Associative	$(A \cdot B) \cdot C = A \cdot (B \cdot C)$ $(A + B) + C = A + (B + C)$
Commutative	$A \cdot B = B \cdot A$ $A + B = B + A$
Distributive	$A \cdot (B + C) = A \cdot B + A \cdot C$ $A + (B \cdot C) = (A + B) \cdot (A + C)$

$$(A + B + C)' = ((A + B) + C)' = ((A + B)') \cdot C' = A' \cdot B' \cdot C'$$

$$A + A \cdot (B) = A$$

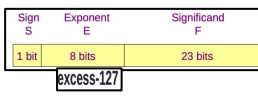
$$A \oplus B = A \cdot \overline{B} + \overline{A} \cdot B$$

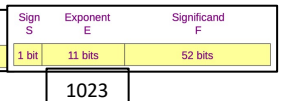
$$A \cdot A = A$$
$$A + A = A$$

$$A \cdot 0 = 0$$
$$A \cdot 1 = A$$
$$A + 0 = A$$
$$A + 1 = 1$$

Operation	Result
N ÷ ± Infinity	0
± Infinity × ± Infinity	± Infinity
± non-zero ÷ 0	± Infinity
Infinity + Infinity	Infinity
± 0 ÷ ± 0	*NaN*
Infinity - Infinity	*NaN*
± Infinity ÷ ± Infinity	*NaN*
± Infinity × 0	*NaN*

IEEE value	Sign field	Exponent	Significand	True exponent
± 0	0 or 1	0	(all zeros)	
± denormalised no.	0 or 1	0	Any non-zero bit pattern	-126
± normalised no.	0 or 1	1...254	Any bit pattern	-126...127
± ∞	0 or 1	255	0 (all zeros)	
Not-a-number	0 or 1	255	Any non-zero bit pattern	

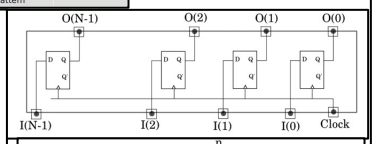


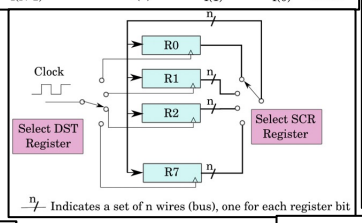


Problem 2. Result = (Number1 plus Number2) div 2

State 1: Load the instruction register
State 2: Load Number 1 into A
State 3: Load Number 2 into B
State 4: Load the instruction register
State 5: Clear C by setting the ALU to zero
State 1: Load the instruction register
State 2: ALU: A plus B (Cin=0); load the result into A
State 3: B and C are loaded but we don't care what with!
State 4: Load the instruction register
State 5: Set the shifter to logical shift right, load RES from shifter

State	Data In	Instruction register (IR)							
		IR7	IR6	IR5	IR4	IR3	IR2	IR1	IR0
1	xxxxxxx1	X	X	X	X	X	X	X	X
2	Number 1	X	X	X	X	X	X	X	1
3	Number 2	X	X	X	X	X	X	X	1
	x000xxxx	X	X	X	X	X	X	X	1
	xxxxxxxx	X	0	0	0	X	X	X	X
	x011x110	X	0	0	0	X	X	X	X
	xxxxxxxx	X	0	1	1	X	1	1	0
	xxxxxxxx	X	0	1	1	X	1	1	0
	x110x0xx	X	0	1	1	X	1	1	0
	xxxxxxxx	X	1	1	0	X	0	X	X





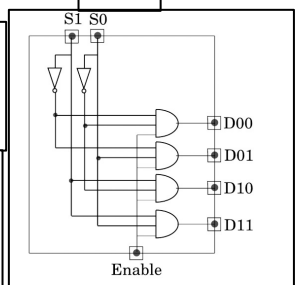
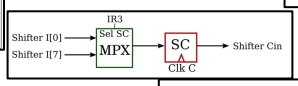


Figure 3: 2 to 4 Decoder

Instruction	Cycle	Transfers	Path
LOAD Rdest, Address	E1	MAR←MDR	Use the bit mask
	E2	MDR←Memory	
	E3	Rdest←MDR	No masking
STORE Rdest, Address	E1	MAR←MDR; A←Rdest	Use the bit mask
	E2	Memory←A	via the Shifter (no change)
JUMP Address	E1	PC←MDR	Use the bit mask
CALL Rdest, Address	E1	PC←PC+1	
	E2	Rdest←PC	
	E3	PC←MDR	Use the bit mask



