Ben-Gurion University of the Negev

Contents

Ben-Gurion University of the Negev

Design and Implementation of a MIPS ISA in a Single-Cycle Architecture with Interrupt Handling

Yarden Levy and Hadar Doron

03.09.2024

Abstract

The objective of this project is to design and implement an Instruction Set Architecture (ISA) for the MIPS processor within a single-cycle architecture. The project involves the creation of a MIPS processor that operates in a single-cycle mode, which means that each instruction is completed in one clock cycle. Additionally, the MIPS unit will be integrated with an I/O interface unit responsible for handling external device interrupts. The system must be capable of managing these interrupts efficiently and providing appropriate responses within the single-cycle architecture constraints.

Introduction

In modern computer systems, the design of a processor's architecture is a critical factor that influences its performance, efficiency, and capability to handle various tasks, including input/output operations and interrupt handling. The MIPS (Microprocessor without Interlocked Pipeline Stages) architecture, known for its simplicity and effectiveness, serves as an ideal model for exploring these design principles.

This project aims to develop a single-cycle MIPS processor, focusing on the following key objectives:

- **ISA Design:** Define and implement the MIPS ISA, ensuring it supports essential instructions required for typical processing tasks.
- **Single-Cycle Operation:** Develop the processor such that each instruction is completed in one clock cycle, optimizing the design for simplicity and speed.
- **I/O Interface Integration:** Design an interface that allows the processor to interact with external devices, enabling the system to receive and handle interrupts effectively.
- **Interrupt Handling:** Implement a robust interrupt handling mechanism to ensure that the processor can respond to external events without compromising performance.

This document will detail the design process, challenges encountered, and solutions implemented during the project, providing a comprehensive overview of the work completed and its outcomes.

Overview of the top-level structural Module

1. **MIPS Processor Overview**:

   - This top-level module connects various submodules that together implement a MIPS processor. These submodules include instruction fetch (**Ifetch**), instruction decode (**Idecode**), control (**control**), execution (**Execute**), and data memory (**dmemory**).

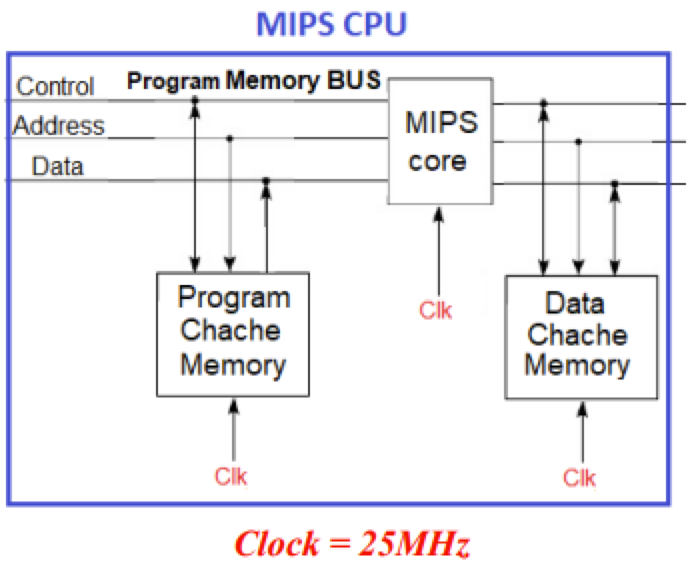


Figure 1 – mips' top

2. **Signal Declaration**:

   - Various signals are declared to connect the components within the MIPS processor, including those for the program counter (**PC_plus_4**), register file data (**read_data_1**, **read_data_2**), ALU results (**ALU_result**), and control signals (**RegDst**, **Regwrite**, **MemWrite**, etc.).

3. **DataBus and AddressBus Handling**:

   - The **DataBus** and **AddressBus** are used for communication with external memory or peripherals.

   - The **DataBus** is tri-stated when not in use, ensuring that it only drives data when necessary.

- The **AddressBus** is driven by the lower bits of the ALU result.

4. **Control Signals**:

   - The **ControlBus** outputs control signals (**MemRead**, **MemWrite**) based on the internal operations of the processor.

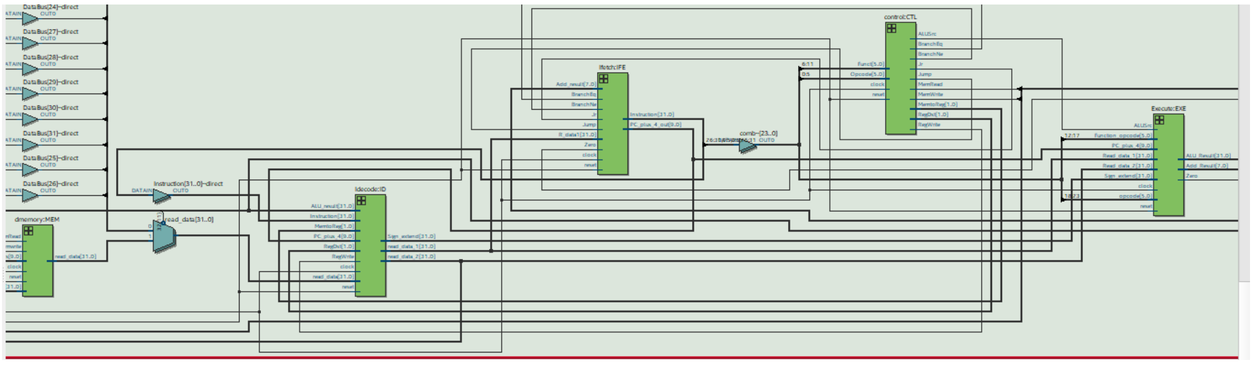


Figure 2 - rtl of mips

5. **Component Connections**:

   - The submodules (**Ifetch**, **Idecode**, **control**, **Execute**, **dmemory**) are connected using the declared signals. Each submodule is responsible for a specific stage in the MIPS:

     - **Ifetch**: Handles instruction fetching and the program counter.
     - **Idecode**: Decodes the instruction and reads data from the register file.
     - **control**: Generates control signals based on the instruction opcode and function.
     - **Execute**: Performs arithmetic and logic operations using the ALU.
     - **dmemory**: Handles data memory operations, reading or writing data based on the control signals.

6. **Memory Access**:

   - Memory read and write operations are managed by the **MemReadInt** and **MemWriteInt** signals, which are derived from the main control signals (**MemRead**, **MemWrite**) but are gated by the ALU result to differentiate between normal memory access and peripheral access.

7. **Optional Signal Outputs for Simulation**:

   - Some signals, like the ALU result and the program counter, are commented out and marked for potential output during simulation for debugging or observation purposes.

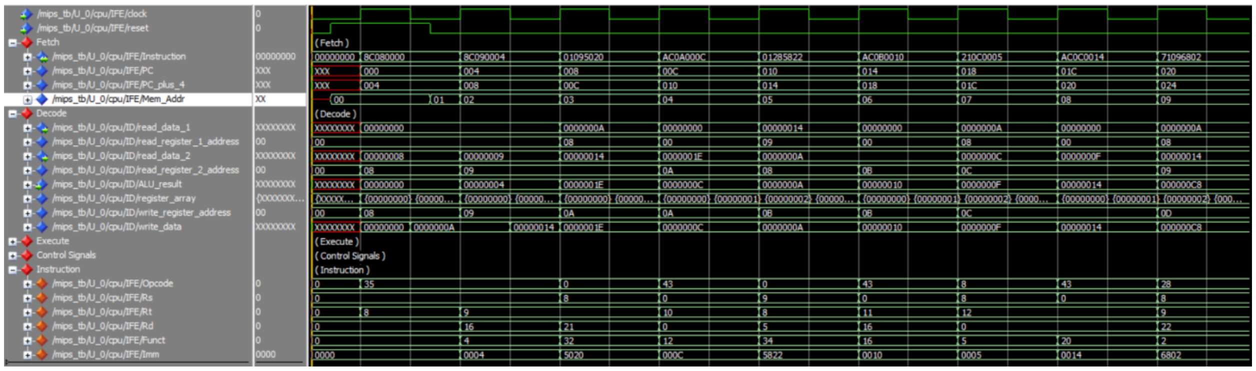

/mips_tb/U_0/cpu/IFE/clock
/mips_tb/U_0/cpu/IFE/reset
Fetch
/mips_tb/U_0/cpu/IFE/Instruction
/mips_tb/U_0/cpu/IFE/PC
/mips_tb/U_0/cpu/IFE/PC_plus_4
/mips_tb/U_0/cpu/IFE/Mem_Addr
Decode
/mips_tb/U_0/cpu/ID/read_data_1
/mips_tb/U_0/cpu/ID/read_register_1_address
/mips_tb/U_0/cpu/ID/read_data_2
/mips_tb/U_0/cpu/ID/read_register_2_address
/mips_tb/U_0/cpu/ID/ALU_result
/mips_tb/U_0/cpu/ID/register_array
/mips_tb/U_0/cpu/ID/write_register_address
/mips_tb/U_0/cpu/ID/write_data
Execute
Control Signals
Instruction
/mips_tb/U_0/cpu/IFE/Opcode
/mips_tb/U_0/cpu/IFE/Rs
/mips_tb/U_0/cpu/IFE/Rt
/mips_tb/U_0/cpu/IFE/Rd
/mips_tb/U_0/cpu/IFE/Funct
/mips_tb/U_0/cpu/IFE/Imm

Figure 3 - wave diagram. An assembly code (see reference) was written on the simulation.

