**TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN ĐHQG-HCM**

**KHOA ĐIỆN TỬ VIỄN THÔNG**

**🙢✰🙠**

**Ảnh có chứa biểu tượng, Phông chữ, văn bản, Nhãn hiệu

Mô tả được tạo tự động**

**BÁO CÁO ĐỒ ÁN**

**Đề tài: CPU 4-bit**

**Giảng viên hướng dẫn :** Nguyễn Thị Thiên Trang

**Nhóm :** 8\_Above

**Lớp :** BASIC DIGITAL IC DESIGN

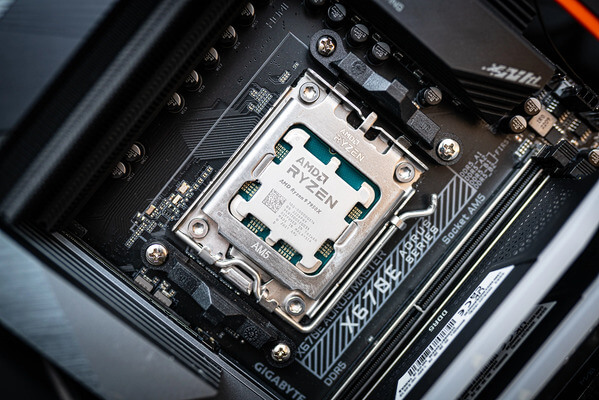
**Thành phố Hồ Chí Minh, tháng 8 năm 2023**

**THÀNH VIÊN NHÓM**

|  |  |
| --- | --- |
| **Họ và tên** | **MSSV** |
| 20200239 | Lê Tuấn Kiệt |
| 20200240 | Nguyễn Tuấn Kiệt |
| 20200241 | Phạm Tuấn Kiệt |
| 20200246 | Dương Đình Linh (Leader) |
| 20200251 | Võ Thành Lộc |
| 20200252 | Huỳnh Văn Vũ Luân |

1. **ĐỊNH NGHĨA**

CPU (Central Processing Unit) là bộ xử lý trung tâm của máy tính, nó thực hiện các phép tính số học, logic, so sánh và các hoạt động nhập/xuất dữ liệu (Input/Output) cơ bản từ mã lệnh được định sẵn trong một máy tính.

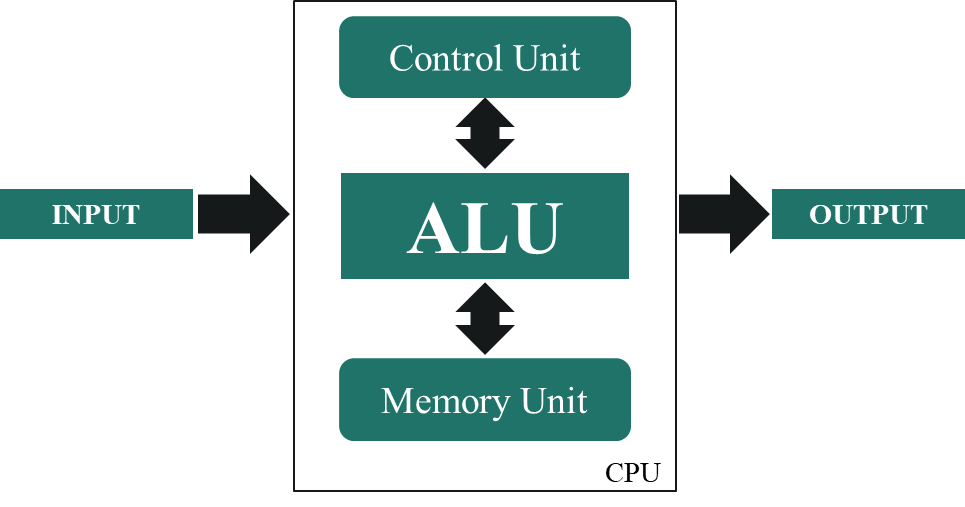


**AMD Ryzen 9 7950X**

Ảnh có chứa văn bản, đồ điện tử, Phông chữ, bộ nhớ flash

Mô tả được tạo tự động

1. **CẤU TRÚC**
2. ***Sơ đồ***

****

1. ***Bộ nhớ - Memory***

Đơn vị này có thể lưu trữ hướng dẫn, dữ liệu và kết quả trung gian. Bộ phận này cung cấp thông tin cho các bộ phận khác của máy tính khi cần thiết. Nó còn được gọi là đơn vị lưu trữ nội bộ hoặc bộ nhớ chính hoặc bộ lưu trữ chính hoặc Bộ nhớ truy cập ngẫu nhiên (RAM).

Kích thước của nó ảnh hưởng đến tốc độ, sức mạnh và khả năng. Bộ nhớ chính và bộ nhớ phụ là hai loại bộ nhớ trong máy tính. Các chức năng của bộ nhớ là :

* Nó lưu trữ tất cả dữ liệu và các hướng dẫn cần thiết để xử lý.
* Nó lưu trữ các kết quả trung gian của quá trình xử lý.
* Nó lưu trữ các kết quả cuối cùng của quá trình xử lý trước khi các kết quả này được đưa ra thiết bị đầu ra.

1. ***Đơn vị điều khiển – Control Unit (CU)***

Bộ phận này kiểm soát hoạt động của tất cả các bộ phận của máy tính nhưng không thực hiện bất kỳ hoạt động xử lý dữ liệu thực tế nào.

