RISC-V FPU and Interrupt CPU Extensions

line 1: 1st Qian Hao Lam
line 2: Electrical and *Computer Engineering*
line 3: *Johns Hopkins University*
line 4: Maryland, United States
line 5: qlam6@jh.edu

***Abstract*— This work presents the design and verification of a RISC-V RV32IMF softcore extended with IEEE-754 single-precision floating-point support and Trap/Break interrupts in both single-cycle and five-stage pipeline microarchitectures. A unified trap/CSR controller cleanly decouples exception entry and return, enabling reuse across both cores. The pipelined design adds flush and replay logic for exceptions and external interrupts, while the single-cycle version integrates the same controller without modification. Correctness is validated through directed assembly tests and FPGA debugging with an Integrated Logic Analyzer. The resulting implementation demonstrates a compact and pedagogically clear approach to extending teaching-oriented RISC-V cores with floating-point and robust interrupt handling.**

I. Introduction

Computer Architecture EN525.612 and EN525.712 Classes currently use MIPS designs to teach core principles. RISC V's open, modular ISA is a modern, future proof alternative to MIPS, which is no longer under active development. Prof. Beser has a working implementation of single-cycle, multi-cycle, and basic pipelined RISC-V architecture, but completing extensions for integer multiply/divide (RV32M), floating point (RV32F), interrupts, and cache is a lengthy process—this independent study will accelerate their development. For the Summer Semester, the deliverables are a pipelined CPU with Floating point support and interrupts, and a single cycle CPU with interrupts.

A. Contributions

1. Unified Trap/CSR Controller usable by both microarchitectures
2. FPU interface with hazard controls and PC stalling
3. Precise exception/interrupt handling with flush & replay in 5-stage pipeline CPU
4. Assembly programs demonstrate functionality running on Nexys A7 FPGA.

II. Background and Related Work

A. Teaching Microarchitectures and the MIPS→RISC-V Migration

EN.525.612 and EN.525.712 Computer Architecture classes have long relied on a classic five-stage MIPS(Microprocess without Interlocked Pipeline Stages) pipeline (IF,ID,EX, MEM, WB) to demonstrate forwarding, hazard detection, and CPU core architectures. In this work, we migrate to RISC-V while intentionally retaining the MIPS datapath patterns to preserve teachability. We then change only what the ISA differences require. Some notable changes are: decode logic, control signals, exception/interrupt handling, interrupt register datapaths.

B. Floating Point extension and IEEE-754 Background

The RV32IMF implementation is built off a RV32IM pipelined Core. The FPU Unit was extracted from Yamin Li's MIPS pipelined with FPU program [1]. The result is a Core that is compliant with the RISC-V F-extension and conforms to IEEE-754 single-precision [2] seen in Figure 1. Additionally, since RISC-V ISA supports rounding mode, rounding mode decoding is added to the extracted Yamin Li's FPU Unit. the provided rounding modes are the following:

- Round to nearest, ties to even
- Round towards zero
- Round towards + infinity (Round Up)
- Round towards – infinity(Round Down)

The FPU supports the same type of instructions Yamin Li's Pipelined CPU with FPu supports. They are the **Fadd**, **Fsub**, **Fmul**, **Fdiv**, **Fsqrt**, **FLW**, and **FSW**. These instructions are sufficient to perform most floating-point operations.

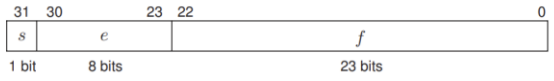


Figure 1: IEEE 754 format of a single-precision floating-point number [3]

C. *RISC-V trap/CSR model background and related work*

Yamin Li's interrupt implementation for both single cycle and pipeline are not easily extracted to port into our RISC-V designs. This is because unlike the FPU extension, an interrupt extension is different between MIPS and RISCV. As a result, We had to implement a custom machine-mode CSR file and trap controller that provides exception and interrupt handling. The CSRs include the following:

- Mtvec (Machine trap vector base and mode)
- Mstatus (Machine status registers)
- MIE (Machine Interrupt Enable)
- MEPC(Trap return PC)
- Mcause (Machine cause registers)

For Trap semantics, our RISC-V design follows the RISC-V Privileged Spec's precise-trap semantics. On a taken exception or enabled interrupt, MEPC captures the PC, MCAUSE records the cause, PC redirects to MTVEC, after ISR (Interrupt service routine) runs, MRET restores MIE and resumes at MEPC. Only 4 Traps are implemented. They are

- Unimplemented Instruction
- External Interrupt
- ALU Overflow
- Machine mode system call (ECALL)

D. *RV32IMF Overview (ISA subset, FP regs vs INT regs)The word "data" is plural, not singular.*

Our core targets RV32IMF in machine mode. RV32I provides 32 integer registers, and fixed 32-bit encodings for ALU, loads/stores, branches, JAL/JALR, and system ops (CSR, ECALL/EBREAK/MRET). RV32M adds integer multiply/divide with results written to rd and forwarding muxes. RV32F adds a 32-entry FP register file (f0–f31) and single-precision ops (FADD.S, FSUB.S, FMUL.S, FDIV.S) plus minimal moves/conversions. Rounding and exception behavior follows the RISC-V F-extension and IEEE-754 (see II-B). The scope of our RV32IMF is machine mode only. There are no VM/caches, no misaligned accesses traps.

The microarchitecture is intentionally designed to exclude the full set of RISC-V supported ISAs. However, it incorporates sufficient subsets to demonstrate the microarchitectures for the FPU, single-cycle interrupt, and pipelined interrupt. Excluding the full set of RISC-V supported ISA will allow future students to perform exercises in implementing instruction extensions.

E. *Single-cycle Interrupt included ISA support*

Our RV32 single-cycle interrupt only includes the CSRRS (CSR Read and Set), CSRRW (CSR Read and Write), MRET, and ECALL instructions. These four CSR instructions are sufficient. For instance the overflow exception in the picture below is handled successfully using only CSRRW, CSRRS and MRET.

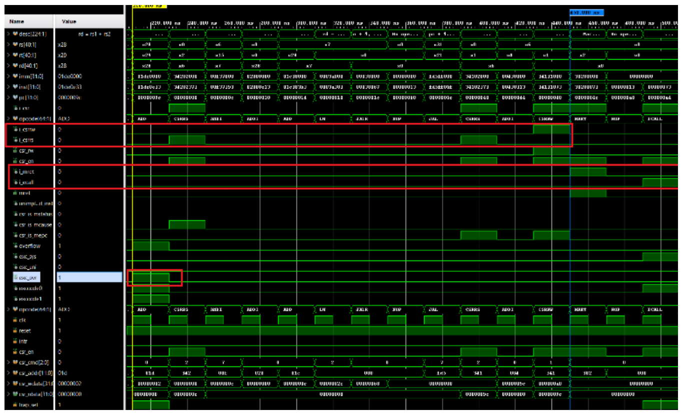


Figure 2: Overflow and Ecall instructions

F. *Pipelined Interrupt*

Our RV32 pipelined interrupt only includes the CSRRS (CSR Read and Set), CSRRW (CSR Read and Write), MRET, and ECALL instructions. Similar to the single-cycle interrupt, these four CSR instructions are sufficient.

III. FPU Design and Integration

A. *Initial State and Plan*

The FPU is built from a pipelined RV32 IM core. The RV32IM core supports the following instructions.

Table 1: Supported RV32IM Instructions

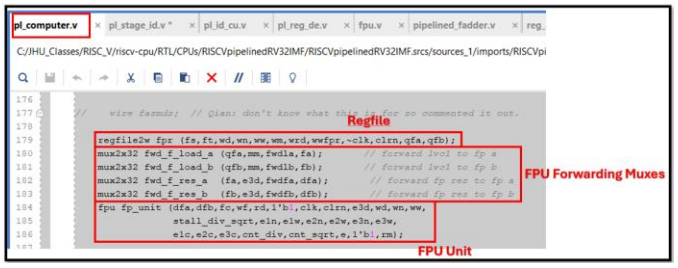
i_slt	i_add	i_sub	i_and	i_or
i_xor	i_slli	i_srli	i_srai	i_jalr
i_addi	i_andi	i_ori	i_xori	i_lw
i_sw	i_beq	i_bne	i_lui	i_jal

The initial RV32IM pipelined core was chosen as a "built off point" as it already contain the infrastructure for a pipelined floating-point extension. The plan is as follows:

- Extract the FPU unit from Yamin Li's MIPS Pipelined with FPU CPU and add it to the RV32 IM
- Extract and port the same MIPS FPU forwarding muxes
- Extract and port the same MIPS FPU regfile2w
- Connect the required ports to the FPU
- Add the 7 floating point instructions (FADD.S, FSUB.S, FMUL.S, FDIV.S, FLW/FSW) to the control unit
- Modify forwarding muxes as needed

B. *Porting the FPU Unit, regfile and Muxes*

Porting the FPU unit was straightforward. The FPU Unit verilog code was added as a source to the RV32IMF and instantiated in the pl_computer.v level see Figure 3. This level was chosen as it mirrors Yamin Li's MIPS design. The regfile and FPU forwarding mux were added as well.


pl_computer.v
pl_stage_id.v *
pl_id_cu.v
pl_reg_de.v
fpu.v
pipelined_fadder.v
reg_
C:/JHU_Classes/RISC_V/riscv-cpu/RTL/CPUs/RISCVpipelinedRV32IMF/RISCVpipelinedRV32IMF.srcs/sources_1/imports/RISCVpi
Regfile
FPU Forwarding Muxes
FPU Unit

```
//    wire fasmds;  // Qian: don't know what this is for so commented it out.

    regfile2w fpr (fs,ft,wd,wn,ww,wm,wrd,wwfpr,~clk,clrn,qfa,qfb);
    mux2x32 fwd_f_load_a (qfa,mm,fwdla,fa);          // forward lvol to fp a
    mux2x32 fwd_f_load_b (qfb,mm,fwdlb,fb);          // forward lvol to fp b
    mux2x32 fwd_f_res_a  (fa,e3d,fwdfa,dfa);          // forward fp res to fp a
    mux2x32 fwd_f_res_b  (fb,e3d,fwdfb,dfb);          // forward fp res to fp b
    fpu fp_unit (dfa,dfb,fc,wf,rd,1'b1,clk,clrn,e3d,wd,wn,ww,
                 stall_div_sqrt,e1n,e1w,e2n,e2w,e3n,e3w,
                 e1c,e2c,e3c,cnt_div,cnt_sqrt,e,1'b1,rm);
```

Figure 3: pl_computer.v instantiation of FPU code

The next step was to connect all input and output signals that the FPU unit, forwarding muxes and regfile have with the RV32IM pipeline. These signals are from the IU (Instruction Unit). Regfile inputs from IU fs, ft, wmo, wrn, wwfpr. Forwarding muxes inputs are mmo, fwdla, fwdlb, fwdfa, fwdfb. The FPU inputs are fd, fc, wf, and rounding mode (rm) These signals can be seen in Figure 4. The output wires to the IU are stall, data write (ed), forwarding logic signals (e1w , e2w, e3w, e1n, e2n, e3n), write enable signals (ww and wn) and write data signal (wd).

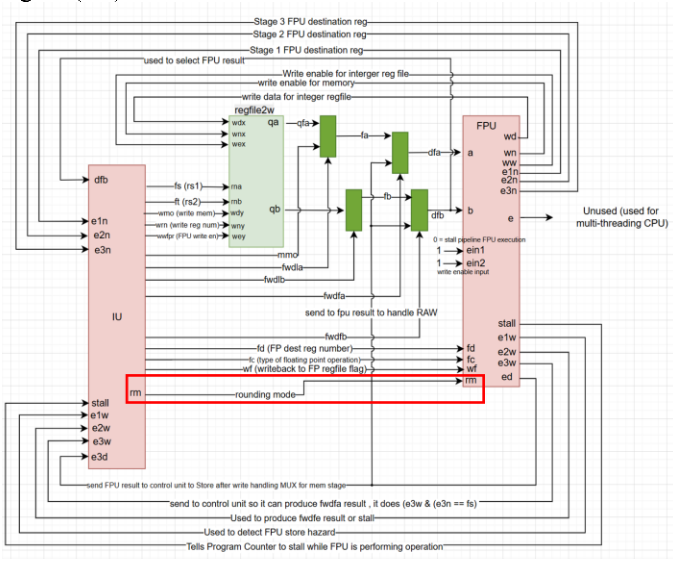

Stage 3 FPU destination reg
Stage 2 FPU destination reg
Stage 1 FPU destination reg
used to select FPU result
Write enable for interger reg file
write enable for memory
write data for integer regfile
regfile2w
wdx
wnx
wex
qa
qb
rna
rnb
wdy
wny
wey
qfa
fa
dfa
fb
dfb
FPU
a
b
wd
wn
ww
e1n
e2n
e3n
e
Unused (used for multi-threading CPU)
0 = stall pipeline FPU execution
1 ein1
1 ein2
write enable input
dfb
e1n
e2n
e3n
fs (rs1)
ft (rs2)
wmo (write mem)
wrn (write reg num)
wwfpr (FPU write en)
mmo
fwdla
fwdlb
fwdfa
IU
send to fpu result to handle RAW
fwdfb
fd (FP dest reg number)
fc (type of floating point operation)
wf (writeback to FP regfile flag)
fd
fc
wf
rm
stall
e1w
e2w
e3w
ed
rm
rounding mode
stall
e1w
e2w
e3w
e3d
send FPU result to control unit to Store after write handling MUX for mem stage
send to control unit so it can produce fwdfa result , it does (e3w & (e3n == fs)
Used to produce fwdfe result or stall
Used to detect FPU store hazard
Tells Program Counter to stall while FPU is performing operation

Figure 4: FPU Unit, forwarding registers, and regfile

C. *MIPS to RISCV Port changes*

The MIPS to RISCV FPU copy and paste cannot be done without understanding the differences between RISCV and MIPS. Here are the differences.

