Ned Bingham, Rajit Manohar. “QDI Constant Time Counters.” IEEE Transactions on Very Large Integrated Systems.

QDI Constant Time Counters

Ned Bingham and Rajit Manohar
Computer Systems Lab
Yale University
New Haven, CT

***Abstract*—Counters are a generally useful circuit that appear in many contexts. Because of this, the design space for clocked counters has been widely explored. However, the same cannot be said for robust clockless counters. To resolve this, we designed an array of constant response time counters using the most robust clockless logic family, Quasi Delay-Insensitive (QDI) circuits. We compare our designs to their closest QDI counterparts from the literature, showing significant improvements in design quality metrics including transistor count, energy per operation, frequency, and latency in a 28nm process. We also compare our designs against prototypical synchronous counters generated by commercial logic synthesis tools.**

Keywords—counter; constant time; asynchronous; quasi-delay insensitive; qdi

Introduction

Counters implement an important piece of functionality with widespread use in both clocked and clockless designs, playing critical roles in the control logic for power gating, clock gating, and pipeline management [7][11][3]; for timers, performance counters, and frequency dividers [5]; and for iterative arithmetic circuits [6]. Its extensive utility draws intense optimization from commercial synthesis tools (like Synopsys Design Compiler and Cadence Genus) that take great care to optimize their structures during logic synthesis.

While clocked counters have been thoroughly explored such as the increment/decrement counter in [31], the increment/write in [27], and the decrement/write in [28][30], clockless counters have not. There are many clockless logic families [22], but this paper is limited to Quasi Delay-Insensitive (QDI) design [21] which has been successfully used in the past to implement many complex integrated circuits including microprocessors [8][9][7][11], FPGAs [12] [4], and neuromorphic chips [10][2][1][5].

QDI design is widely regarded as the most robust of the families since correct operation is independent of gate delay. Circuits are partitioned into a system of components that communicate over message passing channels which are implemented by a bundle or collection of wires that carry both data and flow control information in the form of a request and an acknowledge.

This framework makes it easy to implement sophisticated control circuitry and exploit average-case workload characteristics to reduce energy usage and increase throughput. For counters, the more significant bits typically switch far less often, burning proportionally less dynamic power. This also makes it possible to carefully tune the circuit interface for specific timings. For example, a QDI counter can be designed to operate with a constant response time making its throughput independent from the number of bits. Such a clockless counter is also readily applicable to clocked environments because there is a strict upper bound on the delay between the input request and output response. Constant response time counters are not possible from standard clocked logic synthesis [24].

QDI circuits are often written in a control-flow language called Communicating Hardware Processes (CHP) described in Appendix A and then synthesized into a Production Rule Set (PRS) described in Appendix B using two basic methods.

The first, Syntax-Directed Translation [13][14], maps the program syntax onto a predefined library of clockless processes through structural induction creating a circuit that strictly respects the control flow behavior of the original program. Well formulated examples of this method are Berkel's constant response time decrementing counter with zero detection [24] and increment/decrement counter with zero/full detection [25].

The second, Formal Synthesis [18][20], iteratively applies a small set of formal program transformations like projection and process decomposition, decomposing the program until the resulting processes each represent a single pipeline stage. Then, these stages are synthesized using Martin Synthesis into production rules. This approach respects data dependencies, but not necessarily the original control-flow behavior of the specification [19]. This method was used to construct an increment/decrement counter with constant-time zero detection [32], which was then applied to power gate long pipelines in the ULSNAP processor [11].

In this paper, we use a well-known hybrid approach, Templated Synthesis [17]. First, we apply the formal transformations to decompose a CHP description for each of our robust, clockless, constant response time counters into Dynamic Single Assignment [23] CHP descriptions for each bit. Then, we apply various template patterns and micro-architectural optimizations to synthesize PRS which are then automatically verified and compiled into circuits. We compare our designs against published counters developed using Syntax-Directed Translation and Formal Synthesis, and show significant improvements in energy per operation as well as delay. We also show that our designs compare

favorably to a standard clocked counter produced by commercial synthesis tools.

Our optimization rules, listed below, build upon Andrew Lines' Templated Synthesis method, starting with a flattened DSA CHP specification of a single pipeline stage process and deriving energy-efficient high-throughput PRS.

1. *Share logic between both computation and completion detection.*
2. *Use the simpler Weak Conditioned Half Buffer template (WCHB) when possible.*
3. *Group functionally equivalent behaviors prior to circuit synthesis.*
4. *Use combinational gates when possible.*
5. *Use our new template for internal state.*

Each section will cover a different piece of functionality, giving the abstract specification in CHP and the final circuit implementation in PRS. Section 1 covers the increment, decrement, and clear commands along with constant time zero detection. Section 2 covers the read command. Section 3 covers the write command and Section 4 covers an interface circuit for the write command. Section 5 covers the stream command, and Section 6 discusses our evaluation of all of these counters, making concluding remarks. Finally, the appendices describe the notation we use.

Each counter will be named using the first letter of the commands and statuses it supports. So `idzn` would be an increment/decrement counter with zero/not zero detection. Further, an underscore separates channel boundaries. So `idzn` would be a single channel with a 1of2 encoded request: `i` and `d` and a 1of2 encoded enable: `z` and `n` while `id_zn` would have two channels each with a 1of2 request and dataless enable.

I. IDCZN: Increment, Decrement, Clear

A. Function

We'll start by assuming that the counter won't under or overflow. It starts at zero, then for every iteration the status of the counter is sent across `Lz`, a command is received from `Lc`, then the value, `vn`, is either increased by one, decreased by one, or reset to zero depending upon the command.

```
vn:=0;
*[Lz!(vn=0); Lc?lc;
  [ lc=inc → vn:=vn+1
  ▯ lc=dec → vn:=vn-1
  ▯ lc=clr → vn:=0
  ]
 ]
```

