به نام خدا

دانشگاه صنعتی امیرکبیر

(پلی‌تکنیک تهران)

دانشکده مهندسی برق

پروژه نهایی VHDL درس مدارهای منطقی

دکتر محمدرضا پورفرد

مارال ترابی مهرام ۴۰۲۲۳۰۱۹

کیمیا خودسیانی ۴۰۲۲۳۰۳۰

مرداد ۱۴۰۴

فهرست مطالب

فصل اول) مقدمه

۱-۱. اهمیت تصحیح خطا در سیستم‌های دیجیتال

در دنیای دیجیتال، انتقال و ذخیره‌سازی داده‌ها همیشه با خطر خطا مواجه است. نویز در خطوط انتقال، خطاهای ناشی از نویزهای الکترومغناطیسی، خرابی‌های حافظه یا ناپایداری منابع تغذیه می‌توانند باعث تغییر بیت‌های داده شوند. اگر این خطاها بدون تشخیص باقی بمانند، می‌توانند منجر به خرابی سیستم، از بین رفتن اطلاعات و یا بروز عملکرد نادرست شوند. در همین راستا، استفاده از روش‌هایی برای تشخیص و حتی تصحیح خطا بسیار ضروری است، به‌ویژه در کاربردهایی مانند مخابرات، ذخیره‌سازی داده‌ها، سیستم‌های ایمن و کنترل صنعتی. کدهای تصحیح خطا (Error Correction Codes - ECC) ابزارهایی هستند که با افزودن افزونگی هوشمند به داده، امکان بازیابی داده اصلی حتی در حضور خطا را فراهم می‌سازند.

۱-۲. معرفی سیستم‌های ارسال/دریافت داده

سیستم‌های ارسال و دریافت داده، به عنوان اجزای کلیدی در ارتباطات دیجیتال، وظیفه انتقال مطمئن اطلاعات از یک منبع به یک مقصد را بر عهده دارند. این سیستم‌ها می‌توانند در قالب شبکه‌های کامپیوتری، پروتکل‌های سریال مانند UART یا SPI، یا سامانه‌های نهفته کاربرد داشته باشند. برای افزایش قابلیت اطمینان، استفاده از سازوکارهایی نظیر کدگذاری و رمزگشایی داده، بررسی صحت (Checksum) و ذخیره‌سازی موقت در حافظه ضروری است. در این پروژه، یک سیستم کامل ارسال و دریافت داده طراحی شده که در آن، اطلاعات ابتدا توسط یک ماژول Encoder به کد Hamming تبدیل شده، سپس ذخیره و پردازش می‌شود، و در نهایت با استفاده از Decoder بازسازی شده و صحت آن بررسی می‌شود.

۱-۳. هدف پروژه و کاربردهای آن

هدف این پروژه طراحی و پیاده‌سازی یک سیستم دیجیتال کامل در زبان VHDL است که بتواند یک داده ۸ بیتی را به‌صورت سریالی دریافت کرده، آن را به کد Hamming تبدیل کند، در RAM ذخیره نماید، روی آن عملیات منطقی/حسابی انجام دهد، سپس دوباره آن را به صورت کد شده ارسال یا در صورت نیاز بازیابی کند. کاربردهای چنین سیستمی در حوزه‌هایی نظیر مخابرات امن، پردازش داده در سامانه‌های توزیع‌شده، رابط‌های سریال صنعتی، و حافظه‌های با قابلیت تصحیح خطا بسیار گسترده است.

۱-۴. ساختار کلی سیستم طراحی‌شده

سیستم طراحی‌شده از چندین ماژول اصلی تشکیل شده است که به صورت سلسله‌وار و تحت کنترل یک واحد کنترلی مرکزی (Control Unit) با یکدیگر در ارتباط هستند. اجزای کلیدی عبارتند از:

Encoder (Hamming) : تولید کد ۱۳ بیتی از داده ۸ بیتی

Decoder (Hamming) : بازیابی داده و بررسی خطا

RAM : حافظه ۳۲×۸ بیتی برای ذخیره داده‌ها

ALU : انجام عملیاتAdd/Sub/OR/AND

Control Unit : مدیریت ترتیب اجرای عملیات

Packet Format : ساختار ارسال/دریافت داده باChecksum

این سیستم به گونه‌ای طراحی شده که بتواند پکت‌های مختلف با عملکردهای متفاوت (مثل Immediate، Operand، Array، Indirect) را پردازش کند.

فصل دوم) مبانی نظری و پایه‌ای

۲-۱. کدگذاری Hamming و کاربرد آن

کد Hamming روشی مؤثر برای تشخیص و تصحیح خطاهای تک‌بیتی است که با استفاده از بیت‌های توازن، موقعیت بیت خراب را شناسایی و در صورت امکان آن را اصلاح می‌کند. در کد Hamming، بیت‌های توازن در موقعیت‌هایی از داده قرار می‌گیرند که توان ۲ هستند (۱، ۲، ۴، ۸، ...). در این پروژه، از نسخه ۱۳ بیتی استفاده شده که شامل:

۸ بیت داده (در موقعیت‌های غیر توانی ۲)

۴ بیت توازن(P1 ، P2، P4، P8)

۱ بیت توازن کلی(Overall Parity - P_total)

مزیت مهم این روش، قابلیت تصحیح خطا به‌صورت کاملاً سخت‌افزاری با هزینه پایین منطقی است.

۲-۲. قالب پکت (Packet) و اجزای آن

در این سیستم برای انتقال اطلاعات بین بخش‌های مختلف مانند Encoder، RAM، ALU و Decoder از ساختار استانداردی به نام **پکت (Packet)** استفاده می‌شود.

پکت‌ها مانند بسته‌های اطلاعاتی هستند که فیلدهای مشخصی دارند و هر فیلد وظیفه‌ای خاص را بر عهده دارد. پکت‌ها بسته به نوع عملکردشان، ساختار متفاوتی دارند اما معمولاً شامل فیلدهای زیر هستند:

| Function | Address1 | Address2 / Data | Destination Address | Length | ChecksumH | ChecksumL |

- Function: تعیین می‌کند چه عملیاتی باید انجام شود (مثلاً جمع، نوشتن در حافظه، خواندن و ...)
- Address1 / Address2: آدرس‌هایی از حافظه برای استخراج یا ذخیره داده
- Data: در پکت‌هایی که Immediate هستند، این فیلد داده ورودی را مشخص می‌کند.
- Destination Address: محل نهایی ذخیره‌سازی نتیجه
- Length: در عملیات‌هایی مثل Array ALU مشخص می‌کند عملیات باید روی چند خانه حافظه انجام شود.
- ChecksumH / ChecksumL: بررسی صحت داده در طول انتقال

این طراحی انعطاف‌پذیری زیادی ایجاد می‌کند تا بتوان عملیات‌های مختلف را مانند موارد زیر انجام داد:

- انجام عملیات ALU با دو ورودی(Operand)

- انجام عملیات ALU با داده فوری (Immediate)
- پردازش آرایه‌ای در حافظه (Array)
- خواندن/نوشتن داده در حافظه
- استفاده از آدرس‌دهی غیرمستقیم (Indirect Addressing)

۲-۳. Checksum و روش محاسبه آن

برای اطمینان از صحت پکت دریافتی، از مکانیزم **Checksum** استفاده می‌شود.
در مرحله ارسال پکت، ابتدا مجموع تمام بایت‌های پکت (به‌جز فیلدهای Checksum خودش) محاسبه می‌شود. سپس این مقدار ۱۶ بیتی به دو قسمت ۸ بیتی تقسیم می‌شود:

- **CheckSumL**: ۸ بیت پایین‌تر مجموع
- **CheckSumH**: ۸ بیت بالاتر مجموع

در سمت گیرنده، همان جمع مجدداً انجام می‌شود و با مقادیر Checksum دریافتی مقایسه می‌گردد.
اگر مجموع با Checksum دریافتی برابر باشد، داده معتبر شناخته می‌شود. در غیر این صورت، سیگنال خطا (Error) فعال شده و اجرای عملیات متوقف می‌شود. این کار باعث افزایش اطمینان در سیستم و جلوگیری از اجرای عملیات روی داده‌های خراب می‌شود.

۲-۴. معرفی ALU، RAM و واحد کنترل (FSM)

- **ALU (واحد حساب و منطق)**:
  ALU بخشی از سیستم است که عملیات‌های منطقی و حسابی مانند جمع (Add) ، تفریق (Sub) ، یا (OR) و (AND) را انجام می‌دهد. نوع عملیات از طریق فیلد Function موجود در پکت مشخص می‌شود و ALU بسته به نوع دستور، داده‌های مربوطه را از حافظه خوانده و عملیات را انجام می‌دهد.
- **RAM(حافظه موقت)**:
  یک حافظه با ظرفیت ۳۲ خانه ۸ بیتی است که داده‌ها در آن ذخیره می‌شوند. قابلیت خواندن و نوشتن دارد و در هنگام فعال شدن سیگنال Reset (RST) ، تمام خانه‌های حافظه پاک‌سازی می‌شوند. خواندن و نوشتن داده در RAM با تأخیر مشخصی انجام می‌شود که در طراحی لحاظ شده است (Latency).

- **واحد کنترل–FSM (ماشین حالت متناهی):**

  این ماژول نقش مغز سیستم را دارد. FSM با بررسی پکت دریافتی و سیگنال‌هایی مانند InputRdy، Function و OutputRdy مشخص می‌کند در هر لحظه کدام بخش سیستم باید فعال شود.
  برای مثال اگر پکتی با Function مربوط به ALU دریافت شود، FSM ابتدا داده‌ها را از RAM خوانده، سپس ALU را فعال کرده و در نهایت خروجی را در مقصد ذخیره می‌کند. این ماژول باعث می‌شود تمام اجزای سیستم به‌صورت هماهنگ و دقیق عمل کنند.

فصل سوم) طراحی و پیاده‌سازی سیستم ماژول‌ها

۳-۱. Packages (انواع داده و توابع مشترک)

این فایل نقش کتابخانه‌ی پشتیبان را دارد. همه‌ی ماژول‌ها برای یکپارچگی، انواع داده و توابع مشترک را از این فایل دریافت می‌کنند.

انواع داده (Types)

- byte: STD_LOGIC_VECTOR(7 downto 0) برای نمایش داده‌های ۸بیتی.
- hamming: بردار ۱۳ بیتی برای داده‌های کد همینگ.
- data_packet: آرایه‌ی ۷ بایتی، قالب استاندارد تبادل داده در کل سیستم.
- hamming_packet: آرایه‌ای از ۷ المان نوع hamming برای مدیریت فریم‌های سریال.
- packet_type: نوع شمارشی برای تشخیص دستور (zero, Operand_Alu, Writ_e, Rea_d, Immediate_Alu, Array_Alu, Indirect_Addressing).
- alu_operation: نوع شمارشی برای عملیات منطقی/حسابی (Add, Sub, BitwiseOr, BitwiseAnd).
- ram_matrix: آرایه‌ی ۳۲ خانه‌ای از نوع byte (RAM اصلی).
- ram_resp_pack: قالب ساده‌ی پکت پاسخ RAM.
- alu_read_cash_array: آرایه موقت برای عملیات‌های آرایه‌ای ALU.

```
1   -- Package File
2
3   -- In this file we will introduce different packages used.
4
5   library IEEE;
6   use IEEE.STD_LOGIC_1164.ALL;
7   use IEEE.NUMERIC_STD.ALL;
8
9   package Packages is
10
11     subtype byte is STD_LOGIC_VECTOR(7 downto 0);
12     subtype hamming is STD_LOGIC_VECTOR(0 to 12);
13     type hamming_packet is array (0 to 6) of hamming;
14     type data_packet is array (0 to 6) of byte;
15     type packet_type is (zero, Operand_Alu, Writ_e, Rea_d, Immediate_Alu, Array_Alu, Indirect_Addressing);
16     type alu_operation is (Add, Sub, BitwiseOr, BitwiseAnd);
17
18     --type ram_row is array (0 to 7) of byte;
19     type ram_matrix is array (0 to 31) of byte; --Ram_Row;
20     type ram_resp_pack is array (0 to 3) of byte;
21
22     type alu_read_cash_array is array (0 to 31) of byte;
23
24     function ByteSum (Packet : data_packet) return STD_LOGIC_VECTOR;
25     function CheckSumH (Packet : data_packet) return byte;
26     function CheckSumL (Packet : data_packet) return byte;
27     function Validate (Packet : data_packet) return STD_LOGIC;
28
29  end Packages;
```

