Clocking

The synchronous model is an enormous simplification in designing working Finite State Machines. 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 assuming the clock reaches both (all) devices ‘simultaneously’. i.e. minimal clock skew

Clock distribution

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

- The difference in arrival times is referred to as ‘clock skew’.
- In almost all practical cases, it must be minimised.

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

This is a big task for ‘Place and Route’: fortunately, tools will help with this.

The classic ‘H-tree’ structure is one method of trying to distribute a clock across a chip with minimal skew. Note, all paths are the same length.

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.

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.
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

Disadvantage

- Should give a conservative upper bound
- 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.

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.
Time Stealing

(Not something obtained with a simple design flow.)

This is a technique to get a bit more performance out of a pipeline which is not perfectly balanced.

In the first example the critical path is constraining the clock period; the second stage will evaluate before the result is needed and the result is delayed by 200 ps at the register input. This is the 'normal' type of circuit.

The second example deliberately introduces some skew into the clock to one stage. If clocked at 1 GHz this would mean that there was a 100 ps wait for this register but, because it was clocked ‘late’ the delay to the next register would be smaller (only 100 ps, not 200 ps in the example). The clock period can now be reduced to 900 ps and all the constraints can still be met; the pipeline goes 11% faster.

This is not a trivial process however; there are difficulties:

- Standard design flows will not do this.
- There needs to be a well-controlled delay in the clock distribution tree, which is difficult to make reliable.
- There is a stricter hold-time bound on the ‘lengthened’ stage; it must be guaranteed that no logic path will take less than 100 ps or that data may be captured by the same (but delayed) clock edge which initiated it.

“Time borrowing”

Basically the same principle as time stealing but exploiting transparent latches rather than edge-triggered flip-flops (see “Two-phase clocks”).

Because a latch is transparent for some time it allows an early-arriving result into the next stage before an edge-triggered device would. Thus any time spare in one stage will be exploited – automatically – by the next stage.

[Don’t get too concerned about the names of these techniques; concentrate on the principle!]

Wave pipelining

A “wave pipeline” is a pipeline without latches to re-synchronise data elements; inputs change periodically and “waves” of evaluation activity chase each other through the logic.

- In principle a wave pipeline could be the fastest means of implementing logic.
- In practice it is very hard\textsuperscript{1} to ensure that all parallel logic operations take ‘exactly’ the same time; failure means that part of one data element may catch up with its predecessor, causing a failure.

\[ \mathcal{O} \text{ As the manufacturing variation of IC devices increases this gets even harder.} \]

Wave pipelining is mentioned here to for completeness rather than as a suggested technique.

---

\textsuperscript{1} Read as “nearly impossible”
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

Frequency and phase

It’s only meaningful to talk about frequency with respect to repetitive signals. The frequency of a clock is the reciprocal of its period.

\[ f = \frac{1}{T} \]

With two or more signals there may be a phase relationship.

Same frequency, different phase.

Harmonic frequencies (phase relationship is fixed)

Non-harmonic frequencies: phase relationship drifts

It is increasingly common to have blocks running at different frequencies or possibly the same frequency, but ‘uncertain’ phase.

- Sometimes just reduce (divide) master clock
- Sometimes have separate clocks.

Oscillators

Crystal oscillators

The normal clock source for digital logic is a crystal-controlled oscillator. These use vibrations in a carefully machined (piezo electric, usually quartz) crystal to stabilise an electrical oscillator circuit. Without any special care a frequency within about 50 ppm is usual. If it matters much greater stability is, of course, achievable as is demonstrated by ‘quartz clocks’.

- No two independent oscillators will run at exactly the same frequency.
- If a constant phase relationship is required a single oscillator must be used.

