Trường Đại học Bách khoa Hà Nội
Bộ môn Điều khiển tự động

Tài liệu tóm tắt bài giảng

HỆ THỐNG ĐIỀU KHIỂN NHÚNG

(*Embedded Control Systems*)

TS. Lưu Hồng Việt

Nội dung

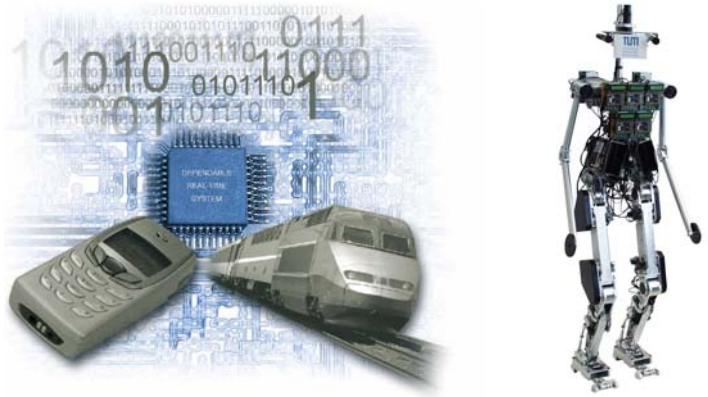
1 MỞ ĐẦU

Kỷ nguyên công nghệ mới đã và đang tiếp tục phát triển không ngừng nhằm thông minh hoá hiện đại hoá thông suốt các hệ thống. Có thể nói đánh dấu sự ra đời và phát triển của hệ nhúng trước tiên phải kể đến sự ra đời của các bộ vi xử lý, vi điều khiển. Nó được đánh dấu bởi sự ra đời của Chip vi xử lý đầu tiên 4004 vào năm 1971 cho mục đích tính toán thương mại bởi một công ty Nhật bản *Busicom* và sau đó đã được chắp cánh và phát triển vượt bậc bởi *Intel* để trở thành các bộ siêu xử lý như các Chip được ứng dụng cho PC như ngày nay. Thập kỷ 80 có thể được coi là khởi điểm bắt đầu kỷ nguyên của sự bùng nổ về phát triển các hệ nhúng. Từ đó khởi nguồn cho làn sóng ra đời của hàng loạt các chủng loại vi xử lý và gắn liền là các hệ nhúng để thâm nhập rộng khắp trong các ứng dụng hàng ngày của cuộc sống chúng ta ví dụ như, các thiết bị điện tử sử dụng cho sinh hoạt hàng ngày (lò vi sóng, TV, tủ lạnh, máy giặt, điều hoà ...) và văn phòng làm việc (máy fax, máy in, máy điện thoại...)... Các bộ vi xử lý và phần mềm cũng ngày càng được sử dụng rộng rãi trong rất nhiều các hệ thống nhỏ. Các loại vi xử lý được sử dụng trong các hệ thống nhúng hiện nay đã vượt xa so với PC về số lượng chủng loại (chiếm đến 79% số các vi xử lý đang tồn tại [2]) và vẫn còn tiếp tục phát triển để nhằm đáp ứng và thoả mãn rất nhiều ứng dụng đa dạng. Trong số đó vẫn còn ứng dụng cả các Chip vi xử lý 8 bit, 16 bit và hiện nay chủ yếu vẫn là 32 bit (chiếm khoảng 75%). Gắn liền với sự phát triển phần cứng, phần mềm cũng đã phát triển với tốc độ nhanh không thua kém thậm chí sẽ tăng nhanh hơn rất nhiều theo sự phát triển hệ nhúng.

1.1 Các khái niệm về hệ nhúng

- **Hệ nhúng ?**

Trong thế giới thực của chúng ta bất kỳ một thiết bị hay hệ thống điện/điện tử có khả năng xử lý thông tin và điều khiển đều có thể tiềm ẩn trong đó một thiết bị hay hệ nhúng, ví dụ như các thiết bị truyền thông, thiết bị đo lường điều khiển, các thiết bị phục vụ sinh hoạt hàng ngày như lò vi sóng, máy giặt, camera…Rất dễ dàng để có thể kể ra hàng loạt các thiết bị hay hệ thống như vậy đang tồn tại quanh ta, chúng là hệ nhúng. Vậy hệ nhúng thực chất là gì và nên hiểu thế nào về hệ nhúng? Hiện nay cũng chưa có một định nghĩa nào thực sự thoả đáng để được chuẩn hoá và thừa nhận rộng rãi cho hệ nhúng mà vẫn chỉ là những khái niệm diễn tả về chúng thông qua những đặc thù chung. Tuy nhiên ở đây chúng ta có thể hiểu ***hệ nhúng là một phần hệ thống xử lý thông tin nhúng trong các hệ thống lớn, phức hợp và độc lập*** ví dụ như trong ôtô, các thiết bị đo lường, điều khiển, truyền thông và thiết bị thông minh nói chung. Chúng là ***những tổ hợp của phần cứng và phần mềm để thực hiện một hoặc một nhóm chức năng chuyên biệt, cụ thể*** (Trái ngược với máy tính PC mà chúng ta thường thấy được sử dụng không phải cho một chức năng mà là rất nhiều chức năng hay phục vụ chung cho nhiều mục đích). PC thực chất lại là một hệ thống lớn, tổ hợp của nhiều hệ thống nhúng ví dụ như card màn hình, âm thanh, modem, ổ cứng, bàn phím…Chính điều này làm chúng ta dễ lúng túng nếu được hỏi nên hiểu thế nào về PC, có phải là hệ nhúng hay không.



Hình 1-1: Một vài hình ảnh về hệ nhúng

- **Hệ thời gian thực ?**

Trong các bài toán điều khiển và ứng dụng chúng ta rất hay gặp thuật ngữ "*thời gian thực*". Thời gian thực có phải là thời gian phản ánh về độ trung thực của thời gian hay không? Thời gian thực có phải là hiển thị chính xác và đồng bộ theo đúng như nhịp đồng hồ đếm thời gian hay không? Không phải hoàn toàn như vậy! Thực chất, theo cách hiểu nếu nói trong các hệ thống kỹ thuật đặc biệt các hệ thống yêu cầu khắt khe về sự ràng buộc thời gian, thời gian thực được hiểu là yêu cầu của hệ thống phải đảm bảo thoả mãn về tính tiền định trong hoạt động của hệ thống. Tính tiền định nói lên hành vi của hệ thống thực hiện đúng trong một khung thời gian cho trước hoàn toàn xác định. Khung thời gian này được quyết định bởi đặc điểm hoặc yêu cầu của hệ thống, có thể là vài giây và cũng có thể là vài nano giây hoặc nhỏ hơn nữa. Ở đây chúng ta phân biệt yếu tố thời gian gắn liền với khái niệm về thời gian thực. Không phải hệ thống thực hiện rất nhanh là sẽ đảm bảo được tính thời gian thực vì nhanh hay chậm hoàn toàn là phép so sánh có tính tương đối vì *mili* giây có thể là nhanh với hệ thống điều khiển nhiệt nhưng lại là chậm đối với các đối tượng điều khiển điện như dòng, áp…. Hơn thế nữa nếu chỉ nhanh không thì chưa đủ mà phải đảm bảo duy trì ổn định bằng một cơ chế hoạt động tin cậy. Chính vì vậy hệ thống không kiểm soát được hoạt động của nó (bất định) thì không thể là một hệ thống đảm bảo tính thời gian thực mặc dù hệ thống đó có thể cho đáp ứng rất nhanh, thậm chí nhanh hơn rất nhiều so với yêu cầu đặt ra. Một ví dụ minh hoạ tiêu biểu đó là cơ chế truyền thông dữ liệu qua đường truyền chuẩn *Ethernet* truyền thống, mặc dù ai cũng biết tốc độ truyền là rất nhanh nhưng vẫn không phải hệ hoạt động thời gian thực vì không thoả mãn tính tiền định trong cơ chế truyền dữ liệu (có thể là rất nhanh và cũng có thể là rất chậm nếu có sự canh tranh và giao thông đường truyền bị nghẽn).

Người ta phân ra làm hai loại đối với khái niệm thời gian thực là cứng (*hard real-time*) và mềm (*soft real-time*). ***Thời gian thực cứng là khi hệ thống hoạt động với yêu cầu thoả mãn sự ràng buộc trong khung thời gian cứng tức là nếu vi phạm thì sẽ dẫn đến hoạt động của toàn hệ thống bị sai hoặc bị phá huỷ***. Ví dụ về hoạt động điều khiển cho một lò phản ứng hạt nhân, nếu chậm ra quyết định có thể dẫn đến thảm hoạ gây ra do phản ứng phân hạch và dẫn đến bùng nổ cả hệ thống. ***Thời gian thực mềm là khi hệ thống hoạt động với yêu cầu thoả mãn ràng buộc trong khung thời gian mềm, nếu vi phạm và sai lệch nằm trong khoảng cho phép thì hệ thống vẫn có thể hoạt động được và chấp nhận được***. Ví dụ như hệ thống phát thanh truyền hình, nếu thông tin truyền đi từ trạm phát tới người nghe/nhìn chậm một vài giây thì cũng không ảnh hưởng đáng kể đến tính thời sự của tin được truyền đi và hoàn toàn được chấp nhận bởi người theo dõi.

Thực tế thấy rằng hầu hết hệ nhúng là các hệ thời gian thực và hầu hết các hệ thời gian thực là hệ nhúng. Điều này phản ánh mối quan hệ mật thiết giữa hệ nhúng và thời gian thực và tính thời gian thực đã trở thành như một thuộc tính tiêu biểu của hệ nhúng. Vì vậy hiện nay khi đề cập tới các hệ nhúng người ta đều nói tới đặc tính cơ bản của nó là tính thời gian thực.

Hình 1-2: Phân bố và quan hệ giữa hệ nhúng và thời gian thực

1.2 Lĩnh vực ứng dụng của hệ nhúng

Chúng ta có thể kể ra được rất nhiều các ứng dụng của hệ thống nhúng đang được sử dụng hiện nay, và xu thể sẽ còn tiếp tục tăng nhanh. Một số các lĩnh vực và sản phẩm thị trường rộng lớn của các hệ nhúng có thể được nhóm như sau:

- Các thiết bị điều khiển
- Ôtô, tàu điện
- Truyền thông
- Thiết bị y tế
- Hệ thống đo lường thẩm định
- Toà nhà thông minh
- Thiết bị trong các dây truyền sản xuất
- Rôbốt
- ...

1.3 Đặc điểm công nghệ và xu thế phát triển của hệ nhúng

1.3.1 Đặc điểm công nghệ

Các hệ thống như vậy đều có chung một số đặc điểm như yêu cầu về khả năng thời gian thực, độ tin cậy, tính độc lập và hiệu quả. Một câu hỏi đặt ra là tại sao hệ thống nhúng lại phát triển và được phổ cập một cách nhanh chóng như hiện nay. Câu trả lời thực ra nằm ở các yêu cầu tăng lên không ngừng trong các ứng dụng công nghệ hiện nay. Một trong những yêu cầu cơ bản đó là:

Khả năng độc lập và thông minh hoá: Điều này được chỉ rõ hơn thông qua một số các thuộc tính yêu cầu, cụ thể như:

- ☑ Độ tin cậy
- ☑ Khả năng bảo trì và nâng cấp
- ☑ Sự phổ cập và tiện sử dụng
- ☑ Độ an toàn
- ☑ Tính bảo mật

Hiệu quả: Yêu cầu này được thể hiện thông qua một số các đặc điểm của hệ thống như sau:

- ☑ Năng lượng tiêu thụ
- ☑ Kích thước về phần cứng và phần mềm
- ☑ Hiệu quả về thời gian thực hiện
- ☑ Kích thước và khối lượng
- ☑ Giá thành

Phân hoạch tác vụ và chức năng hoá: Các bộ vi xử lý trong các hệ nhúng thường được sử dụng để đảm nhiệm và thực hiện một hoặc một nhóm chức năng rất độc lập và cũng đặc thù cho từng phần chức năng của hệ thống lớn mà nó được nhúng vào. Ví dụ như một vi xử lý thực hiện một phần điều khiển cho một chức năng thu thập, xử lý và hiển thị của ôtô hay hệ thống điều khiển quá trính. Khả năng này làm tăng thêm sự chuyên biệt hoá về chức năng của một hệ thống lớn và dễ dàng hơn cho quá trính xây dựng, vận hành và bảo trì.

Khả năng thời gian thực: Các hệ thống đều gắn liền với việc đảm nhiệm một chức năng chính và phải được thực hiện đúng theo một khung thời gian qui định. Thông thường một chức năng của hệ thống phải được thực hiện và hoàn thành theo một yêu cầu thời gian định trước để đảm bảo thông tin cập nhật kịp thời cho phần xử lý của các chức năng khác và có thể ảnh hưởng trực tiếp tới sự hoạt động đúng và chính xác của toàn hệ thống. Tuỳ thuộc vào từng bài toán và yêu cầu của hệ thống mà yêu cầu về khả năng thời gian thực cũng rất khác nhau.

Tuy nhiên, trong thực tế không phải hệ nhúng nào cũng đều có thể thoả mãn tất cả những yêu cầu nêu trên, vì chúng là kết quả của sự thoả hiệp của nhiều yêu cầu và điều kiện nhằm ưu tiên cho chức năng cụ thể mà chúng được thiết kế. Chính điều này lại

càng làm tăng thêm tính chuyên biệt hoá của các hệ/thiết bị nhúng mà các thiết bị đa năng không thể cạnh tranh được.

1.3.2 Xu thế phát triển và sự tăng trưởng của hệ nhúng

Vì sự phát triển hệ nhúng là sự kết hợp nhuần nhuyễn giữa phần cứng và phần mềm nên công nghệ gắn liền với nó cũng chính là công nghệ kết hợp giữa các giải pháp cho phần cứng và mềm. Vì tính chuyên biệt của các thiết bị / hệ nhúng như đã giới thiệu nên các nền phần cứng cũng được chế tạo để ưu tiên đáp ứng cho chức năng hay nhiệm vụ cụ thể của yêu cầu thiết kế đặt ra.

Lớp hệ nhúng ưu tiên phát triển theo tiêu chí về kích thước nhỏ gọn, tiêu thụ năng lượng ít, giá thành thấp. Các chíp xử lý nhúng cho lớp hệ thống ứng dụng đó thường yêu cầu về khả năng tính toán ít hoặc vừa phải nên hầu hết được xây dựng trên cở sở bộ đồng xử lý 8 bít -16 bit hoặc cùng lắm là 32 bit và không hỗ trợ dấu phảy động do sự hạn chế về dung lượng và khả năng tính toán.

Lớp hệ nhúng ưu tiên thực thi khả năng xử lý tính toán với tốc độ thực hiện nhanh. Các chíp xử lý nhúng cho các hệ thống đó cũng sẽ là các Chip áp dụng các công nghệ cao cấp với kiến trúc xử lý song song để đáp ứng được cường độ tính toán lớn và tốc độ mà các Chip xử lý đa chức năng thông thường không đạt tới được.

Lớp hệ thống ưu tiên cả hai tiêu chí phát triển của hai lớp trên, tức là kích thước nhỏ gọn, mức tiêu thụ năng lượng thấp, tốc độ tính toán nhanh. Tuỳ theo sự thoả hiệp giữa các yêu cầu và xu thế phát triển chính vì vậy cũng không có gì ngạc nhiên khi chúng ta thấy sự tồn tại song song của rất nhiều các Chip vi xử lý nhúng, vi điều khiển nhúng 8 bit, 16 bit hay 32 bit cùng với các Chíp siêu xử lý khác vẫn đang được ứng dụng rộng rãi cho hệ nhúng. Đó cũng là sự kết hợp đa dạng và sự ra đời của các hệ nhúng nói chung nhằm thoả mãn các ứng dụng phát triển không ngừng.

Với mỗi một nền phần cứng nhúng thường có những đặc thù riêng và kèm theo một giải pháp phát triển phần mềm tối ưu tương ứng. Không có một giải pháp nào chung và chuẩn tắc cho tất cả các hệ nhúng. Chính vì vậy thông thường các nhà phát triển và cung cấp phần cứng cũng lại chính là nhà cung cấp giải pháp phần mềm hoặc công cụ phát triển phần mềm kèm theo. Rất phổ biến hiện nay các Chip vi xử lý hay vi điều khiển đều có các hệ phát triển (*Starter Kit* hay *Emulator*) để hỗ trợ cho các nhà ứng dụng và xây dựng hệ nhúng với hiểu biết hạn chế về phần cứng. Ngôn ngữ mã hoã phần mềm cũng thường là C hoặc gần giống như C (*Likely C*) thay vì phải viết hoàn toàn bằng hợp ngữ *Assembly*. Điều này cho phép các nhà thiết kế tối ưu và đơn giản hoá rất nhiều cho bước phát triển và xây dựng hệ nhúng.

Trong xu thế phát triển không ngừng và nhằm thoả mãn được nhu cầu phát triển nhanh và hiệu quả có rất nhiều các công nghệ cho phép thực thi các giải pháp hệ nhúng. Đứng sau sự phổ cập rộng rãi của các Chip vi xử lý vi điều khiển nhúng, DSP phải kể đến các công nghệ cũng đang rất được quan tâm hiện nay như ASIC, CPLD,

FPGA, PSOC và sự tổ hợp của chúng...Kèm theo đó là các kỹ thuật phát triển phần mềm cho phép đảm nhiệm được các bài toán yêu cầu khắt khe trên cơ sở một nền phần cứng hữu hạn về khả năng xử lý và không gian bộ nhớ. Giải quyết các bài toán thời gian thực như phân chia tác vụ và giải quyết cạnh tranh chia sẻ tài nguyên chung. Hiện nay cũng đã có nhiều nhà phát triển công nghệ phần mềm lớn đang hướng vào thị trường hệ nhúng bao gồm cả *Microsoft*. Ngoài một số các hệ điều hành *Windows* quen thuộc dùng cho PC, *Microsoft* cũng đã tung ra các phiên bản mini như *WindowsCE*, *WindowsXP Embedded* và các công cụ phát triển ứng dụng kèm theo để phục vụ cho các thiết bị nhúng, điển hình như các thiết bị PDA, một số thiết bị điều khiển công nghiệp như các máy tính nhúng, IPC của Siemens...

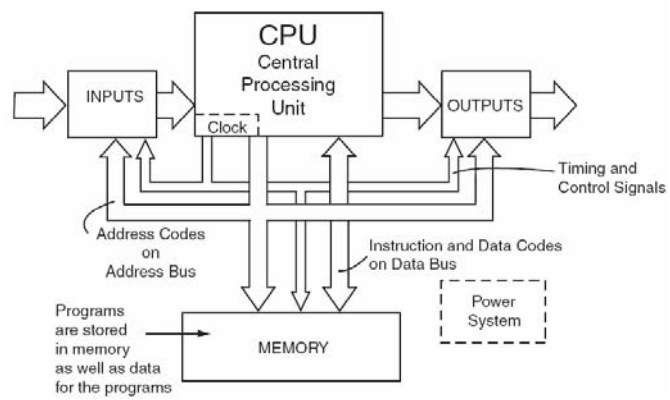
Có thể nói hệ nhúng đã trở thành một giải pháp công nghệ và phát triển một cách nhanh chóng, hứa hẹn nhiều thiết bị nhúng sẽ chiếm lĩnh được thị trường rộng lớn trong tương lai nhằm đáp ứng nhu cầu ứng dụng không ngừng trong cuộc sống của chúng ta. Đối với lĩnh vực công nghiệp về điều khiển và tự động hoá, hệ nhúng cũng là một giải pháp đầy tiềm năng đã và đang được ứng dụng rộng rãi. Nó rất phù hợp để thực thi các chức năng thông minh hoá, chuyên biệt trong các hệ thống và thiết bị công nghiệp, từ các hệ thống tập trung đến các hệ thống phân tán. Giải pháp hệ nhúng có thể thực thi từ cấp thấp nhất của hệ thống công nghiệp như cơ cấu chấp hành cho đến các cấp cao hơn như giám sát điều khiển quá trình.

1.4 Mục đích và nội dung môn học

Hệ điều khiển nhúng là một môn học mới nhằm cung cấp kiến thức cho sinh viên về khả năng phân tích và thiết kế hệ thống điều khiển và thông minh hoá hệ thống theo chức năng theo giải pháp công nghệ. Thiết kế thực thi điều khiển trên nền phần cứng nhúng.

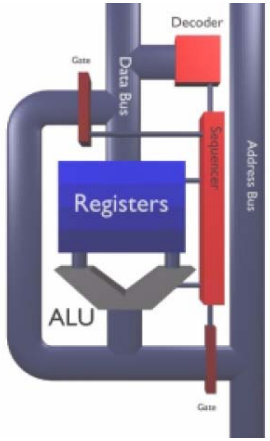
2 CẤU TRÚC PHẦN CỨNG HỆ NHÚNG

2.1 Các thành phần kiến trúc cơ bản



Hình 2-1: Kiến trúc điển hình của các chíp VXL/VĐK nhúng

2.1.1 Đơn vị xử lý trung tâm CPU



Hình 2-2: Cấu trúc CPU

Người ta vẫn biết tới phần lõi xử lý của các bộ VXL là đơn vị xử lý trung tâm CPU (*Central Processing Unit*) đóng vai trò như bộ não chịu trách nhiệm thực thi các phép tính và thực hiện các lệnh. Phần chính của CPU đảm nhiệm chức năng này là đơn vị logic toán học (ALU – *Arthimetic Logic Unit*). Ngoài ra để hỗ trợ cho hoạt động của ALU còn có thêm một số các thành phần khác như bộ giải mã (*decoder*), bộ tuần tự (*sequencer*) và các thanh ghi.

Bộ giải mã chuyển đổi (thông dịch) các lệnh lưu trữ ở trong bộ mã chương trình thành các mã mà ALU có thể hiểu được và thực thi. Bộ tuần tự có nhiệm vụ quản lý dòng dữ liệu trao đổi qua bus dữ liệu của VXL. Các thanh ghi được sử dụng để CPU lưu trữ tạm thời các dữ liệu chính cho việc thực thi các lệnh và chúng có thể thay đổi nội dung trong quá trình hoạt động của ALU. Hầu hết các thanh ghi của VXL đều là các bộ nhớ được tham chiếu (*mapped*) và hội nhập với khu vực bộ nhớ và có thể được sử dụng như bất kỳ khu vực nhớ khác.

Các thanh ghi có chức năng lưu trữ trạng thái của CPU. Nếu các nội dung của bộ nhớ VXL và các nội dung của các thanh ghi tại một thời điểm nào đó được lữu giữ đầy đủ thì hoàn toàn có thể tạm dừng thực hiện phần chương trình hiện tại trong một khoảng thời gian bất kỳ và có thể trở lại trạng thái của CPU trước đó. Thực tế số lượng các thanh ghi và tên gọi của chúng cũng khác nhau trong các họ VXL/VĐK và thường do chính các nhà chế tạo qui định, nhưng về cơ bản chúng đều có chung các chức năng như đã nêu.

Khi thứ tự *byte* trong bộ nhớ đã được xác định thì người thiết kế phần cứng phải thực hiện một số quyết định xem CPU sẽ lưu dữ liệu đó như thế nào. Cơ chế này cũng khác nhau tuỳ theo kiến trúc tập lệnh được áp dụng. Có ba loại hình cơ bản:

(1) Kiến trúc ngăn xếp
(2) Kiến trúc bộ tích luỹ
(3) Kiến trúc thanh ghi mục đích chung

Kiến trúc ngăn xếp sử dụng ngăn xếp để thực hiện lệnh và các toán tử nhận được từ đỉnh ngăn xếp. Mặc dù cơ chế này hỗ trợ mật độ mã tốt và mô hình đơn giản cho việc đánh giá cách thể hiện chương trình nhưng ngăn xếp không thể hỗ trợ khả năng truy nhập ngẫu nhiên và hạn chế hiệu suất thực hiện lệnh.

Kiến trúc bộ tích luỹ với lệnh một toán tử ngầm mặc định chứa trong thanh ghi tích luỹ có thể giảm được độ phức tạp bên trong của cấu trúc CPU và cho phép cấu thành lệnh rất nhỏ gọn. Nhưng thanh ghi tích luỹ chỉ là nơi chứa dữ liệu tạm thời nên giao thông bộ nhớ rất lớn.

Kiến trúc thanh ghi mục đích chung sử dụng các tập thanh ghi mục đích chung và được đón nhận như mô hình của các hệ thống CPU mới, hiện đại. Các tập thanh ghi đó nhanh hơn bộ nhớ thường và dễ dàng cho bộ biên dịch xử lý thực thi và có thể được sử dụng một cách hiệu quả. Hơn nữa giá thành phần cứng ngày càng có xu thế giảm đáng kể và tập thanh ghi có thể tăng nhanh. Nếu cơ chế truy nhập bộ nhớ nhanh thì kiến trúc dựa trên ngăn xếp có thể là sự lựa chọn lý tưởng; còn nếu truy nhập bộ nhớ chậm thì kiến trúc thanh ghi sẽ là sự lựa chọn phù hợp nhất.

Một số thanh ghi với chức năng điển hình thường được sử dụng trong các kiến trúc CPU như sau:

- **Thanh ghi con trỏ ngăn xếp (*stack pointer*):**

Thanh ghi này lưu giữ địa chỉ tiếp theo của ngăn xếp. Theo nguyên lý giá trị địa chỉ chứa trong thanh ghi con trỏ ngăn xếp sẽ giảm nếu dữ liệu được lưu thêm vào ngăn xếp và sẽ tăng khi dữ liệu được lấy ra khỏi ngăn xếp.

- **Thanh ghi chỉ số (*index register*)**

Thanh ghi chỉ số được sử dụng để lưu địa chỉ khi *mode* địa chỉ được sử dụng. Nó còn được biết tới với tên gọi là thanh ghi con trỏ hay thanh ghi lựa chọn tệp (*Microchip*).

- **Thanh ghi địa chỉ lệnh /Bộ đếm chương trình (*Program Counter*)**

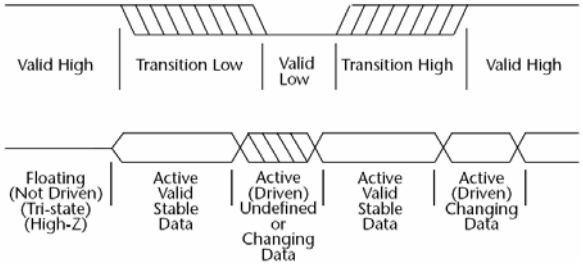
Một trong những thanh ghi quan trọng nhất của CPU là thanh ghi bộ đếm chương trình. Thanh ghi bộ đếm chương trình lưu địa chỉ lệnh tiếp theo của chương trình sẽ được CPU xử lý. Mỗi khi lệnh được trỏ tới và được CPU xử lý thì nội dung giá trị của thanh ghi bộ đếm chương trình sẽ tăng lên một. Chương trình sẽ kết thúc khi thanh ghi PC có giá trị bằng địa chỉ cuối cùng của chương trình nằm trong bộ nhớ chương trình.

- **Thanh ghi tích lũy (*Accumulator*)**

Thanh ghi tích lũy là một thanh ghi giao tiếp trực tiếp với ALU, được sử dụng để lưu giữ các toán tử hoặc kết quả của một phép toán trong quá trình hoạt động của ALU.

2.1.2 Xung nhịp và trạng thái tín hiệu

Trong VXL và các vi mạch số nói chung, hoạt động của hệ thống được thực hiện đồng bộ hoặc dị bộ theo các xung nhịp chuẩn. Các nhịp đó được lấy trực tiếp hoặc gián tiếp từ một nguồn xung chuẩn thường là các mạch tạo xung hoặc dao động thạch anh. Để mô tả hoạt động của hệ thống, các tín hiệu dữ liệu và điều khiển thường được mô tả trạng thái theo giản đồ thời gian và mức tín hiệu như được chỉ ra trong Hình 2-3: Mô tả và trạng thái tín hiệu hoạt động trong VXL



Hình 2-3: Mô tả và trạng thái tín hiệu hoạt động trong VXL

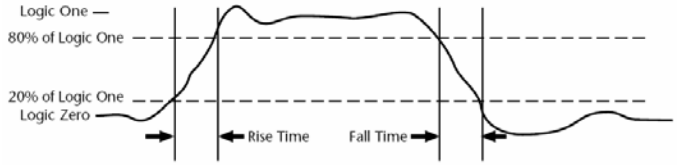
Mục đích của việc mô tả trạng thái tín hiệu theo giản đồ thời gian và mức tín hiệu là để phân tích và xác định chuỗi sự kiện hoạt động chi tiết trong mỗi chu kỳ *bus*. Nhờ việc mô tả này chúng ta có thể xem xét đến khả năng đáp ứng thời gian của các sự kiện thực thi trong hệ thống và thời gian cần thiết để thực thi hoạt động tuần tự cũng như là khả

năng tương thích khi có sự hoạt động phối hợp giữa các thiết bị ghép nối hay mở rộng trong hệ thống. Thông thường thông tin về các nhịp thời gian hoạt động cũng như đặc tính kỹ thuật chi tiết được cung cấp hoặc qui định bởi các nhà chế tạo.

Một số đặc trưng về thời gian của các trạng thái hoạt động cơ bản của các tín hiệu hệ thống gồm có như sau:

- ✓ Thời gian tăng hoặc giảm
- ✓ Thời gian trễ lan truyền tín hiệu
- ✓ Thời gian thiết lập
- ✓ Thời gian giữ
- ✓ Trễ cấm hoạt động và trạng thái treo (*Tri-State*)
- ✓ Độ rộng xung
- ✓ Tần số nhịp xung hoạt động

- **Thời gian tăng hoặc giảm**

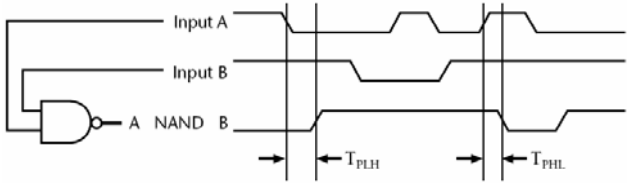


Hình 2-4: Mô tả trạng thái tín hiệu logic tăng và giảm

Thời gian tăng được định nghĩa là khoảng thời gian để tín hiệu tăng từ 20% đến 80% mức tín hiệu cần thiết. Thời gian giảm là khoảng thời gian để tín hiệu giảm từ 80% đến 20% mức tín hiệu cần thiết.

- **Thời gian trễ lan truyền:**

Là khoảng thời gian tín từ khi thay đổi tín hiệu vào cho tới khi có sự thay đổi tín hiệu ở đầu ra. Đặc tính này thường do cấu tạo và khả năng truyền dẫn tín hiệu vật lý trong hệ thống tín hiệu.

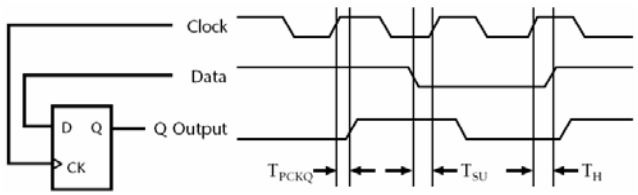


Hình 2-5: Mô tả trạng thái và độ trễ lan truyền tín hiệu

- **Thời gian thiết lập và lưu giữ**

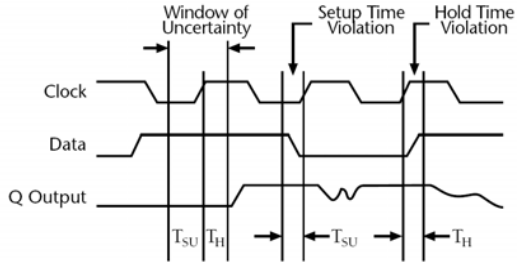
Khoảng thời gian cần thiết để tín hiệu trích mẫu đạt tới một trạng thái ổn định trước khi nhịp xung chuẩn đồng hồ thay đổi được gọi là thời gian thiết lập. Thời gian lưu giữ là

khoảng thời gian cần thiết để duy trì tín hiệu trích mẫu ổn định sau khi xung nhịp chuẩn đồng hồ thay đổi. Thực chất khoảng thời gian thiết lập và thời gian lưu giữ là cần thiết để đảm bảo tín hiệu được ghi nhận chính xác và ổn định trong quá trình hoạt động và chuyển mức trạng thái. Giản đồ thời gian trong Hình 2-6: Thời gian thiết lập và lưu giữ minh họa thời gian thiết lập và lưu giữ trong hoạt động của Triger D.



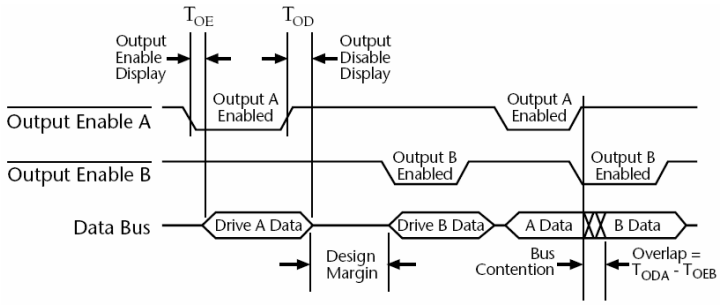
Hình 2-6: Thời gian thiết lập và lưu giữ

Trong trường hợp hoạt động chuyển trạng thái tín hiệu không đồng bộ và không đảm bảo được thời gian thiết lập và lưu giữ sẽ có thể dẫn đến sự mất ổn định hay không xác định mức tín hiệu trong hệ thống. Hiện tượng này được biết tới với tên gọi là *metastabilit*. Để minh họa cho hiện tượng này trong Hình 2-7 mô tả hoạt động lỗi của một *Triger* khi các mức tín hiệu vào không thỏa mãn yêu cầu về thời thiết lập và lưu giữ.



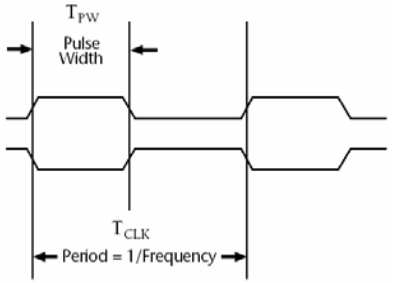
Hình 2-7: Hiện tượng Metastabilit trong hoạt động của Triger D

Chu kỳ tín hiệu 3 trạng thái và *contention*



Hình 2-8: Mô tả chu kỳ tín hiệu 3 trạng thái và *contention*

Độ rộng xung và tần số nhịp xung chuẩn



Hình 2-9: Độ rộng và tần số xung nhịp chuẩn

2.1.3 Bus địa chỉ, dữ liệu và điều khiển

- **Bus địa chỉ**

Bus địa chỉ là các đường dẫn tín hiệu logic một chiều để truyền địa chỉ tham chiếu tới các khu vực bộ nhớ và chỉ ra dữ liệu được lưu giữ ở đâu trong không gian bộ nhớ. Trong qúa trình hoạt động CPU sẽ điều khiển bus địa chỉ để truyền dữ liệu giữa các khu vực bộ nhớ và CPU. Các địa chỉ thông thường tham chiếu tới các khu vực bộ nhớ hoặc các khu vực vào ra, hoặc ngoại vi. Dữ liệu được lưu ở các khu vực đó thường là 8-bit (1 *byte*), 16-bit, hoặc 32-bit tùy thuộc vào cấu trúc từng loại vi xử lý/vi điều khiển. Hầu hết các vi điều khiển thường đánh địa chỉ dữ liệu theo khối 8-bit. Các loại vi xử lý 8-bit, 16-bit và 32-bit nói chung cũng đều có thể làm việc trao đổi với kiểu dữ liệu 8-bit và 16-bit.

Chúng ta vẫn thường được biết tới khái niệm địa chỉ truy nhập trực tiếp, đó là khả năng CPU có thể tham chiếu và truy nhập tới trong một chu kỳ bus. Nếu vi xử lý có N bit địa chỉ tức là nó có thể đánh địa chỉ được 2^N khu vực mà CPU có thể tham chiếu trực tiếp tới. Qui ước các khu vực được đánh địa chỉ bắt đầu từ địa chỉ 0 và tăng dần đến 2^N-1. Hiện nay các vi xử lý và vi điều khiển nói chung chủ yếu vẫn sử dụng phổ biến các bus dữ liệu có độ rộng là 16, 20, 24, hoặc 32-bit. Nếu đánh địa chỉ theo *byte* thì một vi xử lý 16-bit có thể đánh địa chỉ được 2^{16} khu vực bộ nhớ tức là 65,536 byte = 64Kbyte. Tuy nhiên có một số khu vực bộ nhớ mà CPU không thể truy nhập trực tiếp tới tức là phải sử dụng nhiều nhịp bus để truy nhập, thông thường phải kết hợp với việc điều khiển phần mềm. Kỹ thuật này chủ yếu được sử dụng để mở rộng bộ nhớ và thường được biết tới với khái niệm đánh địa chỉ trang nhớ khi nhu cầu đánh địa chỉ khu vực nhớ vượt quá phạm vi có thể đánh địa chỉ truy nhập trực tiếp.

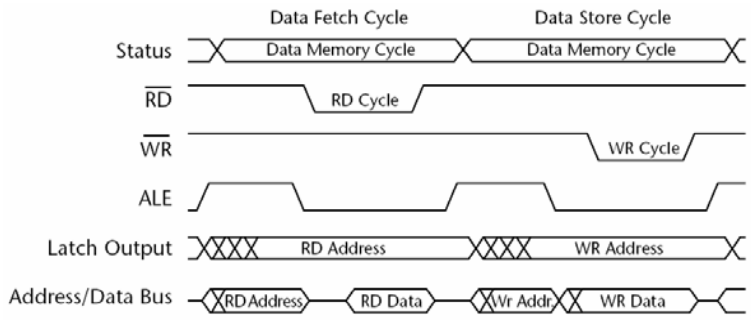
Ví dụ: CPU 80286 có 24-bit địa chỉ sẽ cho phép đánh địa chỉ trực tiếp cho 2^{24} *byte* (16 Mbyte) nhớ. CPU 80386 và các loại vi xử lý mạnh hơn có không gian địa chỉ 32-bit sẽ có thể đánh được tới 2^{32} byte (4*Gbyte*) địa chỉ trực tiếp.

- **Bus dữ liệu**

Bus dữ liệu là các kênh truyền tải thông tin theo hai chiều giữa CPU và bộ nhớ hoặc các thiết bị ngoại vi vào ra. Bus dữ liệu được điều khiển bởi CPU để đọc hoặc viết các dữ liệu hoặc mã lệnh thực thi trong qúa trình hoạt động của CPU. Độ rộng của bus dữ liệu nói chung sẽ xác định được lượng dữ liệu có thể truyền và trao đổi trên bus. Tốc độ truyền hay trao đổi dữ liệu thường được tính theo đơn vị là [byte/s]. Số lượng đường bit dữ liệu sẽ cho phép xác định được số lượng bit có thể lưu trữ trong mỗi khu vực tham chiếu trực tiếp. Nếu một bus dữ liệu có khả năng thực hiện một lần truyền trong 1 µs, thì bus dữ liệu 8-bit sẽ có băng thông là 1Mbyte/s, bus 16-bit sẽ có băng thông là 2Mbyte/s và bus 32-bit sẽ có băng thông là 4Mbyte/s. Trong trường hợp bus dữ liệu 8-bit với chu kỳ bus là T=1µs (tức là sẽ truyền được 1byte/1chu kỳ) thì sẽ truyền được 1 Mbyte trong 1s hay 2Mbyte trong 2s.

