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][2]; 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, Templatized 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

© 2018 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.
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 it supports. So idzn would be an increment/decrement counter with zero/not zero detection. Further, an underscore separates channel boundaries. So idz_n would be a single channel with a 1of2 encoded request: $i$ and $d$ and a 1of2 encoded enable: $z$ and $n$ while idz_n would have two channels each with a 1of2 request and dataless enable.

### 1. 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.

```
v0:=0, vn:=0;
*|Lz!(v0\=0 \& vn\=0); Lc?lc;
    [ lc=inc \rightarrow vn:=vn+1
      ⊢ lc=dec \rightarrow vn:=vn-1
      ⊢ lc=clr \rightarrow 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.

```plaintext
v0:=0, vn:=0;
v0:=v0 & vn:=v0;
*|Lz!(v0\=0 \& vn\=0); Lc?lc;
    [ lc=inc \rightarrow [ v0:=0 & v0:=v0+1
      ⊢ v0:=1 & v0:=0 & vn:=vn+1
    ]
      ⊢ lc=dec \rightarrow [ v0:=0 & v0:=v0-1
      ⊢ v0:=1 & v0:=v0 & vn:=vn-1
    ]
      ⊢ lc=clr \rightarrow v0:=0; [ vn\# \& 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.

```plaintext
v0:=0, vn:=0; (Rz!vn=0 \& Rz?rz);
*|Lz!(v0\=0 \& vn\=0 \& Rz?rz);
    [ lc=inc \rightarrow [ v0:=0 & Rc!inc & Rz?rz
      ⊢ Rc?rc; vn:=vn+1; Rz!vn:=0
    ]
      ⊢ lc=dec \rightarrow [ v0:=0 & Rz!dec & Rz?rz
      ⊢ Rc?rc; vn:=vn-1; Rz!vn:=0
    ]
      ⊢ lc=clr \rightarrow v0:=0; [ ¬rz \& Rc!clr & Rz?rz
      ⊢ Rc?rc; vn:=vn; Rz!true
      ⊢ rz \& skip ]
    ]
]
```

Now, we can project the process into one that implements only the least significant bit of the counter. Thus, we have variables v0, Lc, Rz and one that implements the remaining bits with variables vn, Rz.

```plaintext
v0:=0; Rz?rz;
*|Lz!(v0\=0 \& Rz?rz); Lc?lc;
    [ lc=inc \rightarrow [ v0:=0 \& v0:=v0+1
      ⊢ v0:=1 \& v0:=0 & Rc!inc & Rz?rz
    ]
      ⊢ lc=dec \rightarrow [ v0:=0 \& v0:=v0-1
      ⊢ v0:=1 \& v0:=v0 & Rz!dec & Rz?rz
    ]
      ⊢ lc=clr \rightarrow v0:=0; [ ¬rz \& Rc!clr & Rz?rz
      ⊢ Rz?rc; vn:=vn; Rz!true
      ⊢ rz \& skip ]
    ]
]
```

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 $R_z$ 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.

$$v:=0;$$

*[ $Lz!(v=0 \land \overline{RZ})$ ]:

$$lcz?lc;$$

- $lc=inc A v=1 \rightarrow v:=0; Rz?; Rc!inc$
- $lc=inc A v=0 \rightarrow v:=1$
- $lc=dec A v=0 \rightarrow v:=1; Rz?; Rc!dec$
- $lc=dec A v=1 \rightarrow v:=0$
- $lc=clr A !\overline{RZ} \rightarrow v:=0; Rz?; Rc!clr$
- $lc=clr A \overline{RZ} \rightarrow v:=0$

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.

