TinyRISC Specification

Random Person

July 2025

Contents

1 Instruction Set

1.1 Instruction Opcodes

Instruction	Code	Instruction	Code	Instruction	Code
add*	00000	not*	01000	beq	10000
sub*	00001	mov*	01001	bgt	10001
hlt	00010	lsl*	01010	b	10010
xor*	00011	lsr*	01011	call	10011
asl*	00100	asr*	01100	ret	10100
cmp*	00101	nop	01101	iret	10101
and*	00110	ld*	01110		
or*	00111	st*	01111		

* - instructions that have immediate operands and can have modifiers.

1.2 Instruction Formats

Format	Definition				
branch	op (31–27)	offset (26–0)			
register	op (31–27)	I (26)	rd (25–22)	rs1 (21–18)	rs2 (17–14)
immediate	op (31–27)	I (26)	rd (25–22)	rs1 (21–18)	imm (17–0)

Field Descriptions:

- **op** → Opcode (5 bits)
- **I** → Immediate flag (1 bit)
- **rd** → Destination register (4 bits)
- **rs1** → Source register 1 (4 bits)
- **rs2** → Source register 2 (4 bits)
- **imm** → Modifier (2 bits) + Immediate value (16 bits)
- **offset** → Branch offset (27 bits)

Immediate Modifier Semantics:

- `00` → **Signed lower immediate:**
  - 16-bit immediate is sign-extended using its MSB.
  - Used for signed arithmetic operations.
- `01` → **Unsigned lower immediate:**
  - 16-bit immediate is zero-extended (top 16 bits of result are 0).
  - Used when immediate is a non-negative constant.
- `10` → **Upper immediate:**
  - 16-bit immediate is shifted left by 16 bits and lower 16 bits are filled with 0.
  - Used for loading higher-order values.

1.3 CSR Instruction Format (Special Instruction)

Format	Definition
csr mov	op (31–27) I=1 (26) rd (25–22) csr_mode=1 (21) read (20) set (19) clear (18) modifier (17–16) mask (15–0)

Field Descriptions:

- **op** → 5-bit opcode for `mov` (01001)

- **I** → Immediate flag; always set to 1 for CSR operations
- **rd** → Destination register (GPR)
- **csr_mode** → Set to 1 to enable CSR operation mode
- **read** → 1 = read CSR bit into flags, 0 otherwise
- **set** → 1 = set bits in CSR, 0 otherwise
- **clear** → 1 = clear bits in CSR, 0 otherwise
- **modifier** → 2-bit field, behavior depends on operation:
  - **Read / Set:**
    - `01` → Lower 16 bits (bit 15 to 0)
    - `10` → Upper 16 bits (bit 31 to 16)
  - **Clear:**
    - `00*` → Lower 16 bits (bit 15 to 0)
    - `10` → Upper 16 bits (bit 31 to 16)
- **mask** → 16-bit mask used to select specific CSR bits for the operation

* even if 00 is given but 15th bit is being cleared then it won't be able to do the sign extension, this special case is handled in hardware by checking for this case.

CSR Access Description

- **modifier** values are interpreted based on the type of operation:
  - **Read/Set:** `01` for lower 16 bits, `10` for upper 16 bits
  - **Clear:** `00` for lower 16 bits, `10` for upper 16 bits
- This mapping is managed in hardware to correctly extend and interpret the immediate value.
- The hardware ensures that the mask is aligned with the intended half (upper/lower) of the 32-bit CSR.

Read Operation

- `read=1, set=0, clear=0`
- Sends the value of the bit, shifts it to ALU result's MSB, then sets flags
- Flags:
  - EQ = 1 if bit is 0
  - GT = 1 if bit is 1
- Executed in 1 cycle; flags updated

Set/Clear Operation

- `set=1` and/or `clear=1`
- ALU performs OR/AND with the mask
- CSR updated with result
- Multiple-bit updates supported
- Executed in 1 cycle

1.4 Control Status Registers

The TinyRISC architecture includes a set of memory-mapped Control Status Registers (CSRs), accessible using the `csr mov` instruction in CSR mode. These registers are used to control peripherals, manage interrupts, and configure system-level settings.

Defined CSR Fields:

- `Control_status_register[1]`: I2C control configuration
- `Control_status_register[2]`: SPI control register
- `Control_status_register[3]`: UART control register
- `Control_status_register[4]`: Watchdog control
- `Control_status_register[5]`: Interrupt control register

Register Assignments and Bit Fields:

CSR Index	Mapped Signal	Bits Used
CSR[5]	`interrupt_control_reg`	Bits [21:0] 22 bits
CSR[4]	`watchdog_control`	Bits [4:0] (5 bits)
CSR[3]	`uart_control`	Bits [7:0] (8 bits)
CSR[2]	`SPI_control`	Bits [9:0] (10 bits)
CSR[1]	`i2c_control`	Bits [8:0] (9 bits)

Only the specified fields within each CSR are accessed or modified by software. Remaining bits are reserved unless documented otherwise and should be written as 0 and ignored on read.

2 System Control Components

2.1 Watchdog Timer

The Watchdog Timer is a critical safety mechanism used to detect processor stalls and initiate a reset if the processor fails to progress. It is composed of three main modules:

- **freq_div**: Frequency divider that generates a slow clock.
- **watchdog**: Timer that tracks inactivity and asserts reset if needed.
- **watchdog_rst_counter**: Detects program counter (PC) activity to automatically refresh the watchdog.

How It Works:

1. The `freq_div` module generates a slow clock signal (`watch_clk`) by dividing the system clock using a **3-bit prescaler**. This slows down the timing reference for the watchdog.
2. The `watchdog` module counts the number of `watch_clk` cycles since the last reset signal. If it exceeds a predefined limit (e.g., 4096 cycles) without being refreshed, it asserts the `pc_rst` signal to reset the processor.

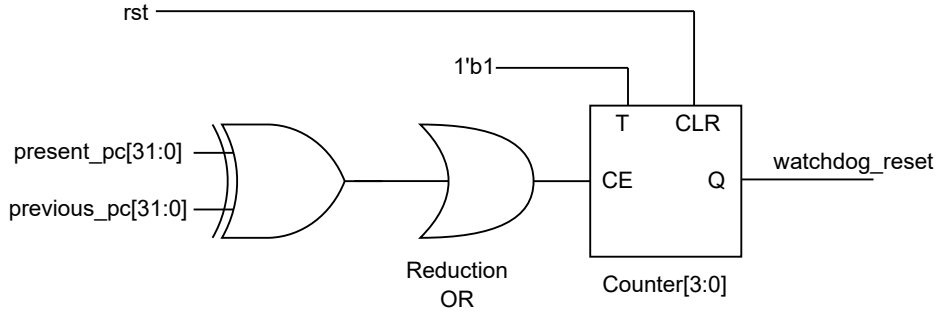


Figure 1: Block that resets the processor if it is not reset often by the processor-side module

3. The `watchdog_rst_counter` module monitors whether the PC is changing every cycle. If the PC is active (changes) for 15 consecutive cycles, it issues a one-cycle `watchdog_rst` pulse to refresh the timer.

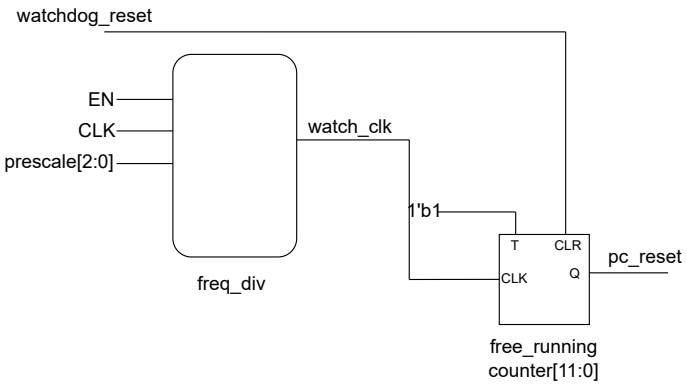


Figure 2: Block that resets the watchdog if processor is working properly

4. If the PC stops changing (i.e., the processor hangs), no refresh pulse is generated, allowing the watchdog to time out and reset the system.

Configurable Inputs and Signals:

- **Enable (`EN`)**: Activates the watchdog system. When disabled, the timer and clock divider reset.
- **Prescaler bits (`prescale_2:0`)**: Sets the frequency division rate of the system clock.
- **watchdog_rst**: Refreshes the watchdog timer when asserted (generated by PC activity monitor).
- **pc_rst**: Output signal used to reset the processor if a stall is detected.

How to Use / Configure:

- Set `EN` = 1 to activate the watchdog.
- Configure the prescaler via `Control_status_register[4][2:0]` (watchdog control CSR) to adjust the slow clock rate.

  The watchdog timeout period is calculated as:

  $$\textbf{Timeout}_{\textbf{cycles}} = \textbf{4096} \times \textbf{2}^{\textbf{prescaler}}$$

  For example, assuming a system clock of 100 MHz:

  - `prescaler` = 0 ⇒ Timeout = 4096 cycles ≈ 81.92 µs
  - `prescaler` = 7 ⇒ Timeout = 524,288 cycles ≈ 10.48 ms

  This provides flexible control over how quickly the watchdog triggers a reset if the processor becomes unresponsive.
- The processor must continuously execute instructions to ensure the PC changes every cycle. This activity automatically generates the `watchdog_rst` signal.
- To disable the watchdog, set `EN` = 0 through the CSR.

2.2 Interrupt Controller

The TinyRISC interrupt controller consists of two parts:

- **Hardware Interrupt Controller**: Evaluates interrupt request lines, prioritizes them, and asserts the corresponding interrupt pins based on enable and flag conditions.
- **Interrupt Handler Unit**: Generates the correct interrupt vector address (`pc_isr`) for the processor and asserts `interrupt_out`.

Hardware Interrupt Controller

The hardware controller supports a total of **8 interrupt sources**, divided as:

- **4 High-priority interrupt lines**
- **4 Low-priority interrupt lines**

Each interrupt has an associated flag and enable bit. These are logically ANDed to determine if an interrupt is valid. Based on which priority group the valid interrupts fall under:

- `interrupt_pin_high` is asserted if any of the high-priority interrupts are active
- `interrupt_pin_low` is asserted if any low-priority interrupt is active and no high-priority interrupt is currently active

Only one of these pins can be high at a time. The hardware does not directly encode priority order — the processor performs polling in software to determine which interrupt (among high or low) to service first.