Table 1 - Port Description for MIPS Entity

Port Name	**Direction**	**Size**	**Functionality**
reset	in	1 bit	Reset signal
clock	in	1 bit	Clock signal
ControlBus	out	2 bits (ControlBusSize)	Control signals for memory operations
DataBus	inout	32 bits (DataBusSize)	Data bus
AddressBus	out	32 bits (AddrBusSize)	Address bus

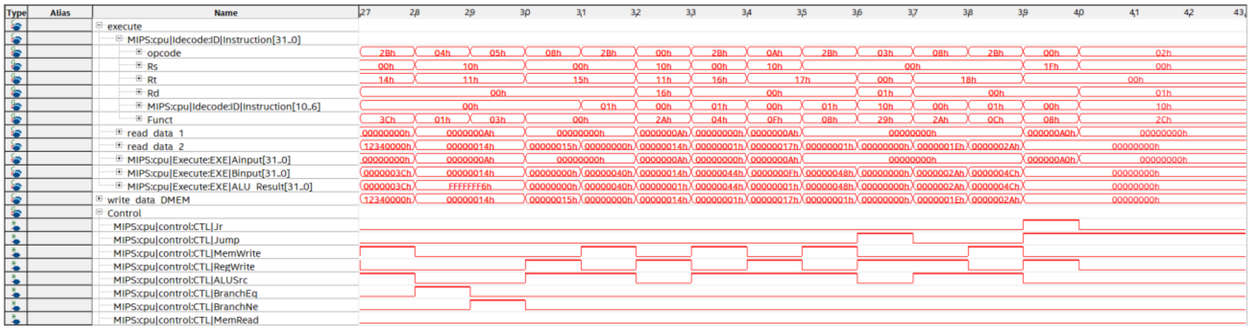

Type
Alias
Name
execute
MIPS:cpu|Idecode:ID|Instruction[31..0]
opcode
Rs
Rt
Rd
MIPS:cpu|Idecode:ID|Instruction[10..6]
Funct
read_data_1
read_data_2
MIPS:cpu|Execute:EXE|Ainput[31..0]
MIPS:cpu|Execute:EXE|Binput[31..0]
MIPS:cpu|Execute:EXE|ALU_Result[31..0]
write_data_DMEM
Control
MIPS:cpu|control:CTL|Jr
MIPS:cpu|control:CTL|Jump
MIPS:cpu|control:CTL|MemWrite
MIPS:cpu|control:CTL|RegWrite
MIPS:cpu|control:CTL|ALUSrc
MIPS:cpu|control:CTL|BranchEq
MIPS:cpu|control:CTL|BranchNe
MIPS:cpu|control:CTL|MemRead

Figure 4 - STP of jump, jal, jr, and branches

Short Description of Jumps, JAL, JR, and Branches in the Simulation:

The waveform shows the control signals and the execution path for various instructions in a MIPS processor simulation. Here's a brief description of the jump, JAL, JR, and branch operations visible in the waveform:

8. ***Jump (Jump)****:*
   - *The* ***Jump*** *signal is activated at specific cycles where the instruction is a jump (J) type. The* ***Jump*** *signal being high indicates that the processor is performing an unconditional jump to a specified address.*
9. ***Jump and Link (JAL)****:*
   - *The* ***Jump*** *signal combined with the* ***RegWrite*** *signal being high in specific cycles indicates the execution of a Jump and Link (JAL) instruction. This instruction performs a jump to a target address while saving the return address in a register.*
10. ***Jump Register (JR)****:*
    - *The* ***Jr*** *signal being high indicates a JR (Jump Register) instruction, where the processor jumps to the address contained in a register (typically used for returning from subroutines).*
11. ***Branches (BranchEq, BranchNe)****:*
    - *The BranchEq signal is used for a branch if equal (BEQ) instruction, where the processor branches to a target address if the two compared registers are equal.*
    - *The BranchNe signal is used for a branch if not equal (BNE) instruction, where the processor branches if the compared registers are not equal.*

*In the waveform, these control signals indicate the execution of specific instructions as the processor evaluates conditions and directs the program counter (**PC**) to the appropriate address based on the instruction type and conditions.*

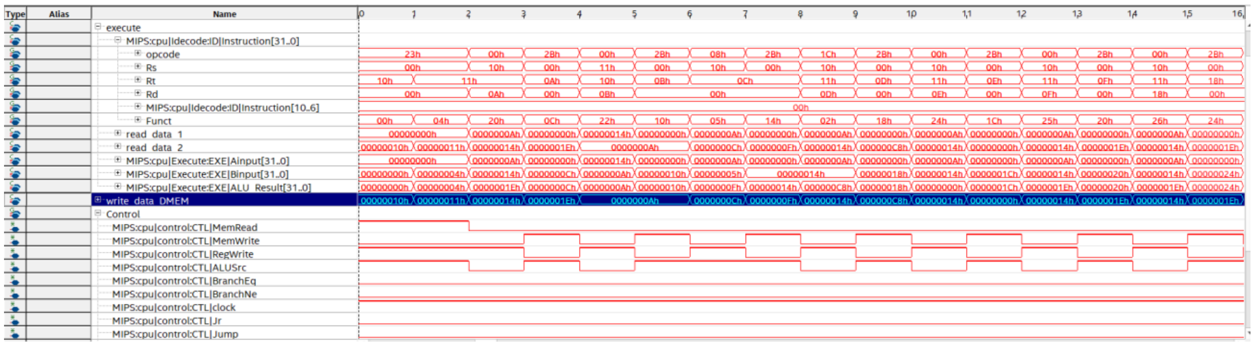

Type
Alias
Name
execute
MIPS:cpu|Idecode:ID|Instruction[31..0]
opcode
Rs
Rt
Rd
MIPS:cpu|Idecode:ID|Instruction[10..6]
Funct
read_data_1
read_data_2
MIPS:cpu|Execute:EXE|Ainput[31..0]
MIPS:cpu|Execute:EXE|Binput[31..0]
MIPS:cpu|Execute:EXE|ALU_Result[31..0]
write_data_DMEM
Control
MIPS:cpu|control:CTL|MemRead
MIPS:cpu|control:CTL|MemWrite
MIPS:cpu|control:CTL|RegWrite
MIPS:cpu|control:CTL|ALUSrc
MIPS:cpu|control:CTL|BranchEq
MIPS:cpu|control:CTL|BranchNe
MIPS:cpu|control:CTL|clock
MIPS:cpu|control:CTL|Jr
MIPS:cpu|control:CTL|Jump

Figure 5 - STP only ALU + load store

Short Description of ALU Operations and Load/Store Instructions in the Simulation:

The waveform illustrates the control signals and data paths for ALU operations and load/store instructions within the MIPS processor. Below is a brief description:

12. ***ALU Operations**:*
    - *The **ALUSrc** signal controls whether the second operand to the ALU is a register value or an immediate value (sign-extended).*
    - *The ALU_Result shows the outcome of the ALU operation, which depends on the opcode and function code of the instruction.*
    - *During cycles where arithmetic or logic operations are executed, the ALU processes the input values (Ainput and Binput) and outputs the result.*
13. ***Load/Store Instructions**:*
    - *The **MemRead** and **MemWrite** signals indicate whether the instruction is a load or store operation, respectively.*
    - *The **DataBus** interaction is visible when the processor writes data to memory (during store operations) or reads data from memory (during load operations).*
    - *The AddressBus provides the memory address for the load/store operation, derived from the ALU_Result.*

This waveform effectively demonstrates the control flow and data handling for both ALU-related instructions and memory access instructions within the MIPS processor.

