

**ĐẠI HỌC QUỐC GIA TP. HỒ CHÍ MINH**  
**TRƯỜNG ĐẠI HỌC BÁCH KHOA**  
**KHOA ĐIỆN – ĐIỆN TỬ**  
**BỘ MÔN ĐIỆN TỬ**

-----o0o-----



**ĐỒ ÁN 2**

**THIẾT KẾ BỘ XỬ LÝ MIPS 32 BIT THEO KỸ  
THUẬT PIPELINE**

**GVHD: Ths. Nguyễn Trung Hiếu**

**SVTH: Huỳnh Quốc Sĩ**

**MSSV: 2014361**

**TP. HỒ CHÍ MINH, THÁNG 3 NĂM 2024**

## **LỜI NÓI ĐẦU**

Lời đầu tiên em xin được bày tỏ lòng biết ơn sâu sắc đến những thầy cô trong khoa Điện - Điện tử trường Đại học Bách Khoa, ĐHQG TP Hồ Chí Minh đã tận tình, dạy dỗ, tạo điều kiện cho em phát triển trong suốt khoảng thời gian học tập tại trường. Đặc biệt, em xin cảm ơn thầy Ths.Nguyễn Trung Hiếu đã trực tiếp giúp đỡ, hướng dẫn em hoàn thành đề tài này.

Trong quá trình thực hiện, em đã cố gắng hết mình để hoàn thành đề tài. Tuy nhiên, do kiến thức, kinh nghiệm còn hạn chế nên không thể tránh khỏi những thiếu sót, em rất mong được sự góp ý từ quý thầy cô để đề tài em hoàn thiện hơn nữa. Em xin chân thành cảm ơn!

## MỤC LỤC

<b>CHƯƠNG 1. TỔNG QUAN VÀ MỤC TIÊU THIẾT KẾ BỘ XỬ LÍ MIPS 32 BITS PIPELINE .....</b>	<b>3</b>
<b>1.1 Giới thiệu bộ xử lí MIPS 32 bits Pipeline .....</b>	<b>3</b>
<b>1.2 Nguyên tắc thiết kế kiến trúc MIPS Pipeline.....</b>	<b>5</b>
<b>1.3 Kiến trúc RISC và CISC.....</b>	<b>6</b>
1.3.1 RISC .....	6
1.3.2 CISC .....	6
<b>1.4 Mục tiêu thiết kế.....</b>	<b>7</b>
<b>1.5 So sánh kiến trúc MIPS với một số kiến trúc có mặt hiện nay.....</b>	<b>7</b>
<b>CHƯƠNG 2. KIẾN TRÚC TẬP LỆNH MIPS .....</b>	<b>9</b>
<b>2.1 Các toán hạng thanh ghi trong MIPS .....</b>	<b>9</b>
<b>2.2 Toán hạng bộ nhớ.....</b>	<b>11</b>
<b>2.3 Toán hạng hàng.....</b>	<b>12</b>
<b>2.4 Các định dạng lệnh MIPS .....</b>	<b>13</b>
2.4.1 Định dạng R .....	13
2.4.2 Định dạng I .....	14
2.4.3 Định dạng J .....	14
2.4.4 Tập lệnh MIPS (một phần) dùng cho thiết kế .....	14
<b>2.5 Bộ cục bộ nhớ ( Memory map ) .....</b>	<b>16</b>
<b>CHƯƠNG 3. THIẾT KẾ BỘ XỬ LÍ MIPS 32 BITS PIPELINE.....</b>	<b>17</b>
<b>3.1 Mô hình thực thi lệnh.....</b>	<b>17</b>
<b>3.2 Các thành phần cơ bản của bộ xử lý.....</b>	<b>19</b>
3.2.1 Đường dữ liệu.....	19
3.2.2 Bộ điều khiển.....	23
<b>3.3 Thiết kế tổng quan.....</b>	<b>23</b>
<b>3.4 Bộ nhớ dữ liệu .....</b>	<b>26</b>
<b>3.5 Bộ nhớ lệnh .....</b>	<b>27</b>
<b>3.6 Tệp thanh ghi .....</b>	<b>27</b>

<b>3.7 Thanh ghi PC .....</b>	<b>28</b>
<b>3.8 Bộ điều khiển.....</b>	<b>28</b>
<b>3.9 Thanh ghi IF-ID.....</b>	<b>29</b>
<b>3.10 Thanh ghi ID-EX.....</b>	<b>30</b>
<b>3.11 Thanh ghi EX-MEM.....</b>	<b>31</b>
<b>3.12 Thanh ghi MEM-WB.....</b>	<b>32</b>
<b>3.13 IF Stage .....</b>	<b>33</b>
<b>3.14 ID Stage.....</b>	<b>34</b>
<b>3.15 EX Stage .....</b>	<b>35</b>
<b>3.16 MEM Stage.....</b>	<b>38</b>
<b>3.17 WB Stage .....</b>	<b>39</b>
<b>3.18 Khối phát hiện xung đột và khối chuyển tiếp dữ liệu .....</b>	<b>39</b>
3.18.1 Cơ chế phát hiện xung đột và giải quyết xung đột bằng phương pháp chuyển tiếp dữ liệu .....	39
3.18.2 Khối phát hiện xung đột .....	40
3.18.3 Khối chuyển tiếp dữ liệu .....	41
<b>CHƯƠNG 4. KẾT QUẢ MÔ PHỎNG THIẾT KẾ .....</b>	<b>44</b>
<b>4.1 Công cụ thực hiện .....</b>	<b>44</b>
<b>4.2 Chương trình 1.....</b>	<b>44</b>
<b>4.3 Chương trình 2.....</b>	<b>49</b>
<b>TÀI LIỆU THAM KHẢO.....</b>	<b>57</b>

## DANH SÁCH HÌNH ẢNH

Hình 2.1 Bô cục bộ nhớ.....	16
Hình 3.0.1 Sơ đồ khói bộ nhớ dữ liệu .....	20
Hình 3.0.2 Sơ đồ khói tập thanh ghi.....	20
Hình 3.0.3 Sơ đồ khói ALU .....	21
Hình 3.0.4 Sơ đồ khói bộ nhớ dữ liệu .....	22
Hình 3.0.5 Sơ đồ khói bộ điều khiển.....	23
Hình 3.0.6 Thiết kế tổng quan bộ xử lý MIPS 32 bits pipeline .....	24
Hình 3.0.7 Nguyên lý thực thi chương trình bằng bộ xử lý pipeline .....	25
Hình 3.0.8 Chương trình gây xung đột trong bộ xử lý pipeline .....	25
Hình 3.0.9 Bộ nhớ dữ liệu.....	26
Hình 3.0.10 Bộ nhớ lệnh.....	27
Hình 3.0.11 Tệp thanh ghi .....	27
Hình 3.0.12 Thanh ghi PC .....	28
Hình 3.0.13 Bộ điều khiển .....	29
Hình 3.0.14 Thanh ghi IF-ID .....	29
Hình 3.0.15 Thanh ghi ID-EX.....	30
Hình 3.0.16 Thanh ghi EX-MEM .....	31
Hình 3.0.17 Thanh ghi MEM-WB .....	32
Hình 3.0.18 IF Stage .....	33
Hình 3.0.19 Thiết kế chi tiết IF Stage .....	33
Hình 3.0.20 ID Stage .....	34
Hình 3.0.21 Thiết kế chi tiết ID Stage .....	35
Hình 3.0.22 EX Stage .....	36
Hình 3.0.23 Thiết kế chi tiết EX Stage .....	37
Hình 3.0.24 Thiết kế chi tiết MEM Stage.....	38
Hình 3.0.25 Thiết kế chi tiết WB Stage .....	39
Hình 3.0.26 Xung đột EX/MEM.....	40
Hình 3.0.27 Xung đột MEM/WB.....	40

Hình 3.0.28 Khối phát hiện xung đột .....	41
Hình 3.0.29 Khối chuyển tiếp dữ liệu .....	42
Hình 4.0.1 Chương trình 1 .....	44
Hình 4.0.4 Kết quả mô phỏng chương trình 1(3).....	48
Hình 4.0.5 Kết quả mô phỏng chương trình 1(4).....	48
Hình 4.0.6 Chương trình 2 .....	49
Hình 4.0.7 Kết quả mô phỏng chương trình 2(1).....	51
Hình 4.0.8 Kết quả mô phỏng chương trình 2(2).....	51
Hình 4.0.9 Kết quả mô phỏng chương trình 2(3).....	52
Hình 4.0.10 Kết quả mô phỏng chương trình 2(4).....	52

# CHƯƠNG 1. TỔNG QUAN VÀ MỤC TIÊU THIẾT KẾ BỘ XỬ LÍ MIPS 32 BITS PIPELINE

## 1.1 Giới thiệu bộ xử lý MIPS 32 bits Pipeline

Kiến trúc MIPS (Microprocessor without Interlocked Pipeline Stages) là một trong những kiến trúc vi xử lý (CPU) nổi tiếng và được sử dụng rộng rãi trong ngành công nghiệp máy tính và điện tử nhúng. Được phát triển từ những năm 1980, kiến trúc MIPS là một ví dụ điển hình của các kiến trúc RISC (Reduced Instruction Set Computing - Máy tính với tập lệnh đơn giản hóa).

Bộ xử lý MIPS đầu tiên được nghiên cứu vào năm 1981 với mục đích chính là tăng cường hiệu năng thông qua sử dụng một đường ống lệnh (pipeline instructions). Pipeline cho phép vi xử lý thực hiện nhiều giai đoạn của một lệnh trong cùng một thời điểm. Thay vì thực hiện một lệnh hoàn toàn trước khi bắt đầu lệnh tiếp theo, thì vi xử lý có thể bắt đầu thực hiện một lệnh mới mà không cần chờ lệnh trước đó hoàn thành tất cả các giai đoạn của nó. Khi một lệnh hoàn thành một giai đoạn, nó di chuyển đến giai đoạn tiếp theo và vi xử lý có thể bắt đầu thực hiện một lệnh mới trong giai đoạn đầu tiên. Việc sử dụng pipeline giúp giảm thời gian chờ đợi giữa các lệnh, tăng cường hiệu suất và tăng tốc độ thực thi chương trình.

Khó khăn trong quá trình tìm hiểu thiết kế: việc thực hiện đường ống lệnh trong thiết kế kiến trúc MIPS đòi hỏi sự sử dụng các khóa đồng bộ (interlocks) để đảm bảo rằng các lệnh chiếm nhiều chu kỳ đồng hồ để thực hiện sẽ không gây ra xung đột và đảm bảo tính chính xác của dữ liệu trong hệ thống. Những khóa đồng bộ này có thể gây ra một số vấn đề và khó khăn:

- **Tăng thời gian đáp ứng:** Việc thêm các khóa đồng bộ có thể làm tăng thêm thời gian để thực hiện mỗi lệnh, do đó làm giảm hiệu suất của hệ thống.

- **Phức tạp hóa thiết kế:** Sự thêm vào của các khóa đồng bộ có thể làm phức tạp hóa thiết kế của vi xử lý và đường ống, tăng chi phí phát triển và kiểm thử.
- **Rủi ro xung đột:** Nếu không cài đặt các khóa đồng bộ một cách phù hợp, có thể xảy ra các xung đột dữ liệu hoặc xung đột điều kiện, dẫn đến lỗi trong kết quả tính toán.

Mặc dù có những khó khăn này, nhưng việc sử dụng đường ống lệnh vẫn mang lại nhiều lợi ích lớn, bao gồm tăng tốc độ xử lý và cải thiện hiệu suất của hệ thống. Các nhà thiết kế đã phát triển các kỹ thuật và công nghệ để giảm thiểu tác động của các khóa đồng bộ và tối ưu hóa hiệu suất của hệ thống. Điều này bao gồm việc sử dụng các kỹ thuật tăng tốc độ như các bộ đệm (buffering) và tối ưu hóa lập lịch lệnh (instruction scheduling) để giảm thiểu thời gian rảnh rỗi và tăng hiệu suất.

Yêu cầu đặt ra trong quá trình thiết kế là yêu cầu tất cả các lệnh phải được hoàn thành trong 1 chu kỳ xung nhịp, mà không cần thiết các khóa đồng bộ. Điều này có thể được đạt được thông qua việc thiết kế một CPU với một tập hợp các lệnh đơn giản và hiệu quả, mỗi lệnh chỉ cần một chu kỳ xung nhịp để hoàn thành. Một số lệnh phức tạp như nhân và chia có thể bị loại bỏ hoặc thay thế bằng các lệnh tương đương đơn giản hơn, có thể hoàn thành trong một chu kỳ xung nhịp.

Tuy nhiên việc loại bỏ một số lệnh phức tạp như nhân và chia có thể giảm đi tính linh hoạt của CPU và các ứng dụng có thể chạy trên đó, nhưng nó có thể tăng đáng kể hiệu suất tổng thể của hệ thống. Bởi vì việc yêu cầu tất cả các lệnh hoàn thành trong một chu kỳ xung nhịp cho phép CPU chạy ở xung nhịp lớn hơn, từ đó tăng tốc độ xử lý và cải thiện hiệu suất của hệ thống.

Nhưng điều này cũng có một số hạn chế, bao gồm:

- **Giới hạn tính linh hoạt:** Loại bỏ các lệnh phức tạp có thể giới hạn khả năng của CPU để thực hiện các tác vụ phức tạp và đa dạng.

- Giảm chất lượng mã máy: Loại bỏ các lệnh phức tạp có thể dẫn đến mã máy không hiệu quả hoặc mã máy phức tạp hơn để thực hiện các tác vụ tương tự.

Tóm lại, việc yêu cầu tất cả các lệnh phải hoàn thành trong một chu kỳ xung nhịp có thể là một phương pháp hiệu quả để tăng hiệu suất tổng thể của hệ thống, mặc dù điều này có thể đến với một số hạn chế về tính linh hoạt và chất lượng mã máy.

Các thế hệ của MIPS:

- + Ban đầu MIPS là kiến trúc 32 bit, sau này mở rộng ra 64bit.
- + MIPS I, MIPS II, MIPS III, MIPS IV, MIPS V, MIPS 32 và MIPS 64. Hiện nay tồn tại MIPS 32 và MIPS 64.
- + Các dòng vi xử lý thương mại MIPS đã được sản xuất:
  - Năm 1985, R2000.
  - Năm 1988, R3000.
  - Năm 1991, R4000 mở rộng tập lệnh đầy đủ cho 64bit, 100MHz, 8kB.
  - Năm 1993, R4400 16kB.
  - Năm 1994, R8000 thiết kế superscalar đầu tiên của MIPS.

## 1.2 Nguyên tắc thiết kế kiến trúc MIPS Pipeline

Kiến trúc MIPS pipeline được thiết kế dựa trên các nguyên tắc sau:

- Tính đơn giản quan trọng hơn tính quy tắc (Simplicity favors regularity)
  - + Chỉ thị kích thước cố định (32bit)
  - + Ít định dạng chỉ thị (3 loại định dạng)
  - + Mã lệnh ở vị trí cố định (6 bit đầu)
- Nhỏ hơn thì nhanh hơn
  - + Số chỉ thị giới hạn

- + Số thanh ghi giới hạn
- + Số chế độ địa chỉ giới hạn
- Tăng tốc trong các trường hợp thông dụng
  - + Các toán hạng số học lấy từ thanh ghi (máy tính dựa trên cơ chế load-store)
  - + Các chỉ thị có thể chứa toán hạng trực tiếp
- Thiết kế đòi hỏi sự thỏa hiệp: Ba loại chỉ thị định dạng.
- Nguyên tắc hoạt động của Pipeline
  - + Chia nhỏ các lệnh thành các giai đoạn đường ống
  - + Bắt đầu lệnh tiếp theo trước khi lệnh hiện tại kết thúc

### **1.3 Kiến trúc RISC và CISC**

#### *1.3.1 RISC*

RISC (Reduced Instruction Set Computer) là một phương pháp thiết kế kiến trúc máy tính tập trung vào sử dụng một tập hợp nhỏ các lệnh máy đơn giản và cơ bản. Trong kiến trúc RISC, các lệnh thường được thiết kế để thực hiện một số lượng giới hạn các chức năng cố định, và hầu hết các lệnh đều thực hiện trong một chu kỳ xung đồng hồ duy nhất. Đặc điểm chính của RISC bao gồm việc sử dụng một số lượng lệnh cố định, các lệnh có kích thước đồng nhất và đơn giản, sử dụng nhiều thanh ghi và các bộ đệm trong việc tăng tốc độ thực thi lệnh.

Tính đơn giản của các lệnh và quy trình thực thi của RISC dẫn đến hiệu suất cao hơn và dễ dàng tối ưu hóa hiệu suất của bộ xử lý. Đồng thời, kiến trúc RISC dễ dàng để phát triển và có chi phí thấp hơn so với kiến trúc CISC (Complex Instruction Set Computer).

#### *1.3.2 CISC*

CISC (Complex Instruction Set Computer), đây là một kiến trúc máy tính mà các lệnh thực hiện nhiều chức năng phức tạp và đa dạng. Trái ngược với RISC (Reduced

Instruction Set Computer), mỗi lệnh CISC thường thực hiện nhiều công việc, bao gồm việc truy cập bộ nhớ, tính toán, và các thao tác khác, trong khi các máy tính RISC thường chỉ thực hiện một công việc đơn giản mỗi lần lệnh được thực hiện. CISC thường có số lượng lệnh lớn và đa dạng, điều này có thể làm tăng độ phức tạp của việc thiết kế và hiệu suất của các máy tính sử dụng kiến trúc này có thể bị ảnh hưởng nếu không được tối ưu hóa đúng cách. Nổi bật của kiến trúc CISC là x86 được dùng trong các bộ xử lý của Intel.

