IC TRAINING CENTER VIET NAM
FUNDAMENTAL DESIGN AND VERIFICATION COURSE


# FINAL PROJECT
# TIMER IP DESIGN SPECIFICATION

**Student : Hoang Thuy Tram**
**Class : IC26**
**Instructor: Pham Thieu Khang**

*Ho Chi Minh city, October 2025*

# Contents

# List of Tables



# List of Figures

# 1 Overview

## 1.1 Introduction

A timer is one of the fundamental components in digital systems and embedded processors. It serves as an essential hardware block in every chip, enabling accurate timing intervals and controlling the timing of various operations within a circuit. Timers are crucial for implementing real-time functionalities such as task scheduling, event triggering, pulse generation, delay creation, PWM generation, communication timing, and interrupt handling.

In this project, a timer is customized from the CLINT module of the industrial RISC-V architecture to generate interrupts based on user-defined settings.

## 1.2 Main features

In this project, the timer is a 32-bit programmable module interfaced with the APB (Advanced Peripheral Bus), offering flexible configuration and reliable operation for embedded systems.

Key features include:

- A counter capable of counting up to 64-bit width.
- A 12-bit input address space.
- A register set configured via the APB bus, where the timer IP acts as an APB slave.
- An active-low asynchronous reset to restore the system state.

Advanced capabilities supported by the timer:

- One wait-state `pready` timing.
- Error handling via `pslverr`.
- Byte-level access.
- Debug mode with halt functionality.
- Interrupt generation, which can be enabled or disabled by software.

## 1.3 Block diagram

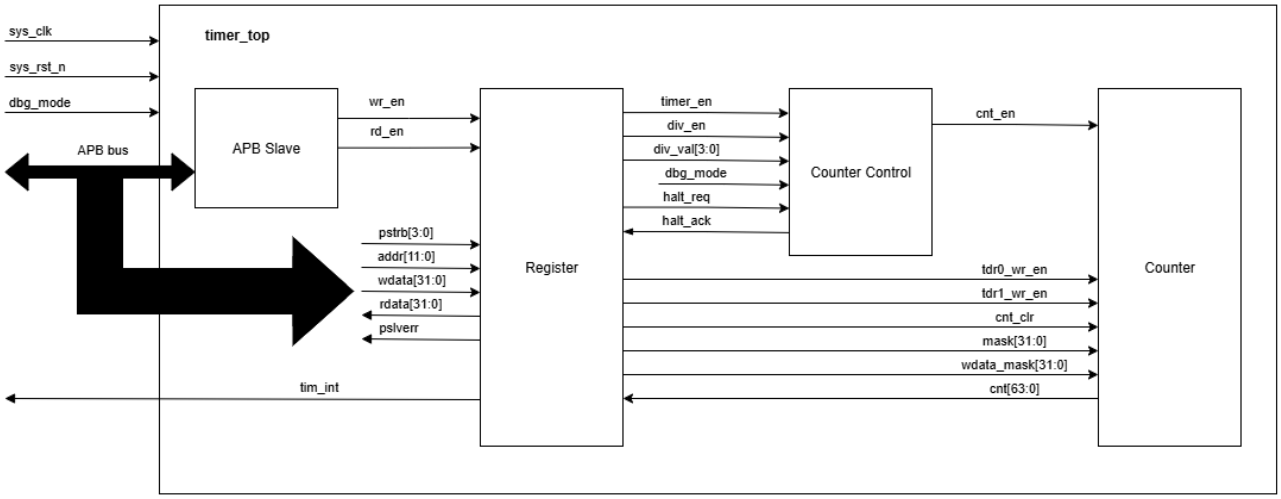



Figure 1: Block diagram of the timer IP.

The following documentation describes each sub-block of the timer IP and its role within the system architecture:

1. **APB Interface**: This block manages communication between the timer and the master (typically the CPU) via the APB. It decodes APB control signals to generate internal read and write enables for accessing the timer's registers. The interface also supports one wait-state timing using the `pready` signal.

2. **Register**: This block stores all control and status information for the timer. It handles register encoding and configuration such as `TCR, TDR` and `TCMP`. It also manages interrupt control, debug halt requests, and error detection using `pslverr` to prevent unauthorized access. Additionally, it monitors interrupt conditions, updates the interrupt status and enable bits, and allows software to clear the interrupt pending bit via the `TISR` register.

3. **Counter control**: This block determines the timer's counting mode, either default mode using the system clock or a control mode with a customized counting frequency via configurable clock division. It synchronizes with enable and clear signals and responds to halt requests during debug mode, ensuring the counter pauses until the halt condition is released. Based on these inputs, it generates an internal count-enable signal for the counter block.

4. **Counter**: The counter block performs the main actual counting operation based on the input clock and divider settings. It continuously updates the current count value and shares it with the register block. The counter restarts based on transitions of the `timer_en` signal. An interrupt is triggered when the current count matches the target value specified in the two `TCMP` registers.

## 1.4 Interface signals

| Signal name | Width | Direction | Description |
|---|---|---|---|
| sys_clk | 1 | Input | Active-high system clock for all synchronization operation |
| sys_rst_n | 1 | Input | Active-low asynchronous reset<br>When 0, all internal registers are cleared |
| tim_psel | 1 | Input | APB select signal<br>Asserted when the timer is selected |
| tim_pwrite | 1 | Input | APB write signal<br>1 for write operation, 0 for read |
| tim_penable | 1 | Input | APB enable signal<br>Active phase of a transfer |
| tim_paddr | 12 | Input | APB address bus specifying the internal register address |
| tim_pwdata | 32 | Input | APB write data bus carrying data to be written to internal registers |
| tim_prdata | 32 | Output | APB read data bus carrying data read from internal registers |
| tim_pstrb | 4 | Input | Byte access signal, used for partial writes<br>Each bit corresponds to one byte lane of tim_pwdata |
| tim_pready | 1 | Output | APB ready signal<br>Indicates the read completion of a read/write transaction<br>One wait state |
| tim_pslverr | 1 | Output | APB slave error signal<br>Asserted when a prohibited access occurs |
| tim_int | 1 | Output | Timer interrupt output signal<br>Asserted when interrupt trigger condition occurs |
| dbg_mode | 1 | Input | Debug mode enable signal<br>Allows timer halt and resume through the THCSR register |

Table 1: Interface signals of timer IP

# 2 Register Specification

## 2.1 Register summary

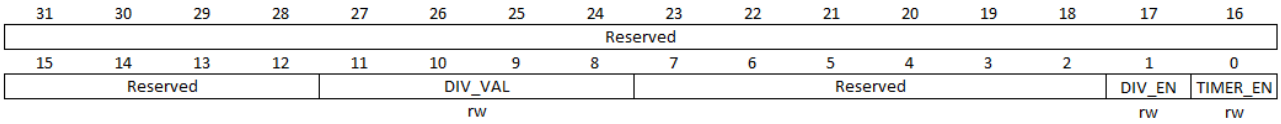
| Offset | Abbreviation | Register name |
|---|---|---|
| 0x00 | TCR | Timer Control Register |
| 0x04 | TDR0 | Timer Data Register 0 |
| 0x08 | TDR1 | Timer Data Register 1 |
| 0x0C | TCMP0 | Timer Compare Register 0 |
| 0x10 | TCMP1 | Timer Compare Register 1 |
| 0x14 | TIER | Timer Interrupt Enable Register |
| 0x18 | TISR | Timer Interrupt Status Register |
| 0x1C | THCSR | Timer Halt Control Status Register |
| Others | Reserved | |

Table 2: Register summary

## 2.2 Timer Control Register (TCR)

Offset address: 0x0

Reset value: 0x0000_0100



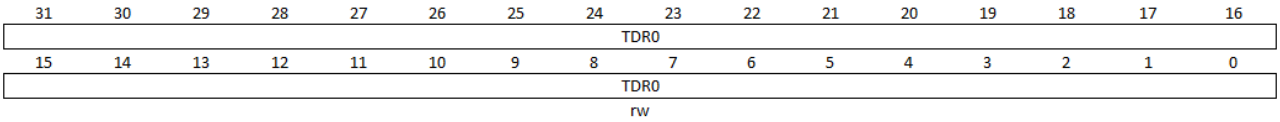

| Bit | Name | Type | Default value | Description |
|---|---|---|---|---|
| 31:12 | Reserved | RO | 20'h0 | Reserved |
| 11:8 | div_val [3:0] | RW | 4'b0001 | Counter control mode setting:<br>4'b0000: Counting speed is not divided<br>4'b0001: Counting speed is divided by 2 (default)<br>4'b0010: Counting speed is divided by 4<br>4'b0011: Counting speed is divided by 8<br>4'b0100: Counting speed is divided by 16<br>4'b0101: Counting speed is divided by 32<br>4'b0110: Counting speed is divided by 64<br>4'b0111: Counting speed is divided by 128<br>4'b1000: Counting speed is divided by 256<br>Others: Reserved (prohibited setting)<br>Note:<br>When setting the prohibited value, div_val is not changed<br>User must not change div_val while timer_en is High |
| 7:2 | Reserved | RO | 6'b0 | Reserved |
| 1 | div_en | RW | 1'b0 | Counter control mode enable<br>0: Disable. Counter counts with normal speed based on system clock<br>1: Enable. The counting speed of counter is controlled based on div_val<br>Note: User must not change div_en while timer_en is High |
| 0 | timer_en | RW | 1'b0 | Timer enable<br>0: Disable. Counter does not count<br>1: Enable. Counter starts counting<br>timer_en changes from H→ L will initialize TDR0/1 to their initial value |

