S1.

a An N-type MOSFET is N-type semiconductor terminals and P-type body: by switching the type of semiconductor used for terminal and body, we get a P-type MOSFET. Therefore the first statement is correct, and the second incorrect. Both types of MOSFET depend on the concept of a threshold voltage: the idea in both cases is that this is the potential difference between gate and source which produces a depletion region matching the size required. Therefore the third statement is correct. Finally, a CMOS cell is a pairing of two transistors; however, the design depends on complementary types (i.e., one N-type and one P-type). Therefore the fourth statement is incorrect, although slightly subtle.

b A buffer is a simple NOP-like logic gate, and a ripple-carry adder is a combinatorial circuit that continuously produces an output based on the inputs. However, both a D-type flip-flop and a clock would more typically be used in a clocked circuit design, e.g., a state machine which uses the flip-flop to remember the state it is in and the clock to synchronise state transitions: these would not typically be used in a combinatorial circuit design.

c Using a Karnaugh map, for example, one can produce the result

$$r = f(x, y, z) = y \vee (z \wedge \neg x) \vee (x \wedge \neg z)$$

which, by inspection, gives

			f			
x	y	z	y	$z \wedge \neg x$	$x \wedge \neg z$	r
0	0	0	0	0	0	0
0	0	1	0	1	0	1
0	1	0	1	0	0	1
0	1	1	1	1	0	1
1	0	0	0	0	1	1
1	0	1	0	0	0	0
1	1	0	1	0	1	1
1	1	1	1	0	0	1

However, notice that the sub-expression

$$(z \wedge \neg x) \vee (x \wedge \neg z)$$

can be simplified to

$$z \oplus x$$

so per the question, the simplest implementation of f is

$$r = f(x, y, z) = y \vee (z \oplus x).$$

d First, note that the following identities can be applied:

$$\begin{array}{lcl} \neg x & \equiv & x \mathbin{\overline{\wedge}} x \\ x \vee y & \equiv & (x \mathbin{\overline{\wedge}} x) \mathbin{\overline{\wedge}} (y \mathbin{\overline{\wedge}} y) \\ x \wedge y & \equiv & (x \mathbin{\overline{\wedge}} y) \mathbin{\overline{\wedge}} (x \mathbin{\overline{\wedge}} y) \end{array}$$

As such, we can write

$$\neg(x \vee y) \equiv ((x \mathbin{\overline{\wedge}} x) \mathbin{\overline{\wedge}} (y \mathbin{\overline{\wedge}} y)) \mathbin{\overline{\wedge}} ((x \mathbin{\overline{\wedge}} x) \mathbin{\overline{\wedge}} (y \mathbin{\overline{\wedge}} y)).$$

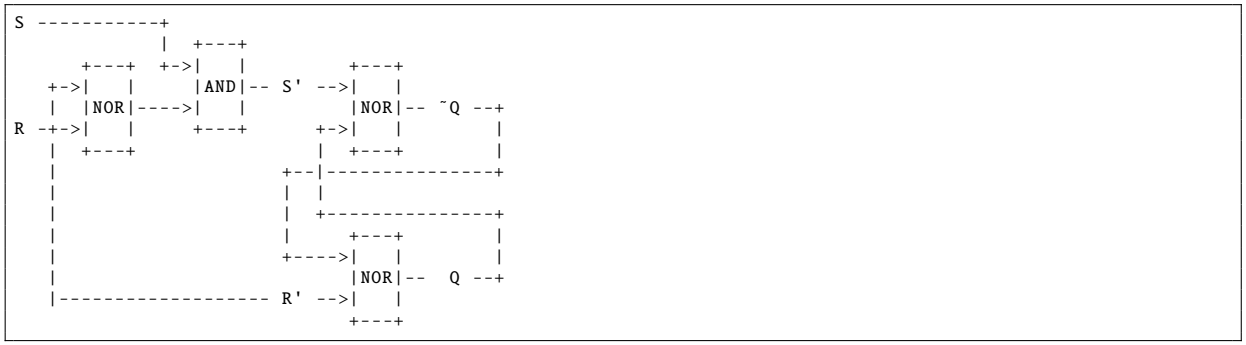
e The excitation table of a standard SR latch is

		Current		*Next*	
S	R	Q	$\neg Q$	Q'	$\neg Q'$
0	0	0	1	0	1
0	0	1	0	1	0
0	1	**?**	**?**	0	1
1	0	**?**	**?**	1	0
1	1	**?**	**?**	**?**	**?**

meaning the reset-dominate alteration gives

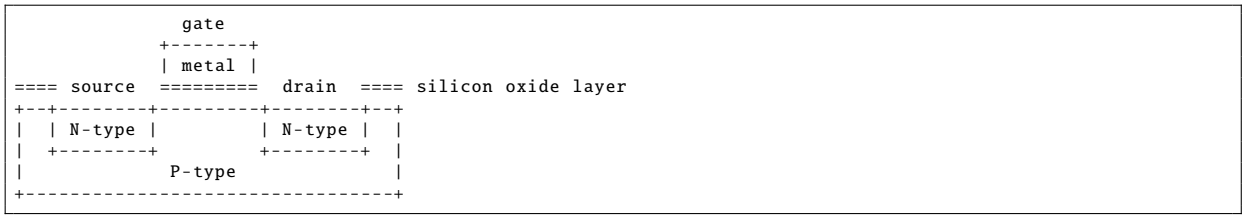
		Current		*Next*	
S	R	Q	$\neg Q$	Q'	$\neg Q'$
0	0	0	1	0	1
0	0	1	0	1	0
0	1	**?**	**?**	0	1
1	0	**?**	**?**	1	0
1	1	**?**	**?**	0	1