- **Bus điều khiển**

Bus điều khiển phục vụ truyền tải các thông tin dữ liệu để điều khiển hoạt động của hệ thống. Thông thường các dữ liệu điều khiển bao gồm các tín hiệu chu kỳ để đồng bộ các nhịp chuyển động và hoạt động của hệ thống. Bus điều khiển thường được điều khiển bởi CPU để đồng bộ hóa nhịp hoạt động và dữ liệu trao đổi trên các bus. Trong trường hợp vi xử lý sử dụng dồn kênh bus dữ liệu và bus địa chỉ tức là một phần hoặc toàn bộ bus dữ liệu sẽ được sử dụng chung chia sẻ với bus địa chỉ thì cần một tín hiệu điều khiển để phân nhịp truy nhập cho phép chốt lưu trữ thông tin địa chỉ mỗi khi bắt đầu một chu kỳ truyền. Một ví dụ về các chu kỳ bus và sự đồng bộ của chúng trong hoạt động của hệ thống bus địa chỉ và dữ liệu dồn kênh được chỉ ra trong Hình 2-10. Đây là hoạt động điển hình trong họ vi điều khiển 8051 và nhiều loại tương tự.



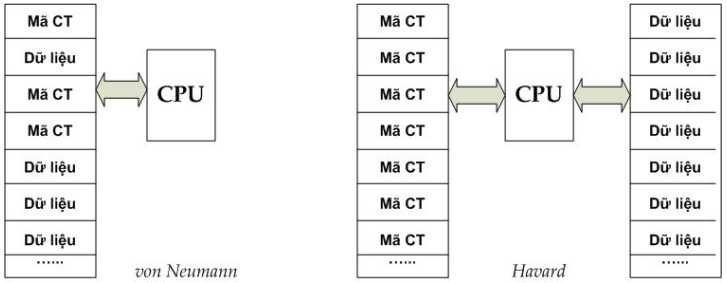
Hình 2-10: Chu kỳ hoạt động bus dồn kênh

2.1.4 Bộ nhớ

Kiến trúc bộ nhớ

Kiến trúc bộ nhớ được chia ra làm hai loại chính và được áp dụng rộng rãi trong hầu hết các Chip xử lý nhúng hiện nay là kiến trúc bộ nhớ *von Neumann* và *Havard*.

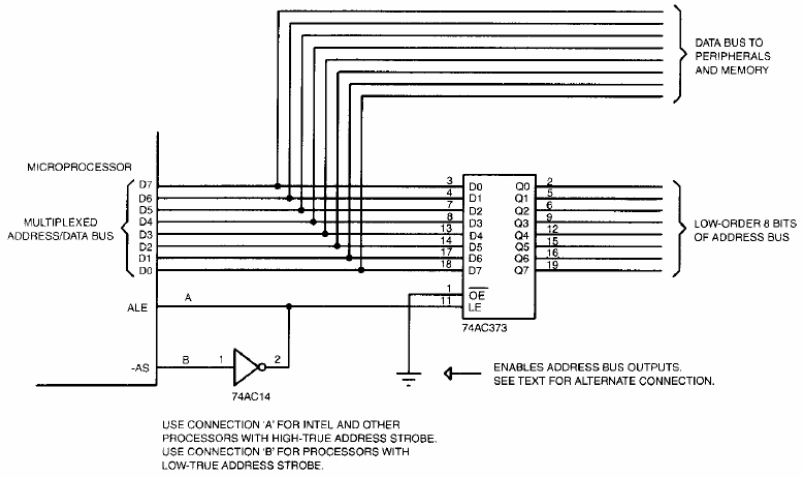
Trong kiến trúc *von Neumann* không phân biệt vùng chứa dữ liệu và mã chương trình. Cả chương trình và dữ liệu đều được truy nhập theo cùng một đường. Điều này cho phép đưa dữ liệu vào vùng mã chương trình ROM, và cũng có thể lưu mã chương trình vào vùng dữ liệu RAM và thực hiện từ đó.



Hình 2-11: Kiến trúc bộ nhớ *von Neumann* và *Havard*

Kiến trúc *Havard* tách/phân biệt vùng lưu mã chương trình và dữ liệu. Mã chương trình chỉ có thể được lưu và thực hiện trong vùng chứa ROM và dữ liệu cũng chỉ có thể lưu và trao đổi trong vùng RAM. Hầu hết các vi xử lý nhúng ngày nay sử dụng kiến trúc bộ nhớ *Havard* hoặc kiến trúc *Havard* mở rộng (tức là bộ nhớ chương trình và dữ liệu tách biệt nhưng vẫn cho phép khả năng hạn chế để lấy dữ liệu ra từ vùng mã chương trình). Trong kiến trúc bộ nhớ *Havard* mở rộng thường sử dụng một số lượng nhỏ các con trỏ để lấy dữ liệu từ vùng mã chương trình theo cách nhúng vào trong các lệnh tức thời. Một số *Chip* vi điều khiển nhúng tiêu biểu hiện nay sử dụng cấu trúc *Havard* là 8031, *PIC*, *Atmel* AVR90S. Nếu sử dụng *Chip* 8031 chúng ta sẽ nhận thấy điều này thông qua việc truy nhập lấy dữ liệu ra từ vùng dữ liệu RAM hoặc từ vùng mã chương trình. Chúng ta có một vài con trỏ được sử dụng để lấy dữ liệu ra từ bộ nhớ dữ liệu RAM, nhưng chỉ có duy nhất một con trỏ DPTR có thể được sử dụng để lấy dữ liệu ra từ vùng mã chương trình. Hình 2-11 mô tả nguyên lý kiến trúc của bộ nhớ *von Neumann* và *Harvard*.

Ưu điểm nổi bật của cấu trúc bộ nhớ *Harvard* so với kiến trúc *von Neumann* là có hai kênh tách biệt để truy nhập vào vùng bộ nhớ mã chương trình và dữ liệu nhờ vậy mà mã chương trình và dữ liệu có thể được truy nhập đồng thời và làm tăng tốc độ luồng trao đổi với bộ xử lý.



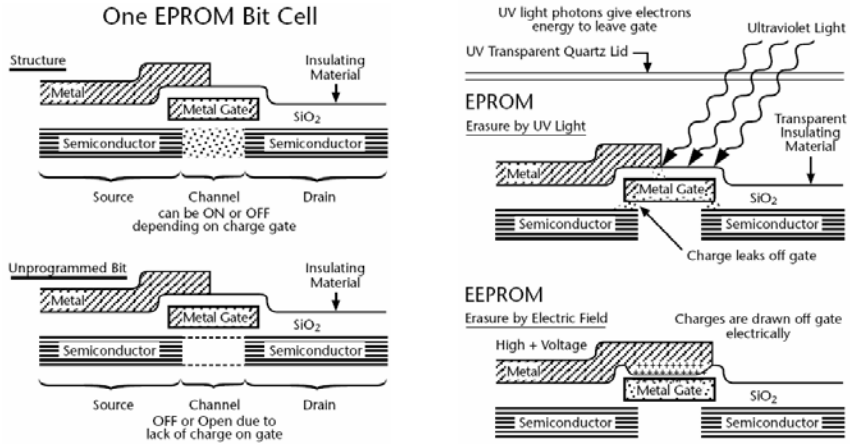
Hình 2-12: Nguyên lý điều khiển tách kênh truy nhập bus địa chỉ và bus dữ liệu

Bộ nhớ chương trình – PROM (*Programmable Read Only Memory*)

Vùng để lưu mã chương trình. Có ba loại bộ nhớ PROM thông dụng được sử dụng cho hệ nhúng và sẽ được giới thiệu lần lượt sau đây.

- **EPROM**

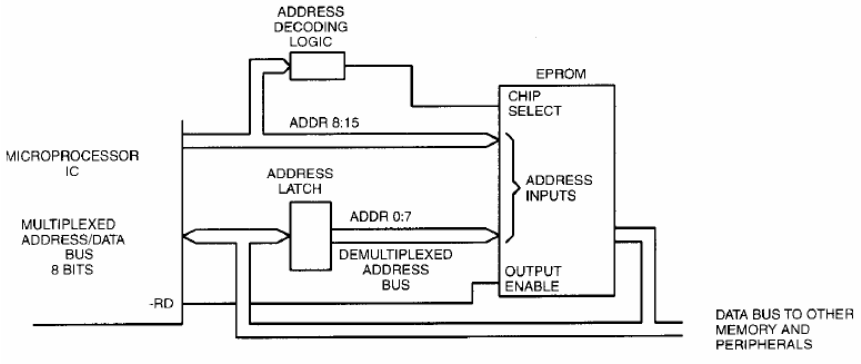
Bao gồm một mảng các *transistor* khả trình. Mã chương trình sẽ được ghi trực tiếp và vi xử lý có thể đọc ra để thực hiện. EPROM có thể xoá được bằng tia cực tím và có thể được lập trình lại. Cấu trúc vật lý của EPROM được mô tả như trong Hình 2-13.



Hình 2-13: Nguyên lý cấu tạo và hoạt động xoá của EPROM

- **Bộ nhớ *Flash***

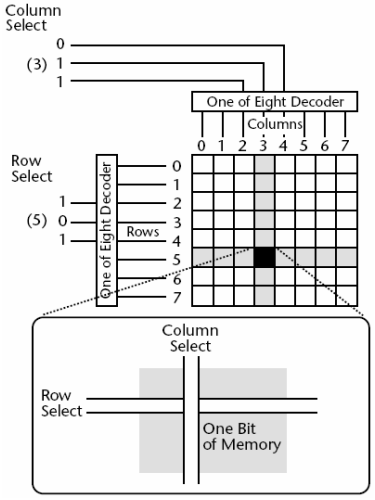
Cũng giống như EPROM được cấu tạo bởi một mảng *transistor* khả trình nhưng có thể xoá được bằng điện và chính vì vậy có thể nạp lại chương trình mà không cần tách ra khỏi nền phần cứng VXL. Ưu điểm của bộ nhớ *flash* là có thể lập trình trực tiếp trên mạch cứng mà nó đang thực thi trên đó.



Hình 2-14: Sơ đồ nguyên lý ghép nối EPROM với VXL

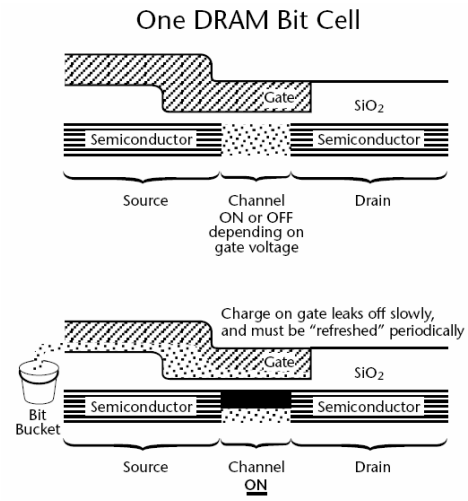
Bộ nhớ dữ liệu - RAM

Vùng để lưu hoặc trao đổi dữ liệu trung gian trong quá trình thực hiện chương trình.

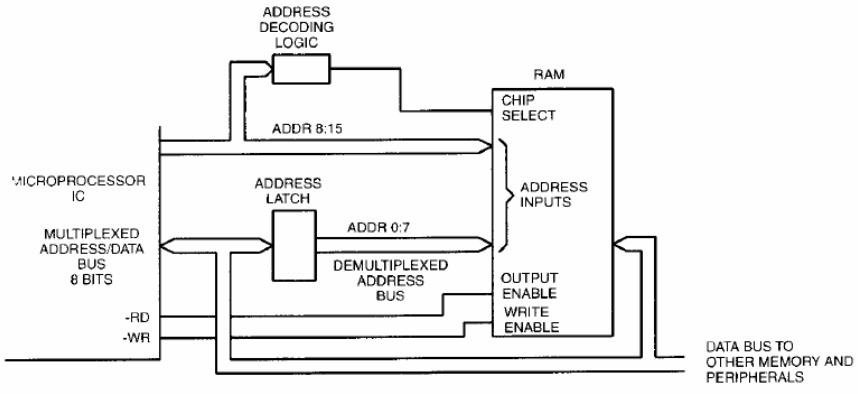


Hình 2-15: Cấu trúc nguyên lý bộ nhớ RAM

Có hai loại SRAM và DRAM



Hình 2-16: Cấu trúc một phần tử nhớ DRAM



Hình 2-17: Nguyên lý ghép nối (mở rộng) RAM với VXL

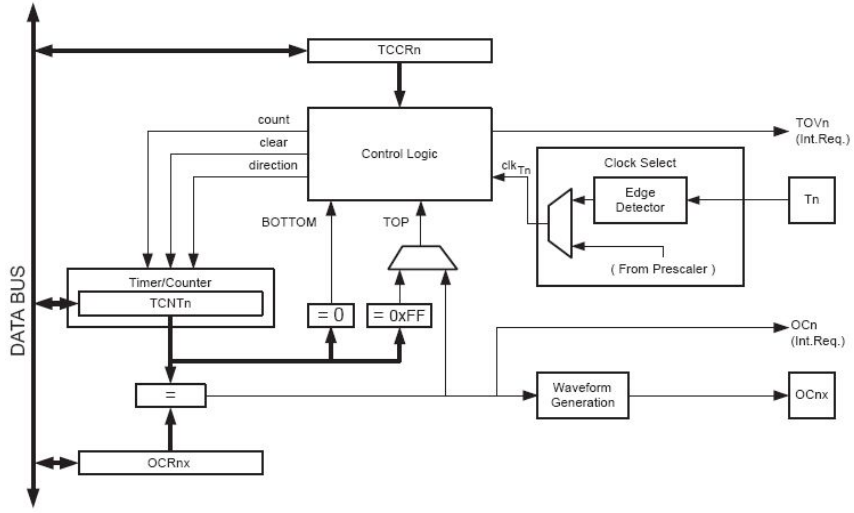
2.1.5 Không gian và phân vùng địa chỉ

2.1.6 Ngoại vi

Bộ định thời gian/Bộ đếm

Hầu hết các chip vi điều khiển ngày nay đều có ít nhất một bộ định thời gian/bộ đếm có thể cấu hình hoạt động linh hoạt theo các *mode* phục vụ nhiều mục đích trong các ứng dụng xử lý, điều khiển. Các bộ định thời gian cho phép tạo ra các chuỗi xung và ngắt thời gian hoặc đếm theo các khoảng thời gian có thể lập trình. Chúng thường được ứng

dụng phổ biến trong các nhiệm vụ đếm xung, đo khoảng thời gian các sự kiện, hoặc định chu kỳ thời gian thực thi các tác vụ. Một trong những ứng dụng quan trọng của bộ định thời gian là tạo nhịp từ bộ tạo xung thạch anh cho bộ truyền thông dị bộ đa năng hoạt động. Thực chất đó là ứng dụng để thực hiện phép chia tần số. Để đạt được độ chính xác, tần số thạch anh thường được chọn sao cho các phép chia số nguyên được thực hiện chính xác đảm bảo cho tốc độ truyền thông dữ liệu được tạo ra chính xác. Chính vì vậy họ vi điều khiển 80C51 thường hay sử dụng thạch anh có tần số dao động là 11.059 thay vì 12MHz để tạo ra nhịp hoạt động truyền thông tốc độ chuẩn 9600.



Hình 2-18: Bộ định thời/ bộ đếm 8 bit của AVR

Bộ điều khiển ngắt

Ngắt là một sự kiện xảy ra làm dừng hoạt động chương trình hiện tại để phục vụ thực thi một tác vụ hay một chương trình khác. Cơ chế ngắt giúp CPU làm tăng tốc độ đáp ứng phục vụ các sự kiện trong chương trình hoạt động của VXL/VĐK. Các VĐK khác nhau sẽ định nghĩa các nguồn tạo ngắt khác nhau nhưng đều có chung một cơ chế hoạt động ví dụ như ngắt truyền thông nối tiếp, ngắt bộ định thời gian, ngắt cứng, ngắt ngoài...Khi một sự kiện yêu cầu ngắt xuất hiện, nếu được chấp nhận CPU sẽ lưu cất trạng thái hoạt động cho chương trình hiện tại đang thực hiện ví dụ như nội dung bộ đếm chương trình (con trỏ lệnh) các nội dung thanh ghi lưu dữ liệu điều khiển chương trình nói chung để thực thi chương trình phục vụ tác vụ cho sự kiện ngắt. Thực chất quá trình ngắt là CPU nhận dạng tín hiệu ngắt, nếu chấp nhận sẽ đưa con trỏ lệnh chương trình trỏ tới vùng mã chứa chương trình phục vụ tác vụ ngắt. Vì vậy mỗi một ngắt đều gắn với một vector ngắt như một con trỏ lưu thông tin địa chỉ của vùng bộ nhớ chứa mã chương trình phục vụ tác vụ của ngắt. CPU sẽ thực hiện chương trình

phục vụ tác vụ ngắt đến khi nào gặp lệnh quay trở về chương trình trước thời điểm sự kiện ngắt xảy ra. Có thể phân ra 2 loại nguồn ngắt: *Ngắt cứng* và *Ngắt mềm*.

- **Ngắt mềm**

Ngắt mềm thực chất thực hiện một lời gọi hàm đặc biệt mà được kích hoạt bởi các nguồn ngắt là các sự kiện xuất hiện từ bên trong chương trình và ngoại vi tích hợp trên Chip ví dụ như ngắt thời gian, ngắt chuyển đổi A/D, … Cơ chế ngắt này còn được hiểu là loại thực hiện đồng bộ với chương trình vì nó được kích hoạt và thực thi tại các thời điểm xác định trong chương trình. Hàm được gọi sẽ thực thi chức năng tương ứng với yêu cầu ngắt. Các hàm đó thường được trỏ bởi một vector ngắt mà đã được định nghĩa và gán cố định bởi nhà sản xuất Chip. Ví dụ như hệ điều hành của PC sử dụng ngắt số 21_{hex} để gán cho ngắt truy nhập đọc dữ liệu từ đĩa cứng và xuất dữ liệu ra máy in.

- **Ngắt cứng**

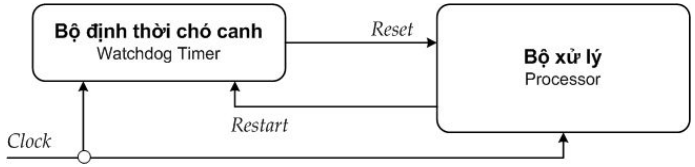
Ngắt cứng có thể được xem như là một lời gọi hàm đặc biệt trong đó nguồn kích hoạt là một sự kiện đến từ bên ngoài chương trình thông qua một cấu trúc phần cứng (thường được kết nối với thế giới bên ngoài qua các chân ngắt). Ngắt cứng thường được hiểu hoạt động theo cơ chế dị bộ vì các sự kiện ngắt kích hoạt từ các tín hiệu ngoại vi bên ngoài và tương đối độc lập với CPU, thường là không xác định được thời điểm kích hoạt. Khi các ngắt cứng được kích hoạt CPU sẽ nhận dạng và thực hiện lời gọi hàm thực thi chức năng phục vụ sự kiện ngắt tương ứng.

Trong các cơ chế ngắt khoảng thời gian từ khi xuất hiện sự kiện ngắt (có yêu cầu phục vụ ngắt) tới khi dịch vụ ngắt được thực thi là xác định và tuỳ thuộc vào công nghệ phần cứng xử lý của Chip.

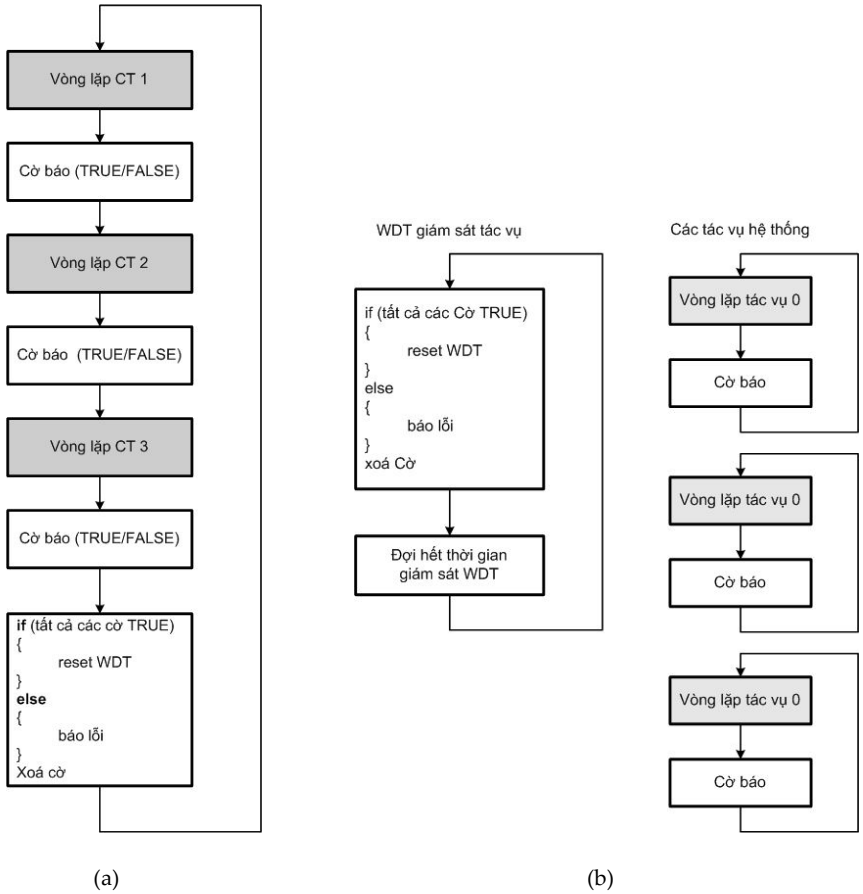
Bộ định thời chó canh – *Watchdog Timer*

Thông thường khi có một sự cố xảy ra làm hệ thống bị treo hoặc chạy quẩn, CPU sẽ không thể tiếp tục thực hiện đúng chức năng. Đặc biệt khi hệ thống phải làm việc ở chế độ vận hành tự động và không có sự can thiệp trực tiếp thường xuyên bởi người vận hành. Để thực hiện cơ chế tự giám sát và phát hiện sự cố phần mềm, một số VXL/VĐK có thêm một bộ định thời chó canh. Bản chất đó là một bộ định thời đặc biệt để định nghĩa một khung thời gian hoạt động bình thường của hệ thống. Nếu có sự cố phần mềm xảy ra sẽ làm hệ thống bị treo khi đó bộ định thời chó canh sẽ phát hiện và giúp hệ thống thoát khỏi trạng thái đó bằng cách thực hiện khởi tạo lại chương trình. Chương trình hoạt động khi có bộ định thời phải đảm bảo *reset* nó trước khi khung thời gian bị vi phạm. Khung thời gian này được định nghĩa phụ thuộc vào sự đánh giá của người thực hiện phần mềm, thiết lập khoảng thời gian đảm bảo chắc chắn hệ thống thực hiện bình thường không có sự cố phần mềm.

Có một số cơ chế thực hiện cài đặt bộ định thời cho canh để giám sát hoạt động của hệ thống như sau:



Hình 2-19: Sơ đồ nguyên lý hoạt động của bộ định thời chó canh



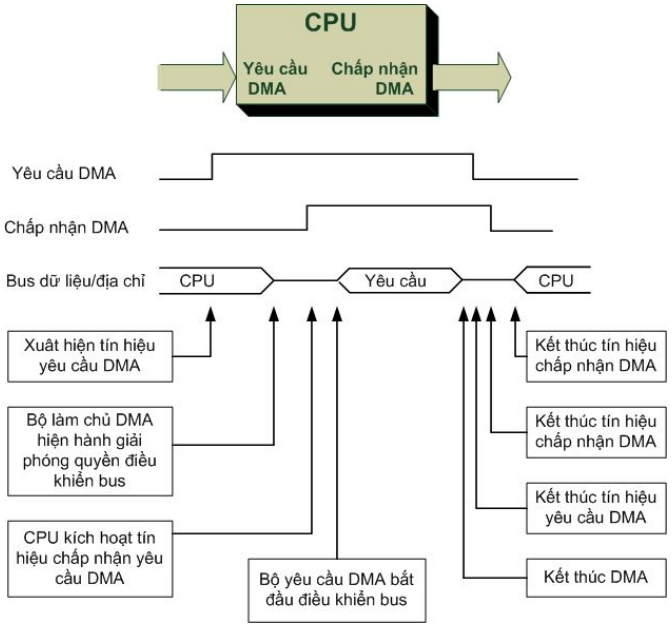
Hình 2-20: Nguyên lý hoạt động bộ định thời chó canh

Bộ điều khiển truy nhập bộ nhớ trực tiếp – DMA

DMA (*Direct Memory Access*) là cơ chế hoạt động cho phép hai hay nhiều vi xử lý hoặc ngoại vi chia sẻ bus chung. Thiết bị nào đang có quyền điều khiển bus sẽ có thể toàn

quyền truy nhập và trao đổi dữ liệu trực tiếp với các bộ nhớ như hệ thống có một vi xử lý. Ứng dụng phổ biến nhất của DMA là chia sẻ bộ nhớ chung giữa hai bộ vi xử lý hoặc các ngoại vi để truyền dữ liệu trực tiếp giữa thiết bị ngoại vi vào/ra và bộ nhớ dữ liệu của VXL.

Truy nhập bộ nhớ trực tiếp được sử dụng để đáp ứng nhu cầu trao đổi dữ liệu vào ra tốc độ cao giữa ngoại vi với bộ nhớ. Thông thường các ngoại vi kết nối với hệ thống phải chia sẻ bus dữ liệu và được điều khiển bởi CPU trong quá trình trao đổi dữ liệu. Điều này làm hạn chế tốc độ trao đổi, để tăng cường tốc độ và loại bỏ sự can thiệp của CPU, đặc biệt trong trường hợp cần truyền một lượng dữ liệu lớn. Cơ chế hoạt động DMA được mô tả như trong Hình 2-21. Thủ tục được bắt đầu bằng việc yêu cầu thực hiện DMA với CPU. Sau khi xử lý, nếu được chấp nhận CPU sẽ trao quyền điều khiển bus cho ngoại vi và thực hiện quá trình trao đổi dữ liệu. Sau khi thực hiện xong CPU sẽ nhận được thông báo và nhận lại quyền điều khiển bus. Trong cơ chế DMA, có hai cách để truyền dữ liệu: kiểu DMA chu kỳ đơn, và kiểu DMA chu kỳ nhóm (*burst*).



Hình 2-21: Nhịp hoạt động DMA

- **DMA chu kỳ đơn và nhóm**

Trong kiểu hoạt động DMA chu kỳ nhóm, ngoại vi sẽ nhận được quyền điều khiển và truyền khối dữ liệu rồi trả lại quyền điều khiển cho CPU. Trong cơ chế DMA chu kỳ đơn ngoại vi sau khi nhân được quyền điều khiển bus chỉ truyền một từ dữ liệu rồi trả lại ngay quyền kiểm soát bộ nhớ và bus dữ liệu cho CPU. Trong cơ chế thực hiện DMA

cần có một bước xử lý để quyết định xem thiết bị nào sẽ đươc nhận quyền điều khiển trong trường hợp có nhiều hơn một thiết bị có nhu cầu sử dụng DMA. Thông thường kiểu DMA chu kỳ nhóm cần ít dữ liệu thông tin điều khiển (*overhead*) nên có khả năng trao đổi với tốc độ cao nhưng lại chiếm nhiều thời gian truy nhập bus do truyền cả khối dữ liệu lớn. Điều này có thể ảnh hưởng đến hoạt động của cả hệ thống do trong suốt quá trình thực hiện DMA nhóm, CPU sẽ bị khoá quyền truy nhập bộ nhớ và không thể xử lý các nhiệm vụ khác của hệ thống mà có nhu cầu bộ nhớ, ví dụ như các dịch vụ ngắt, hoặc các tác vụ thời gian thực...

- **Chu kỳ rỗi (*Cycle Stealing*)**

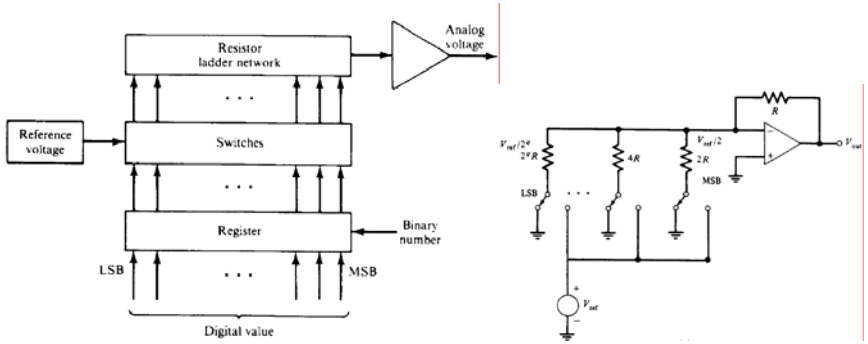
Trong kiểu này DMA sẽ được thực hiện trong những thời điểm chu kỳ bus mà CPU không sử dụng bus do đó không cần thực hiện thủ tục xử lý cấp phát quyền truy nhập và thực hiện DMA.

Hầu hết các vi xử lý hiện đại đều sử dụng gần như 100% dung lượng bộ nhớ và băng thông của bus nên sẽ không có nhiều thời gian dành cho DMA thực hiện. Để tiết kiệm và tối ưu tài nguyên thì cần có một trọng tài phân xử và dữ liệu sẽ được truyền đi xếp chồng theo thời gian. Nói chung kiểu DMA dạng *burst* hiệu quả nhất khi khoảng thời gian cần thực hiện DMA tương đối nhỏ. Trong khoảng thời gian thực hiện DMA, toàn bộ băng thông của bus sẽ được sử dụng tối đa và toàn bộ khối dữ liệu sẽ được truyền đi trong một khoảng thời gian rất ngắn. Nhưng nhược điểm của nó là nếu dữ liệu cần truyền lớn và cần một khoảng thời gian dài thì sẽ dẫn đến việc *block* CPU và có thể bỏ qua việc xử lý các sự kiện và tác vụ khác. Đối với DMA chu kỳ đơn thì yêu cầu truy nhập bộ nhớ, truyền một từ dữ liệu và giải phóng bus. Cơ chế này cho phép thực hiện truyền *interleave* và được biết tới với tên gọi *inteleaved* DMA. Kiểu truyền DMA chu kỳ đơn phù hợp để truyền dữ liệu trong một khoảng thời gian dài mà có đủ thời gian để yêu cầu truy nhập và giải phóng bus cho mỗi lần truyền một từ dữ liệu. Chính vì vậy sẽ giảm băng thông truy nhập bus do phải mất nhiều thời gian để yêu cầu truy nhập và giải phóng bus. Trong trường hợp này CPU và các thiết bị khác vẫn có thể chia sẻ và truyền dữ liệu nhưng trong một dải băng thông hẹp. Trong nhiều hệ thống bus thực hiện cơ chế xử lý và giải quyết yêu cầu truy nhập (trọng tài) thông qua dữ liệu truyền vì vậy cũng không ảnh hưởng nhiều đến tốc độ truyền DMA.

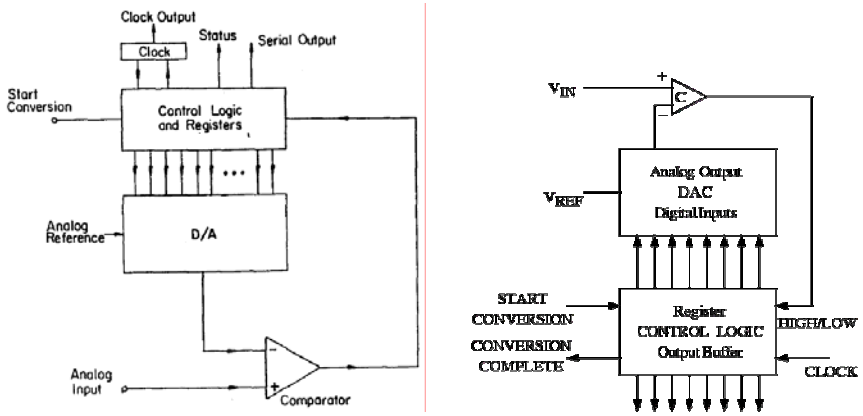
DMA được yêu cầu khi khả năng điều khiển của CPU để truyền dữ liệu thực hiện quá chậm. DMA cũng thực sự có ý nghĩa khi CPU đang phải thực hiện các tác vụ khác mà không cần nhu cầu truy nhập bus.

IC chức năng chuyên dụng

DAC/ADC



Hình 2-22: Sơ đồ nguyên lý mạch chuyển đổi DAC

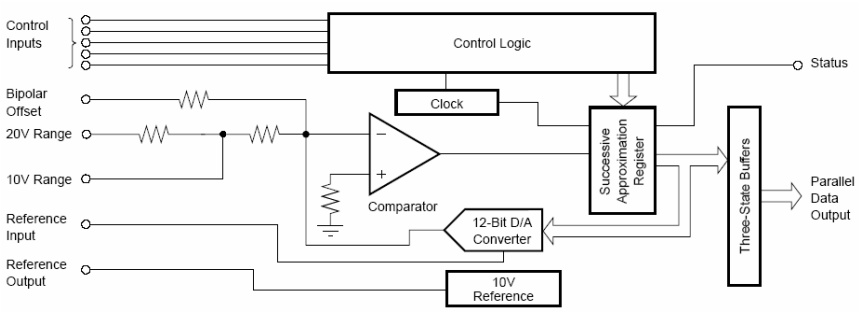


Hình 2-23: Sơ đồ nguyên lý mạch chuyển đổi ADC

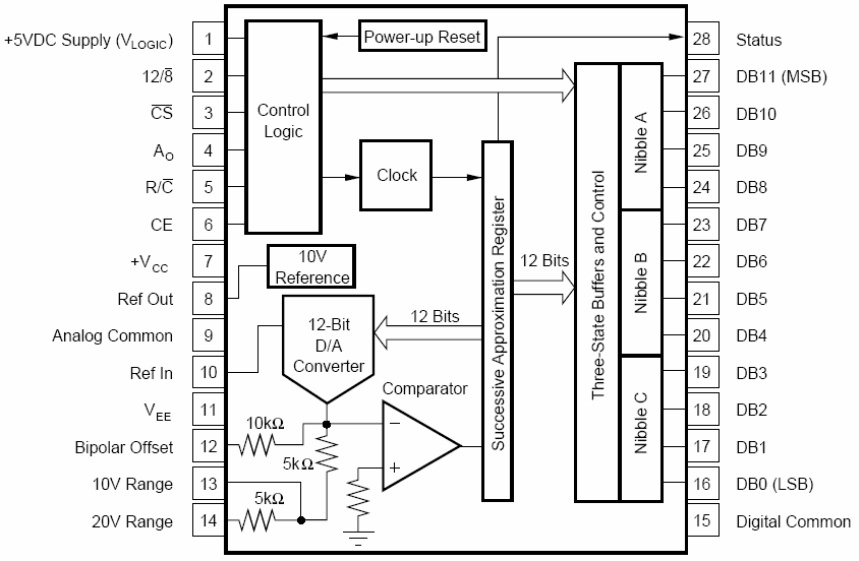
Ví dụ ADC 754A

Đặc điểm kỹ thuật:

- ✓ Chế tạo theo công nghệ CMOS.
- ✓ 12-bit với giao diện tương thích với các loại VXL/VĐK 8, 12 và 16-bit. Có thể lập trình để hoạt động chuyển đổi 8 bit hoặc 12 bit.
- ✓ Tín hiệu dữ liệu ra tương thích với chuẩn TTL và ghép nối thông qua loại cổng logic 3 trạng thái.
- ✓ Dải giá trị điện áp đầu vào có thể lựa chọn nhờ cấu hình giá trị điện trở nội đầu vào để nhận các dải tín hiệu (0÷10)V, (0÷20)V, (-5÷+5)V, và (-10÷+10)V.
- ✓ Có thêm khả năng cung cấp nguồn tham chiếu nội Vref = +10V.
- ✓ Nguồn cung cấp có thể là +5V, ± 12V, hoặc ± 15V
- ✓ Thời gian chuyển đổi cực đại là 25 µs với thời gian truy nhập bus là 150ns.



Hình 2-24: Sơ đồ nguyên lý cấu trúc ADC1754A



Hình 2-25: Sơ đồ bố trí chân của Chip ADC574A

Nguyên lý điều khiển

ADC 574 được điều khiển bởi các chân tín hiệu như mô tả trong bảng sau:

Bảng 1: Tín hiệu điều khiển ADC 574A

Ký hiệu	Định nghĩa	Chức năng
CE (Pin 6)	Chip Enable (active high)	Must be high ("1") to either initiate a conversion or read output data. 0-1 edge may be used to initiate a conversion.
$\overline{CS}$ (Pin 3)	Chip Select (active low)	Must be low ("0") to either initiate a conversion or read output data. 1-0 edge may be used to initiate a conversion.

$R/\overline{C}$ (Pin 5)	Read/Convert (“1” = read) (“0” = convert)	Must be low (“0”) to initiate either 8- or 12-bit conversions. 1-0 edge may be used to initiate a conversion. Must be high (“1”) to read output data. 0-1 edge may be used to initiate a read operation.
AO (Pin 4)	Byte Address Short Cycle	In the start-convert mode, AO selects 8-bit (AO= “1”) or 12-bit (AO= “0”) conversion mode. When reading output data in two 8-bit bytes, AO= “0” accesses 8 MSBs (high byte) and AO= “1” accesses 4 LSBs and trailing “0s” (low byte).
$12/\overline{8}$ (Pin 2)	Data Mode Select (“1” = 12 bits) (“0” = 8 bits)	When reading output data, 12/8 = “1” enables all 12 output bits simultaneously. 12/8 = “0” will enable the MSBs or LSBs as determined by the AOline.

(1) Thiết lập chế độ hoạt động: Mode chuyển đổi 8-bit hay 12-bit được thiết lập bởi tín hiệu A0. Tín hiệu này phải được chốt trước khi nhận được tín hiệu lệnh bắt đầu thực hiện chuyển đổi.

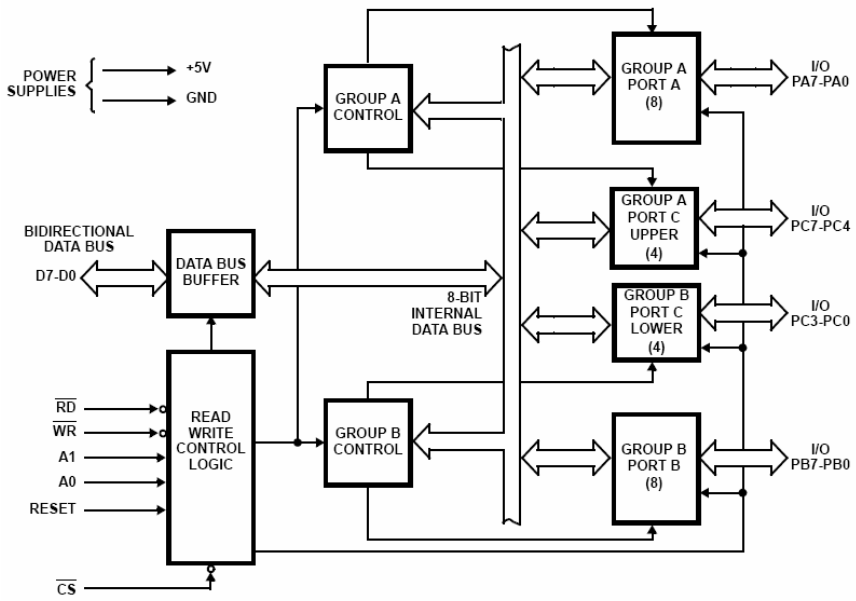
(2) Kích hoạt quá trình chuyển đổi: Bộ chuyển đổi thực hiện chuyển đổi khi nhận được tín hiệu mệnh lệnh tích cực từ chân tín hiệu hoặc CE/CS, hoặc R/C với điều kiện các tín hiệu điều khiển khác đã được xác lập.