Overview of the Execute Module

The VHDL code defines the behavior of the **Execute** module, which is a crucial component of a MIPS processor in a single-cycle architecture. The primary function of this module is to implement the Arithmetic Logic Unit (ALU) and handle branch address calculations. Below is an overview of its logic and the operations it supports:

General Logic

- **ALU Inputs:**
    - The module receives two primary data inputs (`Read_data_1` and `Read_data_2`), a sign-extended immediate value (`Sign_extend`), and control signals such as `Function_opcode` and `ALUSrc`.
    - Based on the `ALU_ctl` signal, it selects the appropriate inputs for the ALU operations. For example, it chooses between using the immediate value or the second data input depending on the `ALUSrc` control signal.
- **ALU Control (`ALU_ctl`):**
    - The control logic determines which operation the ALU should perform. This decision is based on the `opcode` and `Function_opcode` inputs. The operations

include basic arithmetic (ADD, SUB), logical operations (AND, OR, XOR), and shifts (SLL, SRL).

  - The ALU supports R-type instructions (like ADD, SUB, AND, OR, etc.), I-type instructions (like ADDI, ANDI, ORI), and branch instructions (like BEQ, BNE). It also handles special operations such as multiplication (MULT) and loading upper immediate (LUI).

- **ALU Operation:**
  - The ALU performs the selected operation on the inputs and produces the result (`ALU_Result`).
  - For operations such as multiplication, only the lower 32 bits of the product are output.
- **Zero Flag (`Zero`):**
  - The module sets the `Zero` output high if the ALU result is zero, which is particularly useful for branch decisions such as BEQ (branch if equal).
- **Branch Address Calculation:**
  - The module computes the branch target address by adding the sign-extended immediate value to the incremented program counter (`PC_plus_4`). The result is output as `Add_Result`.
- **Output Assignment:**
  - The ALU output is selected based on whether the operation is SLT/SLTI, where only the most significant bit is relevant. Otherwise, the full ALU result is output.

Supported Operations

- **Arithmetic:** ADD, SUB
- **Logical:** AND, OR, XOR
- **Shift Operations:** SLL (Shift Left Logical), SRL (Shift Right Logical)
- **Multiplication:** MULT (lower 32 bits of the result are used)
- **Immediate Operations:** ADDI, ANDI, ORI, LUI
- **Branching:** BEQ (branch if equal), BNE (branch if not equal), JR (jump register)
- **Set Less Than:** SLT (Set Less Than), SLTI (Set Less Than Immediate)

Summary

The **Execute** module is responsible for performing ALU operations and calculating branch addresses in a MIPS processor. It supports a wide range of MIPS instructions and generates the necessary control signals to perform the required operations. The outputs include the ALU result, a zero flag for branch conditions, and the branch address.

Overview of the Control Module

The VHDL code provided implements a **control unit** for a MIPS processor. The control unit is responsible for generating the control signals required to execute different instructions based on the `Opcode` and `Funct` fields of the instruction.

General Overview

- **Purpose:** The control unit decodes the `Opcode` and `Funct` fields from the instruction and generates the appropriate control signals that guide the operation of various components in the MIPS processor, such as the ALU, registers, memory, and branching logic.
- **Inputs:**
  - `Opcode`: A 6-bit field that specifies the type of instruction (e.g., R-type, I-type, J-type).
  - `Funct`: A 6-bit field used in R-type instructions to specify the exact operation (e.g., ADD, SUB, AND).
- **Outputs:**
  - `RegDst`: Determines the destination register (either `rt` or `rd` field) in R-type instructions.
  - `ALUSrc`: Selects the second operand for the ALU (either a register value or an immediate value).
  - `MemtoReg`: Controls whether the data to be written to the register file comes from memory or the ALU.
  - `RegWrite`: Enables writing to the register file.
  - `MemRead`: Enables reading from memory.
  - `MemWrite`: Enables writing to memory.
  - `BranchEq`: Indicates a branch should occur if the two registers are equal (for `BEQ` instruction).

  - `BranchNe`: Indicates a branch should occur if the two registers are not equal (for `BNE` instruction).
  - `Jump`: Indicates a jump instruction.
  - `Jr`: Indicates a jump register (`JR`) instruction.

Logic

- **Opcode Decoding:** The control signals are generated based on the `Opcode` and `Funct` fields. The control unit decodes the instruction type (e.g., R-type, I-type, J-type) and generates a specific set of control signals for each type.
- **Instruction Types Supported:**
  - **R-type Instructions:** Handled when `Opcode` is `000000`. The specific operation (like ADD, SUB, etc.) is determined by the `Funct` field.
  - **I-type Instructions:** Includes instructions like `LW` (Load Word), `SW` (Store Word), `BEQ` (Branch if Equal), `BNE` (Branch if Not Equal), and others.
  - **J-type Instructions:** Includes `J` (Jump) and `JAL` (Jump and Link).
- **Control Signal Generation:**
  - The control signals (`RegDst`, `ALUSrc`, `MemtoReg`, etc.) are set based on the type of instruction. For example, `ALUSrc` is set when an immediate value is needed as the second operand (e.g., in `LW`, `SW`, `ADDI`).
  - `RegWrite` is activated for instructions that write to a register (e.g., `R-type`, `LW`, `ADDI`).
  - `BranchEq` and `BranchNe` are set for `BEQ` and `BNE` instructions, respectively, controlling whether the program counter should branch to a different address.

Overview of the PWM Module

The VHDL code implements a **Pulse Width Modulation (PWM) module**. PWM is a technique used to control the amount of power delivered to an electronic load by varying the width of the pulses in a pulse train. This implementation has several input parameters and control logic to manage the generation of the PWM signal.

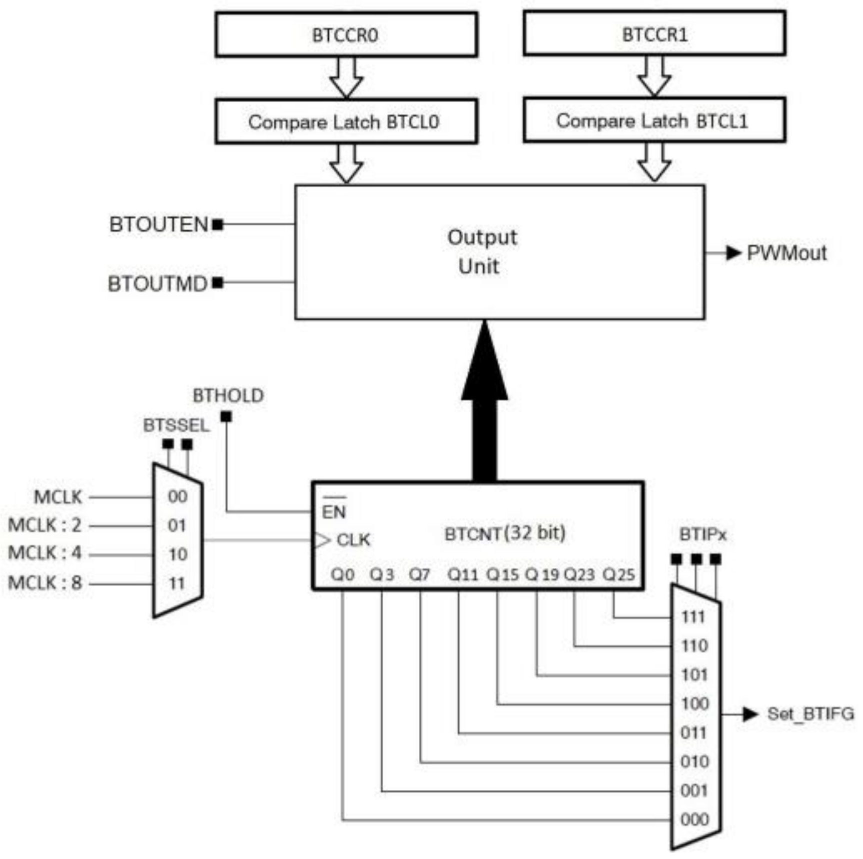


Figure 6 - PWM module and basic timer output compare compatibility

General Logic

- **Clock and Reset Management:**
  - The module uses an input clock (`CLK`) and reset signal (`RST`). When the reset is active, it resets all counters and the PWM output signal.
  - The clock (`CLK`) drives the logic, and the chosen clock (`CHOSEN_CLK`) is selected based on the input selector (`BTSSEL`), which can vary the frequency of the PWM signal by adjusting the clock divider.
- **Counter Logic:**
  - The module maintains several counters: `BTCNT` for counting the main PWM signal period, `bt_counter` for an internal clock divider, and `COUNT_DIVIDER_0` and `COUNT_DIVIDER_1` for dividing the clock frequency based on the `BTSSEL` input.
  - `BTCCR0` and `BTCCR1` define the upper limits for the PWM period (`BTCL0`) and the duty cycle (`BTCL1`).

