گزارش نهایی

درس معماری کامپیوتر – پروژه پایانی سال تحصیلی ۰۳-۱۴۰۲

پردازنده تک‌دوره‌ای(RISC-V (RV32I – subset

دانشجو: آرش گنجه ای

شماره دانشجویی: 40123070

استاد درس: دکتر احمد شعبانی دستیار آموزشی: مهندس هادی صلواتی

فهرست مطالب

۱ .مقدمه

هدف این پروژه، طراحی و شبیه‌سازی یک پردازنده RISC-V RV32I تک‌دوره‌ای با مجموعه دستورهای محدود زیر است:

```
ADD, SUB, AND, OR, ADDI, ANDI, ORI, LW, SW
```

تمام کدها در VHDL-2008 نوشته و در محیط Vivado 2018.3 (64-bit, WebPACK) تست شده‌اند.

شبیه‌سازی رفتاری با XSim و گام زمانی پیش‌فرض 1 ps انجام شده است؛ زمان اجرای خودکار روی 100 ns تنظیم شد تا خروجی موج مختصر و قابل چاپ گردد.

چون آدرس‌دهی به‌صورت کلمه‌ای است، هر افزایش +1 در PC معادل جابه‌جایی ۴ بایت می‌شود PC ← PC+4 در پیاده‌سازی واقعی.

۲ .پیاده‌سازی بخش اوّل – ماژول‌های جداگانه

۲-۱ واحد محاسبات و منطق ALU

- این واحد چهار عملگر اصلی را پیاده می‌کند: ADD برای محاسبهٔ آدرس LW/SW، SUB جهت اختلاف در R-type، و عملگرهای منطقی AND/OR برای آزمایش بیت به بیت.
- ورودی‌ها ۳۲بیتی هستند و کد عملیات در سیگنال سه بیتی ALUSel دریافت می‌شود.
- به‌جای ماژول‌های ترکیب‌کننده مستقل، از یک عبارت case درون یک پروسهٔ کاملاً ترکیبی استفاده شد تا مسیر بحرانی کوتاه شود؛ نتیجه بلافاصله پس از پایدار شدن ورودی‌ها تولید می‌شود.
- در آزمون واحد، برای هر کُد، ۱۰ جفت عدد مثبت/منفی به‌صورت تصادفی تزریق و خروجی با فایل طلایی مقایسه شد.

عملکرد پشتیبانی شده

ALUSel	عملیات	توضیح
"000"	ADD	جمع با علامت (مهم برای LW/SW)
"001"	SUB	اختلاف؛ با استفاده از عملگر - در VHDL
"010"	AND	ضرب منطقی؛ پیاده‌سازی با and برداری
"011"	OR	جمع منطقی؛ با or برداری

پیاده‌سازی

```
Result <= std_logic_vector(
            case ALUSel is
              when "000" => signed(A) + signed(B);
              when "001" => signed(A) - signed(B);
              when "010" => signed(A) and signed(B);
```

```
  when "011" => signed(A) or  signed(B);
  when others => (others=>'0');
end case );
```

- از نوع signed برای جلوگیری از هشدار overflow استفاده شد.
- هیچ رجیستری درون ALU قرار نگرفت تا مسیر بحرانی کوتاه بماند؛ در معماری تک‌دوره‌ای این مهم‌ترین عامل تعیین‌کننده فرکانس است.

Testbench

در آزمون واحد، برای هر کد عملیاتی ۱۰ جفت عدد تصادفی (مثبت و منفی) تولید شد و خروجی با فایل مرجع مقایسه شد؛

Register File ۲-۲

بانک ثبات‌ها شامل ۳۲ رجیستر ۳۲بیتی است. دو پورت خواندن (A, B) به‌صورت Combinational پیاده شد تا در همان سیکل کلاک داده برای ALU حاضر باشد؛ پورت نوشتن روی لبه بالارونده و فقط در صورت RegWEn=1 عمل می‌کند.

رجیستر x0 با شرط if D_addr /= 00000 هرگز تغییر نمی‌کند و همواره صفر باقی می‌ماند. در سنتز، Vivado این ماژول را به RAM دو پورته نگاشت کرده است.

```
type reg_array is array (0 to 31) of std_logic_vector(31 downto 0);
signal regs : reg_array := (others => (others=>'0'));
```

- خواندن ترکیبی است تا مقادیر در همان چرخه در اختیار ALU قرار گیرد.
- نوشتن روی لبه بالارونده کلاک و فقط اگر RegWEn='1'.

- رجیستر x0 همیشه صفر: در فرآیند نوشتن شرط if D_addr /= “00000” then ... end if; اعمال شد.
- زیربلوک دوپورت به‌کمک توصیف رفتاری انجام شد؛ در سنتز به RAM با دو پورت خواندن و یک پورت نوشتن نگاشت می‌شود.

۲-۳ Instruction Memory (IMEM)

حافظه دستور از نوع ROM 64 × 32 است؛ آدرس ورودی ۶ بیتی و بخش خواندن کاملاً ترکیبی است. برای تغییر برنامه فقط مقادیر ثابت آرایه ROM عوض می‌شود؛ نیازی به بازآرایی کد نیست.

دستورهای اسمبلی تست طبق ساختار RISC-V استاندارد به کمک ابزار آنلاین Ripes یا Venus تبدیل به کد ماشین شده و در حافظه دستوری ذخیره شدند. سپس در سیمولیشن پردازنده اجرا شده و نتیجه مقایسه شد با مقادیر مورد انتظار.