Given the inputs S and R, the circuit can be constructed by using a standard SR latch with two cross-coupled NOR gates whose inputs are S' and R'. We simply set $R' = R$ and $S' = \neg R \wedge S$ so S' is only 1 when $R = 0$ and $S = 1$: if $R = 1$ (including the case when both $R = 1$ and $S = 1$), the latch it reset since $S = 0$. The circuit is as follows:



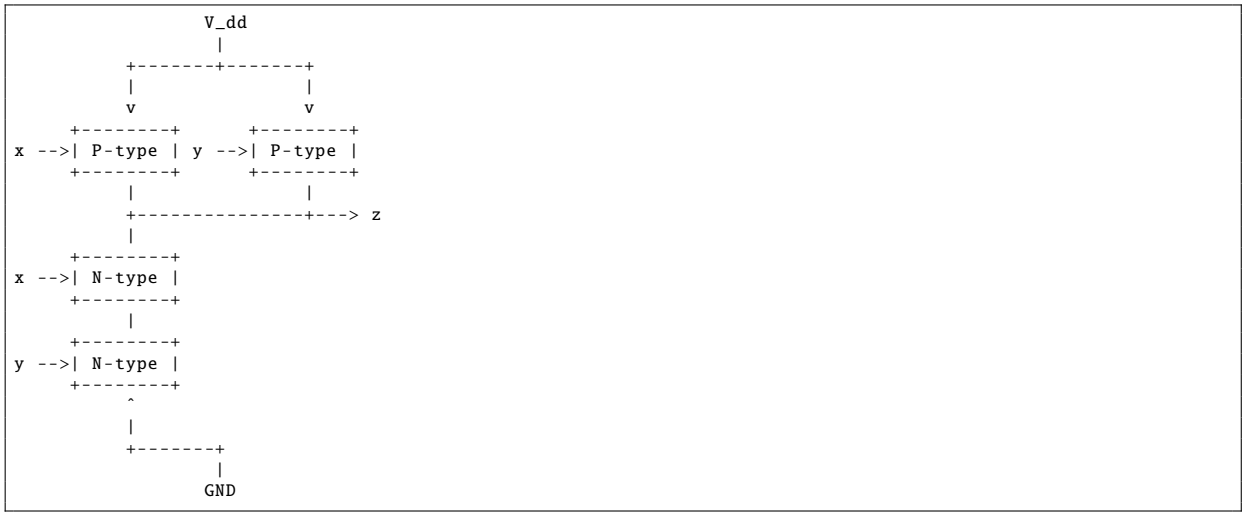
f A wide range of answers are clearly possible. Obvious examples include physical size, and power consumption or heat dissipation. Other variants include worst-case versus average-case versions of each metric, for example in the case of efficiency.

S2. a MOSFET transistors work by sandwiching together N-type and P-type semiconductor layers. The different types of layer are doped with different substances to create more holes or more electrons. For example, in an N-type MOSFET the layers are constructed as follows



with additional layers of silicon oxide and metal. There are three terminals on the transistor. Roughly speaking, applying a voltage to the gate creates a channel between the source and drain through which charge can flow. Thus the device acts like a switch: when the gate voltage is high, there is a flow of charge but when it is low there is little flow of charge. A P-type MOSFET swaps the roles of N-type and P-type semiconductor and hence implements the opposite switching behaviour.

b One can construct an NAND to compute $z = x \overline{\wedge} y$ gate from such transistors as follows:

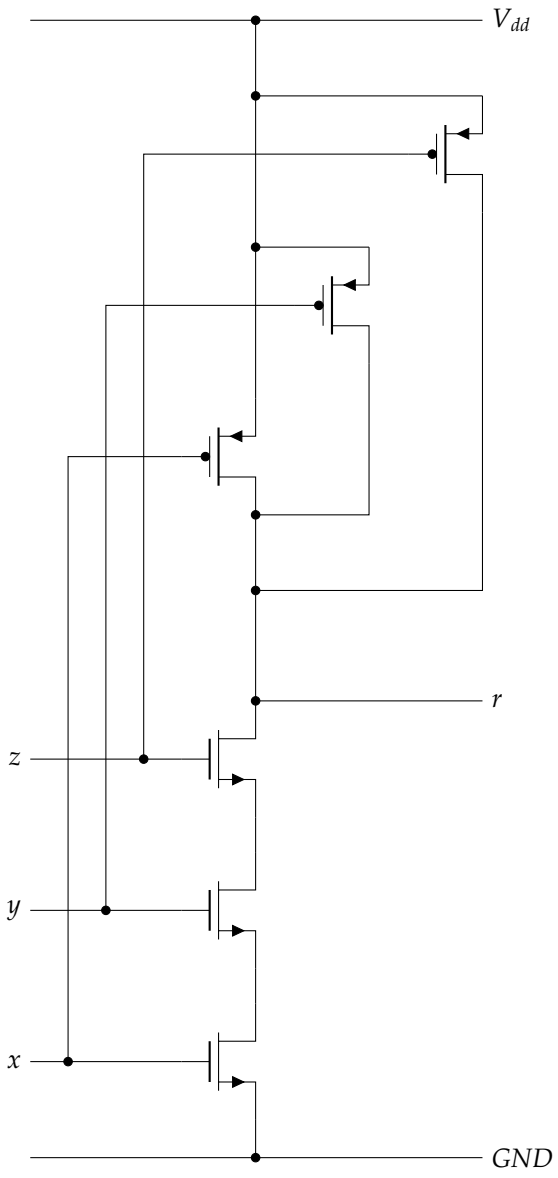


If x and y are connected to *GND* then both top P-type transistors will be connected, and both bottom N-type transistors will be disconnected; r will be connected to V_{dd}. If x and y are connected to V_{dd} and *GND* respectively then the right-most P-type transistor will be connected, and both lower-most N-type transistor will be disconnected; r will be connected to V_{dd}. If x and y are connected to *GND* and V_{dd} respectively then the left-most P-type transistor will be connected, and both upper-most N-type transistor will be disconnected; r will be connected to V_{dd}. If x and y are connected to V_{dd} then both top P-type transistors will be disconnected, and both bottom N-type transistors will be connected; r will be connected to *GND*. In short, the behaviour we get is described by

x	y	r
GND	*GND*	V_{dd}
GND	V_{dd}	V_{dd}
V_{dd}	*GND*	V_{dd}
V_{dd}	V_{dd}	*GND*

which, if we substitute 0 and 1 for *GND* and V_{dd}, is matches that of the NAND operation.