![Diagram of counter decomposed into processes.](image)

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 demand on $Lc$ and always returns true on $Lz$: *[ $Lc?; Lz!true$ ]. This adds an overflow condition to the previous counter specification.

$$vn:=0;$$

*[ $Lz!(vn=0)$ ]:

$$lc?lc;$$

- $lc=inc \rightarrow vn:=vn+1$
- $lc=dec \rightarrow vn:=vn-1$
- $lc=clr \rightarrow vn:=0$

$$[$$

- $vn > pow(2, \text{bits}) \rightarrow$
  - $vn:=vn-pow(2, \text{bits})$
- $vn < 0 \rightarrow vn:=vn+pow(2, \text{bits})$
- $else \rightarrow \text{skip}$

$$]$$

At the moment, if the value of the counter is $pow(2, \text{bits})-1$ (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.

$$v:=z;$$

*[ $Lz!(v=z)$ ]:

$$lc!lc;$$

- $lc=inc A v=1 \rightarrow v:=0; Rz?; Rc!inc$
- $lc=inc A v\neq 1 \rightarrow v:=1$
- $lc=dec A v\neq 1 \rightarrow v:=1; Rz?; Rc!dec$
- $lc=dec A v=1 A \overline{RZ} \rightarrow v:=z$
- $lc=dec A v=1 A !\overline{RZ} \rightarrow v:=0$
- $lc=clr A !\overline{RZ} \rightarrow v:=z; Rz?; Rc!clr$
- $lc=clr A \overline{RZ} \rightarrow v:=z$

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

$$[$$

- $vn > pow(2, \text{bits})+pow(2, \text{bits}-1) \rightarrow$
  - $vn:=vn-pow(2, \text{bits})$
- $vn < 0 \rightarrow vn:=vn+pow(2, \text{bits})$
- $else \rightarrow \text{skip}$

$$]$$

### 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 $Rc$ 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, $Rz$.

$$v1 A (Rz V Rn) A L_d \rightarrow Rz1$$

$$(v8 V vz) A L_1 \rightarrow R1r$$

$$(v8 V vz) A (Rz V Rn) A L_d \rightarrow Rz1$$

$v1 A Rn A L_d \rightarrow R8r$

$Rn A L_c \rightarrow R2r$

$Rc A (v1 A L_d V L_c) \rightarrow Rz1$

To understand what these production rules look like, we've rendered the production rules for $Rz1$ from above and $Rz1$ from below as a CMOS gate structure in black with combinational feedback in grey.

![Diagram of gate controlling $Rz1$ as described by the production rules.](image)

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 $x\overline{0}$, $x\overline{1}$, and $xZ$. This makes the rules for the internal register smaller and frees the reset phase of the forward drivers from various problematic acknowledgment constraints.
The checks for $\neg v\emptyset$, $v1$, 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].

$$v\emptyset \lor v1 \lor x \rightarrow L_z i$$

$$vz \lor x \rightarrow L_1 i$$

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.

$$\neg v1 \land \neg v\emptyset \lor \neg zd \land \neg L_1 \land vz \lor \neg vz \land \neg v\emptyset \lor \neg x z \land \neg L_1 \land \neg L_0 \land vz$$

$$(\neg zd \land vz \land L_0 \land L_1) \land (v\emptyset \lor v1 \lor vz) \rightarrow v1$$

$$(\neg zd \land vz \land L_0 \land L_1) \land (v\emptyset \lor v1 \lor vz) \rightarrow vz$$

$$(\neg zd \land vz \land L_0 \land L_1) \land (v\emptyset \lor v1 \lor vz) \rightarrow vz$$

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

$$\neg R_z \land \neg R_0 \land \neg v1 \rightarrow R_1 i$$

$$\neg vz \land \neg v\emptyset \rightarrow R_1 i$$

$$\neg R_z \land \neg R_0 \land \neg vz \land \neg v\emptyset \rightarrow R_d i$$

$$\neg vz \land \neg v\emptyset \land \neg R_0 \land \neg L_1 \land R_1$$

$$\neg vz \land \neg v1 \land R_z \land \neg L_1 \land R_z$$

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, $R_z i$ and $R_1 i$ 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 $L_1$ 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.

