Giới thiệu ARM

---Φ---

Người viết: Bùi Trung Hiếu
Webmaster: Khoa học và tuổi trẻ

Lời mở đầu:

Ứng dụng cho các hệ thống nhúng hiện nay ngày càng trở nên phức tạp, không phải đơn giản chỉ là điều khiển một chốt đèn giao thông định thời, đếm số người ra vào cửa, điều khiển động cơ ON-OFF, hiển thị một câu thông báo trên LCD ..v.vv.. xu thế tất yếu, các nhân điều khiển cần có cấu hình mạnh hơn, đáp ứng thời gian thực tốt hơn các nhân điều khiển 8bit đang dùng!

Như đã phân tích ở một số bài viết trước, ở ta đa phần vẫn sử dụng nhân điều khiển 8 bit cho các ứng dụng, và như thế, vô tình đã thu hẹp khả năng điều khiển các hệ thống nhúng. Chính sự hạn chế về dung lượng bộ nhớ chương trình-dữ liệu cũng đã ảnh hưởng không ít tới phạm vi ứng dụng của nó. Khi dùng vi điều khiển 8bit làm một bộ điều khiển PID kinh điển <cho động cơ chẳng hạn> cũng là một cố gắng không nhỏ từ người lập trình, đừng nói đến việc dùng nó vào các ứng dụng dựa trên cơ sở lý thuyết điều khiển hiện đại, đòi độ chính xác cao, đáp ứng thời gian thực tốt![1] Tất nhiên, với vi điều khiển 8bit, bạn vẫn có thể dùng để điều khiển mờ lò nhiệt, hay những đối tượng có mức quán tính lớn![2] Đi kèm với việc điều khiển cố gắng ấy là giải thuật sẽ phức tạp lên. Bạn còn hoài nghi? Cứ thử đi, ban đầu, bạn thu nhỏ giải thuật điều khiển bằng các lý thuyết toán học, sự cố gắng hạn chế dung lượng bộ nhớ chương trình sẽ làm tăng thời gian xử lý và cần nhiều ô nhớ trung gian. Tuy nhiên, nếu bạn sử dụng các nhân 8bit có tần số dao động lớn thì kết quả cũng chấp nhận được! Nếu nhân 8bit ấy hỗ trợ tính toán số thực thì kết quả còn tốt hơn!

Không thể nói rằng với nhân điều khiển 8bit, ta chẳng làm nên trò trống gì, vì như vậy, đã phủ nhận kết quả lâu nay của phần đông dân lập trình vi điều khiển-nhất là đối với sinh viên ta, luôn năng động, sáng tạo! Chỉ có thể nói rằng, các kết quả ấy luôn bị hạn chế khi ta ứng dụng vào công nghệ cao: truyền thông đa phương tiện, xử lý âm thanh, hình ảnh, các thiết bị hỗ trợ cá nhân <PDA>, các ứng dụng trong mobile robot linh hoạt, tự hành và 'biết ứng xử'..vv… Yêu cầu những hệ thống cần sự linh động, tiêu tốn ít năng lượng, nhỏ gọn, nhưng cấu hình mạnh và tính năng phức tạp luôn được đặt ra. Nhu cầu thị trường cần, người làm kĩ thuật không thể bỏ qua trong thời buổi cạnh tranh hiện nay!

Và như thế, mời bạn cùng tôi đi vào khám phá những cõi mới! Những bước đi đầu tiên bao giờ cũng có thể vấp ngã! Và thế, tôi cần sự giúp sức, phê bình và đóng góp của mọi người, biết đâu, khi nào đấy, tôi có lỡ đi vào ngõ cụt, còn có tiếng kêu và vòng tay đón về đất mẹ!

Thân chào!

---

[1] Ví dụ đưa ra tôi chưa tính đến sai số do sensor.

[2] Theo tính toán ban đầu của tôi thì ta dư sức dùng một nhân điều khiển 8 bit cho việc điều khiển mờ lò nhiệt với 3 tập biến ngôn ngữ ngõ vào và 2 ngõ ra tuần tự, mỗi biến ngôn ngữ có 7 cấp điều khiển với thời gian lấy mẫu khoảng 0.5 giây

Các đề mục chính:

Danh mục các hình vẽ:

Danh mục các bảng:

A. *ARM - Đôi nét về lịch sử hình thành và phát triển:*

Ngày 26/4/1985, mẫu sản phẩm ARM đầu tiên sản xuất tại công ty kĩ thuật VLSI, SanJose, bang Califonia được chuyển tới trung tâm máy tính Acorn ở Cambridge, Anh Quốc. Một vài giờ sau, chương trình thử nghiệm đầu tiên thành công và họ đã khui sâm banh ăn mừng!

Nửa thập niên sau đó, ARM được phát triển rất nhanh chóng để làm nhân máy tính để bàn của Acorn, nền tảng cho các máy tính hỗ trợ giáo dục ở Anh. Trong thập niên 1990, dưới sự phát triển của Acorn Limited, ARM đã thành một thương hiệu đứng đầu thế giới về các ứng dụng sản phẩm nhúng đòi hỏi tính năng cao, sử dụng năng lượng ít và giá thành thấp.

Chính nhờ sự nổi trội về thị phần đã thúc đẩy ARM liên tục được phát triển và cho ra nhiều phiên bản mới. Những thành công quan trọng trong việc phát triển ARM ở thập niên sau này:

- Giới thiệu ý tưởng về định dạng các chỉ lệnh được nén lại (thumb) cho phép tiết kiệm năng lượng và giá thành ở những hệ thống nhỏ.
- Giới thiệu họ điều khiển ARM9, ARM10 và 'Strong ARM'
- Phát triển môi trường làm việc ảo của ARM trên PC.
- Các ứng dụng cho hệ thống nhúng dựa trên nhân xử lý ARM ngày càng trở nên rộng rãi.

Hầu hết các nguyên lý của hệ thống trên chip (Systems on chip-SoC) và cách thiết kế bộ xử lý hiện đại được sử dụng trong ARM, ARM còn đưa ra một số khái niệm mới <như giải nén động các dòng lệnh>. Việc sử dụng 3 trạng thái nhận lệnh-giải mã-thực thi trong mỗi chu kì máy mang tính quy phạm để thiết kế các hệ thống xử lý thực. Do đó, nhân xử lý ARM được sử dụng rộng rãi trong các hệ thống phức tạp.

B. *Sơ lược về thiết kế nhân điều khiển:*

Việc thiết kế các nhân điều khiển đa dụng vẫn là mong ước của hầu hết kĩ sư. Điều đó, đòi hỏi những nghiên cứu tỉ mỉ, chi tiết, đầy mâu thuẫn và cả sự thỏa hiệp. Trong phần này, ta sẽ bàn về một số khái niệm xung quanh việc thiết kế phần cứng cho nhân điều khiển, thiết kế tập lệnh, cấu trúc máy tính số thực thi lệnh dạng *stored-program*.

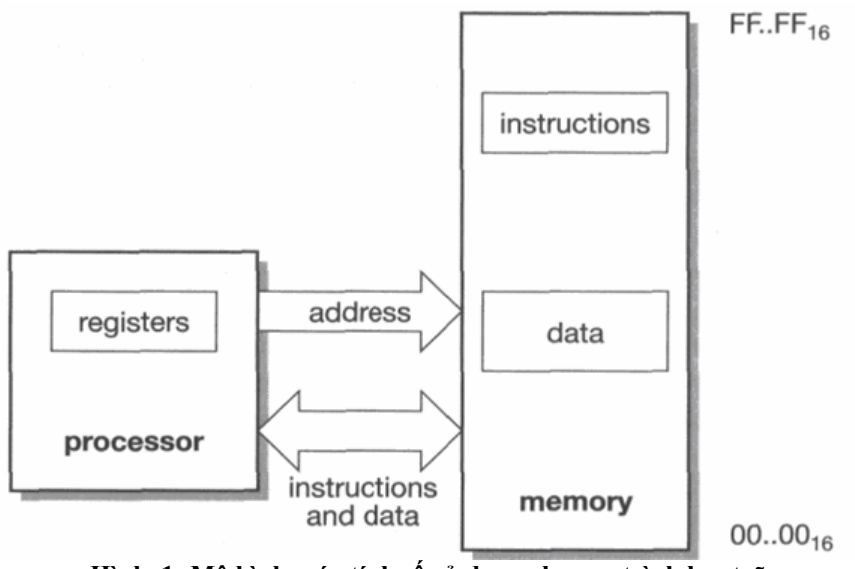
B.I. *Đôi nét về thiết kế phần cứng:*

Máy tính là một thiết bị hoạt động ở tốc độ rất cao. Những vi xử lý hiện đại có thể bao gồm đến vài triệu transistor <hiện tại đang sử dụng công nghệ 90nm> và chúng đóng ngắt đến hàng trăm triệu lần trong một giây. Tất cả chúng (Transistor) đều phải hoàn hảo, chẳng có Transistor nào được phép sai lệch hoặc hoạt động ngẫu nhiên, một sai lệch nhỏ của dù chỉ 1 Transistor cũng có thể gây sai lệch cho cả hệ thống! Bạn cứ tưởng tượng-so sánh một đội quân tinh nhuệ thiện chiến được huấn luyện công phu dàn quân khắp chiến tuyến, bất kì 1 vị trí nào không chiến đấu được cũng làm cả đội quân thất thủ! <hoặc đơn giản hơn, bạn cứ nghĩ là mình phải học qua 5-6 môn tiên quyết mới có thể làm luận văn tốt nghiệp-nếu rớt bất kì môn nào cũng không được nhận đề tài!> Như thế, ta thấy sự phức tạp đến mức nào của một vi điều khiển!

Tuy nhiên, không phải công nghệ bắt đầu vào chế tạo bán dẫn kích thước micro, nano ngay được, đấy là một quá trình phát triển lâu dài, từ ống chân không, một Transistor, vài Transistor, mạch tích hợp (IC) đến mạch tích hợp rất cao (VLSI) với vài triệu Transistor hiệu ứng trường trên mỗi chip đơn lẻ!

B.II. *Cấu trúc máy tính số sử dụng chương trình lưu trữ:<Stored-program>*

Xem hình sau:



Hình 1: Mô hình máy tính số sử dụng chương trình lưu trữ

Ta chú thích một cách dễ hiểu nguyên lý hoạt động của bộ xử lý dạng này: <dựa trên mô hình có PC>

- Bộ xử lý đưa ra địa chỉ kế tiếp cho thanh ghi (PC) truy cập vào vùng nhớ, nhận chỉ lệnh thực thi, chỉ lệnh thực thi chứa các địa chỉ của dữ liệu chương trình <sẽ phân tích ở phần sau>, thanh ghi dùng các địa chỉ này để truy cập vào vùng nhớ để lấy dữ liệu, dữ liệu đấy được dùng để xử lý trong nhân tính toán <qua bộ ALU chẳng hạn>, kết quả cuối cùng sẽ được ghi ngược lại bộ nhớ bằng địa chỉ định trước hoặc địa chỉ trung gian, hoặc qua thanh ghi tích lũy. <phần tổ chức bộ nhớ trong chương trình được viết kĩ trong giáo trình môn *Vi xử lý* của thầy Hồ Trung Mỹ>
- Bạn tưởng tượng bộ nhớ là một kho hàng, chứa tất cả đơn hàng và hàng hóa đã vào sổ sách. Tất cả hàng hóa được bán ra thị trường lần lượt theo trình tự đơn đặt hàng, và người chủ quản lý chỉ biết số thứ tự đơn hàng. Để bán sản phẩm, anh ta đọc số thứ tự đơn hàng, số thứ tự ấy đại diện cho một khách hàng cụ thể, người ta vào kho, đối chiếu số thứ tự đơn hàng để biết tên khách hàng, yêu cầu của khách hàng. Số hàng hóa, loại hàng hóa cần phải lấy…
  - Ta nói cụ thể hơn bằng các con số để dễ hình dung: Người chủ quản lý đã thực hiện xong đơn hàng thứ 3, anh đọc tiếp:
    - "Đến đơn hàng thứ 4"
  - Người ta vào kho hàng lấy đơn hàng thứ 4. Trong đơn hàng thứ 4 ghi: khách hàng Nguyễn Văn A, ông A cần mua 10 chiếc áo lông thú, 100 quần bò. Người quản lý nhìn vào đơn hàng và đọc tiếp:
    - "10 áo lông thú"
  - Anh phụ trách lô hàng áo lông thú vào kho lấy đủ số lượng yêu cầu. Người quản lý đọc:
    - "100 quần bò"
  - Anh phụ trách lô hàng quần bò vào kho lấy đúng số lượng đấy.
  - Người chủ lô hàng cho qua máy đếm từng loại hàng, đọc tên khách hàng đến nhận hàng và tiếp tục đọc:
    - "Đến đơn hàng thứ 5"…vv…

Trong ví dụ trên, người quản lý làm nhiệm vụ của thanh ghi PC, đơn hàng là chỉ lệnh thực thi, các người quản lý kho hàng là các địa chỉ truy cập vào vùng nhớ dữ liệu, máy đếm hàng làm nhiệm vụ của ALU, để hoàn thành chỉ lệnh, ta phải truy cập bộ nhớ dữ liệu 2 lần <2 lần vào kho lấy hàng>…. Nếu ta làm việc trên thanh ghi, chỉ có các mức 0-1 trên thanh ghi là thay đổi sau mỗi lần truy cập vào vùng nhớ hoặc xử lý qua ALU.

