Part 2 – Additional Detail

1 Introduction....................2
2 Logic and Arithmetic....................3
2.1 Logic....................3
2.2 Simple Arithmetic....................6
2.3 Multiple-Byte Addition and Subtraction....................14
2.4 Multiplication....................17
2.5 Division....................20
3 The Control System....................22
3.1 Fetch State....................22
3.2 Execute State....................23
4 More Advanced Processor Features....................31
4.1 16-bit Address Space....................31
4.2 Data Address Indexing....................36
4.3 Subroutines....................44
4.4 Interrupts....................46
4.5 Input/Output Devices....................50
5 Concluding Remarks....................52
Appendix: Boolean Algebra....................54

1 Introduction

Part 1 introduces digital processing from first principles. The descriptions give an insight into the fundamentals of a computer processor. This part provides a more detailed description of some of the key features of the processor and includes some additional capability typical of commercial devices. The result is a more practical version of the simple processor.

This part consists of three sections:

- Logical and arithmetic processing
- The simple processor control system
- Additional processor features

The Logic and Arithmetic section introduces electronic logic and provides more detail on the functionality of the ALU. Also described are examples of multiple-byte arithmetic, multiplication and division.

The Control section describes the implementation of the logic driving the simple processor processing control in its simulated electronic form (Book 3). The detail of the circuit is provided in Book 3.

The final section introduces some additional features that have typically been available on a real processor. These are described here by adding them to the simple processor. The features introduced are in relation to the specific implementation in Part 1 and there are many other ways they can be implemented. There are better ways almost certainly, but the intention here is to keep the overview consistent and accessible to a general readership.

This part concludes with an overview of the constraints on processor performance and real-estate leading to the two major types of processor architectures available today.

2 Logic and Arithmetic

The ALU performs arithmetic and logical operations. However, the ALU does not directly perform multiplication and division. This section describes in more detail operations which deliver the logical and arithmetic functional features including multiplication and division.

At the heart of digital processing devices such as the ALU are electronic logic circuits and the section begins with a summary of these.

2.1 Logic

Part 1 describes in general terms the AND, OR and Exclusive OR functions of the ALU. Table 1 shows the output for each variant containing two inputs (A and B), the notation used and the electronic symbols for circuits which deliver the logic. The circuits shown form the basis of digital electronics.

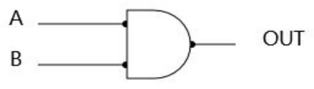
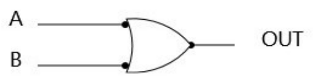
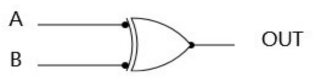
Type	**Inputs** A	B	**Output** OUT	**Notation**	**Symbol**
AND	0 0 1 1	0 1 0 1	0 0 0 1	$A . B$	
OR	0 0 1 1	0 1 0 1	0 1 1 1	$A + B$	
Exclusive OR	0 0 1 1	0 1 0 1	0 1 1 0	$A \oplus B$	

Table 1 Logic AND, OR, Exclusive OR Circuits

Another important electronic logic circuit is the inverter or “NOT”. The circuit consists of one input and one output where the output is the inverted form of the input and is shown in Table 2.

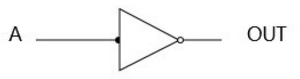
Type	**Input** IN	**Output** OUT	**Notation**	**Symbol**
NOT	0 1	1 0	$\overline{A}$	

Table 2 Not Circuit

In addition the logic circuits “NAND” (NOT AND) and “NOR” (NOT OR) are shown in Table 3. These circuits are logically similar to their counterparts (AND and OR respectively) but with inverted outputs i.e. deliver “negative logic”. The inverted sense of the output in Table 2 and Table 3 is indicated by placing a “bar” over the notation.

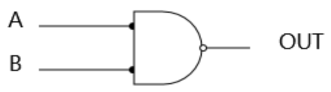
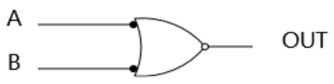
Type	Inputs		Output	Notation	Symbol
	A	B	OUT		
NAND	0 0 1 1	0 1 0 1	1 1 1 0	$\overline{A . B}$	
NOR	0 0 1 1	0 1 0 1	1 0 0 0	$\overline{A + B}$	

Table 3 Logic NAND, NOR Circuits

The circuits in Table 3 are important in the design of digital electronics because combinations of the circuits can deliver both positive and negative logical outcomes. The AND and OR circuits in Table 1 only deliver positive logic. Furthermore, fewer components are required to implement negative logic circuits. A practical AND or OR circuit is a NAND or NOR followed by a NOT. All the circuits described appear in the electronic design described in Book 3.

Arrangements of NAND, NOR and NOT gates create devices referred to as “flip-flops”. These devices can hold bits of data indefinitely and are commonly used as data latches, registers and counters. They are examples of the electronic devices used to hold the byte information referred to in Part 1.

An example (known as a “D flip-flop”) is shown in Figure 1. The bit on “D” (i.e. zero or one) appears at Q when “T” one. When T goes to zero, Q remains unchanged regardless of D. That is, it “remembers” the value. Qbar is the opposite of Q (i.e. NOT Q). Book 3 Part 1 refers further to flip-flops in its Appendix.

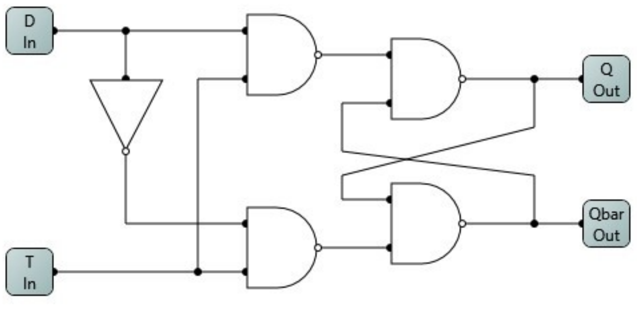


Figure 1 NAND gates arranged as a D flip-flop

The notation used to specify logic forms the basis of Boolean algebra, which is a technique used in the design of electronic logic. Some fundamental concepts of Boolean algebra are included in the Appendix to this part.

The ALU Logic Unit can perform the AND, OR and Exclusive OR logical operations described for two input bytes (accumulator and memory). Each corresponding bit within the input bytes is processed according to the logical function instigated by the instruction and the output placed back into the Accumulator. Figure 2 illustrates the configuration of the logic within the ALU.

The Logic Control function switches the eight bit logic gates between AND, OR and Exclusive OR functions. The inputs are shown as "A" and "B" and the actual data originates from the Accumulator and Memory. Logic Control is decoded from the Instruction Register bits b0 to b2 (described in section 3).

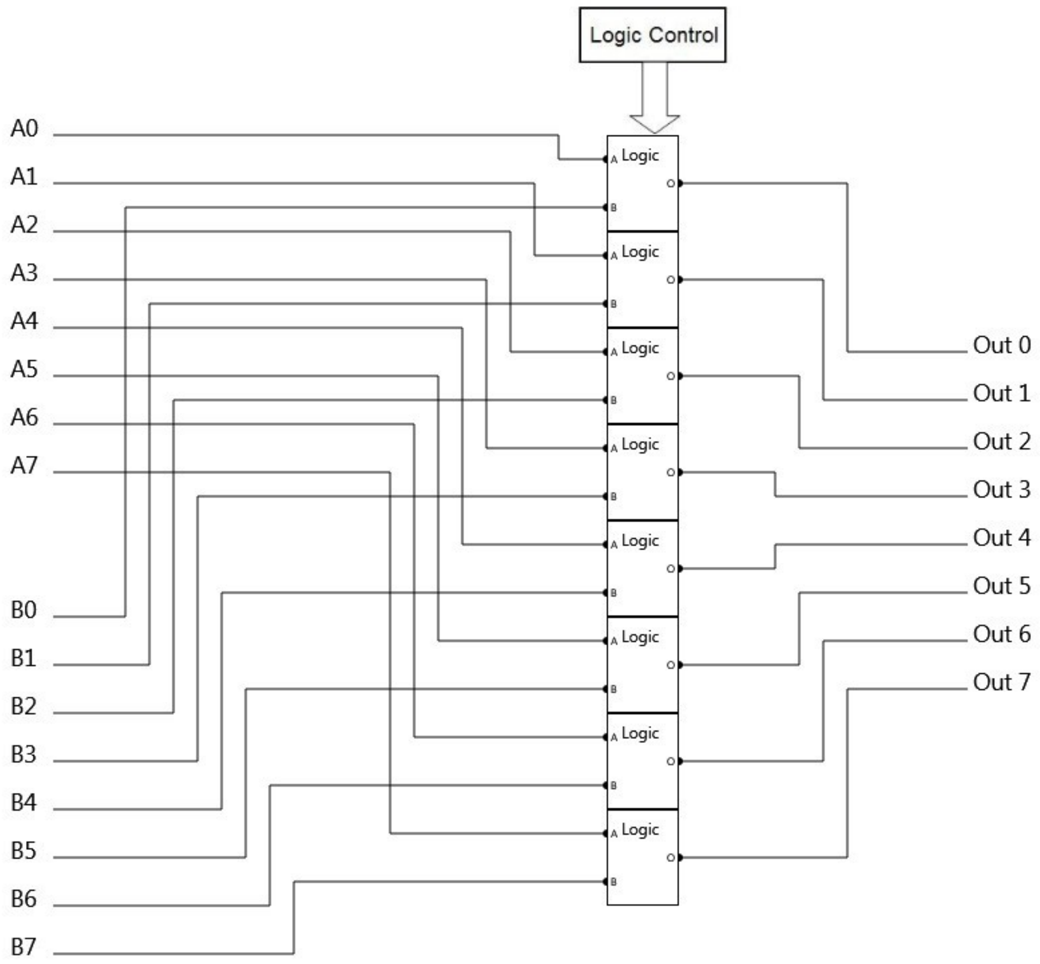


Figure 2 ALU Logic processing arrangement

2.2 *Simple Arithmetic*

2.2.1 Addition

This section describes how the ALU performs the addition of bytes. This is best illustrated through the example additions shown in Table 4.

Decimal	Binary (in bytes)
3	0 0 0 0 0 0 1 1
+ 5	+ 0 0 0 0 0 1 0 1
8	0 0 0 0 1 0 0 0
21	0 0 0 1 0 1 0 1
+ 23	+ 0 0 0 1 0 1 1 1
44	0 0 1 0 1 1 0 0

Table 4 Examples of binary addition

Consideration of how the binary digits add together leads to the following observation

0 + 0 = 0
0 + 1 = 1
1 + 0 = 1
1 + 1 = 0 and carry 1 over to the next higher binary power (next significant bit).

Therefore, for each bit within a byte, the output of the addition consists of the "Sum" and a "Carry" which can be represented in Table 5. The Carry is added to the next higher significant bit in the addition.

A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Table 5 Addition of A and B

This type of table (showing outputs for given inputs) is referred to as a "truth table".

To a computer this is a logical operation and no real concept of addition in terms of quantities is actually performed here. The Sum is the Exclusive OR of A and B, whilst Carry is A AND B. It follows that the logic which delivers this output can be drawn as Figure 3.

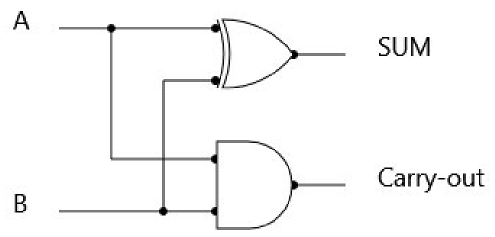


Figure 3 Logic diagram for Half Adder

This circuit is known as a "Half Adder". A "Full Adder" must also add-in any Carry from the next lower significant bit-addition. The truth table for this is shown in Table 6.

Inputs			Outputs	
A	B	Carry-in	Sum	Carry-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

Table 6 Truth Table for Full Adder

The full adder is delivered using the logic gates shown in Figure 4. This circuit consists of two half-adders connected one after another (A, B added first followed by Carry-in) and the Carry-out is an OR function of the Carry from the two half adders.

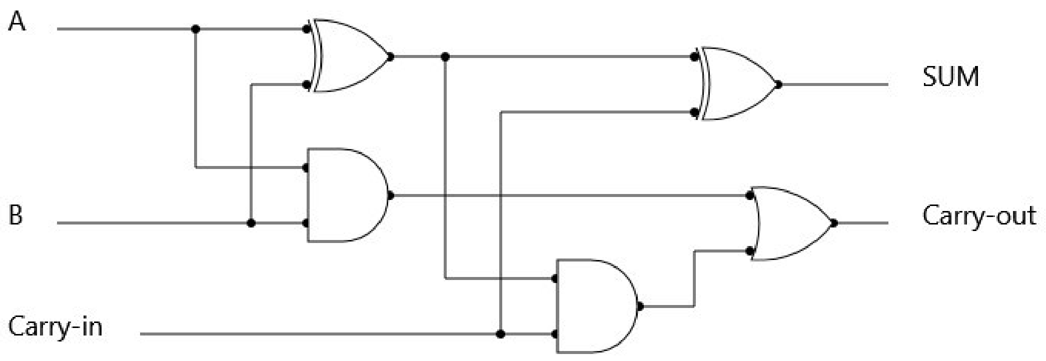


Figure 4 Logic diagram for Full Adder

Finally, the byte-adder consists of eight full-adders wired with successive Carry-out connected to the next higher bit Carry-in. A byte Carry-in is supplied to bit A0/B0 and Carry-out is delivered from bit A7/B7. See Figure 5.

The ALU Logic Control sets the logic to addition and connects each Carry through to the next more significant bit of the byte. The ALU Logic Control is decoded from the Instruction Register. The byte-level “Carry-in” and “Carry-out” entities support multiple-byte arithmetic (described in section 2.3).

In the simple processor of Book 1, Carry-in is always added in the ALU during addition. This is common in real machines, although some have included special add instructions which do not include Carry-in. This means that for the simple processor at the start of addition it is important that the Carry flag is not set. Instructions are provided for the simple processor to control the Carry flag so it can be set to zero at the start of addition.

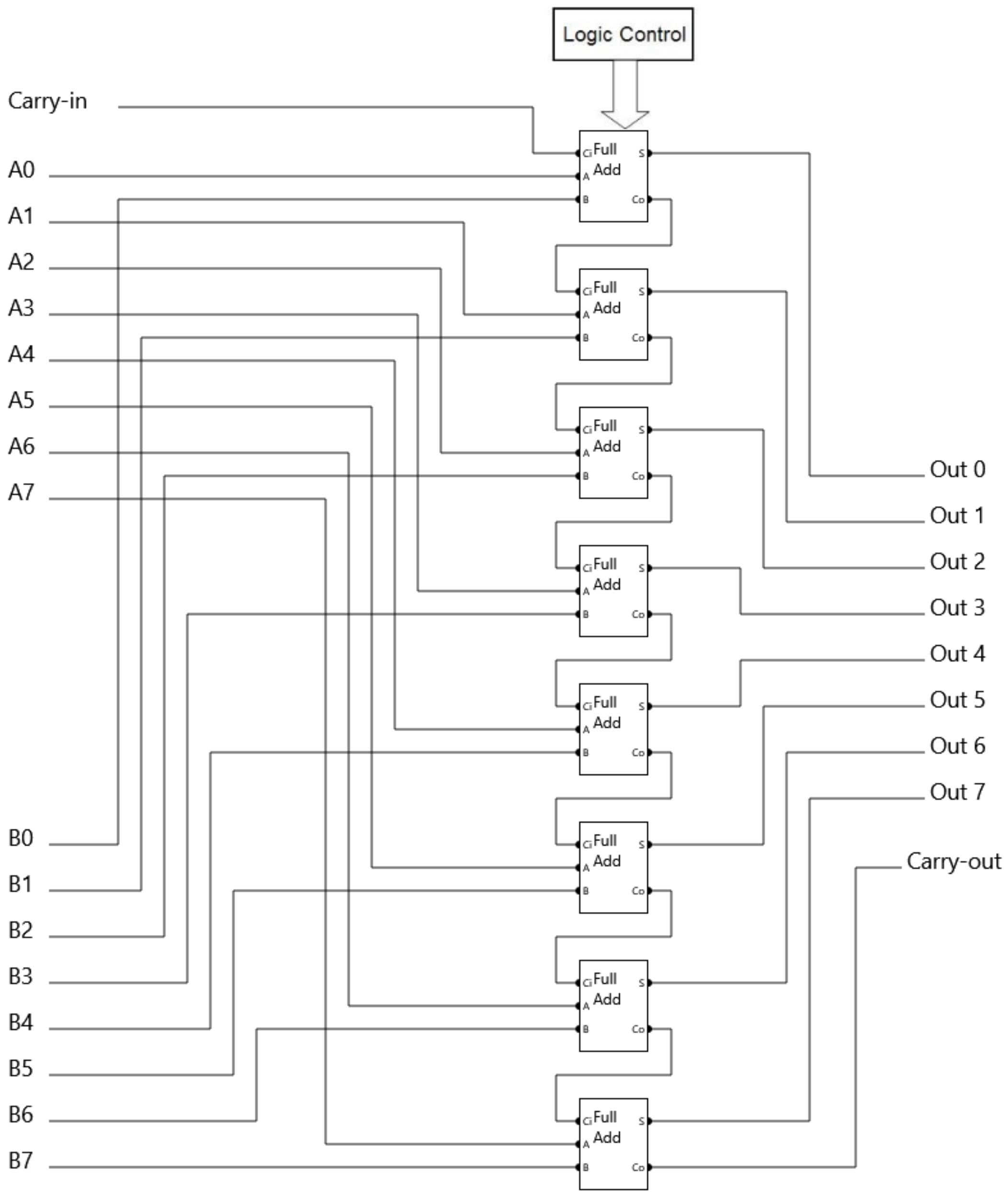


Figure 5 Addition Logic arrangement

2.2.2 Subtraction

Addition is purely a logical operation in the ALU. There is no equivalent logic which provides subtraction in the usual sense. To achieve subtraction, a technique called "complementing" is performed, which achieves subtraction by using addition.

