

Getting DDRs “write” – the 1x output circuit revisited

Paul Zimmer

Zimmer Design Services
1375 Sun Tree Drive
Roseville, CA 95661

paulzimmer@zimmerdesignservices.com

website: www.zimmerdesignservices.com

ABSTRACT

DDR SDRAM interfaces are a difficult challenge in STA. Over the years, both constraint-based and model-based techniques have been proposed in various SNUG papers. The classic 1x clock DDR write circuit will be examined and timed using both constraint-based and model-based approaches, as well as a new twist on the model-based approach developed by the author. The various advantages and disadvantages of each technique will be discussed.

Table of contents

1	Introduction.....	3
2	The basic 1x clock output circuit.....	4
2.1	Timing waveforms.....	5
3	Timing this circuit using constraints	6
3.1	Creating the clocks.....	6
3.2	Creating the output delays	7
3.3	Setting the false and multicycle paths	8
3.4	Dealing with the write dqs blocking gate	22
4	Timing this circuit using stamp models.....	33
4.1	The stamp model and hooking it in.....	34
4.2	The script itself.....	36
4.3	Timing results	37
4.4	Comparison of constraint versus stamp model	40
5	Timing this circuit using stamp models and create_cell	41
5.1	Hooking in the stamp model using create_cell	41
5.2	The script itself.....	42
5.3	Timing traces	43
6	Conclusion.....	46
7	Acknowledgements.....	47
8	References	48
9	Appendix	49
9.1	Code for the two-clock approach.....	49

1 Introduction

At SNUG San Jose 2002, I co-authored a paper with Andrew Cheng called, “Working with DDR’s in Primetime”. In that paper, we addressed the problem of timing the typical 1x output circuit for a DDR interface. The paper needs to be updated for several reasons.

First, that paper was aimed at addressing the issues associated with Double Data Rate interfaces in general. But most of the questions arise in the details of timing DDR SDRAMs. Also, we didn’t address the issue of how the clock might be delayed or what affect this would have on STA.

Second, the technique we recommended at that time, using a pair of virtual clocks, is outdated. PrimeTime has added multiclock propagation, which simplifies the technique, but introduces new considerations.

Third, the Primetime behavior that required creating two output clocks has been changed, and the job can now be done with a single clock (although the two clock technique still works and may be preferred by some users).

Finally, a number of follow-up papers have suggested an entirely different approach using external models. This approach has its merits, but it also has some pitfalls. I wish to explore this approach in more detail and compare it to the (updated) constraint approach. Then I want to propose a slightly modified model approach that I think has some distinct advantages.

All in all, it’s time for an update on handling the DDR SDRAM 1x output circuit.

2 The basic 1x clock output circuit

A typical 1x DDR output circuit for an SDRAM might look like this:

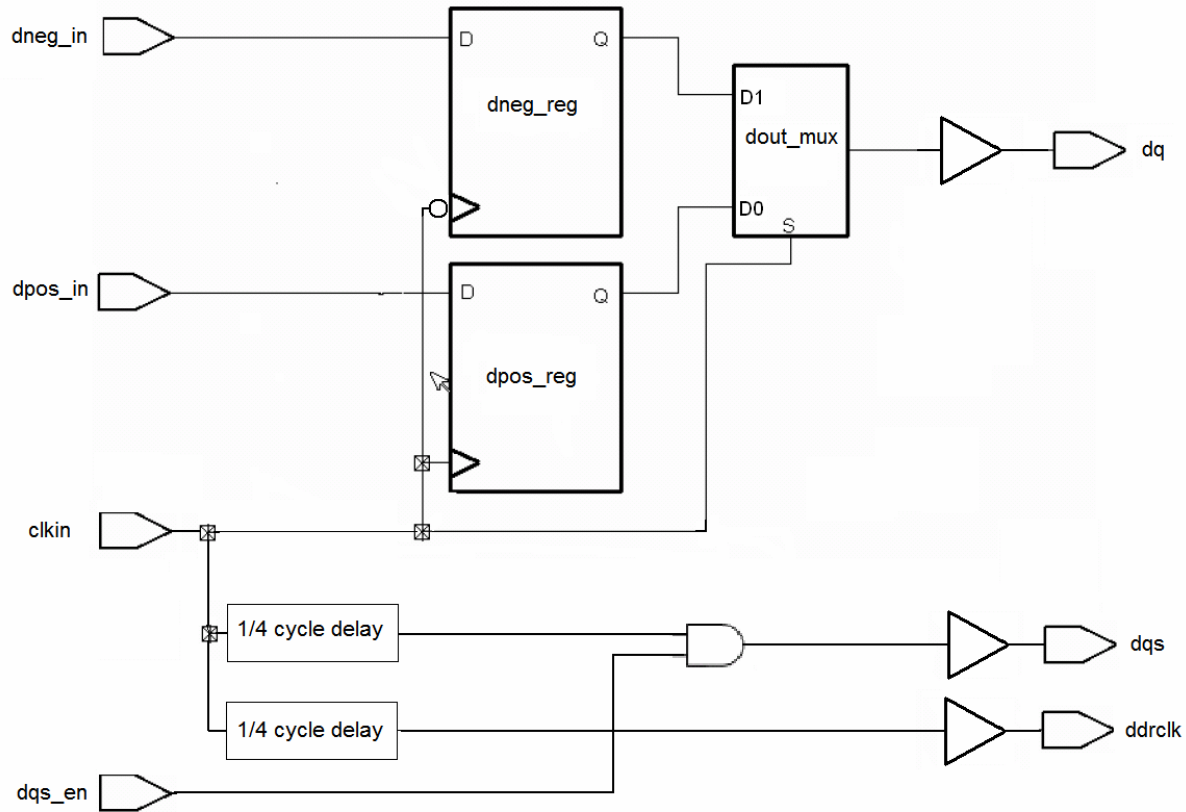


Figure 2-1

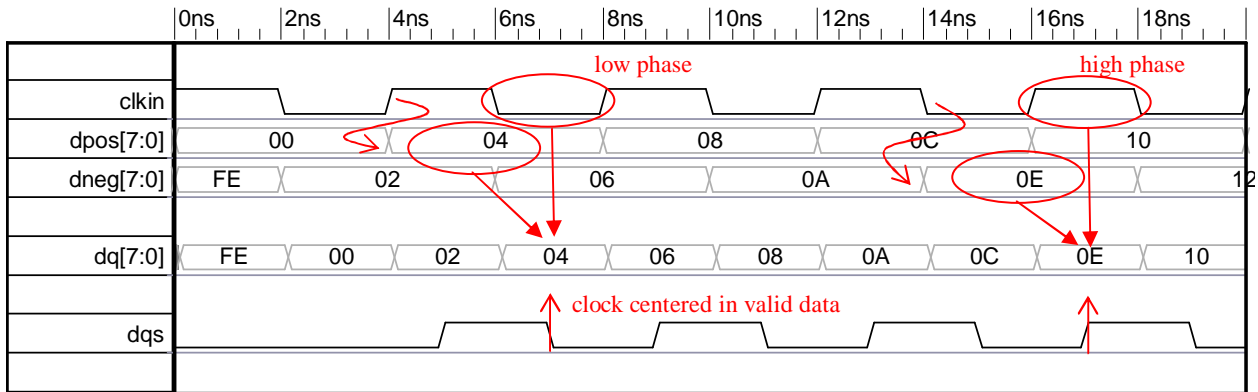
The $\frac{1}{4}$ cycle delays on the `dqs` and `ddrclk` allow the `dqs` to be centered in the valid data window, while keeping `dqs` in phase with `ddrclk` (as required by the DDR spec). The AND gate is used to enable `dqs`, and will be looked at in more detail later.

This is by no means the only way to implement the clock delays. You could do the $\frac{1}{4}$ cycle delays on the board. I have also seen implementations that delay the data by $\frac{3}{4}$ cycle and the `dqs` by 1 cycle, and leave the `ddrclk` unchanged. This has the same net effect.

I will use the circuit above for this paper. Techniques for timing the other circuits are similar.

2.1 Timing waveforms

Here's what the timing waveforms look like for this circuit:



Waveform created by Synapticad Waveformer Pro (www.syncad.com)

Figure 2-2

The rising clk_{in} loads dpos_{reg} and the falling clk_{in} loads dneg_{reg}. To allow maximum setup time, the contents of dneg_{reg} are output by the mux on the high phase of the clock, and the contents of dpos_{reg} are output by the mux on the low phase of the clock.

3 Timing this circuit using constraints

So, how do we time this interface in Primetime?

3.1 Creating the clocks

The first thing to notice is that the outputs need to be timed relative to both edges of the dqs output clock. Although it is possible to create this constraint with `set_output_delay`, Reference (1) showed that it was not possible to create the necessary false paths with a single output clock.

But someone at Synopsys must have been listening! I have been using the two-clock approach for years, but in preparing this paper I went back over the example shown in Reference (1), and discovered that the “`set_false_path -rise_to [get_clocks ...]`” syntax now does what you would want it to do – it sets a false path to a clock edge.

So, I believe it is now possible to time this circuit using only a single output clock. The two-clock approach still works, and some may prefer it. The code is given in appendix 1.

So, now the clock creation code is quite simple. We just create the input clock and the dqs output (generated) clock:

```
set _period 4.0
create_clock -period $_period -name clkin \
  [get_ports clkin]

create_generated_clock \
  -name dqsoutclk \
  -source [get_attribute [get_clocks clkin] sources] \
  -divide_by 1 \
  -master_clock clkin \
  -add \
  [get_ports dqs]

set_propagated_clock [all_clocks]
```

Notice that I created the clock directly on the dqs port. Many people create the clock on the core side of the pad instead. Often this is done to avoid the “loopback” of the write clock into the read circuitry. But this complicates the scripts considerably, and is not really necessary.

Instead, create the read clock(s) on the dqs port as well, but then create a divide-by 1 generated clock(s) on the core side of the dqs pad (the one that drives toward the core) with the master being the read clock(s). This will block the write clock from looping back.

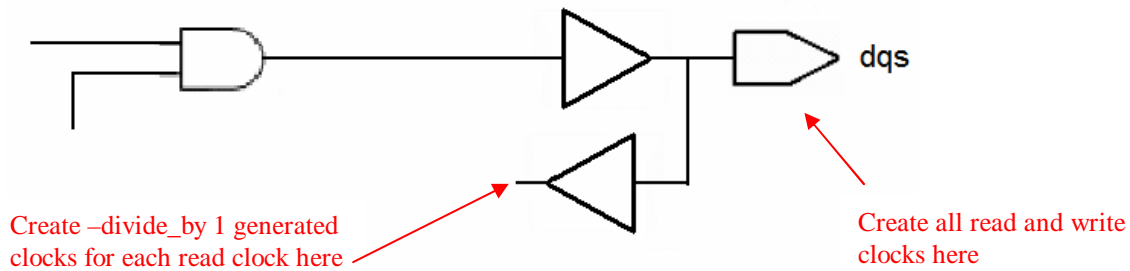


Figure 3-1

3.2 Creating the output delays

Now we set the output delays using the `dqsoutclk`. The value for max is the DDR spec `tDS` and the value for min is the DDR spec `-tDH`. So, ignoring the board delays for a moment, our `set_output_delay` statements (for the rising edge) would look like this:

```
set _tDS 0.5
set _tDH 0.5

set_output_delay -max \
  -clock dqsoutclk \
  $_tDS \
  [get_ports dq]

set_output_delay -min \
  -add_delay \
  -clock dqsoutclk \
  [expr (-1 * $_tDH)] \
  [get_ports dq]
```

(If you don't understand the reason for the "-1 *", take a look at section 2.2 of Reference (1))

But these specs are defined at the DDR pins. We want to account for wire delays as well, so we need to adjust the output delay values accordingly.

