PandABlocks-FPGA Documentation

Release 3.0a1-11-gdb8fdc4-dirty

Tom Cobb

Sep 14, 2021

Introduction

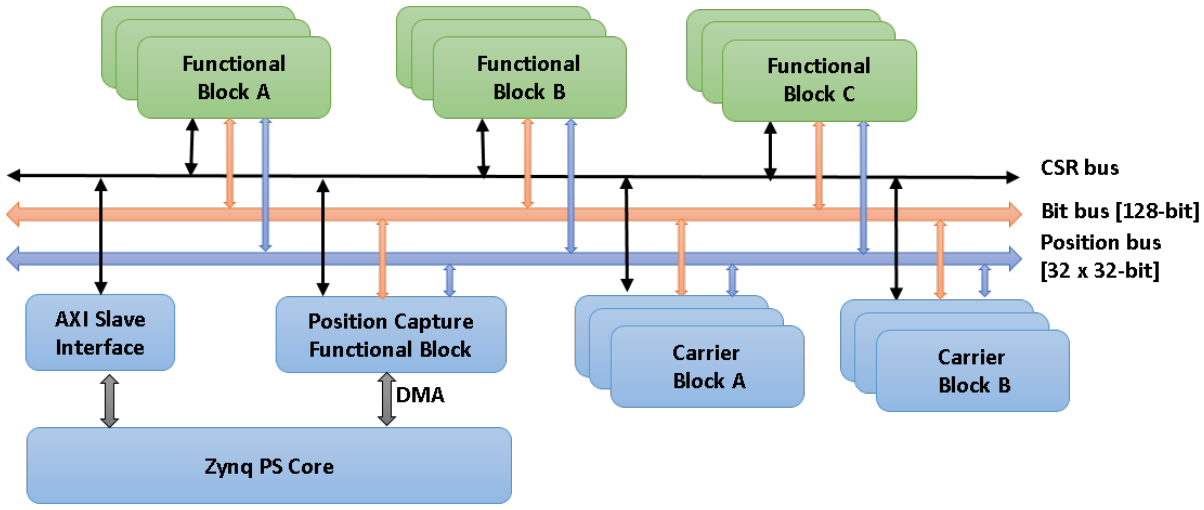
CHAPTER 1

PandABlocks-FPGA

PandABlocks-FPGA contains the firmware that runs on the FPGA inside a Zynq module that is the heart of a *PandABlocks Device* like *PandABox*.

1.1 What can PandABlocks do?

PandABlocks is a framework enabling a number of functional *Block* instances to be written and loaded to an FPGA, with their parameters (including their connections to other Blocks) changed at runtime. It allows flexible triggering and processing systems to be created, by users who are unfamiliar with writing FPGA firmware.



1.2 How is the documentation structured?

The documentation is structured into a series of *Tutorials* and some general *Reference* documentation. End users and developers need different documentation, so links for various categories of user are listed below:

1.2.1 Using an existing PandABlocks device

Work through the *Tutorials*.

1.2.2 Generating a new set of Blocks for a PandABlocks device

Read *Available Blocks* to find out what already exists, then read *Assembling Blocks into an App* to see how to make an *App* of these Blocks that can be loaded to a PandABlocks device.

1.2.3 Extending the functionality of a PandABlocks device

Read *Available Blocks* to see if you need to create a new Block or add to an existing one. Read *Writing a Block* to find out how to specify the interface to a Block, VHDL entity, timing tests and docs.

1.2.4 Working on the core autogeneration framework

Read *Writing a Block* to find out how the process works, then *Autogeneration framework architecture* for more details on specific parts of the autogeneration framework

CHAPTER 2

Blinking LEDs Tutorial

This tutorial will introduce you to the basics of PandABlocks, how to wire Blocks together to make different LEDs flash at different rates

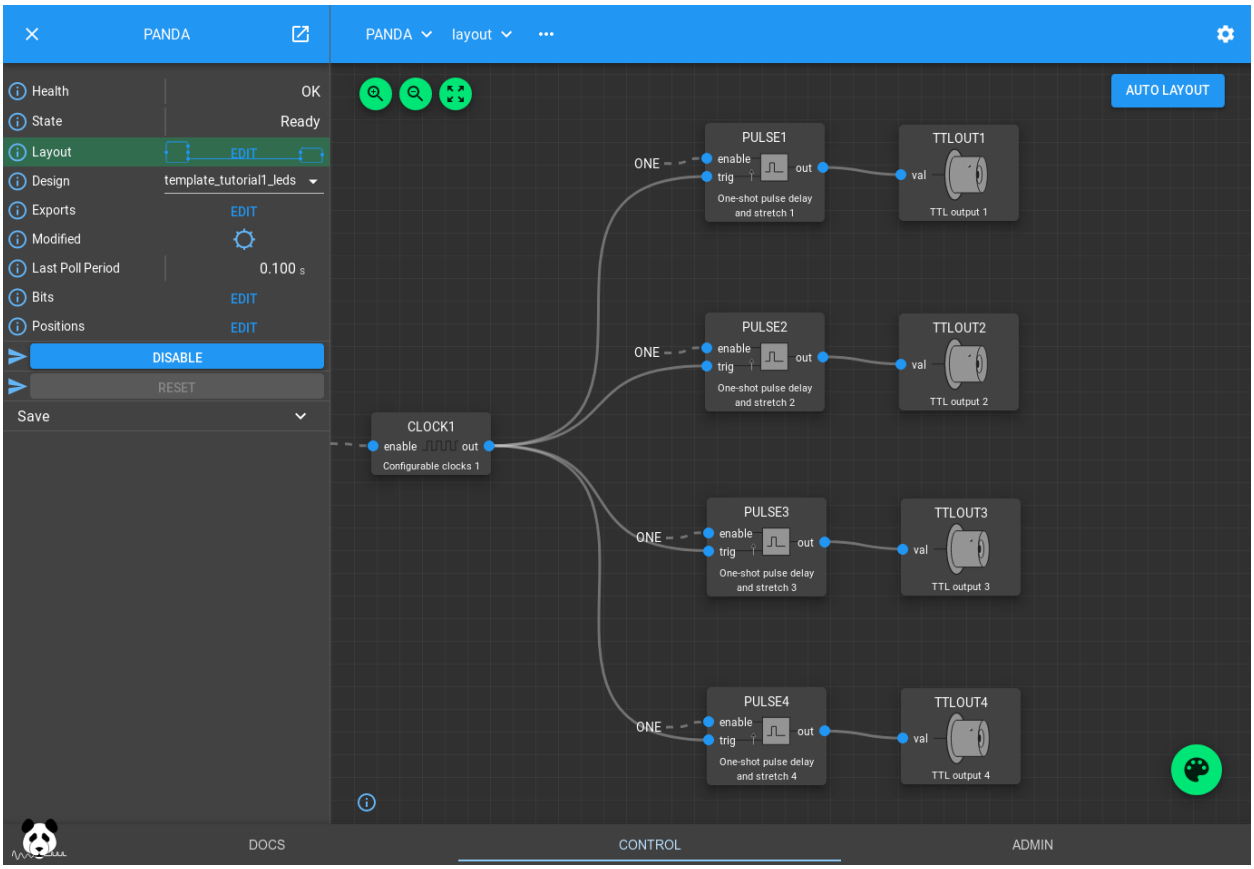
2.1 Opening the GUI

Point your web browser at the ip address or hostname of the PandA and you will be greeted with a welcome page. At the bottom of this page will be links for Docs, Control and Admin. You can use the Control link to open the Web Control page that we will use in these tutorials. For more information on the Web Control, see its entry in the Docs section.

2.2 Loading the tutorial design

The Design dropdown box allows you to select from saved designs stored on the PandA. Selecting an item from this list will load the saved design over the current Block settings. You can use the Save method to save your current design if you wish to keep it.

Select “template_tutorial1_leds” from the box and the settings and wiring of the Blocks in the PandA will be changed to the following:

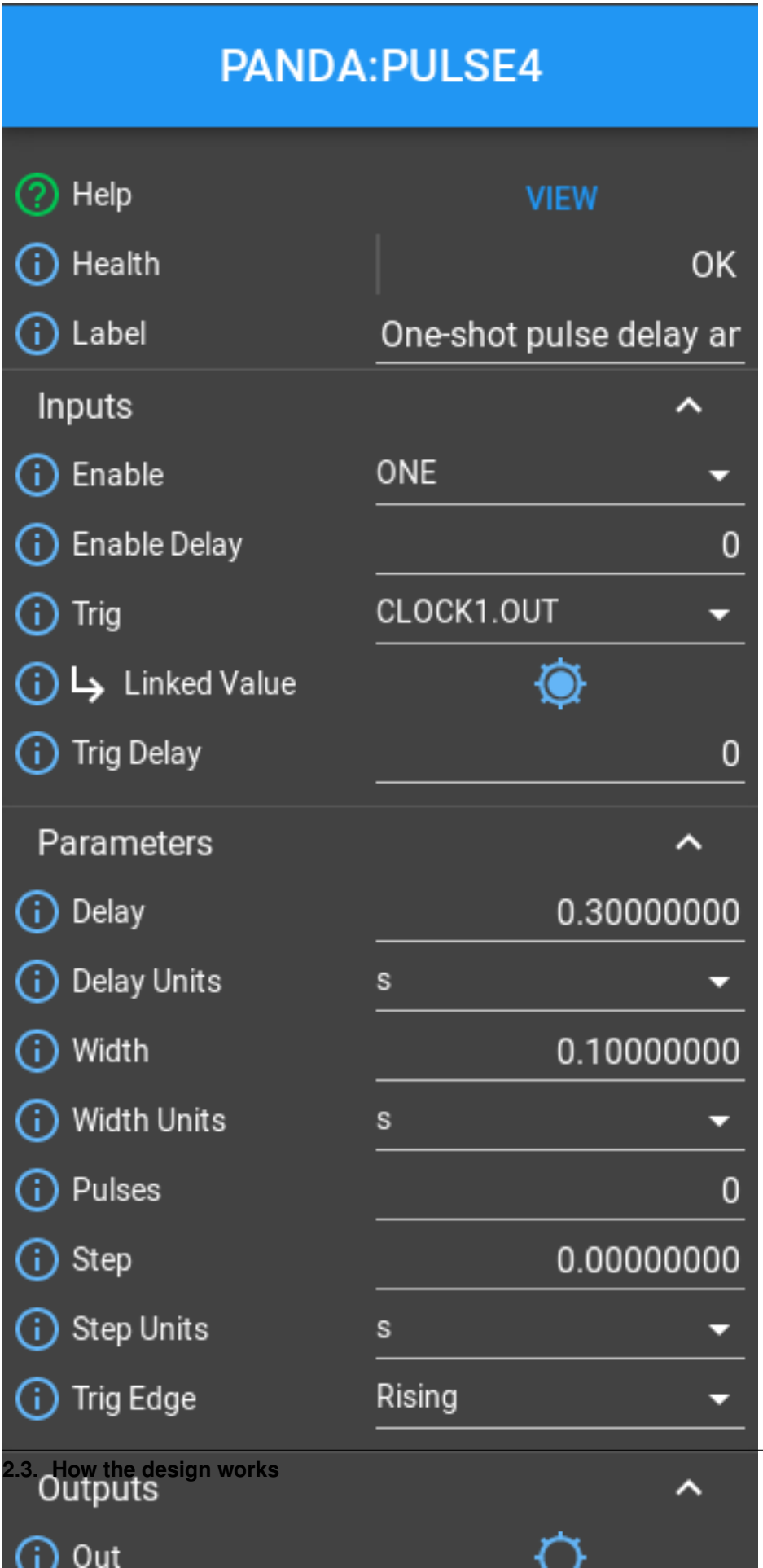


If you now look at the front panel of the PandA you should see the first 4 TTL output LEDs turn on sequentially, then turn off in the opposite order.

2.3 How the design works

The CLOCKS Block is creating a 50% duty cycle pulse train with a period of 1s. PULSE1..4 are taking this as an input trigger, and producing a different width pulse with a different delay for each PULSE Block. These PULSE Blocks work as a delay line, queuing a series of pulses up to be sent out when the delay expires.

If you click on one of them you can see its settings:


PANDA:PULSE4
Help
VIEW
Health
OK
Label
One-shot pulse delay ar
Inputs
Enable
ONE
Enable Delay
0
Trig
CLOCK1.OUT
Linked Value
Trig Delay
0
Parameters
Delay
0.30000000
Delay Units
s
Width
0.10000000
Width Units
s
Pulses
0
Step
0.00000000
Step Units
s
Trig Edge
Rising
Outputs
Out

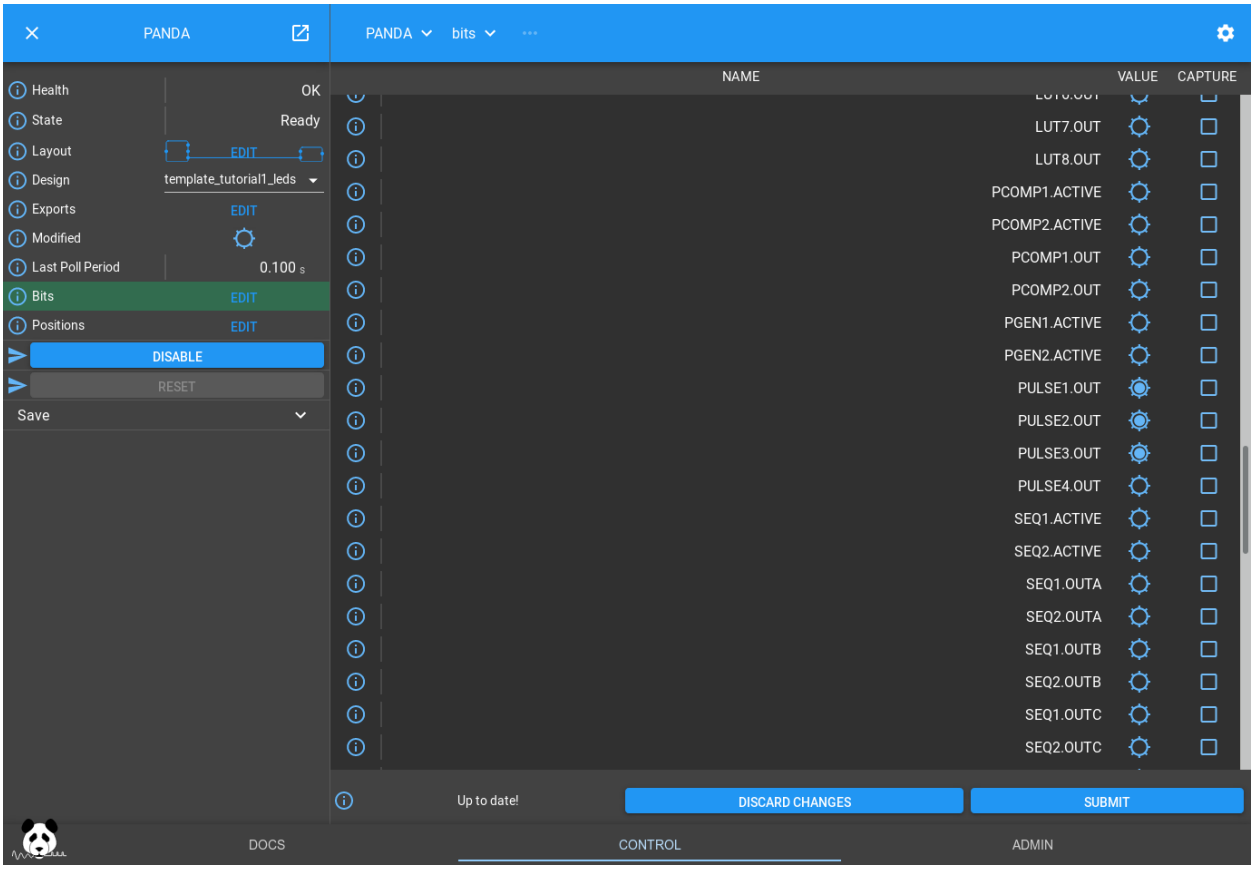
If you increase the delay beyond the 1s period you will notice that the `Queued` field will increase, but the PULSE Block will still continue outputting pulses after the desired delay. However if you increase the width beyond the pulse period the Block will drop the pulse, reporting it via the `Dropped` field. This is so it avoids merging them together.

You can also try clicking on the CLOCKS Block to modify the period of the input pulse train.

You can also try wiring these outputs to different TTLOUT Blocks by clicking the Palette icon, dragging a TTLOUT Block onto the canvas, and connecting it up by dragging the PULSE out port to the TTLOUT val port.

2.4 The Bit Bus

All ports on the visible Blocks are blue. They represent bits, single boolean values that can propagate through the system by connecting Blocks together. These outputs can be viewed on their respective Blocks by clicking them on the design, or all together by clicking the Bits field in the left hand pane:



If you scroll down to the section with the Pulse blocks you will see the same pattern of flashing lights as on the front of the PandA

Note: The web GUI polls the PandA at 10Hz, receiving the current value of each bit and whether it has changed. The web GUI uses this information to reflect the current value of each bit if pulsing at less than 5Hz, and displaying a 5Hz pulsing value if faster than 5Hz. This means that you will see even short pulses reflected on the web GUI. The front panel LEDs have a similar behaviour but with a maximum rate of 10Hz.

2.5 Conclusion

This tutorial has shown how to load a saved design and modify some parameters. It has also introduced the PULSE delay block that is useful for delaying and stretching trigger signals. It has introduced bit outputs and shown how they can be connected to the outside world using the TTLOUT Blocks. In the next tutorial we will read about position outputs, how they can be set and how they can be captured.

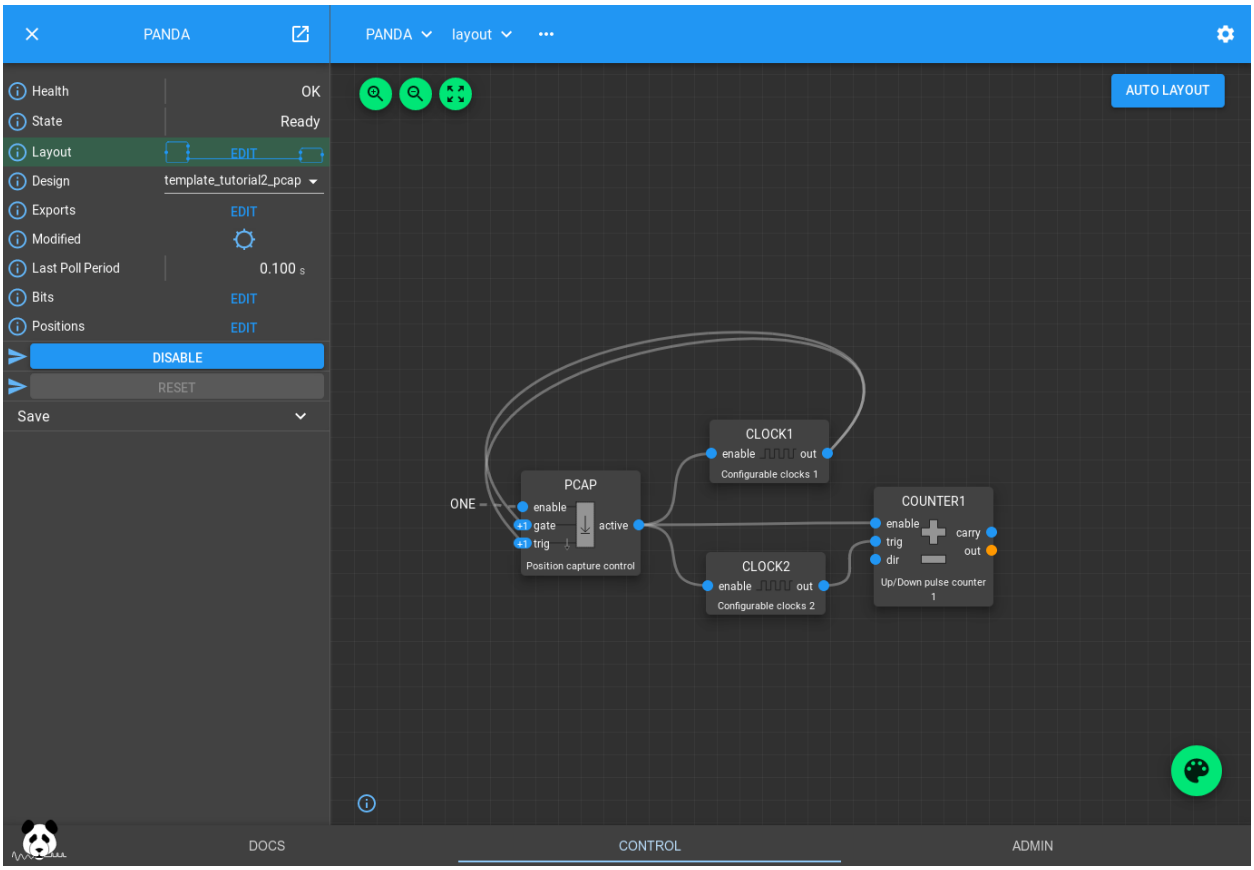
CHAPTER 3

Position Capture Tutorial

This tutorial will introduce you to the Position Capture interface of PandABlocks, how to provide trigger signals to control when these capture points are taken and visualize the data.

3.1 Loading the tutorial design

Select “template_tutorial2_pcap” from the Design dropdown box and the settings and wiring of the Blocks in the PandA will be changed to the following:



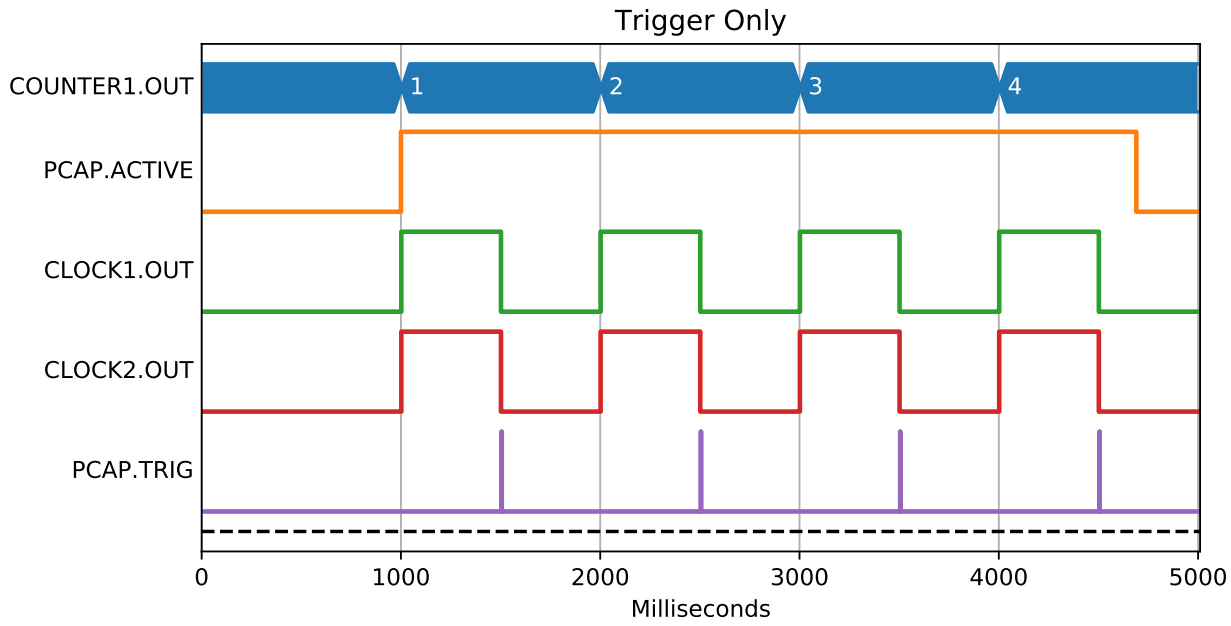
3.2 How the design works

This design has two CLOCK Blocks, which are enabled as soon as the PCAP Block becomes active:

- The first CLOCK is wired to PCAP trigger and gate. The gate is a level driven signal that provides the window of time that a capture should be active over. The trigger is an edge driven signal that actually captures data. In this example, PCAP.TRIG_EDGE=”Falling” so capture will be triggered on a falling edge of the trigger.
- The second CLOCK is wired to a COUNTER, triggering the increment of the counter value.

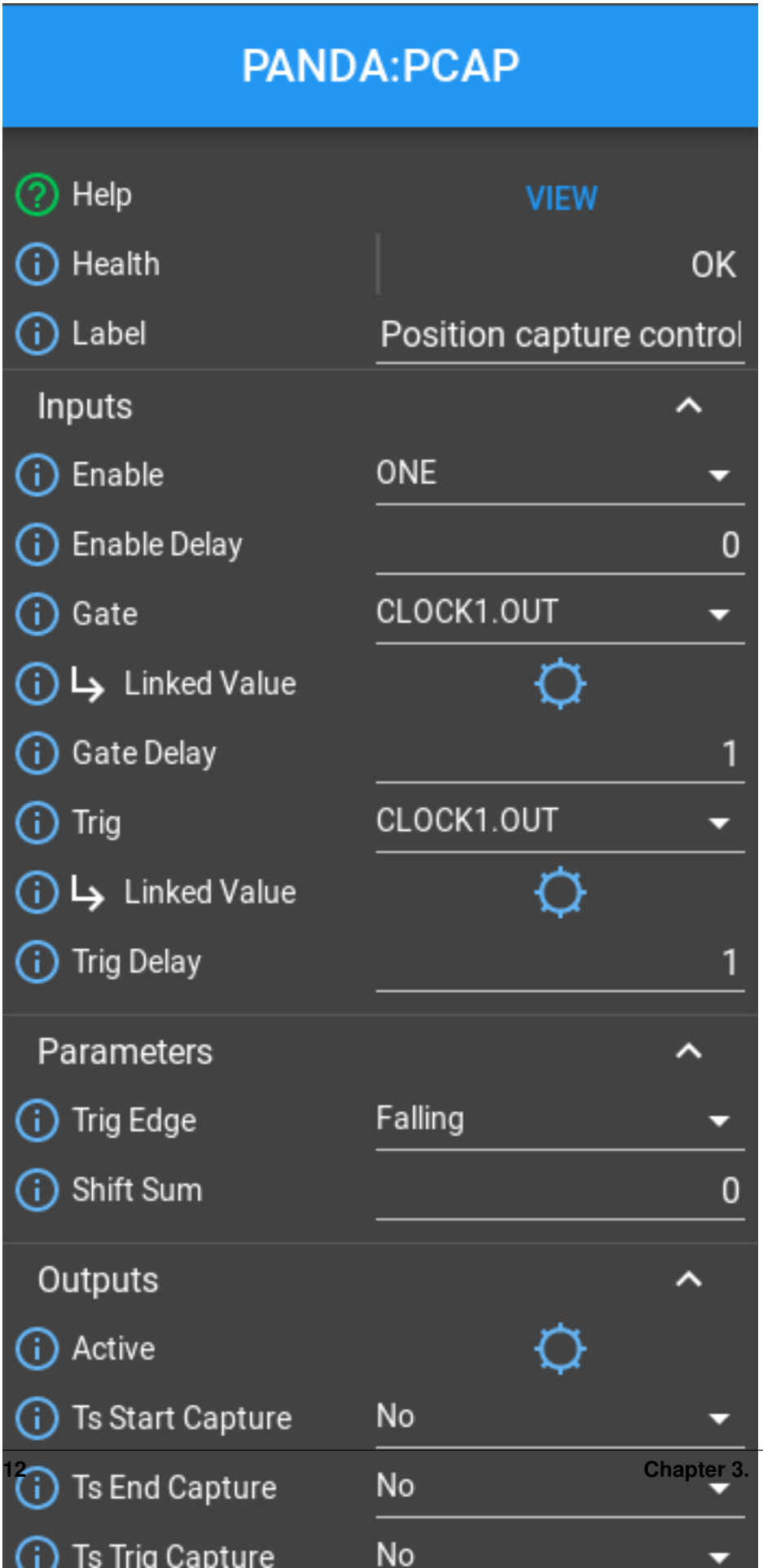
We start off with both CLOCK Blocks set to a period of 1s, so each second the COUNTER will increment by one, followed by a PCAP trigger half a second later. This is best viewed as a timing diagram:

What PCAP does on that trigger is determined by the PCAP Block settings, and the contents of the Bits and Positions tables. For Bits you can turn capture (instantaneous at the time of trigger) on and off. For Positions, you have a choice of:



Capture	Description
No	Don't capture
Value	Instantaneous capture at time of trigger
Diff	The difference in the value while gate was high
Sum	The sum of all the samples while gate was high
Min	The smallest value seen while gate was high
Max	The largest value seen while gate was high
Mean	The average value seen while gate was high
Min Max	Capture both Min and Max
Min Max Mean	Capture Min Max and Mean

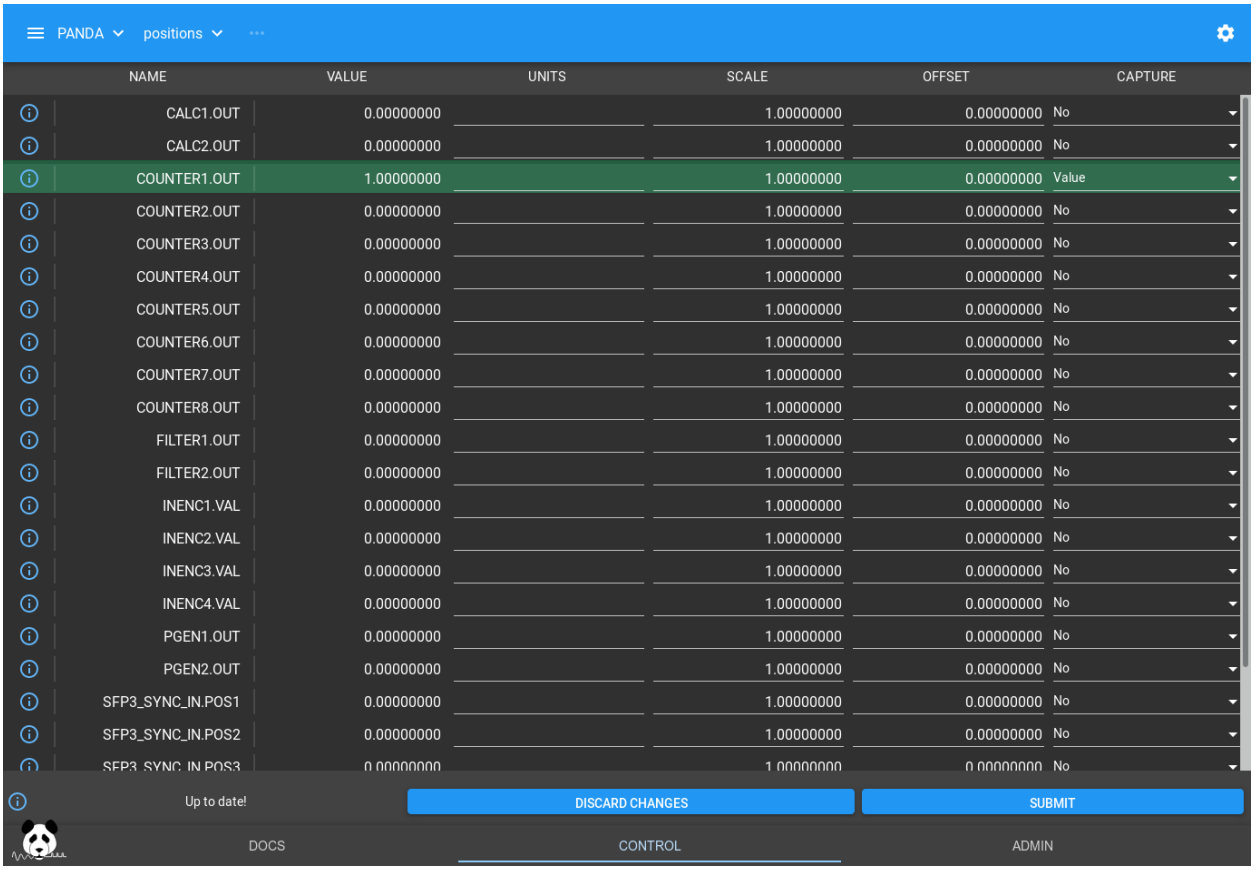
There are also a handful of other fields like the start of frame, end of frame and trigger time that can be captured by setting fields on the PCAP Block. If you click on the PCAP Block you will see them in the Outputs section:


PANDA:PCAP
Help
VIEW
Health
OK
Label
Position capture control
Inputs
Enable
ONE
Enable Delay
0
Gate
CLOCK1.OUT
Linked Value
Gate Delay
1
Trig
CLOCK1.OUT
Linked Value
Trig Delay
1
Parameters
Trig Edge
Falling
Shift Sum
0
Outputs
Active
Ts Start Capture
No
Ts End Capture
No
Ts Trig Capture
No

In the inputs section of the PCAP Block we can see that we have set a delay of 1 for both the Trig and Gate. Delays on bit inputs are in FPGA clock ticks, and are there to compensate for different length data paths that need to be aligned. Each Block and each wire in PandA take 1 clock tick each. In this example, both COUNTER1 and PCAP are being triggered by a CLOCK in the same clock tick, but we want to delay the input to PCAP by one clock tick so that it sees the updated COUNTER1 value *after* the corresponding CLOCK rising edge.

Note: The delay fields of the PCAP Block are also shown as small badges on the input ports of the Block

We can set COUNTER1.OUT to capture the Value at trigger by modifying the Positions table and pressing Submit:



Now we can get a client ready to receive data. We can capture data in ASCII or Binary format as detailed in the TCP server documentation, and TANGO and EPICS have clients to do this. For this tutorial we will just use the ASCII format on the commandline to check:

```
$ nc <panda-ip> 8889
```

Here we could specify binary output and header format, but we'll just stick with the default ASCII output (the default). Press Return again, and we will see:

```
OK
```

Now go back to the PandA layout and select the PCAP Block, pressing the ARM button. The Active light will go on and data will start streaming in the terminal window until Disarm is pressed:

```
missed: 0
process: Scaled
```

(continues on next page)

(continued from previous page)

```
format: ASCII
fields:
 COUNTER1.OUT double Value scale: 1 offset: 0 units:

 1
 2
 3
 4
END 4 Disarmed
```

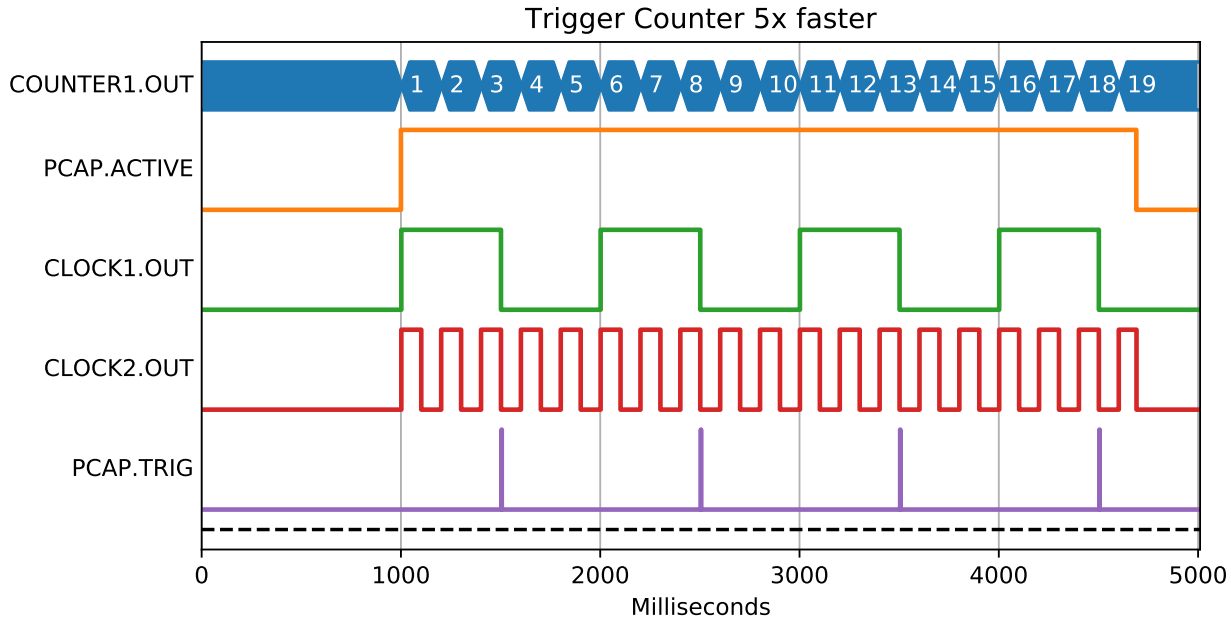
This tallies with the timing diagram we saw above, the captured value matches the instantaneous value of COUNTER1.OUT when PCAP.TRIG went high.

We will now make the COUNTER1.OUT increment 5 times faster. Set CLOCK2.PERIOD to 0.2s, and click PCAP.ARM and you will see the captured value change:

```
missed: 0
process: Scaled
format: ASCII
fields:
 COUNTER1.OUT double Value scale: 1 offset: 0 units:

 3
 8
 13
 18
END 4 Disarmed
```

If we look at the timing plot, we can see this also matched what we expect, the value is captured mid way through each increment of 5:



Now let's investigate the other options. If we change the Positions table so COUNTER1.OUT captures the Diff instead of Value then we will see it captures the difference between the value at the rising edge of the gate, and the falling edge:

```
missed: 0
process: Scaled
format: ASCII
fields:
 COUNTER1.OUT double Diff scale: 1 offset: 0 units:

 2
 2
 2
 2
END 4 Disarmed
```

This again matches the timing plot, GATE rises when COUNTER was at 1, and falls at 3, then rises at 6 and falls at 8.

Note: If we hadn't put in the 1 clock tick delays for Gate and Trig then we would see 3 rather than 2, as GATE would rise at 0 and fall at 3, then rise at 5 and fall at 8

This capture output is generally used with COUNTER Blocks connected to an input fed from a V2F to capture the total counts produced in a given time window.

If we change COUNTER1.OUT to capture Min Max and Mean, we will see the other options:

```
missed: 0
process: Scaled
format: ASCII
fields:
 COUNTER1.OUT double Min scale: 1 offset: 0 units:
 COUNTER1.OUT double Max scale: 1 offset: 0 units:
 COUNTER1.OUT double Mean scale: 1 offset: 0 units:

 1 3 1.8
 6 8 6.8
 11 13 11.8
 16 18 16.8
END 4 Disarmed
```

Here we can see our min and max values as we expected, and also the Mean of the COUNTER value during the total gate:

```
# (sum of counter_value * time_at_value) / gate_time = mean
(1 * 0.2 + 2 * 0.2 + 3 * 0.1) / 0.5 = 1.8
(6 * 0.2 + 7 * 0.2 + 8 * 0.1) / 0.5 = 6.8
```

This capture output is generally used with encoders, to give the min, max and mean value of the encoder over a detector frame.

3.3 Conclusion

This tutorial has shown how to use the Position Capture interface of a PandA to capture entries on the position bus, and introduced the different capture types. It has also introduced the COUNTER block that is useful connecting to the pulse train produced by a V2F. In the next tutorial we will read about how to use position compare to generate triggers from position outputs, and how to configure position capture to work with it.

CHAPTER 4

Position Compare Tutorial

This tutorial will introduce you to the concept of Position Compare. It will show a one dimensional scan of an encoder, how to create trigger pulses at regularly spaced positional intervals, and capture time information.

CHAPTER 5

Snake Scan Tutorial

This tutorial will introduce the concept of table based position compare using the SEQ block to do a two dimensional ‘snake’ scan. This is where the X dimension scans forward over the range, Y steps forward, then X scans backwards, repeated until the scan is complete.

CHAPTER 6

Available Blocks

These are the *Block* types that may be built into an *App*. Some are soft blocks, and some are tied to particular hardware, so not all Blocks will be included in every *PandABlocks Device*.

6.1 BITS - Soft inputs and constant bits

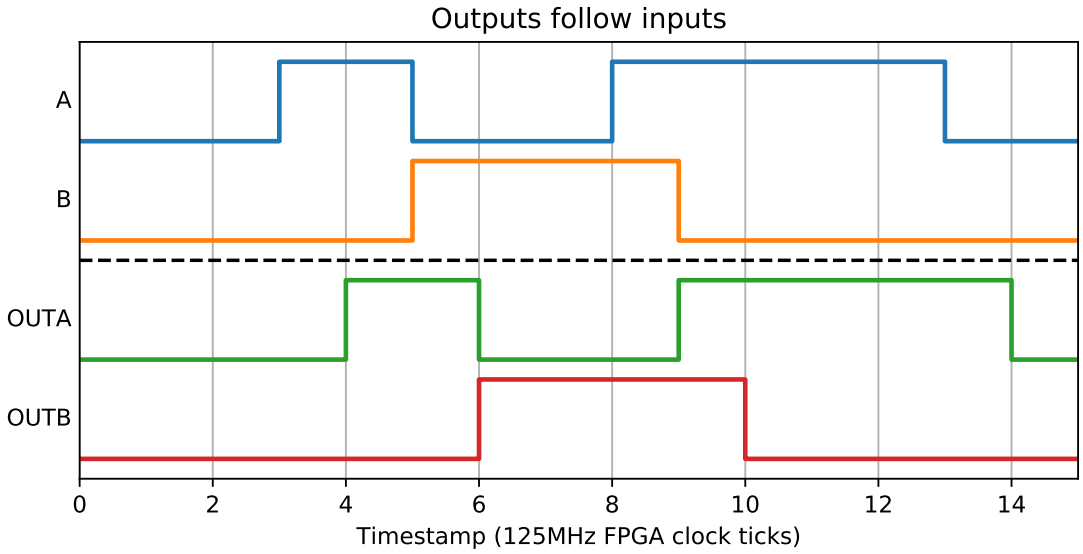
The BITS block contains 4 soft values A..D. Each of these soft values can be set to 0 or 1 by using the SET_A..SET_D parameters.

6.1.1 Fields

Name	Type	Description
A	param bit	The value that output A should take
B	param bit	The value that output B should take
C	param bit	The value that output C should take
D	param bit	The value that output D should take
OUTA	bit_out	The value of A on the bit bus
OUTB	bit_out	The value of B on the bit bus
OUTC	bit_out	The value of C on the bit bus
OUTD	bit_out	The value of D on the bit bus

6.1.2 Outputs follow parameters

This example shows how the values on the bit bus follow the parameter values after a 1 clock tick propagation delay



6.2 CALC - Position Calc

The position calc block has an output which is the sum of the position inputs

6.2.1 Fields

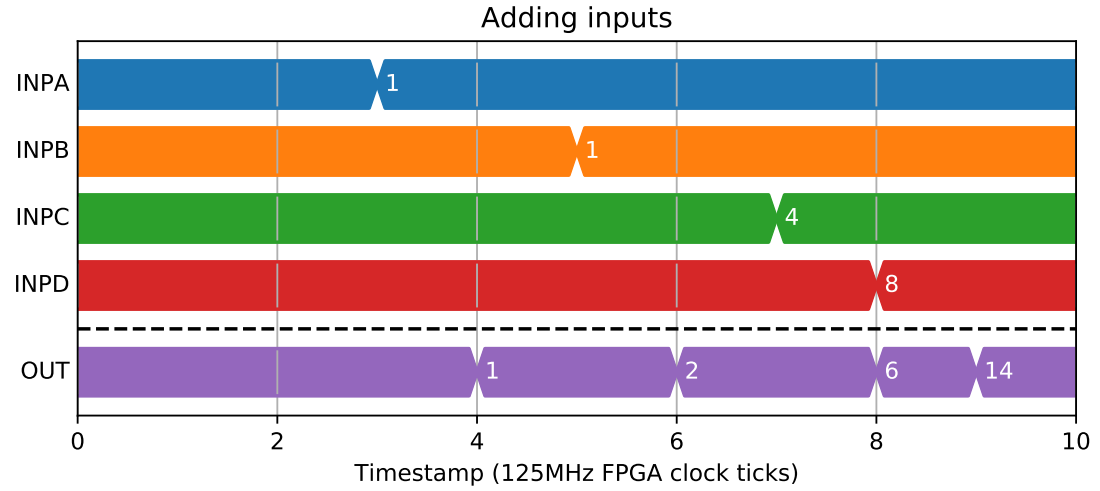
Name	Type	Description
INPA	pos_mux	Position input A
INPB	pos_mux	Position input B
INPC	pos_mux	Position input C
INPD	pos_mux	Position input D
TYPEA	param enum	Source of the value of A for calculation 0 Value 1 -Value
TYPEB	param enum	Source of the value of B for calculation 0 Value 1 -Value
TYPEC	param enum	Source of the value of B for calculation 0 Value 1 -Value
TYPED	param enum	Source of the value of B for calculation 0 Value 1 -Value
FUNC	param enum	Scale divisor after add 0 A+B+C+D
SHIFT	param uint	Number of places to right shift calculation result before output
OUT	pos_out	Position output

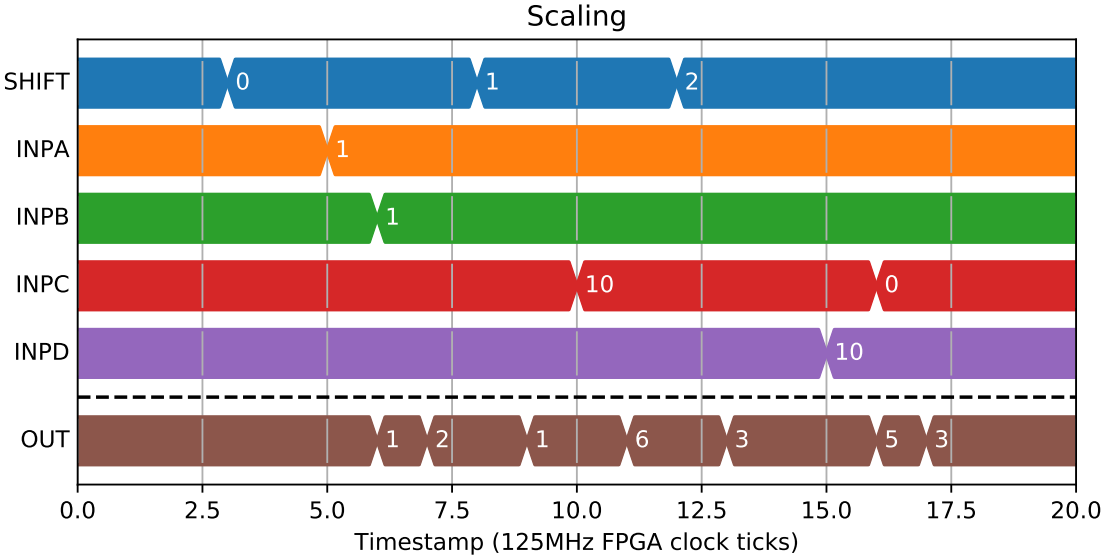
6.2.2 Adding inputs

The output is the sum of the inputs

6.2.3 Scaling

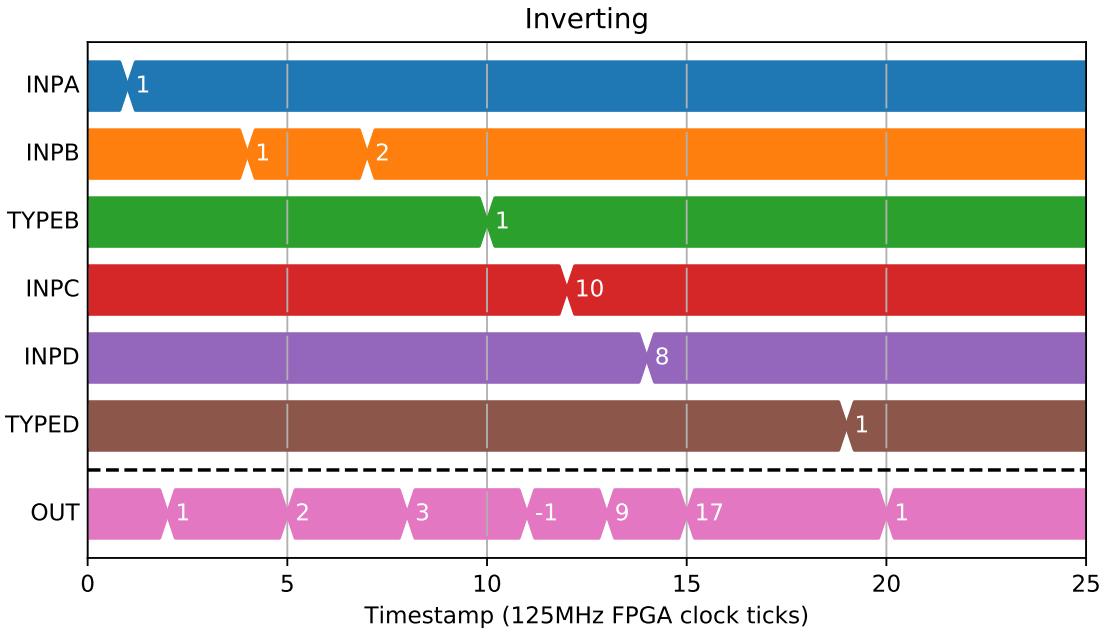
The scale factor is a bit shift and is applied after the sum.


Adding inputs
INPA
INPB
INPC
INPD
OUT
1
1
4
8
1
2
6
14
0
2
4
6
8
10
Timestamp (125MHz FPGA clock ticks)


Scaling
SHIFT
INPA
INPB
INPC
INPD
OUT
0
1
2
1
1
10
0
10
1
2
1
6
3
5
3
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Timestamp (125MHz FPGA clock ticks)

6.2.4 Inverting

Each input can be individually inverted before they are added together



6.3 CLOCK - Configurable clock

The CLOCK block contains a user-settable 50% duty cycle clock.

6.3.1 Fields

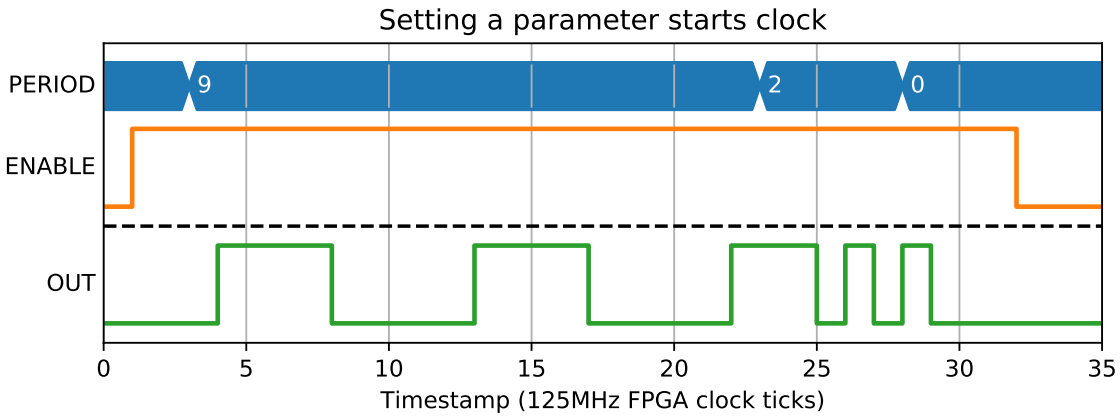
Name	Type	Description
ENABLE	bit_mux	Halt and reset on falling edge, enable on rising
PERIOD	param time	Period of clock output
OUT	bit_out	Clock output

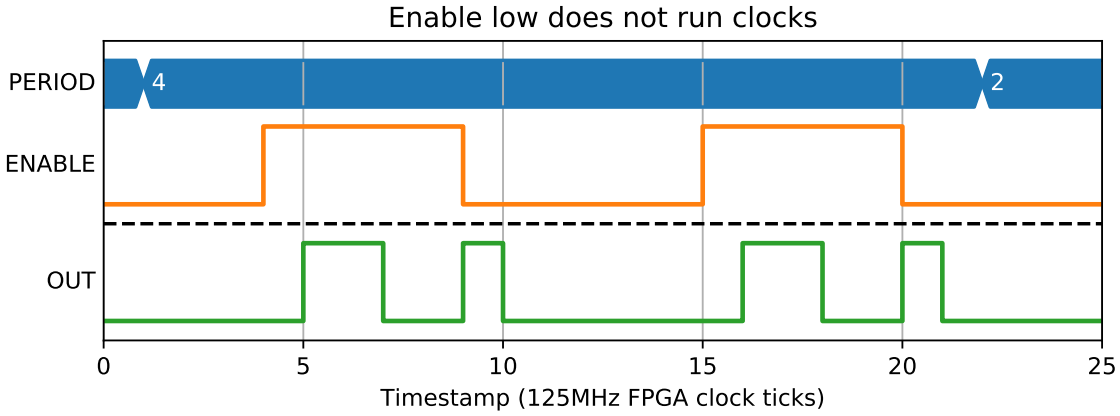
6.3.2 Setting clock period parameters

Each time a clock parameter is set, the clock restarts from that point with the new period value.

6.3.3 Clock settings while disabled

To start the clock synchronously you can set them while the Block is disabled. It will start on rising edge of ENABLE and be zeroed on the falling edge.


Setting a parameter starts clock
PERIOD
ENABLE
OUT
9
2
0
0
5
10
15
20
25
30
35
Timestamp (125MHz FPGA clock ticks)


Enable low does not run clocks
PERIOD
ENABLE
OUT
4
2
0
5
10
15
20
25
Timestamp (125MHz FPGA clock ticks)

6.4 COUNTER - Up/Down pulse counter

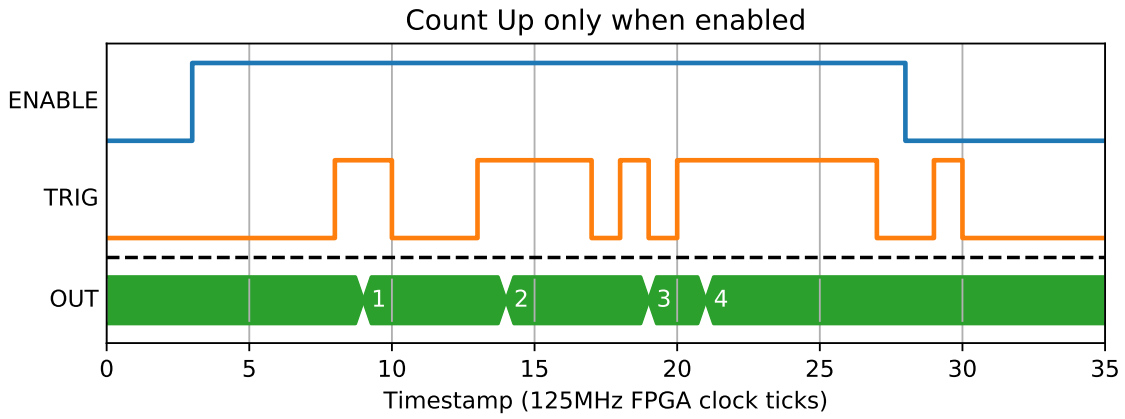
Each counter block, when enabled, can count up/down with user-defined step value on the rising edge on input trigger. The counters can also be initialised to a user-defined START value.

