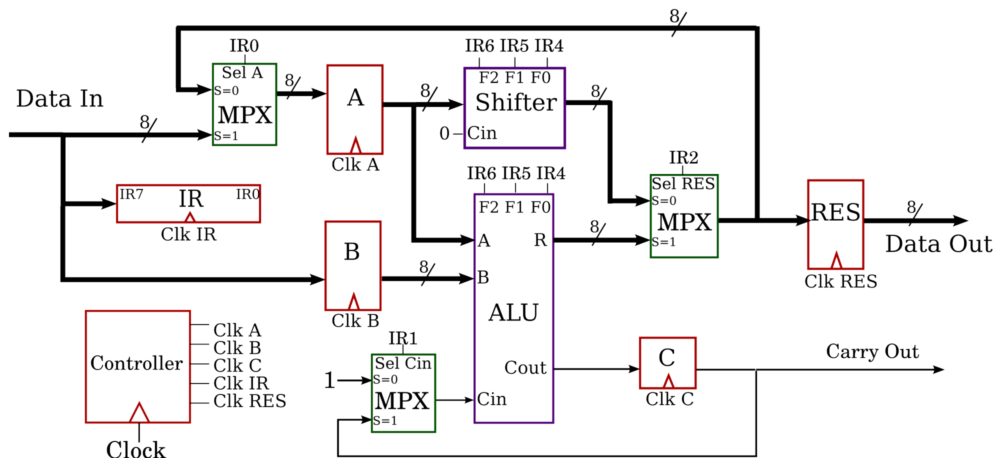
Lecture 15: A Manual Processor (part 2)

In the last lecture a detailed diagram of a manual processor was designed. The multiplexers which control the data paths and the functions of the arithmetic circuits were controlled by the bits of the instruction register (IR). They are shown as IR0 (least significant bit) to IR7 (most significant bit) on the diagram.



The bits of the instruction register (IR) are allocated for specific purposes. Bits 4-6 are used to set the function of the ALU and the shifter. Since only one of these two arithmetic circuits is used at any given time the same instruction bits can be used for both. S/R (bit 2) controls the multiplexer which selects the input to the result register (RES). Similarly S/C (bit 1) selects the input to the ALU carry (Cin) and S/A (bit 0) selects the input to register A. We show schematically these assignments below:

IR7	IR6	IR5	IR4	IR3	IR2	IR1	IR0
UN	F/ALU-SHIFT			UN	S/R	S/C	S/A

An explanation is needed why extra multiplexers were used to allow a connection from the output of the ALU or the shifter to the input of register A. 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 connection path allows the summation of a list of numbers. At first, two numbers are loaded into the A and B registers, respectively, and for example the ALU plus function is selected, which means that the sum A plus B appears at the output of the ALU. When data is next to be loaded into the A register, IR0 is set to 0 and IR2 is set to 1 establishing a path from the output of the ALU to the input of register A, and so the result (A plus B) is loaded back into register A. A new number is then loaded into the B register, and added to the accumulating sum by selecting the plus operation again. In this way a list of numbers can be added up.

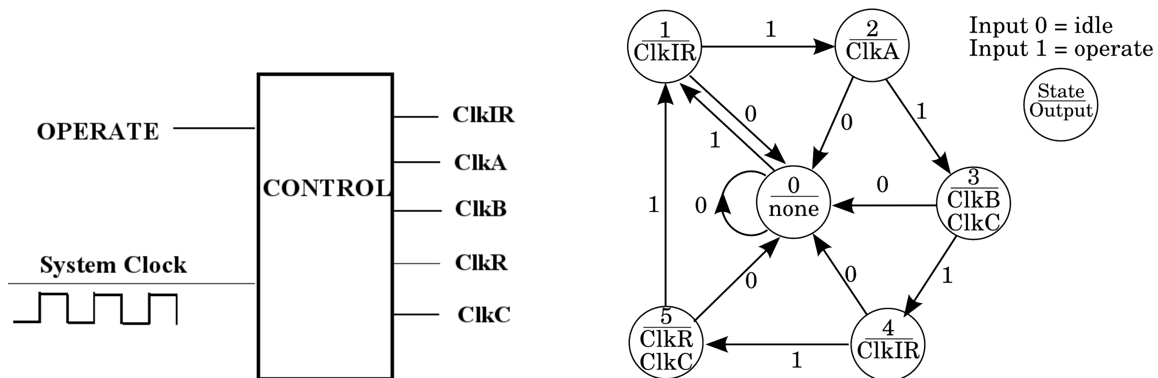
Before the input data is loaded into the A register, an operation code must be loaded into the IR register which then will control the computer's operation during the loading process. It will determine the source from which register A is loaded, and it may be necessary to set the ALU so that the input 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	

The first instruction byte, loaded in state 1 sets up the the correct data paths for the data to be loaded into registers A, B and C which are used as inputs to the arithmetic circuits. The second instruction byte loaded in state 4 determines the actual arithmetic operation used. The 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.

