DST`	`OPCODE`

Example: `MOVE` instruction

```
MOVE mem_address,D0
MOVE D0,mem_address
```

In this example:

- the first line of code would move the contents of the location `mem_address` to the location `D0`; and
- the first line of code would move the contents of the location `D0` to the location `mem_address`.

In addition, it is necessary to specify the size of the data. This is usually achieved by adding a suffix to the instruction mnemonic:

- `MOVE.B` – byte (8 bit);
- `MOVE.W` – word (16-bit); and
- `MOVE.L` – longword (32-bit).

If no suffix is decleared, the default option of a word (16-bit) will be used. However, it is likely that, in the real world, longwords will be used more often for cases such as numbers with great accuracy.

The 68000 Processor

68000 Instructions

Simple instructions

Some simple instructions include:

- `MOVE` – takes data from one place to another;
- `ADD` – the sum of the source and destination is placed in the destination;
- `SUB` – subtracts one number from another;
- `JMP` – go to another place;
- `JSR` – Jump SubRoutine; and
- `RTS` – Return To Subroutine.

Branch instructions

Some branch instructions include:

- `BRA` – branch always;
- `BEQ` – ;
- `BNE` – ;
- `BGT` – ;
- `BPL` – ;
- `BMI` – ;
- `BVS` – ;
- `BVC` – ;
- `BCS` – ;
- `BCC` – ;
- `BLE` – ; and
- `BLT` – .

The 68000 Processor

Binary Shifts

Definition

A **binary shift** is a bitwise operation to manipulate binary values by moving the bits left or right by a number of places.

Rules of binary shifts

- A shift to the left by n number of places will multiply by 2^n.
- A shift to the right by n number of places will divide by 2^n.

Process: Binary shifts

1) Represent the operand in binary.
2) Determine whether the operation is a multiplication or division.
3) Represent the number which the operand is to be multiplied by as a power of 2.
4) Determine in which direction and what magnitude the shift will occur:
    - multiplication by $2^{\boldsymbol{n}}$ – a shift to the left by $\boldsymbol{n}$ places; or
    - division by $2^{\boldsymbol{n}}$ – a shift to the right by $\boldsymbol{n}$ places.
5) Perform the shift.
6) Write the answer.

Examples

Example: Multiplication using binary shifts

Perform the operation 5×16 using binary shifts.

	128	64	32	16	8	4	2	1
Operand	0	0	0	0	0	1	0	1

Represent the operand in binary.

Multiplication

Determine whether the operation is a multiplication or division.

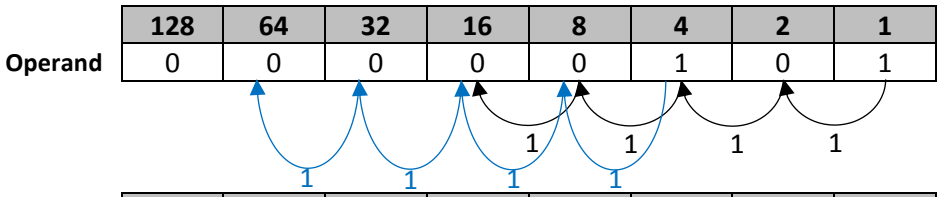
$16 = 2^4$

Represent the number which the operand is to be multiplied by as a power of 2.

Left shift by 4 places.

Determine in which direction and what magnitude the shift will occur.

The 68000 Processor



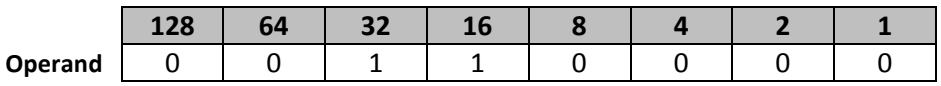
Perform the shift.

Binary **01010000** Denary **80**

Write the answer.

Example: Division using binary shifts

Perform the operation $48 \div 4$ using binary shifts.



Represent the operand in binary.

Division

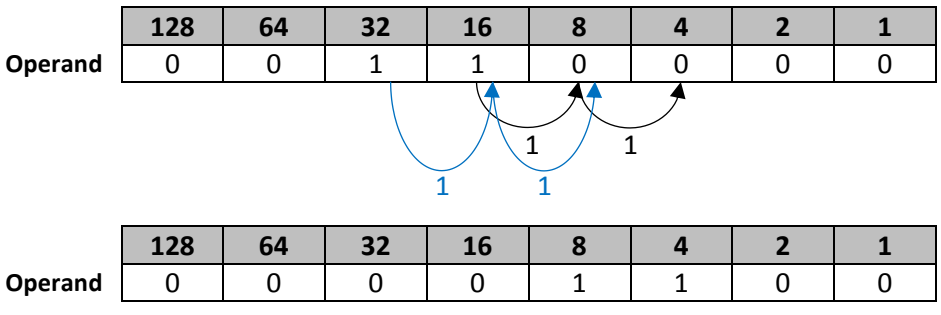
Determine whether the operation is a multiplication or division.

$4 = 2^2$

Represent the number which the operand is to be multiplied by as a power of 2.

Right shift by 2 places.

Determine in which direction and what magnitude the shift will occur.



Perform the shift.

Binary **00001100** Denary **12**

Write the answer.

Precision

A binary shift can lead to an overflow error, where there are not enough bits to represent the new binary value. This means that some precision is lost.

This problem can be overcome by:

- increasing the number of bits; or
- represent the number in floating point binary as a larger range of numbers can be represented.

The 68000 Processor

Shift instructions

Shift instructions move a register by using a:

- shift left that multiplies by two; or
- a shift right that divides by two.

As seen before, this is useful for multiplication or division.

There are variants of shift instructions depending on what must be done to the end of the word:

- sign extend – this is important if shifting right as the most significant bit (MSB) will be replicated to fill the spaces such that sign of the number will remain unchanged.
- logical – fill the spaces with zeros (0); and
- rotational – fill spaces with contents of the other end.

The 68000 Processor

Assembler Directives

<table>
<tr><th>Directive</th><th>Mnemonic</th><th>Information</th></tr>
<tr><td>Origin</td><td>ORG</td><td>Tells the assembler where to put the code in memory.</td></tr>
<tr><td>Define Space</td><td>DS</td><td>Reserves memory locations and assigns them to a named variable. The number of memory locations to be reserved is specified by the operand and can be modified to a byte (.B), word (.W) or longword (.L).

Example: Using define space
<pre>count   ds.l   1</pre>
In this example:
• the reserved memory location is assigned to the named variable is count; and
• the size of the reserved memory location is one longword.

The address of this memory location becomes equilvalent to the named variable, count. This means that, when the program is assembled, references to count will be replaced by the address of its assigned memory location.</td></tr>
<tr><td>Define Constant</td><td>DC</td><td>Assigns an initial value to a named variable. The length of the variable can be modified to a byte (.B), word (.W) or longword (.L).

Example: Using define constant
<pre>count   dc.l   0</pre>
In this example, the code is effectively equivalent to <code>int count = 0;</code> in C++.

This value may be changed later in the program.</td></tr>
<tr><td>Equate</td><td>EQU</td><td>Makes an association between a named variable and a constant value. This does not allocate memory.

Example: Using equate
<pre>SIZE   EQU $60</pre>
In this example:
• SIZE is in the label field;
• $ means hexadecimal;
• an association is made such that the identifier SIZE will be replaced by the hexadecimal number 60 allowing SIZE to be used instead of $60 in an immediate mode instruction; and
• the code is functionally similar to CONST INT in C++.</td></tr>
</table>

The 68000 Processor

Addressing Modes and Program Counter Modes

Addressing modes

Addressing Mode	Operand Syntax	Information	Evaluation	
Immediate	The operand is prefixed by a hash (`#`) character. For example: • `#4`; and • `#size`.	Operand is the data and is used directly.	**Advantages**	**Disadvantages**
			Fast because no memory locations need to be accessed.	**Limited size** as the limit is defined by the maximum size of the operand and therefore often used for constants that have a limited size.
			Useful for instructions involving constants.	
Direct *(or Absolute)*	The operand contains the address and can be written numerically or by using an identifier. For example: • `0004`; and • `size`.	Operand is the address of the data.	**Advantages**	**Disadvantages**
				Can create errors because, if data or instructions are re-located in memory, the memory address will be invalid.
Register *(or indirect)*	The operand contains the standard register name. For example: • `D0`; • `D1`; • `A5`; and • `A6`.	Data is in the register.	**Advantages**	**Disadvantages**
			Enables a larger range of addressable locations.	**Uses more memory** because two memory references are used, one to store the location of the intermediate location and one to store the data or instruction.
Address register indirect	The operand contains the standard register name with an auto-increment or an auto-decrement. For example: • `(A1)+`; • `-(A4)`; and • `(A0)`.	Address register contains the address of the data. The auto-increment is performed after the data is accessed and the auto-decrement is performed before the data is accessed so that this can also be used as a stack pointer. It is also possible to perform an address register indirect with no auto-increment or auto-decrement.		

