zebra2-server Documentation

Release 0.0

Michael Abbott, Tom Cobb

Dec 07, 2018

Contents

Contents:

CHAPTER 1

Zebra2

This is what Zebra2 is

CHAPTER 2

Block functional documentation

Each block is documented with examples

2.1 BITS - Soft inputs and constant bits

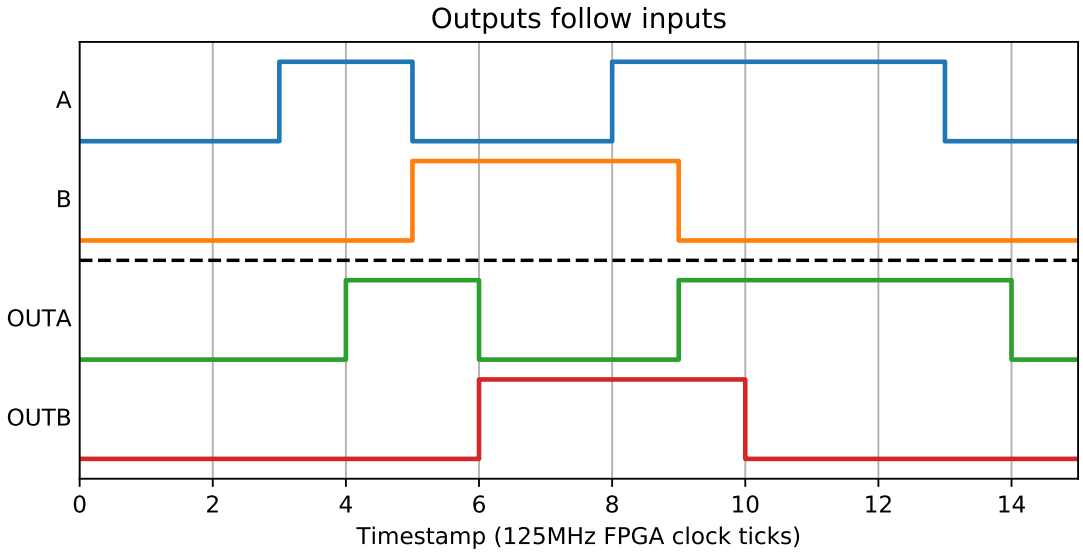
The BITS block contains 4 soft values A..D. Each of these soft values can be set to 0 or 1 by using the SET_A..SET_D parameters.

2.1.1 Parameters

Name	Dir	Type	Description
A	R/W	Bit	The value that output A should take
B	R/W	Bit	The value that output B should take
C	R/W	Bit	The value that output C should take
D	R/W	Bit	The value that output D should take
OUTA	Out	Bit	The value of A on the bit bus
OUTB	Out	Bit	The value of B on the bit bus
OUTC	Out	Bit	The value of C on the bit bus
OUTD	Out	Bit	The value of D on the bit bus

2.1.2 Outputs follow parameters

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



2.2 CLOCKS - Configurable clocks

The CLOCKS block contains 4 user-settable 50% duty cycle clocks. The period can be set for each clock separately. When any clock period is set, all clocks restart from a common synchronous point.

2.2.1 Parameters

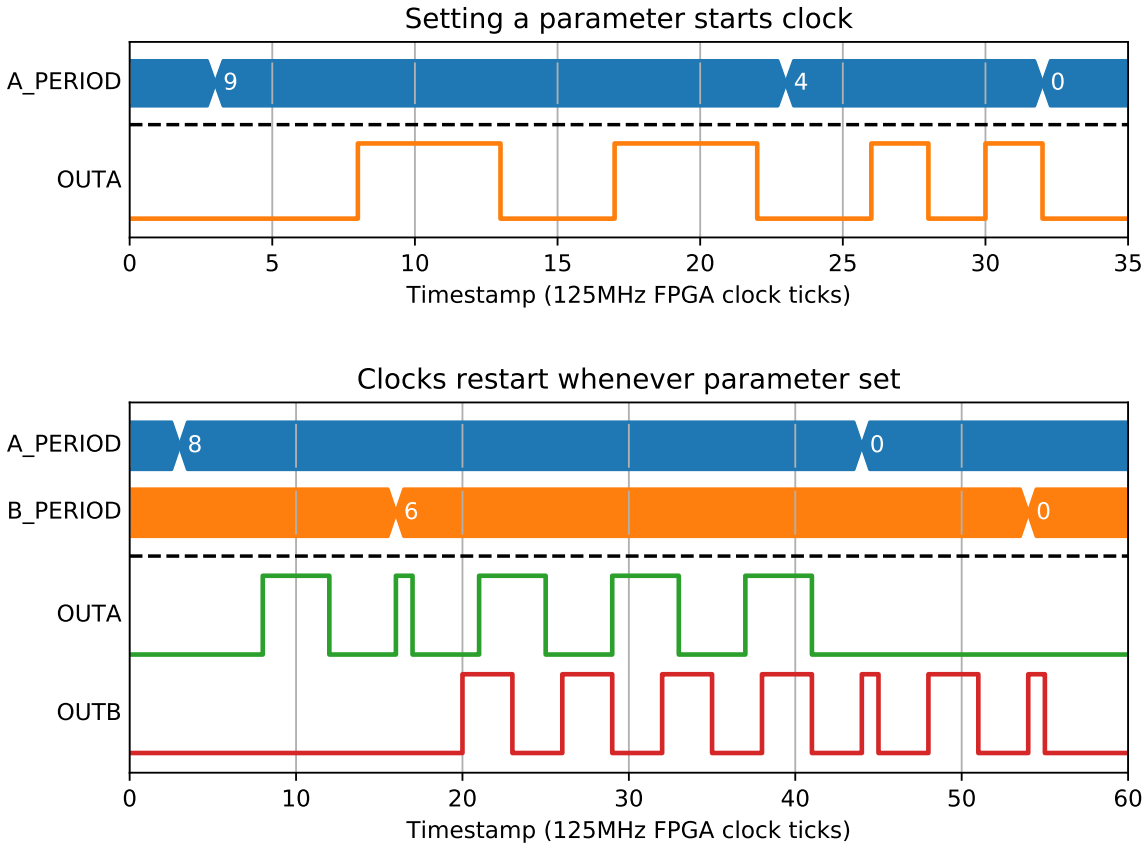
Name	Dir	Type	Description
A_PERIOD	R/W	Time	The period of clock output A, 0 or 1 = off
B_PERIOD	R/W	Time	The period of clock output B, 0 or 1 = off
C_PERIOD	R/W	Time	The period of clock output C, 0 or 1 = off
D_PERIOD	R/W	Time	The period of clock output D, 0 or 1 = off
A	Out	Bit	The current value of clock A
B	Out	Bit	The current value of clock B
C	Out	Bit	The current value of clock C
D	Out	Bit	The current value of clock D

2.2.2 Setting clock period parameters

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

2.2.3 All clocks have the same starting point

When any period parameter is set, all clocks restart from that point.



2.3 PULSE - One-shot pulse delay and stretch [x4]

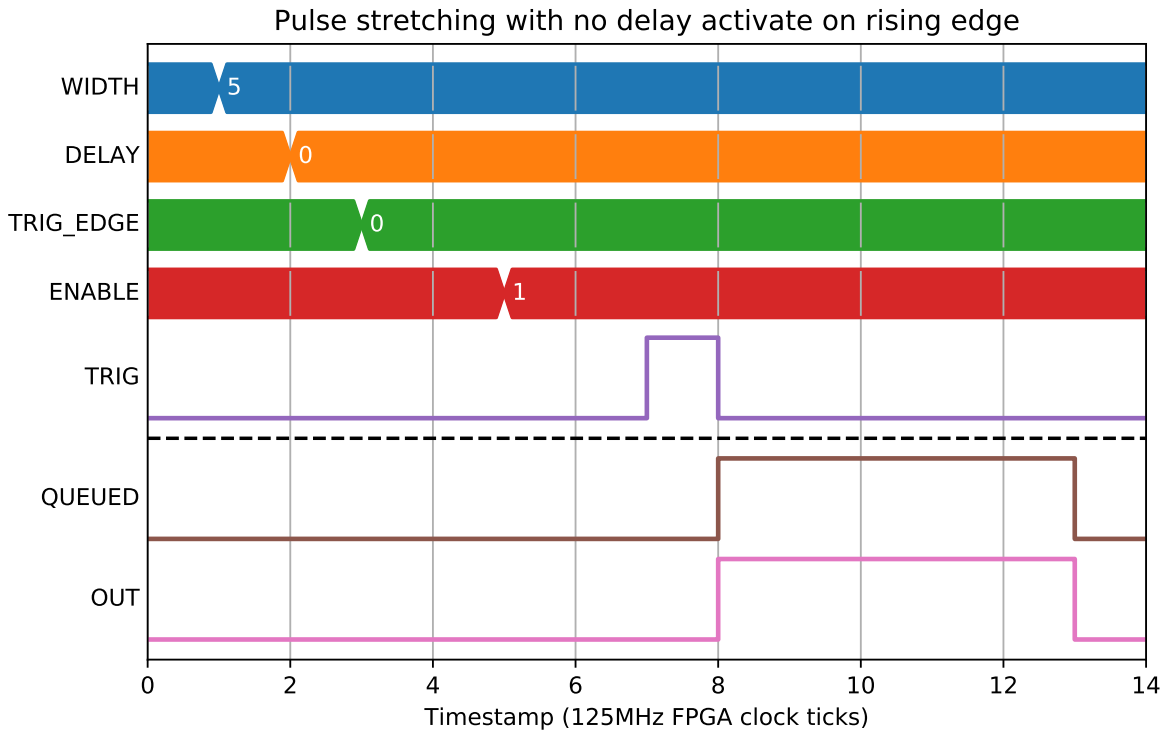
A PULSE block produces configurable width output pulses with an optional delay based on its parameters. If WIDTH is non-zero, the output pulse width will be the specified amount. If DELAY is non-zero, the pulse train will be delayed by that amount. If both are non-zero, the pulses are stretched and delayed as long as the resulting output would still contain the same number of distinct pulses. If this is not the case, then the PERR signal is raised, and the MISSED_CNT counter is incremented. Change of any parameter causes the block to be reset.

2.3.1 Parameters

Name	Dir	Type	Description
DELAY	R/W	Time	Output pulse delay. Must be either 0 (no delay) or >4 clock ticks
WIDTH	R/W	Time	Output pulse width. If 0, the width of the input pulse is used
FORCE_RESET	W	Action	Reset QUEUE and ERR outputs
INP	In	Bit	Input pulse train
RESET	In	Bit	On edge defined by EDGE, reset QUEUE and ERR outputs
EDGE	R/W	Enum	0 - rising edgee 1 - falling edg 2 - either edge
OUT	Out	Bit	Output pulse train
PERR	Out	Bit	Error output. If a pulse could not be generated This will be set to 1 until the block is RESET
ERR_OVERFLOW	R	Bit	Indicates a missed pulse was due to overflow of the internal queue. If DELAY is non-zero then up to 1023 pulse edges can be queued waiting for output.
ERR_PERIOD	R	Bit	If producing a pulse would cause it to overlap with the previous pulse (WIDTH > time between pulses), then this flag is set.
QUEUE	R	UInt32	Length of the delay queue in range [0..1023]
MISSED_CNT	R	UInt32	Number of pulses that have not been produced because of an ERR condition. Will only be non-zero when PERR is 1

2.3.2 Zero Delay

If DELAY=0, then the INP pulse will be stretched with only the propogation delay of the block (1 clock tick). WIDTH may take any value, as long as input pulses are spaced enough to allow stretched pulses to be produced.



2.3.3 Zero Width

If WIDTH=0, then the INP pulse width will be used. DELAY must be >4 clock ticks.

2.3.4 Width and Delay

In this mode, pulses are placed onto an output queue, so a number of restrictions apply:

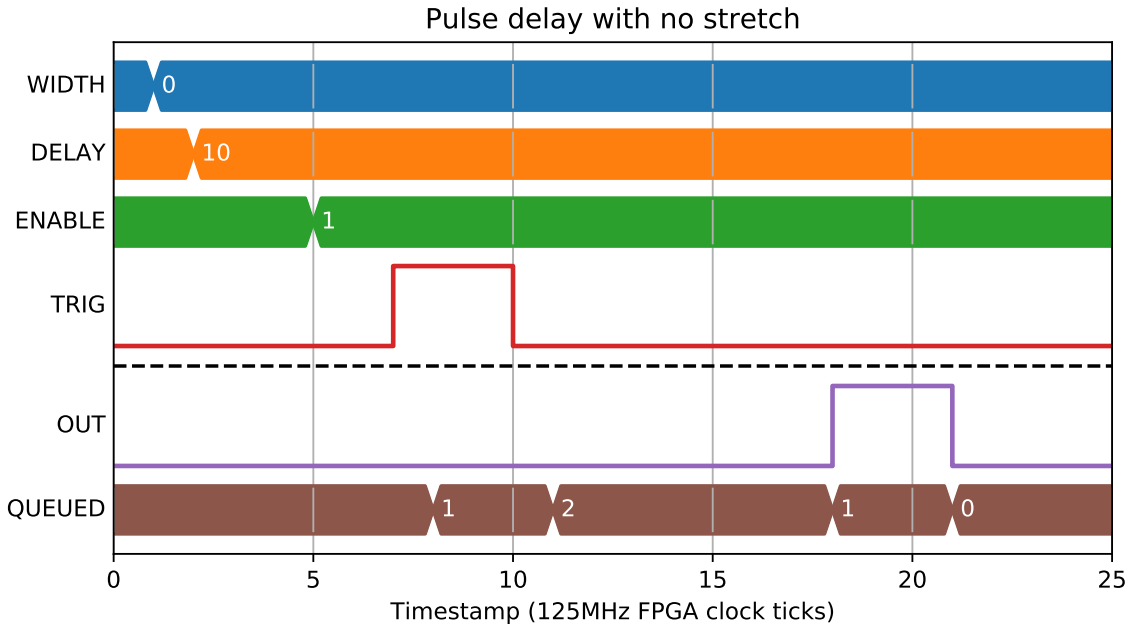
- There must not be more than 1023 pulses on the output queue
- WIDTH must be >3 clock ticks
- There must be >3 clock ticks where output is 0 between pulses. This means that WIDTH < T - 3 where T is the minimum INP pulse period

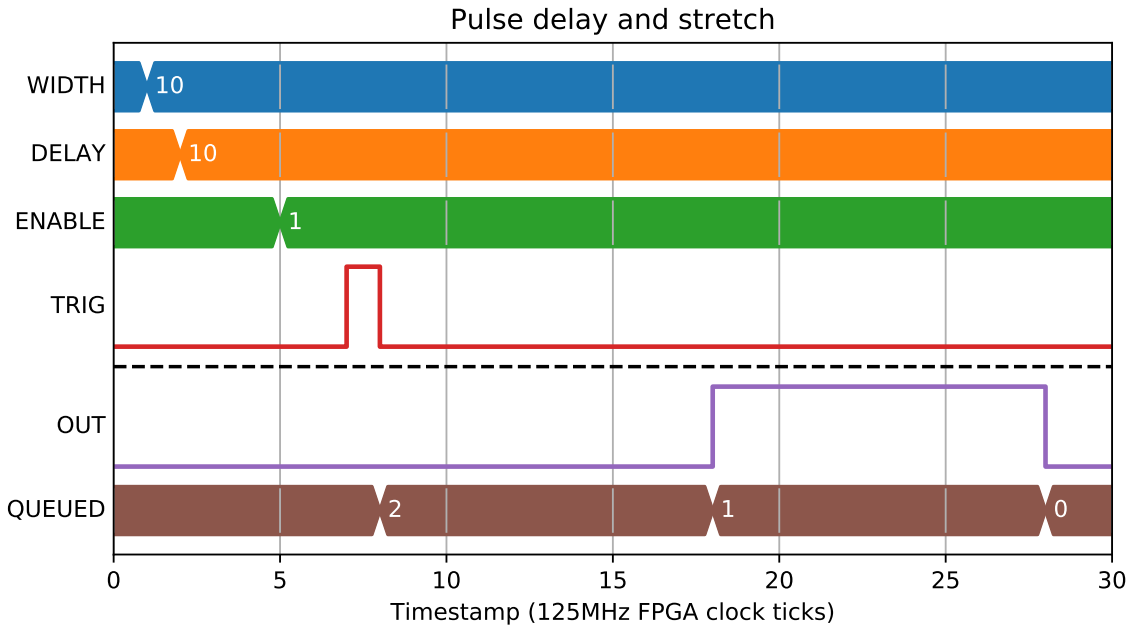
2.3.5 Different Edge Activation

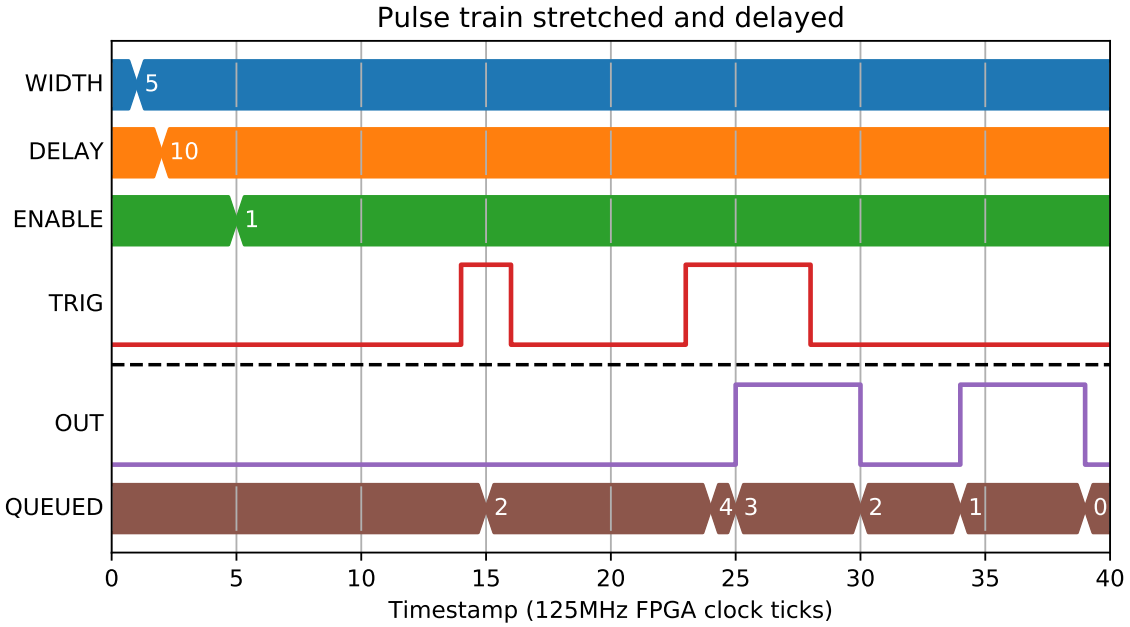
When there is a width specified, it is possible to also specify which edge of the input pulse activates the output.

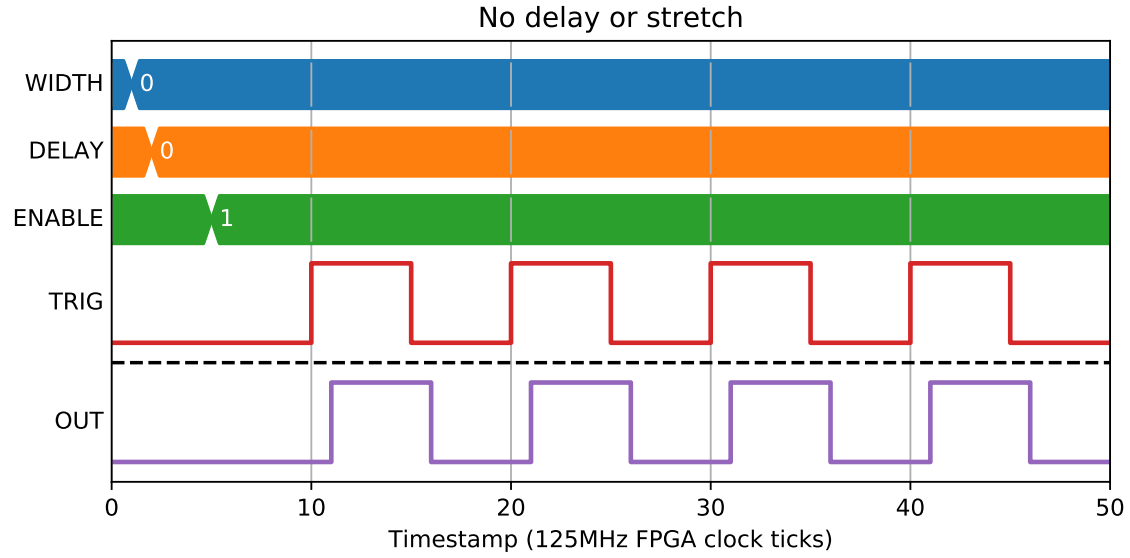
2.3.6 Pulse period error

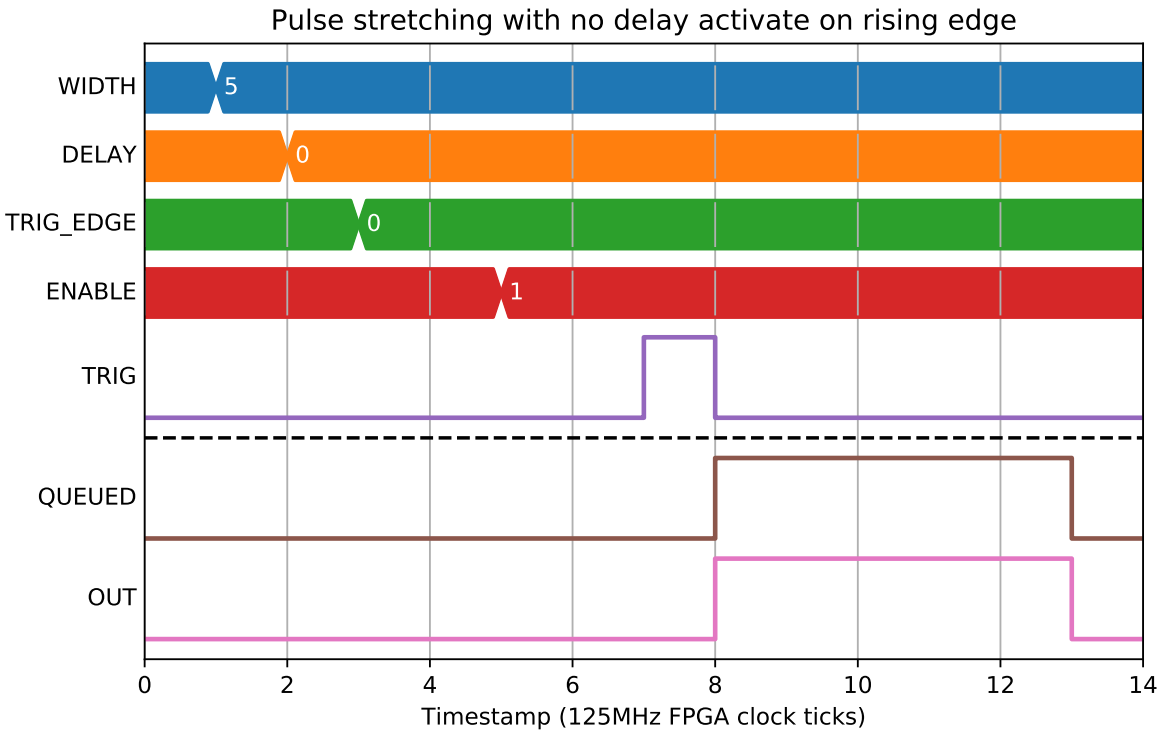
The following example shows what happens when the period between pulses is too short.

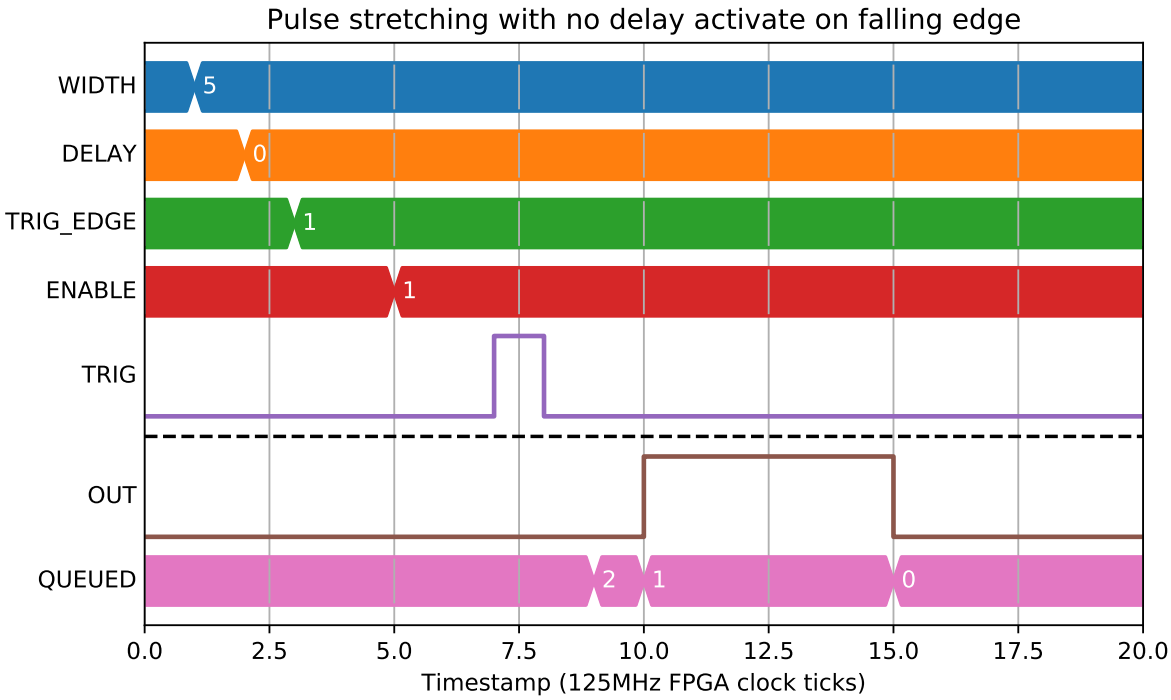

Pulse delay with no stretch
WIDTH
DELAY
ENABLE
TRIG
OUT
QUEUED
0
10
1
1
2
1
0
0
5
10
15
20
25
Timestamp (125MHz FPGA clock ticks)

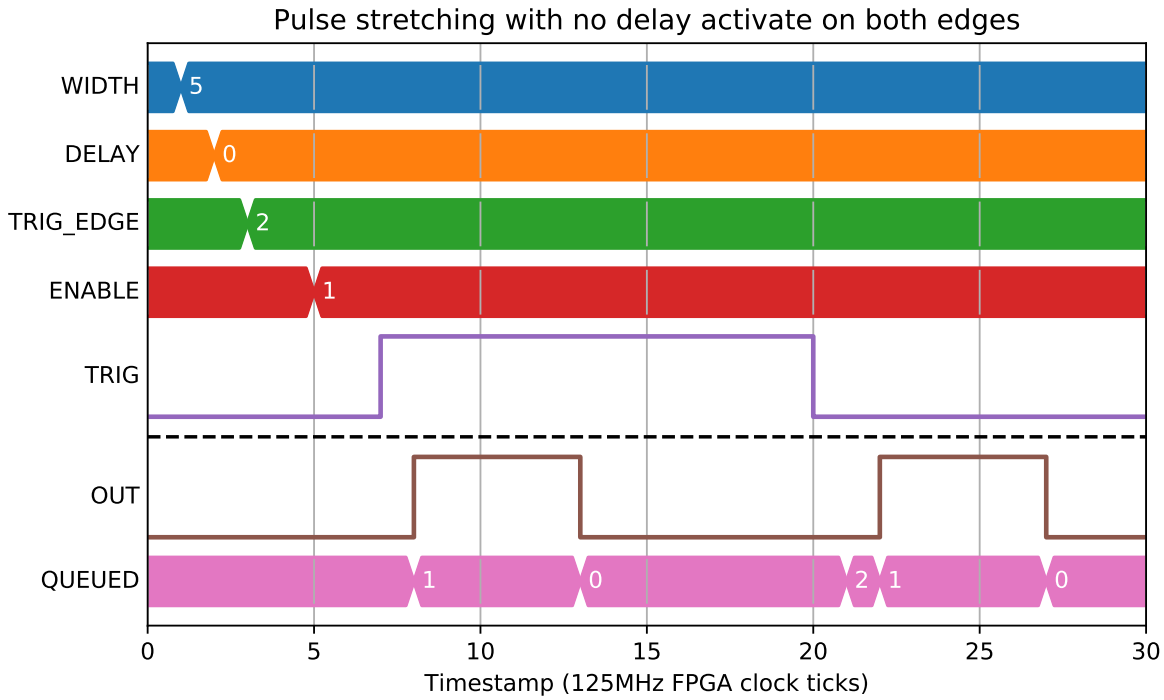

Pulse delay and stretch
WIDTH
DELAY
ENABLE
TRIG
OUT
QUEUED
10
10
1
2
1
0
0
5
10
15
20
25
30
Timestamp (125MHz FPGA clock ticks)

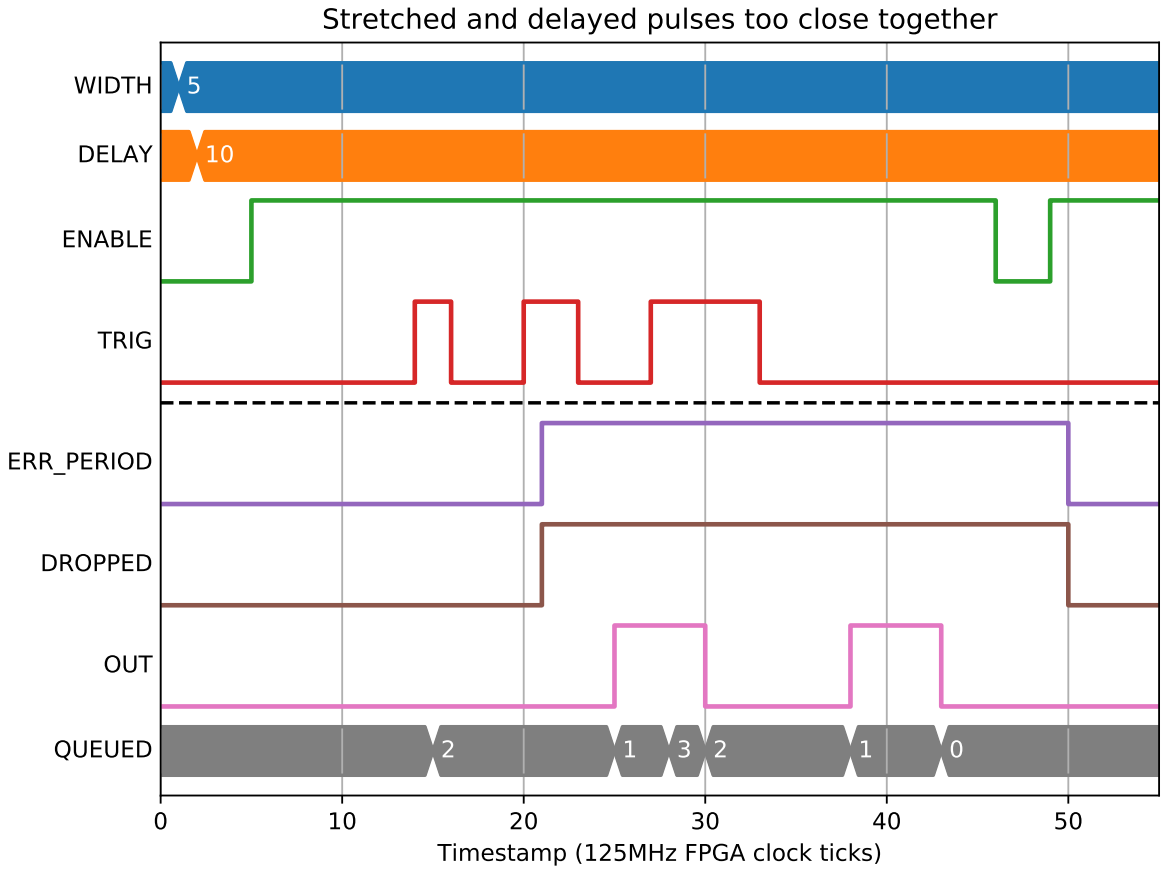

Pulse train stretched and delayed
WIDTH
DELAY
ENABLE
TRIG
OUT
QUEUED
5
10
1
2
4
3
2
1
0
0
5
10
15
20
25
30
35
40
Timestamp (125MHz FPGA clock ticks)


No delay or stretch
WIDTH
DELAY
ENABLE
TRIG
OUT
0
0
1
0
10
20
30
40
50
Timestamp (125MHz FPGA clock ticks)


Pulse stretching with no delay activate on rising edge
WIDTH
DELAY
TRIG_EDGE
ENABLE
TRIG
QUEUED
OUT
5
0
0
1
0
2
4
6
8
10
12
14
Timestamp (125MHz FPGA clock ticks)


Pulse stretching with no delay activate on falling edge
WIDTH
DELAY
TRIG_EDGE
ENABLE
TRIG
OUT
QUEUED
5
0
1
1
2
1
0
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Timestamp (125MHz FPGA clock ticks)


Pulse stretching with no delay activate on both edges
WIDTH
DELAY
TRIG_EDGE
ENABLE
TRIG
OUT
QUEUED
5
0
2
1
1
0
2
1
0
0
5
10
15
20
25
30
Timestamp (125MHz FPGA clock ticks)


Stretched and delayed pulses too close together
WIDTH
DELAY
ENABLE
TRIG
ERR_PERIOD
DROPPED
OUT
QUEUED
5
10
2
1
3
2
1
0
0
10
20
30
40
50
Timestamp (125MHz FPGA clock ticks)

2.4 DIV - Pulse divider [x4]

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.

2.4.1 Parameters

Name	Dir	Type	Description
DIVISOR	R/W	UInt32	Divisor value
FIRST_PULSE	R/W	Enum	0 - OutN: Send first pulse to OUTN 1 - OutD: Send first pulse to OUTD
FORCE_RESET	W	Action	Reset internal counter state machine
INP	In	Bit	Input pulse train
RESET	In	Bit	On rising edge, reset counter state machine
OUTD	Out	Bit	Divided pulse output
OUTN	Out	Bit	Non-divided pulse output
OUT	R	UInt32	Internal counter value in range [0..DIVISOR-1)

2.4.2 Which output do pulses go to

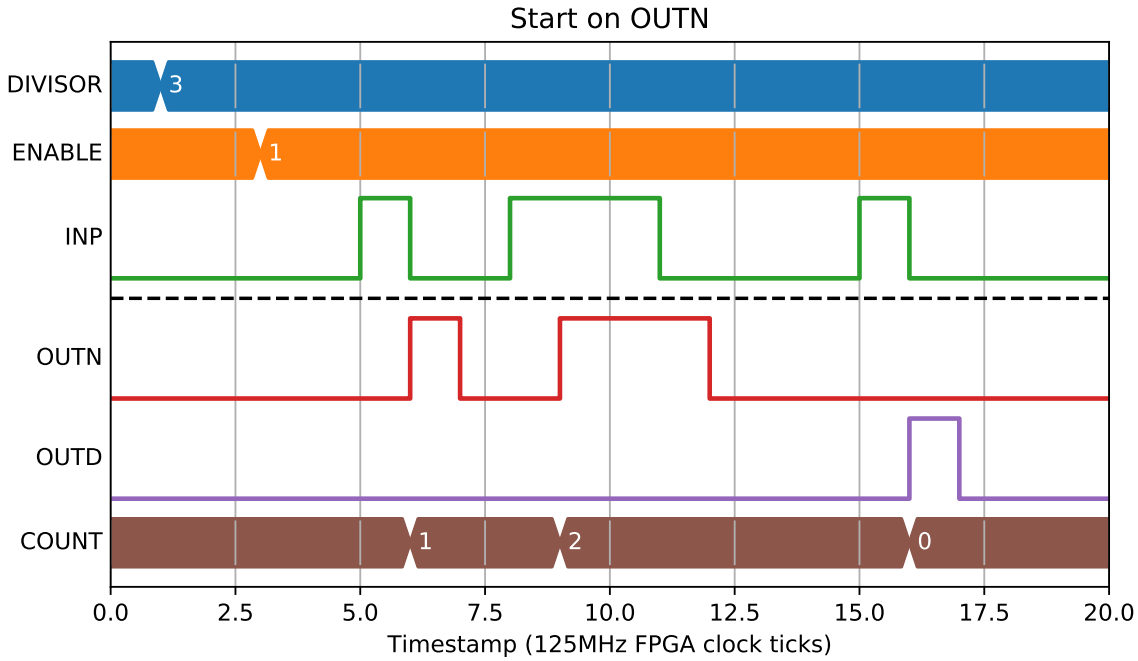
With a DIVISOR of 3, the block will send 1 of 3 INP pulses to OUTD and 2 of 3 INP pulses to OUTN. The following two examples illustrate how the FIRST_PULSE parameter controls the initial value of OUT, which controls whether OUTD or OUTN gets the next pulse.

