GIÁO TRÌNH MÔN KIẾN TRÚC MÁY TÍNH

Chương 1: Máy tính các khái niệm và công nghệ

A. LÝ THUYẾT

1.1. Giới thiệu

- Máy tính – cuộc cách mạng thứ ba của nền văn minh

- Máy tính được sử dụng trong 3 lớp ứng dụng chính

+ Máy tính để bàn (Desktop computers)

~ Sử dụng cá nhân

~ Hiệu năng với người dùng đơn lẻ, giá cả thấp, thực thi chương trình bên phía 3 (shrink-wrap)

~ Tên gọi máy tính cá nhân (personal computer) hay máy tính đa dụng (general-purpose computer).

+ Máy chủ (Servers)

~ Chạy chương trình lớn hoặc có nhiều người dùng đồng thời và hình thức mạng

~ Có thể chạy ứng dụng đơn có tính phức tạp cao hoặc điều khiển nhiều công việc nhỏ

~ Dựa trên các phần mềm phát triển từ một nguồn khác, hiệu chỉnh cho phù hợp với chức năng cụ thể

~ Máy chủ được xây dựng theo cùng công nghệ như máy tính để bàn, nhưng cung cấp khả năng mở rộng lớn về mặt tính toán và số lượng các ngõ nhập xuất

~ Có nhiều loại: Low-end Severs (doanh nghiệp nhỏ, chi phí thấp 1000$), Supercomputer (Kĩ thuật khoa học phức tạp, RAM từ gigabytes đến terabytes, khả năng lưu trữ dữ liệu terabytes đến petabytes), Datacenter (cụm máy tính lớn)

+ Máy tính nhúng (Embedded computers)

~ Bên trong một thiết bị khác, bao gồm các vi xử lý **(microprocessor)**

~ Trong nhiều năm vừa qua, tốc độ phát triển máy tính nhúng nhanh hơn máy tính để bàn và máy chủ

→ Nội dung môn học này và sách tham khảo chính chủ yếu trình bày về máy tính đa dụng (general-purpose computer), tuy nhiên đa số các khái niệm đều có thể áp dụng trực tiếp (hoặc với một số hiệu chỉnh nhỏ) cho các máy tính nhúng.

1.2 Lịch sử phát triển

1.2.1. Thế hệ thứ nhất

- Linh kiện cơ bản chế tạo CPU là bóng đèn điện tử

- Bộ nhớ được chế tạo bằng role điện từ

- Chưa có hệ điều hành, lập trình thủ công bằng cách cắm cáp và cài đặt công tắc, dữ liệu vào được đục lỗ

- 1946 John von Neumann cùng cộng sự tạo ra IAS (Institue for Advanced Studies), gồm 3 thành phần chính: Bộ xử lý, bộ nhớ và nhập/xuất

1.2.2. Thế hệ thứ hai

- Linh kiện cơ bản chế tạo CPU là transistor bán dẫn được lắp đặt trên bảng mạch in

- Bộ nhớ được dùng là xuyến từ

- Hệ điều hành tuần tự (Batch OS), ngôn ngữ cấp cao xuất hiện, chương trình được thực hiện tuần tự

+ FORTRAN năm 1956

+ COBOL năm 1959

+ ALGOL năm 1960

1.2.3. Thế hệ thứ ba

- Linh kiện cơ bản chế tạo CPU là mạch tích hợp IC (Integrated Circuit), chứa hàng trăm linh kiện trên mạch tích hợp

- Hệ điều hành phân chia thời gian, có thể chạy tác vụ đa nhiệm

1.2.4 Thế hệ thứ tư

- IC có mật độ tích hợp cao, kích thước nhỏ, giảm tiêu thụ điện, có nhiều kĩ thuật được phát minh làm tăng tốc độ vi xử lý

- Hiện nay là xu hướng về máy tính lượng tử

1.3. Bên dưới chương trình ứng dụng

*** Phân làm 3 cấp:**

- Ứng dụng (Application):

- Phần mềm hệ thống (System software): Trung gian giữa tầng ứng dụng và phần cứng, làm cầu nối. Có nhiều phần mềm hệ thống, có hai loại điển hình nhất:

+ **Hệ điều hành (Operating System):** Điều hành chương trình, hỗ trợ chương trình chạy trên máy tính đó, là cầu nối giữa chương trình người dùng và phần cứng, đồng thời cung cấp nhiều dịch vụ khác nhau và các chức năng quản lý:

~ Điều khiển hoạt động nhập xuất cơ bản

~ Cấp phát bộ nhớ vùng lưu trữ

~ Quản lý chia sẻ tài nguyên máy tính khi có nhiều ứng dụng chạy đồng thời.

+ **Trình biên dịch (Compiler):** Dịch các câu lệnh ở ngôn ngữ cấp cao (C, Java) sang hợp ngữ (ngôn ngữ assembly, các tập lệnh mà phần cứng máy tính có thể thực thi). Việc biên dịch khá phức tạp do phức tạp của ngôn ngữ lập trình hiện đại và tính đơn giản của các lệnh thực thi bởi phần cứng

- Phần cứng (Hardware):

*** Từ ngôn ngữ cấp cao đến ngôn ngữ phần cứng:**

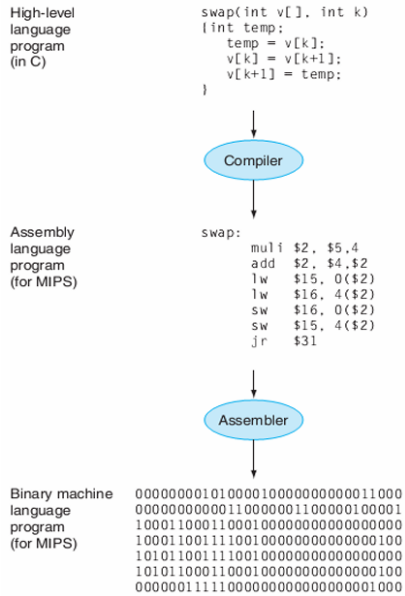
- Giao tiếp với máy tính là gửi đi các tính hiệu điện

+ Bảng chữ cái tiếng Anh có 26 kí tự

+ Chữ cái máy tính có hai kí tự 0 và 1 → Nhị phân, mỗi kí tự là một số nhị phân, còn gọi là bit

- Lệnh (Instruction): Một yêu cầu được đưa ra mà phần cứng máy tính có thể hiểu và đáp ứng

- Cách nhà lập trình giao tiếp với máy tính: Những nhà lập trình đầu tiên giao tiếp với máy tính thông qua các số nhị phân, một công việc khá buồn tẻ, và họ nhanh chóng tìm ra những cách viết mới gần gũi hơn với cách thức suy nghĩ của con người, đó là ngôn ngữ cấp cao.



+ **Ngôn ngữ Assembly (Hợp ngữ):** Ngôn ngữ mô tả lệnh của máy tính thông qua kí hiệu biểu diễn (symbol)

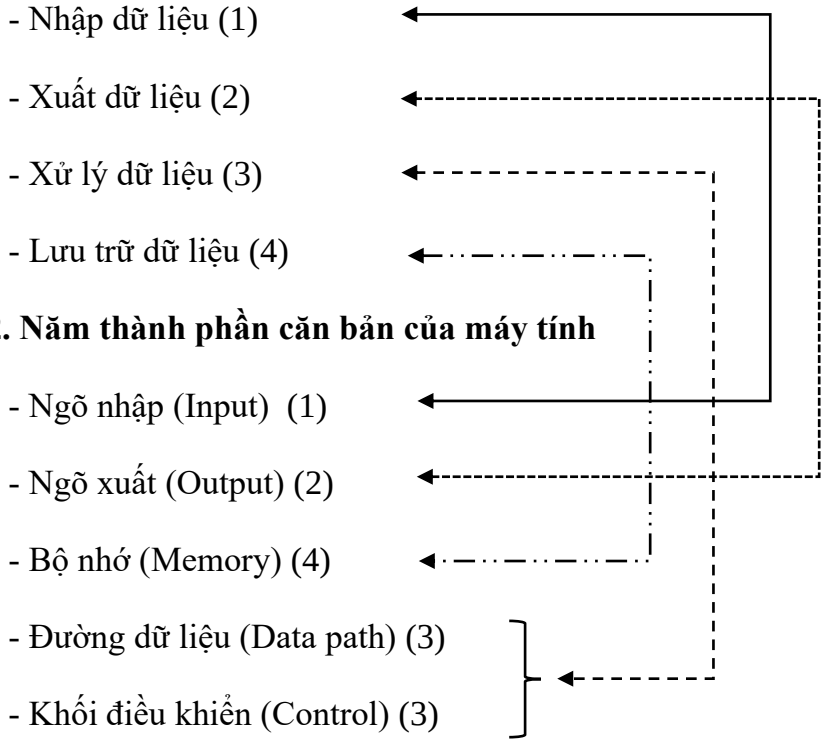
+ **Assembler:** Chương trình dịch lệnh hợp ngữ sang lệnh nhị phân.

+ **Ngôn ngữ lập trình cấp cao:** Các ngôn ngữ có tính linh động (portable) như C, Fortran, Java; bao gồm các từ và kí hiệu số học, có thể được dịch sang ngôn ngữ Assembly bởi một trình biên dịch (Complier)

~ ***Chú ý:*** *Gồm 2 bước, như hình, tuy nhiên một số trình biên dịch thì dịch trực tiếp từ cấp cao sang nhị phân*

1.4. Bên trong máy tính

1.4.1. Phần cứng máy tính thực hiện những chức năng sau:



1.4.2. Năm thành phần căn bản của máy tính

- Ngõ nhập (Input) (1)
- Ngõ xuất (Output) (2)
- Bộ nhớ (Memory) (4)
- Đường dữ liệu (Data path) (3)
- Khối điều khiển (Control) (3)

(Data path và Control thường được kết hợp lại với nhau với tên gọi bộ xử lý (Processor))

Bộ xử lý (Processor): Nhận lệnh và dữ liệu từ bộ nhớ để xử lý.

Ngõ nhập (input) ghi dữ liệu vào bộ nhớ, và ngõ xuất (output) đọc dữ liệu ra từ bộ nhớ.

Khối điều khiển (Control): Gởi các tín hiệu điều khiển hoạt động của đường dữ liệu, bộ nhớ, ngõ nhập và ngõ xuất

Màn hình (Screen): Thiết bị xuất

Bàn phím (Keyboard) và chuột (Mouse): Thiết bị nhập

Thùng máy (Case) chứa bộ xử lý và các thiết bị I/O khác

** Một số thiết bị vừa xuất vừa nhập: ổ đĩa (disk), card mạng*

*** Bốn thành phần chính của một máy tính đa dụng:**

- Bộ xử lý trung tâm (CPU - Central Processing Unit): Đây là thành phần chịu trách nhiệm thực hiện các lệnh từ phần mềm. CPU điều khiển và phối hợp hoạt động của các thành phần khác của máy tính.

- Bộ nhớ (Memory): Bao gồm RAM (Random Access Memory) để lưu trữ tạm thời các dữ liệu và lệnh đang được xử lý, và các loại bộ nhớ lưu trữ khác như ổ cứng (HDD/SSD) để lưu dữ liệu lâu dài.

- Thiết bị nhập/xuất (Input/Output Devices): Bao gồm các thiết bị như bàn phím, chuột, màn hình, máy in và các thiết bị ngoại vi khác, cho phép người dùng tương tác với máy tính.

- Hệ thống lưu trữ (Storage System): Là nơi lưu trữ dữ liệu và phần mềm, ví dụ như ổ đĩa cứng, SSD hoặc các thiết bị lưu trữ ngoài khác.

1.4.3. Bên trong thùng máy

- Board mạch chủ (Mother-board/Main-board): Là một bảng mạch bằng plastic, chứa các khối mạch tích hợp (Integrated circuits hay chips), gồm có bộ xử lý, cache, bộ nhớ, và kết nối cho các thiết bị I/O.

Mạch tích hợp (Integrated circuits): Còn được gọi là chip, chứa đựng hàng chục đến hàng triệu transistors

- Bộ nhớ (Memory): Là vùng lưu trữ chứa đựng chương trình đang chạy và chứa dữ liệu mà chương trình chạy cần dùng

+ RAM (Random access memory): Khác với các bộ nhớ truy cập tuần tự, như đĩa từ (magnetic tapes- sequential access memory), thời gian truy cập vào bất kì vị trí nào trong bộ nhớ RAM cơ bản là như nhau.

~ DRAM *(Dynamic random access memory):* sử dụng tụ điện để lưu trữ dữ liệu, cần được làm tươi liên tục, cho phép tích hợp mật độ cao trên một chip.

→ Rẻ, chậm hơn

~ SRAM *(Static random access memory):* Sử dụng các mạch logic (flip-flop) để lưu trữ dữ liệu. Không cần làm mới, dữ liệu được giữ nguyên miễn là có điện. Cấu trúc phức tạp hơn, mật độ tích hợp thấp hơn.

→ Nhanh, mắc

+ **DIMM (dual inline memory module):** Một board nhỏ chứa chip DRAM trên cả hai mặt của board. **SIMM (single inline memory module)** có DRAM chỉ trên một mặt.

- Đơn vị xử lý trung tâm (Central processor unit- CPU): cũng gọi là bộ xử lý (Processor), bộ phận hoạt động tích cực của máy tính, chứa đường dữ liệu (data path) và khối điều khiển (control), thực hiện việc như cộng số, kiểm tra số, kích hoạt các thiết bị I/O , v.v

+ Datapath: Thành phần của bộ xử lý, thực hiện các tính toán toán học, được xem là cơ bắp

+ Control: Thành phần của bộ xử lý, điều khiển đường dữ liệu, bộ nhớ, và các thiết bị I/O tùy theo lệnh nào đang thực thi của chương trình, được xem là bộ não

+ Cache:

~ Bên trong bộ xử lý còn có một dạng bộ nhớ, gọi là bộ nhớ đệm (Cache memory)

~ Bộ nhớ Cache là một bộ nhớ nhỏ, nhanh, hoạt động như một bộ đệm cho bộ nhớ DRAM

~ Cache được xây dựng trên một công nghệ thiết kế bộ nhớ khác biệt, dựa trên (SRAM). SRAM có tốc độ truy cập nhanh hơn và ít dày đặc hơn, do đó mắc hơn DRAM.

- Nơi lưu dữ liệu an toàn:

+ Các loại bộ nhớ:

~ Bộ nhớ khả biến/bay hơi (Volatile memory): Chỉ lưu dữ liệu khi có nguồn điện (VD: DRAM) → Là bộ nhớ chính (Main/Primary memory): dùng để lưu chương trình đang chạy

~ Bộ nhớ bất biến/không bay hơi (Nonvolatile memory): Có thể lưu trữ dữ liệu khi cả không được cung cấp nguồn điện. Đĩa từ là 1 ví dụ.

→ Là bộ nhớ thứ cấp (Secondary memory): Là bộ nhớ bất biến, dùng để lưu chương trình và dữ liệu giữa các lần chạy.

+ Đĩa từ (hard disk): bộ nhớ bất biến, cấu tạo

~ Đĩa từ (Platter): Là các đĩa kim loại hình tròn, được phủ một lớp vật liệu từ tính để lưu trữ dữ liệu2.

~ Đầu đọc/ghi (Read/Write Head): Di chuyển trên bề mặt đĩa để đọc và ghi dữ liệu2.

~ Cánh tay thiết bị truyền động (Actuator Arm): Điều khiển đầu đọc/ghi di chuyển đến vị trí cần thiết trên đĩa2.

~ Động cơ trục chính (Spindle Motor): Quay các đĩa từ với tốc độ cao để đầu đọc/ghi có thể truy cập dữ liệu2.

+ Đĩa quang: CDs (Compact disks) khoảng 700 MB và DVDs (Digital video disks) từ 4.7 GB đến 17 GB

~ Cấu tạo: Đĩa quang thường được làm từ nhựa hoặc thủy tinh, phủ một lớp phản chiếu và bảo vệ bởi một lớp vỏ trong suốt1.

~ Nguyên lý hoạt động: Dữ liệu được lưu trữ dưới dạng các "hố" (pits) và "đất" (lands) trên bề mặt đĩa. Khi tia laser chiếu vào bề mặt đĩa, nó sẽ phản xạ lại và được cảm biến đọc để giải mã thành dữ liệu nhị phân1.

+ Flash: bộ nhớ không bay hơi, dùng để thay thế cho bộ nhớ đĩa, cấu tạo bằng ô nhớ (cell), đọc bằng cách đo điện áp của ô nhớ, rẻ và chậm hơn Dram nhưng mắc và nhanh hơn đĩa từ, ứng dụng trong SSD, thẻ nhớ, điện thoại,…

- Giao tiếp với máy tính khác:

+ Qua mạng máy tính: Kết nối tất cả máy tính, cho phép người dùng máy tính mở rộng năng lực tính toán thông qua giao tiếp giữa các máy tính. Mạng máy tính ngày càng trở nên phổ biến và là xương sống cho các hệ thống máy tính hiện nay.