The 68000 Processor

Addressing Mode	Operand Syntax	Information	Evaluation
Address register indirect with offset	The operand contains the word `offset` and the standard register name for the address register is enclosed by brackets (`()`). For example `offset(A3)` means $A3 + offset$.	Address of the data is the sum of the contents of the address register and the offset. The offset may be defined using a directive.	
Address register indirect with index and offset	The operand contains the word offset and the standard register name for the address register and data register is enclosed by brackets (`()`). For example `offset(A3,D1.L)` means $offset + A3 + D1$.	Address of the data is the sum of the two registers and the offset. The index register must have a size specifier and the index register can be an address register or a data register. The offset may be defined using a directive. The index is used as, in practice, the offset cannot be large and therefore the index is required to achieve larger ranges.	

Program counter modes

Most address register modes can also be used with the program counter (PC) instead. However, this can only be done for certain instructions – only for the source address, not usually for a destination address.

The 68000 Processor

Instructions

Addressing modes

Mnemonic	Instruction	Information

`lea`	Load effective address	A different way of writing a move instruction where the destination register is an address register.
		
		The address of the source is written as if the data was being moved however, the address of the data is being moved.
		
		An example sequence would be:
		`lea X,a0`
		`move (a0),d0`
		which would be equivalent to the instruction:
		`move X,d0.`

`bsr`	Branch to subroutine	Similar to Jump SubRoutine (JSR) but internally uses a relative address.
`Trap`	Trap instruction	A way of calling certain types of subroutines that are pre-written and form part of the operating system.
`Movem`	Move multiple	A multi-cycle instruction that moves several register to or from memory and uses the stack pointer to supply the address, such that data is pushed on to the stack or popped from the stack.

Interrupts

Definition

An **interrupt** is a signal from a software program, hardware device or the processor's clock which causes the processor to stop the current process and carry out the interrupt task. Once the task is complete, the processor returns to the previous process. Interrupts and traps are commonly known as exceptions.

Why are interrupts required?

Sometimes it is useful to be able to break into a program's flow of execution to perform some task. This task could be a response to a mouse-click or keystroke on the keyboard.

This response could be achieved by constantly testing memory locations using a test instruction in the program that checks the status of an external signal. However, this would require the program to have many test instructions in order to provide a timely response.

Instead, interrupts are used as they provide a solution that allows timely responses to devices with different speeds:

- slow devices, such as keyboard and mice, should not cause the program to "wait around" for an action; and
- fast devices, such as disk drives and video displays, require precise timing.

The operating system requires regular interrupts to allow it to "supervise" multiple programs by sharing time between each program. A timer is run and generates an interrupt, the operating system saves the state of one program and allows another program to be run – this divides up the processor time such that it allocates resources to programs fairly.

Interrupt mechanism

The interrupt mechanism is built into the processor and allows a program's flow of execution to be broken in order to perform something different.

A piece of hardware toggles a signal that is detected by the interrupt mechanism. If this has been toggled, the next instruction is not fetched and instead, a jump to the interrupt instruction is performed. Similar to Jump SubRoutine (JSR), the return address is pushed on to the stack and it jumps to another location. The state of condition codes must also be push on to the stack as an interrupt may happen at any time and therefore it must sae the previous values of the condition codes so that they can be used again when returning to the previous program, that may have been performing arithmetic, in order to prevent errors or corruption.

After the interrupt service routine (ISR) is performed, a return from instruction is performed. The return address and condition codes are popped from the stack and the return address is used to return to the previous program.

Information: 68000 interrupts

Vector Location	Name	Information
Vectors 25-31	Reset	Three external wires called $ip0$, $ip1$ and $ip2$ are decoded into a number N_i. These three bits form a masking setup. If $N = 0$, then nothing is connected and there will be no interrupt. If $N_i > 0$ as an interrupt happens, the vector is $N_i + 24$, unless the interrupt is "masked" Otherwise, vectors 64-255 are used.
Vectors 64-255	Bus Error	Some external hardware puts a vector number, in the range 64-255, on the bus. This uses the same interrupt hardware mechanism as vectors 25-31 however, if something appears on the bus at the right moment, then it will go to the locations 64-255.

The 68000 Processor

Interrupt service routine (ISR)

An **Interrupt Service Routine (ISR)** contains the operations that are performed to deal with an interrupt. Different types of interrupts can have different routines.

The address of an interrupt service routine (ISR) is not specified in an instruction. Instead, it is determined by a hardware mechanism. This hardware mechanism may be such that:

- the address is built into the processor; or
- the address is supplied by a piece of external hardware.

If the processor has a reasonably large address space, it will be rather long to be built directly into the processor. In addition, it will be fiexed at the time of the design. Ideally, this is something that needs to be managed by software but, from a security standpoint, hardware must also be apart of the implementation.

As a result, the address for an ISR is stored in an **interrupt vector table (IVT)**. An IVT is a region of memory which can be, and often is, setup as read-only memory (ROM). This allows the "address of the address" to be built into the hardware, rather than the ISR address. This means that the ISR addresses are not fixed in the processor, rather they are in ROM and can be re-programmed but not easily in order to prevent corruption by viruses or rogue user programs. Therefore, this provides a goodbases on which to provide security for an operating system.

Information: The 68000 interrupt vector table (IVT)

Vector Location	Name	Information
Vector 0	Reset	Resets the supervisor mode stack pointer and program counter (PC): • the supervisor mode stack pointer is loaded from location 0; and • program counter (PC) is loaded from location 4. This occupies eight bytes as it has to setup the stack pointer when resetting.
Vector 2	Bus Error	Caused by accessing non-existent or protected memory.
Vector 3	Address Error	Caused by trying to access a word or instruction at an odd address.
Vector 4	Illegal Instruction	Caused by trying to execute an opcode that has not been assigned.
Vector 8	Privilege Violation	Caused by trying to execute a privileged instruction in user mode.
Vector 9	Trace	Used to implement a single step facility in a real hardware processor. If the trace bit is set, an interrupt will be generated after every instruction, allowing debugging by stepping through each line of the code using the output from the contents of the registers.

Interrupts and security

The 68000 has two modes of operation:

- user mode; and
- supervisor mode.

Other processors may have different modes. For example, ARM processors typically have: systems mode; supervisor mode; and trusted mode, and Intel processors did not originally have any modes but have been added in subsequent years and have lead to many security problems.

A different stack pointer is used for both the user mode and supervisor mode. This is because the operating system (OS) needs to keep a separate stack for every user program and "master" stack. Using the same stack pointer for user program stacks and the "master" stack would mean that if a user program crashed, the entire operating system (OS) would also crash – instead, if a user program crashes, it will only impact that user program's stack pointer and therefore allow other user programs and the operating system (OS) to remain unaffected.

Certain system resources can only access a specific mode and no resource can simple request to change between modes. In user mode, access to some resources is restricted and some instructions are not allowed to be executed. There is no instruction that allows a transition to be made from user mode to supervisior mode as this could cause corruption to the "master" stack.

Transition in to supervisor mode occurs when there is an interrupt. In order to transition from user mode to supervisor mode, a fake interrupt is generated using the IVT, this will be allowed as an operating system (OS) program will access the supervisior mode on behalf of the interrupt. In supervisior mode, there is an instruction available to transition back to user mode and a return from interrupt instruction will cause this transistion from supervisor mode back to user mode to happen automatically.

Traps

Definition

A **trap (or software interrupt)** is an exception in a user process. This is the usual way to invoke a kernel routine (a system call) because those run with using the supervisior mode rather than the user mode.

Why are traps required?

