A Chisel Implementation of RV32I Core

Author:

Date:

Table of Contents

1. Overview....3
  1.1. SOC for the Core....3
  1.2. SOC address arrangement....4
  1.3. Folder Organization....4
2. Benchmark....4
3. Simulation....5
4. FPGA Platform....6
5. Details of the Core....10
  5.1. Organization....10
  5.2. Fetch Stage....10
  5.3. Decode Stage....11
  5.4. Execute Stage....12
  5.5. Memory Stage....12
  5.6. Writeback Stage....13
  5.7. Forward Unit....13
  5.8. Hazard Detection....13
6. Conclusions and Future Works....13

Table of Figures

Figure 1: Diagram of the Core....3
Figure 2: Diagram of the SOC....3
Figure 3: Address Space....4
Figure 4: Folder Structure....4
Figure 5: Dhrystone customization(the differences)....5
Figure 6: Compiler Options....5
Figure 7: Simulation Results....5
Figure 8: Performance Comparison....6
Figure 9: Photo of MicroZUS....6
Figure 10: Resource Consumption....6
Figure 11: IO and Timing constraints....7
Figure 12: Timing Summary....8
Figure 13: Max Delay Path....8
Figure 14: Serial Setup....9
Figure 15: FPGA Result....9
Figure 16: Recapture of the Diagram....10
Figure 17: Decode Code Samples....11

1. Overview

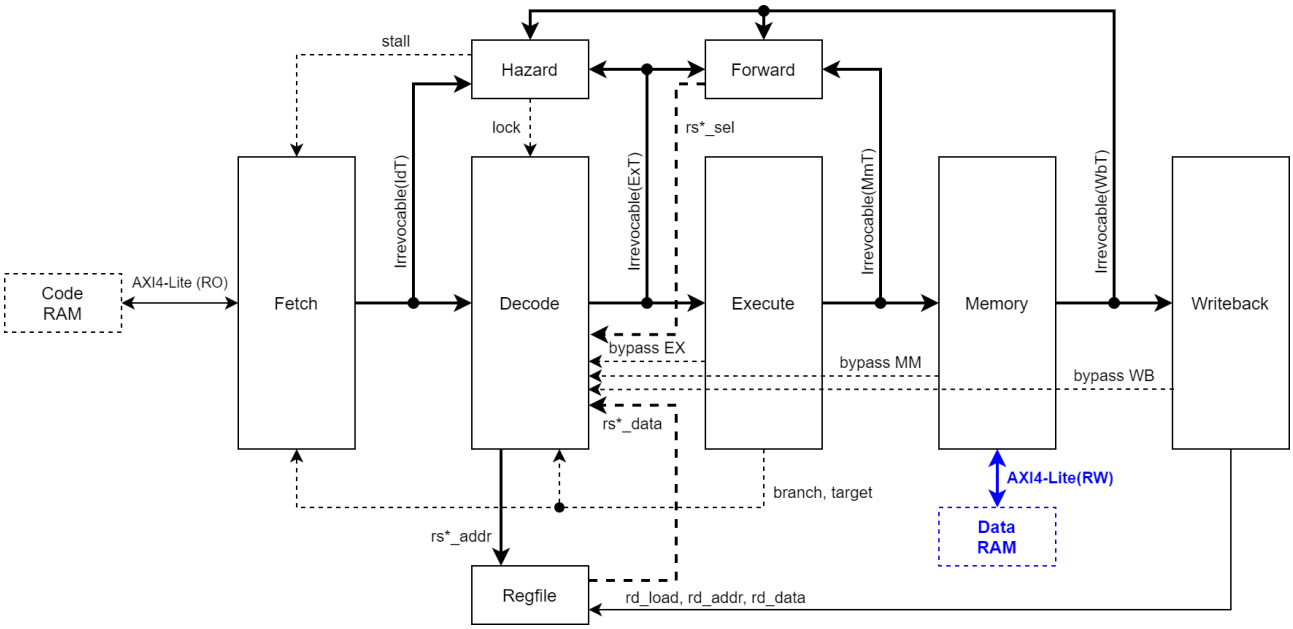

Figure 1: Diagram of the Core

As show in Figure 1, the design is a classic 5-stage pipelined architecture, which is described with details in almost every computer architecture textbook.

Instructions are fetched from the Code RAM and passed down through the pipeline. The register file has two read ports, addressed by RS1 and RS2 from Decode stage. The write port update register states from Writeback stage. The Hazard unit and Forward unit detect special conditions for certain instruction combinations. The taken branch causes a two-cycle penalty in the execution flow, since the instructions held in Fetch stage and Decode stage are canceled. Branches not taken take no penalty. So it is a always-not-take prediction scheme.

Interrupt handling is not fully tested due to limited time. Control/Status Registers are not implemented, either. They can be elaborated in future work.

