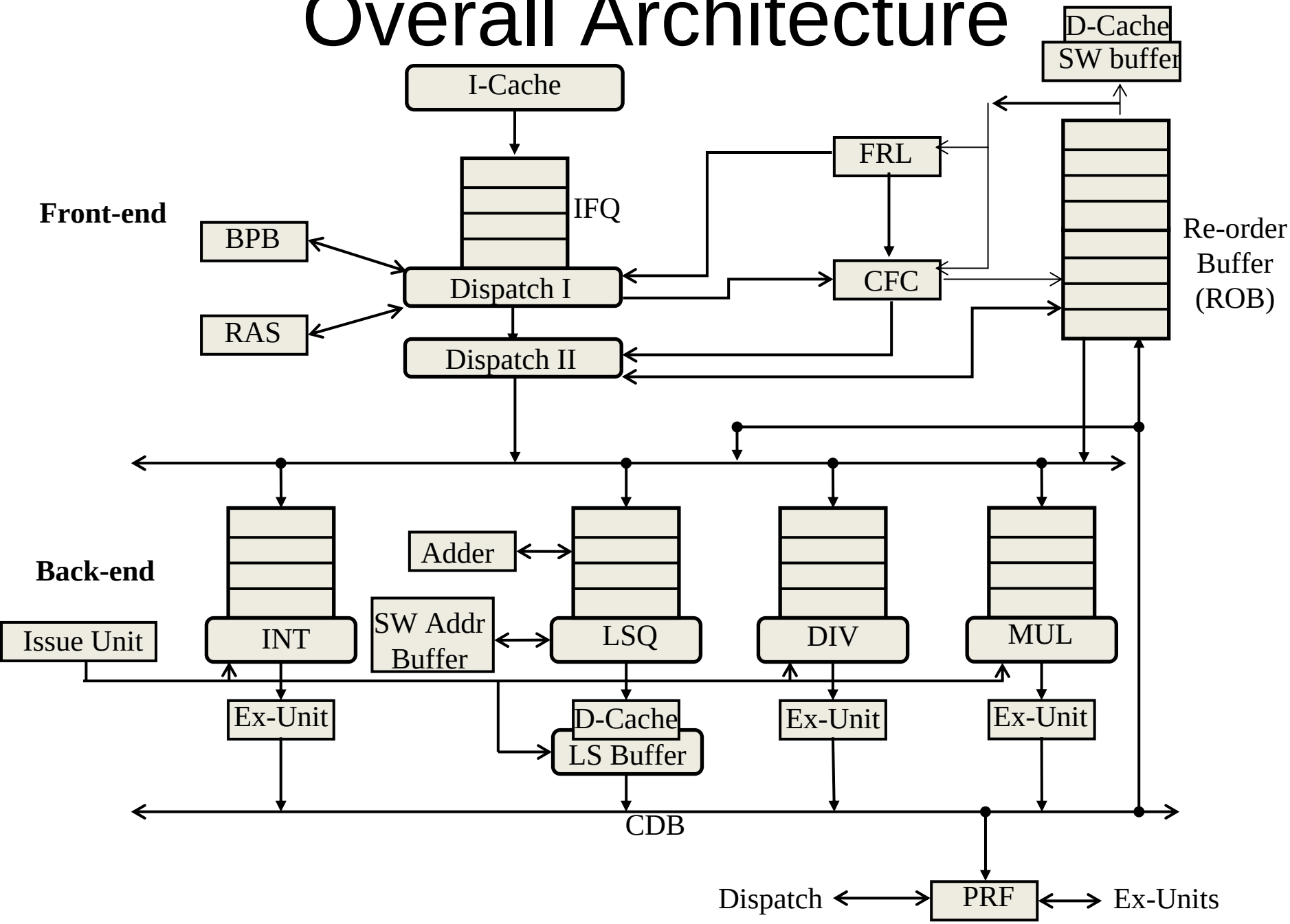
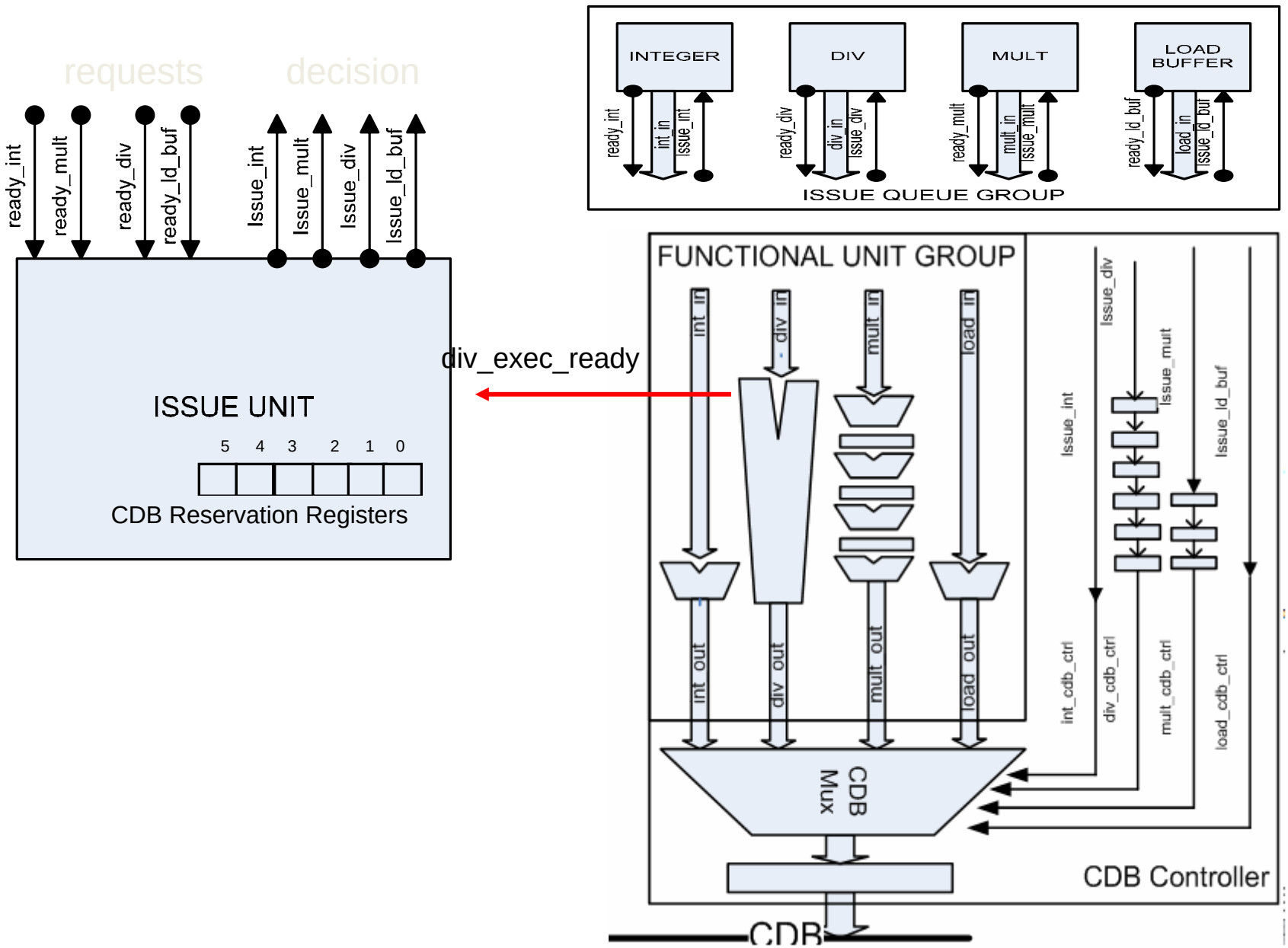
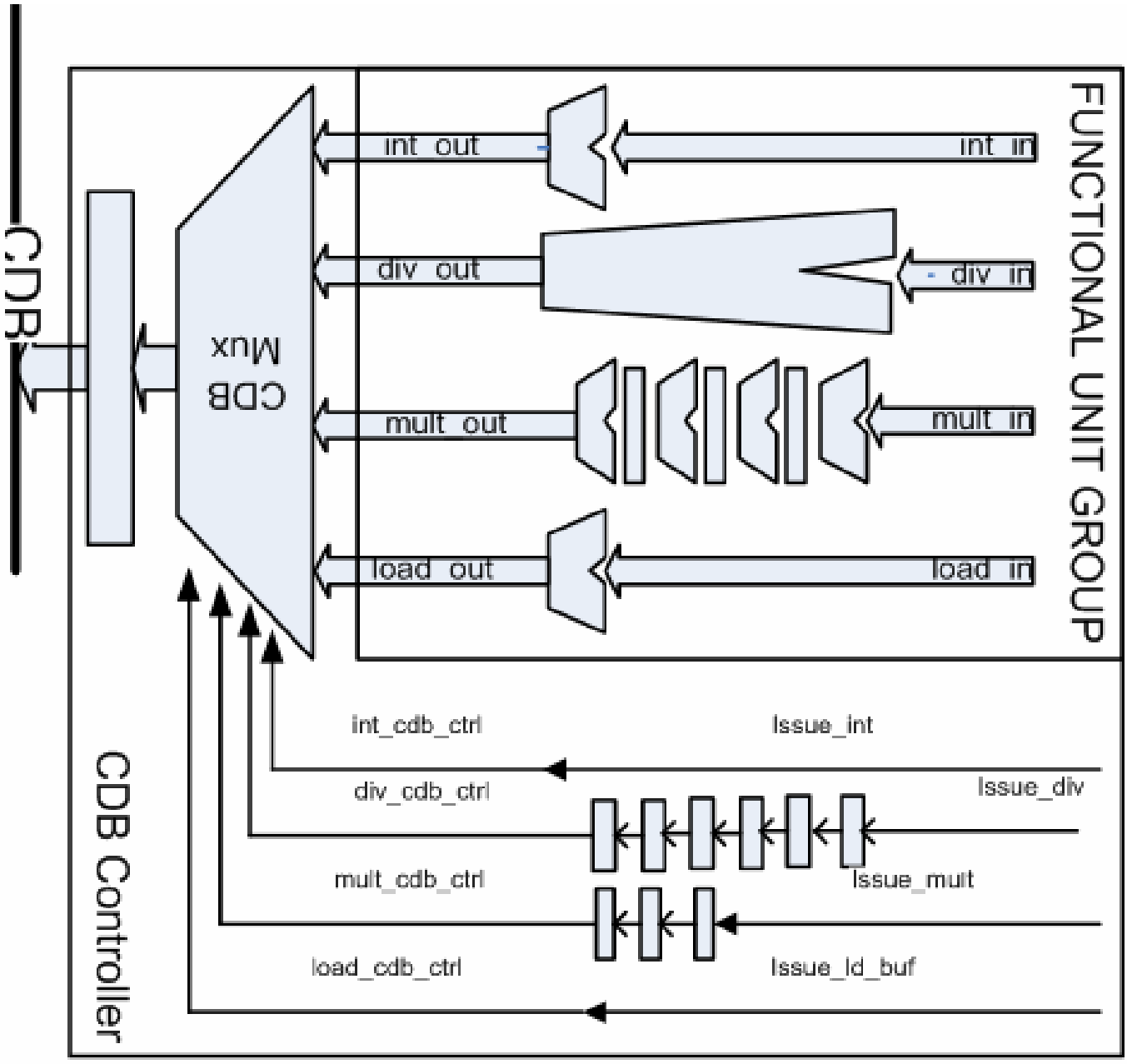
FUNCTION UNITS

