Lecture 19: Designing a Central Processor Unit 3: 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 assume that each state has been decoded into a single line, and we will use Boolean variables:

F1,F2,F3,E1,E2,E3 and E4

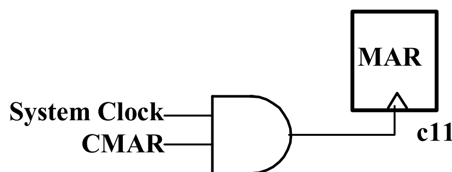
to indicate the state. Thus, for example E1=1 if the processor is in state E1 and 0 at all other times. We will also assume that the instruction being executed has been decoded, and we will name the corresponding Boolean variables the same as the instruction:

LOAD, STORE, JUMP, CALL, LOADINDIRECT etc.

Thus LOAD=1 if and only if we are executing the LOAD instruction. We will return to the question of implementing these Boolean variables later.

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:



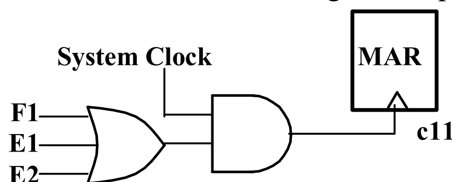
Thus, if we can set the variable CMAR to 1 for each state where the MAR is to be loaded, then it will receive a clock pulse from the system clock at the same time that the state change occurs. If, on the other hand CMAR=0 then the MAR register cannot change. To determine the circuit for CMAR, and the 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 write 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:

CMDR = F2 + E2•LOAD + E3

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

CIR = F2

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

CMemory = E2•STORE + E3•STOREINDIRECT

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

The program counter has rather a nasty equation:

CPC=F1+E1•(CALL + CALLINDIRECT + JUMP + SKIP +
SKIPPOSITIVE•C' + SKIPNEGATIVE•C) +
E2•(JUMPINDIRECT+RETURN) + E3•(CALLINDIRECT+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:

CA = E1 + E2•STOREINDIRECT

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

CB = E2

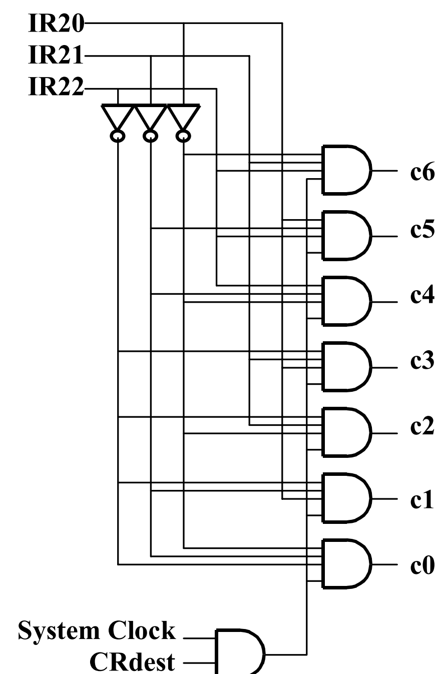
The C register records the result of an arithmetic operation for the next instruction. Since all arithmetic operations reach the result in E3 we could take the easy option and write:

CC = 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:

CC= E3•(ONE+TWO)

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. Here we have been careful in laying out the instructions 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:

CRdest = E4+ E3•(ONE + LOAD) +
E2•(SHIFT + MOVE +
CALL+CALLINDIRECT) +
E1•CLEAR

Where ONE, as above, means all the one register instructions, and SHIFT means any shift instruction, or in Boolean equations SHIFT=ASL+ASR+LSR. 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. The decoder is shown in the figure above.

Output Logic 2: The arithmetic function setters.

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	f4	f3
	0	0
ARITHMETIC SHIFT LEFT (ASL)	0	1
ARITHMETIC SHIFT RIGHT (ASR)	1	0
LOGICAL SHIFT RIGHT (LSR)	1	1

From this table we can see that we can implement these function lines using:

f4 = ASR+LSR

f3 = ASL+LSR

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)	AplB	0	1	1
COMP.E3	A⊕B	1	0	0
OR.E3	AorB	1	0	1
AND.E3	AandB	1	1	0
DEC.E2 + COMP.E2	-1	1	1	1

As with the shifter we need to turn these round, so that

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 defiend as follows

s2	s1	s0	Select	s6	s5	s4	Select	s3	Select
0	0	0	R0	0	0	0	Shifter	0	Bus
0	0	1	R1	0	0	1	ALU	1	incrementer
0	1	0	R2	0	1	0	PC		
0	1	1	R3	0	1	1			
1	0	0	R4	1	0	0	Mask		
1	0	1	R5	1	0	1	MAR		
1	1	0	R6	1	1	0			
1	1	1	Bus	1	1	1			