1.1. SOC for the Core

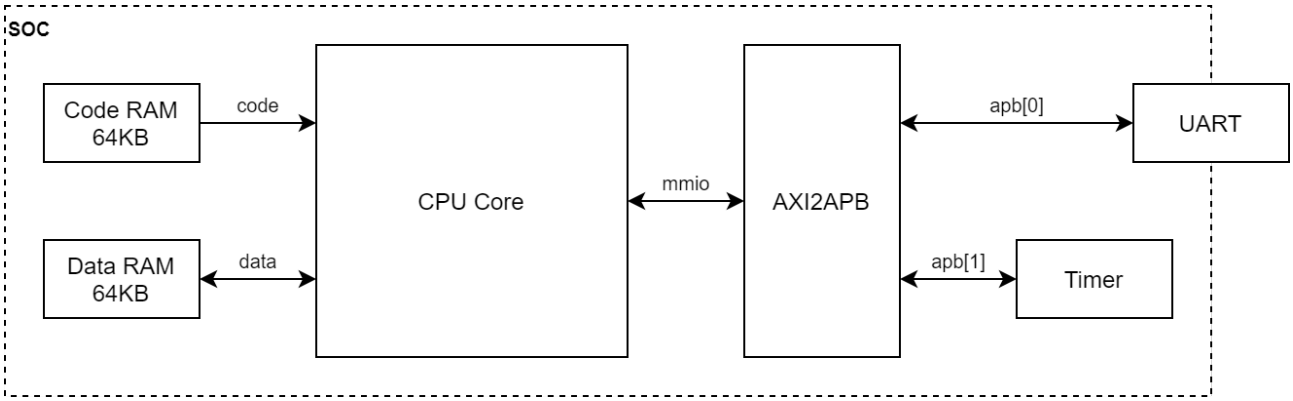

Figure 2: Diagram of the SOC

To test the core, we built a system-on-chip (SOC) by adding some peripherals. The UART communicates with physical world. Since interrupt handling is not available, the design can only do output

at the moment, which is adequate for performance benchmarking. The SOC is illustrated in Figure 2.

1.2. SOC address arrangement

The details of UART and Timer are described in “board.h” in $PROJ/src/main/cc/common.

Figure 3: Address Space

Function	Base	Size
Instructions (Code)	0x0	0x10000 (64KBytes)
Data	0x10000	0x10000 (64KBytes)
UART	0x80000000	0x1000 (4KBytes)
Timer	0x80001000	0x1000 (4KBytes)

1.3. Folder Organization

Folder is organized as the following table.

Figure 4: Folder Structure

Folder	**Description**
`Chisel-rv32`	Top, project setup scripts
`+ doc`	Documents
`+ fpga`	FPGA implementation files
`+ src`	Source codes
`| |+ main/scala/rv32`	Chisel source codes
`| |+ main/cc/common`	Board definitions, headers, linker scripts
`| |+ main/cc/benchmark`	Dhrystone source codes
`| |+ main/cc/sample`	Sample files for building C source codes
`| |+ main/cc/testbench`	Sample files for building ASM source codes
`| |+ main/sv`	Verilog/SystemVerilog sources for SOC
`+ sim`	Simulation testbench, scripts and results

2. Benchmark

We chose Dhrystone here due to limited time. Other benchmarks can be evaluated in future works. For example, Coremark, Mediabench, etc. The source code of Dhrystone benchmark is clone from: https://github.com/riscv/riscv-tests. To compile Dhrystone we need to do a little bit customization on the source, by defining some constants and helper functions. Which are shown in Figure 5.

Figure 5: Dhrystone customization(the differences)

```
diff -Z -r /home/chuser/work/riscv-tests/benchmarks/dhrystone/dhrystone.h ./dhrystone.h
384c384
< #define HZ 1000000
---
> #define HZ 50000000
386,388c386,391
< #define CLOCK_TYPE "rdcycle()"
< #define Start_Timer() Begin_Time = read_csr(mcycle)
< #define Stop_Timer() End_Time = read_csr(mcycle)
---
> #define CLOCK_TYPE "timer()"
> #define Start_Timer() Begin_Time = (long)timer_start(0)
> #define Stop_Timer() End_Time = (long)timer_stop(0)
> #include "board.h"
diff -Z -r /home/chuser/work/riscv-tests/benchmarks/dhrystone/dhrystone_main.c ./dhrystone_main.c
16c17,27
< #include "util.h"
---
> #define SIMULATION
>
> #ifdef SIMULATION
> #define SYS_FREQ  0
> #define BAUD_RATE 1
> #else
> #define SYS_FREQ  50000000
> #define BAUD_RATE 115200
> #endif
>
> //#include "util.h"
70a82,83
>   uart_init(SYS_FREQ, BAUD_RATE, 0x3);
>
237a251
>   stop_simulation();
```

The compiler is obtained from SiFive, version “riscv64-unknown-elf-gcc (SiFive GCC 8.3.0-2020.04.0) 8.3.0”. Compiling options are:

Figure 6: Compiler Options

```
-g -O3 -march=rv32i -mabi=ilp32 -nostdlib -nostartfiles
```