Since it is only possible to transition from user mode to supervisor mode using an interrupt, a trap is used when it is necessary to enter supervisor mode directly from software. This will transfer control to the operating system (OS) as the transition will be achieved through use of the IVT.

Most processors also use traps to handle anomalous events in programming, such as division by zero or invalid memory access – for example, accessing indices outside of an array's bounds.

A return from exception (RTE) instruction, commonly called return from interrupt (RTI) on other processors, is used to make the transition from supervisior mode back to user mode.

Information: 68000 Traps

Vector Location	Name	Information
Vector 5	Zero Divide	Occurs if a divide instruction is attempted with a zero denomination.
Vector 6	`TrapV` instruction	Can be used to detect overflows.
Vector 7	`CHK` instruction	Can be used to detect array bound violations.
Vectors 32-47	Trap instruction	• Always cause an exception. • The instruction has an operand in the range 0-15. • The operand is added to 32 to determine the vector number. • These traps are used for system routines by the Easy68k Simulator.

Cause of interrupts

Hardware Interrupts	Software Interrupts
• Hardware interrupts with a hard-wired IVT entry, such as reset and “autovectors” 1-7, are allocated to interrupts as they occur. • Hardware interrupts with an externally supplied vector IVT entry offer the device the ability to tell the processor where it will go in the IVT.	• Software interrupts caused by anomalous events, such as address error, privilege violation and divide by zero. • Software interrupts caused by specific “checking” instructions, such as `trapv` to check for overflow and `chk` to perform an array bound check – there are specific IVT locations for these instructions. • Software interrupts caused deliberately, known as traps, used to put output to the screen or window that plays a role on the screen.

Why are parameters required?

When a subroutine is called in a program, there may be parameters that must be passed to the subroutine.

Passing parameters using registers

If there are only a few parameters to be passed to a subroutine, register may be used. However, this can cause issues if the number of parameters exceeds the number of available registers or there are nested subroutines, in which a subroutine calls another subroutine. In this case, a stack would be required to allow new parameters to be pushed to the stack and popped from the stack when needed.

Accessing stack based parameters

The parameter to be accessed may not be on top of the stack. This means that a relative addressing mode can be used as the parameter is located a fixed distance from the stack pointer.

Address register indirect with offset will be used where the address is the sum of the address register and the offset. When accessing a parameter, `offset(A7)` would be used as access would be relative to the stack pointer. The offset will be dependent on how far away a parameter is from the stack pointer, this can be calculated as each instance of a subroutine will be accessing the parameters pushed to the stack immediately before that instance started.

Local variables

Each subroutine must be able to have its local variables, there are only accessible within the subroutine and only present while the subroutine is being executed. This is crucial when implementing recursion as each instance must have its own copy of the variables.

The local variables could be added on top of the stack pointer. However, if another subroutine is calld, it will trash the local variables already on the stack and therefore a stack frame is required.

Passing parameters using stack frames

Definition

A **stack frame** is a memory management technique used in some programming languages for generating and eliminating temporary variables.

The 68000 Processor

Structure of the stack frame

Stack frames form part of the call stack, which is a dynamic data structure maintained inside a computer system's memory by the operating system. The call stack can be responsible for maintaining the information required during program execution.

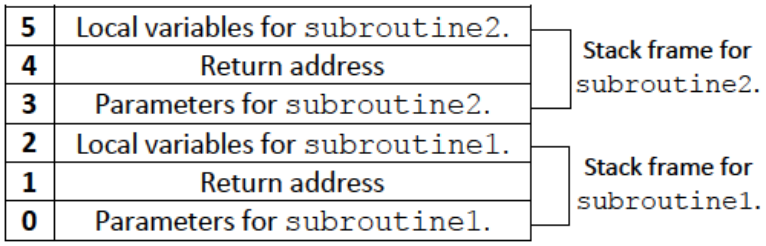
Each stack frame corresponds to a subroutine that is currently in progress.

Assuming the following subroutine definitions:

- `subroutine1(parameter1, parameter2);` and
- `subroutine2(parameter3, parameter4),`

where `subroutine1` is called from the main program and `subroutine1` calls `subroutine2` during its execution, an abstraction of their respective stack frames can be created.

Diagram: Stack frame abstraction



When `subroutine1` is called from the main program:

- the parameters `parameter1` and `parameter2` for `subroutine1` will be pushed on to the stack in reverse order;
- the return address will be pushed on to the stack, this is the current value of the program counter (PC); and
- any local variables declared in `subroutine1` will be pushed on to the stack in normal order.

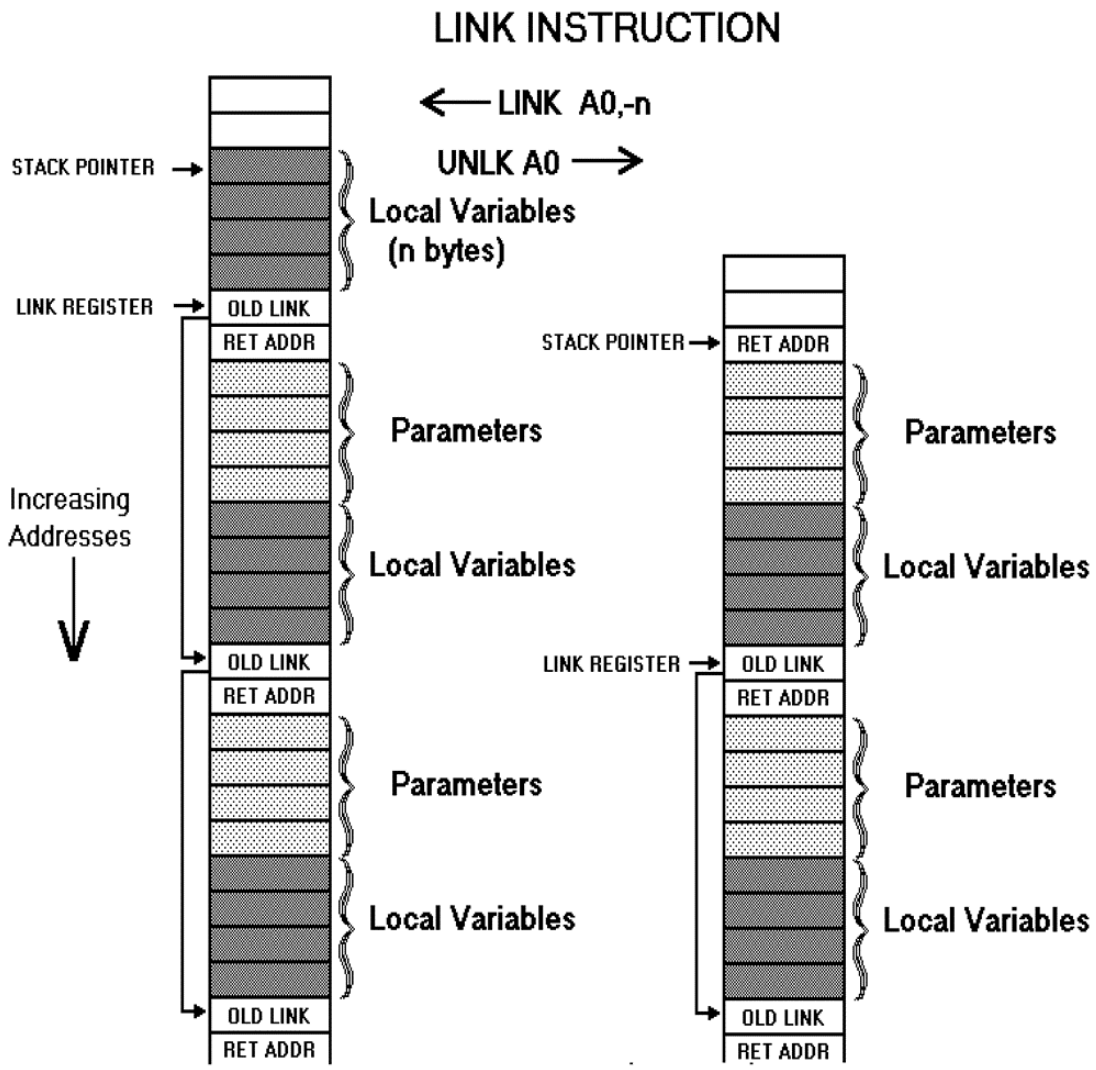
When `subroutine2` is called from `subroutine1`:

- the parameters `parameter3` and `parameter4` for `subroutine2` will be pushed on to the stack in reverse order;
- the return address will be pushed on to the stack, this is the current value of the program counter (PC); and
- any local variables declared in `subroutine2` will be pushed on to the stack in normal order.