To derive an implementable process for the least significant bit, we start by separating the least significant bit, `v0`, of the value of the counter from the remaining bits, `vn`. This requires that we implement the carry circuitry for increment, decrement, and clear. If `Lc` is increment and `v` is `1` or `Lc` is decrement and `v` is `0`, the increment or decrement command should be carried to the remaining bits. Otherwise, the remaining bits are left unchanged. Either way, the value of `v` flips. If `Lc` is clear, then `v` will be set to `0`. If the remaining bits are already zero, then they will be left unchanged. Otherwise, they will also be set to zero.

```
v0:=0, vn:=0;
*[Lz!(v0=0 ∧ vn=0); Lc?lc;
  [ lc=inc → [ v0=0 → v0:=1
             ▯ v0=1 → v0:=0, vn:=vn+1 ]
  ▯ lc=dec → [ v0=0 → v0:=1, vn:=vn-1
             ▯ v0=1 → v0:=0 ]
  ▯ lc=clr → v0:=0; [ vn≠0 → vn:=0
                    ▯ vn=0 → skip ]
  ]
 ]
```

Then, we introduce two new channels: `Rc` to communicate the carried command (inc, dec, clr) and `Rz` to respond with the resulting status (zero, not zero). This removes all direct data dependencies between `v0` and `vn` so that we can apply projection.

```
v0:=0, vn:=0; (Rz!vn=0 ‖ Rz?rz);
*[Lz!(v0=0 ∧ rz); Lc?lc;
  [ lc=inc → [ v0=0 → v0:=1
             ▯ v0=1 →
               v0:=0; Rc!inc; Rz?rz ‖
               Rc?rc; vn:=vn+1; Rz!vn=0 ]
  ▯ lc=dec → [ v0=0 →
               v0:=1; Rc!dec; Rz?rz ‖
               Rc?rc; vn:=vn-1; Rz!vn=0
             ▯ v0=1 → v0:=0 ]
  ▯ lc=clr → v0:=0;
             [ ¬rz → Rc!clr; Rz?rz ‖
                     Rc?rc; vn:=0; Rz!true
             ▯ rz → skip ]
  ]
 ]
```

Now, we can project the process into one that implements only the least significant bit of the counter with variables `v0, lc, rz` and one that implements the remaining bits with variables `vn, rc`.

```
v0:=0; Rz?rz;
*[Lz!(v0=0 ∧ rz); Lc?lc;
  [ lc=inc → [ v0=0 → v0:=1
             ▯ v0=1 → v0:=0; Rc!inc; Rz?rz ]
  ▯ lc=dec → [ v0=0 → v0:=1; Rc!dec; Rz?rz
             ▯ v0=1 → v0:=0 ]
  ▯ lc=clr → v0:=0; [ ¬rz → Rc!clr; Rz?rz
                    ▯ rz → skip ]
  ]
 ] ‖
vn:=0; Rz!vn=0;
*[Rc?rc;
  [ rc=inc → vn:=vn+1; Rz!vn=0
  ▯ rc=dec → vn:=vn-1; Rz!vn=0
  ▯ rc=clr → vn:=0; Rz!true
  ]
 ]
```

The specification for the remaining bits is left unaffected, and each bit has four channels: `Lc` and `Lz` for the command and counter status and `Rc` and `Rz` to carry the command to and receive the status from the remaining bits. We can continue executing this sequence of transformations recursively on the remaining bits to formulate an N-bit counter.

The value pending on the Rz channel can be observed without executing a communication event by using a data probe as indicated by $\overline{Rz}$. This allows us to rotate the communication actions on Rz so they always occur right before the associated communication on Rc. Finally, we flatten the specification into DSA format.

```
v:=0;
*[ Lz!(v=0 ∧ Rz); Lc?lc;
   [ lc=inc ∧ v=1   → v:=0; Rz?; Rc!inc
   ▯ lc=inc ∧ v=0   → v:=1
   ▯ lc=dec ∧ v=0   → v:=1; Rz?; Rc!dec
   ▯ lc=dec ∧ v=1   → v:=0
   ▯ lc=clr ∧ !Rz   → v:=0; Rz?; Rc!clr
   ▯ lc=clr ∧  Rz   → v:=0
   ]
 ]
```

Because Lc and Lz, and Rc and Rz always communicate together, they can be merged into counter-flow channels L and R with the command encoded in the request and the zero status encoded in the acknowledge as shown below.

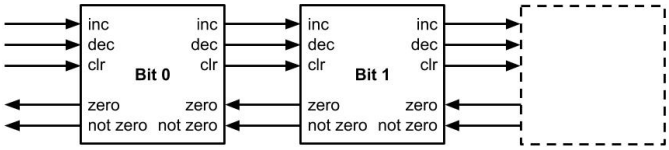

Fig. 1. The idczn counter decomposed into processes.

However, our counter must be of finite size meaning we'll need to cap it off. We'll do this with a circuit attached to the most significant bit (MSB) that sinks the command on Lc and always returns true on Lz: *[Lc?; Lz!true]. This adds an overflow condition to the previous counter specification.

```
vn:=0;
*[Lz!(vn=0); Lc?lc;
  [ lc=inc → vn:=vn+1
  ▯ lc=dec → vn:=vn-1
  ▯ lc=clr → vn:=0
  ];
  [ vn > pow(2, bits) →
              vn:=vn-pow(2, bits)
  ▯ vn < 0 → vn:=vn+pow(2, bits)
  ▯ else → skip
  ]
 ]
```

At the moment, if the value of the counter is pow(2, bits)-1 (the value of each bit is 1), then an increment command and the resulting status signal would have to propagate across the full length of the counter. This means that the zero detection circuitry will take linear time with respect to the number of bits in the worst case.

A constant time zero detection can be implemented by adding a third state to the internal register, v. v=z represents that this and all bits of greater significance are zero, v=0 represents that this bit is zero but there is at least one of greater significance that isn't, and v=1 represents that this bit is one. Now the internal register can be used to calculate the counter status in constant time.

```
v:=z;
*[ Lz!(v=z); Lc?lc;
  [ lc=inc ∧ v=1       → v:=0; Rz?; Rc!inc
  ▯ lc=inc ∧ v≠1       → v:=1
  ▯ lc=dec ∧ v≠1       → v:=1; Rz?; Rc!dec
  ▯ lc=dec ∧ v=1 ∧  Rz → v:=z
  ▯ lc=dec ∧ v=1 ∧ !Rz → v:=0
  ▯ lc=clr ∧       !Rz → v:=z; Rz?; Rc!clr
  ▯ lc=clr ∧        Rz → v:=z
  ]
 ]
```

This increases the maximum value the finite-length counter can store before it overflows by pow(2, bits-1).

```
[ vn ≥ pow(2, bits)+pow(2, bits-1) →
          vn:=vn-pow(2, bits)
▯ vn < 0 → vn:=vn+pow(2, bits)
▯ else → skip
]
```

B. Implementation

Of the 7 conditions listed in the DSA CHP for each bit, conditions 1, 3, and 6 forward the command from Lc to Rc while 2, 4, 5, and 7 don't produce an output. All conditions always acknowledge the input. Conditions 1 and 5 always set v:=0, 2 and 3 set v:=1, and 4, 6, and 7 set v:=z. Conditions 1 through 5 always change the value of v but 6 and 7 might not. Finally R_z must be true if v=z.