Table 3: Timer Control Register (TCR)

## 2.3 Timer Data Register 0 (TDR0)

Offset address: 0x4

Reset value: 0x0000_0000



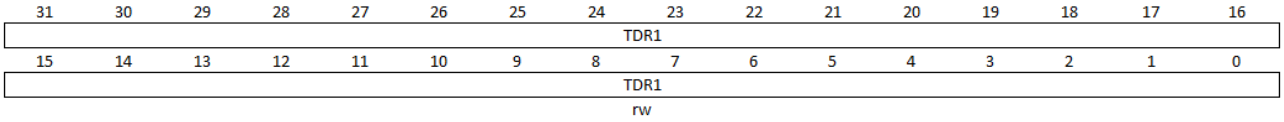

| Bit | Name | Type | Default value | Description |
|---|---|---|---|---|
| 31:0 | TDR0 | RW | 32'h0000_0000 | Lower 32-bit of 64-bit counter<br>Note: Value of this register is cleared to initial value when `timer_en` changes from H→ L |

Table 4: Timer Data Register 0 (TDR0)

## 2.4 Timer Data Register 1 (TDR1)

Offset address: 0x8

Reset value: 0x0000_0000



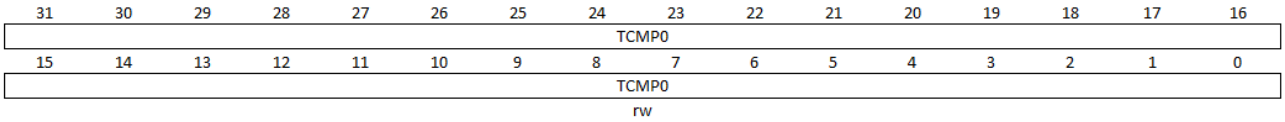

| Bit | Name | Type | Default value | Description |
|---|---|---|---|---|
| 31:0 | TDR1 | RW | 32'h0000_0000 | Upper 32-bit of 64-bit counter<br>Note: Value of this register is cleared to initial value when `timer_en` changes from H→ L |

Table 5: Timer Data Register 1 (TDR1)

## 2.5 Timer Compare Register 0 (TCMP0)

Offset address: 0xC

Reset value: 0xFFFF_FFFF



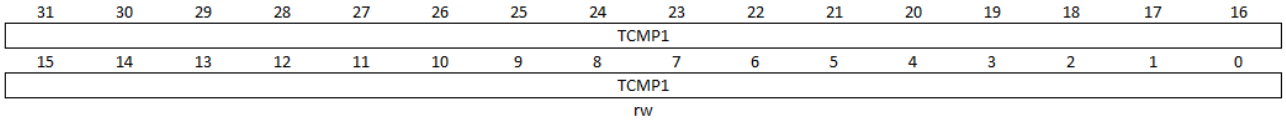

| Bit | Name | Type | Default value | Description |
|---|---|---|---|---|
| 31:0 | TCMP0 | RW | 32'hFFFF_FFFF | Lower 32-bit of 64-bit compare value<br>Interrupt pending bit is asserted when counter value is equal to compare value |

Table 6: Timer Compare Register 0 (TCMP0)

## 2.6 Timer Compare Register 1 (TCMP1)

Offset address: 0x10

Reset value: 0xFFFF_FFFF



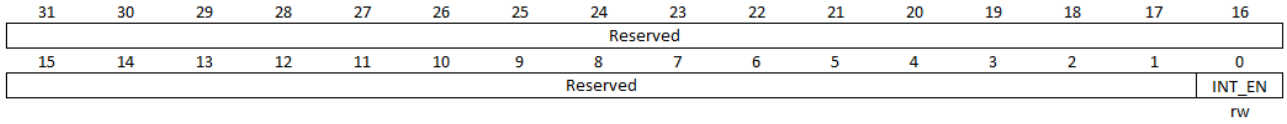

| Bit | Name | Type | Default value | Description |
|---|---|---|---|---|
| 31:0 | TCMP1 | RW | 32'hFFFF_FFFF | Upper 32-bit of 64-bit compare value<br>Interrupt pending bit is asserted when counter value is equal to compare value |

Table 7: Timer Compare Register 1 (TCMP1)

## 2.7 Timer Interrupt Enable Register (TIER)

Offset address: 0x14

Reset value: 0x0000_0000



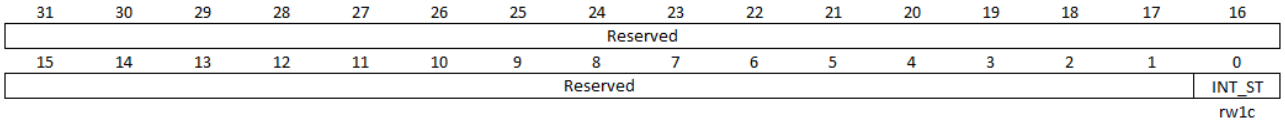

| Bit | Name | Type | Default value | Description |
|---|---|---|---|---|
| 31:1 | Reserved | RO | 31'h0 | Reserved |
| 0 | int_en | RW | 1'b0 | Timer interrupt enable<br>0: Timer interrupt is disabled<br>1: Timer interrupt is enabled<br>When this bit is 0, no timer interrupt is ouptut<br>When this bit is 1, timer interrupt pending bit can be output (interrupt pending bit is set when counter value equals to compare value)<br>Clearing this bit to 0 while interrupt is asserting will mask the interrupt to 0 but does not affect the interrupt pending bit (TISR.int_st bit) |

Table 8: Timer Interrupt Enable Register (TIER)

## 2.8 Timer Interrupt Status Register (TISR)

Offset address: 0x18

Reset value: 0x0000_0000



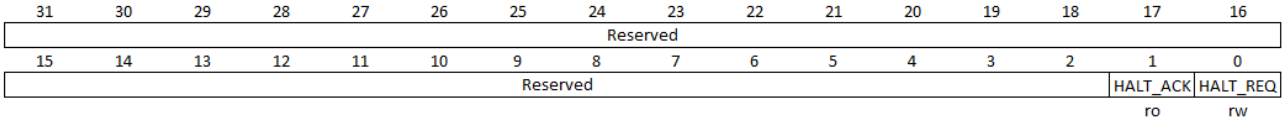
| Bit | Name | Type | Default value | Description |
|---|---|---|---|---|
| 31:1 | Reserved | RO | 31'h0 | Reserved |
| 0 | int_st | RW1C | 1'b0 | Timer interrupt trigger condition status bit (interrupt pending bit)<br>0: The interrupt trigger condition does not occur<br>1: The interrupt trigger condition occured<br>Write 1 when this bit is 1 to clear it to 0<br>Write 0 when this bit is 1 has no effect<br>Write to this bit when it is 0 has no effect<br>Note: When interrupt trigger condition occured (counter reached compare value), counter continues to count normally |

Table 9: Timer Interrupt Status Register (TISR)

## 2.9 Timer Halt Control Status Register (THCSR)

Offset address: 0x1C

Reset value: 0x0000_0000




| Bit | Name | Type | Default value | Description |
|---|---|---|---|---|
| 31:2 | Reserved | RO | 30'h0 | Reserved |
| 1 | halt_ack | RO | 1'b0 | Timer halt acknowledge<br>0: Timer is not halted<br>1: Timer is halted<br>Timer accepts the halt request only in debug mode, indicates by dbg_mode input signal |
| 0 | halt_req | RW | 1'b0 | Timer halt request<br>0: No halt request<br>1: Timer is requested to halt |

Table 10: Timer Halt Control Status Register (THCSR)

# 3 Functional Description

## 3.1 APB slave

The timer IP includes an APB slave which acts as the communication interface between the timer module and the system bus. It follows the AMBA APB protocol, handling all read and write transactions from the bus master. This APB slave has the following features:

1. Supports APB 32-bit transfers.
2. Supports one wait state to improve the timing, implemented using internal pready `int_pready` signal to ensure proper timing for all transactions.
3. Generates write and read enable controls. The APB slave generates `wr_en` and `rd_en` signals for the register blocks to handle write and read transactions, only when valid APB conditions are met.
4. Supports byte access, allowing the bus to access individual bytes in the register. Selective byte writes within 32-bit registers are enabled using the 4-bit strobe signal `pstrb`.
5. Support error handling using the `pslverr` signal in cases of illegal writes, such as modifying `TCR.div_en` or `TCR.div_val` while the timer is active, or writing prohibited values to `TCR.div_val`. When an error occurs, the APB slave prevents updates to the affected register bits or fields.
6. The APB is synchronized to the system clock and supports asynchronous active-low reset.