#### 1.4 Mục tiêu thiết kế

Mục đích của bản thiết kế, nhằm tạo ra một bộ xử lý MIPS, thông qua sử dụng đường ống lệnh (pipeline instructions) để tăng hiệu suất của bộ xử lý. Thiết kế pipeline làm giảm đáng kể thời gian rảnh rỗi của CPU khi thực hiện liên tiếp các câu lệnh. Bộ xử lý này chỉ có khả năng thực hiện được một số lệnh cơ bản:

- Các phép toán số học: add, addi, sub, and, andi, or, slt.
- Truy cập bộ nhớ với 2 chỉ thị: lw, sw.
- Lệnh rẽ nhánh, nhảy: beq, bne, j, jal.

#### 1.5 So sánh kiến trúc MIPS với một số kiến trúc có mặt hiện nay

Kiến trúc MIPS (Microprocessor without Interlocked Pipeline Stages) đã định hình nền tảng cho nhiều loại kiến trúc vi xử lý và vi điều khiển, và nó vẫn đang được sử dụng trong nhiều ứng dụng khác nhau. Dưới đây là một số so sánh giữa kiến trúc MIPS và một số kiến trúc vi xử lý hiện đại khác:

- ARM (Advanced RISC Machine):

**Đặc điểm chung:** Cả MIPS và ARM đều dựa trên mô hình RISC (Reduced Instruction Set Computing) với các bộ lệnh đơn giản và thường có cấu trúc pipeline.

**Khác biệt:** MIPS thường có các bộ lệnh có độ dài cố định (32 bit), trong khi ARM có thể có các bộ lệnh có độ dài biến (16-bit hoặc 32-bit). ARM cũng có một loạt các biến

thể, bao gồm ARM Cortex-A (dành cho ứng dụng cao cấp), Cortex-R (dành cho hệ thống nhúng) và Cortex-M (dành cho hệ thống nhúng cỡ nhỏ).

**Hiệu suất và Tiêu thụ năng lượng:** Một số phiên bản ARM có hiệu suất cao hơn và tiêu thụ năng lượng thấp hơn so với MIPS, nhưng điều này phụ thuộc vào cấu trúc cụ thể và kỹ thuật thiết kế.

- x86 (Intel và AMD):

**Đặc điểm chung:** MIPS và x86 đều là kiến trúc máy tính có ảnh hưởng lớn đối với ngành công nghiệp máy tính.

**Khác biệt:** Kiến trúc x86 được sử dụng rộng rãi trong các máy tính cá nhân và máy chủ, trong khi MIPS thường được sử dụng trong các hệ thống nhúng, router, các thiết bị điện tử tiêu dùng và các ứng dụng như đồng hồ thông minh, camera an ninh, vv.

**Tiêu thụ năng lượng và hiệu suất:** MIPS thường được thiết kế để tiêu thụ ít năng lượng hơn so với x86, nhưng x86 thường có hiệu suất cao hơn và được tối ưu hóa cho các ứng dụng máy tính cá nhân và máy chủ.

- RISC-V:

**Đặc điểm chung:** MIPS và RISC-V đều là kiến trúc RISC, và RISC-V có xu hướng trở thành một tiêu chuẩn nguồn mở và linh động hơn so với MIPS.

**Khác biệt:** MIPS là một kiến trúc cụ thể được phát triển bởi MIPS Technologies, trong khi RISC-V là một kiến trúc mã nguồn mở có sẵn để sử dụng và tùy chỉnh miễn phí cho cộng đồng.

**Phổ biến và hỗ trợ:** RISC-V đang trở nên phổ biến hơn trong các dự án nguồn mở và trong các ứng dụng nhúng, trong khi MIPS vẫn được sử dụng trong một số ứng dụng nhưng không còn phổ biến như trước.

Mỗi kiến trúc có những ưu điểm và hạn chế riêng, và sự lựa chọn giữa chúng phụ thuộc vào yêu cầu cụ thể của ứng dụng.

## CHƯƠNG 2. KIẾN TRÚC TẬP LỆNH MIPS

### 2.1 Các toán hạng thanh ghi trong MIPS

Thanh ghi là các đơn vị cơ bản dùng để lưu trữ dữ liệu trong một hệ thống phân cứng. Điểm khác nhau cơ bản giữa thanh ghi và các biến ( variables ) trong ngôn ngữ lập trình cấp cao là số lượng thanh ghi có giới hạn và thường khá ít. Bộ xử lý MIPS chuẩn có tất cả 32 thanh ghi mỗi thanh ghi có 32 bit.

Tên gọi, số thanh ghi và chức năng được thể hiện bên dưới:

<b>Thanh ghi</b>	<b>Chỉ số</b>	<b>Mục đích sử dụng</b>
\$zero	0	Luôn bằng 0, và là thanh ghi chỉ đọc
\$at	1	Dành riêng cho trình hợp dịch (Assembler)
\$v0 - \$v1	2-3	Dùng chứa dữ liệu trả về (returned values) của hàm
\$a0 - \$a3	4-7	Dùng truyền tham số (parameters) cho các hàm
\$t0 - \$t7	8-15	Dùng lưu trữ các dữ liệu tạm thời
\$s0 - \$s7	16-23	Dùng lưu trữ các biến dữ liệu
\$t8 - \$t9	24-25	Dùng lưu trữ các dữ liệu tạm thời
\$k0 - \$k1	26-27	Dành riêng cho hệ điều hành
\$gp	28	Con trỏ toàn cục (global pointer)
\$sp	29	Con trỏ chòng dữ liệu ( stack pointer)
\$fp	30	Con trỏ khung dữ liệu ( frame pointer)
\$ra	31	Địa chỉ trả về ( returned address) và chỉ được cập nhật bởi phần cứng

## Bảng 2.1. Các thanh ghi trong kiến trúc Mips

### 2.2 Toán hạng bộ nhớ

Vì xử lý chỉ có thể lưu trữ một lượng nhỏ dữ liệu trong các thanh ghi, trong khi bộ nhớ máy tính chứa hàng triệu dữ liệu. Với những dữ liệu ở dạng phức hợp ( complex data structure ) như dãy (arrays) hay cấu trúc ( structures), những dạng này chứa nhiều thành phần dữ liệu khác nhau với số lượng nhiều hơn số thanh ghi trong một bộ xử lý. Do đó, cấu trúc này phải được chứa trong bộ nhớ.

Trong kiến trúc MIPS, các ô nhớ (byte) trong bộ nhớ sẽ có các địa chỉ xác định. Địa chỉ một từ nhớ “word” ( là sự kết hợp của 4 byte liên tục ) sẽ là địa chỉ của ô nhớ thấp nhất trong từ nhớ đó và địa chỉ này phải chia hết cho 4. Ngoài ra, MIPS còn hỗ trợ truy xuất “nửa” từ nhớ (half word), tức là 2 byte cho mỗi lần chuyển dữ liệu.

Ví dụ: sự kết hợp của 4 byte có địa chỉ lần lượt là 8,9,10,11 tạo thành một từ nhớ (word) có địa chỉ là 8. Trong khi đó, 4 byte có địa chỉ 10,11,12,13 không thể tạo thành một từ nhớ vì byte có địa chỉ thấp nhất là 10 không chia hết cho 4.

Toán hạng bộ nhớ có định dạng như sau:

< offset > (< base register >)

Trong đó:

- < offset > bắt buộc là một số nguyên và có thể bằng 0.
- < base register > là một thanh ghi dữ liệu bất kỳ chứa địa chỉ cơ sở của cấu trúc phức hợp.

Ô nhớ được truy xuất sẽ có địa chỉ tuyệt đối được tính bằng công thức:

$$\text{Memory location} = < \text{offset} > + < \text{base register} >$$

Ví dụ: Xác định toán hạng bộ nhớ nếu muốn truy xuất ô nhớ có địa chỉ 12, tức là ô nhớ đang chứa giá trị 0x9A. Giả sử thanh ghi \$s0 chứa giá trị 8, ô nhớ đang chứa giá trị 0x12.

Địa chỉ	8	9	10	11	12	13	14
Giá trị	0x12	0x34	0x56	0x78	0x9A	0xBC	0xDE

Ta có thanh ghi \$s0 làm thanh ghi chứa địa chỉ cơ sở thì địa chỉ cơ sở sẽ là 8. Ô nhớ cần truy có địa chỉ 12, theo công thức ta được:

$$<\text{offset}> = 12 - 8 = 4$$

Vậy để truy xuất ô nhớ có địa chỉ 12, tức là ô nhớ đang chứa giá trị 0x9A, toán hạng bộ nhớ được dùng để truy xuất sẽ có dạng  $4($s0)$ .

Kiến trúc MIPS chuẩn hỗ trợ các đường tín hiệu địa chỉ truy xuất bộ nhớ là 32 bit. Do đó tổng số ô nhớ (byte) bộ nhớ mà MIPS có thể quản lý là 4GB.

### 2.3 Toán hạng hằng

Các chương trình dù thực hiện bằng ngôn ngữ và mức trừu tượng nào cũng cần rất nhiều các hằng số nguyên, ví dụ tăng giảm số vòng lặp *for*. Đối với kiến trúc MIPS, một nửa số lệnh trong các chương trình mẫu SPEC CPU2006 có sử dụng các hằng số làm toán hạng. Với các toán hạng phía trên, muốn xử lý các hằng số này thì phải lưu trữ chúng trong bộ nhớ rồi thông qua các lệnh để truy xuất. Tuy nhiên, cách này khá tốn thời gian vì tốc độ bộ xử lý nhanh hơn rất nhiều lần tốc độ truy xuất bộ nhớ.

Do đó, MIPS cho phép các hằng số nguyên ( dương hoặc âm) được tham gia vào một số lệnh như là một toán hạng. Các số nguyên tham gia làm toán hạng sẽ được mã hóa trực tiếp thành các bit nhị phân trong mã máy. Vì vậy, việc tính toán sẽ rất nhanh chóng và thuận tiện thay vì cách truy xuất bộ nhớ tương ứng cho các số nguyên.

## 2.4 Các định dạng lệnh MIPS

Máy tính hay các bộ xử lý chỉ có thể hiểu được các lệnh mã máy là chuỗi các số nhị phân. Do đó, các lệnh ở dạng hợp ngữ phải được mã hóa thành các dạng lệnh mã máy.

Để thực hiện việc mã hóa thì tất cả các thành phần của lệnh sẽ cần phải chuyển sang dạng nhị phân theo những định dạng nhất định. Tập lệnh MIPS có ba định dạng khác nhau, tất cả các định dạng đều dùng 32 bit để mã hóa lệnh hợp ngữ.

### 2.4.1 Định dạng R

Định dạng R ( Register) là định dạng dùng để mã hóa các lệnh mà tất cả các toán hạng của nó đều là thanh ghi. Định dạng R có tất cả 6 trường:

opcode	rs	rt	rd	shamt	funct
6 bit	5 bit	5 bit	5 bit	5 bit	6 bit

Bảng 2.2 Cấu trúc định dạng lệnh R

- opcode: có kích thước 6 bit kể cả với hai định dạng còn lại, dùng để định nghĩa các tác vụ của lệnh. Đối với định dạng R thì các giá trị trong trường opcode luôn bằng 0.
- rs: Chỉ số thanh ghi toán hạng nguồn thứ nhất
- rt: Chỉ số thanh ghi toán hạng nguồn thứ hai.
- rd: Chỉ số thanh ghi toán hạng đích.
- shamt: Chỉ dùng trong các câu lệnh dịch bit (shift) - chứa số lượng bit cần dịch (không được sử dụng sẽ chứa giá trị 0).
- funct: Vì opcode luôn bằng 0 nên cần thêm trường funct kết hợp với opcode để cho biết mã máy là lệnh gì.

## 2.4.2 Định dạng I

Lệnh định dạng I là định dạng dùng để mã hóa các lệnh có một toán hạng là số nguyên hay có sự tham gia của số nguyên.

opcode 6 bit	rs 5 bit	rt 5 bit	constant/ address 16 bit
-----------------	-------------	-------------	-----------------------------

Bảng 2.3 Cấu trúc định dạng lệnh I

opcode: cho biết đây là lệnh gì, vì lệnh định dạng I không có trường funct nên các lệnh định dạng I không dùng chung opcode như các lệnh R-format.

rs: thanh ghi toán hạng nguồn.

rt: thanh ghi toán hạng đích.

constant: một giá trị hằng số mà lệnh sử dụng.

## 2.4.3 Định dạng J

Lệnh định dạng J dùng để mã hóa các lệnh rẽ nhánh không điều kiện( nhảy), có 2 trường trong định dạng này:

opcode 6 bit	target address 26 bit
-----------------	--------------------------

Bảng 2.4 Cấu trúc định dạng lệnh J

- opcode: định nghĩa các tác vụ.
- target address: dùng để biểu diễn địa chỉ của lệnh( nhãn) được nhảy tới.

## 2.4.4 Tập lệnh MIPS (một phần) dùng cho thiết kế

R-type Instructions	Opcode/Function	Syntax	Operation

ADD	100000	f \$d, \$s, \$t	\$d = \$s + \$t
SUB	100010	f \$d, \$s, \$t	\$d = \$s - \$t
AND	100100	f \$d, \$s, \$t	\$d = \$s & \$t
OR	100101	f \$d, \$s, \$t	\$d = \$s   \$t
SLT	101010	f \$d, \$s, \$t	\$d = (\$s < \$t)

**Bảng 2.5 Tập hợp các lệnh dạng R**

I-type Instructions	Opcode/Function	Syntax	Operation
ADDI	001000	o \$d, \$s, i	\$d = \$s + SE(i)
ANDI	001100	o \$d, \$s, i	\$t = \$s & ZE(i)
LW	100011	o \$t, i (\$s)	\$t = MEM [\$s + i]:4
SW	101011	o \$t, i (\$s)	MEM [\$s + i]:4 = \$t
BEQ	000100	o \$s, \$t, label	if (\$s == \$t) pc += i << 2
BNE	000101	o \$s, \$t, label	if (\$s != \$t) pc += i << 2

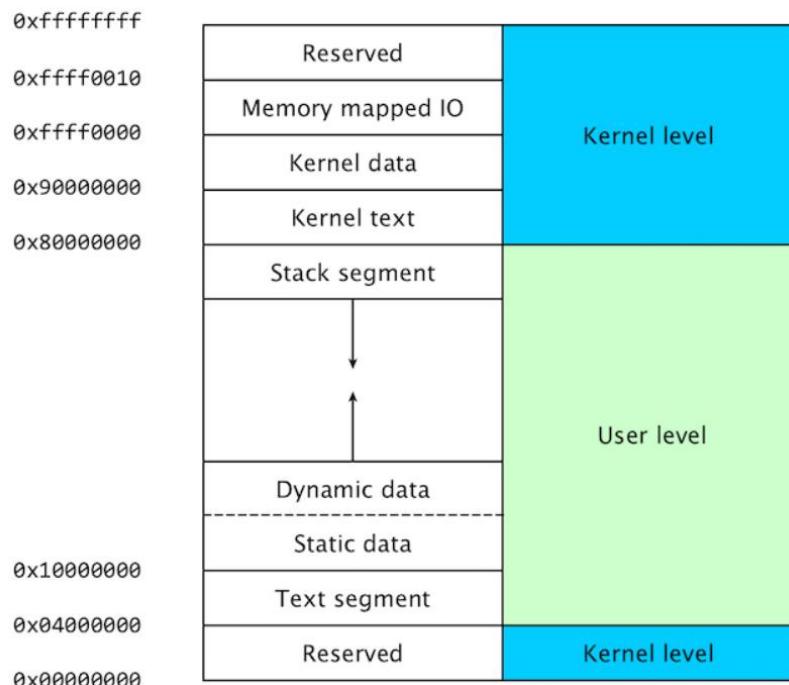
**Bảng 2.6 Tập hợp các lệnh dạng I**

J-type Instructions	Opcode/Function	Syntax	Operation
J	000010	o label	$pc += i \ll 2$
JAL	000011	o label	$\$31 = pc; pc += i \ll 2$

Bảng 2.7 Tập hợp các lệnh dạng J

## 2.5 Bố cục bộ nhớ (Memory map)

Để thực thi một chương trình MIPS, bộ nhớ phải được cấp phát. Bộ xử lý MIPS có thể địa chỉ hóa 4 Gbyte bộ nhớ, từ địa chỉ 0x0000 0000 đến 0xffff ffff. Bộ nhớ người dùng giới hạn trong các vị trí dưới 0x7fff ffff. Bố cục của bộ nhớ được cấp phát cho một chương trình MIPS được biểu diễn như sau:



Hình 2.1 Bố cục bộ nhớ

Trong đó:

- User level: phân vùng của người dùng.

- Kernel level: phân vùng dành riêng cho hệ điều hành, xử lý các ngoại lệ và ngắt
- Chương trình của người dùng sẽ được lưu trữ trong vùng Text segment.
- Static data ( dữ liệu biết được tại thời điểm biên dịch) được sử dụng bởi chương trình người dùng.
- Dynamic data (dữ liệu được cấp phát trong quá trình chạy) bởi chương trình người dùng được lưu trữ trong heap.
- Stack (ngăn xếp) được sử dụng bởi chương trình người dùng để lưu trữ dữ liệu tạm thời, các giá trị trả về khi gọi hàm con.
- Static data được sử dụng bởi kernel sẽ được lưu trữ trong vùng kernel data.
- Các thanh ghi được ánh xạ bộ nhớ cho các thiết bị IO được lưu trữ trong đoạn memory mapped IO.

