Lecture 7

Metastability-Containing Control Loops

Like any clock synchronization algorithm (and many other distributed algorithms), one may view the Lynch-Welch algorithm as a (distributed) *control loop*. Basically, a control loop is seeking to adjust some (measurable) variable. To this end, it repeatedly or continually takes measurements and applies according adjustments, which naturally implies a mechanism to influence the variable of interest (see Figure 7.4). As measurements and corrections may be inaccurate, and the variable is also subject to influence by some external factors, the control loop must react sufficiently quickly and accurately to maintain a desired state against such unwanted "disturbances."

More concretely, for clock synchronization, the variable is the vector of correct nodes' clock values, the regulation is performed by adjusting the clocks, the external influence is given by drifting clocks, and clock drifts and uncertainty in message delays makes measurements of clock differences inaccurate. Two important aspects of control loops is whether they are operating on a continuous or discrete variable and whether the control is applied continuously or in time-discrete steps. An example for the answer being continuous in both cases is the gradient clock synchronization algorithm: logical clocks are continuous functions, and the GCS algorithm adjusts their rates. In contrast, pulse synchronization algorithms are an example for continuous variables (pulses can occur at any real time), but discrete time steps (for each $i \in \mathbb{N}$, each correct node generates exactly one pulse event).

Remarks:

- Note that the discretization is, of course, an abstraction in itself. It is implemented in a physical — and thus, neglecting quantum mechanics, continuous — world.
- If algorithms perform complicated message exchanges and computations, seeing them as control loops is usually not useful. However, the Srikanth-Toueg and Lynch-Welch algorithms can be readily interpreted as distributed control loops.
- The corrections are not applied instantaneously. It takes time to take measurements, compute a correction, and apply it. This contributes to

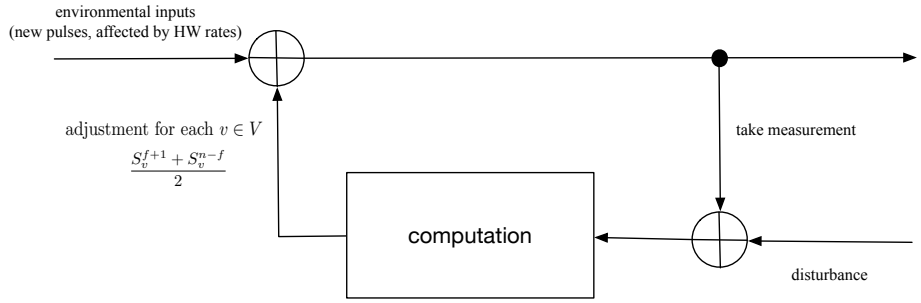

Figure 7.1: The whole network as a control loop.

the quality of control; in extreme cases, the control loop fails to produce anything close to the ideal behavior of the system.

- A lot of theory on control loops assumes very simple feedback mechanisms, like adjustments that are linear in the measured difference to the desired state of the system. This is not the case for our algorithms: the necessity to limit the influence of Byzantine nodes results in non-linear responses to the measurements in both algorithms.
- So why are we talking about control loops if we can't use the existing theory? In part to explain the lecture's title, and in part to clarify where metastability-containing circuits come into play.

7.1 Metastability in Control Loops

In the Lynch-Welch algorithm, we adjust continuous variables (when to generate pulses) for each round of the algorithm. The abstraction of rounds simplifies matters for us. Even better, each node in the system does this independently from the others, in the sense that we can interpret the algorithm at node $v \in V_g$ as a control loop in which all the other nodes are simply part of the environment, see Figure 7.2. But how do we actually decide how to adjust he clocks? After all, computers cannot *actually* use real values in computations. There are, essentially, two solutions:

1. Use an *analog* computation, in which values are represented by continuously-valued physical variables, like the charge of a capacitor or the amount of water in a bucket. (Of course, even these are discrete variables, but they are so fine-grained that it doesn't matter.)
2. Take discrete measurements, which is done by *time-to-digital converters* (TDCs). Considering the rounding error as additional contribution to δ, one then can compute a corresponding adjustment to when the next pulse occurs, just as in the analog case.

Both approaches have their pros and cons. Analog solutions typically require specialized components, can be bulky, and require more work for adapting them to different technologies. However, they can avoid metastability altogether, as they never try to map values from a continuous to a discrete range. On

the other hand, using synchronizers (i.e., time), it is straightforward to resolve metastability sufficiently reliably.

