Performance

The *speed* of an electronic system is often of great interest.

- ❑ The performance of a system is (limited by) the product of its clock period and the number of cycles taken to do the job.
  - ❍ These are not independent.
  - ❍ Both are determined by the design.
- ❑ Determining the number of clock cycles is done by the RTL. (e.g. by pipeline depth).
  - ❍ There may be scope for optimisation.
  - ❍ RTL simulation can easily determine cycle count.
- ❑ Clock period is set by logic depth.
  - ❍ Can *guess* at what this might be.
  - ❍ Can *ask* the logic synthesizer to optimise implementation.
  - ❍ Only *know* the answer when the layout is complete

Performance

The performance of a synchronous system depends on two factors:

❑ The number of clock cycles required to perform a task
❑ The length (period) of the clock cycle

These can sometimes be treated semi-independently although reducing the period means that less can be done in a cycle which may necessitate more clock cycles. The actual elapsed time is (of course) the product of the number of cycles and the cycle period.

Cycle counting

A first guide to the performance of a synchronous system is the (average) number of cycles taken to perform an operation. In a microprocessor this is often given as the CPI (Cycles Per Instruction); this may not be an integer because of other factors which reduce *average* performance such as pipeline flushes. The average CPI of a scalar RISC is likely to be slightly greater than one. [In superscalar architectures this can be less than one because, on some cycles, several instructions can be executed; sometimes the figure is then quoted as ‘IPC’.]

Counting cycles is quite an easy way to assess RTL performance as it can be done using a digital simulator, independently from the detailed design or the layout. It is therefore useful to reduce cycle count – typically by maximising parallelism – at this time.

However, putting too much logic into a single cycle may reduce the cycle count at the expense of the cycle time. An experienced designer will have some idea as to the logic ‘depth’ which should be placed between latches. Too many functions in series without latches in between will slow the system down.

Clock period

Until the layout is complete the minimum clock period (maximum frequency) can only be estimated. However experience of experimentation can guide a designer and this can be used to annotate a HDL (Verilog) description to produce a more representative simulation. An RTL description cannot dictate the delays however, only try to represent what they might be.

At gate level the clock period is set (roughly) by the gate depth (number of series gates) between registers. This will vary according to design criteria. A rough guide might be to aim for a maximum of (say) 30 gates. A very fast clocking pipeline might restrict this number much more severely though.

The clock period is then the propagation delay of the register, plus the logic, plus the set-up time of the subsequent register (plus a margin for reliability).

Considerations

There are two alternative driving factors in setting performance targets:

❑ a real-time constraint
❑ ‘as fast as possible’

A device such as a TV tuner/decoder is constrained to run at least as fast as the datastream; it *must* meet this target. However there is no point in running faster than this as there is no benefit. It is governed by the available throughput.

On the other hand a microprocessor (for example) may be designed to minimise latency in all cases, thus run as fast as possible (possibly halting when not in use). Normally this is harder to do because there is always pressure to improve further; however it is also easier in that a small ‘miss’ can be allowed for with a reduced clock rate.

Partitioning

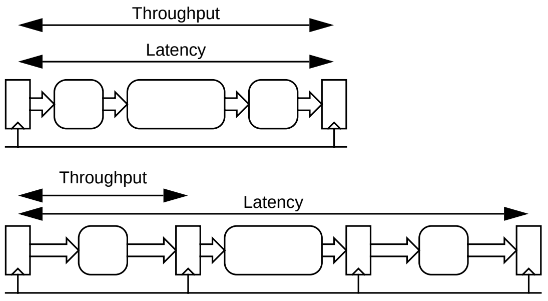
How should a design be partitioned?

If there was one answer to this life would be much simpler!

- ❑ Sometimes there are obvious ‘blocks’ which are difficult/impossible/silly to subdivide
  - ❍ {Memories, ALU …}
  - ❍ Set a minimum period
- ❑ Judgement: experience is often a guiding factor
  - ❍ Know something about similar systems from the past
  - ❍ Useful but may be prejudiced
- ❑ **Modelling**: high-level models can be built to evaluate possibilities
  - ❍ (Approximate?) cycle counts from abstract models
  - ❍ Languages growing to support this {System-C, System-Verilog, Bluespec …}
  - ❍ May be the only answer as complexity increases

Pipelining

Pipelining is just one example of a system partition, but is quite significant in processors and such. Pipelines introduce parallelism and thus, if used sensibly, enable significant throughput increase. At the same time the latency – the total time taken for a single operation – is inevitably increased: firstly latches are added so the logic path is longer. Secondly the latches can only slow the flow down to make each stage take the same time – that of the slowest logic block.