توابع کمکی

- ByteSum(Packet): جمع پنج بایت اول یک Packet و بازگرداندن نتیجه‌ی ۱۶بیتی.
- CheckSumH(ByteSum): بایت بالای جمع.
- CheckSumL(ByteSum): بایت پایین جمع.
- Validate(Packet): بررسی صحت Checksum در پکت و بازگرداندن معتبر بودن یا نبودن آن.

```
31  package body Packages is
32
33     function ByteSum (Packet : data_packet) return STD_LOGIC_VECTOR is
34           variable Sum : integer := 0;
35     begin
36        for i in 0 to 4 loop
37           Sum := Sum + to_integer(signed(Packet(i)));
38        end loop;
39        return STD_LOGIC_VECTOR(to_signed(Sum, 16));
40     end function;
41
42     function CheckSumH (Packet : data_packet) return byte is
43           variable SumL : byte;
44     begin
45        SumL := ByteSum(Packet)(15 downto 8);
46        return SumL;
47     end function;
48
49     function CheckSumL (Packet : data_packet) return byte is
50           variable SumR : byte;
51     begin
52        SumR := ByteSum(Packet)(7 downto 0);
53        return SumR;
54     end function;
55
56     function Validate(Packet : data_packet) return STD_LOGIC is
57        variable CheckH, CheckL : byte;
58     begin
59        CheckH := CheckSumH(Packet);
60        CheckL := CheckSumL(Packet);
61        if (CheckH = Packet(5) and CheckL = Packet(6)) then
62           return '1';
63        else
64           return '0';
65        end if;
66     end function;
67
68
69  end Packages;
```

نحوهٔ استفاده در سایر ماژول‌ها

- در **ControlUnit** از packet_type برای تعیین نوع بسته و از Validate برای کنترل صحت بسته استفاده می‌شود.
- در **RAM** برای ساختن پکت پاسخ از CheckSumH و CheckSumL استفاده می‌شود.
- در **ALU** برای ساخت پکت‌های RAM-Write و بررسی طول آرایه از نوع داده‌ها وenum های این فایل کمک گرفته می‌شود.
- در **TopModule** همه‌ی ماژول‌ها به‌صورت مستقیم از این کتابخانه استفاده می‌کنند.

۲-۳. ماژول Hamming Decoder

هدف: دریافت سریالی کُدهای ۱۳بیتی همینگ، تشخیص و تصحیح تک‌خطا، و تحویل بایت ۸بیتی معتبر + اعلام وضعیت اعتبار خروجی.

ورودی/خروجی‌ها (Entity HammingDecoder)

- ورودی‌ها:

  DecInBit: std_logic (جریان سریال ۱۳ بیت)، CodeBegins: std_logic (آغاز فریم)، RST، clk

- خروجی‌ها:

  DecOutByte: byte (۸ بیت هم‌زمان)، OutRdy: std_logic (آمادگی خروجی)، Valid: std_logic (اعتبار پس از تصحیح)

- همچنین پورت کمکی TestBenchInputDisplay برای مشاهدهٔ ورودی کد.

نحوهٔ عملکرد ماژول دیکودر همینگ:

- رجیستر داخلی Code: hamming و شمارندهٔ cnt بیت‌ها را جمع می‌کنند.
- ورود: برای cnt<13 هر کلاک Code <= Code(1 to 12) & DecInBit.
- محاسبه: (cnt=13) بیت‌های توازن بازساخته می‌شوند ←

```
15  library IEEE;
16  use IEEE.STD_LOGIC_1164.ALL;
17  use IEEE.NUMERIC_STD.ALL;
18  use work.Packages.ALL;
19
20  entity HammingDecoder is
21      Port ( CodeBegins : in STD_LOGIC;
22             DecInBit : in STD_LOGIC;
23             TestBenchInputDisplay : out hamming;
24             OutRdy : inout  STD_LOGIC;
25             DecOutByte : out byte;
26
27             Valid : inout STD_LOGIC;
28             RST : in STD_LOGIC;
29             clk : in STD_LOGIC
30             );
31  end HammingDecoder;
32
33  architecture Behavioral of HammingDecoder is
34
35     signal Code : hamming := (others => '0');
36     signal cnt : integer := 0;
38  begin
39      process (clk)
40         variable ind : integer := 0;  -- possible error position
41         variable ErrorMarker : STD_LOGIC_VECTOR (3 downto 0) := (others => '0');
42         variable ExtentionBit : STD_LOGIC := '0';                          -- rem
43      begin
44         if rising_edge(clk) then
45            OutRdy <= '0';
46            Valid <= '1';
47            if (RST = '1' or CodeBegins = '1') then
48               ErrorMarker := "0000";
49               Code <= "0000000000000";
50               cnt <= 0;
51               ind := 0;
52            else
53               if cnt < 13 then
54                  Code <= Code(1 to 12) & DecInBit;
55                  cnt <= cnt + 1;
56               elsif cnt = 13 then
57                  -- Constructing the new parities
58                  TestBenchInputDisplay <= Code;
59                  ErrorMarker(0) := Code(0) xor (Code(2) xor (Code(4) xor
60                                    (Code(6) xor (Code(8) xor Code(10)))));
61                  ErrorMarker(1) := Code(1) xor (Code(2) xor (Code(5) xor
62                                    (Code(6) xor (Code(9) xor Code(10)))));
63                  ErrorMarker(2) := Code(3) xor (Code(4) xor (Code(5) xor
64                                    (Code(6) xor Code(11))));
65                  ErrorMarker(3) := Code(7) xor (Code(8) xor (Code(9) xor
66                                    (Code(10) xor Code(11))));
67                  ExtentionBit := Code(0) xor (Code(1) xor (Code(2) xor
68                                    (Code(3) xor (Code(4) xor (Code(5) xor
69                                    (Code(6) xor (Code(7) xor (Code(8) xor
70                                    (Code(9) xor (Code(10) xor Code(11)))))))))));
71
72                  cnt <= cnt + 1;
73                  -- Correction
74                  ind :=  to_integer(unsigned(ErrorMarker));
75                  ind := ind - 1;
76                  if (ind > -1 and ind < 13) then
77                        Code(ind) <= not Code(ind);
78                  end if;
```

- سپس ind := to_integer(unsigned(ErrorMarker)) – 1 و اگر 0≤ind<13، بیت خراب **flip** می‌شود.

شرط اعتبار: اگر (بدون خطا و ExtentionBit = Code(12)) یا (تک‌خطا و ExtentionBit = not Code(12))، آنگاه معتبر؛ در غیر این صورت Valid <= '0'.

- خروج: در فاز بعدی خروجی موازی:

```
80                  if not((ind = -1 and ExtentionBit = Code(12)) or
81                       ((ind > -1 and ind < 12) and (ExtentionBit = not(Code(12))))) then
82                     Valid <= '0';
83                  end if;
84               else                                                          -- 1 * C
85                  cnt <= 0;
86                  Code <= (others => '0');
87                  DecOutByte <= code(2) & code(4) & code (5) & code(6) &
88                                code(8) & code(9) & code(10) & code(11);
89                  OutRdy <= '1';
90               end if;
91            end if;
92         end if;
93      end process;
94   end Behavioral;
```

زمان/کلاک: ۱۴ کلاک طول می‌کشد تا دریافت کند و خروجی بدهد.

ارتباط با سایر ماژول‌ها: DecOutByte/OutRdy/Valid به ControlUnit می‌روند.

۳-۳. ماژول Control Unit

هدف: تبدیل جریان بایت‌های ورودی به **Packet**(آرایه ۷بایتی) + تشخیص نوع عملیات + (RAM/ALU) اعتبارسنجی **Checksum** و ارسال به مسیر درست.

ورودی/خروجی (Entity ControlUnit)

```
11  library IEEE;
12  use IEEE.STD_LOGIC_1164.ALL;
13  use IEEE.NUMERIC_STD.ALL;
14  use work.Packages.ALL;
15
16  entity ControlUnit is
17      Port ( InputIsReady : in STD_LOGIC;
18             InByte : in byte;
19
20             Switch : out STD_LOGIC;
21             Packet : inout data_packet;
22             PackType : inout packet_type;
23
24             PackIsReady : inout STD_LOGIC;
25             Validation : out STD_LOGIC;
26             clk : in STD_LOGIC;
27             RST : in STD_LOGIC
28           );
29  end ControlUnit;
```

- ورودی‌ها:

InByte: byte ، InputIsReady: std_logic ،

clk ،RST

- خروجی‌ها:

Packet: inout data_packet ،PackType:

Switch: std_logic ،inout packet_type،

Validation: std_logic ،PackIsReady:

std_logic

نحوه‌ی عملکرد ماژول واحد کنترل:

```
31  architecture Behavioral of ControlUnit is
32
33     signal Step : integer := -1;
34
35  begin
36
37     process (clk)
38        variable PackToCheck : data_packet := (others => (others => '0'));
39     begin
40        if rising_edge(clk) then
41           PackIsReady <= '0';
42
43           if RST = '1' then
44              Step <= 0;
45              PackType <= zero;
46
47           elsif InputIsReady = '1' then
48              case step is
49                 when 0 =>
50                    Packet <= (others => (others => '0'));
51                    Packet(0) <= InByte;
52
53                    ---
54                    case InByte is
55                          when "00000000" | "00000001" | "00000010" | "00000011" =>
56                             PackType <= Operand_Alu;
57                          when "11110000" =>
58                             PackType <= Writ_e;
59                          when "00001111" =>
60                             PackType <= Rea_d;
61                          when "00111100" | "00111101" | "00111110" | "00111111" =>
62                             PackType <= Immediate_Alu;
63                          when "11000000" | "11000001" | "11000010" | "11000011" =>
64                             PackType <= Array_Alu;
65                          when "00110000" | "00110001" | "00110010" | "00110011" =>
66                             PackType <= Indirect_Addressing;
67                          when others =>
68                             PackType <= zero;
69                       end case;
70                    ---
71                    Step <= Step + 1;
```

- **Step=0**: پاک‌سازی Packet و ثبت Packet(0) := InByte، سپس

دسته‌بندی Function :

  - Operand_Alu برای "00000000" تا "00000011"
  - Writ_e برای "11110000"، Rea_d برای "00001111"
  - Immediate_Alu برای "00111100" تا "00111111"
  - Array_Alu برای "11000000" تا "11000011"
  - Indirect_Addressing برای "00110000" تا "00110011"

```
73              when 1 =>
74                 Packet(1) <= InByte;
75                 if PackType = Rea_d then
76                    Step <= 5;
77                 else
78                    Step <= Step + 1;
79                 end if;
80
81              when 2 =>
82                 Packet(2) <= InByte;
83                 if PackType = Writ_e then
84                    Step <= 5;
85                 else
86                    Step <= Step + 1;
87                 end if;
88
89              when 3 =>
90                 Packet(3) <= InByte;
91                 if ((PackType = Operand_Alu or PackType = Immediate_Alu) or
92                       PackType = Indirect_Addressing) then
93                    Step <= 5;
94                 else
95                    Step <= Step + 1;
96                 end if;
97
98              when 4 =>
99                 Packet(4) <= InByte;
100                Step <= Step + 1;
101
102             when 5 =>
103                Packet(5) <= InByte;
104                Step <= Step + 1;
```

- **Step=1**: Packet(1) := InByte؛ اگر Rea_d بود، **پرش به Step=5** (چون فقط تا بایت ۴ نیاز است.)
- **Step=2**: Packet(2) := InByte؛ اگر Writ_e بود، **پرش به Step=5** .
- **Step=3**: Packet(3) := InByte؛ اگر نوع پکت یکی از Operand/Immediate/Indirect بود، **پرش به Step=5** .
- **Step=4**: Packet(4) := InByte و ادامه.
- **Step=5**: Packet(5) := InByte .
- **Step=6**: Packet(6) := InByte ؛ سپس:

```
105
106             when 6 =>
107                Packet(6) <= InByte;
108                if (PackType = Writ_e or PackType = Rea_d) then
109                   Switch <= '0';
110                else
111                   Switch <= '1';
112                end if;
113                PackToCheck := Packet;
114                PackToCheck(6) := InByte;
115                Step <= 0;
116                Validation <= Validate(PackToCheck);
117                PackIsReady <= '1';
118
119             when others =>
120
121          end case;
122       end if;
123    end if;
124 end process;
125
126
127 end Behavioral;
```

ارتباط با سایر ماژول‌ها: خروجی پکت و نوع آن به RAM/ALU می‌رود؛ Switch مسیر را تعیین می‌کند.

