A 13Ghz Loadable Counter

with 20ps/bit Settling Time
and Early Completion, in 40nm CMOS

BWRC Seminar
02-Oct-2009

Adam Megacz
(joint work with Ivan Sutherland and Jo Ebergen)

Berkeley EECS

Outline

- Numbers
  - How to count
  - Problem statement
  - Number representations
    - Redundant representations
    - Resettling
- *Interlude*
- Circuits
  - Algorithm
  - Circuit
  - Layout
  - Implementation
  - Demo

How do we count?

How do we count?

- Counting down

How do we count?

- Counting down

How do we count?

- Counting down
  - >   17

How do we count?

- Counting down

```
>                17
>                16
```

How do we count?

- Counting down
  - 17
  - 16
  - 15

How do we count?

- Counting down

  > 17

  > 16

  > 15

  > 14

How do we count?

- Counting down
  - `17`
  - `16`
  - `15`
  - `14`
  - `13`

How do we count?

- Counting down

```
>        17
>        16
>        15
>        14
>        13
>        12
```

How do we count?

- Counting down
  - 17
  - 16
  - 15
  - 14
  - 13
  - 12
  - ... and so on

How do we count?

How do we count?

- Counting down in binary

How do we count?

- Counting down in binary

How do we count?

- Counting down in binary
  - `10001 = 17`

How do we count?

- Counting down in binary
  - 10001 = 17
  - 10000 = 16

How do we count?

- Counting down in binary
  - `10001 = 17`
  - `10000 = 16`
  - `01111 = 15`

How do we count?

- Counting down in binary
  - 10001 = 17
  - 10000 = 16
  - 01111 = 15
  - 01110 = 14

How do we count?

- Counting down in binary
  - 10001 = 17
  - 10000 = 16
  - 01111 = 15
  - 01110 = 14
  - 01101 = 13

How do we count?

- Counting down in binary
  - 10001 = 17
  - 10000 = 16
  - 01111 = 15
  - 01110 = 14
  - 01101 = 13
  - 01100 = 12

How do we count?

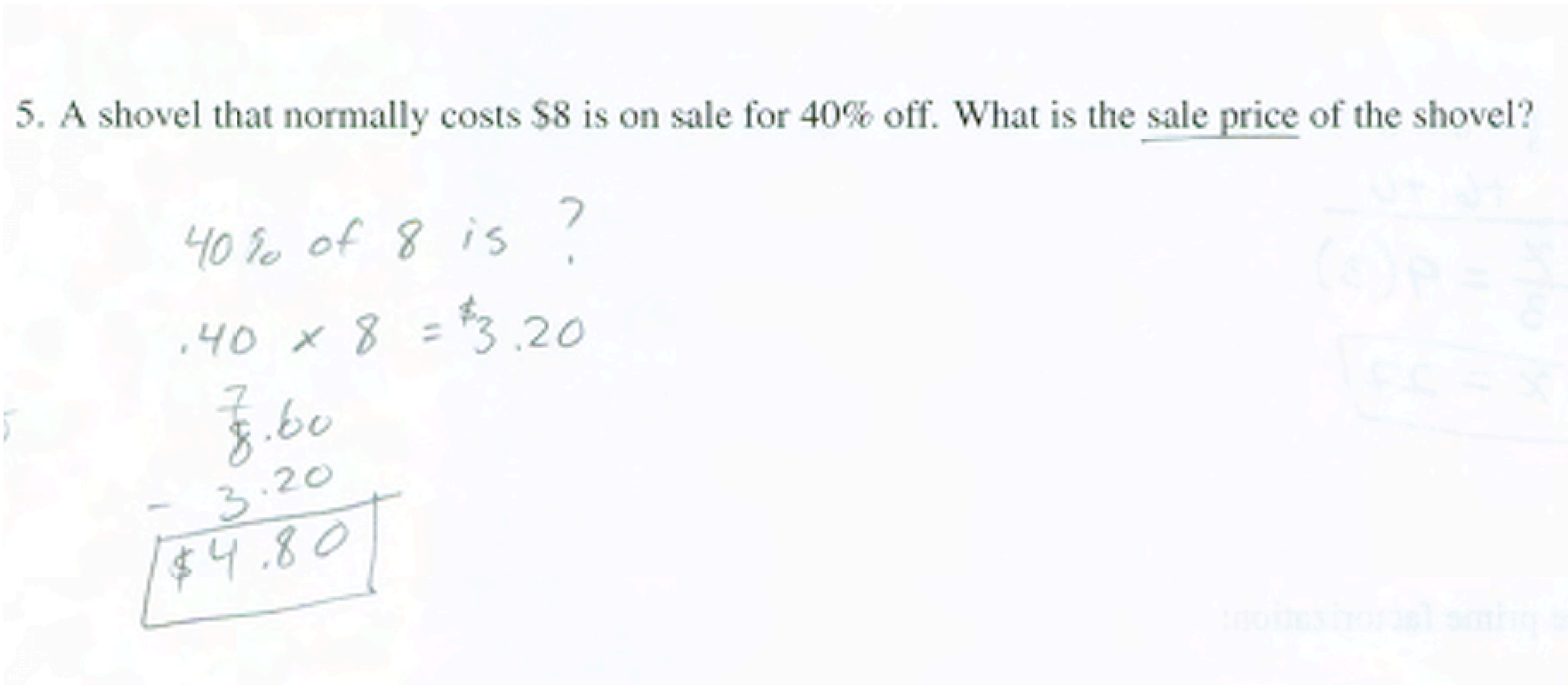
- Counting down in binary
  - 10001 = 17
  - 10000 = 16
  - 01111 = 15
  - 01110 = 14
  - 01101 = 13
  - 01100 = 12
  - ... and so on

How do we count?

- Counting down in binary
  - 10001 = 17
  - $\boxed{10000}$ = 16
  - 01111 = 15
  - 01110 = 14
  - 01101 = 13
  - 01100 = 12
  - ... and so on

Problem: "zeroness" may depend on the state of *all* bits of the count in the worst case. Not scalable.

How do we count?



