Lecture 18: Designing a Central Processor Unit 2: The Controller

We continue defining the functionality of our processor by considering how to implement (in hardware) all the instructions that do not use the memory.

Two Register Instructions

MOVE Rdest, Rsrc	E1	A←Rscr	
	E2	Rdest←Shifter	Shifter set to no change
ADD Rdest, Rsrc	E1	A←Rscr	
	E2	B←Rdest	
	E3	Rdest←ALUres;C←ALUcout	ALU=A+B, Cin=0
COMPARE Rdest, Rsrc	E1	A←Rscr	
	E2	B←Rdest	
	E3	C←ALUcout	ALU=A-B, Cin=0

SUBTRACT, AND, OR and XOR are all done the same way as ADD, they just have different ALU settings. COMPARE is just a subtract with a check to see that the result is zero. Because we have not incorporated any hardware to test to see if the ALU is all zero, all we can do is check the carry. This will be zero if Rscr>=Rdest, and 1 otherwise. Since the COMPARE result is used by the SKIP instructions, the carry register must be an input to the controller.

One Register instructions

CLEAR Rdest	E1	Rdest ←ALUres	ALU = zero out
INC Rdest	E1	A←Rdest	
	E2	B←ALUres	ALU = zero out
	E3	Rdest←ALUres; C←ALUcout	ALU=A+B, Cin=1
DEC Rdest	E1	A←Rdest	
	E2	B←ALUres	ALU = -1 out
	E3	Rdest←ALUres; C←ALUcout	ALU=A+B, Cin=0
COMP Rdest	E1	A←Rdest	
	E2	B←ALUres	ALU = -1 out
	E3	Rdest←ALUres	ALU=A⊕B
ASL Rdest	E1	A←Rdest	
	E2	Rdest←Shifter	Shifter set to Arithmetic left
RETURN Rdest	E1	A←Rdest	
	E2	PC←Shifter	Shifter set to no change

The other shifts will be done in the same way as ASL (arithmetic shift left). They have the same number of cycles, but the shifter settings will change. Notice that the COMP instruction simply flips the bits of the destination register. In order to negate a register we would need to use a COMP followed by and INC instruction.

No Register Instructions

All the skip instructions either increment the program counter or do nothing. The NOP instruction would not even require one cycle to execute.

SKIP	E.1	PC←PC+1	

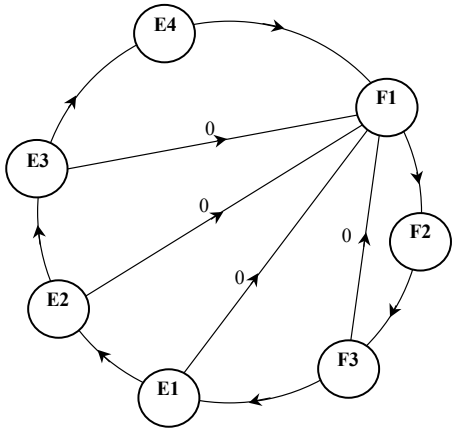
Limitations of the design

The register transfers suggest possible improvements to the hardware design. For example, if we consider the INC, DEC and COMP instructions we see that in each case register B is loaded from the main bus, where register A is loaded from the programmer's registers. Perhaps another multiplexer could be incorporated to allow both these operations to take place at the same time. There are lots of other possible improvements if you look carefully, but we are in a headlong rush to get the mark one processor out on the market.

The Fetch Cycle

Before we can execute an instruction we need to fetch it from memory. This will be the same for every instruction and will need the following register transfers:

F1 MAR←PC; PC←PC+1
F2 MDR←Memory
F3 IR←MDR



If only the memory were able to drive more than one register, we could shorten this to two cycles, but that is life. So, we now know that to execute one instruction we will need between three and seven clock cycles. We can now draw the finite state machine for our controller. At present we have not considered the outputs associated with each state, we will spend the next lecture designing the output logic. At first sight it looks remarkably simple, having only seven states. We scrapped the idle state in the manual processor, since the software department told me that the processor will never be idle. If it's not executing a program it will be interrogating the input/output devices for new data.

