RedPitaya - Timing Module

Firmware

Data types

Definition: *boolean* = 1bit logical, *int* = 4byte integer, *long* = 8byte integer.
Using *int* as data type for DELAY and SEQUENCE limits the maximal duration to little more than 3.5 minutes. Considering quasi continuous pulses, *long* is used. Since some diagnostics require arbitrarily distributed clocks the maximum number of elements the Sequence vector should be maxed out by the hardware constrains but at least 1M samples (8 MB, using DRAM).

private parameters - registers

address	name	type	description
0x0000	INIT	boolean	Used to deinit() the device and break the loops in armed() and program().
0x0008	DELAY	long	Number of tick between the input trigger and the beginning of the first sequence.
0x0010	WIDTH	int	Number of ticks the output remains high when raised.
0x0014	PERIOD	int	Minimum number of ticks between two raising edges:
			1. defines the rate of a pulse train.
			2. defines when the gate closes after the last pulse.
0x0018	CYCLE	long	Number of ticks between two sequence starts (c.f. PERIOD but for the sequence).
0x0020	REPEAT	int	Number of sequence repetitions.
0x0024	COUNT	int	Number of pulses in a sequence or pulse train.
0x0028	SEQUENCE	long[]	Increasing number of tick counts since the sequence start to the raising edge of a pulse.

LED and DIO connections

bank	0	1	2	3	4	5	6	7
led	*trig*	clk	sig	gate	armed	reinit	ext_clk	error
p(inner)	**trig**	**clk**	ch0	ch1	ch2	ch3	ch4	ch5
n(outer)	gnd	gnd	gnd	gnd	gnd	gnd	gnd	gnd

input, output

Channels ch0-ch5 can be inverted or switched to gate using the invert() and gate() method, respectively.

Startup states

INIT is cleared, all parameters are set to 0/null, i.e. the whole register is cleared. However, it is satisfactory to ensure the clearing of the first 30 bytes. All outputs should be on low. Unused digital pins should be configured as input.

Parameter contrains/defaults

These constrains should be checked against by the driver before programming.

name	constrain	default	default*
DELAY	≥ 0	0	$60,000,000$
WIDTH	≥ 1	WIDTH/2	5
PERIOD	$>$ WIDTH	10	10
CYCLE	$\geq$ COUNT $*$ PERIOD	COUNT $*$ PERIOD	0
	$\geq$ SEQUENCE[end] $+$ PERIOD	SEQUENCE[end] $+$ PERIOD	-
REPEAT	≥ 0	1	0
COUNT	≥ 0	1	0
	$\geq 0, \quad \leq$ len(SEQUENCE)	len(SEQUENCE)	-

*) default value if nothing or only DELAY is provided, only valid for makeClock.

Network packages

The body of makeSequence() and makeClock() could be copied to the register directly as byte stream.

address	name	size	description
0x0000	magic	3bytes	Magic bytes, e.g. $0x573758$ ='W7X'
0x0003	method	1byte	Method identifier, i.e. deinit: 0x00, init1: 0x01, init2: 0x02
0x0004	length	4bytes	Length of argument body, i.e. deinit: 0, init2: 40, init1: 40 + 8 count
0x0008	delay	8bytes	1nd argument
0x0010	width	8bytes	2nd argument
0x0014	period	8bytes	3rd argument
0x0018	cycle	8bytes	4th argument
0x0020	repeat	4bytes	5th argument
0x0024	count	4bytes	7th argument
0x0028	sequence[0]	8bytes	1st element of sequence argument (only for init1)
...			

The remote programming is currently available via mdsip with provided tdi functions.

Network interrupts (public methods)

arm (method = 0x00)

INIT is set. This will cause the program to react on trig events.

disarm (method = 0x00)

INIT is cleared. This will interrupt the loops in armed() and program() if running. The device is idling until the next init.

makeSequence (method = 0x01)

The parameters DELAY, WIDTH, PERIOD, CYCLE, REPEAT, COUNT, and SEQUENCE are transmitted. INIT is set. The device is armed().

makeClock (method = 0x02)

The parameters DELAY, WIDTH, PERIOD, CYCLE, REPEAT, and COUNT are transmitted. SEQUENCE is generated from PERIOD and COUNT. INIT is set. The device is armed().

