# Lời mở đầu

RISC-V là một kiến trúc vi xử lý hướng RISC (Reduced Instruction Set Computing) mã nguồn mở, được thiết kế để cung cấp một nền tảng linh hoạt cho việc xây dựng CPU (Central Processing Unit).

RSIC-V kế thừa các đặc điểm của RISC, nhưng có một số cải tiến:

* Khả năng mở rộng: RISC-V có một kiến trúc mở cho phép các nhà phát triển thêm các tính năng và chức năng mới.
* Khả năng tùy biến: RISC-V có thể được tùy chỉnh để đáp ứng các yêu cầu cụ thể của các ứng dụng khác nhau.

RISC-V là một dự án mã nguồn mở, có nghĩa là bất kỳ ai cũng có thể sử dụng, sửa đổi và phân phối nó miễn phí. Điều này có một số lợi ích, bao gồm:

* Tính cạnh tranh: Khả năng truy cập miễn phí vào RISC-V đã thúc đẩy sự cạnh tranh giữa các nhà cung cấp, dẫn đến giá cả thấp hơn và hiệu suất cao hơn.
* Tính đổi mới: Khả năng tùy biến của RISC-V đã khuyến khích sự đổi mới, dẫn đến sự phát triển của các ứng dụng mới và sáng tạo.
* Tính khả dụng: RISC-V có sẵn trên nhiều nền tảng khác nhau, bao gồm máy tính, điện thoại thông minh, thiết bị IoT và máy móc công nghiệp.

Mục đích nghiên cứu:

* Tìm hiểu sâu các kiến thức và kỹ năng cần thiết về ngôn ngữ lập trình phần cứng Verilog HDL.
* Hiểu rõ về cách hoạt động của một datapath ứng dụng pipeline đơn giản thiết kế kiến trúc của RSIC-V CPU.
* So sánh và phân tích về ưu và nhược điểm của RISC-V so với các kiến trúc khác, nhấn mạnh vào những điểm mạnh và đặc điểm nổi bật.

# Giới thiệu đề tài

Đề tài môn học Thiết kế hệ thống số với HDL (CE213.O11) của nhóm sẽ trình bày về thiết kế bộ xử lí có ứng dụng cơ chế pipeline theo kiến trúc tập lệnh 32 bit RISC-V.

Thiết kế sử dụng cơ chế pipline 5 giai đoạn như sau:

1. **Instrucion Fetch (Nạp lệnh)**: CPU đọc lệnh từ bộ nhớ chính (RAM) hoặc bộ nhớ cache và lưu trữ lệnh vào một thanh ghi đặc biệt được gọi là "Instruction Register" (IR) theo con trỏ pc (Program counter).
2. **Instruction Decode (Giải mã lệnh)**: lệnh được giải mã để xác định các phép toán cần thực hiện và các toán hạng liên quan.
3. **ALU (Arithmetic Logic Unit - Đơn Vị Toán Học Logic)**: Giai đoạn này thực hiện các phép toán toán học và logic dựa trên thông tin đã giải mã từ giai đoạn trước.
4. **Memmory Access (Truy cập bộ nhớ)**: Nếu lệnh yêu cầu truy cập bộ nhớ (ví dụ: lưu trữ hoặc đọc dữ liệu), giai đoạn này thực hiện hoạt động truy cập bộ nhớ.
5. **Write Back (Ghi trả)**: Giai đoạn cuối cùng là ghi kết quả của phép toán vào thanh ghi hoặc nơi lưu trữ tương ứng.

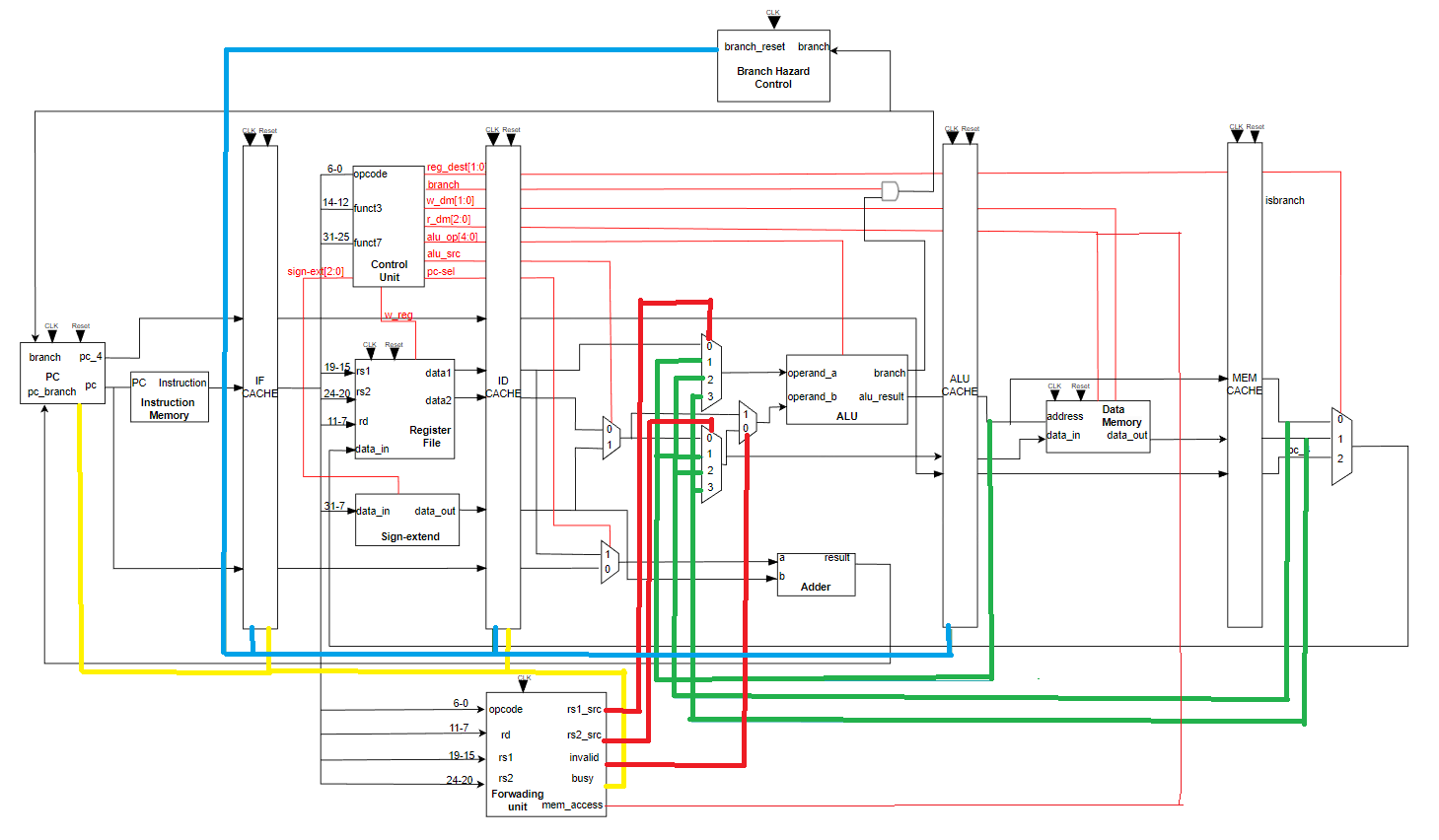


Figure : Pipelined Datapath.

Định dạng mã hóa lệnh (encoding format):

A table with numbers and letters

Description automatically generated

Figure : Format Instruction set.

# Các thành phần chính

1. Thanh ghi PC (Program Counter)

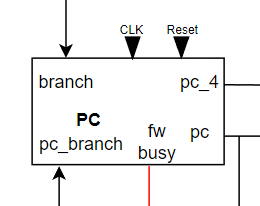


Figure : Thiết kế Program Counter.

- Input:

1. CLK: clock.
2. Reset: Tín hiệu Reset đồng bộ.
3. branch: Tín hiệu cho phép nhảy.
4. pc\_branch: Địa chỉ nhảy của pc.
5. fw\_busy: Tín hiệu xung đột.

- Output:

1. pc: Con trỏ pc hiện tại.
2. pc\_4: Con trỏ pc + 4.

* Lưu địa chỉ lệnh hiện tại ở pc và địa chỉ lệnh tiếp theo là pc\_4 hoặc là pc\_branch nếu thực hiện lệnh nhảy.
* Program counter có tín hiệu Reset bất đồng bộ dùng để khởi tạo giá trị ban đầu cho pc và pc\_4 (pc + 4).
* Khi có tín hiệu xung đột fw\_busy thì pc sẽ được giữ nguyên, không cập nhật pc + 4 mà chỉ cập nhật pc\_branch.
* Program counter sẽ cập nhật pc như sau:

|  |  |  |  |
| --- | --- | --- | --- |
| CLK | branch | fw\_busy | pc |
| Cạnh lên | 0 | 0 | pc\_4 |
| Cạnh lên | 1 | 0 | pc\_branch |
| Cạnh lên | 0 | 1 | pc |
| Cạnh lên | 1 | 1 | pc\_branch |

2. Instruction Memory.

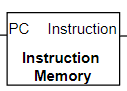


Figure : Thiết kế Instruction Memory.

- Input:

1. PC: địa chỉ lệnh cần đọc và nạp.

- Output:

1. Instruction: Lệnh đọc được tại địa chỉ pc.

3. Register File.

A diagram of a computer

Description automatically generated

Figure : Thiết kế Regiter File.

- Input:

1. rs1[4:0]: Địa chỉ thanh ghi rs1 cần đọc.
2. rs2[4:0]: Địa chỉ thanh ghi rs2 cần đọc.
3. rd[4:0]: Địa chỉ thanh ghi rd được ghi giá trị mới.
4. data\_in[31:0]: Dữ liệu được ghi vào rd.
5. w\_reg: Tín hiệu chi phép ghi từ Controller.
6. clk: clock.
7. Reset: Tín hiệu Reset đồng bộ.

- Output:

1. data1[31:0]: Dữ liệu được đọc từ rs1.
2. data2[31:0: Dữ liệu được đọc từ rs2.

* Register File có thể lưu trữ dữ liệu 32 bits (4 bytes).
* Việc đọc thanh ghi sẽ được diễn ra liên tục.
* Việc ghi thanh ghi sẽ được thực hiện ở clock tích cực mức thấp.
* Register File có tín hiệu Reset bất đồng bộ dùng để khởi tạo giá trị ban đầu cho các thanh ghi trong Register File.

4. Sign extend.

A close-up of a sign

Description automatically generated

Figure : Thiết kế sign-extend

- Input:

1. data\_in[25:0]: Trường IMM cần được mở rộng bit.
2. sign-ext[2:0]: Tín hiệu điều khiển khối.

- Output: data\_out[31:0] là giá trị sau khi mở rộng bit.

* Khối Sign-extend ngoài chức năng mở rộng bit dấu, chúng còn có chức năng sắp xếp thứ tự các bit ở trường IMM thành một giá trị hoàn chỉnh.
* Tín hiệu điều khiển khối Sign extend được quy định thực hiện các chức năng như sau:

|  |  |
| --- | --- |
| sign-ext[2:0] | Chức năng |
| 0 | Mở rộng bit dấu đối với lệnh I |
| 1 | Mở rộng bit không dấu đối với lệnh I. |
| 2 | Mở rộng bit dấu đối với lệnh S. |
| 3 | Mở rộng bit dấu đối với lệnh B. |
| 4 | Mở rộng bit dấu đối với lệnh U. |
| 5 | Mở rộng bit dấu đối với lệnh J. |

Table : sign-ext Op codes.

5. ALU (Arithmetic Logic Unit - Đơn Vị Toán Học Logic)

A diagram of a computer code

Description automatically generated with medium confidence

Figure : Thiết kế ALU.

