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 مانندEncoder ، 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-based)

- 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 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.(Latency)

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

این ماژول نقش مغز سیستم را دارد FSM .با بررسی پکت دریافتی و سیگنال‌هایی مانند InputRdy ، Functionو OutputRdy مشخص می‌کند در هر لحظه کدام بخش سیستم باید فعال شود.

برای مثال اگر پکتی با Function مربوط به ALU دریافت شود، FSMابتدا 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

۳-۱. ماژول Hamming Encoder

هدف این ماژول این است که داده ۸ بیتی را به صورت سریالی دریافت کرده و پس از محاسبه بیت‌های توازن (Parity Bits)و یک بیت توازن کل(Overall Parity) ، آن را به صورت ۱۳ بیتی کد Hamming تبدیل کرده و سریالی ارسال نماید. این کد قابلیت تصحیح یک بیت خطا و تشخیص دو بیت خطا را دارد.

ورودی‌ها و خروجی‌ها:

- Clk: سیگنال ساعت اصلی برای هماهنگی عملیات دریافت و ارسال.
- RST: ریست ماژول و بازنشانی تمام سیگنال‌های داخلی.
- BitIn: بیت ورودی داده (LSB یا MSB بسته به قرارداد سیستم)، در هر سیکل کلاک یک بیت.
- BitOut: بیت خروجی کد Hamming، در هر سیکل کلاک یک 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 STD_LOGIC_VECTOR(0 to 12);
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;
```

درون معماری Behavioral این ماژول، یک process تعریف شده که با لبه بالارونده کلاک (rising_edge(clk))فعال شده و بخش‌های عملکردی آن به صورت زیر است:

۱. بررسی اولیه Reset

در ابتدای هر لبه بالارونده کلاک، اگر RST = '1' باشد:

- رجیستر InCode (۸ بیتی) صفر می‌شود.
- رجیستر Encoded (۱۳ بیتی) صفر می‌شود.
- شمارنده‌های InCnt (برای دریافت) و OutCnt (برای ارسال) به صفر برمی‌گردند.
- سیگنال OutRdy صفر می‌شود.

۲. حالت دریافت داده – Input Mode (زمانی که Open_In = '1')

- تا زمانی که InCnt < 8 باشد، بیت‌های ورودی (BitIn) به‌ترتیب وارد رجیستر InCode می‌شوند.
- هر بیت جدید به صورت شیفت رجیستر ذخیره می‌شود ((BitIn & InCode(7 downto 1)).
- پس از دریافت هشتمین بیت، شمارنده InCnt به صفر باز می‌گردد و فرآیند کدگذاری آغاز می‌شود.

۳. مپینگ بیت‌ها:

- **D0..D7** از بیت‌های InCode گرفته می‌شوند.
- **P1, P2, P4, P8** بعد از قرار دادن داده‌ها محاسبه می‌شوند.
- **P_total** با استفاده از XOR تمام بیت‌های ۰ تا ۱۱ محاسبه می‌شود.

```
architecture Behavioral of HammingEncoder is

    signal InCode : byte := (others => '0');
    signal Encoded : STD_LOGIC_VECTOR(0 to 12) := (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;
```

۴. محاسبه بیت‌های توازن (Parity Bits)

پس از قرارگیری بیت‌های داده:

- P1 (Encoded(0)) = XOR بیت‌های ۲, ۴, ۶, ۸, ۱۰
- P2 (Encoded(1)) = XOR بیت‌های ۲, ۵, ۶, ۹, ۱۰
- P4 (Encoded(3)) = XOR بیت‌های ۴, ۵, ۶, ۱۱

- (Encoded(7)) P8= XOR بیت‌های ۸, ۹, ۱۰, ۱۱
- (Encoded(12)) P_total= XOR تمام بیت‌های ۰ تا ۱۱

```
68                  else                                        -- 14 * Clk -> Output
69                      TestBenchCheck <= Encoded;
70                      OutRdy <= '1';
71                      if OutCnt < 13 then
72                          BitOut <= Encoded(OutCnt);
73                          OutCnt <= OutCnt + 1;
74                      else
75                          InCnt <= 0;
76                          OutCnt <= 0;
77                          OutRdy <= '0';
78                      end if;
79                  end if;
80              end if;
81          end if;
82      end process;
83
84  end Behavioral;
```

۵. حالت ارسال داده (Output Mode)

- وقتی OutRdy = '1' باشد، بیت‌های Encoded به‌ترتیب از $OutCnt = 0$ تا $OutCnt = 12$ روی خروجی BitOut قرار می‌گیرند.
- بعد از ارسال بیت شماره ۱۲، شمارنده OutCnt صفر شده و ماژول دوباره آماده دریافت داده جدید می‌شود.

۶. نکته درباره چرخه دریافت و ارسال

؟؟؟؟

۳-۲. ماژول Hamming Decoder

هدف این ماژول دریافت کد Hamming ۱۳ بیتی به‌صورت سریالی، بررسی صحت آن، شناسایی و تصحیح یک بیت خطا (در صورت وجود) و تولید خروجی ۸ بیتی تصحیح‌شده است. برخلاف نسخه قبلی، در این طراحی خروجی به‌صورت کامل (۸ بیت هم‌زمان) از طریق out_data ارائه می‌شود.

این ماژول دارای ورودی‌ها و خروجی‌های زیر است:

- Clk: سیگنال ساعت اصلی سیستم برای هماهنگی اجرای ترتیبی کد.
- RST: سیگنال ریست که باعث بازنشانی تمام سیگنال‌ها و شمارنده‌ها می‌شود.
- in_start: شروع دریافت پکت جدید را مشخص می‌کند؛ در لبه فعال آن کد آماده دریافت است.
- in_data: ورودی سریالی که بیت‌های کد Hamming به‌صورت یکی‌یکی وارد سیستم می‌شوند.
- out_rdy: خروجی دوطرفه که نشان می‌دهد خروجی محاسبه و آماده ارسال است.
- out_data: خروجی ۸ بیتی، به‌صورت کامل و هم‌زمان ارائه می‌شود.
- valid_out: در صورتی که کد دریافتی حداکثر یک بیت خطا داشته باشد و تصحیح شده باشد، این سیگنال برابر با ۱ خواهد شد.

```
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 ( DecInBit : in STD_LOGIC;                          -- 1 bit input data
22                  TestBenchInputDisplay : out STD_LOGIC_VECTOR (0 to 12);
23                  OutRdy : inout  STD_LOGIC;                       -- is 1 if the output is calculated
24                  DecOutByte : out byte;                               -- 8 bit output data
25
26               Valid : out STD_LOGIC;                          -- is 1 if there is max 1 error in code
27                  RST : in STD_LOGIC;                             -- resets everything
28                  clk : in STD_LOGIC
29                  );
30    end HammingDecoder;
```

۱. بررسی اولیه Reset :

در ابتدای هر لبه‌ی بالارونده‌ی کلاک، ابتدا بررسی می‌شود که آیا سیگنال‌های RST یا in_start فعال شده‌اند یا نه.
در صورت فعال بودن هرکدام:

- مقدار cnt (شمارنده موقعیت بیت) به صفر بازنشانی می‌شود.
- بردار inp_enc_data که برای ذخیره ۱۳ بیت دریافتی به کار می‌رود، صفر می‌شود.
- سیگنال‌های خروجی valid_out و out_rdy خاموش می‌شوند.
- سیگنال out_data نیز به مقدار صفر تنظیم می‌گردد.

هدف از این بخش، آماده‌سازی کامل ماژول برای دریافت یک پکت جدید است.