When execution of `subroutine2` is complete, the return address will be used to return to `subroutine1`.
When execution of `subroutine1` is complete, the return address will be used to return to the main program.

The 68000 Processor

Diagram: Stack frame abstraction



The diagram shows the situation before (right) and after (left) a subroutine called.

The link register (or frame pointer) is a separate register used as a reference for relative addressing rather than the stack pointer, such that:

- local variables are at negative offsets; and
- parameters are at positive offsets.

This allows the stack pointer to be used for flexibly.

At a given point in time, a subroutine has been called and entered at which point the stack frame is in the following state:

- the stack pointer – pointing to the return address which is used when the subroutine completes;
- parameters – the parameters for the function that has just been called are located below the stack pointer; and
- link register – pointing to the location above the previous return address.

When the link instruction is called:

- the stack pointer will move up by a given number of bytes, `n`; and
- The previous value of the link register is pushed to the stack and the link register points to this location.

The link registers form a chain where the current one is in the address register and the location it points to contains the previous value. This provides a good safety mechanism as, in the event that the top of the stack is corrupted, the link register provides chains so that it is possible to use the unlink instruction to restore to an earlier situation in the program efficietly and attempt to recover execution.

The 68000 Processor

Stack frames and high level languages

Tradition high level languages, such as Algol, Pascal and C, only allow variables to be declared at the start of a "block". A block means that part of code between braces ({}) or the keywords `begin` and `end` in older languages.

This allowed the compiler to easily collate all of the local variables and use a single stack frame. This is most likely still the behaviour of more modern complilers however, in order to improve convenience for programmers, variables can now be declared anywhere in the code. This is possible as compliers are now able to "look ahdead" and therefore make the space allocation for the variables at the beginning. This means that the "object" code for a high level language program may not reflect the locations of the variable declarations exactly.

Evaluation

Advantages	Disadvantages
Subroutines are "re-entrant" as recursion can be used such that they can be called when an instance of that subroutine is already in progress or, in more general situations, multiple porcesses can share the same subroutines.	**The stack may become corrupted** as the stack is a dynamic data struction and therefore may "collide" with other data and cause corruption. However, this issue is somewhat addressed as many operating systems have a separate system stack away from the user stack such that the system does not crash if a user program crashes.
Code can easily be made "position independent" as everything is referenced dynamically rather than directly. This is done by relative positions to registers such as the program counter. Although, "position independent" means that code can be put anywhere in memory but does not mean relocatable as there may be consequences when moving a program in memory during execution.	

The 68000 Processor

Example Programs

Example #1

d

This will be a program that adds two bytes, two words, two longs words and then displays the results. This will demonstrate the use of trap #15 for display and subroutine.

Code: Example #1

```
START       ORG     $1000       the program will load into address $1000

            move    #3,d0       task number 3 into D0
* task number 3 is used to display the contents of D1.L as a number
            trap    #15         display number in D1.l

            bsr     newLine

* Display D2
            move    #14,d0      task number 14 (display null string)
            lea     textD2,a1   address of string
            trap    #15         display it
            move.l  d2,d1       put d2 into d1
            move    #3,d0       task number 3 (display number in D0)
            trap    #15         display it
            bsr     newLine

* Display D3
            move    #14,d0      task number 14 (display null string)
            lea     textD3,a1   address of string
            trap    #15         display it
            move.l  d3,d1       put d3 into d1
            move    #3,d0       task number 3 (display number in D0)
            trap    #15         display it
            bsr     newLine

* Display end message
            move    #14,d0      task number 14 (display null string)
            lea     textEnd,a1  address of string
            trap    #15         display it

* Stop execution
            STOP    #$2000


*-------------------------------------
* Subroutine to display Carriage Return and Line Feed
newLine     movem.l d0/a1,-(a7) push d0 & a1
            move    #14,d0      task number into D0
            lea     crlf,a1     address of string
            trap    #15         display return, linefeed
            movem.l (a7)+,d0/a1 restore d0 & a1
            rts                 return


*-------------------------------------
* Variable storage and strings
* ds (define storage) is used to reserve memory for variables
* dc (define constant) is used to place data in memory
sum2        ds.w    1           reserve word of memory for sum2
textD1      dc.b    'D1 contains: ',0    null terminated string
textD2      dc.b    'D2 contains: ',0    null terminated string
textD3      dc.b    'D3 contains: ',0    null terminated string
textEnd     dc.b    'Program End',$d,$a,0null terminated string
crlf        dc.b    $d,$a,0     carriage return & line feed, null

            END     START       end of program with start address specified
```

The 68000 Processor

Code Segment: Example #1

```
START       ORG     $1000       the program will load into address $1000
```

`ORG` sets the origin for the program. This means that the program will be loaded into the memory location with the address `$1000`.

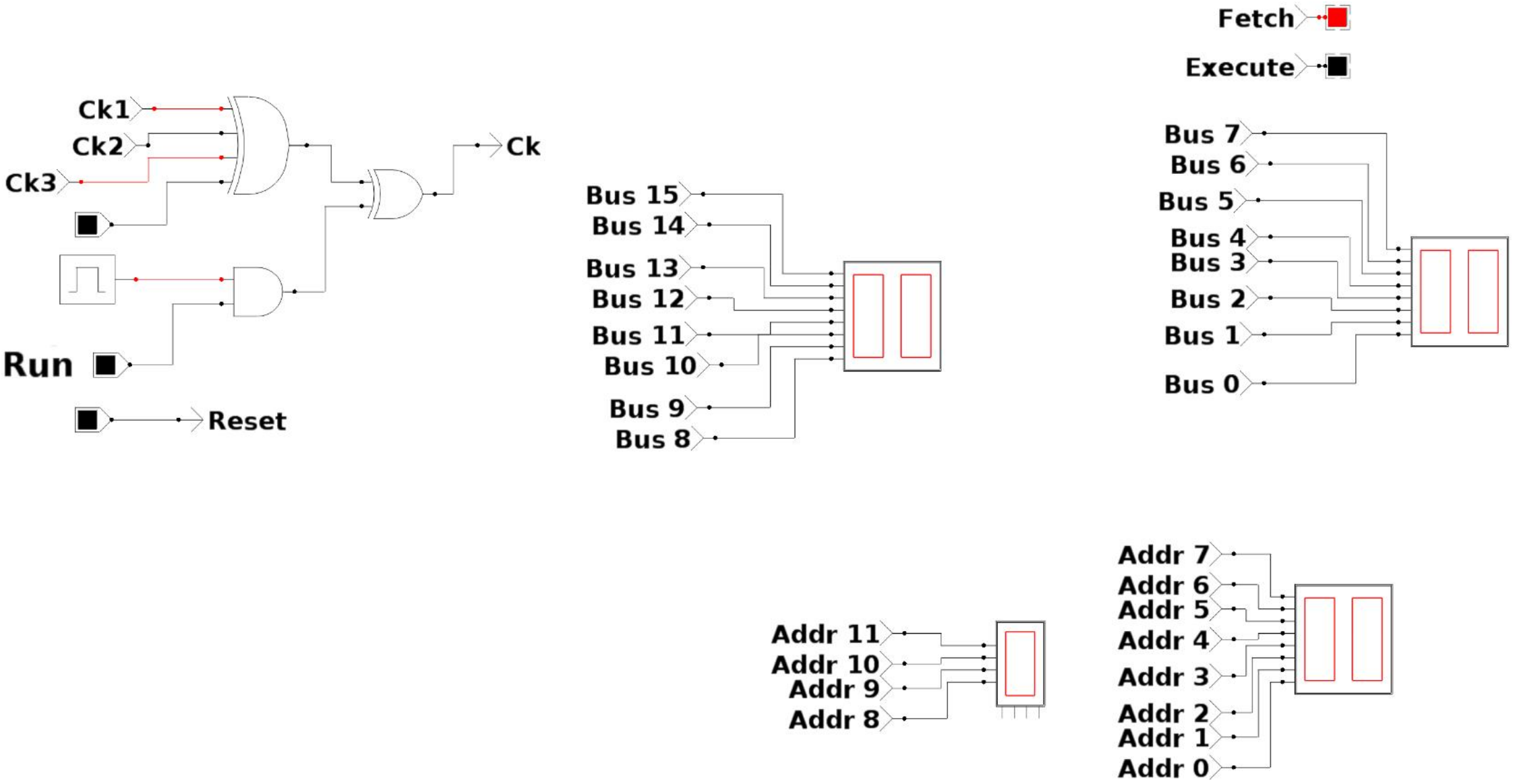
Code Segment: Example #1

```
            move    #3,d0       task number 3 into D0
```