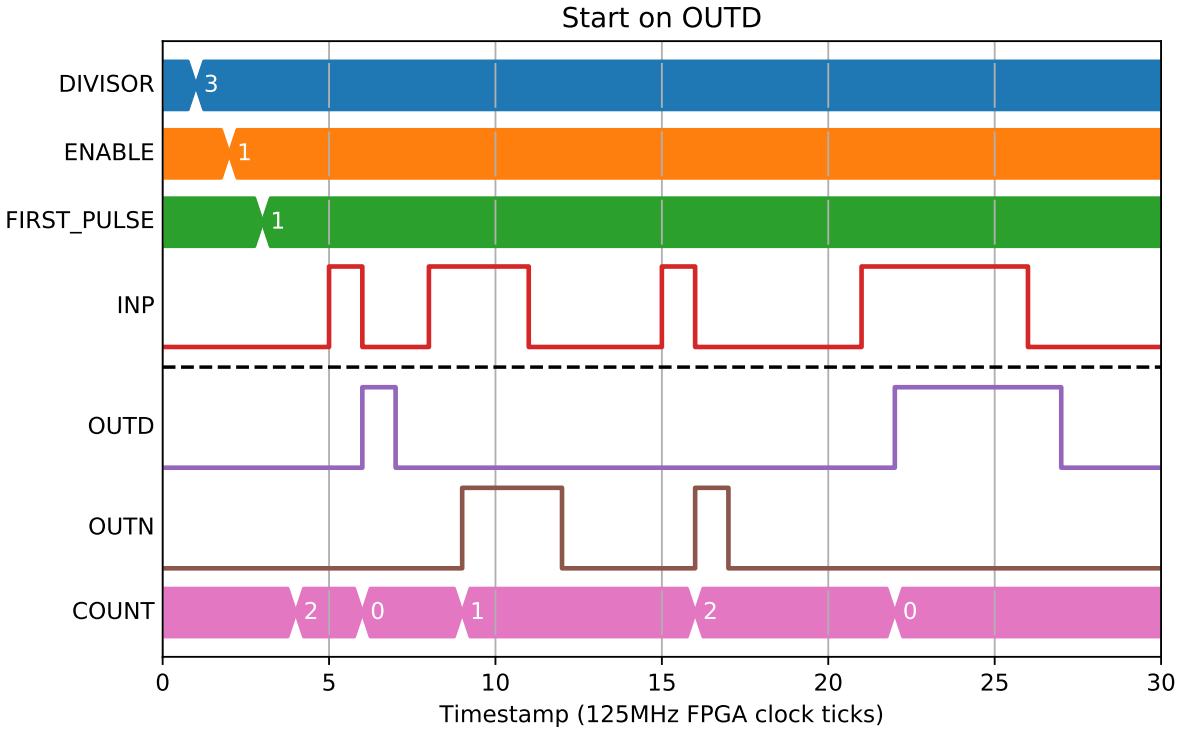
2.4.3 Reset conditions

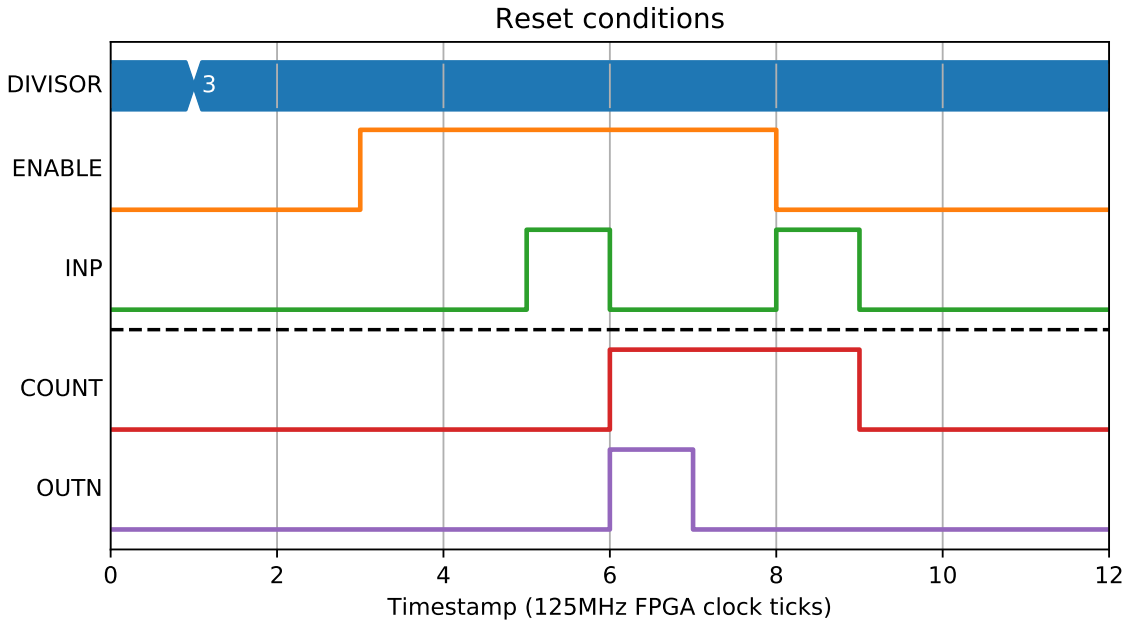
If an ENABLE falling edge is received at the same time as an INP rising edge, the input signal is ignored and the block reset.

2.5 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.


Start on OUTN
DIVISOR
ENABLE
INP
OUTN
OUTD
COUNT
3
1
1
2
0
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Timestamp (125MHz FPGA clock ticks)


Start on OUTD
DIVISOR
ENABLE
FIRST_PULSE
INP
OUTD
OUTN
COUNT
3
1
1
2
0
1
2
0
0
5
10
15
20
25
30
Timestamp (125MHz FPGA clock ticks)


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

2.5.1 Parameters

Name	Dir	Type	Description
WHEN_DISABLED	R/W	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	R/W	Enum	0 - Sets the output to 1 on rising edge 1 - Sets the output to 1 on falling edge 2 - Sets the output to 1 on either edge
RESET_EDGE	R/W	Enum	0 - Resets the output on rising edge 1 - Resets the output on falling edge 2 - Resets the output on either edge
FORCE_RESET	W	Action	Reset output to 0
FORCE_SET	W	Action	Set output to 0
ENABLE	In	Bit	Whether to listen to SET/RST events
SET	In	Bit	A falling/rising edge sets the output to 1
RESET	In	Bit	A falling/rising edge resets the output to 0
OUT	Out	Bit	Output value

2.5.2 Normal conditions

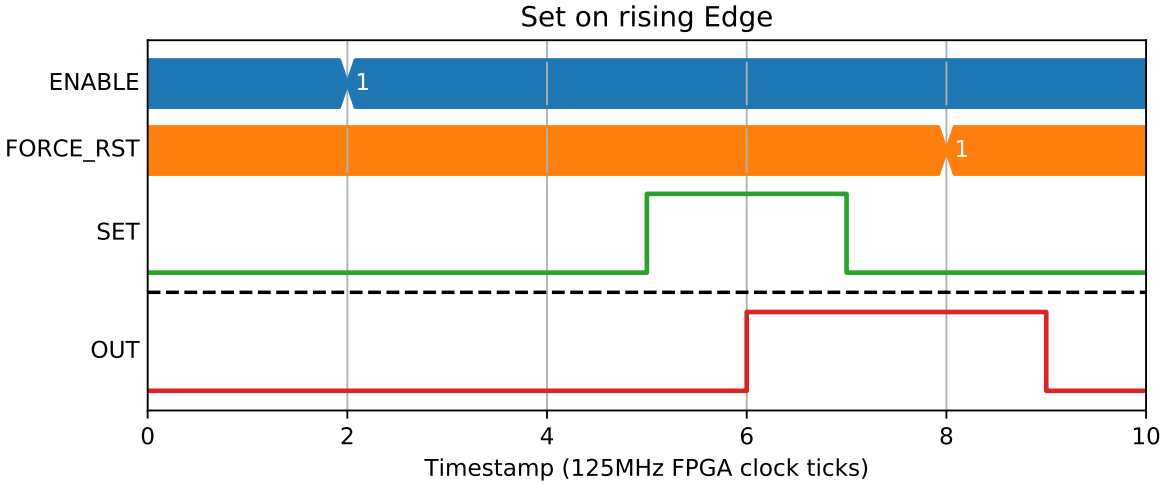
The normal behaviour is to set the output OUT on the configured edge of the SET or RESET input.

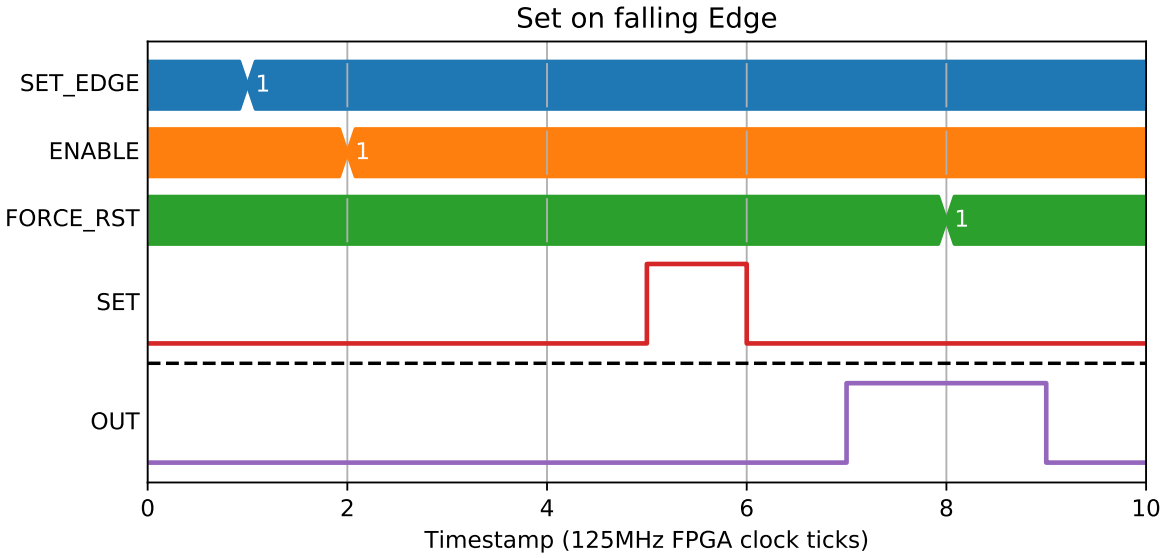
2.5.3 Disabling the block

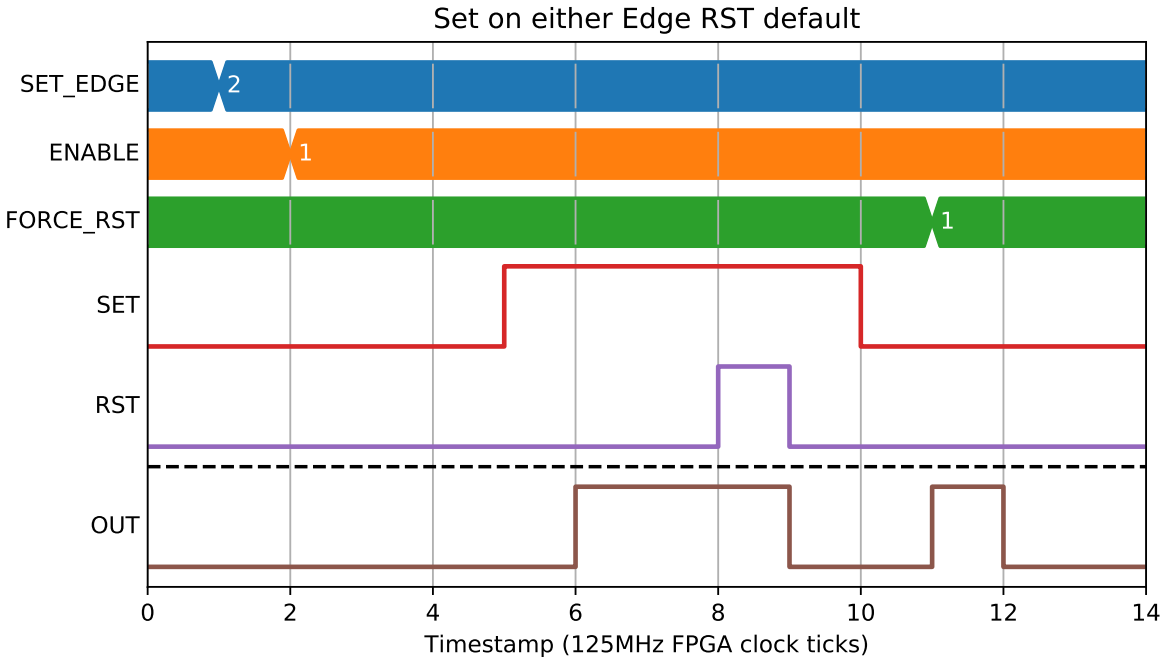
The default behaviour is to force the block output low when disabled, ignoring any SET/RST events:

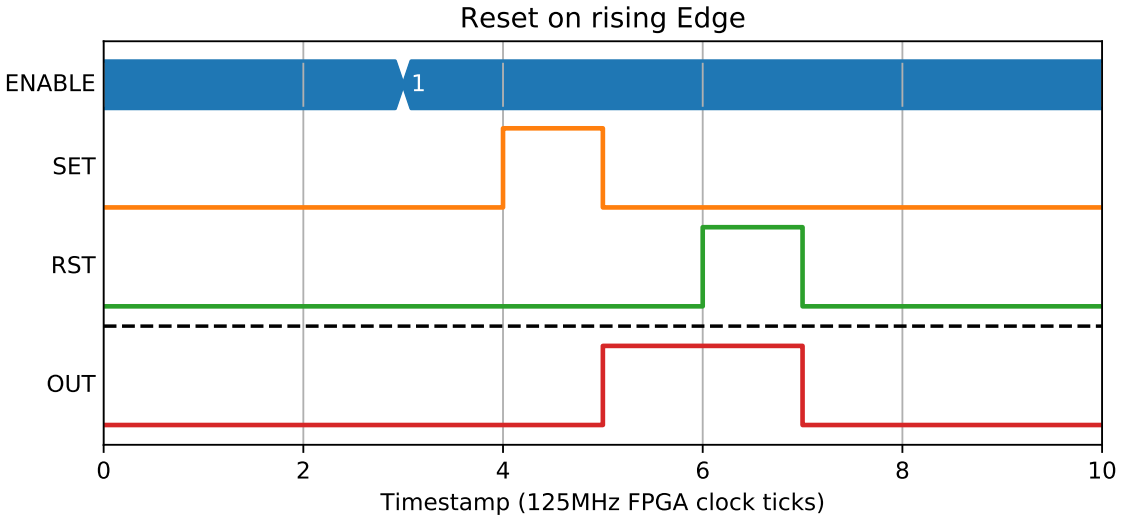
The disabled value can also be set high:

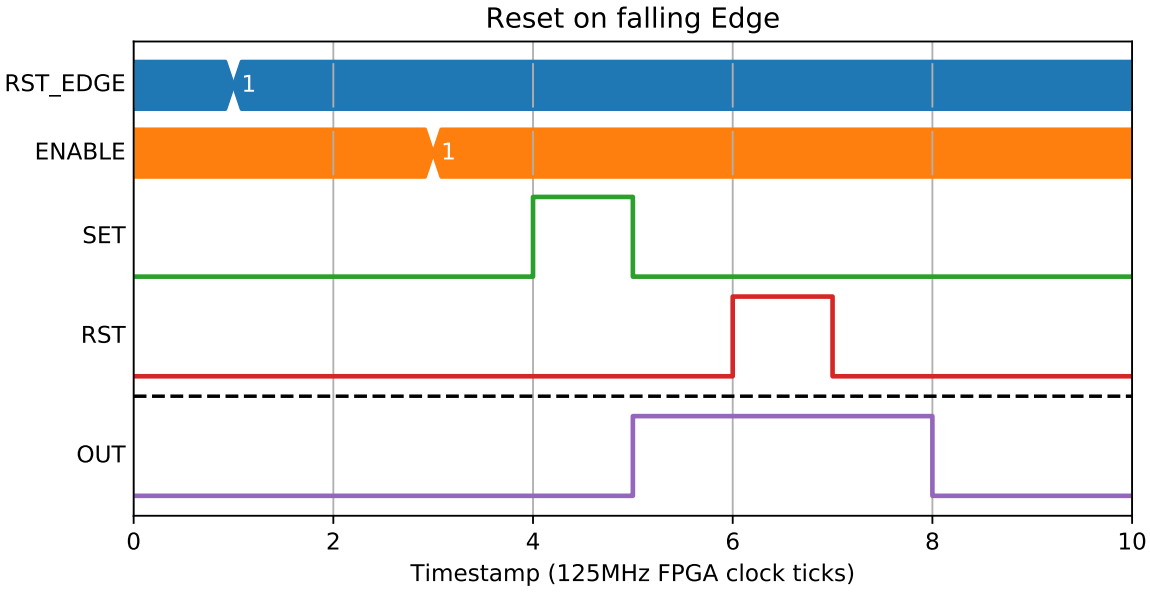
Or left at its current value:

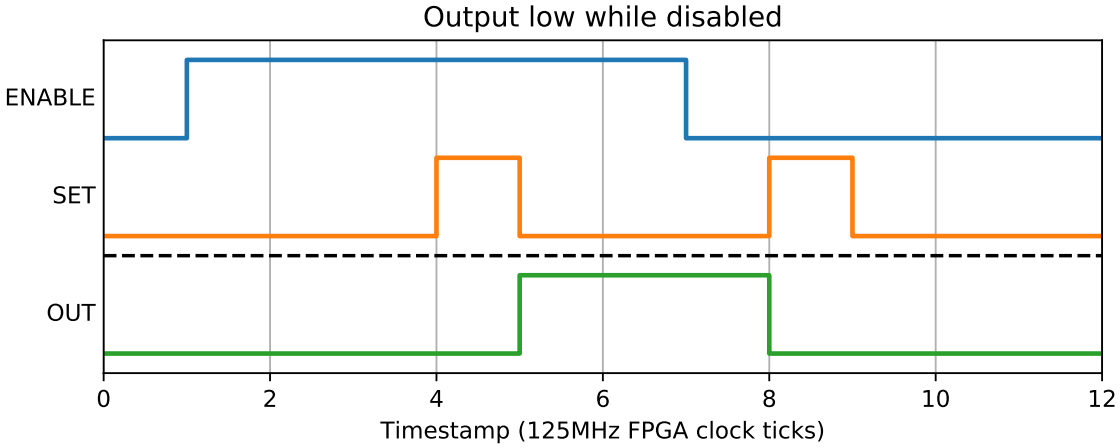

Set on rising Edge
ENABLE
FORCE_RST
SET
OUT
1
1
0
2
4
6
8
10
Timestamp (125MHz FPGA clock ticks)

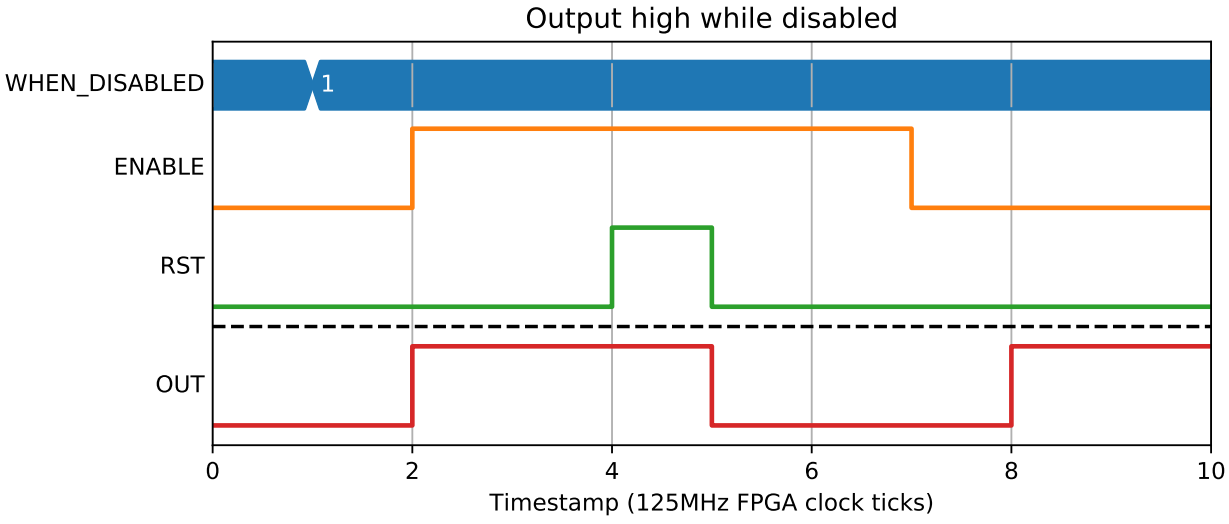

Set on falling Edge
SET_EDGE
ENABLE
FORCE_RST
SET
OUT
1
1
1
0
2
4
6
8
10
Timestamp (125MHz FPGA clock ticks)

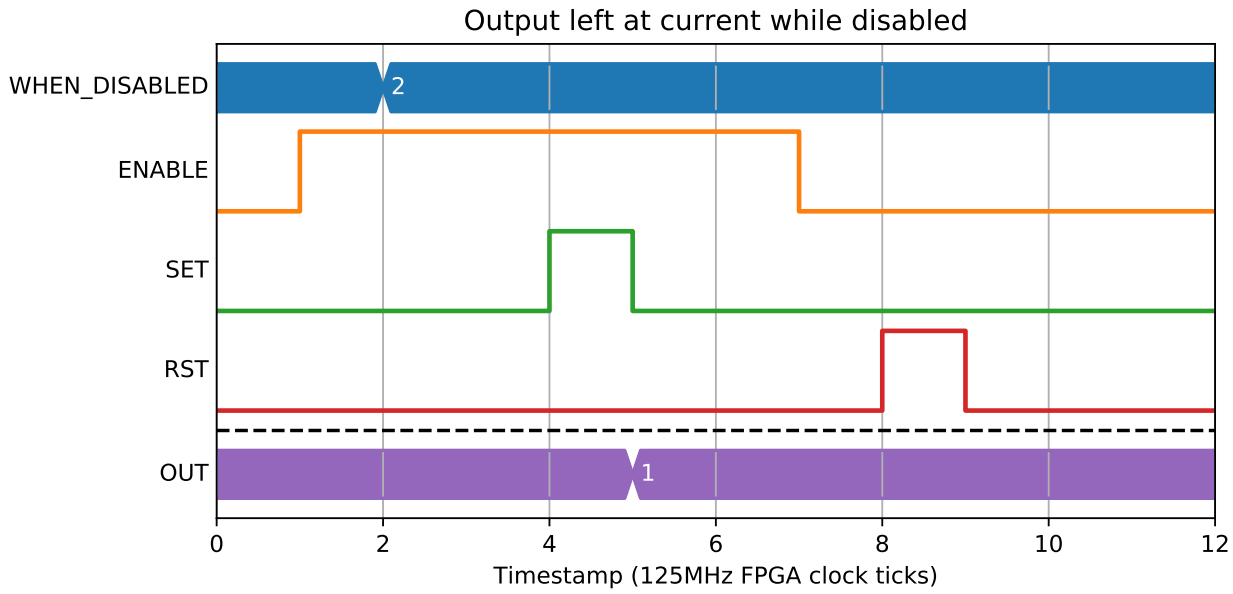

Set on either Edge RST default
SET_EDGE
ENABLE
FORCE_RST
SET
RST
OUT
2
1
1
0
2
4
6
8
10
12
14
Timestamp (125MHz FPGA clock ticks)


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)

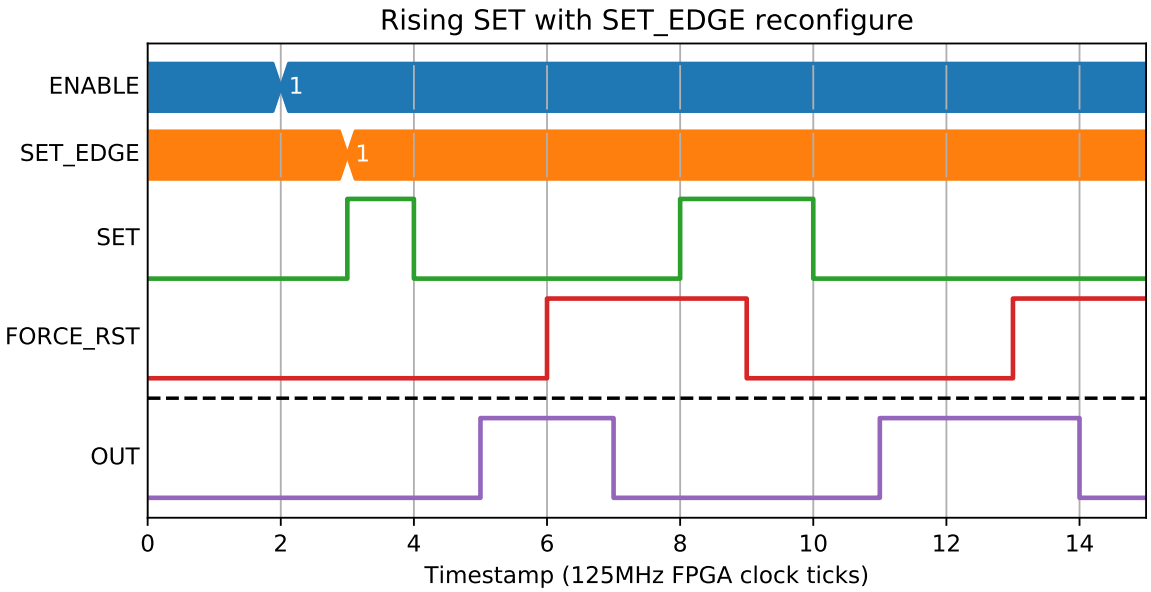

Output low while disabled
ENABLE
SET
OUT
0
2
4
6
8
10
12
Timestamp (125MHz FPGA clock ticks)


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



2.5.4 Active edge configure conditions

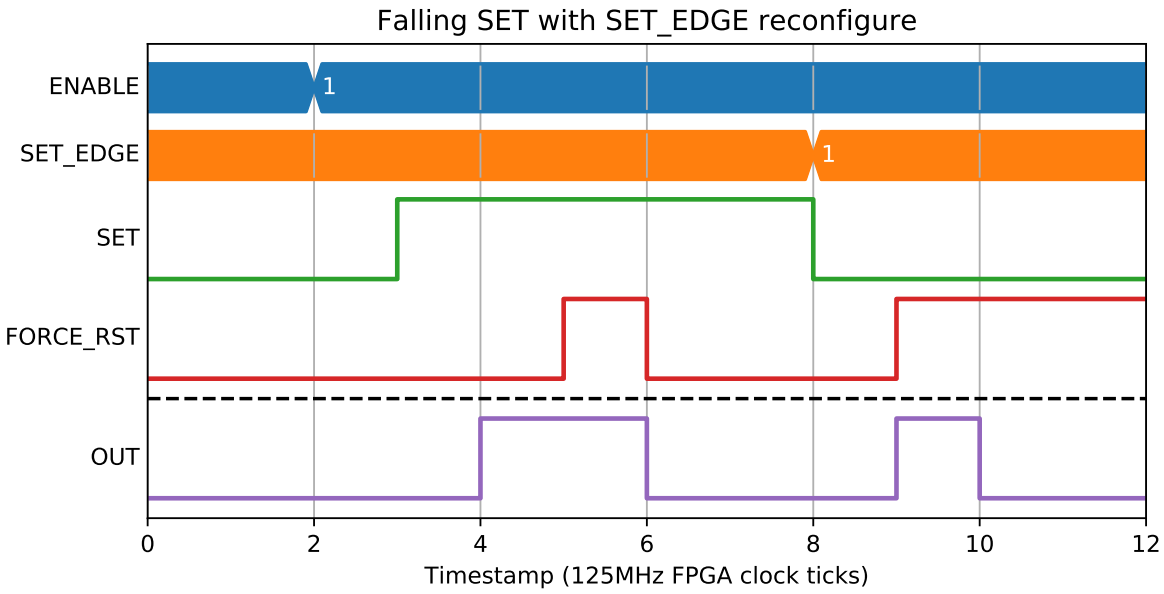
if the active edge is 'rising' then reset to 'falling' at the same time as a rising edge on the SET input, the block will ignore the rising edge and set the output OUT on the falling edge of the SET input.

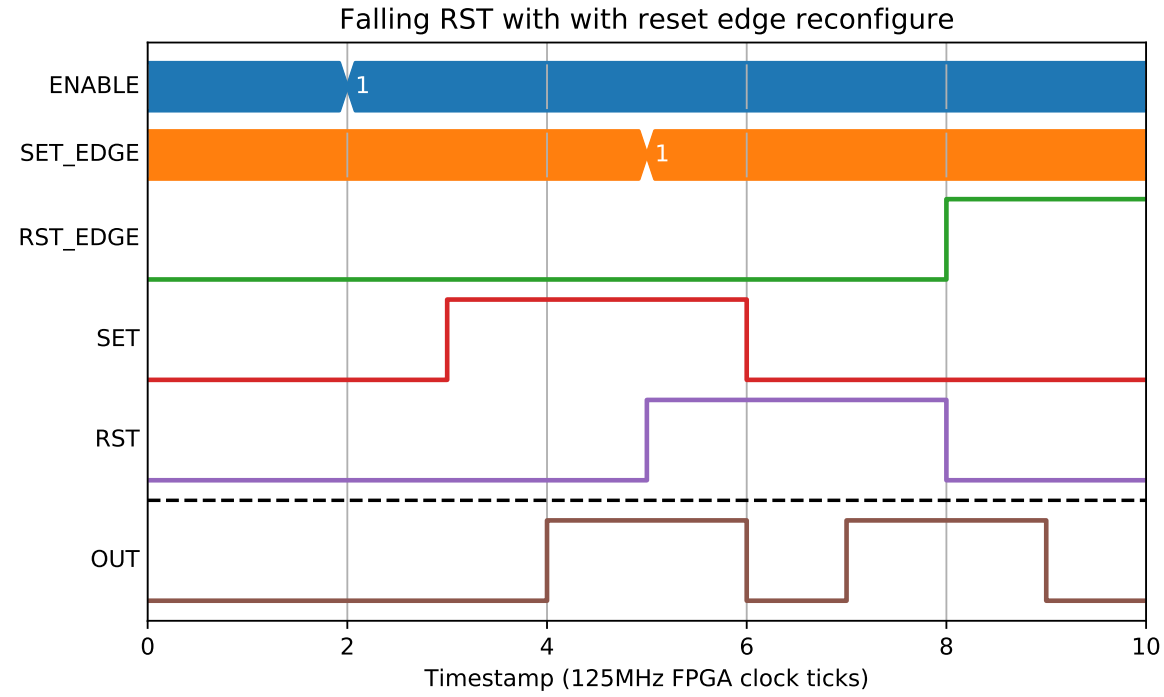


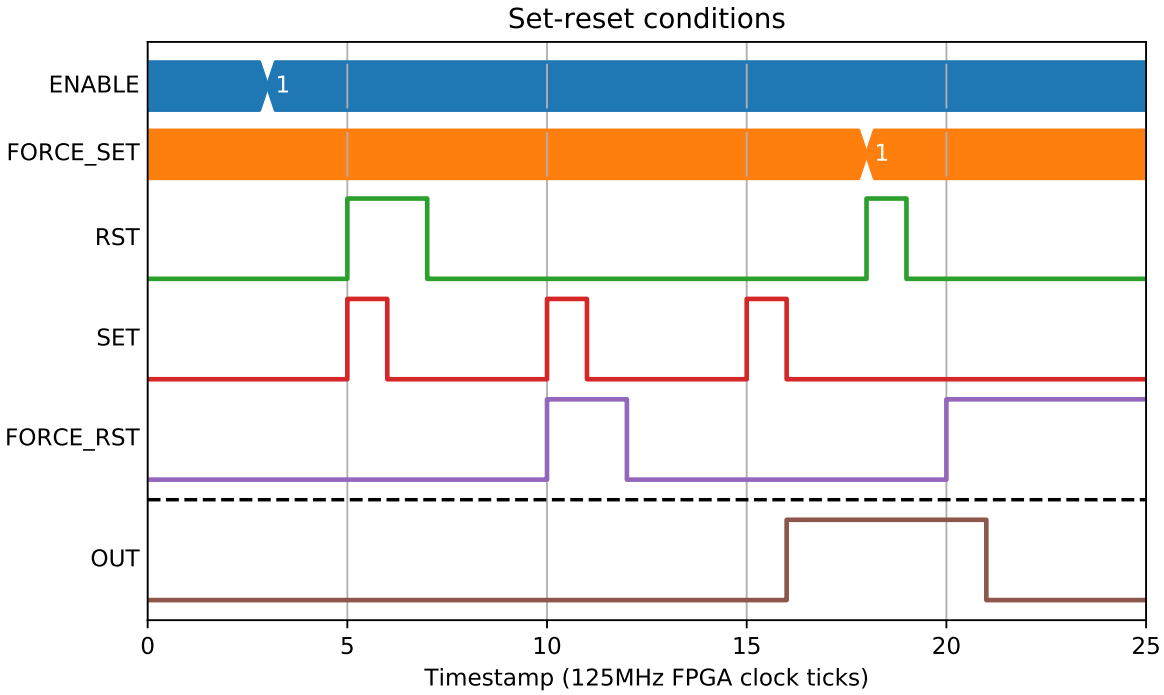
If the active edge changes to 'falling' at the same time as a falling edge on the SET input, the output OUT will be set following this.

2.5.5 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 SET with SET_EDGE reconfigure
ENABLE
SET_EDGE
SET
FORCE_RST
OUT
1
1
0
2
4
6
8
10
12
Timestamp (125MHz FPGA clock ticks)


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)



2.6 LUT - 5 Input lookup table [x8]

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

2.6.1 Parameters

Name	Dir	Type	Description
FUNC	R/W	UInt32	LUT logic function
A	R/W	Enum	Source of the value of A for calculation 0 - Input Value 1 - Rising Edge 2 - Falling Edge 3 - Either Edge
B	R/W	Enum	Source of the value of B for calculation
C	R/W	Enum	Source of the value of C for calculation
D	R/W	Enum	Source of the value of D for calculation
E	R/W	Enum	Source of the value of E for calculation
INPA	In	Bit	Input A
INPB	In	Bit	Input B
INPC	In	Bit	Input C
INPD	In	Bit	Input D
INPE	In	Bit	Input E
OUT	Out	Bit	Output port from the block

2.6.2 Testing Function Output

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

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

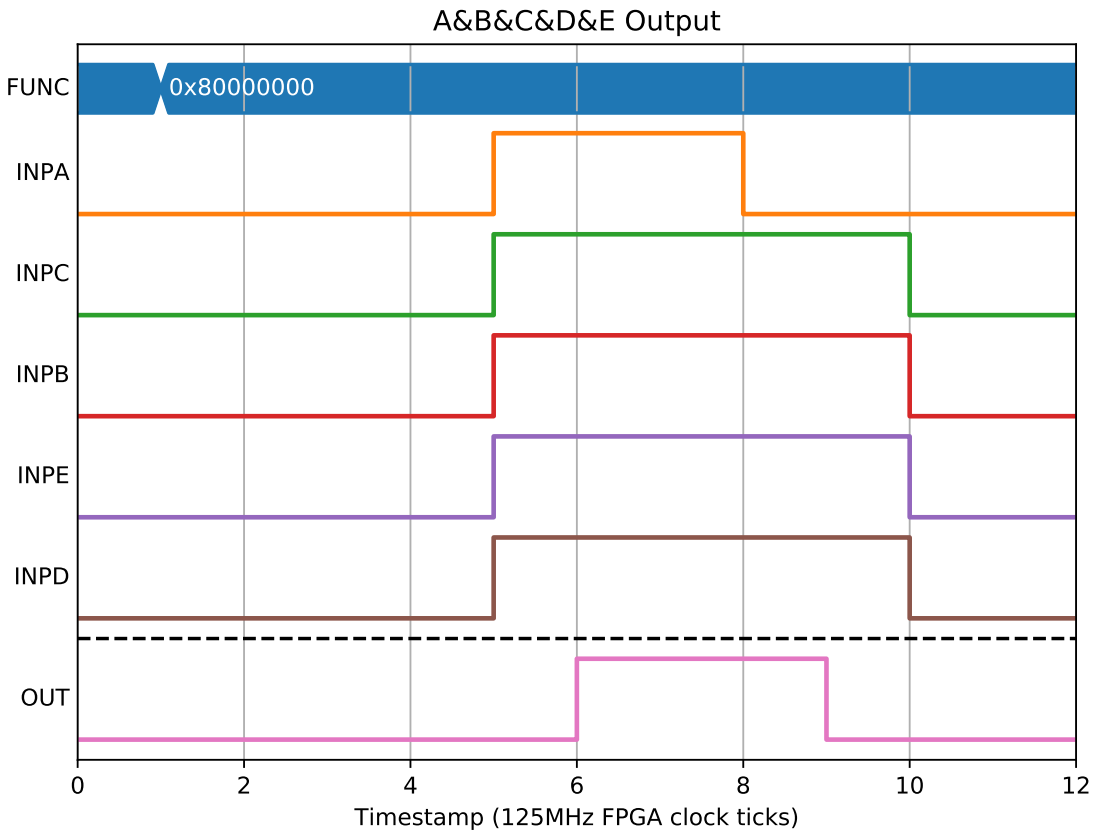
A (FUNC= 0xffff0000). The output should only be 1 if A is 1 irrespective of any other input.