$$\neg R_z \land \neg R_0 \land \neg v1 \land R_1 i$$

$$\neg vz \land \neg v\emptyset \land R_1 i$$

$$\neg R_z \land \neg R_0 \land \neg vz \land \neg v\emptyset \land R_d i$$

$$\neg vz \land \neg R_0 \land \neg L_1 \land R_1$$

$$\neg vz \land \neg v1 \land \neg L_0 \land R_z$$

$$\neg R_0 \land \neg R\emptyset \land xz$$

$$\neg R_d \land \neg R\emptyset \land \neg x 1$$

$$\neg R_c \land \neg R\emptyset \land \neg Rz \land \neg x z$$

$$\neg x z \land \neg x \land \neg x$$

The checks for $v\emptyset$, $v1$, 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].

$$v\emptyset \lor v1 \lor x \rightarrow L_z i$$

$$vz \lor x \rightarrow L_1 i$$

The checks for $v\emptyset$, $v1$, 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].

$$v\emptyset \lor v1 \lor x \rightarrow L_z i$$

$$vz \lor x \rightarrow L_1 i$$

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.

$$\neg v1 \land \neg v\emptyset \lor \neg zd \land \neg L_1 \land vz \lor \neg vz \land \neg v\emptyset \lor \neg x z \land \neg L_1 \land \neg L_0 \land vz$$

$$(\neg zd \land vz \land L_0 \land L_1) \land (v\emptyset \lor v1 \lor vz) \rightarrow v1$$

$$(\neg zd \land vz \land L_0 \land L_1) \land (v\emptyset \lor v1 \lor vz) \rightarrow vz$$

$$(\neg zd \land vz \land L_0 \land L_1) \land (v\emptyset \lor v1 \lor vz) \rightarrow vz$$

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

$$\neg R_z \land \neg R\emptyset \land \neg x\emptyset$$

$$\neg R_d \land \neg R\emptyset \land \neg x 1$$

$$\neg R_c \land \neg R\emptyset \land \neg Rz \land \neg x z$$

$$\neg x z \land \neg x \land \neg x$$

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 \rightarrow count:=count+1
  \land lc=dec \rightarrow count:=count-1
  \land lc=rd \rightarrow !count ]
];
[ count \geq \text{pow}(2, \text{bits}) \land \text{pow}(2, \text{bits}) \rightarrow
  \land count<\theta \rightarrow count:=count+\text{pow}(2, \text{bits})
  \land else \rightarrow \text{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 \rightarrow v:=1 \land Rz?; Rc!inc
  \land lc=dec \rightarrow v:=z \land Rz?; Rc!dec
  \land lc=rd \rightarrow 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 v R_n) A L_r \rightarrow R_r t
```

```
v\emptyset A O_e A L_r \rightarrow O_r t
v1 A O_e A L_r \rightarrow O_r t
vz A O_e A L_r \rightarrow O_r t
```

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