Here are the values we will use. They are unrealistically large, but that makes it easier to see their effect:

```
set _dq_wire_delay_max 1.0
set _dq_wire_delay_min 0.9
set _dqs_wire_delay_max 1.2
set _dqs_wire_delay_min 1.08
```

The `set_output_delay` value really refers to “the amount of external delay of data relative to clock”. Thus, we add in the data delay (dq) and subtract out the clock (dqs) delay. We want the max value, so we use the max data delay (`$_dq_wire_delay_max`) minus the min clock delay (`$_dqs_wire_delay_min`).

```
set_output_delay -max \  
-clock dqsoutclk \  
[expr $_tDS + $_dq_wire_delay_max - $_dqs_wire_delay_min] \  
[get_ports dq]
```

Similarly, the min output delay will be the min data delay (`$_dq_wire_delay_min`) minus the max clock delay (`$_dqs_wire_delay_max`).

```
set_output_delay -min \  
-add_delay \  
-clock dqsoutclk \  
[expr (-1 * $_tDH) + $_dq_wire_delay_min - $_dqs_wire_delay_max] \  
[get_ports dq]
```

Now we can do the same thing for the falling edge constraints. We use `-clock_fall` to tell PT to constrain to the falling edge of the `dqsoutclk`:

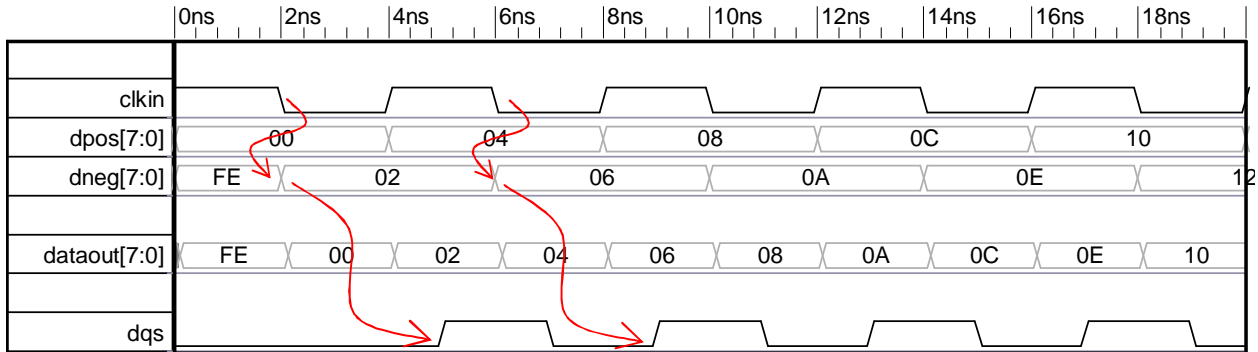
```
set_output_delay -max \  
-add_delay -clock_fall \  
[expr $_tDS + $_dq_wire_delay_max - $_dqs_wire_delay_min] \  
-clock dqsoutclk \  
[get_ports dq]  
set_output_delay -min \  
-add_delay -clock_fall \  
[expr (-1 * $_tDH) + $_dq_wire_delay_min - $_dqs_wire_delay_max] \  
-clock dqsoutclk \  
[get_ports dq]
```

3.3 Setting the false and multicycle paths

That’s fine as far as it goes, but there are several false and multicycle paths we need to take care of as well.

3.3.1 Paths from the pos/neg reg flops

Let's take another look at that timing diagram:



Waveform created by Synapticad Waveformer Pro (www.syncad.com)

Figure 3-2

I have set this diagram up so that the data value refers to the time at which the edge of clk transitioned that caused this data to transition. The first data clocked (“02”) by the rising dqs was generated by the falling edge of clk at time 2.

The thing to notice here is that the dqs rising edge *only* clocks data from dneg, and the dqs falling edge *only* clocks data from dpos.

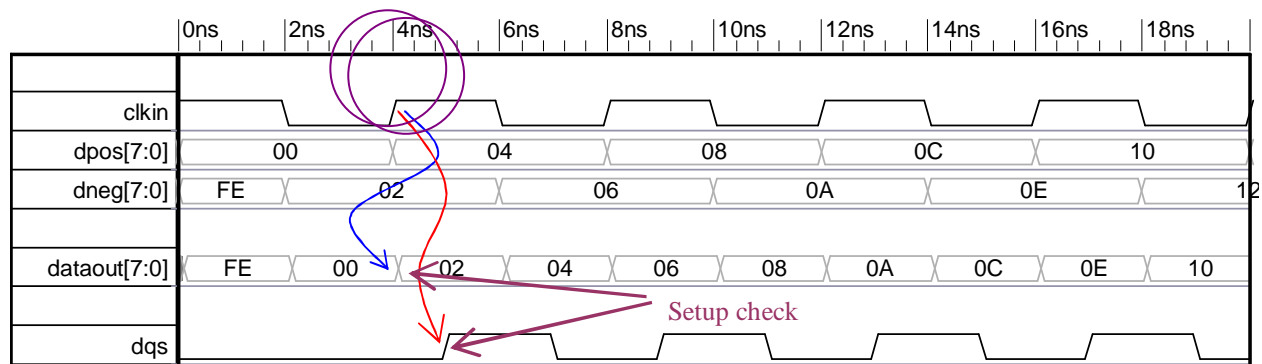
While it is possible for dqs rising to clock data from dpos, this can only happen due to a timing failure in the *mux path*. In other words, there is no way that the timing of dpos_reg can *by itself* cause a timing failure to the rising edge of dqs. If the mux path timing passes, then this path cannot have any timing effect. Therefore, this is a false path, as is the path from dneg to the falling edge of dqs.

```
set_false_path -from [get_pins dpos_reg/CP] -rise_to [get_clocks dqsoutclk]
set_false_path -from [get_pins dneg_reg/CPN] -fall_to [get_clocks dqsoutclk]
```

This is the part that didn't work in older version of PT. I'm not exactly sure where it changed. In PT 2000.11, this did not disable the path correctly. It has worked since at least 2004.06 (which is as far back as I have been able to go).

3.3.2 Paths through the datamux select pin.

Now let's consider the mux paths. This can be a little confusing because it is a data path that is launched by a clock. First, we'll look at the *setup* check of the *rising* edge of clk going through the mux select pin to the rising edge of dqs (posdqsout).



Waveform created by Synapticad Waveformer Pro (www.synacal.com)

Figure 3-3

The data was launched by the rising edge of clk at time 4ns. This edge (at the mux select pin) flips the mux to output the dneg reg value "02" onto the dataout bus, which is then clocked by rising dq.

The capture clock (dq) occurs at time 5ns because of the $\frac{1}{4}$ cycle delay, but is caused by the clk transition at time 4ns.

So, the proper timing trace should launch data at time 4ns and capture it with a clock launched at time 4ns. Since PT will shift everything to the first timing window, the correct PT trace will launch at 0ns and check against 0ns. Let's see what PT gets:

```
pt_shell> report_timing -rise_through [get_pins dout_mux/S] -rise_to
[get_clocks dqsoutclk]
```

```
Startpoint: clkln (clock source 'clkln')
Endpoint: dq (output port clocked by dqsoutclk)
Path Group: dqsoutclk
Path Type: max
```

Point	Incr	Path
clock clkln (rise edge)	0.00	0.00
clock source latency	0.00	0.00
clkln (in)	0.00	0.00 r
dout_mux/S (mx02d0) <-	0.00	0.00 r
dout_mux/Z (mx02d0)	0.23 H	0.24 f
dqpad/Z (bufbd1)	1.72 *	1.96 f
dq (out)	0.00	1.96 f
data arrival time		1.96
clock dqsoutclk (rise edge)	4.00	4.00
clock network delay (propagated)	2.83	6.83
clock reconvergence pessimism	0.00	6.83
output external delay	-0.42	6.41
data required time		6.41
data required time		6.41
data arrival time		-1.96
slack (MET)		4.45

Oops! PT checked 0ns against 4ns. What's wrong?

Well, PT's default assumption for setup checks is that data launched by clock n will be checked against clock $n+1$. However, in this case, the data is captured by the *same clock* edge that launched it. How can this be? It is due to the delays. The clock is delayed by 1/4 cycle. The design intent is to capture data with the same clock that launched it. In PT, this is described as a multicycle path of *ZERO*.

```
set_multicycle_path -setup 0 -rise_through [get_pins dout_mux/S] -rise_to
[get_clocks dqsoutclk]
```

If we apply this mcp, we now get the correct trace.

```
pt_shell> report_timing -rise_through [get_pins dout_mux/S] -rise_to
[get_clocks dqsoutclk]
```

```
Startpoint: clkkin (clock source 'clkkin')
Endpoint: dq (output port clocked by dqsoutclk)
Path Group: dqsoutclk
Path Type: max
```

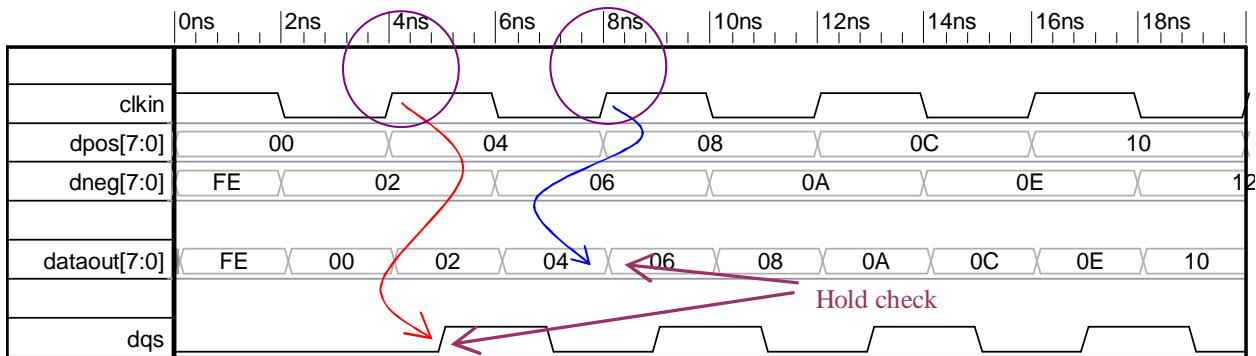
Point	Incr	Path
clock clkkin (rise edge)	0.00	0.00
clock source latency	0.00	0.00
clkkin (in)	0.00	0.00 r
dout_mux/S (mx02d0) <-	0.00	0.00 r
dout_mux/Z (mx02d0)	0.23 H	0.24 f
dqpad/Z (bufbd1)	1.72 *	1.96 f
dq (out)	0.00	1.96 f
data arrival time		1.96

clock dqsoutclk (rise edge)	0.00	0.00
clock network delay (propagated)	2.83	2.83
clock reconvergence pessimism	0.00	2.83
output external delay	-0.42	2.41
data required time		2.41

data required time		2.41
data arrival time		-1.96

slack (MET)		0.45

Now let's check the *hold* timing on this path:



Waveform created by Synapticad Waveformer Pro (www.syncad.com)

Figure 3-4

The next *rising* transition through the mux pin that could cause a hold failure is from the clkkin at time 8ns. So, this check is 8ns against 4ns, or, shifted down, 4ns against 0ns. And that's what PT gets:

```
pt_shell> report_timing -rise_through [get_pins dout_mux/S] -rise_to
[get_clocks dqsoutclk] -delay min
```

```
Startpoint: clkin (clock source 'clkin')
Endpoint: dq (output port clocked by dqsoutclk)
Path Group: dqsoutclk
Path Type: min
```

Point	Incr	Path
clock clkin (rise edge)	4.00	4.00
clock source latency	0.00	4.00
clkin (in)	0.00	4.00 r
dout_mux/S (mx02d0) <-	0.00	4.00 r
dout_mux/Z (mx02d0)	0.22 H	4.22 r
dqpad/Z (bufbd1)	1.60 *	5.82 r
dq (out)	0.00	5.82 r
data arrival time		5.82
clock dqsoutclk (rise edge)	0.00	0.00
clock network delay (propagated)	2.87	2.87
clock reconvergence pessimism	0.00	2.87
output external delay	0.80	3.67
data required time		3.67
data required time		3.67
data arrival time		-5.82
slack (MET)		2.15

You could also make the argument that this is really a false path, since the falling edge through the mux select pin will always have a smaller slack. I've done it both ways.

