ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH

TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN

KHOA KỸ THUẬT MÁY TÍNH

THIẾT KẾ HỆ THỐNG SỐ VỚI HDL

BÁO CÁO ĐỒ ÁN CUỐI KÌ:

THIẾT KẾ BỘ VI XỬ LÍ RISC-V

HỌ VÀ TÊN: NGUYỄN GIA BẢO NGỌC – 21520366
NGUYỄN QUỐC TRƯỜNG AN – 21521810
NGUYỄN TRƯƠNG TIẾN ĐẠT – 21521946

LỚP: CE213.O11.2

GIẢNG VIÊN HƯỚNG DẪN:

HỒ NGỌC DIỄM

TP. HỒ CHÍ MINH – Tháng 12 năm 2023

MỤC LỤC:

Danh mục ảnh:

Danh mục bảng:

I. CƠ SỞ LÝ THUYẾT VÀ TỔNG QUAN ĐỀ TÀI:

I.1 Cơ sở lý thuyết:

I.1.1 Lý do thực hiện đề tài:

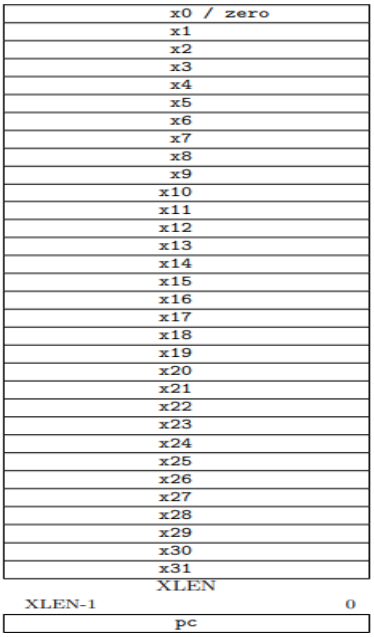
Theo chương trình đào tạo, trong môn học Kiến trúc máy tính (IT006), đã giới thiệu về bộ xử lý được thực thi dựa trên kiến trúc tập lệnh MIPS. Qua môn học, sinh viên được trang bị các kiến thức quan trọng về các thành phần của một vi xử lý (processor) bao gồm: khối đường dữ liệu (datapath) chi tiết hơn là các thiết kế Intruction Memory, Register, ALU, Data Memory,…; khối điều khiển (controller), qua đó hiểu hơn về cách thức hoạt động của một vi xử lý, các công đoạn và cách thức vi xử lý thực hiện một lệnh MIPS.

Nhận thấy đề tài "Thiết kế bộ vi xử lý với kiến trúc tập lệnh RISC-V" mang tính vận dụng kiến thức của 2 môn học là Kiến trúc máy tính và Thiết kế hệ thống số với HDL, đó cũng là lý do nhóm thực hiện đề tài với mục tiêu củng cố và mở rộng, nâng cao kiến thức.

I.1.1 Kiến trúc tập lệnh RISC-V:

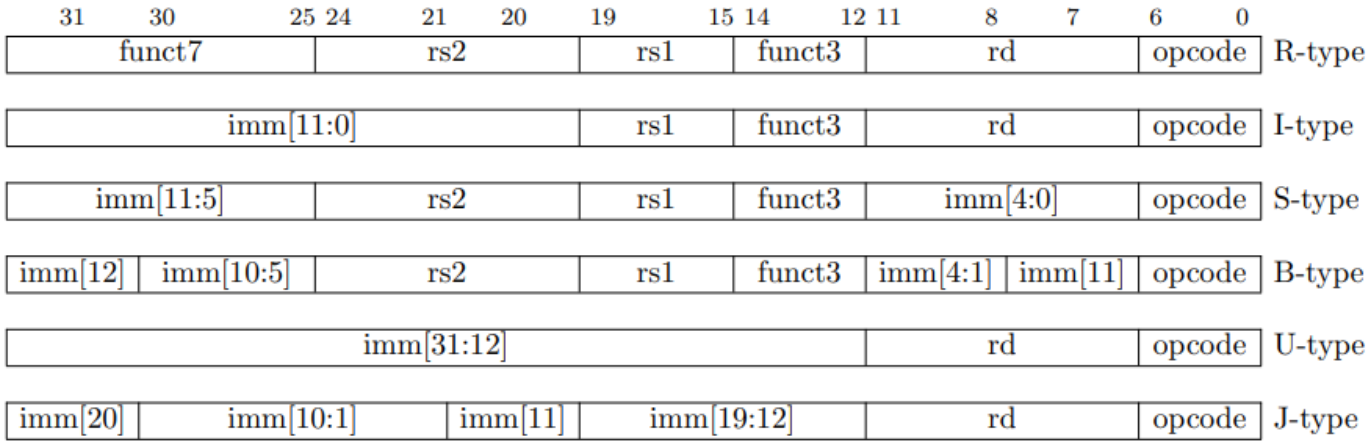
Kiến trúc tập lệnh RISC-V là một kiến trúc tập lệnh mã nguồn mở, với mục đích là phục vụ cho nghiên cứu và giáo dục. Với ưu điểm là mã nguồn mở, kiến trúc tập lệnh RISC-V đang ngày càng trở nên phổ biến và được cộng đồng quan tâm, việc nắm bắt kiến thức về tập lệnh RISC-V là vô cùng hữu ích đối với sinh viên Kỹ thuật máy tính. **Trong phạm vi đề tài sẽ chỉ xoay quanh RV32I** (kiến trúc tập lệnh RISC-V với 32 bit độ dài thanh ghi và 32 bit độ dài lệnh).

Mức độ người dùng, RISC-V cung cấp 32 thanh ghi đa dụng được kí hiệu theo thứ tự từ "x0" đến "x31" và 1 thanh ghi bộ đếm chương trình được kí hiệu là "pc". Đối với RV32I thì mỗi thanh ghi có chiều dài 32 bit, trong đó thanh ghi x0 luôn gắn với hằng số '0'.



Hình 1 - Các thanh ghi ở mức độ người dùng được RISC-V cung cấp.

RV32I thường bao gồm 37 lệnh cơ bản (nếu không kể đến các lệnh môi trường và lệnh hệ thống). Các lệnh RISC-V32I sẽ được chia thanh 6 định dạng là: R-type, I-type, S-type, B-type, U-type, J-type với mỗi lệnh có độ dài 32 bit, tùy theo định dạng mà lệnh RISC-V được chia thành các trường cụ thể như hình bên dưới:



Hình 2 - Định dạng lệnh RISC-V32I

Nguồn tìm hiểu sâu hơn về RISC-V: riscv-spec-v2.2.pdf

I.1.3 Ngôn ngữ mô tả phần cứng Verilog:

Verilog là một ngôn ngữ mô tả phần cứng (HDL - hardware description language), được thiết kế đơn giản và sử dụng hiệu quả ở nhiều mức độ trừu tượng cho nhiều công cụ thiết kế, mô phỏng. Chính vì những ưu điểm này, Verilog được chấp nhận bởi số lượng lớn các nhà thiết kế. Một trong số các đặc trưng quan trọng của Verilog trình bày tiếp sau đây được nhóm áp dụng trong đề tài như là: thiết kế mức hành vi, mức cấu trúc, phân cấp thiết kế,...

Thiết kế theo mức cấu trúc (Structural Modeling): là phương pháp thiết kế trong đó mạch được mô tả theo cấu trúc của mạch dựa trên các thành phần vật lý cơ bản, các cổng logic (AND, OR, XOR, NOT,...) sau đó được kết nối với nhau để tạo hành vi tổng thể của mạch. Mặc dù thiết kế ở mức cấu trúc đã phản ánh chặt chẽ cấu trúc và tính năng của mạch, tuy nhiên ở những mạch phức tạp hơn thì thiết kế mức cấu trúc lại không thể hiện được mức độ trừu tượng cần thiết

Thiết kế theo mức hành vi (Behavior Modeling): là phương pháp thiết kế trong đó mạch được mô tả theo hành vi của mạch. Thiết kế theo mức hành vi có tính trừu tượng cao hơn nhiều so với thiết kế theo hướng cấu trúc, thích hợp trong thiết kế các mạch lớn, phức tạp.

Phân cấp thiết kế (Design Hierarchy): là kĩ thuật cho phép các mô-đun nhỏ hơn được nhúng vào trong mô-đun lớn tạo thành một thiết kế hoàn chỉnh, trong đó mô-đun được hiểu là một phần hay một bộ phần của thiết kế có thể ở là mô hình cấu trúc hay mô hình hành vi.

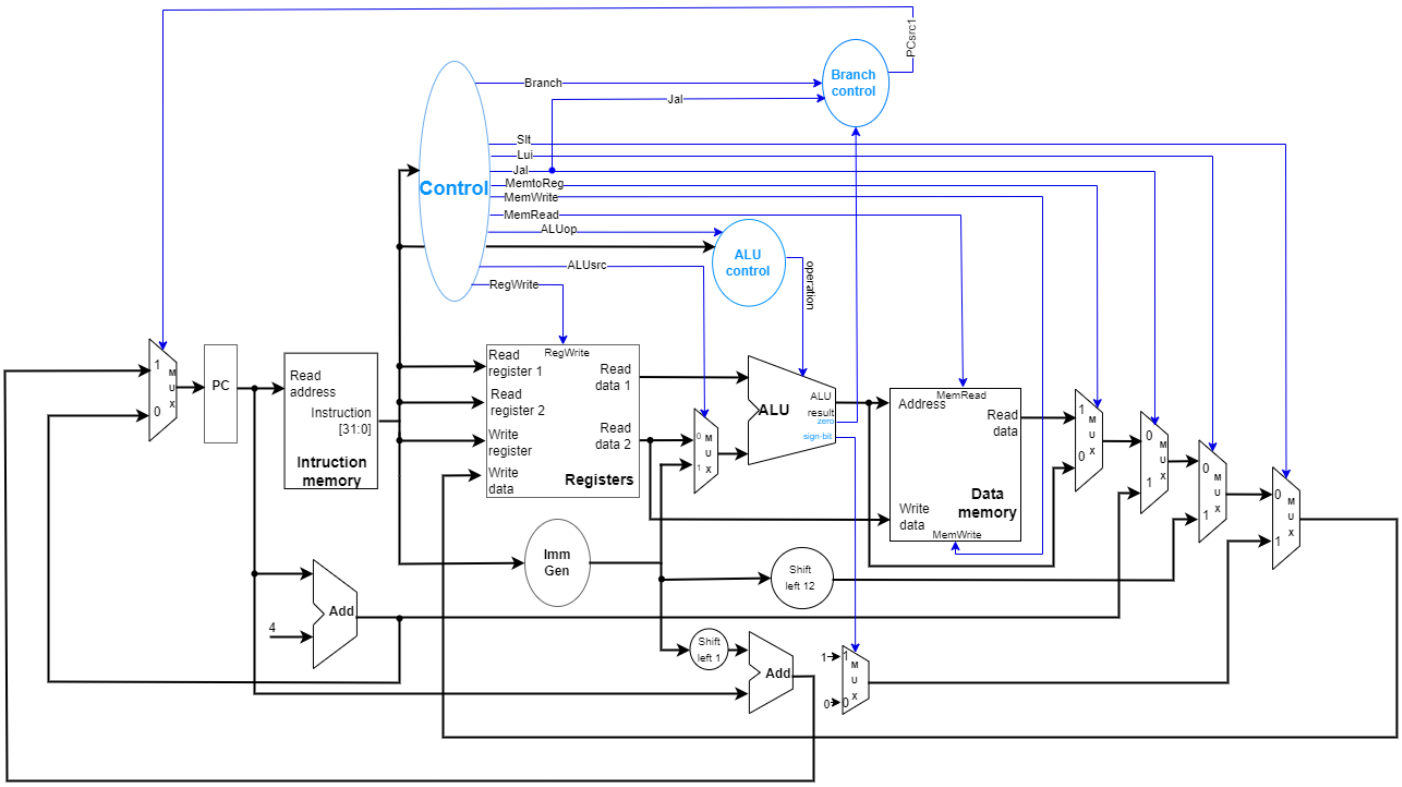
Nguồn tìm hiểu sâu hơn về Verilog: *IEEE Standard for Verilog® Hardware Description Language*

