About

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. It is written in Verilog.

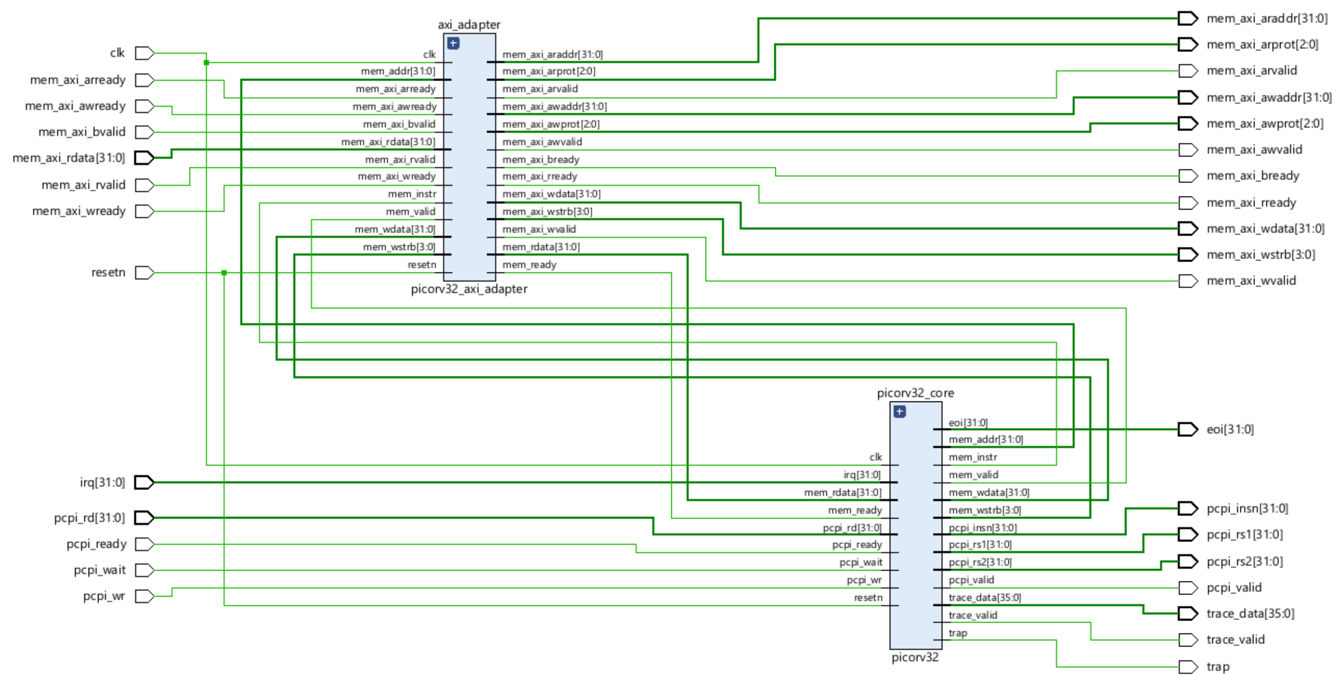
PicoRV32 is free and open hardware licensed under the ISC license.

This CPU is meant to be used as auxiliary processor in FPGA designs and ASICs. Due to its **high fmax** it can be integrated in most existing designs without crossing clock domains. When operated on a lower frequency, it will have a lot of timing slack and thus can be added to a design without compromising timing closure.

The Core exists in 3 variations: -

1. picorv32: a simple native memory interface, that is easy to use in simple environments for simple applications.
2. picorv32_axi: provides an AXI-4 Lite Master interface that can easily be integrated with existing systems that are already using the Advance eXtensible Interface(AXI) standard.
3. Picorv32_wb: provides a wishbone master interface.

Apart from the main core, there exists a separate core picorv32_axi_adapter to bridge between the native memory interface and AXI4. This core can be used to create custom cores that include one or more PicoRV32 cores together with local RAM, ROM, and memory-mapped peripherals, communicating with each other using the native interface, and communicating with the outside world via AXI4.



The Picorv32 is optimized for size and f_{max} and not for performance. However, this can be changed by making necessary changes to the module parameters.

```
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 = 0,
parameter [ 0:0] ENABLE_MUL = 0,
parameter [ 0:0] ENABLE_FAST_MUL = 0,
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
```

IRQ Handling

The IRQ handling features in PicoRV32 do not follow the RISC-V Privileged ISA specification. Instead a small set of very simple custom instructions is used to implement IRQ handling with minimal hardware overhead.

The following custom instructions are only supported when IRQs are enabled via the ENABLE_IRQ parameter.
By Default ENABLE_IRQ=0

The PicoRV32 core has a built-in interrupt controller with 32 interrupt inputs. An interrupt can be triggered by asserting the corresponding bit in the 'irq' input of the core.

When the interrupt handler is started, the 'eoi' End Of Interrupt (EOI) signals for the handled interrupts go high. The eoi signals go low again when the interrupt handler returns.

The IRQs 0-2 can be triggered internally by the following built-in interrupt sources:

- IRQ 0: Timer Interrupt
- IRQ 1: EBREAK/ECALL or Illegal Instruction
- IRQ 2: Bus Error (Unaligned Memory Access)

This interrupts can also be triggered by external sources, like the PCPI.

The core has 4 additional 32-bit registers q0, q1, q2 and q3 that are used for IRQ handling.
When the IRQ handler is called, the register q0 contains the return address and q1 contains a bitmask of all IRQs to be handled.
Registers q2 and q3 are uninitialized and can be used as temporary storage when saving/restoring register values in the IRQ handler.

getq rd, qs
This instruction copies the value from a q-register to a general-purpose register.
Ex: getq x5, q2

setq qd, rs
This instruction copies the value from a general-purpose register to a q-register.
Ex: setq q2, x5

retirq
Return from interrupt. This instruction copies the value from q0 to the program counter and re-enables interrupts.
Ex: retirq

maskirq
The "IRQ Mask" register contains a bitmask of masked (disabled) interrupts. This instruction writes a new value to the irq mask register and reads the old value.
Ex: maskirq x1, x2

waitirq
Pause execution until an interrupt becomes pending. The bitmask of pending IRQs is written to rd.
Ex: waitirq x1

timer
Reset the timer counter to a new value. The counter counts down clock cycles and triggers the timer interrupt when transitioning from 1 to 0. Setting the counter to zero disables the timer. The old value of the counter is written to rd.
Ex: timer x1, x2

The processor starts with all interrupts disabled. An illegal instruction or bus error while the illegal instruction or bus error interrupt is disabled will cause the processor to halt.