- ROM اندازه 64 کلمه × ۳۲ بیت.
- آدرس ورودی ۶ بیتی (address) که از ProgramCounter می‌آید؛ بدون سیگنال Chip-Enable.
- به دلیل تک‌دوره‌ای بودن، خواندن باید ترکیبی باشد؛ لذا فقط یک عبارت انتساب مستقیم استفاده کردیم:

```
instruction <= ROM( to_integer(unsigned(address)) );
```

مقداردهی اولیه برنامه در خود کد) VHDL فهرست .(“...”x برای تغییر برنامه تنها کافی است مقادیر راهنما که در بخش «شبیه‌سازی» آمده‌اند جایگزین شود.

۲-۴ Immediate Generator

این واحد دو حالت دارد: I-type و S-type. به کمک سیگنال ImmSel که از ControlUnit می‌رسد، تشخیص می‌دهد باید فیلدهای ۳۱:۲۰ یا ترکیب ۱۱:۷/۳۱:۲۵ را Sign-Extend کند. خروجی ۳۲بیتی بلافاصله در همان چرخه به ورودی مالتی‌پلکسر B می‌رود.

- ورودی ۱۲ بیتی = (Imm_in) بیت‌های ۳۱:۲۰ دستور.
- برای نوع I (addi/andi/ori/lw) صرفاً Sign-Extend می‌شود.
- برای نوع S (sw) بیت‌های ۳۱:۲۵ | ۱۱:۷ باید چسبانده شوند؛ یک ورودی کنترل ImmSel از CU اضافه گردید.

```
case ImmSel is
  when '0' => Imm_out <= (20 downto 0 => Imm_in(11)) & Imm_in;
  when '1' => Imm_out <= (20 downto 0 => Imm_inS(11)) & Imm_inS(11
downto 5) & Imm_inS(4 downto 0);
end case;
```

۲-۵ Program Counter (PC)

شمارندهٔ ۶ بیتی pc در هر لبه بالارونده یک واحد افزایش می‌یابد؛ در حالت Reset مقدار آن صفر می‌شود. به علت آدرس‌دهی کلمه‌ای (هر کلمه ۴ بایت)، افزایش «+1» معادل همان pc+4 در معماری واقعی است.

- رجیستر ۶ بیتی افزایشی.
- در Reset به صفر بازمی‌گردد.
- به جای pc+4 (مثل پردازنده واقعی) اینجا به دلیل آدرس‌دهی کلمه‌ای فقط pc <= pc + 1 کافی است؛ چون هر عنصر ROM یک کلمه است.

۲-۶ Control Unit

واحد کنترل با استناد به فیلدهای opcode, funct3, funct7 پنج سیگنال اصلی ALUSel, MemRW, MemtoReg را تولید می‌کند.

جدول حالاتی شامل ردیف‌های R-type، I-type (Immediate)، LW و SW است. به‌صورت پیش‌فرض همه سیگنال‌ها صفر شده‌اند تا در صورت دستور ناشناخته پردازنده وارد حالت امن شود.

تابع تطبیق فیلدهای opcode / funct3 / funct7 برای تولید پنج سیگنال اصلی:

سیگنال	نقش
RegWEn	فعال‌سازی نوشتن در بانک ثبات
BSel	انتخاب Immediate یا رجیستر دوم برای ورودی B ALU
ALUSel	تعیین عملگر ALU
MemRW	1: نوشتن در DMEM، 0: خواندن
MemtoReg	1: داده خوانده‌شده از DMEM → بانک ثبات 0: نتیجه ALU

جداول کدنویسی دقیقاً مطابق جدول PDF نوشته شده‌اند؛ برای آینده می‌توان این منطق را با رویکرد «دیکدر ریزعملیات» جایگزین کرد.

۲-۷ Data Memory (DMEM)

حافظهٔ داده یک RAM ترکیبی/سنکرون ۶۴×۳۲ بیتی است؛ در لبه بالارونده و با MemRW=1 مقدار ورودی write_data را در آدرس address(7 downto 2) ذخیره می‌کند.

خواندن کاملاً ترکیبی است تا داده در همان چرخه برای مسیر MemtoReg آماده باشد. آدرس‌دهی بایتی باعث شد دو بیت پایینی آدرس برای هم‌ترازی صفر باشد.

- RAMاندازه 64×32 بیتی؛ آدرس دهی Byte-Addressed ؛ لذا address(7 downto 2) رأس سلول را انتخاب می‌کند.
- نوشتن روی لبه بالارونده اگر MemRW='1'.
- خواندن کاملاً ترکیبی برای رعایت شرط تک‌دوره‌ای.
- در تست #4 مشاهده شد که مقدار 0x2A در چرخه SW نوشته و در چرخه LW مجدداً خوانده شد.

۲-۸. واحد بالادستی CPU

این فایل تمام ماژول‌های قبلی را در یک مسیر داده تک‌دوره‌ای ترکیب می‌کند

- مرحله Fetch: خروجی ProgramCounter مستقیماً به InstructionMemory می‌رود؛ خواندن ROM ترکیبی است.
- مرحله Decode: فیلدهای دستور توسط اتصال‌های برداری استخراج می‌شوند و به ControlUnit، RegisterFile و ImmediateGenerator داده می‌شوند.
- مسیر ALU: خروجی رجیستر A همیشه به ورودی A ALU می‌رود؛ ورودی B با مالتی‌پلکسر BSel بین داده رجیستر B و Immediate گسترش‌یافته انتخاب می‌شود.
- حافظه داده: برای LW و SW، نتیجه ALU به‌عنوان آدرس در DMEM استفاده می‌شود. نوشتن فقط در حضور MemRW=1 انجام می‌شود.
- بازگشت نتیجه: مالتی‌پلکسر MemtoReg انتخاب می‌کند که خروجی ALU (برای همه دستورها به‌جز LW) یا خروجی DMEM (فقط برای LW) در رجیستر مقصد نوشته شود.