```
R_c A (O_r v O_t v O_e) \rightarrow y t
```

```
v\emptyset v v1 v x v y \rightarrow L_z i
vz v x v y \rightarrow L_n i
```
The rules for the internal register remain unchanged, and the forward drivers for the read are reset normally.

\[ \neg R_z \land \neg R_n \land \neg L_r \rightarrow R_i \]

\[ \neg O_i \land \neg L_r \rightarrow 0_i \]

\[ \neg O_i \land \neg L_r \rightarrow 0_i \]

\[ \neg O_i \land \neg L_r \rightarrow 0_i \]

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.

\[ \neg R_r \land \neg O_i \land \neg O_i \rightarrow y_i \]

\[ \neg v_0 \land \neg v_1 \land \neg y \land \neg x \rightarrow L_z \]

\[ \neg v_z \land \neg y \land \neg x \rightarrow L_z \]

2. Bundled Data Read

The second method latchs 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.

\[ (R_z \lor R_n) \land L_r \rightarrow R_i \]

\[ R_r \rightarrow xx \]

\[ \neg x_0 \lor \neg x_1 \lor \neg xz \lor \neg xx \rightarrow x \]

\[ \neg R_z \land \neg R_n \land \neg L_r \rightarrow R_i \]

\[ \neg R_r \rightarrow xx \]

\[ xz \land \neg x_0 \land \neg x_1 \land \neg xx \rightarrow x \]

Then, if you don't care about the third value of the internal register, \( v_z \), we'll need rules to merge it in with \( v_0 \).

\[ D_z = v_1 \]

\[ v_0 \lor v_z \rightarrow D_i \]

\[ \neg v_0 \land \neg v_z \rightarrow D_i \]

Finally, the data, \( D_i \), is latched using the read request.

\[ O_i \lor D_i \land R_r \rightarrow O_i \]

\[ O_i \lor D_i \land R_r \rightarrow O_i \]

\[ \neg O_i \land (\neg D_i \lor \neg R_r) \rightarrow O_i \]

\[ \neg O_i \land (\neg D_i \lor \neg R_r) \rightarrow O_i \]

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.

\[ G_r = R_r \]

\[ G_e \land (R_z \lor R_n) \land L_r \rightarrow R_i \]

\[ \neg G_e \land \neg R_r \land \neg R_n \land \neg L_r \rightarrow R_i \]

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.

\[ L_r \land G_r \rightarrow R_i \]

\[ \neg L_r \land \neg G_r \rightarrow R_i \]

\[ R_e \rightarrow R_i \]

\[ \neg R_e \rightarrow R_i \]

\[ R_a \land R_r \rightarrow G_i \]

\[ R_a \lor L_1 \lor L_o \rightarrow L_z \]

\[ \neg R_a \land \neg L_1 \land \neg L_o \rightarrow L_z \]

\[ 1 \rightarrow L_n \]

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.

\[ \text{count} := \emptyset; \]

\[ \ast \{Lz!(\text{count} = \emptyset)\}; \]

\[ \text{Lc?Lc}; \]

\[ \{ \text{lc=dec} \rightarrow \text{count} := \text{count} - 1 \]

\[ 0 \text{lc=wr} \rightarrow \text{W?count} \}; \]

\[ 0 \text{count} \land \text{count} \rightarrow \text{skip} \]

\[ \} \]

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 \( \emptyset \), \( 1 \), or \( Z \) directly to the internal register.

\[ v := z; \]

\[ \ast \{Lz!(v=z)\}; \]

\[ \text{Lc?Lc}; \]

\[ \{ \text{lc=dec} \land v \neq 1 \rightarrow v := 1; \text{Rz}?; \text{Rc}! \]

\[ 0 \text{lc=dec} \land v = 1 \land \neg \text{Rz} \rightarrow v := 0 \]

\[ 0 \text{lc=dec} \land v = 1 \land \text{Rz} \rightarrow v := z \]

\[ 0 \text{lc=wr} \rightarrow \text{Rz}?; \text{Rc!wr}; \text{W?v} \]

\[ \} \]

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 \( Rw\theta, Rw1, \) and \( Rwz \).

\[
(R_z \lor R_n) \Rightarrow L_w \Rightarrow R_u \\
W_{\theta} \Rightarrow R_w \Rightarrow Rw\theta \\
W_{i} \Rightarrow R_u \Rightarrow Rw1 \\
W_{e} \Rightarrow R_u \Rightarrow Rwz
\]

Now, \( Rw\theta, Rw1, \) 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.

\[
Rw\theta \lor Rw1 \lor Rwz \Rightarrow W_{ei} \\
Rw\theta \lor R_{\theta} \Rightarrow xc_i \\
Rw1 \lor R_{o} \Rightarrow xc_i \\
Rwz \lor R_{z} \Rightarrow xc_i