We start our WCHB template by defining the rules for the forward drivers. Noticing that conditions 4 and 7 both set v:=z and don't forward the command, we can merge them into a single forward rule, Rz.

$$v1 \wedge (R_z \vee R_n) \wedge L_i \rightarrow R_i\uparrow$$
$$(v0 \vee vz) \wedge L_i \rightarrow R1\uparrow$$
$$(v0 \vee vz) \wedge (R_z \vee R_n) \wedge L_d \rightarrow R_d\uparrow$$
$$v1 \wedge R_n \wedge L_d \rightarrow R0\uparrow$$
$$R_n \wedge L_c \rightarrow R_c\uparrow$$
$$R_z \wedge (v1 \wedge L_d \vee L_c) \rightarrow Rz\uparrow$$

To understand what these production rules look like, we've rendered the production rules for $R_i\uparrow$ from above and $R_i\downarrow$ from below as a CMOS gate structure in black with combinational feedback in grey.

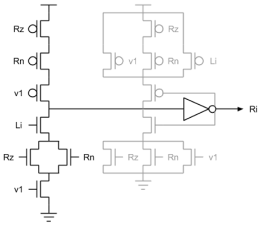

Fig. 2. The gate controlling R_i as described by the production rules.

Because there are 6 forward drivers, we'll need to use a validity tree. However, we can use this to store the next value of the internal register by defining $\overline{x0}$, $\overline{x1}$, and $\overline{xz}$. This makes the rules for the internal register smaller and frees the reset phase of the forward drivers from various problematic acknowledgment constraints.

```
R_i ∨ R0 → x̄0↓
R_d ∨ R1 → x̄1↓
R_c ∨ Rz → x̄z↓
¬x̄0 ∨ ¬x̄1 ∨ ¬x̄z → x↑
```

The checks for `v0`, `v1`, and `vz` make the input enable combinational removing the need for a staticizer and they can be minimally sized since they are not on the critical path of the gate. This kind of feedback structure is not possible in the typical WCHB template for internal state found in [17].

```
v0 ∨ v1 ∨ x → L_z↓
vz ∨ x → L_n↓
```

Before using the validity tree to set the internal register, we have to wait for the input command to go neutral. This keeps all of the forward drivers stable while the register is written. The usual template in [17] doesn't allow simultaneous read/write of the internal state. We also make these three gates combinational using minimally sized transistors to remove the need for staticizers once again.

```
¬v1 ∧ ¬v0 ∨ ¬x̄z ∧ ¬L_c ∧ ¬L_d → vz↑
¬v1 ∧ ¬vz ∨ ¬x̄0 ∧ ¬L_i ∧ ¬L_d → v0↑
¬vz ∧ ¬v0 ∨ ¬x̄1 ∧ ¬L_i ∧ ¬L_d → v1↑

(x̄z ∨ L_c ∨ L_d) ∧ (v0 ∨ v1) → vz↓
(x̄0 ∨ L_i ∨ L_d) ∧ (vz ∨ v1) → v0↓
(x̄1 ∨ L_i ∨ L_d) ∧ (vz ∨ v0) → v1↓
```

The reset phase of our forward drivers looks similar to that of a WCHB. However, checking the correct value of the internal register guarantees that the input request is neutral. This is because we check neutrality before writing the internal register and the internal register is guaranteed to change. This prevents the reset rules from becoming too long as tends to happen in a typical complex WCHB.

There are two rules, `R_c↓` and `Rz↓` from conditions 6 and 7, where this doesn't necessarily happen. Clearing an already zeroed counter isn't guaranteed change the value of the internal register in the LSB. This forces us to check `L_c` in the reset rules of the LSB. Alternatively, we can assume that clearing an already zeroed counter is an error and remove these two transistors.

```
¬R_z ∧ ¬R_n ∧ ¬v1 → R_i↓
¬vz ∧ ¬v0 → R1↓
¬R_z ∧ ¬R_n ∧ ¬vz ∧ ¬v0 → R_d↓
¬v1 → R0↓
¬R_n ∧ ¬v0 ∧ ¬v1 ∧ ¬L_c → R_c↓
¬v0 ∧ ¬v1 ∧ ¬L_c → Rz↓
```

Finally, the validity tree is reset and we can use the value of the internal register to return the status of the counter.

```
¬R_i ∧ ¬R0 → x̄0↑
¬R_d ∧ ¬R1 → x̄1↑
¬R_c ∧ ¬Rz → x̄z↑
x̄0 ∧ x̄1 ∧ x̄z → x↓

¬v0 ∧ ¬v1 ∧ ¬x → L_z↑
¬vz ∧ ¬x → L_n↑
```

The dzn and idzn variations may be derived by deleting the unnecessary rules and their associated acknowledgments.

II. IDRZN: Reading Counters

A. Function

Like the other commands, the read command will propagate from the LSB to the MSB. Each bit will send its value upon receipt the command, producing the counter value with skewed timing.

```
count:=0;
*[Lz!(count=0); Lc?lc;
  [ lc=inc → count:=count+1
  ▯ lc=dec → count:=count-1
  ▯ lc=rd  → R!count
  ];
  [ count ≥ pow(2, bits)+pow(2, bits-1) →
               count:=count-pow(2, bits)
  ▯ count<0 → count:=count+pow(2, bits)
  ▯ else → skip
  ]
 ]
```

We'll build upon the implementation of the `idzn` counter. Upon receiving a read command, it first forwards the command, then sends the bit value. Aside from that, the rest of the command and detection circuitry in a given bit is the same as the `idzn` counter.

```
v:=z;
*[ Lz!(v=z); Lc?lc;
   [ lc=inc ∧ v=1       → v:=0; Rz?; Rc!inc
   ▯ lc=inc ∧ v≠1       → v:=1
   ▯ lc=dec ∧ v≠1       → v:=1; Rz?; Rc!dec
   ▯ lc=dec ∧ v=1 ∧  Rz → v:=z
   ▯ lc=dec ∧ v=1 ∧ !Rz → v:=0
   ▯ lc=rd              → Rz?; Rc!rd; O!v
   ]
 ]
```

B. Implementation

There are two practical methods to implement this read. For each method, we'll start with the idzn counter, showing only the rules that are added or changed.

