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

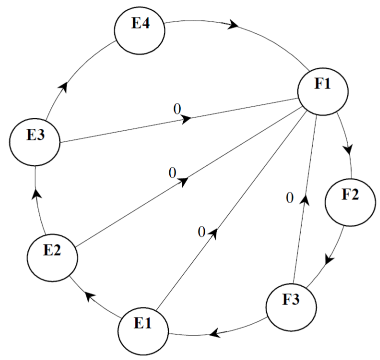
In the last lecture we defined the format of the instructions and the register transfers that are required to execute them. We now turn to the question of building a synchronous machine that will be able to carry out each instruction. At first this looks like a mammouth task, since we have so many possibilities. However with a bit of care, and a judicious use of Boolean algebra we will find that we can manage the complexity quite easily.

The Fetch Cycle

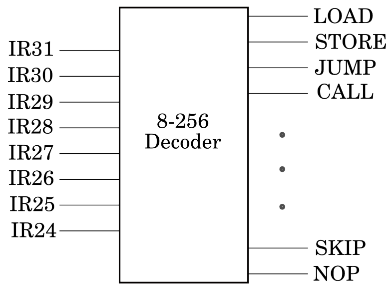
Before we can execute an instruction we need to fetch it from memory. This will be the same for every instruction and will need the following register transfers:

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 design the output logic later. At first sight it looks remarkably simple, having only seven states. We scrapped the idle state in the manual processor, since the software department tells me that the processor will never be idle. If it's not executing a program it will be interogating 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. Instead we will try to create a single input to the controller, called C, which will be 1 if we advance to the next state, and zero if we return to state F1. Our old friend the decoder (aka demultiplexer) will solve this problem for us. It takes a binary number as input and produces a unary output. We previously designed one functionally and it would be quite easy scale that design to 8 inputs and 256 outputs. If we connect the instruction register bits 31-24 to the input bits, then each of the 256 output lines indicates one particular instruction.

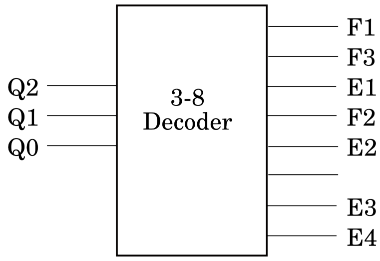


With this arrangement we know that if, for example, STORE=1 then we are executing a STORE instruction, and all other output lines from the decoder will be zero. We can therefore use the instruction names as Boolean variables when trying to determine our state sequences. We also noted that some instructions have exactly the same sequence of register transfers, so we can make this explicity by implementing simple disjunctions of the instructions, for example defined by the equations:

ADDS= ADD + SUBTRACT + AND + OR + XOR
SHIFTS = ASL + ASR + ROR
SKIPS = SKIP + SKIPPOSITIVE + SKIPNEGATIVE

Now suppose also that we do the state assignment, and we choose to use the table below, We can simply use another decoder to determine which state we are in. This time it is just a simple 3-8 decoder.

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



This means we can use the state names as Boolean variables as well, and so we can now straightforwardly write equations expressing the conditions under which the processor will return to state F1, or advance to the next execute state. All we need to do is group our instructions according to how many execute states they take. For example, the instructions that need two execute cycles are RETURN, SHIFTS, MOVE, and JUMPINDIRECT, so the condition for returning from state E2 to state F1 is simply:

$$(\text{E2} \cdot (\text{RETURN + SHIFTS + MOVE + STORE + JUMPINDIRECT}))'$$

Extending this to the other states we get a cumbersome, but easily comprehensible equation for the input to the controller (C):

$$\begin{aligned} \text{C} = \;& (\text{F3.NOP})' \cdot \\ & (\text{E1} \cdot (\text{SKIPS + CLEAR + JUMP}))' \cdot \\ & (\text{E2} \cdot (\text{RETURN + SHIFTS + MOVE + STORE + JUMPINDIRECT})))' \cdot \\ & (\text{E3} \cdot (\text{COMP + DEC + INC + COMPARE + ADD +} \\ & \qquad \text{STOREINDIRECT + LOAD + CALL + CALLINDIRECT}))' \end{aligned}$$

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:

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 Q1 Q0

C Q2

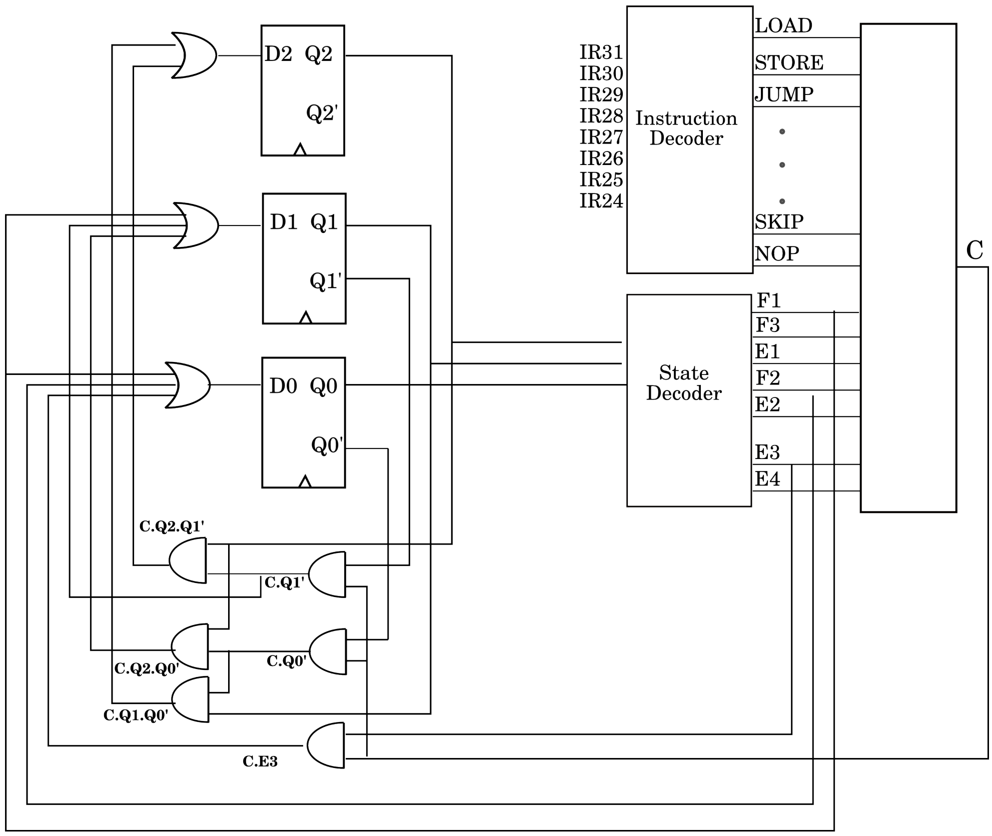
	00	01	11	10
00	1	0	0	0
01	0	*	0	0
11	1	*	0	1
10	1	1	0	0

D0 Q1 Q0

C Q2

	00	01	11	10
00	1	0	1	0
01	0	*	0	0
11	0	*	0	1
10	1	0	1	0

$D2 = C\cdot Q2\cdot Q1' + C\cdot Q1\cdot Q0'$
$D1 = C\cdot Q1' + C\cdot Q2\cdot Q0' + Q2'\cdot Q1'\cdot Q0'$
$D0 = Q2'\cdot Q1'\cdot Q0' + Q2'\cdot Q1\cdot Q0 + C\cdot Q2\cdot Q1\cdot Q0'$
The equation for D0 can be simplified further by use the exclusive or simplification rule
$D0 = Q2'\cdot (Q1 \oplus Q0)' + C\cdot Q2\cdot Q1\cdot Q0'$.
In practice this will not help since we have had to decode the states to generate the C signal. Given that we have already done this we can write:
$D1 = C\cdot Q1' + C\cdot Q2\cdot Q0' + F1$
$D0 = F1 + F2 + C\cdot E3$
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.

The Output Logic

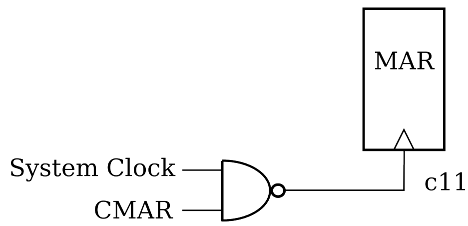
The final step in designing the processor is to design the output logic for the controller. This is the logic that determines the values of the clock gates (c0-c14), the arithmetic functions (f0-f4) and the multiplexer settings (s0-s8). These values will depend on both the state of the processor and the instruction being executed. We will again make use of the Boolean variables:

F1, F2, F3, E1, E2, E3, E4, LOAD, STORE, JUMP, etc.

that we implemented with decoders when designing the state sequenceing logic.