3. Simulation

Simulation is done with Questasim and XSIM. The scripts are put in $PROJ/sim

Figure 7: Simulation Results

```
# [12145190ns] UART: Microseconds for one run through Dhrystone: 23
# [12245330ns] UART: Dhrystones per Second:                      3677
# ** Info: all tests complete
#    Time: 12247170 ns  Scope: cpubench.wait_finish File: /home/chuser/work/fiverr/itecgo/chisel-rv32/sim/cpubench.sv Line: 59
# 1
# Break in Task wait_finish at /home/chuser/work/fiverr/itecgo/chisel-rv32/sim/cpubench.sv line 60
# End time: 15:11:51 on Oct 10,2020, Elapsed time: 1:08:06
# Errors: 0, Warnings: 1
```

The Dhrystone performance is 3677 Dhrystones/second. The DIMPS/MHz number is derived by dividing previous number with 1757, which is 3677/1757 = 2.09 DIMPS/MHz.

Comparing to cores on the market: (https://github.com/lowRISC/rocket)

Figure 8: Performance Comparison

ISA/Implementation	ARM Cortex R5	RISCV Rocket	Our Core
ISA Register Width	32	64	32
Frequency	>1GHz	>1GHz	50MHz
DMIPS/MHz	1.57	1.72	2.09

Our implementation has quite good performance number. The reason behind this is that we do not implement caches, and all the code and data are stored in L1 scratch RAM. It is therefore no cache miss issues, which contributes high ratio of the final results.

4. FPGA Platform

We chose MicroZus MZ7020 from MicroPhase. It is based on XC7Z020CLG400I-2, with 85K logic cells and 4.9Mbit blockrams.

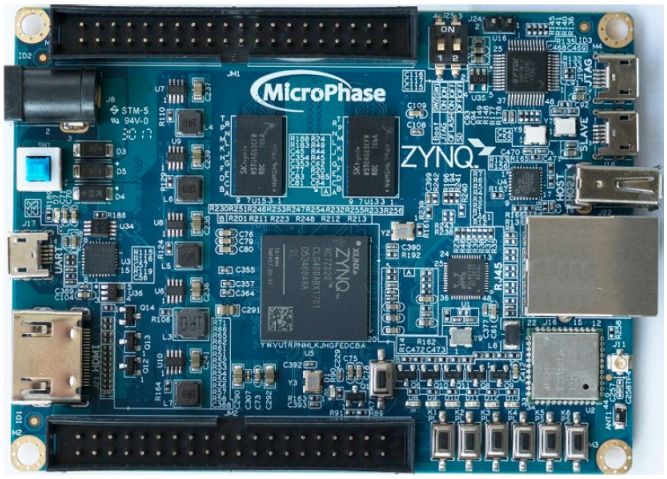

Figure 9: Photo of MicroZUS

Our environment is Vivado 2019.2. The resource consumption of our SOC after implementation (place and route).

Figure 10: Resource Consumption

```
Slice Logic
---------

+----------------------------+------+-------+-----------+-------+
|          Site Type         | Used | Fixed | Available | Util% |
+----------------------------+------+-------+-----------+-------+
| Slice LUTs                 | 6203 |     0 |     53200 | 11.66 |
|   LUT as Logic             | 4988 |     0 |     53200 |  9.38 |
|   LUT as Memory            | 1215 |     0 |     17400 |  6.98 |
|     LUT as Distributed RAM |  200 |     0 |           |       |
|     LUT as Shift Register  | 1015 |     0 |           |       |
| Slice Registers            | 7048 |     0 |    106400 |  6.62 |
|   Register as Flip Flop    | 7040 |     0 |    106400 |  6.62 |
|   Register as Latch        |    8 |     0 |    106400 | <0.01 |
| F7 Muxes                   |  171 |     0 |     26600 |  0.64 |
| F8 Muxes                   |   26 |     0 |     13300 |  0.20 |
+----------------------------+------+-------+-----------+-------+

Memory
---------
```

```
+-------------------+------+-------+-----------+-------+
|     Site Type     | Used | Fixed | Available | Util% |
+-------------------+------+-------+-----------+-------+
| Block RAM Tile    |   36 |     0 |       140 | 25.71 |
|   RAMB36/FIFO*    |   36 |     0 |       140 | 25.71 |
|     RAMB36E1 only |   36 |       |           |       |
|   RAMB18          |    0 |     0 |       280 |  0.00 |
+-------------------+------+-------+-----------+-------+
```

