

به نام خدا



دانشگاه صنعتی امیرکبیر  
(پلی تکنیک 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
- ۲-۱. معرفی سیستم ارسال/دریافت داده
- ۳-۱. هدف پروژه
- ۴-۱. ساختار کلی سیستم طراحی شده

### • فصل دوم - مبانی نظری

- ۱-۲. کدگذاری Hamming و تشخیص/تصحیح خطا
- ۲-۲. قالب Packet و فیلدهای آن
- ۳-۲. Checksum و روش محاسبه آن
- ۴-۲. معرفی ALU, RAM و واحد کنترل (FSM)

### • فصل سوم - طراحی و پیاده‌سازی سیستم ماژول‌ها

- ۱-۳. Packages (انواع داده و توابع مشترک)
- ۲-۳. ماژول Hamming Decoder
- ۳-۳. ماژول Control Unit
- ۴-۳. ماژول RAM
- ۵-۳. ماژول ALU
- ۶-۳. ماژول Error Detection
- ۷-۳. ماژول PackToByte
- ۸-۳. ماژول ByteToBit
- ۹-۳. ماژول Hamming Encoder
- ۱۰-۳. Top Module (تجمیع و ارتباط ماژول‌ها)

### • فصل چهارم - تست‌بنج و نتایج شبیه‌سازی

- تست کلی سیستم (TopModule)

## 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 Decoder از کد Hamming به دیتا تبدیل شده، سپس ذخیره و پردازش می‌شود، و در نهایت با استفاده از Encoder بازسازی شده و صحت آن بررسی می‌شود.

### ۱-۳. هدف پروژه

هدف پروژه دریافت پکت‌های دیتاست تا با توجه به آن عملیاتی بر روی دیتای ذخیره شده در رم انجام شود. در این پروژه پکت‌ها به صورت بیت به بیت و سریالی دریافت می‌شوند که هر بایت ۸ بیتی در قالب ۱۳ بیت به صورت همینگ کد شده است که می‌بایست به کمک دیکودر دیکد شود. خروجی پکت داده‌ای ست که دستور خواندن آن از رم از ورودی صادر شده است. خروجی نیز بایستی به همان شیوه بایت به بایت در ۱۳ بیت به صورت همینگ کد شده و بیت به بیت خارج شود. (لازم به ذکر است که برای بررسی بهتر کارکرد کد خروجی‌های دیگری از جنس پکت نیز اضافه شده است.)

#### ۴-۱. ساختار کلی سیستم طراحی شده

سیستم طراحی شده از چندین ماژول اصلی تشکیل شده است که به صورت سلسله وار 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  $8 \times 32$  بیتی برای ذخیره داده‌ها

ALU : انجام عملیات Add/Sub/OR/AND

Control Unit : تشکیل دهنده پکت از بایت‌های ورودی و مشخص کردن نوع آن

ALU به گونه‌ای طراحی شده که بتواند پکت‌های مختلف با عملکردهای متفاوت (مثل Immediate, Operand,

Indirect, Array) را پردازش کند.

## فصل دوم) مبانی نظری و پایه‌ای

### ۱-۲. 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 به صورت 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 RAM، Encoder، ALU و Decoder از ساختار استاندارد 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 تعدادی از سلول‌های 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 داده

- Data: می‌تواند آدرس یک خانه یا مقداری برای افزودن یا کاستن یا ... به داده‌ی مورد نظر باشد.

- Destination Address: محل نهایی ذخیره‌سازی نتیجه

- Length: در عملیات‌هایی مثل Array ALU مشخص می‌کند عملیات باید روی چند خانه حافظه انجام شود.

- ChecksumH / ChecksumL: برای بررسی صحت داده در طول انتقال

ALU چهار کارکرد اصلی دارد:

- انجام عملیات بر روی داده‌های دو آدرس ورودی (Operand)

- انجام عملیات بر روی داده‌ی آدرس ورودی و داده‌ای که به طور مستقیم مقدار آن داده شده است (Immediate)

- انجام عملیات Immediate Alu بر روی آرایه‌ای از خانه‌های رم (Array)

- انجام عملیات بر روی آدرس داده شده و دیتایی که خود آدرس خانه‌ایست که در آن آدرس خانه دیتای دوم ذخیره شده است (Indirect Addressing)

RAM دو کارکرد اصلی دارد:

- خواندن / نوشتن داده در حافظه

## ۲-۳. Checksum و روش محاسبه آن

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

- **ChecksumL**: ۸ بیت کم ارزش مجموع

- **ChecksumH**: ۸ بیت پر ارزش 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 CU، RAM و ALU

- **Control Unit (واحد کنترل):**

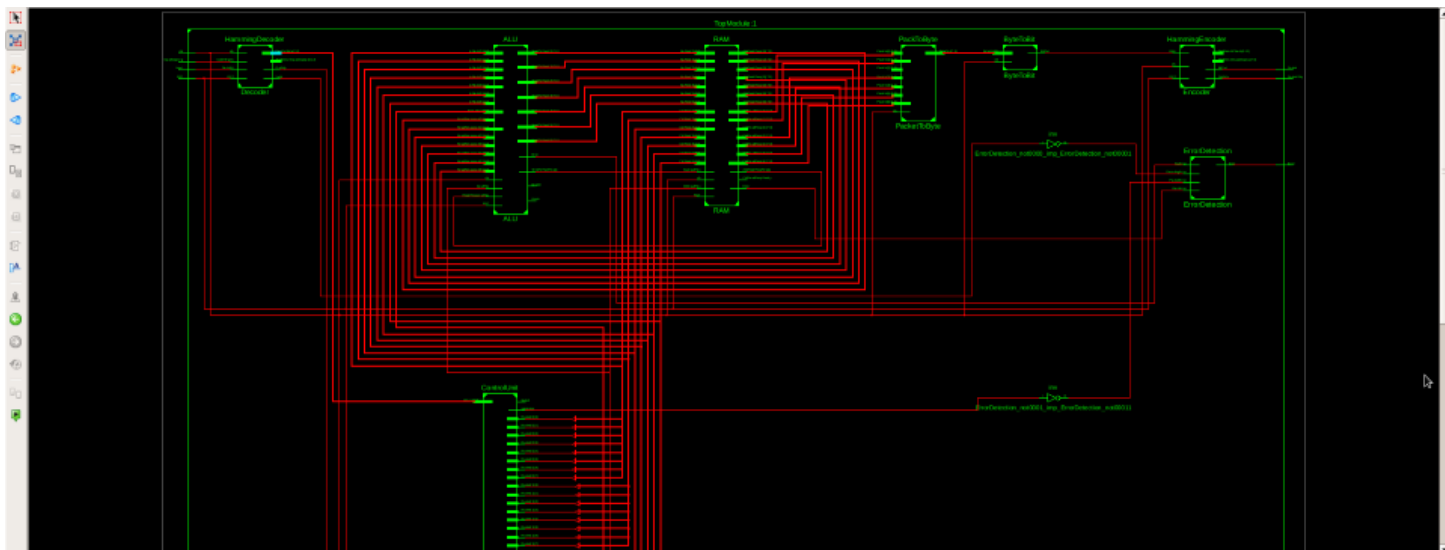
این ماژول بایت‌های خروجی دیکودر را به ترتیب دریافت کرده و یک پکت ۷ بیتی استاندارد می‌سازد. از روی بایت Function، نوع عملیات را به صورت RAM: Read/Write یا حالت‌های ALU تشخیص می‌دهد و Checksum را بررسی کرده و نتیجه را در Validation می‌دهد. در پایان هر پکت، اعلام آمادگی می‌کند تا همان پکت برای شاخه‌ی مناسب RAM یا ALU استفاده شود.

- **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 صفر می‌شوند.

- **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.