1. QDI Read

The first takes an entirely QDI approach, sending the bit values through one bit QDI channels. We'll start by adding one rule for the read command which is always forwarded and a set of rules that output the read result.

```
(R_z ∨ R_n) ∧ L_r → R_r↑

v0 ∧ O_e ∧ L_r → O_f↑
v1 ∧ O_e ∧ L_r → O_t↑
vz ∧ O_e ∧ L_r → O_z↑
```

Then, we add an extra validity check for the read result.

```
R_r ∧ (O_f ∨ O_t ∨ O_z) → y↑

v0 ∨ v1 ∨ x ∨ y → L_z↓
vz ∨ x ∨ y → L_n↓
```

The rules for the internal register remain unchanged, and the forward drivers for the read are reset normally.

$$\neg R_z \wedge \neg R_n \wedge \neg L_r \rightarrow R_r\downarrow$$

$$\neg O_e \wedge \neg L_r \rightarrow O_f\downarrow$$
$$\neg O_e \wedge \neg L_r \rightarrow O_t\downarrow$$
$$\neg O_e \wedge \neg L_r \rightarrow O_z\downarrow$$

The validity tree is reset normally, and the up-going rules for the input enable are lengthened to check for the neutrality of the read result.

$$\neg R_r \wedge \neg O_f \wedge \neg O_t \wedge \neg O_z \rightarrow y\downarrow$$

$$\neg v0 \wedge \neg v1 \wedge \neg y \wedge \neg x \rightarrow L_z\uparrow$$
$$\neg vz \wedge \neg y \wedge \neg x \rightarrow L_n\uparrow$$

2. *Bundled Data Read*

The second method latches the bit values upon receipt of the command. When the command reaches the most significant bit of the counter, it is forwarded from the counter as the request signal for the newly generated bundled data read. This mixed QDI/Bundled Data approach is a fairly rare one. Most Bundled Data circuits have extremely simple pipeline structures and most QDI circuits avoid timing assumptions like the plague.

Much like the QDI read, we'll need to add a set of rules for the read command which is always forwarded. It will be fairly simple since it doesn't interact with much of the other circuitry.

$$(R_z \vee R_n) \wedge L_r \rightarrow R_r\uparrow$$
$$R_r \rightarrow \overline{xx}\downarrow$$
$$\neg\overline{x0} \vee \neg\overline{x1} \vee \neg\overline{xz} \vee \neg\overline{xx} \rightarrow x\uparrow$$
$$\neg R_z \wedge \neg R_n \wedge \neg L_r \rightarrow R_r\downarrow$$
$$\neg R_r \rightarrow \overline{xx}\uparrow$$
$$\overline{xz} \wedge \overline{x0} \wedge \overline{x1} \wedge \overline{xx} \rightarrow x\uparrow$$

Then, if you don't care about the third value of the internal register, `vz`, we'll need rules to merge it in with `v0`.

$$D_t = v1$$
$$v0 \vee vz \rightarrow D_f\downarrow$$
$$\neg v0 \wedge \neg vz \rightarrow D_f\uparrow$$

Finally, the data, `D`, is latched using the read request.

$$O_f \vee D_f \wedge R_r \rightarrow O_t\downarrow$$
$$O_t \vee D_t \wedge R_r \rightarrow O_f\downarrow$$
$$\neg O_f \wedge (\neg D_f \vee \neg R_r) \rightarrow O_t\uparrow$$
$$\neg O_t \wedge (\neg D_t \vee \neg R_r) \rightarrow O_f\uparrow$$

This implements the most basic bundled data read which can handle another command in constant time after a read without problems unless it is another read. For two consecutive reads, the second will overwrite the latched values of the first before it finishes. So we have to delay the second read.

The easiest way is to add a communication event between the first and last bits in the counter for a read. So we'll need to modify the first bit to add this communication event.

$$G_r = R_r$$
$$G_e \wedge (R_z \vee R_n) \wedge L_r \rightarrow R_r\uparrow$$
$$\neg G_e \wedge \neg R_z \wedge \neg R_n \wedge \neg L_r \rightarrow R_r\downarrow$$

Then we'll need to modify the end cap of the counter to handle this new dependency and to forward the request signal for the newly bundled data.

$$L_r \wedge G_r \rightarrow R_r\uparrow$$
$$\neg L_r \wedge \neg G_r \rightarrow R_r\downarrow$$

$$R_e \rightarrow Ra\downarrow$$
$$\neg R_e \rightarrow Ra\uparrow$$
$$\neg Ra \wedge \neg R_r \rightarrow G_e\uparrow$$
$$Ra \vee R_r \rightarrow G_e\downarrow$$

$$Ra \vee L_i \vee L_d \rightarrow L_z\downarrow$$
$$\neg Ra \wedge \neg L_i \wedge \neg L_d \rightarrow L_z\uparrow$$

$$1 \rightarrow L_n\downarrow$$

Now subsequent commands will be delayed only if there are two conflicting reads. This allows us to reduce the energy required by the system while only suffering a minor throughput hit.

