ADVANCED CPU ARCHITECTURE AND HARDWARE ACCELERATORS LAB

FINAL PROJECT

MIPS BASED MCU ARCHITECTURE

361.1.4693

Table of Contents

System Overview **3**

MIPS **5**

Instruction fetch 6

Instruction decode 6

Execute 7

Data memory 7

Write back 7

Basic Timer 8

FIR filter HW-Accelerator 10

Interrupt controller 12

UART 14

GPIO 15

Critical path 16

Results Analysis 17

ModelSIM Wave Analysis 17

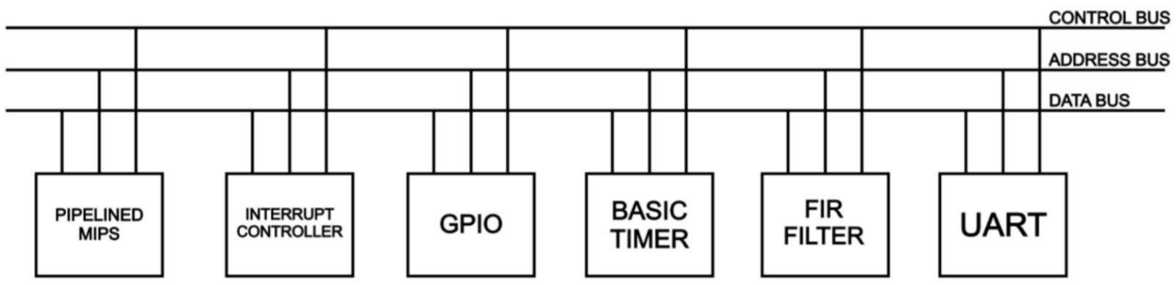
Signal Tab Analysis 18

System Overview

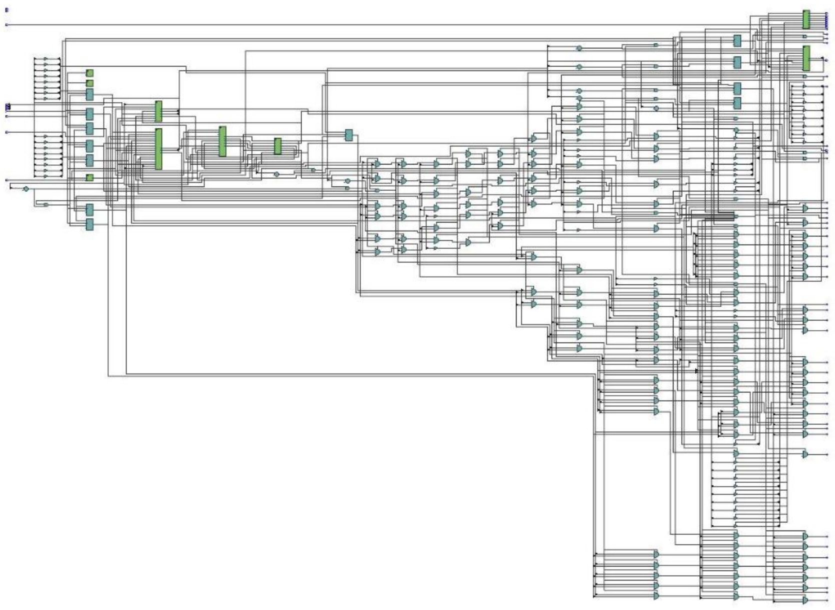
This project focuses on designing an MCU featuring a 5-stage pipelined MIPS processor, comprising the stages WB, MEM, EX, ID, and IF. The processor includes a hazard detection unit to identify and manage potential execution conflicts.

In addition to the CPU, the system integrates several hardware peripherals that interact and operate together as needed. Communication between all components is handled via three shared bus lines, which transmit relevant data between sources and destinations.

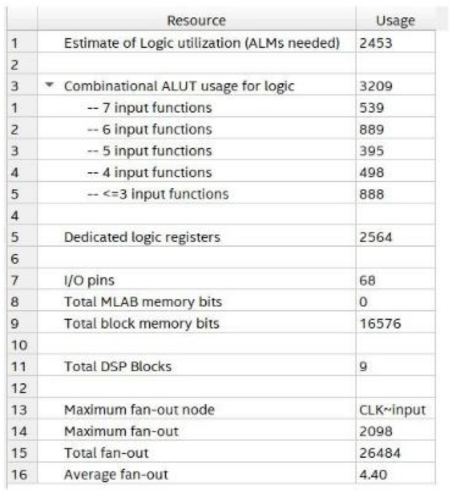
Below is a block diagram representing the overall system architecture –



This section presents an overview of the RTL structure within this module:



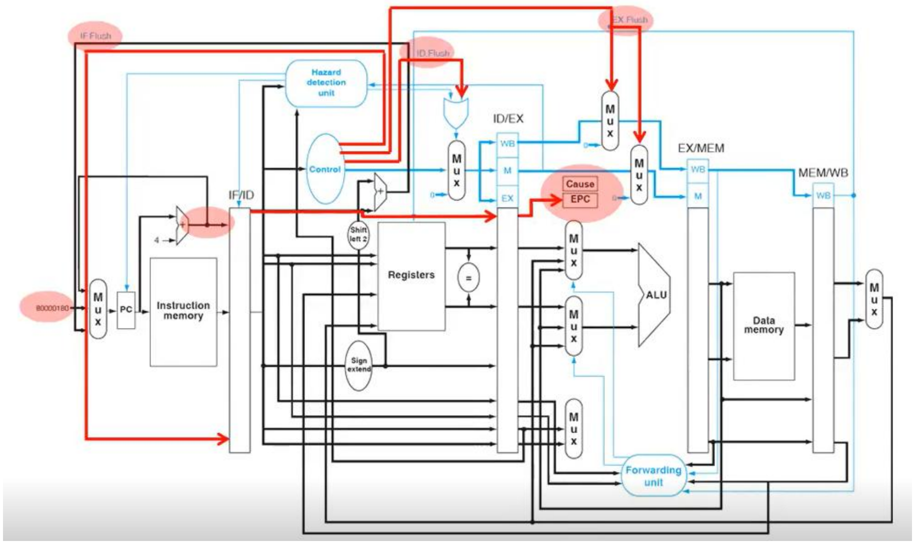
The module employs combinational logic as part of its internal operations:



