CO-Processor Design in Pico-RV32

PE REPORT

Submitted By:
Akshay Godse
MT2019504

- **Introduction:**
  PicoRV32 is a CPU core that implements the RISC-V RV32IMC Instruction Set. It can be configured as RV32E, RV32I, RV32IC, RV32IM, or RV32IMC core, and optionally contains a built-in interrupt controller.
  More information can be found on
  https://github.com/cliffordwolf/picorv32#building-a-pure-rv32i-toolchain
  So here we designed a coprocessor which supports floating point instruction and is able to compute the fused multiply addition instruction.

- **Objective:**
  PicoRV32 should throw the instruction which is not supported by the CPU and generate necessary peripheral signals to carry out the operation.
  Instructions which have more than two operands should be sent in the pair of two operand registers so that we dont change any peripheral pins.

- **Design and Modifications:**
  PCPI:
  PicoRV32 Pico Co-Processor Interface (PCPI) can be used to implement non-branching instructions in external cores:

```
// Pico Co-Processor Interface (PCPI)
output reg        pcpi_valid,
output reg [31:0] pcpi_insn,
output     [31:0] pcpi_rs1,
output     [31:0] pcpi_rs2,
input             pcpi_wr,
input      [31:0] pcpi_rd,
input             pcpi_wait,
input             pcpi_ready,
```

  When an unsupported instruction is encountered and the PCPI feature is activated (see ENABLE_PCPI above), then pcpi_valid is asserted, the instruction word itself is output on pcpi_insn, the rs1 and rs2 fields are decoded and the values in those registers are output on pcpi_rs1 and pcpi_rs2.
  An external PCPI core can then decode the instruction, execute it, and assert pcpi_ready when execution of the instruction is finished. Optionally a result value can be written to pcpi_rd and pcpi_wr asserted.
  The PicoRV32 core will then decode the rd field of the instruction and write the value from pcpi_rd to the respective register. When no external PCPI core acknowledges the instruction within 16 clock cycles, then an illegal instruction exception is raised and the respective interrupt handler is called. A PCPI core that needs more than a couple of cycles to execute an instruction, should assert pcpi_wait as soon as the instruction has been decoded successfully and keep it asserted until it asserts pcpi_ready. This will prevent the PicoRV32 core from raising an illegal instruction exception.

OPCODE:

The OPCODES for unsupported instruction is taken from this documentation. The link for this document for reference: *https://github.com/akshaygodse13/PE_Coprocessor/blob/main/riscv-spec-v2.1.pdf*

MODIFICATIONS IN PICORV32:

There is already a Co-Processor for MUL instruction hence we used it as the reference.

Addition of an Enable:

```
module picorv32 #(
    parameter [ 0:0] ENABLE_COUNTERS = 1,
    parameter [ 0:0] ENABLE_COUNTERS64 = 1,
    parameter [ 0:0] ENABLE_REGS_16_31 = 1,
    parameter [ 0:0] ENABLE_REGS_DUALPORT = 1,
    parameter [ 0:0] LATCHED_MEM_RDATA = 0,
    parameter [ 0:0] TWO_STAGE_SHIFT = 1,
    parameter [ 0:0] BARREL_SHIFTER = 0,
    parameter [ 0:0] TWO_CYCLE_COMPARE = 0,
    parameter [ 0:0] TWO_CYCLE_ALU = 0,
    parameter [ 0:0] COMPRESSED_ISA = 0,
    parameter [ 0:0] CATCH_MISALIGN = 1,
    parameter [ 0:0] CATCH_ILLINSN = 1,
    parameter [ 0:0] ENABLE_PCPI = 1,
    parameter [ 0:0] ENABLE_MUL = 0,
    parameter [ 0:0] ENABLE_FAST_MUL = 0,
    parameter [ 0:0] ENABLE_FADDS = 1,
    parameter [ 0:0] ENABLE_FMADDS = 1,
    parameter [ 0:0] ENABLE_DIV = 0,
    parameter [ 0:0] ENABLE_IRQ = 0,
    parameter [ 0:0] ENABLE_IRQ_QREGS = 1,
    parameter [ 0:0] ENABLE_IRQ_TIMER = 1,
    parameter [ 0:0] ENABLE_TRACE = 0,
    parameter [ 0:0] REGS_INIT_ZERO = 0,
    parameter [31:0] MASKED_IRQ = 32'h 0000_0000,
    parameter [31:0] LATCHED_IRQ = 32'h ffff_ffff,
    parameter [31:0] PROGADDR_RESET = 32'h 0000_0000,
    parameter [31:0] PROGADDR_IRQ = 32'h 0000_0010,
    parameter [31:0] STACKADDR = 32'h ffff_ffff
) (
```

Addition of a Muxing logic which connects internal core to Peripheral:

```
    // Internal PCPI Cores

    wire        pcpi_mul_wr;
    wire [31:0] pcpi_mul_rd;
    wire        pcpi_mul_wait;
    wire        pcpi_mul_ready;

    wire        pcpi_div_wr;
    wire [31:0] pcpi_div_rd;
    wire        pcpi_div_wait;
    wire        pcpi_div_ready;
////////////////////////////////////////////////////////
    wire        pcpi_fadds_wr;
    wire [31:0] pcpi_fadds_rd;
    wire        pcpi_fadds_wait;
    wire        pcpi_fadds_ready;

    wire        pcpi_fmadds_wr;
    wire [31:0] pcpi_fmadds_rd;
    wire        pcpi_fmadds_wait;
    wire        pcpi_fmadds_ready;
////////////////////////////////////////////////////////
```

