Building a Computer From Scratch

Introduction to Computer Architecture (with exercises)

Stephan E. Korsholm

August 2016

Contents

1 Introduction

This document describes how to build a 4 bit micro controller (called the *minimic*) from scratch. The design can easily be extended to 8 or more bits. The amount of mathematical (and other) theory introduced will be just enough to understand how computers function at the lowest levels. The micro controller can be realized in (1) actual electronics, (2) using an FPGA or (3) using a simulator. In the exercises the computer is built using a simulator written in Java.

The purpose of the text and exercises is to give the reader a detailed understanding of how Turing complete computational machines are built.

2 The Binary Numbering System

2.1 Numbers and Computers

Computers are usually built using electronic circuits. This has so far proven to yield the smallest, fastest and most reliable computers. Previously efforts have been made to build computers using gears, cranks and levers, and in theory it should be possible to build advanced architectures using such methods. In the future it may be possible to build computers using e.g. biological materials, or chemical or other mediums. How a computer is realized is a separate issue from how it is designed, and here we will focus on the design, but make sure to create a design that can be easily implemented in any medium.

Computers are machines that can perform fast operations on numbers. They can e.g. add numbers or check if a number is zero. They can store numbers into memory and retrieve them later for further computation. Computers cannot store text, names, music, books or other kinds of information - they can only store numbers. This means that if we want the computer to process names, music, books or other kinds of information, we first have to translate this information into a number representation.

In modern day computers every single informational artifact that the computer works with is translated into a sequence of numbers before it is being processed. As an example let us consider how a computer stores text. The text “Hello” cannot be stored directly as 5 letters, and if you look into the memory section (using a microscope e.g.) where the word “Hello” is stored you will not find 5 letters, instead you will find 5 numbers. There is a commonly agreed upon definition of how to translate letters into numbers. This definition is given in the ASCII table in Figure 1.

0	<NUL>	32	<SPC>	64	@	96	`	128	Ä	160	†	192	¿	224	‡
1	<SOH>	33	!	65	A	97	a	129	Å	161	°	193	¡	225	·
2	<STX>	34	"	66	B	98	b	130	Ç	162	¢	194	¬	226	‚
3	<ETX>	35	#	67	C	99	c	131	É	163	£	195	√	227	„
4	<EOT>	36	$	68	D	100	d	132	Ñ	164	§	196	ƒ	228	‰
5	<ENQ>	37	%	69	E	101	e	133	Ö	165	•	197	≈	229	Â
6	<ACK>	38	&	70	F	102	f	134	Ü	166	¶	198	∆	230	Ê
7	<BEL>	39	'	71	G	103	g	135	á	167	ß	199	«	231	Á
8	<BS>	40	(	72	H	104	h	136	à	168	®	200	»	232	Ë
9	<TAB>	41	)	73	I	105	i	137	â	169	©	201	…	233	È
10	<LF>	42	*	74	J	106	j	138	ä	170	™	202		234	Í
11	<VT>	43	+	75	K	107	k	139	ã	171	´	203	À	235	Î
12	<FF>	44	,	76	L	108	l	140	å	172	¨	204	Ã	236	Ï
13	<CR>	45	-	77	M	109	m	141	ç	173	≠	205	Õ	237	Ì
14	<SO>	46	.	78	N	110	n	142	é	174	Æ	206	Œ	238	Ó
15	<SI>	47	/	79	O	111	o	143	è	175	Ø	207	œ	239	Ô
16	<DLE>	48	0	80	P	112	p	144	ê	176	∞	208	–	240	
17	<DC1>	49	1	81	Q	113	q	145	ë	177	±	209	—	241	Ò
18	<DC2>	50	2	82	R	114	r	146	í	178	≤	210	“	242	Ú
19	<DC3>	51	3	83	S	115	s	147	ì	179	≥	211	”	243	Û
20	<DC4>	52	4	84	T	116	t	148	î	180	¥	212	‘	244	Ù
21	<NAK>	53	5	85	U	117	u	149	ï	181	µ	213	’	245	ı
22	<SYN	54	6	86	V	118	v	150	ñ	182	∂	214	÷	246	ˆ
23	<ETB>	55	7	87	W	119	w	151	ó	183	∑	215	◊	247	˜
24	<CAN>	56	8	88	X	120	x	152	ò	184	∏	216	ÿ	248	¯
25	<EM>	57	9	89	Y	121	y	153	ô	185	π	217	Ÿ	249	˘
26	<SUB>	58	:	90	Z	122	z	154	ö	186	∫	218	⁄	250	˙
27	<ESC>	59	;	91	[	123	{	155	õ	187	ª	219	€	251	˚
28	<FS>	60	<	92	\	124	|	156	ú	188	º	220	‹	252	¸
29	<GS>	61	=	93	]	125	}	157	ù	189	Ω	221	›	253	˝
30	<RS>	62	>	94	^	126	~	158	û	190	æ	222	ﬁ	254	˛
31	<US>	63	?	95	_	127	<DEL>	159	ü	191	ø	223	ﬂ	255	ˇ

Figure 1: The ASCII Table

The ASCII table defines that the letter “H” is translated into the number 72. Why it is 72 is irrelevant. It could have been any other number. What is important is that all parties involved in processing the text “Hello” agrees on using the same translation table (the ASCII table).

In a similar way - which we will not cover here - any other piece of information, e.g. a sound, a picture or information about a customer, will have to be translated into a sequence of numbers in order for a computer to be able to work with that information.

2.2 Storing Numbers

In a computer the data the computer works with is stored as numbers in its memory. As it operates it will compare and change the numbers according to its program. An interesting question is how the numbers are actually stored? In this section we will look at how that can be done using electrical circuits. In electrical circuits a wire can have a certain voltage level, e.g from 0 to 5 Volts. One could imagine to divide that range up into e.g. 255 sub intervals. Then a single wire could be used to store an ASCII number. If using that strategy there will be some upper limit to how many sub intervals we would be able to distinguish, and thus how many numbers we would be able to store on a single wire. At the other end of the spectrum the smallest piece of information that can be stored on a wire is if we divide the full spectrum from 0 to 5 V into only two parts: 0 - 2.5 and 2.5 to 5 V. The lower part we could call *lower* and the upper part we could call *upper*. Using this scheme it is clear that we would be able to store only two numbers using one wire, namely 0 - which would be the same as *lower* - and 1 - which would be the same as *upper*.

Computers use this exact scheme: If the voltage level on a wire is close to 0 it means that the number 0 is stored there, and if the voltage level on a wire is close to 5 V, it means that the number 1 is stored there. This is depicted in Figure 2.

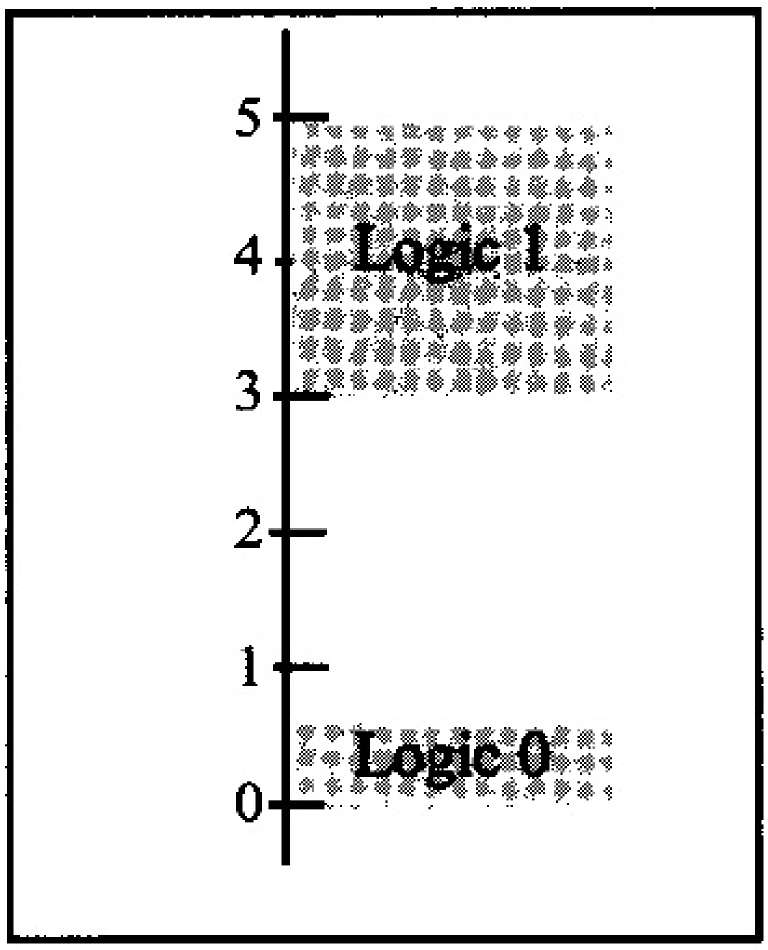


Figure 0-2. Binary Signals

Figure 2: Storing 0/1 On a Wire

Being able to only store the number 0 or 1 is not enough for most uses. E.g. when storing text as numbers using the ASCII table we need at least 28 numbers if we want to store the lower case letters of the alphabet. The way computers solve this issue is by using several wires - each storing a 0 or 1 - to store a larger number. The question now arises how to turn a decimal number, e.g. 72 into to sequence of 0's and 1's. We must come up with a method for translating any decimal number into a sequence to 0's and 1's in such a way that we can translate the number back to decimal in only one way.

If we can translate the number 72 into such a sequence, say 01001000, we can use 8 wires to store that number. The method must have an inverse way to translating the number 01001000 into decimal, and this inverse way must yield the number 72 and that number only.

2.3 Numbering Systems

When talking about numbers - e.g. 72 - we are normally using the decimal numbering systems. The term *deci* is Latin and means 10. The decimal numbering system is called just that because each number is a sequence of one of 10 different digits, namely 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. The position of each digit is important, e.g in the number 72 the digit 7 counts the number of tens and the digit 2 counts the number of ones. The number 72 is 7 tens plus 2 ones. In mathematical terms, if a digit d is positioned at position n (counting from the right) that digit contributes the amount of $d * 10^n$. Thus the number 72 becomes $7 * 10^1 + 2 * 10^0$. The magnitude of any decimal number is calculated in a similar manner, e.g $4096 = 4 * 10^3 + 0 * 10^2 + 9 * 10^1 + 6 * 10^0$.

Now, let's assume we have only two digits, 0 and 1. We could then define a binary numbering system in a very similar manner: if a digit d is positioned at position n (counting from the right) that digit contributes the amount of $d * 2^n$. Notice that the number 10 has been substituted for the number 2. This number is called the base of the numbering system. The decimal numbering system uses base 10, the binary numbering system uses the base 2. Consider the binary number 01001000, what is the magnitude of that number? Using the definition from above we get

$$\begin{aligned} 01001000 &= 0 * 2^7 + 1 * 2^6 + 0 * 2^5 + 0 * 2^4 + 1 * 2^3 + 0 * 2^2 + 0 * 2^1 + 0 * 2^0 \\ &= 0 * 128 + 1 * 64 + 0 * 32 + 0 * 16 + 1 * 8 + 0 * 4 + 0 * 2 + 0 * 1 \\ &= 1 * 64 + 1 * 8 \\ &= 72 \end{aligned} \tag{1}$$

This method of using a sequence of 0's and 1's to represent numbers is called the binary numbering system and has some very good properties:

1. For every distinct binary number representation there is only 1 corresponding decimal number representation

2. For every distinct decimal number representation there is only 1 corresponding binary number representation[1]

In a similar way you could store numbers using any base, e.g tertiary numbers (base 3) or octal numbers (base 8).