Pipeline balance

There is only one critical path in any pipeline stage and that sets the clock period for the whole synchronous domain. Thus there is only one point where timing optimisations need to be concentrated.

An obvious statement?

In practice the critical paths in most or all stages should be roughly matched; no single stage or operation should dominate the timing model; if it does then all other operations are being slowed down, probably unnecessarily.

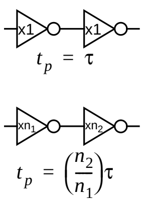
In a microprocessor the critical path may well be memory accesses. Expedients such as pipelining memory (possible but tricky) may be employed to alleviate this. Another possibility is allowing multiple cycles for one operation: imagine fetching two words in parallel from a cache on every alternate cycle. (This last ‘trick’ involves persuading the CAD tools that some timings don’t matter.)

Buffering

- ❑ Gate inputs impose a load (capacitance) on the gate which drives them.
- ❑ Wiring imposes a load (capacitance) on the gate which drives it.
- ❑ More fan out (target gates) leads to more wiring and lots more load.
- ❑ Large loads lead to slow edges, which lead to slow circuits.

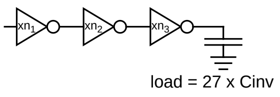
Ignoring wiring loads (for a moment).

When one gate drives another, the delay experienced by the driving gate is proportional to the ratio of the width of the transistor in the receiving gate to the width of the transistor in the driving gate. When a width one inverter drives another width one inverter the driven gate propagation delay is defined as τ. τ is a fundamental parameter of a process and is normally quoted by the manufacturer.



τ enables gate delays for other sizes to be predicted by inspection. For example taking n1 and n2 as the widths of the driving and receiving logic gate transistors relative to that of a minimum sized inverter, the propagation delay of the driving gate is τ x n2/n1.

Consider driving a capacitive load equivalent to 27 times that of the minimum sized inverter with a buffer chain as shown here:



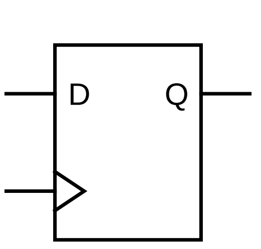
For a selection of gate sizes the delay t_p of the network in terms of τ is shown in the table.

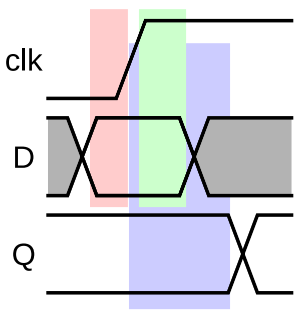
n_1	n_2	n_3	t_p
x1	x1	x1	29τ
x1	x1	x27	29τ
x1	x1	x9	13τ
x1	x9	x9	13τ
x1	x3	x9	9τ

Clocking

The synchronous model is an enormous simplification in designing working Finite State Machines

In order for this to work there are certain assumptions.





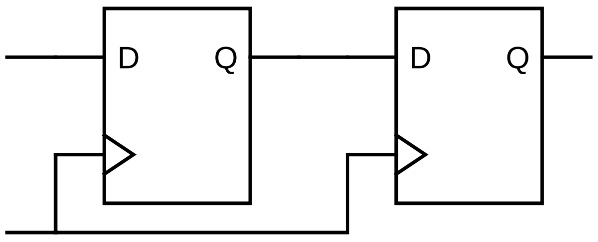
- ❏ data setup time (t_{su})
  - ❍ data stability before the clock
- ❏ data hold time (t_{hold})
  - ❍ data stability after the clock
- ❏ propagation delay (t_{pd})
  - ❍ output lag after the clock

In the synchronous model it is assumed that one flip-flop can feed another

Which means this is possible:

assuming the clock reaches both (all) devices ‘simultaneously’.