\]

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

\[
\neg v_1 \land \neg v_{\theta} \land \neg xz \land \neg L_w \land \neg L_\theta \land \neg v_\theta \\
\neg v_1 \land \neg vz \land \neg x_{\theta} \land \neg L_w \land \neg L_\theta \land v_{\theta} \\
\neg vz \land \neg v_{\theta} \land \neg x_{\theta} \land \neg L_w \land \neg L_\theta \land v_\theta \\
(L_w \land L_\theta \land v_{xz}) \land (v_{\theta} \land v_1) \land vz \\
(L_w \land L_\theta \land v_{xz}) \land (vz \land v_1) \land v_{\theta} \\
(L_w \land L_\theta \land v_{xz}) \land (vz \land v_\theta) \land vz
\]

The output signals are then reset normally using the \( Rw\theta, Rw1, \) and \( Rwz \) signals to check the correct value of the internal register.

\[
\neg R_z \land \neg R_n \land \neg L_w \Rightarrow R_u \\
\neg W_{i} \land \neg vz \land \neg v_1 \land \neg R_w \Rightarrow Rw\theta \\
\neg W_{i} \land \neg v_{\theta} \land \neg vz \land \neg R_w \Rightarrow Rw1 \\
\neg W_{i} \land \neg v_{\theta} \land \neg v_1 \land \neg R_w \Rightarrow Rwz
\]

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

\[
\neg Rw\theta \land \neg Rw1 \land \neg Rwz \Rightarrow W_{ei} \\
\neg Rw\theta \land \neg R_{\theta} \Rightarrow xc_i \\
\neg Rw1 \land \neg R_{o} \Rightarrow xc_i \\
\neg Rwz \land \neg R_{z} \Rightarrow xc_i
\]

**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.

**B. Implementation**

With this implementation, we can take advantage of early out to get logarithmic average case complexity instead of linear. If \( W_1 \) is true or \( Z_1 \) 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.

\[
W_{ei} = L_{e} \\
Z_{i} = L_{e} \\
Z_{o} = Z_{i} \land W_{i} \Rightarrow Z_{o} \\
Z_{o} = Z_{i} \land W_{i} \Rightarrow Z_{o} \\
Z_{i} \land W_{i} \land W_{o} \Rightarrow W_{o} \\
Z_{i} \land W_{i} \land W_{o} \Rightarrow W_{o} \\
Z_{o} \land V_{z} \Rightarrow V_{z} \\
W_{o} \land W_{o} \land V_{z} \Rightarrow W_{o} \\
\neg V_{z} \land \neg W_{o} \Rightarrow L_{e}
\]

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 an 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.

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_t are the same, we make them the same node with no consequences. So this turns into a fairly simple buffer.

\[
C_d = Z_t
\]

\[
Z_{eq} \land L_f \land C_n \rightarrow C_d t
\]

\[
Z_{eq} \land L_f \land C_g \rightarrow Z_t t
\]

\[
L_t \land (C_g \lor C_{eq}) \rightarrow C_f t
\]

\[
Z_i \lor C_i \rightarrow L_{eq}
\]

\[
\neg Z_{eq} \land \neg C_n \rightarrow C_{eq} t
\]

\[
\neg Z_{eq} \land \neg L_f \rightarrow Z_{eq} t
\]

\[
\neg L_t \land \neg C_g \land \neg C_n \rightarrow C_{eq} t
\]

\[
\neg Z_{eq} \land \neg C_{eq} \rightarrow L_{eq} t
\]

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. Circuity necessary for reset was not included in any the above descriptions.

![Fig. 3. Measured Performance and Energy for an array of counters.](image)

<table>
<thead>
<tr>
<th>Type</th>
<th>Trans</th>
<th>Frequency</th>
<th>Energy/Op</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>d_z</td>
<td>50N</td>
<td>2.73 GHz</td>
<td>24.01 fJ</td>
<td>N/A</td>
</tr>
<tr>
<td>dzn</td>
<td>102N+10</td>
<td>2.15 GHz</td>
<td>48.51 fJ</td>
<td>399 ps</td>
</tr>
<tr>
<td>idzn</td>
<td>146N+12</td>
<td>2.03 GHz</td>
<td>56.05 fJ</td>
<td>421 ps</td>
</tr>
<tr>
<td>idczn</td>
<td>174N+14</td>
<td>2.00 GHz</td>
<td>40.62 fJ</td>
<td>442 ps</td>
</tr>
<tr>
<td>idrzn</td>
<td>246N+14</td>
<td>1.88 GHz</td>
<td>89.51 fJ</td>
<td>441 ps</td>
</tr>
<tr>
<td>idrzn_bd</td>
<td>188N+32</td>
