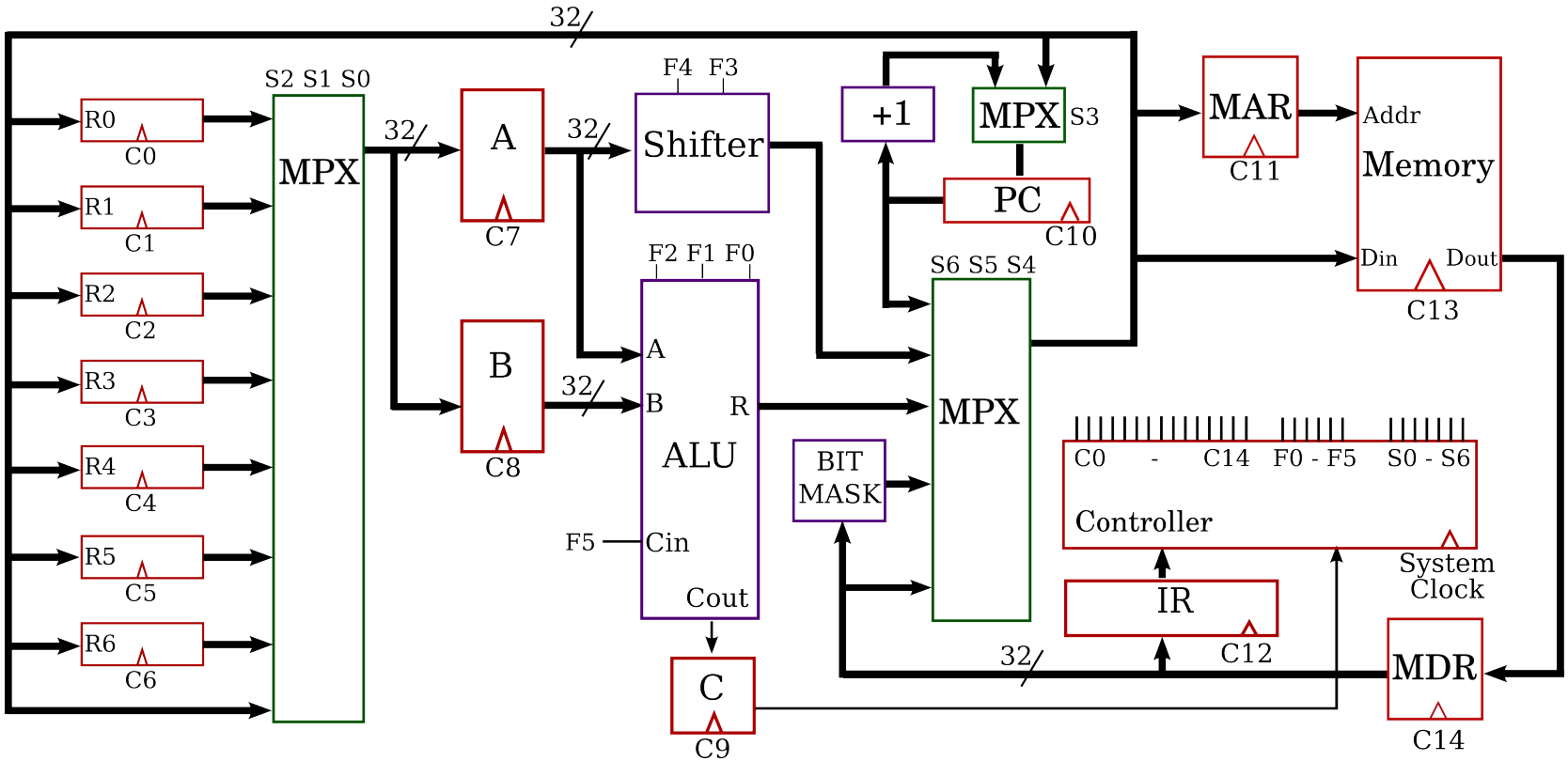
Lecture 18

A 32-bit Processor:

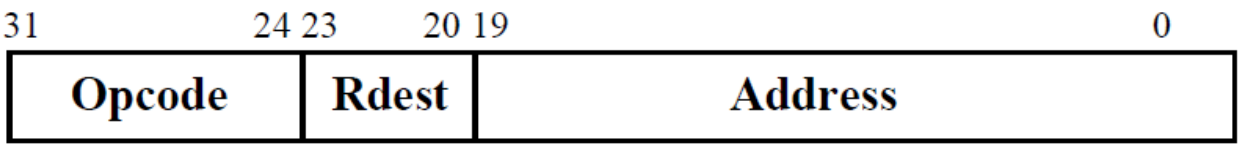
Sequencing and Output Logic

Last lecture we defined the data paths:

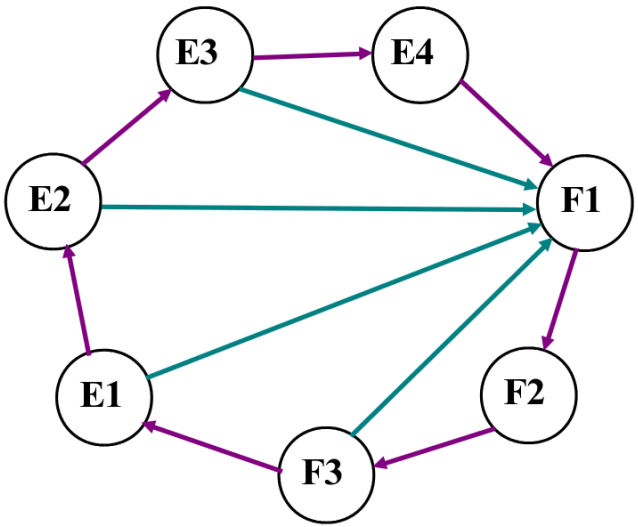


and we specified an instruction set:

Instruction	Cycle	Transfers	Path
LOAD Rdest, Address	E1	MAR←MDR	Use the mask
	E2	MDR←Memory	
	E3	Rdest←MDR	No Mask
STORE Rdest, Address	E1	MAR←MDR; A←Rdest	Via the mask
	E2	Memory←A	Shifter (unchanged)
JUMP Address	E1	PC←MDR	Via the mask
CALL Rdest, Address	E1	PC←PC+1	
	E2	Rdest←PC	
	E3	PC←MDR	Via the mask



The next job is to design the controller

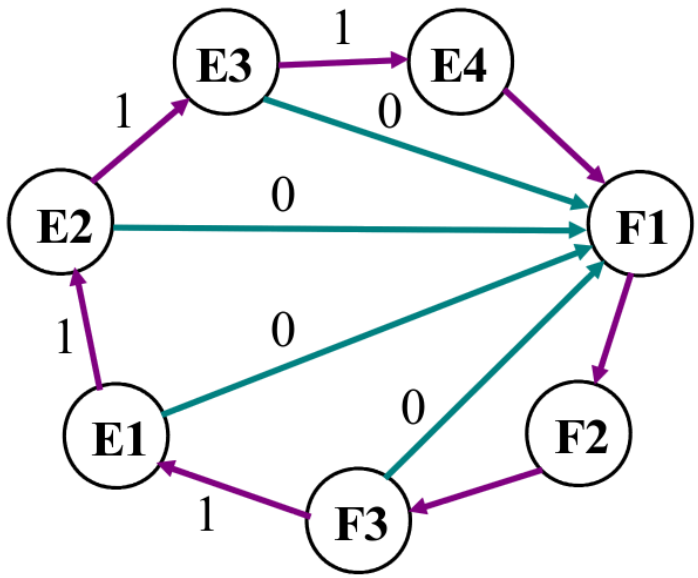


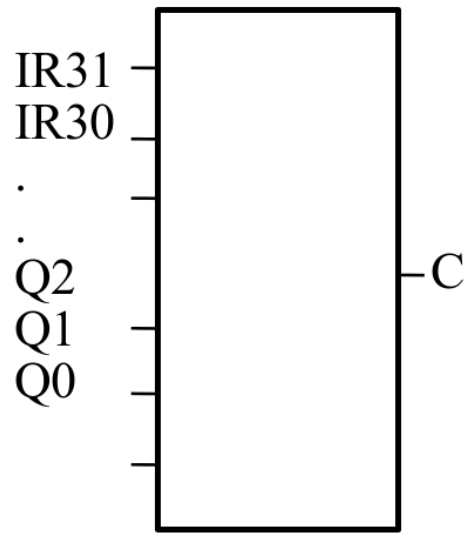
The controller's state sequence looks simple enough, but there is a problem:

What should the input signal(s) be?

Designing the controller

We need to design a combinatorial circuit with one output C. If C=1 we continue to the next execution state, but if C=0 we have finished and can fetch the next instruction. The inputs will be the instruction op-code bits and the state (F2, E1 etc.).