ALU,
DIVIDER,
MULTIPLIER

Overall Architecture




requests
decision
ready_int
ready_mult
ready_div
ready_ld_buf
Issue_int
Issue_mult
Issue_div
Issue_ld_buf
ISSUE UNIT
5 4 3 2 1 0
CDB Reservation Registers
div_exec_ready
INTEGER
DIV
MULT
LOAD BUFFER
ready_int
int_in
issue_int
ready_div
div_in
issue_div
ready_mult
mult_in
issue_mult
ready_ld_buf
load_in
issue_ld_buf
ISSUE QUEUE GROUP
FUNCTIONAL UNIT GROUP
int_in
div_in
mult_in
load_in
int_out
div_out
mult_out
load_out
CDB Mux
Issue_int
Issue_div
Issue_mult
Issue_ld_buf
int_cdb_ctrl
div_cdb_ctrl
mult_cdb_ctrl
load_cdb_ctrl
CDB Controller
CDB


CDB
CDB Mux
int_out
div_out
mult_out
load_out
int_in
div_in
mult_in
load_in
FUNCTIONAL UNIT GROUP
CDB Controller
int_cdb_ctrl
div_cdb_ctrl
mult_cdb_ctrl
load_cdb_ctrl
Issue_int
Issue_div
Issue_mult
Issue_ld_buf

Latencies, pipelined/non-pipelined

- ALU is combinational
  (**one**-clock operation, **one**-clock latency)
- Divider is combinational
  (**six**-clock operation, **seven**-clock latency)
- Multiplier is pipelined
  (**four**-clock operation, **four**-clock latency)

ALU

SLIDES AREN'T MADE

DIVIDER

Divider
(Combinational six-clock operation)

16-bit combinational division

6-clocks time is allowed

A simplified div instruction

div $Rd, $Rs, $Rt

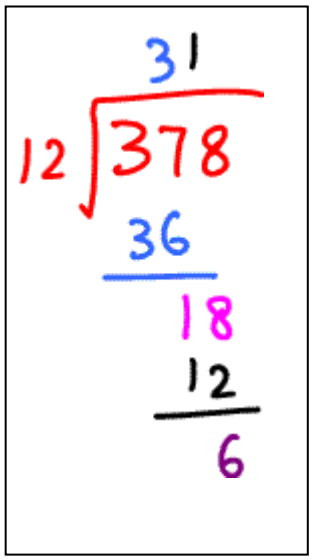
16-bit dividend => $Rs(15:0)

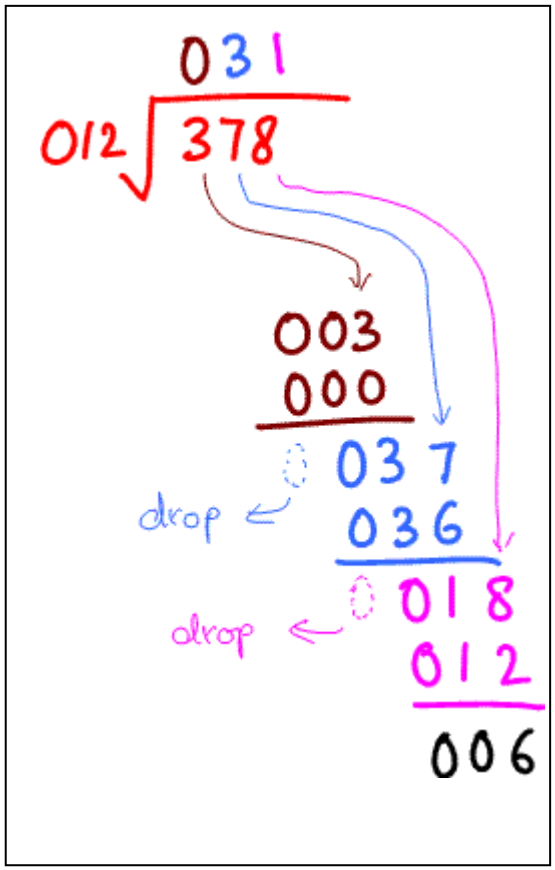
16-bit divisor=> $Rt(15:0)