private methods

armed

The procedure armed() calls getTrigger() until it returns TRUE, in which case armed() executes program(). The device will remain armed until a deinit interrupt.

program

The method program() generates the trigger and gate signal. It is called by armed() when once trigger has occurred.

advance

The method advance(T,G) synchronizes the code. If INIT is set the output channels are set to T and G. Otherwise, they are both cleared. It returns TRUE if INIT was not set.

abstract methods

getTrigger

The method getTrigger() checks the TRIGGER register and returns TRUE if the trigger level has been high.

setOutput

The method setOutput(T,G) sets the state of trigger_out and gate_out to T and G, respectively.

sync

The method sync() synchronizes the code to the with the external clock (raising/falling edge).

Timing

The board cycle is synchronized to an external clock of 10Mhz. The most important aspect is that the jitter is minimized. Fixed delays of a few ticks (< 10 maybe) are acceptable. The most important part is that the timestamps of the TTLs can be calculate with little uncertainty based on the external clock.

Trigger schema

At W7X there will be a trigger $T_0 = -60$s before the experiment starts at $T_1 = 0$s. This trigger will engage the initialization of all diagnostics as well as the timing module. Since the timing module should support output signals even seconds before T_1 it is required to trigger the timing module with T_0. The triggered module should enter the delay phase which should be set to 60s. During the delay phase the module should accept an reconfiguration (makeClock, makeSequence) without loosing track of the tick count with respect to T_0. The updated delay and timing configuration should be used by the device on-the-fly. If the device is not configured before the end or delay, it should return to armed state (i.e. if count=0 or repeat=0).

Program

TIME_WIDTH $\geq$ 40

```
architecture Behavioral of w7x_timing is
   -- summary of valid states                                      SG0PWA
   constant IDLE            : std_logic_vector(0 to 5) := "000000";
   constant ARMED           : std_logic_vector(0 to 5) := "000001";
   constant WAITING_DELAY   : std_logic_vector(0 to 5) := "000010";
   constant WAITING_SAMPLE  : std_logic_vector(0 to 5) := "010110";
   constant WAITING_LOW     : std_logic_vector(0 to 5) := "010010";
   constant WAITING_HIGH    : std_logic_vector(0 to 5) := "110010";
   constant WAITING_REPEAT  : std_logic_vector(0 to 5) := "000110";
   constant ERROR           : std_logic_vector(0 to 5) := "000111";
               -- -- signals
   signal   state           : std_logic_vector(0 to 5) := IDLE;
   -- measure number of samples in sequence, i.e. len(times)
   signal reset        : std_logic := '1'; -- start_cycle    =0, do_waiting_sample ++
   signal sample_count : integer := 0; -- start_cycle    =0, do_waiting_sample ++
   signal sample_total : integer := 0;
   -- measure number of repetitions
   signal repeat_count : integer := 0; -- start_program =0, start_waiting_repeat ++
   signal repeat_total : integer := 0;
   -- measure high and low of signal
   signal period_ticks : unsigned(31 downto 0) := (others => '0');
   signal high_total   : unsigned(31 downto 0) := (others => '0');
   signal period_total : unsigned(31 downto 0) := (others => '0');
   -- sequence counter
   signal cycle_ticks  : unsigned(TIME_WIDTH-1 downto 0) := (others => '0');
   signal delay_total  : unsigned(TIME_WIDTH-1 downto 0) := (others => '0');
   signal cycle_total  : unsigned(TIME_WIDTH-1 downto 0) := (others => '0');
   -- contains the next sample of interrest defined by index_out of the previous cycle
   signal curr_sample  : unsigned(TIME_WIDTH-1 downto 0) := (others => '0');
begin
   -- set input
   delay_total  <= unsigned(delay (TIME_WIDTH-1 downto 0));
   high_total   <= unsigned(width);
   period_total <= unsigned(period);
   cycle_total  <= unsigned(cycle (TIME_WIDTH-1 downto 0));
   repeat_total <= to_integer(unsigned(repeat));
   sample_total <= to_integer(unsigned(count));
   curr_sample  <= unsigned(sample(TIME_WIDTH-1 downto 0));
   state_out    <= state;

  clock_gen:  process(clk_in, init_in, trig_in,
                      delay_total,
                      period_ticks, high_total, period_total,
                      cycle_ticks, cycle_total,
                      repeat_total,
                      sample_total, sample_count, curr_sample) is

   procedure inc_cycle is
   begin
     cycle_ticks <= cycle_ticks  + 1;
   end inc_cycle;

   procedure inc_period is
   begin
     period_ticks <= period_ticks + 1;
     inc_cycle;
   end inc_period;
```

```
procedure start_sample is
-- resets period_ticks (1)
begin
  state <= WAITING_HIGH;
  period_ticks <= (0=> '1', others => '0');
end start_sample;

procedure do_waiting_sample is
-- increments sample_count
begin
  if cycle_ticks = curr_sample then
    sample_count <= sample_count + 1;
    start_sample;
    inc_cycle;
  elsif cycle_ticks > curr_sample then
    state <= ERROR;
  else
    state <= WAITING_SAMPLE;
    inc_cycle;
  end if;
end do_waiting_sample;

procedure start_cycle is
-- resets sample_count (1:0)
-- resets cycle_ticks  (1)
begin
  if curr_sample = 0 then -- short cut if first sample is at 0
    sample_count <= 1;
    start_sample;
  else
    sample_count <= 0;
    period_ticks <= (others => '0');
    state <= WAITING_SAMPLE;
  end if;
  cycle_ticks <= (0=> '1', others => '0');
end start_cycle;

procedure do_waiting_repeat is
begin
  if cycle_ticks = cycle_total then -- short cut if cycle_total just fits sequence
    start_cycle;
  elsif cycle_ticks > cycle_total then
    state <= ERROR;
  else
    state <= WAITING_REPEAT;
    inc_cycle;
  end if;
end do_waiting_repeat;

procedure start_armed is
-- resets everything
begin
  cycle_ticks  <= (others => '0');
  period_ticks <= (others => '0');
  sample_count <= 0;
  repeat_count <= 0;
  state <= ARMED;
end start_armed;

procedure start_waiting_repeat is
-- increments repeat_count
```