Computers use the binary numbering system to store numbers. Each binary digit of 0 or 1 is stored using a voltage level on a wire. If you store the number 72 into a memory cell in an electronic computer and used a microscope to look at the memory cell, you would actually see at least 8 wires with the voltage levels low, high, low, low, high, low, low low (because $72_{10} = 01001000_2$).

The reason why computers use the binary numbering system is only because experience has shown that it is easier to build the electronics that way. You could build a computer using e.g. the tertiary numbering system or the octal numbering system, if you divided the voltage level on a wire into 3 and 8 sub intervals respectively. Such computers could be built, but since all computer manufactures so far have used the binary numbering system, we will do so as well and design the *minimic* to store and operate on binary numbers.

2.4 Bits, Bytes and Words

In the decimal numbering system, if using 4 digits at the most, the largest number one can represent is the number 9999. When designing a new computer using the binary numbering system an important design decision is to decide how many digits (binary digits) can be in a single number. The computer designer must decide on that before production of the computer, because it is impossible to add more wires once the computer has been built.

If using 8 binary digits, we say that the designer has decided on an 8 bit architecture. Today computers are normally 8, 16, 32 or 64 bit architectures. You could easily build e.g. 23 bit or 37 bit architectures if that made sense in terms of your problem domain. The first danish computer (called DASK) was a 42 bit architecture.

A number consisting of 8 bits is called a byte. A number consisting of 16 bits is called a word. Sometimes 32 bit numbers are called long (or double) words and 64 bit numbers are called quad words.

The *minimic* is a 4 bit architecture. This means that all numbers the computer can work with is in the range from 0 to 15. It follows that the longest program we can make can contain at most 16 instructions.

2.5 Exercises

2.5.1

According to the ASCII table (see Figure 1) what sequence of numbers will be used to represent the text “Hello World!”?

[1]These two properties only hold if ignoring leading 0’s in the sense that 0001123 is the same as 1123 and the same for binary numbers

2.5.2

Given two names - each as a sequence of numbers according to the ASCII table - how will a computer determine if the two names are identical?

2.5.3

Let's assume that some program needs to store some information about a person. That information will be the first name, the last name, the age and the gender of the person. How can such information about a person be translated into a sequence of numbers?

2.5.4

Translate the following decimal numbers into binary:

- 72 =
- 37 =
- 2 =
- 1 =
- 15 =
- 0 =
- 8 =
- 4 =
- 4096 =
- 16383 =

2.5.5

Translate the following binary numbers into decimal:

- 1111 =
- 11111111 =
- 1010 =
- 001100 =
- 0111001 =
- 1000001 =

- 1 =
- 0 =
- 100000000 =
- 10000 =

2.5.6

Translate the following decimal numbers into octal (base 8):

- 72 =
- 37 =
- 2 =
- 1 =
- 15 =
- 0 =
- 8 =
- 4 =
- 4096 =
- 16383 =

2.5.7

What is the largest decimal number you can represent using,

- 4 bits?
- 8 bits?
- 16 bits?
- 32 bits?
- 64 bits?

Why do you think 128 bit computers are uncommon? Why do you think 64 bit computers where introduced?

3 Binary Number Arithmetic

Just as a number of operators (+, -, *, /, . . .) are defined for decimal numbers, so are the same (and a number of new ones) defined for binary numbers.

$$\begin{array}{r} 1 \\ 123 \\ 92 \\ \hline 215 \\ \hline \hline \end{array}$$

Figure 3: Decimal Number Addition

Figure 3 illustrates how to add the two decimal numbers 123 and 92. Starting from the right we first add 3 to 2 yielding 5 which is written below to the right. Then we add the numbers 2 and 9 yielding the result 11. Now only the number 1 is written on the result line, the 10 portion of the result 11 is carried over and written on top. This carry is included in the next addition where we add 1 (the carry) to 1 yielding 2, and thus we finally get the result of 215.

To sum up, we add two decimal numbers by adding pairs of digits one by one. This way we break up the full addition into smaller single digit additions. Using this method we are all able to add large numbers together without having to be able to do it in one step. The digit addition method takes two decimal digits as input and produces a sum and a carry. From the example above we see that 2 + 9 yield a sum of 1 and a carry of 1. The way each digit is added is listed in Table 1. This table lists all the possible input to decimal digit addition and the sum and carry results. Parts of the table have been left out.

d_0	d_1	**Sum**	**Carry**
0	0	0	0
0	1	1	0
0	2	2	0
0	3	3	0
...	...	...	...
1	0	1	0
1	1	2	0
1	2	3	0
...	...	...	...
1	9	0	1
2	0	2	0
2	1	3	0
...	...	...	...
2	9	1	1
...	...	...	...
9	9	8	1

Table 1: Decimal Digit Addition

Notice the use of the rule adding the digit 2 to the digit 9 yielding a sum of 1 and a carry of 1. This rule was used in the addition performed in Figure 3.

The table of rules for binary digit addition is fortunately a lot simpler. Since there is only 2 digits d_0 and d_1 can only come in 4 possible combinations (and not 100 as with decimal digit addition). The full set of rules are listed in Table 2.

d_0	d_1	**Sum**	**Carry**
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Table 2: Binary Digit Addition

Using the set of rules for binary digit addition in Table 2 we can add large binary numbers by applying the same rules from right to left - just as we did with decimal numbers. Figure 4 illustrates how to add the binary numbers 01111011 to 01011100.

```
 1111
 01111011
 01011100
----------
 11010111
==========
```

Figure 4: Binary Number Addition

The first 4 binary digit additions (going from right to left) are covered by Table 2. But the 5th binary addition is not. Because of the carry from the 4th addition, we now must add 3 binary digits $1 + 1 + 1$. This is not covered by Table 2. We see that we must extend the table to handle 3 binary digit inputs. This extension has been performed in Table 3.

$Carry_{in}$	d_0	d_1	**Sum**	**Carry**
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

Table 3: 3 Input Binary Digit Addition

Using the rules in Table 3 we are now finally able to add any pair of binary numbers by applying the rules from right to left.

3.1 Binary operators

Apart from addition a number of other operators have been defined for binary digits. Tables 4, 11, 6 and 7 list some of the more common operators.

d_0	d_1	**And**
0	0	0
0	1	0
1	0	0
1	1	1

Table 4: Binary And

d_0	d_1	**Nand**
0	0	1
0	1	1
1	0	1
1	1	0

Table 5: Binary Nand

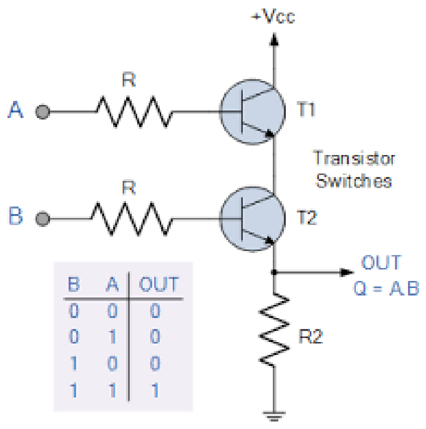
d_0	d_1	**Or**
0	0	0
0	1	1
1	0	1
1	1	1

Table 6: Binary Or

d_0	d_1	**Xor**
0	0	0
0	1	1
1	0	1
1	1	0

Table 7: Binary Xor

In general, a binary operator taking 2 binary inputs (each 0 or 1) and producing 1 binary output (0 or 1) is called a *gate*. It just so happens that gates are easy to build using electrical components. Figure 5 shows how to build an and gate using two transistors and three resistors. In the following section we will see how gates can be used to implement binary addition. This in turn means that it is easy to build an electrical circuit that performs binary addition.



B	A	OUT
0	0	0
0	1	0
1	0	0
1	1	1

Figure 5: The And Gate as Electronics

3.2 Exercises

3.2.1

Fill out the missing rules for decimal digit addition in Table 1.

3.2.2

Manually add the following binary numbers:

- $10101010 + 01010101$
- $10101011 + 01010101$
- $11110000 + 00001111$
- $11111110 + 01010101$
- $00010101 + 10110101$

3.2.3

Apart from the 4 binary operators defined in Tables 4, 11, 6 and 7 how many different binary operators exist? Remember, that each binary operator take 2 binary inputs (each 0 or 1) and produce 1 binary output (0 or 1). How many such functions can be defined?

3.2.4

How would you build an And gate using wood? Imagine two levers going into a box, and one lever sticking out of it. The goal now is to combine pieces of wood inside the box such that the lever sticking out would only be raised if both

levers going in are pressed. We will imagine that we can use nails and strings as well to build the gates, if required.
How would you build an Or gate using wood?
How would you build a Xor gate using wood?

4 Half and Full Adders

Adding 8/16/32 or 64 bit numbers is a very important operation to be able to perform by a computer. Addition is also the basis for other operations. E.g. addition can be used to implement multiplication. Addition can also be used to implement subtraction (as will be described in a later section). Since adding numbers is so important this section will describe how a circuit can be built from gates to add binary numbers. We will restrict ourselves to adding 4 bit binary numbers, but it is very easy to extend the methods to apply to 8/16/32 or 64 bit architectures as well.

Lets look again at the table defining how to add two binary digits (see Table 8).

d_0	d_1	**Sum (Xor)**	**Carry (And)**
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Table 8: Binary Digit Addition (with gates)

Looking closely at the columns for the Sum output and the Carry output, we see that the values are the same as the outputs for the xor gate and the and gate respectively. This means that we can build a circuit performing the addition of two binary digits using an xor and an and gate. This circuit is shown in Figure 6.

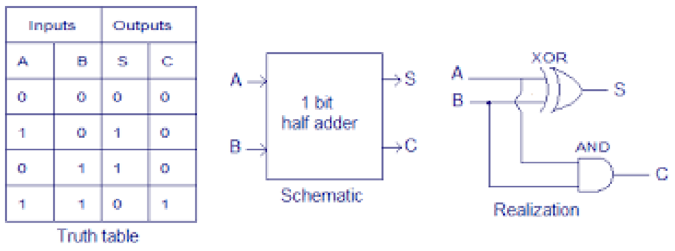

Inputs		Outputs	
A	B	S	C
0	0	0	0
1	0	1	0
0	1	1	0
1	1	0	1

Figure 6: Half Adder

This circuit is called a *half adder*. It takes 2 binary digits as input and produces 2 outputs: the sum and the carry. Section 3 described how we needed to add 3 binary digits in order to perform a full binary addition. So we need to extend the half adder circuit to take 3 binary digits as input while still producing 2 binary output results (the sum and the carry). Such a circuit must implement the rules of Table 3 in Section 3. This extended circuit is called a *full adder* and it can be built by combining two half adders as is illustrated in Figure 7.

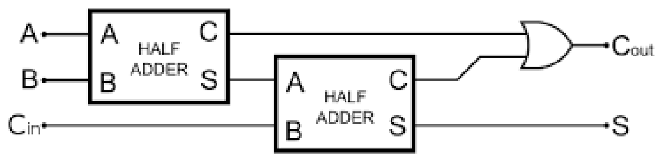


Figure 7: Full Adder

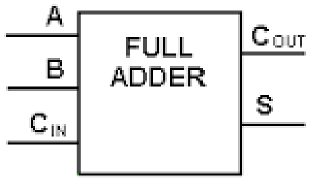


Figure 8: Full Adder Schematic

Using the full adder we can now build a computer that can add two binary digits (each 0 or 1) and produce a sum and a carry. But what if we want to add two 4 bit binary numbers, e.g. 7 + 3 (which will be 0111 + 0011)? This function will take 8 bits coming in (4 bits in each number), let's call these input bits $x_3x_2x_1x_0$ and $y_3y_2y_1y_0$ and 4 bits of sum going out (lets call this $s_3s_2s_1s_0$) and possibly a carry going out from the full addition. The abstract schematic of this component is illustrated in Figure 9.