Before describing how this works in the ALU, the idea of complementing is outlined using more familiar decimal numbers. The complement of a number is the difference between that number and the number base, which for decimal is 10 (there are ten different numbers, 0 to 9). For example, the complement of 3 is 7.

Consider the following sum, which leads to a positive answer

8 – 3 = 5

If, instead of subtracting 3 from 8, the number base complement of 3 (i.e. 7) is added to 8, then the result is 15.

8 + 7 = 15

This may appear to be the wrong answer. However, the sum has overflowed the number base, which should be removed leaving 5. This may be more easily understood from the diagram in Figure 6.

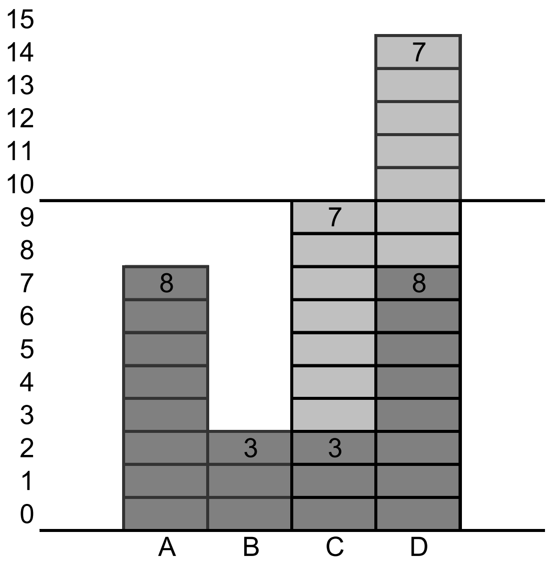


Figure 6 Complementing with positive result

The two numbers in the sum are represented in the diagram by "blocks" in column A (8) and column B (3). The number base is shown by the horizontal line.

Column C shows the number in B and its complement in the lighter grey. The complement is given by the difference between the number and the number base (10). So the complement of B shown in C is 7.

The complement of B is added to A as shown in D. The quantity exceeding the number base is the complement of A subtracted from the complement of B. That is, in effect, 2 is subtracted from 7 leaving 5. Therefore, subtraction can be achieved by adding the complement of the number to be subtracted to the starting number and removing the number base, since this has overflowed.

Now consider an example where the answer is negative.

2 – 8 = – 6

The complement of 8 is 2 and the sum using the complement is

2 + 2 = 4

And the diagram is as shown in Figure 7.

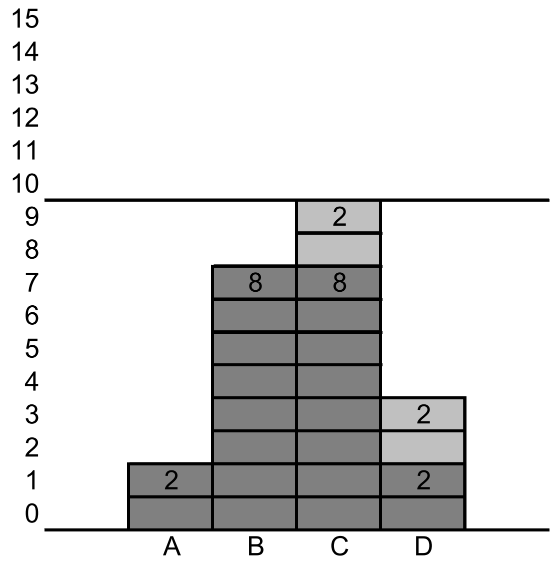


Figure 7 Complementing with negative result

This time in D, the sum does not exceed the number base.

In interpreting this result, it is noted that by adding the complement of B to A, the complement of A is effectively reduced in D by the complement of B. Since the difference between A and B is also the difference between the complements of A and B, the correct answer is the complement of D. That is, the result (4) is the complement of the answer (6). Therefore, when the number base is not exceeded (i.e. no overflow) the answer is negative and is in the complement form.

The complement of 4 is 6 and the answer is – 6.

Of course, the trick here is to find the complement of a number. This does not seem any easier using a base number system like decimal. However, the system is ideal for a machine using binary.

In the binary system of numbers, complementing a number is very straightforward. If every bit in any byte value is inverted (i.e. a 0 becomes a 1 and vice versa) and added to the original byte value, then the answer is FF (hex) or 255 (decimal). An example is shown in Figure 8.

	128	64	32	16	8	4	2	1	Hex
	0	0	1	0	1	1	0	0	2C
Invert bits	1	1	0	1	0	0	1	1	D3
Add	1	1	1	1	1	1	1	1	FF

Figure 8 Inverting a byte and adding

The inverted number is known as the "Ones-complement". So the difference between a byte number value and its number base (256) can be found by inverting the bits (the ones-complement which gives the difference to 255) and adding 1, which gives the complement needed for subtraction and is known as the "Twos-complement".

A more mathematical description shows that for a binary number X which has an inverted form X':

X + X' = (Number Base) – 1

Where Number Base is a power of two (i.e. it depends upon the number of binary bits. A single binary bit is base two. For eight bits together the base is 256). Therefore:

(Number base) – X = X' + 1

When the ALU Logic Control is set to subtract by decoding the Instruction Register, its circuits invert the binary number to be subtracted and add it to the starting number. In the simple processor in Book 1, the extra 1 is added by setting the Carry flag. This is common in real machines although other techniques may also be used. Therefore, at the beginning of subtraction the Carry flag is set for the simple processor.

For example, the diagram Figure 9 shows the sum 16 – 6

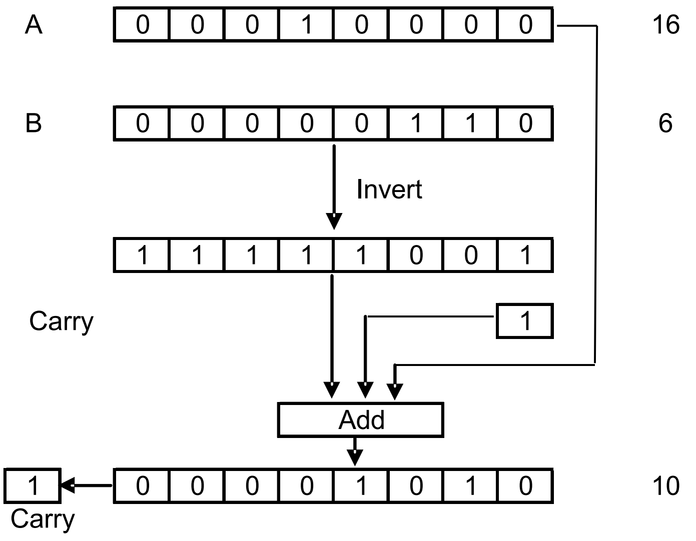


Figure 9 ALU addition positive result

Note that the addition overflows and the carry flag is set, indicating a positive answer. This is the same mechanism as the overflow in the decimal example before.

For the negative case, 6 – 16, the Carry flag at the end of the subtraction is not set and the answer is negative and in complement form. See Figure 10.

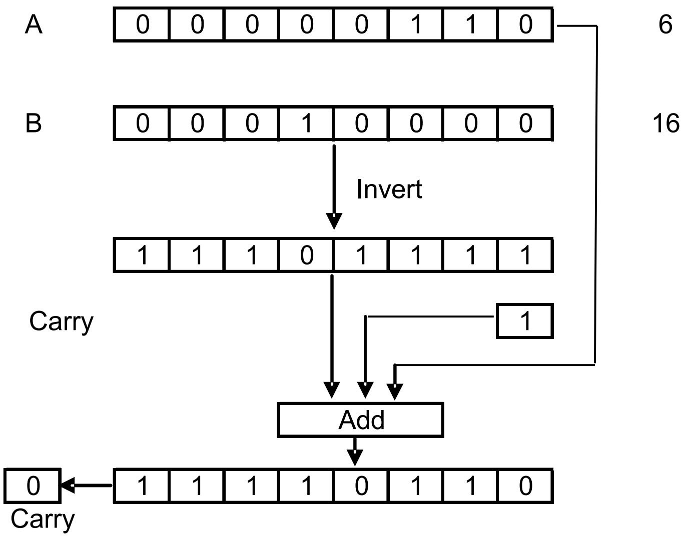


Figure 10 ALU addition negative result

The result of the subtraction from the ALU is F6 with the Carry flag reset to 0, since there is no overflow. Inverting this result and adding 1 gives the expected result 10, which should be interpreted as -10. See Figure 11.

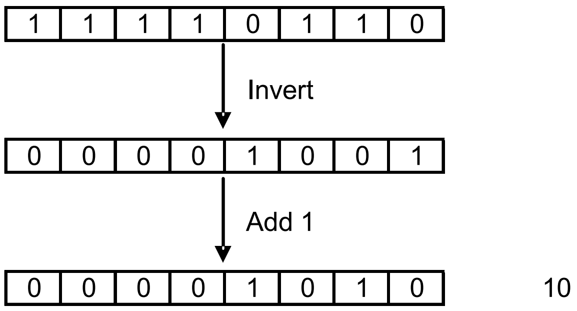


Figure 11 Calculating value from the complement

The numbers held in bytes do not intrinsically signal if the number is positive or negative. The sign of the number is given by the context the arithmetic finds during the calculation (i.e. the result shown by the Carry flag). The programmer records the sign using a chosen convention. For example, the high bit of a byte is sometimes used and this is set to one or zero as required in the program.

For the case where both A and B are the same value the subtraction answer is of course zero. The Carry flag is set indicating that zero is considered to be a positive value.

Finally, another way of thinking about the binary complement is to recognise the manner by which the twos-complement number is represented in this system. For binary numbers in bytes, the values assigned to the results of the twos-complement additions are as shown in Table 7.

Hex	Binary	Decimal	
		Carry set	Carry reset
00	0 0 0 0 0 0 0 0	0	– 256
01	0 0 0 0 0 0 0 1	1	– 255
02	0 0 0 0 0 0 1 0	2	– 254
…			
FE	1 1 1 1 1 1 1 0	254	– 2
FF	1 1 1 1 1 1 1 1	255	– 1

Table 7 Positive integer and complement ranges

The positive numbers run to 255 but effectively negative numbers run to 256. This is because 0 is positive only. So the binary complement of a number is found by inverting the binary bits and adding 1.

2.3 Multiple-Byte Addition and Subtraction

2.3.1 Addition

For example: Add 511 to 255

The numbers can be represented over two bytes (16 binary bits) arranged such that there is a higher value byte (top eight bits of number) and lower value byte (bottom eight bits of number).

The two numbers are held in two pairs of bytes (Figure 12).:
511 is 01FF (hex)
255 is 00FF (hex)

	Binary		Hex	
	Hi Byte	Low Byte	Hi Byte	Low Byte
511	00000001	11111111	01	FF
255	00000000	11111111	00	FF

Figure 12 Representation of larger numbers

Therefore, multiple-byte addition is required and works as shown in Figure 13.

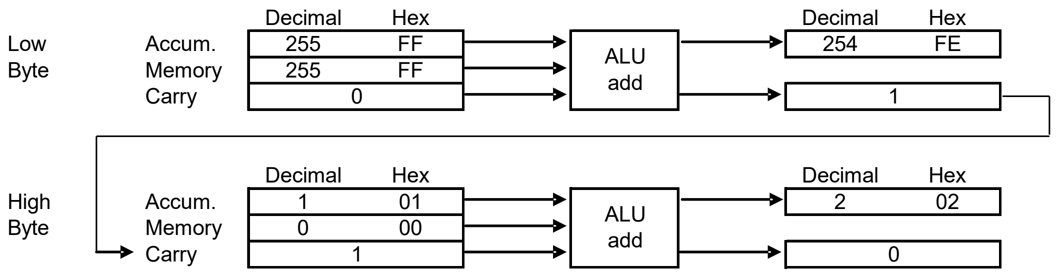


Figure 13 Adding larger numbers with overflow

When the low bytes are added (FF + FF Hex) the result is FE (Hex) with an overflow. The Carry flag is set and this is included in the addition of the high bytes (01 + 00 + Carry) with the result 02. The answer is therefore 02FE (Hex).

For those unfamiliar with hexadecimal, the two numbers 255 when added make 510. An overflow occurs, which is signalled by Carry and takes 256 into the higher sum. This leaves 510 – 256 = 254 in the lower byte. The high byte is 2 x 256 = 512 (decimal) and so 02FE (hex) is 512 + 254 = 766 (decimal).

On completion of the sum the carry flag is zero, indicating no further Carry forward. If the Carry flag is set after the second byte addition, then a third byte is required and 65,536 is carried forward (2 to the power 16) and so on.

An example program for the simple processor is shown in Table 8. The Instruction Codes and labels are described in Book 2 Part 1.

Label	Ins	Op
XH	EQ	1
XL	EQ	255
YH	EQ	0
YL	EQ	255
	CLC	
	LDI	XL
	ADI	YL
	STA	ANSL
	LDI	XH
	ADI	YH
	STA	ANSH
	HLT	
ANSH	&	
ANSL	&	

Table 8 Program demonstrating multiple-byte addition

2.3.2 Subtraction

When performing multiple-byte subtraction, it is possible for partial results (that is the subtraction of lower bytes) to be less than zero. For example, 514 – 255 means the subtraction of two-byte hex numbers 0202 and 00FF respectively. Clearly, for the low bytes, 02 – FF is less than zero (the result is 03). This is dealt with through the use of the Carry flag.

At the beginning of subtraction the Carry flag is set to one. This is to achieve twos-complementing but can also be considered as indicating to the ALU that the number to be subtracted is positive.

If the result of a subtraction is less than zero, the ALU “borrows” the value in the Carry flag which effectively increases the smaller value (by 256) making the sum in this example (effectively) 258 – 255 resulting in 3. The Carry flag is reset to zero if this occurs (it has been “borrowed”) and the ALU accounts for this on the next higher byte subtraction (Figure 14).

Of course, what is actually occurring is that the FF is twos-complemented (resulting in 01) and added to 02. The Carry flag is not set. This amounts to the same result.