Instruction	Cycle	Transfers	Path
LOADINDIRCET Rdest, Rscr	E1	A←Rsrc	
	E2	MAR←A	
	E3	MDR←Memory	
	E4	Rdest←MDR	No masking
STOREINDIRECT Rdest, Rscr	E1	A←Rscr	
	E2	MAR←A;A←Rdest	via the shifter (no change)
	E3	Memory←A	via the shifter (no change)
JUMPINDIRECT Rscr	E1	A←Rscr	
	E2	PC←A	via the shifter (no change)
CALLINDIRECT Rdest, Rscr	E1	PC←PC+1;A←Rscr	
	E2	Rdest←PC;	
	E3	PC←A	via the shifter (no change)

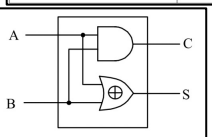


Figure 1: One bit half adder

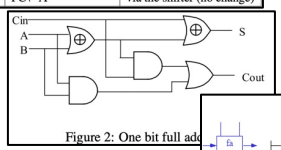
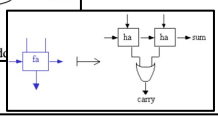


Figure 2: One bit full adder



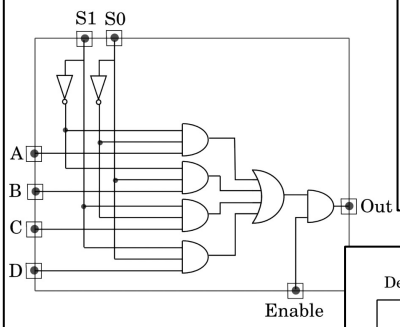


Figure 1: 4 to 1 Multiplexer

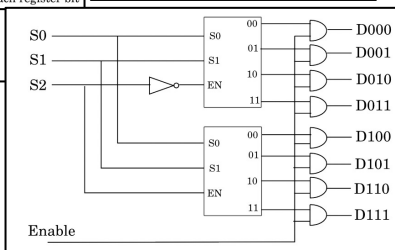


Figure 4: 3 to 8 Decoder

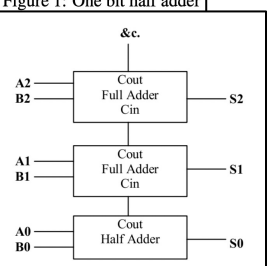


Figure 3: N-bit full adder

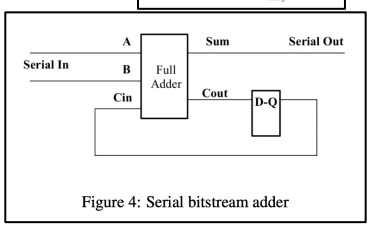


Figure 4: Serial bitstream adder

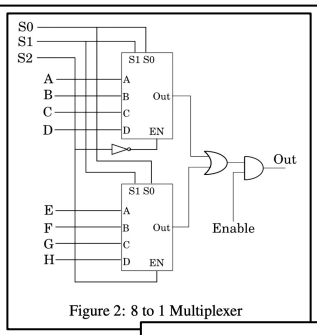
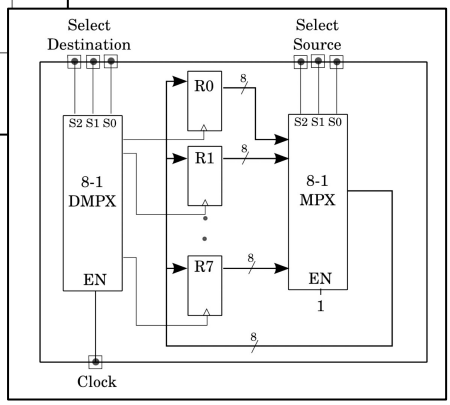