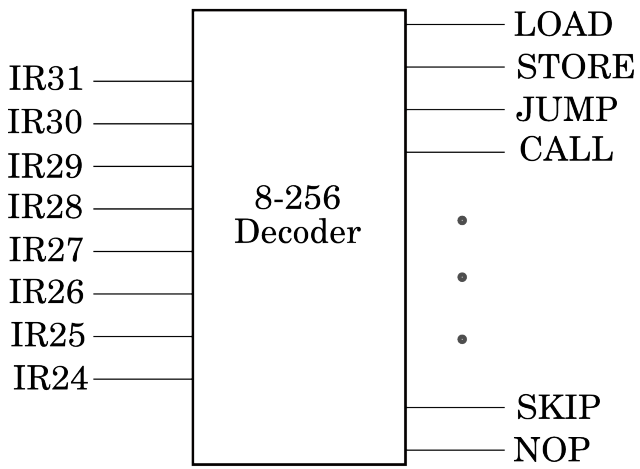




Decoders to the Rescue

We will approach this design problem functionally making use of decoders (aka de-multiplexers or binary to unary converters).

First an 8 to 256 decoder can be used decode the top eight bits of the IR - that is to say the opcode.



Only one output line is non zero for any input, and that indicates the instruction that is being executed.

Instructions with the same state sequence

Having decoded the instructions we can now consider them binary variables in our design.

Several instructions have the same state sequence (though different arithmetic functions). We can implement simple circuits for these from Boolean equations, eg:

$$\text{ADDS} = \text{ADD} + \text{SUBTRACT} + \text{AND} + \text{OR} + \text{XOR}$$

$$\text{SHIFTS} = \text{ASL} + \text{ASR} + \text{ROR}$$

$$\text{SKIPS} = \text{SKIP} + \text{SKIPPOSITIVE} + \text{SKIPNEGATIVE}$$

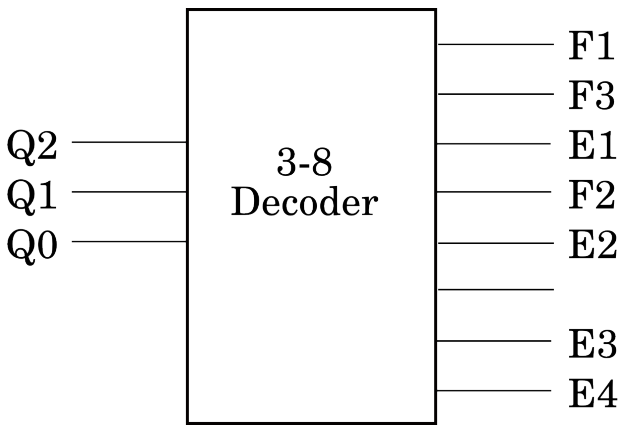
State assignments

We need to make state assignments for the fetch and execute states. Since there are seven states we will need three flip flops.

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	$Q2 \cdot Q1' \cdot Q0$

Decoders to the rescue (again)

Having defined how our states will be represented we can use a 3 to 8 decoder to give us one Boolean variable for each state.



We now have hardware lines that tell us both the state and the instruction or group of instructions being executed.

We can use all these as Boolean variables in our hardware design!

The C input to the finite state machine

- We can now simply write Boolean equations to define when the finite state machine needs to return to fetch a new instruction.
- For example we can go through our register transfer tables and find all the instructions that need exactly 2 execution cycles, and thus determine that the condition for returning from E2 is:

$$(E2 \cdot (RETURN + SHIFTS + MOVE + JUMPINDIRECT))'$$

The C input to the finite state machine

If we now repeat the process for each state where the state machine can branch back to state F1 we get the following Boolean equation:

$$C = (F3 \cdot NOP)' \cdot (E1 \cdot (SKIPS + CLEAR + JUMP))' \cdot (E2 \cdot (RETURN + SHIFTS + MOVE + JUMPINDIRECT)))' \cdot (E3 \cdot (COMP + DEC + INC + COMPARE + ADDS + STOREINDIRECT + LOAD))$$

With a bit of care we can implement this directly.

Designing the state sequencing logic

We are now in a position to design the state sequencing logic using the methodology we know and love!