`move` moves

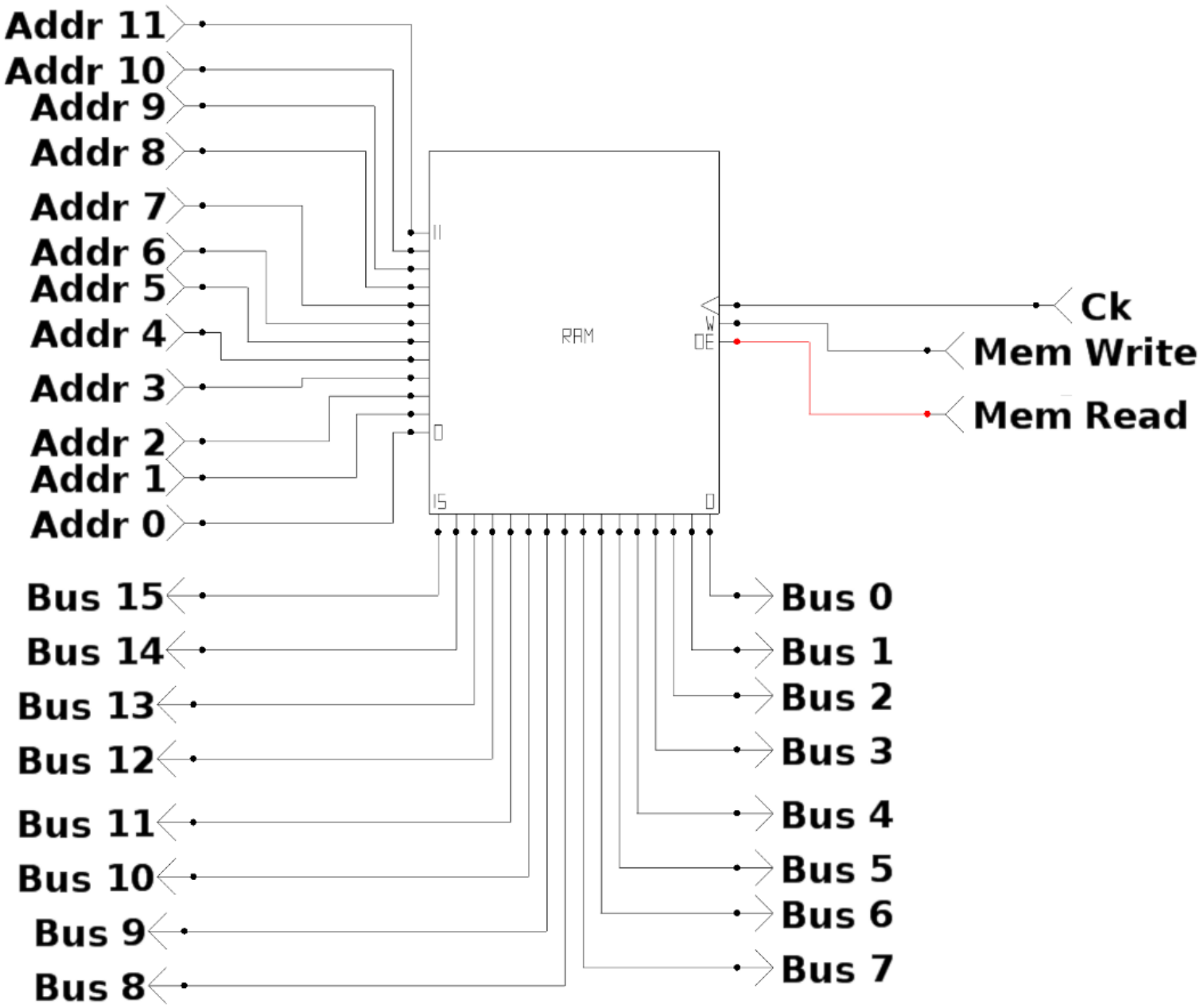
Appendix A – Final Revision Processor Design

Page 1 – Front Panel



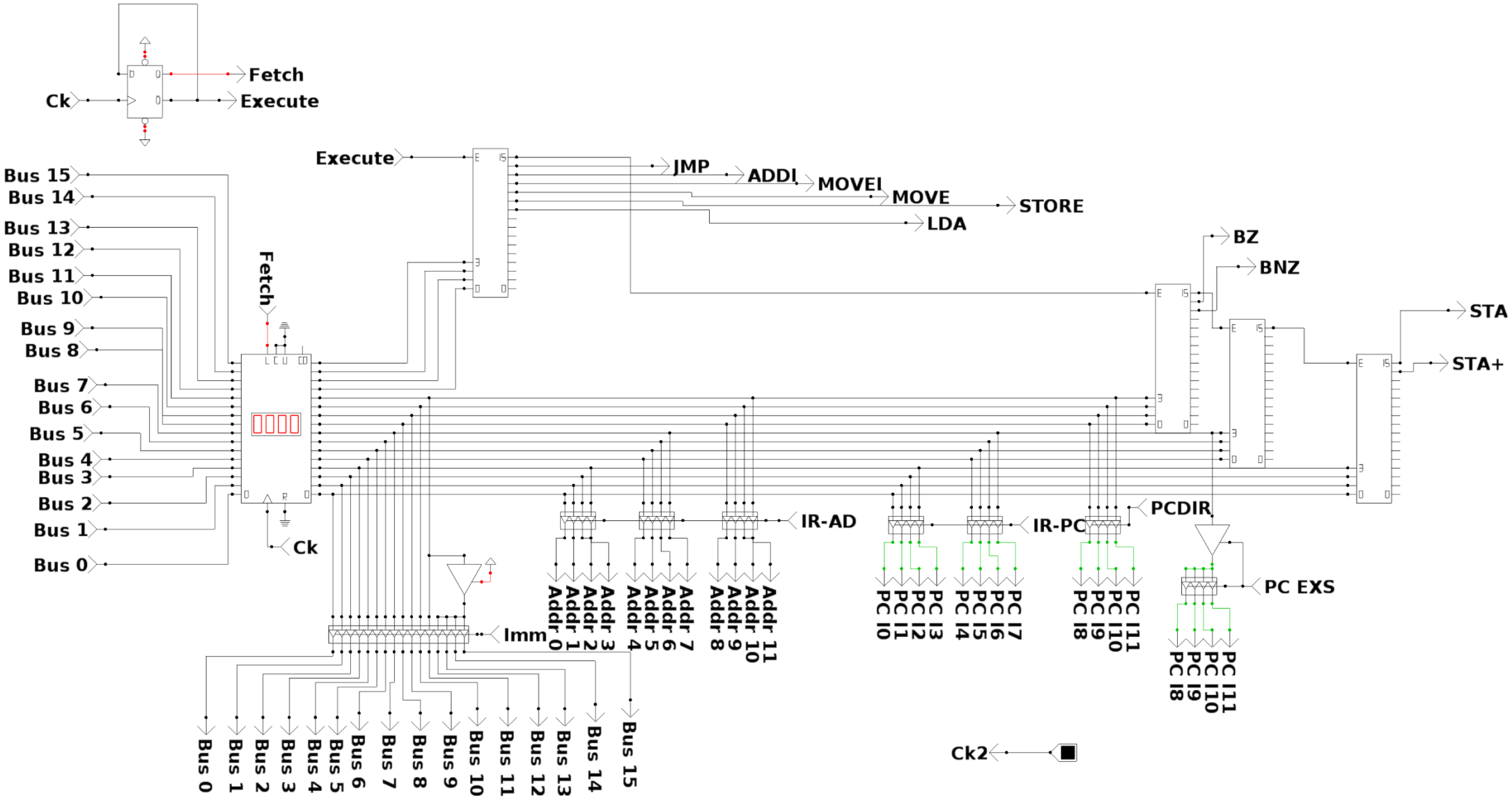
Appendix A – Final Revision Processor Design