۳-۴. ماژول RAM

هدف: RAM با ۳۲ خانهٔ ۸بیتی؛ پشتیبانی Read/Write مبتنی بر پکت و ساخت پاسخ استاندارد خواندن با Checksum.

ورودی/خروجی (Entity RAM):

دو درگاه مستقل برای CU و ALU:

ورودی‌های CtrlInputRdy, CtrlReq و خروجی‌های CtrlReadResp, CtrlReadRespReady؛

ورودی‌های AluInputRdy, AluReq و خروجی‌های AluReadResp, AluReadRespReady؛

همچنین Error, RST, clk.

نحوهٔ عملکرد ماژول رم:

- **شاخهٔ CU (CtrlInputRdy='1'):**
  - تشخیص Mode از CtrlReq(0) (Read="00001111", Write="11110000").
  - **کنترل آدرس:** اگر to_integer(unsigned(CtrlReq(1))) > 31 ⇒ Error <= '1'.
  - **Write:** Memory(RowAddress) <= CtrlReq(2).

```
11  library IEEE;
12  use IEEE.STD_LOGIC_1164.ALL;
13  use IEEE.NUMERIC_STD.ALL;
14  use work.Packages.ALL;
15
16  entity RAM is
17      Port (  CtrlInputRdy : in STD_LOGIC;
18              CtrlReq : in  data_packet;
19              CtrlReadResp : out data_packet;
20              CtrlReadRespReady : out STD_LOGIC;
21              AluInputRdy : in STD_LOGIC;
22              AluReq : in  data_packet;
23              AluReadResp : out data_packet;
24              AluReadRespReady : out STD_LOGIC;
25              Error : out STD_LOGIC;
26              RST : in  STD_LOGIC;
27              clk : in  STD_LOGIC
28            );
29  end RAM;
```

```
31  architecture Behavioral of RAM is
32
33     signal Memory : ram_matrix := (others => (others => '0')); --(others => '0')))
34
35  begin
36
37     process(clk)
38        variable Mode : packet_type := zero;
39        variable RowAddress : integer range 0 to 31 := 0;
40  --    variable ColAddress : integer range 0 to 7;
41        variable WriteData : byte := (others => '0');
42        variable cash : data_packet := (others => (others => '0'));
43     begin
44        if rising_edge(clk) then
45           Error <= '0';
46           CtrlReadRespReady <= '0';
47           AluReadRespReady <= '0';
48
49           if RST = '1' then
50              Memory <= (others => (others => '0'));
51              WriteData := (others => '0');
52              cash := (others => (others => '0'));
53           else
54              if CtrlInputRdy = '1' then
55                 if CtrlReq(0) = "00001111" then  -- Function
56                    Mode := Rea_d;
57                 elsif CtrlReq(0) = "11110000" then
58                    WriteData := CtrlReq(2);
59                    Mode := Writ_e;
60                 else
61                    Mode := zero;
62                 end if;
63
64                 RowAddress := to_integer(unsigned(CtrlReq(1))); --(7 downto 3)));
65     --          ColAddress := to_integer(unsigned(InPack(1)(2 downto 0)));
66                 if (to_integer(unsigned(CtrlReq(1))) > 31) then
67                    Error <= '1';
```

```
            if (to_integer(unsigned(CtrlReq(1))) > 31) then
               Error <= '1';
            else
               case Mode is
                  when Writ_e =>
                     -- Write Operation
                     Memory(RowAddress) <= WriteData;
                  when Rea_d =>
                     -- Read Operation
                     cash(0) := "11001111";
                     cash(1) := Memory(RowAddress);
                     cash(2) := (others => '0');
                     cash(3) := (others => '0');
                     cash(4) := (others => '0');
                     cash(5) := CheckSumH(cash);
                     cash(6) := CheckSumL(cash);
                     CtrlReadResp <= cash;
                     CtrlReadRespReady <= '1';
                  when others =>
               end case;
            end if;
         end if;
```

- **Read**: ساخت پاسخ cash(0)="11001111", cash(1)=Memory(RowAddress), cash(2..4)=0، سپس cash(5):=CheckSumH(cash), و cash(6):=CheckSumL(cash) اعلام CtrlReadRespReady<='1'.

- **شاخهٔ ALU (AluInputRdy='1')**:
  - تشخیص Mode مشابه؛ RowAddress := to_integer(unsigned(AluReq(1)(4 downto 0)))(۵ بیت پایین به‌عنوان آدرس)؛
  - Write/Readهمانند بالا؛ پاسخ در AluReadResp و AluReadRespReady.

```
         if AluInputRdy = '1' then
            if AluReq(0) = "00001111" then   -- Function
               Mode := Rea_d;
            elsif AluReq(0) = "11110000" then
               WriteData := AluReq(2);
               Mode := Writ_e;
            else
               Mode := zero;
            end if;

            RowAddress := to_integer(unsigned(AluReq(1)(4 downto 0)));
            case Mode is
               when Writ_e =>
                  -- Write Operation
                  Memory(RowAddress) <= WriteData;
               when Rea_d =>
                  -- Read Operation
                  cash(0) := "11111111";
                  cash(1) := Memory(RowAddress);
                  cash(2) := (others => '0');
                  cash(3) := (others => '0');
                  cash(4) := (others => '0');
                  cash(5) := CheckSumH(cash);
                  cash(6) := CheckSumL(cash);
                  AluReadResp <= cash;
                  AluReadRespReady <= '1';
               when others =>
            end case;
         end if;
      end if;
   end if;
end process;

end Behavioral;
```

زمان/کلاک: پاسخ Read در همان سیکل فعال شدن InputRdy تولید و Ready می‌شود و سپس با شروع کلاک دوم خروجی داده می‌شود.

ارتباط با سایر ماژول‌ها: ورودی از CU/ALU، پاسخ خواندن به مسیر خروجی (PackToByte) و نیز به ALU برمی‌گردد.

۳-۵. ماژول ALU

هدف ALU: اجرای Add/Sub/Or/And در چهار حالت بسته به PackMode و برگرداندن نتیجه به RAM به‌صورت پکت Write/Read.

ورودی/خروجی (Entity ALU):

```
10  library IEEE;
11  use IEEE.STD_LOGIC_1164.ALL;
12  use IEEE.NUMERIC_STD.ALL;
13  use work.Packages.All;
14
15  entity ALU is
16      Port ( InputRdy : in STD_LOGIC;
17             InPack : in  data_packet;
18             PackMode : in  packet_type;
19
20             SendToRam : out data_packet;
21             SendToRamReady : out STD_LOGIC;
22             ReadResponse : in data_packet;
23             ReadResponseRdy : STD_LOGIC;
24             Finish : inout STD_LOGIC;
25
26             Enable : inout STD_LOGIC;
27             Error : out STD_LOGIC;
28             RST : in STD_LOGIC;
29             clk : in STD_LOGIC
30           );
31  end ALU;
```

ورودی‌ها

- clk, RST: کلاک و ریست همگام.
- Enable: std_logic ← فعال‌سازی ALU؛ تا وقتی عملیات قبلی تمام نشده یا در حالت ALU هستیم فعال است.
- InputRdy: std_logic← اعلام «پکت ورودی آماده است.»
- InPack: data_packet (7×byte) ← پکت دستور و پارامترها: کد عمل (۲ بیت پایین بایت صفر) + آدرس/داده/طول/... بسته به حالت.
- PackMode: packet_type← تعیین حالت عملیات : Operand_Alu / Immediate_Alu / Array_Alu / Indirect_Addressing.
- ReadResponse: data_packet← پاسخ‌های خواندن از RAM (وقتی ALU چیزی از RAM می‌خواهد.)
- ReadResponseRdy: std_logic← اعلام «پاسخ خواندن از RAM حاضر است.»

خروجی‌ها

- SendToRam: data_packet← پکت دستوری به RAM (Read/Write) با فیلدهای استاندارد ([0]=mode, [1]=addr, [2]=data, ...).
- SendToRamReady: std_logic← «درخواست ALU به RAM آماده‌ی ارسال است.»
- Finish: std_logic← پایان عملیات ALU در این سری.
- Error: std_logic ← خطاهای منطقی/پروتکلی (مثلاً طول آرایه <۳۲، آدرس/مد نامعتبر و...).

نحوهٔ عملکرد ماژول:

- **انتخاب عمل:** از InPack(0)(1 downto 0)

"00"→Add, "01"→Sub, "10"→BitwiseOr, "11"→BitwiseAnd.

```
33  architecture Behavioral of ALU is
34
35     function Operator(In1, In2: signed(7 downto 0); operate : Alu_Operation) return byte is
36     begin
37        case operate is
38           when Add =>
39              return byte(In1 + In2);
40           when Sub =>
41              return byte(In1 - In2);
42           when BitwiseOr =>
43              return byte(In1 or In2);      -- Bitwise or for signed = direct or
44           when BitwiseAnd =>
45              return byte(In1 and In2);     -- Bitwise and for signed = direct and
46           when others =>
47              return "00000000";
48        end case;
49     end function;
50
51     ---
52     signal DataI : byte := (others => '0');
53     signal DataII : byte := (others => '0');
54     signal Operation : Alu_Operation;
55
56     signal DestinationAddress : byte;
57     signal ArrayLength : integer range 0 to 32 := 0;
58     signal ArrayIndPusher : integer range 0 to 31 := 0;
59
60     signal ReadArray : alu_read_cash_array := (others => (others => '0'));
61     signal ArrayWriteDone : STD_LOGIC := '1';
62
63     signal Step : integer := 0;
```

- در ابتدای هر دور: Finish<='0', Error<='0', SendToRamReady<='0'

```
65  begin
66
67     process(clk)
68        -- Ram Interaction
69        variable mode : byte := (others => '0');
70        variable RamAddress : byte := (others => '0');
71        variable RamDataToWrite : byte := (others => '0');
72        variable AddressI : byte := (others => '0');
73        variable AddressII : byte := (others => '0');
74        variable AddAddressII : byte := (others => '0');
75
76        -- Calculator Interaction
77
78     begin
79        if rising_edge(clk) then
80           Finish <= '0';
81           Error <= '0';
82           SendToRamReady <= '0';
83
84           if RST = '1' then
85              Step <= 0;
86              DataI <= (others => '0');
87              DataII <= (others => '0');
88              DestinationAddress <= (others => '0');
89              ArrayLength <= 0;
90              ArrayIndPusher <= 0;
91              ReadArray <= (others => (others => '0'));
92              ArrayWriteDone <= '1';
```

```
94          elsif ((InputRdy = '1' or ReadResponseRdy = '1') or ArrayWriteDone = '0') then
95             if Step = 0 then           -- 1 * Clk ->
96                ReadArray <= (others => (others => '0'));
97                case InPack(0)(1 downto 0) is
98                   when "00" =>
99                      Operation <= Add;
100                  when "01" =>
101                     Operation <= Sub;
102                  when "10" =>
103                     Operation <= BitwiseOr;
104                  when "11" =>
105                     Operation <= BitwiseAnd;
106                  when others =>
107               end case;
108            end if;
109
110            case PackMode is
111               when Operand_Alu => -- 3 Clocks
112                  if Step = 0 then                    -- Clock 0 till 1 -> Input
113                     DestinationAddress <= InPack(3);
114                     AddressI := InPack(1);
115                     RamAddress := AddressI;
116                     mode := "00001111";
117                     Step <= 1;
118                  elsif Step = 1 then                 -- Clock 1 till 2
119                     DataI <= ReadResponse(1);
120                     AddressII := InPack(2);
121                     RamAddress := AddressII;
122                     mode := "00001111";
123                     Step <= 2;
124                  elsif Step = 2 then                 -- Clock 2 till 3
125                     RamDataToWrite := Operator(signed(DataI), signed(ReadResponse(1)), Operation);
126                     RamAddress := DestinationAddress;
127                     mode := "11110000";
128                     Step <= 0;
129                     Finish <= '1';
130                  end if;
```

- **Operand_Alu (۳ کلاک):**

۱) Read از آدرس InPack(1)

(mode="00001111").

۲) پس از ReadResponseRdy='1'، DataI := ReadResponse(1)، سپس Read از InPack(2).

۳) نتیجهٔ Operator(signed(DataI), signed(ReadResponse(1)), Operation) در آدرس InPack(3) نوشته می‌شود (mode="11110000")، Finish<='1'.