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	×	×	×	×

Designing the state sequencing logic

From the table we get the following Karnaugh maps and minimised equations

	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

$$D2 = C \cdot Q2 \cdot Q1' + C \cdot Q1 \cdot Q0'$$
$$D1 = C \cdot Q1' + C \cdot Q2 \cdot Q0' + Q2' \cdot Q1' \cdot Q0'$$
$$D0 = Q2' \cdot Q1' \cdot Q0' + Q2' \cdot Q1 \cdot Q0 + C \cdot Q2 \cdot Q1 \cdot Q0'$$

Further Simplifications

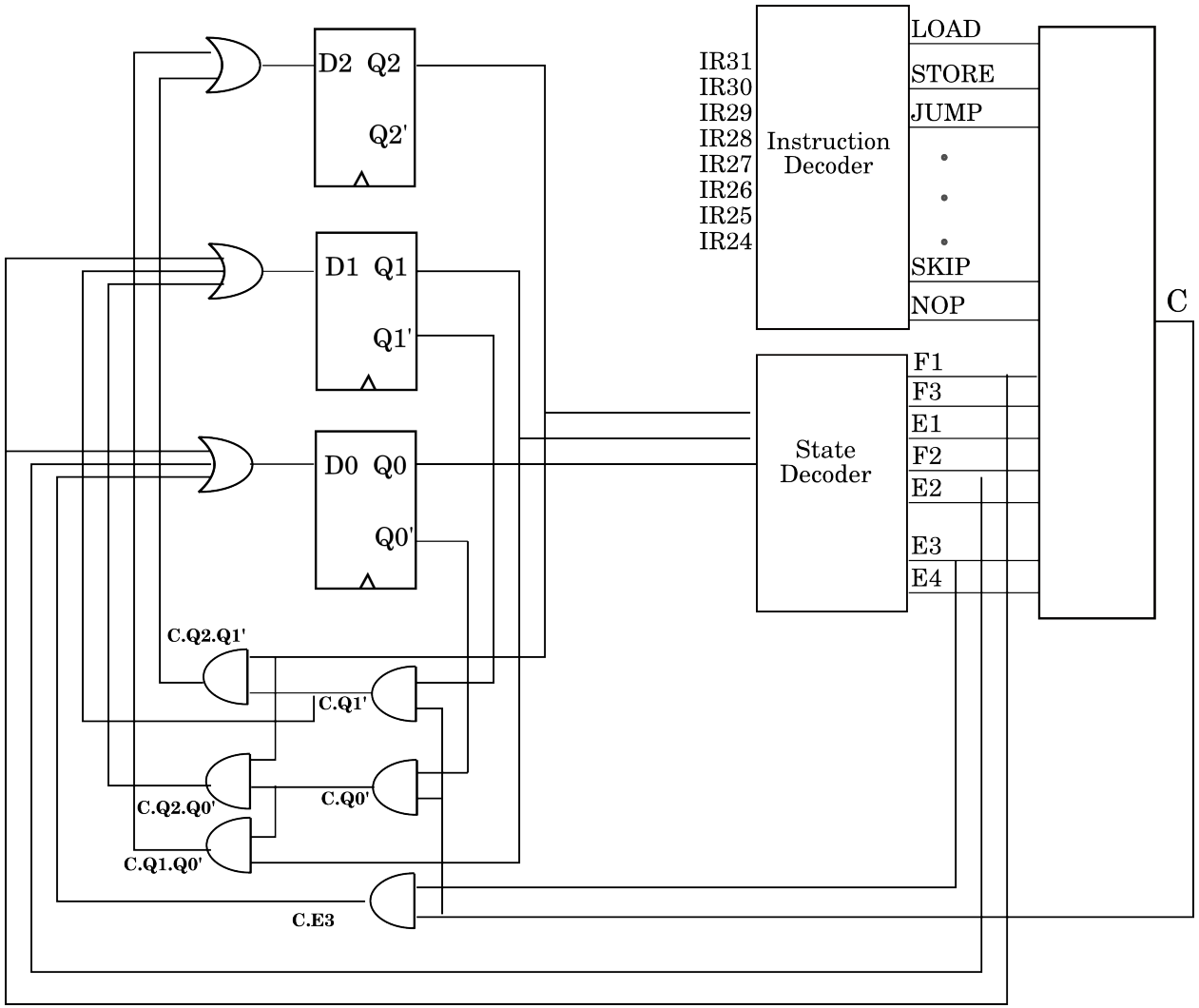
We could apply the XOR simplification rule to give us:

$$D0 = Q2' \cdot (Q1 \oplus Q0)' + C \cdot Q2 \cdot Q1 \cdot Q0'$$

But a better method is to make use of the fact that we have explicitly decoded the states:

$$D2 = C \cdot Q2 \cdot Q1' + C \cdot Q1 \cdot Q0'$$
$$D1 = C \cdot Q1 + C \cdot Q2 \cdot Q0 + F1$$
$$D0 = F1 + F2 + C \cdot E3$$

The final circuit is simpler than expected!



