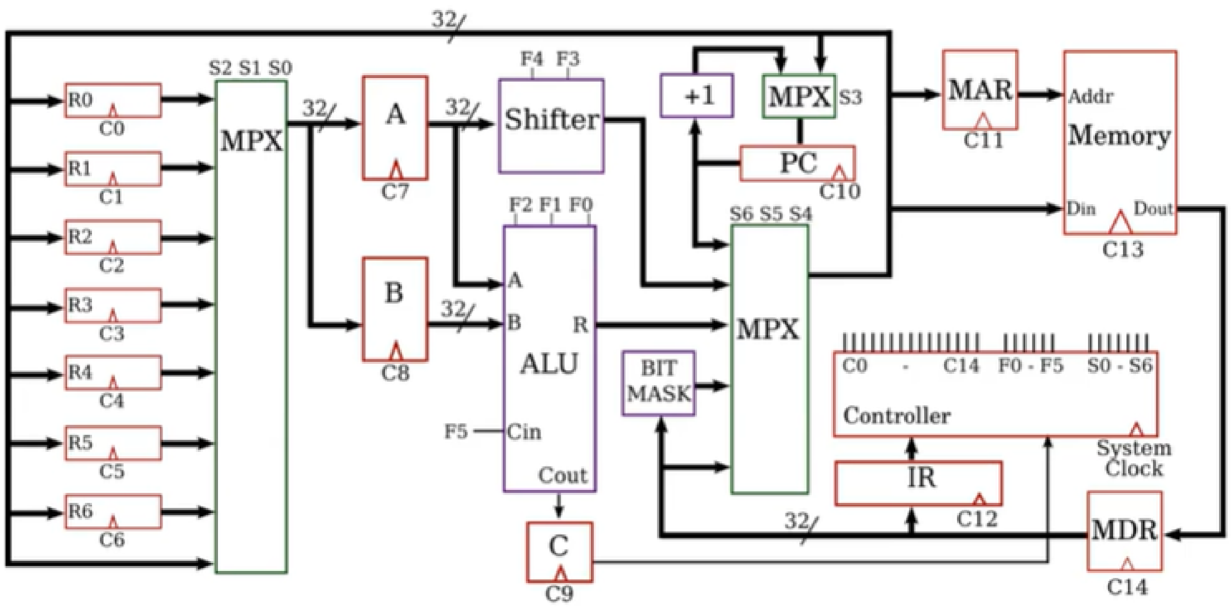
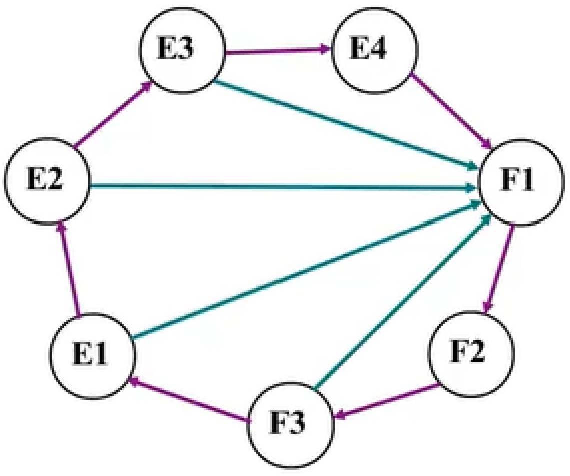
L19 - Designing a Central Processor Unit 2 The Controller