IO Assignment and Timing Constraints

Figure 11: IO and Timing constraints

```
set_property PACKAGE_PIN K17 [get_ports PS_CLK_50M]
set_property IOSTANDARD LVCMOS33 [get_ports PS_CLK_50M]

set_property PACKAGE_PIN E17 [get_ports FPGA_GRST]
set_property IOSTANDARD LVCMOS33 [get_ports FPGA_GRST]

set_property PACKAGE_PIN M15 [get_ports {gpio_o[0]}]
set_property PACKAGE_PIN G14 [get_ports {gpio_o[1]}]
set_property PACKAGE_PIN M17 [get_ports {gpio_o[2]}]
set_property PACKAGE_PIN G15 [get_ports {gpio_o[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {gpio_o[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {gpio_o[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {gpio_o[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {gpio_o[1]}]

set_property PACKAGE_PIN T19 [get_ports PS_UART_RX]
set_property PULLUP true [get_ports PS_UART_RX]
set_property IOSTANDARD LVCMOS33 [get_ports PS_UART_RX]

set_property PACKAGE_PIN T14 [get_ports PS_UART_TX]
set_property PULLUP true [get_ports PS_UART_TX]
set_property IOSTANDARD LVCMOS33 [get_ports PS_UART_TX]

create_clock -period 20.000 -name CLK -waveform {0.000 10.000} -add [get_ports PS_CLK_50M]

set_input_delay -clock [get_clocks CLK] -min -add_delay 1.000 [get_ports FPGA_GRST]
set_input_delay -clock [get_clocks CLK] -max -add_delay 2.000 [get_ports FPGA_GRST]
set_input_delay -clock [get_clocks CLK] -min -add_delay 1.000 [get_ports PS_UART_RX]
set_input_delay -clock [get_clocks CLK] -max -add_delay 2.000 [get_ports PS_UART_RX]
set_output_delay -clock [get_clocks CLK] -min -add_delay 0.000 [get_ports {gpio_o[*]}]
set_output_delay -clock [get_clocks CLK] -max -add_delay 2.000 [get_ports {gpio_o[*]}]
set_output_delay -clock [get_clocks CLK] -min -add_delay 0.000 [get_ports PS_UART_TX]
set_output_delay -clock [get_clocks CLK] -max -add_delay 2.000 [get_ports PS_UART_TX]
```

Timing summary after P&R (All met)

```
Design Timing Summary
---------------------

    WNS(ns)      TNS(ns)  TNS Failing Endpoints  TNS Total Endpoints      WHS(ns)      THS(ns)  THS Failing Endpoints  THS Total Endpoints     WPWS(ns)     TPWS(ns)  TPWS Failing Endpoints  TPWS Total Endpoints
    -------      -------  ---------------------  -------------------      -------      -------  ---------------------  -------------------     --------     --------  ----------------------  --------------------
      3.303        0.000                      0                18613        0.056        0.000                      0                18597        8.870        0.000                       0                  9322
```

Figure 12: Timing Summary

Max Delay Path

```
Max Delay Paths
--------------------------------------------------------------------------------------
Slack (MET) :             3.303ns  (required time - arrival time)
  Source:                 x_uart/regs_q_reg[7][1]_lopt_replica/C
                            (rising edge-triggered cell FDCE clocked by CLK  {rise@0.000ns fall@10.000ns period=20.000ns})
  Destination:            gpio_o[1]
                            (output port clocked by CLK  {rise@0.000ns fall@10.000ns period=20.000ns})
  Path Group:             CLK
  Path Type:              Max at Slow Process Corner
  Requirement:            20.000ns  (CLK rise@20.000ns - CLK rise@0.000ns)
  Data Path Delay:        9.557ns  (logic 3.625ns (37.929%)  route 5.932ns (62.071%))
  Logic Levels:           1  (OBUF=1)
  Output Delay:           2.000ns
  Clock Path Skew:        -5.105ns (DCD - SCD + CPR)
    Destination Clock Delay (DCD):    0.000ns = ( 20.000 - 20.000 )
    Source Clock Delay      (SCD):    5.105ns
    Clock Pessimism Removal (CPR):    0.000ns
  Clock Uncertainty:      0.035ns  ((TSJ^2 + TIJ^2)^1/2 + DJ) / 2 + PE
    Total System Jitter     (TSJ):    0.071ns
    Total Input Jitter      (TIJ):    0.000ns
    Discrete Jitter          (DJ):    0.000ns
    Phase Error              (PE):    0.000ns

    Location             Delay type                Incr(ns)  Path(ns)    Netlist Resource(s)
  -------------------------------------------------------------------    -------------------
                         (clock CLK rise edge)        0.000     0.000 r
    K17                                               0.000     0.000 r  PS_CLK_50M (IN)
                         net (fo=0)                   0.000     0.000    PS_CLK_50M
    K17                  IBUF (Prop_ibuf_I_O)         1.406     1.406 r  PS_CLK_50M_IBUF_inst/O
                         net (fo=1, routed)           2.160     3.566    PS_CLK_50M_IBUF
    BUFGCTRL_X0Y16       BUFG (Prop_bufg_I_O)         0.085     3.651 r  PS_CLK_50M_IBUF_BUFG_inst/O
                         net (fo=8838, routed)        1.454     5.105    x_uart/PS_CLK_50M_IBUF_BUFG
    SLICE_X87Y22         FDCE                                        r  x_uart/regs_q_reg[7][1]_lopt_replica/C
  -------------------------------------------------------------------    -------------------
    SLICE_X87Y22         FDCE (Prop_fdce_C_Q)         0.379     5.484 r  x_uart/regs_q_reg[7][1]_lopt_replica/Q
                         net (fo=1, routed)           5.932    11.416    lopt_1
    G14                  OBUF (Prop_obuf_I_O)         3.246    14.662 r  gpio_o_OBUF[1]_inst/O
                         net (fo=0)                   0.000    14.662    gpio_o[1]
    G14                                                         r  gpio_o[1] (OUT)
  -------------------------------------------------------------------    -------------------

                         (clock CLK rise edge)       20.000    20.000 r
                         clock pessimism              0.000    20.000
                         clock uncertainty           -0.035    19.965
                         output delay                -2.000    17.965
  -------------------------------------------------------------------
                         required time                         17.965
                         arrival time                         -14.662
  -------------------------------------------------------------------
                         slack                                  3.303
```