Checking the don't cares

We did not check whether the circuit will be safe at start up, but it is.

We will need to add extra hardware to make the processor do something particular at start up, (and maybe also on a signal from a reset button), so checking the don't cares is not a major issue.

The Output Logic

The output logic is a huge combinatorial design problem.

The inputs are the states (F1, F2 etc) and the instruction bits (at least the top 16).

The outputs are

- the clock controls (c0, c1, c2 etc)
- the arithmetic function select lines (f0, f1, etc) and
- the multiplexer select lines (s0, s1, etc).

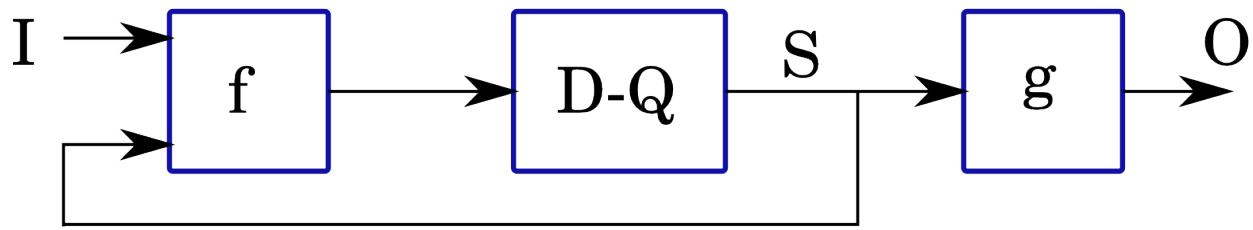
In total we have 28 different combinatorial design problems each with 17 inputs - clearly this is beyond the means of the Karnaugh map!

The Output Logic

Up until now we have been using the Moore finite state machine model for our design methodology

Recall that the Moore machine had no connection between the inputs and the output logic.

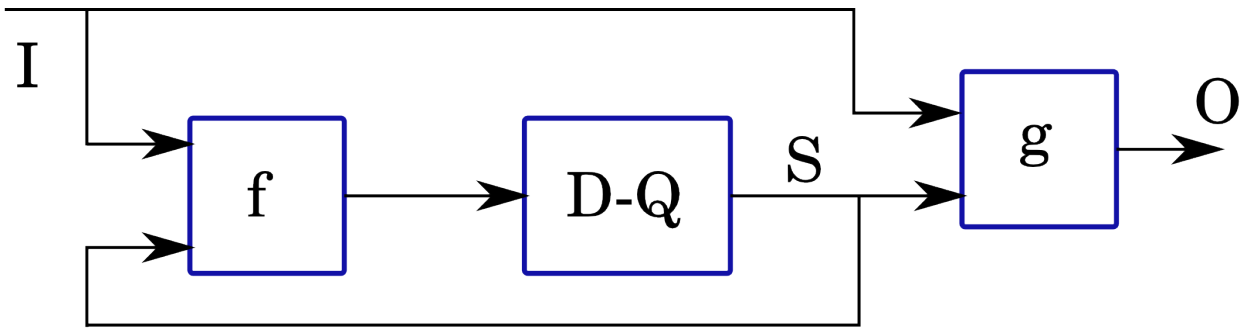
This is a safer design methodology since it makes the design more robust against spikes.



The Output Logic using a Mealey Machine

However, for the processor we use the Mealy machine where the inputs also go to the output logic.

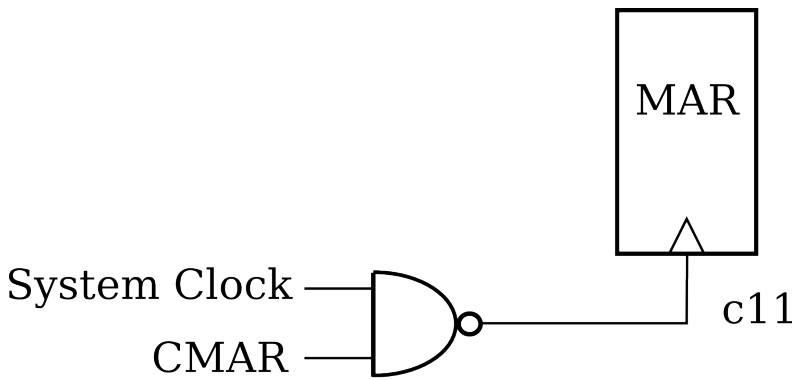
It is not possible to use the Moore machine since, in order to set up the processor correctly, we need to know which instruction is being executed.



