

Working with DDRs in PrimeTime

Andrew Cheng

Cisco Systems
170 West Tasman Dr.
San Jose, CA 95134

yscheng@cisco.com

Paul Zimmer

Zimmer Design Services
1375 Sun Tree Dr.
Roseville, CA 95661

paulzimmer@zimmerdesignservices.com

ABSTRACT

The timing of I/O interfaces can present some challenges for users of STA tools. This paper will discuss using PrimeTime to tackle one of today's common I/O timing problems – the Double Data Rate (DDR) interface.

Building on the techniques for source-synchronous interfaces covered in last year's "Complex Clocking Situations" paper [1], the authors will demonstrate how to correctly analyze and constrain several different implementations of the DDR interface. In the process, the authors will demonstrate a number of generally useful PrimeTime techniques with application beyond DDR's.

Table of contents

1	Introduction	4
2	Review of the SDR case	5
2.1	SDR input timing constraints	6
2.2	SDR output timing constraints.....	9
3	DDR input timing constraints	13
4	DDR output timing constraints – 2x clock case	19
5	DDR output timing constraints – 1x clock case	25
5.1	The “Latch” Approach	29
5.2	Virtual Clock Approach.....	41
6	Conclusions and Recommendations	47
7	Acknowledgements	48
8	References.....	48

1 Introduction

Double Data Rate (DDR) interfaces are becoming increasingly common in the ASIC world. A DDR interface is a type of source-synchronous interface (meaning the data and the reference clock are both sent from the transmitting device) in which data is sampled on both edges of the output clock.

Most modern I/O interfaces send the clock along with the data. This is called a “source synchronous interface”. Traditionally, data will switch its value at rising edge or falling edge of the clock, but not both. This is called “single data rate” (SDR) and data lasts for one full clock cycle. Lately, a new structure called “double data rate” (DDR) source synchronous interface has become more and more popular in the design world. This technique allows data to be transferred at BOTH rising edges and falling edges of the clock, thus providing twice the bandwidth of single data rate. One famous example is DDR SDRAM as opposed to traditionally SDRAM. Actually, most novel memory types, including QDR SRAM, FCRAM, RLDRAM.....etc all use a DDR structure.

However, I/O timing requirements for DDR interfaces are usually more strict. For example, a “250MHz DDR interface” means the clock between interfaces is running at 250MHz. Since the data is transferred at both edges of the clock, the effective data duration is only half a cycle, which is 2ns! Therefore, it is very important for the designer to perform static timing analysis and make sure all the requirements are met. Unfortunately, this is not an easy job because the logic in DDR design usually involves both posedge flops and negedge flops and even latches. Moreover, the fact that data only lasts for half a cycle needs to be handled carefully in timing analysis.

This paper addresses several techniques in PrimeTime to tackle various implementations in DDR design. First we’ll review the SDR case and point out some common mistakes in using PrimeTime commands to constrain I/O. Then we’ll provide some guidelines to constrain both input and output side of DDR interface and demonstrate how they work. Lastly, a challenging real world example is presented and some useful concepts are discussed to fully analyze the logic in great detail.

2 Review of the SDR case

Before going into the DDR case in detail, let's first review the simpler case of a single-edge clock (SDR) source-synchronous interface.

The basic circuit looks like this:

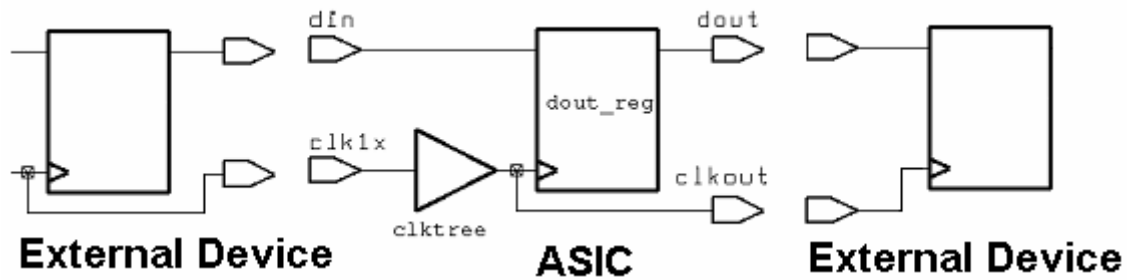


Figure 1

The timing then looks like this:

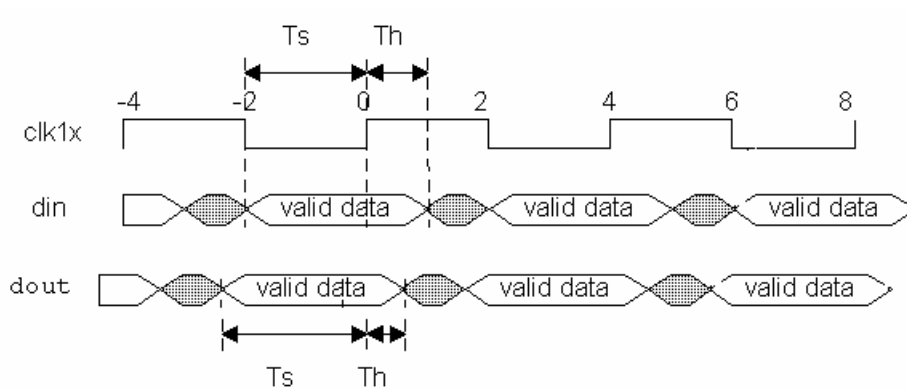


Figure 2

Assume that:

$T_{su}(in) = 2.0$
 $T_{h}(in) = 1.0$
 $T_{su}(out) = 2.2$
 $T_{h}(out) = 0.8$

2.1 SDR input timing constraints

First, we need to create the clock:

```
# Create the clk1x clock
set _period 4.0
create_clock -period $_period -name clk1x \
  [get_ports clk1x]
set_propagated_clock clk1x
```

The input constraints are pretty straightforward. “input_delay” represents the delay of external logic at the input. For the setup case, this is the period minus the setup time. For the hold case, this is the hold time itself. So, the min delay is T_h , and the max delay is $(\text{period} - T_{su})$:

```
# Set the input delays
set_input_delay 1.0 -min -clock clk1x [get_ports din]
set_input_delay [expr $_period - 2.0] -max -clock clk1x [get_ports din] \
  -add_delay
```

We use the “-add_delay” switch to make sure that the second value doesn’t overwrite the first value. While this is not strictly necessary in this case, it’s a good habit to get into.

We can verify our port information using “report_port “:

```
pt_shell> report_port -verbose [get_ports din]
```

Port	Dir	Pin Cap	Wire Cap
din	in	0.0000	0.0000

Port	Max Cap	Min Cap	Max Trans	Max Fanout	Fanout Load
din	--	--	--	--	n/a

Port	External Fanout Number Points	Wire Load Model
din	0	--

Here are our input delays

Input Port	Input Delay				Related Clock
	Min Rise	Min Fall	Max Rise	Max Fall	
din	1.00	1.00	2.00	2.00	clk1x

Input Port	Resistance		Transition	
	Rise	Fall	Rise	Fall
din	--	--	--	--

Note: In all timing traces, we will show the actual command used and the output, except that we have removed the “header” information to save space.

Also, we can look at the timing reports. Here's the max case:

```
pt_shell> report_timing -from [get_ports din] -delay max
```

```
Startpoint: din (input port clocked by clk1x)
Endpoint: dout_reg (rising edge-triggered flip-flop clocked by clk1x)
Path Group: clk1x
Path Type: max
```

Point	Incr	Path
clock clk1x (rise edge)	0.00	0.00
clock network delay (propagated)	0.00	0.00
input external delay	2.00	2.00 f
din (in)	0.00	2.00 f
dout_reg/D (FD1QA)	0.00	2.00 f
data arrival time		2.00
clock clk1x (rise edge)	4.00	4.00
clock network delay (propagated)	0.10	4.10
dout_reg/CP (FD1QA)		4.10 r
library setup time	-0.27	3.83
data required time		3.83
data required time		3.83
data arrival time		-2.00
slack (MET)		1.83

Data is launched externally at time 2.0, which is what we expect. The 0.10 clock network delay is the delay through the clktree buffer. The 0.27 is the setup requirement of the flop.

Here's the min case:

```
pt_shell> report_timing -from [get_ports din] -delay min
```

```
Startpoint: din (input port clocked by clk1x)
Endpoint: dout_reg (rising edge-triggered flip-flop clocked by clk1x)
Path Group: clk1x
Path Type: min
```

Point	Incr	Path
clock clk1x (rise edge)	0.00	0.00
clock network delay (propagated)	0.00	0.00
input external delay	1.00	1.00 r
din (in)	0.00	1.00 r
dout_reg/D (FD1QA)	0.00	1.00 r
data arrival time		1.00

clock clk1x (rise edge)	0.00	0.00
clock network delay (propagated)	0.10	0.10
dout_reg/CP (FD1QA)		0.10 r
library hold time	0.16	0.26
data required time		0.26

data required time		0.26
data arrival time		-1.00

slack (MET)		0.74

Data is launched at time 1.0, which, again, is what we would expect.