I.2 Tổng quan đề tài :

Dựa trên các kiến thức về RISC-V và Verilog, qua quá trình tìm hiểu và làm việc, nhóm thực hiện đã thiết kế thành công một vi xử lý RISC-V bằng ngôn ngữ mô tả phần cứng Verilog có khả năng thực hiện được các lệnh:

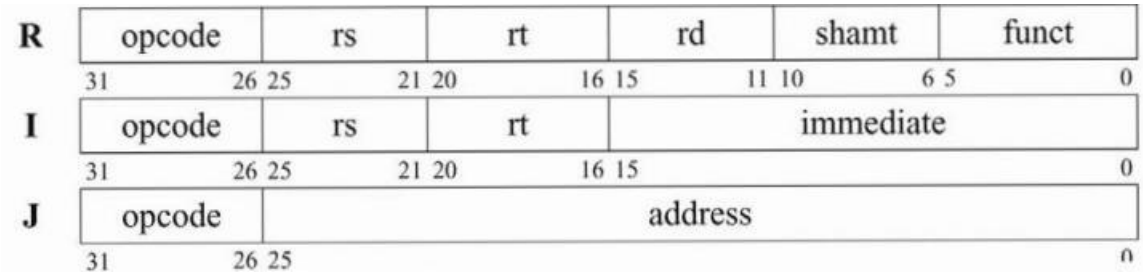
- R-type: add, sub, or, and, slt
- I-type : addi, ori, andi, lw
- S-type: sw
- B-type: beq, bne
- U-type: lui
- J-type : jal

Mô tả thành phần vật lý cũng như sơ đồ các khối trong vi xử lý được minh họa như ảnh bên dưới:

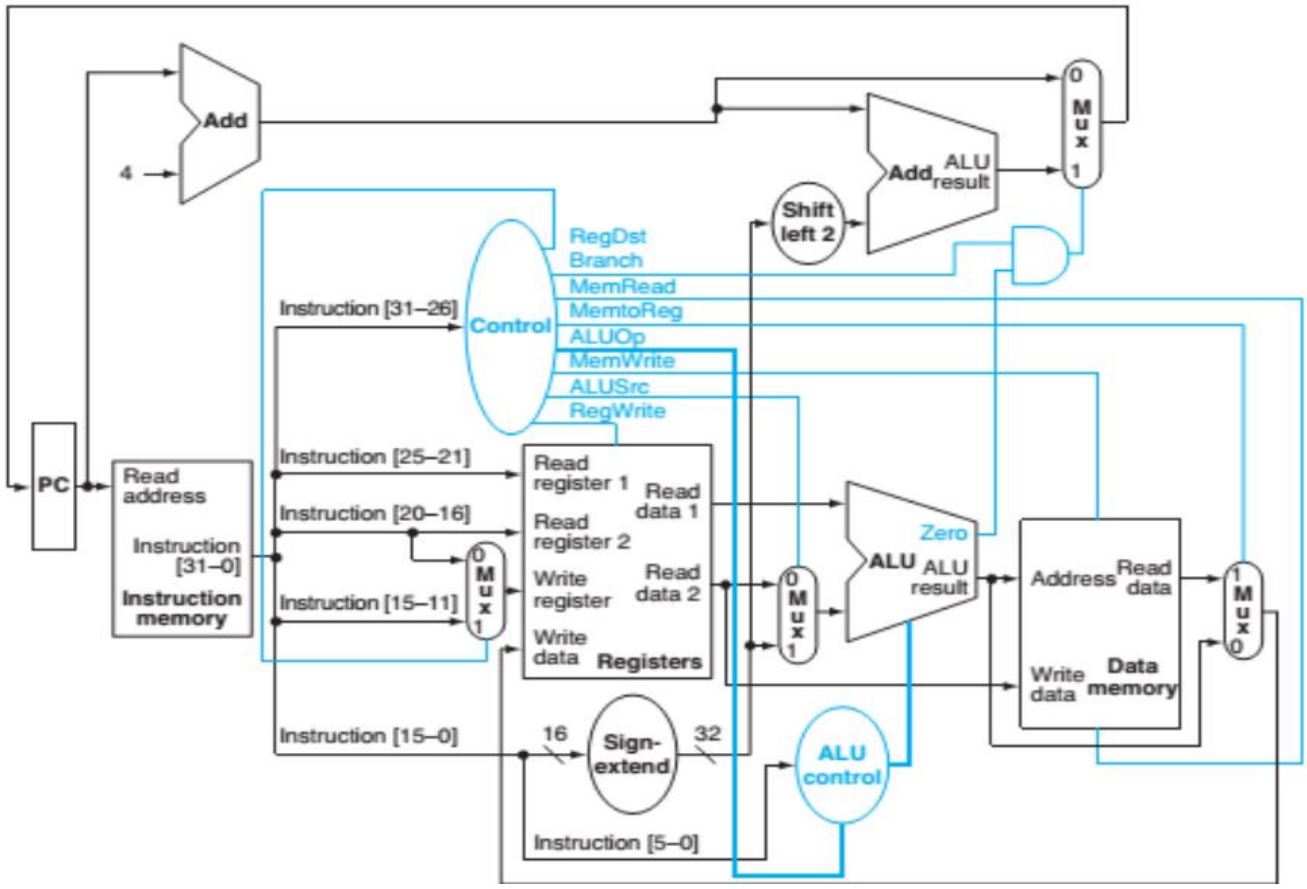


Hình 3 - Bộ vi xử lý RISC-V

Thiết kế bộ vi xử lý RISC-V như trong Hình 3 có những điểm giống nhau nhất định so với bộ vi xử lý MIPS đã được giới thiệu trong môn học Kiến trúc máy tính như các khối Instruction memory, Datamemory, ALU, Register, Control, ALU control,… Tuy nhiên, để phù hợp với kiến trúc tập lệnh mới là RISC-V, đường dữ liệu đã có nhiều sự thay đổi. Sự thay đổi đến chủ yếu đến từ định dạng lệnh khác nhau giữa MIPS và RISC-V. Ngoài ra so với bộ xử lý MIPS được giới thiệu ở môn học trước, bộ vi xử lý nhóm thực hiện có mở rộng thêm một số lệnh liên quan đến hành vi nhảy và rẽ nhánh như *'bne', 'jal'* ngoài ra còn là lệnh *'lui'*, những mở rộng trên đòi hỏi cần thêm sự bổ sung phần cứng đáng kể. Để có sự so sánh, hai hình 4 và 5 bên dưới là định dạng lệnh MIPS và Datapath của bộ xử lý MIPS được giới thiệu trong môn học Kiến trúc máy tính



Hình 4 - Định dạng lệnh MIPS



Hình 5 - Datapath của bộ xử lý MIPS được giới thiệu trong môn học Kiến trúc máy tính

Với tập lệnh được thiết kế gồm 14 lệnh, trong đó 6 định dạng lệnh RISC-V đều có ít nhất 1 lệnh được thiết kế, vi xử lý do nhóm thiết kế có thể thực thi được một số giải thuật đơn giản. Ưu điểm của đề tài là việc ứng dụng chương trình "assembler", qua đó có thể nạp dữ liệu cho vi xử từ ngôn ngữ bậc cao hơn, mở rộng tập người dùng, tạo thuận lợi đáng kể trong quá trình làm việc. Với thiết kế như Hình 3, đề tài có thể phát triển hơn trong việc mở rộng thêm tập lệnh, không chỉ dừng lại ở con số 14, thực thi đa chu kì thông qua kĩ thuật Pipeline là những hướng phát triển trong tương lai mà nhóm có thể thực hiện.

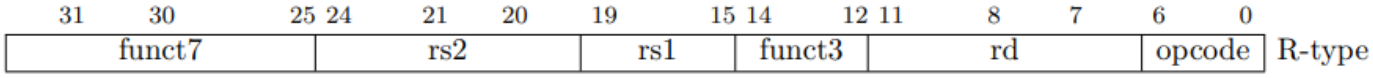
II. CHI TIẾT THIẾT KẾ:

II.1 Kiến trúc tập lệnh:

Như đã trình bày ở phần tổng quan đề tài, bộ vi xử lý RISC-V do nhóm thực hiện có tập lệnh bao gồm 14 lệnh, chi thiết từng lệnh sẽ được trình bày cụ thể như sau:

II.1.1 Các lệnh R-type (add, sub, or, and, slt):

Định dạng lệnh R-type như hình sau:



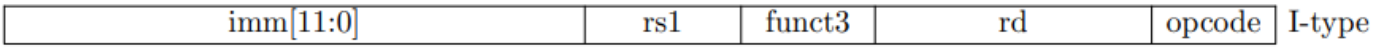
Hình 6 - Định dạng lệnh R-type

Cụ thể:

- Lệnh add: thực hiện phép cộng giá trị tại thanh ghi rs1 với rs2 sau đó gán giá trị vào thanh ghi rd. **Opcode** = 0110011, **funct3** = 000, **funct7** = 0000000.
- Lệnh sub: thực hiện phép trừ giá trị tại thanh ghi rs1 với rs2 sau đó gán giá trị vào thanh ghi rd. **Opcode** = 0110011, **funct3** = 000, **funct7** = 0100000.
- Lệnh or: thực hiện phép toán logic or giá trị tại thanh ghi rs1 với rs2 sau đó gán giá trị vào thanh ghi rd. **Opcode** = 0110011, **funct3** = 110, **funct7** = 0000000.
- Lệnh and: thực hiện phép toán logic and giá trị tại thanh ghi rs1 với rs2 sau đó gán giá trị vào thanh ghi rd. **Opcode** = 0110011, **funct3** = 111, **funct7** = 0000000.
- Lệnh slt: so sánh giá trị tại thanh ghi rs1 với rs2, nếu giá trị tại rs1 lớn hơn thì ghi vào thanh ghi rd giá trị 1 ngược lại ghi vào rd giá trị 0. **Opcode** = 0110011, **funct3** = 010, **funct7** = 0000000.