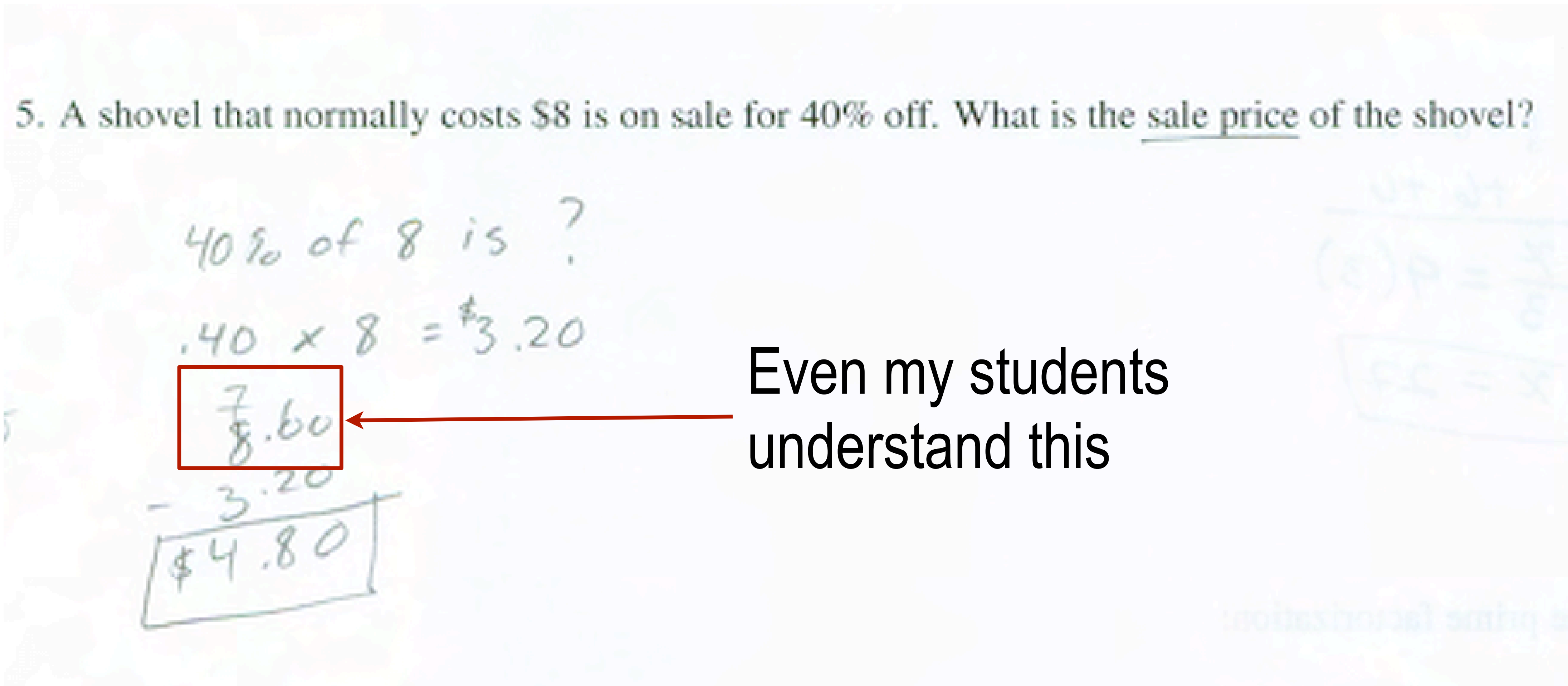
5. A shovel that normally costs $8 is on sale for 40% off. What is the sale price of the shovel?

40% of 8 is ?

$.40 \times 8 = \$3.20$

$$\begin{array}{r} \overset{7}{\cancel{8}}.{}^{1}00 \\ -\ 3.20 \\ \hline \boxed{\$4.80} \end{array}$$

How do we count?



Even my students understand this

What problem are we solving?

- Need an `n`-bit counter with two operations:
  - `load` will:
    - Accept an ordinary binary value (no fancy encodings allowed!)
    - Set the counter to that value
  - `dec` will either:
    - Report `failure` if the counter value was zero
    - Report `success` and decrement the counter if it was nonzero
- Performance requirements:
  - "`dec`" must complete in a bounded amount of time
    - No matter what the count value is.
    - No matter how big the counter (`n`) is.

How do we count?

- Counting down in binary
  - $10001 = 1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 17$
  - 10000 = 16
  - 01111 = 15
  - 01110 = 14
  - 01101 = 13
  - 01100 = 12
  - ... and so on

How do we count?

- Redundant binary representations
  - $>$ 10001 = $1{\times}2^4+0{\times}2^3+0{\times}2^2+0{\times}2^1+1{\times}2^0=17$

How do we count?

- Redundant binary representations
  - $10001 = 1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 17$
  - $02001 = 0 \times 2^4 + 2 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 17$

How do we count?

- Redundant binary representations
  - $>10001 = 1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 17$
  - $>02001 = 0 \times 2^4 + 2 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 17$
  - $>01201 = 0 \times 2^4 + 1 \times 2^3 + 2 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 17$

How do we count?

- Redundant binary representations
  - 10001 = $1\times2^4+0\times2^3+0\times2^2+0\times2^1+1\times2^0=17$
  - 02001 = $0\times2^4+2\times2^3+0\times2^2+0\times2^1+1\times2^0=17$
  - 01201 = $0\times2^4+1\times2^3+2\times2^2+0\times2^1+1\times2^0=17$
  - 01121 = $0\times2^4+1\times2^3+1\times2^2+2\times2^1+1\times2^0=17$

How do we count?

- Redundant binary representations
  - $10001 = 1\times2^4+0\times2^3+0\times2^2+0\times2^1+1\times2^0=17$
  - $02001 = 0\times2^4+2\times2^3+0\times2^2+0\times2^1+1\times2^0=17$
  - $01201 = 0\times2^4+1\times2^3+2\times2^2+0\times2^1+1\times2^0=17$
  - $\boxed{01121} = 0\times2^4+1\times2^3+1\times2^2+2\times2^1+1\times2^0=17$

Fully “settled” representation

How do we count?

- Decrementing a settled number

$$>01121 \;=\; 0\times2^4+1\times2^3+1\times2^2+2\times2^1+1\times2^0=17$$

$$>01120 \;=\; 0\times2^4+1\times2^3+1\times2^2+2\times2^1+0\times2^0=16$$

Theorem 1: if the representation is fully settled, only the least significant bit needs to be used to decrement and test zeroness.

How do we count?

- Resettling

```
>01120 = 16
>01112 = 16
```

How do we count?

- Resettling

```
>01120 = 16    >01110 = 14
>01112 = 16    >01102 = 14
```

How do we count?

- Resettling

```
>01120 = 16   >01110 = 14
>01112 = 16   >01102 = 14
```

Any time you see a 0 to the right of a non-0, you can resettle by decrementing the non-0 and turning the 0 into a 2

How do we count?

- Theorem 2 (Ebergen)
  - On average, no more than 2 settling operations are performed per decrement operation.

- 01121 = 17
- 01120 = 16
- 01112 = 16
- 01111 = 15
- 01110 = 14
- 01102 = 14
- 01022 = 14
- 00222 = 14
- 00221 = 13
- 00220 = 12
- 00212 = 12
- 00211 = 11

How do we count?