2.2 SDR output timing constraints

Now let's look at the output side.

The topic of constraining source-synchronous output interfaces is covered in detail in [1], but the basic technique is to create a generated clock with “-divide_by 1” on the output clock pin and do the set_output_delay constraints relative to this clock:

```
create_generated_clock -name clkout -source [get_ports clk1x] -divide_by 1 \
  [get_ports clkout]
```

What was not dealt with in detail in that paper was proper setting of the output delays, particularly the min case.

Let's start with the max case. The output_delay represents the delay of the external logic. For setup, this is the Tsu requirement itself:

```
set_output_delay 2.2 -clock clkout [get_ports dout] -max
```

And the timing report looks like this:

```
pt_shell> report_timing -to [get_ports dout]
```

```
Startpoint: dout_reg (rising edge-triggered flip-flop clocked by clk1x)
Endpoint: dout (output port clocked by clkout)
Path Group: clkout
Path Type: max
```

Point	Incr	Path

clock clk1x (rise edge)	0.00	0.00
clock network delay (propagated)	0.10	0.10
dout_reg/CP (FD1QA)	0.00	0.10 r
dout_reg/Q (FD1QA)	0.31	0.41 f
dout (out)	0.00	0.41 f
data arrival time		0.41
clock clkout (rise edge)	4.00	4.00
clock network delay (ideal)	0.10	4.10
output external delay	-2.20	1.90
data required time		1.90

data required time		1.90
data arrival time		-0.41

slack (MET)		1.49

And this is correct. If the data were launched at time 0, and checked at time 4.10 (the 0.10 value in “clock network delay (ideal)” is the delay of the clktree buffer – see [1]) and it had to go through 2.2ns of logic, we would indeed end up with 1.49ns of slack.

But what about the min (hold) case? It would seem reasonable to do something like this:

```
set_output_delay 0.8 -clock clkout [get_ports dout] -min -add_delay
```

The resulting timing report would look like this:

```
pt_shell> report_timing -to [get_ports dout] -delay min

Startpoint: dout_reg (rising edge-triggered flip-flop clocked by clk1x)
Endpoint: dout (output port clocked by clkout)
Path Group: clkout
Path Type: min

Point                               Incr          Path
-----
clock clk1x (rise edge)              0.00          0.00
clock network delay (propagated)     0.10          0.10
dout_reg/CP (FD1QA)                  0.00          0.10 r
dout_reg/Q (FD1QA)                   0.30          0.40 r
dout (out)                            0.00          0.40 r
data arrival time                    0.40
-----
clock clkout (rise edge)              0.00          0.00
clock network delay (ideal)          0.10          0.10
output external delay                -0.80         -0.70
data required time                   -0.70
-----
data required time                   -0.70
data arrival time                    -0.40
-----
slack (MET)                           1.10
```

But this isn't correct!

The data came out at time 0.40. The clock came out at time 0.10. This means we had 0.30 ns of hold time. But the requirement is 0.8ns. So, we should have failed by 0.50ns – so the slack should be “-0.50”, not “1.10”.

This illustrates a common mistake people make when starting to use PrimeTime.

What's going on here? Well, think back to the definition of output_delay. It represents the delay through the external logic. Telling PrimeTime that the minimum output_delay value is 0.80 means that there is at least 0.80 ns of delay in the external circuitry. But the conventional definition of hold time is just the opposite. A positive hold time means that the external circuitry NEEDS extra delay in the data path, usually because the clock path is slower than the data path. This corresponds to the external circuitry having NEGATIVE delay. This really means negative delay relative to the clock path.

So, the correct way to describe an output hold requirement of 0.80ns is:

```
set_output_delay [expr -1 * 0.8] -clock clkout [get_ports dout] -min \
-add_delay
```

Now if we report the min timing, we get the expected value of -0.50ns:

```
pt_shell> report_timing -to [get_ports dout] -delay min
```

```
Startpoint: dout_reg (rising edge-triggered flip-flop clocked by clk1x)
Endpoint: dout (output port clocked by clkout)
Path Group: clkout
Path Type: min
```

Point	Incr	Path

clock clk1x (rise edge)	0.00	0.00
clock network delay (propagated)	0.10	0.10
dout_reg/CP (FD1QA)	0.00	0.10 r
dout_reg/Q (FD1QA)	0.30	0.40 r
dout (out)	0.00	0.40 r
data arrival time		0.40
clock clkout (rise edge)	0.00	0.00
clock network delay (ideal)	0.10	0.10
output external delay	0.80	0.90
data required time		0.90

data required time		0.90
data arrival time		-0.40

slack (VIOLATED)		-0.50

3 DDR input timing constraints

In a DDR interface, the data signal can now change its value at both rising and falling edges of the clock. Therefore, on the input side, both posedge and negedge flops will be used to capture the input data:

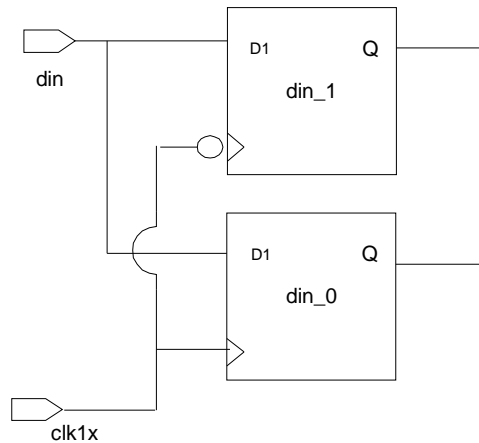


Figure 3

Suppose the timing of input data/clock pair looks like this:

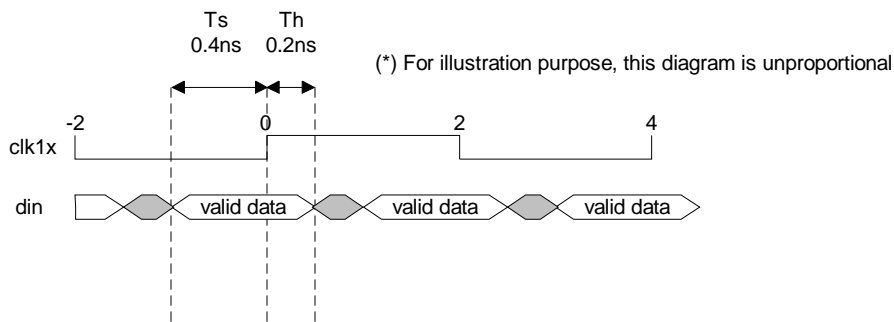


Figure 4

The posedge flop (*din_0*) will be used to capture the input data centered at $t=0,4,8,\dots$ etc while the negedge flop (*din_1*) will be used to capture the input data centered at $t=2,6,10,\dots$ etc.

Below are the guidelines for setting input constraints in DDR interface:

1. Define input clock.
2. Set_input_delay with respect to rising edge of the clock.
3. Set_input_delay with respect to falling edge of the clock using “-clock_fall” switch.

In our example, the corresponding steps will be:

```
create_clock -period $_period -name CLK1x [get_ports clkin]

set_input_delay -clock CLK1X -max [expr $_period / 2 - 0.4] din
set_input_delay -clock CLK1X -min 0.2 din -add_delay
set_input_delay -clock CLK1X -max [expr $_period / 2 - 0.4] din -add_delay \
-clock_fall
set_input_delay -clock CLK1X -min 0.2 din -add_delay -clock_fall
```

Notice that it takes four lines to completely describe the input constraints for a specific input data port: two for setup and hold with respect to rising edge of input clock, and two for setup and hold with respect to falling edge of input clock. Note that we use “-add_delay” switch to prevent the latter input delay value from overwriting the former input delay value. Also note that in max cases we use “expr \$_period / 2 - 0.4” as opposed to “expr \$_period - 0.4” because in DDR case our input data length is only 2ns instead of 4ns in SDR case.

