Dưới đây là một số câu hỏi thường gặp về kiến trúc MIPS kèm theo câu trả lời nhằm giúp bạn hiểu rõ hơn về kiến trúc này:

1. Kiến trúc MIPS là gì?

Câu trả lời:

MIPS (Microprocessor without Interlocked Pipeline Stages) là một kiến trúc vi xử lý RISC (Reduced Instruction Set Computing) được phát triển bởi MIPS Computer Systems vào những năm 1980. MIPS được thiết kế để tối ưu hóa hiệu năng của bộ xử lý bằng cách giảm số lượng lệnh và đơn giản hóa các lệnh, nhờ đó tăng tốc độ thực thi.

2. Các đặc điểm chính của kiến trúc MIPS là gì?

Câu trả lời:

Những đặc điểm chính của kiến trúc MIPS bao gồm:

Thiết kế RISC (Reduced Instruction Set Computing): Bộ lệnh đơn giản, mỗi lệnh thực hiện một tác vụ nhỏ.

Pipeline: MIPS sử dụng cấu trúc pipeline để tăng hiệu suất, cho phép nhiều lệnh được thực hiện đồng thời trong các giai đoạn khác nhau.

Load/Store Architecture: Chỉ có các lệnh load và store mới truy cập bộ nhớ, trong khi các lệnh khác chỉ hoạt động trên các thanh ghi.

Số lượng thanh ghi cố định: MIPS có 32 thanh ghi tổng quát (General Purpose Registers), mỗi thanh ghi 32-bit.

Định dạng lệnh cố định: Lệnh MIPS có độ dài cố định là 32-bit.

3. Các loại lệnh cơ bản của MIPS là gì?

Câu trả lời:

Trong MIPS, các lệnh được chia thành ba loại cơ bản:

R-type (Register): Các lệnh tính toán giữa các thanh ghi, ví dụ: add, sub, and, or.

I-type (Immediate): Các lệnh có sử dụng giá trị tức thời (immediate), ví dụ: addi, lw, sw.

J-type (Jump): Các lệnh nhảy, ví dụ: j, jal.

4. Pipeline trong MIPS hoạt động như thế nào?

Câu trả lời:

Pipeline trong MIPS chia quá trình thực thi một lệnh thành 5 giai đoạn chính:

IF (Instruction Fetch): Đọc lệnh từ bộ nhớ.

ID (Instruction Decode): Giải mã lệnh, đọc các thanh ghi nguồn.

EX (Execute): Thực hiện tính toán hoặc địa chỉ bộ nhớ.

MEM (Memory Access): Truy cập bộ nhớ nếu cần (đối với các lệnh load/store).

WB (Write Back): Ghi kết quả vào thanh ghi đích.

Pipeline giúp MIPS thực hiện nhiều lệnh song song, tăng tốc độ xử lý.

5. Lệnh lw và sw trong MIPS có chức năng gì?

Câu trả lời:

lw (load word): Lệnh này tải một từ (word, 32-bit) từ bộ nhớ vào một thanh ghi. Ví dụ: lw $t0, 0($s0) sẽ tải giá trị từ địa chỉ chứa trong $s0 vào thanh ghi $t0.

sw (store word): Lệnh này lưu một từ từ một thanh ghi vào bộ nhớ. Ví dụ: sw $t0, 0($s0) sẽ lưu giá trị trong thanh ghi $t0 vào địa chỉ chứa trong $s0.

6. Các thanh ghi trong MIPS được sử dụng như thế nào?

Câu trả lời:

MIPS có 32 thanh ghi tổng quát (GPR – General Purpose Registers) được đánh số từ $0 đến $31. Một vài thanh ghi có chức năng đặc biệt:

$zero ($0): Luôn chứa giá trị 0.

$at ($1): Thanh ghi tạm thời, được sử dụng trong các lệnh giả (pseudo-instruction).

$v0-$v1 ($2-$3): Dùng để trả về giá trị kết quả của các hàm.

$a0-$a3 ($4-$7): Dùng để truyền tham số cho hàm.

$t0-$t9 ($8-$15, $24-$25): Thanh ghi tạm thời.

$s0-$s7 ($16-$23): Thanh ghi lưu trữ giá trị lâu dài.

$gp ($28): Global pointer, trỏ tới khu vực dữ liệu toàn cục.

$sp ($29): Stack pointer, trỏ tới đỉnh của ngăn xếp.

$ra ($31): Return address, giữ địa chỉ trả về sau khi gọi hàm.

7. Lệnh nhảy j và jal trong MIPS khác nhau như thế nào?

Câu trả lời:

j (jump): Thực hiện nhảy đến địa chỉ xác định trong lệnh. Lệnh này chỉ thay đổi giá trị của bộ đếm chương trình (PC).

jal (jump and link): Nhảy đến địa chỉ xác định và lưu địa chỉ của lệnh tiếp theo vào thanh ghi $ra để có thể quay lại sau khi thực thi hàm.

