REPLACEMENT OF INTEGER MULTIPLIER IN SHAKTI C-CLASS CORE

SUMMER INTERNSHIP – I REPORT

Submitted by

AKHSHAIYA S - 2023105507

KRISHNASHREE D - 2023105539

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

REPLACEMENT OF INTEGER MULTIPLIER IN SHAKTI C-CLASS CORE

SUMMER INTERNSHIP – I REPORT

Submitted by

AKHSHAIYA S - 2023105507

KRISHNASHREE D - 2023105539

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 2023

ANNA UNIVERSITY: CHENNAI 600025

BONAFIDE CERTIFICATE

Certified that this report titled as, "REPLACEMENT OF INTEGER MULTIPLIER IN SHAKTI C-CLASS CORE" is the Bonafide work of AKHSHAIYA S (2023105507) & KRISHNASHREE D (2023105539) for 5th Semester who carried out the project work for Summer Internship-I, in the month of July 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	SIGNATURE
Dr. Nitya Ranganathan	**Mr. Sriram**
SHAKTI Team,	**SHAKTI Team,**
Indian Institute of Technology, Madras	**Indian Institute of Technology, Madras**
Chennai-600036	**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 report documents the integration of pipelined integer multipliers - specifically 4-stage and 5-stage variants - into the Shakti C-Class RISC-V processor core. These multiplier modules were sourced from the Shakti mbox repository. Our work involves replacing the default combinational multiplier with these pipelined designs, modifying the mbox.bsv module for proper interfacing, and validating the integration through comprehensive testing including RISC-V ISA tests and CoreMark benchmark runs. Key existing modules are explained, and integration and testing approaches are detailed. Performance improvements and correctness are demonstrated.

Table of Contents

CHAPTER 1

INTRODUCTION

1.1 PROJECT MOTIVATION

The SHAKTI C-Class processor, a 64-bit in-order RISC-V core developed by IIT Madras, is designed for embedded and general-purpose computing applications. While it includes a functional single-cycle combinational multiplier, the increasing demand for performance-oriented applications highlights the limitations of such non-pipelined designs. In compute-intensive tasks, the lack of pipelining becomes a bottleneck, especially when executing multiple consecutive multiply instructions.

To address this challenge, pipelined multiplier architectures offer an effective way to improve throughput without compromising the processor's clock frequency. The motivation behind this project lies in enhancing the performance of the SHAKTI C-Class core by integrating pipelined multiplication logic using the existing modules available in the SHAKTI mbox repository. This effort aligns with modern processor design principles, where arithmetic operations are optimized for latency and resource efficiency.

1.2 PROJECT OVERVIEW

This project focuses on integrating two pipelined multiplier modules - a 4-stage and a 5-stage version - into the SHAKTI C-Class core. These modules, written in Bluespec SystemVerilog (BSV), are sourced from the official SHAKTI mbox repository and support all RV64M multiplication instructions (MUL, MULH, MULHSU, and MULHU).

The implementation replaces the default single-cycle combinational multiplier by modifying the mbox.bsv and its related configuration. Careful design ensures the new pipeline stages interface correctly with the existing pipeline and memory systems. To ensure functional correctness and performance, the updated core is tested via BSV simulation, RISC-V ISA compliance tests, and benchmarked using CoreMark.

1.3 PROBLEM STATEMENT

The default multiplier in the SHAKTI C-Class core is a single-cycle combinational unit. While this simplifies design, it is sub-optimal for performance-focused workloads due to:

- Increased combinational delay for large-width multiplications
- Limited throughput in back-to-back multiplication scenarios
- Underutilization of pipelining benefits in arithmetic-heavy code

This project addresses the need for improved performance by integrating pipelined multipliers that enhance instruction throughput and reduce critical path delay, while ensuring full compliance with the RV64M specification.

1.4 OBJECTIVES

The primary objectives of this project are:

- To integrate existing 4-stage and 5-stage pipelined multiplier modules from the SHAKTI mbox repository into the SHAKTI C-Class core.
- To modify the mbox.bsv and supporting configuration files to replace the default combinational multiplier.
- To validate the functional correctness of each integrated multiplier using:
    - Custom Bluespec SystemVerilog (BSV) testbenches
    - RISC-V ISA compliance tests
    - CoreMark performance benchmarking

1.5 SUMMARY

This chapter introduced the motivation and necessity for pipelined multiplication within the SHAKTI C-Class RISC-V core. By replacing the default combinational multiplier with more efficient pipelined versions, the project aims to improve arithmetic performance and pipeline utilization. The integration of these modules ensures that the processor maintains full support for the RV64M extension while offering improved throughput. The following chapters will detail the module design, integration strategy, simulation results, and performance benchmarking.

