IMPLEMENT MCONTROL6 INSTRUCTION TRIGGER IN C-CLASS AND EVALUATING WITH MANUAL 3-WINDOW SIMULATION

SUMMER INTERNSHIP – I REPORT

Submitted by

PRIYADHARSHAN L – 2023105055

VIGNESH B K -2023105509

in partial fulfilment for the award of the degree of

BACHELOR OF ENGINEERING

in

ELECTRONICS AND COMMUNICATION ENGINEERING

COLLEGE OF ENGINEERING, GUINDY

ANNA UNIVERSITY: CHENNAI 600025

JULY 2025

IMPLEMENT MCONTROL6 INSTRUCTION TRIGGER IN C-CLASS AND EVALUATING WITH MANUAL 3-WINDOW SIMULATION

SUMMER INTERNSHIP - I REPORT

Submitted by

PRIYADHARSHAN L – 2023105055

VIGNESH B K -2023105509

in partial fulfilment for the award of the degree of

BACHELOR OF ENGINEERING

in

ELECTRONICS AND COMMUNICATION ENGINEERING

COLLEGE OF ENGINEERING, GUINDY

ANNA UNIVERSITY: CHENNAI 600025

JULY 2025

ANNA UNIVERSITY: CHENNAI 600025

BONAFIDE CERTIFICATE

Certified that this report titled as, "**IMPLEMENT MCONTROL6 INSTRUCTION TRIGGER IN C-CLASS AND EVALUATING WITH MANUAL 3-WINDOW SIMULATION**" is the Bonafide work of **PRIYADHARSHAN L(2023105055) & VIGNESH B K (2023105509) for 5th Semester** who carried out the project work for **Summer Internship-I**, in the month of June 2025 under my supervision. Certified further that to the best of my knowledge, the work reported herein does not form part of any other thesis or dissertation based on which a degree or award was conferred on an earlier occasion on this or any other candidate

SIGNATURE

Dr. Nitya Ranganathan

SHAKTI Team,

Indian Institute of Technology,

Madras, Chennai-600036

SIGNATURE

Mr. Sriram

SHAKTI Team,

Indian Institute of Technology,

Madras,Chennai-600036

ACKNOWLEDGEMENT

I express my sincere gratitude to the Dean, Dr.K.S.Easwarakumar, Professor, College of Engineering, Guindy for his support throughout the project.

I extend my heartfelt appreciation to my Head of the Department, Dr. M.A Bhagyaveni, Professor, Department of Electronics & Communication Engineering for her enthusiastic support and guidance throughout the project.

I am immensely grateful to my project supervisors Dr. Nitya Ranganathan and Mr. Sriram from the SHAKTI Team, for her unwavering assistance, patience, valuable guidance, technical mentorship, and encouragement in my project.

I would like to thank the entire SHAKTI Team for their unwavering support in helping us throughout this project.

I would also like to sincerely thank Mr. Sivakumar Anandan, alumnus of the Department of Electronics and Communication Engineering, College of Engineering Guindy, for bringing us this opportunity and for his continuous encouragement and support that made this project possible.

Certificate page

ABSTRACT

This project focuses on the implementation of a hardware instruction trigger using the mcontrol6 mechanism in the C-Class RISC-V core developed by the Shakti group. The mcontrol6 trigger allows the processor to halt execution when a specific instruction address is fetched, thereby enabling precise and efficient hardware breakpoints for real-time debugging. The trigger is integrated into the instruction fetch stage of the core and configured through standard debug CSRs such as tdata1, tdata2, and tselect.

To validate the implementation, a three-window simulation setup is used involving the C-Class core running in a simulator (e.g., Verilator), OpenOCD as the debug server, and GDB as the front-end interface for configuring triggers and observing core halts. Debug test programs provided by the Shakti team are used to verify the correctness of the trigger behavior under different scenarios. The project also explores the possibility of deploying the design on an FPGA to evaluate the trigger performance on real hardware.

Overall, the project enhances the debugging capabilities of the C-Class core by enabling instruction-level breakpoints and contributes to the broader RISC-V ecosystem by providing a clean, standards-compliant trigger implementation based on the latest debug specification.

TABLE OF CONTENTS

CHAPTER 1
INTRODUCTION

1.1 PROJECT MOTIVATION

In modern processor design, especially with open-source architectures like RISC-V, efficient debugging mechanisms are critical for development and validation. As systems grow more complex, software-based debugging tools alone are not sufficient to capture low-level execution issues such as illegal memory accesses, misaligned jumps, or unexpected instruction execution. These challenges highlight the need for hardware-level debug support.