Let's look at the report and see whether PrimeTime does correctly time the path from data input port din to both flops:

```
pt_shell> report_timing -from [get_ports din] -nworst 2
```

```
Startpoint: din (input port clocked by CLK1X)
Endpoint: din_1 (falling edge-triggered flip-flop clocked by CLK1X)
Path Group: CLK1X
Path Type: max
```

Point	Incr	Path
clock CLK1X (rise edge)	0.00	0.00
clock network delay (propagated)	0.00	0.00
input external delay	1.60	1.60 r
din (in)	0.00	1.60 r
din_1/D1 (FL1S2A0V15)	0.00	1.60 r
data arrival time		1.60
clock CLK1X (fall edge)	2.00	2.00
clock network delay (propagated)	0.11	2.11
din_1/CK (FL1S2A0V15)		2.11 f
library setup time	-0.07	2.04
data required time		2.04
data required time		2.04
data arrival time		-1.60
slack (MET)		0.44

```
Startpoint: din (input port clocked by CLK1X)
Endpoint: din_0 (rising edge-triggered flip-flop clocked by CLK1X)
Path Group: CLK1X
Path Type: max
```

Point	Incr	Path
clock CLK1X (fall edge)	2.00	2.00
clock network delay (propagated)	0.00	2.00
input external delay	1.60	3.60 f
din (in)	0.00	3.60 f
din_0/D1 (FL1S3ANV15)	0.00	3.60 f
data arrival time		3.60
clock CLK1X (rise edge)	4.00	4.00
clock network delay (propagated)	0.12	4.12
din_0/CK (FL1S3ANV15)		4.12 r
library setup time	-0.02	4.10
data required time		4.10
data required time		4.10
data arrival time		-3.60
slack (MET)		0.50

From the report, we can observe two points: first, PrimeTime now checks not only data from rising edge of input clock but also data from falling edge because we use that “-clock_fall” switch. This is exactly what we want in DDR case; second, PrimeTime checks data from falling edge (t=2) against rising edge of the clock (t=4). This is also correct because input data now only lasts for 2ns.

If we ask PrimeTime to report all the timing paths from din, we will get eight paths in the report. Four of these are like the one shown above (pos clock to neg clock with rising data, pos clock to neg clock with falling data, neg clock to pos clock with rising data, neg clock to pos clock with falling data).

The other four are full-cycle paths like this:

```
Startpoint: din (input port clocked by CLK1X)
Endpoint:  din_0 (rising edge-triggered flip-flop clocked by CLK1X)
Path Group: CLK1X
Path Type: max
```

Point	Incr	Path
clock CLK1X (rise edge)	0.00	0.00
clock network delay (propagated)	0.00	0.00
input external delay	1.60	1.60 f
din (in)	0.00	1.60 f
din_0/D1 (FL1S3ANV15)	0.00	1.60 f
data arrival time		1.60
clock CLK1X (rise edge)	4.00	4.00
clock network delay (propagated)	0.12	4.12
din_0/CK (FL1S3ANV15)		4.12 r
library setup time	-0.02	4.10
data required time		4.10
data required time		4.10
data arrival time		-1.60
slack (MET)		2.50

These paths will always have more slack than the corresponding half-cycle paths, so they don't really matter.

When we use “report_timing” command in PrimeTime, by default we will get only one path --- the one which has the worst slack. In DDR interface, whether at input side or output side, it is always a good habit to report more paths by using the “-nworst” switch. This usually will give us more information.

The hold report looks like this:

```
pt_shell> report_timing -from [get_ports din] -nworst 2 -delay min
```

```
Startpoint: din (input port clocked by CLK1X)
Endpoint: din_1 (falling edge-triggered flip-flop clocked by CLK1X)
Path Group: CLK1X
Path Type: min
```

Point	Incr	Path
clock CLK1X (fall edge)	2.00	2.00
clock network delay (propagated)	0.00	2.00
input external delay	0.20	2.20 f
din (in)	0.00	2.20 f
din_1/D1 (FL1S2AQV15)	0.00	2.20 f
data arrival time		2.20
clock CLK1X (fall edge)	2.00	2.00
clock network delay (propagated)	0.11	2.11
din_1/CK (FL1S2AQV15)		2.11 f
library hold time	0.06	2.18
data required time		2.18
data required time		2.18
data arrival time		-2.20
slack (MET)		0.02

```
Startpoint: din (input port clocked by CLK1X)
Endpoint: din_0 (rising edge-triggered flip-flop clocked by CLK1X)
Path Group: CLK1X
Path Type: min
```

Point	Incr	Path
clock CLK1X (rise edge)	0.00	0.00
clock network delay (propagated)	0.00	0.00
input external delay	0.20	0.20 r
din (in)	0.00	0.20 r
din_0/D1 (FL1S3ANV15)	0.00	0.20 r
data arrival time		0.20
clock CLK1X (rise edge)	0.00	0.00
clock network delay (propagated)	0.12	0.12
din_0/CK (FL1S3ANV15)		0.12 r
library hold time	0.00	0.12
data required time		0.12
data required time		0.12
data arrival time		-0.20
slack (MET)		0.08

Again, PrimeTime now checks those input data from both rising edges of the input clock ($t=0$) and falling edges of the clock ($t=2$).

4 DDR output timing constraints – 2x clock case

For DDR output timing constraints, we have similar guidelines:

1. Define output clock.
2. Set_output_delay with respect to rising edge of the clock.
3. Set_output_delay with respect to falling edge of the clock using “-clock_fall” switch.

However, the constraints at the DDR output side are usually more complicated than those at the input side. It is not so trivial to correctly define the output clock, and sometimes it would make our lives easier if we define some virtual clocks to help us constrain the output port. We’ll see that in later sections. Now let’s first focus on one simple example to get the basic idea about DDR output constraints.

If we have internal 2x clock and we want to generate DDR output clock/data pair, usually we’ll use a posedge flop clocked by clk2x for the output data path and a divided_by_two logic for the output clock path:

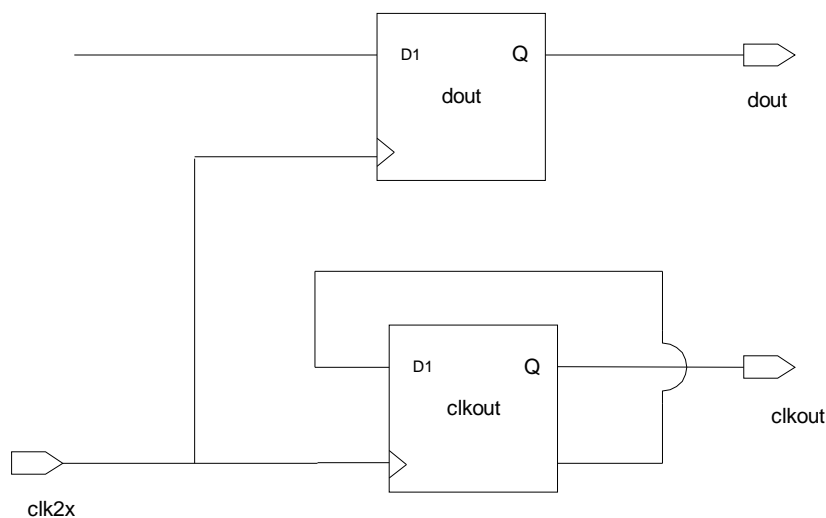


Figure 5

Supposed the desired output timing looks like this:

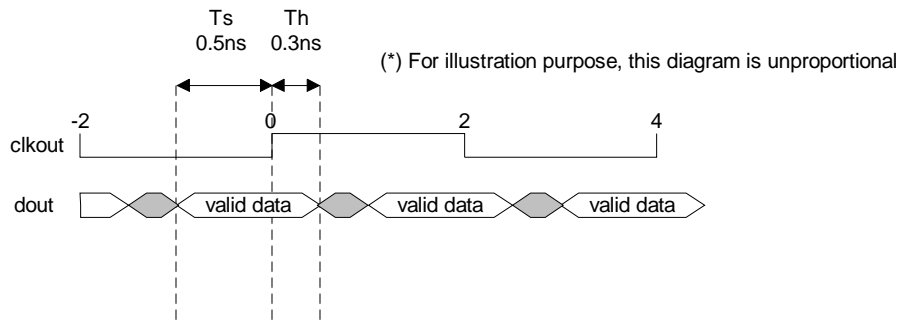


Figure 6

First we need to define the output clock (for more details, see [1]):

```
create_generated_clock -name CLKOUT -source [get_ports clk2x] -divide_by 2
[get_ports clkout]
```

Then we set the output delay according to the requirement:

```
set_output_delay -clock CLKOUT -max 0.5 [get_ports dout]
set_output_delay -clock CLKOUT -min [expr -1 * 0.3] [get_ports dout]
-add_delay
set_output_delay -clock CLKOUT -max 0.5 [get_ports dout] -clock_fall
-add_delay
set_output_delay -clock CLKOUT -min [expr -1 * 0.3] [get_ports dout]
-clock_fall -add_delay
```

Again, it takes four “set_output_delay” commands to fully describe one output port: two for setup and hold with respect to rising edge of output clock, and two for setup and hold with respect to falling edge of output clock. Also note that we put “-0.3”, not “0.3” in min cases.