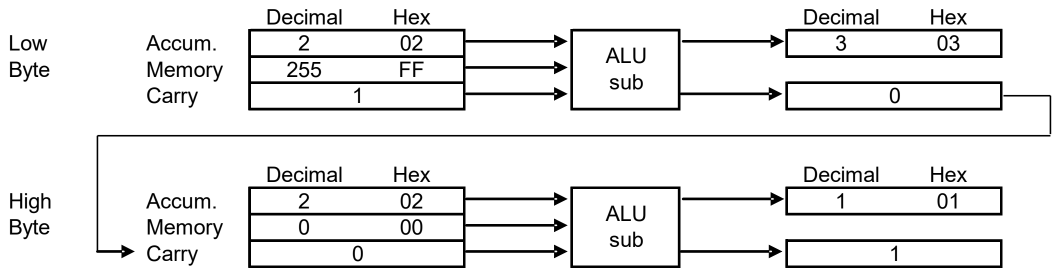


Figure 14 Subtracting larger numbers with borrow

On the second (Hi Byte) subtraction the ALU subtracts one from the Accumulator value to “repay” the Carry flag. So the result of the subtraction is: 1 x 256 (High byte) plus 3 (Low byte) equals 259 (decimal) or 0103 (hex). The Carry flag is set to one which indicates a positive result.

Again, the mechanism behind this is that the number to be subtracted is twos-complemented. However, the Carry is zero and thus 0 is complemented to FF, which is added to 2 resulting in 1 with the Carry flag set.

If the Carry flag is set to zero at the end of subtraction, then the result is negative and in the complement form over both bytes.

The example program in Table 9 reformats negative answers from the complement to show them in the more usual format. The final state of the Carry flag indicates if the answer is positive (set to one) or negative (set to zero). Care has been taken with the Carry flag in the design of this program.

Label	Ins	Op
XH	EQ	2
XL	EQ	2
YH	EQ	0
YL	EQ	255
	SEC	
	LDI	XL
	SBI	YL
	STA	ANSL
	LDI	XH
	SBI	YH
	STA	ANSH
	BCS	here
	LDA	ANSL
	XOI	#FF
	ADI	#01
	STA	ANSL
	LDA	ANSH
	XOI	#FF
	ADI	#00
	STA	ANSH
	CLC	
here	HLT	
ANSH	&	
ANSL	&	

Table 9 Program demonstrating multiple-byte subtraction

2.4 Multiplication

Multiplication can be achieved using electronic hardware that delivers results at great speed. However, such devices are beyond the scope of these books. Generally, if multiplication is required then an algorithm (i.e. a logical sequence of instructions forming part of a program) is developed to deliver the feature. Two examples are shown here.

Multiplication is further complicated by rules surrounding the signs of the two numbers (i.e. if the signs are the same the answer is positive, if the signs are different the answer is negative). It is for the programmer to deal with sign in a manner which fits the application.

A simple form of multiplication is the addition of the multiplicand (first number) to itself a number of times as given by the multiplier (second number). For example, a count could be set to the value of the multiplier and decremented each time the addition occurs. When the count reaches zero the addition would cease. An example algorithm (32 bytes) is shown in Table 10.

Label	Ins	Op
x	EQ	12
y	EQ	5
	LDI	#00
	STA	answer
	LDI	x
	BEZ	end
	LDI	y
	STA	count
	BEZ	end
loop	LDA	answer
	CLC	
	ADI	x
	STA	answer
	LDA	count
	SEC	
	SBI	#01
	STA	count
	BNZ	loop
end	LDA	answer
	HLT	
answer	&	
count	&	

Table 10 Program demonstrating multiplication by repeated addition

With the values chosen (12 x 5) the program executes in 148 cycles. The loop executes the number of times given by the multiplier (in this case 5). The loop is 25 cycles so it is to be expected that a multiplier of 21 (the maximum before overflow) would take the cycle count to 548 cycles. So the program would take nearly four times longer to run.

Another algorithm is based on halving and doubling. The multiplicand is doubled and the multiplier halved with partial results added. This technique is well suited to binary systems since halving and doubling is easily achieved (shift right and shift left respectively).

Table 11 shows the multiplication as a series of shifts and adds in a similar manner to drawing a long multiplication sum. The binary multiplicand is shifted left (multiplied by 2) as each more significant bit is of the multiplier is processed. Only the “1” bits in the multiplier add any value to the sum.

If a “1” bit in the multiplicand is shifted out of the Accumulator then the sum overflows (answer is greater than 255). Further higher bytes would be required to support multiple-byte multiplication.

Decimal	Binary	Hex
12	0 0 0 0 1 1 0 0	0C
5	0 0 0 0 0 1 0 1	05
x	0 0 0 0 1 1 0 0	
	0 0 0 0 0 0 0	
	0 0 1 1 0 0	
	0 0 0 0 0	
	0 0 0 0	
	0 0 0	
	0 0	
	0	
Sum 60	0 0 1 1 1 1 0 0	3C

Table 11 Binary multiplication

The “1” bits in the multiplier are tested by a right shift (divide by 2) and testing the Carry flag which will be set to the value of each bit in turn. The multiplicand is shifted to the left (multiplied by 2) for each shift in the multiplier. The results are added to the sum. An example algorithm (48 bytes) is shown in Table 12.

This program consists of 316 cycles for the values 12 x 5 and 328 for 12 x 21. The algorithm is more consistent with its processing and much more efficient with larger calculations than the first example.

Label	Ins	Op
x	EQ	12
y	EQ	5
	LDI	#00
	STA	answer
	LDI	x
	STA	mcand
	BEZ	end
	LDI	y
	STA	multip
	BEZ	end
	LDI	#08
	CLC	
loop	STA	count
	LDA	multip
	SHR	
	STA	multip
	BCN	here
	CLC	
	LDA	mcand
	ADC	answer
	STA	answer
here	LDA	mcand
	SHL	
	STA	mcand
	LDA	count
	SEC	
	SBI	#01
	BNZ	loop
	LDA	answer
	HLT	
answer	&	
mcand	&	
multip	&	
count	&	

Table 12 Program demonstrating multiplication by halving and doubling

2.5 Division

Division is a little trickier than the other arithmetic forms since the answer generally consists of two parts: a quotient and remainder. Furthermore, the calculation is complicated by the sign of the numbers (as multiplication) and by the possibility that the divisor is zero (which would produce an error since the answer is indeterminate). The program in Table 13 demonstrates the additional complexity.

Like multiplication, many algorithms have been developed to deal with arithmetic division. Since number theory is somewhat beyond the scope of these books, only the simple example of multiple divisor subtraction is included here. The algorithm does give a feel to the exception handling required for division. Error detection provides a mechanism for recovery but simply halts the program in this example.

The quotient is contained in the memory location “answer” and the remainder in “rem”.

Label	Ins	Op
x	EQ	12
y	EQ	5
	LDI	#00
	STA	rem
	STA	answer
	LDI	y
	BEZ	error
	LDI	x
	BEZ	end
	STA	rem
loop	SEC	
	SBI	y
	BCN	end
	STA	rem
	LDA	answer
	CLC	
	ADI	#01
	STA	answer
	LDA	rem
	BNZ	loop
end	HLT	
error	HLT	
rem	&	
answer	&	

Table 13 Program demonstrating complexity of division

3 The Control System

The processor operates by a sequence of instructions being fetched and executed as shown in Part 1. The procedure mimics a clock: Fetch; Execute (tick; tock). In fact, the electronic control systems driving the process are indeed driven by a repeating electronic pulse referred to as the “clock”. Every transition of the clock leads to a furthering of the program execution.

After a processor reset the machine is placed in the Fetch state by the system electronics. When it starts to run, the processor enters the fetch cycle and on completion of the cycle sets the Execute state. The last action of the final execute cycle is to set the Fetch state and the cycles repeat.

There are many ways that the control system can be designed and implemented. This section describes the method chosen for the implementation of the simple processor in Book 3 as an illustration of the kind of approach typical in machines of this type.

3.1 Fetch State

The Fetch state is always the same and involves opening Gates I and F and reading memory. Following this the Program Counter is incremented.

Before the Execute state is invoked it is necessary to determine the number of machine cycles there are to execute. The Machine Code could indicate the figure directly by including the count in its coding. However, that approach would restrict the availability of codes. A better approach is to use the Machine Code as an index into a look-up table which indicates the number of cycles, and provides an effective way of processing the execute cycles too.

The Instruction Register (IR) output code forms the high eight-bits of an 11 bit address into a special look-up memory (not the same as the processor memory) where the microcode is stored (Figure 15). The bottom three bits of the look-up are set by a counter, which is zero in the Fetch state. This means that the output of the IR (i.e. the instruction Machine Code) addresses blocks of eight bytes in the Microcode ROM (read-only memory). Each byte within an eight-byte block is addressed by the counter. An 11 bit address ROM provides 2048 bytes of look-up memory. Although considerably fewer bytes are required a ROM is an inexpensive and simple solution to the design.

During a Fetch cycle the IR code is loaded into the IR and decodes the low byte of the eight-byte block in the look-up (Counter output is zero). Gates I and F are explicitly decoded by the system electronics during the Fetch cycle. The look-up ROM output is eight bits arranged as shown in Table 14.

Fetch State:

b7	b6	b5	b4	b3	b2	b1	b0
x	Inc PC	x	x	x	No. of MC for instruction		

x – Not used during the fetch cycle.

Table 14 ROM output byte - Fetch

Bits b0 to b2 are loaded into the counter. At the end of the Fetch cycle the counter is set to the number of cycles to be executed, but the output of the counter is not enabled (the bits a0 to a2 in Figure 15 remain zero). In this design the maximum number of cycles possible is seven.

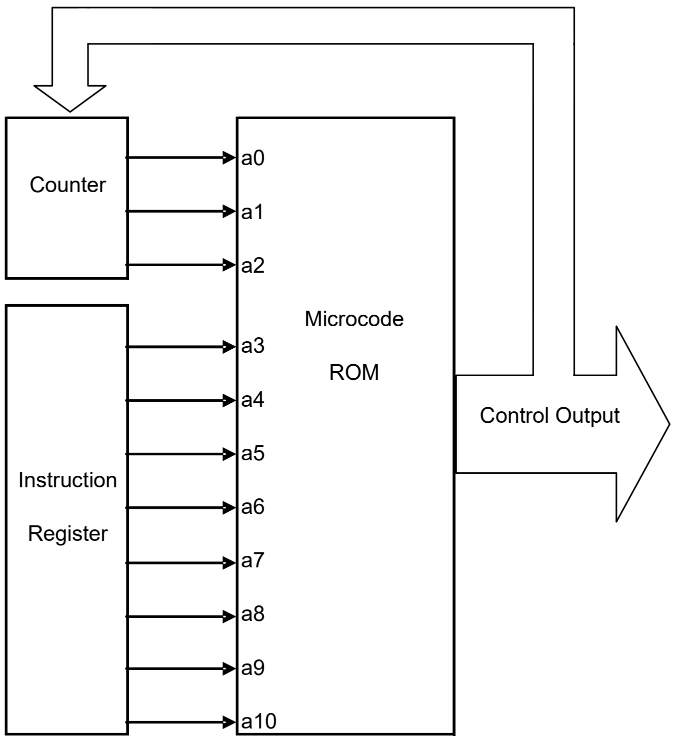


Figure 15 Machine Code look-up to blocks of 8 bytes in ROM

The only other bit used in the eight-bit output is b6 which controls the increment of the PC. This bit is set for all Fetch Microcode ROM locations except for the HLT instruction, which provides the mechanism of the instruction (i.e. the PC never moves beyond the instruction).

On completion of the Fetch cycle (Gates I and F closed) the Execute state is enabled which enables the output of the counter.

3.2 Execute State

At the beginning of the execute cycle the IR and counter decode a byte in the Microcode ROM. For an instruction consisting of three machine cycles (e.g. LDA) the byte pointed to is two bytes above the byte read during the fetch cycle (two more cycles to run). In this design the counter is decremented after every cycle, including fetch.

For the Execute state the Microcode ROM output is defined as shown in Table 15.

Execute State:

b7	b6	b5	b4	b3	b2	b1	b0
Int Op	Inc PC	x	Gate I	Gate J	Decode Gates		

x – Not used during the execute cycle. Note that b5 is effectively “spare”.

Table 15 ROM output byte - Execute

Bits b0 to b2 “Decode Gates” values select the Gate to be opened as shown in Table 16.

Value	Gate opened
0	None
1	B
2	G
3	E
4	C
5	A
6	H
7	None

Table 16 Decode Gates

Only Gates I and J can be opened at the same time as B to H. The opening of Gate H may be subject to conditional processing.

For each Execute cycle, the Gates to be opened are identified (all others are closed), whether the Memory is to be accessed (b7 not set), and whether the Program Counter is to be incremented (b6 set). Selecting Gate C implies a memory write and all other selections are memory reads.

However, this is not the complete story. Some instructions open the same sequences of gates, memory access etc but process the data differently. Specifically, the four instruction types shown in Table 17 contain groups of instructions which have the same ROM output bytes:

ALU	**Accumulator**	**Carry**	**Branch**
Add Subtract AND OR Exclusive OR	Shift Right Shift Left Rotate Right Rotate Left	Clear Carry Set Carry	Branch if Carry Set Branch if Carry not set Branch if Zero Branch if not Zero Branch Unconditionally

Table 17 Instruction types with same ROM outputs

Therefore, the instruction must also decode the instruction type and what specific processing is to be performed.

For the ALU there are two groups of instructions: one group using immediate data; one group using addressed data. This means that there are two groups of ALU instructions with sets of gate sequences which slightly differ from each other.

This is all achieved by further decoding of the instruction in the Instruction Register. There are many ways to achieve this and a simple illustration is described here.

The Machine Code consists of eight bits. Here the most-significant three bits are used to decode the type of instruction and is used to identify the instruction group. The low three bits are used to identify the type of processing to be performed. The values picked are arbitrary but (with experience) take into account that electronic circuits are required to realise the functions and are designed to minimise complexity and cost.

The chosen codes are shown in Table 18.

Not all instruction types need to be decoded in this way. Only those instruction types requiring bespoke processing need to decode the Instruction Register further. The decoded instruction type enables or disables the processing electronics in the ALU, Accumulator, Carry flag and Program Counter update (Branch) as appropriate. The processing type is decoded as required directly by linking b0 to b2 to each of the register electronics.

IR Code	b7	b6	b5		b2	b1	b0
None	0	0	0		x	x	x
Future	0	0	1	not used	x	x	x
ALU	0	1	0	Add	0	0	0
ALU	0	1	0	Sub	0	0	1
ALU	0	1	0	not used	0	1	0
ALU	0	1	0	not used	0	1	1
ALU	0	1	0	not used	1	0	0
ALU	0	1	0	Xor	1	0	1
ALU	0	1	0	Or	1	1	0
ALU	0	1	0	And	1	1	1
Accumulator	0	1	1	Shift Right	0	0	0
Accumulator	0	1	1	Shift Left	0	0	1
Accumulator	0	1	1	Rotate Right	0	1	0
Accumulator	0	1	1	Rotate Left	0	1	1
Accumulator	0	1	1	not used	1	0	0
Accumulator	0	1	1	not used	1	0	1
Accumulator	0	1	1	not used	1	1	0
Accumulator	0	1	1	not used	1	1	1
Carry	1	0	0	Clear	0	0	0
Carry	1	0	0	Set	0	0	1
Carry	1	0	0	not used	0	1	0
Carry	1	0	0	not used	0	1	1
Carry	1	0	0	not used	1	0	0
Carry	1	0	0	not used	1	0	1
Carry	1	0	0	not used	1	1	0
Carry	1	0	0	not used	1	1	1
Branch	1	0	1	Carry Set	0	0	0
Branch	1	0	1	Carry Not Set	0	0	1
Branch	1	0	1	Equals Zero	0	1	0
Branch	1	0	1	Not Equals Zero	0	1	1
Branch	1	0	1	Always	1	0	0
Branch	1	0	1	not used	1	0	1
Branch	1	0	1	not used	1	1	0
Branch	1	0	1	not used	1	1	1
Future	1	1	0	not used	x	x	x
Future	1	1	1	not used	x	x	x

Table 18 Instruction type and processing

The corresponding Instruction Codes are as Table 19, which shows all the bits in the instruction byte (in hex). The ALU instructions include memory operations that operate on immediate or addressed data.