- Input: data1[31:0], operand\_b[31:0], alu\_op[4:0]

- Output: alu\_result[31:0], isbranch

* Thực hiện tính toán số học hoặc logic giữa 2 toán hạng đầu vào. Đầu ra sẽ là kết quả sau khi thực hiện phép toán và tín hiệu branch nếu thực hiện phép toán cho phép nhảy.
* Nếu đầu vào ALUOP không thuộc bảng ALU Op codes thì kết quả RESULT và BRANCH sẽ mặc định là 0.

|  |  |  |
| --- | --- | --- |
| ALUOP[4:0] | RESULT[31:0] | BRANCH |
| 00000 | ADD | 0 |
| 00001 | SLL | 0 |
| 00010 | SLT | 0 |
| 00011 | SLTU | 0 |
| 00100 | XOR | 0 |
| 00101 | SRL | 0 |
| 00110 | OR | 0 |
| 00111 | AND | 0 |
| 01000 | 0 | BEQ |
| 01001 | 0 | BNE |
| 01010 | 0 | 0 |
| 01011 | 0 | 0 |
| 01100 | 0 | BLT |
| 01101 | 0 | BGE |
| 01110 | 0 | BLTU |
| 01111 | 0 | BGEU |
| 10000 | SUB | 0 |
| 10001 | ADD | 1 |
| 10010 | 0 | 0 |
| 10011 | 0 | 0 |
| 10100 | 0 | 0 |
| 10101 | SRA | 0 |
| 10110 | 0 | 0 |
| 10111 | 0 | 0 |
| 11000 | MUL | 0 |
| 11100 | DIV | 0 |
| 11101 | REM | 0 |
| 11110 | FWD | 0 |
| 11111 | REMU | 0 |

Table : ALU Opcode codes

6. Data Memory.

A diagram of a computer

Description automatically generated

Figure : Khối Data Memory.

- Input:

1. address[31:0]: Địa chỉ tương tác trong data memory.
2. data\_in[31:0]: Giá trị được ghi vào data memory tại địa chỉ address.
3. r\_dm[2:0]: Tín hiệu cho phép đọc giá trị trong data memory tại address.
4. w\_dm[1:0]: Tín hiệu cho phép ghi giá trị vào data memory tại address.

- Output: data\_out[31:0] là giá trị đọc được tại địa chỉ address trong data memory.

* Việc đọc giá trị trong data memory sẽ được thực hiện liên tục.
* Việc ghi giá trị trong data memory sẽ được thực hiện tại clock tích cực mức thấp.
* Data memory có tín hiệu Reset bất đồng bộ dùng để khởi tạo giá trị ban đầu cho bộ nhớ trong data memory.
* Tín hiệu điều khiển cho phép đọc và ghi của data memory được quy định thực hiện các chức năng như sau:

|  |  |
| --- | --- |
| r\_dm[2:0] | Chức năng |
| 0 | Không cho phép đọc |
| 1 | Đọc 1 Byte |
| 2 | Đọc một nửa word (2 bytes) |
| 3 | Đọc một word (4 bytes) |
| 4 | Đọc một byte không dấu |
| 5 | Đọc một nửa word không dấu (4 bytes) |

Table : Read data memory.

|  |  |
| --- | --- |
| w\_dm[1:0] | Chức năng |
| 0 | Lưu 1 byte |
| 1 | Lưu một nửa word (2 bytes) |
| 2 | Lưu một word (4 bytes) |

Table : Write data memory.

7. Control Unit.

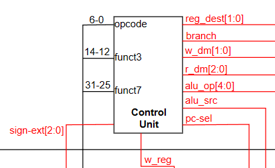


Figure : Thiết kế Control unit.

- Input:

1. opcode[6:0]: Trường opcode của lệnh.
2. funct3[2:0]: Trường funct3 của lệnh.
3. funct7[6:0]: Trường funct7 của lệnh.

- Output:

1. reg\_dest[1:0]: Tín hiệu Register destination.
2. branch : Tín hiệu lệnh nhảy.
3. w\_dn[1:0]: Tín hiệu write data memory.
4. r\_dm[2:0]: Tín hiệu read data memory.
5. alu\_op[4:0]: Tín hiệu ALU Op code.
6. alu\_src : Tín hiệu bộ mux0.
7. pc\_sel : Tín hiệu bộ mux4.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Instruction** | **Type** | **r\_dm** | **w\_dm** | **w\_reg** | **alu\_src** | **Branch** | **reg\_dest** | **sign\_ext** | **pc\_sel** |
| LB | I-Type | **1** | **0** | **1** | **1** | **0** | **1** | **0** | **x** |
| LH | I-Type | **2** | **0** | **1** | **1** | **0** | **1** | **0** | **x** |
| LW | I-Type | **3** | **0** | **1** | **1** | **0** | **1** | **0** | **x** |
| LBU | I-Type | **4** | **0** | **1** | **1** | **0** | **1** | **0** | **x** |
| LHU | I-Type | **5** | **0** | **1** | **1** | **0** | **1** | **0** | **x** |
| ADDI | I-Type | **0** | **0** | **1** | **1** | **0** | **1** | **0** | **x** |
| SLTI | I-Type | **0** | **0** | **1** | **1** | **0** | **1** | **0** | **x** |
| SLTIU | I-Type | **0** | **0** | **1** | **1** | **0** | **1** | **0** | **x** |
| XORI | I-Type | **0** | **0** | **1** | **1** | **0** | **1** | **0** | **x** |
| ORI | I-Type | **0** | **0** | **1** | **1** | **0** | **1** | **0** | **x** |
| ANDI | I-Type | **0** | **0** | **1** | **1** | **0** | **1** | **0** | **x** |
| SLLI | I-Type | **0** | **0** | **1** | **1** | **0** | **1** | **1** | **x** |
| SRLI | I-Type | **0** | **0** | **1** | **1** | **0** | **1** | **1** | **x** |
| SRAI | I-Type | **0** | **0** | **1** | **1** | **0** | **1** | **1** | **x** |
| SB | S-Type | **0** | **1** | **0** | **1** | **0** | **x** | **2** | **x** |
| SH | S-Type | **0** | **2** | **0** | **1** | **0** | **x** | **2** | **x** |
| SW | S-Type | **0** | **3** | **0** | **1** | **0** | **x** | **2** | **x** |
| ADD | R-Type | **0** | **0** | **1** | **0** | **0** | **0** | **x** | **x** |
| SUB | R-Type | **0** | **0** | **1** | **0** | **0** | **0** | **x** | **x** |
| SLL | R-Type | **0** | **0** | **1** | **0** | **0** | **0** | **x** | **x** |
| SLT | R-Type | **0** | **0** | **1** | **0** | **0** | **0** | **x** | **x** |
| SLTU | R-Type | **0** | **0** | **1** | **0** | **0** | **0** | **x** | **x** |
| XOR | R-Type | **0** | **0** | **1** | **0** | **0** | **0** | **x** | **x** |
| SRL | R-Type | **0** | **0** | **1** | **0** | **0** | **0** | **x** | **x** |
| SRA | R-Type | **0** | **0** | **1** | **0** | **0** | **0** | **x** | **x** |
| OR | R-Type | **0** | **0** | **1** | **0** | **0** | **0** | **x** | **x** |
| AND | R-Type | **0** | **0** | **1** | **0** | **0** | **0** | **x** | **x** |
| MUL | R-Type | **0** | **0** | **1** | **0** | **0** | **0** | **x** | **x** |
| DIV | R-Type | **0** | **0** | **1** | **0** | **0** | **0** | **x** | **x** |
| LUI | U-type | **0** | **0** | **1** | **1** | **0** | **0** | **4** | **x** |
| AUIPC | U-type | **0** | **0** | **1** | **x** | **0** | **0** | **4** | **x** |
| JALR | I-Type | **0** | **0** | **1** | **1** | **1** | **3** | **0** | **1** |
| JAL | J-Type | **0** | **0** | **1** | **x** | **1** | **3** | **5** | **0** |
| BEQ | B-Type | **0** | **0** | **0** | **0** | **1** | **x** | **3** | **0** |
| BNE | B-Type | **0** | **0** | **0** | **0** | **1** | **x** | **3** | **0** |
| BLT | B-Type | **0** | **0** | **0** | **0** | **1** | **x** | **3** | **0** |
| BGE | B-Type | **0** | **0** | **0** | **0** | **1** | **x** | **3** | **0** |
| BLTU | B-Type | **0** | **0** | **0** | **0** | **1** | **x** | **3** | **0** |
| BGEU | B-Type | **0** | **0** | **0** | **0** | **1** | **x** | **3** | **0** |

Table : Tín hiệu điều khiển từng lệnh.

8. Thanh ghi Pipeline.

CPU được thiết kế theo cơ chế pipeline, ta sử dụng 4 thanh ghi pipeline để kết nối 5 giai đoạn của bộ xử lý RISC-V.

* Pipeline1 (IF Cache).
* Pipeline2 (ID Cache).
* Pipeline3 (ALU Cache).
* Pipeline4 (MEM Cache).

8.1. Pipeline1 (IF Cache).

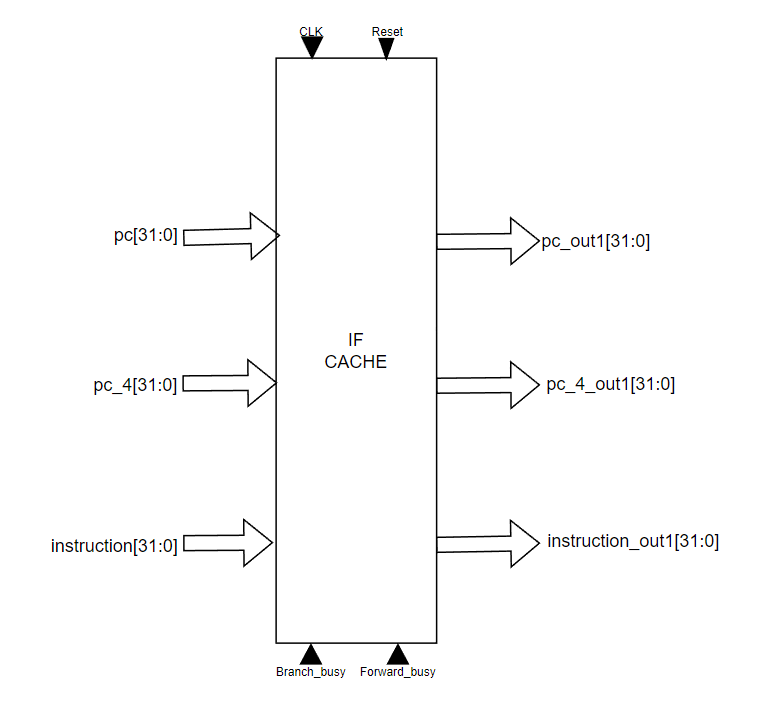


Figure : Pipeline1 (IF Cache).

- Input:

1. pc[31:0].
2. pc\_4[31:0].
3. instruction[31:0].
4. CLK: clock tích cực mức cao.
5. Reset: Tín hiệu reset bất đồng bộ thanh ghi Pipeline.
6. Branch\_busy: Tín hiệu thực hiện NOP lệnh branch đồng bộ với CLK.
7. Forward\_busy: Tín hiệu thực hiện NOP khi Forward đồng bộ với CLK.

- Output:

1. pc\_out1[31:0].
2. pc\_4\_out1[31:0].
3. instruction\_out1[31:0].

* Thanh ghi Pipeline1 (IF cache) nằm ở giữa giai đoạn IF và ID của RSIC-V CPU. Lưu các dữ liệu được truyền từ giai đoạng IF tới giai đoạn ID.
* Tín hiệu Branch\_busy được dùng để giải quyết xung đột điều khiển khi thực hiện các lệnh nhảy trong kiến trúc Pipeline.
* Tín hiệu Forward\_busy được dùng để giải quyết xung đột dữ liệu trong quá trình sử dụng kỹ thuật Forwarding trong kiến trúc Pipeline.
* Tín hiệu Reset bất đồng bộ sẽ mặc định lưu giá trị 0 trong thanh ghi Pipeline.