II.1.2 Các lệnh I-type (addi, ori, andi, lw):

Các lệnh I-type, là các nhóm lệnh làm việc với các giá trị tức thời hay (‘offset’ hay ‘immediate’). Định dạng lệnh I-type như hình sau:

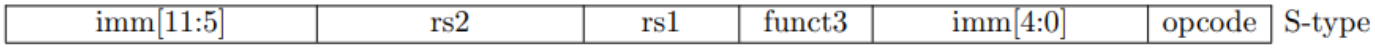


Hình 7 - Định dạng lệnh I-type

- Lệnh addi: thực hiện phép cộng giá trị tại thanh ghi rs1 với giá trị tức thời 12 bit tại trường imm[11:0] sau đó gán giá trị vào thanh ghi rd. **Opcode** = 0010011, **funct3** = 000.
- Lệnh ori: thực hiện phép toán logic or giá trị tại thanh ghi rs1 với giá trị tức thời 12 bit tại trường imm[11:0] sau đó gán giá trị vào thanh ghi rd. **Opcode** = 0010011, **funct3** = 110.
- Lệnh andi: thực hiện phép toán logic and giá trị tại thanh ghi rs1 với giá trị tức thời 12 bit tại trường imm[11:0] sau đó gán giá trị vào thanh ghi rd. **Opcode** = 0010011, **funct3** = 111.
- Lệnh lw: thực hiện việc lấy giá trị 32 bit trong Data memory tại địa chỉ có giá trị bằng với giá trị thanh ghi rs1 cộng với giá trị tức thời 12 bit tại trường imm[11:0]. **Opcode** = 0000011, **funct3** = 010.

II.1.3 Lệnh S-type (sw):

Các lệnh lệnh S-type là các lệnh truy cập Data memory gán giá trị tại vùng nhớ trong Data memory cho thanh ghi đích, các lệnh S-type có định dạng như sau:

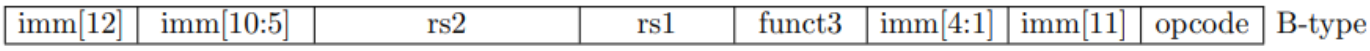


Hình 8 - Định dạng lệnh S-type

- Lệnh sw: thực hiện việc lấy giá trị 32 bit trong Data memory tại địa chỉ có giá trị bằng với giá trị thanh ghi rs1 cộng với giá trị tức thời 12 bit tại trường imm[11:0] sau đó gán cho thanh ghi rd. **Opcode** = 0100011, **funct3** = 010.

II.1.4 Các lệnh B-type (beq, bne):

Các lệnh B-type là các lệnh rẽ nhánh điều kiện, so sánh giá trị trong thanh ghi rs1 và rs2 để làm điều kiện nhảy. Giá trị địa chỉ lệnh nhảy đến được tính toán như sau: pc = pc + {imm, 1'b0}, định dạng lệnh B-type như hình sau:



Hình 9 - Định dạng lệnh B-type

- Lệnh beq: so sánh giá trị giá trị tại thanh ghi rs1 với giá trị thanh ghi rs2 nếu hai giá trị bằng nhau thì thực hiện hành vi nhảy bằng cách gán giá trị pc = pc + {imm, 1'b0}. **Opcode** = 1100011, **funct3** = 000.
- Lệnh bne: so sánh giá trị giá trị tại thanh ghi rs1 với giá trị thanh ghi rs2 nếu hai giá trị không bằng nhau thì thực hiện hành vi nhảy bằng cách gán giá trị pc = pc + {imm, 1'b0}. **Opcode** = 1100011, **funct3** = 001.

II.1.5 Lệnh U-type (lui):

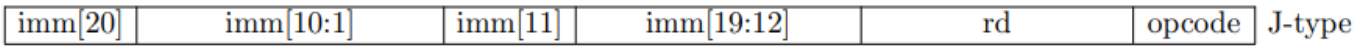
Các lệnh U-type thường được dùng phối hợp với lệnh I-type gán các giá trị lớn hơn 12 bit vào thanh ghi đích. Định dạng lệnh U-type như hình sau:

Hình 10 - Định dạng lệnh U-type

- Lệnh lui: thực hiện gán cho thanh ghi rd giá trị {imm, 12'b0}. **Opcode** = 0110111.

II.1.6 Lệnh J-type (jal):

Các lệnh J-type được sử dụng để nhảy đến một địa chỉ bất kì trong chương trình nhưng khác với các lệnh thuộc B-type thì các lệnh J-type thực hiện hành vi nhảy không cần điều kiện, định dạng lệnh J-type như hình:



Hình 11 - Định dạng lệnh J-type

- Lệnh jal: nhảy đến địa chỉ được xác định bằng pc = pc + {imm, 1'b0}, đồng thời thanh ghi đích lưu giá trị địa chỉ tại lệnh kế tiếp (rd = pc+4). **Opcode** = 1101111

II.2 Thiết kế Verilog:

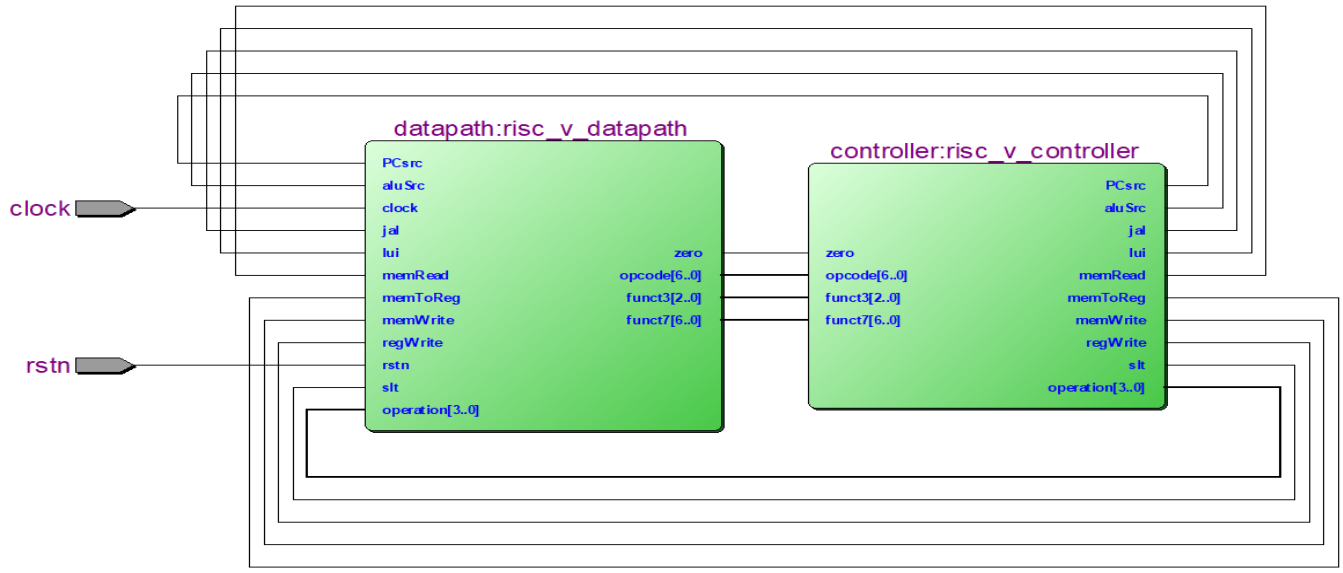
II.2.1 Module RISC_V:

Bảng 1 - module RISC_V

```
module RISC_V (clock,
                rstn);

    parameter PATH_IMEM    = "D:/Final_Project/Risc_V_Verilog/testcase/iMem.bin";
    parameter PATH_REGFILE = "D:/Final_Project/Risc_V_Verilog/testcase/regFile.bin";
    parameter PATH_DMEM    = "D:/Final_Project/Risc_V_Verilog/testcase/dMem.bin";

    input clock;
    input rstn;

    wire [6:0] opcode;
    wire [2:0] funct3;
    wire [6:0] funct7;
    wire zero;

    wire clock;
    wire rstn;
    wire PCsrc;
    wire slt;
    wire lui;
    wire jal;
    wire memToReg;
    wire memWrite;
    wire memRead;
    wire [3:0] operation;
    wire aluSrc;
    wire regWrite;

    controller risc_v_controller ( .PCsrc(PCsrc),
```

```
                                        .slt(slt),
                                        .lui(lui),
                                        .jal(jal),

                                        .memToReg(memToReg),

                                        .memWrite(memWrite),

                                        .memRead(memRead),
                                        .operation(operation),
                                        .aluSrc(aluSrc),
                                        .regWrite(regWrite),
                                        .opcode(opcode),
                                        .funct3(funct3),
                                        .funct7(funct7),
                                        .zero(zero)
                                       );

        datapath #( .PATH_IMEM(PATH_IMEM),
                                        .PATH_REGFILE(PATH_REGFILE),
                                        .PATH_DMEM(PATH_DMEM)
                                        ) risc_v_datapath ( .opcode(opcode),
                                        .funct3(funct3),
                                        .funct7(funct7),
                                        .zero(zero),
                                        .clock(clock),
                                        .rstn(rstn),
                                        .PCsrc(PCsrc),
                                        .slt(slt),
                                        .lui(lui),
                                        .jal(jal),

                                        .memToReg(memToReg),

                                        .memWrite(memWrite),

                                        .memRead(memRead),

                                        .operation(operation),
                                        .aluSrc(aluSrc),
                                        .regWrite(regWrite)
                                    );
endmodule
```

Module này là Top Module của thiết kế được tạo thành từ DATAPATH và CONTROLLER tạo thành một CPU RISC-V hoàn chỉnh.