CHAPTER 2

METHODOLOGY

2.1 BASELINE SETUP: SHAKTI C-CLASS V4.6.0 CORE

- Built and simulated the unmodified SHAKTI C-Class version 4.6.0.
- Established functional correctness and performance baseline by running the entire ISA test suite (isa/ folder from riscv-tests).
- CoreMark benchmark was also executed using the default combinational multiplier for comparison purposes.

2.2 TESTING SHAKTI C-CLASS CORE

- Executed multiply-specific ISA tests including mul.S, mulh.S, mulhsu.S, and mulw.S to verify the functionality of the default multiplier.
- Conducted CoreMark performance evaluation, logging both output and cycle counts.

CoreMark Report

System & Compilation Details

- **Test:** CoreMark 1.0 (2K performance run)
- **CoreMark Size:** 666
  Iterations: 40
- **Compiler Version:** riscv64-unknown-elf-15.1.0
  Compiler Flags: -mcmodel=medany -DCUSTOM -DPERFORMANCE_RUN=1 -DMAIN_HAS_NOARGC=1 -DHAS_STDIO -DHAS_PRINTF -DHAS_TIME_H -DUSE_CLOCK -DHAS_FLOAT -DITERATIONS=40 -O3-fno-unsafe-math-optimizations -fno-tree-loop-vectorize -fno-strict-aliasing -fgnu89-inline -fno-common -funroll-loops -finline-functions-fselective-scheduling -falign-functions=4 -falign-jumps=4 -falign-loops=4 -finline-limit=1000 -nostartfiles -nostdlib -ffast-math-fno-builtin-printf -mabi=lp64d -march=rv64imafdc -mexplicit-relocs --param max-unroll-times=2

CoreMark Output

- **Total ticks:** 14,929,853
- **Total time (secs):** 14.929853
- **Score (Iterations/Sec):** 2.679195
- **seedcrc:** 0xe9f5

CRC Results

- **crclist :** 0xe714
- **crcmatrix :** 0x1fd7
- **crcstate :** 0x8e3a
- **crcfinal :** 0x65c5

Performance Counters (as per SHAKTI event mapping)

Counter	**Value**	**Event Number (from Shakti)**	**Description**
CYCLE	16178574	N/A	Total CPU cycles
INSTRET	13011727	N/A	Instructions retired
HPM3	222601	7 (Number of branches)	Number of branch instructions
HPM4	151301	6 (Number of jumps)	Number of jump instructions

HPM5	2146500	1 (Number of mispredictions)	Branch mispredictions
HPM6	376024	9 (Number of mul/div ops)	MUL/DIV operations

Source: Shakti perf-counters.rst

2.3 MULTIPLIER MODULES FROM THE SHAKTI MBOX REPOSITORY

- Leveraged two existing pipelined multiplier modules:
    - The 4-stage pipelined multiplier
    - The 5-stage pipelined multiplier
- These modules support all necessary RV64M multiplication types and provide handshake interfaces compatible with mbox.
- We did not modify multiplier logic internally, but focused on their integration and verification with the core.

2.4 EXISTING MULTIPLIER MODULES IN SHAKTI C-CLASS

2.4.1 MBOX.BSV

- The mkmbox module acts as the central arithmetic unit, managing both multiplication and division operations.
- It instantiates combo_mul for multiplication and restoring_div for division, routing inputs appropriately.
- A FIFO (ff_ordering) is used to preserve the program order, ensuring the correct sequencing of issued operations and results.
- The module communicates with external units using standard ready/valid handshake signals:

  ma_inputs: receives arithmetic requests

  mv_ready: indicates readiness to accept a new request

  mv_output_valid and mv_output: deliver the computation results

- The design is modular, supporting seamless replacement of combo_mul with a pipelined multiplier to enhance performance.

2.4.2 SIGNEDMUL.BSV

- signedmul is a Verilog module that performs signed multiplication of two inputs.
- It is wrapped in BSV using a Black-Box Verilog Interface (BVI).
- The wrapper exposes three methods:
    1) ia and ib to provide the two operands.
    2) oc to get the signed product.
    3) Used in combo.bsv to compute multiplication results as signed_mul.oc.
- Acts as a combinational block with no internal pipeline, enabling quick integration into arithmetic units.

2.4.3 COMBO.BSV

- The mkcombo_mul module wraps the combinational multiplier (signedmul) and provides a structured interface.
- It includes register vectors to pipeline inputs and outputs based on MULSTAGES_IN and MULSTAGES_OUT, enabling configurable delay stages.
- The module handles:
    1) Input queuing (x1, x2)
    2) Result generation via signed_mul.oc
    3) Output timing alignment through stage registers
- Exposes a ready/valid interface to support integration into larger pipelines like mbox.
- This module can be directly replaced with a pipelined multiplier while maintaining compatibility with surrounding logic.

CHAPTER 3

MULTIPLIER MODULE DESIGN

3.1 4-STAGE PIPELINED MULTIPLIER

3.1.1 MODULE EXPLANATION

This module implements a 4-stage pipelined multiplier by breaking the 64-bit multiplier input into 6-bit chunks, computing partial products incrementally, and accumulating results over multiple cycles.