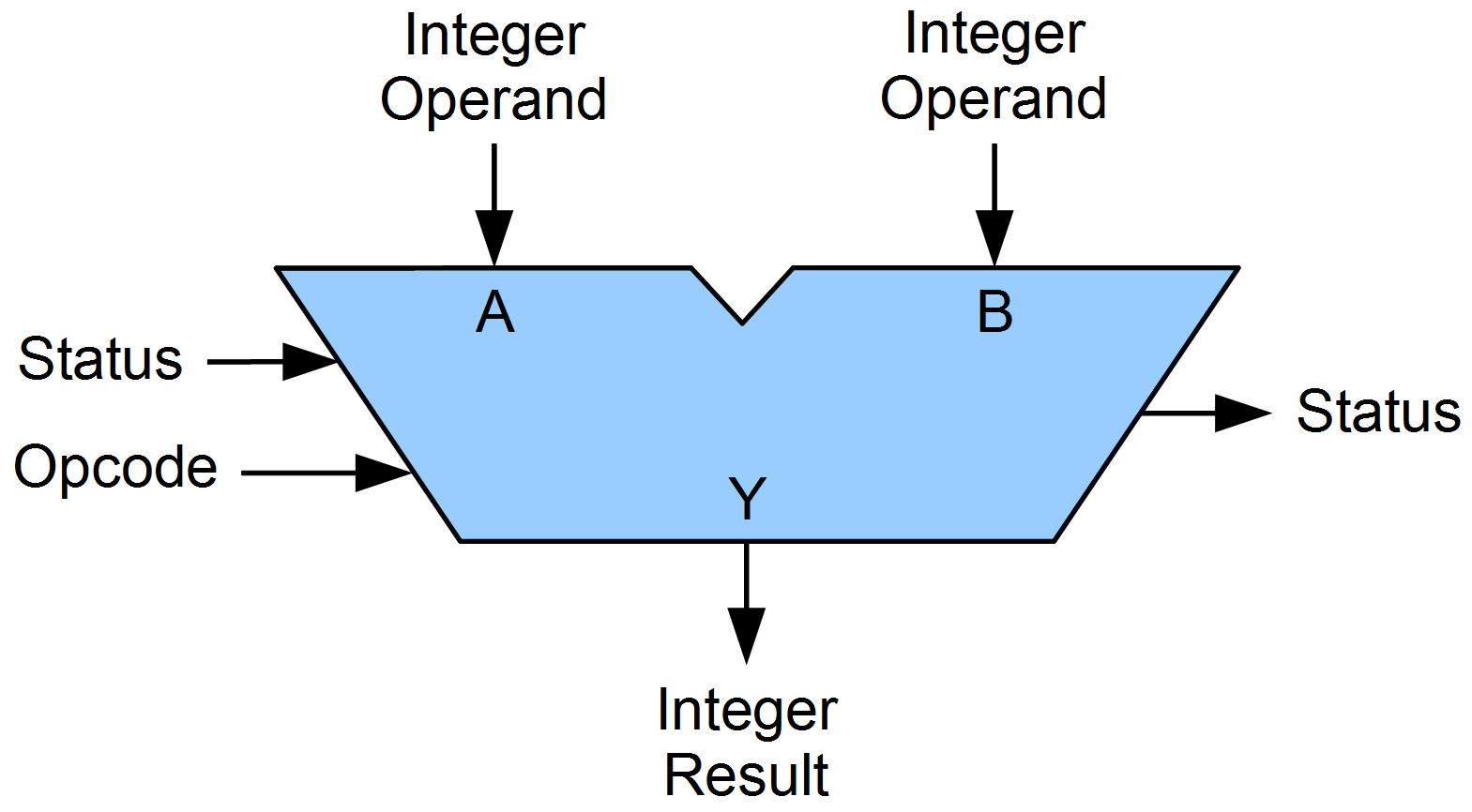
Chức năng của đơn vị này là:

* Nó chịu trách nhiệm kiểm soát việc truyền dữ liệu và hướng dẫn giữa các đơn vị khác của máy tính.
* Nó quản lý và điều phối tất cả các đơn vị của máy tính.
* Nó nhận các hướng dẫn từ bộ nhớ, diễn giải chúng và chỉ đạo hoạt động của máy tính.
* Nó giao tiếp với các thiết bị Đầu vào / Đầu ra để chuyển dữ liệu hoặc kết quả từ bộ lưu trữ.
* Nó không xử lý hoặc lưu trữ dữ liệu.

1. ***Đơn vị luận lý số học – ALU***

Đơn vị này bao gồm hai phần phụ : Phần số học, Phần logic

ALU là một mạch số trong bộ xử lý thực hiện các phép toán số học, các hoạt động logic. Các đầu vào của ALU là các dữ liệu được vận hành (được gọi là toán hạng), thông tin trạng thái từ các hoạt động trước đó và mã (opcode) từ đơn vị điều khiển chỉ ra hoạt động nào cần thực hiện. Tùy thuộc vào lệnh đang thực hiện, các toán hạng có thể đến từ thanh ghi nội hoặc bộ nhớ ngoại, hoặc có thể là hằng số được tạo ra bởi ALU.



Khi tất cả các tín hiệu ngõ vào đã được định vị và truyền qua mạch ALU, kết quả hoạt động được thực hiện sẽ xuất hiện ở các đầu ra của ALU. Kết quả bao gồm data work, có thể được lưu trữ trong thanh ghi hoặc bộ nhớ, và thông tin trạng thái thường được lưu trữ trong một thanh ghi CPU đặc biệt, dành riêng cho mục đích này.

Một vài chức năng cơ bản:

* phép toán số học : cộng, trừ, nhân and chia.
* phép toán logic : and, or, nor ,..
* dịch bit , xoay bit.
* giải mã ( decoder ).
* Các phép toán phức tạp.
* Vào và Ra.