Hình 12 - Kết quả RTL viewer module RISC_V

II.2.2 Module Datapath:

Bảng 2 - module datapath

```
module datapath (opcode,
                 funct3,
                 funct7,
                 zero,
                 clock,
                  rstn,
                 PCsrc,
                 slt,
                 lui,
                  jal,
                 memToReg,
                 memWrite,
                 memRead,
                 operation,
                 aluSrc,
                 regWrite
                                    );

        parameter ADDR_WIDTH = 32;
        parameter DATA_WIDTH = 32;
        parameter PATH_IMEM = "D:/Final_Project/Risc_V_Verilog/testcase/iMem.bin";
        parameter PATH_REGFILE = "D:/Final_Project/Risc_V_Verilog/testcase/regFile.bin";
        parameter PATH_DMEM = "D:/Final_Project/Risc_V_Verilog/testcase/dMem.bin";

        output [6:0] opcode;
```

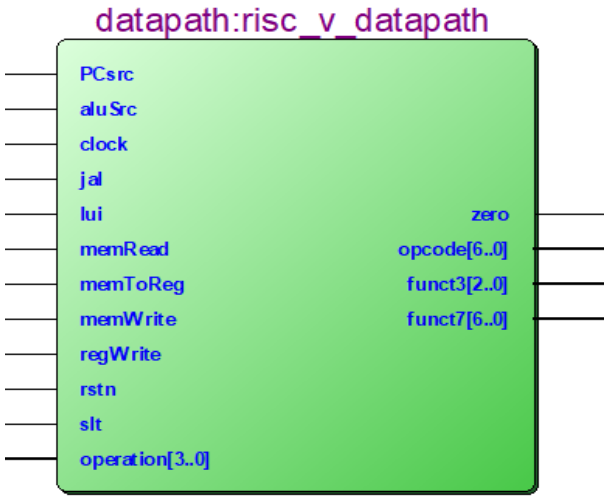
```
output [2:0] funct3;
output [6:0] funct7;
output zero;

input clock;
input rstn;
input PCsrc;
input slt;
input lui;
input jal;
input memToReg;
input memWrite;
input memRead;
input [3:0] operation;
input aluSrc;
input regWrite;

wire [ADDR_WIDTH-1 : 0] pc;
wire [ADDR_WIDTH-1 : 0] pcNext;
wire [ADDR_WIDTH-1 : 0] pc4;
wire [ADDR_WIDTH-1 : 0] pcImm;

wire [DATA_WIDTH-1 : 0] instruction;
wire [DATA_WIDTH-1 : 0] readData1;
wire [DATA_WIDTH-1 : 0] readData2;
wire [DATA_WIDTH-1 : 0] writeData;
wire [DATA_WIDTH-1 : 0] aluOut;
wire [DATA_WIDTH-1 : 0] aluSrcB;
wire [DATA_WIDTH-1 : 0] readDataDmem;
wire [DATA_WIDTH-1 : 0] out_memToReg;
wire [DATA_WIDTH-1 : 0] out_jal;
wire [DATA_WIDTH-1 : 0] out_lui;
wire [DATA_WIDTH-1 : 0] out_shiftLeft12;
wire [DATA_WIDTH-1 : 0] out_signBit;
wire [DATA_WIDTH-1 : 0] out_immGenerator;
wire [DATA_WIDTH-1 : 0] out_shiftLeft1;

wire signBit;


FFD_resettable pcreg( .q(pc),
                      .clock(clock),
                      .rstn(rstn),
                      .d(pcNext)      // pc_next
                      );
```

```
instructionMemory #(.PATH(PATH_IMEM))imem(.readData(instruction),
                                          .clock(clock),
                                          .readAddress(pc[11:2])
                                                );

regFile #(.PATH(PATH_REGFILE)) regfile(.readData1(readData1),
                                       .readData2(readData2),
                                       .clock(clock),
                                       .readReg1(instruction[19:15]),
                                       .readReg2(instruction[24:20]),
                                       .regWrite(regWrite),
                                       .writeData(writeData),
                                       .writeReg(instruction[11:7])
                                       );

alu alu (.zero(zero),
        .signBit(signBit),
        .result(aluOut),
        .operation(operation),
        .a(readData1),
        .b(aluSrcB)                                     // alu source B
                );

dataMemory #(.PATH(PATH_DMEM)) dmem (.readData(readDataDmem),
                                          .clock(clock),
                                          .memRead(memRead),
                                          .memWrite(memWrite),
                                          .writeData(readData2),
                                          .address(aluOut[11:2])
                                                                        );

immGenerator immGen (       .immOut(out_immGenerator),
                            .instruction(instruction)
                                          );

shiftLeft shiftLeft_12(.out(out_shiftLeft12),
                        .a(out_immGenerator),
                        .b(32'd12)
                                           );

shiftLeft shiftLeft_1 ( .out(out_shiftLeft1),
                        .a(out_immGenerator),
                        .b(32'd1)
                        );

mux mux_memToReg (.out(out_memToReg),
```

```
                                .sel(memToReg),
                                .a(aluOut),
                                .b(readDataDmem)
                                                );

mux mux_jal( .out(out_jal),
                .sel(jal),
                .a(out_memToReg),
                .b(pc4)
                                );

mux mux_lui(.out(out_lui),
                .sel(lui),
                .a(out_jal),
                .b(out_shiftLeft12)
                                );

mux mux_slt(.out(writeData),
                .sel(slt),
                .a(out_lui),
                .b(out_signBit)
                                );

mux mux_signBit(.out(out_signBit),
                        .sel(signBit),
                        .a(32'd0),
                        .b(32'd1)
                           );

mux mux_aluSrcB(.out(aluSrcB),
                        .sel(aluSrc),
                        .a(readData2),
                        .b(out_immGenerator)
                                        );

mux mux_pcNext (.out(pcNext),
                        .sel(PCsrc),
                        .a(pc4),
                        .b(pcImm)
                        );

adder adder4(.out(pc4),
                .a(pc),
                .b(32'd4)
                );
```

```
        adder adderShiftLeft1(.out(pcImm),
                              .a(out_shiftLeft1),
                              .b(pc)
                              );

        assign opcode = instruction[6:0];
        assign funct3 = instruction[14:12];
        assign funct7 = instruction[31:25];

endmodule
```

Module này thực hiện gọi, kết nối các module: FFD_resettable, instructionMemory, regFile, alu, dataMemory, immGenerator, shiftLeft, mux, adder để tạo thành một DATAPATH hoàn chỉnh thực hiện được các lệnh theo kiến trúc RISC-V trong một chu kỳ (đơn chu kỳ).



Hình 13 - Kết quả RTL viewer module datapath

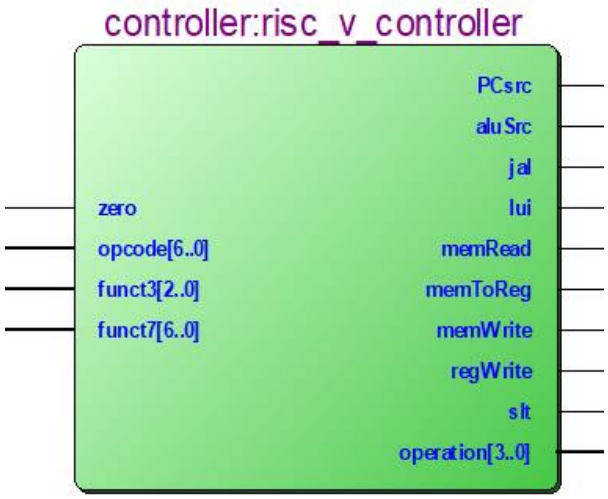
II.2.3 Module Controller:

Bảng 3 - module controller

```
module controller (PCsrc,
                   slt,
                   lui,
                   jal,
                   memToReg,
                   memWrite,
                   memRead,
                   operation,
                   aluSrc,
                   regWrite,
                   opcode,
                   funct3,
                   funct7,
                   zero
                                        );

    output PCsrc;
    output slt;
    output lui;
    output jal;
    output memToReg;
    output memWrite;
    output memRead;
    output [3:0] operation;
    output aluSrc;
    output regWrite;

    input [6:0] opcode;
    input [2:0] funct3;
    input [6:0] funct7;
    input zero;

    wire [1:0] branch;
    wire [1:0] aluOp;

    control ctl( .branch(branch),
                 .slt(slt),
                 .lui(lui),
                 .jal(jal),
                 .memToReg(memToReg),
                 .memWrite(memWrite),
                 .memRead(memRead),
                 .aluOp(aluOp),
```

```
                    .aluSrc(aluSrc),
                    .regWrite(regWrite),
                    .opcode(opcode),
                    .funct3(funct3)
                                );

    aluControl aluCtl(.operation(operation),
                      .aluOp(aluOp),
                      .funct3(funct3),
                      .funct7(funct7)
                                          );

    branchControl brCtl ( .PCsrc(PCsrc),
                      .branch(branch),
                      .jal(jal),
                      .zero(zero)
                                                  );
endmodule
```

Module này thực hiện gọi, kết nối các module: control, aluControl, branchControl để tạo thành một khối CONTROLLER hoàn chỉnh điều khiển mọi hoạt động của datapath.



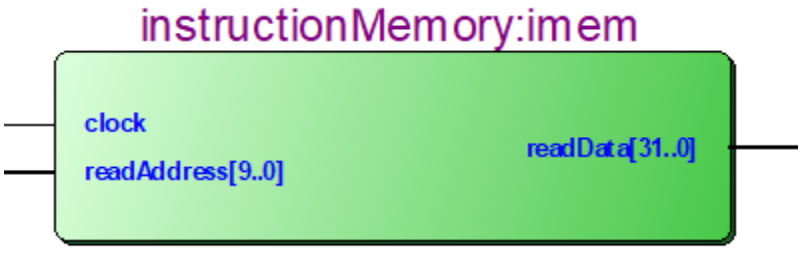
Hình 14 - Kết quả RTL viewer module controller

II.2.3 Các thành phần tạo thành module Datapath:

Bảng 4 - module instructionMemory

```
module instructionMemory (readData,
                          clock,
                          readAddress
                          );

    parameter DATA_WIDTH = 32;
    parameter ADDR_WIDTH = 6;
    parameter PATH = "D:/Final_Project/Risc_V_Verilog/testcase/iMem.bin";

    output [DATA_WIDTH-1 : 0] readData;

    input clock;
    input [ADDR_WIDTH-1 : 0] readAddress;

    reg [DATA_WIDTH-1 : 0] readData;
    reg [DATA_WIDTH-1 : 0] imem[2**ADDR_WIDTH-1 : 0];

    //always @(posedge clock) begin

        //readData <= imem[readAddress];

    //end

    always @(*) begin

        readData = imem[readAddress];

    end
    initial $readmemb(PATH, imem);
endmodule
```

Module này chứa tập mã lệnh ở dạng mã máy được đọc lên từ file "iMem.bin", mỗi lệnh ứng với 32 bits. Module được thiết kế như một module RAM với độ dài một word là 32 bits và có 64 words (ứng với chương trình tối đa 64 lệnh vì giới hạn của phần mềm Quartus không cho thiết kế bộ nhớ có dung lượng lớn).