6.4.1 Fields

Name	Type	Description
ENABLE	bit_mux	Halt on falling edge, reset and enable on rising
TRIG	bit_mux	Rising edge ticks the counter up/down by STEP
DIR	bit_mux	Up/Down direction (0 = Up, 1 = Down)
START	param int	Counter start value
STEP	param	Up/Down step value
MAX	param int	Rollover value
MIN	param int	Value to which counter should rollover to
CARRY	bit_out	Internal counter overflow status
OUT	pos_out	Current counter value

6.4.2 Counting pulses

The most common use of a counter block is when you would like to track the number of rising edges received while enabled:



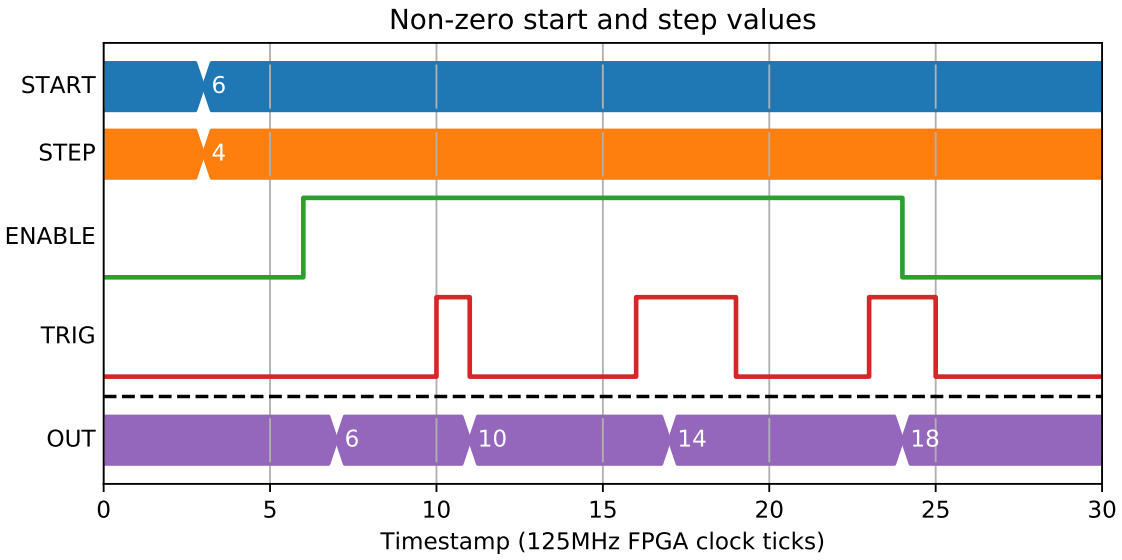
You can also set the start value to be loaded on enable, and step up by a number other than one:

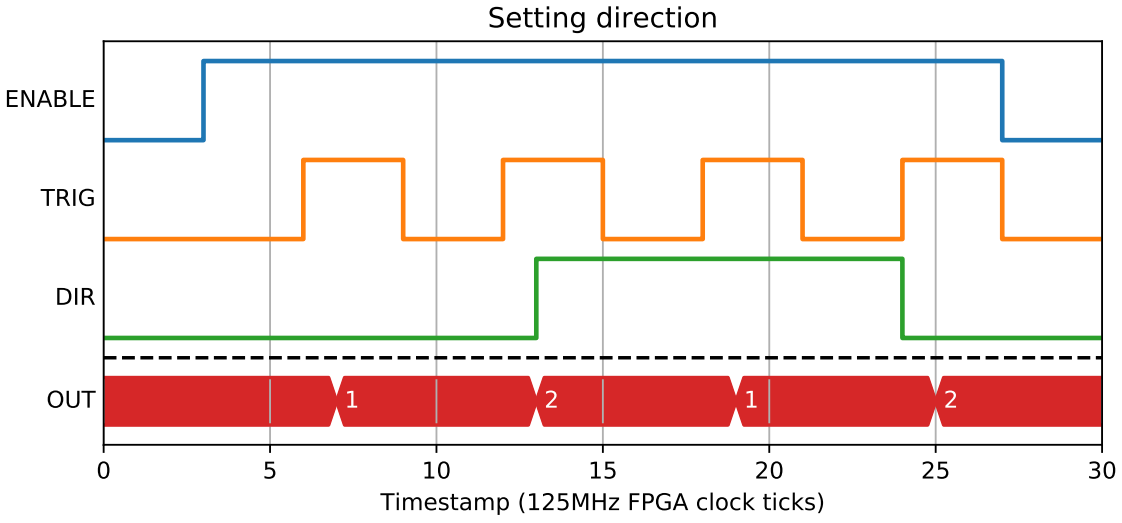
You can also set the direction that a pulse should apply step, so it becomes an up/down counter. The direction is sampled on the same clock tick as the pulse rising edge:

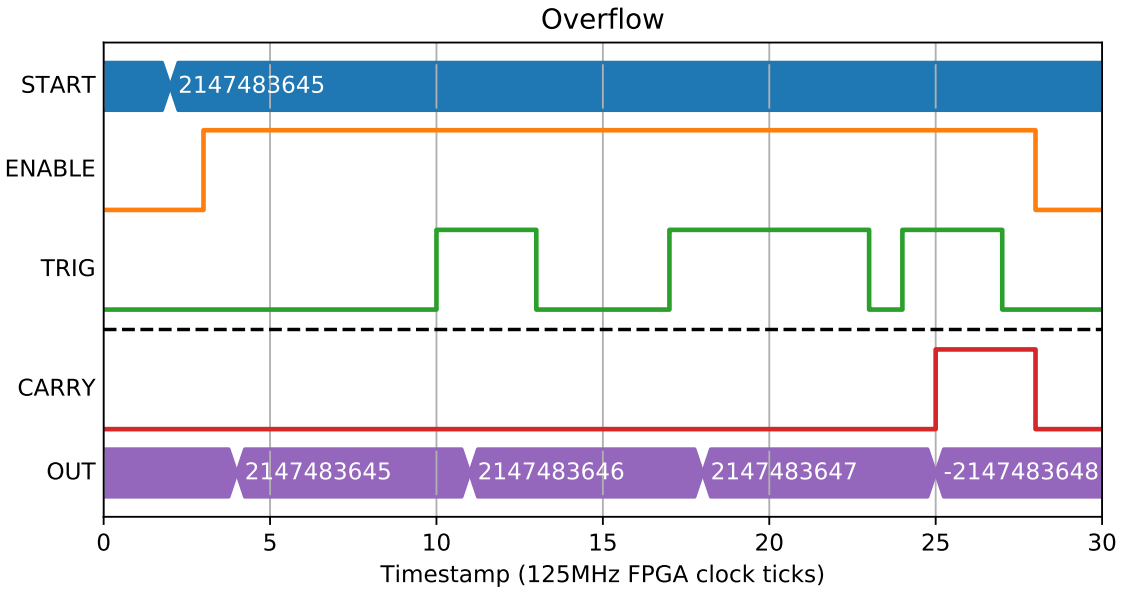
6.4.3 Rollover

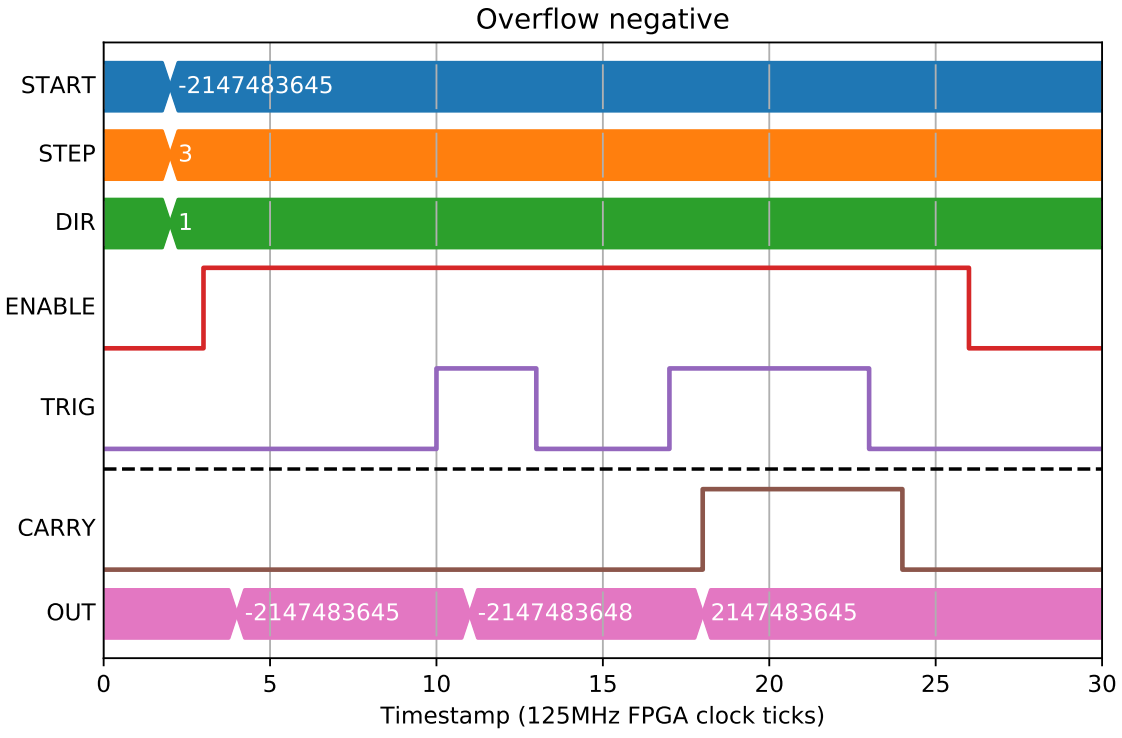
If the count goes higher than the max value for an int32 (2147483647) the CARRY output gets set high and the counter rolls. The CARRY output stays high for as long as the trigger input stays high.

A similar thing happens for a negative overflow:


Non-zero start and step values
START
STEP
ENABLE
TRIG
OUT
6
4
6
10
14
18
0
5
10
15
20
25
30
Timestamp (125MHz FPGA clock ticks)

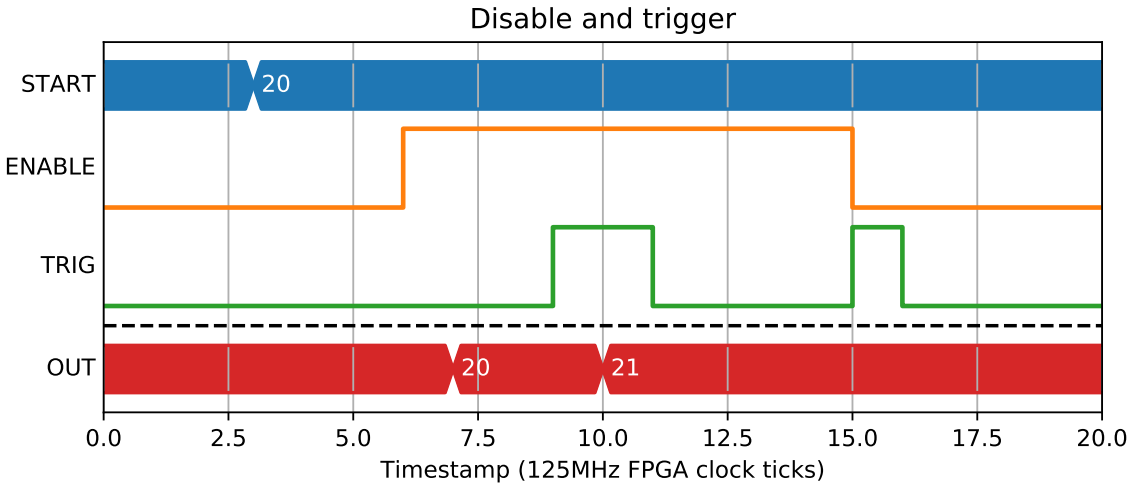

Setting direction
ENABLE
TRIG
DIR
OUT
1
2
1
2
0
5
10
15
20
25
30
Timestamp (125MHz FPGA clock ticks)


Overflow
START
2147483645
ENABLE
TRIG
CARRY
OUT
2147483645
2147483646
2147483647
-2147483648
0
5
10
15
20
25
30
Timestamp (125MHz FPGA clock ticks)

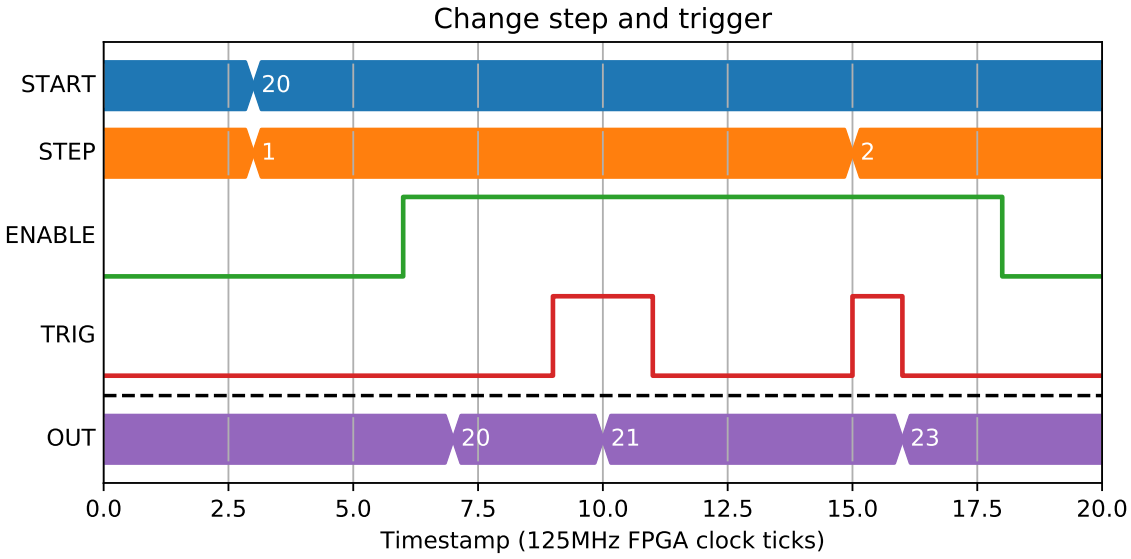

Overflow negative
START
-2147483645
STEP
3
DIR
1
ENABLE
TRIG
CARRY
OUT
-2147483645
-2147483648
2147483645
0
5
10
15
20
25
30
Timestamp (125MHz FPGA clock ticks)

6.4.4 Edge cases

If the Enable input goes low at the same time as a trigger, there will be no output value on the next clock tick.



If the step size is changed at the same time as a trigger input rising edge, the output value for that trigger will be the new step size.



6.5 DIV - Pulse divider

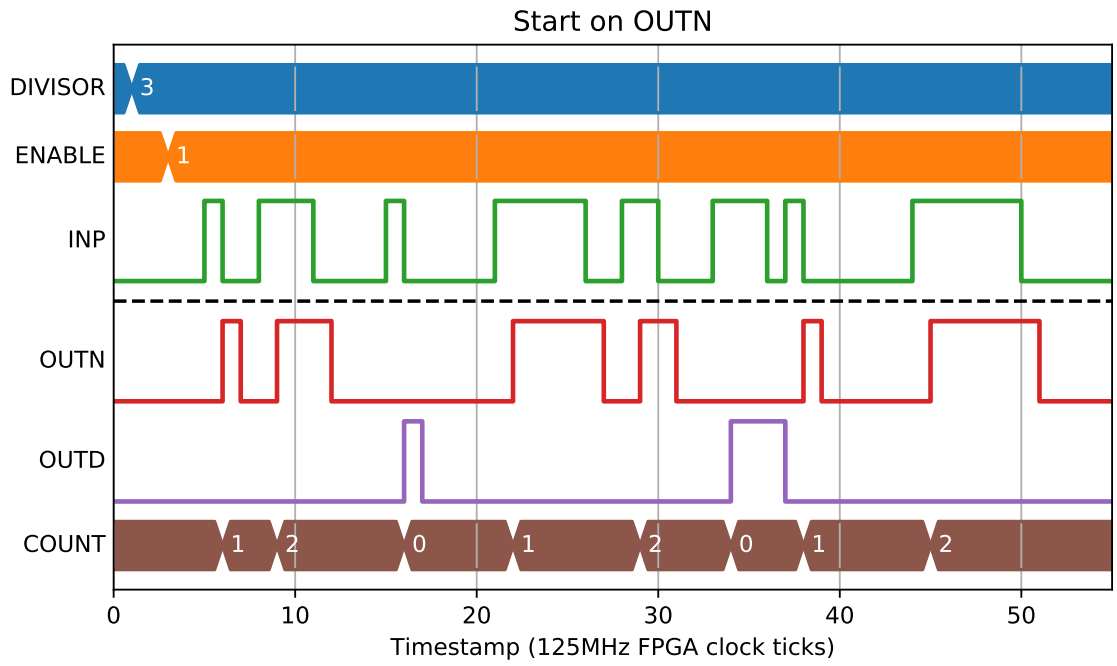
A DIV block is a 32-bit pulse divider that can divide a pulse train between two outputs. It has an internal counter that counts from 0 to DIVISOR-1. On each rising edge of INP, if counter = DIVISOR-1, then it is set to 0 and the pulse is sent to OUTD, otherwise it is sent to OUTN. Change in any parameter causes the block to be reset.

6.5.1 Fields

Name	Type	Description
ENABLE	bit_mux	Reset on falling edge, enable on rising
INP	bit_mux	Input pulse train
DIVISOR	param	Divisor value
FIRST_PULSE	param enum	Where to send first pulse 0 OutN 1 OutD
OUTD	bit_out	Divided pulse output
OUTN	bit_out	Non-divided pulse output
COUNT	read	Internal counter value

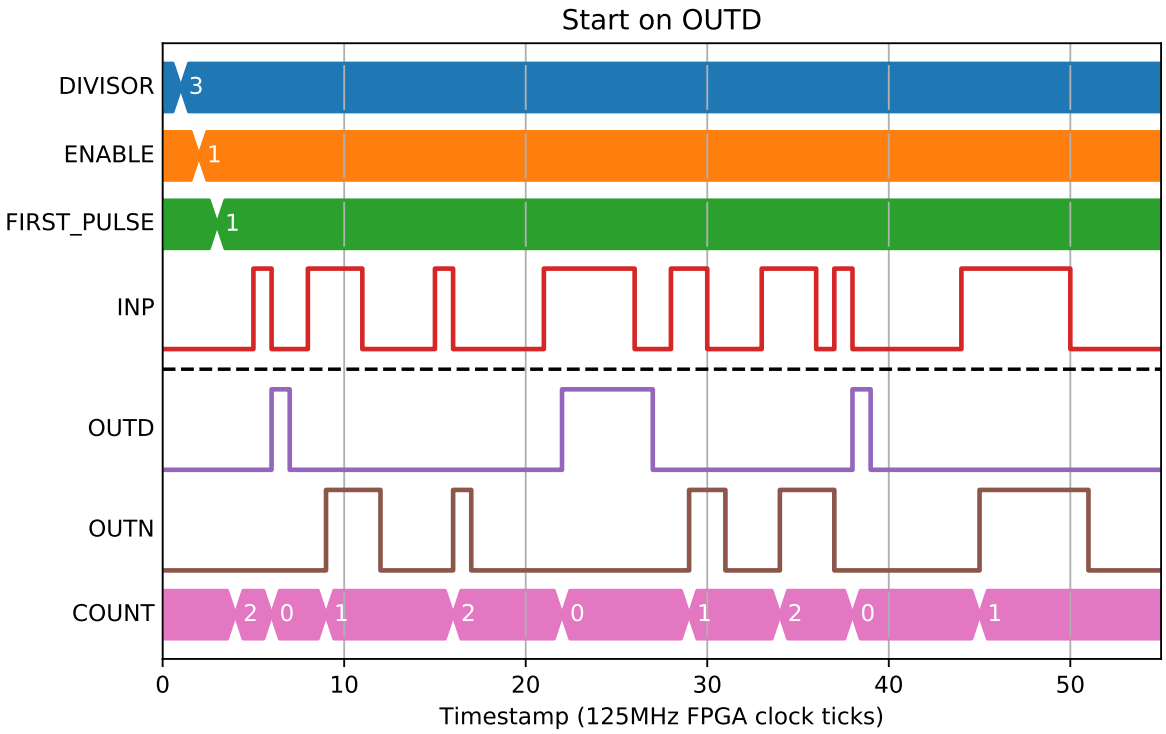
6.5.2 Which output do pulses go to

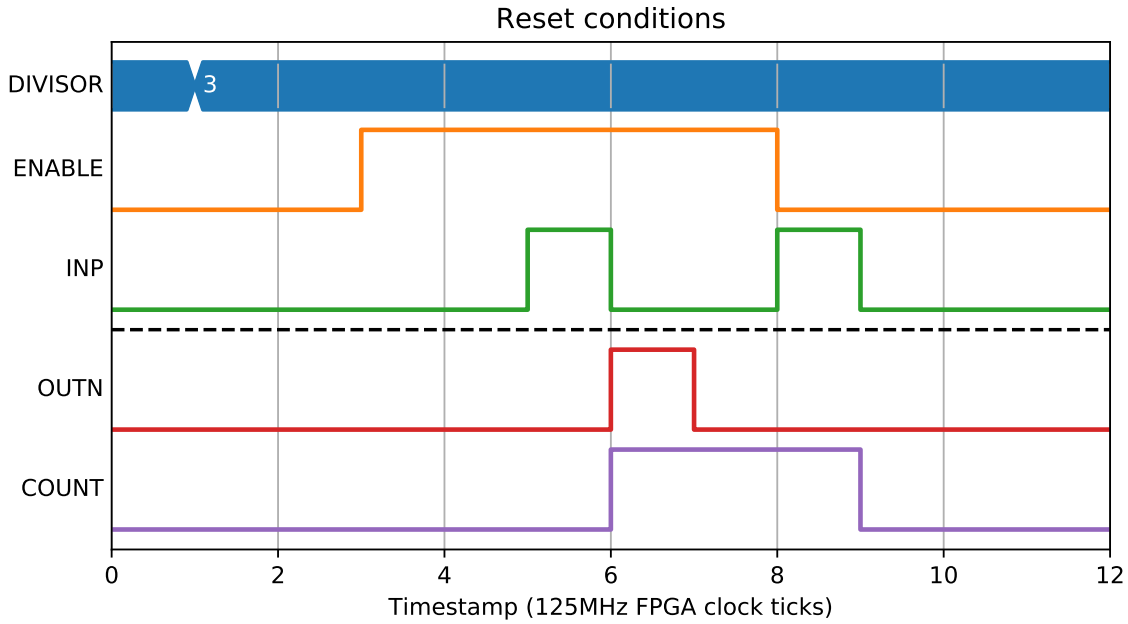
With a DIVISOR of 3, the block will send 1 of 3 INP pulses to OUTD and 2 of 3 INP pulses to OUTN. The following two examples illustrate how the FIRST_PULSE parameter controls the initial value of OUT, which controls whether OUTD or OUTN gets the next pulse.



6.5.3 Reset conditions

If an ENABLE falling edge is received at the same time as an INP rising edge, the input signal is ignored and the block reset.


Start on OUTD
DIVISOR
ENABLE
FIRST_PULSE
INP
OUTD
OUTN
COUNT
3
1
1
2
0
1
2
0
1
2
0
1
0
10
20
30
40
50
Timestamp (125MHz FPGA clock ticks)


Reset conditions
DIVISOR
ENABLE
INP
OUTN
COUNT
3
0
2
4
6
8
10
12
Timestamp (125MHz FPGA clock ticks)

6.6 FILTER - Filter

The filter block has two different modes of operation: Difference and Average. They both work by latching the values on the input and performing an operation comparing to the current value.

6.6.1 Fields

Name	Type	Description
ENABLE	bit_mux	Enable event
TRIG	bit_mux	Trigger event
INP	pos_mux	Input data
MODE	param enum	Select operation mode 0 difference 1 average
READY	bit_out	Output Ready
OUT	pos_out	Output data
HEALTH	read enum	Error 0 OK 1 Accumulator overflow 2 Divider retrigger

6.6.2 Difference

The difference operation works by latching the value on the input on the rising edge of the Enable signal. On a rising edge of the trigger signal the output is given as the the current input value minus the latched value.

After the operation, the latched value is updated to be the current value on the input.

The operation continues to work if the current value is less than the latched value: a negative result is outputted

6.6.3 Average

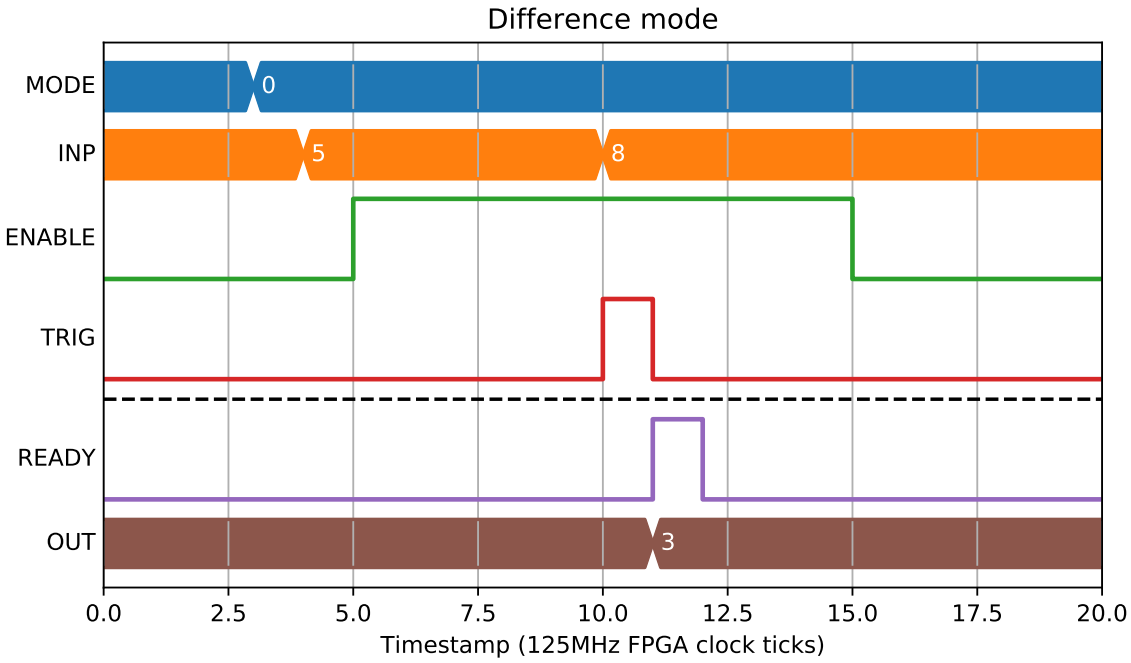
The average function appends a sum value on each clock pulse. When a trigger signal is received it divides the summed value by the number of clock pulses that have passed.

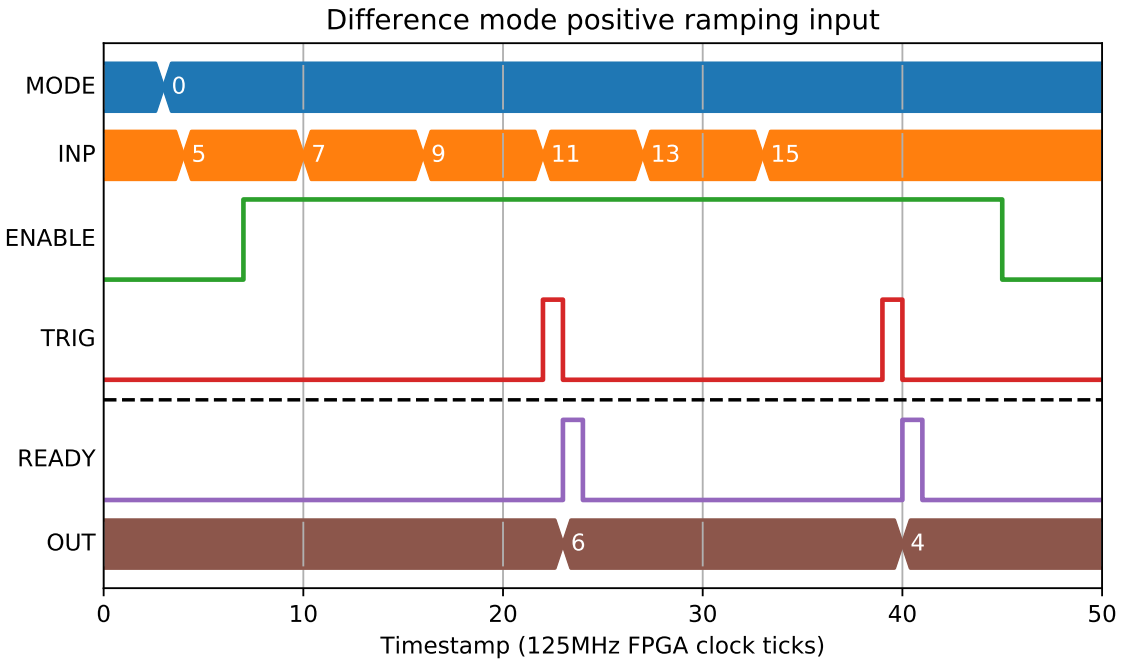
If a calculation is triggered before the calculation is ready, the system will show an error on the HEALTH output and will then need to be re-enabled before another calculation can be sent.

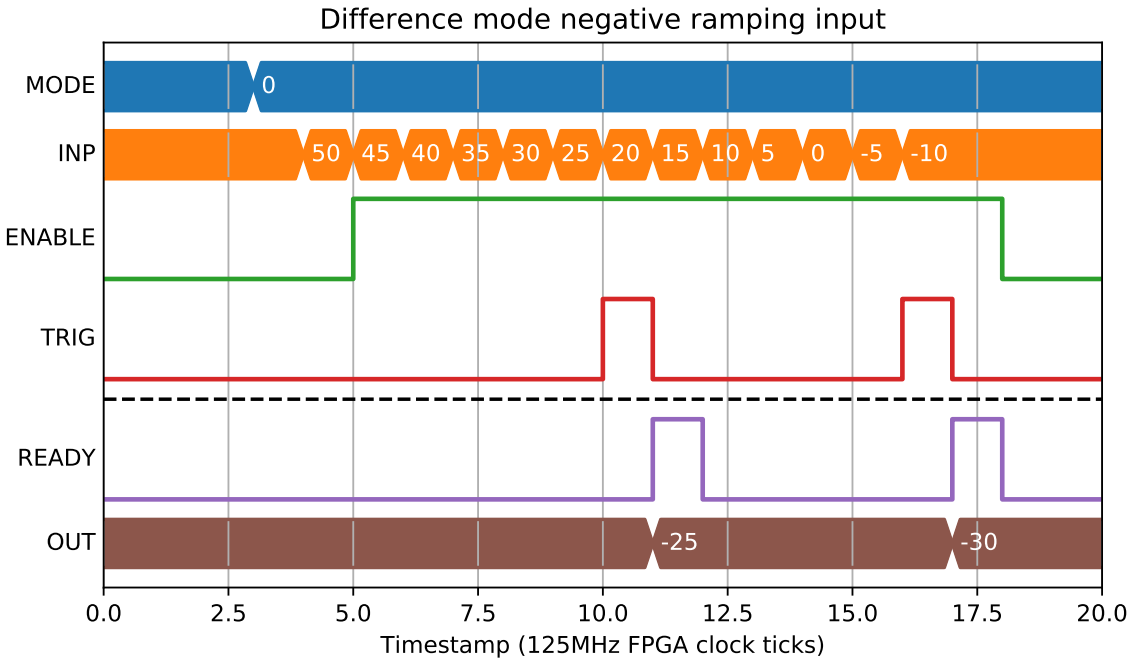
6.7 FMC_24V - FMC 24V IO Module

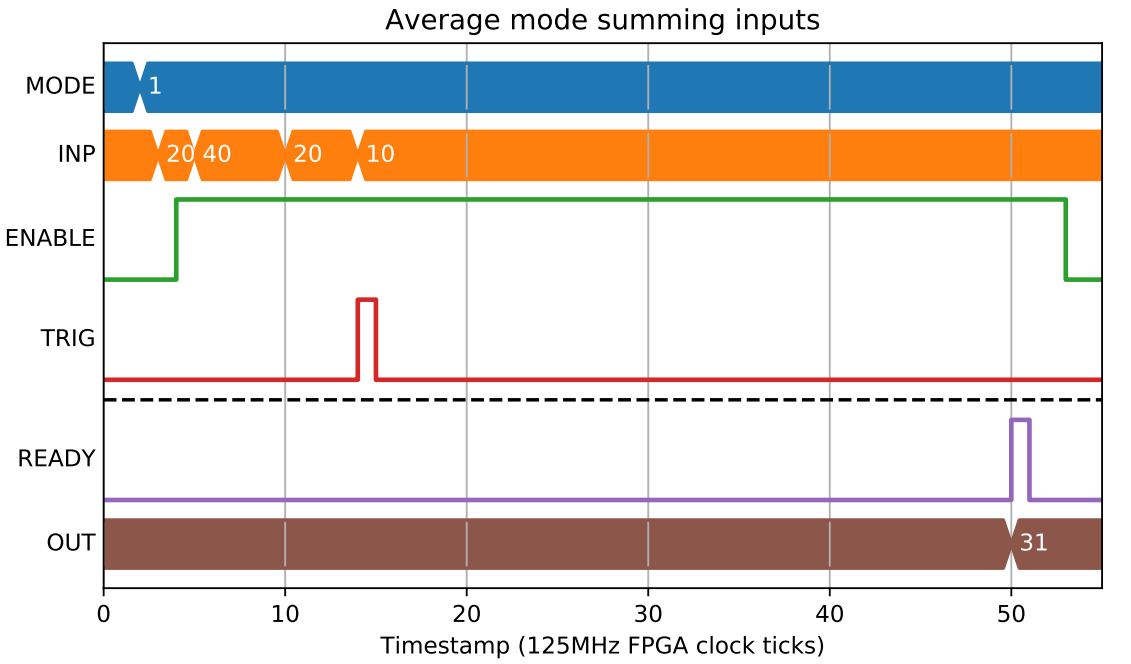
6.7.1 Fields

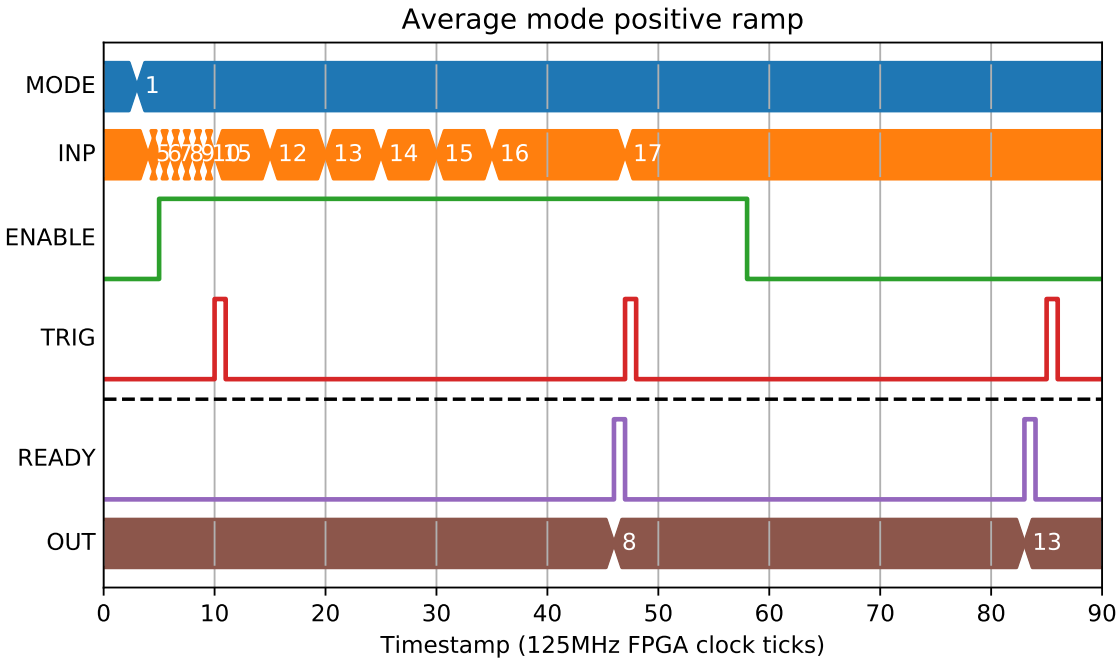
The module has bee splits into two blocks: The inputs and the outputs.

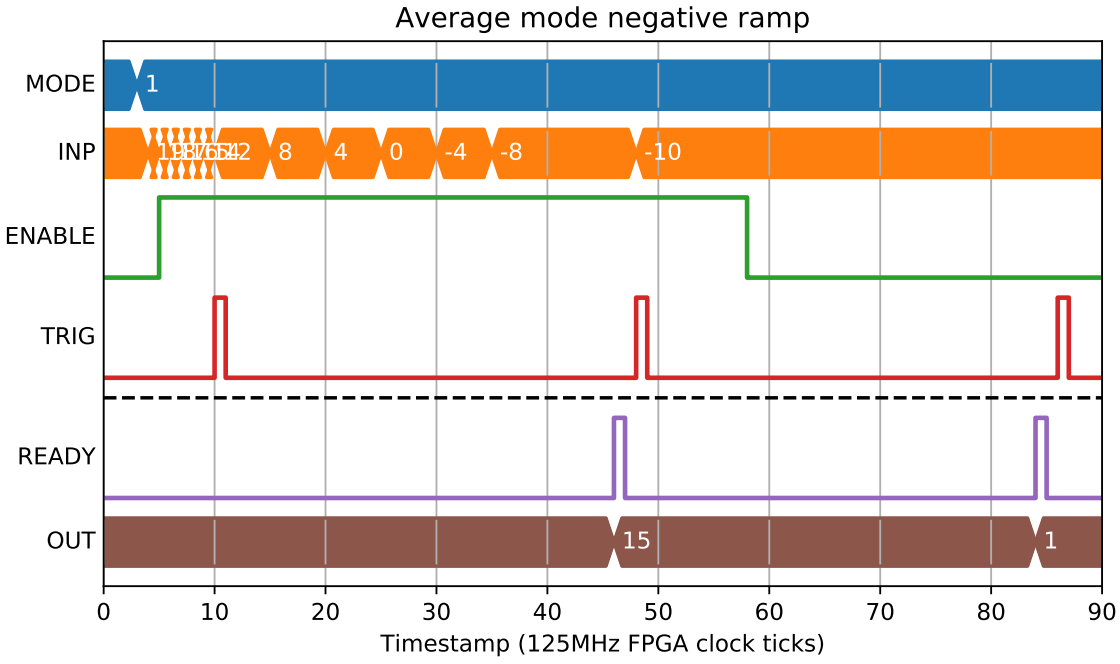

Difference mode
MODE
INP
ENABLE
TRIG
READY
OUT
0
5
8
3
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Timestamp (125MHz FPGA clock ticks)

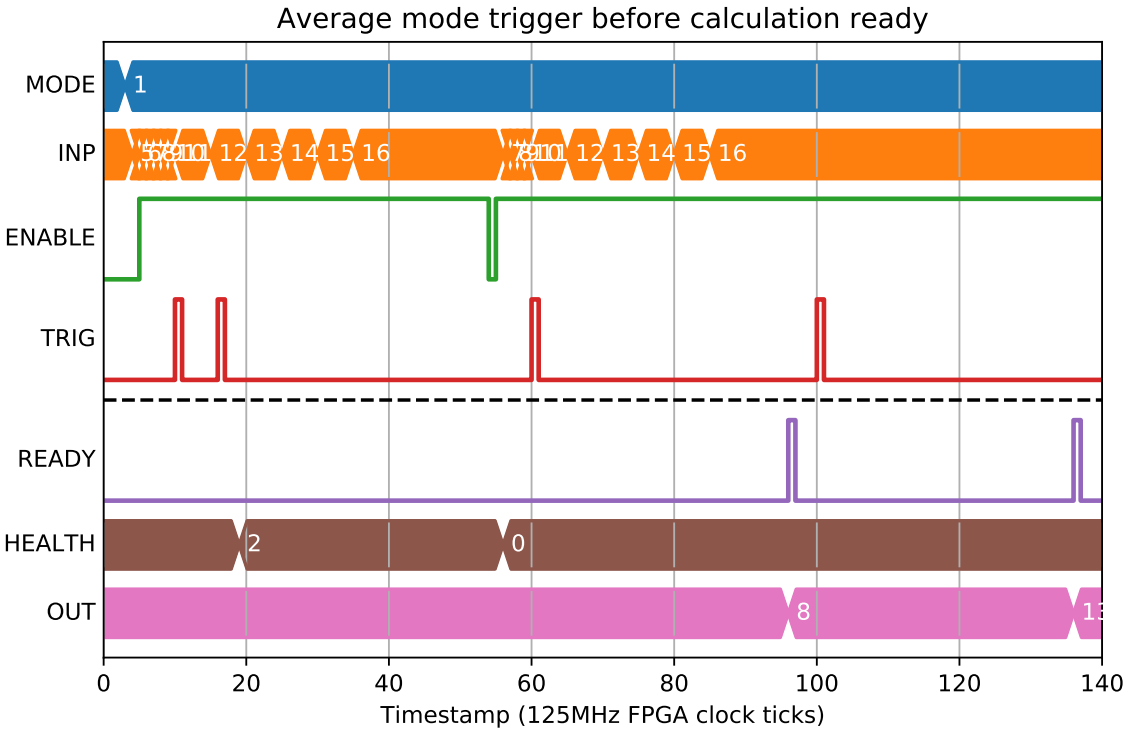

Difference mode positive ramping input
MODE
INP
ENABLE
TRIG
READY
OUT
0
5
7
9
11
13
15
6
4
0
10
20
30
40
50
Timestamp (125MHz FPGA clock ticks)

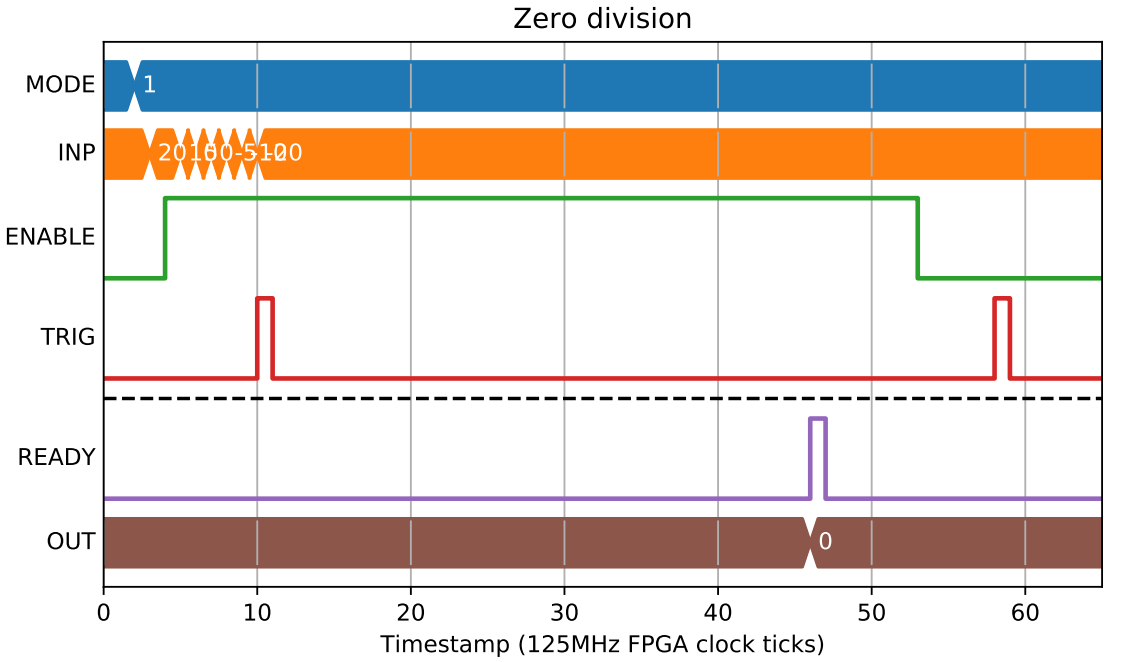

Difference mode negative ramping input
MODE
INP
ENABLE
TRIG
READY
OUT
0
50
45
40
35
30
25
20
15
10
5
0
-5
-10
-25
-30
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Timestamp (125MHz FPGA clock ticks)


Average mode summing inputs
MODE
INP
ENABLE
TRIG
READY
OUT
1
20
40
20
10
31
0
10
20
30
40
50
Timestamp (125MHz FPGA clock ticks)


Average mode positive ramp
MODE
INP
ENABLE
TRIG
READY
OUT
1
12
13
14
15
16
17
8
13
0
10
20
30
40
50
60
70
80
90
Timestamp (125MHz FPGA clock ticks)


Average mode negative ramp
MODE
INP
ENABLE
TRIG
READY
OUT
1
8
4
0
-4
-8
-10
15
1
0
10
20
30
40
50
60
70
80
90
Timestamp (125MHz FPGA clock ticks)


Average mode trigger before calculation ready
MODE
INP
ENABLE
TRIG
READY
HEALTH
OUT
1
2
0
8
0
20
40
60
80
100
120
140
Timestamp (125MHz FPGA clock ticks)


Zero division
MODE
INP
ENABLE
TRIG
READY
OUT
1
0
0
10
20
30
40
50
60
Timestamp (125MHz FPGA clock ticks)

Name	Type	Description
IN.FMC_PRSNT	read	FMC present
IN.VTSEL	param enum	Input Voltage Select 0 5V 1 24V
IN.DB	param enum	Input Debounce Time Select 0 None 1 0.024ms 2 0.75ms 3 3ms
IN.VAL1	bit_out	24V Input-1
IN.VAL2	bit_out	24V Input-2
IN.VAL3	bit_out	24V Input-3
IN.VAL4	bit_out	24V Input-4
IN.VAL5	bit_out	24V Input-5
IN.VAL6	bit_out	24V Input-6
IN.VAL7	bit_out	24V Input-7
IN.VAL8	bit_out	24V Input-8
IN.FAULT	read	Input Voltage and Temp Alarm (Active Low)
OUT.FMC_PRSNT	read	FMC present
OUT.VAL1	bit_mux	24V Output-1
OUT.VAL2	bit_mux	24V Output-2
OUT.VAL3	bit_mux	24V Output-3
OUT.VAL4	bit_mux	24V Output-4
OUT.VAL5	bit_mux	24V Output-5
OUT.VAL6	bit_mux	24V Output-6
OUT.VAL7	bit_mux	24V Output-7
OUT.VAL8	bit_mux	24V Output-8
OUT.PWR_ON	param enum	Enable Output Power 0 Off 1 On
OUT.PUSHPL	param enum	Output Global Push-Pull/High-Side Select 0 High-side 1 Push-pull
OUT.FLTR	param enum	Output Glitch Filter Enable 0 Off 1 On
OUT.SRIAL	param enum	Output Serial/Parallel Select 0 Parallel 1 Serial

6.8 FMC_ACQ427 - FMC ACQ427 Module

6.8.1 Fields

The module has been split into two blocks: the inputs, which controls the ADC; and the outputs, which control the DAC.

Name	Type	Description
IN.GAIN1	param enum	ADC input gain 0 10V 1 5V 2 2.5V 3 1.25V
IN.GAIN2	param enum	ADC input gain 0 10V 1 5V 2 2.5V 3 1.25V
IN.GAIN3	param enum	ADC input gain 0 10V 1 5V 2 2.5V 3 1.25V
IN.GAIN4	param enum	ADC input gain 0 10V 1 5V 2 2.5V 3 1.25V
IN.GAIN5	param enum	ADC input gain 0 10V 1 5V 2 2.5V 3 1.25V
IN.GAIN6	param enum	ADC input gain 0 10V 1 5V 2 2.5V 3 1.25V
IN.GAIN7	param enum	ADC input gain 0 10V 1 5V 2 2.5V 3 1.25V
IN.GAIN8	param enum	

6.8.2 Clock note

The ADC runs at 1MHz

6.9 FMC_ACQ430 - FMC ACQ430 Module

6.9.1 Fields

Name	Type	Description
VAL1	pos_out	ADC Channel 1 Data
VAL2	pos_out	ADC Channel 2 Data
VAL3	pos_out	ADC Channel 3 Data
VAL4	pos_out	ADC Channel 4 Data
VAL5	pos_out	ADC Channel 5 Data
VAL6	pos_out	ADC Channel 6 Data
VAL7	pos_out	ADC Channel 7 Data
VAL8	pos_out	ADC Channel 8 Data
TTL	bit_out	C/T 5V TTL input

6.9.2 Clock note

The ADC runs in High Res mode using a divisor of 5 from the main clock frequency.

125 Mhz/5/512 gives and ADC sample rate of 48.828125 kHz

6.10 FMC_LOOPBACK - FMC Loopback Module

6.10.1 Fields

Name	Type	Description
SOFT_RESET	write action	GTX Soft Reset
LOOP_PERIOD	param	Loopback toggle period for IO
FMC_PRSNT	read	FMC present
LINK_UP	read	GTX link status
ERROR_COUNT	read	GTX loopback
LA_P_ERROR	read	LA_P loopback status
LA_N_ERROR	read	LA_N loopback status
GTREFCLK	read	GT Ref clock freq
FMC_CLK0	read	FMC CLK0 clock freq
FMC_CLK1	read	FMC CLK1 clock freq
EXT_CLK	read	External clock freq
FMC_MAC_LO	read	MAC low in integer value bit 23:0
FMC_MAC_HI	read	MAC high in integer value bit 47:24

6.11 INENC - Input encoder

The INENC block handles the encoder input signals

6.11.1 Fields

Name	Type	Description
CLK	bit_mux	Clock output to slave encoder
PROTOCOL	param enum	Type of absolute/incremental protocol 0 Quadrature 1 SSI 2 BISS 3 enDat
ENCODING	param enum	Position encoding (for absolute encoders) 0 Unsigned Binary 1 Unsigned Gray 2 Signed Binary 3 Signed Gray
CLK_SRC	param enum	Bypass/Pass Through encoder signals 0 Internally Generated 1 From CLK
CLK_PERIOD	param time	Clock rate
FRAME_PERIOD	param time	Frame rate
BITS	param uint 63	Number of bits
LSB_DISCARD	param uint 31	Number of LSB bits to discard
MSB_DISCARD	param uint 31	Number of MSB bits to discard
SETP	write int	Set point
RST_ON_Z	param bit	Zero position on Z rising edge
A	bit_out	Quadrature A if in incremental mode
B	bit_out	Quadrature B if in incremental mode
Z	bit_out	Z index channel if in incremental mode
DATA	bit_out	Data input from slave encoder
CONN	bit_out	Signal detected
HOMED	read bit	Quadrature homed status
HEALTH	read enum	Table status 0 OK 1 Linkup error (=not CONN) 2 Timeout error (for BISS, monitor SSI) 3 CRC error (for BISS) 4 Error bit active (for BISS) 5 ENDAT not implemented
VAL	pos_out	Current position
DCARD_TYPE	read enum	

6.12 LUT - 5 Input lookup table

An LUT block produces an output that is determined by a user-programmable 5-input logic function, set with the FUNC register.

6.12.1 Fields

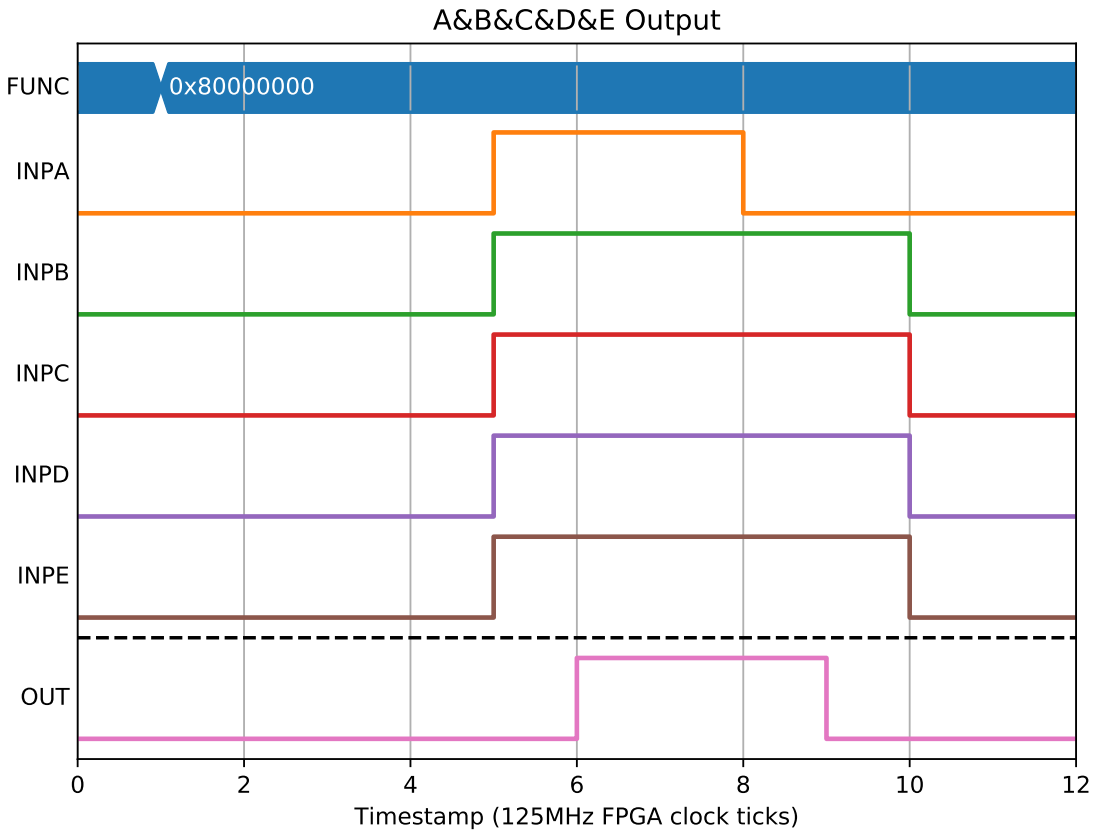
Name	Type	Description
INPA	bit_mux	Input A
INPB	bit_mux	Input B
INPC	bit_mux	Input C
INPD	bit_mux	Input D
INPE	bit_mux	Input E
TYPEA	param enum	Source of the value of A for calculation 0 Input-Level 1 Pulse-On-Rising-Edge 2 Pulse-On-Falling-Edge 3 Pulse-On-Either-Edge
TYPEB	param enum	Source of the value of B for calculation 0 Input-Level 1 Pulse-On-Rising-Edge 2 Pulse-On-Falling-Edge 3 Pulse-On-Either-Edge
TYPEC	param enum	Source of the value of C for calculation 0 Input-Level 1 Pulse-On-Rising-Edge 2 Pulse-On-Falling-Edge 3 Pulse-On-Either-Edge
TYPED	param enum	Source of the value of D for calculation 0 Input-Level 1 Pulse-On-Rising-Edge 2 Pulse-On-Falling-Edge 3 Pulse-On-Either-Edge
TYPEE	param enum	Source of the value of E for calculation 0 Input-Level 1 Pulse-On-Rising-Edge 2 Pulse-On-Falling-Edge 3 Pulse-On-Either-Edge
FUNC	param lut	Input func
OUT	bit_out	Lookup table output

6.12.2 Testing Function Output

This set of tests sets the function value and checks whether the output is as expected

The value of FUNC is a 32-bit unsigned int representing the truth table output of the 5 inputs. The mapping of the string to an integer is done by the PandABlocks TCP server.

A&B&C&D&E (FUNC= 0x80000000). Setting all inputs to 1 results in an output of 1, and changing any inputs produces an output of 0



~A&~B&~C&~D&~E (FUNC= 0x00000001). Setting all inputs to 0 results in an output of 1, and changing any inputs produces an output of 0

A (FUNC= 0xffff0000). The output should only be 1 if A is 1 irrespective of any other input.

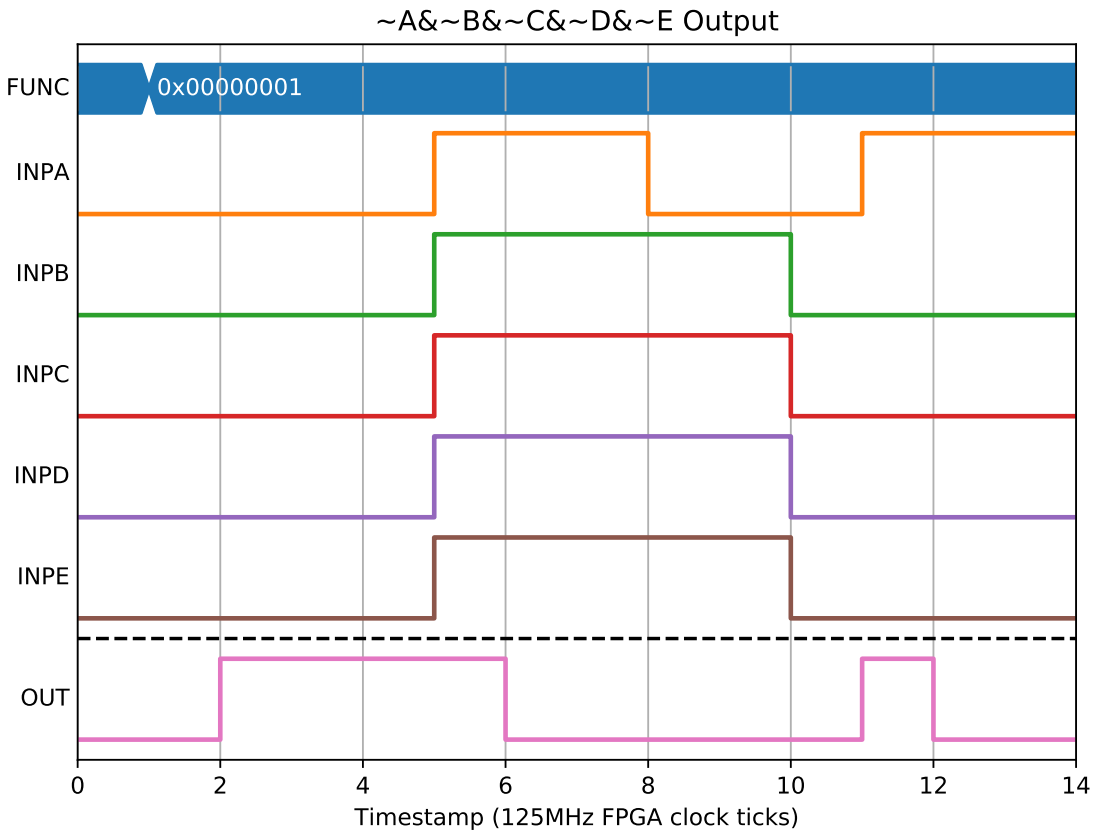
A&B|C&~D (FUNC= 0xff303030)

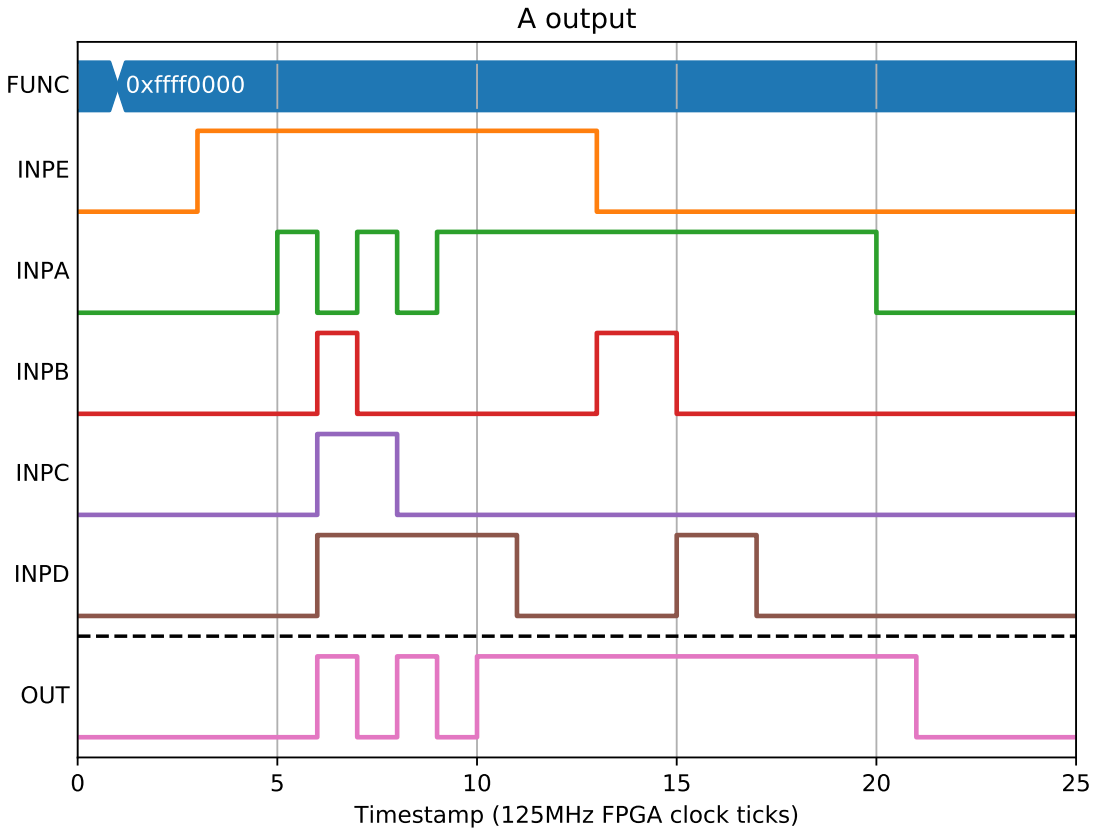
6.12.3 Changing the function in a test

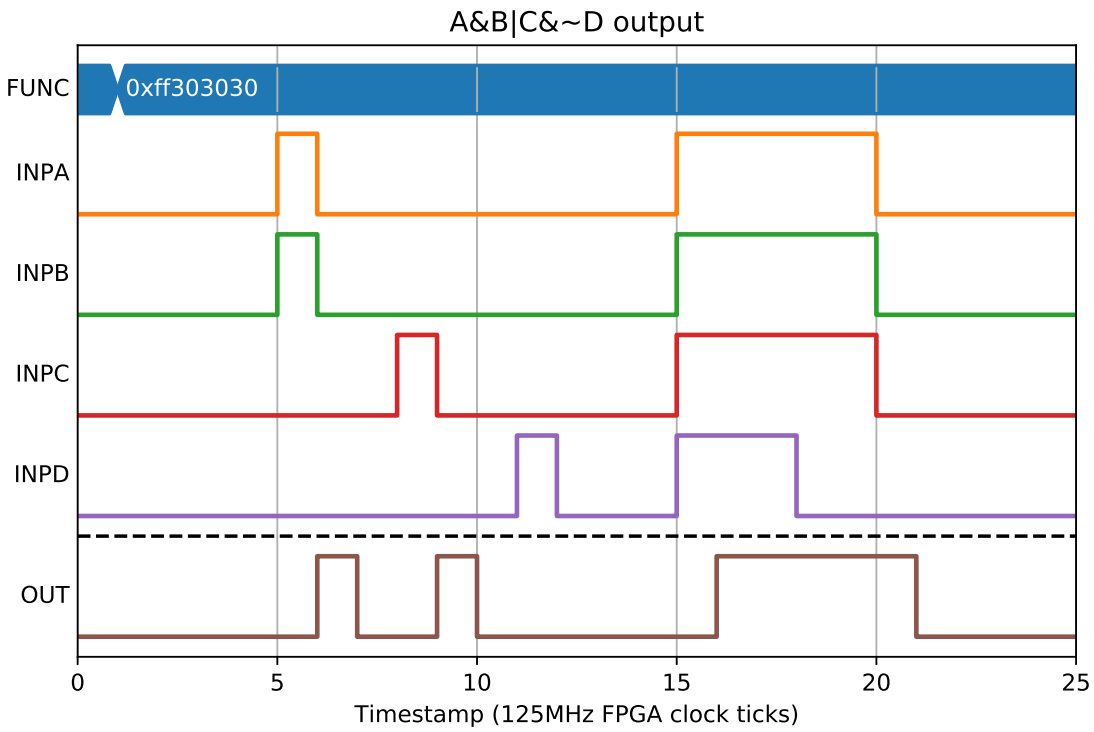
If a function is changed, the output will take effect on the next clock tick

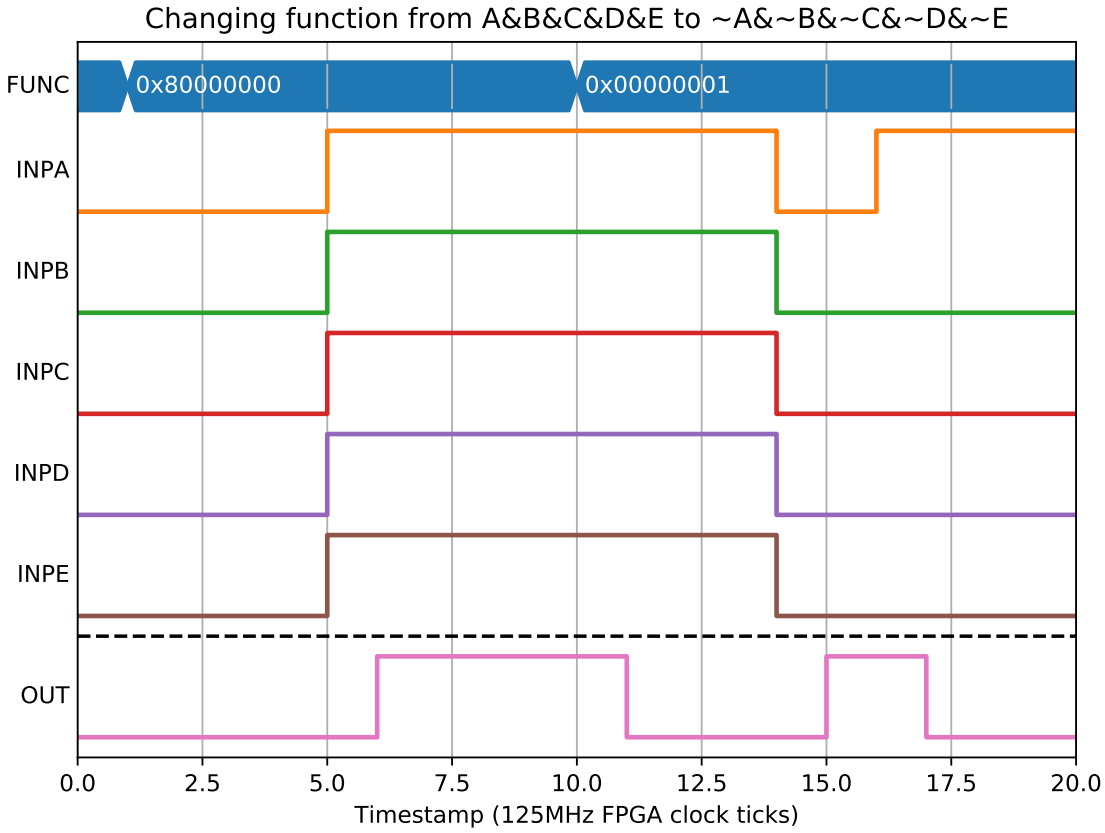
6.12.4 Edge triggered inputs

We can also use the LUT to convert edges into levels by changing A..E to be one clock tick wide pulses based on edges rather than the current level of INPA..INPE.