The first problem is to determine the conditions for the state changes. Clearly F1 to F2, F2 to F3 and E4 to F4 are unconditional, but for the others we will need to determine which instruction we are executing, and how many cycles it will need. This looks difficult, since the instruction is determined by an 8 bit opcode, and we don't really want 8 inputs to the state sequencing logic if we are to follow our established design method. However, we can now make use of the fact that we have not as yet chosen what the opcodes should be for each instruction. If there were some bits of the opcode whose function was to indicate the number of cycles required by an instruction then we could make use of those bits to design our state sequencing logic. Looking at the instructions that we have designed we chose the following pattern:

No of Cycles	No of Instructions	IR Bits 31:30:29
0	1	0 0 0
1	5	0 1 *
2	7	1 0 *
3	13	1 1 *
4	1	0 0 1

If we indicate instructions that take three cycles by putting the top two bits to 11, we can accommodate a possible 64 such instructions. Similarly we can allow for up to 64 two and one cycle instructions and 32 zero or four cycle instructions. This gives us a lot of flexibility since we know that, sooner or later, the software department will be on the phone asking for more instructions despite our warnings. Our problem now has three inputs and three flip flops determining the state, so, unless we can simplify things further, we will not be able to minimise the state sequencing logic using Karnaugh maps. We don't really want to use an un-minimised circuit, so we could look for a way to reduce the problem from a six input one to a four input one. It turns out that by choosing a good state

assignment we can fix things so that the condition for returning to F1 can be evaluated at each state. The problem is, can we find a simple function that will be 1 if we need to continue to the next execute cycle and zero otherwise. The XOR function does the job for us. For example, the condition for returning from E1 to F1 is IR31=0 and IR30=1, if the state E1 had assignment Q2=0, Q1=1, then we could use the condition (IR31⊕Q2 + IR30⊕Q1). This would work everywhere except for four cycle instructions which will yield zero at state F3. To fix this we arrange that Q0=1 at F3 and zero at E1,E2 and E3, and now our condition becomes:

C = IR31⊕Q2 + IR30⊕Q1 + IR29•Q0

C=0 means return to F1, C=1 means continue to the next state. We have allocated completely the states F3, E1, E2 and E3, and we can now complete the state assignment table in any consistent way. We chose:

State	Q2	Q1	Q0	Minterm
F1	0	0	0	Q2'•Q1'•Q0'
F2	0	1	1	Q2'•Q1•Q0
F3	0	0	1	Q2'•Q1'•Q0
E1	0	1	0	Q2'•Q1•Q0'
E2	1	0	0	Q2•Q1'•Q0'
E3	1	1	0	Q2•Q1•Q0'
E4	1	1	1	Q2•Q1•Q0
Unused	1	0	1	Q2•Q1'•Q0

Now we have reduced our problem to a standard sequential design, which we can solve by the usual design method.

The truth table that expresses the sequencing logic is as follows:

C	This State	Q2	Q1	Q0	Next State	D2	D1	D0
0	F1	0	0	0	F2	0	1	1
0	F2	0	1	1	F3	0	0	1
0	F3	0	0	1	F1	0	0	0
0	E1	0	1	0	F1	0	0	0
0	E2	1	0	0	F1	0	0	0
0	E3	1	1	0	F1	0	0	0
0	E4	1	1	1	F1	0	0	0
0	Unused	1	0	1	*	*	*	*
1	F1	0	0	0	F2	0	1	1
1	F2	0	1	1	F3	0	0	1
1	F3	0	0	1	E1	0	1	0
1	E1	0	1	0	E2	1	0	0
1	E2	1	0	0	E3	1	1	0
1	E3	1	1	0	E4	1	1	1
1	E4	1	1	1	F1	0	0	0
1	Unused	1	0	1	*	*	*	*

This yields the following Karnaugh Maps, and equations for the state sequencing logic:

D2

C Q2 \ Q1 Q0	00	01	11	10
00	0	0	0	0
01	0	*	0	0
11	1	*	0	1
10	0	0	0	1

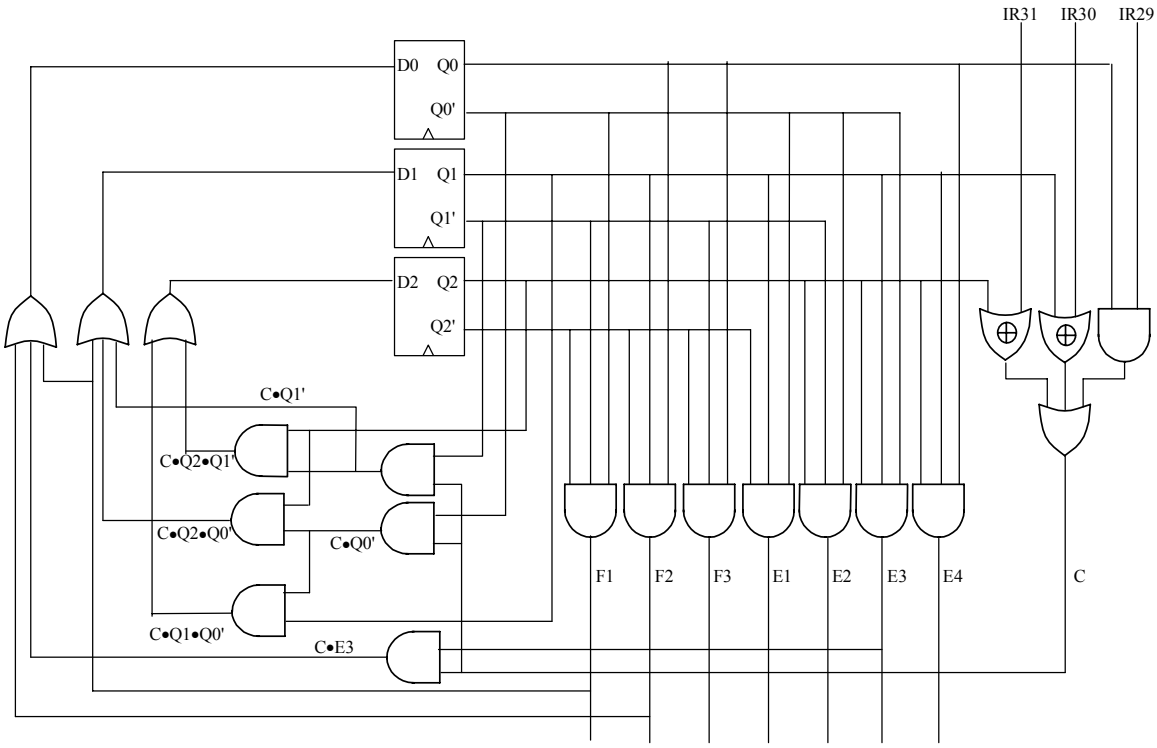
D1

C Q2 \ Q1 Q0	00	01	11	10
00	1	0	0	0
01	0	*	0	0
11	1	*	0	1
10	1	1	0	0

D0

C Q2 \ Q1 Q0	00	01	11	10
00	1	0	1	0
01	0	*	0	0
11	0	*	0	1
10	1	0	1	0

D2 = C•Q2•Q1’ + C•Q1•Q0’
D1 = C•Q1’ + C•Q2•Q0’ + Q2’•Q1’•Q0’
D0 = Q2’•Q1’•Q0’ + Q2’•Q1•Q0 + C•Q2•Q1•Q0’
The equation for D0 can be simplified further by use the exclusive or simplification rule
D0 = Q2’• (Q1⊕Q0)’ + C•Q2•Q1•Q0’.
In practice though, this will not help since we have to decode the states in our output logic. The final circuit for the state sequencing logic is:



We need to back check the don’t care states to see that the machine will not get trapped. It turns out that for C=0, the unused state will jump to F0 (000), and if C=1 it will jump to E2 (110). This is probably acceptable. In any case it should not be a problem as we will definitely need to add some extra hardware to force the processor to do a particular operation at start up. In practice this will be a jump to the start of a stored program in ROM, which will load a minimal operating system.