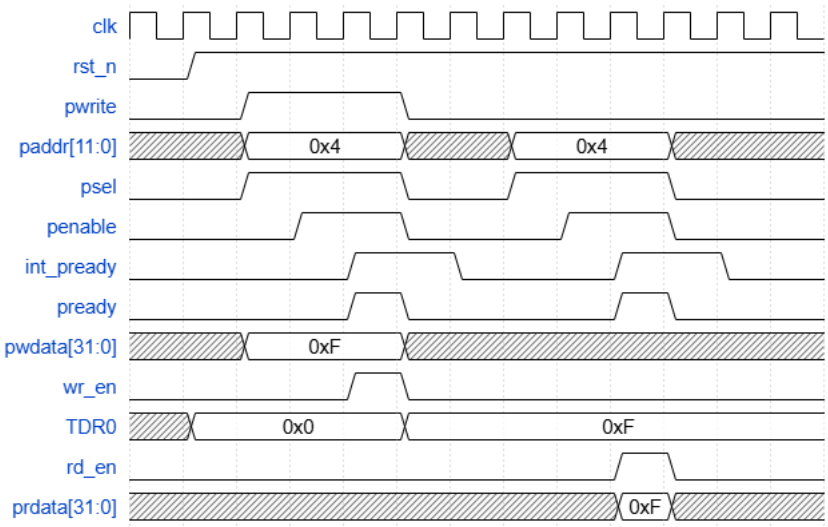



Figure 2: Sample waveform of the APB interface.

The APB interface supports a single wait-state using the `pready` signal. When `pready` transitions from High to Low, related signals `psel` and `penable` are also deasserted, returning the bus to idle.

The waveform illustrates the transaction sequence: after `psel` is asserted, `penable` follows in the next clock cycle (setup phase). After one wait-state, `pready` is asserted (access phase), completing the transfer. Both `wr_en` and `rd_en` are asserted during the access phase, synchronized with `pready`; the distinction between them is determined by the `pwrite` signal (High for write, Low for read).

In the example, a write to register `TDR0` (offset 0x4) with data 0xF is performed, followed by a read to confirm the transaction. When `wr_en` is asserted, `TDR0` is updated; a subsequent read with `rd_en` returns the value 0xF on `prdata[31:0]`, verifying correct operation.

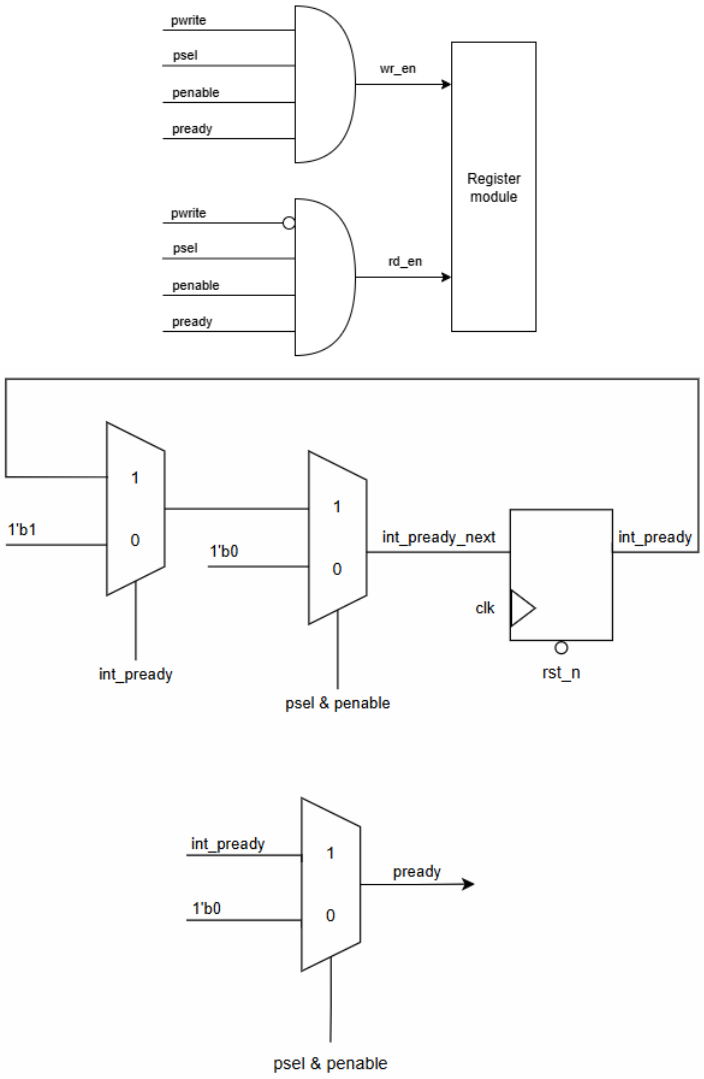



Figure 3: Logic diagram of the APB interface.

The sample waveform in Figure 2 demonstrates the expected APB behavior and guides the logic diagram design. Figure 3 shows how the `wr_en` and `rd_en` signals are generated to meet protocol requirements. As explained, these signals are asserted only during the access phase, when `psel`, `penable`, and `pready` are all High. The difference between write and read enable is determined by the `pwrite` signal: High for write, Low for read. Both signals are sent to the register module to control register write and read operations.

The `pready` signal is asserted one clock cycle after the APB enters the setup state (`psel` and `penable` asserted). If `pready` goes Low, other transaction signals also go Low. To implement this, the internal signal `int_pready` helps control `pready`. As shown in Figure 3, `pready` is combinational: when `psel` and `penable` are High, `pready` follows `int_pready`; otherwise, it remains Low. The logic for `int_pready` is as follows: if `psel` and `penable` are High and `int_pready` is not yet asserted (setup phase), `int_pready` will be set High in the next clock cycle. If `int_pready, psel`, and `penable` are all High (access phase), `int_pready` remains unchanged.

## 3.2 Register

The IP timer includes a register module that serves as the control and configuration center for the timer. It stores and manages all internal timer registers, such as control, data, compare, interrupt and halt registers. This block interfaces directly with the APB slave to receive write and read enable signals, updating internal registers accordingly. The register module also communicates with other functional blocks, providing configurable values (such as `TCR.timer_en`, `TCR.div_en`, `TCR.div_val` and `THCSR.halt_req`) to the counter control module, or `tdr0_wr_en`, `tdr1_wr_en` to the counter module. It receives status feedback, such as `THCSR.halt_ack` from the counter control or `cnt` from the counter. The key features offered by the module are:

1. Manages all timer configuration and status registers, including `TCR`, `TDR0/1`, `TCMP0/1`, `TIER`, `TISR` and `THCSR`.
2. Supports read and write operations via signals from the APB slave.
3. Handles the `pstrb` input to allow partial register updates for efficient data access.
4. Prevents prohibited access, such as invalid writes to `TCR.div_val` or changes to `TCR.div_en` and `TCR.div_val` while the timer is running, by asserting the `pslverr` signal.
5. Performs automatic reset, initializing all registers when reset is asserted, and clears `TDR0` and `TDR1` when the timer transitions from High to Low.
6. Thoroughly manages interrupt and debug modes to help synchronize timer operation.

### 3.2.1 pstrb signal

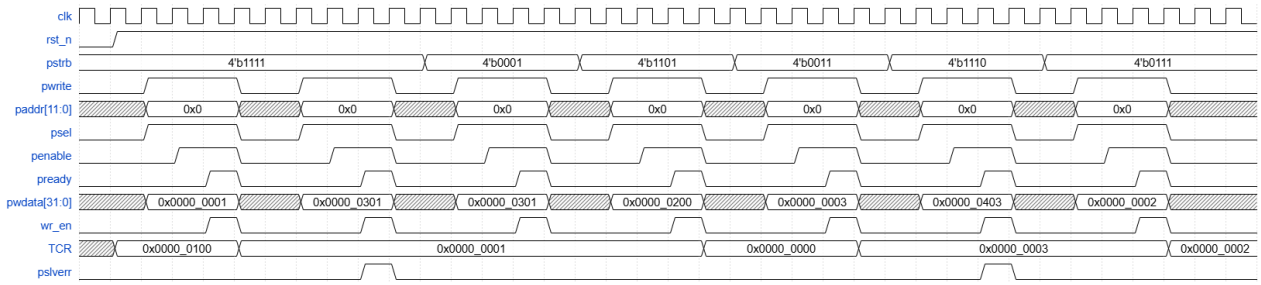



Figure 4: Sample waveform of the `pstrb` signal.