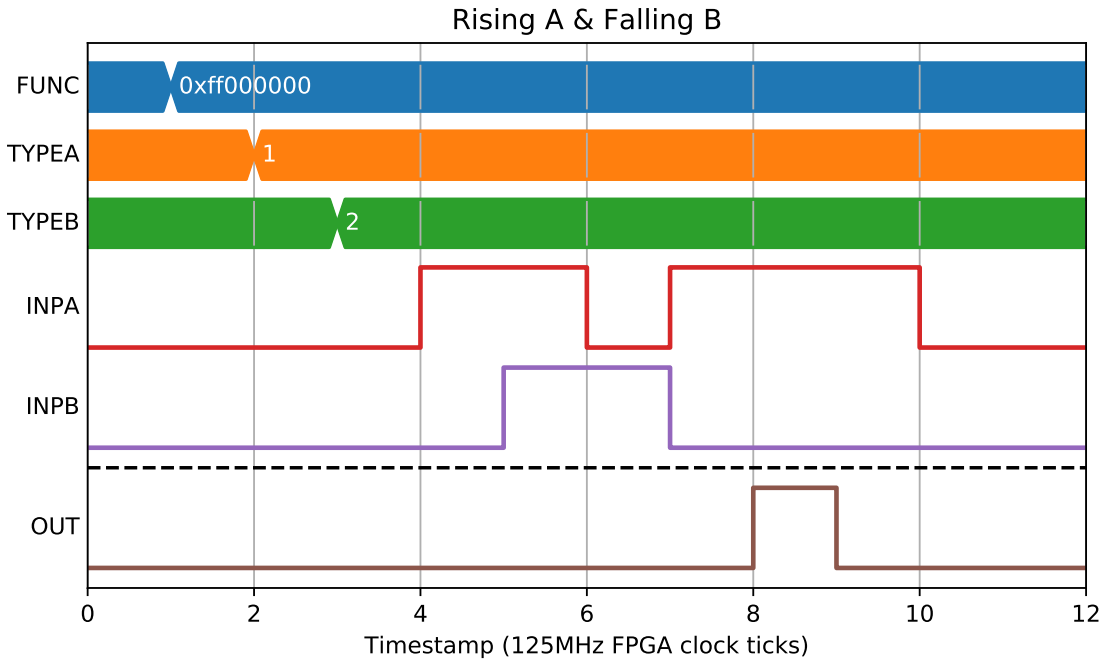

~A&~B&~C&~D&~E Output
FUNC
0x00000001
INPA
INPB
INPC
INPD
INPE
OUT
0
2
4
6
8
10
12
14
Timestamp (125MHz FPGA clock ticks)


A output
FUNC
0xffff0000
INPE
INPA
INPB
INPC
INPD
OUT
0
5
10
15
20
25
Timestamp (125MHz FPGA clock ticks)

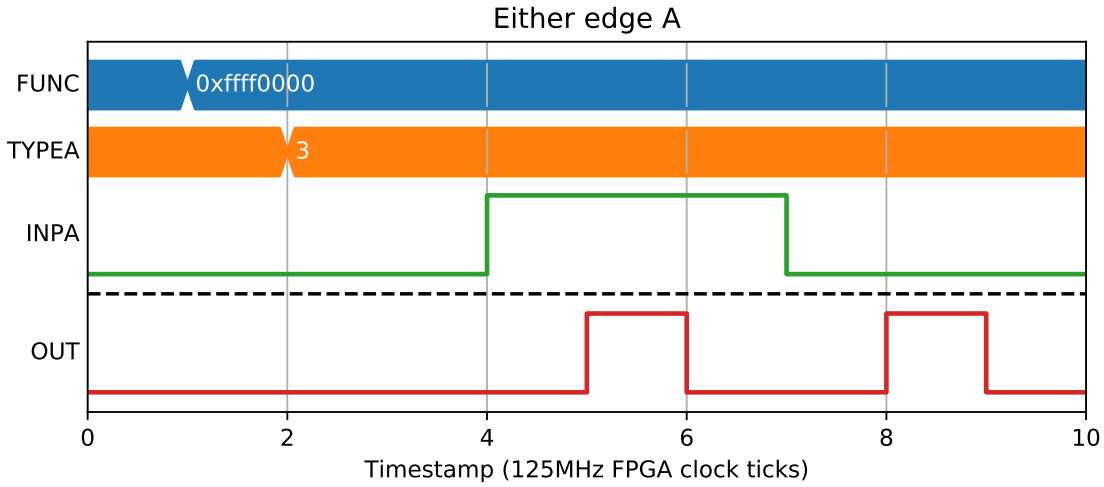

A&B|C&~D output
FUNC
0xff303030
INPA
INPB
INPC
INPD
OUT
0
5
10
15
20
25
Timestamp (125MHz FPGA clock ticks)


Changing function from A&B&C&D&E to ~A&~B&~C&~D&~E
FUNC
0x80000000
0x00000001
INPA
INPB
INPC
INPD
INPE
OUT
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Timestamp (125MHz FPGA clock ticks)

If we wanted to produce a pulse only if INPA had a rising edge on the same clock tick as INPB had a falling edge we could set FUNC=0xff000000 (A&B) and A=1 (rising edge of INPA) and B=2 (falling edge of INPB):



We could also use this for generating pulses on every transition of A:



6.13 LVDSIN - LVDS Input

The LVDSIN block handles the signals from the LVDS Input connectors

6.13.1 Fields

Name	Type	Description
VAL	bit_out	LVDS input value

6.14 LVDSOUT - LVDS Output

The LVDSOUT block handles the signals to the LVDS Output connectors

6.14.1 Fields

Name	Type	Description
VAL	bit_mux	LVDS output value

6.15 OUTENC - Output encoder

The OUTENC block handles the encoder output signals

6.15.1 Fields

Name	Type	Description
ENABLE	bit_mux	Halt of falling edge, reset and enable on rising
GENERATOR_ERROR	param enum	generate error on output 0 No 1 BISS frame error bit
A	bit_mux	Input for A (only straight through)
B	bit_mux	Input for B (only straight through)
Z	bit_mux	Input for Z (only straight through)
DATA	bit_mux	Data output to master encoder
PROTOCOL	param enum	Type of absolute/incremental protocol 0 Quadrature 1 SSI 2 BISS 3 enDat 4 ABZ Passthrough 5 DATA Passthrough
ENCODING	param enum	Position encoding (for absolute encoders) 0 Unsigned Binary 1 Unsigned Gray 2 Signed Binary 3 Signed Gray
BITS	param uint 32	Number of bits
QPERIOD	param time	Quadrature prescaler
CLK	bit_out	Clock input from master encoder
VAL	pos_mux	Input for position (all other protocols)
HEALTH	read enum	Table status 0 OK 1 Biss timeout error (did not received right number of sck for biss frame) 2 ENDAT not implemented
DCARD_TYPE	read enum	Daughter card jumper mode 0 DCARD id 0 1 Encoder Control 2 DCARD id 2 3 Encoder Monitor 4 DCARD id 3 5 DCARD id 4

6.16 PCAP - Position Capture

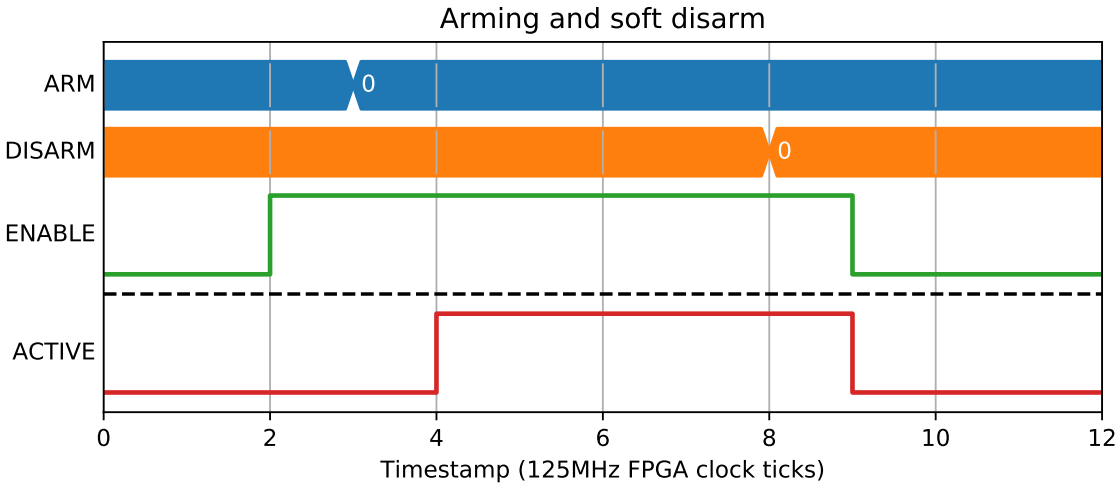
Position capture has the capability to capture anything that is happening on the pos_bus or bit_bus. It listens to ENABLE, GATE and CAPTURE signals, and can capture the value at capture, sum, min and max.

6.16.1 Fields

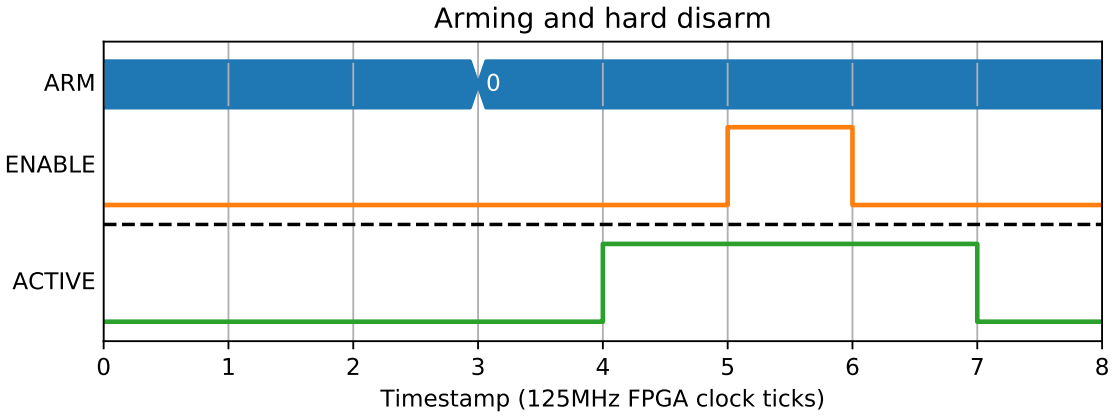
Name	Type	Description
ENABLE	bit_mux	After arm, when high start capture, when low disarm
GATE	bit_mux	After enable, only process gated values if high
TRIG	bit_mux	On selected edge capture current value and gated data
TRIG_EDGE	param enum	Which edge of capture input signal triggers capture 0 Rising 1 Falling 2 Either
SHIFT_SUM	param uint 8	Shift sum/samples data, use if > 2**32 samples required in sum/average
ACTIVE	bit_out	Data capture in progress
TS_START	ext_out timestamp	Timestamp of first gate high in current capture relative to enable
TS_END	ext_out timestamp	Timestamp of last gate high +1 in current capture relative to enable
TS_TRIG	ext_out timestamp	Timestamp of capture event relative to enable
SAMPLES	ext_out samples	Number of gated samples in the current capture
BITS0	ext_out bits 0	Quadrant 0 of bit_bus
BITS1	ext_out bits 1	Quadrant 1 of bit_bus
BITS2	ext_out bits 2	Quadrant 2 of bit_bus
BITS3	ext_out bits 3	Quadrant 3 of bit_bus
HEALTH	read enum	Was last capture successful? 0 OK 1 Capture events too close together 2 Samples overflow

6.16.2 Arming

To start off the block an arm signal is required with a write to `*PCAP.ARM=`. The active signal is raised immediately on ARM, and dropped either on `*PCAP.DISARM`:



Or on the falling edge of ENABLE:



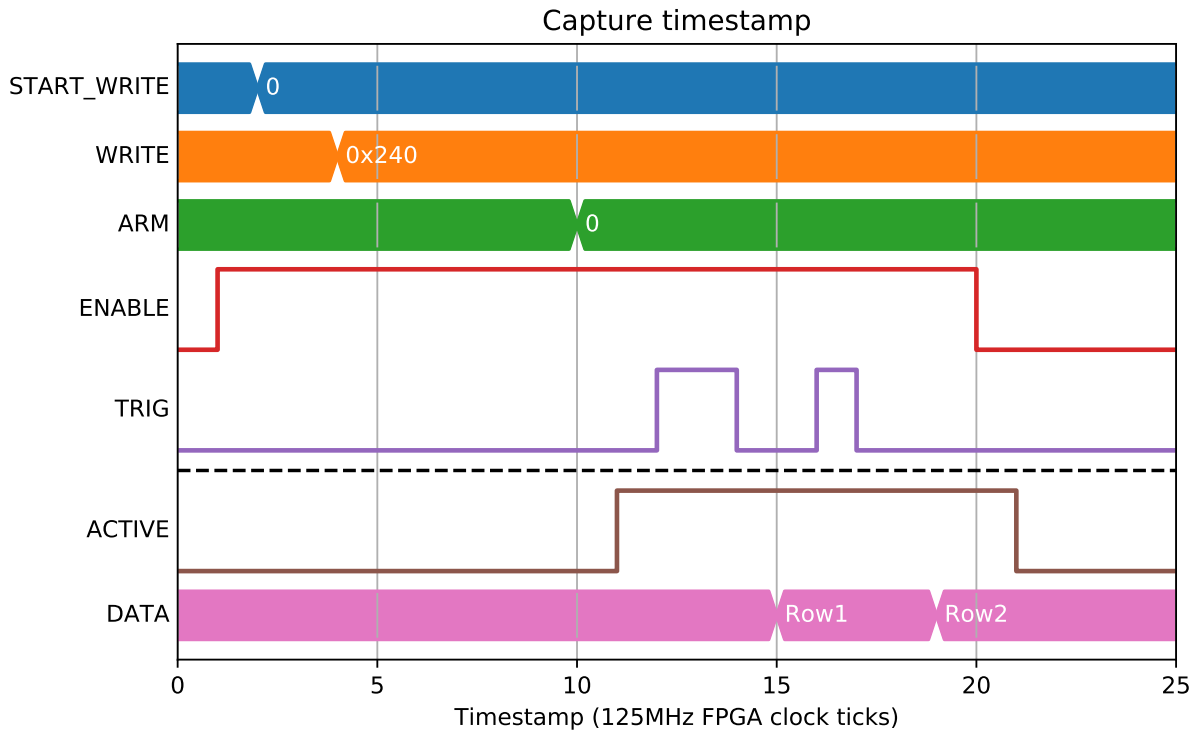
6.16.3 Capturing fields

Capturing fields is done by specifying a series of WRITE addresses. These are made up of a mode in the bottom 4 bits, and an index in the 6 bits above them. Indexes < 32 refer to entries on the pos_bus, while indexes >= 32 are extra entries specific to PCAP, like timestamps and number of gated samples. The values sent via the WRITE register are written from the TCP server, so will not be visible to end users.

Data is ticked out one at a time from the DATA attribute, then sent to the TCP server over DMA, before being sent to the user. It is reconstructed into a table in each of the examples below for ease of reading.

The following example shows PCAP being configured to capture the timestamp when CAPTURE goes high (0x24 is the bottom 32-bits of TS_CAPTURE).

Row	0x240
0	2
1	6



6.16.4 Pos bus capture

As well as general fields like the timestamp, any pos_bus index can be captured. Pos bus fields have multiple modes that they can capture in.

Mode 0 - Value

This gives an instantaneous capture of value no matter what the state of GATE:

Row	0x50
0	20
1	100
2	6

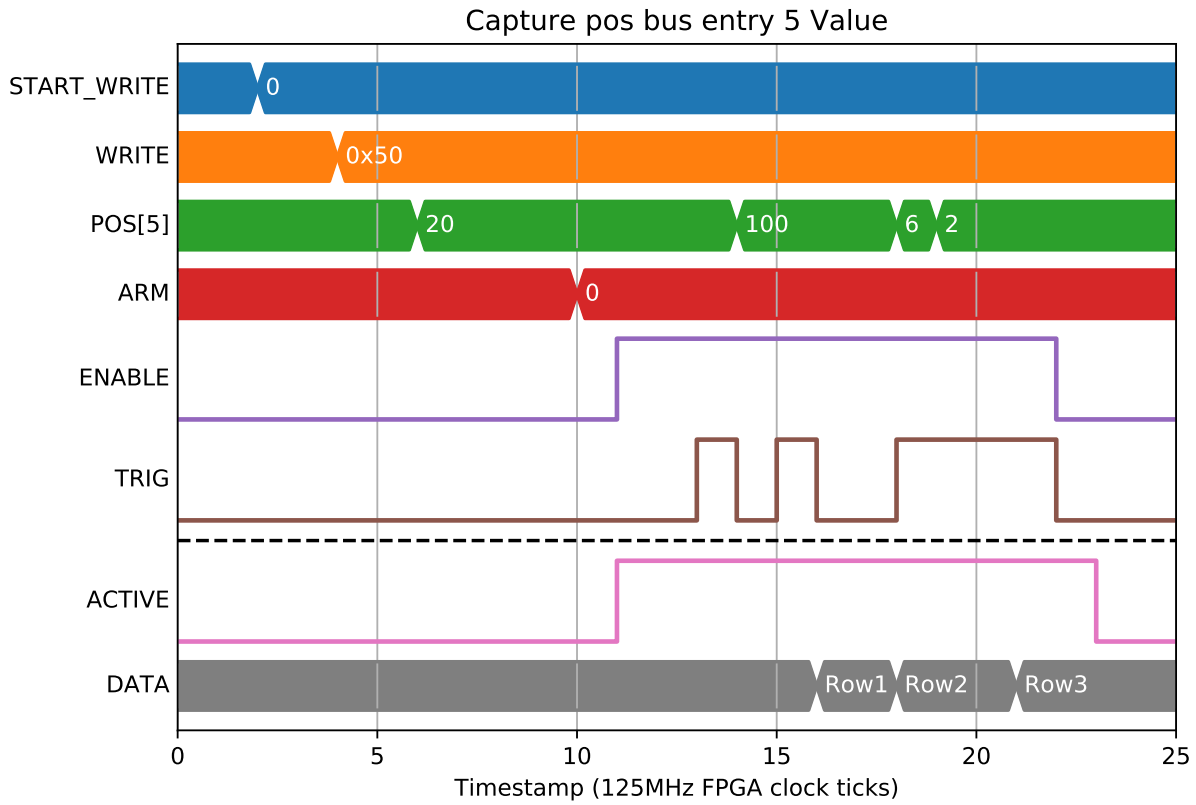
Mode 1 - Difference

This is mainly used for something like an incrementing counter value. It will only count the differences while GATE was high:

Row	0xB1
0	10
1	-5

Mode 2/3 - Sum Lo/Hi

Mode 2 is the lower 32-bits of the sum of all samples while GATE was high:



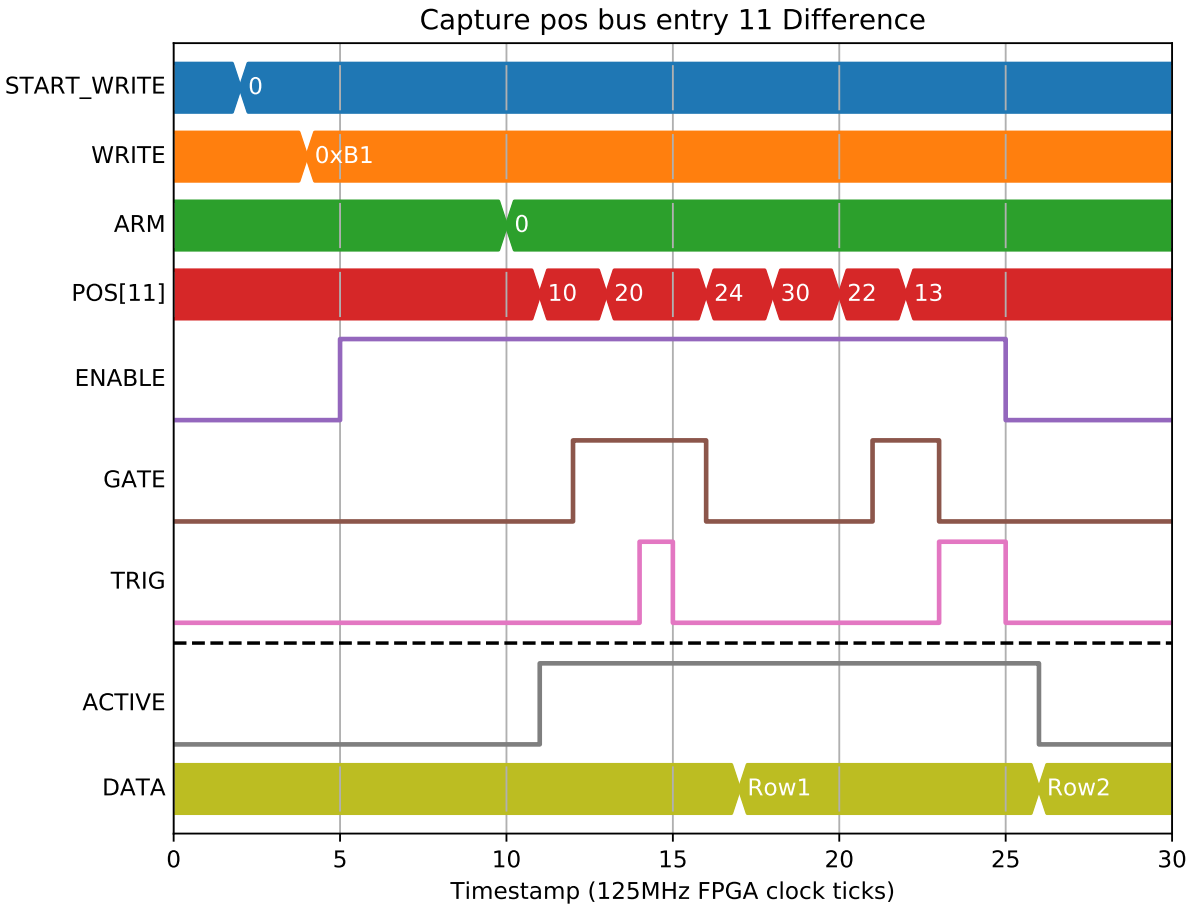
Row	0x32
0	6
1	21
2	206

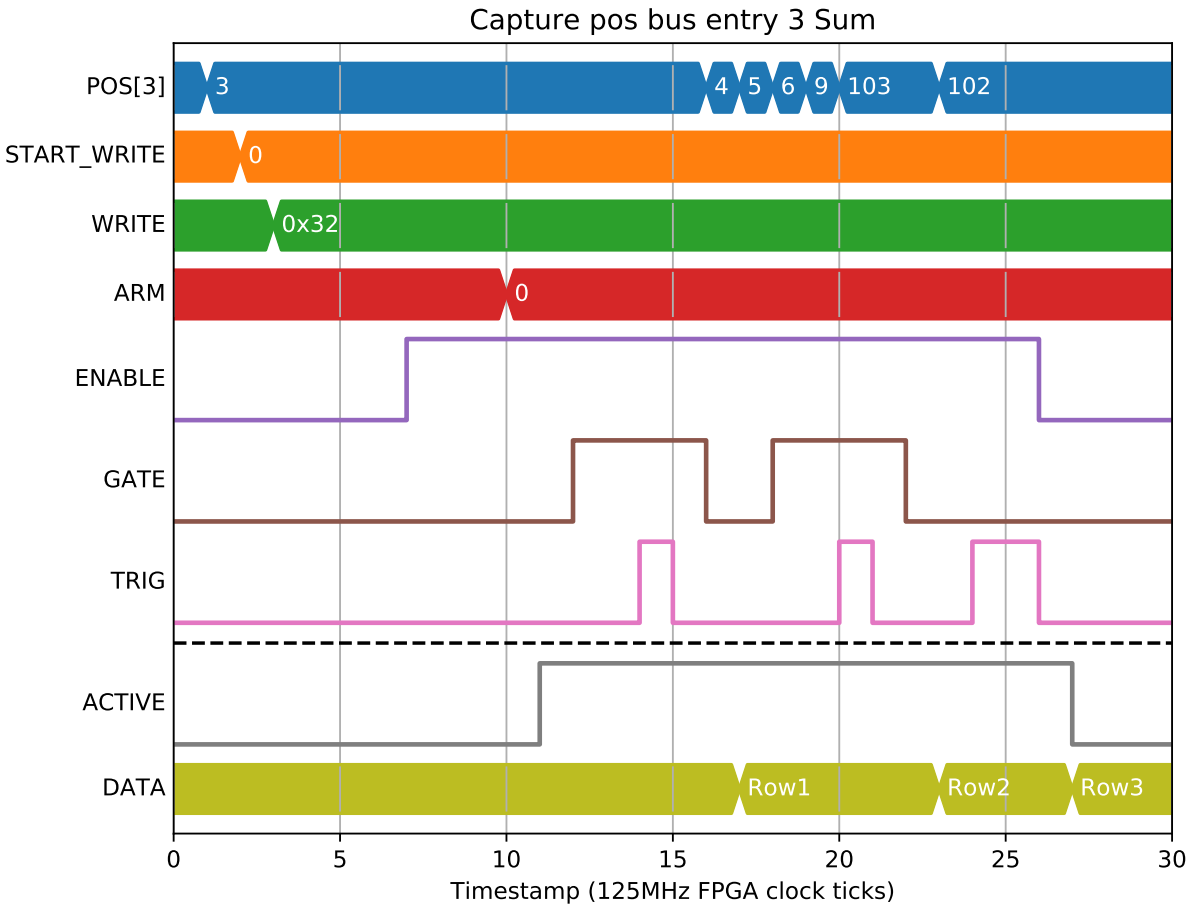
Mode 2 and 3 together gives the full 64-bits of sum, needed for any sizeable values on the pos_bus:

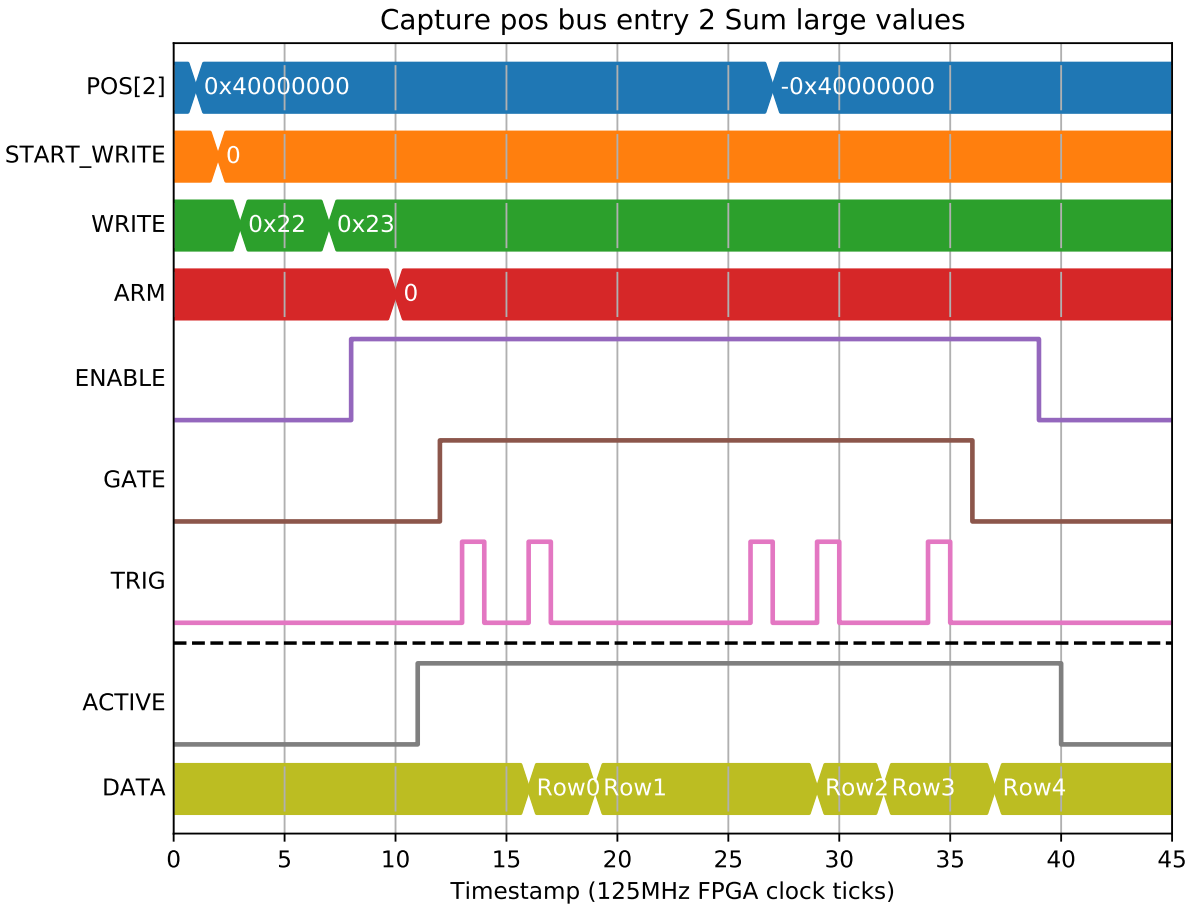
Row	0x22	0x23
0	1073741824	0
1	-1073741824	0
2	-2147483648	2
3	-1073741824	-1
4	-1073741824	-2

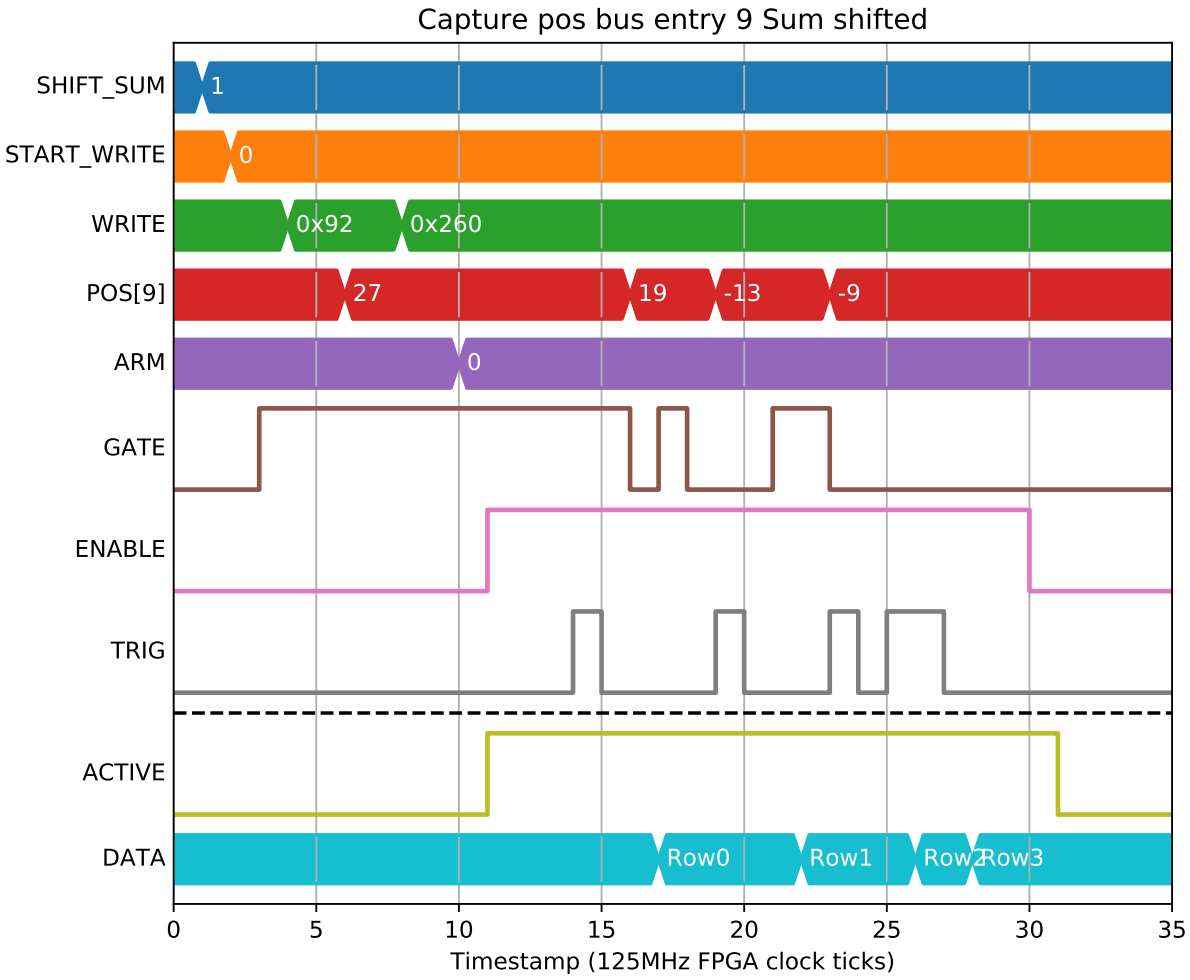
If long frame times (> 2**32 SAMPLES, > 30s), are to be used, then SHIFT_SUM can be used to shift both the sum and SAMPLES field by up to 8-bits to accomodate up to 125 hour frames. This example demonstrates the effect with smaller numbers:

Row	0x92	0x260
0	40	1
1	36	1
2	-13	1
3	0	0


Capture pos bus entry 11 Difference
START_WRITE
WRITE
ARM
POS[11]
ENABLE
GATE
TRIG
ACTIVE
DATA
0
0xB1
0
10
20
24
30
22
13
Row1
Row2
0
5
10
15
20
25
30
Timestamp (125MHz FPGA clock ticks)


Capture pos bus entry 3 Sum
POS[3]
START_WRITE
WRITE
ARM
ENABLE
GATE
TRIG
ACTIVE
DATA
3
4
5
6
9
103
102
0
0x32
0
Row1
Row2
Row3
0
5
10
15
20
25
30
Timestamp (125MHz FPGA clock ticks)

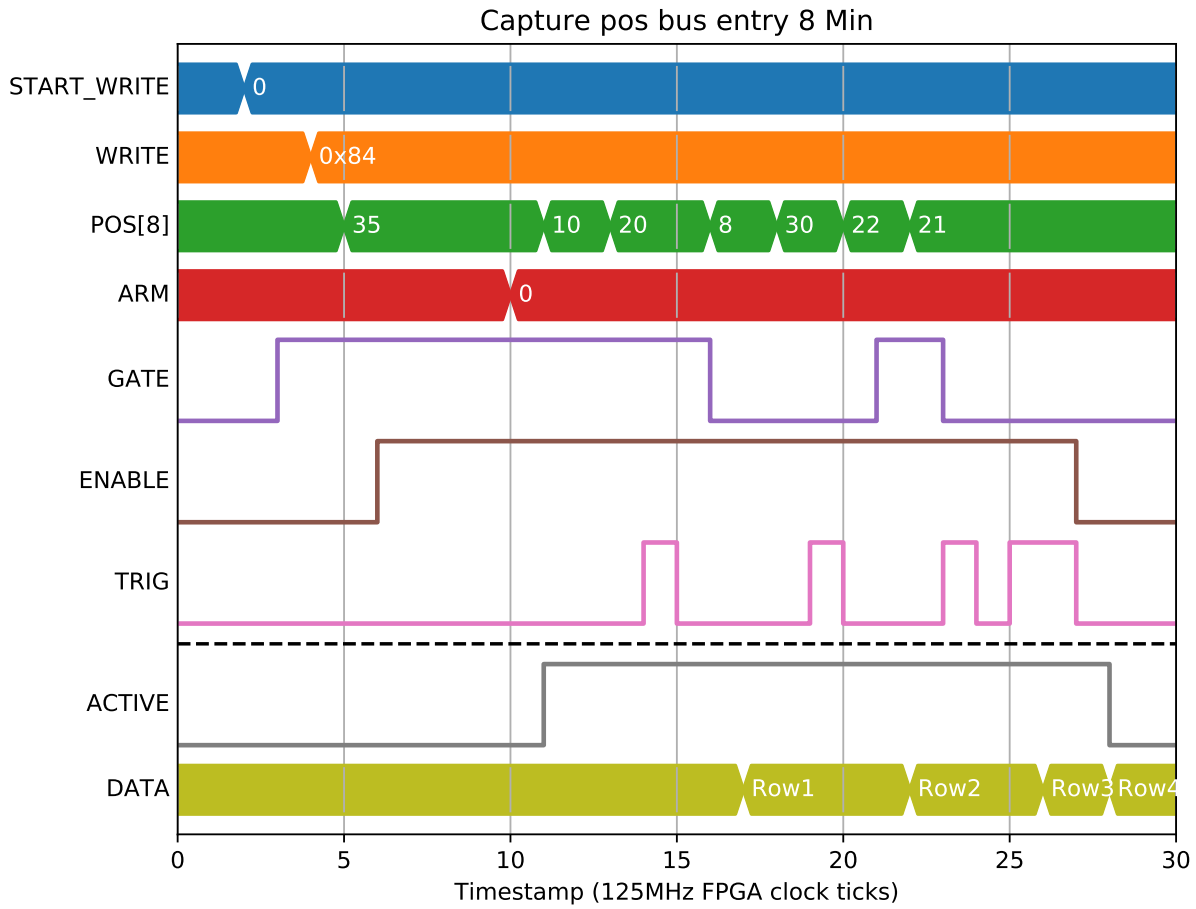

Capture pos bus entry 2 Sum large values
POS[2]
START_WRITE
WRITE
ARM
ENABLE
GATE
TRIG
ACTIVE
DATA
0x40000000
-0x40000000
0
0x22
0x23
0
Row0
Row1
Row2
Row3
Row4
0
5
10
15
20
25
30
35
40
45
Timestamp (125MHz FPGA clock ticks)


Capture pos bus entry 9 Sum shifted
SHIFT_SUM
START_WRITE
WRITE
POS[9]
ARM
GATE
ENABLE
TRIG
ACTIVE
DATA
1
0
0x92
0x260
27
19
-13
-9
0
Row0
Row1
Row2
Row3
0
5
10
15
20
25
30
35
Timestamp (125MHz FPGA clock ticks)

Mode 4/5 - Min/Max

Both of these modes calculate statistics on the value while GATE is high.

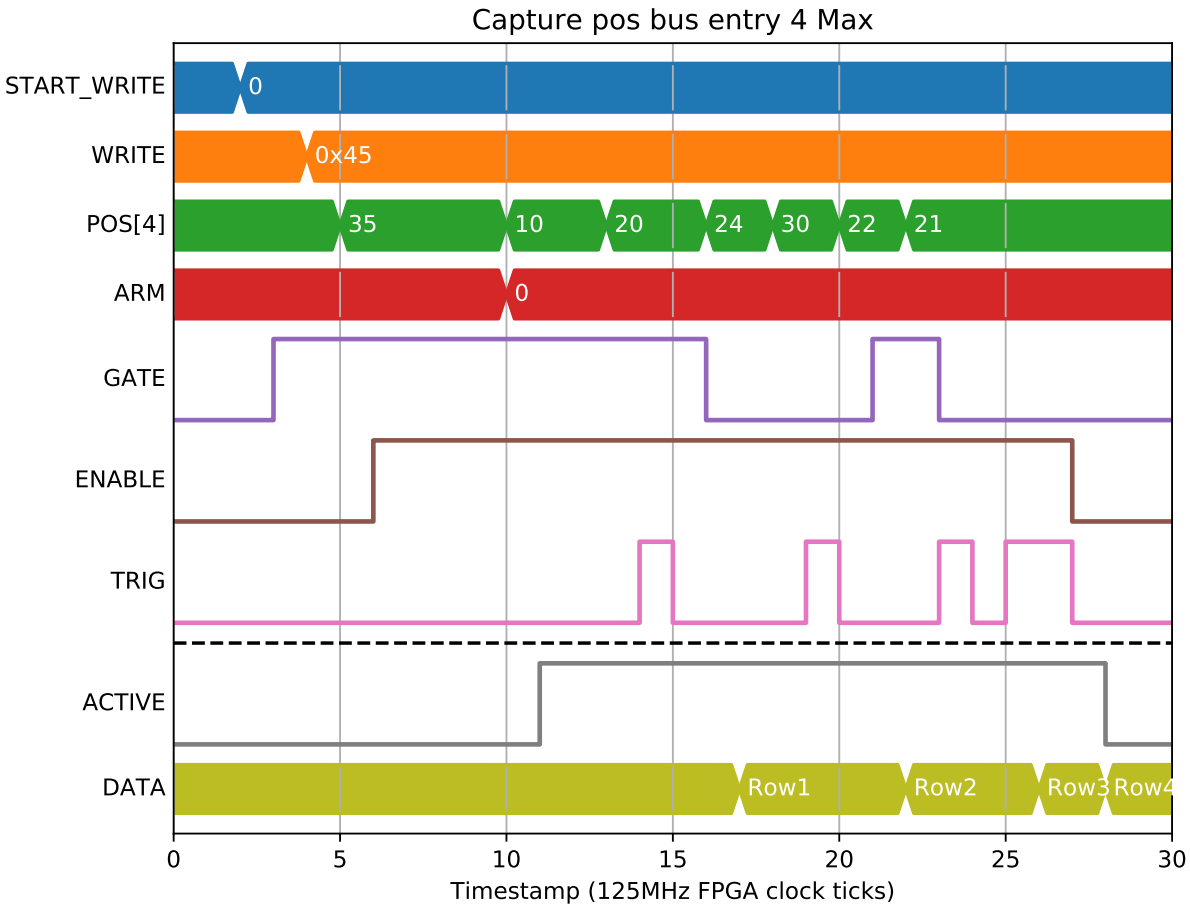
Mode 4 produces the min of all values or zero if the gate was low for all of the current capture:



Row	0x84
0	10
1	20
2	21
3	2147483647

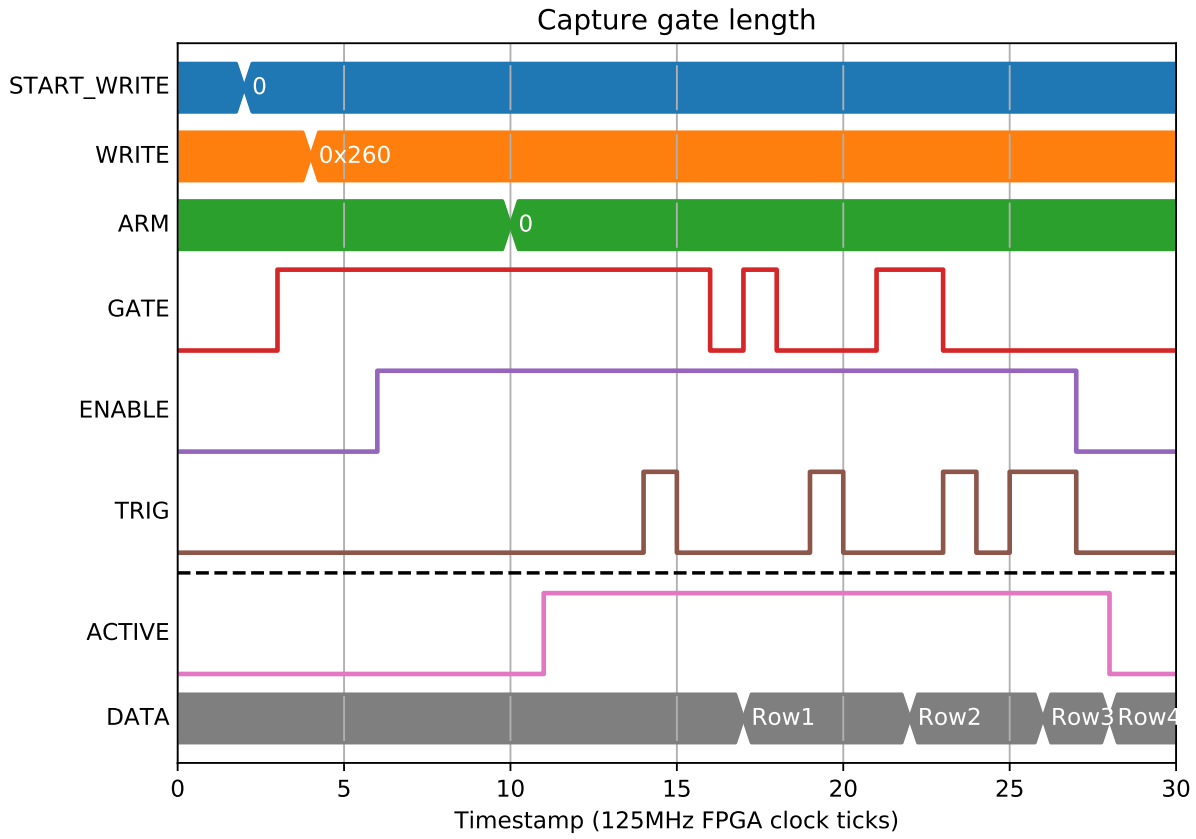
Mode 5 produces the max of all values in a similar way:

Row	0x45
0	20
1	20
2	22
3	-2147483648


Capture pos bus entry 4 Max
START_WRITE
WRITE
POS[4]
ARM
GATE
ENABLE
TRIG
ACTIVE
DATA
0
0x45
35
10
20
24
30
22
21
0
Row1
Row2
Row3
Row4
0
5
10
15
20
25
30
Timestamp (125MHz FPGA clock ticks)

6.16.5 Number of samples

There is a SAMPLES field that can be captured that will give the number of clock ticks that GATE was high during a single CAPTURE. This field allows the TCP server to offer "Mean" as a capture option, dividing "Sum" by SAMPLES to get the mean value of the field during the capture period. It can also be captured separately to give the gate length:

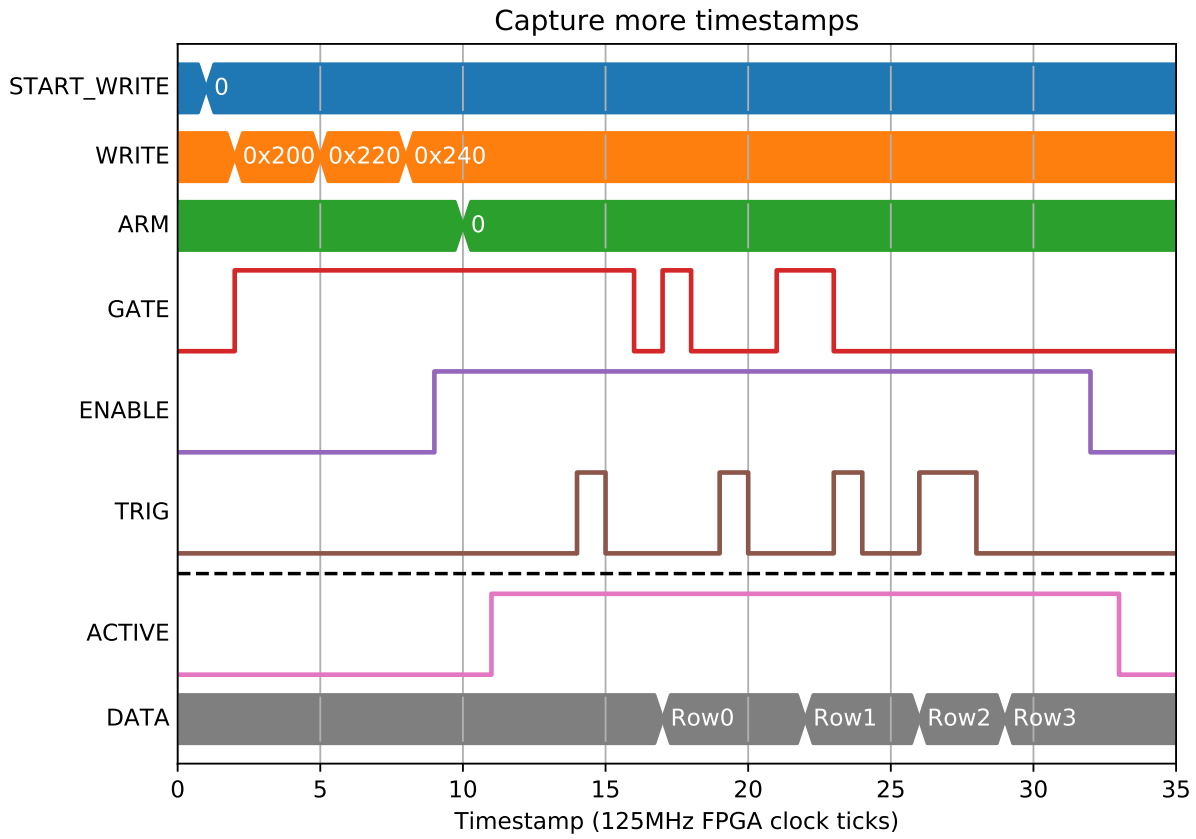


Row	0x260
0	4
1	3
2	2
3	0

6.16.6 Timestamps

As well as the timestamp of the capture signal, timestamps can also be generated for the start of each capture period (first gate high signal) and end (the tick after the last gate high). These are again split into two 32-bit segments so only the lower bits need to be captured for short captures. In the following example we capture TS_START (0x20), TS_END (0x22) and TS_CAPTURE (0x24) lower bits:

Row	0x200	0x220	0x240
0	0	4	4
1	4	8	9
2	11	13	13
3	-1	-1	16



6.16.7 Bit bus capture

The state of the bit bus at capture can also be captured. It is split into 4 quadrants of 32-bits each. For example, to capture signals 0..31 on the bit bus we would use BITS0 (0x27):

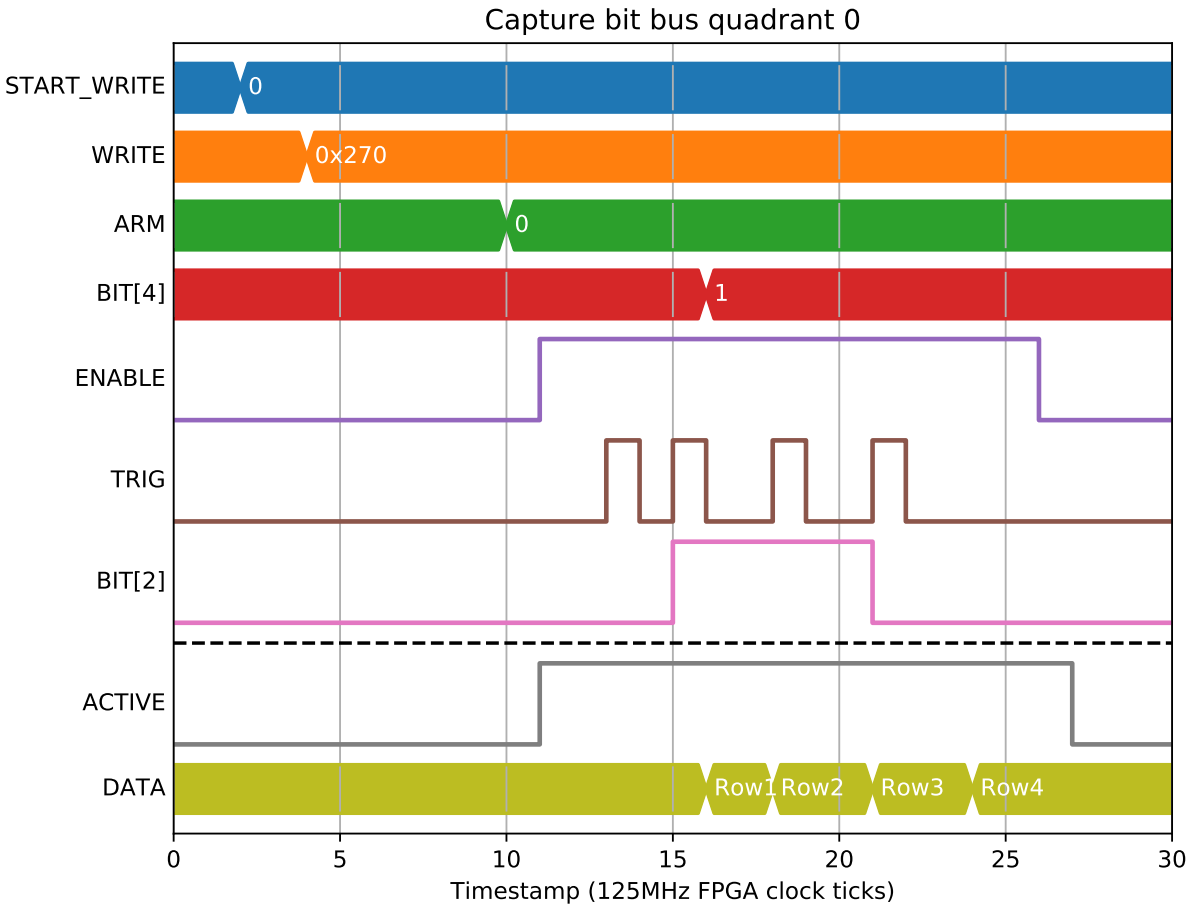
Row	0x270
0	0
1	4
2	20
3	16

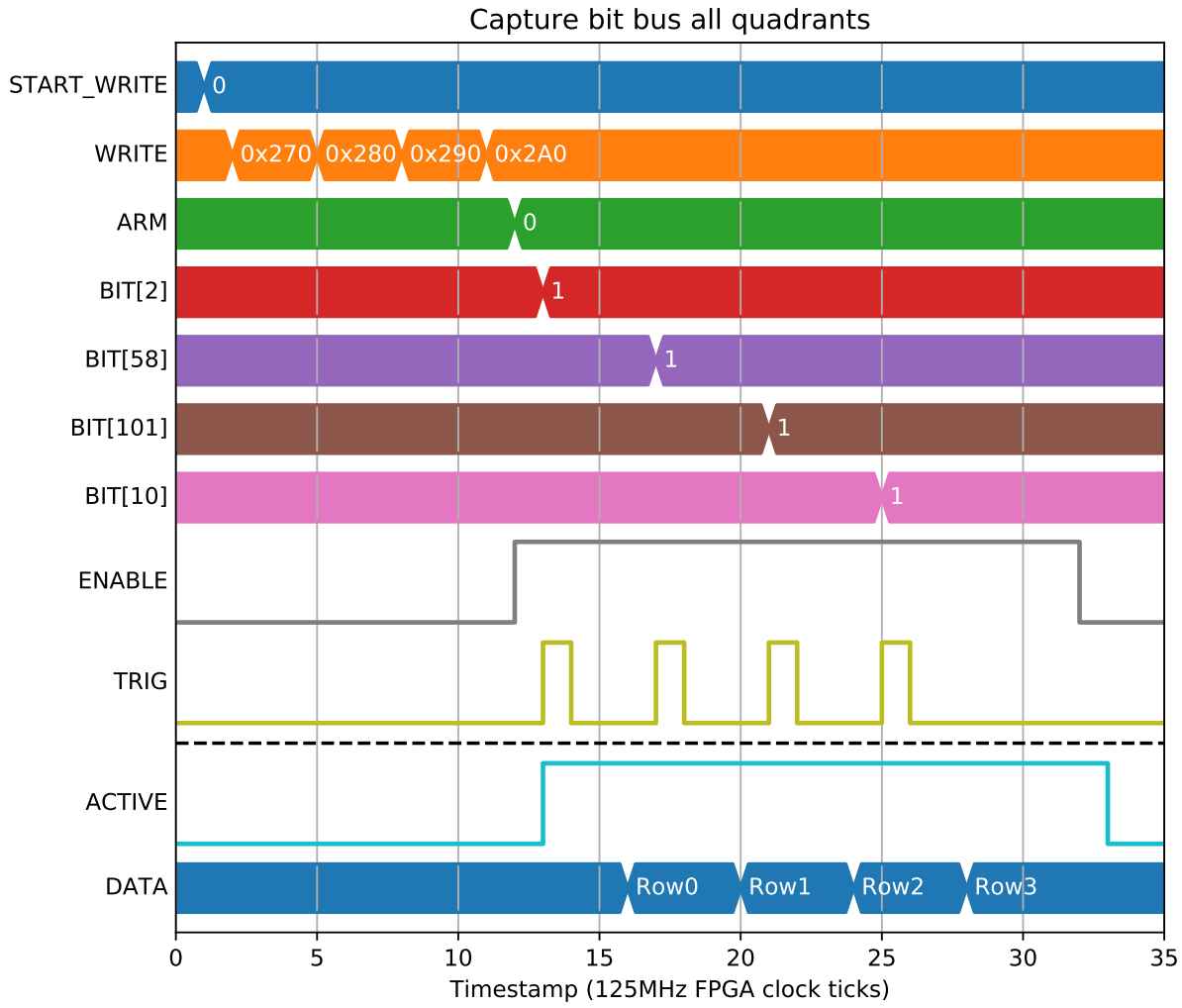
By capturing all 4 quadrants (0x27..0x2A) we get the whole bit bus:

Row	0x270	0x280	0x290	0x2A0
0	4	0	0	0
1	4	67108864	0	0
2	4	67108864	0	32
3	1028	67108864	0	32

6.16.8 Triggering options

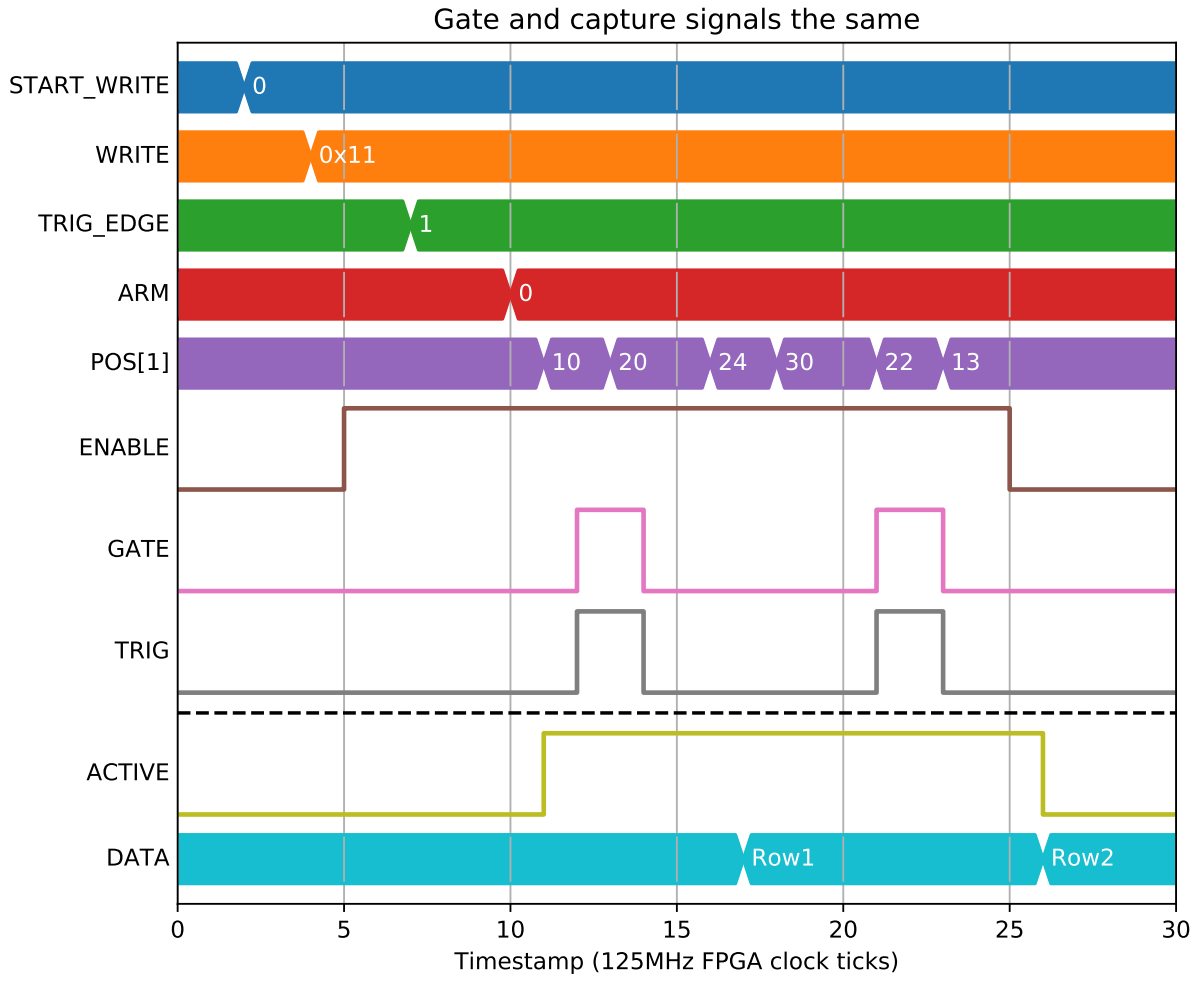
ENABLE and GATE are level triggered, with ENABLE used for marking the start and end of the entire acquisition, and GATE used to accept or reject samples within a single capture from the acquisition. CAPTURE is edge triggered


Capture bit bus quadrant 0
START_WRITE
WRITE
ARM
BIT[4]
ENABLE
TRIG
BIT[2]
ACTIVE
DATA
0
0x270
0
1
Row1
Row2
Row3
Row4
0
5
10
15
20
25
30
Timestamp (125MHz FPGA clock ticks)


Capture bit bus all quadrants
START_WRITE
WRITE
ARM
BIT[2]
BIT[58]
BIT[101]
BIT[10]
ENABLE
TRIG
ACTIVE
DATA
0
0x270
0x280
0x290
0x2A0
0
1
1
1
1
Row0
Row1
Row2
Row3
0
5
10
15
20
25
30
35
Timestamp (125MHz FPGA clock ticks)

with an option to trigger on rising, falling or both edges.