(3) Trạng thái chuyển đổi: Tín hiệu đầu ra STATUS báo trạng thái chuyển đổi hiện hành của ADC; thiết lập ở mức cao nếu đang thực hiện chuyển đổi và ở mức thấp nếu đã hoàn thành. Trong quá trình chuyển đổi các tín hiệu điều khiển bị khoá và dữ liệu không thể được đọc vì các đường tín hiệu ra được chuyển sang trạng thái cao trở.

(4) Đọc dữ liệu ra: Quá trình đọc dữ liệu ra có thể được thực thi nếu các tín hiệu điều khiển xác lập ở trạng thái cho phép đọc và tín hiệu STATUS ở trạng thái thấp. Tuỳ thuộc vào mode chuyển đổi được thiết lập và định dạng dữ liệu đọc ra bởi tổ hợp trạng $12/\overline{8}$ và A0.

Cổng song song khả trình 82C55A

82C55A là một giao diện ngoại vi cổng song song khả trình được chế tạo theo công nghệ CMOS. Nó là một thiết bị ngoại vi vào ra khả trình đa mục đích và có thể được sử dụng với nhiều loại VXL/VĐK khác nhau. 82C55A có 24 chân vào ra on Chip được chia ra thành 2 nhóm, mỗi nhóm 12 chân và có thể được sử dụng theo 3 chế độ hoạt động khác nhau. Hình 2-26 mô tả giản đồ khối chức năng của chip 82C55A.



Hình 2-26: Giản đồ khối chức năng của 82C55A

Chức năng và ý nghĩa của các chân on chip của 82C55A được mô tả trong Bảng 2: Chức năng các chân on chip của 82C55A.

Bảng 2: Chức năng các chân on chip của 82C55A

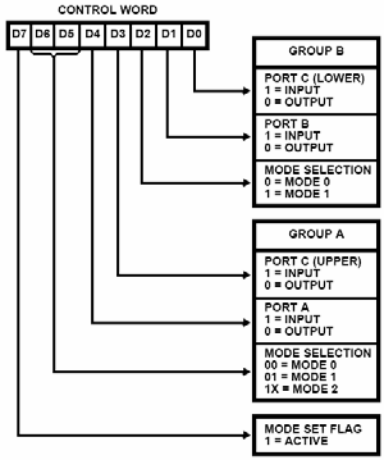
Ký hiệu	Kiểu	Mô tả chức năng
VCC		VCC: The +5V power supply pin. A 0.1μF capacitor between VCC and GND is recommended for decoupling.
GND		GROUND
D0-D7	I/O	DATA BUS: The Data Bus lines are bidirectional three-state pins connected to the system data bus.
RESET	I	RESET: A high on this input clears the control register and all ports (A, B, C) are set to the input mode with the “Bus Hold” circuitry turned on
CS	I	CHIP SELECT: Chip select is an active low input used to enable the 82C55A onto the Data Bus for CPU communications.
RD	I	READ: Read is an active low input control signal used by the CPU to read status information or data via the data bus.
WR	I	WRITE: Write is an active low input control signal used by the CPU to load control words and data into the 82C55A.
A0-A1	I	ADDRESS: These input signals, in conjunction with the RD and WR inputs, control the selection of one of the three ports or the control word register. A0 and A1 are normally connected to the least significant bits of the Address Bus A0, A1.
PA0-PA7	I/O	PORT A: 8-bit input and output port. Both bus hold high and bus hold low circuitry are present on this port.
PB0-PB7	I/O	PORT B: 8-bit input and output port. Bus hold high circuitry is present on this port.

PC0-PC7	I/O	PORT C: 8-bit input and output port. Bus hold circuitry is present on this port.

82C55A cung cấp 3 chế độ hoạt động chính và có thể lập trình để lựa chọn

- Mode 0: Hoạt động vào ra cơ bản
- Mode 1: Hoạt động vào ra nắm bắt (*strobed*)
- Mode 2: Hoạt động Bus 2 chiều

Việc lựa chọn chế độ hoạt động được thực hiện thông qua thanh ghi từ điều khiển và được mô tả như trong Hình 2-27.



Hình 2-27: Thanh ghi từ điều khiển chọn chế độ hoạt động cho 82C55A

Khi đầu vào RESET được điều khiển ở mức cao thì tất cả các cổng sẽ được thiết lập hoạt động ở chế độ cổng vào với 24 đường tín hiệu vào duy trì ở mức logic 1. Sau khi tín hiệu điều khiển RESET ở mức tích cực bị loại bỏ thì 82C55A có thể duy trì chế độ hoạt động mà không cần thêm bất kỳ việc khởi tạo nào nữa. Điều này sẽ giúp loại bỏ được các điện trở treo cao hoặc treo thấp trong các thiết kế cho mạch CMOS. Khi kích hoạt chế độ thiết lập thì thanh ghi từ điều khiển sẽ chứa giá trị 9Bh. Trong qúa trình thực hiện chương trình vẫn có thể thay đổi lựa chọn chế độ hoạt động khác nhau, điều này cho phép 82C55 hoạt động một cách đa dạng đáp ứng cho nhiều bài toán ứng dụng khác nhau. Trong qúa trình thanh ghi từ điều khiển đang được viết thì tất cả các cổng được thiết lập hoạt động ở chế độ cổng ra sẽ được khởi tạo bằng zero.

Mode 0 (Vào ra cơ bản): Cấu hình chế độ hoạt động này cung cấp các hoạt động vào ra đơn giản cho cả 3 cổng A, B và C. Dữ liệu được trao đổi trực tiếp và không cần phải có cơ chế bắt tay. Chế độ hoạt động này hỗ trợ các chức năng cụ thể như sau:

- ✓ Hai cổng 8-bit và 2 cổng 4-bit
- ✓ Bất kỳ cổng nào cũng có thể là cổng vào hoặc cổng ra
- ✓ Các đường dữ liệu tín hiệu ra được chốt

- ✓ Các đường tín hiệu vào không được chốt
- ✓ Có thể cấu hình 16 kiểu hoạt động vào ra khác nhau

Mode 1 (Vào ra có bắt tay): Chế độ hoạt động này cung cấp khả năng truyền dữ liệu tới hoặc đi từ một cổng cụ thể cùng với các tín hiệu bắt tay. Trong chế độ này cổng A, B được sử dụng để truyền dữ liệu và cổng C hoạt động như cổng điều khiển cơ chế động bộ bắt tay. Chế độ hoạt động này cung cấp các chức năng chính sau:

- ✓ Hai nhóm cổng (Nhóm A và Nhóm B). Mỗi nhóm bao gồm 1 cổng 8-bit và một cổng dữ liệu điều khiển 4-bit.
- ✓ Cổng dữ liệu 8-bit có thể hoạt động như hoặc là cổng vào hoặc là cổng ra và cả hai chiều dữ liệu đều được chốt.
- ✓ The 4-bit port is used for control and status of the 8-bit port.

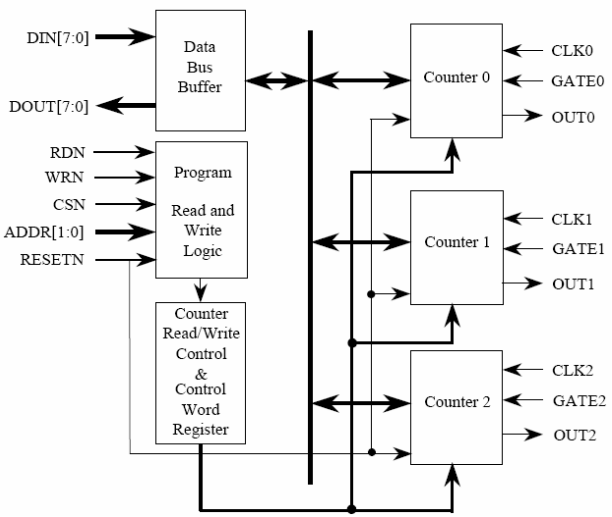
Mode 2 (Bus vào ra hai chiều có bắt tay): Chế độ hoạt động này cung cấp khả năng truyền thông với các ngoại vi hoặc các bus dữ liệu 8-bit cho việc truyền nhận dữ liệu. Các tín hiệu bắt tay được cung cấp để duy trì dòng tín hiệu bus tương tự như chế độ 1. Các cơ chế tạo ngắt cũng có thể được thực hiện ở chế độ này. Một số các chức năng chính hỗ trợ trong chế độ này bao gồm:

- ✓ Chỉ sử dụng nhóm A
- ✓ Một cổng bus 2 chiều 8-bit (cổng A) và một cổng điều khiển 5-bit (Cổng C)
- ✓ Cả hai chiều dữ liệu vào và ra đều đươc chốt.
- ✓ Cổng điều khiển 5-bit (Cổng C) được sử dụng cho mục đích điều khiển và trạng thái cho cổng A để trao đổi dữ liệu 2 chiều 8 bit.

Bộ định thời/Bộ đếm C8254

Đây là bộ đếm tốc độ cao cung cấp 3 bộ định thời 16-bit độc lập và có thể được cấu hình để hoạt động ở nhiều chế độ hoạt động. Mỗi bộ đếm có các kênh dữ liệu và điều khiển riêng biệt. Hỗ trợ 2 kiểu mã hoá đếm nhị phân (0- 65535) hoặc BCD (*binary coded decimal*) (0-9999). Có 4 thanh ghi tích hợp *On-chip* để lưu giá trị đếm và cấu hình hoạt động (từ điều khiển).

Tần số hoạt động của bộ đếm có thể làm việc với xung nhịp tần số 10 MHz và hỗ trợ 6 chế độ hoạt động và có thể cấu hình riêng lẻ.

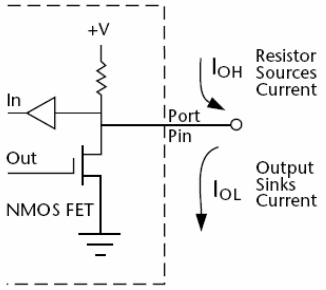


Hình 2-28: Sơ đồ cấu trúc chức năng 8254

2.1.7 Giao diện

Giao diện song song 8bit/16bit

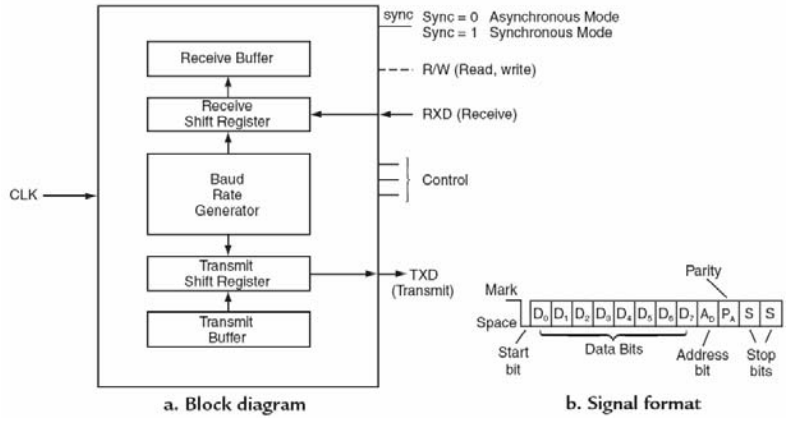
Các cổng song song là một dạng giao diện vào ra đơn giản và phổ biến nhất để kết nối thông tin với ngoại vi. Có nhiều loại cấu trúc giao diện vật lý điện tử từ dạng cổng vào ra đơn giản cực *Collector* TTL hở trong các ứng dụng cổng máy in đến các loại cấu trúc giao diện cổng tốc độ cao như các chuẩn bus IEEE-488 hay SCSI. Hầu hết các chip điều khiển nhúng có một vài cổng vào ra song song khả trình (có thể cấu hình). Các giao diện đó phù hợp với các cổng vào ra đơn giản như các khoá chuyển. Chúng cũng phù hợp trong các bài toán phục vụ giao diện kết nối điều khiển và giám sát theo các giao diện như kiểu role bán dẫn.



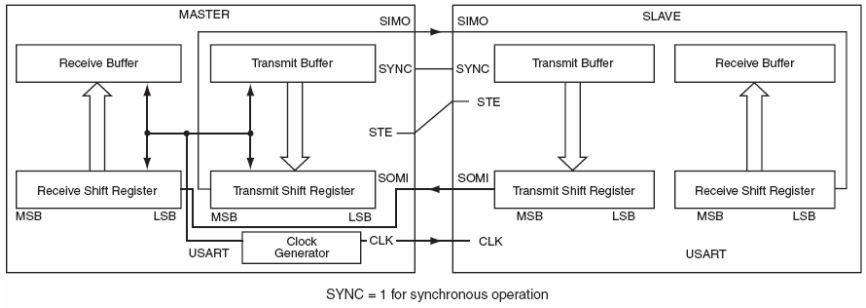
Hình 2-29: Cấu trúc nguyên lý điển hình của một cổng vào/ra logic

Giao diện nối tiếp

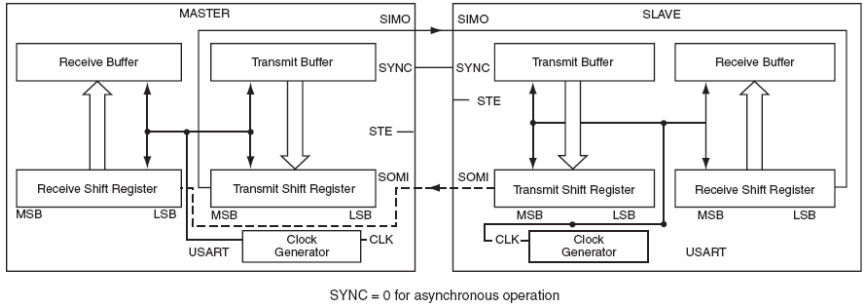
USART



Hình 2-30: Cấu trúc đơn giản hoá của USART



Hình 2-31: Mode hoạt động truyền thông đồng bộ



Hình 2-32: Mode hoạt động truyền thông dị bộ

I²C (Inter-IC)

Giao thức ưu tiên truyền thông nối tiếp được phát triển bởi *Philips Semiconductor* và được gọi là bus I²C. Vì nguồn gốc nó được thiết kế là để điều khiển liên thông IC (*Inter IC*) nên nó được đặt tên là I²C. Tất cả các chíp có tích hợp và tương thích với I²C đều có thêm một giao diện tích hợp trên Chip để truyền thông trực tiếp với các thiết bị tương thích I²C khác. Việc truyền dữ liệu nối tiếp theo hai hướng 8 bit được thực thi theo 3 chế độ sau:

- Chuẩn (*Standard*)—100 Kbits/sec
- Nhanh (*Fast*)—400 Kbits/sec
- Tốc độ cao (*High-Speed*)—3.4 Mbits/sec

Đường bus thực hiện truyền thông nối tiếp I²C gồm hai đường là đường truyền dữ liệu nối tiếp SDA và đường truyền nhịp xung đồng hồ nối tiếp SCL. Vì cơ chế hoạt động là đồng bộ nên nó cần có một nhịp xung tín hiệu đồng bộ. Các thiết bị hỗ trợ I²C đều có một địa chỉ định nghĩa trước, trong đó một số bit địa chỉ là thấp có thể cấu hình. Đơn vị hoặc thiết bị khởi tạo quá trình truyền thông là đơn vị Chủ và cũng là đơn vị tạo xung nhịp đồng bộ, điều khiển cho phép kết thúc quá trình truyền. Nếu đơn vị Chủ muốn truyền thông với đơn vị khác nó sẽ gửi kèm thông tin địa chỉ của đơn vị mà nó muốn truyền trong dữ liệu truyền. Đơn vị Tớ đều được gán và đánh địa chỉ thông qua đó đơn vị Chủ có thể thiết lập truyền thông và trao đổi dữ liệu. *Bus* dữ liệu được thiết kế để cho phép thực hiện nhiều đơn vị Chủ và Tớ ở trên cùng *Bus*.

Quá trình truyền thông I²C được bắt đầu bằng tín hiệu *start* tạo ra bởi đơn vị Chủ. Sau đó đơn vị Chủ sẽ truyền đi dữ liệu 7 bit chứa địa chỉ của đơn vị Tớ mà nó muốn truyền thông, theo thứ tự là các *bit* có trọng số lớn nhất MSB sẽ được truyền trước. Bit thứ tám tiếp theo sẽ chứa thông tin để xác định đơn vị Tớ sẽ thực hiện vai trò nhận (0) hay gửi (1) dữ liệu. Tiếp theo sẽ là một *bit* ACK xác nhận bởi đơn vị nhận đã nhận được 1 *byte* trước đó hay không. Đơn vị truyền (gửi) sẽ truyền đi 1 *byte* dữ liệu bắt đầu bởi MSB. Tại điểm cuối của byte truyền, đơn vị nhận sẽ tạo ra một bit xác nhận ACK mới. Khuôn mẫu 9 bit này (gồm 8 bit dữ liệu và 1 bit xác nhận) sẽ được lặp lại nếu cần truyền tiếp *byte* nữa. Khi đơn vị Chủ đã trao đổi xong dữ liệu cần nó sẽ quan sát *bit* xác nhận ACK cuối cùng rồi sau đó sẽ tạo ra một tín hiệu dừng STOP để kết thúc quá trình truyền thông.

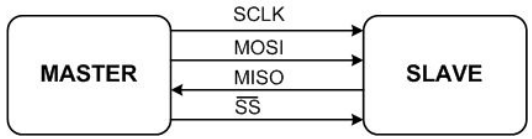
I²C là một giao diện truyền thông đặc biệt thích hợp cho các ứng dụng truyền thông giữa các đơn vị trên cùng một bo mạch với khoảng cách ngắn và tốc độ thấp. Ví dụ như truyền thông giữa CPU với các khối chức năng trên cùng một bo mạch như EEPROM, cảm biến, đồng hồ tạo thời gian thực... Hầu hết các thiết bị hỗ trợ I²C hoạt động ở tốc độ 400Kbps, một số cho phép hoạt động ở tốc độ cao vài Mbps. I²C khá đơn giản để thực thi kết nối nhiều đơn vị vì nó hỗ trợ cơ chế xác định địa chỉ.

SPI

SPI là một giao diện cổng nối tiếp đồng bộ ba dây cho phép kết nối truyền thông nhiều VĐK được phát triển bởi Motorola. Trong cấu hình mạng kết nối truyền thống này phải

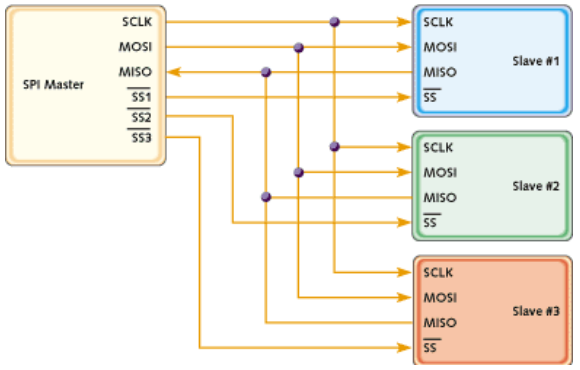
có một VĐK giữ vai trò là Chủ (*Master*) và các VĐK còn lại có thể hoặc là Chủ hoặc là Tớ. SPI có 4 tốc độ có thể lập trình, cực và pha nhịp đồng hồ khả trình và kết thúc ngắt truyền thông. Nhịp đồng hồ không nằm trong dòng dữ liệu và phải được cung cấp như một tín hiệu tách độc lập. Có ba thanh ghi SPSR, SPCR và SPDR cho phép thực hiện các chức năng điều khiển, trạng thái và lữu trữ. Có bốn chân cơ bản cần thiết để thực thi chuẩn giao diện truyền thông này.

- Dữ liệu ra MOSI (Master Output – Slave Input)
- Dữ liệu vào MISO (Master Input – Slave Output)
- Nhịp xung chuẩn SCLK (*Serial Clock*)
- Lựa chọn thành phần tớ SS (*Slave Select*)



Hình 2-33: Kết nối nguyên lý truyền thông SPI giữa một Master và một Tớ

Hình 2-33 chỉ ra nguyên lý kết nối giữa một đơn vị Chủ và một đơn vị Tớ trong truyền thông SPI. Trong đó tín hiệu SCLK sẽ được tạo ra bởi đơn vị Chủ và là tín hiệu vào của đơn vị Tớ. MOSI là đường truyền dữ liệu ra từ đơn vị Chủ tới đơn vị Tớ và MISO là đường truyền dữ liệu vào đơn vị Chủ đến từ đơn vị Tớ. Đơn vị Tớ được lựa chọn khi đơn vị Chủ kích hoạt tín hiệu $\overline{SS}$.



Hình 2-34: Sơ đồ kết nối truyền thống SPI của một đơn vị Chủ với nhiều đơn vị Tớ

Nếu hệ thống có nhiều đơn vị tớ đơn vị Chủ sẽ tạo phải ra các tín hiệu tách biệt để chọn đơn vị Tớ. Cơ chế đó được thực hiện nhờ sơ đồ kết nối nguyên lý mô tả như trong Hình 2-34. Đơn vị Chủ sẽ tạo ra tín hiệu chọn đơn vị Tớ nhờ các chân tín hiệu logic đa chức năng. Các tín hiệu này phải được điều khiển và đảm bảo ổn định về thời gian để tránh trường hợp tín hiệu bị thay đổi trong quá trình đang truyền dữ liệu. Một điều dễ nhận

ra rằng SPI không hỗ trợ cơ chế xác nhận trong quá trình thực hiện truyền thông. Điều này phụ thuộc vào giao thức định nghĩa hoặc phải thực hiện bổ sung thêm một số các mở rộng phụ bên ngoài.

Khả năng truyền thông đồng thời hai chiều với tốc độ lên đến khoảng vài Mbit/s và nguyên lý khá đơn giản nên SPI hoàn toàn phù hợp để thực hiện truyền thông giữa các thiết bị yêu cầu truyền thông tốc độ chậm, đặc biệt hiệu quả trong các ứng dụng một đơn vị Chủ và một đơn vị Tớ. Tuy nhiên trong các ứng dụng với nhiều đơn vị Tớ việc thực thi lại khá phức tạp vì thiếu cơ chế xác định địa chỉ, và sự phức tạp sẽ tăng lên khi số đơn vị Tớ tăng.

2.2 Một số nền phần cứng nhúng thông dụng (µP/DSP/PLA)

Trong phần này giới thiệu ngắn gọn cấu trúc nguyên lý của các chip xử lý nhúng ứng dụng trong các nền phần cứng nhúng hiện nay.

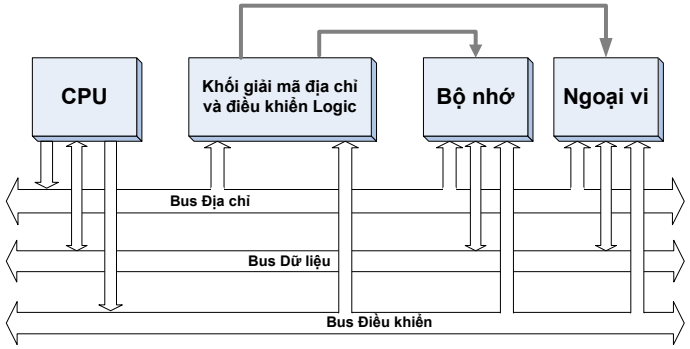
Sự phát triển nhanh chóng các chủng loại Chip khả trình với mật độ tích hợp cao đã và đang có một tác động đáng kể đến sự thay đổi trong việc thiết kế các nền phần cứng thiết bị xử lý và điều khiển số trong thập kỷ gần đây. Mỗi chủng loại đều có những đặc điểm và phạm vi đối tượng ứng dụng và luôn không ngừng phát triển để đáp ứng một cách tốt nhất cho các yêu cầu công nghệ. Chúng đang hướng tới tập trung cho một thị trường công nghệ tiềm năng rộng lớn đó là các thiết bị xử lý và điều khiển nhúng. Trong bài viết này tác giả giới thiệu ngắn gọn về các chủng loại chip xử lý, điều khiển nhúng điển hình đang tồn tại và phát triển về một số đặc điểm và hướng phạm vi ứng dụng của chúng.

Có thể kể ra hàng loạt các Chíp khả trình có thể sử dụng cho các bài toán thiết kế hệ nhúng như các họ vi xử lý/vi điều khiển nhúng (*Microprocessor/ Microcontroller*), Chip DSP (*Digital Signal Processing*), các Chip khả trình trường (FPD – *Field Programmable Device*). Chúng ta dễ bị choáng ngợp nếu bắt đầu công việc thiết kế bằng việc tìm kiếm một Chip xử lý điều khiển phù hợp cho ứng dụng. Vì vậy cần phải có một hiểu biết và sự phân biệt về đặc điểm và ứng dụng của chúng khi lựa chọn và thiết kế. Các thông tin liên quan như nhà sản xuất cung cấp Chip, các kiến thức và công cụ phát triển kèm theo…Một số chủng loại Chip điển hình sẽ được giới thiệu.

2.2.1 Chip Vi xử lý / Vi điều khiển nhúng

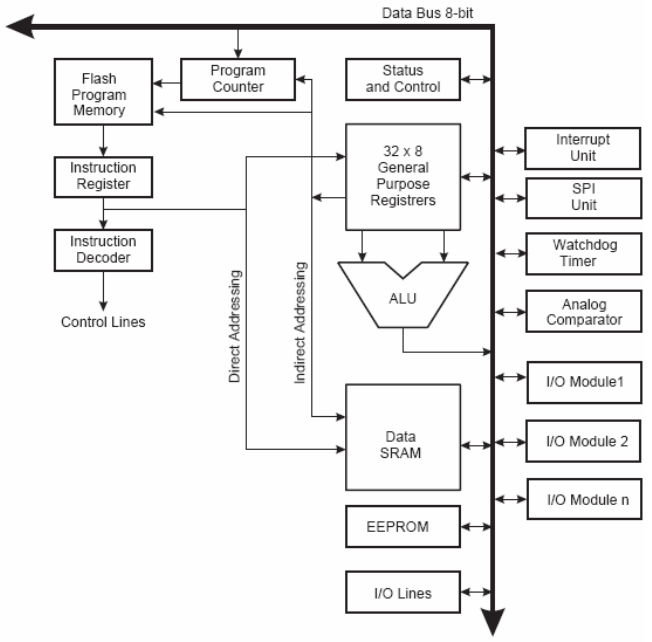
Đây là một chủng loại rất điển hình và đang được sử dụng rất phổ biến hiện này. Chúng được ra đời và sử dụng theo sự phát triển của các Chip xử lý ứng dụng cho máy tính. Vì đối tượng ứng dụng là các thiết bị nhúng nên cấu trúc cũng được thay đổi theo để đáp ứng các ứng dụng. Hiện nay chúng ta có thể thấy các họ vi xử lý điều khiển của rất nhiều các nhà chế tạo cung cấp như, *Intel*, *Atmel*, *Motorola*, *Infineon*. Về cấu trúc, chúng cũng tương tự như các Chíp xử lý phát triển cho PC nhưng ở mức độ đơn giản hơn nhiều về công năng và tài nguyên. Phổ biến vẫn là các Chip có độ rộng bus dữ liệu là 8-bit, 16-bit, 32-bit. Về bản chất cấu trúc, Chip vi điều khiển là chip vi xử lý được tích

hợp thêm các ngoại vi. Các ngoại vi thường là các khối chức năng ngoại vi thông dụng như bộ định thời gian, bộ đếm, bộ chuyển đổi A/D, giao diện song song, nối tiếp…Mức độ tích hợp ngoại vi cũng khác nhau tuỳ thuộc vào mục đích ứng dụng sẽ có thể tìm được Chip phù hợp. Thực tế với các ứng dụng yêu cầu độ tích hợp cao thì sẽ sử dụng giải pháp tích hợp trên chip, nếu không thì hầu hết các Chip đều cung cấp giải pháp để mở rộng ngoại vi đáp ứng cho một số lượng ứng dụng rộng và mềm dẻo.

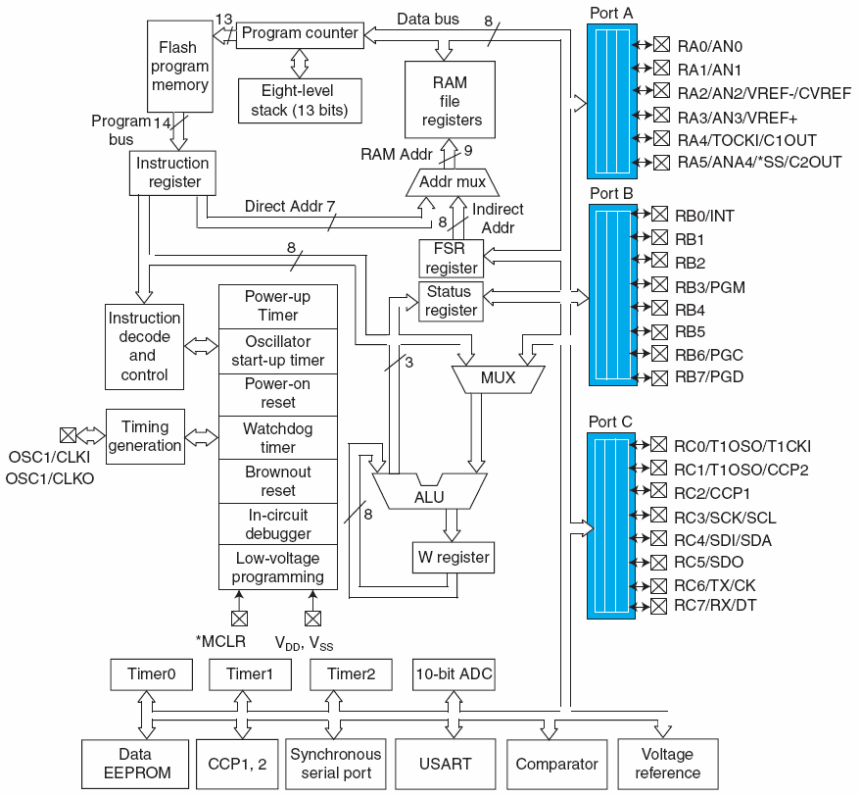


Hình 2-35: Kiến trúc nguyên lý của VĐK với cấu trúc Havard

Ví dụ về kiến trúc của họ VĐK AVR



Hình 2-36: Kiến trúc của họ VĐK AVR



Hình 2-37: Sở đồ khối chức năng kiến trúc PIC16F873A

2.2.2 Chip DSP

[Ref. Sen Kuo]
DSP vẫn được biết tới như một loại vi điều khiển đặc biệt với khả năng xử lý nhanh để phục vụ các bài toán yêu cầu khối lượng và tốc độ xử lý tính toán lớn. Với ưu điểm nổi bật về độ rộng băng thông của bus và thanh ghi tích luỹ, cho phép ALU xử lý song song với tốc độ đọc và xử lý lệnh nhanh hơn các loại vi điều khiển thông thường. Chip DSP cho phép thực hiện nhiều lệnh trong một nhịp nhờ vào kiến trúc bộ nhớ *Havard*.

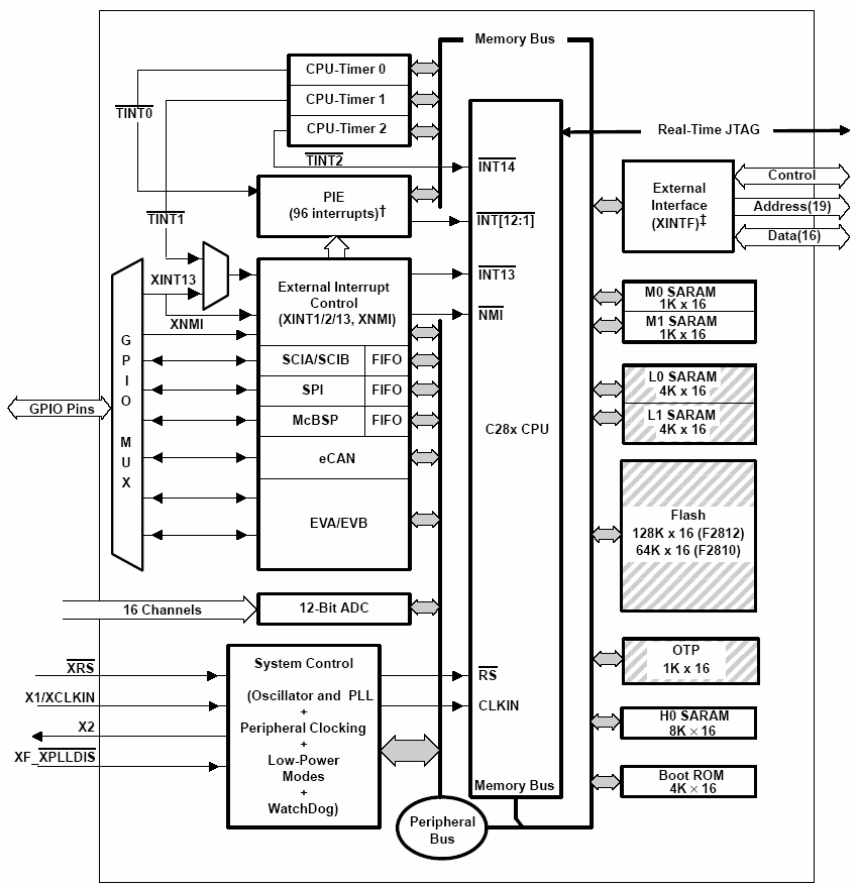
Thông thường khi phải sử dụng DSP tức là để đáp ứng các bài toán tính toán lớn và tốc độ cao vì vậy định dạng biểu diễn toán học sẽ là một yếu tố quan trọng để phân loại và được quan tâm. Hiện nay chủ yếu chúng vẫn được phân loại theo hai kiểu là dấu phảy động và dấu phảy tĩnh. Đây cũng chính là một yếu tố quan trọng phải quan tâm đối với người thiết kế để lựa chọn được một DSP phù hợp với ứng dụng của mình. Các loại DSP dấu phảy tĩnh thường là loại 16-bit hoặc 24-bit còn các loại dấu phảy tĩnh thường là 32-bit. Một ví dụ điển hình về một DSP 16-bit dấu phảy tĩnh là TMS320C55x, lưu các

số nguyên 16-bit hoặc các số thực trong một miền giá trị cố định. Tuy nhiên các giá trị và hệ số trung gian có thể được lưu trữ với độ chính xác là 32-bit trong thanh ghi tích luỹ 40-bit nhằm giảm thiểu lỗi tính toán do phép làm tròn trong quá trình tính toán. Thông thường các loại DSP dấu phảy tĩnh có giá thành rẻ hơn các loại DSP dấu phảy động vì yêu cầu số lượng chân *On-chip* ít hơn và cần sử dụng lượng *silicon* ít hơn.

Ưu điểm nổi bật của các DSP dấu phảy động là có thể xử lý và biểu diễn số trong dải phạm vi giá trị rộng và động. Do đó vấn đề về chuyển đổi và hạn chế về phạm vi biểu diễn số không phải quan tâm như đối với loại DSP dấu phảy tĩnh. Một loại DSP 32-bit dấu phảy tĩnh điển hình là TMS320C67x có thể xử lý và biểu diễn số gồm 24-bit *mantissa* và 8-bit *exponent*. Phần *mantissa* biểu diễn phần số lẻ trong phạm vi -1.0 – +1.0 và phần *exponent* biểu diễn vị trí của dấu phảy nhị phân và có thể dịch chuyển sang trái hoặc phải tuỳ theo giá trị số mà nó biểu diễn. Điều này trái ngược với các thiết kế trên nền DSP dấu phảy tĩnh, người phát triển chương trình phải tự qui ước, tính toán và phân chia ấn định thang biểu diễn số và phải luôn lưu tâm tới khả năng tràn số có thể xảy ra trong quá trình xử lý tính toán. Chính điều này đã gây ra khó khăn không nhỏ đối với người lập trình. Nói chung phát triển chương trình cho DSP dấu phảy động thường đơn giản hơn nhưng giá thành lại cao hơn nhiều và năng lượng tiêu thụ thông thường cũng lớn hơn.

Ví dụ độ chính xác của DSP dấu phảy động 32 bit là 2^{-23} với 24 bit biểu diễn phần *mantissa*. Vùng động là $1.18 \times 10^{-38} \le x \le 3.4 \times 10^{38}$.

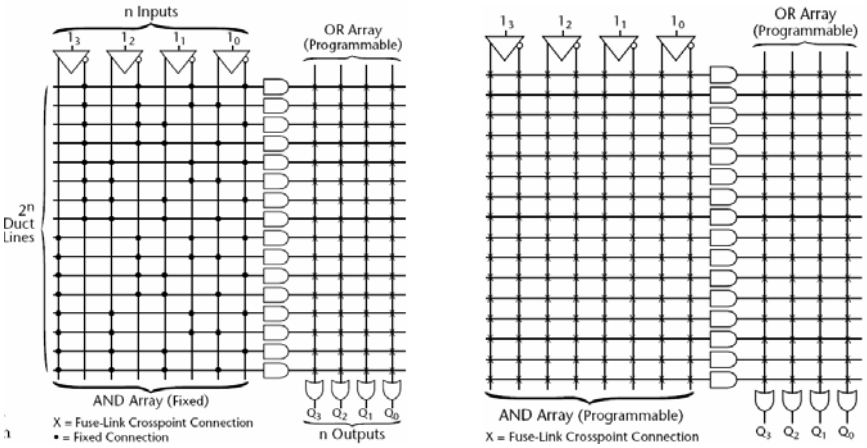
Những nhà thiết kế hệ thống phải quyết định vùng và độ chính xác cần thiết cho các ứng dụng. Các vi xử lý dấu phảy động thường được sử dụng cho các ứng dụng yêu cầu về độ chính xác cao và dải biểu diễn số lớn phù hợp với hệ thống có cấu trúc bộ nhớ lớn Hơn nữa các DSP dấu phảy động cho phép phát triển phần mềm hiệu quả và đơn giản hơn bằng các trình biên dịch ngôn ngữ bậc cao như C do đó có thể giảm được giá thành và thời gian phát triển. Tuy nhiên giá thành lại cao nên các DSP dấu phảy động phù hợp với các ứng dụng khá đặc biệt và thường là với số lượng ít.



Hình 2-38: Giản đồ khối chức năng của DSP TMS320C28xx

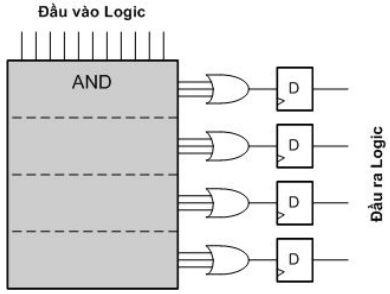
2.2.3 PAL

Ngày nay khi nói đến các chủng loại Chip khả trình mảng ta thường biết tới một số tên gọi như PAL, CPLD, FPGA…Một chút lược sử về sự ra đời và phát triển sau đây sẽ giúp chúng ta hình dung được đặc điểm và nguồn gốc ra đời của chúng.



Hình 2-39: Cấu trúc PROM và PLA

Lịch sử phát triển của các chủng loại Chip khả trình mảng PLA (*Programmable Logic Array*) được bắt nguồn từ nguyên lý bộ nhớ chương trình PROM (*Programmable Read-Only Memory*). Trong đó các đầu vào địa chỉ đóng vai trò như các đường vào của mạch logic và các đường dữ liệu ra đóng vai trò như các đường ra của mạch logic. Vì PROM không thực sự phù hợp cho mục đích thiết kế các mạch logic nên PLA đã ra đời vào đầu thập kỷ 70. Nó rất phù hợp để thực hiện mạch logic có dạng tổng các tích (vì cấu thành bởi các phần tử logic AND và OR). Nhưng nhược điểm là chi phí sản xuất cao và tốc độ hoạt động thấp. Để khắc phục nhược điểm này PAL (*Programmable Array Logic*) đã được phát triển. Nó được cấu thành từ các phần tử AND khả trình và phần tử OR gán cố định và có chứa cả phần tử *flip-flop* ở đầu ra nên có khả năng thực thi các mạch logic tuần tự. Hình 2-40 mô tả cấu trúc chung của PAL.