## 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 ۸ بیتی.
- hamming: بردار ۱۳ بیتی برای داده‌های کد همپینگ.
- 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, Write, Read, Immediate\_Alu, Array\_Alu, Indirect\_Addresssing).
- 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 عملیات Array 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, Write, Read, Immediate_Alu, Array_Alu, Indirect_Addresssing);
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 ۱۶ بیتی.
- CheckSumH(ByteSum): بایت پرارزش جمع.
- CheckSumL(ByteSum): بایت کم‌ارزش 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 end Packages;
69

```

نحوه 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.
- TopModule و همه‌ی ماژول‌ها به‌صورت مستقیم از این کتابخانه استفاده می‌کنند.

## ۳-۲. ماژول Hamming Decoder

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

### ورودی/خروجی ها (Entity HammingDecoder)

- ورودی ها:

std\_logic : DecInBit (جریان سریال ۱۳ بیت)

CodeBegins: std\_logic (آغاز فریم)

clk, RST

- خروجی ها:

DecOutByte: byte (۸ بیت هم زمان)

OutRdy: std\_logic (آمادگی خروجی)

Valid: std\_logic (اعتبار پس از تصحیح)

- همچنین پورت کمکی

TestBenchInputDisplay برای مشاهده

ورودی کد.

نحوه عملکرد ماژول دیکودر همینگ:

- رجیستر داخلی Code: hamming و

شمارنده cnt بیت ها را جمع می کنند.

- ورود: برای cnt < 13 هر کلاک

Code <= Code(1 to 12) & DecInBit

- محاسبه: (cnt=13) بیت های توازن

باز ساخته می شوند ←

```
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;
37
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 <= "00000000000000";
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 1 -  $\text{ind} := \text{to\_integer}(\text{unsigned}(\text{ErrorMarker}))$  و اگر  $0 \leq \text{ind} < 13$ ، بیت خراب flip می‌شود.

شرط اعتبار: اگر (بدون خطا و  $\text{ExtentionBit} = \text{Code}(12)$ ) یا (تک خطا و  $\text{ExtentionBit} = \text{not}$ )  
 $\text{Valid} \leq '0'$ ، آنگاه معتبر؛ در غیر این صورت  $\text{Valid} \leq '0'$ .

- 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
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. مازول Control Unit

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

#### ورودی/خروجی (Entity ControlUnit)

• ورودی‌ها:

std\_logic : InputIsReady , byte : InByte

clk , RST

• خروجی‌ها:

Packet : Packet , inout data\_packet : PacketType

Switch: std\_logic , inout packet\_type

Validation : std\_logic : PackIsReady

std\_logic

نحوه‌ی عملکرد مازول واحد کنترل:

• **Step=0**: پاک‌سازی Packet و ثبت

Packet(0) := InByte , سپس

دسته‌بندی **Function**:

○ Operand\_Alum برای

"00000000" تا "00000011"

○ Writ\_e برای "11110000"

Rea\_d برای "00001111"

○ Immediate\_Alum برای

"00111100" تا "00111111"

○ Array\_Alum برای "11000000" تا "11000011"

○ Indirect\_Addressingu برای "00110000" تا "00110011"

```
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           PacketType : 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;
30
31 architecture Behavioral of ControlUnit is
32     signal Step : integer := -1;
33
34 begin
35     process (clk)
36         variable PackToCheck : data_packet := (others => '0');
37     begin
38         if rising_edge(clk) then
39             PackIsReady <= '0';
40
41             if RST = '1' then
42                 Step <= 0;
43                 PackType <= zero;
44             elsif InputIsReady = '1' then
45                 case step is
46                     when 0 =>
47                         Packet <= (others => '0');
48                         Packet(0) <= InByte;
49
50                     ---
51                     case InByte is
52                         when "00000000" | "00000001" | "00000010" | "00000011" =>
53                             PackType <= Operand_Alum;
54                         when "11110000" =>
55                             PackType <= Writ_e;
56                         when "00001111" =>
57                             PackType <= Rea_d;
58                         when "00111100" | "00111101" | "00111110" | "00111111" =>
59                             PackType <= Immediate_Alum;
60                         when "11000000" | "11000001" | "11000010" | "11000011" =>
61                             PackType <= Array_Alum;
62                         when "00110000" | "00110001" | "00110010" | "00110011" =>
63                             PackType <= Indirect_Addressingu;
64                         when others =>
65                             PackType <= zero;
66                     end case;
67
68                     Step <= Step + 1;
69                 end case;
70             end if;
71         end process;
```

```

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_Addresssing) 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;

```

• Packet(1) := Step=1

؛InByte اگر Rea\_d بود،

پرش به Step=5 (چون فقط تا

بایت ۴ نیاز است).

• Packet(2) := Step=2

؛InByte اگر Writ\_e بود،

پرش به Step=5.

• Packet(3) := Step=3

؛InByte اگر نوع پکت یکی از

Operand/Immediate/Indirect بود، پرش به Step=5.

• Packet(4) := InByte Step=4 و ادامه.

• Packet(5) := InByte Step=5.

• Packet(6) := InByte Step=6؛ سپس:

ارتباط با سایر ماژول‌ها: خروجی پکت و نوع آن

به RAM/ALU می‌رود؛ Switch مسیر را

تعیین می‌کند.

```

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;

```

### ۳-۴. RAM ماژول

هدف: RAM با ۳۲ خانه ۸ بیتی؛ 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 CtrlReq, CtrlInputRdy و خروجی‌های

CtrlReadResp, CtrlReadRespReady

ورودی‌های AluReq, AluInputRdy و خروجی‌های

AluReadResp, AluReadRespReady

همچنین Error, RST, clk.

نحوه عملکرد ماژول رم:

• شاخه '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;
30
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';
```

```

66     if (to_integer(unsigned(CtrlReq(1))) > 31) then
67         Error <= '1';
68     else
69         case Mode is
70             when Writ_e =>
71                 -- Write Operation
72                 Memory(RowAddress) <= WriteData;
73             when Rea_d =>
74                 -- Read Operation
75                 cash(0) := "11001111";
76                 cash(1) := Memory(RowAddress);
77                 cash(2) := (others => '0');
78                 cash(3) := (others => '0');
79                 cash(4) := (others => '0');
80                 cash(5) := CheckSumH(cash);
81                 cash(6) := CheckSumL(cash);
82                 CtrlReadResp <= cash;
83                 CtrlReadRespReady <= '1';
84             when others =>
85                 end case;
86         end if;
87     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'

• شاخه ALU (AluInputRdy='1')

○ تشخیص Mode مشابه؛

RowAddress:= to\_integer(unsigned(AluReq(1)(4 downto 0))) (۵ بیت پایین)

```

88
89     if AluInputRdy = '1' then
90         if AluReq(0) = "00001111" then -- Function
91             Mode := Rea_d;
92         elsif AluReq(0) = "11110000" then
93             WriteData := AluReq(2);
94             Mode := Writ_e;
95         else
96             Mode := zero;
97         end if;
98
99         RowAddress := to_integer(unsigned(AluReq(1)(4 downto 0)));
100         case Mode is
101             when Writ_e =>
102                 -- Write Operation
103                 Memory(RowAddress) <= WriteData;
104             when Rea_d =>
105                 -- Read Operation
106                 cash(0) := "11111111";
107                 cash(1) := Memory(RowAddress);
108                 cash(2) := (others => '0');
109                 cash(3) := (others => '0');
110                 cash(4) := (others => '0');
111                 cash(5) := CheckSumH(cash);
112                 cash(6) := CheckSumL(cash);
113                 AluReadResp <= cash;
114                 AluReadRespReady <= '1';
115             when others =>
116                 end case;
117         end if;
118     end if;
119 end if;
120 end process;
121
122 end Behavioral;
123
124

```

به عنوان آدرس؛)

○ Write/Read همانند بالا؛ پاسخ در

و AluReadResp

AluReadRespReady

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

ارتباط با سایر ماژول ها: ورودی از CU/ALU، پاسخ خواندن به مسیر خروجی (PackToByte) و یا به ALU برمی گردد.