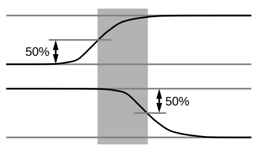
i.e. minimal **clock skew**



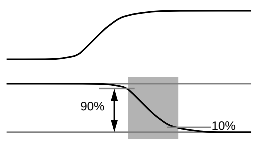
Some Terminology

It is not always clear what is meant by terms such as ‘propagation delay’. Typical definition are:

- ❏ The propagation delay of a gate is the time between an input transition crossing the halfway point between its two logic states and an output caused by that input crossing the same level.
- ❏ The edge speed of a signal is the time between the signal spends between 10% and 90% of the way from one logic level to the other.



Propagation delay



Edge speed

Do not learn these by rote! Remember the pictures. Other definitions could use ‘gate threshold’, rather than the 50% point, for example.

CMOS circuits will ‘swing’ signals from ‘rail to rail’: i.e. the logic voltages will be very close to the power supply voltages … eventually. The edges slow down as the final value is approached (because the capacitive load is almost charged and this opposes reduces the effective charging potential). This is why (a value like) 90% rather than 100% is more useful.

The output edge speed is primarily a result of the driving gate’s ‘strength’ vs. the total output load. However it may be slowed down further if the input edge is particularly slow.

Clock Period Limits

The minimum clock period is determined by:

- ❏ The propagation delay of the register, from clock to output

plus

- ❏ The critical path (slowest logic path) between registers … anywhere

plus

- ❏ The data setup time of a register

plus

- ❏ An allowance for clock skew

plus

- ❏ An allowance for clock jitter

possibly plus

- ❏ A bit of extra safety margin
  - ❍ allowing for temperature variations etc.

If this is not fast enough then – assuming there are no real time restrictions – you might have to run with a slower clock than you might have liked. (This may lead to a restructure to speed up the clock or do more per clock ‘tick’.)

Hold time restrictions

In modern processes some delay from flip-flop to flip-flop may be assumed. If two flip-flops are connected directly – as in the slide – then it may be that the hold time of the second flip-flop could be violated (particularly when skew and jitter are taken into account). This is a result of a **race** between the second flip-flop completing its data capture and the first transmitting its new data. Both these are triggered by the *same* clock edge.

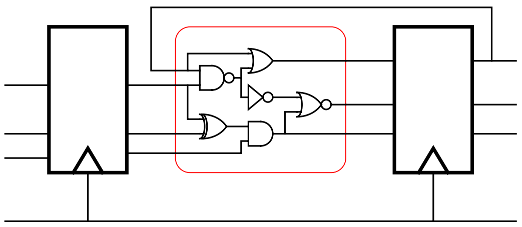
In such cases it is necessary to *insert* some extra delay on the data signal. This will *not* be on the critical path; other registers will be separated by bigger logic delays. It is essential to get this right because, if there can be a violation, there is nothing to be done about it; slowing the clock will not help.

Timing Closure

Simulation

- ❑ Okay for a rough-cut
- ❑ May be difficult to simulate **critical path**
  - ❍ the ‘longest’ (slowest!) path between registers

Static Timing Analysis (STA)



- ❑ Take blocks of logic between synchronously clocked elements
- ❑ Time all possible switching paths in block
- ❑ Find the longest

Advantages

- ❍ Quick to perform
- ❍ Should give a *conservative* upper bound

Disadvantage

- ❍ Can be too pessimistic

Timing closure

Timing closure is, basically, making the logic fit within the desired clock period.

How fast does it go?

This can be difficult to determine, exactly. It is set by the **critical path**. From a HDL source this requires at least *technology mapping* into gates. Accuracy requires knowledge of gate strengths, wire load and layout detail. However it is cost effective to estimate timing early to check that the implementation strategy is feasible. Even pre-layout the tools usually give an estimate of the wire loads to yield a more realistic result.

Simulation

Simulation can indicate whether a *particular sequence* will fail at a particular clock speed. This is a reasonable guide but is not reliable unless either the critical path is known (and exercised) or the simulation is exhaustive.

Example: a ripple carry adder. Simulation with random inputs is unlikely to find the slowest case, when a carry propagates across the whole width.

Static Timing Analysis (STA)

In this case ‘static’ means independent of input state. The delays through each combinatorial path can be summed and compared with the design objective. This reveals the critical path or the ‘slack’ in all logic paths. In the latter case negative slack will reveal where the logic is too slow.