## CHƯƠNG 3. THIẾT KẾ BỘ XỬ LÍ MIPS 32 BITS PIPELINE

### 3.1 Mô hình thực thi lệnh

Quá trình thực thi các lệnh MIPS nói riêng và các lệnh máy của một máy tính nói chung là tương đối giống nhau nếu xét về các giai đoạn thực thi. Tuy nhiên, nét xét về

một cách chi tiết các công việc hay các tác vụ và thành phần tham gia vào các giai đoạn thì nhóm lệnh khác nhau sẽ khác nhau.

Đối với kiến trúc MIPS, tất cả các lệnh được thực thi đều có hai giai đoạn đầu có hành vi và sự tham gia của những thành phần giống nhau:

- Giai đoạn đọc lệnh ( instruction fetch) : ở giai đoạn này, nội dung thanh ghi bộ đếm chương trình PC sẽ được gửi đến bộ nhớ lệnh ( instruction memory), nơi chứa các lệnh chương trình ở dạng mã máy. Sau khi lệnh trong bộ nhớ đã được đọc, giá trị thanh ghi PC sẽ được tăng lên 4.
- Giai đoạn giải mã lệnh ( instruction decode) : ở giai đoạn này, lệnh sẽ được phân tích và chỉ số thanh ghi nguồn sẽ được trích xuất. Chỉ số thanh ghi nguồn này dùng để truy xuất tập thanh ghi ( register file) nhằm lấy các giá trị đang lưu trong thanh ghi thực hiện tính toán lệnh.

Sau hai giai đoạn đọc lệnh và giải mã lệnh, các nhóm lệnh khác nhau sẽ yêu cầu các tác vụ và khối chức năng khác nhau. Ngược lại, các lệnh trong cùng một nhóm lệnh sẽ có các tác vụ và các khối chức năng tham gia vào tính toán tương tự nhau.

Đối với các lệnh số học, sau khi hai giá trị chứa trong hai thanh ghi nguồn được trích xuất sẽ được đưa vào khối luận lý số học (ALU) để thực hiện phép tính tương ứng. Trong trường hợp các lệnh có mặt tham gia của số nguyên thì giá trị của thanh ghi nguồn và giá trị số nguyên trích xuất từ trường offset sẽ được đưa vào khối ALU. Giai đoạn này gọi là giai đoạn thực thi lệnh ( instruction execute). Sau khi khối ALU thực thi xong thì kết quả sẽ được cập nhật vào thanh ghi đích trong tập thanh ghi, đây là giai đoạn cập nhật kết quả ( write back).

Đối với các lệnh truy xuất dữ liệu, trước tiên phải tính toán địa chỉ dữ liệu cần truy xuất theo công thức ( thực hiện phép cộng giữa thanh ghi địa chỉ nền và giá trị độ dời ). Nội dung thanh ghi địa chỉ nền và giá trị độ dời được trích xuất từ lệnh, hai giá trị này

được đưa vào khối ALU để tính toán địa chỉ cần truy xuất. Địa chỉ này cung cấp cho bộ nhớ dữ liệu ( data memory ) để tiến hành truy xuất, đây là giai đoạn truy xuất bộ nhớ. Với các lệnh đọc dữ liệu, giá trị này sẽ được gửi đến bộ nhớ và dữ liệu tại các ô nhớ sẽ được trích xuất rồi ghi vào tập thanh ghi. Ngược lại, với các lệnh ghi dữ liệu thì giá trị địa chỉ này cùng với nội dung trong thanh ghi sẽ được gửi đến bộ nhớ dữ liệu. Bộ nhớ dữ liệu sẽ cập nhật các ô nhớ tương ứng với giá trị chứa trong thanh ghi toán hạng nguồn.

Đối với lệnh rẽ nhánh có điều kiện thì hai thanh ghi toán hạng nguồn sau khi được truy xuất sẽ được đưa vào khối ALU để thực hiện phép trừ. Nếu kết quả bằng 0 thì hai thanh ghi chứa giá trị bằng nhau, ngược lại giá trị hai thanh ghi khác nhau. Dựa vào kết quả và điều kiện của lệnh rẽ nhánh sẽ thực hiện rẽ nhánh khi bằng hay khác nhau mà giá trị thanh ghi PC sẽ được cập nhật lại bằng giá trị địa chỉ đích hay sẽ giữ nguyên giá trị.

### 3.2 Các thành phần cơ bản của bộ xử lý

Bộ xử lý theo kiến trúc MIPS gồm các thành phần cơ bản sau:

- Bộ nhớ lệnh.
- Tập thanh ghi.
- Khối ALU.
- Bộ nhớ dữ liệu.
- Bộ điều khiển ( controller ).

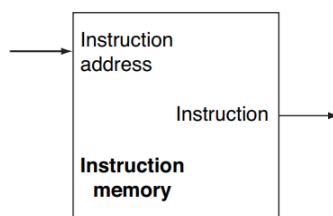
Các thành phần bộ nhớ lệnh, tập thanh ghi, khối ALU và bộ nhớ dữ liệu sẽ tham gia vào việc thực hiện lệnh và xử lý dữ liệu liên quan đến lệnh đó, các thành phần này cấu tạo nên một đường dữ liệu ( datapath ) của bộ xử lý. Ngoài ra, bộ điều khiển sẽ tạo ra các tín hiệu để quyết định các thành phần nào được tham gia hoạt động khi bộ xử lý đang thực hiện một lệnh cụ thể.

#### 3.2.1 Đường dữ liệu

Đường dữ liệu bao gồm các tín hiệu dữ liệu và các khối chức năng. Nhiệm vụ chính của đường dữ liệu là xử lý dữ liệu dưới sự điều khiển của bộ điều khiển và truyền dữ liệu qua lại giữa các khối chức năng.

### Bộ nhớ dữ liệu

Bộ nhớ lệnh là nơi chứa toàn bộ các lệnh của các chương trình đang được thực thi. Các lệnh được chứa tuần tự trong bộ nhớ lệnh theo mô hình lưu trữ mà bộ xử lý đó đang sử dụng (little endian hoặc big endian). Kiến trúc MIPS chuẩn có các thanh ghi đều 32 bit nên thanh ghi PC chứa giá trị của lệnh sẽ được thực thi cũng có kích thước 32 bit.

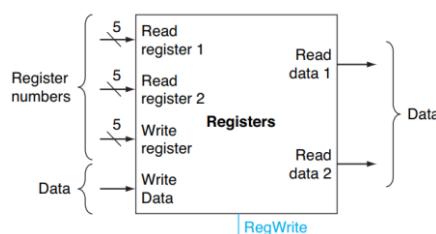


**Hình 3.0.1 Sơ đồ khối bộ nhớ dữ liệu**

Ngõ vào địa chỉ *Instruction\_address* của khối bộ nhớ có kích thước 32 bit. Các lệnh chuẩn MIPS đều có kích thước 32 bit nên ngõ ra *Instruction* sẽ có kích thước 32 bit. Bộ nhớ lệnh tham gia duy nhất vào giai đoạn đọc lệnh trong quá trình thực thi lệnh.

### Tập thanh ghi

Tập thanh ghi trong kiến trúc MIPS là nơi chứa tất cả 32 thanh ghi đa dụng. Vì vậy, các thanh ghi toán hạng nguồn 1 và 2 cần phải có 5 bit để quản lý và truy xuất dữ liệu.



**Hình 3.0.2 Sơ đồ khối tập thanh ghi**

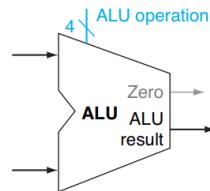
Các lệnh MIPS truy xuất tối đa được hai thanh ghi tại một thời điểm vì vậy cần phải có hai ngõ vào (read register 1 và read register 2). Trong trường hợp bộ xử lý thực

hiện lệnh chỉ cần truy xuất một thanh ghi thì ngõ vào thứ hai ( read register 2) không cần quan tâm. Tương ứng với hai ngõ vào sẽ có hai ngõ ra chứa giá trị của dữ liệu ( data 1 và data 2), hai ngõ ra này đều có cùng kích thước là 32 bit.

Do các thanh ghi trong tập thanh ghi sẽ được cập nhật lại giá trị khi thực hiện các lệnh số học và đọc dữ liệu. Vì vậy, ngoài các ngõ vào ra để truy xuất các thanh ghi nguồn thì cần phải có thêm các ngõ vào để hỗ trợ việc cập nhật giá trị của thanh ghi. Tuy nhiên, việc cập nhật sẽ làm thay đổi giá trị thanh ghi nên cần phải thực hiện cẩn thận và đúng thời điểm. Vì thế, việc cập nhật cần phải được điều khiển bởi một tín hiệu chuyên biệt thông qua ngõ vào *RegWrite* có kích thước 1 bit. Khi tín hiệu bằng 1, thì thanh ghi đích có chỉ số *Write\_register* sẽ được cập nhật bằng giá trị ngõ vào *WriteData*. Ngược lại, sẽ không thực hiện cập nhật dù cho hai ngõ vào *Write\_register* và *WriteData* có giá trị là gì.

### **Khối ALU**

Khối ALU thực hiện những phép toán số học để phục vụ các lệnh số học, lệnh luận lý, lệnh truy xuất dữ liệu, lệnh rẽ nhánh có điều kiện và lệnh so sánh.

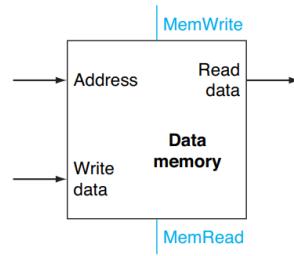


**Hình 3.0.3 Sơ đồ khối ALU**

Khối ALU có hai ngõ vào chứa giá trị toán hạng nguồn 1 và toán hạng nguồn 2, cả hai đều có kích thước 32 bit. Hai ngõ ra *ALUresult* và *Zero* có kích thước lần lượt là 32 bit và 1 bit. Trong đó, ngõ ra *ALUresult* chứa giá trị kết quả phép toán. Ngõ ra *Zero* bằng 1 nếu giá trị ngõ ra *ALUresult* bằng 0, ngược lại ngõ ra *Zero* bằng 0.

Khối ALU phải có khả năng thực hiện nhiều phép tính toán khác nhau như cộng, trừ và các phép toán luận lý. Ngõ vào *ALU\_operation* có kích thước 4 bit, sẽ được điều khiển bởi bộ điều khiển để quyết định phép toán mà bộ ALU thực hiện tại một thời điểm.

## Bộ nhớ dữ liệu



Hình 3.0.4 Sơ đồ khái niệm bộ nhớ dữ liệu

Bộ nhớ dữ liệu là nơi lưu trữ dữ liệu của các chương trình đang thực thi. Dữ liệu được lưu trữ tuần tự trong các ô nhớ có địa chỉ xác định và tuân thủ theo quy tắc little endian hoặc big endian của kiến trúc bộ xử lý đang sử dụng. Khác với bộ nhớ lệnh, bộ nhớ dữ liệu ngoài việc truy xuất đọc bộ nhớ còn được truy xuất để cập nhật dữ liệu. Do đó, ngoài ngoài vào *Address* 32 bit và ngoài ra *Read\_data* 32 bit có chức năng tương tự bộ nhớ lệnh, bộ nhớ dữ liệu còn có thêm ngoài vào *Write\_data* 32 bit để ghi giá trị vào bộ nhớ trong trường hợp thực hiện lệnh lưu trữ dữ liệu. Khi thực hiện lưu trữ dữ liệu, ngoài vào *Address* chứa địa chỉ ô nhớ sẽ chứa dữ liệu và ngoài vào *Write\_data* chứa giá trị của dữ liệu lưu trữ.

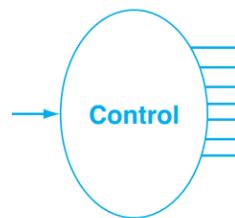
Tương tự như cập nhật thanh ghi, việc cập nhật dữ liệu cũng cần một tín hiệu điều khiển riêng là *MemWrite*. Tín hiệu này sinh ra từ bộ điều khiển và cho phép cập nhật dữ liệu khi tín hiệu bằng 1, ngược lại không được phép cập nhật.

Một điểm khác biệt giữa tập thanh ghi là không phải lúc nào bộ nhớ cũng cho phép đọc như tập thanh ghi. Trong quá trình thực thi, mặc dù không thực thi những lệnh truy xuất bộ nhớ nhưng ngoài vào *Address* luôn luôn có giá trị. Giá trị này được dùng để truy xuất ô nhớ tại địa chỉ tương ứng và dữ liệu ô nhớ sẽ được gửi đến ngoài ra. Tuy nhiên, trong bộ nhớ dữ liệu của bộ xử lý sẽ có những vùng nhớ không được truy xuất từ các chương trình, ví dụ vùng nhớ của hệ điều hành. Mọi truy xuất đến vùng nhớ này đều gây ra các lỗi ngoại lệ và phải được xử lý. Do đó, để tránh trường hợp vô tình gây ra lỗi hệ thống do truy xuất đến vùng nhớ không được phép truy xuất, cần phải có thêm tín hiệu

*MemRead* để điều khiển việc truy xuất dữ liệu. Khi thực hiện lệnh truy xuất thì tín hiệu *MemRead* sẽ bằng 1 và cho phép truy xuất, ngược lại tín hiệu sẽ bằng 0 và không được thực hiện việc truy xuất.

### 3.2.2 Bộ điều khiển

Bộ điều khiển chịu trách nhiệm phân tích lệnh đang thực thi và sinh ra các tín hiệu để điều khiển hoạt động của các khối chức năng. Ngoài các tín hiệu *RegWriter* (*ghi thanh ghi*), *MemRead* (*đọc bộ nhớ*), *MemWriter* (*ghi bộ nhớ*), bộ điều khiển còn sinh ra các tín hiệu để xác định tác vụ của khối ALU và để lựa chọn các đường dữ liệu.

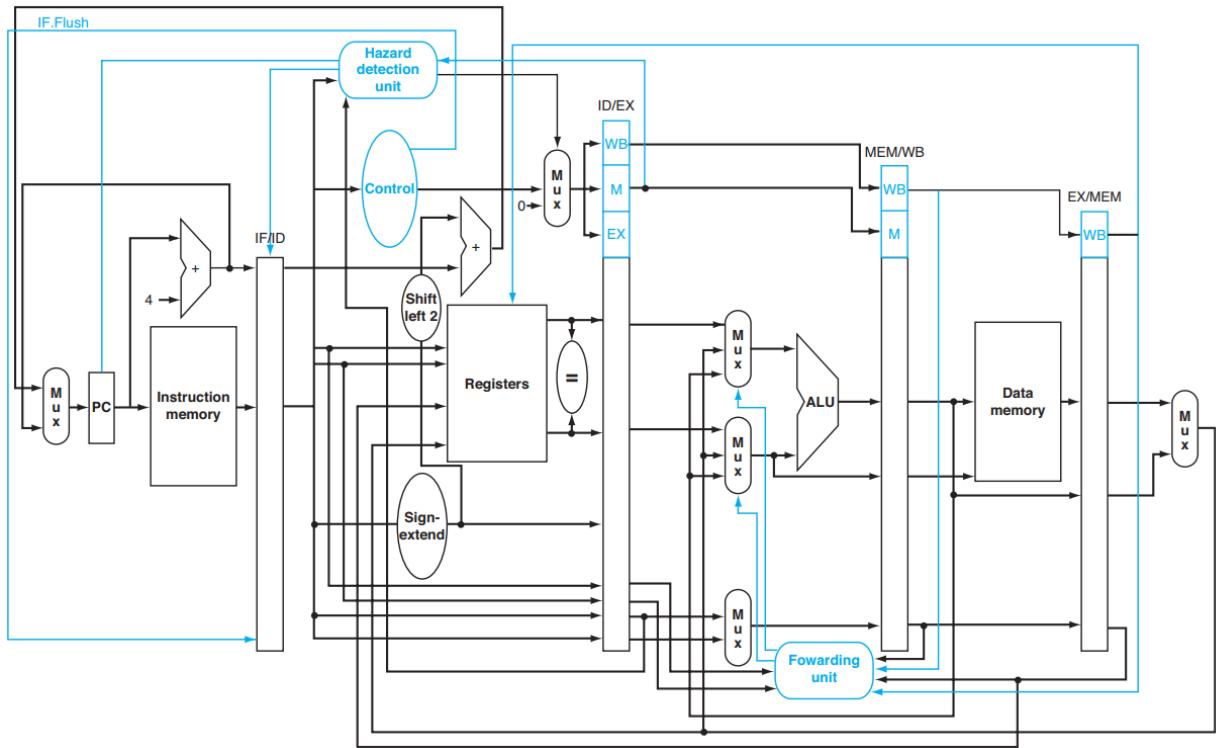


**Hình 3.0.5 Sơ đồ khái niệm bộ điều khiển**

Các lệnh có định dạng giống nhau sẽ có quá trình thực thi tương tự nhau. Ví dụ, với định dạng R đều truy xuất hai thanh ghi và cập nhật một thanh ghi. Do đó, cần sử dụng trường Opcode của lệnh là đủ để xác định hành vi của lệnh đó và sinh ra các tín hiệu điều khiển.