### ۳-۵. مازول ALU

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

#### ورودی/خروجی (Entity ALU):

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 ( 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;

```

• clk, RST : کلاک و ریست همگام.

• Enable : std\_logic ← فعال سازی 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 / Immediate\_Alu / Operand\_Alu / Array\_Alu / Indirect\_Addresssing

• 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...).

نحوه 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       case PackMode is
110           when Operand_Alalu => -- 3 Clocks
111               if Step = 0 then -- Clock 0 till 1 -> Input
112                   DestinationAddress <= InPack(3);
113                   AddressI := InPack(1);
114                   RamAddress := AddressI;
115                   mode := "00001111";
116                   Step <= 1;
117               elsif Step = 1 then -- Clock 1 till 2
118                   DataI <= ReadResponse(1);
119                   AddressII := InPack(2);
120                   RamAddress := AddressII;
121                   mode := "00001111";
122                   Step <= 2;
123               elsif Step = 2 then -- Clock 2 till 3
124                   RamDataToWrite := Operator(signed(DataI), signed(ReadResponse(1)), Operation);
125                   RamAddress := DestinationAddress;
126                   mode := "11110000";
127                   Step <= 0;
128                   Finish <= '1';
129               end if;
130           end case;

```

## • Operand\_Alalu (۳ کلاک):

(۱) Read از آدرس

InPack(1)

(mode="00001111").

(۲) پس از ReadResponseRdy='1', ReadResponse(1), DataI := ReadResponse(1) سپس Read از

InPack(2)

(۳) نتیجه (Operator(signed(DataI), signed(ReadResponse(1)), Operation) در

آدرس InPack(3) نوشته می‌شود (mode="11110000"), Finish<='1'.

## • Immediate\_Alalu (۲ کلاک):

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

```

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

```

## • Array\_Alalu (طول آرایه ۳ کلاک):

DataII := InPack(2), ArrayLength := to\_integer(unsigned(InPack(3))), مقصد

InPack(4)، اگر Error<='1' => 32 > طول.

حلقه 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) Read و AddAddressII := InPack(2), DataI := ReadResponse(1) 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. مازول Error Detection

هدف: تجميع خطاها:

$Error \leq (RamError \text{ or } AluError) \text{ or } (DecodingError \text{ or } PacketError).$

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

ارتباط با سایر مازولها: 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. مازول PackToByte

هدف: تبدیل پاسخ خواندن RAM به جریان بایت برای سریال سازی.

ورودی ها

• clk

• data\_packet : PackIn ← پکت پاسخ خواندن RAM (۷ بایت: کد پاسخ، داده، خالی،

ChecksumH/L).

خروجی ها

• ByteOut : byte (۸ بیت).

تبدیل پکت ورودی به چهار بایت خروجی ترتیبی

برای سریال سازی

نحوه عملکرد مازول:

داخل مازول یک بافر ۴ خانه ای تشکیل می شود

(PacketCash) و در هر کلاک یکی از بایت ها روی

خروجی قرار می گیرد.

ارتباط با سایر مازول ها: خروجی به ByteToBit می رود.

```
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. مازول 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).

نحوه عملکرد مازول:

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

ارتباط با سایر مازول‌ها: BitOut مستقیماً به  
HammingEncoder می‌رود.

### ۳-۹. ماژول Hamming Encoder

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

ورودی/خروجی (Entity HammingEncoder):

ورودی‌ها

• clk, RST

• std\_logic :BitIn ← بیت‌های سریالی داده (۸ بیت برای هر فریم).

خروجی‌ها

• std\_logic :BitOut ← کد همینگ ۱۳ بیتی به صورت سریال (پس از محاسبه‌ی بیت‌های توازن و کلی).

• std\_logic :OutRdy ← «فریم کدگذاری شده 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;
```

نحوه عملکرد ماژول انکودر همینگ:

- شمارنده‌های InCnt/OutCnt؛ رجیسترهای InCode و byte :Encoded.hamming.
- ورود: ۸ کلاک دریافت بیت‌ها در InCode.
- ساختار کد: داده‌ها در موقعیت‌های ۱، ۲، ۳، ۴، ۵، ۶، ۷، 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).



```

29 architecture Behavioral of HammingEncoder is
30
31     signal InCode : byte := (others => '0');
32     signal Encoded : hamming := (others => '0');
33     signal InCnt : integer := 0;
34     signal OutCnt : integer := 0;
35
36 begin
37
38     process (clk)
39     begin
40         if rising_edge(clk) then
41             if (RST = '1') then
42                 InCode <= (others => '0');
43                 Encoded <= (others => '0');
44                 InCnt <= 0;
45                 OutCnt <= 0;
46                 OutRdy <= '0';
47             else
48                 if InCnt < 8 then -- 8 * Clk -> Input
49                     InCode <= BitIn & InCode(7 downto 1);
50                     InCnt <= InCnt + 1;
51                 elsif InCnt = 8 then -- 1 * Clk -> Calculation
52                     TestBenchInputDisplay <= InCode;
53                     Encoded(2) <= InCode(7);
54                     Encoded(4) <= InCode(6);
55                     Encoded(5) <= InCode(5);
56                     Encoded(6) <= InCode(4);
57                     Encoded(8) <= InCode(3);
58                     Encoded(9) <= InCode(2);
59                     Encoded(10) <= InCode(1);
60                     Encoded(11) <= InCode(0);
61                     -- Assigning parities
62                     Encoded(0) <= (((InCode(7) xor InCode(6)) xor InCode(4)) xor InCode(3)) xor InCode(1);
63                     Encoded(1) <= (((InCode(7) xor InCode(5)) xor InCode(4)) xor InCode(2)) xor InCode(1);
64                     Encoded(3) <= (((InCode(6) xor InCode(5)) xor InCode(4)) xor InCode(0));
65                     Encoded(7) <= (((InCode(3) xor InCode(2)) xor InCode(1)) xor InCode(0));
66                     Encoded(12) <= (((InCode(7) xor InCode(6)) xor InCode(5)) xor InCode(3)) xor InCode(2)) xor InCode(0);
67                     InCnt <= InCnt + 1;

```

• خروج: در حالت Output، BitOut<=Encoded(OutCnt) از ۰ تا ۱۲ و سپس ریست شمارنده‌ها؛

OutRdy اعلام می‌شود.

```

68         else
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;

```

زمان/کلاک: ۱۴ کلاک طول می‌کشد تا دریافت

کند و خروجی بدهد.

خروجی نهایی سریالی سیستم از این ماژول

دریافت می‌شود.

### 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):

```
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;
```

• Input : in std\_logic ← بیت سریالی ورودی (کد

همینگ ۱۳ بیتی به‌ازای هر بایت داده).

• InputBegining : in std\_logic ← نشانگر «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 ← آینه‌نمایش در

تست‌بنچ (OutRdy از انکودر؛ در عمل «رفتار خروجی» دارد (برای اعلان آماده بودن فریم گذشته).

• 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 (آینه CUPacketRdy)

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

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

### جریان داده و کنترل

۱. Input ← Decoder: با

InputBeginning='1'، ۱۳ بیت از Input

جمع می‌شود؛ سپس DataByte آماده و

DecDataIsRdy='1' اگر تصحیح ممکن/معتبر

باشد، DecValidation='1'.

۲. Control Unit ← Decoder: CU با هر

DecDataIsRdy یک بایت می‌گیرد، طبق

Function بایت اول، بایت‌های بعدی را تا تکمیل

Packet می‌خواند، '1' CUPacketRdy و

PacketValidation را تنظیم می‌کند؛

PackType نیز تعیین می‌شود.