By the way, I would have expected the default hold check to be wrong, since I shifted the setup check. Normally, when you shift the setup check to n , you have to shift the hold check to $n-1$ if the hold check is not also multicycle. Thus, I had expected this to require a "set_multicycle_path -hold -1". But PT's multicycle path stuff works in mysterious ways when the setup value is 0, and the default value turns out to be correct. Whenever you use a setup multicycle path of 0, or if there are opposite edge checks involved, it is a good idea to double-check that PT is doing the correct edge checks on hold.

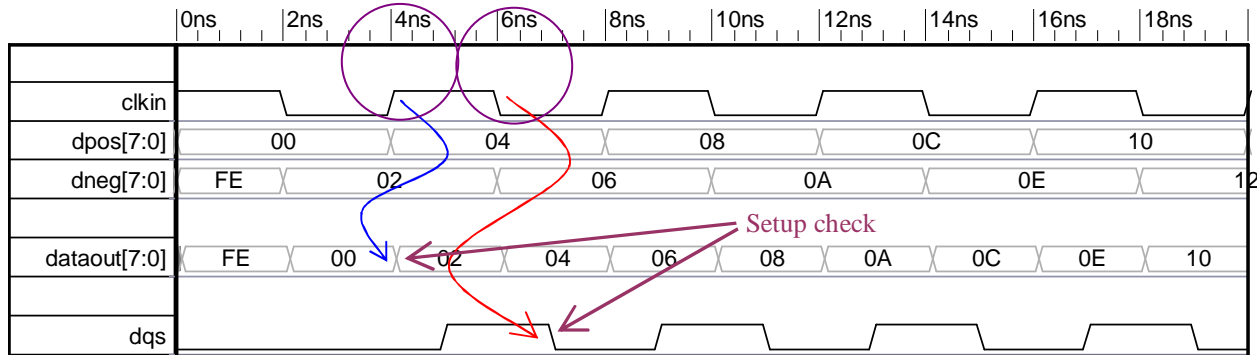
I don't like setting only the setup value, so I set the hold value to 0 just to indicate to myself that I looked at this case:

```
set_multicycle_path -hold 0 -rise_through [get_pins dout_mux/S] -rise_to
[get_clocks dqsoutclk]
```

This is really a no-op, since it tells PT to shift the hold check by 0 clocks from the default. It produces the same traces as above, but it shows that I looked at this path.

Now let's look at the *setup* check of the *rising* edge of clkin going through the mux select pin to the *falling* edge of dq.

This is another one that you could equally well treat as a false path, since the path involving the falling edge through the mux select will always have less slack.



Waveform created by Synapticald Waveformer Pro (www.syn cad.com)

Figure 3-5

Now we have data launched at 4ns against capture clock launched at 6ns. Moving this back to the first cycle, we should get a path of 0ns against 2ns.

Sure enough, that's what we get:

```
pt_shell> report_timing -rise_through [get_pins dout_mux/S] -fall_to
[get_clocks dqsoutclk]
```

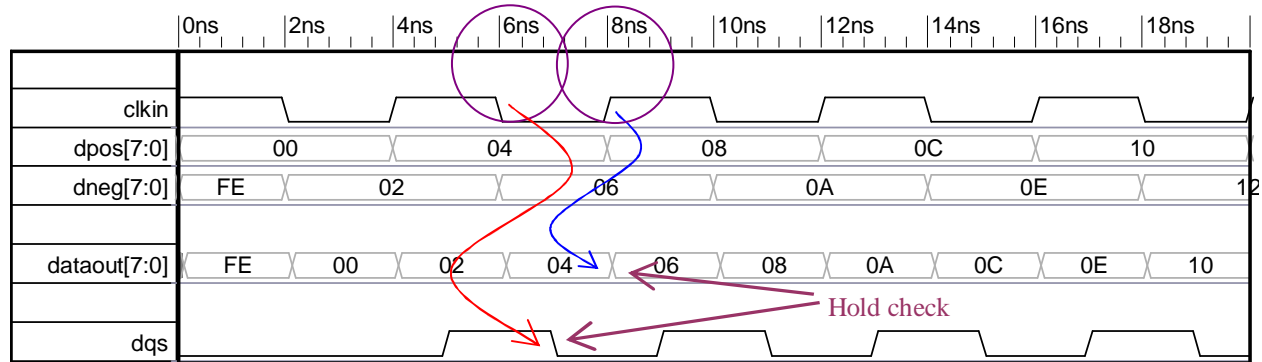
```
Startpoint: clkin (clock source 'clkin')
Endpoint: dq (output port clocked by dqsoutclk)
Path Group: dqsoutclk
Path Type: max
```

Point	Incr	Path
clock clkin (rise edge)	0.00	0.00
clock source latency	0.00	0.00
clkin (in)	0.00	0.00 r
dout_mux/S (mx02d0) <-	0.00	0.00 r
dout_mux/Z (mx02d0)	0.23 H	0.24 f
dqpad/Z (bufbd1)	1.72 *	1.96 f
dq (out)	0.00	1.96 f
data arrival time		1.96
clock dqsoutclk (fall edge)	2.00	2.00
clock network delay (propagated)	2.92	4.92
clock reconvergence pessimism	0.00	4.92
output external delay	-0.42	4.50
data required time		4.50
data required time		4.50
data arrival time		-1.96
slack (MET)		2.55

But, just to indicate it has been checked, we'll set the proper mcp:

```
set_multicycle_path -setup 1 -rise_through [get_pins dout_mux/S] -fall_to  
[get_clocks dqsoutclk]
```

How about hold on this path?



Waveform created by Synapticad Waveformer Pro (www.syncad.com)

Figure 3-6

The expected check here is 8ns against 6ns, or 2ns against 0ns. That's what we get, except that for some reason PT has chosen to do 4ns against 2ns:

```
pt_shell> report_timing -rise_through [get_pins dout_mux/S] -fall_to
[get_clocks dqsoutclk] -delay min
```

```
Startpoint: clkin (clock source 'clkin')
Endpoint: dq (output port clocked by dqsoutclk)
Path Group: dqsoutclk
Path Type: min
```

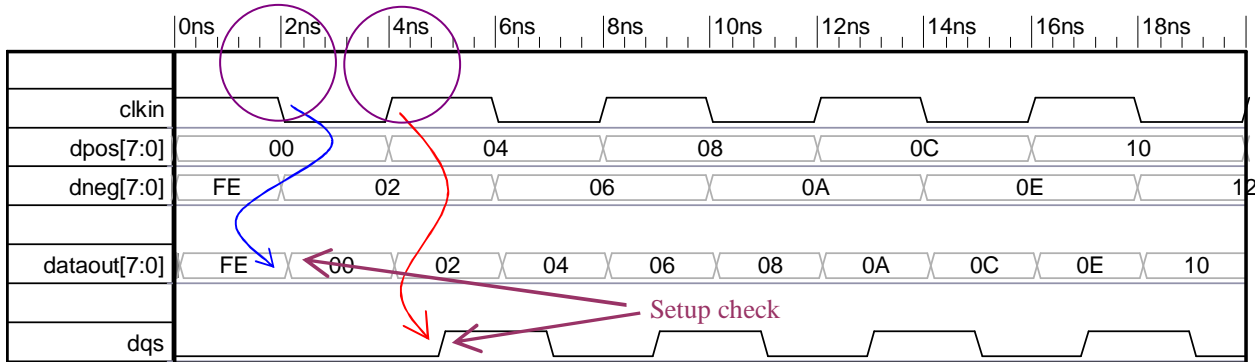
Point	Incr	Path
clock clkin (rise edge)	4.00	4.00
clock source latency	0.00	4.00
clkin (in)	0.00	4.00 r
dout_mux/S (mx02d0) <-	0.00	4.00 r
dout_mux/Z (mx02d0)	0.22 H	4.22 r
dqpad/Z (bufbd1)	1.60 *	5.82 r
dq (out)	0.00	5.82 r
data arrival time		5.82
clock dqsoutclk (fall edge)	2.00	2.00
clock network delay (propagated)	2.96	4.96
clock reconvergence pessimism	0.00	4.96
output external delay	0.80	5.76
data required time		5.76
data required time		5.76
data arrival time		-5.82
slack (MET)		0.06

This doesn't matter – the net check is 2ns. Again, we can do the mcp explicitly to be sure if we want to:

```
set_multicycle_path -hold 0 -rise_through [get_pins dout_mux/S] -fall_to
[get_clocks dqsoutclk]
```

That completes the checks for the paths involving a rising path through the mux select. Now let's tackle the falling edge.

First, setup of the *falling* edge through the mux select to the *rising* dqs.



Waveform created by Synplicity Waveformer Pro (www.synplicity.com)

Figure 3-7

This is another one that you could argue is a false path. But if you're going to have a timing check, it must be 2ns against 4ns (or some equivalent). That is indeed what PT gets:

```
pt_shell> report_timing -fall_through [get_pins dout_mux/S] -rise_to
[get_clocks dqsoutclk]
```

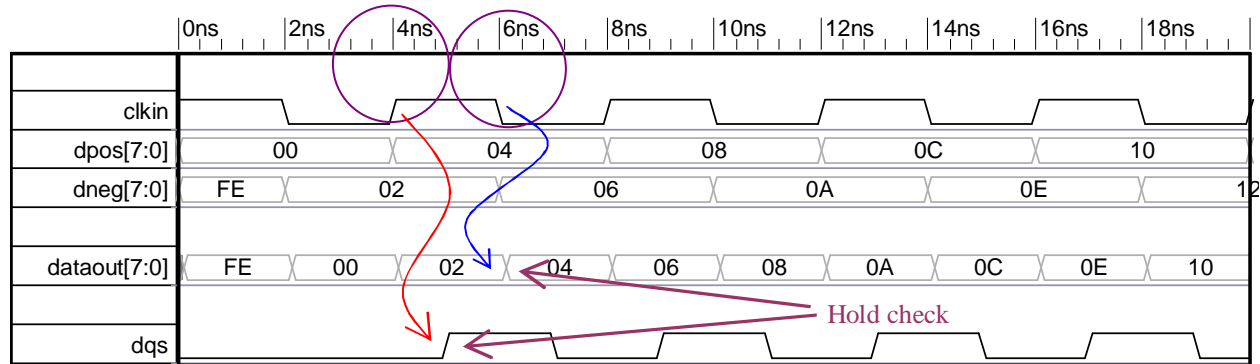
```
Startpoint: clk (clock source 'clk')
Endpoint: dq (output port clocked by dqsoutclk)
Path Group: dqsoutclk
Path Type: max
```

Point	Incr	Path
clock clk (fall edge)	2.00	2.00
clock source latency	0.00	2.00
clk (in)	0.00	2.00 f
dout_mux/S (mx02d0) <-	0.00	2.00 f
dout_mux/Z (mx02d0)	0.23 H	2.24 f
dqpad/Z (bufbd1)	1.72 *	3.96 f
dq (out)	0.00	3.96 f
data arrival time		3.96
clock dqsoutclk (rise edge)	4.00	4.00
clock network delay (propagated)	2.83	6.83
clock reconvergence pessimism	0.00	6.83
output external delay	-0.42	6.41
data required time		6.41
data required time		6.41
data arrival time		-3.96
slack (MET)		2.45

But we can make sure by doing an explicit mcp:

```
set_multicycle_path -setup 1 -fall_through [get_pins dout_mux/S] -rise_to
[get_clocks dqsoutclk]
```

The hold on this path (falling mux select to rising dqs) is *not* false.