- سیگنال RegWEn تضمین می‌کند ثبات مقصد فقط در دستورهای نوشتنی تغییر کند.
- مرحله Fetch : سیگنال pc_out در ابتدای چرخه جاری (t=0+) به ورودی آدرس IMEM می‌رود و همان لحظه بیت‌های دستور به گذرگاه instruction منتقل می‌شود؛ بنابراین عمل خواندن ROM هیچ تأخیری به چرخه اضافه نمی‌کند.

این طراحی باعث می‌شود هر پنج فاز Fetch, Decode, Execute, Memory, Write-Back در یک لبه کلاک کامل شوند؛ اگرچه زمان بحرانی به طولانی‌ترین مسیر (ALU+DMEM) محدود می‌شود.

۹-۲ ترکیب نهایی پردازنده (CPU.vhdl)

در این بخش، فایل CPU.vhdl نقش اصلی در ترکیب و اتصال تمام ماژول‌های طراحی‌شده را ایفا می‌کند و در واقع، ساختار کلی پردازنده تک‌چرخه‌ای را ایجاد می‌نماید. در ادامه، عملکرد آن به صورت مرحله‌به‌مرحله توضیح داده شده است:

واحد شمارنده برنامه(Program Counter)

در ابتدای فایل، واحد PC تعریف شده است. این ماژول با دریافت سیگنال ساعت (clk) و سیگنال ریست (reset) مقدار pc_out را به عنوان آدرس دستور بعدی تولید می‌کند. این مقدار با std_logic_vector(5 downto 0)محدود شده تا بتواند حداکثر ۶۴ دستور را پشتیبانی کند.

```
PC: entity work.ProgramCounter port map(clk, reset, pc_out);
```

حافظه دستور(Instruction Memory)

خروجی pc_out به عنوان ورودی حافظه دستور استفاده می‌شود. خروجی این حافظه که شامل کد باینری دستور است، در سیگنال instruction ذخیره می‌شود و به بخش‌های مختلف برای استخراج فیلدهای دستور منتقل می‌گردد.

```
IMEM: entity work.InstructionMemory port map(address => pc_out, instruction => instruction);
```

استخراج فیلدهای دستور

کد دستور دریافتی از حافظه به صورت زیر به فیلدهای مختلف تجزیه می‌شود:

- opcode بیت‌های ۶ تا ۰
- rd, rs1, rs2 ثبات‌های مقصد و منبع
- funct3, funct7 مشخص‌کننده عملیات
- imm عدد ثابت (immediate) برای دستورهای I-type, S-type و...

```
opcode <= instruction(6 downto 0);
rd     <= instruction(11 downto 7);
funct3 <= instruction(14 downto 12);
rs1    <= instruction(19 downto 15);
rs2    <= instruction(24 downto 20);
funct7 <= instruction(31 downto 25);
imm    <= instruction(31 downto 20);
```

فایل ثبات‌ها(Register File)

ماژول فایل ثبات‌ها با دریافت rs1و rs2مقادیر دو ورودی ALU را از ثبات‌ها می‌خواند (data1, data2) و همچنین خروجی ALU یا داده خوانده‌شده از حافظه را در آدرس rd می‌نویسد.

```
REGFILE: entity work.RegisterFile port map(clk, rs1, rs2, rd,
write_data, RegWEn, data1, data2);
```

واحد کنترل(Control Unit)

با توجه به opcode, funct3,و funct7، این واحد سیگنال‌های کنترلی زیر را تولید می‌کند:

- ALUSel نوع عملیاتALU

- BSel انتخاب بین داده ثبات یا immediate برای ورودی دومALU
- RegWEn فعال‌سازی نوشتن در فایل ثبات‌ها
- MemRW خواندن یا نوشتن در حافظه
- MemtoReg انتخاب خروجی ALU یا حافظه به عنوان داده نهایی

```
CU: entity work.ControlUnit port map(opcode, funct3, funct7, ALUSel, BSel, RegWEn, MemRW, MemtoReg);
```

تولید مقدارImmediate

ماژول ImmediateGenerator مقدار immediate را از دستور استخراج و sign-extend می‌کند.

```
IMMGEN: entity work.ImmediateGenerator port map(instruction, imm_out);
```

انتخاب ورودی دوم(ALU (Multiplexer

یک مالتی‌پلکسر با توجه به سیگنال BSel تصمیم می‌گیرد که ورودی دوم ALU از data2(مقدار ثبات دوم) باشد یا از imm_out

```
B <= data2 when BSel = '0' else imm_out;
```

واحد حساب و منطق(ALU)

ALU با دریافت data1 و B و سیگنال کنترل ALUSel، عملیات مورد نظر را انجام داده و نتیجه را در alu_outقرار می‌دهد.

```
ALU_UNIT: entity work.ALU port map(data1, B, ALUSel, alu_out);
```