8. Điểm khác biệt chính giữa kiến trúc RISC và CISC là gì?

Câu trả lời:

RISC (Reduced Instruction Set Computing):

Số lượng lệnh ít hơn, lệnh đơn giản hơn.

Mỗi lệnh thường được thực thi trong một chu kỳ máy.

Sử dụng nhiều thanh ghi và kiến trúc load/store.

Ví dụ: MIPS, ARM.

CISC (Complex Instruction Set Computing):

Số lượng lệnh nhiều hơn, lệnh phức tạp hơn.

Một lệnh có thể thực hiện nhiều tác vụ và có thể mất nhiều chu kỳ hơn để hoàn thành.

Sử dụng ít thanh ghi hơn, nhiều lệnh có thể truy cập trực tiếp bộ nhớ.

Ví dụ: x86.

9. Tại sao MIPS sử dụng định dạng lệnh cố định 32-bit?

Câu trả lời:

Định dạng lệnh cố định 32-bit giúp đơn giản hóa việc giải mã lệnh trong phần cứng và tối ưu hóa hiệu suất. Các lệnh có độ dài cố định sẽ dễ dàng được nạp và xử lý trong pipeline mà không cần quá nhiều phân tích hay xử lý phức tạp về kích thước lệnh.

10. Tại sao kiến trúc MIPS lại phổ biến trong giáo dục và giảng dạy?

Câu trả lời:

MIPS được sử dụng rộng rãi trong giáo dục vì:

Đơn giản: Kiến trúc RISC với số lượng lệnh ít và dễ hiểu giúp sinh viên dễ dàng nắm bắt các khái niệm cơ bản về vi xử lý.

Pipeline rõ ràng: MIPS có thiết kế pipeline dễ hiểu, giúp sinh viên học tập về các khái niệm liên quan đến pipeline và cách tối ưu hóa hiệu suất.

Tính mô phỏng cao: Có nhiều công cụ mô phỏng hỗ trợ kiến trúc MIPS, giúp sinh viên thực hành và thử nghiệm dễ dàng.

Dưới đây là một số câu hỏi khác về kiến trúc MIPS mà bạn có thể tham khảo để hiểu sâu hơn về các khía cạnh khác nhau của kiến trúc này:

**1. Endian trong MIPS là gì? MIPS sử dụng big-endian hay little-endian?**

**Câu trả lời:**  
**Endian** là cách thức mà dữ liệu được lưu trữ trong bộ nhớ. Có hai kiểu endianness chính:

* **Big-endian:** Byte có giá trị lớn nhất (most significant byte) được lưu trữ ở địa chỉ thấp nhất.
* **Little-endian:** Byte có giá trị nhỏ nhất (least significant byte) được lưu trữ ở địa chỉ thấp nhất.

MIPS có thể hoạt động ở cả **big-endian** và **little-endian**, tùy thuộc vào cách cấu hình hệ thống. Việc lựa chọn big-endian hay little-endian thường phụ thuộc vào yêu cầu của ứng dụng hoặc hệ điều hành chạy trên bộ xử lý MIPS.

**2. Kiến trúc MIPS có hỗ trợ các phép toán dấu phẩy động (floating point) không?**

**Câu trả lời:**  
Có, MIPS hỗ trợ các phép toán dấu phẩy động thông qua một bộ đồng xử lý gọi là **Coprocessor 1** (FPU - Floating Point Unit). FPU có bộ thanh ghi riêng cho các phép toán dấu phẩy động và cung cấp các lệnh để thực hiện các phép tính như cộng, trừ, nhân, chia số dấu phẩy động. Ngoài ra, MIPS cũng có các lệnh chuyển đổi giữa số nguyên và số dấu phẩy động.

**3. Điểm khác biệt giữa lệnh R-type, I-type và J-type trong MIPS là gì?**

**Câu trả lời:**  
MIPS có ba loại lệnh cơ bản, mỗi loại có định dạng riêng:

* **R-type (Register Type):** Được sử dụng cho các lệnh thao tác với thanh ghi, chẳng hạn như add, sub. Các lệnh này sử dụng 3 thanh ghi (2 thanh ghi nguồn và 1 thanh ghi đích).
  + Ví dụ: add $t1, $t2, $t3 (cộng giá trị của $t2 và $t3, lưu kết quả vào $t1).
* **I-type (Immediate Type):** Được sử dụng cho các lệnh có giá trị tức thời (immediate), chẳng hạn như addi, lw, sw. Lệnh này có 1 thanh ghi nguồn, 1 thanh ghi đích và 1 giá trị tức thời.
  + Ví dụ: addi $t0, $t1, 5 (cộng giá trị của $t1 với 5, lưu kết quả vào $t0).
* **J-type (Jump Type):** Được sử dụng cho các lệnh nhảy, chẳng hạn như j hoặc jal. Lệnh này có trường địa chỉ 26-bit để chỉ định địa chỉ nhảy đến.
  + Ví dụ: j 10000 (nhảy tới địa chỉ 10000).