- Uses a send() method to accept operands (a, b) along with the RISC-V funct3 field that determines the type of multiplication operation.
- Uses a receive() method to output a tuple { valid, result }, where:

valid indicates whether the output is ready.

result is the final multiplication output, properly signed and selected based on funct3.

Supports all RV64M multiplication variants, including:

- Signed (MUL, MULH) and unsigned (MULHU) multiplications.
- Mixed signed-unsigned multiplication (MULHSU).
- 32-bit word-based multiplication in 64-bit mode (MULW).

The final result is accumulated across the pipeline stages using a shift-and-add approach. This pipelined architecture:

- Increases throughput (one output per cycle after pipeline fill).
- Reduces critical path delay compared to a fully combinational multiplier.

funct3	**Operation**	**Description**
000	MUL	Low 64 bits of signed × signed
001	MULH	High 64 bits of signed × signed
010	MULHSU	High 64 bits of signed × unsigned

011	MULHU	High 64 bits of unsigned × unsigned
100	MULW (RV64)	Low 32 bits signed, sign-extended to 64

3.1.2 TESTING ON BLUESPEC SIMULATOR (BSC)

This testbench verifies the functionality and timing of a 4-stage pipelined integer multiplier (int_multiplier_pipelined_4stages) designed in Bluespec SystemVerilog. It feeds randomized 64-bit operand pairs into the multiplier and evaluates the correctness of outputs across multiple multiplication function types (encoded via funct3).

- **Random Input Generation**:
  The testbench uses random generators to create diverse 64-bit operand pairs for extensive functional coverage.

- **Functional Mode Testing**:
  It tests multiple multiplication modes based on the RISC-V funct3 encoding:
  - 000: Standard multiplication (MUL)
  - 001: High-order signed multiplication (MULH)
  - 010: Signed-unsigned multiplication (MULHSU)
  - 011: Unsigned multiplication (MULHU)
  - 000 with word32=True (only under RV64) tests 32-bit word multiplication (MULW)

  Each stage in rule rl_stageX (X = 1 to 5) does the following:

  1. **Stage 1–4**: Sends the same rand1 and rand2 values to the multiplier with different funct3 codes (000, 001, 010, 011).

  2. **Stage 5**: Sends again with funct3 = 000, but with the word32 flag set to True (only for RV64).

- **Staged Request Feeding**:
  The operands are sent across five sequential rules (one per cycle), each triggering different multiplication modes for verification.

- **Output Monitoring**:
  A dedicated rule captures the output and logs whether it is valid along with the

corresponding multiplication type (funct3) and result. The simulation ends after 200,000 clock cycles.

TESTBENCH CODE :

```
package tb_new5;


import int_multiplier_pipelined_4stages :: *;
import Randomizable :: *;
`include "Logger.bsv"
`include "mbox_parameters.bsv"


module mk_tb_int_multiplier_pipelined_4stages();


   Reg#(Bit#(32)) cycle <- mkReg(0);
   Reg#(Bit#(32)) feed  <- mkReg(0);
   Reg#(Bit#(64)) rand1 <- mkReg(0);
   Reg#(Bit#(64)) rand2 <- mkReg(0);


   int_multiplier_pipelined_4stages::Ifc_int_multiplier dut <-
int_multiplier_pipelined_4stages::mk_int_multiplier();


   Randomize#(Bit#(64)) rand_in1 <- mkConstrainedRandomizer(64'd0,
64'hffff_ffff_ffff_ffff);
   Randomize#(Bit#(64)) rand_in2 <- mkConstrainedRandomizer(64'd0,
64'hffff_ffff_ffff_ffff);


   rule init(feed == 0);
      rand_in1.cntrl.init();
      rand_in2.cntrl.init();
      feed <= 1;
   endrule
```

```
rule rl_stage1(feed == 1);
   let a <- rand_in1.next();
   let b <- rand_in2.next();
   rand1 <= a;
   rand2 <= b;
   dut.send(a, b, 3'b000
   `ifdef RV64
      , False
   `endif
   );
   feed <= 2;
endrule

rule rl_stage2(feed == 2);
   dut.send(rand1, rand2, 3'b001
   `ifdef RV64
      , False
   `endif
   );
   feed <= 3;
endrule

rule rl_stage3(feed == 3);
   dut.send(rand1, rand2, 3'b010
   `ifdef RV64
      , False
   `endif
   );
   feed <= 4;
```

```
    endrule

    rule rl_stage4(feed == 4);
       dut.send(rand1, rand2, 3'b011
       `ifdef RV64
          , False
       `endif
       );
       feed <= 5;
    endrule

    rule rl_stage5(feed == 5);
       dut.send(rand1, rand2, 3'b000
       `ifdef RV64
          , True
       `endif
       );
       feed <= 1;
    endrule

    rule receive;
       match {.valid, .out} = dut.receive();
       Bit#(3) func =
          (feed == 1) ? 3'b000 :
          (feed == 2) ? 3'b001 :
          (feed == 3) ? 3'b010 :
          (feed == 4) ? 3'b011 : 3'b000;
       $display("Cycle %0d : Valid %0d : funct3 %0b OUT %0h", cycle, valid, func,
out);
    endrule
    rule cycling;
        cycle <= cycle + 1;
        if (cycle > 200000)
            $finish(0);
    endrule

endmodule
endpackage
```