+ Máy tính có kết nối mạng có nhiều thuận lợi:

~ Giao tiếp với nhau: trao đổi thông tin tốc độ cao

~ Chia sẻ tài nguyên: chia sẻ các thiết bị I/O, dùng chung

~ Truy cập từ xa: điều khiển máy tính từ xa

+ Các phương thức kết nối mạng:

~ Ethernet (popular): Có thể dài 1km và bandwidth up to 10Gbps.

→ Kết nối máy tính trong phạm vi tòa nhà, là ví dụ của mạng cục bộ (**LAN – Local area network**)

~ **Wide area networks (WAN)**: xuyên lục địa. là xương sống của mạng internet. Có thể dài hàng trăm km và bandwidth up to thousands of Gbps. Thường dây làm bằng sợi quang (optical fibers) và được cung cấp bởi thông tin viễn thông

~ Không dây (Wireless technology): Phổ biến hiện nay là 802.11, cho phép truyền dữ liệu 1-100 triệu bit/giây

- Công nghệ xây dựng Bộ xử lý và Bộ nhớ:

+ Transistor: Công tắc đóng/mở được điều khiển bằng điện.

+ **Very large scale integrated circuit (VLSI):** Mạch tích hợp chứa hàng trăm ngàn đến hàng triệu transistor

+ Moore's law:

Số lượng transistor của mạch tích hợp sẽ tăng gấp đôi trong khoảng thời gian mỗi 18–24 tháng (Gordon Moore, một trong những nhà sáng lập Intel vào những năm 1960s.)

Định luật Moore dựa trên các quan sát thực nghiệm của Moore, định luật này được công bố lần đầu tiên vào năm 1965. Định luật Moore đã được dự đoán gần chính xác trong hơn 50 năm qua và đã góp phần quan trọng trong sự phát triển của công nghệ điện tử, đặc biệt là trong lĩnh vực sản xuất chip. Định luật Moore đã giúp các nhà sản xuất công nghệ tích hợp ngày càng nâng cao hiệu suất và giảm giá thành các linh kiện điện tử.

- Ảnh màu: Màn hình gồm các điểm ảnh nhỏ gọi là **Pixel** được tính bằng độ phân giải màn hình (1 Megapixel = 1 triệu pixel)

~ **Bit map** là một hình ảnh được thể hiện như một ma trận các bit

~ Sẽ có phần cứng tên *"raster refresh buffer",* hay còn gọi là *"frame buffer"* để lưu nội dung bitmap

~ Mỗi màu cần 8 bit (1 byte), mà biểu diễn cho 3 màu (đỏ, xanh lá, xanh dương) → mỗi pixel cần 8 x 3 = 24 bit x độ phân giải màn hình → số bit để lưu được **trọn 1 hình**

1.5. Phân biệt kiến trúc RIST và CISC

Sự khác biệt giữa kiến trúc RISC (Reduced Instruction Set Computer - Máy tính với tập lệnh rút gọn) và CISC (Complex Instruction Set Computer - Máy tính với tập lệnh phức tạp) chủ yếu nằm ở cách tiếp cận trong thiết kế tập lệnh và hiệu quả phần cứng:

1.5.1. RISC (Reduced Instruction Set Computer):

- Kiến trúc RISC tập trung vào một tập lệnh nhỏ, được tối ưu hóa cao. Mỗi lệnh được thiết kế để thực thi trong một chu kỳ xung nhịp, điều này giúp đơn giản hóa logic điều khiển của bộ xử lý và cải thiện hiệu suất thông qua cách xử lý đường ống.

- Bộ xử lý RISC cần nhiều lệnh đơn giản hơn để thực hiện các thao tác phức tạp, điều này có thể làm mã lệnh lớn hơn nhưng cho phép thực thi nhanh hơn.

- Các ví dụ về bộ xử lý RISC bao gồm ARM và MIPS

1.5.2. CISC (Complex Instruction Set Computer):

- Kiến trúc CISC, như dòng Intel x86, có một tập lệnh lớn hơn có thể thực hiện các tác vụ phức tạp với ít dòng mã hơn. Điều này có thể làm giảm số lượng lệnh trong một chương trình nhưng có thể yêu cầu nhiều chu kỳ cho mỗi lệnh do độ phức tạp của từng thao tác.

- Độ phức tạp của các lệnh CISC khiến đơn vị điều khiển và phần cứng trở nên phức tạp hơn, gây ra thách thức trong tối ưu hóa hiệu suất.

- Kiến trúc x86 là một ví dụ của bộ xử lý CISC, được tối ưu hóa cho khả năng tương thích nhị phân nhưng thường chuyển các lệnh phức tạp thành các vi lệnh đơn giản hơn để xử lý hiệu quả

Tóm lại, thiết kế RISC ưu tiên các lệnh đơn giản, nhanh chóng phù hợp cho xử lý đường ống, trong khi CISC sử dụng một tập lệnh phức tạp hơn nhằm giảm số lượng lệnh cho các thao tác phức tạp.

1.6. Sự khác nhau giữa kiến trúc máy tính và tổ chức máy tính

- Kiến trúc máy tính: Kiến trúc máy tính đề cập đến thiết kế khái niệm và cấu trúc hoạt động cơ bản của một hệ thống máy tính. Nó phác thảo khả năng của hệ thống, bộ lệnh và cách các thành phần tương tác. Kiến trúc máy tính (Computer architecture) đề cập đến các thuộc tính của hệ thống máy tính dưới góc nhìn của con người lập trình. Hay nói cách khác là ảnh hưởng trực tiếp đến quá trình thực hiện logic của chương trình bao gồm: tập lệnh, biểu diễn dữ liệu, các cơ chế vào/ra, kỹ thuật gán địa chỉ…

- Tổ chức máy tính: Tổ chức máy tính đề cập đến các đơn vị hoạt động và các kết nối của chúng thực hiện kiến trúc. Nó liên quan đến việc triển khai vật lý và cách các thành phần được sắp xếp và làm việc cùng nhau.

⟹ Tóm lại, kiến trúc máy tính là về thiết kế trừu tượng và chức năng của một hệ thống máy tính, là khoa học về lựa chọn và kết nối các thành phần phần cứng để đáp ứng yêu cầu về hiệu năng, chức năng, giá cả. trong khi tổ chức máy tính liên quan đến việc triển khai vật lý và kết nối các thành phần đó, là khoa học về nghiên cứu các bộ phận của máy tính và phương thức làm việc của chúng

Khía cạnh	Kiến trúc máy tính	Tổ chức máy tính
Định nghĩa	Là những đặc tả liên quan đến cách máy tính hoạt động và giao tiếp với phần mềm.	Là cách các thành phần phần cứng của máy tính được sắp xếp và kết nối để thực hiện chức năng.
Mục tiêu chính	Tập trung vào cách thiết kế hệ thống để đạt hiệu suất, chức năng và khả năng tương thích với phần mềm.	Tập trung vào cách phần cứng thực hiện chức năng dựa trên thiết kế kiến trúc đã định.

B. BÀI TẬP

1.5. Các hệ đếm cơ bản

1.5.1 Hệ nhị phân

- Hệ cơ số 2 (gồm 2 chữ số nhị phân: 0 và 1)

- Dùng n bit có thể biểu diễn được 2^n giá trị khác nhau

*** Phương pháp chuyển đổi thập phân – nhị phân:**

- Phương pháp 1: chia dần cho 2 rồi lấy phần dư (đọc ngược)

- 105:2 = 52 dư 1
- 52:2 = 26 dư 0
- 26:2 = 13 dư 0
- 13:2 = 6 dư 1
- 6:2 = 3 dư 0
- 3:2 = 1 dư 1
- 1:2 = 0 dư 1

Kết quả: $105_{(10)} = 1101001_{(2)}$

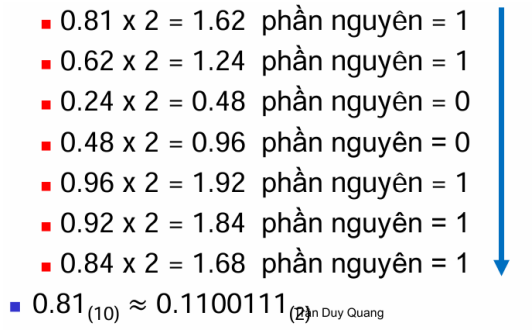
- Phương pháp 2: phân tích thành tổng của các số lũy thừa của 2 (2^k)

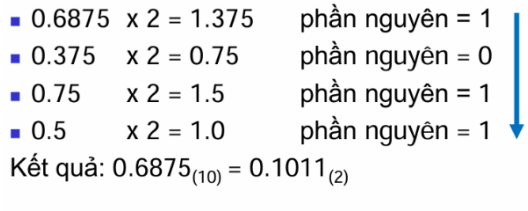
- $105 = 64 + 32 + 8 + 1 = 2^6 + 2^5 + 2^3 + 2^0$

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	8	4	2	1
0	1	1	0	1	0	0	1

- Kết quả: $105_{(10)} = 0110\ 1001_{(2)}$

*** Phương pháp chuyển đổi thập phân – nhị phân:**





1.5.2 Hệ mười sáu

- Cơ số 16 (0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F)

* Cách chuyển đổi:

- Từ số nhị phân sang số Hex: cứ một nhóm 4 bit (một Nibble) sẽ được thay thế bằng 1 chữ số Hex

4 bit	Chữ số Hex	4 bit	Chữ số Hex
0000	0x0	1000	0x8
0001	0x1	1001	0x9
0010	0x2	1010	0xA
0011	0x3	1011	0xB
0100	0x4	1100	0xC
0101	0x5	1101	0xD
0110	0x6	1110	0xE
0111	0x7	1111	0xF

- Từ số Hex sang số nhị phân: cứ 1 chữ số Hex sẽ được thay thế bằng một nhóm 4 bit

- Từ số Hex sang số thập phân:

Ví dụ: $1C_{(16)} = 1x16^1 + 12x16^0 = 16 + 12 = 28_{(10)}$

1.6. Các phép tính trong hệ nhị phân

- Phép cộng:

0 + 0 = 0 0 + 1 = 1 1 + 0 = 1 1 + 1 = 0 nhớ 1 (đem qua bit cao hơn)

- Phép trừ:

0 - 0 = 0 1 - 0 = 1 1 - 1 = 0 0 - 1 = 1 nhớ 1 (mượn qua bit cao hơn)

1.7. Biểu diễn số nguyên

1.7.1 Biểu diễn số nguyên không dấu

- Dải biểu diễn của A: từ 0 đến 2^n -1
- Tràn nhớ nếu số đó vượt quá 2^n -1

1.7.2 Biểu diễn số nguyên có dấu

- Dải biểu diễn của A: từ -2^{n-1} đến 2^{n-1} -1

*** Có 3 phương pháp:**

- Lấy bit đầu biểu hiện cho dấu

- Bằng số bù hai:

+ Giả sử có: **A = 0010 0101**

+ Số bù một của **A = 1101 1010**

+ 1

+ Số bù hai của **A = 1101 1011**

+ Kí hiệu số bù hai của A: **$CP_2(A)$**

+ Vì $A + CP_2(A) = 0 \rightarrow CP_2(A) = -A$

+ Nếu A là số dương: bit $a_{n-1} = 0$. Nếu A là số âm: bit $a_{n-1} = 1$.

- Nợ n (excess n) [bias = n]

1.7.3 Trừ số nguyên bằng số bù hai

- Biến đổi **X – Y = X + (-Y)**
- Bỏ qua số nhớ ra ngoài bit cao nhất

*** Chuyển đổi Byte thành Word:**

- Đối với số dương:

+ 19 = 0001 0011 (8 bit)

+ 19 = 0000 0000 0001 0011 (16 bit)

- Đối với số âm:

- 19 = 1110 1101 (8 bit)

- 19 = 1111 1111 1110 1101 (16 bit)

1.7.4 Cộng số nguyên không dấu

- Nếu cộng n-bit với n-bit

+ Ra n-bit thì không bị tràn $C_{out} = 0$

+ Ra > n-bit thì bị tràn nhớ $C_{out} = 1$

- Xảy ra khi tổng > $2^n - 1$

1.7.5 Cộng số nguyên có dấu

- Nếu cộng hai số khác dấu: kết quả luôn luôn đúng

- Cộng hai số cùng dấu:

+ Nếu kết quả cùng dấu với các số hạng thì kết quả là đúng

+ Nếu kết quả có dấu ngược lại, khi đó có tràn xảy ra

1.8. Biểu diễn số thực có dấu chấm động

- Scientific notation: Một số thực được gọi là "scientific notation" khi bên trái dấu chấm có đúng 1 chữ số.

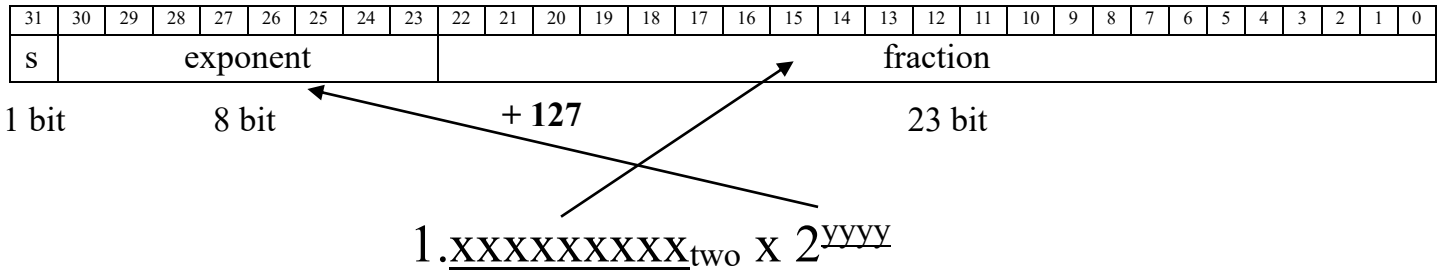
- Normalized number: Một số thực được gọi là "Normalized number" (dạng chuẩn) khi số này được viết trong "scientific notation" và chữ số bên trái dấu chấm không phải là 0.

- Số thực dấu chấm động: **Định nghĩa:** Trong máy tính, các số nhị phân phải được đưa về dạng chuẩn như sau:

$$1.xxxxxxxxx_{two} \times 2^{yyyy}$$

- Cách biểu diễn số thực dấu chấm động

+ Biểu diễn số thực dấu chấm động theo chuẩn **IEEE 754** (với độ chính xác đơn) (chuẩn này được áp dụng cho hầu hết các máy tính được chế tạo từ năm 1980)



+ Trong đó:

s biểu diễn dấu của số thực dấu chấm động (1: - ; 0: +)

Phần mũ(exponent) có kích thước là 8 bit. Exponent là biểu diễn quá 127 của yyyy (excess-127 hoặc bias of 127).

Exponent = y + 127

Phần lẻ (fraction) dùng 23 bits để biểu diễn cho xxxxxxxxx

+ Tổng quát, số thực chấm động được tính dựa theo Bias = 127

$$(-1)^S \text{ x } (1 + \text{Fraciton}) \text{ x } 2^{(\text{Exponent - Bias})}$$

+ Số nhỏ nhất: $\pm \mathbf{1.0000\ 0000\ 0000\ 0000\ 0000\ 0000}_{two} \times \mathbf{2}^{-126}$

+ Số lớn nhất: $\pm \mathbf{1.1111\ 1111\ 1111\ 1111\ 1111\ 1111}_{two} \times \mathbf{2}^{+127}$

- **Các vấn đề cần lưu ý:**

+ Tăng số bit chứa fraction thì tăng độ chính xác.

+ Tăng kích thước phần exponent là tăng tầm trị biểu diễn.

➔ Vì vậy, khi thiết kế một biểu diễn/thể hiện cho số dấu chấm động (ví dụ không sử dụng IEEE 754) thì tùy vào mục đích sử dụng mà lựa chọn số giới hạn cho fraction và exponent sao cho phù hợp nhất.