The Clock Gates

During any state of the processor, certain registers are loaded, but not all registers. The clock gates control which registers are loaded by gating the system clock. For example, if we choose the MAR register, the circuit will be:



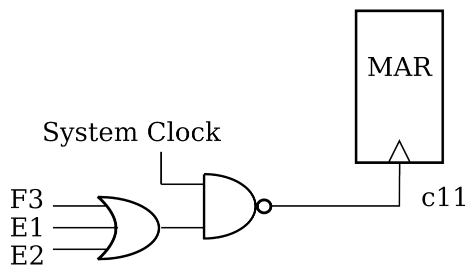
Thus, assuming that the controller changes its state on the falling edge of the clock, and the variable CMAR becomes 1 indicating that the MAR is to be loaded, then it will receive a clock pulse from the system clock on its next rising edge. If, on the other hand, CMAR=0 during any controller state then the MAR register cannot change. To determine the equation for CMAR, and the other clock gates, we need to go back to the register transfers and determine which cycles each register receives a clock pulse and for which instructions. Looking through the instructions we find that:

CMAR = F1 + E1·(LOAD + STORE) + E2·(LOADINDIRECT + STOREINDIRECT)

In other words the MAR changes state always in state F1, and in E1, but only when the LOAD or STORE instructions are being executed, and lastly in state E2 when the LOADINDIRECT and STOREINDIRECT instructions are being executed. This is a simple Boolean equation for which we can construct a direct implementation. However, we can simplify it considerably if we note that during certain cycles it doesn't matter if we load the MAR, since we are not using the memory. Looking at the register transfers we do not need its value to be retained further than the next cycle, so we can write:

CMAR = F1 + E1 + E2

Thus the circuit for determining the output c11 is:



The equation for the MDR is found similarly

CMDR = F2 + E2·LOAD + E3·LOADINDIRECT

and again we can simplify this since we usually only load the MDR on the cycle before we use it. However, we need to preserve its value between E1 and E3, during the LOADINDIRECT instruction, and so:

$$\text{CMDR} = \text{F2} + \text{E2}\cdot\text{LOAD} + \text{E3}$$

The instruction register IR c12 is even simpler since it always receives a clock edge in F2 and at no other time, thus

$$\text{CIR} = \text{F2}$$

The memory clock, c13, receives a pulse only when a store instruction is executed. Hence we have:

$$\text{CMemory} = \text{E2}\cdot\text{STORE} + \text{E3}\cdot\text{STOREINDIRECT}$$

We can't simplify this for fear of corrupting the memory.

The program counter has rather a nasty equation:

$$\text{CPC}=\text{F1}+\text{E1}\cdot(\text{CALL} + \text{CALLINDIRECT} + \text{JUMP} + \text{SKIP} + \text{SKIPPOSITIVE}\cdot C'_{ALU} + \text{SKIPNEGATIVE}\cdot C_{ALU}) + \text{E2}\cdot(\text{JUMPINDIRECT}+\text{RETURN}) + \text{E3}\cdot(\text{CALLINDIRECT}+\text{CALL})$$

Unfortunately it does not seem likely that we can simplify this further, since we need to maintain the value in the program counter from one instruction to the next.

For the A register, (c7) we can always load it in cycle E1, and in some cases we need the value preserved through cycle E2. Hence we can use:

$$\text{CA} = \text{E1} + \text{E2}\cdot\text{STOREINDIRECT}$$

The B register (c8) can always be loaded in the E2 state

$$\text{CB} = \text{E2}$$

The C register records the result of an arithmetic operation for the next instruction. Since all arithmetic operations reach the result in E3 we could take the easy option and write:

$$\text{CC} = \text{E3}$$

though this would mean that if a skip instruction were used after a non arithmetic operation (say a LOAD instruction) its result would be unpredictable, since the ALU carry would have been loaded during the E3 state of the LOAD instruction. A safer implementation would be to load it during E3, but only for one or two register instructions. Rather than write these out in full we could express this as:

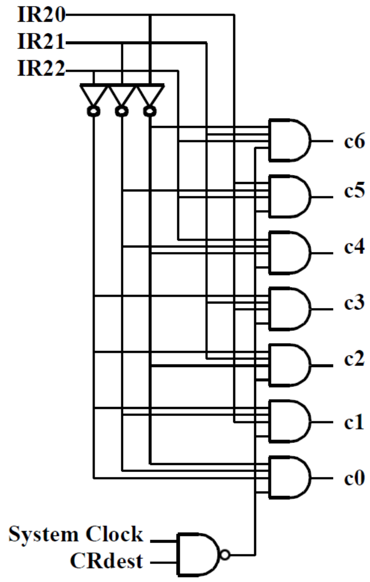
$$\text{CC}= \text{E3}\cdot(\text{ONE}+\text{TWO})$$