1) *Opcode format changes*

This change was relatively trivial as all opcodes are sourced in a single source of truth manner from the instruction decode stage. See Figure 5.

```
// instruction fields
wire       [6:0] op    = inst[6:0];          // op
wire       [4:0] rs1    = inst[19:15];       // rs1
wire       [4:0] rs2    = inst[24:20];        // rs2
wire       [4:0] rd    = inst[11:7];        // rd
wire       [2:0] func3 = inst[14:12];         // func3
wire       [6:0] func7 = inst[31:25];         // func7
wire      [15:0] imm    = inst[15:00];       // immediate
wire      [25:0] addr = inst[25:00];         // address
assign   csr_addr = inst[31:20];        // CSR ADDRESS
```

Figure 5: RISC-V Opcode decoding

Once properly extracted and decoded, the instruction fields was readily accepted by the FPU.

2) *Load and Store of FP regs.*

RISC-V Floating-point load and store uses a base + 12-bit signed immediate, while MIPS uses a base + 16-bit signed immediate. This difference in immediate handling resulted in RISC-V design incorporating a new immediate handling unit called the immeblock, it decides if the immediate is a register file output or a constant.

3) *Addition of Rounding mode support in RISC-V*

In MIPS, you cannot encode a rounding mode directly in the FP instruction. But in RISC-V, you can directly encode a rounding mode. See Figure 6 below of a round to zero declaration.

```
 # Floating-point addition: f3 = f1 + f2
 fadd.s f1, f2, f3, rtz    # Round toward zero
```

Figure 6: Round to zero in RISC-V

To implement this, the rounding mode is decoded and sent to the FPU unit.

```
 wire [1:0] rm = inst[14]? 2'b00: inst[13:12];
```

Figure 7: Rounding mode extraction

4) *Difference in data forwarding between our RISC-V design and MIPS design.*

The FP store data path for both MIPS and RISC-V have one difference that caused bugs in the initial development process. The difference is the immediate microarchitecture. In RISC-V, immediates are used across multiple instruction formats (I, S, B, U, J types). These are then centralized into an immeblock, see instantiation below.

```
 imme  immeblock(inst,alui,imme);
```

Figure 8: Immidiate Block in Pipelined CPU Datapath

The output of the immeblock goes into a dedicated mux "s_ime_qb" see Figure 9, instantiation below.

```
mux2x32 s_ime_qb (b,imme,bimm,db);
```

Figure 9: ALU B input that selects between regfile or immediate

In MIPS, the store_f mux selects between ALU B output, and FP regfile output dfb. Since we are re-using the FPU block in Yamin Li's MIPS design [3], In RISC-V we should also select the ALU B output. Verilog code for the RISC-V FPU modified Store_f to take in b instead of db.

```
mux4x32 alu_a (qa,ealu,malu,mmo,fwda,da);
mux4x32 alu_b (qb,ealu,malu,mmo,fwdb,db);
mux2x32 store_f (db,dfb,swfp,dc);
mux2x32 fwd_f_d (dc,e3d,fwdf,dd);
mux2x32 fwd_f_e (ed,e3d,efwdfe,eb);
```

Figure 10: MIPS store FP and ALU fwd Mux

```
mux4x32 s_a (qa,eal,mal,mm,fwda,da);
mux4x32 s_b (qb,eal,mal,mm,fwdb,b);
mux2x32 s_ime_qb (b,imme,bimm,db);
mux2x32 store_f (b,dfb,swfp,dc);   // F
mux2x32 fwd_f_d (dc,e3d,fwdf,dd);   //F
mux2x32 fwd_f_e (ed,e3d,efwdfe,edata);
```

Figure 11: RISC-V store FP and ALU fwd Mux

D. FPU Unit internal functionality

The FPU Unit is based on the design by Yamin Li [3], which provides pipelined add/sub, multiply, divide and square-root units with latencies of N cycles. We treat the FPU as a black-box functional block and integerate it into our pipeline. For detailed microarchitecture, refer to chapter 9 of Yamin Li's textbook [3]. Here is a high level explanation of how Yamin Li's floating point unit work.

1) Fadd.s implementation

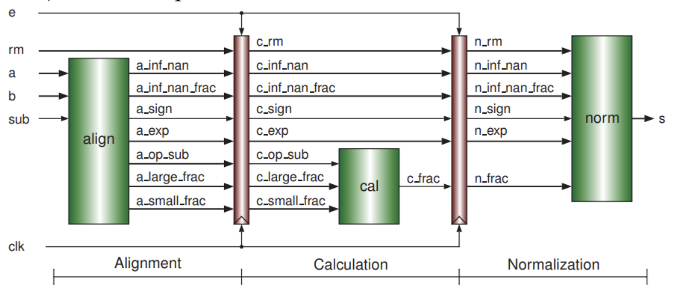


Figure 12: Block Diagram of the pipelined floating-point adder

The operation is divided into three stage, the pipelined registers are inserted in between the stages, alignment, calculation, and normalization. and have a write enable signal "e" which can stall the pipeline. This pipelined float adder/subtracter can perform a float addition or subtraction once every clock cycle. The prefix letter "a" (alignment) , "c" (calculation), and "n" (normalization) are prefixed to signals in the alignment, calculation, and normalization stages, respectively. Yamin Li's pipelined adder is instantiated in the FPU program seen in Figure 13. The only addition we made to Yamin Li's version is to connect rm to our decoded signal, see III-C-2 for rounding mode details.

```
pipelined_fadder f_add  (efa,efb,sub,rm,s_add,clk,clrn,e);
```

Figure 13: instantiation of fadder in FPU.v

2) Floating-Point Multiplier (FMUL) Design

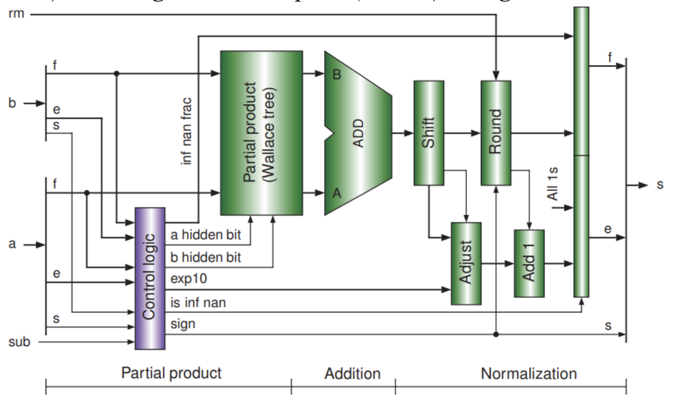


Figure 14: Block diagram of the floating-point multiplier

The floating-point multiplier uses Wallace tree for the partial product stage (sum and carry), an adder for generating the product by adding the sum and carry, and the circuits for normalization. The Wallace tree in the figure is a 24-bit Wallace tree. For detailed microarchitecture, refer to chapter 9 of [3].

3) Floating-point Divide (FDIV) Design

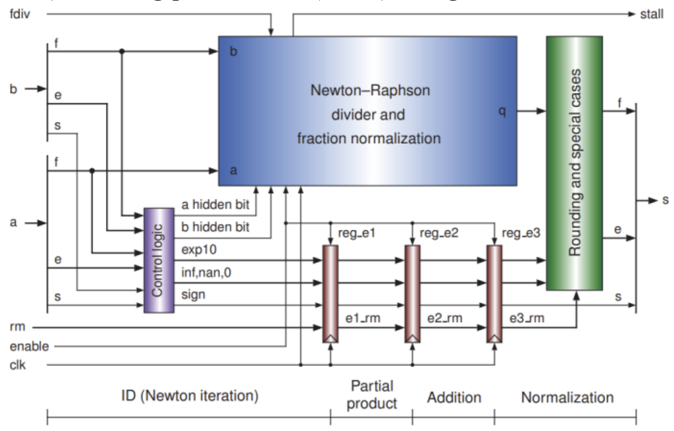


Figure 15: Block diagram of floating-point divider

The floating-point divider uses Newton-Raphson algorithm to perform the significand division. Iterative calculation and quotient calculation happens in the first stage (Netwon Iteration). Multiplication happens in the second and third stage which is implemented with a pipelined Wallace tree and a third stage addition. Normalization is the fourth stage. In Figure 16 below you can see a high level view of the pipeline stages.

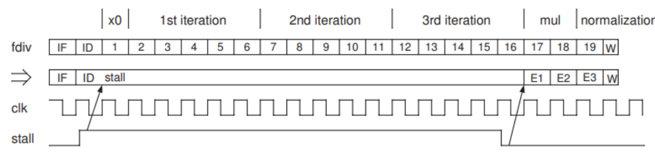


Figure 16: Pipeline stages of the floating-point divider

In Figure 16, you can see the stall of fdiv happening following Figure 15's artwork. For a more detailed microarchitecture explanation, refer to chapter 9 of [3].

4) *Floating-point Square Root (FSqrt) Design*

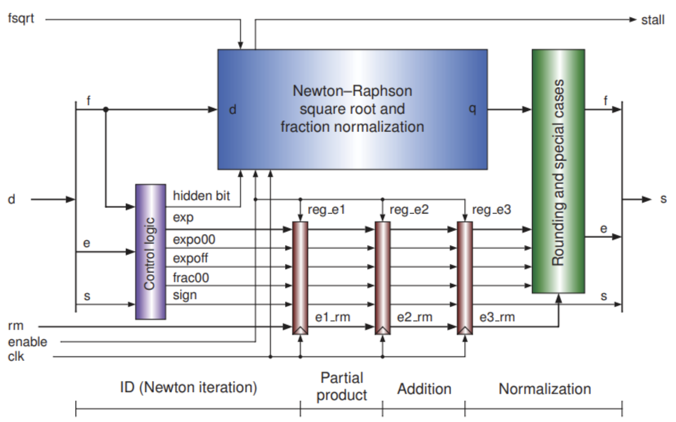
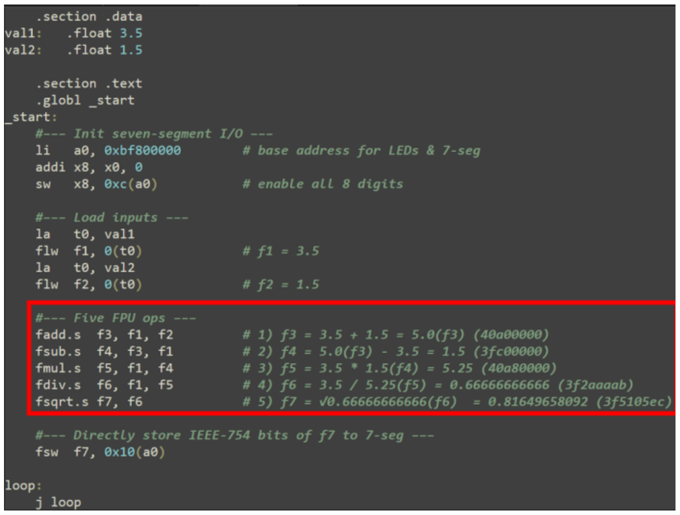


Figure 17: Block diagram of the floating-point square root

The floating-point square root is calculated using Newton-Raphson square root algorithm. The sqrt design is similar to the floating-point divide but with only one input radicand. The first stage performs iterative calculations, the second stage performs multiplication, the third stage performs addition of the second stage carry and sum, and the fourth stage perofrms normalization which generates the final result in the IEEE float format. For a more detailed microarchitecture explanation, refer to chapter 9 of [3].

E. *Functional Verification for FPU Unit*



```
    .section .data
val1:    .float 3.5
val2:    .float 1.5

    .section .text
    .globl _start
_start:
    #--- Init seven-segment I/O ---
    li   a0, 0xbf800000        # base address for LEDs & 7-seg
    addi x8, x0, 0
    sw   x8, 0xc(a0)           # enable all 8 digits

    #--- Load inputs ---
    la   t0, val1
    flw  f1, 0(t0)             # f1 = 3.5
    la   t0, val2
    flw  f2, 0(t0)             # f2 = 1.5

    #--- Five FPU ops ---
    fadd.s  f3, f1, f2         # 1) f3 = 3.5 + 1.5 = 5.0(f3) (40a00000)
    fsub.s  f4, f3, f1         # 2) f4 = 5.0(f3) - 3.5 = 1.5 (3fc00000)
    fmul.s  f5, f1, f4         # 3) f5 = 3.5 * 1.5(f4) = 5.25 (40a80000)
    fdiv.s  f6, f1, f5         # 4) f6 = 3.5 / 5.25(f5) = 0.66666666666 (3f2aaaab)
    fsqrt.s f7, f6             # 5) f7 = √0.66666666666(f6)  = 0.81649658092 (3f5105ec)

    #--- Directly store IEEE-754 bits of f7 to 7-seg ---
    fsw  f7, 0x10(a0)

loop:
    j loop
```

Figure 18: Test program showing RAW Floating-point operations

The test program loads in 3.5 to floating point register 1 (f1) and 1.5 to floating-point register 2 (f2). Demonstrating FLW functionality. Also it demonstrates fadd.s, fsub.s fmul.s fdiv.s and fsqrt.s by performing back to back RAW(Read after Write) operations. Fadd's adds 3.5 to 1.5 which is correctly 40a00000 (5.0). The result 5.0 is taken as a operand for the fsub.s instruction. The RV32IMF Core succesfully handles the RAW hazard and forwards it to the next fusb.s instruction a operand. This is shown as the fsub result being correctly 3fc00000 (5.0 - 3.5 = 1.5). For the next instruction fmul, the same RAW behavior happens and fmul.s correctly outputs 40a80000 (3.5*1.5 = 5.25). The fdiv correctly outputs 3f2aaaab (3.5/5.25 = 0.66666666). The fsqrt correctly outputs 3f5105ec (sqrt(0.66666666) = 0.81649658092). The FSW test is performed by writing the result of the sqrt operation into the seven seg memory mapped register 0x10(a0). FSW correctly stores the FPU result into the Seven seg display. See Figure 20.