The great advantage of static analysis is its low computational complexity. The disadvantage is that the ‘critical path’ may be a *false* path, i.e. one whose switching sequence cannot occur in reality.

In general STA will identify anything which is significantly bad at low cost.

Does it go fast enough?

No, of course it doesn’t; where would the fun be in that?

Seriously, what is ‘fast enough’? In many applications there will be a real-time constraint to be met but exceeding it brings no additional benefit. In other applications (e.g. microprocessors) any performance increase is to be seized.

How can the speed be improved?

How close are you to your target? If you’re ‘miles off’ you need to restructure your architecture to increase parallelism. This may be done by:

- ❑ deeper pipelining ⇒ faster clock
- ❑ increase logic parallelism ⇒ do more within clock cycle
- ❑ evaluating several things at once ⇒ do more with slower clock
- ❑ multi-cycle operations ⇒ sometimes allow more than one period

If close to target you might be able to identify and recode critical modules.

Tools can also be instructed to optimise for certain criteria, such as speed, power, area, … Normally gains in one category are paid for in others.

Technology

Many cells come in families with various drive strengths. Increasing the drive will speed up an output (and slow the input, and probably cost power).

It may be possible to use different cell families to improve performance. E.g.

❑ High-speed	low threshold transistors switch faster but leak more
❑ Standard	a compromise design
❑ Low-leakage	high threshold transistors save power but switch slowly

Post-layout …

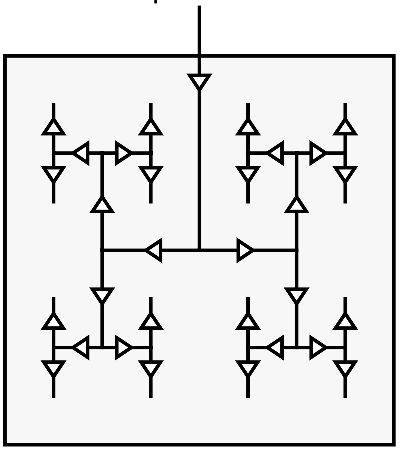
The process may need repeating. After the wiring is factored in things (probably) have slowed down. Buffers may be added which increase the latency but speed up edges.

Hopefully this process converges on something acceptable.

Optimise early to avoid wasting effort on ‘hopeless’ designs. Layout and extraction all take time and more accurate modelling also takes longer.

Clock Distribution

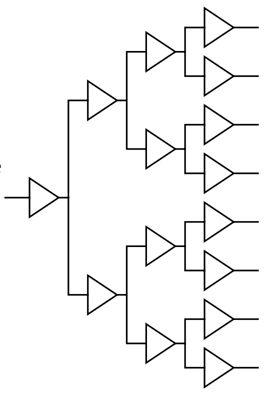
- The synchronous model assumes all flip-flops are clocked simultaneously.
- The difference in effective arrival times of a clock is called the ‘**clock skew**’.
- Skew is unavoidable but must be kept small to validate the original assumptions.



- The ‘classic’ picture of clock distribution is the ‘H’ tree
  - Paths are balanced in drive, load and wiring length

Clock Buffering

- The clock signal goes ‘everywhere’: to the clock input of every flip-flop; its **fan-out is huge**.
- Requires geometric scaling up of the signal
- Often multiple smaller gates are used instead.
  - Easier to build
  - More convenient to place
  - A harder problem to deskew



In practice clock arrival times have to be ‘close enough’.

Observations on clock distribution

- Clocks have high fan-out
  - a clock may fan out to thousands of flip-flops
- Clock edges should be ‘fast’
  - Slow edge speeds increase the uncertainty in exactly when the transition ‘happens’
- Clock signals are repetitive
  - The **latency** of a clock reaching a flip-flop **doesn’t matter**
  - They always arrive at regular intervals

These properties mean that clock signals require buffers … but that’s okay providing the skew is still kept small.

The clock tree needs to be well balanced, taking into consideration:

- Fan-out at each point
- Buffer strength
- Wire lengths

Fortunately, tools will help with this.

Reset skew

Reset *activation* is not normally a timing problem because reset will be present for some time. The *removal* of reset can be a problem though. Imagine reset being removed in one part of a state machine but not quite making it to another. This could cause the machine to enter an illegal state. It is sometimes necessary to synchronise an otherwise asynchronous reset to prevent this.