The sample waveform in Figure 4 shows `pstrb` at 0xF, enabling full 32-bit access. Writing 1 to `TCR` updates the register. Writing 0x0000_0301 to `TCR` while the timer is running asserts `pslverr`, so `TCR` is not updated. With `pstrb` set to 4'b0001, writing 0x0000_0301 does not trigger an error since only the lowest byte is affected. Writing 0x0000_0200 with `pstrb` at 4'b1101 updates `TCR` to 0x0 without error. Writing 0x3 with `pstrb` at 4'b0011 updates TCR to 0x3. Writing 0x0000_0403 with 4'b1110 `pstrb` triggers `pslverr` due to a prohibited access. Finally, writing 0x2 with 4'b0011 `pstrb` updates `TCR` successfully, as no prohibited access occurs.

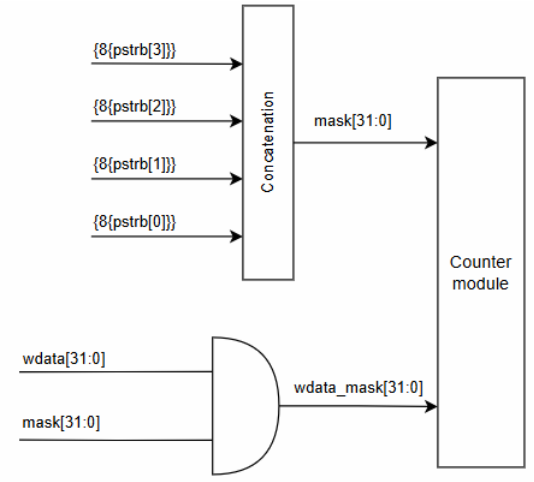



Figure 5: Logic diagram of the `pstrb` signal.

Each `pstrb` bit represents one byte, so each bit is expanded eight times to indicate which byte fields are accessible. This is reflected in the `mask[31:0]` and `wdata_mask[31:0]` signals.

### 3.2.2 TCR register

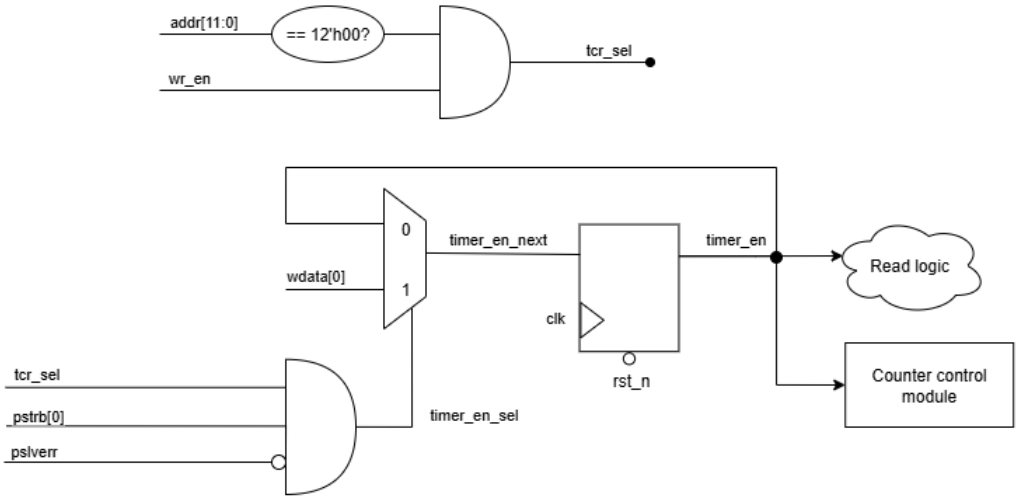



Figure 6: Logic diagram of the `TCR.timer_en` signal.

The `TCR.timer_en` signal is updated when a value is written to the first bit of the `TCR` register (offset 0x0), provided that `pstrb[0]` is active and `pslverr` is not asserted. If either condition is not met, `TCR.timer_en` retains its current value.

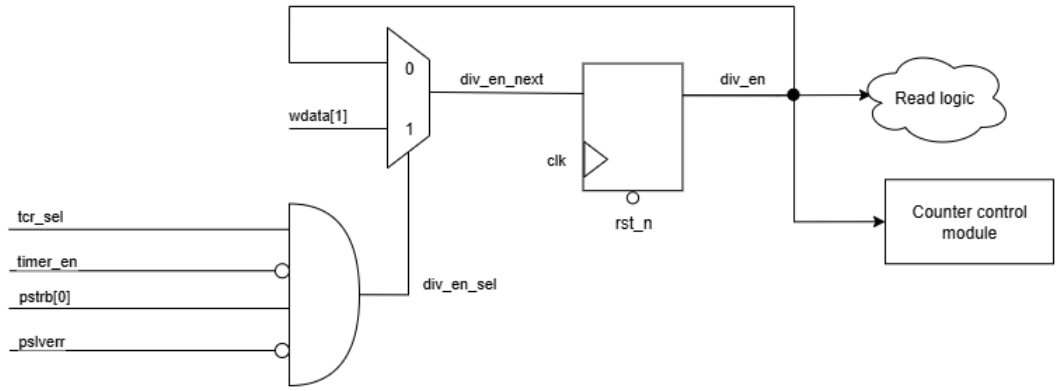



Figure 7: Logic diagram of the `TCR.div_en` signal.

Similar to the `TCR.timer_en` signal, `TCR.div_en` is updated when a value is written to the second bit of the `TCR` register, provided that `pstrb[0]` is active, `pslverr` is not asserted, and the timer is not running. If any of these conditions are not met, `TCR.div_en` retains its current value.

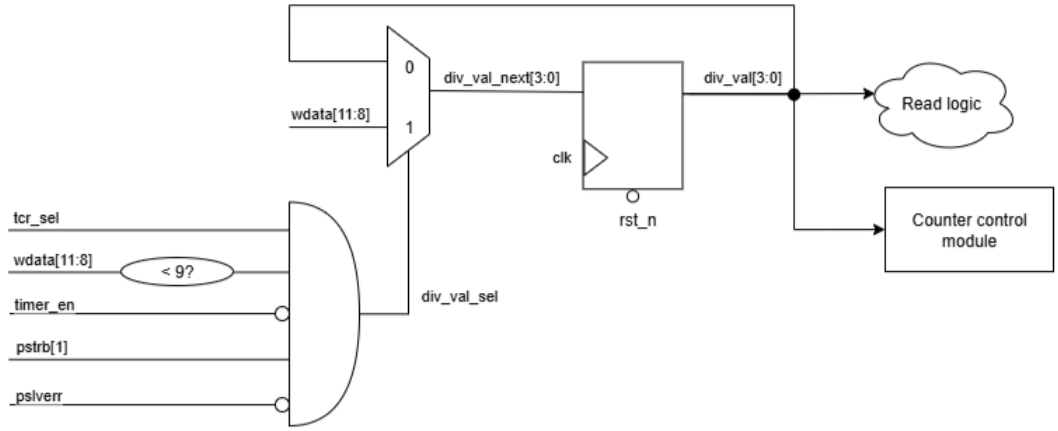



Figure 8: Logic diagram of the `TCR.div_val` signal.

The `div_val` signal has stricter update requirements: it is only updated when writing a value less than 9 to bits 8–11 of the `TCR` register, with `pstrb[1]` active, `pslverr` not asserted, and the timer not running. If any condition is violated, `div_val` retains its current value.

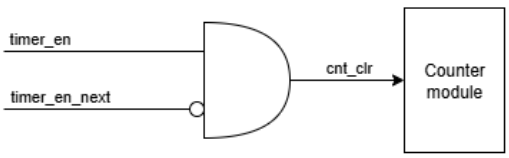



Figure 9: Logic diagram of the `cnt_clr` signal.

The clear condition for `TDR0/1` occurs when `timer_en` transitions from High to Low. If the current value of `timer_en` is High and the next value is Low, a clear signal is triggered and sent to the counter module to reset the `TDR0/1` values.

### 3.2.3 TDR registers

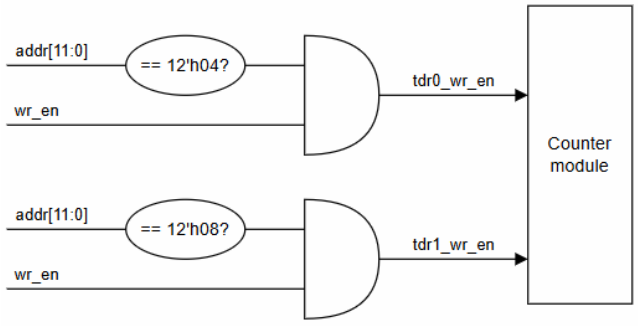



Figure 10: Logic diagram of the `TDR0` and `TDR1` registers.

When a write transaction occurs at address 12'h4, the `tdr0_wr_en` signal is sent to the counter module. Similarly, a write transaction at address 12'h8 triggers `tdr1_wr_en`. These signals enable direct writing to the counter: the lower 32 bits for `TDR0` and the upper 32 bits for `TDR1`, bypassing normal counting.