So, why not always go for the simpler, second option? The problem is that time is critical in many control loops. Recall that the Lynch-Welch algorithm guarantees a skew of $\mathcal{O}(u+(1-1/\vartheta)T)$, where T is the (nominal) duration of a round. We can choose $T \in \mathcal{O}(d)$, but d includes not only communication delays, but also computation. Thus, if we spend T_s time on synchronization, this adds $(\vartheta-1)T_s$ to the skew. On a chip, it may very well be the case that T_s becomes the larger part of T, resulting in $(\vartheta-1)T_s$ being the dominant contribution to the skew (unless local clocks are good enough). Hence, our goal for today is to remove the synchronization delay, despite sticking with the second approach!

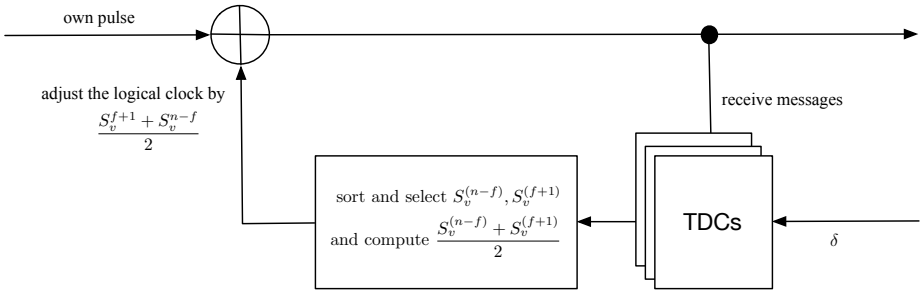

Figure 7.2: The system from the point of view of a single node — also a control loop.

7.2 First Try: Binary Counters

We need to break down the measurements and computations performed by a node executing the Lynch-Welch algorithm and implement each steps in a way that keeps (potential) metastability in check (see Figure 7.3). At each $v \in V_g$, in each round we need to

1. Send a message to each other node ϑS time after the (local) start of the round.
2. Receive the other nodes' messages and derive measurements of the difference in local time, resulting in the (unordered multi)set S_v.
3. Determine $S_v^{(f+1)}$ and $S_v^{(n-f)}$.
4. Adjust v's local clock by $(S_v^{(f+1)} + S_v^{(n-f)})/2$.

The first task is a no-brainer; we simply send the respective message $\vartheta\mathcal{S}$ local time after the time t when $L_v(t) \bmod T = \mathcal{S}$. The analysis shows that this time is unique (so the messages are indeed sent only once) and this does not require to keep track of unbounded clock values, which would be an annoying problem due to our machines having only finite memory.

The second task requires some more thought. Again, we do not want to keep track of unbounded values. Recall that Lemma 5.9 asks us to set

$$\Delta_w^v \coloneqq L_v(t) - (r-1)T - (\vartheta^2-1)\mathcal{S} - \vartheta d\,,$$

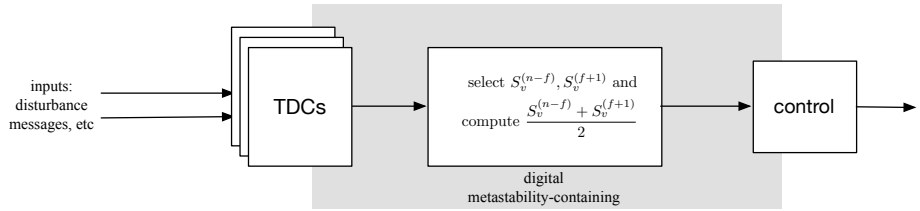

Figure 7.3: Control flow of a single node. The gray area uses digital logic and needs to contain metastability.