PicoRV32 Native Memory Interface

The native memory interface of PicoRV32 is a simple valid-ready interface that can run one memory transfer at a time.

```
output reg        mem_valid,
output reg        mem_instr,
input             mem_ready,

output reg [31:0] mem_addr,
output reg [31:0] mem_wdata,
output reg [ 3:0] mem_wstrb,
input      [31:0] mem_rdata,
```

mem_rdata	I/P	32	stores the instruction read from memory
mem_ready	I/P	1	indicates that memory is ready to perform a transaction
mem_valid	O/P	1	indicates that memory access is valid
mem_instr	O/P	1	indicates whether the current memory transaction is an instruction fetch or a data access
mem_addr	O/P	32	contains the address of the current memory transaction
mem_wdata	O/P	32	contains data to be written in memory in a store operation
mem_wstrb	O/P	4	a write strobe indicating which bytes of 32-bit word to be written to memory

wstrb = 0000	no write
wstrb = 1111	write 32 bits
wstrb = 1100	write upper 16 bits
wstrb = 0011	write lower 16 bits
wstrb = 1000	write 1st byte [MSB]
wstrb = 0100	write 2nd byte
wstrb = 0010	write 3rd byte
wstrb = 0001	write 4th byte [LSB]

The core initiates a memory transfer by asserting mem_valid.
If the memory transfer is an instruction fetch, the core asserts mem_instr.
All core outputs are stable over the mem_valid period.

Read Transfer

In a read transfer mem_wstrb has the value 0 and mem_wdata is unused.
The memory reads the address mem_addr and makes the read value available on mem_rdata in the cycle mem_ready is high.
There is no need for an external wait cycle. The memory read can be implemented asynchronously with mem_ready going high in the same cycle as mem_valid, or mem_ready being tied to constant 1.

Write Transfer
In a write transfer mem_wstrb != 0 and mem_rdata is unused.
The memory write the data at mem_wdata to the address mem_addr and acknowledges the transfer by asserting mem_ready.
There is no need for an external wait cycle. The memory can acknowledge the write immediately with mem_ready going high in the same cycle as mem_valid, or mem_ready being tied to constant 1.

line 378 & 379:
```
wire mem_done = [resetn] &&
[(mem_xfer && |mem_state && (mem_do_rinst || mem_do_rdata || mem_do_wdata)) ||
(&mem_state && mem_do_rinst)] &&
[(!mem_la_firstword || (~&mem_rdata_latched[1:0] && mem_xfer))];
// |mem_state is bit wise OR of mem_state bits, TRUE when any one the two bits are
1(TRUE)
// &mem_state is bit wise AND of mem_state bits, TRUE only when mem_state = 2'b11
i.e. when both bits are TRUE
```

line 399:
```
last_mem_valid <= mem_valid && !mem_ready;
```

line 405 to 429:
```
case (mem_wordsize)
            0: begin
                mem_la_wdata = reg_op2;
                mem_la_wstrb = 4'b1111; // write 32 bits
                mem_rdata_word = mem_rdata;
            end
            1: begin
                mem_la_wdata = {2{reg_op2[15:0]}};
            // replicates lower 16-bit to the upper 16-bit to form complete 32-bit
                mem_la_wstrb = reg_op1[1] ? 4'b1100 : 4'b0011;
                case (reg_op1[1])
                    1'b0: mem_rdata_word = {16'b0, mem_rdata[15: 0]};
                    1'b1: mem_rdata_word = {16'b0, mem_rdata[31:16]};
                endcase
            end
            2: begin
                mem_la_wdata = {4{reg_op2[7:0]}};
                mem_la_wstrb = 4'b0001 << reg_op1[1:0];
                case (reg_op1[1:0])
                    2'b00: mem_rdata_word = {24'b0, mem_rdata[ 7: 0]};
                    2'b01: mem_rdata_word = {24'b0, mem_rdata[15: 8]};
                    2'b10: mem_rdata_word = {24'b0, mem_rdata[23:16]};
                    2'b11: mem_rdata_word = {24'b0, mem_rdata[31:24]};
                endcase
            end
        endcase
```

Memory Interface FSM

line 583 to 638:

```
case (mem_state)

      0: begin
           if (mem_do_prefetch || mem_do_rinst || mem_do_rdata) begin
                mem_valid <= !mem_la_use_prefetched_high_word;
                mem_instr <= mem_do_prefetch || mem_do_rinst;
                mem_wstrb <= 0;
                mem_state <= 1;
           end
           if (mem_do_wdata) begin
                mem_valid <= 1;
                mem_instr <= 0;
                mem_state <= 2;
          end
        end


       1: begin
           `assert(mem_wstrb == 0);
           `assert(mem_do_prefetch || mem_do_rinst || mem_do_rdata);
           `assert(mem_valid == !mem_la_use_prefetched_high_word);
           `assert(mem_instr == (mem_do_prefetch || mem_do_rinst));
           if (mem_xfer) begin
                if (COMPRESSED_ISA && mem_la_read) begin
                     mem_valid <= 1;
                     mem_la_secondword <= 1;
                     if (!mem_la_use_prefetched_high_word)
                          mem_16bit_buffer <= mem_rdata[31:16];
                     end
                     else begin
                          mem_valid <= 0;
                          mem_la_secondword <= 0;
                          if (COMPRESSED_ISA && !mem_do_rdata) begin
                              if (~&mem_rdata[1:0] || mem_la_secondword) begin
                                  mem_16bit_buffer <= mem_rdata[31:16];
                                  prefetched_high_word <= 1;
                              end else begin
                                  prefetched_high_word <= 0;
                              end
                          end
                      mem_state <= mem_do_rinst || mem_do_rdata ? 0 : 3;
                      end
                end
            end
         2: begin
                `assert(mem_wstrb != 0);
                `assert(mem_do_wdata);
                if (mem_xfer) begin
                     mem_valid <= 0;
                     mem_state <= 0;
                end
           end


         3: begin
                `assert(mem_wstrb == 0);
                `assert(mem_do_prefetch);
                if (mem_do_rinst) begin
                     mem_state <= 0;
                end
            end
        endcase
```