- Tràn trên (Overflow): trường hợp này xảy ra khi kích thước của số mũ lớn hơn kích thước giới hạn trên (số mũ dương), làm cho số biểu diễn làm tròn về vô cực

- Tràn dưới (Underflow): trường hợp này xảy ra khi kích thước của số mũ nhỏ hơn kích thước giới hạn dưới (số mũ âm), làm cho số được làm tròn về 0

+ Độ chính xác đơn (Single precision): một số thực dấu chấm động được biểu diễn ở dạng 32 bit. (float)

+ Độ chính xác kép (Double precision): một số thực dấu chấm động được biểu diễn ở dạng 64 bit (double). E: 11 bits, F: 52 bits (Bias = 1023)

Chú ý:

- Tăng số bit chứa phần fraction thì tăng độ chính sát

- Tăng kích thước của phần exponent là tăng tầm trị biểu diễn

→ Nên điều chỉnh cho phù hợp vào yêu cầu bài toán

Single precision		Double precision		Object represented
Exponent	Fraction	Exponent	Fraction	
0	0	0	0	0
0	Nonzero	0	Nonzero	± denormalized number
1-254	Anything	1-2046	Anything	± floating-point number
255	0	2047	0	± infinity
255	Nonzero	2047	Nonzero	NaN (Not a Number)

*** Các bước giải bài tập chuyển đổi từ tập phân sang dấu chấm động:**

- Biến đổi về dạng số thực chuẩn

- Tìm S = 1(0)

- Tìm E

- Tìm F

1.9. Hiệu suất máy tính

Thời gian đáp ứng là tổng thời gian để máy tính hoàn thành một tác vụ nào đó, bao gồm truy cập ổ đĩa, truy cập bộ nhớ, hoạt động I/O, thời gian thực thi của hệ điều hành, so on.

Banhwidth (Throughput): số lượng tác vụ hoàn thành trong một đơn vị thời gian

*** Một số công thức cần chú ý:**

$$\textbf{Performance}_x = \frac{1}{Execution\ time_x}$$

Nếu: $\textbf{Performance}_x > \textbf{Performance}_y$

$\textbf{Execution time}_y > \textbf{Execution time}_x$

Gọi: ET: Execution time: thời gian thực thi chương trình

NC: Number of clock cycles

T: Clock cycle

f: Clock rate/ Clock frequency

CPI: Clock cycle per instruction: Số chu kì xung cần để thực thi một lệnh

NI: Number of instruction: Tổng số lệnh cho một chương trình

$\textbf{ET = NC*T} = \frac{NC}{f}$ **(với T = 1/f) (1)**

NC = NI * CPI

(1) $\rightarrow$ $\textbf{ET = NI * CPI * T} = \frac{\textbf{NI} * \textbf{CPI}}{f}$

Gọi: MIPS (Million instructions per): Số tập triệu lệnh trên giây mà máy thực hiện được

$$\textbf{MIPS} = \frac{NI}{ET\ X\ 10^6} = \frac{f}{CPI\ X\ 10^6}$$

$$\textbf{CPI}_{tb} = \frac{NC_{total}}{NI_{total}}$$

Chương 2: Kiến trúc bộ lệnh

2.1. Giới thiệu

- Để ra lệnh cho máy tính ta phải nói với máy tính bằng ngôn ngữ của máy tính. Các từ của ngôn ngữ máy tính gọi là các lệnh (instructions) và tập hợp tất cả các từ gọi là bộ lệnh (instruction set)

- Hai bộ lệnh thông dụng nhất ngày nay: + ARM (similar MIPS)

+ Intel x86

2.2. Các phép tính

Ví dụ: add a, b, c **# a = b + c**

Một số lệnh trên MIPS

Category	Instruction	Example	Meaning
Arithmetic	add	add $s1,$s2,$s3	$s1 = $s2 + $s3
	subtract	sub $s1,$s2,$s3	$s1 = $s2 – $s3
	add immediate	addi $s1,$s2,20	$s1 = $s2 + 20
Data transfer	load word	lw $s1,20($s2)	$s1 = Memory[$s2 + 20]
	store word	sw $s1,20($s2)	Memory[$s2 + 20] = $s1
	load half	lh $s1,20($s2)	$s1 = Memory[$s2 + 20]
	load half unsigned	lhu $s1,20($s2)	$s1 = Memory[$s2 + 20]
	store half	sh $s1,20($s2)	Memory[$s2 + 20] = $s1
	load byte	lb $s1,20($s2)	$s1 = Memory[$s2 + 20]
	load byte unsigned	lbu $s1,20($s2)	$s1 = Memory[$s2 + 20]
	store byte	sb $s1,20($s2)	Memory[$s2 + 20] = $s1
	load linked word	ll $s1,20($s2)	$s1 = Memory[$s2 + 20]
	store condition. word	sc $s1,20($s2)	Memory[$s2+20]=$s1;$s1=0 or 1
	load upper immed.	lui $s1,20	$s1 = 20 * 2^{16}
Logical	and	and $s1,$s2,$s3	$s1 = $s2 & $s3
	or	or $s1,$s2,$s3	$s1 = $s2 | $s3
	nor	nor $s1,$s2,$s3	$s1 = ~ ($s2 | $s3)
	and immediate	andi $s1,$s2,20	$s1 = $s2 & 20
	or immediate	ori $s1,$s2,20	$s1 = $s2 | 20
	shift left logical	sll $s1,$s2,10	$s1 = $s2 << 10
	shift right logical	srl $s1,$s2,10	$s1 = $s2 >> 10

2.3. Toán hạng

*** 3 loại toán hạng:**

2.3.1. Toán hạng thanh ghi (Register Operands)

- Các toán hạng được đặt trong các vị trí đặc biệt được gọi là **thanh ghi**

- Kích thước thanh ghi trong kiến trúc MIPS là 32 bit; nhóm 32 bit xuất hiện thường xuyên nên chúng được đặt tên là "từ" (word)

(Lưu ý: một "từ" trong kiến trúc bộ lệnh khác có thể không là 32 bit)

- Một sự khác biệt lớn giữa các biến của một ngôn ngữ lập trình và các biến thanh ghi là số thanh ghi bị giới hạn (thường là 32 thanh ghi trên các máy tính hiện nay)

*** Các thanh ghi trong MIPS:**

NAME	NUMBER	USE	PRESERVED ACROSS A CALL?
$zero	0	The Constant Value 0	N.A.
$at	1	Assembler Temporary	No
$v0-$v1	2-3	Values for Function Results and Expression Evaluation	No
$a0-$a3	4-7	Arguments	No
$t0-$t7	8-15	Temporaries	No
$s0-$s7	16-23	Saved Temporaries	Yes
$t8-$t9	24-25	Temporaries	No
$k0-$k1	26-27	Reserved for OS Kernel	No
$gp	28	Global Pointer	Yes
$sp	29	Stack Pointer	Yes
$fp	30	Frame Pointer	Yes
$ra	31	Return Address	Yes

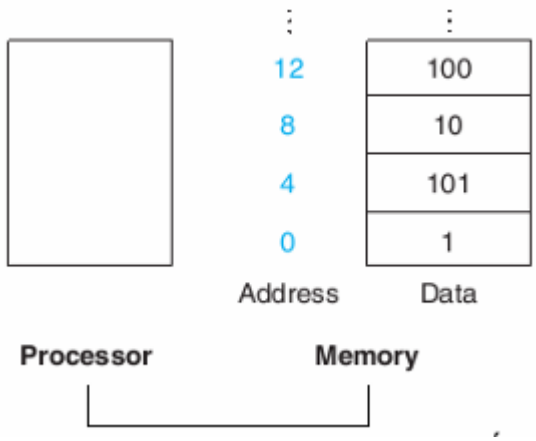
2.3.2. Toán hạng bộ nhớ (Memory Operands)

- MIPS có các lệnh chuyển dữ liệu giữa bộ nhớ và thanh ghi

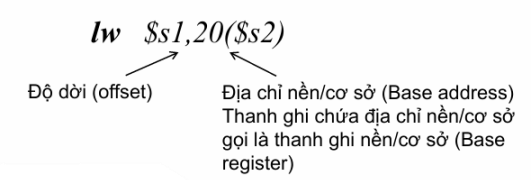
- Lệnh đó phải cung cấp **địa chỉ** bộ nhớ

- Bộ nhớ chỉ là một mảng đơn chiều lớn, với địa chỉ đóng vai trò là chỉ số trong mảng đó, bắt đầu từ 0

- Mỗi từ nhớ (word) của MIPS là 4 bytes. MIPS định địa chỉ theo byte, địa chỉ của mỗi word là địa chỉ của byte đầu tiên trong word đó → địa chỉ của mỗi word là bội 4



*** Lệnh lấy dữ liệu từ bộ nhớ vào thanh ghi (lw – load word)**



~ $1: thanh ghi nạp dữ liệu vào

~ Một hằng số (20) và thanh ghi ($s2) được sử dụng để truy cập vào bộ nhớ. Tổng số của hằng số và nội dung của thanh ghi này là địa chỉ bộ nhớ của phần tử cần truy cập đến. Nội dung của từ nhớ này sẽ được đưa từ bộ nhớ vào thanh ghi $s1

Ví dụ lw: Giả sử rằng A là một mảng của 100 phần tử (mỗi phần tử cần 1 word lưu trữ) và trình biên dịch đã kết hợp các biến g và h với các thanh ghi $s1 và $s2. Giả định rằng địa chỉ bắt đầu của mảng A (hay địa chỉ cơ sở/nền) chứa trong $s3. Hãy biên dịch đoạn lệnh bằng ngôn ngữ C sau sang MIPS:

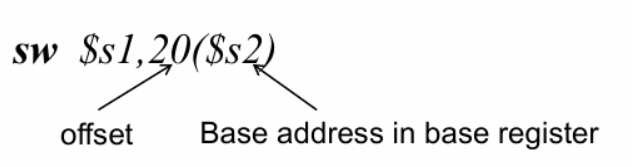
g = h + A[8];

→ *lw $t0, **32**($s3)* # $t0 nhận A[8]

add $s1, $s2, $t0 # g = h + A[8]

Vì 32 bits = 4 bytes nên độ dời phải là 32

*** Lệnh lưu dữ liệu từ thanh ghi ra bộ nhớ (sw – store word)**



~ $s1: thanh ghi chứa dữ liệu cần lưu.

~ Một hằng số (20) và thanh ghi ($s2) được sử dụng để truy cập vào bộ nhớ. Tổng số của hằng số và nội dung của thanh ghi này là địa chỉ bộ nhớ, nơi mà nội dung đang chứa trong thanh ghi $s1 sẽ được lưu vào đây.

Ví dụ **sw**: Giả sử biến h được kết nối với thanh ghi $s2 và địa chỉ cơ sở của mảng A là trong $s3. Biên dịch câu lệnh C thực hiện dưới đây sang MIPS?

A[12] = h +A[8];

→ *lw $t0, 32($s3)* # $t0 = A[8]

add $t0, $s2, $t0 # $t0 = h + A[8]

sw $t0, 48($s3) # A[12] = $t0

*** Phân biệt Leftmost (“Big End”, “Big Endian”) – Rightmost(“Little End”, “Little Endian”)**

- Big Endian: bit cao lưu địa chỉ thấp, bit thấp lưu địa chỉ cao

- Little Endian: bit cao lưu địa chỉ cao, bia thấp lưu địa chỉ thấp

0xFF00AA11.

Little Endian	
Address	Contents
4003	FF
4002	00
4001	AA
4000	11

Big Endian	
Address	Contents
4003	11
4002	AA
4001	00
4000	FF

2.3.3. Toán hạng bằng (Constant or Immediate Operands)

- Có sử dụng hằng số trong bài thì được gọi là toán hạng bằng

- Ví dụ: **addi** *$s3, $s3, 4* **# $s3 = $s3 + 4**

- Thực tế, các phiên bản khác của MIPS làm việc với thanh ghi 64 bits (MIPS-64), nhưng trong phạm vi môn học, chỉ học MIPS-32

2.4. Số có dấu và không dấu

2.5. Biểu diễn lệnh

- Máy chỉ có hiểu "0" và "1" nên cần phải chuyển đổi từ một lệnh sang mã máy (machine code) sử dụng định dạng lệnh

- Định dạng lệnh: Biểu diễn của một lệnh được biểu hiện bằng số nhị phân

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- Trường đầu tiên (op, tức opcode có giá trị 0) và trường cuối cùng (funct, tức function có giá trị 20hex) kết hợp báo cho máy tính biết rằng đây là lệnh cộng (add).

- Trường thứ hai (rs) cho biết toán hạng thứ nhất của phép toán cộng (rs hiện có giá trị 17, tức toán hạng thứ nhất của phép công là thanh ghi $s1)

- Trường thứ ba (rt) cho biết toán hạng thứ hai của phép toán cộng (rt hiện có giá trị 18, tức toán hạng thứ hai của phép công là thanh ghi $s2)

- Trường thứ tư (rd) là thanh ghi đích chứa tổng của phép cộng (rd hiện có giá trị 8, tức thanh ghi đích chứa tổng là $t0).

- Trường thứ năm (shamt) không sử dụng trong lệnh add này

*** Các dạng khác nhau của định dạng lệnh MIPS:**

2.5.1 R- Format.

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- rs: Thanh ghi chứa toán hạng nguồn thứ nhất
- rt: Thanh ghi chứa toán hạng nguồn thứ hai
- rd: Thanh ghi toán hạng đích, nhận kết quả của các phép toán.
- shamt: Chỉ dùng trong các câu lệnh dịch bit (shift)- chứa số lượng bit cần dịch (không được sử dụng sẽ chứa 0)
- funct: Kết hợp với op (khi op bằng 0) để cho biết mã máy là lệnh gì

2.5.2 I – Format và J – Format

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

- Lw, sw thì 16 bits đó sẽ ghi địa chỉ của vùng nhớ mà lệnh này truy cập đến
- Với lệnh khác (như addi, subi): 16 bits sẽ chứa số tức thời

op	address
6 bits	26 bits

Vùng "address" là vùng chứa số 26 bit (dùng cho lệnh 'j')

Bảng một số lệnh MIPS và các trường tương ứng

Instruction	Format	op	rs	rt	rd	shamt	funct	address
add	R	0	reg	reg	reg	0	32_{ten}	n.a.
sub (subtract)	R	0	reg	reg	reg	0	34_{ten}	n.a.
add immediate	I	8_{ten}	reg	reg	n.a.	n.a.	n.a.	constant
lw (load word)	I	35_{ten}	reg	reg	n.a.	n.a.	n.a.	address
sw (store word)	I	43_{ten}	reg	reg	n.a.	n.a.	n.a.	address

*** Chuyển về mã máy:**

VD: lw $t0,1200($t1) #Dùng thanh ghi tạm $t0 nhận A[300]

add $t0,$s2,$t0 #Dùng thanh ghi tạm $t0 nhận h +A[300]

sw $t0,1200($t1) #Lưu h +A[300]trở lại vào A[300]

100011	**01001**	**01000**	**0000 0100 1011 0000**		
000000	**10010**	**01000**	**01000**	**00000**	**100000**
101011	**01001**	**01000**	**0000 0100 1011 0000**		

2.6. Các phép tính logic

Logical operations	C operators	Java operators	MIPS instructions
Shift left	<<	<<	sll
Shift right	>>	>>>	srl
Bit-by-bit AND	&	&	and, andi
Bit-by-bit OR	|	|	or, ori
Bit-by-bit NOT	~	~	nor

Instruction	Meaning	R-Type Format					
and $s1, $s2, $s3	$s1 = $s2 & $s3	op = 0	rs = $s2	rt = $s3	rd = $s1	sa = 0	f = 0x24
or $s1, $s2, $s3	$s1 = $s2 | $s3	op = 0	rs = $s2	rt = $s3	rd = $s1	sa = 0	f = 0x25
xor $s1, $s2, $s3	$s1 = $s2 ^ $s3	op = 0	rs = $s2	rt = $s3	rd = $s1	sa = 0	f = 0x26
nor $s1, $s2, $s3	$s1 = ~($s2|$s3)	op = 0	rs = $s2	rt = $s3	rd = $s1	sa = 0	f = 0x27

sll $s1, $s2, 2	$s1 = $s2 << 2	Op = 0	Rs = 0	Rt = $s2	Rd = $s1	Sa = 2	f= 0x0
srl $s1, $s2, 2	$s1 = $s2 >> 2	Op = 0	Rs = 0	Rt = $s2	Rd = $s1	Sa = 2	f= 0x2

2.7 Các lệnh điều kiện và nhảy

Conditional branch	branch on equal	beq $s1, $s2 25	if ($s1 == $s2) goto PC + 4 + 100
	branch on not equal	bne $s1, $s2, 25	if ($s1 != $s2) goto PC + 4 + 100
	set on less than	slt $s1, $s2, $s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0
	set on less than unsigned	slt $s1, $s2, $s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0
	set on less than immediate	slt $s1, $s2, 20	if ($s2 < 20) $s1 = 1; else $s1 = 0
	set on less than immediate unsigned	slt $s1, $s2, 20	if ($s2 < 20) $s1 = 1; else $s1 = 0

Cặp (slt → beq) tương đương với if(… ≥ …) goto…
Cặp (slt → bne) tương đương với if(… < …) goto…