The motivation behind this project is to implement a hardware instruction trigger, specifically the mcontrol6 trigger, in the C-Class RISC-V core developed by the Shakti group. The mcontrol6 trigger is designed to halt the processor when a particular instruction address is fetched, providing a breakpoint-like feature directly in hardware. This enhances real-time debugging, allowing developers to pause execution precisely when needed, inspect the state of the processor, and resume without disrupting the software environment.

This project aims to bridge the gap between software tools and hardware debugging by integrating a minimal, standards-compliant trigger mechanism into the processor pipeline.

1.2 PROJECT OVERVIEW

This project involves the design and RTL implementation of a hardware instruction trigger using the mcontrol6 trigger type defined in the RISC-V Debug Specification. The implementation is targeted at the C-Class 5-stage in-order processor core, written in Bluespec System Verilog (BSV).

The trigger logic is embedded into the Instruction decode stage of the pipeline. When enabled via CSR (Control and Status Registers), the processor continuously monitors the current program counter (PC) and compares it with a target address. If a match occurs and the exec bit in the trigger is set, the processor halts and enters debug mode.

To validate the design, a three-window simulation setup is used:

1. The C-Class core is simulated using Verilator.
2. OpenOCD connects to the core's debug interface.
3. GDB is used to set trigger registers and monitor halts.

1.3 PROBLEM STATEMENT

C-Class core/debug RTL implementation and simulation: Implement atleast one instruction trigger (hardware breakpoint, mcontrol6 is preferred) in C-Class. Evaluate with manual 3-window simulation with simulator, openocd and gdb. Also, test trigger programs from risc-v debug tests (setup provided by Shakti team). Extra credit: Test on FPGA.

The C-Class RISC-V core, while functional, did not include support for mcontrol6 triggers prior to this project. Without such triggers, developers cannot halt execution based on a specific instruction fetch. This leads to slower debug cycles and more difficult validation.

The project addresses this limitation by implementing mcontrol6, a simplified hardware trigger that halts the core on instruction fetch. This implementation must be tightly integrated into the core's pipeline, CSR system, and debug control logic, and must be verifiable via standard RISC-V tools like OpenOCD and GDB

1.4 OBJECTIVES

The main objectives of the project are:

1) To implement the mcontrol6 instruction trigger (type 6) in the C-Class RISC-V core according to the RISC-V Debug Specification 1.0.
2) To integrate the trigger logic into the Instruction Fetch (IF) stage of the core pipeline.
3) To enable external configuration of trigger CSRs (tdata1, tdata2, tselect) via GDB through OpenOCD.
4) To validate the functionality using manual simulation with Verilator, OpenOCD, and GDB.
5) To test trigger behaviour using Shakti-provided debug test programs.
6) (Optional) To test the trigger on an FPGA platform for hardware validation

1.5 SUMMARY

This project establishes the need for hardware triggers in RISC-V debugging, and highlights the role of mcontrol6 as a minimal and effective solution for instruction-based breakpoints. The project targets the C-Class core and aims to integrate this trigger into the processor pipeline, validate its behavior through simulation and debugging tools, and optionally demonstrate it on FPGA. The following chapters will cover the design process, theoretical background, implementation details, testing methodology, and final results.

CHAPTER 2

OVERVIEW OF TRIGGERS AND 3-WINDOW SIMULATION

Shakti C-Class v4.6.0 Core

- The original, unmodified version (v4.6.0) of the Shakti C-Class core was successfully built and simulated**.**
  The 3 Window setup was built and software breakpoint was tested and executed successfully.

2.1 TRIGGERS:

Triggers can cause a breakpoint exception, entry into Debug Mode, or a trace action without having to execute a special instruction. This makes them invaluable when debugging code from ROM. They can trigger on execution of instructions at a given memory address, or on the address/data in loads/stores A trigger matches when the conditions that it specifies (e.g. a load from a specific address) are met. A trigger fires when a trigger that matches performs the action configured for that trigger. Triggers do not fire while in Debug Mode

2.2 TRIGGER MODULE REGISTERS:

These registers are CSRs, accessible using the RISC-V csr opcodes and optionally also using abstract debug commands. They are the only mechanism to access the triggers. All tdata registers follow write-any-read-legal semantics. If a debugger writes an unsupported configuration, the register will read back a value that is supported.The Trigger Module registers are only accessible in machine and Debug Mode to prevent untrusted user code from causing entry into Debug Mode without the OS’s permission.