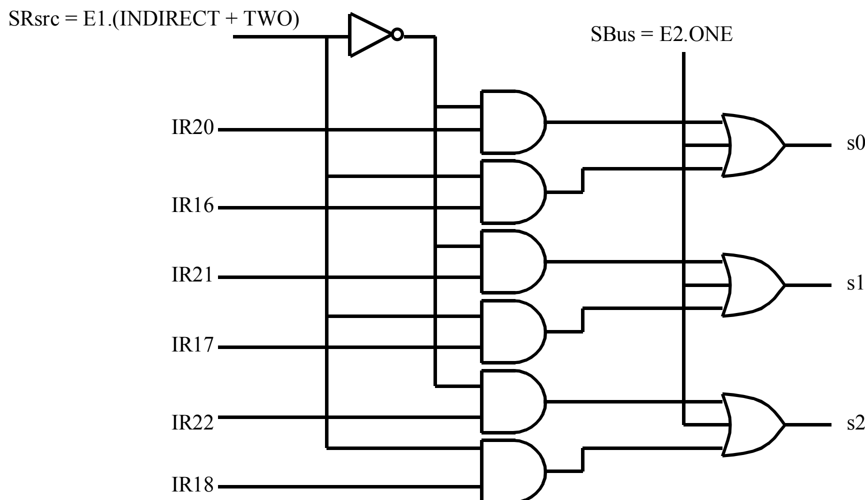
Firstly, the register select, s2,s1,s0 can be taken directly from the Rsrc or the Rdest fields of the instructions. Examining the register transfers we find that A and B are loaded from the bus (as opposed to the registers) during E2•INC, E2•COMP and E2•DEC. In fact we note that if we are executing a 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

SBus = E2•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. We can therefore write another select condition as:

SRsrc = E1•(INDIRECT + 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.

SPC = E2•(CALL+CALLINDIRECT)
SALU = E1•CLEAR + (E2+E3)•(INC+DEC+COMP) + TWO•E3
SMask = E1•LOAD+JUMP + STORE) + E3•CALL
SMDR = LOAD•E3 + E4

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

S4 = SALU + SMAR
S5 = SPC

S6 = SMask + 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•(CALL+CALLINDIRECT)

Defining the opcodes

To finish up, we can consider how to define the opcodes to simplify our output logic as far as possible. We have already specified that the top two or three bits are used to determine the number of memory cycles that are required. We have also seen that it would be useful to have a simple way of telling which type of instruction we have (One register, Two register etc.). We can do this quite simply by choosing bit 29 = 1 for an indirect instruction, and bit 28 to be 1 for a two register instruction. In our equations above we can now substitute INDIRECT=IR29, TWO=IR28 This means that four of our opcodes are now completely defined. These are:

LOADINDIRECT which is the only four cycle instruction:

LOADINDIRECT = IR31'•IR30'•IR29

JUMPINDIRECT which is the only three cycle instruction which is also indirect:

JUMPINDIRECT = IR31•IR30'•IR29

MOVE which is the only 2 register 2 cycle instruction

MOVE = IR31•IR30'•IR29'•IR28

NOP which is the only no cycle instruction, (though if we were clever we might be able to make some of the skips fall into this category).

NOP = IR31'•IR30'•IR29'

The other instructions fall into the following categories:

1101 Three cycle two register instructions, ADD, COMPARE, SUBTRACT, AND, OR and XOR
1100 Three cycle instructions (excluding 2 register and indirect) LOAD, CALL, DEC, INC, COMP
1000 Two cycle instructions (excluding 2 register and indirect) STORE, ASL, ASR, LSR, RETURN
0100 One cycle instructions SKIP, SKIPPOSITIVE, SKIPNEGATIVE, CLEAR, JUMP

We can allocate all these using the next three bits, leaving one bit spare, and we can write Boolean equations for the other variables that we have introduced above (SHIFT and ONE). Notice that we are using the massive redundancy of our instruction set to simplify both the state sequencing logic and the output logic. If we did not have this, we would need to take more care of how we define our opcodes. We can use our last bit to save some hardware in simplifying some of our boolean expressions, though this saving is hardly worth doing, so we will just reserve it for hardware expansion.

The Mark 2 version

This completes the design of the output logic, so we can make a wiring list, build the circuits and test our processor. It will only cost about twice as much as a Pentium IV, 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.