### 3.2.4 TCMP registers

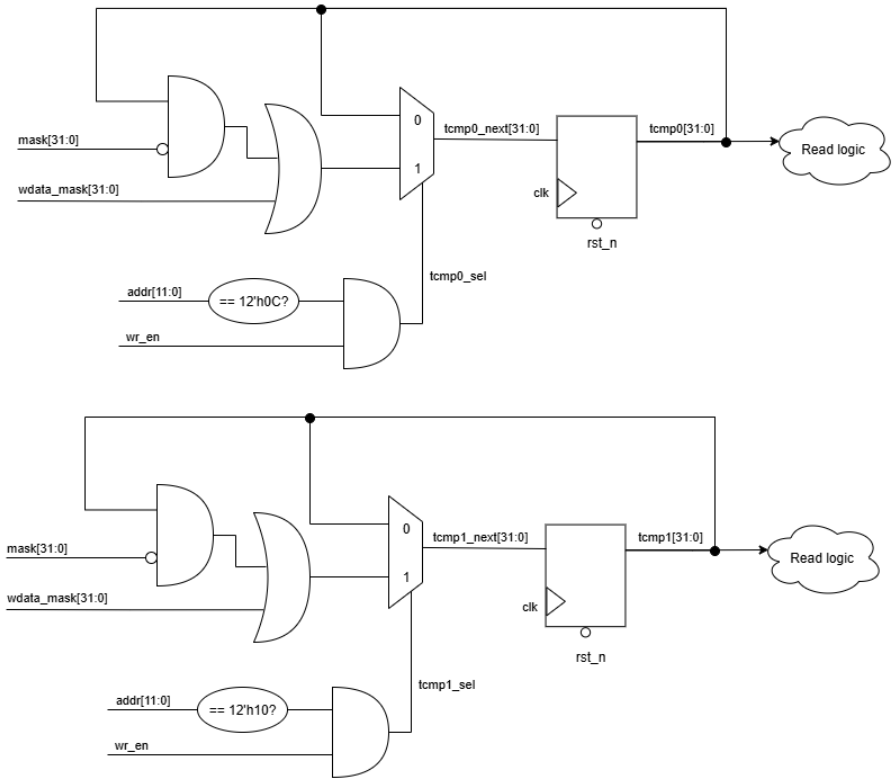



Figure 11: Logic diagram of the `TCMP0` and `TCMP1` registers.

When a write transaction occurs at address 12'hC, the write data is transferred to the `TCMP0` register. The byte access mechanism uses the inverted `mask[31:0]` to preserve unchanged bits, while `wdata_mask[31:0]` ensures only the selected bytes are updated. If the update conditions are not met, `TCMP0` retains its current value. `TCMP1` (offset 0x10) follows the same logic.

### 3.2.5 pslverr signal

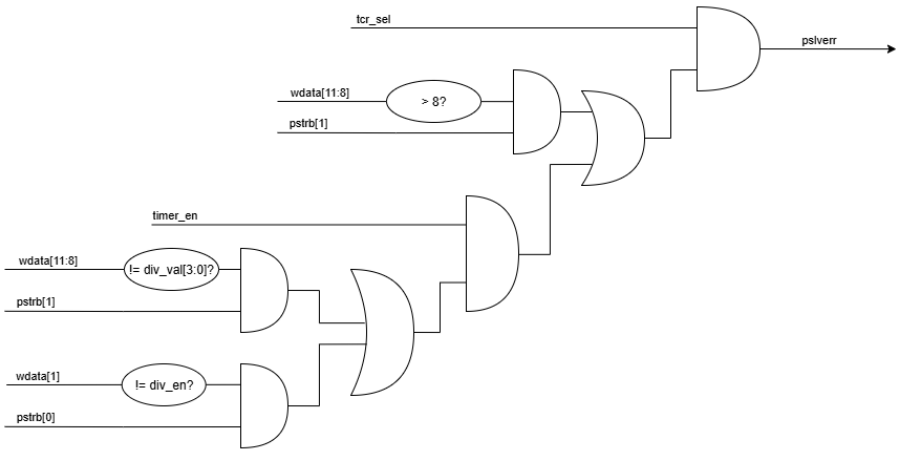



Figure 12: Logic diagram of the `pslverr` signal.

There are three cases that trigger the `pslverr` signal:

1. Writing a prohibited value (greater than 8) to `TCR.div_val`.
2. Attempting to change `TCR.div_en` while the timer is running.
3. Attempting to change `TCR.div_val` while the timer is running.

For `pslverr` to be triggered, the corresponding `pstrb` bit must be active: `pstrb[0]` for `TCR.div_en` and `pstrb[1]` for `TCR.div_val`. If these bits are not active, no update occurs and `pslverr` is not asserted.

### 3.2.6 TIER and TISR registers

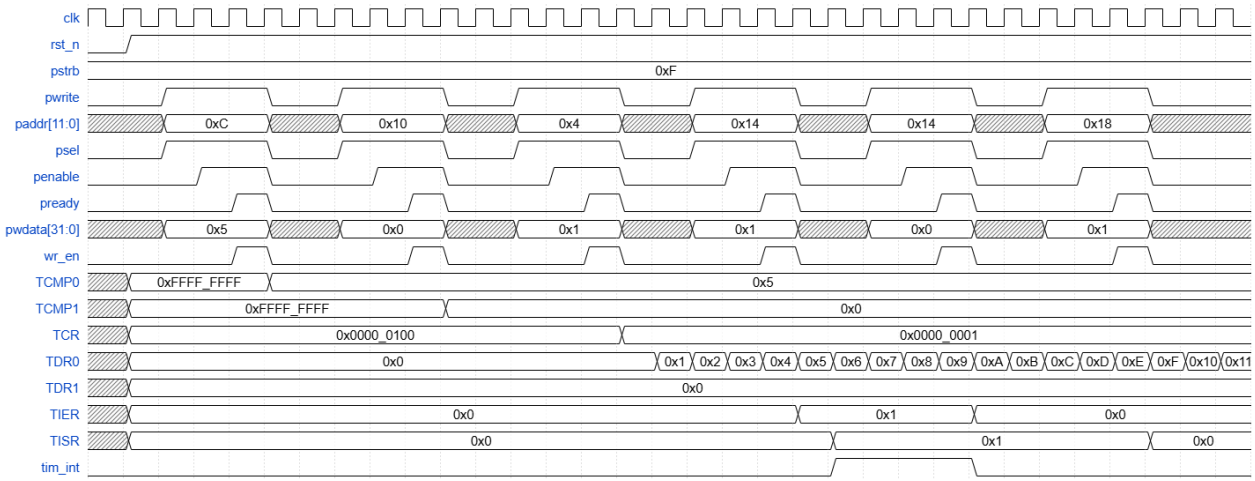



Figure 13: Sample waveform of the `TIER` and `TISR` registers.

The sample waveform sets `TDR0` and `TDR1` to 0x5 and 0x0, respectively, then enables the timer by writing 1 to the `TCR` register (offset 0x0). The counter begins normal operation. Writing 1 to the `TIER` register (offset 0x14) asserts the `TIER.int_en` signal. When the counter

reaches 0x5 (TDR0), TISR is updated to 1 on the next clock cycle, indicating the interrupt pending bit is set. If both TIER.int_en and TISR.int_st are High, the timer interrupt is asserted. Writing 0 to TIER masks and deasserts the interrupt, but the pending bit remains until writing 1 to TISR clears it.