- **Immediate_Alu (۲ کلاک):**

Read از InPack(1)، سپس اعمال عملیات با دادهٔ فوری InPack(2) و Write در InPack(3).

```
131
132               when Immediate_Alu =>
133                  if Step = 0 then                    -- Clock 0 till 1
134                     DataII <= InPack(2);
135                     DestinationAddress <= InPack(3);
136                     AddressI := InPack(1);
137                     RamAddress := AddressI;
138                     mode := "00001111";
139                     Step <= Step + 1;
140                  elsif Step = 1 then                 -- Clock 1 till 2
141                     --DataI <= ReadResponse(1);
142                     RamDataToWrite := Operator(signed(ReadResponse(1)), signed(DataII), Operation);
143                     RamAddress := DestinationAddress;
144                     mode := "11110000";
145                     Step <= 0;
146                     Finish <= '1';
147                  end if;
```

- **Array_Alu(طول آرایه × ۳ کلاک):**

DataII := InPack(2)، ArrayLength := to_integer(unsigned(InPack(3)))، مقصد InPack(4)، اگر Error<='1' ⇒ 32<طول.

حلقهٔ خواندن پشت‌سرهم از مبدا و نوشتن نتایج از آدرس مقصد، با شمارنده‌های داخلی ArrayIndPusher؛
انتها: Finish<='1'.

```
149                when Array_Alu =>
150                   DataII <= InPack(2);
151                   ArrayLength <= to_integer(unsigned(InPack(3)));
152                   DestinationAddress <= InPack(4);
153                   if to_integer(unsigned(InPack(3))) > 32 then
154                      Error <= '1';
155                   end if;
156
157                   if (Step < ArrayLength + 1) then
158                      if Step = 0 then                    -- Clock * size -> max 32
159                         ReadArray <= (others => (others => '0'));
160                         Step <= Step + 1;
161                      elsif (Step > 0 and ReadResponseRdy = '1') then
162                         ReadArray(Step - 1) <= ReadResponse(1);   -- Till A.L.-2
163                         Step <= Step + 1;
164                         ArrayIndPusher <= ArrayIndPusher + 1;
165                      end if;
166                      mode := "00001111";
167                      RamAddress := byte(unsigned(InPack(1)) + to_unsigned(ArrayIndPusher, 8));
168                      if Step = ArrayLength then
169                         ArrayIndPusher <= 0;
170                         ArrayWriteDone <= '0';
171                      end if;
172
173                   elsif (Step > ArrayLength)  then       -- Clock * size -> max 32
174                      mode := "11110000";
175                      RamDataToWrite := Operator(signed(ReadArray(ArrayIndPusher)), signed(DataII), Operation);
176                      RamAddress := byte(unsigned(DestinationAddress) + to_unsigned(ArrayIndPusher, 8));
177                      ArrayIndPusher <= ArrayIndPusher + 1;
178                      Step <= Step + 1;
179                      if ArrayIndPusher = (ArrayLength - 1) then
180                         Step <= 0;
181                         Finish <= '1';
182                         ArrayIndPusher <= 0;
183                         ArrayWriteDone <= '1';
184                      end if;
185                   end if;
```

- **Indirect_Addressing (۴ کلاک):**

۱) Read از InPack(1) برای گرفتن آدرس غیرمستقیم؛

۲) AddAddressII := InPack(2), DataI := ReadResponse(1)و Read دوم؛

۳) اعمال عملیات و Write در InPack(3)؛

۴) Finish<='1'.

در ماژول ALU به علت اینکه خروجی در یک بایت ۸ بیتی ذخیره می‌شود، به مشکل Overflow برنخواهیم خورد.

```
186
187                when Indirect_Addressing =>
188                   if Step = 0 then                                    -- Clock 0 till 1
189                      DestinationAddress <= InPack(3);
190                      AddressI := InPack(1);
191                      RamAddress := AddressI;
192                      mode := "00001111";
193                      Step <= Step + 1;
194                   elsif Step = 1 then                                 -- Clock 1 till 2
195                      AddAddressII := InPack(2);
196                      DataI <= ReadResponse(1);
197                      mode := "00001111";
198                      RamAddress := AddAddressII;
199                      Step <= Step + 1;
200                   elsif Step = 2 then                                 -- Clock 2 till 3
201                      AddressII := ReadResponse(1);
202                      RamAddress := AddressII;
203                      mode := "00001111";
204                      Step <= Step + 1;
205                   elsif Step = 3 then                                 -- Clock 3 till 4
206                      --DataII <= ReadResponse(1);
207                      RamDataToWrite := Operator(signed(DataI), signed(ReadResponse(1)), Operation);
208                      RamAddress := DestinationAddress;
209                      mode := "11110000";
210                      Step <= 0;
211                      Finish <= '1';
212                   end if;
213                when others =>
214                   Error <= '1';
215             end case;
216             SendToRam(0) <= mode;
217             SendToRam(1) <= RamAddress;
218             SendToRam(2) <= RamDataToWrite;
219             SendToRam(3) <= (others => '0');
220             SendToRam(4) <= (others => '0');
221             SendToRam(5) <= (others => '0');
222             SendToRam(6) <= (others => '0');
223             SendToRamReady <= '1';
224          end if;
225       end if;
226    end process;
227
228 end Behavioral;
```

۳-۶. ماژول Error Detection

هدف: تجمیع خطاها:

Error <= (RamError or AluError) or (DecodingError or PacketError).

ورودی/خروجی: ورودی چهار پرچم خطا؛ خروجی Error.

ارتباط با سایر ماژول‌ها: خروجی به Top برای مانیتورینگ/توقف مسیر.

```
 8 library IEEE;
 9 use IEEE.STD_LOGIC_1164.ALL;
10 use IEEE.NUMERIC_STD.ALL;
11 use work.Packages.ALL;
12
13 entity ErrorDetection is
14     Port ( DecodingError : in  STD_LOGIC;
15            PacketError : in  STD_LOGIC;
16            RamError : in  STD_LOGIC;
17            AluError : in  STD_LOGIC;
18            Error : out  STD_LOGIC
19            );
20 end ErrorDetection;
21
22 architecture Behavioral of ErrorDetection is
23
24 begin
25
26    Error <= (RamError or AluError) or (DecodingError or PacketError);
27
28 end Behavioral;
```

۳-۷. ماژول PackToByte

هدف: تبدیل پاسخ خواندن RAM به جریان بایت برای سریال‌سازی.

ورودی‌ها

- clk
- PackIn: data_packet ← پکت پاسخ خواندن RAM (۷ بایت: کد پاسخ، داده، رزرو، CheckSumH/L).

خروجی‌ها

- ByteOut: byte (۸ بیت).

تبدیل پکت ورودی به چهار بایت خروجی ترتیبی برای سریال‌سازی

نحوهٔ عملکرد ماژول:

داخل ماژول یک بافر ۴ خانه‌ای تشکیل می‌شود (PacketCash)و در هر کلاک یکی از بایت‌ها روی خروجی قرار می‌گیرد.

ارتباط با سایر ماژول‌ها: خروجی به ByteToBit می‌رود.

```
10  library IEEE;
11  use IEEE.STD_LOGIC_1164.ALL;
12  use IEEE.NUMERIC_STD.ALL;
13  use work.Packages.ALL;
14
15  entity PackToByte is
16      Port ( PackIn : in  data_packet;
17             ByteOut : out  byte;
18             clk : in  STD_LOGIC);
19  end PackToByte;
20
21  architecture Behavioral of PackToByte is
22
23     signal CellCnt : integer range 0 to 4 := 4;
24     signal PacketCash : ram_resp_pack ;
25
26  begin
27
28     PacketCash(0) <= "11001111";
29
30     process(clk)
31     begin
32        if rising_edge(clk) then
33           if CellCnt = 4 then
34              CellCnt <= 1;
35              PacketCash(1) <= PackIn(1);
36              PacketCash(2) <= PackIn(2);
37              PacketCash(3) <= PackIn(6);
38              ByteOut <= PacketCash(0);
39           else
40              ByteOut <= PacketCash(CellCnt);
41              CellCnt <= CellCnt + 1;
42           end if;
43        end if;
44     end process;
45
46  end Behavioral;
```

۳-۸. ماژول ByteToBit

هدف: تبدیل یک بایت به ۸ بیت سریالی (LSB→MSB) برای ورودی انکودر.

```
10  library IEEE;
11  use IEEE.STD_LOGIC_1164.ALL;
12  use IEEE.NUMERIC_STD.ALL;
13  use work.Packages.ALL;
14
15  entity ByteToBit is
16      Port ( ByteIn : in  byte;
17             BitOut : out  STD_LOGIC;
18
19             clk : in  STD_LOGIC);
20  end ByteToBit;
21
22  architecture Behavioral of ByteToBit is
23
24     signal BitCnt : integer range 0 to 8 := 8;
25     signal ByteCash : byte;
26
27  begin
28
29     process(clk)
30     begin
31        if rising_edge(clk) then
32           if BitCnt = 8 then
33              BitCnt <= 1;
34              ByteCash <= ByteIn;
35              BitOut <= ByteCash(0);
36           else
37              BitOut <= ByteCash(BitCnt);
38              BitCnt <= BitCnt + 1;
39           end if;
40        end if;
41     end process;
42
43  end Behavioral;
```

ورودی/خروجی (Entity ByteToBit):

ورودی‌ها

- clk
- ByteIn: byte ← بایتی که باید به ۸ بیت سریال شکسته بشود.

خروجی‌ها

- BitOut: std_logic ← جریان سریالی ۸ بیت به ترتیب (LSB→MSB).

نحوهٔ عملکرد ماژول:

با رسیدن بایت جدید (وقتی شمارنده ۸ است) لچ می‌شود و در ۸ کلاک بعدی بیت‌ها یکی‌یکی روی خروجی می‌آیند.

ارتباط با سایر ماژول‌ها: BitOut مستقیماً به HammingEncoder می‌رود.

۳-۹. ماژول Hamming Encoder

هدف: دریافت سریالی ۸ بیت داده و تولید کد همینگ ۱۳بیتی و ارسال سریالی.

ورودی/خروجی (Entity HammingEncoder):

ورودی‌ها

- clk, RST
- BitIn: std_logic ← بیت‌های سریالی داده (۸ بیت برای هر فریم).

خروجی‌ها

- BitOut: std_logic ← کد همینگ ۱۳بیتی به‌صورت سریال (پس از محاسبه‌ی بیت‌های توازن و کلی).
- OutRdy: std_logic ← «فریم کدگذاری‌شده آماده‌ی ارسال است» (هم‌زمان با سیکل‌های خروجی ۱۳ بیت).

```
12  library IEEE;
13  use IEEE.STD_LOGIC_1164.ALL;
14  use IEEE.NUMERIC_STD.ALL;
15  use work.Packages.All;
16
17  entity HammingEncoder is
18      Port ( BitIn : in  STD_LOGIC;       -- Input data
19             BitOut : out  STD_LOGIC;     -- Hamming coded data
20             TestBenchCheck : out hamming;
21             TestBenchInputDisplay : out byte;
22             OutRdy : inout STD_LOGIC := '0';
23
24             RST : in  STD_LOGIC;              -- Setting everything to the default
25             clk : in STD_LOGIC                -- Receiving sequential data
26           );
27  end HammingEncoder;
```

نحوهٔ عملکرد ماژول انکودر همینگ:

- شمارنده‌های InCnt/OutCnt؛ رجیسترهای InCode: byte و Encoded: hamming.
- ورود: ۸ کلاک دریافت بیت‌ها در InCode.
- ساختار کد: داده‌ها در موقعیت‌های ۲,۴,۵,۶,۸,۹,۱۰,۱۱ قرار می‌گیرند؛ بیت‌های توازن در ۰,۱,۳,۷ و بیت کلی در ۱۲ محاسبه و قرار می‌گیرد (ماژول Encoder :با «Odd mode = 0» پیاده‌سازی شده است).

```
architecture Behavioral of HammingEncoder is

   signal InCode : byte := (others => '0');
   signal Encoded : hamming := (others => '0');
   signal InCnt : integer := 0;
   signal OutCnt : integer := 0;

begin

   process (clk)
   begin
      if rising_edge(clk) then
         if (RST = '1') then
            InCode <= (others => '0');
            Encoded <= (others => '0');
            InCnt <= 0;
            OutCnt <= 0;
            OutRdy <= '0';
         else
            if InCnt < 8 then                              -- 8 * Clk -> Input
               InCode <= BitIn & InCode(7 downto 1);
               InCnt <= InCnt + 1;
            elsif InCnt = 8 then                           -- 1 * Clk -> Calculation
               TestBenchInputDisplay <= InCode;
               Encoded(2) <= InCode(7);
               Encoded(4) <= InCode(6);
               Encoded(5) <= InCode(5);
               Encoded(6) <= InCode(4);
               Encoded(8) <= InCode(3);
               Encoded(9) <= InCode(2);
               Encoded(10) <= InCode(1);
               Encoded(11) <= InCode(0);
               -- Assigning parities
               Encoded(0) <= (((InCode(7) xor InCode(6)) xor InCode(4)) xor InCode(3)) xor InCode(1);
               Encoded(1) <= (((InCode(7) xor InCode(5)) xor InCode(4)) xor InCode(2)) xor InCode(1);
               Encoded(3) <= ((InCode(6) xor InCode(5)) xor InCode(4)) xor InCode(0);
               Encoded(7) <= ((InCode(3) xor InCode(2)) xor InCode(1)) xor InCode(0);
               Encoded(12) <= ((((InCode(7) xor InCode(6)) xor InCode(5)) xor InCode(3)) xor InCode(2)) xor InCode(0);
               InCnt <= InCnt + 1;
```

- خروج: در حالت Output، BitOut<=Encoded(OutCnt) از ۰ تا ۱۲ و سپس ریست شمارنده‌ها؛ OutRdy اعلام می‌شود.