حافظه داده(Data Memory)

در صورت اجرای دستورهای LWیا SW، داده از حافظه خوانده یا در آن نوشته می‌شود. آدرس توسط alu_out تعیین می‌شود و سیگنال MemRW مشخص می‌کند که عملیات خواندن یا نوشتن انجام شود.

```
DMEM: entity work.DataMemory port map(clk, alu_out(5 downto 0), data2, MemRW, mem_out);
```

مالتی‌پلکسر خروجی نهایی(MemtoReg Mux)

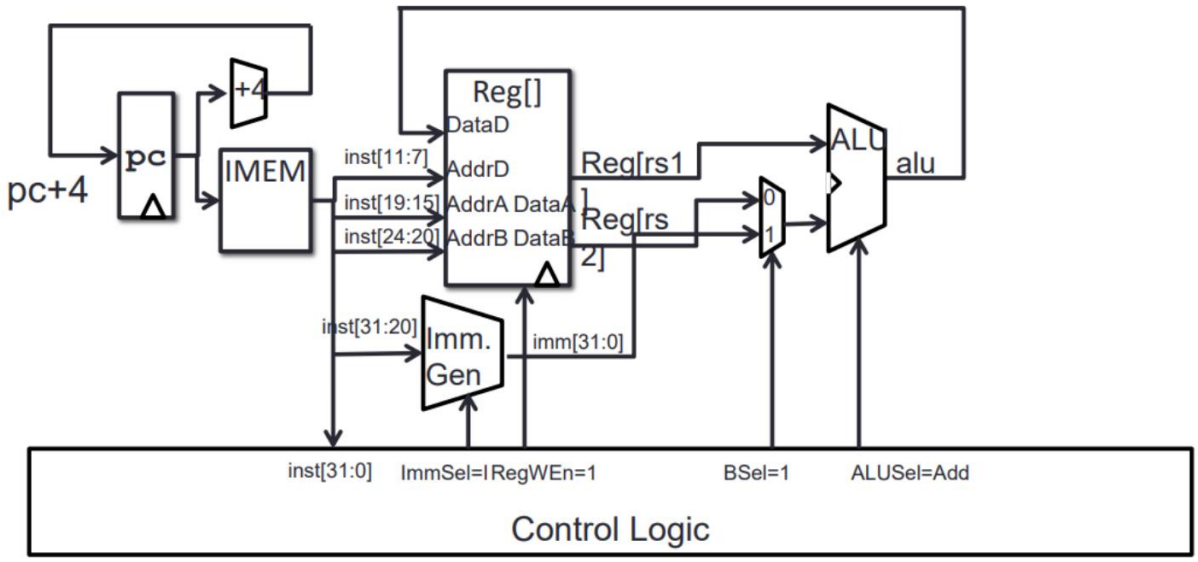
در نهایت، یک مالتی‌پلکسر تصمیم می‌گیرد که مقدار نهایی برای نوشتن در فایل ثبات‌ها از کجا گرفته شود: خروجی ALU یا داده‌ی خوانده‌شده از حافظه.

```
write_data <= alu_out when MemtoReg = '0' else mem_out;
```

۳ .پیاده‌سازی بخش دوم – یکپارچه‌سازی پردازنده

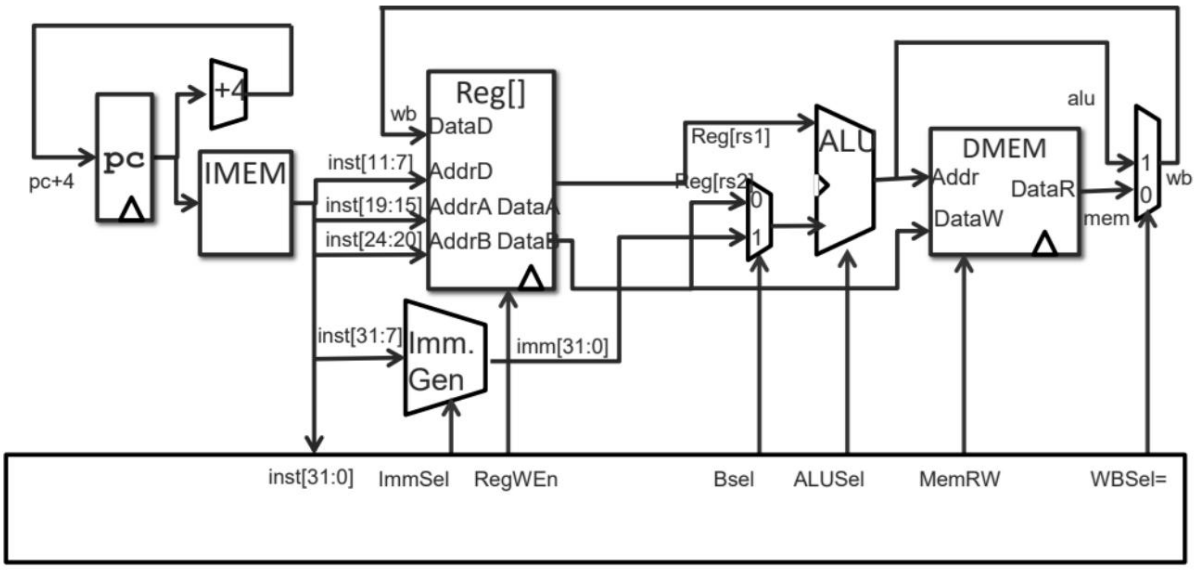
- مسیر داده همان شکل ارجاعی PDF است.
- مالتی‌پلکسر BSelبین خروجی Immediate و رجیستر B.
- مالتی‌پلکسر MemtoRegبین خروجی ALU و خروجی) DMEM در بخش ۳ فعال می‌شود.(
- همه سیگنال‌های کنترل مستقیماً از ControlUnit وارد دیتاپث شده‌اند.