- Theorem 2 (Ebergen)
  - On average, no more than 2 settling operations are performed per decrement operation.

Toggle rate is “n”

- 01121 = 17
- 01120 = 16
- 01112 = 16
- 01111 = 15
- 01110 = 14
- 01102 = 14
- 01022 = 14
- 00222 = 14
- 00221 = 13
- 00220 = 12
- 00212 = 12
- 00211 = 11

How do we count?

- Theorem 2 (Ebergen)
  - On average, no more than 2 settling operations are performed per decrement operation.

Toggle rate is “n/2”

- 01121 = 17
- 01120 = 16
- 01112 = 16
- 01111 = 15
- 01110 = 14
- 01102 = 14
- 01022 = 14
- 00222 = 14
- 00221 = 13
- 00220 = 12
- 00212 = 12
- 00211 = 11

How do we count?

- Theorem 2 (Ebergen)
  - On average, no more than 2 settling operations are performed per decrement operation.

Toggle rate is “n/4”

- 01121 = 17
- 01120 = 16
- 01112 = 16
- 01111 = 15
- 01110 = 14
- 01102 = 14
- 01022 = 14
- 00222 = 14
- 00221 = 13
- 00220 = 12
- 00212 = 12
- 00211 = 11

How do we count?

- Theorem 2 (Ebergen)
  - On average, no more than 2 settling operations are performed per decrement operation.

Toggle rate is “n/8”

- 01121 = 17
- 01120 = 16
- 01112 = 16
- 01111 = 15
- 01110 = 14
- 01102 = 14
- 01022 = 14
- 00222 = 14
- 00221 = 13
- 00220 = 12
- 00212 = 12
- 00211 = 11

How do we count?

- Theorem 2 (Ebergen)
  - On average, no more than 2 settling operations are performed per decrement operation.

- 01121 = 17
- 01120 = 16
- 01112 = 16
- 01111 = 15
- 01110 = 14
- 01102 = 14
- 01022 = 14
- 00222 = 14
- 00221 = 13
- 00220 = 12
- 00212 = 12
- 00211 = 11

How do we count?

- Theorem 2 (Ebergen)
  - On average, no more than 2 settling operations are performed per decrement operation.

```
> 01121 = 17
> 01120 = 16
> 01112 = 16
> 01111 = 15
> 01110 = 14
> 01102 = 14
> 01022 = 14
> 00222 = 14
> 00221 = 13
> 00220 = 12
> 00212 = 12
> 00211 = 11
```

$$\sum_{i=0}^{\infty} \frac{n}{2^i} = n/1 + n/2 + n/4 + \ldots = 2n$$

How do we count?

- Resettling

```
>01112 = 16
>01111 = 15
>01110 = 14
>01102 = 14
>01021 = 13
```

Theorem 3: resettling the upper bits can be performed concurrently with decrementing the lower bits. If the number was already settled, it will resettle *as fast or faster than* it can be decremented.

Interlude

- Have we assumed that the counter is only finitely long?

Circuits

Syntax

- To avoid terrible confusion, I will...
  - Use words for *states*: `Zero`, `One`, `Two`, `Done`
  - Use numbers for *numerals* (duh): `0`, `1`, `2`
  - Use symbols for *logic levels*: `-`, `+`

Circuit Implementation

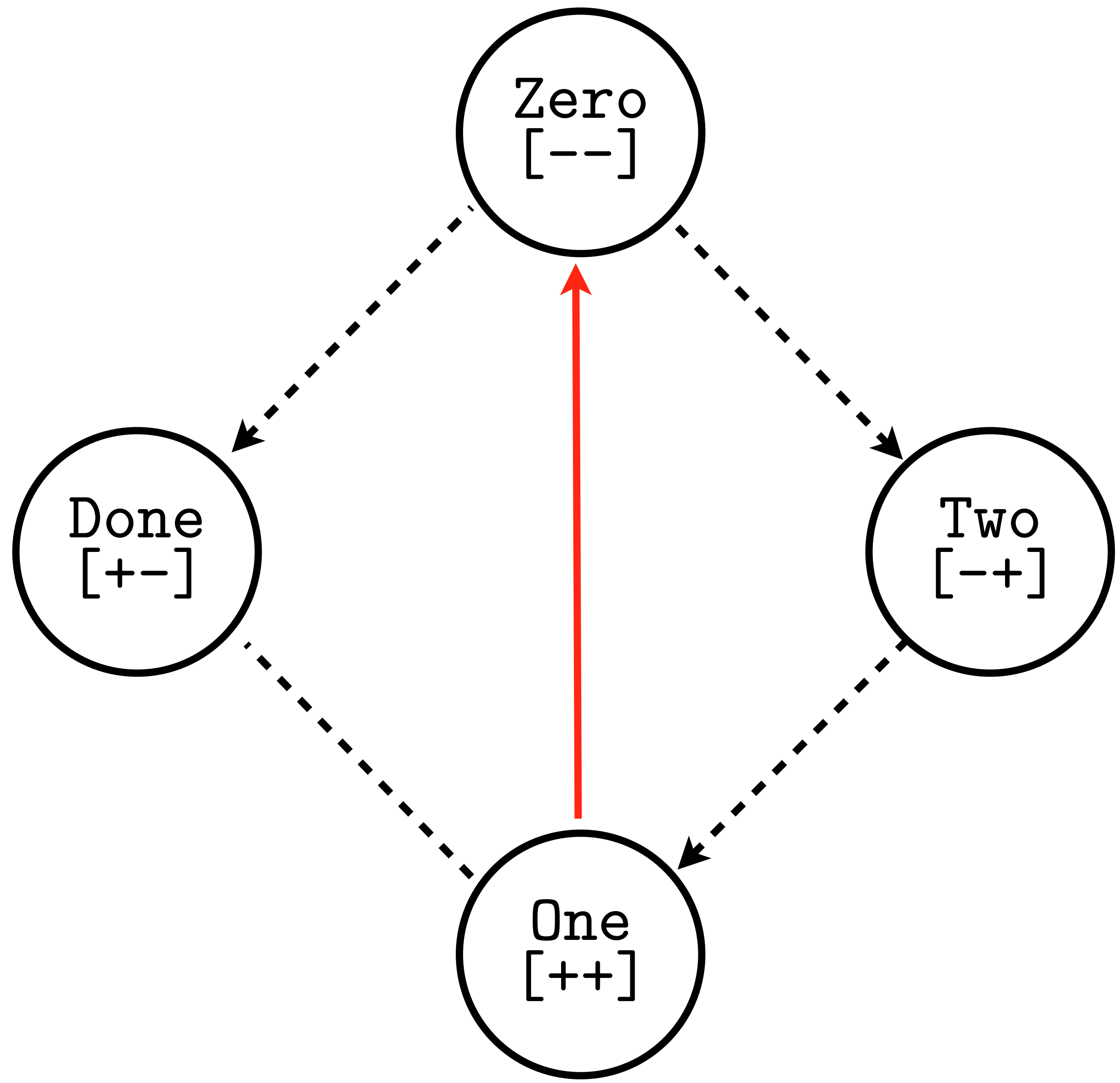
- Each "bit" of the counter is a pair of state wires
  - Four possible states: `Zero`, `One`, `Two`, `Done`
- A GasP module sits between each pair of bits
  - When the *more significant neighbor* is not `Zero`
  - .. and the *least significant neighbor* is `Zero`
  - then fire, and:
    - If the more significant neighbor is `Done` set the less significant neighbor to `Done`
    - Otherwise decrement the more significant neighbor and set the less significant neighbor to `Two`