8.2. Pipeline2 (ID Cache).

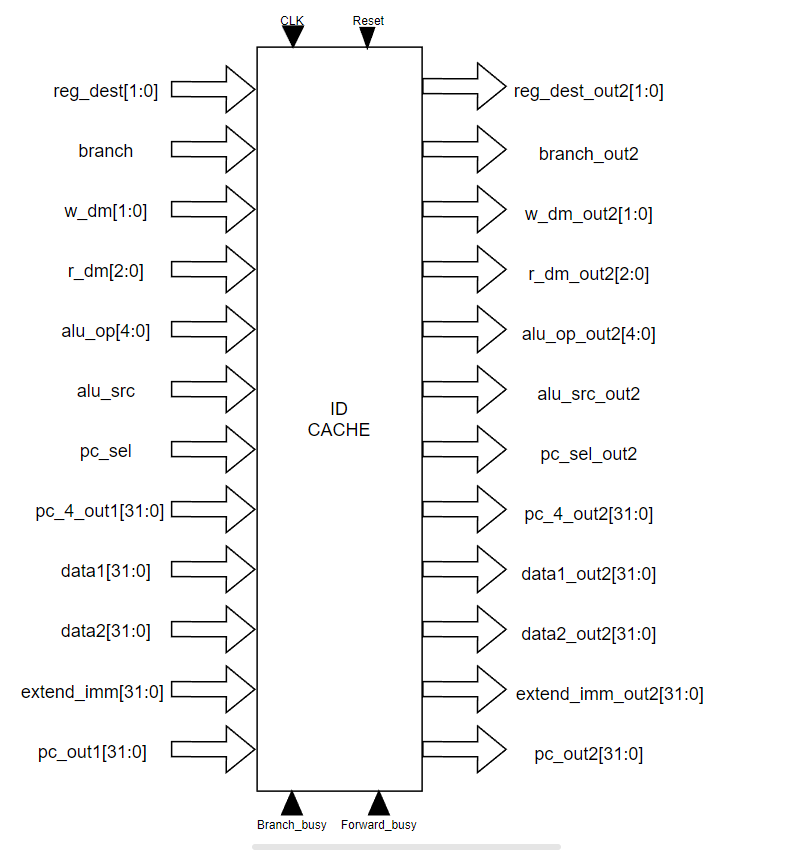


Figure : Pipeline2 (ID Cache).

- Input:

1. reg\_dest[1:0].
2. branch.
3. w\_dm[1:0].
4. r\_dm[2:0].
5. alu\_op[4:0].
6. alu\_src.
7. pc\_sel.
8. pc\_4\_out1[31:0].
9. data1[31:0].
10. data2[31:0].
11. extend\_imm[31:0].
12. pc\_out[31:0].
13. CLK: clock tích cực mức cao.
14. Reset: Tín hiệu reset thanh ghi Pipeline.
15. Branch\_busy: Tín hiệu thực hiện NOP lệnh branch đồng bộ với CLK.
16. Forward\_busy: Tín hiệu thực hiện NOP khi Forward đồng bộ với CLK.

- Output:

1. reg\_dest\_out2[1:0].
2. branch\_out2.
3. w\_dm\_out2[1:0].
4. r\_dm\_out2[2:0].
5. alu\_op\_out2[4:0].
6. alu\_src\_out2.
7. pc\_sel\_out2.
8. pc\_4\_out2[31:0].
9. data1\_out2[31:0].
10. data2\_out2[31:0].
11. extend\_imm\_out2[31:0].
12. pc\_out2[31:0].

* Thanh ghi Pipeline2 (ID cache) nằm ở giữa giai đoạn ID và ALU của RSIC-V CPU. Lưu các dữ liệu được truyền từ giai đoạng ID tới giai đoạn ALU.
* Tín hiệu Branch\_busy được dùng để giải quyết xung đột điều khiển khi thực hiện các lệnh nhảy trong kiến trúc Pipeline.
* Tín hiệu Forward\_busy được dùng để giải quyết xung đột dữ liệu trong quá trình sử dụng kỹ thuật Forwarding trong kiến trúc Pipeline.
* Tín hiệu Reset bất đồng bộ sẽ mặc định lưu giá trị 0 trong thanh ghi Pipeline.

8.3 Pipeline3 (ALU Cache)

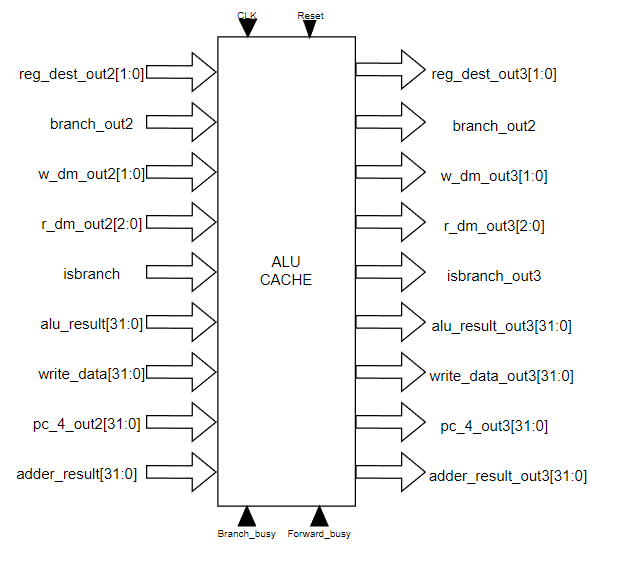


Figure : Pipeline3 (ALU Cache).

- Input:

1. reg\_dest\_out2[1:0].
2. branch\_out2.
3. w\_dm\_out2[1:0].
4. r\_dm\_out2[2:0].
5. isbranch.
6. alu\_result[31:0].
7. write\_data[31:0].
8. pc\_4\_out2[31:0].
9. adder\_result[31:0].
10. CLK: clock tích cực mức cao.
11. Reset: Tín hiệu reset thanh ghi Pipeline.
12. Branch\_busy: Tín hiệu thực hiện NOP lệnh branch đồng bộ với CLK.
13. Forward\_busy: Tín hiệu thực hiện NOP khi Forward đồng bộ với CLK.

- Output :

1. reg\_dest\_out3[1:0].
2. branch\_out3.
3. w\_dm\_out3[1:0].
4. r\_dm\_out3[2:0].
5. isbranch\_out3.
6. alu\_result\_out3 [31:0].
7. write\_data\_out3 [31:0].
8. pc\_4\_out3[31:0].
9. adder\_result\_out3 [31:0].

* Thanh ghi Pipeline3 (ALU cache) nằm ở giữa giai đoạn ALU và MEM của RSIC-V CPU. Lưu các dữ liệu được truyền từ giai đoạng ALU tới giai đoạn MEM.
* Tín hiệu Branch\_busy được dùng để giải quyết xung đột điều khiển khi thực hiện các lệnh nhảy trong kiến trúc Pipeline.
* Tín hiệu Forward\_busy được dùng để giải quyết xung đột dữ liệu trong quá trình sử dụng kỹ thuật Forwarding trong kiến trúc Pipeline.
* Tín hiệu Reset bất đồng bộ sẽ mặc định lưu giá trị 0 trong thanh ghi Pipeline.

8.4. Pipeline4 (MEM Cache).

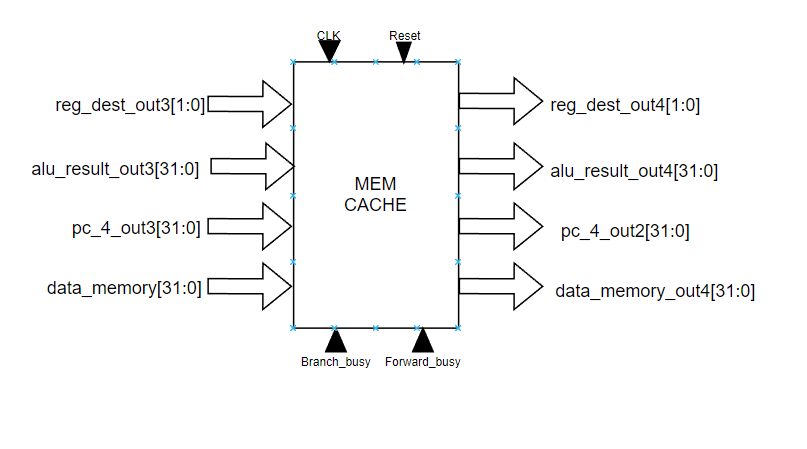


Figure : Pipeline4 (MEM Cache).

- Input:

1. reg\_dest\_out3[1:0].
2. alu\_result\_out3[31:0].
3. pc\_4\_out3[31:0].
4. data\_memory[31:0].
5. CLK: clock tích cực mức cao.
6. Reset: Tín hiệu reset thanh ghi Pipeline.
7. Branch\_busy: Tín hiệu thực hiện NOP lệnh branch đồng bộ với CLK.
8. Forward\_busy: Tín hiệu thực hiện NOP khi Forward đồng bộ với CLK.

- Output:

1. reg\_dest\_out4[1:0].
2. alu\_result\_out4[31:0].
3. pc\_4\_out4[31:0].
4. data\_memory\_out4[31:0].

* Thanh ghi Pipeline4 (MEM cache) nằm ở giữa giai đoạn MEM và WB của RSIC-V CPU. Lưu các dữ liệu được truyền từ giai đoạng MEM tới giai đoạn WB.
* Tín hiệu Branch\_busy được dùng để giải quyết xung đột điều khiển khi thực hiện các lệnh nhảy trong kiến trúc Pipeline.
* Tín hiệu Forward\_busy được dùng để giải quyết xung đột dữ liệu trong quá trình sử dụng kỹ thuật Forwarding trong kiến trúc Pipeline.
* Tín hiệu Reset bất đồng bộ sẽ mặc định lưu giá trị 0 trong thanh ghi Pipeline.

Các thành phần trên đã đủ để RSIC-V CPU áp dụng cơ chế pipeline hoạt động thành công. Tuy nhiên chúng vẫn chưa đủ để giải quyết các vấn đề xung đột trong kiến trúc pipeline.

Để xử lý các xung đột trong kiến trúc pipeline, nhóm đã thiết kế thêm các thành phần nâng cao để giải quyết và được trình bày rõ trong phần xử lý xung đột.

# Xử lý xung đột.

1. Xung đột dữ liệu.

Xung đột dữ liệu trong kiến trúc pipeline thường xuất hiện khi có sự cạnh tranh giữa các thành phần hoặc quy trình khác nhau trong pipeline để truy cập, sửa đổi hoặc sử dụng dữ liệu.

Để xử lý xung đột dữ liệu, nhóm đã áp dụng kỹ thuật forwarding (kỹ thuật nhìn trước) bằng cách thiết kế thêm khối chức năng nâng cao Forwarding unit.

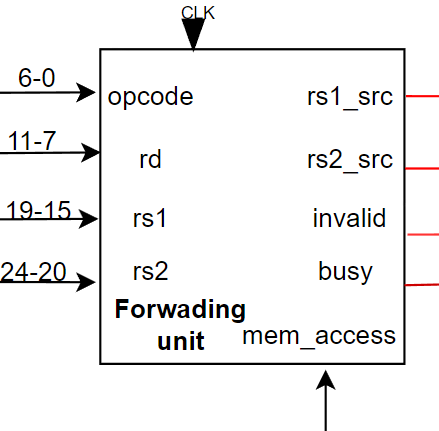


Figure : Thiết kế Forwarding unit.

- Input:

1. opcode[6:0]: Trường opcode của lệnh.
2. rd[4:0]: Trường rd của lệnh.
3. rs1[4:0]: Trường rs1 của lệnh.
4. rs2[4:0]: Trường rs2 của lệnh.
5. mem\_access: Tín hiệu truy cập bộ nhớ
6. CLK: clock tích cực mức cao.

- Output:

1. rs1\_src: Tín hiệu lựa chọn ngõ vào cho toán hạng a của ALU.
2. rs2\_src: Tín hiệu lữa chọn ngõ vào cho toán hạng b của ALU.
3. invalid: Tín hiệu các trường hợp không cần forwarding.
4. busy: Tín hiệu NOP khi forwarding.