Function	Instruction	Hi 4 bits	Low 4 bits
Halt	HLT	0	0
Read Memory	LDI	1	0
	LDA		1
	LDX		3
Write Memory	STA	2	0
	STX		2
ALU	ADC	4	0
	SBC		1
	XOR		5
	ORA		6
	AND		7
	ADI		8
	SBI		9
	XOI		D
	ORI		E
	ANI		F
Accumulator	SHR	6	0
	SHL		1
	ROR		2
	ROL		3
Carry	CLC	8	0
	SEC		1
Branch	BCS	A	0
	BCN		1
	BEZ		2
	BNZ		3
	BRA		4

Table 19 Instruction Codes

There are many other possible codes (since only 27 are used out of 256 possibilities). All other codes are illegal and could cause the processor to malfunction if the design does not prevent this. The first byte of the unused code ROM groups could be set to resemble a HLT. Alternatively, a ROM containing all zeroes in unused bytes effectively halts operation.

The decoded instruction determines the type of processing over all the execute cycles.

As each execute cycle completes the counter is decremented to point to the next (lower) Microcode ROM address and hence the set of Gates and activities to occur for the cycle. When the counter is zero, the Execute is complete and the machine is placed back into the Fetch state.

In conclusion, the complete controller in the execute cycle looks like Figure 16. The connection back to the counter is shown but is not enabled during execute.

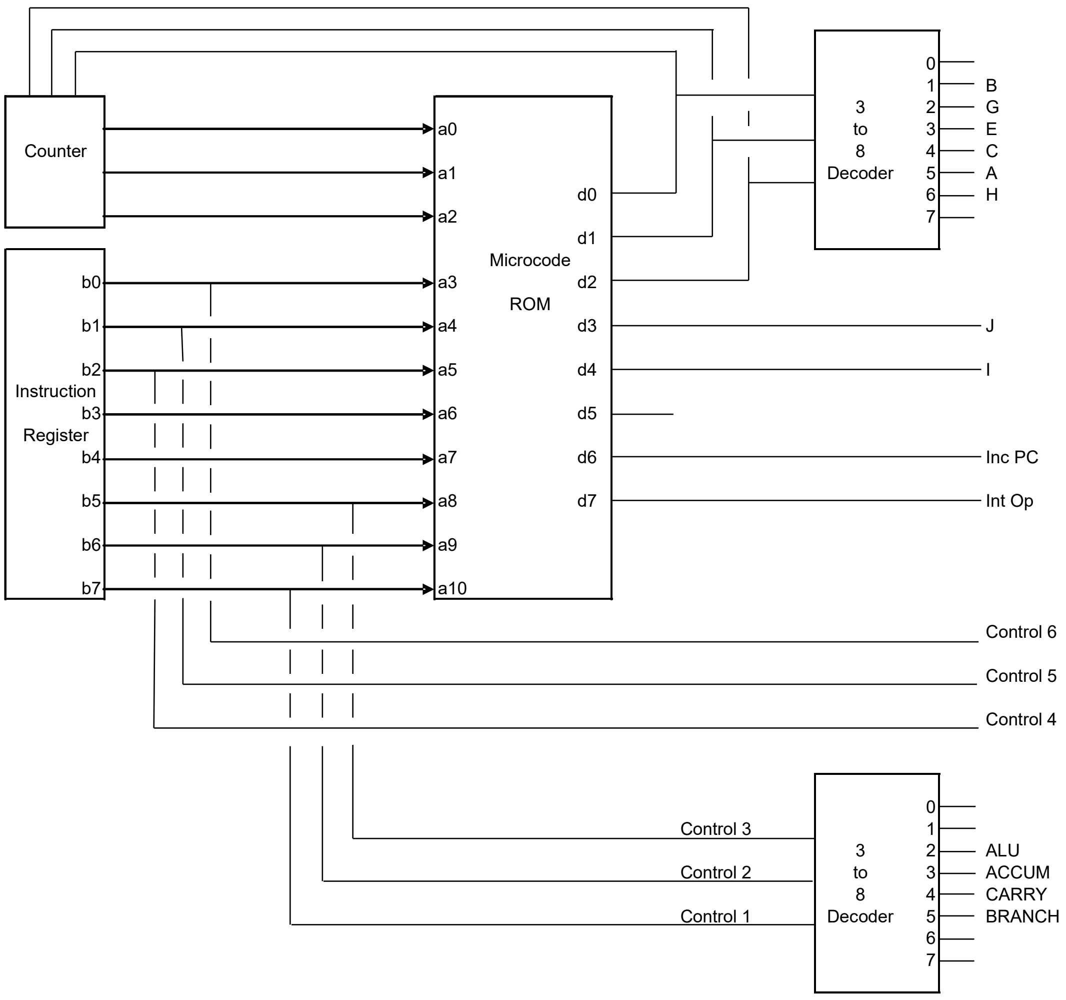


Figure 16 Full decode of the Instruction Register

Figure 16 omits the controlling electronics for the Fetch/Execute states and control of the decoders. The circuit is described in Book 3 Part 2. To summarise: The three most significant bits of the IR code (labelled Control 1, 2 and 3 in the figure) decode the instruction type in the processor. The three least significant bits (labelled Control 4, 5 and 6 in the figure) are passed to each of the register circuits (ALU, ACCUM, CARRY and BRANCH) which decode the operation directly. These are derived from Table 18.

The write-memory instructions (20 and 22 hex) set b7b6b5 to 001 in Table 18 which decodes to output 1 on the 3-to-8 Decoder in Figure 16. This design means that conceptual future expansion on output 1 is unlikely.

The instructions and their microcode are shown in Table 20. Increment Program Counter is depicted as “PC=PC+1”. “Int Op” represents Internal Operation and disables processor access to memory in the simulator Book 3.

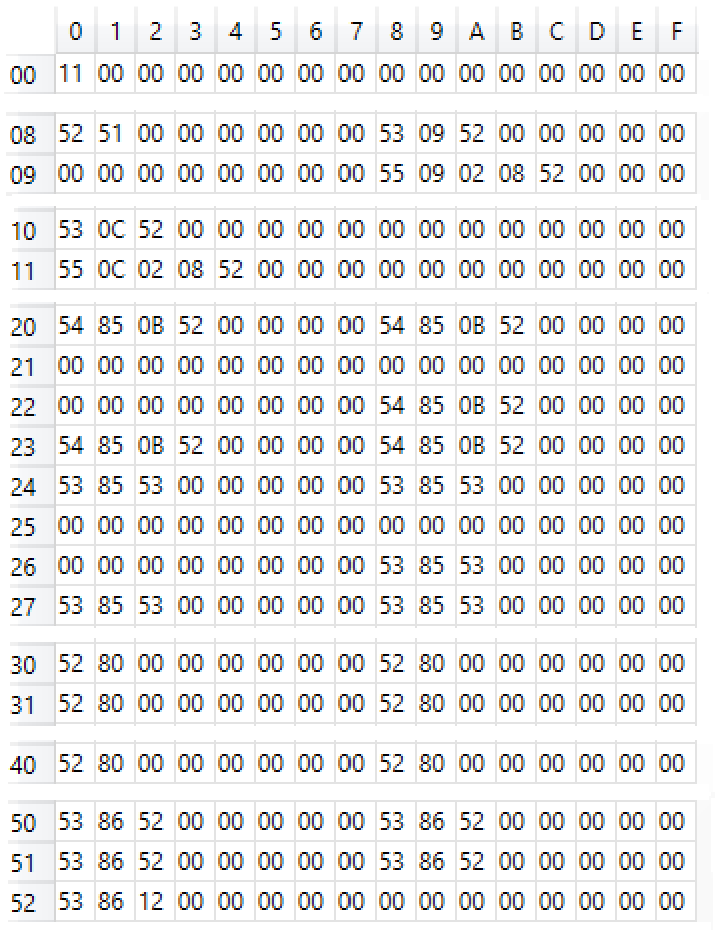
Inst.	Code	MC	Fetch	Execute 1	Execute 2	Execute 3	Execute 4
HLT	00	1	I, F				
LDI	10	2	I, F PC=PC+1	I, B PC=PC+1			
LDA	11	3	I, F PC=PC+1	I, G PC=PC+1	J, B		
LDX	13	5	I, F PC=PC+1	I, G PC=PC+1	J	G	J, B
STA	20	3	I, F PC=PC+1	I, G PC=PC+1	J, C		
STX	22	5	I, F PC=PC+1	I, G PC=PC+1	J	G	J, C
ADC	40	4	I, F PC=PC+1	I, G PC=PC+1	J, E	A Int Op	
SBC	41	4	I, F PC=PC+1	I, G PC=PC+1	J, E	A Int Op	
XOR	45	4	I, F PC=PC+1	I, G PC=PC+1	J, E	A Int Op	
ORA	46	4	I, F PC=PC+1	I, G PC=PC+1	J, E	A Int Op	
AND	47	4	I, F PC=PC+1	I, G PC=PC+1	J, E	A Int Op	
ADI	48	3	I, F PC=PC+1	I, E PC=PC+1	A Int Op		
SBI	49	3	I, F PC=PC+1	I, E PC=PC+1	A Int Op		
XOI	4D	3	I, F PC=PC+1	I, E PC=PC+1	A Int Op		
ORI	4E	3	I, F PC=PC+1	I, E PC=PC+1	A Int Op		
ANI	4F	3	I, F PC=PC+1	I, E PC=PC+1	A Int Op		
SHR	60	2	I, F PC=PC+1	Int Op			
SHL	61	2	I, F PC=PC+1	Int Op			
ROR	62	2	I, F PC=PC+1	Int Op			
ROL	63	2	I, F PC=PC+1	Int Op			
CLC	80	2	I, F PC=PC+1	Int Op			
SEC	81	2	I, F PC=PC+1	Int Op			
BCS	A0	3	I, F PC=PC+1	I, G PC=PC+1	H Int Op		
BCN	A1	3	I, F PC=PC+1	I, G PC=PC+1	H Int Op		
BEZ	A2	3	I, F PC=PC+1	I, G PC=PC+1	H Int Op		
BNZ	A3	3	I, F PC=PC+1	I, G PC=PC+1	H Int Op		
BRA	A4	3	I, F PC=PC+1	I, G	H Int Op		

Table 20 Complete Simple Processor Instruction Code Microcode

It follows that the map of the Microcode ROM is as Figure 17. The map only includes address lines that contain values other than zero. All lines not shown contain all zero-filled bytes in this design.

The Machine Code maps to the addresses shown by shifting the code three bits left (the low bits addressed by the counter). For example, LDI is Machine Code 10 becomes 80 and maps to 080 in Figure 17. LDA maps to 088.

In the ROM, bit b4 is set for all the fetch cycles. This bit is not used during fetch but controls Gate I during the execute cycle. However, the emulator in Book 2 uses this bit to control Gate I always and was used to generate this table. It has no impact on the simulator design (Book 3).



	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00	11	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
08	52	51	00	00	00	00	00	00	53	09	52	00	00	00	00	00
09	00	00	00	00	00	00	00	00	55	09	02	08	52	00	00	00
10	53	0C	52	00	00	00	00	00	00	00	00	00	00	00	00	00
11	55	0C	02	08	52	00	00	00	00	00	00	00	00	00	00	00
20	54	85	0B	52	00	00	00	00	54	85	0B	52	00	00	00	00
21	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
22	00	00	00	00	00	00	00	00	54	85	0B	52	00	00	00	00
23	54	85	0B	52	00	00	00	00	54	85	0B	52	00	00	00	00
24	53	85	53	00	00	00	00	00	53	85	53	00	00	00	00	00
25	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
26	00	00	00	00	00	00	00	00	53	85	53	00	00	00	00	00
27	53	85	53	00	00	00	00	00	53	85	53	00	00	00	00	00
30	52	80	00	00	00	00	00	00	52	80	00	00	00	00	00	00
31	52	80	00	00	00	00	00	00	52	80	00	00	00	00	00	00
40	52	80	00	00	00	00	00	00	52	80	00	00	00	00	00	00
50	53	86	52	00	00	00	00	00	53	86	52	00	00	00	00	00
51	53	86	52	00	00	00	00	00	53	86	52	00	00	00	00	00
52	53	86	12	00	00	00	00	00	00	00	00	00	00	00	00	00

Figure 17 Complete ROM Micro Code

Only 81 bytes out of the 2,048 bytes of the ROM are programmed. All other bytes are zero.

The values in the map appear in the implementation in Book 3. During the fetch cycle the counter is loaded with the count of the cycles for the instruction. This is because in the implementation the counter decrements the count on every machine cycle including the fetch.

Further description of the processor control system can be found in Book 3.

4 More Advanced Processor Features

The simple processor contains the features required to perform digital computation. However, the simple architecture (developed here to be easily understood) does not necessarily deliver efficient and effective computing. Furthermore, its limited memory capacity does not make for a particularly practical machine. This section shows how some additional processor features can enhance the processing.

Note that even the enhanced machine described here is still far short of the capability of modern processors. Modern processors typically contain 64-bit data and address buses (an astronomic memory capability) and greatly enhanced processing speed techniques. Section 5 briefly discusses the development of the modern processor.

The simple processor is enhanced by providing 16-bit address space for memory and four features which are typically used in commercial processors:

- Data address indexing
- Subroutine handling
- Interrupt processing
- Input/output device handling

Each of the features is described in turn by adding them to the simple processor until all are present in the final upgraded processor.

4.1 16-bit Address Space

Providing a 16-bit Address Bus Register expands the memory capability to 65536 bytes, which is typical of small commercial 8-bit data processors. For the simple processor in Book 1 this means the accompanying registers Program Counter and Address Register also need to be 16-bit registers.

The other impacts of such an upgrade are as follows.

- Operand addresses need to be 16 bits, which can be accomplished by moving to two-byte address operands where required. This means instructions may contain no, one-byte or two-byte operands.
- Indirect addressing obtains an address from data memory and so needs two eight-bit read operations. The current simple implementation will not suffice.

The two-byte address operand can be achieved by providing an extra G gate into the Address Register so that the High and Low 8-bits of the address can be handled separately. The microcode can be altered to handle an extra machine cycle and appropriate Gate operation.

Indirect addressing requires a buffer register for the first of the two bytes read which will be placed in the Address Register. The Address Register also needs the capability to increment (similar to the Program Counter) so that the second address byte can be read in this mode. The increment is controlled by the microcode.

The figure shows the updated processor. The 16-bit registers are marked. Gates I and J are 16 bits. All other registers remain 8-bit.

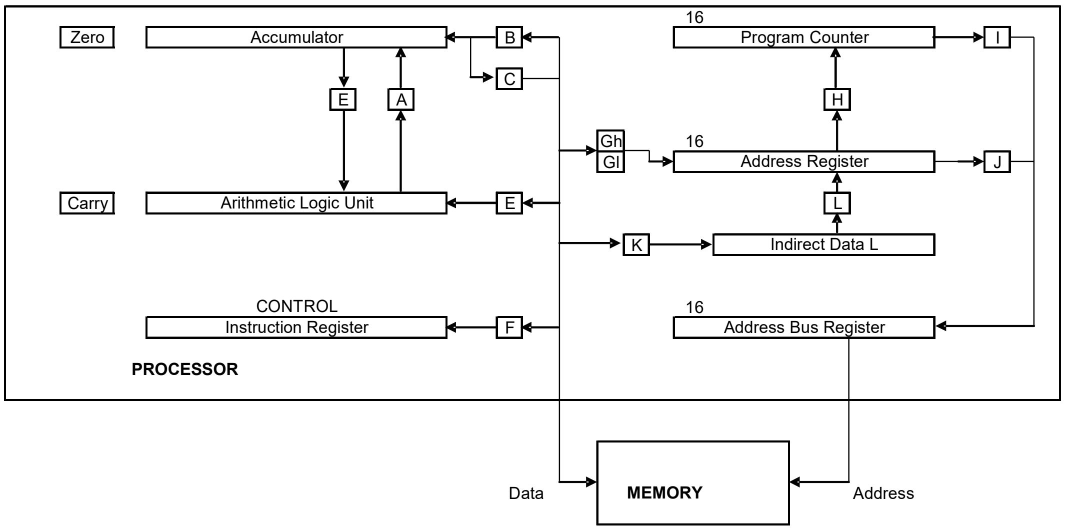


Figure 18 Simple Processor with 16-bit address space

The function of the additional Gates when open is shown in Table 21.