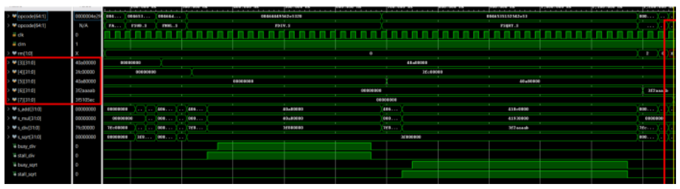


Figure 19: Screenshot of test program waveform without rounding mode modifier.

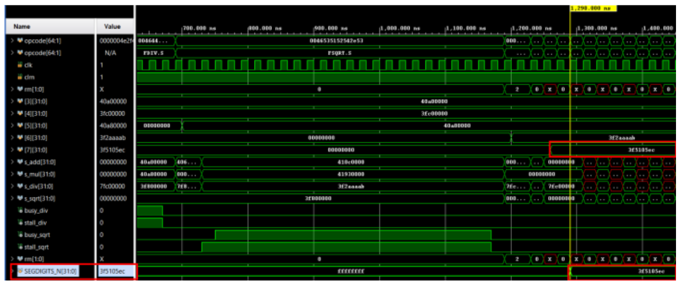


Figure 20: Screenshot of Floating-point store to seven Segment display

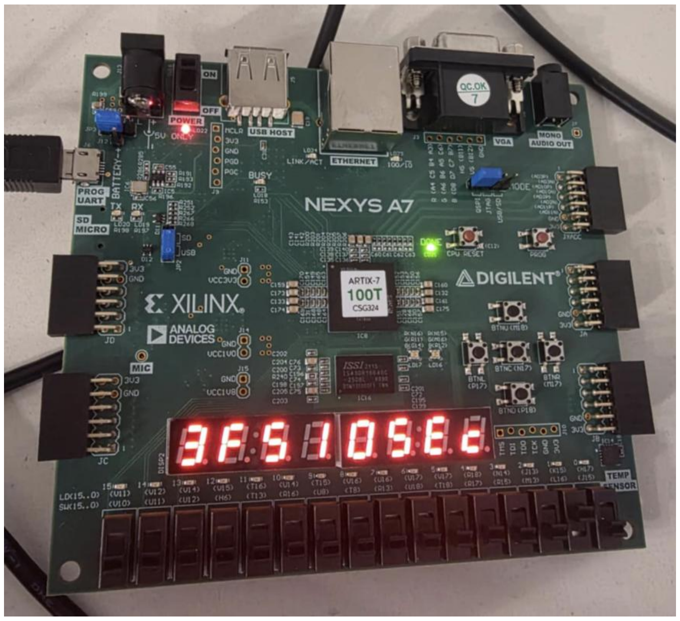


Figure 21: FPGA verification on Nexys A7 FPGA

FPGA verification was done by generating bitstream and running the program. The result correctly shows the floating-point operations functioning on an FPGA. This shows that the RTL design is functionally correct. Note edge case testing is not done as it is not in the scope of this independent study. Figure 21 shows the fsqrt result printed on the seven segment display.

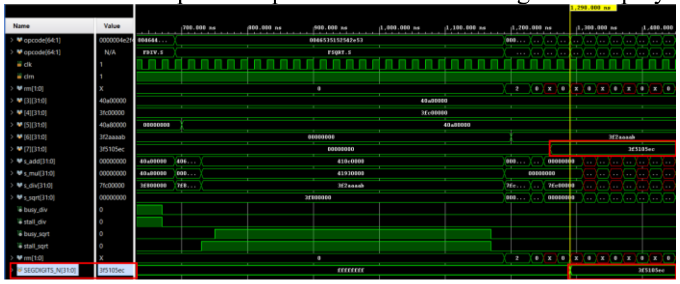
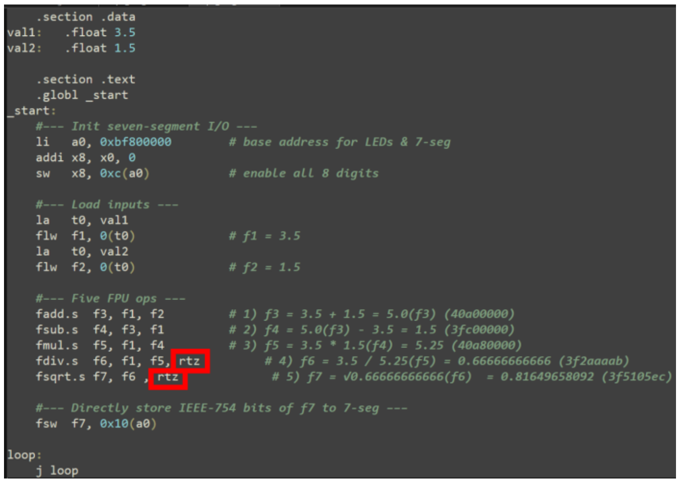


Figure 22: Screenshot of FPU result written to seven segment display

Rounding mode is tested by adding rtz to any floating point operation in assembly. For this test, we add a rtz (round to zero) modifier to fdiv and fsqrt. The code is listed in Figure 23. In Figure 24 you can see the fdiv result is round down to zero from 3f2aaaab to 3f2aaaaa. Same for fsqrt where it rounds 3f5105ec to 3f5105eb. This shows that rounding mode is correctly rounding down to zero.



```
    .section .data
val1:    .float 3.5
val2:    .float 1.5

    .section .text
    .globl _start
_start:
    #--- Init seven-segment I/O ---
    li   a0, 0xbf800000        # base address for LEDs & 7-seg
    addi x8, x0, 0
    sw   x8, 0xc(a0)           # enable all 8 digits

    #--- Load inputs ---
    la   t0, val1
    flw  f1, 0(t0)             # f1 = 3.5
    la   t0, val2
    flw  f2, 0(t0)             # f2 = 1.5

    #--- Five FPU ops ---
    fadd.s  f3, f1, f2         # 1) f3 = 3.5 + 1.5 = 5.0(f3) (40a00000)
    fsub.s  f4, f3, f1         # 2) f4 = 5.0(f3) - 3.5 = 1.5 (3fc00000)
    fmul.s  f5, f1, f4         # 3) f5 = 3.5 * 1.5(f4) = 5.25 (40a80000)
    fdiv.s  f6, f1, f5, rtz         # 4) f6 = 3.5 / 5.25(f5) = 0.66666666666 (3f2aaaab)
    fsqrt.s f7, f6 , rtz            # 5) f7 = √0.66666666666(f6)  = 0.81649658092 (3f5105ec)

    #--- Directly store IEEE-754 bits of f7 to 7-seg ---
    fsw  f7, 0x10(a0)

loop:
    j loop
```

Figure 23: Rounding Mode Test Code

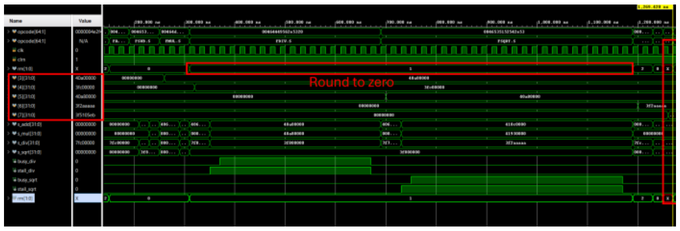


Figure 24: Rounding Mode Test Results

IV. Single-Cycle Trap/CSR/Interrupt Subsystem

A. Initial State and Plan

Yamin Li's MIPS Single Cycle Interrupt cannot be ported to RISC-V due to RISC-V's CSR-based mechanism. RISC-V has a unified CSR file that allows reading and writing to a list of privileged registers via a 12-bit address known as the CSR Address field. In MIPS, interrupt registers are special-purpose Coprocessor registers like Status, Cause and/or EPC, whereas in RISC-V, interrupt register are stored in a CSR file. Accessing the CSR file depends on the CSR address encoded in the instruction access assembly code. MIPS does not have a general CSRRS/CSRRC/CSRRW mechanism. Access to the MIPS registers are the MFC0/MTC0 instruction instead of the CSRRS and CSRRW. As a result, a CSR design that follows Yamin Li's MIPS design but matches RISC-V's CSR file style was created. Here is the plan outline for the Interrupt design:

1) *Define CSR Input Output ports*
2) *Priority-ordered sequential side effects (trap -> mret -> CSR ops -> pending)*
3) *Combinatorial read mux + output generation*
4) *Instantiating CSR Unit in sccpu_intr.v level*
5) *Trap and return sequence*
6) *Functional Testing*

B. Define CSR Input Output Ports

The CSR unit defines the machine-level registers and establishes the required Input Output interfaces. The primary

CSRs involved are mstatus, mie, mip, mtvec, mepc and mcause. These together provides the minimal machine-mode environment required by the RISC-V spec. The module accepts input from the decode logic (in control unit) via csr_en, csr_cmd, csr_addr, csr_wdata and produces the corresponding read value via csr_rdata. The module also interfaces with the trap and break mechanism through trap_set, trap_cause, trap_pc, and mret. External interrupt signaling is managed by intr and cu_intr_ack. To integrate and execute interrupt service routines, outputs such as mstatus_out, mepc_out and trap vector (mtvec) are exposed as outputs. You can see the output ports in Figure 11 on the right-hand side of the CSR Unit and the input ports on the left-hand side of the CSR Unit.

Output ports:

- mstatus: on trap_set, CSR unit transfers MIE -> MIP and clear MIE (Machine Interrupt Enable). This is done so only one type of trap can happen at a time. On MRET, we restore the MIP.
- mepc: captures the pc at the time the cycle in which interrupt happen trap set.
- mcause: records the interrupt/exception code. Informs the csr what type of interrupt happened.
- mtvec: tells the PC which address to jump to to reach the ISR jump routine.
- csr_rdata: passes data to regfile to write to register
- mie: machine interrupt enable register
- mip: machine interrupt pending register

Input ports:

- trap_set: trap set is the enable bit for trap handling
- trap_cause: is the value to write to the mcause register
- mret: comes from the control unit, indicates an mret instruction is decoded
- intr_synced: is the synchronized external interrupt signal
- cu_intr_ack: informs the CSR unit that external interrupt is executed. CSR will then clear MIP so external interrupt does not repeat.
- Csr_wdata: is the data that is written to csr_addr. Usually the value stored in the register in csrrs or csrrw assembly instruction.
- Csr_en: csr instruction decoded flag, it acts as an enable for CSR register operation. Ensures CSR writes happen only during CSR instructions
- Trap_pc: trap pc gets the current program counter. The CSR Unit will latch the current program counter when an exception happens.
- Csr_addr: csr address is a 12-bit field data that determines which CSR register the a csrrs, csrrw or other csrr instructions in RISC-V ISA is targeting.
- Csr_cmd: is a 3 bit field data that informs the CSR Unit which csr instruction is decoded, 001=CSRRW, 010=CSRRS, 011=CSRRC.

Figure 25 shows a high level on Input Output port interactions.