Unconditional jump	jump	j label
	jump register	jr $ra
	jump and link	jal label

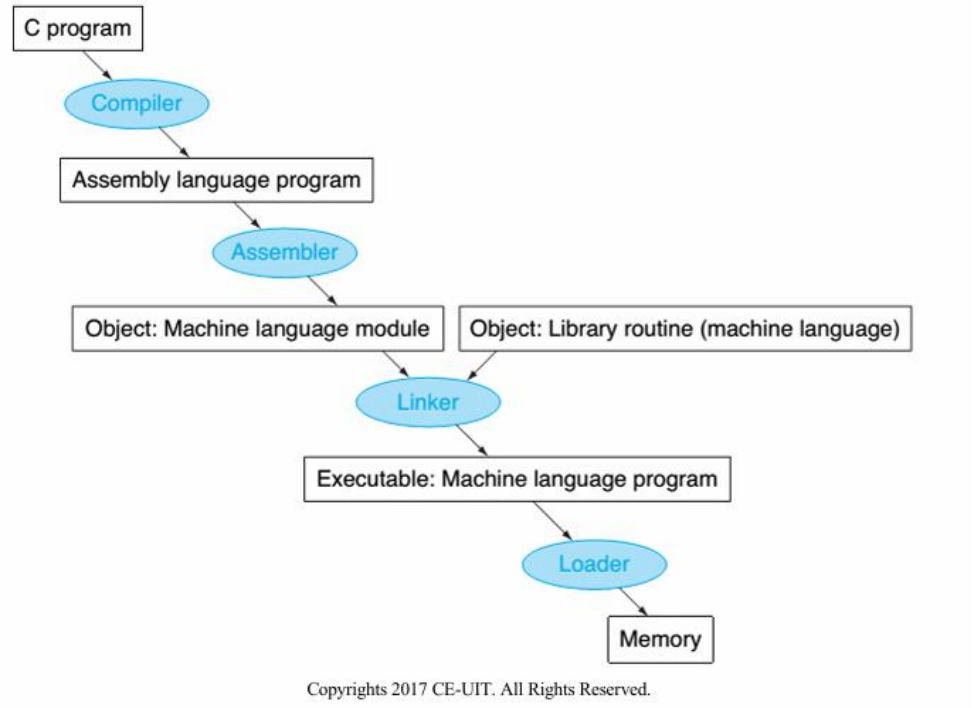
If else

```
        bne $s3,$s4,Else      # go to Else if i != j
        add $s0, $s1, $s2     # f = g + h (skipped if i != j)
        j exit                # go to Exit
Else:   sub  $s0, $s1, $s2    # f = g – h (skipped if i = j)
exit:
```

Loop:

```
Loop:   sll $t1,$s3,2       # Temp reg $t1 = 4 * i
        add $t1,$t1,$s6     # $t1 = address of save[i]
        lw $t0,0($t1)       # Temp reg $t0 = save[i]
        bne $t0,$s5, Exit   # go to Exit if save[i] != k
        addi $s3,$s3,1      # i = i + 1
        j Loop              # go to Loop
Exit:
```

*** Chuyển đổi và bắt dầu một chương trình:** Gồm 4 bước



2.8 Thủ tục (Procedure) cho assembly MIPS

- Nó là một hàm, làm dễ hiểu hơn, tái sử dụng mã nguồn

*** Các công đoạn của thủ tục phải tuân theo sáu bước sau:**

1. Đặt các tham số ở một nơi mà thủ tục có thể truy xuất được.
2. Chuyển quyền điều khiển cho thủ tục.
3. Yêu cầu tài nguyên lưu trữ cần thiết cho thủ tục đó.
4. Thực hiện công việc (task).
5. Lưu kết quả ở một nơi mà chương trình có thể truy xuất được.
6. Trả điều khiển về vị trị mà thủ tục được gọi. Vì một thủ tục có thể được gọi từ nhiều vị trí trong một chương trình.

*** Một số quy ước của MIPS với thanh ghi cần lưu ý:**

- Thanh ghi $at ($1), $k0 ($26), $k1 ($27) được dành cho hệ điều hành và assembler; không nên sử dụng trong lúc lập trình thông thường.

- Nếu có hơn 4 tham số truyền vào, các tham số còn lại được đưa vào stack ngoài ($a0-$a3)

- Các thanh ghi $t0-$t9 (9-15, 24, 25) được sử dụng như các **thanh ghi tạm**. Các thanh ghi $s0-$s7 (16-23) được sử dụng như các thanh ghi lưu **giá trị bền vững.**

- $v0-$v1 lưu giá trị trả về của hàm

- Theo quy ước của MIPS, một thủ tục nếu sử dụng bất kỳ thanh ghi loại $s nào sẽ phải lưu lại giá trị của thanh ghi $s đó trước khi thực thi hàm. Và trước khi thoát ra khỏi hàm, các giá trị cũ của các thanh ghi $s này cũng phải được trả về lại. Các thanh ghi $s này được xem như biến cục bộ của hàm.

Thanh ghi loại $t được xem là các thanh ghi tạm, nên việc sử dụng thanh ghi loại $t trong thủ tục không cần lưu lại như các thanh ghi loại $s

→ Do thanh ghi có tốc độ truy xuất nhanh nhất trong máy tính nên các nhà lập trình viên cần sử dụng các thanh ghi một cách hợp lý

*** Các khái niệm và tên gọi:**

- Thanh ghi $ra lưu địa chỉ trả về (return address): là một liên kết tới vùng đang gọi cho phép thủ tục trả về đúng địa chỉ

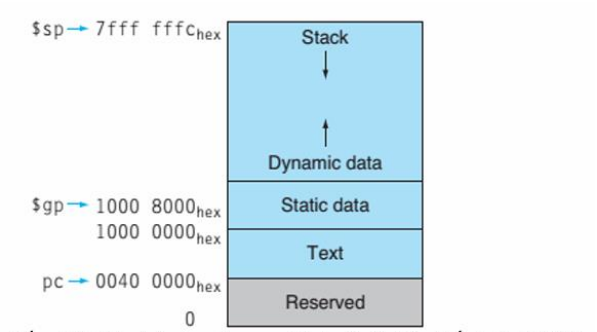
- Caller: Là chương trình gọi một thủ tục và cung cấp những giá trị tham số cần thiết jal <tenham>

- Callee: Là một thủ tục thực thi một chuỗi những lệnh được lưu trữ dựa trên những tham số được cung cấp bởi caller và sau đó trả điều khiều về cho caller

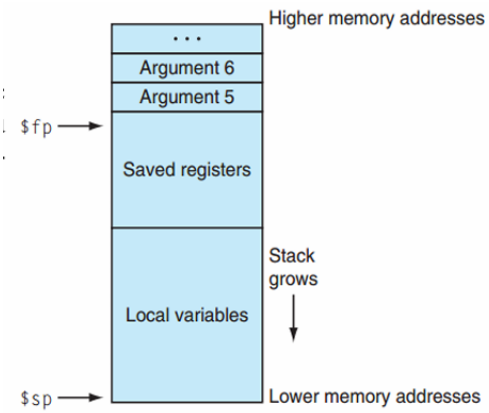
- Program counter (PC: còn được gọi là con trỏ PC hay con trỏ lệnh): Là thanh ghi chứa địa chỉ của lệnh kế tiếp được thực thi trong chương trình. Muốn chỉ lệnh kế tiếp thì PC tăng lên 4 (do là 32 bits = 4 bytes)

- Stack (ngăn xếp): cấu trúc dữ liệu last-in first-out

+ Mỗi thủ tục cần thêm 1 vùng nhớ để lưu các biến cục bộ, các tham số input nếu lớn hơn 4 và tham số trả về nếu lớn hơn 2, ngoài ra nếu thủ tục có sử dụng thanh ghi loại $s. Vùng nhớ này được quy ước như là một stack.



Cấp phát bộ nhớ cho chương trình và dữ liệu kiến trúc MIPS



Hình ảnh ví dụ của một stack với các frame (stack này phục vụ cho thủ tục mà bốn tham số đầu vào đã được truyền thông qua thanh ghi, tham số thứ 5 và 6 được lưu trong stack như hình)

+ Heap: Vùng nhớ cấp phát động, có thể cấp thêm khi tới giới hạn

+ Text segment: Đoạn mã chương trình

+ **Stack pointer** ($sp = 29): Địa chỉ của phần tử được cấp phát gần nhất

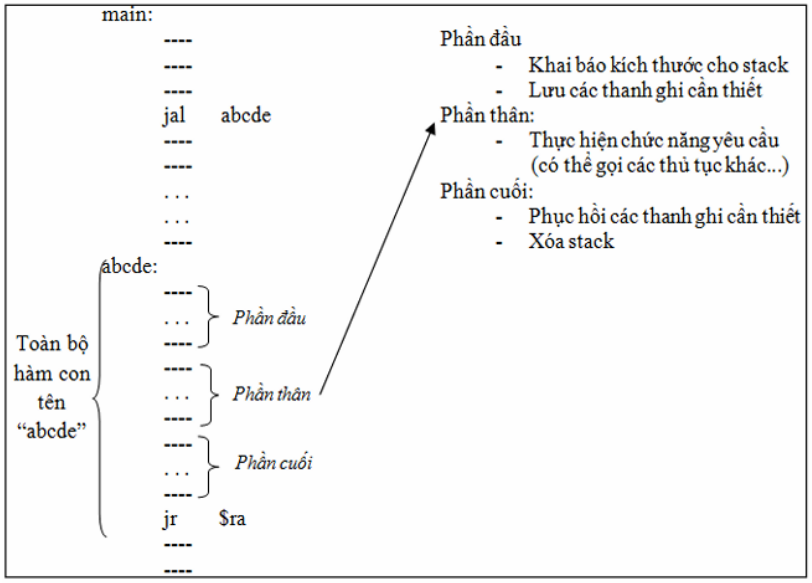
+ **Frame pointer** ($fp = 30): Stack gồm nhiều frame; frame cuối cùng trong stack được trỏ tới bởi $sp, frame đầu tiên được trỏ tới bởi $fp

+ Stack được xây dựng theo kiểu từ địa chỉ **cao giảm dần xuống thấp**, vì thế frame pointer luôn ở trên stack pointer. (theo quy chuẩn của MIPS là như vậy)

+ **Push:** Việc lưu một phần tử vào stack gọi là Push. Theo quy ước của MIPS, khi PUSH một phần tử vào stack, đầu tiên $sp nên trừ đi 4 byte (1 word) rồi sau đó lưu dữ liệu vào word nhớ có địa chỉ đang chứa trong $sp.

+ **Pop:** Việc lấy một phần tử ra khỏi stack gọi là Pop. Ngược lại, khi POP một phần tử từ stack, dữ liệu được load ra từ word nhớ có địa chỉ đang chứa trong $sp rồi sau đó $sp được cộng thêm 4

*** Cấu trúc một thủ tục/ hàm con:**



- Trong chương trình, phần đầu và phần cuối có thể có hoặc không, tùy vào yêu cầu của chương trình con

- Trong chương trình chính, khi thủ tục được gọi, con trỏ PC sẽ chuyển quyền điều khiển xuống vị trí của thủ tục; sau khi thủ tục được thực hiện xong, con trỏ PC sẽ chuyển về thực hiện lệnh ngay sau lệnh gọi thủ tục. Việc này được thực hiện nhờ vào cặp lệnh:

jal ten_thu_tuc

jr $ra

(ten_thu_tuc là nhãn của lệnh đầu tiên trong thủ tục, cũng được xem là tên của thủ tục)

- Để gọi thủ tục, dùng lệnh jal (Lệnh nhảy-và-liên kết: jump-and-link)

+ Đầu tiên, địa chỉ của lệnh ngay sau lệnh jal phải được lưu lại để sau khi thủ tục thực hiện xong, con trỏ PC có thể chuyển về lại đây (thanh ghi dùng để lưu địa chỉ trả về là $ra); sau khi địa chỉ trả về đã được lưu lại, con trỏ PC chuyển đến địa chỉ của lệnh đầu tiên của thủ tục để thực hiện.

- Sau khi thủ tục thực hiện xong, để trả về lại chương trình chính, dùng lệnh jr (jump regiser)

+ Lệnh "jr $ra" đặt ở cuối thủ tục thực hiện việc lấy giá trị đang chứa trong thanh ghi $ra gán vào PC, giúp thanh ghi quay trở về thực hiện lệnh ngay sau lệnh gọi thủ tục này

- Vậy có hai công việc được thực hiện trong jal theo thứ tự:

jal <address>	*# $ra=PC+4*
	# PC = <address>
jr $ra	# PC = $ra

- Nested procedure:

+ Các thủ tục có thể lồng vào nhau: như ví dụ dưới đây

+ Các thủ tục mà không gọi các thủ tục khác là các thủ tục lá (leaf procedures). Ngược lại là nested procedures.

+ Phải cẩn thận khi dùng các thanh ghi trong các thủ tục, càng phải cẩn thận hơn khi gọi một nested procedure.

Ví dụ:

```
or $a0, $zero, $s3
   or $a1, $zero, $s4
   ori $a2, $zero, 3
   ori $a3, $zero, 4
   or $s0, $zero, $v0

   j exit
func:
   addi $sp, $sp, -4
   sw $ra, 0($sp)

   jal tong              # Gọi hàm tong(a, b)
   move $t0, $v0         # Lưu kết quả tong(a, b) vào $t0
   sub $t1, $a2, $a3     # $t1 = c - d

   or $a0, $zero, $t0    # Đưa kết quả tong(a, b) vào $a0
   or $a1, $zero, $t1    # Đưa kết quả c - d vào $a1
   jal tong              # Gọi hàm tong(tong(a, b), c - d)

   addi $sp, $sp, 4
   lw $ra, 0($sp)
   jr $ra                # Quay lại nơi gọi hàm

# Hàm tong(int a, int b)
tong:
   add $v0, $a0, $a1     # Thực hiện phép cộng a + b
   jr $ra                # Quay lại nơi gọi hàm

exit:
```

Chương 3: Phép toán số học trên máy tính

3.1. Phép cộng & Phép trừ

- Đã được đề cập ở chương 1

- Xét overflow:

+ Nếu là add, addi, sub là có lệnh có xét tràn, có thể làm chương trình bị dừng, đó là một ngoại lệ (exception)

+ Nếu là addu, addiu, subu không gây ra ngoại lệ tràn

- Khi một chương trình đang thực thi, nếu bị tác động đột ngột (lỗi hoặc phải thi hành một tác vụ khác, …), buộc phải dừng luồng chương trình đang chạy này và gọi

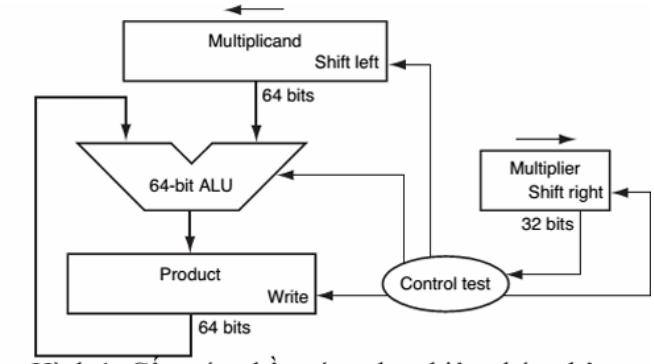
đến một chương trình không định thời trước đó thì được gọi là một **"interrupt"** hay một **"exception"** (Trong một số hệ thống máy tính, thuật ngữ 'interrupt'được sử dụng như exception, nhưng ở một số hệ thống thì có sự phân biệt hai thuật ngữ này)

3.2. Phép nhân

- Giải thuật thực hiện phép nhân theo cấu trúc phần cứng 3 thanh ghi

- 3 bước này được lặp lại 32 lần. Nếu mỗi bước tốn 1 chu kì thì yêu cầu gần 96 chu kì cho phép toán nhân hai số 32 bits