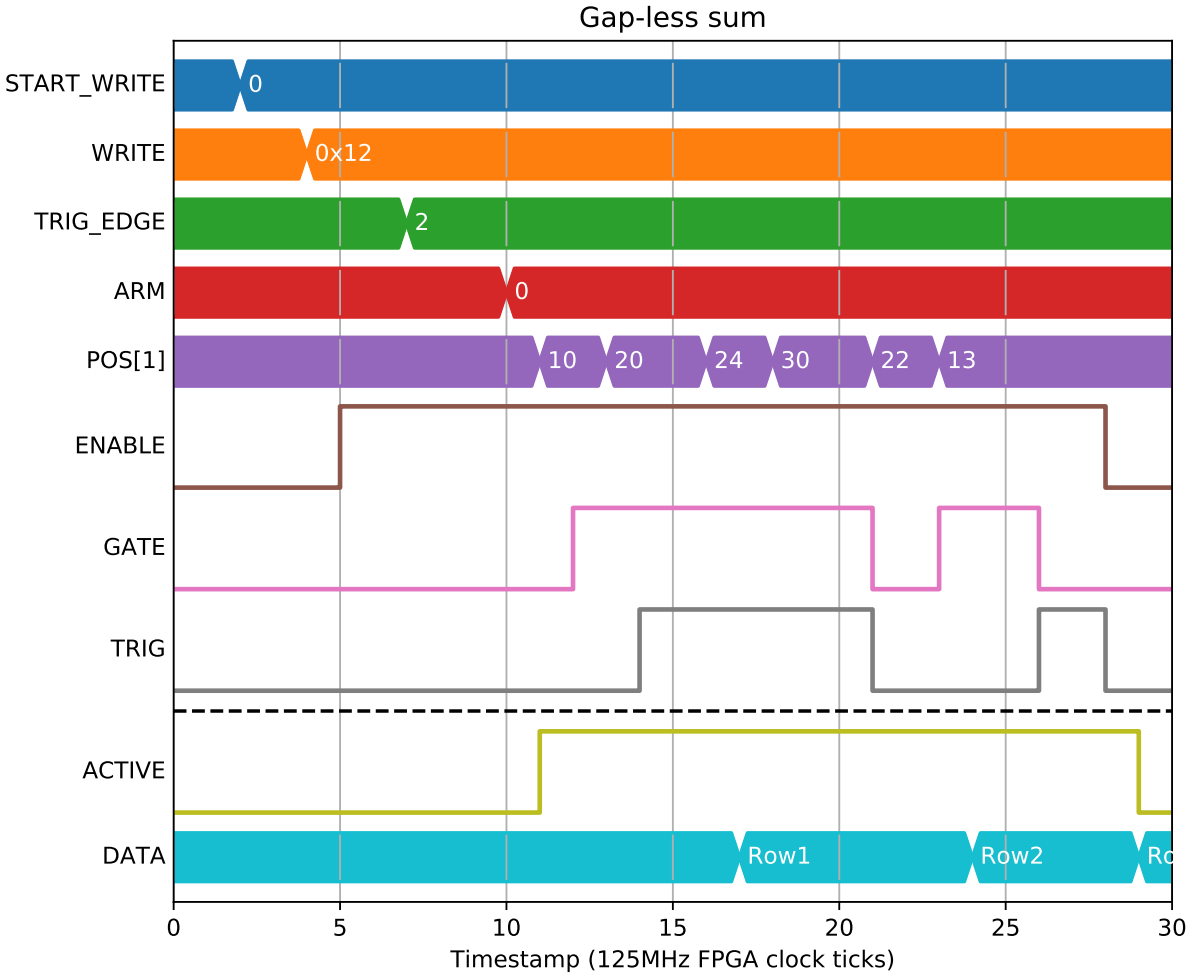
Triggering on rising is the default, explored in the preceding examples. Triggering on falling edge would be used if you have a gate signal that marks the capture boundaries and want sum or difference data within. For example, to capture the amount POS[1] changes in each capture gate we could connect GATE and CAPTURE to the same signal:



Row	0x11
0	10
1	-9

Another option would be a gap-less acquisition of sum while gate is high with capture boundaries marked with a toggle of CAPTURE:

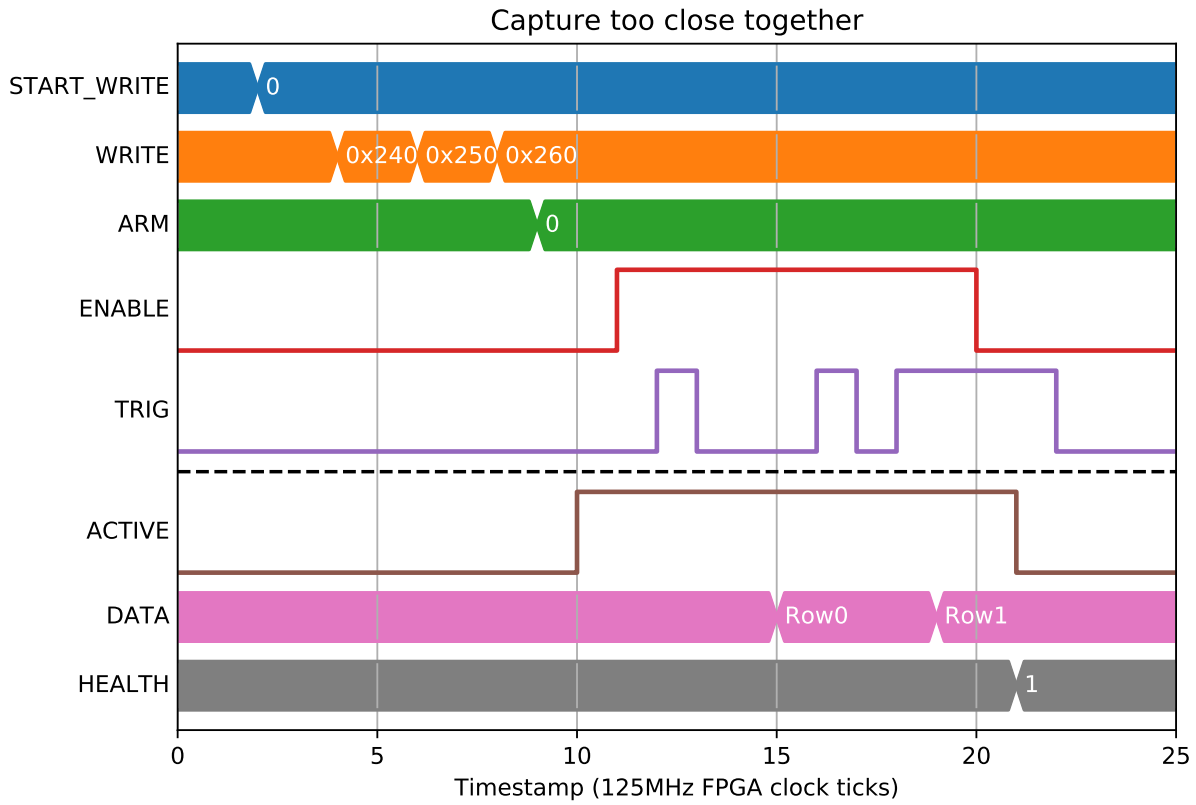
Row	0x12
0	30
1	178
2	39


Gap-less sum
START_WRITE
WRITE
TRIG_EDGE
ARM
POS[1]
ENABLE
GATE
TRIG
ACTIVE
DATA
0
0x12
2
0
10
20
24
30
22
13
Row1
Row2
0
5
10
15
20
25
30
Timestamp (125MHz FPGA clock ticks)

6.16.9 Error conditions

The distance between capture signals must be at least the number of 32-bit capture fields. If 2 capture signals are too close together HEALTH will be set to 1 (Capture events too close together).

In this example there are 3 fields captured (TS_CAPTURE_L, TS_CAPTURE_H, SAMPLES), but only 2 clock ticks between the 2nd and 3rd capture signals:



Row	0x240	0x250	0x260
0	1	0	0
1	5	0	0

6.17 PCOMP - Position Compare

The position compare block takes a position input and allows a regular number of threshold comparisons to take place on a position input. The normal order of operations is something like this:

- If PRE_START > 0 then wait until position has passed START - PRE_START
- If START > 0 then wait until position has passed START and set OUT=1
- Wait until position has passed START + WIDTH and set OUT=0
- Wait until position has passed START + STEP and set OUT=1
- Wait until position has passed START + STEP + WIDTH and set OUT=0
- Continue until PULSES have been produced

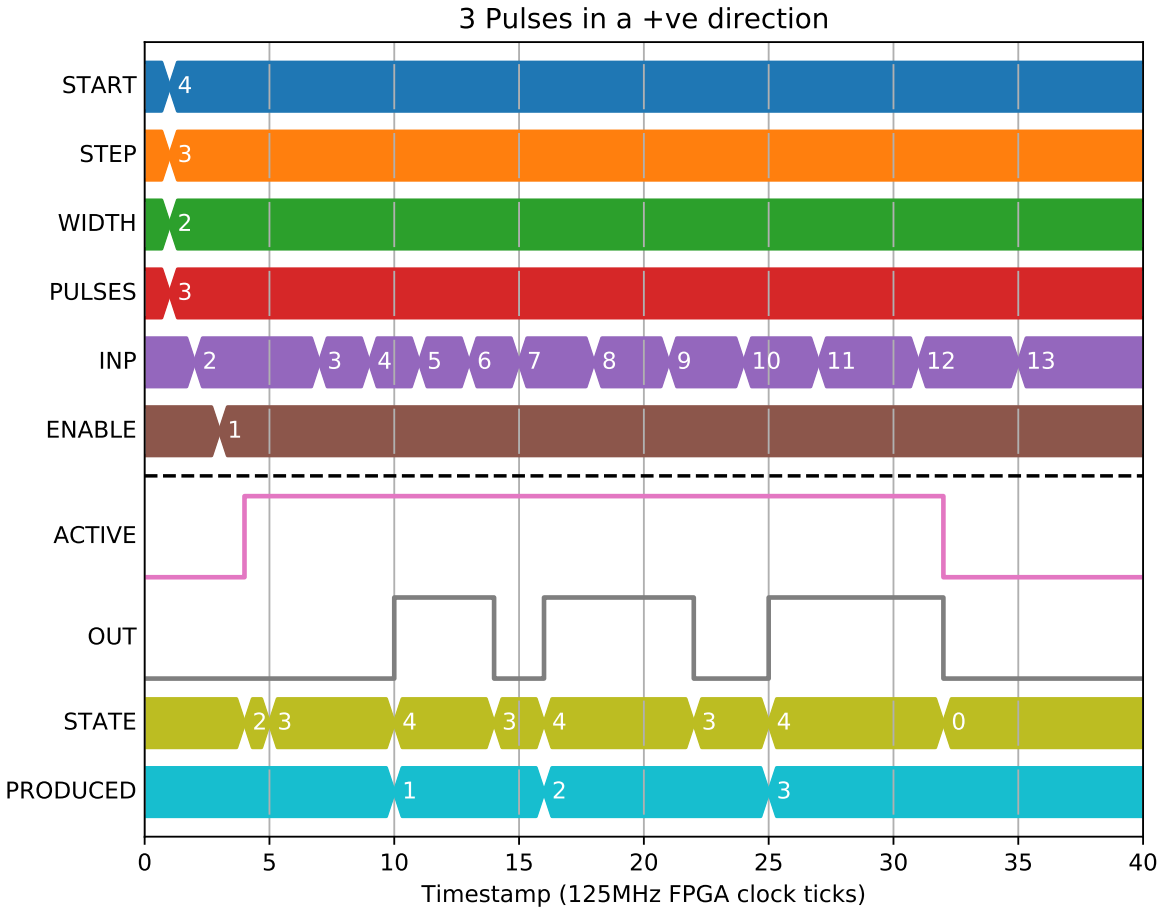
It can be used to generate a position based pulse train against an input encoder or analogue system, or to work as repeating comparator.

6.17.1 Fields

Name	Type	Description
ENABLE	bit_mux	Stop on falling edge, reset and enable on rising edge
INP	pos_mux	Position data from position-data bus
PRE_START	param int	INP must be this far from START before waiting for START
START	param int	Pulse absolute/relative start position value
WIDTH	param int	The relative distance between a rising and falling edge
STEP	param int	The relative distance between successive rising edges
PULSES	param	The number of pulses to produce, 0 means infinite
RELATIVE	param enum	If 1 then START is relative to the position of INP at enable 0 Absolute 1 Relative
DIR	param enum	Direction to apply all relative offsets to 0 Positive 1 Negative 2 Either
ACTIVE	bit_out	Active output is high while block is in operation
OUT	bit_out	Output pulse train
HEALTH	read enum	Error details if anything goes wrong 0 OK 1 Position jumped by more than STEP 2 Can't guess DIR when RELATIVE and PRE_START=0 and START=0
PRODUCED	read	The number of pulses produced
STATE	read enum	The internal statemachine state 0 WAIT_ENABLE 1 WAIT_DIR 2 WAIT_PRE_START 3 WAIT_RISING 4 WAIT_FALLING

6.17.2 Position compare is directional

A typical example would setup the parameters, enable the block, then start moving a motor to trigger a series of pulses:

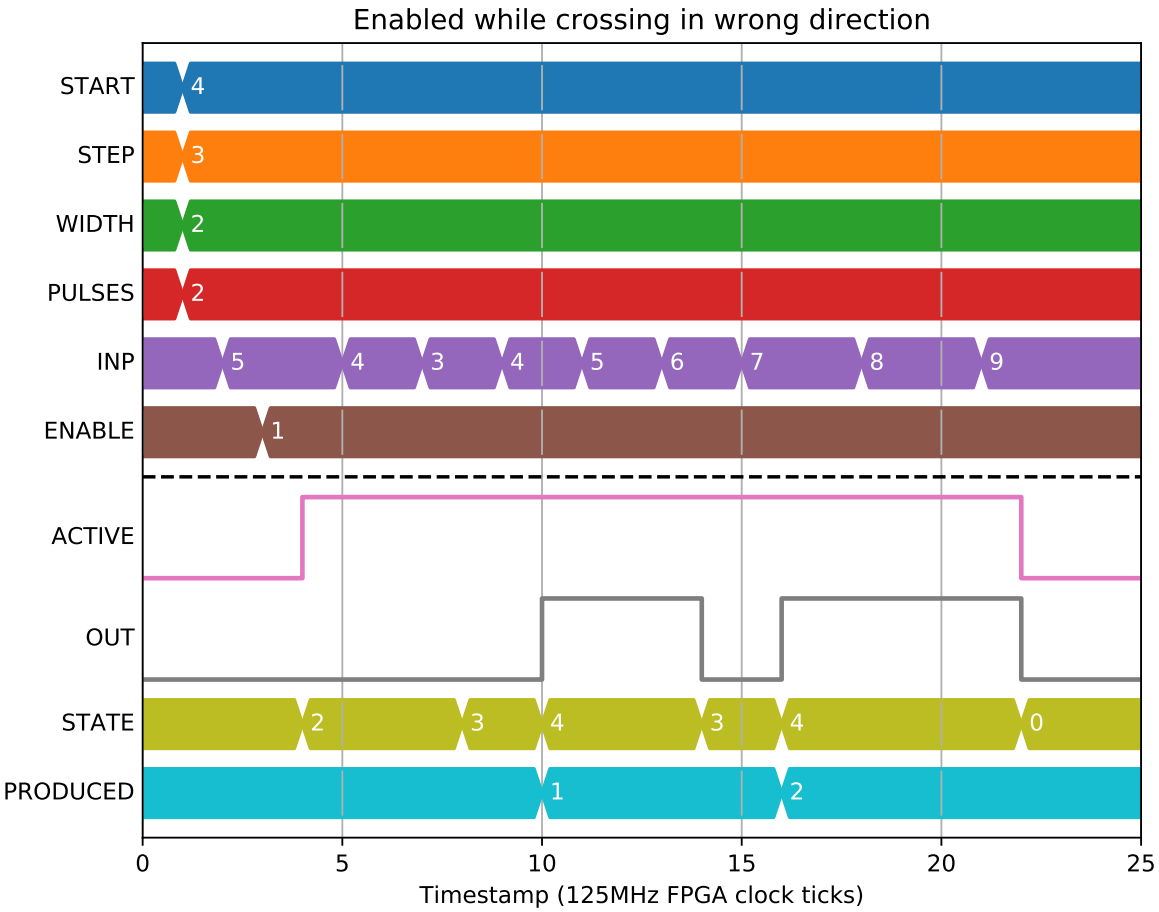


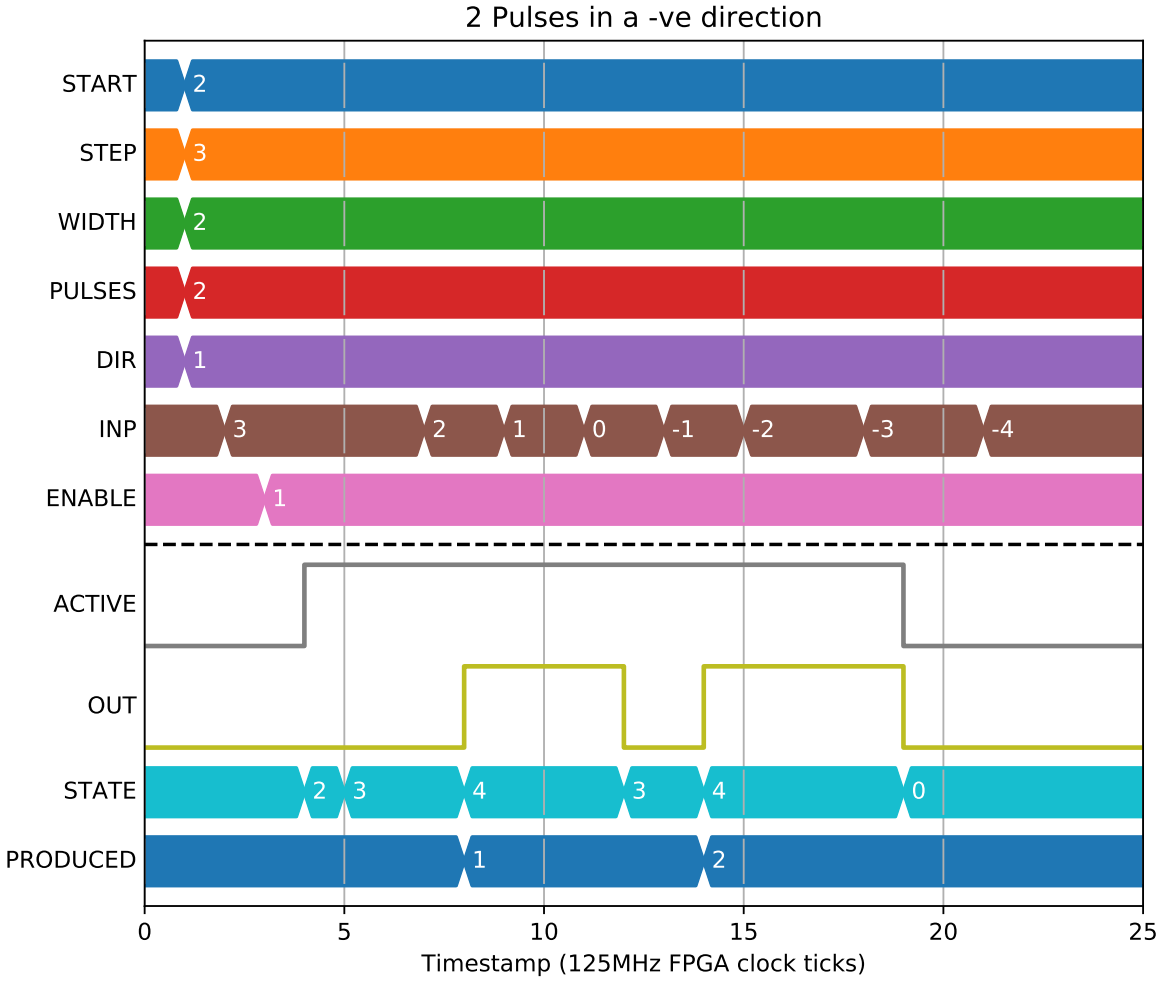
But if we get the direction wrong, we won't get the first pulse until we cross START in the correct direction:

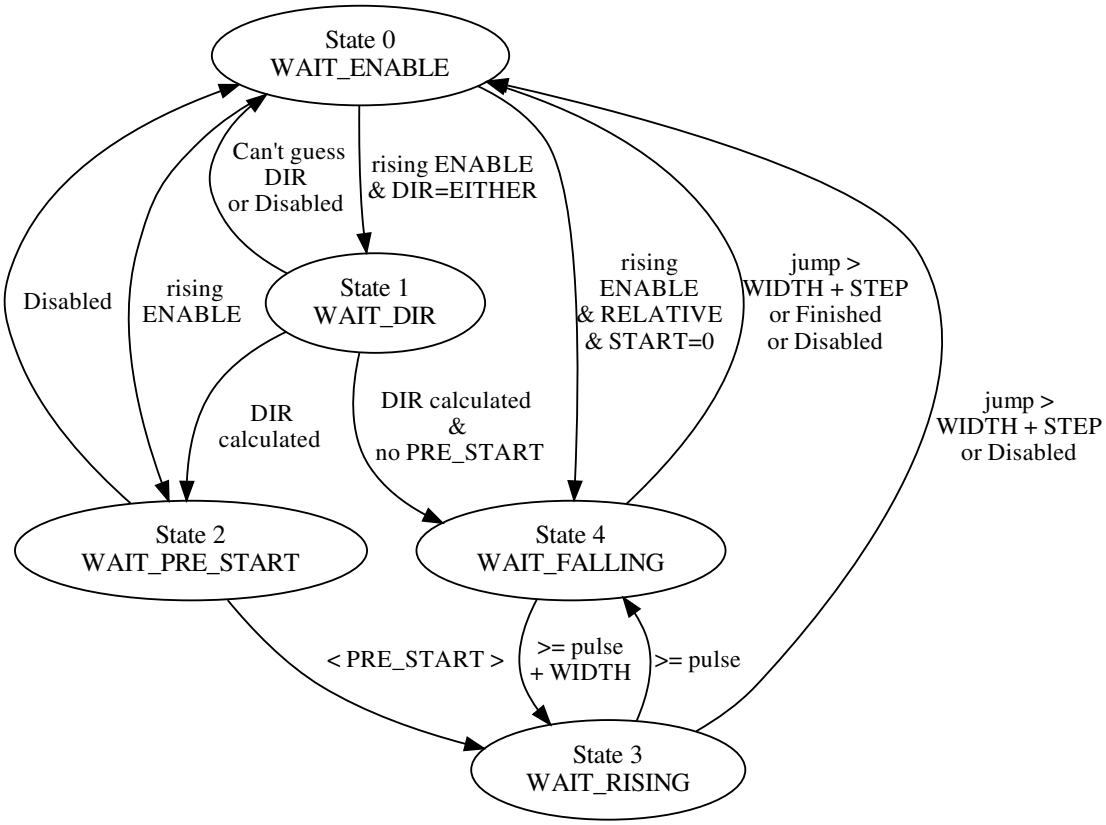
Moving in a negative direction works in a similar way. Note that WIDTH and PULSE still have positive values:

6.17.3 Internal statemachine

The Block has an internal statemachine that is exposed as a parameter, allowing the user to see what the Block is currently doing:


Enabled while crossing in wrong direction
START
STEP
WIDTH
PULSES
INP
ENABLE
ACTIVE
OUT
STATE
PRODUCED
0
5
10
15
20
25
Timestamp (125MHz FPGA clock ticks)


2 Pulses in a -ve direction
START
STEP
WIDTH
PULSES
DIR
INP
ENABLE
ACTIVE
OUT
STATE
PRODUCED
2
3
2
2
1
3
2
1
0
-1
-2
-3
-4
1
2
3
4
3
4
0
1
2
0
5
10
15
20
25
Timestamp (125MHz FPGA clock ticks)



6.17.4 Not generating a pulse more than once

A key part of position compare is not generating a pulse at a position more than once. This is to deal with noisy encoders:

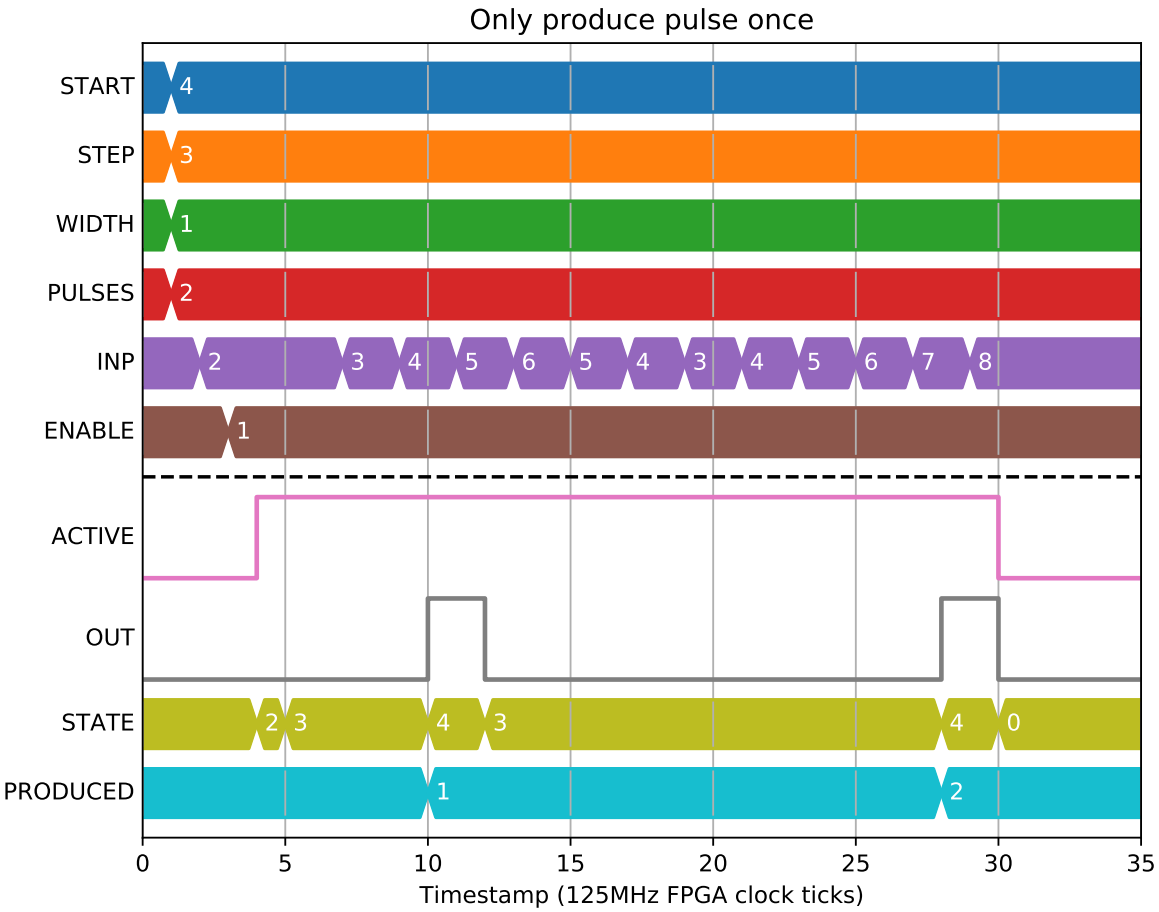
This means that care is needed if using direction sensing or relying on the directionality of the encoder when passing the start position. For example, if we approach START from the negative direction while doing a positive position compare, then jitter back over the start position, we will generate start at the wrong place. If you look carefully at the statemachine you will see that the Block crossed into WAIT_START when INP < 4 (START), which is too soon for this amount of jitter:

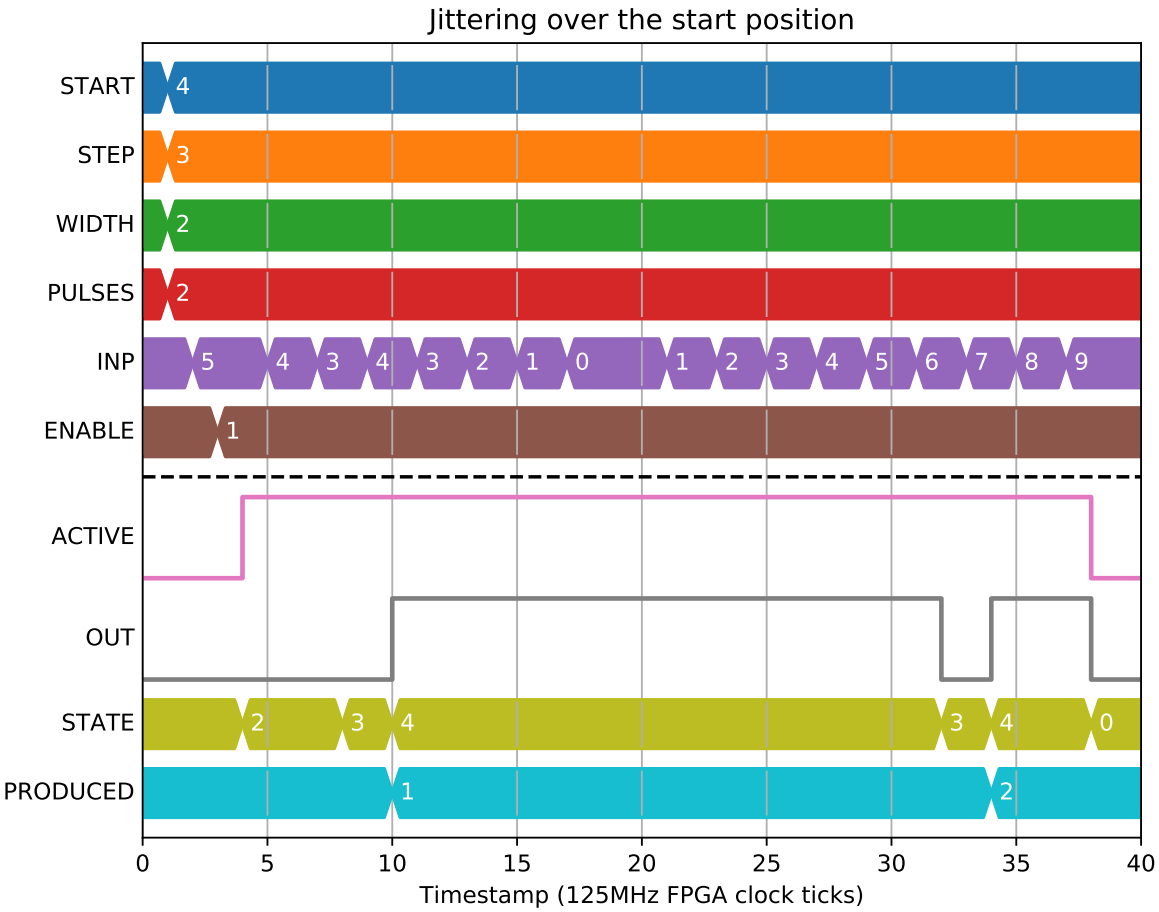
We can fix this by adding to the PRE_START deadband which the encoder has to cross in order to advance to the WAIT_START state. Now INP < 2 (START-PRE_START) is used for the condition of crossing into WAIT_START:

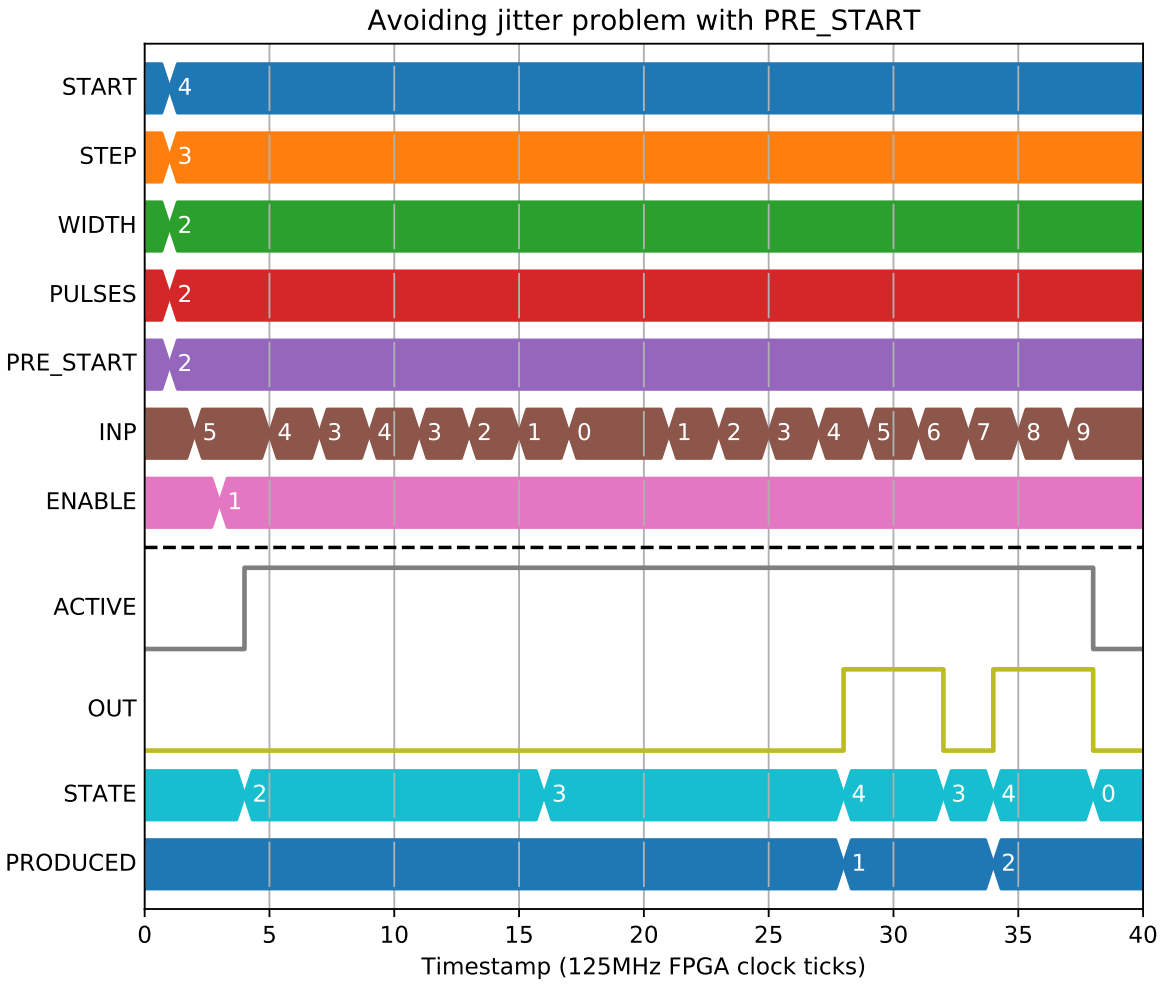
6.17.5 Guessing the direction

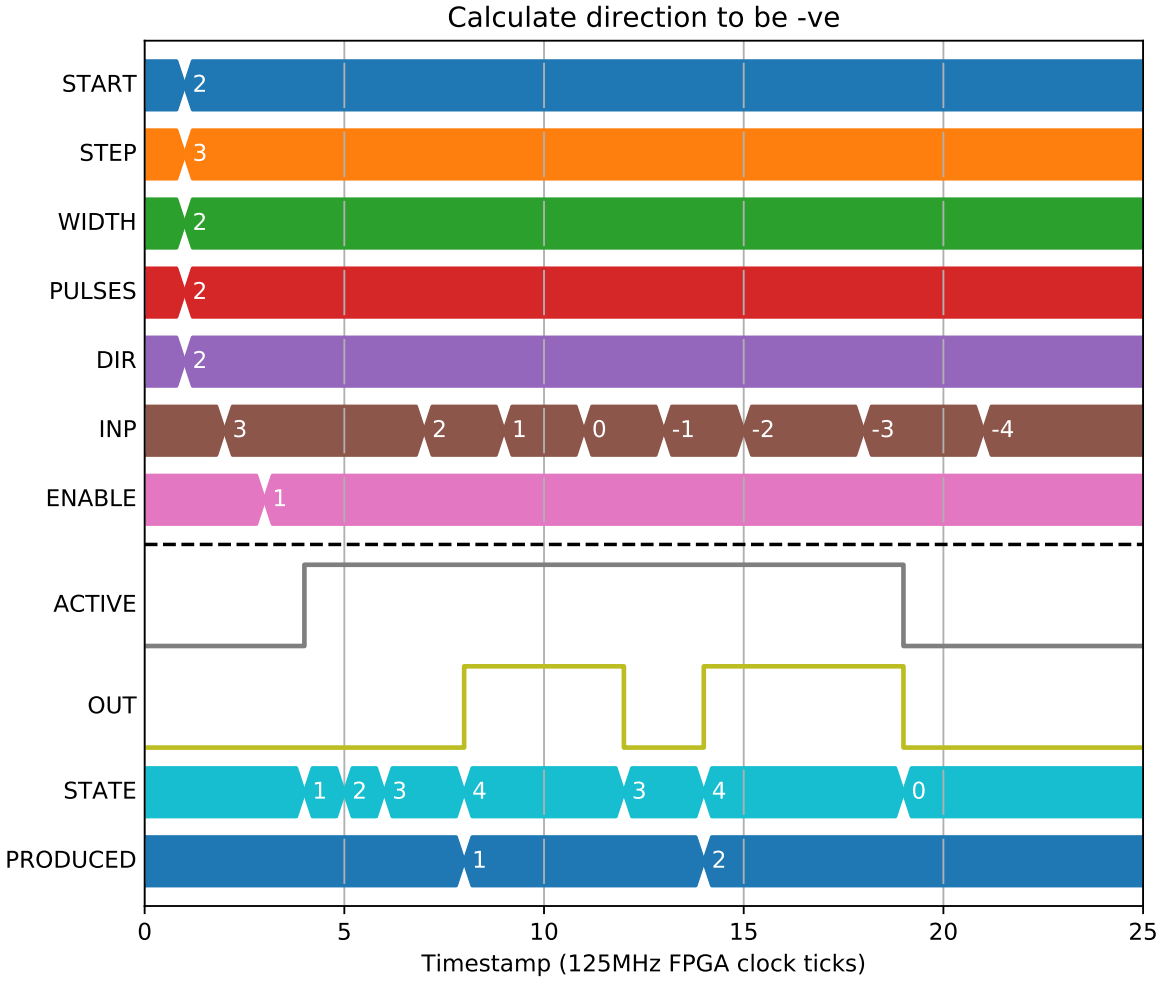
We can also ask to the Block to calculate direction for us:

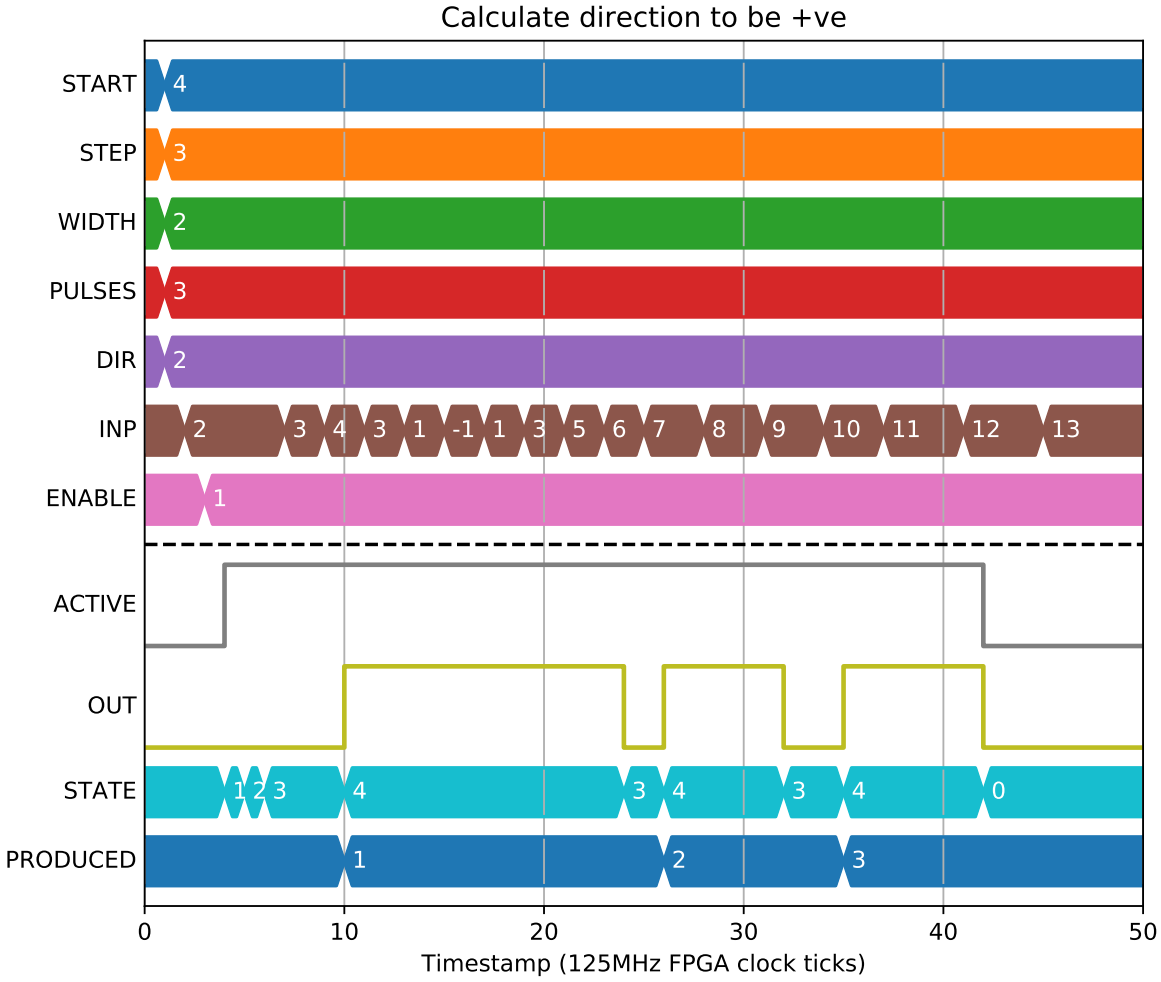
This is a one time calculation of direction at the start of operation, once the encoder has been moved enough to guess the direction then it is fixed until the Block has finished producing pulses:


Only produce pulse once
START
STEP
WIDTH
PULSES
INP
ENABLE
ACTIVE
OUT
STATE
PRODUCED
0
5
10
15
20
25
30
35
Timestamp (125MHz FPGA clock ticks)


Jittering over the start position
START
STEP
WIDTH
PULSES
INP
ENABLE
ACTIVE
OUT
STATE
PRODUCED
0
5
10
15
20
25
30
35
40
Timestamp (125MHz FPGA clock ticks)

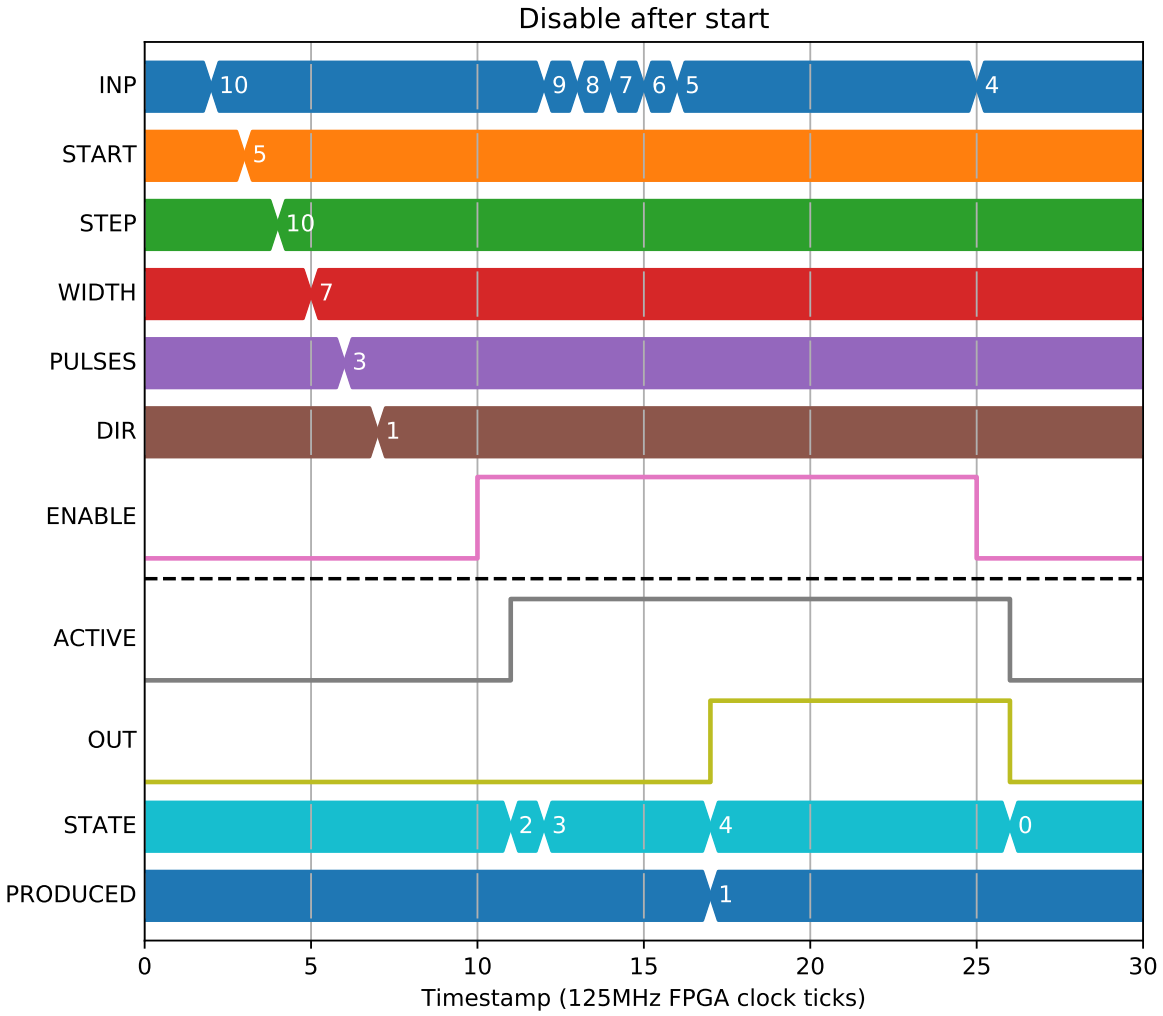

Avoiding jitter problem with PRE_START
START
STEP
WIDTH
PULSES
PRE_START
INP
ENABLE
ACTIVE
OUT
STATE
PRODUCED
0
5
10
15
20
25
30
35
40
Timestamp (125MHz FPGA clock ticks)


Calculate direction to be -ve
START
STEP
WIDTH
PULSES
DIR
INP
ENABLE
ACTIVE
OUT
STATE
PRODUCED
2
3
2
2
2
3
2
1
0
-1
-2
-3
-4
1
1
2
3
4
3
4
0
1
2
0
5
10
15
20
25
Timestamp (125MHz FPGA clock ticks)


Calculate direction to be +ve
START
STEP
WIDTH
PULSES
DIR
INP
ENABLE
ACTIVE
OUT
STATE
PRODUCED
0
10
20
30
40
50
Timestamp (125MHz FPGA clock ticks)

6.17.6 Interrupting a scan

When the ENABLE input is set low the output will cease. This will happen even if the ENABLE is set low when there are still cycles of the output pulse to generate, or if the ENABLE = 0 is set at the same time as a position match.