Frequency examples

- Digital logic is operated at frequencies of several GHz
  - For ASIC design, typically think 100s MHz
- Humans tend to prefer simple numbers such as 20 MHz
- A ‘serial line’ (old fashioned now) has standard baud rates of 9600, 19200, 38400, 115200, … Hence multiples of such frequencies are not uncommon.
  - Example: 18.432 MHz = 30 * 16 * 38400 = 10 * 16 * 115200
- USB uses bit rates of 12 Mb/s (USB 1), 480 Mb/s (USB 2)
- In I/O applications there is commonly some tolerance.
  - E.g. RS232 ± a few percent
  - USB 480.00 Mbit/s ±500 ppm, 12.000 Mbit/s ±2500 ppm

Clocking and power

The clock network is a significant source of power dissipation. The power used is (effectively) proportional to clock frequency. Thus it makes no sense to clock a circuit faster than is necessary. Clock gating may be introduced to stop clocks when a block is unused – but this should be done with caution!

1. Equivalent to about 4 s error per day.
Crossing clock domains

There are various possibilities for relationships between clocks.

- **Synchronous circuits avoid this difficulty**
- **Isochronous** circuits have a known, constant, phase relationship
  - Maybe with blocks with **harmonic** frequencies
  - This may be exploited (with care!) in inter-block communication
- **Asynchronous clock sources cause problems!**
  - Sending signals between asynchronous domains is **impossible with 100% reliability.**
  - At some stage a flip-flop set-up/hold constraint will be violated.
  - We can make the probability of failure **very small.**

There is also the need for **arbitration:** *which* receiver cycle did the data arrive in?

---

**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-flop 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. A possible metastable flip-flop should only go to one place.

**Synchronisers**

A typical synchroniser looks like this:

Operation is simple.
If the first flip-flop latches a valid level the second one copies this one clock period later.
Else the first flip-flop may go metastable but has a whole clock period to resolve to a digital state. As the violation is caused by an input data transition the chosen state will determine whether the data changed before or after the clock.
If determined that the data changed ‘after’ the clock then it will be picked up on the next clock edge.

The first flip-flop probably doesn’t remain metastable for a whole clock period. The probability depends on the properties of the flip-flop and the length of the clock period.

If the flip-flop doesn’t resolve in time it will be forced to a digital state on the next clock edge – but the second flip-flop may go metastable.

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.

[Remember that 3 GHz translates to $3 \times 10^9$ clocks/second or about $10^{19}$/century.]
Crossing clock domains

There is no need to synchronise every signal crossing a boundary explicitly.

If the request is synchronised, accompanying data will have had plenty of time to arrive.

When crossing a clock boundary, there is always:

- some latency
- a chance of failure due to persistent metastability
  - small: may be reduced by adding extra flip-flops
  - special flip-flops which resolve faster may be available
    (though not from logic synthesis!)

Crossing timing domains in the lab.

The system we are constructing has been kept as synchronous as possible. Thus the master frequency is set by the pixel clock and the drawing engine is run at the same rate. There is, however, an asynchronous input in terms of the processor bus, which is governed by a completely separate clock.

The bus arriving from the ARM is an ‘asynchronous’ bus. In this context this means there is no clock signal within the bus. Timing is provided by pulses on control signals, the length of which is governed by the bus master (i.e. the processor). This type of bus is a typical – arguably ‘old fashioned’ – interface used by many memory and I/O devices. The various parameter registers are therefore built as transparent latches, enabled by the strobe pulses.

Most of the time, writing to the interface has no effect on the clocked part of the circuit. Parameters are set up but not yet read. This happens on ‘software’ time scales where it is easy to be confident the values will be stable long before they are used. Synchronization is therefore unnecessary.

When a command is issued a signal must cross into the clocked domain. In this case the synchroniser shown here is used.

```verilog
always @(posedge up_nwr, posedge cmd_ack) begin
  if (cmd_ack) go <= 0;
  else if (!uP_nws && (uP_address == 6'h08)) go <= 1;
end
always @(posedge clk, posedge cmd_ack) begin
  go_1 <= go;
  cmd_req <= go_1;
end
```

The operation is triggered by the end of the write pulse which allows time for data to be propagated through transparent latches in the same cycle. cmd_ack is a one clock long pulse in response to an accepted cmp_req from the synchronous side.