**4. Bộ thanh ghi trong MIPS tổ chức như thế nào?**

**Câu trả lời:**  
MIPS có **32 thanh ghi tổng quát (General Purpose Registers)**, mỗi thanh ghi rộng **32-bit**. Các thanh ghi này được đánh số từ $0 đến $31, và một số thanh ghi có chức năng đặc biệt:

* **$zero ($0):** Luôn chứa giá trị 0.
* **$at ($1):** Thanh ghi tạm thời (assembler temporary), được sử dụng trong các lệnh giả.
* **$v0-$v1 ($2-$3):** Dùng để lưu trữ giá trị trả về của hàm.
* **$a0-$a3 ($4-$7):** Dùng để truyền tham số vào hàm.
* **$t0-$t9 ($8-$15, $24-$25):** Thanh ghi tạm thời.
* **$s0-$s7 ($16-$23):** Thanh ghi lưu trữ giá trị lâu dài.
* **$gp ($28):** Thanh ghi con trỏ toàn cục (global pointer).
* **$sp ($29):** Con trỏ ngăn xếp (stack pointer).
* **$fp ($30):** Con trỏ khung (frame pointer).
* **$ra ($31):** Thanh ghi địa chỉ trả về (return address).

**5. Branch delay slot là gì và tại sao nó quan trọng trong MIPS?**

**Câu trả lời:**  
**Branch delay slot** là một đặc điểm của kiến trúc MIPS, nơi lệnh ngay sau lệnh nhảy (branch) vẫn được thực thi, ngay cả khi nhảy được thực hiện. Điều này xảy ra vì thiết kế pipeline của MIPS không thể dừng ngay lập tức sau khi gặp lệnh nhảy, do lệnh kế tiếp đã được nạp vào pipeline.

Do đó, lập trình viên hoặc trình biên dịch phải đảm bảo rằng lệnh nằm trong **branch delay slot** không gây ra bất kỳ vấn đề nào hoặc có thể là một lệnh hữu ích để tận dụng tối đa pipeline. Ví dụ, lệnh trong delay slot có thể là một lệnh an toàn để thực hiện bất kể nhánh có thành công hay không.

**6. Lệnh giả (pseudo-instruction) trong MIPS là gì?**

**Câu trả lời:**  
**Lệnh giả** là các lệnh có vẻ như là một lệnh đơn khi lập trình, nhưng thực chất chúng không phải là lệnh trực tiếp trong tập lệnh MIPS. Lệnh giả thường được trình biên dịch hoặc assembler chuyển thành một hoặc nhiều lệnh MIPS thực sự.

Ví dụ:

* Lệnh giả move $t1, $t2 (sao chép giá trị từ $t2 sang $t1) sẽ được dịch thành lệnh thực sự add $t1, $t2, $zero (cộng $t2 với $zero, kết quả là $t2 được sao chép sang $t1).

**7. Tại sao MIPS chỉ sử dụng các lệnh load/store để truy cập bộ nhớ?**

**Câu trả lời:**  
MIPS được thiết kế theo triết lý **RISC** (Reduced Instruction Set Computing), trong đó các lệnh đơn giản và hiệu quả là ưu tiên hàng đầu. Một trong những nguyên tắc của RISC là **load/store architecture**, tức là chỉ các lệnh load và store mới có thể truy cập bộ nhớ.

Các lệnh khác (như cộng, trừ, nhân, v.v.) chỉ hoạt động trên các giá trị trong thanh ghi. Điều này giúp đơn giản hóa thiết kế của bộ xử lý và tối ưu hóa tốc độ thực thi, vì việc truy cập bộ nhớ thường chậm hơn nhiều so với thao tác trên thanh ghi.

**8. Kỹ thuật forwarding (chuyển tiếp) trong pipeline MIPS là gì?**

**Câu trả lời:**  
**Forwarding** (hay còn gọi là **data forwarding** hoặc **data bypassing**) là một kỹ thuật được sử dụng để **giảm thiểu xung đột dữ liệu** trong pipeline. Khi một lệnh cần kết quả từ lệnh trước, nhưng lệnh trước chưa hoàn thành toàn bộ chu kỳ pipeline, forwarding cho phép kết quả trung gian (chưa được ghi hẳn vào thanh ghi) được chuyển trực tiếp từ giai đoạn EX hoặc MEM đến giai đoạn EX của lệnh cần sử dụng dữ liệu.

Kỹ thuật này giúp tránh việc phải tạm dừng pipeline (stall) khi một lệnh cần sử dụng kết quả của lệnh trước.