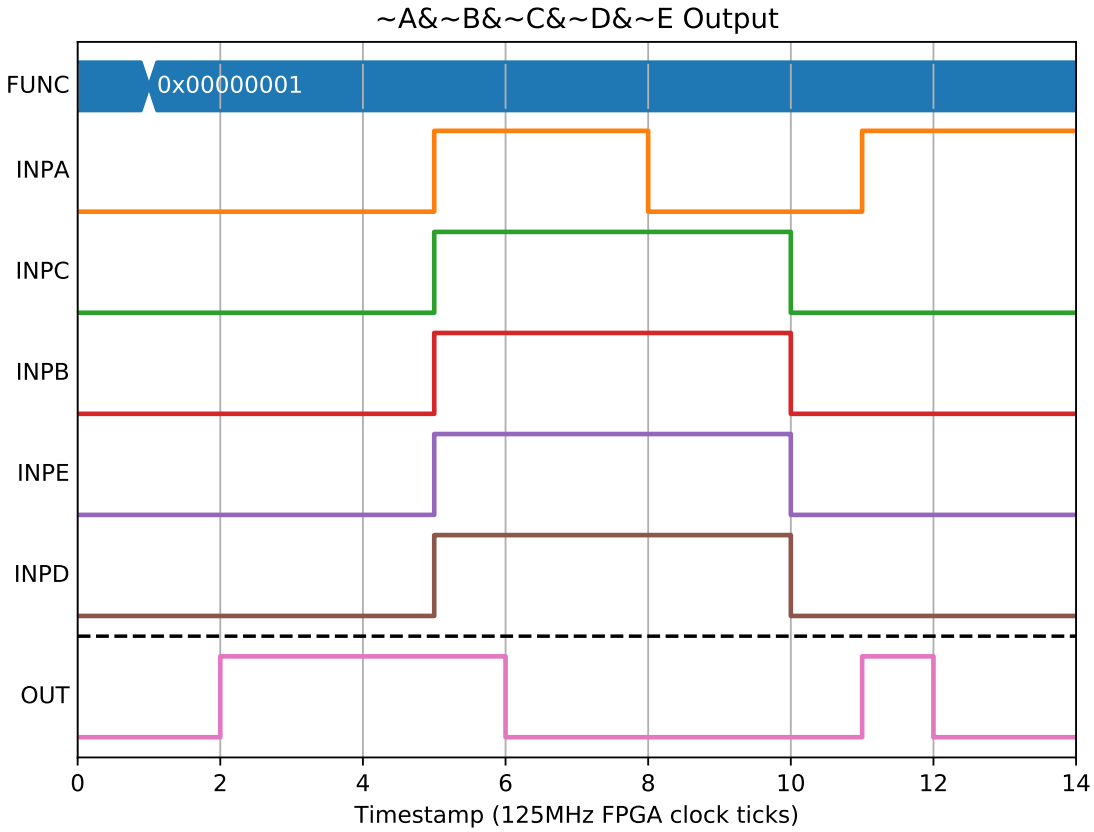
A&B|C&~D (FUNC= 0xff303030)

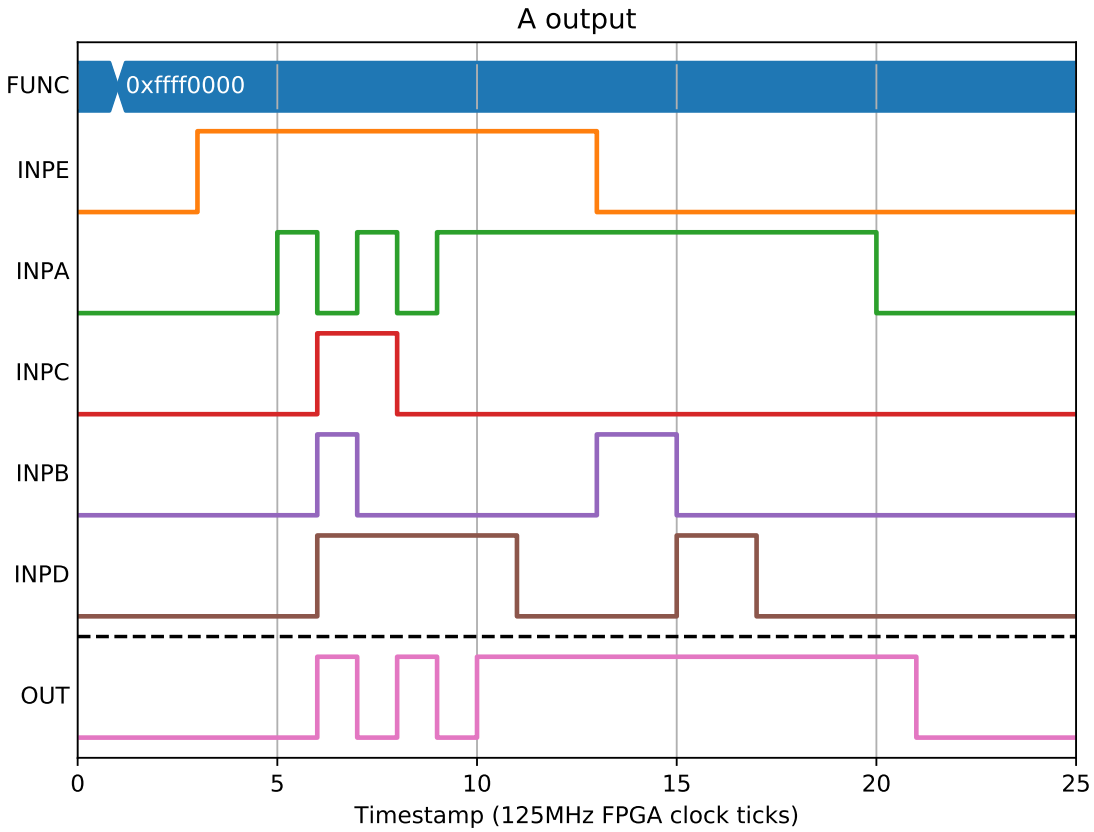
A?(B):D&E (FUNC= 0xff008888)

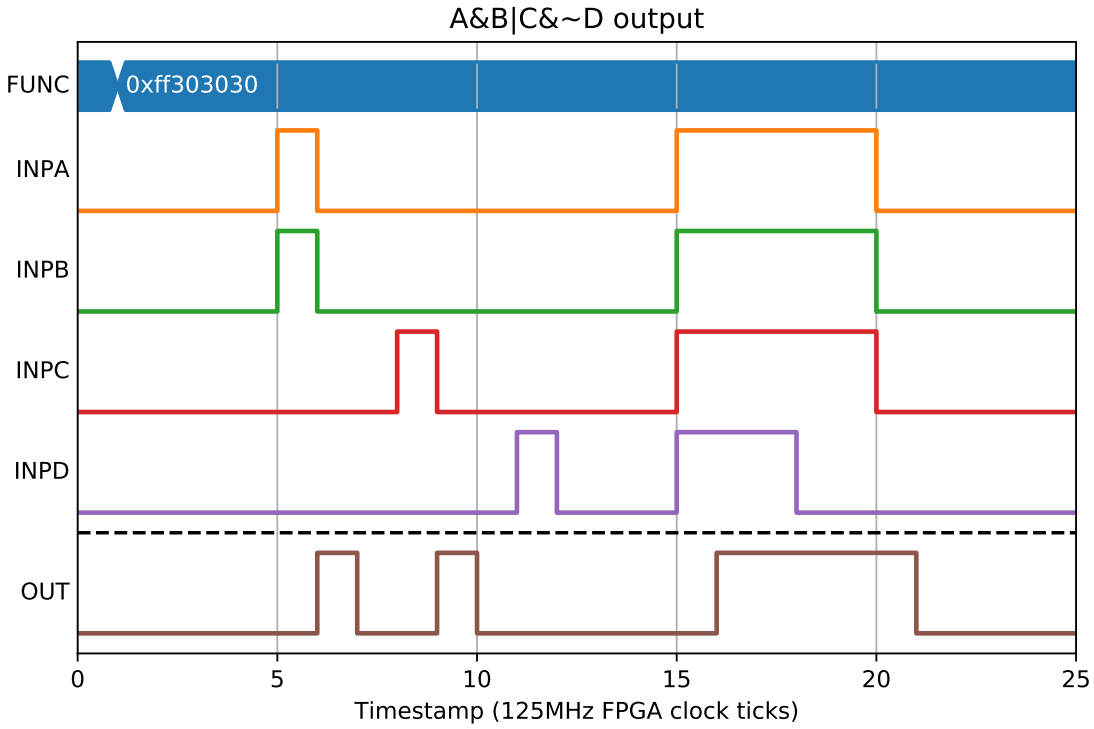
2.6.3 Changing the function in a test

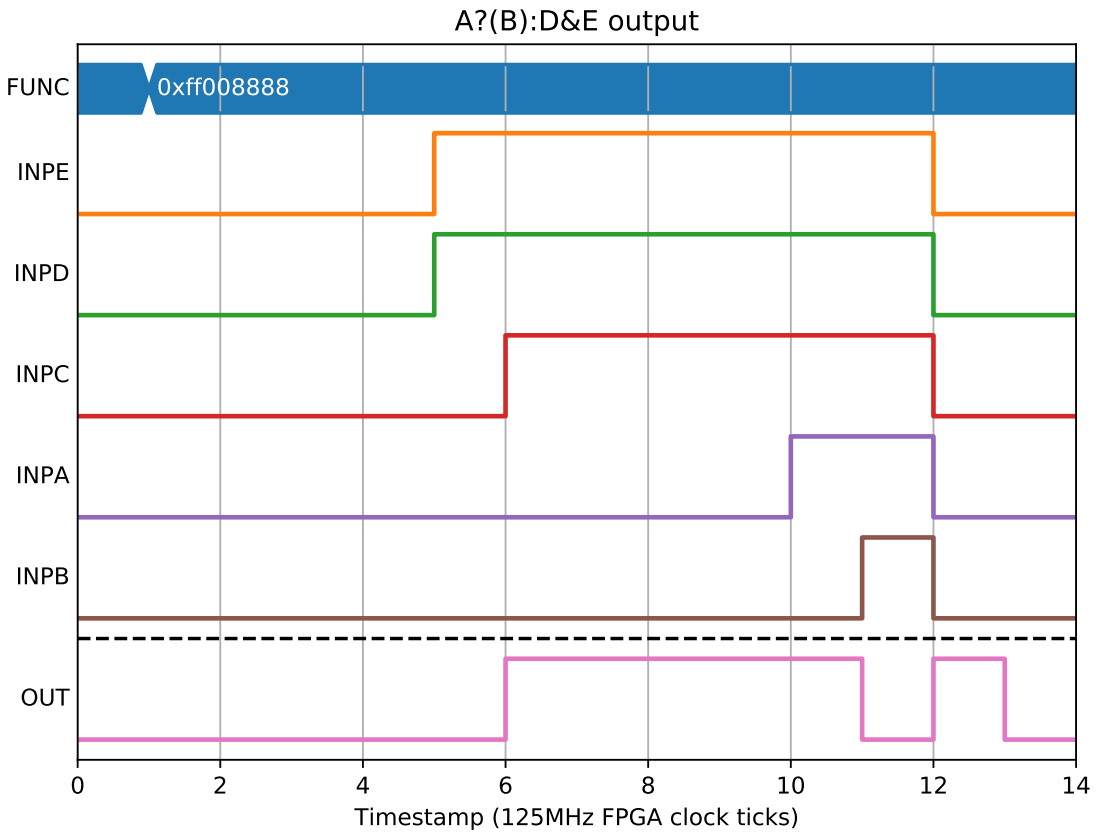
If a function is changed, the output will take effect on the next clock tick

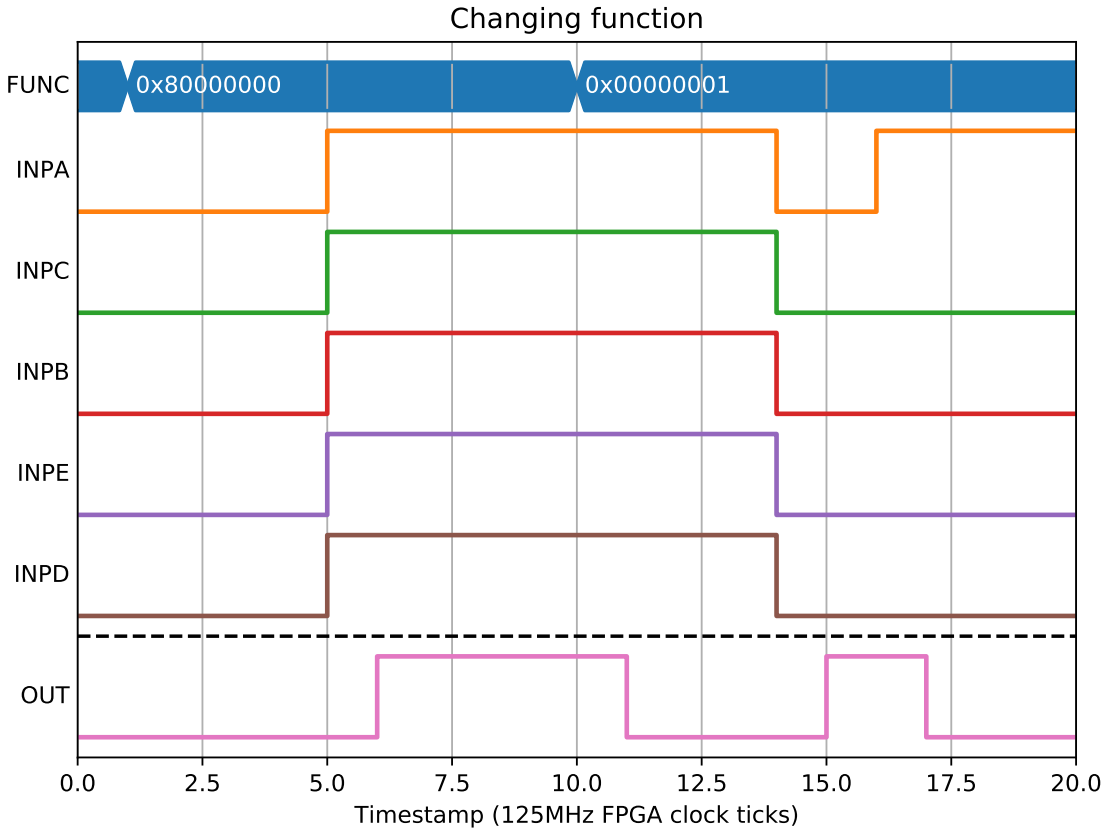

A&B&C&D&E Output
FUNC
0x80000000
INPA
INPC
INPB
INPE
INPD
OUT
0
2
4
6
8
10
12
Timestamp (125MHz FPGA clock ticks)


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


A output
FUNC
0xffff0000
INPE
INPA
INPC
INPB
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)

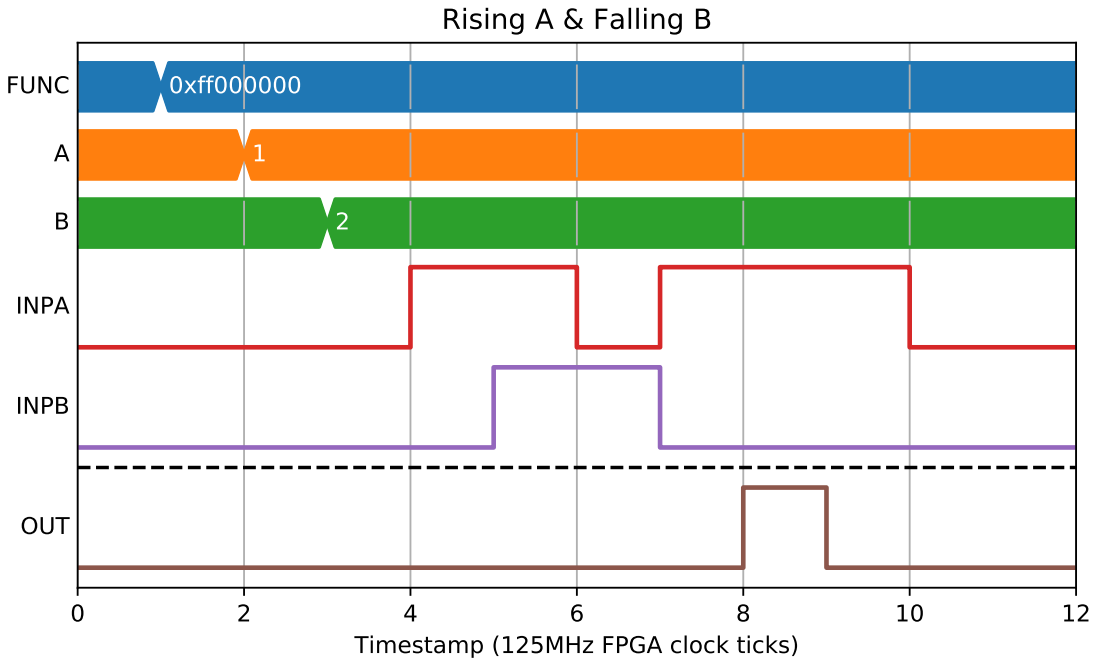

A?(B):D&E output
FUNC
0xff008888
INPE
INPD
INPC
INPA
INPB
OUT
0
2
4
6
8
10
12
14
Timestamp (125MHz FPGA clock ticks)


Changing function
FUNC
0x80000000
0x00000001
INPA
INPC
INPB
INPE
INPD
OUT
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Timestamp (125MHz FPGA clock ticks)

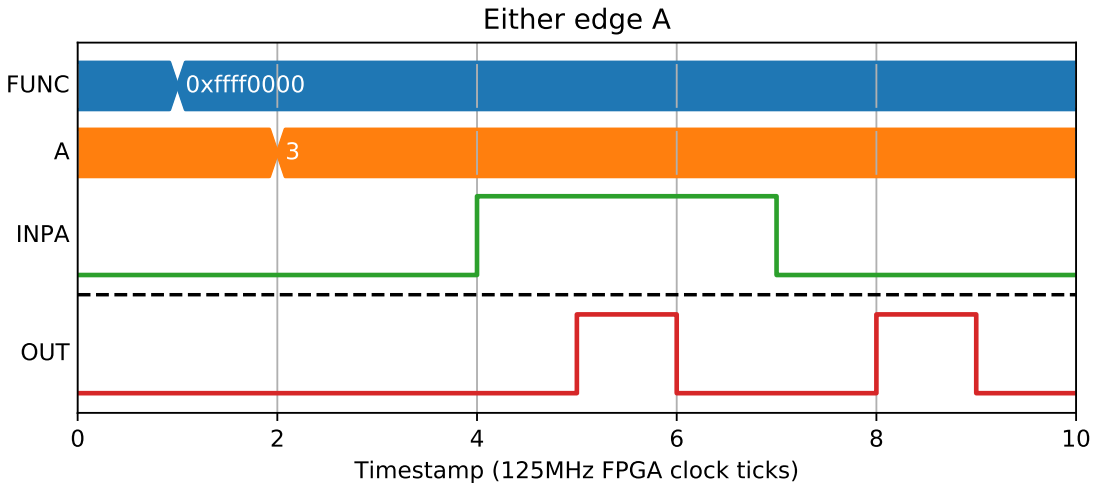
2.6.4 Edge triggered inputs

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.

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=1 (falling edge of INPB):



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



2.7 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.

2.7.1 Parameters

Name	Dir	Type	Description
TABLE	W	Table	Sequencer table of lines
PRESCALE	W	UInt32	Prescalar for system clock
ENABLE	In	Bit	Stop on falling edge, reset and enable on rising edge
INPA	In	Bit	BITA for optional trigger condition
INPB	In	Bit	BITB for optional trigger condition
INPC	In	Bit	BITC for optional trigger condition
POSA	In	Pos	POSA for optional trigger condition
POSB	In	Pos	POSB for optional trigger condition
POSC	In	Pos	POSC for optional trigger condition
ACTIVE	Out	Bit	Sequencer Active Flag
OUTA	Out	Bit	Output A for phase outputs
OUTB	Out	Bit	Output B for phase outputs
OUTC	Out	Bit	Output C for phase outputs
OUTD	Out	Bit	Output D for phase outputs
OUTE	Out	Bit	Output E for phase outputs
OUTF	Out	Bit	Output F for phase outputs
TABLE_REPEAT	R	UInt32	Current iteration through the entire table
TABLE_LINE	R	UInt32	Current line in the table that is active
LINE_REPEAT	R	UInt32	Current iteration of the active table line
STATE	R	Enum	Internal statemachine state 0: WAIT_ENABLE 1: LOAD_TABLE 2: WAIT_TRIGGER 3: PHASE1 4: PHASE2

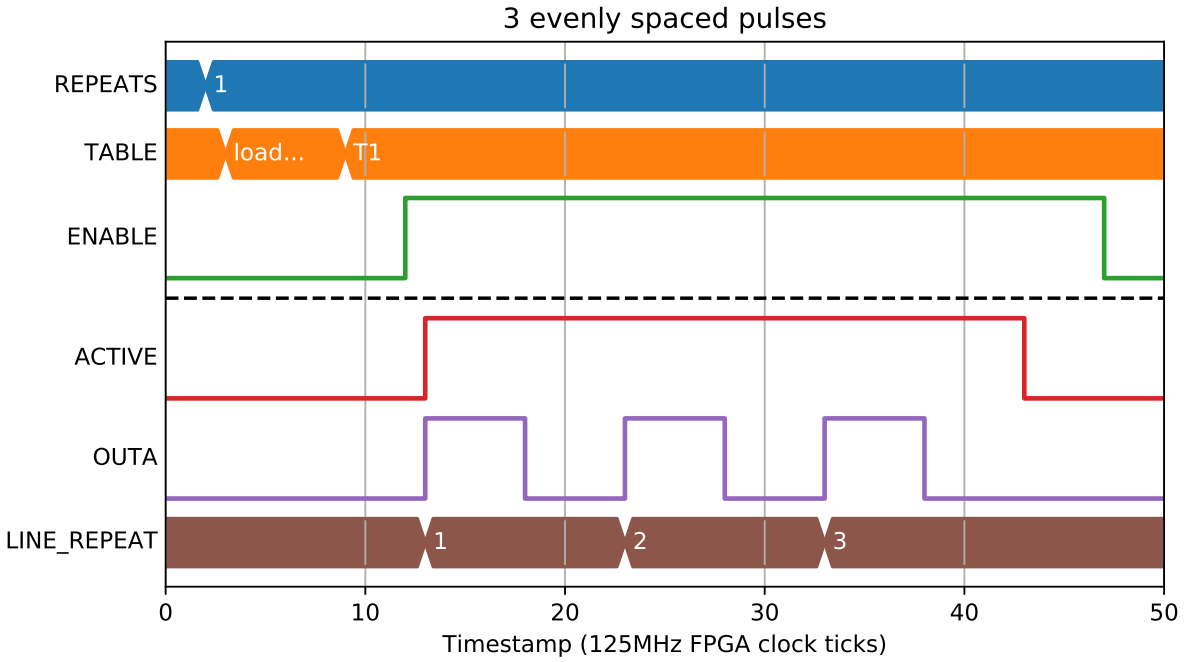
2.7.2 Sequencer Table Line Composition

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

2.7.3 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:

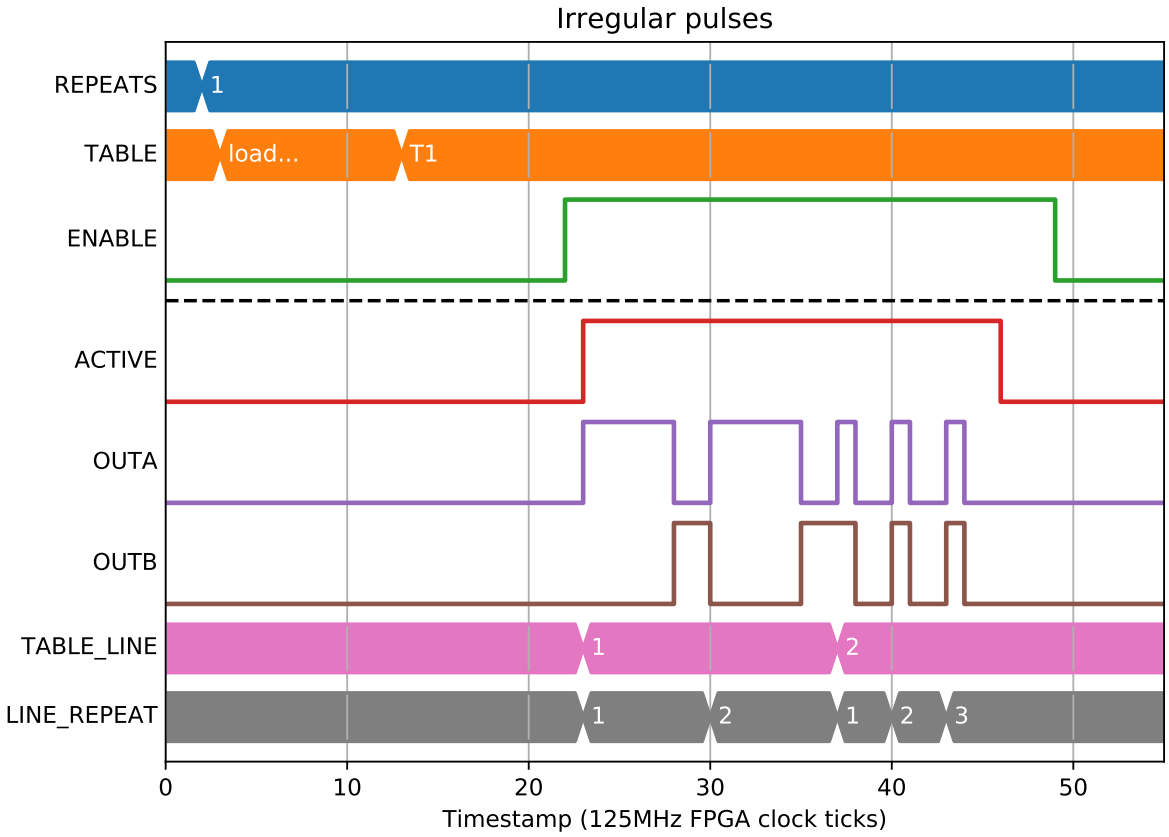
T1																
#	Trigger		Phase1	Phase1 Outputs						Phase2	Phase2 Outputs					
Repeats	Condition	Position	Time	A	B	C	D	E	F	Time	A	B	C	D	E	F
3	Immediate	0	5	1	0	0	0	0	0	5	0	0	0	0	0	0



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:

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

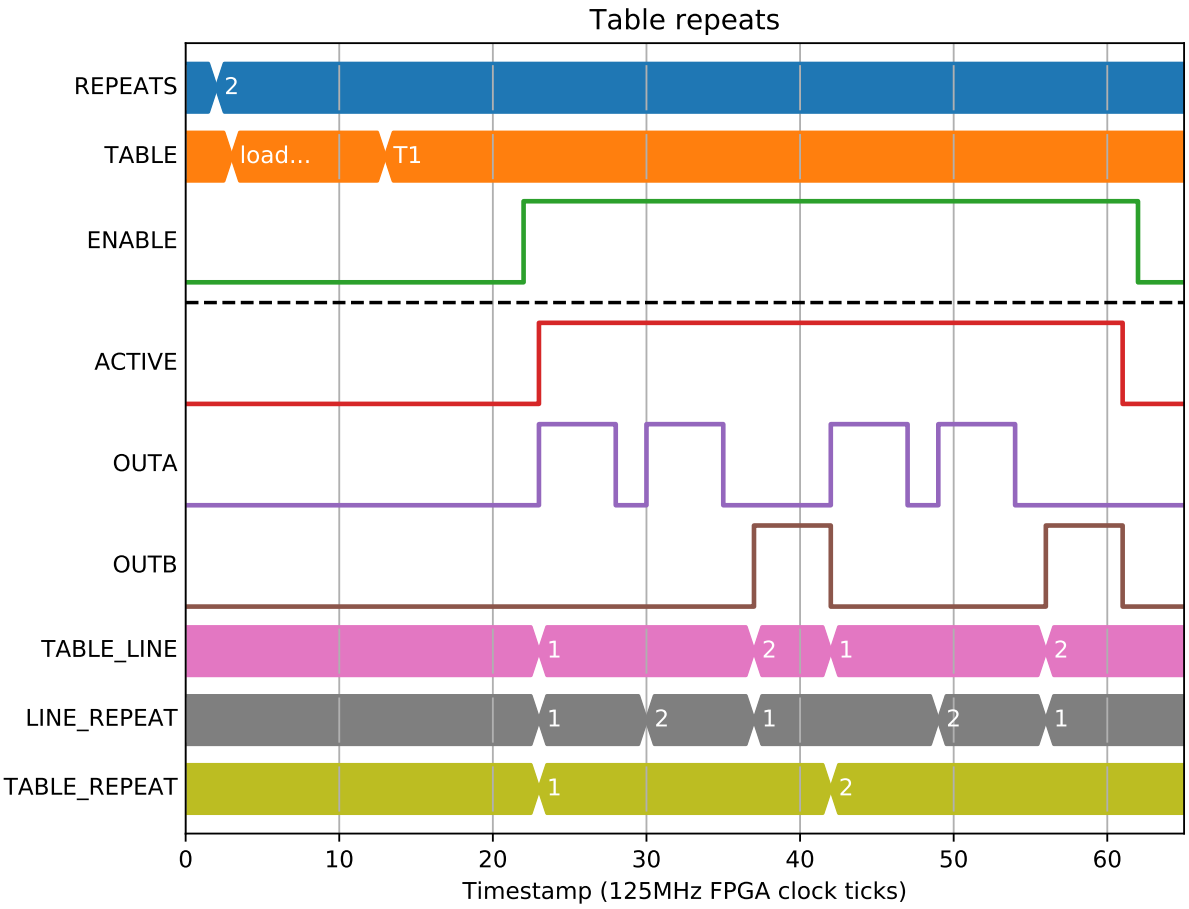
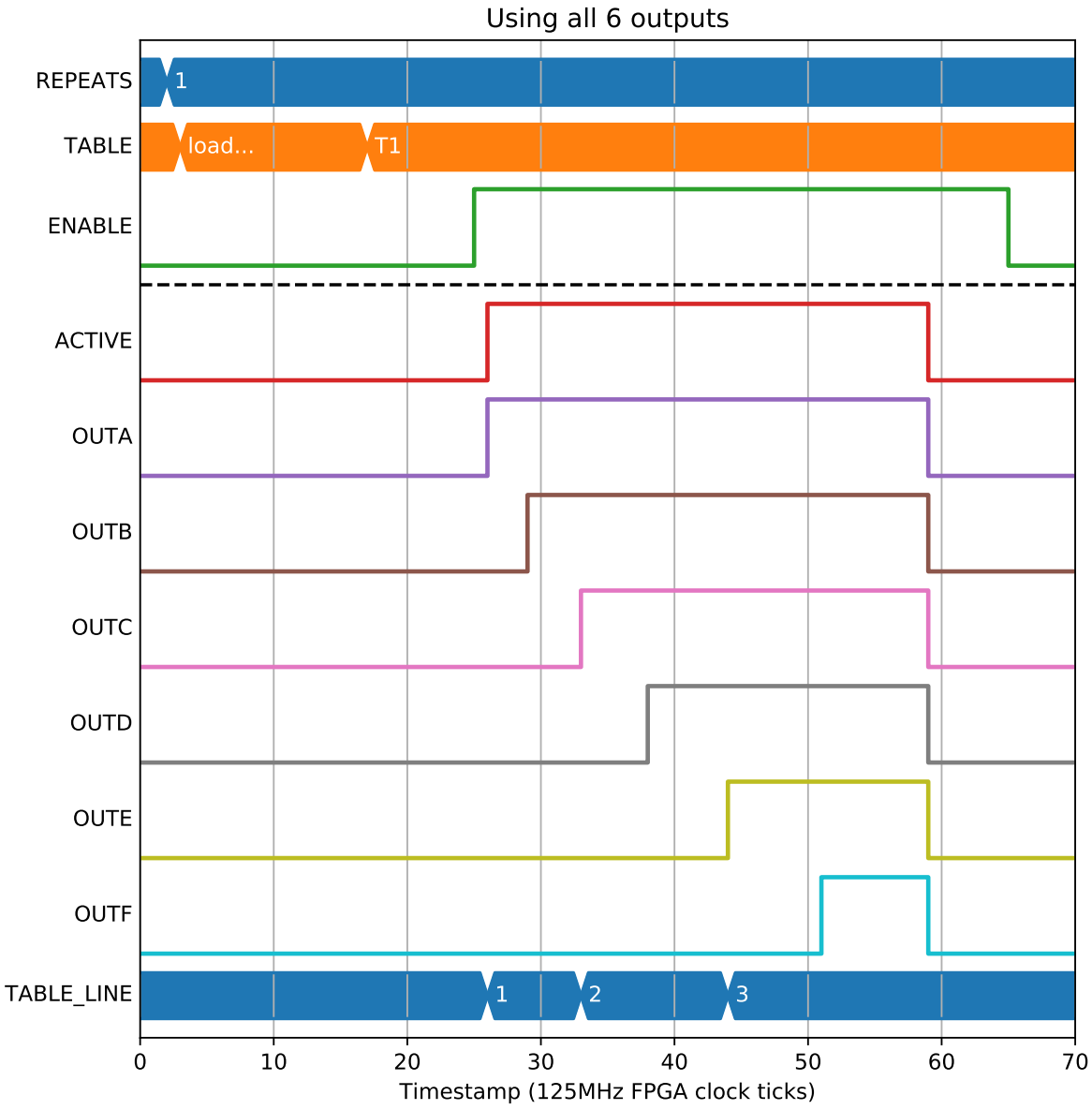
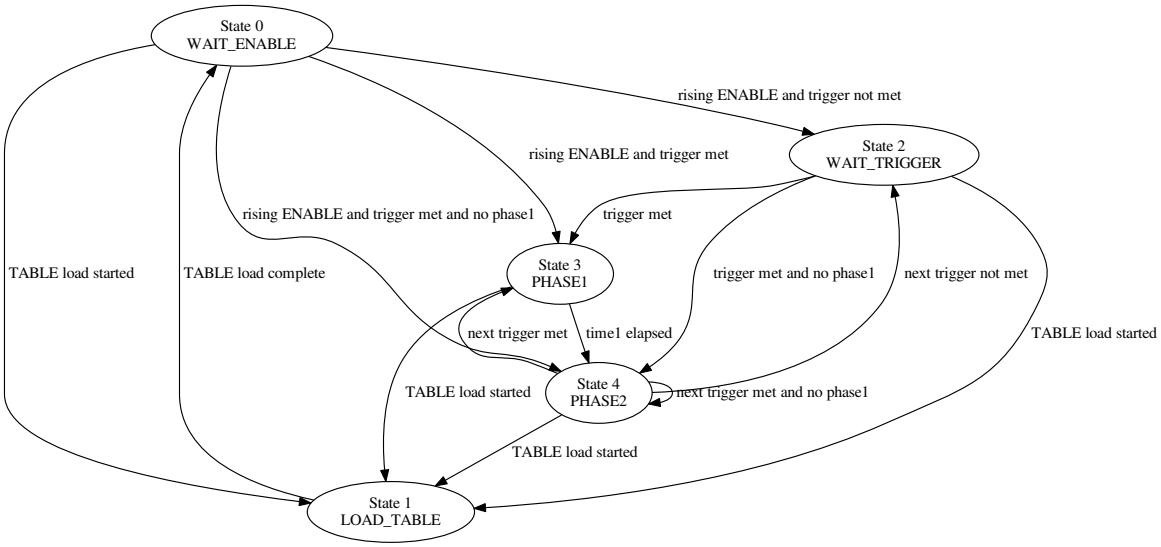


Table repeats
REPEATS
2
TABLE
load...
T1
ENABLE
ACTIVE
OUTA
OUTB
TABLE_LINE
1
2
1
2
LINE_REPEAT
1
2
1
2
1
TABLE_REPEAT
1
2
0
10
20
30
40
50
60
Timestamp (125MHz FPGA clock ticks)


Using all 6 outputs
REPEATS
1
TABLE
load...
T1
ENABLE
ACTIVE
OUTA
OUTB
OUTC
OUTD
OUTE
OUTF
TABLE_LINE
1
2
3
0
10
20
30
40
50
60
70
Timestamp (125MHz FPGA clock ticks)

2.7.4 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.



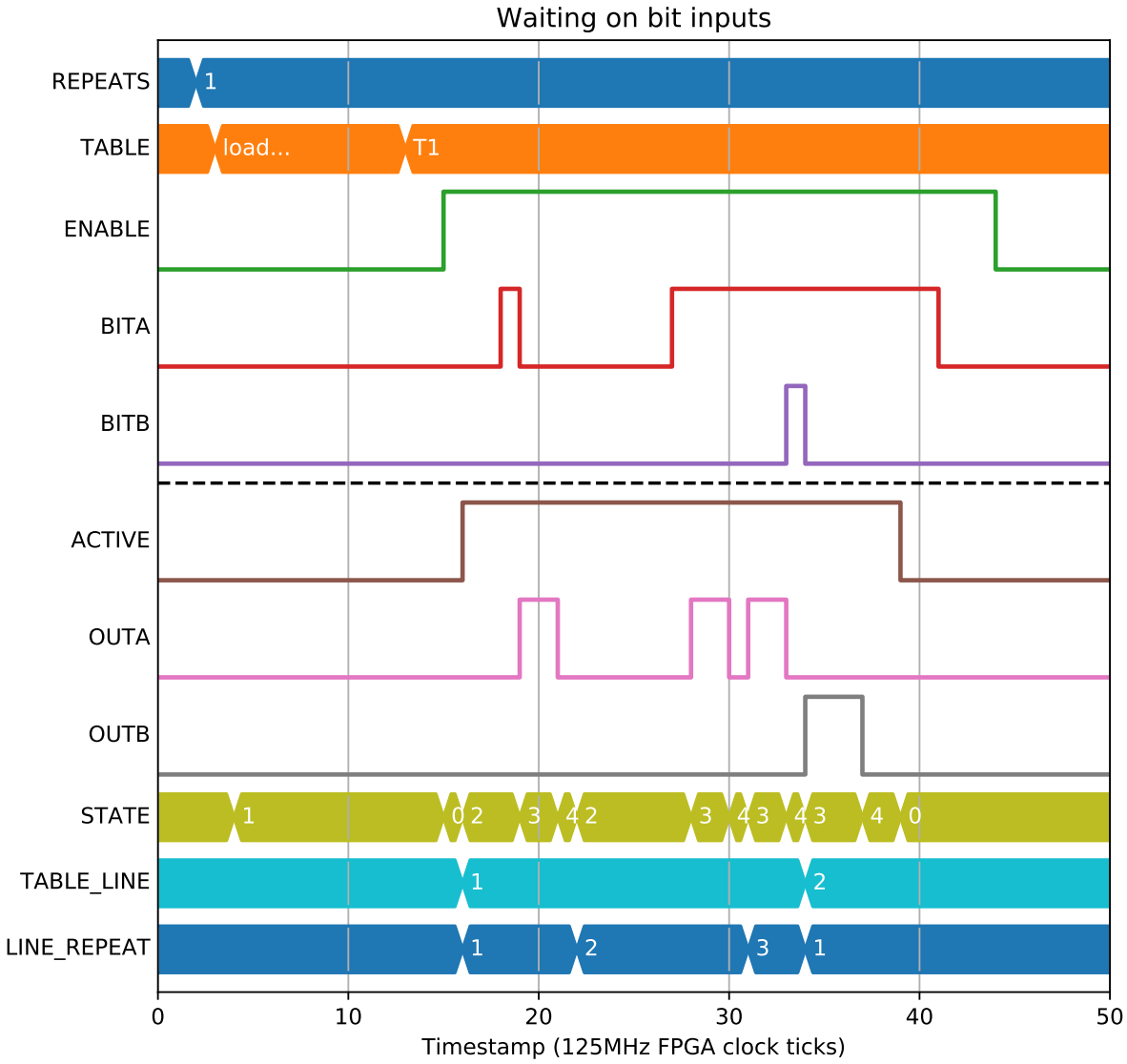
2.7.5 External trigger sources

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

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


Waiting on bit inputs
REPEATS
TABLE
ENABLE
BITA
BITB
ACTIVE
OUTA
OUTB
STATE
TABLE_LINE
LINE_REPEAT
1
load...
T1
1 0 2 3 4 2 3 4 3 4 3 4 0
1 2
1 2 3 1
0
10
20
30
40
50
Timestamp (125MHz FPGA clock ticks)

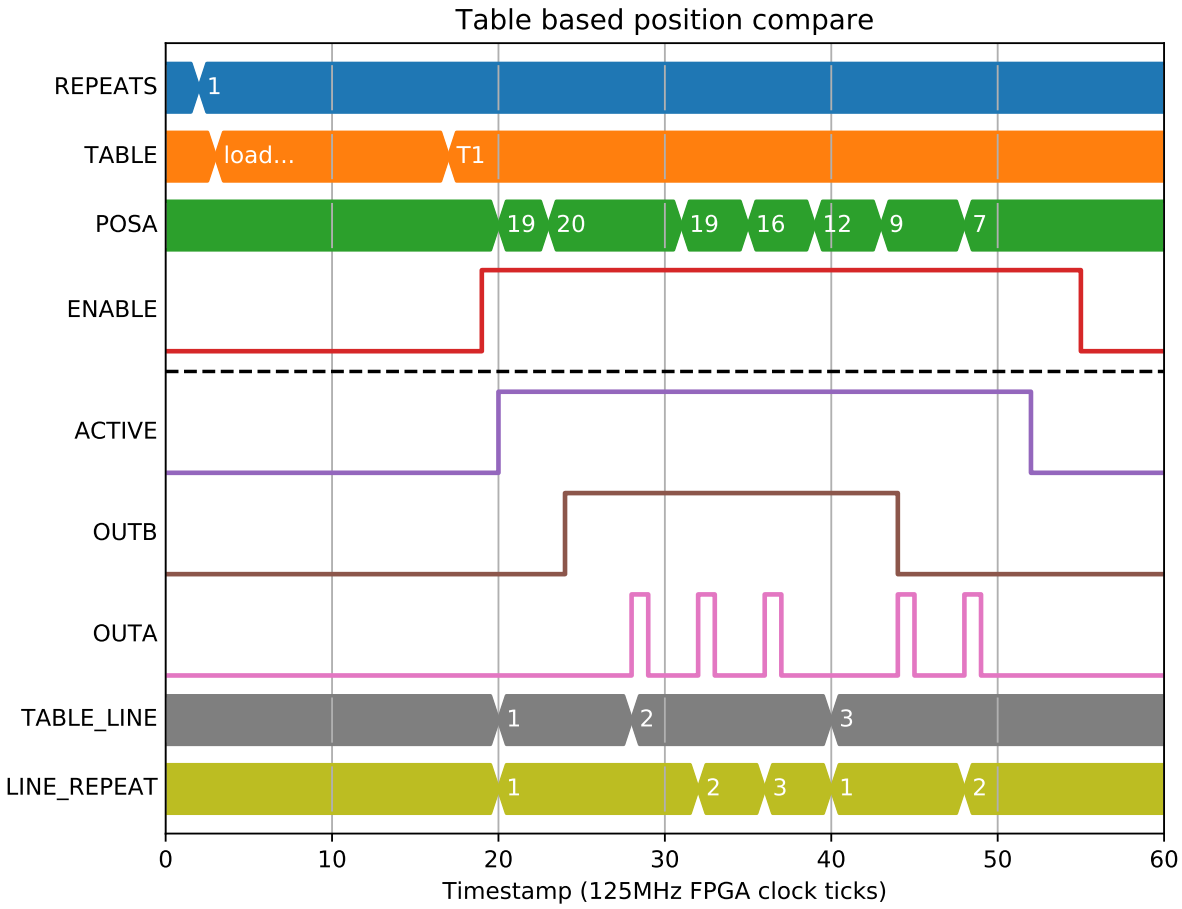
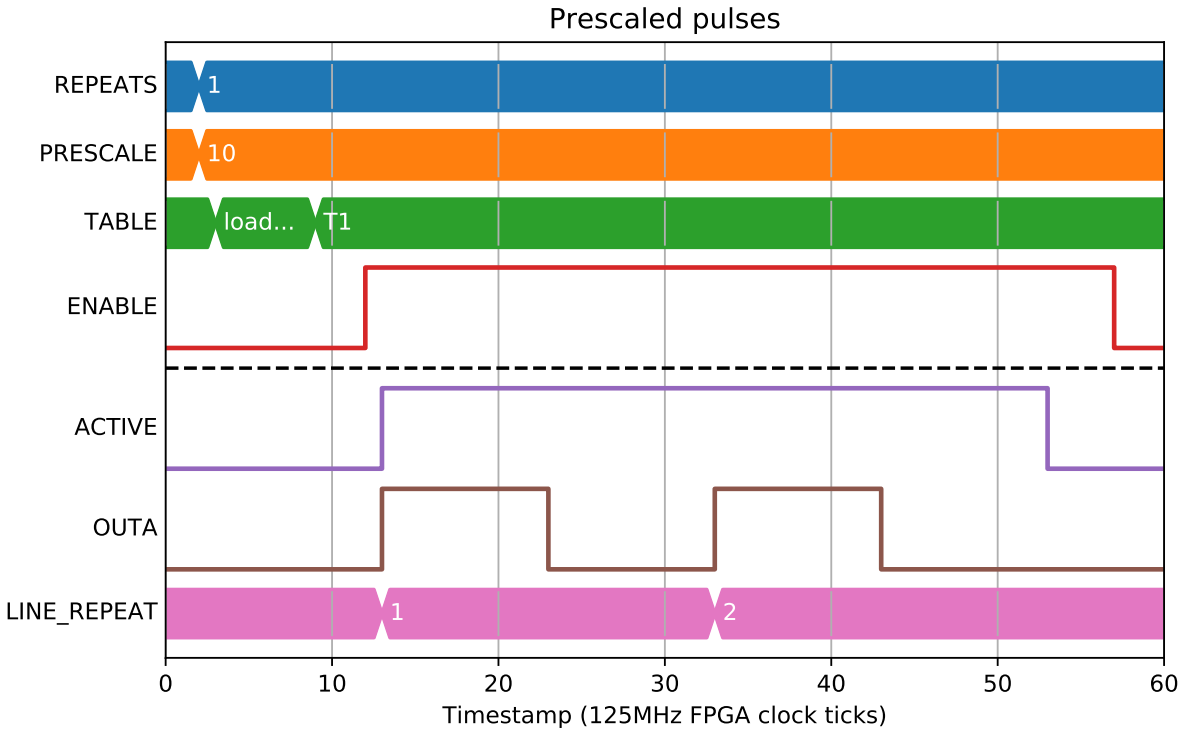