1. ***Full Adder:***

Là mạch tổ hợp kỹ thuật số được sử dụng để thực hiện phép cộng hai số nhị phân và cộng thêm một bit nhớ từ phép cộng trước đó. Trong ngữ cảnh của CPU 4-bit, Full Adder thường được sử dụng để thực hiện các phép tính số học như cộng trên các dãy bit 4-bit. Nó được sử dụng trong bộ xử lý máy tính, đơn vị logic số học (ALU) và các thiết bị điện tử khác thực hiện các phép toán.

Chức năng của Full Adder trong CPU 4-bit:

1. *Phép cộng số học:* Full Adder thực hiện phép cộng hai số nhị phân có độ dài 1 bit, bao gồm cả bit nhớ từ phép cộng trước đó (carry-in).
2. *Tính toán bit cộng (carry-out):* Full Adder tính toán bit nhớ (carry-out) từ phép cộng. Nếu tổng của hai bit cộng và bit nhớ từ phép cộng trước lớn hơn hoặc bằng 2, thì bit nhớ sẽ được truyền cho phép cộng tiếp theo.

Thành phần của Full Adder trong CPU 4-bit:

1. *Ba đầu vào (A, B, Cin):* Ba đầu vào của Full Adder lần lượt là hai bit cần cộng (A và B) và bit nhớ từ phép cộng trước (carry-in, thường ký hiệu là Cin).
2. *Hai đầu ra (Sum, Cout):* Full Adder có hai đầu ra chính. Một là kết quả tổng của phép cộng (Sum), và một là bit nhớ để chuyển cho phép cộng tiếp theo (carry-out, ký hiệu là Cout).
3. *Ba cổng XOR và một cổng AND:* Full Adder được xây dựng bằng cách sử dụng các cổng logic XOR (hoặc XNOR) và AND. Các cổng này được kết hợp để tính toán kết quả tổng (Sum) và bit nhớ (Cout).
4. *Mạch logic:* Bộ phận chính của Full Adder là mạch logic, gồm các cổng XOR và AND, được kết hợp theo cách phù hợp để tính toán kết quả Sum và Cout dựa trên ba đầu vào.

Ứng dụng trong: Bộ ghép kênh, Đơn vị logic số học (ALU), Phần cứng.

1. ***Multiplexer:***

Ta thấy MUX hoạt động như 1 công tắc nhiều vị trí được điều khiển bởi mã số dạng số nhị phân, tuỳ tổ hợp số nhị phân này mà ở bất kì thời điểm nào chỉ có 1 ngõ vào được chọn và cho phép đưa tới ngõ ra. Các mạch dồn kênh thường gặp là 2 sang 1, 4 sang 1, 8 sang 1, …Nói chung là từ 2n sang 1.

Chức năng của Mux trong CPU 4-bit:

*Lựa chọn dữ liệu* - cho phép chọn một trong các nguồn dữ liệu đầu vào để đưa ra đầu ra duy nhất. Điều này cho phép CPU 4-bit chọn dữ liệu từ các nguồn khác nhau để thực hiện các phép tính, quản lý bộ nhớ và thực hiện các hoạt động khác.

Thành phần của Mux trong CPU 4-bit:

1. *Tín hiệu đầu vào:* Tượng trưng cho các nguồn dữ liệu muốn chọn.
2. *Tín hiệu đầu ra:* Đây là đầu ra chính của Mux, mang giá trị của nguồn dữ liệu được chọn.
3. *Các tín hiệu điều khiển:* Đây là các tín hiệu điều khiển (control signals) xác định nguồn dữ liệu nào sẽ được chọn làm đầu ra. Sự kết hợp của các tín hiệu này xác định nguồn nào sẽ được chọn.

*Mạch logic:* Mux được xây dựng bằng cách sử dụng các cổng logic như cổng AND và cổng OR. Mạch logic này thực hiện chức năng chọn dữ liệu dựa trên các tín hiệu điều khiển.

Ứng dụng :

* *Mở rộng kênh ghép:* Các mạch ghép kênh ít ngõ vào có thể được kết hợp với nhau để tạo mạch ghép kênh nhiều ngõ vào.
* *Chuyển đổi song song sang nối tiếp:*

Các dữ liệu nhị phân nhiều bit, chẳng hạn mã ASCII, word, … thường được xử lí song song cùng một lúc. Trong máy tính, dữ liệu được di chuyển từ nơi này đến nơi khác cùng 1 lúc trên các đường dẫn điện song song gọi là các bus. Khi dữ liệu được truyền đi qua khoảng cách dài chẳng hạn hàng chục mét thì cách truyền song song không còn thích hợp vì tốn nhiều đường dây, rồi nhiễu, …. Lúc này mạch dồn kênh có thể dùng như mạch chuyển đổi song song sang nối tiếp tương tự như mạch ghi dịch mà ta đã xét ở phần trước.

* *Dùng dồn kênh để thiết kế tổ hợp*: Các mạch dồn kênh với hoạt động logic như đã xét ở trước ngoài cách dùng để ghép nhiều đường ngõ vào còn có thể dùng để thiết kế mạch tổ hợp đôi khi rất dễ dàng vì:
* Không cần phải đơn giản biểu thức nhiều
* Thường dùng ít IC
* Dễ thiết kế

1. ***Decoder 3-to-8:***

Một Decoder 3-to-8 là một mạch logic có nhiệm vụ chuyển đổi một dãy ba tín hiệu đầu vào thành một trong tám đầu ra tương ứng. Trong ngữ cảnh của một CPU 4-bit, Decoder 3-to-8 có thể được sử dụng để chọn một trong tám tín hiệu điều khiển dựa trên một dãy ba bit đầu vào. Đây là một phần quan trọng của hệ thống điều khiển của CPU, giúp xác định tín hiệu điều khiển nào sẽ được kích hoạt dựa trên các tín hiệu đầu vào.