where t is the time when $v \in V_g$ receives the message for round r from $w \in V_g$. We also saw that all rounds are executed correctly (assuming we can implement the correct behavior of the nodes!), i.e., $t \in [p_{v,r}, \tau_{v,r}]$. This is good news, as we know that $L_v(p_{v,r}) = (r-1)T + \mathcal{S}$ and $L_v(\tau_{v,r}) = (r-1)T + (\vartheta^2 + \vartheta + 1)\mathcal{S} + \vartheta d$. Thus, we can simply start a counter at time $p_{v,r}$ and stop it at time t (when the message is received), where we know that the maximum (local) time difference which we the counter must be able to represent is $(\vartheta^2 + \vartheta)\mathcal{S} + \vartheta d$. Here, the counter is driven by the local clock and stopped by the arriving message. Thus, if the counter value at time t is c and the local time between consecutive up-counts of the counter is g, we have that

$$\begin{aligned} L_v(t) &\in [L_v(p_{v,r}) + cg, L_v(p_{v,r}) + (c+1)g] \\ &= [(r-1)T + \mathcal{S} + cg, (r-1)T + \mathcal{S} + (c+1)g]\,. \end{aligned}$$

Corollary 7.1. *Let $v \in V_g$ start a counter driven by its local clock at time $p_{v,r}$ that is stopped when receiving a message from node $w \in V_g$. If round r of the Lynch-Welch algorithm is executed correctly and the local time between up-counts of the counter is g, setting*

$$\Delta_v^w := cg - \vartheta^2\mathcal{S} - \vartheta d$$

yields an estimate satisfying $\delta \leq u + (\vartheta - 1)d + 2(\vartheta^2 - \vartheta)\mathcal{S} + g$. Moreover, $c \leq ((\vartheta^2 + \vartheta)\mathcal{S} + \vartheta d)/g$.

Pretty straightforward, so all we need now is a fast counter, i.e., one for which g is sufficiently small to not matter much, right? The answer to that is an emphatic ***no!*** We have neglected that there is no guaranteed timing relation between the counter's up-counts and when the arrival of the message from w stops the counter. Here is a simple argument why this must potentially argue in metastability.

Lemma 7.2. *Assume that a counter is driven by a free-running clock source, started at time 0, and stopped at an arbitrary time $\tau \in (0, t_{\max}]$ (where $t_{\max} \geq g$ and the counter increments every g time). Let $s(\tau, t)$ be the $k \in \mathbb{N}$ bits stored in the counter's registers at an time $t > t_{\max}$ for a given τ. If this state is a continuous function of τ (w.r.t. to the standard topologies on $\mathbb{R}$ and $\{0,1\}^k$), then we cannot have that $s(\tau, t) \in \{0,1\}^k$ for all τ.*

Proof. Assume for contradiction that for any τ, $s(\tau, t) \in \{0,1\}^k$. As $t_{\max} \geq g$, this implies that there are choices $0 \leq \ell_0 \neq r_0 \leq t_{\max}$ so that $s(\ell_0, t) \neq s(r_0, t)$.

Now we apply the technique of nested intervals. For $i \in \mathbb{N}$, set $\tau := (\ell_{i-1} + r_{i-1})/2$. Clearly, $s(\tau, t) \neq s(\ell_{i-1}, t)$ or $s(\tau, t) \neq s(r_{i-1}, t)$. In the former case, set $\ell_i := \ell_{i-1}$ and $r_i := \tau$, otherwise $\ell_i := \tau$ and $r_i := r_{i-1}$. We have that

- The sequence $(\ell_i)_{i\in\mathbb{N}}$ is increasing and upper bounded by r_i for any $i \in \mathbb{N}$, hence it converges to some value $\ell^* \leq \inf_{i\in\mathbb{N}}\{r_i\}$.
- The sequence $(r_i)_{i\in\mathbb{N}}$ is decreasing and lower bounded by ℓ_i for any $i \in \mathbb{N}$, hence it converges to some value $r^* \geq \sup_{i\in\mathbb{N}}\{\ell_i\}$.
- We have that $\ell^* = r^*$, as $\lim_{i\to\infty}(r_i - \ell_i) = 0$.
- By continuity of $s(\cdot, t)$, we have that $s(\ell^*, t) = \lim_{i\to\infty} s(\ell_i, t)$. As $\{0, 1\}^k$ is a discrete space, this means that there is some $i_\ell \in \mathbb{N}$ so that $s(\ell_i, t) = s(\ell^*, t)$ for all $i \geq i_\ell$.
- Likewise, there is some i_r so that $s(r_i, t) = s(r^*, t)$ for all $i \geq i_r$.
- We have that $s(\ell_i, t) \neq s(r_i, t)$ for all $i \in \mathbb{N}_0$ by construction.

