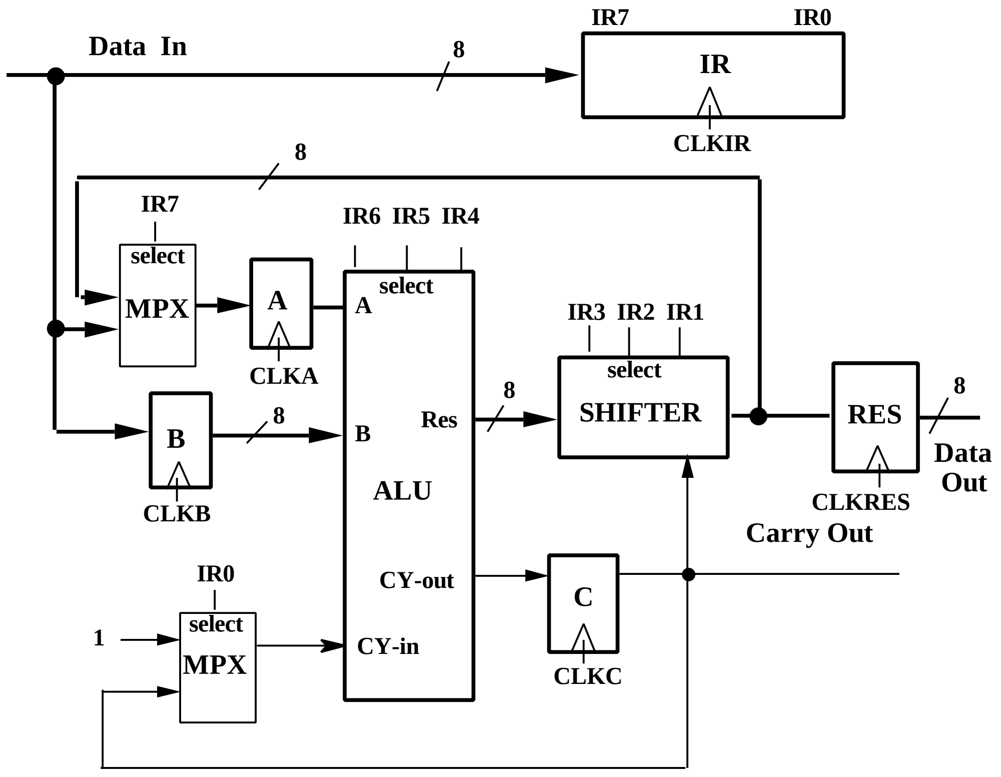
Department of Computing
Course DOC 112 -- Hardware

Lecture 15: Let's Put it All Together! -- A Manual Processor (part 2)

In the last figure of the last lecture a detailed diagram of our manual processor was shown but it only indicated that function and multiplexer selection lines were needed, it did not show where they came from. These lines determine the operations the processor executes and they are hard wired to the **Instruction Regiter (IR)** outputs. In order not to clutter up the diagram, we will indicate the **IR** outputs as **IR0** (least significant bit) to **IR7** (most significant bit). We also add names to the clocks which operate the registers and now the circuit diagram becomes more complete:



We show the assignment of the **IR** register bits below:

IR7	**IR6**	**IR5**	**IR4**	**IR3**	**IR2**	**IR1**	**IR0**

These bits are assigned in groups: **SELA**, which is one bit, selects the type of input into the **A** register; **SELALU** (3 bits) which selects the **ALU** function; **SELSHFT** (3 bits) selects the **SHIFTER** function, and, finally, **SELCY** (1 bit) which selects the **Carry-in** to the **ALU**. We show schematically these group assignments below:

IR7	IR6 ... IR4	IR3 ... IR1	IR0
SELA	SELALU	SELSHFT	SELCY

An explanation is needed why extra multiplexers were connected between the **ALU** and the input lines such that the ouput of the shifter may be loaded into the **A** register instead of the data on the input lines. This hardware arrangement used to be standard practice for very old and very small computers when the **ALU** was refered to as the **Accumulator**. This trick will allow the summation of a list of numbers. At first, two numbers are loaded into the **A** and **B** registers, respectively, the **plus** and **unchanged** functions are selected, which means that the sum **A plus B** appears at the output of the shifter. When data are clocked into the **A** register, the upper path is chosen (by setting **IR7** to **0**). During the next cycle the result at the output of the shifter is loaded into the **A** register (which is **A+B**) and then a new number is loaded into the **B** register. Selecting the plus operation again, will give the result of the sum of three numbers ... and so forth.