Interrupt Handler Unit

Once an interrupt pin is asserted:

- The handler asserts `interrupt_out = 1`

- The corresponding ISR address is loaded into `pc_isr` from the Interrupt Vector Table (IVT)
- When `interrupt_out` is high, the IF-OF and OF-EX pipelines are flushed. The PC stored in `r12` will be the PC at the OF-EX pipeline stage.
- When there is a branch instruction at OF-EX, due to a control hazard, two NOPs are inserted in the IF and OF stages. In the next clock cycle, if an interrupt occurs, it may capture the wrong PC. To mitigate this, a dummy instruction and its corresponding PC are inserted into the OF-EX pipeline following the branch. This ensures that if an interrupt occurs, the correct PC is stored in `r12`.
- The processor jumps to the ISR; all remaining handling is software-driven

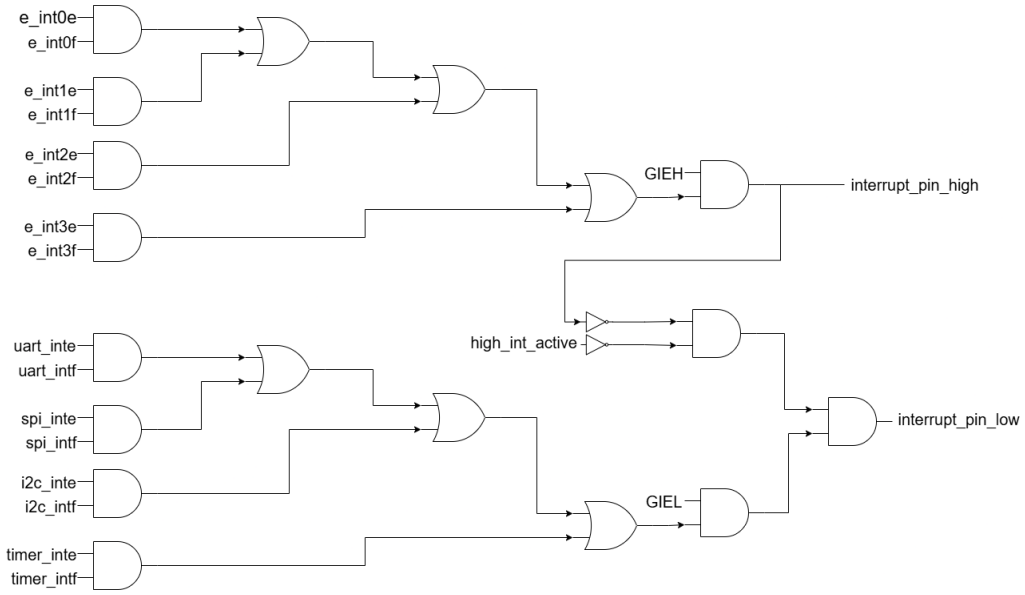


Figure 3: Hardware and handler logic for high and low priority interrupt resolution

Software Interrupt Handling

Interrupts are serviced according to the pin that is asserted, and in the following software-defined order:

1. Poll all 4 high-priority interrupt flags (in fixed priority order)
2. If no high-priority interrupt is pending, poll all 4 low-priority interrupts

At processor boot, both global interrupt enables are set:

- `GIEH` (Global Interrupt Enable High)
- `GIEL` (Global Interrupt Enable Low)

When `interrupt_pin_high` is asserted:

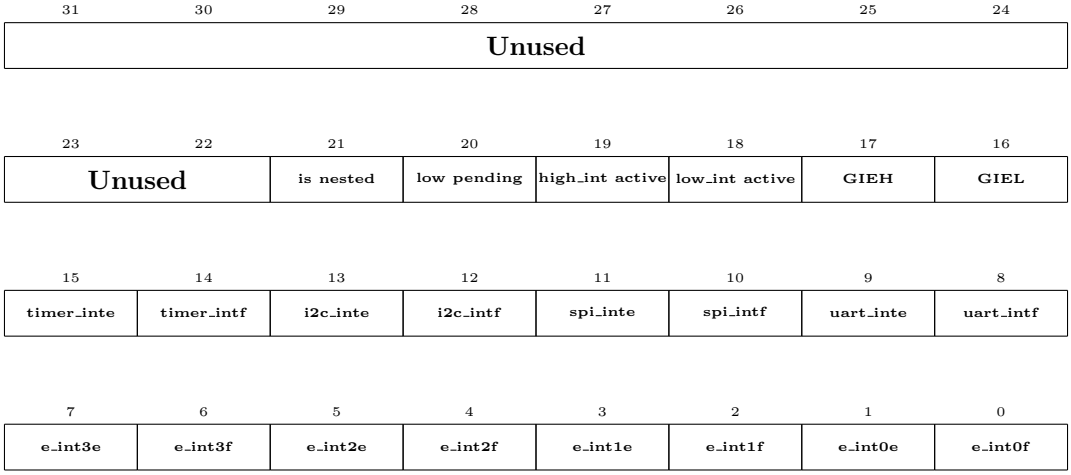
- `GIEH` and `GIEL` are cleared — no further interrupts allowed
- `high_int_active` flag is set
- Return address is pushed to `r12`
- Flags are pushed to `r13`
- Processor polls high-priority flags and jumps to the ISR
- Software pushes all necessary registers; ISR ends with `ret`
- Before `iret`, if `low_pending = 1`, the processor immediately services the pending low-priority ISR instead of returning

When `interrupt_pin_low` is asserted:

- Only `GIEL` is cleared; high interrupts remain enabled
- Current PC is saved to `r12`, `low_int_active` flag is set
- Processor polls low-priority flags and jumps to the corresponding ISR
- If a high-priority interrupt arrives during this:
  - `is_nested_interrupt` is set
  - Processor jumps to high-priority ISR after saving context
  - After servicing, only `GIEH` is re-enabled to allow completion of low-priority ISR
- At end of low ISR:
  - `r12` is popped to restore program execution but only when nested is high other wise it is not poped.
  - Both `GIEH` and `GIEL` are re-enabled
  - `iret` is executed to return from interrupt

intcon

The `int_reg` register is **22 bits** wide. It is visualized below using 4 bytefield rows for clarity, from MSB (bit 31) to LSB (bit 0). Bits [31:22] are reserved.



Field Descriptions:

- **Bits 31–24**: `Unused` — **Reserved**.
  - These bits are not used and should be written as 0.
- **Bits 23–22**: `Unused` — **Reserved**.
  - These bits are not used and should be written as 0.
- **Bit 21**: `is_nested_interrupt` — **R**.
  - Indicates the processor is currently handling a nested interrupt.
- **Bit 20**: `low_pending` — **R**.
  - Set if a low-priority interrupt is pending.
- **Bit 19**: `high_int_active` — **R**.
  - Set internally when a high-priority interrupt is being serviced.
- **Bit 18**: `low_int_active` — **R**.
  - Set internally when a low-priority interrupt is being serviced.

- **Bit 17**: `GIEH` — **RW**.
  - Global Interrupt Enable for High-priority interrupts.
- **Bit 16**: `GIEL` — **RW**.
  - Global Interrupt Enable for Low-priority interrupts.
- **Bit 15**: `timer_inte` — **RW**.
  - Enable bit for Timer interrupt.
- **Bit 14**: `timer_intf` — **R-Clr**.
  - Flag set when a timer interrupt occurs; cleared by software.
- **Bit 13**: `i2c_inte` — **RW**.
  - Enable bit for I2C interrupt.
- **Bit 12**: `i2c_intf` — **R-Clr**.
  - Flag set when an I2C interrupt occurs; cleared by software.
- **Bit 11**: `spi_inte` — **RW**.
  - Enable bit for SPI interrupt.
- **Bit 10**: `spi_intf` — **R-Clr**.
  - Flag set when an SPI interrupt occurs; cleared by software.
- **Bit 9**: `uart_inte` — **RW**.
  - Enable bit for UART interrupt.
- **Bit 8**: `uart_intf` — **R-Clr**.
  - Flag set when a UART interrupt occurs; cleared by software.
- **Bit 7**: `e_int3e` — **RW**.
  - Enable bit for External Interrupt 3.
- **Bit 6**: `e_int3f` — **R-Clr**.
  - Flag for External Interrupt 3.
- **Bit 5**: `e_int2e` — **RW**.
  - Enable bit for External Interrupt 2.
- **Bit 4**: `e_int2f` — **R-Clr**.
  - Flag for External Interrupt 2.
- **Bit 3**: `e_int1e` — **RW**.
  - Enable bit for External Interrupt 1.
- **Bit 2**: `e_int1f` — **R-Clr**.
  - Flag for External Interrupt 1.
- **Bit 1**: `e_int0e` — **RW**.
  - Enable bit for External Interrupt 0.
- **Bit 0**: `e_int0f` — **R-Clr**.
  - Flag for External Interrupt 0.

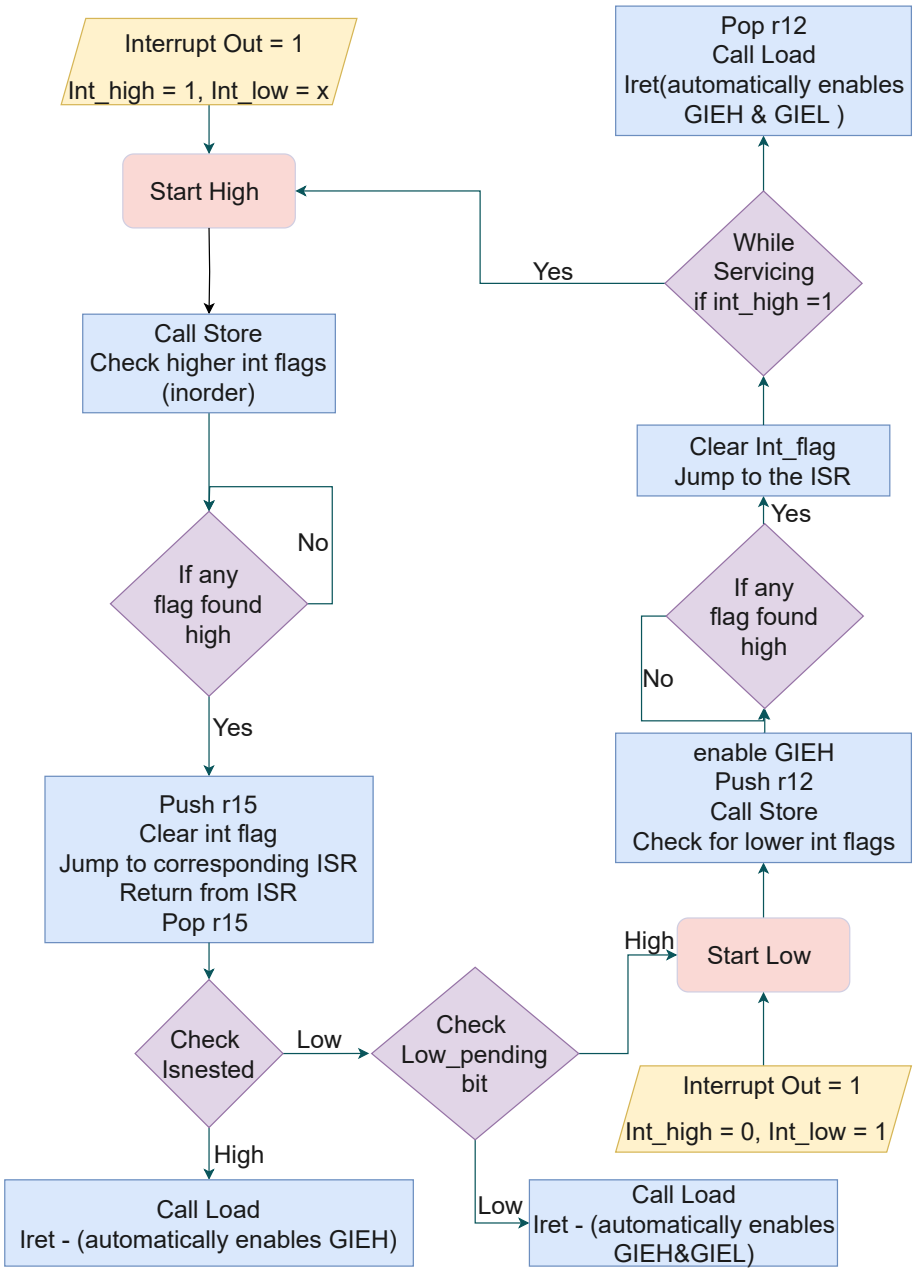


Figure 4: Software handling logic for high and low priority interrupt servicing

Interrupt Nesting

TinyRISC supports nested interrupts up to a depth of **2 levels**:

- A low-priority interrupt can be interrupted by a high-priority interrupt
- While in a high-priority ISR, all interrupts are blocked — no further nesting is possible

2.3 APB Interface

Although the Advanced Peripheral Bus (APB) is traditionally used as a *secondary bus* in ARM-based systems for connecting slower peripherals, we have adapted it as the *primary bus interface* in our embedded processor. Given the simplicity and resource constraints of our system, the APB protocol fits well as a lightweight alternative to more complex buses like AHB or AXI.