زمان/کلاک: ۱۴ کلاک طول می‌کشد تا دریافت کند و خروجی بدهد.

خروجی نهایی سریالی سیستم از این ماژول دریافت می‌شود.

```
            else
               TestBenchCheck <= Encoded;
               OutRdy <= '1';
               if OutCnt < 13 then
                  BitOut <= Encoded(OutCnt);
                  OutCnt <= OutCnt + 1;
               else
                  InCnt <= 0;
                  OutCnt <= 0;
                  OutRdy <= '0';
               end if;
            end if;
         end if;
      end if;
   end process;

end Behavioral;
```

۳-۱۰. Top Module (تجمیع و ارتباط ماژول‌ها)

هدف: اتصال و زمان‌بندی بین مسیر ورودی سریال کد شده تا خروجی سریال. این ماژول جریان داده را از

HammingDecoder → ControlUnit → (RAM/ALU) → PackToByte → ByteToBit → HammingEncoder

مدیریت می‌کند، میانبرها را هماهنگ می‌کند (Enable/Finish برای ALU، انتخاب ورودی RAM)، و پرچم خطای کلی را از ErrorDetection ارائه می‌دهد.

ورودی/خروجی (Entity TopModule):

- Input: in std_logic ← بیت سریالی ورودی (کد همینگ ۱۳ بیتی به‌ازای هر بایت داده).
- InputBegining: in std_logic ← نبض «شروع فریم» برای دیکودر؛ با ۱ شدن، جمع‌آوری ۱۳ بیت بعدی آغاز می‌شود.
- Output: out std_logic ← بیت سریالی خروجی از انکودر (۱۳ بیت به‌ازای هر بایت خروجی).
- OutputRdy: inout std_logic ← آینهٔ(نمایش در تست‌بنچ) OutRdy از انکودر؛ در عمل «رفتار خروجی» دارد (برای اعلان آماده بودن فریم کدشده).
- RST: in std_logic و clk: in std_logic ← ریست هم‌زمان و کلاک اصلی.

پورت‌های داخلی

- InputPacket: out data_packet ← پکت ۷بایتی ساخته‌شده توسط ControlUnit.
- InputPacketRdy: out std_logic ← آماده بودن پکت ورودی (آینهٔ CUPacketRdy)

```
12  library IEEE;
13  use IEEE.STD_LOGIC_1164.ALL;
14  use IEEE.NUMERIC_STD.ALL;
15  use work.Packages.All;
16
17  entity TopModule is
18      Port ( Input : in STD_LOGIC;
19             InputBegining : in STD_LOGIC;
20             Output : out STD_LOGIC;
21             OutputRdy : inout STD_LOGIC;
22             --
23             InputPacket : out data_packet;
24             InputPacketRdy : out STD_LOGIC;
25             CURRCheck : out data_packet;
26             CURRCheckRdy : out STD_LOGIC;
27             RamPackIn : OUT data_packet;
28             RTAResp : OUT data_packet;
29             --
30             Error : out STD_LOGIC;
31             RST : in STD_LOGIC;
32             clk : in STD_LOGIC
33             );
34  end TopModule;
35
36  architecture Behavioral of TopModule is
37
38     signal DataByte : byte;
39     signal DecValidation : STD_LOGIC;
40     signal DecDataIsRdy : STD_LOGIC;
41
42     signal Switch : STD_LOGIC;
43     signal CUPacket : data_packet;
44     signal CUPacketRdy : STD_LOGIC;
45     signal PacketValidation : STD_LOGIC;
46
47     signal RamReadResp : data_packet;
48     signal RamToAlu : data_packet;
49     signal RespForAluRdy : STD_LOGIC;
50     signal RamReadRespRdy : STD_LOGIC;
51     signal RamError : STD_LOGIC;
52
53     signal AluEnable : STD_LOGIC;
54     signal PackType : packet_type;
55     signal AluToRam : data_packet;
56     signal SendToRamReady : STD_LOGIC;
57     signal AluDone : STD_LOGIC;
58     signal AluError : STD_LOGIC;
59
60     signal EncByte : byte;
61     signal EncInBit : STD_LOGIC;
```

- CURRCheck: out data_packet ← آخرین پاسخ خواندن RAM که برای ارسال سریالی انتخاب شده (RamReadResp).
- CURRCheckRdy: out std_logic ← آماده بودن پاسخ RAM (آینهٔ RamReadRespRdy).
- RamPackIn: out data_packet ← درخواست‌های ALU به RAM (آینهٔ AluToRam).
- RTAResp: out data_packet ← پاسخ‌های RAM به ALU (آینهٔ RamToAlu).
- Error: out std_logic ← پرچم خطای کلی از ErrorDetection.

این پورت‌های «Check/Packet/RamPackIn/RTAResp» فقط برای مشاهده در شبیه‌سازی هستند و در مسیر اصلی پردازش تاثیر نمی‌گذارند.

```
63  begin
64
65      Decoder: entity work.HammingDecoder
66          port map
67          (
68              CodeBegins => InputBegining,
69              DecInBit => Input,
70              DecOutByte => DataByte,
71              OutRdy => DecDataIsRdy,
72              Valid => DecValidation,
73              RST => RST,
74              clk => clk
75          );
76
77      CtrlUnit: entity work.ControlUnit
78          port map
79          (
80              InputIsReady => DecDataIsRdy,
81              InByte => DataByte,
82              Validation => PacketValidation,
83              Switch => Switch,
84              PackIsReady => CUPacketRdy,
85              Packet => CUPacket,
86              PackType => PackType,
87              RST => RST,
88              clk => clk
89          );
90
91      RAM: entity work.RAM
92          port map
93          (   CtrlInputRdy => CUPacketRdy,
94              CtrlReq => CUPacket,
95              CtrlReadResp => RamReadResp,
96              CtrlReadRespReady => RamReadRespRdy,
97              AluInputRdy => SendToRamReady,
98              AluReq => AluToRam,
99              AluReadResp => RamToAlu,
100             AluReadRespReady => RespForAluRdy,
101             Error => RamError,
102             RST => RST,
103             clk => clk
104         );
```

جریان داده و کنترل

۱. **Input ← Decoder**: با InputBegining='1'، ۱۳ بیت از Input جمع می‌شود؛ سپس DataByte آماده و DecDataIsRdy='1'. اگر تصحیح ممکن/معتبر باشد، DecValidation='1'.

۲. **Decoder ← Control Unit**: CU با هر DecDataIsRdy یک بایت می‌گیرد، طبق Function بایت اول، بایت‌های بعدی را تا تکمیل Packet می‌خواند، CUPacketRdy='1' و PacketValidation را تنظیم می‌کند؛ PackType نیز تعیین می‌شود.

۳. **تغذیه هم‌زمان RAM و ALU**: با CUPacketRdy='1' هر دو ماژول اتفاقا یک پکت می‌بینند:

- **RAM (شاخهٔ Ctrl)**: اگر CUPacket(0) کد Read/Write باشد، عمل می‌کند؛ در Read، RamReadResp/RamReadRespRdy تولید می‌شود.

- **ALU**: اگر PackType از حالات ALU باشد، ALU اجرا را شروع می‌کند؛ نیازهای خواندن را از RAM با AluToRam/SendToRamReady درخواست می‌کند و پاسخ‌ها را در RamToAlu/RespForAluRdy می‌گیرد؛ در پایان، نتیجه را با پکت Write در RAM می‌نویسد.

۴. **قالب‌بندی پاسخ برای ارسال سریالی:** پاسخ خواندن RAM (شاخهٔ Ctrl) به **PackToByte → ByteToBit → HammingEncoder** می‌رود؛ انکودر ۱۳ بیت کد را روی Output می‌فرستد و OutputRdy را اعلام می‌کند.

۵. **تجمیع خطا**: اگر هرکدام از (DecodingError, PacketError, RamError, AluError) فعال باشند، Error='1'.

```
106     ALU: entity work.ALU
107        port map
108        (
109           InputRdy => CUPacketRdy,
110           InPack => CUPacket,
111           PackMode => PackType,
112           SendToRam => AluToRam,
113           ReadResponse => RamToAlu,
114           ReadResponseRdy => RespForAluRdy,
115           SendToRamReady => SendToRamReady,
116           Finish => AluDone,
117           Error => AluError,
118           RST => RST,
119           clk => clk
120        );
121
122     PacketToByte: entity work.PackToByte
123        port map
124        (
125           PackIn => RamReadResp,
126           ByteOut => EncByte,
127           clk => clk
128        );
129
130     ByteToBit: entity work.ByteToBit
131        port map
132        (
133           ByteIn => EncByte,
134           BitOut => EncInBit,
135           clk => clk
136        );
137
138     Encoder: entity work.HammingEncoder
139        port map
140        (
141           BitIn => EncInBit,
142           BitOut => Output,
143           OutRdy => OutputRdy,
144           RST => RST,
145           clk => clk
146        );
147
148     ErrorDetection: entity work.ErrorDetection
149        port map
150        (
151           DecodingError => not(DecValidation),
152           PacketError => not(PacketValidation),
153           RamError => RamError,
154           AluError => AluError,
155           Error => Error
156        );
157
158     CURRCheck <= RamReadResp;
159     CURRCheckRdy <= RamReadRespRdy;
160     RamPackIn <= AluToRam;
161     RTAResp <= RamToAlu;
162     InputPacket <= CUPacket;
163     InputPacketRdy <= CUPacketRdy;
164
165  end Behavioral;
```

فصل چهارم) تست بنچ و نتایج کلی سیستم

هدف تست بنچ تاپ ماژول: اعتبارسنجی انتهابه‌انتهای سیستم از لحظه‌ی ورود فریم‌های همینگ روی خط ورودی تا تولید پاسخ کدشده روی خروجی. در این مسیر، تشکیل پکت در واحد کنترل، اجرای شاخه‌های RAM وALU ، قالب‌بندی Read Response، و بازکدگذاری همینگ بررسی می‌شود. پورت‌های کمکی تاپ‌ماژول نیز برای مشاهدهٔ پکت‌های داخلی و تعامل‌های ALU↔RAM در شبیه‌سازی در دسترس قرار می‌گیرند.

در این تست‌بنچ، ATRPack نقش آینه‌ی درخواست‌های ALU به RAM و RTAResp نقش آینه‌ی پاسخ‌های RAM به ALU را دارد (همان آینه‌هایی که در تاپ ماژول در نظر گرفته شده‌اند).