Figure 2: 8 to 1 Multiplexer



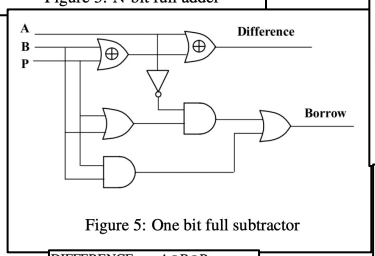


Figure 5: One bit full subtractor

DIFFERENCE	= A⊕B⊕P
BORROW	= A'·(B + P) + B·P

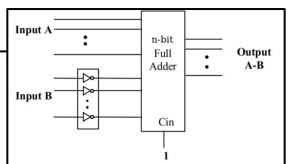
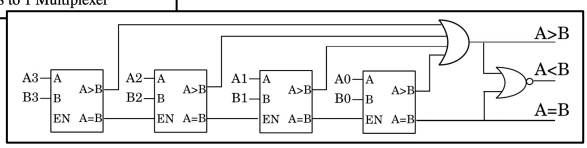
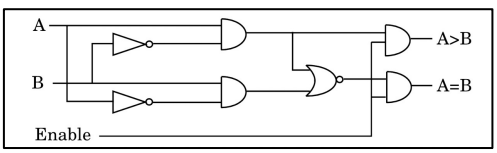


Figure 6: Twos Complement Subtractor





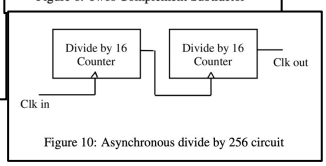


Figure 10: Asynchronous divide by 256 circuit

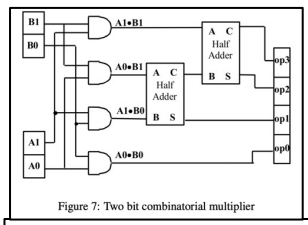


Figure 7: Two bit combinatorial multiplier

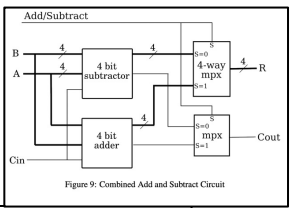


Figure 9: Combined Add and Subtract Circuit

Two Register Instructions

Instruction	Cycle	Transfers	Path
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

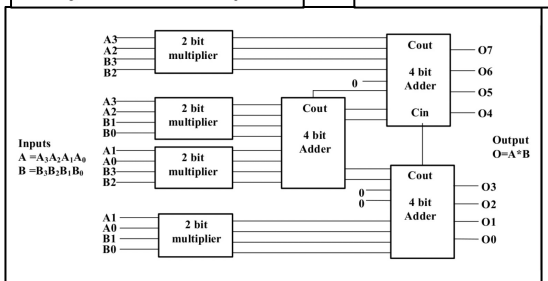


Figure 8: Four bit combinatorial multiplier

One Register instructions

Instruction	Cycle	Transfers	Path
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 eor 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

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

Instruction	Cycle	Transfers	Path
SKIP	E1	PC←PC+1	

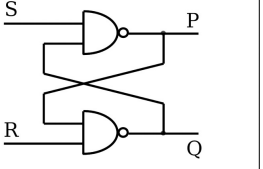


Figure 1: The Set-Reset Flip Flop

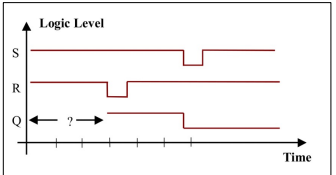


Figure 2: The Set-Reset Flip Flop Timing

NOW				NEXT		
S	R	P_t	Q_t	P_{t+1}	Q_{t+1}	
1	1	0	0	1	1	Unstable
1	1	0	1	0	1	Stable
1	1	1	0	1	0	Stable
1	1	1	1	0	0	Unstable

Combinatorial: output is a combination of **only inputs**

Sequential: output is a combination of **inputs and previous outputs** (we need a transition table or state diagram, it needs some notion of time)

Asynchronous: output will have correct value after an **undefined** amount of time

Synchronous: output will have correct value after a **defined** amount of time (a clock signal tells us time)

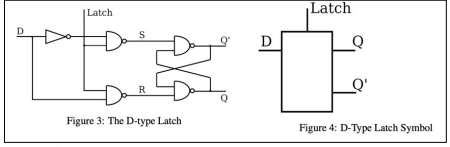


Figure 3: The D-type Latch

Figure 4: D-Type Latch Symbol

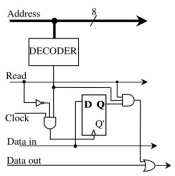


Figure 1: One bit of static RAM

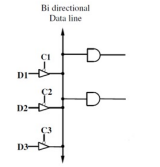
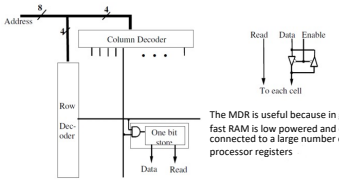
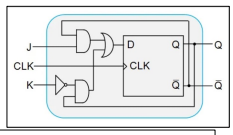