<td>1.77 GHz</td>
<td>75.20 fJ</td>
<td>441 ps</td>
</tr>
<tr>
<td>dwzn</td>
<td>192N+12</td>
<td>1.86 GHz</td>
<td>43.81 fJ</td>
<td>487 ps</td>
</tr>
<tr>
<td>is_zn</td>
<td>146N+61</td>
<td>2.08 GHz</td>
<td>45.52 fJ</td>
<td>139 ps</td>
</tr>
</tbody>
</table>

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.
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.

<table>
<thead>
<tr>
<th>Type</th>
<th>Trans</th>
<th>Frequency</th>
<th>Energy/Op</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>id_c_zn[24]</td>
<td>117N+32</td>
<td>1.42 GHz</td>
<td>73.34 fJ</td>
<td>468 ps</td>
</tr>
<tr>
<td>id_c_zn[32]</td>
<td>398N+26</td>
<td>0.60 GHz</td>
<td>152.76 fJ</td>
<td>1150 ps</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type</th>
<th>Trans</th>
<th>Frequency</th>
<th>Energy/Op</th>
</tr>
</thead>
<tbody>
<tr>
<td>id_c_zn</td>
<td>74N</td>
<td>1.00 GHz</td>
<td>169.18 fJ</td>
</tr>
<tr>
<td>id_c_zn</td>
<td>74N</td>
<td>2.00 GHz</td>
<td>116.75 fJ</td>
</tr>
<tr>
<td>id_c_zn</td>
<td>74N</td>
<td>3.00 GHz</td>
<td>98.24 fJ</td>
</tr>
<tr>
<td>id_c_zn</td>
<td>74N</td>
<td>4.00 GHz</td>
<td>86.12 fJ</td>
</tr>
</tbody>
</table>

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** \( \text{skip} \) does nothing and continues to the next command.
- **Dataless Assignment** \( \text{a} := \text{e} \) sets the voltage of the node \( \text{a} \) to \( \text{Vdd} \) and \( \text{c} := \text{t} \) sets it to \( \text{GND} \).
- **Assignment** \( \text{a} := \text{e} \) waits until the expression \( \text{e} \) has a valid value, then assigns that value to the variable, \( \text{a} \).
- **Send** \( \text{X}! \text{e} \) waits until the expression \( \text{e} \) has a valid value, then sends that value across the channel \( \text{X} \). \( \text{X}! \) is a dataless send.
- **Receive** \( \text{X}? \text{a} \) waits until there is a valid value on the channel \( \text{X} \), then assigns that value to the variable \( \text{a} \). \( \text{X}? \) is a dataless receive.
- **Probe** \( \text{X} \) returns the value to be received from the channel \( \text{X} \) without executing a receive.
- **Sequential Composition** \( \text{S} \rightarrow \text{T} \) executes the programs \( \text{S} \) and \( \text{T} \) in any order.
- **Parallel Composition** \( \text{S} \parallel \text{T} \) executes the programs \( \text{S} \) and \( \text{T} \) in any order.
- **Deterministic Selection** \( \text{[G} \rightarrow \text{S}_1 \ldots \text{G}_n \rightarrow \text{S}_n\] \) where \( \text{G}_i \) is a guard or \( \text{S}_i \) 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, \( \text{G}_i \), evaluates to \( \text{Vdd} \), then executes the corresponding program, \( \text{S}_i \). The guards must be mutually exclusive. The notation \( \text{[G]} \) is shorthand for \( \text{G} \rightarrow \text{skip} \).
- **Repetition** \( \ast \text{[G} \rightarrow \text{S}_1 \ldots \text{G}_n \rightarrow \text{S}_n\] \) is similar to the selection statements. However, the action is repeated until no guard evaluates to \( \text{Vdd} \). \( \ast \text{[true} \rightarrow \text{S}\] \) is shorthand for \( \ast \text{[true} \rightarrow \text{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 \( \text{a} = \text{b} \) aliases two names to one circuit node. A rule \( \text{G} \rightarrow \text{A} \) represents a guarded action where \( \text{G} \) is a guard (as described above) and \( \text{A} \) 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


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)