* Trong Transcript, cd D:/Test
  + vlib work
* Biên dịch lại: vlog \*.v (Hoặc liệt kê tên file, đảm bảo testbench cuối cùng)
* **Kiểm tra lại dòng $readmemh trong testbench:** Đảm bảo nó chỉ là $readmemh("program.hex", ...); (đường dẫn tương đối). Nếu trước đó bạn đã đổi thành đường dẫn tuyệt đối, hãy đổi lại thành tương đối và biên dịch lại testbench (vlog tb\_riscv\_processor.v).
* Chạy mô phỏng: vsim work.tb\_riscv\_processor
* Thêm (một vài) tín hiệu vào Wave (ví dụ chỉ clk, rst, pc\_current, instruction).
* Chạy: run -all
* run 500 ns

Các khối chính của một bộ xử lý RISC-V 32I **đơn xung nhịp (single-cycle)** đơn giản, bao gồm code Verilog ví dụ, mô tả chức năng, testbench cơ bản và các câu hỏi có thể gặp.

**Tổng Quan Bộ Xử Lý Đơn Xung Nhịp**

Ý tưởng chính là mỗi lệnh được hoàn thành trong **một** chu kỳ clock dài. Chu kỳ clock này phải đủ dài để tín hiệu đi qua đường đi dài nhất trong datapath (thường là lệnh load lw).

* **alu.v**: Chứa module ALU.
* **control\_unit.v**: Chứa module Bộ Điều Khiển.
* **data\_memory.v**: Chứa module Bộ Nhớ Dữ Liệu.
* **immediate\_generator.v**: Chứa module Sinh Giá Trị Tức Thời.
* **instruction\_memory.v**: Chứa module Bộ Nhớ Lệnh.
* **register\_file.v**: Chứa module Khối Thanh Ghi.
* **riscv\_single\_cycle\_processor.v**: Đây là module **Top-Level**, nơi bạn sẽ kết nối tất cả các khối con ở trên lại với nhau.
* **tb\_riscv\_processor.v**: Đây là module **Testbench** dùng để mô phỏng và kiểm tra riscv\_single\_cycle\_processor.
* **program.hex**: File chứa mã máy cho chương trình kiểm thử.

**Trình Bày Chi Tiết Về Các Khối (Dựa trên Code Bạn Cung Cấp):**

***(Code Verilog bạn dán cho từng khối là các ví dụ khá chuẩn và phù hợp với chức năng của chúng, tôi sẽ không lặp lại toàn bộ code ở đây mà chỉ tóm tắt và đưa ra nhận xét)***