S3. This question is a lot easier than it sounds; basically we just add two extra transistors (one P-MOSFET and one N-MOSFET) to implement a similar high-level approach. That is, we want r connected to *GND* only when each of x, y *and* z are connected to V_{dd}; this means the bottom, N-MOSFETs are in series. If *any* of x, y or z are connected to *GND*, we want r connected to V_{dd}; this means the top, P-MOSFETs are in parallel. Diagrammatically, the result is as follows:



S4. This is quite an open-ended question, but basically it asks for high-level explanations only. As such, some example answers include the following:

a CMOS transistors are constructed from atomic-level understanding and manipulation; the immutable size of atoms therefore acts as a fundamental limit on the size of any CMOS-based transistor.

b Feature scaling improves the operational efficiency of transistors, simply because smaller features reduce delay. Beyond this however, one must utilise the extra transistors to achieve some useful task if computational efficiency is to scale as well: improvements to an architecture or design are often required, for instance, to exploit parallelism and so on.

c Even assuming the transistors available can be harnessed to improve computational efficiency, this has implications: more transistors within a fixed size area will increase power consumption and also heat dissipation for example, both of which act as limits even if managed (e.g., via aggressive forms of cooling).

d On one hand, smaller transistors mean less cost per-transistor: with a fixed number of transistors, their area and manufacturing cost will decrease. With a fixed sized area and hence more transistors in it however, this probably means increase defect rate during manufacture. The resulting cost implication could act as an economic limit to transistor size.

S5. a The most basic interpretation (i.e., not really doing any grouping using Karnaugh maps but just picking out each cell with a 1 in it) generates the following SoP equations

$$\begin{aligned} e &= (\neg a \wedge \neg b \wedge c \wedge \neg d) \vee (a \wedge \neg b \wedge \neg c \wedge \neg d) \vee (\neg a \wedge b \wedge \neg c \wedge d) \\ f &= (\neg a \wedge \neg b \wedge \neg c \wedge d) \vee (\neg a \wedge b \wedge \neg c \wedge \neg d) \vee (a \wedge \neg b \wedge c \wedge \neg d) \end{aligned}$$

b From the basic SoP equations, we can use the don't care states to eliminate some of the terms to get

$$\begin{aligned} e &= (\neg a \wedge \neg b \wedge c) \vee (a \wedge \neg c \wedge \neg d) \vee (b \wedge d) \\ f &= (\neg a \wedge \neg b \wedge d) \vee (b \wedge \neg c \wedge \neg d) \vee (a \wedge c) \end{aligned}$$

then, we can share both the terms $\neg a \wedge \neg b$ and $\neg c \wedge \neg d$ since they occur in e and f.

S6. Simply transcribing the truth table into a suitable Karnaugh map gives

		y		
		z		
		1	1	0	1
x		0	1	**?**	0

from which we can derive the SoP expression

$$r = (\neg y \wedge z) \vee (\neg x \wedge \neg z).$$

S7. Define $\overline{\wedge}$ as the NAND operation with the truth table:

x	y	$x \mathbin{\overline{\wedge}} y$
0	0	1
0	1	1
1	0	1
1	1	0

Using NAND, we can implement NOT, AND and OR as follows:

$$\begin{aligned} \neg x &= x \mathbin{\overline{\wedge}} x \\ x \wedge y &= (x \mathbin{\overline{\wedge}} y) \mathbin{\overline{\wedge}} (x \mathbin{\overline{\wedge}} y) \\ x \vee y &= (x \mathbin{\overline{\wedge}} x) \mathbin{\overline{\wedge}} (y \mathbin{\overline{\wedge}} y) \end{aligned}$$

To prove this works, we can construct truth tables for the expressions and compare the results with what we would expect; for NOT we have:

x	$x \mathbin{\overline{\wedge}} x$	$\neg x$
0	1	1
1	1	0

while for AND we have:

x	y	$x \overline{\wedge} y$	$(x \overline{\wedge} y) \overline{\wedge} (x \overline{\wedge} y)$	$x \wedge y$
0	0	1	0	0
0	1	1	0	0
1	0	1	0	0
1	1	0	1	1

and finally for OR we have:

x	y	$x \overline{\wedge} x$	$y \overline{\wedge} y$	$(x \overline{\wedge} y) \overline{\wedge} (x \overline{\wedge} y)$	$x \vee y$
0	0	1	1	0	0
0	1	1	0	1	1
1	0	0	1	1	1
1	1	0	0	1	1

such that it should be clear all three are correct.

S8. Conventionally a 4-input, 1-bit multiplexer might be described using a truth table such as the following:

c_1	c_0	w	x	y	z	r
0	0	?	?	?	0	0
0	0	?	?	?	1	1
0	1	?	?	0	?	0
0	1	?	?	1	?	1
1	0	?	0	?	?	0
1	0	?	1	?	?	1
1	1	0	?	?	?	0
1	1	1	?	?	?	1

This assumes that there are four inputs, namely w, x, y and z, with two further control signals c_1 and c_0 deciding which of them provides the output r. However, another valid way to write the same thing would be

c_1	c_0	r
0	0	w
0	1	x
1	0	y
1	1	z

This reformulation describes a 2-input, 1-output Boolean function whose behaviour is selected by fixing w, x, y and z, i.e., connecting each of them directly to either 0 or 1. For instance, if $w = x = y = 0$ and $z = 1$ then the truth table becomes

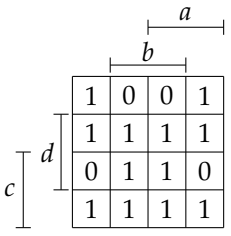
c_1	c_0	r
0	0	$w = 0$
0	1	$x = 0$
1	0	$y = 0$
1	1	$z = 1$

which is of course the same as AND. So depending on how w, x, y and z are fixed (on a per-instance basis) we can form *any* 2-input, 1-output Boolean function; this includes NAND and NOR, which we know are universal, meaning the multiplexer is also universal.