This design choice is especially suitable for small, embedded processors where area and complexity must be minimized. The APB interface offers an easy-to-implement protocol with a predictable two-phase transfer mechanism and minimal logic.

For more information on the standard APB protocol, refer to the ARM AMBA specification: AMBA APB Protocol Specification (IHI 0024).

The figure below illustrates the finite state machine (FSM) of our APB interface, which operates on the *negative edge of the clock*. The transitions depend on the 'transfer' signal and the state of the peripheral 'PREADY' signal.

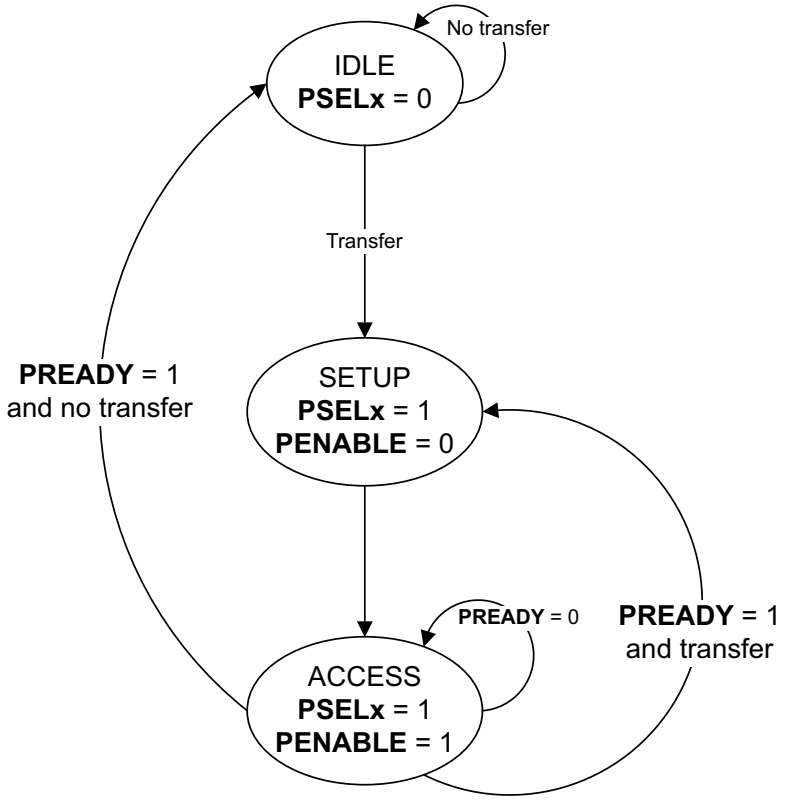


Figure 5: APB State Machine used in the processor

In our APB protocol implementation, all state transitions occur on the **negative edge** of the clock. A key control signal in this protocol is `transfer`, which determines whether a new APB transaction should be initiated. This signal is updated based on the current APB state and the presence of load/store instructions in the pipeline:

- If the current APB state is `IDLE`, then `transfer` is assigned the value of `MA_transfer`, which is high if a

load/store instruction is present in the Memory Access (MA) stage.

- If the current state is `ACCESS`, then `transfer` is assigned the value of `EX_transfer`, which reflects the presence of memory instructions still in the Execute (EX) stage.
- If the current state is `SETUP`, then `transfer` is **deasserted (set to 0)** to prepare for the current transaction.

The address (`PADDR`) is latched when the peripheral interface enters the `SETUP` state. In this implementation, the peripheral (memory) is always ready (`PREADY = 1`), so the data transfer completes in the subsequent `ACCESS` state. During this time, the processor pipeline is stalled.

We illustrate different APB scenarios below using timing diagrams and corresponding behavior descriptions.

2.3.1 Single Read

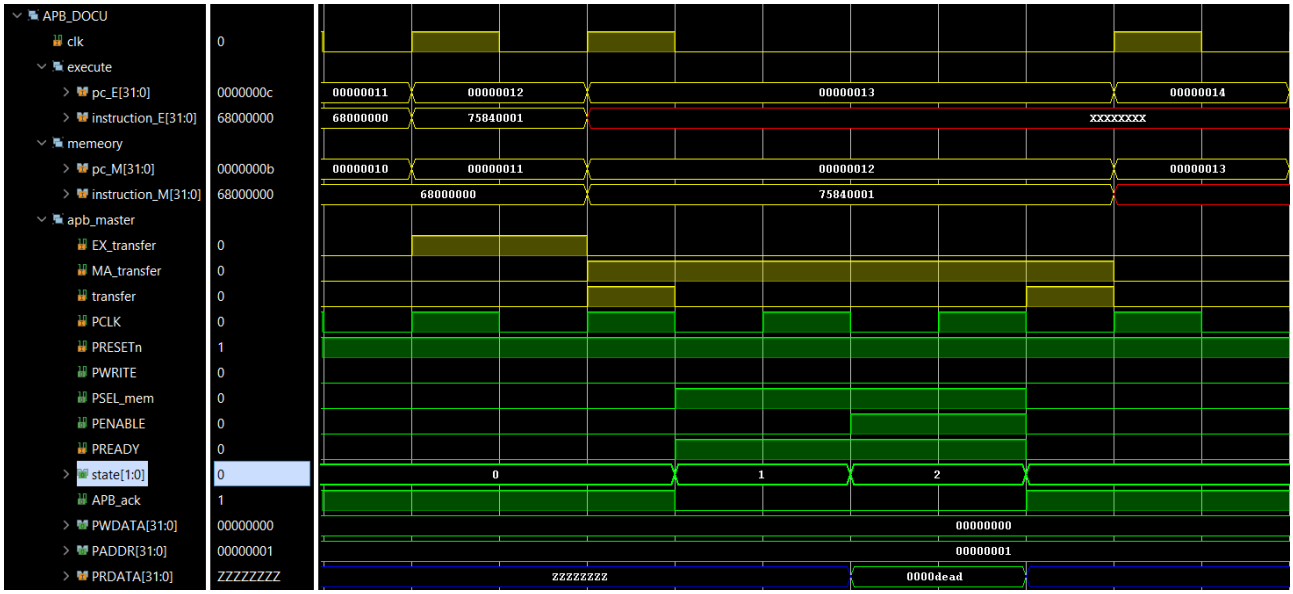


Figure 6: Single Read Transaction: Load Instruction `75840001`

- Instruction: `75840001` (load)
- On 1st negedge: `IDLE` → `SETUP`; `PADDR` is latched.
- On 2nd negedge: `SETUP` → `ACCESS`; `PRDATA` is latched with `0x0000DEAD`.
- On 3rd negedge: `ACCESS` → `IDLE`.
- Total stall: 3 cycles while the instruction is in the MA stage.

2.3.2 Single Write

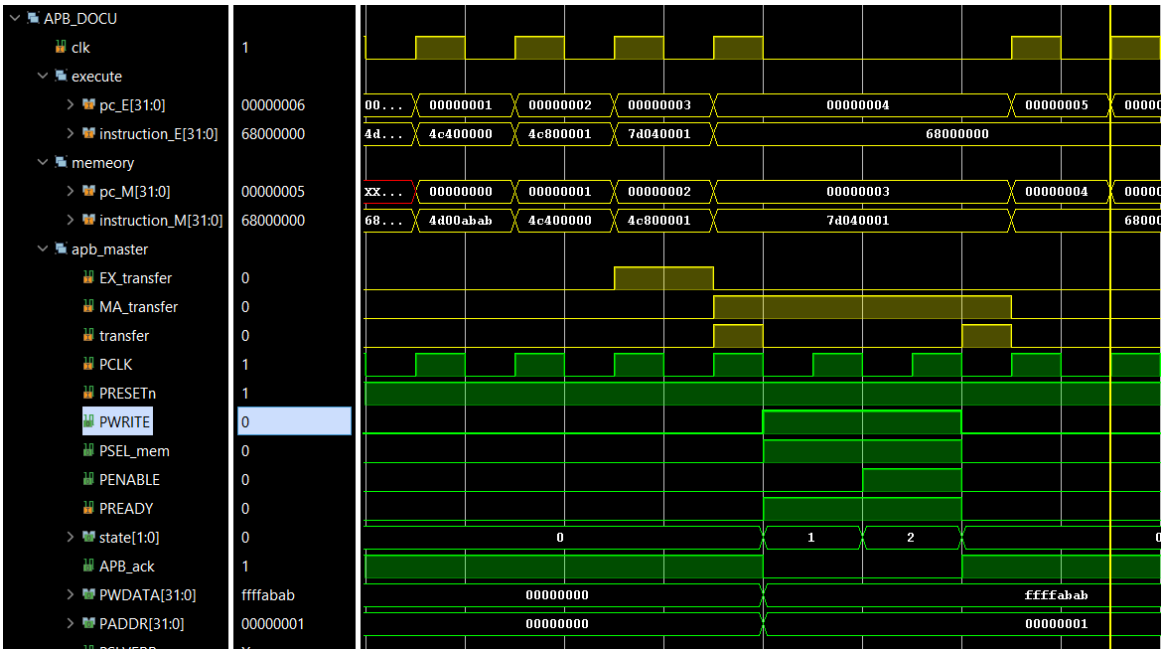


Figure 7: Single Write Transaction: Store Instruction `7D040001`

- Instruction: `7D040001` (store)
- On 1st negedge: `IDLE` → `SETUP`; both `PADDR` and `PWDATA` are latched.
- On 2nd negedge: `SETUP` → `ACCESS`.
- On 3rd negedge: `ACCESS` → `IDLE`.
- `PWDATA` is latched at the first posedge after the instruction enters the MA stage.