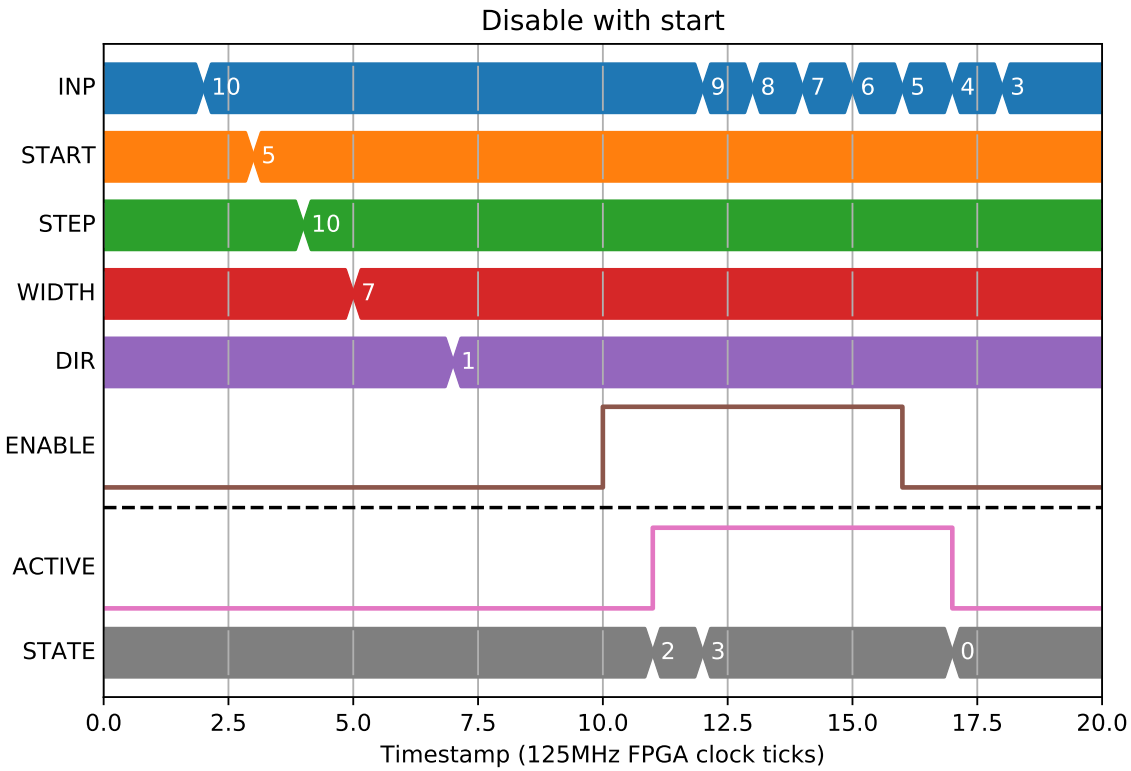
6.17.7 Position compare on absolute values

Doing position compare on an absolute value adds additional challenges, as we are not guaranteed to see every transition. It works in much the same way as the previous examples, but we trigger on greater than or equal rather than just greater than:

But what should the Block do if the output is 0 and the position jumps by enough to trigger a transition to 1 and then back to 0? We handle this by setting HEALTH="Error: Position jumped by more than STEP" and aborting the compare:

Likewise if the output is 1 and the position causes us to need to produce a 0 then 1:

And if we skipped a larger number of points we get the same error:



6.17.8 Relative position compare

We may want to nest position compare blocks, or respond to some external event. In which case, we expose the option to a position compare relative to the latched position at the start:

If we want it to start immediately on ENABLE then we set START and PRE_START=0:

We can also guess the direction in relative mode:

This works when going negative too:

And with a PRE_START value we guess the direction to be the opposite to the direction the motor is travelling when it exceeds PRE_START:

We cannot guess the direction when RELATIVE mode is set with no START or PRE_START though, the Block will error in this case:

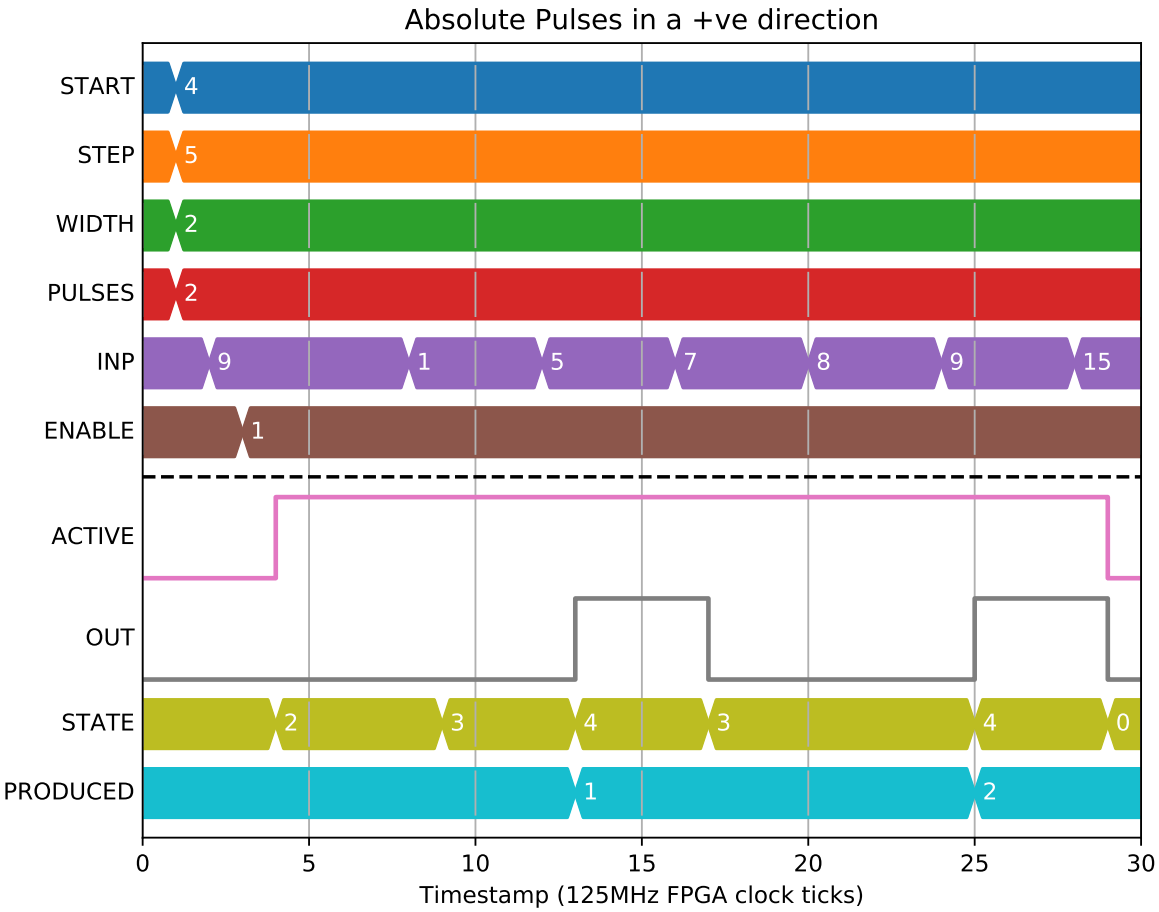
6.17.9 Use as a Schmitt trigger

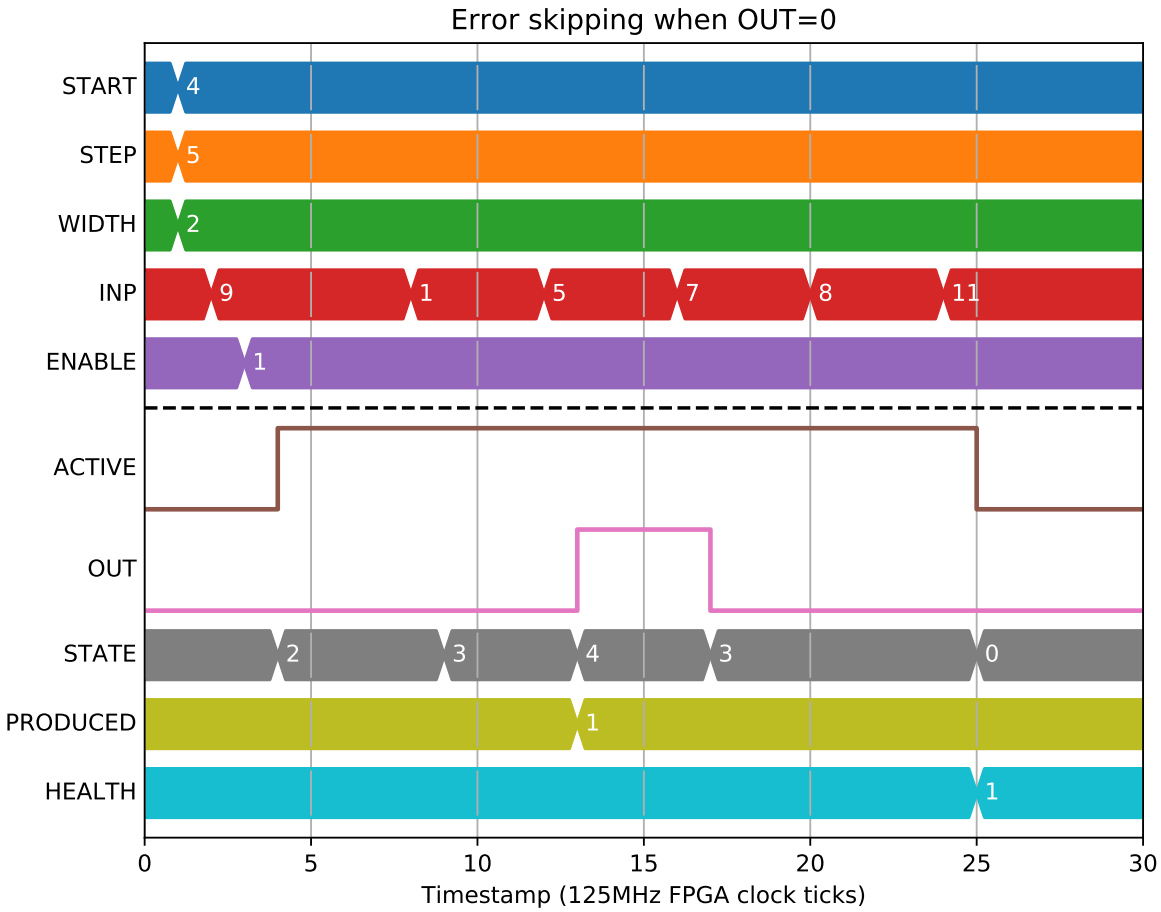
We can also make use of a special case with STEP=0 and a negative WIDTH to create a Schmitt trigger that will always trigger at START, and turn off when INP has dipped WIDTH below START:

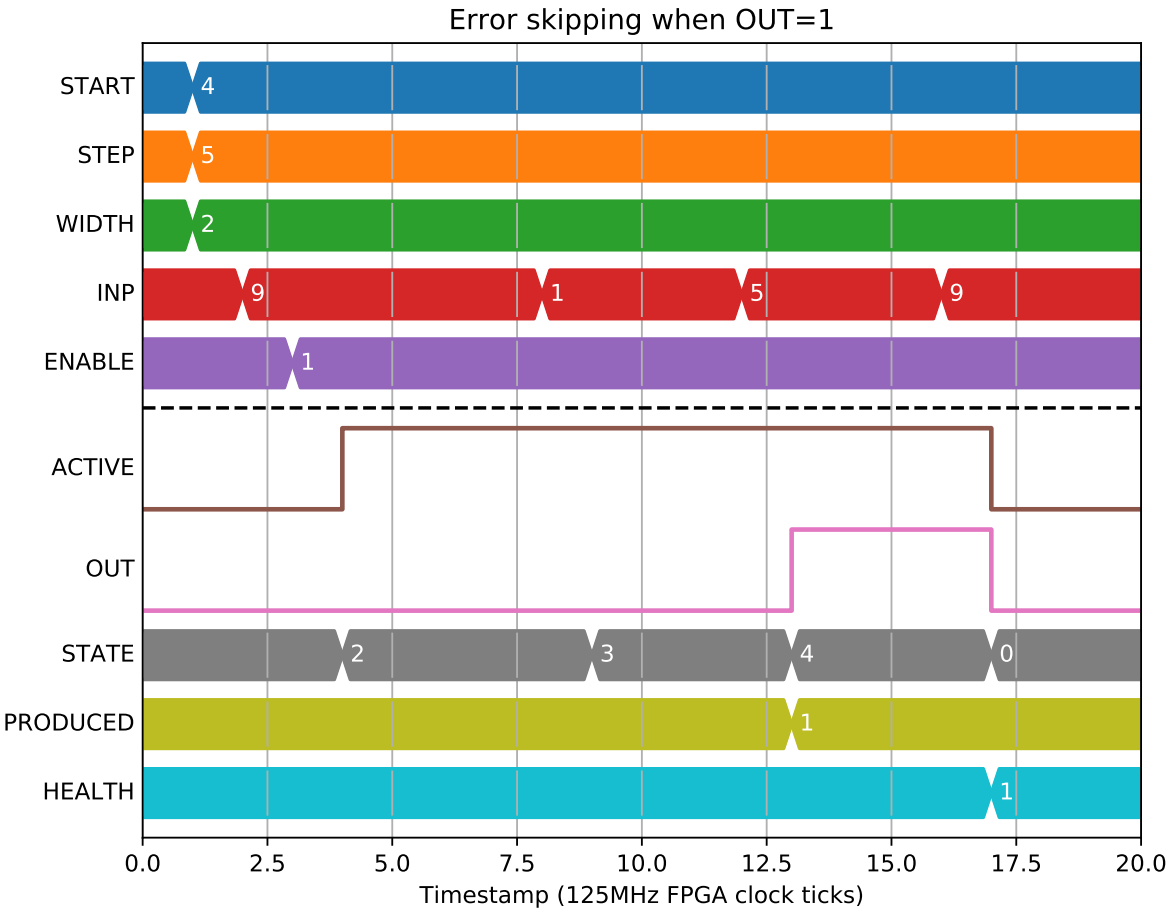
We can use this same special case with a positive width to make a similar comparator that turns on at START and off at START+WIDTH, triggering again when INP <= START:

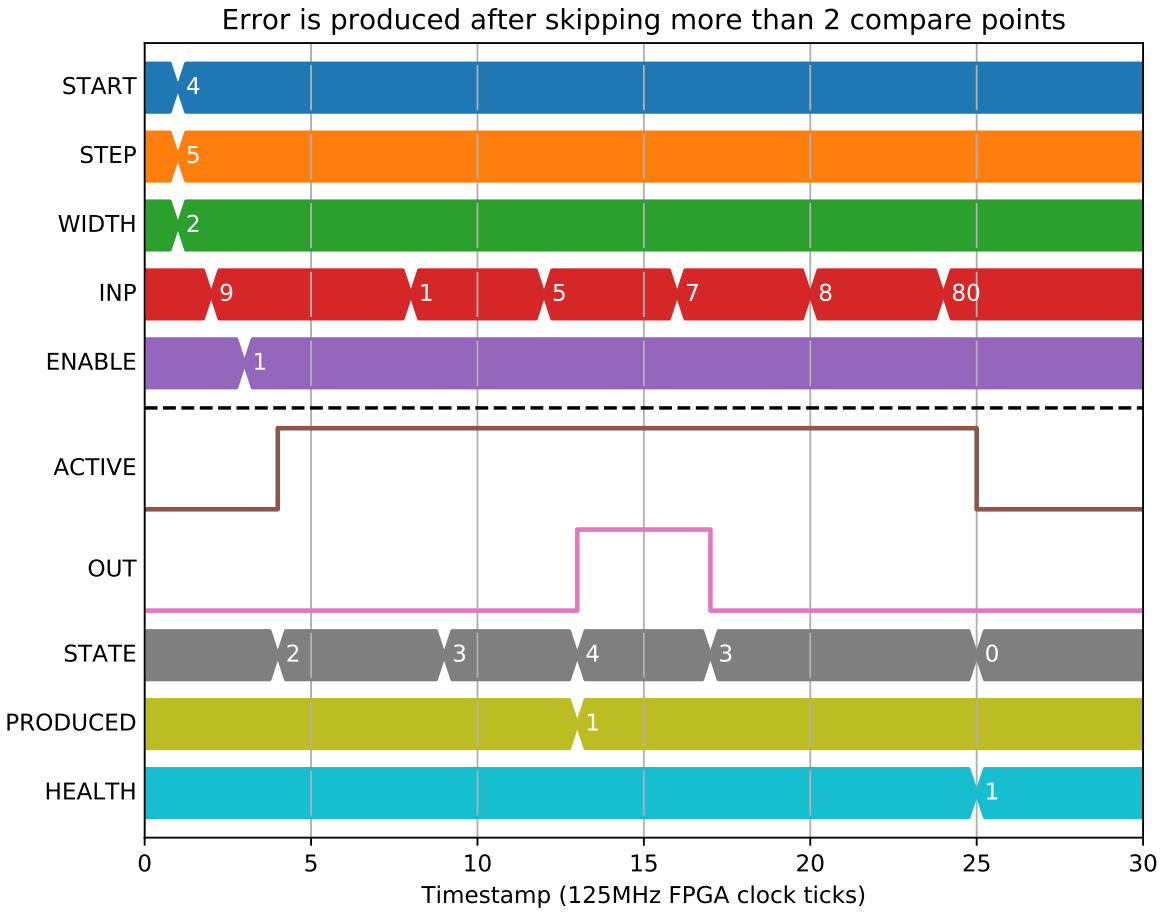
6.18 PGEN - Position Generator

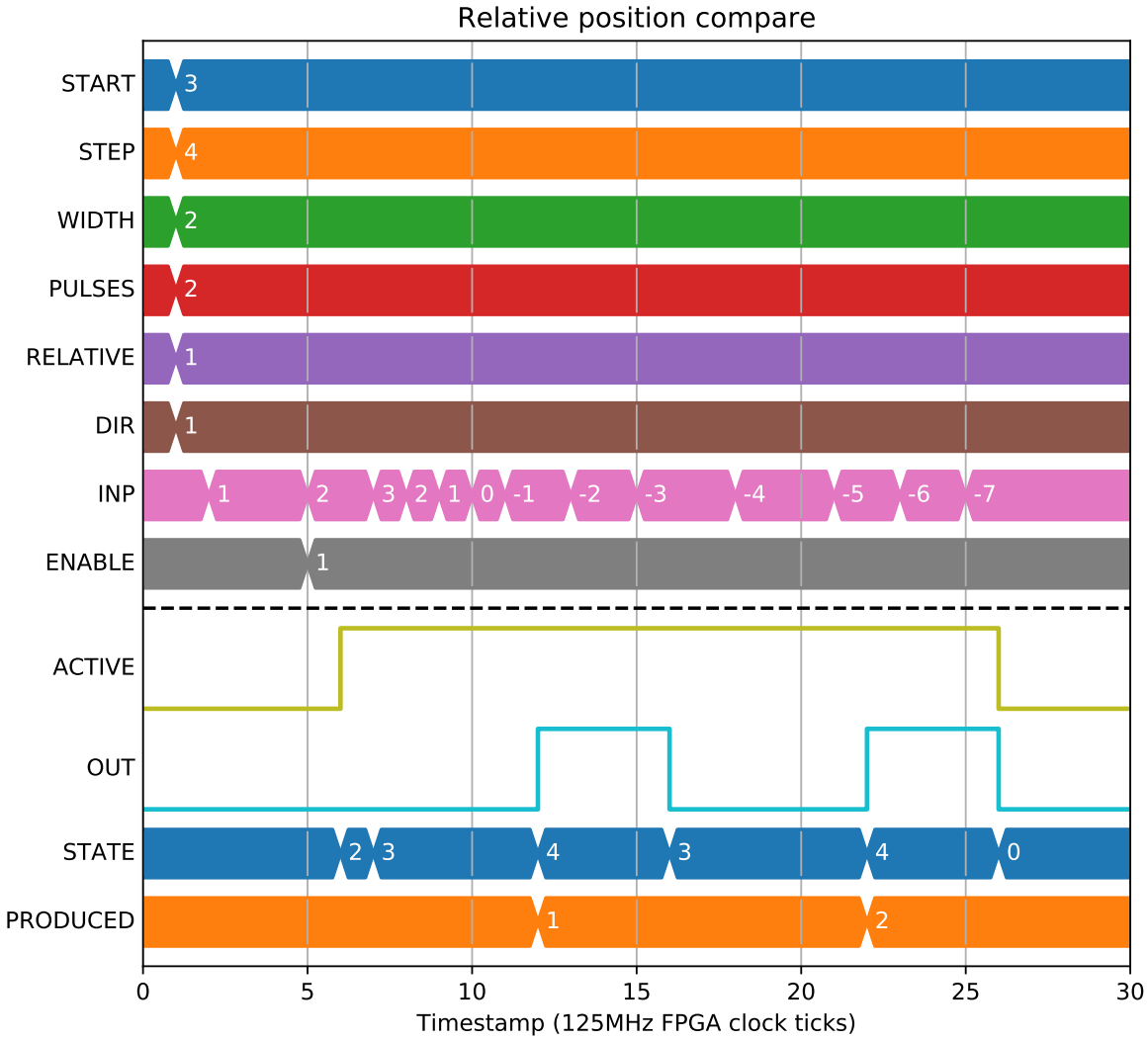
The position generator block produces an output position which is pre-defined in a table

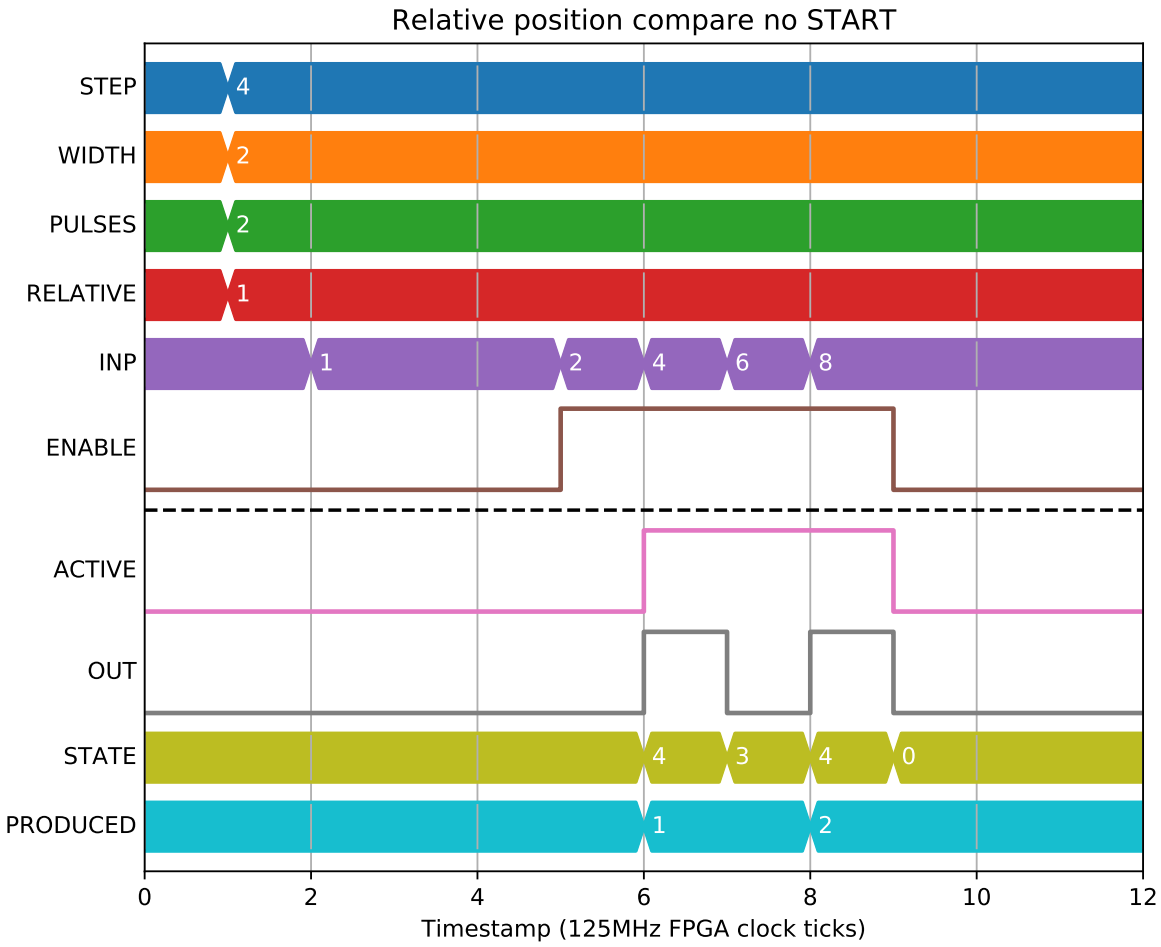

Absolute Pulses in a +ve direction
START
STEP
WIDTH
PULSES
INP
ENABLE
ACTIVE
OUT
STATE
PRODUCED
4
5
2
2
9
1
5
7
8
9
15
1
2
3
4
3
4
0
1
2
0
5
10
15
20
25
30
Timestamp (125MHz FPGA clock ticks)

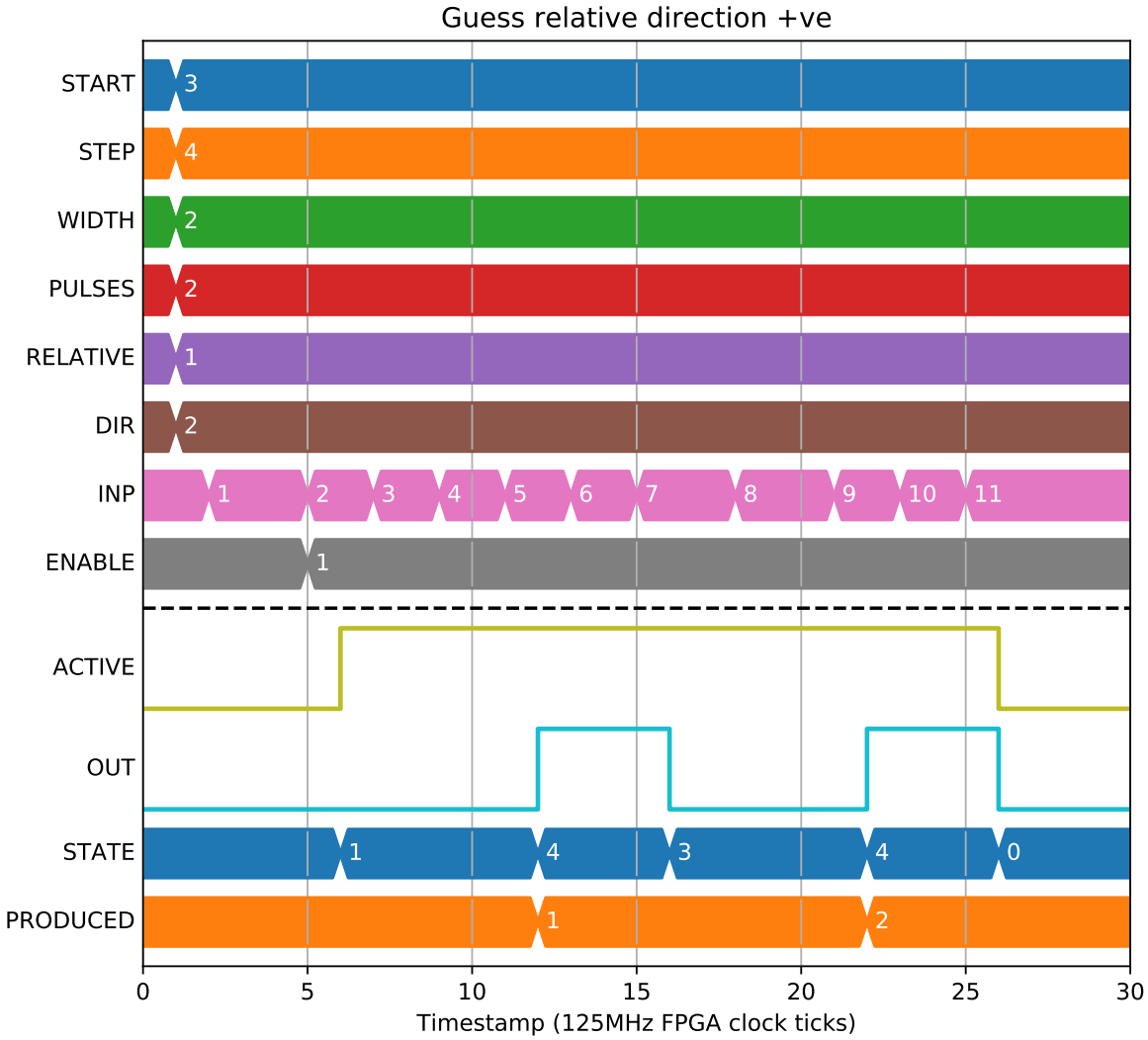

Error skipping when OUT=0
START
STEP
WIDTH
INP
ENABLE
ACTIVE
OUT
STATE
PRODUCED
HEALTH
4
5
2
9
1
5
7
8
11
1
2
3
4
3
0
1
1
0
5
10
15
20
25
30
Timestamp (125MHz FPGA clock ticks)

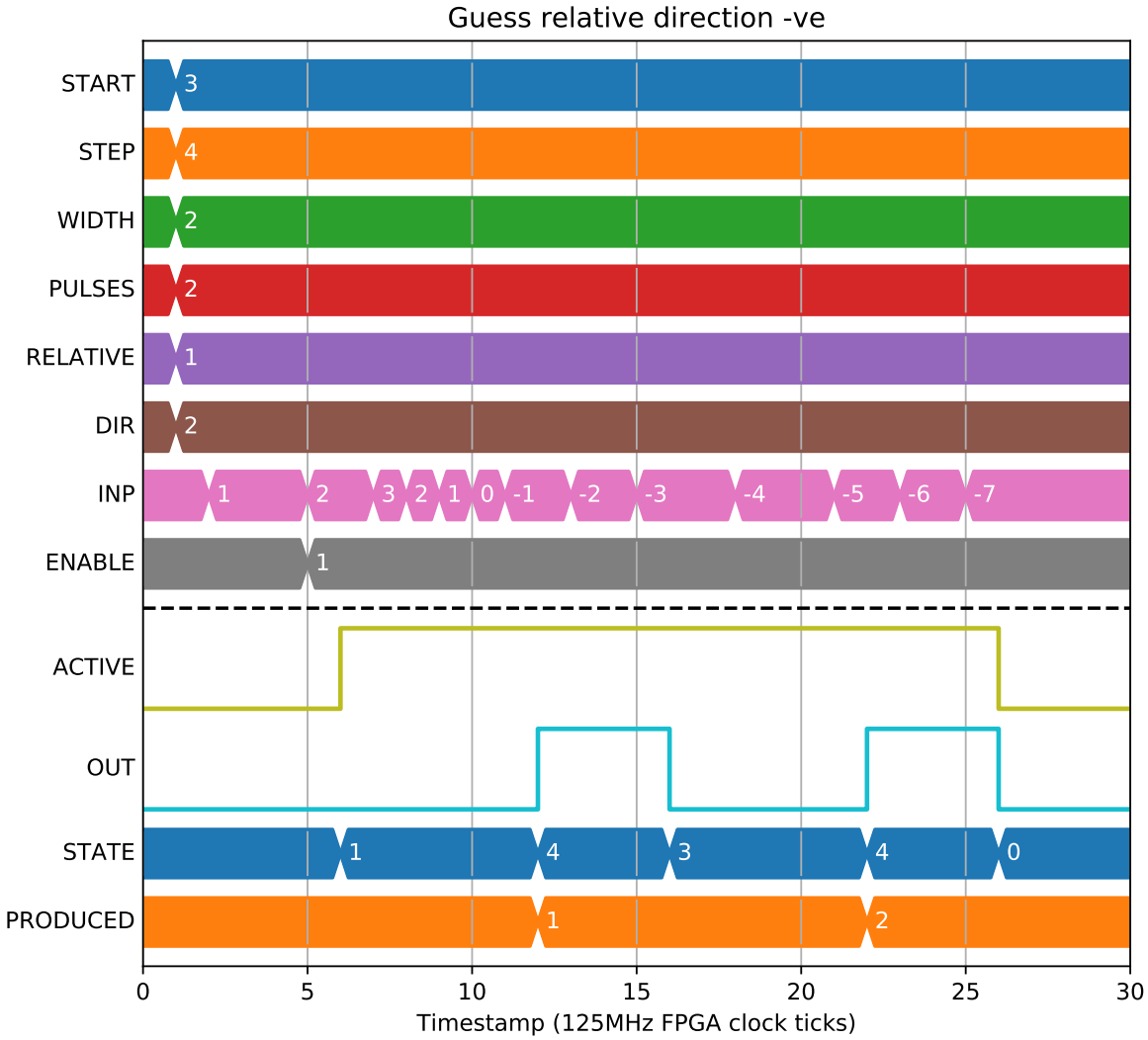

Error skipping when OUT=1
START
STEP
WIDTH
INP
ENABLE
ACTIVE
OUT
STATE
PRODUCED
HEALTH
4
5
2
9
1
5
9
1
2
3
4
0
1
1
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Timestamp (125MHz FPGA clock ticks)

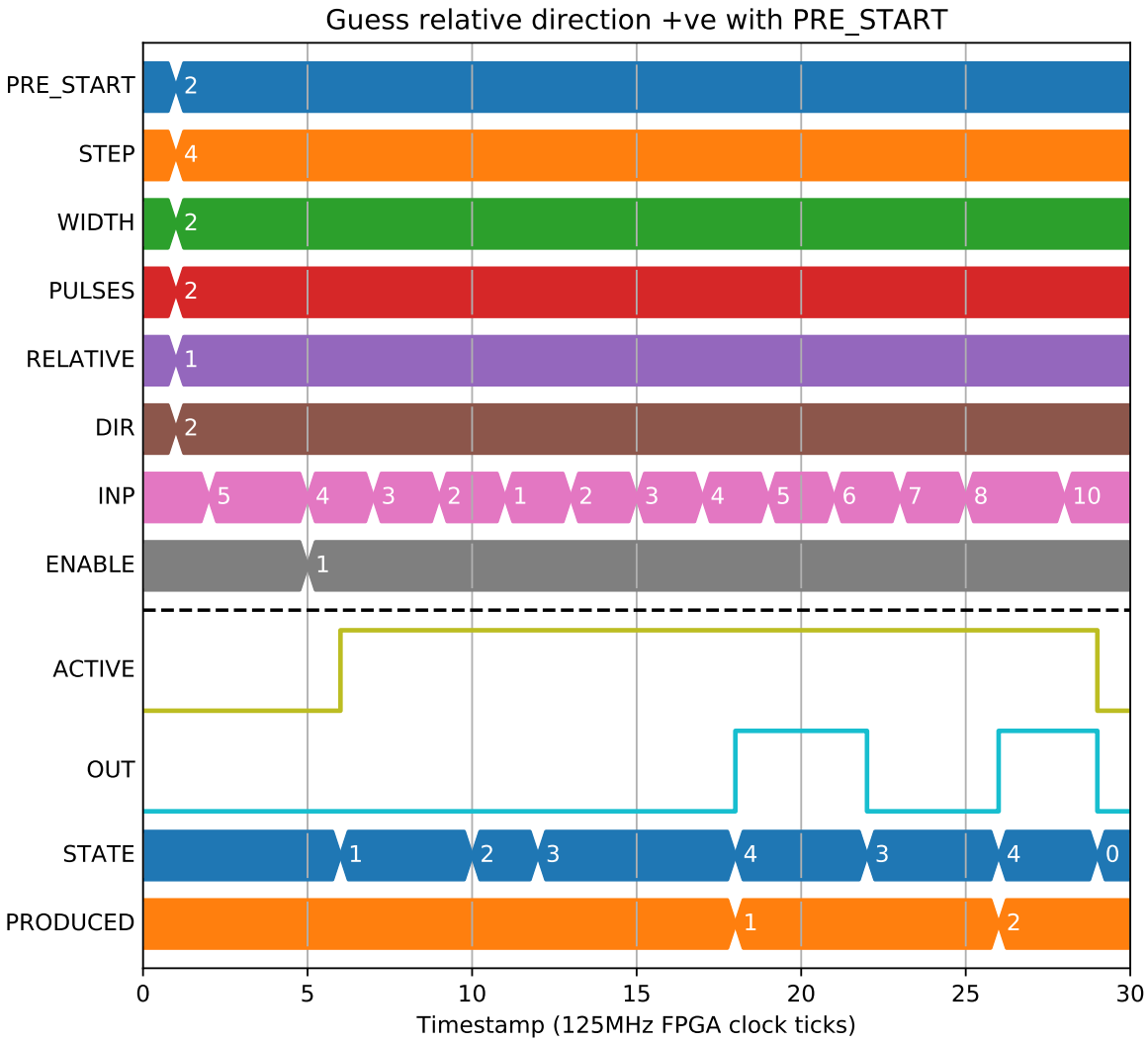

Error is produced after skipping more than 2 compare points
START
STEP
WIDTH
INP
ENABLE
ACTIVE
OUT
STATE
PRODUCED
HEALTH
4
5
2
9
1
5
7
8
80
1
2
3
4
3
0
1
1
0
5
10
15
20
25
30
Timestamp (125MHz FPGA clock ticks)

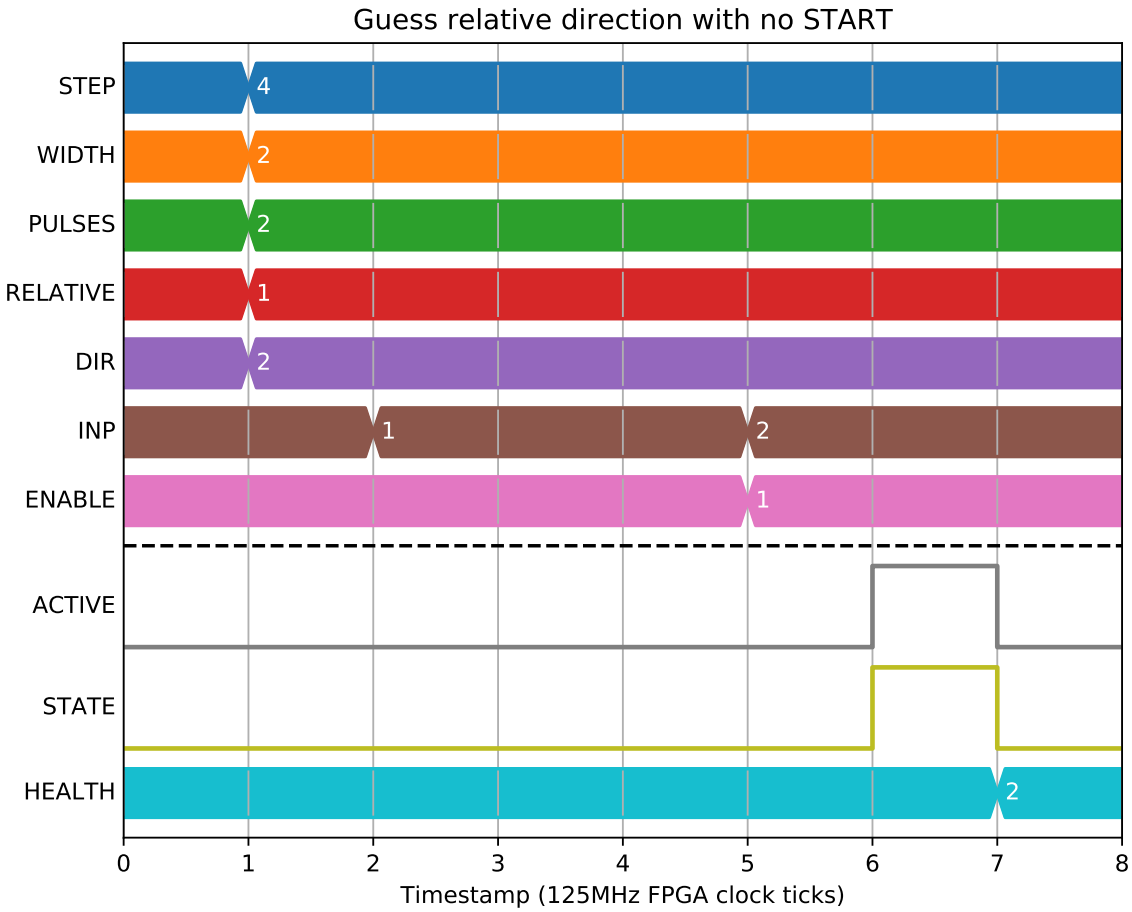

Relative position compare
START
STEP
WIDTH
PULSES
RELATIVE
DIR
INP
ENABLE
ACTIVE
OUT
STATE
PRODUCED
3
4
2
2
1
1
1 2 3 2 1 0 -1 -2 -3 -4 -5 -6 -7
1
2 3 4 3 4 0
1 2
0 5 10 15 20 25 30
Timestamp (125MHz FPGA clock ticks)

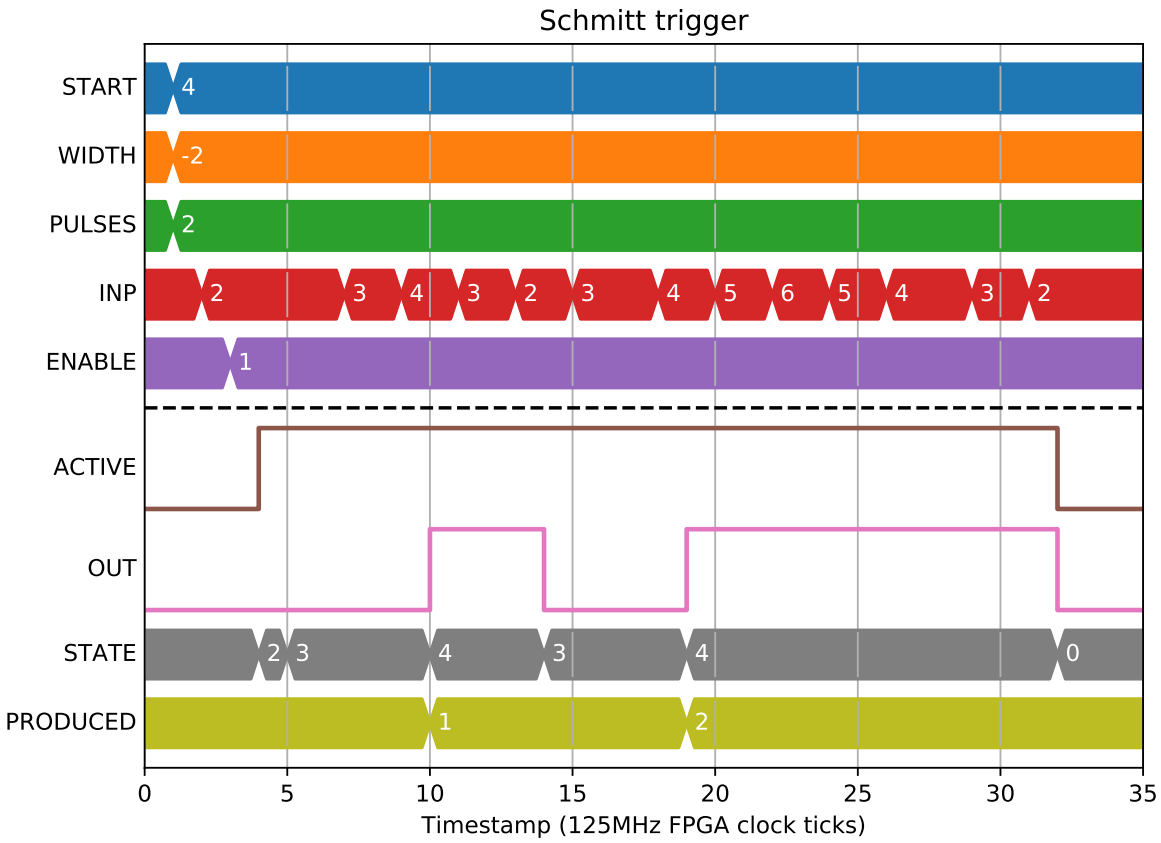

Relative position compare no START
STEP
WIDTH
PULSES
RELATIVE
INP
ENABLE
ACTIVE
OUT
STATE
PRODUCED
4
2
2
1
1
2
4
6
8
4
3
4
0
1
2
0
2
4
6
8
10
12
Timestamp (125MHz FPGA clock ticks)


Guess relative direction +ve
START
STEP
WIDTH
PULSES
RELATIVE
DIR
INP
ENABLE
ACTIVE
OUT
STATE
PRODUCED
0
5
10
15
20
25
30
Timestamp (125MHz FPGA clock ticks)


Guess relative direction -ve
START
STEP
WIDTH
PULSES
RELATIVE
DIR
INP
ENABLE
ACTIVE
OUT
STATE
PRODUCED
3
4
2
2
1
2
1 2 3 2 1 0 -1 -2 -3 -4 -5 -6 -7
1
1 4 3 4 0
1 2
0 5 10 15 20 25 30
Timestamp (125MHz FPGA clock ticks)

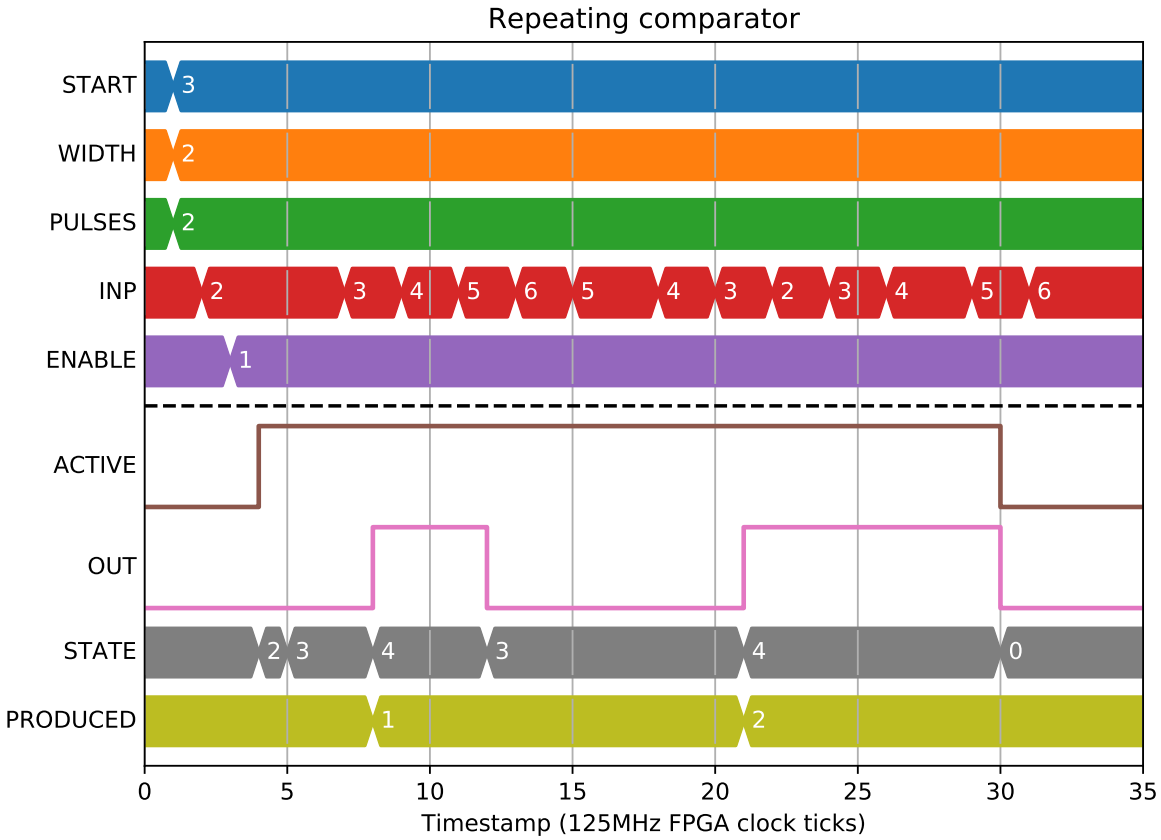

Guess relative direction +ve with PRE_START
PRE_START
STEP
WIDTH
PULSES
RELATIVE
DIR
INP
ENABLE
ACTIVE
OUT
STATE
PRODUCED
0
5
10
15
20
25
30
Timestamp (125MHz FPGA clock ticks)


Guess relative direction with no START
STEP
WIDTH
PULSES
RELATIVE
DIR
INP
ENABLE
ACTIVE
STATE
HEALTH
4
2
2
1
2
1
2
1
2
0
1
2
3
4
5
6
7
8
Timestamp (125MHz FPGA clock ticks)



6.18.1 Fields

Name	Type	Description
ENABLE	bit_mux	Halt on falling edge, reset and enable on rising
TRIG	bit_mux	Trigger a sample to be produced
TABLE	table	Table of positions to be output POSITION The position to set OUT to on trigger 31:0 POSITION int
REPEATS	param	Number of times the table will repeat
ACTIVE	bit_out	High when output is being produced from the table
OUT	pos_out	Current sample
HEALTH	read enum	Table status 0 OK 1 Table not ready 3 DMA overrun



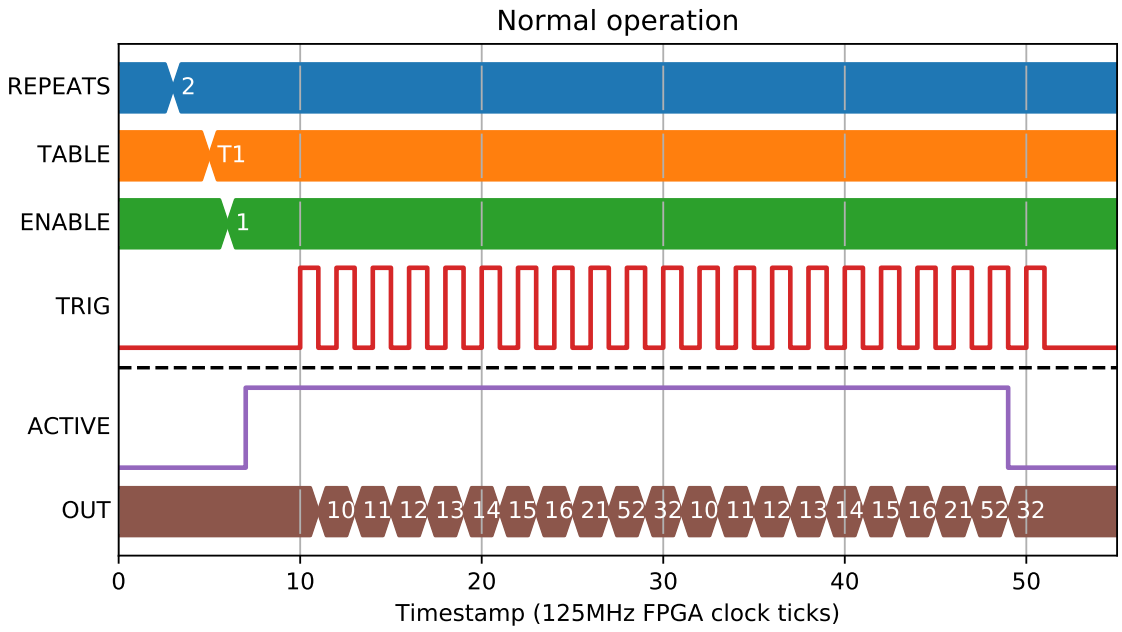
6.18.2 Normal operation

The output pulse will be generated regardless of the direction of the INP data

T1
POS
10
11
12
13
14
15
16
21
52
32

6.19 POSENC - Quadrature and step/direction encoder

The POSENC block handles the Quadrature and step/direction encoding



6.19.1 Fields

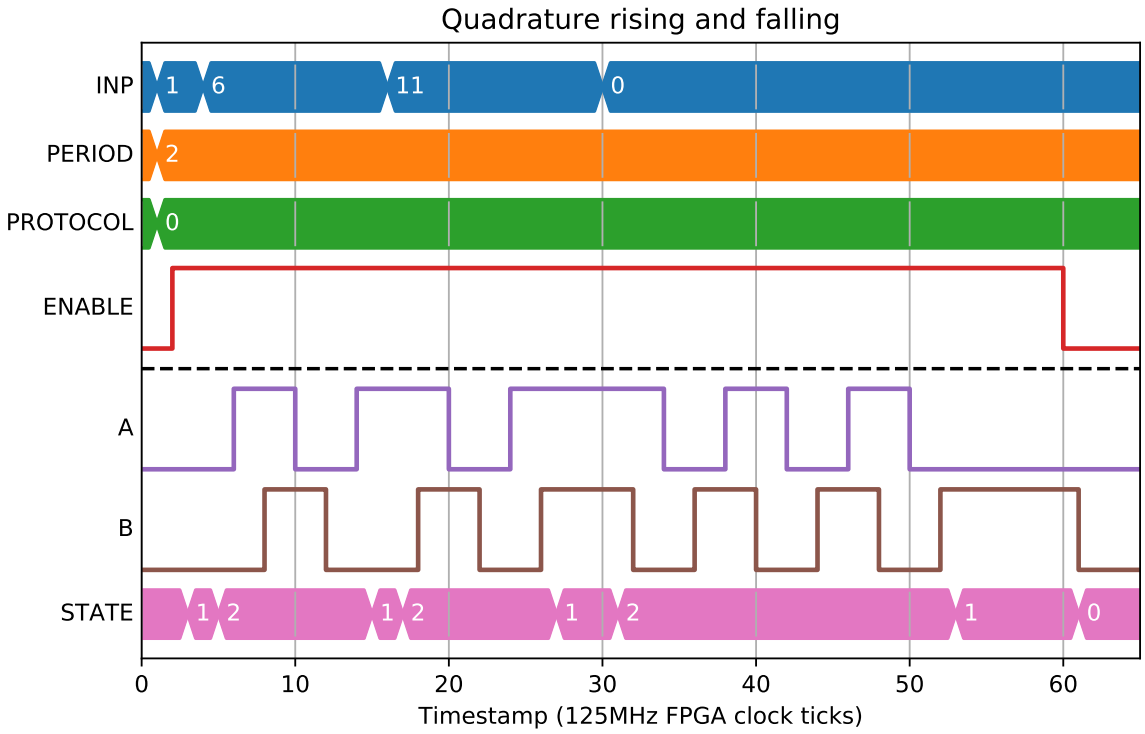
Name	Type	Description
ENABLE	bit_mux	Halt on falling edge, reset and enable on rising
INP	pos_mux	Output position
PERIOD	param time	Minimum time between Quadrature transitions of step pulses
PROTOCOL	param enum	Quadrature or step/direction 0 Quadrature 1 Step/Direction
A	bit_out	Quadrature A/Step output
B	bit_out	Quadrature B/Direction output
STATE	read enum	State of quadrature output 0 Disabled 1 At position 2 Slewing

6.19.2 Quadrature

When in the quadrature mode, the module will output signals A and B in different states as it counts up or down. When counting up B will follow A and when counting down A will follow B. The period is the time between an edge on one signal to the next edge of the other signal.

The input is initially set as the value of the INP line when ENABLE goes high. The system will then count to the current value on the INP line, and when it reaches this value the output signals will stay as they are.

The state output is '0' while ENABLE is low, '1' when the count is equal to the signal on the INP line and '2' while it is counting towards the INP value.



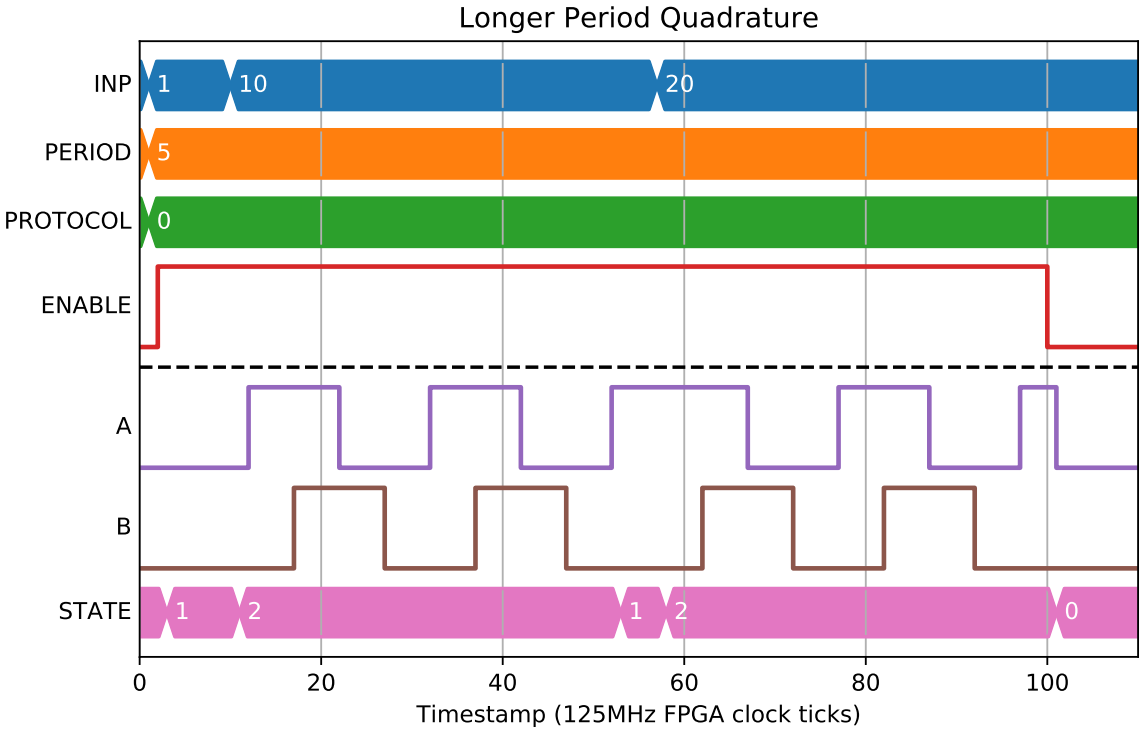
6.19.3 Step/Direction

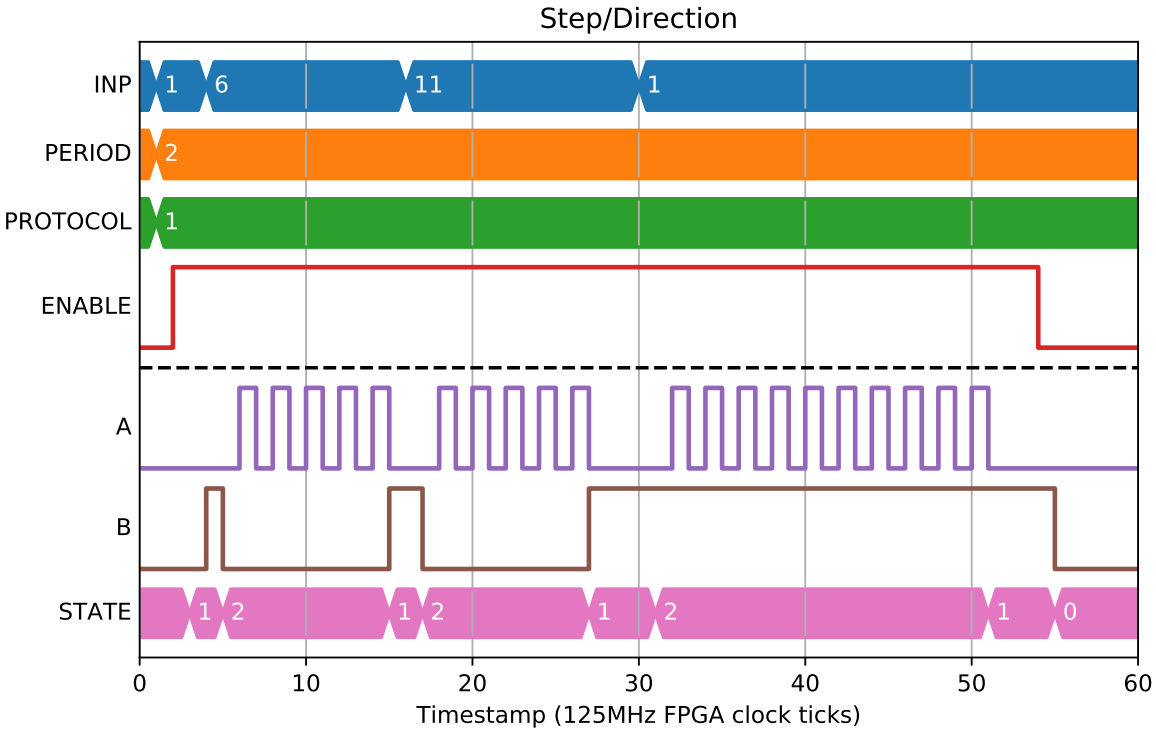
In the Step/Direction mode the A output becomes a step output. This goes high on every period for one clock cycle and is low for the remainder of the period. The B output becomes the direction output, it is '0' when the internal counter is lower than the inputted target value (it is counting up), and '1' when it is greater or equal to.