Figure 2: Tri-State Buffering



The MDR is useful because in general fast RAM is low powered and cannot be connected to a large number of processor registers

Figure 3: Layout of RAM



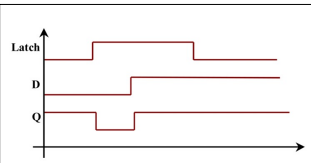


Figure 5: A Spike in the D-Type Latch

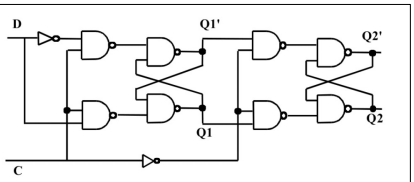
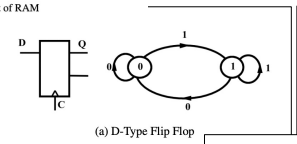
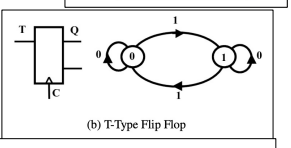


Figure 6: The master-slave D-Type flip flop

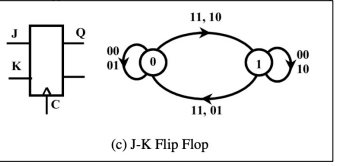


(a) D-Type Flip Flop



(b) T-Type Flip Flop

J	K	Function	Output $Q_s(t_{n+1})$
0	0	No Change	$Q_s(t_n)$
0	1	Reset	0
1	0	Set	1
1	1	Toggle	$Q'_s(t_n)$



(c) J-K Flip Flop

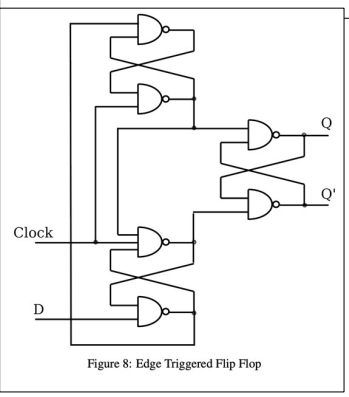


Figure 8: Edge Triggered Flip Flop

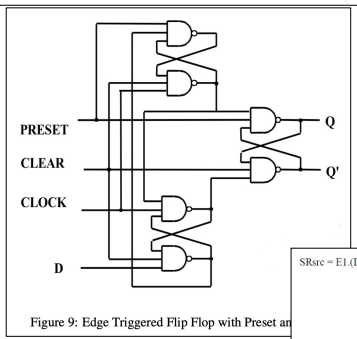
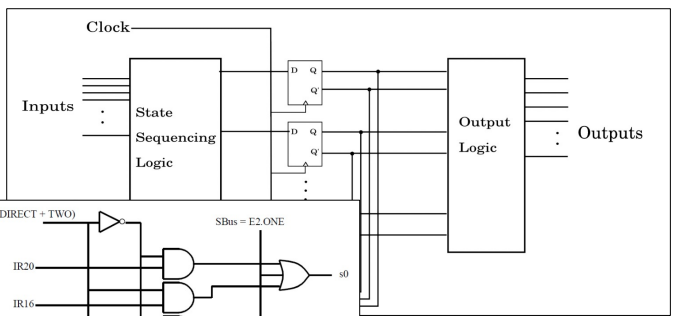
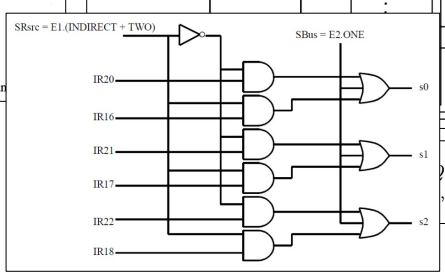