State 0
If Read operation (i.e. mem_do_prefetch OR mem_do_rdata OR mem_do_inst)
then, mem_inst = mem_do_prefetch OR mem_do_rinst
mem_wstrb = 0 (no write)
go to mem_state = 1

If Write operation (i.e. mem_do_wdata)
then, mem_valid = 1
mem_inst = 0
go to mem_state = 2

State 1
prefetching of high word takes place in this state
If Memory Transfer operation (i.e. mem_xfer)

State 2
assert mem_wstrb = 4'b1111 (write 32 bits)
assert mem_do_wdata
go to state 0 if memory transfer operation (mem_xfer = 1)

State 3
assert mem_wstrb = 0 (no write)
assert mem_do_prefetch = 1
go to state 0 if mem_do_rinst = 1

Architecture

Main State Machine

The main state machine has 8 states, each state signified by a unique 8-bit code (hot-one encoding)

```
    localparam cpu_state_trap   = 8'b10000000; //Synchronous interrupt for illegal
    instruction [hex=80]
    localparam cpu_state_fetch  = 8'b01000000; // instruction fetch from instruction
    decoder for scheduling and execution [hex=40]
    localparam cpu_state_ld_rs1 = 8'b00100000; // dual port register file for smaller
    core [hex=20]
    localparam cpu_state_ld_rs2 = 8'b00010000; // single port register file for
    better performance [hex=10]
    localparam cpu_state_exec   = 8'b00001000; // for instructions other than
    Shift/Load/Store [hex=08]
    localparam cpu_state_shift  = 8'b00000100; // for Shift instructions [hex=04]
    localparam cpu_state_stmem  = 8'b00000010; // for Store instructions [hex=02]
    localparam cpu_state_ldmem  = 8'b00000001; // for Load instructions [hex=01]
```

ALU Operations in the Main State machine: -

1. Addition, Subtraction
2. Equality
3. Less Than Signed/Unsigned
4. Logical Left Shift
5. Arithmetic Right Shift

```
always @* begin //SINGLE CYCLE ALU Operations
     alu_add_sub = instr_sub ? reg_op1 - reg_op2 : reg_op1 + reg_op2; //ADD_SUB
     alu_eq = reg_op1 == reg_op2; // COMPARE EQUALITY
     alu_lts = $signed(reg_op1) < $signed(reg_op2); //SIGNED COMPARE LESS THAN
     alu_ltu = reg_op1 < reg_op2; // UNSIGNED COMPARE LESS THAN
     alu_shl = reg_op1 << reg_op2[4:0]; //LOGICAL LEFT SHIFT (w.k.t only applicable
     to unsigned)
     alu_shr = $signed({instr_sra || instr_srai ? reg_op1[31] : 1'b0, reg_op1}) >>>
     reg_op2[4:0]; // ARITH RIGHT SHIFT (w.k.t only applicable to signed)
end
```

TWO_CYCLE_ALU adds an additional Flip-Flop to improve timing(synch) at the cost of an additional clock cycle.

By Default, TWO_CYCLE_ALU = 0

For a given instruction from the instruction decoder, the output from the ALU is stored in alu_out_0 or alu_out.

Instructions from the instruction decoder are separated into set of two.

```
//INSTRUCTION SET 1
alu_out_0 = 'bx;
(* parallel_case, full_case *)
 case (1'b1)
      instr_beq: //branch if equal to
          alu_out_0 = alu_eq;
      instr_bne: //branch if not equal to
          alu_out_0 = !alu_eq;
      instr_bge: //branch if greater or equal
          alu_out_0 = !alu_lts;
      instr_bgeu: //branch if greater or equal (unsigned)
          alu_out_0 = !alu_ltu;
      is_slti_blt_slt && (!TWO_CYCLE_COMPARE ||
      !{instr_beq,instr_bne,instr_bge,instr_bgeu}): //less than signed
          alu_out_0 = alu_lts;
      is_sltiu_bltu_sltu && (!TWO_CYCLE_COMPARE ||
      !{instr_beq,instr_bne,instr_bge,instr_bgeu}): //less than unsigned
          alu_out_0 = alu_ltu;
endcase
```

```
// INSTRUCTION SET 2
alu_out = 'bx;
(* parallel_case, full_case *)
 case (1'b1)
      is_lui_auipc_jal_jalr_addi_add_sub:
            alu_out = alu_add_sub;
      is_compare:
            alu_out = alu_out_0;
      instr_xori || instr_xor: //xor operation
            alu_out = reg_op1 ^ reg_op2;
      instr_ori || instr_or: //or operation
            alu_out = reg_op1 | reg_op2;
      instr_andi || instr_and: //and operation
            alu_out = reg_op1 & reg_op2;
      BARREL_SHIFTER && (instr_sll || instr_slli): //not active currently because
      BARREL_SHIFTER=0
            alu_out = alu_shl;
      BARREL_SHIFTER && (instr_srl || instr_srli || instr_sra || instr_srai): //not
      active currently because BARREL_SHIFTER=0
            alu_out = alu_shr;
  endcase
```

count_cycle

count_cycle can be very useful to check how many clock cycles have passed. resets to 0 when resetn=0

```
if (ENABLE_COUNTERS) begin
            count_cycle <= resetn ? count_cycle + 1 : 0;
           if (!ENABLE_COUNTERS64) count_cycle[63:32] <= 0;

        end
```

count_instr

keeps the count of instructions

```
if (ENABLE_COUNTERS) begin
                        count_instr <= count_instr + 1;
                        if (!ENABLE_COUNTERS64) count_instr[63:32] <= 0;
                    end
// keeps the count of instructions
```