Gh	Data on the data bus is placed in the high 8 bits of the Address Register.
Gl	Data on the data bus is placed in the low 8 bits of the Address Register.
K	Data on the data bus is placed in the Indirect Data L register.
L	Data in the Indirect Data L register is placed in the low 8 bits of the Address Register.

Table 21 additional Gates for 16 bit addresses

Instructions which provide an address are three bytes. For example, Load Accumulator from Address (LDA) is of the form

LDA Address

where Address is two 8-bit bytes. If the memory address is, say, 0FFF, then the machine code could be either of the following

- 11 0F FF – The 16 bit address is specified as high byte followed by low byte (and known as “Big-endian”).
- 11 FF 0F – The 16 bit address is specified as low byte followed by high byte (and known as “Little-endian”).

Little-endian has been common in small 8-bit machines and is used here. Figure 18 shows the buffer register Indirect Data L (eight bits) collecting the low byte of the address. The differences, characteristics and history of each endian type can be found in Wikipedia.

4.1.1 LDA, STA – direct addressing

Loading the Address Register with a two-byte operand requires an additional machine cycle. Figure 19 and Figure 20 show the two cycles required to load the Address Register after which the data in memory is loaded into the Accumulator by opening Gates J and B as before.

Storing the Accumulator to memory follows the same addressing procedures but writes the Accumulator to memory by opening Gate C as before.

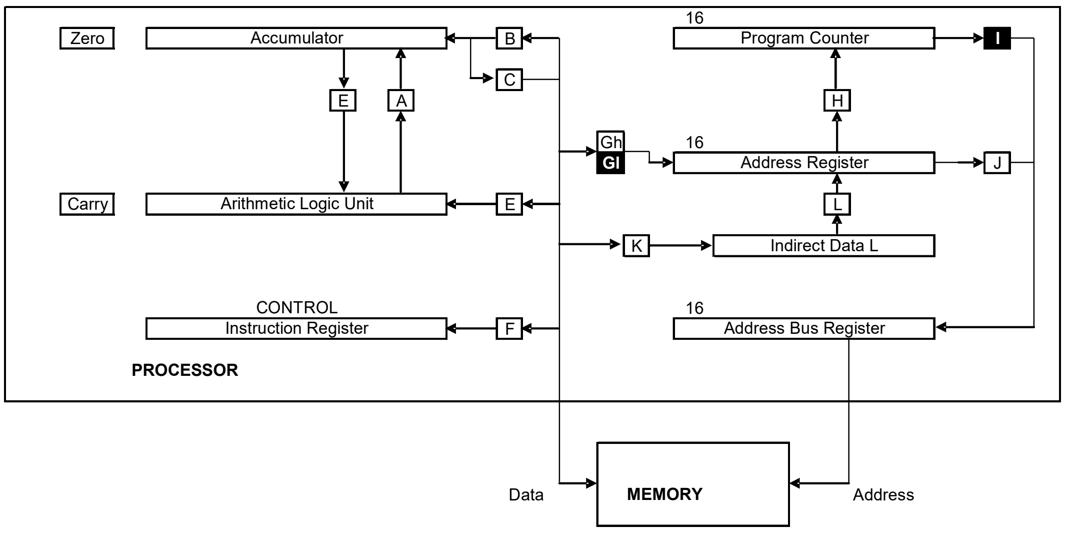


Figure 19 Loading Low Byte of address into Address Register

In the execute cycle Figure 19 Gates I and Gl are opened and data is loaded into the low address of the Address Register. The Program Counter is incremented.

In the execute cycle Figure 20 Gates I and Gh are opened and data is loaded into the high address of the Address Register. The Program Counter is incremented.

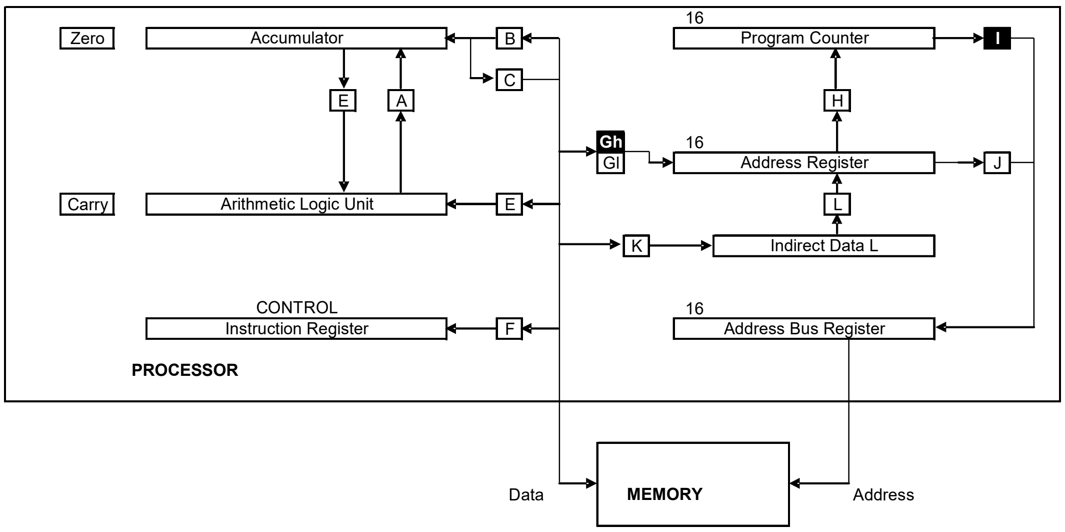


Figure 20 Loading High Byte of address into Address Register

4.1.2 LDX, STX – indirect addressing

The Address Register is loaded with the 16-bit address of the data address in the manner described in the previous section. However, the data address is also 16 bits and occupies the byte addressed and the byte following. Both bytes need to be loaded into the Address Register so that the location of the data is identified.

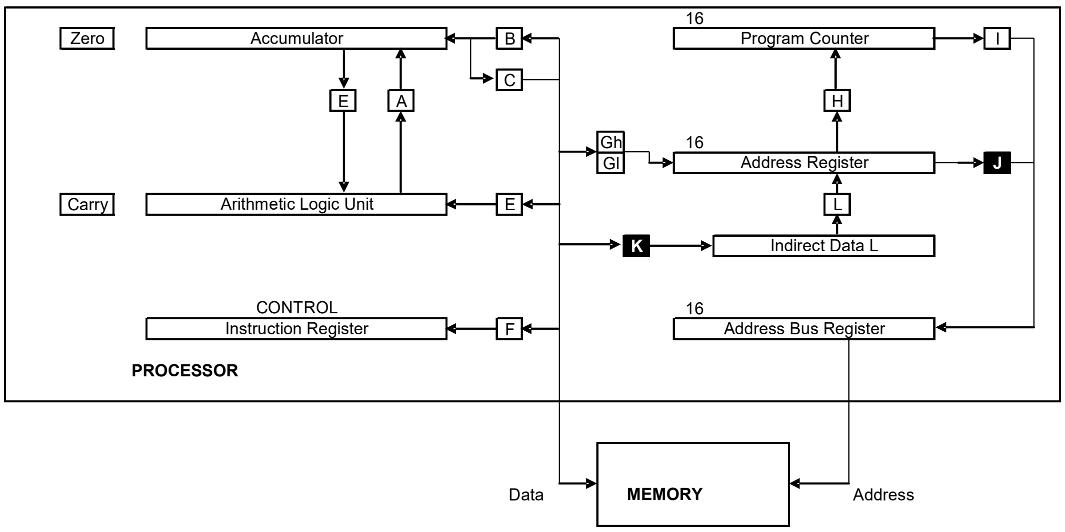


Figure 21 Loading the low byte data address into the buffer

The first read operation places the low address byte into the “Indirect Data L” register (i.e. the low byte of the data address) by opening Gates J and K (Figure 21). The Address Register is incremented by Control in a manner similar to the increment of the Program Counter.

Subsequently, Gate J is opened and places the location of the high byte of the indirect address in the Address Bus Register (Figure 22). At this point the high-byte of the address of the data to load is addressed by the Address Bus Register and the low-byte is contained in Indirect Data L.

The required data address is now placed in the Address Register by opening Gate Gh (placing the data from memory into the high byte of the Address Register) and Gate L (placing the Indirect Data L content into the low byte of the Address Register). Gate Gl is not used in this part of the process (Figure 23).

With the required address loaded into the Address Register the required LDA or STA as seen in Part 1 now occurs.

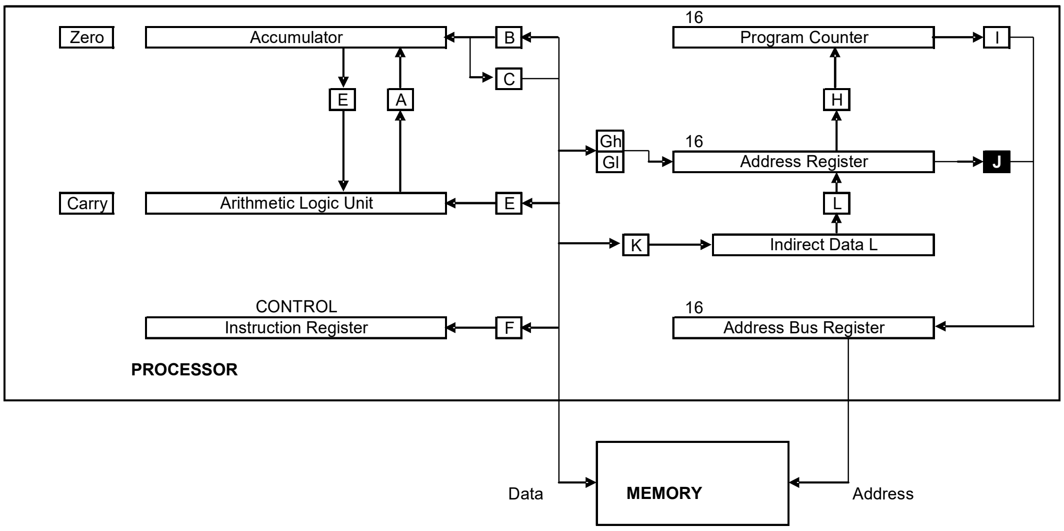


Figure 22 Placing the address of the high byte of the data address into the Address Bus Register

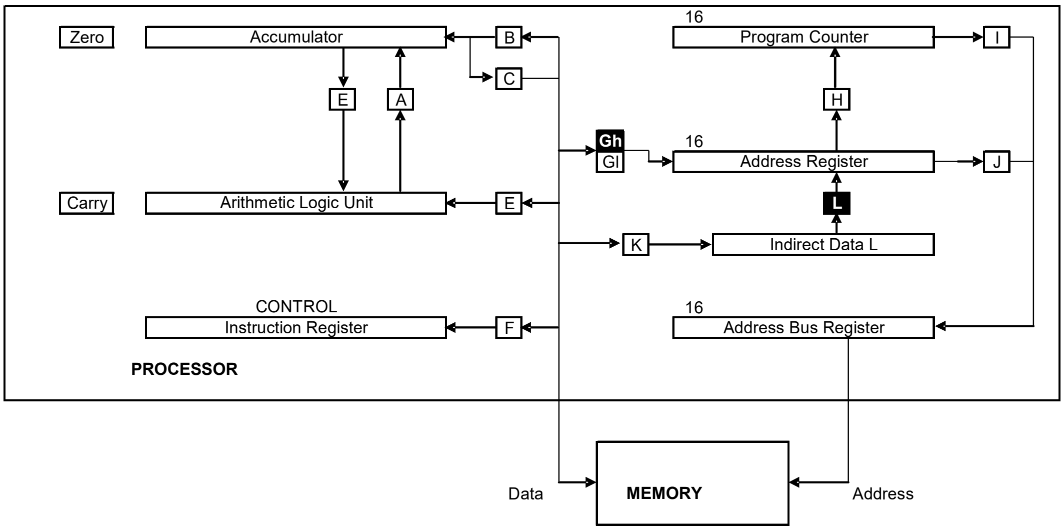


Figure 23 Placing the data address into the Address Register

4.2 Data Address Indexing

Much of what is required of processors involves moving information. A smartphone connecting to a web-site reads information from the site and displays it on the smartphone screen (a kind of write-only memory) with some formatting procedures. In each case, data making up the information may be held in consecutive bytes of memory.

Therefore, many tasks required of the processor involve consecutive bytes of memory, for example to move copy or find data. To do this efficiently it is useful for the processor to contain some means of moving along the bytes of a section of memory. One way of achieving this is via index registers which contain a counter that is added to a base address held in the Address Register. The count is incremented or decremented thus providing a movable pointer into memory.

The processor is enhanced to include index registers. The index register content is added to the starting memory address of the data to be processed. Two index registers (each eight bits) are added to the simple processor in this example.

Figure 24 shows the addition of index registers to the 16-bit address machine. Gate J is divided into two in order to control the high byte and low byte of the address separately.

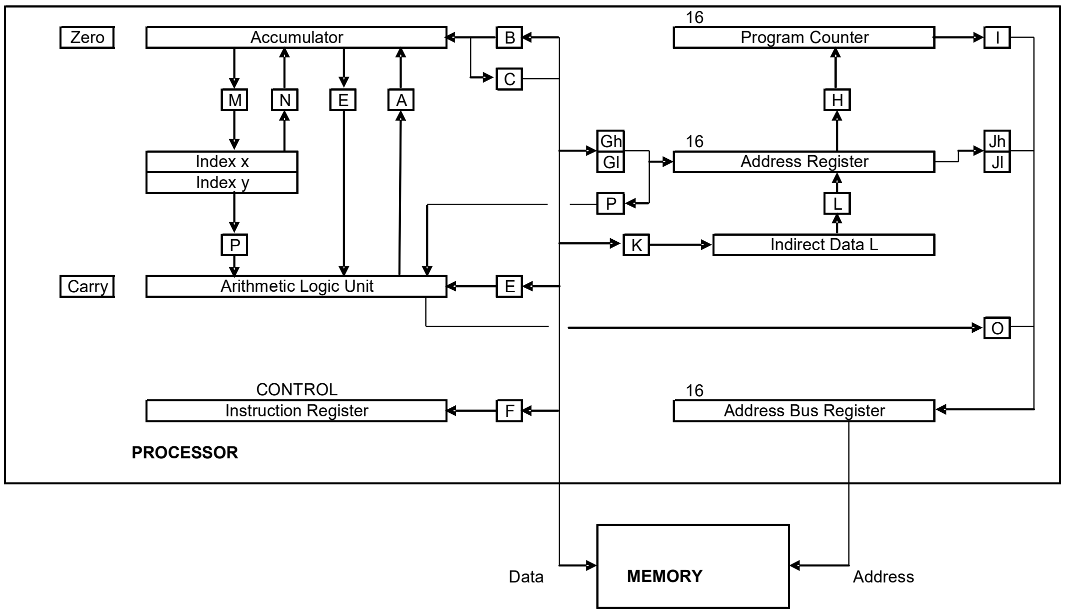


Figure 24 Index Registers added

The function of the additional Gates when open is shown in Table 22.

Jh	The high 8 bits of the Address Register are placed into the high 8 bits of the Address Bus Register.
Jl	The low 8 bits of the Address Register are placed into the low 8 bits of the Address Bus Register.
M	Data from the Accumulator is placed in the identified Index Register.
N	Data from the identified Index Register is placed in the Accumulator.
O	The output of the ALU is placed into the low 8 bits of the Address Bus Register.
P	The data in the index register referenced in the Instruction Code and low 8 bits of the Address Register are placed in the ALU set to ADD.