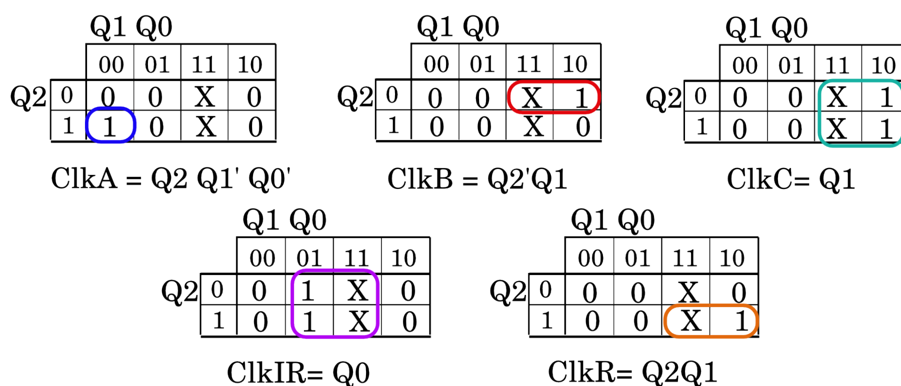
However, something is still dodgy. We need to ensure that the sequence is at Step 1 when we start the computer?. To do this, we must have an outside line which synchronises the operation of this processor. We will call this the OPERATE (value is 1) or IDLE (value is 0) input (I for short). We also must 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. 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 for the traffic light design).

$Q_2Q_1Q_0$	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	State Now	$Q_2Q_1Q_0$	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:

I 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 = I (Q0 + Q2'Q1)

I 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 = I Q2 Q1'

I Q2 \ Q1 Q0	00	01	11	10
00	0	0	X	0
01	0	0	X	0
11	0	0	X	1
10	1	0	X	1

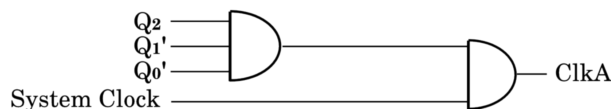
D0 = I(Q1 + Q2'Q0')

Checking the unused states, we have:

Operate	State Now	$Q_2Q_1Q_0$	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.

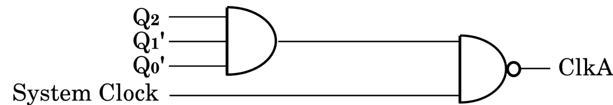
Are we ready now? It seems so; we have the sequential circuit working properly, all we need to do is to send the design to a silicon foundary, wait for our chip to come back, apply power and we are ready to operate the computer. But, unfortunately, it may not work. There is one small thing we have overlooked. 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 it's 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 six bits which control the operation type, three for the arithmetic function selection, and three for the path configuration. Each instruction is made up of two of these six bit parts. If we want to be a optimistic, we can say that it has $2^7 = 128$ instructions, but we would not be very honest. Many of the possible combinations would not carry out a useful or even meaningful task. Later in the course we will see how to design a processor where we can specify useful instructions more easily.

To complete these two lectures on a processor design, let us examine how we could execute a simple programme using two execute cycles with our computer. The sequence of operations is given below:

	Compute (A+B)/2	
0	Set the OPERATE input to 0	The processor is put into the IDLE state.
1	Set the Data In lines to x000xxx1 and the OPERATE input to 1	The instruction x000xxx1 is loaded in the IR, after which the ALU output and Cout are set to 0 and the Sel A multiplexer selects the Data In lines.
2	Set the Data In lines to Number A	IR0=1 and so the Data In lines are loaded into register A.
3	Set the Data In lines to Number B	The Data In lines are loaded into register B and 0 is loaded into register C.
4	Set the Data In lines to x011x11x	The instruction x011x11x is loaded in the IR, after which the ALU output is set to A plus B and the Sel RES multiplexer selects the ALU output. The Sel C multiplexer selects the C register making Cin=0.
5	-	The intermediate result ”A plus B” is loaded into the RES register, the C bit indicates whether there has been an overflow.

The Second execute cycle

Step	Action	Effect
1	Set the Data In lines to x011x110	The instruction x011x110 is loaded in the IR, after which the ALU output remains as (A plus B) and the Sel A and Sel RES multiplexers selects the result of the ALU.
2	-	The ALU output (A plus B) is loaded into register A.
3	-	Register B is loaded with whatever happens to be on the Data In lines - it isn't being used now. C is loaded with the ALU/Cout again.
4	Set the Data In lines to x110x0xx	The instruction x110x0xx is loaded in the IR, after which the shifter function is set to Arithmetic Right Shift and the Sel RES multiplexer selects the shifter output.
5	-	The final result (A plus B)/2 is loaded into the RES register. The carry out is not needed for this instruction. It will in fact be 0.
	Set Operate=0 to put the processor in the idle state	

In this way we can execute a program!