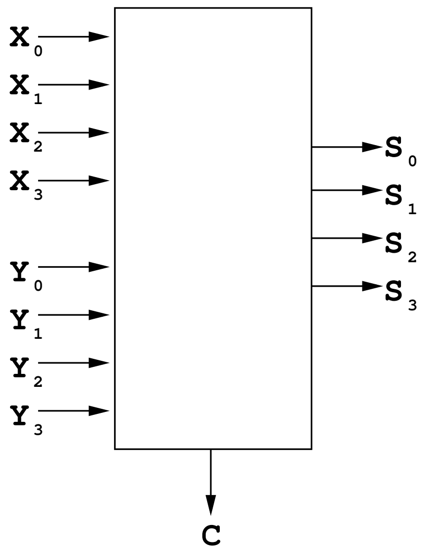


Figure 9: 4 bit ALU

Just like two half adders can be combined to form a single full adder, it is also possible to combine 4 full adders to form a single 4 bit binary adder. Such a component is called a 4 bit ALU (Arithmetical Logical Unit). How to wire 4 full adders together to realize the abstract schematic of the 4 bit ALU in Figure 9 is left as an exercise.

We have now seen how transistors and resistors can be combined to build e.g. and and xor gates. These gates can in turn be used to build half adders. Half adders can be combined to build full adders and full adders can be combined to build a 4 bit ALU. It is thus possible to build an electrical circuit that, if applied two 4 bit binary numbers to it's input lines, will automatically produce the result of adding the two input numbers.

This operation is at the core of all computers. Still there is some way to explaining how computers execute larger programs containing other operations than just addition, and this is the topic of the remainder of the document.

4.1 Boolean Arithmetic

The combination of two half adders to form a full adder is simple and easy to understand, but there are other ways of building a full adder. To illustrate a general method of building circuits that implement a specific set of Boolean rules, we will here make another version of the full adder. Figure 9 shows the table defining the value of the sum output of the full adder (we ignore the carry output for now).

A	**B**	**C**	**Sum**
0	0	0	0
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

Table 9: Sum part of full adder

This table defines a Boolean function of three variables A, B and C. The function can be written in the following manner:

$$f(A,B,C) = \bar{A} * \bar{B} * C + \bar{A} * B * \bar{C} + A * \bar{B} * \bar{C} + A * B * C \tag{2}$$

In the expression above the multiplication stands for the *and* operator and the addition sign stands for the *or* operator. Usually the multiplication operator is omitted, and the function is written as:

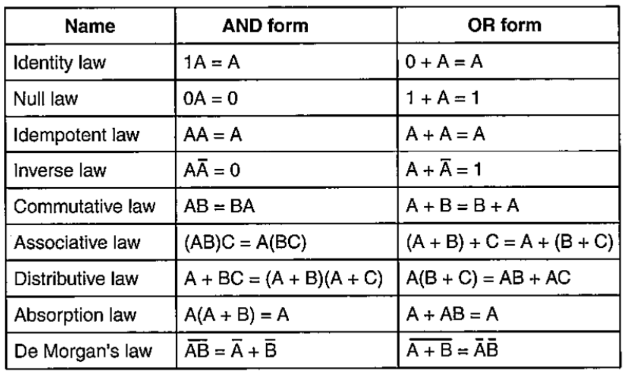
$$f(A,B,C) = \bar{A}\bar{B}C + \bar{A}B\bar{C} + A\bar{B}\bar{C} + ABC \tag{3}$$

The meaning of (2) and (3) is exactly the same, but we will use (3) because it is shorter. The method to convert from the table form in Table 9 into the equation form in (3) is the following:

1. Find all entries in the table that has as output 1. This is the case in 4 entries in Table 9
2. For each of these entries a term made up of the inputs will be added to the final equation. If the input A is 0, it appears as $\bar{A}$, if it is 1 it appears as A. For the first entry where the Sum is 1 in Table 9 the value of A and B is 0 and the value of C is 1. Thus the term for this entry becomes $\bar{A}\bar{B}C$

In this manner a Boolean function listed in table form can be transferred into equation form. These two ways of expressing a Boolean function describes the same Boolean function.

Once a Boolean function has been expressed in equation form, it may be possible to reorganize the terms in the equation according to the rules of Boolean algebra. The purpose of such a reorganization is to make the equation shorter, while still expressing the same function. A shorter expression can more easily be realized in hardware using Boolean gates.



Name	AND form	OR form
Identity law	1A = A	0 + A = A
Null law	0A = 0	1 + A = 1
Idempotent law	AA = A	A + A = A
Inverse law	$A\bar{A} = 0$	$A + \bar{A} = 1$
Commutative law	AB = BA	A + B = B + A
Associative law	(AB)C = A(BC)	(A + B) + C = A + (B + C)
Distributive law	A + BC = (A + B)(A + C)	A(B + C) = AB + AC
Absorption law	A(A + B) = A	A + AB = A
De Morgan's law	$\overline{AB} = \bar{A} + \bar{B}$	$\overline{A + B} = \bar{A}\bar{B}$

Figure 10: Laws of Boolean Algebra

Figure 10 lists all the laws of Boolean algebra. Using some of the laws on (4) above we get:

$$\begin{aligned} f(A,B,C) &= \bar{A}\bar{B}C + \bar{A}B\bar{C} + A\bar{B}\bar{C} + ABC \\ &= \bar{A}(\bar{B}C + B\bar{C}) + A(\bar{B}\bar{C} + BC) \end{aligned} \tag{4}$$

This does not seem significantly shorter, but notice the term $\bar{B}C + B\bar{C}$. The truth table for that is given in Table 10.

B	C		$\bar{B}C + B\bar{C}$
0	0	$1*0+0*1$	0
0	1	$1*1+0*0$	1
1	0	$0*0+1*1$	1
1	1	$0*1+1*0$	0

Table 10: $\bar{B}C + B\bar{C}$

So we see that the expression $\bar{B}C + B\bar{C}$ is actually the same as B *xor* C, which we will write as $B \oplus C$. The other expression $\bar{B}\bar{C} + BC$ can be shown to be equivalent to $\overline{B \oplus C}$ (*not xor*). Thus the equation in(4) can be written as:

$$\begin{aligned} f(A,B,C) &= \bar{A}\bar{B}C + \bar{A}B\bar{C} + A\bar{B}\bar{C} + ABC \\ &= \bar{A}(\bar{B}C + B\bar{C}) + A(\bar{B}\bar{C} + BC) \\ &= \bar{A}(B \oplus C) + A(\overline{B \oplus C}) \end{aligned} \tag{5}$$

This can shortened even further. We know from Table 10 that $\bar{B}C + B\bar{C} = B \oplus C$. In general, if we rename the variables, this is the same as $\bar{X}Y + X\bar{Y} = X \oplus Y$. Now, if we substitute A for X and $B \oplus C$ for Y we see that:

$$
\begin{aligned}
f(A,B,C) &= \bar{A}\bar{B}C + \bar{A}B\bar{C} + A\bar{B}\bar{C} + ABC \\
&= \bar{A}(\bar{B}C + B\bar{C}) + A(\bar{B}\bar{C} + BC) \\
&= \bar{A}(B \oplus C) + A(\overline{B \oplus C}) \\
&= A \oplus B \oplus C
\end{aligned}
\tag{6}
$$

Equation (6) shows us that we can build a circuit to produce the Sum output of a full adder by only using two *xor* gates.

4.2 Simulating Electrical Components in Java

4.3 Exercises

4.3.1

Verify that the circuit in Figure 7 indeed implements the rules in Table 3 in Section 3. For each line in the table apply the digit values to the input lines and trace the output being produced and compare it against the expected output in the table.

4.3.2

Make a drawing of how two full adders can be wired together to form a 2 bit ALU. Extend your drawing to build a 4 bit ALU.

4.3.3

In a similar manner as is done in Table 10, show that the expression $\bar{B}\bar{C} + BC$ is the same as $\overline{B \oplus C}$.

4.3.4

Write the Carry bit of the full adder as a Boolean expression. Use the laws of Boolean algebra to find the smallest expression - and thus the smallest circuit - that computes the carry bit of the full adder.

4.3.5

Using the result from equation (6) and the solution to the previous exercise, write the diagram of the smallest circuit realizing a full adder.

4.3.6

The CircuitSim framework contains tests and implementations of the Xor and And gates. Download, import and run the existing tests in the CircuitSim package available on the course home page. Add an Or gate including a test for the Or gate.
Solution available: `https://youtu.be/4E1v6TwA9gk`

4.3.7

Figure 6 illustrates how to build a half adder using an Xor and an And gate. The CircuitSim simulation framework contains a test and an implementation of the half adder component. Run the half adder test. Visit the half adder implementation and compare the code to Figure 6.
Solution available: `https://youtu.be/7nM3RS1chFc`

4.3.8

Figure 7 illustrates how to build a full adder using two half adders. Use the CircuitSim simulation framework to create a new class FullAdder. Create a thorough test for the class as well, verifying all possible combinations of input. Use the existing half adder implementation as a template.
Solution available: `https://youtu.be/y6pIo95_Ssw`

4.3.9

Use the CircuitSim framework and your implementation of the FullAdder from above to combine 4 full adders into a 4 bit ALU (see Figure 9). Test all possible combinations of input to the 4 bit ALU.
Solution available: `https://youtu.be/QbapfRwSwMI`

5 Flip/flops

The previous sections introduced the concept of binary numbers and how these can be added using a combination of components like half and full adders. This section demonstrates how memory circuits can be built from simple gates, such that numbers can be stored in that memory and act as input to computational operations such as addition.

5.1 Storing Bits

If we want to store numbers in memory, it must be possible to store individual bits. A full number will then be a sequence of individual bits. So we can reduce the problem of storing full numbers to the more simple problem of just storing a single bit. Using electrical circuits a bit can be stored on a wire by setting the voltage of the wire to 5 V (for storing 1) or 0 V (for storing 0). A first attempt at designing such a circuit is shown in Figure 11.

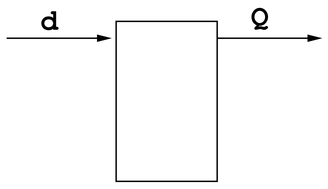


Figure 11: Simple Bit Store

If we put the value 1 (5 V) on the input wire `d`, the purpose of the circuit is to store this value. This can be done by setting the value of `Q` to 1 as well. The wire `d` going into the component will always have a value in between 0 and 5 V. It is not possible to turn off the wire. Thus, if we put a value of 1 on the wire and want it to be stored, later the value may change, and then we don't want the value of `Q` to change as well. Rather we want the value of `Q` to remain the same until we store a new value into the bit store. We see that we need something more in order to tell the circuit when to sample the value we want to store, and then remember that value until we tell it to sample a new value. Figure 12 has extended Figure 11 with an additional input line.

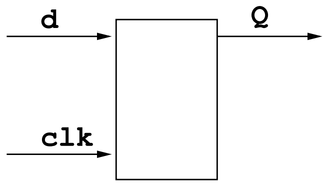


Figure 12: Simple Bit Store With Clock

The purpose of the new `clk` input line is that the value at the input line `d` will only be propagated to `Q` when the input line `clk` goes from 0 to 1. We say that pulling the `clk` high will store the value present at `d` into `Q`. What happens at `d` when `clk` is low (0 V) must not effect the value of `Q`. If we can build such a circuit using simple components (gates) then we will have the ability to store a single bit. Combining e.g. 4 such circuits we will then be able to store a 4 bit number.