3.2 5-STAGE PIPELINED MULTIPLIER

3.2.1 MODULE EXPLANATION

This module implements a 5-stage pipelined integer multiplier that supports all standard RV64M instructions: MUL, MULH, MULHU, MULHSU, and MULW.

Inputs: 64-bit operands a and b. Operand a is extended to 65 bits (a_ext) .

- Adding sign extension bit (bit 64) and correct handling of arithmetic shifts and prevents incorrect carry/overflow in signed multiplications.
- A 64-bit signed number can produce a 128-bit result when multiplied.
- During partial product generation, bits are shifted and added.

Working:

- Operand b is split into 6-bit chunks (total ~11 chunks for 64 bits).
- Each pipeline stage computes and adds the partial product for one chunk.
- The function fn_gen_pp(chunk, a_ext) generates a 130-bit partial product by adding (a_ext << i) for every bit i that is set in the 6-bit chunk.

Pipeline Stages:

- Stage 1–4: Sequentially process 6-bit chunks from b, compute partial products, and accumulate intermediate sums.
- Stage 5: Applies funct3 to select the correct result form (e.g., lower 64 bits for MUL, upper bits for MULH, etc.).

Interface:

- send(a, b, funct3): Inputs operands and function selector.
- receive(): Outputs { valid, result } — result is ready when valid is high.

Advantages:

- Fully pipelined: one result per cycle after initial fill.
- Reduces critical path delay (vs. single-cycle multiplier).
- Supports all RISC-V multiplication types (signed, unsigned, 32-bit word-based).

3.2.2 TESTING ON BLUESPEC SIMULATOR (BSC)

- Tests the 5-stage pipelined multiplier module: int_multiplier_pipelined_5stages.
- Inputs (rand1, rand2) are 64-bit random values, generated using Randomize.
- Five stages are used to test various multiplication operations:
    1. rl_stage1: funct3 = 000 (signed * signed)
    2. rl_stage2: funct3 = 001 (signed * signed, return upper 64 bits)
    3. rl_stage3: funct3 = 010 (signed * unsigned, return upper 64 bits)
    4. rl_stage4: funct3 = 011 (unsigned * unsigned, return upper 64 bits)
    5. rl_stage5: funct3 = 000, with word32 = True (32-bit word multiplication, RV64 only)

- All stages send the same rand1, rand2 pair with different modes to the multiplier.
- The receive rule prints the output value, valid bit, current cycle, and the funct3 field cycle variable tracks simulation progress.
- After 200,000 cycles, the simulation stops automatically.

<u>TESTBENCH CODE:</u>

```
package tb_new_5stage;


import int_multiplier_pipelined_5stages :: *;

import Randomizable :: *;

`include "Logger.bsv"

`include "mbox_parameters.bsv"


module mk_tb_int_multiplier_pipelined_5stages();


   Reg#(Bit#(32)) cycle <- mkReg(0);
   Reg#(Bit#(32)) feed  <- mkReg(0);
   Reg#(Bit#(64)) rand1 <- mkReg(0);
   Reg#(Bit#(64)) rand2 <- mkReg(0);

   int_multiplier_pipelined_5stages::Ifc_int_multiplier dut <-
int_multiplier_pipelined_5stages::mk_int_multiplier();

   Randomize#(Bit#(64)) rand_in1 <- mkConstrainedRandomizer(64'd0,
64'hffff_ffff_ffff_ffff);
   Randomize#(Bit#(64)) rand_in2 <- mkConstrainedRandomizer(64'd0,
64'hffff_ffff_ffff_ffff);

   rule init(feed == 0);
      rand_in1.cntrl.init();
      rand_in2.cntrl.init();
      feed <= 1;
   endrule

   rule rl_stage1(feed == 1);
      let a <- rand_in1.next();
      let b <- rand_in2.next();
      rand1 <= a;
      rand2 <= b;
      dut.send(a, b, 3'b000
      `ifdef RV64
         , False
      `endif
      );
      feed <= 2;
   endrule
```

```
rule rl_stage2(feed == 2);
   dut.send(rand1, rand2, 3'b001
   `ifdef RV64
      , False
   `endif
   );
   feed <= 3;
endrule

rule rl_stage3(feed == 3);
   dut.send(rand1, rand2, 3'b010
   `ifdef RV64
      , False
   `endif
   );
   feed <= 4;
endrule

rule rl_stage4(feed == 4);
   dut.send(rand1, rand2, 3'b011
   `ifdef RV64
      , False
   `endif
   );
   feed <= 5;
endrule

rule rl_stage5(feed == 5);
   dut.send(rand1, rand2, 3'b000
   `ifdef RV64
```

```
            , True
         `endif
         );
         feed <= 1;
      endrule

      rule receive;
         match {.valid, .out} = dut.receive();
         Bit#(3) func =
            (feed == 1) ? 3'b000 :
            (feed == 2) ? 3'b001 :
            (feed == 3) ? 3'b010 :
            (feed == 4) ? 3'b011 : 3'b000;
         $display("Cycle %0d : Valid %0d : funct3 %0b OUT %0h", cycle, valid,
func, out);
      endrule

      rule cycling;
         cycle <= cycle + 1;
         if (cycle > 200000)
            $finish(0);
      endrule

endmodule
endpackage
```