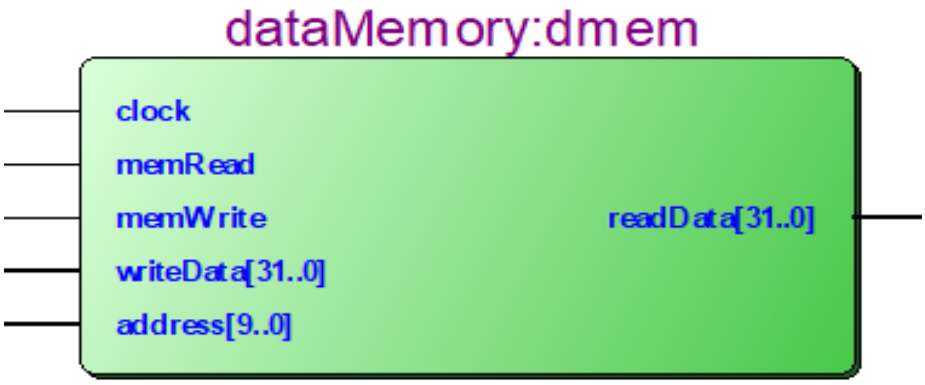
Hình 15 - Kết quả RTL viewer module instructionMemory

Bảng 5 - module dataMemory

```
module dataMemory (readData,
                   clock,
                   memRead,
                   memWrite,
                   writeData,
                   address
                                        );

    parameter DATA_WIDTH = 32;
    parameter ADDR_WIDTH = 6;
    parameter PATH = "D:/Final_Project/Risc_V_Verilog/testcase/dMem.bin";

    output [DATA_WIDTH-1 : 0] readData;

    input clock;
    input memRead;
    input memWrite;
    input [DATA_WIDTH-1 : 0] writeData;
    input [ADDR_WIDTH-1 : 0] address;

    reg [DATA_WIDTH-1 : 0] readData;
    reg [DATA_WIDTH-1 : 0] dmem[2**ADDR_WIDTH-1 : 0];

    initial begin: INIT
        integer i;
        for(i = 0; i < 2**ADDR_WIDTH; i = i + 1)
            dmem[i] <= 32'h00000000;
    end

    always @(posedge clock) begin

        if(memWrite) begin

            dmem[address] <= writeData;

        end
        //if(memRead) begin

            //readData <= dmem[address];

        //end

        $writememb(PATH, dmem);

    end
```

```
        always @(*)
                if(memRead)
                        readData <= dmem[address];

endmodule
```

Module này là một module RAM có 64 words mỗi words 32 bits dùng để lưu và chứa dữ liệu của chương trình khi thực thi các lệnh sw, lw.



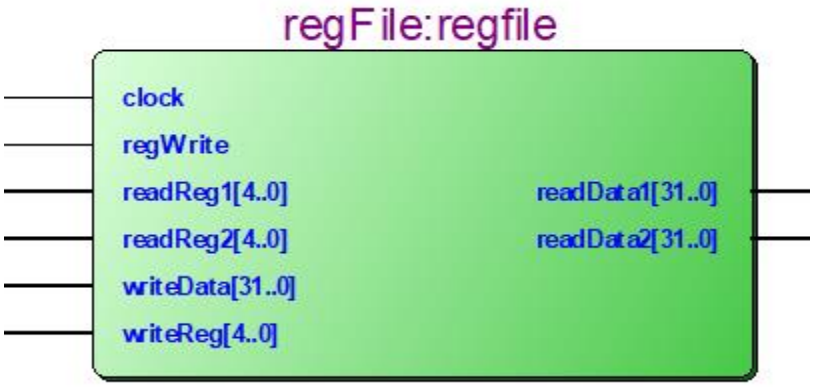
Hình 16 - Kết quả RTL viewer module dataMemory

Bảng 6 - module regFile

```
module regFile(readData1,
                readData2,
                clock,
                readReg1,
                readReg2,
                regWrite,
                writeData,
                writeReg
                                );

        parameter DATA_WIDTH = 32;
        parameter ADDR_WIDTH = 5;
        parameter PATH = "D:/Final_Project/Risc_V_Verilog/testcase/regFile.bin";

        output [DATA_WIDTH-1 : 0] readData1;
        output [DATA_WIDTH-1 : 0] readData2;

        input clock;
        input [ADDR_WIDTH-1 : 0] readReg1;
        input [ADDR_WIDTH-1 : 0] readReg2;
        input regWrite;
        input [DATA_WIDTH-1 : 0] writeData;
        input [ADDR_WIDTH-1 : 0] writeReg;

        reg [DATA_WIDTH-1 : 0] readData1;
        reg [DATA_WIDTH-1 : 0] readData2;
        reg [DATA_WIDTH-1 : 0] regfile[2**ADDR_WIDTH-1 : 0];

        initial begin: INIT
                integer i;
                for(i = 0; i < 2**ADDR_WIDTH; i = i + 1)
                        regfile[i] <= 32'h00000000;
        end

        always @(posedge clock) begin

                if(regWrite && writeReg != 0) begin

                        regfile[writeReg] <= writeData;

                end

                $writememb(PATH, regfile);

        end
```

```
    always @(readReg1 or readReg2) begin

        readData1 = (readReg1 != 0) ? regfile[readReg1] : 0;
        readData2 = (readReg2 != 0) ? regfile[readReg2] : 0;

    end

endmodule
```

Module này là một Register File gồm 32 thanh ghi, mỗi thanh ghi có độ dài 32 bits ứng với các thanh ghi x0-x31 trong kiến trúc RISC-V. Dùng để lưu dữ liệu của các thanh ghi khi thực hiện chương trình.



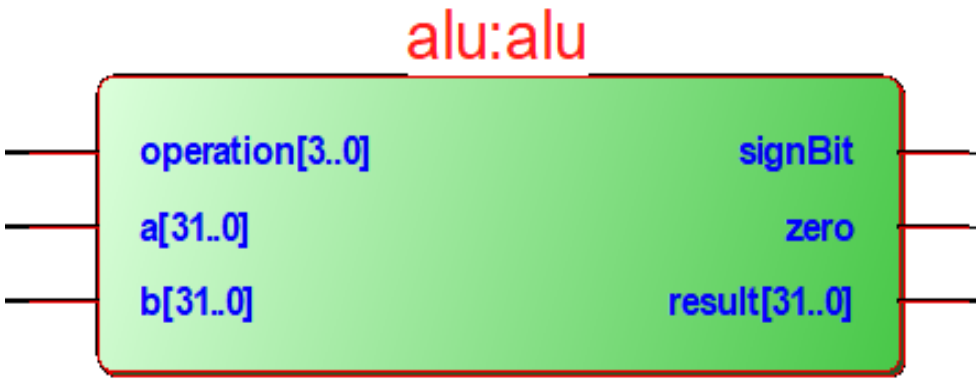
Hình 17 - Kết quả RTL viewer module regFlie

Bảng 7 - module ALU

```
module alu (zero,
                signBit,
                result,
                operation,
                a,
                b
                );

        parameter DATA_WIDTH = 32;

        output zero;
        output signBit;
        output [DATA_WIDTH-1 : 0] result;

        input [3:0] operation;
        input [DATA_WIDTH-1 : 0] a;
        input [DATA_WIDTH-1 : 0] b;

        reg [DATA_WIDTH-1 : 0] result;

        assign zero = (result == 32'd0) ? 1'b1 : 1'b0;
        assign signBit = result[DATA_WIDTH-1];

  always @(operation or a or b) begin

    casex (operation)

      4'b0000: result = a & b;
      4'b0001: result = a | b;
      4'b0010: result = a + b;
      4'b0110: result = a - b;
      default: result = 0;

    endcase

        end

                /*
    // Function to signed operation
    function [31:0] OUT;
       input [31:0] a, b;
       begin
                                casex(b[31])
                                        1'b1:   begin
                                                                b = ~b;
```

```
                                                    b = b + 1'b1;
                                                    OUT = a - b;
                                                    end
                                default: OUT = a + b;
                            endcase
        end
    endfunction
                */

endmodule
```

ALU là một thành phần cực kì quan trọng trong bất cứ kiến trúc máy tính nào, trong RISC-V cũng không ngoại lệ. Module này thực hiện tính toán các toán tử số học và logic: '&', '|', '+', '-'.



Hình 18 - Kết quả RTL viewer module ALU

Bảng 8 - module adder

```
module adder (out,
                    a,
                    b
                    );

          parameter DATA_WIDTH = 32;

          output [DATA_WIDTH-1 : 0] out;

          input [DATA_WIDTH-1 : 0] a;
          input [DATA_WIDTH-1 : 0] b;

   assign out = a + b;

          /*
   function [31:0] OUT;
      input [31:0] a, b;
      begin
      casex(b[31])
         1'b1:   begin
                b = ~b;
                b = b + 1'b1;
                OUT = a - b;
                end
        default: OUT = a + b;
      endcase
    end
   endfunction
          */

endmodule
```

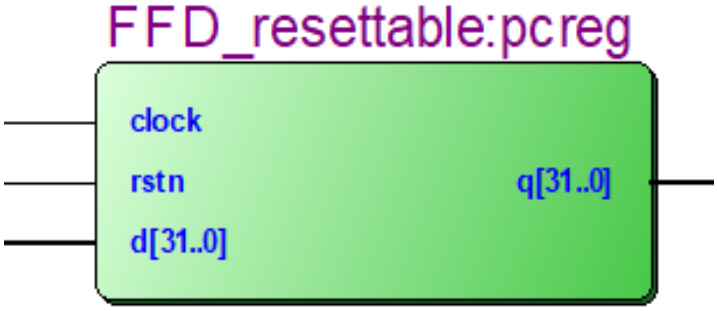
Module này thực hiện tình tổng hai số đầu vào. Phục vụ cho việc tính tổng PC+4, PC+Imm.

Bảng 9 - module FFD_resettable

```
module FFD_resettable(q,
                      clock,
                      rstn,
                      d
                      );

   parameter WIDTH = 32;

   output [WIDTH-1:0] q;
```

```
  input  [WIDTH-1:0] d;
  input clock, rstn;

  reg [WIDTH-1:0] register;

  always @(posedge clock) begin

                if (!rstn)
                        register <= 0;

    else
                        register <= d;

        end

        assign q = register;

endmodule
```

Module này thực hiện ghi giá tiếp theo cho thanh ghi PC, reset thanh ghi PC về 0 khi có tín hiệu reset (reset khi bắt đầu thực hiện chương trình).