## 3.3 Thiết kế tổng quan

Thiết kế tổng quan bộ xử lý MIPS 32 bits Pipeline được thể hiện trên hình 3.5.



**Hình 3.0.6 Thiết kế tổng quan bộ xử lý MIPS 32 bits pipeline**

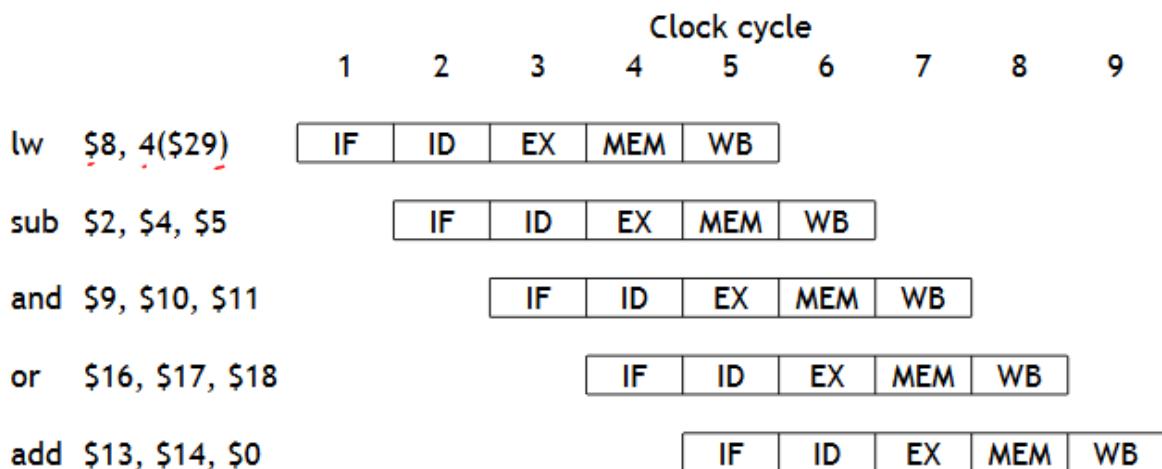
Khác với bộ xử lý đơn xung nhịp khi các lệnh đều được thực hiện xong trong một chu kỳ máy, bộ xử lý pipeline thực hiện 1 lệnh trong 5 stages, mỗi stage thực hiện trong 1 chu kỳ máy:

- Instruction fetch (IF): Nạp lệnh và cập nhật giá trị thanh ghi PC xác định địa chỉ lệnh tiếp theo
- Instruction decode (ID): Giải mã lệnh, xác định toán tử thực thi lệnh, xác định toán hạng bằng cách đọc tệp thanh ghi từ địa chỉ cho trong lệnh
- Execution (EX): Thực thi phép toán bằng ALU
- Memory (MEM): Đọc hoặc ghi dữ liệu trên bộ nhớ
- Write back (WB): Ghi dữ liệu vào tệp thanh ghi

Bộ xử lý pipeline cải thiện hiệu năng theo thông lượng. Câu lệnh sau không cần đợi câu lệnh trước hoàn tất mới bắt đầu thực hiện mà mỗi stage sẽ được thực hiện liên tiếp bởi các thanh ghi pipeline. Có tất cả 4 thanh ghi pipeline giữa 5 stages:

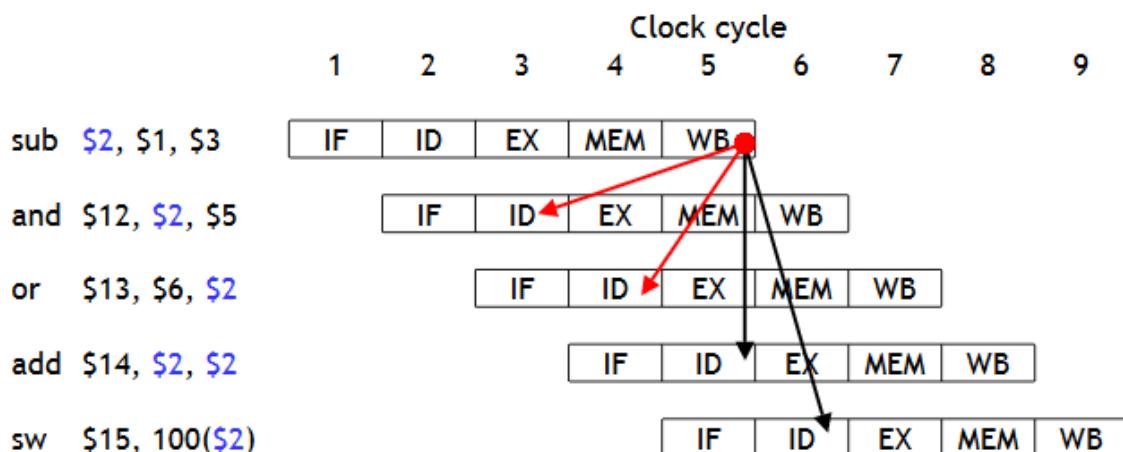
- Thanh ghi IF-ID: Ở giữa IF stage và ID stage
- Thanh ghi ID-EX: Ở giữa ID stage và EX stage
- Thanh ghi EX-MEM: Ở giữa EX stage và MEM stage
- Thanh ghi MEM-WB: Ở giữa MEM stage và WB stage

Nguyên lý thực thi của 1 chương trình thực thi bằng bộ xử lý pipeline được thể hiện trên hình 3.2 với 1 ví dụ chương trình MIPS



**Hình 3.0.7 Nguyên lý thực thi chương trình bằng bộ xử lý pipeline**

Đây là 1 ví dụ đơn giản (lý tưởng) thực thi chương trình bằng bộ xử lý pipeline. Mỗi lệnh cần 5 clock cycles để thực hiện. Trong 1 clock cycle, bộ xử lý thực hiện đồng thời các stages IF, ID, EX, MEM, WB của các lệnh liên tiếp. Đoạn code này là lý tưởng, không có sự phụ thuộc giữa các lệnh với nhau, do đó không xảy ra xung đột. Xét 1 chương trình có xung đột (hình 3.3).



**Hình 3.0.8 Chương trình gây xung đột trong bộ xử lý pipeline**

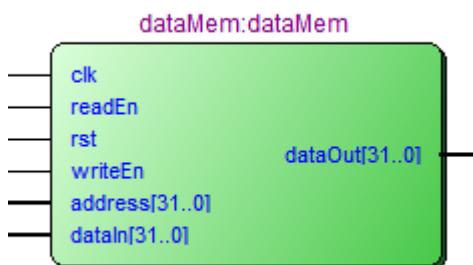
Các mũi tên màu đỏ chỉ ra xung đột dữ liệu, khi mà lệnh đầu tiên ở WB stage mới cập nhật giá trị thanh ghi \$2, trong khi 2 lệnh sau cần truy cập tệp thanh ghi để lấy giá trị \$2 ở ID stage. Đây chỉ là một loại xung đột pipeline, có tất cả 3 loại xung đột:

- Xung đột cấu trúc: Tài nguyên (bộ nhớ, tệp thanh ghi) được truy cập cùng lúc ở nhiều stage.

- Xung đột dữ liệu: Phụ thuộc dữ liệu giữa các lệnh, lệnh sau cần đợi lệnh trước hoàn thành việc ghi dữ liệu.
- Xung đột điều khiển: Xác định lệnh tiếp theo phụ thuộc vào lệnh trước. Để giải quyết xung đột, có thể dùng 2 phương pháp:
  - Phương pháp dừng và chờ: Chèn các chu kỳ đợi vào giữa các lệnh có xung đột
  - Phương pháp chuyển tiếp dữ liệu: Chuyển dữ liệu đến stage cần sử dụng của lệnh sau ngay khi có kết quả từ lệnh trước

### 3.4 Bộ nhớ dữ liệu

Bộ nhớ dữ liệu sử dụng trong bài tập lớn này là bộ nhớ dung lượng 1 KB, gồm các ngăn nhớ 8 bits. Vì bộ xử lý là 32 bits nên mỗi lần truy cập bộ nhớ sẽ đọc hoặc ghi trên 4 ngăn nhớ liên tiếp với địa chỉ cơ sở là 0, 4, 8, ...1000. Thiết kế bộ nhớ dữ liệu được thể hiện trên hình 3.4.



**Hình 3.0.9 Bộ nhớ dữ liệu**

Các đầu vào gồm:

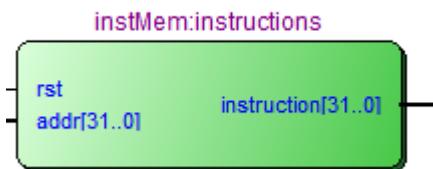
- clk, rst: Xung clock và tín hiệu reset các ô nhớ về 0
- readEn, writeEn: Tín hiệu cho phép đọc hoặc ghi bộ nhớ
- address (32 bits): Địa chỉ truy cập bộ nhớ
- dataIn (32 bits): Dữ liệu ghi vào bộ nhớ

Đầu ra chỉ có:

- dataOut (32 bits): Dữ liệu đọc ra từ bộ nhớ

### 3.5 Bộ nhớ lệnh

Bộ nhớ lệnh cũng tương tự bộ nhớ dữ liệu, có dung lượng 1 KB, gồm các ngăn nhớ 8 bits. Mỗi lần truy cập bộ nhớ lệnh để đọc lệnh 32 bits ra sẽ đọc trên 4 ngăn nhớ liên tiếp. Thiết kế bộ nhớ lệnh được thể hiện trên hình 3.5.



Hình 3.0.10 Bộ nhớ lệnh

Các đầu vào gồm:

- rst: Khởi tạo bộ nhớ lệnh
- addr (32 bits): Địa chỉ truy cập bộ nhớ lệnh

Đầu ra chỉ có:

- instruction (32 bits): Lệnh đọc ra

Các lệnh đã được viết sẵn trong code vào bộ nhớ lệnh, theo quy ước chuyển mã MIPS sang mã máy cho các lệnh loại R, I, J. Khi triển khai, bộ xử lý sẽ thực hiện lần lượt các lệnh bắt đầu từ địa chỉ 0.

### 3.6 Tệp thanh ghi

Tệp thanh ghi cũng là 1 kiểu bộ nhớ, với dung lượng 32 (32 thanh ghi trong MIPS), mỗi ngăn nhớ 32 bits (độ lớn 1 thanh ghi). Vì vậy, thiết kế tệp thanh ghi cũng tương tự bộ nhớ dữ liệu và bộ nhớ lệnh, được thể hiện trên hình 3.6.



Hình 3.0.11 Tệp thanh ghi

Các đầu vào gồm:

- clk, rst: Xung clock và tín hiệu reset giá trị các thanh ghi về 0
- writeEn: Tín hiệu cho phép ghi vào tệp thanh ghi
- src1, src2, dest (5 bits): Địa chỉ truy cập tệp thanh ghi (rs, rt, rd)
- writeVal (32 bits): Giá trị ghi vào tệp thanh ghi

Tệp thanh ghi gồm 2 đầu ra:

- reg1, reg2 (32 bits): Giá trị 2 thanh ghi đọc ra để làm 2 toán hạng đưa vào ALU

### 3.7 Thanh ghi PC

Thanh ghi PC 32 bits lưu địa chỉ lệnh cần truy cập vào bộ nhớ lệnh. Thiết kế thanh ghi PC được thể hiện trên hình 3.7.



Hình 3.0.12 Thanh ghi PC

Các đầu vào gồm:

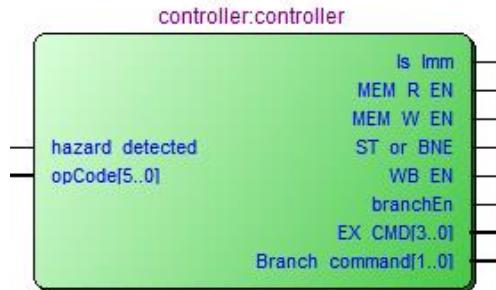
- clk, rst: Xung clock và tín hiệu reset địa chỉ lệnh tiếp theo là 0
- writeEn: Tín hiệu cho phép xác định địa chỉ lệnh tiếp theo
- regIn (32 bits): Xác định địa chỉ lệnh tiếp theo

Đầu ra chỉ có:

- regOut (32 bits): Địa chỉ lệnh cần truy cập vào bộ nhớ lệnh

### 3.8 Bộ điều khiển

Bộ điều khiển nhận 6 bits opCode từ mã máy và phát ra các tín hiệu điều khiển. Thiết kế bộ điều khiển được thể hiện trên hình 3.8



**Hình 3.0.13 Bộ điều khiển**

Các đầu vào gồm:

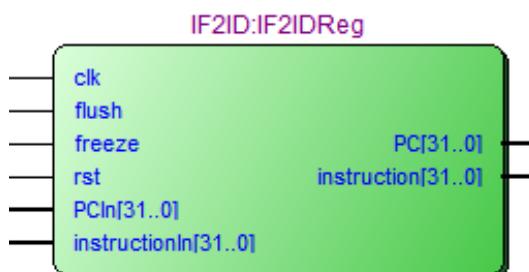
- Hazard\_detected: Xác định có xung đột không
- opCode (6 bits): Mã máy xác định tín hiệu điều khiển

Các đầu ra gồm:

- Is\_Imm: Xác định có phải lệnh loại I không
- MEM\_R\_EN: Tín hiệu cho phép đọc memory
- MEM\_W\_EN: Tín hiệu cho phép ghi memory
- ST\_or\_BNE: Xác định lệnh là store hoặc BNE
- WB\_EN: Tín hiệu cho phép ghi vào tệp thanh ghi
- branchEn: Tín hiệu xác định lệnh có phải rẽ nhánh không
- EX\_CMD (4 bits): Xác định phép toán thực hiện trong ALU
- Branch\_command (2 bits): Xác định loại lệnh rẽ nhánh

### 3.9 Thanh ghi IF-ID

Thanh ghi IF-ID 64 bits nằm ở giữa IF stage và ID stage, chứa 32 bits PC và 32 bits lệnh. Thiết kế thanh ghi IF-ID được thể hiện trên hình 3.9.



**Hình 3.0.14 Thanh ghi IF-ID**

Các đầu vào gồm:

- clk, rst: Xung clock và tín hiệu reset PC và lệnh đầu ra về 0
- flush: Tín hiệu xác định “vỡ đường ống” (khi thực hiện lệnh rẽ nhánh)
- freeze: Tín hiệu xác định có xung đột hay không
- PCIn, InstructionIn (32 bits): Giá trị PC và lệnh đưa vào từ IF Stage

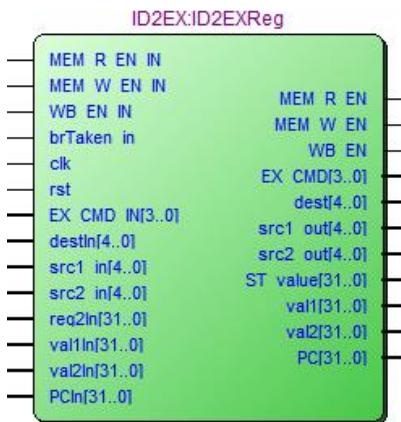
Các đầu ra gồm:

- PC (32 bits): Giá trị PC đưa vào thanh ghi EX-MEM
- instruction (32 bits): Địa chỉ lệnh đưa vào ID Stage

Nếu không có xung đột và lệnh không phải rẽ nhánh, các giá trị PC và instruction được lấy ra từ PCIn và instructionIn, ngược lại thì bằng 0.

### 3.10 Thanh ghi ID-EX

Thanh ghi ID-EX 151 bits nằm ở giữa ID Stage và EX Stage, chứa 8 bits điều khiển, 10 bits rt và rd, 5 bits (rd hoặc 0) địa chỉ thanh ghi dùng trong chuyển tiếp dữ liệu, 64 bits dữ liệu từ 2 thanh ghi đọc ra từ tệp thanh ghi, 32 bits mở rộng dấu và 32 bits PC. Thiết kế thanh ghi ID-EX được thể hiện trên hình 3.10.



Hình 3.0.15 Thanh ghi ID-EX

Các đầu vào gồm:

- MEM\_R\_EN\_IN, MEM\_W\_EN\_IN, WB\_EN\_IN, brTaken\_in: Các tín hiệu điều khiển từ ID Stage
- EX\_CMD\_IN (4 bits): Từ ID Stage, dùng để điều khiển chọn toán hạng ALU
- destIN, src1\_in, src2\_in (5 bits): rt, rd và địa chỉ thanh ghi dùng trong chuyển tiếp dữ liệu, được đưa sang từ thanh ghi IF-ID

- reg2In, val1In, val2In, PCIn (32 bits): 64 bits dữ liệu từ 2 thanh ghi đọc ra từ tệp thanh ghi, 32 bits mở rộng dấu và 32 bits PC

Các đầu ra gồm:

- MEM\_R\_EN, MEM\_W\_EN, WB\_EN: Các tín hiệu điều khiển đưa đến thanh ghi EX-MEM
- brTaken: Tín hiệu điều khiển đưa vào bộ MUX trong IF Stage để chọn địa chỉ lệnh tiếp theo
- EX\_CMD (4 bits): Đưa vào ALU trong EX Stage
- dest (5 bits): Đưa vào thanh ghi EX-MEM
- src1\_out, src2\_out: Đưa vào khối chuyển tiếp dữ liệu
- ST\_value: Giá trị cần lưu vào bộ nhớ dữ liệu trong MEM Stage
- val1, val2: 2 toán hạng đưa vào ALU trong EX Stage
- PC (32 bits): Đưa vào thanh ghi EX-MEM

### 3.11 Thanh ghi EX-MEM

Thanh ghi EX-MEM 104 bits nằm ở giữa EX Stage và MEM Stage, chứa 3 bits điều khiển, 5 bits rt hoặc rd, 32 bits PC, 32 bits kết quả từ ALU, 32 bits giá trị cần lưu vào bộ nhớ dữ liệu. Thiết kế thanh ghi EX-MEM được thể hiện trên hình 3.11.