Altogether, we arrive at the contradiction that, for any $i \geq \max\{i_\ell, i_r\}$, it holds that $s(\ell_i, t) \neq s(r_i, t) = s(r^*, t) = s(\ell^*, t) = s(\ell_i, t)$. □

Remarks:

- If you are puzzled by the lemma requiring the "standard topologies," don't worry about it. On $\mathbb{R}$, this simply means the open and closed sets you know. On $\{0, 1\}^k$, just intersect the open and closed sets in $\mathbb{R}^k$ with $\{0, 1\}^k$ to get the open and closed sets, respectively. As a ball of radius smaller than 1 around a point in $\{0, 1\}^k$ just contains the point, this means that any convergent series becomes constant at some point. This is what we used in the proof.
- These choices of topologies actually make sense. Any physical circuit will respond to continuous changes of its input with continuous changes of the output. However, we want stable and clearly distinguishable values in our registers. This means to consider clearly separated regions of the state space: The "0-region" of a register's (physical) state space should be clearly separated from its "1-region." This separation means that a small change cannot make the register "jump" from the 0- to the 1-region — which is reflected by the discrete topology on $\{0, 1\}$.
- By inserting M as a third value covering the "gap" between 0 and 1, we can properly reflect that circuits cannot do this job. In the topology, this is reflected by the fact that no matter how small a ball becomes, it doesn't separate the 0- and the M-region of the register's state space. We defined that M stands for *any* state that is not in the 0- or the 1-region!

Does this mean we're in trouble? In the previous lecture we saw that we can deal with metastability to some extent. Unfortunately, following conventional wisdom won't work here.

Corollary 7.3. *Consider the same setting as in Lemma 7.2. If the counter uses standard binary encoding and $t_{\max}$ is large enough for it to count up to 2^b, $b \in \mathbb{N}_0$, then we can force the counter register holding the $(b+1)$-least significant bit to be* M *at any time $t > t_{\max}$.*

Proof. We use essentially the same argument, but we start from more specific times ℓ_0 and r_0. As the counter can count up to 2^b, we can choose ℓ_0 and r_0 such that $s(\ell_0, t) = 0 \dots 01 \dots 1$ and $s(r_0, t) = 0 \dots 010 \dots 0$, where we wrote the least significant bits to the right and in each case the identical bits to the right are k many. This follows from the fact that the counter increment from 2^{b-1} to 2^b must change the register states between these two (stable) states. Now we can construct our nested intervals by performing our case distinction according to the $(k+1)^{th}$ bit (counting from the least significant one). By the same arguments as before, we obtain a time at which the bit cannot be stable and therefore must be M. □

Remarks:

- Unless one is very careful when implementing the counter, things actually get worse: we may end up with state $0 \dots 0\mathrm{M} \dots \mathrm{M}$. In case the full range of the counter is utilized, we may face a memory state of $\mathrm{M} \dots \mathrm{M}$!
- Think about this for a second. We started with being uncertain whether an up-count of the counter took place or not, because the counter was stopped in the middle of an increment. But we lost *all* information about the relative timing of the start of the counter and the stop signal!
- Even if the counter was particularly cleverly implemented, Corollary 7.3 shows that we might end up with very wrong encoded values.
- The problem here lies with the encoding. When containing metastability, the encoding matters!

7.3 Second Try: Unary "Counters"

We need to look for an encoding without such flaws. A very simple solution is to use a unary encoding. In a B-bit unary code, $k \in [B+1]$ is represented by $1^k 0^{B-k}$. A unary code "counter" is implemented by a *delay line,* which consists of a sequence of B buffers of uniform delay g, where we connect to each stage the set input of a register (which is initialized to 0). The counter is stopped by latching all registers on occurence of the stop signal. When g is large enough to guarantee that only a single register is unstable (transitioning) at any given point, at most one register ends up in a metastable state when stopping the counter. This is a sensible measure in most cases, as otherwise even after stabilization we may end up with a stored string like 11101000. Basically, we can't make the measurement more accurate than imposed by the speed at which registers can be set.

Alright, we measured time differences in terms of unary encodings, $v \in V_g$ has for each $w \in V_g$ stored a time difference in unary, i.e., a B-bit string of the form $1 \dots 10 \dots 0$ or, possibly, $1 \dots 1\mathrm{M}0 \dots 0$. We refer to these strings as s_v^w, and can easily translate them to the measured time difference by multiplication

with g (up to an error of up to roughly g), where M can be interpreted either way.