- Số vòng lặp bằng số bit biểu diễn



Hình 1: Cấu trúc phần cứng thực hiện phép nhân

Lưu đồ giải thuật đi kèm cho cấu trúc phần cứng

VD: 4 x 5

Phổ thông

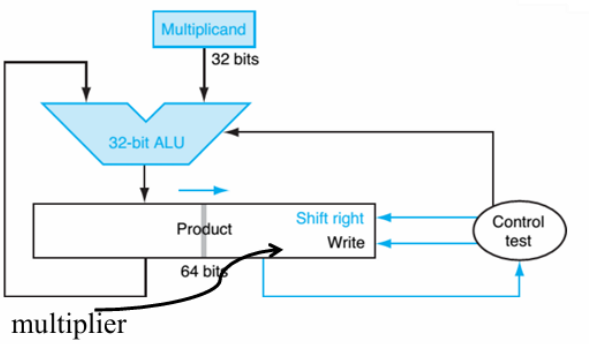
Lần lặp	Bước	Multiplier	Multiplicand	Product
0	Initial values	0101	0000 0100	0000 0000
1	1a: 1 => Prod = Prod + Mcand	0101	0000 0100	0000 0100
	2: Shift left Multiplicand	0101	0000 1000	0000 0100
	3: Shift right Multiplier	0010	0000 1000	0000 0100
2	1a: 0 => No operation	0010	0000 1000	0000 0100
	2: Shift left Multiplicand	0010	0001 0000	0000 0100
	3: Shift right Multiplier	0001	0001 0000	0000 0100
3	1a: 1 => Prod = Prod + Mcand	0001	0001 0000	0001 0100
	2: Shift left Multiplicand	0001	0010 0000	0001 0100
	3: Shift right Multiplier	0000	0010 0000	0001 0100
4	1a: 0 => No operation	0000	0010 0000	0001 0100
	2: Shift left Multiplicand	0000	0100 0000	0001 0100
	3: Shift right Multiplier	0000	0100 0000	0001 0100

Cải tiến

So với giải thuật trước đó thì thanh ghi số bị nhân, bộ ALU, thanh ghi số nhân tất cả đều 32 bits, chỉ có thanh ghi tích là khác – 64 bits

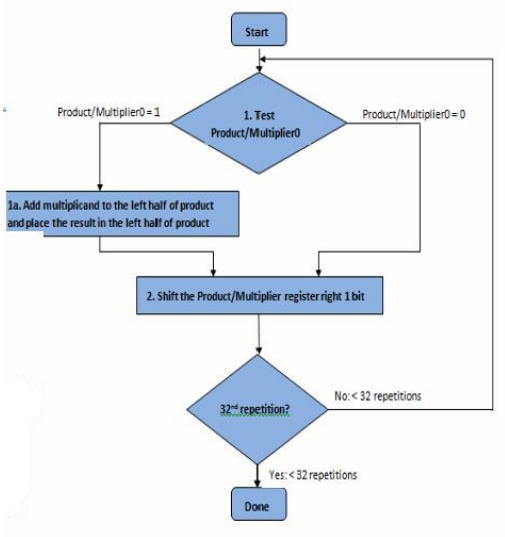
Trong mỗi vòng lặp, số chu kỳ xung clock tiêu tốn có thể giảm xuống chỉ còn 1 chu kỳ

Thanh ghi product 64 bit (khi khởi tạo, đưa multiplier vào 32 bit thấp của product, còn nữa cao khởi tạo 0)



Cấu trúc phần cứng của phép nhân có cải tiến

Lần lặp	Bước	Multiplicand	Product/Multiplier
0	Initial values	0100	0000 0101
1	1: $M_0 = [P/M]_0 = 1$	0100	0000 0101
	1a): Prod = Prod + Mcand	0100	0100 0101
	2: Shift right [P/M]	0100	0010 0010
2	1: $M_0 = [P/M]_0 = 0$	0100	0010 0010
	=> No operation	0100	0010 0010
	2: Shift right [P/M]	0100	0001 0001
3	1: $M_0 = [P/M]_0 = 1$	0100	0001 0001
	1a): Prod = Prod + Mcand	0100	0101 0001
	2: Shift right [P/M]	0100	0010 1000
4	1: $M_0 = [P/M]_0 = 0$	0100	0010 1000
	=> No operation	0100	0010 1000
	2: Shift right [P/M]	0100	0001 0100



*** Dấu của phép nhân:**

- Nhân độ lớn hai số trước, sau đó xét dấu: Nếu hai số trái dấu thì là âm, không thì dương (có thể dùng phép XOR để xét dấu)

*** Phép nhân trong MIPS**

- MIPS sử dụng 2 thanh ghi đặc biệt 32 bit là ***Hi và Lo*** để chứa 64 bit kết quả của phép nhân

- Để lấy giá trị từ thanh ghi ***Hi*** và ***Lo*** ra một thanh ghi khác, sử dụng hai lệnh dành riêng là ***mfhi*** và ***mflo***

- Nhân hai số không dấu, MIPS cung cấp lệnh ***multu***. Nhân hai số có dấu, MIPS cung cấp lệnh ***mult***

3.3. Phép chia

- Ngược lại của phép nhân là phép chia

- Trường hợp ngoại lệ - chia 0

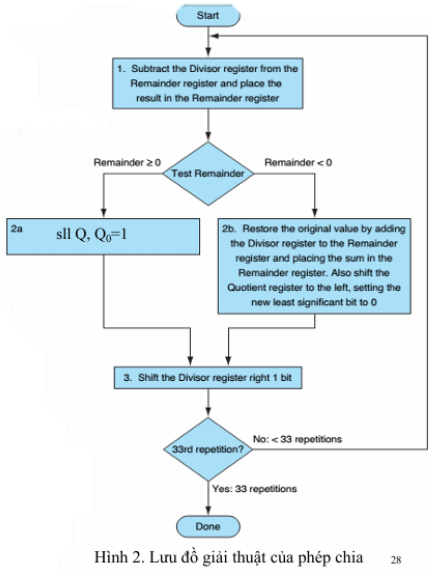
$$\begin{array}{l|l} \underline{a} & \underline{b} \\ r & c \end{array}$$

a: Dividend (Số bị chia)

b: Divisor (Số chia)

c: Quotient (Thương số)

r: Remainder (Số dư)

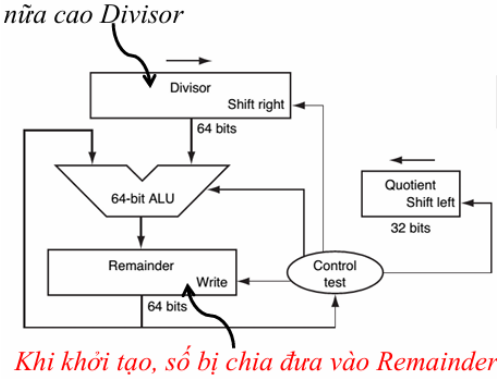


Hình 2. Lưu đồ giải thuật của phép chia 28

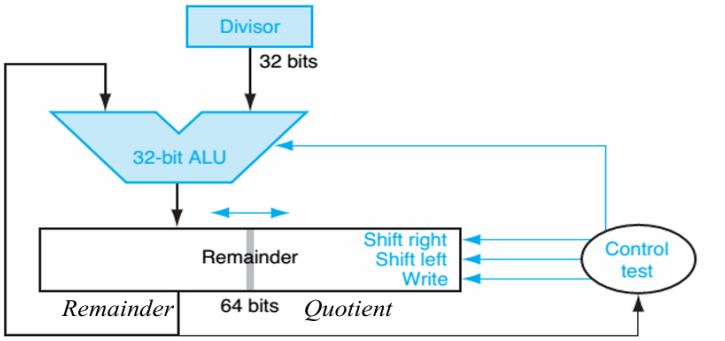
- Giải thuật thực hiện phép chia trên phần cứng

Chú ý: Hai số chia và bị chia là số dương, do đó kết quả thương và số dư là không âm. Thực hiện phép toán trên số dương, do đó, thương và các toán hạng của phép chia có giá trị là 32 bit, bỏ qua các số có dấu.

Khi khởi tạo, số chia đưa vào nữa cao Divisor



Khi khởi tạo, số bị chia đưa vào Remainder



Cải tiến

Ví dụ: Chia 1011/0010

Lần lặp	Bước	Quotient	Divisor	Remainder
0	Initial values	0000	0010 0000	0000 1011
1	1: Rem = Rem - Div	0000	0010 0000	1110 1011
	2b: Rem < 0 => +Div , sll Q, Q_0 = 0	0000	0010 0000	0000 1011
	3: Shift right Div	0000	0001 0000	0000 1011
2	1: Rem = Rem - Div	0000	0001 0000	1111 1011
	2b: Rem < 0 => +Div , sll Q, Q_0 = 0	0000	0001 0000	0000 1011
	3: Shift right Div	0000	0000 1000	0000 1011
3	1: Rem = Rem - Div	0000	0000 1000	0000 0011
	2a: Rem ≥ 0 => sll Q, Q_0 = 1	0001	0000 1000	0000 0011
	3: Shift right Div	0001	0000 0100	0000 0011
4	1: Rem = Rem - Div	0001	0000 0100	1111 1111
	2b: Rem < 0 => +Div , sll Q, Q_0 = 0	0010	0000 0100	0000 0011
	3: Shift right Div	0010	0000 0010	0000 0011
5	1: Rem = Rem - Div	0010	0000 0010	0000 0001
	2a: Rem ≥ 0 => sll Q, Q_0 = 1	0101	0000 0010	0000 0001
	3: Shift right Div	0101	0000 0001	0000 0001

Cải tiến

Lần lặp	Bước	Divisor	R/Q
0	Initial values	0010	0000 1011
1	1: Sll [R/Q]	0010	0001 0110
	Rem = Rem - Div	0010	1111 0110
	2b: Rem < 0 => +Div	0010	0001 0110
2	1: Sll [R/Q]	0010	0010 1100
	Rem = Rem - Div	0010	0000 1100
	2a: Rem ≥ 0 => Q_0 = 1	0010	0000 1101
3	1: Sll [R/Q]	0010	0001 1010
	Rem = Rem - Div	0010	1111 1010
	2b: Rem < 0 => +Div	0010	0001 1010
4	1: Sll [R/Q]	0010	0011 0100
	Rem = Rem - Div	0010	0001 0100
	2a: Rem ≥ 0 => Q_0 = 1	0010	0001 0101

*** Phép chia có dấu:**

- Bước 1: Bỏ qua dấu, thực hiện phép chia như thông thường

- Bước 2: Xét dấu

+ Dùng phép XOR để xét dấu của Quotient

+ Dấu của Remainder cùng dấu với Dividend, hoặc dùng công thức

Số bị chia = Thương × Số chia + Số dư

→ Số dư= Số bị chia – (Thương × Số chia)

*** Phép chia trong MIPS:**

- Trong cấu trúc phần cứng cho phép nhân có cải tiến, hai thanh ghi ***Hi*** và ***Lo*** được ghép lại để hoạt động như thanh ghi 64 bit của Product/Multiplier (tương tự như phép nhân)

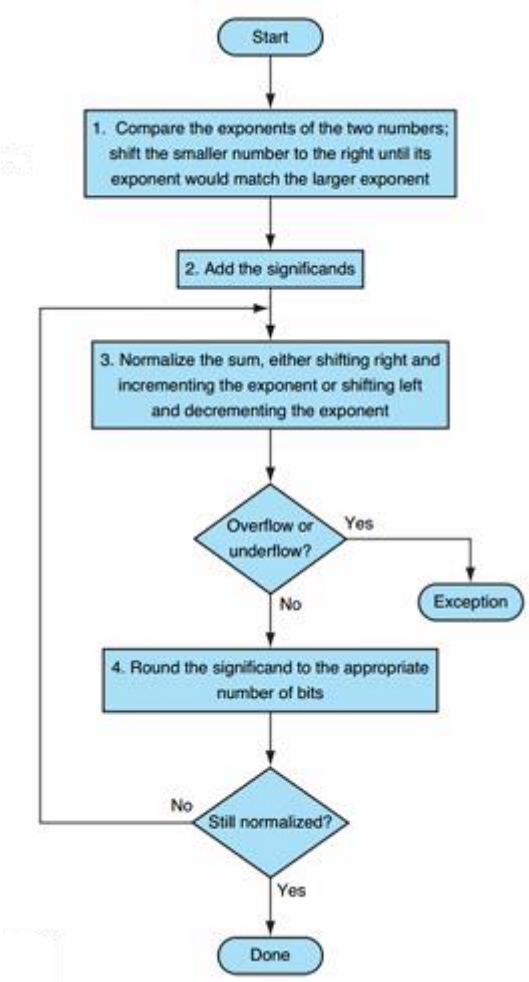
- Sau khi thực hiện phép chia:

+ ***Hi*** chứa phần dư + ***Lo*** chứa thương số

- Để xử lý cho các số có dấu và số không dấu, MIPS có 2 lệnh: phép chia có dấu (**div**), và phép chia không dấu (**divu**).

3.4. Số chấm động

*** Phép toán cộng trên số thực dấu chấm động:**



Bước 1: So sánh mũ (tăng mũ nhỏ bằng mũ lớn)

Bước 2: Cộng phần trị

Bước 3: Chuẩn hóa, xét tràn số mũ

Bước 4: Làm tròn tổng, kiểm tra lại

Ví dụ: 0,5 + -(0.4375)

$0.5 = 0.1_2 = 1.000 \times 2^{-1}$

$-0.4375 = -0.0111_2 = -1.110 \times 2^{-2}$

1) So sánh mũ (nhỏ ➞ lớn)

$-1.110 \times 2^{-2} = -0.111 \times 2^{-1}$

2) Cộng phần trị

$1.000 \times 2^{-1} + (-0.111 \times 2^{-1}) = (1.000 - 0.111) \times 2^{-1} = 0.001 \times 2^{-1}$

*** Lưu ý:**

+ Nếu số lớn trừ số nhỏ → giữ nguyên

+ Nếu số nhỏ trừ số lớn → - (số lớn – số nhỏ) → để tránh bù hai.

VD: 3 – 5 → - (5-3)

3) Chuẩn hóa tổng, xét tràn số mũ

$0.001 \times 2^{-1} = 1.000 \times 2^{-4}$

$y = -4 \Rightarrow E = y + 127 = -4 + 127 = 123 \in [1; 254]$ (Không tràn số mũ)

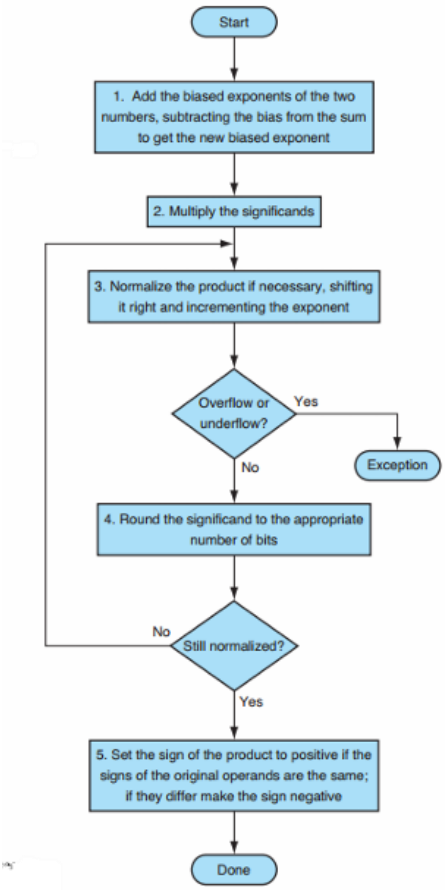
4) Làm tròn tổng

1.000×2^{-4} (không đổi)

Kiểm tra lại

$1.000 \times 2^{-4} = 0.0001 = 0.0625_{10}$ (0.5 + (-0.4375)

Phép nhân trên dấu chấm động



Bước 1: Cộng số mũ

Bước 2: Nhân phần trị

Bước 3: Chuẩn hóa, xét tràn số mũ

Bước 4: Làm tròn

Bước 5: Xét dấu

Ví dụ: 0.5 x (-0.4375)

$0.5 = 0.1_2 = 1.000 \times 2^{-1}$

$-0.4375 = -0.0111_2 = -1.110 \times 2^{-2}$

1) Cộng phần mũ

$-1 + (-2) = -3$

$E = y + 127 = -3 + 127 = 124$

2) Nhân phần trị

$1.000 \times 1.110 = 1.110000$

$\Rightarrow 1.110000 \times 2^{-3}$ $\Rightarrow 1.110 \times 2^{-3}$

3) Chuẩn hóa tích, xét tràn số mũ

1.110×2^{-3} (không đổi)

$y = -3 \rightarrow E = y + 127 = -3 + 127 = 124 \in [1; 254]$

(Không tràn số mũ)

4) Làm tròn tích: 1.110×2^{-3} (không đổi)

5) Xét dấu: Hai thừa số trái dấu ➞ tích âm

Tích = -1.110×2^{-3}

Kiểm tra lại

$-1.110 \times 2^{-3} = -0.001110 = -0.21875_{10}$

Chương 4: Bộ xử lý processor

4.1. Giới thiệu

*** Hiệu suất của máy tính được xác định bởi 3 yếu tố:**

- **Tổng số câu lệnh:** Được xác định bởi trình biên dịch và kiến trúc tập lệnh.

- **Chu kỳ xung clock:** Xác định tốc độ xử lý của máy tính.