III. dwzn: Writing Counters

A. Function

The write command operates much like the first method for reading. Propagate the command through the counter and have each bit write its value upon receipt of the command.

```
count:=0;
*[Lz!(count=0); Lc?lc;
  [ lc=dec → count:=count-1
  ▯ lc=wr  → W?count
  ];
  [ count < 0 → count:=count+pow(2, bits)
  ▯ count ≥ 0 → skip
  ]
 ]
```

However, determining the location of the MSB is logarithmic with the number of bits. To ensure this doesn't hinder the performance of the counter, we will introduce a device that does this detection in parallel in worst case linear time. This way we can do operations while the zero detection for the write is taking place and the command can write `0`, `1`, or `z` directly to the internal register.

```
v:=z;
*[Lz!(v=z); Lc?lc;
  [ lc=dec ∧ v≠1       → v:=1; Rz?; Rc!
  ▯ lc=dec ∧ v=1 ∧ !Rz → v:=0
  ▯ lc=dec ∧ v=1 ∧  Rz → v:=z
  ▯ lc=wr              → Rz?; Rc!wr; W?v
  ]
 ]
```

B. Implementation

This implementation will build off the dzn counter, showing only the rules that are added or changed. The production rules for the write are structured similarly to the

read. We have a signal R_w that is always forwarded during a write, and then an input W that we save to $Rw0$, $Rw1$, and Rwz.

$$(R_z \vee R_n) \wedge L_w \rightarrow R_w{\uparrow}$$

$$W_f \wedge R_w \rightarrow Rw0{\uparrow}$$
$$W_t \wedge R_w \rightarrow Rw1{\uparrow}$$
$$W_z \wedge R_w \rightarrow Rwz{\uparrow}$$

Now, $Rw0$, $Rw1$, and Rwz stores the value to be written, allowing us to lower the input enable immediately and use the built in method to set the internal register.

$$Rw0 \vee Rw1 \vee Rwz \rightarrow W_e{\downarrow}$$
$$Rw0 \vee R0 \rightarrow \overline{x0}{\downarrow}$$
$$Rw1 \vee R_d \rightarrow \overline{x1}{\downarrow}$$
$$Rwz \vee Rz \rightarrow \overline{xz}{\downarrow}$$

To ensure that the validity, x, is acknowledged we have to check L_w when writing the internal variable.

$$\neg v1 \wedge \neg v0 \vee \neg\overline{xz} \wedge \neg L_w \wedge \neg L_d \rightarrow vz{\uparrow}$$
$$\neg v1 \wedge \neg vz \vee \neg\overline{x0} \wedge \neg L_w \wedge \neg L_d \rightarrow v0{\uparrow}$$
$$\neg vz \wedge \neg v0 \vee \neg\overline{x1} \wedge \neg L_w \wedge \neg L_d \rightarrow v1{\uparrow}$$

$$(L_w \vee L_d \vee \overline{xz}) \wedge (v0 \vee v1) \rightarrow vz{\downarrow}$$
$$(L_w \vee L_d \vee \overline{x0}) \wedge (vz \vee v1) \rightarrow v0{\downarrow}$$
$$(L_w \vee L_d \vee \overline{x1}) \wedge (vz \vee v0) \rightarrow v1{\downarrow}$$

The output signals are then reset normally using the $Rw0$, $Rw1$, and Rwz signals to check the correct value of the internal register.

$$\neg R_z \wedge \neg R_n \wedge \neg L_w \rightarrow R_w{\downarrow}$$

$$\neg W_f \wedge \neg vz \wedge \neg v1 \wedge \neg R_w \rightarrow Rw0{\downarrow}$$
$$\neg W_t \wedge \neg v0 \wedge \neg vz \wedge \neg R_w \rightarrow Rw1{\downarrow}$$
$$\neg W_z \wedge \neg v0 \wedge \neg v1 \wedge \neg R_w \rightarrow Rwz{\downarrow}$$

Then the rest of the validity tree continues as usual and the input enable rules are left unchanged.

$$\neg Rw0 \wedge \neg Rw1 \wedge \neg Rwz \rightarrow W_e{\uparrow}$$
$$\neg Rw0 \wedge \neg R0 \rightarrow \overline{x0}{\uparrow}$$
$$\neg Rw1 \wedge \neg R_d \rightarrow \overline{x1}{\uparrow}$$
$$\neg Rwz \wedge \neg Rz \rightarrow \overline{xz}{\uparrow}$$

IV. dwzn: Writing Counter Interface

A. Function

The zero detection block consumes an N bit base two integer and converts it to the three-valued format necessary for this counter.

Once again, we'll use a recursive implementation, pulling bit into its own process so that it plugs into the W channel of the writing counter. It simply propagates the zero detection from the MSB to LSB until it either reaches a non-zero bit or the LSB. If every bit of greater significance is zero and this bit is zero, then we forward $true$ on the Zo channel. If this bit is one, then we need to forward $false$.

```
*[Wi?w; Zi?z;
  [ w=0 ∧ z=0 → Zo!0; Wo!0
  ▯ w=0 ∧ z=1 → Zo!1; Wo!2
  ▯ w=1       → Zo!0; Wo!1
  ]
 ]
```

B. Implementation

With this implementation, we can take advantage of early out to get logarithmic average case complexity instead of linear. If Wi is $true$ or Zi is $false$, then we already know we need to forward $false$ on the Zo channel before we receive anything on the other channel. This allows us to break the dependency chain, reducing the average propagation time.

Upon receiving both inputs and setting the output on Wo, the input enables are lowered and Zo reset. This leaves the value on Wo unaffected while waiting for the counter, making the interface much less costly in terms of throughput and response time because it can complete its reset phase very quickly after Wo is finished.

$$Wi_e = Le$$
$$Zi_e = Le$$
$$Zo_e \wedge (Zi_f \vee Wi_t) \rightarrow Zo_f{\uparrow}$$
$$Zo_e \wedge Zi_t \wedge Wi_f \rightarrow Zo_t{\uparrow}$$

$$Zi_f \wedge Wi_f \wedge Wo_e \rightarrow Wo_f{\uparrow}$$
$$(Zi_f \vee Zi_t) \wedge Wi_t \wedge Wo_e \rightarrow Wo_t{\uparrow}$$
$$Zi_t \wedge Wi_f \wedge Wo_e \rightarrow Wo_z{\uparrow}$$

$$Zo_f \vee Zo_t \rightarrow \overline{Zv}{\downarrow}$$
$$Wo_f \vee Wo_t \vee Wo_z \rightarrow \overline{Wv}{\downarrow}$$
$$\neg\overline{Zv} \wedge \neg\overline{Wv} \rightarrow Le{\downarrow}$$