Table based position compare
REPEATS
TABLE
POSA
ENABLE
ACTIVE
OUTB
OUTA
TABLE_LINE
LINE_REPEAT
1
load...
T1
19
20
19
16
12
9
7
1
2
3
1
2
3
0
10
20
30
40
50
60
Timestamp (125MHz FPGA clock ticks)

2.7.6 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:

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

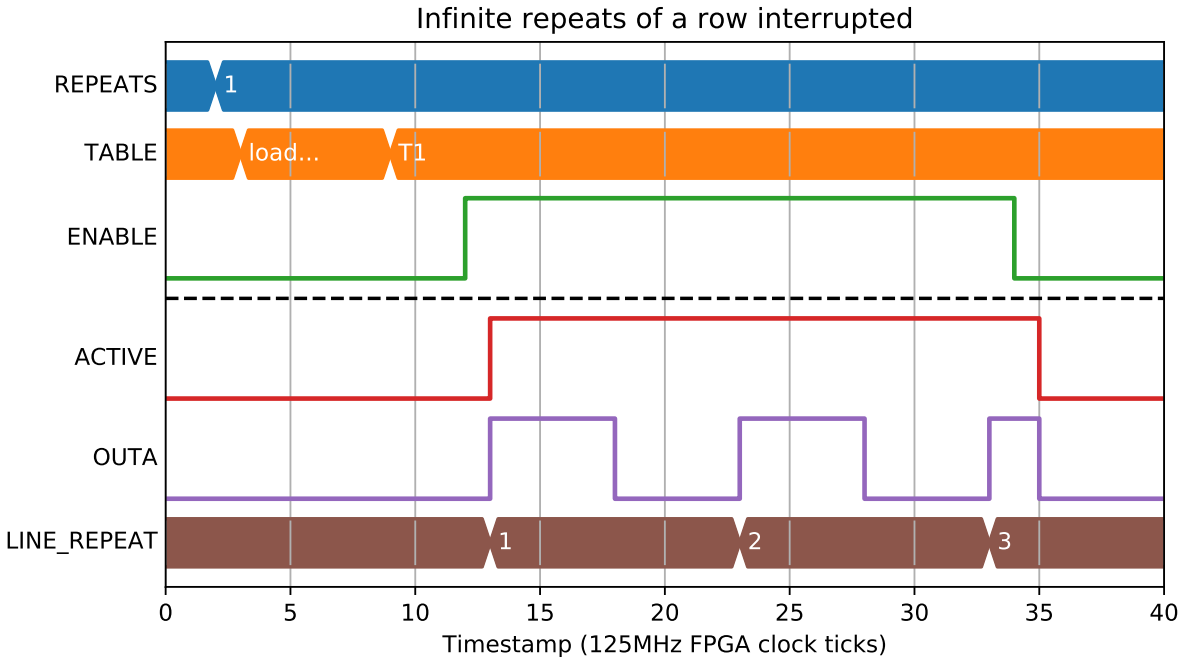


2.7.7 Interrupting a sequence

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:



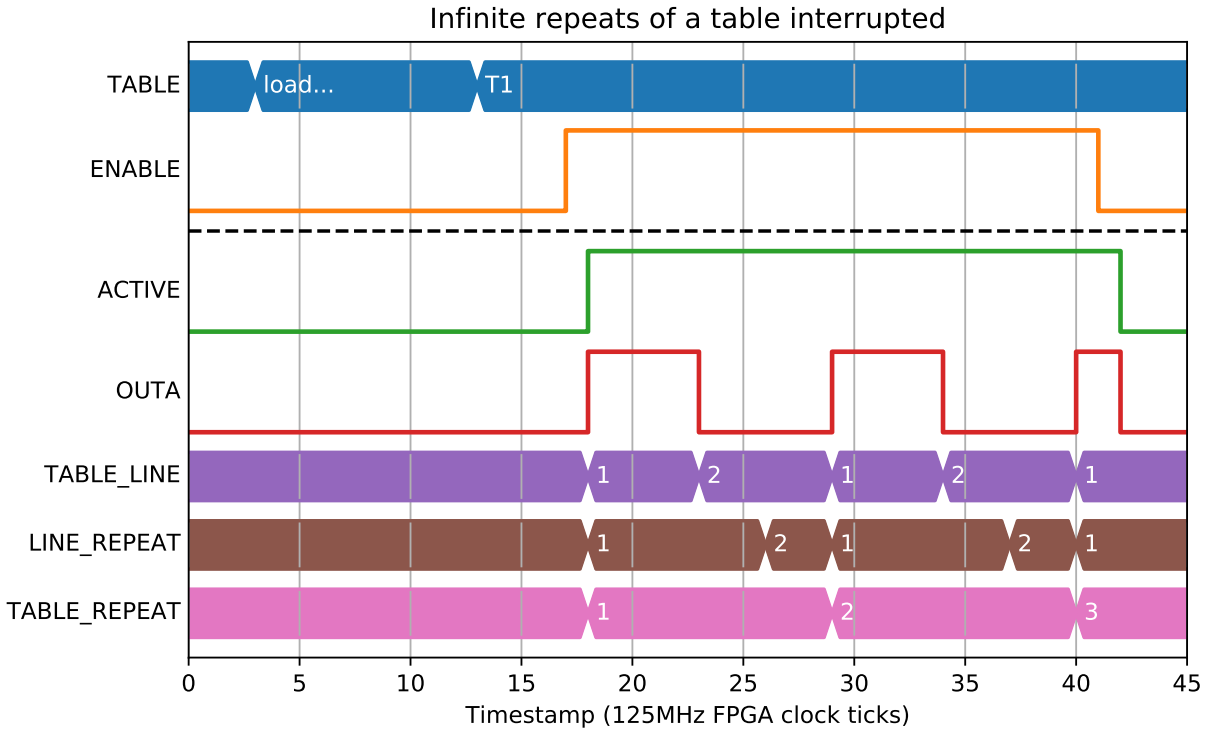
T1																
#	Trigger		Phase1	Phase1 Outputs						Phase2	Phase2 Outputs					
Repeats	Condition	Position	Time	A	B	C	D	E	F	Time	A	B	C	D	E	F
1	Immediate	0	0	0	0	0	0	0	0	5	1	0	0	0	0	0
2	Immediate	0	0	0	0	0	0	0	0	3	0	0	0	0	0	0

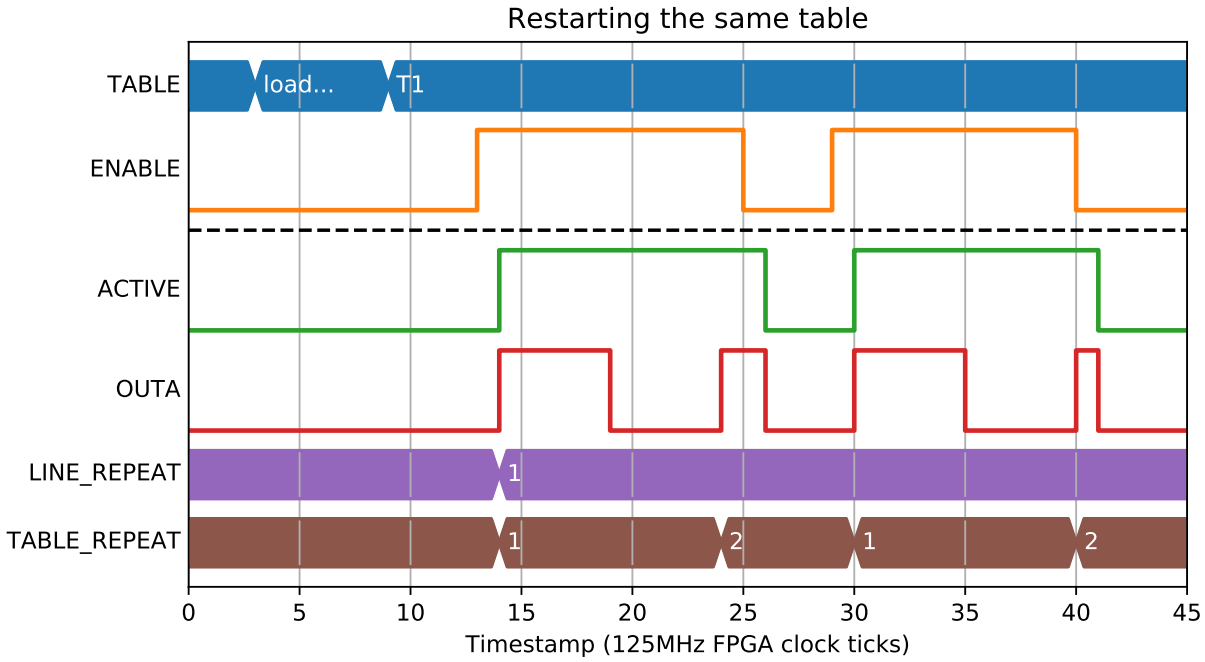
And a rising edge of the ENABLE will re-run the same table from the start:

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

2.7.8 Table rewriting

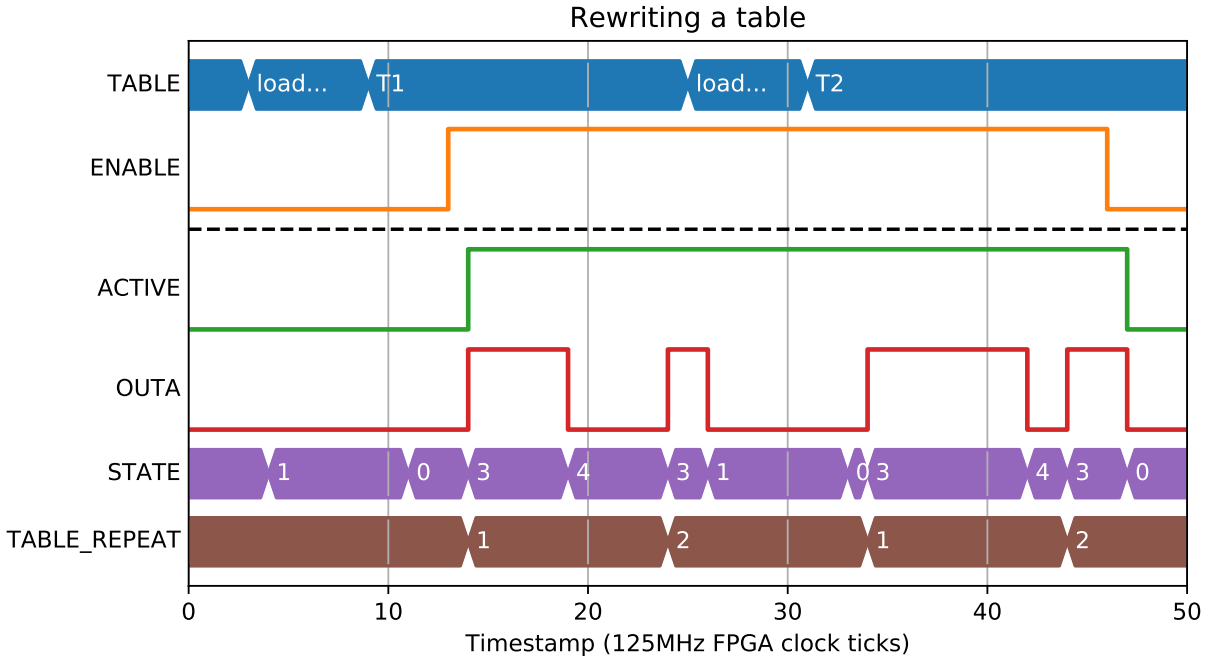
If a table is written while enabled, the outputs and table state are reset and operation begins again from the first repeat of the first line of the table:


Infinite repeats of a table interrupted
TABLE
load...
T1
ENABLE
ACTIVE
OUTA
TABLE_LINE
1
2
1
2
1
LINE_REPEAT
1
2
1
2
1
TABLE_REPEAT
1
2
3
0
5
10
15
20
25
30
35
40
45
Timestamp (125MHz FPGA clock ticks)


Restarting the same table
TABLE
load...
T1
ENABLE
ACTIVE
OUTA
LINE_REPEAT
1
TABLE_REPEAT
1
2
1
2
0
5
10
15
20
25
30
35
40
45
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



2.8 COUNTER [x8]

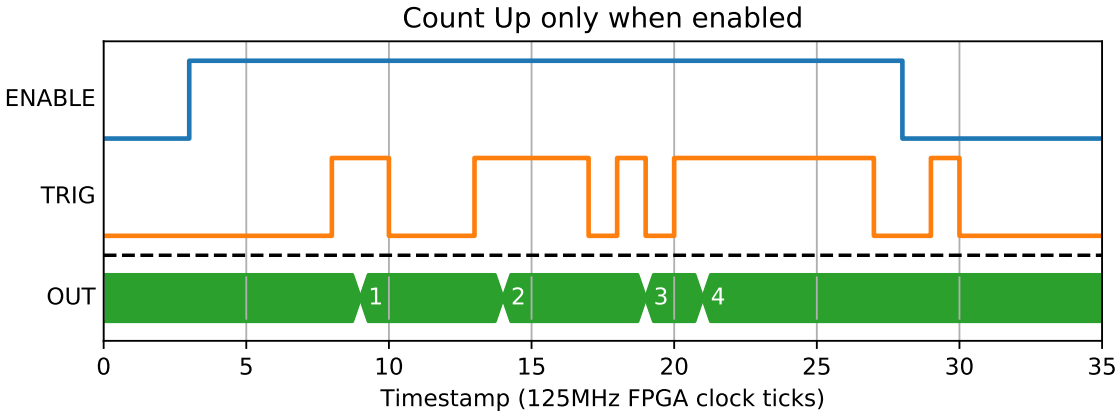
Each counter block, when enabled, can count up/down with user-defined step value on the rising edge on input trigger. The counters can also be initialised to a user-defined START value.

2.8.1 Parameters

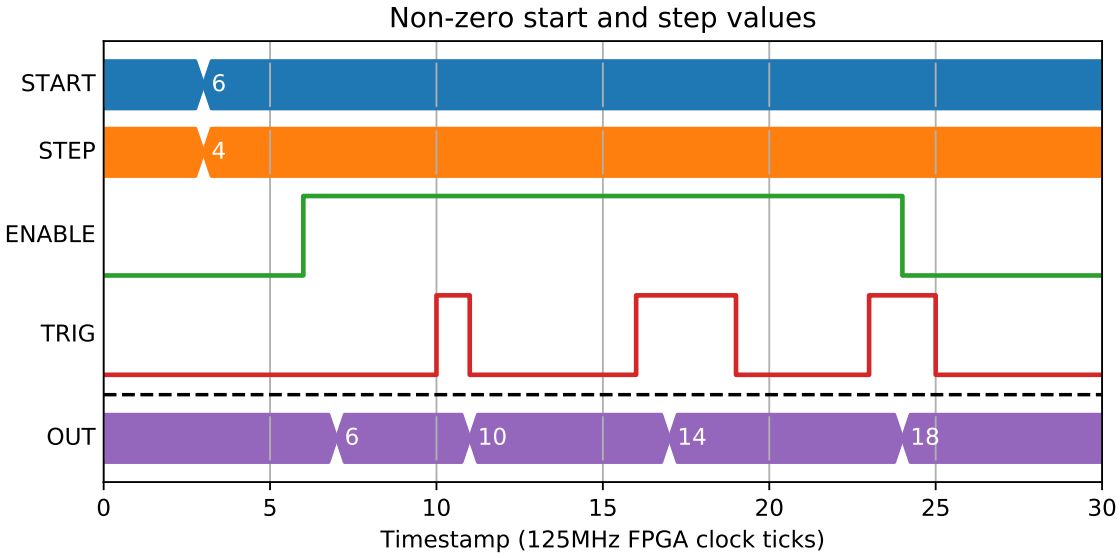
Name	Dir	Type	Description
START	W	UInt32	Counter start value
STEP	W	UInt32	Up/Down step value
ENABLE	In	Bit	Halt on falling edge, reset and enable on rising
TRIG	In	Bit	Rising edge ticks the counter up/down by STEP
DIR	In	Bit	Up/Down direction (0 = Up, 1 = Down)
CARRY	Out	Bit	Internal counter overflow status
OUT	Out	Pos	Current counter value

2.8.2 Counting pulses

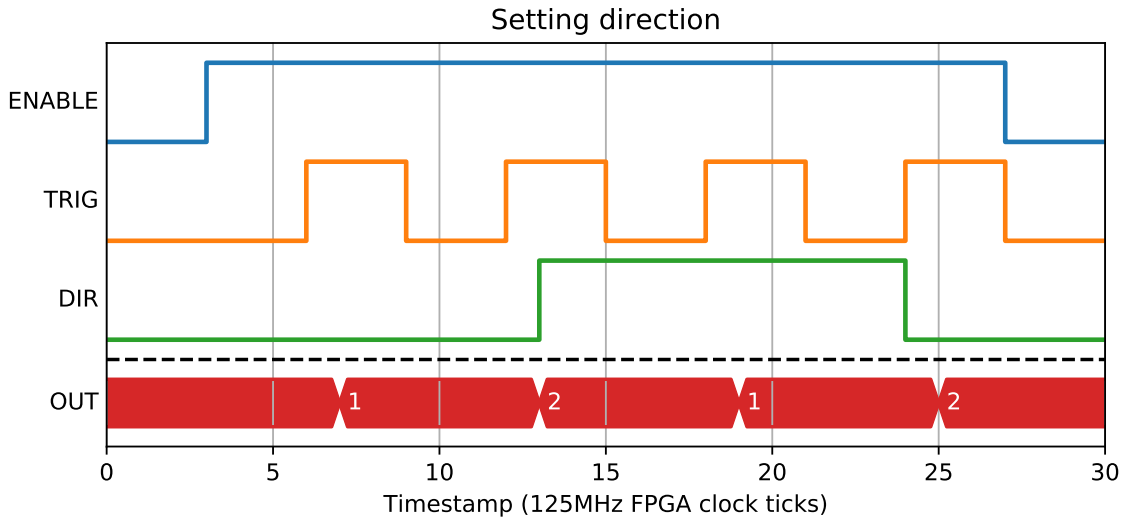
The most common use of a counter block is when you would like to track the number of rising edges received while enabled:



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

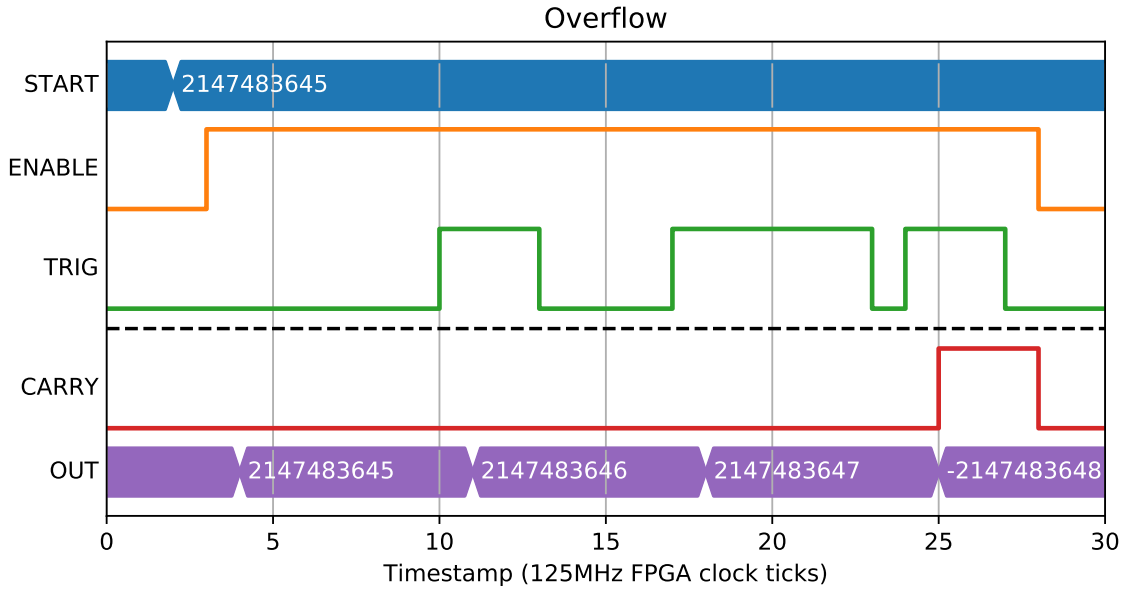


You can also set the direction that a pulse should apply step, so it becomes an up/down counter. The direction is sampled on the same clock tick as the pulse rising edge:



2.8.3 Rollover

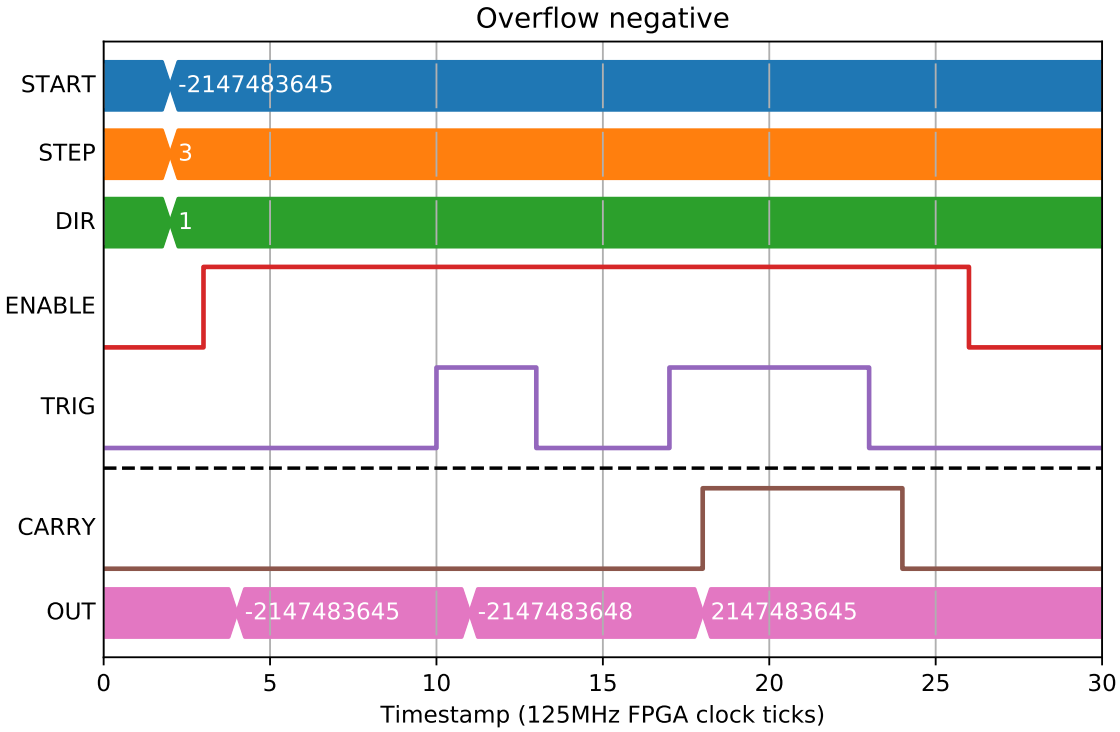
If the count goes higher than the max value for an int32 (2147483647) the CARRY output gets set high and the counter rolls. The CARRY output stays high for as long as the trigger input stays high.

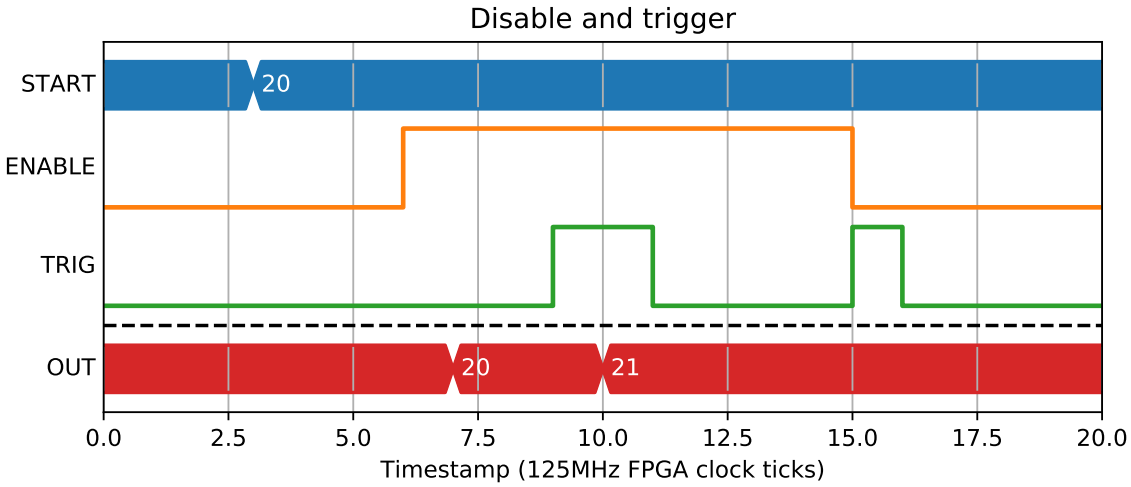


A similar thing happens for a negative overflow:

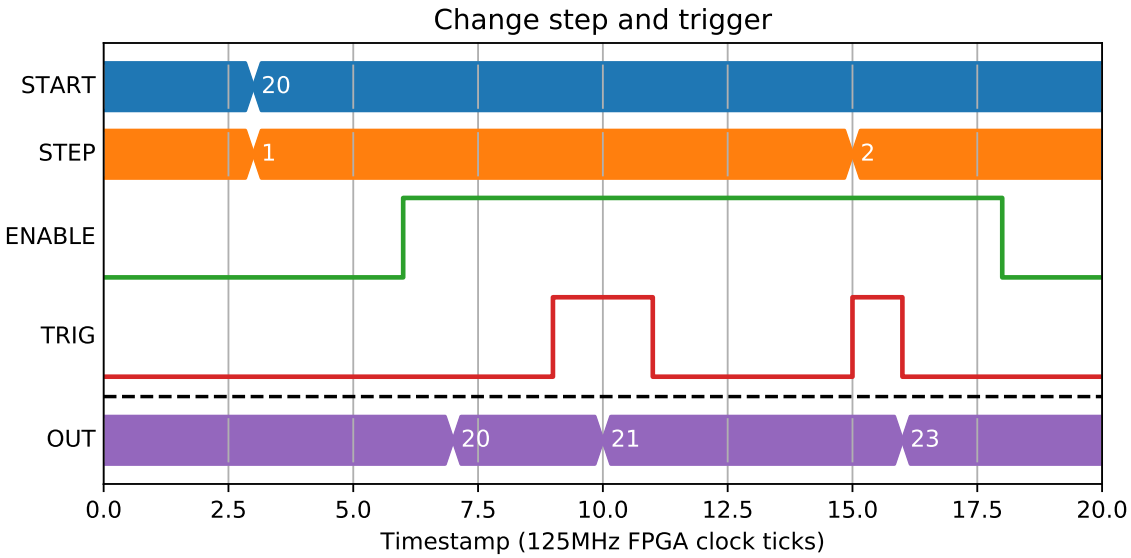
2.8.4 Edge cases

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


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


Disable and trigger
START
ENABLE
TRIG
OUT
20
20
21
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Timestamp (125MHz FPGA clock ticks)

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.



2.9 PCOMP - Position Compare [x4]

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

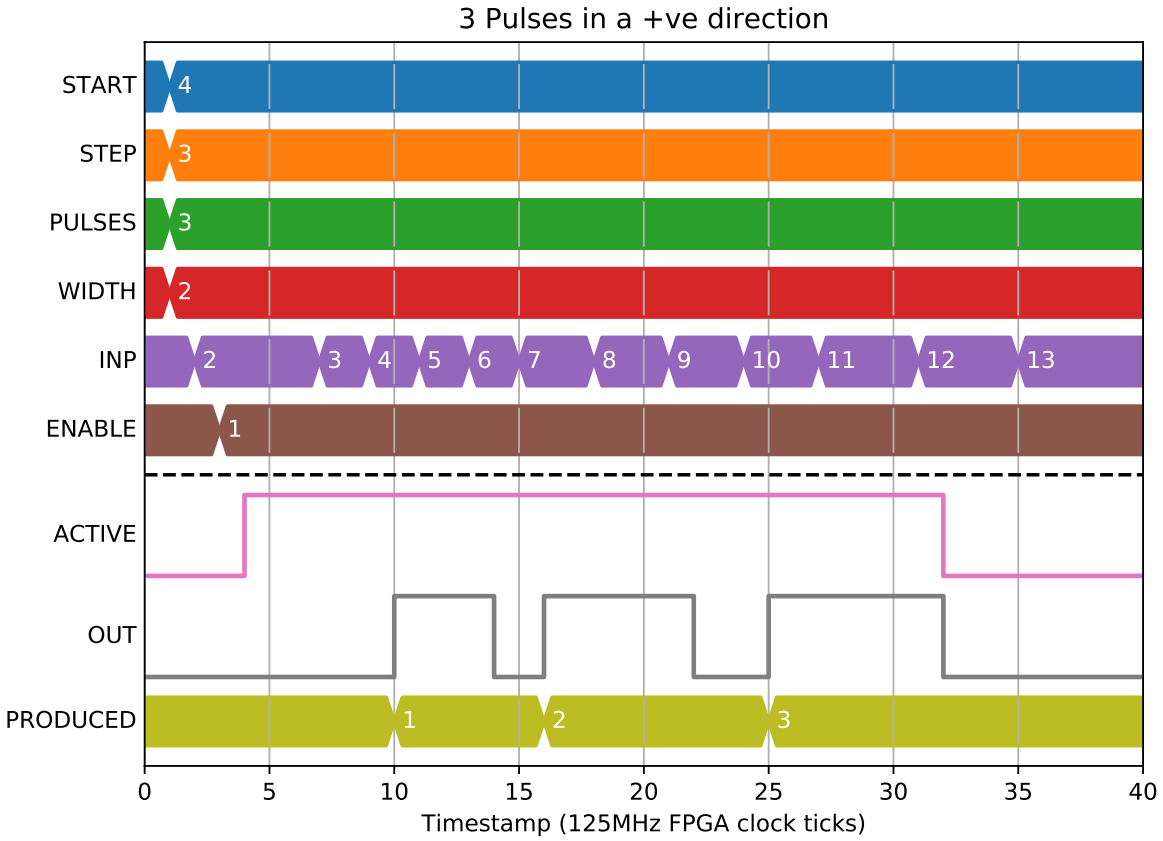
It can be used to generate a position based pulse train against an input encoder or analogue system, or to work as repeating comparator.

2.9.1 Parameters

Name	Dir	Type	Description
PRE_START	RW	Pos	INP must be this far from START before waiting for START
START	RW	Pos	Pulse absolute/relative start position value
WIDTH	RW	Pos	The relative distance between a rising and falling edge
STEP	RW	Pos	The relative distance between successive rising edges
PULSES	RW	UInt32	The number of pulses to produce, 0 means infinite
RELATIVE	RW	Enum	If 1 then START is relative to the position of INP at enable * 0: Absolute * 1: Relative
DIR	RW	Enum	Direction to apply all relative offsets to - 0: Positive - 1: Negative - 2: Either
ENABLE	In	Bit	Stop on falling edge, reset and enable on rising edge
INP	In	Pos	Position data from position-data bus
ACTIVE	Out	Bit	Active output is high while block is in operation
OUT	Out	Bit	Output pulse train
HEALTH	R	Enum	0: OK 1: Error: Position jumped by more than STEP
PRODUCED	R	UInt32	The number of pulses produced
STATE	R	Enum	The internal statemachine state - 0: WAIT_ENABLE - 1: WAIT_PRE_START - 2: WAIT_START - 3: WAIT_WIDTH - 4: WAIT_STEP

2.9.2 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:

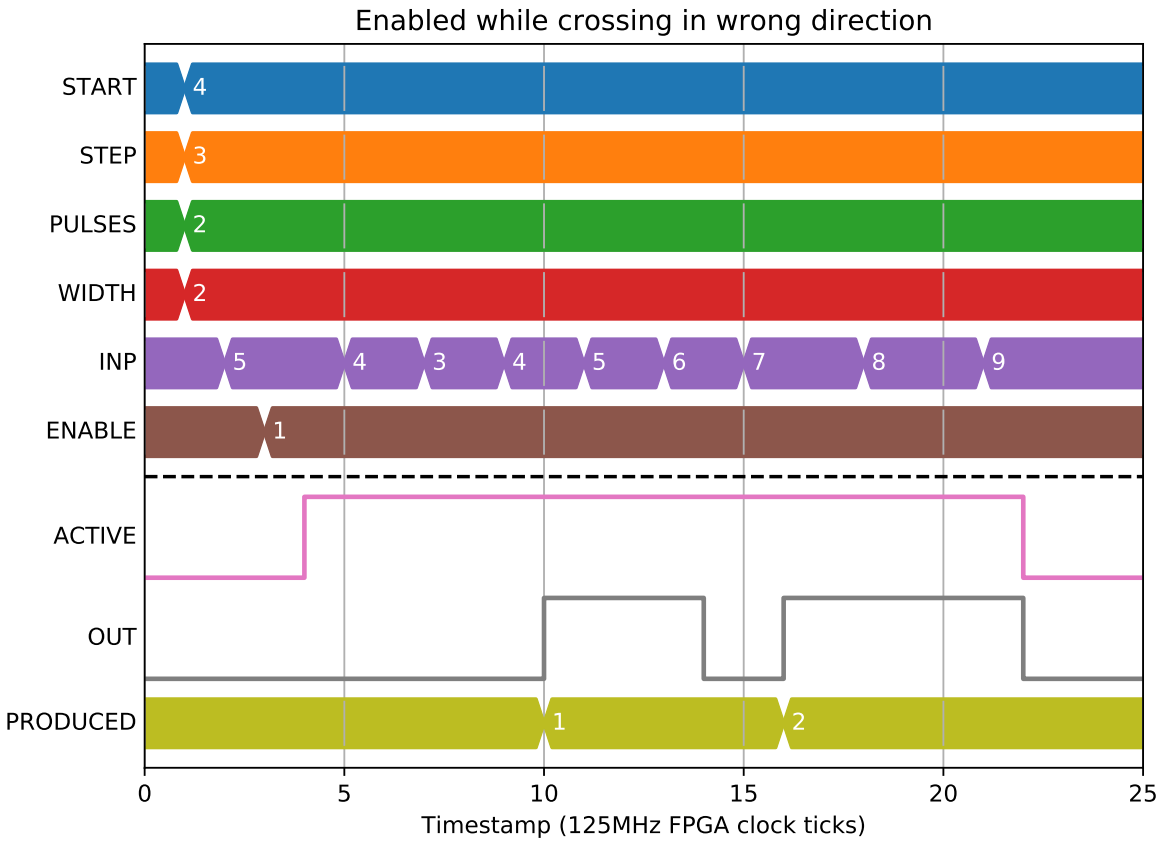


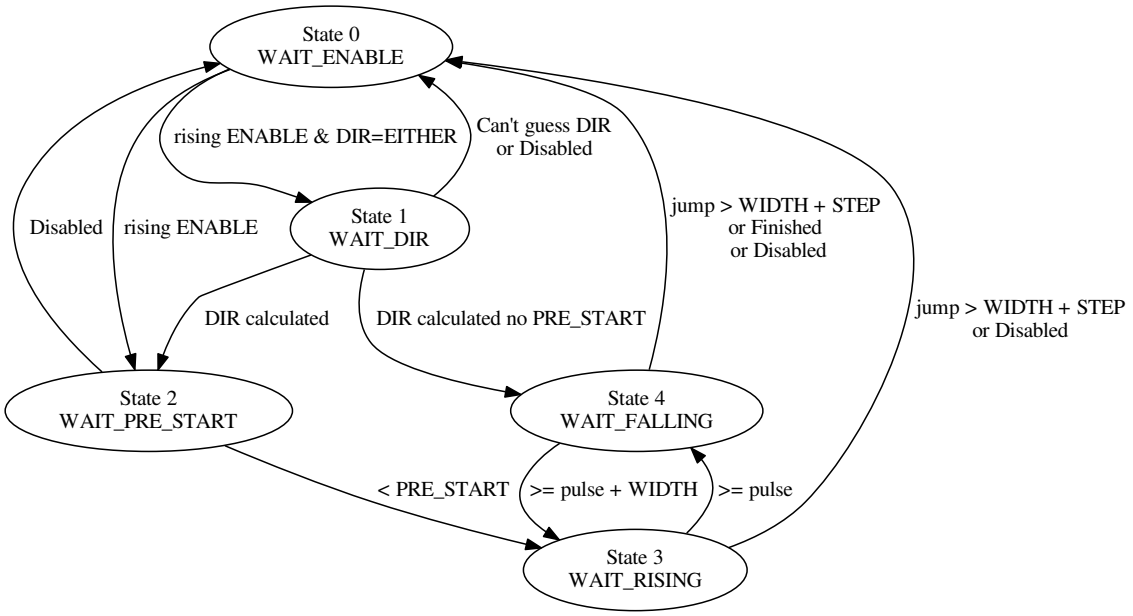
But if we get the direction wrong, we won't get the first pulse until we cross START in the correct direction:

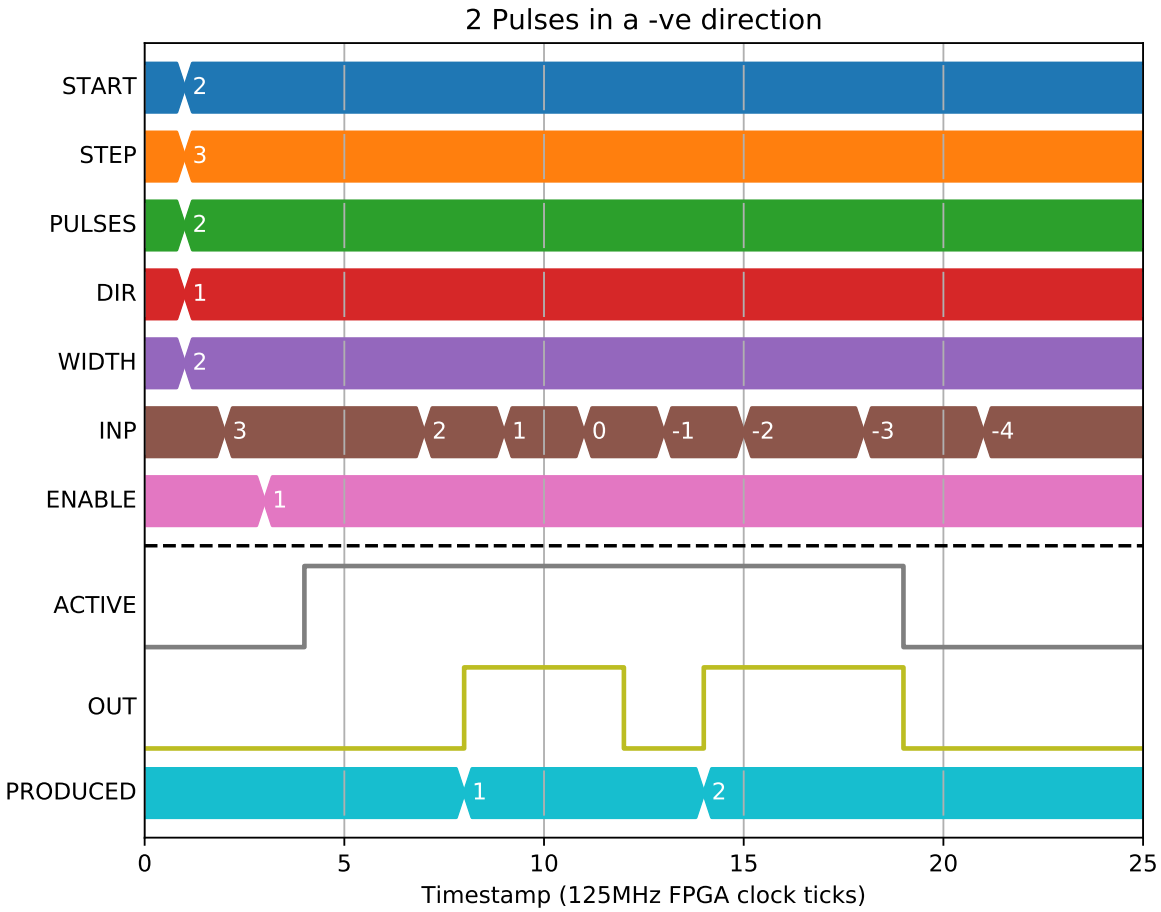
Moving in a negative direction works in a similar way. Note that WIDTH and PULSE still have positive values:

2.9.3 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:

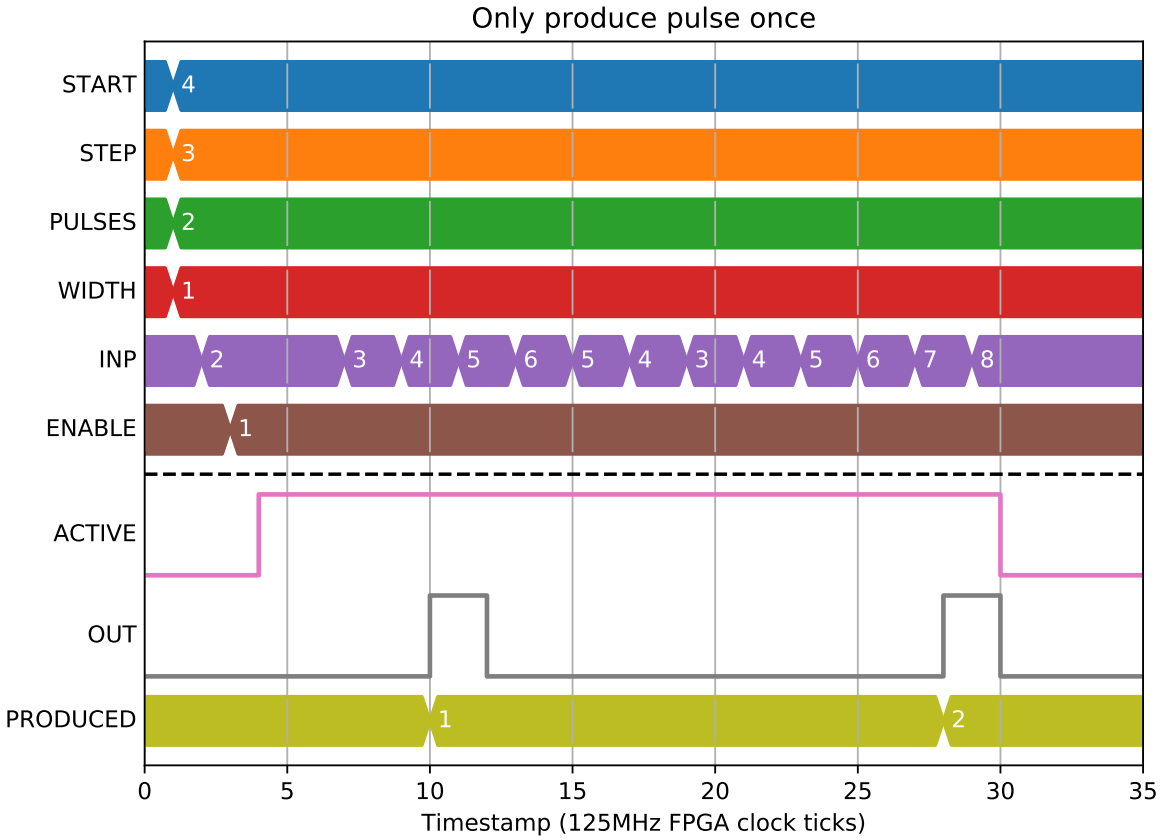

Enabled while crossing in wrong direction
START
STEP
PULSES
WIDTH
INP
ENABLE
ACTIVE
OUT
PRODUCED
4
3
2
2
5
4
3
4
5
6
7
8
9
1
1
2
0
5
10
15
20
25
Timestamp (125MHz FPGA clock ticks)


State 0
WAIT_ENABLE
rising ENABLE & DIR=EITHER
Can't guess DIR
or Disabled
State 1
WAIT_DIR
Disabled
rising ENABLE
jump > WIDTH + STEP
or Finished
or Disabled
DIR calculated
DIR calculated no PRE_START
jump > WIDTH + STEP
or Disabled
State 2
WAIT_PRE_START
State 4
WAIT_FALLING
< PRE_START
>= pulse + WIDTH
>= pulse
State 3
WAIT_RISING


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

2.9.4 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:

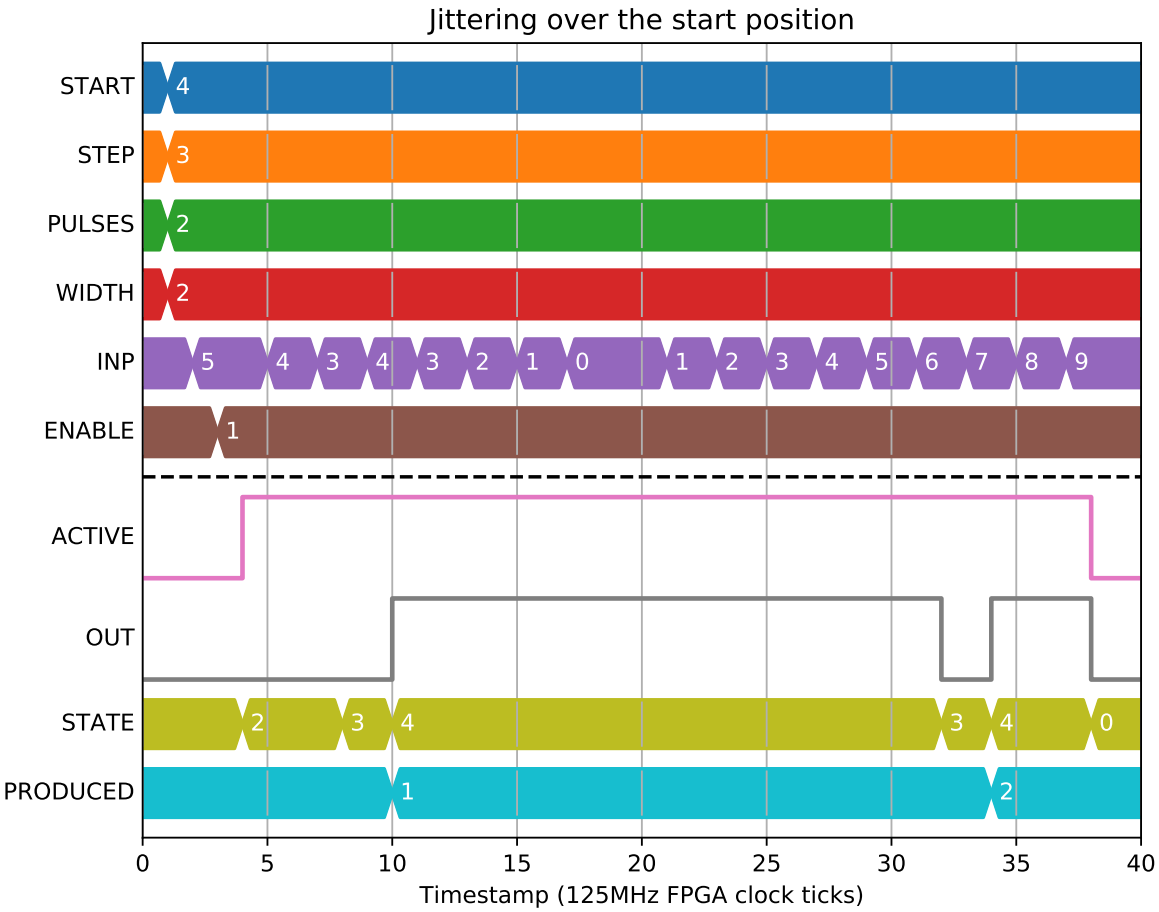
2.9.5 Guessing the direction

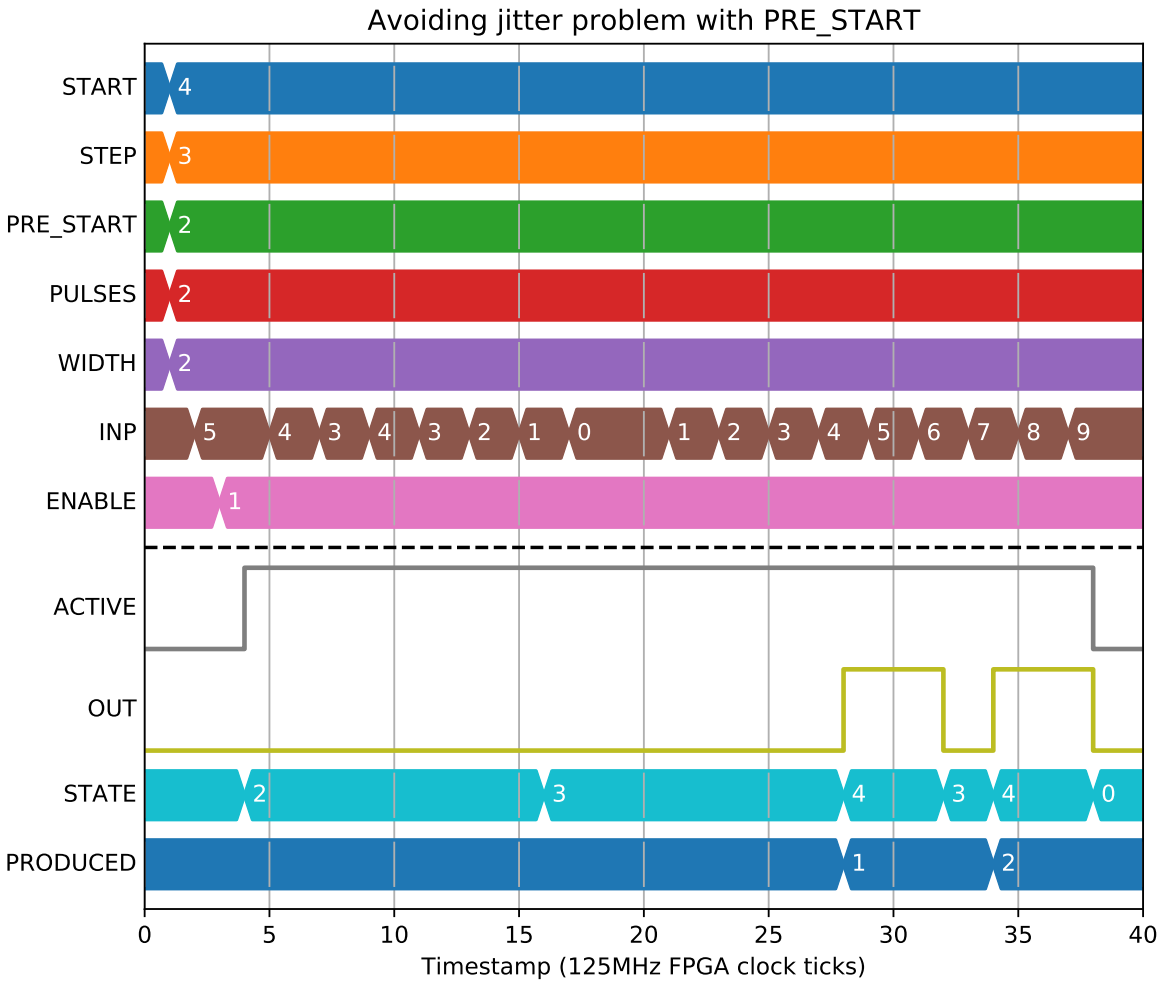
We can also ask to the Block to calculate direction for us:

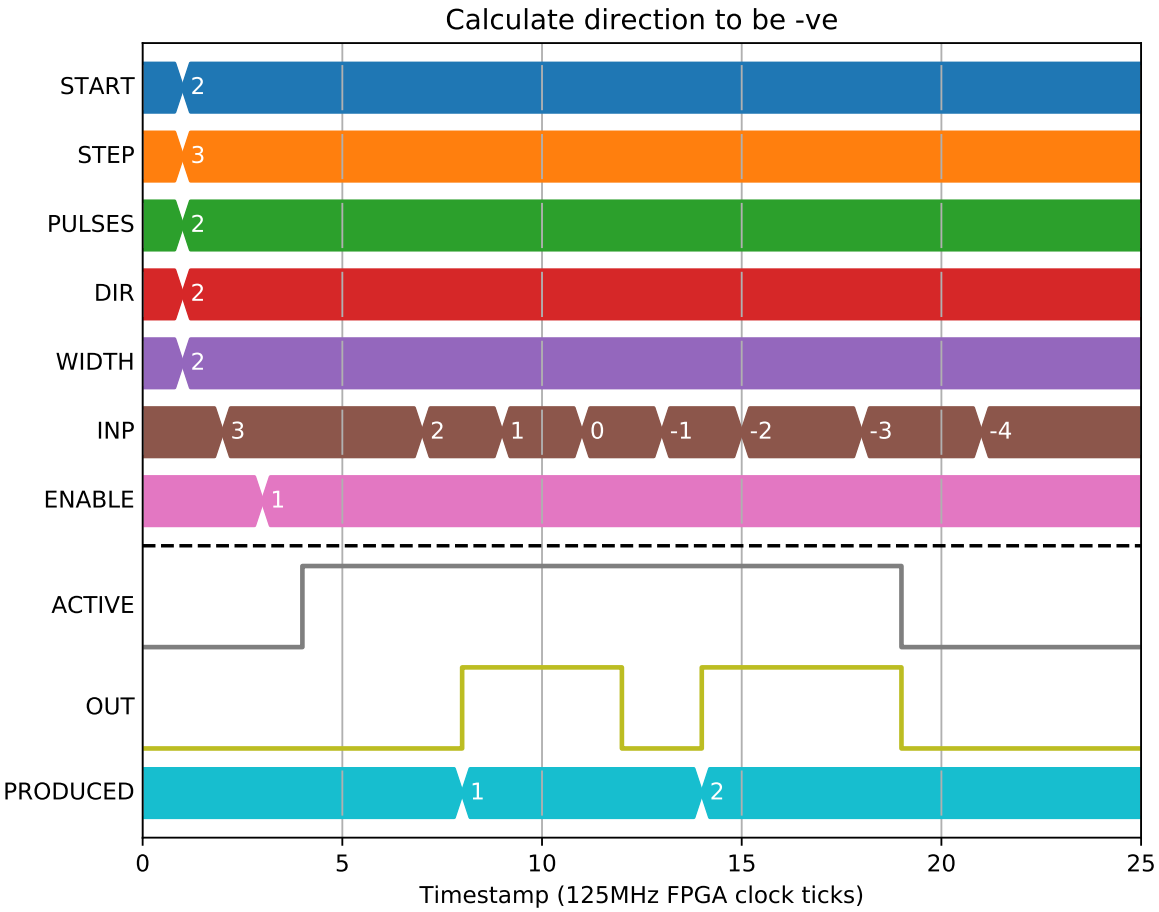
This is a one time calculation of direction at the start of operation, once the encoder has been moved enough to guess the direction then it is fixed until the Block has finished producing pulses:

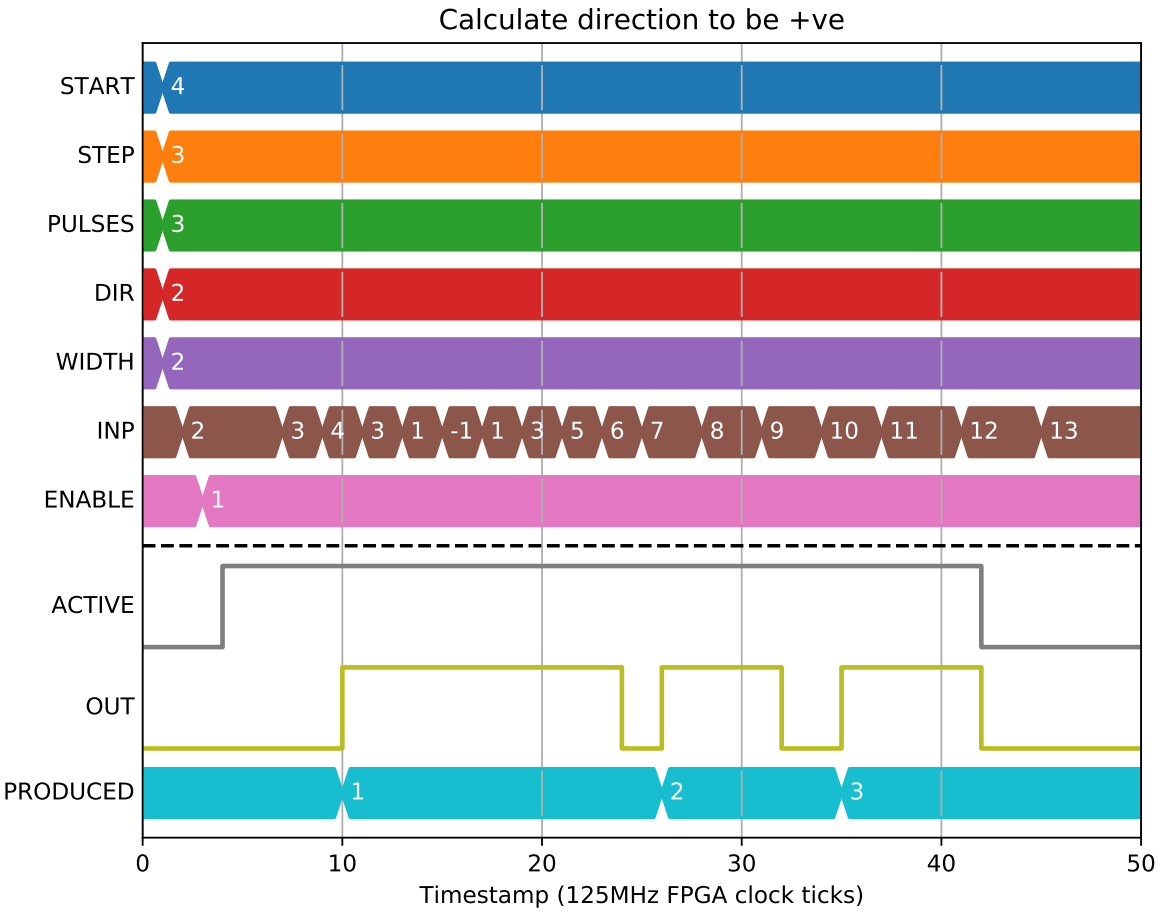
2.9.6 Interrupting a scan

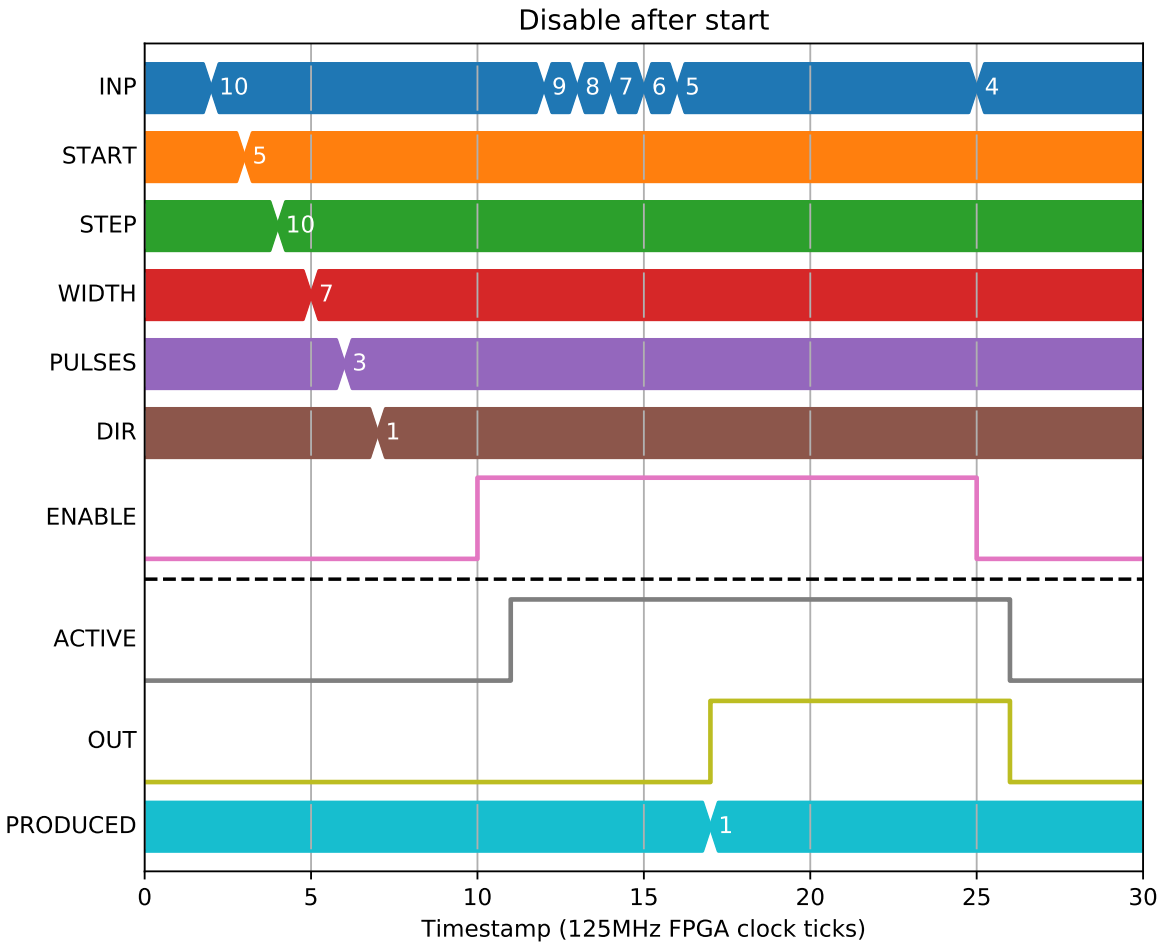
When the ENABLE input is set low the output will cease. This will happen even if the ENABLE is set low when there are still cycles of the output pulse to generate, or if the ENABLE = 0 is set at the same time as a position match.

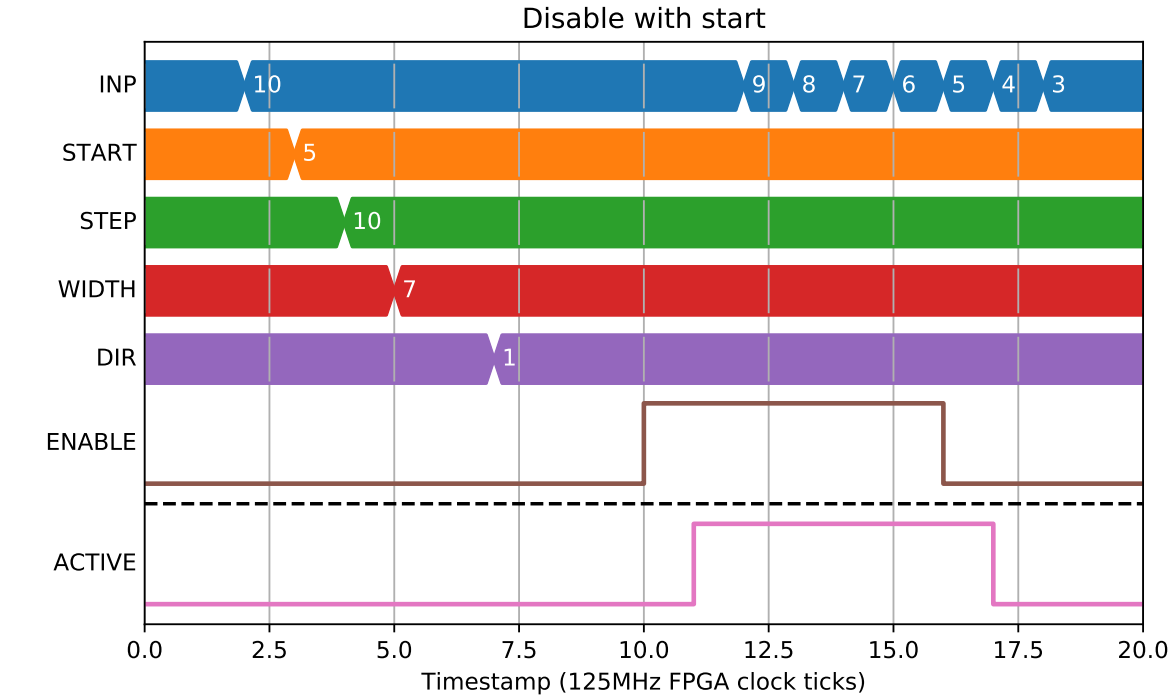

Jittering over the start position
START
STEP
PULSES
WIDTH
INP
ENABLE
ACTIVE
OUT
STATE
PRODUCED
4
3
2
2
5 4 3 4 3 2 1 0 1 2 3 4 5 6 7 8 9
1
2 3 4 3 4 0
1 2
0 5 10 15 20 25 30 35 40
Timestamp (125MHz FPGA clock ticks)


Avoiding jitter problem with PRE_START
START
STEP
PRE_START
PULSES
WIDTH
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
PULSES
DIR
WIDTH
INP
ENABLE
ACTIVE
OUT
PRODUCED
2
3
2
2
2
3
2
1
0
-1
-2
-3
-4
1
1
2
0
5
10
15
20
25
Timestamp (125MHz FPGA clock ticks)

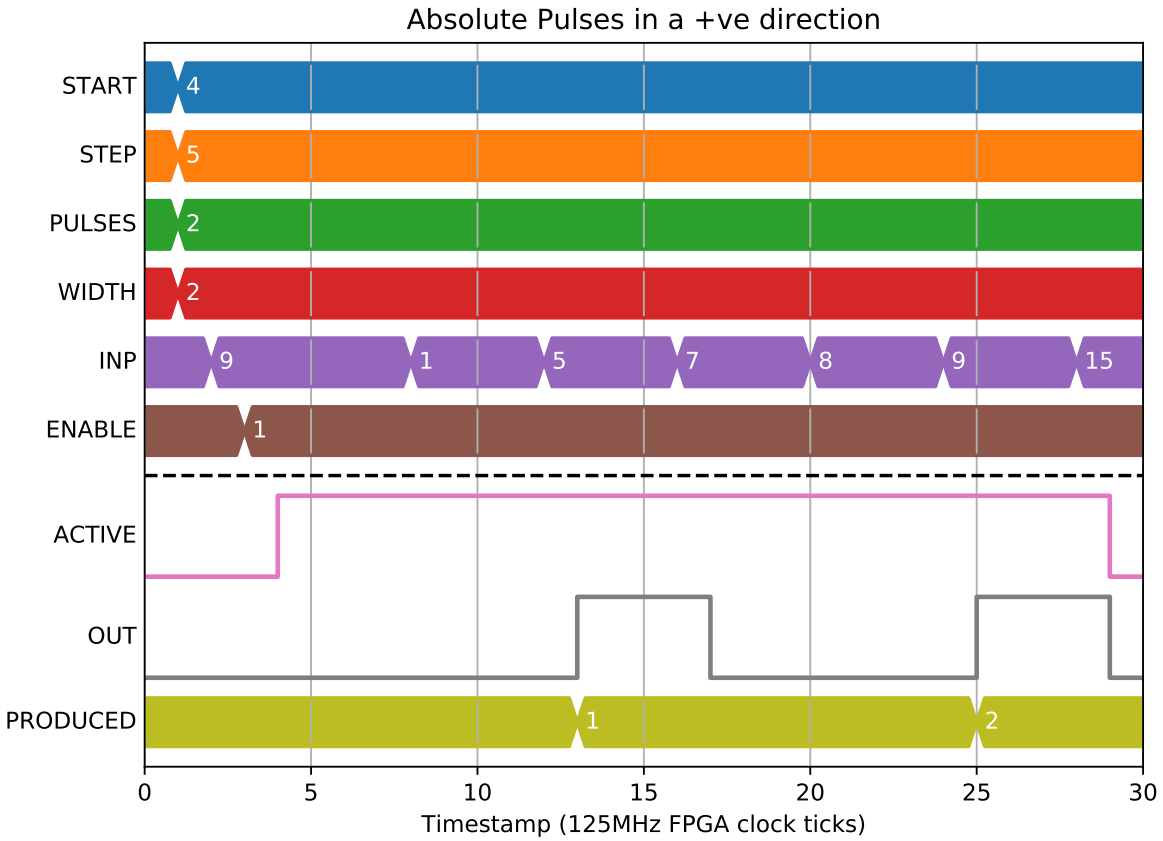

Calculate direction to be +ve
START
STEP
PULSES
DIR
WIDTH
INP
ENABLE
ACTIVE
OUT
PRODUCED
4
3
3
2
2
2 3 4 3 1 -1 1 3 5 6 7 8 9 10 11 12 13
1
1
2
3
0
10
20
30
40
50
Timestamp (125MHz FPGA clock ticks)


Disable after start
INP
START
STEP
WIDTH
PULSES
DIR
ENABLE
ACTIVE
OUT
PRODUCED
10
9
8
7
6
5
4
5
10
7
3
1
1
0
5
10
15
20
25
30
Timestamp (125MHz FPGA clock ticks)


Disable with start
INP
START
STEP
WIDTH
DIR
ENABLE
ACTIVE
10
9
8
7
6
5
4
3
5
10
7
1
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Timestamp (125MHz FPGA clock ticks)

2.9.7 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:

2.9.8 Relative position compare

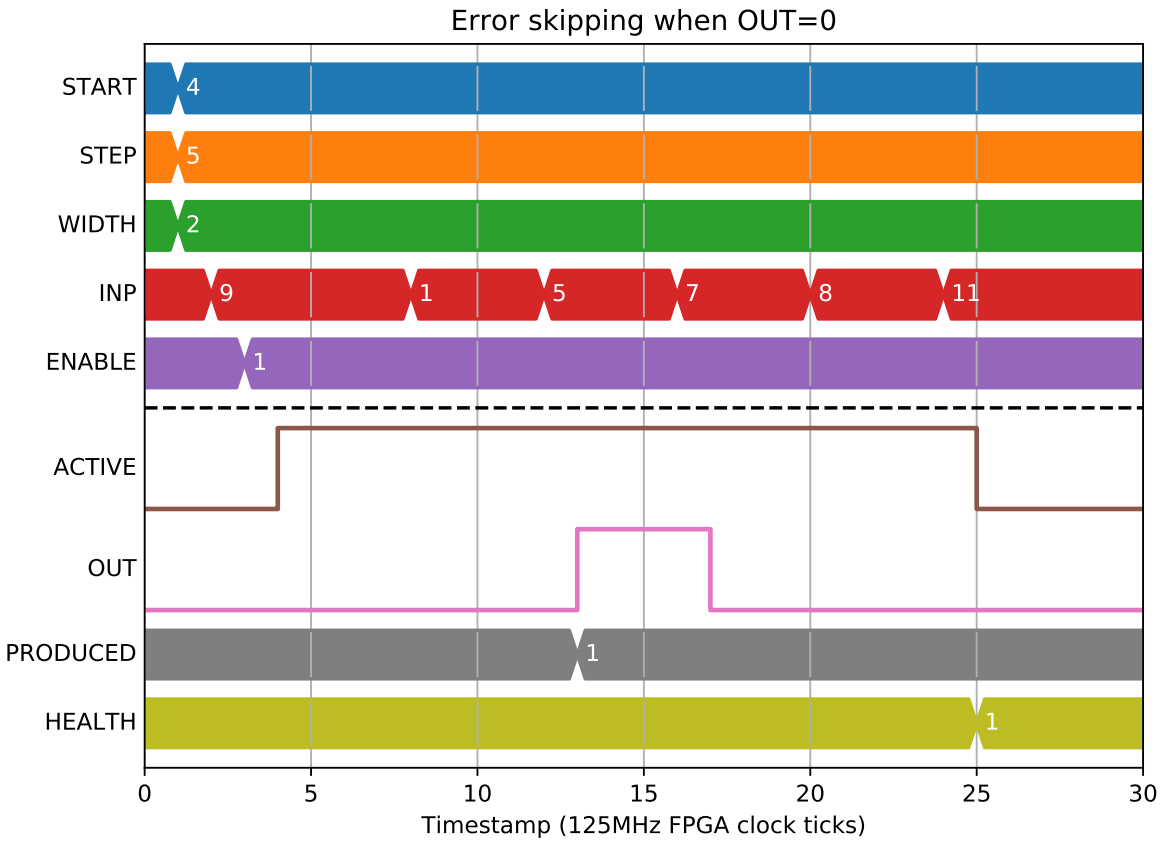
We may want to nest position compare blocks, or respond to some external event. In which case, we expose the option to a position compare relative to the latched position at the start:

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:



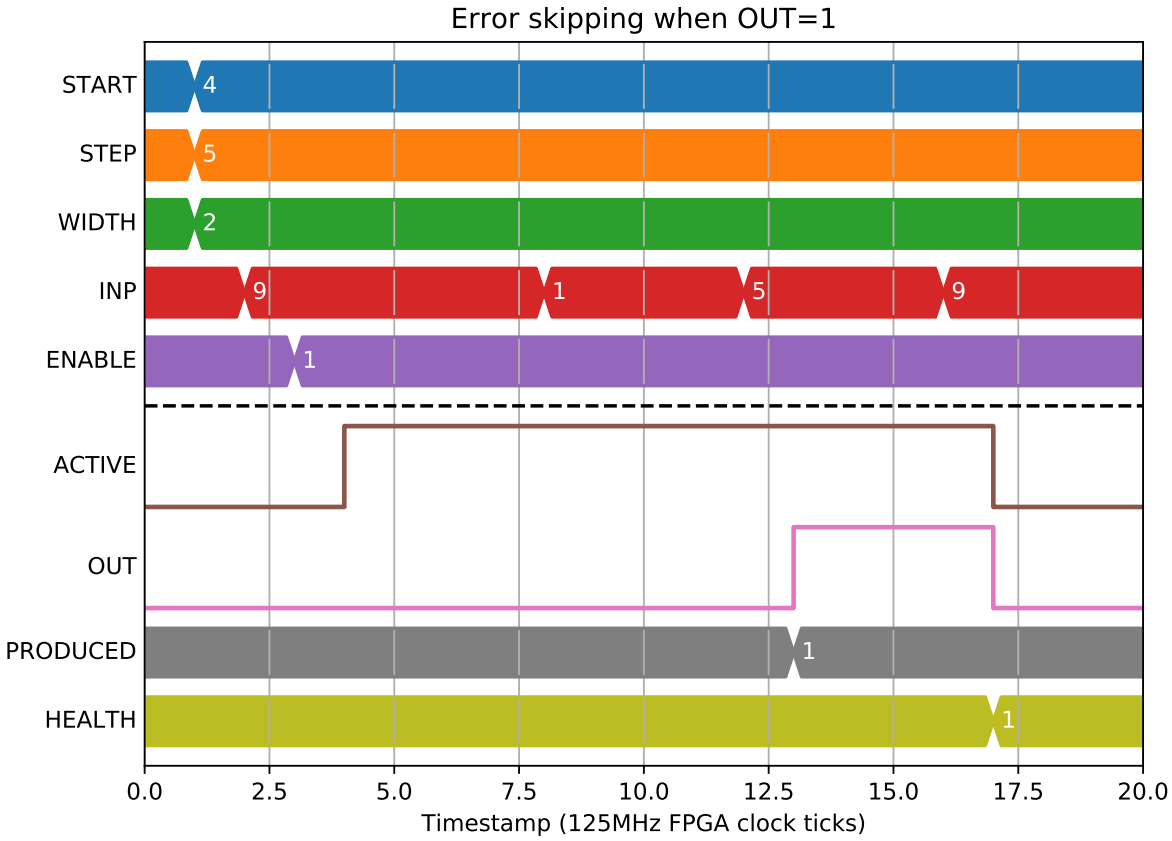
2.9.9 Use as a Schmitt trigger

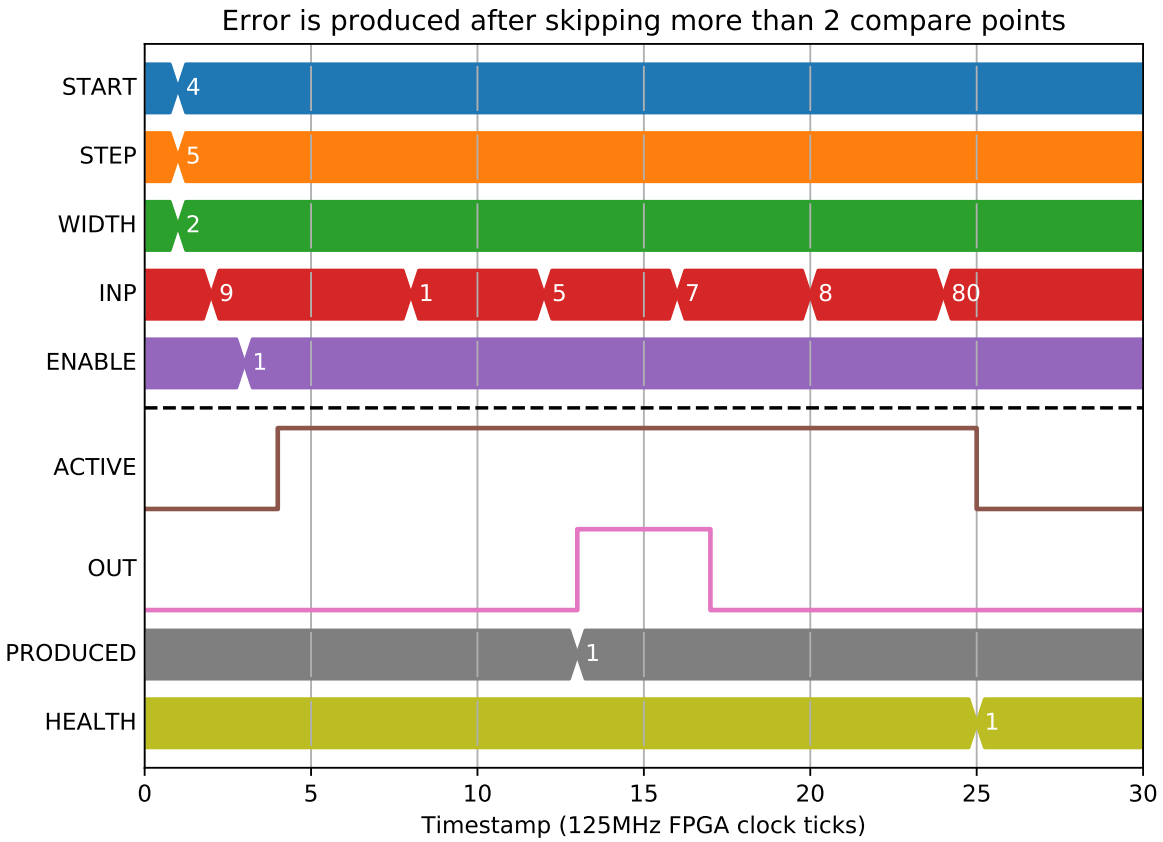
We can also make use of a special case with STEP=0 and a negative WIDTH to create a Schmitt trigger that will always trigger at START, and turn off when INP has dipped WIDTH below START:

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:

2.10 PCAP - Position Capture

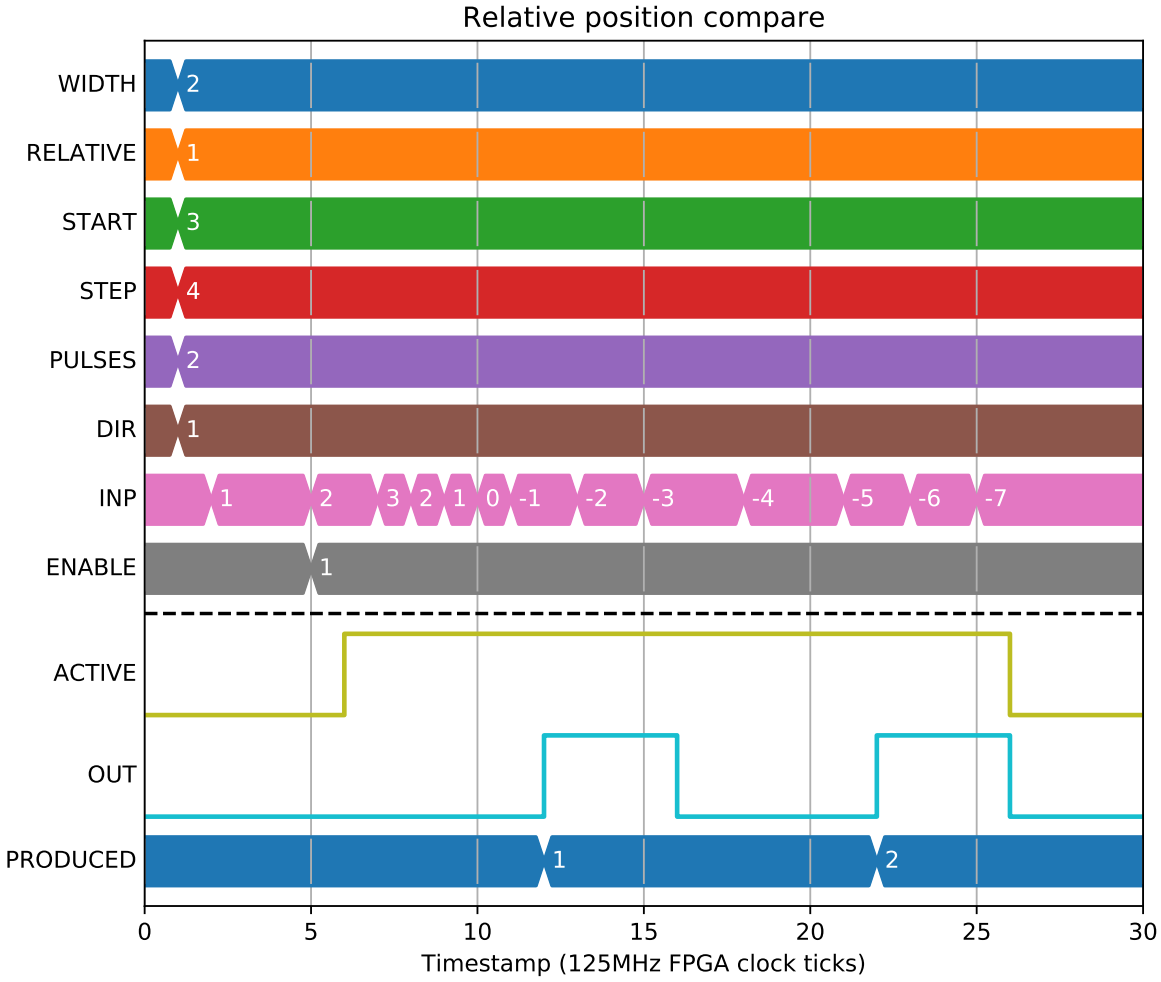
Position capture has the capability to capture anything that is happening on the pos_bus or bit_bus. It listens to ENABLE, GATE and CAPTURE signals, and can capture the value at capture, sum, min and max.

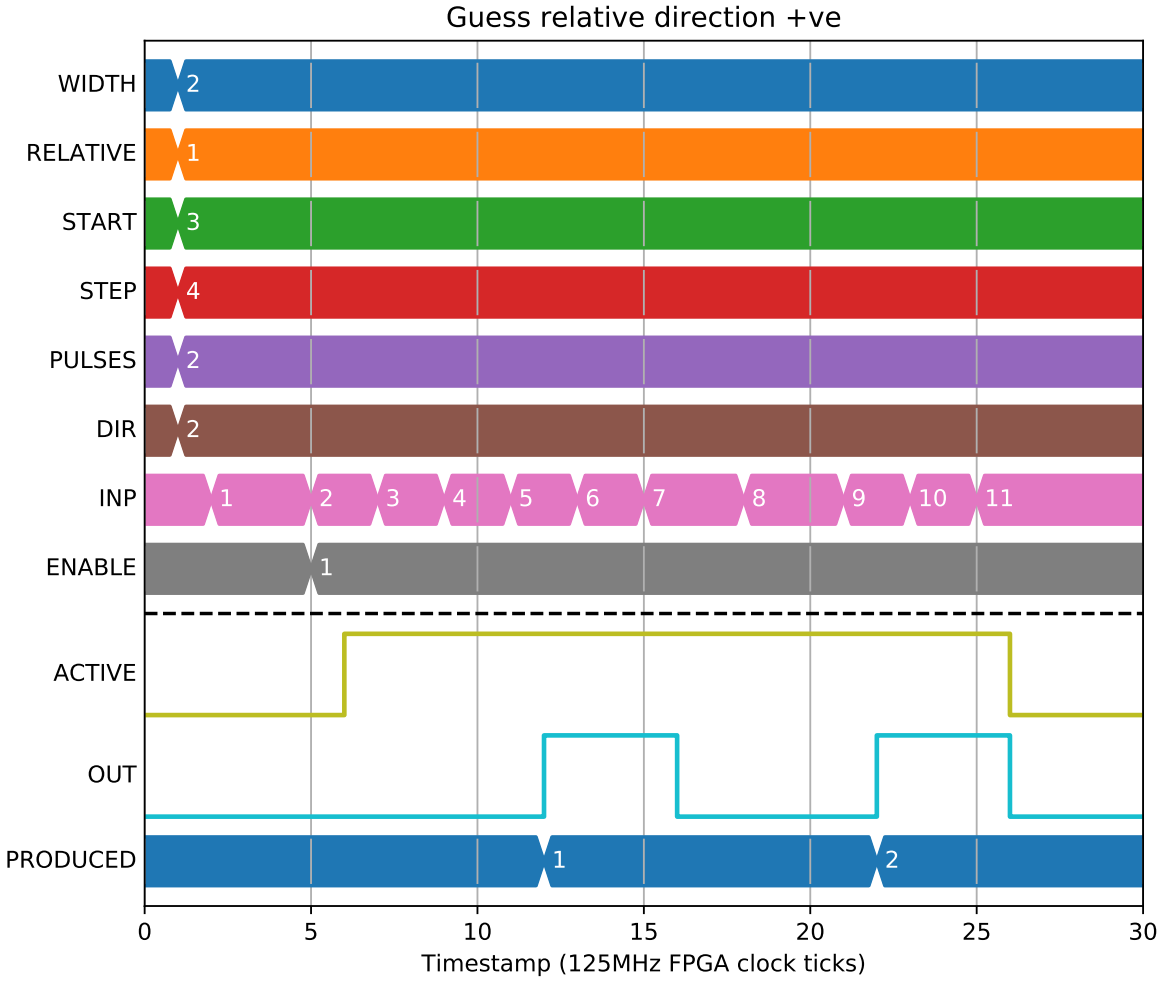

Error skipping when OUT=1
START
STEP
WIDTH
INP
ENABLE
ACTIVE
OUT
PRODUCED
HEALTH
4
5
2
9
1
5
9
1
1
1
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Timestamp (125MHz FPGA clock ticks)

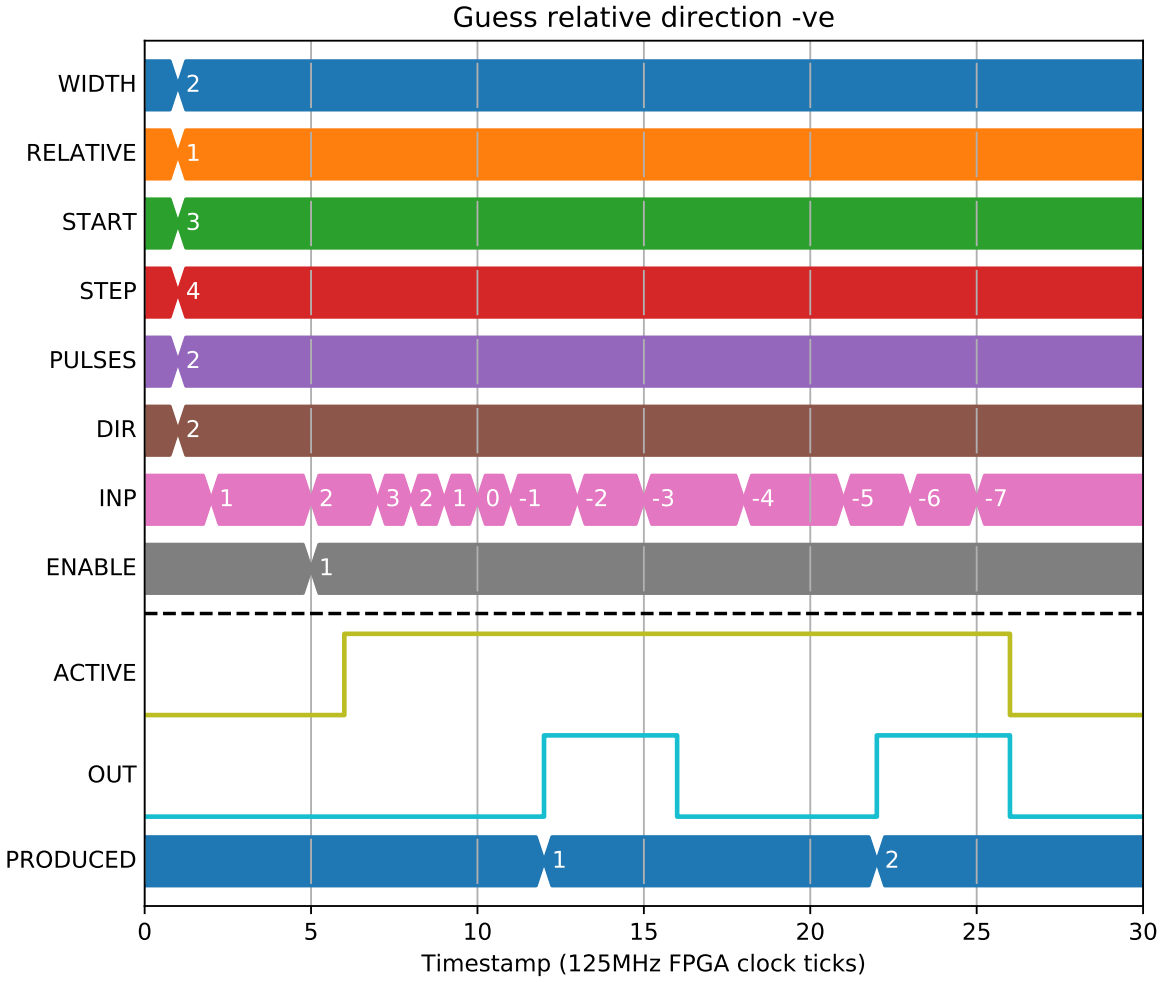


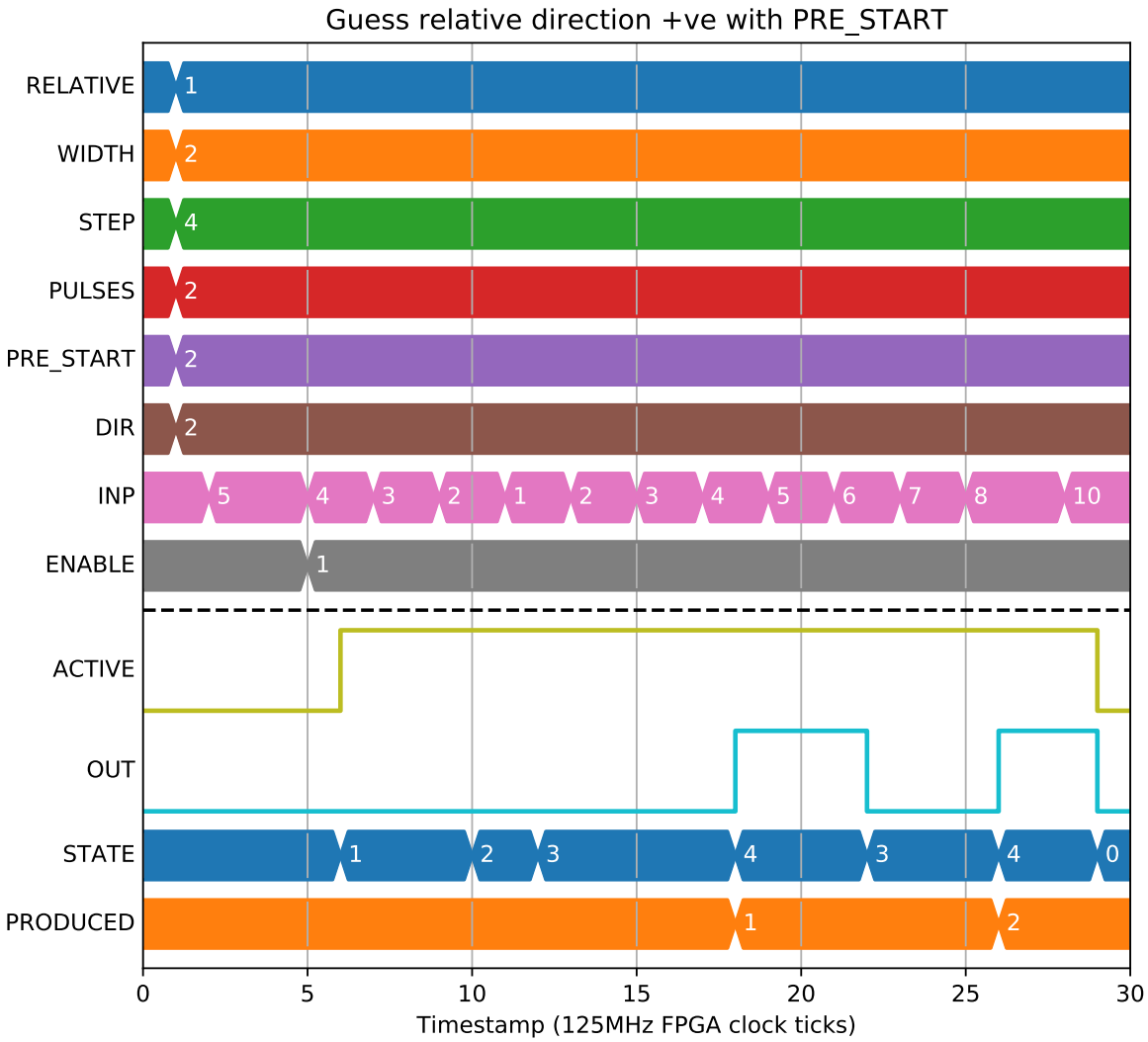
2.10.1 Parameters

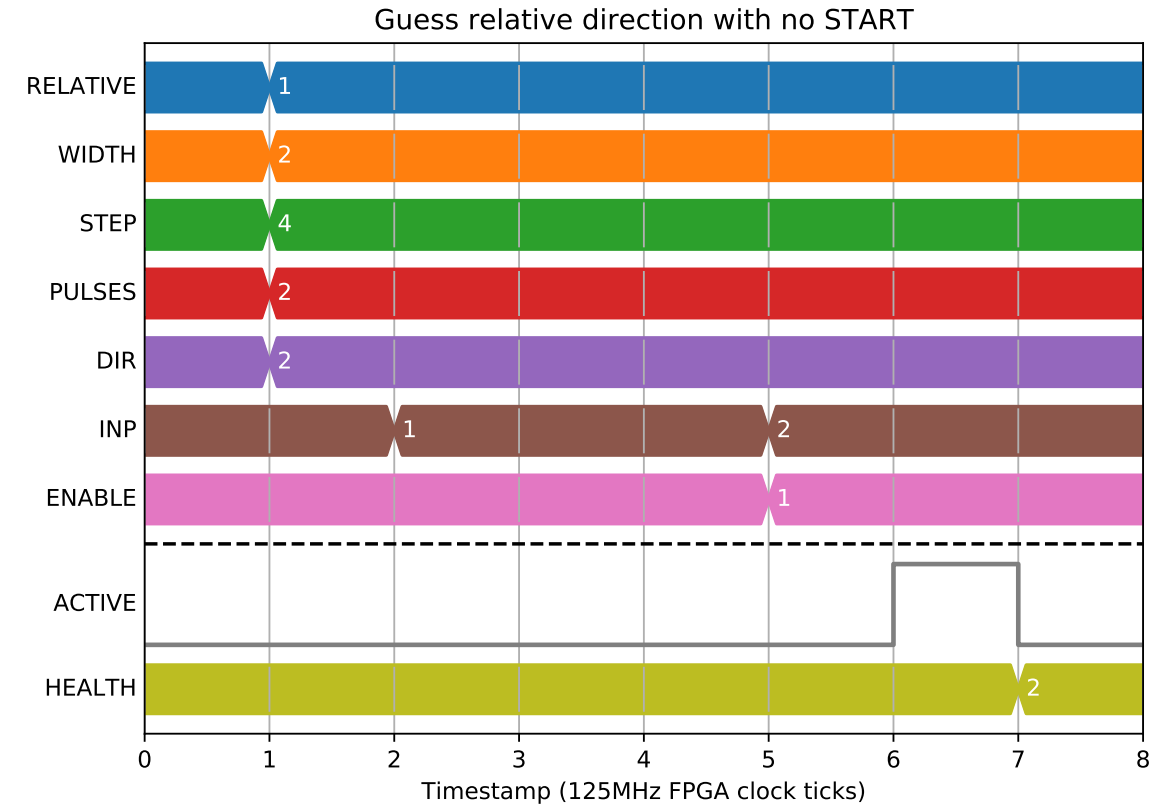
Name	Dir	Type	Description
ENABLE	In	Bit	After arm, when high start capture, when low disarm
GATE	In	Bit	After enable, only process gated values if high
CAPTURE	In	Bit	On selected edge capture current value and gated data
CAPTURE_EDGE	R/W	Enum	Which edge of capture input signal triggers capture 0 - Rising 1 - Falling 2 - Rising or Falling
SHIFT_SUM	R/W	Int	Shift sum data, use if > 2**32 samples required in sum/average
HEALTH	R	Enum	Was last capture successful? 0 - OK 1 - Capture events too close together 2 - Samples overflow

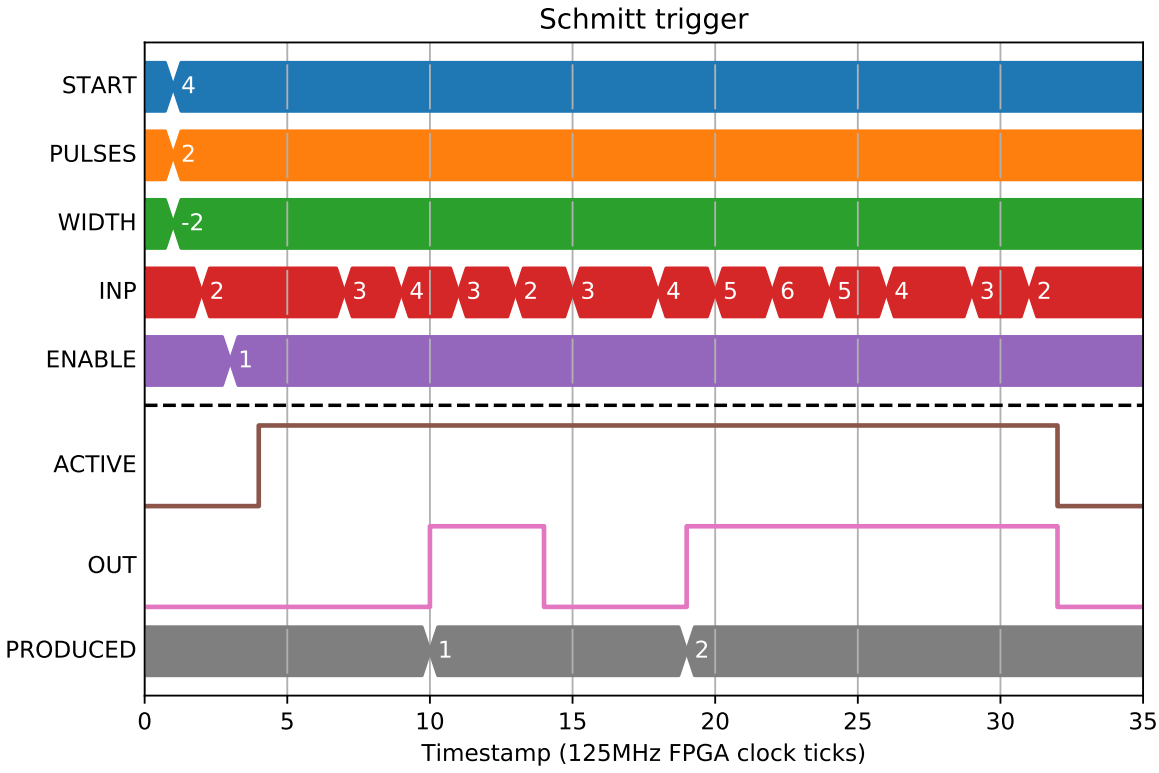

Relative position compare
WIDTH
RELATIVE
START
STEP
PULSES
DIR
INP
ENABLE
ACTIVE
OUT
PRODUCED
2
1
3
4
2
1
1 2 3 2 1 0 -1 -2 -3 -4 -5 -6 -7
1
1
2
0
5
10
15
20
25
30
Timestamp (125MHz FPGA clock ticks)

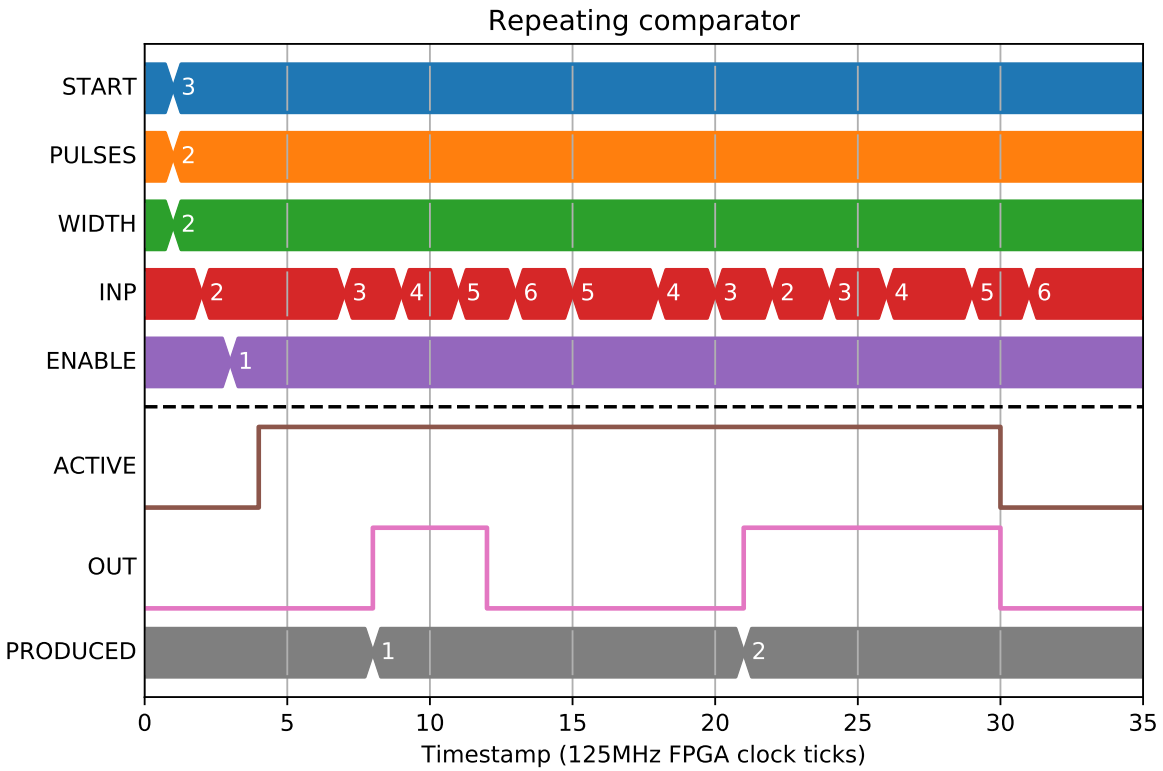

Guess relative direction +ve
WIDTH
RELATIVE
START
STEP
PULSES
DIR
INP
ENABLE
ACTIVE
OUT
PRODUCED
2
1
3
4
2
2
1 2 3 4 5 6 7 8 9 10 11
1
1
2
0 5 10 15 20 25 30
Timestamp (125MHz FPGA clock ticks)


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


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

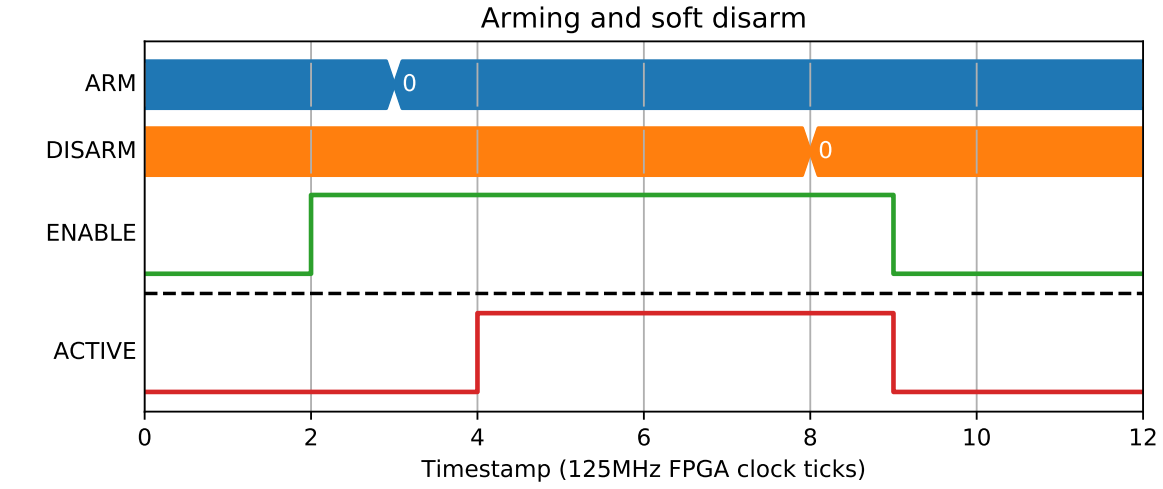

Guess relative direction with no START
RELATIVE
WIDTH
STEP
PULSES
DIR
INP
ENABLE
ACTIVE
HEALTH
1
2
4
2
2
1
2
1
2
0
1
2
3
4
5
6
7
8
Timestamp (125MHz FPGA clock ticks)


Schmitt trigger
START
PULSES
WIDTH
INP
ENABLE
ACTIVE
OUT
PRODUCED
4
2
-2
2 3 4 3 2 3 4 5 6 5 4 3 2
1
1
2
0
5
10
15
20
25
30
35
Timestamp (125MHz FPGA clock ticks)

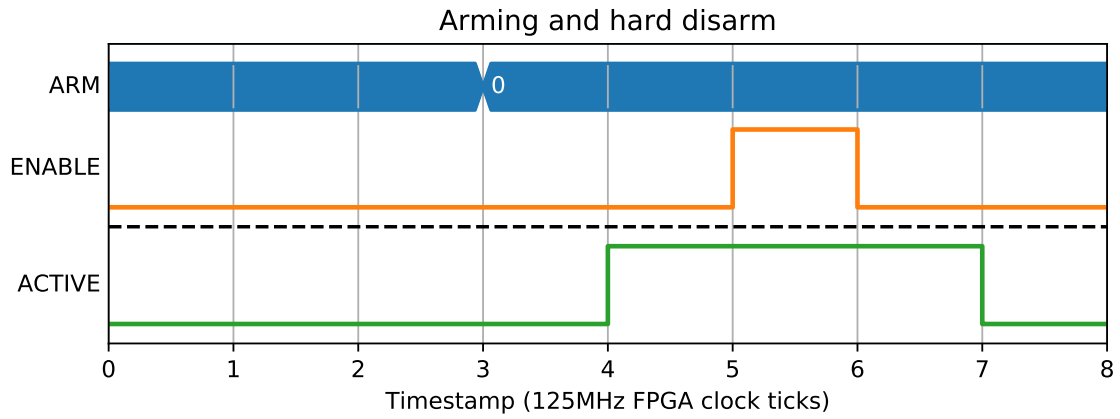


2.10.2 Arming

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:

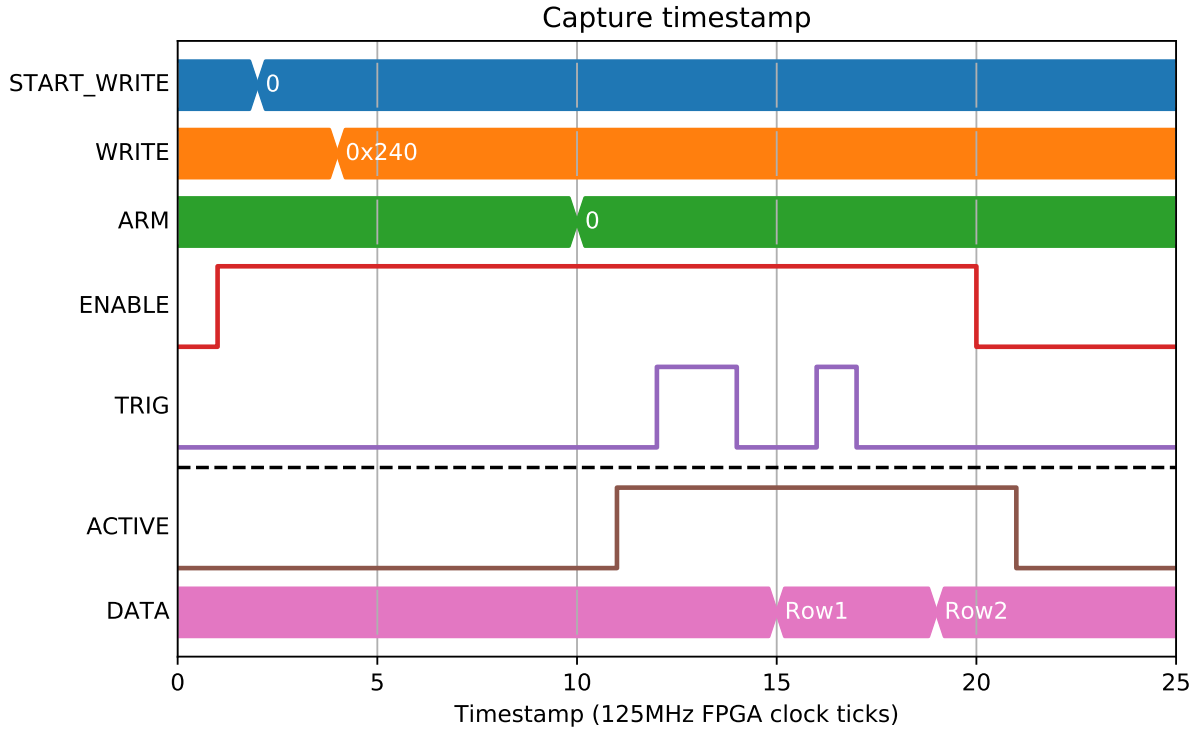


2.10.3 Capturing fields

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).