Figure 13: Max Delay Path

Serial Setup

We use PUTTY as terminal emulator here.

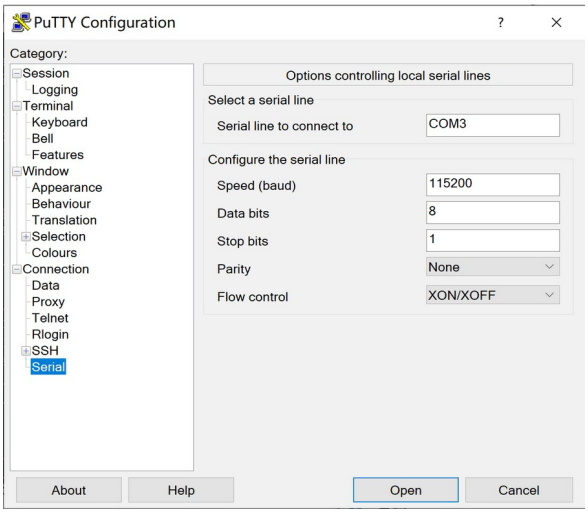

Figure 14: Serial Setup

The result

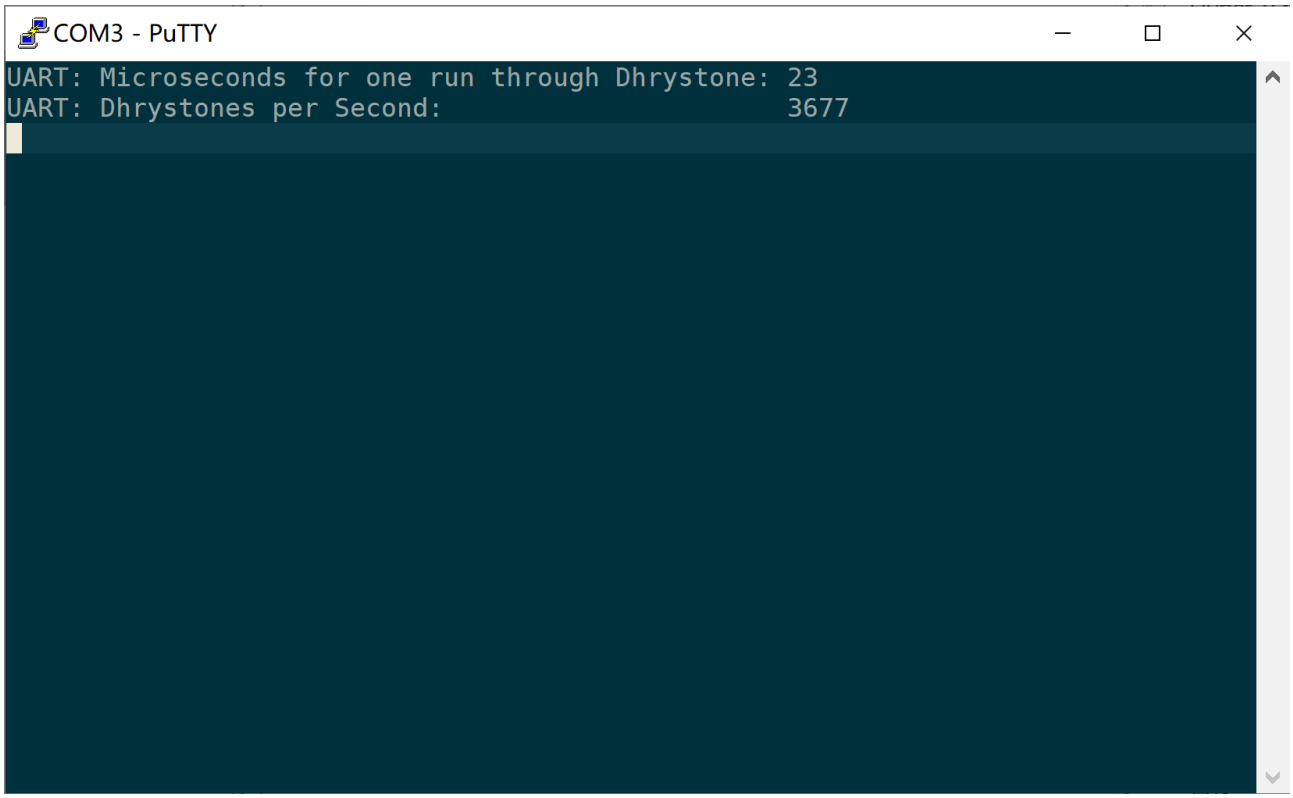

Figure 15: FPGA Result

The same as simulation.