Hình 3.0.16 Thanh ghi EX-MEM

Các đầu vào gồm

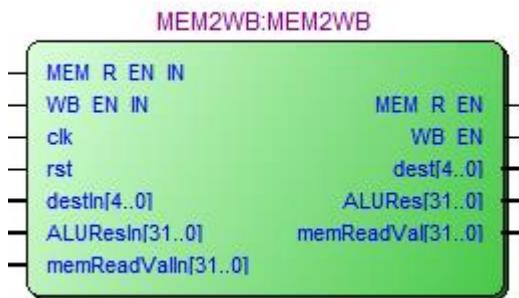
- MEM\_R\_EN\_IN, MEM\_W\_EN\_IN, WB\_EN\_IN: Các tín hiệu điều khiển từ thanh ghi ID-EX
- destIn (5 bits): rt hoặc rd, từ thanh ghi ID-EX
- PCIn (32 bits): Từ thanh ghi ID-EX
- ALURes, STVal (32 bits): Kết quả ALU và giá trị cần lưu vào bộ nhớ dữ liệu từ EX Stage

Các đầu ra gồm:

- MEM\_R\_EN: Đưa vào memory trong MEM Stage và thanh ghi MEM-WB
- MEM\_W\_EN: Đưa vào bộ nhớ dữ liệu trong MEM Stage
- WB\_EN: Đưa vào thanh ghi MEM-WB
- dest (5 bits): Đưa vào thanh ghi MEM-WB
- PC (32 bits): Đưa vào thanh ghi MEM-WB
- ALURes (32 bits): Đưa vào memory trong MEM Stage và thanh ghi MEM-WB
- STVal (32 bits): Đưa vào memory trong MEM Stage

### 3.12 Thanh ghi MEM-WB

Thanh ghi MEM-WB 81 bits, nằm ở giữa MEM Stage và WB Stage, chứa 2 bits điều khiển, 5 bits rt hoặc rd, 32 bits kết quả từ ALU, 32 bits dữ liệu đọc từ memory. Thiết kế thanh ghi MEM-WB được thể hiện trên hình 3.12.



Hình 3.0.17 Thanh ghi MEM-WB

Các đầu vào gồm:

- WB\_EN\_IN, MEM\_R\_EN\_IN: Các tín hiệu điều khiển từ thanh ghi EX-MEM
- destIn (5 bits): rt hoặc rd, từ thanh ghi EX-MEM
- ALUResIn (32 bits): Kết quả thực hiện ALU từ thanh ghi EX-MEM
- memReadValIn (32 bits): Giá trị đọc memory từ MEM Stage

Các đầu ra gồm:

- MEM\_R\_EN: Lựa chọn ALURes hoặc memReadVal làm giá trị write back
- WB\_EN: Đưa vào tệp thanh ghi và khởi chuyển tiếp dữ liệu
- dest (5 bits): Địa chỉ write back, đưa vào tệp thanh ghi
- ALURes, memReadVal (32 bits): Đưa vào tệp thanh ghi

### 3.13 IF Stage

Trong IF Stage, bộ xử lí lấy ra lệnh cần thực hiện trong bộ nhớ lệnh và xác định địa chỉ lệnh tiếp theo. Địa chỉ lệnh tiếp theo được xác định như sau:

$$PC_{next} = PC_{current} + 4, \text{ nếu lệnh tuần tự}$$

$$PC_{next} = PC_{current} + 4 + 4 * offset, \text{ nếu lệnh rẽ nhánh}$$

Thiết kế IF Stage được thể hiện trên hình 3.13.



Hình 3.0.18 IF Stage

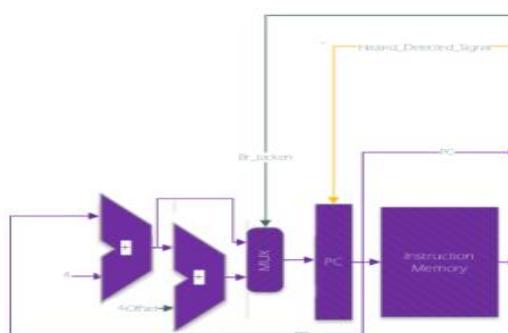
Các đầu vào gồm:

- brTaken: Xác định lệnh có phải lệnh rẽ nhánh không
- freeze: Xác định có xung đột không
- brOffset (32 bits): Xác định địa chỉ lệnh tiếp theo trong lệnh rẽ nhánh

Các đầu ra gồm:

- PC (32 bits): Địa chỉ lệnh cần truy cập vào bộ nhớ lệnh
- instruction (32 bits): Lệnh đọc ra từ bộ nhớ lệnh

Thiết kế chi tiết IF Stage được thể hiện trên hình 3.19

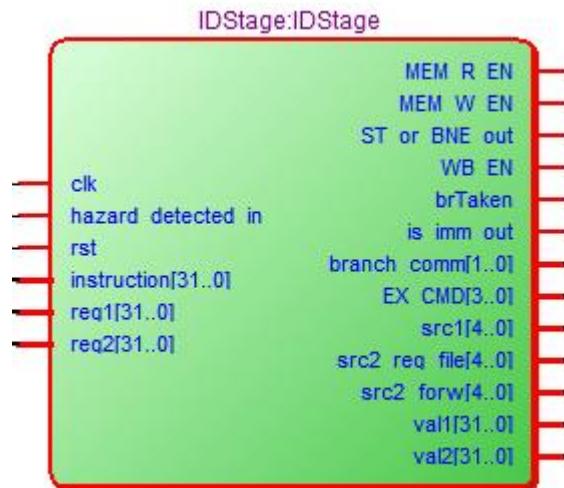


Hình 3.0.19 Thiết kế chi tiết IF Stage

2 bộ cộng 32 bits dùng để xác định  $PC_{current} + 4$  và  $PC_{current} + 4 + 4 * offset$ , bộ MUX 2 to 1 dùng để xác định địa chỉ lệnh tiếp theo với tín hiệu chọn brTaken.

### 3.14 ID Stage

Trong ID Stage, bộ điều khiển phát ra các tín hiệu điều khiển, bộ xử lý truy cập tệp thanh ghi để lấy ra giá trị 2 thanh ghi làm toán hạng ALU. Thiết kế ID Stage được thể hiện trên hình 3.15.



Hình 3.0.20 ID Stage

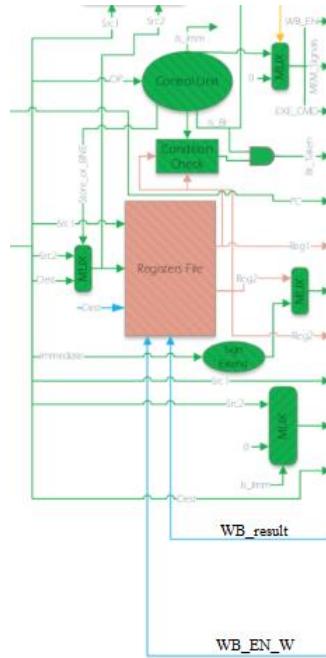
Các đầu vào gồm:

- hazard\_detected\_in: Từ khói phát hiện xung đột, đưa vào bộ điều khiển
- instruction (32 bits): Từ thanh ghi IF-ID
- reg1, reg2 (32 bits): 2 giá trị đọc ra từ tệp thanh ghi, dùng để kiểm tra điều kiện trong lệnh rẽ nhánh

Các đầu ra gồm:

- MEM\_R\_EN, MEM\_W\_EN, ST\_or\_BNE\_out, WB\_EN, brTaken, is\_imm\_out, branch\_comm, EX\_CMD: Các tín hiệu điều khiển phát ra từ bộ điều khiển
- src1 (5 bits): rt, đưa vào tệp thanh ghi và khói phát hiện xung đột
- src2\_req\_file (5 bits): rs hoặc rd, đưa vào tệp thanh ghi
- val1, val2 (32 bits): 2 giá trị đọc ra từ tệp thanh ghi hoặc có 1 giá trị là từ bộ mở rộng dấu, đưa vào thanh ghi ID-EX dùng làm toán hạng đưa vào ALU

Thiết kế chi tiết ID Stage được thể hiện trên hình 3.21



**Hình 3.0.21 Thiết kế chi tiết ID Stage**

Khối kiểm tra điều kiện sử dụng reg1 và reg2 để kiểm tra có thỏa mãn điều kiện trong lệnh rẽ nhánh không. Bộ mở rộng dấu mở rộng số 16 bits thành 32 bits dùng trong lệnh loại I. Các bộ MUX lựa chọn tín hiệu phù hợp tùy theo trường hợp lệnh loại I, lệnh rẽ nhánh, phát hiện xung đột. Các đường tín hiệu màu xanh biển thuộc về WB Stage (sẽ trình bày ở phần WB Stage).

### 3.15 EX Stage

Trong EX Stage, ALU thực hiện phép toán, kết quả có thể dùng để ghi lại vào tệp thanh ghi hoặc xác định địa chỉ truy cập memory. Thiết kế EX Stage được thể hiện trên hình 3.17.



### Hình 3.0.22 EX Stage

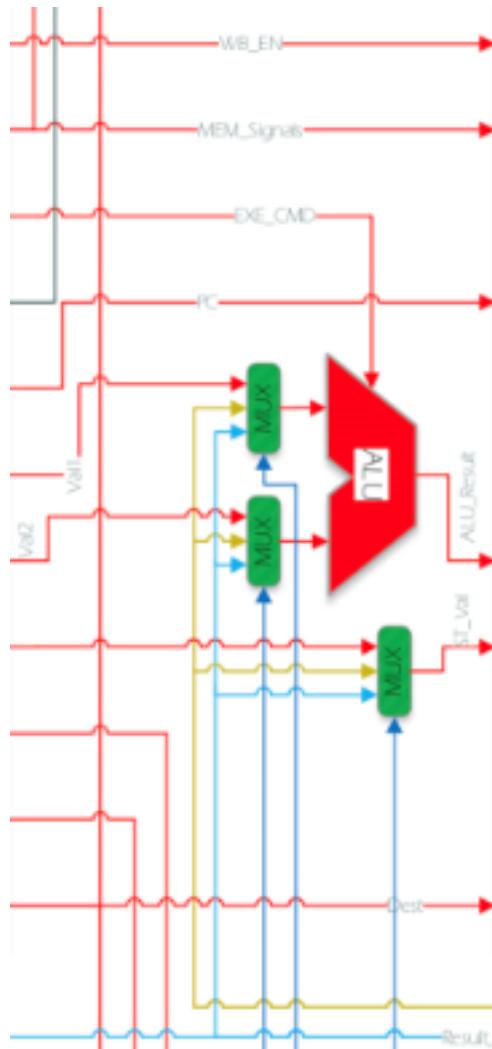
Các đầu vào gồm:

- val1\_sel, val2\_sel (2 bits): Tín hiệu chọn dùng cho 2 bộ MUX lựa chọn 2 toán hạng vào ALU khi có chuyển tiếp dữ liệu
- ST\_val\_sel (2 bits): Tín hiệu chọn cho bộ MUX lựa chọn giá trị lưu ghi vào memory
- val1, val2 (32 bits): 2 giá trị từ thanh ghi ID-EX, được đưa vào 2 bộ MUX để làm 2 toán hạng ALU
- ST\_value\_in (32 bits): Đưa vào bộ MUX lựa chọn giá trị ghi vào memory
- ALU\_res\_MEM, result\_WB(32 bits): Kết quả tính toán ALU và giá trị write back được đưa ngược lại 3 bộ MUX kể trên dùng trong chuyển tiếp dữ liệu

Các đầu ra gồm:

- ALUResult, ST\_value\_out (32 bits): Đưa vào thanh ghi EX-MEM

Thiết kế chi tiết EX Stage được thể hiện trên hình 3.23

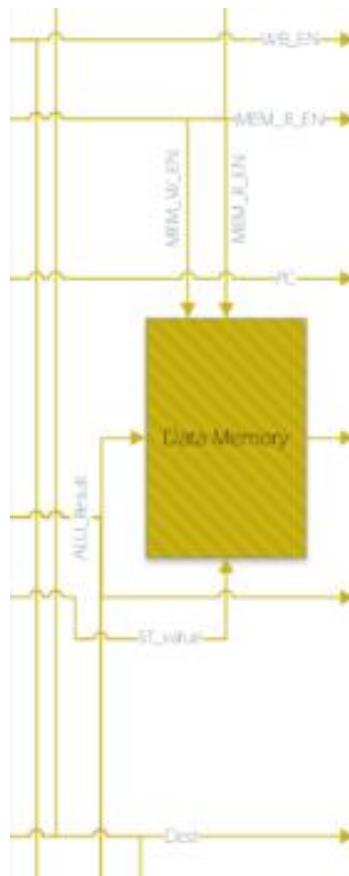


**Hình 3.0.23 Thiết kế chi tiết EX Stage**

Đường tín hiệu màu vàng là kết quả tính toán ALU, được đưa vào từ thanh ghi EX-MEM, đường tín hiệu màu xanh biển là giá trị write back, được đưa vào từ WB Stage. Các tín hiệu chọn cho 3 bộ MUX được đưa vào từ khối chuyển tiếp dữ liệu, việc xác định các tín hiệu này sẽ được trình bày ở khối chuyển tiếp dữ liệu.

### 3.16 MEM Stage

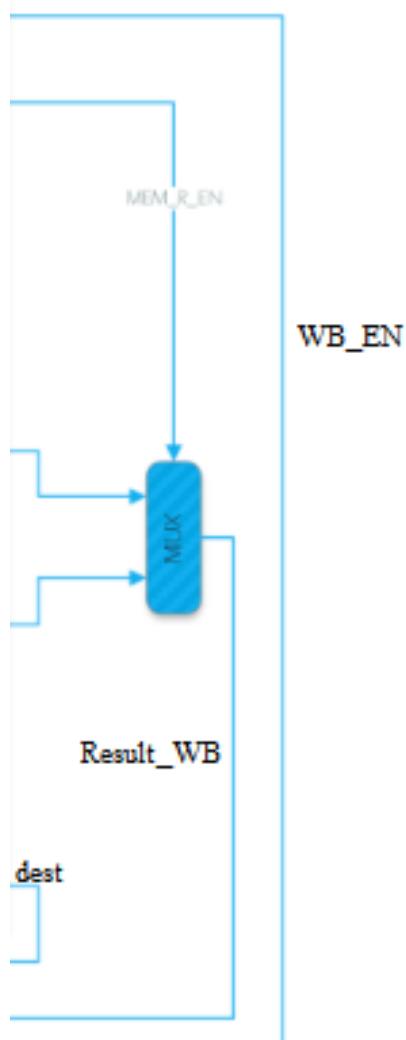
Trong Stage này, bộ xử lí truy cập vào bộ nhớ dữ liệu để tiến hành đọc hoặc ghi, kết quả đọc được đưa vào thanh ghi MEM-WB. Thiết kế MEM Stage chỉ gồm có memory, thiết kế chi tiết MEM Stage được thể hiện trên hình 3.24.



Hình 3.0.24 Thiết kế chi tiết MEM Stage

### 3.17 WB Stage

Trong WB Stage, bộ xử lý tiến hành ghi giá trị (đọc từ memory hoặc kết quả ALU) vào tệp thanh ghi. Thiết kế chi tiết WB Stage được thể hiện trên hình 3.25.



Hình 3.0.25 Thiết kế chi tiết WB Stage

Bộ MUX 2 to 1 dùng để xác định giá trị sẽ write back là đọc từ memory hay kết quả ALU. Các đường tín hiệu dest, Result\_WB và WB\_EN được đưa vào tệp thanh ghi.

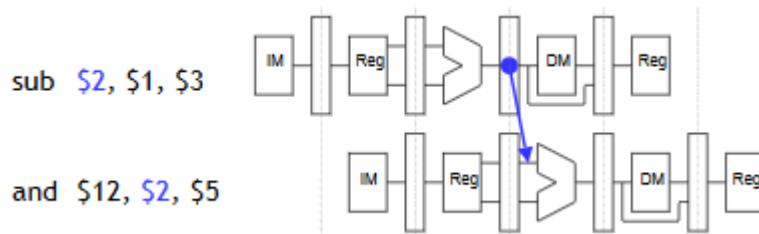
### 3.18 Khối phát hiện xung đột và khôi chuyển tiếp dữ liệu

#### 3.18.1 Cơ chế phát hiện xung đột và giải quyết xung đột bằng phương pháp chuyển tiếp dữ liệu

- Xung đột EX/MEM xảy ra giữa lệnh đang ở EX Stage và lệnh trước đó nếu:
  - Lệnh trước đó sẽ ghi vào tệp thanh ghi

- Địa chỉ thanh ghi sẽ ghi vào là 1 trong 2 toán hạng vào ALU của lệnh đang thực thi ở EX Stage

Xét một ví dụ xung đột EX/MEM (hình 3.26)

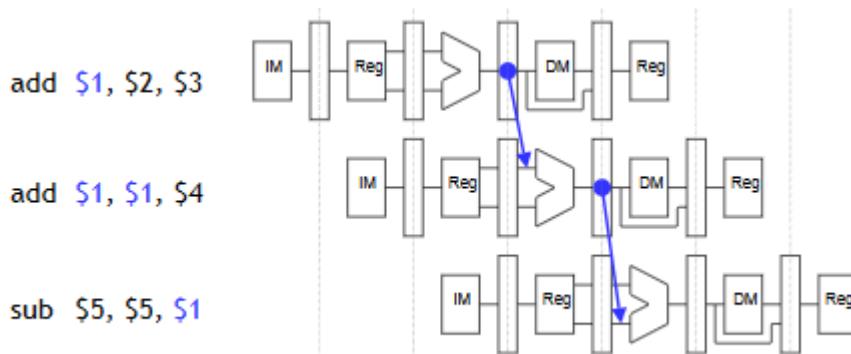


**Hình 3.0.26 Xung đột EX/MEM**

Sự xung đột dữ liệu xảy ra do thanh ghi \$2 được sử dụng trong EX Stage ở lệnh thứ hai vẫn chưa được cập nhật kết quả tại WB Stage ở lệnh thứ nhất. Để giải quyết xung đột, dữ liệu sau khi tính toán trong EX Stage ở lệnh thứ nhất được đưa vào EX Stage ở lệnh thứ hai (từ thanh ghi EX-MEM) làm một toán hạng vào ALU.