2.3.3 Multiple Reads

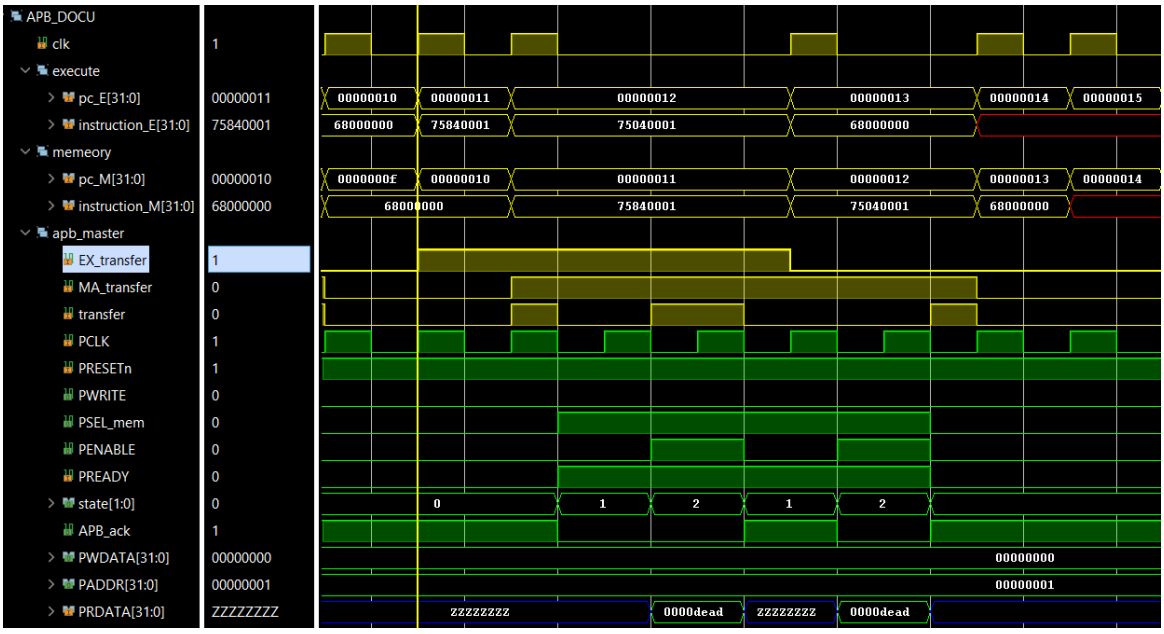


Figure 8: Multiple Consecutive Reads: Instructions `75840001`, `75040001`

- Instructions: `75840001`, `75040001` (both loads)

- Follows the same behavior as a single read for the first instruction.
- On 3rd negedge: Instead of transitioning to `IDLE`, the presence of the second load keeps `transfer` high, so `ACCESS` → `SETUP`.
- `PRDATA` is latched at the 2nd and 4th negedges respectively for both loads.

2.3.4 Multiple Writes

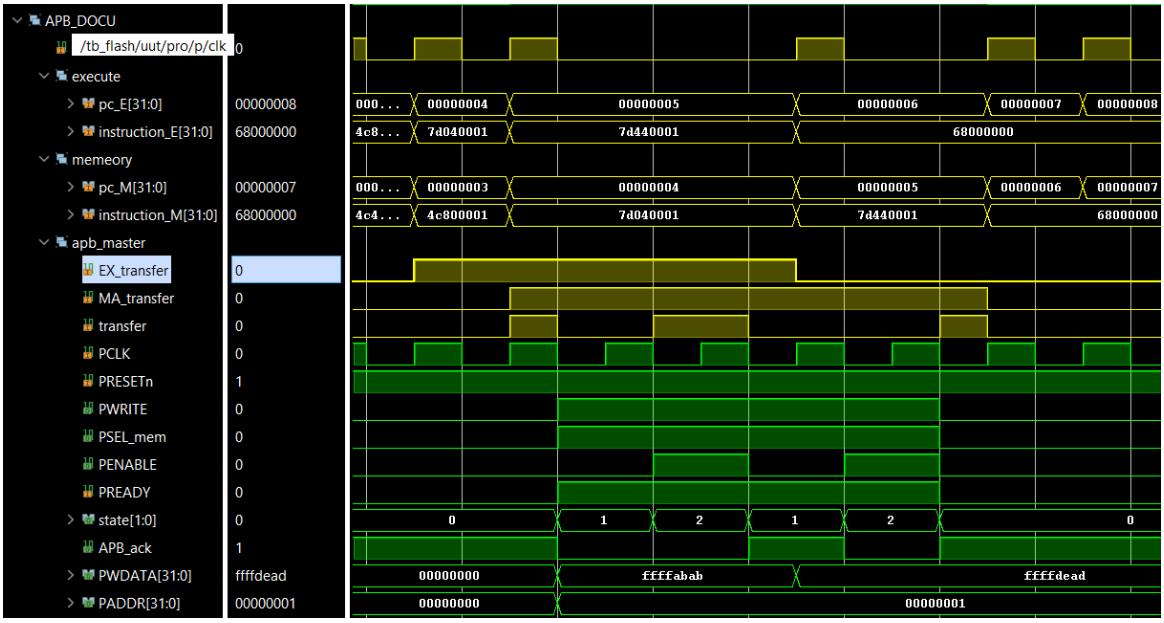


Figure 9: Multiple Consecutive Writes: Instructions `7D040001`, `7D440001`

- Instructions: `7D040001`, `7D440001` (both stores)
- Similar to the multiple read case in control flow.
- `PWDATA` is latched at the 1st and 4th posedges, corresponding to each store instruction (roughly half a cycle after the 0th and 3rd negedges).

2.3.5 Read Followed by Write

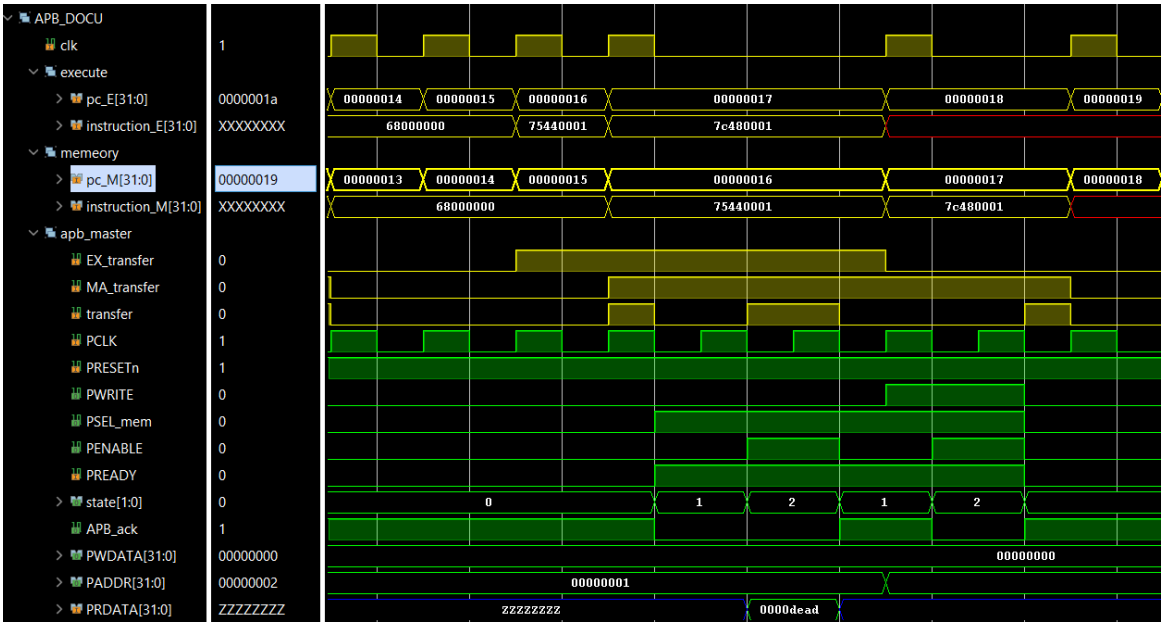


Figure 10: Read Followed by Write: Load `75440001`, Store `7C480001`

- Instructions: `75440001` (load), `7C480001` (store)
- `PRDATA` is latched at the 2nd negedge for the load.
- `PWDATA` is latched at the 4th posedge for the store.
- Demonstrates correct pipelined execution of mixed memory operations.

2.3.6 Pin and Signal Interface

The figure below shows the APB pin-level interface used between the processor core and APB peripherals:

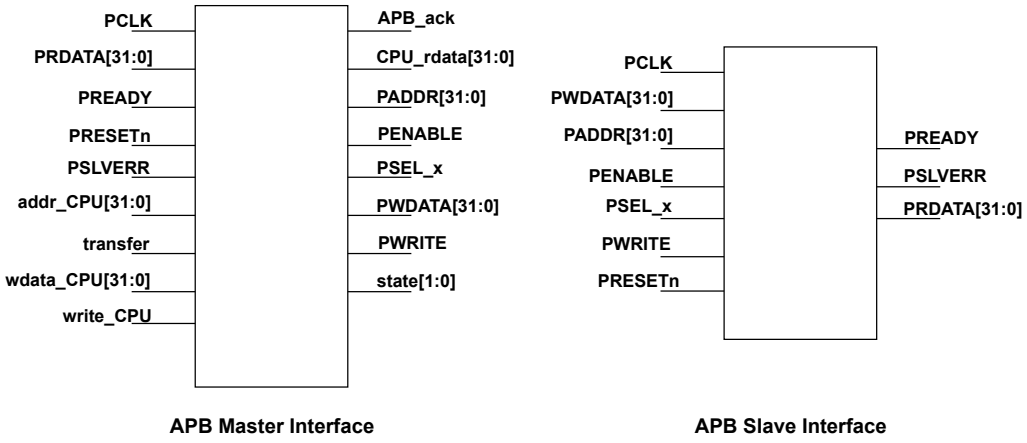


Figure 11: APB Master-Peripheral Signal Interface

The following table summarizes the APB signals and custom controller signals used in our design. Inputs and outputs are specified with respect to the APB *master (processor)* and *peripheral (slave)*. In addition to standard APB signals:

- `transfer` is generated in the APB controller using the pipeline's `MA_transfer` and `EX_transfer` signals. It controls state transitions.
- `APB_ack` is used to notify the processor to resume execution, especially when `PREADY = 0` and the pipeline is stalled.
- `state[1:0]` reflects the current state of the APB FSM (IDLE, SETUP, ACCESS).

Signal	Direction	Description
Master (Processor) Side		
PCLK	Input	APB clock, shared by all peripherals and controller
PRESETn	Input	Active-low reset for the APB interface
PRDATA[31:0]	Input	Data read from the peripheral during a read transaction
PREADY	Input	Indicates peripheral is ready to complete the transaction
PSLVERR	Input	Slave error response, indicates failed transaction
addr_CPU[31:0]	Input	Address from CPU to initiate a load/store operation
wdata_CPU[31:0]	Input	Write data from CPU for store instructions
transfer	Input	Controller-generated signal to initiate a new APB transfer
state[1:0]	Output	Current APB controller FSM state (IDLE, SETUP, ACCESS)
APB_ack	Output	Acknowledge signal to CPU to resume pipeline after stall
CPU_rdata[31:0]	Output	Read data returned to the CPU during a load
PADDR[31:0]	Output	Address placed on the APB for the transaction
PWDATA[31:0]	Output	Write data placed on the bus for a store operation
PWRITE	Output	Indicates if the operation is a write (1) or read (0)
PSEL_x	Output	Select signal to enable the targeted peripheral
PENABLE	Output	Indicates the second phase of the APB transaction
Peripheral (Slave) Side		
PCLK	Input	APB clock signal from master
PRESETn	Input	Global APB reset
PADDR[31:0]	Input	Address bus from master to specify register
PWDATA[31:0]	Input	Write data sent from master to peripheral
PWRITE	Input	Operation type: write (1) or read (0)
PSEL_x	Input	Peripheral is selected by the master
PENABLE	Input	Indicates active data phase
PRDATA[31:0]	Output	Data returned to master in case of a read
PREADY	Output	Indicates peripheral is ready to respond
PSLVERR	Output	Indicates an error in transaction execution

Table 1: APB Master–Slave Signal Descriptions

2.3.7 Summary

- Each APB transaction typically spans 3 cycles: IDLE → SETUP → ACCESS → IDLE.
- When there is a back to back APB transaction, it spans 3+2n (n = total number of back to back transfers - 1) cycles: IDLE → SETUP → ACCESS → SETUP → ACCESS → IDLE.
- When multiple memory operations are issued back-to-back, the transfer signal enables pipelining by preventing the FSM from returning to IDLE.
- All APB control and data signal transitions respect the negedge-based FSM and posedge-based latching of data/addresses.