Page 2 – Memory



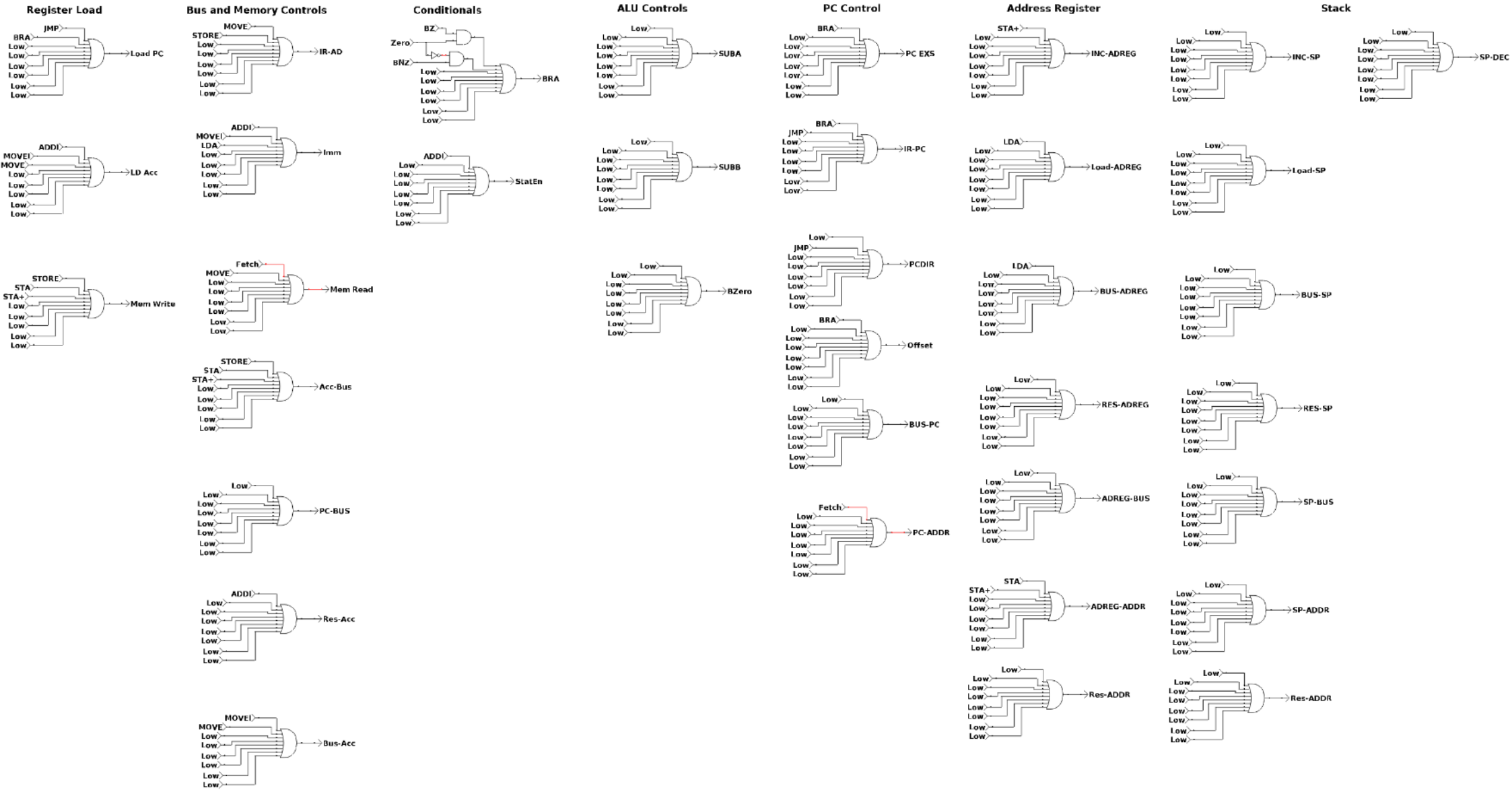
Appendix A – Final Revision Processor Design

Page 3 – Instruction Register and Initial Decode



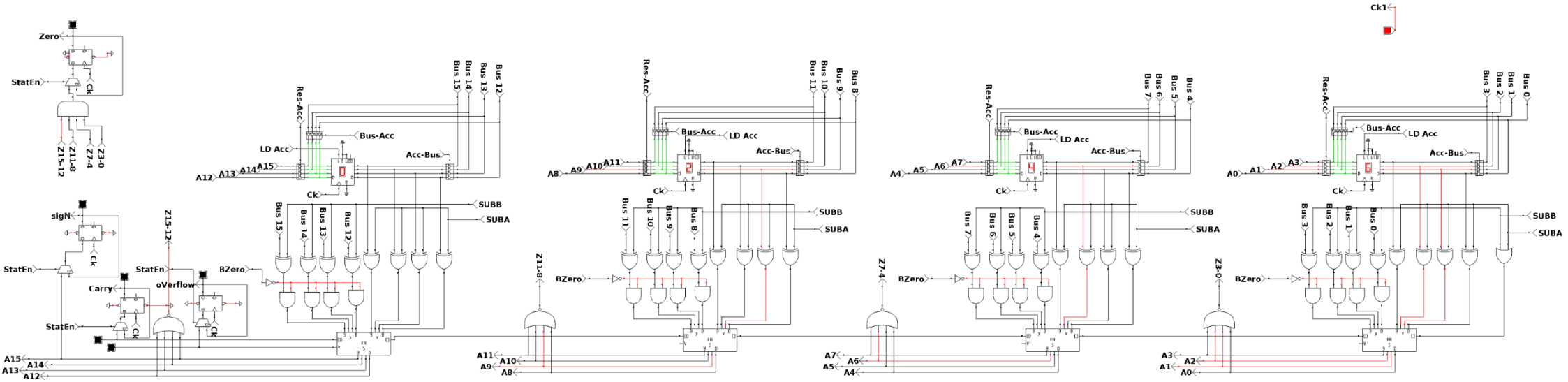
Appendix A – Final Revision Processor Design

Page 4 – Instruction Decode



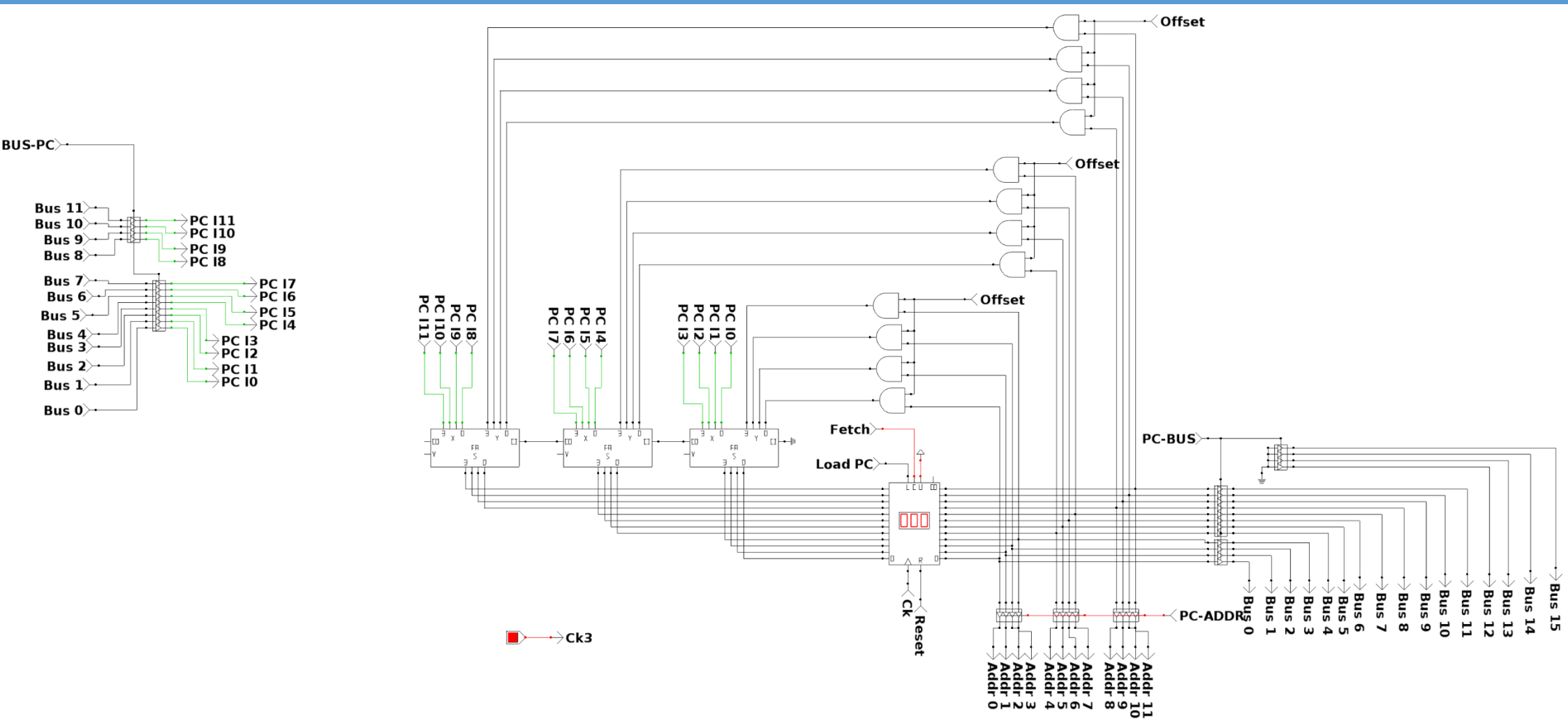
Appendix A – Final Revision Processor Design