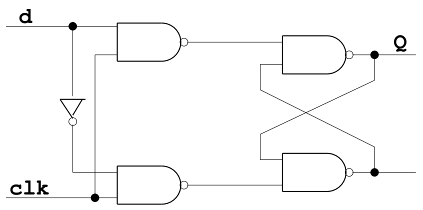


Figure 13: D Flip/Flop Circuit

Consider the circuit in Figure 13. This is made up of 4 *nand* gates and an inverter. A nand gate is a *not and* gate and can be built using transistors and resistors in a similar manner to an and gate. The truth table for the nand gate is listed in Table 11.

X	Y	**X Nand Y**
0	0	1
0	1	1
1	0	1
1	1	0

Table 11: The Nand Gate Truth Table

Let us apply some initial values to this circuit. We assume that the `d`, `clk`, and `Q` lines are all 0 initially. In that case Figure 14 illustrates the values that will have to be on the other lines.

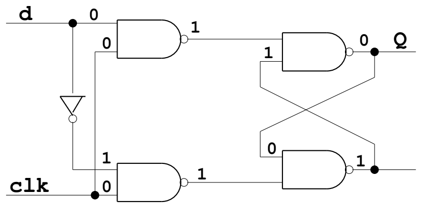


Figure 14: D Flip/Flop Circuit

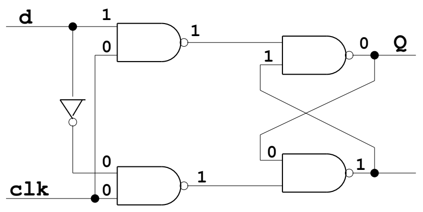


Figure 15: D Flip/Flop Circuit

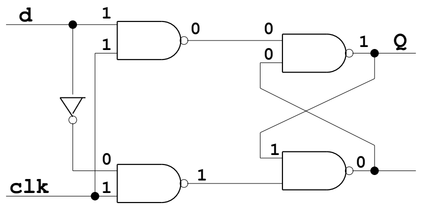


Figure 16: D Flip/Flop Circuit

Let us try from this situation to pull `d` high (to value 1) while keeping `clk` low (value 0). Figure 15 illustrates the values that will then be propagated to the remaining lines. We notice that the value at `Q` remains unchanged. If we pulled `d` low in this situation - while `clk` remains low - we would see that the value at `Q` would remain 0. So, we conclude changing the value at `d` while `clk` is low does not effect the value at `Q`.

Now, from the situation depicted in Figure 15, we will pull `clk` high. The resulting scenario is show in Figure 16. We notice that the value at `Q` changes. It becomes 1 (the same a `d`). If we pulled `clk` low at this point, we would see that the value at `Q` stayed at 1. In effect we have now stored the value previously at `d` in `Q`, and as long as `clk` remains low, the value at `Q` will stay the same.

5.2 Registers

Using Flip Flop circuits we can now store single bits. The minimic operates on 4 bit values, and the ALU designed earlier is a 4 bit ALU. To store 4 bit values in memory we introduce the concept of a *register*. A register is a 4 bit variable that can hold 4 bit values. The minimic will contain 4 such registers. A 4 bit register can be constructed using 4 D Flip Flops, as is illustrated in Figure 17.

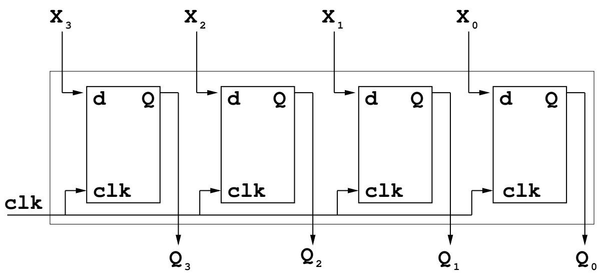


Figure 17: 4 Bit Register Using 4 D Flip/Flops

When seen as a single component the register has 5 input lines: 4 lines of data (X_3, X_2, X_1 and X_0) and 1 clock signal. The same clock signal is propagated into all 4 flip flops. This means that if a 4 bit binary number is put on the input lines and the clock is pulled high, that value will be stored in the register. The value stored in the register can be read from the 4 output lines (Q_3, Q_2, Q_1 and Q_0).

5.3 Exercises

5.3.1

Use the CircuitSim framework to build a d flip flop as illustrated in Figure 13. Test it thoroughly. This includes completing a series of scenarios where the values of the `d` and `clk` input lines are changed in some order to store both 0 and 1 at the flip flop. Verify in the tests that the bit stored at `Q` takes on the expected value.
Solution available: `https://youtu.be/WAYczmRXQsk`
Solution available: `https://youtu.be/qGJ3aPEjnnY`

5.3.2

Use the CircuitSim framework to build the 4 bit register shown in Figure 17. Make a detailed test first that tests that all possible values can be stored and retrieved from the register.
Solution available: `https://youtu.be/riioxKdb0cU`

6 The Register Bank

In the previous sections we have looked at how to construct a 4 bit memory cell (a register) that can hold a 4 bit value. In this section we will describe how to combine 4 registers into a single component called a register bank.

The purpose of the register bank is to hold number values currently being used by the computer to perform its operations. Eventually in the final version of the minimic, values in the register bank can be retrieved from main memory and they can act as input to the ALU when performing arithmetic operations.

Seen as a single component the initial version of the register bank will have the following input lines:

- 1 *rw* bit. This bit determines if we want to store a new value into the register bank, or if we want to read an existing value from the register bank
- 4 value input bits $i_3i_2i_1i_0$. When storing a value into one of the registers in the register bank, the value to store must first be placed in binary format on these input bits
- 2 address bits for storing w_1w_0. Since there are 4 registers in the minimic register bank, there must be some facility for selecting which register to store the value present at $i_3i_2i_1i_0$ into. The value should be stored into that register only, leaving the values in the other registers unchanged. These two address bits identify which register is the target register
- 4 value output bits $o_3o_2o_1o_0$. When reading values from one of the registers, the value read is placed on the output lines $o_3o_2o_1o_0$
- 2 address bits for reading r_1r_0. These bits select which register to read from
- 1 clock signal *clk*. When the value to be stored has been placed at $i_3i_2i_1i_0$ and the address of the target register has been placed at a_1a_0, pulling the *clk* high will perform the actual storing. Nothing will happen to the content of the registers in the register bank until the *clk* input line is pulled high. If reading, the user of the register bank first has to put the address of the register to read from on the r_1r_0 input lines and then pull the *clk* high. After that the value read from the register will be present at the output lines $o_3o_2o_1o_0$

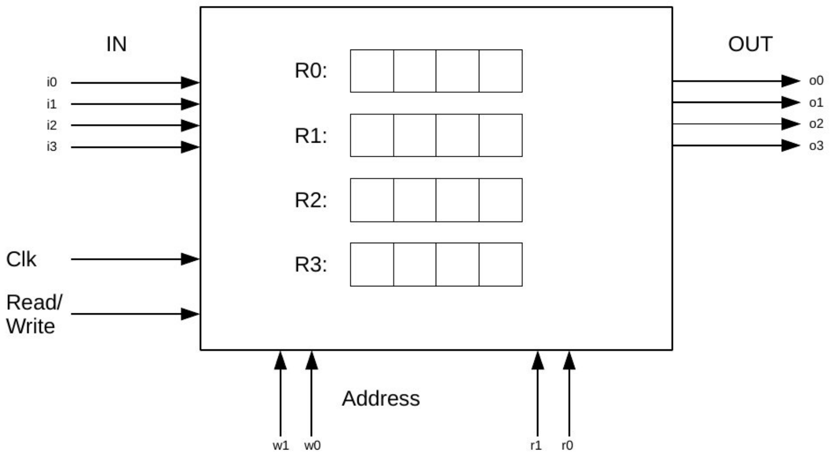


Figure 18: The Minimic Register Bank

The abstract schematic of the register bank is shown in Figure 18. We want to construct this register bank using only simple gates. The registers can be built using flip flops as explained earlier. The following sections will show how to add components to control storing into and reading from the register bank.

6.1 Storing Into the Register Bank

Imagine that we wire up the 4 input lines to all 4 registers, and propagate the same clock signal into all register as well. This is depicted in Figure 19.

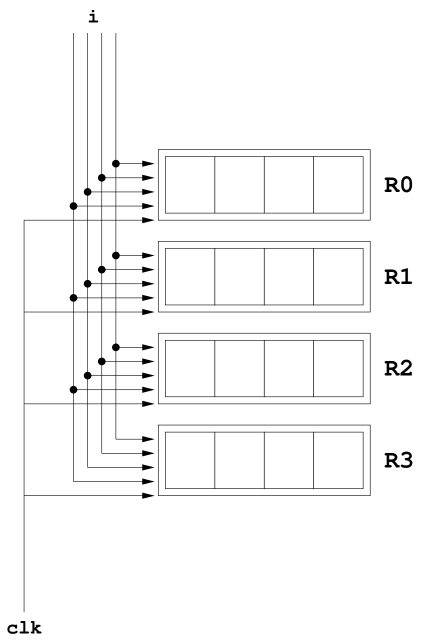


Figure 19: Storing Into the Register Bank

If placing a value at the input lines and pulling the *clk* signal high, that value would be stored at all registers and not just a single one. What is missing is that we must use the w_1w_0 address bits to control which register should get the clock signal. If we can ensure that only one register gets the clock signal while the others do not, then the value on the input line would be stored in the selected register only, leaving the others untouched.

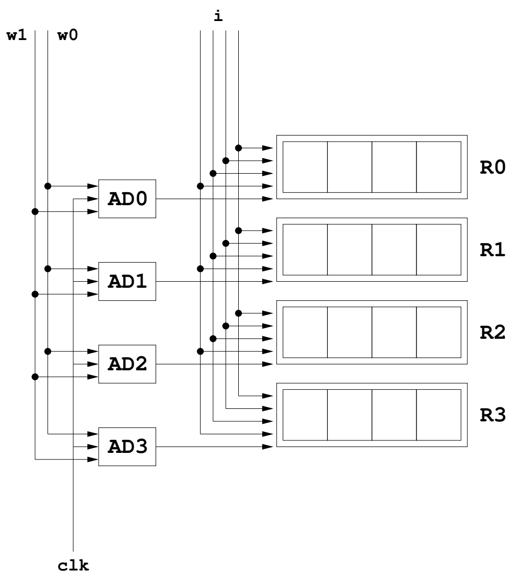


Figure 20: Address Decoders

Figure 20 shows a solution to this problem. The *clk* signal is routed through a new component called an `Address Decoder`. The input lines w_1w_0 are connected to the address decoder. The purpose of each address decode is to only forward the *clk* signal if a particular register has been identified on w_1w_0. The rules are,

1. AD0 should allow the *clk* signal through if and only if $w_1w_0 = 00$
2. AD1 should allow the *clk* signal through if and only if $w_1w_0 = 01$
3. AD2 should allow the *clk* signal through if and only if $w_1w_0 = 10$
4. AD3 should allow the *clk* signal through if and only if $w_1w_0 = 11$

As an example of how to build the individual address decoders we will look at $AD1$. This decoder should allow the *clk* signal through only in the case where $w_1w_0 = 01$, in all other cases its output should be 0. The truth table for this is shown in Table 12.

w_1	w_0	clk_{in}	clk_{out}
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

Table 12: Address Decoder Truth Table

Section 4.1 explained how a truth table like Table 12 can be translated into an equivalent Boolean function, and in this case we get,

$$f(w_1, w_0, clk_{in}) = \bar{w_1} * w_0 * clk_{in} \tag{7}$$

This Boolean function can be realized using the circuit in Figure 21.