All these wires will get connected to Peripheral pins when enable is on.

```
    generate if (ENABLE_FMADDS) begin
        picorv32_pcpi_fmadds pcpi_fmadds (
            .clk       (clk            ),
            .resetn    (resetn         ),
            .pcpi_valid(pcpi_valid     ),
            .pcpi_insn (pcpi_insn      ),
            .pcpi_rs1  (pcpi_rs1      ),
            .pcpi_rs2  (pcpi_rs2      ),
            .pcpi_wr   (pcpi_fmadds_wr    ),
            .pcpi_rd   (pcpi_fmadds_rd    ),
            .pcpi_wait (pcpi_fmadds_wait  ),
            .pcpi_ready(pcpi_fmadds_ready )
        );
    end else begin
        assign pcpi_fmadds_wr = 0;
        assign pcpi_fmadds_rd = 32'bx;
        assign pcpi_fmadds_wait = 0;
        assign pcpi_fmadds_ready = 0;
    end endgenerate


    always @* begin
        pcpi_int_wr = 0;
        pcpi_int_rd = 32'bx;
        pcpi_int_wait  = |{ENABLE_PCPI && pcpi_wait,  (ENABLE_MUL || ENABLE_FAST_MUL) && pcpi_mul_wait,
                           ENABLE_DIV && pcpi_div_wait,
                           ENABLE_FADDS && pcpi_fadds_wait, ENABLE_FMADDS && pcpi_fmadds_wait};
        pcpi_int_ready = |{ENABLE_PCPI && pcpi_ready, (ENABLE_MUL || ENABLE_FAST_MUL) && pcpi_mul_ready,
                           ENABLE_DIV && pcpi_div_ready,
                           ENABLE_FADDS && pcpi_fadds_ready, ENABLE_FMADDS && pcpi_fmadds_ready};

        (* parallel_case *)
        case (1'b1)
            ENABLE_PCPI && pcpi_ready: begin
                pcpi_int_wr = ENABLE_PCPI ? pcpi_wr : 0;
                pcpi_int_rd = ENABLE_PCPI ? pcpi_rd : 0;
            end
            (ENABLE_MUL || ENABLE_FAST_MUL) && pcpi_mul_ready: begin
                pcpi_int_wr = pcpi_mul_wr;
                pcpi_int_rd = pcpi_mul_rd;
            end
            ENABLE_DIV && pcpi_div_ready: begin
                pcpi_int_wr = pcpi_div_wr;
                pcpi_int_rd = pcpi_div_rd;
            end
            ENABLE_FADDS && pcpi_fadds_ready: begin
                pcpi_int_wr = pcpi_fadds_wr;
                pcpi_int_rd = pcpi_fadds_rd;
            end
            ENABLE_FMADDS && pcpi_fmadds_ready: begin
                pcpi_int_wr = pcpi_fmadds_wr;
                pcpi_int_rd = pcpi_fmadds_rd;
            end
        endcase
    end
```

Addition of FSM for sending data in pairs for instructions having 3 operands. For example fused multiply will have 3 operands. So first two operands are sent in one clk cycle and in the next one the remaining.
Operand_3_valid will be raised when an instruction will have more than 3 operands. When it is low the CPU will work normally.

```
// logic for 3 operand fmadds
wire operand_3_valid;
wire [4:0] decoded_rs3;
assign operand_3_valid = (pcpi_insn[6:0]==7'b1000011) ? 1:0;
assign decoded_rs3 = pcpi_insn[31:27];
wire [31:0] reg_op3_1, reg_op3_2;
reg [1:0] st=0;

assign  reg_op3_1=cpuregs[decoded_rs3];
assign  reg_op3_2=0;
assign pcpi_rs1[31:0] =(st==1)?reg_op3_1[31:0]:reg_op1[31:0];
assign pcpi_rs2[31:0] =(st==1)?reg_op3_2[31:0]:reg_op2[31:0];

always@(posedge clk)
begin
   if(operand_3_valid && st==0 && pcpi_valid==1)
    begin
    st<=1;
    end
else if(operand_3_valid && st==1)
    begin
    st<=2;
    end
else if(operand_3_valid && st==2)
    begin
    if(pcpi_int_ready==1)
    st<=0;
    else
    st<=2;
    end
end
```

Now a separate module is written for the operation.
This module will generate signals as mentioned in the PCPI interface.

Operation:
For the operation to be performed in the coprocessor we just instantiate the module and pass the operands to the module.
https://github.com/akshaygodse13/PE_Coprocessor/tree/main/Fused_multiply_add

TESTBENCH:

So we wrote a program which will be read by CPU

```
initial begin
  memory[0] = 32'h 00001217; //        lui      x4,1
  memory[1] = 32'h 00001197; //        lui      x3,1
  memory[2] = 32'h 003202d3; //        fadds x5,x3,x4
  memory[3] = 32'h 28418543; //        fmadds      x10,x3,x4,x5
  memory[4] = 32'h 283505C3;//         fmadds    x11,x10,x3,x5
end
```

The purpose of writing this code was to check whether the signals are generated properly and see the output. We forcefully generated dependency in the program. So the output of one instruction is used as the operand in the next instruction to verify whether it was stored correctly in the previous instruction.

The output can be seen in the following link:

https://www.edaplayground.com/x/UeWm

To see waveforms add signals under uut.