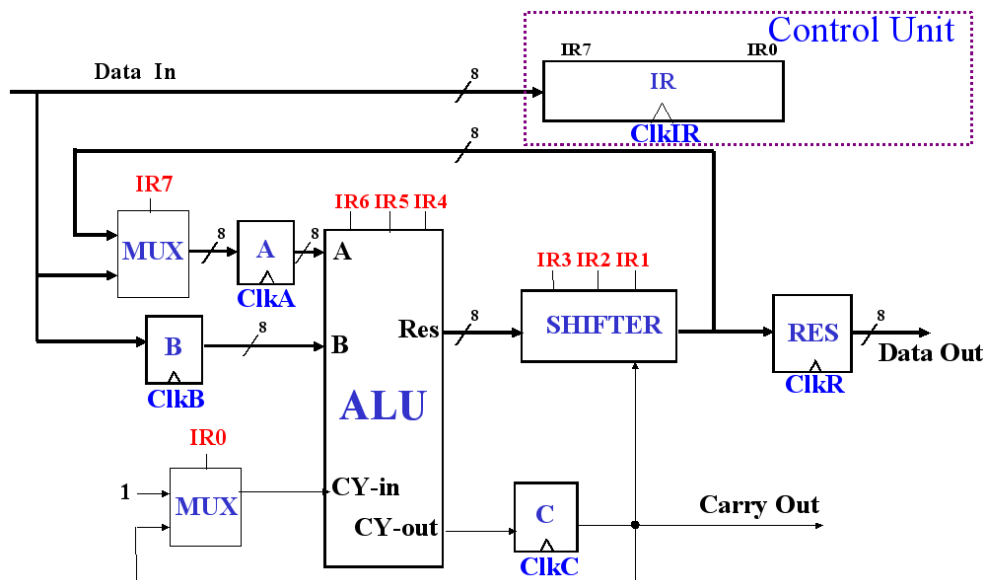


## Lecture 15: A Manual Processor (part 2)

In the last lecture a detailed diagram of a 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 Register (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 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; SELSHTF (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	

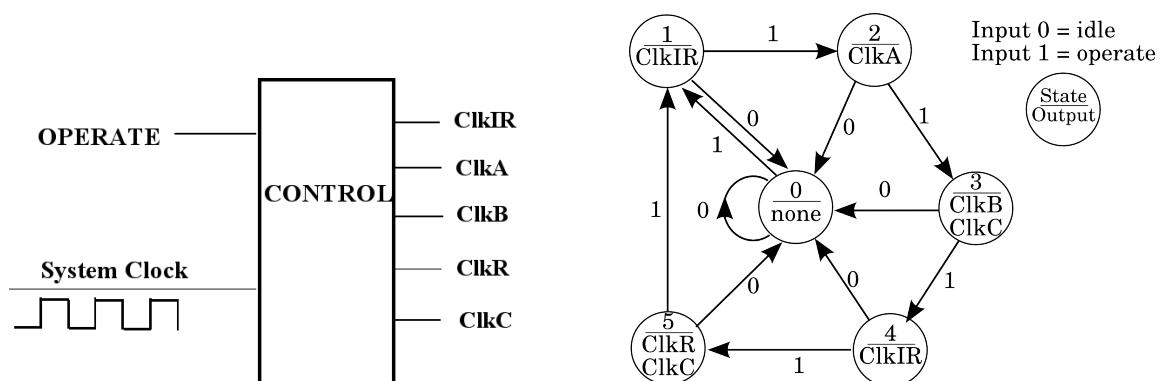
An explanation is needed why extra multiplexers were connected between the ALU and the input lines such that the output of the shifter may be loaded into the A register instead of the data on the input lines. This hardware arrangement used to be standard practice for very old and very small computers when the A register forming the input to the ALU was referred 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  $A + B$  appears at the output of the shifter. When data are clocked into the A register, the upper path is chosen (by setting  $IR_7$  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.

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. It will determine the source from which register A is loaded, and it may be necessary to set the ALU so that the input to the C flip-flop is zero. C must be set to 0 if, for example, the arithmetic operation is the A plus B function, since this adds in the carry to the result. Thus, the C flip-flop must be also loaded during the first two loading operations. The operating sequence is therefore:

- |                   |  |                                |
|-------------------|--|--------------------------------|
| 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 usually 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 of the computation |
| Go back to step 1 |  |                                |

This sequence of operations can be achieved by controlling the clock signals given to the individual registers. Thus we must now design a sequential circuit which produces these clock signals at the appropriate time within the sequence.

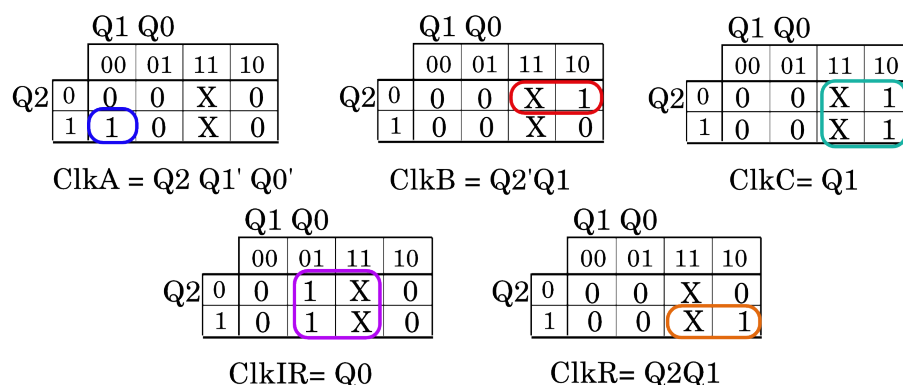
However, something is still dodgy. We need to ensure that the sequence is at Step 1 when we start the computer?. To do this, we must have an outside line which synchronises the operation of this processor. We will call this the OPERATE (value is 1) or IDLE (value is 0) input (I for short). We also must have an IDLE state in which the processor is sitting while the OPERATE/IDLE line is at value 0. When this input line is set to 1 the first clock pulse will start the processor properly. This means we need the following finite state machine diagram for the controller:



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, we will start with the output circuits, and choose a state assignment to make them as simple as possible. (This seemed to work well in for the traffic light design).