Here we are using Trigger Data 1 (tdata1, at 0x7a1)

Trigger Data 2 (tdata2, at 0x7a2)

Mcontrol6 (mcontrol6, at 0x7a1)

2.1.1 Trigger Data 1 (tdata1, at 0x7a1):

This register provides access to the trigger selected by tselect. The reset values listed here apply to every underlying trigger. This CSR is read/write.

2.1.2 Trigger Data 2 (tdata2, at 0x7a2):

This optional register is only accessible in M-mode and Debug Mode and provides various control bits related to triggers. This CSR is read/write.

2.1.3 Mcontrol6 (mcontrol6, at 0x7a1):

This register provides access to the trigger selected by tselect. The reset values listed here apply to every underlying trigger. This register is accessible as tdata1 when type is 6. This replaces mcontrol in newer implementations and serves to provide additional functionality. Address and data trigger implementation are heavily dependent on how the processor core is implemented. To accommodate various implementations, execute, load, and store address/data triggers may fire at whatever point in time is most convenient for the implementation.

This CSR is read/write.

2.3 3 Window Simulation

To validate the implementation of the mcontrol6 instruction trigger, a three-window simulation setup was used. This approach is commonly employed for testing hardware-level debugging features in RISC-V processors and involves using three separate software tools, each

running in a dedicated terminal or window. These tools work together to simulate the processor, establish a debug communication channel, and enable external debugging control**.**

1. Simulator Window

The first window runs the simulation of the C-Class RISC-V core using a hardware simulation tool such as Verilator or VCS. This simulates the behavior of the processor, including the trigger logic integrated into the instruction fetch stage. It acts as the target hardware for debugging and produces the output waveform or execution logs.

2. OpenOCD Window

The second window launches OpenOCD (Open On-Chip Debugger), which serves as a bridge between the simulated processor and the debugger. OpenOCD communicates with the core's debug interface using protocols like JTAG or Debug Module Interface (DMI). It allows GDB to send and receive debug commands such as setting or reading CSR values (e.g., tdata1, tdata2, tselect).

3. GDB Window

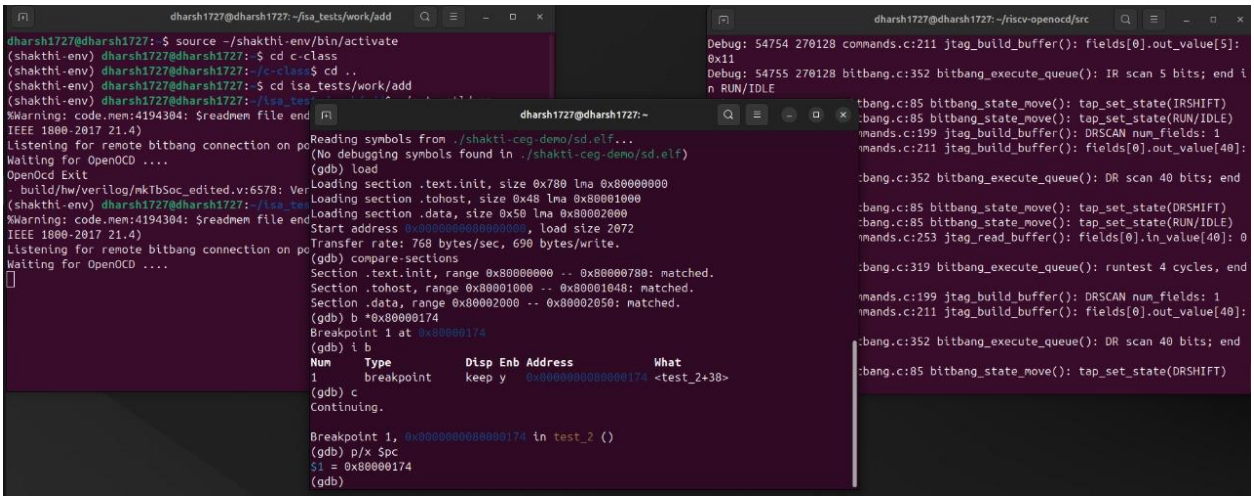
The third window runs GDB (GNU Debugger), which is used to interact with the simulated processor through OpenOCD. Within this interface, users can:

- Configure trigger registers (tdata1, tdata2)
- Start or halt execution
- Monitor if the trigger correctly halts the processor upon instruction fetch
- Inspect register and memory contents during debug mode

Purpose of the Setup

The 3-window simulation enables complete end-to-end testing of the hardware trigger without needing physical hardware. It ensures that:

- The trigger activates correctly when the instruction address matches
- The core halts and enters debug mode
- External tools (GDB, OpenOCD) can configure and monitor trigger behavior.