**9. Kiến trúc MIPS có hỗ trợ xử lý ngắt (interrupts) không?**

**Câu trả lời:**  
Có, MIPS hỗ trợ xử lý ngắt thông qua **Coprocessor 0**, còn được gọi là **System Control Coprocessor** (CP0). CP0 có các thanh ghi đặc biệt để quản lý ngắt và ngoại lệ (exceptions). Khi xảy ra ngắt hoặc ngoại lệ, bộ xử lý sẽ lưu trạng thái hiện tại và nhảy đến trình xử lý ngắt để xử lý sự kiện. Sau khi xử lý xong, nó có thể trở lại tiếp tục thực thi chương trình đã bị tạm dừng.

**10. Làm thế nào để xử lý control hazards trong pipeline của MIPS?**

**Câu trả lời:**  
**Control hazards** (xung đột điều khiển) xảy ra chủ yếu khi có các lệnh nhảy (branch). Để giảm thiểu tác động của control hazards, MIPS sử dụng một số kỹ thuật:

* **Branch delay slot:** Như đã đề cập, MIPS cho phép lệnh ngay sau lệnh nhảy được thực thi, bất kể kết quả của lệnh nhảy.
* **Branch prediction (dự đoán nhánh):** Bộ xử lý dự đoán xem lệnh nhảy có xảy ra hay không và tiếp tục nạp lệnh dựa trên dự đoán. Nếu dự đoán đúng, pipeline tiếp tục hoạt động mượt mà; nếu dự đoán sai, các lệnh sai phải bị hủy và pipeline phải khởi động lại từ đúng nhánh.
* **Static prediction:** Một phương pháp đơn giản hơn là luôn dự đoán rằng nhánh sẽ không xảy ra, hoặc dựa trên một quy tắc cố định (ví dụ: luôn dự đoán nhánh có điều kiện sẽ không nhảy trừ khi điều kiện rõ ràng).

Quy trình thực thi lệnh:

1. Giai đoạn nạp lệnh (Instruction Fetch):

CPU nạp lệnh từ bộ nhớ dựa trên địa chỉ được lưu trữ trong Bộ đếm chương trình (Program Counter - PC).

PC sẽ được tăng lên sau mỗi lần nạp lệnh để trỏ đến lệnh tiếp theo.

Lệnh nạp được sẽ đi vào thanh ghi lệnh (Instruction Register) và được giải mã trong giai đoạn tiếp theo.

2. Giai đoạn giải mã lệnh (Instruction Decode):

Lệnh chứa nhiều trường khác nhau (opcode, rs, rt, rd, immediate), và opcode sẽ được đưa vào khối Control.

Khối Control sẽ phân tích opcode và tạo ra các tín hiệu điều khiển cho các thành phần khác trong CPU, bao gồm các thanh ghi, ALU, và bộ nhớ.

Dưới đây là phân tích kỹ hơn về các tín hiệu điều khiển, cách chúng hoạt động và lấy dữ liệu trong quá trình thực thi các lệnh khác nhau:

1. RegDst (Register Destination):

Vai trò: Quyết định thanh ghi đích sẽ nhận dữ liệu sau khi lệnh được thực thi.

Cách hoạt động:

Đối với các lệnh kiểu R (như add, sub, and, or), thanh ghi đích sẽ là trường rd (vị trí [15-11] trong lệnh).

Đối với các lệnh kiểu I (như lw, addi), thanh ghi đích sẽ là trường rt (vị trí [20-16] trong lệnh).

Ví dụ:

Lệnh add $t1, $t2, $t3:

Opcode của lệnh này là kiểu R, vì vậy tín hiệu RegDst sẽ chọn thanh ghi đích là rd (ở đây là $t1).

Lệnh lw $t1, 0($t2):

Đây là lệnh kiểu I, nên RegDst sẽ chọn thanh ghi đích là rt (ở đây là $t1).

2. Branch:

Vai trò: Quyết định CPU có thực hiện nhảy (branch) đến một địa chỉ mới nếu điều kiện nhánh đúng hay không.

Cách hoạt động:

Khi gặp các lệnh kiểu nhánh có điều kiện (như beq, bne), tín hiệu Branch sẽ được kích hoạt (Branch = 1).

Bộ so sánh (Comparator) sẽ kiểm tra xem hai thanh ghi nguồn (rs, rt) có bằng nhau (đối với beq) hoặc không bằng nhau (đối với bne). Nếu điều kiện đúng, địa chỉ nhảy sẽ được tính toán và cập nhật vào PC.

Ví dụ:

Lệnh beq $t1, $t2, label:

Nếu giá trị của $t1 bằng $t2, tín hiệu Branch sẽ được kích hoạt để CPU nhảy đến địa chỉ được tính toán từ label.

3. MemRead (Memory Read):

Vai trò: Điều khiển việc đọc dữ liệu từ bộ nhớ.

Cách hoạt động:

Tín hiệu MemRead sẽ được bật (MemRead = 1) khi CPU cần đọc dữ liệu từ bộ nhớ, thường gặp trong các lệnh nạp dữ liệu từ bộ nhớ (như lw - load word).