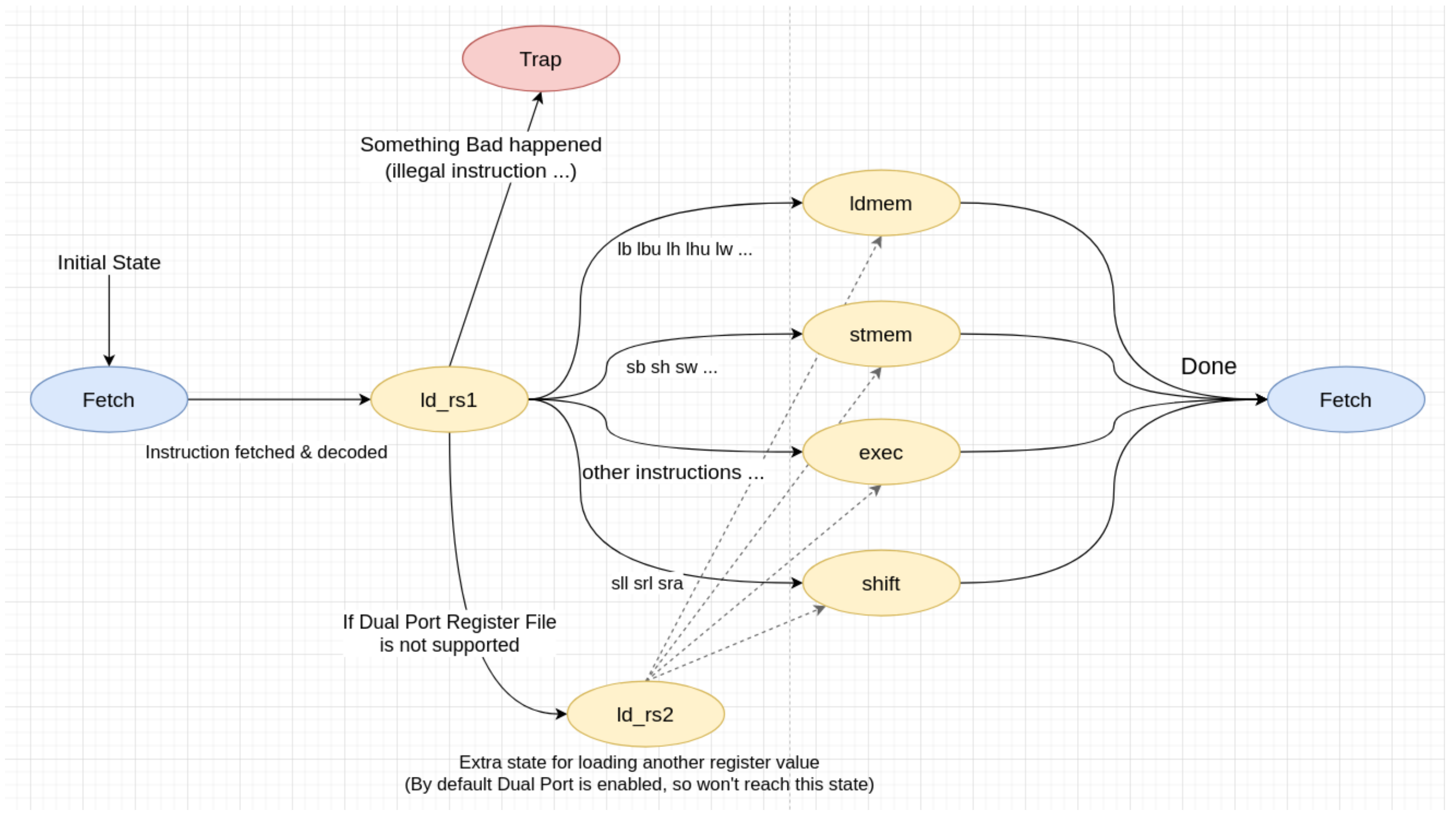
resetn

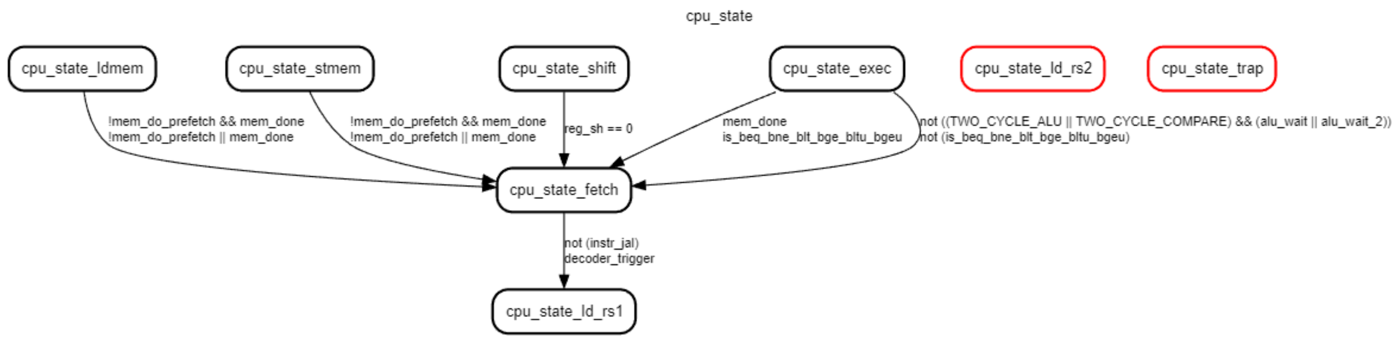
Active Low signal.

```
if (!resetn) begin //everything reseting to the given default values when resetn = 0
      reg_pc <= PROGADDR_RESET;
      reg_next_pc <= PROGADDR_RESET;
      if (ENABLE_COUNTERS)
          count_instr <= 0;
      latched_store <= 0;
      latched_stalu <= 0;
      latched_branch <= 0;
      latched_trace <= 0;
      latched_is_lu <= 0;
      latched_is_lh <= 0;
      latched_is_lb <= 0;
      pcpi_valid <= 0;
      pcpi_timeout <= 0;
      irq_active <= 0;
      irq_delay <= 0;
      irq_mask <= ~0;
      next_irq_pending = 0;
      irq_state <= 0;
      eoi <= 0;
      timer <= 0;
      if (~STACKADDR) begin
          latched_store <= 1;
          latched_rd <= 2;
          reg_out <= STACKADDR;
      end
      cpu_state <= cpu_state_fetch;
end
```

When resetn = 0, the state is set to FETCH. This implies that the default state of the processor is “FETCH”.