Clock Gates

The clock gate signals c0 to c8 determine which register is loaded at each cycle.

The MAR will use this typical gating circuit:



Defining the CMAR signal

We proceed using the Boolean algebra to find the clock signals. Looking through the register transfer tables we find all the places where the MAR is to be set and derive the equation from these.

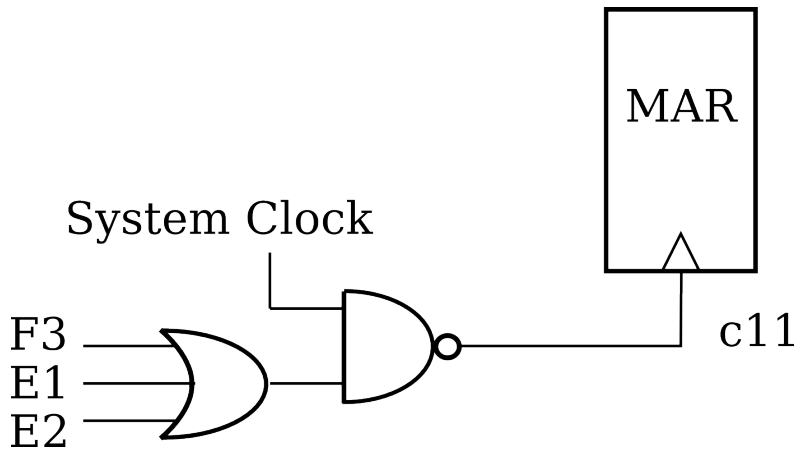
$$\begin{aligned} CMAR &= F1 + \\ &\quad E1 \cdot (LOAD + STORE) + \\ &\quad E2 \cdot (LOADINDIRECT + STOREINDIRECT) \end{aligned}$$

Don't cares about register contents

In practice we only need the MAR to be correct when we are about to load the MDR, so we can give it a clock pulse and load random data at other times without disturbing processor execution.

This allows us to simplify the equation for CMAR:

$$CMAR = F1 + E1 + E2$$



The MDR Clock

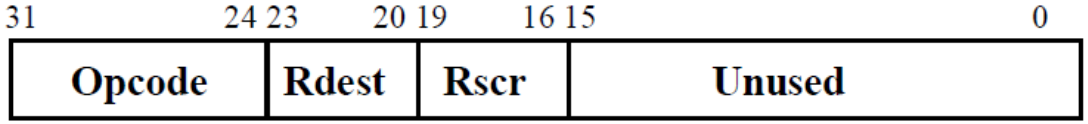
The same procedure is followed for many of the other register clocks. Looking at the MDR, from the register transfers we find:

$$CMDR = F2 + E2 \cdot LOAD + E3 \cdot LOADINDIRECT$$

The MDR (loaded in F2) is needed in cycle 3 by the CALL instruction, but only LOADINDIRECT uses it after E3, so we can simplify the equation to:

$$CMDR = F2 + E2 \cdot LOAD + E3$$

The Programmable Registers R0-R6



The register that changes (Rdest) is recorded in IR bits 22-20 in the present design. If we expanded the design to include 16 registers then bit 23 would be used as well. The condition that the destination register should change is:

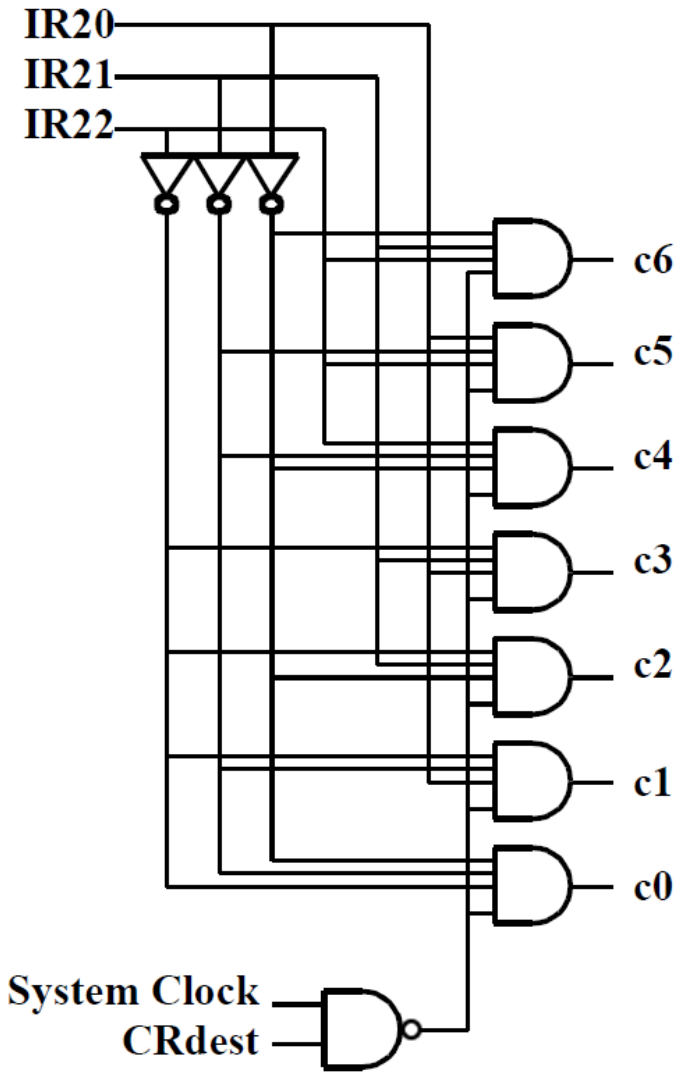
$$
\begin{aligned}
CRdest \quad = \quad & E4 + \\
& E3 \cdot (LOAD + ADDS + ONE) + \\
& E2 \cdot (SHIFTS + MOVE + CALL + CALLINDIRECT) + \\
& E1 \cdot CLEAR
\end{aligned}
$$

SHIFTS=ASL+ASR+ROR,
ONE=All one register instructions.

The Programmable Registers Clocks

A decoder is required to determine which register receives a clock pulse.

A four bit decoder would be required if we expanded the design to 16 registers



The Shifter function select bits

In this design we are using a simple four function shifter. In most cases where data is routed through the shifter we want "No Action" so we choose that to be the default.

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 the above table we can determine equations for the function select bits:

- f4 = ASR+ROR
- f3 = ASL+ROR

The ALU function select bits

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$\oplus$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

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 ALU carry in bit

The only place that a 1 carry is required in the current instruction set is $INC \cdot E3$

Its default will be 0

Thus

$$f5 = INC \cdot E3$$

Note that we connected the carry out bit as an input to the controller. We need to use it in our state sequencing logic for implementing SKIPPOSITIVE and SKIPNEGATIVE instructions.

The multiplexer selection bits

The selection inputs are defined by the following three tables.

Selection	s2	s1	s0
R0	0	0	0
R1	0	0	1
R2	0	1	0
R3	0	1	1
R4	1	0	0
R5	1	0	1
R6	1	1	0
Bus	1	1	1

Selection	s6	s5	s4
Shifter	0	0	0
ALU	0	0	1
PC	0	1	0
	0	1	1
Mask	1	0	0
MDR	1	0	1
	1	1	0
	1	1	1

Selection	s3
Bus	0
Incrementer	1

The Internal Bus Selector

First we need to look at the register transfer tables to determine when the different paths are selected. Using, for example, SPC to mean the condition when the PC is selected we find:

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

Where TWO is Boolean variables which becomes 1 when a two register instruction is being executed.

The shifter is the default selection.

The Internal Bus Selection Bits: S6, S5, S4.

Using the unallocated selections as don't cares we can write:

$$s4 = SALU + SMDR$$
$$s5 = SPC$$
$$s6 = SMask + SMDR$$

Selection	s6	s5	s4
Shifter	0	0	0
ALU	0	0	1
PC	0	1	0
	0	1	1
Mask	1	0	0
MDR	1	0	1
	1	1	0
	1	1	1