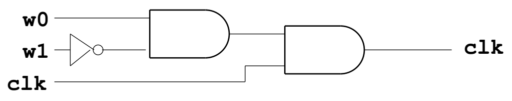


Figure 21: Address Decoder for R1

In a similar way the remaining address decoders (AD0, AD2 and AD3) can be built. The way to store a number into the register bank is now,

1. Place the value to store on the input lines $i_3i_2i_1i_0$
2. Place the address of the target register on the address lines w_1w_0
3. Pull the *clk* signal high

Now, the value on $i_3i_2i_1i_0$ has been stored in the selected register leaving the other registers unchanged.

6.2 Reading From the Register Bank

Apart from storing a value into the register bank it is also possible to read the content of a selected register and have that content appear on the $o_3o_2o_1o_0$ output lines. Which register to read from is selected by the r_1r_0 input lines.

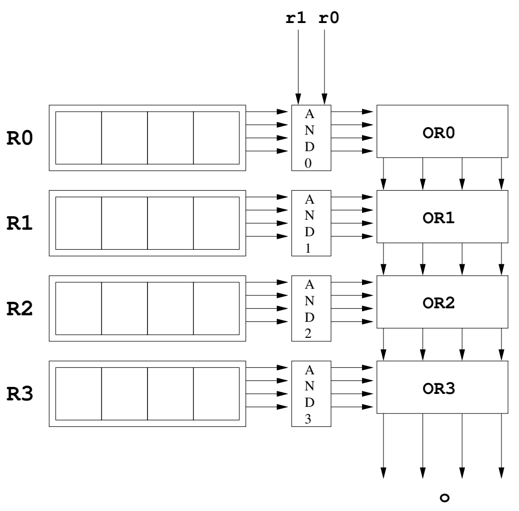


Figure 22: Reading From the Register Bank

Figure 22 shows the abstract schematic of the read portion of the register bank. The challenge is the same here as with the store functionality: It is not possible to wire all outputs of all 4 registers to the same output lines, since we only want to read the selected register and not all of them at the same time.

The figure introduces two new components (1) the AND_x component and (2) the OR_x component. The read lines r_1r_0 are connected to all of the AND_x components (only the first connection shown in the diagram). The purpose of these two new components are:

- AND_x: If the read lines r_1r_0 indicate register x then the output of the AND_x component is the content of register x, else the output is 0.
- OR_x: This component implements a bitwise logical or between each pair of input bits. In terms of the diagram the topmost line coming in from the left is or'ed with the left most line coming in from the top to produce the left most line of output, and so on for the remaining lines. The input coming in from the top OR_0 is 0000 (not shown in the diagram)

The AND_x components can be viewed as *read gates*: they only allow the output from a single selected register to flow out from the register bank. Output from the unselected read gates is 0.

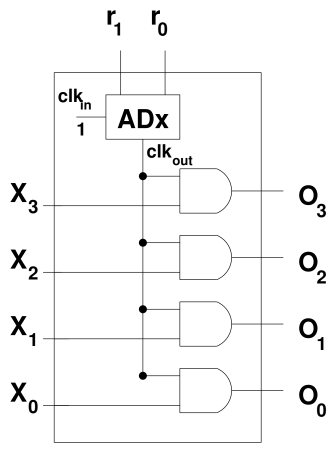


Figure 23: AND_x Using Address Decoders

Figure 23 shows how to build the read gates by reusing the address decoders from Figure 20. The address decoders have 3 input lines: w_1w_0 to control if the decoder is on or off and the clk_{in} line which is the signal that will be allowed through if the decoder is on. The r_1r_0 lines can be routed directly into w_1w_0 and the clk_{in} line can be hardwired to 1 (always on), then clk_{out} of the address decoder can be used to control 4 and gates - each connected to one of the 4 output lines from the attached register.

Realizing the diagram in Figure 22 allows the user to read from the register bank in the following manner:

1. Place the address of the register to read from on the r_1r_0 input lines
2. The content of the selected register is now available at the $o_3o_2o_1o_0$ output lines

The register bank can be seen as a small piece of memory holding 4 memory locations. The input and output lines can be used to control which memory cells to store to or read from. All CPUs contain at their core a similar register bank area. In the following sections we will connect the register bank to the ALU, such that we are able to store values into the register bank, add them to each other and store the result back into the register bank.

6.3 Exercises

6.3.1

Make a drawing of the circuits realizing the address decoders for AD0, AD2 and AD3.

6.3.2

Implement each address decode A0, AD1, AD2 and AD3 as individual components using the CircuitSim framework. Test each address decoder extensively. Solution available: `https://youtu.be/Pb7o2_Zn1F8`

6.3.3

Design and implement the AND_x components from Figure 23. Test the components extensively.

6.3.4

Design and implement the OR_x components from Figure 22. Test the components extensively.

6.3.5

Design and implement the full register bank component including both store and read functionality. Test it extensively. The rw input line (to control if a store or read operation is in effect) has not been introduced earlier. Where would you connect it? Why is it necessary?

6.3.6

Implement a test of the register bank that performs the following operations:

1. Store the value 13 into R0
2. Store the value 15 into R1
3. Store the value 1 into R2
4. Store the value 17 into R3
5. Read the value in R0
6. Read the value in R1
7. Read the value in R2
8. Read the value in R3
9. Store the value 7 into R1
10. Read the value of R0
11. Read the value of R1

7 The CPU Core

The core behavior of a CPU is to fetch values from memory, perform operations on them, possibly store the result of the operation and finally check the outcome of the operation in order to decide what to do next.

As an example consider sorting an array of numbers. To do that the CPU must - amongst other things - be able to compare 2 numbers. This can be done by fetching each number from main memory and place them in two registers. Then the content of the two registers can be subtracted. If the result of the subtraction is 0 the two values are the same. If the result is not zero the sign of the result gives information as to how the numbers relate. When programming such a feature, e.g. sorting, developers use high level programming languages, e.g. Java or C#, but these are translated into a sequence of instructions that can be executed by the CPU core.

This section describes how the minimic CPU core can execute the addition instruction.

7.1 Connecting the Register Bank With the ALU

The ALU designed previously can be used to add two 4 bit binary numbers. But where does the input numbers come from and where should the result be stored? In the minimic we make the restriction that the first input number to the ALU is always the same register, namely R3. From now on we will call this register the *accumulator* or ACC for short. The other input to the addition can be any of the other registers R0, R1 or R2. The result of adding R_x to the ACC register is always stored back into the ACC register. These restrictions are not general for all CPU designs, but they are not unreasonable either: e.g. the Commodore 64 had similar restrictions and also operated with a dedicated accumulator register. The abstract schematic of the minimic CPU core is illustrated in Figure 24.

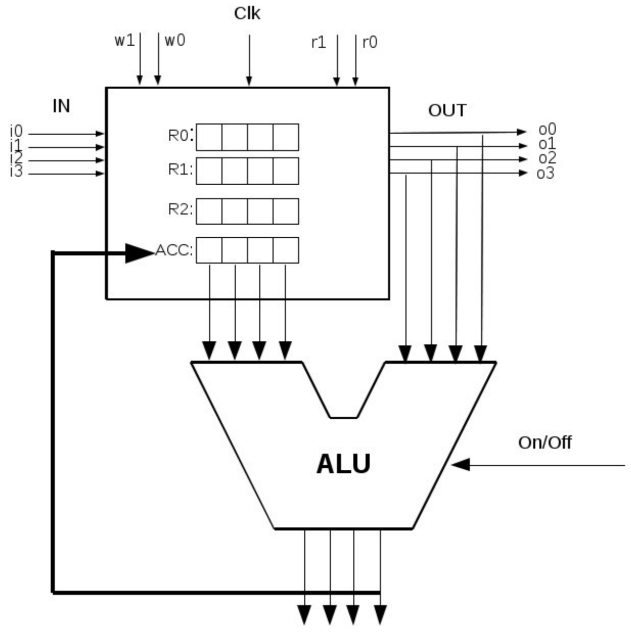


Figure 24: The Minimic CPU Core

An addition can be executed through the following steps,

1. The first value to add is loaded into R0, R1 or R2. This is done by first placing the value on the $i_3i_2i_1i_0$ input lines, then select which register to store into by loading w_1w_0. Finally pulling the *clk* signal high. Now the first operand has been loaded
2. In a similar manner as above the accumulator register (R3) is loaded with the second operand
3. Using the r_1r_0 read lines the register used for the first operand is now selected. At this point the two operands are present at the input of the ALU and after a while the result will propagate to the output of the ALU and be present at the input to the register bank. Finally the w_1w_0 control lines to the register bank must be selected. Then the *clk* line is pulled high and the result of the addition will now be present at the accumulator register

7.1.1 Storing into *ACC* - The Input Selector

The result of the addition is always stored into *ACC*, but according to Section 6 the $i_3i_2i_1i_0$ input lines to the register bank is also connected to R3 (*ACC*). This apparently impossible scenario is depicted in Figure 25.

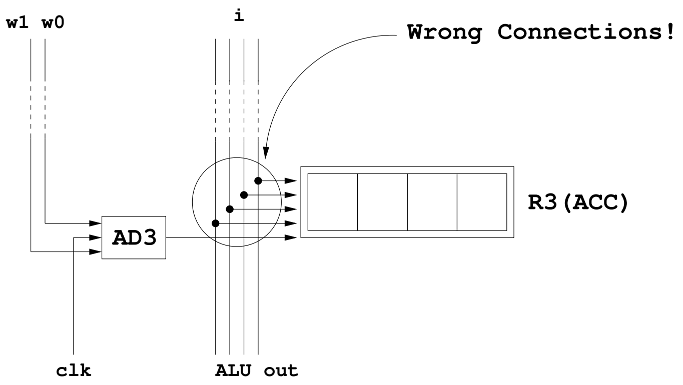


Figure 25: Wrong *ACC* connections

The problem is that if we want to be able to store a value into R3 then the $i_3i_2i_1i_0$ input lines to the register bank must be propagated to R3. On the other hand if we want to store the result of the addition - which will be available on the output lines from the ALU - then the output from the ALU must also be propagated to R3. Connecting the lines directly as in Figure 25 does not make sense. We have to put some value on the input lines $i_3i_2i_1i_0$. If they e.g. get the value 0000 and the output from the ALU is anything different (e.g 1111), then the circuit is attempting to pull the lines up and down at the same time. This is a short circuit and will physically destroy the wires and components. It follows that a facility for selecting where the input to R3 should come from, if it should be from the $i_3i_2i_1i_0$ input lines or from the output of the ALU, must be devised.

Figure 26 shows the schematic for the *inputselector* component. This is a 1 bit input selector. It works in the way that if the s bit is enabled (true) then input bit A is routed to the output, otherwise input bit B is routed to the output.

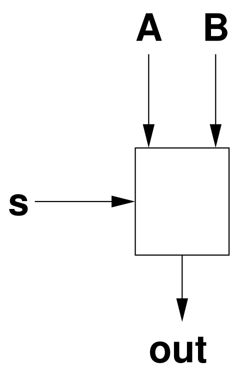


Figure 26: 1 Bit Input Selector

This behavior is described partially in the truth table 13. The table shows the scenarios where the s bit is off. As can be seen the B input is selected and routed to the output. The values in the rest of the table has been left out and filling them in is left as an exercise.

s	A	B	out
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
?	?	?	?
?	?	?	?
?	?	?	?
?	?	?	?