Our next step is to determine which of these strings represent $S_v^{(f+1)}$ and $S_v^{(n-f)}$, respectively. One way of doing this is to sort the strings. For this to be meaningful, we need to give a total order on the potential input strings. The only sensible order is in accordance with the time measured.

Definition 7.4 (Total Order of Inputs for Unary Encoding)**.** *Consider the set of strings*

$$U_B \coloneqq \{1^k 0^{B-k} \mid k \in [B+1]\} \cup \{1^k \mathrm{M} 0^{B-k-1} \mid k \in [B]\}\,.$$

For $x, y \in U_B$,

$$x \leq_U y \Leftrightarrow \begin{cases} k \leq k' & \text{for } x = 1^k 0^{B-k} \text{ and } y = 1^{k'} 0^{B-k'} \\ k \leq k' & \text{for } x = 1^k 0^{B-k} \text{ and } y = 1^{k'} \mathrm{M} 0^{B-k'-1} \\ k \leq k' & \text{for } x = 1^k \mathrm{M} 0^{B-k-1} \text{ and } y = 1^{k'} \mathrm{M} 0^{B-k'-1} \\ k < k' & \text{for } x = 1^k \mathrm{M} 0^{B-k-1} \text{ and } y = 1^{k'} 0^{B-k'-1}. \end{cases}$$

A crucial observation is that this order is also sensible in another regard: When resolving metastability, a string does not "pass" any stable strings in the order. We can apply the results from the previous lecture to see that sorting according to this order is indeed possible with a circuit.

Lemma 7.5. *Given n strings from U_B, there is a circuit sorting them according to the order from Definition 7.4.*

Proof. We claim that the metastable closure of the function sorting the stable inputs sorts according to the order from Definition 7.4. The statement of the lemma then follows from Theorem 6.6.

To see this, sort a fixed set of input strings in accordance with the order and consider the i^{th} output string. If it is stable, observe that picking arbitrary stabilizations for the other strings and sorting accordingly will not change the string in position i, as stabilizing a string $1^k \mathrm{M} 0^{B-k-1}$ does not move it "past" any stable string in the order. On the other hand, if the string is not stable, i.e., $1^k \mathrm{M} 0^{B-k-1}$ for some $k \in [B]$, observe that stabilizing all input strings by replacing M with 0 results in the sorted sequence having $1^k 0^{B-k}$ on position i in the sorted list (as nothing moves past stable strings). Likewise, stabilizing all input strings by replacing M with 1 results in the sorted sequence having $1^{k+1} 0^{B-k-1}$ in position i, so bit $k+1$ of the i^{th} output must be M. Any other stabilization will result in either $1^k 0^{B-k}$ or $1^{k+1} 0^{B-k-1}$ on position i. The claim follows, completing the proof. □

However, using the construction from Theorem 6.6 would result in a circuit of exponential size, so let's be more clever. In absence of metastability, *sorting networks* are simple and fast solutions to compute what we need.

Definition 7.6 (Sorting Network)**.** *An n-input sorting network consists of n parallel* wires *oriented from left to right and a number of* comparators *(cf. Figure* **??***). Each comparator connects two of the wires, by a straight connection orthogonal to the wires. Moreover, no two of the comparators connect to the same point on a wire.*

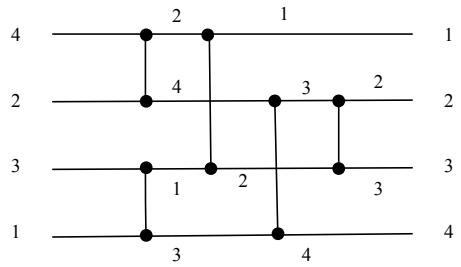

Figure 7.4: A sorting network with for inputs. Each comparator performs a compare and (if necessary) swap operation of its two inputs. The outputs are shown for the input sequence $(4, 3, 2, 1)$, whose order needs to be reversed.

A sorting network is fed an input from a totally ordered set to the start of each wire. Each comparator takes the two inputs provided by to it, outputting the larger input to the top wire and the smaller input to the bottom wire. A correct sorting network guarantees that for any choice of inputs, the outputs are the sequence resulting from ordering the inputs descendingly from top to bottom.