5. Details of the Core

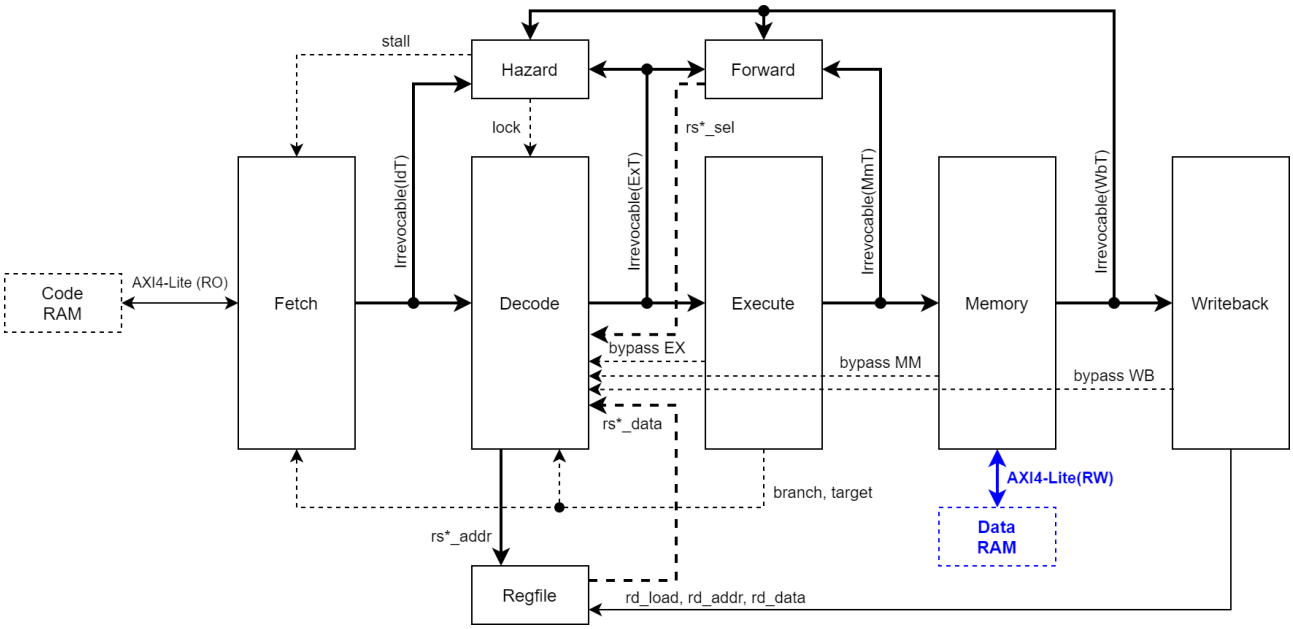

Figure 16: Recapture of the Diagram

5.1. Organization

Filename	Description
ALU.scala	Arithmetic/Logic Unit
Arbitrate.scala	Demux Memory Stage Access to Code, Data, MMIO
Axi.scala	Definition of AXI interface
Cpu.scala	Top Level of the Core
Decode.scala	Decode Stage
Execute.scala	Execute Stage
Fetch.scala	Fetch Stage
Forward.scala	Forward Unit
Hazard.scala	Hazard Detection Unit
Main.scala	Chisel Generator Application, generating FIRRTL/Verilog targets
Memory.scala	Memory Stage
Regfile.scala	Register File
rv32.scala	RISCV Constants, Internal interfaces and data structures
Writeback.scala	Writeback Stage

5.2. Fetch Stage

Fetch instructions from Code RAM through a simplified AXI4-Stream interface. The address is multiplexed through several sources: PC+4, trap handler, and branch target. The branch target is

passed from Execute Stage. When a taken branch is asserted, current states of Fetch and Decode must be dropped. These causes a two-cycle penalty.

5.3. Decode Stage

The Decode Stage leverages the Chisel feature, largely simplifies the codes. What's more, the readability of code becomes quite well.