$$\neg Zo_e \wedge \neg Zi_f \wedge \neg Wi_t \rightarrow Zo_f{\downarrow}$$
$$\neg Zo_e \wedge \neg Zi_t \wedge \neg Wi_f \rightarrow Zo_t{\downarrow}$$

$$\neg Zi_f \wedge \neg Wi_f \wedge \neg Wo_e \rightarrow Wo_f{\downarrow}$$
$$\neg Zi_f \wedge \neg Zi_t \wedge \neg Wi_t \wedge \neg Wo_e \rightarrow Wo_t{\downarrow}$$
$$\neg Zi_t \wedge \neg Wi_f \wedge \neg Wo_e \rightarrow Wo_z{\downarrow}$$

$$\neg Zo_f \wedge \neg Zo_t \rightarrow \overline{Zv}{\uparrow}$$
$$\neg Wo_f \wedge \neg Wo_t \wedge \neg Wo_z \rightarrow \overline{Wv}{\uparrow}$$
$$\overline{Zv} \wedge \overline{Wv} \rightarrow Le{\uparrow}$$

V. is_zn: Streaming Counters

A. Function

Finally, there are several applications where you might need to store a large number of tokens to be released later. For that purpose, we have an interface that converts the increment/decrement counter to a increment/stream counter in which one stream command will continuously produce tokens and decrement the counter until it is empty.

```
count:=0;
*[ L?cmd;
   [ cmd=inc → count:=count+1;
     [ count≥pow(2, bits)+pow(2, bits-1) →
       count:=count-pow(2, bits)
     ▯ else → skip
     ]
   ▯ cmd=stream → Z!(count=0);
     *[ count ≠ 0 →
       count:=count-1; Z!(count=0) ]
   ]
 ]
```

The interface has three channels. The first channel, `L`, is the input request with increment or stream. The second channel, `C`, is an idzn channel that talks to the counter. The third channel, `Z`, responds with zero or not zero when the counter is being streamed. This interface is implemented by repeatedly producing decrement requests until the zero flag is set, at which point it acknowledges it's input request.

```
Cz?cz;
*[ L?cmd;
   [ cmd=inc → Cc!inc; Cz?cz;
   ▯ cmd=dec → Z!cz;
     *[ ¬cz → Cc!dec; Cz?cz; Z!cz ]
   ]
 ]
```

B. Implementation

To implement this interface, the internal loop must be flattened into its parent conditional statement. Instead of just one condition for decrement, there are now two. One for decrement zero and one for decrement not zero.

Because the rules for C_d and Z_f are the same, we make them the same node with no consequences. So this turns into a fairly simple buffer.

$$C_d = Z_f$$

$$Z_e \wedge L_f \wedge C_n \rightarrow C_d\uparrow$$
$$Z_e \wedge L_f \wedge C_z \rightarrow Z_t\uparrow$$
$$L_t \wedge (C_z \vee C_n) \rightarrow C_i\uparrow$$

$$Z_t \vee C_i \rightarrow L_e\downarrow$$

$$\neg Z_e \wedge \neg C_n \rightarrow C_d\downarrow$$
$$\neg Z_e \wedge \neg L_f \rightarrow Z_t\downarrow$$
$$\neg L_t \wedge \neg C_z \wedge \neg C_n \rightarrow C_i\downarrow$$

$$\neg Z_t \wedge \neg C_i \rightarrow L_e\uparrow$$

This is the one type of counter that is not applicable to clocked environments. Because the input must wait until the counter empty before continuing and an output is not produced on a increment, this counter cannot be clocked.

VI. Evaluation

We used a set of in-house tools to develop and evaluate these circuits. Production rule specifications are verified with a switch-level simulation which identifies instability, interference, and deadlock then automatically translated into netlists. These netlists are then verified using `vcs-hsim`. The CHP was simulated using C++ to generate inject and expect values which were tied into both the switch level and analog simulations using Python. This allowed us to verify circuit and behavioral correctness by checking the behavioral, digital, and analog simulations against each other.

To evaluate frequency and energy per operation we simulated a 1V 28nm process on 5 bit instances of each counter with a uniform random distribution of input commands. Latency was measured from the `0.5v` level of the input command to the `0.5v` level of the detection event. To get more accurate results, we protected each of the digitally driven channels with a FIFO of three WCHBs isolated to a different power source. All counters are sized minimally with a pn-ratio of `2`. In all of our implementations, we avoid using the HCTA. However, it would be fairly easy to make the necessary modifications to take advantage of it. In all implementations, we use combinational feedback for C-elements. Circuitry necessary for reset was not included in any the above descriptions.

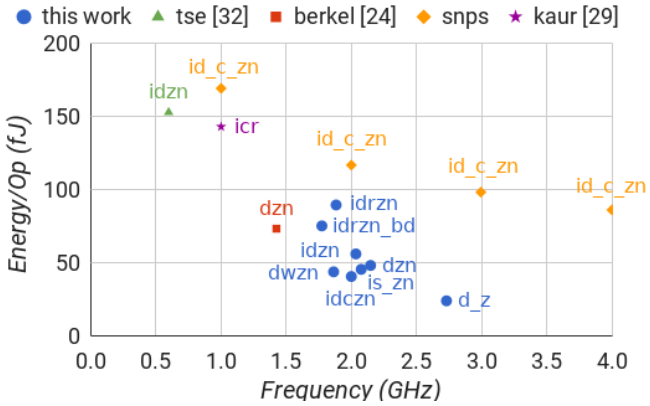

Fig. 3. Measured Performance and Energy for an array of counters.

Type	Trans	Frequency	Energy/Op	Latency
d_z	50N	2.73 GHz	24.01 fJ	N/A
dzn	102N+10	2.15 GHz	48.17 fJ	399 ps
idzn	146N+12	2.03 GHz	56.05 fJ	421 ps
idczn	174N+14	2.00 GHz	40.62 fJ	442 ps
idrzn	246N+14	1.88 GHz	89.51 fJ	441 ps
idrzn_bd	188N+32	1.77 GHz	75.20 fJ	441 ps
dwzn	192N+12	1.86 GHz	43.81 fJ	487 ps
is_zn	146N+61	2.08 GHz	45.52 fJ	139 ps

We simulated [24] and [32] in the same 28nm process with the same minimal interface elements to get as close a comparison as possible. This allowed us to identify any functional differences between the two implementations as well.