Waveform created by Synapticad Waveformer Pro (www.synacad.com)

Figure 3-8

This must be 6ns against 4ns, or something equivalent.

```
pt_shell> report_timing -fall_through [get_pins dout_mux/S] -rise_to
[get_clocks dqsoutclk] -delay min
```

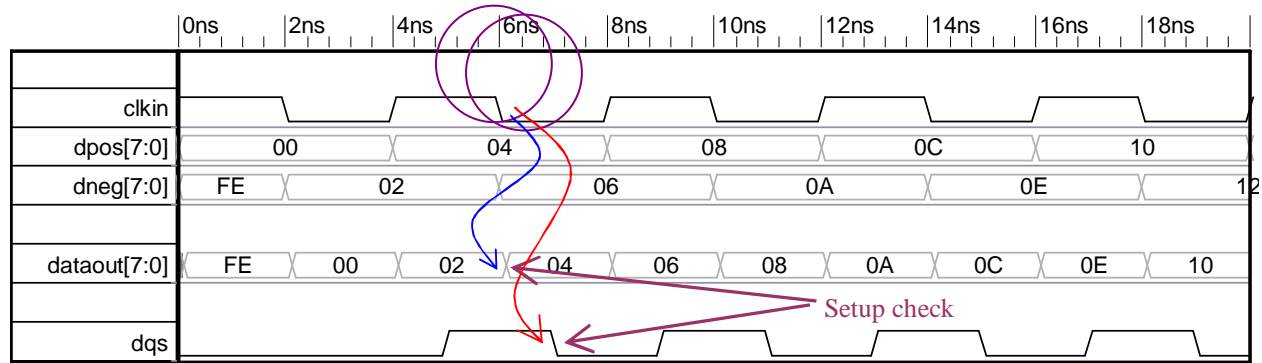
```
Startpoint: clk (clock source 'clk')
Endpoint: dq (output port clocked by dqsoutclk)
Path Group: dqsoutclk
Path Type: min
```

Point	Incr	Path
clock clk (fall edge)	2.00	2.00
clock source latency	0.00	2.00
clk (in)	0.00	2.00 f
dout_mux/S (mx02d0) <-	0.00	2.00 f
dout_mux/Z (mx02d0)	0.22 H	2.22 r
dqpad/Z (bufbd1)	1.60 *	3.82 r
dq (out)	0.00	3.82 r
data arrival time		3.82
clock dqsoutclk (rise edge)	0.00	0.00
clock network delay (propagated)	2.87	2.87
clock reconvergence pessimism	0.00	2.87
output external delay	0.80	3.67
data required time		3.67
data required time		3.67
data arrival time		-3.82
slack (MET)		0.15

So, the default is again correct. We can make it explicit with the following mcp:

```
set_multicycle_path -hold 0 -fall_through [get_pins dout_mux/S] -rise_to  
[get_clocks dqsoutclk]
```

On to setup of the falling mux select to the *negative* dqs.



Waveform created by Synapticad Waveformer Pro (www.syncad.com)

Figure 3-9

This time the default check is not correct. We expect to get 6ns against 6ns (or 2ns against 2ns, or some equivalent), but instead we get:

```
pt_shell> report_timing -fall_through [get_pins dout_mux/S] -fall_to
[get_clocks dqsoutclk]
```

```
Startpoint: clkin (clock source 'clkin')
Endpoint: dq (output port clocked by dqsoutclk)
Path Group: dqsoutclk
Path Type: max
```

Point	Incr	Path
clock clkin (fall edge)	2.00	2.00
clock source latency	0.00	2.00
clkin (in)	0.00	2.00 f
dout_mux/S (mx02d0) <-	0.00	2.00 f
dout_mux/Z (mx02d0)	0.23 H	2.24 f
dqpad/Z (bufbd1)	1.72 *	3.96 f
dq (out)	0.00	3.96 f
data arrival time		3.96
clock dqsoutclk (fall edge)	6.00	6.00
clock network delay (propagated)	2.92	8.92
clock reconvergence pessimism	0.00	8.92
output external delay	-0.42	8.50
data required time		8.50
data required time		8.50
data arrival time		-3.96
slack (MET)		4.55

Again, we need an mcp of zero:

```
set_multicycle_path -setup 0 -fall_through [get_pins dout_mux/S] -fall_to
[get_clocks dqsoutclk]
```

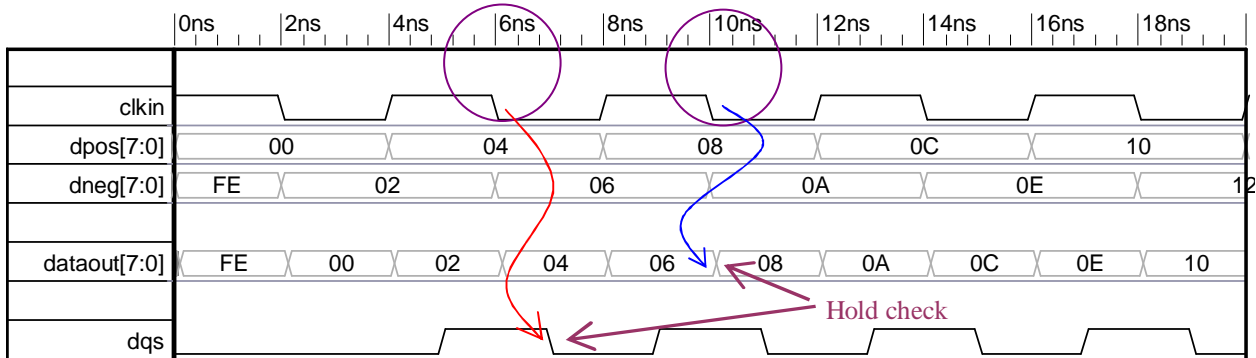
And we now get:

```
pt_shell> report_timing -fall_through [get_pins dout_mux/S] -fall_to
[get_clocks dqsoutclk]
```

```
Startpoint: clkin (clock source 'clkin')
Endpoint: dq (output port clocked by dqsoutclk)
Path Group: dqsoutclk
Path Type: max
```

Point	Incr	Path
clock clkin (fall edge)	2.00	2.00
clock source latency	0.00	2.00
clkin (in)	0.00	2.00 f
dout_mux/S (mx02d0) <-	0.00	2.00 f
dout_mux/Z (mx02d0)	0.23 H	2.24 f
dqpad/Z (bufbd1)	1.72 *	3.96 f
dq (out)	0.00	3.96 f
data arrival time		3.96
clock dqsoutclk (fall edge)	2.00	2.00
clock network delay (propagated)	2.92	4.92
clock reconvergence pessimism	0.00	4.92
output external delay	-0.42	4.50
data required time		4.50
data required time		4.50
data arrival time		-3.96
slack (MET)		0.55

And, finally, how about the hold on this path?



Waveform created by Synaptical Waveformer Pro (www.syn cad.com)

Figure 3-10

This is another one of those that could be considered a false path. But if it is treated as a multicycle path, the check should be 10ns against 6ns (or 4ns against 0, or some equivalent). Once we have done the mcp 0 for the setup above, PT’s default will give the correct trace:

```
pt_shell> report_timing -fall_through [get_pins dout_mux/S] -fall_to
[get_clocks dqsoutclk] -delay min
```

```
Startpoint: clkkin (clock source 'clkkin')
Endpoint: dq (output port clocked by dqsoutclk)
Path Group: dqsoutclk
Path Type: min
```

Point	Incr	Path
clock clkkin (fall edge)	6.00	6.00
clock source latency	0.00	6.00
clkkin (in)	0.00	6.00 f
dout_mux/S (mx02d0) <-	0.00	6.00 f
dout_mux/Z (mx02d0)	0.22 H	6.22 r
dqpad/Z (bufbd1)	1.60 *	7.82 r
dq (out)	0.00	7.82 r
data arrival time		7.82

clock dqsoutclk (fall edge)	2.00	2.00
clock network delay (propagated)	2.96	4.96
clock reconvergence pessimism	0.00	4.96
output external delay	0.80	5.76
data required time		5.76

data required time		5.76
data arrival time		-7.82

slack (MET)		2.06

But I prefer an explicit 0 mcp to indicate I have looked at this path:

```
set_multicycle_path -hold 0 -fall_through [get_pins dout_mux/S] -fall_to
[get_clocks dqsoutclk]
```

So, why have I bothered to go through all of this? The defaults were correct on most of the paths anyway. The reason that the default checks were mostly correct is that most of the paths involved opposite edges. But whenever there are cases of delayed clocks or opposite edges, I believe it is important to review all of the cases and make sure that the correct edges are being checked. One client design I saw didn't use the falling edge for dneg_reg, and most of the defaults checks ended up being wrong. So I recommend always checking these sorts of paths carefully.

3.4 Dealing with the write dqs blocking gate

I included that AND gate for blocking the outgoing dqs, but so far I haven't said much about it. I didn't want to cloud the basic technique by dealing with the complexity introduced by this gate. But since this gate appears in most ddr output circuits, I feel I need to address the issues.

Suppose we add a little more detail to the circuit, and show how the enable might be driven?