3 Peripherals

3.1 Pulse Width Modulation (PWM) Peripheral

Introduction

Pulse Width Modulation (PWM) is a technique to control the width (duty) of digital pulses while maintaining a constant frequency. Optionally, both duty and frequency can be modulated. PWM is commonly used to:

- Control the brightness of LEDs.
- Drive and control the speed of DC motors.
- Generate analog-like signals such as sine waves via filtering.

PWM Module Architecture

The PWM peripheral includes the following sub-modules:

1. **Prescaler (prescaler.v)**: A programmable clock divider. It gates the incoming clock with `en_prescaler` and uses a **5-bit** `prescale_value` to select a divided clock ($\frac{f_{clk}}{2^{(prescale+1)}}$). Supports divisions from 2^1 to 2^{32}. When disabled, acts as a bypass.
2. **Period Register (period_reg.v)**: A 12-bit terminal-count register (`PR`) loaded via `pr_in`. On reset, initializes to 0xFFFF. Determines the PWM period.
3. **Timer (tmr_cnt.v)**: A 12-bit counter (`TMR`) that resets to 0 on `rst` or when `EN_TMR=0`. It increments on each rising clock edge and rolls over on reaching `PR`.
4. **Comparator B (comparator_b.v)**: Continuously compares `TMR` and `PR`. When equal, sets `S=1` to trigger PWM rollover.
5. **Duty Cycle Registers**: - **Master (duty_reg_master.v)**: Stores input `duty_cycle_in` (or 0 on reset). - **Slave (duty_slave.v)**: On rollover, captures master register value into `duty_cycle_out` for glitch-free update.
6. **Comparator A (comparator_a.v)**: Compares `TMR` with `duty_cycle_out`. When equal, sets `R=1` to reset the PWM output.
7. **SR Latch (sr_latch.v)**: Holds PWM signal based on comparator outputs. - `R=1` → Output low - `S=1` → Output high - Else → Hold

Specifications

- **PWM Frequency Formula**:

$$f_{PWM} = \frac{f_{clk}}{2^{(\texttt{prescale}+1)} \cdot (\texttt{Period} + 1)}$$

  For a 100 MHz clock, the minimum possible frequency is approximately **5.68 µHz**.

- **Duty Cycle Formula**:

$$\text{Duty } \% = \left(\frac{\texttt{Duty\_reg\_value}}{\texttt{Period} + 1} \right) \times 100$$

  The duty cycle can range from **0% to 100%**.

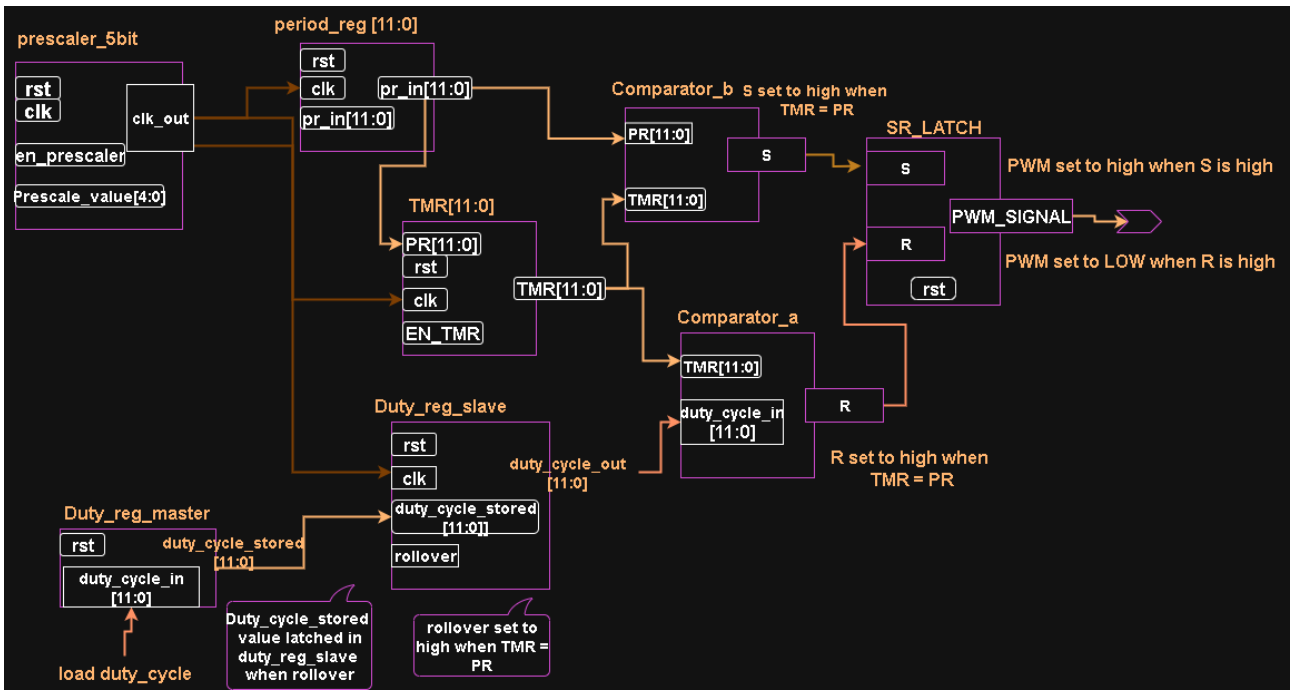


Figure 12: PWM Output Waveform or Architecture

PWM Control Register (32-bit)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
-	DC_in												PR												PSC					TE	PE

Bit Fields Explanation:

- **PE (PWM Enable)**: Bit 0 – Enables the PWM module. Corresponds to `pwm_reg[0]`.
- **TE (Timer Enable)**: Bit 1 – Enables the internal timer. Corresponds to `pwm_reg[1]`.
- **PSC (Prescaler)**: Bits [6:2] – Sets the prescaler value to divide the input clock. Corresponds to `pwm_reg[6:2]`.
- **PR (Period Register)**: Bits [18:7] – Sets the period of the PWM signal. Corresponds to `pwm_reg[18:7]`.
- **DC_in (Duty Cycle Input)**: Bits [30:19] – Defines the duty cycle of the PWM signal. Corresponds to `pwm_reg[30:19]`.
- **- (Unused)**: Bit 31 – Reserved/unused.

3.1.1 How to Use PWM

The PWM peripheral is memory-mapped at address `0x00010000`. It is used to generate a pulse-width modulated signal, where the duty cycle and period are programmable via a control register. The processor writes the PWM control word to this address through the APB interface.

Example:

```
pwm : ; PWM address is 0x00010000
    org 0x00000000:
    movu r2,0x31AC          ; Load upper part of control word
    addh r2,r2,0x000A       ; Final control word = 0x31ACA

    movu r1,0x0001
    addh r1,r1,0x0000       ; r1 = 0x00010000 (PWM address)

    st r2,0[r1]             ; Write control word to PWM register
```

This example sets up the PWM module by writing a 20-bit control word (`0x31ACA`) to its memory-mapped register. The control word typically encodes settings such as duty cycle, period, and enable bits (based on the design of the PWM control format).

3.2 UART Module Description

3.2.1 UART Inputs

Signal	Width	Description
EN_50MHz	1 bit	Enables the internal clock path for 50MHz operation. Used in baud rate generation. If 0, internal 100MHz clock is used.
PENABLE	1 bit	APB protocol signal: Enables the data phase transfer.
PCLK	1 bit	APB clock input. All APB interface logic is synchronized to this clock.
PRESETn	1 bit	Active-low asynchronous reset. Resets the internal state of the UART module.
PWRITE	1 bit	APB control signal: High for write operations, low for read.
PSEL	1 bit	APB select signal: High when peripheral is selected for an APB transaction.
TXEN	1 bit	UART transmit enable: Enables baudrate generation in TX logic.
SPEN	1 bit	UART serial port enable: Enables UART functionality globally.
RX	1 bit	UART receive line input: Captures serial data from external source.
RXEN	1 bit	UART receive enable: Enables baudrate generation in RX logic.
PWDATA	32 bits	APB write data bus: Data/control info written to UART.
BAUD_SEL	2 bits	Selects one of four predefined baud rates (9600, 19200, 57600, 115200).
uart_control	2 bits	UART word length control (8, 16, 24, or 32 bits).

3.2.2 UART Output Ports

Signal	Width	Description
TXIF	1 bit	Transmit interrupt flag / APB PREADY: Indicates readiness for APB transactions.
RXIF	1 bit	Receive interrupt flag: Data has been received and is ready.
TX	1 bit	UART transmit line output: Sends serial data externally.
PRDATA	8 bits	APB read data output: Provides received data to APB master.

3.2.3 UART Control Register

Bit-level description:

7 6	5	4 3	2	1	0
DATA CONTROL	SPEN	BAUDSEL	RXEN	TXEN	CS

- **Bit 0 – Clock Select**: Selects system clock for UART:
  - `0` = 100 MHz
  - `1` = 50 MHz
- **Bit 1 – TXEN (Transmit Enable)**: Enables the transmitter. Uses the selected baud rate.
- **Bit 2 – RXEN (Receive Enable)**: Enables the receiver. Uses the selected baud rate.
- **Bits 4:3 – BAUDSEL (Baud Rate Select)**: Selects baud rate:
  - `00` = 9600
  - `01` = 15200
  - `10` = 57600
  - `11` = 115200

- **Bit 5 – SPEN (Serial Port Enable)**: Enables the serial port. Must be set for TX/RX to function.
- **Bits 7:6 – DATA CONTROL**: Selects number of bytes per word:
  - `00` = 1 byte
  - `01` = 2 bytes
  - `10` = 3 bytes
  - `11` = 4 bytes