Chức năng của Decoder 3-to-8 trong CPU 4-bit:

*Chọn tín hiệu điều khiển:* CPU cần sử dụng nhiều tín hiệu điều khiển để quản lý các hoạt động khác nhau, chẳng hạn như chọn đích đến cho dữ liệu, quản lý các hoạt động bộ nhớ, hay điều khiển các phép tính và lưu trữ. Decoder 3-to-8 giúp xác định tín hiệu điều khiển cần được kích hoạt dựa trên một dãy ba bit đầu vào.

Thành phần của Decoder 3-to-8 trong CPU 4-bit:

1. *Đầu vào (Input):* Decoder 3-to-8 có ba đầu vào nhận các tín hiệu đầu vào dạng bit (0 hoặc 1). Các đầu vào này xác định tín hiệu điều khiển cần được kích hoạt.
2. *Đầu ra (Output):* Có tám đầu ra tương ứng với tất cả các tổ hợp có thể của ba tín hiệu đầu vào. Mỗi đầu ra có thể được kích hoạt khi tín hiệu đầu vào tương ứng được kích hoạt. Đầu ra có thể là tín hiệu điều khiển cần để thực hiện các hoạt động trong CPU, chẳng hạn như chọn thanh ghi, quản lý bộ nhớ, hay kích hoạt các phép tính.
3. *Mạch Logic Decoder:* Bộ phận quan trọng nhất của Decoder 3-to-8 là mạch logic nằm trong đó, có chức năng xác định đầu ra nào sẽ được kích hoạt dựa trên các tín hiệu đầu vào. Mạch logic này thực hiện việc giải mã (decode) dãy ba bit đầu vào thành một tín hiệu điều khiển tương ứng.

Thông thường bộ giải mã được xây dựng bằng cổng NAND - trong trường hợp đó, đầu ra được chọn là 0 và các đầu ra khác là 1. Đôi khi bộ giải mã cũng có đầu vào kích hoạt - nếu nó không đúng - có nghĩa là một trong các đầu ra đã được chọn.

Decoder cũng được sử dụng để xây dựng các mạch khác như Multiplexer hay Demultiplexer.

1. ***Kiến trúc RISC và CISC:***

Hiện nay, sự chuyển dịch của người dùng sang các thiết bị điện toán di động đã làm gia tăng nhu cầu về một bộ xử lí với triết lí ưu tiên tiết kiệm năng lượng hơn là triết lí ưu tiên hiệu suất cao. Sự đối lập trong 2 triết lí thiết kế này được thể hiện sâu sắc qua 2 kiến trúc là RISC (Reduced Instruction Set Computing) và (Complex Instructions Set Computer).

1. ***RISC****:* là một phương pháp thiết kế các bộ vi xử lý theo hướng đơn giản hóa tập lệnh, trong đó thời gian thực thi tất cả các lệnh đều như nhau. Hiện nay các bộ vi xử lý RISC phổ biến là ARM, SuperH, MIPS, SPARC, DEC Alpha, PA-RISC, PIC và PowerPC của IBM.

* Ưu điểm
* Tốc độ xử lý nhanh hơn: Kiến trúc RISC sử dụng các lệnh đơn giản và cơ bản giúp giảm thời gian xử lý và tăng tốc độ của CPU.
* Hiệu quả về mặt năng lượng: Vì các lệnh đơn giản hơn, CPU sử dụng ít năng lượng hơn để thực hiện các tác vụ, giúp tiết kiệm năng lượng và kéo dài tuổi thọ của PIN.
* Dễ dàng để thiết kế và xây dựng: Kiến trúc RISC đơn giản, dễ dàng hơn để thiết kế và xây dựng, giúp giảm chi phí sản xuất và tăng tính đáng tin cậy của CPU.
* Dễ dàng để tối ưu hóa: Vì các lệnh đơn giản hơn, kiến trúc RISC dễ dàng để tối ưu hóa và cải thiện hiệu suất của CPU.
* Dễ dàng để mở rộng: Kiến trúc RISC dễ dàng để mở rộng và thêm các tính năng mới, giúp tăng tính linh hoạt và đa dạng của CPU.
* Nhược điểm
* Số lượng lệnh giới hạn: Kiến trúc RISC giới hạn số lượng lệnh có sẵn, điều này có thể làm giảm khả năng linh hoạt của CPU trong việc xử lý các tác vụ phức tạp.
* Chi phí cao: Thiết kế CPU RISC yêu cầu nhiều bộ nhớ và bộ đệm để lưu trữ các lệnh và dữ liệu, điều này có thể làm tăng chi phí sản xuất.
* Khó khăn trong việc lập trình: Vì số lượng lệnh giới hạn nên việc lập trình trên kiến trúc RISC có thể khó khăn hơn so với kiến trúc CISC (Complex Instruction Set Computing).
* Không tối ưu cho các ứng dụng đa nhiệm: Kiến trúc RISC không được tối ưu cho các ứng dụng đa nhiệm, điều này có thể làm giảm hiệu suất của CPU trong các tác vụ đòi hỏi xử lý đa nhiệm.

1. ***CSIC:*** là một kiểu kiến trúc máy tính trong đó các lệnh máy tính được thiết kế để thực hiện nhiều chức năng khác nhau. Kiến trúc CISC có thể thực hiện các lệnh phức tạp chỉ bằng một lệnh duy nhất, giúp giảm thiểu số lượng lệnh cần thiết để thực hiện một tác vụ. Kiến trúc CISC cũng có thể sử dụng bộ nhớ trực tiếp để truy cập dữ liệu giúp tăng tốc độ xử lý.