S9. a The expression for this circuit can be written as

$$e = (\neg c \wedge \neg b) \vee (b \wedge d) \vee (\neg a \wedge c \wedge \neg d) \vee (a \wedge c \wedge \neg d)$$

which yields the Karnaugh map



and from which we can derive a simplified SoP form for e, namely

$$e = (b \wedge d) \vee (\neg b \wedge \neg c) \vee (c \wedge \neg d)$$

b The advantages of this expression over the original are that is is simpler, i.e., contains less terms and hence needs less gates for implementation, and shows that the input a is essentially redundant. We have probably also reduced the critical path through the circuit since it is more shallow. The disadvantages are that we still potentially have some glitching due to the differing delays through paths in the circuit, although these existed before as well, and the large propagation delay.

c The longest sequential path through the circuit goes through a NOT gate, two AND gates and two OR gates; the critical path is thus 90 ns long. This time bounds how fast we can used it in a clocked system since the clock period must be at least 90 ns. So the shortest clock period would be 90 ns, meaning the clock ticks about 11111111 times a second (or at about 11 MHz).

S10. a There are various odd things about this circuit, which is an example of something called a ring oscillator. For example:

i There is effectively no input to the circuit, which is odd: the circuit clearly forms a loop whereby the output is fed back as the input (although arguable one can sample one of the wires to provide output).

ii The feedback loop is logically inconsistent: since there is an odd number of NOT gates, if we assume the input to the left-most NOT gate is 0 then it should actually be 1 and vice versa. In addition, the fact that there are three NOT gates is odd, since this has the same effect as one.

Since there are no storage components into which intermediate values could be latched, there is no opportunity for them to settle into a stable state. The result is that the value at the output of the right-most NOT gate, for example, will oscillate between 0 and 1 at a frequency dictated by various delays (e.g., wire and gate).

b The most obvious use for clock generation: the regular oscillation could be utilised as a crude clock signal. Since the frequency is determined by delay, and the delay is influenced by factors such as temperature, the circuit could also be used to measure temperature in a crude way. Finally, such circuits are often used as components for random bit generation: the idea is that that frequency "jitters" as a result of various factors rather than being perfect. This can be harnessed to produce random numbers.

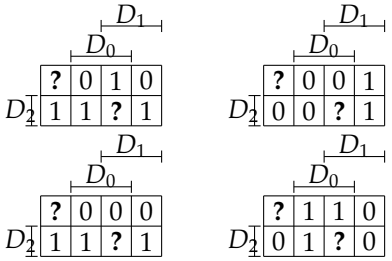
S11. a Examining the behaviour required, we can construct the following truth table:

D_2	D_1	D_0	L_8	L_7	L_6	L_5	L_4	L_3	L_2	L_1	L_0
0	0	0	**?**	**?**	**?**	**?**	**?**	**?**	**?**	**?**	**?**
0	0	1	0	0	0	0	1	0	0	0	0
0	1	0	0	1	0	0	0	0	0	1	0
0	1	1	1	0	0	0	1	0	0	0	1
1	0	0	1	0	1	0	0	0	1	0	1
1	0	1	1	0	1	0	1	0	1	0	1
1	1	0	1	1	1	0	0	0	1	1	1
1	1	1	**?**	**?**	**?**	**?**	**?**	**?**	**?**	**?**	**?**

Note that

$$\begin{aligned} L_3 &= 0 \\ L_5 &= 0 \\ L_6 &= L_2 \\ L_7 &= L_1 \\ L_8 &= L_0 \end{aligned}$$

so actually we only need expressions for $L_{0\ldots2}$ and L_4, and that don't care states are used to capture the idea that $D = 0$ and $D = 7$ never occur. The resulting four Karnaugh maps



can be translated into the expressions:

$$\begin{array}{lcl} L_0 & = & D_2 \vee (D_1 \wedge D_0) \\ L_1 & = & (D_1 \wedge \neg D_0) \\ L_2 & = & D_2 \\ L_4 & = & D_0 \end{array}$$

b All the LEDs can be driven in parallel, i.e., the critical path relates to the single expression whose critical path is the most. $L_{2...6}$ have no logic involved, so we can discount them immediately. Of the two remaining LEDs, we find

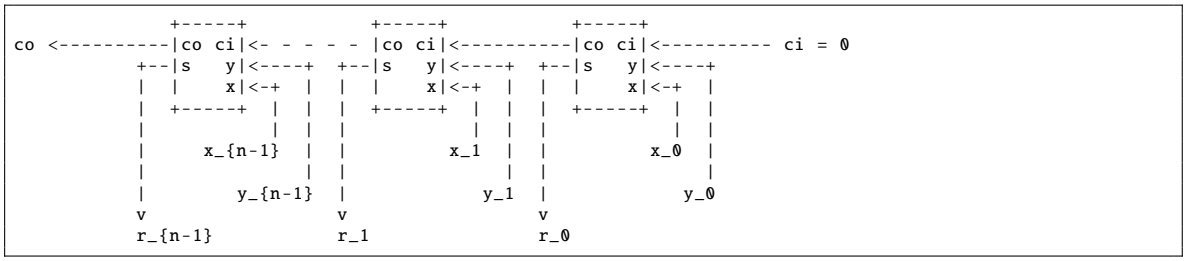
$$\begin{array}{lcl} L_0 & \rightsquigarrow & 20 \text{ ns} + 20 \text{ ns} \\ L_1 & \rightsquigarrow & 10 \text{ ns} + 20 \text{ ns} \end{array}$$

hence L_0 represents the critical path of 40 ns. Thus if one throw takes 40 ns, we can perform

$$\frac{1 \text{ s}}{40 \text{ ns}} = \frac{1 \cdot 10^9 \text{ ns}}{40 \text{ ns}} = 25,000,000$$

throws per-second. Which is quite a lot, and certainly too many to actually see with the human eye!