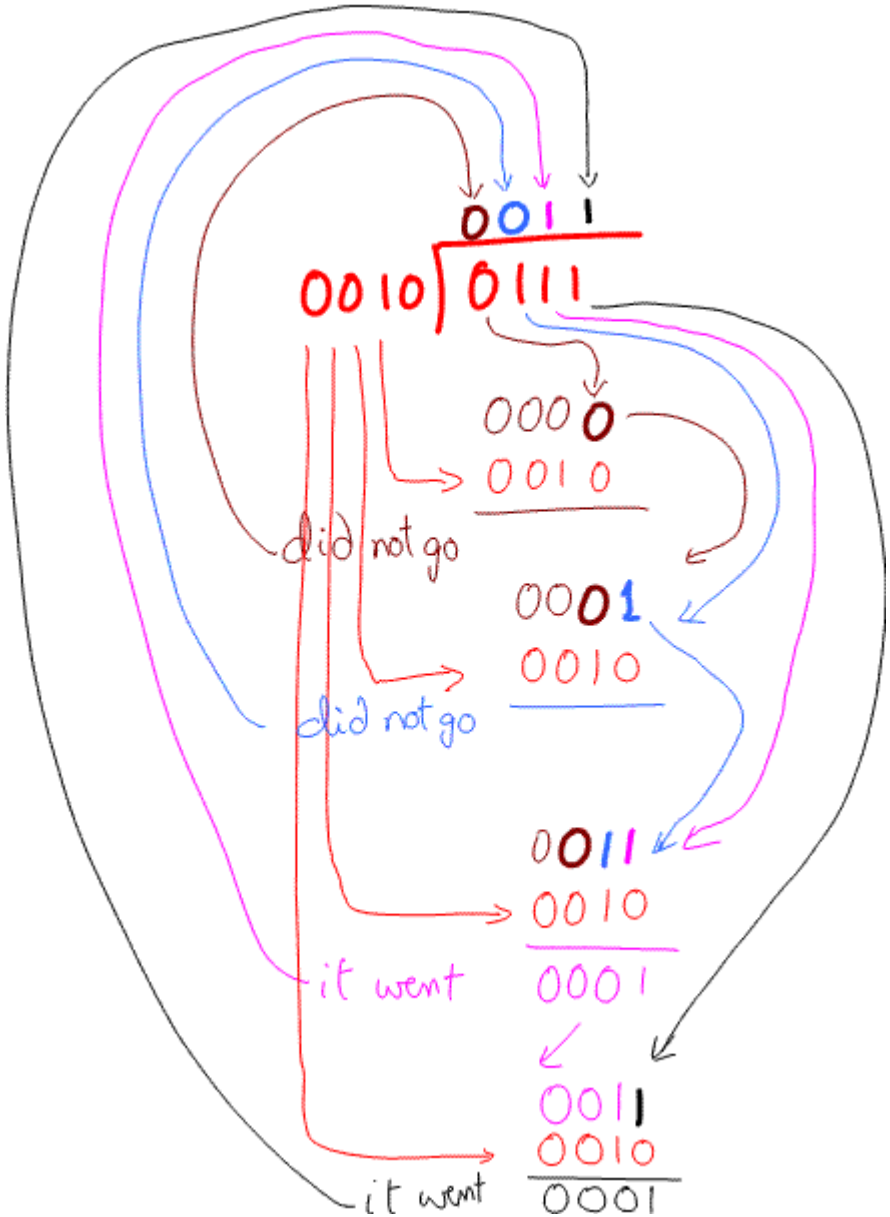
16-bit Remainder => $Rd(31:15)

16-bit Quotient => $Rd(15:0)

Combinational division (divider_core)



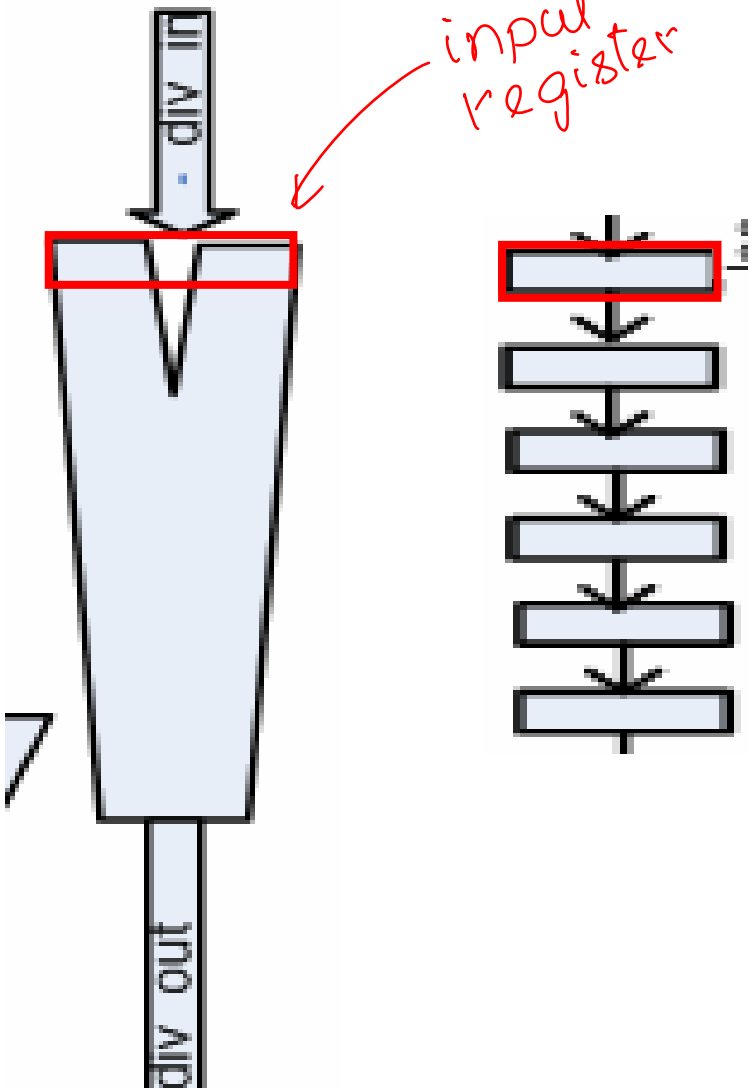




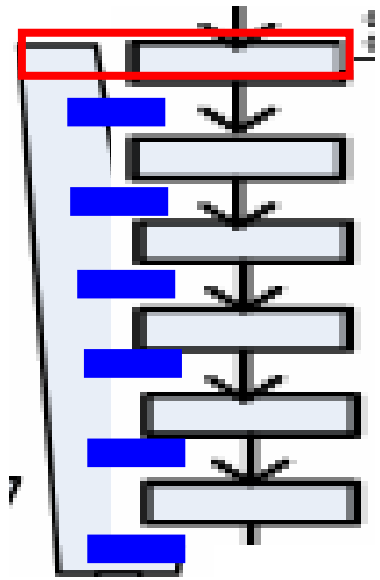
Combinational for-loop

```
for i in 0 to 15 loop

    Remain := Remain(14 downto 0) & Dvd(15 - i);
    IF ( unsigned(Remain) >= unsigned(Dvr) ) THEN
        Remain := unsigned(Remain) - unsigned(Dvr);
        Quo(15 - i) := '1';
    ELSE
        Quo(15 - i) := '0';
    END IF;

end loop;
```