۳. تغذیه هم‌زمان RAM و ALU: با

'1' CUPacketRdy هر دو ماژول اتفاقاً یک

پکت می‌بینند:

```

63 begin
64
65     Decoder: entity work.HammingDecoder
66     port map
67     (
68         CodeBegins => InputBeginning,
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     (
94         CtrlInputRdy => CUPacketRdy,
95         CtrlReq => CUPacket,
96         CtrlReadResp => RamReadResp,
97         CtrlReadRespReady => RamReadRespRdy,
98         AluInputRdy => SendToRamReady,
99         AluReq => AluToRam,
100        AluReadResp => RamToAlu,
101        AluReadRespReady => RespForAluRdy,
102        Error => RamError,
103        RST => RST,
104        clk => clk
105    );

```

- RAM (شاخه 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 (شاخه Ctrl) به

PackToByte → ByteToBit →  
HammingEncoder می‌رود؛ انکودر ۱۳ بیت کد را روی  
Output می‌فرستد و OutputRdy را اعلام می‌کند.

۵. تجمیع خطا: اگر هر کدام 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 و ALU، قالب‌بندی Read Response، و بازکدگذاری همینگ بررسی می‌شود. پورت‌های کمکی تاپ‌ماژول نیز برای مشاهده‌ی پکت‌های داخلی و تعامل‌های RAM ↔ ALU 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 ns دوره؛ ریست

- هم‌زمان (سنکرون) در ابتدای شبیه‌سازی به‌مدت ۱۰۰ ns فعال 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 ۷ بایتی ساخته‌شده توسط ControlUnit و سیگنال آماده بودن آن.

- CURRCheck / CURRCheckRdy: پاسخ خواندن RAM در قالب ۷ بایتی 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 دادن ورودی

برای هر سناریو، آرایه‌ی HammingPacket مقاداردهی 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 RAM یا 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 -- Stimulus process
115 stim_proc: process
116     variable HammingPacket : hamming_packet;
117 begin
118     -- hold reset state for 100 ns.
119     RST <= '1';
120     wait for clk_period*10;
121     RST <= '0';
122     wait for 10 ns;
123     -- 1) Write: Uses 5 cells. Writing "00000001" in "00001100".
124     HammingPacket := ("11111110000001",
125         "11000000011000",
126         "0001000100011",
127         "1110111011110",
128         "0010111111010",
129         "00000000000000",
130         "00000000000000");
131
132     for i in 0 to 4 loop
133         InputBegining <= '1';
134         wait for 10 ns;
135         InputBegining <= '0';
136         for j in 0 to 12 loop
137             Input <= HammingPacket(i)(j);
138             wait for 10 ns;
139         end loop;
140         wait for clk_period*2;
141     end loop;
142
143     -- 2) Write: Uses 5 cells. Writing "00000100" in "00000011".
144     HammingPacket := ("11111110000001",
145         "1101000000111",
146         "0100000101001",
147         "1110111011110",
148         "0110111101111",
149         "00000000000000",
150         "00000000000000");
151
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;

```

```

163 -- 3) Alu: Operand Alu, Uses 6 cells. Writing {"00001100"* + "00000011"* in "00000001".
164 -- "00001100"* = "00000001", "00000011"* = "00000100" => "00000001"* = "00000101".
165 HammingPacket := ("00000000000000",
166                   "11000000110000",
167                   "11010000001111",
168                   "00010001000111",
169                   "00000000000000",
170                   "11010010000000",
171                   "00000000000000");
172 for i in 0 to 5 loop
173   InputBeginning <= '1';
174   wait for 10 ns;
175   InputBeginning <= '0';
176   for j in 0 to 12 loop
177     Input <= HammingPacket(i)(j);
178     wait for 10 ns;
179   end loop;
180   wait for clk_period*2;
181 end loop;
182
183 -- 4) Read: Uses 4 cells. Reading "00000101" from "00000001".
184 HammingPacket := ("00010000111111",
185                   "00010001000111",
186                   "00000000000000",
187                   "11010010000000",
188                   "00000000000000",
189                   "00000000000000",
190                   "00000000000000");
191 for i in 0 to 3 loop
192   InputBeginning <= '1';
193   wait for 10 ns;
194   InputBeginning <= '0';
195   for j in 0 to 12 loop
196     Input <= HammingPacket(i)(j);
197     wait for 10 ns;
198   end loop;
199   wait for clk_period*2;
200 end loop;
201
202 -- 5) Alu: Immediate Alu, Uses 6 cells. Writing {"00000000"* + "01000000" in "00011111".
203 -- "0000000000"* = "0000000000", => "00011111"* = "01000000".
204 HammingPacket := ("01000011011001",
205                   "00000000000000",
206                   "10011000000001",
207                   "11000001011111",
208                   "00000000000000",
209                   "01100011101111",
210                   "00000000000000");
211 for i in 0 to 5 loop
212   InputBeginning <= '1';
213   wait for 10 ns;
214   InputBeginning <= '0';
215   for j in 0 to 12 loop
216     Input <= HammingPacket(i)(j);
217     wait for 10 ns;
218   end loop;
219   wait for clk_period*2;
220 end loop;
221
222 -- 6) Read: Uses 4 cells. Reading "01000000" from "00011111".
223 HammingPacket := ("00010000111111",
224                   "11000001011111",
225                   "00000000000000",
226                   "10001000111110",
227                   "00000000000000",
228                   "00000000000000",
229                   "00000000000000");
230 for i in 0 to 3 loop
231   InputBeginning <= '1';
232   wait for 10 ns;
233   InputBeginning <= '0';
234   for j in 0 to 12 loop
235     Input <= HammingPacket(i)(j);
236     wait for 10 ns;
237   end loop;
238   wait for clk_period*2;
239 end loop;

```

```

241 -- 7) Alu: Array Alu, Uses 7 cells. Writing {"00000000" + i* + "00000001" in "00011111 + i*".
242 -- "00000000"* = "00000000",
243 -- "00000001"* = "00000101",
244 -- "00000010"* = "00000000",
245 -- =>
246 -- "00011111"* = "00000001",
247 -- "00000000"* = "00000110",
248 -- "00000001"* = "00000001".
249 HammingPacket := ("01111000000000",
250 "00000000000000",
251 "0001000100011",
252 "11010000000111",
253 "1100001011111",
254 "1110111011110",
255 "1111110000110");
256 for i in 0 to 6 loop
257   InputBeginning <= '1';
258   wait for 10 ns;
259   InputBeginning <= '0';
260   for j in 0 to 12 loop
261     Input <= HammingPacket(i)(j);
262     wait for 10 ns;
263   end loop;
264   wait for clk_period*2;
265 end loop;
266 wait for 30 ns;
267
268 -- 8) Read: Uses 4 cells. Reading "00000001" from "00011111".
269 HammingPacket := ("0001000011111",
270 "1100001011111",
271 "0000000000000",
272 "1000100011110",
273 "0000000000000",
274 "0000000000000",
275 "0000000000000");
276 for i in 0 to 3 loop
277   InputBeginning <= '1';
278   wait for 10 ns;
279   InputBeginning <= '0';
280   for j in 0 to 12 loop
281     Input <= HammingPacket(i)(j);
282     wait for 10 ns;
283   end loop;
284   wait for clk_period*2;
285 end loop;
286 wait for clk_period*2;
287
288 -- 8.33) Read: Uses 4 cells. Reading "00000110" from "00000000".
289 HammingPacket := ("0001000011111",
290 "0000000000000",
291 "0000000000000",
292 "0001000011111",
293 "0000000000000",
294 "0000000000000",
295 "0000000000000");
296 for i in 0 to 3 loop
297   InputBeginning <= '1';
298   wait for 10 ns;
299   InputBeginning <= '0';
300   for j in 0 to 12 loop
301     Input <= HammingPacket(i)(j);
302     wait for 10 ns;
303   end loop;
304   wait for clk_period*2;
305 end loop;
306 wait for clk_period*2;
307
308 -- 8.66) Read: Uses 4 cells. Reading "00000001" from "00000001".
309 HammingPacket := ("0001000011111",
310 "0001000100011",
311 "0000000000000",
312 "1101001000000",
313 "0000000000000",
314 "0000000000000",
315 "0000000000000");
316 for i in 0 to 3 loop
317   InputBeginning <= '1';
318   wait for 10 ns;
319   InputBeginning <= '0';
320   for j in 0 to 12 loop
321     Input <= HammingPacket(i)(j);
322     wait for 10 ns;
323   end loop;
324   wait for clk_period*2;
325 end loop;
326 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                  "00000000000000",
334                  "1100001011111",
335                  "1000000110001",
336                  "00000000000000",
337                  "1101101101111",
338                  "00000000000000");
339 for i in 0 to 5 loop
340     InputBeginning <= '1';
341     wait for 10 ns;
342     InputBeginning <= '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                  "00000000000000",
354                  "0100001101110",
355                  "00000000000000",
356                  "00000000000000",
357                  "00000000000000");
358 for i in 0 to 3 loop
359     InputBeginning <= '1';
360     wait for 10 ns;
361     InputBeginning <= '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 نتایج مورد انتظار