c i A rough block diagram would resemble



ii If we sum 8 values $1 \leq x_i \leq 6$, where x_i is the i-th throw (or i-th value of D supplied), then the maximum total is $8 \cdot 6 = 48$. We can represent this in 6 bits, hence $n = 6$.

iii A full-adder cell computes outputs via

$$\begin{array}{lcl} co & = & (x \wedge y) \vee ((x \oplus y) \wedge ci) \\ s & = & x \oplus y \oplus ci \end{array}$$

Hence the critical path is 80 ns, either through two XOR gates for s or one XOR, one AND and an OR gate for co. With n such cells in sequence, the total total propagation delay of the ripple-carry adder is hence $80 \cdot n$ ns.

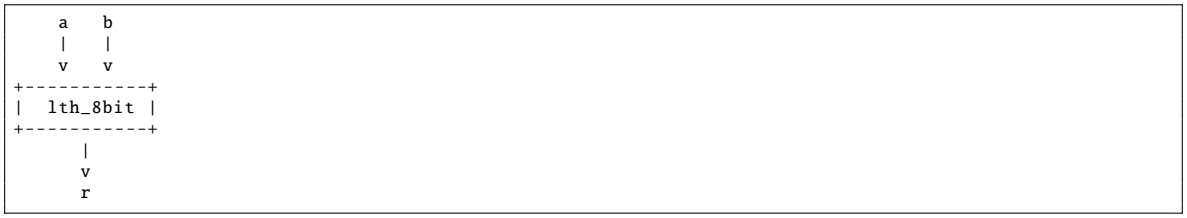
iv Using the left-shift method, we compute $D' = 2 \cdot D$ by simply relabelling the bits in D. That is, $D'_0 = 0$ and $D'_{i+1} = D_i$ for $0 \leq i < 3$. For example, given $D = 6_{(10)} = 110_{(2)}$ we have

$$\begin{array}{lcl} D'_0 & & = 0 \\ D'_1 & = D_0 & = 0 \\ D'_2 & = D_1 & = 1 \\ D'_3 & = D_2 & = 1 \end{array}$$

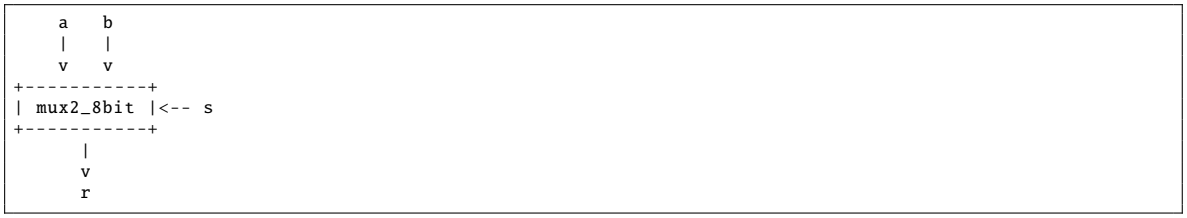
and hence $D' = 1100_{(2)} = 12_{(10)}$. Since there is no need for any logic gates to implement this method, the critical path is essentially nil: the only propagation delay relates to (small) wire delays. In comparison to the larger critical path of a suitable n-bit adder, this clearly means the left-shift approach is preferable.

S12. a A basic design would use two building blocks:

- `lth_8bit` compares two 8-bit inputs a and b and produces a 1-bit result r, where $r = 1$ if $a < b$ and $r = 0$ if $a \geq b$:



- `mux2_8bit` selects between two 8-bit inputs; if the inputs are a and b, the output $r = a$ if the control signal $s = 0$, or $r = b$ if $s = 1$:



Based on these building blocks, one can describe the component C as follows:

From a functional perspective, C compares x and y using an instance of the `lth_8bit` building block, and then uses the result r as a control signal for two instances of `mux2_8bit`. The left-hand instance selects y as the output if $r = 0$ and x if $r = 1$; that is, if $x < y$ then the output is $x = \min(x, y)$ otherwise the output is $y = \min(x, y)$. The right-hand instance swaps the inputs so it selects x as the output if $r = 0$ and y if $r = 1$; that is, if $x < y$ then the output is $y = \max(x, y)$ otherwise the output is $x = \max(x, y)$.

b The short answer (which gets about half the marks) is that the longest path through the mesh will go through $2n - 1$ of the C components: this is the path from the top-left corner down along one edge to the bottom-left and then along another edge to the bottom-right. So in a sense, if we write the propagation delay associated with each instance of C as T_C then the overall critical path is