پیچیده‌ترین قسمت، تطبیق هم‌زمان سیگنال‌های BSelو ALUSelبرای دستورات فوری بود؛ با جدول حالت واحد حل شد.



۴ .پیاده‌سازی بخش سوم – حافظه داده و دستورهای LW/SW

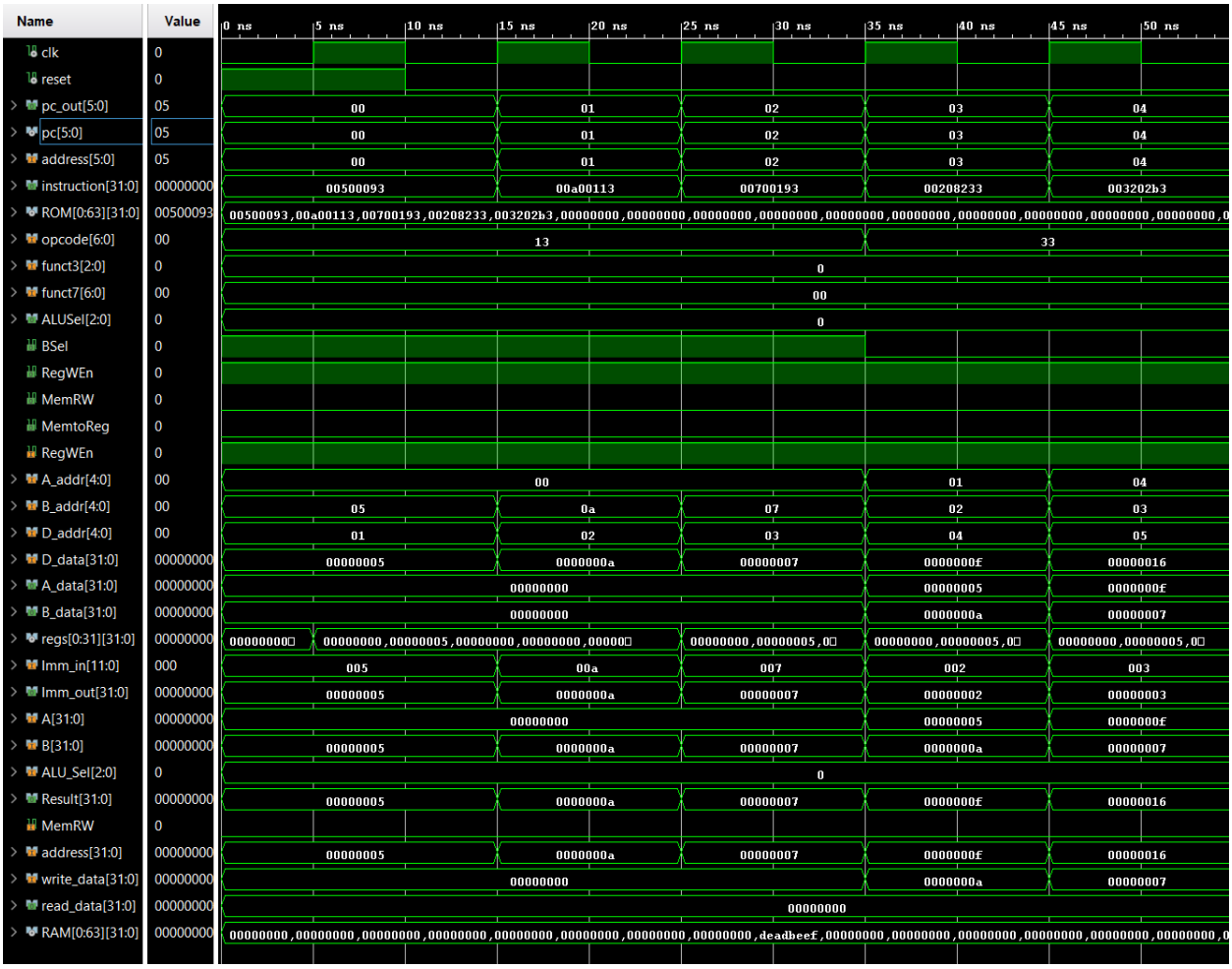
- DataMemory به‌عنوان RAM 64×32 بیتی با آدرس دهی Byte-Addressed (address(7 downto 2)).
- بیت کنترل MemRW = 1 → نوشتن؛ 0 ⟶ فقط خواندن.
- ImmediateGenerator یک بیت اضافی ImmSel دریافت کرد تا برای نوع S (SW) جای فیلدها جابجا شود.
- سیگنال جدید WBsel لازم نشد؛ همان MemtoReg پاسخ‌گو بود.