B.III. *Dạng đơn giản của bộ xử lý:*

Có thể gồm những phần cơ bản sau:

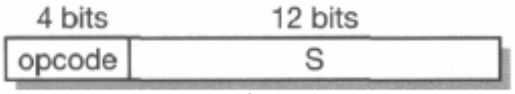
i. *Program Counter* (PC): thanh ghi giữ địa chỉ của chỉ lệnh hiện tại.

ii. *Thanh ghi tích lũy* (ACC): giữ giá trị dữ liệu khi đang làm việc.

iii. *Đơn vị xử lý số học* (ALU): thực thi các lệnh nhị phân như cộng, trừ, gia tăng…

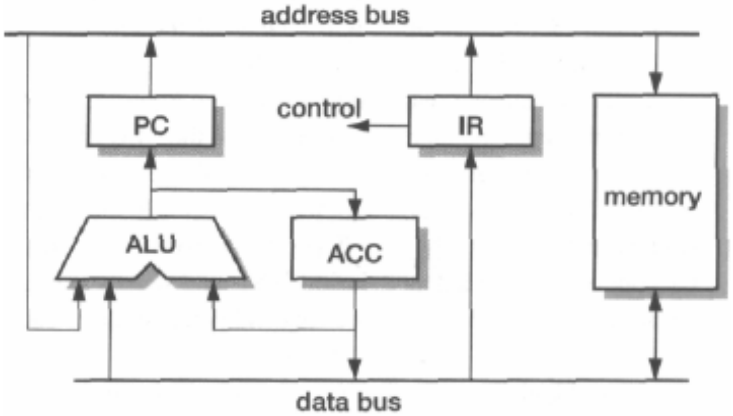
iv. *Thanh ghi chỉ lệnh* (IR): giữ chỉ lệnh hiện tại đang thực thi.

Nhân xử lý đơn giản MU0 được phát triển đầu tiên ở đại học Manchester-Anh Quốc là nhân xử lý có chỉ lệnh dài 16bit, với 12bit địa chỉ <8kBytes> và 4 bit opcode; cấu trúc chuẩn cho chỉ lệnh có dạng:



Hình 2: Cấu trúc chuẩn cho chỉ lệnh của MU0

Ví dụ đơn giản về thiết kế đường truyền dữ liệu<datapath> của nhân xử lý MU0:

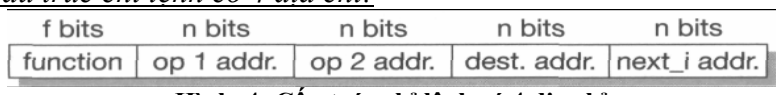


Hình 3: Ví dụ về đường truyền dữ liệu của MU0

Ta đang xét việc thiết kế ở cấp chuyển đổi mức thanh ghi(RTL): PC chỉ đến chỉ lệnh cần thực thi, load vào IR, giá trị chứa trong IR chỉ đến vùng địa chỉ ô nhớ, nhận giá trị, kết hợp với giá trị đang chứa trong ACC qua đơn vị xử lý ALU để tạo giá trị mới, chứa vào ACC. Mỗi chỉ lệnh như vậy, tùy vào số lần truy cập ô nhớ mà tốn số chu kì xung nhịp tương đương. Sau mỗi chỉ lệnh thực thi, PC sẽ được tăng thêm.

B.IV. *Sơ qua về cách thiết kế cấu trúc tập lệnh:*

IV.1. *Cấu trúc chỉ lệnh có 4 địa chỉ:*

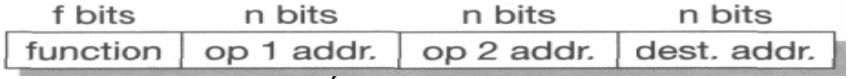


Hình 4: Cấu trúc chỉ lệnh có 4 địa chỉ

Ví dụ:

```
ADD d, s1, s2, next_i          ;d := s1 +s2
```

IV.2. *Cấu trúc chỉ lệnh có 3 địa chỉ:*

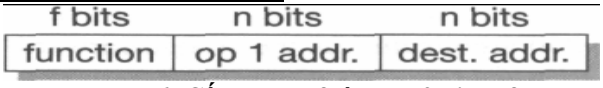


Hình 5: Cấu trúc chỉ lệnh có 3 địa chỉ

Ví dụ:

```
ADD d, s1, s2                  ; d := s1 + s2
```

IV.3. *Cấu trúc chỉ lệnh có 2 địa chỉ:*

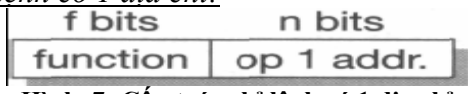


Hình 6: Cấu trúc chỉ lệnh có 2 địa chỉ

Ví dụ:

```
ADD d, s1                      ; d := d + s1
```

IV.4. Cấu trúc chỉ lệnh có 1 địa chỉ:

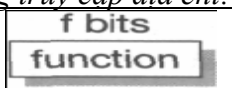


Hình 7: Cấu trúc chỉ lệnh có 1 địa chỉ

Ví dụ:

```
ADD s1                  ; accumulator := accumulator + s1
```

IV.5. Cấu trúc chỉ lệnh không truy cập địa chỉ:



Hình 8: Cấu trúc chỉ lệnh không truy cập địa chỉ

Ví dụ:

```
ADD                     ; top_of_stack := top_of_stack +next_on_stack
```

B.V. Các chế độ định địa chỉ:

Các chế độ định địa chỉ cơ bản, bạn tham khảo ở sách *Vi xử lý* của thầy Hồ Trung Mỹ. Có thể tùy nhà sản xuất đưa ra 8 hoặc 9 chế độ khác nhau. Tuy nhiên, nó cùng một mức nền như nhau.

B.VI. Cấu trúc lệnh CISC và RISC:

Máy tính chỉ hiểu các mức 0/1 trên mỗi transistor cụ thể, người sử dụng muốn thực hiện một chương trình nào đấy, phải nạp các mã lệnh 0-1 vào bộ nhớ cho máy tính. Có 3 cách cơ bản để làm việc ấy:

1. Viết ngay dạng mã máy 0-1 và nạp vào bộ nhớ. Cách này rất khó thực thi.
2. Viết dạng tên gợi nhớ bằng hợp ngữ, sau đó biên dịch ra mã máy, cấp này cũng rất gần với ngôn ngữ máy và cũng khó thực hiện với các chương trình phức tạp. Tuy nhiên, cấu trúc gọn nhẹ.
3. Viết bằng một ngôn ngữ cấp cao, sau đó dùng một trình biên dịch <compiler> để dịch ra mã máy. Cách này tuy dễ với người viết chương trình nhưng cũng sẽ làm chương trình có dung lượng lớn hơn nếu viết bằng ASM. Và thách thức là làm sao các nhà sản xuất phần mềm, phần cứng bắt tay nhau để chương trình biên dịch này thật chuẩn tắc, nhỏ gọn, không tạo nhiều code trung gian.

VI.1. Nêu vấn đề:

Trong suốt thập niên 1980, các nhà thiết kế cố gắng thu hẹp khoảng cách giữa ngôn ngữ cấp cao của con người và ngôn ngữ máy, họ đã đưa ra cấu trúc các chỉ lệnh phức tạp-CISC, có các chế độ định địa chỉ khác nhau, mỗi lệnh thực thi cần nhiều lần định địa chỉ để lấy dữ liệu, và do đó, tốn nhiều chu kì xung nhịp cho mỗi chỉ lệnh.

Nếu việc giảm thiểu ranh giới giữa tập lệnh của vi điều khiển và ngôn ngữ cấp cao không phải là một cách hay để máy tính hoạt động hiệu quả, các nhà thiết kế phải làm sao để tối ưu tốc độ xử lý?

VI.2. Thiết kế tập lệnh dựa trên CISC và RISC:

Nếu muốn biết cách làm để vi xử lý hoạt động nhanh hơn, ta phải biết vi xử lý dùng hầu hết thời gian của chúng vào việc gì? Chúng ta dễ nghĩ rằng: '*Vi xử lý tất nhiên dùng hầu hết thời gian của nó để tính toán*'; nghĩa là thời gian hầu hết ở bộ ALU. Thật ra, theo thống kê thì suy đoán này hoàn toàn sai lầm:

Loại chỉ lệnh	**Sử dụng**
Chuyển dữ liệu	43%
Điều khiển dòng chảy <lệnh>	23%
Tính toán số học	15%
So sánh	13%
Phép toán logic	5%
Khác	1%

Bảng 1: Bảng thống kê các loại chỉ lệnh thường dùng

i. Chu kì lệnh:

Thông thường, để thực thi một chỉ lệnh, quá trình có thể bao gồm các bước sau:

1. Nhận lệnh từ bộ nhớ <fetch>
2. Giải mã lệnh, xác định các tác động cần có và kích thước lệnh <decode>
3. Truy cập các toán hạng có thể được yêu cầu từ thanh ghi <reg>
4. Kết hợp toán hạng đấy với để tạo thành kết quả hay địa chỉ bộ nhớ <ALU>
5. Truy cập vào bộ nhớ cho toán hạng dữ liệu nếu cần thiết <mem>
6. Viết kết quả ngược lại băng thanh ghi <res>

ii. So sánh CISC và RISC:

ii.a. Kiến trúc tập lệnh RISC:

Có các đặc điểm quan trọng sau:

◊ Kích thước các chỉ lệnh là cố định <32 bit> với chỉ một vài định dạng. <CISC có kích thước tập lệnh thay đổi với rất nhiều định dạng khác nhau>

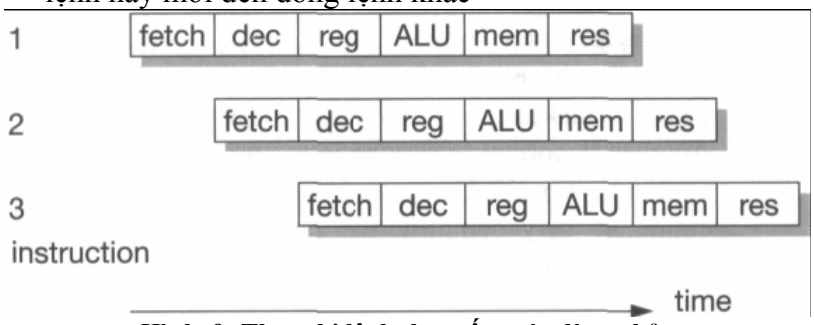
◊ Sử dụng kiến trúc load-store các chỉ lệnh xử lý dữ liệu hoạt động chỉ trong thanh ghi và cách ly với các chỉ lệnh truy cập bộ nhớ <CISC cho phép giá trị trong bộ nhớ được dùng như như toán hạng trong các chỉ lệnh xử lý dữ liệu>