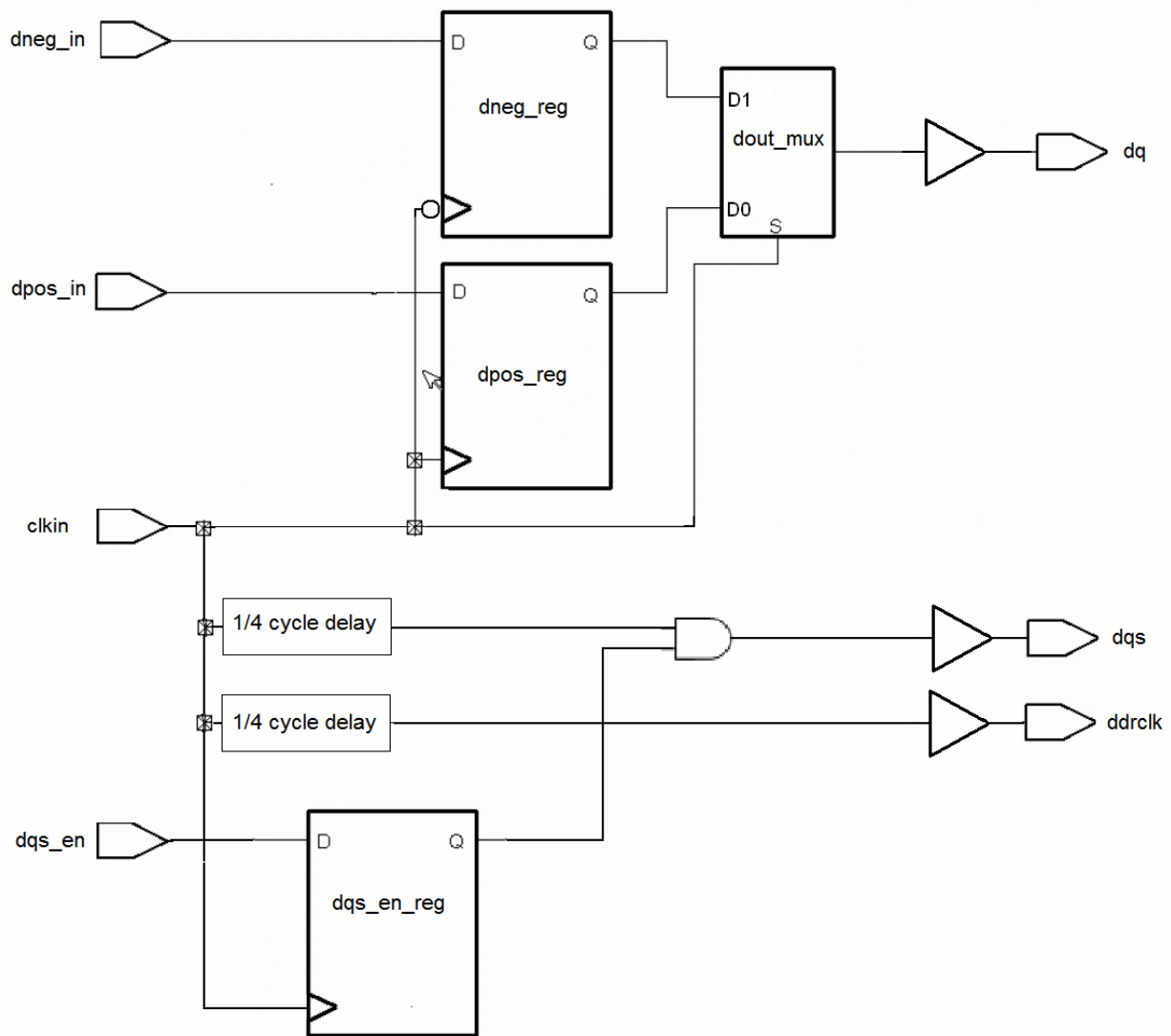


Figure 3-11

I realize that the circuitry that controls the `dqs` blocking gate is often much more complex than this single flop, but the single flop is sufficient to illustrate the problem and the solution.

3.4.1 “Promiscuous” backtracing

If we run the same commands as used earlier, and report that last *setup* path, we get the following trace:

```
pt_shell> report_timing -fall_through [get_pins dout_mux/S] -fall_to
[get_clocks dqsoutclk]
```

```
Startpoint: clkin (clock source 'clkin')
Endpoint: dq (output port clocked by dqsoutclk)
Path Group: dqsoutclk
Path Type: max
```

Point	Incr	Path
clock clkin (fall edge)	2.00	2.00
clock source latency	0.00	2.00
clkin (in)	0.00	2.00 f
dout_mux/S (mx02d0) <-	0.00	2.00 f
dout_mux/Z (mx02d0)	0.23 H	2.24 f
dqpad/Z (bufbd1)	1.72 *	3.96 f
dq (out)	0.00	3.96 f
data arrival time		3.96
clock dqsoutclk (fall edge)	2.00	2.00
clock network delay (propagated)	2.11	4.11
clock reconvergence pessimism	0.00	4.11
output external delay	-0.42	3.69
data required time		3.69
data required time		3.69
data arrival time		-3.96
slack (VIOLATED)		-0.26

But the original trace had 0.55ns of positive slack. What happened?

Let's run the reports again with the `-path_type full_clock_expanded` switch. Here's the trace without the `dqs enable flop`:

```
pt_shell> report_timing -fall_through [get_pins dout_mux/S] -fall_to
[get_clocks dqsoutclk] -path_type full_clock_expanded
```

```
Startpoint: clkin (clock source 'clkin')
Endpoint: dq (output port clocked by dqsoutclk)
Path Group: dqsoutclk
Path Type: max
```

Point	Incr	Path

clock clkin (fall edge)	2.00	2.00
clock source latency	0.00	2.00
clkin (in)	0.00	2.00 f
dout_mux/S (mx02d0) <-	0.00	2.00 f
dout_mux/Z (mx02d0)	0.23 H	2.24 f
dqpad/Z (bufbd1)	1.72 *	3.96 f
dq (out)	0.00	3.96 f
data arrival time		3.96

clock dqsoutclk (fall edge)	2.00	2.00
clock clkin (source latency)	0.00	2.00
clkin (in)	0.00	2.00 f
clkdelaybuf_clk/Z (bufbd1)	1.00 H	3.00 f
dqsand/Z (an02d0)	0.22 *	3.22 f
dqspad/Z (bufbd1)	1.71 *	4.92 f
dqs (out)	0.00	4.92 f
clock network delay (propagated)	0.00	4.92
clock reconvergence pessimism	0.00	4.92
output external delay	-0.42	4.50
data required time		4.50

data required time		4.50
data arrival time		-3.96

slack (MET)		0.55

And here's the new trace with the dqs enable flop:

```
pt_shell> report_timing -fall_through [get_pins dout_mux/S] -fall_to
[get_clocks dqsoutclk] -path_type full_clock_expanded
```

```
Startpoint: clkin (clock source 'clkin')
Endpoint: dq (output port clocked by dqsoutclk)
Path Group: dqsoutclk
Path Type: max
```

Point	Incr	Path

clock clkin (fall edge)	2.00	2.00
clock source latency	0.00	2.00
clkin (in)	0.00	2.00 f
dout_mux/S (mx02d0) <-	0.00	2.00 f
dout_mux/Z (mx02d0)	0.23 H	2.24 f
dqpad/Z (bufbd1)	1.72 *	3.96 f
dq (out)	0.00	3.96 f
data arrival time		3.96

clock dqsoutclk (fall edge)	2.00	2.00
clock clkin (source latency)	0.00	2.00
clkin (in)	0.00	2.00 r
dqsen_reg/Q (dfnrbl)	0.31	2.31 f
dqsand/Z (an02d0)	0.09 H	2.40 f
dqspad/Z (bufbd1)	1.71 *	4.11 f
dqs (out)	0.00	4.11 f
clock network delay (propagated)	0.00	4.11
clock reconvergence pessimism	0.00	4.11
output external delay	-0.42	3.69
data required time		3.69

data required time		3.69
data arrival time		-3.96

slack (VIOLATED)		-0.26

Look at the difference in the capture clock path. The first one looks normal. The path starts at clkin, goes through the delay buffer, through the AND gate and the pad, and out to the dqs port. But the second one starts on the clkin pin, then *through the dqsen_reg flop!!!* What's going on here?

This is an example of what one Synopsys engineer humorously calls “promiscuous” backtracing. When calculating source latencies (which is what this is) on a generated clock, PT will backtrack through ANY number of sequential or combinational elements to find the shortest and longest paths.

So, how do we fix this? You would immediately think of trying to set a false path through that flop, but this won't work. In PT, set_false_path in only affects data paths, not clock paths.

I could use set_disable_timing, or set_case_analysis, but then I'll disable the clock gating check (and possibly other timing checks).

In this case, we create a `-divide_by 1` generated clock on the dqs AND gate clock pin slaved to the input clock. We then create the dqs write clock slaved to this generated clock, like this:

```
# Create the clkin clock
create_clock -period $_period -name clkin \
  [get_ports clkin]

# Create the cascaded clock
create_generated_clock \
  -name dqsclk_steer0 \
  -source [get_attribute [get_clocks clkin] sources] \
  -divide_by 1 \
  -master_clock clkin \
  -add \
  [get_pins dqsand/A2]

create_generated_clock \
  -name dqsoutclk \
  -source [get_attribute [get_clocks dqsclk_steer0] sources] \
  -divide_by 1 \
  -master_clock dqsclk_steer0 \
  -add \
  [get_ports dqs]

set_propagated_clock [all_clocks]
```

Now when we report the path, we get the correct clock trace and slack:

```
pt_shell> report_timing -fall_through [get_pins dout_mux/S] -fall_to
[get_clocks dqsoutclk] -path_type full_clock_expanded
```

```
Startpoint: clkin (clock source 'clkin')
Endpoint: dq (output port clocked by dqsoutclk)
Path Group: dqsoutclk
Path Type: max
```

Point	Incr	Path

clock clkin (fall edge)	2.00	2.00
clock source latency	0.00	2.00
clkin (in)	0.00	2.00 f
dout_mux/S (mx02d0) <-	0.00	2.00 f
dout_mux/Z (mx02d0)	0.23 H	2.24 f
dqpad/Z (bufbd1)	1.72 *	3.96 f
dq (out)	0.00	3.96 f
data arrival time		3.96
clock dqsoutclk (fall edge)	2.00	2.00
clock clkin (source latency)	0.00	2.00
clkin (in)	0.00	2.00 f
clkdelaybuf_clk/Z (bufbd1)	1.00 H	3.00 f
dqsand/A2 (an02d0) (gclock source)	0.10 *	3.10 f
dqsand/Z (an02d0)	0.11 *	3.22 f
dqspad/Z (bufbd1)	1.71 *	4.92 f
dqs (out)	0.00	4.92 f
clock network delay (propagated)	0.00	4.92
clock reconvergence pessimism	0.00	4.92
output external delay	-0.42	4.50
data required time		4.50

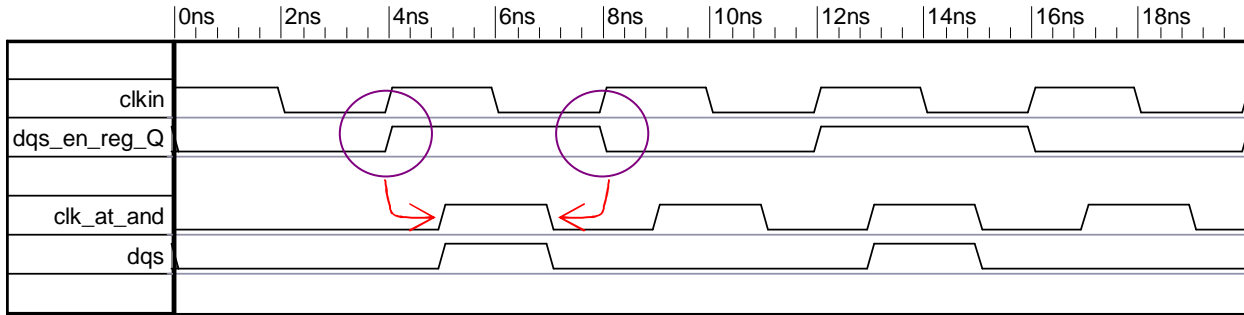
data required time		4.50
data arrival time		-3.96

slack (MET)		0.55

Notice the dqsand/A2 pin is now a “gclock source”.

3.4.2 The clock gating check

But we’re not done with this troublesome little gate yet. The reason it was inserted *after* the delay element was to make for a clean gating of the clock. The timing diagram looks like this:



Waveform created by Synapticad Waveformer Pro (www.syncad.com)

Figure 3-13

Since the clock is delayed by $\frac{1}{4}$ cycle, and the gating flop is clocked on a rising edge, this should provide nominally $\frac{1}{4}$ cycle of setup and hold for the clock gating check.

What does PT get?