Figure 17: Decode Code Samples

```
rv32.scala: 192
class CtrlT extends Bundle {
  val op = UInt(4.W)
  val fn = UInt(4.W)
  val br = UInt(3.W)
  val op1 = UInt(1.W)
  val op2 = UInt(3.W)
}
```

```
Decode.scala: 6
object DecCtrl {
  def apply(op: UInt, fn: UInt, br: UInt, op1: UInt, op2: UInt): CtrlT = {
    val ret = Wire(new CtrlT)
    ret.op := op(3, 0)
    ret.fn := fn(3, 0)
    ret.br := br(2, 0)
    ret.op1 := op1(0)
    ret.op2 := op2(2, 0)
    ret
  }
}
```

```
Decode.scala: 36
  val NONE  = DecCtrl(op_t.NONE,               fn_t.ANY,   br_t.NA,   op1_t.XX,  op2_t.XXX)
  val ADDI  = DecCtrl(op_t.INTEGER,            fn_t.ADD,   br_t.NA,   op1_t.RS1, op2_t.I_IMM)
  val SLTI  = DecCtrl(op_t.INTEGER,            fn_t.SLT,   br_t.NA,   op1_t.RS1, op2_t.I_IMM)
  val SLTIU = DecCtrl(op_t.INTEGER,            fn_t.SLTU,  br_t.NA,   op1_t.RS1, op2_t.I_IMM)
  val ANDI  = DecCtrl(op_t.INTEGER,            fn_t.AND,   br_t.NA,   op1_t.RS1, op2_t.I_IMM)
  val ORI   = DecCtrl(op_t.INTEGER,            fn_t.OR,    br_t.NA,   op1_t.RS1, op2_t.I_IMM)
  val XORI  = DecCtrl(op_t.INTEGER,            fn_t.XOR,   br_t.NA,   op1_t.RS1, op2_t.I_IMM)
  val SLLI  = DecCtrl(op_t.INTEGER,            fn_t.SLL,   br_t.NA,   op1_t.RS1, op2_t.I_IMM)
  val SRLI  = DecCtrl(op_t.INTEGER,            fn_t.SRL,   br_t.NA,   op1_t.RS1, op2_t.I_IMM)
  val SRAI  = DecCtrl(op_t.INTEGER,            fn_t.SRA,   br_t.NA,   op1_t.RS1, op2_t.I_IMM)
  val LUI   = DecCtrl(op_t.INTEGER,            fn_t.OP2,   br_t.NA,   op1_t.XX,  op2_t.U_IMM)
  val AUIPC = DecCtrl(op_t.INTEGER,            fn_t.ADD,   br_t.NA,   op1_t.PC,  op2_t.U_IMM)
  val ADD   = DecCtrl(op_t.INTEGER,            fn_t.ADD,   br_t.NA,   op1_t.RS1, op2_t.RS2)
  val SLT   = DecCtrl(op_t.INTEGER,            fn_t.SLT,   br_t.NA,   op1_t.RS1, op2_t.RS2)
  val SLTU  = DecCtrl(op_t.INTEGER,            fn_t.SLTU,  br_t.NA,   op1_t.RS1, op2_t.RS2)
  val AND   = DecCtrl(op_t.INTEGER,            fn_t.AND,   br_t.NA,   op1_t.RS1, op2_t.RS2)
  val OR    = DecCtrl(op_t.INTEGER,            fn_t.OR,    br_t.NA,   op1_t.RS1, op2_t.RS2)
  val XOR   = DecCtrl(op_t.INTEGER,            fn_t.XOR,   br_t.NA,   op1_t.RS1, op2_t.RS2)
  val SLL   = DecCtrl(op_t.INTEGER,            fn_t.SLL,   br_t.NA,   op1_t.RS1, op2_t.RS2)
  val SRL   = DecCtrl(op_t.INTEGER,            fn_t.SRL,   br_t.NA,   op1_t.RS1, op2_t.RS2)
  val SUB   = DecCtrl(op_t.INTEGER,            fn_t.SUB,   br_t.NA,   op1_t.RS1, op2_t.RS2)
  val SRA   = DecCtrl(op_t.INTEGER,            fn_t.SRA,   br_t.NA,   op1_t.RS1, op2_t.RS2)
  val JAL   = DecCtrl(op_t.JUMP,               fn_t.ADD,   br_t.JAL,  op1_t.PC,  op2_t.J_IMM)
  val JALR  = DecCtrl(op_t.JUMP,               fn_t.ADD,   br_t.JAL,  op1_t.RS1, op2_t.I_IMM)
  val BEQ   = DecCtrl(op_t.BRANCH,             fn_t.ADD,   br_t.BEQ,  op1_t.PC,  op2_t.B_IMM)
  val BNE   = DecCtrl(op_t.BRANCH,             fn_t.ADD,   br_t.BNE,  op1_t.PC,  op2_t.B_IMM)
  val BLT   = DecCtrl(op_t.BRANCH,             fn_t.ADD,   br_t.BLT,  op1_t.PC,  op2_t.B_IMM)
  val BLTU  = DecCtrl(op_t.BRANCH,             fn_t.ADD,   br_t.BLTU, op1_t.PC,  op2_t.B_IMM)
  val BGE   = DecCtrl(op_t.BRANCH,             fn_t.ADD,   br_t.BGE,  op1_t.PC,  op2_t.B_IMM)
  val BGEU  = DecCtrl(op_t.BRANCH,             fn_t.ADD,   br_t.BGEU, op1_t.PC,  op2_t.B_IMM)
  val LW    = DecCtrl(op_t.LOAD_WORD,          fn_t.ADD,   br_t.NA,   op1_t.RS1, op2_t.I_IMM)
  val LH    = DecCtrl(op_t.LOAD_HALF,          fn_t.ADD,   br_t.NA,   op1_t.RS1, op2_t.I_IMM)
  val LHU   = DecCtrl(op_t.LOAD_HALF_UNSIGNED, fn_t.ADD,   br_t.NA,   op1_t.RS1, op2_t.I_IMM)