CHAPTER 3

MODIFICATIONS AND WORK FLOW

3.1 CODE MODIFICATIONS:

The trigger implementation is not defined in c-class, so around 12 files needed modifications to implement this trigger logic inorder to work properly. Those files include generated csrbox_bsv files such as csrbox.bsv, csrbox_grp2.bsv, csr_types.bsv, csrbox.defines, csrbox.decoder, csr_probe.bsv and the files in src folder of c-class like stage1.bsv, stage5.bsv, ccore_types.bsv, trap.defines, riscv.bsv.

3.1.1 csrbox_grp2.bsv:

Purpose:

This module manages Group 2 CSRs including:

Debug: dpc, dcsr

Triggers: tdata1, tdata2, mcontrol6

It acts as a responder to CSR accesses from the core and provides trigger-related signals to the pipeline stages.

Role of csrbox_grp2.bsv

1) This file holds the actual values of the trigger registers (tdata1, tdata2)

2)Implements WARL behavior (force valid values)

3)Converts the packed tdata1 into structured Mcontrol6

4)Returns all trigger information to csrbox for wiring into stage1

5)Implements mav_upd_on_trigger, called by stage5 through csrbox

```
method Bit#(64) mv_csr_tdata1;      // inserted
method Bit#(64) mv_csr_tdata2;
method Bit#(64) mv_csr_mcontrol6;
method TriggerData0 mv_trigger_data;   // inserted
```

```
function Reg#(Bit#(64)) warlReg_tdata1(Reg#(Bit#(64)) r);
       return (interface Reg;
            method Bit#(64) _read = r;
            method Action _write(Bit#(64) x);
              Mcontrol6 val = unpack(truncate(x));
              val.type_t = 6;
              val.dmode = 1;
              r._write(zeroExtend(pack(val)));
           endmethod
       endinterface);
endfunction: warlReg_tdata1

Reg#(Bit#(64)) rg_tdata1_warl <- mkReg(0);
Reg#(Bit#(64)) rg_tdata1 = warlReg_tdata1(rg_tdata1_warl);

function Reg#(Bit#(64)) warlReg_tdata2(Reg#(Bit#(64)) r);
        return (interface Reg;
             method Bit#(64) _read = r;
             method Action _write(Bit#(64) x);
                  r._write(x);
             endmethod
        endinterface);
endfunction: warlReg_tdata2

Reg#(Bit#(64)) rg_tdata2_warl <- mkReg(0);
Reg#(Bit#(64)) rg_tdata2 = warlReg_tdata2(rg_tdata2_warl);
```

```
`TDATA1 : begin                                          /*inserted*/
    Bit#(64) readdata = rg_tdata1;
    rg_resp_to_core <= CSRResponse { hit: True, data: readdata
               `ifdef rtldump ,csr_updated: True `endif };
    let word = fn_csr_op(req.writedata, readdata, req.funct3);
    rg_tdata1 <= truncate(word);
end                                                      /*inserted*/

`TDATA2 : begin                                          /*inserted*/
    Bit#(64) readdata = zeroExtend(rg_tdata2);
    rg_resp_to_core <= CSRResponse { hit: True, data: readdata
            `ifdef rtldump ,csr_updated: True `endif };
    let word = fn_csr_op(req.writedata, readdata, req.funct3);

    rg_tdata2 <= truncate(word);

end                                                      /*inserted*/

`MCONTROL6: begin                                        /*inserted*/
     Bit#(64) readdata = zeroExtend(rg_tdata1);
     rg_resp_to_core <= CSRResponse { hit: True,data: readdata
            `ifdef rtldump ,csr_updated: False `endif };
     let word = fn_csr_op(req.writedata, readdata, req.funct3);
     rg_tdata1 <= truncate(word);

 end
```

```
method Bit#(64) mv_csr_tdata1 = rg_tdata1;
method Bit#(64) mv_csr_tdata2 = rg_tdata2;
method Mcontrol6 mv_csr_mcontrol6 = unpack(truncate(rg_tdata1));

method TriggerData0 mv_trigger_data();
    return TriggerData0 {
        tdata1: truncate(rg_tdata1),
        tdata2: truncate(rg_tdata2)
    };
endmethod
```

3.1.2 csrbox.bsv:

This is the top-level CSR controller module. It connects all the CSR submodules and provides access to them for other pipeline stages (e.g., stage1, stage5).