Địa chỉ bộ nhớ để đọc được tính toán bằng cách cộng giá trị của thanh ghi nguồn (rs) với hằng số (immediate).

Dữ liệu lấy từ bộ nhớ sẽ được đưa vào thanh ghi đích.

Ví dụ:

Lệnh lw $t1, 0($t2):

CPU sẽ đọc dữ liệu từ địa chỉ bộ nhớ được tính bằng $t2 + 0, và sau đó tín hiệu MemRead sẽ được bật để đọc dữ liệu từ địa chỉ này và lưu vào $t1.

4. MemtoReg (Memory to Register):

Vai trò: Quyết định dữ liệu nào sẽ được ghi vào thanh ghi đích.

Cách hoạt động:

Khi lệnh nạp dữ liệu từ bộ nhớ (lw), tín hiệu MemtoReg sẽ được bật (MemtoReg = 1), và dữ liệu từ bộ nhớ sẽ được ghi vào thanh ghi đích.

Đối với các lệnh kiểu R (như add, sub), dữ liệu từ ALU sẽ được ghi vào thanh ghi đích (MemtoReg = 0).

Ví dụ:

Lệnh lw $t1, 0($t2):

Tín hiệu MemtoReg sẽ được bật, và dữ liệu từ bộ nhớ sẽ được ghi vào thanh ghi $t1.

5. ALUOp (ALU Operation):

Vai trò: Điều khiển loại thao tác mà ALU sẽ thực hiện.

Cách hoạt động:

ALUOp là một tín hiệu mã hóa nhiều bit, cho phép CPU chọn loại phép toán mà ALU sẽ thực hiện (cộng, trừ, AND, OR, so sánh, v.v.).

Các loại lệnh khác nhau sẽ yêu cầu các thao tác khác nhau từ ALU. Ví dụ:

Lệnh add yêu cầu ALU thực hiện phép cộng.

Lệnh sub yêu cầu ALU thực hiện phép trừ.

Lệnh beq yêu cầu ALU so sánh hai giá trị để kiểm tra xem chúng có bằng nhau không.

Ví dụ:

Lệnh add $t1, $t2, $t3:

Tín hiệu ALUOp sẽ yêu cầu ALU thực hiện phép cộng giữa hai giá trị $t2 và $t3.

6. MemWrite (Memory Write):

Vai trò: Điều khiển việc ghi dữ liệu vào bộ nhớ.

Cách hoạt động:

Tín hiệu MemWrite sẽ được bật (MemWrite = 1) khi CPU cần ghi dữ liệu vào bộ nhớ, thường gặp trong các lệnh lưu dữ liệu vào bộ nhớ (như sw - store word).

Địa chỉ bộ nhớ để ghi được tính toán bằng cách cộng giá trị của thanh ghi nguồn (rs) với hằng số (immediate), và dữ liệu để ghi sẽ lấy từ thanh ghi đích.

Ví dụ:

Lệnh sw $t1, 0($t2):

Tín hiệu MemWrite sẽ được bật, và dữ liệu từ $t1 sẽ được ghi vào địa chỉ được tính bằng $t2 + 0.

7. ALUSrc (ALU Source):

Vai trò: Quyết định nguồn dữ liệu thứ hai (operand) cho ALU.

Cách hoạt động:

Đối với các lệnh kiểu R (như add, sub), dữ liệu thứ hai cho ALU sẽ được lấy từ thanh ghi rt.

Đối với các lệnh kiểu I (như addi, lw, sw), dữ liệu thứ hai cho ALU sẽ là một giá trị hằng số (immediate).

Ví dụ:

Lệnh addi $t1, $t2, 5:

Tín hiệu ALUSrc sẽ được bật, và giá trị thứ hai cho ALU sẽ là hằng số 5 thay vì giá trị từ một thanh ghi khác.

8. RegWrite (Register Write):

Vai trò: Điều khiển việc ghi dữ liệu vào một thanh ghi.

Cách hoạt động:

Tín hiệu RegWrite sẽ được bật (RegWrite = 1) khi CPU cần ghi dữ liệu vào một thanh ghi đích, thường gặp trong các lệnh như add, sub, lw, v.v.

Dữ liệu để ghi có thể là từ ALU (đối với các lệnh toán học) hoặc từ bộ nhớ (đối với các lệnh nạp dữ liệu).

Ví dụ:

Lệnh add $t1, $t2, $t3:

Tín hiệu RegWrite sẽ được bật để ghi kết quả của phép cộng vào thanh ghi $t1.