```
32  architecture Behavioral of HammingDecoder is
33
34      signal Code : STD_LOGIC_VECTOR (0 to 12);          -- 13bit data
35      signal cnt : integer := 0;                                  -- We don't want it to be initialized after eavh clock!
36
37  begin
38          process (clk)
39              variable ind : integer := 0;  -- possible error position
40              variable ErrorMarker : STD_LOGIC_VECTOR (3 downto 0);       -- possible 1 error position
41              variable ExtentionBit : STD_LOGIC;                         -- remade bit #13
42          begin
43              if rising_edge(clk) then
44                  if RST = '1' then
45                      Valid <= '1';
46                      OutRdy <= '0';
47                      ErrorMarker := "0000";
48                      Code <= "0000000000000";
49                      cnt <= 0;
50                      ind := 0;
51                  else                                                                    -- 13 * Clk -> Input
52                      if cnt < 13 then
53                          Code <= Code(1 to 12) & DecInBit;
54                          cnt <= cnt + 1;
55                      elsif cnt = 13 then                                                 -- 1 * Clk -> Calculation
56                          -- Constructing the new parities
57                          TestBenchInputDisplay <= Code;
58                          ErrorMarker(0) := Code(0) xor (Code(2) xor (Code(4) xor
59                                          (Code(6) xor (Code(8) xor Code(10)))));
60                          ErrorMarker(1) := Code(1) xor (Code(2) xor (Code(5) xor
61                                          (Code(6) xor (Code(9) xor Code(10)))));
62                          ErrorMarker(2) := Code(3) xor (Code(4) xor (Code(5) xor
63                                          (Code(6) xor Code(11))));
64                          ErrorMarker(3) := Code(7) xor (Code(8) xor (Code(9) xor
65                                          (Code(10) xor Code(11))));
66                          ExtentionBit := Code(0) xor (Code(1) xor (Code(2) xor
67                                          (Code(3) xor (Code(4) xor (Code(5) xor
68                                          (Code(6) xor (Code(7) xor (Code(8) xor
69                                          (Code(9) xor (Code(10) xor Code(11)))))))))));
70
71                          cnt <= cnt + 1;
```

۲. حالت دریافت داده Input Mode:

بعد از ریست، ماژول وارد حالت دریافت داده می‌شود. در این مرحله:

- در هر لبه‌ی کلاک، یک بیت از ورودی in_data خوانده می‌شود و در یکی از موقعیت‌های inp_enc_data(cnt) قرار می‌گیرد.
- شمارنده cnt از ۰ تا ۱۲ افزایش می‌یابد.
- به ازای هر مقدار cnt، یکی از بیت‌های ۱۳تایی کد Hamming در بردار inp_enc_data ذخیره می‌شود.

در پایان این فاز (زمانی که $cnt = 13$):

- دریافت داده کامل شده و شمارنده cnt مجدداً صفر می‌شود.
- سیستم وارد مرحله بررسی و تصحیح می‌شود.

```
72                      -- Correction
73                      ind :=  to_integer(unsigned(ErrorMarker));
74                      ind := ind - 1;
75                      if ((ind = -1 and ExtentionBit = Code(12)) or
76                           ((ind > -1 and ind < 12) and (ExtentionBit = not Code(12)))) then
77                          Valid <= '1';
78                          if ind > -1 then
79                              Code(ind) <= not Code(ind);
80                          end if;
81                      else
82                          Valid <= '0';
83                      end if;
84                  else
85                      cnt <= 0;
86                      DecOutByte <= code(2) & code(4) & code (5) & code(6) &
87                                    code(8) & code(9) & code(10) & code(11);
88                      OutRdy <= '1';
89                  end if;
90              end if;
91          end if;
92      end process;
93  end Behavioral;
```

۳. مپینگ بیت‌ها:

- D0..D7: بیت‌های داده اصلی (۸ بیت)

```
-- 1 * Clk -> Output
```

- P1, P2, P4, P8: بیت‌های توازن موقعیتی
- P_total: بیت توازن کلی

۴. حالت بررسی و تصحیح Processing Mode:

- چهار بیت توازن داخلی با استفاده از XOR بین بیت‌های مشخص‌شده محاسبه شده و در marker(3 downto 0) قرار می‌گیرند.
- موقعیت احتمالی خطا با تبدیل marker به عدد صحیح (ind) به‌دست می‌آید.
- یک بیت توازن کلی (overall parity) با استفاده از XOR تمام بیت‌های ۰ تا ۱۱ محاسبه می‌شود و با بیت inp_enc_data(12) مقایسه می‌گردد.

شرایط تصمیم‌گیری به این صورت است:

- اگر $ind = -1$ (یعنی $marker = 0000$) و $extention = inp\_enc\_data(12)$ ← داده بدون خطاست.
- اگر $ind \neq -1$ و extention مخالف inp_enc_data(12) ← یک بیت خطا وجود دارد و اصلاح می‌شود.
- در سایر حالات ← داده معتبر نیست و valid_out برابر با صفر می‌ماند.

در صورت تشخیص داده‌ی معتبر:

- بیت خراب (در صورت وجود) اصلاح می‌شود.
- valid_out روشن می‌شود.
- out_rdy نیز برابر ۱ می‌شود و خروجی آماده است.

۵. حالت تولید خروجی Output Mode :

پس از بررسی و تصحیح داده:

- داده اصلی ۸ بیتی از بین ۱۳ بیت موجود استخراج شده و به ترتیب در out_dataقرار می‌گیرد.
- ترتیب قرارگیری بیت‌های داده اصلی در خروجی به صورت زیر است:

$$out\_data = inp\_enc\_data(11)\ \&\ inp\_enc\_data(10)\ \&\ inp\_enc\_data(9)\&\ inp\_enc\_data(8)\ \&\ inp\_enc\_data(6)\ \&\ inp\_enc\_data(5)\ \&\ inp\_enc\_data(4)\ \&\ inp\_enc\_data(2);$$

این ترتیب دقیقاً همان است که در انکودر برای قرار دادن بیت‌های داده در موقعیت‌های خاص استفاده شده بود.

پس از تولید خروجی:

- ماژول آماده‌ی دریافت داده جدید خواهد بود.
- لازم است سیگنال in_start یا RST برای آغاز دوباره فاز دریافت فعال شوند.

۳-۳. فایل Packages

این فایل نقش یک کتابخانه پشتیبان را دارد و همه‌ی انواع داده و توابع مشترک بین ماژول‌های مختلف در آن تعریف شده است.

در ابتدای کار، انواع داده‌ی پایه مشخص شده‌اند تا همه ماژول‌ها از یک قالب مشترک استفاده کنند. برای مثال:

- Byte: یک بردار ۸ بیتی برای نمایش واحدهای داده.
- data_packet: آرایه‌ای ۷ بایتی که قالب استاندارد تبادل داده در سیستم است.
- packet_type: یک نوع شمارشی که نوع عملیات (RAM، ALU و حالت‌های مختلف آن‌ها) را مشخص می‌کند.
- ram_matrix: حافظه اصلی با ۳۲ خانه ۸ بیتی.
- ram_resp_pack: قالب ساده‌تر پکت برای پاسخ RAM .
- alu_read_cash_array: بافر موقتی برای عملیات‌های آرایه‌ای در ALU .

```
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
13      type data_packet is array (0 to 6) of byte;
14      type packet_type is (zero, Operand_Alu, Writ_e, Rea_d, Immediate_Alu, Array_Alu, Indirect_Addressing);
15      type alu_operation is (Add, Sub, BitwiseOr, BitwiseAnd);
16
17      --type ram_row is array (0 to 7) of byte;
18      type ram_matrix is array (0 to 31) of byte; --Ram_Row;
19      type ram_resp_pack is array (0 to 3) of byte;
20
21      type alu_read_cash_array is array (0 to 31) of byte;
22
23      function ByteSum (Packet : data_packet) return STD_LOGIC_VECTOR;
24      function CheckSumH (Packet : data_packet) return byte;
25      function CheckSumL (Packet : data_packet) return byte;
26      function Validate (Packet : data_packet) return STD_LOGIC;
27
28  end Packages;
```

در ادامه، چند تابع کمکی تعریف شده‌اند که بیشتر برای محاسبه و بررسی Checksum استفاده می‌شوند:

- ByteSum: پنج بایت اول پکت را جمع می‌کند و نتیجه را به صورت ۱۶ بیتی بازمی‌گرداند.
- CheckSumHو CheckSumL: به ترتیب نیمه بالایی و پایینی این جمع را برمی‌گردانند.
- Validate: بررسی می‌کند که Checksum ذخیره‌شده در پکت با مقدار واقعی جمع یکسان باشد.

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

۳-۴. ماژول Control Unit

هدف ماژول ControlUnit این است که داده‌های ۸ بیتی ورودی را دریافت کرده، آن‌ها را به پکت ۷ بایتی تبدیل می‌کند، نوع آن را تشخیص می‌دهد و مسیر مناسب پردازش(RAMیاALU) را مشخص می‌سازد. این ماژول در کنار فایل Packages پایه‌گذار مرحله‌ی «تحلیل ورودی» در سیستم هستند.