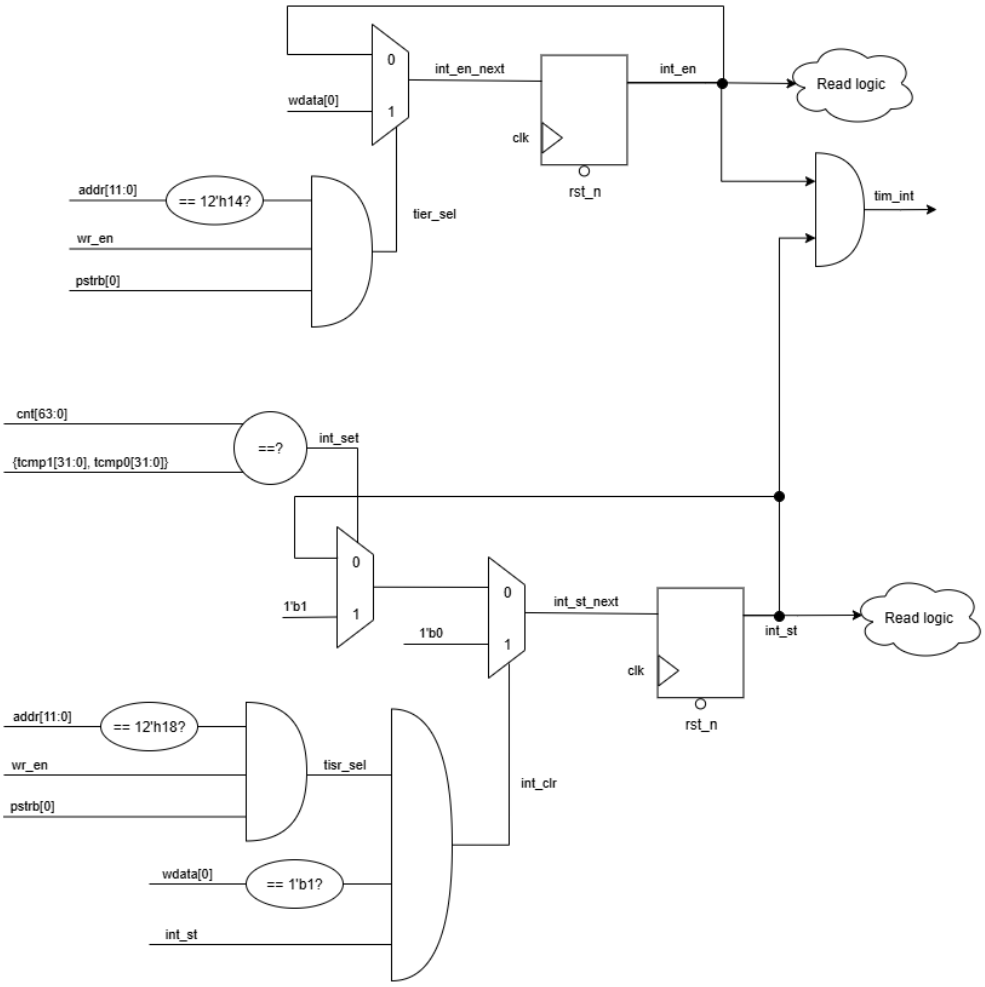



Figure 14: Logic diagram of the TIER and TISR registers.

The interrupt enable signal (TIER.int_en) is set by writing a value of 1 to its register address (12'h14) with pstrb[0] asserted, ensuring the first byte of the write data is used for the update. If these conditions are met, TIER.int_en is updated; otherwise, it retains its previous value. This mechanism allows precise control over interrupt activation, preventing unintended changes when the required address or strobe is not asserted.

The interrupt status bit (TISR.int_st), also called the interrupt pending bit, is set when the counter matches the value programmed in the TCMP0/1 registers. To clear TISR.int_st, write a value of 1 to the TISR register at address 12'h18 with pstrb[0] asserted, ensuring the first byte of the write data is used. Clearing only occurs if int_st is already set; writing 1 when it is 0, or writing 0 at any time, has no effect. The clear action takes priority over setting, so if both conditions are met, TISR.int_st will be cleared first. If neither clear nor set conditions are satisfied, TISR.int_st retains its previous value. This logic ensures reliable interrupt status management and prevents accidental clearing or setting.

When both signal TIER.int_en and TISR.int_st are High, the timer interrupt will be asserted, indicating that the timer has been interrupted.

### 3.2.7 THCSR register

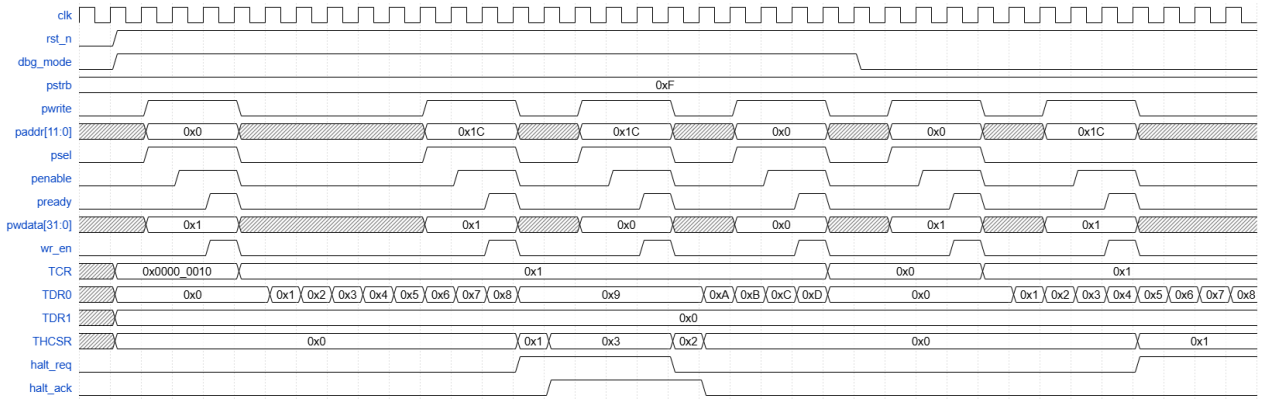



Figure 15: Sample waveform of the `THCSR` register.

To successfully halt the counter, debug mode must be enabled. First, the timer is started by writing 1 to the `TCR` register, allowing the counter to increment as shown in `TDR0`. Next, a halt request is issued by writing 1 to the `THCSR` register (offset 0x1C). Once the transfer completes, `THCSR.halt_req` is asserted, and after one clock cycle, `THCSR.halt_ack` acknowledges the request, resulting in `THCSR` holding a value of 3. When `THCSR.halt_req` is active, the counter stops immediately and resumes from the same value when released. If debug mode is disabled, asserting `THCSR.halt_req` does not generate `THCSR.halt_ack`, so the counter continues running. Thus, halt requests only affect the timer when debug mode is active, ensuring controlled debugging behavior.

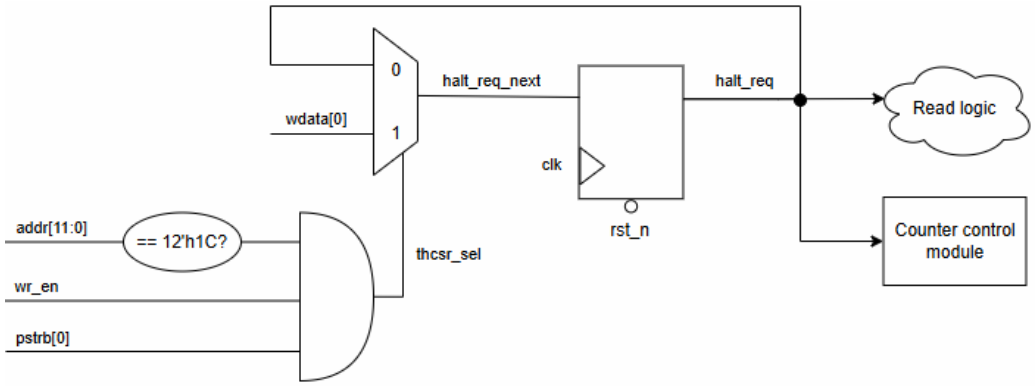



Figure 16: Logic diagram of the `THCSR.halt_req` signal.

The `THCSR.halt_req` signal is updated when a write occurs at its register address (12'h1C) with `pstrb[0]` asserted, ensuring the first byte of the write data is used. If these conditions are met, `THCSR.halt_req` is set; otherwise, it retains its previous value. Once asserted, `THCSR.halt_req` is sent to the counter control module, which uses it to generate `THCSR.halt_ack` and to determine when the counter should stop. This mechanism ensures that halt requests are processed only under valid conditions and provides reliable control over counter operation during debugging.

### 3.2.8 Read logic

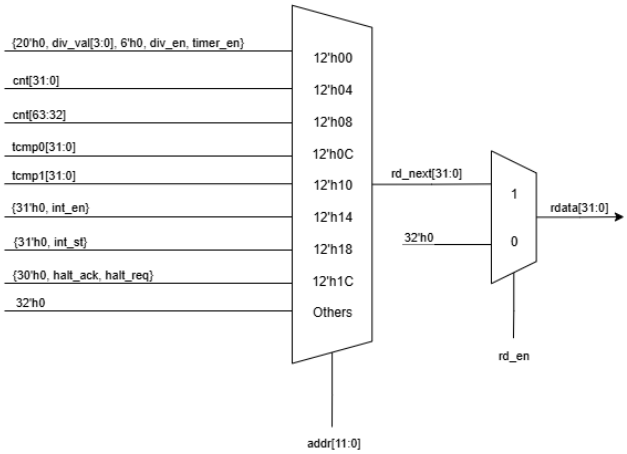



Figure 17: Logic diagram of the read logic.

When the `rd_en` signal from the APB interface is asserted, `rdata[31:0]` outputs the value of the selected timer register. Valid timer registers return their current values, while reserved or undefined registers output 32'h0 to indicate they are not accessible. This ensures only legitimate timer data is read and prevents unintended access to reserved areas.

## 3.3 Counter control

The counter control module acts as a bridge between the register module and the counter module, managing configuration tasks such as debug mode and control mode. This approach simplifies the IP timer design while meeting all requirements. Key features of the counter control module include:

1. Processes the `THCSR.halt_ack` signal, which indicates successful activation of debug mode, and returning this signal to the register module for read transactions of the `THCSR` register.