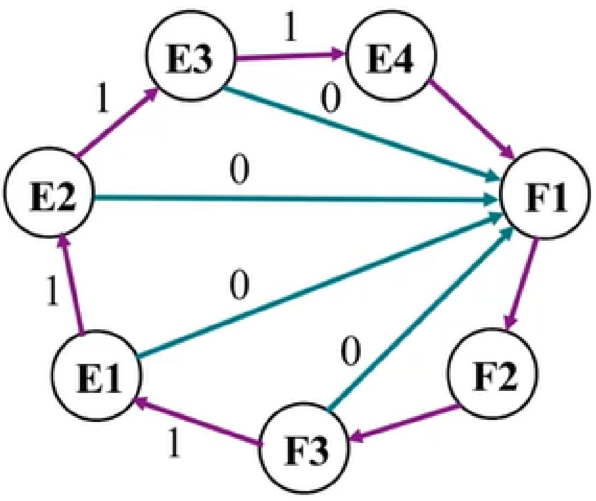


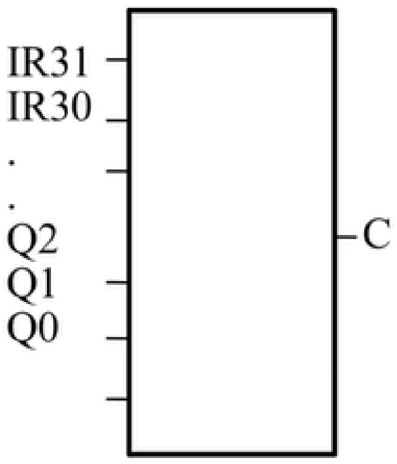
Design the controller



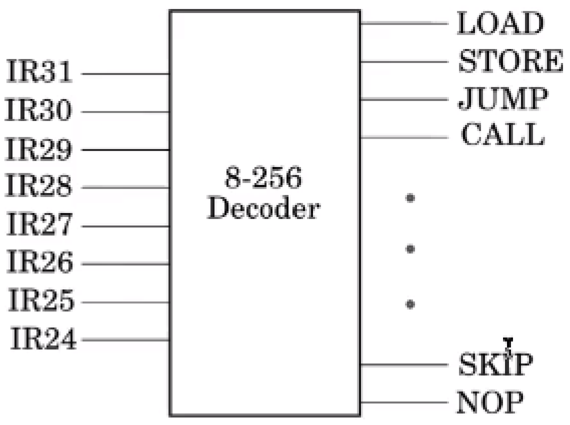
Looks simple. But, what should the input signals be?

Design a combinatorial circuit with one output C. If C = 1, we continue to the next execution state. 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).





1. Use an 8 to 256 decoder to decode the top 8 bits of the IR - the opcode; only one output line is 1 for any input, and that indicates the instruction that is being executed



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, e.g.

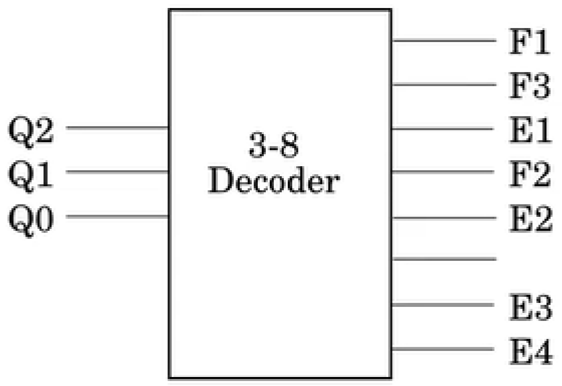
$$\text{ADDS} = \text{ADD} + \text{SUBTRACT} + \text{AND} + \text{OR} + \text{XOR}$$
$$\text{SHIFTS} = \text{ASL} + \text{ASR} + \text{ROR}$$
$$\text{SKIPS} = \text{SKIP} + \text{SKIPPOSITIVE} + \text{SKIPNEGATIVE}$$

State assignments

Make SA for F and E states. There are 7 states so we will need 3 FFs.