There is an assumption that a second write will not occur ‘too soon’. This can be prevented by, for example, checking the ‘go’ signal in software as a status bit.

Asynchronous arbitration

It is possible to enter an asynchronous domain with 100% reliability using an arbiter or mutual exclusion element. This is a cell which determines which of its (usually two) inputs arrived ‘first’. It achieves reliability by detecting metastability and delaying its decision until this is resolved.

Unfortunately the time taken to make a decision is unbounded so this process could always take more than a clock period – however long that is.
Crossing clock domains

Synchronisers introduce **latency** and may ‘cripble’ performance:

- Synchronising every item: low bandwidth
- Buffer ‘packet’: longer latency, higher bandwidth
- Read/write FIFO: low(ish) latency, high bandwidth – more complex

More buffers reduce waiting time

Various solutions are possible, depending on requirements & complexity.

**DANGER**

If, as is likely, \( w \) (for example) is generated from combinatorial logic it could glitch to the wrong value during evaluation. If such a glitch is captured by the other clock, all sorts of problems may occur! The flip-flops filter out any glitches.

**Optimisation?**

Removing the flip-flops delaying \( a \Rightarrow b \) and \( w \Rightarrow x \) would reduce the cycle time.

**Buffering**

A circuit can pass one ‘thing’ per clock cycle to another circuit in the same clock domain.

Synchronising latency will apply to every ‘thing’ passed across an interface between clock domains. This reduces the communication bandwidth considerably (in things/cycle).

Here are a couple of (related) techniques to achieve higher bandwidth across the interface.

- **Pack many bytes into a ‘thing’**. Fill up a ‘bucket’ (RAM) of data then signal its transfer at the end. There is one synchronisation penalty for the bucketload which is shared by all the data. The disadvantage of this is that the latency is increased because the bucket must be filled *then* the transfer requested, so the first datum takes longer to be received (although they come close together after that).

- As above but **double buffer**. Fill up a bucket and notify the receiver that it’s ready. Whilst that is synchronising and being emptied, fill up the next one. The disadvantage is that more (independent) RAMs are needed; the advantage is increased bandwidth, closer to the maximum rate (which is the slower of the corresponding processes).

A **decoupling FIFO** can extend the concept further. Conceptually this is a bit harder to conceive. Think of a dual-port RAM (you could build it out of flip-flops) where the transmitter writes to successive locations and the receiver subsequently reads them at its own rate. Every time a write completes a ‘counter’ is incremented (Tx clock) and when a read completes it is decremented (Rx clock). The control logic does need a synchroniser but synchronisation is not necessary every cycle: for example if the FIFO contains four data, following a read it contains at least three (more if writes are ongoing) so no need to check before reading the next one. This can be complicated to build but can offer close to maximum throughput with close to minimum latency.
Changing frequency

Reducing frequency – by an integer factor – is easy.

- Note that dividing by an odd number will result in an uneven duty-cycle
  - This may or may not matter to you
- The output clock will have a fixed (unknown) phase relationship with the input

Increasing frequency is more difficult: use a Phase-Locked Loop (PLL)

These include some mixed-signal (analogue) components … … but can usually be bought-in from a specialist designer.

Phased-Locked Loops (PLLs)

A PLL is a machine capable of matching the frequency of an input signal.

**Everyday example**

Consider a television set. It must display images at the same rate as they are broadcast. Thus it needs synchronisation information so that it can adjust its internal timing to match the transmitter. Of course, in modern sets at these slow speeds this can be done digitally by varying the number of ‘local’ clock cycles in each line, frame, etc. slightly.

**Clock multiplication**

The slide shows a clock multiplier which works by matching a division of an output clock to an input reference.

\[
\frac{f_{\text{out}}}{N} = f_{\text{in}} \quad \therefore f_{\text{out}} = N \times f_{\text{in}}
\]

**LPF – Low Pass Filter**

A typical phase comparator produces pulses on its outputs which indicate which input edge came first. These need integrating (smoothing) to produce a voltage which is (approximately) stable over many clock periods.