◊ Gồm một số lớn các thanh ghi đa dụng 32 bit, cho phép cấu trúc load-store hoạt động hiệu quả. <CISC có rất nhiều thanh ghi, nhưng hầu hết đề chỉ để sử dụng cho một mục đích riêng biệt nào đấy-ví dụ các thanh ghi dữ liệu và địa chỉ trong Motorola MC68000>

ii.b. Tổ chức tập lệnh RISC:

◊ Giải mã các chỉ lệnh logic bằng kết nối phần cứng <CISC sử dụng rất nhiều code trong ROM giải mã các chỉ lệnh>

◊ Thực thi chỉ lệnh theo cấu trúc dòng chảy <xem hình dưới> <CISC ít khi cho phép các dòng lệnh thực thi kiểu này, chúng phải tuần tự hết dòng lệnh này mới đến dòng lệnh khác>



Hình 9: Thực thi lệnh theo cấu trúc dòng chảy

◊ Một chỉ lệnh thực thi trong 1 chu kì xung nhịp <CISC cần nhiều chu kì xung nhịp để hoàn thành một lệnh>

ii.c. Điểm mạnh của bộ xử lý dùng tập lệnh RISC:

◊ Kích thước miếng bán dẫn nhỏ hơn: bộ xử lý đơn giản đòi hỏi ít transistor hơn, do đó, kích thước cần dùng nhỏ lại, dành vùng diện tích trống để tăng các chức năng như bộ nhớ cache, chức năng quản lý bộ nhớ, ..vv…

◊ Thời gian phát triển một sản phẩm ngắn hơn <do kĩ thuật đơn giản hơn>

◊ Cấu hình mạnh hơn: điều này có vẻ khó tin, tuy nhiên, để ý các lập luận:

- Những vật nhỏ hơn thì có tần số hoạt động tự nhiên lớn hơn <côn trùng đập cánh nhanh hơn bồ câu, bồ câu đập cánh nhanh hơn đại bàng…>
- Khi ta đặt ra các chỉ lệnh phức tạp, tuy nó gần gũi với ngôn ngữ cấp cao, nhưng như thế, vô tình cũng làm các chỉ lệnh khác phức tạp lên, và để thực thi một chỉ lệnh như vậy cần tốn nhiều chu kì xung nhịp. Trong khi đó, nếu dùng RISC chỉ mất một chu kì xung nhịp cho mỗi lệnh, khi ta phân nhỏ vấn đề phức tạp thành các vấn đề đơn giản thì cách giải quyết sẽ tốt hơn.

ii.d. *Tần số hoạt động tối đa của RISC và CISC:*

◊ Như ví von ở trên, RISC nhỉnh hơn, tuy nhiên, cũng có giai đoạn CISC có tần số xung nhịp lớn hơn RISC, mặc dù vậy, thời gian để hoàn tất một lệnh của CISC cũng chậm hơn RISC do 1 chỉ lệnh của CISC cần nhiều chu kì xung nhịp để hoàn tất.

ii.e. *Những điểm bất tiện của RISC:*

◊ Không phải RISC chỉ có điều thuận lợi, nó cũng có một vài bất cập, mà cụ thể là:

- Mã lệnh của RISC không phong phú bằng CISC.
- Không thể thực thi các mã lệnh của x86.

◊ Điểm bất tiện thứ 2 được kể trên khó để sửa đổi, phải dùng các phần mềm hỗ trợ nền cơ sở cho RISC, tuy nhiên, với máy tính của IBM, có thể bị từ chối.

◊ Điểm bất tiện thứ nhất phát sinh vì cấu trúc tập lệnh của RISC là cố định, nó sẽ trở nên nghiêm trọng<làm chậm hệ thống> nếu phải giải quyết các công việc phức tạp. Nếu không có bộ nhớ phụ cache sẽ dẫn tới việc cần nhiều băng thông của bộ nhớ chính, điều đó làm tiêu tốn nhiều năng lượng.

C. Kiến trúc tổ chức của ARM:

C.I. Sơ lược về tên gọi:

ARM lúc đầu được đặt tên theo công ty Acorn. *ARM=Acorn RISC Machine* (dịch mô mo na là chiếc máy sử dụng tập lệnh đơn giản của công ty Acorn). Sau này, do có thêm nhiều công ty cùng phát triển và một số lý do khác, người ta thống nhất gọi *ARM=Advance RISC Machine.*

C.II. Sự kế thừa cấu trúc:

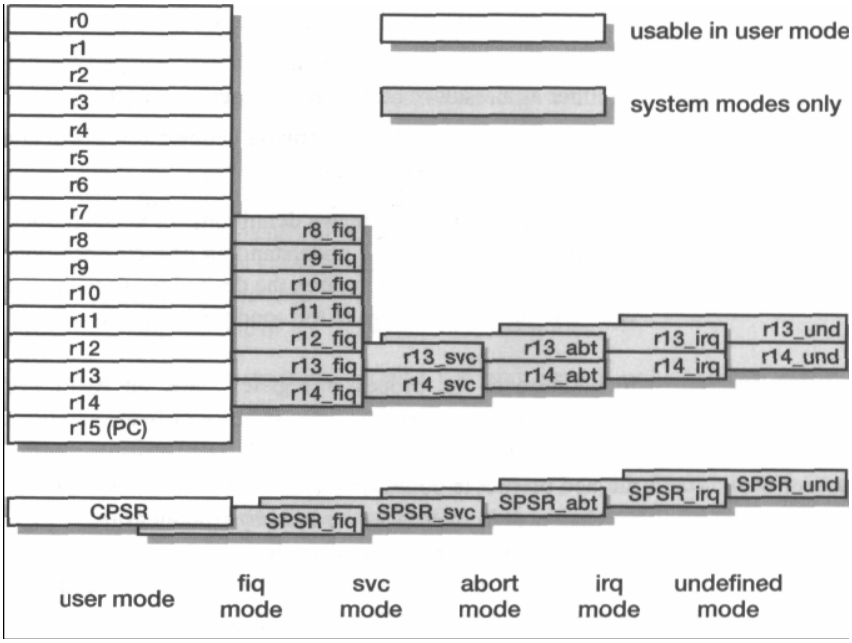
Với các mô hình RISC của Berkeley RISC I và II, Stanford MIPS, ARM kế thừa:

II.1. Cấu trúc cơ bản:

- Cấu trúc load-store
- Chỉ lệnh có chiều dài cố định <32bit>
- Cấu trúc chỉ lệnh có 3 địa chỉ.
- Thay vì chỉ dùng 1 chu kì xung nhịp cho tất cả các chỉ lệnh, ARM thiết kế để sao cho tối giản số chu kì xung nhịp cho một chỉ lệnh, do đó tăng được sự phức tạp cho các chỉ lệnh đơn lẻ.

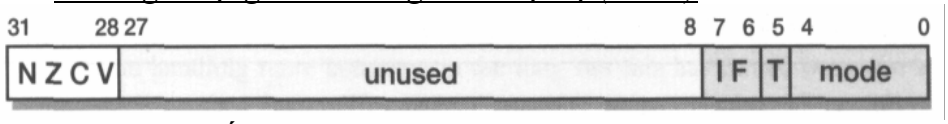
II.2. Mô hình thiết kế ARM:

Để phục vụ mục đích của người dùng: r0-r14: 15 thanh ghi đa dụng, r15 là thanh ghi PC, thanh ghi trạng thái chương trình hiện tại (CPSR). Các thanh ghi khác được giữ lại cho hệ thống <thanh ghi chứa các ngắt chẳng hạn>



Hình 10: Các thanh ghi của ARM

i. Thanh ghi trạng thái chương trình hiện tại(CPSR)



Hình 11: Cấu trúc của thanh ghi trạng thái chương trình hiện tại

Thanh ghi CPSR được người dùng sử dụng chứa các bit điều kiện.

- N: Negative- cờ này được bật khi bit cao nhất của kết quả xử lý ALU bằng 1.
- Z: Zero- cờ này được bật khi kết quả cuối cùng trong ALU bằng 0.
- C: Carry- cờ này được bật khi kết quả cuối cùng trong ALU lớn hơn giá trị 32bit và tràn.
- V: Overflow-cờ báo tràn sang bit dấu.

II.3. Cấu trúc load-store:

Cũng như hầu hết các bộ xử lý dùng tập lênh RISC khác, ARM cũng sử dụng cấu trúc load-store. Điều đó có nghĩa là: tất cả các chỉ lệnh <cộng, trừ…> đều được thực hiện trên thanh ghi. Chỉ có lệnh copy giá trị từ bộ nhớ vào thanh ghi<load> hoặc chép lại giá trị từ thanh ghi vào bộ nhớ<store> mới có ảnh hưởng tới bộ nhớ.

Các bộ xử lý CISC cho phép giá trị trên thanh ghi có thể cộng với giá trị trong bộ nhớ, đôi khi còn cho phép giá trị trên bộ nhớ có thể cộng với giá trị trên thanh ghi. ARM không hỗ trợ cấu trúc lệnh dạng 'từ bộ nhớ đến bộ nhớ'. Vì thế, tất cả các lệnh của ARM có thể thuộc 1 trong 3 loại sau:

1. Chỉ lệnh xử lý dữ liệu: chỉ thay đổi giá trị trên thanh ghi.
2. Chỉ lệnh truyền dữ liệu: copy giá trị từ thanh ghi vào bộ nhớ và chép giá trị từ bộ nhớ vào thanh ghi.<load-store>
3. Chỉ lệnh điều khiển dòng lệnh: Bình thường, ta thực thi các chỉ lệnh chứa trong một vùng nhớ liên tiếp, chỉ lệnh điều khiển dòng lệnh cho phép chuyển sang các địa chỉ khác nhau khi thực thi lệnh, tới những nhánh cố định, <lệnh rẽ nhánh> hoặc là lưu và trở lại địa chỉ để phục hồi chuỗi lệnh ban đầu <chỉ lệnh rẽ nhánh và kết nối> hay là đè lên vùng code của hệ thống <gọi giám sát-ngắt phần mềm>.

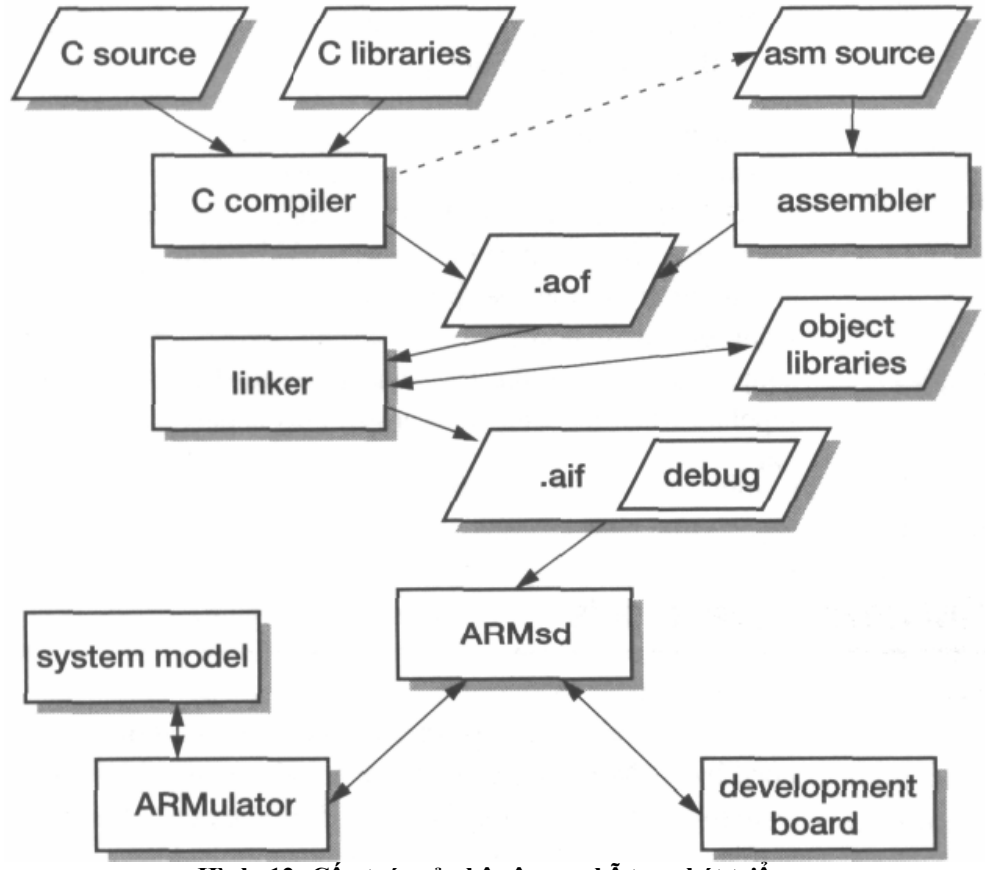
II.4. Tập lệnh của ARM:

Tất cả lệnh của ARM đều là 32bit:

- Có cấu trúc dạng load-store.