```
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 ( InByte : in byte;                          -- 8 bit input data
18
19                Switch : out STD_LOGIC;                   -- 0 is RAM mode / 1 is ALU mode
20                Packet : inout data_packet;               -- The output packet
21                PackType : inout packet_type;
22
23                PackIsReady : inout STD_LOGIC;
24                Validation : out STD_LOGIC;
25                clk : in STD_LOGIC;
26                RST : in STD_LOGIC                            -- Resets everything
27              );
28    end ControlUnit;
```

ورودی‌ها:

- InByte: داده ۸ بیتی ورودی از دیکودر.
- Clk: سیگنال کلاک.
- RST: سیگنال ریست سیستم.

خروجی‌ها:

- Packet: پکت کامل ۷ بایتی ساخته‌شده.
- PackMode: نوع عملیات پکت (از نوع packet_type).
- Switch: مسیر پردازش(0= RAM ، 1 = ALU).
- Validation: نتیجه بررسی اعتبار پکت (۱ = معتبر، ۰ = نامعتبر).

نحوه عملکرد ماژول Control Unit:

در این ماژول داده‌های ورودی به صورت بایت‌های ۸ بیتی به ماژول وارد می‌شوند. در هر سیکل کلاک یک بایت جدید دریافت شده و داخل آرایه PackHold ذخیره می‌شود. به محض دریافت اولین بایت، نوع پکت با بررسی مقدار آن تعیین می‌شود. در ادامه بسته به نوع پکت، تعدادی بایت دیگر نیز دریافت می‌شود.

اگر نوع پکت از نوع Rea_d باشد، پس از دریافت دو بایت ورودی، دریافت متوقف شده و ماژول بلافاصله وارد مرحله پردازش می‌شود. برای پکت‌های Writ_e این مقدار سه بایت است.

در صورتی که پکت از نوع ALU باشد (Immediate، Operand یا Indirect)، دریافت تا چهار بایت ادامه می‌یابد. برای حالت Array_Alu پنج بایت مورد نیاز است. بعد از آن نیز دو بایت برای Checksum دریافت شده و

پکت تکمیل می‌شود. در انتهای مرحله دریافت، ماژول با استفاده از تابع Validate، صحت پکت را بررسی کرده و نتیجه آن را از طریق سیگنال Validation گزارش می‌دهد.
پکت نهایی نیز از طریق خروجی Packet به ماژول‌های بعدی ارسال می‌شود. در کنار آن، سیگنال PackMode نوع پکت را مشخص می‌کند و سیگنال Switch مسیر جریان داده را تعیین می‌کند. اگر مقدار Switch صفر باشد، پکت برای RAM ارسال می‌شود. اگر مقدار آن یک باشد، پکت به ALU هدایت خواهد شد.

۳-۵. ماژول RAM

ماژول RAM به‌عنوان حافظه اصلی سیستم، وظیفه‌ی **ذخیره و بازیابی داده‌ها** را دارد. این ماژول می‌تواند بسته به نوع عملکرد، یک مقدار مشخص را در یک آدرس مشخص از حافظه بنویسد یا از آن بخواند. همچنین در فرآیند خواندن، پاسخ را به‌صورت یک **پکت استاندارد ۷ بایتی** به همراه Checksum باز می‌گرداند تا در مراحل بعدی مورد استفاده قرار گیرد.

ورودی‌ها:

- CtrlReq: پکت ورودی از Control Unit.
- AluReq: پکت ورودی از ALU.
- InChoose: انتخاب ورودی (0= CtrlReq، 1 = AluReq).
- Clk: سیگنال کلاک.
- RST: سیگنال ریست.

خروجی‌ها:

- ReadResp: پکت پاسخ خواندن از RAM.
- Error: سیگنال خطا (آدرس نامعتبر یا نوع عملیات غیرمجاز).

```
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 (  CtrlReq : in  data_packet;                  -- Input data packet
18                AluReq : in  data_packet;
19                ReadResp : out data_packet;                 -- Output of read mode
20                ReadRespReady : out STD_LOGIC;
21                InChoose : in STD_LOGIC;
22                Error : out STD_LOGIC;                      -- Address Out of Band / CheckSum Fail
23                RST : in  STD_LOGIC;
24                clk : in  STD_LOGIC
25              );
26    end RAM;
```

ساختار حافظه:

درون ماژول، ساختاری به نام Memory تعریف شده است که یک آرایه ۳۲ خانه‌ای از نوع byte (یعنی ۸ بیتی) است. به هر خانه از این حافظه از طریق آدرس‌دهی مستقیم با مقدار InPack(1) دسترسی پیدا می‌شود.

```
28    architecture Behavioral of RAM is
29
30        signal Memory : ram_matrix := (others => (others => '0')); --(others => '0')));    -- initial value is zerop
31
32    begin
33
34        process(clk)
35            variable InPack : data_packet;
36            variable Mode : packet_type;
37            variable RowAddress : integer range 0 to 31;
38    --      variable ColAddress : integer range 0 to 7;
39            variable WriteData : byte;
40            variable cash : data_packet;
41
42        begin
43            if rising_edge(clk) then
44                if RST = '1' then
45                    Memory <= (others => (others => '0'));
46                    ReadRespReady <= '0';
47                    InPack :=(others => (others => '0'));
48                    WriteData := (others => '0');
49                    cash := (others => (others => '0'));
50                else
51                    if InChoose = '1' then
52                        InPack := AluReq;
53                    else
54                        InPack := CtrlReq;
55                    end if;
56                    Error <= '0';
57
58                    if InPack(0) = "00001111" then  -- Function
59                        Mode := Rea_d;
60                    elsif InPack(0) = "11110000" then
61                        Mode := Writ_e;
62                    else
63                        Mode := zero;
64                        Error <= '1';
65                    end if;
```

نحوه عملکرد ماژول RAM:

در هر لبه بالارونده کلاک:

۱. **اگر سیگنال Reset فعال باشد**، تمام خانه‌های حافظه با صفر مقداردهی مجدد می‌شوند و سیستم به حالت اولیه باز می‌گردد.

۲. **در حالت عادی:**

- ابتدا مشخص می‌شود که ورودی از طرف ALU خوانده شود یا از طرف ControlUnit، و در نتیجه InPack با مقدار مناسب مقداردهی می‌شود.
- بررسی می‌شود که آیا بایت اول InPack(0) معادل "00001111" (برای Read) یا "11110000"(برای Write) است.
- اگر نباشد، نوع عملکرد zero تلقی شده و سیگنال Error فعال می‌شود.

اگر نوع عملکرد Writ_e باشد:

- آدرس از InPack(1) استخراج شده و به عدد صحیح بین ۰ تا ۳۱ تبدیل می‌شود.
- مقدار InPack(2) در خانه مشخص‌شده از حافظه نوشته می‌شود.
- اگر آدرس از ۳۱ بیشتر باشد، عملیات انجام نمی‌شود و Error فعال می‌شود.

اگر نوع عملکرد Rea_d باشد:

- داده‌ی ذخیره‌شده در آدرس مورد نظر از حافظه استخراج می‌شود.
- سپس یک پکت خروجی ۷ بایتی ساخته می‌شود:
  ۱. بایت اول برابر "11001111"به‌عنوان نشانه‌ی پاسخ حافظه
  ۲. بایت دوم مقدار خوانده‌شده از حافظه
  ۳. سه بایت میانی cash(2) تا cash(4) با صفر مقداردهی می‌شوند
  ۴. سپس Checksum بالا ((cash(5)) و پایین ((cash(6)) محاسبه می‌شود.
- این پکت در خروجی ReadResp قرار می‌گیرد تا به ماژول‌های بعدی منتقل شود.

Checksumها:

برای تضمین صحت داده‌های خروجی از حافظه، از توابع CheckSumH و CheckSumL که در فایل Packages تعریف شده‌اند استفاده می‌شود. این توابع مجموع ۵ بایت اول پکت را محاسبه کرده و به‌ترتیب ۸ بیت بالا و پایین آن را در بایت‌های ۵ و ۶ ذخیره می‌کنند.