1)Instantiates grp2 (for debug CSRs like dpc, dcsr, tdata1, tdata2.

2)Forwards CSR read/write requests from the core to the appropriate group

3)Exposes CSRs via sbread interface

4)Handles trap/ret logic (e.g., what happens on ecall, mret, dret, etc.

5)Wires in the instruction trigger logic using:

mv_csr_tdata1, mv_csr_tdata2

mv_csr_mcontrol6 (decoded form of tdata1)

mav_upd_on_trigger() to update debug state

```
method Bit#(64) mv_csr_tdata1;      // inserted
method Bit#(64) mv_csr_tdata2;
method Bit#(64) mv_csr_mcontrol6;
method TriggerData0 mv_trigger_data;   // inserted
```

```
method Action mav_upd_on_trigger(Bit#(7) cause, Bit#(64) pc, Bit#(64) tval);
```

```
   method mv_csr_tdata1 = grp2.mv_csr_tdata1;                    //inserted
   method mv_csr_tdata2 = grp2.mv_csr_tdata2;
   method mv_csr_mcontrol6 = zeroExtend(pack(grp2.mv_csr_mcontrol6));
   method TriggerData0 mv_trigger_data = grp2.mv_trigger_data;   //inserted

endinterface
method Action mav_upd_on_trigger(Bit#(7) cause, Bit#(64) pc, Bit#(64) tval);
        grp2.mav_upd_on_trigger(cause, pc);  //line 435
endmethod
```

3.1.3 stage5.bsv

This is the writeback stage (stage 5) of the SHAKTI C-Class pipeline. It is the last stage of instruction execution where:

1)Results are written back to registers

2)Traps are finalized

3)Interrupts and debug entries are handled

4)Triggers (here mcontrol6) cause a redirect to debug mode.

Handles Traps (Exceptions, Interrupts, Triggers):

This is where the **instruction-trigger (mcontrol6) logic** happens

1)Detects that the instruction caused a hardware breakpoint

2)Calls the CSR logic to **enter debug mode**

3)Flushes the pipeline and redirects to debug entry (tvec)

```
else `endif begin                                          //inserted
    if (trapout.cause == `INSTR_TRIGGER_CAUSE) begin
      `logLevel(stage5, 0, $format("[%2d]STAGE5 : Detected INSTRUCTION TRIGGER TRAP", hartid))
      let tvec = fuid.pc;
      csr.mav_upd_on_trigger(trapout.cause, fuid.pc, trapout.mtval);
      wr_flush <= WBFlush{flush: True, newpc : tvec, fencei: False
        `ifdef supervisor , sfence: False `endif
        `ifdef hypervisor , hfence: False `endif };
      `logLevel( stage5, 0, $format("[%2d]STAGE5 : INSTR TRIGGER Trap to *TVEC:%h",hartid, tvec))
    end
    else begin
      let tvec = fuid.pc;
      csr.mav_upd_on_trap(trapout.cause, fuid.pc, trapout.mtval
        `ifdef hypervisor , trapout.mtval2 `endif );
      wr_flush <= WBFlush{flush: True, newpc : tvec, fencei: False
        `ifdef supervisor , sfence: False `endif
        `ifdef hypervisor , hfence: False `endif };
      `logLevel( stage5, 0, $format("[%2d]STAGE5 : Going to *TVEC:%h",hartid, tvec))
    end                                                    //inserted
```

3.1.4 Stage1.bsv:

1) It Properly reads tdata1, tdata2, enable signals from csrbox via the common interface.
2) Uses check_trigger() to match current PC against mcontrol6 settings.
3) If a trigger match occurs, it sets trap = True and assigns cause = trig_cause, which will be passed downstream.
4) Fully supports compressed instructions (with correct PC handling).
5) Integrates cleanly into the pipeline (transfers trap info to stage2 via PIPE1).

```
`ifdef triggers
  Vector#(`trigger_num, Wire#(TriggerData)) v_trigger_data1 <- replicateM(mkWire());
  Vector#(`trigger_num, Wire#(Bit#(`xlen))) v_trigger_data2 <- replicateM(mkWire());
  Vector#(`trigger_num, Wire#(Bool)) v_trigger_enable <- replicateM(mkWire());
`endif
```

check_trigger logic:

1) **Iterates through all mcontrol6 triggers** enabled via GDB (tdata1, tdata2, etc.).
2) **Compares PC (compare_value) with tdata2** based on match type (==, ≥, <).
3) **Checks if the trigger is enabled** and of type mcontrol6 with execute == 1.
4) **Supports chaining** of triggers using the chain bit and previous trap result.
5) Returns trap = True if a trigger matches → stage5 raises an exception.

```
`ifdef triggers

  function ActionValue#(Tuple2#(Bool, Bit#(`causesize))) check_trigger (Bit#(`vaddr) pc,
                        Bit#(32) instr `ifdef compressed , Bool compressed `endif ) = actionvalue
    Bool trap = False;
    Bit#(`causesize) cause = `Breakpoint;
    Bit#(`xlen) compare_value = pc; //modified
    Bool chain = False;
    for(Integer i=0; i < `trigger_num; i=i+1)begin
      `logLevel( stage1, 3, $format("[%2d]STAGE1: Trigger[%2d] Data1: ",hartid, i,
                                                                fshow(v_trigger_data1[i])))
      `logLevel( stage1, 3, $format("[%2d]STAGE1: Trigger[%2d] Data2: ",hartid, i,
                                                                fshow(v_trigger_data2[i])))
      `logLevel( stage1, 3, $format("[%2d]STAGE1: Trigger[%2d] Enable: ",hartid, i,
                                                                fshow(v_trigger_enable[i])))
      if(v_trigger_enable[i] &&& v_trigger_data1[i] matches tagged MCONTROL6 .mc &&&                    //i
                            ((!trap && !chain) || (chain && trap)) &&& mc.execute == 1)begin
       Bit#(`xlen) trigger_compare = v_trigger_data2[i];  // Always match against PC --- inserted
                                                                                                       //
        if(mc.matched == 0)begin
          if(trigger_compare == compare_value)
            trap = True;
          else if(chain)
            trap = False;
        end
        if(mc.matched == 2)begin
          if(compare_value >= trigger_compare)
            trap = True;
          else if(chain)
            trap = False;
        end
        if(mc.matched == 3)begin
          if(compare_value < trigger_compare)
            trap = True;
          else if(chain)
            trap = False;
        end

       if(trap) begin
          `logLevel(stage1, 0, $format("[%2d]STAGE1: TRIGGERED at PC=%h, cause=%h", hartid, pc, cause));
       end
```

```
      `ifdef debug
        if(trap && mc.action_ == 1)begin
          cause = `HaltTrigger;
          cause[`causesize - 1] = 1;
        end
      `endif
        chain = unpack(mc.chain);
      end
    end
    return tuple2(trap, cause);
  endactionvalue;
`endif
```

```
  method Action trigger_data1(Vector#(`trigger_num, TriggerData) t);
    for(Integer i=0; i<`trigger_num; i=i+1)
      v_trigger_data1[i] <= t[i];
  endmethod
  method Action trigger_data2(Vector#(`trigger_num, Bit#(`xlen)) t);
    for(Integer i=0; i<`trigger_num; i=i+1)
      v_trigger_data2[i] <= t[i];
  endmethod
  method Action trigger_enable(Vector#(`trigger_num, Bool) t);
    for(Integer i=0; i<`trigger_num; i=i+1)
      v_trigger_enable[i] <= t[i];
  endmethod
```

3.1.5 pipe_ifcs.bsv:

1)Allow CSR-driven trigger setup (stage1)

2)Propagate instruction-trigger-based trap info (stage1 → stage5)

3)stage5 inform csrbox via dedicated update call

```
`ifdef triggers
  method Action trigger_data1(Vector#(`trigger_num, TriggerData) t);
  method Action trigger_data2(Vector#(`trigger_num, Bit#(`xlen)) t);
  method Action trigger_enable(Vector#(`trigger_num, Bool) t);
`endif
endinterface:Ifc_s1_common
```