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

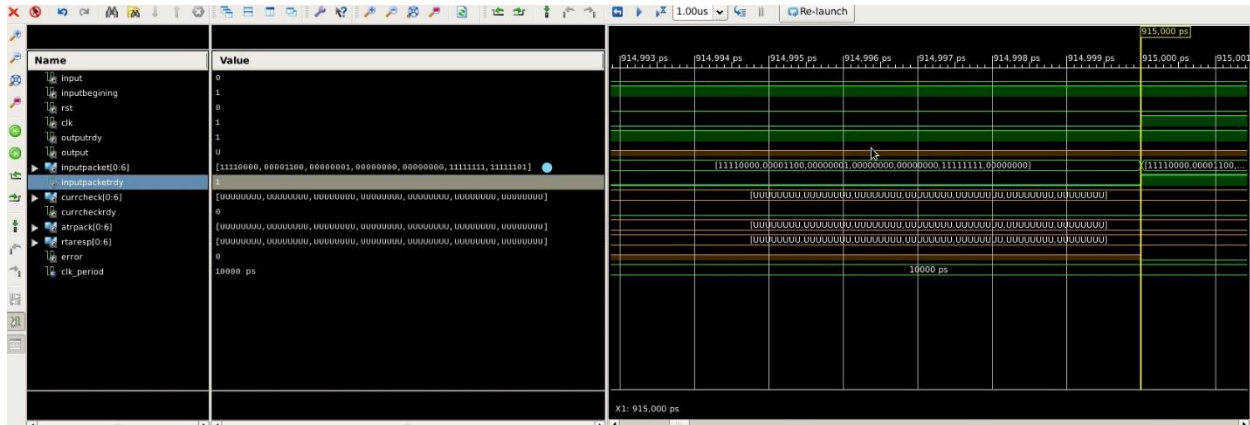
00001100.

انتظار: تشکیل پکت Write، انجام نوشتن در RAM، بدون پاسخ

خواندن (عدم فعال شدن CURRCheckRdy، Error='0'.

```
-- 1) Write: Uses 5 cells. Writing "00000001" in "00001100".
HammingPacket := ("11111110000001",
  "1100000011000",
  "0001000100011",
  "1110111011110",
  "001011111010",
  "0000000000000",
  "0000000000000");

for i in 0 to 4 loop
  InputBeginning <= '1';
  wait for 10 ns;
  InputBeginning <= '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;
```



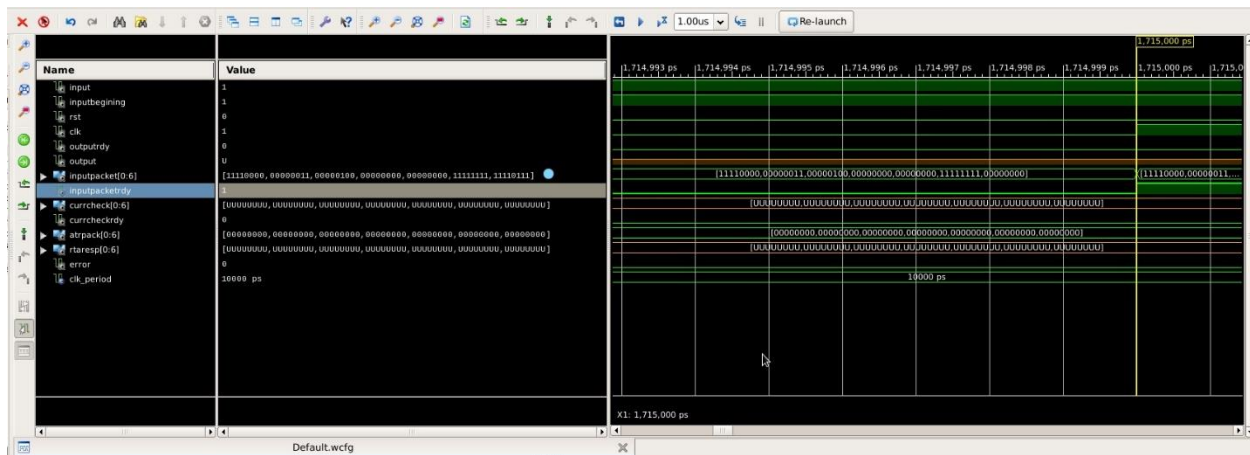
۲. Write (۵ سلول) ← نوشتن 00000100 در

آدرس 00000011.

انتظار: مشابه سناریوی ۱؛ Error='0'.

```
-- 2) Write: Uses 5 cells. Writing "00000100" in "00000011".
HammingPacket := ("11111110000001",
  "11010000000111",
  "0100000101001",
  "1110111011110",
  "0110111101111",
  "0000000000000",
  "0000000000000");

for i in 0 to 4 loop
  InputBeginning <= '1';
  wait for 10 ns;
  InputBeginning <= '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;
```



### ۳. Operand\_ALU (۶ سلول) ← جمع محتویات آدرس‌های 00001100 و 00000011 و نوشتن

نتیجه در 00000001.

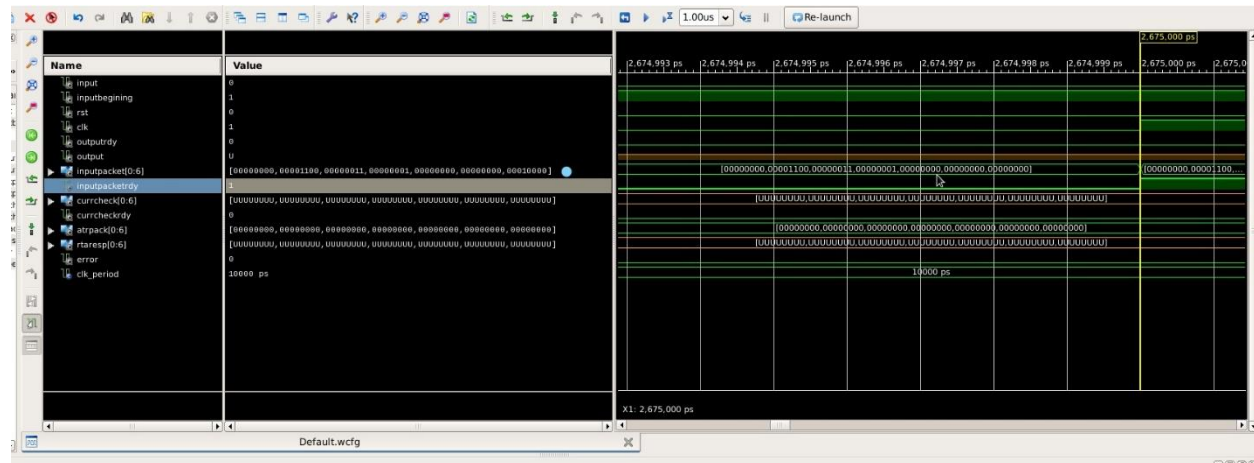
انتظار: توالی Read→Read→Write بین ALU و

RAM (در ATRPack/RTAResp قابل رؤیت)،

نتیجه مقصد باید 00000101 شود.

```
-- 3) Alu: Operand Alu, Uses 6 cells. Writing ("00001100" + "00000011" in "00000001".
-- "00001100" = "00000001", "00000011" = "00000100" => "00000001" + "00000101".
HammingPacket := ("00000000000000",
                  "1100000011000",
                  "1101000000111",
                  "0001000100011",
                  "00000000000000",
                  "1101001000000",
                  "0000000000000");

for i in 0 to 5 loop
  InputBeginning <= '1';
  wait for 10 ns;
  InputBeginning <= '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;
```