* Ưu điểm
* Đa chức năng: Kiến trúc CISC có thể thực hiện nhiều chức năng khác nhau trong một lệnh duy nhất, giúp tối ưu hóa thời gian thực hiện các tác vụ.
* Dễ sử dụng: Kiến trúc CISC có số lượng lệnh lớn giúp cho việc lập trình và sử dụng dễ dàng hơn.
* Tiết kiệm bộ nhớ: Kiến trúc CISC có thể lưu trữ nhiều thông tin trong một lệnh duy nhất, giúp tiết kiệm bộ nhớ và tăng tốc độ thực hiện các tác vụ.
* Hỗ trợ các lệnh phức tạp: Kiến trúc CISC có thể hỗ trợ các lệnh phức tạp như lệnh điều khiển dòng lệnh xử lý chuỗi, lệnh xử lý số thực giúp cho việc xử lý dữ liệu đa dạng trở nên dễ dàng hơn.
* Tương thích ngược: Kiến trúc CISC có thể tương thích ngược với các phiên bản cũ hơn giúp cho việc nâng cấp và bảo trì hệ thống dễ dàng hơn.
* Nhược điểm
* Phức tạp: Kiến trúc CISC có nhiều lệnh phức tạp và đa dạng, điều này dẫn đến việc thiết kế triển khai và bảo trì phần cứng phức tạp hơn so với kiến trúc RISC.
* Tốc độ chậm: Do có nhiều lệnh phức tạp, kiến trúc CISC thường có tốc độ xử lý chậm hơn so với kiến trúc RISC.
* Không hiệu quả với các ứng dụng đòi hỏi tính toán đơn giản: Kiến trúc CISC được thiết kế để xử lý các tác vụ phức tạp, do đó nó không hiệu quả với các ứng dụng đòi hỏi tính toán đơn giản.
* Không linh hoạt: Kiến trúc CISC không linh hoạt trong việc mở rộng và nâng cấp, do đó nó không phù hợp với các ứng dụng đòi hỏi tính linh hoạt và mở rộng.

1. **THÔNG SỐ KĨ THUẬT**
2. ***Clock rate***

Khi xem cấu hình máy tính, phần CPU thường có các thông số CPU 1.2 GHz, 2.2GHz,… đây chính là **Clock rate**.

Clock rate là tần số xung clock được tạo ra bởi một mạch dao động bên ngoài, nó sẽ tạo ra một số xung nhất định mỗi giây theo dạng sóng vuông. Tần số xung clock xác định tốc độ CPU thực hiện các lệnh, do đó, tốc độ xung clock càng nhanh thì càng có nhiều lệnh mà CPU thực hiện mỗi giây. Để đảm bảo hoạt động thích hợp của CPU, thời gian xung clock phải dài hơn thời gian tối đa cần cho tất cả các tín hiệu truyền đi qua CPU. Việc thiết lập thời gian xung clock cho một giá trị lớn hơn thời gian truyền tín hiệu trong trường hợp xấu nhất, có thể thiết kế toàn bộ CPU và cách nó di chuyển dữ liệu xung quanh cạnh của tín hiệu xung clock (cạnh lên hoặc xuống). Điều này có lợi thế là làm đơn giản hóa CPU đáng kể, từ góc độ thiết kế lẫn các thành phần. Tuy nhiên, nó cũng mang đến một bất lợi là CPU phải chờ đợi yếu tố chậm nhất của nó, mặc dù các thành phần khác nhanh hơn nhiều. Hạn chế này được bù đắp bằng các phương pháp tính toán song song của CPU. Để tránh trì hoãn bởi một tín hiệu duy nhất, CPU hiện đại đòi hỏi nhiều tín hiệu xung clock được cung cấp. Tốc độ xung clock ngày càng cao trong các CPU làm cho việc giữ tín hiệu đồng bộ trong toàn đơn vị trở nên khó khăn hơn và khi tốc đọ xung clock tăng lên đáng kể thì lượng nhiệt phát ra bởi CPU cũng tăng lên.

Tần số xung clock càng cao thì tốc độ CPU càng nhanh. Tuy nhiên, tốc độ xung clock không thể tăng lên vô hạn. Khi tốc độ xung clock tăng lên đáng kể, lượng nhiệt phát ra bởi CPU cũng tăng lên. Điều này có thể dẫn đến các vấn đề về nhiệt độ và ảnh hưởng đến hiệu suất của CPU. Ngoài ra, khi tốc độ xung clock ngày càng cao, việc giữ tín hiệu đồng bộ trong toàn đơn vị trở nên khó khăn hơn. Điều này dẫn đến CPU hiện đại đòi hỏi nhiều tín hiệu xung clock được cung cấp để tránh trì hoãn bởi một tín hiệu duy nhất, đủ làm toàn bộ CPU hoạt động sai .

1. ***Integer range***

Mỗi CPU đều được biểu diễn các giá trị bằng một số cách cụ thể. Ví dụ, các máy tính kỹ thuật số ban đầu được biểu diễn bằng các giá trị thập phân (cơ số 10) quen thuộc, một số khác sử dụng cách biểu diễn khác như termary (cơ sở 3). Gần như tất cả các CPU hiện đại biểu diễn bằng các số trong giá trị nhị phân, với mỗi chữ số là bội số của 2. Ví dụ như 4, 8, 12, 16, 32, 64, thậm chí 128,..

Các con số này nói lên kích thước và độ chính xác của số nguyên mà một CPU có thể biểu diễn. Trong trường hợp của một CPU nhị phân, nó được đo bằng số bit mà CPU có thể xử lý trong một thao tác, chúng thường được gọi là “word size”, “bit width”, “data path width”, “integer precision”, “integer size”. Chúng xác định phạm vi giá trị của các số nguyên mà nó có thể tác động trực tiếp. Ví dụ, một CPU 8-bit, có thể trực tiếp thao tác với các số nguyên được biểu diễn bởi 8-bit, nghĩa là 256 (28) giá trị rời rạc.