Figure 9: Edge Triggered Flip Flop with Preset an





The D type flip flop law

State Sequencing Logic

$Q_2(t_n), \ldots)$

$, Q_2(t_n), \ldots)$

Output Logic

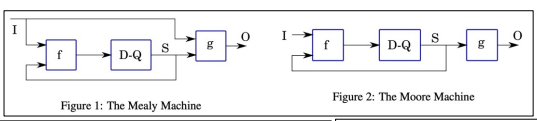


Figure 1: The Mealy Machine

Figure 2: The Moore Machine

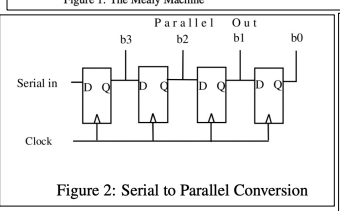


Figure 2: Serial to Parallel Conversion

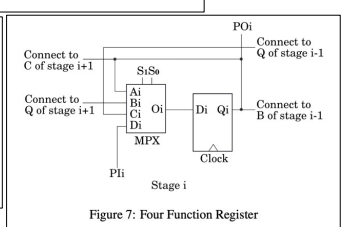
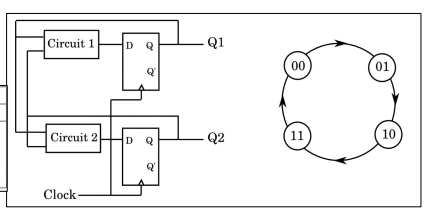


Figure 7: Four Function Register

Inputs (t_n)		Outputs (t_{n+1})	
Q_1	Q_2	D_1	D_2
0	0	0	1
0	1	1	0
1	0	1	1
1	1	0	0



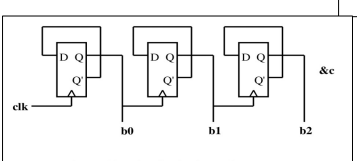


Figure 11: The ripple through counter

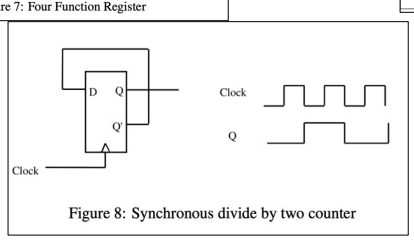


Figure 8: Synchronous divide by two counter

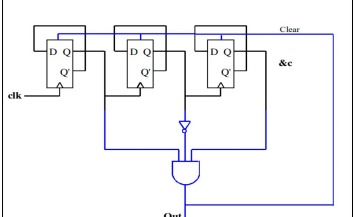
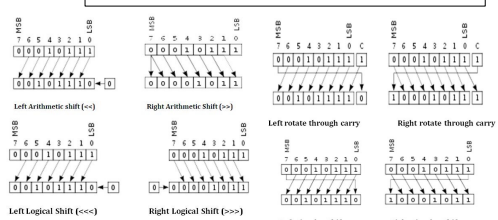
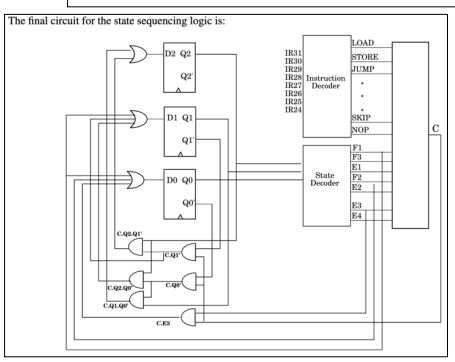
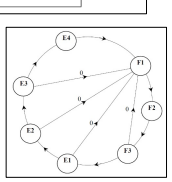


Figure 12: Count to five circuit







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.

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.

Asynchronous sequential circuits do not have a global clock signal to control the timing of operations. Instead, the state of the circuit changes in response to changes in inputs. Due to the differing propagation delays in different paths of the circuit, when an input changes, some parts of the circuit can respond more quickly than others, leading to a temporary inconsistency in the output, known as a hazard or a glitch

Synchronous sequential circuits use a global clock signal to control the timing of the operations.

Setup time: before the clock edge during which the data must be stable

Hold time: after the clock edge during which the data must remain stable.

If these timing constraints are violated, the output of the flip-flop can momentarily go to an undefined state, causing a spike.

if different flip-flops are driven by the same clock but have different propagation delays due to different paths, the signals at the outputs can arrive at different times, which may create a spike if these outputs are combined

Combinatoral Logic Spike

During a change to the input, different paths in the circuit have different propagation delays, and the outputs from these paths will change at slightly different times. If these outputs are combined further, there could be a moment when the final output is some incorrect intermediate value

F[2]	F[1]	F[0]	Shift	Carry	Function
0	0	0			unchanged
0	0	1	left		rotate left
0	1	0	left	0	arithmetic left shift
0	1	1	left	Cin	left shift with carry
1	0	0	right		rotate right
1	0	1	right	0	logical right shift
1	1	0	right	Input[7]	arithmetic right shift
1	1	1	right	Cin	shift right with carry