CHAPTER 4
INTEGRATION OF MULTIPLIERS

4.1 4-STAGE MULTIPLIER INTEGRATION

4.1.1 MODIFYING MBOX.BSV CODE

- Replaced instantiation of original combo multiplier with the 4-stage pipelined multiplier module.
- Increased the depth of the ff_ordering FIFO to 4 to track pipeline latency and maintain program-order result sequencing.
- Adapted input methods to use the send() function of the 4-stage multiplier.
- Changed result collection rules to poll receive() and enqueue valid results.
- Kept divider integration unaltered.
- Preserved assertion and logging mechanisms in the mbox.

MODIFIED MBOX.BSV

```
package mbox;
import ccore_types    :: *;
`include "Logger.bsv"
import int_multiplier_pipelined_4stages :: *;  // NEW IMPORT
import restoring_div  :: * ;
`ifdef async_rst
import SpecialFIFOs_Modified :: * ;
`else
import SpecialFIFOs :: * ;
`endif
import FIFOF          :: * ;
import TxRx           :: * ;
import Assert         :: * ;

typedef struct{
`ifdef RV64
```

```
  Bool wordop ;
`endif
  Bit#(`xlen) in1;
  Bit#(`xlen) in2;
  Bit#(3) funct3;
}MBoxIn deriving(Bits, FShow, Eq);

typedef struct{
  Bool mul;
  Bool div;
} MBoxRdy deriving(Bits, FShow, Eq);

interface Ifc_mbox;
           method Action ma_inputs(MBoxIn inputs);
  method MBoxRdy mv_ready;
  method TXe#(Bit#(`xlen)) tx_output;
  `ifdef arith_trap
    method Action ma_arith_trap_en(Bit#(1) en);
    method TXe#(Tuple2#(Bool, Bit#(`causesize))) tx_arith_trap_output;
  `endif
endinterface: Ifc_mbox

`ifdef mbox_noinline
`ifdef core_clkgate
(*synthesize,gate_all_clocks*)
`else
(*synthesize*)
`endif
`endif
module mkmbox#(parameter Bit#(`xlen) hartid) (Ifc_mbox);
```

```
String mbox = "";

Ifc_int_multiplier mul_ <- mk_int_multiplier;  // CHANGED: Replaced combo_mul
Ifc_restoring_div div_ <- mkrestoring_div(hartid);

FIFOF#(Bool) ff_ordering <- mkUGSizedFIFOF(max(`MULSTAGES_TOTAL,2));
TX#(Bit#(`xlen)) tx_mbox_out <- mkTX;
`ifdef arith_trap
  TX#(Tuple2#(Bool, Bit#(`causesize))) tx_arith_trap_out <- mkTX;
`endif

rule rl_fifo_full(!tx_mbox_out.u.notFull());
  `logLevel( mbox, 0, $format("[%2d]MBOX: Buffer is FULL",hartid))
  dynamicAssert(!ff_ordering.first && !div_.mv_output_valid,
            "MUL/DIV provided result when O/P FIFO is full");
endrule

rule rl_capture_output(ff_ordering.notEmpty);
  if (ff_ordering.first) begin // mul operation
    match {.valid, ._x} = mul_.receive();  // CHANGED: New multiplier interface
    if (valid == 1) begin
      tx_mbox_out.u.enq(_x);
      `ifdef arith_trap
        tx_arith_trap_out.u.enq(unpack(0));
      `endif
      ff_ordering.deq;
      `logLevel( mbox, 0, $format("MBOX: Collecting MUL o/p"))
    end
    else
      `logLevel( mbox, 0, $format("MBOX: Waiting for Mul o/p"))
    end
    else begin // div operation
      if (div_.mv_output_valid) begin
        let _x <- div_.mv_output;
        tx_mbox_out.u.enq(_x);
        `ifdef arith_trap
          tx_arith_trap_out.u.enq(div_.mv_arith_trap_out);
        `endif
        ff_ordering.deq;
        `logLevel( mbox, 0, $format("MBOX: Collecting DIV o/p"))
      end
      else
        `logLevel( mbox, 0, $format("MBOX: Waiting for Div o/p"))
    end
  endrule


          method Action ma_inputs(MBoxIn inputs);
  `ifdef ASSERT
    dynamicAssert(ff_ordering.notFull(), "Enquing MBOX inputs to full fifo");
  `endif
    if( inputs.funct3[2] == 0 ) begin // Multiplication ops
      `logLevel( mbox, 0, $format("MBOX: To MUL. Op1:%h Op2:%h ", inputs.in1,
inputs.in2 ))
      mul_.send(inputs.in1, inputs.in2, inputs.funct3 `ifdef RV64 ,inputs.wordop `endif
);  // CHANGED: New interface
      ff_ordering.enq(True);
    end
    else begin // Division ops
      ff_ordering.enq(False);
      `logLevel( mbox, 0, $format("MBOX: To DIV. Op1:%h Op2:%h sign:%b",
inputs.in1, inputs.in2, inputs.in1[valueOf(`xlen)-1] ))
```

```
      div_.ma_inputs( inputs.in1, inputs.in2, inputs.funct3 `ifdef RV64 ,inputs.wordop
`endif );
    end
  endmethod


  method mv_ready = MBoxRdy{
    mul: ff_ordering.notFull,  // CHANGED: New multiplier is always ready
    div: div_.mv_ready && ff_ordering.notFull()
  };


  method tx_output = tx_mbox_out.e;
  `ifdef arith_trap
    method Action ma_arith_trap_en(Bit#(1) en);
      `logLevel( mbox, 0, $format("MBOX: arith_en: %h ", en ))
      div_.ma_div_arith_trap_en(en);
    endmethod
    method tx_arith_trap_output = tx_arith_trap_out.e;
  `endif
endmodule
endpackage
```