ورودی/خروجی‌ها و سیگنال‌های اصلی تست

- clk / RST: کلاک ۱۰ nsدوره؛ ریست هم‌زمان(سنکرون) در ابتدای شبیه‌سازی به‌مدت ۱۰۰ nsفعال می‌شود.
- InputBegining: نبض شروع فریم برای دیکودر همینگ.
- Input: جریان سریال ۱۳ بیت برای هر «سلول» (هر سلول متناظر با یک بایت داده).
- Output / OutputRdy: خروجی سریال کد همینگ و نشانگر آماده بودن فریم کد شده.
- InputPacket / InputPacketRdy: پکت ۷بایتی ساخته‌شده توسط ControlUnit و سیگنال آماده بودن آن.
- CURRCheck / CURRCheckRdy: پاسخ خواندن RAM در قالب ۷بایتی و آماده بودن آن برای ارسال.
- ATRPack / RTAResp: آینه‌ی درخواست‌های ALU به RAM و پاسخ‌های RAM به ALU.
- Error: پرچم خطای کلی سیستم.

```
33  ENTITY ModuleTestbench IS
34  END ModuleTestbench;
35
36  ARCHITECTURE behavior OF ModuleTestbench IS
37
38      -- Component Declaration for the Unit Under Test (UUT)
39
40      COMPONENT TopModule
41      PORT(
42           Input : IN  std_logic;
43           InputBegining : IN  std_logic;
44           Output : OUT  std_logic;
45           OutputRdy : INOUT  std_logic;
46           ---
47           InputPacket : OUT  data_packet;
48           InputPacketRdy : OUT  std_logic;
49           CURRCheck : OUT  data_packet;
50           CURRCheckRdy : OUT  std_logic;
51           ATRPack : OUT data_packet;
52           RTAResp : OUT  data_packet;
53           ---
54           Error : OUT  std_logic;
55           RST : IN  std_logic;
56           clk : IN  std_logic
57          );
58      END COMPONENT;
59
60     --Inputs
61     signal Input : std_logic := '0';
62     signal InputBegining : std_logic := '0';
63     signal RST : std_logic := '0';
64     signal clk : std_logic := '0';
65
66     --BiDirs
67     signal OutputRdy : std_logic;
68
69     --Outputs
70     signal Output : std_logic;
71     ---
72     signal InputPacket : data_packet;
73     signal InputPacketRdy : std_logic;
74     signal CURRCheck : data_packet;
75     signal CURRCheckRdy : std_logic;
76     signal ATRPack : data_packet;
77     signal RTAResp : data_packet;
78     ---
79     signal Error : std_logic;
80
81     -- Clock period definitions
82     constant clk_period : time := 10 ns;
```

روش اعمال محرک

برای هر سناریو، آرایه‌ی HammingPacket مقداردهی و فقط تعداد «سلول»های اعلام‌شده همان سناریو داده می‌شود (۴ تا ۷ سلول بسته به نوع عملیات). ControlUnit از روی خروجی دیکودر، پکت ۷بایتی را تشکیل می‌دهد و سپس مسیر مناسب RAMیا ALU فعال می‌شود.

```
84  BEGIN
85
86     -- Instantiate the Unit Under Test (UUT)
87     uut: TopModule PORT MAP (
88            Input => Input,
89            InputBegining => InputBegining,
90            Output => Output,
91            OutputRdy => OutputRdy,
92            ---
93            InputPacket => InputPacket,
94            InputPacketRdy => InputPacketRdy,
95            CURRCheck => CURRCheck,
96            CURRCheckRdy => CURRCheckRdy,
97            ATRPack => ATRPack,
98            RTAResp => RTAResp,
99            ---
100           Error => Error,
101           RST => RST,
102           clk => clk
103         );
104
105    -- Clock process definitions
106    clk_process :process
107    begin
108       clk <= '0';
109       wait for clk_period/2;
110       clk <= '1';
111       wait for clk_period/2;
112    end process;
113
114
115    -- Stimulus process
116    stim_proc: process
117       variable HammingPacket : hamming_packet;
118    begin
119       -- hold reset state for 100 ns.
120       RST <= '1';
121       wait for clk_period*10;
122       RST <= '0';
123       wait for 10 ns;
125       -- 1) Write: Uses 5 cells. Writing "00000001" in "00001100".
126       HammingPacket := ("1111111000001",
127                         "1100000011000",
128                         "0001000100011",
129                         "1110111011110",
130                         "0010111111010",
131                         "0000000000000",
132                         "0000000000000");
133       for i in 0 to 4 loop
134          InputBegining <= '1';
135          wait for 10 ns;
136          InputBegining <= '0';
137          for j in 0 to 12 loop
138             Input <= HammingPacket(i)(j);
139             wait for 10 ns;
140          end loop;
141          wait for clk_period*2;
142       end loop;
143
144       -- 2) Write: Uses 5 cells. Writing "00000100" in "00000011".
145       HammingPacket := ("1111111000001",
146                         "1101000000111",
147                         "0100000101001",
148                         "1110111011110",
149                         "0110111101111",
150                         "0000000000000",
151                         "0000000000000");
152       for i in 0 to 4 loop
153          InputBegining <= '1';
154          wait for 10 ns;
155          InputBegining <= '0';
156          for j in 0 to 12 loop
157             Input <= HammingPacket(i)(j);
158             wait for 10 ns;
159          end loop;
160          wait for clk_period*2;
161       end loop;
```

```
        -- 3) Alu: Operand Alu, Uses 6 cells. Writing {"00001100"* + "00000011"* in "00000001".
        -- "00001100"* = "00000001", "00000011"* = "00000100" => "00000001"* = "00000101".
        HammingPacket := ("0000000000000",
                          "1100000011000",
                          "1101000000111",
                          "0001000100011",
                          "0000000000000",
                          "1101001000000",
                          "0000000000000");
        for i in 0 to 5 loop
           InputBegining <= '1';
           wait for 10 ns;
           InputBegining <= '0';
           for j in 0 to 12 loop
              Input <= HammingPacket(i)(j);
              wait for 10 ns;
           end loop;
           wait for clk_period*2;
        end loop;

        -- 4) Read: Uses 4 cells. Reading "00000101" from "00000001".
        HammingPacket := ("0001000011111",
                          "0001000100011",
                          "0000000000000",
                          "1101001000000",
                          "0000000000000",
                          "0000000000000",
                          "0000000000000");
        for i in 0 to 3 loop
           InputBegining <= '1';
           wait for 10 ns;
           InputBegining <= '0';
           for j in 0 to 12 loop
              Input <= HammingPacket(i)(j);
              wait for 10 ns;
           end loop;
           wait for clk_period*2;
        end loop;

        -- 5) Alu: Immediate Alu, Uses 6 cells. Writing {"00000000"* + "01000000" in "00011111".
        -- "000000000"* = "000000000", => "00011111"* = "01000000".
        HammingPacket := ("0100011011001",
                          "0000000000000",
                          "1001100000001",
                          "1100001011111",
                          "0000000000000",
                          "0110001110111",
                          "0000000000000");
        for i in 0 to 5 loop
           InputBegining <= '1';
           wait for 10 ns;
           InputBegining <= '0';
           for j in 0 to 12 loop
              Input <= HammingPacket(i)(j);
              wait for 10 ns;
           end loop;
           wait for clk_period*2;
        end loop;

        -- 6) Read: Uses 4 cells. Reading "01000000" from "00011111".
        HammingPacket := ("0001000011111",
                          "1100001011111",
                          "0000000000000",
                          "1000100011110",
                          "0000000000000",
                          "0000000000000",
                          "0000000000000");
        for i in 0 to 3 loop
           InputBegining <= '1';
           wait for 10 ns;
           InputBegining <= '0';
           for j in 0 to 12 loop
              Input <= HammingPacket(i)(j);
              wait for 10 ns;
           end loop;
           wait for clk_period*2;
        end loop;
```

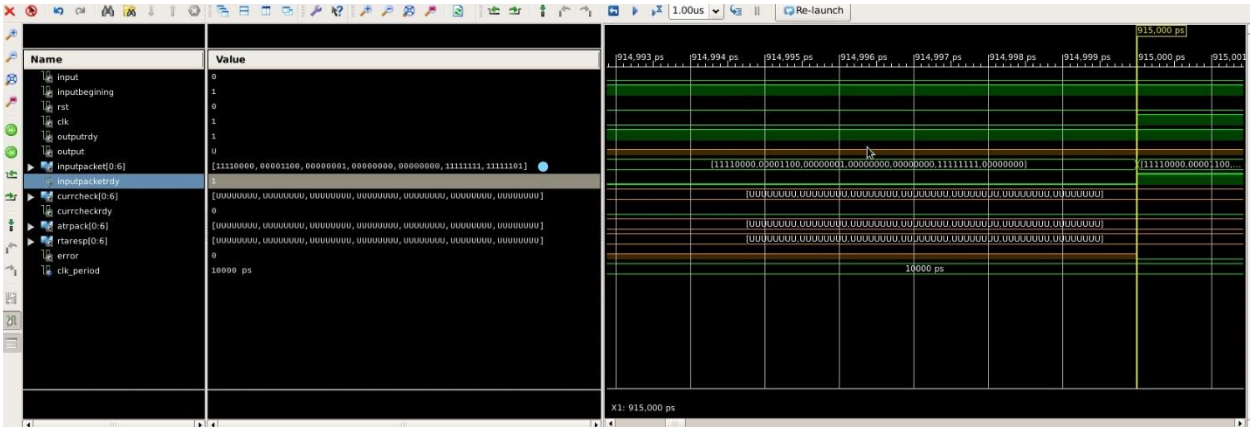
```
        -- 7) Alu: Array Alu, Uses 7 cells. Writing {"00000000" + i"* + "00000001" in "00011111 + i"*.
        -- "00000000"* = "00000000",
        -- "00000001"* = "00000101",
        -- "00000010"* = "00000000",
        -- =>
        -- "00011111"* = "00000001",
        -- "00000000"* = "00000110",
        -- "00000001"* = "00000001".
        HammingPacket := ("0111100000000",
                          "0000000000000",
                          "0001000100011",
                          "1101000000111",
                          "1100001011111",
                          "1110111011110",
                          "1111110000110");
        for i in 0 to 6 loop
           InputBegining <= '1';
           wait for 10 ns;
           InputBegining <= '0';
           for j in 0 to 12 loop
              Input <= HammingPacket(i)(j);
              wait for 10 ns;
           end loop;
           wait for clk_period*2;
        end loop;
        wait for 30 ns;

        -- 8) Read: Uses 4 cells. Reading "00000001" from "00011111".
        HammingPacket := ("0001000011111",
                          "1100001011111",
                          "0000000000000",
                          "1000100011110",
                          "0000000000000",
                          "0000000000000",
                          "0000000000000");
        for i in 0 to 3 loop
           InputBegining <= '1';
           wait for 10 ns;
           InputBegining <= '0';
           for j in 0 to 12 loop
              Input <= HammingPacket(i)(j);
              wait for 10 ns;
           end loop;
           wait for clk_period*2;
        end loop;
        wait for clk_period*2;
        -- 8.33) Read: Uses 4 cells. Reading "00000110" from "0000000".
        HammingPacket := ("0001000011111",
                          "0000000000000",
                          "0000000000000",
                          "0001000011111",
                          "0000000000000",
                          "0000000000000",
                          "0000000000000");
        for i in 0 to 3 loop
           InputBegining <= '1';
           wait for 10 ns;
           InputBegining <= '0';
           for j in 0 to 12 loop
              Input <= HammingPacket(i)(j);
              wait for 10 ns;
           end loop;
           wait for clk_period*2;
        end loop;
        wait for clk_period*2;

        -- 8.66) Read: Uses 4 cells. Reading "00000001" from "00000001".
        HammingPacket := ("0001000011111",
                          "0001000100011",
                          "0000000000000",
                          "1101001000000",
                          "0000000000000",
                          "0000000000000",
                          "0000000000000");
        for i in 0 to 3 loop
           InputBegining <= '1';
           wait for 10 ns;
           InputBegining <= '0';
           for j in 0 to 12 loop
              Input <= HammingPacket(i)(j);
              wait for 10 ns;
           end loop;
           wait for clk_period*2;
        end loop;
        wait for clk_period*2;
```

```
327
328         -- 9) Alu: Indirect Addressing, Uses 6 cells. Writing {"00000000"* + "00011111"** in "00001000".
329         -- "00000000"* = "00000110",
330         -- "00011111"* = "00000001", "00000001"* = "00000001"
331         -- => "00001000"* = "00000111".
332         HammingPacket := ("1000011000001",
333                           "0000000000000",
334                           "1100001011111",
335                           "1000000110001",
336                           "0000000000000",
337                           "1101101101111",
338                           "0000000000000");
339         for i in 0 to 5 loop
340            InputBegining <= '1';
341            wait for 10 ns;
342            InputBegining <= '0';
343            for j in 0 to 12 loop
344               Input <= HammingPacket(i)(j);
345               wait for 10 ns;
346            end loop;
347            wait for clk_period*2;
348         end loop;
349
350         -- 10) Read: Uses 4 cells. Reading "00000111" from "00001000".
351         HammingPacket := ("0001000011111",
352                           "1000000110001",
353                           "0000000000000",
354                           "0100001101110",
355                           "0000000000000",
356                           "0000000000000",
357                           "0000000000000");
358         for i in 0 to 3 loop
359            InputBegining <= '1';
360            wait for 10 ns;
361            InputBegining <= '0';
362            for j in 0 to 12 loop
363               Input <= HammingPacket(i)(j);
364               wait for 10 ns;
365            end loop;
366            wait for clk_period*2;
367         end loop;
368
369         wait for clk_period*2;
370      end process;
371
372   END;
```