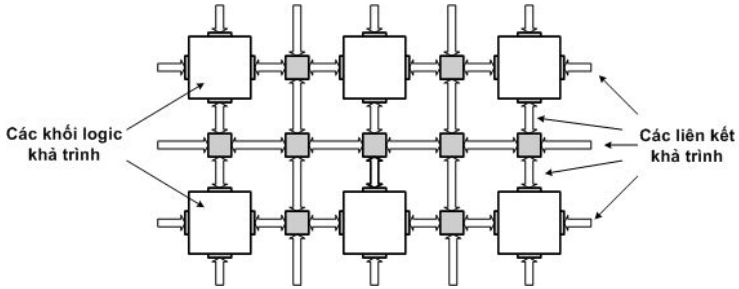


Hình 2-40: Cấu trúc chung của PAL

Từ khi được ra đời và phát triển PAL trở thành cơ sở cho sự ra đời của hàng loạt các chủng loại Chip khả trình mảng với cấu trúc phức tạp hơn như SPLD (*Simple Programmable Logic Device*), CPLD (*Com-plex Programmable Logic Device*), và sau này là FPGA (*Field Pro-grammable Gate Array*). SPLD cũng là tên gọi cho nhóm các chủng loại Chip

kiểu tương tự như PAL, PLA. Về mặt cấu trúc thì SPLD cho phép tích hợp logic với mật độ cao hơn so với PAL thông thường, nhưng kích thước của nó sẽ tăng lên rất nhanh nếu tiếp túc mở rộng và tăng mật độ tích hợp số đầu vào. Để đáp ứng nhu cầu mở rộng mật độ tích hợp CPLD đã được phát triển. Nó là sự tích hợp của nhiều khối SPLD và cung cấp thêm khả năng kết nối khả trình giữa các khối SPLD đơn lẻ với nhau. Với nguyên lý cấu trúc này CPLD có khả năng tích hợp với mật độ cao tương đương với 50 khối SPLD thông thường.

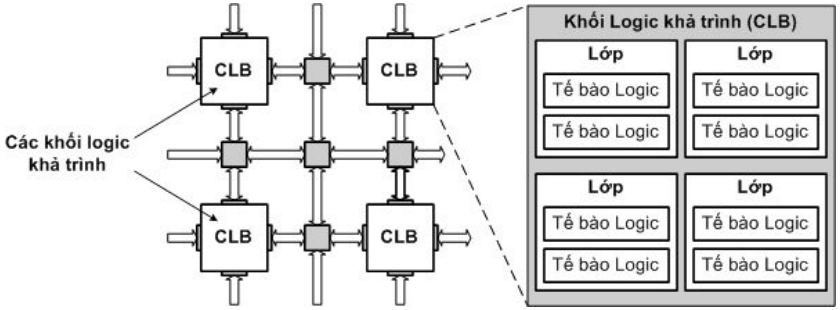
Nếu chỉ dừng đến đây chúng ta có thể thấy một đặc điểm chung của các chủng loại chip kiểu PLA hay CPLD đều cho phép thực hiện các mạch logic trên cơ sở tổ hợp logic của các đầu vào và ra bằng các phần tử AND và OR. Với nguyên lý này rõ ràng sẽ gặp khó khăn khi thực thi các ứng dụng đòi hỏi các phép tính toán logic phức tạp với tốc độ cao. Để đáp ứng điều này FPGA (*Field Programmable Gate Arrays*) đã ra đời. Nó là sự cấu thành của các khối logic khả trình cùng với các kênh kết nối liên thông khả trình giữa các khối đó với nhau. Một hình ảnh tiêu biểu về cấu trúc nguyên lý của FPGA được mô tả như trong Hình 2-41: Cấu trúc nguyên lý của FPGA.



Hình 2-41: Cấu trúc nguyên lý của FPGA

FPGA - đang trở thành một sự lựa chọn thay thế rất cạnh tranh của các chip xử lý nhúng ASICs. Nó hỗ trợ các ưu điểm về chức năng lựa chọn giống như ASICs nhưng cho phép chỉnh sửa và thiết kế lại sau khi sử dụng và giá thành phát triển thấp hơn. FPGA cho phép khả năng thiết kế linh hoạt và thích nghi dễ dàng cho các tiện ích thiết bị tối ưu, trong khi vẫn duy trì được không gian kích thước phần cứng và năng lượng tiêu thụ của hệ thống. Điều này không dễ dàng nhận được khi thiết kế dựa trên nền các Chip DSP.

FPGA thực sự phù hợp cho các ứng dụng đòi hỏi lượng tính toán lớn như trong xử lý tín hiệu. FPGA có thể được lập trình hoạt động đồng thời với một số các đường dữ liệu song song. Chúng là các đường dữ liệu hoạt động của tổ hợp nhiều các chức năng từ đơn giản đến phức tạp như bộ cộng, bộ nhân, bộ đếm, bộ lưu trữ, bộ so sánh, bộ tính tương quan, …



Hình 2-42: Cấu trúc CLB và LAB

Ngày nay có thể phân loại ra một số kiểu chủng loại FPGA dựa vào cấu tạo của chúng:

■ **Cấu tạo từ SRAM**:
Với loại này các mắt kết nối khả trình được thực hiện bằng các phần tử SRAM, chính vì vậy cho phép thực hiện lập trình lặp lại nhiều lần. Ưu điểm nổi bật của loại này là các ý tưởng thiết kế mới có thể được thực thi và thử nghiệm nhanh chóng. Hơn nữa SRAM cũng đang là một hướng phát triển rất mạnh hiện nay trong nền công nghiệp sản xuất bộ nhớ và cũng đều thực thi theo công nghệ CMOS rất phù hợp với công nghệ chế tạo FPGA.

Tuy nhiên một đặc điểm có thể xem như là nhược điểm của FPGA cấu tạo từ các phần tử SRAM là chúng phải cấu hình lại mỗi khi nguồn hệ thống được cung cấp. Công việc này thường được thực hiện bởi một bộ nhớ ngoài chuyên dụng hoặc bởi một bộ vi điều khiển kèm theo mạch. Chính vì vậy cũng làm giá thành của FPGA tăng thêm.

■ **Cấu tạo từ cầu chì** (*anti-fused)*
Không giống như loại FPGA cấu tạo từ SRAM, FPGA với cấu tử kiểu cầu chì được lập trình *offline* bằng một thiết bị lập trình chuyên dụng. Ý tưởng chế tạo loại FPGA này xuất phát từ nhu cầu về một thiết bị khả trình có khả năng lưu cấu hình sau khi được sử dụng. Tức là nó không phải làm công việc cấu hình mỗi khi nguồn hệ thống được cung cấp. Khi FPGA *anti-fused* đã được lập trình thì nó không thể bị thay đổi hay được lập trình lại nữa. Chính nhờ điều này nên nó không cần bất kỳ một bộ nhớ ngoài nào để lưu trữ cấu hình và có thể tiết kiệm, giảm giá thành của thiết bị.

Một ưu điểm nổi bật của FPGA *anti-fused* là kiểu cấu trúc liên kết khá bền vững với các loại nhiễu bức xạ. Đặc điểm này khá quan trọng khi thiết bị phải làm việc trong môi trường tiềm năng như quân sự hoặc hàng không vũ trụ. Vì vậy nó tránh được trường hợp rủi ro có thể xảy ra nếu sử dụng công nghệ SRAM là hiện tượng lật trạng thái (*flipped*). Tuy nhiên hiện tượng này cũng có thể được khắc phục bằng cơ chế dự phòng chập 3 nhưng lại làm tăng thêm chi phí chế tạo.

Một ưu điểm nổi bật của loại FPGA *anti-fused* là khả năng bảo vệ công nghệ. Tức là dữ liệu cấu hình lập trình cho FPGA có thể được bảo vệ bởi việc đọc bất hợp pháp hoặc không cho phép đọc. Trong qúa trình xử lý hoặc phát triển, người lập trình sẽ sử dụng một tệp dữ liệu cấu hình để lập trình và kiểm tra quá trình nạp cấu hình cho FPGA. Công việc này chỉ thực hiện một lần và sẽ không thể thay đổi được nữa. Khi thực hiện xong nó có thể được thiết lập thêm một thuộc tính là chống đọc trực tiếp từ FPGA dữ liệu liên quan đến cấu hình. Ngoài ra chúng ta cũng có thể biết thêm rằng FPGA *anti-fused* thường sử dụng ít năng lượng hơn loại FPGA SRAM, kích thước cũng nhỏ hơn, và tốc độ cũng nhanh hơn một chút nhờ khoảng cách kết nối cứng giữa các phần tử ngắn hơn.

Tuy nhiên nhược điểm lớn nhất của FPGA *anti-fused* là chỉ có thể được lập trình và cấu hình một lần. Vì vậy nó chỉ thực sự phù hợp khi thực thi hoàn chỉnh sản phẩm cuối cùng và không phù hợp với mục đích thiết kế phát triển.

■ **Cấu tạo từ EEPROM/FLASH**

EEPROM or FLASH-based FPGAs cũng có nguyên lý cấu tạo tương tự như loại FPGA-SRAM. Các phần tử cấu hình của nó được kết nối dựa trên một chuỗi thanh ghi dịch dài. Chúng có thể được cấu hình *offline* bằng các thiết bị lập trình chuyên dụng. Cũng có một số có thể lập trình *online* nhưng thời gian lập trình cấu hình sẽ gấp khoảng 3 lần thời gian thực thi với nền FPGA-SRAM. Khi đã được cấu hình đã được lập trình thì chúng có thể được duy trì và không bị mất đi như nguyên lý lưu giữ của EEPROM hoặc FLASH. Loại FPGA-EEPROM/FLASH có cấu tạo nhỏ hơn so với loại FPGA-SRAM vì vậy cũng có thể giảm được thời gian lan truyền tín hiệu kết nối liên thông giữa các phần tử *logic*.

Để bảo vệ công nghệ khi FPGA đã được cấu hình và đưa ra sử dụng, ta có thể bảo vệ bằng cơ chế khóa mã mềm (cấu tạo từ khoảng 50 bit đến vài trăm bit). Muốn đọc được thông tin cấu hình trực tiếp từ FPGA, người ta cần phải có mã khóa đó và cũng rất khó hoặc không thể mò được theo nguyên lý thử sai. Vì muốn vậy theo ước tính cũng phải mất đến hàng triệu năm mới hy vọng thành công để mò ra được.

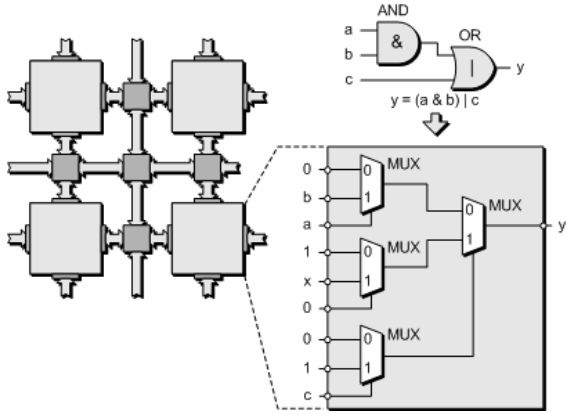
Tuy nhiên công nghệ chế tạo FPGA-EEPROM/FLASH đòi hỏi thực thi qua nhiều công đoạn xử lý hơn so với loại FPGA-SRAM vì vậy mà sự phát triển của chúng cũng chậm hơn. Hơn nữa năng lượng tiêu thụ của chúng cũng lớn hơn vì phải nuôi rất nhiều các phần tử điện trở kéo (*pull-up resistor*).

■ **Cấu tạo từ tổ hợp FLASH-SRAM**

Ngày nay người ta cũng phát triển chế tạo các loại FPGA cấu tạo từ các tổ hợp SRAM và FLASH để tận dụng được các ưu điểm của cả hai chủng loại này. Thông thường các phần tử cấu hình FLASH sẽ được sử dụng để lưu các nội dung cấu hình để sao chép cho các phần tử cấu hình SRAM. Và các phần tử cấu hình SRAM hoàn toàn có thể được cấu hình lại theo yêu cầu thiết kế trong khi vẫn duy trì một phần thiết kế cấu hình gốc lưu trong các phần tử FLASH.

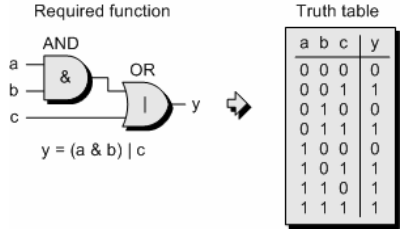
Người ta cũng thường phân loại FPGA dựa vào phần tử kiến trúc của chúng và bao gồm 3 loại chính: mịn, thô và trung bình. Bản chất việc phân loại này là dựa vào kiểu khối logic khả trình cấu thành nên FPGA. Với loại FPGA mịn thì kiến trúc các khối *logic* khả trình thường là các cổng logic đơn giản (kiểu AND, OR…, và các phần tử lưu giữ như *Triger* D…). Kiểu kiến trúc này phù hợp và thường sử dụng hiệu quả với kiến trúc ASICó. Gần đây xu thế phát triển của FPGA đang tập trung vào loại kiến trúc thô. Tức là các khối *logic* khả trình là các khối có khả năng xử lý logic lớn với nhiều tổ hợp liên kết và phức tạp với nhiều đầu vào và ra liên kết. Tuỳ theo mức độ của khối *logic* khả trình đó mà người ta phân ra thành các loại trung bình.

Có hai loại cấu trúc cơ bản cấu thành nên các khối logic khả trình trong kiến trúc FPGA thô hoặc trung bình là MUX (*Multiplexer*) và LUT (*Lookup Table*). Trong loại cấu trúc MUX thì các phần tử logic được cấu thành theo cấu trúc tổ hợp các đầu vào ra theo nguyên lý MUX như mô tả trong Hình 2-43: Khối logic dạng MUX.



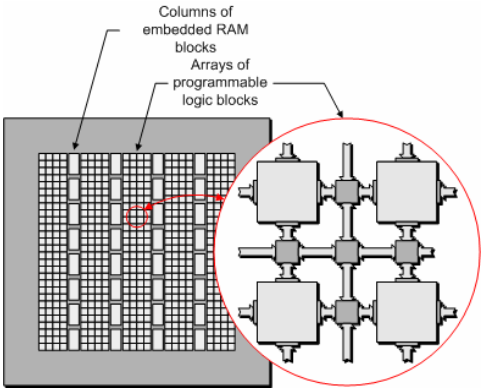
Hình 2-43: Khối logic dạng MUX

Đối với loại cấu trúc LUT thì các đầu vào thực chất là các tổ hợp để chọn ra giá trị trong bảng chất lý của hàm chức năng cần thực thi. Nguyên lý của loại khối logic này được mô tả như trong Hình 2-44.



Hình 2-44: LUT thực hiện hàm tổ hợp AND và OR

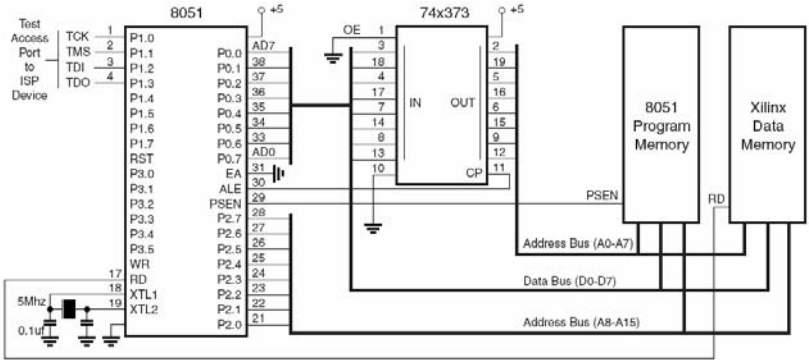
Hầu hết các ứng dụng đều có nhu cầu về bộ nhớ RAM on Chip vì vậy một số dòng FPGA hiện nay cũng tích hợp thêm cả các phần tử nhớ RAM và được gọi là RAM nhúng (*embedded* RAM). Các phần tử RAM đó được tổ chức thành từng khối và tuỳ thuộc vào kiến trúc của FPGA nó sẽ được phân bố linh hoạt, thường là xung quanh các phần tử ngoại vi hoặc phân bố đều trên bề mặt Chip. Một hình ảnh minh hoạ về phân bố RAM trong kiến trúc FPGA được mô tả như trong Hình 2-45.



Hình 2-45: Hình ảnh của Chip có các cột là các khối RAM nhúng

■ **FPGA với hạt nhân DSP**

Thực chất đó là một tổ hợp nhằm tăng tốc và khả năng tính toán. Khái niệm này cũng tương tự như các bộ đồng xử lý toán học trong kiến trúc máy tính. Nguyên lý là nhằm san sẻ và giảm bớt tải sang FPGA để thực thi các chức năng tính toán lớn (thông thường đòi hỏi thực hiện trong nhiều nhịp hoạt động của Chip DSP) và cho phép Chip DSP tập trung thực hiện các chức năng đơn nhịp tối ưu. Tổ hợp FPGA và DSP là một kiến trúc rất linh hoạt và đặc biệt cải thiện được hiệu suất thực hiện và tăng tốc hơn rất nhiều so với kiến trúc nhiều Chip DPS hoặc ASICs đồng thời giá thành lại thấp hơn.



Hình 2-46: Sơ đồ nguyên lý mạch ghép nối VĐK và FPGA

3 CƠ SỞ KỸ THUẬT PHẦN MỀM NHÚNG

3.1 Đặc điểm phần mềm nhúng

- ✓ Hướng chức năng hoá đặc thù
- ✓ Hạn chế về tài nguyên bộ nhớ
- ✓ Yêu cầu thời gian thực

3.2 Biểu diễn số và dữ liệu

- Đơn vị cơ bản nhất trong biểu diễn thông tin của hệ thống số được gọi là *bit*, chính là ký hiệu viết tắt của thuật ngữ *binary digit*.
- 1964, IBM đã thiết kế và chế tạo máy tính số sử dụng một nhóm 8 bit để đánh địa chỉ bộ nhớ và định nghĩa ra thuật ngữ 8 *bit* = 1 *byte*.
- Ngày nay sử dụng rộng rãi thuật ngữ *word* là một từ dữ liệu dùng để biểu diễn kích thước dữ liệu mà được xử lý một cách hiệu quả nhất đối với mỗi loại kiến trúc xử lý số cụ thể. Chính vì vậy một từ có thể là 16 *bits*, 32 *bits*, hoặc 64 *bits*…
- Mỗi một *byte* có thể được chia ra thành hai nửa 4 *bit* và được gọi là các *nibble*. *Nibble* chứa các bít trọng số lớn được gọi là *nibble* bậc cao, và *nibble* chứa các bit trọng số nhỏ được gọi là *nibble* bậc thấp.

3.2.1 Các hệ thống cơ số

Trong các hệ thống biểu diễn số hiện nay đều được biểu diễn ở dạng tổng quát là tổng luỹ thừa theo cơ số, và được phân loại theo giá trị cơ số. Một cách tổng quát một hệ biểu diễn số cơ số b và a là một số nguyên nằm trong khoảng giá trị cơ số b được biểu diễn như sau:

$$A = a_n b^n + a_{n-1} b^{n-1} + \cdots + a_0 = \sum_{i=0}^{n} a_i \cdot b^i \qquad (1.1)$$

Ví dụ như cơ số *binary* (nhị phân), cơ số *decimal* (thập phân), cơ số *hexadecimal*, cơ số 8 *Octal* (bát phân).

Ví dụ về biểu diễn các giá trị trong các hệ cơ số khác nhau:

$243.51_{10} = 2 \times 10^2 + 4 \times 10^1 + 3 \times 10^0 + 5 \times 10^{-1} + 1 \times 10^{-2}$

$212_3 = 2 \times 3^2 + 1 \times 3^1 + 2 \times 3^0 = 23_{10}$

$10110_2 = 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 22_{10}$

Hai loại cơ số biểu diễn thông dụng nhất hiện nay cho các hệ thống xử lý số là cơ số nhị phân và cơ số mười sáu.

3.2.2 Số nguyên

Trong biểu diễn số có dấu để phân biệt số dương và số âm người ta sử dụng *bit* trọng số lớn nhất qui ước làm *bit* dấu và các *bit* còn lại được sử dụng để biểu diễn giá trị độ lớn của số. Ví dụ một từ 8 *bit* được sử dụng để biểu diễn giá trị -1 sẽ có dạng nhị phân là **1**0000001, và giá trị +1 sẽ có dạng **0**0000001. Như vậy với một từ 8 *bit* có thể biểu diễn

được các số trong phạm vi từ -127 đến +127. Một cách tổng quát một từ N *bit* sẽ biểu diễn được $-2^{(N-1)}-1$ đến $+2^{(N-1)}-1$.

Chú ý khi thực hiện cộng hai số có dấu:

- ✓ Nếu hai số cùng dấu thì thực hiện phép cộng phần biểu diễn giá trị và sử dụng *bit* dấu cùng dấu với hai số đó.
- ✓ Nếu hai số khác dấu thì kết quả sẽ nhận dấu của toán tử lớn hơn, và thực hiện phép trừ giữa toán tử có giá trị lớn hơn với toán tử bé hơn.

Ví dụ 1: Cộng hai số có dấu 01001111_2 và 00100011_2.

```
          1 1 1 1     ⇐   carries
  0   1 0 0 1 1 1 1         (79)
+ 0   0 1 0 0 0 1 1       + (35)
  0   1 1 1 0 0 1 0         (114)
```

Ví dụ 2: Cộng hai số có dấu 01001111_2 và 01100011_2.

```
Nhớ cuối cùng   1 ←       1 1 1 1     ⇐ carries
Tràn            0     1 0 0 1 1 1 1       (79)
bỏ nhớ          0  +  1 1 0 0 0 1 1     + (99)
                0     0 1 1 0 0 1 0       (50)
```

Ví dụ 3: Trừ hai số có dấu 01001111_2 và 01100011_2.

```
          0 1 1 2     ⇐ borrows
  0   1 1 0 0 0 1 1       (99)
- 0   1 0 0 1 1 1 1     - (79)
  0   0 0 1 0 1 0 0       (20)
```

Ví dụ 4: Cộng hai số khác dấu 10010011_2 (-19) và 00001101_2 (+13)

```
          0 1 2     ⇐ borrows
  1   0 0 1 0 0 1 1       (-19)
- 0   0 0 1 1 0 1 1     + (13)
  1   0 0 0 0 1 1 0       (-6)
```

Thuật toán thực hiện phép tính có dấu:

(1) Khai báo và xóa các biến lưu giá trị và dấu để chuẩn bị thực hiện phép tính.
(2) Kiểm tra dấu của toán tử thứ nhất để xem có phải số âm không. Nếu là số âm thì thực hiện bù dấu và bù toán tử. Nếu không thì chuyển qua thực hiện bước 3.
(3) Kiểm tra dấu của toán tử thứ hai để xem có phải số âm không. Nếu là số âm thì thực hiện bù dấu và bù toán tử. Nếu không thì chuyển sang thực hiện bước 4.
(4) Thực hiện phép nhân hoặc chia với các toán tử vừa xử lý.
(5) Kiểm tra dấu. Nếu *zero* thì coi như đã kết thúc. Nếu bằng -1 (0ffh) thì thực hiện phép tính bù hai với kết quả thu được và kết thúc.

Hiện nay người ta sử dụng hai qui ước biểu diễn số nguyên phân biệt theo thứ tự của *byte* trọng số trong một từ được biểu diễn:

- **Litte edian**: *byte* trọng số nhỏ nhất đứng trước → thuận lợi cho phép cộng hoặc trừ và
- **Big endian**: *byte* trọng số lớn nhất đứng trước → thuận lợi cho phép nhân hoặc chia.

Ví dụ xét một số nhị phân 4-*byte*

Theo qui ước biểu diễn *litte edian* thì thứ tự địa chỉ lưu trong bộ nhớ sẽ là:

Địa chỉ cở sở + 0 = Byte 0
Địa chỉ co sở + 1 = Byte 1
Địa chỉ cơ sở + 2 = Byte 2
Địa chỉ cơ sở + 3 = Byte 3

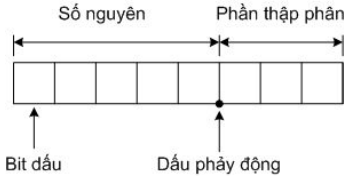
Và theo qui ước biểu diễn số *big edian* sẽ là:

Địa chỉ cở sở + 0 = Byte 3
Địa chỉ co sở + 1 = Byte 2
Địa chỉ cơ sở + 2 = Byte 1
Địa chỉ cơ sở + 3 = Byte 0

3.2.3 Số dấu phảy tĩnh

Chúng ta có thể sử dụng một ký hiệu dấu chấm ảo để biểu diễn một số thực. Dấu chấm ảo được sử dụng trong từ dữ liệu dùng để phân biệt và ngăn cách giữa phần biểu diễn giá trị nguyên của dữ liệu và một phần lẻ thập phân. Ví dụ về một từ 8-bit biểu diễn số dấu phảy động được chỉ ra như trong Hình 3-1. Với cách biểu diễn này, giá trị thực của số được tính như sau:

$$N = a_4 2^4 + a_3 2^3 + a_2 2^2 + a_1 2^1 + a_0 2^0 + a_{-1} 2^{-1} + a_{-2} 2^{-2} + a_{-3} 2^{-3}$$
$$= 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 + 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3}$$
$$= 8 + 2 + 1 + 1/2 + 1/8$$
$$= 11.625$$



Hình 3-1: Định dạng biểu diễn số dấu phảy tĩnh 8 bit

Nhược điểm của phương pháp biểu diễn số dấu phảy tĩnh là vùng biểu diễn số nguyên bị hạn chế bởi dấu phảy tĩnh được gán cố định. Điều này dễ xảy ra hiện tượng tràn số khi thực hiện các phép nhân hai số lớn.

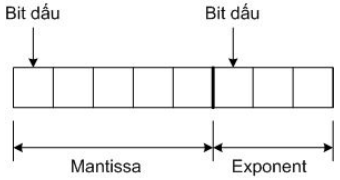
3.2.4 Số dấu phảy động

Phương pháp biểu diễn số chính xác và linh hoạt được sử dụng rộng rãi hiện nay là hệ thống biểu diễn số dấu phảy động. Đây cũng là một phương pháp biểu diễn số khoa học bao gồm 2 phần: phần biểu diễn lưu trữ số *mantissa* và một phần lưu trữ biểu diễn số *exponent*. Ví dụ trong hệ cơ số thập phân, một số nguyên bằng 5 có thể được biểu diễn hoặc là $0.5 \cdot 10^1$, $50 \cdot 10^{-1}$, hoặc $0.05 \cdot 10^2$, …Trong máy tính số hoặc hệ thống số nói chung, các số dấu phảy động nhị phân thường được biểu diễn dạng

$$N = M \cdot 2^E \tag{1.2}$$

Trong đó, M là phần giá trị số *mantissa*, E là phần lũy thừa của số N. M thường là các giá trị lẻ mà phần thập phân của nó thường nằm trong khoảng $0.5 \le M \le 1$.

Hình 3-2 mô tả biểu diễn một số dấu phảy động của từ 8 bit gồm 5 bit biểu diễn phần số có nghĩa *mantissa*, và 3 bit biểu diễn phần lũy thừa. Vì các phần *mantissa* và lũy thừa đều có thể nhận các giá trị âm vì vậy các bit đầu tiên của các phần giá trị đó đều có thể được sử dụng để biểu diễn dấu khi cần thiết.



Hình 3-2: Biểu diễn dấu phảy động 8 bít

Trong một số VXL, VĐK do độ rộng từ nhị phân nhỏ nên có thể sử dụng 2 từ để biểu diễn một số dấu phảy động. Một từ sẽ dùng để biểu diễn giá trị *mantissa*, và một phần biểu diễn giá trị *exponent*.

Nếu phần *mantissa* được chuẩn hóa thành một số lẻ có giá trị trong khoảng $0.5 \le M \le 1$ thì bit đầu tiên sau bit dấu thường là một và sẽ có một dấu phảy nhị phân ẩn ngay sau bit dấu.

Phần biểu diễn *exponent* E sẽ quyết định vị trí của dấu phảy động sẽ dịch sang trái (E>0) hay sang phải (E<0) bao nhiêu vị trí. Ví dụ biểu diễn một số thập phân 6.5 bằng một từ 8 bit dấu phảy động như sau:

$$N = .1101 \cdot 2^{11_2}$$
$$= \left[\frac{1}{2} + \frac{1}{4} + \frac{1}{16}\right] 2^3 = 6.5$$

Trong trường hợp này phần *mantissa* gồm 4 bit và phần *exponent* gồm 3 bit. Nếu ta dịch dấu phảy sang phải 3 vị trí bit thì chúng ta sẽ có một số nhị phân dấu phảy động biểu diễn được sẽ là 110.1.

Tổng quát hóa trong trường hợp một số nhị phân dấu phảy động *n* bit gồm *m* bit biểu diễn phần *mantissa* và *e* bit biểu diễn phần *exponent* thì giá trị của số lớn nhất có thể biểu diễn được sẽ là

$$N_{\max} = (1 - 2^{-m+1}) 2^{(2^{e-1}-1)} \tag{1.3}$$

Và số dương nhỏ nhất có thể biểu diễn là

$$N_{\min} = 0.5 \cdot 2^{-(2^{e-1}-1)} \tag{1.4}$$

Theo tiêu chuẩn IEEE 754 và 854 có 2 định dạng chính cho số dấu phảy động là số thực dài (*long*) và số thực ngắn (*short*) chúng khác nhau về dải biểu diễn và độ lớn lưu trữ yêu cầu. Theo chuẩn này, số thực dài được định dạng 8 byte bao gồm 1 bit dấu, 11 bit *exponent* và 53 bit lưu giá trị số có nghĩa. Một số thực ngắn được định dạng 4 byte bao gồm 1 bit dấu, 8 bit lũy thừa và 24 bit lưu giá trị số có nghĩa. Một số thực ngắn có thể biểu diễn và xử lý được số có giá trị nằm trong dải 10^{38} to 10^{-38} và số thực dài có thể biểu diễn và xử lý được số có giá trị thuộc dải 10^{308} to 10^{-308}. Để biểu diễn một giá trị tương đương như vậy bằng số dấu phảy tĩnh thì cần tới 256 bit hay 32 byte dữ liệu.

3.2.5 Một số phép tính cơ bản

- **Thực hiện phép nhân**

Vì trong các VĐK nhúng thường không hỗ trợ các phép nhân nhiều byte. Công việc này phải được thực hiện bởi người phát triển chương trình và thể hiện dưới dạng một thuật toán dựa trên các phép toán có sẵn áp dụng cho số nhị phân là cộng/trừ và dịch. Để có một sự hiểu biết rõ ràng hơn về thuật toán thực hiện phép nhân, chúng ta xét một ví dụ về một phép tính nhân hai số nhị phân tổng quát như sau:

$$\begin{array}{r} A = a_n \cdot 2^n + \cdots + a_1 \cdot 2^1 + a_0 \cdot 2^0 \\ B = b_n \cdot 2^n + \cdots + b_1 \cdot 2^1 + b_0 \cdot 2^0 \\ \hline b_n \cdot (A) \cdot 2^n + \cdots + b_1 \cdot (A) \cdot 2^1 + b_0 \cdot (A) \cdot 2^0 \end{array}$$

Nguyên lý thực hiện phép nhân cũng giống như ta thực hiện phép nhân hai đa thức. Trong trường hợp nhân hai số nhị phân thì mỗi phần tử là một bit, byte hoặc từ. Ví dụ cụ thể với hai số nhị phân 4 bit ta thu được phép nhân thực hiện như sau:

$$\begin{array}{r} a_3 \cdot 2^3 + a_2 \cdot 2^2 + a_1 \cdot 2^1 + a_0 \cdot 2^0 \\ b_3 \cdot 2^3 + b_2 \cdot 2^2 + b_1 \cdot 2^1 + b_0 \cdot 2^0 \\ \hline a_3 \cdot b_0 \cdot 2^3 + a_2 \cdot b_0 \cdot 2^2 + a_1 \cdot b_0 \cdot 2^1 + a_0 \cdot b_0 \cdot 2^0 \\ a_3 \cdot b_1 \cdot 2^4 + a_2 \cdot b_1 \cdot 2^3 + a_1 \cdot b_1 \cdot 2^2 + a_0 \cdot b_1 \cdot 2^1 \\ a_3 \cdot b_2 \cdot 2^5 + a_2 \cdot b_2 \cdot 2^4 + a_1 \cdot b_2 \cdot 2^3 + a_0 \cdot b_2 \cdot 2^2 \\ a_3 \cdot b_3 \cdot 2^6 + a_2 \cdot b_3 \cdot 2^5 + a_1 \cdot b_3 \cdot 2^4 + a_0 \cdot b_3 \cdot 2^3 \end{array}$$

Thuật toán thực hiện phép nhân 32 bit theo trình tự sau:

(1) Cấp phát vùng nhớ đủ lớn để lưu số được nhân 32 bit và có thể thực hiện phép dịch trái 32 lần. Đặt giá trị khởi tạo cho bộ đếm bit bằng 32 và xóa thanh ghi hay biến lưu giữ kết quả phép nhân. (Chú ý: Số lượng bit cần để lưu giá trị kết quả phải bằng tổng số lượng bit cần để lưu các số hạng phép nhân)

(2) Dịch số nhân sang phải một vị trí bit và kiểm tra cờ nhớ. Nếu không có cờ nhớ thì tiếp tục thực hiện bước 3. Nếu xuất hiện cờ nhớ thì cộng thêm vào biến lưu kết quả hiện tại của phép nhân một giá trị bằng giá trị của số được nhân.

(3) Dịch số được nhân sang trái một vị trí bit và giảm bộ đếm dịch đi một. Kiểm tra xem giá trị của bộ đếm dịch có bằng không không? Nếu bằng không thì thực hiện tiếp bước 4, còn không thì quay trở lại thực hiện bước 2.
(4) Kết quả cuối cùng của phép nhân được lưu trong thanh ghi biến kết quả.

Ví dụ phép nhân từ nhị phân 4 bit 1100 x 1101

0.	**A**	**1100 (12)**
	B	1101 (13)
	Counter	100 (4)
	Product	0
1.	**A**	**11000 (24)**
	B	0110 (6)
	Counter	011 (3)
	Product	1100 (12)
2.	**A**	**110000 (48)**
	B	0011 (3)
	Counter	010 (2)
	Product	1100 (12)
3.	**A**	**1100000 (96)**
	B	0001 (1)
	Counter	001 (1)
	Product	111100 (60)
4.	**A**	**11000000 (192)**
	B	0001 (1)
	Counter	000 (0)
	Product	10011100 (156)

Thực thi thuật toán thực hiện phép nhân số nguyên không dấu bằng ngôn ngữ C/C++:

```
long product = 0;
while (multipier != 0){
      if (multiplier & 1){
          product += multiplicand;
      }
      multiplier >> =1;
      multiplicand <<= 1;
}
```

- **Thực hiện phép chia**

Phép chia có thể được thực hiện bằng cách chuyển đổi thành phép nhân và phép dịch. Ví dụ muốn thực hiện phép chia 5 trong hệ thập phân chúng ta có thể thực hiện bởi một phép nhân 2 và dịch dấu phảy của kết quả thu được sang trái một đơn vị. Một cách tổng quát có thể thực hiện chuyển đổi một phép chia tương đương như sau:

$$\frac{x}{a} = \frac{n}{a}\frac{x}{n}$$

Đối với phép chia nhị phân thì n sẽ được chọn là một số lũy thừa của 2 và phải lớn hơn a.

Thuật toán thực hiện phép chia có thể được thực thi bởi phép dịch, cộng và trừ như sau:
(1) Nạp biến lưu giá trị thương số bằng giá trị của số bị chia; số bước dịch cần thực hiện bằng số bit lưu số bị chia.
(2) Dịch trái biến lưu giá trị thương số vào phần biến lưu giá trị dư của phép chia.
(3) So sánh số dư với số chia. Nếu số dư lớn hơn hoặc bằng số chia thì thực hiện phép trừ số dư đi một giá trị bằng giá trị số chia. Nếu không thì chuyển sang thực hiện bước tiếp theo.
(4) Giảm biến lưu giá trị số lần lặp và kiểm tra xem nó đã bằng không chưa. Nếu chưa bằng không thì quay trở lại bước 2 thực hiện tiếp, còn nếu bằng không thì giá trị của phép chia được lưu trong ô nhớ chứa số dư và thương số.

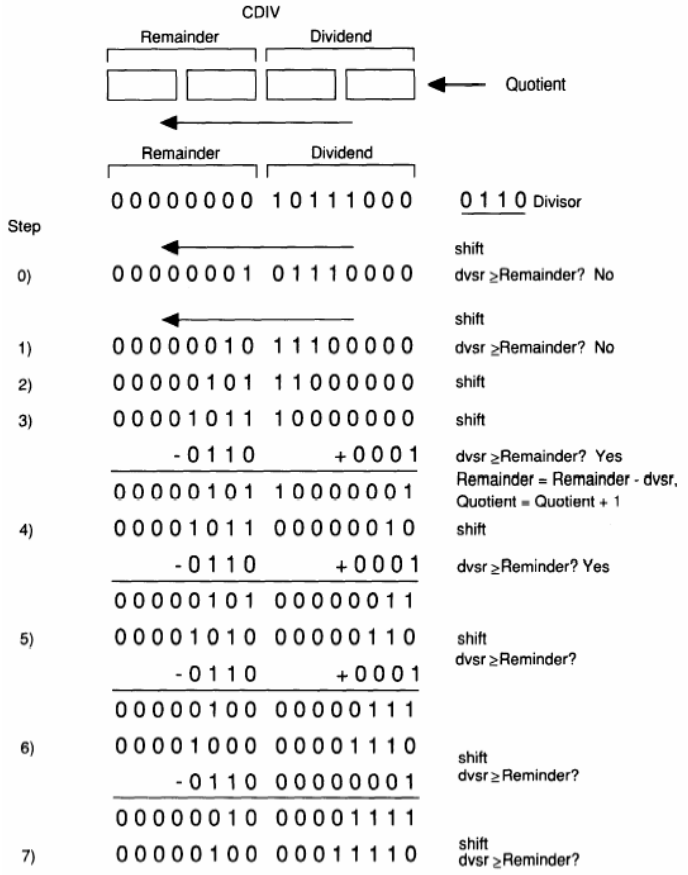
Thực thi thuật toán bằng ngôn ngữ C/C++

```
i = 0;  quotient = 0;
if (divisor == 0) goto  error;
while (dividend > divisor) divisor <<= 1; i++;
divisor >>= 1;
while (i != 0){
      quotient <<= 1;
      if (divisor < dividend ) dividend -= divisor;
      quotient ++;
      divisor >>=1 ,
      i--;
}
```

Trước khi thực hiện phép chia yêu cầu cần phải kiểm tra lỗi chia không có thể xảy ra. Thuật toán thực hiện phép chia chủ yếu dựa trên phép dịch và phép trừ. Số bị chia sẽ dịch sang trái và lưu vào một biến, phần dư sẽ được so sánh với số chia. Nếu phần dư bằng hoặc lớn hơn số chia thì phần dư sẽ được trừ đi một giá trị bằng số chia và số bị chia sẽ được cộng thêm một và dịch sang trái một vị trí bit và đó chính được gọi là thương số. Quá trình này được lặp lại và tiếp tục cho đến khi số lần dịch bằng đúng số bit của từ lưu số bị chia.