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	$Q2 \cdot Q1' \cdot Q0$

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 write Boolean equations to define when the finite state machine needs to return to fetch a new instruction. E.g. 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:

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

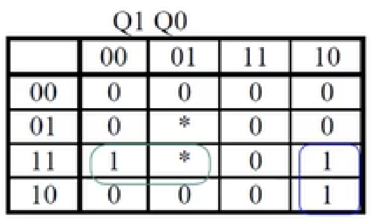
Repeat this process for each state where the state machine can branch back to state F1 to get the following BE:

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

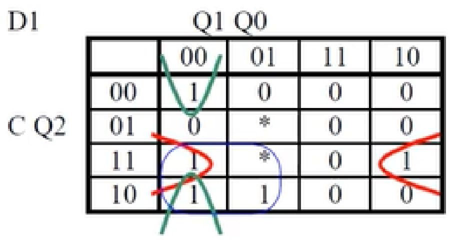
Design State Sequencing Logic

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	×	×	×	×

Convert to K Maps and B equations

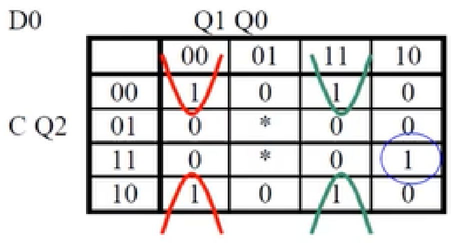


	Q1 Q0			
	00	01	11	10
00	0	0	0	0
01	0	*	0	0
11	1	*	0	1
10	0	0	0	1



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



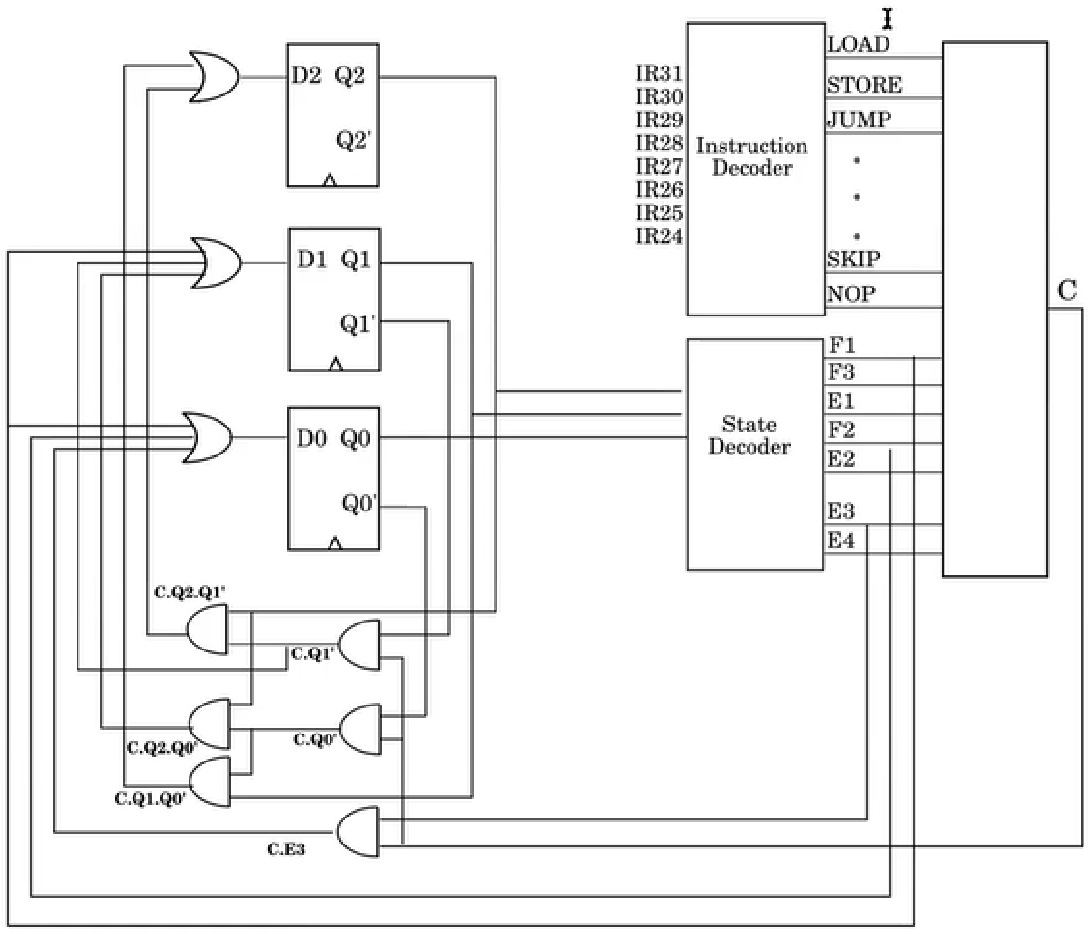
D0

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