* Forwarding unit có chức năng chuyển các giá trị từ giai đoạn MEM và WB để RSIC-V CPU xử lý khi xảy ra xung đột dữ liệu (kỹ thuật nhìn trước).
* Khối được đồng bộ với CPU thông qua tín hiệu CLK.
* Forwarding unit sẽ lấy các giá trị rs1 và rs2 nhận được so sánh với giá trị rd của các lệnh trước đó. Từ đó cho ra các output là tín hiệu rs1\_src và rs2\_src để lựa chọn ngõ vào để tính toán trong công đoạn ALU.
* Tuy nhiên vẫn có các trường hợp ngoại lệ: Khi các lệnh có rs1 và rs2 trùng với rd trước đó nhưng chúng không xảy ra xung đột dữ liệu và không cần forwarding. Ngược lại khi forwarding thì CPU lại tính toán sai. Do đó ta có tín hiệu ngõ ra là invalid để xử lý các trường hợp như vậy.
* Một trường hợp đặc biệt là mặc dù CPU đã áp dung kỹ thuật forwarding (kỹ thuật nhìn trước) mà vẫn không xử lý dược xung đột dữ liệu thì ta có tín hiệu ngõ ra là busy có chức năng thực hiện NOP CPU một chu kỳ.
* Trong khối forwarding unit sẽ có các thanh ghi sau:

1. storage\_1[4:0]: Lưu thanh ghi rd đọc được khi clock tích cực dương.
2. storage\_2[4:0]: Lưu thanh ghi storage\_1 khi clock tích cực dương.
3. storage\_op[6:0]: Lưu opcode đọc được khi clock tích cực dương.

|  |  |  |
| --- | --- | --- |
| Điều kiện | rs1\_src | Giá trị được forward |
| storage\_1 = rs1 | 1 | alu\_result từ giai đoạn MEM |
| storage\_2 = rs1 && !mem\_access (lệnh trước đó không là lw, lh, …) | 2 | alu\_result từ giai đoạn WB |
| storage\_2 = rs1 && mem\_access (lệnh trước đó là lw, lh,…) | 3 | mem\_result từ giai đoạn WB |
| Còn lại | 0 | không forward |

|  |  |  |
| --- | --- | --- |
| Điều kiện | rs2\_src | Giá trị được forward |
| storage\_1 = rs2 | 1 | alu\_result từ giai đoạn MEM |
| storage\_2 = rs1 && !mem\_access (lệnh trước đó không là lw, lh, …) | 2 | alu\_result từ giai đoạn WB |
| storage\_2 = rs1 && mem\_access (lệnh trước đó là lw, lh,…) | 3 | mem\_result từ giai đoạn WB |
| Còn lại | 0 | không forward. |

|  |  |
| --- | --- |
| Điều kiện | invalid |
| storage\_2 = rs2 && opcode là các lệnh không có rs2 như lệnh U, I, J. | 1 |
| Còn lại | 0 |

|  |  |
| --- | --- |
| Điều kiện | busy |
| (storage\_1 = rs1 | | storage\_1 = rs2) && storage\_op = 0000011 (lệnh trước đó là lw, lh,…). | 1 |
| Còn lại | 0 |

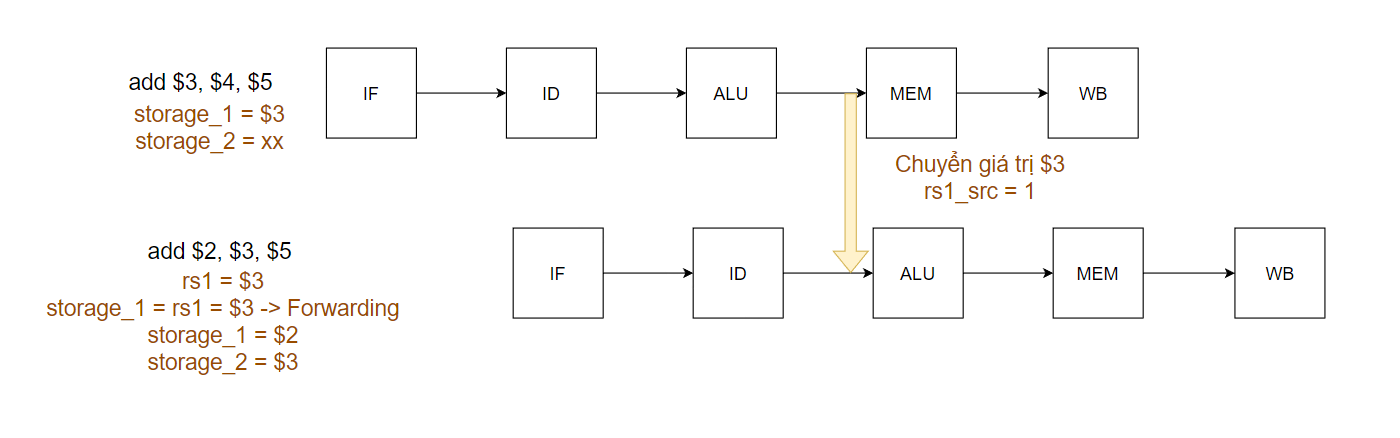
 Từ các điều kiện trên thì sơ đồ dưới đây biểu diễn cho cách forwarding unit xử lý xung đột dữ liệu:

Figure : Chuyển giá trị alu\_result từ giai đoạn MEM tới ALU

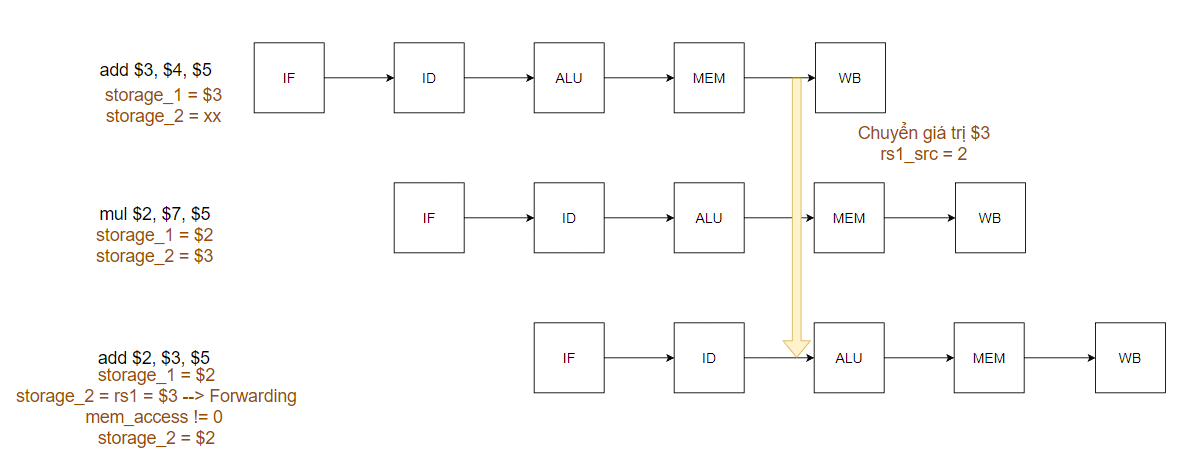
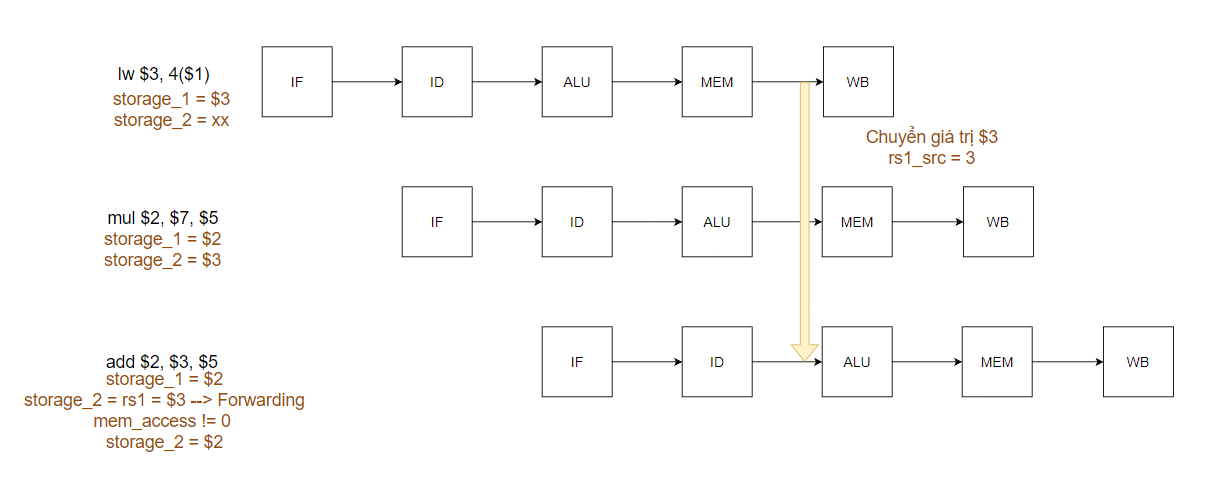


Figure : Chuyển giá trị alu\_result từ giai đoạn WB tới ALU



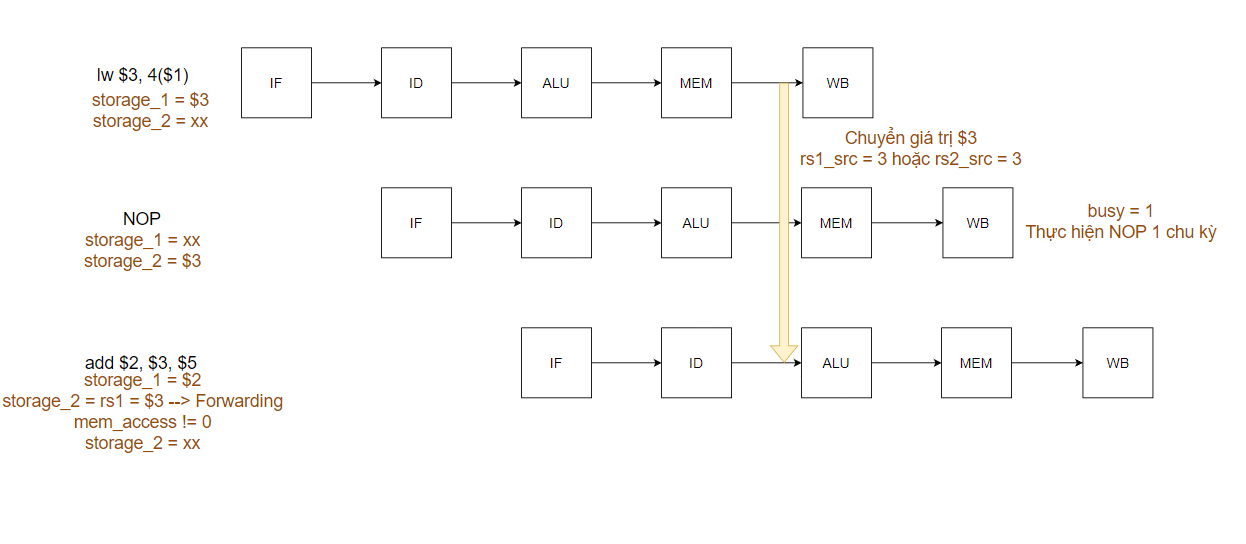
Figure : Chuyển giá trị mem\_result từ giai đoạn WB tới ALU

Figure : Thực hiện NOP khi Forwarding.