3.1.6 riscv.bsv

1) Connects mv_tdata1, mv_tdata2, mv_tenable (from CSR in stage5) to stage1.common.trigger_*
2) Acts as the top-level pipeline integration file connecting all pipeline stages (stage0 to stage5) and control/status modules like csrbox.
3) Wires together interfaces such as trigger_data1, trigger_data2, and trigger_enable from csrbox to stage1 for debug trigger evaluation.
4) Serves as the central point where pipeline-wide configurations, control signals, and method connections are established.

```
mkConnection(stage1.common.ma_csr_misa_c, stage5.csrs.mv_csr_misa_c);
`ifdef triggers
      mkConnection(stage1.common.trigger_data1, stage5.csrs.mv_tdata1);
      mkConnection(stage1.common.trigger_data2, stage5.csrs.mv_tdata2);
      mkConnection(stage1.common.trigger_enable, stage5.csrs.mv_tenable);
`endif
```

3.1.7 csr_types.bsv

1)Defines core types for CSR request/response handling and includes support for RISC-V debug triggers via TriggerData0 and Mcontrol6.

2)Provides structures to store and decode trigger CSRs (tdata1, tdata2) into meaningful fields like execute, match_type, etc.

3)Enables the stage1 pipeline to evaluate instruction-based hardware breakpoints using parsed trigger data.

```
typedef struct {
    Bit#(32) tdata1;
    Bit#(32) tdata2;
} TriggerData0 deriving (Bits, FShow);
```

```
typedef struct {
  Bit#(4)   type_t;         // bits [3:0]
  Bit#(1)   dmode;        // bit [4]
  Bit#(6)   maskmax;      // bits [10:5]
  Bit#(2)   sizehi;       // bits [12:11]
  Bit#(1)   select;       // bit [13]
  Bit#(1)   timing;       // bit [14]
  Bit#(2)   size;         // bits [16:15]
  Bit#(4)   action_;       // bits [20:17]
  Bit#(1)   chain;        // bit [21]
  Bit#(4)   match_type;   // bits [25:22]
  Bit#(1)   machine;      // bit [26]
  Bit#(1)   supervisor;   // bit [27]
  Bit#(1)   user;         // bit [28]
  Bit#(1)   execute;      // bit [29]
  Bit#(1)   store;        // bit [30]
  Bit#(1)   load;         // bit [31]
} Mcontrol6 deriving (Bits, Eq, FShow);
```

3.1.8 csr_probe.bsv:

1) Reads and returns the value of a given CSR using its address.
2) Uses a large case block to match against all implemented CSRs.
3) Fetches the values from soc.soc_sb.sbread, which provides safe access to read-only CSR values.

```
`TDATA1 : return zeroExtend(soc.soc_sb.sbread.mv_csr_tdata1);
`TDATA2 : return zeroExtend(soc.soc_sb.sbread.mv_csr_tdata2);
`MCONTROL6 : return zeroExtend(soc.soc_sb.sbread.mv_csr_mcontrol6);
```

3.1.9 csrbox_decoder.bsv

1) This module defines how CSR addresses are validated, identified, and controlled during execution.
2) Provides utility functions for debug name mapping, trigger CSR access, and side-effect management.
3) Essential for safe, extensible support of standard and custom CSRs including debug triggers like TDATA1/2 and MCONTROL6.

```
`TDATA1: valid = True;
`TDATA2: valid = True;
`MCONTROL6: valid = True;
```

```
`TDATA1: return "c1953_tdata1";
`TDATA2: return "c1954_tdata2";
`MCONTROL6: return "c1955_mcontrol6";
```

```
case(addr)
`MVENDORID , `MARCHID , `MIMPID , `MHARTID , `PMPADDR4 , `PMPADDR5 , `PMPADDR6 , `PMPADDR7 , `PMPADDR8 , `PMPADDR9 , `PMPADDR10 , `PMPADDR11 , `PMPADDR12 , `PMPADDR13 ,
`PMPADDR14 , `PMPADDR15 , `MCOUNTINHIBIT , `MHPMEVENT7 , `MHPMCOUNTER7 , `MHPMEVENT8 , `MHPMCOUNTER8 , `MHPMEVENT9 , `MHPMCOUNTER9 , `MHPMEVENT10 , `MHPMCOUNTER10 , `MHPMEVENT11 ,
`MHPMCOUNTER11 , `MHPMEVENT12 , `MHPMCOUNTER12 , `MHPMEVENT13 , `MHPMCOUNTER13 , `MHPMEVENT14 , `MHPMCOUNTER14 , `MHPMEVENT15 , `MHPMCOUNTER15 , `MHPMEVENT16 , `MHPMCOUNTER16 ,
`MHPMEVENT17 , `MHPMCOUNTER17 , `MHPMEVENT18 , `MHPMCOUNTER18 , `MHPMEVENT19 , `MHPMCOUNTER19 , `MHPMEVENT20 , `MHPMCOUNTER20 , `MHPMEVENT21 , `MHPMCOUNTER21 , `MHPMEVENT22 ,
`MHPMCOUNTER22 , `MHPMEVENT23 , `MHPMCOUNTER23 , `MHPMEVENT24 , `MHPMCOUNTER24 , `MHPMEVENT25 , `MHPMCOUNTER25 , `MHPMEVENT26 , `MHPMCOUNTER26 , `MHPMEVENT27 , `MHPMCOUNTER27 ,
`MHPMEVENT28 , `MHPMCOUNTER28 , `MHPMEVENT29 , `MHPMCOUNTER29 , `MHPMEVENT30 , `MHPMCOUNTER30 , `MHPMEVENT31 , `MHPMCOUNTER31 ,`TDATA1 , `TDATA2 , `MCONTROL6 : return False;
default: return True;
endcase
endfunction:fn_csr_flush
```