Safest Binary Encoding

- I will name the two state wires
  - `OneOrDone` , which is + when the state is `One` or `Done`
  - `OneOrTwo` , which is + when the state is `One` or `Two`
- I will write the state of a pair of wires as
  - `[OneOrDone,OneOrTwo]`
  - So,
    - `Zero=[--]`
    - `One =[++]`
    - `Two =[-+]`
    - `Done=[+-]`

Hamming Distances

- Diagram
  - Hamming-adjacent codes connected by dashed lines
  - Arrowheads indicate possible transitions
- One transition (`One-to-Zero`) is not Hamming adjacent
  - Neighbor might "see" a `Done` or a `Two` during the transition.
  - Must manually check that this will not cause misbehavior.
    - Fortunately, it does not.

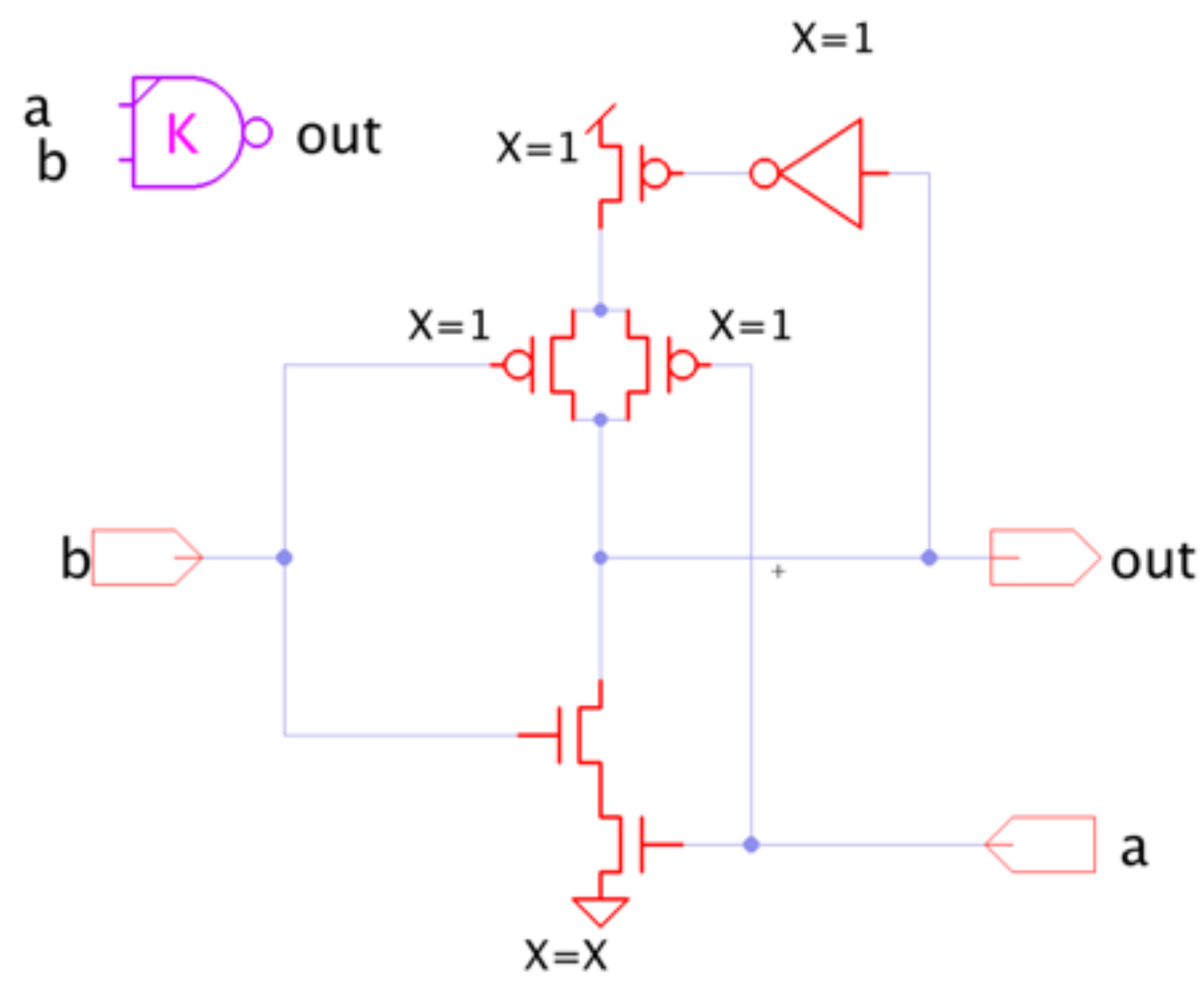


Adding a Timing Constraint

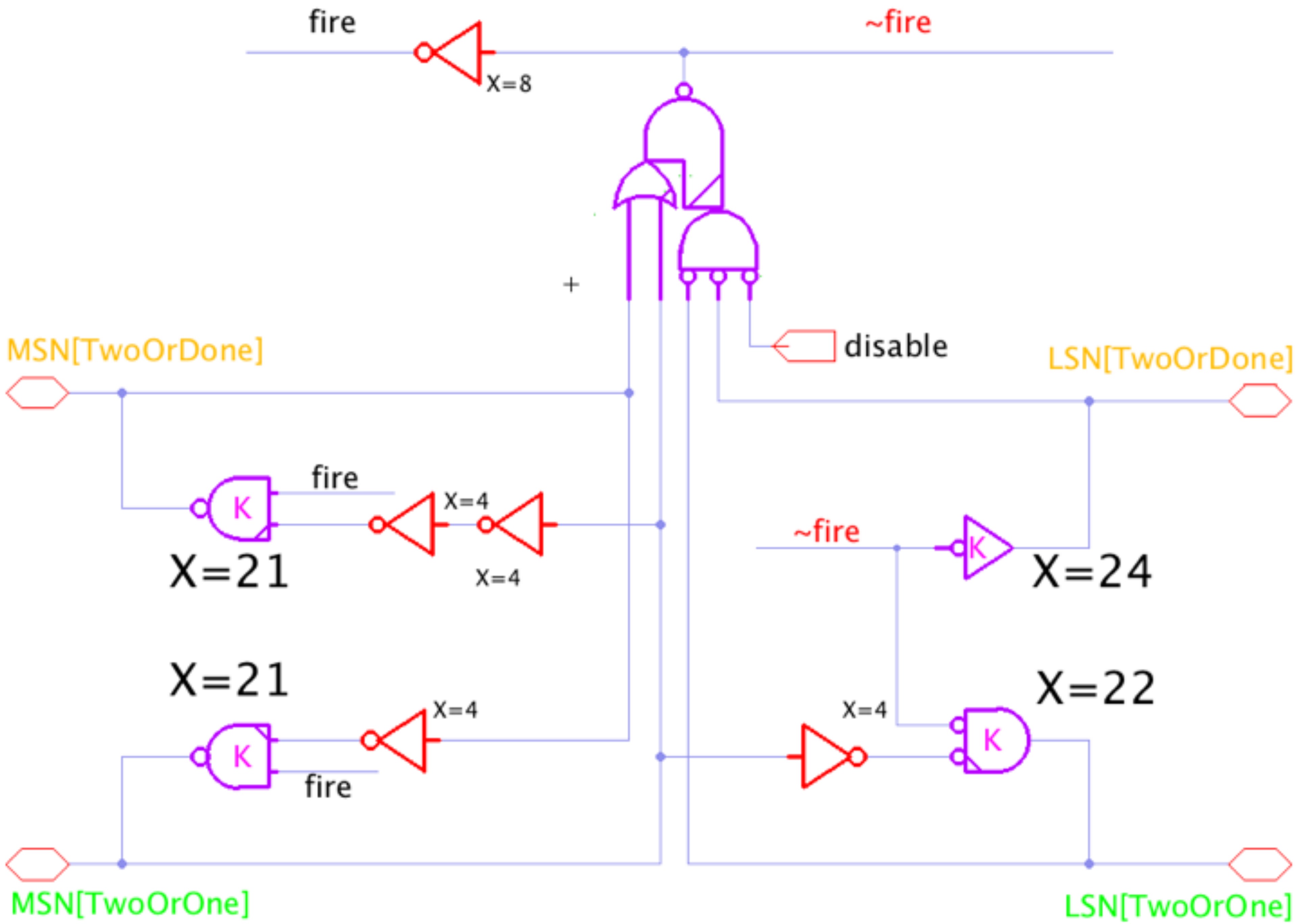
- If we are willing to assume a timing constraint, we can simplify the circuit
  - This turns out to speed it up considerably
  - New state wires:
    - `TwoOrDone`, which is + when the state is `Two` or `Done`
    - `TwoOrOne`, which is + when the state is `Two` or `One`

Half-Drivers

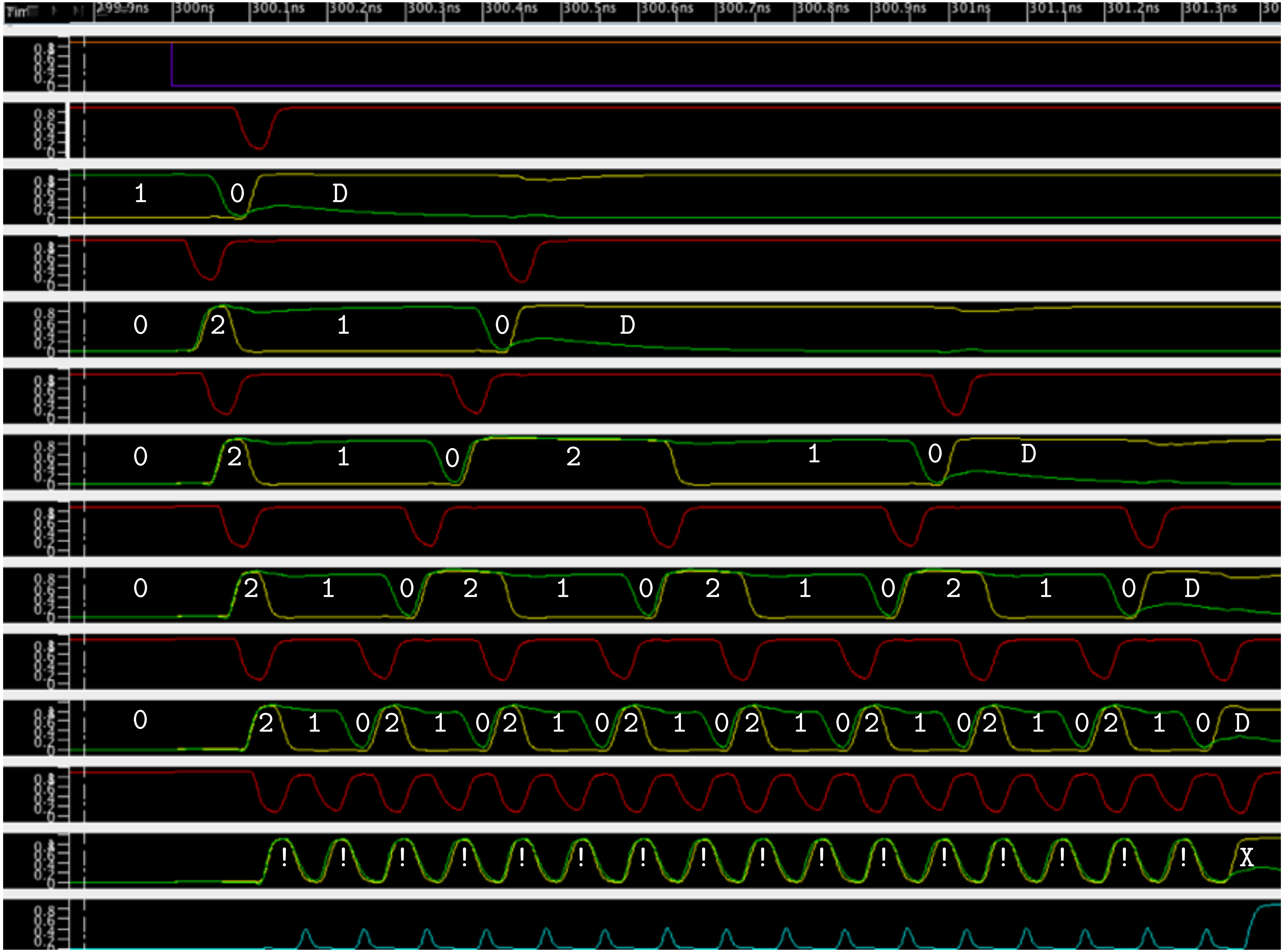
- Pull-up network:
  - > active only when output is high
  - > weak (X=1) transistors
  - > otherwise same behavior