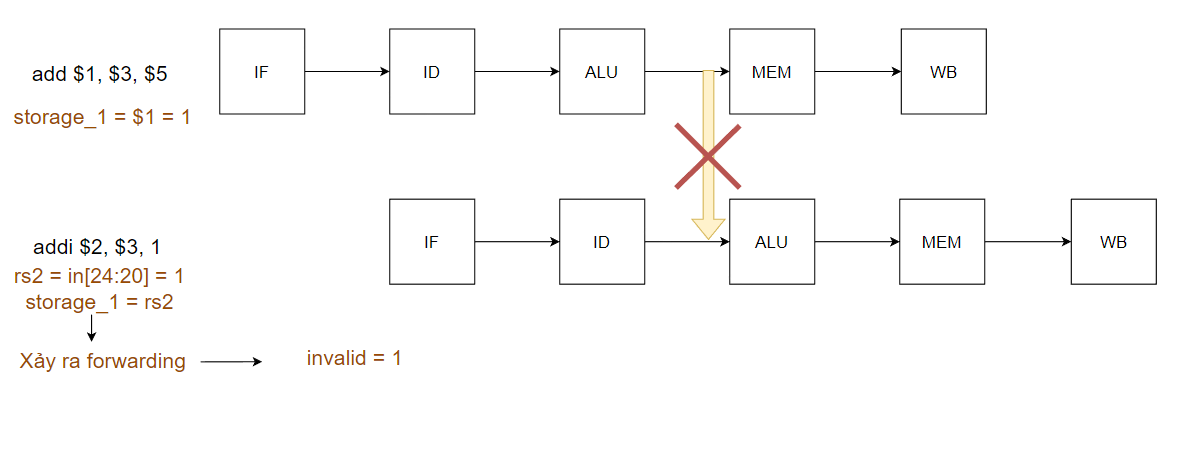


Figure : Không Forward khi invalid = 1.

Tuy nhiên, vẫn còn một vài trường hợp xung đột dữ liệu mà Forwarding unit trên chưa xử lý được. Nhóm vẫn tiếp tục tìm hiểu và cập nhật mới.

2. Xử lý xung đột điều khiển.

Một số lệnh nhảy có điều kiện và không điều kiện (branches, jumps) tạo ra xung đột điều kiển này.

Để xử lý xung đột điều khiển, CPU sẽ dự đoán điều kiện bằng sẽ không xảy ra, tức nhánh nhảy tới sẽ không được lấy (Predict Not Taken) bằng cách thiết kế thêm khối chức năng nâng cao Branch Hazard Control.

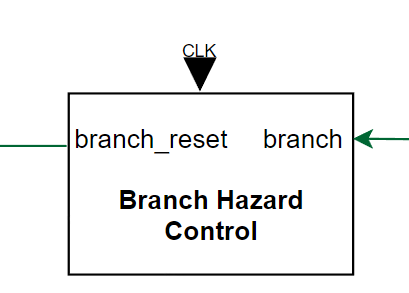


Figure : Thiết kế Branch Hazard Control.

- Input:

1. branch: Tín hiệu cho phép nhảy của RISC-V CPU.
2. CLK: clock tích cực mức cao.

- Output:

1. branch\_reset: Tín hiệu xử lý khi CPU có lệnh nhảy.

* Branch Hazard Control có chức năng reset các pipeline đã nạp các lệnh được dự đoán sai khi xảy ra xung đột điều khiển (xuất hiện lệnh nhảy) và tốn một vài chu kỳ để reset.
* Khối được đồng bộ với CPU thông qua tín hiệu CLK.
* Khi không xuất hiện lệnh nhảy thì CPU hoạt động bình thường và CPU không tiêu tốn bất kì chu kỳ nào.

|  |  |
| --- | --- |
| Điều kiện | branch\_reset |
| branch = 0 | 0 |
| branch = 1 | 1 |

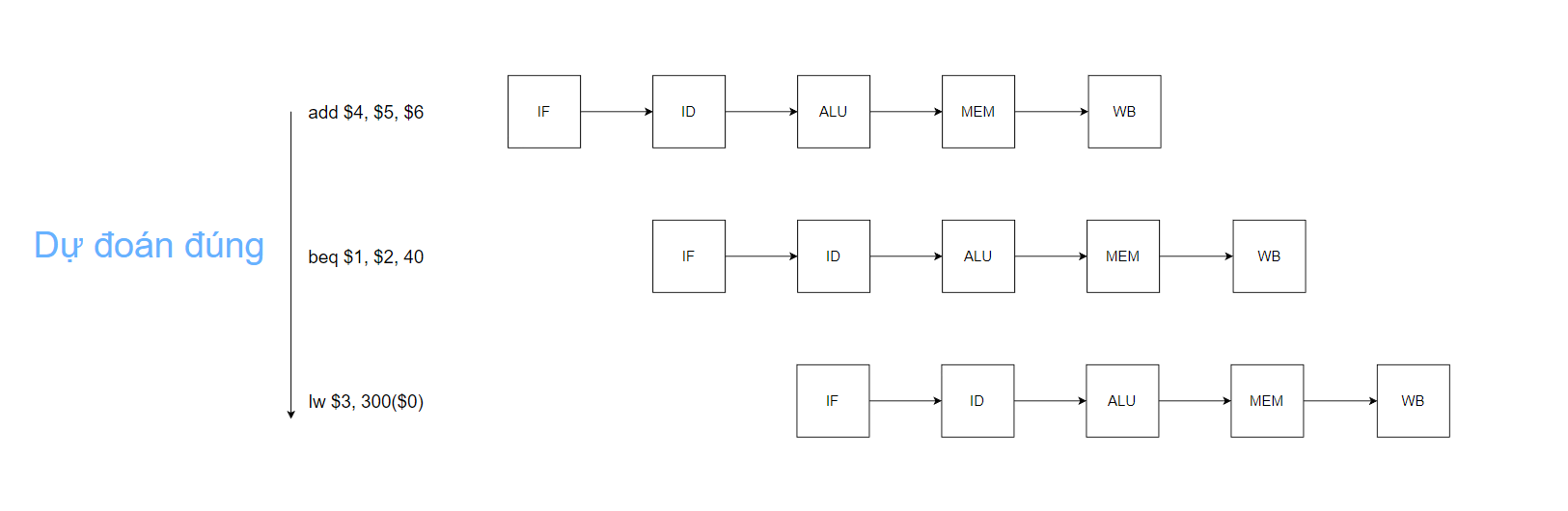
* Branch Hazard Control đơn giản là các thanh ghi dùng để đồng bộ các tín hiệu với nhau. Khi không có các thanh ghi này hay nói cách khác là chúng không được đồng bộ với nhau thì ngay thời điểm CPU nhảy, branch\_reset = 1 và các pipeline ngay lập tức được reset 🡪 tín hiệu branch được reset về 0. Do đó khi mà tín hiệu branch ở mức dương thì nó sẽ ngay lập tức chuyển thành mức thấp, từ đó khiến cho CPU hoạt động không chính xác.
* Với cách thiết kế datapath ở Figure 1 thì CPU có tín hiệu branch tại giai đoạn ALU. Do đó khi dự đoán sai, ta cần tốn 1 chu kỳ để nhận biết xung đột và thêm 1 chu kỳ để reset pipeline. Tổng cộng ta tốn tối đa 2 chu kỳ để xử lý xung đột lệnh nhảy. Ta biểu diễn kỹ thuật Predict Not Taken như sau:

Figure : Xủ lý xung đột điều khiển (1).

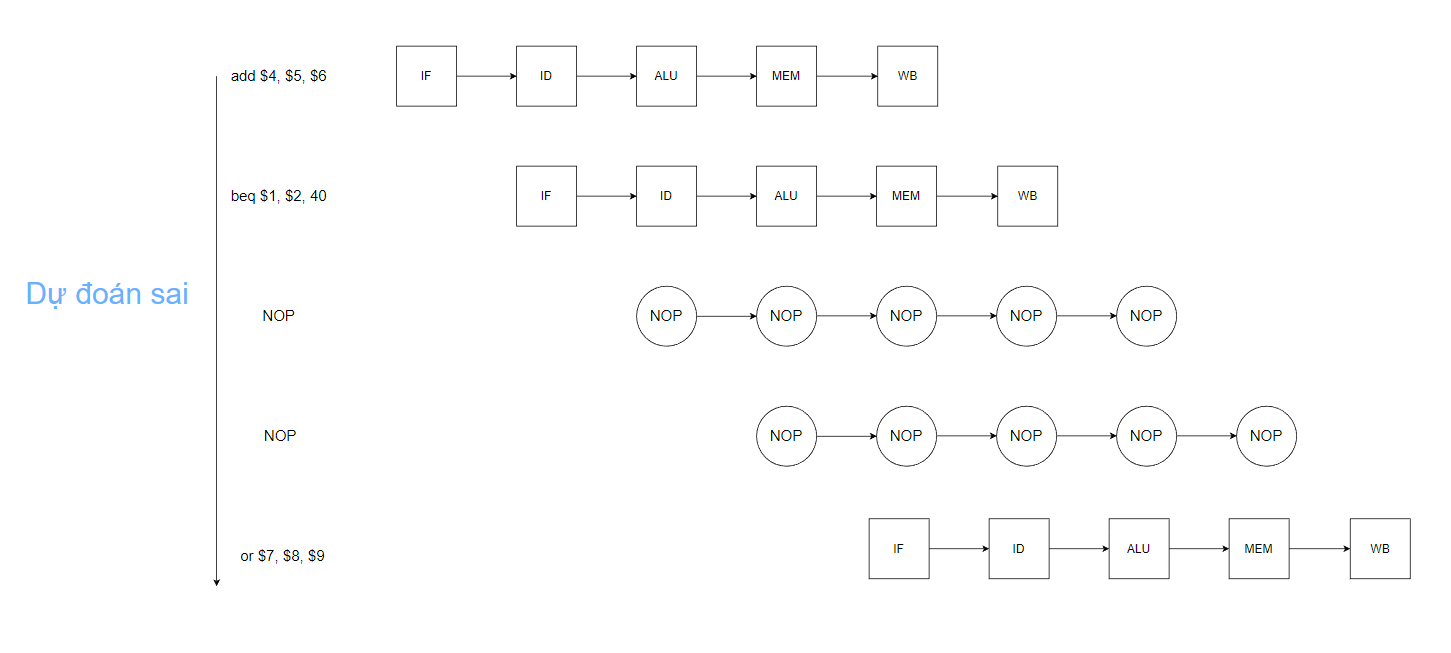


Figure : Xử lý xung đột điều khiển (2).

# Hoạt động của Datapath.

RSIC-V CPU áp dụng cơ chế pipeline 5 giai đoạn lần lượt là: IF 🡪 ID 🡪 ALU 🡪 MEM 🡪 WB. Do đó để xử lý n lệnh không có xung đột dữ liệu thì sẽ mất nhanh nhất là 4 + n chu kỳ.

Ví dụ cho 3 lệnh không xung đột sau:

add $4, $5, $6

sub $1, $2, $3

lw $3, 300($0)

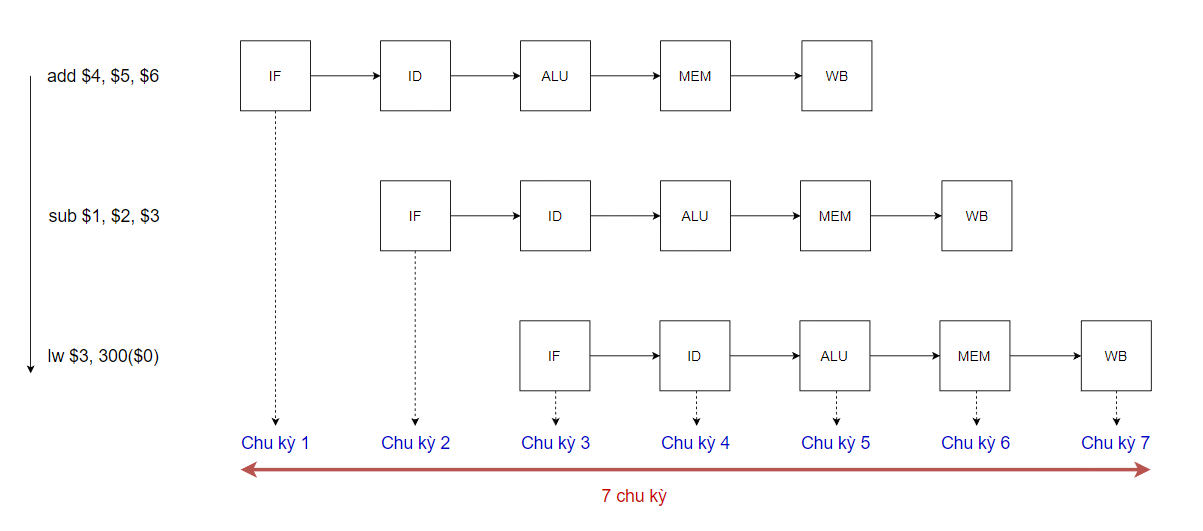
thì các lệnh sẽ hoạt động theo thứ tự như sau:  


Figure : Ví dụ hoạt động của Datapath (1).