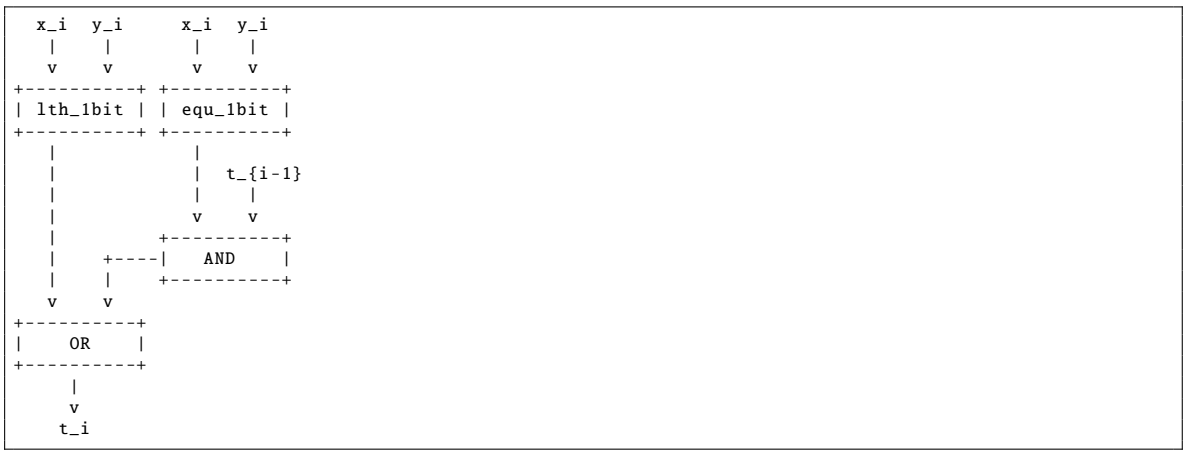
$$(2n - 1) \cdot T_C.$$

In a bit more detail, the critical path through C is through one instance of `lth_8bit` and one instance of `mux2_8bit`. So we could write the overall critical path is

$$(2n - 1) \cdot (T_{\texttt{lth\_8bit}} + T_{\texttt{mux2\_8bit}}).$$

To be more detailed than this, we need to think about individual logic gates. Imagine we assume $T_{AND} = 50$ ns, $T_{AND} = 20$ ns, $T_{OR} = 20$ ns and $T_{NOT} = 10$ ns.

- `mux2_8bit` is simply eight `mux2_1bit` instances placed in parallel with each other; that is, the i-th such instance produces the i-th bit of the output based on the i-th bit of the inputs (but all using the same control signal). Assuming that the propagation delay of AND and OR gates dominates that of a NOT gate, the critical path through `mux2_1bit` will be $T_{AND} + T_{OR}$.
- `lth_8bit` is a combination of eight sub-components:



Each of these sub-components is placed in series so that t_{i-1} is an input from the previous sub-component and t_i is an output provided to the next.

Based on simple circuits derived from their truth tables, the critical paths for `lth_1bit` and `equ_1bit` are $T_{AND} + T_{NOT}$ and $T_{XOR} + T_{NOT}$ respectively. Thus the critical path of the whole sub-component is $T_{XOR} + T_{NOT} + T_{AND} + T_{OR}$ (since the critical path of `equ_1bit` is longer). Overall, the critical path of `lth_8bit` is

$$8 \cdot (T_{XOR} + T_{NOT} + T_{AND} + T_{OR}),$$

or more exactly

$$7 \cdot (T_{XOR} + T_{NOT} + T_{AND} + T_{OR}) + T_{AND} + T_{NOT}$$

because the 0-th sub-component is "special": there is no input from the previous sub-component.

Using this we can write the overall critical path for the mesh as

$$(2n - 1) \cdot (7 \cdot T_{XOR} + 8 \cdot T_{NOT} + 9 \cdot T_{AND} + 8 \cdot T_{OR})$$

or roughly $(2n - 1) \cdot 770$ ns if we plug in the assumed delays.

c One problem is that the mesh does not always give the right result! If you were to build a 4×4 mesh and feed $5, 6, 7, 8$ into the top and $4, 3, 2, 1$ into the left-hand side, the bottom reads $5, 6, 7, 8$ and the right-hand side reads $4, 3, 2, 1$: numbers cannot move from bottom to top or from right to left, so there are some inputs a mesh cannot sort.

Beyond this trick question, the main idea is that the mesh is special-purpose, the processor is general-purpose: this implies a number of trade-offs in either direction that could be viewed as advantages or disadvantages in certain cases. For example, depending on n, one might argue that the processor will be require more logic to realise it (since it will include features extraneous to the task of sorting). Since it operates a fetch-decode-execute cycle to complete each instruction, there is an overhead (i.e., the fetch and decode at least) which means it potentially performs the task of sorting less quickly. On the other hand, once constructed the mesh is specialised to one task: it cannot be used to sort strings for example, and the size of input (i.e., n) is fixed. The processor makes the opposite trade-off; it should be clear that while it might be slower and potentially larger, it is vastly more flexible.

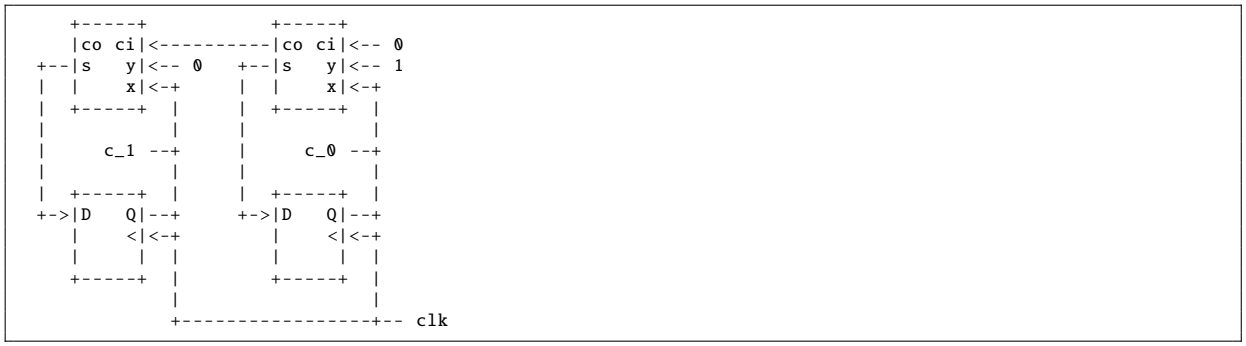
S13. a Imagine a component which is enabled (i.e., "turned on") using the input *en*:

- The idea of the component being level triggered is that the value of *en* is important, not a change in *en*: the component is enabled when *en* has a particular value, rather than at an edge when the value changes.
- The fact *en* is active high means that the component is enabled when $en = 1$ (rather than $en = 0$ which would make it active low). Though active high might seem the more logical choice, this is just part of the component specification: as long as everything is consistent, i.e., uses the right semantics to "turn on" the component, there is sometimes no major benefit of one approach over the other.

b Assume that M is a 4-state switch represented by a 2-bit value $M = \langle M_0, M_1 \rangle$: $\langle 0, 0 \rangle$ means off, $\langle 1, 0 \rangle$ means slow, $\langle 0, 1 \rangle$ means fast and $\langle 1, 1 \rangle$ means very fast. Also assume there is a clock signal called *clk* available, for example supplied by an oscillator of some form.