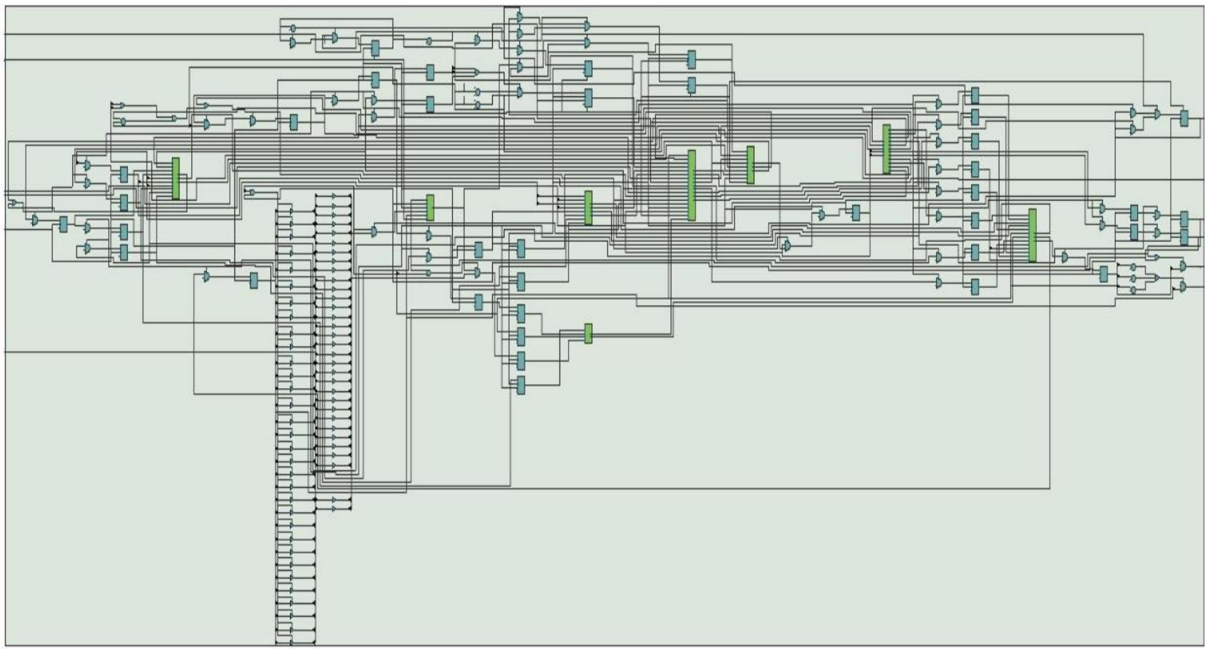
	Resource	Usage
1	Estimate of Logic utilization (ALMs needed)	2453
2		
3	▾ Combinational ALUT usage for logic	3209
1	-- 7 input functions	539
2	-- 6 input functions	889
3	-- 5 input functions	395
4	-- 4 input functions	498
5	-- <=3 input functions	888
4		
5	Dedicated logic registers	2564
6		
7	I/O pins	68
8	Total MLAB memory bits	0
9	Total block memory bits	16576
10		
11	Total DSP Blocks	9
12		
13	Maximum fan-out node	CLK~input
14	Maximum fan-out	2098
15	Total fan-out	26484
16	Average fan-out	4.40

MIPS

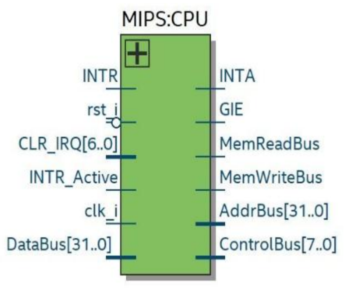
This module represents the MCU's CPU. It is a 5-stage pipelined processor following the Von Neumann architecture, with separate instruction and data memories. The design includes forwarding between stages to optimize clock cycle usage. The CPU supports all instructions required for the exercises, enabling execution of programs that rely exclusively on these operations. A diagram of the processor is shown below:



This section presents an overview of the RTL structure within this module:



Graphical representation of the module:



Instruction Fetch

This stage retrieves the instruction located at the current program counter (PC) from instruction memory. It also handles updating the PC, which can happen in several ways:

- Jumping to a specific target address.
- Branching to a calculated address.
- Moving to the next sequential instruction.

Instruction Decode

In this stage, the fetched instruction is analyzed according to its type. The instruction is broken down into its components using the instruction encoding, as shown below:

Type	-31-			format (bits)		-0-
R	opcode (6)	rs (5)	rt (5)	rd (5)	shamt (5)	funct (6)
I	opcode (6)	rs (5)	rt (5)	immediate (16)		
J	opcode (6)	address (26)				

The required registers are identified and their values are read. Branch and jump decisions are also evaluated at this stage, since the necessary information is now available. Control logic interprets the instruction and generates the necessary signals for later stages.

Execute

In the execution stage, arithmetic, logic, or memory-related calculations are performed. This includes evaluating branch conditions and determining the next PC.

For pipelined processors, branches can create hazards because the next instruction may depend on the branch outcome. Solutions include stalling or predicting the branch result. In this design, branch decisions are moved earlier to reduce wasted cycles and improve efficiency.

Data forwarding is applied when needed to resolve dependencies between instructions.

Data Memory

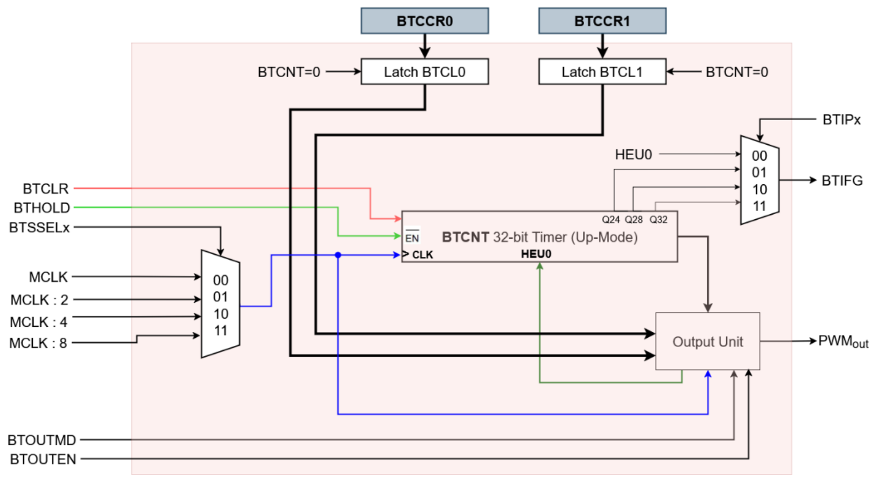
This stage handles reading from and writing to the system's main memory.