### ۴. Read (۴ سلول) ← خواندن از 00000001.

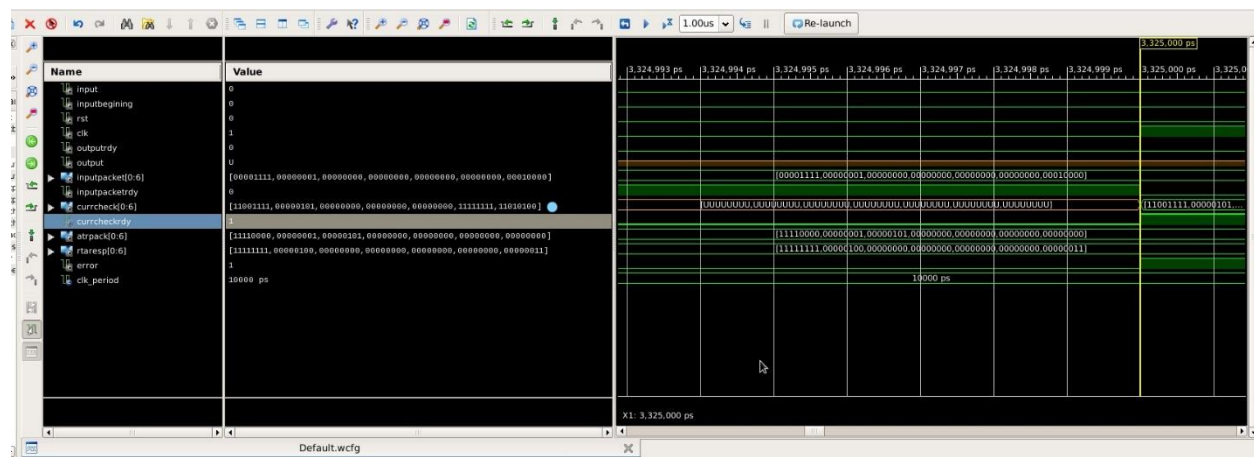
انتظار: تولید پاسخ خواندن با شناسه

"11001111" و داده 00000101، چک‌سام

معتبر، ارسال سریال روی خروجی.

```
-- 4) Read: Uses 4 cells. Reading "00000101" from "00000001".
HammingPacket := ("0001000011111",
                  "0001000100011",
                  "00000000000000",
                  "1101001000000",
                  "00000000000000",
                  "00000000000000",
                  "00000000000000");

for i in 0 to 3 loop
  InputBeginning <= '1';
  wait for 10 ns;
  InputBeginning <= '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;
```



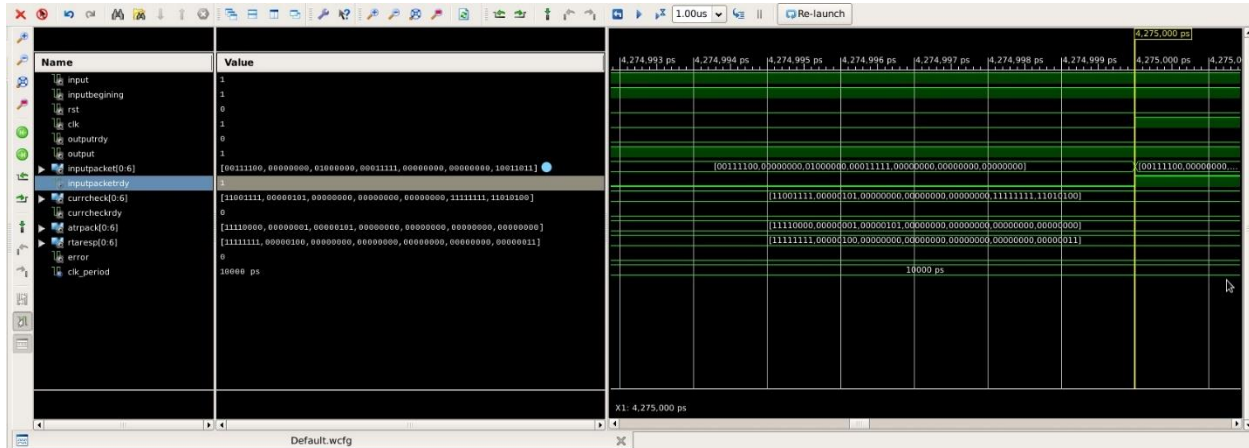
## ۵. Immediate\_ALU (۶ سلول) ← ترکیب یک خانه RAM با داده فوری و نوشتن در 00011111.

انتظار: الگوی Read→Write؛ مقدار آدرس

مقصد به 01000000 تغییر می‌کند.

```
-- 5) Alu: Immediate Alu, Uses 6 cells. Writing "00000000" + "01000000" in "00011111".
-- "0000000000" = "0000000000", => "00011111" = "01000000".
HammingPacket := ("0100011011001",
"00000000000000",
"10011000000001",
"1100001011111",
"00000000000000",
"0110001110111",
"00000000000000");

for i in 0 to 5 loop
  InputBeginning <= '1';
  wait for 10 ns;
  InputBeginning <= '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;
```



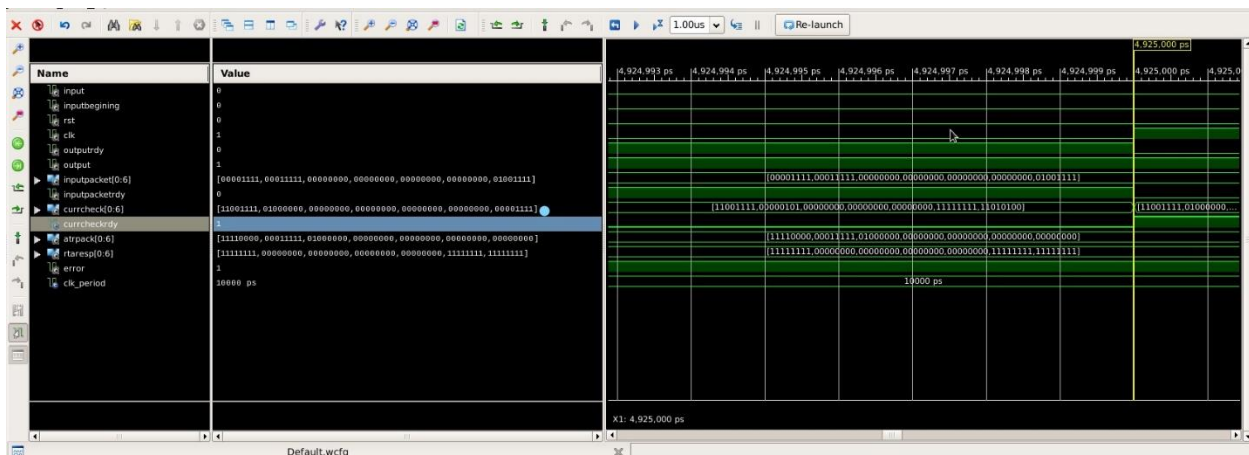
## ۶. Read (۴ سلول) ← خواندن از 00011111.