Các thành phần chính như là Register File, Data Memory,… đều sẽ thực hiện đúng với thứ tự từng giai đoạn như Figure 23. Còn các khối chức năng nâng cao dùng để xử lý xung đột như là Forwarding unit và Branch Hazard Control thì sẽ hoạt động theo thứ tự như sau:

|  |  |  |
| --- | --- | --- |
| Thời điểm ngay khi kích dương xung clock | Forwarding unit xử lý lệnh | Branch Hazard Control xử lý lệnh |
| Clock thứ nhất | xx | xx |
| Clock thứ hai | xx | xx |
| Clock thứ ba | add $4, $5, $6 | xx |
| Clock thứ tư | sub $1, $2, $3 | add $4, $5, $6 |
| Clock thứ năm | lw $3, 300($0) | sub $1, $2, $3 |
| Clock thứ sáu | xx | lw $3, 300($0) |
| Clock thứ bảy | xx | xx |
| Clock thứ tám | xx | xx |

Table : Bảng hoạt động của các khối nâng cao.

Như vậy tất cả các khối thành phần đều được đồng bộ hoạt động với nhau qua clock. Ngoài trừ tín hiệu reset datapath bất đồng bộ dùng để khởi tạo CPU tại thời điểm ban đầu.

# Hiện thực Datapath và mô phỏng hoạt động.

* Ngôn ngữ: Verilog HDL.
* Phền mềm thiết kế: Quartus.
* Phần mềm mô phỏng: ModelSim-Altera.

1. Hiện thực Datapath.

Source code Verilog thiết kế datapath của nhóm được gửi đính kèm với báo cáo. Ngoài ra để theo dõi thêm thì có thể truy cập link dưới đây để xem thiết kế:

[*https://github.com/TungnxUIT/RISC-V.git*](https://github.com/TungnxUIT/RISC-V.git)

2. Mô phỏng hoạt động.

a. Test 1.

Ở lần mô phỏng đầu tiên, nhóm sẽ mô phỏng datapath thực hiện các lệnh sau đây để kiểm tra xung đột và hoạt động của Datapath nhóm thiết kế:

addi $0, $1, 8 //$0 = 8

sw $0, 1($2)

lw $1, 1($2) //$1 = 8

mul $3, $0, $1 //$3 = 64

beq $0, $1, End //will jump to End

add $3, $0, $1 //$3 = 16

End: lui $3, 0xFFFFF //$3 = -4096

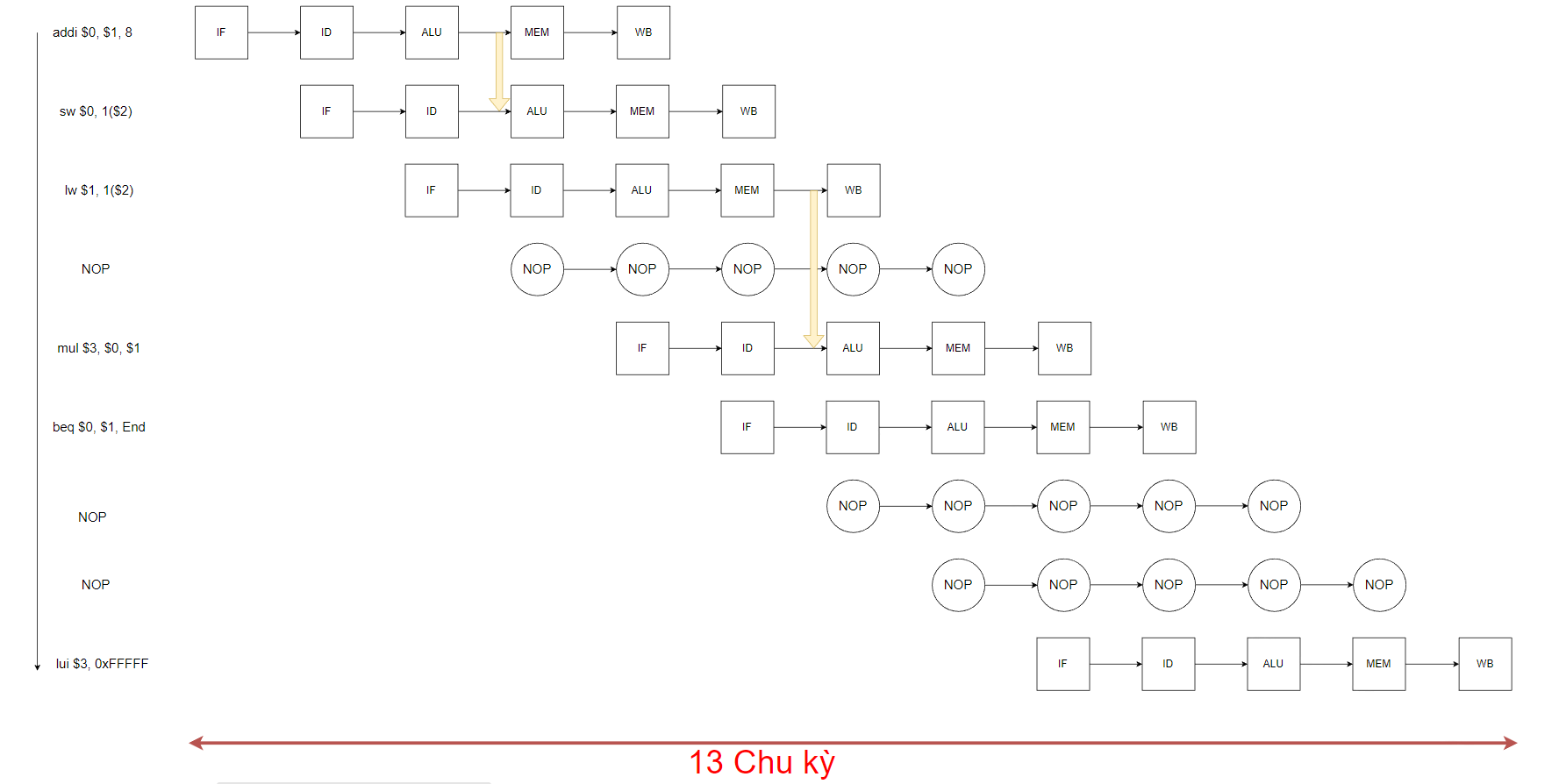
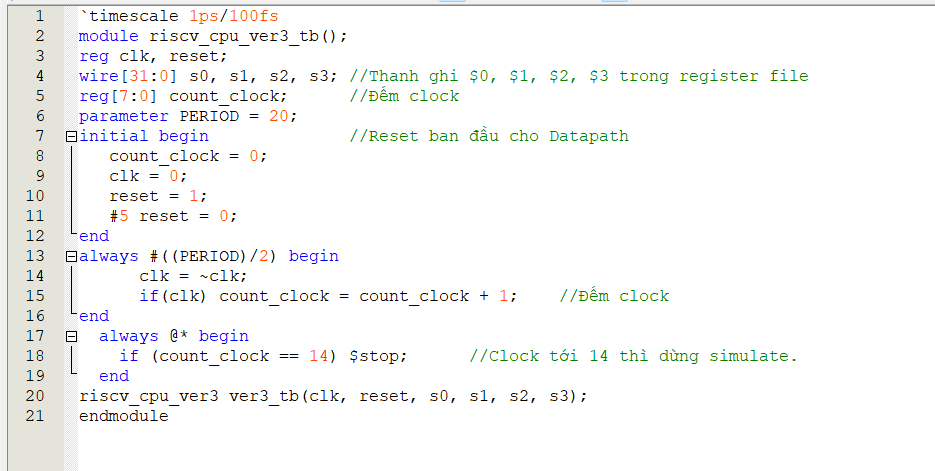


Figure : Hoạt động của Test 1.

* Lệnh addi $0, $1, 8 hoàn thành tại chu kỳ thứ 5 🡪 $0 = 8 tại chu kỳ 5.
* Lệnh lw $1, 1($2) hoàn thành tại chu kỳ thứ 7 🡪 $1 = 8 tại chu kỳ 7.
* Lệnh mul $3, $0, $1 hoàn thành tại chu kỳ thứ 9 🡪 $3 = 64 tại chu kỳ 9.
* Lệnh lui $3, 0xFFFFF hoàn thành tại chu kỳ thứ 13 🡪 $3 = -4096 tại chu kỳ 13

Ta có file testbench như sau:



Kết quả mô phỏng trong ModelSim-Altera :

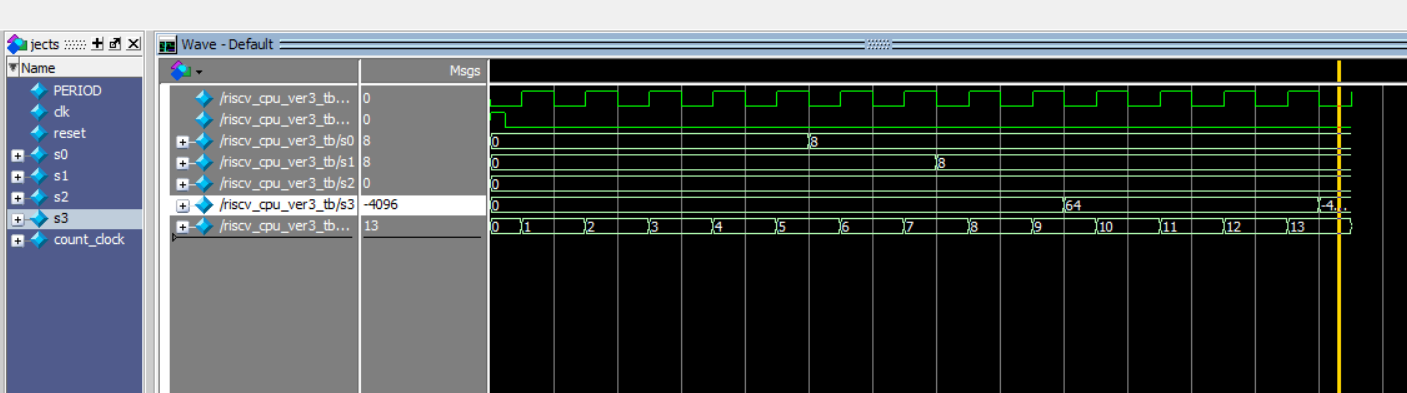
b. Test 2.

Figure : Kết quả mô phỏng Test 1.

Ở lần mô phỏng thứ 2, nhóm sẽ mô phỏng datapath thực hiện thuật toán tìm số Fibonanci ở vị trí thứ n (n > 0).