- Cấu trúc lệnh định dạng 3 địa chỉ (nghĩa là địa chỉ của 2 toán hạng nguồn và toán hạng đích đều là các địa chỉ riêng biệt)
- Mỗi chỉ lệnh thực thi một điều kiện.
- Có cả chỉ lệnh load-store nhiều thanh ghi đồng thời.
- Có khả năng dịch bit kết hợp với thực thi lệnh ALU trong chỉ 1 chu kì máy.
- Thumb code.

II.5. ARM C-Compiler:



Hình 12: Cấu trúc của bộ công cụ hỗ trợ phát triển.

D. Lập trình hợp ngữ cho ARM:

D.I. Lệnh xử lý dữ liệu:

Lệnh xử lý dữ liệu của ARM cho phép thực thi các lệnh số học, logic trên các thanh ghi. Những phép toán dạng này có 2 tóan tử tham gia và sinh ra 1 kết quả duy nhất. Trong 2 tóan hạng nguồn, toán hạng thứ 2 có thể là thanh ghi, giá trị tức thời, tóan hạng này có thể được dịch bit trước khi tham gia vào phép tính số học mà vẫn tính là trong 1 chu kì máy.

+Tất cả các toán hạng đều có chiều dài 32bit.

+Nếu là 1 kết quả thì nó cũng có chiều dài là 32bit <trừ trường hợp nhân sinh ra kết quả dài 64bit>

+ARM sử dụng cấu trúc chỉ lệnh có 3 địa chỉ.

- Các lệnh tóan học.

```
ADD     r0, r1, r2          ; r0 := r1 + r2
ADC     r0, r1, r2          ; r0 := r1 + r2 + C
SUB     r0, r1, r2          ; r0 := r1 - r2
SBC     r0, r1, r2          ; r0 := r1 - r2 + C - 1
RSB     r0, r1, r2          ; r0 := r2 - r1
RSC     r0, r1, r2          ; r0 := r2 - r1 + C - 1
```

Hình 13: Các lệnh toán học

- Các lệnh logic.

```
AND     r0, r1, r2          ; r0 := r1 and r2
ORR     r0, r1, r2          ; r0 := r1 or r2
EOR     r0, r1, r2          ; r0 := r1 xor r2
BIC     r0, r1, r2          ; r0 := r1 and not r2
```

Hình 14: Các lệnh logic

- Tác vụ chuyển giá trị các thanh ghi.

```
MOV     r0, r2              ; r0 := r2
MVN     r0, r2              ; r0 := not r2
```

Hình 15: Tác vụ chuyển các giá trị của thanh ghi

- Chức năng so sánh:

```
CMP CMN  r1, r2             ; set cc on r1 - r2
TST TEQ  r1, r2             ; set cc on r1 + r2
         r1, r2             ; set cc on r1 and r2
         r1, r2             ; set cc on r1 xor r2
```

Hình 16: Chức năng so sánh

- ….

D.II. Chỉ lệnh chuyển dữ liệu:

Chỉ lệnh chuyển dữ liệu cũng tương tự như lệnh số học, có các dạng như chuyển dữ liệu giữa 2 thanh ghi, giữa 1 thanh ghi và 1 địa chỉ trực tiếp.

- Chỉ lệnh load và store 1 thanh ghi.
- Chỉ lệnh load và store nhiều thanh ghi.
- Chỉ lệnh trao đổi giá trị các thanh ghi

D.III. Định địa chỉ gián tiếp qua thanh ghi:

- Ví dụ:

```
LDR r0, [r1]        ;r0 := mem32[r1]
STR r0, [r1]        ;mem32[r1] := r0
```

D.IV. Khởi tạo địa chỉ pointer: <r15=PC>

D.V. Định địa chỉ stack.

D.VI. Các chỉ lệnh điều khiển dòng lệnh:

- Chỉ lệnh rẽ nhánh.
- Nhánh điều kiện.

..vv…

Do các phần này tương đối đơn giản, vả lại tôi cũng không muốn chú trọng vào nên ta lược bỏ qua. Bạn xem thêm ở sách tiếng Anh.

D.VII. Viết chương trình đơn giản:

Xét chương trình sau:

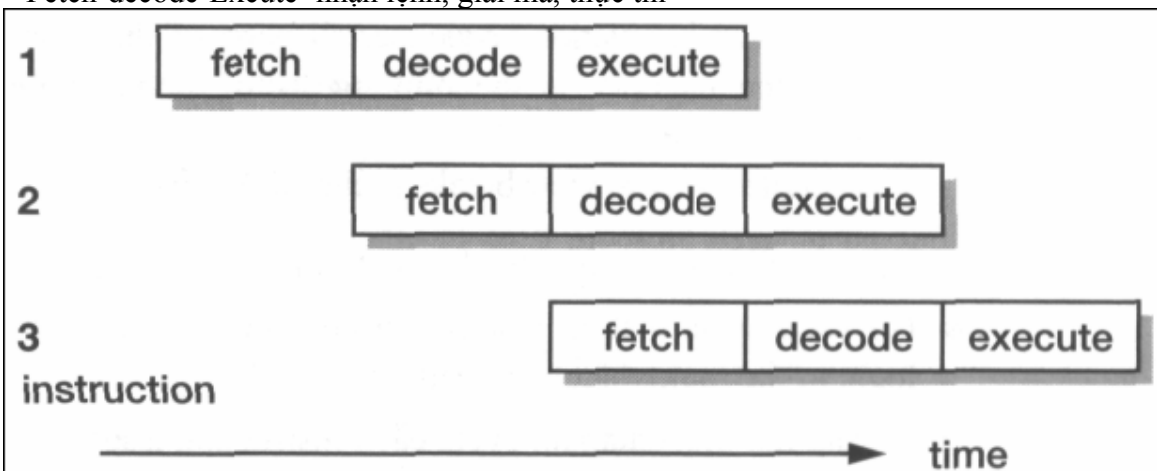
```
        AREA HelloW,CODE,READONLY ;Khai bao vung code
SWI_SwiteC  EQU  &0               ;Ki tu xuat o R0
SWI_Exit    EQU  &11              ;Ket thuc chuong trinh
        ENTRY                     ;Diem bao hieu vao chuong trinh
START       ADR  R1,TEXT          ;R1 chi den vung dia chi cua TEXT
LOOP LDRB R0,[R1],#1              ;R0:=[R1];R1:=R1+1
        CMP  R0,#0                ;R0 chi toi gia tri cuoi hay chua
        SWINE       SWI_WriteC    ;Neu chua ket thuc in
        BNE  LOOP                 ;thi quay nguoc lai vong lap
        SWI  SWI_Exit             ;Quay nguoc ve lai chuong trinh quan ly
TEXT  ="Hello World",&0a,&0d,0    ;Khai bao doan Text
        END                       ;Chuong trinh ket thuc
;Do không chú trọng tới ASM nên lược qua phần này.
```

E. Cách tổ chức và thực thi tập lệnh của ARM:

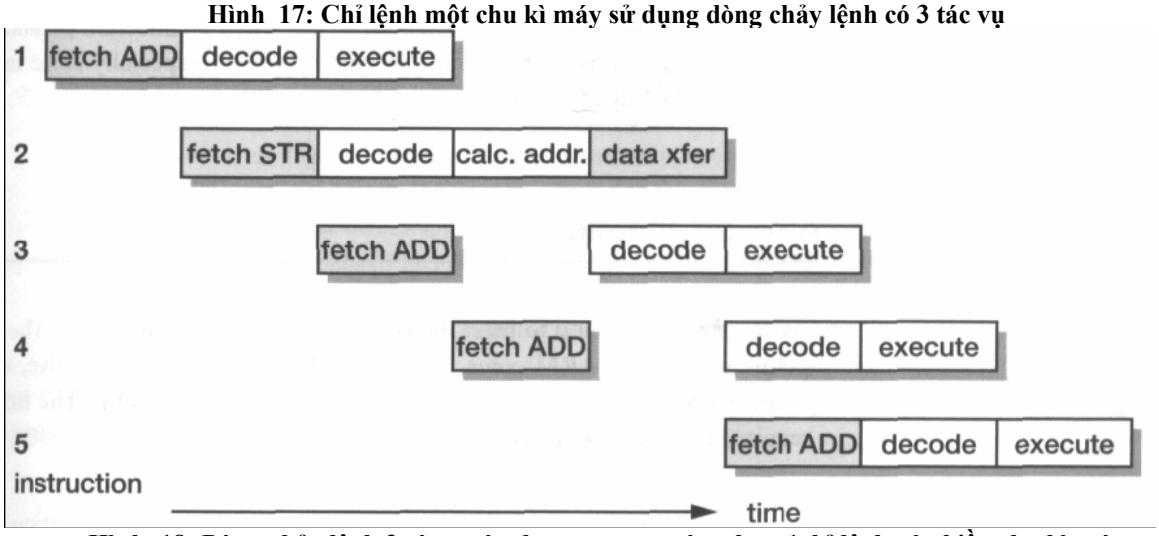
Cách tổ chức của nhân ARM không thay đổi nhiều trong khoảng 1983-1995:đến ARM7-sử dụng dòng chảy lệnh sử dụng 3 tác vụ. Từ 1995 trở về sau, đã xuất hiện một vài nhân ARM mới được giới thiệu có dòng chảy lệnh sử dụng 5 tác vụ.

E.I. Dòng chảy lệnh có 3 tác vụ:

Fetch-decode-Excute<nhận lệnh, giải mã, thực thi>



Hình 17: Chỉ lệnh một chu kì máy sử dụng dòng chảy lệnh có 3 tác vụ



Hình 18: Dòng chảy lệnh 3 tác vụ áp dụng trong trường hợp 1chỉ lệnh có nhiều chu kì máy

E.II. Dòng chảy lệnh có 5 tác vụ:

<Fetch-decode-excute-buffer/data-write back>

Thời gian để bộ xử lý thực thi một chương trình: $T_{prog} = \frac{N_{inst} \times CPI}{f_{clk}}$ <Công thức 1>.