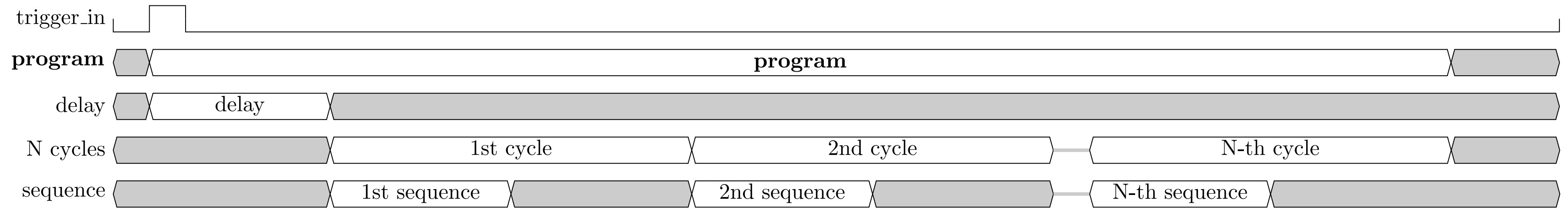
```
begin
  if repeat_count < repeat_total then
    repeat_count <= repeat_count + 1;
    do_waiting_repeat;
  else
    start_armed;
  end if;
end start_waiting_repeat;

procedure start_waiting_sample is
begin
  if sample_count < sample_total then
    do_waiting_sample;
  else
    start_waiting_repeat;
  end if;
end start_waiting_sample;

procedure do_waiting_low is
begin
  if period_ticks = period_total then
    start_waiting_sample;
  elsif period_ticks > period_total then
    state <= ERROR;
  else
    state <= WAITING_LOW;
    inc_period;
  end if;
end do_waiting_low;

procedure do_waiting_high is
begin
  if period_ticks = high_total then
    state <= WAITING_LOW;
    inc_period;
  elsif period_ticks > high_total then
    state <= ERROR;
  else
    state <= WAITING_HIGH;
    inc_period;
  end if;
end do_waiting_high;

procedure do_waiting_delay is
begin
  if cycle_ticks = delay_total then
    if repeat_total > 0 and sample_total > 0 then
      start_cycle;
    else
      start_armed;
    end if;
  elsif cycle_ticks > delay_total then
    state <= ERROR;
  else
    state <= WAITING_DELAY;
    inc_cycle;
  end if;
end do_waiting_delay;

procedure start_program is
-- resets repeat_count (1)
```

```
    -- resets cycle_ticks (_:1)
    begin
      repeat_count <= 1;
      if delay_total = 0 then -- short cut if no delay
        start_cycle;
      else
        state <= WAITING_DELAY;
        cycle_ticks <= (0=> '1', others => '0');
       end if;
    end start_program;

  begin  -- main program

    if init_in = '0' then
      state <= IDLE;
    elsif rising_edge(clk_in) then
      case state is
        when ARMED =>
          if trig_in = '1' then
            start_program;
          end if;
        when WAITING_DELAY =>
          do_waiting_delay;
        when WAITING_SAMPLE =>
          do_waiting_sample;
        when WAITING_HIGH =>
          do_waiting_high;
        when WAITING_LOW =>
          do_waiting_low;
        when WAITING_REPEAT =>
          do_waiting_repeat;
        when IDLE =>
          start_armed;
        when others =>
          state <= ERROR;
      end case;
    end if; -- rising_edge(clk)
    if sample_count<sample_total then
      index_out <= std_logic_vector(to_unsigned(sample_count,32));
    else
      index_out <= (others => '0');
    end if;

  end process clock_gen;
end Behavioral;
```