Now let's take a look at the setup check report. We intentionally use "-nworst" switch with a large number for PrimeTime to report all timing paths to output port dout:

```
pt_shell> report_timing -to dout -nworst 20
```

```
Startpoint: dout (rising edge-triggered flip-flop clocked by CLK2X)
Endpoint: dout (output port clocked by CLKOUT)
Path Group: CLKOUT
Path Type: max
```

Point	Incr	Path

clock CLK2X (rise edge)	2.00	2.00
clock network delay (propagated)	0.10	2.10
dout/CK (FL1S3ANV15)	0.00	2.10 r
dout/Q (FL1S3ANV15)	0.29	2.39 f
dout (out)	0.00	2.39 f
data arrival time		2.39
clock CLKOUT (rise edge)	4.00	4.00
clock network delay (ideal)	0.35	4.35
output external delay	-0.50	3.85
data required time		3.85

data required time		3.85
data arrival time		-2.39

slack (MET)		1.47

```
Startpoint: dout (rising edge-triggered flip-flop clocked by CLK2X)
Endpoint: dout (output port clocked by CLKOUT)
Path Group: CLKOUT
Path Type: max
```

Point	Incr	Path

clock CLK2X (rise edge)	2.00	2.00
clock network delay (propagated)	0.10	2.10
dout/CK (FL1S3ANV15)	0.00	2.10 r
dout/Q (FL1S3ANV15)	0.25	2.35 r
dout (out)	0.00	2.35 r
data arrival time		2.35
clock CLKOUT (rise edge)	4.00	4.00
clock network delay (ideal)	0.35	4.35
output external delay	-0.50	3.85
data required time		3.85

data required time		3.85
data arrival time		-2.35

slack (MET)		1.50

```
Startpoint: dout (rising edge-triggered flip-flop clocked by CLK2X)
Endpoint: dout (output port clocked by CLKOUT)
Path Group: CLKOUT
Path Type: max
```

Point	Incr	Path

clock CLK2X (rise edge)	0.00	0.00
clock network delay (propagated)	0.10	0.10
dout/CK (FL1S3ANV15)	0.00	0.10 r
dout/Q (FL1S3ANV15)	0.29	0.39 f
dout (out)	0.00	0.39 f
data arrival time		0.39
clock CLKOUT (fall edge)	2.00	2.00
clock network delay (ideal)	0.39	2.39
output external delay	-0.50	1.89
data required time		1.89

data required time		1.89
data arrival time		-0.39

slack (MET)		1.50

Startpoint: dout (rising edge-triggered flip-flop clocked by CLK2X)
 Endpoint: dout (output port clocked by CLKOUT)
 Path Group: CLKOUT
 Path Type: max

Point	Incr	Path

clock CLK2X (rise edge)	0.00	0.00
clock network delay (propagated)	0.10	0.10
dout/CK (FL1S3ANV15)	0.00	0.10 r
dout/Q (FL1S3ANV15)	0.25	0.35 r
dout (out)	0.00	0.35 r
data arrival time		0.35
clock CLKOUT (fall edge)	2.00	2.00
clock network delay (ideal)	0.39	2.39
output external delay	-0.50	1.89
data required time		1.89

data required time		1.89
data arrival time		-0.35

slack (MET)		1.53

Totally we got four timing paths. Data launched at t=2 was captured at t=4, while data launched at t=0 was captured at t=2. If we didn't use "-clock_fall" switch to further constrain the output port, we would only get the first two paths. The third and fourth paths, which correspond to the falling edge of output clock, would not show up.

The hold report will look like this:

```
pt_shell> report_timing -to dout -nwor 9 -delay min
```

```
Startpoint: dout (rising edge-triggered flip-flop clocked by CLK2X)
Endpoint: dout (output port clocked by CLKOUT)
Path Group: CLKOUT
Path Type: min
```

Point	Incr	Path
clock CLK2X (rise edge)	2.00	2.00
clock network delay (propagated)	0.10	2.10
dout/CK (FL1S3ANV15)	0.00	2.10 r
dout/Q (FL1S3ANV15)	0.25	2.35 r
dout (out)	0.00	2.35 r
data arrival time		2.35
clock CLKOUT (fall edge)	2.00	2.00
clock network delay (ideal)	0.39	2.39
output external delay	0.30	2.69
data required time		2.69
data required time		2.69
data arrival time		-2.35
slack (VIOLATED)		-0.33

Now let's trace where that "clock network delay" of clkout (0.35 and 0.39) comes from:

```
pt_shell> report_timing -to clkout -nworst 4
```

```
Startpoint: clkout (rising edge-triggered flip-flop clocked by CLK2X)
Endpoint: clkout (output port)
Path Group: (none)
Path Type: max
```

Point	Incr	Path
clock network delay (propagated)	0.10	0.10
clkout/CK (FL1S3ANV15)	0.00	0.10 r
clkout/Q (FL1S3ANV15)	0.29	0.39 f
clkout (out)	0.00	0.39 f
data arrival time		0.39

(Path is unconstrained)

```
Startpoint: clkout (rising edge-triggered flip-flop clocked by CLK2X)
Endpoint: clkout (output port)
Path Group: (none)
Path Type: max
```

Point	Incr	Path
clock network delay (propagated)	0.10	0.10
clkout/CK (FL1S3ANV15)	0.00	0.10 r
clkout/Q (FL1S3ANV15)	0.25	0.35 r
clkout (out)	0.00	0.35 r
data arrival time		0.35

(Path is unconstrained)

Even though we intentionally ask PrimeTime to report 4 worst paths for output clock, in reality there're only two paths, which are associated with rising edges and falling edges of clkout. Compare this to the SDR case: in SDR we don't set_output_delay with respect to the falling edge of the clock, so the clock network delay for output clock will always be "0.35" in all max delay reports and min delay reports. That "0.39" will never show up in timing report.

5 DDR output timing constraints – 1x clock case

Authors' note:

This section of the paper is out of date. For an updated treatment of the 1x clock output circuit, please see Paul Zimmer's paper "Getting DDRs write – the 1x output circuit revisited". It is available on Paul's web site (www.zimmerdesignservices.com) or at www.snug-universal.org (see Proceedings from SNUG San Jose 2006).

If there is no 2x clock available, the DDR transmit circuit might be implemented something like this:

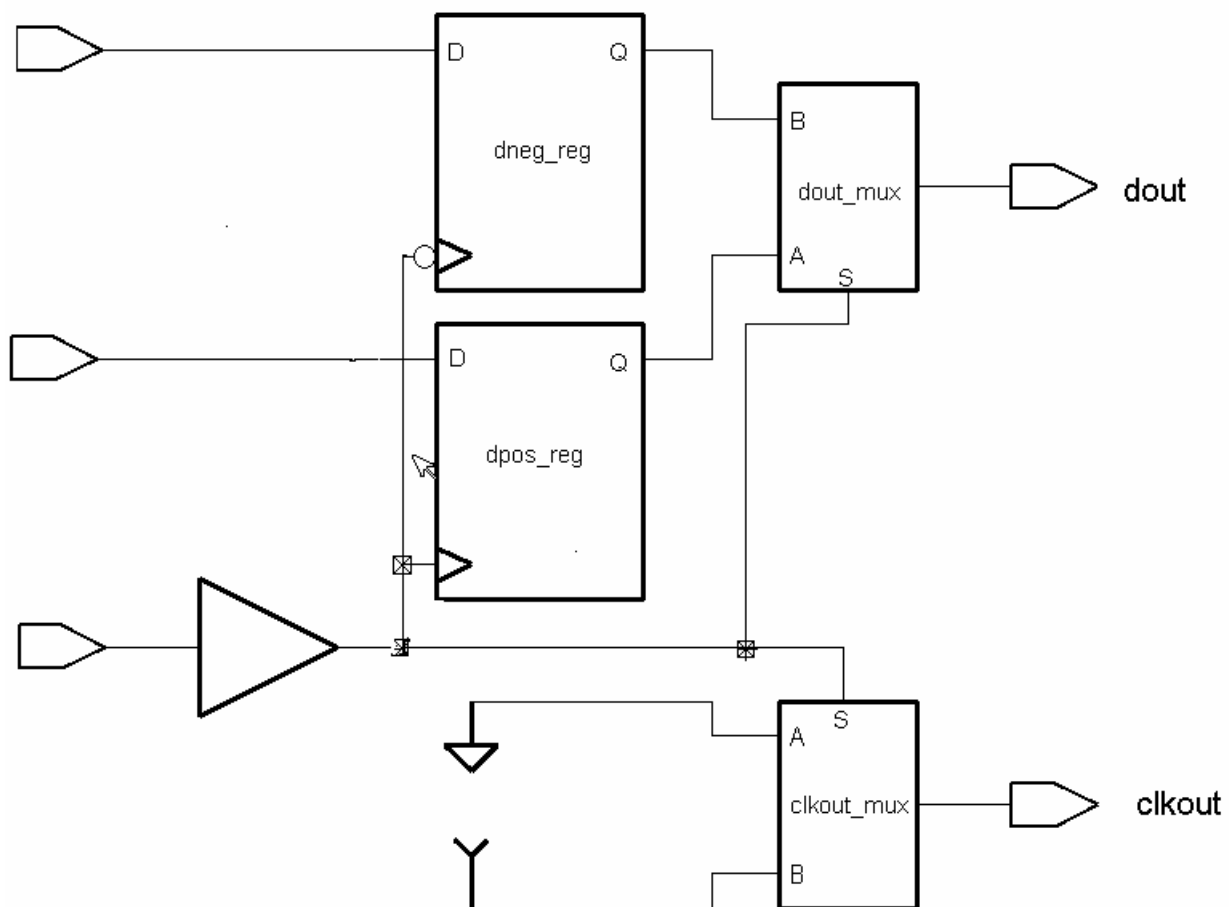


Figure 7

Two output flops are used – one for rising edge data and one for falling edge data. These two data streams are then multiplexed to create the output data. The mux is controlled by the phase of the 1x clock. The output clock is then the 1x clock itself.