Main State Machine FSM


Trap
Something Bad happened
(illegal instruction ...)
Initial State
Fetch
Instruction fetched & decoded
ld_rs1
ldmem
lb lbu lh lhu lw ...
stmem
sb sh sw ...
exec
other instructions ...
shift
sll srl sra
Done
Fetch
If Dual Port Register File
is not supported
ld_rs2
Extra state for loading another register value
(By default Dual Port is enabled, so won't reach this state)


cpu_state
cpu_state_ldmem
cpu_state_stmem
cpu_state_shift
cpu_state_exec
cpu_state_ld_rs2
cpu_state_trap
!mem_do_prefetch && mem_done
!mem_do_prefetch || mem_done
!mem_do_prefetch && mem_done
!mem_do_prefetch || mem_done
reg_sh == 0
mem_done
is_beq_bne_blt_bge_bltu_bgeu
not ((TWO_CYCLE_ALU || TWO_CYCLE_COMPARE) && (alu_wait || alu_wait_2))
not (is_beq_bne_blt_bge_bltu_bgeu)
cpu_state_fetch
not (instr_jal)
decoder_trigger
cpu_state_ld_rs1

State Descriptions

1. **Trap**

   Unwanted state. Illegal instruction encountered.

```
cpu_state_trap: begin
          trap <= 1;
      end
```

2. **Fetch**

```
mem_do_rinst <= !decoder_trigger && !do_waitirq;
// instruction is read from the memory only when decoder_trigger=0 &&
do_waitirq=0, i.e. instruction is read only when no decoding or irq wait
mem_wordsize <= 0;
```

state change FETCH --> LD_RS1 is signaled when decoder_trigger is activated

```
if (decoder_trigger) begin
      if (instr_jal) begin

            mem_do_rinst <= 1;
            reg_next_pc <= current_pc + decoded_imm_j;
            latched_branch <= 1;
         end
      mem_do_rinst <= 0;
      mem_do_prefetch <= !instr_jalr && !instr_retirq; //mem prefetch operation
      except for retirq and jalr(jump and link register) instructions
      cpu_state <= cpu_state_ld_rs1; //signals state change from fetch to ld_rs1
  end
```

3. **LD_RS1**

If DUAL_PORT

If Asynch interrupt, FETCH state to retry, else, TRAP

If not DUAL_PORT

LD_RS2 state

```
if (ENABLE_REGS_DUALPORT) begin
      if (CATCH_ILLINSN && (pcpi_timeout || instr_ecall_ebreak)) begin

            if (ENABLE_IRQ && !irq_mask[irq_ebreak] && !irq_active) begin
                  next_irq_pending[irq_ebreak] = 1;
                  cpu_state <= cpu_state_fetch;
                  //asynch interrupt, retry to get correct instruction from fetch
               end
            else
                cpu_state <= cpu_state_trap; // failed, trap state
            end
      end
else begin
      cpu_state <= cpu_state_ld_rs2;
      //if dual_port not supported go to LD_RS2 state
  end
```

State transition from LD_RS1 according to current instruction from FETCH read from instruction decoder.

```
is_rdcycle_rdcycleh_rdinstr_rdinstrh --> FETCH

is_lui_auipc_jal --> EXEC

instr_getq --> FETCH

instr_setq --> FETCH

instr_retirq --> FETCH

instr_maskirq --> FETCH

instr_timer --> FETCH

is_lb_lh_lw_lbu_lhu --> LDMEM

is_slli_srli_srai --> SHIFT

is_jalr_addi_slti_sltiu_xori_ori_andi --> EXEC

is_sb_sh_sw --> STMEM

is_sll_srl_sra --> SHIFT

** by default state goes to EXEC if dual-port is supported

** if no dual port support by default from LD_RS1 state goes to LD_RS2
```

4. **EXEC**

in EXEC state latched_store and latched_brach keeps the ALU value computed at line #1254 in alu_out_0.

```
if (is_beq_bne_blt_bge_bltu_bgeu) begin
      latched_rd <= 0;
      latched_store <= TWO_CYCLE_COMPARE ? alu_out_0_q : alu_out_0;
      latched_branch <= TWO_CYCLE_COMPARE ? alu_out_0_q : alu_out_0;
      if (mem_done)
            cpu_state <= cpu_state_fetch;
            // on completion of operation of EXEC state state moves to FETCH state
    end
```

5. SHIFT

reg_sh keeps the count how many shifts have to be done.

When reg_sh =0, state moves to FETCH state

```
if (reg_sh == 0) begin
      reg_out <= reg_op1;
      mem_do_rinst <= mem_do_prefetch;
      cpu_state <= cpu_state_fetch;
 end


case (1'b1) //shift operations
      instr_slli || instr_sll: reg_op1 <= reg_op1 << 1;
      instr_srli || instr_srl: reg_op1 <= reg_op1 >> 1;
      instr_srai || instr_sra: reg_op1 <= $signed(reg_op1) >>> 1;
endcase
reg_sh <= reg_sh - 1;
```

6. ST_MEM

no other memory operation for ST_MEM state operation

when no other memory operation or ST_MEM operation state goes to FETCH state

```
if (!mem_do_prefetch || mem_done) begin
      if (!mem_do_wdata) begin
        (* parallel_case, full_case *)
            case (1'b1)
                  instr_sb: mem_wordsize <= 2;
                  instr_sh: mem_wordsize <= 1;
                  instr_sw: mem_wordsize <= 0;
            endcase
            reg_op1 <= reg_op1 + decoded_imm;
            set_mem_do_wdata = 1;
       end
 if (!mem_do_prefetch && mem_done) begin
      cpu_state <= cpu_state_fetch;
      decoder_trigger <= 1;
      decoder_pseudo_trigger <= 1;
 end
```

7. LD_MEM

no other memory operation for LD_MEM state operation

when no other memory operation or LD_MEM operation state goes to FETCH state

```
if (!mem_do_prefetch || mem_done) begin
      if (!mem_do_rdata) begin
            (* parallel_case, full_case *)
            case (1'b1)
                  instr_lb || instr_lbu: mem_wordsize <= 2;
                  instr_lh || instr_lhu: mem_wordsize <= 1;
                  instr_lw: mem_wordsize <= 0;
            endcase
           latched_is_lu <= is_lbu_lhu_lw;
           latched_is_lh <= instr_lh;
           latched_is_lb <= instr_lb;
            reg_op1 <= reg_op1 + decoded_imm;
            set_mem_do_rdata = 1;

if (!mem_do_prefetch && mem_done) begin
 (* parallel_case, full_case *)
      case (1'b1)
            latched_is_lu: reg_out <= mem_rdata_word;
            latched_is_lh: reg_out <= $signed(mem_rdata_word[15:0]);
            latched_is_lb: reg_out <= $signed(mem_rdata_word[7:0]);
      endcase
      decoder_trigger <= 1;
      decoder_pseudo_trigger <= 1;
      cpu_state <= cpu_state_fetch;
 end
```

INSTRUCTION DECODER:-

Initially all the instructions are declared using reg kind of data type and followed by, there is also declaring of registers for storing the addresses of rs1, rs2 (source registers) and rd (destination register) which of are 5-bit length and immediate register of 32-bit size.