Ngoài ra, “word size” cũng ảnh hưởng đến số vị trí bộ nhớ mà CPU có thể trực tiếp định địa địa chỉ (một địa chỉ là một số nguyên đại diện cho một vị trí bộ nhớ cụ thể). Ví dụ, một CPU nhị phân sử dụng 32 bit để đại diện cho một địa chỉ bộ nhớ thì nó có thể trực tiếp giải quyết 2^32 địa chỉ bộ nhớ. Để phá vỡ các giới hạn này và vi nhiều lý do khác nhau, một số CPU sử dụng các cơ chế (bank switch) cho phép bổ sung bộ nhớ.

Các CPU có “word size” lớn hơn đòi hỏi nhiều mạch hơn và do đó có kích thước lớn hơn, chi phí nhiều hơn, và cũng tiêu tốn nhiều điện hơn (và do đó tạo ra nhiều nhiệt hơn). Kết quả là các bộ vi điều khiển 4 hoặc 8 bit thường được sử dụng trong các ứng dụng hiện đại, mặc dù các CPU có “word size” lớn hơn (như 16, 32, 64 hay thậm chí 128) đều có sẵn. Tuy nhiên, khi hệ thống cần một hiệu suất lớn hơn, thì người ta có thể bỏ qua những bất lợi đó để sử dụng các CPU có “word size” lớn hơn.

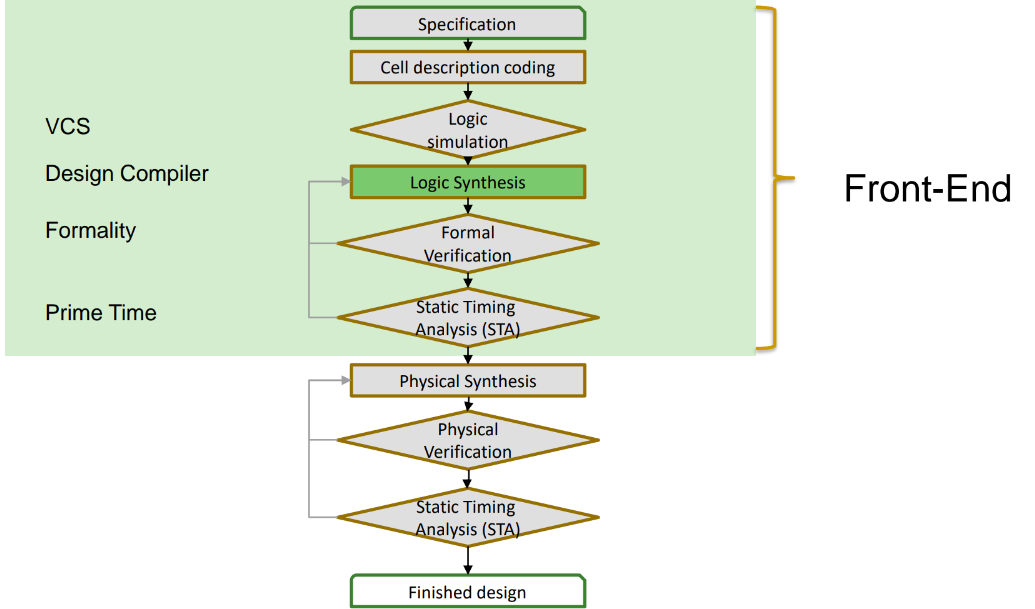
Để đạt một số lợi thế do cả độ dài bit thấp hơn và cao hơn, nhiều bộ chỉ lệnh có độ rộng bit khác nhau cho dữ liệu số nguyên và số thực, cho phép các CPU thực hiện các lệnh đó có “data path width” khác nhau cho các thành phần khác nhau của thiết bị. Ví dụ, tập lệnh IBM System / 360 chủ yếu là 32 bit, nhưng hỗ trợ các giá trị số thực 64 bit để tạo sự chính xác và phạm vi lớn hơn trong số thực. Nhiều thiết kế CPU sau đó sử dụng chiều rộng bit hỗn hợp, nơi cần có sự cân bằng hợp lý cho cả số nguyên lẫn số thực.

1. **THIẾT KẾ**
2. **Các lớp của CPU:**

**Ảnh có chứa văn bản, ảnh chụp màn hình, biểu đồ, Phông chữ

Mô tả được tạo tự động**

1. **Quy trình thiết kế:**

****

1. **Project:**
   1. **Cell description coding:**

module CPU(clk, reset, DataIn, opcode, A, B, DataOut, OpAddOut, ZF, CF, SF, halt);

input clk, reset; // Đầu vào xung clock và tín hiệu reset

input [3:0] DataIn; // Đầu vào dữ liệu 4-bit

input [7:0] opcode; // Đầu vào opcode 8-bit

output reg [3:0] OpAddOut, DataOut, A, B; // Đầu ra các thanh ghi và dữ liệu

output reg ZF, CF, SF, halt; // Đầu ra các cờ và tín hiệu dừng

reg [3:0] DATA[15:0], STACK[15:0], DSTACK[15:0]; // Khai báo các thanh ghi và bộ nhớ