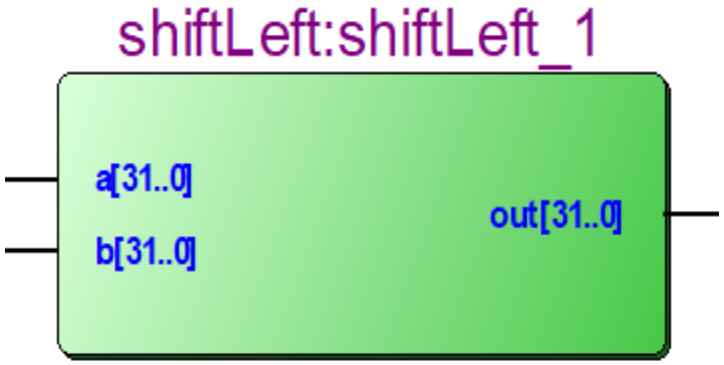


Hình 19 - Kết quả RTL viewer module FFD_resettable

Bảng 10 - module shiftLeft

```
module shiftLeft(out,
                 a,
                 b
                 );

        parameter DATA_WIDTH = 32;

        output [DATA_WIDTH-1 : 0] out;

        input [DATA_WIDTH-1 : 0] a;
        input [DATA_WIDTH-1 : 0] b;

        assign out = a << b;

endmodule
```

Module này thực hiện dịch trái số tức thì để đáp ứng cho việc tính toán số imm, địa chỉ và thực hiện lệnh lui. Module này có thể nạp chồng parameter để thực hiện shift theo số bits mong muốn.

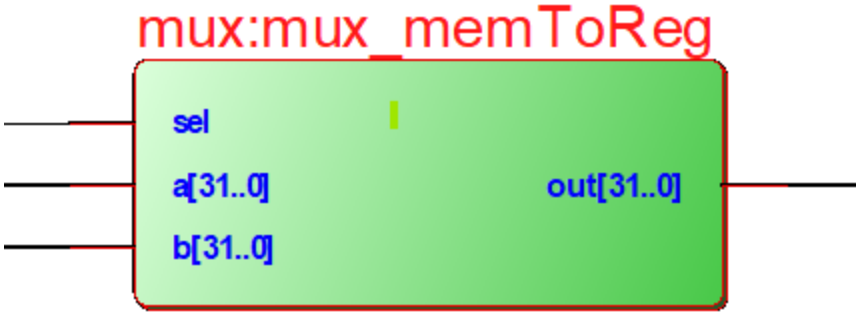


Hình 20 - Kết quả RTL viewer shiftLeft

Bảng 11 - module mux

```
module mux (out,
            sel,
            a,
            b
                        );

    parameter DATA_WIDTH = 32;

    output [DATA_WIDTH-1 : 0] out;

    input sel;
    input [DATA_WIDTH-1 : 0] a;
    input [DATA_WIDTH-1 : 0] b;

    assign out = (sel) ? b : a;

endmodule
```

Mux (2-to-1): Ta thiết kế mux 2-to-1 để lựa chọn các tín hiệu đầu vào cho: Thanh ghi PC; Ngõ vào B của ALU; Dữ liệu Write Back.

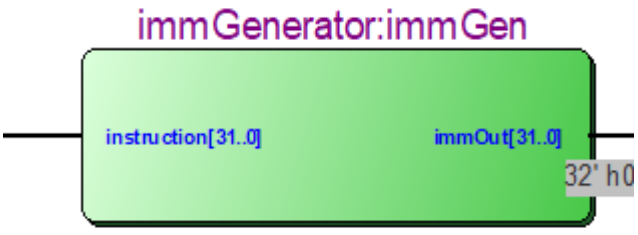


Hình 21 - Kết quả RTL viewer module mux

Bảng 12 - module immGenerator

```
module immGenerator (immOut,
                     instruction
                     );

  output[31:0] immOut;
  input [31:0] instruction;
   reg[31:0] immOut;
  always @(instruction) begin //#0.1
        case(instruction[6:0])

        // ADDI, ANDI, ORI     -> I-Type
   7'b0010011: immOut = { {20{instruction[31]}}, instruction[31:20] };
        // LW       -> I-Type
   7'b0000011: immOut = { {20{instruction[31]}}, instruction[31:20] };
        // SW       -> S-Type
   7'b0100011: immOut = { {20{instruction[31]}}, instruction[31:25], instruction[11:7] };
        // BEQ, BNE -> B-Type
   7'b1100011: immOut = { {20{instruction[31]}}, instruction[31], instruction[7], instruction[30:25], instruction[11:8]};
        // LUI              -> U-Type
   7'b0110111: immOut = { {12{instruction[31]}}, instruction[31:12] };
        // JAL -> J-Type
   7'b1101111: immOut ={{12{instruction[31]}}, instruction[31], instruction[19:12], instruction[20], instruction[30:21]};

        default: immOut <= 32'd0;
        endcase

        end

endmodule
```

Module này thực hiện tách số tức thì từ 32 bits mã máy đầu vào theo kiến trúc RISC-V để thực hiện tính toán khi thực hiện lệnh.



Hình 22 - Kết quả RTL viewer module immGenerator

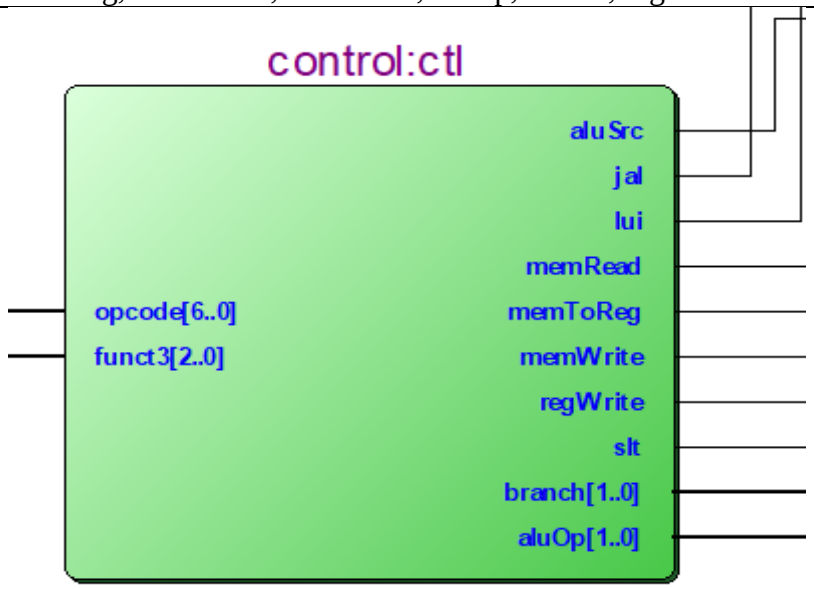
II.2.4 Các thành phần tạo thành module Controller:

Bảng 13 - module control

```
module control (branch,
                slt,
                lui,
                jal,
                memToReg,
                memWrite,
                memRead,
                aluOp,
                aluSrc,
                regWrite,
                opcode,
                funct3
                                        );

        output [1:0] branch;
        output slt;
        output lui;
        output jal;
        output memToReg;
        output memWrite;
        output memRead;
        output [1:0] aluOp;
        output aluSrc;
        output regWrite;

        input [6:0] opcode;
        input [2:0] funct3;

        reg [11:0] controlSignals;

        always @(opcode or funct3) begin

                case(opcode)

                  7'b0110011: case(funct3)
                                3'b010: controlSignals = 12'b001000001001;  // SLT
                                default: controlSignals = 12'b000000001001;  // ADD, SUB, OR, AND
                                endcase

                  7'b0000011: controlSignals = 12'b000001010011;  // LW

                  7'b0010011: controlSignals = 12'b000000001111;  // ADDI, ORI, ANDI
```

```
                7'b0100011: controlSignals = 12'b000000100010;  // SW

                7'b1100011: case(funct3)
                                3'b000: controlSignals = 12'b100000000100;  // BEQ

                                default: controlSignals = 12'b010000000100;  // BNE
                                endcase
                7'b0110111: controlSignals = 12'b00010000xx01;  // LUI

                7'b1101111: controlSignals = 12'b00001000xx01;  // JAL

                default: controlSignals = 12'bxxxxxxxxxxxx; // illegal op
              endcase
        end

        assign {branch, slt, lui, jal, memToReg, memWrite, memRead, aluOp, aluSrc, regWrite} =
controlSignals;

endmodule
```

Module này dựa vào opcode và funct3 để giải mã thành các tín hiệu điều khiển datapath như: branch, slt, lui, jal, memToReg, memWrite, memRead, aluOp, aluSrc, regWrite.

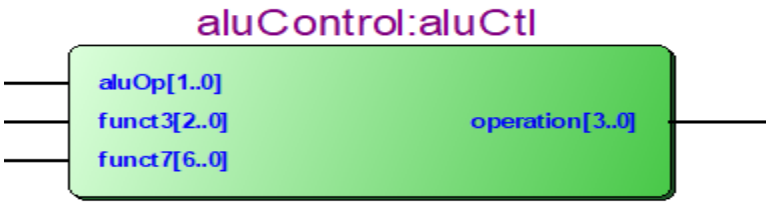


Hình 23 - Kết quả RTL viewer module control

Bảng 14 - module aluControl

```
module aluControl ( operation,
                    aluOp,
                    funct3,
                    funct7
                    );
        output [3:0] operation;
        input [1:0] aluOp;
        input [6:0] funct7;
        input [2:0] funct3;
        reg [3:0] operation;
        always @(aluOp or funct7 or funct3)
            case(aluOp)
                2'b00: operation = 4'b0010; //add (for lw/sw)
                2'b01: operation = 4'b0110; //sub (for beq, bne)
                // R-type instructions
                2'b10: casex(funct7)
                    7'b0000000: casex(funct3)
                                    3'b000: operation = 4'b0010; //add (for add)
                                    3'b010: operation = 4'b0110; //slt (for slt)
                                    3'b110: operation = 4'b0001; //or (for or)
                                    3'b111: operation = 4'b0000; //and (for and)
                                    default: operation = 4'bxxxx;
                                    endcase
                    7'b0100000: operation = 4'b0110; //sub (for sub)
                    default: operation = 4'bxxxx;  //illegal operation
                            endcase
                2'b11: casex(funct3)
                    3'b000: operation = 4'b0010; //add (for addi)
                    3'b110: operation = 4'b0001; //or (for ori)
                    3'b111: operation = 4'b0000; //and (for andi)
                    default: operation = 4'bxxxx;
                    endcase
                default: operation = 4'bxxxx;
            endcase
endmodule
```

Module này thực hiện giải mã aluOp, funct3 và funct7 thành tín hiệu điều khiển ALU thực hiện theo phép tính mong muốn.