One Bit (Full Circuit)

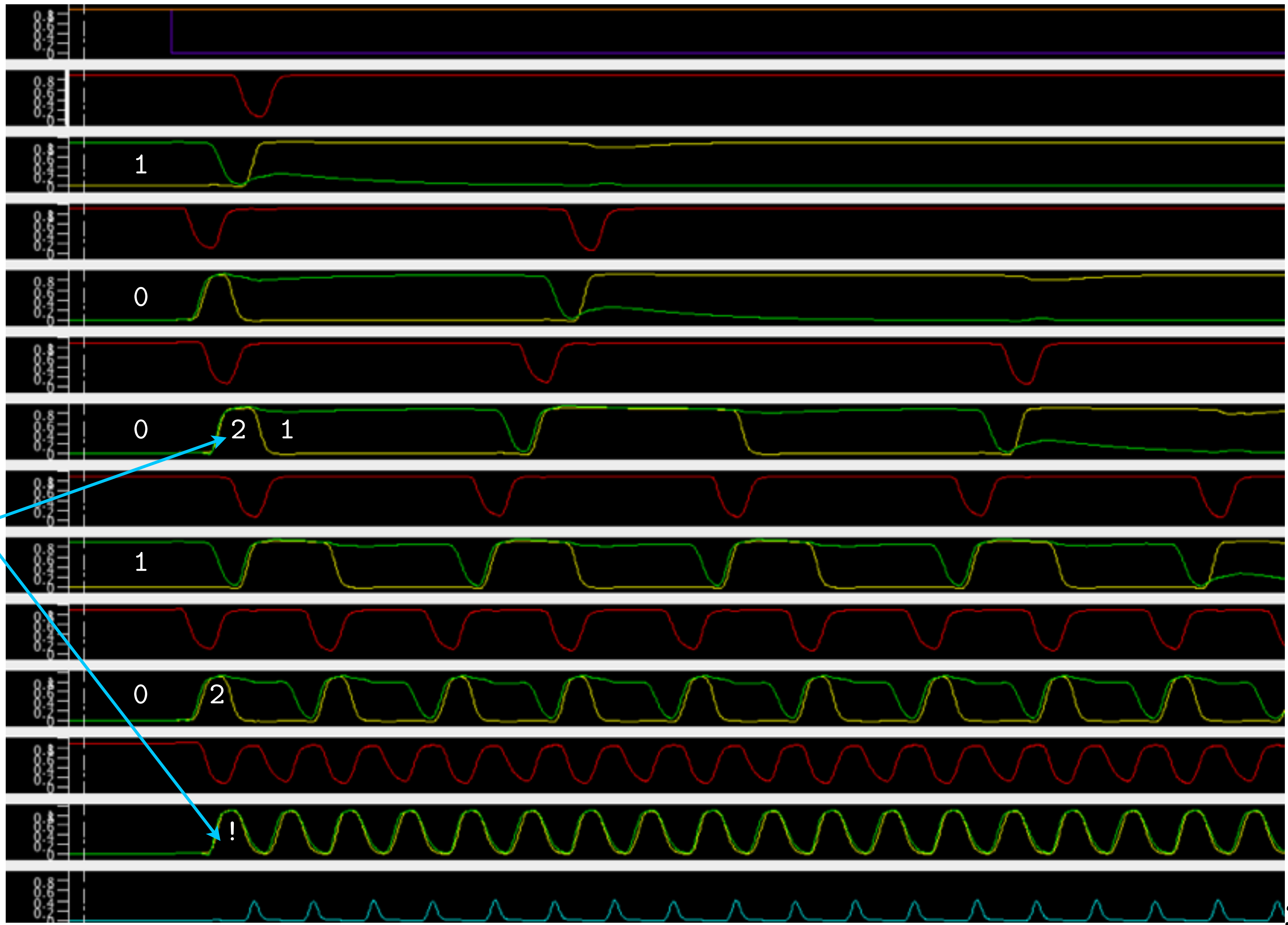


Loading 16 (binary 10000)



Loading 18 (binary 10010)