Tóm tắt:

Khối Control hoạt động như một "nhà điều hành" trong CPU, kiểm soát cách các thành phần khác nhau tương tác với nhau dựa trên loại lệnh được thực thi. Mỗi tín hiệu điều khiển có một vai trò cụ thể trong việc điều phối các hoạt động như đọc/ghi bộ nhớ, thực hiện phép toán trên ALU, và cập nhật các thanh ghi. Tất cả những tín hiệu này đều dựa vào opcode của lệnh để tạo ra cách điều khiển phù hợp cho từng loại lệnh.

Tín hiệu ALUOp là một tín hiệu điều khiển nhiều bit được tạo ra bởi khối Control và được sử dụng để chỉ định loại phép toán mà ALU (Arithmetic Logic Unit) sẽ thực hiện. Tuy nhiên, ALUOp không trực tiếp chỉ ra phép toán cụ thể mà ALU sẽ thực hiện mà thay vào đó, nó kết hợp với các bit chức năng (funct) từ lệnh kiểu R để xác định chính xác thao tác.

Các trường hợp tín hiệu của ALUOp:

Trong tập lệnh MIPS và kiến trúc tương tự, ALUOp thường có 2 hoặc 3 bit. Dưới đây là các trường hợp thông thường của ALUOp và cách nó hoạt động với các loại lệnh khác nhau:

1. ALUOp = 00 (Phép cộng - Dùng cho lệnh Load/Store)

Giải thích: Khi ALUOp = 00, ALU sẽ thực hiện phép cộng. Điều này được sử dụng cho các lệnh như:

lw (load word): Nạp dữ liệu từ bộ nhớ vào thanh ghi.

sw (store word): Lưu dữ liệu từ thanh ghi vào bộ nhớ.

Hoạt động: Với lệnh lw hoặc sw, ALU cần tính toán địa chỉ bộ nhớ bằng cách cộng giá trị của thanh ghi cơ sở (base register) và giá trị hằng số (immediate). Do đó, ALU sẽ luôn thực hiện phép cộng trong trường hợp này.

Ví dụ:

Lệnh lw $t1, 4($t2):

ALU sẽ thực hiện phép cộng: $t2 + 4 để tính địa chỉ bộ nhớ.

Lệnh sw $t1, 8($t2):

ALU sẽ thực hiện phép cộng: $t2 + 8 để tính địa chỉ bộ nhớ.

2. ALUOp = 01 (Phép trừ - Dùng cho lệnh Branch)

Giải thích: Khi ALUOp = 01, ALU sẽ thực hiện phép trừ. Điều này được sử dụng cho các lệnh nhánh có điều kiện:

beq (branch if equal): Nhảy đến một nhãn nếu hai thanh ghi bằng nhau.

bne (branch if not equal): Nhảy đến một nhãn nếu hai thanh ghi không bằng nhau (có thể cần mở rộng khối điều khiển với tín hiệu so sánh).

Hoạt động: Đối với lệnh beq, ALU sẽ thực hiện phép trừ giữa hai thanh ghi nguồn (rs và rt):

Nếu kết quả phép trừ bằng 0 (hai thanh ghi bằng nhau), tín hiệu nhánh sẽ được kích hoạt, và CPU sẽ nhảy đến nhãn chỉ định.

Nếu kết quả khác 0, CPU sẽ tiếp tục thực hiện các lệnh tuần tự.

Ví dụ:

Lệnh beq $t1, $t2, label:

ALU thực hiện phép trừ $t1 - $t2.

Nếu kết quả là 0, CPU sẽ nhảy đến địa chỉ được tính từ label.

3. ALUOp = 10 (Dùng cho các lệnh kiểu R - Xác định bằng trường funct)

Giải thích: Khi ALUOp = 10, ALU sẽ thực hiện một phép toán cụ thể dựa trên trường funct của lệnh kiểu R. Các lệnh kiểu R sử dụng trường funct (6 bit cuối của lệnh) để xác định chính xác phép toán mà ALU cần thực hiện.

Hoạt động: Với ALUOp = 10, khối điều khiển sẽ chuyển tiếp các bit funct từ lệnh để xác định phép toán cụ thể cho ALU. Các giá trị thường gặp của trường funct bao gồm:

100000 (add): Phép cộng.

100010 (sub): Phép trừ.

100100 (and): Phép AND bitwise.

100101 (or): Phép OR bitwise.

101010 (slt): So sánh nhỏ hơn (set less than).

Ví dụ:

Lệnh add $t1, $t2, $t3:

ALUOp = 10, và trường funct = 100000 chỉ ra rằng ALU cần thực hiện phép cộng.

Lệnh sub $t1, $t2, $t3:

ALUOp = 10, và trường funct = 100010 chỉ ra rằng ALU cần thực hiện phép trừ.

4. ALUOp = 11 (Tùy chọn mở rộng, ít được sử dụng)

Giải thích: Một số thiết kế CPU có thể mở rộng thêm tín hiệu ALUOp = 11 để thực hiện các hoạt động đặc biệt hoặc lệnh tùy chỉnh (custom instructions). Điều này không thường xuất hiện trong thiết kế cơ bản của MIPS, nhưng có thể được sử dụng trong các kiến trúc khác.