Các biến được sử dụng trong quá trình thực hiện phép chia bao gồm 5 biến số: số bị chia, số chia, thương số, số dư và số lần dịch. Trong quá trình thực hiện thì số bị chia, thương số, và số dư cùng chia sẻ chung một vùng ô nhớ. Số dư và số bị chia sẽ thuộc cùng một từ lớn. Số bị chia nằm trong phần từ trọng số thấp và số dư sẽ nằm trong phần từ trọng số cao. Sau khi thực hiện xong phép chia thì số bị chia sẽ được dịch toàn bộ sang trái vào phần biến số dư và được thay thế bằng thương số. Kết quả còn lại thu

được chỉ còn là số dư và thương số. Hình ảnh về bộ nhớ lưu các biến số thực hiện trong thuật toán này được minh hoạ như trong Hình 3-3.



Hình 3-3: Thực hiện phép chia

3.3 Tập lệnh

3.3.1 Cấu trúc tập lệnh CISC và RISC

Hầu hết các vi điều khiển và VXL nhúng có cấu trúc được phát triển dựa theo kiến trúc máy tính tập lệnh phức hợp CISC (*Complex Instruction Set Computer*). CISC là một cấu trúc xử lý các lệnh phức hợp, tức là một lệnh phức hợp sẽ bao gồm một vài lệnh đơn. Theo nguyên lý này có thể giảm bớt được thời gian dùng để truy nhập và đọc mã chương trình từ bộ nhớ. Điều này rất có ý nghĩa với các kiến trúc thiết kế xử lý tính toán theo kiểu tuần tự. Lý do cho sự ra đời của tập lệnh phức hợp nhằm giảm thiểu dung lượng bộ nhớ cần thiết để lưu giữ chương trình thực hiện, và sẽ giảm được giá thành về

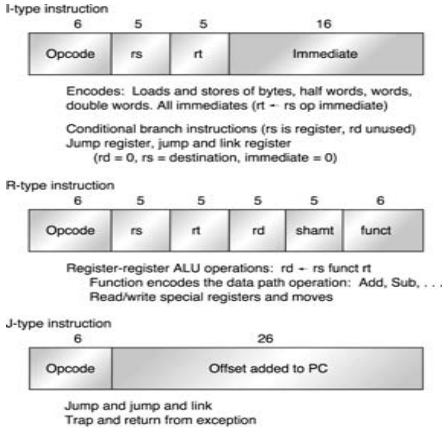
bộ nhớ cần cung cấp cho CPU. Các lệnh càng gọn và phực hợp thì sẽ cần càng ít không gian bộ nhớ chương trình. Kiến trúc tập lệnh phức hợp sử dụng các lệnh với độ dài biến đổi tuỳ thuộc vào độ phức hợp của các lệnh từ đơn giản đến phức tạp. Trong đó sẽ có một số lượng lớn các lệnh có thể truy nhập trực tiếp bộ nhớ. Vì vậy với kiến trúc tập lệnh phức hợp chúng ta sẽ có được một tập lệnh đa dạng phức hợp, gọn, với độ dài lệnh thay đổi và dẫn đến chu kỳ thực hiện lệnh cũng thay đổi tuỳ theo độ phức hợp trong từng lệnh. Một vài lệnh phức hợp, đặc biệt là các lệnh truy nhập bộ nhớ cần tới vài chục chu kỳ để thực hiện. Trong một số trường hợp các nhà thiết kế VXL thấy rằng cần phải giảm chu kỳ nhịp lệnh để có đủ thời gian cho các lệnh hoàn thành điều này cũng dẫn đến thời gian thực hiện bị kéo dài hơn.

Một số VĐK được phát triển theo kiến trúc máy tính tập lệnh rút gọn RISC (*Reduced Instruction Set Computer*). RISC phù hợp với các thiết kế kiến trúc xử lý các lệnh đơn. Thuật ngữ "rút gọn" (*reduced*) đôi khi bị hiểu không thật chính xác theo nghĩa đen của nó thực chất ý tưởng gốc xuất phát từ khả năng cung cấp một tập lệnh tối thiểu để thực hiện tất cả các hoạt động chính như: chuyển dữ liệu, các hoạt động ALU và rẽ nhánh điều khiển chương trình. Chỉ có các lệnh nạp (*load*), lữu trữ (*store*) là được phép truy nhập trực tiếp bộ nhớ.

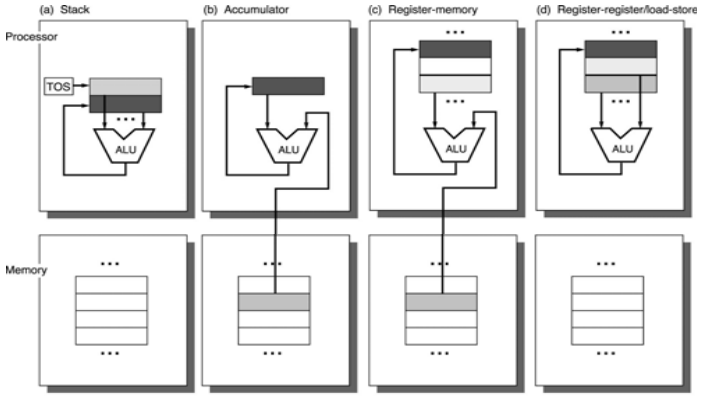
B- 1: So sánh đặc điểm của CISC và RISC

CISC	RISC
Bất kỳ lệnh nào cũng có thể tham chiếu tới bộ nhớ	Chỉ có các lệnh Nạp (Load) hoặc Lưu trữ (Store) là có thể tham chiếu tới bộ nhớ
Tồn tại nhiều lệnh và kiểu địa chỉ	Tồn tại ít lệnh và kiểu địa chỉ
Khuôn dạng lệnh đa dạng	Khuôn dạng lệnh cố định
Chỉ có một tập thanh ghi	Có nhiều tập thanh ghi
Các lệnh thực hiện trong nhiều nhịp chu kỳ	Các lệnh thực hiện trong một nhịp chu kỳ
Có một chương trình nhỏ để thông dịch lệnh	Lệnh được thực hiện trực tiếp ngay bởi phần cứng
Chương trình thông dịch lệnh phức tạp	Chương trình biên dịch mã nguồn phức tạp
Không hỗ trợ cơ chế pipeline	Hỗ trợ cơ chế pipeline
Kích thước mã chương trình nhỏ gọn	Kích thước mã chương trình lớn

3.3.2 Định dạng lệnh



Hình 3-4: Định dạng lệnh MIPS



Hình 3-5: Phân loại các phép thực thi lệnh

3.3.3 Các kiểu truyền địa chỉ toán tử lệnh

Các kiểu đánh/truyền địa chỉ cho phép chúng ta chỉ ra/truyền toán tử tham gia trong các lệnh thực thi. Kiểu địa chỉ có thể chỉ ra là một hằng số, môt thanh ghi hoặc một khu vực cụ thể trong bộ nhớ. Một số kiểu đánh địa chỉ cho phép sử dụng địa chỉ ngắn và một số loại thì cho phép chúng ta xác định khu vực chứa toán tử lệnh, và thường được

gọi là địa chỉ hiệu dụng của toán tử và thường là động. Chúng ta sẽ xét một số loại hình đánh địa chỉ cơ bản hiện đang được sử dụng rộng rãi trong cơ chế thực hiện lệnh.

❒ **Đánh địa chỉ tức thì** (*Immediate Addressing*)
Phương pháp này cho phép truyền giá trị toán tử lệnh một cách tức thì như một phần của câu lệnh được thực thi. Ví dụ nếu sử dụng kiểu đánh địa chỉ tức thời cho câu lệnh Load 0x0008 thì giá trị 0x0008 sẽ được nạp ngay vào AC. Trường bit thường dùng để chứa toán tử lệnh sẽ chứa giá trị thực của toán tử chứ không phải địa chỉ của toán tử cần truyền cho lệnh thực thi. Kiểu địa chỉ tức thời cho phép thực thi lệnh rất nhanh vì không phải thực hiện truy xuất bộ nhớ để nạp giá trị toán tử mà giá trị toán tử đã được gộp như một phần trong câu lệnh và có thể thực thi ngay. Vì toán tử tham gia như một phần cố định của chương trình vì vậy kiểu đánh địa chỉ này chỉ phù hợp với các toán tử hằng và biết trước tại thời điểm thực hiện chương trình, hay đã xác định tại thời điểm biên dịch chương trình.

❒ **Đánh địa chỉ trực tiếp** (*Direct Addressing*)
Phương pháp này cho phép truyền toán tử lệnh thông qua địa chỉ trực tiếp chứa toán tử đó trong bộ nhớ. Ví dụ nếu sử dụng cơ chế đánh địa chỉ toán tử trực tiếp thì trong câu lệnh Load 0x0008 sẽ được hiểu là dữ liệu hay toán tử được nạp trong câu lệnh này nằm trong bộ nhớ tại địa chỉ 0x0008. Cơ chế đánh địa chỉ trực tiếp cũng thuộc loại hình khá nhanh mặc dù không nhanh được như cơ chế truyền địa chỉ tức thời nhưng độ mềm dẻo cao hơn vì địa chỉ của toán tử không nằm trong phần mã lệnh và giá trị có thể thay đổi trong quá trình thực thi chương trình.

❒ **Đánh địa chỉ thanh ghi** (*Register Addressing*)
Trong cách đánh địa chỉ và truyền toán tử này thì toán tử không nằm trong bộ nhớ như trường hợp đánh địa chỉ trực tiếp mà nằm tại chính trong thanh ghi. Khi toán tử đã được nạp vào thanh ghi thì việc thực hiện có thể rất nhanh vì tốc độ truy xuất thanh ghi nhanh hơn so với bộ nhớ. Nhưng số lượng thanh ghi chỉ có hạn và phải được chia sẻ trong quá trình thực hiện chương trình vì vậy các toán tử phải được nạp vào thanh ghi trước khi nó được thực thi.

❒ **Đánh địa chỉ gián tiếp** (*Indirect Addressing*)
Trong phương pháp truyền toán tử này, trường toán tử trong câu lệnh được sử dụng để tham chiếu tới một con trỏ nằm trong bộ nhớ để trỏ tới địa chỉ hiệu dụng của toán tử. Cơ chế truyền này có thể nói là mềm dẻo nhất so với các cơ chế truyền địa chỉ khác trong quá trình thực thi chương trình. Ví dụ nếu áp dụng cơ chế truyền địa chỉ gián tiếp trong câu lệnh Load 0x0008 thì sẽ được hiểu là giá trị dữ liệu có địa chỉ tại 0x0008 thực chất là chứa địa chỉ hiệu dụng của toán tử cần truyền cho câu lệnh. Giả thiết tại vị trí ô nhớ 0x0008 đang chứa giá trị 0x02A0 thì 0x02A0 chính là giá trị thực của toán tử sẽ được nạp vào AC. Một biến thể khác cũng có thể thực hiện theo cơ chế này là truyền tham chiếu tới con trỏ nằm trong khu vực thanh ghi. Cơ chế này còn được biết tới với tên gọi là đánh địa chỉ gián tiếp thanh ghi. Ví dụ một câu lệnh Load R1 sử dụng cơ chế

truyền địa chỉ gián tiếp thanh ghi thì chúng ta có thể dễ dàng thông dịch được toán tử truyền trong câu lệnh này có địa chỉ hiệu dụng nằm trong thanh ghi R1.

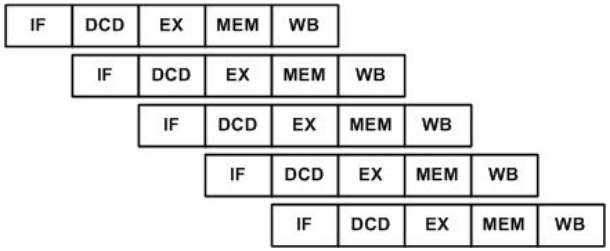
❒ **Cách đánh địa chỉ cơ sở và chỉ số** (*Indexed and Based Addressing*)
Trong cơ chế này người ta sử dụng một thanh ghi để chứa *offset* (độ chênh lệch tương đối) mà sẽ được cộng với toán tử để tạo ra một địa chỉ hiệu dụng. Ví dụ, nếu toán tử X của lệnh Load X được đánh địa chỉ theo cơ chế địa chỉ chỉ số và thanh ghi R1 là thanh ghi chứa chỉ số và có giá trị là 1 thì địa chỉ hiệu dụng của toán tử thực chất sẽ là X+1. Cơ chế đánh địa chỉ cơ sở cũng giống như vậy ngoại trừ một điều là thay vì sử dụng thanh ghi địa chỉ offset thì ở đây sử dụng thanh ghi địa chỉ cơ sở. Về mặt lý thuyết sự khác nhau giữa hai cơ chế tham chiếu địa chỉ này là chúng được sử dụng thế nào chứ không phải các toán tử được tính toán thể nào. Một thanh ghi chỉ số sẽ lưu chỉ số mà sẽ được sử dụng như một offset so với địa chỉ được đưa ra trong trường địa chỉ của lệnh thực thi. Thanh ghi cơ sở lưu một địa chỉ cơ sở và trường địa chỉ trong câu lệnh thực thi sẽ lưu giá trị dịch chuyển từ địa chỉ này. Hai cơ chế tham chiếu địa chỉ này rất hữu ích trong việc truy xuất với các phần tử kiểu mảng. Tuỳ thuộc vào thiết kế tập lệnh các thanh ghi mục đích chung thường hay được sử dụng trong cơ chế đánh địa chỉ này.

❒ **Đánh địa chỉ ngăn xếp** (*Stack Addressing*)
Trong cơ chế truyền địa chỉ này thì toán tử nhận được từ đỉnh ngăn xếp. Thay vì sử dụng thanh ghi mục đích chung hay ô nhớ kiến trúc dựa trên ngăn xếp lưu các toán tử trên đỉnh của ngăn xếp, và có thể truy xuất với CPU. Kiến trúc này không chỉ hiệu quả trong việc lưu giữ các giá trị trung gian trong các phép tính phức tạp mà còn cung cấp một phương pháp hiệu quả trong việc truyền các tham số trong các lời gọi hàm cũng như để lưu cất các cấu trúc dữ liệu cục bộ và định nghĩa ra phạm vi tồn tại của các biến và các hàm con. Trong các cấu trúc lệnh truyền toán tử dựa trên ngăn xếp, hầu hết các lệnh chỉ bao gồm phần mã, tuy nhiên cũng có một số lệnh đặc biệt chỉ có một toán tử ví dụ như lệnh cất vào (*push*) hoặc lấy ra (*pop*) từ ngăn xếp. Chỉ có một số lệnh yêu cầu hai toán tử thì hai giá trị chứa trong 2 ô nhớ trên đỉnh ngăn xếp sẽ được sử dụng. Ví dụ như lệnh Add, CPU lấy ra khỏi ngăn xếp hai phần tử nằm trên đỉnh rồi thực hiện phép cộng và sau đó lưu kết quả trở lại đỉnh ngăn xếp.

❒ **Các cách đánh địa chỉ khác**
Có rất nhiều biến thể tạo bởi các cơ chế đánh địa chỉ giới thiệu ở trên. Đó là sự tổ hợp trong việc tạo ra hoặc xác định địa chỉ hiệu dụng của toán tử truyền cho lệnh thực thi. Ví dụ như cơ chế đánh địa chỉ chỉ số gián tiếp sử dụng đồng thời cả hai cơ chế đánh địa chỉ đồng thời, tương tự như vậy cũng có cơ chế đánh địa chỉ cơ sở/*offset*…Cũng có một số cơ chế tự động tăng hoặc giảm thanh ghi sử dụng trong lệnh đang thực thi nhờ vậy mà có thể giảm được độ lớn của mã chương trình đặc biệt phù hợp cho các ứng dụng Nhúng.

3.3.4 Nguyên lý thực hiện *pipeline*

Vi xử lý có thể thực thi các lệnh với một tốc độ rất nhanh. RISC sử dụng kỹ thuật *pipeline* để tăng cường tốc độ xử lý các lệnh đồng thời nhờ vào khả năng thực hiện xếp chồng cuốn chiếu liên tục các lệnh theo các phân đoạn thực hiện lệnh. Ví dụ một lệnh có thể được đọc từ bộ nhớ trong khi một lệnh khác đang được giải mã để chuẩn bị đưa vào xử lý và một lệnh khác thì đang được thực hiện. Cũng có một số VĐK có tên gọi là máy tính tập lệnh đặc biệt SISC (*Specific Instruction Set Computer*) vì chúng được phát triển dựa trên tập lệnh được thiết kế đặc chủng cho mục đích điều khiển.

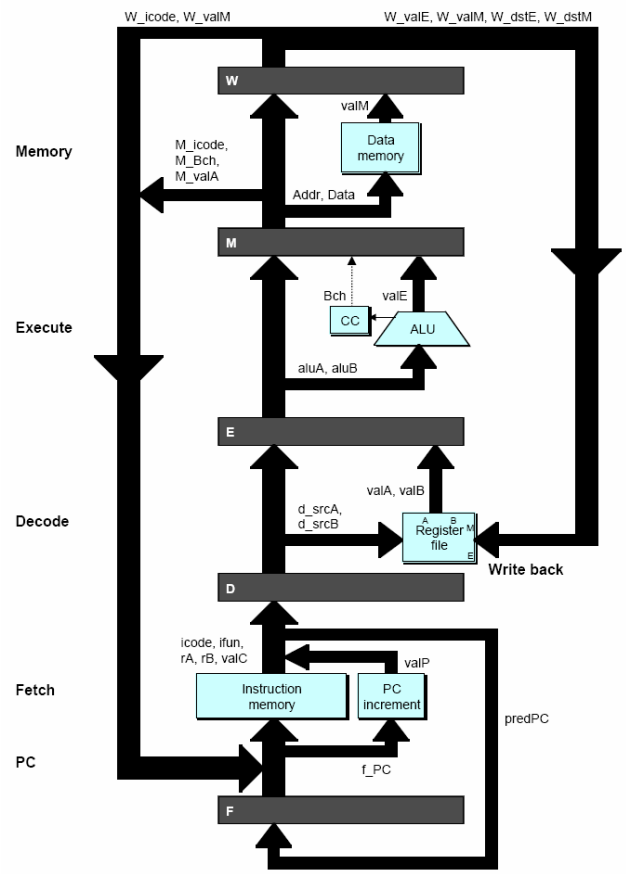


Hình 3-6: Nguyên lý thực hiện pipeline

Pipeline được thực hiện dựa trên nguyên lý xếp chồng cuốn chiếu các phân đoạn trong mỗi một lệnh. Thông thường mỗi một lệnh được chia ra làm nhiều phân đoạn thực hiện, phổ biến hiện nay là 5 phân đoạn tuần tự như sau:

(1) **Trỏ lệnh** (*Instruction Fetch*): Thực hiện trỏ tới lệnh thực hiện bằng cách đọc địa chỉ lệnh từ thanh ghi con trỏ lệnh (PC), đọc lệnh đó ra từ bộ nhớ chương trình và tính toán rồi nạp giá trị mới vào trong thanh ghi con trỏ lệnh để trỏ tới lệnh sẽ thực thi tiếp theo.
(2) **Giải mã lệnh** (*Decode*): Thực hiện thông dịch và chuyển đổi mã lệnh thành dạng mã để ALU có thể hiểu và chuẩn bị thực thi. Quá trình này thực chất là quá trình đọc và chuyển đổi nội dung trong các thanh ghi chương trình.
(3) **Thực hiện lệnh** (*Execute*): ALU thực thi lệnh vừa được giải mã.
(4) **Truy nhập bộ nhớ dữ liệu** (*Memory*): Đọc ra hoặc viết vào bộ nhớ dữ liệu nếu lệnh thực hiện có nhu cầu này.
(5) **Viết trở lại** (*Write back*): Hoàn thành và cập nhật nội dung các thanh ghi.

Chúng ta cần phân biệt cơ chế *pipeline* và cơ chế thực hiện song song mặc dù cả hai đều nhằm đáp ứng yêu cầu thực thi cạnh tranh và tăng tốc độ thực thi. Cơ chế *pipeline* giải quyết vấn đề cạnh tranh và tăng tốc độ thực hiện bằng cách chia nhỏ tính toán thành các bước nhỏ trong khi đó cơ chế song song sẽ sử dụng nhiều nguồn tài nguyên độc lập để thực hiện.



Hình 3-7: Quá trình thực hiện lệnh theo nguyên lý pipeline

3.3.5 Harzard

Trong cơ chế thực hiện lệnh *pipeline* thể hiện rõ được ưu điểm trong việc thúc đẩy hiệu suất thực hiện lệnh, tuy nhiên có thể xảy ra hiện tượng thực thi sai do sự thiếu đồng bộ và phụ thuộc lẫn nhau giữa các lệnh trong nhóm thực thi *pipeline*.

- **Hazard dữ liệu**

Hiện tượng harzard xảy ra khi có sự phụ thuộc lẫn giữa các lệnh nằm trong khoảng xếp chồng thực hiện cuốn chiếu theo nguyên lý *pipeline*. Điều này có thể dễ dàng hình dung khi hai hoặc nhiều lệnh thực hiện xếp chồng khi có nhu cầu đọc giá trị của cùng một toán tử. Do sự phụ thuộc như vậy nên khi viết chương trình chúng ta phải kiểm soát được thứ tự chương trình mà các lệnh sẽ được thực hiện như thế nào. Mục đích của việc thực thi là làm sao để hỗ trợ được cơ chế thực hiện song song và tăng được hiệu suất

thực thi chương trình. Việc phát hiện và tránh được hiện tượng *hazard* là cần thiết để đảm bảo chương trình được thực thi đúng. Tuỳ theo nguyên nhân gây ra *hazard* người ta phân ra 3 loại hình chính tuỳ thuộc vào thứ tự đọc hoặc viết truy nhập lệnh của các nhóm lệnh phụ thuộc nhau trong cơ chế thực hiện song song.

Xét hai lệnh i và j trong đó lệnh i được thực hiện trước lệnh j trong chương trình. Hiện tượng Hazard dữ liệu có thể xảy ra như sau:

- RAW (*read after write*): Đọc sau khi viết

Khi lệnh i và j đều cần sử dụng và trao đổi thông tin với cùng một giá trị ô nhớ, trong đó lệnh i cần phải thực hiện xong và cập nhật giá trị vào ô nhớ đó rồi lệnh j mới có thể đọc và sử dụng. Nếu lệnh i chưa thực hiện xong mà lệnh j đã đọc giá trị ô nhớ đó thì sẽ xảy ra hiện tượng được gọi là hazard dữ liệu. Lệnh j đọc thông tin từ một ô nhớ trước khi lệnh i kịp viết vào vì vậy lệnh j sẽ chỉ đọc được giá trị cũ chứ không phải giá trị mới cần phải sử dụng. Trong cơ chế thực hiện *pipeline* 5 phân đoạn sẽ gặp phải hiện tượng hzard dữ liệu khi có một lệnh nạp (load) theo sau một lệnh ALU số nguyên và sử dụng trực tiếp kết quả nạp.

- WAW (*write after write*): Viết sau khi viết

Lệnh j viết vào một toán tử trước khi lệnh i viết vào. Mà yêu cầu thực thi đúng chương trình là lệnh i phải viết trước lệnh j và giá trị cuối cùng lưu trong toán tử phải do lệnh j đưa ra chứ không phải lệnh i. Hiện tượng này được gọi là *hazard* dữ liệu khi có sự phụ thuộc đầu ra và nhiều lệnh cùng có nhu cầu truy nhập viết vào cùng một biến hay một ô nhớ.

- WAR (*write after read*): Viết sau khi đọc

j viết vào toán tử đích trước khi nó được đọc bởi lệnh i do đó lệnh I sẽ nhận được giá trị sai. Hiện tượng Hazard này xuất hiện khi có sự phụ thuộc toán hạng trong các phép tính

- **Hazard do sự phụ thuộc điều khiển**

Kiểu phụ thuộc cũng khá phổ biến là do cấu trúc điều khiển. Sự phụ thuộc điều khiển được quyết định trình tự thực thi của một lệnh i theo lệnh rẽ nhánh đảm bảo sao cho nó được thực thi đúng như thứ tự mong muốn. Tất cả các lệnh ngoại trừ khối cơ bản đầu tiên của chương trình đều được điều khiển theo cấu trúc lệnh rẽ nhánh và phải được đảm bảo để thực thi đúng theo thứ tự. Một ví dụ đơn giản nhất về sự phụ thuộc điều khiển là sự phụ thuộc điều khiển theo cấu trúc *if…then…*Phần thực thi trong phần "*then*" sẽ phụ thuộc câu lệnh điều kiện *if*. Ví dụ đoạn mã chương trình minh họa như sau:

```
if (p1) {
   S1;
}
if (p2) {
   S2;
}
```

Câu lệnh được điều khiển phụ thuộc vào p1 và S2 được điều khiển phụ thuộc p2 chứ không phải p1.

Nói chung, có 2 ràng buộc có thể giả thiết trong sự phụ thuộc điều khiển:
(1) Một lệnh thực hiện phụ được quyết định bởi một lệnh điều khiển rẽ nhánh thì không thể được phép chuyển lên trước câu lệnh thực hiện kiểm tra điều kiện. Ví dụ chúng ta không thể đưa lệnh từ phần **`then`** lên trước phần **`if`**.
(2) Một lệnh thực hiện độc lập và không phụ thuộc vào lệnh rẽ nhánh không thể được chuyển vào khu vực sau phần thực hiện của nhánh thực hiện phụ thuộc. Ví dụ không thể đưa một lệnh lên trước phần lệnh **`if`** và chuyển nó vào trong phần **`then`**.

Sự phụ thuộc điều khiển phải được đảm bảo bởi 2 thuộc tính trong cơ chế *pipeline* đơn giản. Thứ nhất, các lệnh thực hiện trong chương trình phải đúng theo trình tự được điều khiển của nó. Trình tự này phải được đảm bảo rằng một lệnh mà phải thực thi trước một nhánh điều khiển thì phải thực hiện trước nhánh đó. Thứ hai, việc phát hiện ra sự xung đột về điều khiển (*control hazard*) sẽ đảm bảo rằng một lệnh mà được điều khiển phụ thuộc vào một nhánh thì không được thực hiện chừng nào hướng thực hiện của nhánh đó rõ ràng. Bảo đảm được sự phụ thuộc điều khiển là cần thiết và cũng là một cách đơn giản để đảm bảo đúng trình tự thực hiện chương trình. Sự phụ thuộc điều khiển không phải là một sự hạn chế cơ bản về khả năng thực thi chương trình. Chúng ta có thể sẵn sàng thực thi thêm những lệnh mà lẽ ra không nên được thực thi nếu chúng không gây ảnh hưởng gì đến tính đúng đắn của chương trình, nếu không sự xung đột gây ra bởi sự phụ thuộc điều khiển có thể xảy ra. Sự phụ thuộc về điều khiển không phải là một thuộc tính kịch tính bắt buộc phải bảo đảm. Thay vì điều đó, hai thuộc tính kịch tính cho việc lập trình một cách đúng đắn và thường được bảo đảm là phải tránh được xung đột bởi cả sự phụ thuộc về dữ liệu và điều khiển và đó chính là hành vi ngoại lệ có thể xảy ra trong luồng dữ liệu thực thi chương trình.

3.4 Ngôn ngữ và môi trường phát triển

3.4.1 Ngôn ngữ

Một trong những ngôn ngữ lập trình có lẽ phổ cập rộng rãi nhất hiện nay là ngôn ngữ C. So với bất kỳ ngôn ngữ lập trình nào khác đang tồn tại C thực sự phù hợp và trở thành một ngôn ngữ phát triển của hệ nhúng. Điều này không phải là cố hữu và sẽ tồn tại mãi, nhưng tại thời điểm này thì C có lẽ là một ngôn ngữ gần gũi nhất để trở thành một chuẩn ngôn ngữ trong thế giới hệ nhúng. Trong phần này chúng ta sẽ cùng tìm hiểu tại sao C lại trở thành một ngôn ngữ phổ biến đến vậy và tại sao chúng ta lựa chọn nó như một ngôn ngữ minh họa cho việc lập trình hệ nhúng.

Sự thành công về phát triển phần mềm thường là nhờ vào sự lựa chọn ngôn ngữ phù hợp nhất cho một dự án đặt ra. Cần phải tìm một ngôn ngữ để ***có thể đáp ứng được yêu cầu lập trình cho các bộ xử lý từ 8-bit đến 64-bit***, trong các hệ thống chỉ có hữu hạn về

bộ nhớ vài Kbyte hoặc Mbyte. Cho tới nay, điều này chỉ có C là thực sự có thể thỏa mãn và phù hợp nhất.

Rõ ràng C có một số ưu điểm nổi bật tiêu biểu như khá nhỏ và ***dễ dàng cho việc học, các chương trình biên dịch thường khá sẵn cho hầu hết các bộ xử lý đang sử dụng hiện nay***, và có rất ***nhiều người đã biết và làm chủ được ngôn ngữ này*** rồi, hay nói cách khác cũng ***đã được phổ cập từ lâu***. Hơn nữa C có lợi thế là ***không phụ thuộc vào bộ xử lý thực thi mã nguồn***. Người lập trình chỉ phải tập trung chủ yếu vào việc xây dựng thuật toán, ứng dụng và thể hiện bằng ngôn ngữ thân thiện thay vì phải tìm hiểu sâu về kiến thức phần cứng, cũng như rất nhiều các ưu điểm nổi bật khác của ngôn ngữ bậc cao nói chung.

Có lẽ một thế mạnh lớn nhất của ***C là một ngôn ngữ bậc cao mức thấp nhất***. Tức là với ngôn ngữ C chúng ta vẫn có thể điều khiển và truy nhập trực tiếp phần cứng khá thuận tiện mà không hề phải hy sinh hay đánh đổi bất kỳ một thế mạnh nào của ngôn ngữ bậc cao. Thực chất đây cũng là một trong những tiêu chí xây dựng của những người sáng lập ra ngôn ngữ C muốn hướng tới. Thực tế điều này đã được đề cập đến khi hai nhà sáng lập ra ngôn ngữ C, Kernighan và Ritchie đã đưa vào trong phần giới thiệu của cuốn sách của họ "*The C Programming Language*" như sau:

"C is a relatively "low level" language. This characterization is not pejorative; it simply means that C deals with the same sort of objects that most computers do. These may be combined and moved about with the arithmetic and logical operators implemented by real machines…"

Tất nhiên là C không phải là ngôn ngữ duy nhất cho các nhà lập trình nhúng. Ít nhất hiện nay người ta cũng có thể biết tới ngoài ngôn ngữ C là Assembly, C++, và Ada.

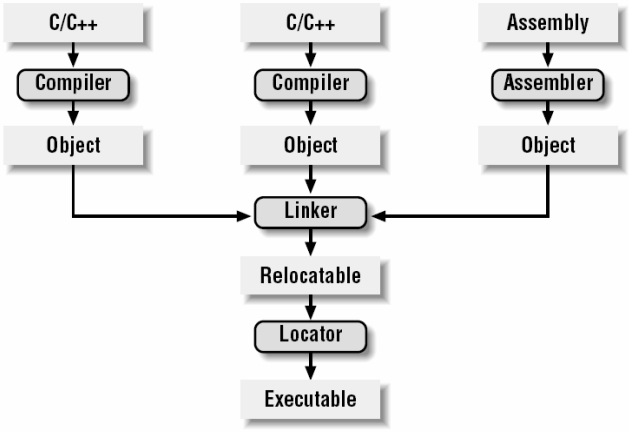
Trong những buổi đầu phát triển hệ nhúng thì ngôn ngữ Assembly chủ yếu được sử dụng cho các vi xử lý đích. Với ngôn ngữ này ***cho phép người lập trình điều khiển và kiểm soát hoàn toàn vi xử lý cũng như phần cứng hệ thống trong việc thực thi chương trình***. Tuy nhiên ngôn ngữ Assembly có nhiều nhược điểm mà cũng chính là lý do tại sao hiện nay nó ít được phổ cập và sử dụng. Đó là, ***việc học và sử dụng ngôn ngữ Assembly rất khó khăn và đặc biệt khó khăn trong việc phát triển các chương trình ứng dụng lớn phức tạp***. Hiên nay nó chỉ được sử dụng chủ yếu như điểm nối giữa ngôn ngữ bậc cao và bậc thấp và được sử dụng khi có yêu cầu đặc biệt về hiệu suất thực hiện và tối ưu về tốc độ mà không thể đạt được bằng ngôn ngữ khác. ***Ngôn ngữ Assembly chỉ thực sự phù hợp cho những người có kinh nghiệm và hiểu biết tốt về cấu trúc phần cứng đích*** cũng như nguyên lý thực hiện của bộ lệnh và chíp xử lý.

C++ là một ngôn ngữ kế thừa từ C để nhằm vào các lớp ứng dụng và tư duy lập trình hướng đối tượng và cũng bắt đầu chiếm được số lượng lớn quan tâm trong việc ứng dụng cho phát triển hệ nhúng. Tất cả các đặc điểm cốt lõi của C vẫn được kế thừa hoàn toàn trong ngôn ngữ C++ và ngoài ra còn ***hỗ trợ khả năng mới về trừu tượng hóa dữ liệu và phù hợp với tư duy lập trình hiện đại; hướng đối tượng***. Tuy nhiên điều này bị đánh

đổi bởi hiệu suất và thời gian thực thi do đó chỉ ***phù hợp với các dự án phát triển chương trình lớn và không chịu sức ép lớn về thời gian thực thi***.

Ada cũng là một ngôn ngữ hướng đối tượng mặc dù nó không được phổ cập rộng rãi như C++. Ada được xây dựng bởi cơ quan quốc phòng Mỹ để phục vụ phát triển các phần mềm quân sự chuyên dụng đặc biệt. Mặc dù cũng đã được chuẩn hóa quốc tế (Ada83 và Ada95) nhưng nó vẫn không được phổ cập rộng rãi ngoài việc ứng dụng chủ yếu trong các lĩnh vực quân sự và hàng không vũ trụ. Và nó cũng dần dần bị mất ưu thế và sự phổ cập trong thời gian gần đây, đây cũng là một điều đáng tiếc vì bản thân Ada cũng là một ngôn ngữ có nhiều đặc điểm phù hợp cho việc phát triển phần mềm hệ nhúng thay vì phải sử dụng C++.

3.4.2 Biên dịch



Hình 3-8: Quá trình phát triển và biên dịch phần mềm nhúng

- **Quá trình biên dịch (*Compiling*)**

Nhiệm vụ chính của bộ biên dịch là chuyển đổi chương trình được viết bằng ngôn ngữ thân thiện với con người ví dụ như C, C++,…thành tập mã lệnh tương đương có thể đọc và hiểu bởi bộ vi xử lý đích. Theo cách hiểu này thì bản chất một bộ hợp ngữ cũng là một bộ biên dịch để chuyển đổi một-một từ một dòng lệnh hợp ngữ thành một dạng mã lệnh tương đương cho bộ vi xử lý có thể hiểu và thực thi. Chính vì vậy đôi khi người ta vẫn nhầm hiểu giữa khái niệm bộ hợp ngữ và bộ biên dịch. Tuy nhiên việc biên dịch của bộ hợp ngữ sẽ được thực thi đơn giản hơn rất nhiều so với các bộ biên dịch cho các mã nguồn viết bằng ngôn ngữ bậc cao khác.

Mỗi một bộ xử lý thường có riêng ngôn ngữ máy vì vậy cần phải chọn lựa một bộ biên dịch phù hợp để có thể chuyển đổi chính xác thành dạng mã máy tương ứng với bộ xử lý đích. Đối với các hệ thống nhúng, bộ biên dịch là một chương trình ứng dụng luôn

được thực thi trên máy chủ (môi trường phát triển chương trình) và còn có tên gọi là bộ biên dịch chéo (*cross-compiler*). Vì bộ biên dịch chạy trên một nền phần cứng để tạo ra mã chương trình chạy trên môi trường phần cứng khác. Việc sử dụng bộ biên dịch chéo này là một thành phần không thể thiếu trong quá trình phát triển phần mềm cho hệ nhúng. Các bộ biên dịch chéo thường có thể cấu hình để thực thi việc chuyển đổi cho nhiều nền phần cứng khác nhau một cách linh hoạt. Và việc lựa chọn cấu hình biên dịch tương ứng với các nền phần cứng đôi khi cũng khá độc lập với chương trình ứng dụng của bộ biên dịch.

Kết quả đầu tiên của quá trình biên dịch nhận được là một dạng mã lệnh được biết tới với tên gọi là tệp đối tượng (*object file*). Nội dung của tệp đối tượng này có thể được xem như ***là một cấu trúc dữ liệu trung gian*** và thường được định nghĩa như một định dạng chuẩn COFF (*Common Object File Format*) hay định dạng của bộ liên kết mở rộng ELF (*Extended Linker Format*)… Nếu sử dụng nhiều bộ biên dịch cho các *modul* mã nguồn của một chương trình lớn thì cần phải đảm bảo rằng các tệp đối tượng được tạo ra phải có chung một kiểu định dạng.

Hầu hết nội dung của các tệp đối tượng đều bắt đầu bởi một phần *header* để mô tả các phần theo sau. Mỗi một phần sẽ chứa một hoặc nhiều khối mã hoặc dữ liệu như được sử dụng trong tệp mã nguồn. Tuy nhiên các khối đó được nhóm lại bởi bộ biên dịch vào trong các phần liên quan. Ví dụ như tất cả các khối mã được nhóm lại vào trong một phần được gọi là *text*, các biến toàn cục đã được khởi tạo (cùng các giá trị khởi tạo của chúng) vào trong phần dữ liệu, và các biến toàn cục chưa được khởi tạo vào trong phần bss.

Cũng khá phổ biến thường có một bảng biểu tượng chứa trong nội dung của tệp đối tượng. Nó chứa tên và địa chỉ của tất cả các biến và hàm được tham chiếu trong tệp mã nguồn. Các phần chứa trong bảng này không phải lúc nào cũng đầy đủ vì có một số biến và hàm được định nghĩa và chứa trong các tệp mã nguồn khác. Chính vì vậy cần phải có bộ liên kết để thực thi xử lý vấn đề này.

- **Quá trình liên kết (*Linking*)**

Tất cả các tệp đối tượng nhận được sau bước thực hiện biên dịch đầu tiên đều phải được tổ hợp lại theo một cách đặc biệt trước khi nó được nạp và chạy ở trên môi trường phần cứng đích. Như đã thấy ở trên, bản thân các tệp đối tượng cũng có thể là chưa hoàn thiện vì vậy bộ liên kết phải xử lý để tổ hợp các tệp đối tượng đó với nhau và hoàn thiện nốt phần còn khuyết cho các biến hoặc hàm tham chiếu liên thông giữa các tệp mã nguồn được biên dịch độc lập.

Kết quả đầu ra của bộ liên kết ***là một tệp đối tượng mới có chứa tất cả mã và dữ liệu trong tệp mã nguồn và cùng kiểu định dạng tệp***. Nó thực thi được điều này bằng cách tổ hợp một cách tương ứng các phần *text*, dữ liệu và phần *bss* …từ các tệp đầu vào và tạo ra một tệp đối tượng theo định dạng mã máy thống nhất. Trong qúa trình bộ liên kết thực hiện tổ hợp các phần nội dung tương ứng nó còn thực hiện thêm cả vấn đề hoàn

chỉnh các địa chỉ tham chiếu của các biến và hàm chưa được đầy đủ trong bước thực hiện biên dịch.

Các bộ liên kết có thể được kích hoạt thực hiện độc lập với bộ biên dịch và các tệp đối tượng được tạo ra bởi bộ biên dịch được coi như các tham biến vào. Đối với các ứng dụng nhúng nó thường chứa phần mã khởi tạo đã được biên dịch cũng phải được gộp ở trong danh sách tham biến vào này.

Nếu cùng một biểu tượng được khai báo hơn một lần nằm trong một tệp đối tượng thì bộ liên kết sẽ không thể xử lý. Nó sẽ kích hoạt cơ chế báo lỗi để người phát triển chương trình xem xét lại. Hoặc khi một biểu tượng không thể tìm được địa chỉ tham chiếu thực trong toàn bộ các tệp đối tượng thì bộ liên kết sẽ cố gắng tự giải quyết theo khả năng cho phép dựa vào các thông tin ví dụ như chứa trong phần mô tả của thư viện chuẩn. Điều này cũng thường hoặc có thể gặp với trường hợp các hàm tham chiếu trong chương trình.