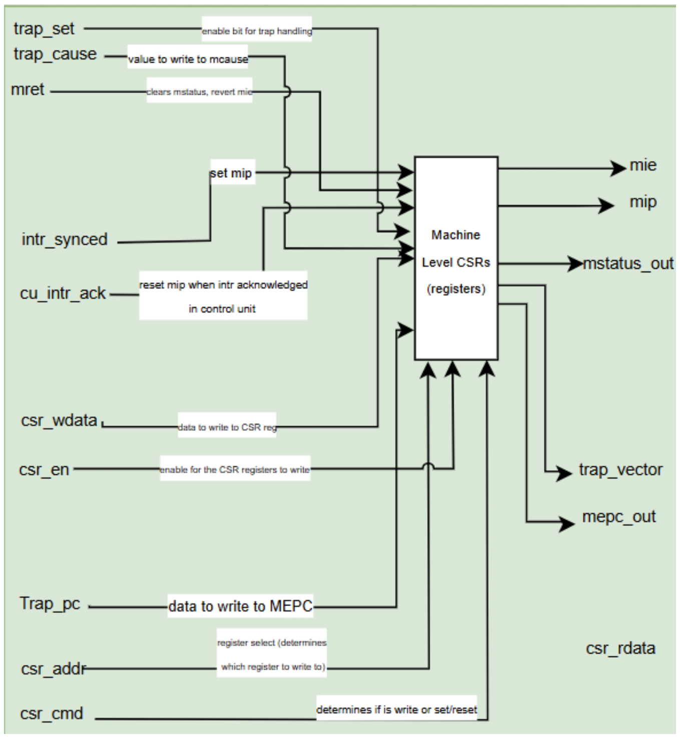

trap_set
enable bit for trap handling
trap_cause
value to write to mcause
mret
clears mstatus, revert mie
set mip
intr_synced
cu_intr_ack
reset mip when intr acknowledged in control unit
Machine Level CSRs (registers)
mie
mip
mstatus_out
trap_vector
mepc_out
csr_wdata
data to write to CSR reg
csr_en
enable for the CSR registers to write
Trap_pc
data to write to MEPC
csr_addr
register select (determines which register to write to)
csr_cmd
determines if is write or set/reset
csr_rdata

Figure 25: CSR Unit Internals

C. *Priority-ordered sequential side effects (trap -> mret -> CSR ops -> pending)*

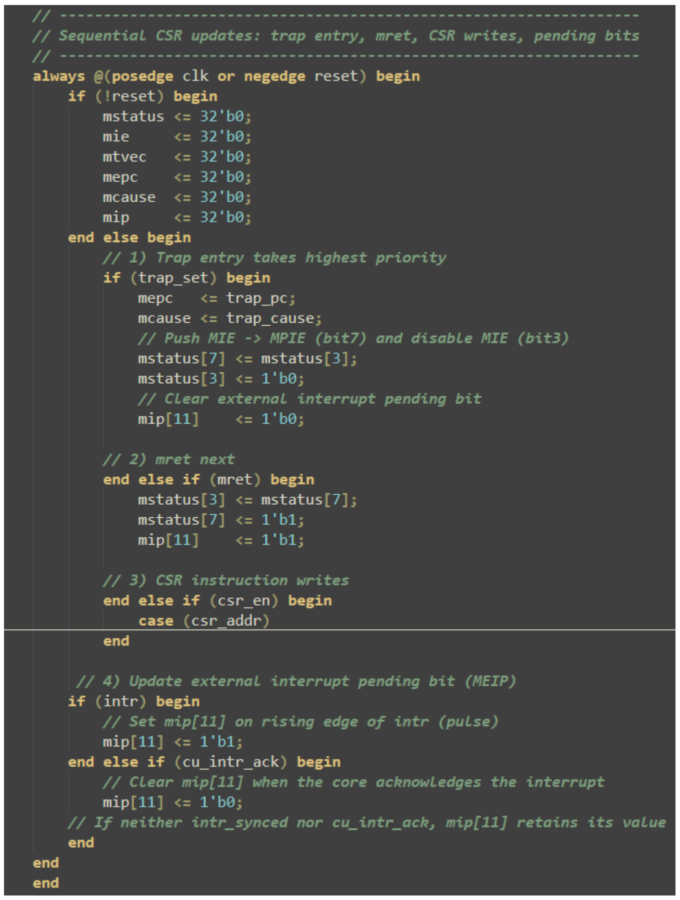
The bulk of the CSR Unit logic is in a priority ordered CSR register update logic seen in Figure 26. The 4 priorities in order of priority are, trap_set, mret, CSR instructions, MIP pending logic. Trap set is the highest priority while MIP pending is the lowest priority.

When Trap_set is asserted, the hardware saves the current PC into mepc, records the cause in mcause, pushes MIE into MPIE, clears MIE to disable further interrupts.

When Mret is decoded in the control unit, mret logic restores the interrupt enable state by copying MPIE back into MIE and setting MPIE to one.

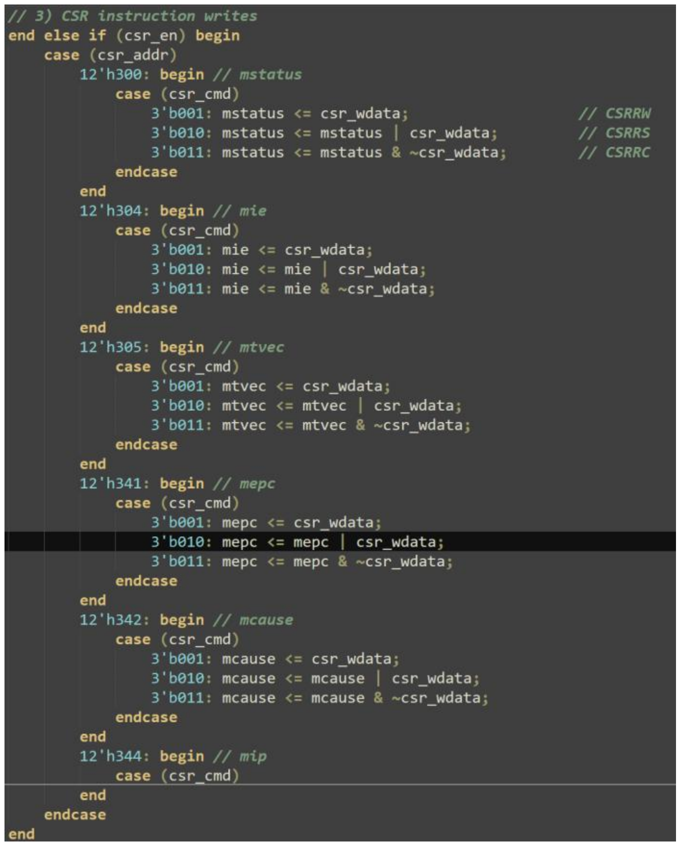
CSR instructions take effect if no trap or mreturn is in progress. CSR instructions update the target register with masking to ensure only legal bits change. CSR determines which register to update based off the CSR_addr data. See Figure 28 for the case statement code that implement this behavior. There is also a csr_en bit that gates the CSR instruction logic to ensure writing to CSR registers only happen during a CSR instruction.

Finally, at the lowest priority the pending bits logic. When an external interrupt assertion sets MIP and an acknowledgement from the control unit clears it.



```
// -----------------------------------------------------------------
// Sequential CSR updates: trap entry, mret, CSR writes, pending bits
// -----------------------------------------------------------------
always @(posedge clk or negedge reset) begin
    if (!reset) begin
        mstatus <= 32'b0;
        mie     <= 32'b0;
        mtvec   <= 32'b0;
        mepc    <= 32'b0;
        mcause  <= 32'b0;
        mip     <= 32'b0;
    end else begin
        // 1) Trap entry takes highest priority
        if (trap_set) begin
            mepc   <= trap_pc;
            mcause <= trap_cause;
            // Push MIE -> MPIE (bit7) and disable MIE (bit3)
            mstatus[7] <= mstatus[3];
            mstatus[3] <= 1'b0;
            // Clear external interrupt pending bit
            mip[11]    <= 1'b0;

        // 2) mret next
        end else if (mret) begin
            mstatus[3] <= mstatus[7];
            mstatus[7] <= 1'b1;
            mip[11]    <= 1'b1;

        // 3) CSR instruction writes
        end else if (csr_en) begin
            case (csr_addr)
        end

     // 4) Update external interrupt pending bit (MEIP)
    if (intr) begin
        // Set mip[11] on rising edge of intr (pulse)
        mip[11] <= 1'b1;
    end else if (cu_intr_ack) begin
        // Clear mip[11] when the core acknowledges the interrupt
        mip[11] <= 1'b0;
    // If neither intr_synced nor cu_intr_ack, mip[11] retains its value
    end
end
end
```

Figure 26: Verilog code for CSR Priority sequential update

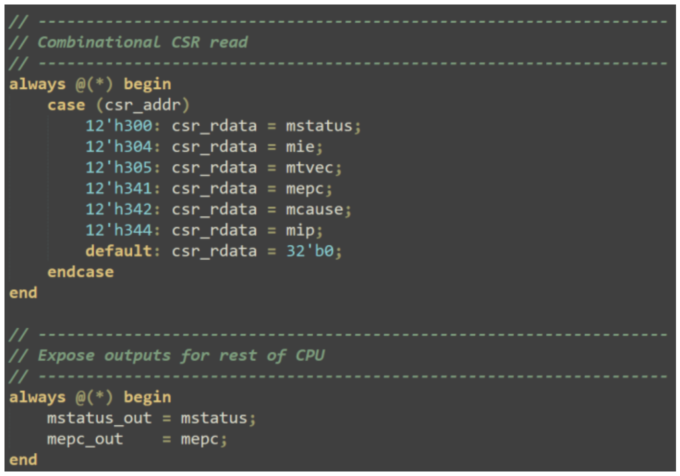


```
// 3) CSR instruction writes
end else if (csr_en) begin
    case (csr_addr)
        12'h300: begin // mstatus
            case (csr_cmd)
                3'b001: mstatus <= csr_wdata;                // CSRRW
                3'b010: mstatus <= mstatus | csr_wdata;      // CSRRS
                3'b011: mstatus <= mstatus & ~csr_wdata;     // CSRRC
            endcase
        end
        12'h304: begin // mie
            case (csr_cmd)
                3'b001: mie <= csr_wdata;
                3'b010: mie <= mie | csr_wdata;
                3'b011: mie <= mie & ~csr_wdata;
            endcase
        end
        12'h305: begin // mtvec
            case (csr_cmd)
                3'b001: mtvec <= csr_wdata;
                3'b010: mtvec <= mtvec | csr_wdata;
                3'b011: mtvec <= mtvec & ~csr_wdata;
            endcase
        end
        12'h341: begin // mepc
            case (csr_cmd)
                3'b001: mepc <= csr_wdata;
                3'b010: mepc <= mepc | csr_wdata;
                3'b011: mepc <= mepc & ~csr_wdata;
            endcase
        end
        12'h342: begin // mcause
            case (csr_cmd)
                3'b001: mcause <= csr_wdata;
                3'b010: mcause <= mcause | csr_wdata;
                3'b011: mcause <= mcause & ~csr_wdata;
            endcase
        end
        12'h344: begin // mip
            case (csr_cmd)
        end
    endcase
end
```

Figure 27: CSR Write Instruction case statement

D. Combinatorial read mux + output generation

The final ingredient to the CSR unit is the combinatorial read mux and output exposing logic. A simple combinatorial multiplexer drives csr_rdata based on the requested address (csr_addr). This allows CSR reads to complete in a single cycle. Additionally this simple logic allows the addition of other supported register in the future. Currently only mstatus and mepc are exposed as outputs since these two are needed in the control unit and "nxtpc" program counter mux.



```
// -----------------------------------------------------------------
// Combinational CSR read
// -----------------------------------------------------------------
always @(*) begin
    case (csr_addr)
        12'h300: csr_rdata = mstatus;
        12'h304: csr_rdata = mie;
        12'h305: csr_rdata = mtvec;
        12'h341: csr_rdata = mepc;
        12'h342: csr_rdata = mcause;
        12'h344: csr_rdata = mip;
        default: csr_rdata = 32'b0;
    endcase
end

// -----------------------------------------------------------------
// Expose outputs for rest of CPU
// -----------------------------------------------------------------
always @(*) begin
    mstatus_out = mstatus;
    mepc_out    = mepc;
end
```

Figure 28: Combinatorial read mux and Output expose logic

E. *Instantiating CSR-Unit in sccpu_intr.v level*

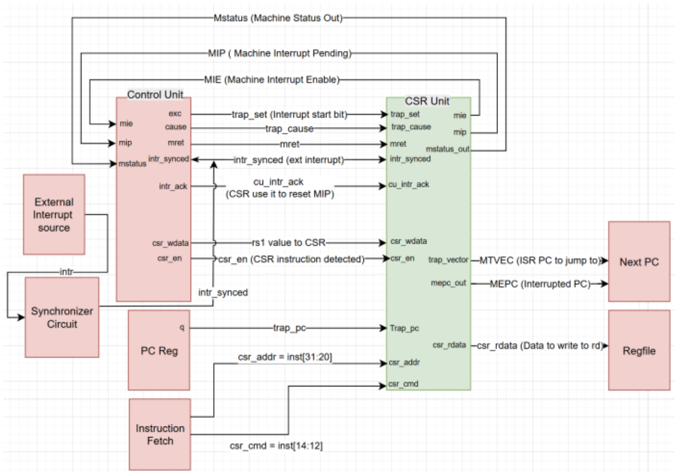


Figure 29: CSR Unit port interactions in Single-cycle with Interrupt

The CSR Unit is instantiated following the block diagram in Figure 29. Control unit provides trap_set, trap_cause, mret, cu_intr_ack, csr_wdata, csr_en while CSR Unit feeds the control unit mie, mip and mstatus_out so that it can generate cu_intr_ack. Synchronizer circuit provides external interrupt input intr_synched. PC reg provides Trap_pc. Instruction fetch logic provides csr_addr and csr_cmd. CSR Unit provides mtvec and mepc to Next PC mux. CSR Unit also provides csr_rdata to regfile.

F. *Trap and return logic flow*

1) *Sequencing: synchronous interrupts*

Synchronous Interrupts are decoded in the control unit. The decode logic code is seen in Figure 30 below. Unimplemented instruction trap is set when the decoder does not recognize the incoming instruction. System call happens when the ecall instruction is decoded. Overflow happens when the ALU detects an overflow via the "v" wire.

```
//------------------------------------------------------------------
// 2) Synchronous exceptions (always trap when they occur)
//------------------------------------------------------------------
wire unimplemented_inst = ~(i_csrrw | i_csrrs | i_mret | i_ecall | i_slt|
       i_add | i_sub | i_and | i_or | i_xor | i_slli| i_srli| i_srai|
       i_jalr | i_addi | i_andi | i_ori | i_xori | i_lw | i_sw | i_beq |
       i_bne| i_lui| i_jal);
wire i_ecall = (opcode == 7'b1110011) && (func3 == 3'b000) && (csr_addr == 12'h000);
wire overflow = v & (i_add | i_sub | i_addi);         // overflow

wire exc_sys  = i_ecall;              // ECALL
wire exc_uni  = unimplemented_inst; // Illegal opcode
wire exc_ovr  = overflow;             // Your custom overflow trap
assign exc   = int_int | exc_sys | exc_uni | exc_ovr; // exc or int occurs
```

Figure 30: Decode logic for Synchronous exception (Verilog)

Once any of the synchronous exception is detected, the exception detected wire ("exc" in Figure 30) will inform the CSR unit that trap is set. This can be seen in Figure 31. The "exc" wire is an output port on the right-hand side of the Control Unit block.