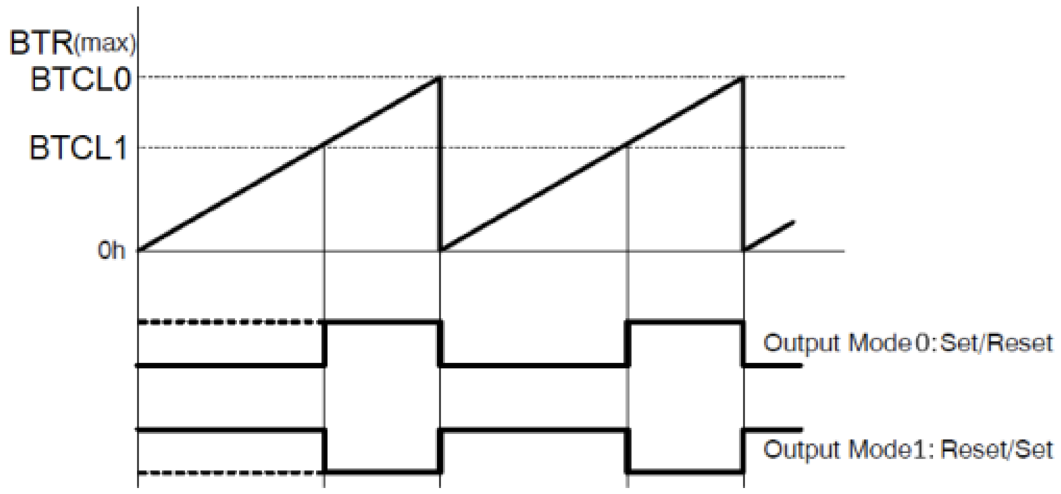


Figure 7 - counter scheme

- **Clock Divider and Selection:**
  - The code defines a process where the clock is divided based on the `BTSSEL` input, creating different frequencies for the PWM signal.
  - Depending on the value of `BTSSEL`, different clock frequencies are chosen, with the `CHOSEN_CLK` toggling between different counts.
- **PWM Signal Generation:**
  - The module generates the PWM output (`PWM_OUT_REG`) by comparing the counter value (`BTCNT`) with `BTCL0` and `BTCL1`.
  - Two modes are supported:
    - **Mode 0:** The PWM signal is high when `BTCNT` is between `BTCL1` and `BTCL0`.
    - **Mode 1:** The PWM signal is high when `BTCNT` is less than `BTCL1`.
  - If the `BTOUTEN` signal is active, the PWM signal is generated; otherwise, the output is set to low.
- **Interrupt Generation:**
  - The module sets the `Set_BTIFG` signal when `bt_counter` reaches its limit, which can be used as an interrupt flag to signal the completion of a PWM period.

Usage and Purpose

- **PWM Signal Control:** This module is used to generate a PWM signal that can be used for applications like controlling the brightness of LEDs, motor speed control, or other power management tasks.
- **Frequency Adjustment:** The module allows adjusting the frequency of the PWM signal using the `BTSSEL` input, which selects different clock dividers.
- **Duty Cycle Adjustment:** The duty cycle, which determines the proportion of time the PWM signal is high, can be adjusted using the `BTCCR1` register, allowing fine control over the output power.

Summary

The PWM module is designed to generate a PWM signal with adjustable frequency and duty cycle, controlled by input parameters. The logic includes clock division, counter management, and condition checking to determine the output signal. This module is highly configurable and can be used in various applications requiring precise control of power delivery through PWM.

Overview of the Division Module

The VHDL code provided implements a **division algorithm** using a sequential logic process. This module divides a `dividend` by a `divisor` to produce a `result` (quotient) and a `remainder`. The operation is controlled by a clock (`divclk`), a reset signal (`rst`), and an enable signal (`ena`). Below is an overview of the logic and operation:

General Logic

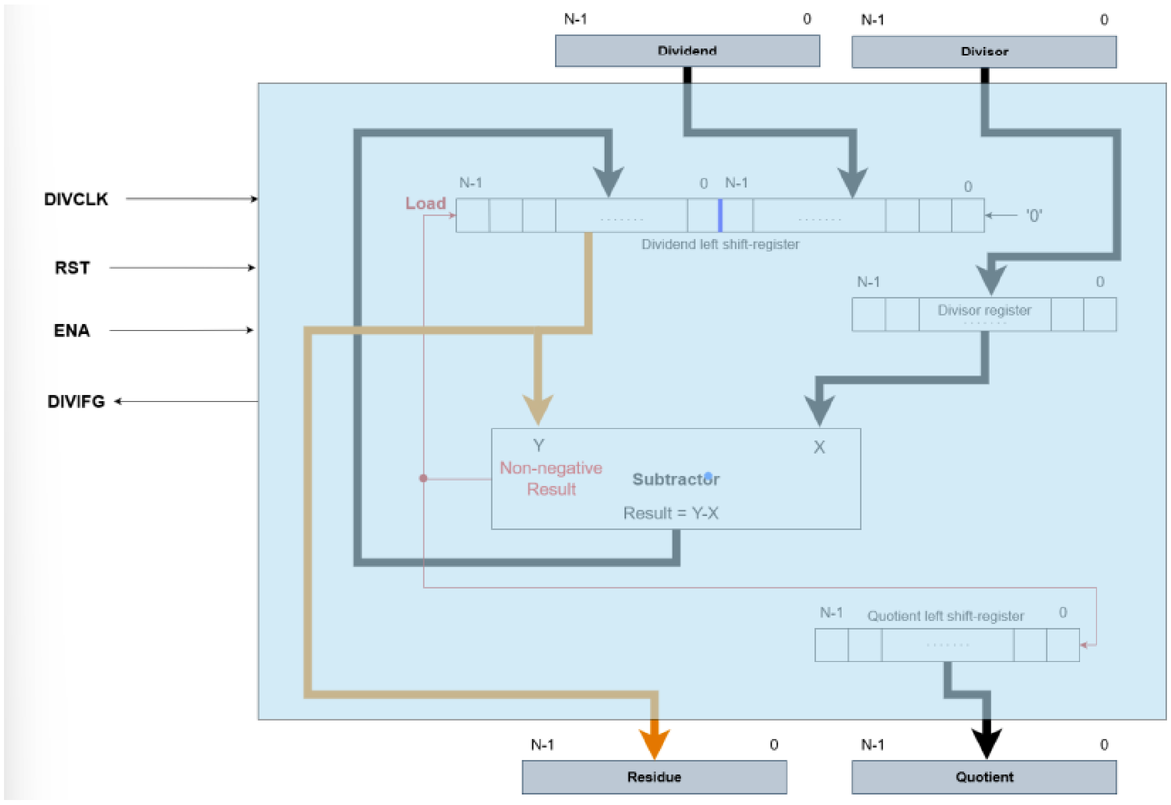


Figure 8 - divider scheme

- **Initialization and Control Signals:**
  - The `rst` (reset) signal initializes all internal signals (`temp_dividend`, `temp_result`, `temp_remainder`, `count`, `busy`, `monkey_place_holder`, and `div_ifg`) to zero.
  - The `ena` (enable) signal starts the division process when active, provided that the division is not already in progress (`busy = ’0’`).
- **Division Process:**
  - The division process is sequential, performed over multiple clock cycles. Each cycle represents one bit of the division process, starting from the most significant bit (MSB) of the `dividend`.
  - The process works by repeatedly subtracting the `divisor` from the `dividend` while shifting the `dividend` to the left and checking if the `dividend` is greater than or equal to the `divisor`.
- **Shifting and Subtraction:**
  - A placeholder signal (`monkey_place_holder`) is used to hold the shifted value of `temp_dividend` combined with the current bit of the `dividend`.

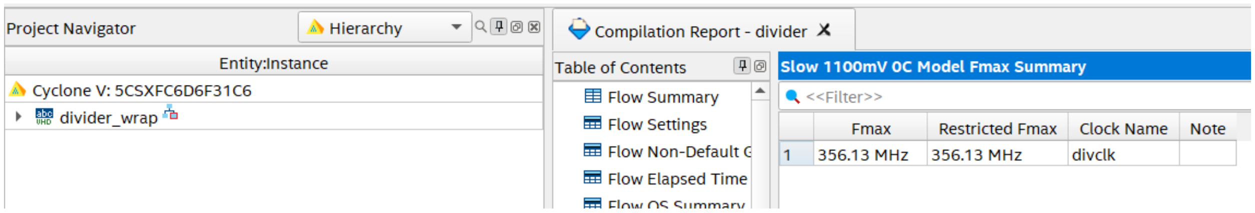
  - In each iteration, if the current `monkey_place_holder` is greater than or equal to the `divisor`, the `divisor` is subtracted from it, and a '1' is recorded in the corresponding bit of `temp_result`. Otherwise, a '0' is recorded.

  - The loop continues, decrementing the bit counter (`count`) until all bits have been processed.