```
66
67                  RowAddress := to_integer(unsigned(InPack(1))); --(7 downto 3)));
68      --          ColAddress := to_integer(unsigned(InPack(1)(2 downto 0)));
69                  WriteData := InPack(2);
70                  if (to_integer(unsigned(InPack(1))) > 31) then
71                      Error <= '1';
72                  else
73                      case Mode is
74                          when Writ_e =>
75                              -- Write Operation
76                              Memory(RowAddress) <= WriteData;
77                          when Rea_d =>
78                              -- Read Operation
79                              cash(0) := "11001111";
80                              cash(1) := Memory(RowAddress);
81                              cash(2) := (others => '0');
82                              cash(3) := (others => '0');
83                              cash(4) := (others => '0');
84                              cash(5) := CheckSumH(cash);
85                              cash(6) := CheckSumL(cash);
86                              ReadResp <= cash;
87                              ReadRespReady <= '1';
88                          when others =>
89                              Error <= '1';
90                      end case;
91                  end if;
92              end if;
93          end if;
94      end process;
95
96  end Behavioral;
```

۳-۶. ماژول ALU و Error Detection

هدف ماژول ALU این است که عملیات‌های منطقی و حسابی متنوعی مانند **جمع، تفریق، AND و OR** را روی داده‌هایی که از حافظه خوانده شده‌اند یا در پکت ورودی موجودند، انجام دهد و نتیجه را به حافظه برگرداند. این ماژول به‌طور مستقیم به RAM متصل است و بسته به نوع عملکرد پکت، از حافظه می‌خواند، عملیات را انجام می‌دهد و نتیجه را ذخیره می‌کند.

در کنار ALU، ماژول کوچکی به نام **ErrorDetection** نیز تعریف شده که وظیفه ترکیب و گزارش نهایی هرگونه خطا در سیستم را بر عهده دارد. این خطاها می‌توانند از دیکودر، کنترل یونیت، RAM یا ALU منشأ گرفته باشند.

ورودی‌ها:

```
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 ( InPack : in  data_packet;
17             PackMode : in  packet_type;
18
19             SendToRam : out data_packet;
20             ReadResponse : in data_packet;
21             Finish : inout STD_LOGIC;            -- Is 1 when the process is done.
22
23             Enable : in STD_LOGIC;               -- Alu is On only when we are in Alu Mode.
24             Error : out STD_LOGIC;
25             RST : in STD_LOGIC;
26             clk : in STD_LOGIC
27            );
28  end ALU;
```

- InPack: پکت دریافتی از کنترل یونیت.
- PackMode: نوع عملکرد پکت (مانند Operand_Alu ، Immediate_Alu و...).
- ReadResponse: داده‌هایی که از RAM برای خواندن دریافت شده‌اند.
- Enable: فعال‌سازی ALU فقط زمانی که پکت مربوط به ALU باشد.
- RST: ریست تمام مقادیر داخلی.
- Clk: سیگنال ساعت.

خروجی‌ها:

- SentToRam: پکت ۷بایتی خروجی که برای نوشتن به حافظه ارسال می‌شود.
- Finish: سیگنالی برای اعلام پایان عملیات.
- Error: سیگنال گزارش خطا.

نحوه عملکرد ماژول ALU:

ماژول ALU دارای حالت‌های مختلف است که بسته به نوع پکت در PackMode رفتار متفاوتی از خود نشان می‌دهد.
در همه حالت‌ها، روند کلی به شکل زیر است:

۱. بررسی نوع عملیات بر اساس ۲ بیت آخر InPack(0)

۲. استخراج آدرس‌ها، داده‌ها یا پارامترها از پکت

۳. خواندن داده‌ها از RAM از طریق ReadResponse

۴. انجام عملیات منطقی/ریاضی

۵. ساخت پکت خروجی و ارسال آن برای نوشتن در RAM

۶. گزارش پایان عملیات از طریق سیگنال Finish

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

```
61  begin
62
63      process(clk)
64          -- Ram Interaction
65          variable mode : byte := (others => '0');
66          variable RamAddress : byte := (others => '0');
67          variable RamDataToWrite : byte := (others => '0');
68          variable AddressI : byte := (others => '0');
69          variable AddressII : byte := (others => '0');
70          variable AddAddressII : byte := (others => '0');
71
72          -- Calculator Interaction
73
74      begin
75          if rising_edge(clk) then
76              Error <= '0';
77              if RST = '1' then
78                  Error <= '0';
79                  Step <= 0;
80                  DataI <= (others => '0');
81                  DataII <= (others => '0');
82                  DestinationAddress <= (others => '0');
83                  ArrayLength <= 0;
84                  ArrayIndPusher <= 0;
85                  ReadArray <= (others => (others => '0'));
86                  Finish <= '1';
87              elsif Enable = '1' then
88                  if (Finish = '1') then              -- 1 * Clk ->
89                      Finish <= '0';
90                      case InPack(0)(1 downto 0) is
91                          when "00" =>
92                              operation <= Add;
93                          when "01" =>
94                              operation <= Sub;
95                          when "10" =>
96                              operation <= BitwiseOr;
97                          when "11" =>
98                              operation <= BitwiseAnd;
99                          when others =>
100                     end case;
101                 end if;
```

۱.عملیات با دو داده حافظه (Operand_Alu)

- از دو آدرس InPack(1) و InPack(2) داده‌ها را از حافظه می‌خواند.
- آن‌ها را با عملیات مشخص‌شده ترکیب کرده و نتیجه را در آدرس InPack(3) می‌نویسد.
- این عملیات طی **۳ سیکل کلاک** انجام می‌شود.

۲.عملیات با داده ثابت (Immediate_Alu)

- یکی از داده‌ها مستقیماً در پکت (InPack(2))قرار دارد.
- داده دیگر از حافظه خوانده می‌شود.
- نتیجه عملیات در آدرس InPack(3)ذخیره می‌شود.
- این عملیات طی **۲ سیکل کلاک** انجام می‌شود.

۳.عملیات روی آرایه از حافظه (Array_Alu)

- از آدرس InPack(1) داده‌ها را پشت سر هم می‌خواند.

```
103                 case PackMode is
104                     when Operand_Alu => -- 3 Clocks
105                         if Step = 0 then                            -- Clock 0 till 1
106                             DestinationAddress <= InPack(3);
107                             AddressI := InPack(1);
108                             RamAddress := AddressI;
109                             mode := "00001111";
110                         elsif Step = 1 then                         -- Clock 1 till 2
111                             DataI <= ReadResponse(1);
112                             AddressII := InPack(2);
113                             RamAddress := AddressII;
114                             mode := "00001111";
115                         elsif Step = 2 then                         -- Clock 2 till 3
116                             --DataII <= ReadResponse(1);
117                             RamDataToWrite := Operator(signed(DataI), signed(ReadResponse(1)), Operation);
118                             RamAddress := DestinationAddress;
119                             mode := "11110000";
120                             Step <= -1;                             -- Will + 1
121                             Finish <= '1';
122                         end if;
123 
124                     when Immediate_Alu =>
125                         if Step = 0 then                            -- Clock 0 till 1
126                             DataII <= InPack(2);
127                             DestinationAddress <= InPack(3);
128                             AddressI := InPack(1);
129                             RamAddress := AddressI;
130                             mode := "00001111";
131                         elsif Step = 1 then                         -- Clock 1 till 2
132                             --DataI <= ReadResponse(1);
133                             RamDataToWrite := Operator(signed(ReadResponse(1)), signed(DataII), Operation);
134                             RamAddress := DestinationAddress;
135                             mode := "11110000";
136                             Step <= -1;
137                             Finish <= '1';
138                         end if;
139 
140                     when Array_Alu =>
141                         DataII <= InPack(2);
142                         ArrayLength <= to_integer(unsigned(InPack(3)));
143                         DestinationAddress <= InPack(4);
144                         if to_integer(unsigned(InPack(3))) > 32 then
145                             Error <= '1';
146                         end if;
```

- داده دوم ثابت است (InPack(2)) .
- به تعداد InPack(3) خانه حافظه پیمایش و روی آن عملیات انجام می‌شود.
- نتایج در خانه‌های پشت سر هم از InPack(4) ذخیره می‌شوند.

- اگر طول آرایه بیشتر از ۳۲ باشد، سیگنال Error فعال می‌شود.
- این عملیات طی مقدار **ArrayLength** **سیکل کلاک** انجام می‌شود.

۴.آدرس غیر مستقیم (Indirect_Addressing)

- ابتدا از InPack(1) داده‌ای خوانده می‌شود.
- سپس از InPack(2) آدرسی خوانده می‌شود که حاوی آدرس دوم واقعی است.

```
140                     when Array_Alu =>
141                         DataII <= InPack(2);
142                         ArrayLength <= to_integer(unsigned(InPack(3)));
143                         DestinationAddress <= InPack(4);
144                         if to_integer(unsigned(InPack(3))) > 32 then
145                             Error <= '1';
146                         end if;
147 
148                         if (Step > ArrayLength and Finish = '0')  then          -- Clock * size -> max 32
149                             mode := "11110000";
150                             RamDataToWrite := Operator(signed(ReadArray(ArrayIndPusher)), signed(DataII), Operation); -- Initail ArrayIndPusher = 0
151                             RamAddress := byte((unsigned(DestinationAddress) + to_unsigned(ArrayIndPusher, 8)) mod 32);
152                             ArrayIndPusher <= ArrayIndPusher + 1;
153                             if ArrayIndPusher = (ArrayLength - 1) then
154                                 Step <= -1;                         -- Will + 1
155                                 Finish <= '1';
156                                 ArrayIndPusher <= 0;
157                             end if;
158                         else
159                             if Step = 0 then                            -- Clock * size -> max 32
160                                 --AddressI := InPack(1);
161                                 ReadArray <= (others => (others => '0'));
162                                 mode := "00001111";
163                             end if;
164                             if Step > 0 then
165                                 ReadArray(Step - 1) <= ReadResponse(1);
166                             end if;
167                             RamAddress := byte((unsigned(InPack(1)) + to_unsigned(ArrayIndPusher, 8)) mod 32);
168                             ArrayIndPusher <= ArrayIndPusher + 1;
169                             if Step = ArrayLength then
170                                 ArrayIndPusher <= 0;
171                             end if;
172                         end if;
```

- از این آدرس دوم داده خوانده شده و عملیات انجام می‌شود.
- نتیجه در InPack(3) ذخیره می‌شود.

- این عملیات طی **۴ سیکل کلاک** انجام می‌شود.