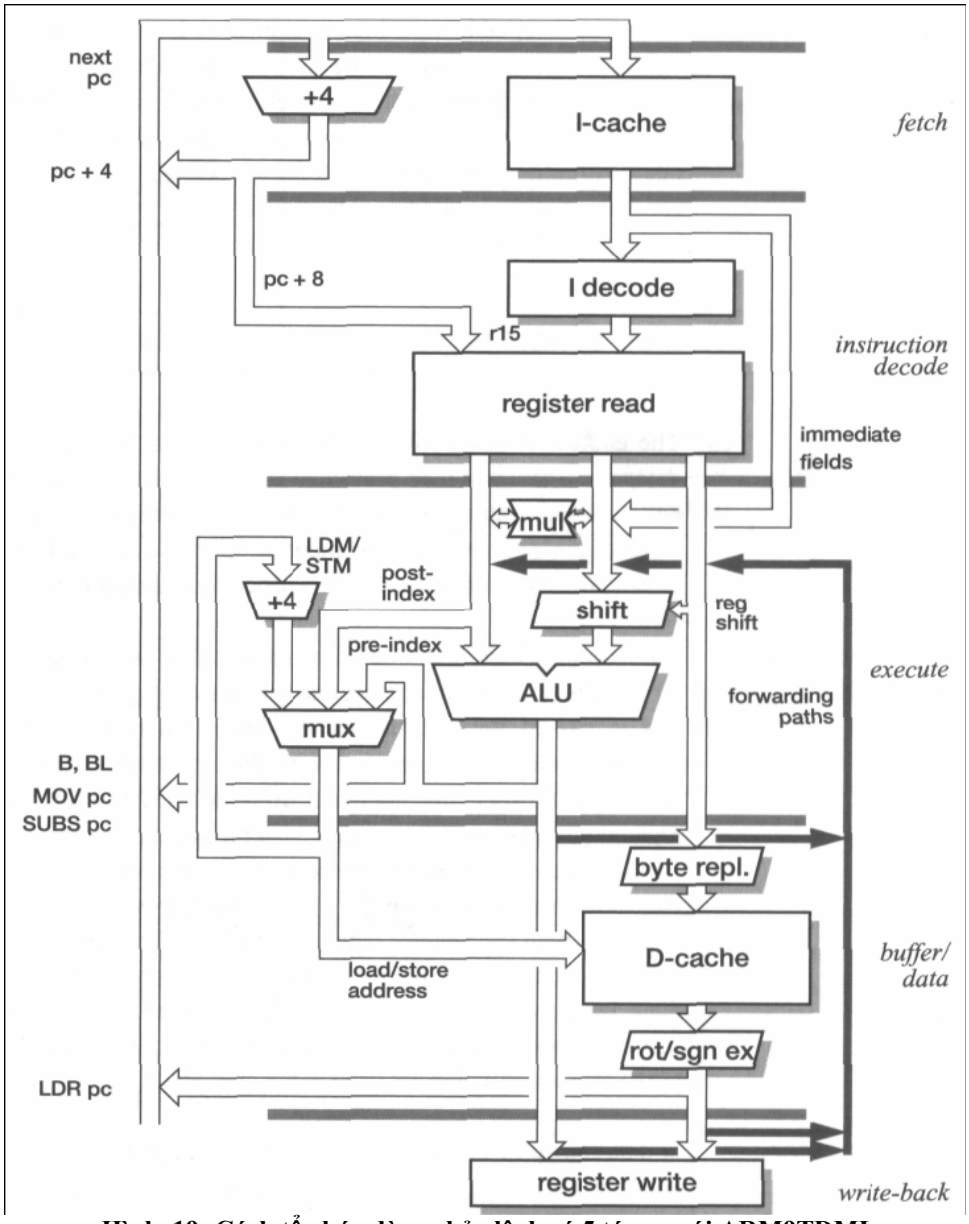
Trong đó CPI là số xung nhịp trung bình cần cho mỗi chỉ lệnh, N_{inst} là số chỉ lệnh thực thi một chương trình<là cố định>, f_{clk} là tần số xung nhịp. Với công thức trên, ta có 2 cách để giảm thời gian thực thi một chương trình:

+Tăng tần số xung nhịp: điều này đòi hỏi trạng thái của mỗi tác vụ trong dòng chảy lệnh là đơn giản, và, do đó, số tác vụ sẽ tăng thêm.

+Giảm CPI: điều này đòi hỏi mỗi chỉ lệnh cần nhiều dòng chảy lệnh hơn với tác vụ không đổi, hoặc các tác vụ cần đơn giản hơn, hoặc kết hợp cả 2 lại với nhau.

→ARM đưa ra cấu trúc mỗi dòng chảy lệnh có 5 tác vụ, với cách mô phỏng tựa như cấu trúc von Neumann, với vùng nhớ dữ liệu và chương trình riêng biệt. Từ cấu trúc lệnh có 3 tác vụ được chia nhỏ lại thành 5 tác vụ cũng làm cho mỗi chu kì xung nhịp sẽ thực hiện một công việc đơn giản hơn ở mỗi trạm, cho phép có thể tăng chu kì xung nhịp của hệ thống. Sự tách rời bộ nhớ chương trình và bộ nhớ dữ liệu <cache chứa các chỉ lệnh I-cache và cache chứa dữ

liệu D-cache là tách rời nhau> cũng cho phép giảm đáng kể tài nguyên chiếm của mỗi chỉ lệnh trong một chu kì máy.
<ta có thể quay lại ví dụ ví von ban đầu với đơn hàng đặt ngay phòng làm việc của người quản lý>



Hình 19: Cách tổ chức dòng chảy lệnh có 5 tác vụ với ARM9TDMI

F. Tập lệnh của ARM:

F.I. Kiểu dữ liệu:

ARM hỗ trợ 6 loại dữ liệu:
+8 bit có dấu và không dấu.
+16 bit có dấu và không dấu.
+32 bit có dấu và không dấu.

Như phần trên đề cập, các tóan tử của ARM có 32 bit, bởi vậy, khi làm việc với các dữ liệu ngắn hơn,các bit cao của toán tử sẽ được thay thế bằng bit '0'.

F.II. Chế độ hoạt động:

CPSR[4:0]	Mode	Use	Registers
10000	User	Normal user code	user
10001	FIQ	Processing fast interrupts	_fiq
10010	IRQ	Processing standard interrupts	_irq
10011	SVC	Processing software interrupts (SWIs)	_svc
10111	Abort	Processing memory faults	_abt
11011	Undef	Handling undefined instruction traps	_und
11111	System	Running privileged operating system tasks	user

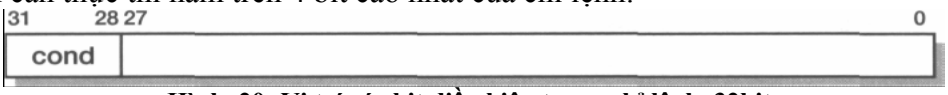
Bảng 2: Các chế độ hoạt động của ARM và sử dụng thanh ghi

Exception	Mode	Vector address
Reset	SVC	0x00000000
Undefined instruction	UND	0x00000004
Software interrupt (SWI)	SVC	0x00000008
Prefetch abort (instruction fetch memory fault)	Abort	0x0000000C
Data abort (data access memory fault)	Abort	0x00000010
IRQ (normal interrupt)	IRQ	0x00000018
FIQ (fast interrupt)	FIQ	0x0000001C

Bảng 3: Các địa chỉ dùng cho hệ thống

F.III. Thực thi các điều kiện:

Điều kiện cần thực thi nằm trên 4 bit cao nhất của chỉ lệnh.



Hình 20: Vị trí các bit điều kiện trong chỉ lệnh 32bit

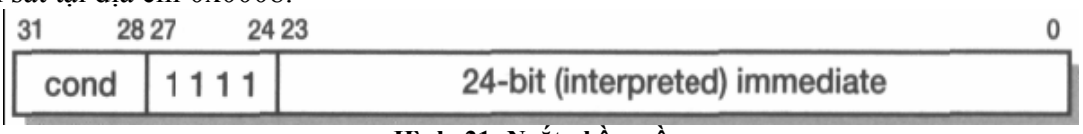
Tổ hợp các điều kiện này được quyết định bởi các bit <N,Z,C,V> nằm trong thanh ghi trạng thái chương trình hiện tại <CPSR>

Opcode [31:28]	Mnemonic extension	Giải thích	Trạng thái cờ để thực thi
0000	EQ	Bằng 0 hoặc bằng nhau	Z=1
0001	NE	Không bằng	Z=0
0010	CS/HS	Có nhớ, cao hơn số không có dấu	C=1
0011	CC/LO	Xóa cờ nhớ, thấp hơn số có dấu	C=0
0100	Ml	Trừ/âm	N=1
0101	PL	Cộng/dương hay zero	N=0
0110	VS	Cờ tràn	V=1
0111	VC	Không tràn	V=0
1000	HI	Lớn hơn số không dấu	C=1 và Z=0
1001	LS	Bé hơn hoặc bằng số không dấu	C=0 hoặc Z =1
1010	GE	Lớn hơn hoạc bàng số có dấu	N=V
1011	LT	Nhỏ hơn số có dấu	N≠V
1100	GT	Lớn hơn số có dấu	Z=0 và N=V
1101	LE-	Nhỏ hơn hoặc bằng số có dấu	Z=1 hoặc N≠V
1110	AL	Luôn luôn	Tùy định
1111	NV	Không được sử dụng!	Không tồn tại

Bảng 4: Các điều kiện

F.IV. Ngắt phần mềm<SWI>:

Các chỉ lệnh ngắt phần mềm gọi chương trình giám sát. Nó đưa vi xử lý vào chế độ giám sát tại địa chỉ 0x0008.



Hình 21: Ngắt phần mềm

Trường 24bit của vector này không ảnh hưởng tới hoạt động các chỉ lệnh nhưng có thể tác động tới code hệ thống. Nếu vào được chế độ giám sát, vi xử lý sẽ:

+Lưu địa chỉ PC vào thanh ghi r14.

+Lưu giá trị thanh ghi trạng thái chương trình chương trình hiện tại vào thanh ghi SPSR

+Vào chế độ giám sát và không cho phép các ngắt bình thường xảy ra <nhưng các ngắt nhanh vẫn còn tác động> bằng cách gán CPSR[4:0]=10011_2 và CPSR[7]=1.

+gán PC=0x08 và thực thi các chỉ lệnh tại đấy.

Để trở về lại chương trình bình thường sau SWI:

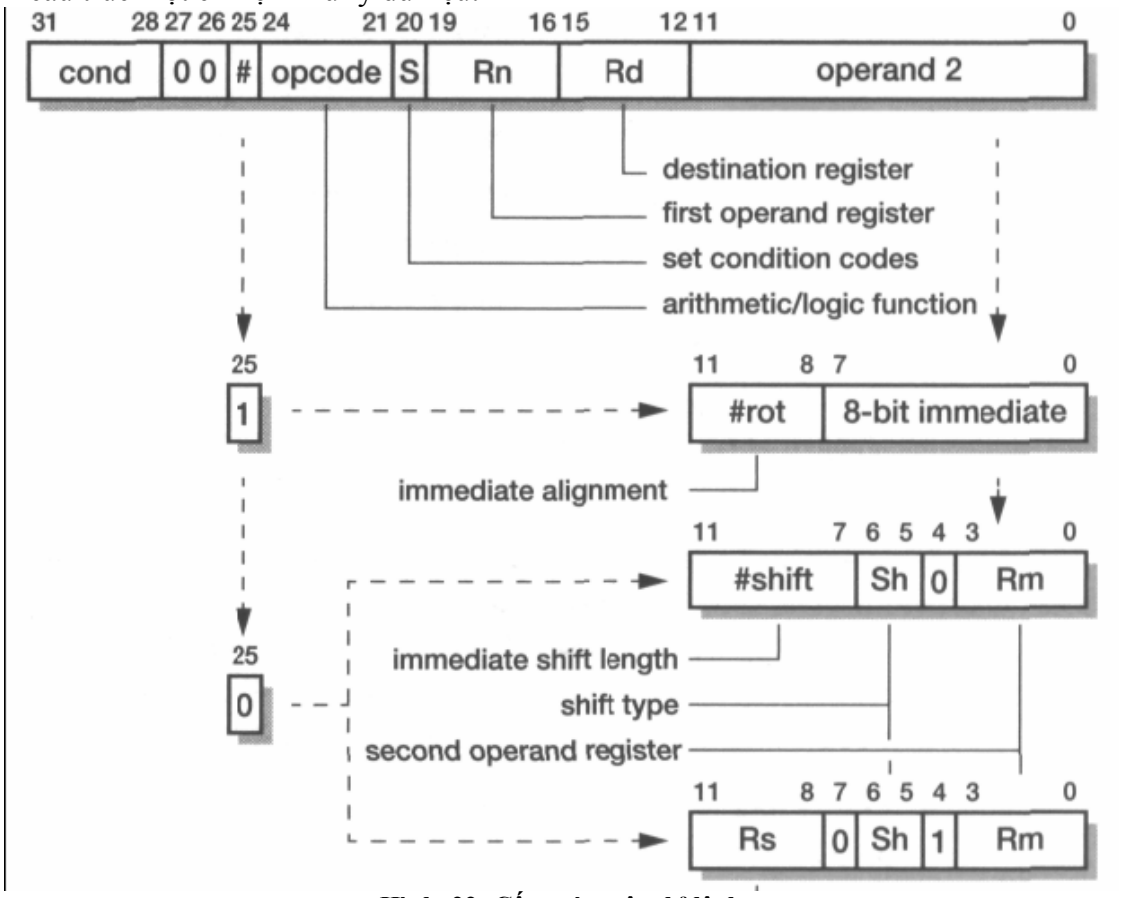
+Copy giá trị thanh ghi r14 vào PC.