Rất đáng tiếc là các hàm thư viện chuẩn thường yêu cầu một vài thay đổi trước khi nó có thể được sử dụng trong chương trình ứng dụng nhúng. Vấn đề ở đây là các thư viện chuẩn cung cấp cho các bộ công cụ phát triển chỉ dừng đến khả năng định dạng và tạo ra tệp đối tượng. Hơn nữa chúng ta cũng rất ít khi có thể truy nhập được vào mã nguồn của các thư viện chuẩn để có thể tự thay đổi. Hiện nay cũng có một số nhà cung cấp dịch vụ phần mềm hỗ trợ công cụ chuyển đổi hay thay đổi thư viện C chuẩn để ứng dụng cho các ứng dụng nhúng, ví dụ như *Cygnus*. Gói phần mềm này được gọi là *newlib* và được cung cấp miễn phí. Chúng ta có thể tải về trang web của *Cygnus*. Nó sẽ hỗ trợ chúng ta giải quyết vấn đề mà bộ liên kết có thể gặp phải khi chương trình sử dụng các hàm thuộc thư viện C chuẩn.

Sau khi đã hợp nhất thành công tất cả các thành phần mã và phần dữ liệu tương ứng cũng như các vấn đề về tham chiếu tới các biểu tượng chưa được thực thi trong quá trình biên dịch đơn lẻ, ***bộ liên kết sẽ tạo ra một bản sao đặc biệt của chương trình có khả năng định vị lại (relocatable)***. Hay nói cách khác, chương trình được hoàn thiện ngoại trừ một điều: Không có địa chỉ bộ nhớ nào chưa được gán bên trong các phần mã và dữ liệu. Nếu chúng ta không phải là đang phát triển phần mềm cho hệ nhúng thì quá trình biên dịch có thể kết thúc tại đây. Tuy nhiên, với hệ nhúng ngay cả hệ thống nhúng đã bao gồm cả hệ điều hành chúng ta vẫn cần phải có một mã chương trình (*image*) nhị phân được định vị tuyệt đối. Thực tế nếu có một hệ điều hành thì phần mã và dữ liệu cũng thường gộp cả vào bên trong chương trình có khả năng định vị lại. Toàn bộ ứng dụng nhúng bao gồm cả hệ điều hành thường liên kết tĩnh với nhau và thực hiện như một mã chương trình nhị phân thống nhất.

- **Quá trình định vị (*Locating*)**

Công cụ thực hiện việc chuyển đổi một chương trình có khả năng định vị lại thành một dạng mã chương trình nhị phân có thể thực thi được gọi là bộ định vị. Nó sẽ đảm nhiệm vai trò của bước đơn giản nhất trong các bước thực thi biên dịch nói chung. Thực

tế chúng ta phải tự làm hầu hết công việc của bước này bằng cách cung cấp thông tin về bộ nhớ đã được cấu hình trên nền phần cứng mà chúng ta đang phát triển và đó chính là tham số đầu vào cho việc thực thi của bộ định vị. Bộ định vị sẽ sử dụng thông tin này để gán các địa chỉ vật lý cho mỗi phần mã lệnh và dữ liệu bên trong chương trình được thực thi mà có thể định vị lại. Tiếp theo nó sẽ tạo ra một tệp có chứa chương trình bộ nhớ nhị phân để có thể nạp trực tiếp vào bộ nhớ chương trình trên nền phần cứng thực thi.

Trong nhiều trường hợp bộ định vị là một chương trình khá độc lập với các phần công cụ khác trong hệ thống phần mềm phát triển. Tuy nhiên trong các bộ công cụ phát triển GNU chức năng này được tích hợp luôn trong bộ liên kết. Tuy nhiên không nên nhầm lẫn về chức năng của chúng trong quá trình thực thi biên dịch. Thông thường chương trình chạy trên các máy tính mục đích chung thì hệ điều hành sẽ thực hiện việc chuyển đổi và gán chính xác địa chỉ thực cho các phần mã và dữ liệu trong chương trình ứng dụng, còn với chương trình phát triển chạy trên hệ nhúng thì việc này phải được thực hiện bởi bộ định vị. Đây cũng chính là điểm khác biệt cơ bản khi thực hiện biên dịch một chương trình ứng dụng cho hệ nhúng.

Thông tin về bộ nhớ vật lý của hệ thống phần cứng phát triển mà cần phải cung cấp cho bộ định vị GNU phải được định dạng theo kiểu biểu diễn của bộ liên kết. Thông tin này đôi khi được sử dụng để điều khiển một cách chính xác thứ tự trong các phần mã chương trình và dữ liệu bên trong chương trình có thể định vị lại. Nhưng ở đây chúng ta cần phải thực hiện nhiều hơn thế, tức là phải thiết lập chính xác khu vực của mỗi phần trong bộ nhớ.

Sau đây là một ví dụ minh họa của một tệp thông tin liên kết được cung cấp cho một nền phần cứng nhúng, giả thiết là có 512 KB RAM và 512 KB ROM.

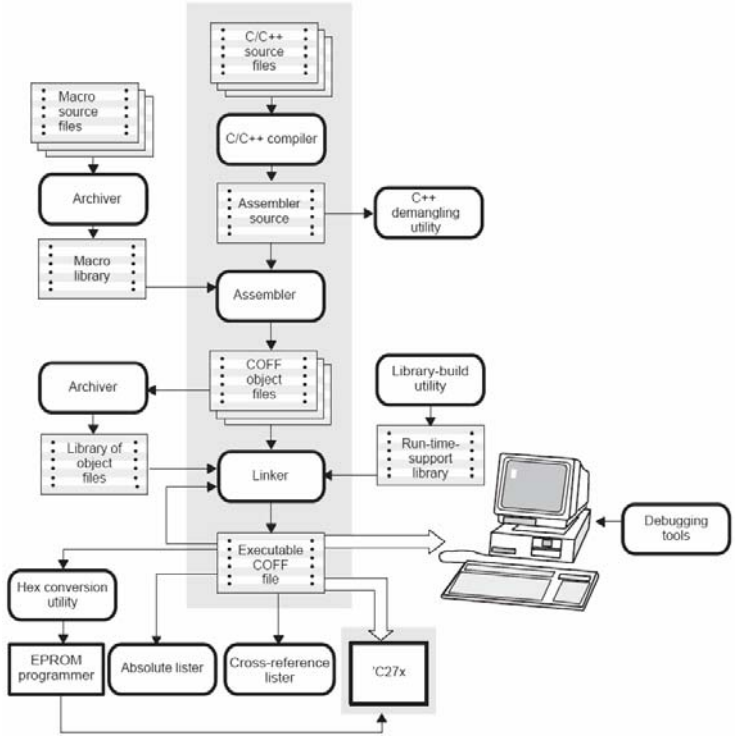
```
MEMORY
{
    ram : ORIGIN = 0x00000, LENGTH = 512K
    rom : ORIGIN = 0x80000, LENGTH = 512K
}
SECTIONS
{
    data ram : /* Initialized data. */
{
    _DataStart = . ;
    *(.data)
    _DataEnd = . ;
} >rom
```

```
bss : /* Uninitialized data. */
{
    _BssStart = . ;
    *(.bss)
    _BssEnd = . ;
}
_BottomOfHeap = . ; /* The heap starts here. */
_TopOfStack = 0x80000; /* The stack ends here. */
text rom : /* The actual instructions. */
{
    *(.text)
}
}
```

Đoạn mã này được cung cấp cho cho bộ định vị của bộ liên kết GNU về thông tin bộ nhớ đã được cấu hình trên nền mạch cứng hệ nhúng đích và chỉ ra các phần dữ liệu và bss sẽ được định vị trong RAM (bắt đầu tại địa chỉ 0x00000) và phần mã chương trình sẽ được định vị trong ROM (bắt đầu tại địa chỉ 0x80000). Tuy nhiên các giá trị khởi tạo trong các đoạn dữ liệu sẽ được thực hiện một phần trong ở ROM bắt đầu từ phần định nghĩa của khu vực định vị cuối cùng trong mã chương trình.

Tất cả các tên bắt đầu bởi dấu gạch dưới ("_") là các biến có thể được tham chiếu từ bên trong mã nguồn. Bộ liên kết sẽ sử dụng các biểu tượng để xử lý các tham chiếu trong các tệp đối tượng. Ví dụ có thể có một phần chương trình ứng dụng nhúng (thường là thuộc phần mã khởi tạo chương trình) sao chép các giá trị khởi tạo của các biến đã được khởi tạo trong ROM sang khu vực dữ liệu trong RAM. Các địa chỉ bắt đầu và kết thúc cho hoạt động này có thể được thiết lập một cách biểu tượng bởi tham chiếu tới các biến số nguyên **_DataStart** và **_DataEnd**.

Kết quả của bước cuối cùng này của quá trình biên dịch là một mã chương trình nhị phân có thể được nạp trực tiếp và chạy được trên nền phần cứng hệ nhúng đích, tức là được nạp vào bộ nhớ chương trình của hệ thống đích. Trong ví dụ trên mã chương trình nhị phân được tạo ra có dung lượng chính xác là 1MB. Tuy nhiên bởi vì các giá trị cho phần dữ liệu được khởi tạo nằm trong ROM nên nửa phần thấp 512KB của mã chương trình nhị phân này chỉ chứa giá trị *zero* và chỉ có nửa phần cao được sử dụng là chủ yếu.



Hình 3-9: Ví dụ về một lưu đồ phát triển phần mềm cho DSP TMS320Cxx

3.4.3 Simulator

Simulator là một chương trình phần mềm cho phép người phát triển mã chương trình chạy mô phỏng một chương trình viết cho một nền VXL/VĐK (nền phần cứng đích) trên một môi trường phần cứng khác (hay còn gọi là môi trường phát triển). Thực chất đó là quá trình mô phỏng hoạt động của chương trình thực thi theo đúng như điều kiện thực hiện của môi trường đích trên môi trường phát triển.

Sử dụng bộ mô phỏng mã chương trình có thể được chạy thử từng bước hoặc từng phần và có thể được chỉnh sửa trực tiếp để thử nghiệm các giải pháp khác nhau cho các bài toán thực thi phần mềm. Tuy nhiên các bộ mô phỏng không hỗ trợ các ngắt thực và các thiết bị ngoại vị.

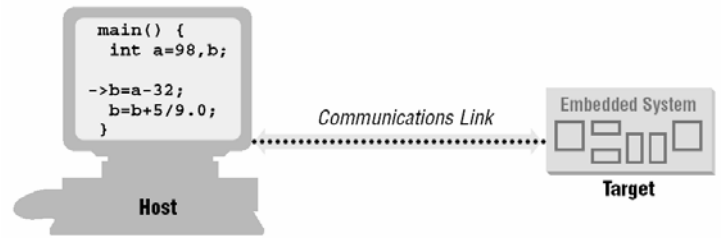
Bộ mô phỏng trực tiếp (bộ mô phỏng phần cứng) bao gồm một thiết bị phần cứng kết nối trực tiếp với hệ phát triển và cho phép thực thi để có được phản ứng giống như bộ xử lý đích. Bộ mô phỏng trực tiếp trên mạch có tất cả các chức năng của một bộ mô

phỏng phần mềm đồng thời hỗ trợ cả các chức năng *emulation* cho các cổng vào ra của VĐK.

3.4.4 Emulator

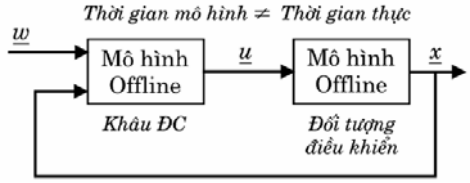
Emulator là một thiết bị phần cứng có khả năng thực hiện như một nền phần cứng đích. Nó còn được biết tới như một tên gọi khác là cộng cụ phát triển thời gian thực bởi vì nó cho ta phản ứng với các sự kiện như VĐK đích thực thi. Các bộ *Emulator* thường có kèm theo cả phần chương trình giám sát (*monitor program*) để cho phép người phát triển chương trình cho VĐK đích kiểm tra nội dung, trạng thái các thanh ghi và các khu vực bộ nhớ và thiết lập các điểm dừng khi thực hiện chạy chương trình.

3.4.5 Thiết kế hệ thống bằng máy tính



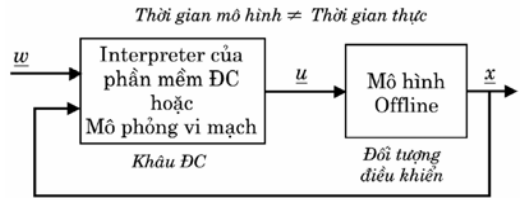
Trong quá trình phát triển phần mềm cần phải được thử nghiệm với đối tượng điều khiển. Tuỳ thuộc vào từng môi trường phát triển chúng ta có thể tiến hành theo một số các phương pháp sau.

- **Mô phỏng *offline***



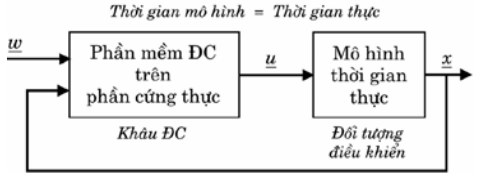
Trong hệ thống phát triển này nền ***phần cứng nhúng đích được mô phỏng bằng mô hình chạy trên PC và đối tượng điều khiển cũng là mô hình mô phỏng chạy trên PC***. Vì vậy quá trình phát triển thực chất là quá trình chạy mô phỏng hệ thống được thực hiện hoàn toàn trên PC. Với hệ thống này không thể thử nghiệm cho các sự kiện đáp ứng thời gian thực vì thời gian của mô phỏng khác với thời gian diễn biến thực của hệ thống.

- **Hệ thống phát triển (*software in the loop*)**



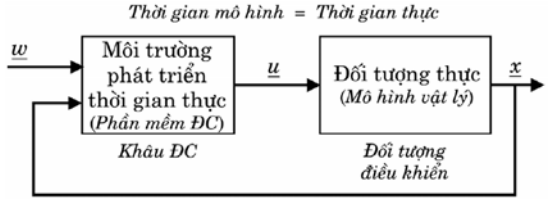
Hệ thống này ***mô phỏng nền phần cứng thực trên PC cho đáp ứng hành vi giống như với vi mạch cứng thực và mô hình đối tượng được mô hình thực thi trên PC***. Loại hệ thống này cũng tương tự như hệ thống mô phỏng *offline* tuy nhiên có ưu điểm hơn vì khả năng mô phỏng hành vi và đáp ứng của vi mạch nhúng chính xác hơn và trung thực hơn. Và cũng có một nhược điểm là không thử nghiệm được bài toán thời gian thực.

- **Mô phỏng thời gian thực (*Hardware in the Loop*)**



Hệ thống này ***sử dụng nền phần cứng nhúng đích thực nhưng đối tượng thì chỉ là mô hình thời gian thực không phải đối tượng thực***. Ưu điểm là khá mềm dẻo và thay đổi cấu hình đơn giản trong quá trình phát triển để thử nghiệm với các hành vi khác nhau của đối tượng. Rút ngắt và đơn giản hóa công việc xây dựng đối tượng.

- **Mô hình phát triển thực**



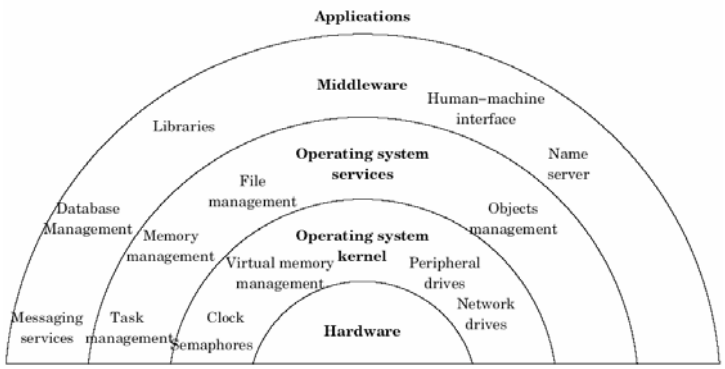
Hệ thống này ***sử dụng nền phần cứng nhúng đích thực với đối tượng thực***. Tuy nhiên có sự hỗ trợ của công cụ phát triển để có thể cài đặt và thử nghiệm trực tiếp trên nền phần cứng thực. Đây là một dạng mô hình cho kết quả trung thực và chính xác nhất trong các dạng hệ thống phát triển nêu trên. Tuy nhiên các nền phần cứng này thường được phát triển và hỗ trợ bởi các nhà cung cấp để có thể tương thích với công cụ phần mềm kèm theo.

4 HỆ ĐIỀU HÀNH NHÚNG

4.1 Hệ điều hành

Nguồn gốc ra đời của hệ điều hành là để đảm nhiệm vai trò trung gian để tương tác trực tiếp với phần cứng của máy tính, phục vụ cho nhiều ứng dụng đa dạng. Các hệ điều hành cung cấp một tập các chức năng cần thiết để cho phép các gói phần mềm điều khiển phần cứng máy tính mà không cần phải can thiệp trực tiếp sâu. Hệ điều hành của máy tính có thể thấy nó bao gồm các *drivers* cho các ngoại vi tích hợp với máy tính như card màn hình, card âm thanh... Các công cụ để quản lý tài nguyên như bộ nhớ và các thiết bị ngoại vi nói chung. Điều này tạo ra một giao diện rất thuận lợi cho các ứng dụng và người sử dụng phát triển phần mềm trên các nền phần cứng đã có. Đồng thời tránh được yêu cầu và hiểu biết sâu sắc về phần cứng và có thể phát triển dựa trên các ngôn ngữ bậc cao.

Hệ thống điều hành bản chất cũng là một loại phần mềm nhưng nó khác với các loại phần mềm thông thường. Sự khác biệt điển hình là hệ thống điều hành được nạp và thực thi đầu tiên khi hệ thống bắt đầu khởi động và được thực hiện trực tiếp bởi bộ xử lý của hệ thống. Hệ thống điều hành được viết để phục vụ điều khiển bộ xử lý cũng như các tài nguyên khác trong hệ thống bởi vì nó sẽ đảm nhiệm chức năng quản lý và lập lịch các quá trình sử dụng CPU và cùng chia sẻ tài nguyên.



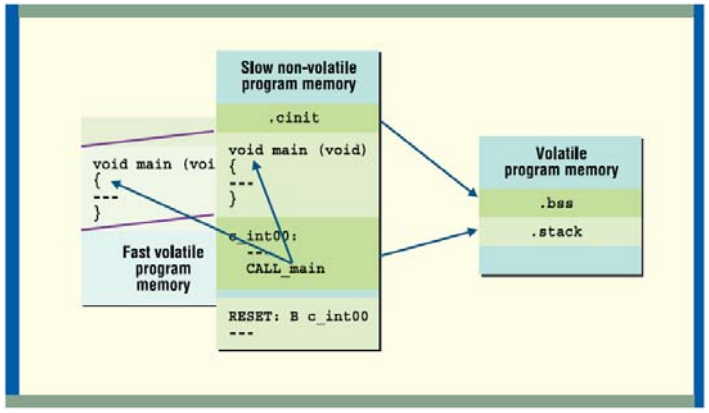
Hình 4-1: Kiến trúc hệ điều hành

Tóm lại, ***hệ điều hành thực chất chính là một giao diện quan trọng, giao tiếp trực tiếp với tầng phần cứng cấp thấp phục vụ cho cả người sử dụng cũng như các chương trình ứng dụng thực thi trên nền phần cứng hệ thống***. Hơn nữa hệ điều hành còn có vai trò quan trọng trong việc đảm nhiệm 3 tác vụ nguyên lý chính: (1) Quản lý quá trình, (2) Quản lý tài nguyên, (3) Bảo vệ tài nguyên khỏi sự xâm phạm của các quá trình thực thi sai.

4.2 Bộ nạp khởi tạo (*Boot-loader*)

Thuật ngữ "*bootstrap*" bắt nguồn từ cách diễn đạt cổ xưa có nghĩa là tự mình vươn lên bằng chính nỗ lực của bản thân "*pulling yourself up by your own bootstraps*". Nó đã được sử dụng như một thuật ngữ rất phổ biến để gọi tên một phần mềm thực thi việc khởi tạo chương trình thực thi trên các nền vi điện tử khả trình nói chung. Chương trình này thường rất nhỏ gọn và đảm nhiệm chức năng tiền hoạt động của hệ điều hành. Cũng rất phổ biến hiện nay người ta cũng thường dùng thuật ngữ "*boot-loader*" (bộ nạp khởi tạo) thay vì "*bootstrap*". Bộ nạp khởi tạo thực chất là một chương trình nhỏ thực hiện trong hệ thống và đảm nhiệm chức năng cần thiết để đưa hệ điều hành vào hoạt động. Trong các hệ nhúng, các lệnh được thực hiện đầu tiên thường nằm trong các vùng nhớ ROM và thường thuộc loại chậm. Dó đó, một trong những tác vụ phổ biến của bộ nạp khởi tạo là sao chép chương trình ứng dụng chính (*main program*) vào trong vùng bộ nhớ nhanh trước khi chúng được thực hiện. Bộ nạp khởi tạo cũng có nhiệm vụ khởi tạo vùng nhớ dữ liệu và các thanh ghi hệ thống trước khi nhảy tới chương trình ứng dụng chính. Cũng có rất nhiều dạng khác nhau của bộ nạp khởi tạo, từ dạng đơn giản đến phức tạp. Dạng đơn giản nhất có thể chỉ là một lệnh nhảy tới chương trình ứng dụng chính ngay sau khi *reset* mà không thực hiện bất kỳ một tác vụ khởi tạo hay nạp chương trình gì. Chương trình ứng dụng chính sẽ phải tự thiết lập để thực thi tác vụ của mình. Các bộ nạp khởi tạo phức tạp hơn có thể thực hiện nhiệm vụ chuẩn đoán bộ nhớ và khởi tạo hệ thống, kiểm tra chương trình và nạp chúng trước khi cho bộ xử lý nhảy tới thực hiện chương trình ứng dụng chính.

Sau đây chúng ta sẽ tìm hiểu về một môi trường phát triển khá điển hình và thảo luận về một số các thuộc tính nguyên lý cơ bản của bộ nạp khởi tạo.



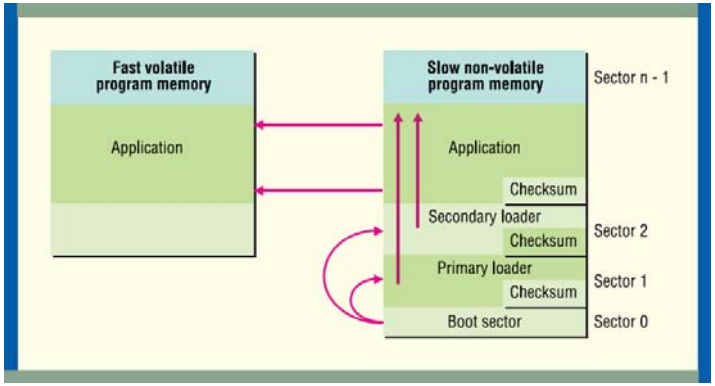
Hình 4-2: Nguyên lý thực hiện của bộ nạp khởi tạo Boot-loader

Trong môi trường phát triển hệ nhúng điển hình, nền phần cứng đích cần phát triển được kết nối với trạm chủ (*host*) thông qua một giao diện truyền thông. Một môi trường phát triển bao gồm một chương trình gỡ rối (*debugger*) ví dụ như *Code Composer Studio*

của *Texas Instrument*, để cho phép người phát triển chương trình nạp và thực hiện thử nghiệm các chương trình trên phần cứng đích. Một số các công cụ hỗ trợ ví dụ như để thiết lập các điểm dừng (*breakpoint*)…và các nhiệm vụ phụ trợ khác để bám sát trạng thái quá trình thực thi thời gian thực của chương trình thử nghiệm. Điều này rất có ý nghĩa và tạo nên một sự dễ dàng hơn trong quá trình phát triển và gỡ rối một chương trình ứng dụng mới cho nền phần cứng đích.

Thông thường các ứng dụng được phát triển trong môi trường ngôn ngữ C thì chương trình ứng dụng chính được thực thi và nằm trong phạm vi hàm *main()* phần khởi tạo chương trình và nạp tiền thực hiện chương trình chính thường không tường minh hoặc bị ẩn đi. Thực chất điều này chỉ đúng đối với những người phát triển mã chương trình ứng dụng chính bằng ngôn ngữ bậc cao (đặc biệt cho các ứng dụng không phải cho hệ nhúng) mà không cần phải quan tâm nhiều đến các tác vụ cơ sở đảm nhiệm việc khởi tạo các thanh ghi hệ thống, ngăn xếp và dữ liệu…Điều này cũng rất có ý nghĩa để tạo ra một cảm giác và môi trường phát triển thân thiện cho người phát triển chương trình và chỉ cần tập trung phần thực hiện chức năng chính của hệ thống. Tuy nhiên trong môi trường phát triển hệ thống nhúng việc thực thi chương trình thường bắt đầu tại địa chỉ chương trình nơi bắt đầu tác vụ khởi tạo hệ thống trước khi nhảy tới thực hiện chương trình chính *main()*. Quá trình này được bắt đầu thực chất là thực thi một tác vụ ngắt kích hoạt bởi sự kiện *reset*.

Boot-loader cũng có nhiều dạng khác nhau. Hình 4-2 mô tả một bộ nạp khởi tạo cho một ứng dụng C nhúng. Trong ví dụ này vector RESET trỏ tới thủ tục c_int00 thực hiện tác vụ khởi tạo. Ngoài việc khởi tạo các thanh ghi, ngăn xếp… các biến C cũng cần được khởi tạo trước khi được thực thi. Quá trình này sẽ sao chép từ phần .cinit và viết vào các địa chỉ dữ liệu tương ứng của chúng trong phần .bss. Sau khi hoàn thành chương trình chính main() mới được gọi và bắt đầu thực thi. Trong ví dụ đơn giản này bộ nạp khởi tạo tổ hợp vector RESET cùng với hàm khởi tạo c_int00 và giả thiết rằng cả chương trình bộ nạp khởi tạo và chương trình ứng dụng chính đều nằm cùng trong vùng nhớ vật lý *non-volatile*. Trong các trường hợp hệ thống phức tạp hơn, bộ nạp khởi tạo có thể bao hàm cả tác vụ sao chép chương trình chính vào trong vùng nhớ *fast volatile* trước khi nó được gọi và thực thi. Bộ nạp khởi tạo cũng có thể đảm nhiệm cả chức năng chuẩn đoán, gỡ rối và nâng cấp hệ thống nếu có. Chức năng chuẩn đoán có thể chỉ là kiểm tra bộ nhớ, ngoại vi và độ tương thích tích hợp trong hệ thống. Chức năng gỡ rối cũng có thể là một giao diện giám sát cung cấp thông tin và trạng thái thời gian thực về hệ thống mà người ta vẫn thường biết tới với tên gọi là chương trình *monitoring*. Việc nâng cấp hoặc thay đổi chương trình bộ nạp khởi tạo cũng có thể được thực thi nhờ khả năng lập trình FLASH *in-circuit* và nạp từ bộ nhớ ngoài thông qua giao diện với trạm chủ hoặc chức năng tương tự.



Hình 4-3: Cấu trúc của bộ nạp khởi tạo Boot-loader

4.3 Các yêu cầu chung

Như chúng ta đã được biết đối với các hệ thống thời gian thực, yêu cầu thiết kế một hệ điều hành khá đặc biệt. Hệ nhúng thời gian thực lại yêu cầu hệ điều hành phải thực hiện với một nguồn tài nguyên thường rất hạn hẹp. Mặc dù kích thước bộ nhớ tích hợp o*n-chip* sẽ có thể tăng lên trong tương lai nhưng với sự phát triển hiện nay hệ điều hành cho các hệ nhúng chỉ nên cỡ khoảng nhỏ hơn 32 *Kbytes*.

Hệ thống điều hành đảm nhiệm việc điều khiển các chức năng cơ bản của hệ thống bao gồm chủ yếu là quản lý bộ nhớ, ngoại vi và vào ra giao tiếp với hệ thống phần cứng. Một điểm khác biệt cơ bản như chúng ta đã biết về hệ điều hành với các phần mềm khác là nó thực hiện chức năng điều khiển sự kiện thực thi trong hệ thống. Có nghĩa là nó thực hiện các tác vụ theo mệnh lệnh yêu cầu từ các chương trình ứng dụng, thiết bị vào ra và các sự kiện ngắt.

Bốn nhân tố chính tác động trực tiếp tới quá trình thiết kế hệ điều hành là (1) khả năng thực hiện, (2) năng lượng tiêu thụ, (3) giá thành, và (4) khả năng tương thích. Hiện nay chúng ta cũng có thể bắt gặp rất nhiều hệ điều hành khác nhau đặc biệt cho các hệ nhúng cũng vì sự tác động của 4 nhân tố nêu trên. Hầu hết chúng đều có kiểu dạng và giao diện khá giống nhau nhưng cơ chế quản lý và thực thi các tác vụ bên trong rất khác nhau. Mỗi hệ điều hành được thiết kế phục vụ trực tiếp các chức năng đặc thù phần cứng của hệ nhúng và không dễ dàng so sánh được giữa chúng với nhau.

Hai thành phần chính trong thiết kế hệ điều hành là: phần hạt nhân (*kernel*) và các chương trình hệ thống. Hạt nhân nó chính là phần lõi của hệ điều hành. Nó được sử dụng để phục vụ cho các bộ quản lý quá trình, bộ lập lịch bộ quản lý tài nguyên và bộ quản lý vào ra. Phần hạt nhân đảm nhiệm chức năng lập lịch, đồng bộ và bảo vệ hệ thống bởi việc sử dụng sai, xử lý ngắt…Chức năng điều khiển chính của nó là phục vụ điều khiển phần cứng bao gồm ngắt, các thanh ghi điều khiển, các từ trạng thái và các

bộ định thời gian. Nó nạp các phần mềm điều khiển thiết bị để cung cấp các tiện ích chung và phối hợp với các hoạt động vào ra với hệ thống. Phần hạt nhân có vai trò điều khiển rất quan trọng để đảm bảo tất cả các phần của hệ thống có thể làm việc ổn định và thống nhất.

Hai kiến trúc thiết kế phần hạt nhân kinh điển nhất là kiến trúc vi hạt nhân và đơn hạt nhân (*monolithic*). Các vi hạt nhân cung cấp các chức năng điều hành cơ bản cốt lõi (thô) theo cơ chế các *module* tương đối độc lập đảm nhiệm các tác vụ cụ thể và chuyển rời rất nhiều các dịch vụ điển hình điều hành hệ thống thực thi trong không gian người sử dụng. Nhờ cơ chế này mà các dịch vụ có thể được khởi tạo hoặc cấu hình lại mà không nhất thiết phải khởi tạo lại toàn bộ hệ thống. Kiến trúc vi hạt nhân cung cấp độ an toàn cao bởi vì dịch vụ hệ thống chạy ở tầng người sử dụng với hạn chế về truy nhập vào tài nguyên của hệ thống và có thể được giám sát. Kiến trúc vi hạt nhân có thể được xây dựng một cách mềm dẻo để phù hợp với cấu hình phần cứng khác nhau một cách llinh hoạt hơn so với kiểu kiến trúc hạt nhân *monilithic*. Tuy nhiên do tính độc lập tương đối giữa các *modul* trong vi hạt nhân nên cần thiết phải có một cơ chế trao đổi thông tin hay truyền thông giữa các *modul* đó vì vậy có thể là lý do làm chậm tốc độ và giảm tính hiệu quả hoạt động của hệ thống. Đặc điểm nổi bật và cốt lõi của kiến trúc vi hạt nhân là kích thước nhỏ và dễ dàng sửa đổi cũng như xây dựng linh hoạt hơn. Các dịch vụ thực thi ở tầng trên của hạt nhân vì vậy đạt được độ an toàn cao. Kiến trúc vi hạt nhân được phát triển mạnh mẽ trong các hệ thống đa xử lý ví dụ như Windows 2000, Mach và QNX.

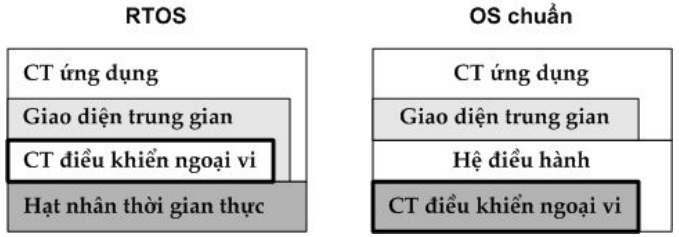
Kiểu kiến trúc *monolithic* cung cấp tất cả chức năng/dịch vụ chính yếu thông qua một qua trình xử lý đơn lẻ. Chính vì vậy kích thước của chúng thường lớn hơn kiểu kiến trúc vi hạt nhân. Loại hình kiến trúc này thường được áp dụng chủ yếu cho các phần cứng cụ thể mà hạt nhân *monolithic* có sự tương tác trực tiếp với phần cứng nhờ vậy mà khả năng tối ưu cũng dễ dàng hơn so với áp dụng kiểu kiến trúc vi hạt nhân. Chính vì vậy cũng là lý do tại sao kiến trúc *monolithic* không thể thay đổi mềm dẻo linh hoạt như kiểu vi hạt nhân. Ví dụ điển hình về loại hình kiến trúc hạt nhân *monolithic* bao gồm Linux, MacOS, và DOS.

Vì hệ điều hành cũng đòi hỏi về tài nguyên và kiêm cả chức năng quản lý chúng vì vậy người thiết kế cần phải nắm được thông tin về chúng một cách đầy đủ. Ví dụ như đối với hệ thống điều hành cho *Sun Microsystem Solaris* yêu cầu tối thiểu không gian bộ nhớ trên đĩa là 8MB; Windows 2000 yêu cầu khoảng gấp hai lần như vậy.

4.4 Hệ điều hành thời gian thực

QNX là một ví dụ điển hình về hệ thống thời gian thực RTOS được thiết kế để đáp ứng các yêu cầu về lập lịch rất khắt khe. QNX cũng chưa thực sự phù hợp để có thể được thực thi cho các hệ thống nhúng bởi vì nó đòi hỏi dung lượng bộ nhớ không nhỏ và thường phù hợp cho các ứng dụng đòi hỏi về độ an toàn và độ tin cậy lớn.

Hệ thống điều hành thời gian thực là hệ điều hành hỗ trợ khả năng xây dựng các hệ thống thời gian thực.



Hình 4-4: So sánh kiến trúc RTOS và OS chuẩn

Hệ thống điều hành với phần lõi là hạt nhân phải đảm nhiệm các tác vụ chính như sau:
Xử lý ngắt

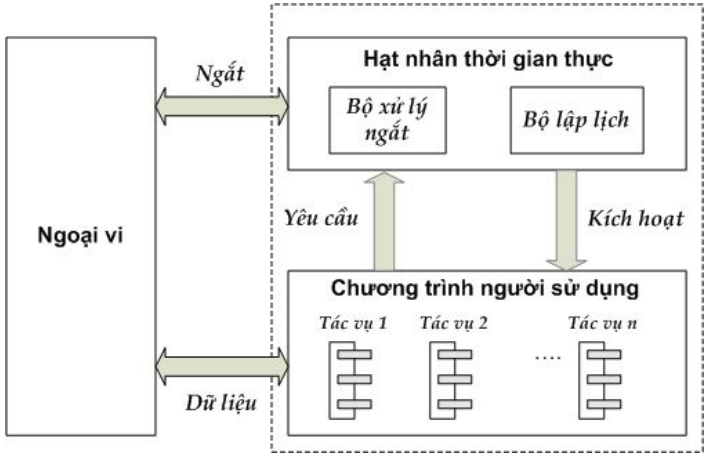
- Lưu trữ ngữ cảnh chương trình tại thời điểm xuất hiện ngắt
- Nhận dạng và lựa chọn đúng bộ xử lý và phục vụ dịch vụ ngắt

Điều khiển quá trình

- Tạo và kết thúc quá trình/tác vụ
- Lập lịch và điều phối hoạt động hệ thống
- Định thời

Điều khiển ngoại vi

- Xử lý ngắt
- Khởi tạo giao tiếp vào ra



Hình 4-5: Cấu trúc hệ điều hành thời gian thực

Tùy theo cơ chế thực hiện và xây dựng hoạt động của hạt nhân người ta phân loại một số loại hình:

(1) Hệ thống thời gian thực nhỏ: Với loại này các phần mềm được phát triển mà không cần có hệ điều hành, người lập trình phải tự quản lý và xử lý các vấn đề về điều khiển hệ thống bao gồm:

- Xử lý ngắt
- Điều khiển quá trình/ tác vụ
- Quản lý bộ nhớ

(2) Công nghệ đa nhiệm

- Mỗi quá trình có một không gian bộ nhớ riêng
- Các quá trình phải được chia nhỏ thành các *Thread* cùng chia sẻ không gian bộ nhớ.

(3) Các dịch vụ cung cấp bởi hạt nhân

- Tạo và kết thúc quá trình/ tác vụ
- Truyền thống giữa các quá trình
- Các dịch vụ về định thời gian
- Một số các dịch vụ cung cấp hỗ trợ việc thực thi liên quan đến điều khiển hệ thống

Đặc điểm cơ bản của hạt nhân thời gian thực điển hình:

- Kích thước nhỏ (lưu trữ toàn bộ trong ROM)
- Hệ thống ngắt
- Không nhất thiết phải có các cơ chế bảo vệ
  - ✓ Chỉ hỗ trợ phần kiểm tra chương trình ứng dụng
  - ✓ Tăng tốc độ chuyển ngữ cảnh và truyền thông giữa các quá trình
  - ✓ Khi các quá trình ứng dụng đang thực hiện thì các yêu cầu hệ thống điều hành có thể được thực hiện thông qua các lời gọi hàm thay vì sử dụng cơ chế ngắt mềm
- Vi hạt nhân (*Micro-kernel*): Bao gồm một tập nhỏ các dịch vụ hỗ trợ
  - ✓ Quản lý quá trình
  - ✓ Các dịch vụ truyền thông giữa các quá trình nếu cần
  - ✓ Các phần mềm điều khiển thiết bị là các quá trình ứng dụng

Hạt nhân điển hình cơ bản

- Loại hạt nhân đơn giản nhất là một vòng lặp vô hạn thăm dò các sự kiện xuất hiện trong hệ thống và phản ứng lại theo sự thay đổi nếu có.
- Với một bộ xử lý cấu hình nhỏ nhất, không phải lúc nào nó cũng có thể lưu cất ngữ cảnh vì không thể thay đổi con trỏ ngăn xếp hoặc vùng ngăn xếp rất hạn chế.
- Thay vì sử dụng các thanh ghi thiết bị, vòng lặp thăm dò có thể giám sát các biến mà chịu sự thay đổi cập nhật bởi các bộ xử lý ngắt.
- Hạt nhân có thể được xây dựng sao cho tất cả các tín hiệu logic được điều khiển bởi vòng lặp và nhịp được điều khiển bởi các ngắt.
- Các tác vụ lớn cần nhiều thời gian thực hiện có thể được chia nhỏ thành các tác vụ nhỏ và được thực hiện tại các thời điểm khác nhau nhờ vào cơ chế chuyển và sử dụng bộ đếm.

- Các hạt nhân thực thi theo cơ chế ngắt rất giống với loại hạt nhân thực hiện theo cơ chế vòng lặp thăm dò. Nó xử lý tất cả các tác vụ thông qua các dịch vụ ngắt.
- Các hạt nhân lớn và phức tạp hơn sẽ bao gồm một số các dịch vụ phụ phục vụ cho việc truyền thông giữa các quá trình. Và nếu được bổ sung đầy đủ nó sẽ trở thành một hệ điều hành đầy đủ.