Table 13: Partial Truth Table For the 1 Bit Input Selector

According to the method described in Section 4.1 for translating truth tables into Boolean functions, the equation describing the first part of Table 13 is,

$$f(s, A, B) = \bar{s} * \bar{A} * B + \bar{s} * A * B + \ldots \tag{8}$$

This equation is not complete. Completing it is left as an exercise. Reducing the equation and building a 1 bit input selector now becomes possible. Combining 4 1 bit input selectors into a 4 bit input selector can now be used to solve the issue raised above. Using a 4 bit input selector it is possible to select where input to R3 should come from, whether it should be from input lines $i_3 i_2 i_1 i_0$ or the output from the ALU. This solution has been illustrated in Figure 27.

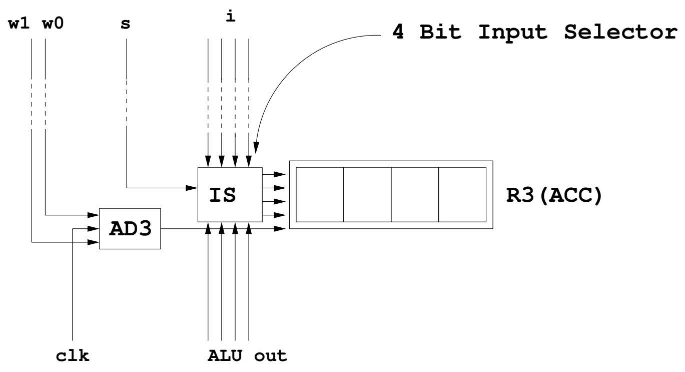


Figure 27: Using the 4 Bit Input Selector

It is now possible to execute the add instruction: (1) first the first operand is placed in a register R_x where x is 0, 1 or 2 (2) then the other operand is placed in ACC (s is set to configure the 4 bit input selector to take input from $i_3i_2i_1i_0$) (3) then the register bank is configured using the input lines r_1r_0 to select the content of R_x to flow to the input lines into the ALU (4) finally the input selector s is set to configure the 4 bit input selector to take input from the ALU and the clk is pulled high to store the result of the addition into ACC. There is still one challenge left that needs to be solved before the ALU is properly connected with the register bank. This is the topic of the next section.

7.1.2 Storing into *ACC* - The ALU Output Buffer

The CPU core design arrived at so far is illustrated in Figure 28. This design uses all of the components introduced so far in order to implement a 4 bit CPU core. There is one very subtle error in this design that reveals itself if trying to simulate or build the design. If an attempt is made to add a number (e.g. R0) to the content of ACC and store the result back into ACC it happens that the simulation never stabilizes. If implemented and run in actual hardware (e.g. using an FPGA) the result appearing in ACC will appear to be a random number, and not the expected result. The reason is that there is a loop in the design involving the Input Selector, the ACC register and the ALU. To explain this problem further lets imagine what will happen when an addition is performed:

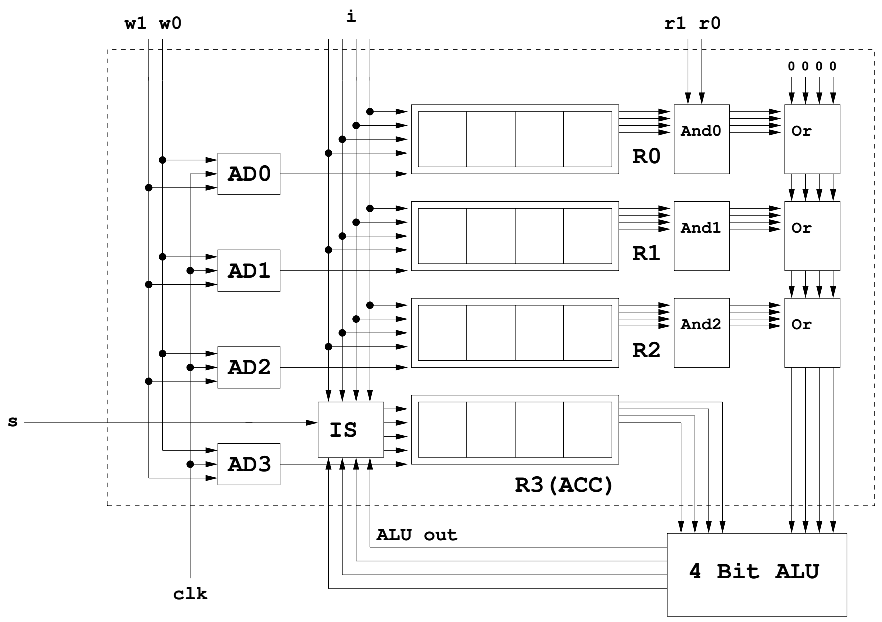


Figure 28: 4 Bit CPU Core (Version 1)

1. The r_1r_0 input lines are set to choose e.g. $R0$ as the input operand for the addition
2. The current content of the ACC register is the other operand. Both these values will now be present at the input of the ALU
3. The output from the ALU is not driven by a clock signal. Rather it flows out of the ALU as soon as the circuit settles. This result is now present at the input of the input selector attached to ACC
4. The input selector is configured using the s input line to let the output from the ALU flow into ACC. This output is now present at the flip/flop input lines in ACC
5. So far nothing has been stored in ACC. This will only happen when the clk input line is pulled high. Lets assume this is done now. Now the result of the addition is stored into ACC, as intended. But the output from ACC will now change, and immediately flow into the ALU producing a new result that passes back up to the input selector and into the flip/flops at ACC

In an actual implementation the loop described in step 5) above will result in undefined behavior. Expressed in prose we can say that the ALU is producing a new result that immediately becomes input for the next addition that proceeds in an uncontrolled fashion. The way we will solve this is by introducing an extra internal register that acts as a buffer for the ALU output, and only forwards this output when it gets a clock signal. This solution is shown in the final minimic CPU core design illustrated in Figure 29.

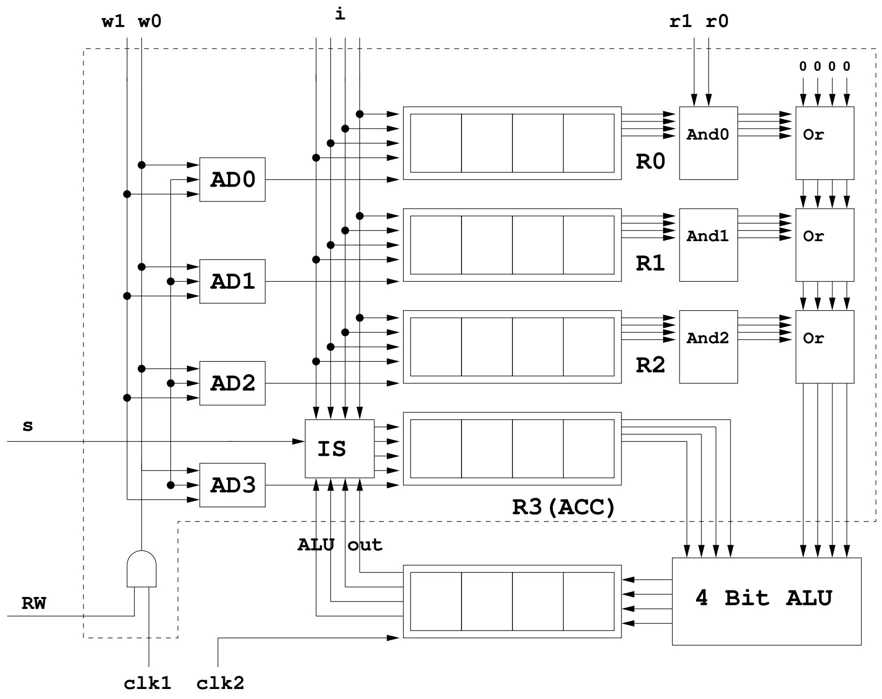


Figure 29: 4 Bit CPU Core (Final Version)

Now the problem with the ALU loop can solved in the following manner: before pulling *clk*1 high, the new clock signal *clk*2 is pulled high and then low again. This will store the output of the ALU into the new internal register. Then *clk*1 can be pulled high. Even though the ALU will continue to operate and produce a new result, the loop is broken since *clk*2 remains low, preventing the new output from the ALU to appear at the input of *ACC*.

Now, the core design of the minimic micro controller has been introduced. What remains is just additional sugar: we need an instruction register, a program memory area, a program counter and a status register. These concepts are the subject of the ensuing sections.

7.2 Exercises

7.2.1

Fill out the remaining parts of the truth table in Table 13.

7.2.2

Complete equation 8 in Section 7.1.1. Reduce the resulting equation as much as possible using the rules of Boolean algebra from Figure 10 in Section 4.1.

7.2.3

Draw a diagram consisting of gates and wires implementing the reduced Boolean expression from Exercise 7.2.2.

7.2.4

Using the CircuitSim frame work implement the 1 bit input selector as a new component. This is about implementing the design arrived at in Exercise 7.2.3.

7.2.5

Using the CircuitSim framework implement a 4 bit input selector by combining 4 1 bit input selectors (see Exercise 7.2.4).

7.2.6

Add the 4 bit input selector from Exercise 7.2.5 to the internals of the Register-Bank component. So far let the default (s == false) enable the $i_3 i_2 i_1 i_0$ input lines to be propagated to the ACC register.

7.2.7

Describe how you would store two numbers into R0 and ACC and perform an addition using the 4 bit minimic CPU core from Figure 29. This involves describing in which order to set the input lines and which values to set.

7.2.8

Using the CircuitSim framework, implement the 4 bit minimic CPU core from Figure 29. This involves writing the test first, which can be done based on the solution to Exercise 7.2.7.

8 The Instruction Register

The minimic CPU core (see Figure 29) has a number of input lines to control its behavior:

- $i_3 i_2 i_1 i_0$ - Prior to executing a store operation these 4 input lines must be loaded with the values to store
- $w_1 w_0$ - Prior to executing a store operation these 2 input lines must be loaded with the address of the register to store
- $r_1 r_0$ - Prior to executing a read operation these 2 input lines must be loaded with the address of the register to read. After the read operation is performed the content of the selected register will be available at the $o_3 o_2 o_1 o_o$ output lines
- RW - Selects if a write or read operation is in effect. Set to true if writing into the register bank
- s - Selects if an add operation is in effect. Set to true if the output of the ALU should be stored into ACC
- $clk_{1/2}$ - These clock signals must be clocked in the correct order to perform a addition operation
- $o_3 o_2 o_1 o_o$ - These 4 output lines are only used for debugging. The output from the register bank is always directed into the ALU, no matter what kind of operation is being performed

With the current design of the CPU core it is possible to execute 3 types of operations (1) store operations, (2) read operations and (3) add operations. We will call such operations *instructions*. Each instruction is performed by first setting the proper input lines and then clocking the CPU core. We will name these three instructions accordingly:

- $OUT\ R_x$ - Loads the content of the selected register $x \in 0, 1, 2, 3$ onto the output lines $o_3 o_2 o_1 o_o$
- $LDI\ R_x, imm$ - Stores the 4 bit value $imm \in 0, .., 15$ into the selected register $x \in 0, 1, 2, 3$
- $ADD\ R_x$ - Adds the content of the selected register $x \in 0, 1, 2$ to the content of R_3 (aka ACC)

The minimic CPU core could be implemented in actual hardware, and the input lines could be wired to some switches and the output lines to some LEDs. Now an operator would be able to manually set switches and operate the clock to perform a sequence of instructions. To help the operator we will introduce the concept of the *instruction register*. The instruction register is a sequence of bits that is connected to the input lines of the CPU core. How the bits are connected can be chosen in many equally good ways. Figure 30 shows how the minimic instruction register is defined.