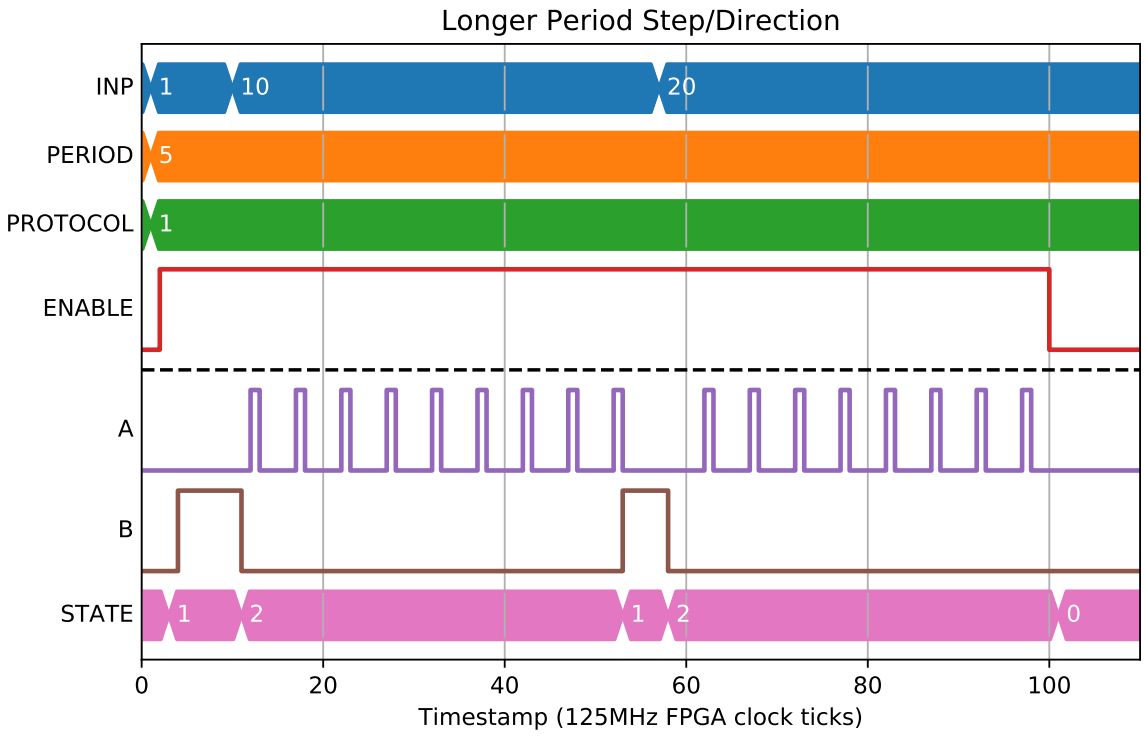
6.20 PULSE - One-shot pulse delay and stretch

A PULSE block produces configurable width output pulses with an optional delay based on its parameters. It operates in one of two modes:

- If WIDTH=0, then it acts as a delay line. The input pulse train will just be replayed after the given DELAY
- If WIDTH is non-zero, then each pulse edge that matches TRIG_EDGE will be delayed by the specified DELAY, then generate NPULSES pulses of width WIDTH, with rising edges separated by STEP


Longer Period Quadrature
INP
PERIOD
PROTOCOL
ENABLE
A
B
STATE
1
10
20
5
0
1
2
1
2
0
0
20
40
60
80
100
Timestamp (125MHz FPGA clock ticks)


Step/Direction
INP
PERIOD
PROTOCOL
ENABLE
A
B
STATE
1
6
11
1
2
1
1
2
1
2
1
2
1
0
0
10
20
30
40
50
60
Timestamp (125MHz FPGA clock ticks)



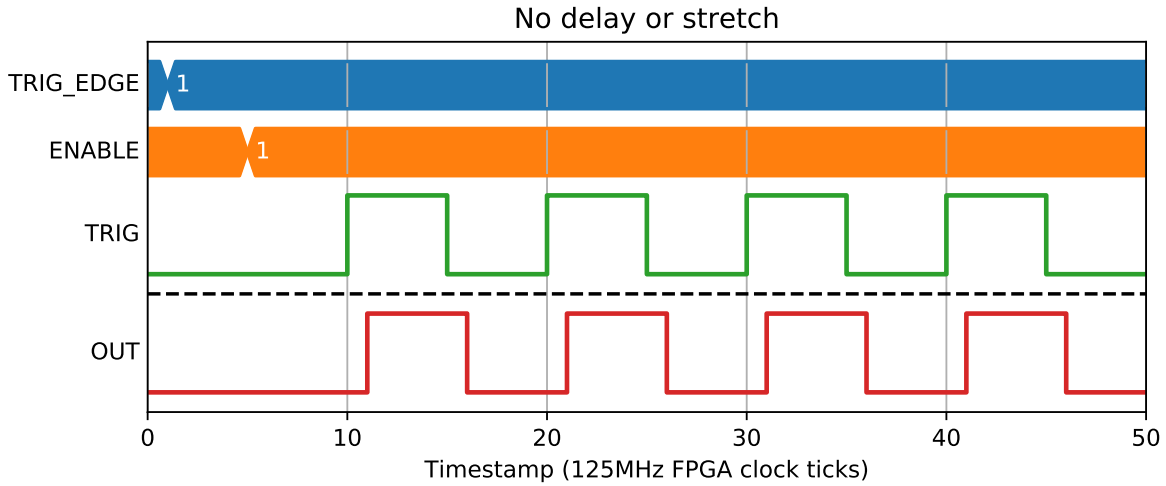
6.20.1 Fields

Name	Type	Description
ENABLE	bit_mux	Reset on falling edge, enable on rising
TRIG	bit_mux	Input pulse train
DELAY	time	Output pulse delay (0 for no delay)
WIDTH	time	Output pulse width (0 for input pulse width)
PULSES	param	The number of pulses to produce on each trigger, 0 means 1
STEP	time	If pulses > 1, the time between successive pulse rising edges
TRIG_EDGE	param enum	INP trigger edge 0 Rising 1 Falling 2 Either
OUT	bit_out	Output pulse train
QUEUED	read uint 1023	Length of the delay queue
DROPPED	read	Number of pulses not produced because of an ERR condition

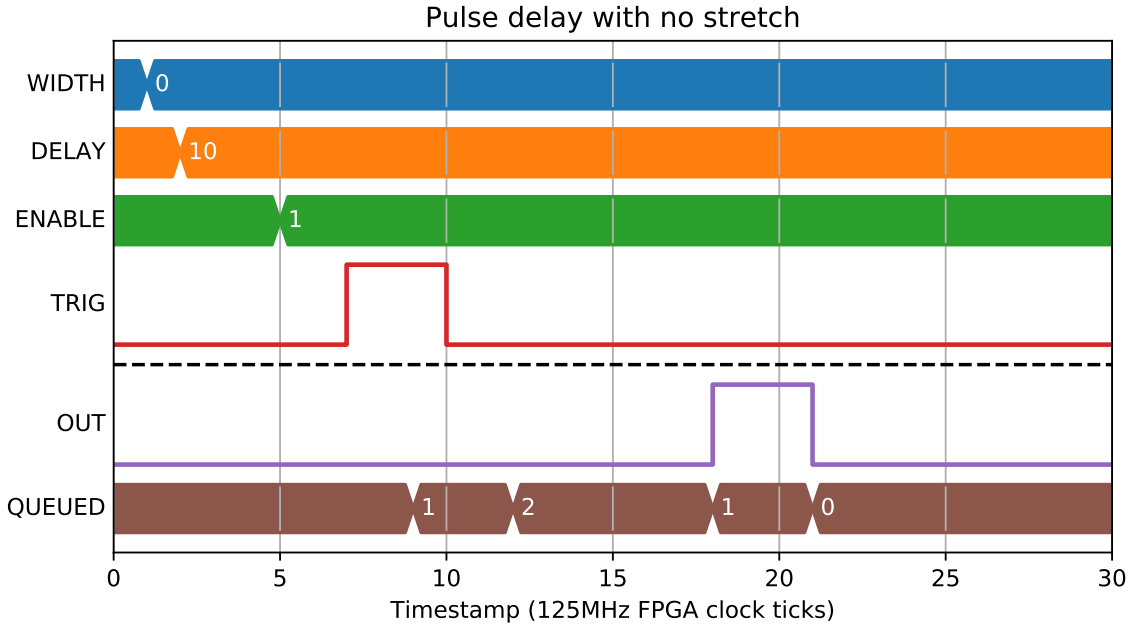
6.20.2 Delay line

If WIDTH=0, then the Block acts as a delay line. DELAY must either be 0 or 5+ clock ticks. TRIG_EDGE, STEP, and NPULSES are ignored.

If DELAY=0 the Block is a simple pass through:



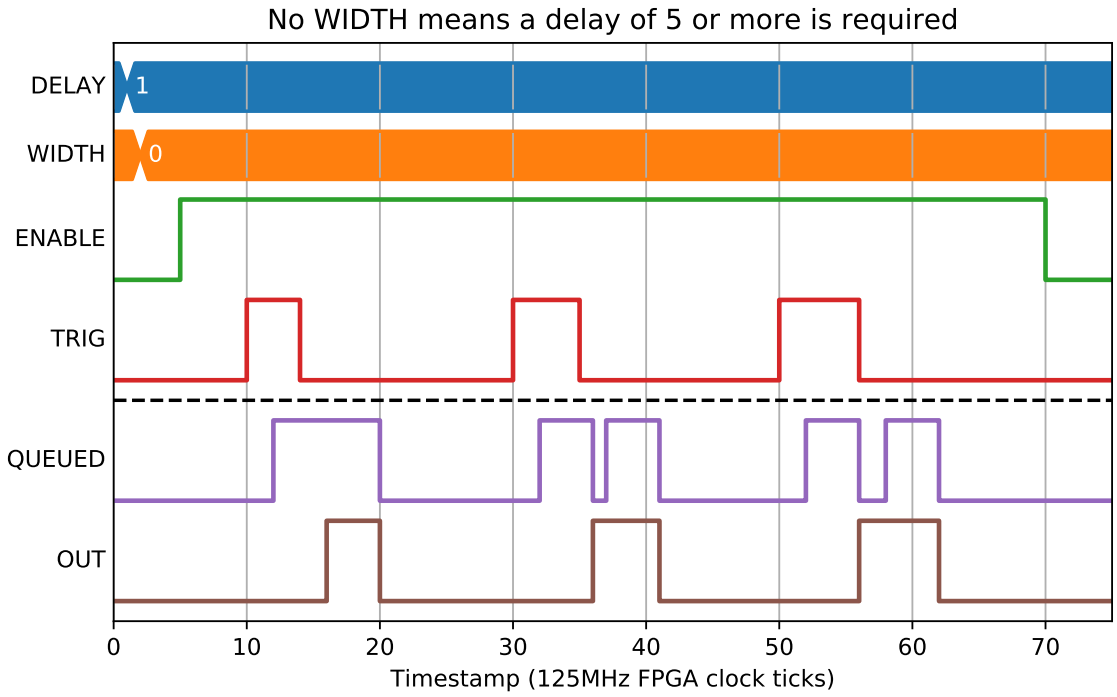
If DELAY is non-zero, rising and falling edges will be inserted in the queue and output after the given DELAY:

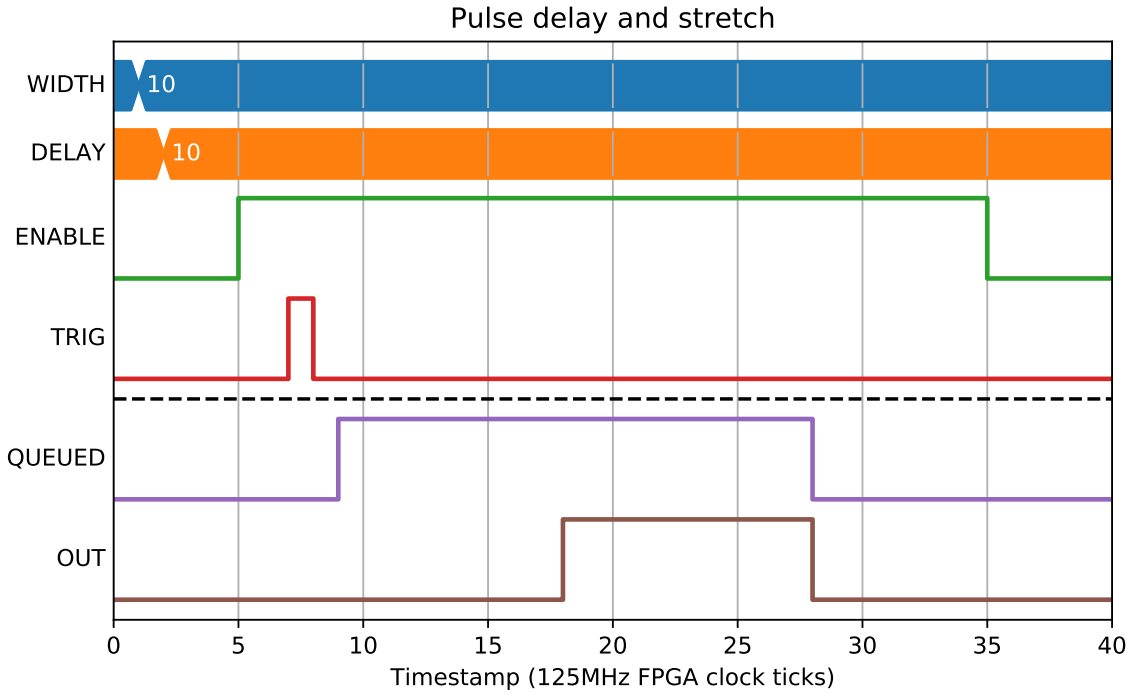


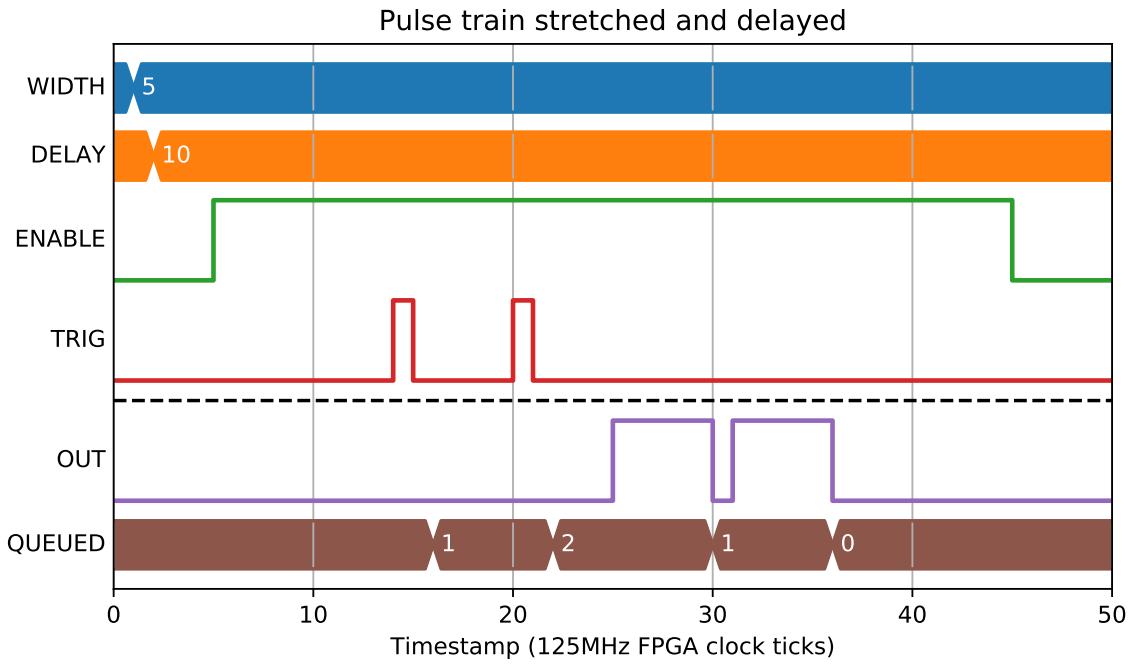
0 < DELAY < 5 will be treated as DELAY=5:

6.20.3 Pulse train generation

If WIDTH != 0 then the Block will operate in pulse train mode. If NPULSES is 0 or 1 then it will produce a single pulse for each matching input pulse:


No WIDTH means a delay of 5 or more is required
DELAY
1
WIDTH
0
ENABLE
TRIG
QUEUED
OUT
0
10
20
30
40
50
60
70
Timestamp (125MHz FPGA clock ticks)


Pulse delay and stretch
WIDTH
10
DELAY
10
ENABLE
TRIG
QUEUED
OUT
0
5
10
15
20
25
30
35
40
Timestamp (125MHz FPGA clock ticks)



The output pulses are queued, so multiple pulses can be queued before output:

The TRIG_EDGE field can be used to select whether an input pulse queues an output on rising, falling, or both edges:

0 < WIDTH < 5 will be treated as WIDTH=5:

If PULSES > 1 then multiple output pulses will be generated, separated by STEP:

6.20.4 Pulse period error

The following example shows what happens when the period between pulses is too short. To avoid running output pulses together, the DROPPED field is incremented and the input is dropped:

The queue length is 255, so if QUEUED reaches 255 then any new pulse will be dropped and also increment DROPPED.

The DROPPED count is zeroed on rising edge of ENABLE.

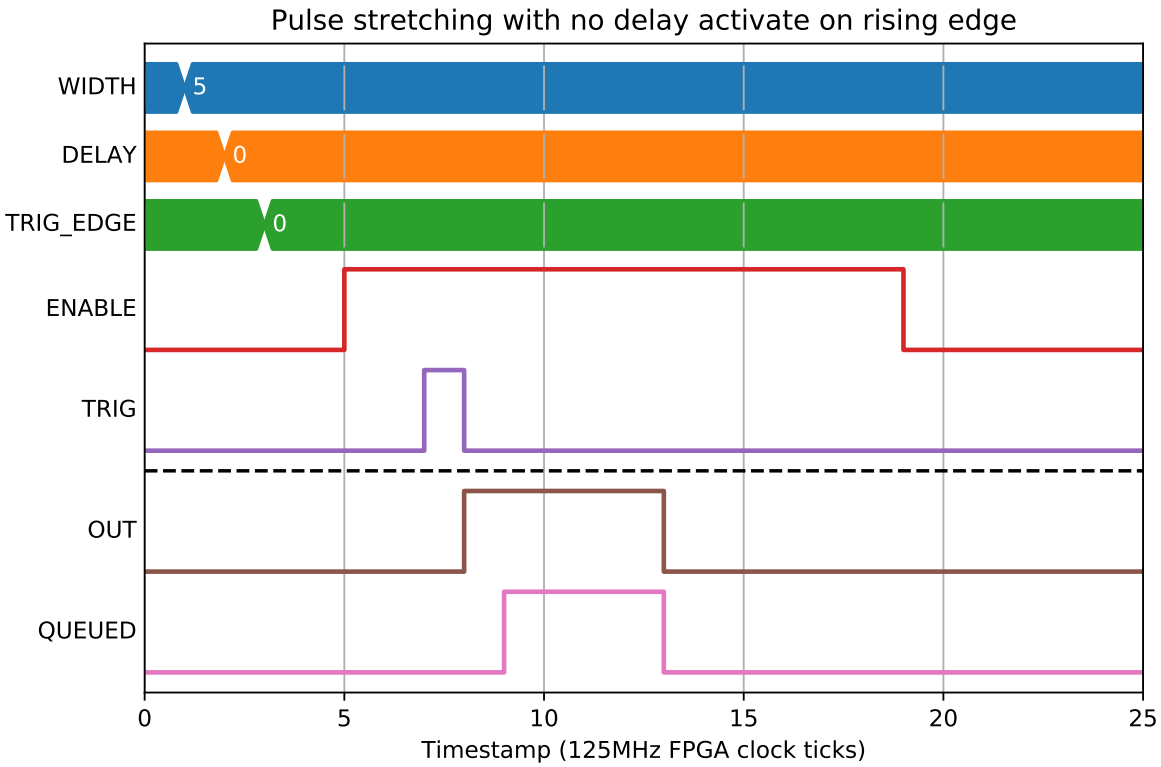
6.20.5 Enabling the Block

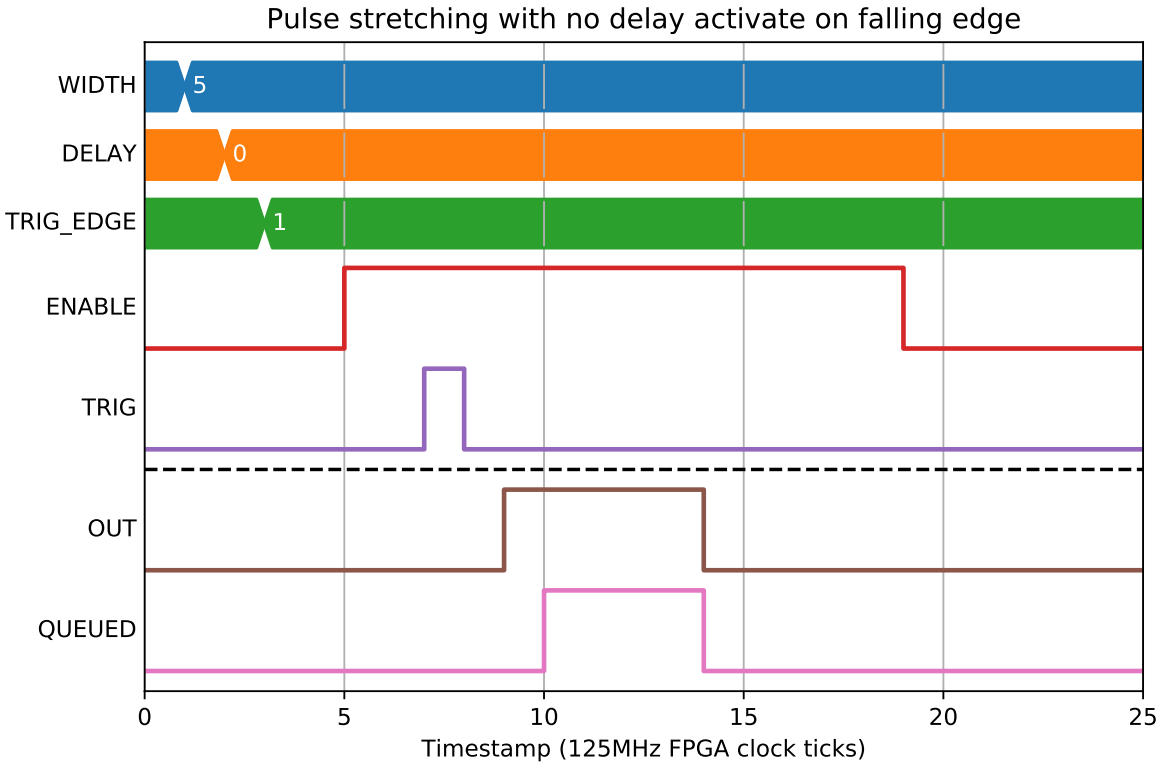
There is an Enable signal that stops the Block from producing signals. Edges must occur while Enable is high to trigger a pulse creation

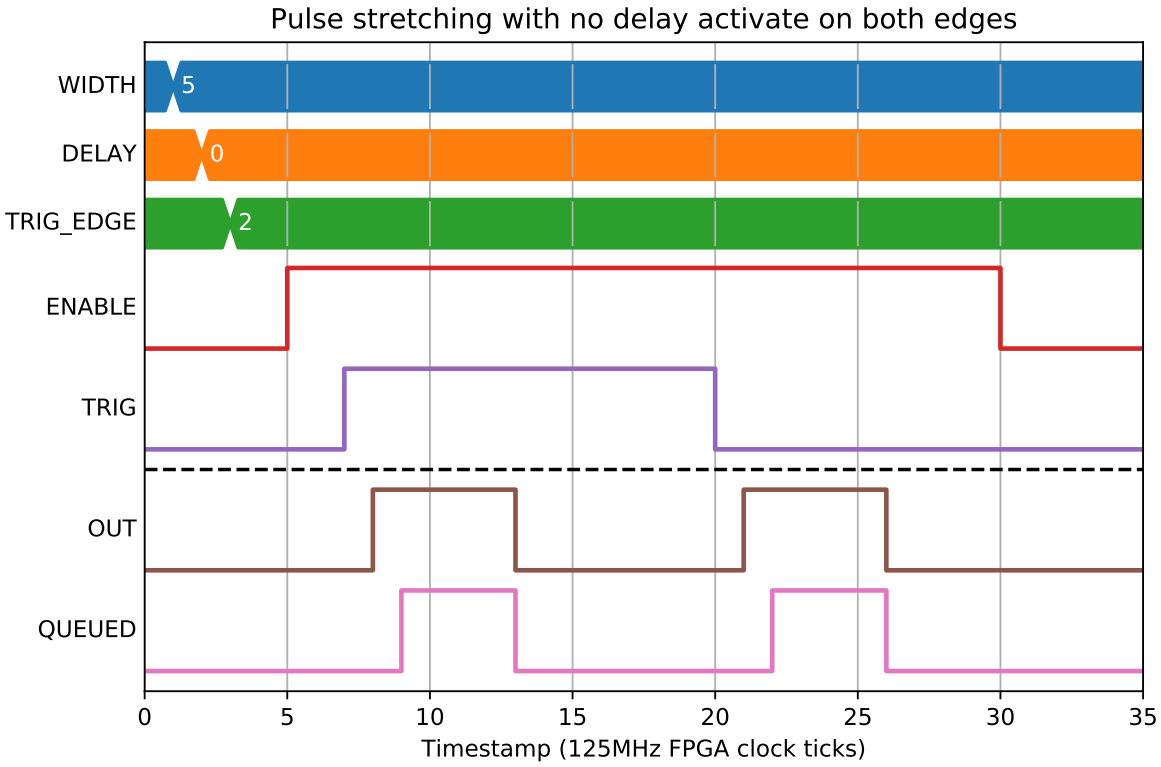
If enable is dropped mid way through a pulse train, the output is set low and the QUEUED output is set to zero.

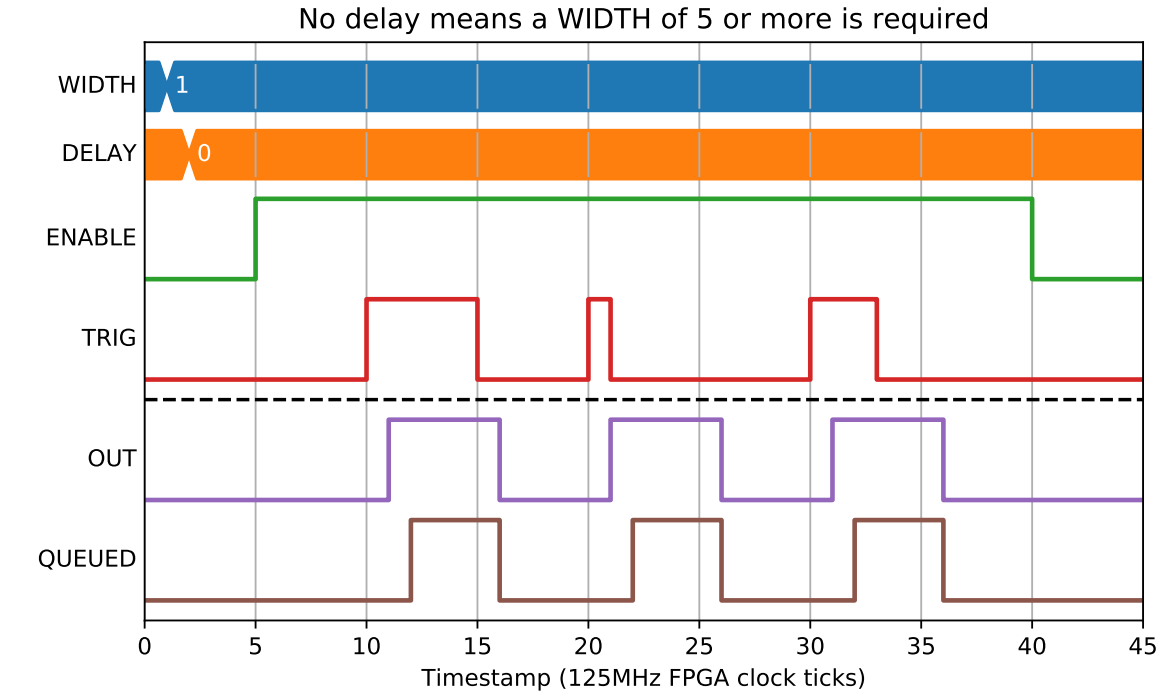
6.20.6 Changing parameters while Enabled

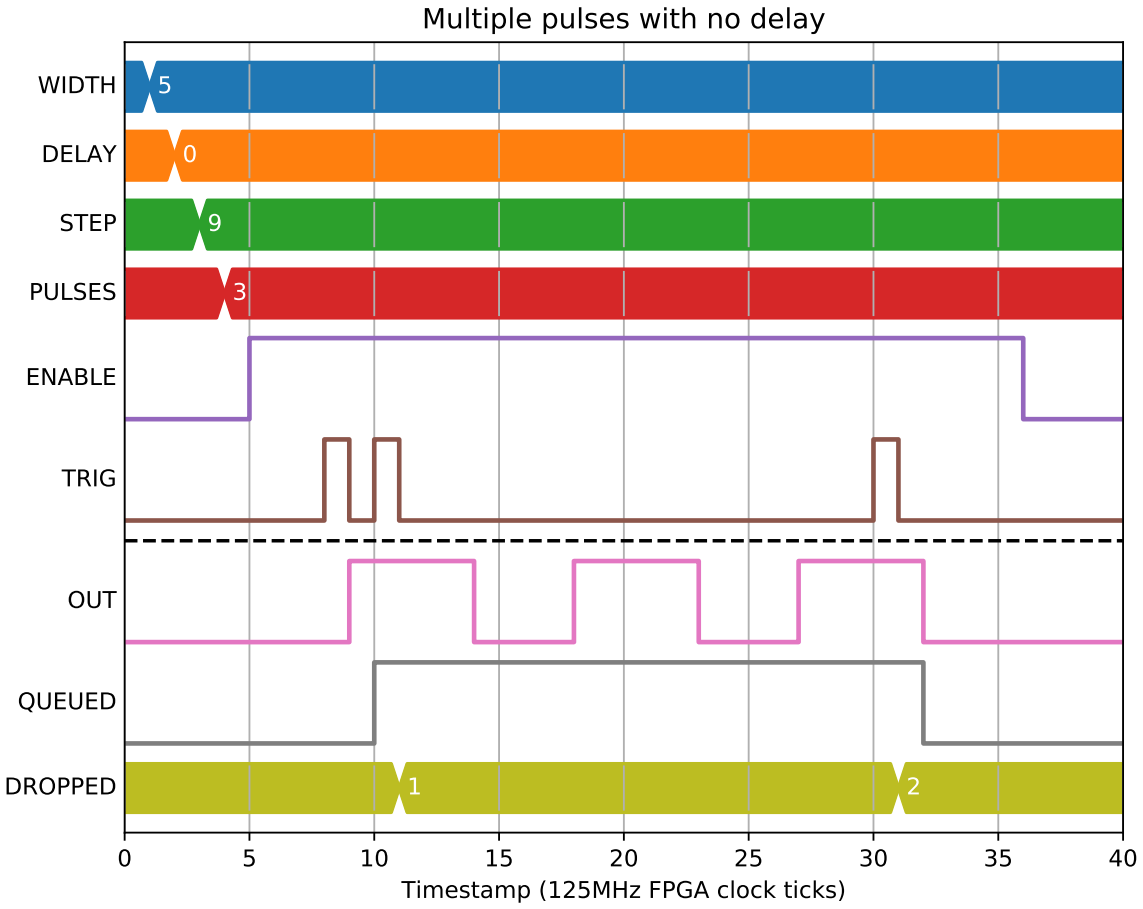
If any of the input parameters are changed while enabled, the queue is dropped and the state of the Block is reset:

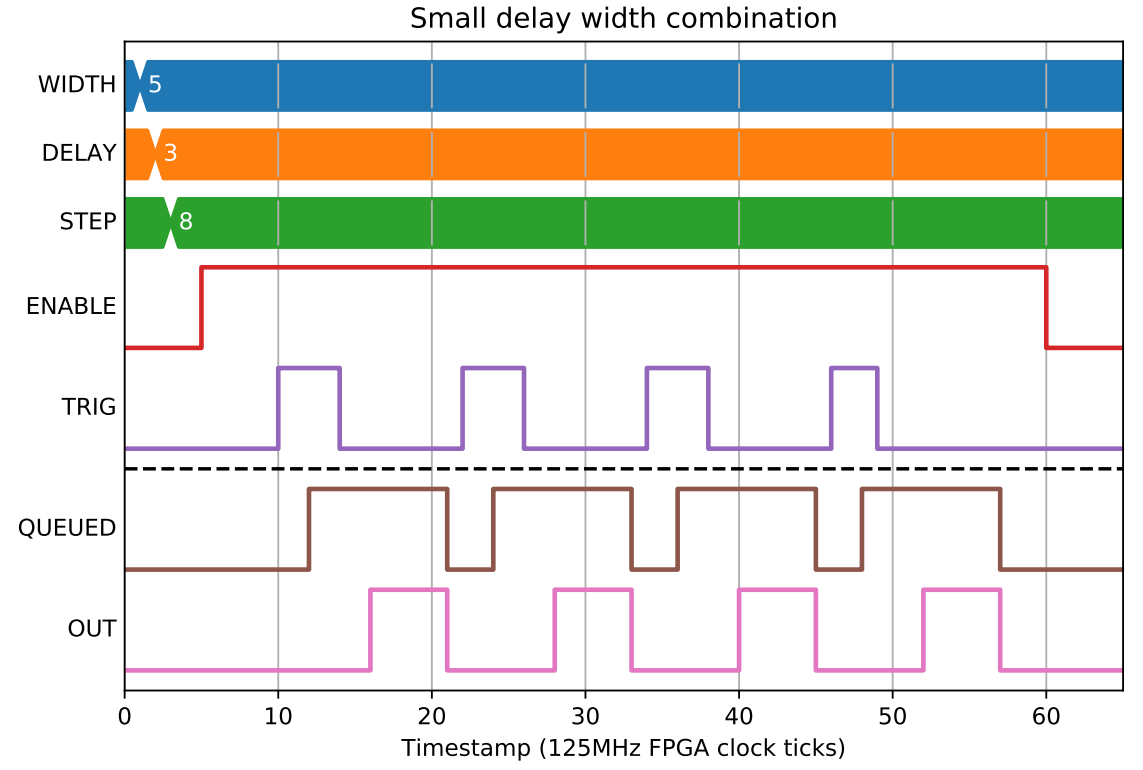

Pulse stretching with no delay activate on rising edge
WIDTH
5
DELAY
0
TRIG_EDGE
0
ENABLE
TRIG
OUT
QUEUED
0
5
10
15
20
25
Timestamp (125MHz FPGA clock ticks)

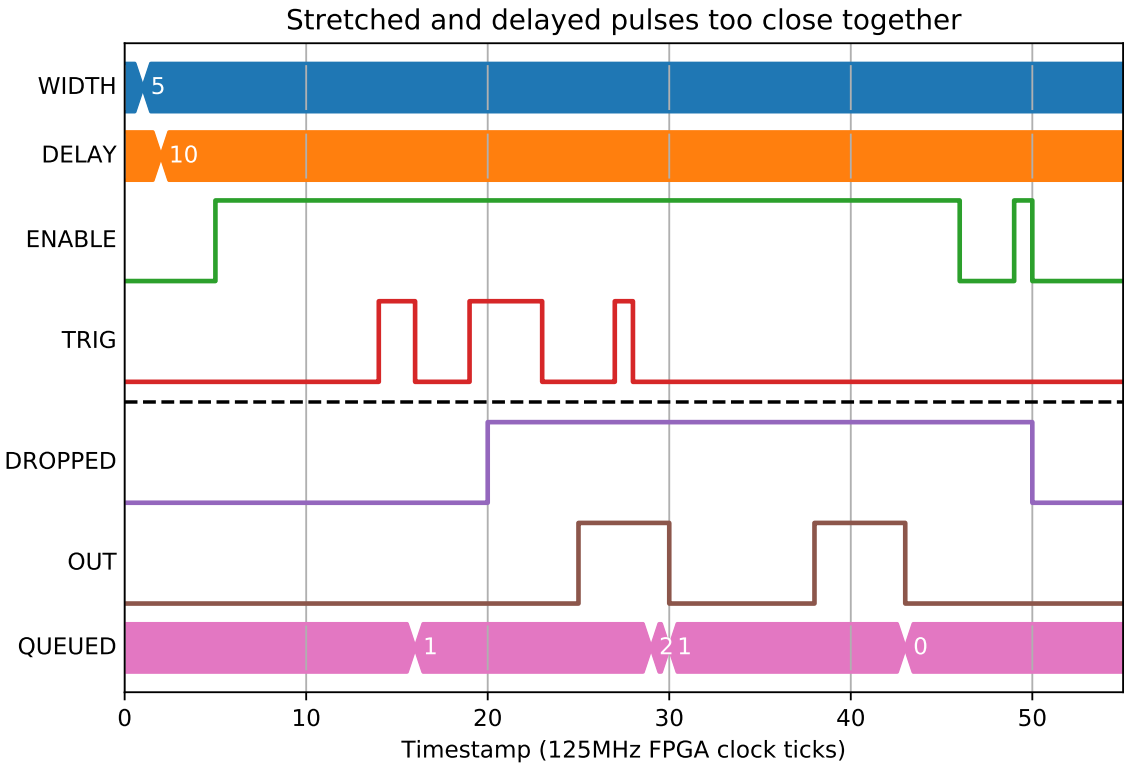

Pulse stretching with no delay activate on falling edge
WIDTH
5
DELAY
0
TRIG_EDGE
1
ENABLE
TRIG
OUT
QUEUED
0
5
10
15
20
25
Timestamp (125MHz FPGA clock ticks)

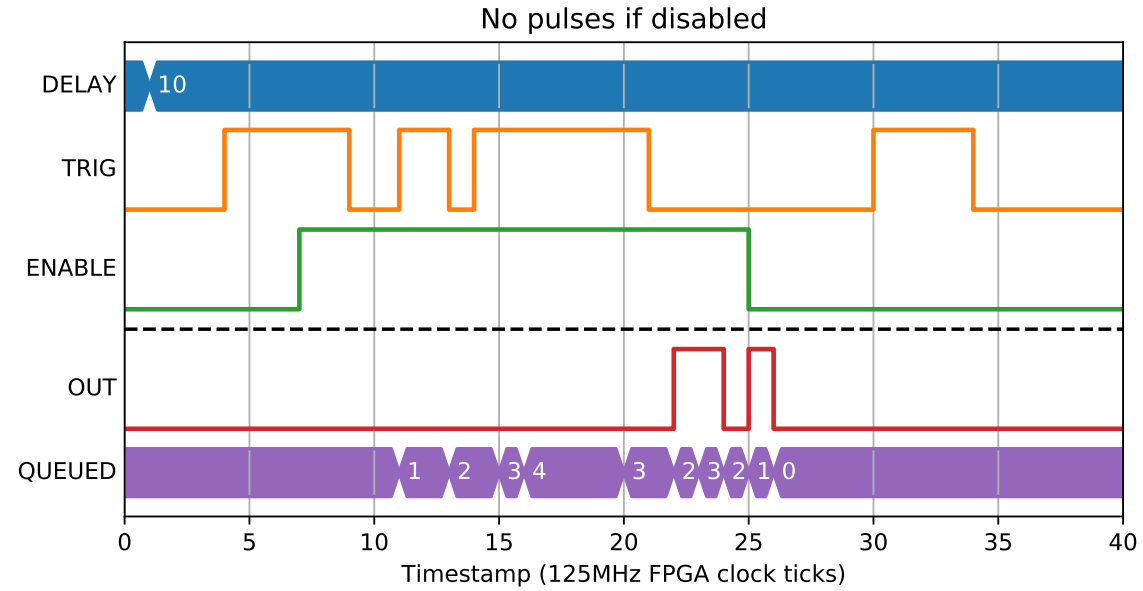

Pulse stretching with no delay activate on both edges
WIDTH
5
DELAY
0
TRIG_EDGE
2
ENABLE
TRIG
OUT
QUEUED
0
5
10
15
20
25
30
35
Timestamp (125MHz FPGA clock ticks)

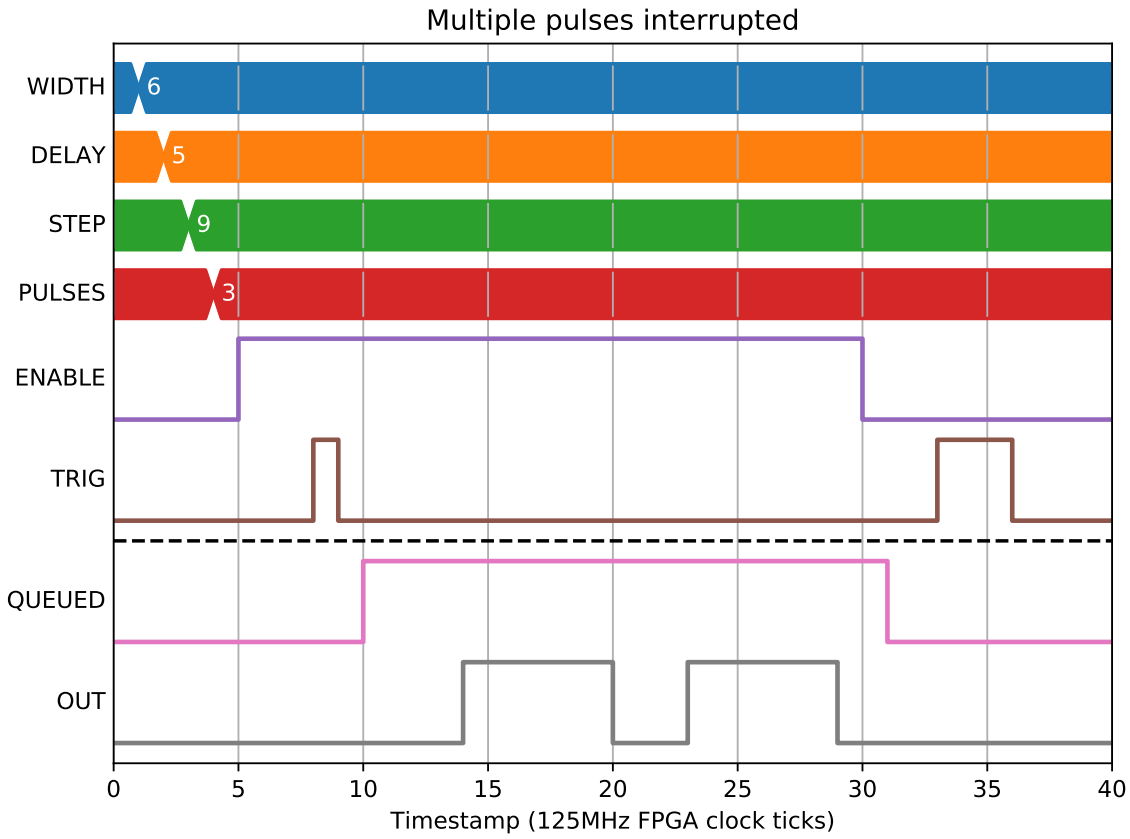

No delay means a WIDTH of 5 or more is required
WIDTH
1
DELAY
0
ENABLE
TRIG
OUT
QUEUED
0
5
10
15
20
25
30
35
40
45
Timestamp (125MHz FPGA clock ticks)

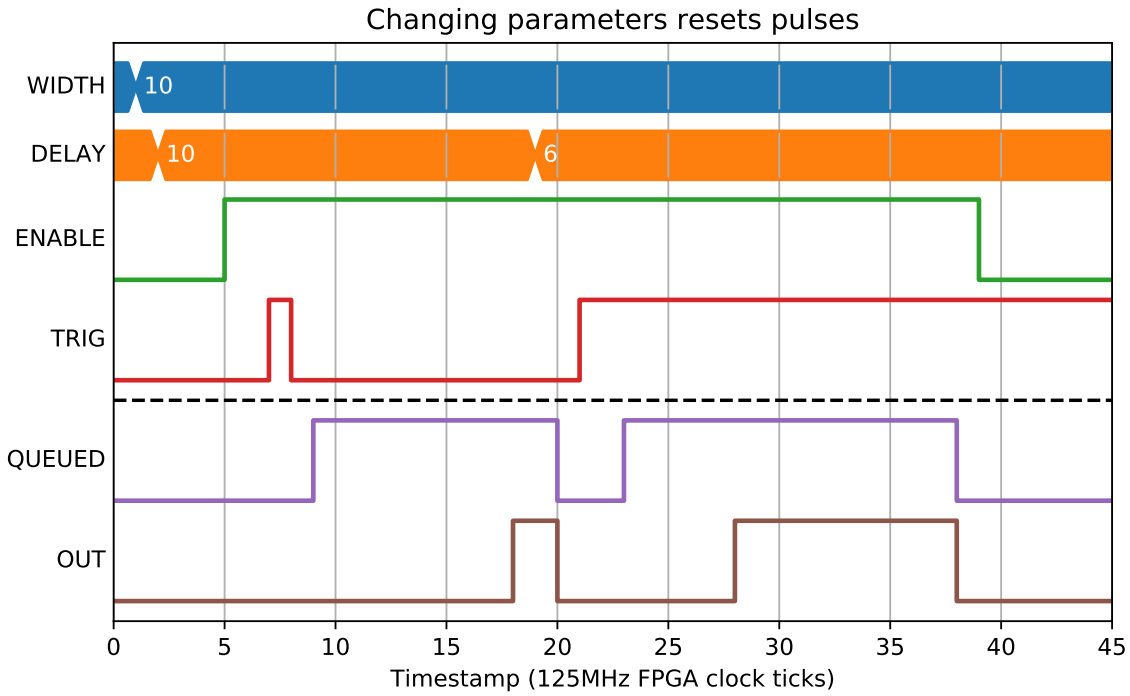

Multiple pulses with no delay
WIDTH
DELAY
STEP
PULSES
ENABLE
TRIG
OUT
QUEUED
DROPPED
5
0
9
3
1
2
0
5
10
15
20
25
30
35
40
Timestamp (125MHz FPGA clock ticks)


Small delay width combination
WIDTH
DELAY
STEP
ENABLE
TRIG
QUEUED
OUT
5
3
8
0
10
20
30
40
50
60
Timestamp (125MHz FPGA clock ticks)


Stretched and delayed pulses too close together
WIDTH
DELAY
ENABLE
TRIG
DROPPED
OUT
QUEUED
5
10
1
2 1
0
0
10
20
30
40
50
Timestamp (125MHz FPGA clock ticks)


No pulses if disabled
DELAY
TRIG
ENABLE
OUT
QUEUED
10
1
2
3 4
3
2 3 2 1 0
0
5
10
15
20
25
30
35
40
Timestamp (125MHz FPGA clock ticks)


Multiple pulses interrupted
WIDTH
DELAY
STEP
PULSES
ENABLE
TRIG
QUEUED
OUT
6
5
9
3
0
5
10
15
20
25
30
35
40
Timestamp (125MHz FPGA clock ticks)


Changing parameters resets pulses
WIDTH
DELAY
ENABLE
TRIG
QUEUED
OUT
10
10
6
0
5
10
15
20
25
30
35
40
45
Timestamp (125MHz FPGA clock ticks)

6.21 QDEC - Quadrature Decoder

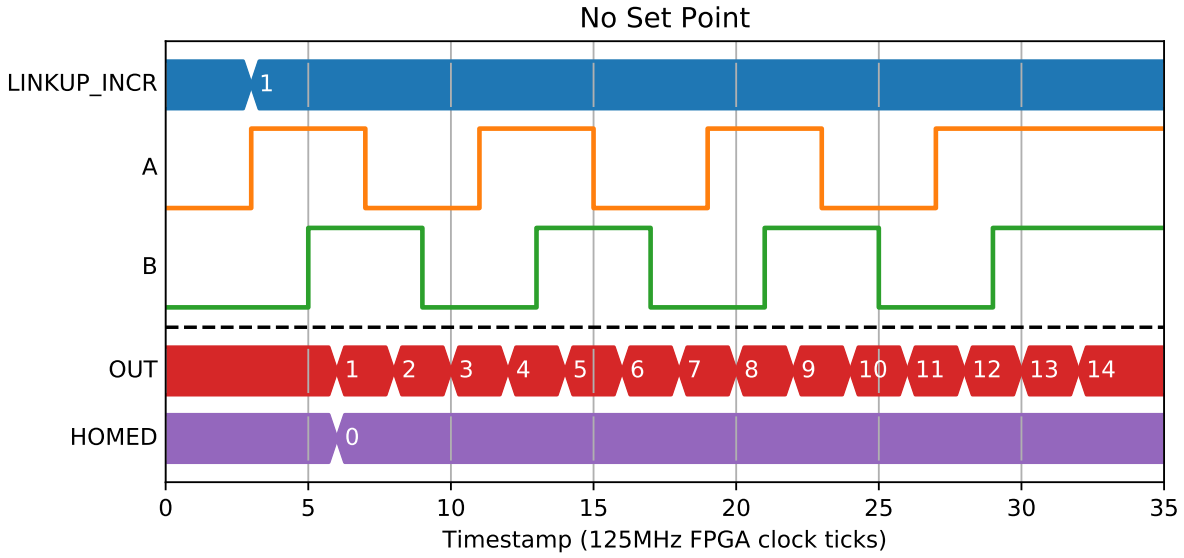
The QDEC block handles the encoder Decoding

6.21.1 Fields

Name	Type	Description
LINKUP_INCR	param bit	link up incremental coder signal
A	bit_mux	Quadrature A
B	bit_mux	Quadrature B
Z	bit_mux	Z index channel
RST_ON_Z	param bit	Zero position on Z rising edge
SETP	write int	Set point
HOMED	read bit	Quadrature homed status
OUT	pos_out	Output position

6.21.2 Counting

The quadrature decoder counts, incrementing at each rising or falling edge of the sequence. If the sequence is reversed the count will decrease at each edge. The initial value is set to the value of the SETP input.

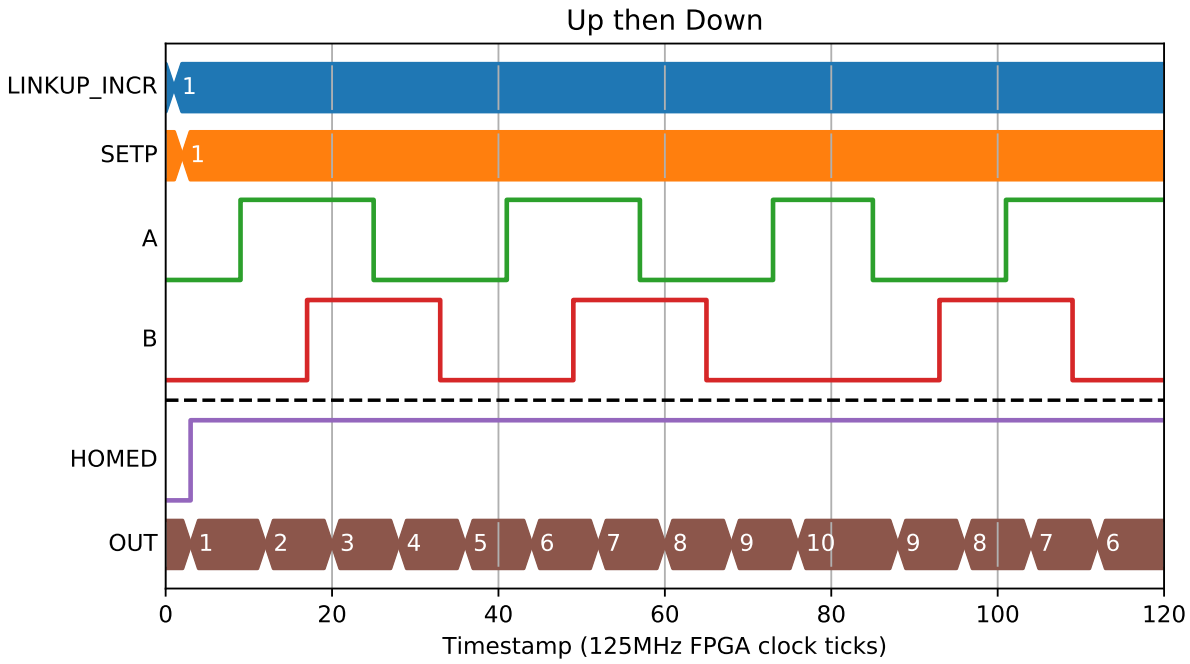


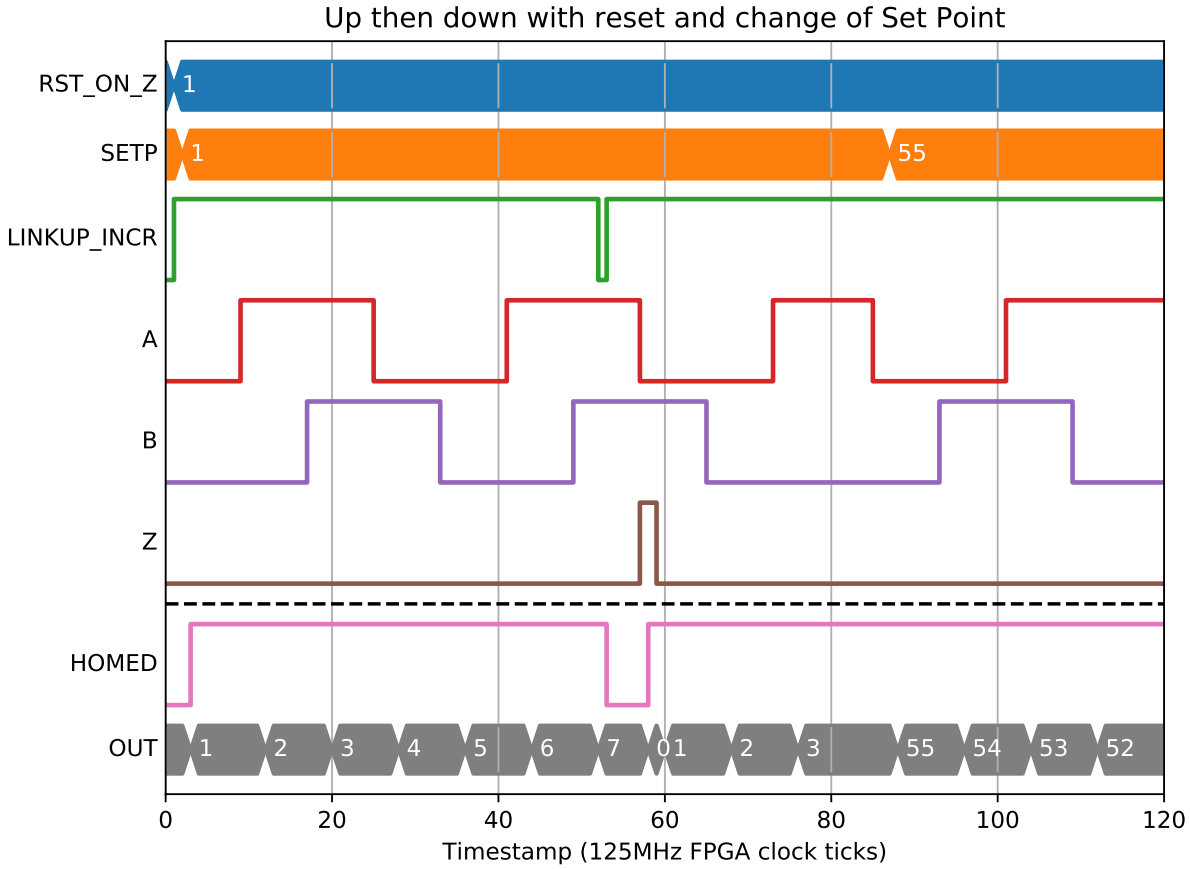
6.21.3 Resetting

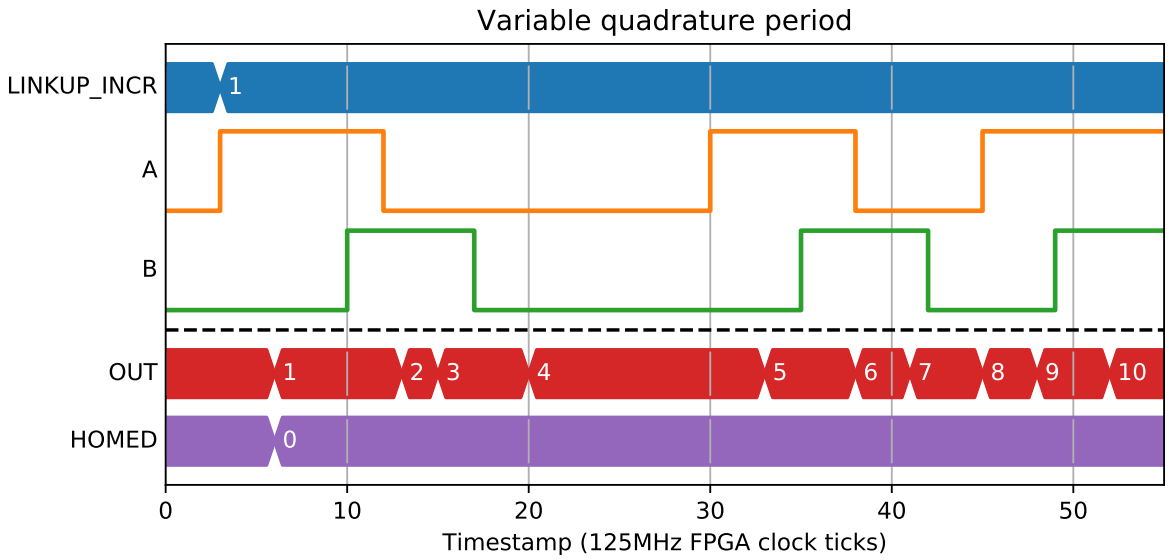
Whilst counting, it can be reset to '0' on while the Z input is high, provided that this functionality is enabled by setting the RST_ON_Z input to '1'. If the SETP input is changed the count value changes to the new value.

6.21.4 Limitations

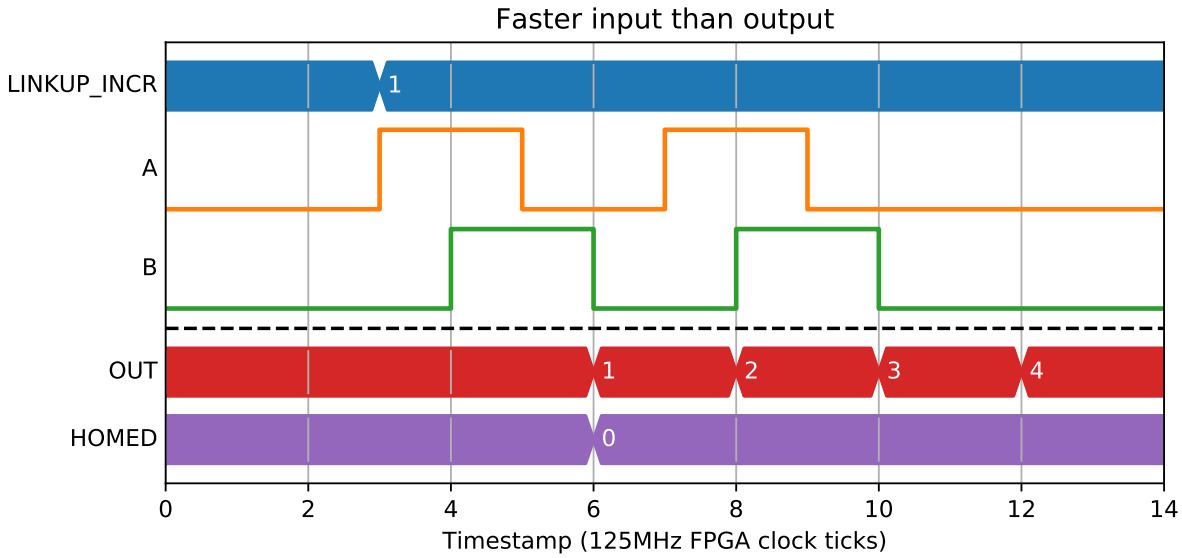
The block can continue to count when there is not a constant period between the pulses.


Up then Down
LINKUP_INCR
SETP
A
B
HOMED
OUT
1
2
3
4
5
6
7
8
9
10
9
8
7
6
0
20
40
60
80
100
120
Timestamp (125MHz FPGA clock ticks)


Up then down with reset and change of Set Point
RST_ON_Z
SETP
LINKUP_INCR
A
B
Z
HOMED
OUT
1
55
1
2
3
4
5
6
7
01
2
3
55
54
53
52
0
20
40
60
80
100
120
Timestamp (125MHz FPGA clock ticks)



The output takes three clock pulses to update. If the inputs are changing faster than this, inputs can be lost.



6.22 SEQ - Sequencer

The sequencer block performs automatic execution of sequenced lines to produce timing signals. Each line optionally waits for an external trigger condition and runs for an optional phase1, then a mandatory phase2 before moving to the next line. Each line sets the block outputs during phase1 and phase2 as defined by user-configured mask. Individual lines can be repeated, and the whole table can be repeated, with a value of 0 meaning repeat forever.