انتظار: پاسخ خواندن با داده 01000000 و

چک‌سام صحیح، سپس ارسال به شکل سریال.

```
-- 6) Read: Uses 4 cells. Reading "01000000" from "00011111".
HammingPacket := ("0001000011111",
"1100010111111",
"00000000000000",
"1000100011110",
"00000000000000",
"00000000000000",
"00000000000000");

for i in 0 to 3 loop
  InputBeginning <= '1';
  wait for 10 ns;
  InputBeginning <= '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;
```



## ۷. Array\_ALU (۷ سلول) ← اجرای عمل آرایه‌ای با طول معتبر و نوشتن پی‌درپی نتایج.

انتظار: مجموعه‌ای از Read/Write متوالی متناسب با

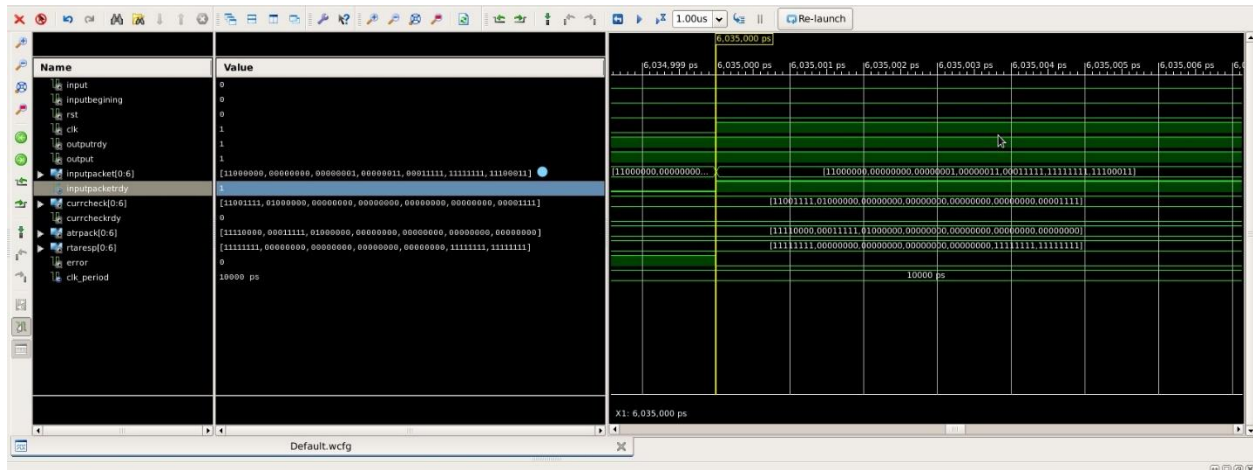
طول؛ پس از اتمام، وضعیت حافظه مطابق توضیح سناریو

به این صورت باشد:

00000000 ← 00011111 ← 00000001

00000001 ← 00000001 00000110

```
-- 7) Alu: Array Alu, Uses 7 cells. Writing ("00000000" + i* + "00000001" in "00011111 + i*").
-- "00000000" = "00000000",
-- "00000001" = "00000001",
-- "00000010" = "00000000",
-- =>
-- "00011111" = "00000001",
-- "00000000" = "00000000",
-- "00000001" = "00000001",
HammingPacket := ("011100000000",
                  "000000000000",
                  "000100010011",
                  "110100000011",
                  "100000001111",
                  "111011011110",
                  "111110000110");
for i in 0 to 6 loop
  InputBeginning <= '1';
  wait for 10 ns;
  InputBeginning <= '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;
```

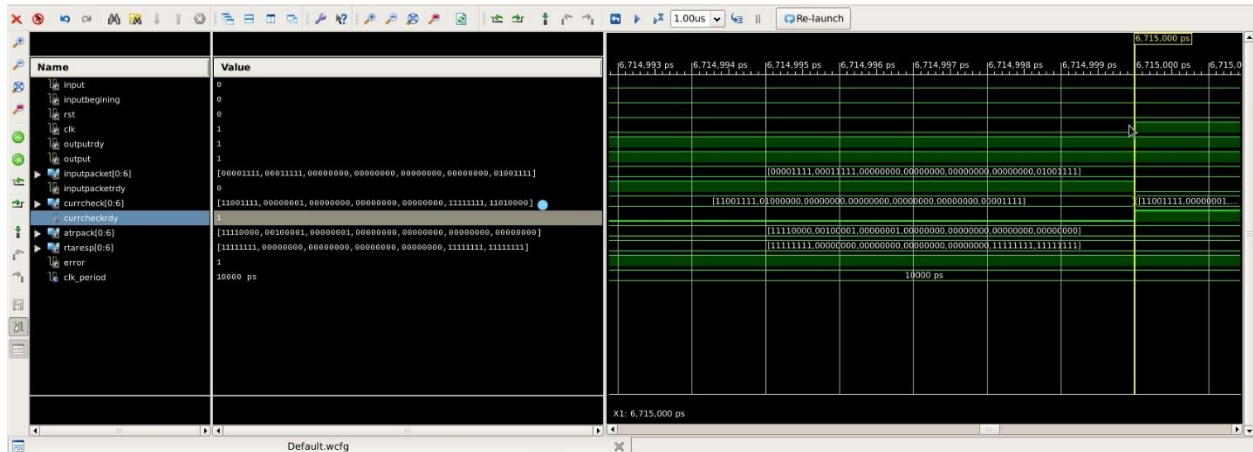


## ۸. Read (۴ سلول) ← خواندن از 00011111.

انتظار: داده‌ی خوانده‌شده 00000001 (اثر سناریوی

آرایه‌ای).

```
-- 8) Read: Uses 4 cells. Reading "00000001" from "00011111".
HammingPacket := ("000100001111",
                  "110000101111",
                  "00000000000000",
                  "100010001110",
                  "00000000000000",
                  "00000000000000",
                  "00000000000000");
for i in 0 to 3 loop
  InputBeginning <= '1';
  wait for 10 ns;
  InputBeginning <= '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;
```

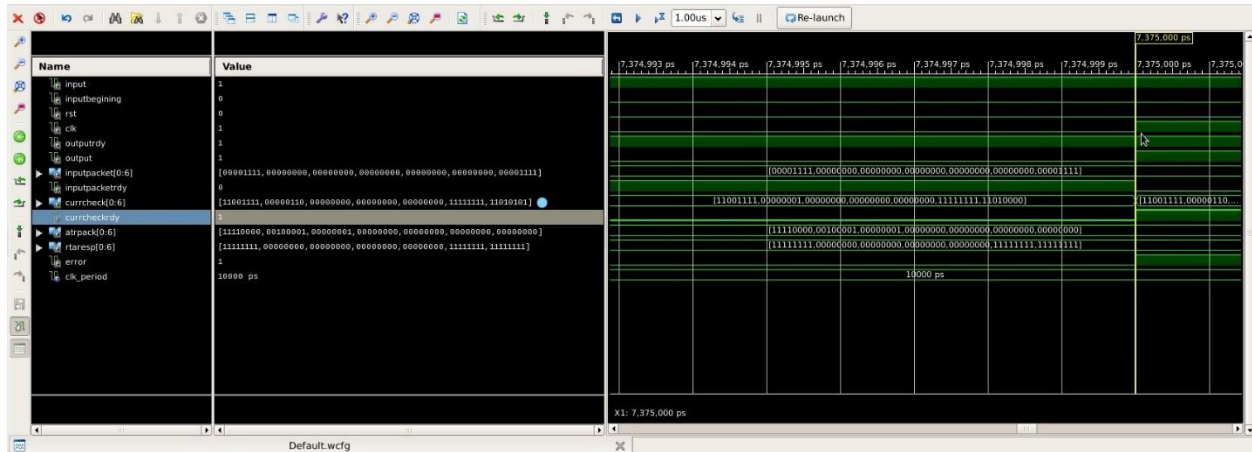


```
-- 8.33) Read: Uses 4 cells. Reading "00000110" from "00000000".
HammingPacket := ("0001000011111",
"0000000000000000",
"0000000000000000",
"0001000011111",
"0000000000000000",
"0000000000000000",
"0000000000000000");

for i in 0 to 3 loop
  InputBeginning <= '1';
  wait for 10 ns;
  InputBeginning <= '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;
```

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

انتظار: داده‌ی خوانده‌شده 00000110.