1. **alu.v (ALU):**
   * **Chức năng: Thực hiện phép cộng (operand\_a + operand\_b) và phép trừ (operand\_a - operand\_b) dựa trên tín hiệu alu\_op (4 bit).**
   * **Đầu ra: result (32 bit), zero\_flag (1 bit).**
   * **Logic zero\_flag: Được tính bằng (alu\_op == OP\_SUB && result == 32'b0). Cách này hoạt động khi Control Unit đảm bảo gửi OP\_SUB cho ALU khi cần kiểm tra bằng nhau (như trong lệnh BEQ). *Câu hỏi có thể là: Có cách nào khác để tính zero\_flag cho BEQ không? Tại sao cách (operand\_a == operand\_b) lại được đề cập trong comment?***
2. **control\_unit.v (Control Unit):**
   * **Chức năng: Giải mã opcode (và funct3 cho Branch) để tạo các tín hiệu điều khiển (RegWrite, MemRead, MemWrite, MemtoReg, ALUSrc, ALUOp, Branch).**
   * **Các lệnh hỗ trợ (dựa trên case): R-type (ADD - giả định dựa trên OPCODE\_R), I-type ALU (ADDI), Load (LW), Store (SW), Branch (BEQ).**
   * **Logic điều khiển: Tạo tín hiệu đúng cho từng loại lệnh. Ví dụ: LW bật RegWrite, MemRead, MemtoReg và dùng ALU để cộng địa chỉ (ALUOp=ALU\_ADD, ALUSrc=1). BEQ bật Branch, dùng ALU để trừ (ALUOp=ALU\_SUB), và so sánh toán hạng từ thanh ghi (ALUSrc=0). *Câu hỏi có thể là: Giải thích các giá trị tín hiệu điều khiển cho lệnh SW? Tại sao MemtoReg không quan trọng với SW?***
   * **funct7: Được khai báo là input nhưng không được sử dụng trong logic case hiện tại (trừ khi bạn mở rộng cho SUB R-type).**
3. **data\_memory.v (Data Memory):**
   * **Chức năng: Lưu/tải dữ liệu 32-bit.**
   * **Hoạt động: Ghi đồng bộ theo posedge clk nếu mem\_write\_en=1. Đọc không đồng bộ (giá trị read\_data thay đổi ngay khi address hoặc nội dung bộ nhớ thay đổi) nếu mem\_read\_en=1, nếu không thì xuất giá trị 'X'. *Câu hỏi có thể là: Sự khác biệt giữa ghi đồng bộ và đọc không đồng bộ là gì? Hậu quả của việc gán 'X' cho read\_data khi không đọc là gì?***
4. **immediate\_generator.v (Immediate Generator):**
   * **Chức năng: Tạo giá trị tức thời 32-bit đã mở rộng dấu.**
   * **Các loại hỗ trợ: I-type (cho ALU và Load), S-type (cho Store), B-type (cho Branch). Logic trích xuất bit và mở rộng dấu ({{...{instruction[31]}}}) và dịch trái cho B-type (1'b0 ở cuối) trông chính xác. *Câu hỏi có thể là: Giải thích cách tạo immediate cho lệnh loại B? Tại sao cần dịch trái?***
5. **instruction\_memory.v (Instruction Memory):**
   * **Chức năng: Cung cấp lệnh 32-bit dựa trên địa chỉ byte từ PC.**
   * **Hoạt động: Đọc tổ hợp (dùng assign) từ mảng mem. Chia địa chỉ cho 4 (address >> 2). Hoạt động như ROM. *Câu hỏi có thể là: Tại sao cần address >> 2?***
6. **register\_file.v (Register File):**
   * **Chức năng: Lưu 32 thanh ghi, hỗ trợ đọc 2 và ghi 1.**
   * **Hoạt động: Ghi đồng bộ theo posedge clk nếu reg\_write\_en=1 và write\_addr khác 0. Đọc tổ hợp (dùng assign), luôn trả về 0 nếu đọc read\_addr là 0. *Câu hỏi có thể là: Điều gì xảy ra nếu cố gắng ghi vào thanh ghi x0? Tại sao việc đọc lại là tổ hợp?***
7. **riscv\_single\_cycle\_processor.v (Top-Level):**
   * **Chức năng: Kết nối tất cả các module trên. Chứa logic cập nhật PC. Định nghĩa các module phụ trợ pc\_reg, adder, mux2to1 bên trong.**
   * **Kết nối: Các kết nối chính (PC -> IMEM, IMEM -> Control/RegFile/ImmGen, RegFile -> ALU, ImmGen -> ALU MUX, ALU -> DMEM/WB MUX, DMEM -> WB MUX, WB MUX -> RegFile) trông phù hợp với datapath đơn xung nhịp.**
   * **Logic PC: Tính pc\_plus\_4, branch\_target\_addr. Dùng pc\_src = Branch & alu\_zero\_flag; để điều khiển PC\_MUX chọn giữa pc\_plus\_4 và branch\_target\_addr. Logic này đúng cho BEQ.**
   * **Unused Tasks: Phần task fetch;, task decode;... được định nghĩa nhưng không được gọi trong thiết kế single-cycle này. Chúng không ảnh hưởng đến hoạt động nhưng có thể gây nhầm lẫn. *Câu hỏi có thể là: Các task fetch, decode... trong file này có được sử dụng không? Tại sao?***
8. **tb\_riscv\_processor.v (Testbench):**
   * **Chức năng: Tạo môi trường mô phỏng.**
   * **Hoạt động: Tạo clock 10ns, tạo reset (12ns hoặc 15ns), nạp program.hex vào uut.IMEM.mem, chạy mô phỏng trong 150ns sau reset, hiển thị kết quả thanh ghi/bộ nhớ, sử dụng $monitor để theo dõi, và kết thúc bằng $finish. Testbench này đã được chứng minh là hoạt động chính xác qua các lần chạy mô phỏng thành công của bạn. *Câu hỏi có thể là: Giải thích vai trò của từng khối initial trong testbench?***