Clock Jitter

All the foregoing assumptions rely on the clock period being constant. Although it is supposed to be regular it probably isn’t. The clock is an oscillator which is stabilised by feedback; if it varies it is subsequently corrected but this has to be observed first. Thus there is some variation, known as ‘**jitter**’ in the clock period. This may mean any particular clock period is longer *or shorter* than the average. Exact values are circuit dependent but a rule of thumb would suggest a clock jitter of ~5% of the period.

Jitter is unpredictable and typically unavoidable. It is therefore essential that the circuit will always function ‘a bit faster’ than its specified operational speed to allow for this.

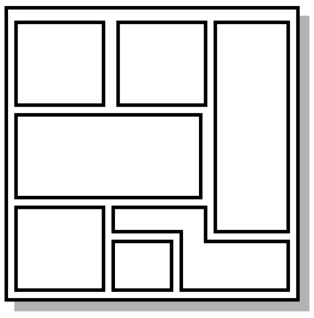
FPGA clock distribution

Modern FPGAs have a dedicated set of clock distribution networks built onto the chip which deliver clock signals to all flip-flops with minimal skew. There are typically a small number (e.g. four) of these networks so that a number of different clocks may be used. These networks can only be used for clocking flip-flops.

Clock Domains

Synchronous design is a Good Thing

- ❏ Simplifies RTL design
  - ❍ May be easier to think about state diagrams
- ❏ Simplifies debugging – can take a ‘global’ view of **state**
- ❏ Tool chains optimised for such



However it is not always possible to have one clock across an SoC

- ❏ Synchronous clock distribution increasingly difficult
- ❏ Blocks may work optimally at different frequencies
  - ❍ May be IP from different vendors
- ❏ Some I/O may require specific frequencies

Clocking Different Domains

Different clock domains may be driven by:

- ❏ the same clock (but with some phase skew)
  - ❍ possibly for convenience, simplifying clock distribution
  - ❍ may be deliberate to reduce radiated noise (see box below)
- ❏ multiples/divisions of a common clock
  - ❍ typically to tailor clocks to appropriate frequencies for different IP
- ❏ completely independent clock sources
  - ❍ allows ‘exact’ matching of interface frequencies {USB, Ethernet, Radio, …}
  - ❍ means no fixed phase relationship (two ‘identical’ clocks will never quite agree)

In all cases there is need for care when a signal is generated in one domain and is read in another. With a fixed phase relationship it may be possible to ensure set-up and hold times are met. Typically, though, it is easiest to assume the input is asynchronous and then synchronise it to its new clock.

Electromagnetic noise

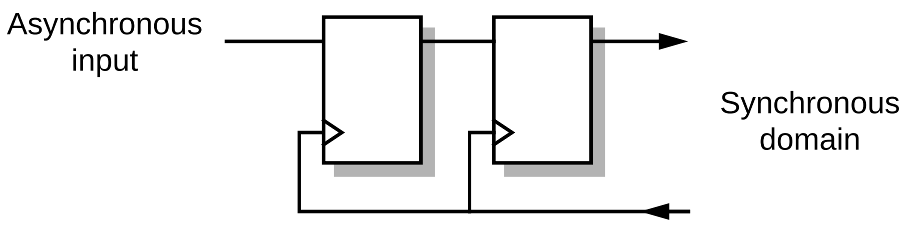
Radio waves are electromagnetic and are created by accelerating charged particles. Example: when a signal switches a current (of electrons) starts to flow and then stops again; this will produce some electromagnetic noise, typically as RFI (Radio Frequency Interference). This is usually a Bad Thing: think of all those radio receivers nearby!

Synchronous digital circuits start to switch with the clock and activity increases and then dies out until the next clock. The clock therefore correlates activity and concentrates the RFI into particular wavebands.

Decorrelating clocks tends to ‘even out’ the switching over a chip as a whole, which can reduce the consequent RFI.