3.2.4 Transmitter

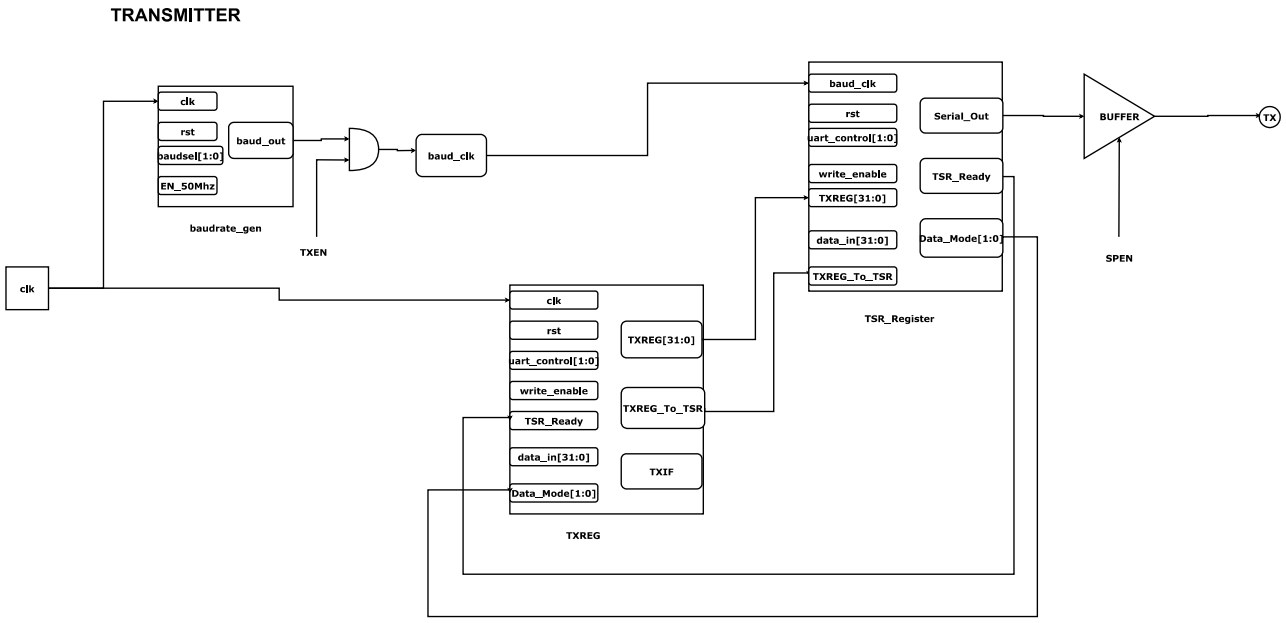


Figure 13: UART Transmitter architecture

Baudrate Generator Logic

- `baud_sel` = **2'b00 (9600 baud)**: Full divider chain used: mod3 + 4×mod2 + mod7 + mod31
- `baud_sel` = **2'b01 (19200 baud)**: One mod2 is skipped; the rest of the divider chain is the same as 9600.
- `baud_sel` = **2'b10 (57600 baud)**: mod2 and mod3 stages are skipped.
- `baud_sel` = **2'b11 (115200 baud)**: Only one mod2, mod7, and mod31 stages are used.

Divider Chain Structure (9600):

```
clk → [modu3] → [modu2] → [modu2] → [modu2] → [modu2] → [modu7] → [modu31] → baud
```

Selective bypassing of divider stages based on baud_sel enables different baud rates with minimal logic.

TXREG Block

- Loads data from PWDATA into TXREG when TXIF and write_enable are high.
- 1-clock delay before TXIF is cleared.
- TXIF auto-reasserts for 8-bit mode.
- For multi-byte modes: TXIF depends on TSR_Ready, uart_control match, and mode[2] = 0.
- TXREG_to_TSR_en manages transfer handshake.

Shift Register Block

- FSM transmits bytes: TXREG[7:0], [15:8], [23:16], [31:24].
- On falling clk_ext: byte loaded into TX shift register.
- On rising clk: right-shift TX_shift_reg, output LSB.
- After 10 shifts (start + 8 data + stop), TSR returns to idle.
- TSR_Ready indicates transmitter is idle.

3.2.5 Receiver

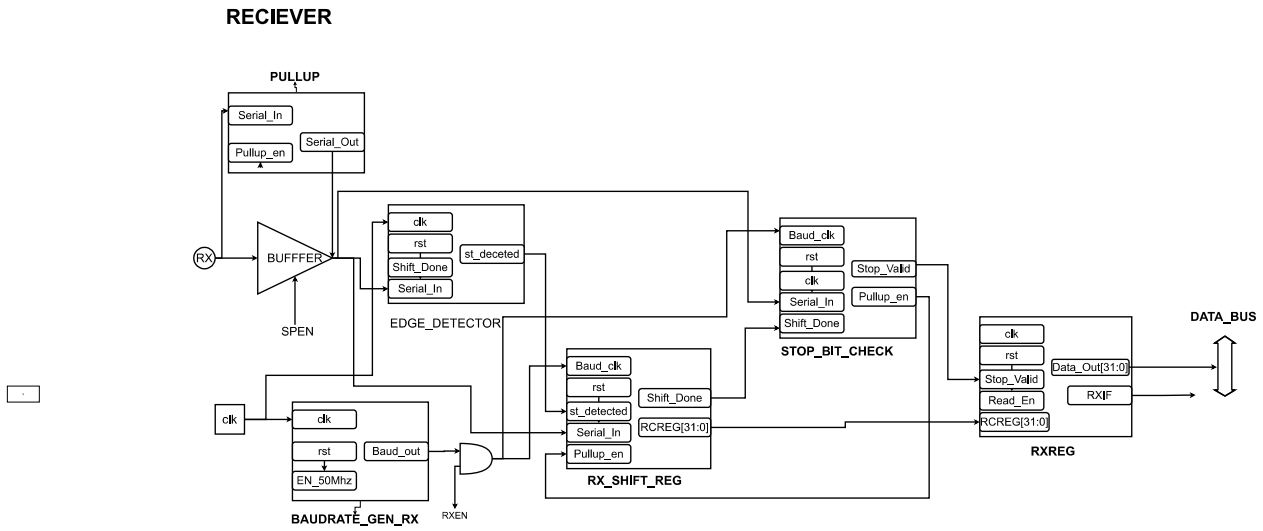


Figure 14: UART Receiver architecture

Baud Rate Generator

- Fixed at 9600 baud; not configurable.

Edge Detector

- Detects start bit (logic 0) on RX line.
- Sets `st_bit_detected` high until shift is done.

UART Shift Register

- Shifts in 8 bits of serial data on rising baud clock.
- Uses `st_bit_detected` to start.
- Asserts `shift_done` after receiving full byte.

Checking Stop Bit

- Checks if stop bit is logic 1 after data reception.
- If valid: asserts `stop_valid`.
- If invalid: asserts `pullup_en`.

Signal Name	Width	Description
stop_valid	1 bit	High when valid stop bit is detected.
pullup_en	1 bit	High on framing error (invalid stop bit).

RCREG

- Stores received data only if stop bit is valid.
- Asserts `RXIF` when data is ready to be read.
- Clears `RXIF` on `read_en`.
- Buffers received UART data.

Pull-Up Module

- Forces RX line high if stop bit is invalid.
- Uses tri-state buffers based on `pullup_en`.
- Restores idle condition on RX line.

3.2.6 How to Use UART

UART is mapped to address `0x00010003`. Configuration steps include:

- Enabling TX/RX via `set uart` with appropriate control byte.
- Computing the UART address using `movu` and `addh`.
- Writing the data to be transmitted using `st`.

Example:

```
UART : ; UART address is 0x00010003
    org 0x00000000:
    set uart 1,2,5,6,7  ; TXEN = 1, control = 11 (32-bit), RX enabled
    movu r1,0x0001
    addh r1,r1,0x0003   ; UART address = 0x00010003
    mov r2,0xF0F0       ; Data to send
    st r2,0[r1]         ; Send data via UART
```

3.3 SPI Module Implementation

Inputs (All are wires by default):

- `clk` (1 bit, wire): System clock.
- `rst` (1 bit, wire): Asynchronous reset.
- `PWDATA` (32 bits, wire): Data from APB write channel.
- `PWRITE` (1 bit, wire): APB write control signal.
- `PENABLE` (1 bit, wire): APB enable signal.
- `PSEL` (1 bit, wire): APB peripheral select.
- `miso` (1 bit, wire): Master-In Slave-Out data line (input from slave).
- `control_reg` (11 bits, wire): Control and configuration register.

Outputs:

- `mosi` (1 bit, reg): Master-Out Slave-In data line (output to slave).
- `interrupt_spi` (1 bit, reg): Interrupt signal set after successful transmission.
- `control_reg_out_bo` (3 bits, reg): Status flags output (overflow and buffer full flags).
- `PREADY` (1 bit, reg): Indicates SPI is ready for the next APB transaction.
- `PRDATA` (32 bits, wire): Read data sent to APB from the SPI buffer.

Inouts (All are wires by default):

- `cs` (1 bit, wire): Chip Select line (driven in master mode, input in slave mode).
- `sclk` (1 bit, wire): Serial Clock line (driven in master mode, input in slave mode).

3.3.1 FSM and Module Coordination in SPI

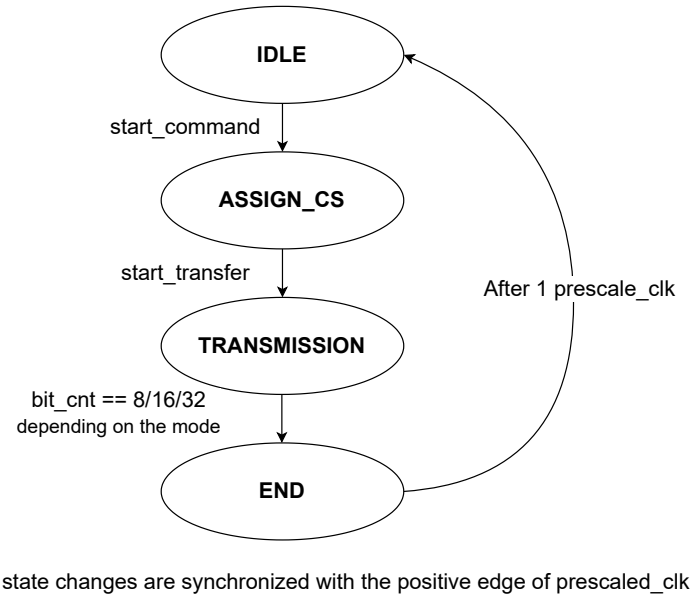


Figure 15: SPI Module State Machine

The SPI module operates using a 4-state finite state machine (FSM): `IDLE`, `ASSIGN_CS`, `TRANSMISSION`, and `END`. This FSM governs the entire data flow — from command initiation to SPI transfer and interrupt generation.

State transitions and data operations are performed on the prescaled clock, which is generated by the prescaler module based on control bits. The prescaler reduces the system clock frequency to match SPI timing requirements.

The `phase` signal toggles with each `prescale_clk` edge and is used to distinguish between actions performed at the positive edge (e.g., shift in MISO, FSM transition) and the negative edge (e.g., driving MOSI) — enabling CPHA-style behavior.

Some control logic, like APB interface handling, buffer management, and interrupt generation, runs on the system clock (`clk`), allowing synchronization with the host system.

Three submodules are instantiated to offload specialized tasks:

- `prescaler`: Divides the system clock as per `control_reg[4:2]`, generating `prescale_clk`.
- `sclk_controller`: Generates the SPI serial clock (`sclk`) based on CPOL/CPHA and master/slave mode.
- `cs_controller`: Manages chip select (`cs`), detects start conditions, and coordinates slave enablement.

This modular approach keeps the design scalable, timing accurate, and protocol-compliant.

3.3.2 Data Buffering System in SPI Module

The SPI module uses a structured 3-stage data buffering system to separate write and read operations and ensure smooth data transfer:

1. **Write Data Flow (APB to SPI)**
   When the APB issues a write command (`PWRITE = 1`, `PSEL = 1`, `PENABLE = 1`), the 32-bit data from `PWDATA` is latched into the internal write buffer named `buffer`.
   The signal `start_command` is set, initiating the SPI FSM.
   In the `IDLE` → `ASSIGN_CS` transition (posedge `prescale_clk`), the `buffer` value is loaded into `shift_reg` for transmission to the SPI slave.