Testbench của bạn sử dụng nhiều khối initial, mỗi khối đảm nhận một nhiệm vụ cụ thể. Đây chính là sự "phân chia theo khối" rõ ràng:

**Khối 1: Tạo Xung Nhịp (initial begin clk = 0; forever #5 clk = ~clk; end)**

* + **Chức năng:** Chỉ tập trung vào việc tạo ra tín hiệu clk với chu kỳ 10ns. Tín hiệu này là trái tim điều khiển hoạt động tuần tự của bộ xử lý (cập nhật PC, ghi thanh ghi, ghi bộ nhớ). Nó độc lập với các hoạt động khác của testbench.
  + **Liên quan đến "pha":** Clock cần thiết cho *mọi* chu kỳ hoạt động, nơi các pha diễn ra.

**Khối 2: Tạo Reset và Trình Tự Kiểm Thử (initial begin rst = 1; #12; rst = 0; ... #150; ... $finish; end)**

* + **Chức năng:** Khối này điều khiển luồng chính của quá trình mô phỏng:
    1. **Khởi tạo:** Đặt rst lên 1 để đưa bộ xử lý về trạng thái ban đầu.
    2. **Bắt đầu thực thi:** Đưa rst xuống 0 sau 12ns (hoặc 15ns) để bộ xử lý bắt đầu chạy.
    3. **Thời gian chạy:** Dùng #150 để cho bộ xử lý đủ thời gian thực thi chương trình mẫu.
    4. **Kiểm tra cuối cùng:** Sau khi chạy xong, dùng các lệnh $display để **xác minh kết quả cuối cùng** trong các thanh ghi và bộ nhớ dữ liệu. Đây là bước kiểm tra tổng thể quan trọng.
    5. **Kết thúc:** Dùng $finish để dừng mô phỏng.
  + **Liên quan đến "pha":** Khối này kiểm soát *khi nào* các pha bắt đầu (sau reset) và kiểm tra *kết quả tích lũy* của tất cả các pha sau một khoảng thời gian.

**Khối 3: Nạp Bộ Nhớ Lệnh (initial begin $readmemh(...) ... end)**

* + **Chức năng:** Chuẩn bị môi trường thực thi cho bộ xử lý bằng cách nạp mã máy từ file program.hex vào bộ nhớ lệnh (IMEM) của bộ xử lý (uut.IMEM.mem).
  + **Liên quan đến "pha":** Trực tiếp chuẩn bị nội dung cho pha **Fetch**. Nếu không có khối này, bộ xử lý sẽ nạp các lệnh không xác định.