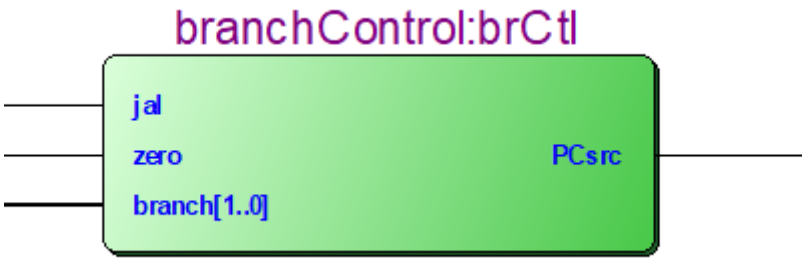


Hình 24 - Kết quả RTL viewer module aluControl

Bảng 15 - module branchControl

```
module branchControl (PCsrc,
                      branch,
                      jal,
                      zero
                      );

    output PCsrc;

    input [1:0] branch;
    input jal;
    input zero;

    assign PCsrc = ((branch[1] && zero) | (branch[0] && (!zero)) | (jal)) === 1 ? 1'b1 : 1'b0;

endmodule
```

Thực hiện nhận tính hiệu ‘branch’, tín hiệu ‘jal’ và tín hiệu ‘zero’ để kiểm tra và đưa ra quyết định giá trị tiếp theo của thanh ghi PC: PC+4 (không nhảy) và PC+Imm (nhảy).



Hình 25 - Kết quả RTL viewer module branchControl

III. THỰC THI GIẢI THUẬT TÌM N (N≤47) SỐ ĐẦU TIÊU TRONG DÃY SỐ FIBONACCI:

III.1 Chương trình “Assembler”:

Chương trình ‘assembler’ do nhóm thực thi dựa trên ngôn ngữ lập trình Python, có chức năng chuyển đổi câu lệnh RISC-V sang mã máy. Với tính năng quan trọng đó ‘assembler’ giúp cho việc tìm hiểu nghiên cứu về vi xử lí RISC-V được dễ dàng hơn, tiếp cận được nhiều người dùng hơn. Trong phạm vi đề tài, chương trình ‘assembler’ đóng vai trò quan trọng trong việc thực thi giải thuật, nó giúp rút ngắn thời gian chuyển đổi từ hợp ngữ sang mã máy một cách đáng kể, tạo thuận lợi cho việc thực thi, kiểm tra nhiều lần giải thuật. Nhằm mục đích có thể mở rộng tập lệnh của vi xử lý do nhóm thực hiện, ‘assembler’ có khả năng chuyển đổi toàn bộ lệnh trong tập lệnh RV32I sang mã máy. Cụ thể hơn, chương trình ‘assembler’ nhận đầu vào là chương trình RISC-V từ file 'Code_editor.txt' (bao gồm cả câu lệnh, nhãn, ghi chú,..) sau quá trinh xử lí như cắt nhãn, xóa ghi chú,... kết quả đầu ra của chương trình ‘assembler’ sẽ là mã máy của các câu lệnh trong file 'Code_editor.txt' đúng theo thứ tự từ trên xuống, xuất ra file 'Machine_code.txt'. Đoạn chương trình chính trong chương trình ‘assembler’:

Bảng 16 - Chương trình 'Assembler'

```
def assembler ():
    fi= open('Code_editor.txt', 'r')
    string= fi.read()
    fi.close()
    fo= open('Machine_code.txt', 'w')

    result = ''
    ins = string.split('\n')
    PC = 0
    pos = 0
    while pos <(len(ins))-1:
        ins[pos]= handler_string(ins[pos])
        if ins[pos] =='.text':
            break
        pos+=1

    for i in range (pos+1,len(ins)) :
        ins[i]= handler_string(ins[i])
        if ins[i] == ' ' or ins[i]=='' or ins[i]=='\n':
            continue

        li = ins[i].split()
        if len(li) ==1 :
            while ins[i].find(':') != -1 :
                address[ins[i][:ins[i].find(':')]] = PC
                ins[i] = ins[i][ins[i].find(':')+1:]
```

```
        else :
            if ins[i].find(':') != -1 :
                label = ins[i] [:ins[i].find (':')].strip()
                instruction = ins[i] [ins[i].find (':')+1:].strip()
                address [label] = PC
                address [instruction] = PC
                ins[i]=instruction
                PC+=4
            else:
                ins[i]+=" "+ str (i)
                address[ins[i]] = PC
                PC+=4

    for i in range (pos+1,len(ins)) :
        ins[i]=handler_string(ins[i])
        t = ins[i].split()
        if len(t) <2:
            continue
        if FMT[t[0]] == "R":
            string= RType(ins[i])
        if FMT[t[0]] == "I":
            string= IType(ins[i])
        if FMT[t[0]] == "S":
            string= SType(ins[i])
        if FMT[t[0]] == "SB":
            string= SBType(ins[i])
        if FMT[t[0]] == "U":
            string= UType(ins[i])
        if FMT[t[0]] == "UJ":
            string= UJType(ins[i])
        result += (string + '\n')
    fo.write(result)
    fo.close()
```

Vì lý do độ dài của chương trình, không phù hợp để trình bày toàn bộ trong flie báo cáo, để xem toàn bộ chương trình truy cập vào đường link sau:
https://drive.google.com/drive/folders/1VR79akv9wMYvzQLR6eHl2J1jevx7WnLF?usp=sharing

Hình 26 - Mô tả chức năng của assembler

III.2 Thực thi giải thuật:

Fibonacci là dãy số tự nhiên vô tận được thiết lập theo quy tắc mỗi phần tử có giá trị bằng tổng hai phần liền trước đó. Dãy số Fibonacci xuất hiện và được nghiên cứu trong nhiều lĩnh vực khoa học tự nhiên, bài toán tìm dãy số fibonacci với số lượng phần tử cụ thể là một trong những bài lập trình kinh điển. Với tính ứng dụng và phổ biến nêu trên của dãy Fibonacci, cũng chính là lí do nhóm quyết định thực hiện giải thuật tìm 47 số đầu tiên trong dãy Fibonacci (có thể tùy chỉnh để tìm dãy có số phần tử bé hơn 47), trên vi xử lý RISC-V do nhóm thiết kế.

Đầu tiên, để thực hiện giải thuật cần xác định lưu đồ thuật toán cho giải thuật, từ đó viết chương trình hợp ngữ phù hợp. Chương trình hợp ngữ do nhóm thực hiện có nội dung như sau:

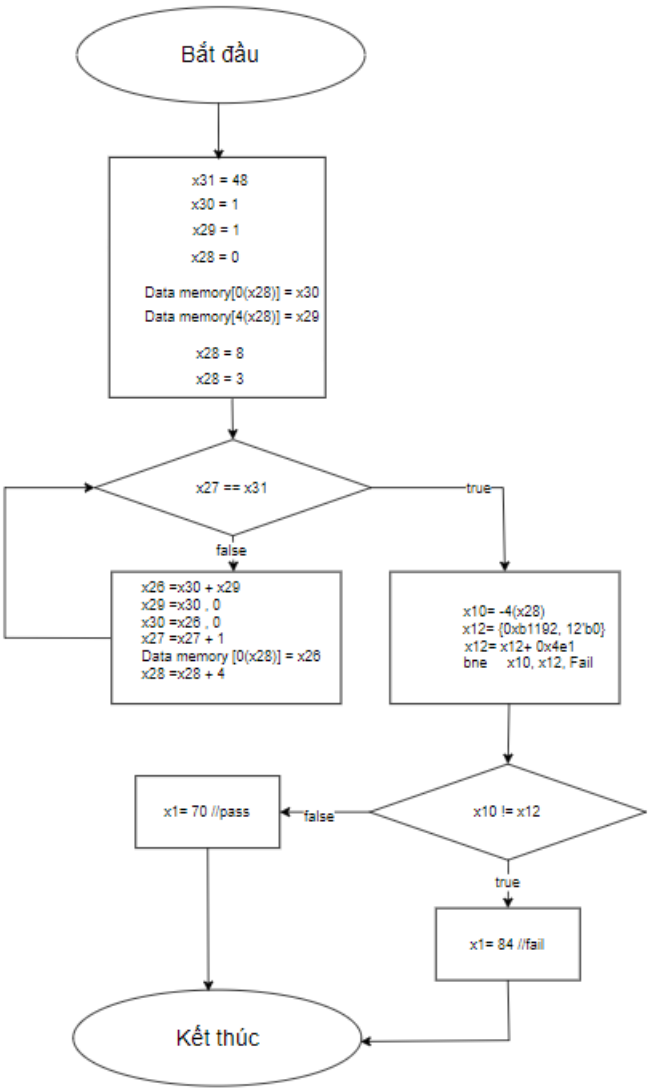
Bảng 17 - Chương trình hợp ngữ tìm 47 phần tử đầu tiên trong dãy Fibonacci

```
.text
#Program to find the first 47 values of fibonacci
Init:
        addi    x31 , x0 , 48
        addi    x30 , x0 , 1
        addi    x29 , x0 , 1
        sw      x30 , 0(x28)
        sw      x29 , 4(x28)
        addi    x28 , x0 , 8
        addi    x27 , x0 , 3
Fibo:
        beq     x27 , x31 , Check
        add     x26 , x30 , x29
        addi    x29 , x30 , 0
        addi    x30  , x26 , 0
        addi    x27  , x27 , 1
        sw      x26  , 0(x28)
        addi    x28  , x28 , 4
```

```
                jal     x0    , Fibo
Check:
                lw      x10 ,  -4(x28)
                lui     x12 , 0xb1192
                addi    x12 ,  x12,  0x4e1
                bne     x10 ,  x12,  Fail
Pass:
                addi    x1,  x0, 70
                jal     x0,  End
Fail:
                addi    x1,  x0,  84
                jal     x0,  End
End:
```

Trong đoạn chương trình trên, 2 thanh ghi x30 và x29 là các thanh ghi chứa giá trị 2 số Fibonacci liền trước đó, x26 mang giá trị số Fibonacci hiện tại. Thanh ghi x31 mang giá trị của số lần lặp trừ đi 1, x27 sẽ là biến đếm số lần lặp, x28 là con trỏ để trỏ đến các ô nhớ trong Data memory. Sau mỗi vòng lặp thanh ghi x30 sẽ được gán giá trị trong thanh ghi x26, x29 được gán giá trị của x30 để cập nhật 2 số liền trước trong dãy Fibonacci, x26 sẽ là tổng của x30 và x29 để tìm số kế tiếp trong dãy, lưu lại giá trị đó vào Data memory thông qua con trỏ là thanh ghi x28. Các câu lệnh trong nhãn 'Check' dùng để kiểm tra giá trị của phần tử thứ 47, bằng cách kết hợp lệnh 'lui' và 'addi' ta có thể gán giá trị của phần thứ 47 trong dãy Fibonacci cho thanh ghi x12, kiểm tra giá trị thứ 47 trong Data memory với giá trị trong thanh ghi x12, nếu không bằng nhau nhảy đến nhãn 'Fail' gán giá trị 84 cho thanh ghi x1 rồi kết thúc chương trình, ngược lại nhảy đến nhãn 'Pass' gán giá trị 70 cho thanh ghi x1 rồi kết thúc chương trình.