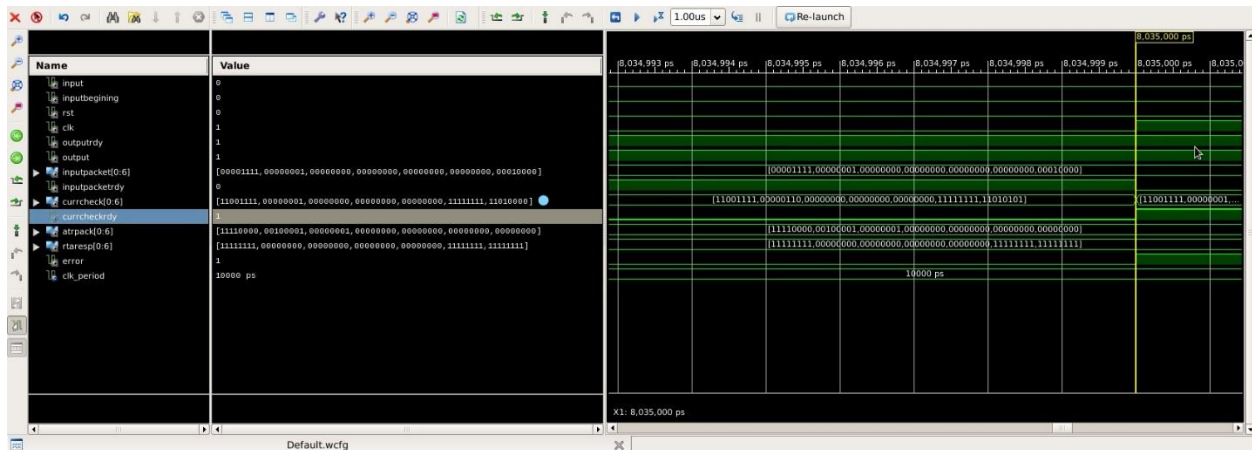
```
-- 8.66) Read: Uses 4 cells. Reading "00000001" from "00000001".
HammingPacket := ("0001000011111",
"0001000100011",
"0000000000000000",
"11010010000000",
"0000000000000000",
"0000000000000000",
"0000000000000000");

for i in 0 to 3 loop
  InputBeginning <= '1';
  wait for 10 ns;
  InputBeginning <= '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;
```

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

انتظار: داده‌ی خوانده‌شده 00000001 (تأیید وضعیت

پس از سناریوهای قبل).





## ۹. Indirect\_Address (۶ سلول) ←

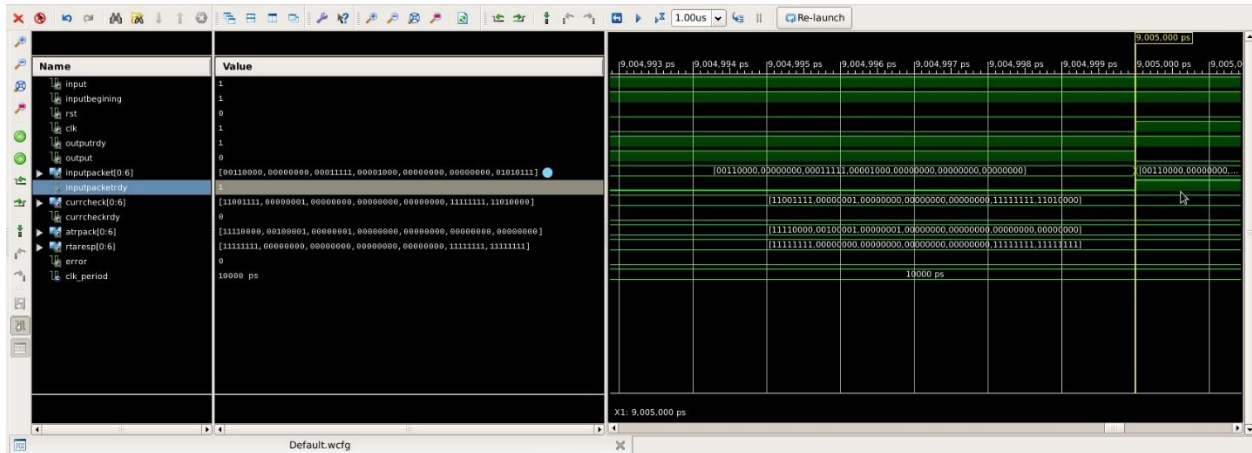
آدرس‌دهی غیرمستقیم: خواندن آدرس

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.

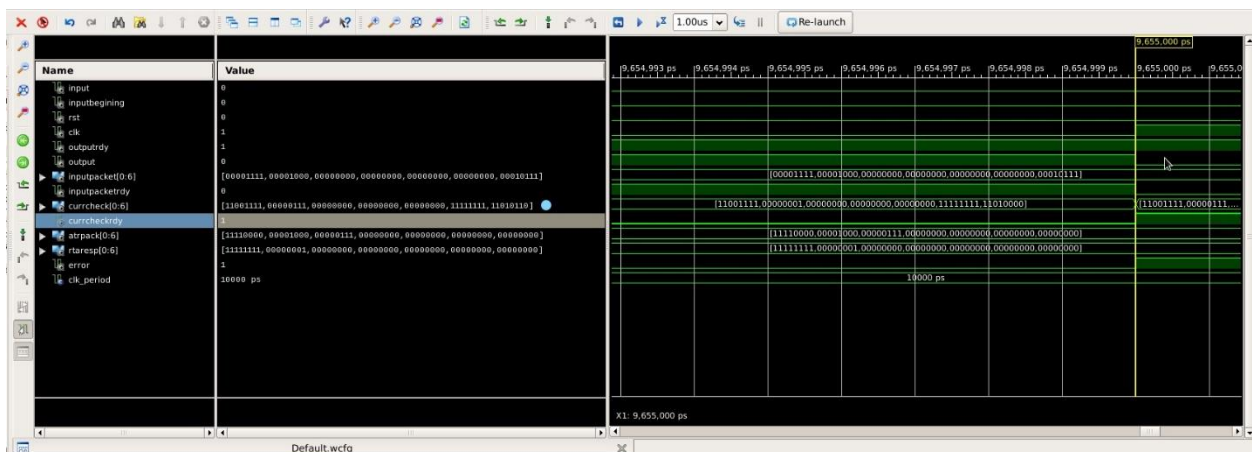
انتظار: داده‌ی خوانده‌شده 00000111 و پاسخ

استاندارد با چک‌سام صحیح؛ ارسال سریال روی خروجی.

```
-- 10) Read: Uses 4 cells. Reading "00000111" from "00001000".
HammingPacket := ("0001000011111",
  "1000000110001",
  "0000000000000",
  "0100001101110",
  "0000000000000",
  "0000000000000",
  "0000000000000");

for i in 0 to 3 loop
  InputBeginning <= '1';
  wait for 10 ns;
  InputBeginning <= '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;
```



### 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 ۷ بیتی شامل شناسه "11001111"، داده انتظار و چک‌سام صحیح باشد و سپس به صورت سریالی کد ارسال گردد. در سناریوهای ALU، توالی دسترسی‌ها و داده مقصد در خواندن‌های بعدی تأیید شود. در کل سناریوهای معتبر، پرچم Error غیرفعال باقی بماند.