One approach would basically be to take *clk* and divide it to create two new clock signals c_0 and c_1 which have a longer period: each of the clock signals could then satisfy the criteria of toggling the fire button on and off at various speeds. A clock divider is fairly simple: the idea is to have a counter c clocked by *clk* and to sample the $(i - 1)$-th bit of the counter: this behaves like *clk* divided by 2^i. For example the 0-th bit acts like *clk* but with twice the period.

A circuit to do this is fairly simple: we need some D-type flip-flops to hold the counter state, and some full-adders to increment the counter:



Given such a component which runs freely as long as it is driven by *clk*, we want to feed the original fire button F_0 through to form the new fire button input F'_0 when $M = 0$, and c_1, c_0 or *clk* through when $M = 1$, $M = 2$ or $M = 3$ (meaning a slow, fast or very fast toggling behaviour). We can describe this as the following truth table:

M_1	M_0	F'_0
0	0	F_0
0	1	c_1
1	0	c_0
1	1	*clk*

This is essentially a multiplexer controlled by M, and permits us to write

$$\begin{array}{rcccccccc} F'_0 & = & (& \neg M_0 & \wedge & \neg M_1 & \wedge & F_0 &) \\ & & (& M_0 & \wedge & \neg M_1 & \wedge & c_1 &) \\ & & (& \neg M_0 & \wedge & M_1 & \wedge & c_0 &) \\ & & (& M_0 & \wedge & M_1 & \wedge & clk &) \end{array}$$

c i A synchronous protocol demands that the console and controller share a clock signal which acts to synchronise their activity, e.g., ensures each one sends and receives data at the right time. The problem with this is ensuring that the clock is not skewed for either component: since they are physically separate, this might be hard and hence this is not such a good option.

An asynchronous protocol relies on extra connections between the components, e.g., "request" and "acknowledge", that allow them to engage in a form of transaction: the extra connections essentially signal when data has been sent or received on the associated bus. This is more suitable given the scenario: the extra connections could potentially be shared with those that already exist (e.g., F_0, F_1, F_2 and D) thereby reducing the overhead, plus performance is not a big issue here (the protocol will presumably only be executed once when the components are turned on or plugged in).

Both approaches have an issue in that

- once the protocol is run someone could just plug in another, fake controller, or
- or simply intercept c and $T(c)$ pairs until it recovers the whole look-up table and then "imitate" it using a fake controller

so neither is particularly robust from a security point of view!

ii The temptation here is to say that the use of a 3-bit memory (or register) is the right way to go. Although this allows some degree of flexibility which is not required since the function is fixed, the main disadvantage is retention of the content when the controller or console is turned off: some form of non-volatile memory is therefore needed.

However, we can easily construct some dedicated logic to do the same thing. If we say that $y = T(x)$, the we can describe the behaviour of T using the following truth table:

x_2	x_1	x_0	y_2	y_1	y_0
0	0	0	0	1	0
0	0	1	1	1	0
0	1	0	1	1	1
0	1	1	0	0	1
1	0	0	1	0	0
1	0	1	0	0	0
1	1	0	1	0	1
1	1	1	0	1	1

This can be transformed into the following Karnaugh maps for y_0, y_1 and y_2

x_2 \ x_1 x_0	00	01	11	10
0	0	0	1	1
1	0	0	1	1

x_2 \ x_1 x_0	00	01	11	10
0	1	1	0	1
1	0	0	1	0

x_2 \ x_1 x_0	00	01	11	10
0	0	1	0	1
1	1	0	0	1

which in turn can be transformed into the following equations

$$\begin{array}{lcl}
y_0 &=& (\quad\quad\quad\quad x_1 \quad\quad\quad\quad) \\
y_1 &=& (\quad \neg x_2 \quad \wedge \quad \neg x_1 \quad\quad\quad\quad) \quad \vee \\
&& (\quad \neg x_2 \quad\quad\quad\quad \wedge \quad \neg x_0 \quad) \quad \vee \\
&& (\quad x_2 \quad \wedge \quad x_1 \quad \wedge \quad x_0 \quad) \\
y_2 &=& (\quad\quad\quad\quad x_1 \quad \wedge \quad \neg x_0 \quad) \quad \vee \\
&& (\quad x_2 \quad\quad\quad\quad \wedge \quad \neg x_0 \quad) \quad \vee \\
&& (\quad \neg x_2 \quad \wedge \quad \neg x_1 \quad \wedge \quad x_0 \quad)
\end{array}$$

which are enough to implement the look-up table: we pass x as input, and it produces the right y (for this fixed T) as output.