Variables like **decoder_trigger** which is dependent on **mem_do_rinst** (which indicates if a read instruction operation is requested) and mem_done, will get activated when both are high.

There is a D flip-flop and the input is decoder_trigger the output is decoder_trigger_q ,it means decoder_trigger_q will get the last state decoder_trigger.

In the next step same set of instruction types are merged together because of their common kind of decoding pattern like

1. U, J type with "is_lui_auipc_jal"
2. Load type of instructions in "is_lb_lh_lw_lbu_lhu";
3. Store type of instructions in "is_sb_sh_sw "
4. R-type shifting are merged in is_sll_srl_sra
5. Whole R-type are merged in "is_alu_reg_imm , is_alu_reg_reg"

So later we would see in other stage, what will be the advantage of clubbing them together like for storing the decoded source and destination addresses because of common type of ISA.

```
// Instruction Decoder

reg instr_lui, instr_auipc, instr_jal, instr_jalr;
reg instr_beq, instr_bne, instr_blt, instr_bge, instr_bltu, instr_bgeu;
reg instr_lb,instr_lh,instr_lw,instr_lbu,instr_lhu,instr_sb,instr_sh,instr_sw;

reg instr_addi, instr_slti, instr_sltiu, instr_xori, instr_ori, instr_andi, instr_slli,
instr_srli, instr_srai;

reg instr_add, instr_sub, instr_sll, instr_slt, instr_sltu, instr_xor, instr_srl, instr_sra,
instr_or, instr_and;

reg instr_rdcycle, instr_rdcycleh, instr_rdinstr, instr_rdinstrh, instr_ecall_ebreak;

reg instr_getq,instr_setq,instr_retirq,instr_maskirq,instr_waitirq,instr_timer

wire instr_trap;
reg [regindex_bits-1:0] decoded_rd, decoded_rs1, decoded_rs2;
reg [31:0] decoded_imm, decoded_imm_j;
reg decoder_trigger;
reg decoder_trigger_q;
reg decoder_pseudo_trigger;
reg decoder_pseudo_trigger_q;
reg compressed_instr;
reg is_lui_auipc_jal;
reg is_lb_lh_lw_lbu_lhu;
reg is_slli_srli_srai;

reg is_jalr_addi_slti_sltiu_xori_ori_andi;
reg is_sb_sh_sw;
reg is_sll_srl_sra;
reg is_lui_auipc_jal_jalr_addi_add_sub;
reg is_slti_blt_slt;
reg is_sltiu_bltu_sltu;
reg is_beq_bne_blt_bge_bltu_bgeu;
reg is_lbu_lhu_lw;
reg is_alu_reg_imm;
reg is_alu_reg_reg;
reg is_compare;
```

And instruction trap which gets enabled when none of the instruction is currently executing and when one of the CATCH_ILLINSN (Set this to 0 to disable the circuitry for catching illegal instructions) , WITH_PCPI becomes high.

And note that there are some user defined instructions along with picorv32 Instruction set.

```
assign instr_trap = (CATCH_ILLINSN || WITH_PCPI) && !{instr_lui, instr_auipc, instr_jal,
instr_jalr, instr_beq, instr_bne, instr_blt, instr_bge, instr_bltu,
instr_bgeu,instr_lb,instr_lh,instr_lw,instr_lbu,instr_lhu, instr_sb, instr_sh,
instr_sw,instr_addi, instr_slti, instr_sltiu,instr_xori, instr_ori,instr_andi, instr_slli,
instr_srli, instr_srai,instr_add, instr_sub, instr_sll, instr_slt, instr_sltu, instr_xor,
instr_srl,instr_sra,instr_or,instr_and,     instr_rdcycle, instr_rdcycleh, instr_rdinstr,
instr_rdinstrh,instr_getq, instr_setq, instr_retirq, instr_maskirq, instr_waitirq,
instr_timer};

wire is_rdcycle_rdcycleh_rdinstr_rdinstrh;
assign is_rdcycle_rdcycleh_rdinstr_rdinstrh = |{instr_rdcycle, instr_rdcycleh,
                                              instr_rdinstr, instr_rdinstrh};
```

And new_ascii_instr reg stores the string of whatever instruction type that got activated.

Exact decoding part:-

Here in this procedural block during posedge of the clock, the common merged instruction set will be enabled if any of that instruction type from the set gets active so that in later decoding stage using this common merged instruction set, addresses of decoded rs1, rs2, rd will get fetched.

And here on, there are simply two loops, which would explain us:-

To identify which instruction that is going to execute, this loop is defined.

```
If (mem_do_rinst && mem_done) begin
….
end
```

Here decoding of opcode (identify the instruction) and rs1, rs2, rd, imm progression occurs in this block of the code. Which is based on one reg called mem_rdata_latched .

So here in the first kind of instructions like U-type and J-type, if they are going to execute will get identified with the individual matching of each of them to first 7 lsb bits of mem_rdata_latched.
And other instruction types like load, store and R-type, merged instruction set is used for matching so in the next loop, decoding of specified instruction amongst them will be done .

In this loop only for those instructions which are not compressed the decoded rs1, rs2 and rd are fetched. Proper ISA along with opcode and operand addresses are being mentioned below.