Write Back

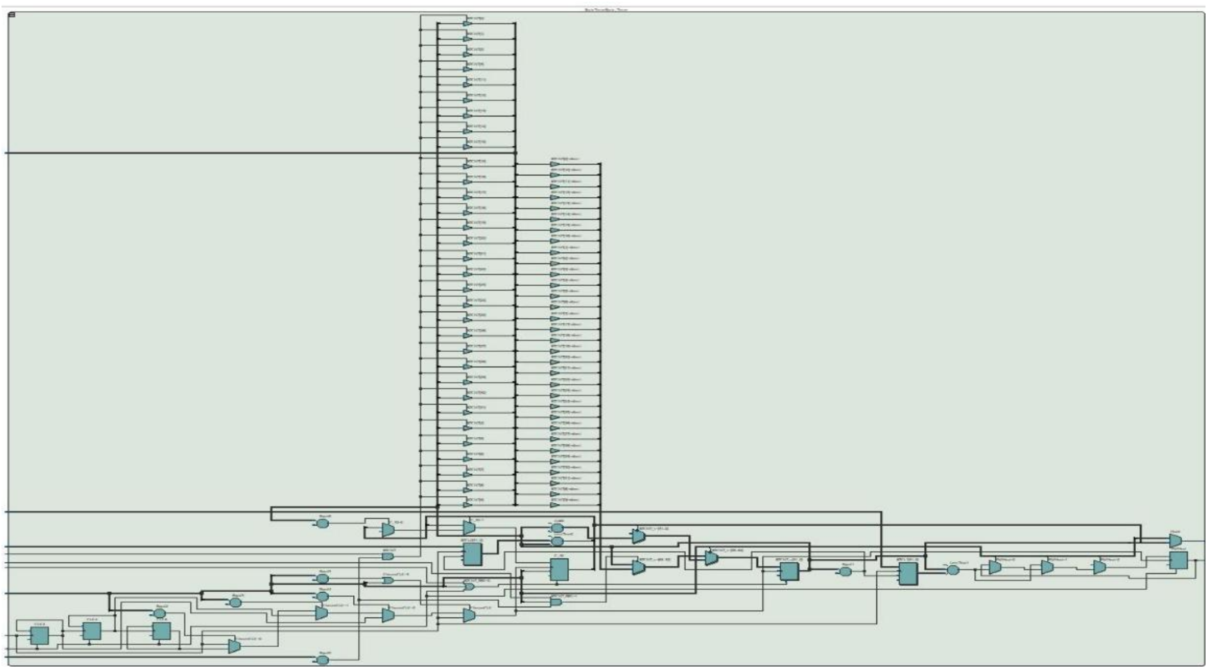
The final stage returns results to the registers. Instructions that produce a result, whether from memory or computations, update the appropriate register to complete execution.

Basic Timer

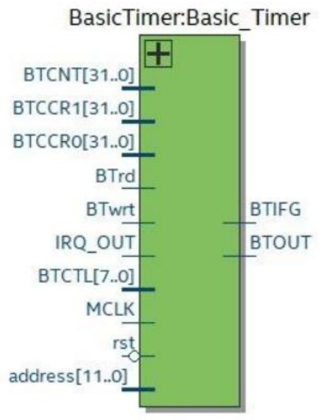
The BasicTimer module is a 32-bit timer peripheral that provides counting, PWM generation, and interrupt signaling. It increments a counter based on a selected internal clock, with configurable compare values to control PWM output. The timer supports clock division, counter reload, and overflow detection. An interrupt flag output indicates events such as compare matches or counter overflow, allowing external systems to respond to timer events.



This section presents an overview of the RTL structure within this module:

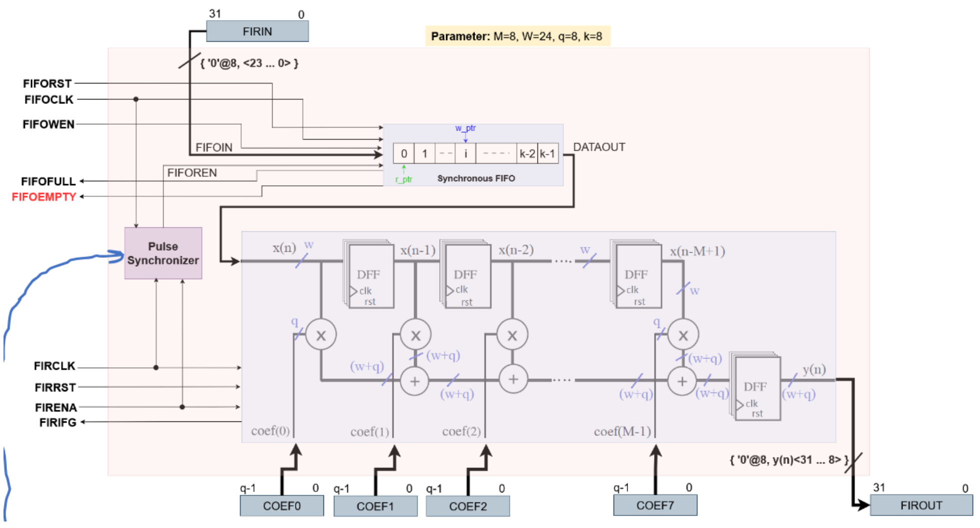


Graphical representation of the module:

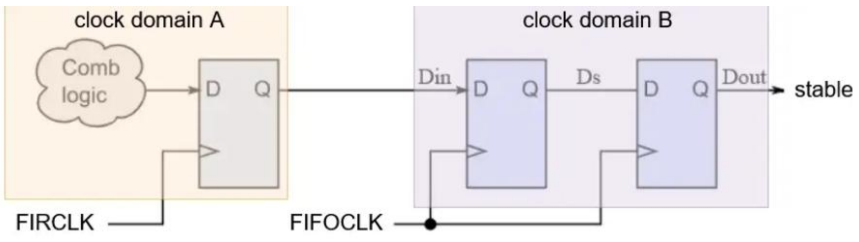


FIR filter HW-Accelerator

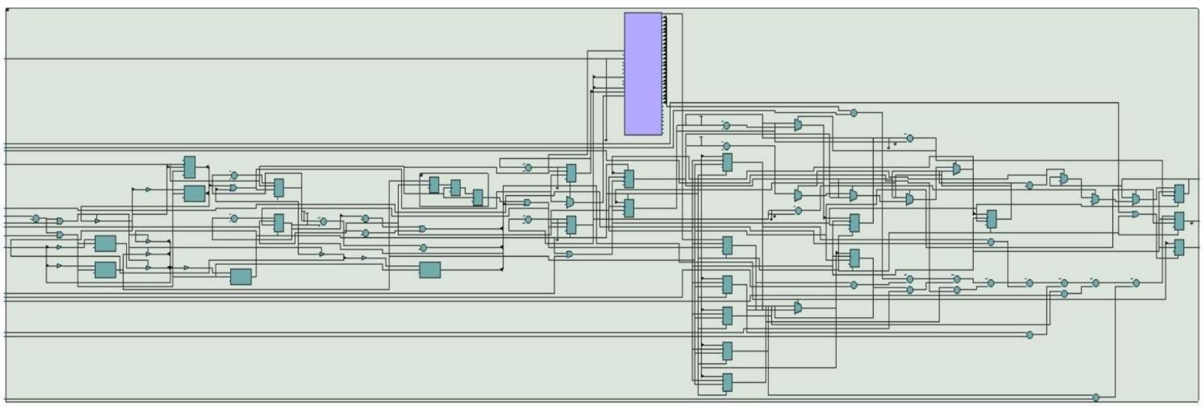
The module implements a FIR filter with an 8-word FIFO and dual-clock operation. FIFOCLK handles high-speed data writes, while FIRCLK processes filtering at a lower rate using eight configurable coefficients. The design multiplies current and previous input samples by the coefficients, sums the results, and outputs the filtered data. Status signals indicate FIFO full/empty conditions, and a pulse synchronizer ensures safe transfer of the enable signal between the two clock domains.



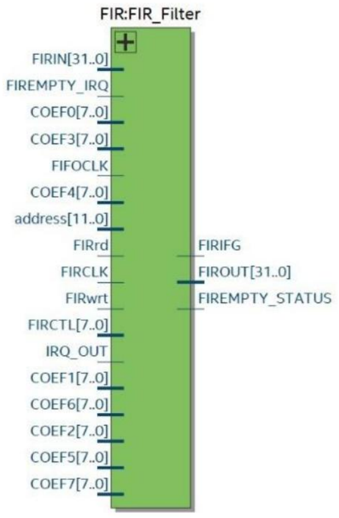
Pulse Synchronizer Diagram:



This section presents an overview of the RTL structure within this module:



Graphical representation of the module:

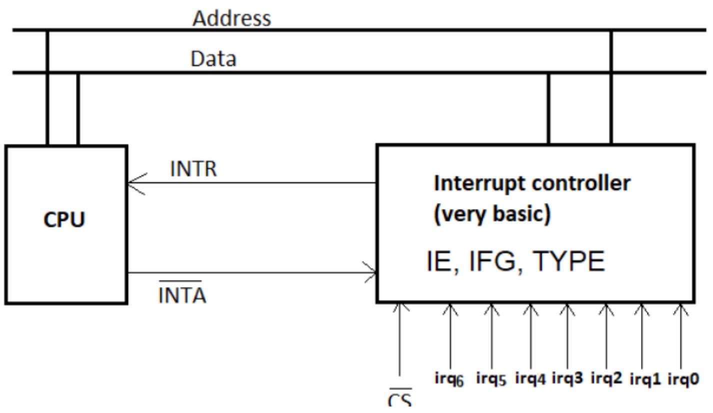


Interrupt Controller

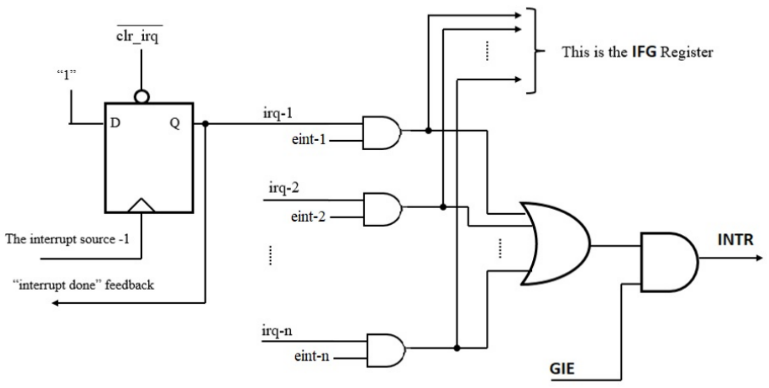
The module implements an **interrupt controller** for a microcontroller. It monitors multiple interrupt sources, manages their status, and signals the CPU when an interrupt occurs. The controller tracks which interrupts are enabled, which are pending, and the type of active interrupt.

Special cases like UART errors and FIFO-empty conditions are handled separately. Each interrupt can be cleared when acknowledged by the CPU, and the module provides outputs for individual interrupt sources, active interrupt status, and the corresponding interrupt vector.

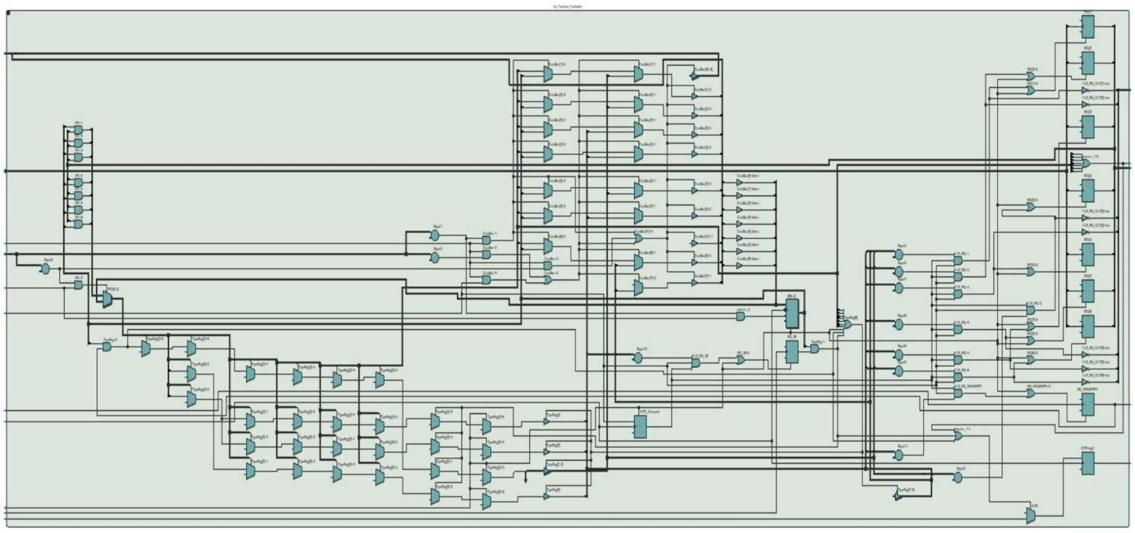
In summary, it centralizes interrupt handling, allowing the CPU to efficiently detect and respond to multiple hardware events.