First decrement happens while borrowing is still in progress

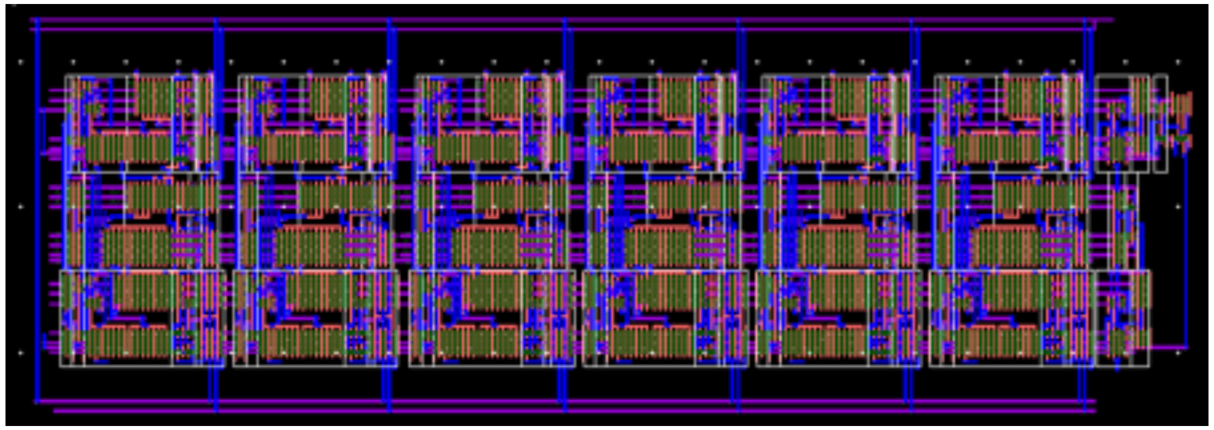


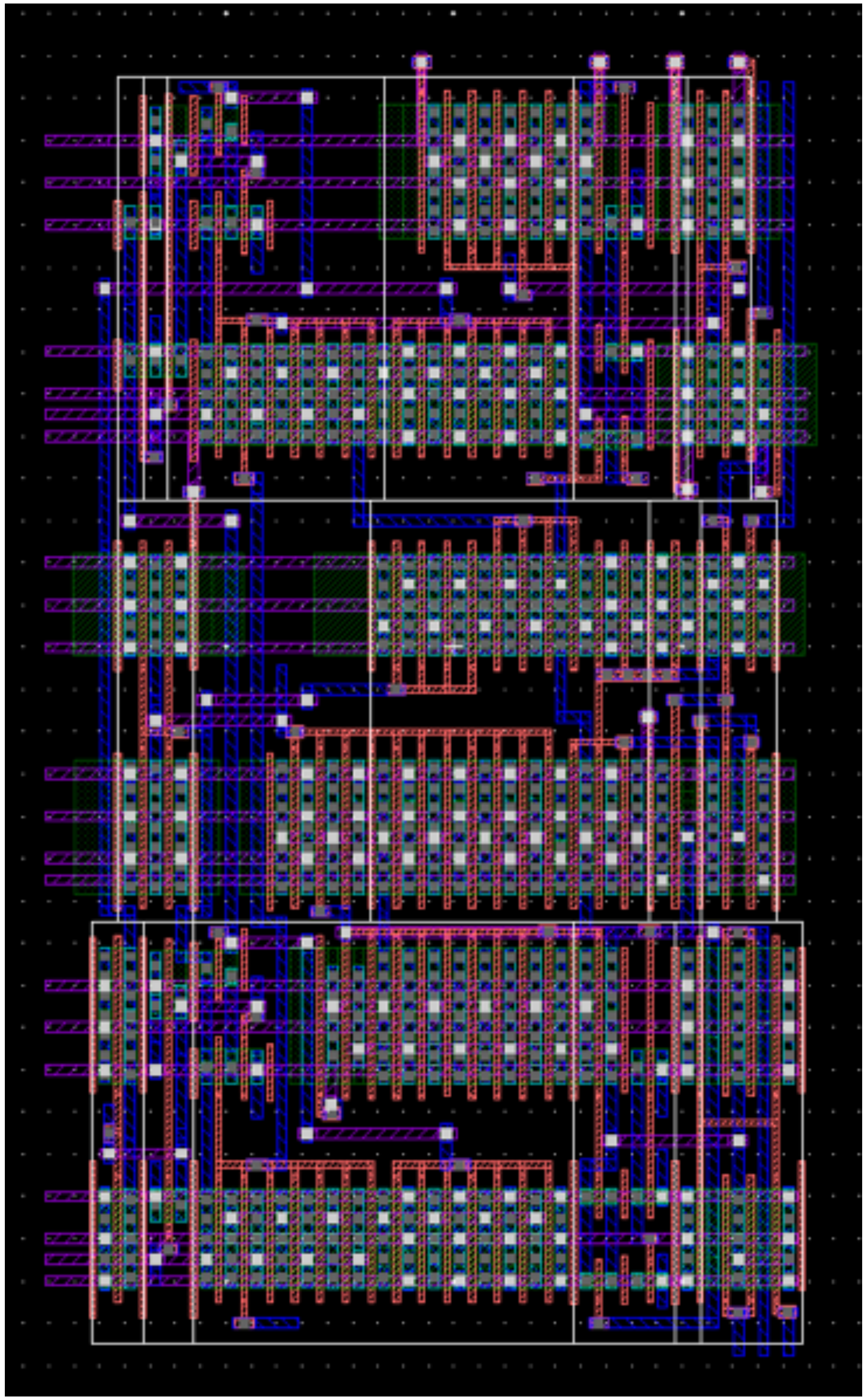
40nm Implementation

- Loadable down counter designed and implemented
- Summary
  - Calibre DRC clean
  - Simulation from *extracted layout*
  - 486 lambda bit pitch (x 3 rows tall)
  - Ready for tape-out (just needs fill+scan)
  - Performance: 13Ghz (76ps cycle), 20ps/bit settling time

40nm Layout

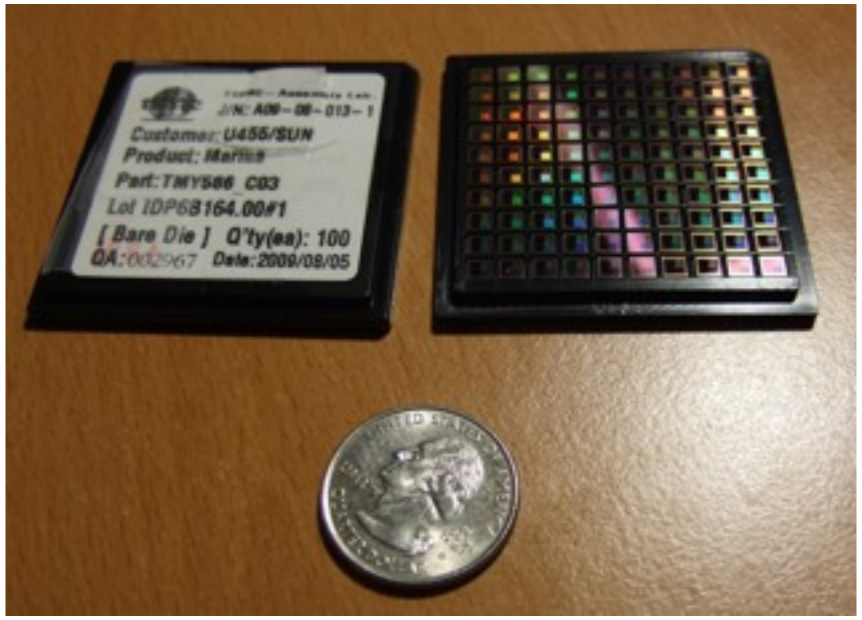
- Calibre DRC clean
- 486λ x 810λ per bit
- M1+M2 only