In a real implementation, the designer often tries to match the path lengths as much as possible so that the data and clock will “track” each other over process/voltage/temp. In this circuit, we have added a matching mux to the clock path for this reason.

The timing looks like this:

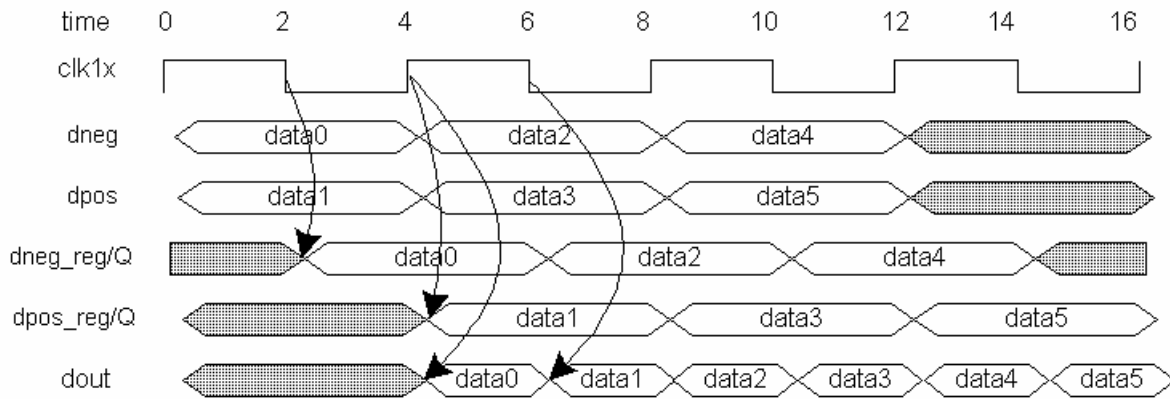


Figure 8

So, how do we time this with PrimeTime?

Using the techniques shown earlier, we might try something like this:

```
create_clock -period $_period -name clk1x \
  [get_ports clk1x]
set_propagated_clock clk1x

create_generated_clock -name clkout -source [get_ports clk1x] -divide_by 1 \
  [get_ports clkout]

set_output_delay 0.5 -clock clkout [get_ports dout] -max
set_output_delay [expr -1 * 0.3] -clock clkout [get_ports dout] -min -
add_delay
set_output_delay 0.5 -clock clkout [get_ports dout] -max -clock_fall -
add_delay
set_output_delay [expr -1 * 0.3] -clock clkout [get_ports dout] -min -
clock_fall -add_delay
```

If we now report timing to the dout pin, we get:

```
pt_shell> report_timing -to dout
```

```
Startpoint: dneg_reg (falling edge-triggered flip-flop clocked by clk1x)
Endpoint: dout (output port clocked by clkout)
Path Group: clkout
Path Type: max
```

Point	Incr	Path

clock clk1x (fall edge)	2.00	2.00
clock network delay (propagated)	0.18	2.18
dneg_reg/CPN (FDN1QA)	0.00	2.18 f
dneg_reg/Q (FDN1QA)	0.34	2.52 r
dout_mux/Z (MUX21HA)	0.15	2.67 r
dout (out)	0.00	2.67 r
data arrival time		2.67
clock clkout (rise edge)	4.00	4.00
clock network delay (ideal)	0.31	4.31
output external delay	-0.50	3.81
data required time		3.81

data required time		3.81
data arrival time		-2.67

slack (MET)		1.14

But this isn't correct. The data switched at time 2 by dneg_reg won't be sampled on the interface until the FALLING edge of clkout. There is a similar path in the other direction (dpos_reg to falling clkout).

The min path calculations *ARE* correct:

```
pt_shell> report_timing -delay min -to [get_ports dout]
```

```
Startpoint: clk1x (clock source 'clk1x')
Endpoint: dout (output port clocked by clkout)
Path Group: clkout
Path Type: min
```

Point	Incr	Path

clock clk1x (fall edge)	2.00	2.00
clk1x (in)	0.00	2.00 f
clktree/Z (BUFC)	0.18	2.18 f
dout_mux/Z (MUX21HA)	0.21	2.39 f
dout (out)	0.00	2.39 f
data arrival time		2.39
clock clkout (fall edge)	2.00	2.00
clock network delay (ideal)	0.39	2.39
output external delay	0.30	2.69
data required time		2.69

data required time		2.69
data arrival time		-2.39

slack (VIOLATED)		-0.30

```
pt_shell> report_timing -delay min -from [get_pins dneg_reg/Q]
```

```
Startpoint: dneg_reg (falling edge-triggered flip-flop clocked by clk1x)
Endpoint: dout (output port clocked by clkout)
Path Group: clkout
Path Type: min
```

Point	Incr	Path

clock clk1x (fall edge)	2.00	2.00
clock network delay (propagated)	0.18	2.18
dneg_reg/CPN (FDN1QA)	0.00	2.18 f
dneg_reg/Q (FDN1QA) <-	0.29	2.47 f
dout_mux/Z (MUX21HA)	0.18	2.65 f
dout (out)	0.00	2.65 f
data arrival time		2.65
clock clkout (fall edge)	2.00	2.00
clock network delay (ideal)	0.39	2.39
output external delay	0.30	2.69
data required time		2.69

data required time		2.69
data arrival time		-2.65

slack (VIOLATED)		-0.04

What we want to do is tell PrimeTime that the path from dneg_reg/CPN to the RISING edge of clkout is false – at least for the setup check.. But, unfortunately, PrimeTime doesn't have any such option on set_false_path. It has has a “-rise_to” option, but this doesn't do what we want – even with the “to” point being the clkout clock itself:

```
pt_shell> set_false_path -from [get_pins dneg_reg/CPN] -rise_to [get_clocks
clkout]
1
pt_shell> report_timing -to dout
```

```
Startpoint: dneg_reg (falling edge-triggered flip-flop clocked by clk1x)
Endpoint: dout (output port clocked by clkout)
Path Group: clkout
Path Type: max
```

Point	Incr	Path

clock clk1x (fall edge)	2.00	2.00
clock network delay (propagated)	0.18	2.18
dneg_reg/CPN (FDN1QA)	0.00	2.18 f
dneg_reg/Q (FDN1QA)	0.29	2.47 f
dout_mux/Z (MUX21HA)	0.18	2.65 f
dout (out)	0.00	2.65 f
data arrival time		2.65
clock clkout (rise edge)	4.00	4.00
clock network delay (ideal)	0.31	4.31
output external delay	-0.50	3.81
data required time		3.81

data required time		3.81
data arrival time		-2.65

slack (MET)		1.16

All that this did was to false path the rise edge AT THE ENDPOINT, which is dout.

Another alternative would be to do set_multicycle_path of some large number to make this path go away, but again, PrimeTime doesn't have any options to do path exceptions to EDGES of clocks.

What to do?

5.1 The “Latch” Approach

Well, one approach is to notice that the dout_mux behaves something like a latch. Actually, it behaves more like a clock gating element, but we have been calling this the “latch” solution for so long that we'll stay with that name.

As long as the path from each flip-flop to the dout_mux is less than half a cycle, the critical path will be from the clk1x input, through the dout_mux “S” (select) input. So, we can apply a

technique where we ASSUME that the “latch condition”, in this case that the flop-to-mux time is less than half a cycle, is true, time accordingly, then check the assumption.