Chương trình hợp ngữ tìm 47 giá trị đầu tiên của dãy fibonacci sẽ được lưu trong 47 giá trị đầu tiên trong Data memory theo đúng thứ tự tương ứng. Đoạn lệnh phía sau nhãn 'Check' dùng để kiểm tra giá trị cuối cùng trong Data memory, nếu bằng với giá trị phần tử thứ 47 thì nhảy đến nhãn 'Pass' gán giá trị x1 bằng 70 sau đó nhảy đến 'End' để kết thúc chương trình, nếu không đúng thì nhảy đến nhãn 'Fail' gán giá trị cho thanh ghi x1 bằng 84 sau đó nhảy đến 'End' để kết thúc chương trình. Việc gán giá trị cho thanh ghi x1 để kiểm tra chương trình thực thi đúng hay sai sau khi kết thúc.



Hình 27 - Lưu đồ thuật toán tìm 47 phần tử đầu tiên trong dãy Fibonacci

Sau khi xác định được chương trình hợp ngữ với giải thuật chính xác, việc kế tiếp là chuyển chương trình hợp ngữ sang mã máy thông qua chương trình 'Assembler' được trình bày như trên. Chương trình 'Assembler' trả về kết quả mã máy như sau:

Bảng 18 - Mã máy chương trình hợp ngữ tìm 47 phần tử đầu tiên trong dãy Fibonacci

```
00000011000000000000111110010011
00000000000100000000111100010011
00000000000100000000111010010011
00000001111011100010000000100011
00000001110111100010001000100011
00000000100000000000111000010011
00000000001100000000110110010011
00000011111111011000000001100011
```

```
00000001110111110000110100110011
00000000000011110000111010010011
00000000000011010000111100010011
00000000000111011000110110010011
00000001101011100010000000100011
00000000010011100000111000010011
11111110010111111111000001101111
11111111110011100010010100000011
10110001000110010010011000110111
01001110000101100000011000010011
00000000110001010001011001100011
00001000110000000000000010010011
00000000110000000000000001101111
00000101010000000000000010010011
00000000010000000000000001101111
```

III.3 Testbench cho thiết kế Verilog:

Các lệnh trong chương trình hợp ngữ tìm N (N≤ 47) dựa theo chương trình hợp ngữ nêu trong phần III.2 tuy chưa đầy đủ toàn bộ 14 mà vi xử lý RISC-V trong phạm vi đề tài có thể thực hiện, nhưng hoàn toàn có thể thể hiện đầy đủ hành vi có thể có của mạch (ví dụ như các lệnh R-type có hành vi mạch giống nhau, I-type ngoài lw cũng có hành vi giống nhau). Do đó có thể kết hợp giữa việc kiểm tra giải thuật với kiểm tra tính năng của mạch thông qua một module testbench.

Kiểm tra thiết kế tìm 25 số fibonacci đầu tiên:

Bảng 19 - Testbench kiểm tra thiết tìm 25 số fibonacci đầu tiên

```
module risc_v_testbench2();

parameter PATH_IMEM ="D:/Final_Project/Risc_V_Verilog/testcase/iMem2.bin";
parameter PATH_REGFILE ="D:/Final_Project/Risc_V_Verilog/testcase/regFile.bin";
parameter PATH_DMEM  ="D:/Final_Project/Risc_V_Verilog/testcase/dMem.bin";

	reg clock;
	reg rstn;

	initial begin
		{clock, rstn} = 0;					// reset and prepare for
starting
		repeat(2) @(posedge clock);
		rstn = 1;

		repeat(200) @(posedge clock);			// calculate 25th fibonacci

		if(risc_v_core.risc_v_datapath.regfile.regfile[10] == 32'd75025)
			$display("PASSED");
```

```
            else
                $display("FAILED");

            $stop;
        end

        always #5 clock = ~clock;

        RISC_V #(.PATH_IMEM(PATH_IMEM),
                 .PATH_REGFILE(PATH_REGFILE),
                 .PATH_DMEM(PATH_DMEM)) risc_v_core (.clock(clock),

                .rstn(rstn));

        initial begin
            //$monitor("[%2t]   %d,    %d", $time, risc_v_core.risc_v_datapath.pc,
risc_v_core.risc_v_controller.branch, risc_v_core.risc_v_controller.jal,
risc_v_core.risc_v_datapath.zero, risc_v_core.risc_v_controller.PCsrc);
            $monitor("[%2t],     %d", $time, risc_v_core.risc_v_datapath.writeData);
        end

endmodule
```

Module này thực hiện truyền parameter chứa đường dẫn tới file chứa mã máy là "iMem.bin". Module này thực hiện tính toán toán số Fibonacci 25th và thực hiện so sánh với kết quả mong đợi (Fibonacci 25th = 75025). Nếu kết quả tính được đúng với mong đợi, sẽ gán x1 = 70, ngược lại gán x1 = 84 và kết thúc.

Sau khi thực hiện xong chương trình, kiểm tra giá trị 25 phần tử đầu tiên trong Data memory cũng chính là 25 phần tử đầu tiên của dãy Fibonacci.

```
// memory data file (do not edit the following line - required for mem load use)
// instance=/risc_v_testbench2/risc_v_core/risc_v_datapath/dmem/dmem
// format=bin addressradix=h dataradix=b version=1.0 wordsperline=1 noaddress
00000000000000000000000000000001
00000000000000000000000000000001
00000000000000000000000000000010
00000000000000000000000000000011
00000000000000000000000000000101
00000000000000000000000000001000
00000000000000000000000000001101
00000000000000000000000000010101
00000000000000000000000000100010
00000000000000000000000000110111
00000000000000000000000001011001
00000000000000000000000010010000
00000000000000000000000011101001
00000000000000000000000101111001
00000000000000000000001001100010
00000000000000000000001111011011
00000000000000000000011000111101
00000000000000000000101000011000
00000000000000000001000001010101
00000000000000000001101001101101
00000000000000000010101011000010
00000000000000000100010100101111
00000000000000000110111111110001
00000000000000001011010100100000
00000000000000001001001010010001
```

Hình 28 - Kết quả kiểm tra giá trị 25 phần tử đầu tiên trong Data memory

Kiểm tra thiết kế với việc tìm 47 số Fibonacci đầu tiên:

Tiếp tục kiểm tra với dãy gồm nhiều phần tử hơn với dãy gồm 47 phần tử, giải thuật chỉ có thể thực hiện tìm tối đa 47 phần tử đầu tiên trong dãy Fibonacci vì lí do kể từ phần tử thứ 48 trở đi sẽ có giá trị với độ rộng lớn hơn 32 bit, vượt quá khả năng lưu trữ của một thanh ghi cơ bản trong RISC-V.

Bảng 20 - Testbench kiểm tra thiết tìm 47 số fibonacci đầu tiên

```
module risc_v_testbench1();

	parameter PATH_IMEM = "D:/Final_Project/Risc_V_Verilog/testcase/iMem.bin";
	parameter PATH_REGFILE = "D:/Final_Project/Risc_V_Verilog/testcase/regFile.bin";
	parameter PATH_DMEM = "D:/Final_Project/Risc_V_Verilog/testcase/dMem.bin";

	reg clock;
	reg rstn;

	initial begin
		{clock, rstn} = 0;                                          // reset and prepare for starting
```

```
		repeat(2) @(posedge clock);
		rstn = 1;

		repeat(376) @(posedge clock);				// calculate 47th fibonacci

		if(risc_v_core.risc_v_datapath.regfile.regfile[10] == 32'd2971215073)
			$display("PASSED");
		else
			$display("FAILED");

		$stop;
	end

	always #5 clock = ~clock;

	RISC_V #(.PATH_IMEM(PATH_IMEM),
				.PATH_REGFILE(PATH_REGFILE),
				.PATH_DMEM(PATH_DMEM)) risc_v_core (.clock(clock),

		.rstn(rstn));

	initial begin
		//$monitor("[%2t],	%d,	%d", $time, risc_v_core.risc_v_datapath.pc,
risc_v_core.risc_v_controller.branch, risc_v_core.risc_v_controller.jal, risc_v_core.risc_v_datapath.zero,
risc_v_core.risc_v_controller.PCsrc);
		$monitor("[%2t],	%d", $time, risc_v_core.risc_v_datapath.writeData);
	end

endmodule
```

Module này thực hiện truyền parameter chứa đường dẫn tới file chứa mã máy là "iMem2.bin". Module này thực hiện tính toán toán số Fibonacci 47th và thực hiện so sánh với kết quả mong đợi (Fibonacci 47th = 2,971,215,073). Nếu kết quả tính được đúng với mong đợi, sẽ gán x1 = 70, ngược lại gán x1 = 84 và kết thúc.

Sau khi thực hiện xong chương trình, kiểm tra giá trị 47 phần tử đầu tiên trong Data memory cũng chính là 47 phần tử đầu tiên của dãy Fibonacci.

Bảng 21 - Kết quả kiểm tra giá trị 47 phần tử đầu tiên trong Data memory

```
// memory data file (do not edit the following line - required for mem load use)
// instance=/risc_v_testbench1/risc_v_core/risc_v_datapath/dmem/dmem
// format=bin addressradix=h dataradix=b version=1.0 wordsperline=1 noaddress
00000000000000000000000000000001
00000000000000000000000000000001
00000000000000000000000000000010
00000000000000000000000000000011
00000000000000000000000000000101
00000000000000000000000000001000
```

00000000000000000000000000001101
00000000000000000000000000010101
00000000000000000000000000100010
00000000000000000000000000110111
00000000000000000000000001011001
00000000000000000000000010010000
00000000000000000000000011101001
00000000000000000000000101111001
00000000000000000000001001100010
00000000000000000000001111011011
00000000000000000000011000111101
00000000000000000000101000011000
00000000000000000001000001010101
00000000000000000001101001101101
00000000000000000010101011000010
00000000000000000100010100101111
00000000000000001101111111110001
00000000000000010110101001000000
00000000000000010010010100010001
00000000000000011101101000110001
00000000000000101111111101000010
00000000000001001101100101110011
00000000000001111101100010110101
00000000000011001011001000101000
00000000000101001000101011011101
00000000001000010011110100000101
00000000011010111000111111100010
00000000101011100000100111100111
00000001000110011001100110011001
00000001110001111010001101100000
00000010111000010011110011111001
00000100101010001110000000101001
00000111100010100001110101000010
00001100001100101111111011001011
00010011101111010001101011011101
00011111111110000000110000111000
00110011101011010011001101001010
01010011100111010100101110111010
01000011101001010011111110000010
01101101011100111110010101011111
10110001000110010010010011100001