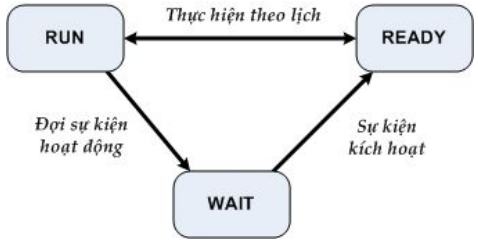
Các kiểu loại hạt nhân cơ bản

- Hạt nhân thực hiện vòng lặp thăm dò
- Hạt nhân thực hiện theo cơ chế ngắt
- Hạt nhân quá trình vận hành quá trình

Việc lựa chọn loại hạt nhân nào hoàn toàn tùy thuộc vào các bộ xử lý và kích thước phần mềm, tuy nhiên riêng loại hạt nhân vận hành theo quá trình không phù hợp với các bộ xử lý nhỏ.

Hạt nhân quá trình

Các hạt nhân quá trình rõ ràng là phức tạp hơn các hạt nhân thực hiện theo cơ chế thăm dò và điều khiển ngắt. Các đường truyền tín hiệu logic bên trong các quá trình và các dịch vụ ngắt được tích hợp và thực hiện thông qua việc truyền dữ liệu.



Hình 4-6: Mô hình trạng thái của quá trình

Hạt nhân sẽ phải đảm nhiệm chức năng lập lịch cho các quá trình theo đúng mô hình trạng thái.

- **RUN**: quá trình được thực hiện
- **WAIT**: các quá trình chờ một sự kiện hoặc tín hiệu vào ra kích hoạt quá trình
- **READY**: các quá trình sẵn sàng được thực hiện

Các phần tử thuộc tính của một quá trình: Các phần tử này cần thiết để phục vụ cho việc lập lịch. Ví dụ đối với cơ chế lập lịch theo mức độ ưu tiên sẽ yêu cầu thông tin sau với mỗi quá trình:

- ✓ Tên (địa chỉ bộ nhớ của phần tử quá trình)
- ✓ Trạng thái: RUN, WAIT, READY
- ✓ Mức độ ưu tiên
- ✓ Ngữ cảnh (dùng con trỏ để quản lý lưu cất thông tin trong ngăn xếp)

5 KỸ THẬT LẬP TRÌNH NHÚNG

5.1 Tác vụ và quá trình (*process*)

- Tác vụ (*task*) ? Là một công việc cần thực thi tham gia trong hệ thống
- Quá trình (*process*) là một diễn biến thực thi một tác vụ của hệ thống.

Đôi khi người ta vẫn dùng lẫn hai khái niệm này và không phân biệt.
Tác vụ chu kỳ (*period*) và không chu kỳ (*aperiod*)

Các tác vụ phải thực hiện lặp lại một cách đều đặn theo những khoảng thời gian p được gọi là các tác vụ có chu kỳ và p được gọi là chu kỳ của tác vụ. Các loại tác vụ khác thì được gọi là tác vụ không chu kỳ.

5.2 Lập lịch (*Scheduling*)

Tại sao phải lập lịch?
Để đảm bảo được cơ chế thực thi chia sẻ tài nguyên hữu hạn và thoả mãn yêu cầu thời gian thực. Lập lịch phải nhằm thoả mãn hay đạt tới được sự thoả hiệp giữa các ràng buộc về tài nguyên, sự phụ thuộc lẫn nhau hay thời gian thực hiện.

5.2.1 Các khái niệm

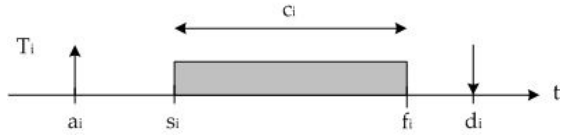
Lập lịch là một phép thực hiện phân bổ và gán quy trình thực thi các tác vụ cho bộ xử lý sao cho mỗi tác vụ được thực hiện hoàn toàn.

Lập lịch = tìm kiếm một giản đồ phân bố thời gian thực hiện đa nhiệm hợp lý với các điều kiện ràng buộc cho trước. Hay nói cách khác là bộ lập lịch phải xử lý để quyết định và điều phối quá trình/tác vụ thực hiện.

Có một số thông tin về tác vụ luôn phải quan tâm đối với bất kỳ bộ lập lịch thời gian thực nào, bao gồm:

- Thời gian xuất hiện a_i (*arrival time*): Khi sự kiện xảy ra và tác vụ tương ứng được kích hoạt.
- Thời điểm bắt đầu thực thi r_i (*release time*): Thời điểm sớm nhất khi việc xử lý đã sẵn sàng và có thể bắt đầu.
- Thời điểm bắt đầu thực hiện s_i (*starting time*): Là thời điểm mà tại đó tác vụ bắt đầu việc thực hiện của mình.
- Thời gian tính toán/thực thi c_i (*Computation time*): Là khoảng thời gian cần thiết để bộ xử lý thực hiện xong nhiệm vụ của mình mà không bị ngắt.
- Thời điểm hoàn thành f_i (*finishing time*): Là thời điểm mà tại đó tác vụ hoàn thành việc thực hiện của mình.
- Thời gian rủi ro/ xấu nhất w_i (*worst case time*): khoảng thời gian thực hiện lâu nhất có thể xảy ra.

- Thời điểm kết thúc d_i (*due time*): Thời điểm mà tác vụ phải hoàn thành.

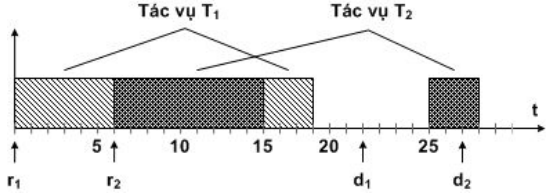


Hình 5-1: Giản đồ thực hiện của một tác vụ Ti

Trên cơ sở đó bộ lập lịch sẽ phải thực hiện bài toán tối ưu về:

- Thời gian đáp ứng (*response time*)
- Hiệu suất thực hiện (số lượng công việc thực hiện xong trong một đơn vị thời gian)
- Sự công bằng và thời gian chờ đợi (các tác vụ không phải chờ đợi quá lâu)

Ví dụ về một lịch thực hiện 2 tác vụ được mô tả như trong Hình 5-2.

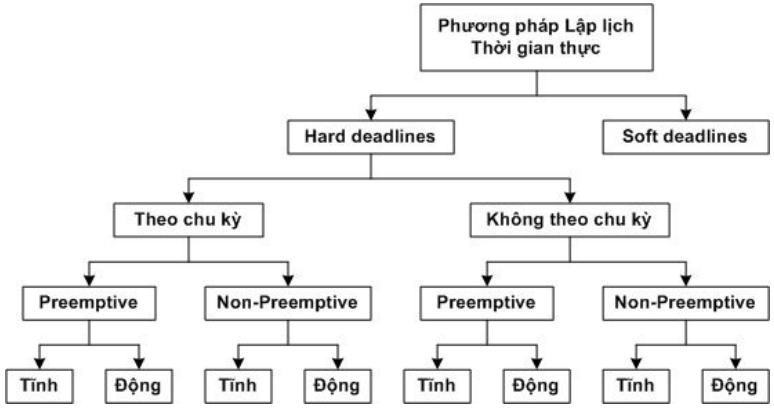


Hình 5-2: Giản đồ lập lịch thực hiện 2 tác vụ

Trong trường hợp của ví dụ này các thông số về thời gian thực hiện của các tác vụ tính được như sau:

- Thời gian tính toán $C_1 = 9$ và $C_2 = 12$.
- Thời gian bắt đầu thực hiện: $s_1 = 0$, $s_2 = 6$.
- Thời điểm hoàn thành: $f_1 = 18$, $f_2 = 28$.
- Khoảng thời gian chênh lệch thời điểm kết thúc và *deadline* (*Lateness*) $L_i = f_i - d_i$: $L_1 = -4$, $L_2 = 1$.
- Khoảng thời gian rỗi/dư thừa giữa thời gian cho phép thực hiện và thời gian cần để thực hiện tác vụ (*Laxity*) $X_i = d_i - a_i - C_i$: $X_1 = 13$, $X_2 = 11$.

5.2.2 Các phương pháp lập lịch phổ biến



Hình 5-3: Phân loại các phương pháp lập lịch

Tuỳ thuộc vào loại hình tác vụ, người ta ra hai phương pháp lập lịch là có chu kỳ và không có chu kỳ.

Lập lịch *non-preemptive*: Phương pháp này đảm bảo các tác vụ được thực hiện hoàn thành mỗi khi thực thi, vì vậy thời gian đáp ứng cho các sự kiện khác có thể lâu.

Lập lịch *preemptive*: Phương pháp này khắc phục nhược điểm của lập lịch *non-preemptive* khi thời gian thực thi các tác vụ lâu. Các tác vụ sẽ được thực hiện và có thể bị ngắt giữa chừng để phục vụ thực thi các tác vụ khác. Cơ chế lập lịch này cho phép đảm bảo thời gian đáp ứng cho các sự kiện và tác vụ ngắn và nhanh hơn.

Lập lịch *offline*/tĩnh: Việc lập lịch được thực hiện dựa trên các hiểu biết hoặc dự báo về các sự kiện tác vụ thực hiện trong hệ thống (thời điểm xuất hiện, thời gian thực hiện, *deadline*…) và được quyết định tại thời điểm thiết kế và được áp dụng cố định trong suốt quá trình hoạt động của hệ thống. Việc lập lịch trước có một số các ưu điểm sau:

- Tác vụ tiếp theo có thể được lựa chọn thực thi trong khoảng thời gian là hằng số
- Khả năng đáp ứng yêu cầu thời gian thực có thể được biết trước và được đảm bảo

Nhược điểm:

- Không thể thay đổi lịch trình thực hiện của hệ thống trong quá trình thực hiện
- Đòi hỏi phải có thông tin thời gian chính xác về các tác vụ để tính toán lập lịch

Một thuật toán lập lịch tĩnh được gọi là tối ưu nếu nó luôn luôn có thể tìm được một lịch điều phối thoả mãn các ràng buộc đã cho trong khi một thuật toán tĩnh khác cũng tìm được một lời giải.

Lập lịch *online*/động: Bộ xử lý thực hiện việc lập lịch trong quá trình thực thi dựa trên cơ sở các thông tin hoạt động hiện hành của hệ thống. Sơ đồ lập lịch là không xác định trước và thay đổi động theo quá trình thực hiện.
Các thuật toán lập lịch tĩnh tối ưu không phải là tối ưu trong hệ thống động.

Không tồn tại một lời giải tối ưu cho việc lập lịch trong hệ thống nhiều bộ xử lý nếu thời điểm xuất hiện yêu cầu thực thi của các tác vụ không được biết trước.
Các hạt nhân được điều khiển theo cơ chế ngắt thường thực thi cơ chế lập lịch *non-preemtive* động trong khi loại hạt nhân vận hành theo quá trình lại thực thi theo cơ chế *preemptive* động.

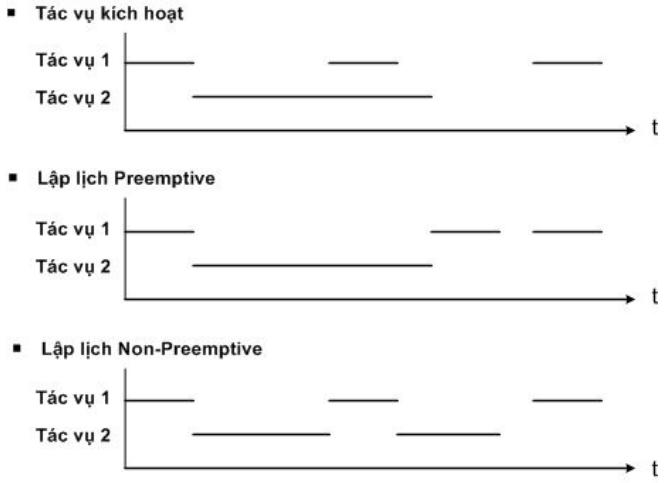
Một thuật toán lập lịch động được gọi là tối ưu nếu nó có thể tìm ra được một lịch điều phối điều khiển hệ thống thoả mãn các ràng buộc thời gian đã cho bất kể khi nào mà thuật toán tĩnh không tìm ra được.

Lập lịch tập trung hoặc phân tán: Việc lập lịch được thực hiện áp dụng cho các tác vụ thực thi bởi một (tập trung) hoặc nhiều bộ xử lý (phân tán).

Lập lịch *Mono* hay *Multi- processor*: Nhiệm vụ lập lịch và thực thi được đảm nhiệm bởi một (*mono*) hoặc nhiều bộ vi xử lý (*multi*). Tuỳ thuộc vào độ phức tạp về thuật toán cần xử lý khi lập lịch mà người ta quyết định phải sử dụng phương pháp lập lịch *mono* hay *multi-processor*.

Tính khả lập lịch: Một hệ thống với một tập các tác vụ và các điều kiện ràng buộc được gọi là khả lập lịch nếu tồn tại ít nhất một cơ chế lịch trình thực hiện thoả mãn các tác vụ và điều kiện ràng buộc đó.

Ví dụ về lập lịch cho hệ thống đa nhiệm với 2 tác vụ. Tác vụ 1 thực hiện theo chu kỳ và tác vụ 2 thực hiện không theo chu kỳ với thời gian thực thi lớn hơn thời gian chu kỳ lặp lại của tác vụ 1.



Hình 5-4: Giản đồ thời gian thực hiện lịch của tác vụ

5.2.3 Kỹ thuật lập lịch

❒ FCFS

Trong cơ chế lập lịch đến trước được phụ vụ trước thì các quá tình được xử lý theo thứ tự mà nó xuất hiện yêu cầu và cho đến khi hoàn thành. Cơ chế lập lịch này thuộc loại không ngắt được và có ưu điểm là dễ dàng thực thi. Tuy nhiên, nó không phù hợp cho các hệ thống mà hỗ trợ nhiều người sử dụng vì có một sự biến đổi lớn về thời gian trung bình mà một quá trình hay tác vụ phải chờ đợi để được xử lý. Hơn nữa do việc xử lý không ngắt được nên có hiện tượng chiếm hữu độc quyền bộ xử lý trong thời gian dài và có thể gây ra sự trễ bất thường trong quá trình thực hiện của các tác vụ phải chờ đợi khác.

❒ **Shortest Job First** - SJF

Trong cơ chế lập lịch này tác vụ có thời gian thực thi ngắn nhất sẽ có quyền ưu tiên cao nhất và sẽ được phục vụ trước. Vấn đề chính gặp phải trong cơ chế lập lịch này là không biết trước được thời gian thực thi của các tác vụ tham gia trong chương trình và thông thường phải áp dụng cơ chế tiên đoán và đánh giá dựa vào kinh nghiệm về các tác vụ thực thi trong hệ thống. Điều này chắc chắn rất khó để luôn đảm bảo được độ chính xác. Cơ chế lập lịch này có thể áp dụng cho cả loại ngắt được và không ngắt được.

❒ **Rate monotonic** (RM):

Phương pháp lập lich RM có lẽ hiện này là thuật toán được biết tới nhiều nhất áp dụng cho các tác vụ hay quá trình độc lập. Phương pháp này dựa trên một số giả thiết sau:

(1) Tất cả các tác vụ tham gia hệ thống phải có *deadline* kiểu chu kỳ
(2) Tất cả các tác vụ độc lập với nhau
(3) Thời gian thực hiện của các tác vụ biết trước và không đổi
(4) Thời gian chuyển đổi ngữ cảnh thực hiện là rất nhỏ và có thể bỏ qua

Thuật toán RM được thực thi theo nguyên lý ***gán mức ưu tiên cho các tác vụ dựa trên chu kỳ của chúng***. Tác vụ nào có chu kỳ nhỏ thì sẽ có được gán mức ưu tiên cao. Theo nguyên lý này với các tác vụ chu kỳ không thay đổi thì RM sẽ là phương pháp lập lịch cho phép ngắt và mức ưu tiên cố định. Tuy nhiên RM hỗ trợ yêu cầu hệ thống không tốt.

❒ **Earliest-deadline-first** (EDF)

Như đúng tên gọi của phương pháp, thuật toán lập lich theo phương pháp này sử dụng *deadline* của tác vụ hay như điều kiện ưu tiên để xử lý điều phối hoạt động. Tác vụ có *deadline* gần nhất sẽ có mức ưu tiên cao nhất và các tác vụ có *deadline* xa nhất sẽ nhận mức ưu tiên thấp nhất. Ưu điểm nổi bật của phương pháp này là giới hạn có thể lập lịch đáp ứng được 100% cho tất cả các tập tác vụ. Hơn nữa mức ưu tiên gán cho mỗi tác vụ trong quá trình hoạt động là động vì vậy chu kỳ của tác vụ có thể thay đổi bất kỳ lúc nào theo thời gian.

EDF có thể được áp dụng cho các tập tác vụ chu kỳ và cũng có thể mở rộng để đáp ứng cho các trường hợp các *deadline* thay đổi khác nhau theo chu kỳ.

Vấn đề chính của thuật toán lập lich EDF là không thể đảm bảo được tác vụ nào trong hệ thống có thể không được thực thi trong tình huống quá độ hệ thống bị quá tải. Trong nhiều trường hợp mặc dù mức độ sử dụng trung bình nhỏ hơn 100% những vẫn có thể trong một tình huống nào đó vẫn vượt qua khả năng đáp ứng của hệ thống tức là sẽ có tác vụ không được đảm bảo thực thi đúng. Trong những trường hợp như vậy cần phải điều khiển để biết tác vụ nào bị lỗi không thực hiện thành công hoặc tác vụ nào được thực hiện thành công trong quá trình quá độ.

❒ **Minimum Laxity first** (MLF)

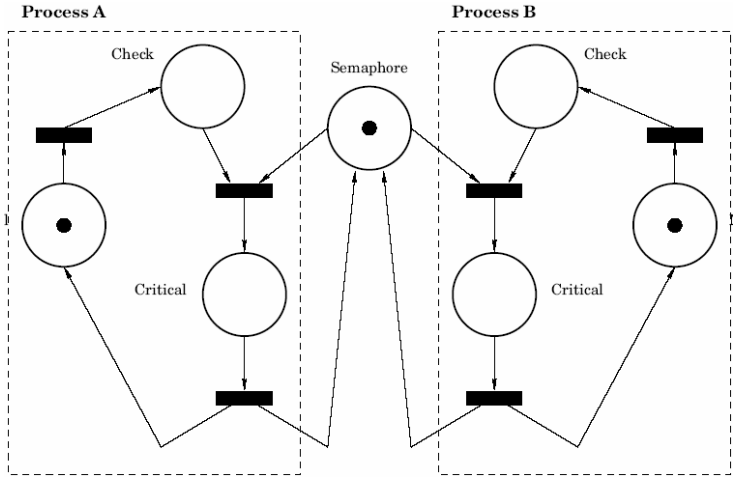
Cơ chế lập lịch này sẽ ưu tiên tác vụ nào còn ít thời gian còn lại để thực hiện nhất trước khi nó phải kết thúc để đảm bảo yêu cầu thực thi đúng. Đây được xem là cơ chế lập lịch gán quyền ưu tiên động và dễ đạt được sự tối ưu về hiệu suất thực hiện và sự công bằng trong hệ thống.

❒ **Round Robin**

Đây là một cơ chế lập lịch phân bổ đều đặn, ngắt được và đơn giản. Mỗi một tác vụ được xử lý/phục vụ trong một khoảng thời gian nhất định và lặp lại theo một chu trình xuyên suốt toàn bộ các tác vụ tham gia trong hệ thống. Khoảng thời gian phục vụ cho mỗi tác vụ trong quá trình là một sự thoả hiệp giữa thời gian thực hiện của các tác vụ và thời gian thực hiện một chu trình. Có thể chọn khoảng thời gian đó rất nhỏ và đôi lúc chúng ta không nhận được ra rằng đang có sự phân bổ thực hiện trong hệ thống. Tuy nhiên nếu thời gian đó quá nhỏ có thể làm mất tính hiệu quả thực hiện toàn hệ thống vì cần nhiều thời gian trong việc chuyển đổi ngữ cảnh cho mỗi tác vụ sau mỗi chu trình thực hiện.

5.3 Truyền thông và đồng bộ

5.3.1 Semaphore



Hình 5-5: Truyền thông quá trình

Semaphores là một cấu trúc dữ liệu được định nghĩa để loại trừ khả năng xung đột trong quá trình chia sẻ tài nguyên của các tác vụ trong hoạt động của hệ thống.

Semaphores hỗ trợ hai hoạt động chính như sau:

- ***wait***(*semaphore*): giảm và khoá cho tới khi *semaphore* được mở
- ***signal***(*semaphore*): tăng và cho phép thêm một luồng mới được tham gia hoạt động

Trong hoạt động phối hợp cùng với *semaphore* có một hàng đợi gồm các tác vụ cần được thực thi sẽ có một số tình huống hoạt động cơ bản như sau:

- Khi một luồng (*thread*) gọi ***wait***():
  - Nếu *semaphore* được mở thì luồng đó sẽ được gia nhập và tiếp tục thực thi
  - Nếu *semaphore* đang bị đóng thì nhánh đó sẽ bị khoá và phải nằm chờ trong hàng đợi cho tới khi nào *semaphore* được mở
- ***signal***() sẽ mở *semaphore*:
  - nếu một luồng đang nằm trong hàng đợi và không bị khoá
  - nếu không có luồng nào trong hàng đợi và tín hiệu ***signal*** sẽ được nhớ và dành cho luồng tiếp theo

Các loại *Semaphore*

- ***Mutex semaphore***
  - ✓ Cho phép điều khiển hoạt động truy nhập đơn lẻ vào tài nguyên chia sẻ của hệ thống.

  - ✓ Đảm bảo loại trừ xung đột trong hoạt động truy nhập đồng thời của nhiều tác vụ, và chỉ có một tác vụ được thực thi tại mỗi thời điểm.

- ***Counting semaphore***
  - ✓ Điều khiển tài nguyên mà có thể phục vụ cùng một lúc nhiều tác vụ hoặc một nguồn tài nguyên cho phép phục vụ một số nhất định các tác vụ không đồng bộ và hoạt động đồng thời.
  - ✓ Nhiều luồng có thể truyền *Semaphore*
  - ✓ Số lượng luồng được quyết định bởi biến đếm N của *Semaphore*

Thực chất *mutex semaphore* là một dạng đặc biệt của *counting semaphore* với biến đếm N=1.

Thực thi *Semaphore*

```
typedef struct {
    int value;
    struct process *L;
} semaphore;
```

```
void wait(semaphore S) {
    S.value--;
    if (S.value < 0) {
        add this process to S.L;
        sleep();
    }
}
```

```
void signal(semaphore S) {
    S.value++;
    if (S.value <= 0) {
        remove a process P from S.L;
        wakeup(P);
    }
}
```

Sử dụng *Semaphore* trong việc đồng bộ hai quá trình tạo và sử dụng hạng mục thông qua bộ đệm trung gian.

```
semaphore mutex = 1;        /* controls access to critical section*/
semaphore empty = N;        /* counts empty buffer slots */
semaphore full = 0;         /* counts full buffer slots */
```

```
void producer() {
    while (TRUE) {
        int item = produce_item();
        wait(&empty);
        wait(&mutex);
        insert_item(item);
        signal(&mutex);
        signal(&full);
    }
}
```

```
void consumer() {
    while (TRUE) {
        wait(&full);
        wait(&mutex);
        item = remove_item();
        signal(&mutex);
        signal(&empty);
        consume_item(item);
    }
}
```

Nhận xét:

- ✓ *Semaphores* có thể được sử dụng để giải quyết bất kỳ một bài toán hay vấn đề đồng bộ truyền thống nào
- ✓ Tuy nhiên chúng có một số nhược điểm
  - o Chúng chủ yếu sử dụng các biến toàn cục trong việc điều khiển hoạt động đồng bộ nên có thể truy nhập bất kỳ đâu trong hệ thống → khó kiểm soát
  - o Không có sự liên kết chặt chẽ giữa *semaphore* và dữ liệu mà được nó điều khiển.
  - o Được sử dụng đồng thời cho cả việc loại trừ xung đột (*mutual exclusion*) và hoạt động đồng bộ cho các tác vụ (*scheduling*)

5.3.2 Monitor

Monitor là một ngôn ngữ lập trình được xây dựng để điều khiển việc truy nhập vào vùng dữ liệu chia sẻ trong hoạt động của hệ thống. Mã chương trình đồng bộ được bổ sung vào trong bộ biên dịch và thực thi khi chạy chương trình.

✓ *Monitor* là một *modul* đóng gói

- Các cấu trúc dữ liệu được chia sẻ
- Các thủ tục hoạt động thao tác trên các cấu trúc dữ liệu chia sẻ
- Đồng bộ các luồng thực thi đồng thời mà có thể kích hoạt các thủ tục trong hoạt động hệ thống

✓ *Monitor* có thể bảo vệ dữ liệu khỏi sự truy nhập không có cấu trúc. Nó đảm bảo rằng các luồng truy nhập vào dữ liệu thông qua các thủ tục tương tác theo những cách hợp pháp và có kiểm soát.

✓ *Monitor* đảm bảo loại trừ xung đột

- Chỉ có một luồng có thể thực thi bất kỳ thủ tục nào tại mỗi một thời điểm (luồng trong monitor)
- Nếu có một luồng đang thực thi bên trong một *monitor* nó sẽ khoá các luồng khác muốn vào, do đó *monitor* cũng phải có một hàng đợi.

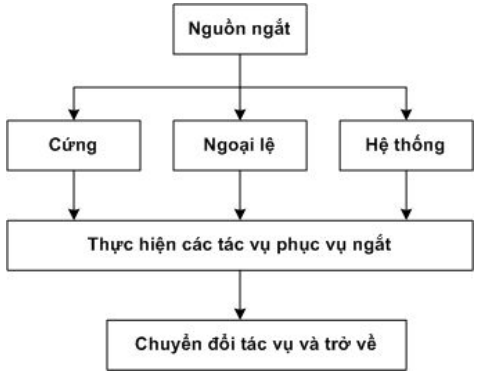
```
monitor ProducerConsumer {
    condition full, empty;
    int count=0;
    void insert(int item) {
        if (count == N) full.wait();
        insert_item(item);
        count++;
        if (count == 1) empty.signal();
    }
    int remove() {
        if (count == 0) empty.wait();
        int item = remove_item;
        count--;
        if (count == N-1) full.signal();
    }
}
```

```
void Producer() {
    while (TRUE) {
        int item = produce_item();
        ProducerConsumer.insert(item);
    }
}
```

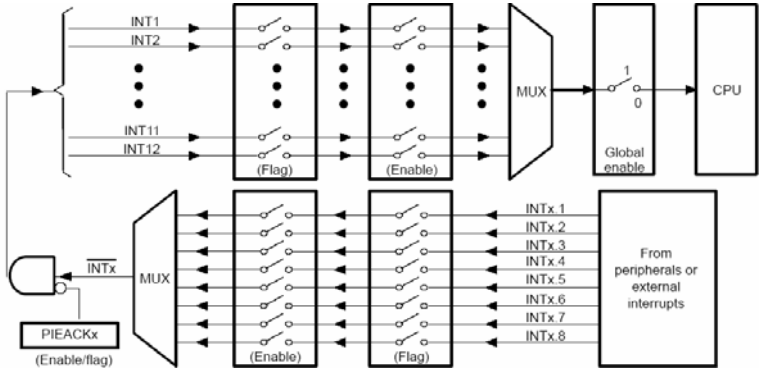
```
void Consumer() {
    while(TRUE) {
        int item = ProducerConsumer.remove();
        consume_item(item);
    }
}
```


5.4 Xử lý ngắt

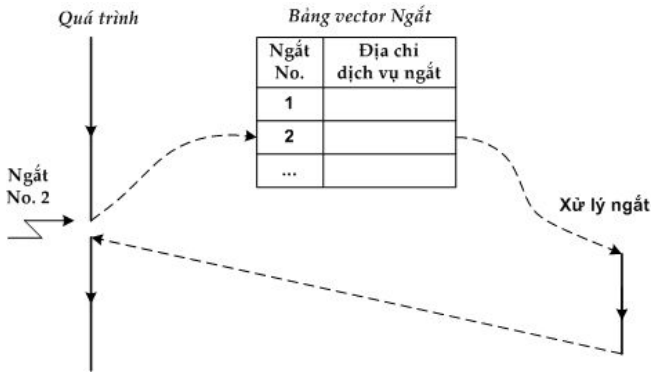
Tín hiệu điều khiển bộ VXL kích hoạt bởi một sự kiện tham gia trong quá trình hoạt động của hệ thống làm hệ thống ngừng và chuyển hướng thực thi được gọi là tín hiệu ngắt. Nó sẽ ngắt bộ VXL khỏi hoạt động mà nó đang thực thi và chuyển sang thực hiện một công việc khác phục vụ cho sự kiện kích hoạt ngắt tương ứng. Ví dụ như trong quá trình thu thập dữ liệu, VXL luôn phải chờ đợi thời điểm đón nhận dữ liệu và sẽ kích hoạt sự kiện ngắt CPU mỗi khi có dữ liệu xuất hiện để kịp thời ghi dữ liệu vào bộ nhớ. Sau khi hoàn thành, CPU phục hồi lại trạng thái của hệ thống và trở lại tiếp tục thực hiện chương trình từ thời điểm mà nó bị ngắt. Đối với bộ xử lý ngắt, nó sẽ phải thực hiện hai nhiệm vụ chính đó là: (1) Xác định có sự kiện ngắt và (2) nhận dạng sự kiện ngắt trước khi tác vụ phục vụ ngắt tương ứng được kích hoạt. Hình 5-6 mô tả một chu trình cơ bản thực hiện ngắt trong các hệ VXL/VĐK.



Hình 5-6: Chu trình thực hiện ngắt

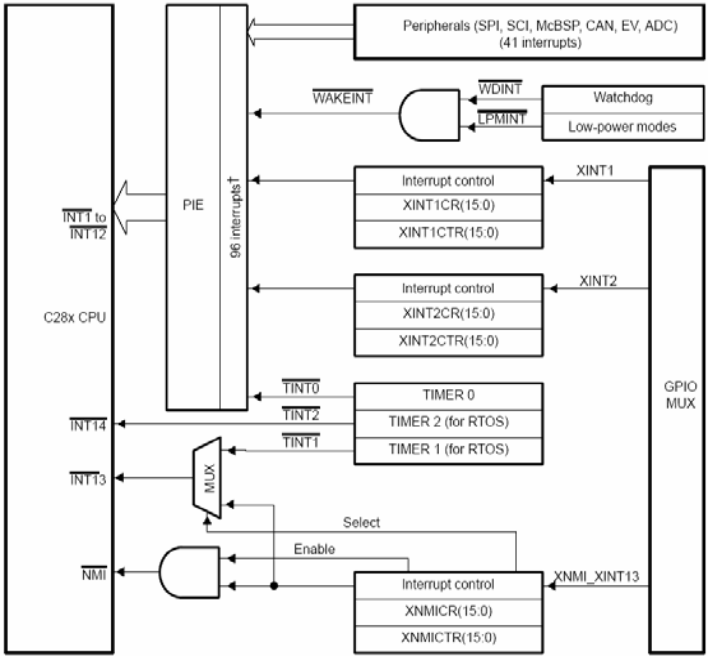


Hình 5-7: Ví dụ về cấu trúc phần cứng xử lý ngắt



Hình 5-8: Cơ chế thực hiện thủ tục ngắt

Thủ tục kích hoạt một tác vụ phục vụ sự kiện ngắt được mô tả như trong Hình 5-8. Thông thường người ta hay quan tâm nhiều đến đáp ứng của CPU với sự kiện ngắt và thời gian thực hiện tác vụ ngắt. Ở đây thời gian đáp ứng phụ thuộc và quyết định bởi tốc độ và khả năng xử lý của phần cứng còn thời gian thực hiện tác vụ ngắt chủ yếu quyết định bởi tác vụ ngắt đó dài hay ngắn và do chương trình quyết định.



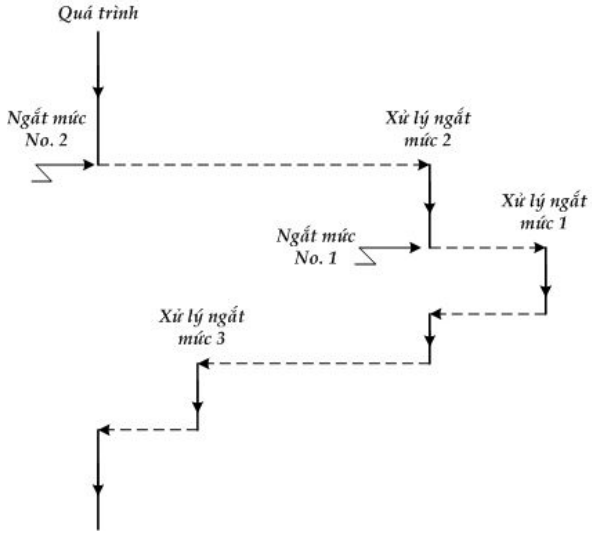
Hình 5-9: Ví dụ về nguồn ngắt (DSP TMS320C2812)

Các nguồn ngắt ngoài/cứng có thể được nhận dạng theo kiểu tín hiệu ngắt

- Theo sườn xung (ngắt được kích hoạt khi xuất hiện sườn xung dương tới chân nhận tín hiệu ngắt)
- Theo mức (ngắt được kích hoạt khi xuất hiện một tín hiệu xung mức tích cực tới chân nhận tín hiệu ngắt)

Một sự kiện ngắt cũng có thể được kích hoạt chỉ bởi một hoạt động đọc hoặc viết vào một thanh ghi thiết bị ngoại vi hoặc các thanh ghi điều khiển hoặc trạng thái.

Sự xung đột tranh chấp giữa các nguồn ngắt cùng xuất hiện tại một thời điểm có thể được giải quyết bằng mức độ ưu tiên hoặc kết nối cứng với bộ xử lý. Các nguồn ngắt ngoài có thể được tối giản việc xử lý bằng sự kết hợp với phần mềm và cùng chia sẻ các đường tín hiệu ngắt. Cơ chế thực hiện ngắt có sự tranh chấp và giải quyết bằng mức độ ưu tiên được mô tả như trong Hình 5-10.



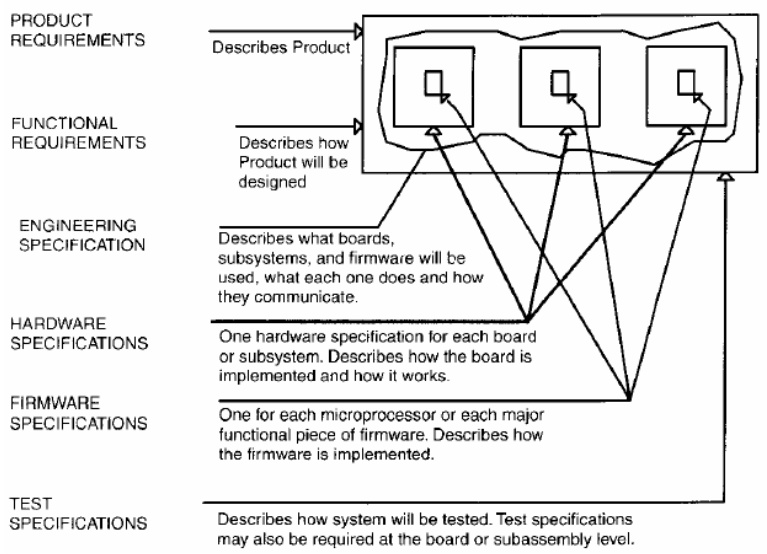
Hình 5-10: Cơ chế thực hiện ngắt theo mức độ ưu tiên

6 THIẾT KẾ HỆ NHÚNG: TỔ HỢP PHẦN CỨNG VÀ MỀM

6.1 Qui trình phát triển

Quá trình phát triển phần mềm nhúng thực hiện theo chu trình sau:

(1) Problem specification
(2) Tool/chip selection
(3) Software plan
(4) Device plan
(5) Code/debug
(6) Test
(7) Integrate



6.2 Phân tích yêu cầu

6.3 Mô hình hoá sự kiện và tác vụ

6.3.1 Phương pháp mô hình Petrinet

Năm 1962 *Carl Adam Petri* đã công bố phương pháp mô hình hình hoạ tác vụ hay quá trình theo sự phụ thuộc nhân quả đã được phổ cập rộng rãi và được biết tới như ngày này với tên gọi là mạng Petri.

Mạng Petri được sử dụng phổ biến để biểu diễn mô hình và phân tích các hệ thống có sự cạnh tranh trong quá trình hoạt động. Một hệ thống có thể hiểu là một tổ hợp của

nhiều thành phần và mỗi thành phần thì đều có các thuộc tính. Các thuộc tính đó có thể thay đổi và được đặc trưng bởi các biến trạng thái. Một chuỗi các trạng thái sẽ mô tả quá trình động của một hệ thống.

Mạng Petri thực sự là một giải pháp mô tả hệ thống động với các sự kiện rời rạc tác động làm thay đổi trạng thái của các đối tượng trong hệ thống theo từng điều kiện cụ thể trạng thái của hệ thống.

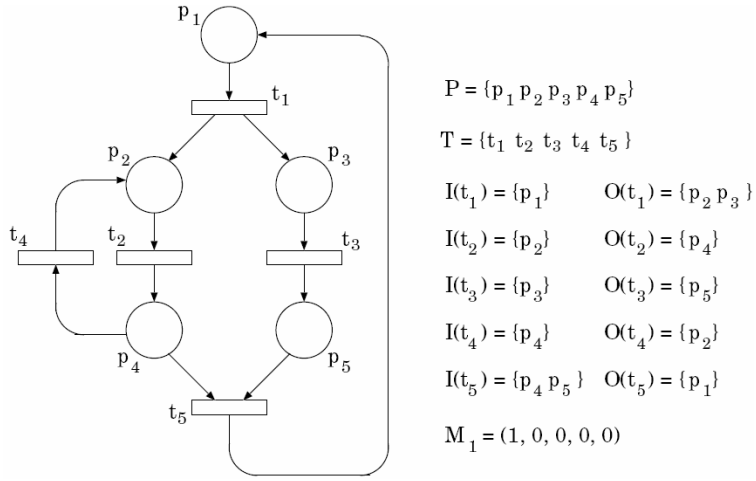
Mạng Petri được thiết lập dựa trên 3 thành phần chính: (1) Các điều kiện, (2) các sự kiện, và (3) quan hệ luồng. Các điều kiện có thể là thoả mãn hoặc không thoả mãn. Các sự kiện là có thể xảy ra hoặc không. Và quan hệ luồng mô tả điều kiện của hệ trước khi sự kiện xảy ra.

Các điều kiện đòi hỏi phải thoả mãn để một sự kiện xảy ra hoặc chuyển trạng thái thực hiện thì được gọi là điều kiện trước (*precondition*). Các điều kiện mà được thoả mãn khi một sự kiện nào đó xảy ra thì được gọi là điều kiện sau (*postcondition*).

6.3.2 Qui ước biểu diễn mô hình Petrinet

Trong qui ước biểu diễn hình hoạ thì mạng Petri sử dụng các vòng tròn để biểu diễn các điều kiện, các hộp để biểu diễn các sự kiện, và mũi tên biểu diễn quan hệ luồng. Một ví dụ minh hoạ về mạng Petri được mô tả trong Hình 6-1, trong đó:

- $P = \{p_1, p_2, ..., p_{np}\}$ là tập gồm np vị trí được biểu diễn trong mô hình (được mô tả bởi các vòng tròn);
- $T = \{t_1, t_2, ..., t_{nt}\}$ là tập gồm nt chuyển đổi trong tập chuyển đổi biểu diễn trong mô hình(được mô tả bởi các hình chữ nhật);
- I biểu diễn quan hệ đi vào chuyển đổi và được ký hiệu bởi đường mũi tên theo hướng từ các vị trí tới các chuyển đổi;
- O biểu diễn quan hệ đi ra khỏi chuyển đổi và được ký hiệu bởi các đường mũi tên theo hướng từ các chuyển đổi tới các vị trí;
- $M = \{m_1, m_2, ...m_{np}\}$ là dấu trạng thái của các chuyển đổi trong hệ thống. Các giá trị m_i là số các thẻ bài (được ký hiệu như các chấm tròn đen) chứa bên trong các vị trí p_i trong tập dấu M.