Here's the setup clock gating check:

```
pt_shell> report_timing -group {**clock_gating_default**}
```

```
Startpoint: dqs_en_reg (rising edge-triggered flip-flop clocked by clk_in)
Endpoint: dqsand (rising clock gating-check end-point clocked by
dqsclk_steer0)
Path Group: **clock_gating_default**
Path Type: max
```

Point	Incr	Path
clock clk_in (rise edge)	0.00	0.00
clock network delay (propagated)	0.00	0.00
dqs_en_reg/CP (dfnrbl)	0.00	0.00 r
dqs_en_reg/Q (dfnrbl)	0.31	0.31 f
dqsand/A1 (an02d0)	0.00	0.31 f
data arrival time		0.31
clock dqsclk_steer0 (rise edge)	4.00	4.00
clock network delay (propagated)	1.10	5.10
clock reconvergence pessimism	0.00	5.10
dqsand/A2 (an02d0)		5.10 r
clock gating setup time	0.00	5.10
data required time		5.10
data required time		5.10
data arrival time		-0.31
slack (MET)		4.79

It passed, but it isn't right. PT defines a clock gating setup check as a comparison between the arrival time of the clock gating signal at the clock gate and the arrival time of the next edge at the clock gate.

This works great for the normal case of an opposite-edge enable flop directly to the AND gate. But our case is different. The clock edge we're checking is launched by the same edge as the gating signal. Once again, this is a 0 cycle multicycle path. We should be checking 0 against 0, not 0 against 4.

A similar problem occurs on clock gating hold.

```
pt_shell> report_timing -group {**clock_gating_default**} -delay min

Startpoint: dqsen_reg (rising edge-triggered flip-flop clocked by clkin)
Endpoint: dqsand (rising clock gating-check end-point clocked by
dqsclock_steer0)
Path Group: **clock_gating_default**
Path Type: min

Point                               Incr      Path
-----
clock clkin (rise edge)              0.00      0.00
clock network delay (propagated)     0.00      0.00
dqsen_reg/CP (dfnrb1)                0.00      0.00 r
dqsen_reg/Q (dfnrb1)                 0.31      0.31 r
dqsand/A1 (an02d0)                   0.00      0.31 r
data arrival time                     0.31
-----
clock dqsclock_steer0 (fall edge)    2.00      2.00
clock network delay (propagated)     1.12      3.12
clock reconvergence pessimism        0.00      3.12
dqsand/A2 (an02d0)                   0.00      3.12 f
clock gating hold time                0.00      3.12
data required time                    3.12
-----
data required time                    3.12
data arrival time                     -0.31
-----
slack (VIOLATED)                     -2.81
```

The clock gating hold check is defined as the latest arrival time of the clock edge at the clock gate against the earliest arrival of the clock gating signal at the clock gate. For our circuit, this should be a 2 against 0 or 4 against 2 check, not a 0 against 2 check.

The solution is to use `set_multicycle_path` to tell PT what the edge relationship is supposed to be. For setup, this is our old friend `set_multicycle_path 0`. It turns out that this moves the hold check to where we want it as well. So, for hold, we also use `set_multicycle_path 0` to show it has been checked:

```
set_multicycle_path -setup 0 -to dqsand/A1
set_multicycle_path -hold 0 -to dqsand/A1
```

Now we get the correct traces:

```
pt_shell> report_timing -group {**clock_gating_default**}
```

```
Startpoint: dqsen_reg (rising edge-triggered flip-flop clocked by clkin)
Endpoint: dqsand (rising clock gating-check end-point clocked by
dqsclock_steer0)
Path Group: **clock_gating_default**
Path Type: max
```

Point	Incr	Path
clock clkin (rise edge)	0.00	0.00
clock network delay (propagated)	0.00	0.00
dqsen_reg/CP (dfnrb1)	0.00	0.00 r
dqsen_reg/Q (dfnrb1)	0.31	0.31 f
dqsand/A1 (an02d0)	0.00	0.31 f
data arrival time		0.31

clock dqsclock_steer0 (rise edge)	0.00	0.00
clock network delay (propagated)	1.10	1.10
clock reconvergence pessimism	0.00	1.10
dqsand/A2 (an02d0)		1.10 r
clock gating setup time	0.00	1.10
data required time		1.10

data required time		1.10
data arrival time		-0.31

slack (MET)		0.79

```
pt_shell> report_timing -group {**clock_gating_default**} -delay min
```

```
Startpoint: dqsen_reg (rising edge-triggered flip-flop clocked by clkin)
Endpoint: dqsand (rising clock gating-check end-point clocked by
dqsclock_steer0)
Path Group: **clock_gating_default**
Path Type: min
```

Point	Incr	Path
clock clkin (rise edge)	4.00	4.00
clock network delay (propagated)	0.00	4.00
dqsen_reg/CP (dfnrb1)	0.00	4.00 r
dqsen_reg/Q (dfnrb1)	0.31	4.31 r
dqsand/A1 (an02d0)	0.00	4.31 r
data arrival time		4.31

clock dqsclock_steer0 (fall edge)	2.00	2.00
clock network delay (propagated)	1.12	3.12
clock reconvergence pessimism	0.00	3.12
dqsand/A2 (an02d0)		3.12 f
clock gating hold time	0.00	3.12
data required time		3.12

data required time		3.12
data arrival time		-4.31

slack (MET)		1.19

4 Timing this circuit using stamp models

<Added 21 Jan 2007>

Starting with version 2006.12, PrimeTime no longer supports STAMP models. The techniques described below are still valid, but QTM models must be substituted for STAMP. An addendum on how to do this is available on my website www.zimmerdesignservices.com.

References (2) and (3) propose timing this circuit by instantiating the chip in a larger “board” level environment and hooking the DDR pins to external models to do the checking. This allows you to write the models once and reuse them, and also makes the timing reports much more readable. It eliminates the need for output constraints, and turns all paths into “internal” paths.

The technique looks like this:

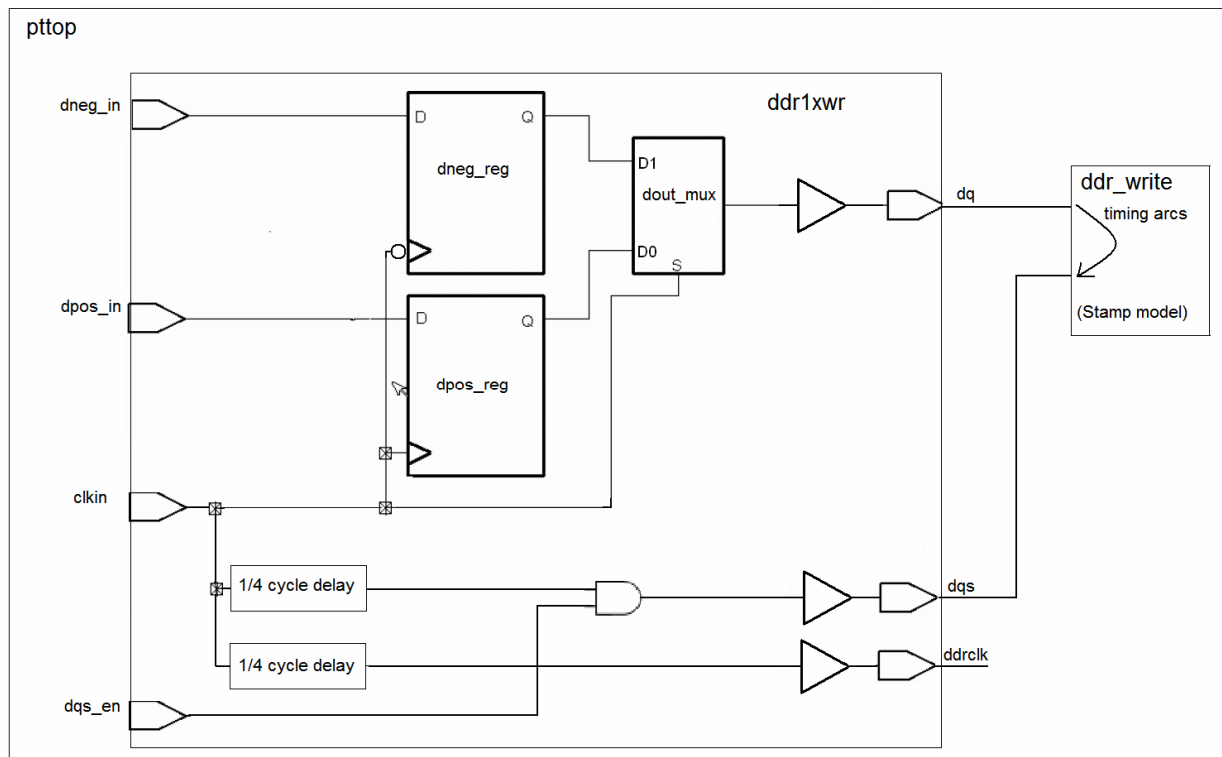


Figure 4-1

To make this work, we need to instantiate both the chip and the ddr_write model in a higher level. The top level code might look like this:

```
module pttop (
);

ddrlxwr ddrlxwr (
    .clkkin(clkin),
    .dpos_in(dpos_in),
    .dneg_in(dneg_in),
    .dqs_en(dqs_en),
    .dq(dq),
    .dqs(dqs),
    .ddrclk(ddrclk)
);

ddr_write ddr_write(
    .DQ_I (dq),
    .DQS_I (dqs)
);

endmodule
```

Notice that I have no ports at the top level (pttop). Whether the chip ports are re-instantiated at the top level is up to you. Re-instantiating and hooking up the ports will allow you to use “get_ports” without changing to “get_pins” if you’re using parts of existing scripts. If you’re starting from scratch, it is simpler to make the top have no ports at all.

4.1 The stamp model and hooking it in

The stamp model code itself is pretty simple. There are two files – the ddr_write.mod and ddr_write.data files. The ddr_write.mod file just defines the ports and the timing arcs:

```
MODEL

/* MODEL HEADER */
DESIGN "ddr_write";

INPUT DQ_I;
INPUT DQS_I;

/* Data In */

Tds_dq_pos_0      : SETUP (POSEDGE, EQUIVALENT) DQ_I  DQS_I;
Tdh_dq_pos_0      : HOLD  (POSEDGE, EQUIVALENT) DQ_I  DQS_I;

Tds_dq_neg_0      : SETUP (NEGEDGE, EQUIVALENT) DQ_I  DQS_I;
Tdh_dq_neg_0      : HOLD  (NEGEDGE, EQUIVALENT) DQ_I  DQS_I;

ENDMODEL
```

So, the model has two input ports – DQ_I and DQS_I. There is a setup check of DQ_I to posedge DQS_I called Tds_dq_pos_0, a hold check of DQ_I to posedge DQS_I of Tdh_dq_pos_0, etc.

The ddr_write.data gives the values for the timing arcs:

```
MODELDATA
DESIGN "ddr_write";

PORTDATA
  DQ_I : cap (0.0) ;
  DQS_I : cap (0.0) ;
ENDPORTDATA

TIMINGDATA
ARCDATA
  Tds_dq_pos_0, Tds_dq_neg_0 :
    CONSTRAINT(SCALAR) {
      VALUES("0.5")
    }
  Tdh_dq_pos_0, Tdh_dq_neg_0 :
    CONSTRAINT(SCALAR) {
      VALUES("0.5")
    }
ENDARCDATA

ENDTIMINGDATA
ENDMODELDATA
```

I have set the setup and hold constraints to 0.5 to match the values of `_tDS` and `_tDH` used earlier. So, we should end up with the same slack values.

That's all easy enough. But, being new to stamp models, I had a tough time figuring out how to link this in.

First, we need to compile the stamp model:

```
# vim:filetype=pt
compile_stamp_model \
  -model_file ddr_write.mod \
  -data_file ddr_write.data \
  -output ddr_write \
  -library
```

This creates a file `“ddr_write_lib.db”`. The `“_lib”` is added automatically. To get PT to link this into the design, we have to do two things. First, we need to add it to the `link_library` variable:

```
if {[lsearch $link_library ddr_write_lib.db] == -1} {
  set link_library [concat $link_library ddr_write_lib.db]
}
```

(The `lsearch` checks to see if it is already there, so we can re-run the script multiple times without piling onto the `link_library` variable).