The register selector

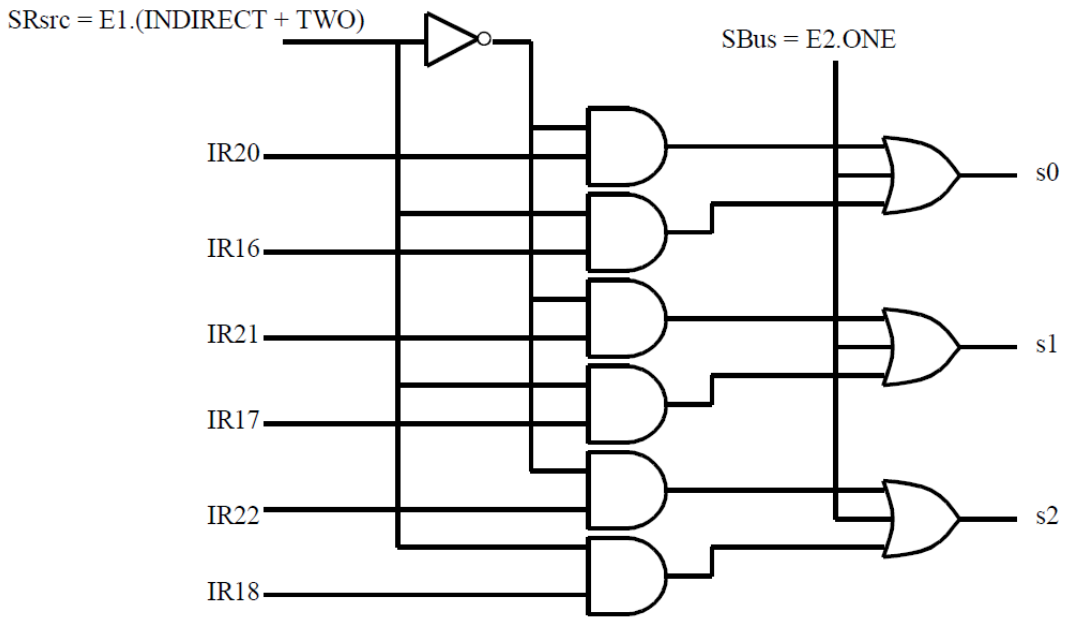
We can find the conditions defining the register selector from entries in the register transfer tables where A or B are loaded. Sometimes the register to be selected is the source (Rsrc: bits 19-16) sometimes it is the destination (Rdest: bits 23-20), sometimes the internal bus.

$$\begin{aligned} SRsrc &= E1 \cdot (INDIRECT + TWO) \\ SBus &= E2 \cdot ONE \\ SRdest &= (SRsrc + SBus)' \end{aligned}$$

INDIRECT, ONE, and TWO are Boolean variables indicating the instruction type.

Individual Register Selection

The individual selection is done by a multiplexer, with an additional set of gates to impose the Sbus condition.



The PC selector

Last but not least we can get the conditions for the PC to be connected to the internal bus from the register transfer tables:

$$s3 = F1 + E1 \cdot (CALL + CALLINDIRECT)$$

How did we do?

- We can now make a wiring list, buy the components from maplin and test it.
- The components will cost £200-£300 (over twice the price of a Intel Core 2.
- The clock could be set at about 10KHz (A bit faster if we fabricate it on a single chip)
- So it looks as if we had better consider the Mark 2 version straight away.

Improvements - The Instruction Formats

- All instructions are 32 bit, but mostly the bottom 16 bits are empty.
- This means that we are wasting memory space and doing many more fetch cycles than we need.
- We could pack up the instructions on byte boundaries and introduce some multiplexing hardware to load the IR correctly.

Improvements - Arithmetic

We have three unused inputs on the multiplexer that selects the internal bus.

Additional arithmetic hardware could include:

- A sixteen bit multiplier (multiply the bottom 16 bits of A and B to obtain a 32 bit result)
- An incrementer
- A decrementer

Improvements - The Data Paths

- Additional multiplexers could help us to reduce the instruction cycles of many instructions. For instance a multiplexer to select the input to B independently of A would reduce many three cycle instructions to two cycles.
- A data path from the registers to the internal bus would reduce some instructions by one cycle.
- This would require an additional input on the bus selector multiplexer, and so might be considered an alternative to the additional arithmetic functions already discussed.

Improvements - The Combinatorial Circuits

- This is the hard part.
- We want to have the minimum time delays in all our combinational logic.
- This is partly a question of path length, but does require looking at low level transistor models to calculate the time accurately

And that is the end of the course

And that is the end of the course

- well nearly!

And that is the end of the course

- well nearly!

Coursework 2 takes the place of the first lab exercise of next term.

You can do it as soon as you like. The instruction sheets are on the course website.

And that is the end of the course

- well nearly!

Coursework 2 takes the place of the first lab exercise of next term.

You can do it as soon as you like. The instruction sheets are on the course website.

There will be a revision session before the exam at the start of the summer term. Watch the course website for the time and venue.

And that is the end of the course

- in the meantime

And that is the end of the course

- in the meantime

Have a great Christmas break