```
174                     when Indirect_Addressing =>
175                         if Step = 0 then                                    -- Clock 0 till 1
176                             DestinationAddress <= InPack(3);
177                             AddressI := InPack(1);
178                             RamAddress := AddressI;
179                             mode := "00001111";
180                         elsif Step = 1 then                                 -- Clock 1 till 2
181                             AddAddressII := InPack(2);
182                             DataI <= ReadResponse(1);
183                             mode := "00001111";
184                             RamAddress := AddAddressII;
185                         elsif Step = 2 then                                 -- Clock 2 till 3
186                             AddressII := ReadResponse(1);
187                             RamAddress := AddressII;
188                             mode := "00001111";
189                         elsif Step = 3 then                                 -- Clock 3 till 4
190                             --DataII <= ReadResponse(1);
191                             RamDataToWrite := Operator(signed(DataI), signed(ReadResponse(1)), Operation);
192                             RamAddress := DestinationAddress;
193                             mode := "11110000";
194                             Step <= -1;                                     -- Will + 1
195                             Finish <= '1';
196                         end if;
197                     when others =>
198                         Error <= '1';
199                 end case;
200                 Step <= Step + 1;
201                 SendToRam(0) <= mode;
202                 SendToRam(1) <= RamAddress;
203                 SendToRam(2) <= RamDataToWrite;
204                 SendToRam(3) <= (others => '0');
205                 SendToRam(4) <= (others => '0');
206                 SendToRam(5) <= (others => '0');
207                 SendToRam(6) <= (others => '0');
208             end if;
209         end if;
210     end process;
211
212 end Behavioral;
```

نتیجه هر عملیات داخل متغیر Output ذخیره شده و به‌همراه اطلاعات آدرس و نوع عملیات در قالب یک پکت کامل به نام SentToRamPack ساخته می‌شود. در پایان، مقادیر Checksum با صفر مقداردهی شده و پکت خروجی نهایی از طریق خروجی SentToRam ارسال می‌شود.

مدیریت خطا

در موارد زیر سیگنال Error برابر با ۱ قرار می‌گیرد:

- فعال بودن RST
- نوع پکت ناصحیح یا غیرقابل تشخیص
- طول آرایه بزرگ‌تر از ۳۲
- اشکال در آدرس‌دهی غیرمستقیم یا پردازش داخلی

ماژول بسیار ساده‌ای به نام **ErrorDetection** در پروژه تعریف شده که تمامی سیگنال‌های خطا از ماژول‌های مختلف شامل دیکودر، کنترل یونیت، RAMو ALU را دریافت کرده و در صورتی که هرکدام از آن‌ها فعال باشند، سیگنال نهایی Error را روشن می‌کند. فرمول منطقی این کار:

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

به این ترتیب، سیستم همیشه از وقوع خطا در هر یک از بخش‌ها مطلع می‌شود.

```
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 ماژول PackToByte این است که داده‌های ۷ بایتی موجود در خروجی RAM (یعنی یک پکت کامل data_packet) را به‌صورت بایت‌به‌بایت استخراج کرده و آن‌ها را برای ارسال به انکودر آماده کند. این فرآیند شامل سازمان‌دهی مجدد پکت به صورت ram_resp_pack است که تنها شامل ۴ بایت است.

ورودی PackIn: پکت ۷بایتی دریافتی از RAM که شامل Function ، داده، و Checksum است.

خروجی ByteOut: بایتی که باید به صورت سریالی برای تبدیل به بیت به ماژول بعدی ارسال شود.

سیگنال clk: سیگنال کلاک برای همگام‌سازی عملیات.

در ابتدا، ماژول یک آرایه کمکی به نام PacketCash از نوع ram_resp_pack(۴ بایتی) را مقداردهی اولیه می‌کند:

- PacketCash(0) برابر "11001111" (کد شناسه پاسخ خواندن از RAM)
- PacketCash(2) برابر "00000000" (رزرو یا مقدار صفر)

سپس در لبه‌ی بالارونده کلاک، با استفاده از یک شمارنده داخلی CellCnt، در هر سیکل یکی از این بایت‌ها را در خروجی قرار می‌دهد.

- در اولین سیکل (CellCnt = 4) داده‌های اصلی از PackIn(1) (داده RAM) و PackIn(6) (CheckSum) به آرایه PacketCash منتقل می‌شوند.
- سپس از CellCnt = 1 تا CellCnt = 3، خروجی‌ها به ترتیب از PacketCash خوانده شده و در ByteOutقرار می‌گیرند.

```
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 این ماژول این است که یک بایت (۸ بیت) را به‌صورت سریالی و بیت‌به‌بیت در هر سیکل کلاک روی خروجی قرار دهد تا مستقیماً به انکودر همینگ ارسال شود.

```
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** ByteIn: بایت ۸ بیتی که باید به بیت‌های مجزا تقسیم شود.
- **خروجی** BitOut: خروجی سریالی که در هر کلاک یکی از بیت‌های ByteIn را تولید می‌کند.
- **سیگنال** clk: برای همگام‌سازی عملیات.

نحوه عملکرد ماژول:

- در ابتدا، یک سیگنال کمکی ByteCash مقدار بایت ورودی را ذخیره می‌کند.
- شمارنده داخلی BitCnt مشخص می‌کند که کدام بیت از ByteCash باید در BitOut قرار گیرد.
  - وقتی $BitCnt = 8$ باشد (یعنی ابتدای دریافت بایت جدید)، ByteCash مقدار جدیدی می‌گیرد و بیت صفرم آن در BitOut قرار می‌گیرد.
  - در سیکل‌های بعدی (تا $BitCnt = 7$) بیت‌های باقی‌مانده از ByteCash به ترتیب در خروجی قرار می‌گیرند.

ارتباط دو ماژول PackToByte و ByteToBit

۱. PackToByte خروجی RAM را به ترتیب در بایت‌های مجزا قرار می‌دهد.

۲. ByteToBit هر بایت را به ۸ بیت جداگانه تبدیل کرده و برای رمزگذاری Hamming آماده می‌کند.

این دو ماژول در کنار هم، وظیفه دارند داده ۷ بایتی خروجی RAM را به‌صورت سریالی و بیت‌به‌بیت به Hamming Encoder تحویل دهند، تا برای ارسال در سیستم ارتباطی مورد استفاده قرار گیرد.

۳-۹. تاپ ماژول Top Module

ماژول TopModuleبه‌عنوان **واحد تجمیع‌کننده**، تمام بخش‌های طراحی‌شده در سیستم را به یکدیگر متصل می‌کند و فرآیند کلی ارسال و دریافت داده به همراه پردازش منطقی را مدیریت می‌نماید. این ماژول مانند اسکلت اصلی سیستم عمل کرده و ارتباط بین ماژول‌های مختلف را هماهنگ می‌کند.

```
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
20             Output : out STD_LOGIC;
21                Error : out STD_LOGIC;
22
23             RST : in STD_LOGIC;
24             clk : in STD_LOGIC
25                );
26  end TopModule;
27
28  architecture Behavioral of TopModule is
29
30      signal DataByte : byte;
31      signal DecValidation : STD_LOGIC;
32
33      signal Switch : STD_LOGIC;
34      signal Packet : data_packet;
35      signal PacketValidation : STD_LOGIC;
36
37      --signal PackToRam : data_packet;
38      signal RamReadResp : data_packet;
39      signal RamError : STD_LOGIC;
40
41      signal AluEnable : STD_LOGIC;
42      --signal PackToAlu : data_packet;
43      signal PackType : packet_type;
44      signal AluToRam : data_packet;
45      --signal RamToAlu : ram_resp_pack;
46      signal AluDone : STD_LOGIC;
47      signal AluError : STD_LOGIC;
48      --signal RamRespError : STD_LOGIC;
49
50      signal EncByte : byte;
51      signal EncInBit : STD_LOGIC;
```

ورودی‌ها و خروجی‌ها

- Input: ورودی سریالی ۱ بیتی که داده رمزگذاری‌شده (Hamming)را دریافت می‌کند.
- Output: خروجی سریالی ۱ بیتی که داده رمزگذاری‌شده نهایی را ارسال می‌کند.
- Error: سیگنالی برای نشان‌دادن وقوع خطا در هر مرحله از فرآیند.
- RST: سیگنال Reset سراسری برای بازنشانی کل سیستم.
- Clk: سیگنال کلاک برای زمان‌بندی ماژول‌ها.