  val LB    = DecCtrl(op_t.LOAD_BYTE,          fn_t.ADD,   br_t.NA,   op1_t.RS1, op2_t.I_IMM)
  val LBU   = DecCtrl(op_t.LOAD_BYTE_UNSIGNED, fn_t.ADD,   br_t.NA,   op1_t.RS1, op2_t.I_IMM)
  val SW    = DecCtrl(op_t.STORE_WORD,         fn_t.ADD,   br_t.NA,   op1_t.RS1, op2_t.S_IMM)
  val SH    = DecCtrl(op_t.STORE_HALF,         fn_t.ADD,   br_t.NA,   op1_t.RS1, op2_t.S_IMM)
  val SB    = DecCtrl(op_t.STORE_BYTE,         fn_t.ADD,   br_t.NA,   op1_t.RS1, op2_t.S_IMM)
```

```
Decode.scala: 93
```

```
ctrl := NONE
switch(ir.r_opcode) {
  is(opcode_t.OP_IMM) {
    switch(ir.r_funct3) {
      is(funct3_t.ADD) { ctrl := ADDI }
      is(funct3_t.SLL) { ctrl := SLLI }
      is(funct3_t.SLT) { ctrl := SLTI }
      is(funct3_t.SLTIU) { ctrl := SLTIU }
      is(funct3_t.XOR) { ctrl := XORI }
      is(funct3_t.SRL) { ctrl := Mux(ir.r_funct7(5), SRAI, SRLI) }
      is(funct3_t.OR ) { ctrl := ORI  }
      is(funct3_t.AND) { ctrl := ANDI }
    }
  }
  is(opcode_t.OP) {
    switch(ir.r_funct3) {
      is(funct3_t.ADD) { ctrl := Mux(ir.r_funct7(5), SUB, ADD) }
      is(funct3_t.SLL) { ctrl := SLL }
      is(funct3_t.SLT) { ctrl := SLT }
      is(funct3_t.SLTIU) { ctrl := SLTU }
      is(funct3_t.XOR) { ctrl := XOR }
      is(funct3_t.SRL) { ctrl := Mux(ir.r_funct7(5), SRA, SRL) }
      is(funct3_t.OR ) { ctrl := OR  }
      is(funct3_t.AND) { ctrl := AND }
    }
  }
  is(opcode_t.LUI)    { ctrl := LUI   }
  is(opcode_t.AUIPC)  { ctrl := AUIPC }
  is(opcode_t.JAL)    { ctrl := JAL   }
  is(opcode_t.JALR)   { ctrl := JALR  }
  is(opcode_t.BRANCH) {
    switch(ir.r_funct3) {
      is(funct3_t.BEQ)  { ctrl := BEQ }
      is(funct3_t.BNE)  { ctrl := BNE }
      is(funct3_t.BLT)  { ctrl := BLT }
      is(funct3_t.BLTU) { ctrl := BLTU }
      is(funct3_t.BGE)  { ctrl := BGE }
      is(funct3_t.BGEU) { ctrl := BGEU }
    }
  }
  is(opcode_t.LOAD) {
    switch(ir.r_funct3) {
      is(funct3_t.LW)   { ctrl := LW }
      is(funct3_t.LH)   { ctrl := LH }
      is(funct3_t.LHU)  { ctrl := LHU }
      is(funct3_t.LB)   { ctrl := LB }
      is(funct3_t.LBU)  { ctrl := LBU }
    }
  }
  is(opcode_t.STORE) {
    switch(ir.r_funct3) {
      is(funct3_t.SB)   { ctrl := SB }
      is(funct3_t.SH)   { ctrl := SH }
      is(funct3_t.SW)   { ctrl := SW }
    }
  }
}
```

5.4. Execute Stage

The Execute Stage controls the ALU to calculate arithmetic results for integer instructions, destination address for control transfer instructions (branches/jumps), and memory locations for load/store instructions.

5.5. Memory Stage

The memory stage handles memory accesses, or the load/store instructions. It has nothing do to with other instructions.

5.6. Writeback Stage

Writeback Stage just write the valid result to register file. Some instructions have no effect in this stage. For example, the branch instructions. Note that jump instructions (jal / jalr) may update register file. The they must be considered as forward sources.

5.7. Forward Unit

The forward unit forwards data that have not been written back to register file, while they are needed by the Decode Stage, by comparing RS1/RS2 of the Decode Stage and later Stages.

5.8. Hazard Detection

The Hazard Unit detects interlock conditions between instructions. It is now simply stall one cycle when seeing jump instructions emitted from the Decode stage.

6. Conclusions and Future Works

Previously we described our design. It is a classic single thread, 5-stage pipeline, Harvard architecture. The exception handling needs further effort to make it works. Also, the UART received path needs further tuning. And more benchmarks can be tried, like Coremark, Mediabench, etc.