Flip-Flop Outputs	State	Required Clock Output
000	0	none
001	1	ClkIR
100	2	ClkA
010	3	ClkB, ClkC
101	4	ClkIR
110	5	ClkC, ClkRES

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

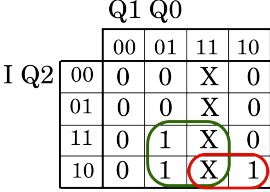


Not too bad ....

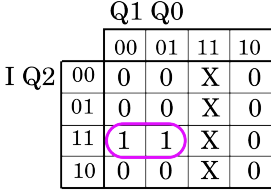
We can now design the state sequencing logic of the controller by constructing its state transition table using the above state assignment:

Operate	Current State	Flip-Flops	Next State	$D_2$	$D_1$	$D_0$
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	?	×	×	×
0	7	111	?	×	×	×
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	?	×	×	×
1	7	111	?	×	×	×

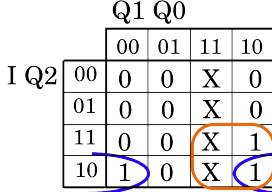
And the Karnaugh Maps and the minimised Boolean circuit equations are shown below:



$D_2 = I (Q_0 + Q_2'Q_1)$



$D_1 = I Q_2 Q_1'$



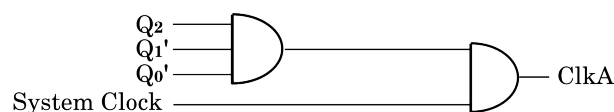
$D_0 = I(Q_1 + Q_2'Q_0')$

Checking the unused states, we have:

Operate	Current State	Flip-Flops	Next State	$D_2$	$D_1$	$D_0$
0	6	011	0	0	0	0
0	7	111	0	0	0	0
1	6	011	4	1	0	1
1	7	111	4	1	0	1

Thus, if the Operate signal is at logical 0 the system drops into the Idle state immediately and the processor is ready to start working properly.

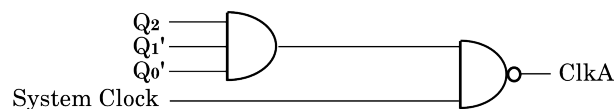
Are we ready now? It seems so; we have the sequential circuit working properly, all we need is to buy the components, wire them, apply power and we are ready to operate the computer. But, unfortunately, it may not work. There is one small thing we have overlooked. The operation of the processor relies on clock signals, however, the sequential circuit provides steady signals (outputs of flip-flops), but we need edge signals to be supplied to the registers. 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. However, remember that flip-flop outputs always change just a bit after the clock pulse. Let's suppose that we are using flip flops that change on the falling edge of the clock (like the master-slave design we saw earlier in the course). The AND gate that controls the clock pulse that goes to the register, will receive its 1 input from the controller just after the falling edge of the clock. Thus the registers will only be loaded on the next clock pulse. This is

OK if we are sure that the clock signal arrives everywhere at the same time. However, we need to ensure that the data on the data in lines is correct at the time the register is loaded. We will see in the next lecture, that the input data, which comes from RAM memory, will also be synchronised with the system clock. Everything seems to be happening at the same time and we are running the risk of race hazards, which may mean that the registers are loaded before the data is correctly in place.

To solve this problem we can make use of both the rising edge and the falling edge of the clock. Suppose we change the controller state and synchronise the data input on the falling edge, and then load the registers on the rising edge, then we can be certain that all the data is in place before the registers are loaded. The easiest way of reliably achieving this is to invert the clock signal by using a NAND gate instead of an AND gate; a trivial but significant change!



The processor has now been designed and can be built and it will work, but exactly what can it do? 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 these two lectures on a processor design, let us examine how we could execute one operation with our computer. The machine code is given below:

#### Compute (A+B)/2

- |   |  |   |
|---|--|---|
| 0 | Set OPERATE input to 0 apply two clock pulses                                    | The processor is put into the IDLE state  |
| 1 | Set input data to 1000000, set the OPERATE input to 1 and apply one clock pulse. | The instruction 10000000 is loaded in the IR, the ALU output is set to 00000000 and the shifter is set to unchanged |
| 2 | Set the Data In lines to Number A and apply one clock pulse                      | IR7=1 and so the Data In lines are loaded into the A register   |
| 3 | Set input data to Number B   | The Data In lines are loaded into the B register and 0 is loaded into register C.                                   |
| 4 | Set the Data In lines to 00110111 and apply one clock pulse                      | The instruction 00110111 is loaded in the IR, the ALU output is A plus B and the shifter does logical shift right   |
| 5 | Load the RES and the C registers   | The result "(A plus B)/2" is clocked into the RES register, the C bit indicates whether there has been an overflow. |

Go to step 1 of the next instruction

In this way we can execute a program!