**Khối 4: Giám Sát Tín Hiệu (initial begin #1; $monitor(...) end)**

* + **Chức năng:** Đây là công cụ chính để bạn **quan sát hoạt động bên trong** bộ xử lý một cách liên tục. Nó in ra giá trị của các tín hiệu quan trọng mỗi khi chúng thay đổi.
  + **Làm Rõ Các Pha (Quan Trọng):** Dòng $monitor của bạn đã được thiết kế rất tốt để hiển thị thông tin liên quan đến các pha *khái niệm* của bộ xử lý đơn xung nhịp:
    1. **Time=%0t PC=%h Inst=%h**: Hiển thị thông tin của pha **Fetch** (PC hiện tại và lệnh vừa nạp được).
    2. **| RegWr=%b ALUSrc=%b MemRd=%b MemWr=%b Mem2Reg=%b Branch=%b**: Hiển thị các tín hiệu điều khiển chính được tạo ra trong pha **Decode** (bởi Control Unit).
    3. **| ALURes=%h**: Hiển thị kết quả đầu ra của pha **Execute** (từ ALU).
    4. **(Gián tiếp)** Các tín hiệu MemRd, MemWr cho thấy hoạt động của pha **Memory**.
    5. **WBData=%h**: Hiển thị dữ liệu được chuẩn bị để ghi về thanh ghi trong pha **Writeback**.

**Các Câu Hỏi Ngẫu Nhiên Có Thể Gặp (Dựa trên Code Của Bạn):**

1. **Trong riscv\_single\_cycle\_processor.v, bạn đã định nghĩa module adder. Nó được sử dụng ở những đâu trong thiết kế? (Trả lời: Tính PC+4 và tính địa chỉ đích cho lệnh Branch).**
2. **Giải thích luồng dữ liệu của lệnh ADDI trong thiết kế của bạn, từ Instruction Memory đến Register File Write.**
3. **Tín hiệu ALUSrc điều khiển MUX nào và lựa chọn giữa những gì? Module nào tạo ra tín hiệu này?**
4. **Trong control\_unit.v, tại sao trường hợp default lại quan trọng trong khối always @(\*) và case? Nó giúp tránh vấn đề gì? (Trả lời: Tránh tạo latch, đảm bảo tín hiệu luôn có giá trị xác định).**
5. **Logic tính zero\_flag trong alu.v hiện tại có hoạt động đúng cho lệnh BNE (Branch if Not Equal) không? Nếu muốn hỗ trợ BNE, cần thay đổi gì ở Control Unit và logic chọn PC? (Trả lời: Control Unit vẫn có thể dùng ALU\_SUB, nhưng logic chọn PC cần đảo ngược cờ zero: pc\_src = Branch & ~alu\_zero\_flag; hoặc thêm funct3 vào logic).**
6. **Module immediate\_generator.v của bạn đã xử lý sign-extension như thế nào cho các lệnh loại I?**
7. **Tại sao module instruction\_memory không cần tín hiệu clk?**
8. **Trong testbench, làm thế nào bạn xác minh được rằng lệnh sw đã ghi đúng giá trị vào bộ nhớ? (Trả lời: Dùng $display để kiểm tra uut.DMEM.data\_mem[...] ở cuối).**
9. **Các task được định nghĩa trong riscv\_single\_cycle\_processor.v có được gọi không? Chúng phù hợp hơn cho loại thiết kế nào? (Trả lời: Không được gọi trong single-cycle. Phù hợp hơn cho debugging pipeline hoặc mô tả hành vi).**
10. **Nếu tín hiệu RegWrite luôn bằng 0, điều gì sẽ xảy ra với trạng thái của Register File sau khi chạy chương trình? (Trả lời: Không có thanh ghi nào (trừ x0) được cập nhật giá trị mới).**

| * **Câu hỏi** | * **Trả lời gợi ý** |
| --- | --- |
| * **Tại sao chia task trong testbench?** | * Giúp modular hóa, dễ tái sử dụng và đọc code rõ ràng |
| * **Khi nào dùng disable fork?** | * Khi muốn dừng tất cả các process như clock sau khi kết thúc mô phỏng |
| * **Làm sao để test nhiều chương trình hex khác nhau?** | * Viết task run\_test\_case(string file) rồi gọi với file khác nhau |
| * **Vì sao dùng registers[rd] <= wd; chỉ khi rd != 0?** | * Vì x0 luôn bằng 0 trong RISC-V |