+Lấy lại giá trị CPSR từ SPSR

F.V. Lệnh xử lý dữ liệu:

V.1. Mã hóa nhị phân:

Xem cấu trúc một chỉ lệnh xử lý dữ liệu:



Hình 22: Cấu trúc một chỉ lệnh

V.2. Phân tích:

Như đã nói ở các phần trước, mỗi chỉ lệnh của ARM có 32bit, 2 toán tử nguồn và 1 tóan tử đích. Toán tử nguồn thứ nhất là 1 thanh ghi, toán tử nguồn thứ 2 có thể là 1 thanh ghi, một thanh ghi được dịch(hoặc xoay) bit, hoặc là một giá trị cụ thể

i. *Opcode:*

Có tất cả 16 lệnh opcode=[0000_2-1111_2]; tham khảo cụ thể bàng bảng sau:

Opcode	Giả lệnh	Ý nghĩa	Tác động
0000	AND	Logical bit-wise AND	Rd:=Rn AND Op2
0001	EOR	Logical bit-wise exclusive OR	Rd := Rn EOR Op2
0010	SUB	Subtract	Rd := Rn – Op2
0011	RSB	Reverse subtract	Rd := Op2 - Rn
0100	ADD	Add	Rd := Rn + Op2
0101	ADC	Add with carry	Rd := Rn + Op2 + C
0110	SBC	Subtract with carry	Rd := Rn – Op2 + C - 1
0111	RSC	Reverse subtract with carry	Rd := Op2 - Rn + C - 1
1000	TST	Test	Sec onRn AND Op2
1001	TEQ	Test equivalence	Sec on Rn EOR Op2
1010	CMP	Compare	Sec on Rn - Op2
1011	CMN	Compare negated	Sec on Rn + Op2
1100	ORR	Logical bit-wise OR	Rd := Rn OR Op2
1101	MOV	Move	Rd := Op2
1110	BIC	Bit clear	Rd:=Rn ANDNOT Op2
1111	MVN	Move negated	Rd:=NOT Op2

Bảng 5: Bảng Opcode

ii. *Điều kiện:*

Bị ảnh hưởng bởi các bit cờ, trạng thái các cờ được quy định:

+Cờ N=1 nếu kết quả là âm <N=bit cao nhất của kết quả>

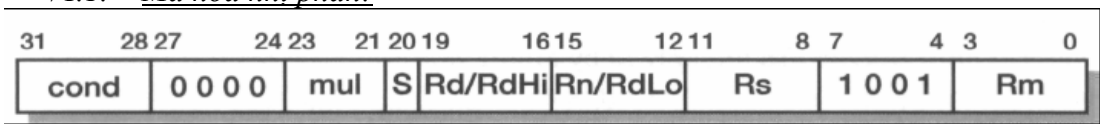
+cờ Z=1 nếu kết quả bằng 0.

+Cờ C được bật nếu kết quả có nhớ từ ALU(ADD, ADC, SUB, SBC, RSB, RSC, CMP, CMN) hay từ kết quả của phép dịch bit. Nếu không có phép dịch bit, cờ C được giữ giá trị trước đó.

+Cờ V chỉ bị ảnh hưởng trong các phép toán số học. V=1 khi có tràn từ bit 30 sang 31. Nó quan trọng chỉ trong phép toán học bù 2 có dấu.

F.VI. Lệnh nhân:

VI.1. Mã hóa nhị phân:



Hình 23: Mã hóa nhị phân cho chỉ lệnh nhân

VI.2. Phân tích:

i. *Opcode:*

Giả lệnh hợp ngữ trong thanh ghi từ 21-23 được cho bởi bảng sau:

Opcode [23:21]	Mnemonic	Ý nghĩa	Tác động
000	MUL	Nhân kết quả 32-bit.	Rd:=(Rm*Rs)[31:0]
001	MLA	Nhân -tích lũy cho giá trị kết quả 32 bit.	Rd:=(Rm*Rs + Rn)[31:0]
100	UMULL	Nhân không dấu 64bit	RdHi: RdLo := Rm * Rs
101	UMLAL	Nhân và tích lũy giá trị không dấu 64bit	RdHi: RdLo += Rm * Rs
110	SMULL	Nhân có dấu 64 bit	RdHi: RdLo := Rm * Rs
111	SMLAL	Nhân và tích lũy giá trị 64bit	RdHi: RdLo+=Rm*Rs

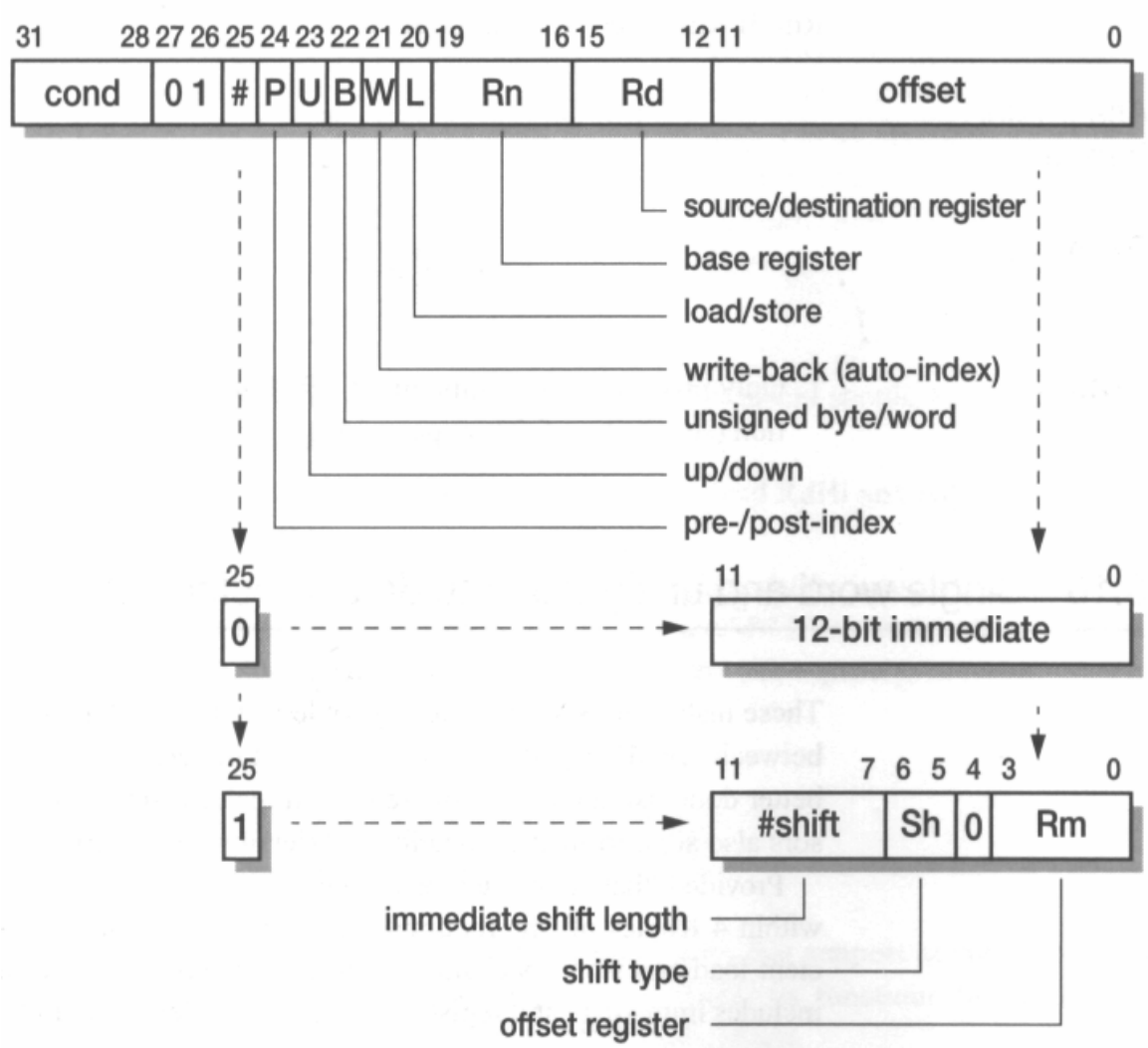
Bảng 6: Giả lệnh hợp ngữ cho phép nhân

ii. *Lệnh hợp ngữ:*

MUL{<cond>}{S} Rd, Rm, Rs
MLA{<cond>}{S} Rd, Rm, Rs, Rn
<mul>{<cond>}{S} RdHi, RdLo, Rm, Rs với <mul> là một trong các lệnh(UMULL, UMLAL, SMULL, SMLAL).
<Các phiên bản ARM có kí hiệu 'M' trong tên là các phiên bản có hỗ trợ nhân 64bit-Ví dụ: ARM7TDMI>

F.VII. Lệnh chuyển dữ liệu: byte không dấu và 1 word:

VII.1. Mã hóa nhị phân:



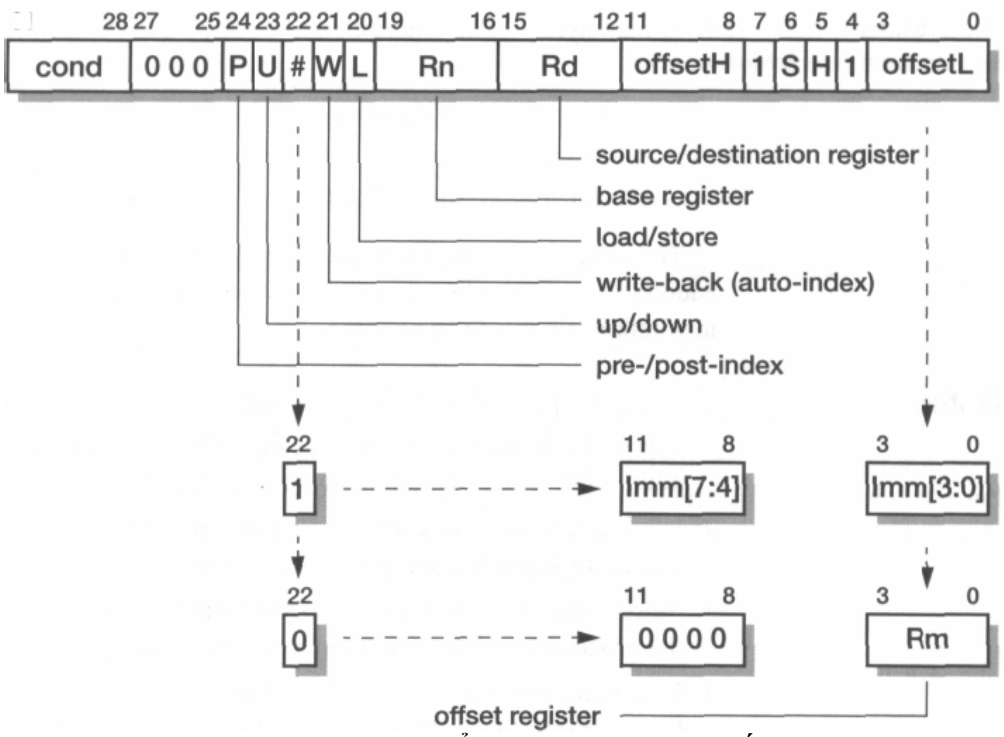
Hình 24: Mã hóa nhị phân cho cấu trúc truyền dữ liệu dạng byte không dấu hoặc word

VII.2. Lệnh hợp ngữ:<p135-136>

Dạng định chỉ số trước: **LDRISTR{<cond>}{B} Rd, [Rn, <offset>]{!}**
Dạng định chỉ số sau: **LDRISTR{<cond>}{B}{T} Rd, [Rn], <offset>**
Dạng tương đối nhờ thanh ghi PC: **LDRISTR{<cond>}{B} Rd, LABEL**

F.VIII. Lệnh chuyển dữ liệu: byte có dấu và nửa word:

VIII.1. Mã hóa nhị phân:



Hình 25: Mã hóa nhị phân chuyển dữ liệu dạng byte có dấu và nửa word

VIII.2. Chú thích:

S	H	Data type
1	0	Signed byte
0	1	Unsigned half-word
1	1	Signed half-word

Bảng 7: Mã hóa loại dữ liệu

Bit S và H cho biết loại dữ liệu truyền được quy ước như bảng trên.