First, we’ll set false paths from the flops to get them out of the timing reports:

```
set_false_path -setup -from [get_pins dpos_reg/CP] -to [get_ports dout]
set_false_path -setup -from [get_pins dneg_reg/CPN] -to [get_ports dout]
```

Now, the timing report looks like this:

```
pt_shell> report_timing -to dout
```

```
Startpoint: clk1x (clock source 'clk1x')
Endpoint: dout (output port clocked by clkout)
Path Group: clkout
Path Type: max
```

Point	Incr	Path

clock clk1x (fall edge)	2.00	2.00
clk1x (in)	0.00	2.00 f
clktree/Z (BUFC)	0.18	2.18 f
dout_mux/Z (MUX21HA)	0.23	2.41 r
dout (out)	0.00	2.41 r
data arrival time		2.41
clock clkout (rise edge)	4.00	4.00
clock network delay (ideal)	0.31	4.31
output external delay	-0.50	3.81
data required time		3.81

data required time		3.81
data arrival time		-2.41

slack (MET)		1.40

This is the path we're looking for. Similarly, there is the other clock path:

```
pt_shell> report_timing -rise_from [get_ports clk1x] -to dout
```

```
Startpoint: clk1x (clock source 'clk1x')
Endpoint: dout (output port clocked by clkout)
Path Group: clkout
Path Type: max
```

Point	Incr	Path

clock clk1x (rise edge)	0.00	0.00
clk1x (in)	0.00	0.00 r
clktree/Z (BUFC)	0.14	0.14 r
dout_mux/Z (MUX21HA)	0.25	0.40 f
dout (out)	0.00	0.40 f
data arrival time		0.40
clock clkout (fall edge)	2.00	2.00
clock network delay (ideal)	0.39	2.39
output external delay	-0.50	1.89
data required time		1.89

data required time		1.89
data arrival time		-0.40

slack (MET)		1.49

But we still have to verify that our assumption that the delays from the flops to the dout_mux are less than half a cycle. Actually, it's a little more complicated than that. What we really want to verify is that the time from the flops through their dout_mux pins is less than the delay from clk1x input through the dout_mux select pin, plus half a cycle. In other words, we want to verify that the data from the flops gets through the mux before the clock (which is delayed by half a cycle).

We'll use set_max_delay to do this, but first we need to know the delay through the S pin. We'll use get_timing_paths to get this. We need separate values for the rising and falling case. Also, the timing we're looking for is all the way through the mux to the mux output (Z) pin.

Let's start with the rise case:

```
set_path [get_timing_paths -rise_through dout_mux/S]
foreach_in_collection point [get_attribute $path points] {
  set object [get_attribute $point object]
  set point_name [get_attribute $object full_name]
  if {$point_name == "dout_mux/Z"} {
    set _srise_delay [get_attribute $point arrival]
  }
}
echo "_srise_delay is $_srise_delay"
```

When we source this, we get:

```
pt_shell> source temp.pt
_srise_delay is 0.397521
```

Whenever using `get_timing_paths`, it generally a good idea to check the result:

```
pt_shell> report_timing -to dout_mux/S -rise_through dout_mux/S
```

```
Startpoint: clk1x (clock source 'clk1x')
Endpoint: dout (output port clocked by clkout)
Path Group: clkout
Path Type: max
```

Point	Incr	Path

clock clk1x (rise edge)	0.00	0.00
clk1x (in)	0.00	0.00 r
clktree/Z (BUFC)	0.14	0.14 r
dout_mux/S (MUX21HA) <-	0.00	0.14 r
dout_mux/Z (MUX21HA)	0.25	0.40 f
dout (out)	0.00	0.40 f
data arrival time		0.40

clock clkout (fall edge)	2.00	2.00
clock network delay (ideal)	0.39	2.39
output external delay	-0.50	1.89
data required time		1.89

data required time		1.89
data arrival time		-0.40

slack (MET)		1.49

Yep, 0.40 is what we should get.

Now, the falling edge:

```
set _path [get_timing_paths -fall_through dout_mux/S]
foreach_in_collection point [get_attribute $_path points] {
  set object [get_attribute $point object]
  set point_name [get_attribute $object full_name]
  if {$point_name == "dout_mux/Z"} {
    set _sfall_delay [get_attribute $point arrival]
  }
}
echo "_sfall_delay is $_sfall_delay"
```

And we get:

```
pt_shell> source temp.pt
_sfall_delay is 0.409921
```

Now, check it:

```
pt_shell> report_timing -to dout_mux/S -fall_through dout_mux/S
```

```
Startpoint: clk1x (clock source 'clk1x')
Endpoint: dout (output port clocked by clkout)
Path Group: clkout
Path Type: max
```

Point	Incr	Path

clock clk1x (fall edge)	2.00	2.00
clk1x (in)	0.00	2.00 f
clktree/Z (BUFC)	0.18	2.18 f
dout_mux/S (MUX21HA) <-	0.00	2.18 f
dout_mux/Z (MUX21HA)	0.23	2.41 r
dout (out)	0.00	2.41 r
data arrival time		2.41

clock clkout (rise edge)	4.00	4.00
clock network delay (ideal)	0.31	4.31
output external delay	-0.50	3.81
data required time		3.81

data required time		3.81
data arrival time		-2.41

slack (MET)		1.40

The answer is what we wanted (0.41), but notice that the report_timing trace looks different. It has the clock fall edge timing added to it. So, why does the “arrival” attribute on that point in get_timing_paths return “0.41”, but in the report_timing it looks like “2.41”. We have no idea, but this is why one should ALWAYS check get_timing_path results!!

So, now we can apply our max_delay constraint to make sure that the flops are not the critical path. Since dpos_reg’s data is sent through the mux on the falling clock edge, we’ll use the _sfall_delay timing for dpos_reg. Similarly, we’ll use the _srise_delay timing for dneg_reg:

```
set_max_delay [expr 2 + $_sfall_delay] \  
-from [get_pins dpos_reg/CP] \  
-to [get_pins dout_mux/Z]  
set_max_delay [expr 2 + $_srise_delay] \  
-from [get_pins dneg_reg/CPN] \  
-to [get_pins dout_mux/Z]
```

Except, this still isn’t quite right. The problem is that putting a set_max_delay on the output pin of the mux will create a timing endpoint there, which will kill the path through this pin from clk1x, which is definitely not what we want to do.

The only way around this is to do the set_max_delay on the input pins of the mux. But, to be accurate, we need to do this with the time through the mux already accounted for by subtracting this amount from our set_max_delay value.

This isn't going to be pretty.

First, we have to get the delay through the mux via the A and B (D0 and D1) pins. To be completely accurate, we need to do this separately for rise and fall. This is a little tedious, but it does illustrate some useful techniques. Here's one way to get the data for paths through the A pin:

```
set _path [get_timing_paths -rise_through dout_mux/A]
foreach_in_collection point [get_attribute $_path points] {
  set object [get_attribute $point object]
  set point_name [get_attribute $object full_name]
  if {$point_name == "dout_mux/A"} {
    set _arise_delay [get_attribute $point arrival]
  }
  if {$point_name == "dout_mux/Z"} {
    set _azrise_delay [get_attribute $point arrival]
  }
}
echo "_arise_delay is [format %2.2f $_arise_delay]"
echo "_azrise_delay is [format %2.2f $_azrise_delay]"
```

This is similar to how we got the S path timing, except we look explicitly for the point names "dout_mux/A" and "dout_mux/Z". Also, note the use of the "format" command to produce output with 2 decimal places. Source this code, and we get:

```
pt_shell> source temp.pt
_arise_delay is 0.34
_azrise_delay is 0.51
```

As always, we check this with report_timing:

```
pt_shell> report_timing -input_pins -rise_through dout_mux/A

Startpoint: dpos_reg (rising edge-triggered flip-flop clocked by clk1x)
Endpoint: dout (output port clocked by clkout)
Path Group: (none)
Path Type: max

Point                                     Incr      Path
-----
clock network delay (propagated)         0.14      0.14
dpos_reg/CP (FD1QA)                       0.00      0.14 r
dpos_reg/Q (FD1QA)                       0.34      0.48 r
dout_mux/A (MUX21HA) <-                   0.00      0.48 r
dout_mux/Z (MUX21HA)                     0.17      0.66 r
dout (out)                                0.00      0.66 r
data arrival time                        0.66
-----
(Path is unconstrained)
```

More strangeness. Our values are correct in the sense that `_arise_delay` is the delay from the flop clock edge to the A pin of the mux (0.34), and `_azrise_delay` is the delay from the A pin of the mux to the Z pin of the mux (0.34 + 0.17 = 0.51). But that 0.14 clock network delay shows up in

the timing report, but doesn't show up in `get_timing_paths`. In this instance, it turns out that it doesn't matter. But, still, this difference between `report_timing` and `get_timing_paths` is disconcerting.

Similar code gets the fall delay timing (we need to get it now, before doing the `set_max_delay`, because the `set_max_delay` will create a timing endpoint that will hide the timing we're looking for).

```
set _path [get_timing_paths -fall_through dout_mux/A]
foreach_in_collection point [get_attribute $_path points] {
  set object [get_attribute $point object]
  set point_name [get_attribute $object full_name]
  if {$point_name == "dout_mux/A"} {
    set _afall_delay [get_attribute $point arrival]
  }
  if {$point_name == "dout_mux/Z"} {
    set _azfall_delay [get_attribute $point arrival]
  }
}
echo "_afall_delay is [format %2.2f $_afall_delay]"
echo "_azfall_delay is [format %2.2f $_azfall_delay]"
```

```
pt_shell> source temp.pt
_afall_delay is 0.34
_azfall_delay is 0.52
```

Now, we can use this information to constrain the flop path using `set_max_delay`. The maximum delay (latest arrival time) at `dout_mux/A` is half a cycle plus the delay from S through the mux, minus the delay from A through the mux:

```
set_max_delay [expr 2 + $_sfall_delay - ($_azrise_delay - $_arise_delay)] \
  -rise \
  -from [get_pins dpos_reg/CP] \
  -to [get_pins dout_mux/A]
```

Note that we use `_sfall_delay` because it is the falling edge of the select pin that activates the path from A.