Synchronisation

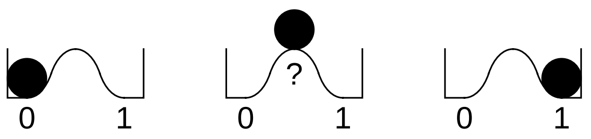
- The danger is that sometimes **a set-up or hold time will be violated**
  - Unavoidable unless input has a fixed, known phase relationship
- Employ a synchroniser to give time for any metastability to resolve



- Principle: if first flip-flop goes metastable it is allowed a clock period to resolve
  - The problem is contained in the synchroniser
- Danger: it may not resolve by the next clock edge (low probability)
  - Depends on: clock period, flip-flop design

Metastability

A model flip-flop



- The flip-flop has *three* stable positions: ‘0’, ‘1’ and a **metastable** position ‘half-way’ between.
- Violating set-up/hold conditions can result in the flip-flip entering the metastable state.
- In principle the flip-flop can stay metastable indefinitely
  - But if it starts to resolve one way, positive feedback pushes it further in that direction
- The probability of *remaining* metastable decreases exponentially with time.

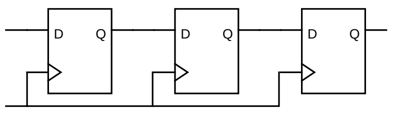
The dangers in a metastable state lie in that it can be interpreted as different values by different inputs, or at different times.

Synchroniser flip-flops

Some cell libraries provide flip-flops specifically to address this problem. They can still go metastable but they have a ‘steeper hill’ so they tend to resolve more quickly. They have worse properties in other respects.

Synchronisers

An ‘improved’ synchroniser may use more flip-flops:



The first flip-flop *probably* doesn’t remain metastable for a whole clock period. If metastability persists, the next flip-flop may go metastable; there is then another clock period for it to resolve. The probability of remaining metastable is a negative exponential in time, so the probabilities are multiplied.

More paranoid designers may add more flip-flops. Each multiplies the probability of remaining metastable by the same small number, thus if (say) 1 in 10^6 is too high, go for 1 in 10^{12}, 1 in 10^{18}, etc. Each flip-flop (delay) also increases the latency, of course.

There is **no certain guarantee** that this will always work. However the probability of failure can be made *very* small.

Adding more flip-flops increases the **latency** of sensing the input, which is often a concern.

[Remember that 3 GHz translates to 3×10^9 clocks/second or about 10^{19} cycles/century.]

Verilog synchroniser example

```
    always @ (posedge clk)      // Synchroniser
      begin
      sync_1 <= async_input;
      sync_2 <= sync_1;
      sync_out <= sync_2;
      end
```

Summary

- ❏ Timing is an important issue.
  - ❍ Dividing up a design into appropriate steps takes practice …
  - ❍ … and, increasingly, modelling.
- ❏ Clock distribution is difficult!
  - ❍ It is largely managed by CAD tools.
  - ❍ Stay synchronous whenever expedient.
- ❏ With ever-increasing capacity of SoC multiple synchronous domains are becoming common
  - ❍ Interconnection: a problem for the future?

Network on Chip (NoC)

With the proliferation of IP blocks in a modern chip the interconnection is an increasing problem. Various **bus** standards have appeared, probably dominated by ARM’s (open) AMBA specifications. These are (mostly) synchronous.

Buses can be bottlenecks in system performance as only one operation can happen at once. These is an increasing interest in Network on Chip (NoC) where components are connected by switched networks, much as ‘big’ computers have been for some time.

NoC is a subject in itself. Here it is sufficient to speculate on the timing implications.

One approach is to keep all the IP and the network synchronous. Another is to have parts of the network synchronised, say with the transmitting block. A more radical approach is to use a GALS (Globally Asynchronous, Locally Synchronous) paradigm where conventional, clocked blocks are connected with an independent network with its own clock(s) – or even no clocks at all. This has promise as a solution for the rapid assemble of components – there is no ‘timing closure’ issue – but is still in its infancy.

Asynchronous Design

Many early computers did not have clocks. Although synchronous design dominates the industry now it is still possible to build systems that do without a clock. (Considerable work on this was done here in the 1990s and some still persists.) Research is still ongoing.

Synchronous design dominates the industry. It simplifies the design problems and is well-supported by CAD tools. This is unlikely to change in the near future.

However, the problems of maintaining synchronous operation are getting worse! Note, for example, that PC chips have largely stopped getting faster. Clocks need ever wider distribution and, as geometries shrink, manufactured transistor sizes vary by greater percentages resulting in increasingly unpredictable timings. It may be that ‘self-timed’ circuits will be the only way to exploit the fastest devices and there could be a resurgence of clock-free logic in the future.