- Xung đột MEM/WB xảy ra giữa lệnh đang ở EX stage và lệnh từ 2 cycles trước với các điều kiện tương tự xung đột EX/MEM

Xét một ví dụ xung đột MEM/WB (hình 3.27)

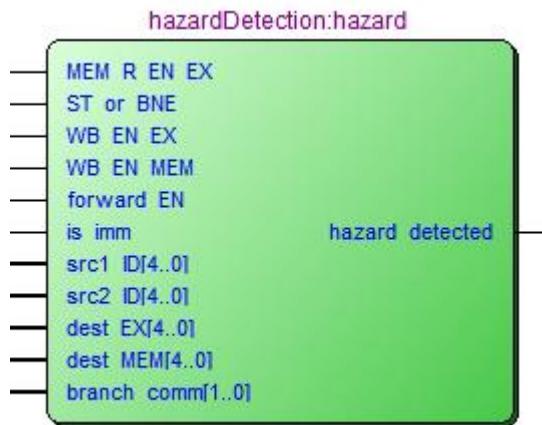


**Hình 3.0.27 Xung đột MEM/WB**

Thanh ghi \$1 được ghi bởi cả lệnh thứ nhất và thứ hai, tuy nhiên chỉ cần chuyển tiếp dữ liệu kết quả gần nhất (từ lệnh thứ hai) cho lệnh thứ ba.

### 3.18.2 Khối phát hiện xung đột

Thiết kế khôi phát hiện xung đột được thể hiện trên hình 3.28



**Hình 3.0.28 Khối phát hiện xung đột**

Các đầu vào gồm:

- MEM\_R\_EN\_EX, WB\_EN\_EX: Các tín hiệu điều khiển từ thanh ghi ID-EX
- WB\_EN\_MEM: Tín hiệu điều khiển từ thanh ghi EX-MEM
- ST\_or\_BNE, is\_imm: Các tín hiệu điều khiển từ bộ điều khiển
- forward\_EN: Xác định có dùng chuyển tiếp dữ liệu không
- src1\_ID, src2\_ID (5 bits): Từ ID Stage, src1 là rt, src2 là rs hoặc rd
- dest\_EX (5 bits): Địa chỉ write back tệp thanh ghi từ thanh ghi ID-EX
- dest\_MEM (5 bits): Địa chỉ write back tệp thanh ghi từ thanh ghi EX-MEM
- branch\_comm (2 bits): Loại lệnh rẽ nhánh, từ bộ điều khiển

Đầu ra chỉ có:

- hazard\_detected: Xác định có xung đột không

Dựa vào các điều kiện phát hiện xung đột, các xung đột EX/MEM và MEM/WB được xác định trong code Verilog như sau:

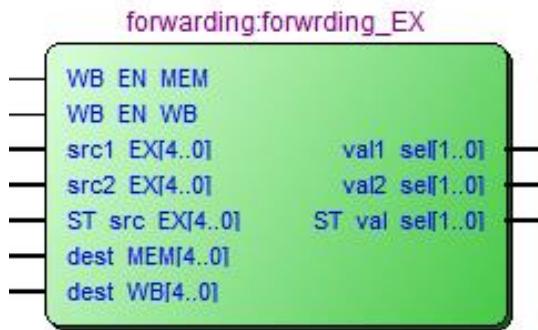
```

assign src2_is_valid= (~is_imm) || ST_or_BNE;
assign EX_has_hazard=WB_EN_EX && (src1_ID==dest_EX || (src2_is_valid && src2_ID==dest_EX));
assign mem_has_hazard=WB_EN_MEM && (src1_ID==dest_MEM || (src2_is_valid && src2_ID==dest_MEM));

```

### 3.18.3 Khối chuyển tiếp dữ liệu

Thiết kế khối chuyển tiếp dữ liệu được thể hiện trên hình 3.29



**Hình 3.0.29 Khối chuyển tiếp dữ liệu**

Các đầu vào gồm:

- `WB_EN_MEM`: Tín hiệu điều khiển từ thanh ghi EX/MEM
- `WB_EN_WB`: Tín hiệu điều khiển từ thanh ghi MEM/WB
- `src1_EX`, `src2_EX`, `ST_src_EX` (5 bits): Từ ID Stage, src1 là rt, src2 là rs (nếu lệnh không phải loại I) hoặc 0 (nếu lệnh loại I), `ST_src_EX` là địa chỉ truy cập memory
- `dest_MEM` (5 bits): Địa chỉ write back tệp thanh ghi từ thanh ghi EX-MEM
- `dest_WB` (5 bits): Địa chỉ write back tệp thanh ghi từ thanh ghi MEM-WB

Các đầu ra gồm:

- `val1_sel`, `val2_sel`, `ST_val_val` (2 bits): Các tín hiệu lựa chọn của 3 bộ MUX chọn toán hạng vào ALU và giá trị ghi vào memory

Dựa vào các điều kiện phát hiện xung đột, các tín hiệu lựa chọn `val1_sel`, `val2_sel` và `ST_val_val` được xác định trong code Verilog như sau:

```

always @*
begin
    //Mac dinh =0, neu co forwarding cac gia tri nay se thay doi
    {val1_sel, val2_sel, ST_val_sel} <= 0;
    //Chon du lieu chuyen tiep ghi vao memory
    if(WB_EN_MEM && ST_src_EX==dest_MEM)
        ST_val_sel<=2'd1;
    else if(WB_EN_WB && ST_src_EX==dest_WB)

```

```

ST_val_sel<=2'd2;

//Chon du lieu chuyen tiep cho toan hang thu nhat vao ALU

if(WB_EN_MEM && src1_EX==dest_MEM)
    val1_sel<=2'd1;

else if(WB_EN_WB && src1_EX==dest_WB)
    val1_sel<=2'd2;

//Chon du lieu chuyen tiep cho toan hang thu hai vao ALU

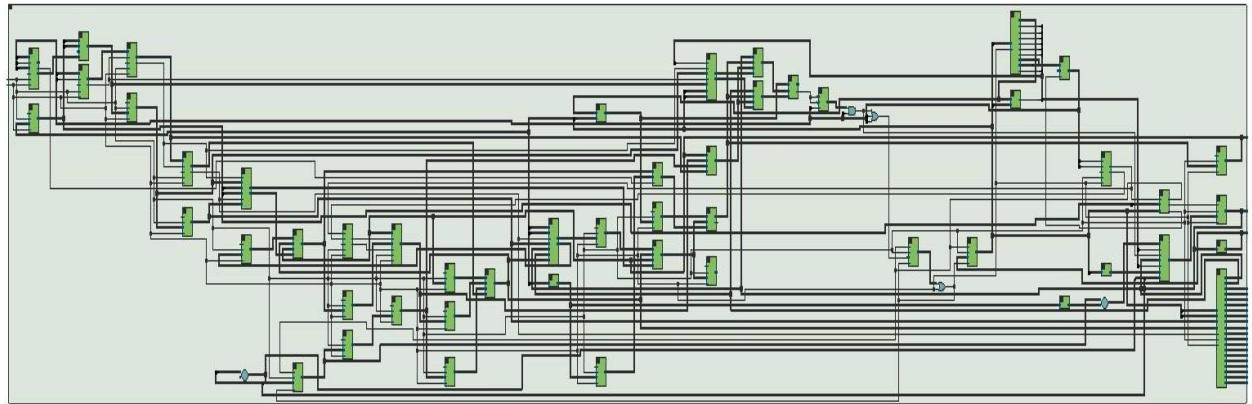
if(WB_EN_MEM && src2_EX == dest_MEM)
    val2_sel <= 2'd1;

else if(WB_EN_WB && src2_EX == dest_WB)
    val2_sel <= 2'd2;

end

```

Sơ đồ thiết kế bộ xử lý sau khi hoàn thành ghép các khối:



## CHƯƠNG 4. KẾT QUẢ MÔ PHỎNG THIẾT KẾ

## 4.1 Công cụ thực hiện

Quá trình mô phỏng thiết kế, em sử dụng những công cụ sau để thực hiện:

- Trình biên dịch assembly của kiến trúc MIPS32 chạy trên hệ điều hành ubuntu. Cài đặt thông qua WSL( Window System Linux), mục đích dùng để biên dịch chương trình thực thi thành các lệnh dưới dạng mã hex.
  - Phần mềm mô phỏng thiết kế ModelSim 10.1, dùng để kiểm tra tín hiệu của thiết kế.

## 4.2 Chương trình 1

Chương trình dùng để kiểm tra kết quả thiết kế

**Hình 4.0.1 Chương trình 1**

## Tóm tắt hoạt động:

Chương trình hợp ngữ MIPS này thực hiện một loạt các phép toán số học và logic, lưu trữ và tải giá trị vào và từ bộ nhớ, so sánh giá trị và điều kiện rẽ nhánh. Kết quả thực thi chương trình:

1.Khởi tạo các thanh ghi \$t0 và \$t1:

`$t0 = 32 (0x20) và $t1 = 39 (0x27)`

## 2.Phép toán AND và OR:

and \$s0, \$t0, \$t1 : \$s0 = \$t0 & \$t1 = 32 & 39 = 32 (0x20)

or \$s0, \$t0, \$t1 : \$s0 = \$t0 | \$t1 = 32 | 39 = 39 (0x27)

## 3.Lưu giá trị vào bộ nhớ:

sw \$s0, 4(\$zero) -> Bộ nhớ [4] = 39 (0x27)

sw \$t0, 8(\$zero) -> Bộ nhớ[8] = 32 (0x20)

## 4.Phép toán ADD và SUB:

add \$s1, \$t0, \$t1 : \$s1 = \$t0 + \$t1 = 32 + 39 = 71 (0x47)

sub \$s2, \$t0, \$t1 : \$s2 = \$t0 - \$t1 = 32 - 39 = -7 (0xffffffff9)

## 5.Điều kiện rẽ nhánh BEQ:

beq \$s1, \$s2, error0 : Không nhảy vì \$s1 (71) != \$s2 (-7)

## 6.Tải giá trị từ bộ nhớ:

lw \$s1, 4(\$zero) : Tải \$s1 từ địa chỉ 4

\$s1 = 39 (0x27) (Bộ nhớ[4] = 39)

## 7.Phép toán logic ANDI:

andi \$s2, \$s1, 0x18 : \$s2 = \$s1 & 0x18 = 0x27 & 0x18 = 0

\$s2 = 0

## 8.Điều kiện rẽ nhánh BEQ:

beq \$s1, \$s2, error1: Không nhảy, vì \$s1(39) != \$s2(0)

## 9.Tải giá trị từ bộ nhớ:

lw \$s3, 8(\$zero) : Tải \$s3 từ địa chỉ 8

\$s3 = 32 (0x20) (Bộ nhớ[8] = 32)

## 10.Điều kiện rẽ nhánh BEQ:

beq \$s0, \$s3, error2 : Không nhảy vì \$s0 (39) != \$s3 (32)

## 11.Phép toán SLT:

slt \$s4, \$s2, \$s1 : \$s4 = (\$s2 < \$s1) = (0 < 39) = 1

\$s4 = 1

12. Điều kiện rẽ nhánh BEQ:

beq \$s4, \$zero, EXIT : Không nhảy vì \$s4 (1) != 0

13. Sao chép giá trị từ \$s1 sang \$s2:

add \$s2, \$s1, \$zero : \$s2 = \$s1

\$s2 = 39 (0x27)

14. Nhảy đến nhãn Last:

j Last : Nhảy đến nhãn Last (chạy lại lệnh SLT)

15. Thực thi lại lệnh SLT

slt \$s4, \$s2, \$s1: Đặt \$s4 = (\$s2 < \$s1) = (39 < 39) = 0

\$s4 = 0

16. Điều kiện rẽ nhánh BEQ:

beq \$s4, \$zero, EXIT: Nếu \$s4 == 0 thì nhảy đến nhãn EXIT.

Vì \$s4 hiện tại là 0, chương trình sẽ nhảy đến nhãn EXIT.

Khi chương trình nhảy đến nhãn EXIT, nó sẽ không thực hiện thêm bất kỳ lệnh nào nữa vì đây là điểm kết thúc của chương trình. Các giá trị thanh ghi cuối cùng:

\$t0 = 32 (0x20)

\$t1 = 39 (0x27)

\$s0 = 39 (0x27)

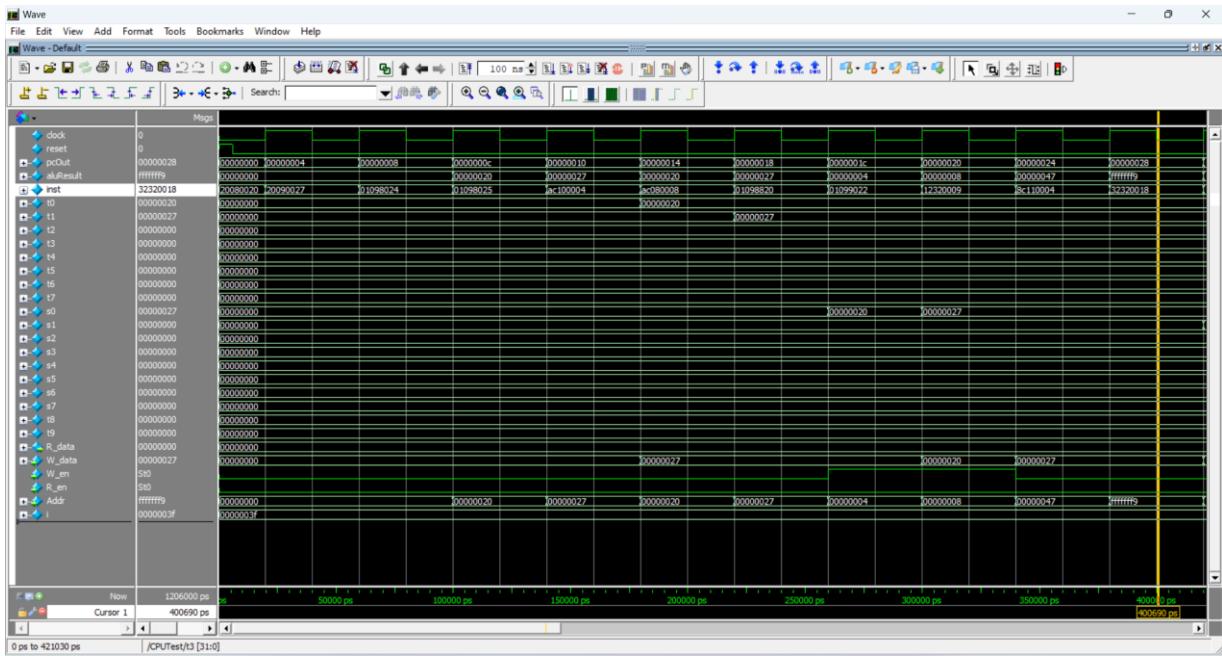
\$s1 = 39 (0x27)

\$s2 = 39 (0x27)