- **Số chu kỳ xung clock trên mỗi lệnh (CPI):** Được xác định bởi quá trình hiện thực bộ xử lý.

*** Mục đích của chương này:**

- Giải thích quy tắc hoạt động và hướng dẫn xây dựng datapath cho một bộ xử lý chứa một số lệnh đơn giản (giống kiến trúc tập lệnh dạng MIPS), gồm hai ý chính:

+ Thiết kế datapath
+ Hiện thực datapath đã thiết kế

MIPS (bắt nguồn từ chữ viết tắt của 'Microprocessor without Interlocked Pipeline Stages') là một kiến trúc tập tập lệnh dạng RISC, được phát triển bởi MIPS Technologies (trước đây là MIPS Computer Systems, Inc.)

* Chương này chỉ xem xét 8 lệnh trong 3 nhóm chính của tập lệnh MIPS:

- Nhóm lệnh tham khảo bộ nhớ (**lw** và **sw**)

- Nhóm lệnh liên quan đến logic và số học (**add, sub, AND, OR, và slt**)

- Nhóm lệnh nhảy (Lệnh nhảy với điều kiện bằng **beq**)

* Tổng quan các lệnh cần xem xét:

Nhóm lệnh tham khảo bộ nhớ: Nạp lệnh → Đọc một/hai thanh ghi → Sử dụng ALU → Truy xuất bộ nhớ để đọc/ghi dữ liệu

Nhóm lệnh logic và số học: Nạp lệnh → Đọc một/hai thanh ghi → Sử dụng ALU → Ghi dữ liệu vào thanh ghi

Nhóm lệnh nhảy: Nạp lệnh → Đọc một/hai thanh ghi → Sử dụng ALU → Chuyển đến địa chỉ lệnh tiếp theo dựa trên kết quả so sánh

4.2. Nhắc lại các quy ước thiết kế logic

*** Các khái niệm:**

- Mạch tổ hợp (Combinational): ALU

- Mạch tuần tự (Sequential): instruction/data memories và thanh ghi

- Tín hiệu điều khiển (Control signal)

- Tín hiệu dữ liệu (Data signal)

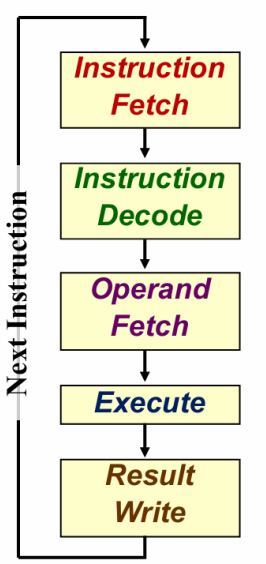
+ Asserted (assert): Khi tín hiệu ở mức cao hoặc 'true'

+ Deasserted (deassert): Khi tín hiệu ở mức thấp hoặc 'false'

+ Edge-triggered clocking (Rising/Falling)

4.3. Xây dựng đường dữ liệu (datapath) đơn giản

*** Quy trình thực thi lệnh:**



- Instruction Fetch (tìm nạp lệnh):
+ Nạp lệnh từ bộ nhớ (memory)

+ Địa chỉ của lệnh lưu trong thanh ghi Program Counter (PC)

- Instruction Decode (giải mã lệnh): Tìm ra lệnh thực hiện

- Operand Fetch (tìm nạp toán hạng): Lấy các toán hạng cần thiết cho lệnh

- Execute (thực thi): Thực hiện câu lệnh

- Result Write (lưu trữ): Lưu trữ kết quả

Bảng sau mô tả ba giai đoạn thực thi lệnh trong ba nhóm lệnh cơ bản của MIPS (Giai đoạn *Fetch* and *Decode* không được hiển thị)

	add $3, $1, $2	lw $3, 20($1)	beq $1, $2, label
Fetch & Decode	*standard*	*standard*	*standard*
Operand Fetch	- Đọc thanh ghi $1, xem như toán hạng opr1 - Đọc thanh ghi $2, xem như toán hạng opr2	- Đọc thanh ghi $1, xem như toán hạng opr1 - Sử dụng 20 như toán hạng opr2	- Đọc thanh ghi $1, xem như toán hạng opr1 - Đọc thanh ghi $2, xem như toán hạng opr2
Execute	Result = opr1+ opr2	MemAddr = opr1 + opr2 Sử dụng MemAddr để đọc dữ liệu từ bộ nhớ	Taken = (opr1== opr2)? Target = PC+ Label*
Result Write	Result được lưu trữ vào $3	Dữ liệu của từ nhớ có địa chỉ MemAddr được lưu trữ vào $3	if (Taken) PC= Target

Opr = Operand

MemAddr = Memory Address

* = simplification, not exact

*** Quy trình thực thi lệnh của MIPS (5 công đoạn):**

- Thay đổi thiết kế các giai đoạn thực hiện lệnh:

+ Gộp giai đoạn Decode và Operand Fetch –Giai đoạn Decode của MIPS khá đơn giản

+ Tách giai đoạn Execute thành ALU (Calculation) và Memory Access

	add $3, $1, $2	lw $3, 20($1)	beq $1, $2, label
Fetch	Đọc lệnh (địa chỉ của lệnh lưu trong thanh ghi PC)	Đọc lệnh (địa chỉ của lệnh lưu trong thanh ghi PC)	Đọc lệnh (địa chỉ của lệnh lưu trong thanh ghi PC)
Decode & Operand Fetch	- Đọc thanh ghi $1, xem như toán hạng opr1 - Đọc thanh ghi $2, xem như toán hạng opr2	- Đọc thanh ghi $1, xem như toán hạng opr1 - Sử dụng 20 như toán hạng opr2	- Đọc thanh ghi $1, xem như toán hạng opr1 - Đọc thanh ghi $2, xem như toán hạng opr2
ALU	*Result = opr1+ opr2*	MemAddr = opr1 + opr2	Taken = (opr1== opr2)? Target = PC+ Label*
Execute		Sử dụng *MemAddr* để đọc dữ liệu từ bộ nhớ	
Result Write	*Result* được lưu trữ vào $3	Dữ liệu của từ nhớ có địa chỉ *MemAddr* được lưu trữ vào $3	if (*Taken*) PC= *Target*

*** Các bước thực thi theo sơ đồ:**

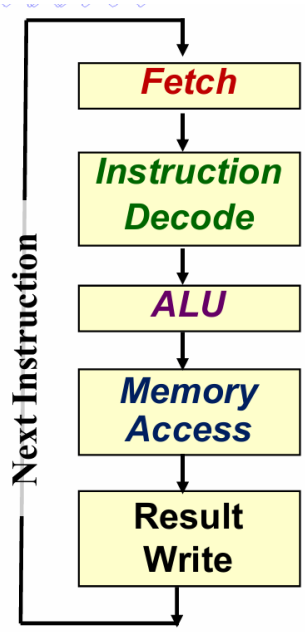
- Instruction Fetch (Nạp lệnh)

- Instruction Decode & Operand Fetch (Giải mã và lấy các toán hạng cần thiết, gọi tắt là "Instruction Decode")

- ALU (Giai đoạn sử dụngALU hay giai đoạn thực thi)

- Memory Access (Giai đoạn truy xuất vùng nhớ)

- Result Write (Giai đoạn ghi lại kết quả/lưu trữ)



4.3.1. Giai đoạn tìm nạp lệnh (Instruction Fetch)

* Sử dụng thanh ghi Program Counter (PC) để tìm nạp lệnh từ **bộ nhớ**

- Thanh ghi PC là một thanh ghi đặt biệt trong bộ vi xử lý

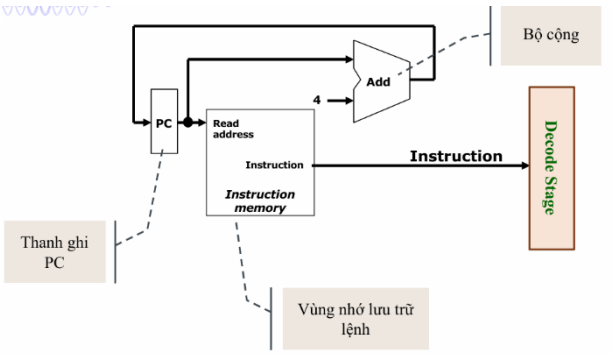
* **Tăng giá trị** trong thanh ghi PC lên 4 đơn vị để lấy địa chỉ của lệnh tiếp theo

- Lệnh tiếp theo là PC + 4

- Chú ý, lệnh rẽ nhánh (branch) và lệnh nhảy (jump) là một trường hợp ngoại lệ

=> Kết quả của giai đoạn này là đầu vào cho giai đoạn tiếp theo (Decode): Kết quả của giai đoạn này là 32 bit mã máy của lệnh cần thực thi. Chuỗi 32 bits này sẽ sử dụng như đầu vào (input) cho giai đoạn tiếp theo là Decode

*** Khối instruction memory:**

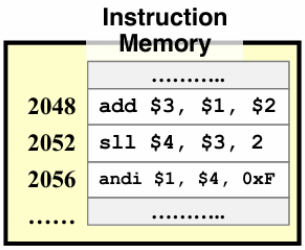


- Vùng nhớ lưu trữ lệnh

- Đầu vào: địa chỉ của lệnh

- Đầu ra: là nội dung lệnh tương ứng với địa chỉ được cấp

- Địa chỉ của từng lệnh phải là bội chung của 4 (sắp xếp như hình)

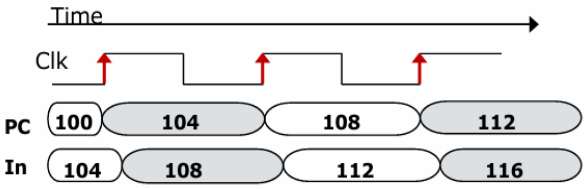


*** Bộ cộng:** Mạch logic kết hợp để cộng 2 số

- Đầu vào: Hai số 32-bit, A, B

- Đầu ra: A + B

*** Thanh ghi PC** sau khi kết thúc chu kì liền thay đổi liền. In sẽ + 4 ngay sau đó và chờ hết chu kì để đưa vào thanh ghi PC



4.3.2. Giải mã và lấy các toán hạng cần thiết (Instruction Decode & Operand Fetch)

- Giai đoạn decode: Lấy nội dung dữ liệu trong các trường (field) của lệnh:

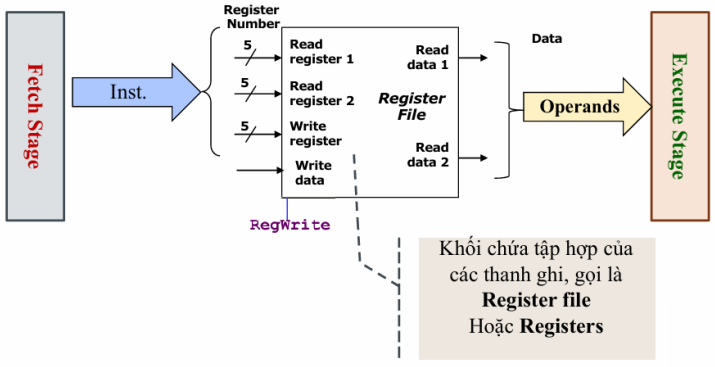
+ Đọc opcode để xác định kiểu lệnh và chiều dài của từng trường trong mã máy

+ Đọc dữ liệu từ các thanh ghi cần thiết (có thể 2 (add), 1 (addi), hoặc 0 (j))

- Đầu vào từ giai đoạn trước (Fetch): Lệnh cần được thực thi

- Đầu ra cho giai đoạn tiếp theo (Execute): Phép tính và các toán hạng cần thiết

* Khối Resgister:



- Một tập 32 thanh ghi:

+ Mỗi thanh ghi có chiều dài 32 bit và có thể được đọc hoặc ghi bằng cách chỉ ra chỉ số của thanh ghi

+ Với mỗi lệnh được đọc nhiều nhất 2 thanh ghi và ghi nhiều nhất 1 thanh ghi.

- RegWrite: là một tín hiệu điều khiển nhằm mục đích:

+ Cho phép ghi vào một thanh ghi hay không (1 True (Write), 0 False (No))

* Chú ý: Đối với lệnh I – type, cần phải có sử dụng một Multiplexer để lựa chọn chỉ số thanh ghi cho ngõ *write resgister* chính sát dựa trên từng loại lệnh.

- Ngoài ra cần thêm một MUX nữa cho đầu ra (Ngõ Data2) để chọn Output thứ 2 là

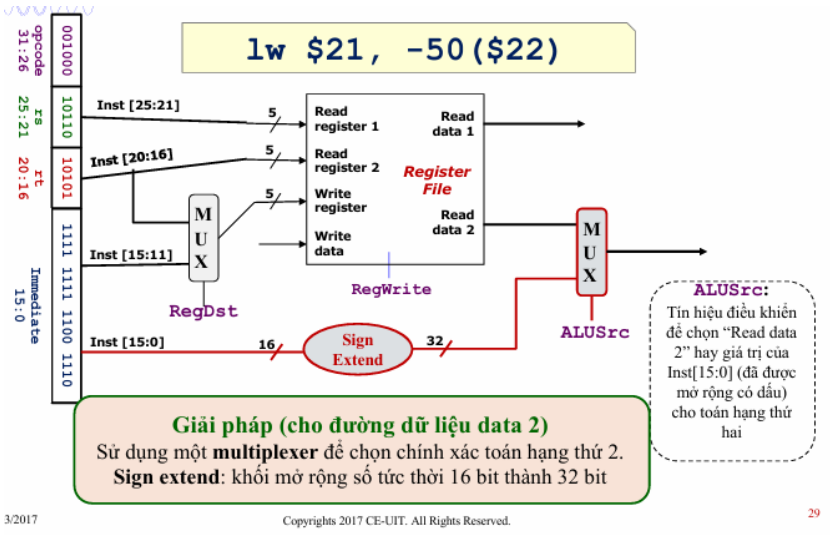
* Khối Multiplexer (MUX)

- Chức năng: Chọn một input từ tập input đầu vào

- Inputs: n đường vào có cùng chiều rộng

- Control: Cần m bit trong đó $n = 2^m$

- Output: Chọn đường output thứ i nếu giá trị tín hiệu điều khiển control = i



* Khối sign extend: Mở rộng số tức thời 16 bit thành 32 bit

4.3.3. Giai đoạn sử dụng ALU hay giai đoạn thực thi (ALU)

- Công đoạn ALU:

+ ALU = Arithmetric – Logic Unit

+ Công việc thực thật sự của hầu hết các lệnh được hiện chủ yếu trong giai đoạn này

~ Số học (add, sub), Logic (and, or): ALU tính ra kết quả cuối cùng

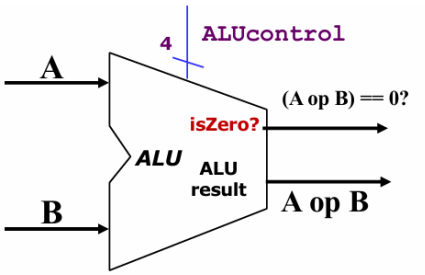
~ Lệnh làm việc với bộ nhớ (lw, sw): ALU dùng tính toán địa chỉ của bộ nhớ

~ Lệnh nhảy/ nhánh (bne, beq): ALU thực hiện so sánh các giá trị trên thanh ghi và tính toán địa chỉ đích sẽ nhảy tới

- Đầu vào từ công đoạn trước (Decode): Các thao tác (operation) và toán hạng (operand(s))

- Đầu ra cho công đoạn tiếp theo (Memory): Tính toán kết quả (Đối với lệnh lw và sw: Kết quả của công đoạn này sẽ là địa chỉ cung cấp cho memory để lấy dữ liệu)

*** Khối ALU:**



- ALU: Dùng để thực hiện các phép tính logic và số học

- Inputs: 2 số 32-bit

- Điều khiển khối ALU: Do ALU có thể thực hiện nhiều chức năng → dùng 4-bit để quyết định chức năng/phép tính cụ thể nào cho ALU

- Outputs: + Kết quả của phép toán số học hoặc logic

+ Một bit tín hiệu chỉ ra rằng kết quả có bằng 0 hay không (bne, beq)

*** Chú ý:** Các lệnh có rẽ nhánh cần thêm 1 phép toán nữa để tính địa chỉ mới sau khi nhảy

- Sử dụng ALU thứ nhất để so sánh thanh ghi

Instruction opcode	ALUop	ALU control input
load word	00	0010
Store word	00	0010
Branch equal	01	0110
add	10	0010
subtract	10	0110
AND	10	0000
OR	10	0001
slt	10	0111

- Sử dụng ALU thứ hai để tính địa chỉ nếu mà có nhảy (Cần offset)

4.3.4. Giai đoạn truy xuất vùng nhớ (Memory Access)

- Chỉ có lệnh Load và Store cần thực hiện các thao tác trong giai đoạn này:

  + Sử dụng địa chỉ vùng nhớ được tính toán ở giai đoạn ALU

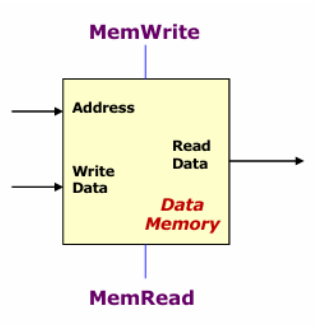
  + Đọc dữ liệu ra hoặc ghi dữ liệu vào vùng nhớ dữ liệu

- Tất cả các lệnh khác sẽ rảnh trong thời gian này

*** Đầu vào từ giai đoạn trước (ALU):** Kết quả tính toán được dùng làm địa chỉ vùng nhớ (nếu có thể ứng dụng)

*** Đầu ra cho giai đoạn tiếp theo (Result Write):** Kết quả được lưu trữ lại (nếu cần)

*** Khối data memory:**



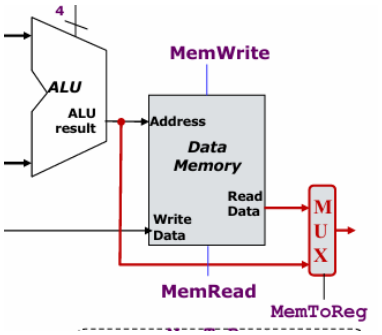
- Vùng nhớ này lưu trữ dữ liệu cần thiết của chương trình

- Inputs: + Address: Địa chỉ vùng nhớ

+ Write Data: Dữ liệu sẽ được ghi vào vùng nhớ đối với lệnh Store.