[24] was closest to the `dzn` counter that we implemented. Though ours is limited to powers of two and uses only one channel. [24] can implement any max value and all three signals are split into separate dataless channels.

[32] was closest to the `idzn` counter that we implemented. However, instead of sending the zero status before receiving a command, they send the zero status after receiving a command, though this only matters for the first command. They also split the status signal and the command into two separate channels instead of one.

Type	Trans	Frequency	Energy/Op	Latency
d_z_n[24]	117N+32	1.42 GHz	73.34 fJ	468 ps
id_zn[32]	398N+26	0.60 GHz	152.76 fJ	1150 ps

Our counter template performs better in every metric operating 1.51 times faster than [24] and 3.38 times faster than [32] using 34% less energy than [24] and 63% less energy than [32]. Furthermore, our counter template is extensible to cover much more of the design space while [24] and [32] are limited to a single problem statement.

Finally, we wrote a simple `id_c_zn` counter in Verilog and synthesized it using Synopsys Design Compiler (DC). Examining the verilog netlist, DC placed an array of clocked registers which outputs to and receives inputs from a parallel ripple-carry incrementer and outputs to a parallel zero detector. We evaluated this using the same setup that we use to evaluate the other designs and our equivalent counter uses 65% less energy at the same frequency.

Type	Trans	Frequency	Energy/Op
id_c_zn	74N	1.00 GHz	169.18 fJ
id_c_zn	74N	2.00 GHz	116.75 fJ
id_c_zn	74N	3.00 GHz	98.24 fJ
id_c_zn	74N	4.00 GHz	86.12 fJ

VII. Conclusion

This paper presents an array of QDI constant response time counters for use in clocked and clockless systems showing a frequency and energy usage superior to many other designs. However, there are still a few things left to explore.

Combinations of detection signals including full, equal, less than, and greater than have yet to be explored. These could provide useful information regarding the state of the counter to the external system. Further, at the time of design a sufficient relative timing assumption framework and toolset was not available. It is plausible that significant performance and efficiency gains could be realized by applying such a framework to the designs found in this paper. Finally, some of these optimizations can be incorporated into a logic optimization tool for designing asynchronous circuits.

Appendix

A. CHP Notation

Communicating Hardware Processes (CHP) is a hardware description language used to describe clockless circuits derived from C.A.R. Hoare's Communicating Sequential Processes (CSP) [15]. A full description of CHP and its semantics can be found in [20]. Below is an informal description of that notation listed top to bottom in descending precedence.

- **Skip** `skip` does nothing and continues to the next command.
- **Dataless Assignment** `c↑` sets the voltage of the **node** `c` to `Vdd` and `c↓` sets it to `GND`.
- **Assignment** `a := e` waits until the **expression**, **e**, has a valid value, then assigns that value to the **variable**, `a`.
- **Send** `X!e` waits until the expression `e` has a valid value, then sends that value across the **channel** `X`. `X!` is a dataless send.
- **Receive** `X?a` waits until there is a valid value on the channel `X`, then assigns that value to the variable `a`. `X?` is a dataless receive.
- **Probe** $\overline{X}$ returns the value to be received from the channel `X` without executing a receive.
- **Sequential Composition** `S ; T` executes the programs `S` followed by `T`.
- **Parallel Composition** `S ‖ T` executes the programs `S` and `T` in any order.
- **Deterministic Selection** `[G1 → S1▯...▯Gn → Sn]` where G_i is a guard and S_i is a program. A **guard** is a dataless expression or an expression that is implicitly cast to dataless. This waits until one of the guards, G_i, evaluates to `Vdd`, then executes the corresponding program, S_i. The guards must be mutually exclusive. The notation `[G]` is shorthand for `[G → skip]`.
- **Repetition** `*[G1 → S1▯...▯Gn → Sn]` is similar to the selection statements. However, the action is repeated until no guard evaluates to `Vdd`. `*[S]` is shorthand for `*[true → S]`.

B. PRS Notation

In a Production Rule Set (PRS), a Production Rule is a compact way to specify a single pull-up or pull-down network in a circuit. An **alias** `a = b` aliases two names to one circuit node. A **rule** `G → A` represents a guarded action where `G` is a guard (as described above) and `A` is a dataless assignment as described above. A **gate** is made up of multiple rules that describe the up and down assignments. The guard of each rule in a gate represents a part of the pull-up or pull-down network of that gate depending upon the corresponding assignment. If the rules of a gate do not cover all conditions, then the gate is state-holding with a staticizer.

References

[1] Akopyan, Filipp, et al. "Truenorth: Design and tool flow of a 65 mw 1 million neuron programmable neurosynaptic chip." IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 34.10 (2015): 1537-1557.

[2] Benjamin, Ben Varkey, et al. "Neurogrid: A mixed-analog-digital multichip system for large-scale neural simulations." Proceedings of the IEEE 102.5 (2014): 699-716.

[3] Virantha Ekanayake, Clinton Kelly IV, and Rajit Manohar. "BitSNAP: Dynamic Significance Compression for a Low Power Sensor Network Asynchronous Processor". Proceedings of the 11th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC), March 2005.

[4] Hill, Benjamin, et al. "A split-foundry asynchronous FPGA." Custom Integrated Circuits Conference (CICC), 2013 IEEE. IEEE, 2013.

[5] Nabil Imam, et al. "A digital neurosynaptic core using event-driven qdi circuits". Asynchronous Circuits and Systems (ASYNC), 2012 18th IEEE International Symposium on. IEEE, 2012.

[6] Navaneeth Jamadagni, and Jo Ebergen. "An asynchronous divider implementation". Asynchronous Circuits and Systems (ASYNC), 2012 18th IEEE International Symposium on. IEEE, 2012.

[7] Kelly, Clinton, Virantha Ekanayake, and Rajit Manohar. "SNAP: A sensor-network asynchronous processor." Asynchronous Circuits and Systems, 2003. Proceedings. Ninth International Symposium on. IEEE, 2003.

[8] Alain J. Martin, Steven M. Burns, Tak-Kwan Lee, Drazen Borkovic, and Pieter J. Hazewindus. "The Design of an Asynchronous Microprocessor." ACM SIGARCH Computer Architecture News 17.4 (1989): 99-110.