Timing

Program

A program contains the list of strictly monotonic increasing tick counts relative to the trigger input for all pulses that will be generated when a trigger arrives. It is the lowest level of control. A program can be compiled as a initial DELAY and REPEAT identical, subsequent cycles.

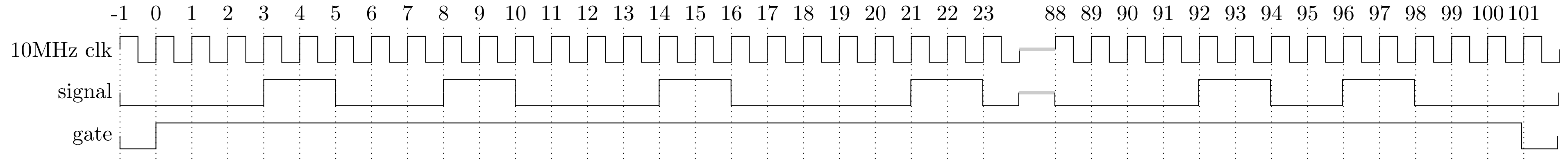


Cycle

During every cycle a digital signal is generated. A gate signal will be high during the sequence and low during idle time. The duration of one cycle is defined by CYCLE and controls the repetition rate.

Sequence $(\mathrm{WIDTH} = 2, \mathrm{PERIOD} = 5, \mathrm{SEQUENCE} = [3, 8, 14, 21, \cdots, 92, 96])$**, init**

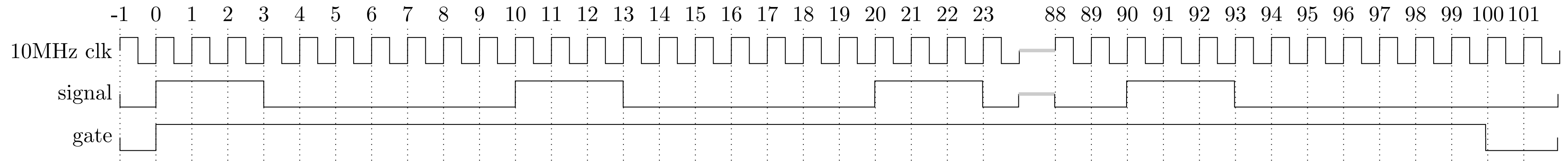
The digital signal is generated as a sequence of pulses of a given WIDTH. The timing of the pulses is defined by SEQUENCE that contains tick counts relative to the beginning of the cycle. The low-high transition of the gate is defined by the beginning of the cycle. The high-low transition is defined by the last element of SEQUENCE plus PERIOD.



Pulse train $(\mathrm{WIDTH} = 3, \mathrm{PERIOD} = 10, \mathrm{COUNT} = 10)$**, init2**

In pulse train mode the parameter PERIOD and COUNT are used generate a sequence of equidistant pulses.

$\mathrm{SEQUENCE} = [0 : \mathrm{COUNT} - 1] * \mathrm{PERIOD}$


-1 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 88 89 90 91 92 93 94 95 96 97 98 99 100 101
10MHz clk
signal
gate