6.22.1 Fields

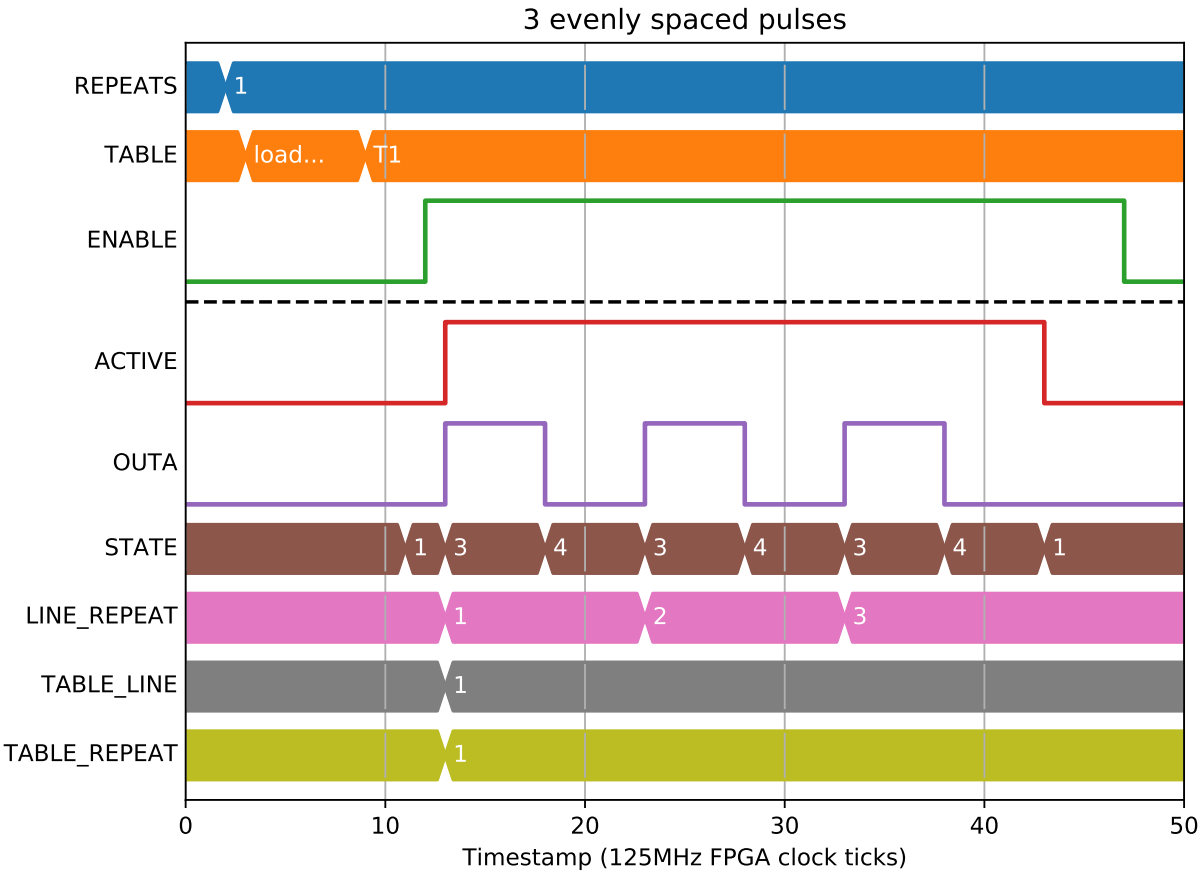
Name	Type	Description
ENABLE	bit_mux	Stop on falling edge, reset and enable on rising edge
BITA	bit_mux	BITA for optional trigger condition
BITB	bit_mux	BITB for optional trigger condition
BITC	bit_mux	BITC for optional trigger condition
POSA	pos_mux	POSA for optional trigger condition
POSB	pos_mux	POSB for optional trigger condition
POSC	pos_mux	POSC for optional trigger condition
TABLE	table short	Sequencer table of lines REPEATS Number of times the line will repeat TRIGGER The trigger condition to start the phases POSITION The position that can be used in trigger condition TIME1 The time the optional phase 1 should take OUTA1 Output A value during phase 1 OUTB1 Output B value during phase 1 OUTC1 Output C value during phase 1 OUTD1 Output D value during phase 1 OUTE1 Output E value during phase 1 OUTF1 Output F value during phase 1 TIME2 The time the mandatory phase 2 should take OUTA2 Output A value during phase 2 OUTB2 Output B value during phase 2 OUTC2 Output C value during phase 2 OUTD2 Output D value during phase 2 OUTE2 Output E value during phase 2 OUTF2 Output F value during phase 2 15:0 REPEATS 19:16 TRIGGER enum 0 Immediate 1 BITA=0 2 BITA=1 3 BITB=0 4 BITB=1 5 BITC=0 6 BITC=1 7 POSA>=POSITION 8 POSA<=POSITION 9 POSB>=POSITION 10 POSB<=POSITION 11 POSC>=POSITION 12 POSC<=POSITION 63:32 POSITION int 95:64 TIME1 20:20 OUTA1 21:21 OUTB1 22:22 OUTC1

6.22.2 Sequencer Table Line Composition

Bit Field	Name	Description
[15:0]	REPEATS	Number of times the line will repeat
[19:16]	TRIGGER	The trigger condition to start the phases 0: Immediate 1: BITA=0 2: BITA=1 3: BITB=0 4: BITB=1 5: BITC=0 6: BITC=1 7: POSA>=POSITION 8: POSA<=POSITION 9: POSB>=POSITION 10: POSB<=POSITION 11: POSC>=POSITION 12: POSC<=POSITION
[63:32]	POSITION	The position that can be used in trigger condition
[95:64]	TIME1	The time the optional phase 1 should take
[20:20]	OUTA1	Output A value during phase 1
[21:21]	OUTB1	Output B value during phase 1
[22:22]	OUTC1	Output C value during phase 1
[23:23]	OUTD1	Output D value during phase 1
[24:24]	OUTE1	Output E value during phase 1
[25:25]	OUTF1	Output F value during phase 1
[127:96]	TIME2	The time the mandatory phase 2 should take
[26:26]	OUTA2	Output A value during phase 2
[27:27]	OUTB2	Output B value during phase 2
[28:28]	OUTC2	Output C value during phase 2
[29:29]	OUTD2	Output D value during phase 2
[30:30]	OUTE2	Output E value during phase 2
[31:31]	OUTF2	Output F value during phase 2

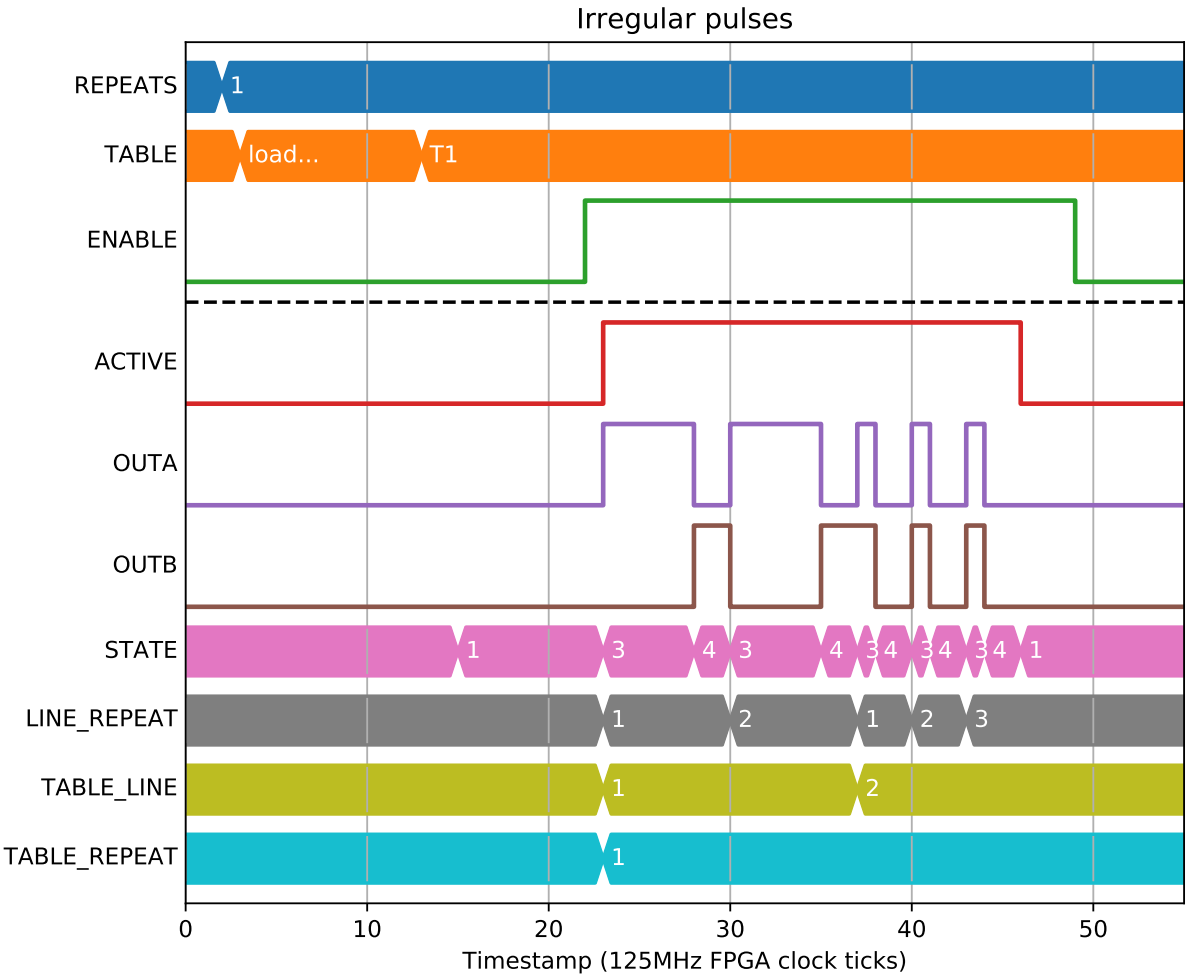
6.22.3 Generating fixed pulse trains

The basic use case is for generating fixed pulse trains when enabled. For example we can ask for 3x 50% duty cycle pulses by writing a single line table that is repeated 3 times. When enabled it will become active and immediately start producing pulses, remaining active until the pulses have been produced:


3 evenly spaced pulses
REPEATS
TABLE
ENABLE
ACTIVE
OUTA
STATE
LINE_REPEAT
TABLE_LINE
TABLE_REPEAT
1
load...
T1
1
3
4
3
4
3
4
1
1
2
3
1
1
0
10
20
30
40
50
Timestamp (125MHz FPGA clock ticks)

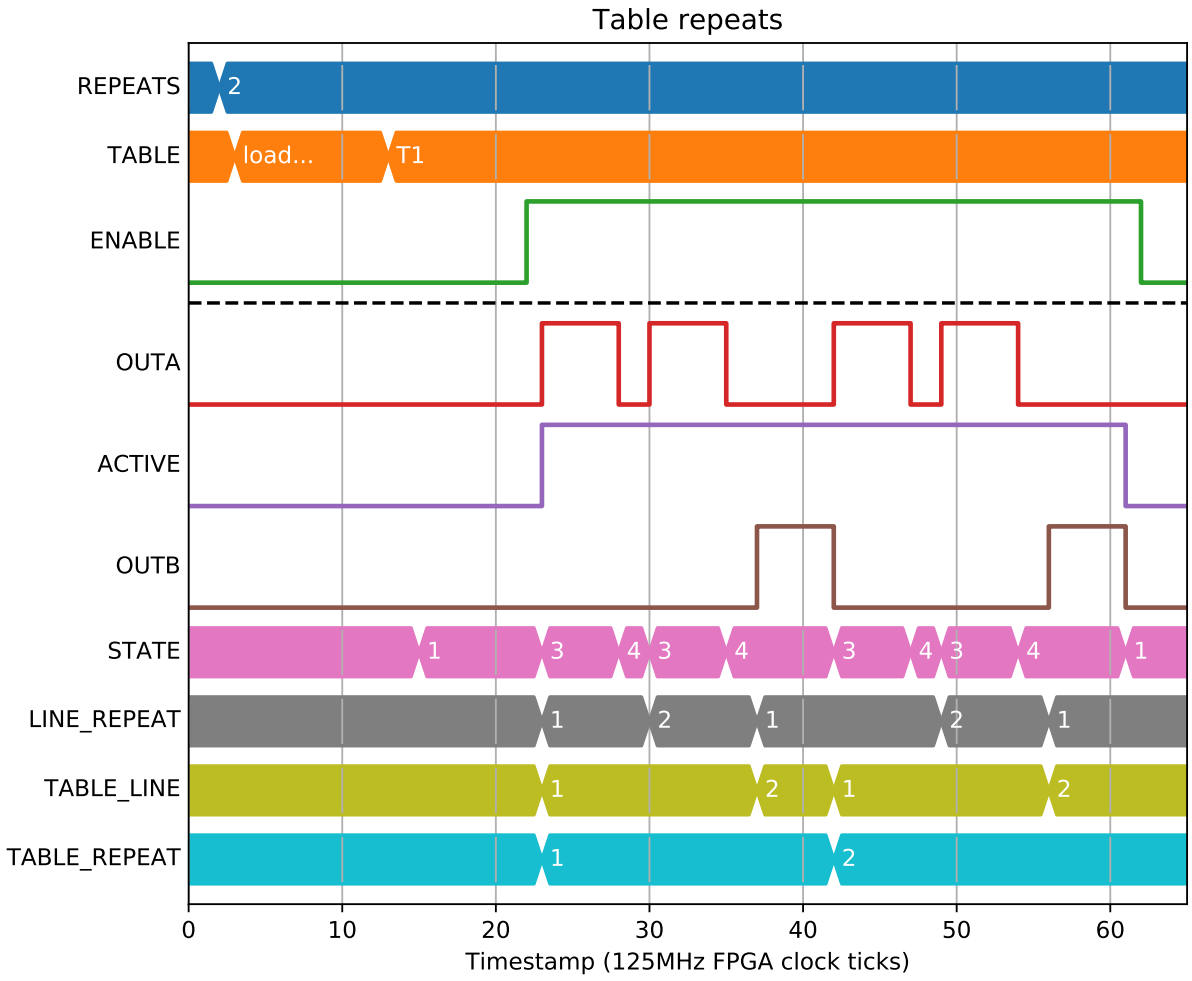
T1																
#	Trigger		Phase1	Phase1 Outputs						Phase2	Phase2 Outputs					
Repeats	Condition	Position	Time	A	B	C	D	E	F	Time	A	B	C	D	E	F
3	Immediate	0	5	1	0	0	0	0	0	5	0	0	0	0	0	0

We can also use it to generate irregular streams of pulses on different outputs by adding more lines to the table. Note that OUTB which was high at the end of Phase2 of the first line remains high in Phase1 of the second line:



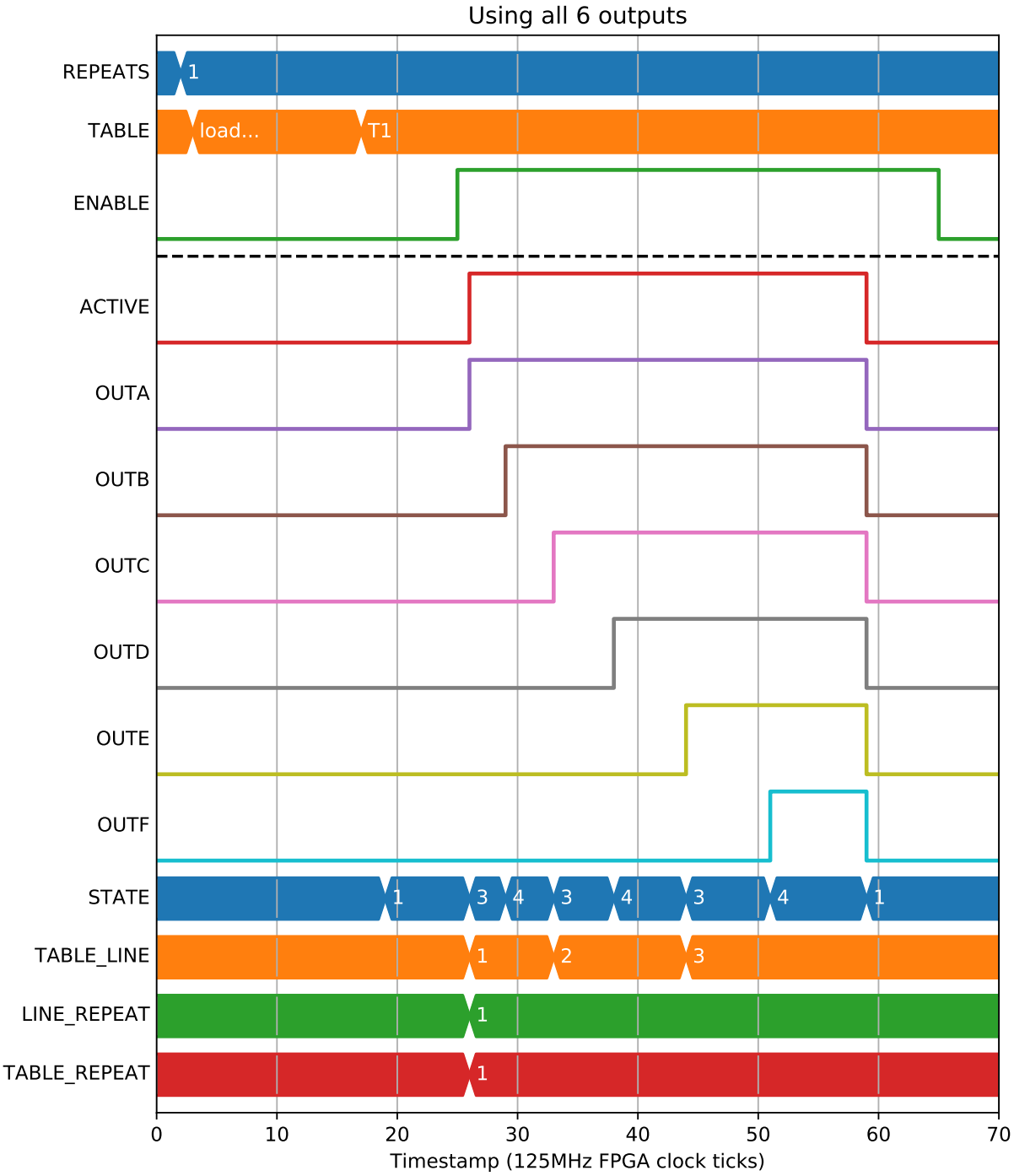
T1																
#	Trigger		Phase1	Phase1 Outputs						Phase2	Phase2 Outputs					
Repeats	Condition	Position	Time	A	B	C	D	E	F	Time	A	B	C	D	E	F
2	Immediate	0	5	1	0	0	0	0	0	2	0	1	0	0	0	0
3	Immediate	0	1	1	1	0	0	0	0	2	0	0	0	0	0	0

And we can set repeats on the entire table too. Note that in the second line of this table we have suppressed phase1 by setting its time to 0:



T1																
#	Trigger		Phase1	Phase1 Outputs						Phase2	Phase2 Outputs					
Repeats	Condition	Position	Time	A	B	C	D	E	F	Time	A	B	C	D	E	F
2	Immediate	0	5	1	0	0	0	0	0	2	0	0	0	0	0	0
1	Immediate	0	0	0	0	0	0	0	0	5	0	1	0	0	0	0

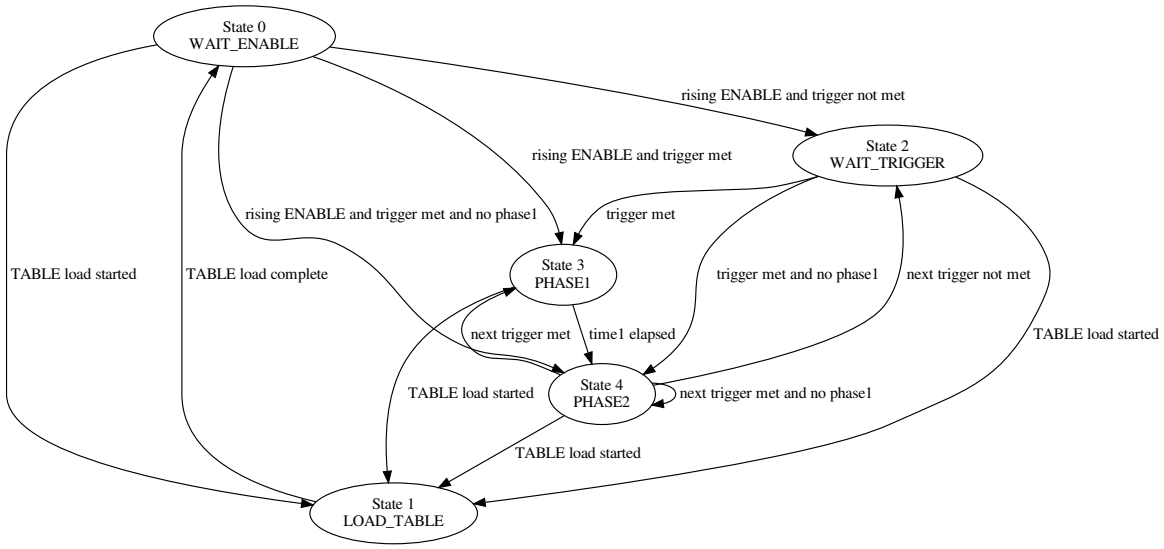
There are 6 outputs which allow for complex patterns to be generated:


Using all 6 outputs
REPEATS
TABLE
ENABLE
ACTIVE
OUTA
OUTB
OUTC
OUTD
OUTE
OUTF
STATE
TABLE_LINE
LINE_REPEAT
TABLE_REPEAT
load...
T1
0
10
20
30
40
50
60
70
Timestamp (125MHz FPGA clock ticks)

T1																
#	Trigger		Phase1	Phase1 Outputs						Phase2	Phase2 Outputs					
Repeats	Condition	Position	Time	A	B	C	D	E	F	Time	A	B	C	D	E	F
1	Immediate	0	3	1	0	0	0	0	0	4	1	1	0	0	0	0
1	Immediate	0	5	1	1	1	0	0	0	6	1	1	1	1	0	0
1	Immediate	0	7	1	1	1	1	1	0	8	1	1	1	1	1	1

6.22.4 Statemachine

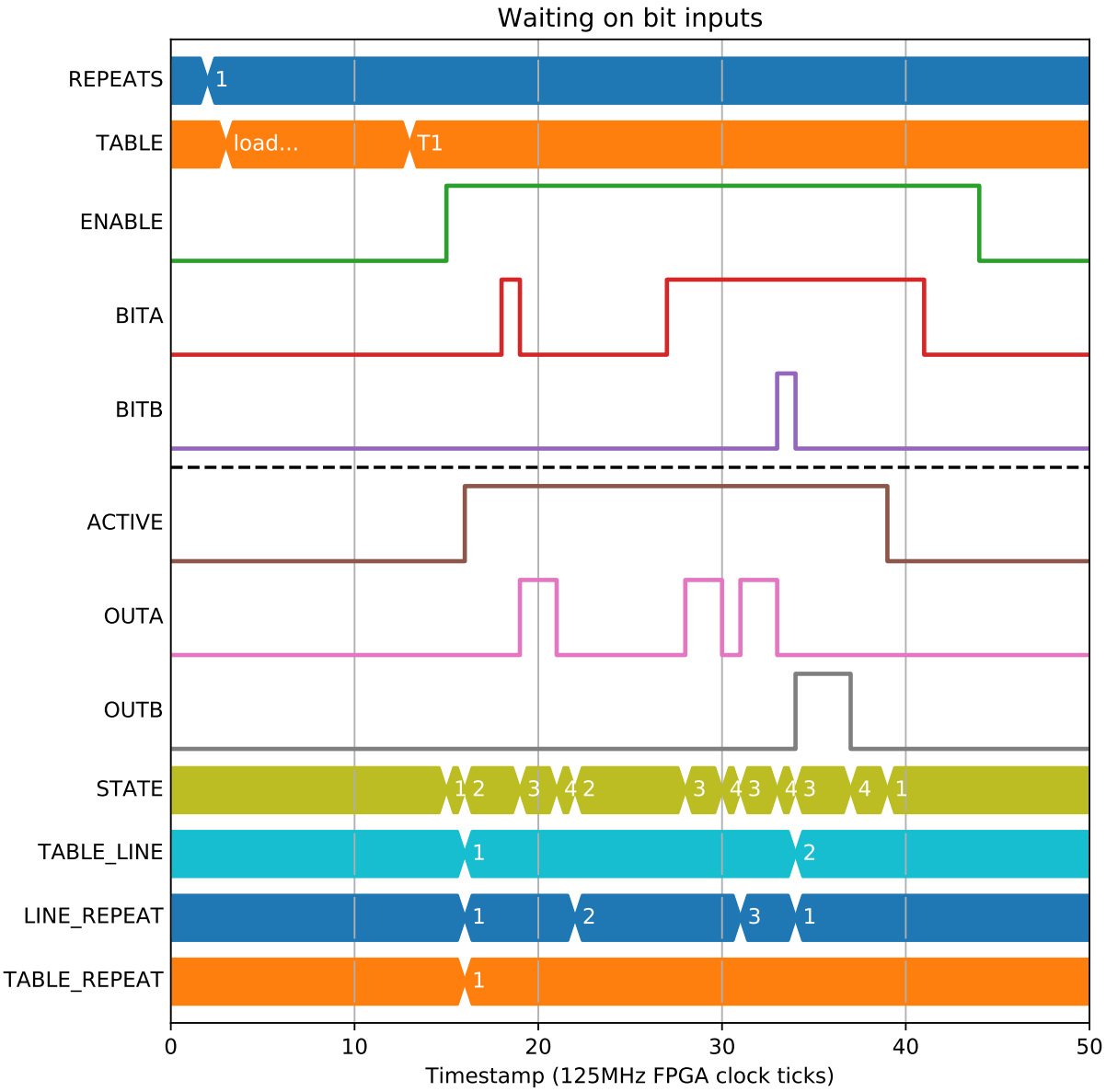
There is an internal statemachine that controls which phase is currently being output. It has a number of transitions that allow it to skip PHASE1 if there is none, or skip WAIT_TRIGGER if there is no trigger condition.



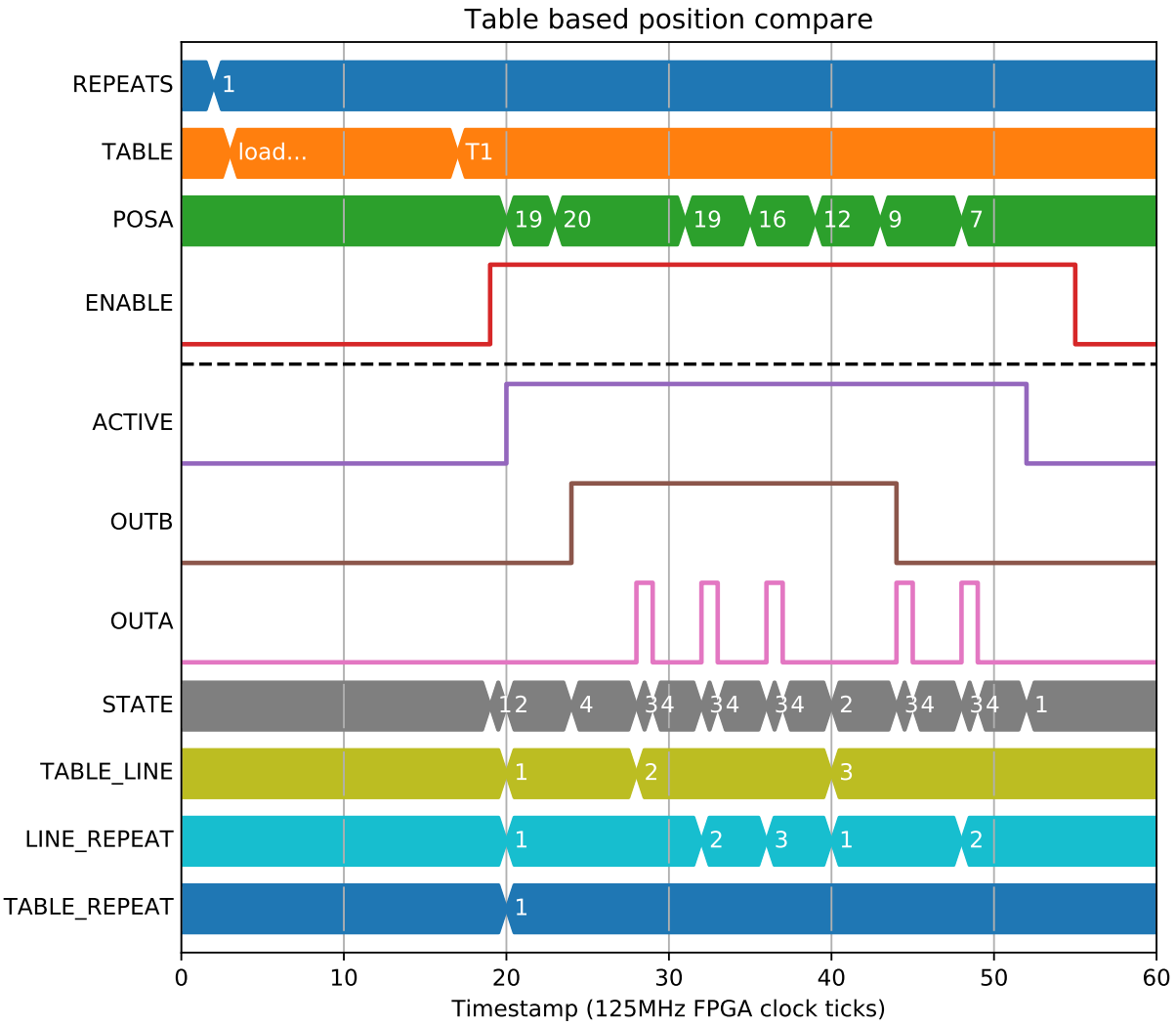
6.22.5 External trigger sources

The trigger column in the table allows an optional trigger condition to be waited on before the phased times are started. The trigger condition is checked on each repeat of the line, but not checked during phase1 and phase2. You can see when the Block is waiting for a trigger signal as it will enter the WAIT_TRIGGER(2) state:

T1																
#	Trigger		Phase1	Phase1 Outputs						Phase2	Phase2 Outputs					
Repeats	Condition	Position	Time	A	B	C	D	E	F	Time	A	B	C	D	E	F
3	BITA=1	0	2	1	0	0	0	0	0	1	0	0	0	0	0	0
1	BITB=1	0	3	0	1	0	0	0	0	2	0	0	0	0	0	0


Waiting on bit inputs
REPEATS
TABLE
ENABLE
BITA
BITB
ACTIVE
OUTA
OUTB
STATE
TABLE_LINE
LINE_REPEAT
TABLE_REPEAT
load...
T1
0
10
20
30
40
50
Timestamp (125MHz FPGA clock ticks)

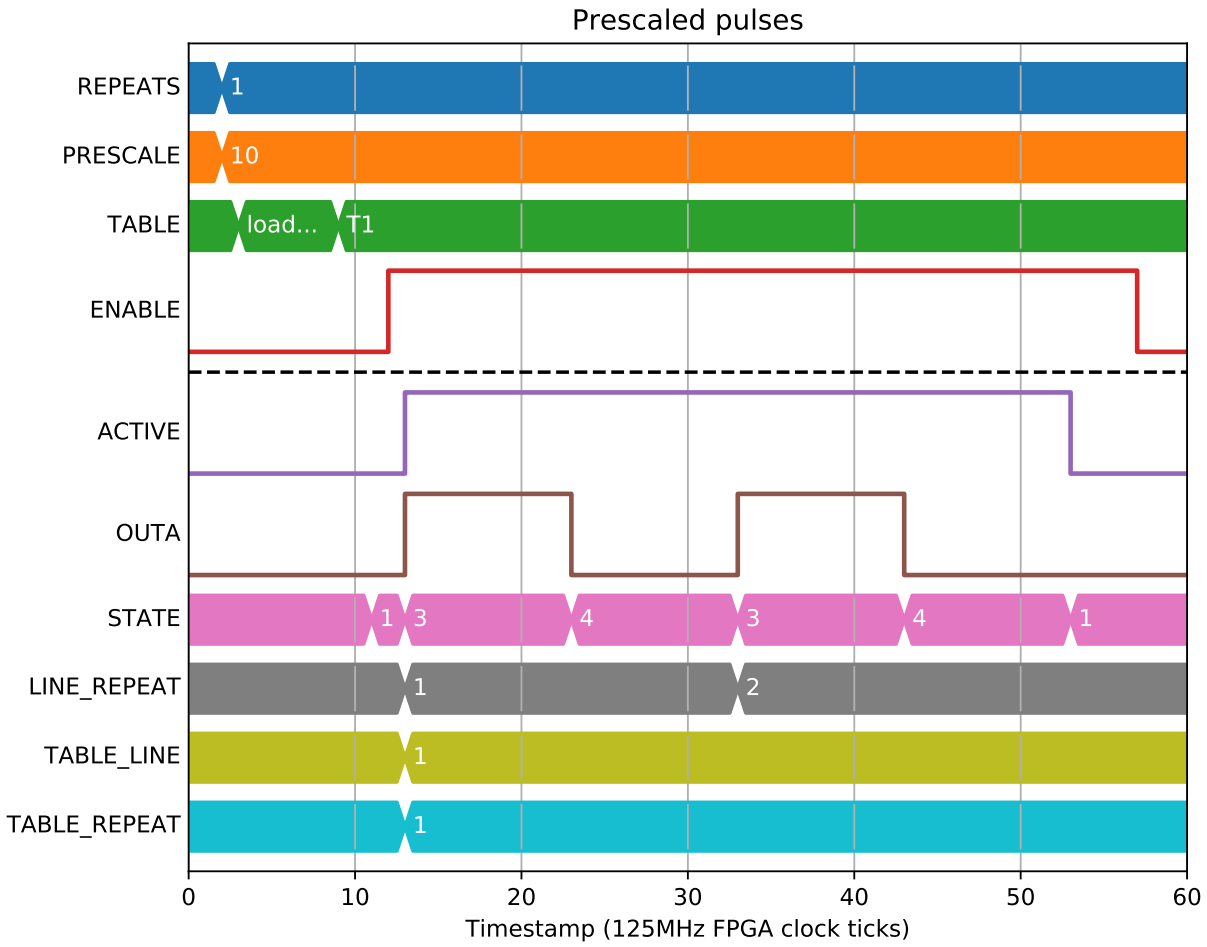
You can also use a position field as a trigger condition in the same way, this is useful to do a table based position compare:



T1																
#	Trigger		Phase1	Phase1 Outputs						Phase2	Phase2 Outputs					
Repeats	Condition	Position	Time	A	B	C	D	E	F	Time	A	B	C	D	E	F
1	POSA>=POSITION	20	0	0	0	0	0	0	0	4	0	1	0	0	0	0
3	Immediate	0	1	1	1	0	0	0	0	3	0	1	0	0	0	0
2	POSA<=POSITION	10	1	1	0	0	0	0	0	3	0	0	0	0	0	0

6.22.6 Prescaler

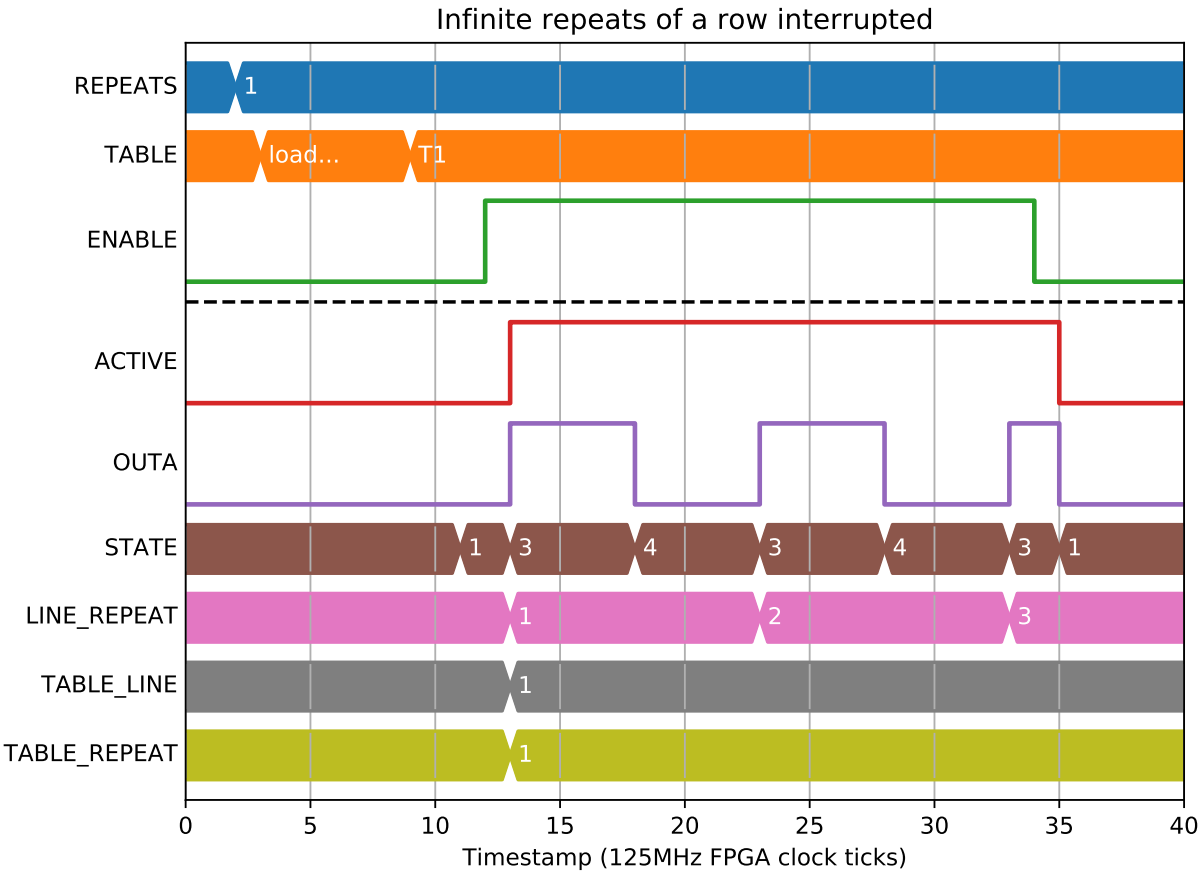
Each row of the table gives a time value for the phases. This value can be scaled with a block wide prescaler to allow a frame to be longer than 2**32 * 8e-9 = about 34 seconds. For example:


Prescaled pulses
REPEATS
PRESCALE
TABLE
ENABLE
ACTIVE
OUTA
STATE
LINE_REPEAT
TABLE_LINE
TABLE_REPEAT
1
10
load...
T1
1
3
4
3
4
1
1
2
1
1
0
10
20
30
40
50
60
Timestamp (125MHz FPGA clock ticks)

T1																
#	Trigger		Phase1	Phase1 Outputs						Phase2	Phase2 Outputs					
Repeats	Condition	Position	Time	A	B	C	D	E	F	Time	A	B	C	D	E	F
2	Immediate	0	1	1	0	0	0	0	0	1	0	0	0	0	0	0

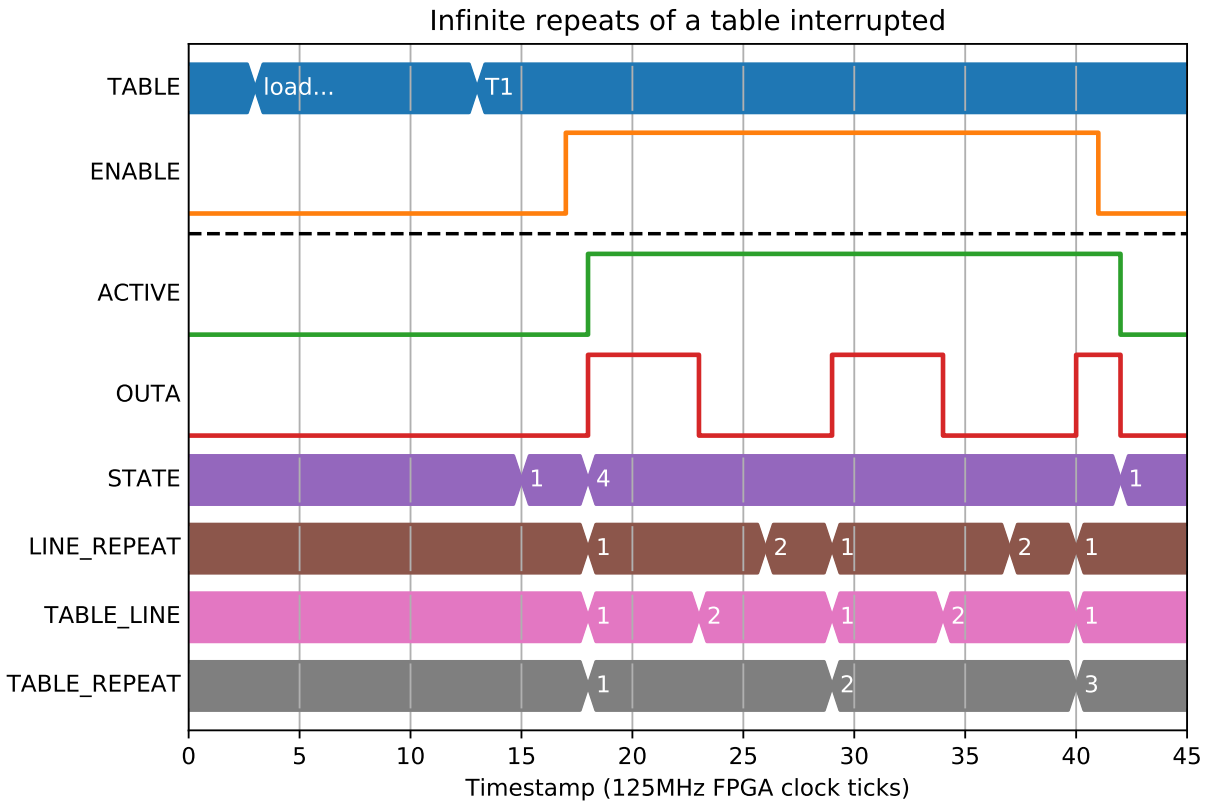
6.22.7 Interrupting a sequence

Setting the repeats on a table row to 0 will cause it to iterate until interrupted by a falling ENABLE signal:



T1																
#	Trigger		Phase1	Phase1 Outputs						Phase2	Phase2 Outputs					
Repeats	Condition	Position	Time	A	B	C	D	E	F	Time	A	B	C	D	E	F
0	Immediate	0	5	1	0	0	0	0	0	5	0	0	0	0	0	0

In a similar way, REPEATS=0 on a table will cause the whole table to be iterated until interrupted by a falling ENABLE signal:



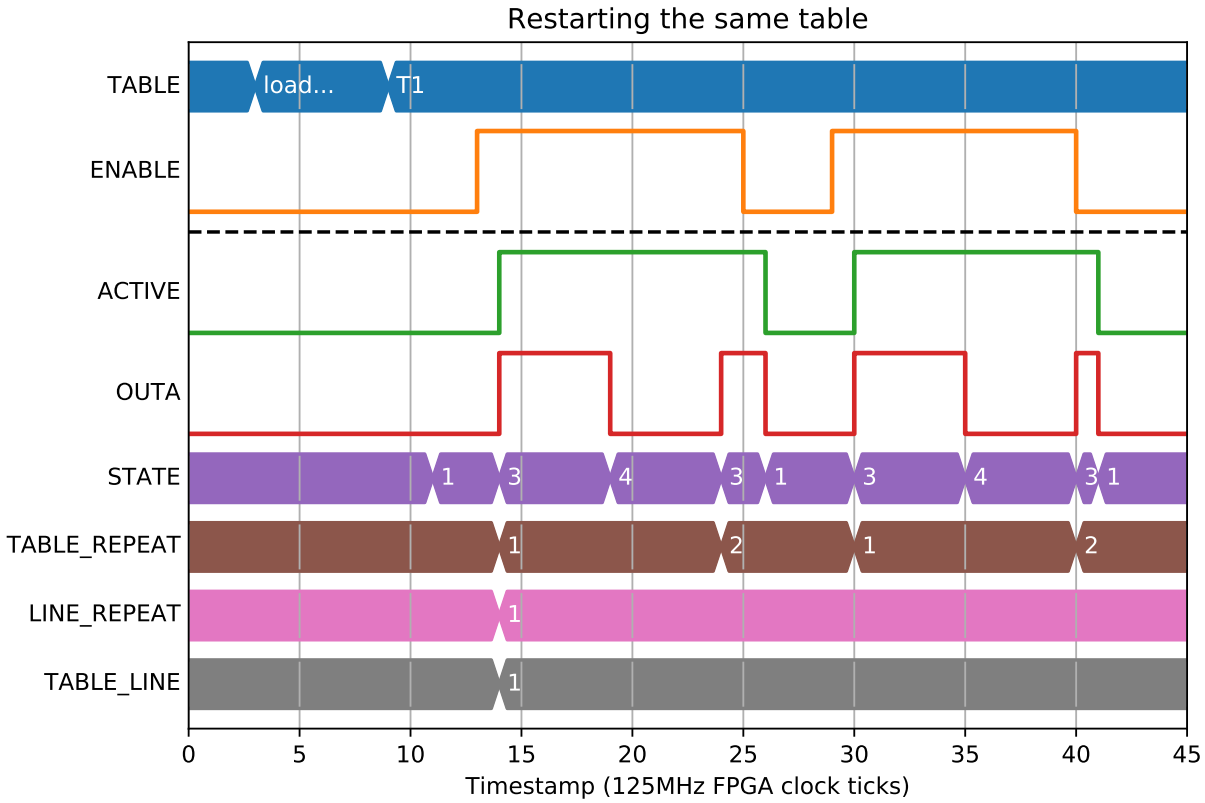
T1																
#	Trigger		Phase1	Phase1 Outputs						Phase2	Phase2 Outputs					
Repeats	Condition	Position	Time	A	B	C	D	E	F	Time	A	B	C	D	E	F
1	Immediate	0	0	0	0	0	0	0	0	5	1	0	0	0	0	0
2	Immediate	0	0	0	0	0	0	0	0	3	0	0	0	0	0	0

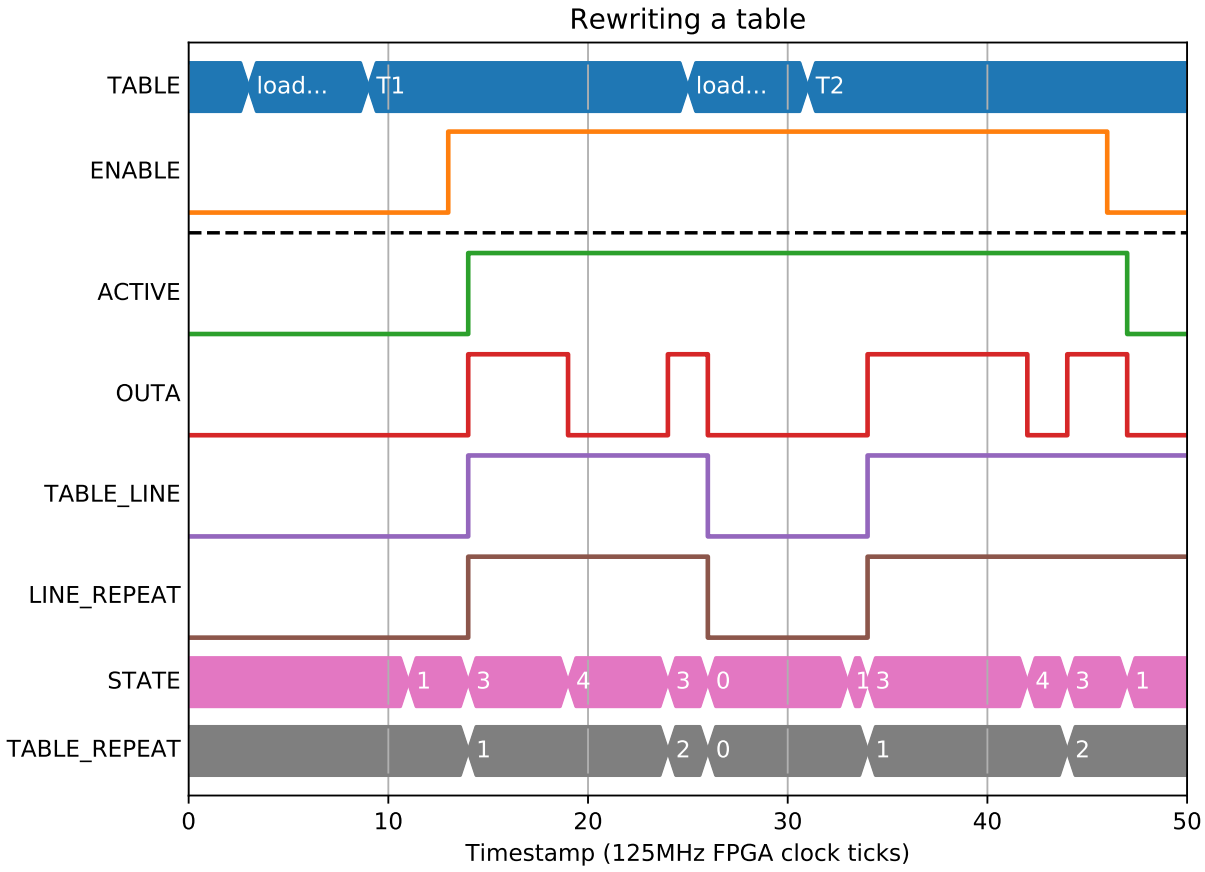
And a rising edge of the ENABLE will re-run the same table from the start:

T1																
#	Trigger		Phase1	Phase1 Outputs						Phase2	Phase2 Outputs					
Repeats	Condition	Position	Time	A	B	C	D	E	F	Time	A	B	C	D	E	F
1	Immediate	0	5	1	0	0	0	0	0	5	0	0	0	0	0	0

6.22.8 Table rewriting

If a table is written while enabled, the outputs and table state are reset and operation begins again from the first repeat of the first line of the table:


Restarting the same table
TABLE
load...
T1
ENABLE
ACTIVE
OUTA
STATE
1
3
4
3
1
3
4
3
1
TABLE_REPEAT
1
2
1
2
LINE_REPEAT
1
TABLE_LINE
1
0
5
10
15
20
25
30
35
40
45
Timestamp (125MHz FPGA clock ticks)


Rewriting a table
TABLE
load...
T1
load...
T2
ENABLE
ACTIVE
OUTA
TABLE_LINE
LINE_REPEAT
STATE
1
3
4
3
0
1
3
4
3
1
TABLE_REPEAT
1
2
0
1
2
0
10
20
30
40
50
Timestamp (125MHz FPGA clock ticks)

T1																
#	Trigger		Phase1	Phase1 Outputs						Phase2	Phase2 Outputs					
Repeats	Condition	Position	Time	A	B	C	D	E	F	Time	A	B	C	D	E	F
1	Immediate	0	5	1	0	0	0	0	0	5	0	0	0	0	0	0

T2																
#	Trigger		Phase1	Phase1 Outputs						Phase2	Phase2 Outputs					
Repeats	Condition	Position	Time	A	B	C	D	E	F	Time	A	B	C	D	E	F
1	Immediate	0	8	1	0	0	0	0	0	2	0	0	0	0	0	0

6.23 SFP_DLS_EVENTR - SFP Event Receiver Module

6.23.1 Fields

Name	Type	Description
EVENT_RESET	write action	Resets the event receiver
EVENT1	param enum	Event1 bit8 0 selects Event codes 1 selects DBus, bits7-0 event 272 MHz 288 Booster Clk 320 Storage Clk 122 Heart Beat 123 Reset Presc 124 Event Code 125 Reset Event 112 UT Seconds 0 113 UT Seonds 1 37 5Hz Event
EVENT2	param enum	Event2 bit8 0 selects Event codes 1 selects DBus, bits7-0 event 272 MHz 288 Booster Clk 320 Storage Clk 122 Heart Beat 123 Reset Presc 124 Event Code 125 Reset Event 112 UT Seconds 0 113 UT Seonds 1 37 5Hz Event
EVENT3	param enum	Event3 bit8 0 selects Event codes 1 selects DBus, bits7-0 event 272 MHz 288 Booster Clk 320 Storage Clk 122 Heart Beat 123 Reset Presc 124 Event Code 125 Reset Event 112 UT Seconds 0 113 UT Seonds 1 37 5Hz Event
EVENT4	param enum	Event4 bit8 0 selects Event codes 1 selects DBus, bits7-0 event 272 MHz

6.24 SFP_LOOPBACK- SFP Loopback Module

6.24.1 Fields

Name	Type	Description
SOFT_RESET	write action	GTX Soft Reset
SFP_LOS	read	SFP Loss Of Signal (from SFP module)
LINK_UP	read	GTX link status
ERROR_COUNT	read	GTX error count
SFP_CLK	read	SFP clock freq
SFP_MAC_LO	read	MAC low in integer value bit 23:0
SFP_MAC_HI	read	MAC high in integer value bit 47:24

6.25 SFP_PANDA_SYNC - Synchronize data between 2 PandAs

6.25.1 Fields

Name	Type	Description
IN.SYNC_RESET	write action	Resets the event receiver
IN.LINKUP	read	GTX_SPS link status
IN.BIT1	bit_out	SFP panda sync bit 1 input
IN.BIT2	bit_out	SFP panda sync bit 2 input
IN.BIT3	bit_out	SFP panda sync bit 3 input
IN.BIT4	bit_out	SFP panda sync bit 4 input
IN.BIT5	bit_out	SFP panda sync bit 5 input
IN.BIT6	bit_out	SFP panda sync bit 6 input
IN.BIT7	bit_out	SFP panda sync bit 7 input
IN.BIT8	bit_out	SFP panda sync bit 8 input
IN.BIT9	bit_out	SFP panda sync bit 9 input
IN.BIT10	bit_out	SFP panda sync bit 10 input
IN.BIT11	bit_out	SFP panda sync bit 11 input
IN.BIT12	bit_out	SFP panda sync bit 12 input
IN.BIT13	bit_out	SFP panda sync bit 13 input
IN.BIT14	bit_out	SFP panda sync bit 14 input
IN.BIT15	bit_out	SFP panda sync bit 15 input
IN.BIT16	bit_out	SFP panda sync bit 16 input
IN.POS1	pos_out	SFP panda sync pos 1 input
IN.POS2	pos_out	SFP panda sync pos 2 input
IN.POS3	pos_out	SFP panda sync pos 3 input
IN.POS4	pos_out	SFP panda sync pos 4 input
OUT.BIT1	bit_mux	SFP panda sync bit 1 output
OUT.BIT2	bit_mux	SFP panda sync bit 2 output
OUT.BIT3	bit_mux	SFP panda sync bit 3 output
OUT.BIT4	bit_mux	SFP panda sync bit 4 output
OUT.BIT5	bit_mux	SFP panda sync bit 5 output
OUT.BIT6	bit_mux	SFP panda sync bit 6 output
OUT.BIT7	bit_mux	SFP panda sync bit 7 output

Continued on next page

Table 1 – continued from previous page

Name	Type	Description
OUT.BIT8	bit_mux	SFP panda sync bit 8 output
OUT.BIT9	bit_mux	SFP panda sync bit 9 output
OUT.BIT10	bit_mux	SFP panda sync bit 10 output
OUT.BIT11	bit_mux	SFP panda sync bit 11 output
OUT.BIT12	bit_mux	SFP panda sync bit 12 output
OUT.BIT13	bit_mux	SFP panda sync bit 13 output
OUT.BIT14	bit_mux	SFP panda sync bit 14 output
OUT.BIT15	bit_mux	SFP panda sync bit 15 output
OUT.BIT16	bit_mux	SFP panda sync bit 16 output
OUT.POS1	pos_mux	SFP panda sync pos 1 output
OUT.POS2	pos_mux	SFP panda sync pos 2 output
OUT.POS3	pos_mux	SFP panda sync pos 3 output
OUT.POS4	pos_mux	SFP panda sync pos 4 output

6.26 SFP_UDPONTRIG - SFP UDP on trig Module

6.26.1 Fields

Name	Type	Description
SFP_TRIG	bit_mux	Rising edge to send UDP user-defined frame
SFP_START_COUNT	write action	Start counting Rising edge from zero and send UDP user-defined frame
SFP_STOP_COUNT	write action	Stop counting Rising edge and stop sending UDP user-defined frame
SFP_DEST_UDP_PORT	param uint 131071	Destination UDP Port (16 bits integer value)
SFP_OUR_UDP_PORT	param uint 131071	Source UDP Port (16 bits integer value)
SFP_DEST_IP_AD_BYTE1	param uint 255	Destination ip address byte 1 (byte integer value) ip=BYTE1.BYTE2.BYTE3.BYTE4
SFP_DEST_IP_AD_BYTE2	param uint 255	Destination ip address byte 2 (byte integer value) ip=BYTE1.BYTE2.BYTE3.BYTE4
SFP_DEST_IP_AD_BYTE3	param uint 255	Destination ip address byte 3 (byte integer value) ip=BYTE1.BYTE2.BYTE3.BYTE4
SFP_DEST_IP_AD_BYTE4	param uint 255	Destination ip address byte 4 (byte integer value) ip=BYTE1.BYTE2.BYTE3.BYTE4
SFP_OUR_IP_AD_BYTE1	param uint 255	Our source ip address byte 1 (byte integer value) ip=BYTE1.BYTE2.BYTE3.BYTE4
SFP_OUR_IP_AD_BYTE2	param uint 255	Our source ip address byte 2 (byte integer value) ip=BYTE1.BYTE2.BYTE3.BYTE4
SFP_OUR_IP_AD_BYTE3	param uint 255	Our source ip address byte 3 (byte integer value) ip=BYTE1.BYTE2.BYTE3.BYTE4
SFP_OUR_IP_AD_BYTE4	param uint 255	Our source ip address byte 4 (byte integer value) ip=BYTE1.BYTE2.BYTE3.BYTE4
SOFT_RESET	write action	GTX Soft Reset
SFP_TRIG_RISE_COUNT	read	Rising edge count
SFP_COUNT_UDPTX_ERR	read	UDP TX ERROR count
SFP_STATUS_COUNT	read	SFP count status (‘0’ => not started, ‘1’ => count enabled)
SFP_LOS	read	Loss Of Signal (from SFP module)
SFP_MAC_LO	read	MAC low in integer value bit 23:0
SFP_MAC_HI	read	MAC high in integer value bit 47:24

6.27 SRGATE - Set Reset Gate

An SRGATE block produces either a high (SET) or low (RST) output. It has configurable inputs and an option to force its output independently. Both Set and Rst inputs can be selected from bit bus, and the active-edge of its inputs is configurable. An enable signal allows the block to ignore its inputs.