Hình 6-1: Ví dụ về một mô hình mạng Petri

Hệ thống động có thể được mô tả bởi mạng Petri nhờ sự chuyển dịch các thẻ bài trong các vị trí của hệ thống mô hình và tuân thủ theo luật sau:

- Một chuyển đổi được phép thực thi nếu tất cả các vị trí đi vào chuyển đổi đó chứa ít nhất một thẻ bài.
- Khi một chuyển đổi đã được thực thi xong (hoàn thành) thì một thẻ bài sẽ bị loại ra khỏi vị trí đi vào chuyển đổi đó đồng thời bổ sung thêm một thẻ bài vào các vị trí đầu ra tương ứng của chuyển đổi đó.

Các trạng thái động của hệ thống được mô tả bởi tập $\mathcal{R}(M)$ đánh dấu bởi các dấu trong tập M. Trong ví dụ trên có 5 phần tử dấu trong tập $\mathcal{R}$ lần lượt là M_1, M_2, M_3, M_4, M_5. Tương ứng lần lượt như sau:

$M_1 = (1,0,0,0,0)$:

$M_2 = (0,1,1,0,0)$:

$M_3 = (0,1,0,0,1)$:

$M_4 = (0,0,0,1,1)$:

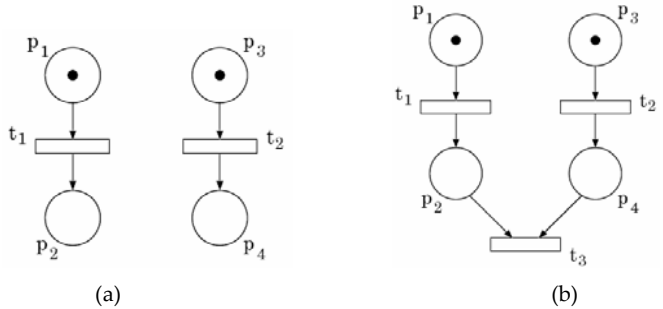
$M_5 = (0,0,1,1,0)$:

6.3.3 Mô tả các tình huống hoạt động cơ bản với Petrinet

- **Đồng hành (Song song) và đồng bộ**

Trong mô hình PN mô tả như trong Hình 6-2 (a), các chuyển đổi t_1 và t_2 được phép thực hiện đồng thời; hoạt động của chúng không ảnh hưởng đến nhau. Các hoạt động được mô hình bởi hai chuyển đổi thực hiện song song. Trong hệ thống dự phòng với độ tin cậy cao, mô hình này được sử dụng để biểu diễn hai thành phần C_1 và C_2 song song để đảm bảo hoạt động dự phòng; trong trường hợp này các vị trí p_1 và p_3 biểu diễn điều

kiện làm việc, các vị trí p_2 và p_4 biểu diễn điều kiện lỗi, t_1 và t_2 là các sự kiện lỗi trong các tác vụ C_1 và C_2 một cách tương ứng.

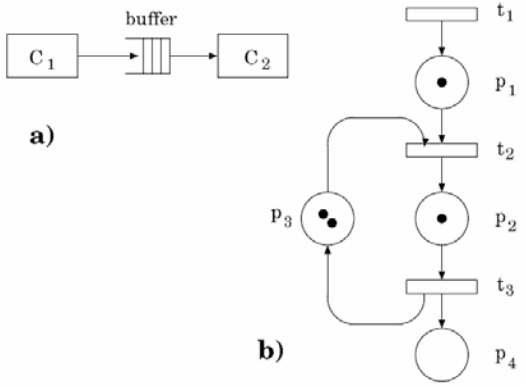


Hình 6-2: Mô hình Petrinet 2 hoạt động song song a) độc lập và b) đồng bộ

Trong hoạt động song song, các tác vụ hoàn toàn độc lập, tuy nhiên nếu các sự kiện đó cần phải kết thúc và là điều kiện để cho một chuyển đổi khác thì hoạt động đồng bộ có thể được thực hiện nhờ bổ sung một chuyển đổi t_3 như mô tả trong Hình 6-2 (b). Khi đó chuyển đổi t_3 cần thẻ bài đồng thời của cả p_2 và p_4.

- **Chia sẻ đồng bộ**

Một yếu tố đặc trưng trong hoạt động của hệ thống phân tán là thường phải chia sẻ một số tài nguyên hữu hạn. Sự thiếu thốn về tài nguyên làm hạn chế hoạt động của hệ thống trong quá trình xử lý thậm chí làm tắc nghẽn hệ thống. Việc mô hình và phân tích các hệ thống có hiện tượng tắc nghẽn là một tác vụ khó khăn trong hầu hết các quá trình mô hình có thể gặp phải.



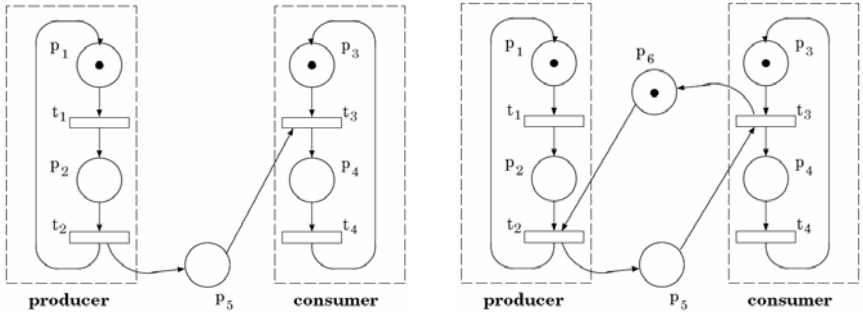
Hình 6-3: Hoạt động của bộ đệm với dung lượng hữu hạn

Để minh hoạ tình huống này, biểu diễn hoạt động của bộ đệm với dung lượng hữu hạn được mô tả bởi PN trong Hình 6-3. Vị trí p_3 mô hình số các vị trí bộ đệm còn trống và vị trí p_2 mô hình số vị trí đã được điền đầy; chú ý rằng tổng các thẻ bài chứa trong các vị

trí p_2 và p_3 luôn là hằng số (trong ví dụ này là 3). Chuyển đổi t_2 mô hình quá trình điền đầy một vị trí bộ đệm và hoàn thành nếu có ít nhất một vị trí bộ đệm còn trống cùng với thẻ bài chứa trong vị trí p_1 và p_3. Chuyển đổi t_3 được phép thực hiện nếu có ít nhất một vị trí bộ đệm đã được điền đầy. Khi hoàn thành chuyển đổi t_3, một thẻ bài sẽ được chuyển từ vị trí p_2 sang vị trí p_3.

- **Tuần tự**

Hoạt động tuần tự sẽ được mô tả và minh hoạ bởi hoạt động của bộ tạo và bộ sử dụng thông qua một bộ đệm. Bộ tạo sẽ sinh ra các đối tượng để đưa vào trong một bộ đệm và sẽ được lấy ra bởi bộ sử dụng. Quá trình sử dụng sẽ phải được thực hiện một cách tuần tự theo quá trình tạo ra đối tượng. Mô hình cho hoạt động này được diễn tả bởi PN như trong Hình 6-4 (a). Thẻ bài chứa trong vị trí p_1 có nghĩa là bộ tạo đã sẵn sàng thực hiện. Khi các chuyển đổi t_1 và t_2 hoàn thành thì một đối tượng được tạo ra (một thẻ bài tương ứng cũng sẽ được chuyển vào trong bộ đệm mô hình bởi vị trí p_5) và bộ tạo lại sẵn sàng trở lại. Nếu bộ sử dụng có nhu cầu tiêu thụ (được mô hình bởi thẻ bài chứa trong vị trí p_3) và đang có ít nhất một đối tượng trong bộ đệm thì một thẻ bài chứa trong vị trí p_5 sẽ được lấy đi và chuyển đổi t_3 sẽ hoàn thành.



(a) (b)

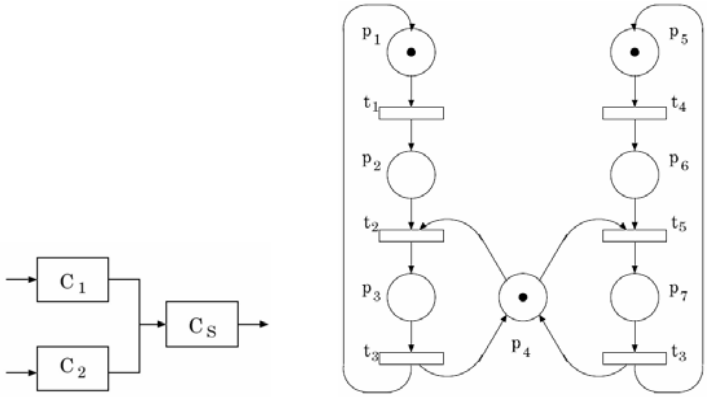
Hình 6-4: Hoạt động tạo và sử dụng với bộ đệm a) vô hạn và b) hữu hạn

Trong cách mô tả trong Hình 6-4 (a) thì việc tạo và sử dụng được thực hiện thông qua một bộ đệm với giả thiết là có dung lượng vô hạn. Trong thực tế thì các bộ đệm là hữu hạn, để mô tả hoạt động với bộ đệm loại này Hình 6-4 (b) được sử dụng. Vị trí p_6 mô hình các vị trí bộ đệm còn trống và vị trí p_5 mô hình các vị trí bộ đệm đã được điền đầy. Tổng số lượng các thẻ bài chứa trong các vị trí p_5 và p_6 phải luôn là hằng số. Nếu một thẻ bài được gán cho vị trí p_5 trong dấu khởi tạo thì bộ tạo sẽ không thể tạo thêm đối tượng chừng nào bộ sử dụng vẫn chưa tiêu thụ đối tượng trong bộ đệm.

- **Loại trừ xung đột**

Hai tác vụ C_1 và C_2 được phép làm việc song song và cùng chia sẻ tài nguyên C_S, nhưng không được truy nhập vào tài nguyên đồng thời. Giản đồ PN cho hoạt động này được mô tả như trong Hình 6-5. Các vị trí p_1 và p_5 biểu diễn các tác vụ C_1 và C_2 làm việc độc lập; vị trí p_2 và p_6 biểu diễn các yêu cầu của các tác vụ C_1 và C_2 một cách tương ứng khi

muốn truy nhập vào tài nguyên chia sẻ C_S; p_3 và p_7 biểu diễn C_S đang bị chiếm dụng bởi các tác vụ C_1 và C_2 một cách tương ứng. Vị trí p_4 mô tả quyết định xem tác vụ nào có thể truy nhập tài nguyên Cs và tránh các vị trí p_3 và p_7 bị đánh dấu đồng thời. Thực tế khi p_2 và p_6 được đánh dấu thì các chuyển đổi t_2 và t_5 xung đột. Việc hoàn thành một trong hai tác vụ sẽ khoá/cấm lẫn nhau. Việc hoàn thành chuyển đổi t_3 hoặc t_6 sẽ mô hình việc giải phóng nguồn tài nguyên chung (chuyển thẻ bài trở lại vị trí p_4) và trở về điều kiện làm việc bình thường.



Hình 6-5: Hoạt động loại trừ của hai tác vụ song song chia sẻ chung tài nguyên

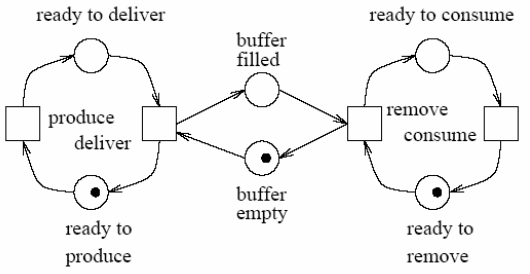
Để bắt đầu làm quen với nguyên lý biểu diễn mô hình hóa bằng mạng Petri chúng ta xét hoạt động của một hệ thống đồng bộ giữa hoạt động tạo và sử dụng một hạng mục (*item*) thông qua bộ đệm như được môt tả trong hình dưới.

Bộ tạo - *Producer*:

- ✓ Tạo ra hạng mục và
- ✓ bổ sung vào bộ đệm

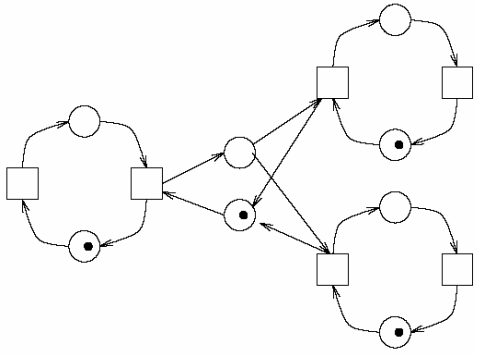
Bộ sử dụng (tiêu thụ) - *Consumer*:

- ✓ Lấy hạng mục ra khỏi bộ đệm và
- ✓ Sử dụng hạng mục



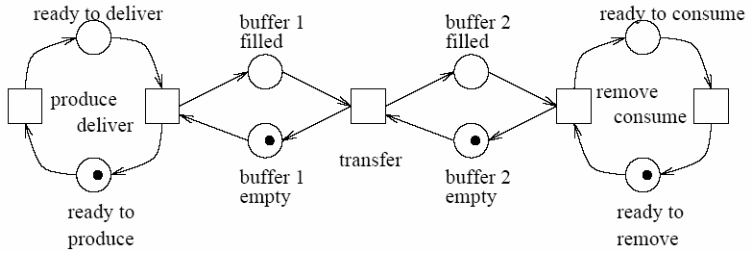
Hình 6-6: Hoạt động của hệ thống gồm 1 bộ tạo và 1 bộ sửdụng

Trong trường hợp có nhiều hơn một bộ sử dụng thì hệ thống được biểu diễn như sau:



Hình 6-7: Hoạt động của hệ thống gồm 1 bộ tạo và 2 bộ sử dụng

Hệ thống có 2 bộ đệm



Hệ thống vừa xét được mô hình hóa bởi điều kiện và sự kiện. Các điều kiện được mô tả bởi các vòng tròn và nếu điều kiện thỏa mãn thì khi đó vòng tròn sẽ được biểu diễn với một chấm tròn nằm trong tương ứng với một thẻ bài (*token*).

Sự kiện được ký hiệu bởi các hộp hình chữ nhật. Với mỗi một sự kiện thì sẽ tồn tại

- một tập các điều kiện trước và được nhận biết bởi các mũi tên đi vào các sự kiện từ các điều kiện đó và
- một tập các điều kiện sau được nhận biết bởi các mũi tên đi ra khỏi các sự kiện và đi vào các điều kiện đó.

Một sự kiện có thể xảy ra (được thực thi) khi và chỉ khi

- ✓ tất cả các điều kiện trước tương ứng được thỏa mãn (nhận được thẻ bài) và
- ✓ tất cả các điều kiện sau tương ứng chưa được thỏa mãn.

Nếu một sự kiện xảy ra thì

- ✓ tất cả các điều kiện trước tương ứng sẽ bị xóa bỏ (*reset*) và
- ✓ tất cả các điều kiện sau tương ứng sẽ được thiết lập (*set*).

Với loại mạng biểu diễn như trên người ta gọi là mạng Petri cơ bản (*Elementary Net*) và ký hiệu tắt là EN.

Để thuận tiện và đơn giản hóa trong việc biểu diễn người ta có thể sử dụng các mũi tên có thêm trọng số nguyên để mô tả hệ thống có chung nhiều điều kiện trước và sau tương ứng cùng với một sự kiện hoặc điều kiện. Đặc biệt khi số hạng mục trao đổi giữa bộ tạo và bộ sử dụng lớn hơn 1. Với loại mạng như vậy người ta phân loại và gọi là mạng Petri Chuyển đổi/Vị trí (*Transitions*/*Places*) ký hiệu tắt là P/T-net.

Cũng tương tự như EN, P/T-net bao gồm:

- Các vị trí được ký hiệu và mô tả bởi các vòng tròn: Các vị trí có thể chứa một số nguyên dương các thẻ bài.
- Các chuyển đổi được mô tả bởi các hình chữ nhật: Các chuyển đổi sẽ lấy đi hoặc thêm vào số thẻ bài từ hoặc tới một vị trí.
- Các mũi tên kết nối trực tiếp giữa các vị trí và chuyển đổi: Các mũi tên có kèm theo các trọng số tương ứng với số lượng thẻ bài mà nó có thể được lấy ra hoặc thêm vào trong các vị trí.

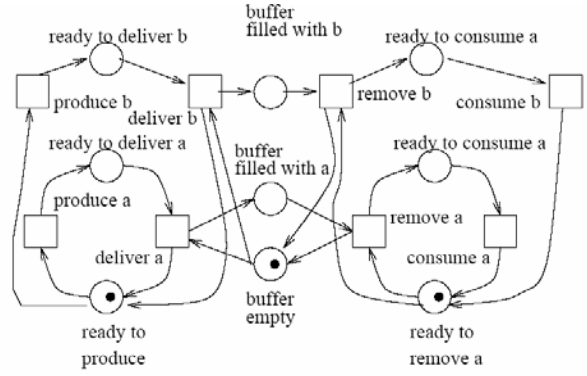
Qui ước: Một tập vị trí kết nối với chuyển đổi thông qua một mũi tên trực tiếp theo chiều từ vị trí tới chuyển đổi được gọi là tập các tiền chuyển đổi. Ngược lại, một tập vị trí kết nối với chuyển đổi thông qua một mũi tên trực tiếp theo chiều ngược từ vị trí tới chuyển đổi thì được gọi là tập các hậu chuyển đổi.

Một chuyển đổi có thể xảy ra (thực hiện) khi và chỉ khi tất cả các vị trí trong tập tiền vị trí chứa một số lượng tối thiểu thẻ bài như được định nghĩa bởi các trọng số của các mũi tên tương ứng.

Khi một chuyển đổi được thực thi nó sẽ

- ✓ loại bỏ bớt số thẻ bài từ tập tiền vị trí bằng đúng số lượng đã được định nghĩa cho các trọng số của các mũi tên tương ứng và
- ✓ cộng thêm vào số lượng các thẻ bài vào tập hậu vị trí đúng bằng với trọng số của các mũi tên tương ứng.

Ví dụ biểu diễn mô tả một hoạt động hệ thống với 2 hạng mục cần đồng bộ giữa bộ tạo và bộ sử dụng.



Hình 6-8: Hoạt động đồng bộ với hai hạng mục

Để có thể biểu diễn hệ thống một cách khoa học và logic cần có một định nghĩa đầy đủ mô tả bởi mạng Petri.

- **Mạng điều kiện/ sự kiện**

Định nghĩa: $N = (C, E, F)$ được gọi là một mạng nếu và chỉ nếu nó thoả mãn các thuộc tính sau:

☑ C và E là các tập độc lập và $C \cap E \neq \varnothing$.

☑ $F \subseteq (E \times C) \cup (C \times E)$ là quan hệ nhị phân và được gọi là quan hệ luồng.

C được gọi là các điều kiện và E được gọi là các sự kiện.

Định nghĩa: Cho một mạng N và $x \in (C \cup E)$. $\bullet x := \{y \mid yFx\}$ được gọi là tập các điều kiện trước của x và $x\bullet := \{y \mid xFy\}$ được gọi là điều kiện sau của x.

Hay nói cách khác là một điều kiện cần phải được thoả mãn để một sự kiện nào đó xảy ra thì được gọi là điều kiện trước và một điều kiện được thoả mãn sau khi một sự kiện nào đó xảy ra thì được gọi là điều kiện sau của sự kiện đó.

Định nghĩa: Cho một tập $(c, e) \in C \times E$

(c, e) được gọi là một vòng lặp nếu $cFe \wedge eFc$

Mạng N được gọi là thuần nhất nếu F không chứa bất kỳ một vòng lặp nào.

Định nghĩa : Một mạng được gọi là đơn giản nếu không có bất kỳ hai chuyển đổi t_1, t_2 nào có cùng tập các điều kiện trước và các điều kiện sau.

Các mạng mà không chứa bất kỳ phần tử tách biệt nào cũng như không có thêm bất kỳ một hạn chế nào thì được gọi là mạng điều kiện /sự kiện.

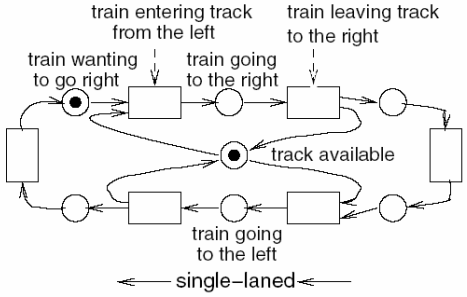
- **Mạng chuyển đổi/vị trí**

Trong các mạng điều kiện/sự kiện chỉ chứa nhiều nhất là một *token* cho mỗi một điều kiện. Để hạn chế điều này tức là một điều kiện có thể chứa nhiều *token* và người ta gọi

là mạng chuyển đổi/vị trí. Các vị trí tương ứng với các điều kiện và các chuyển đổi tương ứng với các sự kiện trong mạng điều kiện/sự kiện.

Số lượng *token* cho mỗi một điều kiện được gọi là *Marking*. Về mặt toán học, *Marking* chính là một ánh xạ toán học cho phép chuyển một tập các vị trí vào một tập các số tự nhiên được mở rộng bởi các biểu tượng đặc biệt ∞.

Ví dụ : Mô tả chương trình điều khiển luồng tàu điện bằng mạng Petrinet điều kiện/sự kiện để tránh trường hợp xung đột trên một đường ray theo hai hướng tàu chạy.



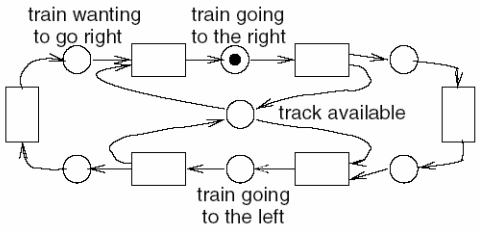
Các điều kiện :

- Tàu muốn vào đường ray theo chiều sang phải.
- Tàu đang chuyển động trên đường ray theo chiều phải.
- Tàu thoát ra khỏi đường ray theo chiều phải.
- Tàu muốn vào đường ray theo chiều sang trái.
- Tàu đang chuyển động trên đường ray theo chiều trái.
- Tàu thoát ra khỏi đường ray theo chiều trái.

Các sự kiện :

- Tàu vào đường ray từ chiều bên trái
- Tàu rời khỏi đường ray theo chiều phải
- Tàu rời đường ray
- Tàu vào đường ray từ chiều bên phải
- Tàu rời khỏi đường ray theo chiều trái

Token : Đường ray sẵn sàng cho tàu vào theo một trong hai chiều



6.3.4 Ngôn ngữ mô tả phần cứng (VHDL)

VHDL (*Very High Speed Integrated Circuit Hardware Description Lanuage*) là một ngôn ngữ chung để mô tả các thiết kế phần cứng ở mức phần tử logic cơ bản cấu thành nên hệ thống và đã được phát triển bởi tổ chức quốc phòng Mỹ. Mục đích chính là để thuận tiện cho việc trao đổi dữ liệu thiết kế phần cứng theo một định dạng chuẩn mà mọi người có thể hiểu và thông dịch, tạo điều kiện thuận lợi trong việc phối hợp hay hợp tác trong các dự án thiết kế. Đặc biệt nó rất thuận tiện trong việc chuyển đổi hay tổng hợp biên dịch thành một dạng ngôn ngữ thực thi phần cứng thực. Điều này rất khó thực hiện bằng các ngôn ngữ bậc cao như C nhưng với VHDL điều này chính là ưu điểm nổi bật và là thế mạnh trong việc mô hình hoá hệ thống, mô tả một cách chi tiết các phần tử cứng cấu thành tham gia trong hệ thống.

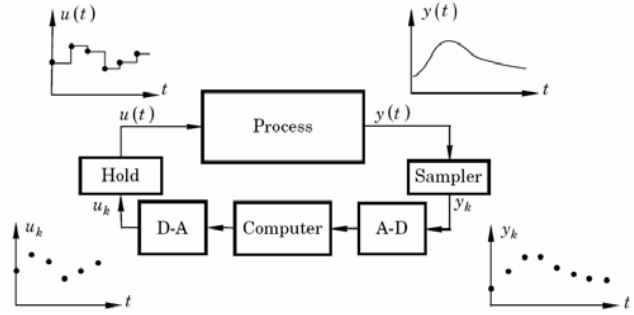
VHDL là một chuẩn IEEE (Std-1076) đã được sự hỗ trợ bởi rất nhiều nhà cung cấp phát triển phần cứng. Ứng dụng một cách chuyên nghiệp ngôn ngữ này là phục vụ cho việc mô tả các mạch ASICs phức hợp, chế tạo thực thi các mạch FPGA...

Ngôn ngữ VHDL có thể đọc hiểu khá dễ dàng với cấu trúc cú pháp rõ ràng gần giống như ngôn ngữ *Visual Basic* và *Pascal*. Nó có thể phát huy được thế mạnh về cú pháp để định nghĩa xây dựng kiểu dữ liệu mới và hỗ trợ cho việc lập trình theo nhóm. Với xu thế hiện nay các nhóm phát triển có thể thực thi với điều kiện cách xa nhau về khoảng cách địa lý, vì vậy việc phối hợp và thiết kế theo nhóm là rất cần thiết.

„Tom Cantrell recently wrote that the future is bright for FPGAs, which will play a large role in mainstream applications ("More Flash, Less Cash," Circuit Cellar, 178, May 2005). I agree with Tom, but I'll go further and predict that VHDL will become the premier technology used to define FPGA content either as output from design tools or with direct programming. In combination with VHDL, FPGAs provide a lowcost approach to defining complex hardware designs that were inconceivable only a few decades ago. Perhaps most importantly, using VHDL to define hardware is fun…"

6.4 Thiết kế phần mềm điều khiển

6.4.1 Mô hình thực thi bộ điều khiển nhúng



Hình 6-9: Hệ thống điều khiển số

Để thực thi một bộ điều khiển số trên thiết bị vật lý thực phải đòi hỏi xét xem bộ điều khiển với mô hình hàm truyền đã cho có thể hiện thực hóa được không. Điều kiện phải xét thực ra là để đảm bảo rằng không có đầu ra nào của hệ thống lại xuất hiện trước khi có tín hiệu vào. Hay nói cách khác hệ thống xây dựng phải tuân thủ tính nhân quả.

Nếu khai triển hàm truyền của bộ điều khiển số được mô tả ở dạng tổng quát

$$G_R(z) = \frac{b_0 + b_1 z^{-1} + \cdots + b_m z^{-m}}{a_0 + a_1 z^{-1} + \cdots + a_n z^{-n}} \tag{1.5}$$

thành chuỗi lũy thừa theo z thì nó phải không được phép chứa bất kỳ phần tử nào chứa lũy thừa dương của z. Hay nói cách khác là bộ điều khiển được mô tả như (1.5) phải có bậc ≤ 0 tức là bậc của tử số phải nhỏ hơn hoặc bằng bậc của mẫu số ($n \geq m$).

Sau khi đã thiết kế được bộ điều khiển số thì việc còn lại là lập trình và nạp vào các bộ điều khiển vật lý khả trình. Thực chất quá trình này là thực thi hàm truyền của bộ điều khiển số bằng lập trình số trên các bộ điều khiển vật lý đã có. Ở đây chúng ta sẽ chủ yếu quan tâm đến việc triển khai để chuẩn bị cho bước lập trình các hàm truyền của bộ điều khiển số. Xuất phát từ mô tả hàm truyền dạng tổng quát của bộ điều khiển số

$$G_R(z) = \frac{U(z)}{E(z)} = \frac{b_0 + b_1 z^{-1} + \cdots + b_m z^{-m}}{a_0 + a_1 z^{-1} + \cdots + a_n z^{-n}} \tag{1.6}$$

trong đó, $a_0 \neq 0$ nếu $b_0 \neq 0$; m và n là các số nguyên dương.

Có thể triển khai để thực thi một hàm truyền của bộ điều khiển số theo 3 cách như sau:

- **Triển khai lập trình số trực tiếp**

Để triển khai lập theo phương pháp lập trình trực tiếp thì hàm truyền bộ điều khiển đã cho biểu diễn trong miền z phải được chuyển đổi về dạng hàm truyền rời rạc

$$a_0 u^*(t) + \sum_{k=1}^{n} a_k u^*(t - kT) = \sum_{k=0}^{m} b_k e^*(t - kT) \tag{1.7}$$

Từ đẳng thức (1.7) dễ dàng tính ra được giá trị của đầu ra $u^*(t)$ của bộ điều khiển số đã cho theo các giá trị hiện tại và quá khứ của đầu vào $e^*(t)$ cũng như các giá trị quá khứ của chính nó

$$u^*(t) = \frac{1}{a_0}\sum_{k=0}^{m} b_k e^*(t-kT) - \frac{1}{a_0}\sum_{k=1}^{n} a_k u^*(t-kT) \quad (1.8)$$

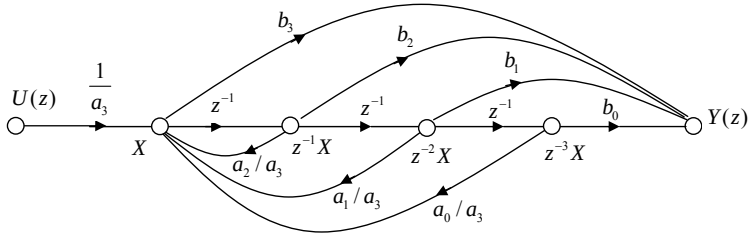
Để thực hiện bộ điều khiển này yêu cầu phải lưu trữ các giá trị quá khứ của đầu vào và đầu ra của bộ điều khiển. Với bộ điều khiển đã cho yêu cầu phải có $n+m$ giá trị cần phải lưu trữ hay nói cách khác cần phải có $n+m$ phần tử lưu trữ.

Một phương pháp khác để triển khai lập trình trực tiếp là sử dụng cơ chế tách trực tiếp đầu vào và đầu ra của bộ điều khiển theo một biến trung gian X(z). Không mất tính tổng quát nếu chúng ta nhân cả tử và mẫu của hàm truyền bộ điều khiển số đã cho với một biến X(z). Từ đó rút ra được hàm truyền của đầu vào E(z) theo X(z) và hàm truyền của đầu ra U(z) theo X(z). Phương pháp này thực hiện như sau:

$$U(z) = \frac{1}{a_0}(b_0 + b_1 z^{-1} + \cdots + b_m z^{-m}) X(z) \quad (1.9)$$

$$X(z) = \frac{1}{a_0}E(z) - \frac{1}{a_0}(a_1 z^{-1} + a_2 z^{-2} + \cdots + a_n z^{-n}) X(z) \quad (1.10)$$

Theo phương pháp này yêu cầu số phần tử lưu trữ chính bằng giá trị n, bằng bậc của đa thức mẫu số trong hàm truyền bộ điều khiển số đã cho. Từ các đẳng thức (1.9) và (1.10) ta cũng dễ dàng xây dựng được giản đồ trạng thái mô tả hàm truyền của bộ điều khiển số (giả thiết $m = n = 3$).



Hình 6-10: Giản đồ trạng thái của hệ thống số

- **Triển khải lập trình số ghép tầng**

Cách triển khai này yêu cầu chuyển đổi bộ điều khiển về dạng tích của các hàm truyền đơn giản để có thể dễ dàng thực hiện bằng các chương trình đơn giản. Hay nói cách khác bộ điều khiển số đã cho là kết quả ghép tầng của nhiều bộ điều khiển nhỏ.

- **Triển khai lập trình số song song**

Bộ điều khiển đã cho sẽ được tách ra thành tổng của các bộ điều khiển đơn giản và có thể thực hiện lập trình song song cho các bộ điều khiển đó.

6.4.2 Ví dụ triển khai bộ điều khiển PID số

Xấp xỉ hoá thành phần vi tích phân

Có 3 phương pháp xấp xỉ gián đoạn phổ biến áp dụng cho các thành phần tích phân: vượt trước (*forward*), vượt sau (*backward*), và *trapezoidal*.

- **Xấp xỉ sai phân vượt trước**

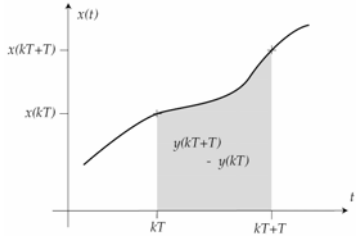
$$y_f(kT+T) - y_f(kT) = Tx(kT) \quad (1.11)$$

Áp dụng chuyển đổi z cho (1.11) ta thu được

$$\frac{y_f(z)}{x(z)} = \frac{T}{z-1} \quad (1.12)$$

Dó đó xấp xỉ hoá tích phân sẽ là:

$$\frac{1}{s} \approx \frac{T}{z-1} \quad (1.13)$$

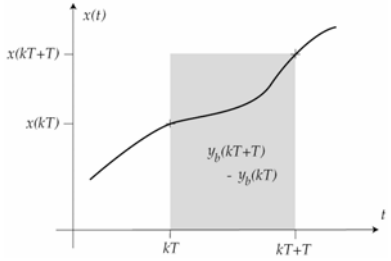


Hình 6-11: Xấp xỉ sai phân vượt trước

- **Xấp xỉ sai phân vượt sau**

Tương tự như sai phân vượt trước ta có xấp xỉ tích phân như sau:

$$\frac{1}{s} \approx \frac{Tz}{z-1} \quad (1.14)$$

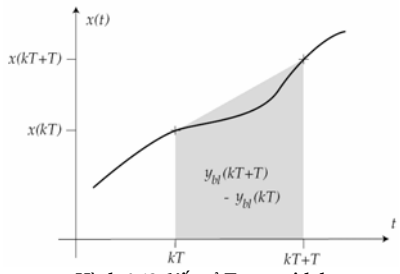


Hình 6-12: Xấp xỉ sai phân vượt sau

- **Xấp xỉ Trapezoidal**

Phép xấp xỉ tích phân thu được sẽ là:

$$\frac{1}{s} \approx \frac{T}{2}\frac{z+1}{z-1} \quad (1.15)$$



Hình 6-13: Xấp xỉ Trapezoidal

Đẳng thức lý tưởng mô tả bộ điều khiển PID

$$u(t) = u_P(t) + u_I(t) + u_D(t)$$
$$= K\left[e(t) + \frac{1}{T_I}\int_0^t e(\tau)d\tau + T_D\frac{de(t)}{dt}\right] \quad (1.16)$$

trong đó, K là hệ số khuếch đại, T_I là hằng số thời gian tích phân, T_D là hằng số thời gian vi phân.

Trong trường hợp chu kỳ trích mẫu nhỏ, đẳng thức (1.16) có thể được chuyển sang dạng đẳng thức sai phân bằng phương pháp rời rạc hoá. Trong đó, thành phần vi phân có thể được xấp xỉ như phép tính sai phân bậc nhất và thành phần tích phân được xấp xỉ dạng vượt trước. Bằng phép rời rạc này ta thu được đẳng thức mô tả bộ điều khiển PID số như sau:

$$u(k) = K_P\left[e(k) + \frac{T_s}{T_I}\sum_{i=0}^{k-1} e(i) + \frac{T_D}{T_s}\big(e(k) - e(k-1)\big)\right] \quad (1.17)$$

Từ đẳng thức (1.17) ta dễ dàng nhận thấy rằng để thực thi bộ điều khiển PID cần thông tin của tất cả các sai lệch *e* trong quá khứ. Để thuận tiện cho việc thực hiện lập trình, dạng đệ qui sẽ phù hợp hơn và có thể rút ra từ (1.17) như sau:

$$u(k-1) = K\left[e(k-1) + \frac{T_s}{T_I}\sum_{i=0}^{k-2} e(i) + \frac{T_D}{T_s}\big(e(k-1) - e(k-2)\big)\right] \quad (1.18)$$

Từ (1.17) và (1.18) ta rút ra được *algorithm* điều khiển của PID số:

$$u(k) - u(k-1) = a_0 e(k) + a_1 e(k-1) + a_2 e(k-2) \quad (1.19)$$

trong đó, $a_0 = K\left(1 + \frac{T_D}{T_s}\right)$, $a_1 = -K\left(1 + 2\frac{T_D}{T_s} - \frac{T_s}{T_I}\right)$, $a_2 = K\frac{T_D}{T_s}$

Mô hình bộ điều khiển ở dạng hàm truyền ta có:

$$G_{PID} = K_P + K_I\frac{1}{s} + K_D s \quad (1.20)$$

trong đó, thành phần tích phân có thể xấp xỉ theo một trong ba cách như mô tả trong phần 6.1, thành phần vi phân có thể được xấp xỉ như sau:

$$\left.\frac{de(t)}{dt}\right|_{t=T} = \frac{e(kT) - e(kT-T)}{T} \quad (1.21)$$

từ (1.21) có thể xấp xỉ hàm truyền thành phần vi phân

$$G_D(z) = K_D\frac{z-1}{Tz} \quad (1.22)$$

Như vậy hàm truyền của bộ điều khiển PID số có thể được xấp xỉ theo một trong 3 dạng như sau:

- *Xấp xỉ vượt trước*:

$$G_{PID} = \frac{(K_P T + K_D)z^2 + (K_I T^2 - K_P T - 2K_D)z + K_D}{Tz(z-1)} \quad (1.23)$$

- *Xấp xỉ vượt sau*:

$$G_{PID} = \frac{(K_P T + K_D + K_I T^2)z^2 - (K_P T + 2K_D)z + K_D}{Tz(z-1)} \quad (1.24)$$

- *Xấp xỉ Trapezoidal*:

$$G_{PID} = \frac{(2K_P T + K_I T^2 + 2K_D)z^2 + (K_I T^2 - 2K_P T - 4K_D)z + 2K_D}{2Tz(z-1)} \quad (1.25)$$

TÀI LIỆU THAM KHẢO

[1] Peter Marweden. *Embedded Systems Design:* Springer, 2006.

[2] Michael Barr. *Programming Embedded Systems in C and C++*. O'Reilly, 1999.

[3] Jack Ganssle. *The Art of Designing Embedded Systems*. Newnes, 1999.

[4] Stuart R.Ball. *Embedded Microprocessor Systems*. Newnes, 2002

[5] Qing Li and Carolyn Yao. *Real-time Concepts for Embedded Systems*, CMP Books, 2003

[6] Olli S., Jaakko A.. *Embedded Systems*, Lecture Notes, Helsinki University of Tech. , 2006.

[7] Lothar Thiele. *Embedded Systems*, Lecture Notes, Swiss Federal Institute of Tech. , 2006.

[8] Don Morgan. *Numerical Methods: Realtime and Embedded Systems Programming*. M&T, 1992.

[9] Jerrry Lueke. *Analog and Digital Circuits for Electronic Control System Application.* Newnes, 2005.

[10] Adrea Bobbio. *System Modelling with Petri Nets*. A.G. Colombo, 1990.

[11] Linda Null and Julia Lobur. *The essentials of computer Organization and Architecture*: Jones and Bartlett Publishers, 2003.

[12] Hennessy, J. L., & Patterson, D. A. *Computer Architecture: A Quantitative Approach*, San Francisco: Morgan Kaufmann, 1990.

[13] Sen M. Kuo, Bob H. Lee, Wenshun Tian. *Real-time Digital Signal Processing: Implementations and Applications*, John Wiley & Son, 2006.

[14] Kuo. *Digital Control Systems*, Oxford, 2005.