Table 22 Additional Gates for Indexed addressing

The index registers are eight bits since the ALU is eight bits. This limits the indexing technique to 256 bytes of memory in this design.

The eight bit registers “Index x” and “Index y” can be read and written to via the Accumulator through Gates N and M respectively. Each index may be incremented or decremented through the issue of the appropriate Instruction Code (e.g. INX or DEX for the x index). Indexed addressing requires the instruction to specify which index is to be used. For example, to load the accumulator with data using index x the Instruction Code could be

LDA,x address – where address consists of two-bytes and LDA,x represents “load accumulator from address using index x”.

4.2.1 Direct Addressing

The actual address which is placed in the Address Bus Register is calculated from the low byte of the Address Register added to the content of the specified index register. The Carry flag is not affected.

If the addition overflows in the ALU then the high byte of the Address Bus Register is incremented (requiring additional electronics). Gate P on the Address Register only passes the low 8 bits. Following the addition (and any increment), Gate Jh (not Jl) and Gate O (which only provides the low 8 bits) are opened and the indexed address passes to the Address Bus Register.

An example is shown in the figure. The index x contains 02 (hex) Instruction Code is LDA,x #FFE0

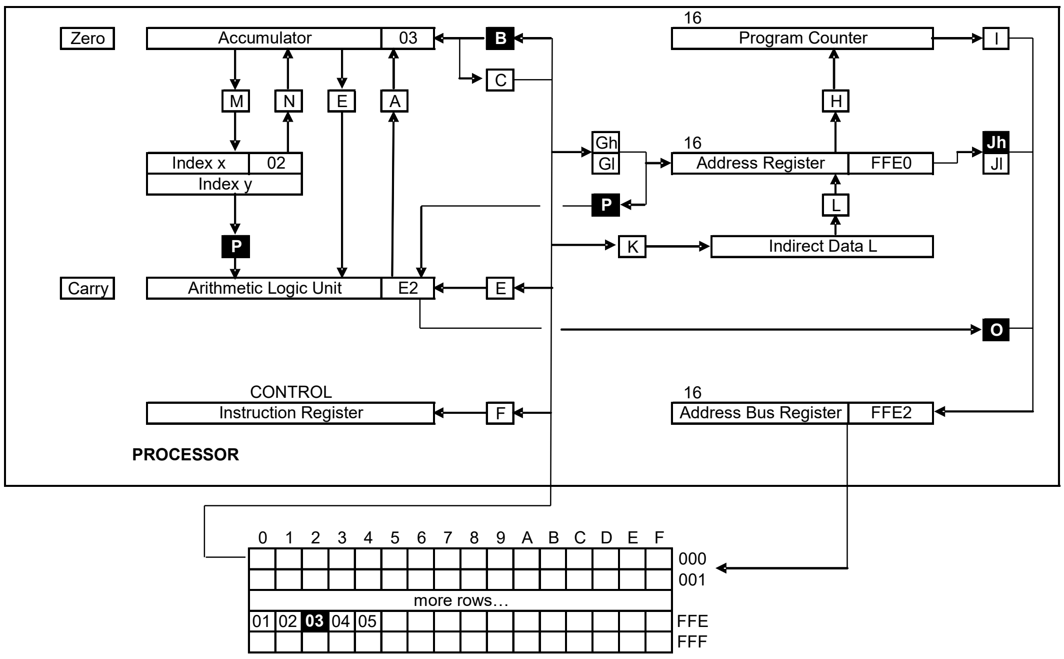


Figure 25 Final execute cycle of LDA,x address FFE0 loaded into Address Register

Possible Instruction Codes supporting indexed direct addressing are shown in Table 23.

Ins	Operand	Feature
LDA,x	Y	Load Acc index register x
LDA,y	Y	Load Acc index register y
STA,x	Y	Store Acc index register x
STA,y	Y	Store Acc index register y
INX	N	Increment index x
INY	N	Increment index y
DEX	N	Decrement index x
DEY	N	Decrement index y
ATX	N	Data in Accumulator to index x
ATY	N	Data in Accumulator to index y
XTA	N	Data in index x to Accumulator
YTA	N	Data in index y to Accumulator

Table 23 Indexed direct addressing Instruction Codes

Indexed addressing can also be applied to the ALU instructions, but the additional Instruction Codes are omitted here for brevity.

To demonstrate the effectiveness of indexed addressing a simple program is described which copies a number of bytes across memory.

For the direct addressing codes available to the simple processor (indirect addressing covered in section 4.2.2), an algorithm that fulfils the requirement of the program is shown in Table 24. Each consecutive source and destination byte is marked with consecutive numbers.

	LDA	Source1
	STA	Dest1
	LDA	Source2
	STA	Dest2
	LDA	Source3
	STA	Dest3
	…	

Table 24 Program for copying data

The LDA/STA statements are repeated until all the required data are copied. Clearly, for large amounts of data this requires many statements.

The same requirement is achieved using index addressing by the program in Table 25. This algorithm is complete and simply requires the start addresses for the source and destination memory locations and the number of bytes to be copied to be supplied in the constants “source”, “dest” and “num”.

	LDI	#00
	ATX	
loop	LDA,x	source
	STA,x	dest
	INX	
	XTA	
	XOI	num
	BNZ	loop
	HLT	

Table 25 Program for copying data using indexing

4.2.2 Indirect Addressing

Indirect addressing is used where the addressed data is itself an address to the required data. This is particularly useful for calculated addresses.

Two techniques can be applied to indexing indirect addresses:

- The instruction address loads an address which is indexed to load the data. This is represented here as LDX,x or LDX,y (and equivalent STX) and referred to as indirect indexed.
- The instruction address is indexed and the result loads the address to the data. This is represented here as LDX,(x) or LDX,(y) (and equivalent STX) and referred to as indexed indirect.

Table 26 summarises the indirect addressing instructions using indexes. All the instructions include an operand.

Ins	Feature
LDX,x	Load Acc indirect index x
LDX,y	Load Acc indirect index y
LDX,(x)	Load Acc index indirect x
LDX,(y)	Load Acc index indirect y
STX,x	Store Acc indirect index x
STX,y	Store Acc indirect index y
STX,(x)	Store Acc index indirect x
STX,(y)	Store Acc index indirect y

Table 26 Indirect addressing Instruction Codes

Each technique is illustrated in the following figures, since the indirect nature can be tricky to grasp.

Indirect Indexed

The Address Register is loaded with the address of the data as described for the LDX and STX Instructions in section 4.1.2. The final cycle of the instructions acts upon the data in the manner illustrated in Figure 26.

In Figure 26 the Instruction is LDX,x #FFE0 and the index x contains 02 (hex). The address stored at FFE0 is FFF0 (stored as low byte followed by high byte). The index is added to the address in Address Register to derive the final address to the data. Control opens Gates P, Jh, O and B and reads memory.

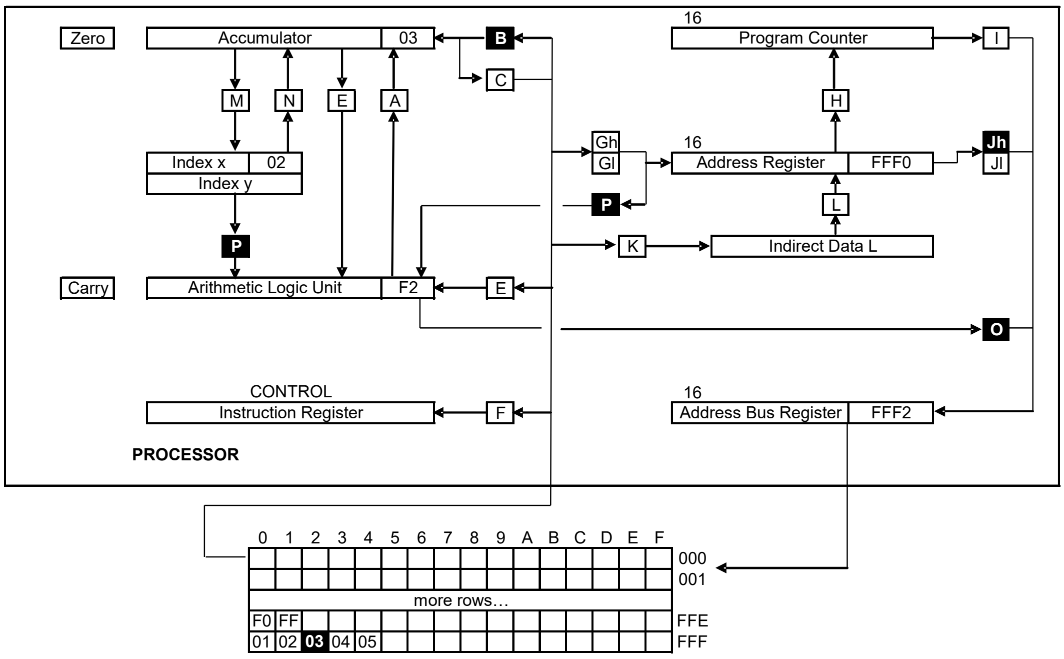


Figure 26 Final execute cycle of LDX,x #FFE0

The program in Table 27 provides a general purpose data copying routine without using index registers. The source address, destination address and number of bytes to copy are provided in set memory locations. The routine requires a workspace ("temp") where the required end-point is calculated for reference. The same routine can be achieved using an index and is shown in Table 28.

It can be seen that the indexing technique provides a more efficient and higher performance program.

The routines described have a structure which lends itself to the idea of subroutines described in section 4.3.

	LDA	dest
	CLC	
	ADC	num
	STA	temp
loop	LDX	source
	STX	dest
	LDA	source
	CLC	
	ADI	#01
	STA	source
	LDA	dest
	ADI	#01
	STA	dest
	XOR	temp
	BNZ	loop
	HLT	
source	&	
dest	&	
num	&	
temp	&	

Table 27 General copy routine without indexing

	LDI	#00
	ATX	
loop	LDX,x	source
	STX,x	dest
	INX	
	XTA	
	XOR	num
	BNZ	loop
	HLT	
source	&	
dest	&	
num	&	

Table 28 General copy routine with indexing

Indexed Indirect

The instruction address is indexed to find the data address. Following the load of the instruction address (i.e. the operand) the execution cycles continue as follows.

In Figure 27 the Instruction is LDX,(x) #FFE0 and the index x contains 02 (hex). Control opens Gates P, Jh, O and K and reads memory.

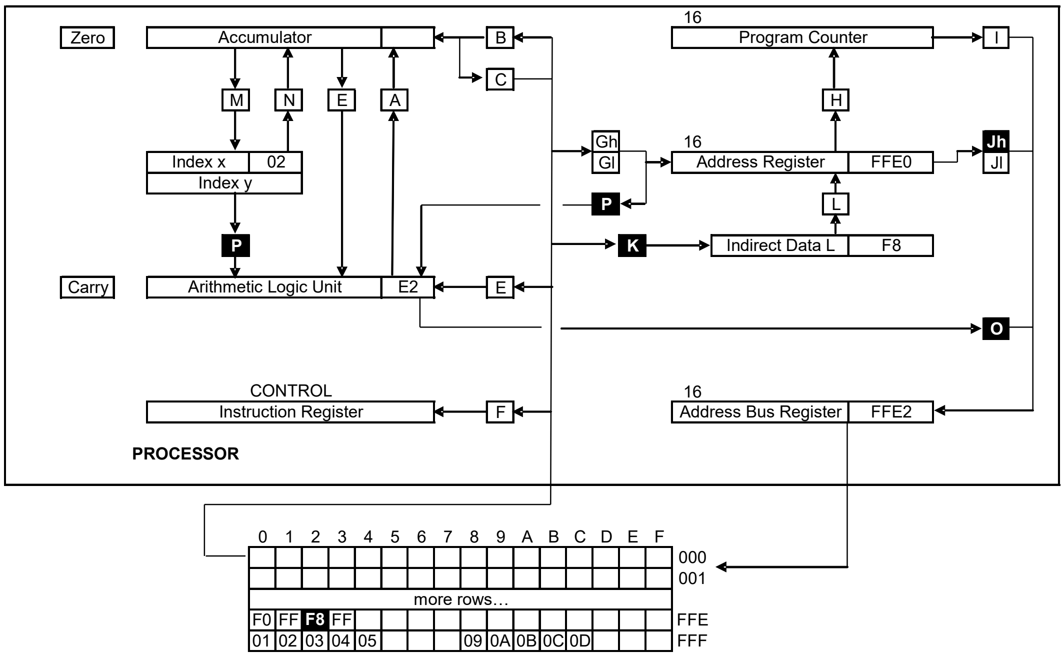


Figure 27 Execute cycle for LDX,(x) following load of operand FFE0

The Address Register value is indexed to find the location of the data address. The resulting address is FFE2. The content forms low byte of the data address and is loaded into Indirect Data L (Figure 27).

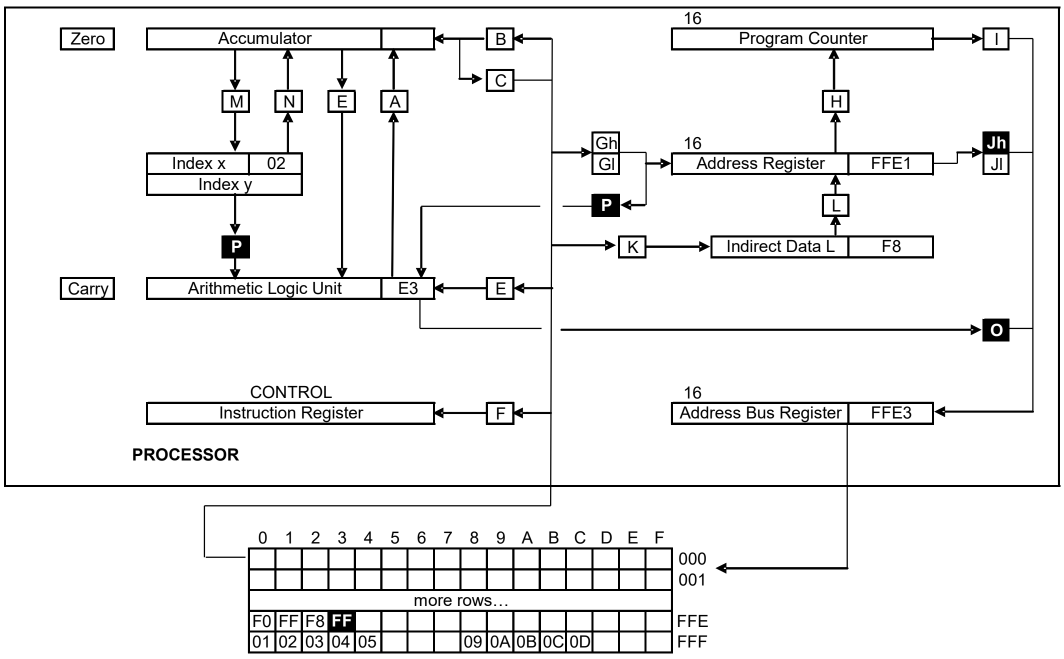


Figure 28 Setting the high byte address of the high byte data address

In the next cycle (Figure 28) the Address Register is incremented so that when Control opens Gates P, Jh and O the location of the high byte of the data address is placed in the Address Bus Register. The following execute cycle Figure 29 loads the address of the data into the Address register by opening Gates Gh and L.