[9] Alain J. Martin, et al. "The design of an asynchronous MIPS R3000 microprocessor." Advanced Research in VLSI, 1997. Proceedings., Seventeenth Conference on. IEEE, 1997.

[10] Merolla, Paul, et al. "A digital neurosynaptic core using embedded crossbar memory with 45pJ per spike in 45nm." Custom Integrated Circuits Conference (CICC), 2011 IEEE. IEEE, 2011.

[11] Otero, Carlos Tadeo Ortga, et al. “ULSNAP: An ultra-low power event-driven microcontroller for sensor network nodes.” Quality Electronic Design (ISQED), 2014 15th International Symposium on. IEEE, 2014.

[12] Teifel, John, and Rajit Manohar. “Highly pipelined asynchronous FPGAs.” Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays. ACM, 2004.

[13] Kees van Berkel, et al. “The VLSI-programming language Tangram and its translation into handshake circuits”. Proceedings of the conference on European design automation. IEEE Computer Society Press, 1991.

[14] Steven Burns and Alain J. Martin. “Syntax-directed translation of concurrent programs into self-timed circuits”. Computer Science Department at California Institute of Technology: Caltech-CS-TR-88-14, 1988.

[15] Sir Charles Antony Richard Hoare. “Communicating Sequential Processes”. Communications of the ACM, pages 666-677, 1978.

[16] Christopher LaFrieda, and Rajit Manohar. “Reducing power consumption with relaxed quasi delay-insensitive circuits.” Asynchronous Circuits and Systems, 2009. ASYNC'09. 15th IEEE Symposium on. IEEE, 2009.

[17] Andrew Matthew Lines. “Pipelined asynchronous circuits”. California Institute of Technology Pasadena, CA, USA, 1998.

[18] Rajit Manohar, Tak-Kwan Lee, and Alain J. Martin. “Projection: A Synthesis Technique for Concurrent Systems”. Proceedings of the 5th IEEE International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC), pp. 125--134, April 1999.

[19] Manohar, Rajit, and Alain J. Martin. “Slack elasticity in concurrent computing.” International Conference on Mathematics of Program Construction. Springer, Berlin, Heidelberg, 1998.

[20] Alain J. Martin. “Synthesis of Asynchronous VLSI Circuits”. Computer Science Department at California Institute of Technology: Caltech-CS-TR-93-28, 1991.

[21] Martin, Alain J. “Compiling communicating processes into delay-insensitive VLSI circuits.” Distributed computing 1.4 (1986): 226-234.

[22] Janusz A. Brzozowski, and Carl-Johan H. Seger. “Asynchronous circuits.” Springer Science and Business Media, 2012.

[23] Catherine Wong and Alain Martin. “High-level synthesis of asynchronous systems by data-driven decomposition”. Proceedings of the 40th annual Design Automation Conference. Proceedings of the 40th annual Design Automation Conference (DAC), pp. 508--513, June 2003.

[24] Kees Van Berkel. “VLSI programming of a modulo-N counter with constant response time and constant power”. Proceedings of the Working Conference Asynchronous Design Methodologies, Manchester, U.K., 1993, pp. 1-11.

[25] Kees Van Berkel. “Handshake Circuits: an Asynchronous Architecture for VLSI programming”. Vol. 5. Cambridge University Press, 1993.

[26] Ebergen, J.C. and A. Megacz. “Asynchronous loadable down counter”. US Patent 8,027,425, September 27, 2011.

[27] David H. Eby. “Programmable ripple counter having exclusive OR gates”. US Patent 4,612,658, September 16, 1986.

[28] Kim H. Eckert. “Ripple counter with reverse-propagated zero detection”. US Patent 5,060,243, October 22, 1991.

[29] Kaur, Upwinder, and Rajesh Mehra. “Low Power CMOS Counter Using Clock Gated Flip-Flop.”International Journal of Engineering and Advanced Technology 2.4 (2013): 796-798.

[30] Larsson, Patrik. “High-speed architecture for a programmable frequency divider and a dual-modulus prescaler”. IEEE Journal of Solid-State Circuits 31.5 (1996): 744-748.

[31] Mircea R. Stan, Alexandre Tenca, and Milos Ercegovac. “Long and fast up/down counters”. IEEE Transactions on computers 47.7 (1998): 722-735.

[32] Jon Tse and Derek Lockhart. “An Asynchronous Constant-Time Counter for Empty Pipeline Detection”. jontse.com, 2009.

Ned Bingham is a PhD student at Yale. He received his B.S. (2013) and M.S. (2017) from Cornell. During his Masters, he designed a set of tools for working with self-timed systems using a control-flow specification called Handshaking Expansions. Currently, he is researching self-timed systems as a method of leveraging average workload characteristics in general compute architectures. Between his studies, he has worked at Intel on Pre-Silicon Validation (2011, 2012), Qualcomm researching arithmetic architecture (2014), and Google researching self-timed systems (2016). In his spare time, he reads about governmental systems and dabbles in building collaborative tools. (www.nedbingham.com)

Rajit Manohar is the John C. Malone Professor of Electrical Engineering and Professor of Computer Science at Yale. He received his B.S. (1994), M.S. (1995), and Ph.D. (1998) from Caltech. He has been on the Yale faculty since 2017, where his group conducts research on the design, analysis, and implementation of self-timed systems. He is the recipient of an NSF CAREER award, nine best paper awards, nine teaching awards, and was named to MIT technology review's top 35 young innovators under 35 for contributions to low power microprocessor design. His work includes the design and implementation of a number of self-timed VLSI chips including the first high-performance asynchronous microprocessor, the first microprocessor for sensor networks, the first asynchronous dataflow FPGA, the first radiation hardened SRAM-based FPGA, and the first deterministic large-scale neuromorphic architecture. Prior to Yale, he was Professor of Electrical and Computer Engineering and a Stephen H. Weiss Presidential Fellow at Cornell. He has served as the Associate Dean for Research and Graduate studies at Cornell Engineering, the Associate Dean for Academic Affairs at Cornell Tech, and the Associate Dean for Research at Cornell Tech. He founded Achronix Semiconductor to commercialize high-performance asynchronous FPGAs. (csl.yale.edu/~rajit)