2) *Sequencing: asynchronous interrupts*

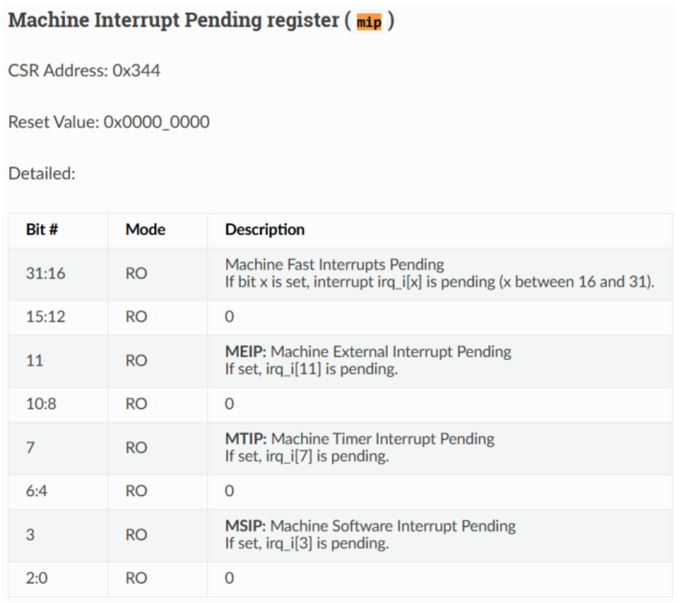
Our single-cycle RV32IM interrupt has one Asynchronous Interrupt. Since it is asynchronous, a two stage flip flop is used to synchronize the input , seen in Figure 29. The sampled and stable external interrupt signal intr_synced is fed into the CSR unit and control unit. The CSR unit will use it to set MIP. Using MIP and MIE set by the CSR, the control unit will check if any synchronous interrupts are executing. If none are executing, it will set "exc". When CSR Unit receives the trap_set, it will clear the MIP, ensuring external interrupt do not repeat over multiple cycles (MIP resetting causes one cycle pulse for external interrupt).

3) *Trap Set logic in CSR*

When trap is set, the CSR modifies mepc, mcause, mstatus, and mip following the Verilog code shown in Figure 31. Current program counter is written into mepc, trap cause is written into mcause, Mip[11], the 11th bit of MIP is set/reset because it is determined in RISC-V ISA, see figure 32. Mip[11] is cleared to disable external interrupt while trap is already set. Once mret triggers, all the changes made during trap_set are resetted to default.

```
if (trap_set) begin
    mepc   <= trap_pc;
    mcause <= trap_cause;
    // Push MIE -> MPIE (bit7) and disable MIE (bit3)
    mstatus[7] <= mstatus[3];
    mstatus[3] <= 1'b0;
    // Clear external interrupt pending bit
    mip[11]    <= 1'b0;

// 2) mret next
end else if (mret) begin
    mstatus[3] <= mstatus[7];
    mstatus[7] <= 1'b1;
```

Figure 31: Trap_set logic


Machine Interrupt Pending register (mip)

CSR Address: 0x344

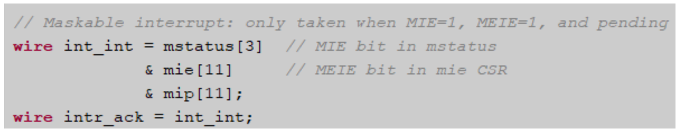
Reset Value: 0x0000_0000

Detailed:

Bit #	Mode	Description
31:16	RO	Machine Fast Interrupts Pending If bit x is set, interrupt irq_i[x] is pending (x between 16 and 31).
15:12	RO	0
11	RO	**MEIP:** Machine External Interrupt Pending If set, irq_i[11] is pending.
10:8	RO	0
7	RO	**MTIP:** Machine Timer Interrupt Pending If set, irq_i[7] is pending.
6:4	RO	0
3	RO	**MSIP:** Machine Software Interrupt Pending If set, irq_i[3] is pending.
2:0	RO	0

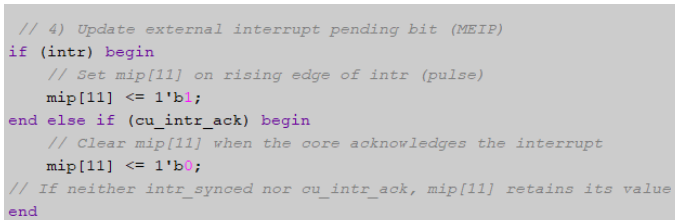
Figure 32: MIP register mapping

4) *External/timer/software interrupt pending logic*



```
// Maskable interrupt: only taken when MIE=1, MEIE=1, and pending
wire int_int = mstatus[3]  // MIE bit in mstatus
             & mie[11]     // MEIE bit in mie CSR
             & mip[11];
wire intr_ack = int_int;
```

Figure 33: Control Unit External interrupt polling



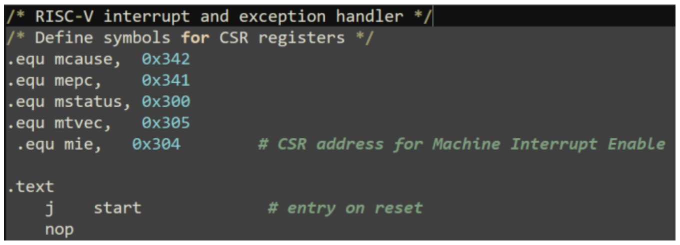
```
 // 4) Update external interrupt pending bit (MEIP)
if (intr) begin
    // Set mip[11] on rising edge of intr (pulse)
    mip[11] <= 1'b1;
end else if (cu_intr_ack) begin
    // Clear mip[11] when the core acknowledges the interrupt
    mip[11] <= 1'b0;
// If neither intr_synced nor cu_intr_ack, mip[11] retains its value
end
```

Figure 34: CSR Unit MIP set and reset logic

On an asynchronous interrupt trap_set, Figure 33 and 34 logic is executed. Figure 15's logic will provide the asynchronous interrupt the ability to poll until interrupt can be executed. This is required since asynchronous interrupt are lower priority than synchronous and thus will have to wait for synchronous interrupts to return before executing. Once the synchronous interrupt calls mret, and mstatus[3] bit is set back to 1, the external interrupt triggers. Control unit sends cu_intr_ack to the CSR Unit and CSR Unit resets the mip bit, preventing the same external interrupt to trigger a trap.

G. Functional Verification

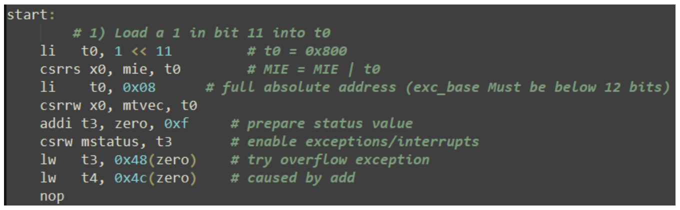
To test the functionality of the single cycle interrupts, we used a test program that cycles through the 4 different types of exceptions. The program first prepares the CPU for interrupts by declaring the csr address types and jumping to start



```
/* RISC-V interrupt and exception handler */
/* Define symbols for CSR registers */
.equ mcause,  0x342
.equ mepc,    0x341
.equ mstatus, 0x300
.equ mtvec,   0x305
 .equ mie,    0x304        # CSR address for Machine Interrupt Enable

.text
    j     start            # entry on reset
    nop
```

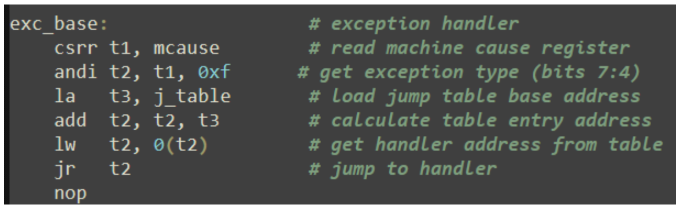
Figure 35: Declare CSR address commands

The start section uses csrrs to enable interrupt via the csrrs x0, mie, t0 line. This line will enable overflow and unimplemented instruction. It also writes 0x08 to mtvec which is the address of the interrupt service routine vector table processing seen in Figure 37. Then it prepares mstatus by writing 0xf. This sets the mie[3] to 1 which enables external interrupt. Next it loads 0x48 and 0x4c to t3 and t4 to prepare for an overflow exception.



```
start:
        # 1) Load a 1 in bit 11 into t0
    li   t0, 1 << 11         # t0 = 0x800
    csrrs x0, mie, t0        # MIE = MIE | t0
    li    t0, 0x08      # full absolute address (exc_base Must be below 12 bits)
    csrrw x0, mtvec, t0
    addi t3, zero, 0xf     # prepare status value
    csrw mstatus, t3       # enable exceptions/interrupts
    lw   t3, 0x48(zero)    # try overflow exception
    lw   t4, 0x4c(zero)    # caused by add
    nop
```

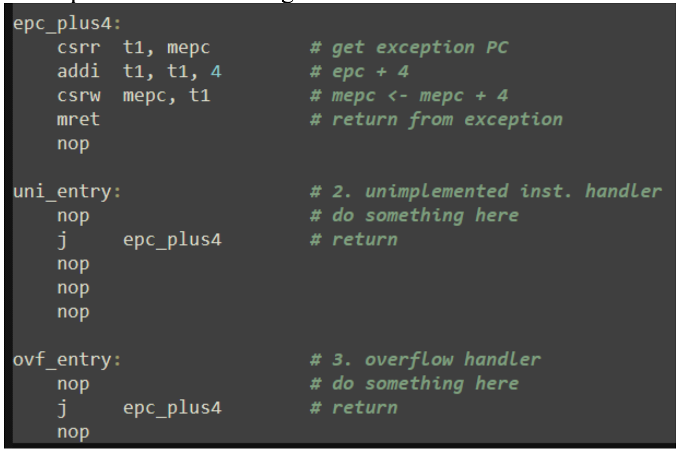
Figure 36: Initializing CSR enable bits



```
exc_base:                    # exception handler
    csrr t1, mcause          # read machine cause register
    andi t2, t1, 0xf        # get exception type (bits 7:4)
    la   t3, j_table         # load jump table base address
    add  t2, t2, t3          # calculate table entry address
    lw   t2, 0(t2)           # get handler address from table
    jr   t2                  # jump to handler
    nop
```

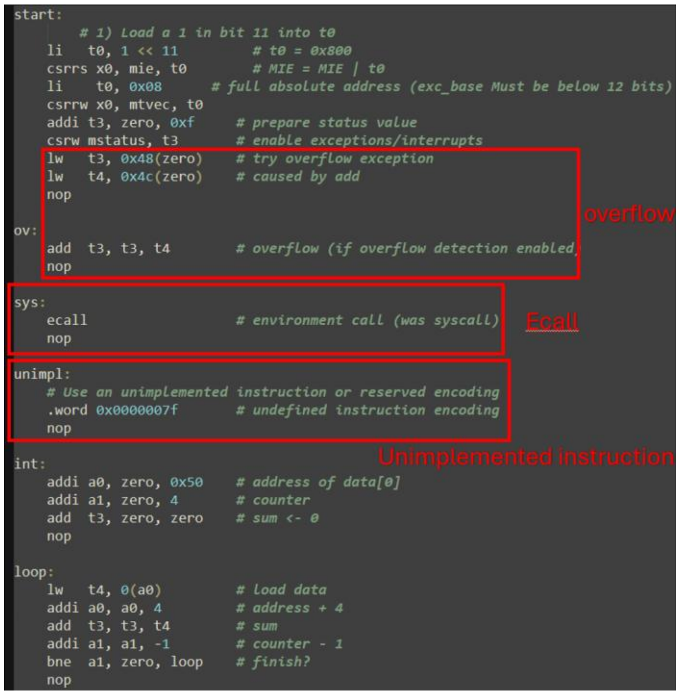
Figure 37: mtvec Vector table jump logic

When t3 and t4 add together in Figure 39 overflow section, the overflow exception is triggered shown in Figure 40 simulation and Figure 41 ILA waveform. The overflow exception correctly causes the PC to jump to mtvec value of 0x08 which stores exc base. The trap_PC value is correctly +4 based on Figure 38 and the nop address of a0 in figure 44 before ecall was executed.



```
epc_plus4:
    csrr  t1, mepc           # get exception PC
    addi  t1, t1, 4          # epc + 4
    csrw  mepc, t1           # mepc <- mepc + 4
    mret                     # return from exception
    nop

uni_entry:                   # 2. unimplemented inst. handler
    nop                      # do something here
    j     epc_plus4          # return
    nop
    nop
    nop

ovf_entry:                   # 3. overflow handler
    nop                      # do something here
    j     epc_plus4          # return
    nop
```

Figure 38: epc_plus4 exception handler



```
start:
        # 1) Load a 1 in bit 11 into t0
    li   t0, 1 << 11         # t0 = 0x800
    csrrs x0, mie, t0        # MIE = MIE | t0
    li    t0, 0x08      # full absolute address (exc_base Must be below 12 bits)
    csrrw x0, mtvec, t0
    addi t3, zero, 0xf     # prepare status value
    csrw mstatus, t3       # enable exceptions/interrupts
    lw   t3, 0x48(zero)    # try overflow exception
    lw   t4, 0x4c(zero)    # caused by add
    nop


ov:
    add  t3, t3, t4        # overflow (if overflow detection enabled)
    nop

sys:
    ecall                  # environment call (was syscall)
    nop

unimpl:
    # Use an unimplemented instruction or reserved encoding
    .word 0x0000007f       # undefined instruction encoding
    nop

int:
    addi a0, zero, 0x50    # address of data[0]
    addi a1, zero, 4       # counter
    add  t3, zero, zero    # sum <- 0
    nop

loop:
    lw   t4, 0(a0)         # load data
    addi a0, a0, 4         # address + 4
    add  t3, t3, t4        # sum
    addi a1, a1, -1        # counter - 1
    bne  a1, zero, loop    # finish?
    nop
```