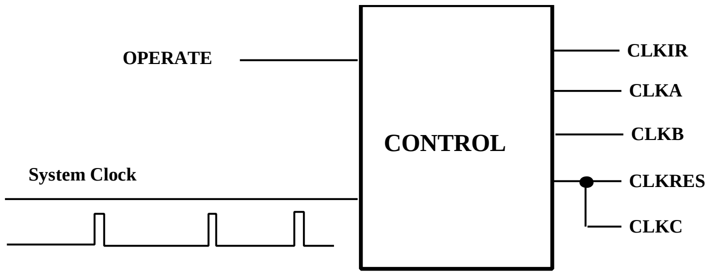
This seems to work; however, a modification is needed to the program execution steps shown in the last lecture. Before the input data is loaded into the **A** register, an operation code must be loaded into the **IR** register which then will control the computer's operation during the loading process. Also, we should not forget about the role of the **C** flip-flop. It must hold value **0** by the time the arithmetic operation is executed since only this value will provide us with the correct **A plus B** function. Thus, the **C** flip-flop must be also loaded during the first two loading operations. The modified operating sequence is then:

1. Load the bits on the "Data In" lines into the IR register.	The first op-code
2. Load the A register	From **Data In** or **SHIFTER**
3. Load the B and the C registers	**C** is set to zero
4. Load the "Data In" lines into the IR register	The second op-code
5. Load the RES and the C registers	The results

.... Go back to step 1 ....

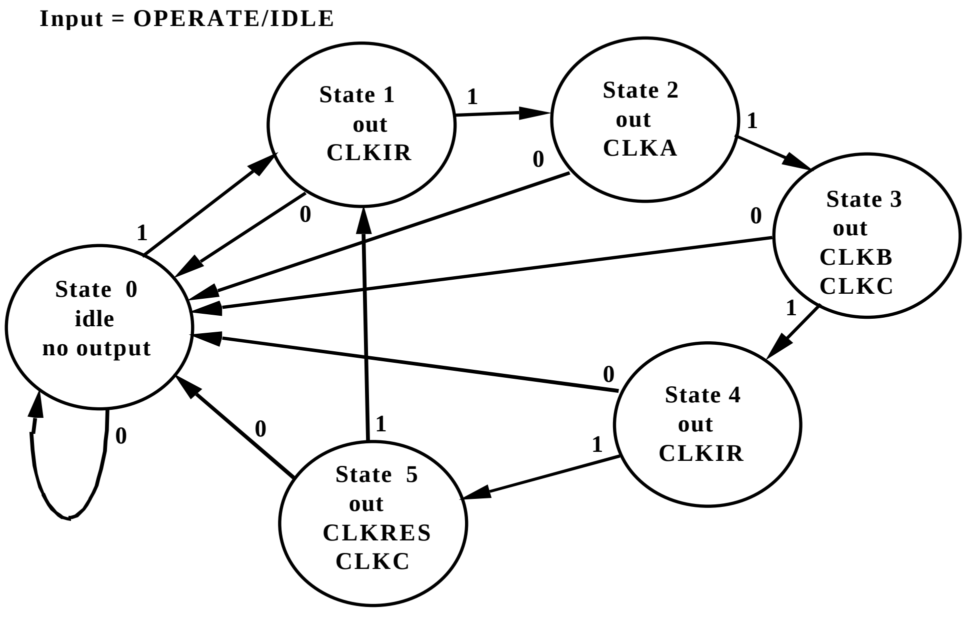
All these required operations can be expressed as the apearance of clock signals. Thus we must design now a sequential circuit which produces these clock signals at the appropriate time within the time sequence. However, something is still dodgy.

How do we know that the sequence is at **Step 1** when we want to start the computer? We must have an outside line which synchronises the operation of this processor. We will call this the **OPERATE** (its value is **1**) or **IDLE** (its value is **0**). We also must have an **IDLE** state in which the processor is sitting while the **OPERATE/IDLE** line is at value **0**. When this line is set to **1** the first clock pulse will start the processor properly. Finally we have the proper form of the control circuit:



.

Thus we have six states (three flip-flops) and the following transition diagram:

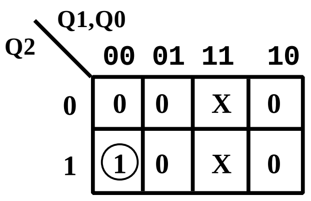


We have now a standard sequential circuit design with three flip-flops, six useful states and five outputs. Instead of starting with state assignment for the state transition diagram, I start with the output ciruits, trying to make them as simple as possible. (This seemed to work well in Lecture 10).