That constrained the rising edge at pin A, now we do the same thing for the falling edge:

```
set_max_delay [expr 2 + $_sfall_delay - ($_azfall_delay - $_afall_delay)] \
  -fall \
  -from [get_pins dpos_reg/CP] \
  -to [get_pins dout_mux/A]
```

The B pin gets similar treatment, except that the max_delay constraint uses _srise_delay:

```
set _path [get_timing_paths -rise_through dout_mux/B]
foreach_in_collection point [get_attribute $_path points] {
  set object [get_attribute $point object]
  set point_name [get_attribute $object full_name]
  if {$point_name == "dout_mux/B"} {
    set _brise_delay [get_attribute $point arrival]
  }
  if {$point_name == "dout_mux/Z"} {
    set _bzrise_delay [get_attribute $point arrival]
  }
}
echo "_brise_delay is [format %2.2f $_brise_delay]"
echo "_bzrise_delay is [format %2.2f $_bzrise_delay]"

set _path [get_timing_paths -fall_through dout_mux/B]
foreach_in_collection point [get_attribute $_path points] {
  set object [get_attribute $point object]
  set point_name [get_attribute $object full_name]
  if {$point_name == "dout_mux/B"} {
    set _bfall_delay [get_attribute $point arrival]
  }
  if {$point_name == "dout_mux/Z"} {
    set _bzfall_delay [get_attribute $point arrival]
  }
}
echo "_bfall_delay is [format %2.2f $_bfall_delay]"
echo "_bzfall_delay is [format %2.2f $_bzfall_delay]"

set_max_delay [expr 2 + $_srise_delay - ($_bzfall_delay - $_bfall_delay)] \
  -fall \
  -from [get_pins dneg_reg/CPN] \
  -to [get_pins dout_mux/B]

set_max_delay [expr 2 + $_srise_delay - ($_bzrise_delay - $_brise_delay)] \
  -rise \
  -from [get_pins dneg_reg/CPN] \
  -to [get_pins dout_mux/B]
```

We use _srise_delay because it is the rising edge of the select pin that activates the path from B.

Our dout timing report hasn't changed, but if we now report the timing from one of the flops, we get:

```
pt_shell> report_timing -from [get_pins dpos_reg/Q]
```

```
Startpoint: dpos_reg (rising edge-triggered flip-flop clocked by clk1x)
Endpoint: dout_mux/A (internal path endpoint)
Path Group: **default**
Path Type: max
```

Point	Incr	Path

clock network delay (propagated)	0.14	0.14
dpos_reg/CP (FD1QA)	0.00	0.14 r
dpos_reg/Q (FD1QA) <-	0.34	0.48 f
dout_mux/A (MUX21HA)	0.00	0.48 f
data arrival time		0.48
max_delay	2.23	2.23
output external delay	0.00	2.23
data required time		2.23

data required time		2.23
data arrival time		-0.48

slack (MET)		1.75

There's our max_delay requirement, and it is indeed met.

How about min delays (hold)?

If we do a `report_timing -delay min`, we now get:

```
pt_shell> report_timing -to [get_ports dout] -delay min
```

```
Startpoint: clk1x (clock source 'clk1x')
Endpoint: dout (output port clocked by clkout)
Path Group: clkout
Path Type: min
```

Point	Incr	Path

clock clk1x (fall edge)	2.00	2.00
clk1x (in)	0.00	2.00 f
clktree/Z (BUFC)	0.18	2.18 f
dout_mux/Z (MUX21HA)	0.21	2.39 f
dout (out)	0.00	2.39 f
data arrival time		2.39
clock clkout (fall edge)	2.00	2.00
clock network delay (ideal)	0.39	2.39
output external delay	0.30	2.69
data required time		2.69

data required time		2.69
data arrival time		-2.39

slack (VIOLATED)		-0.30

This is correct.

What about the min timing from the flops?

```
pt_shell> report_timing -delay min -from [get_pins dneg_reg/Q]
*****
Report : timing
        -path full
        -delay min
        -max_paths 1
Design : ddrou1x
Version: 2000.11
Date   : Fri Feb 22 14:46:11 2002
*****
```

```
Startpoint: dneg_reg (falling edge-triggered flip-flop clocked by clk1x)
Endpoint: dout_mux/B (internal path endpoint)
Path Group: (none)
Path Type: min
```

Point	Incr	Path
clock network delay (propagated)	0.18	0.18
dneg_reg/CPN (FDN1QA)	0.00	0.18 f
dneg_reg/Q (FDN1QA) <-	0.29	0.47 f
dout_mux/B (MUX21HA)	0.00	0.47 f
data arrival time		0.47

(Path is unconstrained)

Oops. Our previously-correct hold check is now broken. This is because the `set_max_delay` created timing endpoints at the A and B pins. We can't get around this by changing the pin as we did before, so we'll have to do a similar `set_min_delay` at the A and B pins.

This `set_min_delay` will, like the `set_max_delay`, just verify what we have been calling the "latch assumption" – i.e. it will verify that the critical path is through the S pin of the mux.

Similar to the `set_max_delay`, what we are trying to constrain is that the delay through the flop path is SLOWER than the delay through the data path. The code looks like this:

```
set_min_delay [expr $_sfall_delay - ($_azrise_delay - $_arise_delay)] \
  -rise \
  -from [get_pins dpos_reg/CP] \
  -to [get_pins dout_mux/A]

set_min_delay [expr $_sfall_delay - ($_azfall_delay - $_afall_delay)] \
  -fall \
  -from [get_pins dpos_reg/CP] \
  -to [get_pins dout_mux/A]

set_min_delay [expr $_srise_delay - ($_bzfall_delay - $_bfall_delay)] \
  -fall \
  -from [get_pins dneg_reg/CPN] \
  -to [get_pins dout_mux/B]

set_min_delay [expr $_srise_delay - ($_bzrise_delay - $_brise_delay)] \
  -rise \
  -from [get_pins dneg_reg/CPN] \
  -to [get_pins dout_mux/B]
```

And here's our new min report:

```
pt_shell> report_timing -delay min -from [get_pins dneg_reg/Q]

Startpoint: dneg_reg (falling edge-triggered flip-flop clocked by clk1x)
Endpoint: dout_mux/B (internal path endpoint)
Path Group: **default**
Path Type: min
```

Point	Incr	Path

clock network delay (propagated)	0.18	0.18
dneg_reg/CPN (FDN1QA)	0.00	0.18 f
dneg_reg/Q (FDN1QA) <-	0.29	0.47 f
dout_mux/B (MUX21HA)	0.00	0.47 f
data arrival time		0.47
min_delay	0.22	0.22
output external delay	0.00	0.22
data required time		0.22

data required time		0.22
data arrival time		-0.47

slack (MET)		0.25

Remember that we're only checking the latch assumption here. Recall the flop to dout min path had -0.4ns of slack, while the clock to dout path had -0.30ns of slack. So, we satisfied the latch assumption by about 0.26ns, which matches what we see above (minus rounding errors).

OK, this "latch" approach works, but it is not entirely satisfying, because it over-constrains the design. It is possible for the flop path to exceed the limit above, and have the circuit still meet

timing. It isn't likely, but it's possible. Also, the whole thing is a mess to code! Is there perhaps another way?

5.2 Virtual Clock Approach

Well, there is another way – we just didn't find it until after tape-out.

If you recall, the original problem was that we had paths from the `dpos_reg` to the falling clock edge and from the `dneg_reg` to the rising clock edge that were false, but we couldn't use `set_false_path` or `set_multicycle_path` to get rid of them because they were edge-specific.

One way to get around this would be to have TWO external clocks, one for the rising edge checks and one for the falling edge checks. We could then use `set_false_path` from the `dpos_reg` to the neg clock and from the `dneg_reg` to the pos clock.

Unfortunately, current rev's of PrimeTime won't allow 2 clocks on the same pin, so we can't use `create_generated_clock` to do this. We can, however, create virtual clocks. We'll have to calculate the timing of these clocks ourselves using `get_timing_paths` rather than letting the `create_generated_clock` command do it for us, but it works.

First, we create the `clk1x` input clock as usual:

```
create_clock -period $_period -name clk1x \  
  [get_ports clk1x]  
set_propagated_clock clk1x
```

Now, we time the clock paths:

```
set _path [get_timing_paths \  
  -from [get_ports clk1x] -to [get_ports clkout] -delay max_rise]  
set _rise_latency [get_attribute $_path arrival]  
echo "_rise_latency is [format %2.2f $_rise_latency]"  
  
set _path [get_timing_paths \  
  -from [get_ports clk1x] -to [get_ports clkout] -delay max_fall]  
set _fall_latency [get_attribute $_path arrival]  
echo "_fall_latency is [format %2.2f $_fall_latency]"
```

The result is:

```
pt_shell> source temp.pt  
_rise_latency is 0.31  
_fall_latency is 0.39
```

Notice that these values (rounded to 2 decimal places) match the values shown in previous timing traces for the `clkout` "clock network delay (ideal)". So, we've got the right values.