reg [3:0] address, tmp, tmpd, tmpadd; // Khai báo các biến trung gian

always @(posedge clk) begin // Luôn thực thi khi có cạnh dương của xung đồng hồ

if (reset) halt = 0; // Đặt `halt` về 0 nếu tín hiệu `reset` được kích hoạt

CF = 0; // Đặt cờ tràn CF về 0

ZF = 0; // Đặt cờ zero ZF về 0

SF = 0; // Đặt cờ dấu SF về 0

if (!halt) begin // Nếu không bị dừng, thực hiện các hoạt động

case(opcode[7:4]) // Lựa chọn lệnh dựa trên 4 bit đầu của opcode

4'b0000: begin // Lệnh 4'b0000

B = opcode[3:0]; // Lấy giá trị B từ 4 bit cuối của opcode

if (A+B>15) CF = 1; // Nếu A + B lớn hơn 15, đặt cờ tràn CF

A = A + B; // Thực hiện phép cộng A và B

if (A==0) ZF = 1; // Nếu A bằng 0, đặt cờ zero ZF

4'b0001: begin

B = opcode[3:0]; // Lấy giá trị B từ 4 bit cuối của opcode

if (A<B) SF = 1; // Nếu A nhỏ hơn B, đặt cờ dấu SF

A = A - B; // Thực hiện phép trừ A và B

if (A==0) ZF = 1; // Nếu A bằng 0, đặt cờ zero ZF

end

4'b0010: begin

tmp = B; // Sao chép giá trị của B vào biến tạm thời tmp

B = A; // Sao chép giá trị của A vào B

A = tmp; // Sao chép giá trị của biến tạm thời tmp vào A

if (A==0) ZF = 1; // Nếu A bằng 0, đặt cờ zero ZF

end

4'b0011: begin

DATA[opcode[3:0]] = 4'b0101; // Gán giá trị 4'b0101 vào bộ nhớ DATA tại địa chỉ được chỉ định bởi 4 bit cuối của opcode

DataOut = DATA[opcode[3:0]]; // Đọc giá trị từ bộ nhớ DATA và đặt vào DataOut

A = DATA[opcode[3:0]]; // Gán giá trị từ bộ nhớ DATA vào thanh ghi A

if (A==0) ZF = 1; // Nếu A bằng 0, đặt cờ zero ZF

if (A<0) SF = 1; // Nếu A âm, đặt cờ dấu SF

end

4'b0100: begin

DATA[opcode[3:0]] = B; // Gán giá trị của thanh ghi B vào bộ nhớ DATA tại địa chỉ được chỉ định bởi 4 bit cuối của opcode

DataOut = B; // Đặt giá trị của B vào thanh ghi DataOut

end

4'b0101: begin

DataOut = A; // Đặt giá trị của thanh ghi A vào thanh ghi DataOut

if (A==0) ZF = 1; // Nếu A bằng 0, đặt cờ zero ZF

if (A<0) SF = 1; // Nếu A âm, đặt cờ dấu SF

end

4'b0110: begin

B = opcode[3:0]; // Lấy giá trị B từ 4 bit cuối của opcode

if (!A & B) ZF = 1; // Nếu (NOT A) AND B là đúng, đặt cờ zero ZF

end

4'b0111: begin

B = B | DATA[opcode[3:0]]; // Thực hiện phép OR giữa thanh ghi B và giá trị từ bộ nhớ DATA tại địa chỉ được chỉ định bởi 4 bit cuối của opcode

DataOut = B; // Đặt giá trị của B vào thanh ghi DataOut

End

4'b1000: begin

if (ZF==0) begin // Kiểm tra cờ zero ZF, nếu không đặt (không bằng 0)

OpAddOut = STACK[tmpadd];// Lấy giá trị từ bộ nhớ STACK tại địa chỉ được chỉ định bởi biến tmpadd và đặt vào thanh ghi OpAddOut

tmpadd = tmpadd + 1; // Tăng giá trị của biến tmpadd lên 1

end

end

4'b1001: begin

OpAddOut = STACK[tmpadd]; // Lấy giá trị từ bộ nhớ STACK tại địa chỉ được chỉ định bởi biến tmpadd và đặt vào thanh ghi OpAddOut

tmpadd = tmpadd + 1; // Tăng giá trị của biến tmpadd lên 1

end

4'b1010: begin

A = DataIn; // Đặt giá trị của đầu vào DataIn vào thanh ghi A

if (A==0) ZF = 1; // Nếu A bằng 0, đặt cờ zero ZF

if (A<0) SF = 1; // Nếu A âm, đặt cờ dấu SF

end

4'b1011: begin

DSTACK[tmpd] = A; // Gán giá trị của thanh ghi A vào bộ nhớ DSTACK tại địa chỉ được chỉ định bởi biến tmpd

A = 0; // Đặt giá trị của thanh ghi A về 0

tmpd = tmpd + 1; // Tăng giá trị của biến tmpd lên 1

end

4'b1100: begin

tmpd = tmpd - 1; // Giảm giá trị của biến tmpd đi 1

A = DSTACK[tmpd]; // Lấy giá trị từ bộ nhớ DSTACK tại địa chỉ được chỉ định bởi biến tmpd và đặt vào thanh ghi A

tmpd = tmpd + 1; // Tăng giá trị của biến tmpd lên 1

end

**Giải thích chức năng:**

4'b1101: begin

OpAddOut = STACK[opcode[3:0]]; // Lấy giá trị từ bộ nhớ STACK tại địa chỉ được chỉ định bởi 4 bit cuối của opcode và đặt vào thanh ghi OpAddOut

end

4'b1110: begin

tmpd = 0; // Đặt giá trị của biến tmpd về 0

tmpadd = 0; // Đặt giá trị của biến tmpadd về 0

end

4'b1111: halt = 1; // Đặt tín hiệu dừng halt thành 1

endcase: // Kết thúc mệnh đề case, đã thực hiện xử lý cho opcode hiện tại

DATA[address] = A; // Lưu giá trị của thanh ghi A vào bộ nhớ DATA tại địa chỉ được chỉ định bởi biến address