نحوه عملکرد تاپ ماژول

ماژول اصلی شامل ۸ ماژول زیر است که به ترتیب متصل شده‌اند و با سیگنال‌های میانی با یکدیگر تعامل دارند:

۱.HammingDecoder

- ورودی سریالی Input را دریافت می‌کند.
- پس از بررسی توازن و تصحیح احتمالی خطا، داده ۸ بیتی خروجی می‌دهد.
- خروجی:
  - DataByte: بایت رمزگشایی‌شده

```
begin

    Decoder: entity work.HammingDecoder
        port map
        (
            DecInBit => Input,
            DecOutByte => DataByte,
            Valid => DecValidation,
            RST => RST,
            clk => clk
        );

    CtrlUnit: entity work.ControlUnit
        port map
        (
            InByte => DataByte,
            Validation => PacketValidation,
            Switch => Switch,
            Packet => Packet,
            PackType => PackType,
            RST => RST,
            clk => clk
        );

    RAM: entity work.RAM
        port map
        (
            InChoose => AluEnable,
            CtrlReq => Packet,
            AluReq => AluToRam,
            ReadResp => RamReadResp,
            Error => RamError,
            RST => RST,
            clk => clk
        );
```

- DecValidation: اعتبار داده پس از رمزگشایی

۲. ControlUnit

- داده ۸ بیتی را از DataByte می‌گیرد.
- بسته (Packet) متشکل از چند بایت ساخته و نوع عملکرد آن(ALU یا RAM) را مشخص می‌کند.
- خروجی‌ها:
  - Packet: پکت ساخته‌شده
  - PackMode: نوع پکت (خواندن، نوشتن، عملیات منطقی و ...)
  - Switch: مشخص‌کننده ALU mode یا RAM mode
  - PacketValidation: اعتبار پکت ساخته‌شده) بر اساس Checksum)

۳. RAM

- بسته به Switch، بین ورودی‌های CtrlReq (از ControlUnit) یا AluReq (از ALU) یکی را انتخاب می‌کند.
- اگر عملکرد نوشتن باشد، داده را در حافظه ذخیره می‌کند.
- اگر عملکرد خواندن باشد، داده را از حافظه بازیابی کرده و در قالب پکت ۷بایتی خروجی می‌دهد.
- خروجی‌ها:
  - RamReadResp: پکت حاوی پاسخ خواندن از RAM

- RamError: خطای دسترسی به حافظه یا Checksum نادرست

۴. ALU (Arithmetic Logic Unit)

- اگر نوع پکت نیازمند عملیات منطقی باشد، فعال می‌شود.
- داده را از RAM می‌گیرد و پس از پردازش، نتیجه را برای نوشتن به RAM آماده می‌کند.
- خروجی‌ها:
  - AluToRam: پکت خروجی شامل نتیجه عملیات
  - AluDone: اتمام عملیات
  - AluError: اگر مشکلی رخ دهد(مثلاً overflowیا دسترسی غیرمجاز)، فعال می‌شود.

۵. PackToByte

- پکت ۷بایتی دریافتی از RAM را به بایت‌های مجزا تقسیم می‌کند.
- هر بار، یک بایت را در خروجی قرار می‌دهد.

۶. ByteToBit

- بایت خروجی را به بیت‌های مجزا (از بیت ۰ تا ۷) در هر سیکل کلاک تقسیم می‌کند.
- خروجی سریالی تولید می‌کند.

۷. HammingEncoder

- بیت‌های ورودی را دریافت کرده و با محاسبه بیت‌های توازن(Parity) ، آن را به کد Hamming ۱۳ بیتی تبدیل کرده و به صورت سریالی ارسال می‌کند.

```
89          ALU: entity work.ALU
90              port map
91              (
92                  Enable => AluEnable,
93                  InPack => Packet,
94                  PackMode => PackType,
95                  SendToRam => AluToRam,
96                  ReadResponse => RamReadResp,
97                  Finish => AluDone,
98                  Error => AluError,
99                  RST => RST,
100                 clk => clk
101             );
102
103         PacketToByte: entity work.PackToByte
104             port map
105             (
106                 PackIn => RamReadResp,
107                 ByteOut => EncByte,
108                 clk => clk
109             );
110
111         ByteToBit: entity work.ByteToBit
112             port map
113             (
114                 ByteIn => EncByte,
115                 BitOut => EncInBit,
116                 clk => clk
117             );
118
119         Encoder: entity work.HammingEncoder
120             port map
121             (
122                 BitIn => EncInBit,
123                 BitOut => Output,
124                 RST => RST,
125                 clk => clk
126             );
```

- خروجی نهایی از طریق Output به صورت سریال خارج می‌شود.

۸. ErrorDetection

- این ماژول بررسی می‌کند آیا در بخش‌های مختلف سیستم خطایی رخ داده است یا خیر:
  - خطای رمزگشایی (DecValidation)
  - خطای صحت پکت (PacketValidation)
  - خطای RAM یا ALU
- اگر هر کدام از این خطاها فعال باشند، سیگنال Error برابر ۱ می‌شود.

```
128         ErrorDetection: entity work.ErrorDetection
129             port map
130             (
131                 DecodingError => not(DecValidation),
132                 PacketError => not(PacketValidation),
133                 RamError => RamError,
134                 AluError => AluError,
135                 Error => Error
136             );
137         AluEnable <= Switch or (not(AluDone));
138 
139     end Behavioral;
```

مدیریت حالت ALU

- سیگنال AluEnable تعیین می‌کند که ماژول ALU فعال باشد یا نه.
- با استفاده از: AluEnable <= Switch or (not(AluDone));

این شرط باعث می‌شود ALU تنها زمانی فعال شود که یا در ALU mode باشیم (Switch = 1) یا عملیات قبلی هنوز به اتمام نرسیده باشد.(AluDone = 0)

ماژول TopModule در واقع سیستم کامل انتقال داده رمزگذاری‌شده است که:

۱. داده سریالی را دریافت می‌کند،

۲. آن را رمزگشایی کرده،

۳. در صورت نیاز عملیات منطقی انجام می‌دهد یا در حافظه ثبت می‌کند،

۴. خروجی را مجدداً رمزگذاری کرده و سریالی ارسال می‌کند.

هم‌زمان با این فرآیند، کلیه خطاها به‌صورت سراسری تحت نظارت قرار دارند و سیگنال Error در صورت وقوع مشکل فعال می‌شود.

فصل چهارم- تست بنچ ماژول‌های سیستم

۱.تست بنچ ماژول RAM

هدف از این تست بنچ اعتبارسنجی مسیرهای نوشتن/خواندن RAM و انتخاب ورودی (CtrlReq/AluReq) و اعلام پاسخ خواندن (ReadResp + ReadRespReady) و وضعیت خطا است.