2. Supports multiple control modes to ensure the counter operates according to user configuration. For example, it handles both normal counting mode (counter runs at system clock speed) and controlled counting mode (counter speed determined by the `TCR.div_val` value).

3. Manages debug mode activation to halt the counter accurately and promptly. Regardless of the counting mode, when debug mode is active, the counter remains halted until debug mode is deactivated by the user.

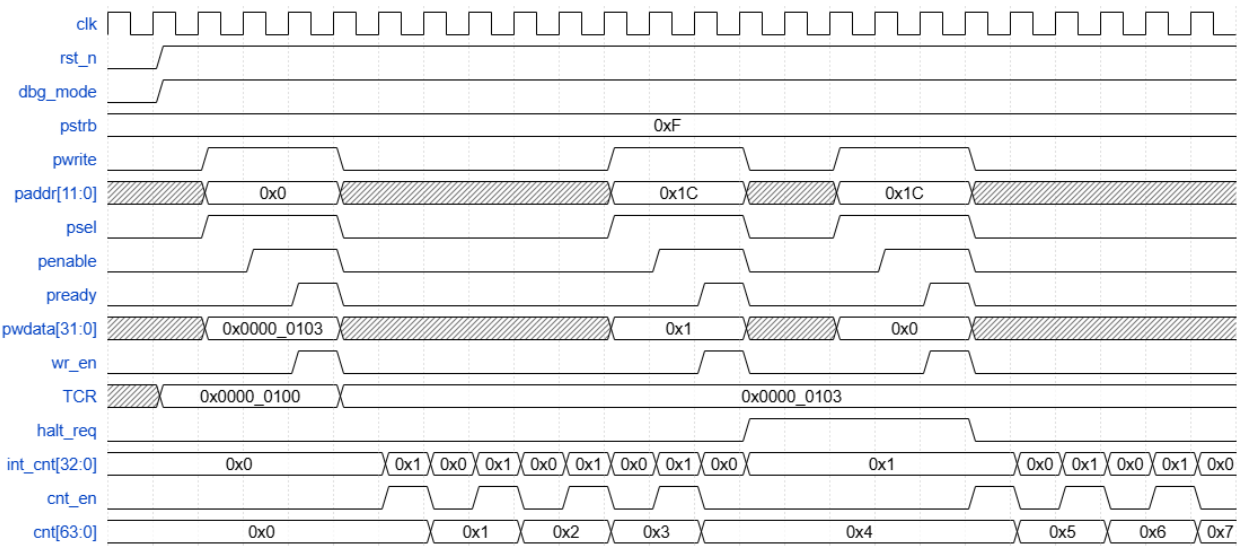



Figure 18: Sample waveform of the counter control module.

A basic counting mode is used to illustrate how the counter control module manages the counter in Figure 18. With `TCR.div_val` set to `4'b0001`, the counting speed is divided by 2. After a write transaction, the `TCR` register updates to 0x0000_0103, indicating the timer is enabled, control mode is active, and `TCR.div_val` = 1.

One clock cycle after this update, the internal counter begins counting. Since the speed is divided by 2, the counter requires $2^{div\_val}$ cycles to increment once. Starting from 0, the internal counter's maximum value is $2^{div\_val} - 1$; in this case, 1. When the internal counter reaches its maximum value, the `cnt_en` signal is asserted, triggering the actual counter to increment on the next cycle. Thus, the counter updates its value every two clock cycles.

A halt request demonstrates the effect of debug mode on the counter. When the halt request is asserted by writing 1 to the `THCSR` register and debug mode is active, the internal counter stops, preventing further updates. As a result, `cnt_en` is not triggered and the actual counter

does not increment. Once the halt request signal `THCSR.halt_req` is deasserted, the internal counter `int_cnt` resumes counting, allowing `cnt_en` to be asserted and the counter to increment again.

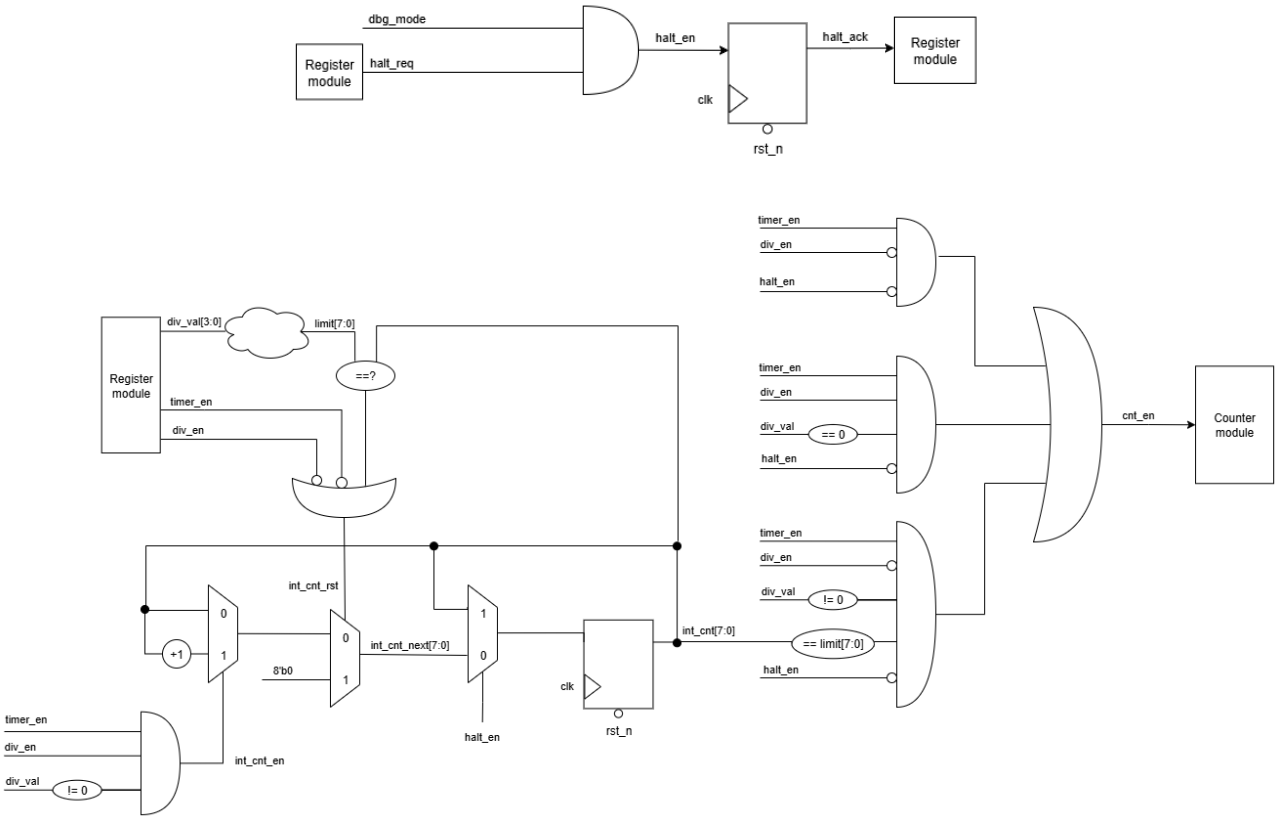



Figure 19: Logic diagram of the counter control module.

Based on the configuration and the illustrated operation of the counter control in Figure 18, the logic was designed as follows. In Figure 19, the `THCSR.halt_ack` signal is asserted when both debug mode and halt request `THCSR.halt_req`, set by writing 1 to the `THCSR` register, are active, indicating that debug mode is acknowledged. This signal is then sent back to the register module as an input for the `THCSR` register during read transactions.

The `cnt_en` signal is determined by three cases:

1. **Normal mode:** If control mode is not asserted (`TCR.div_en` is Low), the counter operates at the system clock speed, unaffected by `TCR.div_val`. As long as `TCR.timer_en` is High, `TCR.div_en` is Low, and no halt request is active (`halt_en` is Low), `cnt_en` is asserted to enable counting.

2. **Control mode, no division:** If control mode is asserted (`TCR.div_en` is High) but `TCR.div_val` is 0, the counter still runs at the system clock speed, similar to normal mode, and the internal counter is not used. In this case, if `TCR.timer_en` and `TCR.div_en` are High, `TCR.div_val` is 0, and no halt request is active, `cnt_en` is asserted.

3. **Control mode, with division:** If control mode is asserted and `TCR.div_val` is not 0, the counter runs at a divided speed as specified by `TCR.div_val`. The internal counter counts up to its maximum value to trigger `cnt_en`. The `int_cnt_en` signal determines when the internal counter can increment, while `int_cnt_clr` clears the internal counter

if deactivation conditions occur; such as `TCR.timer_en` or `TCR.div_en` going Low, or the internal counter reaching its maximum value. If a halt request occurs, it takes priority over `int_cnt_en` and `int_cnt_clr`, stopping the internal counter immediately. When the halt request is released in debug mode, the internal counter resumes from where it stopped. Therefore, if `TCR.timer_en` and `TCR.div_en` are High, `TCR.div_val` is not 0, and no halt is active, `cnt_en` is asserted and the counter can count.