Where ONE and TWO are Boolean variables which become 1 when the instruction being executed uses, respectively, one or two registers. We can derive the equations for these from the register transfer tables, and implement them straightforwardly. We will see later that it will be helpful to use the Boolean variables ONE and TWO in other cases.

We have now determined all the clock gates except for those of the seven general purpose registers. remember that we have been careful in defining the instruction bits to ensure that if a register is to be loaded its number will be stored in the instruction register at bits 20-23. Going through our register transfer definitions we can write that there should be a clock pulse on Rdest in the following conditions:

$$\text{CRdest} = \text{E4}+ \text{E3}\cdot(\text{LOAD}+\text{ONE}+\text{ADDS}) + \text{E2}\cdot(\text{SHIFTS} + \text{MOVE} + \text{CALL}+\text{CALLINDIRECT}) + \text{E1}\cdot\text{CLEAR}$$

This equation cannot be easily simplified, since the general purpose registers are used by the programmers and we cannot arbitrarily change their values. We need to decode bits 20 to 23. In the instruction we have allowed four bits for the register indicators, and so we will be able to incorporate up to sixteen registers if we choose to expand the processor. However, in the present design we have only seven, so a 3bit decoder will serve the purpose.



Output Logic 2: The arithmetic function selectors.

The shifter function can be easily determined, since most of the time it is used it is in the unchanged mode (passing data from A to the internal bus). The shifts only occur in the shift instructions. The shifter function is described by the following table:

Instruction	Function	f4	f3
Default	No Action	0	0
ASL	Arithmetic Shift Left	0	1
ASR	Arithmetic Shift Right	1	0
ROR	Rotate Right	1	1

From this table we can see that we can implement these function lines using:
f4 = ASR+ROR
f3 = ASL+ROR
The default is then 00 and need not be explicitly set.

The use of the ALU is as follows:

Instruction	Function	f2	f1	f0
Default	Zero	0	0	0
Unused	B-A	0	0	1
E3·(SUBTRACT + COMPARE)	A-B	0	1	0
E3·(DEC + INC + ADD)	AplusB	0	1	1
E3·COMP	A⊕B	1	0	0
E3·OR	A + B	1	0	1
E3·AND	A·B	1	1	0
E2·(DEC + COMP)	-1	1	1	1

As with the shifter we need look at the table columns to determine the equations for the function bits, so:
f2 = E3·(COMP+OR+AND) + E2·(COMP+DEC)
f1 = E3·(SUBTRACT+COMPARE+DEC+INC+ADD+AND) + E2·(COMP+DEC)
f0 =E3·(DEC + INC + ADD + OR) +E2·(COMP+DEC)
The carry input is required to be a 1 in only 1 place, which is INC·E3. For all other cases it must be set to zero. Hence we have:
f4 = INC·E3

Output Logic 3: The Select Inputs

The selection inputs are defined by the following three tables.

Selection	s2	s1	s0	Selection	s6	s5	s4	Selection	s3
R0	0	0	0	Shifter	0	0	0	Bus	0
R1	0	0	1	ALU	0	0	1	Incrementer	1
R2	0	1	0	PC	0	1	0		
R3	0	1	1		0	1	1		
R4	1	0	0	Mask	1	0	0		
R5	1	0	1	MDR	1	0	1		
R6	1	1	0		1	1	0		
Bus	1	1	1		1	1	1		