**VCO – Voltage Controlled Oscillator**

An oscillator which runs ‘naturally’ in a certain range of frequencies which is ‘tuned’ by an analogue input voltage.

Because a PLL circuit is controlled by feedback its output frequency will vary slightly around the nominal frequency. This contributes to clock jitter – the perceived variation in clock frequency. Jitter is a Bad Thing because the logic must always evaluate within the shortest clock period (not the average) and the more variation there is the shorter this minimum time will be.

**Definitions to remember**

- Skew: the difference in arrival time of a signal at different destinations.
- Clock jitter: the variation of a clock frequency around its specified value.
Miscellany

A collection of other timing-related issues.

Timing checking tools

A number of tools exist in order to assist with timing closure. Many of these are only appropriate when a physical realisation of the chip is available.

- Static Timing Analysis (STA)
- Edge speed analysis
- Hold time checking
- Clock skew analysis
- ...

Multi-cycle paths

It is sometimes expedient (and convenient) to allow logic more than one clock period to settle. This may be sensible but you need to tell the tools.

Tools

A Static Timing Analyser (introduced earlier) will give an estimate of the critical path in a system by searching all paths between clocked registers and finding the slowest. This then sets the ‘standard’ for other logic speeds; there is (usually) no point in optimising any logic paths already faster than the critical path.

The delay of the critical path will depend on the number of serial logic gates, their type, the fanout and other factors affecting the electrical load (particularly wire lengths) and their output impedance or ‘drive strength’. All these factors go into ‘the mix’ when attempting to optimise the circuit.

Typically, synthesis tools will have options which allow the engineer to put more importance on speed, size, power etc. It may be that a circuit can be optimised for speed but this may result in it being larger or more power hungry.

Edge speeds are the time it takes a digital circuit to switch between states. They depend on the outputting gates drive and the (capacitive) load it needs to switch. Edges which are ‘too slow’ may introduce problems such as:

- induced noise near the threshold may be received and amplified
- different target gates may ‘see’ the input switch at (significantly) different times
- increased time spent near the ‘half way’ level may result in an extra power drain

Tools are available to identify any slow edges, possibly for further attention.

With ‘challenging’ speed targets a flip-flop may be designed with a data hold time longer than its propagation delay. With such it would be dangerous to connect one flip-flop output directly to another’s input. Any logic in-between will naturally act as an additional delay and help meet the true constraints. Hold-time checking will identify any remaining risks here and allow extra buffer insertion.

Note that problems with a too-long critical path may be accommodated by reducing the clock frequency. Hold-time problems are a property of the circuit and there is no cure if they appear in the chip!

Frequency and power

The majority of the power dissipation in CMOS logic is dynamic; it occurs when gates/wires switch. Thus – when executing – the power dissipation is roughly proportional to the clock frequency. Reducing the frequency saves power (dissipates less heat).

Delay lines

It is possible – and sometimes necessary – to build delays onto ASICs. An approximate delay can be produced with a ‘chain’ of inverters or buffers; the actual delay on a given design and process may vary by a factor of two or more depending on the manufacturing and operation conditions of the chip.

Precise delays need to be calibrated against a reliable reference frequency. These are typically chains of gates (as above) whose length can be altered (e.g. by multiplexing output taps) to give the nearest available approximation to the required delay. Periodic recalibration may be needed due to thermal drift.

An example would be a Delay-Locked Loop (DLL). For instance Xilinx FPGAs contain a small number of DLLs which allow the insertion of a known delay. A typical application is to delay a clock signal so that edges at the leaves of the distribution tree are in phase (via a total delay of a number of clock cycles) with an incoming reference. This effectively ‘removes’ the clock buffer delays.

Chip variation

Gate speed depends on various manufacturing and operation conditions, normally referred to as PVT for Process, Voltage, Temperature.

- Process: variation in manufacturing such as transistor doping density.
- Voltage: the supply voltage at a gate will be less than that at the chip’s pins (Ohm’s Law); this varies across the chip and may fluctuate due to other power demands elsewhere.
- Temperature: hotter is slower.

1. The user can define what “slow” means.