But that isn't all we need to do. After reading in the top level and chip level code, we need to use the “-keep” option on link to make the stamp model link in properly:

```
read_verilog {pttop.gv ddr1xwr.gv}
current_design pttop ; link -keep
```

4.2 The script itself

One of the biggest drawbacks of this technique is the fact that every path that references the chip will have to have a hierarchical prefix. To simplify this somewhat, I define a variable to cover these paths. I define it with the “/” at the end so that I can set it to {} if I want to use the code in a non-hierarchical context:

```
set _chip ddr1xwr/
```

Remember that PT doesn't know what is chip and what is not. So, it will treat the dummy hookup wires for the stamp model as real chip traces. When timing with parasitics, removing the effect of these wires can be tricky. Be sure to look at the timing traces carefully and pay attention to any warnings PT gives. In particular, watch out for extra delay on the wires and for timing threshold issues.

To keep this example simple, I'm using an sdf flow and I can just use set_annotated_delay to force the delays I want onto these wires. These correspond to the board delays, so I use the same variables for these wire delays as I used when I calculated the output delays in the constraints code earlier:

```
set_annotated_delay -net -max $_dq_wire_delay_max \
  -from [get_pins $_chip]dqpad/Z] \
  -to [get_pins ddr_write/DQ_I]

set_annotated_delay -net -min $_dq_wire_delay_min \
  -from [get_pins $_chip]dqpad/Z] \
  -to [get_pins ddr_write/DQ_I]

set_annotated_delay -net -max $_dqs_wire_delay_max \
  -from [get_pins $_chip]dqspad/Z] \
  -to [get_pins ddr_write/DQS_I]

set_annotated_delay -net -min $_dqs_wire_delay_min \
  -from [get_pins $_chip]dqspad/Z] \
  -to [get_pins ddr_write/DQS_I]
```

One note on this: the set_annotated_delay command must be run on real, physical cell pins. Thus, I had to use “\$_chip]/dqspad/Z” instead of just “\$_chip]/dqs” (which is a hierarchical pin).

The rest of the script is pretty much unchanged, except for the adding of the `$_chip` prefix. Also, since I chose not to pull all the chip ports up to the top level, references to “get_ports” have to be changed to “get_pins `$_chip`”. So, the clock code now looks like this:

```
create_clock -period $_period -name clkkin \  
  [get_pins $_chip]clkkin]  
  
create_generated_clock \  
  -name dqsoutclk \  
  -source [get_attribute [get_clocks clkkin] sources] \  
  -divide_by 1 \  
  -master_clock clkkin \  
  -add \  
  [get_pins $_chip]dqs]
```

The `set_output_delay` code is eliminated completely, since all the paths will now be internal.

The false and multicycle paths are the same except for the path prefix:

```
set_false_path -from [get_pins $_chip]dpos_reg/CP] -rise_to [get_clocks  
dqsoutclk]  
set_false_path -from [get_pins $_chip]dneg_reg/CPN] -fall_to [get_clocks  
dqsoutclk]  
  
set_multicycle_path -setup 0 -rise_through [get_pins $_chip]dout_mux/S] -  
rise_to [get_clocks dqsoutclk]  
set_multicycle_path -hold 0 -rise_through [get_pins $_chip]dout_mux/S] -  
rise_to [get_clocks dqsoutclk]  
set_multicycle_path -setup 1 -rise_through [get_pins $_chip]dout_mux/S] -  
fall_to [get_clocks dqsoutclk]  
set_multicycle_path -hold 0 -rise_through [get_pins $_chip]dout_mux/S] -  
fall_to [get_clocks dqsoutclk]  
set_multicycle_path -setup 0 -fall_through [get_pins $_chip]dout_mux/S] -  
fall_to [get_clocks dqsoutclk]  
set_multicycle_path -hold 0 -fall_through [get_pins $_chip]dout_mux/S] -  
fall_to [get_clocks dqsoutclk]  
set_multicycle_path -setup 1 -fall_through [get_pins $_chip]dout_mux/S] -  
rise_to [get_clocks dqsoutclk]  
set_multicycle_path -hold 0 -fall_through [get_pins $_chip]dout_mux/S] -  
rise_to [get_clocks dqsoutclk]
```

4.3 Timing results

The path slacks all come out the same (as you would expect), but the trace reports are a little different. Here's the path we looked out earlier with the 0.55 slack (using the constraints code):

```
pt_shell> report_timing -fall_through [get_pins dout_mux/S] -fall_to
[get_clocks dqsoutclk] -path_type full_clock_expanded
```

```
Startpoint: clkin (clock source 'clkin')
Endpoint: dq (output port clocked by dqsoutclk)
Path Group: dqsoutclk
Path Type: max
```

Point	Incr	Path
clock clkin (fall edge)	2.00	2.00
clock source latency	0.00	2.00
clkin (in)	0.00	2.00 f
dout_mux/S (mx02d0) <-	0.00	2.00 f
dout_mux/Z (mx02d0)	0.23 H	2.24 f
dqpad/Z (bufbd1)	1.72 *	3.96 f
dq (out)	0.00	3.96 f
data arrival time		3.96
clock dqsoutclk (fall edge)	2.00	2.00
clock clkin (source latency)	0.00	2.00
clkin (in)	0.00	2.00 f
clkdelaybuf_clk/Z (bufbd1)	1.00 H	3.00 f
dqsand/Z (an02d0)	0.22 *	3.22 f
dqspad/Z (bufbd1)	1.71 *	4.92 f
dqs (out)	0.00	4.92 f
clock network delay (propagated)	0.00	4.92
clock reconvergence pessimism	0.00	4.92
output external delay	-0.42	4.50
data required time		4.50
data required time		4.50
data arrival time		-3.96
slack (MET)		0.55

Here it is using this stamp model technique:

```
pt_shell> report_timing -fall_through [get_pins ${_chip}dout_mux/S] -fall_to
[get_clocks dqsoutclk] -path_type full_clock_expanded
```

```
Startpoint: ddr1xwr/clkin
              (clock source 'clkin')
Endpoint: ddr_write (falling edge-triggered flip-flop clocked by dqsoutclk)
Path Group: dqsoutclk
Path Type: max
```

Point	Incr	Path

clock clkin (fall edge)	2.00	2.00
clock source latency	0.00	2.00
ddr1xwr/clkin (ddr1xwr)	0.00	2.00 f
ddr1xwr/dout_mux/S (mx02d0) <-	0.00	2.00 f
ddr1xwr/dout_mux/Z (mx02d0)	0.23 *	2.23 f
ddr1xwr/dqpad/Z (bufbd1)	1.72 *	3.95 f
ddr1xwr/dq (ddr1xwr)	0.00 *	3.95 f
ddr_write/DQ_I (ddr_write)	1.00 *	4.95 f
data arrival time		4.95
clock dqsoutclk (fall edge)	2.00	2.00
clock clkin (source latency)	0.00	2.00
ddr1xwr/clkin (ddr1xwr)	0.00	2.00 f
ddr1xwr/clkdelaybuf_clk/Z (bufbd1)	1.00 *	3.00 f
ddr1xwr/dqsand/Z (an02d0)	0.22 *	3.22 f
ddr1xwr/dqspad/Z (bufbd1)	1.71 *	4.92 f
ddr1xwr/dqs (ddr1xwr) (gclock source)	0.00 H	4.92 f
ddr_write/DQS_I (ddr_write)	1.08	6.00 f
clock reconvergence pessimism	0.00	6.00
library setup time	-0.50	5.50
data required time		5.50

data required time		5.50
data arrival time		-4.95

slack (MET)		0.55

The stamp timing trace is much nicer. Instead of having an output delay that is the result of a messy calculation involving the board delays (0.50 – 1.08 + 1.00), you get the true setup requirement (0.50), and you can see the board paths (1.00 through dq and 1.08 through dqs). Very nice.

The fact that PT can now see the board delays explicitly also means that it can do CRPR (clock reconvergence pessimism removal) automatically. This doesn't matter for the DDR write case we are examining, but it is a huge advantage for the DDR read, where paths go out to the SDRAM and back again. Timing the read path using constraints involves all sorts of messy stuff to manage the CRPR problem without help from PT.

4.4 Comparison of constraint versus stamp model

Advantages of Stamp model approach:

- Conceptually simpler
- No need for messy calculations to create output delays
- More portable (model can be developed once and easily reused without specialized PT knowledge)
- Timing traces make more sense
- **PT can do CRPR**

Disadvantages of Stamp model approach:

- Somewhat unusual
- Controlling effects of dummy wires in parasitics mode
- Can't be output as SDC for other tools
- **Need for prefixes throughout the script**

Wouldn't it be nice if we could somehow use the stamp models, but without introducing an extra level of hierarchy?

5 Timing this circuit using stamp models and create_cell

That extra level of hierarchy required by the stamp model approach is a bit of a problem, but there is a way around it. Once, when I was lamenting the lack of a command in PT to create arbitrary timing arcs, Steve Golson pointed out to me that the command “create_cell” works in PT, which got me to thinking.

Why not hook up the stamp model directly in the chip hierarchy using create_cell?

The resulting design would then look like this:

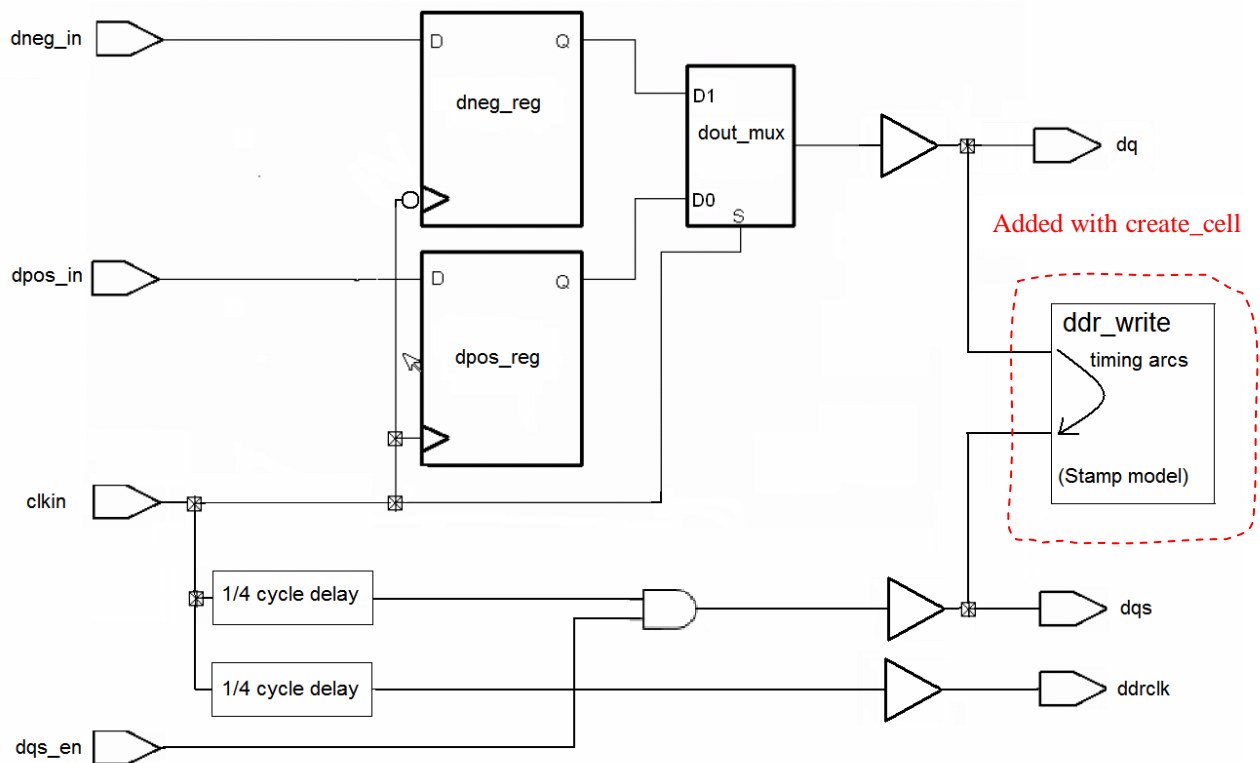


Figure 5-1

5.1 Hooking in the stamp model using create_cell

One nice thing about this approach is that the linking is simpler, because there’s nothing in the design now that needs to be linked to the stamp model at the time the design is read in.

```

if {[lsearch $link_library ddr_write_lib.db] == -1} {
    set link_library [concat $link_library ddr_write_lib.db]
}

compile_stamp_model \
    -model_file ddr_write.mod \
    -data_file ddr_write.data \
    -output ddr_write \
    -library

read_verilog ddr1xwr.gv
link
read_sdf ddr1xwr.sdf

```

Now we stitch in stamp model. The code to do this actually turns out to be quite simple, since the nets already exist:

```

# Stitch in the ddr model
create_cell ddr_write ddr_write_lib/ddr_write
connect_net [all_connected [get_ports dq]] [get_pins ddr_write/DQ_I]
connect_net [all_connected [get_ports dqs]] [get_pins ddr_write/DQS_I]

```

The “all_connected [get_ports..” gets the actual net connected to the port.