A black and white screen with white text

Description automatically generated

A screenshot of a computer

Description automatically generated

**Ví dụ chi tiết**:

* **Lệnh**: add $t1, $t2, $t3
  + **Opcode**: 000000 (Lệnh kiểu R)
  + **funct**: 100000 (add)
  + Khi **ALUOp = 10**, các bit funct = 100000 sẽ chỉ dẫn ALU thực hiện phép cộng $t2 + $t3.
* **Lệnh**: sub $t1, $t2, $t3
  + **Opcode**: 000000 (Lệnh kiểu R)
  + **funct**: 100010 (sub)
  + Khi **ALUOp = 10**, các bit funct = 100010 sẽ chỉ dẫn ALU thực hiện phép trừ $t2 - $t3.

**Tóm lại:**

Tín hiệu **ALUOp** đóng vai trò quan trọng trong việc chỉ định loại phép toán mà **ALU** sẽ thực hiện. Nó hoạt động dựa trên loại lệnh:

* **ALUOp = 00**: ALU thực hiện phép cộng, thường dùng cho các lệnh load/store.
* **ALUOp = 01**: ALU thực hiện phép trừ, thường dùng cho các lệnh nhánh.
* **ALUOp = 10**: ALU thực hiện các phép toán phức tạp hơn cho lệnh kiểu R, dựa trên giá trị của trường funct.

**1. MemRead (Memory Read)**

* **Vai trò**: Tín hiệu **MemRead** điều khiển việc **đọc dữ liệu từ bộ nhớ**. Khi tín hiệu này được bật (MemRead = 1), CPU sẽ thực hiện một hành động đọc dữ liệu từ một địa chỉ cụ thể trong bộ nhớ.
* **Khi nào MemRead = 1**:
  + Tín hiệu này thường được kích hoạt trong các lệnh **nạp dữ liệu (load)** từ bộ nhớ, chẳng hạn như lệnh **lw (load word)**.
  + Khi CPU cần nạp dữ liệu từ bộ nhớ vào một thanh ghi, **MemRead** sẽ được bật để cho phép đọc dữ liệu từ bộ nhớ.
* **Ví dụ**:
  + Lệnh **lw $t1, 0($t2)**:
    - CPU sẽ đọc dữ liệu từ địa chỉ bộ nhớ tính toán từ thanh ghi $t2 + 0.
    - Tín hiệu **MemRead** = 1 để cho phép CPU đọc dữ liệu từ bộ nhớ.
* **Tóm tắt**:
  + **MemRead = 1**: CPU sẽ đọc dữ liệu từ bộ nhớ.
  + **MemRead = 0**: Không có hoạt động đọc bộ nhớ.

**2. MemtoReg (Memory to Register)**

* **Vai trò**: Tín hiệu **MemtoReg** quyết định **dữ liệu nào sẽ được ghi vào thanh ghi đích** sau khi lệnh được thực thi. Nó chọn giữa:
  + **Dữ liệu từ bộ nhớ** (khi MemtoReg = 1), hoặc
  + **Kết quả từ ALU** (khi MemtoReg = 0).
* **Khi nào MemtoReg = 1**:
  + Tín hiệu này được kích hoạt trong các lệnh nạp dữ liệu từ bộ nhớ, như lệnh **lw**, khi dữ liệu đọc từ bộ nhớ cần được ghi vào một thanh ghi đích.
* **Khi nào MemtoReg = 0**:
  + Trong các lệnh số học hoặc logic (lệnh kiểu R như add, sub, and, or), kết quả từ ALU sẽ được ghi trực tiếp vào thanh ghi đích, và tín hiệu **MemtoReg = 0**.
* **Ví dụ**:
  + Lệnh **lw $t1, 0($t2)**:
    - Sau khi CPU đọc dữ liệu từ bộ nhớ, tín hiệu **MemtoReg** = 1 để ghi dữ liệu từ bộ nhớ vào thanh ghi $t1.
  + Lệnh **add $t1, $t2, $t3**:
    - Kết quả của ALU (phép cộng $t2 + $t3) sẽ được ghi vào thanh ghi $t1, và tín hiệu **MemtoReg** = 0, vì dữ liệu từ ALU được chọn.
* **Tóm tắt**:
  + **MemtoReg = 1**: Dữ liệu từ bộ nhớ sẽ được ghi vào thanh ghi.
  + **MemtoReg = 0**: Dữ liệu từ ALU sẽ được ghi vào thanh ghi.