7-clock latency



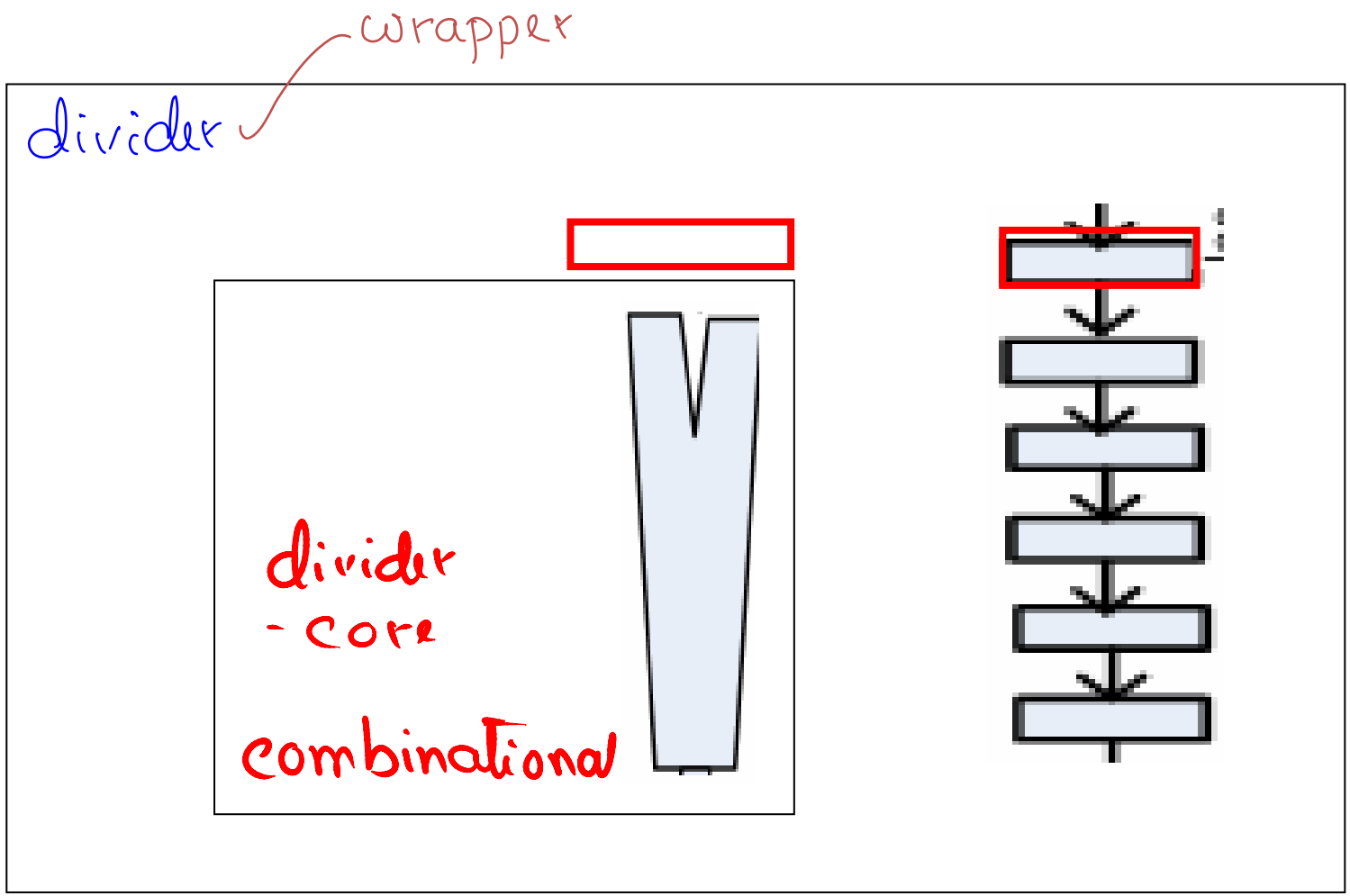
input register



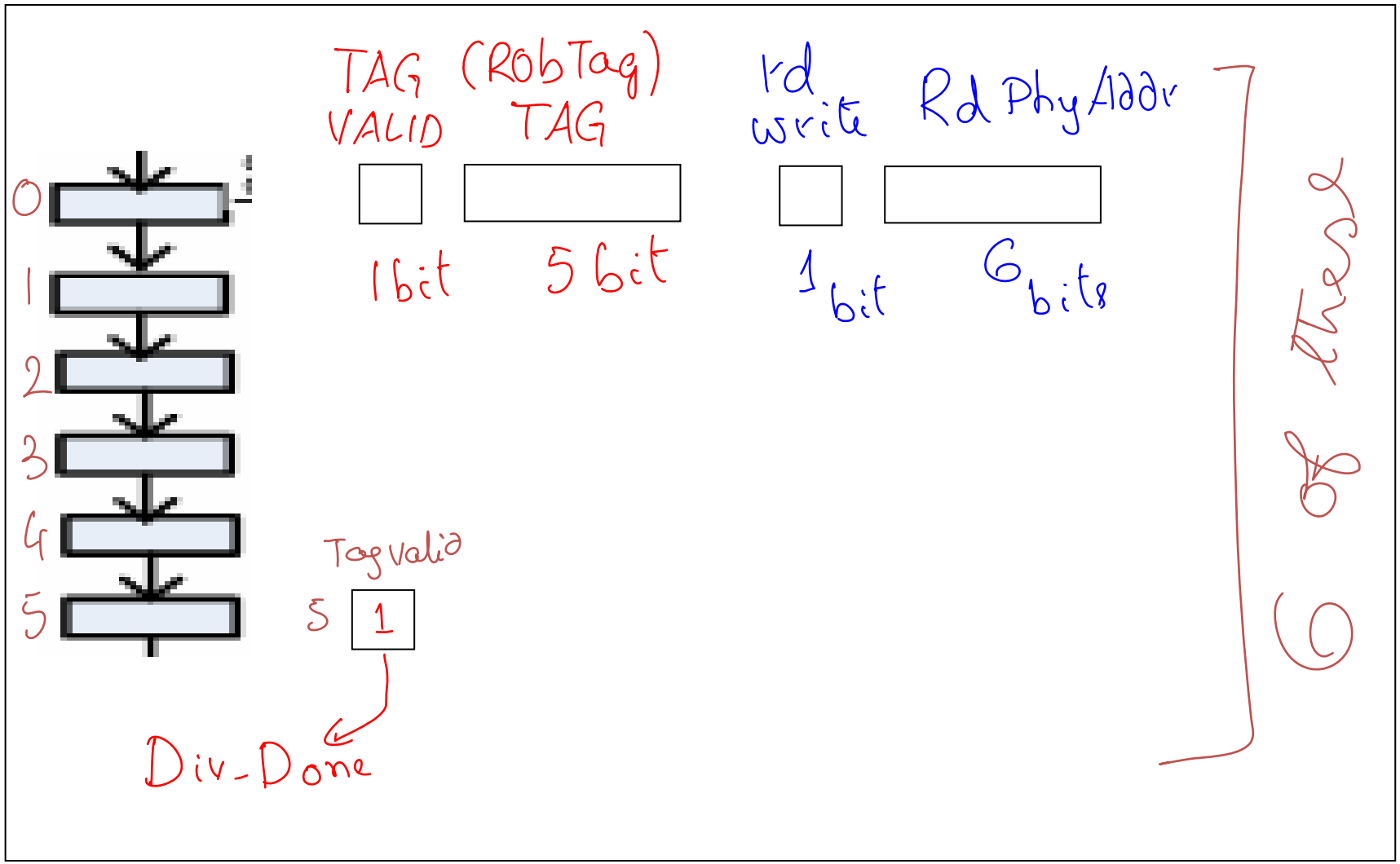
a clock-long combinational operation

Total 6 clocks for the divider to finish.

divider.vhd and divider_core.vhd



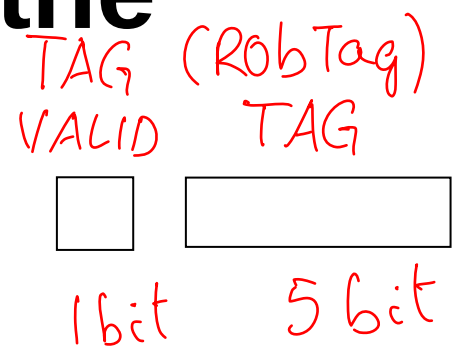
divider.vhd (the wrapper)

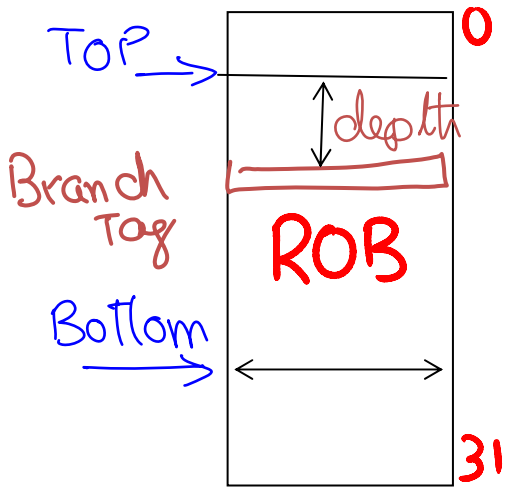


divider.vhd (the wrapper)

Selective flushing of the on-going division, if this divide instruction is younger to the mis-predicted instruction on the CDB