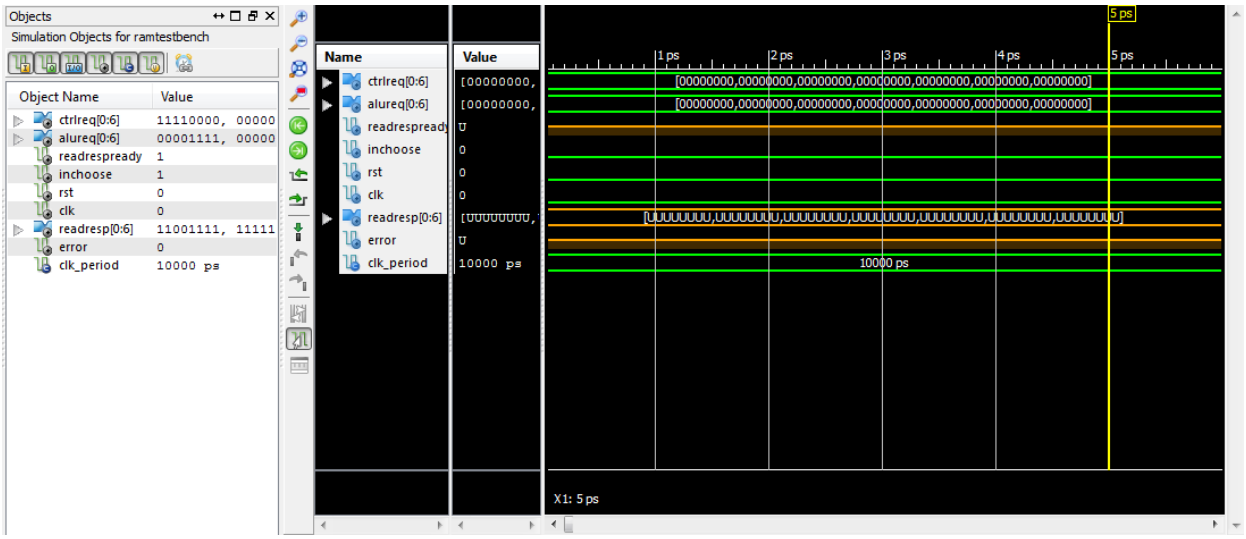
DUTو پورت‌ها

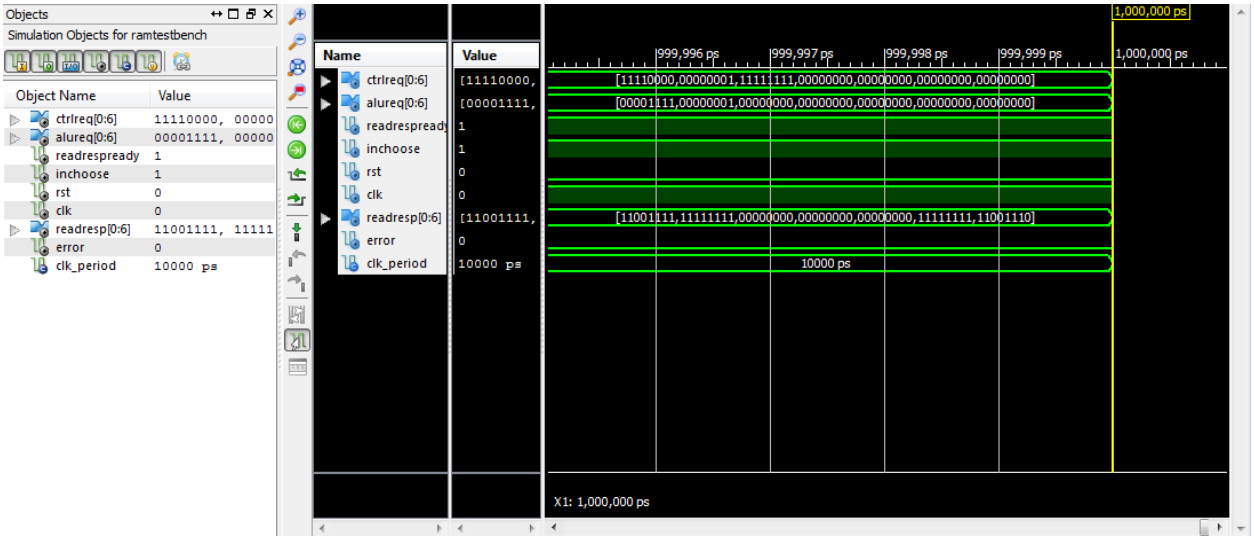
- CtrlReq, AluReq : data_packetورودی‌های بسته از کنترل وALU
- ReadResp : data_packetخروجی خواندن
- ReadRespReady : std_logicاعلان آمادگی پاسخ
- InChoose: انتخاب مسیر ورودی ctrl ←0, Alu←1
- Error: پرچم خطا
- RST, clk: ریست همزمان/کلاک

پیکربندی تست‌بنچ

- تناوب کلاک: clk_period = 10 ns
- راه‌اندازی: نگه‌داشتن RST='0' به‌مدت ۱۰ سیکل
- نگاشت UUT مطابق Entity پروژه

```
library IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;
USE work.Packages.All;

ENTITY RamTestBench IS
END RamTestBench;

ARCHITECTURE behavior OF RamTestBench IS

    -- Component Declaration for the Unit Under Test (UUT)

    COMPONENT RAM
    PORT(
         CtrlReq : IN  data_packet;
         AluReq : IN  data_packet;
         ReadResp : OUT  data_packet;
            ReadRespReady : OUT  std_logic;
         InChoose : IN  std_logic;
         Error : OUT  std_logic;
         RST : IN  std_logic;
         clk : IN  std_logic
        );
    END COMPONENT;


   --Inputs
   signal CtrlReq : data_packet := (others => (others => '0'));
   signal AluReq : data_packet := (others => (others => '0'));
    signal ReadRespReady : std_logic;
   signal InChoose : std_logic := '0';
   signal RST : std_logic := '0';
   signal clk : std_logic := '0';

    --Outputs
   signal ReadResp : data_packet;
   signal Error : std_logic;

   -- Clock period definitions
   constant clk_period : time := 10 ns;

BEGIN

     -- Instantiate the Unit Under Test (UUT)
   uut: RAM PORT MAP (
          CtrlReq => CtrlReq,
          AluReq => AluReq,
          ReadResp => ReadResp,
             ReadRespReady => ReadRespReady,
          InChoose => InChoose,
          Error => Error,
          RST => RST,
          clk => clk
        );

   -- Clock process definitions
   clk_process :process
   begin
        clk <= '0';
        wait for clk_period/2;
        clk <= '1';
        wait for clk_period/2;
   end process;


   -- Stimulus process
   stim_proc: process
   begin
      -- hold reset state for 100 ns.
      RST <= '0';
      wait for clk_period*10;
        RST <= '0';

        InChoose <= '0';
        CtrlReq <= ("11110000",
                    "00000001",
                    "11111111",
                  "00000000",
                  "00000000",
                  "00000000",
                  "00000000");
        wait for 10 ns;

        -- 11001111 + 111111111 = 11111111 11001110
        InChoose <= '1';
        AluReq <= ("00001111",
                    "00000001",
                    "00000000",
                  "00000000",
                  "00000000",
                  "00000000",
                  "00000000");
        wait for 10 ns;
    end process;

END;
```


Objects
Simulation Objects for ramtestbench
Object Name
Value
ctrlreq[0:6]
11110000, 00000
alureq[0:6]
00001111, 00000
readrespready
1
inchoose
1
rst
0
clk
0
readresp[0:6]
11001111, 11111
error
0
clk_period
10000 ps
Name
Value
[00000000,
[00000000,
U
0
0
0
[UUUUUUUU,
U
10000 ps
5 ps
1 ps
2 ps
3 ps
4 ps
5 ps
[00000000,00000000,00000000,00000000,00000000,00000000,00000000,00000000]
[00000000,00000000,00000000,00000000,00000000,00000000,00000000,00000000]
[UUUUUUUU,UUUUUUUU,UUUUUUUU,UUUUUUUU,UUUUUUUU,UUUUUUUU,UUUUUUUU,UUUUUUUU]
10000 ps
X1: 5 ps


Objects
Simulation Objects for ramtestbench
Object Name
Value
ctrlreq[0:6]
11110000, 00000
alureq[0:6]
00001111, 00000
readrespready
1
inchoose
1
rst
0
clk
0
readresp[0:6]
11001111, 11111
error
0
clk_period
10000 ps
Name
Value
[11110000,
[00001111,
1
1
0
0
[11001111,
0
10000 ps
1,000,000 ps
999,996 ps
999,997 ps
999,998 ps
999,999 ps
1,000,000 ps
[11110000,00000001,11111111,00000000,00000000,00000000,00000000,00000000]
[00001111,00000001,00000000,00000000,00000000,00000000,00000000,00000000]
[11001111,11111111,00000000,00000000,00000000,11111111,11001110]
10000 ps
X1: 1,000,000 ps

۲.تست بنچ ماژول Hamming decoder

هدف آزمون

دریافت کد همینگ ۱۳ بیتی، تصحیح تک‌بیت، اعلام اعتبار (Valid) و تولید بایت خروجی.