**3. MemWrite (Memory Write)**

* **Vai trò**: Tín hiệu **MemWrite** điều khiển việc **ghi dữ liệu vào bộ nhớ**. Khi tín hiệu này được bật (MemWrite = 1), CPU sẽ thực hiện hành động ghi dữ liệu từ một thanh ghi vào một địa chỉ cụ thể trong bộ nhớ.
* **Khi nào MemWrite = 1**:
  + Tín hiệu này được kích hoạt trong các lệnh **lưu dữ liệu (store)**, chẳng hạn như lệnh **sw (store word)**. CPU sẽ ghi dữ liệu từ một thanh ghi vào một địa chỉ bộ nhớ được tính toán.
* **Ví dụ**:
  + Lệnh **sw $t1, 0($t2)**:
    - CPU sẽ ghi dữ liệu từ thanh ghi $t1 vào địa chỉ bộ nhớ tính toán từ $t2 + 0.
    - Tín hiệu **MemWrite** = 1 để cho phép CPU ghi dữ liệu vào bộ nhớ.
* **Tóm tắt**:
  + **MemWrite = 1**: CPU sẽ ghi dữ liệu từ thanh ghi vào bộ nhớ.
  + **MemWrite = 0**: Không có hoạt động ghi bộ nhớ.

A black and white screen with white text

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a black and white text

Description automatically generated

**Kết luận:**

* **MemRead**: Được sử dụng khi CPU **đọc dữ liệu từ bộ nhớ** (ví dụ: lệnh lw).
* **MemtoReg**: Điều khiển việc **chọn dữ liệu từ bộ nhớ** hoặc **ALU** để ghi vào thanh ghi đích. Nó thường được sử dụng trong các lệnh **nạp dữ liệu từ bộ nhớ (load)**.
* **MemWrite**: Được sử dụng khi CPU **ghi dữ liệu vào bộ nhớ** (ví dụ: lệnh sw).

Số lượng tín hiệu của khối ALU Control và tín hiệu ALU Control có sự khác nhau, và chúng phục vụ các mục đích khác nhau trong hệ thống xử lý MIPS. Để hiểu rõ hơn, chúng ta cần phân biệt giữa "khối ALU Control" và "tín hiệu ALU Control".

1. Khối ALU Control (ALU Control Unit)

Khối ALU Control là một phần của Control Unit (đơn vị điều khiển), chịu trách nhiệm tạo ra các tín hiệu điều khiển cho ALU (Arithmetic Logic Unit). Khối này nhận vào các tín hiệu từ Control Unit chính, bao gồm:

ALUOp (thường là 2 bit): Tín hiệu này đến từ Control Unit và quyết định kiểu lệnh mà ALU sẽ thực hiện (lệnh R-type, lệnh lw, sw, beq, v.v.).

Funct field (thường là 6 bit): Đây là trường funct trong lệnh R-type, chỉ định phép toán cụ thể (như add, sub, and, or, v.v.), được khối ALU Control giải mã.

Đầu ra của khối ALU Control là tín hiệu ALU Control (thường là 4 bit), được đưa vào ALU để điều khiển phép toán mà ALU sẽ thực hiện.

Ví dụ, khối ALU Control có thể nhận các đầu vào như:

ALUOp = 10 (cho lệnh R-type)

Funct field = 100000 (cho phép cộng add)

Khối ALU Control sẽ giải mã và tạo ra tín hiệu ALU Control phù hợp, ví dụ: tín hiệu ALU Control = 0010 (đại diện cho phép cộng).

2. Tín hiệu ALU Control

Tín hiệu ALU Control là tín hiệu đầu ra từ khối ALU Control, thường là 4 bit, và được đưa trực tiếp vào ALU để điều khiển phép toán mà ALU thực hiện. Mỗi mã nhị phân của tín hiệu ALU Control sẽ đại diện cho một phép toán cụ thể mà ALU phải thực hiện, chẳng hạn như:

0000: Phép AND

0001: Phép OR

0010: Phép Cộng (add)

0110: Phép Trừ (sub)

0111: Phép Set on Less Than (slt)

1100: Phép NOR

Sự khác biệt về số lượng tín hiệu:

Khối ALU Control nhận vào tín hiệu ALUOp (thường là 2 bit) và Funct field (thường là 6 bit). Vì vậy, số lượng tín hiệu đầu vào cho khối ALU Control thường là 8 bit (2 bit ALUOp và 6 bit Funct field).

Tín hiệu ALU Control là tín hiệu đầu ra từ khối ALU Control, thường có 4 bit (đại diện cho các phép toán cơ bản mà ALU có thể thực hiện).

Tóm tắt:

Khối ALU Control nhận vào các tín hiệu điều khiển từ Control Unit (thường là 8 bit, bao gồm 2 bit ALUOp và 6 bit Funct field từ lệnh R-type).

Tín hiệu ALU Control (thường là 4 bit) là tín hiệu đầu ra từ khối ALU Control và được đưa vào ALU để điều khiển các phép toán cụ thể mà ALU sẽ thực hiện.

Vì vậy, số lượng tín hiệu của khối ALU Control (8 bit) khác với số lượng tín hiệu của tín hiệu ALU Control (4 bit).