4.1.2 EXPLANATION

- **Pipelined Multiplier (4-stages):** The multiplier is explicitly now a 4-stage pipeline (int_multiplier_pipelined_4stages), meaning results are delayed.
- **receive() Method for Multiplier:** Multiplier results are fetched using a mul_.receive() method, which returns a (valid bit, result) tuple, reflecting its pipelined nature.
- **FIFO for Result Ordering:** An ff_ordering FIFO (depth based on max of 4 multiplier stages or 2 for divider) ensures results are delivered in the same order inputs were received.
- **Multiplier Always Ready:** The multiplier's input (mul_.send) is assumed "always ready," so mv_ready.mul just checks the ordering FIFO's capacity.

- **Simplified rl_capture_output:** The logic for capturing results from both multiplier and divider is streamlined.

4.1.3 TESTING ON ISA

- Deployed updated core in RTL simulation with the integrated 4-stage multiplier.
- Ran full rv64um ISA test suite targeting multiply/divide instructions.
- All tests passed, confirming the correctness of the integration.

4.2 5-STAGE MULTIPLIER INTEGRATION

4.2.1 MODIFYING MBOX.BSV CODE

- Similar to 4-stage integration, replaced multiplier instantiation with the 5-stage module.
- ff_ordering FIFO depth increased to 5.
- Adapted handshake interface usage accordingly.
- Maintained modular interfaces preserving divider handling, assertion checks, and logging.