mem_rdata_latched (ISA) :-

31 25	24 20	19 15	14 12	11 7	6 0
opcode	rs2	rs1	opcode	rd	opcode

```
always @(posedge clk) begin
is_lui_auipc_jal <= |{instr_lui, instr_auipc, instr_jal};
is_lui_auipc_jal_jalr_addi_add_sub <= |{instr_lui, instr_auipc, instr_jal,       instr_jalr,
instr_addi, instr_add, instr_sub};
is_slti_blt_slt <= |{instr_slti, instr_blt, instr_slt};
is_sltiu_bltu_sltu <= |{instr_sltiu, instr_bltu, instr_sltu};
is_lbu_lhu_lw <= |{instr_lbu, instr_lhu, instr_lw};
is_compare <= |{is_beq_bne_blt_bge_bltu_bgeu, instr_slti, instr_slt, instr_sltiu,
instr_sltu};

 if (mem_do_rinst && mem_done) begin
      instr_lui     <= mem_rdata_latched[6:0] == 7'b0110111;
      instr_auipc   <= mem_rdata_latched[6:0] == 7'b0010111;
      instr_jal     <= mem_rdata_latched[6:0] == 7'b1101111;
      instr_jalr    <= mem_rdata_latched[6:0] == 7'b1100111 &&
                       mem_rdata_latched[14:12] == 3'b000;
      instr_retirq  <= mem_rdata_latched[6:0] == 7'b0001011 &&
                       mem_rdata_latched[31:25] == 7'b0000010 && ENABLE_IRQ;
      instr_waitirq <= mem_rdata_latched[6:0] == 7'b0001011 &&
                        mem_rdata_latched[31:25] == 7'b0000100 && ENABLE_IRQ;
      is_beq_bne_blt_bge_bltu_bgeu <= mem_rdata_latched[6:0] == 7'b1100011;
      is_lb_lh_lw_lbu_lhu          <= mem_rdata_latched[6:0] == 7'b0000011;
      is_sb_sh_sw                  <= mem_rdata_latched[6:0] == 7'b0100011;
      is_alu_reg_imm               <= mem_rdata_latched[6:0] == 7'b0010011;
      is_alu_reg_reg               <= mem_rdata_latched[6:0] == 7'b0110011;

      { decoded_imm_j[31:20], decoded_imm_j[10:1], decoded_imm_j[11], decoded_imm_j[19:12],
decoded_imm_j[0] } <= $signed({mem_rdata_latched[31:12], 1'b0});

      decoded_rd <= mem_rdata_latched[11:7];
      decoded_rs1 <= mem_rdata_latched[19:15];
      decoded_rs2 <= mem_rdata_latched[24:20];
```

And there is a separate loop defined for compressed instructions inside this if loop only, and this is activated by COMPRESSED_ISA signal. Entering into the loop, decoded_rd , rs1, rs2 are initialized to “0” .

So here there is Quadrant division by using case statement on first 2 lsb bits of mem_rdata_latched, which is same as in memory interface for mem_rdata_q using mem_rdata_latched.

Inside every case statement of this another case statements are defined by taking the mem_rdata_latched [15:13] bits so that separate decoding process can be defined for each of the instruction set of different addressing modes.

Like here in this quadrant 0 for Load , so if that case got satisfied then the common merged instruction type (is_lb_lh_lw_lbu_lhu) will get enable first and then here as we know that in the load instruction there will be rs1 and rd required so here decoded_rs1 and decoded_rs2 will get decoded again using mem_rdata_latched [9:7] , [4:2] (this might be different for different instruction type decoding)

Similarly for store instruction first is_sb_sh_sw will get enabled and we need rs1 and rs2 so here decoded_rs1, decoded_rs2 will get decoded and get the addresses of two source registers.

```
if (COMPRESSED_ISA && mem_rdata_latched[1:0] != 2'b11) begin
compressed_instr <= 1;
decoded_rd <= 0;
decoded_rs1 <= 0;
decoded_rs2 <= 0;

{decoded_imm_j[31:11],decoded_imm_j[4],decoded_imm_j[9:8],decoded_imm_j[10],
decoded_imm_j[6],decoded_imm_j[7],decoded_imm_j[3:1],decoded_imm_j[5],
decoded_imm_j[0] } <= $signed({mem_rdata_latched[12:2], 1'b0});

 case (mem_rdata_latched[1:0])
      2'b00: begin // Quadrant 0
            case (mem_rdata_latched[15:13])
            3'b000: begin // C.ADDI4SPN
            is_alu_reg_imm <= |mem_rdata_latched[12:5];
            decoded_rs1 <= 2;
            decoded_rd <= 8 + mem_rdata_latched[4:2];
            end
            3'b010: begin // C.LW
            is_lb_lh_lw_lbu_lhu <= 1;
             decoded_rs1 <= 8 + mem_rdata_latched[9:7];
             decoded_rd <= 8 + mem_rdata_latched[4:2];
            end
            3'b110: begin // C.SW
             is_sb_sh_sw <= 1;
              decoded_rs1 <= 8 + mem_rdata_latched[9:7];
              decoded_rs2 <= 8 + mem_rdata_latched[4:2];
            end
            endcase
            end
            2'b01: begin // Quadrant 1
            case (mem_rdata_latched[15:13])
                  3'b000: begin // C.NOP / C.ADDI
                  is_alu_reg_imm <= 1;
                  decoded_rd <= mem_rdata_latched[11:7];
                  decoded_rs1 <= mem_rdata_latched[11:7];
                  end
                  3'b001: begin // C.JAL
                  instr_jal <= 1;
                  decoded_rd <= 1;
                  end
                  3'b 010: begin // C.LI
                  is_alu_reg_imm <= 1;
```

```
                   decoded_rd <= mem_rdata_latched[11:7];
                   decoded_rs1 <= 0;
                   end
                   3'b 011: begin
                   if (mem_rdata_latched[12] || mem_rdata_latched[6:2]) begin
                   if (mem_rdata_latched[11:7] == 2) begin // C.ADDI16SP
                   is_alu_reg_imm <= 1;
                   decoded_rd <= mem_rdata_latched[11:7];
                   decoded_rs1 <= mem_rdata_latched[11:7];
                   end else begin // C.LUI
                   instr_lui <= 1;
                   decoded_rd <= mem_rdata_latched[11:7];
                   decoded_rs1 <= 0;
                   end
                   end
             end
```

If it is R-type or I-type then it must be alu related instruction so either is_alu_reg_imm (decoded_rs1, decoded_rd) or is_alu_reg_imm (decoded_rs1, decoded_rd, decoded_rs2) will get enabled again all through mem_rdata_latched bits. Even for shifting this is enabled but then here as there should be Shift on rs1 left or right by the number of bits specified in the least significant 5 bits of rs2 and store the result in rd 1 so here you are required to have decoded_rs1 ,rs2, rd.

```
3'b100: begin
if (!mem_rdata_latched[11] && !mem_rdata_latched[12]) begin // C.SRLI, C.SRAI
      is_alu_reg_imm <= 1;
      decoded_rd <= 8 + mem_rdata_latched[9:7];
      decoded_rs1 <= 8 + mem_rdata_latched[9:7];
      decoded_rs2 <= {mem_rdata_latched[12], mem_rdata_latched[6:2]};
      end
      if (mem_rdata_latched[11:10] == 2'b10) begin // C.ANDI
      is_alu_reg_imm <= 1;
      decoded_rd <= 8 + mem_rdata_latched[9:7];
      decoded_rs1 <= 8 + mem_rdata_latched[9:7];
      end
if (mem_rdata_latched[12:10] == 3'b011) begin // C.SUB, C.XOR, C.OR, C.AND
      is_alu_reg_reg <= 1;
      decoded_rd <= 8 + mem_rdata_latched[9:7];
      decoded_rs1 <= 8 + mem_rdata_latched[9:7];
      decoded_rs2 <= 8 + mem_rdata_latched[4:2];
      end
      end
```

And to identify the exact executing instruction, this loop is defined.

```
if (decoder_trigger && !decoder_pseudo_trigger) begin
….
End
```

Here as mentioned for knowing the exact instruction, mem_data_q and common merged instruction register is used because in the previous loop we could analyse and decode for the common merged instruction type.