Now we can create the virtual clocks. There are several ways to do this. You could create the "pos" clock with a waveform of {0.31 2.39} and the "neg" clock with {2.39 4.31} and make all

the timing relative to rising edges. You could do the same thing with a standard {0 2} waveform and use source latency to shift the timing. Or you could create two nearly identical clocks, using either waveform or source latency, and reference dout to the rising edge of the “pos” clock and the falling edge of the “neg” clock.

We chose to create two nearly identical clocks with standard waveforms, and shift their timing using source latency. Then, we will reference the data to the appropriate edge of each clock:

```
create_clock -period $_period -waveform {0 2.0} -name posclkout
set_clock_latency $_rise_latency -rise -source [get_clocks posclkout]
set_clock_latency $_fall_latency -fall -source [get_clocks posclkout]

create_clock -period $_period -waveform {0 2.0} -name negclkout
set_clock_latency $_rise_latency -rise -source [get_clocks negclkout]
set_clock_latency $_fall_latency -fall -source [get_clocks negclkout]
```

Now, we can set the output delay using these clocks. The output delays for negclkout will, of course, use the `-clock_fall` switch:

```
set_output_delay 0.5 -clock posclkout [get_ports dout] -max
set_output_delay [expr -1 * 0.3] -clock posclkout [get_ports dout] -min \
-add_delay

set_output_delay 0.5 -clock negclkout [get_ports dout] -max -add_delay -
clock_fall
set_output_delay [expr -1 * 0.3] -clock negclkout [get_ports dout] -min \
-add_delay -clock_fall
```

Now if we report the timing from one of the flops, we get:

```
pt_shell> report_timing -from [get_pins dpos_reg/Q] -to dout -path_type
full_clock
```

```
Startpoint: dpos_reg (rising edge-triggered flip-flop clocked by clk1x)
Endpoint: dout (output port clocked by posclkout)
Path Group: posclkout
Path Type: max
```

Point	Incr	Path

clock clk1x (rise edge)	0.00	0.00
clock source latency	0.00	0.00
clk1x (in)	0.00	0.00 r
clktree/Z (BUFC)	0.14	0.14 r
dpos_reg/CP (FD1QA)	0.00	0.14 r
dpos_reg/CP (FD1QA)	0.00	0.14 r
dpos_reg/Q (FD1QA) <-	0.34	0.48 f
dout_mux/Z (MUX21HA)	0.18	0.66 f
dout (out)	0.00	0.66 f
data arrival time		0.66
clock posclkout (rise edge)	4.00	4.00
clock network delay (ideal)	0.31	4.31
output external delay	-0.50	3.81
data required time		3.81

data required time		3.81
data arrival time		-0.66

slack (MET)		3.14

This is correct.

Now check the clk1x to dout paths:

```
pt_shell> report_timing -from [get_ports clk1x] -to [get_ports dout]
```

```
Startpoint: clk1x (clock source 'clk1x')
Endpoint: dout (output port clocked by negclkout)
Path Group: negclkout
Path Type: max
```

Point	Incr	Path
clock clk1x (rise edge)	0.00	0.00
clk1x (in)	0.00	0.00 r
clktree/Z (BUFC)	0.14	0.14 r
dout_mux/Z (MUX21HA)	0.25	0.40 f
dout (out)	0.00	0.40 f
data arrival time		0.40
clock negclkout (fall edge)	2.00	2.00
clock network delay (ideal)	0.39	2.39
output external delay	-0.50	1.89
data required time		1.89
data required time		1.89
data arrival time		-0.40
slack (MET)		1.49

```
Startpoint: clk1x (clock source 'clk1x')
Endpoint: dout (output port clocked by posclkout)
Path Group: posclkout
Path Type: max
```

Point	Incr	Path
clock clk1x (fall edge)	2.00	2.00
clk1x (in)	0.00	2.00 f
clktree/Z (BUFC)	0.18	2.18 f
dout_mux/Z (MUX21HA)	0.23	2.41 r
dout (out)	0.00	2.41 r
data arrival time		2.41
clock posclkout (rise edge)	4.00	4.00
clock network delay (ideal)	0.31	4.31
output external delay	-0.50	3.81
data required time		3.81
data required time		3.81
data arrival time		-2.41
slack (MET)		1.40

This is also correct. Although the constraints are now to the two virtual clocks, the slacks are the same as we found earlier. We have correctly constrained the circuit for max timing using the 2 virtual clocks. What about the min paths?

The min (hold) report looks like this:

```
pt_shell> report_timing -to [get_ports dout] -delay min
```

```
Startpoint: clk1x (clock source 'clk1x')
Endpoint: dout (output port clocked by negclkout)
Path Group: negclkout
Path Type: min
```

Point	Incr	Path

clock clk1x (fall edge)	2.00	2.00
clk1x (in)	0.00	2.00 f
clktree/Z (BUFC)	0.18	2.18 f
dout_mux/Z (MUX21HA)	0.21	2.39 f
dout (out)	0.00	2.39 f
data arrival time		2.39
clock negclkout (fall edge)	2.00	2.00
clock network delay (ideal)	0.39	2.39
output external delay	0.30	2.69
data required time		2.69

data required time		2.69
data arrival time		-2.39

slack (VIOLATED)		-0.30

```
Startpoint: clk1x (clock source 'clk1x')
Endpoint: dout (output port clocked by posclkout)
Path Group: posclkout
Path Type: min
```

Point	Incr	Path

clock clk1x (rise edge)	0.00	0.00
clk1x (in)	0.00	0.00 r
clktree/Z (BUFC)	0.14	0.14 r
dout_mux/Z (MUX21HA)	0.16	0.31 r
dout (out)	0.00	0.31 r
data arrival time		0.31
clock posclkout (rise edge)	0.00	0.00
clock network delay (ideal)	0.31	0.31
output external delay	0.30	0.61
data required time		0.61

data required time		0.61
data arrival time		-0.31

slack (VIOLATED)		-0.30

Again, this matches what we saw before.

This time, however, the min path from the flops is going to show up as well:

```
pt_shell> report_timing -from [get_pins dneg_reg/Q] -to dout -delay min
```

```
Startpoint: dneg_reg (falling edge-triggered flip-flop clocked by clk1x)  
Endpoint: dout (output port clocked by negclkout)  
Path Group: negclkout  
Path Type: min
```

Point	Incr	Path

clock clk1x (fall edge)	2.00	2.00
clock network delay (propagated)	0.18	2.18
dneg_reg/CPN (FDN1QA)	0.00	2.18 f
dneg_reg/Q (FDN1QA) <-	0.29	2.47 f
dout_mux/Z (MUX21HA)	0.18	2.65 f
dout (out)	0.00	2.65 f
data arrival time		2.65
clock negclkout (fall edge)	2.00	2.00
clock network delay (ideal)	0.39	2.39
output external delay	0.30	2.69
data required time		2.69

data required time		2.69
data arrival time		-2.65

slack (VIOLATED)		-0.04

This is what we saw earlier, and it is correct! Yeah!

6 Conclusions and Recommendations

We have shown how PrimeTime can be used to analyze DDR circuits on both the input and output sides. In the case of the input, the technique is fairly straightforward. The output cases, however, can be somewhat more complex and depend on the type of circuit implementation.

The technique for timing the output circuit which uses a 2x clock, while a bit more involved, is at least fairly straightforward to understand and implement.

The techniques described for timing the output circuit which uses a 1x clock are much messier. What we have called the “latch approach” is by far the messier of the two, and it results in an over-constrained design. Thus, we expect to use the virtual clock approach on future designs (we developed this technique after tape-out, unfortunately).

Along the way, we have touched on a number of techniques that we hope will find wider application beyond the DDR circuit, as well as pointing out issues commonly encountered by PrimeTime users:

- Proper use of `set_output_delay` for the hold constraint.
- Various uses for `get_timing_paths`, including both specific locations along a path and specific edges.
- Pointed out the often confusing differences between `report_timing` and `get_timing_paths`.
- Use of `set_output_delay` to create a timing endpoint where there was none previously.
- Use of `set_multicycle_path` to effectively get rid of unwanted paths.
- Splitting a real clock into 2 virtual clocks in order to constrain differently to the rising and falling edges.

We hope that you have learned something useful from this discussion.

7 Acknowledgements

The authors would like to thank the following people for their careful review and thoughtful input:

Stephan Scharfenberg of Motorola GmbH

Steve Cochran of Cisco Systems

8 References

[1] P. Zimmer, “Complex Clocking Situations Using PrimeTime”, SNUG 2001