MODIFIED MBOX.BSV

```
package mbox;


import ccore_types      :: *;
`include "Logger.bsv"


// Import the new pipelined integer multiplier module
import int_multiplier_pipelined_5stages :: * ; // <--- NEW IMPORT


import restoring_div  :: * ;
`ifdef async_rst
import SpecialFIFOs_Modified :: * ;
`else
import SpecialFIFOs :: * ;
`endif
import FIFOF          :: * ;
import TxRx          :: * ;
import Assert        :: * ;

typedef struct{
`ifdef RV64
  Bool wordop ;
`endif
  Bit#(`xlen) in1;
  Bit#(`xlen) in2;
  Bit#(3) funct3;
}MBoxIn deriving(Bits, FShow, Eq);

typedef struct{
  Bool mul;
  Bool div;
} MBoxRdy deriving(Bits, FShow, Eq);

interface Ifc_mbox;
  method Action ma_inputs(MBoxIn inputs);
  method MBoxRdy mv_ready;
  method TXe#(Bit#(`xlen)) tx_output;
  `ifdef arith_trap
    method Action ma_arith_trap_en(Bit#(1) en);
    method TXe#(Tuple2#(Bool, Bit#(`causesize))) tx_arith_trap_output;
  `endif
endinterface: Ifc_mbox

`ifdef mbox_noinline
`ifdef core_clkgate
(*synthesize,gate_all_clocks*)
```

```
`else
(*synthesize*)
`endif
`endif
module mkmbox#(parameter Bit#(`xlen) hartid) (Ifc_mbox);
  String mbox = "";

  // Instantiate the new pipelined multiplier
  Ifc_int_multiplier mul_pipe_ <- mk_int_multiplier; // <--- REPLACED
Ifc_combo_mul mul_
  Ifc_restoring_div div_ <- mkrestoring_div(hartid);

  // The FIFO depth for ordering should now consider the maximum latency.
  // Assuming `MULSTAGES_TOTAL` from the old definition was sufficient or 5
stages.
  // Here, we explicitly use 5 for the multiplier latency.
  FIFOF#(Bool) ff_ordering <- mkUGSizedFIFOF(max(5,2)); // <--- Adjusted FIFO
depth for multiplier latency
  TX#(Bit#(`xlen)) tx_mbox_out <- mkTX;
  `ifdef arith_trap
    TX#(Tuple2#(Bool, Bit#(`causesize))) tx_arith_trap_out <- mkTX;
  `endif

  /*doc:rule: */
  rule rl_fifo_full(!tx_mbox_out.u.notFull());
    `logLevel( mbox, 0, $format("[%2d]MBOX: Buffer is FULL",hartid))
    // Dynamic asserts should now check the *new* multiplier's output validity
    // FIXED: Use unpack to convert Bit#(1) to Bool
    dynamicAssert(!unpack(tpl_1(mul_pipe_.receive)) ,"MUL provided result when
O/P FIFO is full");
```

```
    dynamicAssert(!div_.mv_output_valid ,"DIV provided result when O/P FIFO is
full");
  endrule:rl_fifo_full


  /*doc:rule: */
  rule rl_capture_output(ff_ordering.notEmpty);
    if (ff_ordering.first) begin // mul operation
      // Check the valid bit from the new multiplier's receive method
      // FIXED: Use unpack to convert Bit#(1) to Bool for the if condition
      if (unpack(tpl_1(mul_pipe_.receive))) begin
        let _x = tpl_2(mul_pipe_.receive); // Extract the result from the tuple
        tx_mbox_out.u.enq(_x);
        `ifdef arith_trap
          tx_arith_trap_out.u.enq(unpack(0)); // Multiplier doesn't generate arithmetic
traps based on its interface
        `endif
        ff_ordering.deq;
        `logLevel( mbox, 0, $format("MBOX: Collecting MUL o/p"))
      end
      else
        `logLevel( mbox, 0, $format("MBOX: Waiting for Mul o/p"))
    end
    else if (!ff_ordering.first) begin // div operation
      if (div_.mv_output_valid) begin
        let _x <- div_.mv_output;
        tx_mbox_out.u.enq(_x);
        `ifdef arith_trap
          tx_arith_trap_out.u.enq(div_.mv_arith_trap_out);
        `endif
        ff_ordering.deq;
        `logLevel( mbox, 0, $format("MBOX: Collecting DIV o/p"))
      end
      else
        `logLevel( mbox, 0, $format("MBOX: Waiting for Div o/p"))
    end
  endrule: rl_capture_output

  method Action ma_inputs(MBoxIn inputs);
  `ifdef ASSERT
    dynamicAssert(ff_ordering.notFull(), "Enquing MBOX inputs to full fifo");
  `endif
    if( inputs.funct3[2] == 0 ) begin // Multiplication ops
      `logLevel( mbox, 0, $format("MBOX: To MUL. Op1:%h Op2:%h ", inputs.in1,
inputs.in2 ))
      // Use the new multiplier's send method
      mul_pipe_.send(inputs.in1, inputs.in2, inputs.funct3 `ifdef RV64 ,inputs.wordop
`endif );
      ff_ordering.enq(True);
    end
    if (inputs.funct3[2] == 1) begin
      ff_ordering.enq(False);
      `logLevel( mbox, 0, $format("MBOX: To DIV. Op1:%h Op2:%h sign:%b",
inputs.in1, inputs.in2, inputs.in1[valueOf(`xlen)-1] ))
      div_.ma_inputs( inputs.in1, inputs.in2, inputs.funct3 `ifdef RV64 ,inputs.wordop
`endif ) ;
    end
  endmethod

  // The 'ready' signal for the new multiplier isn't explicitly exposed as 'mv_ready'
  // It's an `always_ready` method. So, the multiplier is always ready to accept input.
  // We only need to check if the ordering FIFO is not full.
  method mv_ready= MBoxRdy{mul: ff_ordering.notFull, div: div_.mv_ready &&
ff_ordering.notFull()};


  method tx_output = tx_mbox_out.e;
  `ifdef arith_trap
    method Action ma_arith_trap_en(Bit#(1) en);
      `logLevel( mbox, 0, $format("MBOX: arith_en: %h ", en ))
        div_.ma_div_arith_trap_en(en);
    endmethod
    method tx_arith_trap_output = tx_arith_trap_out.e;
  `endif
endmodule
endpackage
```

4.2.2 Explanation

- It integrates a int_multiplier_pipelined_5stages module, implying multiplication operations now take 5 clock cycles to complete.