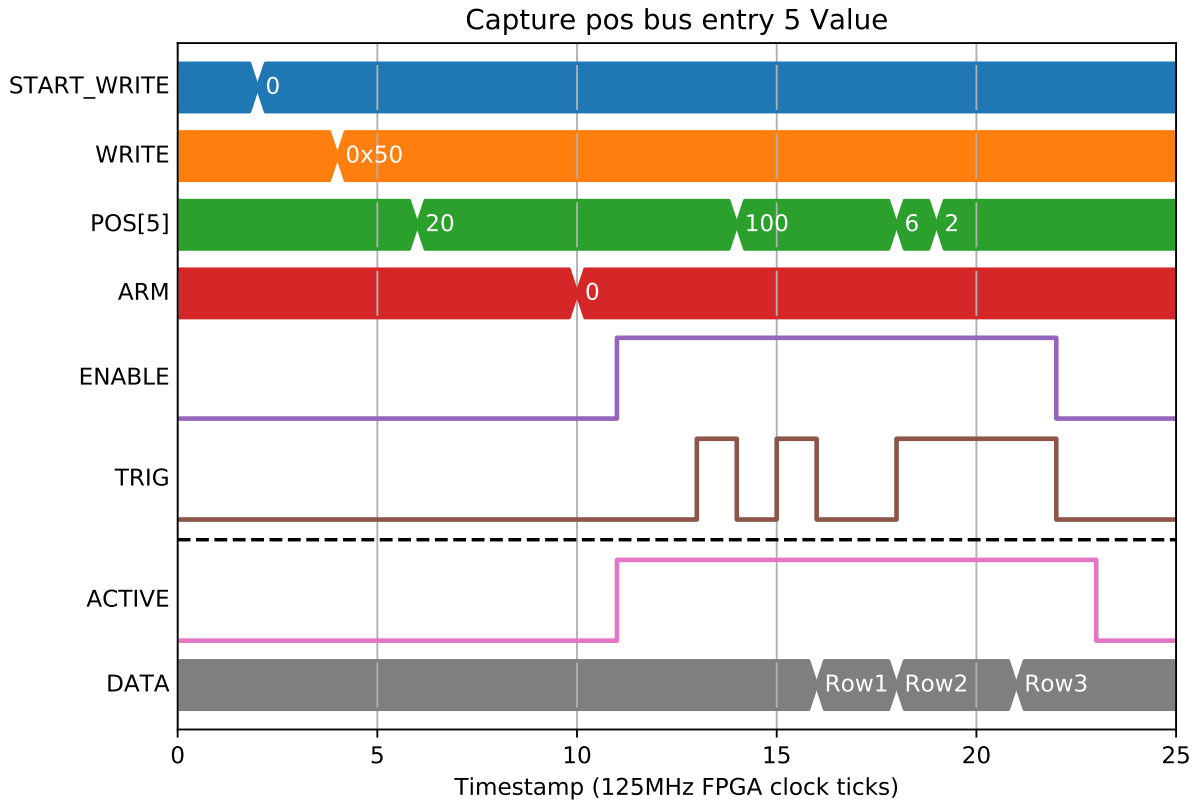
Row	0x240
0	2
1	6

2.10.4 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

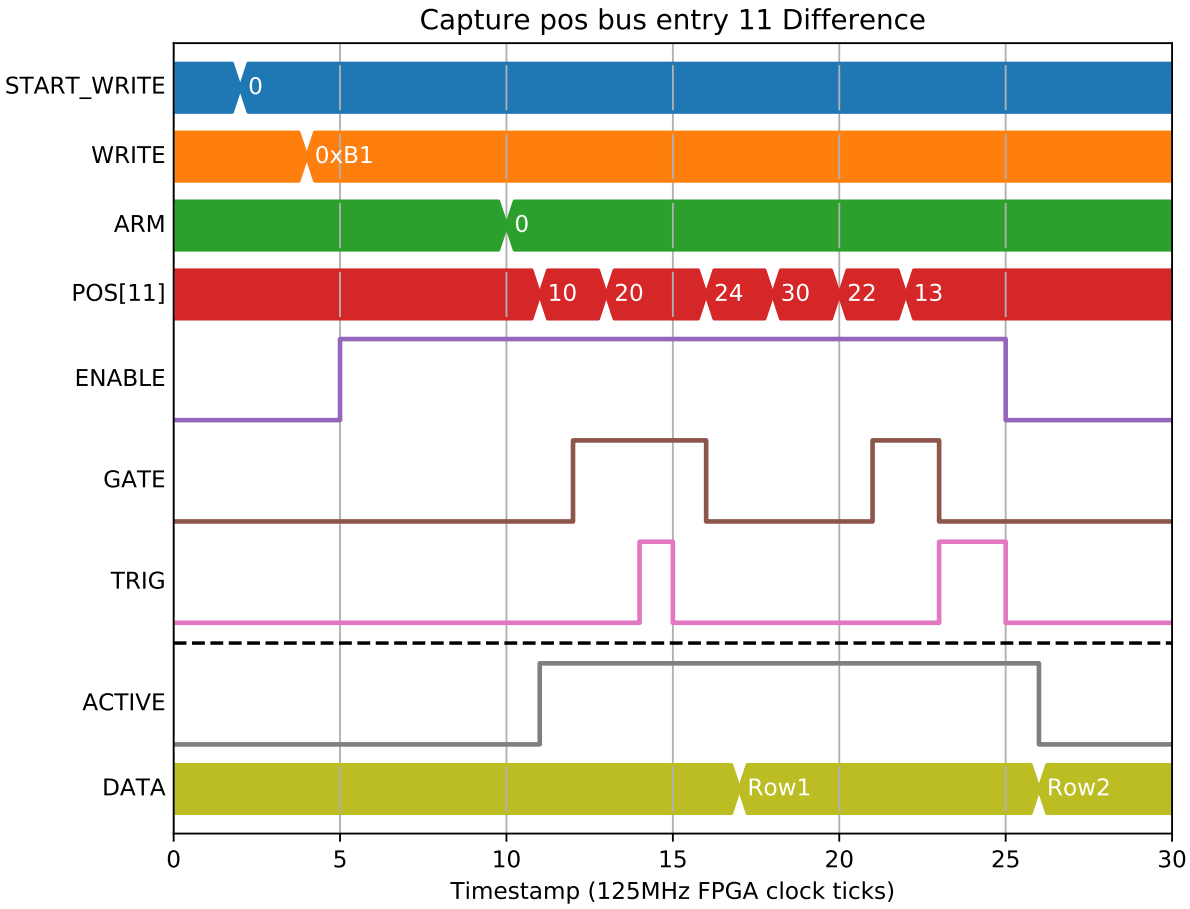
This gives an instantaneous capture of value no matter what the state of GATE:



Row	0x50
0	20
1	100
2	6

Mode 1 - Difference

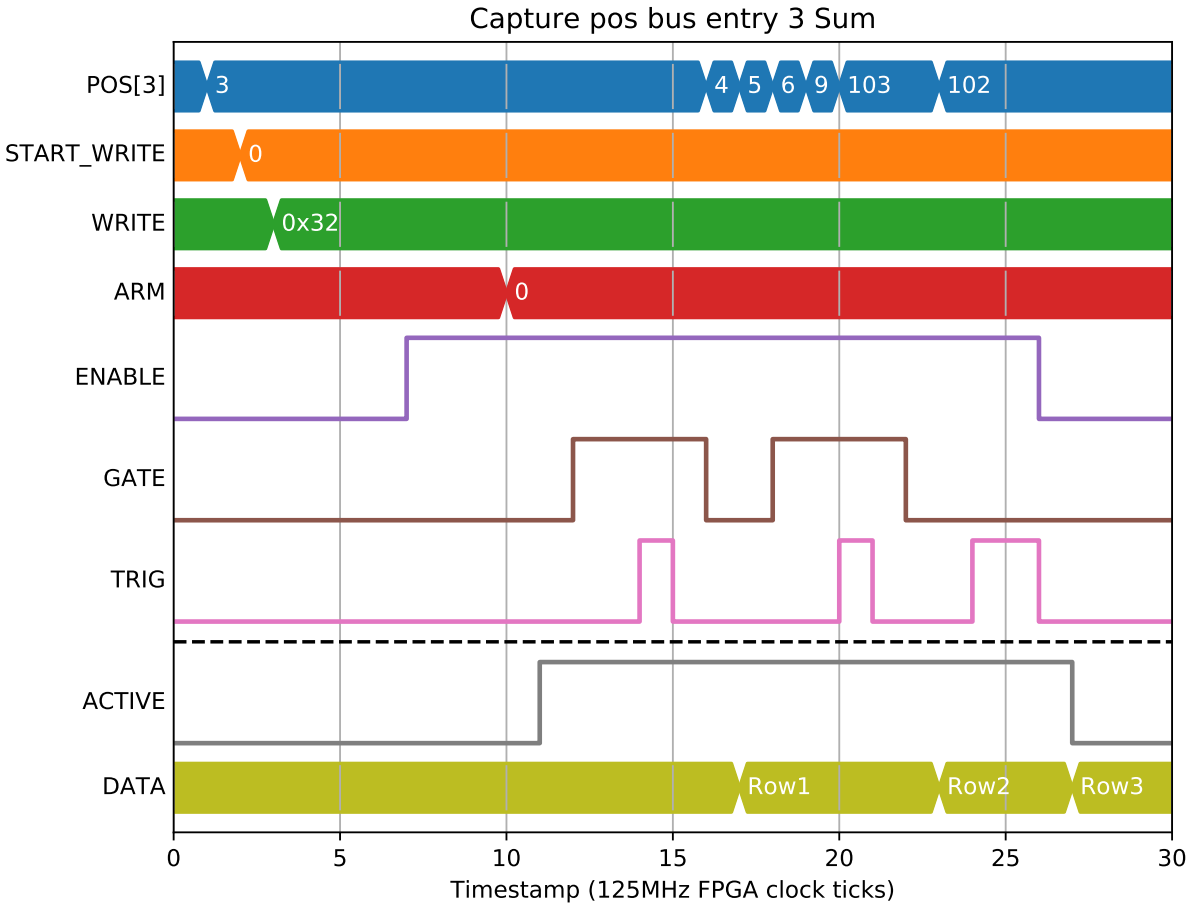
This is mainly used for something like an incrementing counter value. It will only count the differences while GATE was high:


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)

Row	0xB1
0	10
1	-5

Mode 2/3 - Sum Lo/Hi

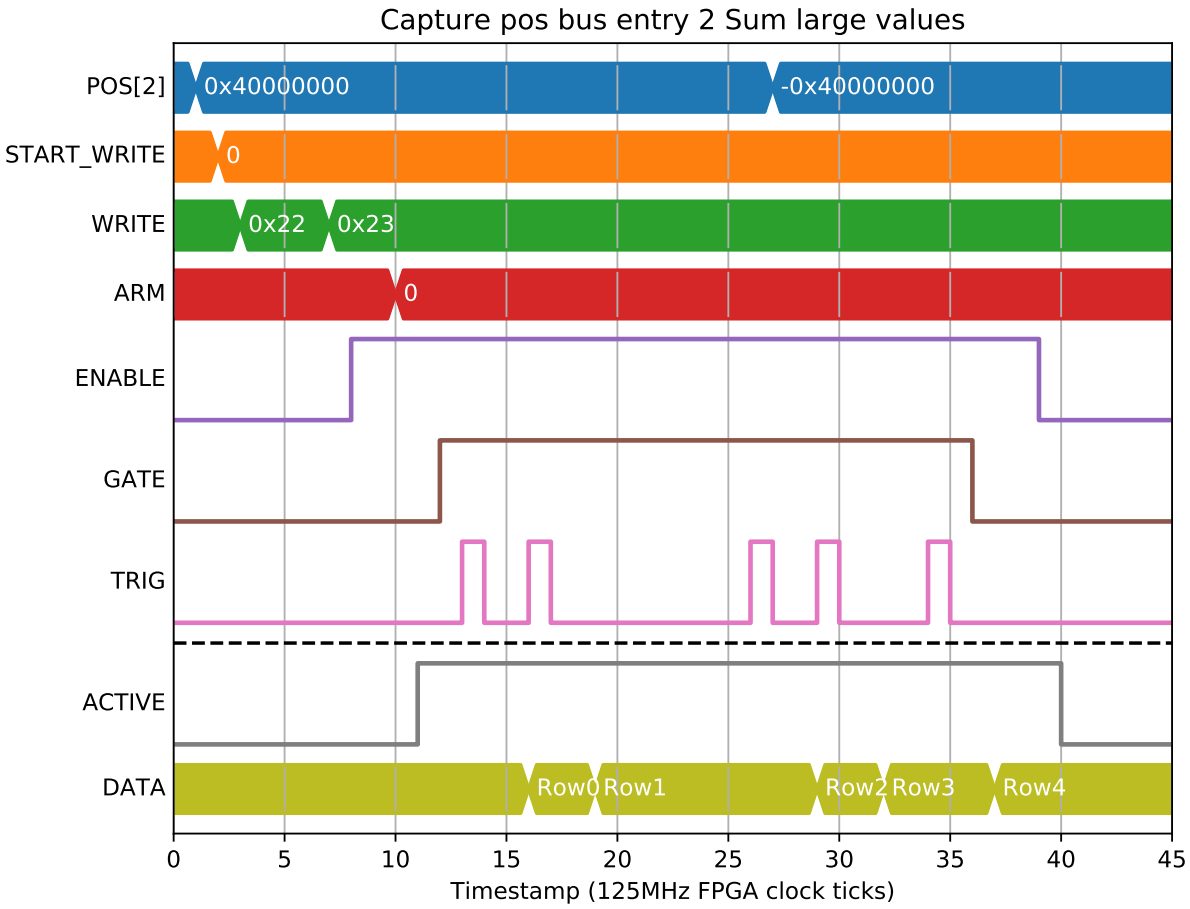
Mode 2 is the lower 32-bits of the sum of all samples while GATE was high:



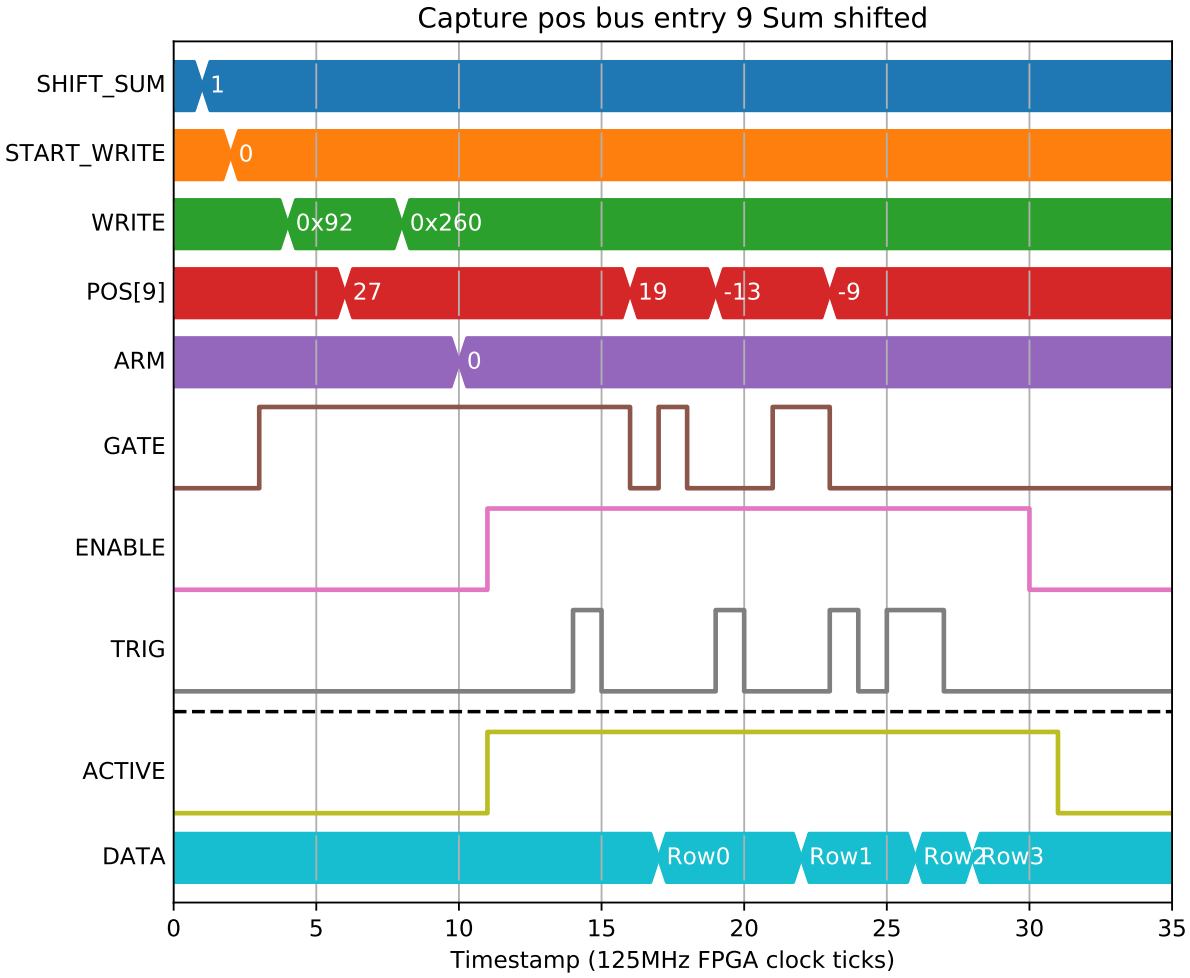
Row	0x32
0	6
1	21
2	206

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


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)

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:

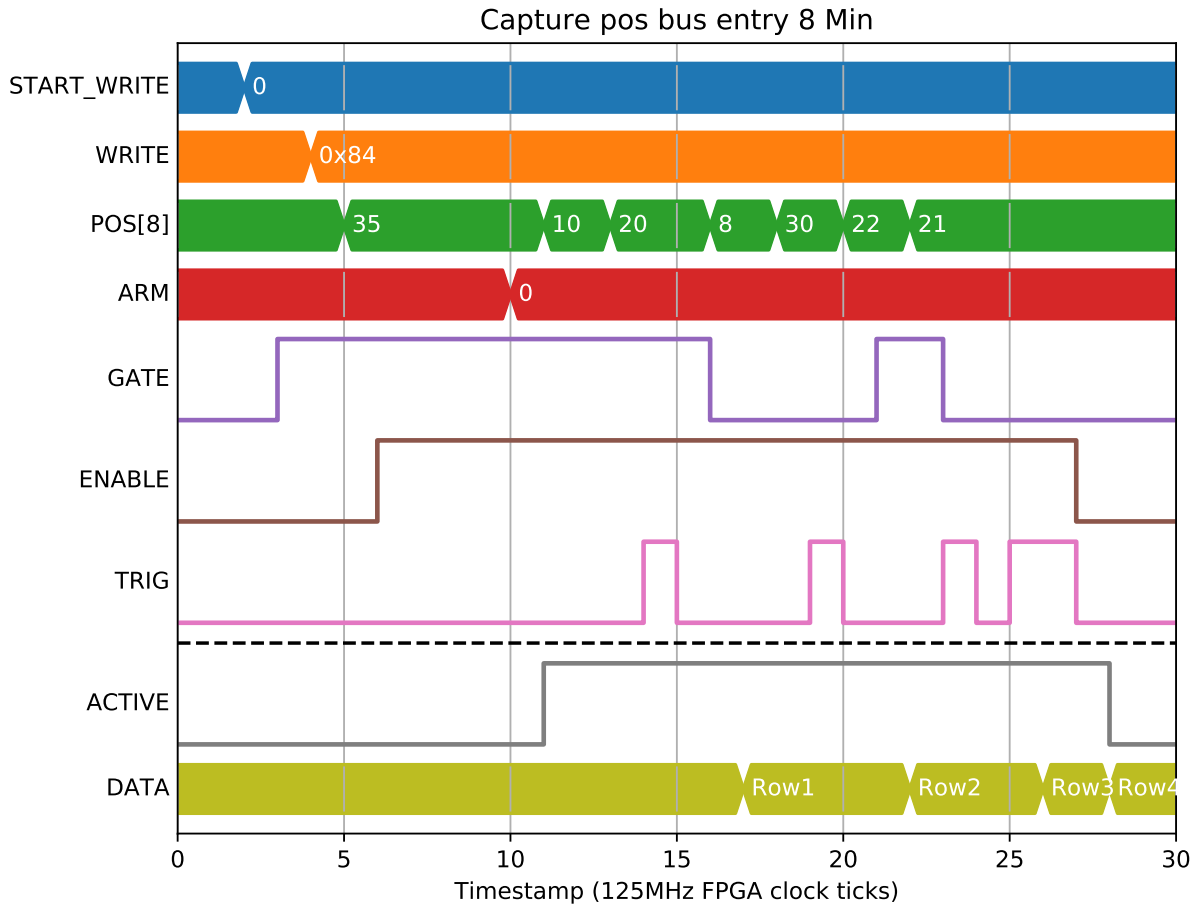


Row	0x92	0x260
0	40	1
1	36	1
2	-13	1
3	0	0

Mode 4/5 - Min/Max

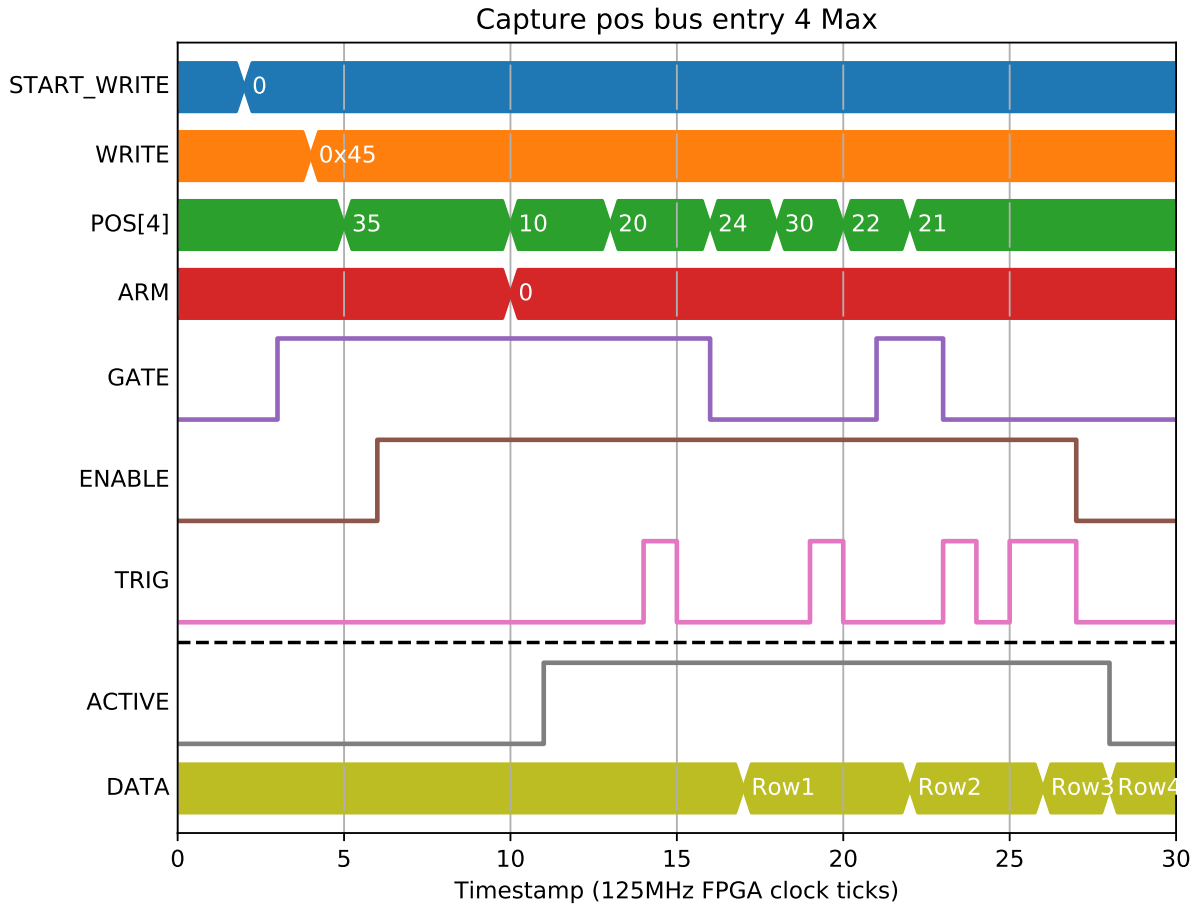
Both of these modes calculate statistics on the value while GATE is high.

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


Capture pos bus entry 8 Min
START_WRITE
WRITE
POS[8]
ARM
GATE
ENABLE
TRIG
ACTIVE
DATA
0
0x84
35
10
20
8
30
22
21
0
Row1
Row2
Row3
Row4
0
5
10
15
20
25
30
Timestamp (125MHz FPGA clock ticks)

Row	0x84
0	10
1	20
2	21
3	2147483647

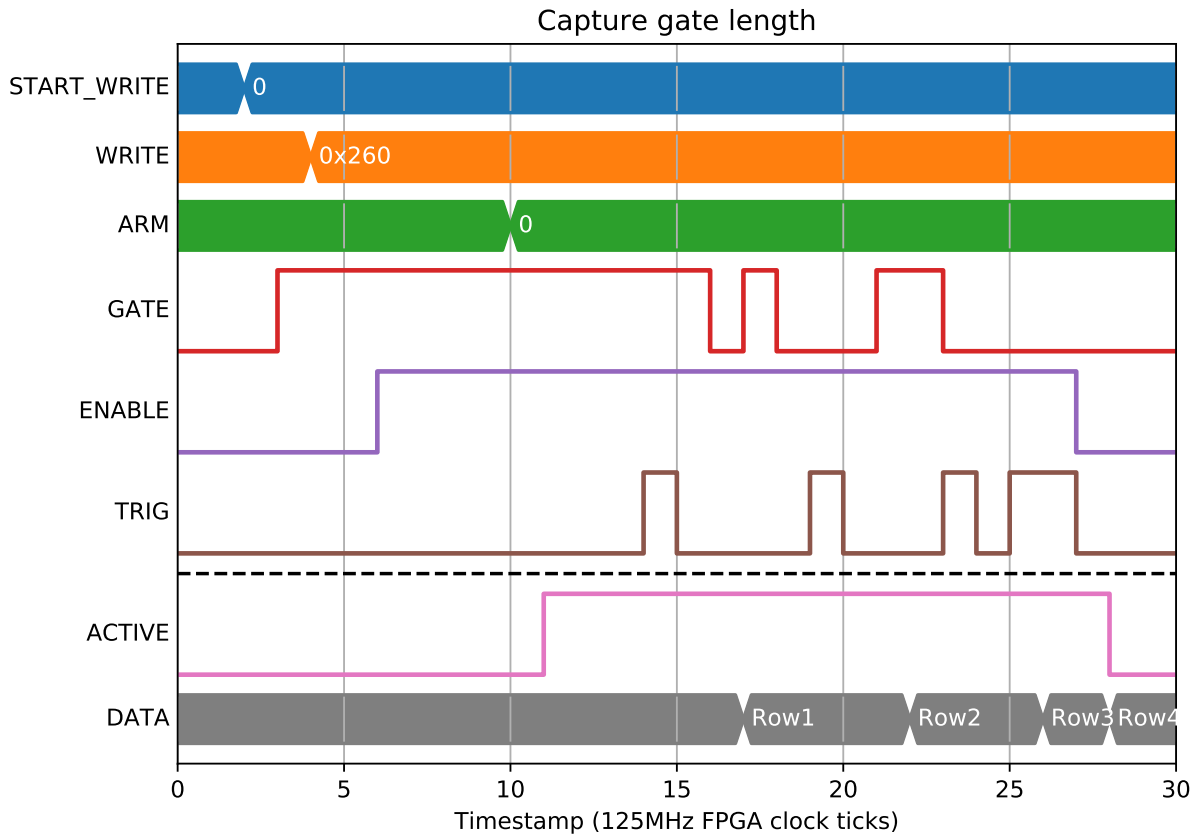
Mode 5 produces the max of all values in a similar way:



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

2.10.5 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:

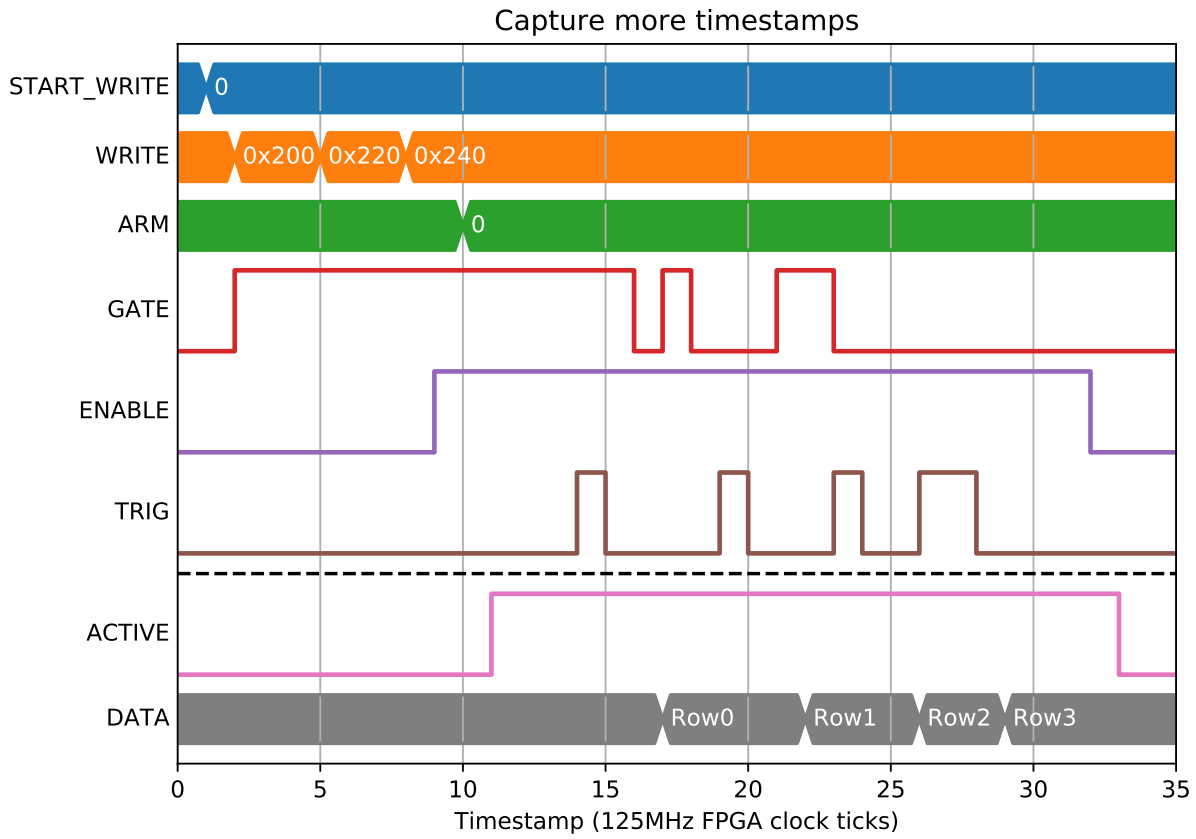


Row	0x260
0	4
1	3
2	2
3	0

2.10.6 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



2.10.7 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):

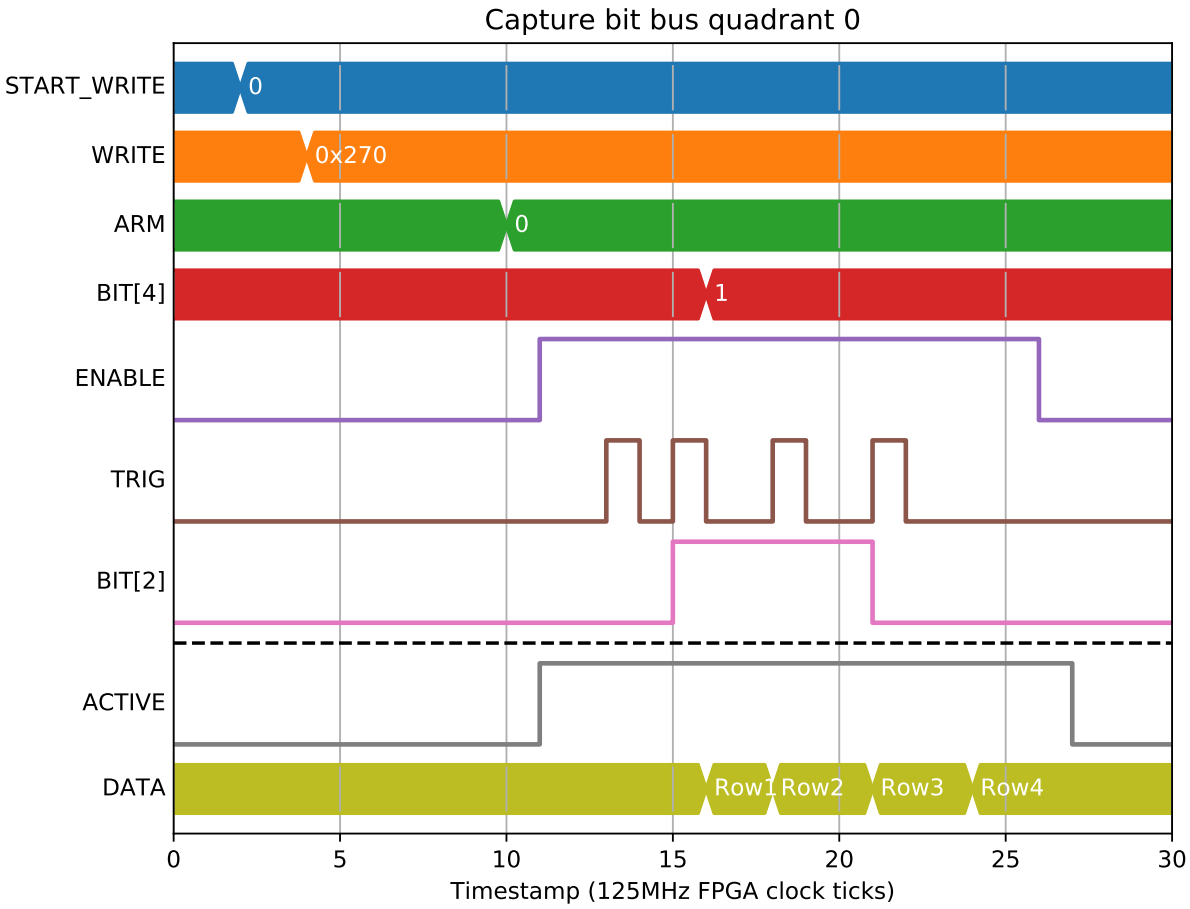
Row	0x270
0	0
1	4
2	20
3	16

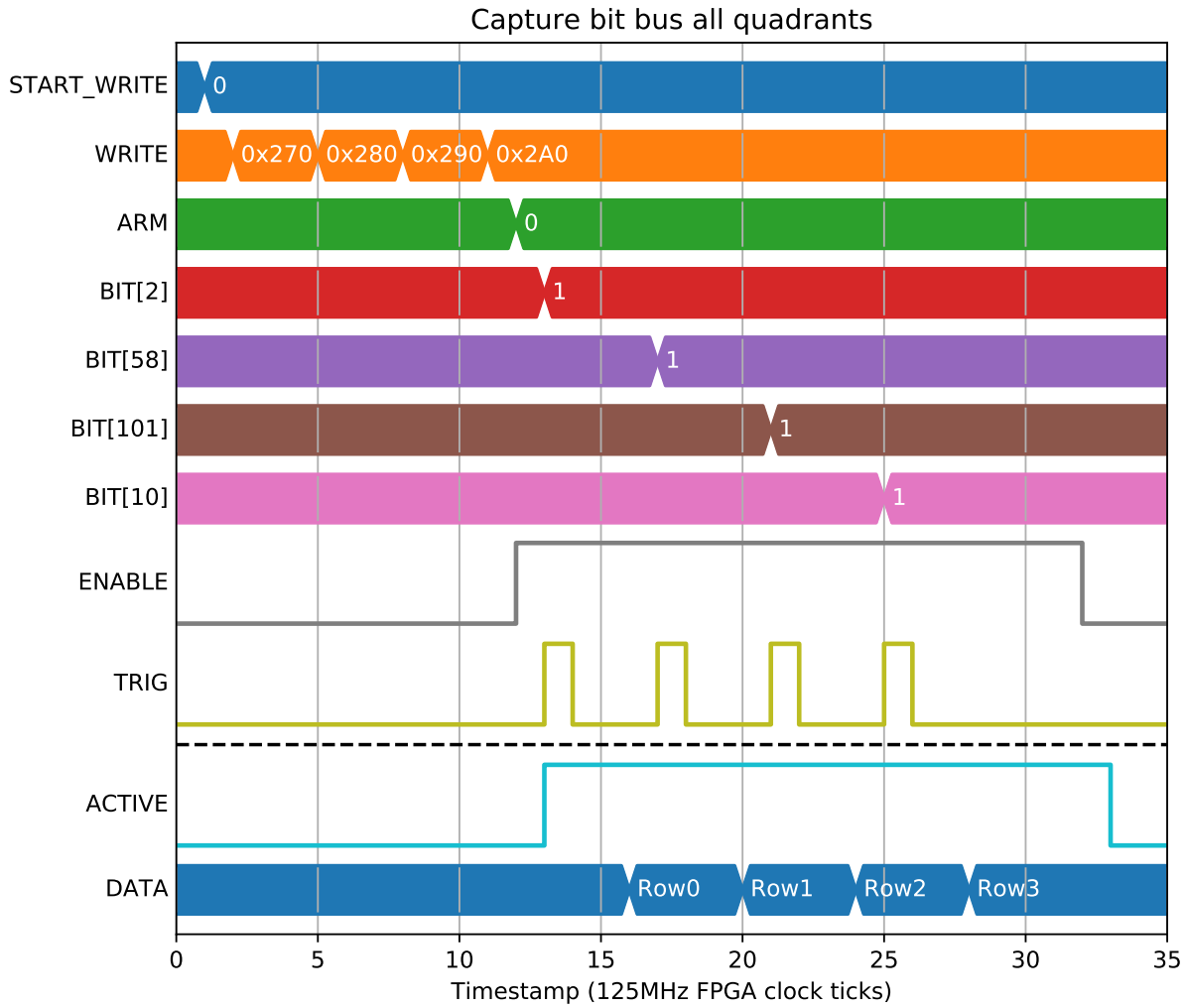
By capturing all 4 quadrants (0x27..0x2A) we get the whole bit bus:

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

2.10.8 Triggering options

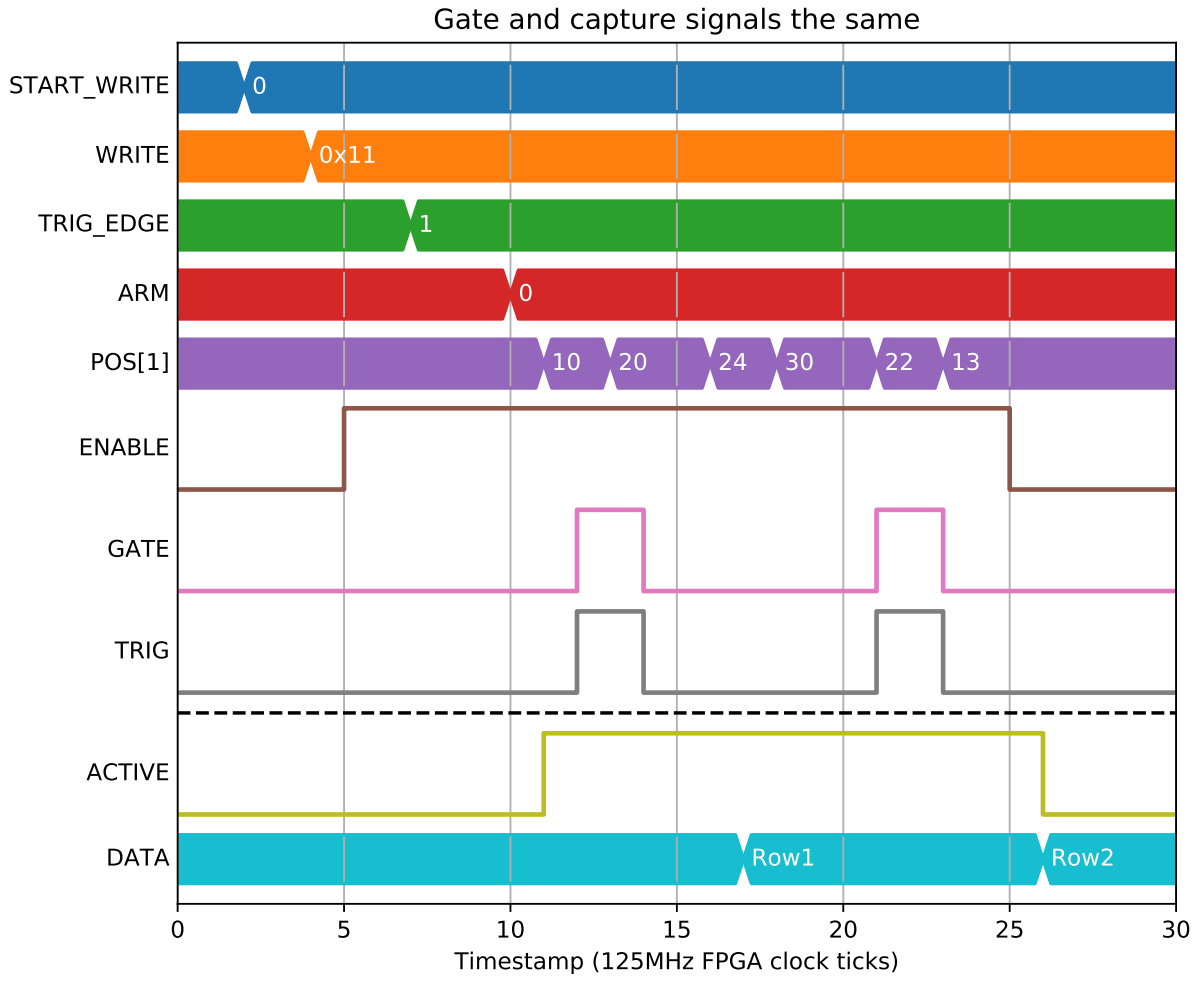
ENABLE and GATE are level triggered, with ENABLE used for marking the start and end of the entire acquisition, and GATE used to accept or reject samples within a single capture from the acquisition. CAPTURE is edge triggered


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.