Address
Data
CPU
INTR
INTA
Interrupt controller (very basic)
IE, IFG, TYPE
CS
irq6 irq5 irq4 irq3 irq2 irq1 irq0

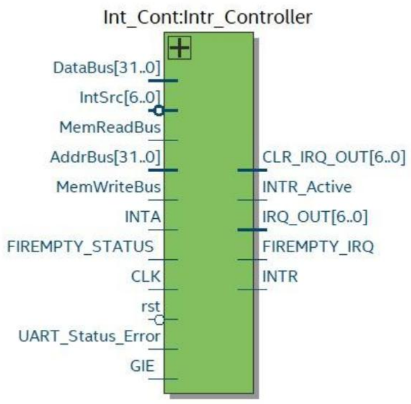
Interrupt Controller Design:



This section presents an overview of the RTL structure within this module:



Graphical representation of the module:



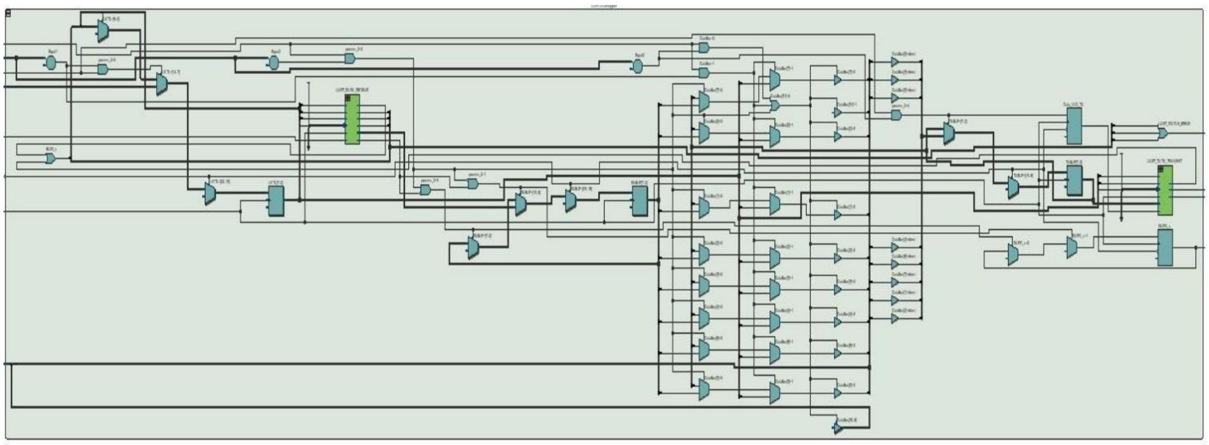
UART

The UART module provides serial communication for a system. It supports data transmission and reception through memory-mapped registers, allows configuration via control settings, and reports status and errors such as framing, parity, and overrun.

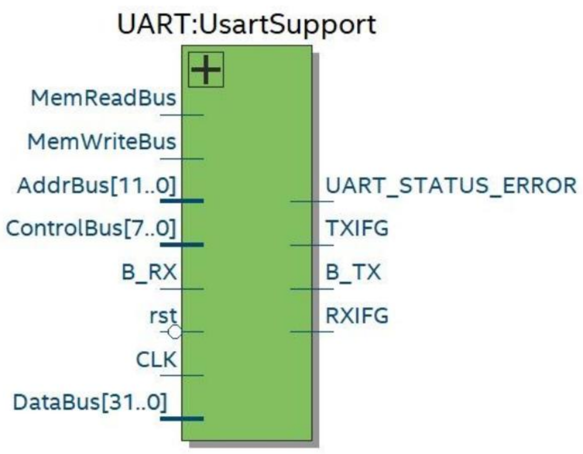
It includes registers for control (UCTL), transmit data (TXBUF), and received data (RXBUF). The module uses a clock divider based on the system clock and baud rate to ensure correct timing for serial communication.

The module integrates separate receiver (UART_RX) and transmitter (UART_TX) components. The receiver captures incoming serial data and signals when valid data is available, while the transmitter sends data from the buffer and indicates when transmission is complete, also reflecting busy status.

This section presents an overview of the RTL structure within this module:

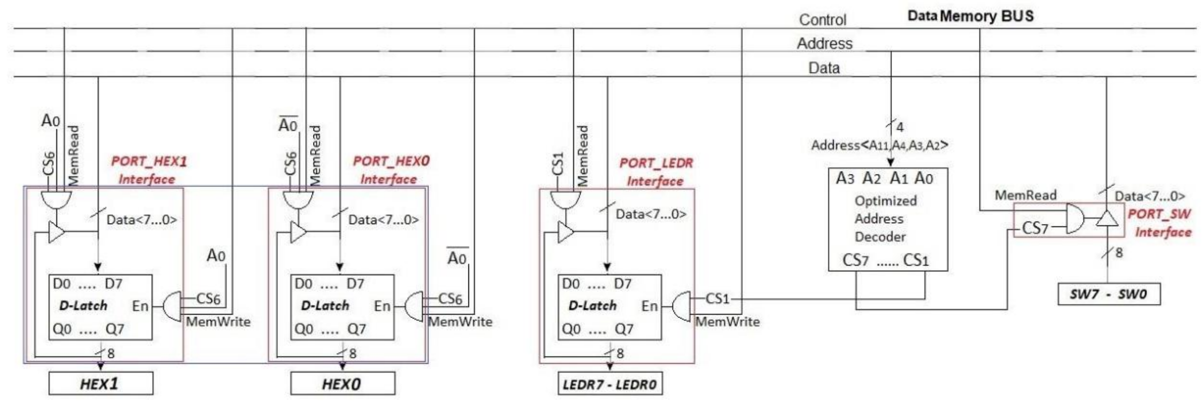


Graphical representation of the module:

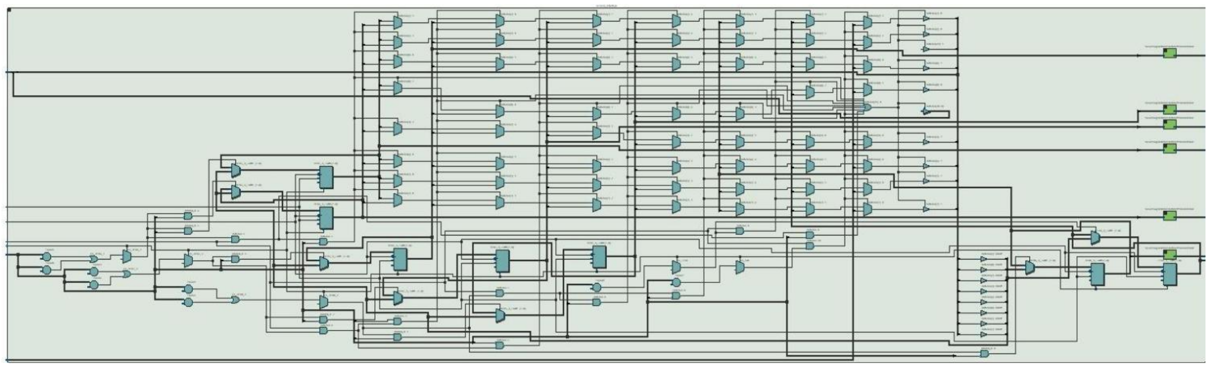


GPIO

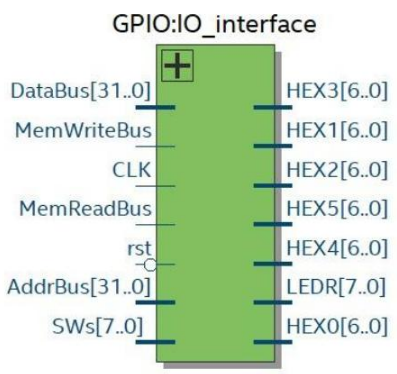
This GPIO module for an FPGA interfaces with switches, LEDs, and six 7-segment displays via a memory-mapped bus. It uses an address decoder to select peripherals and latches to store output values. On writes, the corresponding latch is updated, and on reads, the data from switches or stored outputs is placed on the data bus. The 7-segment outputs are driven through decoder modules, and the LEDs reflect the stored latch values.



This section presents an overview of the RTL structure within this module:

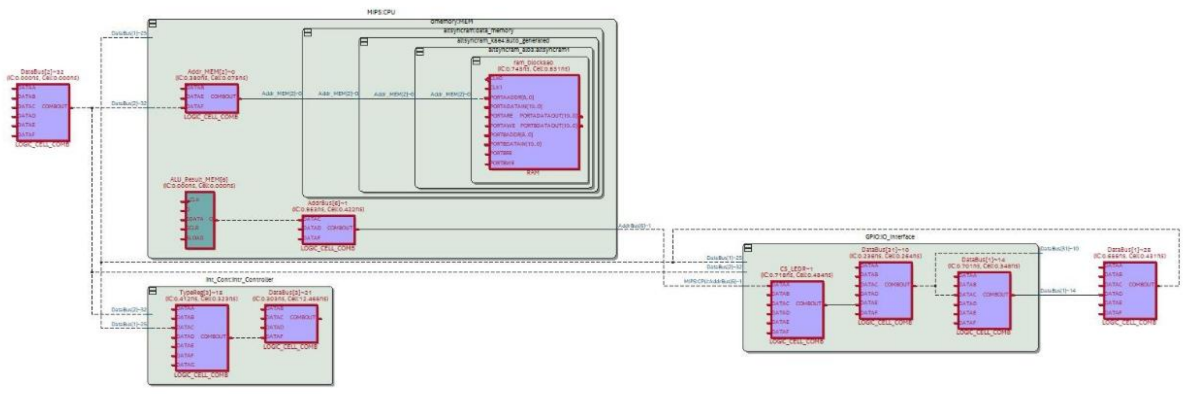


Graphical representation of the module:



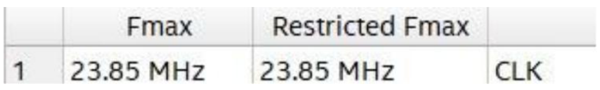
Critical Path

The critical path refers to the longest sequence of logic propagation within the digital circuit of the MCU design developed during this project. It determines the slowest part of the system, which in turn sets the maximum clock frequency that the MCU can safely operate at. The critical path in our system is as follows –



the process starts by placing the calculated address onto the address bus to select the target memory location. Next, the value of the Type register is computed and sent onto the data bus. This value then travels along the critical path to the data memory, enabling the intended read or write operation. By properly sequencing address calculation, Type register evaluation, and data transfer, the MCU ensures correct timing and maintains data integrity throughout memory access.

The maximum frequency of the MCU that is cause by the critical path is:



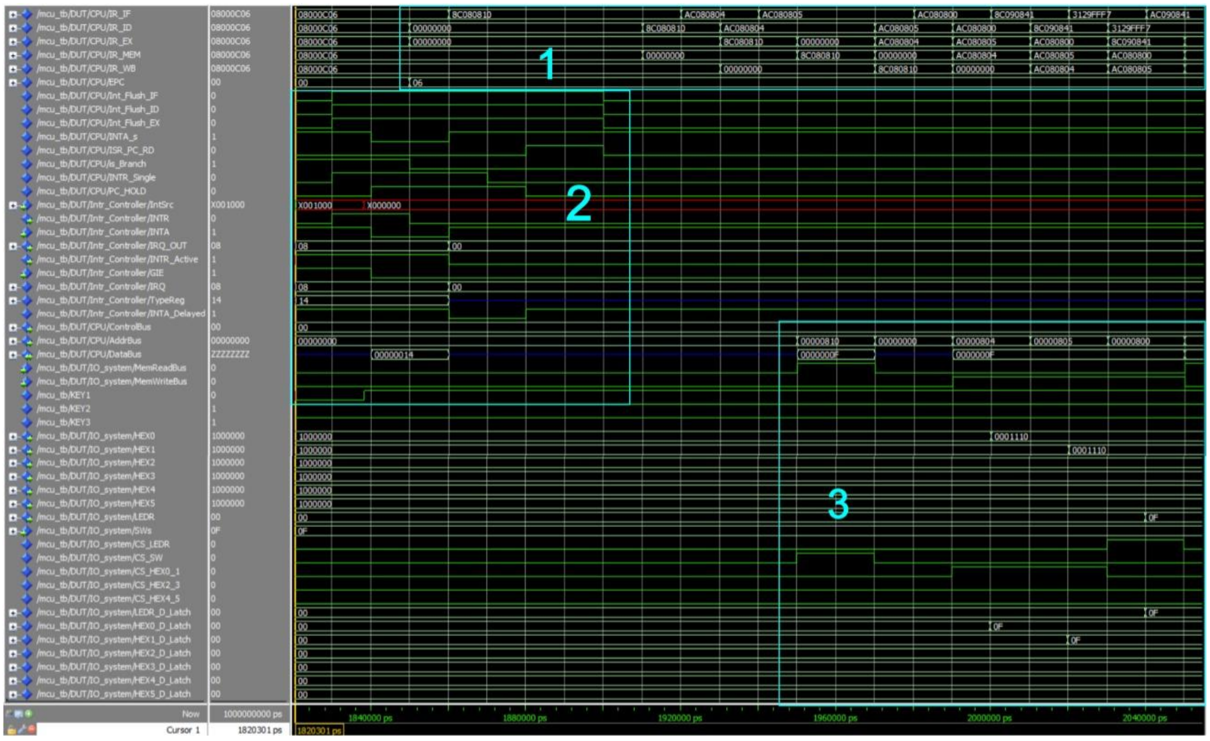
	Fmax	Restricted Fmax	
1	23.85 MHz	23.85 MHz	CLK

Results Analysis

This section evaluates the performance of the implemented system and presents an analysis of the obtained results. The assessment focuses on two main areas:

1. **Timing Evaluation** – Examining critical delays and determining the feasible operating frequencies of the system.
2. **Simulation Testing** – Recording and inspecting signal waveforms using the FPGA's Signal Tap tool to verify correct functionality and ensure compliance with design specifications.