- Order-Preserving FIFO (ff_ordering): A FIFO ff_ordering (depth 5, max of mul and div latency) is used to track the order of operations (true for multiply, false for divide) to maintain program order.

- Dispatch and Receive: The ma_inputs method dispatches operations to either the multiplier (mul_pipe_.send) or the divider (div_.ma_inputs). rl_capture_output rule waits for results and enqueues them into tx_mbox_out.

- Multiplier Result Handling: The mul_pipe_.receive method is a tuple Tuple2#(Bit#(1), Bit#(xlen))wheretpl_1is a valid bit (needsunpackto convert toBool) and tpl_2` is the result.

- Ready Signal: The mv_ready method indicates the mbox is ready for new inputs if the ff_ordering FIFO isn't full and the divider (if applicable) is ready.

- Arithmetic Traps (Conditional): If arith_trap is defined, it handles arithmetic trap signals, primarily from the divider, which can generate such traps.

4.2.3 Testing on ISA

- RTL simulations with the 5-stage integrated core passed all relevant rv64um ISA test cases.
- CoreMark benchmark execution confirmed functionality and registered improved throughput relative to the baseline.

CHAPTER 5

RESULTS AND PERFORMANCE

CONFIGURATION	ISA TEST	COREMA-RK	CYCLES (ticks)	COMMENTS
Default (combo_mul)	Pass	Passed	1,61,78,574	Reference implementation
Default (combo_mul) With higher latency (5 cycles)	Pass	Passed	1,72,86,993	To check if the ticks varies on changing latency 6.85% increase vs default
4-Stage Pipelined	Pass	Passed	1,73,06,64 6	6.97% increase vs default
5-Stage Pipelined	Pass	Passed	1,76,82,670	9.29% increase vs default

CHAPTER 6

CONCLUSION

The project successfully integrated 4-stage and 5-stage pipelined multiplier modules from the Shakti mbox repository into the Shakti C-Class core. Modifications to the mbox module enabled seamless multi-cycle latency handling and ensured program order correctness. Comprehensive functional testing at unit and system levels validated correctness. Performance evaluations indicate substantial throughput improvements with the pipelined designs compared to the original combinational multiplier.

Key Learnings:

- Gained hands-on experience with RISC-V-based open-source hardware, especially the Shakti ecosystem.
- Learned integration techniques for pipelined hardware units into existing arithmetic logic.
- Understood the role of handshake protocols (valid/ready) in synchronizing data flow in multi-cycle hardware modules.
- Acquired proficiency in simulating and benchmarking hardware designs using testbenches, riscv-tests, and CoreMark.

- The performance results show that increasing the multiplier latency, whether by higher cycle latency or pipelining, leads to a noticeable increase in total CPU cycles - ranging from about 6.8% to 9.3% compared to the default single-cycle combinational multiplier. This happens because, although pipelining allows breaking down the multiplication into stages, the SHAKTI multiplier design executes only one multiplication instruction at a time, causing subsequent multiply operations to stall until the previous completes. As a result, the longer latency directly adds to the overall execution time, reflecting in higher cycle counts without increasing instruction throughput for multiply operations.

Future Work:

- Extend support to other arithmetic operations like division and floating-point multiplication using pipelined approaches.

REFERENCES

[1] Shakti Project – C-Class Core, *mbox module (unmodified)*, GitLab Repository, version 4.6.0 tag.
https://gitlab.com/shaktiproject/cores/c-class/-/tree/4.6.0/src/mbox

[2] Shakti Project – C-Class Core, *Sample Configuration Files (YAML)*, GitLab Repository, master branch.
https://gitlab.com/shaktiproject/cores/c-class/-/tree/master/sample_config

[3] Shakti Project – mbox Core, *Pipelined Multiplier (4-stage and 5-stage) and mbox parameters*, GitLab Repository. *(Internal reference)*
shakti / cores / mbox · GitLab

[4] Shakti Project – Benchmarks Core, *CoreMark Benchmark Suite*, GitLab Repository.
https://gitlab.com/shaktiproject/cores/benchmarks/-/tree/master/coremarks

[5] Shakti Project – RISC-V Tests Core, *ISA Test Suite*, GitLab Repository.
https://gitlab.com/shaktiproject/cores/riscv-tests/-/tree/master/isa

[6] Shakti Project – Common Verilog Repository, *signedmul Verilog and BSV Wrapper Code*, GitLab Repository.
https://gitlab.com/shaktiproject/common_verilog/-/tree/master

[7] Shakti Project – C-Class Core, *Performance Counter Descriptions (used in CoreMark statistics mapping)*, GitLab Repository.
https://gitlab.com/shaktiproject/cores/c-class/-/blob/master/perf-counters.rst