۵ .شبیه‌سازی و آزمون برنامه‌ها

۵-۱ .کد شماره ۱

```
addi x1,x0,5
addi x2,x0,10
addi x3,x0,7
add  x4,x1,x2
add  x5,x4,x3
```

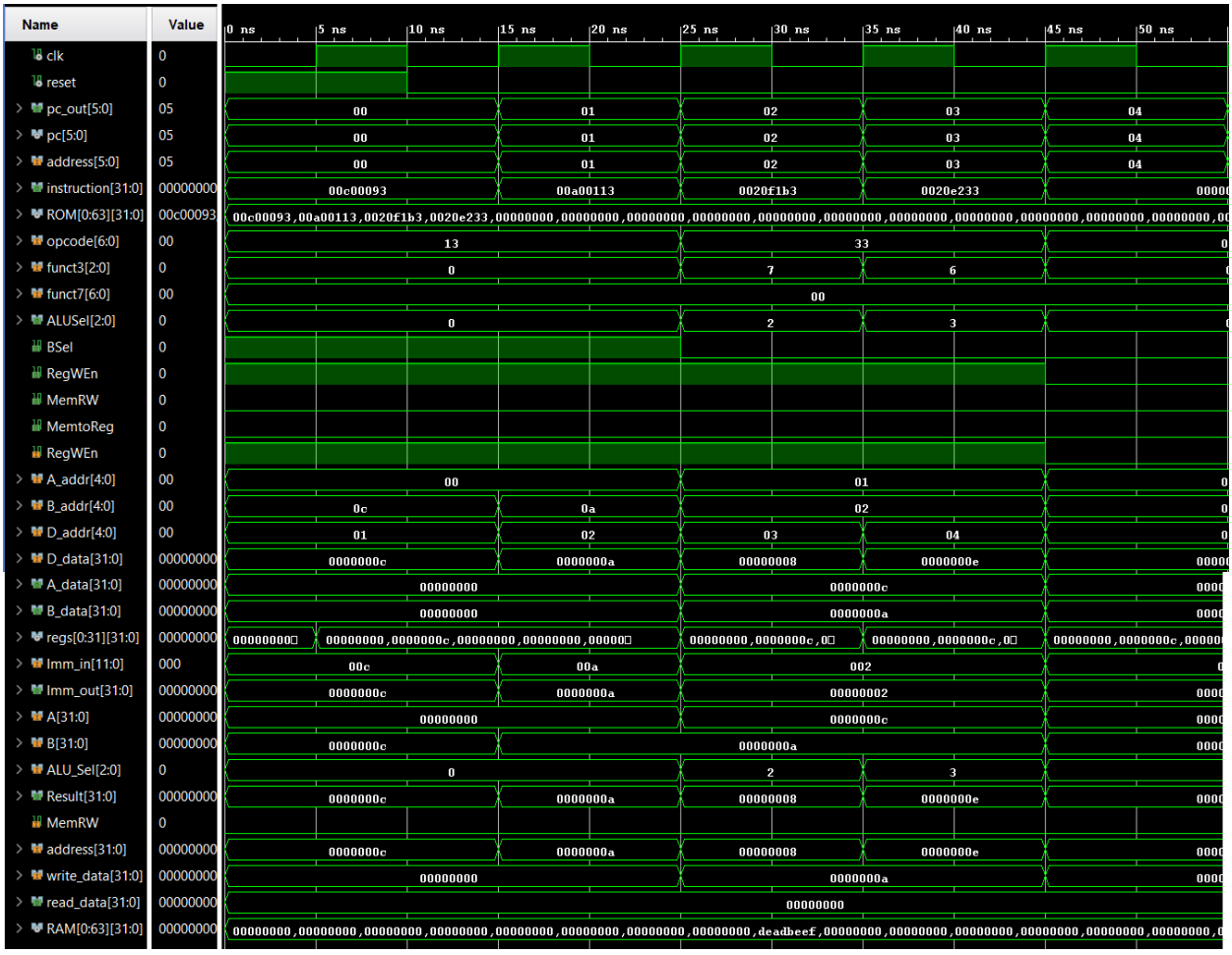


در این تست، ابتدا رجیسترهای x1، x2 و x3 به ترتیب با مقادیر ۵، ۱۰ و ۷ مقداردهی می‌شوند. سپس x4 حاصل جمع x1 و x2 خواهد بود، یعنی x4 = 5 + 10 = 15. در ادامه x5 حاصل جمع x4 و x3 خواهد بود،

یعنی x5 = 15 + 7 = 22. در شبیه‌سازی مشاهده می‌شود که مقدار رجیسترها دقیقاً مطابق انتظار مقداردهی شده‌اند.