Figure 39: Exception test assembly code

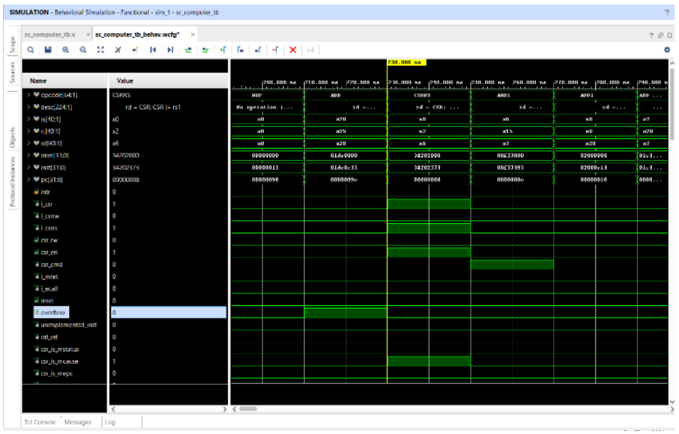


Figure 40: Overflow exception Simulation

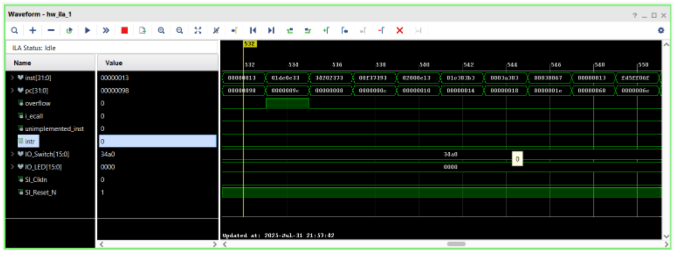


Figure 41: Overflow exception ILA

Once the overflow service routine calls mret, the pc correctly returns to the next instruction after trap_pc which in this case is the ecall instruction seen in Figure 38. Note the ecall's previous instruction is the nop which is the next instruction after overflow.

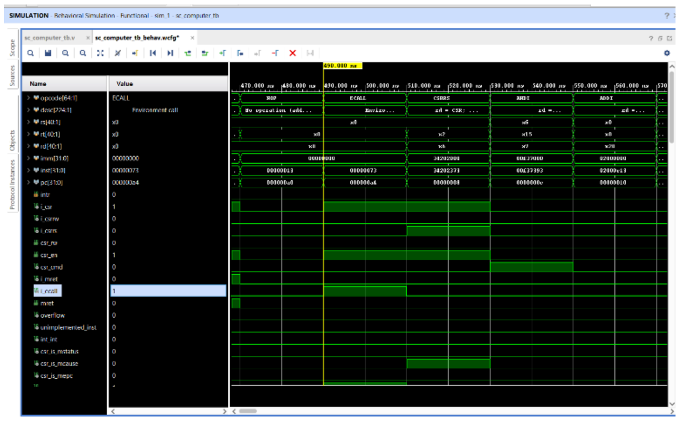


Figure 42: Ecall exception simulation

Ecall is handled similar to overflow and correctly jumps to the exc base. To verify that Ecall correctly jumps to its service routine, a sub instruction was added. In Figure 43, you can see the sub routine correctly executing.

```
sys_entry:                  # 1. syscall (ecall) handler
    li   t5, 10             # t5 ← 10
    li   t6,  3             # t6 ←  3
    sub  t3, t5, t6         # t7 ← 10-3 = 7  <-- proof ecall handler ran
    nop
```
Figure 43: ecall section on test program

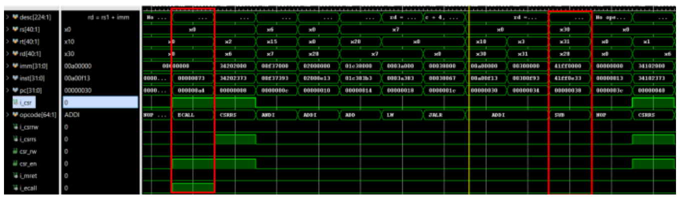


Figure 44: Ecall exception Simulation

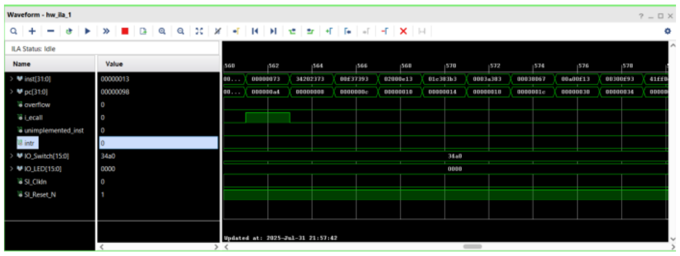


Figure 45: Ecall exception ILA

Unimplemented instruction is the next exception after the Ecall exception test. Seen in Figure 39, unimplemented instruction correctly jumps to exc base and executes epc4_plus4 code in Figure 38.

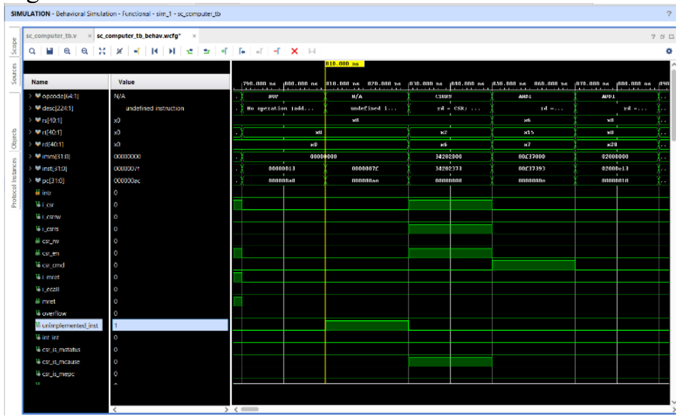


Figure 46: Unimplemented instruction exception

External interrupt exception is tested in simulation using a 50ns pulse with the testbench Verilog code, see results in Figure 47. The FPGA version is tested using a button tied to the intr input signal Figure 48, note that intr_sync our external interrupt input pulses only for 1 clock cycle. A test program with a counter increment logic in the external interrupt successfully verifies our external interrupt functionality. This counter program increments the counter in the external interrupt service routine and displays it on the seven seg display, program shown in Figure 49, result shown in Figure 50.

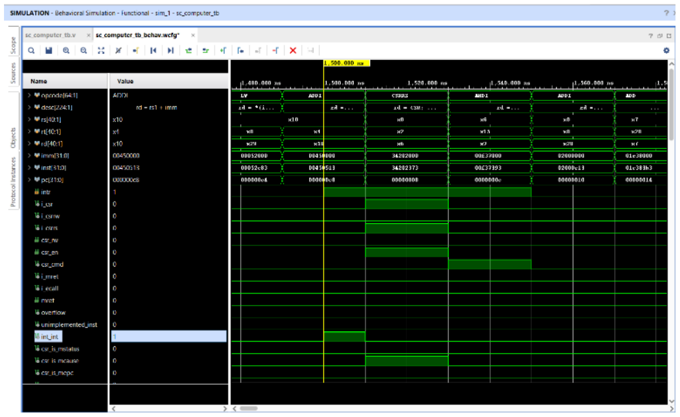


Figure 47: 50ns External Interrupt simulation

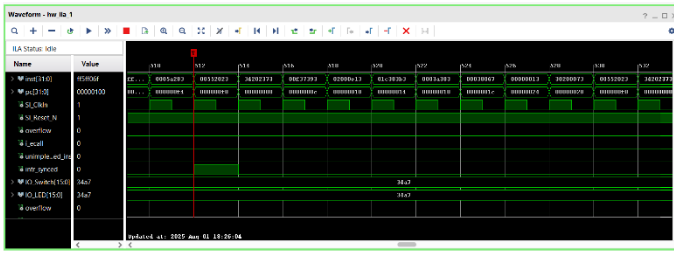
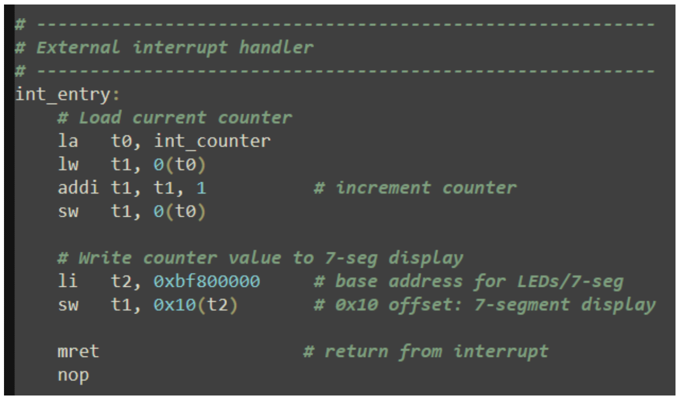


Figure 48: External Interrupt button ILA



```
# -----------------------------------------------------------
# External interrupt handler
# -----------------------------------------------------------
int_entry:
    # Load current counter
    la   t0, int_counter
    lw   t1, 0(t0)
    addi t1, t1, 1          # increment counter
    sw   t1, 0(t0)

    # Write counter value to 7-seg display
    li   t2, 0xbf800000     # base address for LEDs/7-seg
    sw   t1, 0x10(t2)       # 0x10 offset: 7-segment display

    mret                    # return from interrupt
    nop
```

Figure 49: External Interrupt Button press coutner service routine

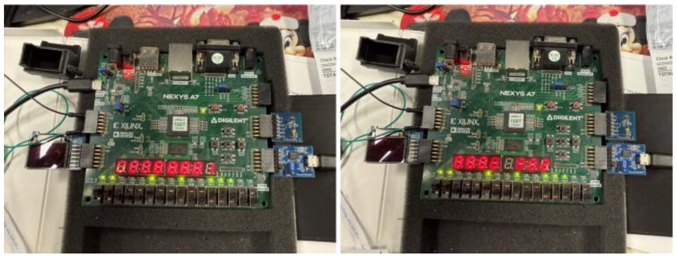


Figure 50: Pressing Button once increments count by 1

V. Pipelined Trap/CSR/Interrupt Implementation

A. *Initial State and Plan*

In a five-stage RISC-V Pipeline, the CSR Unit cannot be connected directly as in the single-cycle interrupt implementation. This is because CSR instructions must both read values in the EX stage and commit updates precisely in the WB stage. To resolve this, the design introduces a CSR pipeline adapter, a thin wrapper around the single-cycle CSR unit that enforces pipeline ordering, handles bypassing, and guarantees precise trap. The adapter ensures that CSR writes, MRET returns, and trap bookkeeping only take effect once the instruction reaches WB. At the same time, the adapter provides EX with up-to-date read data, including hazards caused by EX and WB accessing same data. It also gates trap redirects to avoid hazards, such as delaying a redirect if mtvec is written in the same cycle. The adapter addition plan will be similar to the FPU implementation since both are porting design over. Here is the implementation plan:

1) *Define CSR adapter Input and Output Port*
2) *Designing CSR*
   a) *Designing Stage decoupling and commit ordering (Only perform writes and mret at WB stage*
   b) *Designing Read bypass & data forwarding (WB -> EX stage)*
   c) *Designing Trap redirect gating (fix mtvec write race)*
3) *External interrupt handshake passthrough*
4) *Instantiating CSR_unit_pipelined_adapter in pl_computer.v*
5) *Trap entry sequencing and priority (sync vs. interrupts) — pipelin*
6) *Functional Verification and Testing*

Stage decoupling and commit ordering (Only perform writes and mret at WB stage)

- Read bypass & data forwarding (WB -> EX stage)
- Trap redirect gating (fix mtvec write race)
- External interrupt handshake passthrough

B. *Defeine CSR adapter Input and Output port*

The input and output of the CSR are defined as follows:

Output port:

- Take_trap
- Trap_vector
- csr_rdata_ex
- mstatus
- mie
- mip
- mepc
- mcause
- mtvec

Input port:

- intr_synced
- cu_intr_ack
- take_trap_r
- cause_low_sel_r
- is_intr_sel_r
- trap_pc_source
- csr_is_ex

- csr_is_cmd_ex
- csr_addr_ex
- csr_wdata_ex
- is_mret_ex
- csr_is_wb
- csr_cmd_wb
- csr_addr_wb
- csr_wdata_wb
- is_mret_wb