2. **SPI Transmission (Shift Register Behavior)**
   In the `TRANSMISSION` state, `shift_reg` operates as a shift register:
   On posedge of `prescale_clk` and when `phase = 1`, the `MISO` bit is shifted in:

```
shift_reg <= {shift_reg[30:0], miso};
```

On negedge of prescale_clk (phase = 0), the MSB of shift_reg is driven out to MOSI:

```
mosi <= shift_reg[31];
```

The number of bits shifted (8, 16, or 32) is determined by control_reg[6:5].

3. **Post-Transmission Data Storage**
   When all bits are transmitted (bit_cnt reaches 7, 15, or 31), the FSM moves to the END state.
   At this point:
   - The final contents of shift_reg (which now holds received MISO bits) are moved to buffer_1.
   - The existing buffer_1 value is moved to buffer_2, implementing a read data pipeline.

3.3.3 Control Register Output Status Bits

The module provides status feedback via control_reg_out_bo[10:8]:

- control_reg_out_bo[8] (Overflow Flag): Set if a third read occurs before the previous buffers are emptied.
- control_reg_out_bo[9] (Buffer Full 1): Set when buffer_1 holds unread data.
- control_reg_out_bo[10] (Buffer Full 2): Set when both buffer_1 and buffer_2 are filled, i.e., no read occurred since last two transmissions.

These flags are cleared during an APB read operation (PWRITE = 0) depending on which buffer is read. The logic in the always @(posedge clk) block updates the status bits based on this.

3.3.4 Interrupt Generation

The output interrupt_spi is asserted in the END state, indicating that a new word has been received and the buffer(s) have been updated.

This allows the host to respond asynchronously (e.g., via an interrupt service routine) and read the updated data.

3.3.5 Read Data Flow (SPI to APB)

The host reads received data through the PRDATA output.

If a read occurs (PWRITE = 0) while PREADY = 1:

- If only buffer_1 is filled (control_reg[9] = 1 and control_reg[10] = 0), it outputs buffer_1.
- If both buffers are full, it outputs buffer_2.

The assign statement controlling PRDATA ensures that only valid data is placed on the APB read bus.

3.3.6 APB Protocol and PREADY Signal

In the APB protocol:

- PREADY = 1 indicates the slave is ready to accept a write or return a read.
- PREADY = 0 tells the bus master (e.g., processor) to wait and retry the transaction later.

Thus, PREADY acts as a handshake acknowledgment from the SPI module to the bus interface.

3.3.7 When is PREADY Set and Cleared?

PREADY is controlled in the always @(posedge clk) block based on the FSM state and the start_command signal:

1. In the IDLE state:
   If no transfer is ongoing (start_command == 0), PREADY is set to 1.
   This tells the APB master that the SPI is ready to accept a new write (PWDATA) or return read data (PRDATA).
   This is also when read requests are served via the PRDATA output, and buffer/status bits may get cleared.

2. When a transfer is about to begin (`start_command == 1`):
   `PREADY` is immediately set to 0, indicating that the module is busy handling the SPI transfer and cannot respond to new APB requests.

3. In all other FSM states (`ASSIGN_CS`, `TRANSMISSION`, `END`):
   `PREADY` remains low (0), maintaining bus synchronization.

4. At the end of a transmission (`END` state):
   Once the FSM returns to `IDLE`, `PREADY` is reasserted (set to 1), signaling readiness for the next APB transaction.

3.3.8 Importance of Proper PREADY Control

Why is This Important?

Proper control of `PREADY` ensures:

- No write happens while SPI is still busy shifting out previous data.
- No read occurs before fresh data is valid.
- Synchronization between asynchronous APB transactions and sequential SPI protocol timing.
- Prevents data corruption and maintains protocol compliance.

In a real-world SPI implementation, using both the posedge and negedge of `SCLK` to drive logic—such as sampling and shifting the same shift register—can lead to conflicting behavior. If both edges control the shift register simultaneously, there's a risk of data corruption or race conditions, especially if the same register is written from two different clock edges. Moreover, such dual-edge-triggered logic is generally not synthesizable in most FPGA or ASIC tools, as it creates ambiguity in timing and resource allocation.

To avoid this, the current design uses a single-edge-controlled clock (`prescale_clk`) that is internally faster than `SCLK` and toggles a `phase` signal. This `phase` controls whether the current half-cycle performs input sampling or output shifting, achieving safe, deterministic behavior without relying on both edges of `SCLK`.

3.3.9 Need for `prescale_clk` When `SCLK` Exists

To manage precise timing for data transmission and reception, the SPI module uses a signal named `phase`, which toggles on every positive edge of the `prescale_clk`. This `prescale_clk` is a divided-down version of the system clock and operates at twice the frequency of the generated `SCLK` signal.

The `phase` signal effectively creates two internal sub-cycles per `SCLK` cycle:

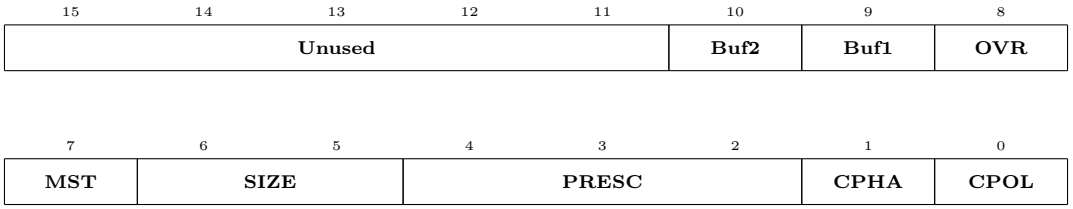
- When `phase` is high, the module handles tasks such as state transitions and sampling incoming data (`MISO`) into the shift register.
- When `phase` is low, it drives the next data bit onto the `MOSI` line.

This clean separation ensures that data is always driven and sampled in different half-cycles, avoiding race conditions. If only `SCLK` had been used with both its posedge and negedge for triggering logic (e.g., sampling and shifting), it would introduce timing conflicts—since the same clock edge might be used simultaneously by both the master and slave for different operations.

The use of a faster internal clock (`prescale_clk`) along with `phase` provides finer control, prevents such conflicts, and ensures reliable communication across all SPI modes defined by `CPOL` and `CPHA`.

3.3.10 Explanation of `control_reg` Bits (11 bits total)

Bit-level description:



You gave a quick overview in comments — but for completeness, you may want to fully document:

- `control_reg[0]`: **CPOL**
- `control_reg[1]`: **CPHA**
- `control_reg[4:2]`: **Prescaler value** (for `SCLK` division)
- `control_reg[6:5]`: **Transfer size** — 8, 16, or 32 bits
- `control_reg[7]`: **Master (1) or Slave (0)**
- `control_reg[8]`: **Overflow flag (writable/clearable?)**
- `control_reg[9]`: **Buffer 1 full flag**
- `control_reg[10]`: **Buffer 2 full flag**

Out of all these, only `control_reg[10:8]` are set by the SPI module. All other bits come from the processor.

Note: This overflow bit, when high, does not stop further data transfer — it is actually an indication to the programmer that some data is lost as the interrupt was not serviced properly.

3.3.11 Master and Slave Mode

In master mode (`control_reg[7] = 1`), the SPI module generates the `SCLK` and controls the `CS` (chip select) line. It initiates communication by actively driving these signals and managing the transmission process. In contrast, in slave mode (`control_reg[7] = 0`), the module does not drive `SCLK` or `CS`; instead, it waits for an external master to assert `CS` and provide the clock signal.

To support both roles, the `sclk` and `cs` lines are declared as `inout` and are only driven when the module is in master mode. In slave mode, these signals are placed in a high-impedance (tristate) state, allowing an external master to control them without contention. This tristate behavior is essential for ensuring safe and conflict-free operation when multiple devices share the SPI bus.

It also needs to be noted that in our implementation the master can work on all the four clock modes but the slave can only work in mode `cpol = 0` & `cpha = 0`.

3.3.12 Why Two Buffers Are Used for Reading Data

The SPI module includes two buffers during read operations to prevent data loss in full-duplex communication. When a transfer completes, the received data is stored in `buffer_1`, and an interrupt is triggered to notify the APB side.

However, if a second SPI transfer occurs before the APB master services the first interrupt and reads the data, and since `PREADY` is low during this time (indicating the interface is not yet ready), the newly received data cannot overwrite `buffer_1` without risking loss. To handle this, the second received data is stored in `buffer_2`. This ensures both data words are preserved even if the APB interface hasn't completed the previous read transaction, maintaining reliable data integrity across consecutive SPI transfers.

3.3.13 Default Reset Behavior

Clarify what happens at `rst = 1`:

- FSM goes to `IDLE`
- Buffers and control bits cleared
- `PREADY` = 1

3.3.14 Implementation of `SCLK`

The SPI hardware is structured in such a way that, for SPI **Mode 0** (`CPOL = 0, CPHA = 0`), it appears to the slave that data toggling happens on the falling edge of `SCLK` and sampling occurs on the rising edge — as expected for that mode. However, for other SPI modes like **Mode 3** (`CPOL = 1, CPHA = 1`), this behavior must be reversed: toggling should occur on the rising edge, and sampling on the falling edge.

Since our design does not internally use the edges of `SCLK` for logic control, it instead generates the `SCLK` output purely for the slave's reference, while internally relying on a faster `prescale_clk` and adjusted control logic. This allows the SPI slave to perceive correct edge behavior according to the selected mode, even though the hardware itself is not synchronized directly to the actual rising or falling edge of `SCLK`. This abstraction ensures SPI compliance while keeping the internal circuit clean, synthesizable, and free of clock edge conflicts.

The sclk_controller is responsible for generating the serial clock (SCLK) in master mode while ensuring compatibility with external clocking in slave mode. It takes several control inputs: cpol (clock polarity), cpha (clock phase), master_slave (mode select), rst, and the internal prescale_clk. The output sclk is declared as inout because its behavior changes depending on the mode of operation.

Inside the module, a clock signal is toggled on every positive edge of prescale_clk, producing a square wave at half the frequency — which forms the base for SCLK. The signal edge_selector is computed as cpol ^ cpha to align with the SPI mode and determine whether the clock should be phase-shifted or not.

The sclk line is assigned using a conditional drive:

```
assign sclk = (master_slave) ? (clock ^ edge_selector) : 1'bz;
```

This ensures that in master mode (master_slave = 1), the module actively drives the sclk line with the appropriate waveform and phase shift, depending on the selected SPI mode. In slave mode (master_slave = 0), sclk is placed in a high-impedance (1'bz) state, allowing an external SPI master to drive the line without contention.

3.3.15 cs_controller Module Implementation

Port List **Inputs:** rst, prescale_clk, PWRITE, phase, state[1:0], master_slave, cpha, cpol
Outputs: slave_enable, start_transfer
Inouts: cs

Master Mode (master_slave = 1)

- Drives the cs line using an internal cs_out signal.
- cs_out is set to 0 during ASSIGN_CS or TRANSMISSION states, and 1 during IDLE or END.
- Uses internal cnt_clk to delay start_transfer depending on the SPI mode (CPOL/CPHA).
- SPI Mode-specific start delays:
  - Mode 0 / Mode 2 (CPHA = 0): Transfer starts after 1 cycle.
  - Mode 1 (CPHA = 1, CPOL = 0): Transfer starts after 1 cycle.
  - Mode 3 (CPHA = 1, CPOL = 1): Transfer starts after 3 cycles.

Slave Mode (master_slave = 0)

- cs is tristated (1'bz) and monitored externally.
- If cs == 0 and the FSM is in IDLE, slave_enable is set.
- If cs == 1, slave_enable is cleared.
- start_transfer is not driven; the slave reacts to external CS and clock.