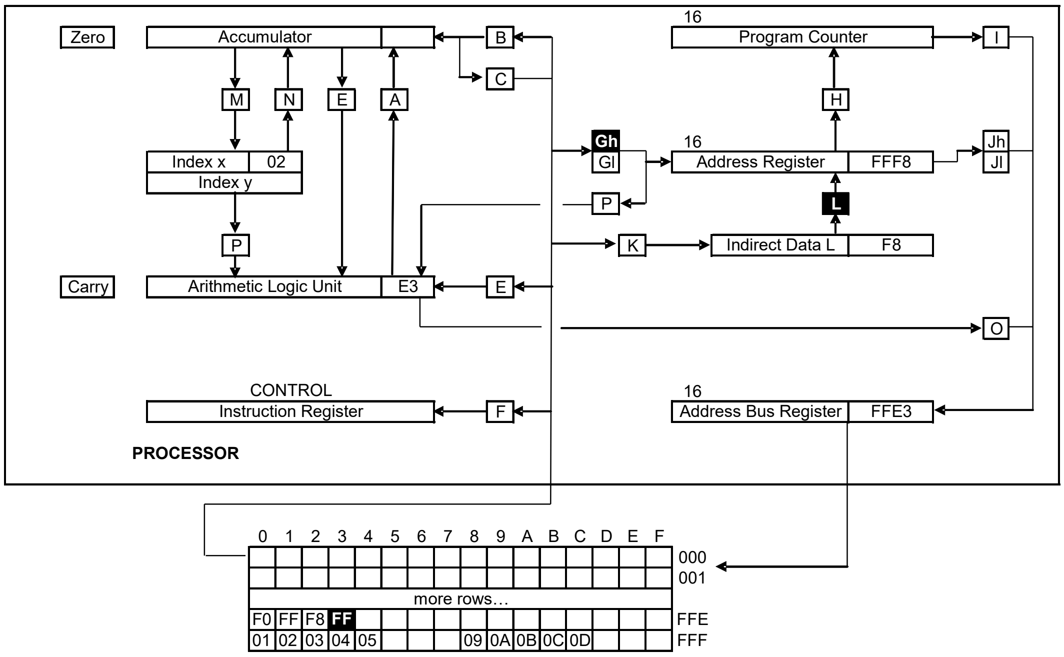


Figure 29 Setting the data address into the Address Register

The final execute cycle loads the data addressed in the Address Register. Control opens Gates Jh, Jl and B and reads memory (Figure 30). A similar process stores the Accumulator in memory for STX,(x).

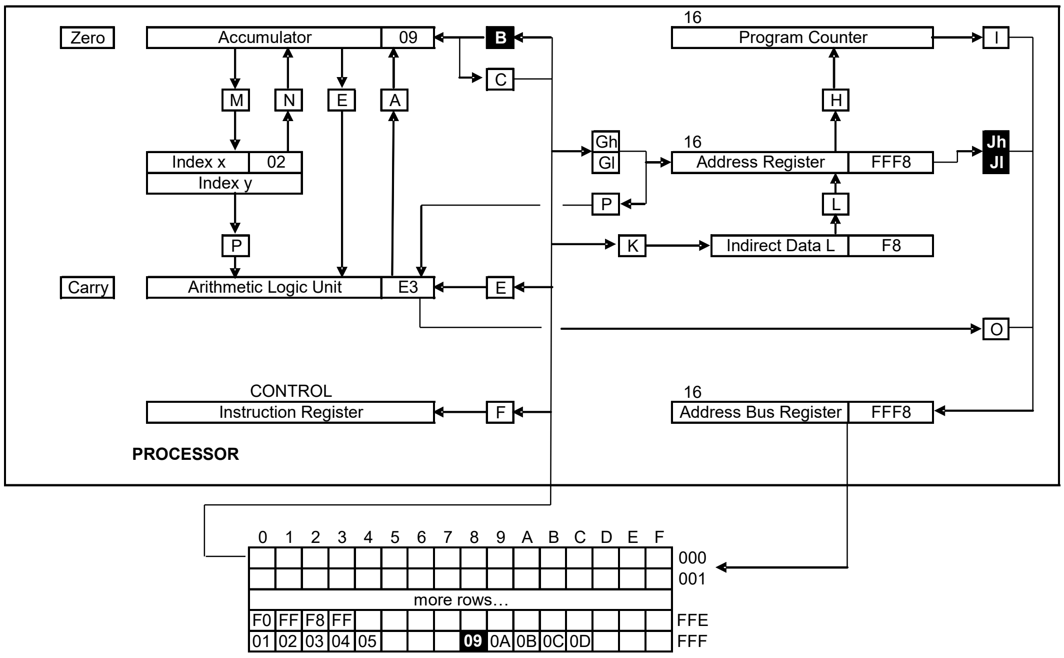


Figure 30 Final execute cycle of LDX,(x) #FFE0

The major use of indexed indirect addressing is for finding a particular data item from a list or table of item addresses.

4.3 Subroutines

Programs often need to perform certain tasks repeatedly. For example, move data, copy data, find, delete etc. The task may operate with different data but will essentially consist of the same set of instructions. Clearly, this could account for a large amount of the Instruction Code in a program. A more code-efficient approach is for the program to contain one copy of the task and “call” it from other parts of the program that require the task, passing the data pertinent to the calling program. This is the subroutine.

Programs usually consist of a main routine (often a loop) and a number of subroutines. Parameters are passed to the subroutine to qualify the actual requirement where necessary. The essential character of the subroutine is that its code is somewhere else and not inline with the current program code flow. Therefore, when a subroutine is called the program jumps to the location of the subroutine Instructions Codes, executes the codes and on completion jumps back to the Instruction Code following the call. This is achieved by using an additional register “Stack Pointer” (Figure 31) and instructions as described in the following.

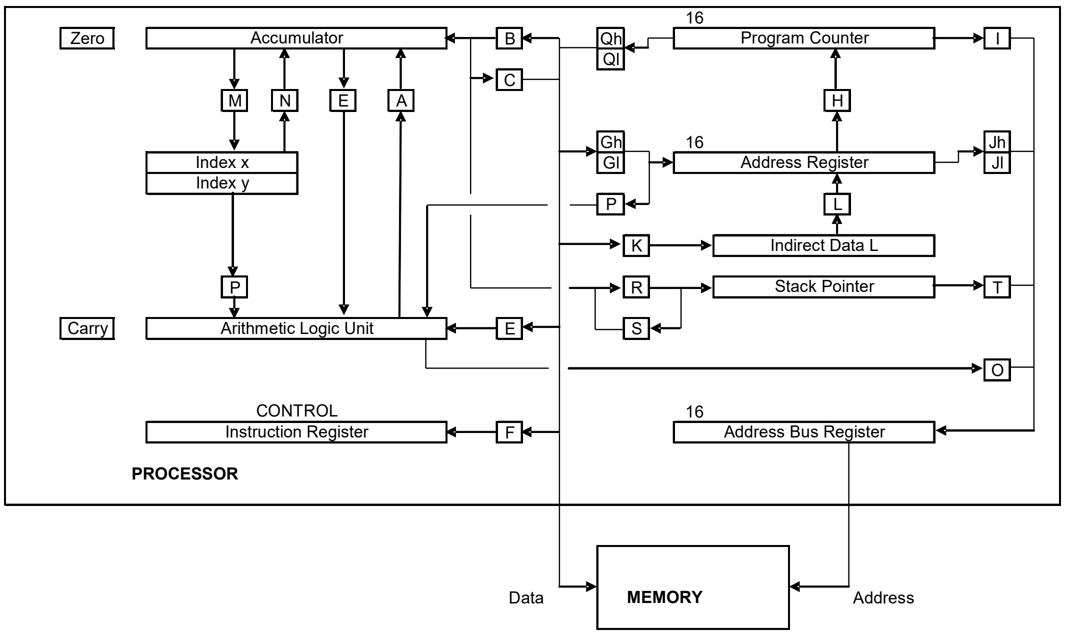


Figure 31 Stack Pointer and saving Program Counter added

The initial location of the Stack Pointer is set via the Accumulator through Gate R and can be monitored through Gate S.

The function of the additional Gates when open is as shown in Table 29.

Qh	Places the high 8 bits of the Program Counter on the data bus.
Ql	Places the low 8 bits of the Program Counter on the data bus.
R	The Accumulator is placed in the Stack Pointer.
S	The Stack Pointer is placed in the Accumulator.
T	The Stack Pointer is placed into the low 8 bits of the Address Bus Register.

Table 29 Additional Gates for Subroutines

The Stack Pointer is an 8-bit register providing the low byte of a 16-bit address for the location of the “Stack”. The high-byte of the stack could be provided by an extended register. In this example the stack is fixed at address FFxx for simplicity. (Some real processors do something similar). That is, when Gate T is opened the high 8 bits to the Address Bus Register are set to FF (hex).

Stack memory operates by setting the pointer to an address, storing register data in the addressed memory byte and subsequently decrementing the stack pointer to the next address byte below. This is called a data “push” onto the stack. A data “pull” from the stack operates in the opposite way. The Stack Pointer is incremented and the data at the stack address is loaded into the identified register. The register here is the Program Counter (which requires two stack operations to push the 16 bits).

A subroutine is called using a branch instruction known as “Jump Subroutine”. The operand of the instruction contains the address at which the subroutine Machine Code is present. This is loaded into the Address Register. The Program Counter is incremented to point to the next instruction. Before making the jump (i.e. opening Gate H), the current value in the PC (i.e. the address of the next instruction) is saved on the stack. This involves two push operations to store the low byte and high byte of the PC through Gates Ql and Qh respectively. Opening the Q Gates sets the memory for a data write.

The six-cycle sequence for Jump Subroutine is as follows:

Fetch	Execute 1	Execute 2	Execute 3	Execute 4	Execute 5
I, F inc PC	I, Gl inc PC	I, Gh inc PC	T, Ql dec stack	T, Qh dec stack	H

After the subroutine has completed its task the program flow returns to the point after the call. This is achieved using the instruction “Return from Subroutine”, which has no operand. The instruction restores the PC from the stack and branches.

The four-cycle sequence for Return from Subroutine is as follows:

Fetch	Execute 1	Execute 2	Execute 3
I, F	inc stack T, Gh	inc stack T, Gl	H

The fetch cycle may or may not increment the Program Counter (may not means HLT is less unique) since there is no operand and the Program Counter is over-written anyway.

Example instructions supporting subroutine processing are shown in Table 30.

Ins	**Operand**	**Feature**
JSR	Y	Jump to subroutine
RTS	N	Return from subroutine
TAS	N	Data in Accumulator to Stack Pointer
TSA	N	Data in stack Pointer to Accumulator

Table 30 Subroutine Instruction Codes

4.4 Interrupts

A product comprising a processor very often interacts with another device asynchronously. That is, the device signals the processor system of some event that needs some attentive processing but there is very little prior notice of when the processing is required. The required processing may be time-critical.

It may be possible sometimes for the processor to wait for the event by polling the device. For example in a processing loop by waiting for the event to occur, be detected and branch out of the loop. However, this is a waste of processing potential and often is not practical, especially if the event occurring is infrequent compared to processing time.

Instead, the processor is enhanced to provide “interrupt” processing. An external event is signalled in hardware to the processor which can respond very quickly to the event and begin the necessary processing. The following shows a typical example of how this can be achieved.

The processor is further enhanced in Figure 32 by providing a hardware input where the need for attention by an event is signalled. The signal is monitored and initially processed in hardware by the Interrupt Request controller. This also determines if an interrupt is allowed to be processed immediately. This feature is required because sometimes the processor may be completing a critical task so is unable to go to interrupt processing immediately. The interrupt is held-over until the processor completes the critical task whereupon the interrupt can occur. Instructions are provided to enable and disable interrupt processing.

An additional register is required: “Interrupt Vector” which holds the indirect address of the interrupt program.

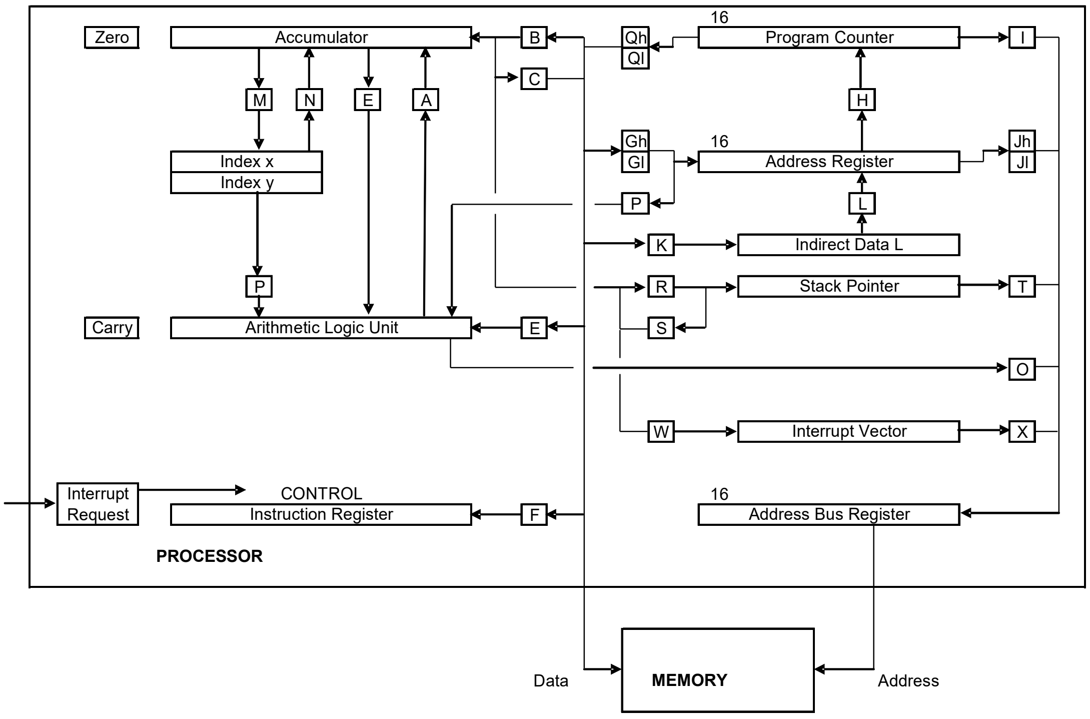


Figure 32 Interrupt Vector added

The additional Gates when open perform the functions shown in Table 31.

W	Data from the Accumulator is placed in the Interrupt Vector.
X	The Interrupt Vector is placed into the low 8 bits of the Address Bus Register.

Table 31 Additional Gates for Interrupt processing

The Interrupt Vector register contains the low-byte address of memory holding the address of the “interrupt routine”, which is the program to which processing begins when the processor responds to the interrupt event. In this machine the vector is set-up through the Accumulator. Some practical processors pick-up the vector from fixed memory space.

When an interrupt is accepted the processor completes execution of the current instruction and performs interrupt processing. An interrupt automatically disables further interrupts from occurring until the program explicitly re-enables them. The vector for the interrupt routine is contained at address FFxx, where xx is set by the Interrupt Vector. The memory byte pointed to by the vector is loaded into the Address Register low byte. The Interrupt Vector is incremented and the next byte loaded into the Address Register high byte. The Interrupt Vector is decremented to restore its value for the next interrupt.

Interrupt processing breaks the current program thread and jumps to the location given by the interrupt vector. When the interrupt processing is completed, the program returns to continue the thread. To do this, the interrupt processing saves the PC to the stack. The return from the interrupt can occur by restoring the PC from the stack.

However, this is not sufficient. Because the interrupt can occur anywhere in the process flow (for example between a LDA and STA) it is essential to preserve any information in registers (or memory if applicable) that could be impacted in subsequent instructions. For the simple processor this means any of the Accumulator, flags (Zero and Carry) and either or both the index registers must be saved and restored along with the PC.

The hardware could be arranged to save and restore all the required states in the processor. However, the implementation here keeps hardware to a minimum and places requirements upon the interrupt program. Furthermore, it is most efficient to save only the registers that are used by the interrupt program.

The hardware saves and restores the PC. However, the Accumulator (with Zero flag), Carry flag and index registers (if used) are saved and restored as the first and last activities of the interrupt program. This can be achieved through the instructions “Save Accumulator to Stack” (STS) and “Load Accumulator from Stack” (LDS). The instructions have no operand. Other items are saved and restored through the Accumulator. The Carry flag is tested with a conditional branch and a suitable marker placed on the stack to determine if the flag is set or reset before the interrupt returns.

The cycles for the instructions are as follows.

STS:

Fetch	Execute 1
I, F inc PC	T, C dec stack

LDS:

Fetch	Execute 1
I, F inc PC	inc stack T, B

The cycles processing the interrupt are special in that they are effectively a set of cycles between instructions in the program. This is illustrated in Figure 33.