## 3.4 Counter

The counter module is the core functional block of the timer system, responsible for counting clock cycles according to configuration values received from the register module. It operates based on the control signal `cnt_en` from the counter control module, which enables advanced features such as counting in control mode and halting in debug mode. Key features of the counter module include:

1. Allows flexible counting operation, the counter can synchronize with the system clock and operate with or without clock division, depending on `TCR.div_en` and `TCR.div_val`.

2. Counting is triggered by the `cnt_en` signal from the counter control module, enabling the timer's internal operation.

3. Receives `tdr0/1_wr_en` signals from the register module to update its value via write transactions to `TDR0/1`, allowing direct value assignment instead of sequential counting.

4. The counter outputs its value `cnt` to the register module for comparison with TCMP0/1, which is used to trigger interrupt pending bit.

5. When debug mode is active and `THCSR.halt_req` is asserted, the counter stops immediately and can resume from its previous value when the halt request is released.

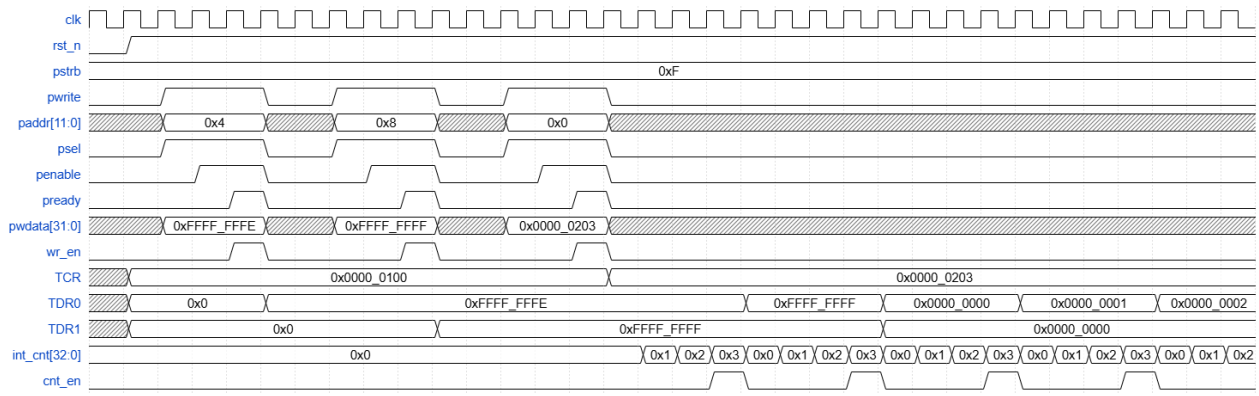



Figure 20: Sample waveform of the counter module.

The sample waveform in Figure 20 illustrates the basic operation of the counter. Values can be written to the `TDR0` and `TDR1` registers, allowing the counter to start from these values instead of counting up from zero. In this example, 0xFFFF_FFFE is written to `TDR0` and 0xFFFF_FFFF to `TDR1`, then the timer is enabled with the control mode set to divide the clock by 4. Once counting begins, the internal counter increments from 0 to 3, which is the maximum value determined by `TCR.div_val`. The internal counter implements clock division by counting up to the value specified by `TCR.div_val`, ensuring the main counter only increments at the desired rate. As a result, the counter increments by one every four clock cycles. After reaching the maximum value of 0xFFFF_FFFF_FFFF_FFFF, the counter wraps around to 0 and continues counting. This is the expected behavior, if the counter does not wrap and continue counting after reaching its maximum value, it indicates a malfunction in the design.

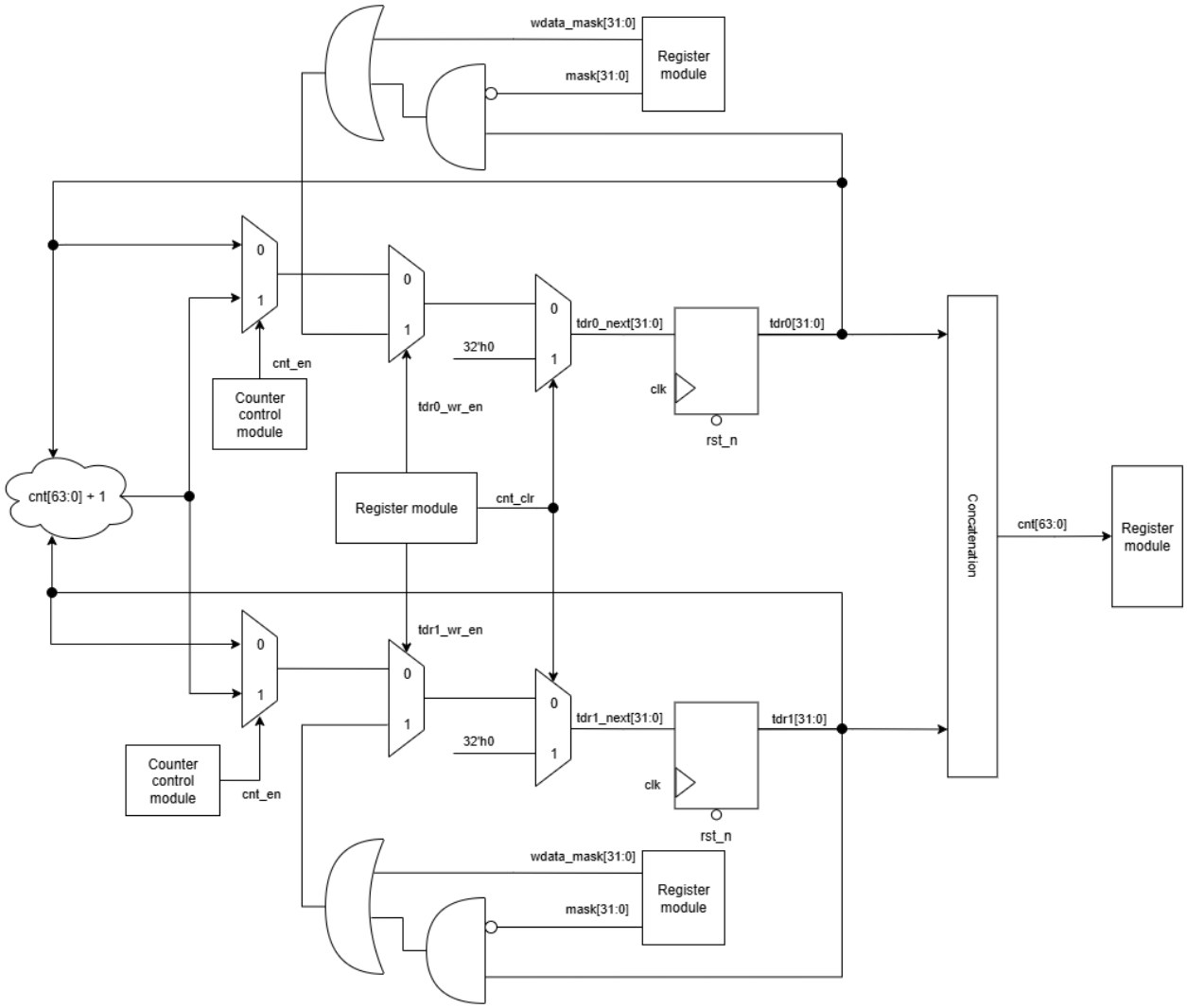



Figure 21: Logic diagram of the counter module.

The clear operation has the highest priority: when the counter receives the signal `cnt_clr` (asserted when `TCR.timer_en goes from High to Low`) from the register module, it immediately stops counting and clears the values in `TDR0` and `TDR1` registers. The write operation has the second priority: if no clear operation occurs, the counter checks for a write transfer to `TDR0` or `TDR1` via the `tdr0/1_wr_en` signals from the register module. If write enable is asserted for either `TDR0` or `TDR1`, the counter updates its value with the data provided by the user. The specific bit fields in the counter are updated according to the active bits in `pstrb`, similar to the write operation for `TCMP0/1` registers. If neither clear nor write operations occur, the counter defaults to normal counting, incrementing its value based on the `cnt_en` signal. Each time `cnt_en` is asserted, the counter increments by one; otherwise, it retains its current value.

This priority logic ensures that clear and write operations are handled immediately, while normal counting only proceeds when no higher-priority operation is active. The design guarantees reliable counter updates and correct handling of register writes and clears, as shown in the logic diagram in Figure 21.

# 4 History

| Date | Version | Description | Author |
|---|---|---|---|
| 01/10/2025 | 1.0 | Initial creation | Hoang Thuy Tram |
| 11/10/2025 | 1.0 | Specification finalized | Hoang Thuy Tram |
| | | | |

Table 11: History.