Importance of cnt_clk

- cnt_clk introduces a delay in asserting start_transfer to match correct SPI edge timing.
- Ensures data sampling aligns with SPI clock edges according to the selected CPOL/CPHA mode.
- Enables compatibility with all 4 SPI modes using a unified FSM design.

3.3.16 How to Use SPI

The SPI peripheral is located at 0x00010004. To operate in master mode:

- Use set spi to configure mode (CPOL/CPHA), data size, and master/slave.
- Compute the SPI address.
- Write the transmit data.

Example:

```
SPI : ; SPI address is 0x00010004
    org 0x00000000:
    set spi 7,6,5       ; Master mode, 32-bit, SPI Mode 0 (CPHA=0, CPOL=0)
    movu r1,0x0001
    addh r1,r1,0x0004   ; SPI address = 0x00010004
    mov r2,0xF0F0       ; Data to transmit
    st r2,0[r1]         ; Start SPI transfer
```

3.4 Programmable Timer

3.4.1 Overview

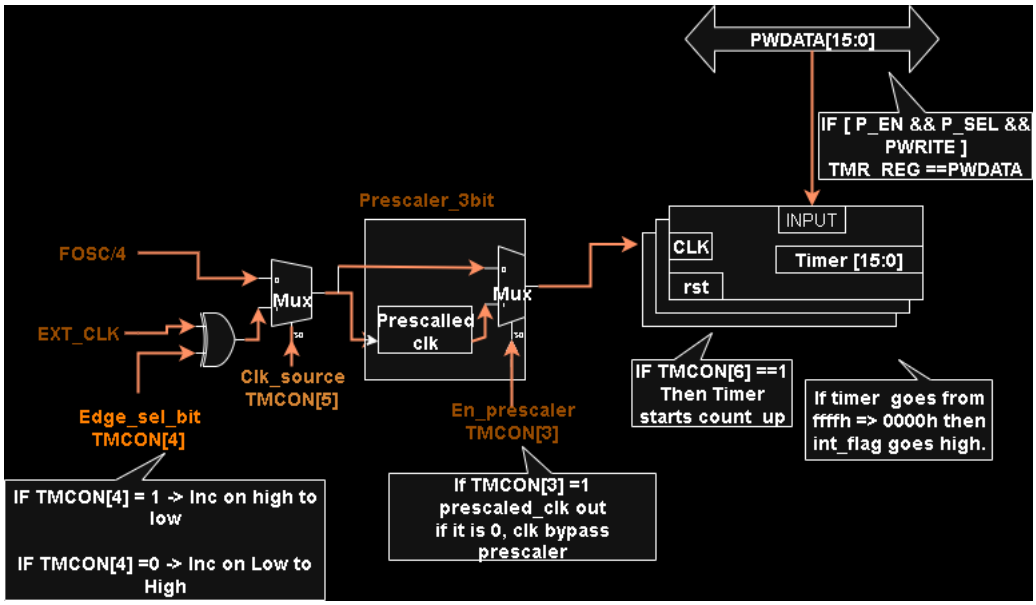


Figure 16: Programmable Timer Architecture

This document describes a parametrizable **16-bit timer** with interrupt detection, along with a **3-bit prescaler** module. The Timer module can operate in **timer mode** (driven by internal clock) or **event-counter mode** (driven by external clock). Control is provided via a **7-bit control register** `TMCON` and a CPU bus interface for parallel loading (`PWDATA`).

Bit-level description:

7	6	5	4	3	2	1	0
Unused	TIMER_ON	CLK_SRC	EDGE	PRE_EN	PRESCALE_VALUE		

TMCON Control Bits The `TMCON` register is a 7-bit control register with the following bit definitions:

- **[6]** `TIMER_ON`: Enables the timer. Set to 1 to allow counting; set to 0 to freeze the timer.
- **[5]** `CLOCK_SOURCE`: Selects the clock source. Set to 0 for internal clock (`int_clk`); set to 1 for external edge-triggered clock (`ext_clk`).
- **[4]** `EDGE_SELECT`: Selects the active edge for counting in external mode. Set to 0 for low-to-high edge; set to 1 for high-to-low edge.
- **[3]** `PRESCALE_ENABLE`: Enables the prescaler module. When set to 1, clock division is enabled.
- **[2:0]** `PRESCALE_VALUE`: Sets the prescale factor. The clock frequency is divided by $2^{(\texttt{prescale\_value}+1)}$.

3.4.2 Functional Behaviour

1. **Reset** (`rst`): Clears the timer and `int_flag`. Sets timer = 0.
2. **Parallel load** occurs when `P_EN`, `PWRITE`, and `P_SEL` are all high: Loads `PWDATA` into the timer and clears the `int_flag`.
3. **Counting** when `TMCON[6]` = 1: On each rising edge of the prescaled clock, the timer increments. If the count overflows from `0xFFFF` to `0x0000`, `int_flag` is set.
4. **Hold** when `TMCON[6]` = 0: The timer value remains constant.

3.4.3 Prescaler Module

- Implements an 8-bit toggle flip-flop chain (`T_FF_freq_div`) to divide the clock by powers of two: $2^{(\texttt{prescale\_value}+1)}$.
- Selects the output stage `Qt[prescale_value]` when `en_prescaler` = 1.
- When `en_prescaler` = 0, the prescaler is bypassed and raw clock is forwarded to `clk_out`.

3.4.4 Functional Flow

1. **Clock Selection**:
   - `TMCON[4]` selects edge polarity for external clock transitions.
   - `TMCON[5]` selects the clock source: internal or external.
2. **Prescaling**:
   - The selected clock drives the prescaler chain.
   - The output clock is either divided (`en_prescaler` = 1) or bypassed (`en_prescaler` = 0).
3. **Timer FSM**:
   - Operates on the positive edge of the clock.
   - Execution priority: `rst` → parallel-load → counting → hold.
   - If the timer overflows from `0xFFFF` to `0x0000`, the `int_flag` is set and the timer stops counting.

Note: To resume operation after an interrupt, `int_flag` must be cleared and the timer must be reloaded with a new initial value as per the required delay.

3.4.5 Timing and Usage

- **Time-Delay Mode**: Set `TMCON[5]` = 0, `TMCON[3]` = 0 (bypass prescaler) or 1 (enable prescaler), and load the timer with `PWDATA`. On each internal `int_clk` rising edge, the timer increments. On overflow, service the `int_flag` and reload the timer.
- **Event-Counter Mode**: Set `TMCON[5]` = 1, `TMCON[3]` = 0 or 1, and provide `ext_clk` pulses. Each edge (per `TMCON[4]`) causes the timer to increment.
- **Prescaled Count**: Enable the prescaler by setting `TMCON[3]` and choose the division factor using `TMCON[2:0]`.

3.4.6 How to Use the Timer

The timer is memory-mapped at address `0x00010002` (with the 17th bit set to 1). To configure the timer:

- Use the `set timer` instruction to configure prescaler and enable bits.
- Use `movu` and `addh` to compute the timer address.
- Store the preload value into the timer register.
- Turn the timer on using the final `set timer` instruction.

Example:

```
TMR : ; Timer address is 0x00010002 with 17th bit = 1
    org 0x00000000:
    set timer 0,2,5     ; Prescaler = 001 (÷4)
    movu r1,0x0001      ; Upper bits of timer address
    addh r1,r1,0x0002   ; Full timer address = 0x00010002
    mov r2,0xFFF6       ; Count value = 9 (0xFFF6 to 0xFFFF rolls over)
    st r2,0[r1]         ; Store count to timer
    set timer 6         ; Turn timer ON
```

3.5 GPIO

Bidirectional General Purpose Input/Output (GPIO) pin with programmable direction and data control. It allows an external system to configure the pin as either an input or an output, write data to it when set as an output, and read its value when set as an input.

The `WR_TRIS` signal is used to enable writing to the TRIS register, which determines the direction of the GPIO pin. When the TRIS register is set to logic 0, the pin is configured as an output. When it is set to logic 1, the pin becomes an input.

The actual value to be written into the TRIS or output data register is provided through a serial input data line called `PWDATA_s`.

The `WR_PORT` signal enables writing to the PORT register, which holds the value that should be driven onto the GPIO pin when it is configured as an output. If the TRIS register indicates an output mode, the value stored in the PORT register is driven onto the pin. If the pin is configured as input, it is placed in a high-impedance (tri-state) state to allow external signals to drive it safely.

For reading from the pin, the `RD_PORT` signal is used. When asserted, it triggers a latch that captures the current value present on the GPIO pin. This latched value is then made available on the output port `PRDATA_tri_buf`, which is a tri-state buffer. The buffer ensures that the data is only driven onto the bus when a read is intended, thus avoiding bus contention.

Internally, the module uses dedicated D flip-flops to store the direction (TRIS) and output value (PORT), as well as a latch to capture the input data from the pin. A buffer is used to tap the value from the bidirectional pin and feed it into the read latch. Finally, a tri-state buffer conditionally drives the output read value onto the `PRDATA_tri_buf` line based on the read enable signal.

Step 1: Pin Configuration (Input/Output) The first operation in the GPIO system is to configure the direction of each pin. This is done by writing to a specific address (**address = 94 or 95**) using the APB interface. When the `PSEL` and `PWRITE` signals are high and the `PADDR` value is `5'b11110`, it indicates a write operation to the direction (TRIS) registers. In this case, each bit of the 16-bit `PWDATA` input determines the mode of the corresponding GPIO pin. If a bit is 0, the corresponding pin is configured as an output; if it is 1, the pin becomes an input. This configuration is applied to all 16 pins simultaneously.

Step 2: Sending Data to Output Pins Once the pins are configured, the processor can send data to those configured as outputs. This is done by writing to the GPIO PORT registers. If `PSEL` and `PWRITE` are high and the `PADDR` value is `5'b11111`, the module interprets this as a write to the output data registers. Each bit of the `PWDATA` signal is then written to the corresponding GPIO pin, but only if that pin is set as an output.

Step 3: Reading Data from Input Pins To read data from the GPIO pins configured as inputs, the processor issues a read request. This occurs when `PSEL` is high, `PWRITE` is low, and `PADDR` is `5'b11111`. The module then reads the actual logic levels from all GPIO pins and returns their states through the 16-bit `PRDATA` output. Each bit in `PRDATA` reflects the real-time logic level of the corresponding pin.

Summary: The CPU first sets the direction for each pin (input or output), then writes data to the output pins or reads data from the input pins.

3.5.1 How to Use GPIO

The TRIS register address is `0x00010001`, and the data (PORT) register address is `0x00010011`.

The following example shows how to declare all 16 GPIO ports as output and store the data `0xF0F0` to the data register.

Example:

```
gpio : GPIO address is 0x00010001 for TRIS register
       and 0x00010011 for data register

    org 0x00000000
    movu r1,0x0001
    addh r1,r1,0x0001          ; r1 = 0x00010001 (TRIS address)

    movu r2,0x0001
    addh r2,r2,0x0011          ; r2 = 0x00010011 (PORT address)

    mov r3,0x00000000          ; Declaring all 16 ports as output (TRIS = 0)
    st r3,0[r1]                ; Write to TRIS register

    mov r3,0xF0F0              ; Data to write to PORT
    st r3,0[r2]                ; Store data to PORT register

    nop
    nop
```