YOUNGER = Farther from the TOP of ROB

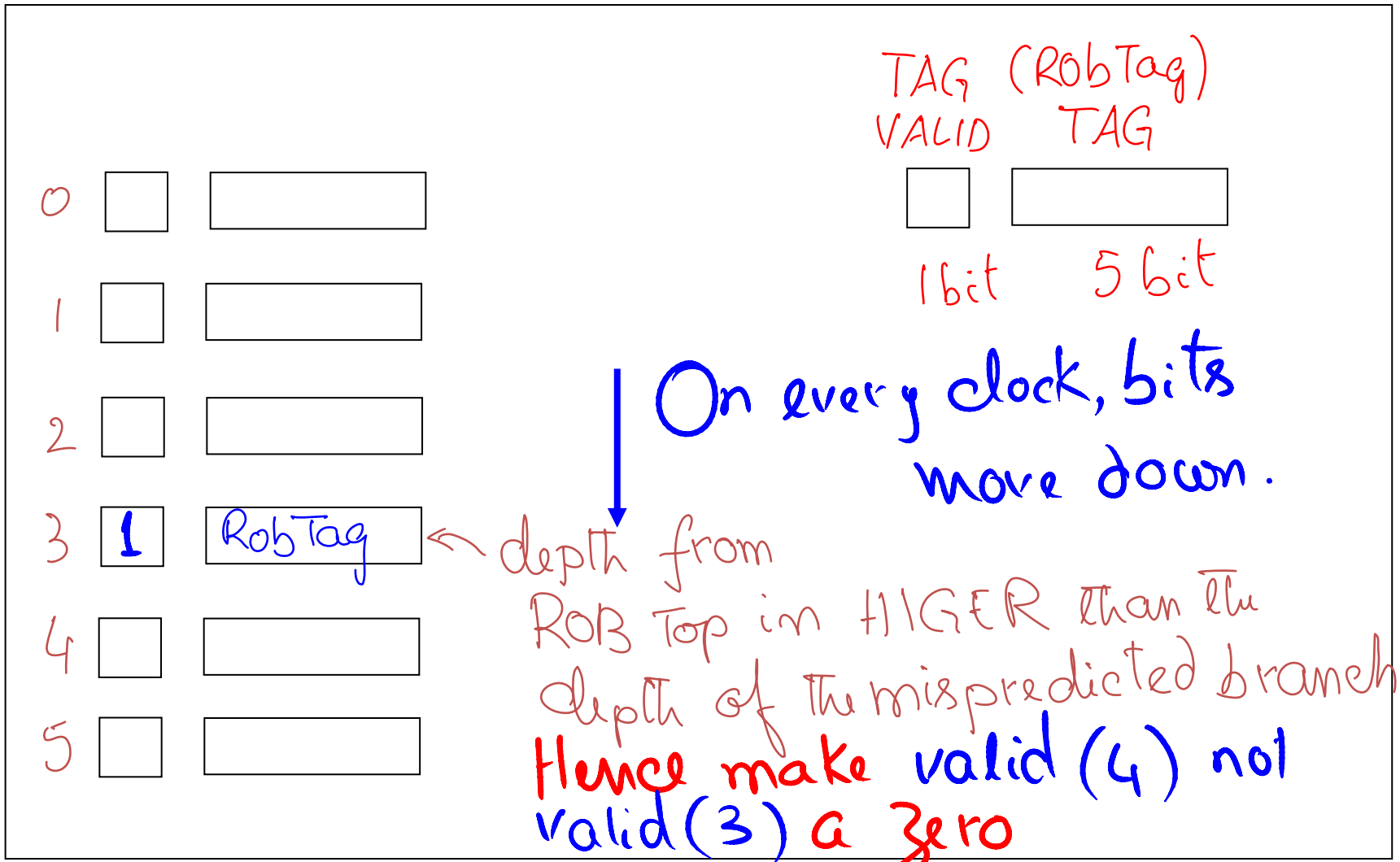




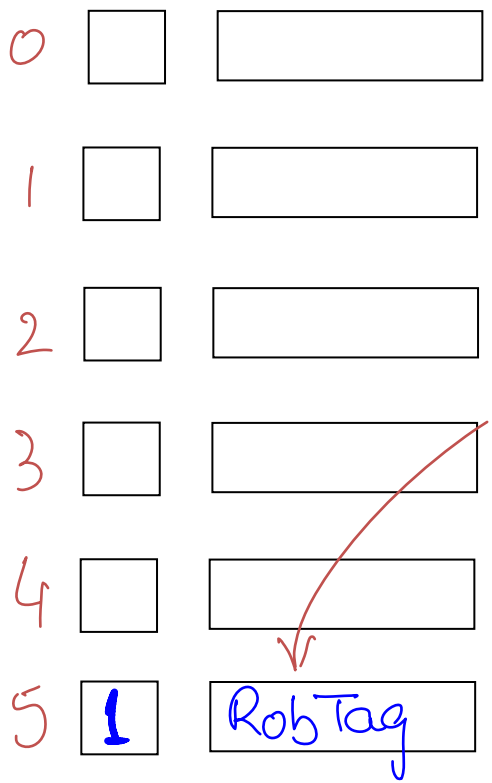
CDB

Mis-predicted branch

Selective flushing



Selective flushing



depth from ROB Top is HIGER than the depth of the mispredicted branch.

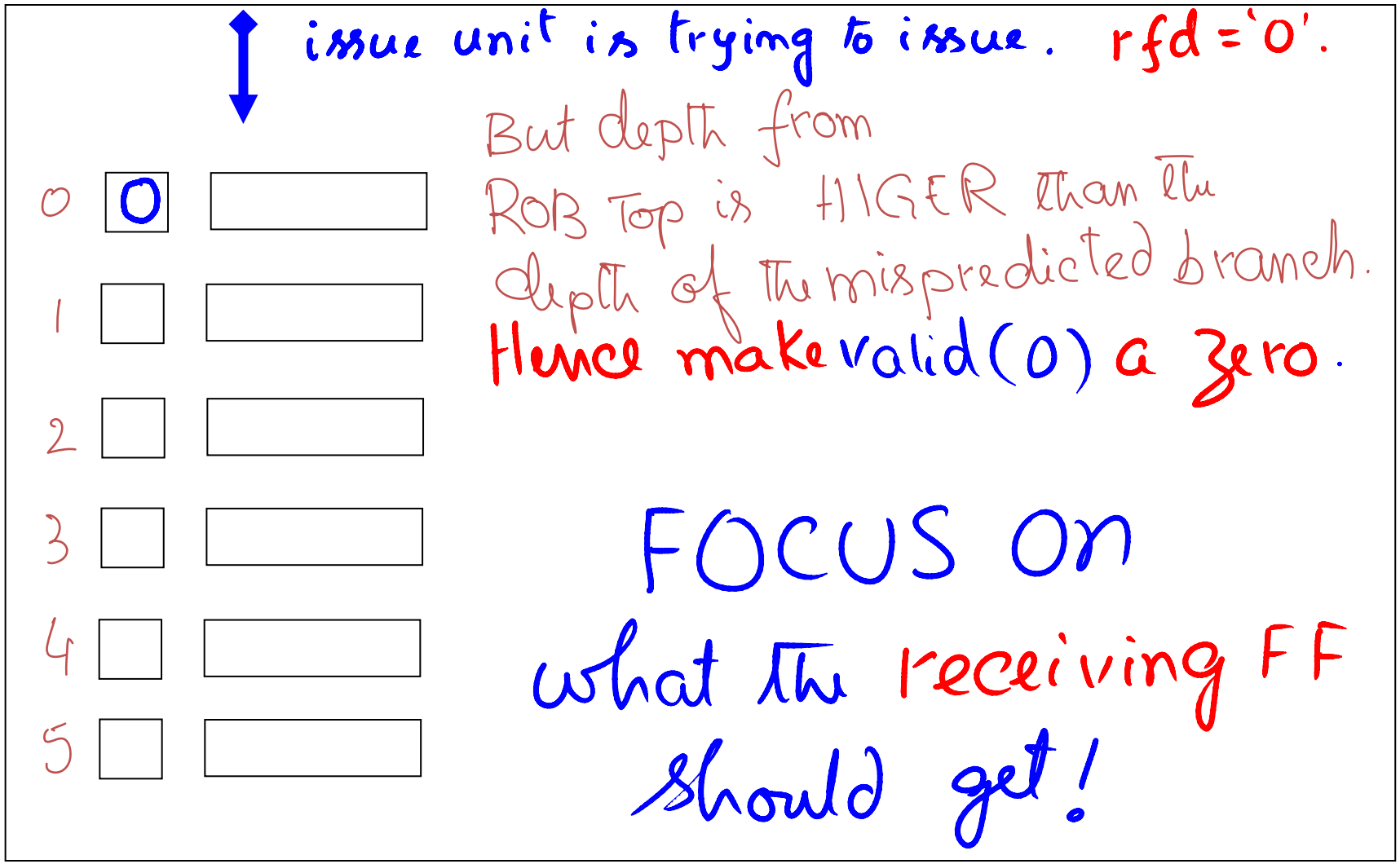
Hence make CDB valid, not valid(5), a Zero

This is take care of in cdb code.