Các lệnh sẽ được mô phỏng để thực hiện thuật toán trên với n = 15 như sau:

addi s0, s5, 0

addi s1, s5, 1

addi s2, s5, 1

addi s3, s5, 15 //The final value stored in s1 is the 15th fibonacci number

loop:

beq s2, s3, end

add s4, s0, s1

add s0, s5, s1

add s1, s5, s4

addi s2, s2, 1

j loop

end:

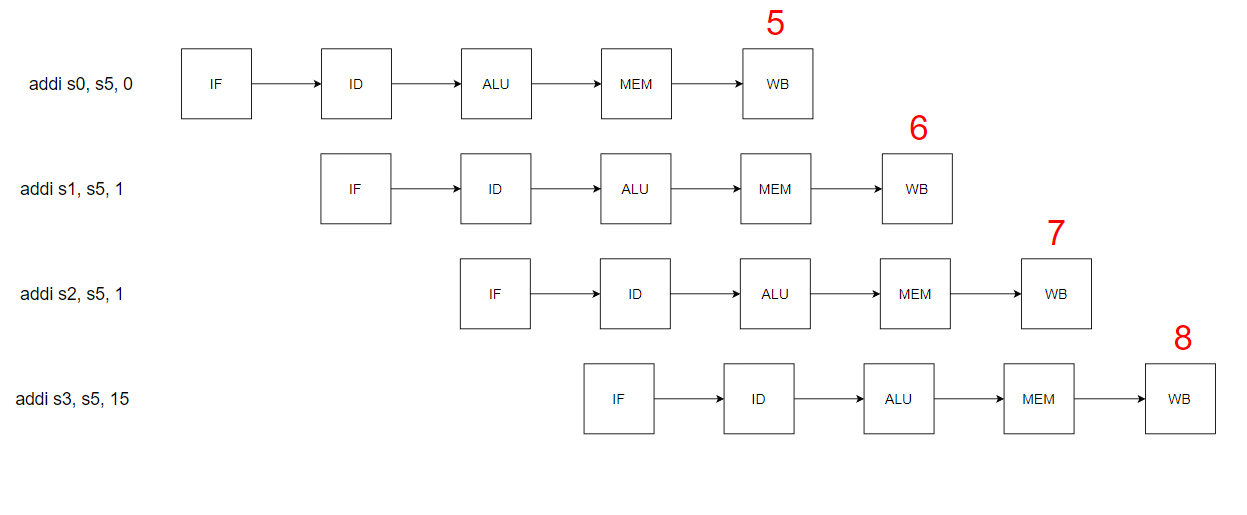
addi s2, s5, -1 //The final value stored in s2 is -1 which is the signal for the end

Kiểm tra tính đúng đắn của thuật toán trên, ta kiểm tra bằng cách nạp code trên trang web: [WebRISC-V - RISC-V PIPELINED DATAPATH SIMULATION ONLINE (unisi.it)](https://webriscv.dii.unisi.it/) và kiểm tra kết quả theo bảng hình dưới đây:

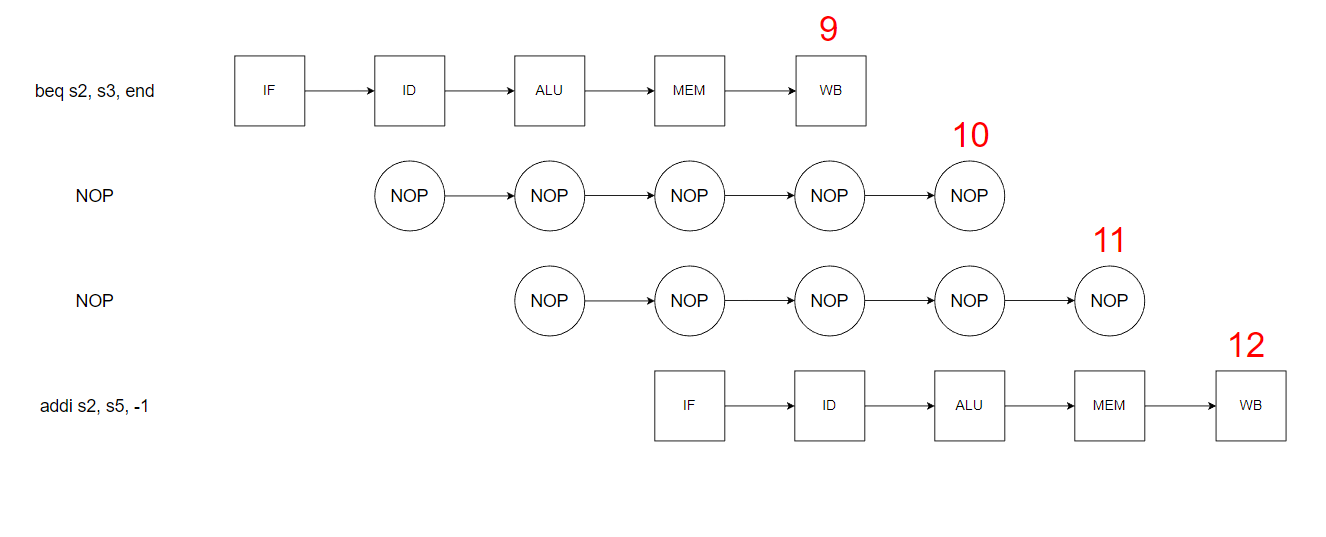


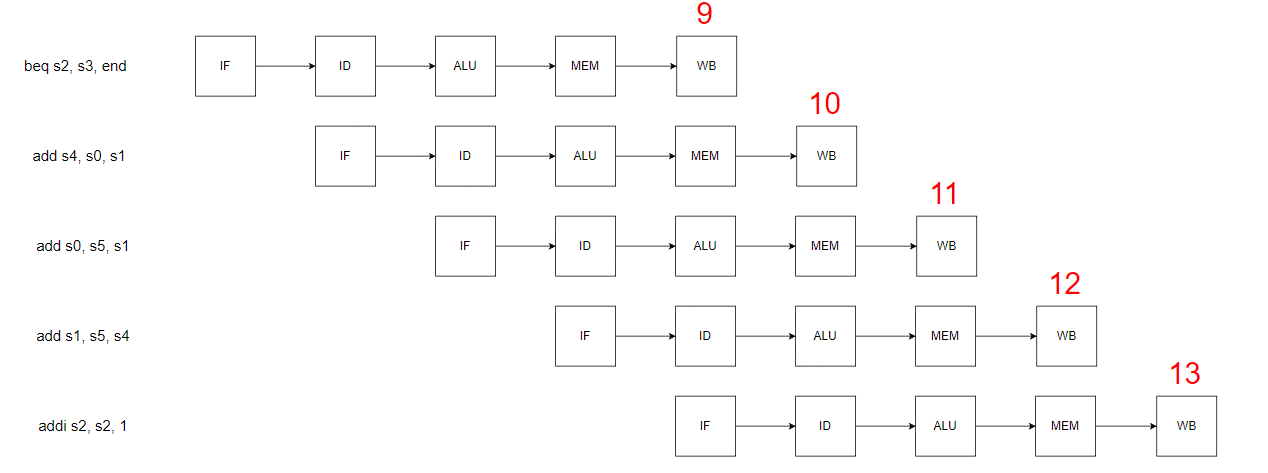
Figure : Dãy các số Fibonanci (Wikipedia).

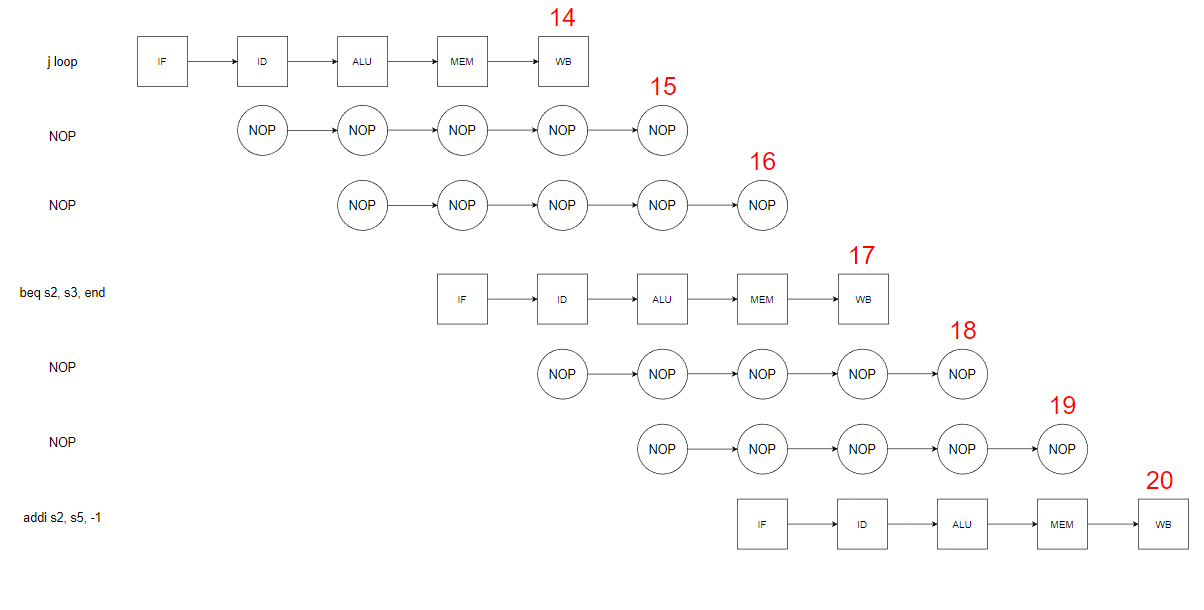
Dự đoán số chu kỳ thực hiện:

* Với n = 1:

1

 Như vậy với n = 1 thì số chu kỳ hoạt động là 12 chu kỳ.

* Với n = 2:

 Như vậy, với n = 2 thì CPU hoạt động với số chu kỳ là 20 chu kỳ.

Tương tự như vậy, ta có số chu kỳ hoạt động của CPU theo n như sau:

* n = 1: tốn 12 chu kỳ.
* n = 2: tốn 20 chu kỳ.
* n = 3: tốn 28 chu kỳ.
* n = 4: tốn 36 chu kỳ.

…

* n = a: Tốn 12 + (a - 1) \* 8 chu kỳ.

Kết luận: Với các lệnh mô phỏng trên, để tìm số Fibonanci ở vị trí thứ 15 ta sẽ cần tốn 12 + (15 -1) \* 8 = 124 chu kỳ đối với datapath nhóm đã thiết kế.

Testbench:

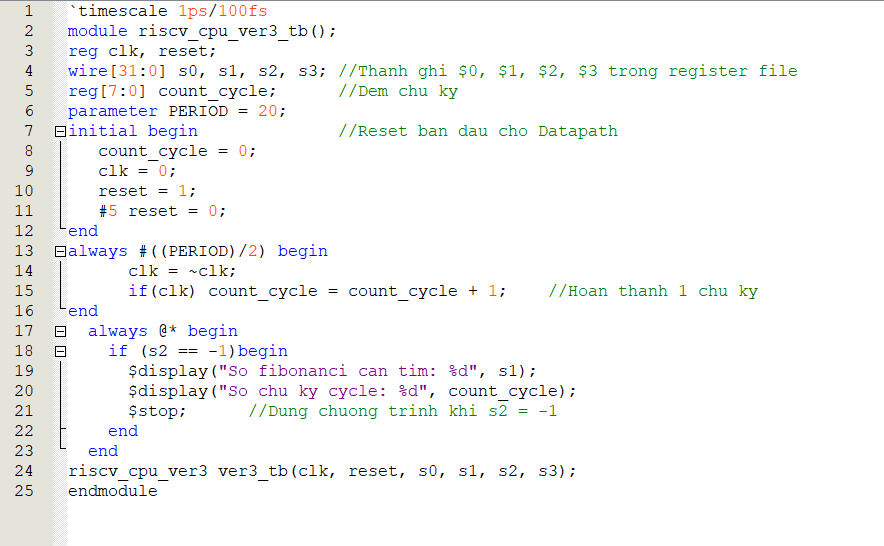
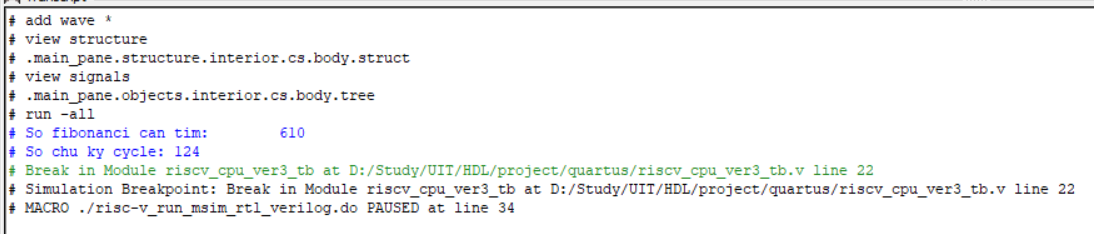
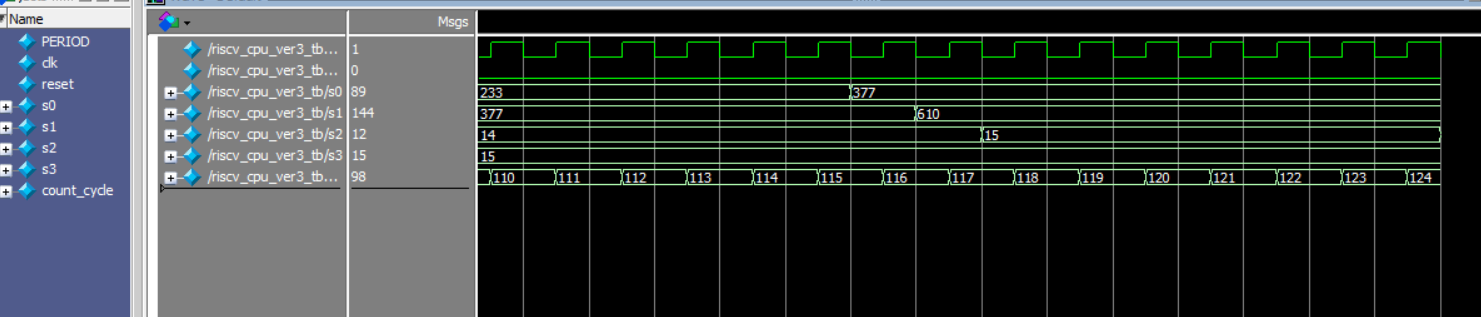


Figure : Testbench Test2.