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

Simplify by using the fact that we have decoded the states:

$$D2 = C \cdot Q2 \cdot Q1' + C \cdot Q1 \cdot Q0'$$
$$D1 = C \cdot Q1 + C \cdot Q2 \cdot Q0 + F1$$
$$D0 = F1 + F2 + C \cdot F3$$



Need to check the DC, so that the circuit will be safe at start up, but in this case, it works.

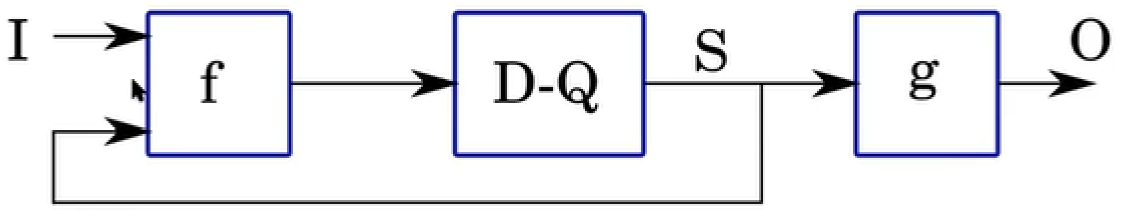
We will need to add extra hardware to make the processor do something particular at start up, (and maybe also on a signal from a reset button), so checking the don't cares is not a major issue.

The Output Logic
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:

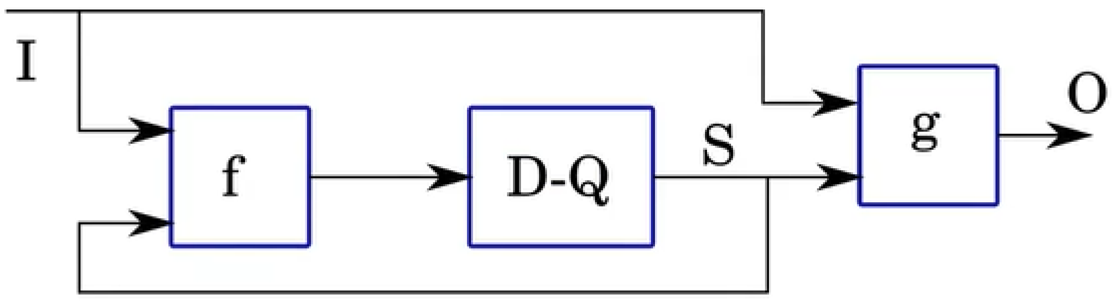
1. The clock controls (C0, C1, C2, etc)
2. The arithmetic function select lines (F0, F1, etc) and
3. The multiplexer select lines (S0, S1, etc)

In total, we have 28 different combinatorial design problems each with 17 inputs, beyond the means of the K Map!