Div_ExeRdy
(= rfd = ready for division)

- Currently a flipflop is used to record and convey the rest of the system, if the divider is ready to accept another division. (perhaps it can be done combinationally)
- rfd is set (on the next clock edge) if none of the V(0) to V(4) is a ‘1’ (or even if it is 1, the div instr. is being flushed). V(5) = ‘1’ is not included to gain one clock.

Selective flushing



MULTIPLIER

Multiplier
(4-stage pipelined, 4-clock operation, 3-stage registers)

16x16 multiplication producing 32-bit product

product

Multiplicand

Multiplier

P[31:0] = M[15:0] * Q[15:0]

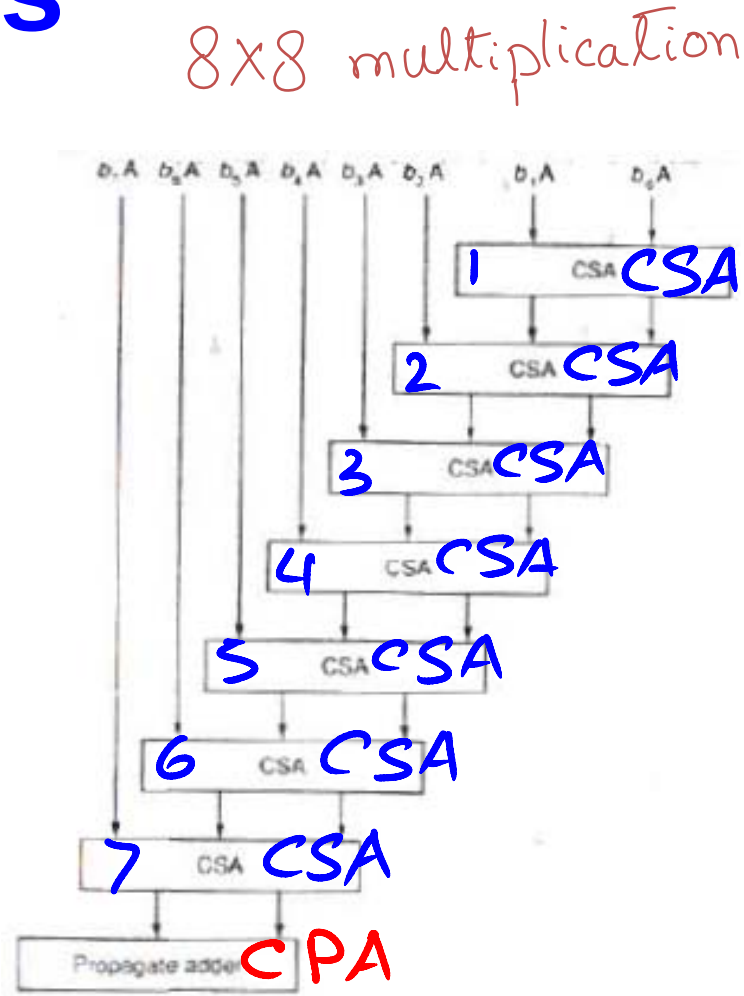
r_d r_s r_t

Linear cascade of

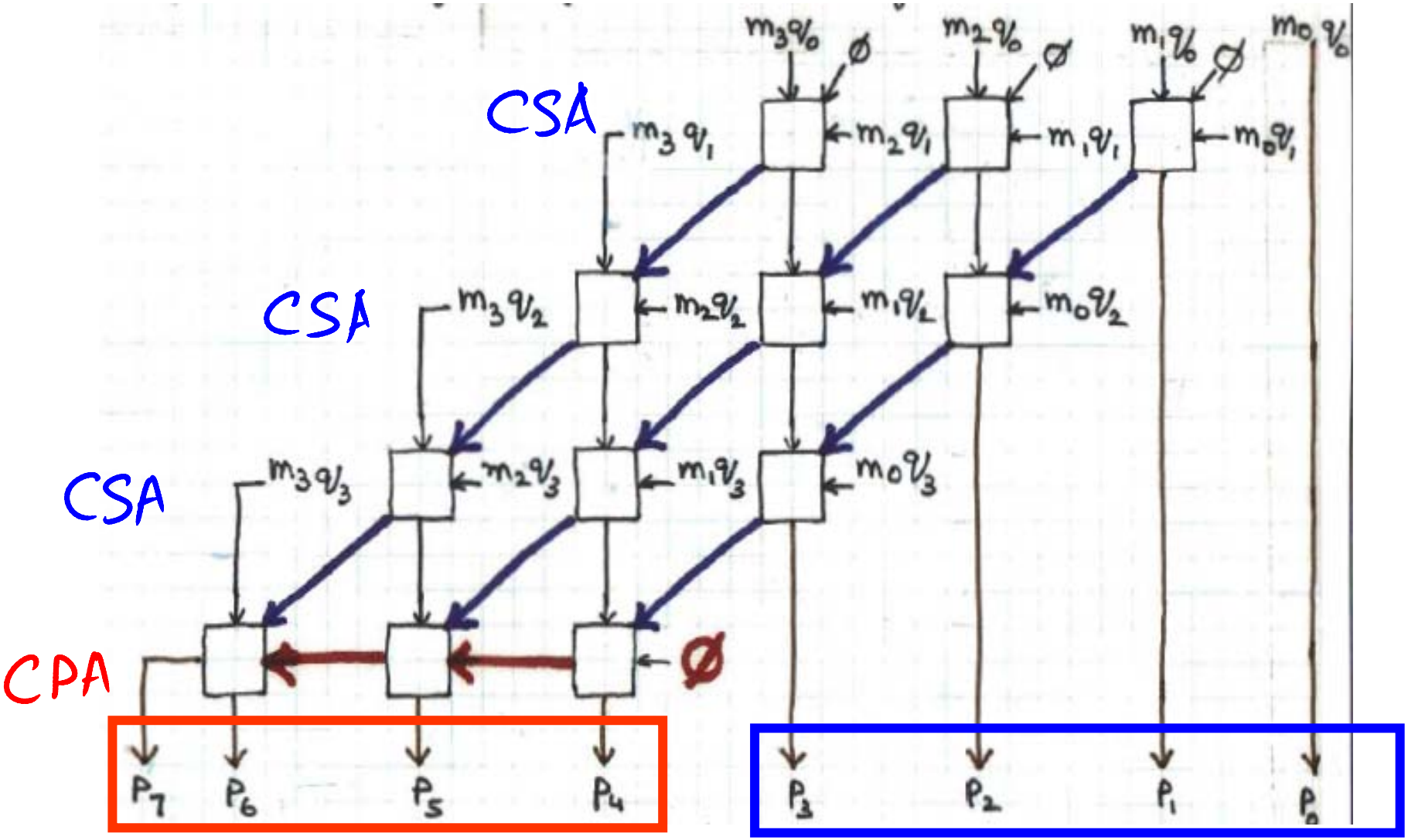
15-stages of CSAs

followed by

a CPA stage

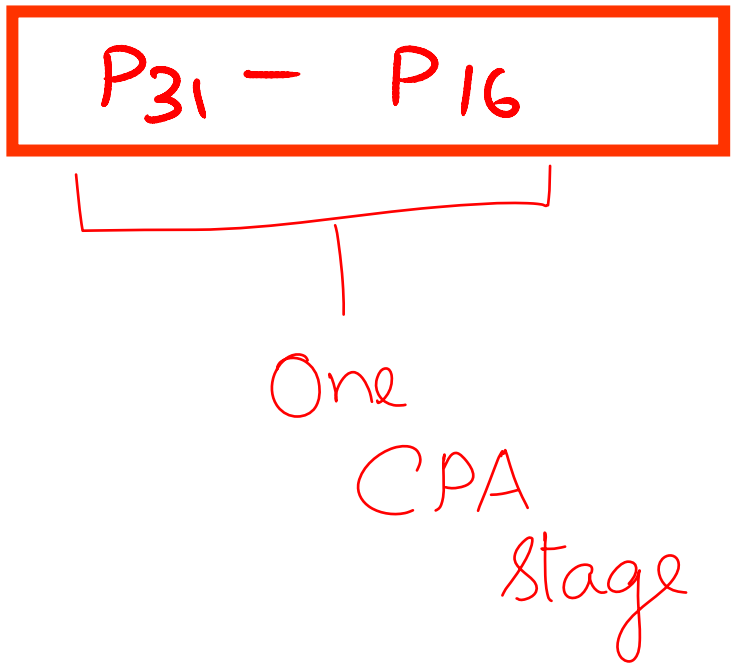


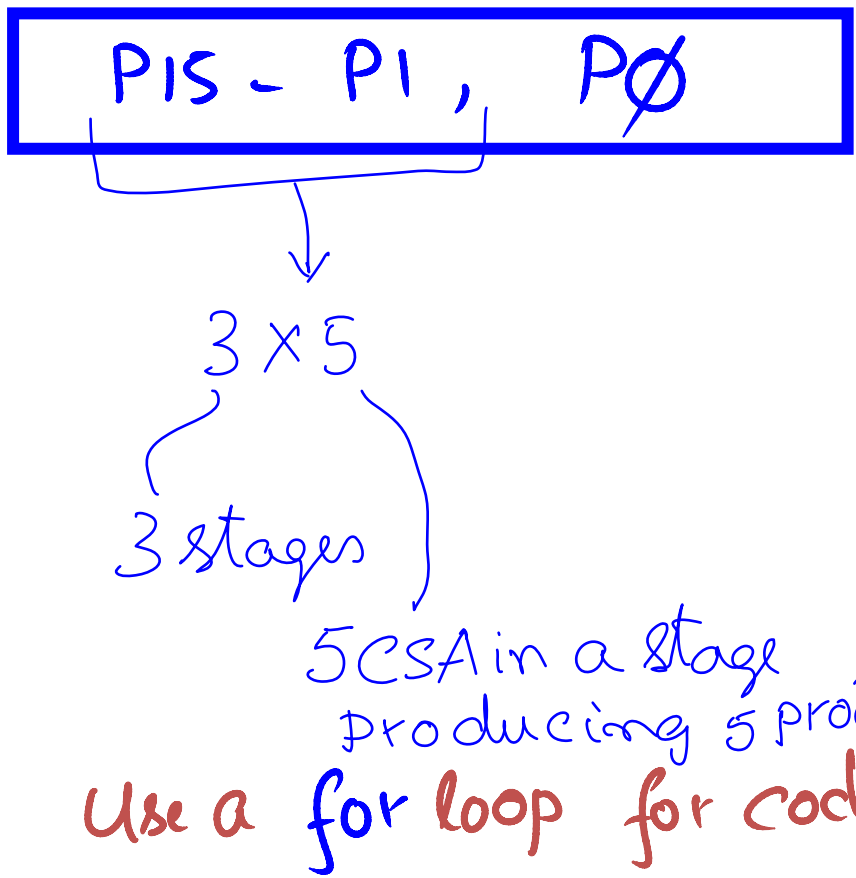
4x4 multiplier Three 3-bit CSAs and one 3-bit CPA