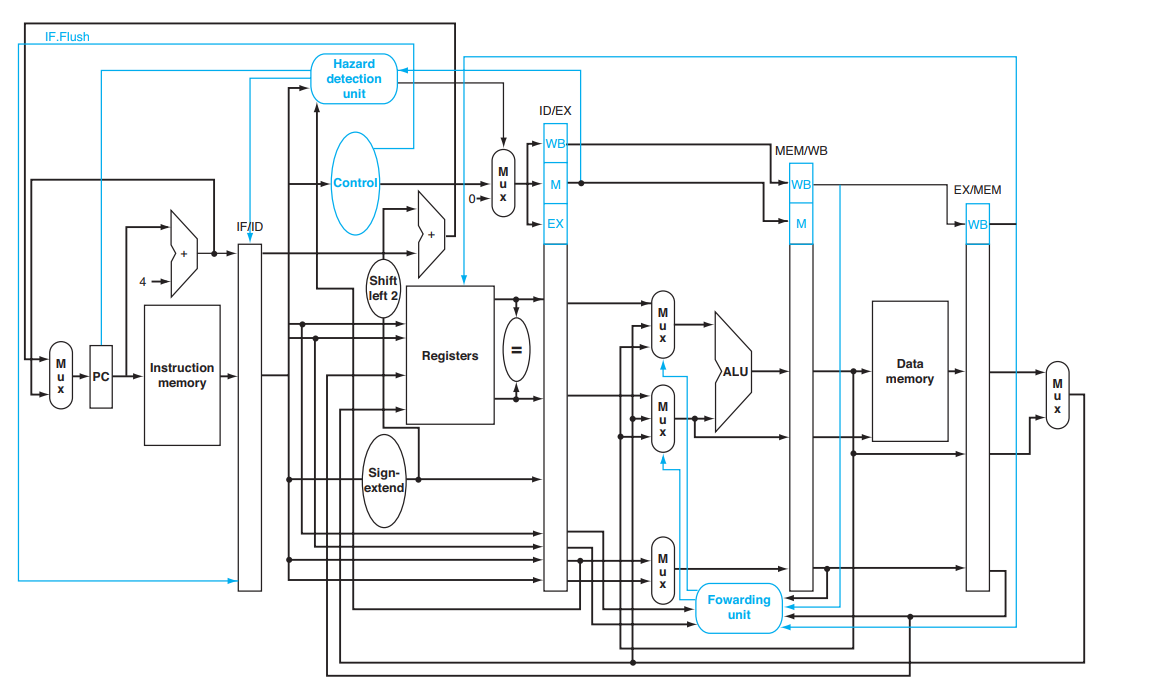
Kết quả mô phỏng bằng phần mềm ModelSim-Altera:

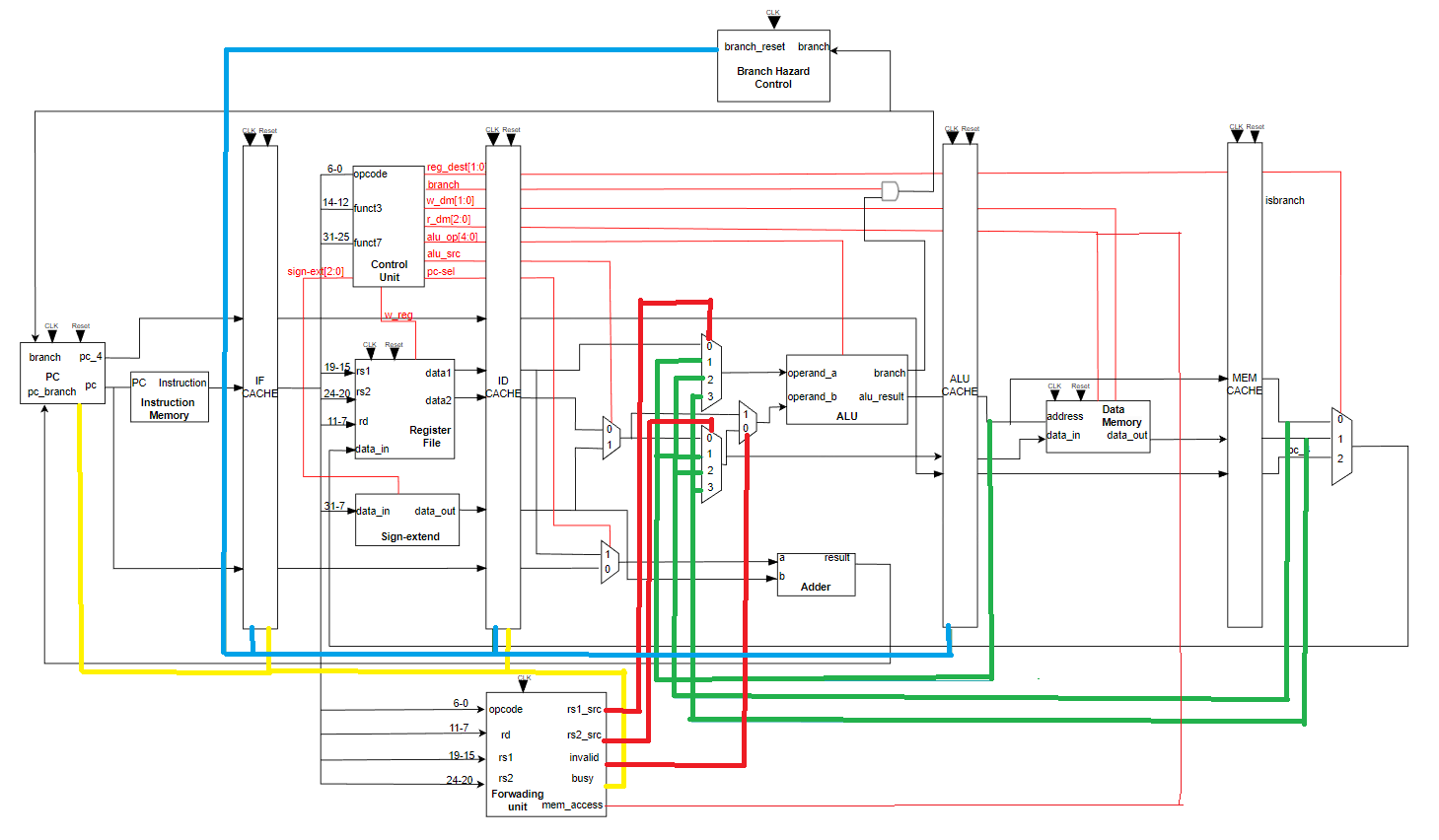
Xuất ra màn hình:



Dạng sóng waveform:

# So sánh với kiến trúc MIPS.





Giống nhau:

* Cần các khối thành phần chính để hoạt động như là: Register File, Sign Extend, ALU, Data Memory, Instruction Memory, Program Counter, Mux, Adder.
* Đều sử dụng cơ chế pipeline 5 giai đoạn: IF 🡪 ID 🡪 EX 🡪 MEM 🡪 WB.
* Đều sử dụng hai khối chức năng để xử lý xung đột: Forwarding unit và Branch Hazard Control (Hazard dectection unit ở kiến trúc MIPS).

Khác nhau:

|  |  |
| --- | --- |
| RSIC-V CPU của nhóm | MIPS |
| Program Counter có chức năng kiểm tra tín hiệu và lưu con trỏ pc địa chỉ lệnh kế tiếp hoặc địa chỉ lệnh nhảy. | Program Counter không quan tâm lệnh tiếp theo là lệnh gì, khối chỉ thực hiện chức năng lưu địa chỉ nhận được vào pc. |
| Program Counter nhận tín hiệu xử lý xung đột từ Forwarding unit. | Program Counter nhận tín hiệu xử lý xung đột từ Hazard dectecion unit. |
| Kiểm tra CPU được phép nhảy tại giai đoạn EX. | Kiểm tra CPU được phép nhảy tại giai đoạn ID. |
| Xử lý xung đột điều khiển mất tối đa hai chu kỳ. | Xử lý xung đột điều khiển mất tối đa một chu kỳ. |
| Chưa tối ưu được việc sử dụng các khối thành phần phụ. Đặc biệt tại giai đoạn EX, CPU chưa tối ưu việc sử dụng MUX để lựa chọn giá trị. | Sử dụng hợp lý các thành phần phụ tại các giai đoạn. |

# Tổng kết.

Từ đồ án thiết kế RSIC-V CPU kết hợp cơ chế pipeline, nhóm đã đạt được các yêu cầu sau:

* Hiểu biết rõ về việc sử dụng ngôn ngữ Verilog HDL để thiết kế phần cứng.
* Nắm bắt được kiến trúc RSIC-V cũng như việc ứng dụng pipeline.
* Ứng dụng cơ chế pipeline trong kiến trúc RSIC-V.
* Mô phỏng thành công các lệnh cơ bản và xử lý một số xung đột khi ứng dụng cơ chế pipeline vào kiến trúc RSIC-V.

Tuy nhiên, RSIC-V CPU do nhóm thiết kế cần được cải thiện một vài điểm như sau:

* Xây dựng đa dạng thêm các tập lệnh mới.
* Tối ưu phần cứng sử dụng như là các bộ MUX.
* Rất có khả năng rằng CPU vẫn chưa giải quyết được hoàn toàn các xung đột khi ứng dụng cơ chế pipeline. Cần tìm hiểu và cập nhật thêm.
* Tối ưu việc xử lý xung đột điều khiển trong tối đa một chu kỳ thay vì hai chu kỳ.
* Tìm hiểu thêm về thời gian hoạt động của từng giai đoạn, đo lường thời gian một cách hợp lý để tính ra thời gian của một chu kỳ. Từ đó có thể khai triển trên các phần cứng thật sự thay vì chỉ mô phỏng trên phần mềm.
* Xây dựng thêm Assembler dành riêng cho RSIC-V CPU do nhóm thiết kế.

# Tài liệu tham khảo.

* Computer Organization and Design: The Hardware/Software Interface, Patterson, D. A., and J. L. Hennessy, Morgan Kaufman, Revised Fourth Edition, 2011.
* Computer Organization and Design RISC-V Edition: The Hardware Software Interface, David A. PatterSon, 1996.
* IEEE Standard for Verilog® Hardware Description Language.
* THE COMPLETE VERILOG BOOK by Vivek Sagdeo (Sun Micro Systems, Inc)
* [WebRISC-V - RISC-V PIPELINED DATAPATH SIMULATION ONLINE (unisi.it)](https://webriscv.dii.unisi.it/) (RSIC-V online simulator).
* [Online RISC-V Assembler (lucasteske.dev)](https://riscvasm.lucasteske.dev/). (RSIC-V online assembler).