STACK[address] = opcode[7:1]; // Lưu giá trị của các bit từ 7 đến 1 của opcode vào bộ nhớ STACK tại địa chỉ được chỉ định bởi biến address

address = address + 1; // Tăng giá trị của biến address lên 1 để chuẩn bị cho việc lưu trữ vào các địa chỉ tiếp theo trong bộ nhớ DATA và STACK

end

end

endmodule

1. **Lưu trữ Dữ liệu và Chương Trình**: CPU này có khả năng lưu trữ dữ liệu trong các thanh ghi **DATA**, **STACK**, và **DSTACK**. Nó cũng lưu trữ chương trình (opcode) trong thanh ghi **STACK**.
2. **Xử lý Lệnh Opcode**: CPU thực hiện xử lý các lệnh opcode (8-bit) được cung cấp thông qua cổng **opcode**. Các lệnh opcode được thực hiện bằng cách sử dụng một câu lệnh **case**, với mỗi lệnh opcode tương ứng với một trường hợp trong câu lệnh **case**.
3. **Các Lệnh Thực Hiện**:
   * **Phép Cộng (Opcode: 4'b0000)**: Lệnh này thực hiện phép cộng giữa giá trị **A** và giá trị được lấy từ 4 bit cuối của opcode (biến **B**). Nếu kết quả cộng lớn hơn 15, cờ tràn (**CF**) được đặt thành 1. Nếu kết quả cộng bằng 0, cờ zero (**ZF**) được đặt thành 1.
   * **Phép Trừ (Opcode: 4'b0001)**: Lệnh này thực hiện phép trừ giữa giá trị **A** và giá trị **B** lấy từ 4 bit cuối của opcode. Nếu **A** nhỏ hơn **B**, cờ dấu (**SF**) được đặt thành 1. Nếu kết quả trừ bằng 0, cờ zero (**ZF**) được đặt thành 1.
   * **Hoán Đổi Giá Trị (Opcode: 4'b0010)**: Lệnh này hoán đổi giá trị của **A** và **B**.
   * **Ghi Dữ Liệu Vào DATA (Opcode: 4'b0011)**: Lệnh này ghi giá trị **4'b0101** vào một vị trí trong bộ nhớ DATA, sau đó cập nhật giá trị **DataOut** và **A** với giá trị này. Nếu giá trị **A** bằng 0, cờ zero (**ZF**) được đặt thành 1, và nếu giá trị **A** là số âm, cờ dấu (**SF**) được đặt thành 1.
   * **Ghi Dữ Liệu Vào DATA (Opcode: 4'b0100)**: Lệnh này ghi giá trị **B** vào một vị trí trong bộ nhớ DATA và cập nhật **DataOut** với giá trị này.
   * **Đọc Dữ Liệu từ DATA (Opcode: 4'b0101)**: Lệnh này đặt **DataOut** bằng giá trị của **A**. Nếu giá trị **A** bằng 0, cờ zero (**ZF**) được đặt thành 1, và nếu giá trị **A** là số âm, cờ dấu (**SF**) được đặt thành 1.
   * **Kiểm Tra Điều Kiện (Opcode: 4'b0110)**: Lệnh này kiểm tra giá trị **A** và **B**. Nếu cả hai đều bằng 0, cờ zero (**ZF**) được đặt thành 1.
   * **Phép OR (Opcode: 4'b0111)**: Lệnh này thực hiện phép OR giữa giá trị **B** và một giá trị từ bộ nhớ DATA, sau đó lưu kết quả vào **DataOut**.
   * **Đọc Giá Trị Từ STACK (Opcode: 4'b1000 và 4'b1001)**: Đọc giá trị từ bộ nhớ STACK và lưu vào **OpAddOut**. Điều này xảy ra dựa trên điều kiện ZF.
   * **Gán Giá Trị DataIn Vào A (Opcode: 4'b1010)**: Lệnh này gán giá trị từ cổng **DataIn** vào **A**. Nếu giá trị **A** bằng 0, cờ zero (**ZF**) được đặt thành 1, và nếu giá trị **A** là số âm, cờ dấu (**SF**) được đặt thành 1.
   * **Ghi Dữ Liệu vào DSTACK (Opcode: 4'b1011)**: Lệnh này ghi giá trị **A** vào vị trí hiện tại trong bộ nhớ DSTACK và tăng địa chỉ.
   * **Đọc Dữ Liệu Từ DSTACK (Opcode: 4'b1100)**: Lệnh này giảm địa chỉ hiện tại trong bộ nhớ DSTACK và đọc giá trị **A** từ vị trí đó.
   * **Đọc Giá Trị Từ STACK (Opcode: 4'b1101)**: Đọc giá trị từ bộ nhớ STACK và lưu vào **OpAddOut** dựa trên giá trị của opcode.
   * **Khởi Tạo (Opcode: 4'b1110)**: Lệnh này đặt lại các biến trung gian **tmpd** và **tmpadd** về 0.
   * **Dừng (Opcode: 4'b1111)**: Lệnh này đặt tín hiệu dừng **halt** thành 1, dừng CPU.
4. **Cập Nhật Dữ Liệu và Địa Chỉ**: CPU này cập nhật dữ liệu trong bộ nhớ DATA và STACK dựa trên giá trị của **A**, **B**, và opcode. Địa chỉ cũng được tăng sau mỗi lệnh.
5. **Các Cờ**: CPU duy trì các cờ **ZF** (Zero Flag), **CF**
   1. **DC (Design Compiler)**

**Ảnh có chứa ảnh chụp màn hình, văn bản, phần mềm, Phần mềm đa phương tiện

Mô tả được tạo tự động**

**Ảnh có chứa ảnh chụp màn hình, Phần mềm đa phương tiện, màn hình, phần mềm

Mô tả được tạo tự động**

* 1. **STA (Static timing Analysis)**

**Ảnh có chứa ảnh chụp màn hình, văn bản, thiết kế

Mô tả được tạo tự động**

**Ảnh có chứa văn bản, màn hình, ảnh chụp màn hình, phần mềm

Mô tả được tạo tự động**