- Tín hiệu điều khiển: Tín hiệu đọc (MemRead) và ghi (MemWrite); chỉ một tín hiệu được bật lên tại bất kì một thời điểm nào.

- Output: Dữ liệu được đọc từ vùng nhớ đối với lệnh load, lệnh store **không** có output



* Chú ý: Các lệnh không sử dụng Mem cần sử dụng thêm một MUX để chọn kết quả lưu trữ vào thanh ghi.

- MemToReg: Tín hiệu điều khiển giúp lựa chọn giá trị lưu vào thanh ghi là từ Read Data hay từ ALU Result

4.3.5. Giai đoạn ghi lại kết quả/ lưu trữ (Result Write)

*** Công đoạn Result Write:**

- Những lệnh ghi kết quả của các phép toán vào thanh ghi:

  + Ví dụ: số học, logic, shifts, load, set-less-than

  + Cần chỉ số thanh ghi đích và kết quả tính toán

- Những lệnh không ghi kết quả như: store, branch, jump: Không có ghi kết quả

→ Những lệnh này sẽ rảnh trong giai đoạn này

* Đầu vào từ giai đoạn trước (Memory): Kết quả tính toán hoặc từ Memory hoặc là từ ALU

* Chỉ đơn giản là đưa kết quả vào thanh ghi (ngõ Write data của khối Registers/Register file). Chỉ số của thanh ghi được ghi vào (ngõ vào ***Write Register***) được sinh ra trong giai đoạn **Decode Stage**

4.4. Hiện thực datapath đơn chu kì

(tài liệu tổng hợp đi kèm)

*** Hiện thực bộ xử lý đơn chu kỳ** (Single-cycle implementation hay single clock cycle implementation): là cách hiện thực sao cho bộ xử lý đáp ứng thực thi mỗi câu lệnh chỉ trong 1 chu kỳ xung clock → đòi hỏi chu kỳ xung clock phải bằng thời gian của lệnh dài nhất.

* Cách hiện thực bộ xử lý như đã trình bày trên là cách hiện thực đơn chu kỳ: Lệnh dài nhất là lw, gồm truy xuất vào “Instruction memory”, “Registers”, “ALU”, “Data memory” và quay trở lại “Registers”, trong khi các lệnh khác không đòi hỏi tất cả các công đoạn trên à chu kỳ xung clock thiết kế phải bằng thời gian thực thi lệnh lw.

* Mặc dù hiện thực bộ xử lý đơn chu kỳ có CPI = 1 nhưng hiệu suất rất kém, vì một chu kỳ xung clock quá dài, các lệnh ngắn đều phải thực thi cùng thời gian với lệnh dài nhất. Vì vậy, **hiện thực đơn chu kỳ hiện tại không còn được sử dụng (hoặc chỉ có thể chấp nhận cho các tập lệnh nhỏ)**

Tín hiệu điều khiển	Tác động khi ở mức thấp	Tác động khi ở mức cao
RegDst	Thanh ghi đích cho thao tác ghi sẽ từ thanh ghi ri (bits 20:16)	Thanh ghi đích cho thao tác ghi sẽ từ thanh ghi rd (bits 15:11)
RegWrite	Khối "Registers" chỉ thực hiện mỗi chức năng đọc thanh ghi	Ngoài chức năng đọc, khối "Registers" sẽ thực hiện thêm chức năng ghi.
ALUSrc	Input thứ hai cho ALU đến từ "Read data 2" của khối "Registers"	Input thứ hai cho ALU đến từ output của khối "Sign-extend"
Branch	Cho biết lệnh nạp vào không phải "beq". Thanh ghi PC nhận giá trị là PC + 4	Lệnh nạp vào là lệnh "beq", kết hợp với điều kiện bằng thông qua cổng AND nhằm xác định xem lệnh tiếp theo có nhảy 2 đến địa chỉ mới hay không.

MemRead	(Không)	Khối "Data register" thực hiện chức năng đọc dữ liệu.
MemWrite	(Không)	Khối "Data register" thực hiện chức năng ghi dữ liệu.
MemtoReg	Giá trị đưa vào ngõ "Write data" đến từ ALU	Giá trị đưa vào ngõ "Write data" đến từ khối "Data memory"

Chương 5: Kỹ thuật ống dẫn (pineline)

5.1. Tổng quan về pipelining (Mục 4.5):

- Định nghĩa pipelining: Pipeline là một kỹ thuật mà trong đó các lệnh được thực thi theo kiểu chồng lấn lên nhau (overlap).

- Ý tưởng cốt lõi: Các phần khác nhau của một lệnh hoặc nhiệm vụ có thể được xử lý đồng thời ở các giai đoạn khác nhau của đường ống.

5.2. Thiết kế datapath và control cho pipelining (Mục 4.6):

- Các lệnh MIPS được chia làm 5 công đoạn:

+ Công đoạn 1: Nạp lệnh từ bộ nhớ **- IF**

+ Công đoạn 2: Giải mã lệnh và đọc các thanh ghi cần thiết (MIPS cho phép đọc và giải mã đồng thời) **- ID**

+ Công đoạn 3: Thực thi các phép tính hoặc tính toán địa chỉ **- EX**

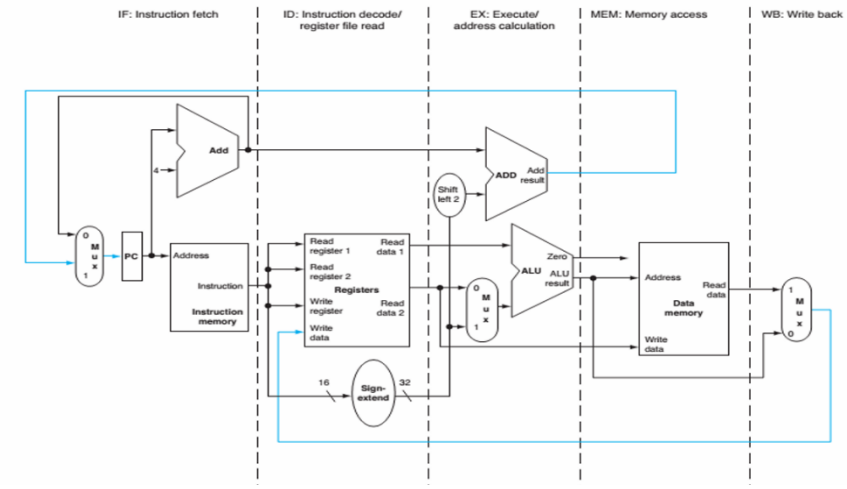
+ Công đoạn 4: Truy xuất các toán hạng trong bộ nhớ **- MEM**

+ Công đoạn 5: Ghi kết quả cuối vào thanh ghi **– WB**

- Xét một bộ xử lý với 8 lênh cơ bản trong chương này.

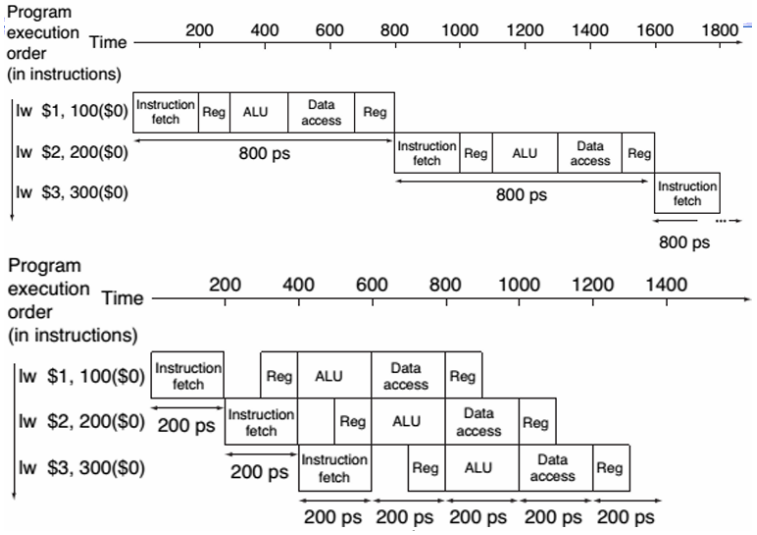
- Giả sử thời gian hoạt động của các công đoạn như trong bảng

- So sánh thời gian trong bình giữa các lệnh



Instruction	IF	ID	EX	MEM	WB	Total
lw	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
sw	200 ps	100 ps	200 ps	200 ps		700 ps
R-Format	200 ps	100 ps	200 ps		100 ps	600 ps
beq	200 ps	100 ps	200 ps			500 ps

- Thời gian giữa lệnh thứ nhất và thứ tư trong không pinline là 3 x 800 = 2400 ps, nhưng trong pinline là 3 x 200 = 600 ps



*** Tăng tốc trong pinline**

- Trong trường hợp lý tưởng: khi mà các công đoạn pineline hoàn toàn bằng nhau thì:

Thời gian giữa hai lệnh liên tiếp pineline = $\frac{\text{thời gian giữa 2 lần liên tiếp không pineline}}{\text{số tầng pineline}}$

Như ví dụ trên thì thời gian giữa 2 lệnh liên tiếp có pineline = (800:5 = 160ps)

Speed up $\approx \frac{\text{thời gian giữa 2 lần liên tiếp không pineline}}{\text{thời gian giữa hai lệnh liên tiếp pineline}}$ tăng tốc $\leq$ số tầng pineline

Speed up = $\frac{n \times k \times t_c}{(n+k-1) \times t_c} = \frac{n.T_{single}}{(k+n-1).T_{pineline}}$ Với k và số tầng, n là số lượng lệnh

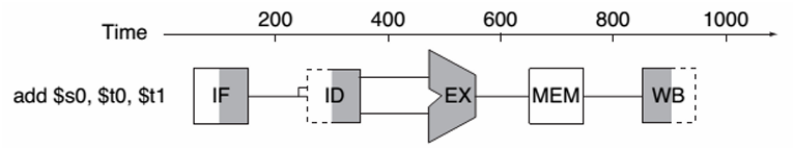
*** Lưu ý:**

- Kỹ thuật pipeline **không** giúp giảm thời gian thực thi của từng lệnh riêng lẻ mà giúp giảm **tổng thời gian thực thi của đoạn lệnh/chương trình chứa nhiều lệnh** (từ đó giúp thời gian trung bình của mỗi lệnh giảm)

- Việc giúp giảm thời gian thực thi cho nhiều lệnh vô cùng quan trọng, vì các chương trình chạy trong thực tế thông thường lên đến **hàng tỉ lệnh**

*** Quy ước trình bày 5 công đoạn thực thi một lệnh của pipeline:**

- Lưu ý cách vẽ hình các công đoạn pipeline như sau:

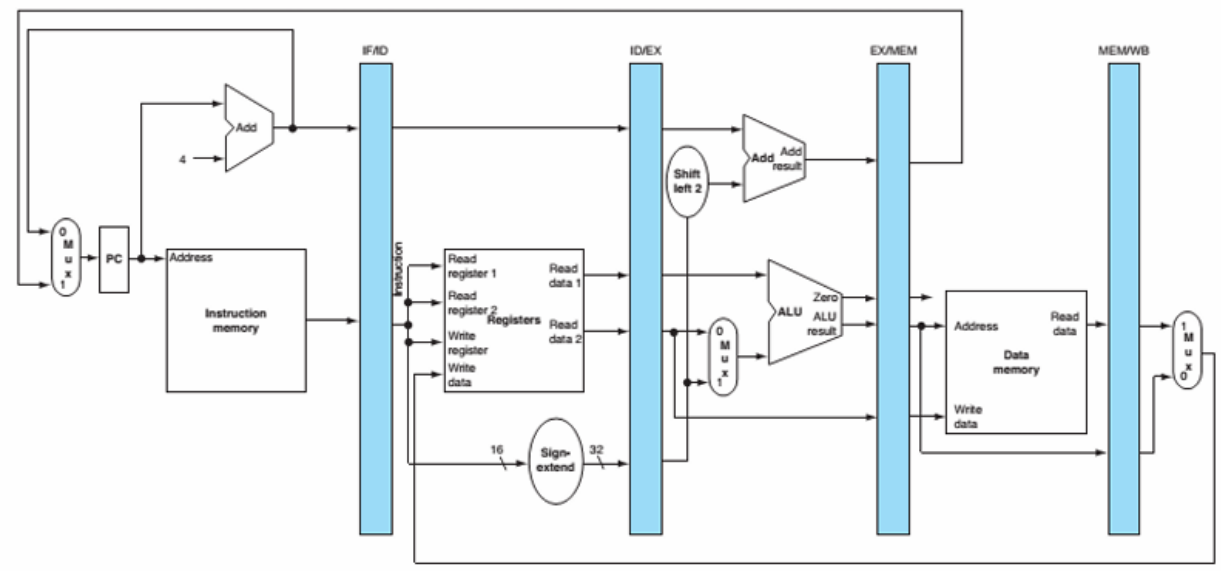


+ Khối **tô đen** hoàn toàn hoặc **để trắng** hoàn toàn: Trong mỗi công đoạn pipeline, nếu lệnh thực thi không làm gì trong công đoạn này sẽ được tô trắng, ngược lại sẽ được tô đen.

Ví dụ lệnh "add" có EX đen và MEM trắng tức lệnh này có tính toán trong công đoạn EX và không truy xuất bộ nhớ dữ liệu trong công đoạn MEM.

+ Các công đoạn liên qua đến bộ nhớ và thanh ghi có thể tô **nữa trái** hoặc **nữa phải** đen: Nếu nữa phải tô đen, tức công đoạn đó đang thực hiện thao tác đọc; ngược lại nếu nữa trái tô đen, công đoạn đó đang thực hiện thao tác ghi.

* Hình ảnh datapath có hỗ trợ pineline:



5.3. Vấn đề về hazard trong pipelining:

5.3.1. Data Hazards (Xung đột dữ liệu):

- Xảy ra khi một lệnh cần sử dụng dữ liệu mà lệnh trước đó chưa xử lý xong hoặc chưa đưa ra kết quả. Là khi một lệnh dự kiến không thể thực thi trong đúng chu kỳ pipeline của nó do dữ liệu mà lệnh này cần vẫn chưa sẵn sàng

- Các loại xung đột dữ liệu:

+ **RAW (Read After Write):** Lệnh sau cần đọc dữ liệu mà lệnh trước đang ghi.

+ **WAR (Write After Read):** Lệnh sau muốn ghi vào vị trí mà lệnh trước đang đọc.

+ **WAW (Write After Write):** Hai lệnh cùng ghi vào một vị trí, có thể gây nhầm lẫn về thứ tự kết quả.

- Cách xử lý:

+ Forwarding/Bypassing: Truyền dữ liệu từ giai đoạn sau đến giai đoạn trước mà không đợi lưu trữ vào bộ nhớ.

+ Stalling: Tạm dừng pipeline bằng cách chèn các "bong bóng" (bubbles) vào đường ống.