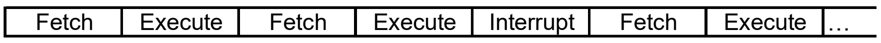


Figure 33 Additional cycles inserted by interrupt processing

The additional interrupt cycles are as follows

Interrupt 1	Interrupt 2	Interrupt 3	Interrupt 4	Interrupt 5
X, Gl inc IV	X, Gh dec IV	T, Ql dec Stack	T, Qh dec Stack	H

When the interrupt program is complete it signals a return to processing where the interrupt occurred. This is accomplished with a “Return-from-Interrupt” instruction (e.g. Instruction Code RTI), which has no operand. The sequence for this is as follows.

Fetch	Execute 1	Execute 2	Execute 3
I, F	inc stack T, Gh	inc stack T, Gl	H

As with the return from subroutine instruction, the fetch cycle may or may not increment the Program Counter (may not means HLT is less unique) since there is no operand and the Program Counter is over-written anyway.

The instructions required to support interrupt processing include a means to enable and disable interrupt processing (so that critical tasks can be protected against interruption). Examples are shown in Table 32.

Ins	**Operand**	**Feature**
ENI	N	Enable interrupt processing
DIS	N	Disable interrupt processing
RTI	N	Return from Interrupt processing
STS	N	Store Accumulator to Stack
LDS	N	Load Accumulator from Stack

Table 32 Interrupt Instruction Codes

On power-up and hardware reset the processor would start processing with interrupts disabled.

4.5 Input/Output Devices

The processor described in these books performs basic computations and memory manipulation. A real processor system also interacts with input/output devices in order to communicate with the outside world. Examples of devices include communication ports, real-time clocks, data storage devices, keyboards and displays.

Interaction between the processor and a device requires the device to be identified, the direction of information flow established and data transferred. The direction of flow may be intrinsic (e.g. a keyboard would always be input data and a display output data). The read/write function of the processor is used to determine the direction of data. Data is transferred on the Data Bus.

The processor needs to access particular devices. Therefore, a means to identify devices is required.

The simple processor could be enhanced to include specific i/o ports (in the form of registers) which would provide direct control lines to decoding electronics. The decoders would enable the specified device or devices.

A simpler approach requiring no hardware enhancement is to map the devices into memory. Some commercial processors have taken this approach. The electronic decoders for memory could divide the available memory space between memory and devices. An example based upon the enhanced processor here which can map up to 16 devices is shown in Figure 34.

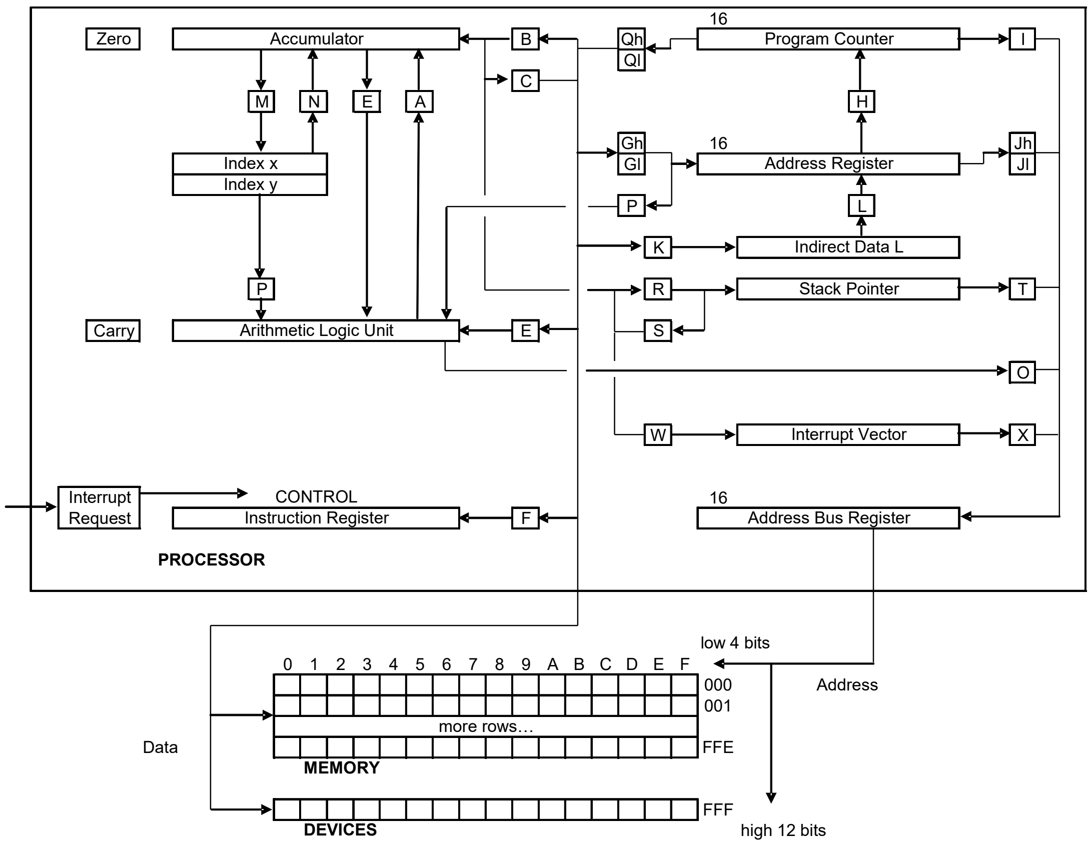


Figure 34 Memory-mapped input/output devices

This approach requires the location of the Stack Pointer and Interrupt Vector to be more flexible than previously described.

Some commercial processors have defined device instructions and signal the instruction type (memory or i/o) during the machine cycles. By specifically signalling device and memory operations the memory space is preserved for programs and data.

There are many other ways input/output devices could be decoded according to the imagination of the designer.

5 Concluding Remarks

Book 1 Part 1 introduces a simple processor and instruction set. The expansion in this part resembles processors developed in the past. Although some small systems may require such low-level design there are fewer practical applications today.

Simple processors developed in the 1970s followed a development path which added larger data and address capability along with additional features (e.g. direct memory manipulation also known as “read-modify-write”, device input/output, hardware multiplier). Many of the operations rely upon direct interaction of the processor with its memory, as demonstrated by the simple processor in this Book.

In the 1980s a complementary development path explored an alternative approach. There was recognition that processor chip real-estate is limited and there is a trade-off between hardware enhancements and performing the most commonly used instructions as quickly as possible. Two approaches to more complex functionality are

- Processor hardware can be further developed to provide additional functionality directly.
- Retain a simple instruction set, greatly enhance processor performance and develop complex functionality through the use of software tools to generate the required additional volume of instruction codes.

Processors were developed to handle the most commonly used instructions as quickly as possible and complex functionality was left to the development tools i.e. high-level language software and compilers. Processors of this type are known as “Reduced Instruction Set Computers” or RISC machines. Machines with enhanced hardware features have become known as “Complex Instruction Set Computers” or CISC machines.

A faster processor may be limited in its performance by the time taken to interact with its memory. This can easily be seen today where processor core clocking has exceeded 1GHz. The clock cycle time at this rate is one nano-second (10^{-9} seconds). Light travels only 30cm in this time, so the electrical operation of remote memory at this rate is exceedingly difficult to impossible.

Therefore, processors today make use of many more registers within the processor microchip (where distance is short) and interaction with memory is minimised by architecture and “sub-contracting” to a memory manager. Processing occurs between registers. The memory manager stores results (as required) and fetches instructions and data in anticipation of the processor’s needs (“pipelining”). This also means there is some overlap of the fetch and execute cycles unlike the clear separation previously described.

Modern commercial processors are very high speed and very inexpensive. Advances in software engineering have evolved parallel-processing techniques so that some program code threads can be run simultaneously in multiple-CPU machines. The human interface is so abstract that no knowledge of what the processor is actually doing is required (or indeed possible). The development of high-level programming environments has allowed the specialisation and simplification of software application creation.

Nevertheless, ultimately the processor machine is manipulating data through Gates and registers in the manner seen in these Books.

Appendix: Boolean Algebra

Boolean algebra is a technique for writing and calculating logical functions. The input and output quantities are given letter names (like ordinary algebra) such as x, y, z etc. The letters represent the general state of the quantities but each can have only two possible values: zero or one.

Therefore, it can be stated that, for a quantity x subjected to a logical OR with its own invert:

$x + \overline{x} = 1$

Similarly for x subjected to a logical AND with its invert

$x . \overline{x} = 0$

Tables can be constructed showing outputs for particular inputs. For example, take two inputs x and y which generate the output z according to Table 33:

x	y	z	$\overline{x}$	$\overline{y}$	$\overline{z}$
0	0	0	1	1	1
0	1	1	1	0	0
1	0	1	0	1	0
1	1	1	0	0	0

Table 33 Truth table for output z with inputs x and y

The left side of the table shows the output z for two inputs x and y. The function is clearly z equals x OR y which is expressed as

$z = x + y$

The right side shows all the values in the left but each is inverted. The output zbar is clearly xbar AND ybar expressed as

$\overline{z} = \overline{x} . \overline{y}$

The AND to the right is the invert of the OR to the left. The AND expression above (with zbar) is the invert of the OR above (with z). Therefore, the following expression can be written

$\overline{x} . \overline{y} = \overline{x + y}$ (rule 1)

Similarly, by starting with an AND table on the left of Table 33, it can be shown that

$\overline{x . y} = \overline{x} + \overline{y}$ (rule 2)

The two rules are called D'Morgan's rules and are used extensively when designing logic circuits.

For example, a design for a full-adder using only NAND gates needs to include the Exclusive OR function, for which the output required is shown in Table 34.

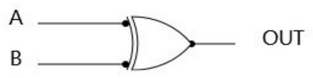
Type	**Inputs** A	B	**Output** OUT	**Notation**	**Symbol**
Exclusive OR	0	0	0	A ⊕ B	
	0	1	1		
	1	0	1		
	1	1	0		

Table 34 Exclusive OR function

A NAND function is an inverted AND of the form

$$z = \overline{x \,.\, y}$$

The Boolean expression for Exclusive OR is

$$z = \bar{x} \,.\, y + x \,.\, \bar{y}$$

Using D'Morgan this can be rearranged as

$$z = \overline{\overline{\bar{x} \,.\, y} \,.\, \overline{x \,.\, \bar{y}}}$$

This shows two levels of NAND function. The two expressions involving x and y are NANDs and their result leads to a further NAND. The circuit may be drawn as shown in Figure 35. Partial results are shown.

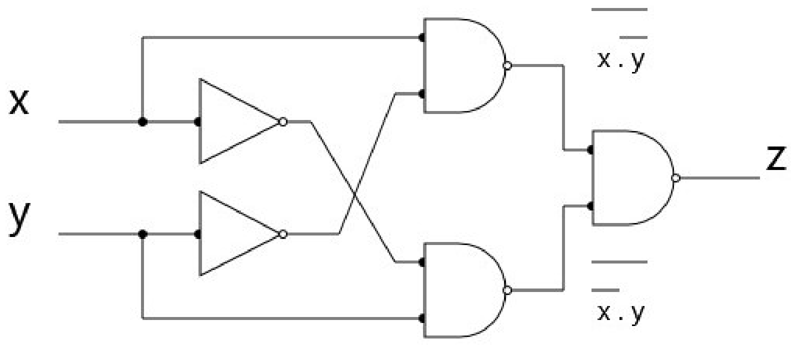


Figure 35 Exclusive OR

However, to use only NAND gates the x and y expressions need to be formulated using the non-inverted forms. That is, the NAND needs to be between x and y and the NOT gates removed.

The forms to be replaced are

$\overline{x} \, . \, y$ and $x \, . \, \overline{y}$

To do this, note the following trick using D'Morgan's rules and simple algebra

$$\overline{x \, . \, y} \, . \, y =$$

$$(\overline{x} + \overline{y}) \, . \, y =$$

$$\overline{x} \, . \, y + \overline{y} \, . \, y$$

But ybar AND y is zero (as seen above) and the second part of the final expression is zero. Therefore the mixed forms in the Exclusive OR function can be replaced as follows

$$z = \overline{\overline{\overline{x \, . \, y} \, . \, y} \, . \, \overline{x \, . \, \overline{x \, . \, y}}}$$

This function shows three levels of NAND combining the inputs x and y. The circuit can immediately be drawn as shown in Figure 36. The partial results for each NAND gate are shown.

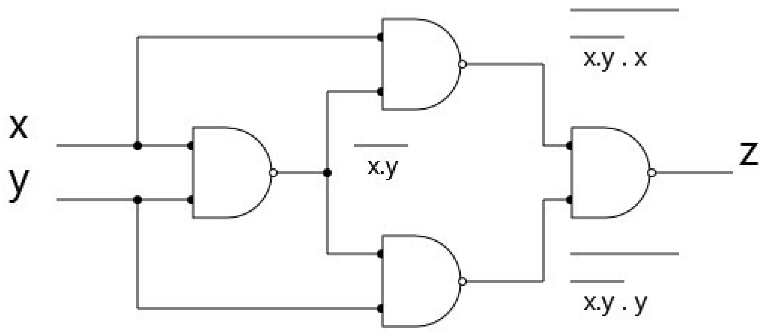


Figure 36 Exclusive OR in NAND gates

A half-adder can be achieved by inverting the partial result as shown in the circuit Figure 37.

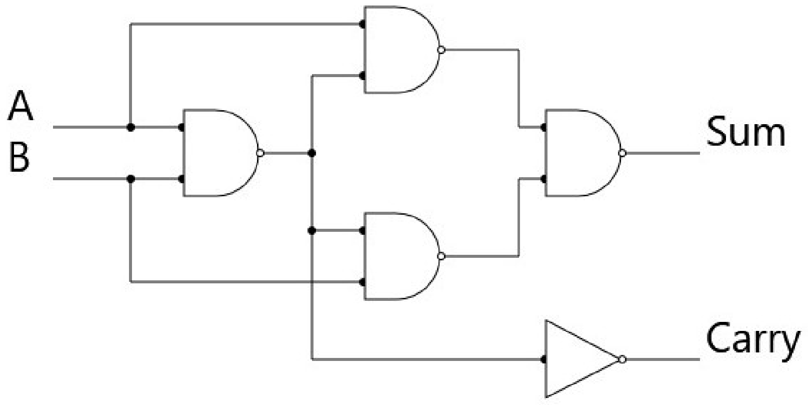


Figure 37 Half Adder in NAND gates

Although the half-adder includes a NOT gate the full-adder requires the OR of the two half-adders used in its design (section 2.2.1). Two inverted forms into a NAND achieve a logical OR. Therefore, a full-adder in NAND gates can be drawn as shown in Figure 38.

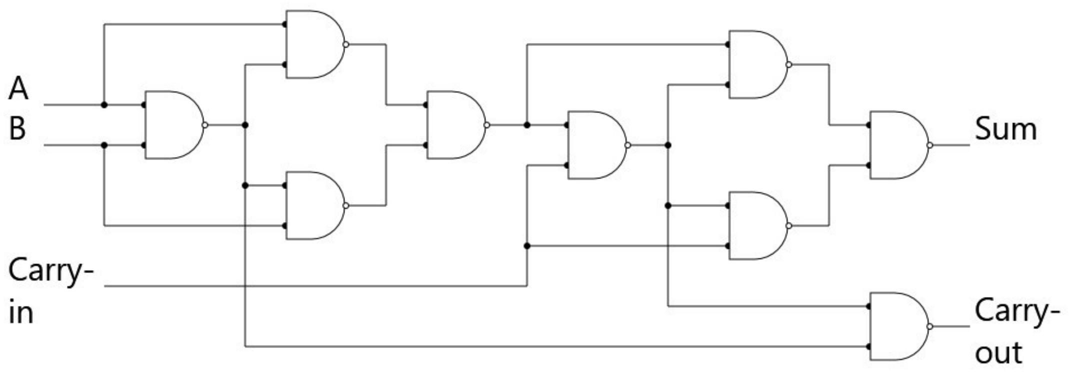


Figure 38 Full Adder in NAND gates

There are many other techniques to achieve logic gate designs.