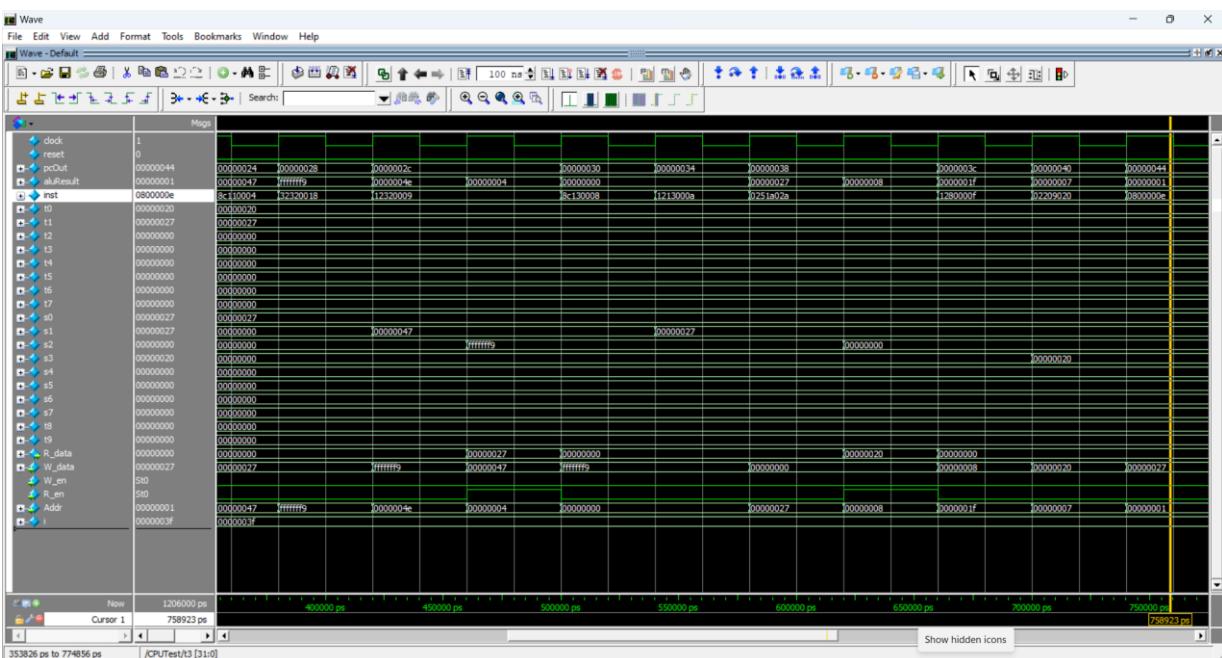
\$s3 = 32 (0x27)

\$s4 = 0 (0x27)

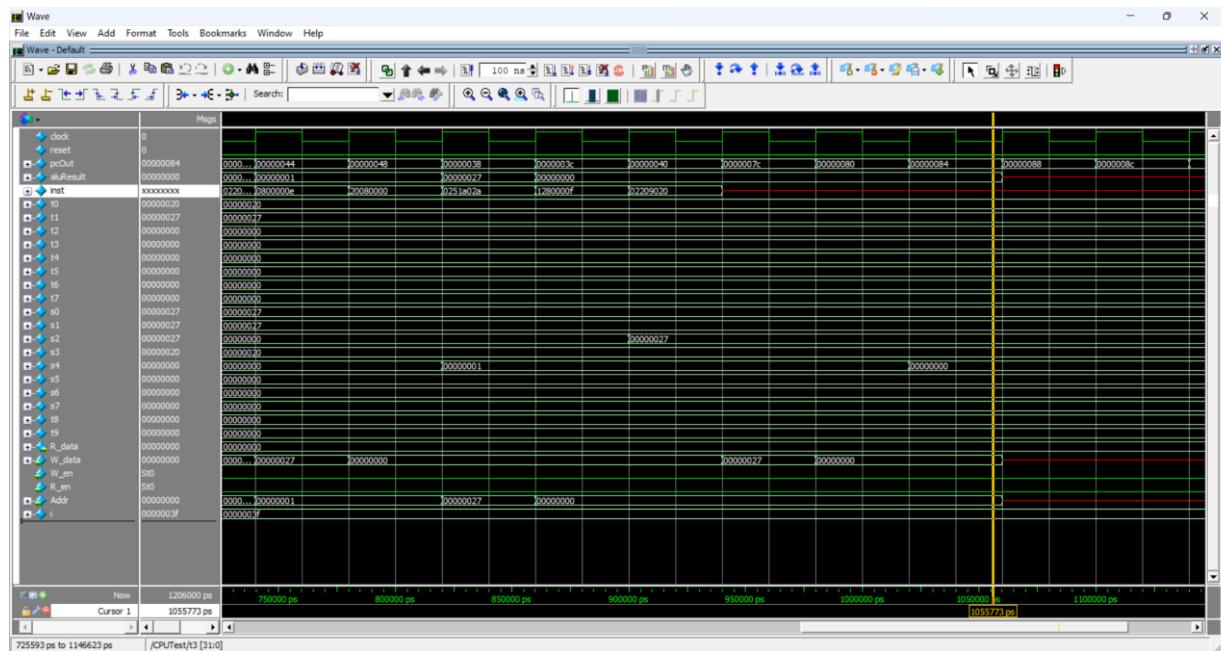
Kết quả mô phỏng trên phần mềm ModelSim:



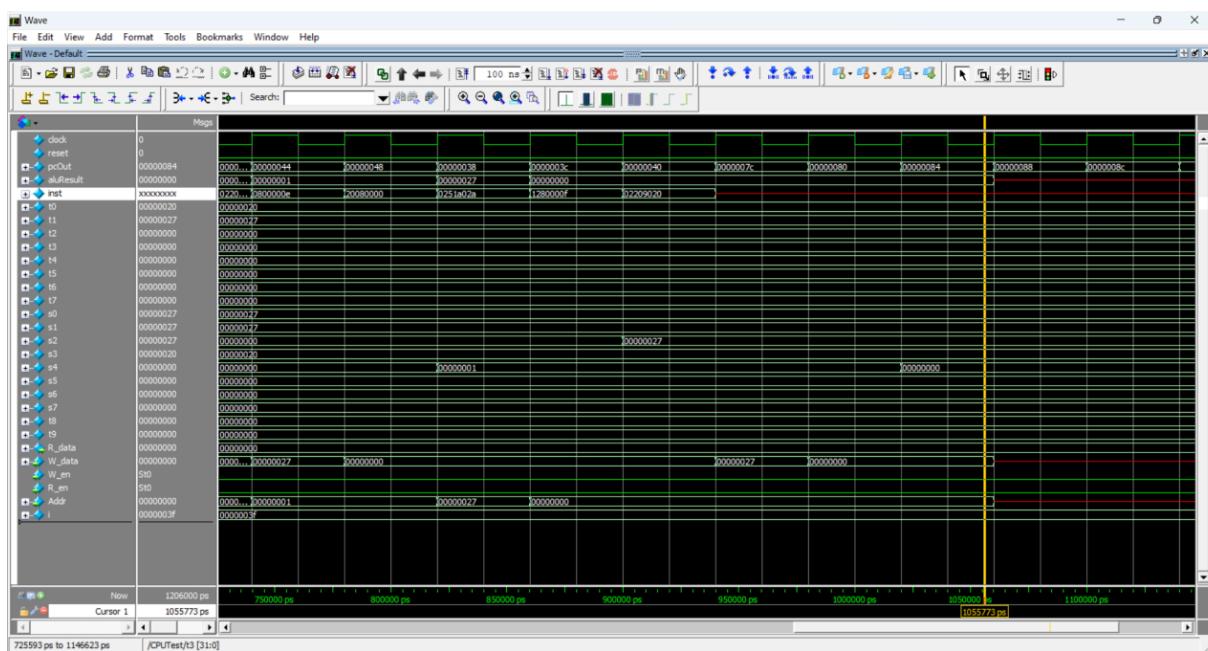
Hình 4.0.2 Kết quả mô phỏng chương trình 1(1)



Hình 4.0.3 Kết quả mô phỏng chương trình 1(2)



**Hình 4.0.2 Kết quả mô phỏng chương trình 1(3)**



**Hình 4.0.3 Kết quả mô phỏng chương trình 1(4)**

**Nhận xét:** Các giá trị cuối cùng của thanh ghi đích đúng với kết quả tính toán lý thuyết. Kết quả mô phỏng cho thấy, bộ xử lý thực thi các lệnh chính xác.

### 4.3 Chương trình 2

Chương trình này tập trung vào kiểm tra xử lý xung đột của thiết kế, nạp chương trình sau vào bộ nhớ lệnh:

```
b0100000000100000000000000000100111 //addi $s0, $zero, 0x27  
00100000001000100000000000100000 //addi $s1, $zero, 0x20  
000010000000000000000000000000101 // j next1 [Control Hazard]  
0010000000100000000000000000000001 //addi $s0, $zero, 0x01 (next2)  
0010000000100001000000000000000001 //addi $s1, $zero, 0x01  
00000010001100001001000000100010 //sub $s2, $s1, $s0 (next1) [Data hazard frwd]  
000101100001000111111111111100 //bne $s0, $s1, next2 [Data and control hazard frwd]  
000000100001000110001100000100000 //add $s3, $s0, $s1  
10101110010100110000000000010000 //sw $s3, 16($s2) [Data hazard frwd]  
100011100101010100000000000010000 //lw $s4, 16($s2)  
000000100001010010101000000000101010 //slt $s5, $s0, $s4 [Data hazard stall + frwd]  
10001110010100110000000000010000 //lw $s3, 16($s2)  
00100010010100110000000000000001 //addi $s3, $s2, 0x01 [No data hazard stall]  
001000101011010101000000000000001 //addi $s5, $s5, 0x01
```

**Hình 4.0.4 Chương trình 2**

Phân tích chương trình:

1. addi \$s0, \$zero, 0x27: Thêm giá trị 0x27 vào thanh ghi \$s0. Vì vậy, \$s0 sẽ giữ giá trị 0x27.
2. addi \$s1, \$zero, 0x20: Thêm giá trị 0x20 vào thanh ghi \$s1. Vậy, \$s1 sẽ giữ giá trị 0x20.
3. j next1: Nhảy tới nhãn next1. Điều này là một lệnh nhảy không điều kiện, nên nó chỉ đơn giản là thay đổi con trỏ lệnh tới next1.
4. addi \$s0, \$zero, 0x01: Thêm giá trị 0x01 vào thanh ghi \$s0. Vậy, \$s0 sẽ giữ giá trị 0x01.
5. addi \$s1, \$zero, 0x01: Thêm giá trị 0x01 vào thanh ghi \$s1. Vậy, \$s1 sẽ giữ giá trị 0x01.
6. sub \$s2, \$s1, \$s0: Trừ giá trị của \$s0 từ \$s1 và lưu kết quả vào \$s2. Vì \$s1 và \$s0 đều giữ giá trị 0x01, nên \$s2 sẽ giữ giá trị 0x00.
7. bne \$s0, \$s1, next2: Nếu \$s0 và \$s1 không bằng nhau, nhảy tới nhãn next2. Vì \$s0 và \$s1 đều giữ giá trị 0x01 (bằng nhau), nên nhảy không được thực hiện và chương trình sẽ tiếp tục thực thi lệnh tiếp theo.
8. add \$s3, \$s0, \$s1: Thêm giá trị của \$s0 và \$s1 và lưu kết quả vào \$s3. Với \$s0 và \$s1 đều giữ giá trị 0x01, kết quả sẽ là 0x02 và lưu vào \$s3.

9. sw \$s3, 16(\$s2): Lưu giá trị của \$s3 vào bộ nhớ ở địa chỉ \$s2 + 16. Với \$s2 giữ giá trị 0x00, giá trị của \$s3 (0x02) sẽ được lưu vào địa chỉ 16.

10. lw \$s4, 16(\$s2): Tải giá trị từ địa chỉ \$s2 + 16 vào thanh ghi \$s4. Với \$s2 giữ giá trị 0x00, lệnh này sẽ tải giá trị 0x02 từ bộ nhớ.

11. slt \$s5, \$s0, \$s4: So sánh nếu \$s0 nhỏ hơn \$s4, nếu đúng gán 1 vào \$s5, ngược lại gán 0. Với \$s0 giữ giá trị 0x01 và \$s4 giữ giá trị 0x02, vậy \$s5 sẽ giữ giá trị 1.

12. lw \$s3, 16(\$s2): Tải giá trị từ địa chỉ \$s2 + 16 vào thanh ghi \$s3.

13. addi \$s3, \$s2, 0x01: Thêm giá trị 0x01 vào thanh ghi \$s2 và lưu vào \$s3. Vì \$s2 giữ giá trị 0x00, kết quả sẽ là 0x01 và lưu vào \$s3.

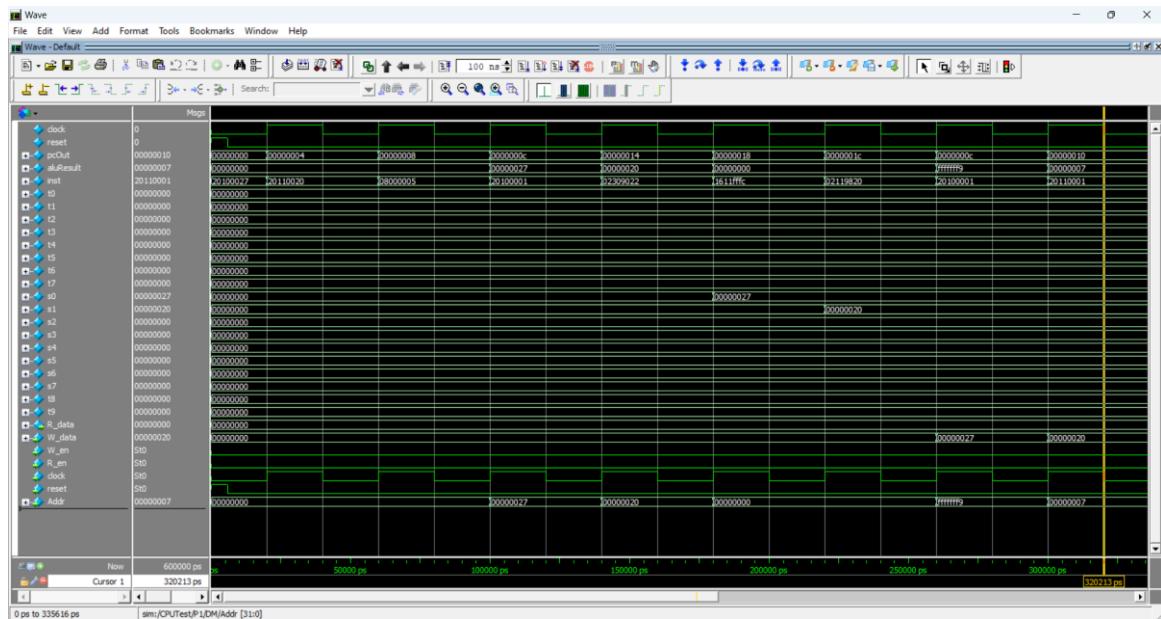
14. addi \$s5, \$s5, 0x01: Thêm giá trị 0x01 vào thanh ghi \$s5. Vì \$s5 giữ giá trị 1, kết quả sẽ là 2 và lưu vào \$s5. Giá trị cuối cùng của các thanh ghi là:

\$s0: 0x01; \$s1: 0x01; \$s2: 0x00; \$s3: 0x01; \$s4: 0x02; \$s5: 0x02

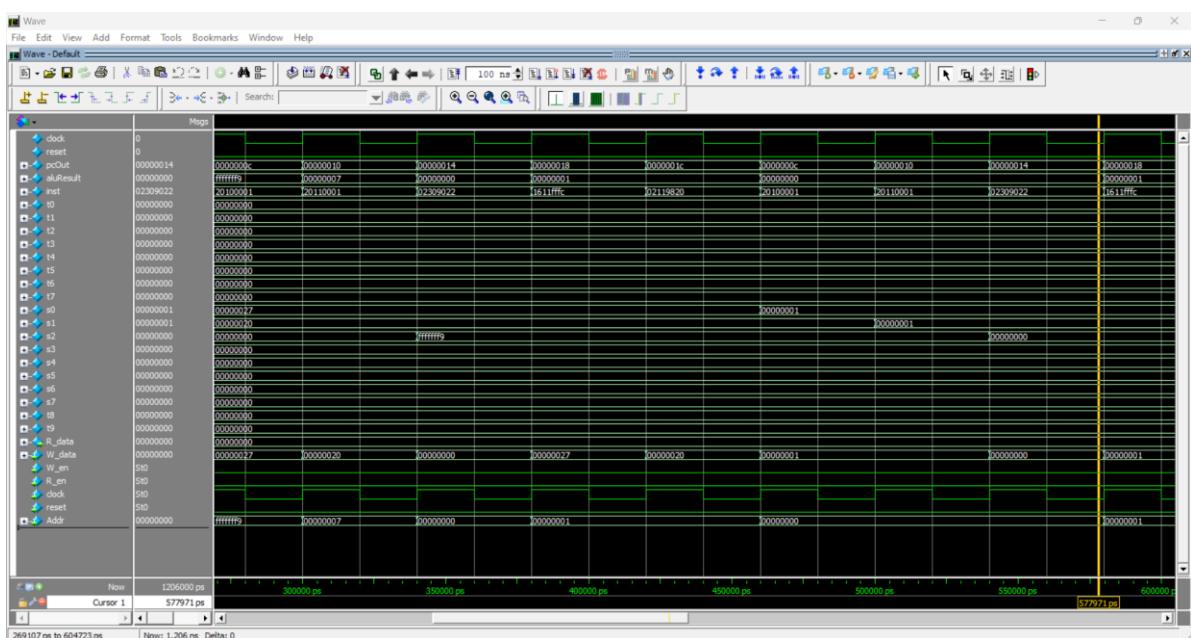
Xét tính xung đột:

- Lệnh thứ sáu và lệnh thứ bảy: sub \$s2, \$s1, \$s0 (lệnh thứ sáu) và bne \$s0, \$s1, next2 (lệnh thứ bảy) đều sử dụng thanh ghi \$s0 và \$s1. Lệnh thứ bảy sử dụng \$s0 và \$s1 để so sánh, trong khi lệnh thứ sáu sử dụng chúng để tính toán.
- Lệnh thứ mười và lệnh thứ mười một: lw \$s4, 16(\$s2) (lệnh thứ mười) và slt \$s5, \$s0, \$s4 (lệnh thứ mười một) đều sử dụng thanh ghi \$s4. Lệnh thứ mười tải giá trị từ bộ nhớ vào \$s4, trong khi lệnh thứ mười một sử dụng \$s4 để so sánh với \$s0.