Sorting networks are understood very well. Constructions that are simultaneously (asymptotically) optimal both with respect to size — the total number of comparators — and depth — the maximum number of comparators "through" which a value passes — are known. Conveniently, sorting networks are correct if and only if the correctly sort 0s and 1s, so it suffices if we can figure out how to implement a comparator that correctly sorts two values according to our chosen order.

Lemma 7.7. *A correct comparator implementation for unary encoding is given by the bit-wise* Or *for the upper and the bit-wise* And *for the lower output.*

Proof. Follows from the behavior of the basic gates and a case distinction. □

With sorting in place, we can determine $S_v^{(f+1)}$ and $S_v^{(n-f)}$; refer to the encodings of these values as $s_v^{(f+1)}$ and $s_v^{(n-f)}$, respectively. It remains to perform the last step, the phase correction. One solution would be an analog control of the oscillator that serves as the local clock of v. Unfortunately, such an approach is too slow or too inaccurate in practice; either would defeat the purpose of our approach. A fast "digital" solution is to have the local clock drive a counter that basically counts modulo T (where T is represented as a multiple of the time for a counter increment) and adjust this counter. Unfortunately, this is unsafe when the adjustment values suffer from potential metastability: The counter registers could become metastable, causing all kinds of problems.

Of course, we could wait for stabilization first and *then* apply the corrections to such a counter. But in that case we wouldn't have to jump through all these hoops in the first place — if we're not able to apply the computed phase correction right away, we could have waited for stabilization *before* computing it, without losing time and saving us a lot of trouble. There is something else we can do, however. We can use the unary encoded values in a delay line to shift the clock in a safe way despite metastability. Of course, we cannot have a

delay line for each round of the Lynch-Welch algorithm (that would be infinite memory again!), but we can use a few in a round-robin fashion. The one which was written the longest time ago then has stabilized with sufficiently large probability to risk transferring the respective phase shift into our counter — while being used to shift the clock, the registers of the delay line have simultaneously operated as a synchronizer! See Figure 7.5 for an overview of the circuit.

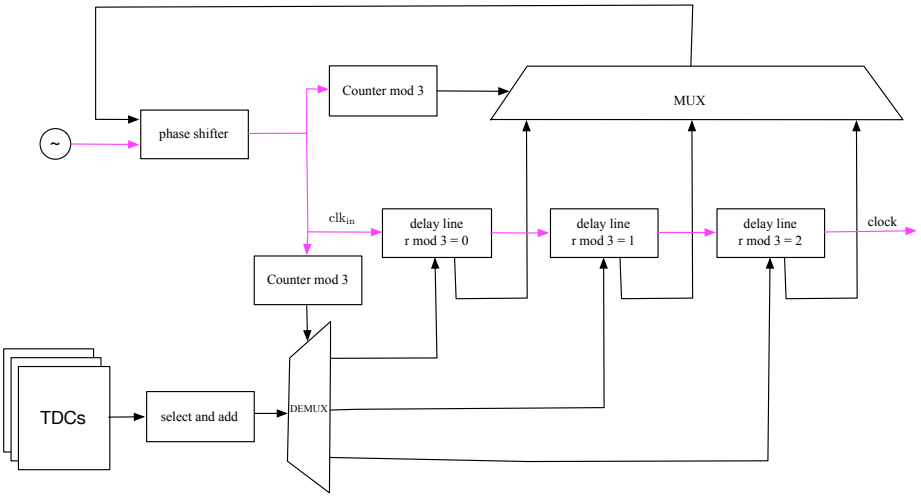

Figure 7.5: Rough overview of a circuit using a (non-containing) phase shifter and several delay lines to perform the phase shifts required by the Lynch-Welch algorithm. The delay lines are used in a round-robin fashion. In between two consecutive clock pulses, the current value held by the delay line which is to be rewritten yet is provided to the phase shifter as input, it adjusts its internal counter accordingly (making the phase shift permanent), and the registers of the delay element are latched to the current output of the computational logic. All this needs to be performed in the right order and be complete before the next pulse propragates through the phase shifter and the delay lines; the complete design requires additional circuitry ensuring this and a corresponding timing analysis.

There's still a catch: As we may have a metastable register in the delay line, the respective AND gate will output a bad signal when the clock flank arrives. This would be remedied shortly after, when the delayed clock signal reaches the next stage (with a stable 0 in the register), as then the OR will have a stable input. The solution is a *high-threshold* inverter, which switches from output 1 to output 0 at a higher voltage threshold, thus "masking" the bad medium voltage. Figure 7.6 shows how the resulting delay lines look like.