C. *Designing the CSR Adapter*

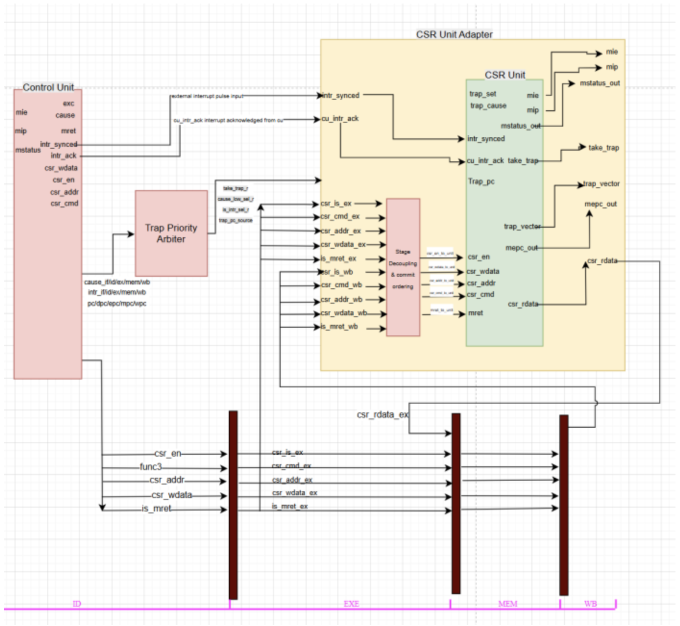
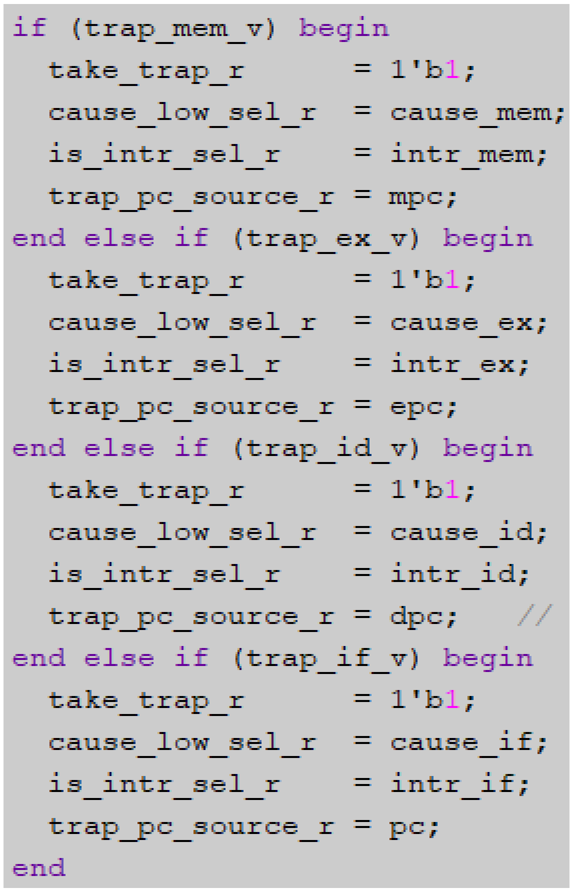


Figure 51: CSR High levell block diagram

1) *Trap Priority Arbiter*

In pipeline interrupts, exceptions can come from multiple stages. Since certain stages logically should be prioritized over another stage, a priority standard has to be determined. The chosen priority is oldest stage wins. For instance, if we have a page fault in mem stage, the page fault exception will take priority over an execution or decode stage logic. This logic can be seen in Figure 16. It works by having a generic take_trap (take_trap_r), cause register (cause_low_sel_r), intr detected (is_intr_sel_r), trap pc source (trap_pc_source_r). These generic signals are required inputs to the CSR Unit. This generic signals allow us to easily port the CSR Unit from single-cycle to pipeline. The next notable signal is the trap_mem_v, trap_ex_v, trap_id_v, trap_if_v signals. These signals are the "interrupt detected" signal from their respective stages.

The Trap Priority Arbiter takes each stages cause, trap_pc and intr signal, chooses the priority and outputs 4 signals which are take_trap_r, cause_low_sel_r, is_intr_sel_r and trap_pc_source_r. These 4 signal enters the CSR Unit Adapter. Figure 52 shows the logic for the Trap Priority Arbiter. It is a priority mux.

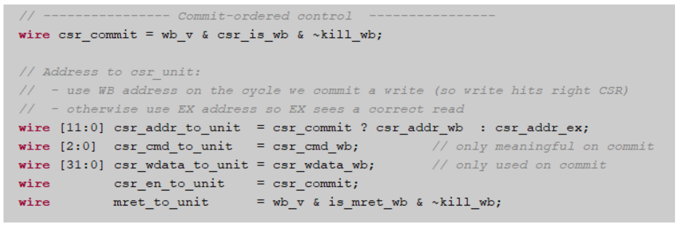


```
if (trap_mem_v) begin
  take_trap_r        = 1'b1;
  cause_low_sel_r  = cause_mem;
  is_intr_sel_r      = intr_mem;
  trap_pc_source_r = mpc;
end else if (trap_ex_v) begin
  take_trap_r        = 1'b1;
  cause_low_sel_r  = cause_ex;
  is_intr_sel_r      = intr_ex;
  trap_pc_source_r = epc;
end else if (trap_id_v) begin
  take_trap_r        = 1'b1;
  cause_low_sel_r  = cause_id;
  is_intr_sel_r      = intr_id;
  trap_pc_source_r = dpc;    //
end else if (trap_if_v) begin
  take_trap_r        = 1'b1;
  cause_low_sel_r  = cause_if;
  is_intr_sel_r      = intr_if;
  trap_pc_source_r = pc;
end
```

Figure 52: Oldest exception wins Priority logic

2) *Stage decoupling & commit ordering*

The stage decoupling & commit ordering logic can be seen in Figure 51. It takes in the is_csr, csr_cmd, csr_addr, csr_wdata and is_mret signal from both the write back and execution stage. The execution stage inputs qualifies a CSR access in EX stage. The write back stage inputs defines the actual CSR commit at WB stage. This is needed because the CSR executes interrupt at EX but writes to the registers at WB. The code is shown in Figure 53. "Csr_commit" assertion can be gated by kill_wb and wb_v. "csr_addr_to_unit" gets WB stage csr_addr when csr_commit is true, else it gets csr_addr_ex. This is the main purpose of the stage decoupler.

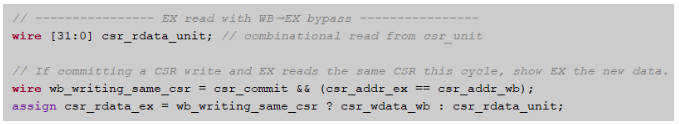


```
// ---------------- Commit-ordered control  ----------------
wire csr_commit = wb_v & csr_is_wb & ~kill_wb;

// Address to csr_unit:
//  - use WB address on the cycle we commit a write (so write hits right CSR)
//  - otherwise use EX address so EX sees a correct read
wire [11:0] csr_addr_to_unit  = csr_commit ? csr_addr_wb  : csr_addr_ex;
wire [2:0]  csr_cmd_to_unit   = csr_cmd_wb;         // only meaningful on commit
wire [31:0] csr_wdata_to_unit = csr_wdata_wb;       // only used on commit
wire        csr_en_to_unit    = csr_commit;
wire        mret_to_unit      = wb_v & is_mret_wb & ~kill_wb;
```

Figure 53: Stage decoupling & commit ordering code

3) *Read Bypass & Data Forwarding*

The adapter has hazard resolution for the scenario where WB stage is writing to the same CSR that EX stage is reading from. Data forwarding will happen, see Figure 54 for the forwarding code.

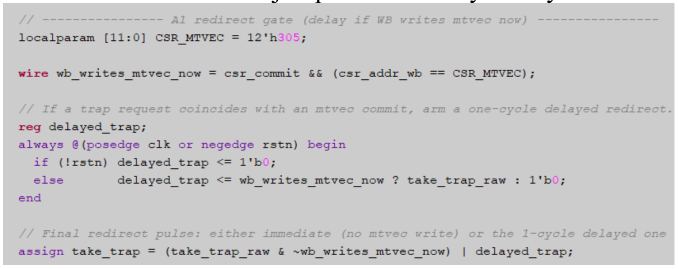


```
// ---------------- EX read with WB→EX bypass ----------------
wire [31:0] csr_rdata_unit; // combinational read from csr_unit

// If committing a CSR write and EX reads the same CSR this cycle, show EX the new data.
wire wb_writing_same_csr = csr_commit && (csr_addr_ex == csr_addr_wb);
assign csr_rdata_ex = wb_writing_same_csr ? csr_wdata_wb : csr_rdata_unit;
```

Figure 54: WB and EX write/read same CSR_addr resolution

4) Read after write hazard on MTVEC

Trap redirect gating ensures architectural precision by resolving the race between mtvec writes and trap requests. If a trap is requested in the same cycle that WB commits an mtvec update, the CPU stalls the mtvec jump instruction by one cycle.

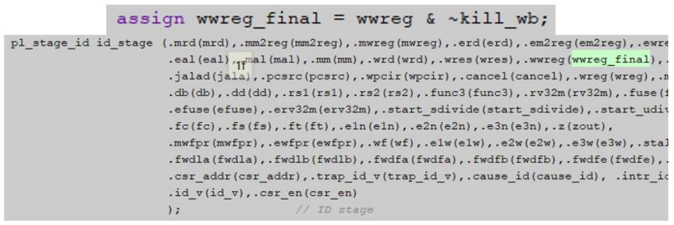


```
// ---------------- A1 redirect gate (delay if WB writes mtvec now) ----------------
localparam [11:0] CSR_MTVEC = 12'h305;

wire wb_writes_mtvec_now = csr_commit && (csr_addr_wb == CSR_MTVEC);

// If a trap request coincides with an mtvec commit, arm a one-cycle delayed redirect.
reg delayed_trap;
always @(posedge clk or negedge rstn) begin
  if (!rstn) delayed_trap <= 1'b0;
  else       delayed_trap <= wb_writes_mtvec_now ? take_trap_raw : 1'b0;
end

// Final redirect pulse: either immediate (no mtvec write) or the 1-cycle delayed one
assign take_trap = (take_trap_raw & ~wb_writes_mtvec_now) | delayed_trap;
```

Figure 55: Trap Redirect logic

5) Flushing younger instructions

When the CPU encounters a trap, the CSR Unit and Adapter will assert kill_wb. Kill_wb will prevent any register write by asserting asserting wwreg_final to 0. See Figure 56.

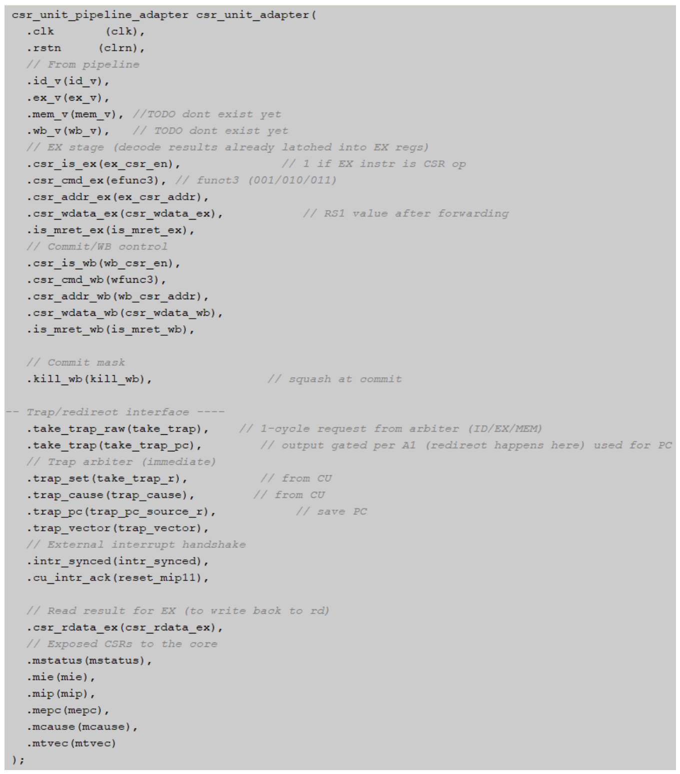


```
assign wwreg_final = wwreg & ~kill_wb;
```

```
pl_stage_id id_stage (.mrd(mrd),.mm2reg(mm2reg),.mwreg(mwreg),.erd(erd),.em2reg(em2reg),.ewre
                      .eal(eal),.mal(mal),.mm(mm),.wrd(wrd),.wres(wres),.wwreg(wwreg_final),.
                      .jalad(jala),.pcsrc(pcsrc),.wpcir(wpcir),.cancel(cancel),.wreg(wreg),.m
                      .db(db),.dd(dd),.rs1(rs1),.rs2(rs2),.func3(func3),.rv32m(rv32m),.fuse(f
                      .efuse(efuse),.erv32m(erv32m),.start_sdivide(start_sdivide),.start_udiv
                      .fc(fc),.fs(fs),.ft(ft),.e1n(e1n),.e2n(e2n),.e3n(e3n),.z(zout),
                      .mwfpr(mwfpr),.ewfpr(ewfpr),.wf(wf),.e1w(e1w),.e2w(e2w),.e3w(e3w),.stal
                      .fwdla(fwdla),.fwdlb(fwdlb),.fwdfa(fwdfa),.fwdfb(fwdfb),.fwdfe(fwdfe),.
                      .csr_addr(csr_addr),.trap_id_v(trap_id_v),.cause_id(cause_id), .intr_ic
                      .id_v(id_v),.csr_en(csr_en)
                      );                 // ID stage
```

Figure 56: kill_wb flush logic

D. Instantiating the CSR Unit Adapter



```
csr_unit_pipeline_adapter csr_unit_adapter(
  .clk       (clk),
  .rstn      (clrn),
  // From pipeline
  .id_v(id_v),
  .ex_v(ex_v),
  .mem_v(mem_v), //TODO dont exist yet
  .wb_v(wb_v),   // TODO dont exist yet
  // EX stage (decode results already latched into EX regs)
  .csr_is_ex(ex_csr_en),              // 1 if EX instr is CSR op
  .csr_cmd_ex(efunc3), // funct3 (001/010/011)
  .csr_addr_ex(ex_csr_addr),
  .csr_wdata_ex(csr_wdata_ex),           // RS1 value after forwarding
  .is_mret_ex(is_mret_ex),
  // Commit/WB control
  .csr_is_wb(wb_csr_en),
  .csr_cmd_wb(wfunc3),
  .csr_addr_wb(wb_csr_addr),
  .csr_wdata_wb(csr_wdata_wb),
  .is_mret_wb(is_mret_wb),

  // Commit mask
  .kill_wb(kill_wb),                // squash at commit

-- Trap/redirect interface ----
  .take_trap_raw(take_trap),    // 1-cycle request from arbiter (ID/EX/MEM)
  .take_trap(take_trap_pc),        // output gated per A1 (redirect happens here) used for PC
  // Trap arbiter (immediate)
  .trap_set(take_trap_r),          // from CU
  .trap_cause(trap_cause),        // from CU
  .trap_pc(trap_pc_source_r),           // save PC
  .trap_vector(trap_vector),
  // External interrupt handshake
  .intr_synced(intr_synced),
  .cu_intr_ack(reset_mip11),

  // Read result for EX (to write back to rd)
  .csr_rdata_ex(csr_rdata_ex),
  // Exposed CSRs to the core
  .mstatus(mstatus),
  .mie(mie),
  .mip(mip),
  .mepc(mepc),
  .mcause(mcause),
  .mtvec(mtvec)
);
```

Figure 57: Instantiation of CSR Unit Adapter

The CSR unit adapter is instantiated at the pl_computer.v level. Figure 51 shows a block diagram version of the instantiation. Most of the input signals from the control unit passes through either the priority arbiter in C-1 or the ID/EX pipeline register.