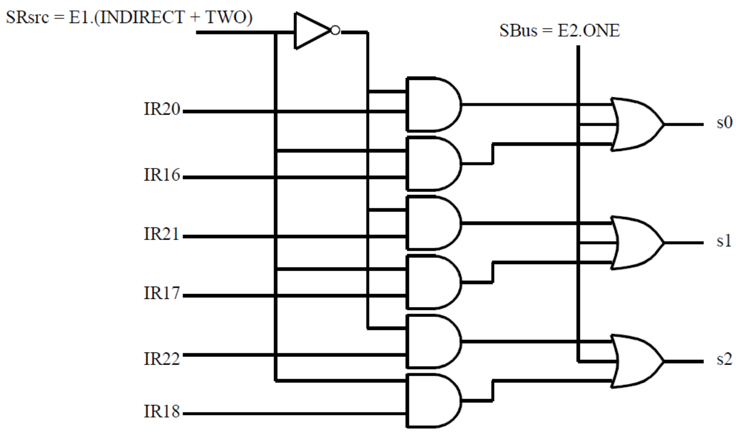
In most cases the register select, s2,s1,s0 can be taken directly from the Rsrc or the Rdest fields of the instructions. Examining the register transfers we find that A and B are loaded from the bus (as opposed to the registers) during E2·INC, E2·COMP and E2·DEC. In fact we note that if we are executing a one register instruction we can always set the selector for A and B to the bus (1,1,1). So we can write this condition as

$$\text{SBus} = \text{E2}\cdot\text{ONE}$$

At other times A and B can be loaded from the registers. Going through the register transfers we see that A and B need to be connected to Rsrc, (which will be the instruction register bits 16-19) during state E1 for all the indirect instructions and all the two register instructions. At all other times it can be connected to Rdest, which is IR bits 20-23. This is required for the STORE instructions. We can therefore write another select condition as:

$$\text{SRsrc} = \text{E1}\cdot(\text{INDIRECT} + \text{TWO})$$

With these two functions implemented we see that the selection bits s0,s1 and s2 can be provided by the following circuit.



The internal bus selector has three spare inputs, which we could use to add further functionality to the processor. The most common usage is to loop A back to the registers via the shifter, so we will set this as default, and look at the conditions where we need to select the other inputs.

$$\text{SPC} = \text{E2}\cdot(\text{CALL+CALLINDIRECT})$$
$$\text{SALU} = \text{E1}\cdot\text{CLEAR} + (\text{E2+E3})\cdot(\text{INC+DEC+COMP}) + \text{TWO}\cdot\text{E3}$$
$$\text{SMask} = \text{E1}\cdot\text{LOAD+JUMP} + \text{STORE}) + \text{E3}\cdot\text{CALL}$$
$$\text{SMDR} = \text{LOAD}\cdot\text{E3} + \text{E4}$$

Looking again at the table defining the selection bits we can see that:

$$\text{S4} = \text{SALU} + \text{SMAR}$$
$$\text{S5} = \text{SPC}$$
$$\text{S6} = \text{SMask} + \text{SMDR}$$

With thought we could probably simplify these equations by looking at the times when the internal bus is and isn't used, but for the moment we will press on.

The PC selector has the default to be BUS (0) and we set it to the incrementer (1) with the following conditions:

$$S3 = F1 + E1 \cdot (CALL+CALLINDIRECT)$$

The Mark 2 version

This completes the design of the output logic, so we can now make a wiring list, simulate test and build the circuits. If we use components from maplin, it will only cost about twice as much as an Intel Core processor, and we can probably run it as fast as 100KHz (yes K not M). So, if we want to sell any we will need to think about a mark 2 version.

Our aim in designing the mark one processor was to keep everything simple, but several of our choices would result in the speed being slower. So here are one or two suggestions as to how to make the mark 2 version of the processor go faster.

Firstly, we used 32 bits for each instruction, but only four of our instruction set use those 32 bits. The rest are at most 16 bits. This means that for most of our fetch cycles we are fetching redundant bits from memory. In order to keep the speed up we still should fetch 32 bits at a time, but we could reduce the number of fetches if we could pack all instructions, other than the memory reference ones into 16 bits. This would reduce the number of fetch cycles by nearly half, and thus increase the speed by more than one quarter. There are many possible strategies we could adopt for this. The simplest would be to arrange some multiplexers so that we can switch either the top or the bottom sixteen bits of the IR to the controller inputs. The memory reference instructions could be forced to start on a 32 bit boundary by inserting a NOP instruction where necessary. There are more complex, and more efficient ways of packing up the program instructions.

Secondly, we could introduce more arithmetic hardware. For example, we could introduce a 16 bit multiplier that multiplies the bottom bits 16 of A and B to produce a 32 bit result. Other hardware additions could include a register to record when the result of an arithmetic operation was zero. By adding in a multiplexer to select register B independently from A we could almost certainly reduce the number of execution cycles for a large number of instructions.

Lastly, we could consider ways in which we could speed up the clock. This would mean looking at the combinational logic (multiplexers and arithmetic circuits and the state sequencing and output logic) to see if we can optimise its speed.