Remarks:

- This works, but is still inefficient. Unary encodings are exponentially larger than binary encodings!
- Let's do better, using an encoding without redundancy that also changes only a single bit on each up-count!

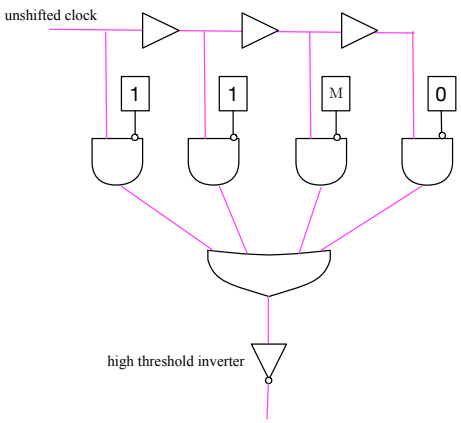

Figure 7.6: Straightforward delay line implementation. The high-threshold inverter at the output ensures that metastability is "masked," effectively transforming it into a (potentially) late, but clean transition. As a metastable register may stabilize at any time (and to either value), this may result in any delay between what we would get for a stable 0 or 1 in the register, respectively.

7.4 Third Try: Gray Codes

Unary encoding worked, but results in large circuits. A B-bit unary encoding can represent only $B+1$ different values, while a binary encoding has 2^B codewords. Binary encoding causes trouble, because a bit that may become metastable *due to an interrupted up-count* makes a huge difference with respect to the encoded value. We need a code where each up-count changes exactly one bit.

Definition 7.8 (Gray Code)**.** *A B-bit Gray code $G\colon [2^B] \to \{0,1\}^B$ maps its range $[2^B]$ one-to-one to $\{0,1\}^B$, with the property that for $x, x+1 \in [2^B]$, the resulting codewords differ in a single bit.*

Transforming unary encoding to Gray code is easy, even in face of metastability. However, we need some notation.

Definition 7.9. *For $x, y \in \{0,1,\mathrm{M}\}^k$, $k \in \mathbb{N}$, set*

$$(x * y)_i := \begin{cases} 1 & \text{if } x_i = y_i = 1 \\ 0 & \text{if } x_i = y_i = 0 \\ \mathrm{M} & \text{else.} \end{cases}$$

It is easy to see that $x * y$ is the largest common predecessor of x and y with respect to $\preceq$, i.e., $x * y \preceq x$, $x * y \preceq y$, and if $z \in \{0,1,\mathrm{M}\}^k$ satisfies $z \preceq x$ and $z \preceq y$, then $z \preceq x * y$. In other words, $x * y$ is the "most stable" string so that both x and y are stabilizations of it.

Lemma 7.10. *Let $s \in U_{2^B-1}$ for some $B \in \mathbb{N}$. If $s \in U_{2^B-1} \cap \{0,1\}^{2^B-1}$, denote the encoded number by $x \in [2^B]$. For $s \in U_{2^B-1} \setminus \{0,1\}^{2^B-1}$, let $x, x+1 \in [2^B]$*

denote the numbers encoded by the stabilizations of s. For any fixed Gray code G, there is a circuit of size $\mathcal{O}(2^B)$ and depth at most $\mathcal{O}(B)$ that computes

$$\begin{array}{ll} G(x) & \text{if } s \in U_{2^B-1} \cap \{0,1\}^{2^B-1} \\ G(x) * G(x+1) & \text{if } s \in U_{2^B-1} \setminus \{0,1\}^{2^B-1}. \end{array}$$

from input s.

Proof. For bit i of the code, consider the subset of $[2^B + 1] \setminus \{0\}$, for which an up-count to the respective value changes bit i. We connect all corresponding registers by a tree of (2-input) XOR gates. Such a XOR tree implements an XOR with more inputs, i.e., it keeps track of the number of times the i^{th} bit changed. Accordingly, depending on whether the i^{th} bit is 0 or 1 in the first bit, this circuit generates the correct output or its conjugate for stable values; in the latter case, we simply add a NOT gate. The i^{th} output bit can only become M if some input to the respective XOR tree is M. However, in this case, the respective output bit transitions on the up-count corresponding to the register holding the respective bit of the unary encoding, so the output bit ought to be M.