Page 5 – Accumulator (arithmetic unit)



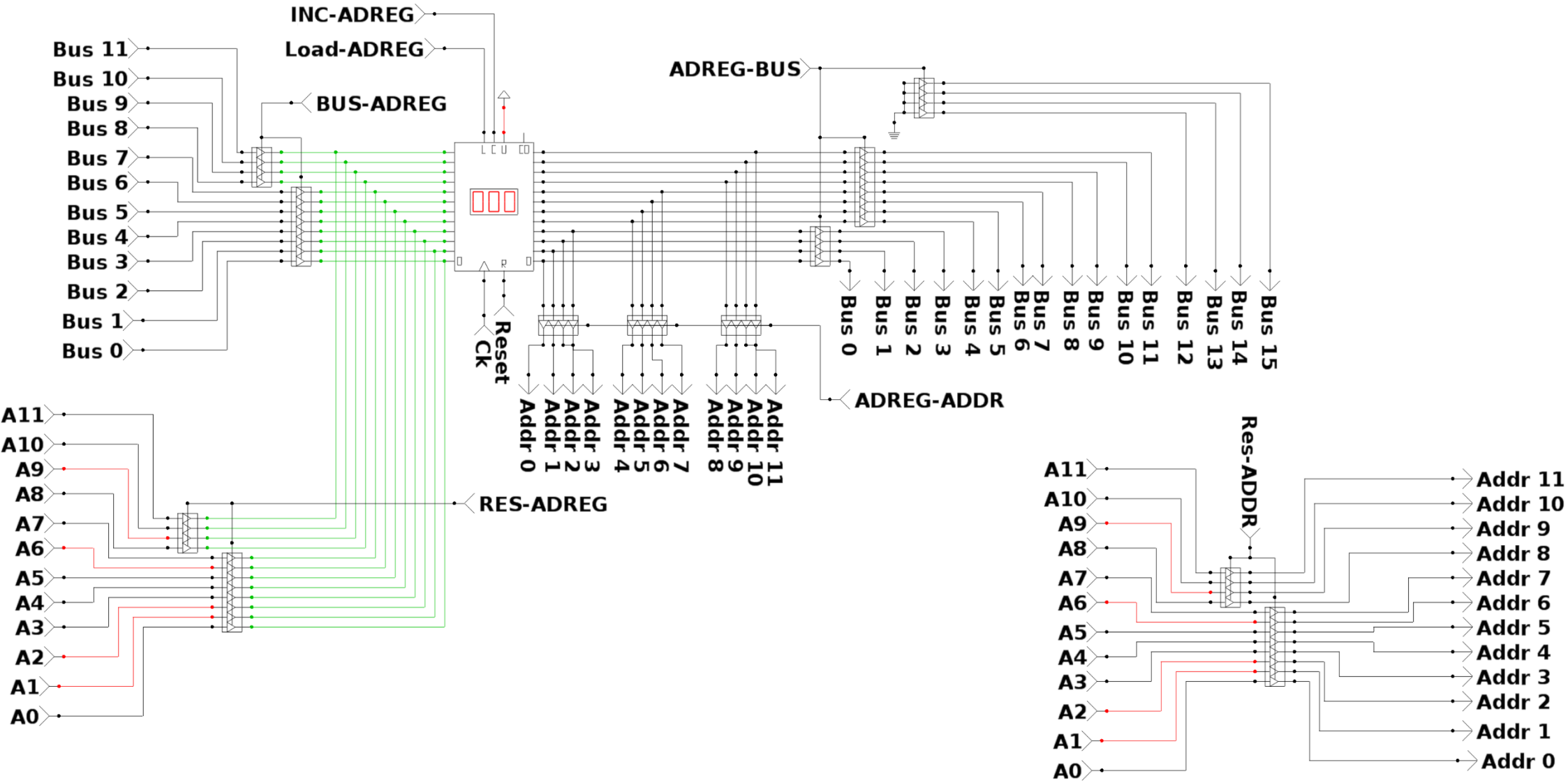
Appendix A – Final Revision Processor Design

Page 6 – Program Counter



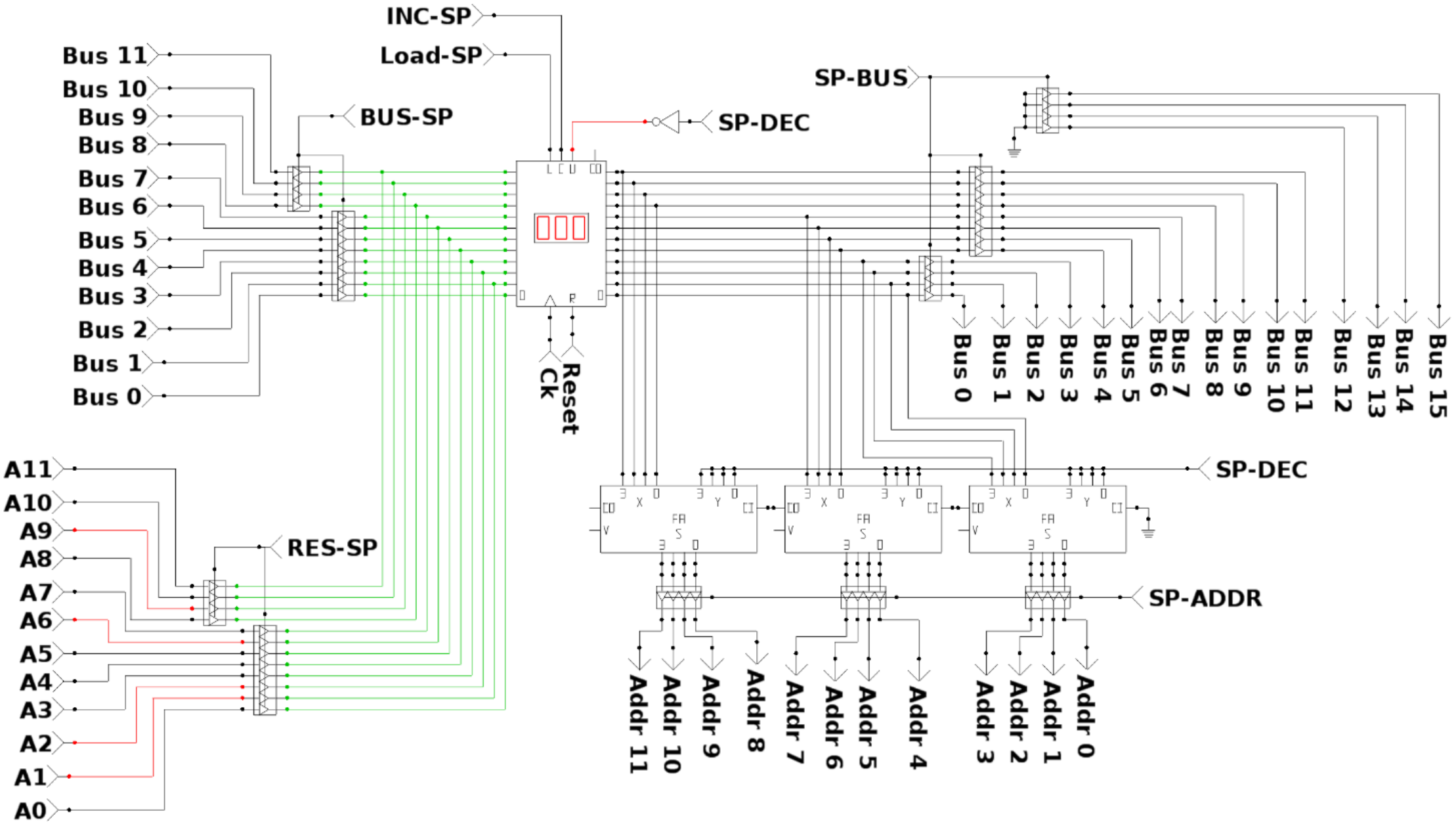
Appendix A – Final Revision Processor Design