And this mem_rdata_q is redirected from the memory interface which would get yielded same using case statements on mem_rdata_latched and its same kind of case loops that we have in instruction decoded that are there in memory interface for mem_rdata_q.

So for example if exactly sh (store half) instruction is executing among all S-type like (sh, sb, sw) then instr_sh   <= is_sb_sh_sw && mem_rdata_q[14:12] == 3'b001; statement will get enabled if is_sb_sh_sw will get activated from previous loop and mem_rdata_q is extra factor from which exact instruction could be known to us, so from mapping for this mem_rdata_q bits to different values, exact instruction can be decoded which serves the opcode purpose here and decoding of rs1 and rs2 and rd is already done in the previous loop.

Similarly for lw instruction instr_lw   <= is_lb_lh_lw_lbu_lhu && mem_rdata_q[14:12] == 3'b010

```
if (decoder_trigger && !decoder_pseudo_trigger) begin
      pcpi_insn <= WITH_PCPI ? mem_rdata_q : 'bx;

    instr_beq   <= is_beq_bne_blt_bge_bltu_bgeu && mem_rdata_q[14:12] == 3'b000;
    instr_bne   <= is_beq_bne_blt_bge_bltu_bgeu && mem_rdata_q[14:12] == 3'b001;
    instr_blt   <= is_beq_bne_blt_bge_bltu_bgeu && mem_rdata_q[14:12] == 3'b100;
   instr_bge   <= is_beq_bne_blt_bge_bltu_bgeu && mem_rdata_q[14:12] == 3'b101;
instr_bltu  <= is_beq_bne_blt_bge_bltu_bgeu && mem_rdata_q[14:12] == 3'b110;
instr_bgeu  <= is_beq_bne_blt_bge_bltu_bgeu && mem_rdata_q[14:12] == 3'b111;

instr_lb    <= is_lb_lh_lw_lbu_lhu && mem_rdata_q[14:12] == 3'b000;
instr_lh    <= is_lb_lh_lw_lbu_lhu && mem_rdata_q[14:12] == 3'b001;
instr_lw    <= is_lb_lh_lw_lbu_lhu && mem_rdata_q[14:12] == 3'b010;
instr_lbu   <= is_lb_lh_lw_lbu_lhu && mem_rdata_q[14:12] == 3'b100;
instr_lhu   <= is_lb_lh_lw_lbu_lhu && mem_rdata_q[14:12] == 3'b101;

instr_sb    <= is_sb_sh_sw && mem_rdata_q[14:12] == 3'b000;
instr_sh    <= is_sb_sh_sw && mem_rdata_q[14:12] == 3'b001;
instr_sw    <= is_sb_sh_sw && mem_rdata_q[14:12] == 3'b010;

instr_addi  <= is_alu_reg_imm && mem_rdata_q[14:12] == 3'b000;
instr_slti  <= is_alu_reg_imm && mem_rdata_q[14:12] == 3'b010;
instr_sltiu <= is_alu_reg_imm && mem_rdata_q[14:12] == 3'b011;
instr_xori  <= is_alu_reg_imm && mem_rdata_q[14:12] == 3'b100;
instr_ori   <= is_alu_reg_imm && mem_rdata_q[14:12] == 3'b110;
instr_andi  <= is_alu_reg_imm && mem_rdata_q[14:12] == 3'b111;

instr_slli  <= is_alu_reg_imm && mem_rdata_q[14:12] == 3'b001 &&
               mem_rdata_q[31:25] == 7'b0000000;
instr_srli  <= is_alu_reg_imm && mem_rdata_q[14:12] == 3'b101 &&
               mem_rdata_q[31:25] == 7'b0000000;
instr_srai  <= is_alu_reg_imm && mem_rdata_q[14:12] == 3'b101 &&
                mem_rdata_q[31:25] == 7'b0100000;

instr_add   <= is_alu_reg_reg && mem_rdata_q[14:12] == 3'b000 &&
               mem_rdata_q[31:25] == 7'b0000000;
```

```
instr_sub   <= is_alu_reg_reg && mem_rdata_q[14:12] == 3'b000 &&
                mem_rdata_q[31:25] == 7'b0100000;
instr_sll   <= is_alu_reg_reg && mem_rdata_q[14:12] == 3'b001 &&
                mem_rdata_q[31:25] == 7'b0000000;
instr_slt   <= is_alu_reg_reg && mem_rdata_q[14:12] == 3'b010 &&
                 mem_rdata_q[31:25] == 7'b0000000;
instr_sltu  <= is_alu_reg_reg && mem_rdata_q[14:12] == 3'b011 &&
                mem_rdata_q[31:25] == 7'b0000000;
instr_xor   <= is_alu_reg_reg && mem_rdata_q[14:12] == 3'b100 &&
                mem_rdata_q[31:25] == 7'b0000000;
instr_srl   <= is_alu_reg_reg && mem_rdata_q[14:12] == 3'b101 &&
                 mem_rdata_q[31:25] == 7'b0000000;
instr_sra   <= is_alu_reg_reg && mem_rdata_q[14:12] == 3'b101 &&
                 mem_rdata_q[31:25] == 7'b0100000;
instr_or    <= is_alu_reg_reg && mem_rdata_q[14:12] == 3'b110 &&
                 mem_rdata_q[31:25] == 7'b0000000;
instr_and   <= is_alu_reg_reg && mem_rdata_q[14:12] == 3'b111 &&
                mem_rdata_q[31:25] == 7'b0000000;
```