Now we annotate the delays as before:

```

set_annotated_delay -net -max $_dq_wire_delay_max \
    -from [get_pins dqpad/Z] \
    -to [get_pins ddr_write/DQ_I]

set_annotated_delay -net -min $_dq_wire_delay_min \
    -from [get_pins dqpad/Z] \
    -to [get_pins ddr_write/DQ_I]

set_annotated_delay -net -max $_dqs_wire_delay_max \
    -from [get_pins dqspad/Z] \
    -to [get_pins ddr_write/DQS_I]

set_annotated_delay -net -min $_dqs_wire_delay_min \
    -from [get_pins dqspad/Z] \
    -to [get_pins ddr_write/DQS_I]

```

5.2 The script itself

Most of the rest of the script is pretty familiar. The clock creation code needs one slight modification. Instead of creating the generated clock on the port, we need to create it on the *pad pin* so that it can propagate to our stamp model:

```

create_clock -period $_period -name clkin \
  [get_ports clkin]

create_generated_clock \
  -name dqsoutclk \
  -source [get_attribute [get_clocks clkin] sources] \
  -divide_by 1 \
  -master_clock clkin \
  -add \
  [get_pins dqspad/Z]

```

The rest of the script is unchanged from the constraints version, except we don't need the output delay code:

```

set_false_path -from [get_pins dpos_reg/CP] -rise_to [get_clocks dqsoutclk]
set_false_path -from [get_pins dneg_reg/CPN] -fall_to [get_clocks dqsoutclk]

set_multicycle_path -setup 0 -rise_through [get_pins dout_mux/S] -rise_to
[get_clocks dqsoutclk]
set_multicycle_path -hold 0 -rise_through [get_pins dout_mux/S] -rise_to
[get_clocks dqsoutclk]
set_multicycle_path -setup 1 -rise_through [get_pins dout_mux/S] -fall_to
[get_clocks dqsoutclk]
set_multicycle_path -hold 0 -rise_through [get_pins dout_mux/S] -fall_to
[get_clocks dqsoutclk]
set_multicycle_path -setup 0 -fall_through [get_pins dout_mux/S] -fall_to
[get_clocks dqsoutclk]
set_multicycle_path -hold 0 -fall_through [get_pins dout_mux/S] -fall_to
[get_clocks dqsoutclk]
set_multicycle_path -setup 1 -fall_through [get_pins dout_mux/S] -rise_to
[get_clocks dqsoutclk]
set_multicycle_path -hold 0 -fall_through [get_pins dout_mux/S] -rise_to
[get_clocks dqsoutclk]

```

5.3 Timing traces

Again, we get all the same path slacks as before. Let's look at our sample trace using this technique:

```
pt_shell> report_timing -fall_through [get_pins dout_mux/S] -fall_to
[get_clocks dqsoutclk] -path_type full_clock_expanded
```

```
Startpoint: clkin (clock source 'clkin')
Endpoint: ddr_write (falling edge-triggered flip-flop clocked by dqsoutclk)
Path Group: dqsoutclk
Path Type: max
```

Point	Incr	Path

clock clkin (fall edge)	2.00	2.00
clock source latency	0.00	2.00
clkin (in)	0.00	2.00 f
dout_mux/S (mx02d0) <-	0.00	2.00 f
dout_mux/Z (mx02d0)	0.23 H	2.24 f
dqpad/Z (bufbd1)	1.72 *	3.96 f
ddr_write/DQ_I (ddr_write)	1.00 *	4.96 f
data arrival time		4.96
clock dqsoutclk (fall edge)	2.00	2.00
clock clkin (source latency)	0.00	2.00
clkin (in)	0.00	2.00 f
clkdelaybuf_clk/Z (bufbd1)	1.00 H	3.00 f
dqsand/Z (an02d0)	0.22 *	3.22 f
dqspad/Z (bufbd1) (gclock source)	1.71 *	4.92 f
ddr_write/DQS_I (ddr_write)	1.08 *	6.00 f
clock reconvergence pessimism	0.00	6.00
library setup time	-0.50	5.50
data required time		5.50

data required time		5.50
data arrival time		-4.96

slack (MET)		0.55

As with the hierarchical stamp model approach, we get a nice clean trace. And, as with the hierarchical stamp model, PT will do the CRPR for us automatically.

I think this technique has a lot of potential. It provides the advantages of the stamp model approach (which really show up in the read direction) without the accompanying complexity of adding an extra level of hierarchy.

Let's revisit that list of advantages and disadvantages, but using the stitched-in stamp model instead:

Advantages of Stitched-in Stamp model approach:

- Conceptually simpler
- No need for messy calculations to create output delays
- More portable (model can be developed once and easily reused without specialized PT knowledge)
- **Timing traces make more sense**
- **PT can do CRPR**

Disadvantages of Stamp model approach:

- Somewhat unusual
- Can't be output as SDC directly for other tools, but easier to deal with because no extra hierarchy.

Other than the fact that people will wonder what you're up to, the only real disadvantage is that the output can't be used as SDC for other tools. But I have found this to be true even for the constraint approach in the DDR *read* direction. There are some things that just cannot be described with SDC.

At least the removal of the extra level of hierarchy makes it possible to delete the stamp model and write out the non-DDR SDC.

I used the stitched-in stamp model technique on a recent chip to model an external feedback path, and it worked great.

6 Conclusion

Recent changes to PT have made timing the write interface of DDR SDRAMs much simpler. This paper has presented three different approaches (four if you count the two-clock constraints approach in the appendix). They all work and produce the same timing results, and each has its own advantages and disadvantages. As they say in perl-land: TMTOWTDI – There’s More Than One Way To Do It!

The stamp model approaches, and the stitched-in stamp model approach in particular, show a lot of promise. While I haven’t covered the DDR read direction in this paper, the stamp model approach really shines in the read direction.

In the past, I have used the two-clock constraints approach, but I think I’ll try the stitched-in stamp model approach on my next DDR project.

7 Acknowledgements

The author would like to acknowledge the following people for their assistance and review:

Erich Whitney, Mercury Computer Systems

Stuart Hecht, Independent ASIC Design Consultant

8 References

- (1) Working with DDRs in PrimeTime
Paul Zimmer, Andrew Cheng
Synopsys Users Group 2002 San Jose
(available at www.zimmerdesignservices.com)
- (2) System Level STA Methodology with *DDR*
Robin Ko
Synopsys Users Group 2004 San Jose
- (3) Static Timing Verification for Complex SoC Design – Part I: *DDR* SDRAM Timing
Check in Primetime Revisit
Hui Fu
Synopsys Users Group 2002 Singapore

9 Appendix

9.1 Code for the two-clock approach

Reference (1) also showed how to do this using virtual clocks. Now that Primitime has multiclock propagation, we don't need to create virtual clocks and time the paths ourselves to determine the source latency (when using the two-clock approach). We can just create the two generated clocks directly.

First we turn on multiclock propagation and create the main input clock:

```
set timing_enable_multiple_clocks_per_reg true
# Create the clkin clock
create_clock -period $_period -name clkin \
  [get_ports clkin]
```

Then we create the two generated clocks:

```
create_generated_clock \
  -name posdqsout \
  -source [get_attribute [get_clocks clkin] sources] \
  -divide_by 1 \
  -master_clock clkin \
  -add \
  [get_ports dqs]

create_generated_clock \
  -name negdqsout \
  -source [get_attribute [get_clocks clkin] sources] \
  -divide_by 1 \
  -master_clock clkin \
  -add \
  [get_ports dqs]

set_propagated_clock [all_clocks]
```

Next we set the output delays to each clock:

```
set_output_delay -max \
  -clock posdqsout \
  [expr $_tDS + $_dq_wire_delay_max - $_dqs_wire_delay_min] \
  [get_ports dq]
set_output_delay -min \
  -add_delay \
  -clock posdqsout \
  [expr (-1 * $_tDH) + $_dq_wire_delay_min - $_dqs_wire_delay_max] \
  [get_ports dq]

set_output_delay -max \
  -add_delay -clock_fall \
  [expr $_tDS + $_dq_wire_delay_max - $_dqs_wire_delay_min] \
  -clock negdqsout \
  [get_ports dq]
```

```

set_output_delay -min \
  -add_delay -clock_fall \
  [expr (-1 * $_tDH) + $_dq_wire_delay_min - $_dqs_wire_delay_max] \
  -clock negdqsout \
  [get_ports dq]

```

As with the 1 clock code, we need false paths from the dpos/dneg flops to the clock that *doesn't* sample their data:

```

set_false_path -from [get_pins dpos_reg/CP] -to [get_clocks posdqsout]
set_false_path -from [get_pins dneg_reg/CPN] -to [get_clocks negdqsout]

```

The multicycle paths are just like the 1 clock code, except that instead of “-rise_to” we can just use “-to” the rising edge clock:

```

set_multicycle_path -setup 0 -rise_through [get_pins dout_mux/S] -rise_to
[get_clocks dqsoutclk]
set_multicycle_path -hold 0 -rise_through [get_pins dout_mux/S] -rise_to
[get_clocks dqsoutclk]

```

And so forth for the other multicycle paths:

```

set_multicycle_path -setup 1 -rise_through [get_pins dout_mux/S] -fall_to
[get_clocks dqsoutclk]
set_multicycle_path -hold 0 -rise_through [get_pins dout_mux/S] -fall_to
[get_clocks dqsoutclk]
set_multicycle_path -setup 0 -fall_through [get_pins dout_mux/S] -fall_to
[get_clocks dqsoutclk]
set_multicycle_path -hold 0 -fall_through [get_pins dout_mux/S] -fall_to
[get_clocks dqsoutclk]
set_multicycle_path -setup 1 -fall_through [get_pins dout_mux/S] -rise_to
[get_clocks dqsoutclk]
set_multicycle_path -hold 0 -fall_through [get_pins dout_mux/S] -rise_to
[get_clocks dqsoutclk]

```

This produces identical timing results to the other techniques.