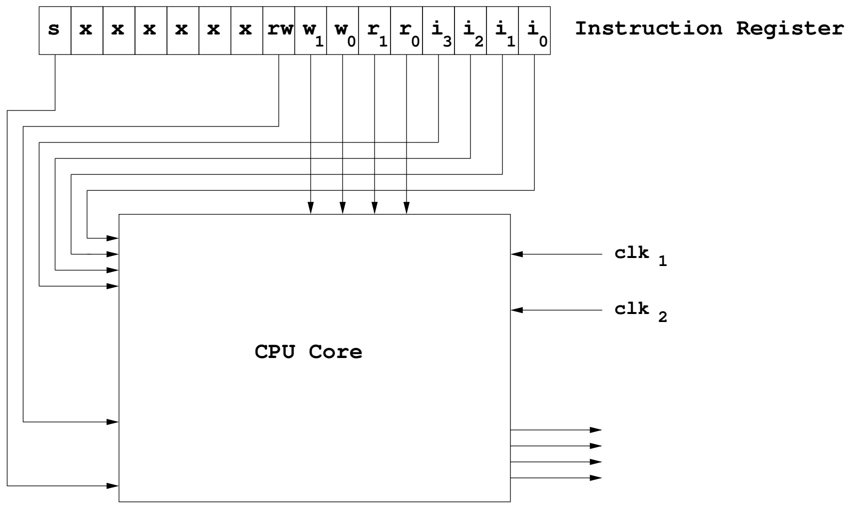


Figure 30: The Minimic Instruction Register

As illustrated all control lines into the CPU core has been collected into a single 16 bit instruction register. Currently not all bits are used. The unused bits are marked with an x. To prepare for instruction execution the correct binary value is loaded into the instruction register. This value should be constructed in such a way that the desired bits are turned on/off to select the correct CPU core behavior. Then the clock signals are pulled high and low in the correct order. This constitutes the execution of an instruction.

8.1 Exercises

8.1.1

What binary values should be placed in the instruction register to execute the following instructions:

1. **LDI** $R_0, 7$
2. **LDI** $R_2, 10$
3. **ADD** R_1
4. **ADD** R_2
5. **OUT** $R_0 a$
6. **OUT** $R_3 a$

9 The Program Memory

Manually pulling switches high or low to set the content of the instruction register (see Section 8) quickly becomes tedious. Instead all instructions are kept in the program memory. It follows that the types of values that can be stored in the program memory are of such a magnitude that they fill out all bits in the instruction register (and not any more). It follows that each memory cell in the minimic program memory will have the same width as the minimic instruction register, which is 16 bits.

The program memory can be viewed as a store of instructions, each of which can be transferred to the instruction register in an order selected by the operator. The abstract schematic of the program memory has been illustrated in Figure 31.

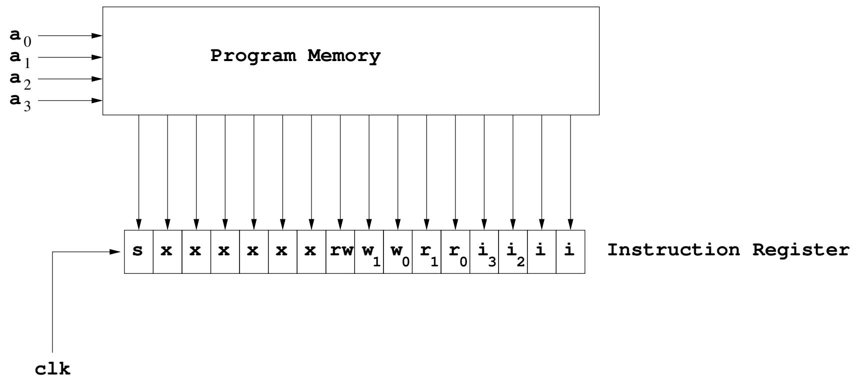


Figure 31: The Minimic Program Memory

The program memory has 4 input lines $a_3a_2a_1a_0$, and 16 output lines wired directly to the instruction register. To select the next instruction for execution the operator places the address of the instruction on the address input lines $a_3a_2a_1a_0$. After the circuit settles the instruction bits are now available on the output lines and are propagated to input lines of the instruction register. Pulling the clock high will clock the flip/flops of the instruction register effectively storing the next instruction to execute into the instruction register.

Since there are only 4 address lines into the minimic program memory it is only possible to select 1 of 16 addresses. Since each of the memory cells in the program memory contains 16 bits - the same as 1 word, or two bytes - it follows that the minimic program memory contains at most 32 bytes of memory. While this is not much it is enough for small simple programs.

The program memory is actually a special version of the register bank. It is the same as a register bank with 16 16 bit registers. It follows that the program memory can be built using the same components as the register bank, namely a sequence of address decoders and 16 bit wide memory cells built using flip/flops.

9.1 Exercises

9.1.1

Show the truth table for a circuit constituting a 4 bit address decoder. The output of the decoder should be 1 only in the case that the address 7 is selected.

9.1.2

Construct the Boolean formula for the truth table defined in exercise 9.1.1. The form of the equation will be,

$$f(a_3, a_2, a_1, a_0) = \ldots \tag{9}$$

9.1.3

Construct the circuit diagram showing how to build the 4 bit address decoder from Exercise 9.1.2.

9.1.4

Use the CircuitSim framework to test and simulate the 4 bit address decoder from Exercise 9.1.3.

9.1.5

1. How many flip/flops will be required to build the minimic program memory?
2. How many address decoders will be required to build the minimic program memory?

9.1.6

Building the minimic program memory using actual components is possible but tedious. Instead construct a Java class simulating the minimic program memory behavior. The class must have the proper interface in terms of input and output, but is allowed to cheat in its internal implementation. The natural data type to use for program memory content is an array of Java *short* values.

10 The Program Counter

Section 9 explained how instructions to execute can be stored in the minimic program memory, and how the program memory can by built using basic components such as gates. The content of the program memory, which is a sequence of instructions, we will call *the program.* The minimic always start with executing the instruction at address zero. So, initially, the operator must make sure that the address 0 has been placed on the input lines $a_3a_2a_1a_0$ to the program memory before the clock starts ticking. After executing the instruction at address 0 the next instruction to execute will be at address 1, and so on. This first version of the minimic instruction execution method is restricted and does not allow for jumps in the code. This will be fixed in an ensuing section.

It is apparent that for the operator to manually select the next instruction to execute is rather tedious. To help the operator we will introduce the concept of a *program counter* (in the following we will call the program counter the PC). The PC is a circuit that contains a single register called the *PC register*, and increases the content of this register by 1 every time the clock signal is pulled high. The input and output of the PC circuit is as follows:

- $o_3o_2o_1o_0$ - These four output lines contain the current value of the PC as a 4 bit binary number
- *clk* - This 1 bit input line constitutes the clock. When it is pulled high, the output in the $o_3o_2o_1o_0$ output lines increase by one.

This circuit can be built using components already introduced:

- The PC register is built like the registers in the register bank, simply by combining 4 1 bit flip/flops
- The facility for adding 1 to the current value of the PC register can be implemented by embedding a 4 bit ALU into the circuit. The first input operand to the ALU is the current content of the PC and the second is hardwired to the constant 1
- The output of the ALU is propagated to an intermediate buffer register that in turn is fed back into the PC register. This is needed for the same reasons as discussed in Section 7.1.2 regarding the ALU output in the register bank

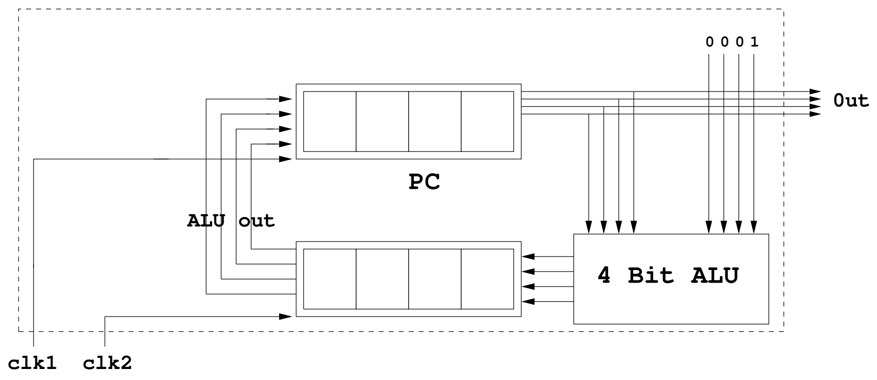


Figure 32: The Minimic Program Counter

The PC circuit internals are illustrated in Figure 32. The output lines of the program counter can be connected to the address input lines of the program memory (see Section 9). If the program memory is combined with the instruction register and CPU core, we now have a complete circuit that can execute a sequence of instructions, stored in the program memory, one by one, by only using two clocks.

The version of the program counter depicted in Figure 32 only allows for the PC to increment by one. In order to execute non trivial programs two types of jumps - where the next instruction to execute is not the one directly following - are required. Without the ability to perform jumps it is not possible to execute if-statements or loops of any kind. Removing if-statements and loops from a program disallows it from being Turing complete. In other words, without if-statements and loops, there is a lot that computers can normally do, that would not be possible. Implementing any kind of programming language as we know it would not be possible. The two types of jumps that are still missing are the *unconditional* (or *absolute*) jump and the *conditional* jump. The following sections will extend the design of the minimic PC module and other components to allow for both these types of jumps.

10.1 Exercises

10.1.1

Using the CircuitSim framework build a test and implementation of the minimic program counter circuit from Figure 32.

11 Unconditional Jumps

All programming languages have loop constructs or at least a programming construct allowing for branches in program execution. If they had not a program would only be able to start at some point and then execute a predetermined sequence of instructions and then end when the last instruction was executed. In such a scenario the program would always execute in the exact same manner every time it was started. Such a program would be deterministic. Deterministic programs are not very interesting since they will always do the same and thus it is only needed to run them once in order to know what will happen everytime the program is executed.

The most simple branch instruction is the goto instruction or unconditional branch. This instruction is - like any other instruction - located at some address in program memory, and when executed it will make the program counter become another address that may not (most likely will not) be the next instruction. E.g. if the goto instruction is at address 7 it may tell the program counter to move to 10 and not 8 as would otherwise be the next instruction to execute.

Figure 33 shows the abstract overview of the minimic containing the components so far introduced.