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

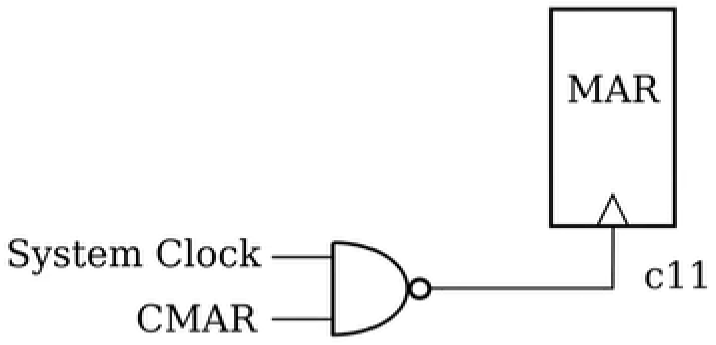


However, for the processor, we will use the Mealey machine where the inputs also go to the output logic. It is not possible to use the Moore machine since, in order to set up the processor correctly, we need to know which instruction is being executed.



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:



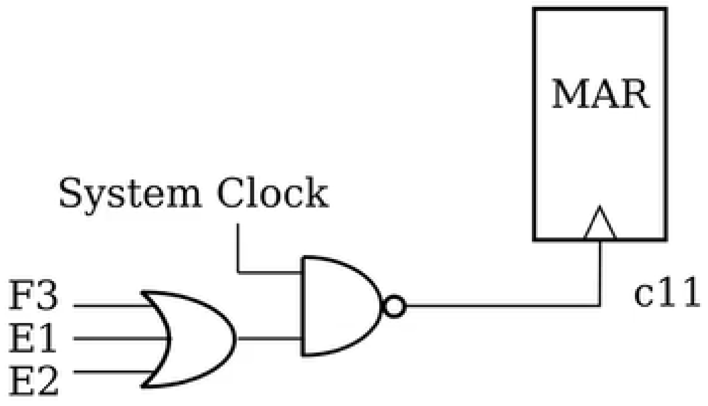
Defining CMAR signal

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.

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

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:

$$CMAR = F1 + E1 + E2$$



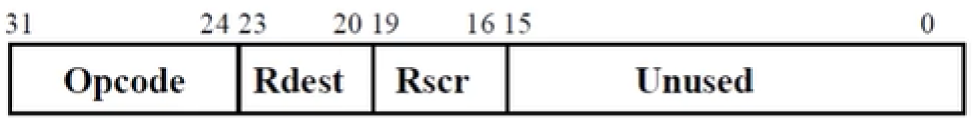
MDR Clock
The same procedure is followed for many of the other register clocks. Looking at the MDR, from the register transfers, we find:

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

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



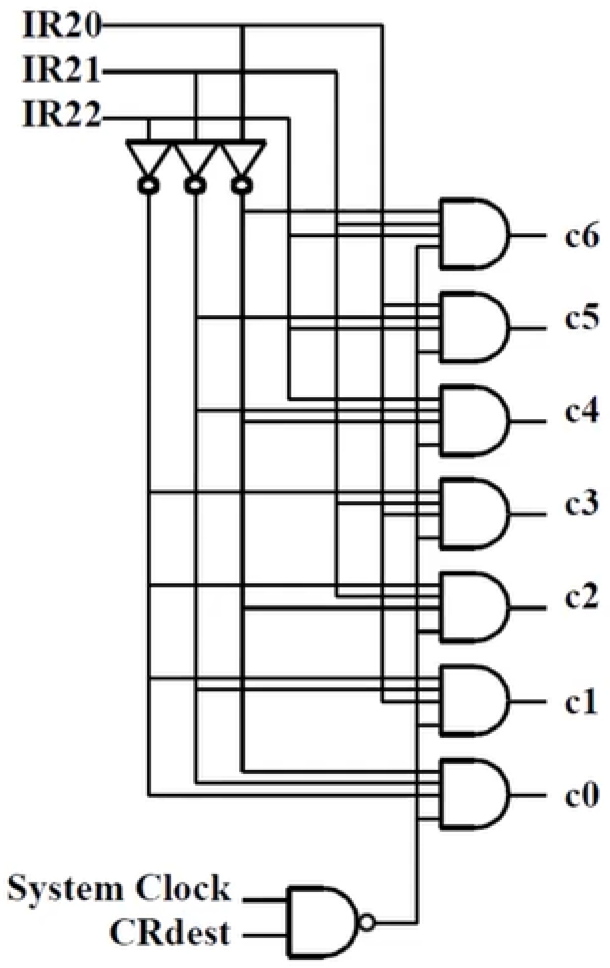
The register that changes (Rdest) is recorded in IR bits 22-20 in the present design. If we expanded the design to include 16 registers then bit 23 would be used as well. The condition that the destination register should change is:

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

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 4-bit decoder would be required if we expanded the design to 16 registers.