- **Completion and Output:**

  - Once the division process completes (`count` reaches zero), the `busy` signal is cleared, indicating the end of the division. The `result` (quotient) and `remainder` are then assigned to their respective outputs.

  - The `div_ifg` (interrupt flag) signal is set to '1' to indicate that the division operation has finished and the results are ready.


Project Navigator | Hierarchy

Entity:Instance

Cyclone V: 5CSXFC6D6F31C6

divider_wrap

Compilation Report - divider

Table of Contents

Flow Summary

Flow Settings

Flow Non-Default G

Flow Elapsed Time

Flow OS Summary

Slow 1100mV 0C Model Fmax Summary

<<Filter>>

	Fmax	Restricted Fmax	Clock Name	Note
1	356.13 MHz	356.13 MHz	divclk	

Figure 9 - Fmax of our implementation

Summary

- **Division Process:** The code implements a bitwise sequential division algorithm where each bit of the `dividend` is processed in a loop. The process continues until the entire dividend is divided, generating a quotient (`result`) and remainder.

- **Control Logic:** The division is controlled by the `divclk` clock, `rst` reset, and `ena` enable signals. The process ensures that division occurs one bit at a time, updating the quotient and remainder with each step.

- **Status Indication:** The `busy` signal is used to indicate whether the division is in progress, and the `div_ifg` flag is set once the division is complete.

This division module is designed for sequential operation, ensuring that each division process is handled methodically over several clock cycles. It efficiently computes the quotient and remainder using a standard long-division approach in digital logic.

Overview of the IF Module

The VHDL code defines a module that handles the **Program Counter (PC) and instruction memory** for a MIPS processor. Here's what the code does:

1. **Program Counter (PC) Management**:

- The module maintains the Program Counter (PC), which keeps track of the current instruction address.
- The PC is incremented by 4 for the next instruction address, aligning with the instruction word boundaries.

2. **Instruction Memory**:
   - A ROM (Read-Only Memory) is used to store the instructions. The memory is addressed based on the current value of the PC.
   - The instruction is fetched from the ROM and used in the processor.
3. **Control Signals**:
   - The module responds to various control signals (**BranchNe**, **BranchEq**, **Jr**, **Jump**) to determine the next value of the PC.
   - It supports branch instructions (**BEQ**, **BNE**), jump instructions (**J**, **JAL**), and jump register (**JR**).
4. **Interrupt Handling**:
   - The module includes handling for interrupt service routines (**JAL_ISR**) and has provisions to hold the PC during certain interrupt states (**INT_FSM**). The final code does not have interrupt states thus it is not included in the design anymore.
5. **Clock Process**:
   - On every clock cycle, the module updates the PC unless it is reset or held due to an interrupt.
   - When reset is active, the PC is initialized to 0.
6. **Simulation vs FPGA Mode**:

The code differentiates between simulation mode and FPGA implementation, adjusting memory addressing accordingly.

Overview of the idecode

7. **Registers**: It has 32 registers, each 32 bits wide, where data can be stored and read.
8. **Instruction Handling**:
   - It takes an instruction as input and decodes it to determine which registers to read from or write to.

- Depending on the **INT_FSM** signal, the instruction can be overridden to handle interrupts.

9. **Data Reading**:
    - The module reads data from two registers based on the instruction and outputs this data.

10. **Data Writing**:
    - It decides which register to write to based on the control signals (**RegDst** and **MemtoReg**).
    - The data to be written can either come from the ALU result, a memory read, or the program counter (PC).

11. **Sign Extension**:
    - It converts a 16-bit immediate value in the instruction to a 32-bit value, which is necessary for certain operations.

12. **Clock Process**:
    - On every clock cycle, if the reset signal is active, it initializes the registers.
    - If the **RegWrite** signal is active, it writes data to the specified register.

Overview of the GPIO Module

The VHDL code defines a module that handles **General-Purpose Input/Output (GPIO)** operations for a MIPS processor. Here's what the code does:

1. **Address Decoding**:
    - The module decodes specific address bits to generate chip select signals (**CS_LEDR**, **CS_SW**, **CS_HEX0**, etc.), which are used to control different peripherals like LED displays and switches.

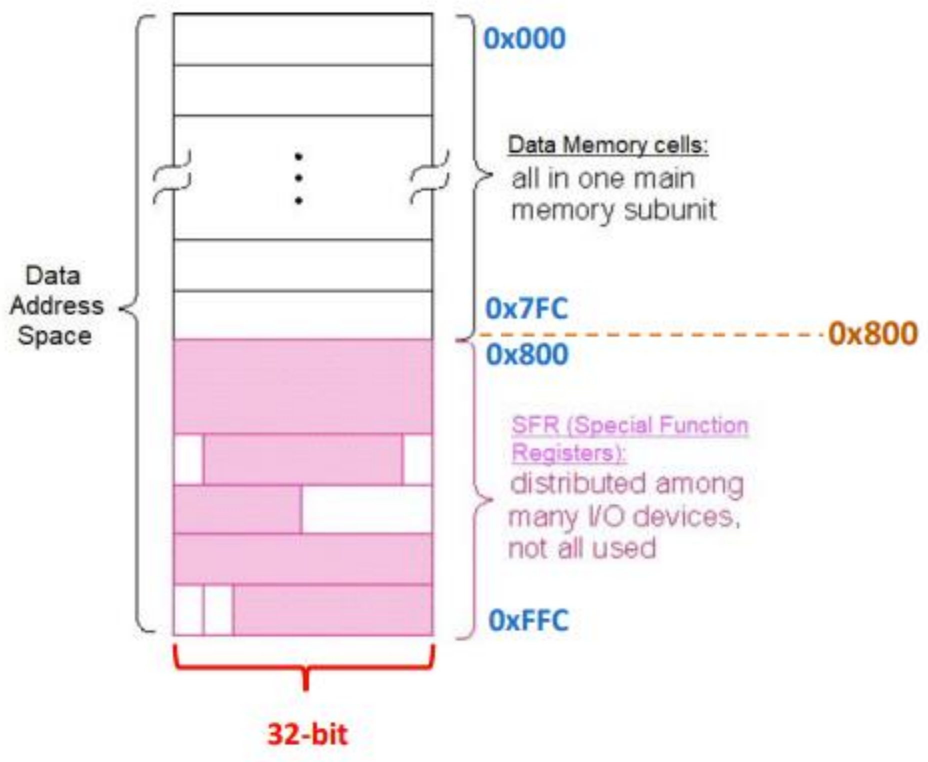

0x000
Data Memory cells:
all in one main memory subunit
Data Address Space
0x7FC
0x800
0x800
SFR (Special Function Registers):
distributed among many I/O devices, not all used
0xFFC
32-bit

Figure 10 - memory map

2. **HEX Display Control**:

   - The module controls six seven-segment displays (**HEX0** to **HEX5**).
   - For each display, it uses a **GPO** (General-Purpose Output) component to handle memory read (**MemRead_Signal**) and write (**MemWrite_Signal**) operations.
   - The data to be displayed is driven from the **DataBus** and is selected based on the chip select signals.

3. **LED Control**:

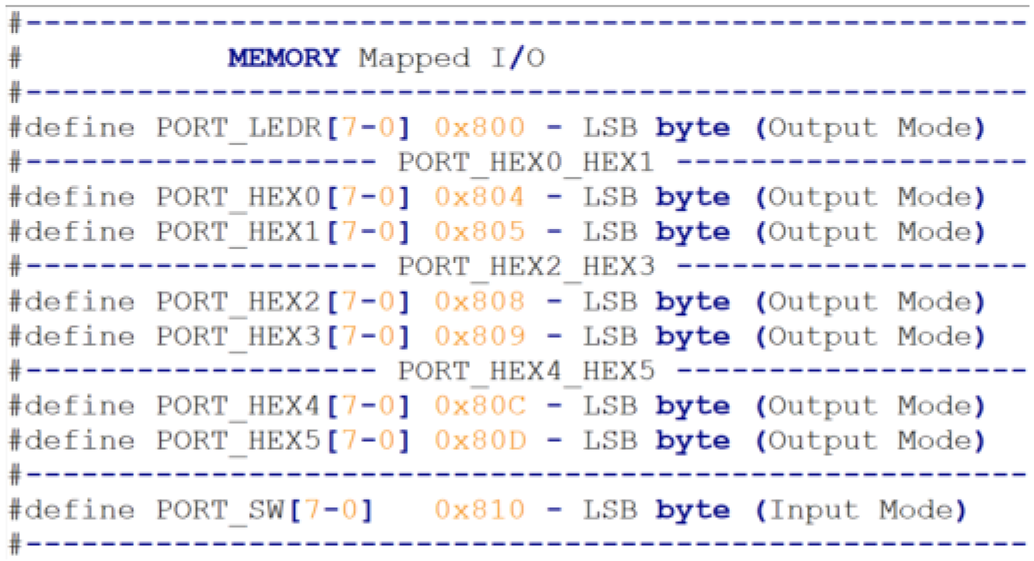
   - Similar to the HEX displays, the **LEDR** output is controlled by a **GPO** component.
   - This component drives the LED display based on data from the **DataBus**.