سناریوهای تست و انتظار نتایج

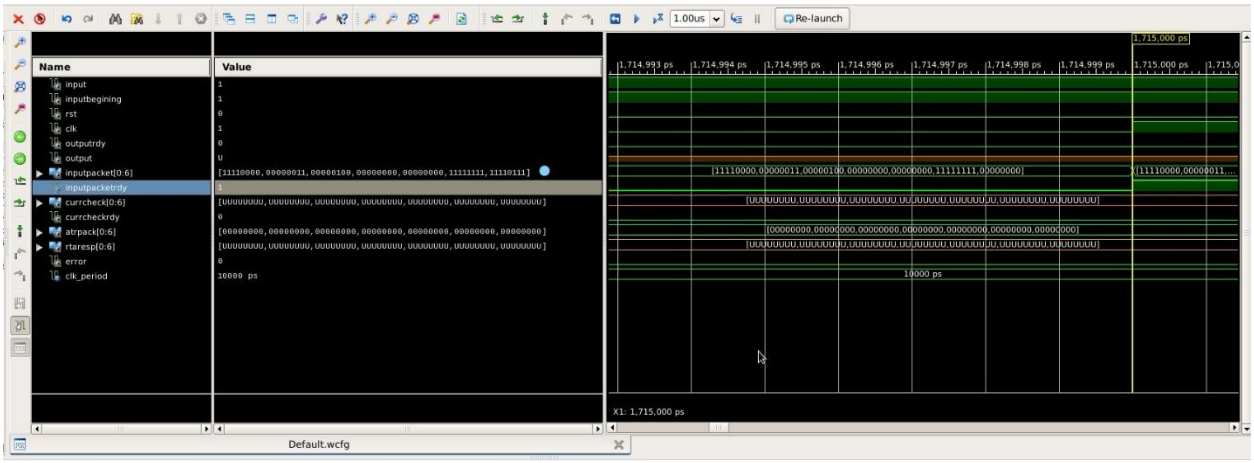
۱. **Write(۵ سلول)** ← نوشتن 00000001 در آدرس 00001100.

انتظار: تشکیل پکت Write، انجام نوشتن در RAM، بدون پاسخ خواندن (عدم فعال‌شدن CURRCheckRdy)، Error='0'.



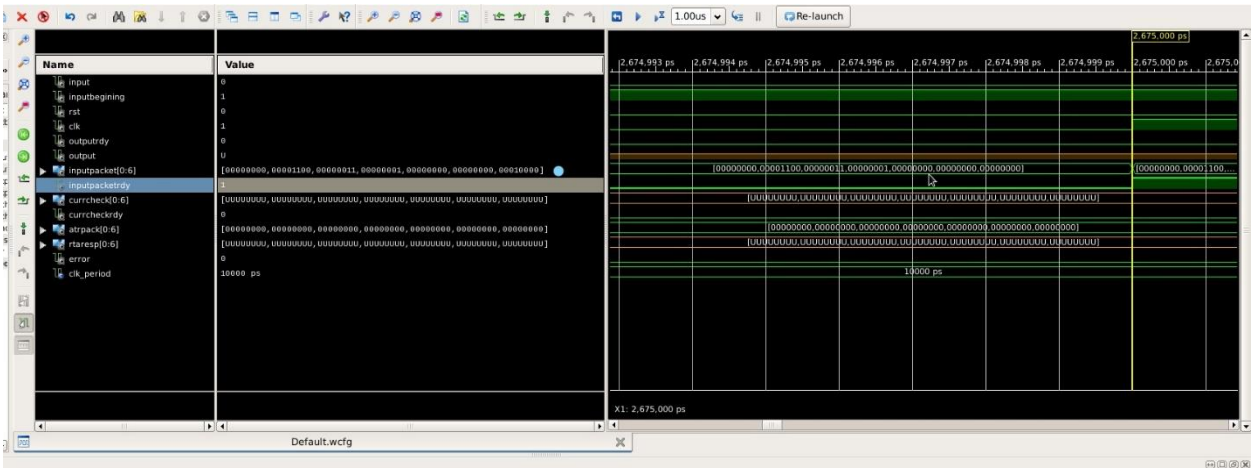
۲. **Write(۵ سلول)**← نوشتن 00000100 در آدرس 00000011.

انتظار: مشابه سناریوی ۱؛ Error='0'.



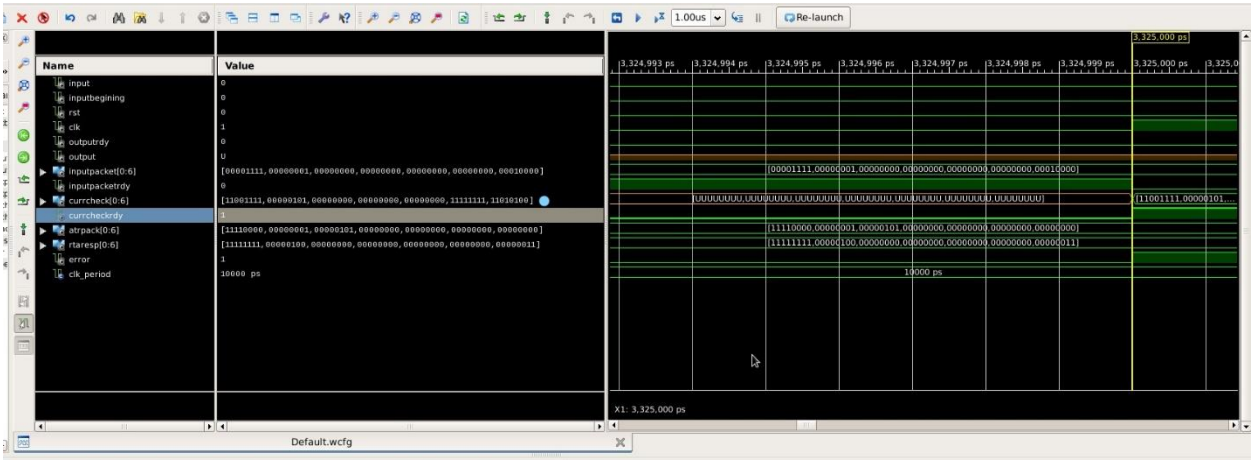
۳. **Operand_ALU (۶ سلول)** ← جمع محتویات آدرس‌های 00001100 و 00000011 و نوشتن نتیجه در 00000001.

انتظار: توالی Read→Read→Write بین ALU و RAM (در ATRPack/RTARespقابل رؤیت)، نتیجهٔ مقصد باید 00000101 شود.



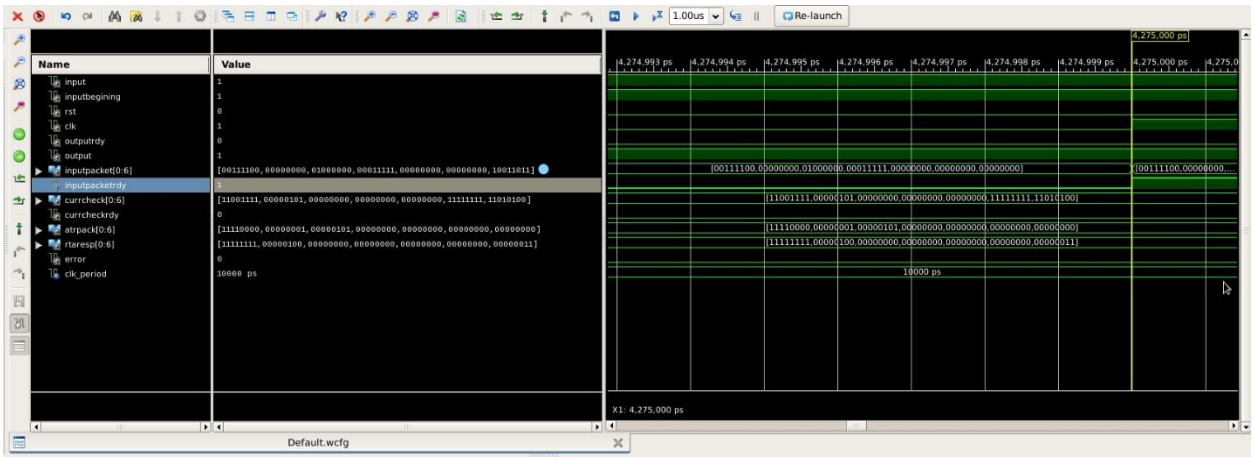
٤. **Read(۴ سلول)** ← خواندن از 00000001.

انتظار: تولید پاسخ خواندن با شناسهٔ "11001111" و دادهٔ 00000101، چک‌سام معتبر، ارسال سریال روی خروجی.



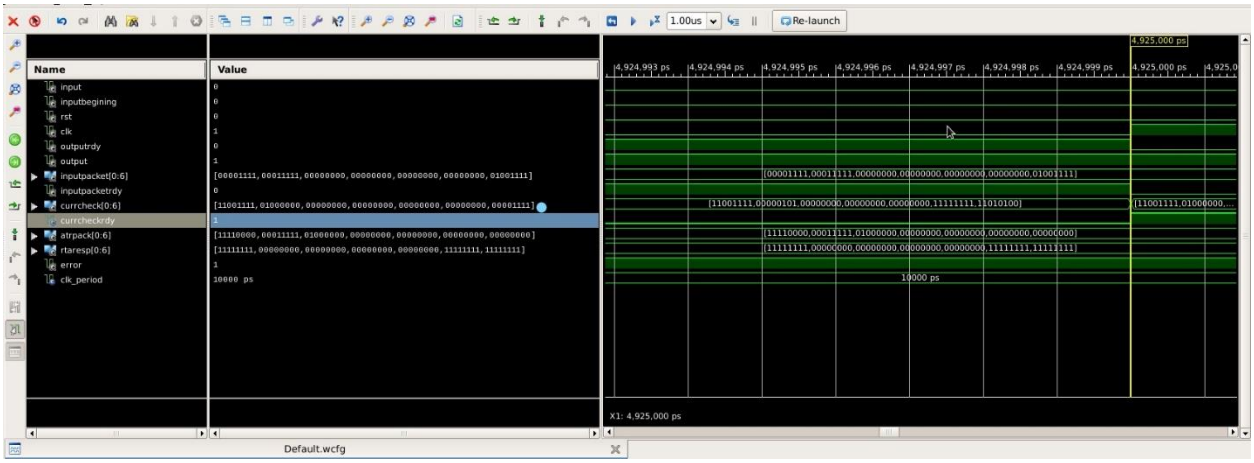
۵. **Immediate_ALU (۶ سلول)** ← ترکیب یک خانهٔ RAM با دادهٔ فوری و نوشتن در 00011111.

انتظار: الگوی Read→Write؛ مقدار آدرس مقصد به 01000000 تغییر می‌کند.



۶. **Read(۴ سلول)** ← خواندن از 00011111.

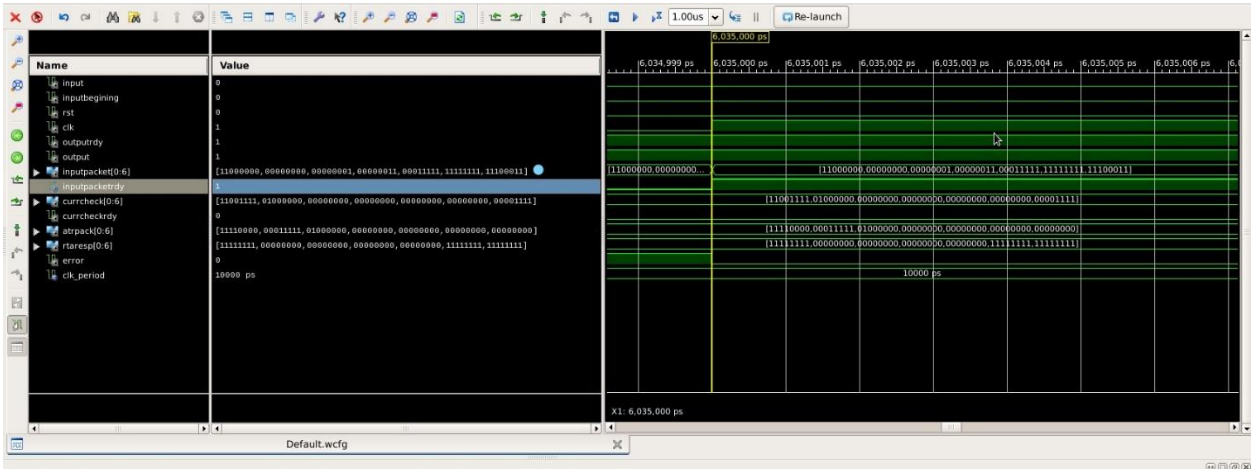
انتظار: پاسخ خواندن با دادهٔ 01000000 و چک‌سام صحیح، سپس ارسال به شکل سریال.