Concerning the complexity, the total number of XOR gates needed is $2^B - 1 - B$ (the number of input bits minus the number of output bits), plus up to B NOT gates. By balancing the XOR trees, their depth becomes bounded by the logarithm of their size (rounded up). If no XOR gates are available, we can implement them by constant-sized subcircuits composed of basic gates, increasing size and depth of the circuit by constant factors only. □

Remarks:

- This is promising: $G(x) * G(x+1)$ has only a single metastable bit, as $G(x)$ and $G(x+1)$ differ only in a single bit.
- This means that there are exactly two stabilizations of $G(x) * G(x+1)$, namely $G(x)$ and $G(x+1)$. We did not lose information, and Theorem 6.6 shows that we can convert the Gray code back to unary, even with metastability!
- The circuit for this provided by the Theorem 6.6 will have exponential size, but this time this doesn't matter as much, as the output already has exponential size by itself! One can still do better (you will do so in one of the exercises).
- For this to pay of, we now need very efficient circuits for sorting Gray codes, including strings of the form $G(x) * G(x+1)$. Ordering $G(x) \leq G(x)*G(x+1) \leq G(x+1)$ and arguing analogously to Lemma 7.5, we know that we can design suitable comparators *in principle,* which then can be used in sorting networks. In the next lecture, we will find asymptotically optimal comparator circuits for a simple, convenient Gray code.

Bibliographic Notes

The concept of implementing Lynch-Welch using metastability-containing logic was proposed in [FFL18], where it was shown to be feasible. However, the underlying construction was generic (as in Theorem 6.6), resulting in large circuits.

Such circuits would incur computational delays negating the advantage of not requiring synchronizers. In [FKLP17], TDCs are given that can directly output Binary Reflected Gray Code (BRGC) with the same guarantees as provided by Lemma 7.10. This further reduces the depth and size of circuits for follow-up computations, as the conversion circuit can be skipped. Various comparators for such BRGC values have been proposed in [BLM18, BLM17, LM16]; we will discuss the currently best one next lecture. The idea for using the computed phase shifts in delay lines until they have stabilized with sufficient probability is, essentially, applied to a different problem in [FKLW18].

Asymptotically optimal sorting networks were given in [AKS83]. For a proof that sorting networks are correct if and only if they correctly sort 0-1 inputs, see [Knu98].

Bibliography

[AKS83] Miklós Ajtai, János Komlós, and Endre Szemerédi. An $\mathcal{O}(n \log n)$ Sorting Network. In *15th Symposium on Theory of Computing (STOC)*, 1983.

[BLM17] Johannes Bund, Christoph Lenzen, and Moti Medina. Near-Optimal Metastability-Containing Sorting Networks. In *Design, Automation, and Test in Europe (DATE)*, 2017.

[BLM18] Johannes Bund, Christoph Lenzen, and Moti Medina. Optimal Metastability-Containing Sorting Networks. In *Design, Automation and Test in Europe (DATE)*, 2018. To appear. Preliminary version available at https://arxiv.org/abs/1801.07549.

[FFL18] Stephan Friedrichs, Matthias Függer, and Christoph Lenzen. Metastability-Containing Circuits. *IEEE Transactions on Computers*, 2018. To appear, online first.

[FKLP17] Matthias Függer, Attila Kinali, Christoph Lenzen, and Thomas Polzer. Metastability-Aware Memory-Efficient Time-to-Digital Converters. In *Symposium on Asynchronous Circuits and Systems (ASYNC)*, 2017.

[FKLW18] Matthias Függer, Attila Kinali, Christoph Lenzen, and Ben Wiederhake. Fast All-Digital Clock Frequency Adaptation Circuit for Voltage Droop Tolerance. In *Symposium on Asynchronous Circuits and Systems (ASYNC)*, 2018. To appear. Preprint available at https://people.mpi-inf.mpg.de/~clenzen/pubs/FKLW18droop.pdf.

[Knu98] Donald E. Knuth. *The Art of Computer Programming Vol. 3: Sorting and Searching*. Addison-Wesley, 1998.

[LM16] Christoph Lenzen and Moti Medina. Efficient Metastability-Containing Gray Code 2-Sort. In *Symposium on Asynchronous Circuits and Systems (ASYNC)*, 2016.