ModelSIM Wave Analysis:

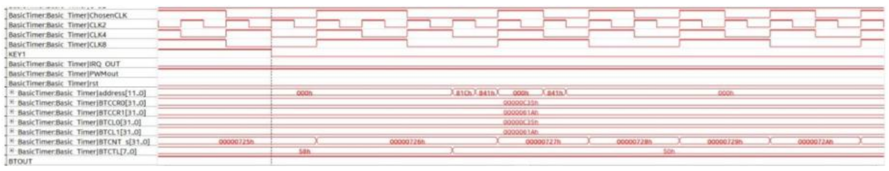


1. **Interrupt Handling and Control Flow:** In Circle 1, the controller interrupts its execution in the main infinite loop upon detection of a Button 1 (KEY1) press, immediately transferring control to the corresponding interrupt service routine (ISR). This demonstrates proper prioritization of asynchronous events. After the ISR completes, the system resumes execution in the main loop without loss of state, indicating effective context preservation and seamless interrupt management.

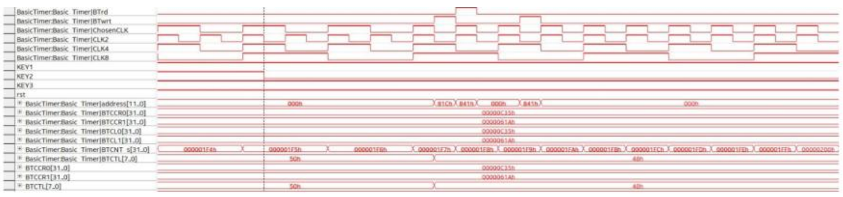
2. **CPU Interrupt Signaling:** In Circle 2, the CPU receives the interrupt request (INTR) triggered by the button press. The subsequent acknowledgment via the INTA signal confirms that the CPU correctly recognizes and processes the interrupt. This sequence validates the interrupt handshake mechanism, ensuring that signal timing and CPU response align with expected hardware behavior.

3. **GPIO Data Capture and Output Verification:** In Circle 3, the GPIO module accurately captures input data and presents it on the LEDs. Observed register states correspond precisely to the expected values, confirming correct latching and data propagation. This also indicates reliable coordination between the interrupt-driven ISR and the GPIO output, demonstrating that the system maintains data integrity even during asynchronous events.

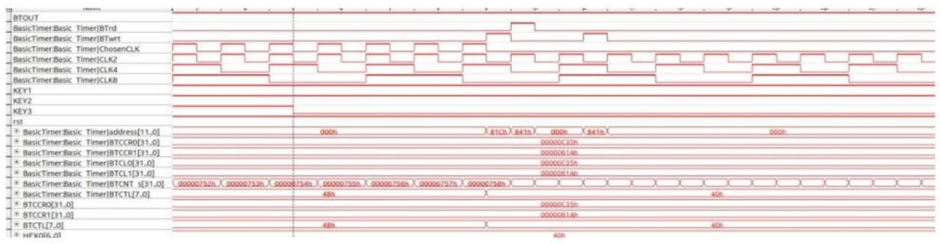
__Signal Tab Analysis:__


BasicTimer:Basic Timer|ChosenCLK
BasicTimer:Basic Timer|CLK2
BasicTimer:Basic Timer|CLK4
BasicTimer:Basic Timer|CLK8
KEY1
BasicTimer:Basic Timer|IRQ_OUT
BasicTimer:Basic Timer|PWMout
BasicTimer:Basic Timer|rst
BasicTimer:Basic Timer|address[11..0]
BasicTimer:Basic Timer|BTCCR0[31..0]
BasicTimer:Basic Timer|BTCCR1[31..0]
BasicTimer:Basic Timer|BTCL0[31..0]
BasicTimer:Basic Timer|BTCL1[31..0]
BasicTimer:Basic Timer|BTCNT_s[31..0]
BasicTimer:Basic Timer|BTCTL[7..0]
BTOUT
00000725h

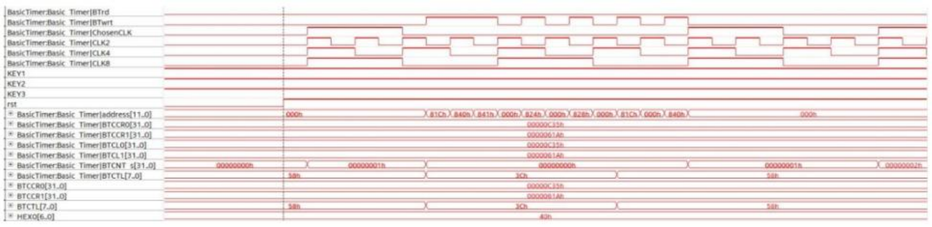
When KEY 1 is pressed, the system selects the CLK/4 clock as the timer's source. As a result, the timer begins counting based on this divided clock frequency. This demonstrates how the first key can slow down the timer relative to the main clock, allowing for longer timing intervals or more controlled operations.