۷. **Array_ALU (۷ سلول)** ← اجرای عمل آرایه‌ای با طول معتبر و نوشتن پی‌درپی نتایج.

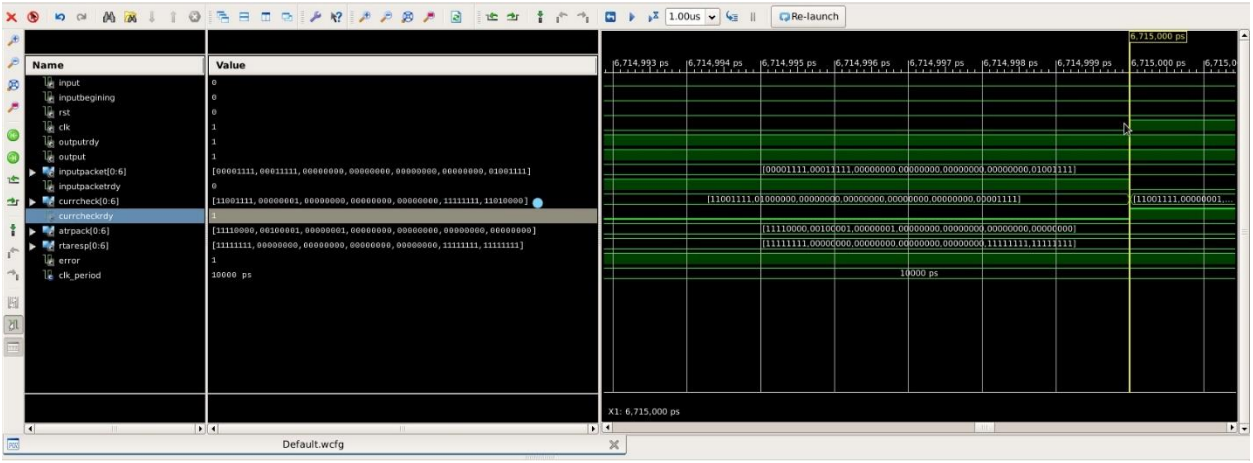
انتظار: مجموعه‌ای از Read/Write متوالی متناسب با طول؛ پس از اتمام، وضعیت حافظه مطابق توضیح سناریو به این صورت باشد:

00000001 ← 00011111، 00000110 ← 00000000، 00000001 ← 00000001.



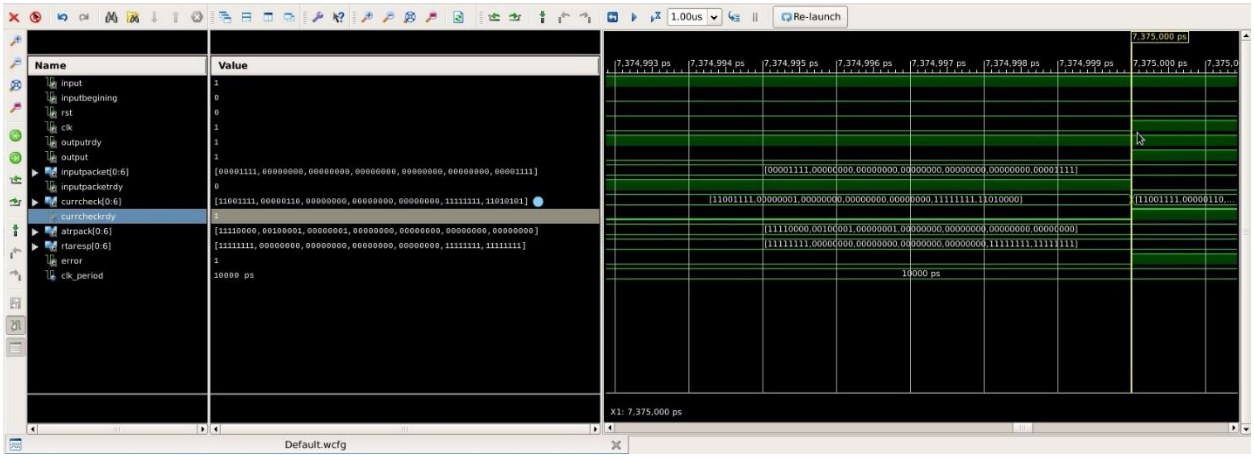
۸. **Read(۴ سلول)** ← خواندن از 00011111.

انتظار: داده‌ی خوانده‌شده 00000001 (اثر سناریوی آرایه‌ای).



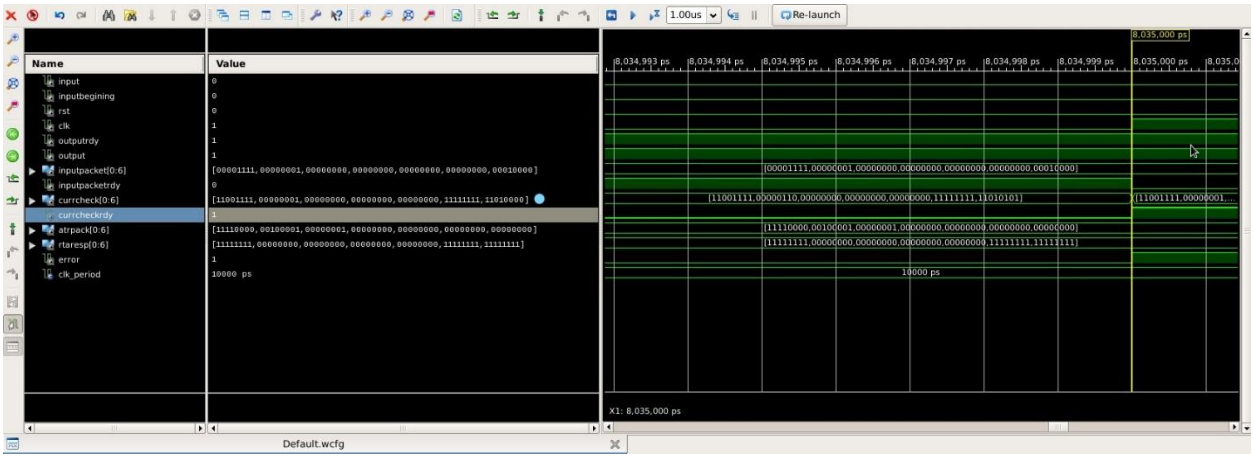
۸.۳۳) **Read(۴ سلول)** ← خواندن از 00000000.

انتظار: داده‌ی خوانده‌شده 00000110.



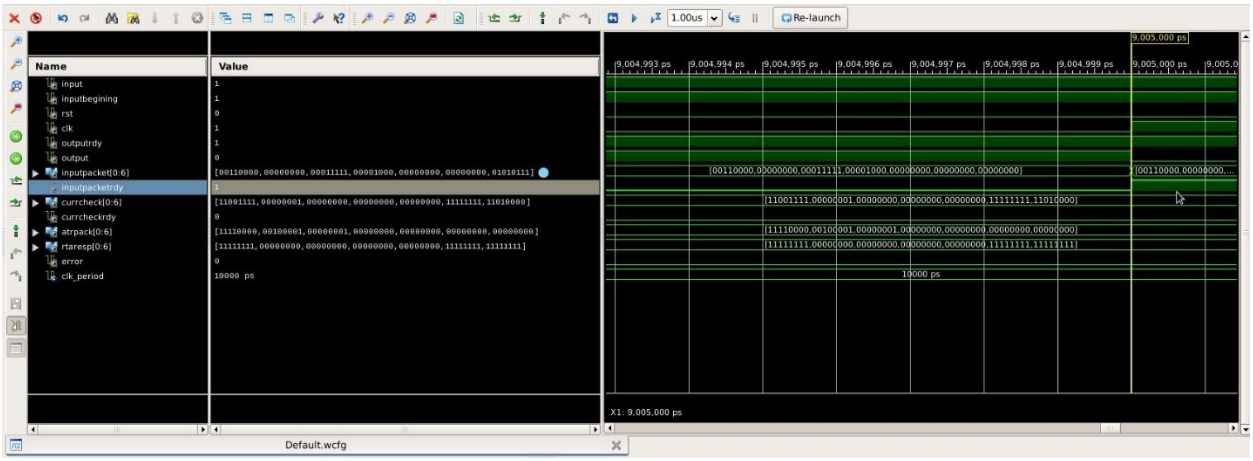
۸.۶۶) **Read (۴ سلول)** ← خواندن از 00000001.

انتظار: داده‌ی خوانده‌شده 00000001 (تأیید وضعیت پس از سناریوهای قبل).



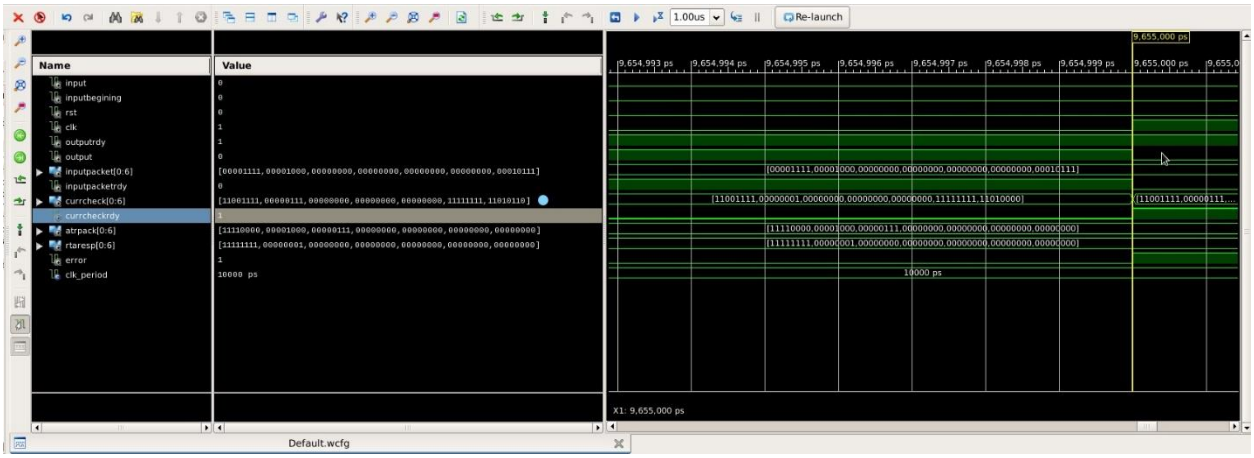
۹. **Indirect_Addressing(۶ سلول)** ← آدرس‌دهی غیرمستقیم: خواندن آدرس غیرمستقیم، سپس خواندن از آدرس واقعی، اعمال عمل و نوشتن در 00001000.

انتظار: الگوی Read(Indirect)→Read(Real)→Write؛ مقدار مقصد باید به 00000111 برسد.



۱۰. **Read(۴ سلول)** ← خواندن از 00001000.

انتظار: داده‌ی خوانده‌شده 00000111 و پاسخ استاندارد با چک‌سام صحیح؛ ارسال سریال روی خروجی.



معیار پذیرش

با تکمیل هر پکت ورودی، InputPacketRdy فعال و محتوای InputPacket(0..6) مطابق سلول‌های داده‌شده باشد. در سناریوهای Read، CURRCheckRdy فعال شود و پاسخ ۷بایتی شامل شناسهٔ "11001111"، دادهٔ انتظار و چک‌سام صحیح باشد و سپس به‌صورت سریالی کد ارسال گردد. در سناریوهای ALU، توالی دسترسی‌ها و دادهٔ مقصد در خواندن‌های بعدی تأیید شود. در کل سناریوهای معتبر، پرچم Error غیرفعال باقی بماند.