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

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)

ALU carry in bit

The only place that a 1 carry is required in the current instruction set is (INC.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.

Multiplexer Selection bits

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		

The Internal Bus Selector
First, we need to look at the register transfer tables to determine when the different paths are selected. Using, e.g., SPC to mean the condition when the PC is selected we find:

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

Where TWO is a Boolean variable which becomes 1 when a two register instruction is being executed. The shifter is the default selection.
Using the unallocated selections as DC we can write:

$$
\begin{aligned}
s4 &= SALU + SMDR \\
s5 &= SPC \\
s6 &= SMask + SMDR
\end{aligned}
$$

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

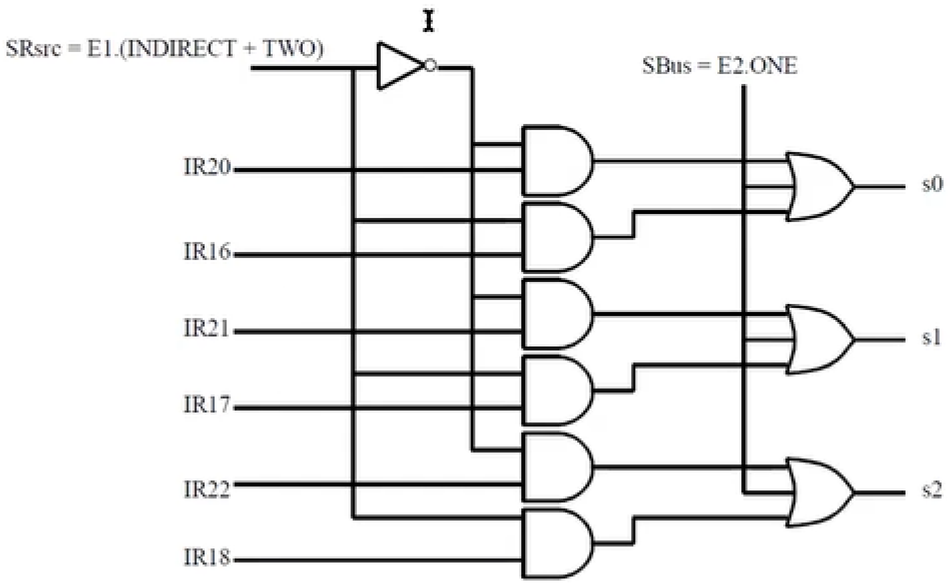
The Register Selector
We can find the conditions defining the register selector from entries in the register transfer tables where A or B are loaded. Sometimes the register to be selected is the source (Rsrc: bits 19-16) sometimes it is the destination (Rdest: bits 23-20), sometimes the internal bus.

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

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.



PC Selector

Conditions for the PC to be connected to the internal bus from the register transfer tables:

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

We are done!

However, it will be expensive and very slow.

Improvements - The instruction Formats

1. All instructions are 32 bit, but mostly the bottom 16 bits are empty.
2. This means that we are wasting memory space and doing many more fetch cycles than we need
3. 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:

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

3. A decrementer

Improvements - The Data Paths

1. 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 2 cycles.
2. A data path from the registers to the internal bus would reduce some instructions by one cycle
3. 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 hard.
We want to have the minimum time delays in all our combinatorial logic. This is partly a question of path length, but does require looking at low level transistor models to calculate the time accurately.