S14. This is a classic "puzzle" question in digital logic. There are a few ways to describe the strategy, but the one used here is based on counting the number of inputs which are 1. In short, we start by computing

$$\begin{array}{lcl}
t_1 &=& \neg(x \wedge y \vee y \wedge z \vee x \wedge z) \\
t_2 &=& \neg((x \wedge y \wedge z) \vee t_1 \wedge (x \vee y \vee z))
\end{array}$$

which use our quota of NOT gates. The idea is that $t_1 = 1$ iff. one or zero of x, y and z are 1, and in the same way $t_2 = 1$ iff. two or zero of x, y and z are 1. This can be hard to see, so consider a truth table

x	y	z	$x \wedge y$	$y \wedge z$	$x \wedge z$	$x \wedge y \wedge z$	$x \vee y \vee z$
0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	1
0	1	0	0	0	0	0	1
0	1	1	0	1	0	0	1
1	0	0	0	0	0	0	1
1	0	1	0	0	1	0	1
1	1	0	1	0	0	0	1
1	1	1	1	1	1	1	1

meaning

x	y	z	$x \wedge y \vee y \wedge z \vee x \wedge z$	t_1	$(x \wedge y \wedge z) \vee t_1 \wedge (x \vee y \vee z)$	t_2
0	0	0	0	1	0	1
0	0	1	0	1	1	0
0	1	0	0	1	1	0
0	1	1	1	0	0	1
1	0	0	0	1	1	0
1	0	1	1	0	0	1
1	1	0	1	0	0	1
1	1	1	1	0	1	0

and hence t_1 and t_2 are as required. Now, we can generate the three results as

$$\begin{array}{lcl}
\neg x &=& (t_1 \wedge t_2) \vee (t_1 \wedge (y \vee z)) \vee (t_2 \wedge y \wedge z) \\
&=& (t_1 \wedge t_2) \vee (t_1 \wedge y) \vee (t_1 \wedge z) \vee (t_2 \wedge y \wedge z) \\
\neg y &=& (t_1 \wedge t_2) \vee (t_1 \wedge (x \vee z)) \vee (t_2 \wedge x \wedge z) \\
&=& (t_1 \wedge t_2) \vee (t_1 \wedge x) \vee (t_1 \wedge z) \vee (t_2 \wedge x \wedge z) \\
\neg z &=& (t_1 \wedge t_2) \vee (t_1 \wedge (x \vee y)) \vee (t_2 \wedge x \wedge y) \\
&=& (t_1 \wedge t_2) \vee (t_1 \wedge x) \vee (t_1 \wedge y) \vee (t_2 \wedge x \wedge y)
\end{array}$$

S15. Imagine that for some n-bit input x, we let $y_i = C_i(x)$ denote the evaluation of C_i to get an output y_i. As such, the equivalence of C_1 and C_2 can be stated as a test whether $y_1 = y_2$ for all values of x; another way to say the same thing is to test whether an x exists such that $y_1 \neq y_2$ which will distinguish the circuits, i.e., imply they are not equivalent.

Using the second formulation, we can write the test as $y_1 \oplus y_2$ since the XOR will produce 1 when y_1 differs from y_2 and 0 otherwise. As such, we have n Boolean variables (the bits of x) and want an assignment that implies the expression $C_1(x) \oplus C_2(x)$ will evaluate to 1. This is the same as described in the description of SAT, so if we can solve the SAT instance we prove the circuits are (not) equivalent.

S16. a The latency of the circuit is the time taken to perform the computation, i.e., to compute some r given x. For this circuit, the latency is simply the sum of the critical paths.

b The throughput is the number of operations performed per unit time period. This is essentially the number of operations we can start (resp. that finish) within that time period.

By pipelining the circuit, using say 3 stages, one might expect the latency to increase slightly (by virtue of having to add pipeline registers between each stage) but the throughput to increase (by virtue of decreasing the overall critical path to the longest stage, and hence increasing the maximum clock frequency). The trade-off is strongly influenced by the number of and balance between stages, meaning careful analysis of the circuit before applying the optimisation is important.

S17. a The latency of a circuit is the time elapsed between when a given operation starts and when it finishes. The throughput of a circuit is the number of operations that can be started in each time period; that is, how long it takes between when two subsequent operations can be started.

b The latency of the circuit is the sum of all the latencies of the parts,i.e.,

$$40 \text{ ns} + 10 \text{ ns} + 30 \text{ ns} + 10 \text{ ns} + 50 \text{ ns} + 10 \text{ ns} + 10 \text{ ns} = 160 \text{ ns}.$$

The throughput relates to the length of the longest pipeline stage; the circuit is not pipelined, so more specifically we can say it is $\frac{1}{160 \cdot 10^{-9}}$.

c The new latency is still the sum of all the parts, but now includes the extra pipeline register:

$$40 \text{ ns} + 10 \text{ ns} + 30 \text{ ns} + 10 \text{ ns} + 10 \text{ ns} + 50 \text{ ns} + 10 \text{ ns} + 10 \text{ ns} = 170 \text{ ns}.$$

However, the throughput is now more because the longest pipeline stage only has a latency of 100 ns (including the extra register). Specifically, the throughput increases to $\frac{1}{100 \cdot 10^{-9}}$ which essentially means we can start new operations more often than before.

d To maximise the throughput we need to minimise the latency of the longest pipeline stage (i.e., the one whose individual latency is the largest) since this will act as a limit. The latency of part E is largest (at 50 ns) and hence represents said limit: the longest pipeline stage cannot have a latency of less than 60 ns (i.e., the latency of part E plus the latency of a pipeline register).

We can achieve this by creating a 4-stage pipeline: adding two more pipeline registers, between parts B and C and parts E and F, ensures the stages have latencies of

$$\begin{array}{lcrcr} A + B + REG & \leadsto & 40 \text{ ns} + 10 \text{ ns} + 10 \text{ ns} & = & 60 \text{ ns} \\ C + D + REG & \leadsto & 30 \text{ ns} + 10 \text{ ns} + 10 \text{ ns} & = & 50 \text{ ns} \\ E + REG & \leadsto & 50 \text{ ns} + 10 \text{ ns} & = & 60 \text{ ns} \\ F + REG & \leadsto & 10 \text{ ns} + 10 \text{ ns} & = & 20 \text{ ns} \end{array}$$

Overall, the latency is increased to 190 ns but the throughput is $\frac{1}{60 \cdot 10^{-9}}$.