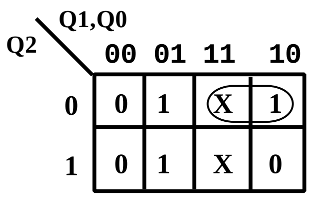
Flip-Flop Outputs	State	Required Clock Output
000	0	none
001	1	CLKIR
100	2	CLKA
010	3	CLKC , CLKB
101	4	CLKIR
110	5	CLKC, CLKRES

. .

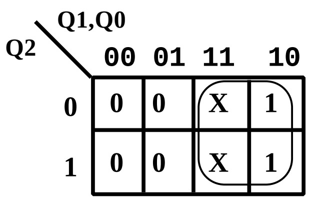
The K-maps and minimized Boolean expressions for the output clocks are shown next.



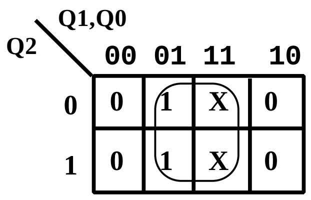
CLKA = Q2•Q1'•Q0'



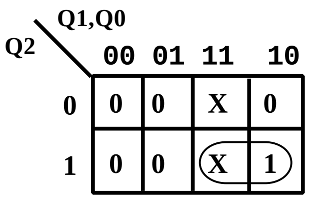
CLKB = Q0



CLKC = Q1



CLKIR = Q0



CLKRES = Q2•Q1

. .

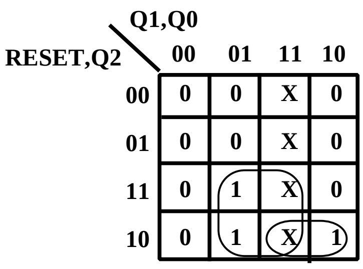
Not bad ....

We can now design the controlled counter by constructing its state transition table:

OPERATE/IDLE	Current State	Flip-flops	Next State	D2	D1	D0
0	0	000	0	0	0	0
0	1	001	0	0	0	0
0	2	100	0	0	0	0
0	3	010	0	0	0	0
0	4	101	0	0	0	0
0	5	110	0	0	0	0
0	6	011	?	X	X	X
0	7	111	?	X	X	X
1	0	000	1	0	0	1
1	1	001	2	1	0	0
1	2	100	3	0	1	0
1	3	010	4	1	0	1
1	4	101	5	1	1	0
1	5	110	1	0	0	1
1	6	011	?	X	X	X
1	7	111	?	X	X	X

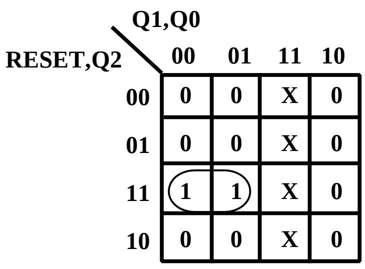
.

And the Karnaugh Maps and the minimised Boolean circuit equations are shown below:



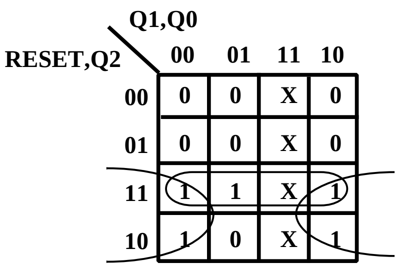
RESET,Q2 \ Q1,Q0	00	01	11	10
00	0	0	X	0
01	0	0	X	0
11	0	1	X	0
10	0	1	X	1

D2 = RESET•(Q0 + Q2'•Q1)



RESET,Q2 \ Q1,Q0	00	01	11	10
00	0	0	X	0
01	0	0	X	0
11	1	1	X	0
10	0	0	X	0

D1 = RESET•Q2•Q1'



RESET,Q2 \ Q1,Q0	00	01	11	10
00	0	0	X	0
01	0	0	X	0
11	1	1	X	1
10	1	0	X	1

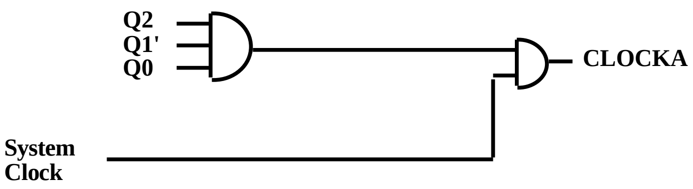
D1 = RESET•(Q2 + Q0')