3.1.10 csr.defines.bsv

1) These macros define addresses for TDATA1, TDATA2, and MCONTROL6, key CSRs used for implementing **instruction-based hardware breakpoints**.
2) MCONTROL6 is an alias for TDATA1, indicating its contents follow the **mcontrol version 6 format**
3) These constants are referenced throughout the pipeline and CSR modules to enable **debug-trigger decoding and GDB integration**.

```
`define TDATA1 12'h7A1
`define TDATA2 12'h7A2
`define MCONTROL6 12'h7A1
```

11)trap.defines.bsv

1) INSTR_TRIGGER_CAUSE is a custom exception code (0x7F) used to mark traps caused by instruction-trigger breakpoints.
2) It enables precise identification of **trigger-based debug traps** by setting mcause accordingly.
3) Used in stage5.bsv during trap updates and visible to external debuggers via CSR reads.

```
`define INSTR_TRIGGER_CAUSE 7'h7f  // unique code
```

12)ccore_types.bsv

1) It defines core-wide data types and enumerations used across pipeline stages and the CSR system.

2) Contains key structures like Cause, Exc_Code, and flags for PipelineStage, enabling clean exception signaling.

3)The trigger trap system uses this file to assign and interpret custom causes, like INSTR_TRIGGER_CAUSE.

4)Helps decouple core logic from external files (e.g. csrbox.bsv) by centralizing core-specific type definitions.

Here just the importing of csr_types.bsv is used in order to make it visible for the higher level files.

3.2 WORKFLOW

1. GDB writes TDATA1 (mcontrol6) and TDATA2 via OpenOCD

2. csrbox_grp2 stores them in internal registers (e.g., rg_tdata1, rg_tdata2)

3. stage1.bsv reads tdata1/tdata2 using mv_trigger_data

4. It matches against the current PC using mcontrol6 logic

5. If match → raise a flag to trigger a debug trap

6. stage5.bsv detects this trigger and calls:

   csrbox.mav_upd_on_trigger(cause, pc, tval)

7. csrbox calls grp2.mav_upd_on_trigger

   Sets DPC = current PC

   Sets DCSR cause

   Enters Debug Mode

RESULT:

REFERENCES

[1] Shakti Processor Program – C-Class Core (Version 4.6.0), GitLab Repository:
https://gitlab.com/shaktiproject/cores/c-class/-/tree/4.6.0?ref_type=tags

[2] Shakti C-Class Core Source Files, GitLab Repository:
https://gitlab.com/shaktiproject/cores/c-class/-/tree/4.6.0/src?ref_type=tags

[3] Shakti Debug Trigger Project Setup – Internal Documentation:
https://docs.google.com/document/d/1ULOyApK_KxQpYgcpcxYDryNKXq63OqLxa4uaGWN_EVI/edit?tab=t.0

[4] RISC-V Debug Test Program Files – Shakti Team Shared Resource :
https://drive.google.com/file/d/1h_f9NgB_8m2fS6uCnKP1Oho-3x1MpBEl/view