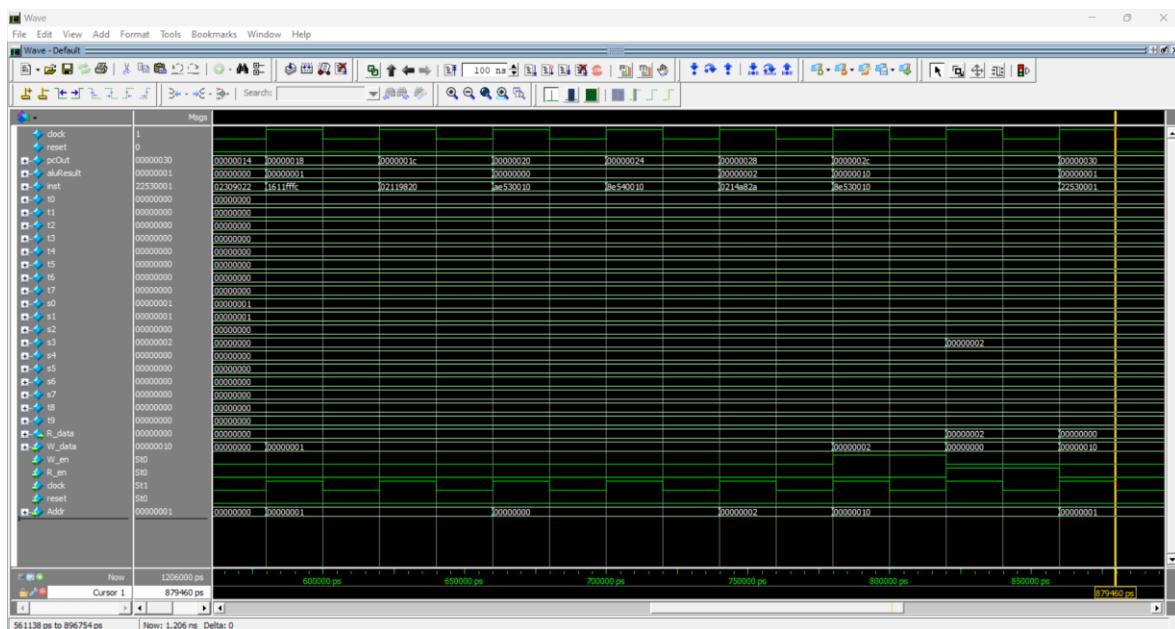
Kết quả mô phỏng trên ModelSim:



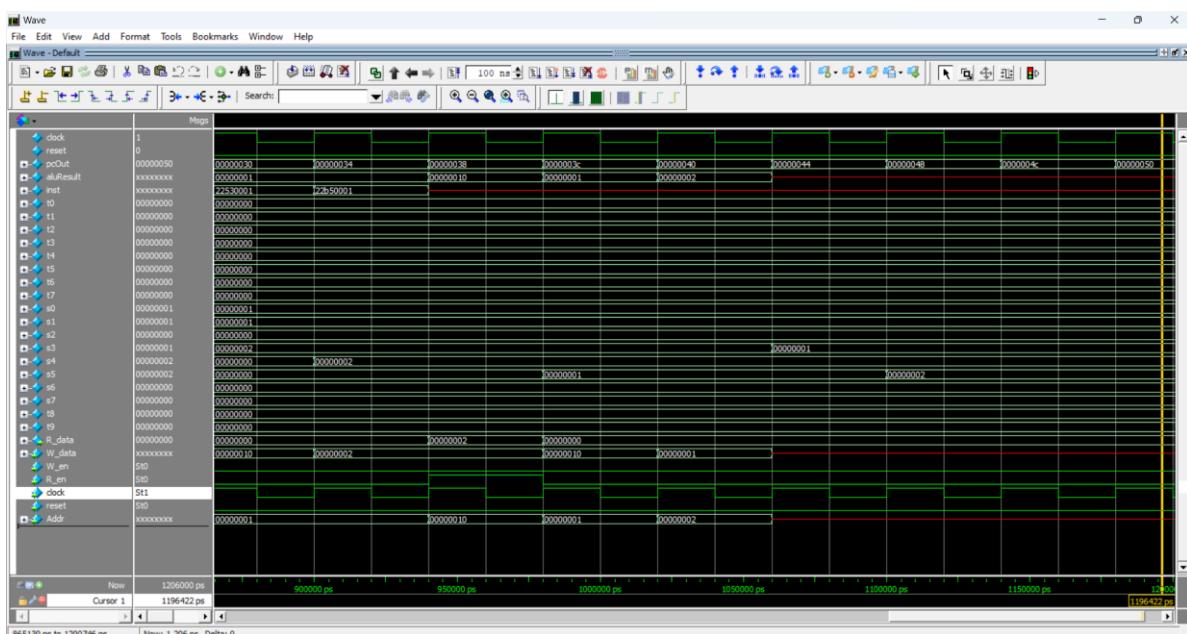
Hình 4.0.5 Kết quả mô phỏng chương trình 2(1)



Hình 4.0.6 Kết quả mô phỏng chương trình 2(2)



**Hình 4.0.7 Kết quả mô phỏng chương trình 2(3)**

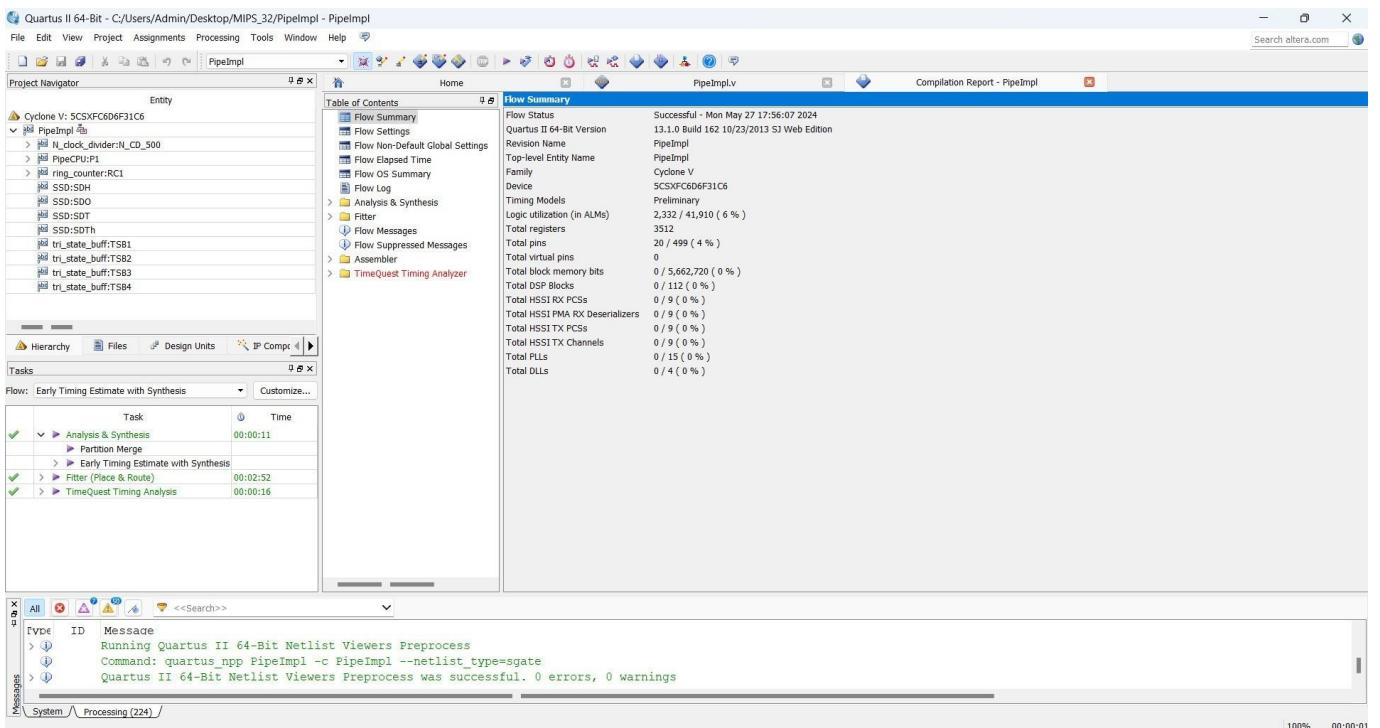


**Hình 4.0.8 Kết quả mô phỏng chương trình 2(4)**

**Nhận xét:** Các giá trị cuối cùng của thanh ghi đích đúng với kết quả phân tích lý thuyết. Bằng cách sử dụng các kỹ thuật như chuyển tiếp dữ liệu (forwarding) và đợi (stalling), bộ xử lý đã giải quyết các xung đột dữ liệu một cách chính xác, đảm bảo rằng các lệnh được thực thi đúng và kết quả là như mong đợi.

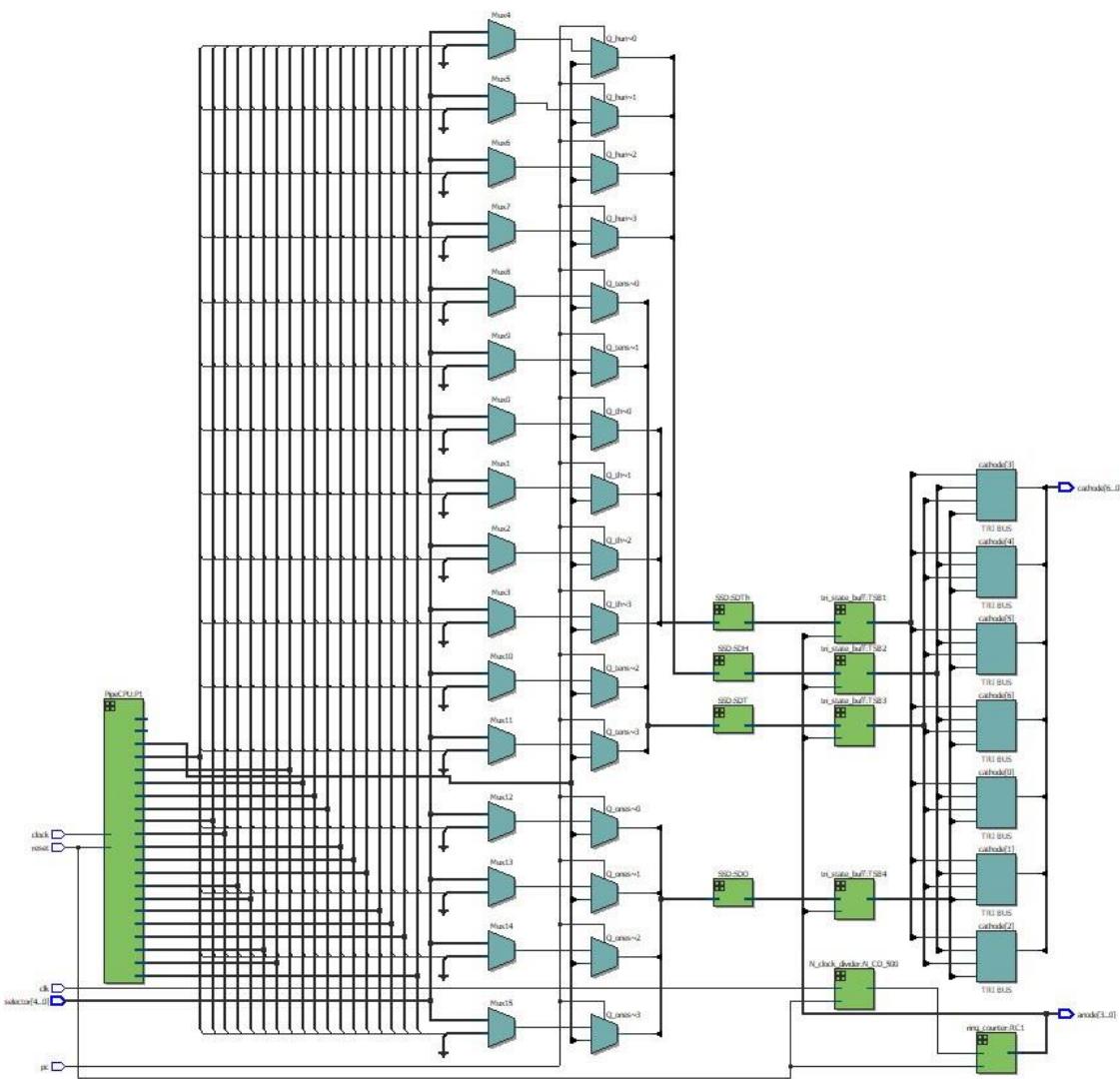
#### 4.4 Triển khai thiết kế lên FPGA DE10

Việc triển khai thiết kế trên kit FPGA DE10 chưa được hoàn thành, chỉ dừng lại ở giai đoạn tổng hợp chương trình. Bên dưới là minh chứng cho việc tổng hợp code thành công:



Hình 4.0.11 Kết quả tổng hợp thiết kế

Sơ đồ thiết kế (RTL) của bộ xử lý khi triển khai lên FPGA DE10:



**Hình 4.0.12 Sơ đồ thiết kế**

## KẾT LUẬN

Trong quá trình thiết kế bộ xử lý MIPS32 pipeline, em đã đạt được những kết quả quan trọng thể hiện qua các giai đoạn thiết kế và tối ưu hóa. Bộ xử lý MIPS32 pipeline được xây dựng với mục tiêu nâng cao hiệu suất xử lý, giảm độ trễ và tối đa hóa thông lượng hệ thống. Những kết quả này được thể hiện rõ qua việc giảm thiểu thời gian thực hiện lệnh, tăng cường khả năng xử lý song song và cải thiện hiệu quả tổng thể của bộ xử lý.

Quá trình thiết kế bắt đầu từ việc phân tích yêu cầu, xác định các thành phần chính của bộ xử lý MIPS32 và tiến hành thiết kế từng thành phần chi tiết như IF (Instruction Fetch), ID (Instruction Decode), EX (Execute), MEM (Memory Access) và WB (Write Back). Các thành phần này sau đó được tích hợp và tối ưu hóa để giảm thiểu các xung đột pipeline, tăng cường khả năng dự đoán nhánh và quản lý dữ liệu.

Một số thách thức chính đã được giải quyết bao gồm:

- Xung đột về điều khiển: Các chiến lược dự đoán nhánh tiên tiến đã được áp dụng để giảm thiểu các xung đột về điều khiển, từ đó giảm số chu kỳ chờ do nhánh không đúng.
- Xung đột về dữ liệu: Các kỹ thuật xử lý dữ liệu như hazard detection và forwarding đã được sử dụng hiệu quả để xử lý các xung đột về dữ liệu.

Thiết kế bộ xử lý MIPS32 pipeline không chỉ đáp ứng các yêu cầu về hiệu năng mà còn đảm bảo tính khả thi trong việc sản xuất và triển khai. Kết quả đạt được từ việc mô phỏng và thử nghiệm cho thấy bộ xử lý có thể hoạt động ổn định, đáp ứng được các tiêu chuẩn về hiệu năng và độ tin cậy.

Trong tương lai, để phát triển tài có thể có thể được phát triển theo nhiều hướng khác nhau để tiếp tục nâng cao hiệu suất, giảm thiểu tiêu thụ năng lượng và mở rộng ứng dụng trong các lĩnh vực công nghệ mới. Một số hướng phát triển tiềm năng bao gồm:

- Mở rộng tập lệnh: giúp nâng cao hiệu suất, mở rộng khả năng ứng dụng và tăng cường tính linh hoạt của bộ xử lý.
- Tối ưu hóa năng lượng:

Kỹ thuật DVFS (Dynamic Voltage and Frequency Scaling): Áp dụng DVFS để điều chỉnh động điện áp và tần số làm việc của bộ xử lý nhằm tối ưu hóa tiêu thụ năng lượng mà không ảnh hưởng đến hiệu suất.

Thiết kế các chế độ ngủ sâu (deep sleep modes): Đưa ra các chế độ nghỉ ngơi cho các thành phần không sử dụng để giảm tiêu thụ năng lượng khi bộ xử lý không hoạt động toàn bộ.

- Cải thiện kiến trúc pipeline:

Tăng cường độ sâu pipeline: Nghiên cứu và thử nghiệm các kiến trúc pipeline sâu hơn để tăng khả năng xử lý song song và nâng cao hiệu suất.

Pipeline đa luồng (Multithreading): Thiết kế các pipeline hỗ trợ đa luồng để tối ưu hóa việc sử dụng tài nguyên và tăng hiệu suất xử lý.

- Tăng cường dự đoán nhánh:

Sử dụng thuật toán dự đoán nhánh: Áp dụng các thuật toán dự đoán nhánh như neural branch prediction hoặc hybrid branch prediction để cải thiện độ chính xác của dự đoán nhánh.

Nghiên cứu về cơ chế dự đoán động: Xây dựng các cơ chế dự đoán động, cho phép điều chỉnh thuật toán dự đoán dựa trên hành vi thực tế của chương trình.

## **TÀI LIỆU THAM KHẢO**

- [1] David A. Patterson and John L. Hennessy, *Computer Organization and Design, The Hardware/ Software Interface*, 4<sup>th</sup> Edition, Elsevier Inc, 2014
- [2] CS61C Course Summer 2017, *Great Ideas in Computer Architecture (Machine Structures)*, University of California, Berkeley, 2017
- [3] CSE378 Course Autumn 2007, *Machine Organization & Assembly Language*, Paul G.Allen School, 2007