Page 7 – Address Register



Appendix A – Final Revision Processor Design

Page 8 – Stack Pointer



Appendix B – Final Revision Processor Instruction Set

Instruction Set: Instructions with 4-bit opcode and 12-bit operand

Binary Value	Hexadecimal Value	Operation	Remarks
1111 $XXXX\ XXXX\ XXXX$	**F**XXX	See below	Enables other formats.
1110 $AAAA\ AAAA\ AAAA$	**E**AAA	JMP	Unconditional jump.
1101 $DDDD\ DDDD\ DDDD$	**D**DDD	$ADDI$	ADD immediate data. $\text{Acc} = \text{Acc} + DDD$
1100 $DDDD\ DDDD\ DDDD$	**C**DDD	$MOVEI$	MOVE immediate data $\text{Acc} = DDD$
1011 $AAAA\ AAAA\ AAAA$	**B**AAA	$MOVE$	MOVE from location. $Acc = contents\ of\ AAA$
1010 $AAAA\ AAAA\ AAAA$	**A**AAA	$STORE$	MOVE to location. $AAA = Acc$
1001 $DDDD\ DDDD\ DDDD$	**9**DDD	LDA	Load an immediate value into the address register
1000 $DDDD\ DDDD\ DDDD$	**8**DDD	JSR	Jump SubRoutine. Load the program counter from a 12-bit immediate value in the instruction, pre-decrement the program counter and use the decremented value of the program counter as a memory address to store the current value of the program counter.
0111 $DDDD\ DDDD\ DDDD$	**7**DDD	ADD_AD+	Add address register with auto-increment. Add to the accumulator a value taken from a memory location that is specified by the address held in the address register and increments the value in the address register.

X – Placeholder
A – Address
D – Data

Instruction Set: Instructions with 8-bit opcode and 8-bit operand

Binary Value	Hexadecimal Value	Operation	Remarks
1111 1111 $XXXX\ XXXX$	**FF**XX	See below	Enables other formats.
1111 1110 $RRRR\ RRRR$	**FE**RR	BZ	Branch if zero (relative).
1111 1101 $RRRR\ RRRR$	**FD**RR	BNZ	Branch if not zero (relative).

X – Placeholder
R – Relative Address

Instruction Set: Instructions with 16-bit opcode and no operand

Binary Value	Hexadecimal Value	Operation	Remarks
1111 1111 1111 1111	**FFFF**	STA	Stores the contents of the accumulator at a memory location specified by the address register.
1111 1111 1111 1110	**FFFE**	$STA+$	Stores the contents of the accumulator at a memory location specified by the address register and then increments the address register.

Appendix C – Control Signals

Control Signal Label	Description
Major Buses	
$Bus\ 0$ to $Bus\ 15$	Main Data bus.
$Addr\ 0$ to $Addr\ 15$	Main Address bus.
$PCI\ 0$ to $PCI\ 11$	Program Counter Input bus.
$A\ 0$ to $A\ 15$	Adder Output bus.
Main Timing Controls	
Ck	Clock – controls overall timing.
$Ck1$ to $Ck3$	Alternative clock switches to allow clock to be switched from other pages.
$Reset$	Reset program counter to 0.
Run	Enable continuous clocking.
$Fetch$	Active $high$ when an instruction is being fetched.
$Execute$	Active $high$ when an instruction is being executed.
Memory	
$Mem\ Read$	Memory read – enables the memory output onto the main data bus
$Mem\ Write$	Memory write – write the content of the main data bus into the currently addressed memory location
Instruction Register	
Imm	Immediate data – enables the bottom 12 bits of the instruction register on to the data bus top four bits are sign extended (copied from the top bit of the 12).
$IR - AD$	Instruction register to address enables the bottom 12 bits of the instruction register onto the address bus.
$IR - PC$	Instruction register to program counter. Enables the bottom 8 bits of the instruction register onto the program counter input bus.
$PCDIR$	Program counter direct. Enable bits 8-11 of instruction register on to program counter input bus.
$PC - EXS$	Program counter extend sign. Top 4 bits of program counter input are sign extended from bit 7 of instruction register.
Accumulator (Data Register)	
$Bus - Acc$	Bus to accumulator – enable main data bus onto accumulator input.
$Res - Acc$	Result to accumulator – enable adder result (A0-A15) onto accumulator input.
$Acc - Bus$	Accumulator to bus.
$LD - Acc$	Load Accumulator (from whatever input is enabled).
$Stat\ En$	Status Enable – status register bits will be loaded according to the outcome of the current operation. This must be $high$ when an arithmetic operation is taking place.
$SUBA$	Negate the Accumulator input to the adder so that accumulator value is subtracted from the bus value.
$SUBB$	Negate the Bus input to the adder.
$BZERO$	Zero the bus input to the adder.
Program Counter	
$BUS - PC$	Route bus to Program counter input.
$PC - BUS$	Route program counter to main bus.
$Load\ PC$	Load the program counter. Note that the PC is loaded from the outputs of the adder next to it.
$Offset$	When true the adder next to the program counter is used to compute a value relative to the current Program counter – when false the value on the second adder input is zero and so the value on the PC I bus is loaded directly.
$PC - ADDR$	Program Counter to address bus – use the program counter as an address from memory, such as in a fetch operation.
Address Register	
$BUS - ADREG$	Bus to address register – route main data bus to address register inputs.
$Load - ADREG$	Load enable for address register.
$RES - ADREG$	Result from main adder is routed to address register inputs.
$RES - ADDR$	Result from main adder is routed to address bus.
$ADREG - ADDR$	Address register is routed to address bus.

Appendix C – Control Signals

$ADREG - BUS$	Address register is routed to main data bus.
$INC - ADREG$	Increment address register.
Stack Pointer	
$BUS - SP$	Bus to stack pointer – route main data bus to stack pointer inputs.
$RES - SP$	Result from main adder is routed to stack pointer inputs.
$SP - BUS$	Stack pointer is routed to main data bus.
$SP - ADDR$	Stack pointer is routed to address bus.
$INC - SP$	Increment or decrement stack pointer (depending on the value of SP-DEC).
$SP - DEC$	The stack pointer is decremented if INC-SP is also true.
$Load - SP$	Load stack pointer from its inputs.
Control/Status Bits	
$Carry$	Output from carry status bit.
$sigN$	Output from sign status bit.
$oVerflow$	Output from overflow status bit.
$Zero$	Output from Zero status bit.
Miscellaneous	
BRA	Signal that will be $high$ whenever a conditional branch instruction is active and the condition is $high$.
Decoded Instructions	
JMP	Decoded jump instruction.
$ADDI$	Decoded add immediate instruction.
$MOVEI$	Decoded move immediate instruction.
$MOVE$	Decoded move instruction (direct address).
$STORE$	Decoded store instruction.
LDA	Decoded load address register instruction.
BZ	Decoded branch if zero instruction.
BNZ	Decoded branch if not zero instruction.
STA	Decoded store accumulator at address register (indirect) instruction.
$STA +$	Decoded store accumulator at address register (indirect) with auto increment of address register instruction.