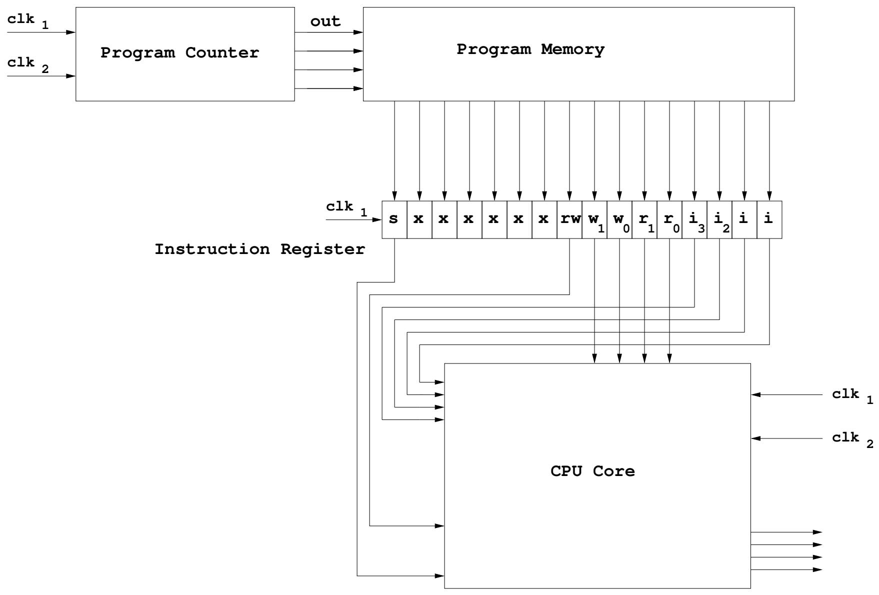


Figure 33: Overview of the Minimic CPU Core

To support the goto instruction it must somehow be possible to load the PC with an address to use that is not the next address in sequence. This can be achieved by adding some input lines to the program counter:

- $i_3i_2i_1i_0$ - Input lines that contain an address

- s - Input line that selects if the next address to output on the $a_3a_2a_1a_0$ output lines will be the output of the internal ALU (PC + 1) or the address loaded onto $i_3i_2i_1i_0$

Some changes are of course required in the internals of the program counter to select if the next value to load into the PC will be the output of the internal ALU or what has been put on the $i_3i_2i_1i_0$ input lines. If these changes have been implemented the instruction register can be connected with the program counter in a manner depicted in Figure 34.

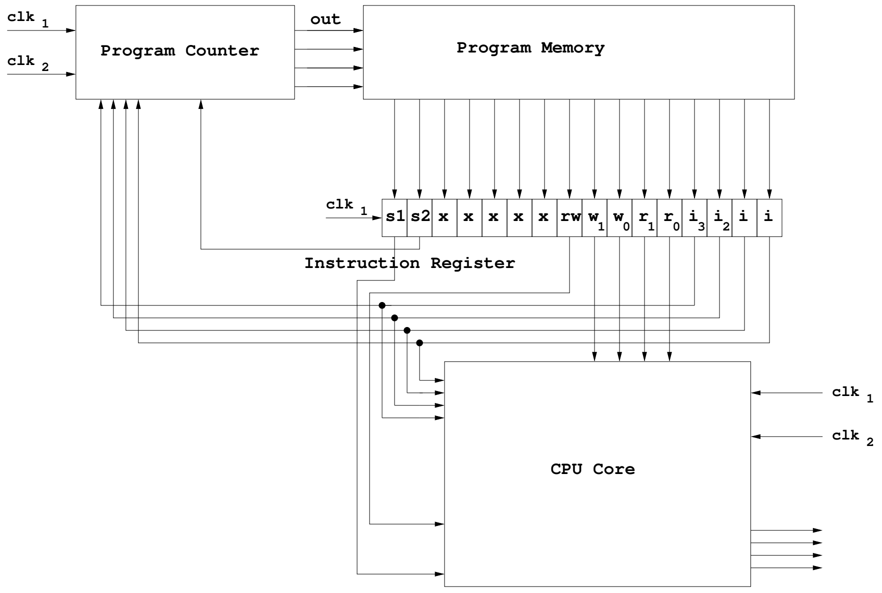


Figure 34: Overview of the Minimic CPU Core (with goto)

This figure has been changed by connecting the first (LSB) 4 bits of the instruction register to the $i_3i_2i_1i_0$ input lines of the PC. Bit 14 (previously unused) is connected directly to the new selector bit going into the PC. The previous design of the program counter (see Figure 32) can be updated using a 4 bit input selector. The details of this is left as an exercise (see Exercise 11.1.1).

With the update of the minimic CPU core design depicted in Figure 34, a goto instruction can now be constructed in this manner:

- The address to goto is stored in $i_3i_2i_1i_0$
- The selector bit in bit 14 is set to 1
- To avoid side effects the RW bit is set to 0 to signal a read operation
- Even though we don't want to read, some value has to be put on the r_1 and r_0 bits. These can be left to 0. A side effect is that the content of R_0 will go onto the output bits of the register bank

- The s_1 bit, to select what to load the *ACC* with, can be either 0 or 1, since the *RW* bit is off, nothing will be loaded anyway

The addition of the new goto instruction and the previous instructions already defined now gives the minimic the following instruction set:

- *OUT* R_x - Loads the content of the selected register $x \in 0, 1, 2, 3$ onto the output lines $o_3o_2o_1o_o$
- *LDI* R_x, imm - Stores the 4 bit value $imm \in 0, .., 15$ into the selected register $x \in 0, 1, 2, 3$
- *ADD* R_x - Adds the content of the selected register $x \in 0, 1, 2$ to the content of R_3 (aka *ACC*)
- *GOTO imm* - Jumps to the address *imm* before executing the next instruction

Since the goto instruction is an unconditional jump - which will always jump to the mentioned address - all minimic programs are still deterministic. The next section will introduce conditional jumps and finally the minimic will become Turing complete.

11.1 Exercises

11.1.1

Update the implementation in CircuitSim of the program counter to contain 4 additional input lines and a selector to chose if the value on the additional 4 input lines are loaded into the PC or the output of the internal ALU is used as previously.

Construct a thorough test first before starting to update the PC internals. A 4 bit input selector (designed previously) can be used in the solution to this exercise.

11.1.2

Follow the description of the goto instruction at the end of Section 11 and construct a binary pattern that can be loaded into the instruction register to implement a goto instruction.

11.1.3

Using the new goto instructions create a small program that continously adds the value 2 to the content of *ACC*. You can use the 4 instructions listed at the end of Section 11.

12 Conditional Jumps

The unconditional jump instruction introduced in Section 11 always jumps to a specific address. The conditional jump is different, it only jumps if some condition is met, otherwise execution continues with the next instruction in sequence. In terms of the program counter component (see Figure 34) a conditional jump will sometimes load as the next address the value on the $i_3i_2i_1i_0$ input lines, and sometimes load the output of the internal ALU (PC + 1). The question remains what to use as the condition? What should determine if the instruction will perform a jump or if it will just continue with the next instruction in sequence?

To answer this question we must look at what we will use the conditional jump instruction for. This instruction can be used for implementing if statements. If some condition is met a block of code will be executed, otherwise the block will be skipped. In the case of the minimic we choose as the condition if the previous result propagated out of the ALU was zero or not. If it was zero the jump will be performed, if it was not zero the jump is not performed and the program continues with the next instruction in sequence.

With this behavior if-statements that check if the result of an addition is 0, can be constructed. This means that we can make loops that will run as long as a variable does not become 0.

How can it be determined if the output from the ALU is 0 or not? Figure 29 shows the internals of the CPU core, including the ALU. The output from the ALU is placed into a buffer register for reasons explained in Section 7.1.2. The content of this buffer register will become 0 if the output of the ALU is zero. Whether the output of the last ALU operation became 0 or not can be stored using a single bit. This bit we will call the z bit (or zero bit). Let's call the content of the buffer register $a_3a_2a_1a_0$. The value of the z-bit should be 1 only in the case that $a_3a_2a_1a_0$ are all zero. Following the method introduced in Section 4.1 we will write the truth table for the z-bit as a function of $a_3a_2a_1a_0$. A portion of this truth table is listed in Table 14.

a_3	a_2	a_1	a_0	**Z**
0	0	0	0	1
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
..	..	..	..	0
1	1	1	1	0

Table 14: Z bit

It is apparent that only in the case where where all bits in $a_3a_2a_1a_0$ are zero should the z-bit become 1, since only in that case will the value of the outcome of the last ALU operation be 0. Written as a Boolean function this can be expressed as:

$$f(a_3, a_2, a_1, a_0) = \bar{a_3} * \bar{a_2} * \bar{a_1} * \bar{a_0} \quad (10)$$

Equation 10 can be realized using some and gates and 4 inverters. Assume now, that we have constructed a component that takes as input the 4 input bits $a_3a_2a_1a_0$ from the ALU buffer register and produces as output a single bit, the z-bit. Then the conditional jump can be performed if the z-bit is propagated into the selector input bit of the program counter. If a conditional jump instruction is executed, then the selector input bit of the program counter should be the z-bit, for all other instructions, the selector input bit of the program counter should be as it was previously. The overview of the minimic CPU core has been updated to reflect this (see Figure 35).

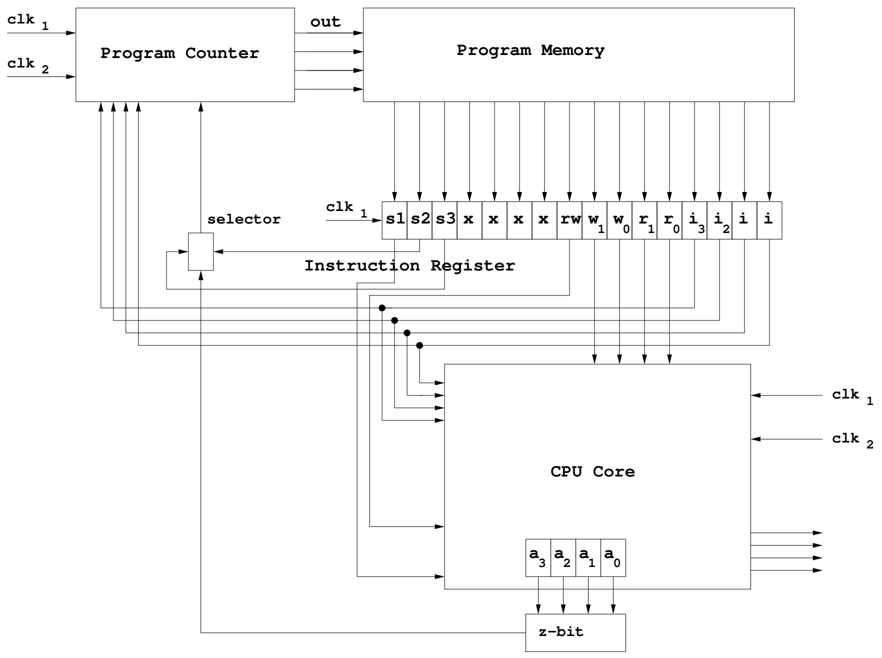


Figure 35: Overview of the Minimic CPU Core (with if)

The ALU buffer register has been made visible and the output of the register flip/flops have been routed into the component calculating the value of the z-bit according to equation 10. The z-bit coming out of that component is routed into a new 1-bit input selector that has been added to the design. The other input to that selector is the previous value from bit 14 in the instruction register, since we still want to support the unconditional goto instruction. Bit 13 is no longer unused but has been introduced to determine if the selector input bit to the program counter should be the z-bit of the goto-bit from the instruction register.

This new diagram gives the ability to execute a new type of instruction: the conditional branch instruction. In this case we can perform a conditional branch if and only if the z-bit is set. We will call this instruction the *branch equal* instruction, or *BEQ imm* for short.

It is now possible to create programs that run in a loop counting down a specific variable and terminates when that variable becomes 0.

12.1 Exercises

12.1.1

Show the diagram implementing the Boolean function in Equation 10 using a proper amount of and gates and inverters. Implement the diagram using the CircuitSim simulation framework. Remember to make the test first, then build the simulation class.

12.1.2

Construct a bit pattern that implements a *BEQ imm* instruction that jumps to address 10 if and only the z-bit are set.

12.1.3

Using all the instructions introduced so far - including the *BEQ imm* instruction - write a small program that counts the content of the ACC register starting from 0. The program should add 1 to ACC in each loop iteration. Since ACC is only a 4 bit register, the content of ACC will wrap around to become 0 again eventually. Make the program break out of the loop when that happens (when the content of ACC becomes 0).