4x4 multiplier **Three 3-bit CSAs and one 3-bit CPA**

16x16 15 15-bit CSAs one 15-bit CPA

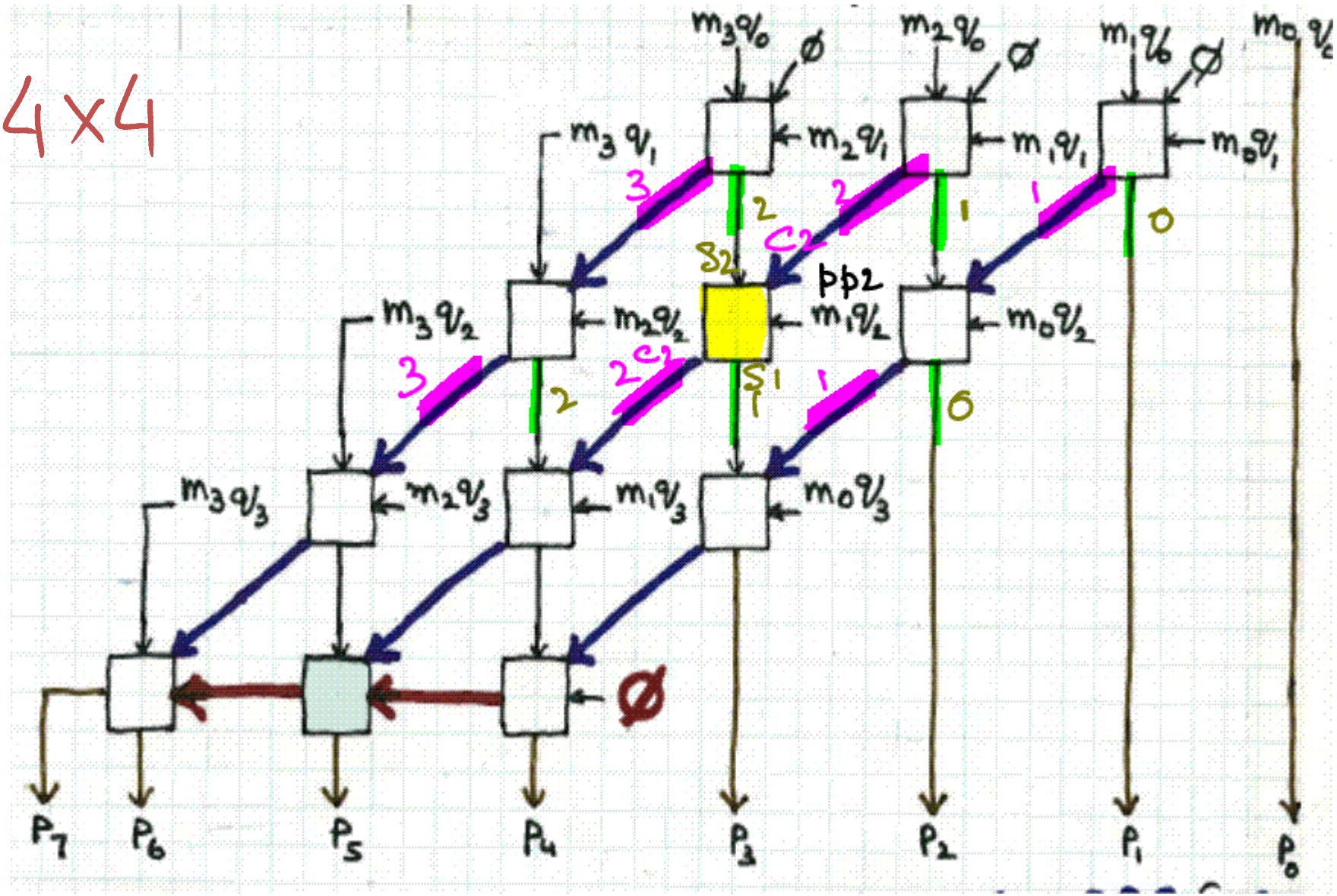




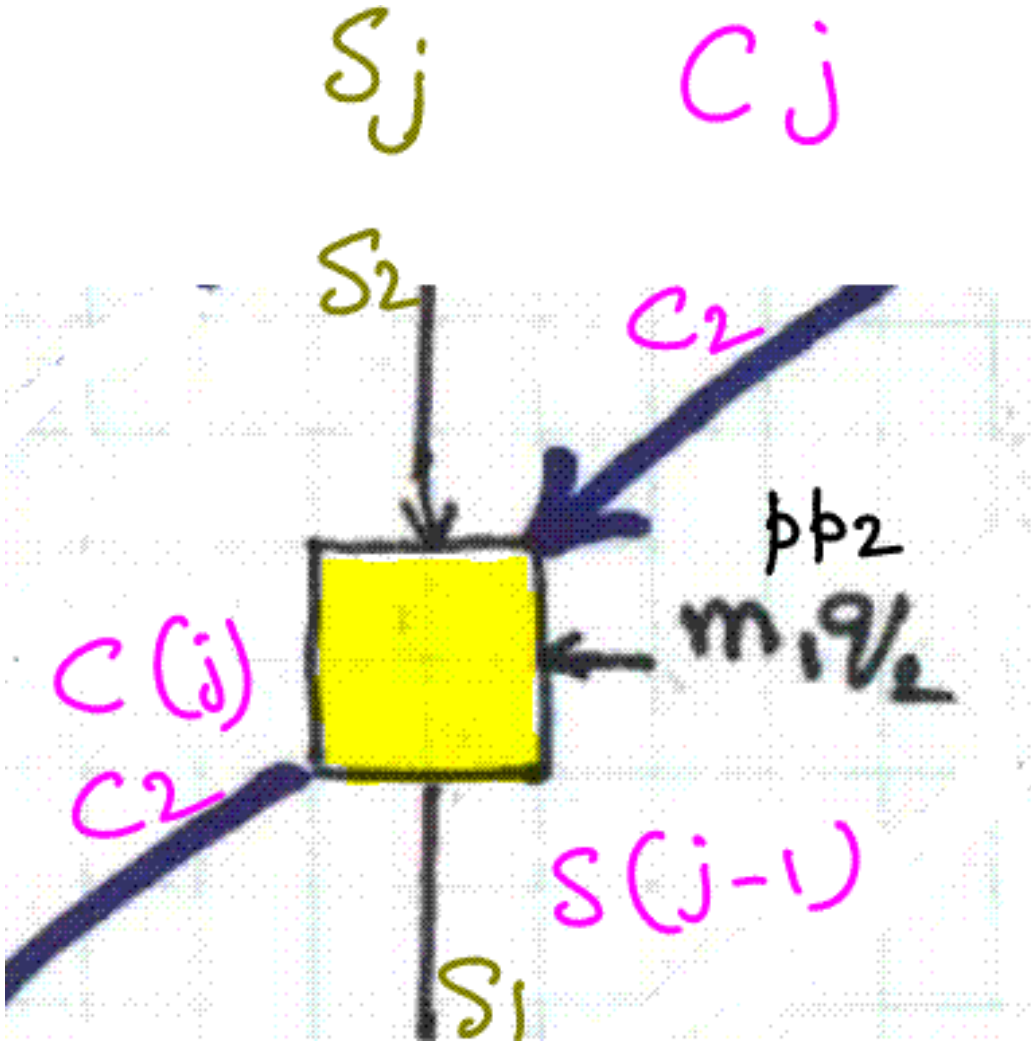
3 Sums $j = 1$ to 3

3 carries $j = 1$ to 3

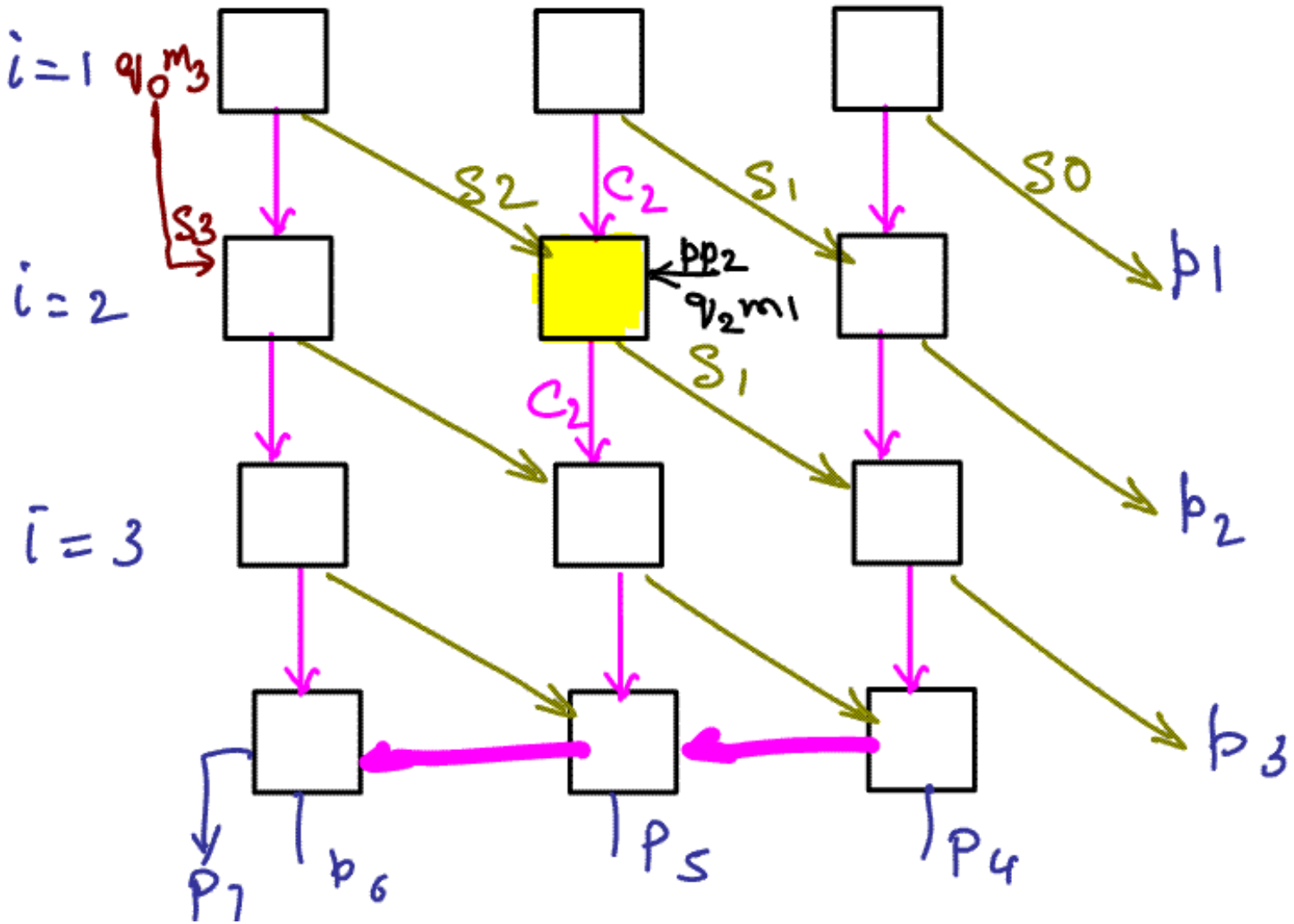
4x4



j^{th} CSA in i^{th} row



Redrawn for the convenience of creating a for loop



Code segment

```
mult_stage1_comb: process (m_A, q_A)
variable s_v_A : std_logic_vector (15 downto 0); -- sum input. -- note s_v_A is 16 bits where as s_A signal is 15 b
variable pp_v_A : std_logic_vector (15 downto 1); -- partial product for stage 1.
variable c_v_A : std_logic_vector (15 downto 1); -- carry input.
begin

  c_v_A(15 downto 1) := "000000000000000"; -- carry input for stage 1 is 0.

  s_v_A(15 downto 0) := (m_A(15) and q_A(0)) & (m_A(14) and q_A(0)) & (m_A(13) and q_A(0)) &
                        (m_A(12) and q_A(0)) & (m_A(11) and q_A(0)) & (m_A(10) and q_A(0)) &