4. **Switch Input**:

   - The **SW** signal represents input from switches.
   - A **GPI** (General-Purpose Input) component reads the state of the switches and outputs the corresponding data onto the **DataBus**.

5. **Control Signals**:

   - The module uses control signals (**MemRead_Signal**, **MemWrite_Signal**, **reset**, **clock**) to manage the read and write operations for each of these peripherals.



```
#--------------------------------------------------------
#           MEMORY Mapped I/O
#--------------------------------------------------------
#define PORT_LEDR[7-0] 0x800 - LSB byte (Output Mode)
#------------------- PORT_HEX0_HEX1 -------------------
#define PORT_HEX0[7-0] 0x804 - LSB byte (Output Mode)
#define PORT_HEX1[7-0] 0x805 - LSB byte (Output Mode)
#------------------- PORT_HEX2_HEX3 -------------------
#define PORT_HEX2[7-0] 0x808 - LSB byte (Output Mode)
#define PORT_HEX3[7-0] 0x809 - LSB byte (Output Mode)
#------------------- PORT_HEX4_HEX5 -------------------
#define PORT_HEX4[7-0] 0x80C - LSB byte (Output Mode)
#define PORT_HEX5[7-0] 0x80D - LSB byte (Output Mode)
#--------------------------------------------------------
#define PORT_SW[7-0]   0x810 - LSB byte (Input Mode)
#--------------------------------------------------------
```

Figure 11 - memory map of MCU ports

6. **Component Instantiation**:

   - The code instantiates multiple **GPO** and **GPI** components to handle the various input/output devices.
   - The specific data width (**IOWidth**) and type (seven-segment or general output) are configured for each component.

Table 2 - Port Description for GPIO_handler Entity

Port Name	**Direction**	**Size**	**Functionality**
MemRead_Signal	in	1 bit	Memory read signal
clock	in	1 bit	Clock signal
reset	in	1 bit	Reset signal
MemWrite_Signal	in	1 bit	Memory write signal
AddressBus	in	32 bits (AddrBusSize)	Address bus

DataBus	inout	32 bits (DataBusSize)	Data bus
HEX0	out	7 bits	Output for HEX0 display (7-segment)
HEX1	out	7 bits	Output for HEX1 display (7-segment)
HEX2	out	7 bits	Output for HEX2 display (7-segment)
HEX3	out	7 bits	Output for HEX3 display (7-segment)
HEX4	out	7 bits	Output for HEX4 display (7-segment)
HEX5	out	7 bits	Output for HEX5 display (7-segment)
LEDR	out	8 bits	Output for LEDR (LED array)
SW	in	8 bits	Input from SW (switches)

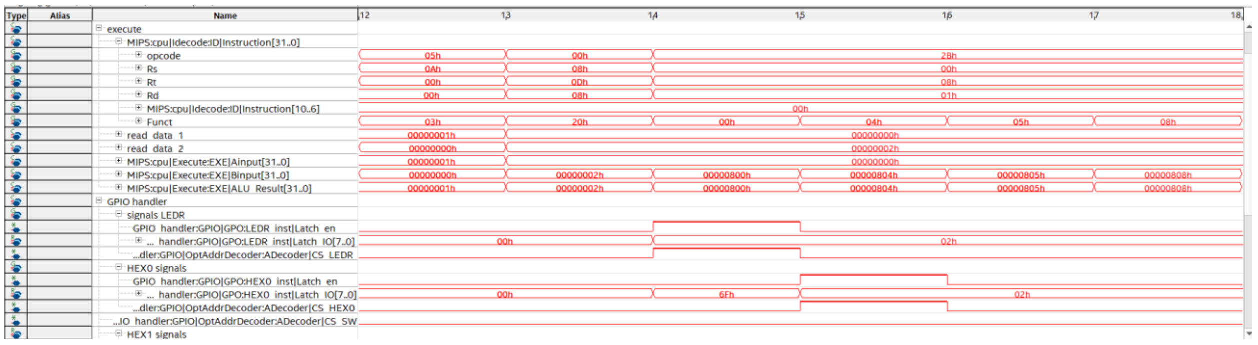

Type
Alias
Name
execute
MIPS:cpu|Idecode:ID|Instruction[31..0]
opcode
Rs
Rt
Rd
MIPS:cpu|Idecode:ID|Instruction[10..6]
Funct
read data 1
read data 2
MIPS:cpu|Execute:EXE|Ainput[31..0]
MIPS:cpu|Execute:EXE|Binput[31..0]
MIPS:cpu|Execute:EXE|ALU Result[31..0]
GPIO handler
signals LEDR
GPIO handler:GPIO|GPO:LEDR inst|Latch en
... handler:GPIO|GPO:LEDR inst|Latch IO[7..0]
...dler:GPIO|OptAddrDecoder:ADecoder|CS LEDR
HEX0 signals
GPIO handler:GPIO|GPO:HEX0 inst|Latch en
... handler:GPIO|GPO:HEX0 inst|Latch IO[7..0]
...dler:GPIO|OptAddrDecoder:ADecoder|CS HEX0
...IO handler:GPIO|OptAddrDecoder:ADecoder|CS SW
HEX1 signals

Figure 12 - STP write to LEDR and HEX

Short Description of Writing to LEDR and HEX Displays in the Simulation:

The waveform illustrates the interaction between the MIPS processor and the GPIO handler module, specifically focusing on writing to the LEDR and HEX displays. Below is a brief description:

7. ***Writing to LEDR****:*
   - *The CS_LEDR (Chip Select for LEDR) signal is asserted when the MIPS processor writes data to the LEDR.*
   - *The Latch_en signal for the LEDR indicates that the data is latched onto the LEDR output.*

  - *The IO[7:0] bus shows the data being written to the LEDR, which corresponds to the value that will be displayed on the LED array.*

8. ***Writing to HEX Displays (HEX0 and HEX1)****:*
   - *Similar to LEDR, the CS_HEX0 and CS_HEX1 signals are asserted when the processor writes to the corresponding HEX displays.*
   - *The Latch_en signals for HEX0 and HEX1 indicate when the data is latched onto the respective HEX displays.*
   - *The IO[7:0] bus shows the binary data being sent to each HEX display, which corresponds to the hexadecimal value shown on the seven-segment displays.*

This waveform demonstrates how the MIPS processor interacts with the GPIO handler to update the state of LEDR and HEX displays, crucial for visual output in embedded systems.

Overview of the Interrupt Handler Module

This is the concept of the architecture of the Interrupt Handler module, however, it was not done due to time limitation. The VHDL code defines an **interrupt handler** module that manages interrupt requests (IRQs) for a system. Here's what the code does:

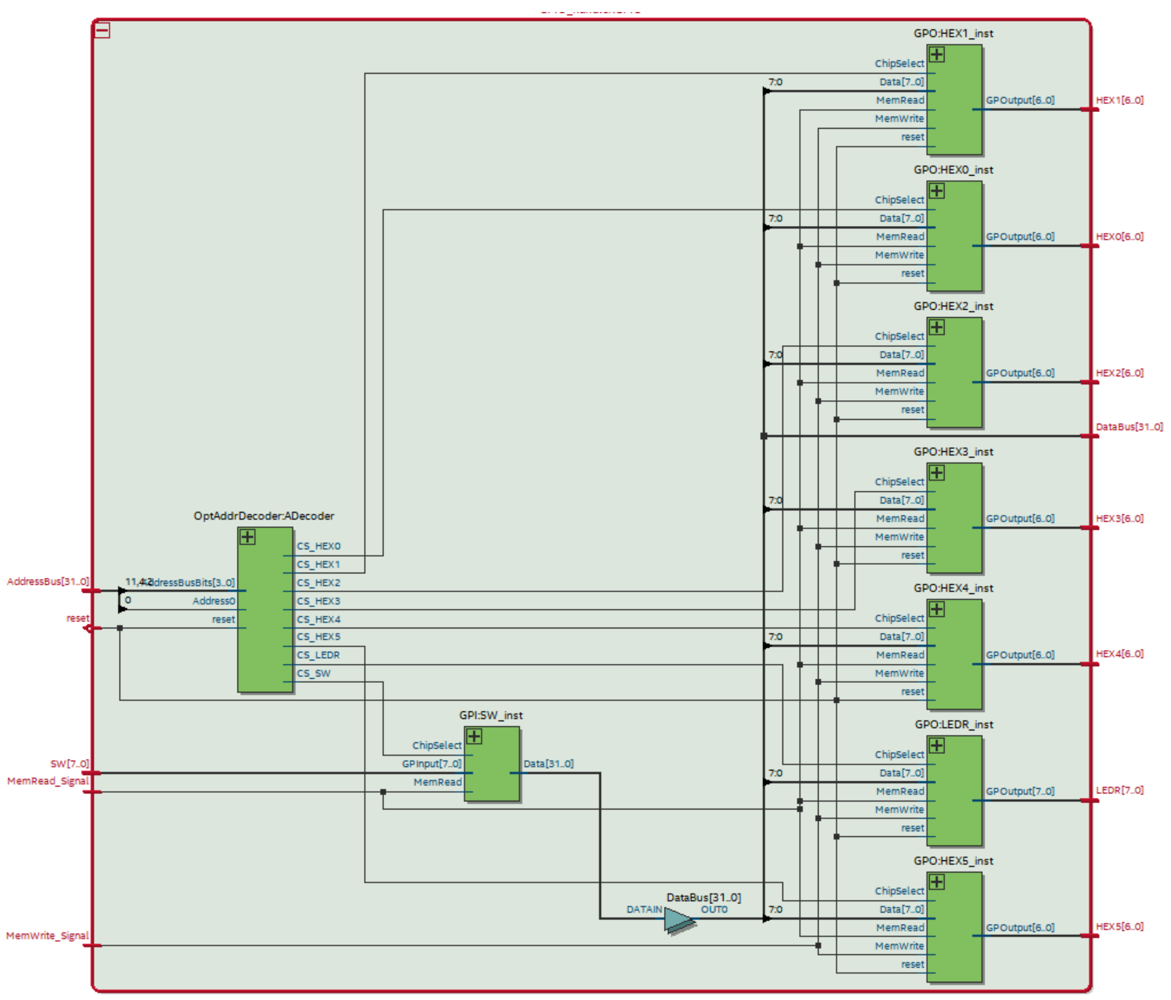
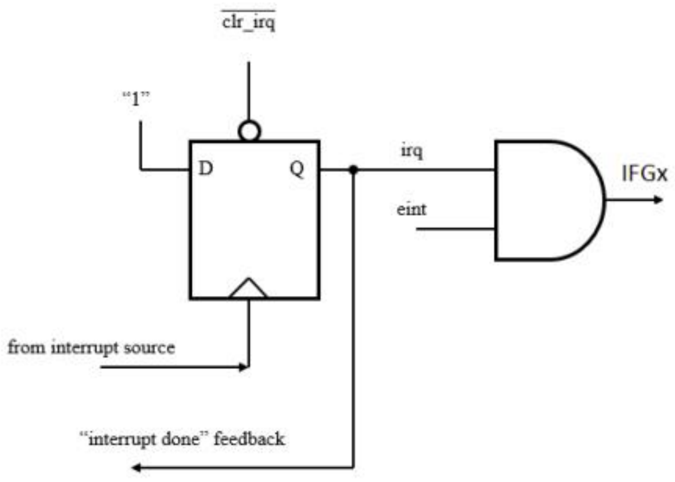


Figure 13 - rtl of handler

1. **Interrupt Request Handling**:



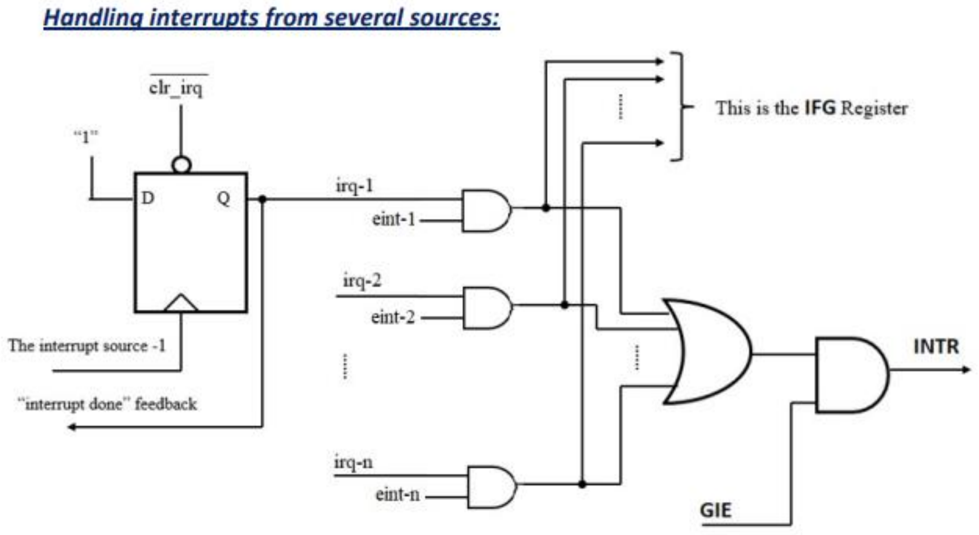


Figure 14 - interrupts handler scheme

- The module handles up to 6 interrupt requests (**IRQ**) and processes them based on different conditions and signals such as keys, a divider (**DIV**), and a set flag (**set_btifg**).
- Each interrupt request is stored in the **IRQ** signal, which is a 7-bit vector, and is managed by individual processes for each interrupt line.

2. **Interrupt Enable and Flag Registers**:

- The **IntrEn** signal is used to enable specific interrupts. The enabled interrupts are determined by writing to the **DataBus** when the **MemWriteBus** signal is active.
- The **IFG** (Interrupt Flag) register stores the status of interrupt requests, which are generated by the combination of **IRQ** and **IntrEn** signals. The flag is also updated when specific memory addresses are written to or read from.

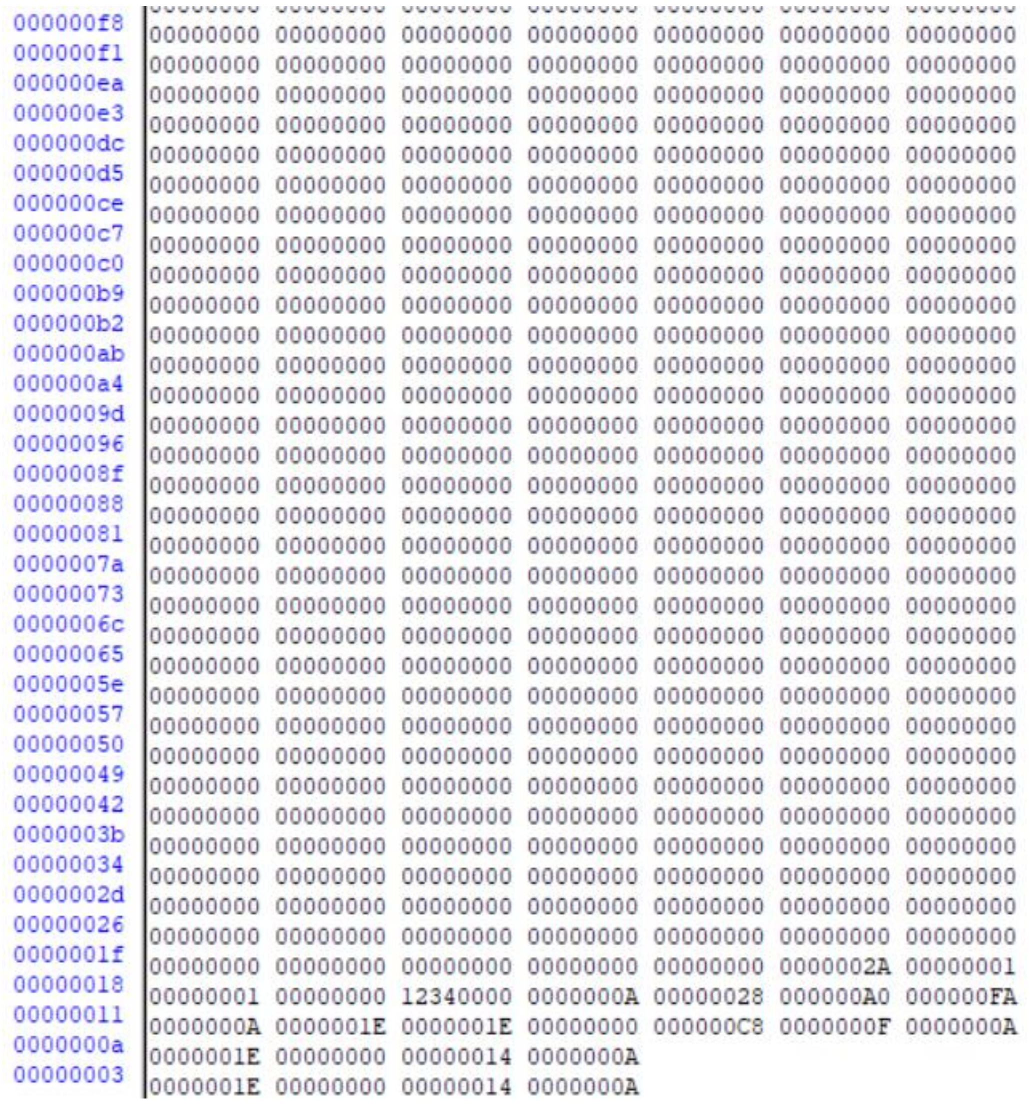
3. **Generating Interrupt Signals**:
   - The module generates an overall interrupt signal (**INTR**) if any of the interrupt flags are set (**IFG**) and if global interrupts are enabled (**GIE**).
   - The **IRQ_OUT** signal is outputted, representing the status of all interrupts.
4. **Interrupt Acknowledge (INTA) Handling**:
   - The **INTA** signal is used to acknowledge interrupts. The acknowledgment is delayed using **INTA_Delayed** to manage the timing of interrupt clear operations.
   - When **INTA** is asserted and specific conditions are met (e.g., certain **typereg** values), the corresponding interrupt in **IRQ** is cleared.
5. **Process Execution**:
   - For each interrupt source (e.g., **key_1**, **key_2**, **key_3**), there is a dedicated process that sets or clears the corresponding bit in the **IRQ** vector based on the clock signal, reset signal (**rst**), and the clear IRQ signal (**CLR_IRQ**).
6. **DataBus Handling**:
   - The module reads from or writes to the **DataBus** depending on specific memory addresses, updating the interrupt enable (**IntrEn**) and flag registers (**IFG**), and outputting the current status based on the address.
7. **Typereg Management**:
   - The **typereg** signal is updated based on the active interrupts in **IRQ**. It represents the type of interrupt currently being processed.