6.27.1 Fields

Name	Type	Description
ENABLE	bit_mux	Whether to listen to SET/RST events
SET	bit_mux	A falling/rising edge sets the output to 1
RST	bit_mux	a falling/rising edge resets the output to 0
WHEN_DISABLED	param enum	What to do with the output when Enable is low 0 Set output low 1 Set output high 2 Keep current output
SET_EDGE	param enum	Output set edge 0 Rising 1 Falling 2 Either
RST_EDGE	param enum	Output reset edge 0 Rising 1 Falling 2 Either
FORCE_SET	write action	Set output to 1
FORCE_RST	write action	Reset output to 0
OUT	bit_out	output value

6.27.2 Normal conditions

The normal behaviour is to set the output OUT on the configured edge of the SET or RESET input.

6.27.3 Disabling the block

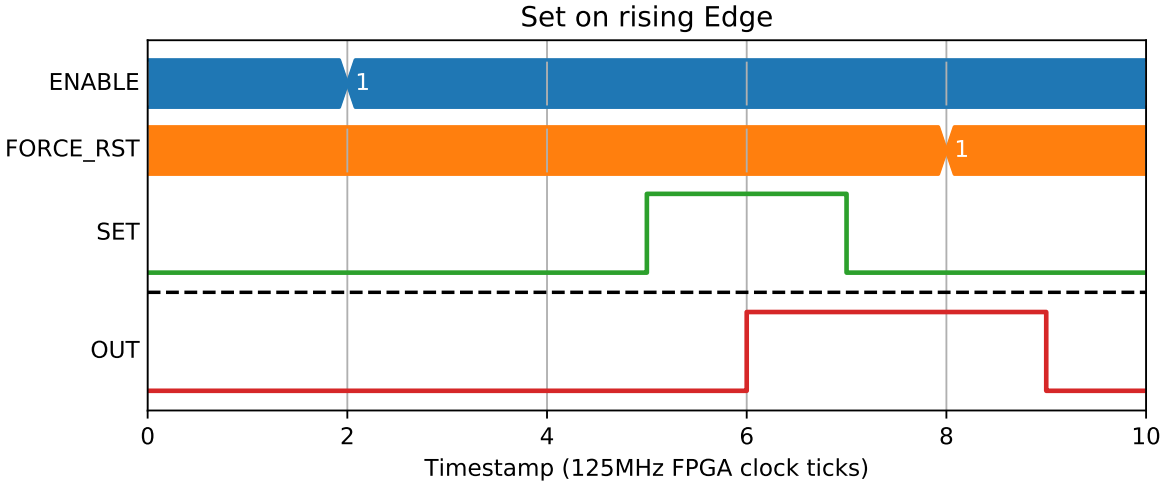
The default behaviour is to force the block output low when disabled, ignoring any SET/RST events:

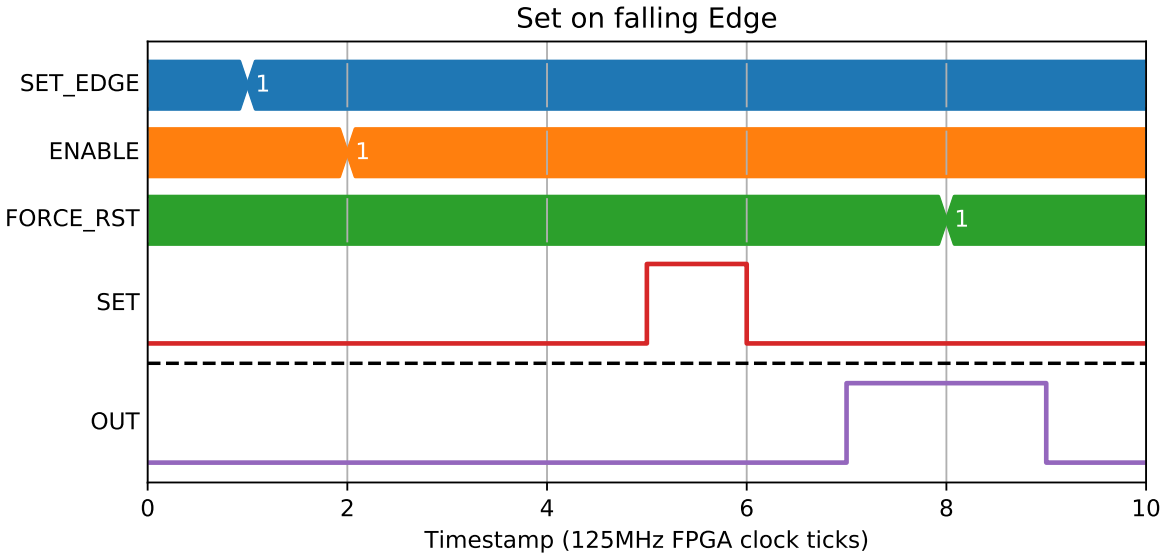
The disabled value can also be set high:

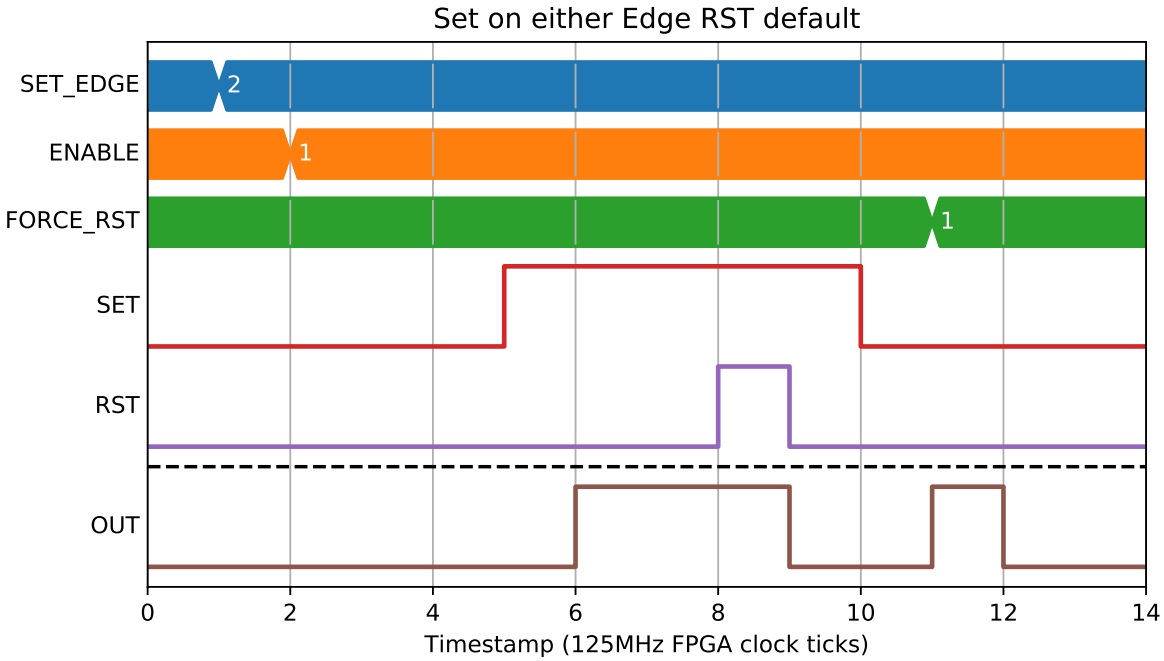
Or left at its current value:

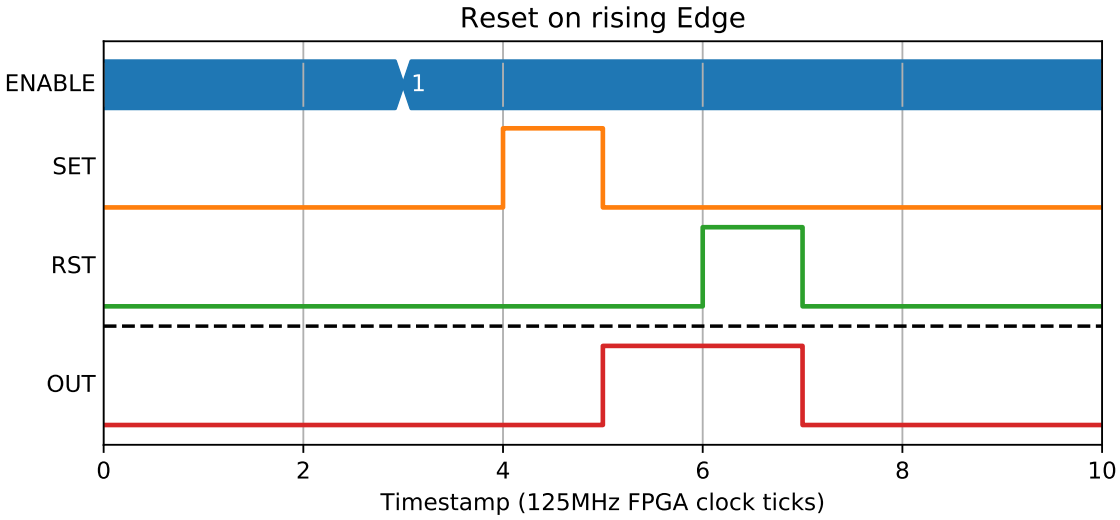
6.27.4 Active edge configure conditions

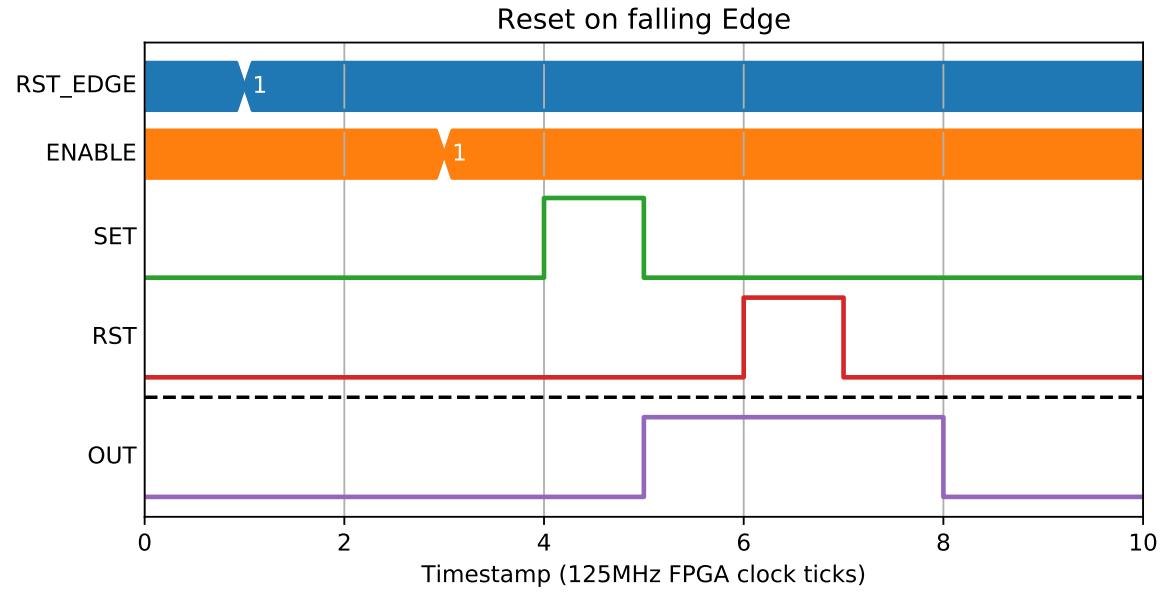
if the active edge is 'rising' then reset to 'falling' at the same time as a rising edge on the SET input, the block will ignore the rising edge and set the output OUT on the falling edge of the SET input.

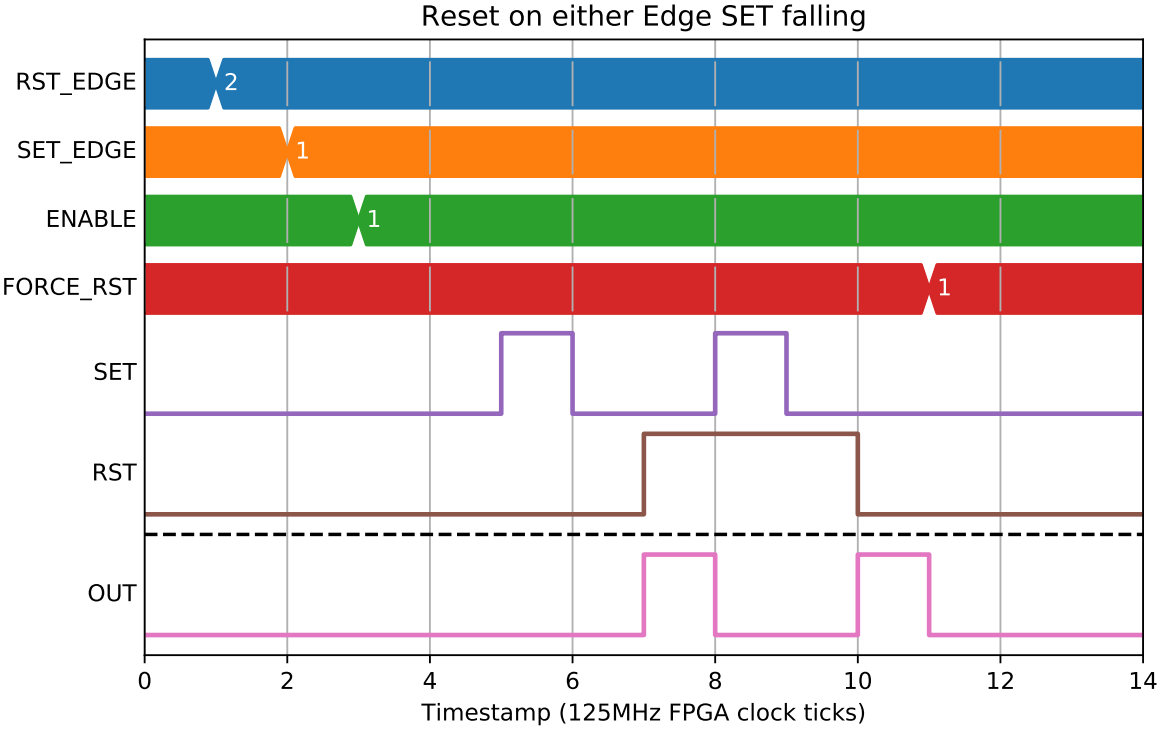

Set on rising Edge
ENABLE
FORCE_RST
SET
OUT
1
1
0
2
4
6
8
10
Timestamp (125MHz FPGA clock ticks)

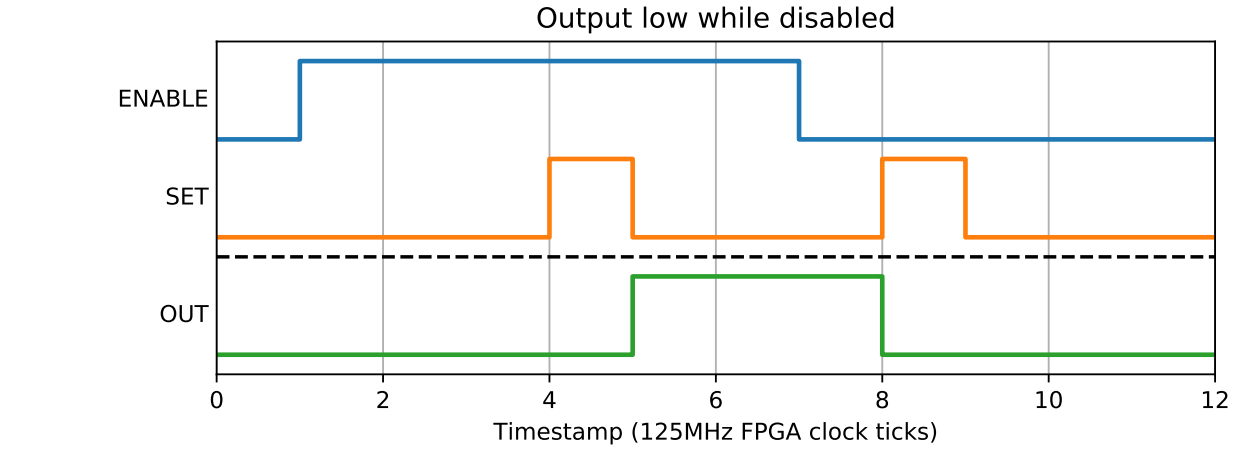

Set on falling Edge
SET_EDGE
ENABLE
FORCE_RST
SET
OUT
1
1
1
0
2
4
6
8
10
Timestamp (125MHz FPGA clock ticks)

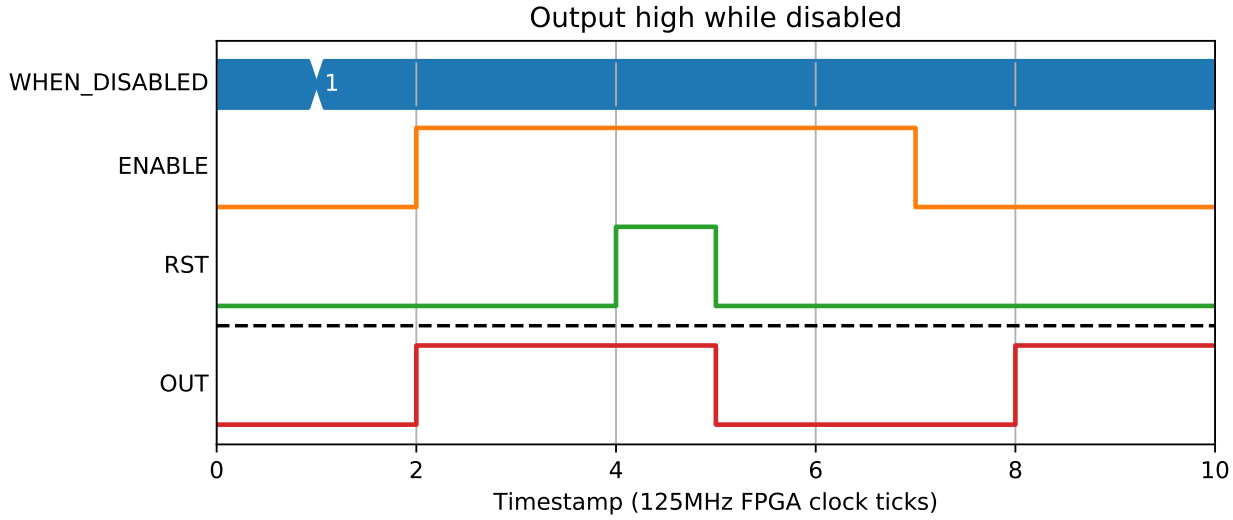

Set on either Edge RST default
SET_EDGE
ENABLE
FORCE_RST
SET
RST
OUT
2
1
1
0
2
4
6
8
10
12
14
Timestamp (125MHz FPGA clock ticks)

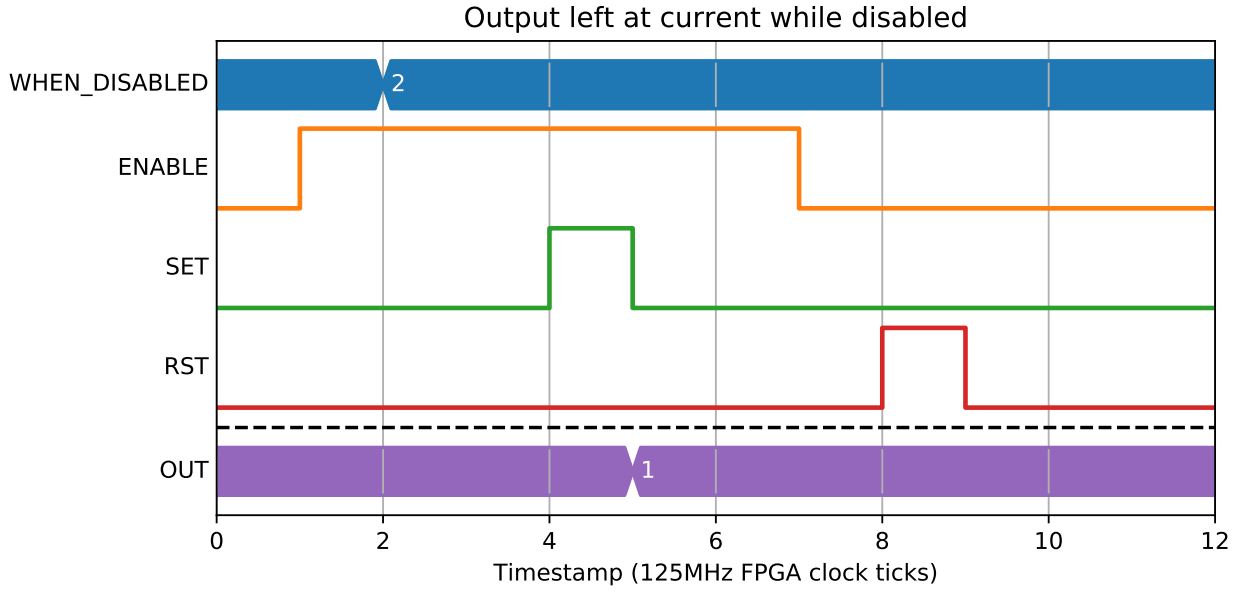

Reset on rising Edge
ENABLE
SET
RST
OUT
1
0
2
4
6
8
10
Timestamp (125MHz FPGA clock ticks)

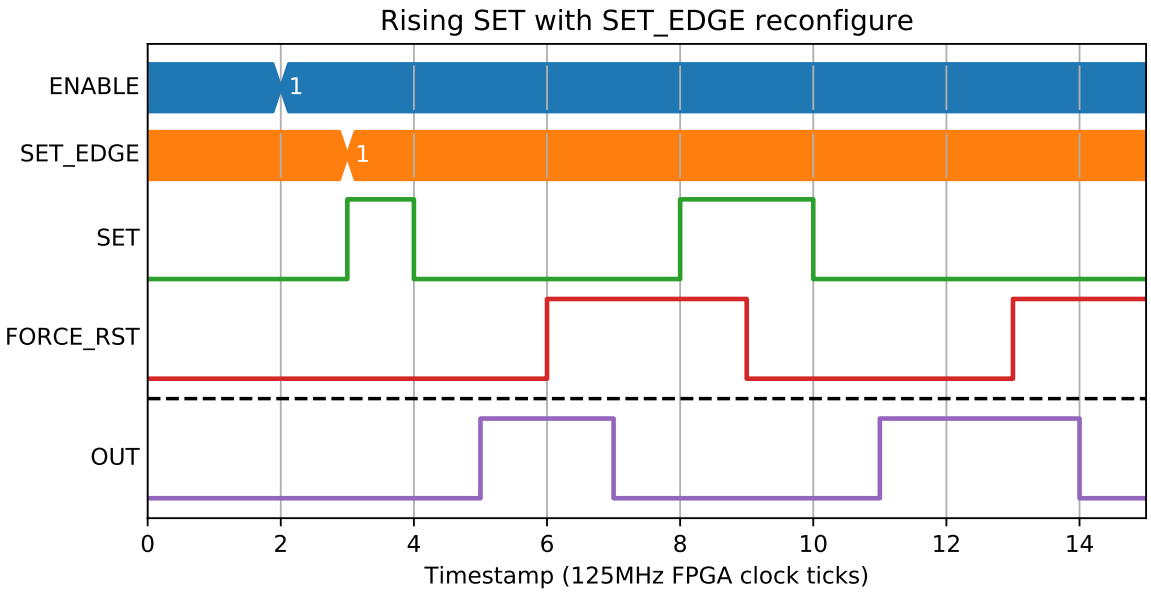

Reset on falling Edge
RST_EDGE
ENABLE
SET
RST
OUT
1
1
0
2
4
6
8
10
Timestamp (125MHz FPGA clock ticks)


Reset on either Edge SET falling
RST_EDGE
SET_EDGE
ENABLE
FORCE_RST
SET
RST
OUT
2
1
1
1
0
2
4
6
8
10
12
14
Timestamp (125MHz FPGA clock ticks)

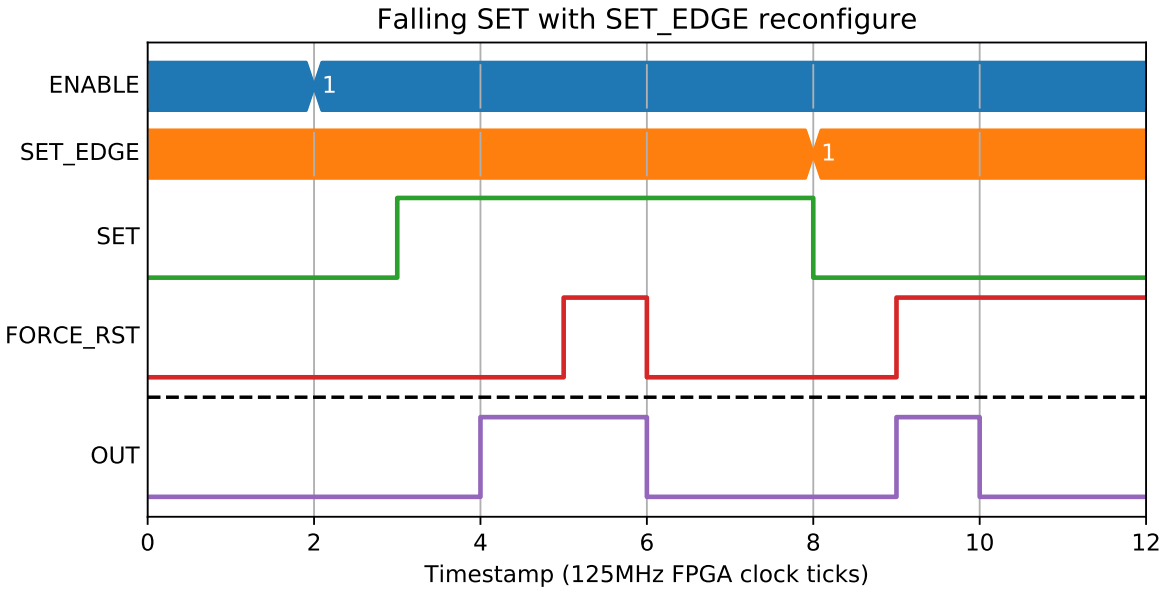

Output low while disabled
ENABLE
SET
OUT
0
2
4
6
8
10
12
Timestamp (125MHz FPGA clock ticks)


Output high while disabled
WHEN_DISABLED
1
ENABLE
RST
OUT
0
2
4
6
8
10
Timestamp (125MHz FPGA clock ticks)


Output left at current while disabled
WHEN_DISABLED
2
ENABLE
SET
RST
OUT
1
0
2
4
6
8
10
12
Timestamp (125MHz FPGA clock ticks)

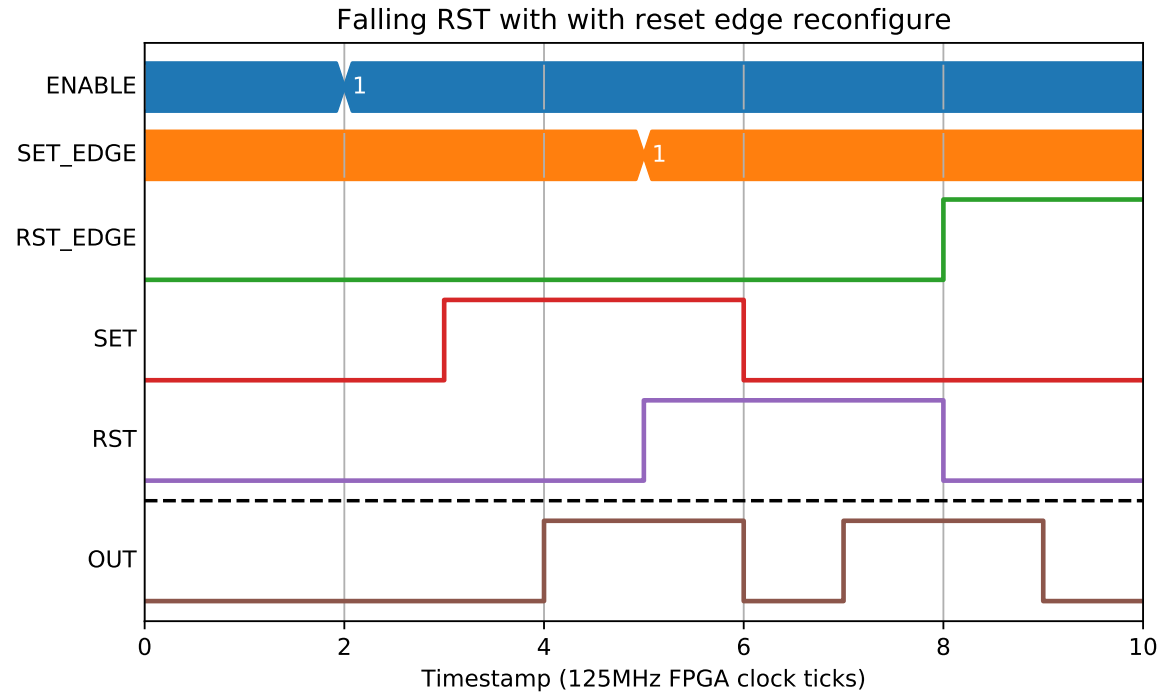


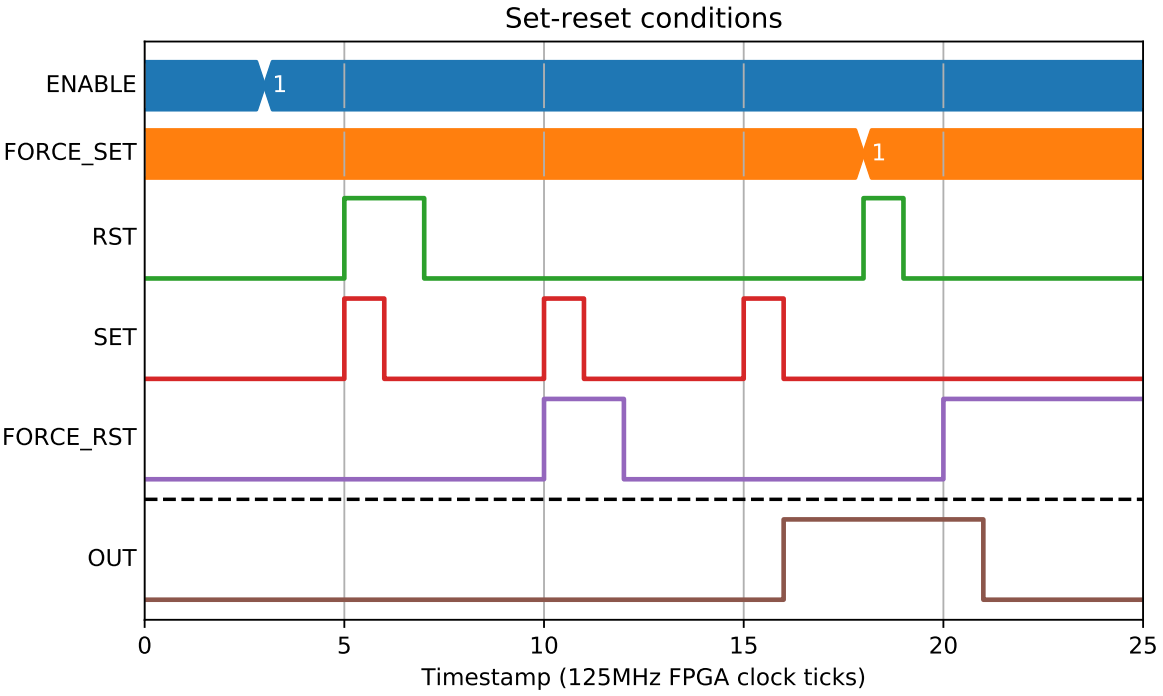
If the active edge changes to 'falling' at the same time as a falling edge on the SET input, the output OUT will be set following this.



6.27.5 Set-reset conditions

When determining the output if two values are set simultaneously, FORCE_SET and FORCE_RESET registers take priority over the input bus, and reset takes priority over set.


Falling RST with with reset edge reconfigure
ENABLE
SET_EDGE
RST_EDGE
SET
RST
OUT
1
1
0
2
4
6
8
10
Timestamp (125MHz FPGA clock ticks)


Set-reset conditions
ENABLE
FORCE_SET
RST
SET
FORCE_RST
OUT
1
1
0
5
10
15
20
25
Timestamp (125MHz FPGA clock ticks)

6.28 SYSTEM - System control FPGA

6.28.1 Fields

Name	Type	Description
TEMP_PSU	read int	On-board temperature [Power Supply]
TEMP_SFP	read int	On-board temperature [SFP]
TEMP_ENC_L	read int	On-board temperature [Left Encoder]
TEMP_PICO	read int	On-board temperature [Picozed]
TEMP_ENC_R	read int	On-board temperature [Right Encoder]
TEMP_ZYNQ	read scalar	On-board zynq temperature
ALIM_12V0	read scalar	On-board voltage sensor values
PICO_5V0	read scalar	On-board voltage sensor values
IO_5V0	read scalar	On-board voltage sensor values
SFP_3V3	read scalar	On-board voltage sensor values
FMC_15VN	read scalar	On-board voltage sensor values
FMC_15VP	read scalar	On-board voltage sensor values
ENC_24V	read scalar	On-board voltage sensor values
FMC_12V	read scalar	On-board voltage sensor values
PLL_LOCKED	read	PLL locked for SMA external clock
EXT_CLOCK	param enum	External sma and event receiver clock enables 0 int clock 1 sma clock 2 event receiver
EXT_CLOCK_FREQ	read	External clock freq
VCCINT	read scalar	On-board voltage sensor
CLK_SEL_STAT	read	Read-back of EXT/event reciever clock select

6.29 TTLIN - TTL Input

The TTLIN block handles the signals from the TTL Input connectors

6.29.1 Fields

Name	Type	Description
TERM	param enum	Select TTL input termination 0 High-Z 1 50-Ohm
VAL	bit_out	TTL input value

6.30 TTLOUT - TTL Output

The TTLOUT block handles the signals to the TTL Output connectors

6.30.1 Fields

Name	Type	Description
VAL	bit_mux	TTL output value

CHAPTER 7

Contributing

Contributions and issues are most welcome! All issues and pull requests are handled through github on the PandABlocks repository. Also, please check for any existing issues before filing a new one. If you have a great idea but it involves big changes, please file a ticket before making a pull request! We want to make sure you don't spend your time coding something that might not fit the scope of the project.

7.1 Running the tests

To get the source source code and run the unit tests, run:

```
$ git clone git://github.com/PandABlocks/PandABlocks-FPGA.git
$ cd PandABlocks-FPGA
$ virtualenv venv
$ source venv/bin/activate
$ pip install --upgrade pip
$ pip install -r tests/requirements.txt
$ cp CONFIG.example CONFIG
$ make test_python
$ make sim_timing
```

7.2 Writing VHDL

Code styling here...

7.3 Writing Python

Please arrange imports with the following style

```
# Standard library imports
import os

# Third party package imports
from mock import patch

# Local package imports
from common.python.configs import BlockConfig
```

Please follow Google's python style guide wherever possible.

7.4 Documentation

There are some conventions:

- First usage of a term in a page should link to an entry in the *Glossary*
- Glossary entries should define a reference with a trailing underscore

You can build the docs When in the project directory:

```
$ source venv/bin/activate
$ pip install -r docs/requirements.txt
$ make docs
$ firefox docs/index.html
```

7.5 Release Checklist

Before a new release, please go through the following checklist:

- Add a release note in CHANGELOG.rst
- Git tag the version

CHAPTER 8

Assembling Blocks into an App

A collections of *Block* instances that can be loaded to a *PandABlocks Device* is called an *App*. This section details how to create and build a new App.

8.1 App ini file

An ini file is used to specify the Blocks that make up an App. It lives in the `apps/` directory and has the extension `.app.ini`. It consists of a top level section with information about the App, then a section for every *Block* in the App.

8.1.1 The [.] section

The first section contains app wide information. It looks like this:

```
[.]
description: Short description of what this app will do
target: device_type
```

The `description` value is a human readable description of what the app contains and why it should be used.

The `target` value must correspond to a directory name in `targets/` that will be used to wrap the blocks in a top level entity that is loadable on the given PandABlocks device.

8.1.2 [BLOCK] sections

All other sections specify Block instance information. They look like this:

```
[MYBLOCK]
number: 4
module: mymodule
ini: myblock.block.ini
```

The section name is used to determine the name of the Block in the resulting App. It should be made of upper case letters and underscores with no numbers.

The `number` value gives the number of blocks that will be instantiated in the App. If not specified it will default to 1.

The `module` value gives the directory in `modules/` that the *Block ini* file lives in. If not specified it is the lowercase version of the section name.

The `ini` value gives the *Block ini* filename relative to the module directory. If not specified it is the lowercase version of the section name + `.block.ini`

8.2 App build process

Run:

```
make
```

And it will make a *Zpkg* for each App that can be loaded onto the PandABlocks Device. You can specify a subset of Apps to be built in the top level CONFIG file by specifying something like:

```
APPS = PandABox-no-fmc
```

8.3 Querying the App at runtime

The app name can be queried at run time via the TCP server:

```
< *METADATA.APPNAME?
> OK =PandABox-fmc_24vio
```

CHAPTER 9

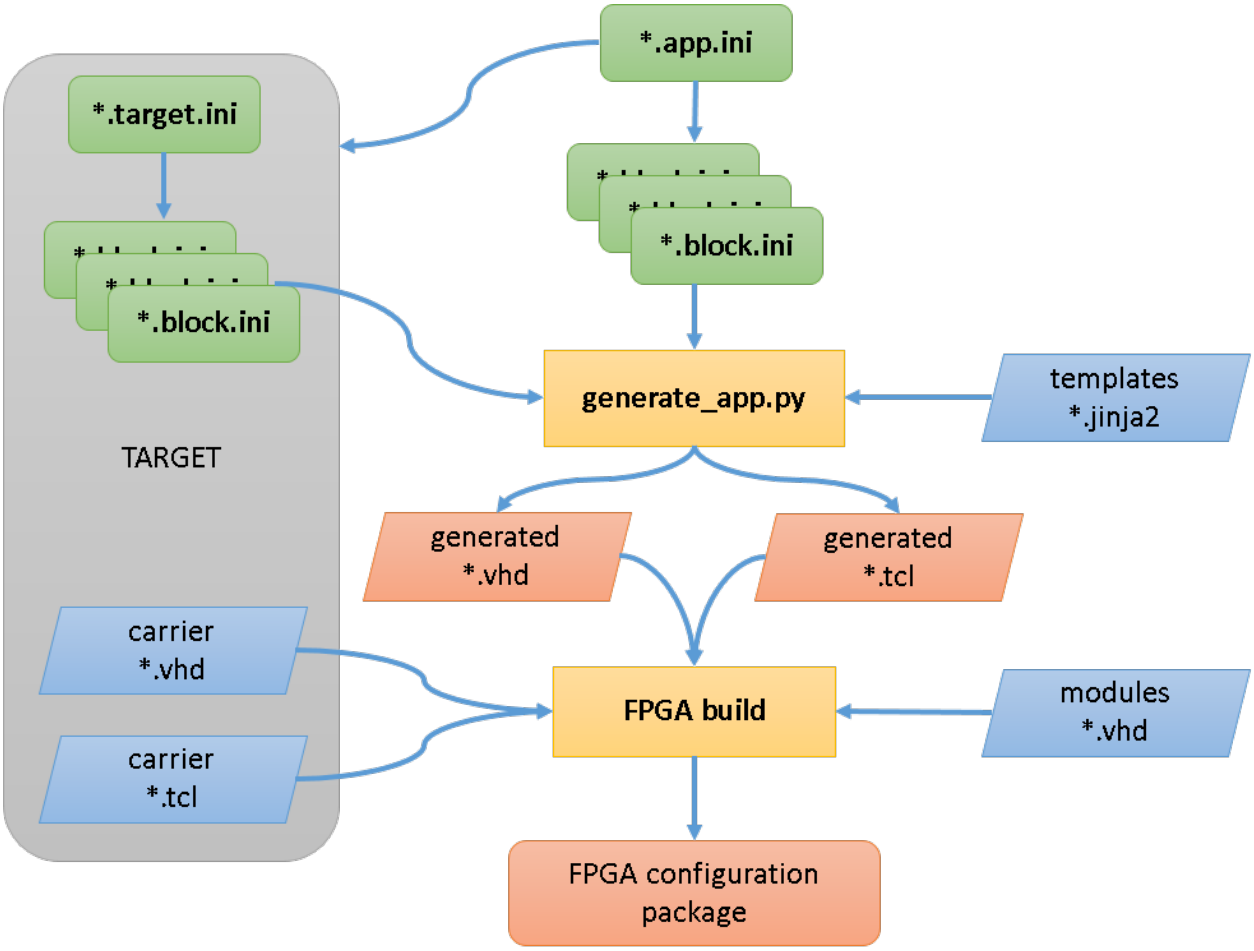
Writing a Block

If you have checked the list of *Available Blocks* and need a feature that is not there you can extend an existing *Block* or create a new one. If the feature fits with the behaviour of an existing Block and can be added without breaking backwards compatibility it is preferable to add it there. If there is a new type of behaviour it may be better to make a new one.

This page lists all of the framework features that are involved in making a Block, finding a *Module* for it, defining the interface, writing the simulation, writing the timing tests, documenting the behaviour, and finally writing the logic.

9.1 Architecture

An overview of the build process is shown in this diagram, the stages and terminology are defined below:



9.2 Modules

Modules are subdirectories in `modules/` that contain Block definitions. If you are writing a soft Block then you will typically create a new Module for it. If you are writing a Block with hardware connections it will live in a Module for that hardware (e.g. for the FMC card, or for that *Target Platform*).

To create a new module, simply create a new directory in `modules/`

9.3 Block ini

The first thing that should be defined when creating a new Block is the interface to the rest of the framework. This consists of an ini file that contains all the information that the framework needs to integrate some VHDL logic into the system. It lives in the Module directory and has the extension `.block.ini`. It consists of a top level section with information about the Block, then a section for every *Field* in the Block.

9.3.1 The [.] section

The first entry to the ini file describes the block as a whole. It looks like this:

```
[.]
description: Short description of the Block
entity: vhdl_entity
type: dma or sfp or fmc
constraints:
ip:
otherconst:
extension:
```

The `description` should be a short (a few words) description that will be visible as a Block label to users of the *PandABlocks Device* when it runs.

The `entity` should be the name of the VHDL entity that will be created to hold the logic. It is typically the lowercase version of the Block name.

The `type` field will identify if the block is an SFP, FMC or DMA. These are special cases and need to be handled differently. This field is automatically set to soft for soft blocks or carrier for carrier blocks.

The `constraints` is used to identify the location of any xdc constraints files, relative to the module's directory.

The `ip` field holds the name of any ip blocks used in the module's vhdl code.

`otherconst` is used to locate a tcl script if the block needs any further configuration.

If the `extension` field is present then the `extensions` directory in the module must exist and contain a python server extension file.

9.3.2 [FIELD] sections

All other sections specify the Field that will be present in the Block. They look like this:

```
[MYFIELD]
type: type subtype options
description: Short description of the Field
extension: extension-parameter
extension_reg:
wstb:
```

The section name is used to determine the name of the Field in the resulting Block. It should be made of upper case letters, numbers and underscores.

The `type` value gives information about the type which specifies the purpose and connections of the Field to the system. It is passed straight through to the field specific line in the config file for the TCP server so should be written according to type documentation. Subsequent indented lines in the config file are supplied according to the `type` value and are documented in *Extra Field Keys*.

The `description` value gives a short (single sentence) description about what the Field does, visible as a tooltip to users.

If `extension` is specified then this field is configured as an extension field. If the `extension_reg` field is also specified then this field is also a hardware register.

If a signal uses a write strobe `wstb` should be set to True.

9.3.3 Extra Field Keys

Some field types accept extra numeric keys in the Field section to allow extra information to be passed to the TCP server via its config file.

Enum fields would contain numeric keys to translate specific numbers into user readable strings. Strings should be lowercase letters and numbers with underscores and no spaces. A typical field might look like this:

```
[ENUM_FIELD]
type: param enum  # or read enum or write enum
description: Short description of the Field
0: first_value
1: next_value
2: another_value
8: gappy_value
```

Tables will be defined here too

9.4 Block Simulation

The Block simulation framework allows the behaviour to be specified in Python and timing tests to be written against it without writing any VHDL. This is beneficial as it allows the behaviour of the Block to be tied down and documented while the logic is relatively easy to change. It also gives an accurate simulation of the Block that can be used to simulate an entire *PandABlocks Device*.

The first step in making a Block Simulation is to define the imports:

```
from common.python.simulations import BlockSimulation, properties_from_ini, \
    TYPE_CHECKING

if TYPE_CHECKING:
    from typing import Dict
```

The `typing` imports allow IDEs like PyCharm to infer the types of the variables, increasing the chance of finding bugs at edit time.

The BlockSimulation is a baseclass that our simulation should inherit from:

class common.python.simulations.**BlockSimulation**

changes = None
This will be dictionary with changes pushed by any properties created with properties_from_ini()

classmethod bits_to_int(*bits*)
Convert 32 element bit array into an int number

on_changes(*ts*, *changes*)
Handle field changes at a particular timestamp

Parameters

- **ts** (*int*) – The timestamp the changes occurred at
- **changes** (*dict*) – Field names that changed with their integer value

Returns If the Block needs to be called back at a particular ts then return that int, otherwise return None and it will be called when a field next changes

Next we read the block ini file:

```
NAMES, PROPERTIES = properties_from_ini(__file__, "myblock.block.ini")
```

This generates two objects:

- NAMES: A collections.namedtuple with a string attribute for every field, for comparing field names with.
- PROPERTIES: A property for each Field of the Block that can be attached to the *BlockSimulation* class

Now we are ready to create our simulation class:

```
class MyBlockSimulation(BlockSimulation):
    INP, ANOTHER_FIELD, OUT = PROPERTIES

    def on_changes(self, ts, changes):
        """Handle field changes at a particular timestamp

        Args:
            ts (int): The timestamp the changes occurred at
            changes (Dict[str, int]): Fields that changed with their value

        Returns:
             If the Block needs to be called back at a particular ts then return
             that int, otherwise return None and it will be called when a field
             next changes
        """
        # Set attributes
        super(MyBlockSimulation, self).on_changes(ts, changes)

        if NAMES.INP in changes:
            # If our input changed then set our output high
            self.OUT = 1
            # Need to be called back next clock tick to set it back
            return ts + 1
        else:
            # The next clock tick set it back low
            self.OUT = 0
```

This is a very simple Block, when INP changes, it outputs a 1 clock tick pulse on OUT. It checks the changes dict to see if INP is in it, and if it is then sets OUT to 1. The framework only calls on_changes() when there are changes unless informed when the Block needs to be called next. In this case we need to be called back the next clock tick to set OUT back to zero, so we do this by returning ts + 1. When we are called back next clock tick then there is nothing in the changes dict, so OUT is set back to 0 and return None so the framework won't call us back until something changes.

Note: If you need to use a field name in code, use an attribute of NAMES. This avoids mistakes due to typos like:

```
if "INPP" in changes:
    code_that_will_never_execute
```

While if we use NAMES:

```
if NAMES.INPP in changes:  # Fails with AttributeError
```

9.5 Timing ini

The purpose of the .timing.ini file is to provide expected data for comparison in the testing of the modules. Data should be calculated as to how and when the module will behave with a range of inputs.

9.5.1 The [.] section

The first entry to the ini file describes the timing tests as a whole. It looks like this:

```
[.]
description: Timing tests for Block
scope: block.ini file
```

9.5.2 [TEST] sections

The other sections will display the tests inputs and outputs. It looks like this:

```
[NAME_OF_TEST]
1:  inputA=1, inputB=2          -> output=3
5:  inputC=4                    -> output=7
6:  inputD=-10                  -> output=0, Error=1
```

The numbers at the left indicate the timestamp at which a change occurs, followed by a colon. Any assignments before the -> symbol indicate a change in an input and assignments after the -> symbol indicate a change in an output.

9.6 Target ini

A target.ini is written for the blocks which are specific to the target. This ini file declares the blocks and their number similar to the app.ini file.

9.6.1 The [.] section

The first entry to the ini file defines information for the SFP sites for the target:

```
[.]
sfp_sites:
sfp_constraints:
```

The `sfp_sites` type is the number of available SFP sites on the target, and the `sfp_sites` type is the name of the constraints file for each SFP site, located in the target/const directory.

9.6.2 [BLOCK] sections

The block sections are handled in the same manner as those within the app.ini file, however the type, unless overwritten in the block.ini files for these blocks is set to carrier, rather than soft.

9.7 Writing docs

Two RST directives, how to structure

9.8 Block VHDL entity

How to structure the VHDL entity

CHAPTER 10

Autogeneration framework architecture

10.1 Softblocks

10.2 Wrappers

How wrapper, config, desc, vhdl entities, test benches are generated

10.3 Config_d entries

common.python.configs.**pad**(*name*, *spaces=19*)
Pad the right of a name with spaces until it is at least spaces long

common.python.configs.**all_subclasses**(*cls*)
Recursively find all the subclasses of cls

class common.python.configs.**BlockConfig**(*name*, *type*, *number*, *ini_path*, *site=None*)
The config for a single Block

name = None
The name of the Block, like LUT

number = None
The number of instances Blocks that will be created, like 8

module_path = None
The path to the module that holds this block ini

ini_path = None
The path to the ini file for this Block, relative to ROOT

block_address = None
The Block section of the register address space

site = None
If the type == sfp, which site number

entity = None
The VHDL entity name, like lut

type = None
Is the block soft, sfp, fmc or dma?

constraints = None
Any constraints?

ip = None
Does the block require IP?

description = None
The description, like "Lookup table"

fields = None
All the child fields

block_suffixes = None
Are there any suffixes?

register_addresses(*block_counters*)
Register this block in the address space

filter_fields(*regex*, *matching=True*)
Filter our child fields by typ. If not matching return those that don't match

generateInterfaceConstraints()
Generate MGT Pints constraints

class common.python.configs.**RegisterConfig**(*name*, *number=-1*, *prefix=''*, *extension=''*)
A low level register name and number backing this field

name = None
The name of the register, like INPA_DLY

number = None
The register number relative to Block, like 9

extension = None
For an extension field, the register path

class common.python.configs.**BusEntryConfig**(*name*, *bus*, *index*)
A bus entry belonging to a field

name = None
The name of the register, like INPA_DLY

bus = None
The bus the output is on, like bit

index = None
The bus index, like 5

class common.python.configs.**FieldConfig**(*name*, *number*, *type*, *description*, *extra_config*)
The config for a single Field of a Block

type_regex = None
Regex for matching a type string to this field

name = None
The name of the field relative to it's Block, like INPA

number = None
The number of instances Blocks that will be created, like 8

type = None
The complete type string, like param lut

description = None
The long description of the field

registers = None
The list of registers this field uses

bus_entries = None
The list of bus entries this field has

wstb = None
If a write strobe is required, set wstb to 1

extension = None
Store the extension register info

extra_config_lines = None
All the other extra config items

parse_extra_config(*extra_config*)
Produce any extra config lines from self.kwargs

register_addresses(*counters*)
Create registers using the FieldCounter object

address_line()
Produce the line that should go in the registers file after name

config_line()
Produce the line that should go in the config file after name

numbered_registers()
Filter self.registers, only producing registers with a number (not those that are purely extension registers)

class common.python.configs.**BitOutFieldConfig**(*name*, *number*, *type*, *description*, *extra_config*)
These fields represent a single entry on the bit bus

register_addresses(*counters*)
Create registers using the FieldCounter object

class common.python.configs.**PosOutFieldConfig**(*name*, *number*, *type*, *description*, *extra_config*)
These fields represent a position output

register_addresses(*counters*)
Create registers using the FieldCounter object

parse_extra_config(*extra_config*)
Produce any extra config lines from self.kwargs

config_line()
Produce the line that should go in the config file after name

class common.python.configs.**ExtOutFieldConfig**(*name*, *number*, *type*, *description*, *extra_config*)

These fields represent a ext output

register_addresses(*counters*)
Create registers using the FieldCounter object

class common.python.configs.**ExtOutTimeFieldConfig**(*name*, *number*, *type*, *description*, *extra_config*)

These fields represent a ext output timestamp, which requires two registers

register_addresses(*counters*)
Create registers using the FieldCounter object

class common.python.configs.**TableFieldConfig**(*name*, *number*, *type*, *description*, *extra_config*)

These fields represent a table field

words = None
How many 32-bit words per line?

register_addresses(*counters*)
Create registers using the FieldCounter object

config_line()
Produce the line that should go in the config file after name

parse_extra_config(*extra_config*)
Produce any extra config lines from self.kwargs

class common.python.configs.**TableShortFieldConfig**(*name*, *number*, *type*, *description*, *extra_config*)

These fields represent a table field

lines = None
How many lines in the table?

parse_extra_config(*extra_config*)
Produce any extra config lines from self.kwargs

register_addresses(*counters*)
Create registers using the FieldCounter object

class common.python.configs.**ParamFieldConfig**(*name*, *number*, *type*, *description*, *extra_config*)

These fields represent all other set/get parameters backed with a single register

register_addresses(*counters*)
Create registers using the FieldCounter object

class common.python.configs.**EnumParamFieldConfig**(*name*, *number*, *type*, *description*, *extra_config*)

An enum field with its integer entries and string values

parse_extra_config(*extra_config*)
Produce any extra config lines from self.kwargs

class common.python.configs.**UintParamFieldConfig**(*name*, *number*, *type*, *description*, *extra_config*)

A special These fields represent all other set/get parameters backed with a single register

parse_extra_config(*extra_config*)
Produce any extra config lines from self.kwargs

config_line()
Produce the line that should go in the config file after name

class common.python.configs.**ScalarParamFieldConfig**(*name*, *number*, *type*, *description*, *extra_config*)
A special Read config for reading the different config of a read scalar

parse_extra_config(*extra_config*)
Produce any extra config lines from self.kwargs

config_line()
Produce the line that should go in the config file after name

class common.python.configs.**BitMuxFieldConfig**(*name*, *number*, *type*, *description*, *extra_config*)
These fields represent a single entry on the pos bus

register_addresses(*counters*)
Create registers using the FieldCounter object

class common.python.configs.**PosMuxFieldConfig**(*name*, *number*, *type*, *description*, *extra_config*)
The fields represent a position input multiplexer selection

register_addresses(*counters*)
Create registers using the FieldCounter object

class common.python.configs.**TimeFieldConfig**(*name*, *number*, *type*, *description*, *extra_config*)
The fields represent a configurable timer parameter

register_addresses(*counters*)
Create registers using the FieldCounter object

class common.python.configs.**TargetSiteConfig**(*name*, *info*)
The config for the target sites

type_regex = None
Regex for matching a type string to this field

name = None
The type of target site (SFP/FMC etc)

number = None
The info i in a string such as "3, i, io, o"

10.4 Test benches

A generic outline is common across the testbenches for the different blocks. There are four main areas of required functionality: Assigning signals, reading expected data, assigning inputs to the UUT and reading the outputs and comparing the outputs.

A template can therefore be used to autogenerate the testbench, with this common functionality, with the modifications required for use with the different blocks.

10.4.1 Required signals in the block

Python code used has extracted the different signals which are required from the `.block.ini` file for each block. Using this information, register signals are produced in the testbench for each signal, using the names from the INI file.

However, not all signals are used in the same manner. Therefore the field type of each signal is also read to determine the size of the required register for each signals. This is also used to determine whether the signal is an input or an output signal. Each output signal requires a register signal similar to the the inputs, however they also require wire signals for use with the UUT, this is differentiated by the suffix "_UUT" and an error register which is differentiated by the suffix "_error".

Integer signals are also declared for holding the file identifier, the `$fscanf` return value and the timestamp.

10.4.2 Read expected.csv

From the `.timing.ini` file within the block, a CSV file is generated which describes how the UUT should behave under certain inputs at different times. The first line of the file contains strings with the names of each of the signals, the first column being the timestamp data. All other lines contain numeric data for the timestamp, inputs and corresponding outputs.

The file is opened in the testbench and read line by line. The first line, containing the names of the signals is discarded. The numeric data is then read, when the timestamp value is equal to that in the file the values are assigned to the corresponding registers in the testbench. The data in the file is ordered in the same way as the .block.ini file so iterating through the signals in order, will assign the data to the correct registers.

10.4.3 Assign signals

The inputs to the entity for the block will have the same name as for those used in the testbench. It is therefore straightforward to connect the signals. The registers with the same name as the outputs are being used for holding the expected values, therefore the wire signals with the suffix "_uut" are used to read the output signals.

10.4.4 Compare output signals

To verify the correct functionality of the block, the outputted values will need to be compared to the expected values. A simple comparison is implemented, if the two signals are not equal, set that output's error signal to one and display an error message to the user.

CHAPTER 11

Change Log

All notable changes to this project will be documented in this file. This project adheres to Semantic Versioning.

11.1 Unreleased

Added:

- Started a changelog

Changed:

- Interface to the server, require 1.0 release of the server package

CHAPTER 12

Glossary

This section defines some commonly used PandABlocks terms.

12.1 App

An ini file that contains the type and number of Blocks that should be built together to form an FPGA image (loadable on a PandABlocks device as a *Zpkg*).

12.2 Block

A piece of FPGA logic that has a number of *Field* instances and does some specified calculations on each FPGA clock tick. It may be a soft Block like a SEQ, or have hardware connections like a TTLIN Block.

12.3 Field

An input, output or parameter of a *Block*.

12.4 Module

A directory containing *Block* definitions, logic, simulations and timing. Modules will typically contain a single soft Block definition, or a number of hardware Blocks tied to a particular *Target Platform*, SFP or FMC card.

12.5 PandABox

A *PandABlocks Device* manufactured by Diamond Light Source and SOLEIL. Schematics on Open Hardware

12.6 PandABlocks Device

A Zynq 7030 based device loaded with PandABlocks rootfs so that it runs the PandABlocks framework.

12.7 Target Platform

The physical Zynq based hardware that will be loaded with firmware to become a *PandABlocks Device* like a *PandABox* or a Picozed Carrier

12.8 Zpkg

A specially formatted tar file of built files that can be deployed to a PandABlocks device

CHAPTER 13

Running the tests

There are a number of different test systems in place within the PandABlocks-FPGA directory. There are python tests to check the output of some of the Jinja2 templates, python tests to check the logic of the timing diagram and then there are hdl testbenches which test the functionality of the blocks. The python tests are ran as part of the Travis tests when a commit is made to the git repository, however the hdl testbenches have to be manually ran.

13.1 Python tests

The first of the python tests, checking the output of the Jinja2 templates, can be run from the Makefile:

```
make python_tests
```

The python simulation tests, can be run with the following Makefile command:

```
make python_timing
```

13.2 HDL tests

There are two Makefile functions which can be used to run the hdl testbenches:

```
make hdl_test (MODULE="module name")

make single_hdl_test TEST="MODULE_NAME TEST_NUMBER"
```

The first, by default, will run every testbench. However if the optional argument of MODULE is given it will instead run every test for the specified module. Please note that the module name is the entity name for the top level hdl filein that module.

The second command will run a single testbench as specified by the module name, and the test number separated by a space.

Python Module Index

C

Index

S

T

U

W