BasicTimer:Basic Timer|BTrd
BasicTimer:Basic Timer|BTwrt
BasicTimer:Basic Timer|ChosenCLK
BasicTimer:Basic Timer|CLK2
BasicTimer:Basic Timer|CLK4
BasicTimer:Basic Timer|CLK8
KEY1
KEY2
KEY3
rst
BasicTimer:Basic Timer|address[11..0]
BasicTimer:Basic Timer|BTCCR0[31..0]
BasicTimer:Basic Timer|BTCCR1[31..0]
BasicTimer:Basic Timer|BTCL0[31..0]
BasicTimer:Basic Timer|BTCL1[31..0]
BasicTimer:Basic Timer|BTCNT_s[31..0]
BasicTimer:Basic Timer|BTCTL[7..0]
BTCCR0[31..0]
BTCCR1[31..0]
BTCTL[7..0]
50h

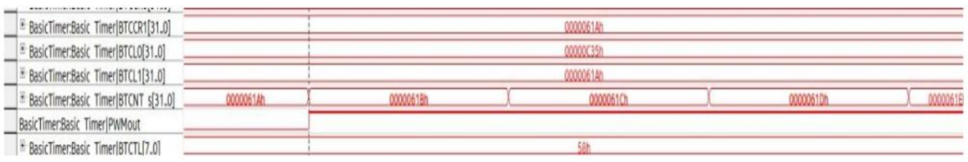
Pressing KEY 2 switches the timer to use the CLK/2 clock. The timer then runs at a faster rate than CLK/4 but still slower than the main clock. This illustrates the intermediate timing option provided by the second key, showing that the timer's behavior can be adjusted dynamically based on the selected clock division.


BTOUT
BasicTimer:Basic Timer|BTrd
BasicTimer:Basic Timer|BTwrt
BasicTimer:Basic Timer|ChosenCLK
BasicTimer:Basic Timer|CLK2
BasicTimer:Basic Timer|CLK4
BasicTimer:Basic Timer|CLK8
KEY1
KEY2
KEY3
rst
BasicTimer:Basic Timer|address[11..0]
BasicTimer:Basic Timer|BTCCR0[31..0]
BasicTimer:Basic Timer|BTCCR1[31..0]
BasicTimer:Basic Timer|BTCL0[31..0]
BasicTimer:Basic Timer|BTCL1[31..0]
BasicTimer:Basic Timer|BTCNT_s[31..0]
BasicTimer:Basic Timer|BTCTL[7..0]
BTCCR0[31..0]
BTCCR1[31..0]
BTCTL[7..0]
HEX0[6..0]

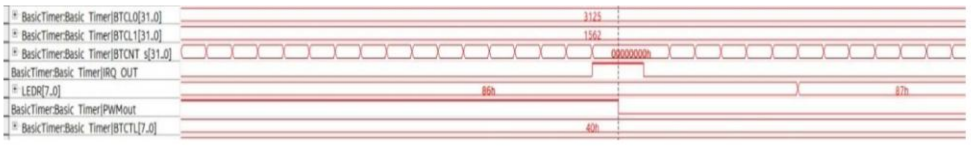
When KEY 3 is pressed, the timer uses the main clock without any division. Consequently, the timer operates at full speed. This confirms that all three keys correctly select the expected clock sources, allowing the system to operate at different rates depending on the chosen input.


BasicTimer:Basic Timer|BTrd
BasicTimer:Basic Timer|BTwrt
BasicTimer:Basic Timer|ChosenCLK
BasicTimer:Basic Timer|CLK2
BasicTimer:Basic Timer|CLK4
BasicTimer:Basic Timer|CLK8
KEY1
KEY2
KEY3
rst
BasicTimer:Basic Timer|address[11..0]
BasicTimer:Basic Timer|BTCCR0[31..0]
BasicTimer:Basic Timer|BTCCR1[31..0]
BasicTimer:Basic Timer|BTCL0[31..0]
BasicTimer:Basic Timer|BTCL1[31..0]
BasicTimer:Basic Timer|BTCNT_s[31..0]
BasicTimer:Basic Timer|BTCTL[7..0]
BTCCR0[31..0]
BTCCR1[31..0]
BTCTL[7..0]
HEX0[6..0]

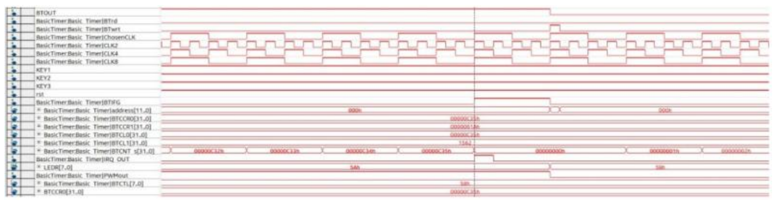
Pressing rst resets the clock selection back to CLK/8, returning the timer to a default slower rate. This demonstrates the reset functionality of the system, ensuring it can always return to a known starting configuration after user interaction or an event.


BasicTimer:Basic Timer|BTCCR1[31..0]
0000061Ah
BasicTimer:Basic Timer|BTCL0[31..0]
00000C35h
BasicTimer:Basic Timer|BTCL1[31..0]
0000061Ah
BasicTimer:Basic Timer|BTCNT_s[31..0]
0000061Ah
0000061Bh
0000061Ch
0000061Dh
BasicTimer:Basic Timer|PWMout
BasicTimer:Basic Timer|BTCTL[7..0]
58h

The PWM signal increases when the counter reaches the value of BTCL1. This shows how the timer output can be used to generate precise pulse-width modulation, linking the timer's state to real output signals for control or signaling purposes.


BasicTimer:Basic Timer|BTCL0[31..0]
3125
BasicTimer:Basic Timer|BTCL1[31..0]
1562
BasicTimer:Basic Timer|BTCNT_s[31..0]
00000000h
BasicTimer:Basic Timer|IRQ_OUT
LEDR[7..0]
86h
87h
BasicTimer:Basic Timer|PWMout
BasicTimer:Basic Timer|BTCTL[7..0]
40h

After an interrupt occurs, the value of the LED port increments by 1. This shows the system's response to asynchronous events, where the interrupt triggers a predictable change in output, illustrating the integration of timing and event-driven control.


BTOUT
BasicTimer:Basic_Timer|BTrd
BasicTimer:Basic_Timer|BTwrt
BasicTimer:Basic_Timer|ChosenCLK
BasicTimer:Basic_Timer|CLK2
BasicTimer:Basic_Timer|CLK4
BasicTimer:Basic_Timer|CLK8
KEY1
KEY2
KEY3
rst
BasicTimer:Basic_Timer|BTIFG
BasicTimer:Basic_Timer|address[11..0]
BasicTimer:Basic_Timer|BTCCR0[31..0]
BasicTimer:Basic_Timer|BTCCR1[31..0]
BasicTimer:Basic_Timer|BTCL0[31..0]
BasicTimer:Basic_Timer|BTCL1[31..0]
BasicTimer:Basic_Timer|BTCNT_s[31..0]
BasicTimer:Basic_Timer|IRQ_OUT
LEDR[7..0]
BasicTimer:Basic_Timer|PWMout
BasicTimer:Basic_Timer|BTCTL[7..0]
BTCCR0[31..0]
5Ah

The system state after the interrupt is clearly observable, confirming that the LED port value has been updated correctly. This provides a visual verification that the interrupt mechanism and subsequent updates function as designed.