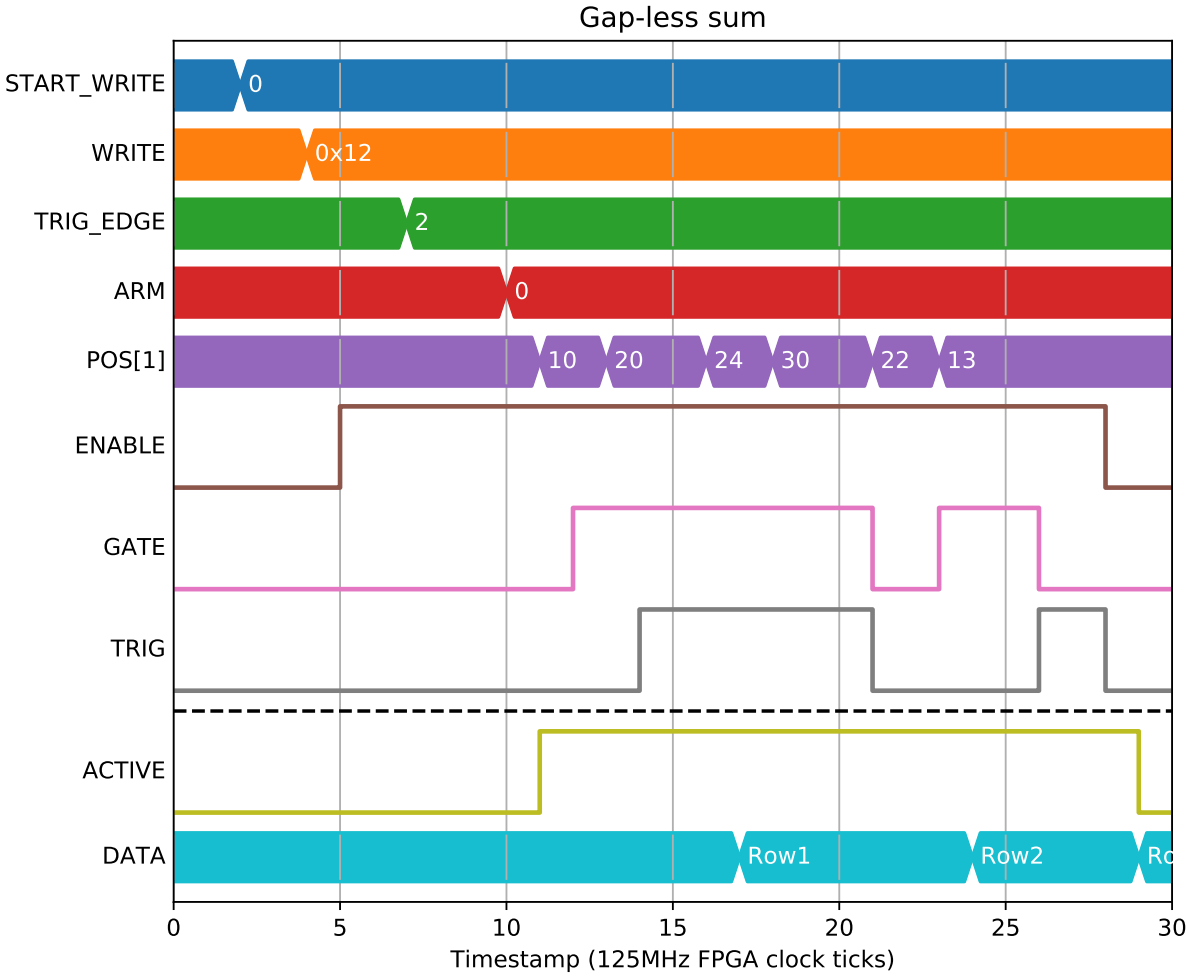
Triggering on rising is the default, explored in the preceding examples. Triggering on falling edge would be used if you have a gate signal that marks the capture boundaries and want sum or difference data within. For example, to capture the amount POS[1] changes in each capture gate we could connect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

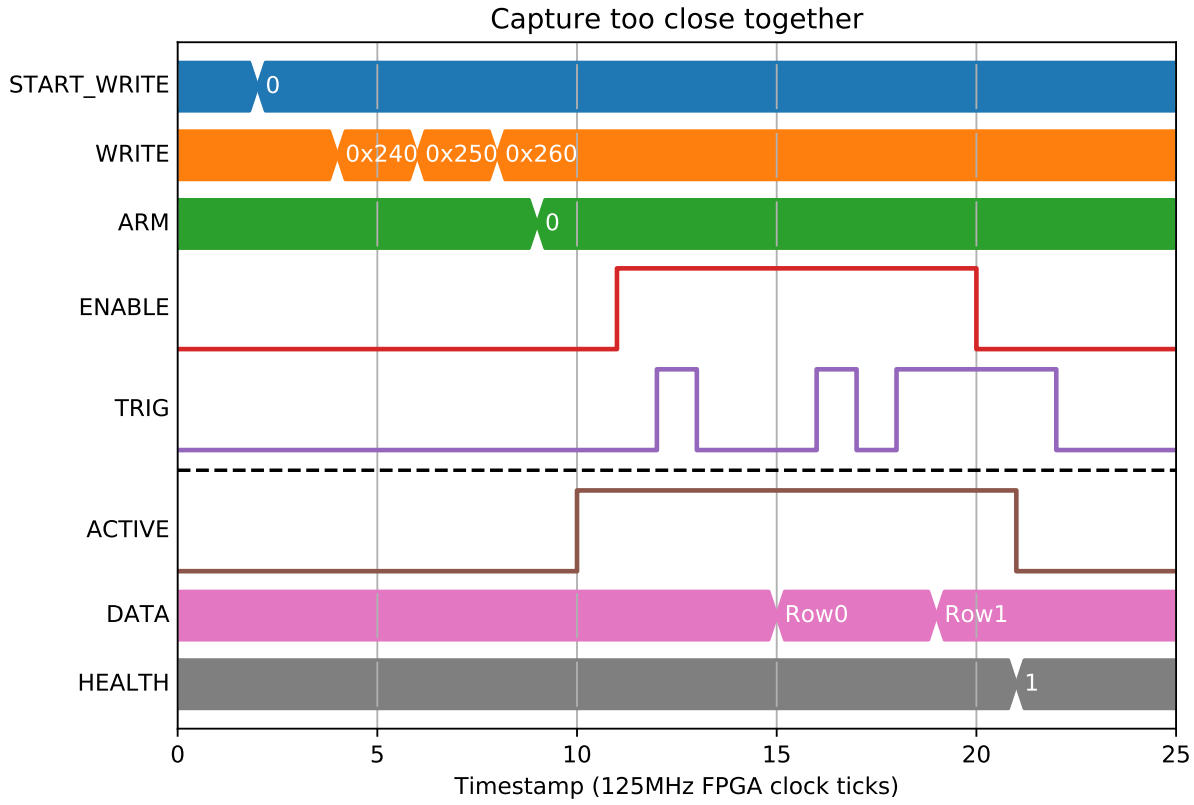
Row	0x12
0	30
1	178
2	39


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)

2.10.9 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

2.11 PGEN - Position Generator [x2]

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

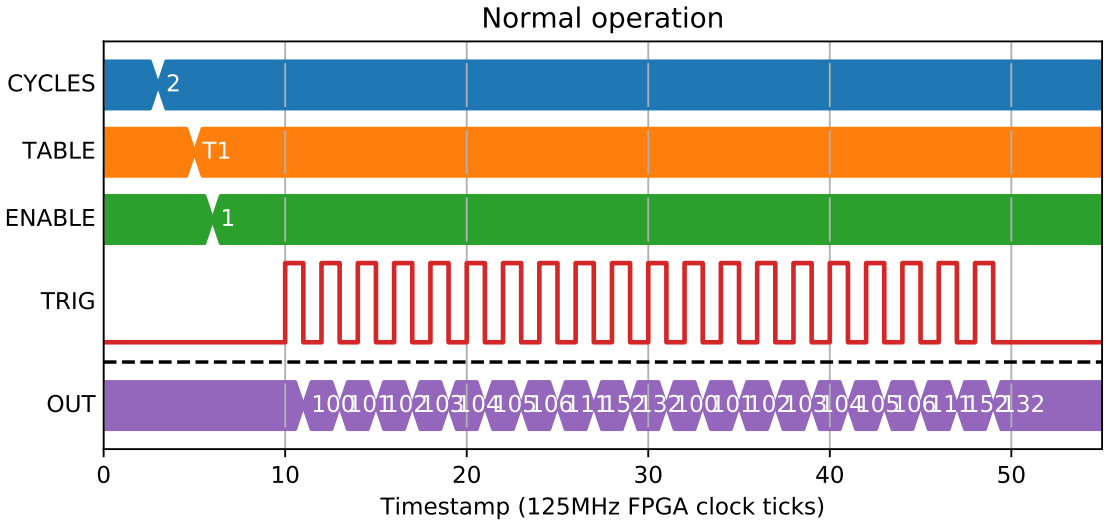
2.11.1 Parameters

Name	Dir	Type	Description
CYCLES	W	UInt32	Number of cycles
ENABLE	In	Bit	Halt on falling edge, reset and enable on rising
TRIG	In	Bit	Trigger a sample to be produced
OUT	Out	Bit	Current sample
TABLE			Table of positions to be output

2.11.2 Normal operation

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

T1
POS
100
101
102
103
104
105
106
111
152
132



2.12 INENC - Input encoder

The INENC block handles the encoder input signals

2.12.1 Parameters

Name	Dir	Type	Description
PROTOCOL	R/W	Enum	0 - Protocol type = Quadrature 1 - Protocol type = SSI 2 - Protocol type = BISS 3 - Protocol type = enDat
CLK_PERIOD	W	UInt32	Clock rate
FRAME_PERIOD	W	UInt32	Frame rate
BITS	W	UInt63	Number of bits
SETP	In	Pos	Set point
RST_ON_Z	W	Bit	Zero position on Z rising edge
STATUS	R	Enum	0 - Encoder status = All OK 1 - Encoder status = Link Down 2 - Encoder status = Encoder Error 3 - Encoder status = Link Down and Error
DCARD+MODE	R		Daughter card jumper mode
A	Out	Bit	Quadrature A if in incremental mode
B	Out	Bit	Quadrature B if in incremental mode
Z	Out	Bit	Z index channel if in incremental mode
CONN	Out	Bit	Signal detected
TRANS	Out	Bit	Position transition
VAL	Out	Pos	Current position

2.13 LVDSIN - LVDS Input

The LVDSIN block handles the signals from the LVDS Input connectors

2.13.1 Parameters

Name	Dir	Type	Description
VAL	Out	Bit	LVDS Input value

2.14 LVDSOUT - LVDS Output

The LVDSOUT block handles the signals to the LVDS Output connectors

2.14.1 Parameters

Name	Dir	Type	Description
VAL	Out	Bit	LVDS Output value

2.15 OUTENC - Output encoder

The OUTENC block handles the encoder output signals

2.15.1 Parameters

Name	Dir	Type	Description
PROTOCOL	R/W	Enum	0 - Protocol type = Quadrature 1 - Protocol type = SSI 2 - Protocol type = BISS 3 - Protocol type = enDat
BITS	W	UInt63	Number of bits
Q_PERIOD	W	UInt32	Quadrature prescaler
ENABLE	W	BIT	Halt on falling edge, reset and enable on rising
A	Out	Bit	Input for A (only straight through)
B	Out	Bit	Input for B (only straight through)
Z	Out	Bit	Input for Z (only straight through)
VAL	Out	Pos	Input for position (all other protocols)
CONN	Out	Bit	Input for connected
QSTATE	R	Enum	0 - Quadrature state = Disabled 1 - Quadrature state = At position 2 - Quadrature state = Slewing

2.16 POSENC - Quadrature and step/direction encoder

The POSENC block handles the Quadrature and step/direction encoding

2.16.1 Parameters

Name	Dir	Type	Description
INP	IN		Zero position on Z rising edge
QPERIOD	W	Pos	Set point
ENABLE	In		Halt on falling edge, reset and enable on rising
PROTOCOL	R/W	Enum	0 - Quadrature 1 - Step/Direction
A	Out	Bit	Quadrature A/Step output
B	Out	Bit	Quadrature B/Direction output
QSTATE	R	Enum	0 - Quadrature output state = Disabled 1 - Quadrature output state = At position 2 - Quadrature output state = Slewing

2.17 QDEC - Quadrature Decoder

The QDEC block handles the encoder Decoding

2.17.1 Parameters

Name	Dir	Type	Description
RST_ON_Z	W	BIT	Zero position on Z rising edge
SETP	W	Pos	Set point
A	Out	Bit	Quadrature A
B	Out	Bit	Quadrature B
Z	Out	Bit	Z index channel
OUT	Out	Pos	Output position

2.18 TTLIN - TTL Input

The TTLIN block handles the signals from the TTL Input connectors

2.18.1 Parameters

Name	Dir	Type	Description
TERM	R/W	Enum	0 - Sets input termination to High-Z 1 - Sets input termination to 50-Ohm
VAL	Out	Bit	TTL Input value

2.19 FILTER - Filter

desc...

2.19.1 Parameters

Name	Dir	Type	Description
.			

2.19.2 Difference

desc....

2.19.3 Average

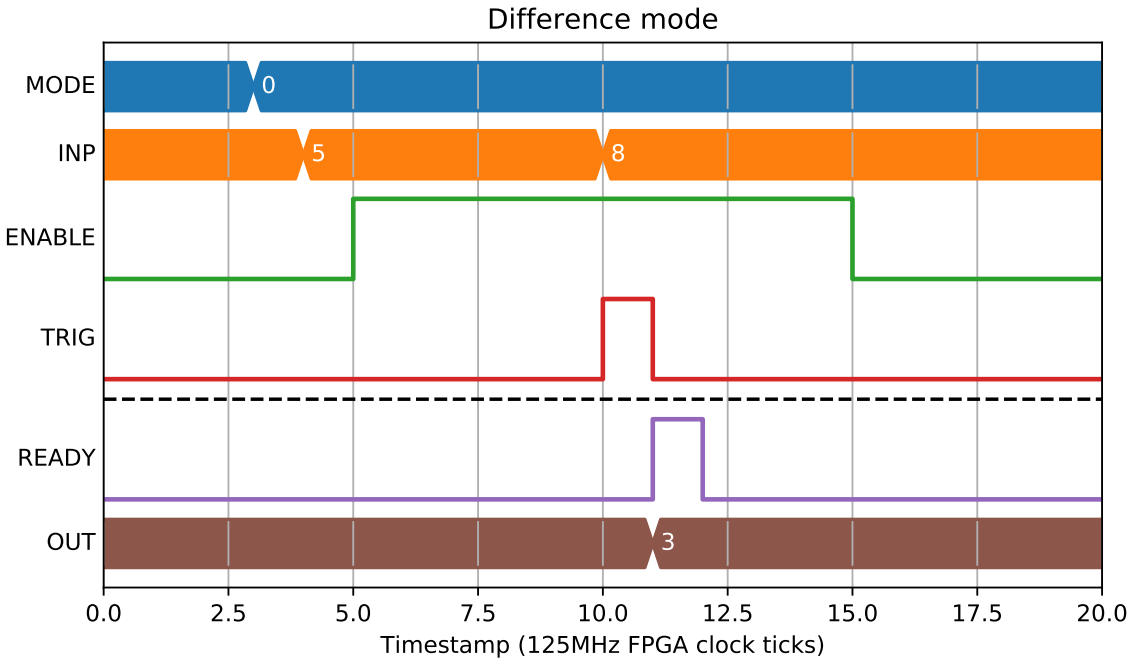
desc.....

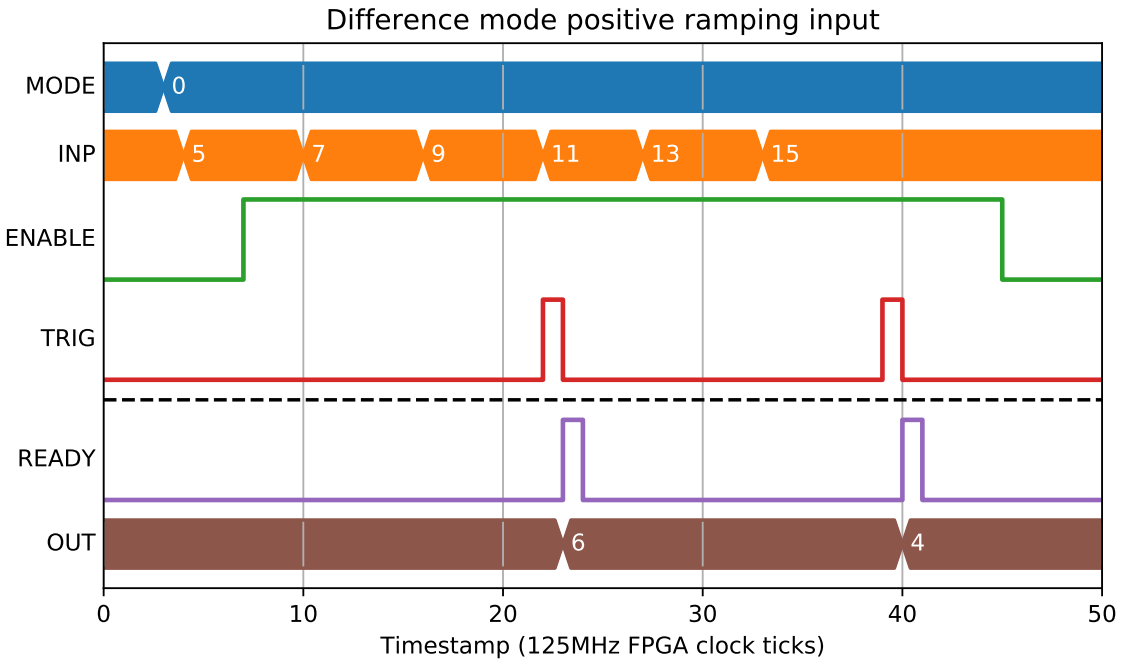
2.20 TTLOUT - TTL Output

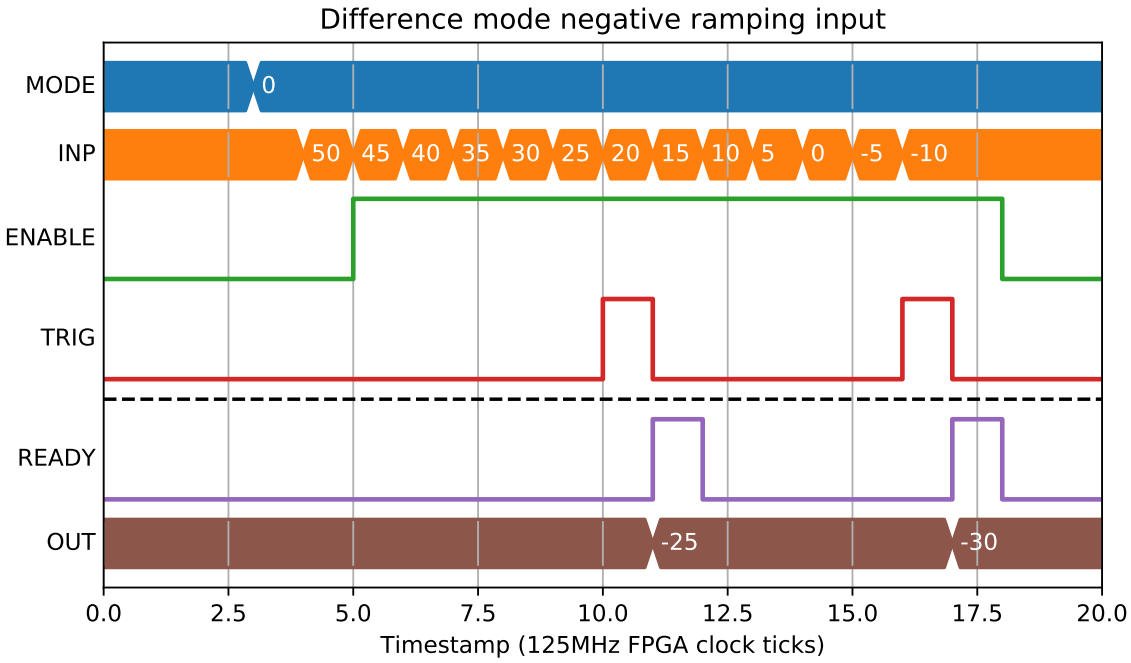
The TTLOUT block handles the signals to the TTL Output connectors

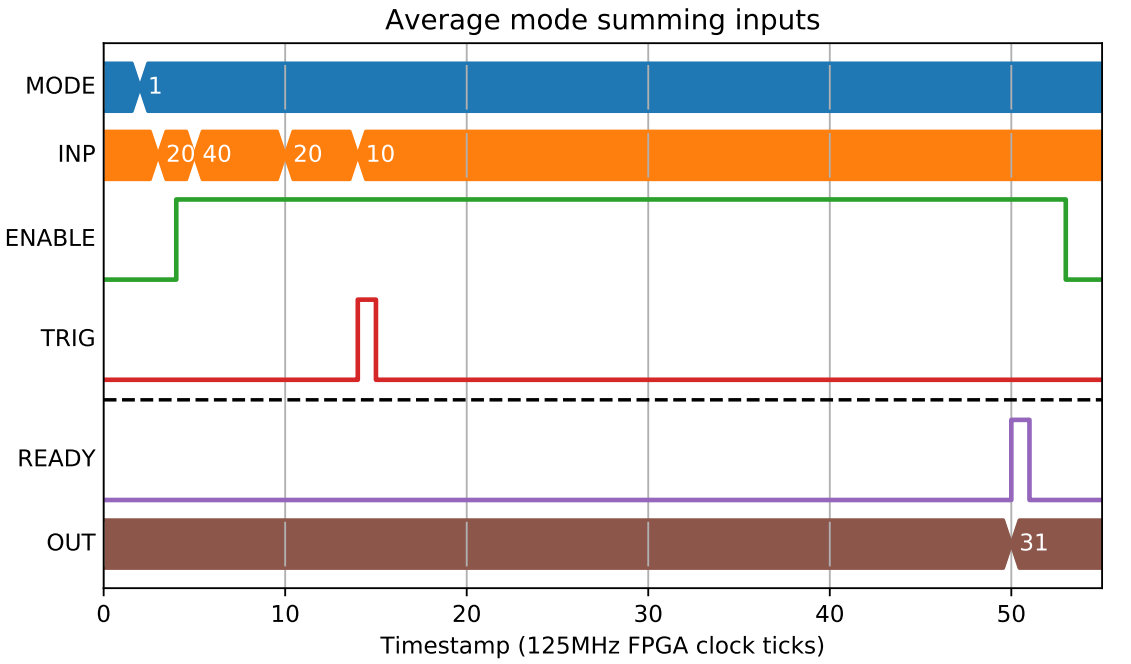
2.20.1 Parameters

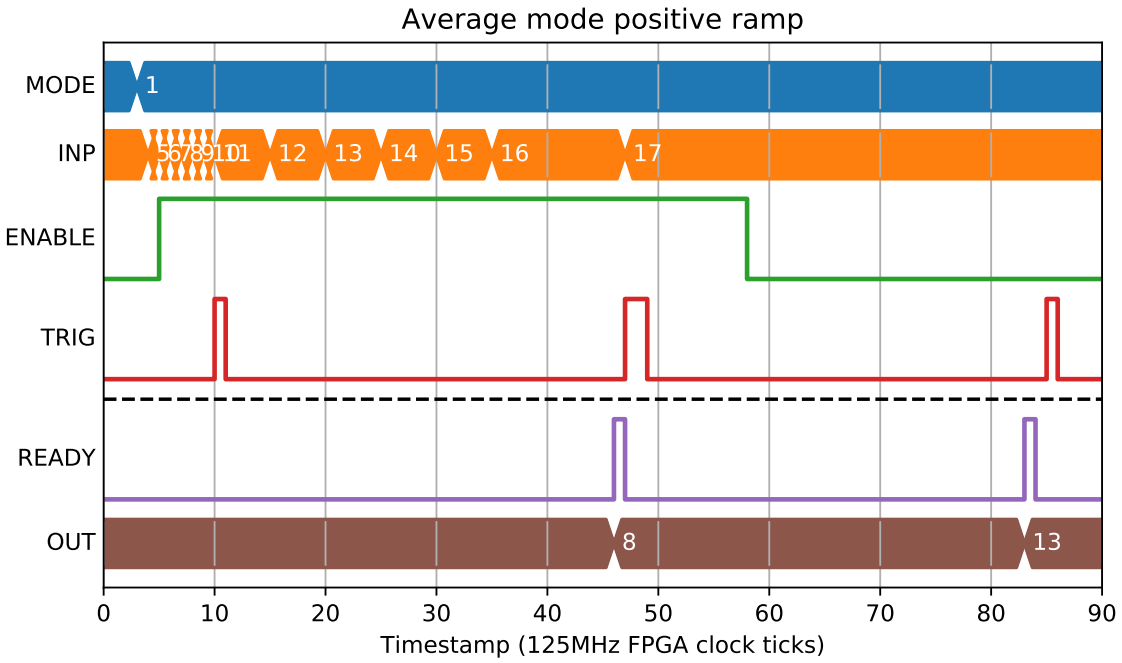
Name	Dir	Type	Description
VAL	Out	Bit	TTL Output value

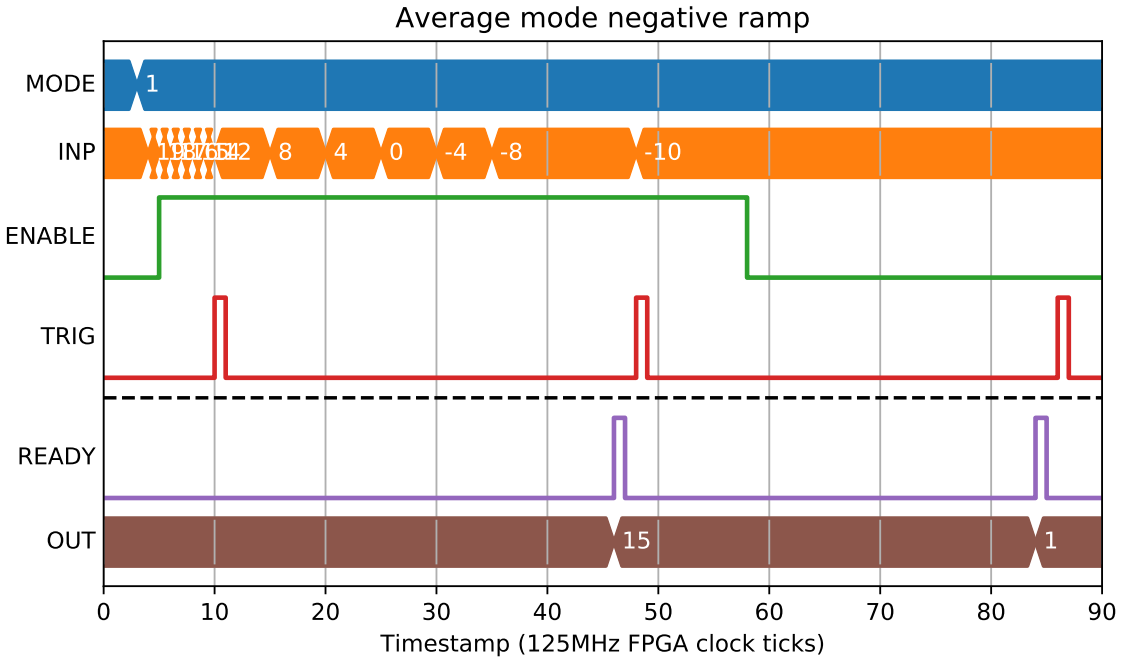

Difference mode
MODE
INP
ENABLE
TRIG
READY
OUT
0
5
8
3
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Timestamp (125MHz FPGA clock ticks)


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)

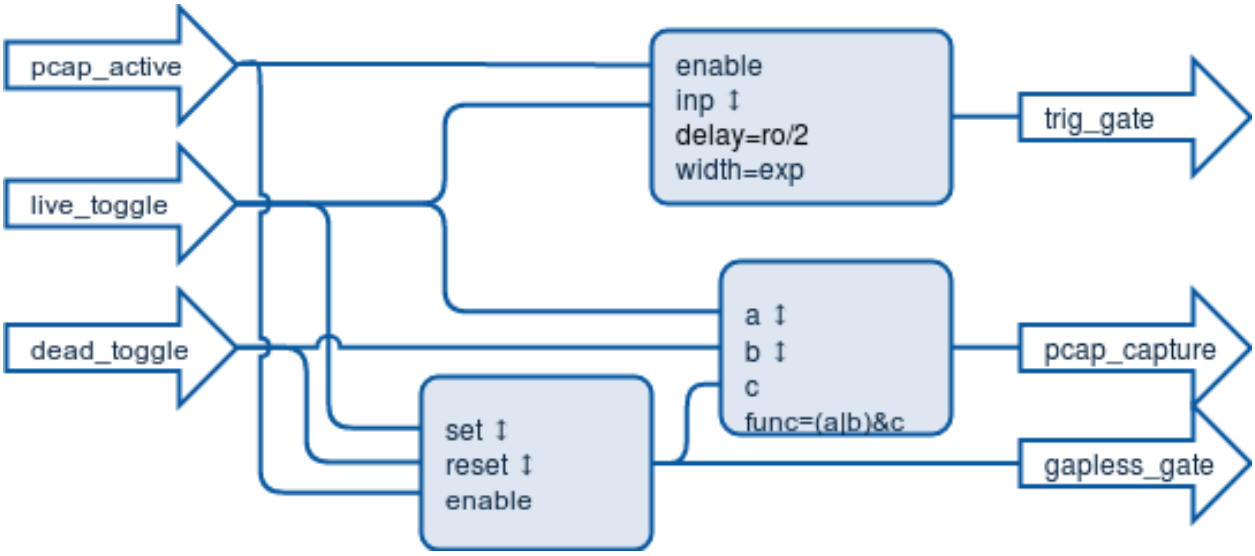

Average mode negative ramp
MODE
INP
ENABLE
TRIG
READY
OUT
1
8
4
0
-4
-8
-10
15
1
0
10
20
30
40
50
60
70
80
90
Timestamp (125MHz FPGA clock ticks)

CHAPTER 3

Triggering schemes

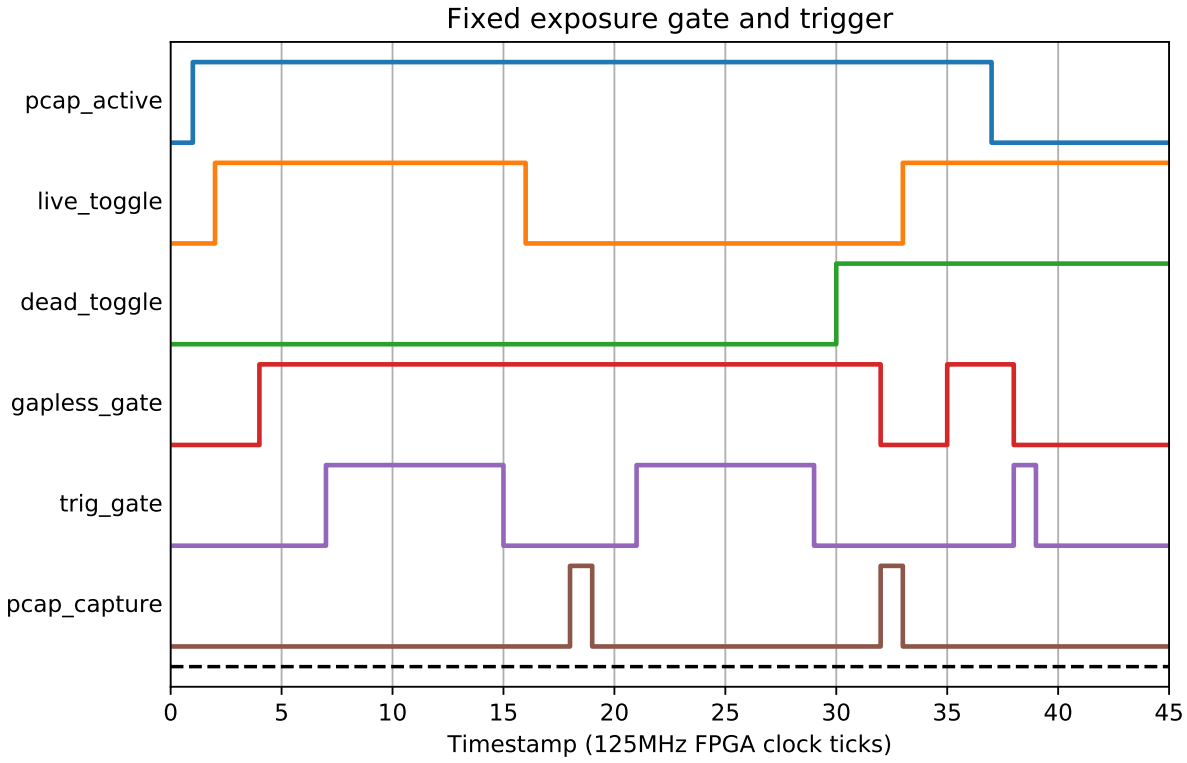
There are a number of ways that the PandA can be used with a live/dead frame signal to trigger a detector and PCAP.

3.1 Fixed exposure gate and trigger



(Edit the diagram with draw.io, opening the png file from the docs directory).

In this scheme triggers are expected to be a fixed distance apart. The live and dead signals are used in an SRGate to give a gapless gate signal while the detector is active. The LUT relies on the extra clock ticks it takes for the signal to get through the SRGate so that capture signals are generated at the end of every live frame. A number of detectors can be triggered from Pulse blocks with delay of readout/2 and width of exposure.


Fixed exposure gate and trigger
pcap_active
live_toggle
dead_toggle
gapless_gate
trig_gate
pcap_capture
0
5
10
15
20
25
30
35
40
45
Timestamp (125MHz FPGA clock ticks)

CHAPTER 4

Unit testing FPGA blocks

In order to capture block level race conditions and document behaviour at the clock tick level, there is a mechanism for unit testing FPGA blocks. It consists of a number of parts:

- A Python simulation of the block (`simulation/sim_zebra2/<block>.py`)
- A number of unit test sequences (`tests/sim_zebra2_sequences/<block>.seq`)
- A Python test runner (`tests/test_sim_zebra2.py`)
- The generated FPGA test vectors (`tests/fpga_sequences/<block>_*.txt`)
- The block documentation plots (`docs/blocks/<block>.rst`)

The Python simulation models the expected behaviour of the block by returning the expected outputs from a given set of inputs. Test sequences list specific sets of inputs at specific FPGA clock tick numbers, and list the expected outputs. The test runner scans the sequence files and provides the given inputs to the Python simulation, checking the responses agains the expected outputs. If this succeeds, it produces FPGA test vectors for all sequences merged together. There is also a tool that generates plots for the documentation from named sequences, so that any examples in the documentation have been tested to be correct.

4.1 Python block simulation

Each block simulation should inherit from the Block class, implementing the on_event(event) function that returns next_event:

- event contains the register, bit bus, position bus, and external signal changes that take place at a given FPGA timestamp.
- next_event contains the changes that take place as a result of event, and an FPGA timestamp when the block next needs to be called

The base class will initialiase attributes on the class for each parameter, and initialise bit_out and pos_out values to be their position on the relevant bus.

4.2 Unit test sequences

A test sequence consists of the following grammar:

```
sequence-list = sequence*
     sequence = header event*
       header = "$" [mark] title
         mark = "!"
        event = ts ":" changes [":" changes]
      changes = "" | assignment ["," assignment]*
   assignment = name "=" value
```

Where:

- `title` is a string describing the sequence
- `ts` is the integer FPGA clock tick
- `name` is the string name of the register or signal
- `value` is the integer value of that register or signal

Empty lines and lines starting with `#` are ignored.

For example:

```
#########################
$ Pulse delay and stretch
1       : WIDTH=10
2       : DELAY=10
7       : INP=1          : QUEUE=2
8       : INP=0
17      :                : QUEUE=1, OUT=1
27      :                : QUEUE=0, OUT=0
```

This says:

- At FPGA clock tick 1, set reg WIDTH=10, expect no changes (apart from this register set operation)
- At tick 2, set reg DELAY=10, expect no changes
- At tick 7, set signal INP=1, expect reg QUEUE to be 2
- At tick 8, set signal INP=0, expect no changes
- At tick 17, don't set anything, expect reg QUEUE to be 1, and signal OUT to be 1
- At tick 27, don't set anything, expect reg QUEUE to be 0, and signal OUT to be 0

4.3 Running the test

You can invoke the test runner by doing:

```
python tests/test_sim_zebra2.py
```

This will then search for all `<block>.seq` files, and scan them. It will build a sequence for each one found in the file, adding one called "All" that contains all of them one after another, and will be used to generate the FPGA test vectors.

If a test title starts with "$!" instead of just "$", then it will be marked, and only the marked tests will be run. No FPGA test vectors will be generated. This is used for running just one test while debugging the Python simulation.

4.4 The generated FPGA test vectors

When the "All" test has completed successfully, the following files will exist in `tests/fpga_sequences/`:

- `<block>_bus_in.txt`: The bit and position bus inputs at each clock tick.
- `<block>_reg_in.txt`: The registers that should be set at each clock tick.
- `<block>_bus_out.txt`: The expected bit and position bus outputs at each clock tick. Note that these are 1 tick after the inputs.
- `<block>_reg_out.txt`: The expected register values at each clock tick. Again, note that these are 1 tick after the inputs.

4.5 Running the FPGA test vectors

There is a new firmware commit. This includes the first block that I want you to simulate. It is the panda_pulse.vhd.

All the test stimulus vector files (generated by Tom's framework) are already copied in the sim/panda_pulse/do directory.

The testbench is located in sim/panda_pulse/, and it is called panda_pulse_tb.v. Yes, I did use Verilog for the testbench because the file I/O is much easier.

To run the simulations:

1. First step, you will need to re-run "build_ips.tcl" to generate the required IP for this block.
2. Run compile.do under sim/panda_pulse/do directory, and observe the windows.

As you will this, the module passes the test and does not report any error between its outputs and expected outputs.

4.6 Generating the plots for the block level documentation

In `docs/block_plot.py` there is a function `make_block_plot(block, title)` that will generate a plot of a given sequence. You can embed this plot into the block level documentation by writing the following directive:

```
.. plot::

    from common.python.block_plot import make_block_plot
    make_block_plot("<block>", "<title>")
```

For instance:

```
.. plot::

    from common.python.block_plot import make_block_plot
    make_block_plot("pulse", "Pulse stretching with no delay")
```

CHAPTER 5

API doc for configparser

class common.python.pandablocks.configparser.**ConfigParser**(*config_dir*)

Parser for config/register/description file

Will populate itself with the blocks and fields described in the config files, checking for validity

Variables

- **blocks** (*OrderedDict*) – map str block_name -> *ConfigBlock* instance where block name doesn't include number (e.g. "SEQ")
- **bit_bus** (*OrderedDict*) – map str block_name.field_name -> int bit_bus_idx for each block and field
- **pos_bus** (*OrderedDict*) – map str block_name.field_name -> int pos_bus_idx for each block and field
- **ext_names** (*OrderedDict*) – map int ext_bus_idx -> str block_name.field_name for each block and field

Populate parser with files from config_dir

Parameters **config_dir** (*str*) – Path to config directory

class common.python.pandablocks.configparser.**ConfigBlock**(*reg_line*, *config_line=None*, *desc_line=None*)

Represents a block definition in the config file.

Variables

- **name** (*str*) – The block name (e.g. PULSE)
- **num** (*int*) – The number of blocks that should be created (e.g. 2)
- **base** (*int*) – The base register offset for this block
- **desc** (*str*) – The description for this block
- **fields** (*OrderedDict*) – map str field_name -> *ConfigField* instance for each field the block has

- **registers** (*OrderedDict*) – map str attr_name -> (int reg num, ConfigField)
- **outputs** (*OrderedDict*) – map str attr_name -> ([int out idx], ConfigField)

Also, there will be an attribute for each attr_name in registers.keys() that also has that string as its value. This will allow lookup of register strings in a safe way. For example:

self.TABLE_DATA = "TABLE_DATA"

Initialise with relevant config/reg/desc lines for this block.

Should include block definition and all field definitions for this block

Parameters

- **reg_line** (*str*) – Line specifying block in registers file
- **config_line** (*str*) – Optional line specifying block in config file
- **desc_line** (*str*) – Optional line specifying block in descriptions file

add_field(*field*)

Add a ConfigField instance to self.fields dictionary

This also sets an attribute on itself so we can do safer lookups. E.g. self.FORCE_RST = "FORCE_RST"

Parameters **field** (ConfigField) – ConfigField instance

class common.python.pandablocks.configparser.**ConfigField**(*name*, *reg_lines*, *config_lines=None*, *desc_lines=None*)

Represents a field of a field definition.

The information held here spans the config, description and register files

Variables

- **name** (*str*) – The field name (e.g. OUTD)
- **reg** (*list*) – The register string (e.g. ["3", "2", ">3"])
- **cls** (*str*) – The field class (e.g. pos_out)
- **cls_args** (*list*) – The arguments needed to configure the cls (e.g. ["encoder"])
- **cls_extra** (*list*) – Any extra data associated with cls (e.g. enum values ["0 Falling", "1 Rising"])
- **desc** (*str*) – The description of the field

Initialise with relevant config/reg/desc lines for this field

Parameters

- **reg_lines** (*list*) – Lines specifying field in registers file
- **config_lines** (*list*) – Optional lines specifying field in config file
- **desc_lines** (*list*) – Optional line specifying field in descriptions
- **file** –

Index

A

C