                        (m_A(9) and q_A(0)) & (m_A(8) and q_A(0)) & (m_A(7) and q_A(0)) &
                        (m_A(6) and q_A(0)) & (m_A(5) and q_A(0)) & (m_A(4) and q_A(0)) &
                        (m_A(3) and q_A(0)) & (m_A(2) and q_A(0)) & (m_A(1) and q_A(0)) & (m_A(0) and q_A(0));
  P_A_5_to_0(0) <= s_v_A(0); -- the lowest partial product retires as product outputs 0th bit.

  for i in 1 to 5 loop -- this loop instantiates 5 stages of the 15-bit  CSA stages in the 16x16 multiplication.

    for j in 1 to 15 loop -- this loop makes one 15-bit CSA (one row of Full-Adder boxes)
      pp_v_A(j) := q_A(i) and m_A(j-1);
      s_v_A(j -1) := s_v_A(j) xor c_v_A(j) xor pp_v_A(j);
      c_v_A(j) := (s_v_A(j) and c_v_A(j)) or (c_v_A(j) and pp_v_A(j)) or (s_v_A(j) and pp_v_A(j));
    end loop;

    P_A_5_to_0(i) <= s_v_A(0);
    s_v_A(15) := m_A(15) and q_A(i);
  end loop;
  s_A_out <= s_v_A(15 downto 1); -- note s_v_A is 16 bits where as s_A signal is 15 bits
  c_A_out <= c_v_A;
end process mult_stage1_comb;
```