۲-۵ .کد شماره ۲

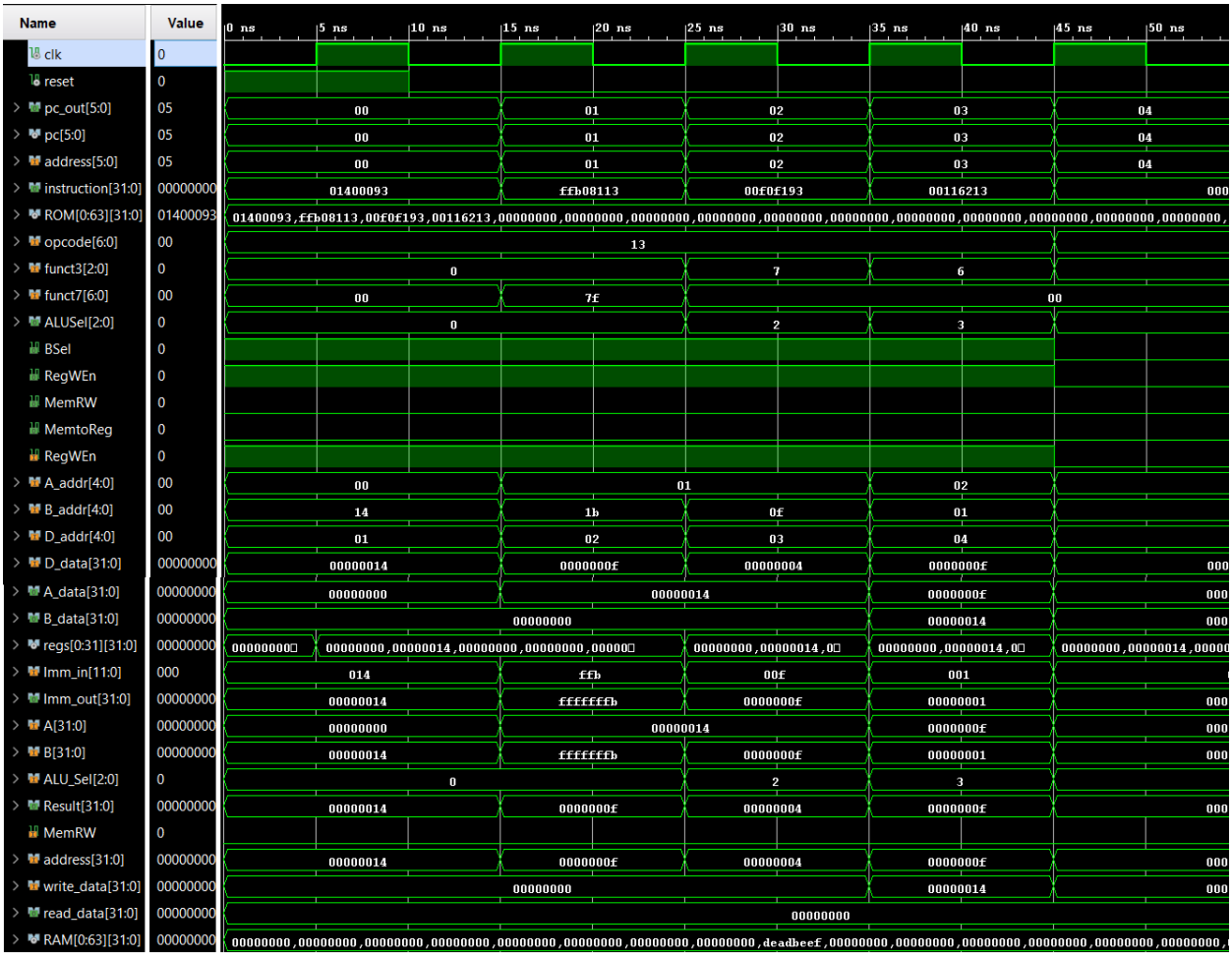
```
addi x1,x0,12
addi x2,x0,10
and  x3,x1,x2
or   x4,x1,x2
```



در این تست، x1 و x2 به‌ترتیب برابر ۱۲ (b11000) و ۱۰ (b10100) می‌شوند. حاصل AND این دو، مقدار x3 = 0b1000 = 8 خواهد بود. حاصل OR آن‌ها نیز x4 = 0b1110 = 14 می‌شود. مقادیر خروجی در شبیه‌سازی این نتایج را تأیید می‌کنند.

۵-۳ .کد شماره ۳

```
addi x1,x0,20
addi x2,x1,-5
andi x3,x1,0x0F
ori  x4,x2,0x01
```