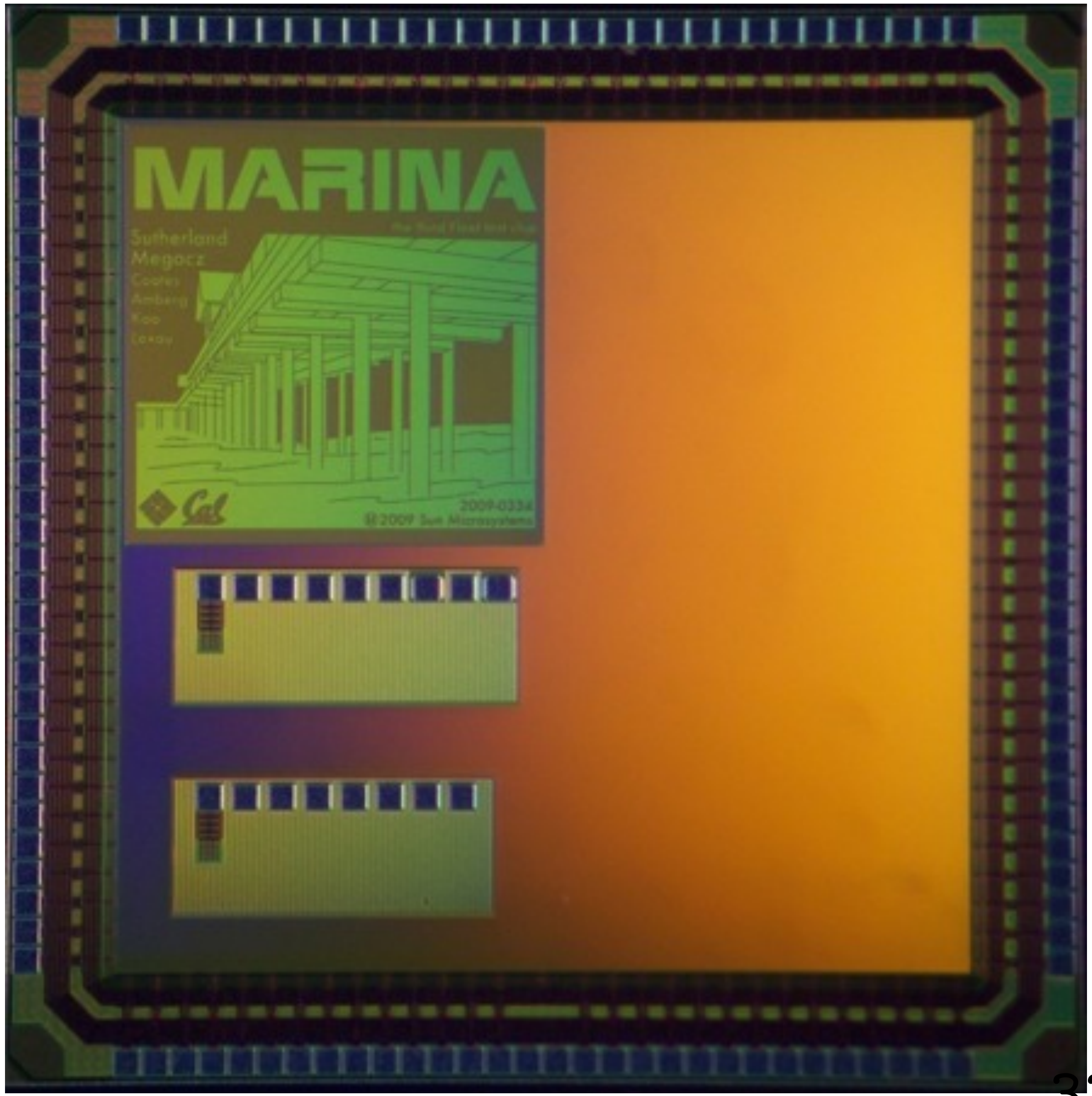




Marina test chip

- Includes earlier 6/4 GasP counter design, 90nm
  - >6 bits wide
  - >Fully interfaced to Dock





Marina Demo

- Caveats
  - > This design was done *before* the 40nm counter I just presented
  - > This design was done in three weeks, conception to tape-out
    - > ... because we finished the main project early and had extra space
  - > This design is in 90nm CMOS, not 40nm
  - > The counter is deployed in an application
    - > No test harness, so measuring performance is a bit tricky

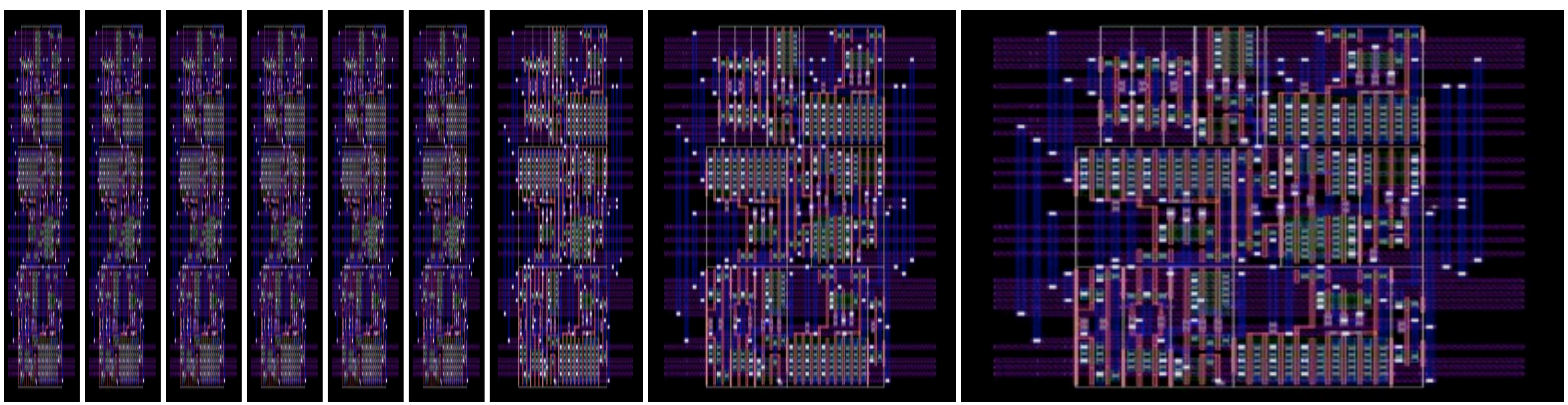
Marina Demo

- Program you will see (for varying values of X):
  - Repeat forever:
    - Load counter with X
    - Run the counter down
    - Send a token
- Token pulses are passed through 16 frequency dividers (each divides by two) before going to the pad.

Demo

The Power of Asynchrony

- Different bits need not be sized the same!
  - > No clock constraint to meet, so:
    - > Size the least significant bits very large (fast, lots of area)
    - > Size the more significant bits exponentially smaller
      - – Down to min-size
      - – Big area savings in large (>=64bit) counters



What does this have to do with Fleet? (*important slide*)

- In a conventional processor, the clock is the "animating force" which drives computation forward
- In Fleet, the "animating force" is a *counter running down*
  - Every asynchronous system is "just" a mass of coupled ring oscillators
  - Voltages move around rings like teeth of gears
  - *Counters* are the engines which drive the network of gears
- Fast, wide counters are important for fast Fleets

Questions?