Table 3 - Port Description for interupt_handler Entity

Port Name	**Direction**	**Size**	**Functionality**
set_btifg	in	1 bit	Set interrupt flag
key_1	in	1 bit	Key 1 input
key_2	in	1 bit	Key 2 input
key_3	in	1 bit	Key 3 input
DIV	in	1 bit	Divide signal input
clk	in	1 bit	Clock signal
rst	in	1 bit	Reset signal
INTR	out	1 bit	Interrupt request output
IRQ_OUT	out	7 bits (IrqSize + 1)	Interrupt request outputs

MemReadBus	in	1 bit	Memory read bus
MemWriteBus	in	1 bit	Memory write bus
AddressBus	in	12 bits (AddrBusSize)	Address bus
DataBus	inout	32 bits (DataBusSize)	Data bus
INTA	in	1 bit	Interrupt acknowledge
GIE	in	1 bit	Global interrupt enable

Reference



```
000000f8 00000000 00000000 00000000 00000000 00000000 00000000 00000000
000000f1 00000000 00000000 00000000 00000000 00000000 00000000 00000000
000000ea 00000000 00000000 00000000 00000000 00000000 00000000 00000000
000000e3 00000000 00000000 00000000 00000000 00000000 00000000 00000000
000000dc 00000000 00000000 00000000 00000000 00000000 00000000 00000000
000000d5 00000000 00000000 00000000 00000000 00000000 00000000 00000000
000000ce 00000000 00000000 00000000 00000000 00000000 00000000 00000000
000000c7 00000000 00000000 00000000 00000000 00000000 00000000 00000000
000000c0 00000000 00000000 00000000 00000000 00000000 00000000 00000000
000000b9 00000000 00000000 00000000 00000000 00000000 00000000 00000000
000000b2 00000000 00000000 00000000 00000000 00000000 00000000 00000000
000000ab 00000000 00000000 00000000 00000000 00000000 00000000 00000000
000000a4 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0000009d 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000096 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0000008f 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000088 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000081 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0000007a 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000073 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0000006c 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000065 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0000005e 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000057 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000050 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000049 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000042 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0000003b 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000034 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0000002d 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000026 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0000001f 00000000 00000000 00000000 00000000 00000000 0000002A 00000001
00000018 00000001 00000000 12340000 0000000A 00000028 000000A0 000000FA
00000011 0000000A 0000001E 0000001E 00000000 000000C8 0000000F 0000000A
0000000a 0000001E 00000000 00000014 0000000A
00000003 0000001E 00000000 00000014 0000000A
```

Figure 15 - memory of our testing from model sim simulation

The code that was used is:

```
.data

 # Define some data in memory

 var1: .word 10       # variable 1 initialized to 10

 var2: .word 20       # variable 2 initialized to 20

 result: .word 0      # variable to store result
 t2_result: .word 0     # to store $t2

 t3_result: .word 0     # to store $t3

 t4_result: .word 0     # to store $t4

 t5_result: .word 0     # to store $t5

 t6_result: .word 0     # to store $t6

 t7_result: .word 0     # to store $t7

 t8_result: .word 0     # to store $t8

 t9_result: .word 0     # to store $t9

 s0_result: .word 0     # to store $s0

 s1_result: .word 0     # to store $s1

 s2_result: .word 0     # to store $s2

 s3_result: .word 0     # to store $s3

 s4_result: .word 0     # to store $s4

 s5_result: .word 0     # to store $s5

 s6_result: .word 0     # to store $s6

 s7_result: .word 0     # to store $s7

 s8_result: .word 0     # to store $s8


.text

.globl main

main:

 # Load values from memory to registers

 lw   $t0, var1        # Load var1 into $t0

 lw   $t1, var2        # Load var2 into $t1


 # Test ADD, SUB, and ADDI

 add  $t2, $t0, $t1     # $t2 = $t0 + $t1

 sw   $t2, t2_result    # Store $t2 into memory
 sub  $t3, $t1, $t0     # $t3 = $t1 - $t0
 sw   $t3, t3_result    # Store $t3 into memory
 addi $t4, $t0, 5       # $t4 = $t0 + 5
 sw   $t4, t4_result    # Store $t4 into memory

 # Test MUL, AND, OR, XOR
#  mul  $t5, $t0, $t1     # $t5 = $t0 * $t1
#  sw   $t5, t5_result    # Store $t5 into memory
 and  $t6, $t0, $t1     # $t6 = $t0 & $t1
 sw   $t6, t6_result    # Store $t6 into memory
 or   $t7, $t0, $t1     # $t7 = $t0 | $t1
 sw   $t7, t7_result    # Store $t7 into memory
 xor  $t8, $t0, $t1     # $t8 = $t0 ^ $t1
 sw   $t8, t8_result    # Store $t8 into memory

 # Test ANDI, ORI, XORI
 andi $t9, $t0, 0x0F    # $t9 = $t0 & 0x0F
 sw   $t9, t9_result    # Store $t9 into memory
 ori  $s0, $t0, 0xF0    # $s0 = $t0 | 0xF0
 sw   $s0, s0_result    # Store $s0 into memory
 xori $s1, $t0, 0xAA    # $s1 = $t0 ^ 0xAA
 sw   $s1, s1_result    # Store $s1 into memory

 # Test SLL, SRL
 sll  $s2, $t0, 2       # $s2 = $t0 << 2
 sw   $s2, s2_result    # Store $s2 into memory
 srl  $s3, $t1, 1       # $s3 = $t1 >> 1
 sw   $s3, s3_result    # Store $s3 into memory
 # Test LUI (Load Upper Immediate)
 lui  $s4, 0x1234       # $s4 = 0x12340000
 sw   $s4, s4_result    # Store $s4 into memory

 # Test BEQ, BNE
 beq  $t0, $t1, equal   # If $t0 == $t1, jump to equal
 bne  $t0, $t1, notequal # If $t0 != $t1, jump to notequal

equal:
 addi $s5, $zero, 1     # $s5 = 1 (flag for equal)
 sw   $s5, s5_result    # Store $s5 into memory
 j end

notequal:
 addi $s5, $zero, 0     # $s5 = 0 (flag for not equal)
 sw   $s5, s5_result    # Store $s5 into memory

 # Test SLT, SLTI
 slt  $s6, $t0, $t1     # $s6 = 1 if $t0 < $t1, else $s6 = 0
 sw   $s6, s6_result    # Store $s6 into memory
 slti $s7, $t0, 15      # $s7 = 1 if $t0 < 15, else $s7 = 0
 sw   $s7, s7_result    # Store $s7 into memory

 # Test J, JR, JAL
 jal  func              # Jump to function
 j end
```

```
func:
  addi $t8, $zero, 42    # $s8 = 42
  sw   $t8, s8_result    # Store $s8 into memory
  jr   $ra               # Return from function


end:
  # Infinite loop to stop execution
  j end
```