Checking the unused states, we have:

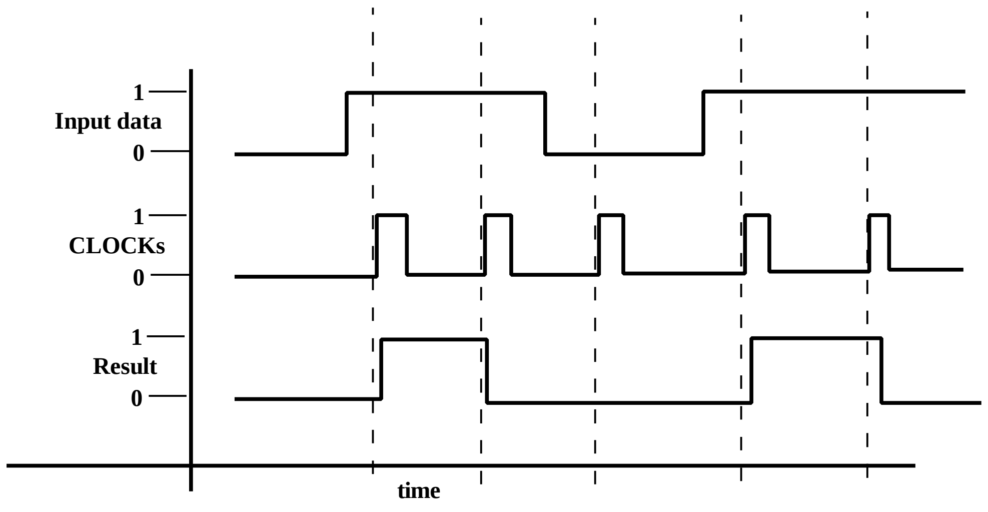
OPERATE/IDLE	Current State	Flip-flops	Next State	D2	D1	D0
0	6	011	0	0	0	0
0	7	111	0	0	0	0
1	6	011	2	1	0	0
1	7	111	4	1	0	1

Thus, if the **OPERATE/IDLE** signal is at logical **0** the system drops into the **IDLE** state immediately and the processor is ready to start working properly.

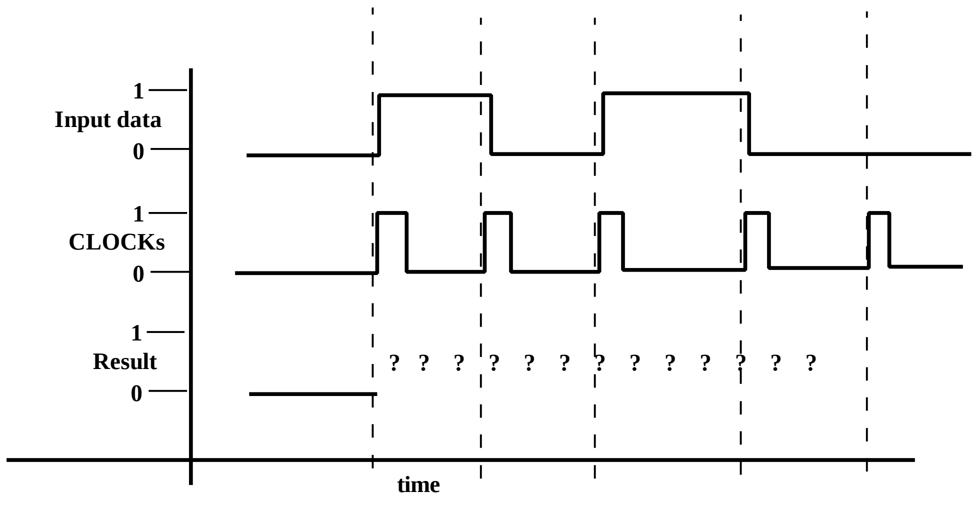
Are we ready now? It seems so; we have the sequential circuit working properly, all we need is to buy the components, wire them, apply power and we are ready to operate the computer. But, unfortunately, it may not work. There is one "small" fact we have overlooked. The operation of the processor relies on clock signals; however, the sequential circuit provides steady signals (outputs of flip-flops). What we called **CLOCKA, CLOCKB,** etc., are ordinary outputs, not clock signals. Well, one idea is to use **AND** gates to provide clock signals:



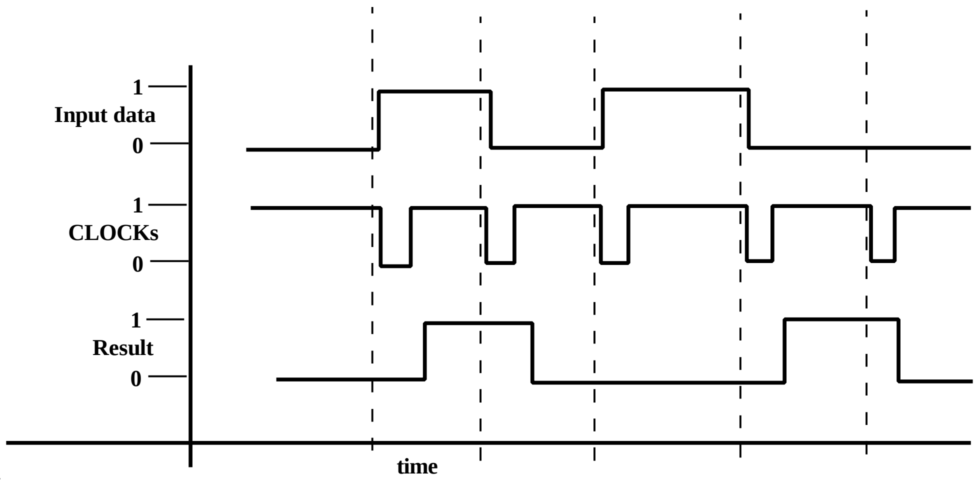
As long as we use this processor as a calculator (set up input data by hand), this will work well. Remember that flip-flop outputs always change just a bit after (delay!) of the rising edge of the clock:



However, externally programmed processors are also used as integral elements of digital systems in which case the input data would be synchronised with the system clock. In this case we have the following situation:

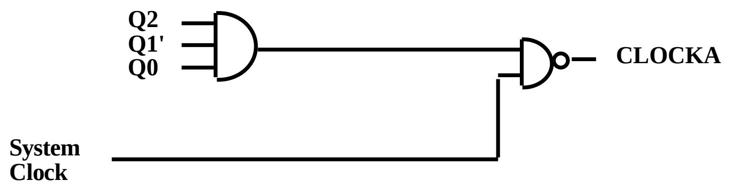


The input data are changing just about the same time when the clocks have their rising edges. This means that the actual data seen by the flip-flops is "undetermined", i.e. we cannot predict what the results will be. This does not make a very good computer, does it? Thus, we have to delay the rising edges of the clock signals which are input to the flip-flops of the sequential circuit we have just designed. The easiest way of reliably achieving this is to invert the clock signal and now we have the following stable timing diagram:



. .

Inverting the clocks means using **NAND** gates instead of **AND** gates; a trivial but significant change!



.

The processor has been now designed and can be built and it will work. How shall we advertise it?

Well, it is an eight-bit (easily expandable for any number of bits) externally programmed processor, but how many different operations can it execute? There are basically seven bits which control the operation type, three for the **ALU** function selection, three for the **Shifter** circuit, and one for the input **Carry**. If we want to be a bit optimistic, we can say that it has 2^7=128 instructions, but we would not be very honest. First of all, not all the shifter function selections with different carry input selections provide unique and meaningful operations; secondly, some operations of the **ALU** with a specific carry selection do not provide useful operations either. We will leave the determination of the number of useful operations as an exercise for you.

To complete this two lectures on a processor design, let us examine how we could produce the **decrement** operation with our computer. The machine code (explained) is given below:

Decrement a Number

1.	**Set OPERATE/IDLE signal to 0 and apply a couple of clocks.**	Processor is in **IDLE** sate
2.	**Set OPERATE/IDLE signal to 1 and apply one clock pulse.**	Processor is now waiting for the first function selection data.
3.	**Set input data to 11110000 and apply one clock pulse.**	Code is in the **IR** register. Output of **ALU** is **11111111** (= -1), **shifter** = unchanged.
4.	**Apply one clock pulse**	Because **IRX=1** the **ALU** output is clocked into the **A** register; it has now -1.
5.	**Set input data to dddddddd; the number you want to decrement**	Number is clocked now into the **B** register. **0** is clocked into the **B** register.
6.	**Set input data to 00110000 and apply one clock pulse.**	Code is in the **IR** register. The selected operation is: **plus**.
7.	**Apply one clock pulse.**	The result **"B plus (-1)"** is clocked into the **RES** register, the **C** bit indicates whether the result is equal to -1 in which case it is **0**.

Thus, we can do it !!