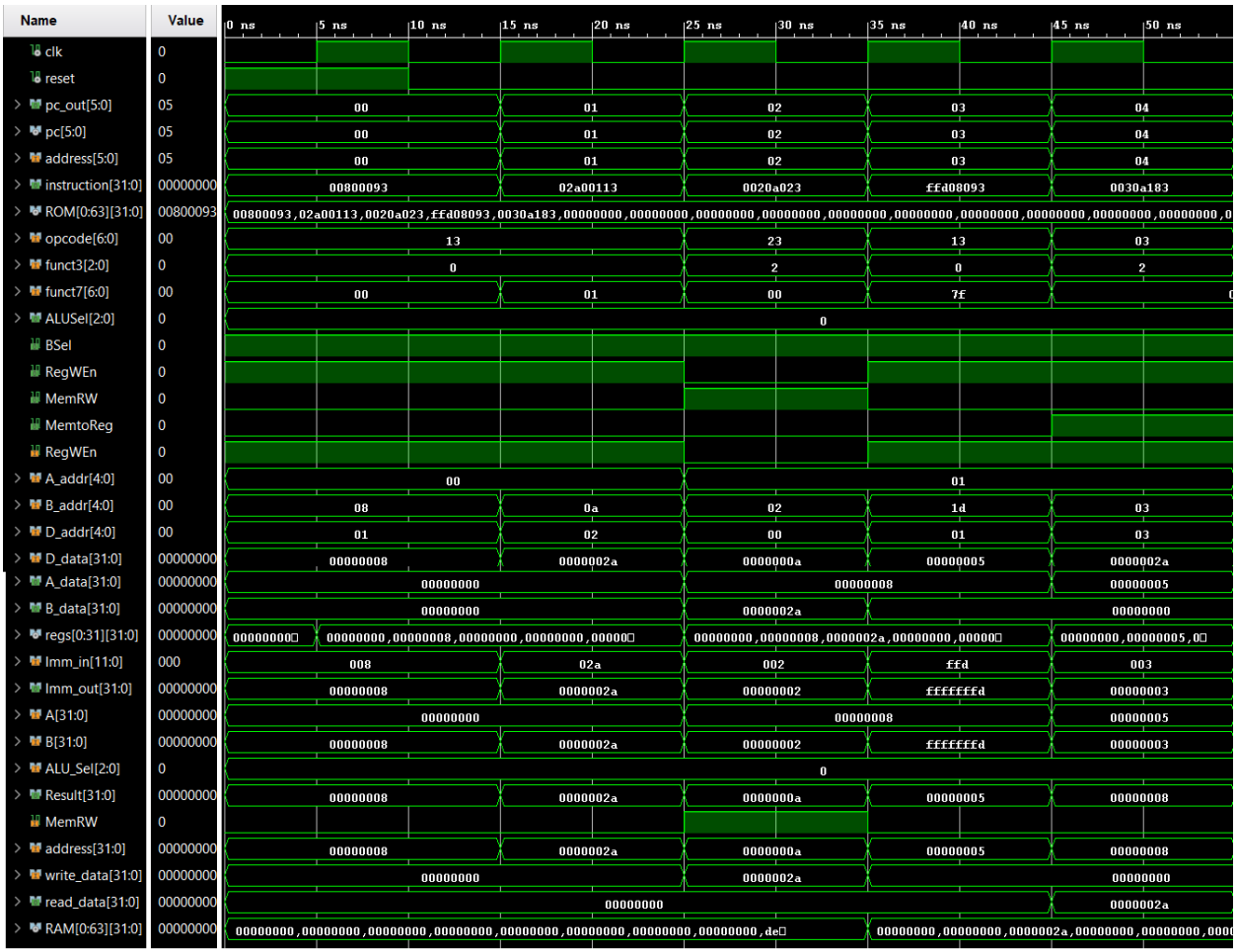
در ابتدا x1 برابر ۲۰ و x2 برابر x1 - 5 = 15 خواهد شد. سپس andi روی x1 با x0F0 انجام می‌شود: x3 = 20 & 0x0F = 4. همچنین ori روی x2 با 1 انجام می‌شود: x4 = 15 | 1 = 15. نتایج رجیسترها در شبیه‌سازی با مقادیر صحیح مطابقت دارند.

۵-۴ .کد شماره ۴ (بخش امتیازی)

```
addi x1,x0,8
```

```
addi x2,x0,42
sw   x2,0(x1)
addi x1,x1,-3
lw   x3,3(x1)
```

- در سیکل SW مقدار 42 در کلمه آدرس 8 ذخیره شد.
- در سیکل LW همان مقدار به x3 بازگشت.



در این بخش، ابتدا x1 = 8 و x2 = 42 می‌شوند. سپس مقدار x2 در آدرس 8 ذخیره می‌شود. در ادامه، x1 به 5 کاهش می‌یابد و سپس مقدار حافظه در آدرس x1 + 3 = 8 بارگذاری می‌شود و در x3 قرار می‌گیرد. بنابراین انتظار می‌رود x3 = 42، که در شبیه‌سازی نیز تأیید شده است.

جمع‌بندی

در این پروژه، یک پردازنده‌ی ساده‌ی تک‌چرخه‌ای بر اساس ISA رایج RISC-V پیاده‌سازی شد که توانایی اجرای مجموعه‌ای از دستورهای محاسباتی (مانند add, sub, and, or، دستورات) immediate مانند addi, andi, oriو همچنین دستورات دسترسی به حافظه (lw, sw) را دارد. در فاز نخست، اجزای پایه شامل فایل ثبات‌ها، واحدALU ، حافظه دستور و حافظه داده، به صورت مجزا طراحی و تست شدند. سپس این ماژول‌ها در قالب فایل CPU.vhdl با یکدیگر ترکیب شدند تا یک مسیر داده‌ی کامل و قابل شبیه‌سازی ایجاد گردد.

در ادامه، برای اطمینان از صحت عملکرد سیستم، چندین برنامه‌ی اسمبلی ساده طراحی شد که پس از ترجمه به دستورالعمل‌های دودویی، در حافظه دستور بارگذاری شدند. نتایج شبیه‌سازی نشان داد که مقادیر تولیدشده در ثبات‌ها با پیش‌بینی‌های تئوری کاملاً منطبق است و در نتیجه، پردازنده از لحاظ عملکردی قابل اتکا می‌باشد.

نکته‌ی قابل توجه آن بود که با وجود سادگی طراحی، مفاهیم پایه‌ای همچون گذرگاه داده، کنترل مسیر، انتخاب ورودی‌های ALU و مدیریت حافظه، به شکل کاملاً عملی تجربه شدند. همچنین توانایی استفاده از ماژول‌های مجزا و ترکیب آن‌ها در یک پروژه‌ی کلی، یکی از مهارت‌های کلیدی در طراحی سیستم‌های دیجیتال محسوب می‌شود که با این پروژه تقویت شد.

در نهایت، باید اشاره کرد که با افزودن بخش حافظه و پیاده‌سازی صحیح کنترل‌های MemRW و MemtoReg، این پروژه نه‌تنها تمامی موارد خواسته‌شده برای نمره‌ی کامل را پوشش داد، بلکه با اجرای صحیح بارگذاری و ذخیره‌سازی حافظه، بخش امتیازی پروژه نیز با موفقیت انجام شد.