E. Trap entry sequencing and priority (sync vs. interrupts) — pipeline

1) Exception Trap and their stages

Detection stages. In our pipelined interrupt implementation, exception detection is by nature detected in their respective states. For example, overflow exception is an exception thrown after ALU detects an overflow in the EX stage. For our current design the 3 exceptions and their detecting stage are:

- Ecall: decode stage
- Unimplemented Instruction: decode stage
- Overflow exception: Execution stage

Similarly to single-cycle, when implemented, synchronous interrupts will take priority over external interrupt.

2) Sequencing: ID stage

Decode stage synchronous interrupts are decoded in the control unit. The decode logic code is seen in Figure 58 below. Unimplemented instruction trap is set when the decoder does not recognize the incoming instruction. System call exception is thrown when the ecall instruction is decoded. Trap_id_v will be set when any decode stage exception is detected. Note that Trap_id_v is propagated to the priority logic in Figure 52.

```
wire unimplemented_inst = ~(i_csrrw | i_csrrs | i_mret | i_ecall | i_lui |
                            i_jal | i_jalr| i_beq | i_bne | i_lw| i_sw |
                            i_addi | i_xori | i_ori | i_andi | i_slli|
                            i_srli | i_srai | i_add  | i_sub | i_slt |
                            i_xor | i_or | i_and | i_mul | i_mulh | i_mulhsu |
                            i_mulhu  | i_div | i_divu  | i_rem  | i_remu |
                            i_fadd | i_fsub | i_fmul | i_fdiv | i_fsqrt |
                            i_lwc1 | i_swc1);

wire ecall_id  = i_ecall;                          // id_v already inside
wire illegal_id= id_v & ~ecancel & unimplemented_inst;
assign intr_id   = id_v & ~ecancel & mstatus[3] & mie[11] & mip[11];
assign trap_id_v = illegal_id | i_ecall | intr_id;     // no double id_v/~ecancel
```

Figure 58: Decode stage interrupt logic

3) Sequencing: EX stage

Execution stage interrupts are decoded and written to trap_ex_v similar to decode stage. Our implementation logic can be seen in Figure 59. Note the trap_ex_cause value changes based on the exception type. This value depends on the handler entry number. Overflow is assigned 3 in our vector table

```
alu alunit (ea,eb,ealuc,ealu,zout,overflow);                // alu

wire [31:0] csrdata_or_alu;                                //Qian new
mux2x32 csr_or_alu (ealu, csr_rdata_ex, csr_is_ex, csrdata_or_alu);
mux2x32 alu_eal   (csrdata_or_alu, epc4, ecall, eal);  // Qian new

wire        trap_ex_v        = ex_v & overflow;
wire        trap_ex_is_intr  = 1'b0;
wire [3:0]  trap_ex_cause    = 4'd3;              // 3 = overflow
```

Figure 59: Execution Stage overflow detection

4) MRET Return semantics and state restore — pipeline

When mret reaches EX, pc_redirect will get mepc. The CPU will wait until WB stage to commit the jump to mtvec. Also when it commits, the same logic in Figure 31 happens. Kill_wb will be asserted again, all younger instruction are flushed.

5) External/timer/software interrupt sampling and masking — pipeline

The external interrupt executes the same way it does in single cycle interrupt. The trap and redirect mechanism follows the same as i_ecall, and unimplemented instruction.

F. Functional Verification and Testing

The verification method for pipelined is similar to single cycle. The same testing code was used. Exception Trap and redirect handling functionality was tested. The test does not include any hazards as we ran out of testing time towards the end of the semester. Since the code is the same, only the results will be posted. Figure 60 is the IlA capture of an external interrupt. Note the PC before external interrupt captured was 0x000000fc. The CSR Unit and Adapter successfully redirects the PC to 0x00000008 which is the mtvec address, shown in Figure 61. The PC performs ecall at 0x00000034, PC correctly returns to fc+4 which is 0x00000100 shown in Figure 62. Overflow exception detection and jump to 0x08 is shown in Figure 63. Unimplemented instruction exception detection and jump to 0x08 is shown in Figure 64. And lastly ecall is shown in Figure 64.

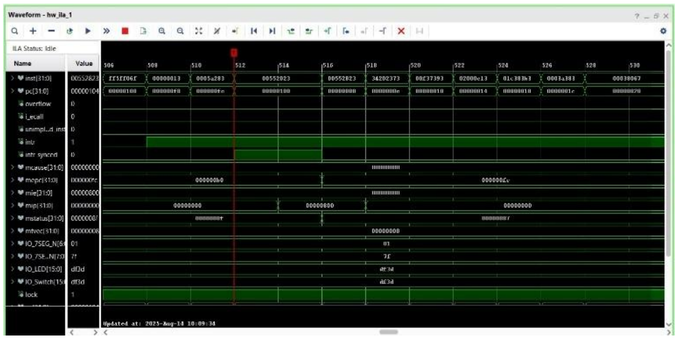


Figure 60: External Interrupt detected ILA

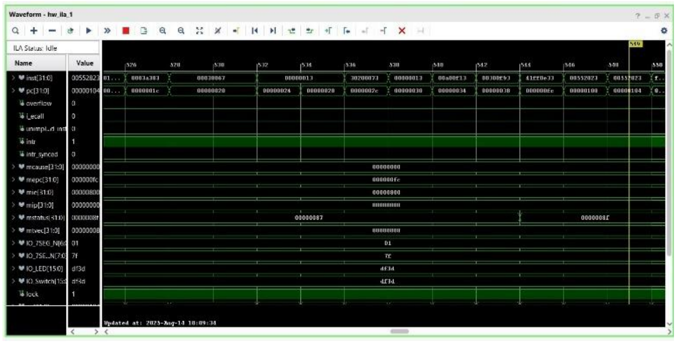


Figure 61: External Interrupt processed ILA

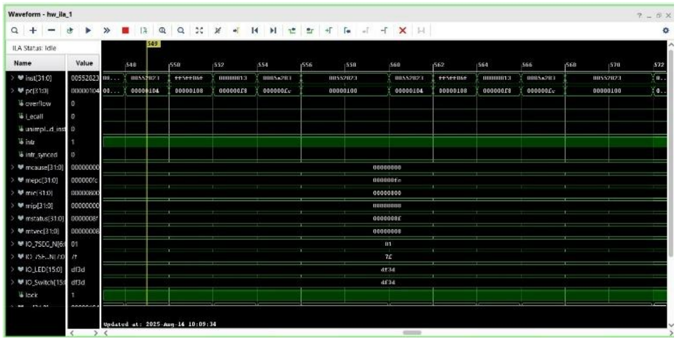


Figure 62: External Interrupt mret ILA

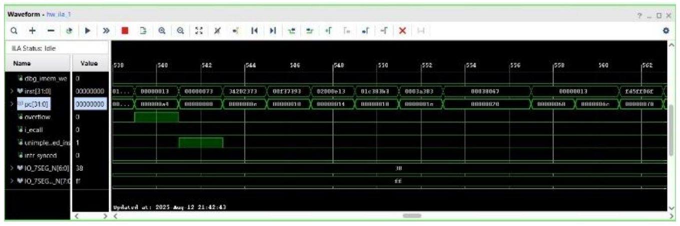


Figure 63: Overflow exception ILA

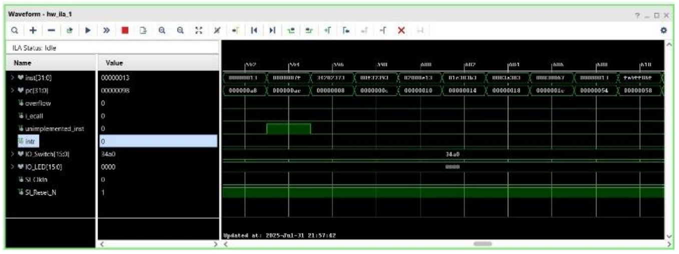


Figure 64: Unimplemented Instruction ILA

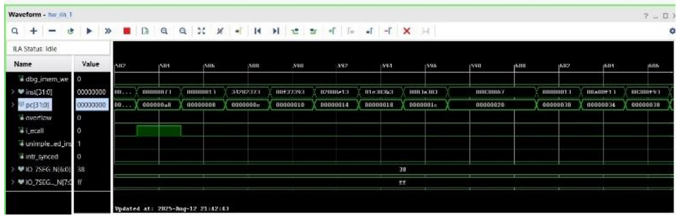


Figure 65: System call exception ILA

VI. Verification methodology

A. Directed assembly tests

The processor designs were validated primarily through directed assembly test programs, hand-written to exercise corner cases in the instruction set and the trap/interrupt subsystem. For the single-cycle interrupt implementation, tests focused on ensuring that synchronous exceptions (e.g., illegal instructions, ECALL) and asynchronous external interrupts were correctly prioritized and that the architectural state (mstatus, mepc, mcause, mtvec) was updated atomically on trap entry.

For the pipelined interrupt implementation, the emphasis was on verifying precise exceptions. Due to time constraints, only functional tests were performed. Tests confirmed that instructions younger than the faulting one were flushed and that no incorrect state updates reached the write-back stage was not done.

For the floating-point extension, directed tests were crafted to check for both functional correctness (IEEE-754 compliant results across add, multiply, divide, conversions) and hazard behavior, ensuring that multi-cycle operations did not cause RAW, WAW, or structural hazards in the pipeline. Together, these directed assembly tests provided a lightweight but targeted methodology that exposed control-path and data-path errors early in development.

B. Onboard FPGA debug with ILA (probe list)

Beyond simulation, the processor designs were synthesized, place and routed, and finally bitstream generated to deploy onto FPGA hardware. This validates that the design can execute in real-time. Debugging on FPGA was supported by Xilinx Integrated Logic Analyzer (ILA) cores, which probed selected internal signals for cycle-accurate observation. Typical probe list included the program counter, inst (instruction fetch passed to decode stage), trap control signals, CSR register values. For floating-point unit, additional probes was added to identify the result is correct. The FPGA bitstream generation and the ILA verification step ensures the design functions in a real time environment.

VII. Limitations and Future work

VIII. Conclusion

This work presented the design and implementation of a RISC-V RV32IMF processor with floating-point support, interrupts, and CSR-based exception handling in both single-cycle and pipelined forms. Verification was carried out using directed assembly tests and FPGA-based debug with ILA, confirming correct trap, interrupt, and floating-point execution. Future improvements may extend privilege support and expand verification coverage.

IX. Future work

The planned work for Fall 2025 will extend the CPU prototypes toward memory integration and advanced scheduling research. The first area of focus is a DDR2-backed cache subsystem, where a direct-mapped cache will be interfaced with external DDR2 memory to evaluate realistic memory latency, bandwidth utilization, and cache refill strategies. This step moves beyond on-chip memory toward external memory hierarchy design, providing insight into cache controller complexity and verification requirements.

In parallel, research will be conducted to identify the architectural and hardware requirements for implementing a Tomasulo-controlled multithreaded pipeline. The study will emphasize dynamic scheduling, register renaming, and hazard resolution, with the goal of outlining a set of requirements and feasibility metrics for porting Tomasulo's algorithm to a student-scale RISC-V implementation. While a full implementation is not planned within the term, this work will establish the foundation for subsequent prototypes and form the basis of multi-threaded execution exploration in future semesters.

Together, these efforts will expand the project from functional single-core execution toward more realistic system-level constraints and modern processor scheduling strategies, while maintaining a focus on producing reusable Verilog modules, testbenches, and FPGA prototypes

X. Acknowledgement

The author would like to thank Professor Nick Beser, advisor of this project, for his guidance, feedback, and support in verification and ILA integration.

XI. References

[1] "CV32E40P Control and Status Registers," *CV32E40P User Manual*, OpenHW Group, accessed Aug. 2025.

[2] IEEE, “IEEE Standard for Floating-Point Arithmetic,” IEEE Std 754-2019, approved June 13, 2019, published July 22, 2019. [Online]. Available: IEEE Standards Association website. [Accessed: Aug. 2025].

[3] Y. Li, *Computer Principles and Design in Verilog HDL*, John Wiley & Sons, 2015.