DUTو پورت‌ها

- DecInBit: بیت ورودی سریال
- TestBenchInputDisplay[0..12]
- OutRdy
- DecOutByte[7..0]
- Valid
- RST, clk

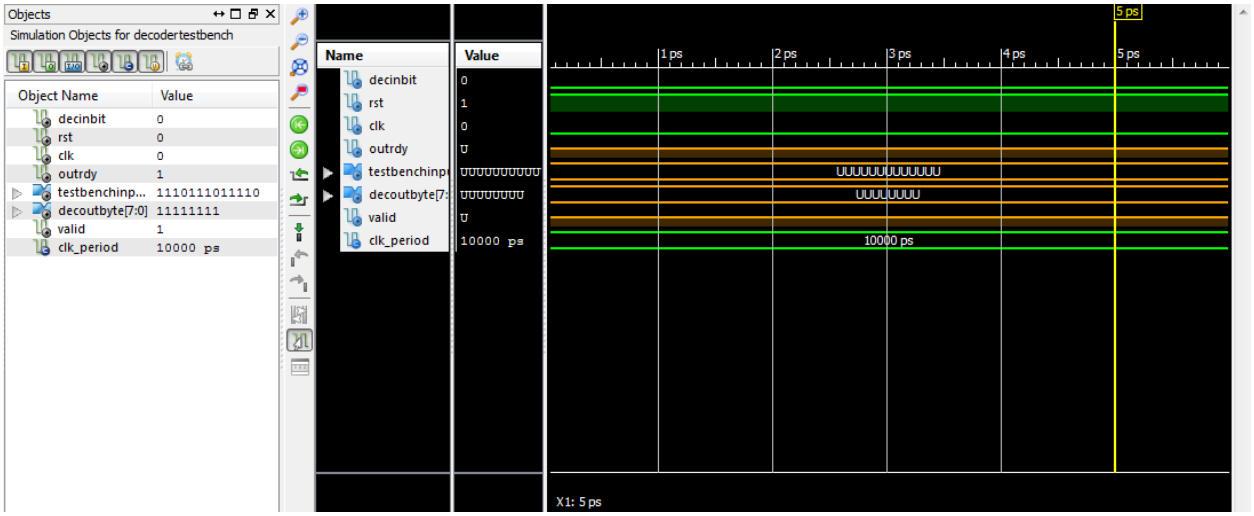
تست‌ها

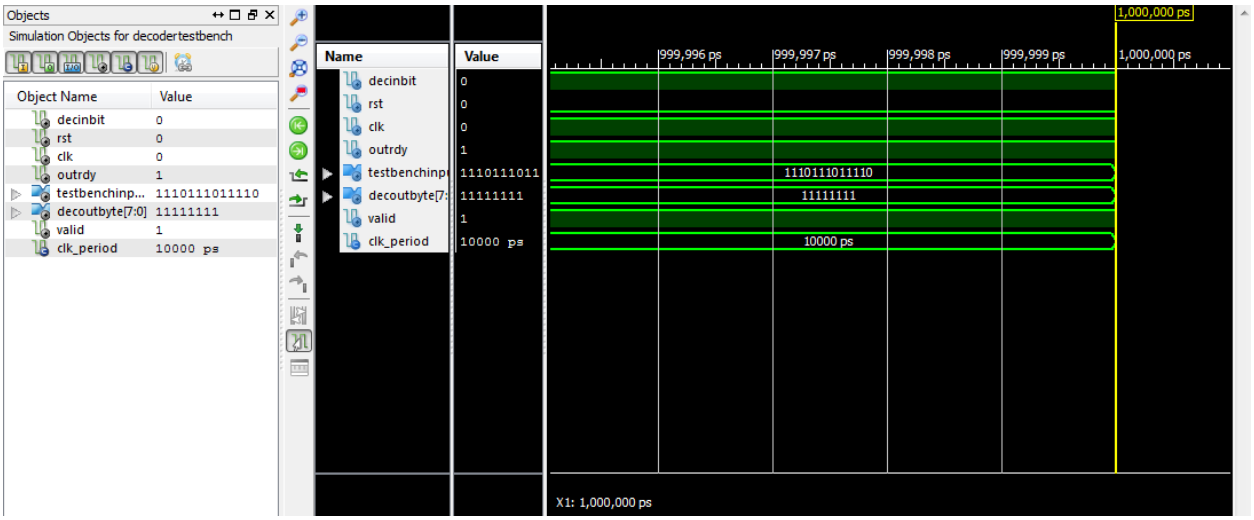
۱. 1110111011110 → 11111111
۲. 0110101100100 → 11010010
۳. 1110101100100 → 11010010 (۱خطا)
۴. 1111110011011 → Invalid (۲خطا)

معیار پذیرش

- تست ۱،۲،۳: Valid=1و داده درست
- تست ۴: Invalid

```
library IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;
USE work.Packages.ALL;

ENTITY DecoderTestBench IS
END DecoderTestBench;

ARCHITECTURE behavior OF DecoderTestBench IS

    -- Component Declaration for the Unit Under Test (UUT)

    COMPONENT HammingDecoder
    PORT(
         DecInBit : IN  std_logic;
         TestBenchInputDisplay : out STD_LOGIC_VECTOR (0 to 12);
         OutRdy : INOUT  std_logic;
         DecOutByte : OUT  std_logic_vector(7 downto 0);
         Valid : INOUT  std_logic;
         RST : IN  std_logic;
         clk : IN  std_logic
        );
    END COMPONENT;


   --Inputs
   signal DecInBit : std_logic := '0';
   signal RST : std_logic := '0';
   signal clk : std_logic := '0';

    --BiDirs
   signal OutRdy : std_logic;

    --Outputs
   signal TestBenchInputDisplay : STD_LOGIC_VECTOR (0 to 12);
   signal DecOutByte : std_logic_vector(7 downto 0);
   signal Valid : std_logic;

   -- Clock period definitions
   constant clk_period : time := 10 ns;

BEGIN

    -- Instantiate the Unit Under Test (UUT)
   uut: HammingDecoder PORT MAP (
          DecInBit => DecInBit,
          TestBenchInputDisplay => TestBenchInputDisplay,
          OutRdy => OutRdy,
          DecOutByte => DecOutByte,
          Valid => Valid,
          RST => RST,
          clk => clk
        );

   -- Clock process definitions
   clk_process :process
   begin
        clk <= '0';
        wait for clk_period/2;
        clk <= '1';
        wait for clk_period/2;
   end process;


   -- Stimulus process
   stim_proc: process
        variable Input : hamming;
   begin
      -- hold reset state for 100 ns.
      RST <= '1';
      wait for clk_period*10;
        RST <= '0';

        -- Rightest is first Psition.
      -- Test #1
        -- Input : "1110111011110" , Answer : "11111111"
        Input := "1110111011110";
        for i in 0 to 12 loop
            DecInBit <= Input(i);
            wait for 10 ns;
        end loop;
        wait for 2*clk_period;

        -- Test #2
        -- Input : "0110101100100" , Answer : "11010010"
        Input := "0110101100100";
        for i in 0 to 12 loop
            DecInBit <= Input(i);
            wait for 10 ns;
        end loop;
        wait for 2*clk_period;

        -- Test #3
        -- Input : "0110101100100" With 1 error , Answer : "11010010"
        Input := "1110101100100";
        for i in 0 to 12 loop
            DecInBit <= Input(i);
            wait for 10 ns;
        end loop;
        wait for 2*clk_period;

        -- Test #4
        -- Input : "1111110000011" With 2 error , Answer : "11110000"
        Input := "1111110011011";
        for i in 0 to 12 loop
            DecInBit <= Input(i);
            wait for 10 ns;
        end loop;
        wait for 2*clk_period;
    end process;


END;
```


Objects
Simulation Objects for decodertestbench
Object Name
Value
decinbit 0
rst 0
clk 0
outrdy 1
testbenchinp... 1110111011110
decoutbyte[7:0] 11111111
valid 1
clk_period 10000 ps
Name
Value
1 ps
2 ps
3 ps
4 ps
5 ps
UUUUUUUUUUUU
UUUUUUUU
10000 ps
X1: 5 ps


Objects
Simulation Objects for decodertestbench
Object Name
Value
decinbit 0
rst 0
clk 0
outrdy 1
testbenchinp... 1110111011110
decoutbyte[7:0] 11111111
valid 1
clk_period 10000 ps
Name
Value
999,996 ps
999,997 ps
999,998 ps
999,999 ps
1,000,000 ps
1110111011110
11111111
10000 ps
X1: 1,000,000 ps

۳.تست بنچ ماژول Hamming Encoder

هدف آزمون

تولید کد همینگ ۱۲ یا ۱۳ بیتی از ۸ بیت ورودی سریال و بررسی با داده نمونه.

DUTو پورت‌ها

- BitIn: بیت ورودی سریال
- BitOut: بیت خروجی کدگذاری‌شده
- TestBenchCheck[0..12]: بردار بررسی
- TestBenchInputDisplay[7..0]: نمایش ورودی
- OutRdy: آماده بودن خروجی
- RST, clk

پیکربندی تست‌بنچ

- کلاک ۱۰ns، ریست ۱۰ سیکل
- خوراندن ۸ بیت با فاصله ۱۰ns

تست‌ها

۱. ورودی11111111 ← خروجی011011101111

۲. ورودی01011100 ← خروجی 1000101011000

معیار پذیرش

- انطباق TestBenchCheck و BitOut با مقادیر انتظار