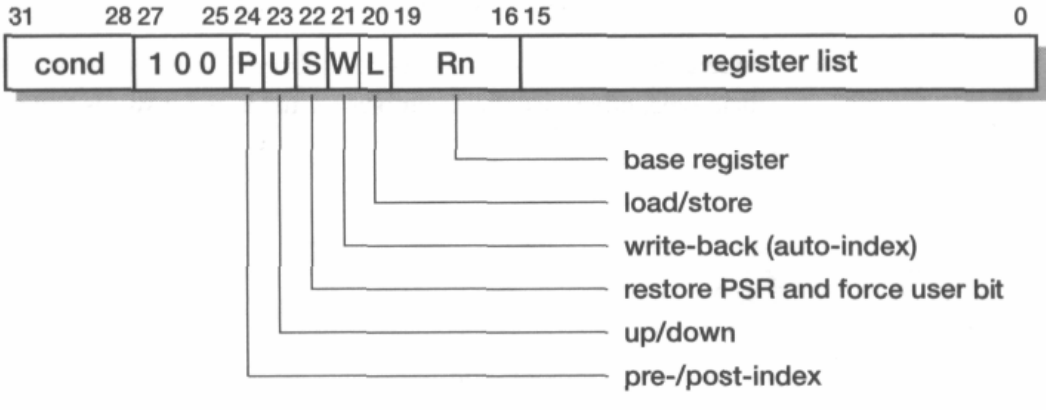
VIII.3. Lệnh hợp ngữ:

Định dạng chỉ số trước: **LDR|STR{<cond>}H|SHI SB Rd, [Rn, <offset>] { ! }**

Định dạng chỉ số sau: **LDRlSTR{<cond>}H|SHlSB Rd, [Rn], <offset>**

F.IX. Lệnh chuyển dữ liệu nhiều thanh ghi:

IX.1. Mã hóa nhị phân:



Hình 26: Mã hóa nhị phân lệnh chuyển dữ liệu nhiều thanh ghi

IX.2. Chú thích:

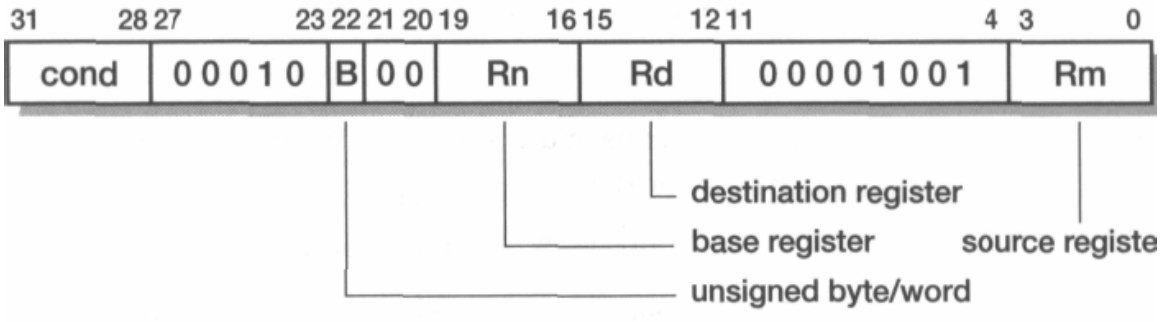
Danh sách các thanh ghi bao gồm một mảng 16 bit thay thế cho 16 thanh ghi từ R0 đến R15 <vị trí bit tương ứng với số thứ tự thanh ghi>. U=1 địa chỉ nền tăng và ngược lại, P=1, địa chỉ nền tăng trước và ngược lại.

IX.3. Lệnh hợp ngữ:

LDMISTM{<cond>}<add mode> Rn{!}, <registers>

F.X. Lệnh hoán đổi giá trị của bộ nhớ và thanh ghi:

X.1. Mã hóa nhị phân:



Hình 27: Mã hóa nhị phân chỉ lệnh đổi giá trị của bộ nhớ và thanh ghi

X.2. Chú thích:

B=1=>load byte không dấu, B=0=>load word ở ô nhớ được định địa chỉ bởi Rn tới Rd, chứa giá trị từ Rm vào ô nhớ tương ứng. Rd và Rm có thể là 1 thanh ghi.

X.3. Lệnh hợp ngữ:

SWP{<cond>}{B} Rd, Rm,Rn

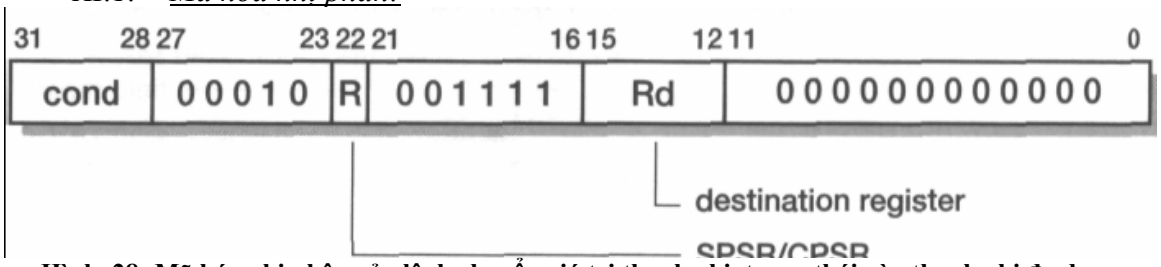
X.4. Chú ý:

+Thanhghi PC không được dùng trong chỉ lệnh này.

+Thanh ghi Rn phải khác với thanh ghi Rm và thanh ghi Rd

F.XI. Lệnh chuyển giá trị từ thanh ghi trạng thái vào thanh ghi đa dụng:

XI.1. Mã hóa nhị phân:



Hình 28: Mã hóa nhị phân của lệnh chuyển giá trị thanh ghi trạng thái vào thanh ghi đa dụng

XI.2. Chú thích:

R=1=>Rd=SPSR

R=0=>Rd=CPSR

XI.3. Lệnh hợp ngữ:

MRS{<cond>} Rd, CPSR

MRS{<cond>} Rd, SPSR

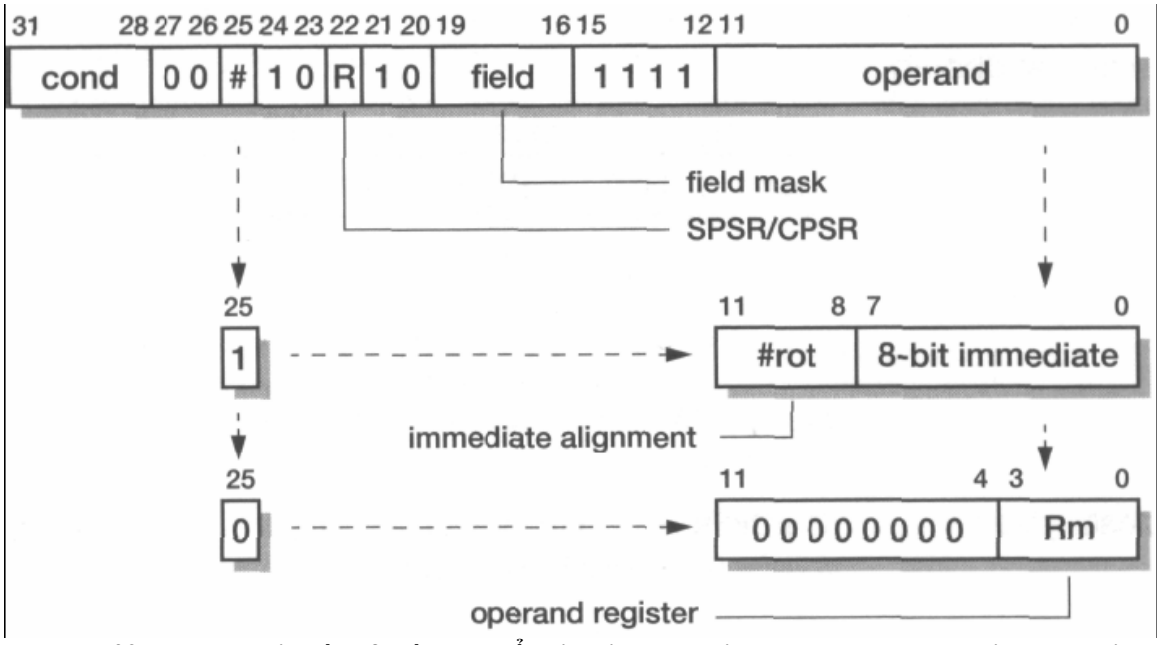
XI.4. Chú ý:

+Chỉ có thể truy cập giá trị SPSR nhờ lệnh này.

+Khi chỉnh sửa các giá trị CPSR hoặc SPSR phải chú ý các bit không được sử dụng.

F.XII. Lệnh chuyển giá trị từ thanh ghi đa dụng vào thanh ghi trạng thái:

XII.1. Mã hóa nhị phân:



Hình 29: Mã hóa nhị phân của lệnh chuyển giá trị thanh ghi đa dụng vào thanh ghi trạng thái

XII.2. *Lệnh hợp ngữ:*

MSR{<cond>} CPSR_f, #<32-bit immediate>
MSR{<cond>}, SPSR_f, #<32-bit immediate>
MSR{<cond>} CPSR_<field>, Rm
MSR{<cond>} SPSR_<field>, Rm

Với <f-field>:

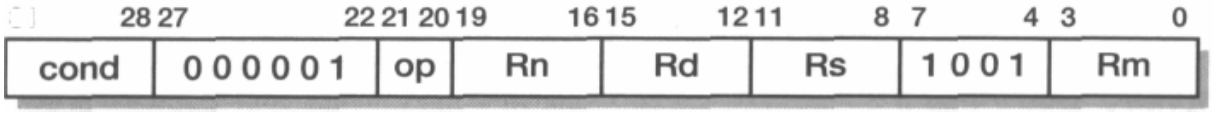
• c – Điều khiển field-PSR[7:0].
• x – Phần mở rộng của field-PSR[15:8] (không sử dụng ở mô hình ARMs hiện tại).
• s - trạng thái field - PSR[23:16] (không sử dụng ở mô hình ARMs hiện tại).
• f- Cờ của field -PSR[31:24].

XII.3. *Chú ý:*

+Người lập trình không thay đổi được giá trị CPSR[23:0]
+Tránh truy cập SPSR khi không thật cần thiết.

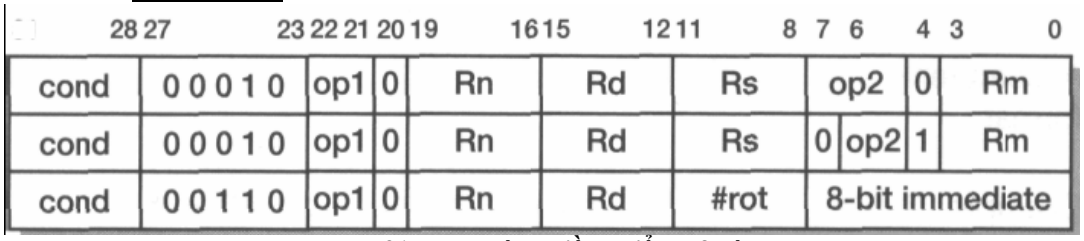
F.XIII. *Vùng không được dùng trong các chỉ lệnh:*

i. *Số học:*



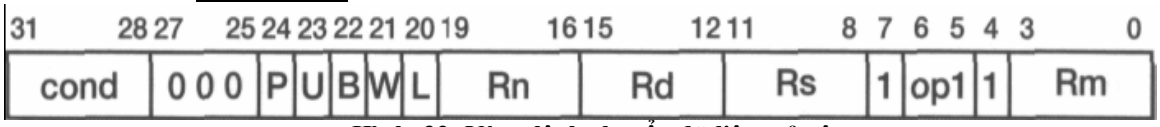
Hình 30: Vùng lệnh số học mở rộng

ii. *Điều khiển:*



Hình 31: Vùng lệnh điều khiển mở rộng

iii. *Load-store:*



Hình 32: Vùng lệnh chuyển dữ liệu mở rộng

iv. *Vùng lệnh không dùng tới:*

Hình 33: Vùng không được định nghĩa trong mã lệnh

F.XIV. Ghi chú:

Để biết thêm chi tiết và khoa học hơn về phần này, bạn đọc File: *Assemler Guide* của *ARM Developer Suite* có tại trang http://www.arm.com hoặc có thể load về tại đây.

G. Hỗ trợ của kiến trúc ARM cho ngôn ngữ cấp cao:

H. Tập lệnh Thumb:

<Các phần này, bạn xem thêm ở sách tiếng Anh *ARM-SoC Architecture*, ngay từ đầu tôi không muốn đi sâu vào các tổ chức phần cứng và hợp ngữ, ta sẽ đề cập đến nó ở phần khác, khi nói về ứng dụng và lập trình với C-Compiler>

I. Bộ nhớ cache:

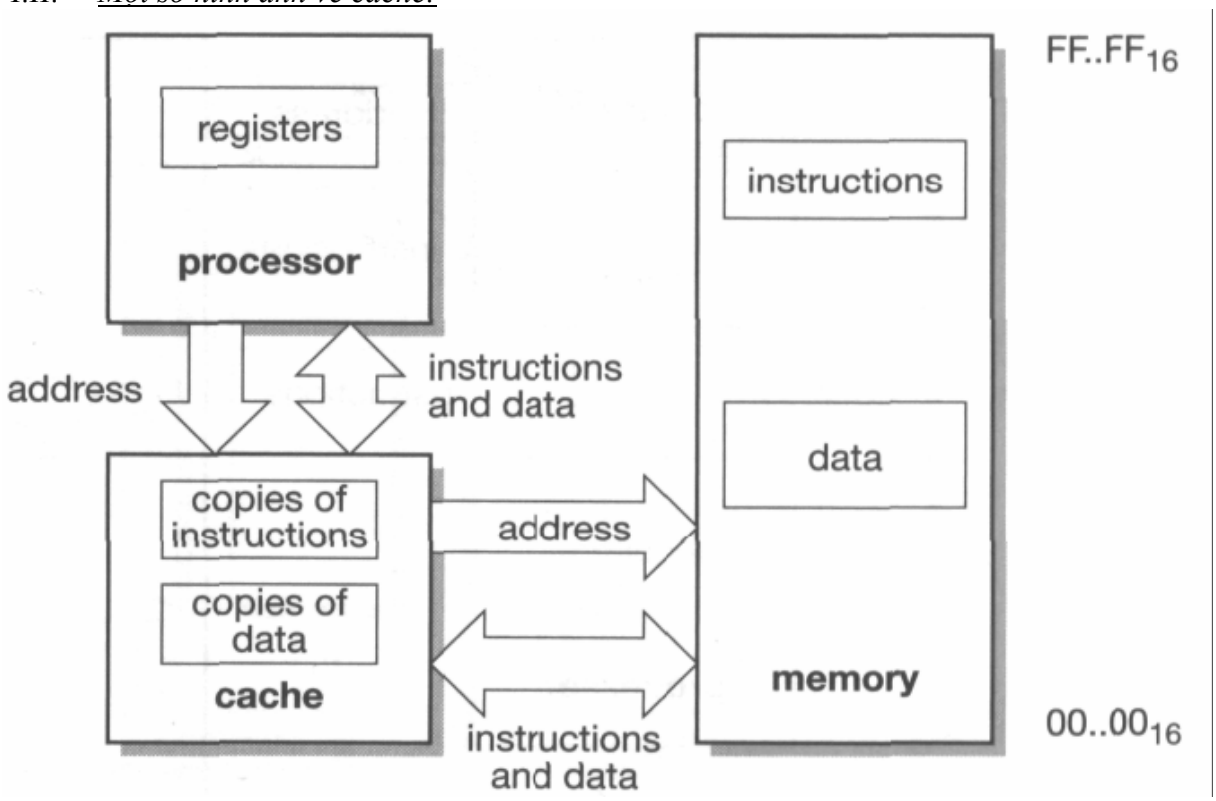
Phần này, ta tóm lược và xem thử tại sao lại dùng bộ nhớ phụ cache? Nó giúp ích gì cho ta?

I.I. Cache là gì?-Vì sao phải dùng cache:

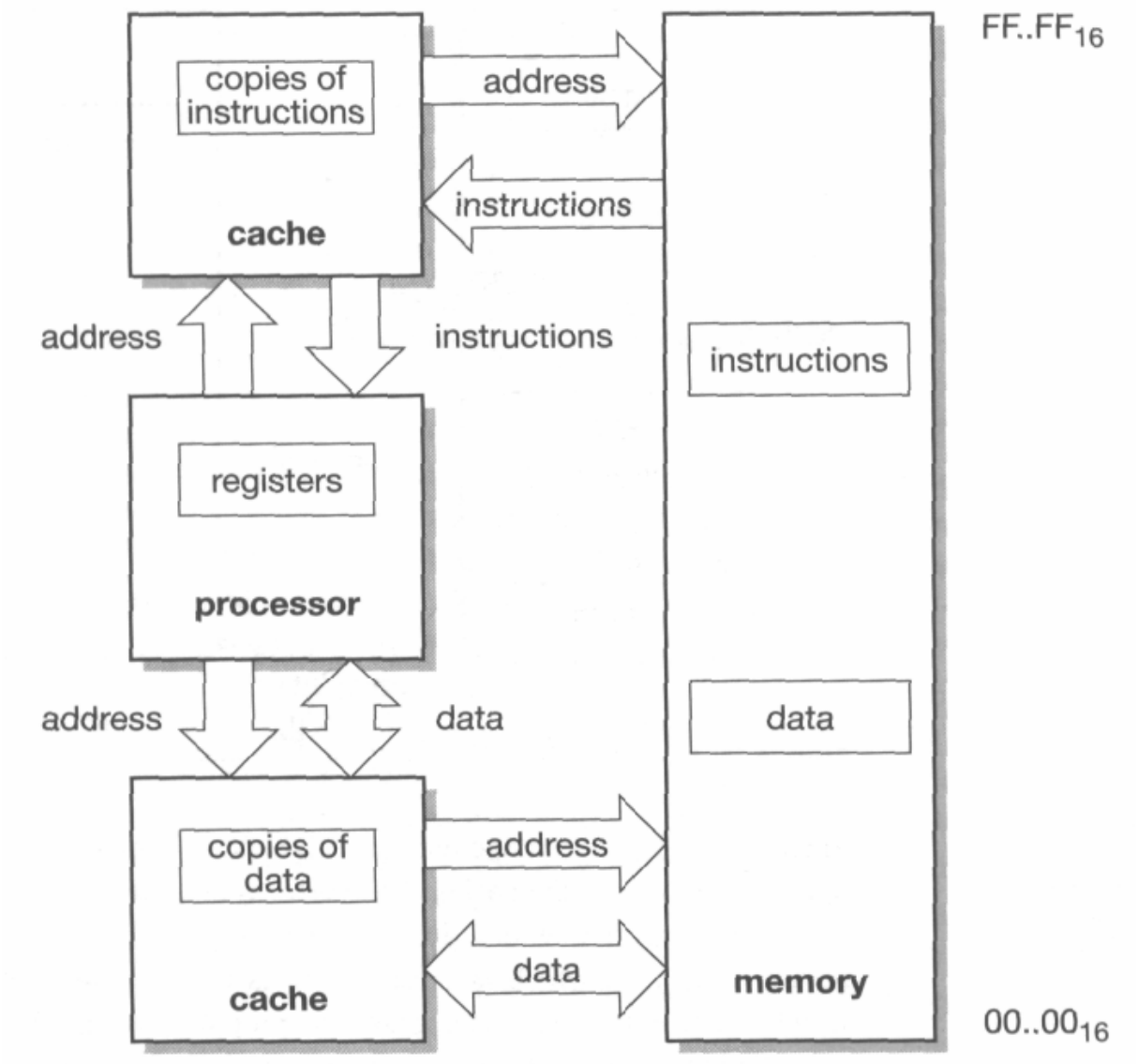
Nói một cách nhanh chóng và đơn giản nhất, cache là một bộ nhớ trung gian giữa bộ nhớ chính <ROM/RAM> và các thanh ghi đa dụng, có đáp ứng nhanh, chứa các dòng lệnh hay dùng. Khi mà việc truy cập địa chỉ hay giá trị vào các bộ nhớ DRAM hoặc ROM luôn bị giới hạn về đáp ứng phần cứng, cache được coi là một giải pháp tốt để thích nghi.

Cấu trúc bộ nhớ của cache cũng có thể phân ra 2 loại riêng biệt, loại chứa chung vùng dữ liệu và chỉ lệnh, loại tách rời chúng ra thành 2 bộ nhớ riêng biệt, Ta thường gọi là cấu trúc von Neuman và cấu trúc Harvard.

I.II. Một số hình ảnh về cache:



Hình 34: Cache dùng chung cho vùng nhớ dữ liệu và địa chỉ <Von-Neuman>



Hình 35: Cache có vùng nhớ dữ liệu và địa chỉ tách rời nhau <Cấu trúc Harvard>

J. Kết luận:

Trong phần trên, tôi đã xét qua một cách khái quát về ARM, từ lịch sử hình thành-phát triển đến một số diễn tả phần cứng-phần mềm. Dù đã rất cố gắng nhưng do kiến thức có giới hạn nên chắc còn nhiều sai sót, mong bạn lượng thứ!

Trong bài sau, tôi sẽ đi cụ thể vào lập trình với LPC2214, sử dụng chương trình Keil-uV3 tool ARM.

Mọi đóng góp và trao đổi xin gửi về:

Bùi Trung Hiếu
Email : buitrunghieu@khvt.com
Cell : (+84)98.3210.906
YM : khvt_sites

K. Tài lệu tham khảo chính:

1. ***ARM-SoC Architecture*** - Steve Furber- *Addison Wesley Publishing*-ISBN: 0-201-67519-6
2. ***ARM Developer Suite -Assemler Guide*** – ©2001 ARM Limited