* Một số cách giải quyết xung đột của một số lệnh:

- Lệnh add:

+ Nếu không dùng gì cả phải chờ 3 chu kì

+ Chờ 2 chu kì xung clock (nếu đã có kĩ thuật ghi nửa chu kì đầu và đọc cuối chu kì)

+ Dùng kĩ thuật nhìn trước: không cần ngưng

- Lệnh lw:

+ Nếu không dùng gì cả phải chờ 3 chu kì

+ Chờ 2 chu kì xung clock (nếu đã có kĩ thuật ghi nửa chu kì đầu và đọc cuối chu kì)

+ Dùng kĩ thuật nhìn trước: vẫn cần phải chờ 1 chu kì xung clock

5.3.2. Control Hazards (Xung đột điều khiển):

- Xảy ra khi có các lệnh rẽ nhánh hoặc lệnh nhảy (branch/jump), và bộ xử lý chưa biết hướng đi tiếp theo trong pipeline. Là khi một lệnh dự kiến không thể thực thi trong đúng chu kỳ pipeline của nó do lệnh nạp vào không phải là lệnh được cần. Xung đột này xảy ra trong trường hợp luồng thực thi chứa các lệnh nhảy.

Ví dụ: Một lệnh điều kiện (if-else) yêu cầu quyết định luồng điều khiển, nhưng kết quả điều kiện chưa sẵn sàng.

- Cách xử lý:

+ Branch Prediction: Dự đoán hướng đi của nhánh và tiếp tục thực thi lệnh theo dự đoán. Nếu dự đoán sai, hủy bỏ các lệnh sai.

+ Delayed Branch: Thực thi một số lệnh nằm ngay sau nhánh, bất kể nhánh nào được chọn, để giảm thời gian trễ.

+ Flush Pipeline: Xóa các lệnh không hợp lệ khỏi pipeline khi phát hiện dự đoán sai.

- Lệnh beq/bne:

+ Chờ một chu kì clock để chờ điều kiện bằng xảy ra

+ Cải tiến bằng phương pháp dự đoán, chương trình sẽ không lãng phí hoặc lãng phí 1 chu kì xung clock

5.3.3. Structural Hazards (Xung đột cấu trúc):

- Xảy ra khi các phần cứng trong pipeline (như bộ nhớ hoặc ALU) không đủ để hỗ trợ nhiều lệnh đồng thời. Là khi một lệnh dự kiến không thể thực thi trong đúng chu kỳ pipeline của nó do phần cứng cần không thể hỗ trợ. Nói cách khác, xung đột cấu trúc xảy ra khi có hai lệnh cùng truy xuất vào một tài nguyên phần cứng nào đó cùng một lúc.

- Các loại xung đột:

+ Vừa lấy lệnh (IF) vừa truy xuất lấy dữ liệu (MEM)

+ Thanh ghi vừa đọc vừa phải ghi

+ Sử dụng chung một tài nguyên (ALU, ID, IF, MEM)

+ Không thể vừa đọc vừa ghi trong một chu kì

Ví dụ: Hai lệnh cùng cần truy cập bộ nhớ nhưng chỉ có một cổng bộ nhớ.

- Cách xử lý:

+ Resource Duplication: Nhân đôi tài nguyên, ví dụ thêm cổng bộ nhớ hoặc đơn vị tính toán.

+ Stalling: Dừng pipeline để một lệnh hoàn thành trước khi lệnh tiếp theo sử dụng tài nguyên đó.

5.3.4. Exception Hazards (Xung đột do ngoại lệ):

- Xảy ra khi một ngoại lệ (exception) hoặc lỗi xảy ra trong pipeline.

Ví dụ: Lỗi chia cho 0 hoặc truy cập địa chỉ bộ nhớ không hợp lệ.

- Cách xử lý:

+ Precise Interrupts: Đảm bảo các lệnh trước ngoại lệ hoàn tất và các lệnh sau bị hủy.

+ Out-of-Order Execution: Lệnh được xử lý không theo thứ tự nhưng ngoại lệ phải được quản lý theo đúng thứ tự ban đầu.

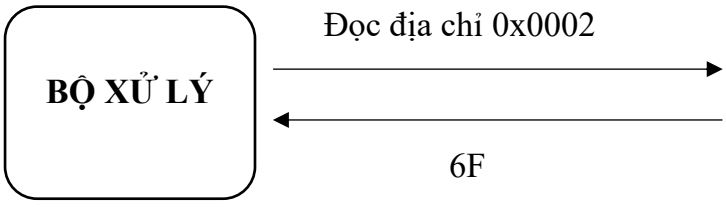
Chương 6: Hệ thống bộ nhớ phân cấp

6.1. Bộ nhớ chính

- Bên cạnh bộ xử lý, bộ nhớ chính (main memory, primary memory) là một thành phần không thể thiếu của một hệ thống máy tính. Trong quá trình chương trình thực thi, bộ nhớ chính lưu giữ chỉ thị của chương trình cùng với các dữ liệu trạng thái liên quan.

- Về mặt cấu trúc, có thể xem bên trong bộ nhớ là các lưới của tế bào nhớ (memory cell). Thông thường, bộ nhớ có thể được truy xuất theo từng byte (byte-addressable), nghĩa là một địa chỉ trong bộ nhớ tham chiếu đến một tập 8 bit dữ liệu. Ví dụ, xét bộ nhớ có kích thước là 64 KB = 64 x 1024 = 65.636 byte. Địa chỉ bộ nhớ sẽ có giá trị từ 0 đến 65.635. Địa chỉ phải được biểu diễu ở dạng nhị phân để có thể sử dụng được trong máy tính số. Với 65.636 giá trị có thể, số lượng bịt cần thiết để biểu diễn địa chi bộ nhớ đang xét sẽ là log2(65.636) = 16. Để cho ngắn gọn, thông thường địa chỉ bộ nhớ sẽ được ghi dạng thập lục phân với mỗi ký tự đại diện cho 4 bit (0x000002, 0x1 = 00012, ..., 0xF = 11112). Như vậy, tập địa chỉ đang xét có thể được ghi ngắn gọn dưới dạng từ 0x0000 đến 0xFFFF.

- Trong thực tế, để hỗ trợ nhiều chương trình cùng truy xuất bộ nhớ chính đồng thời, mỗi chương trình đang thực thi tham chiếu đến một vị trí trong bộ nhớ chính dùng một địa chỉ ảo (virtual address). Địa chỉ ảo được dịch sang địa chỉ vật lý (physical address, machine address) và được chuyển đến bộ điều khiển bộ nhớ (memory controller) để truy xuất dữ liệu. Việc dịch địa chỉ có sự tham gia của hệ điều hành và thành phần phần cứng gọi là đơn vị quản lý bộ nhớ (memory management unit).



Bộ nhớ chính

Địa chỉ bộ nhớ	Dữ liệu
0x0000	00
0x0001	48
0x0002	6F
…	
0xFFFE	65
0xFFFF	00

- Vị trí trong bộ nhớ chính có thể được truy xuất ngẫu nhiên, do đó bộ nhớ chính được gọi là **bộ nhớ truy cập ngẫu nhiên (RAM).** Công nghệ phổ biến được sử dụng trong RAM là SRAM (nhanh, cao) và DRAM (chậm, thấp).

6.2. Phân cấp bộ nhớ

- Theo thời gian, hiệu năng của vi sử lý và bộ nhớ chính khác biệt ngày càng lớn

→ Cần tối ưu dung lượng, tốc độ và giá thành từng loại bộ nhớ

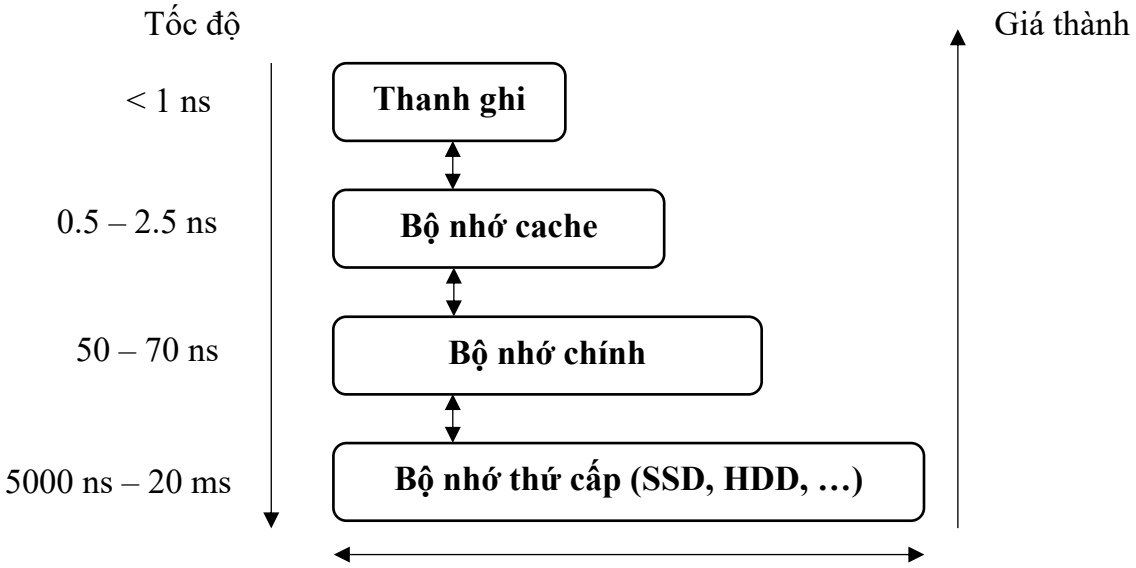
- Đã từng có máy tính có một lượng nhỏ bộ nhớ tốc độ cao SRAM kết hợp với một lượng lớn bộ nhớ DRAM.

*** Tính cục bộ về thời gian và không gian:**

- Tính cục bộ về thời gian: nếu một vị trí bộ nhớ được truy xuất thì nhiều khả năng nó sẽ được truy xuất trong tương lai gần.

- Tính cục bội về không gian: nếu một vị trí bộ nhớ được truy xuất thì các vị trí có địa chỉ gần xung quanh nó nhiều khả năng sẽ được truy xuất tiếp theo. Do các chỉ thị của chương trình được thực thi tuyến tính, hay việc truy xuất các phần tử trong mảng thường diễn ra một cách tuyến tính.

→ Tận dụng nguyên lý này, các nhà sản xuất cài đặt bộ nhớ máy tính dưới dạng một hệ thống bộ nhớ phân cấp. Càng gần bộ xử lý, tốc độ càng cao và chi phí càng cao, dung lượng càng nhỏ



- Mục tiêu của hệ thống bộ nhớ phân cấp: cung cấp cho người dùng lượng bộ nhớ lớn có thể của bộ nhớ rẻ nhất, đồng thời cũng cung cấp khả năng truy xuất với tốc độ của bộ nhớ nhanh nhất. Để thực hiện ước mơ này cũng một phần nhờ nỗ lực của lập trình viên.

6.3. Nguyên lý hoạt động bộ nhớ đệm của bộ xử lý

- Bộ nhớ đệm hay bộ nhớ cache của bộ xử lý (CPU Cache) là cấp độ bộ nhớ nằm giữa bộ nhớ chính và bộ xử lý. Hệ thống bộ nhớ máy tính có thể có nhiều cấp độ bộ nhớ cache (L1, L2, L3).

- Bộ nhớ chính được chia thành các khối dữ liệu liền kề. Bộ nhớ cache cũng tương tự, có cùng kích thước nhưng số lượng ít hơn. Các khối được gọi là các dòng cache (cache line). Một dòng cache đại diện cho đơn vị nhỏ nhất có thể được di

chuyển giữa bộ nhớ cache và bộ nhớ chính. Các byte trong một dòng có thể được đọc và ghi riêng lẻ.

- Bộ xử lý không truy cập trực tiếp vào bộ nhớ cache. Phải xem dữ liệu cần có trong cache hay không, nếu có rồi thì gọi là "cache hit" và dữ liệu sẽ được truy xuất từ bộ nhớ cache. Ngược lại, nếu không tìm thấy thì được gọi là "cache miss". Khi miss xảy ra, khối bộ nhớ chứa dữ liệu cần truy xuất được sao chép từ bộ nhớ chính vào bộ nhớ cache cho các lần truy xuất sau đó.

- Tỉ lệ hit (hit rate, hit ratio) được định nghĩa là tỉ lệ truy xuất bộ nhớ được tìm thấy trong bộ nhớ cache. Tỷ lệ này thường được sử dụng để đánh giá hiệu năng của bộ nhớ phân cấp.

- Tỉ lệ mis (miss rate, miss ratio), ngược lại, là tỉ lệ truy xuất bộ nhớ không tìm thấy trong bộ nhớ cache.

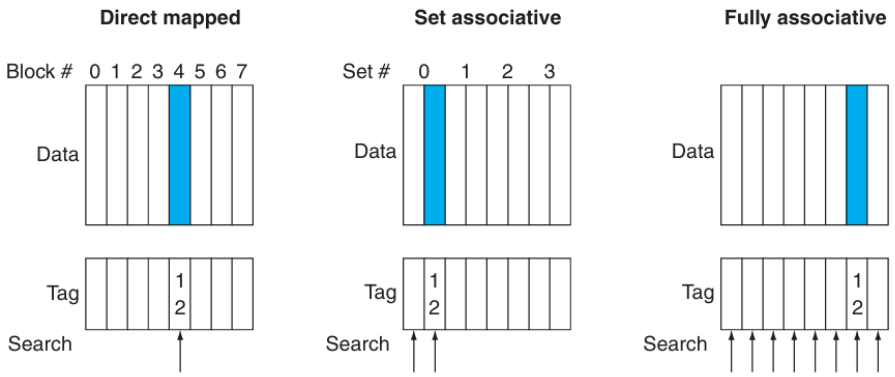
- Một cache miss đắt hơn nhiều so với một cache hit do tốc độ bộ nhớ chính chậm hơn nhiều so với bộ nhớ cache.

- Một trong những tham số quan trọng của các bộ nhớ cache là tính liên kết hay khả năng liên kết (associativity). Tính liên kết của bộ nhớ cache do biết số lượng dòng cache mà một khối bộ nhớ cụ thể sẽ được ánh xạ vào.

Đặc điểm	**Liên kết Toàn phần**	**Ánh xạ Trực tiếp**	**Liên kết Tập hợp**
Vị trí lưu trữ	Bất kỳ dòng nào trong bộ nhớ đệm	Mỗi khối bộ nhớ ánh xạ đến một dòng duy nhất	Mỗi khối ánh xạ đến một tập, có thể vào nhiều dòng trong tập
Tìm kiếm	Tìm tất cả các dòng để tìm thẻ khớp	Chỉ cần kiểm tra một dòng duy nhất	Chỉ kiểm tra trong một tập nhất định
Khả năng xung đột	***Ánh xạ mềm, ít xung đột*** vì khối có thể lưu ở bất kỳ đâu	***Ánh xạ cứng, dễ bị xung đột*** nếu nhiều khối ánh xạ đến cùng dòng	Ánh xạ mềm, trung bình, ít xung đột hơn ánh xạ trực tiếp
Hiệu suất	Tỷ lệ ***lỗi bộ nhớ đệm thấp*** nhưng thời gian truy cập có thể ***chậm*** hơn	Tỷ lệ ***lỗi bộ nhớ đệm cao*** hơn nhưng truy cập ***nhanh*** hơn	Hiệu suất tốt, giảm xung đột với chi phí hợp lý
Chi phí triển khai	Cao, yêu cầu phần cứng ***phức tạp*** để tìm kiếm toàn bộ bộ nhớ đệm	Thấp, phần cứng ***đơn giản*** hơn	***Trung bình***, chi phí hợp lý giữa hai loại trên

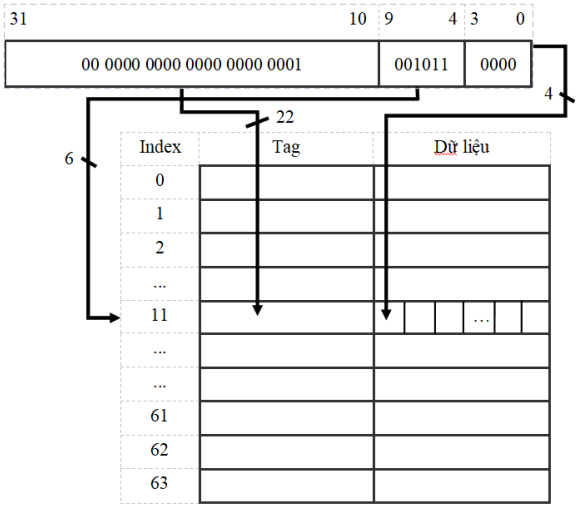
Ứng dụng phổ biến	Hệ thống cần độ chính xác cao, ít xung đột	Các hệ thống yêu cầu hiệu suất cao, chi phí thấp	Các hệ thống cần cân bằng giữa hiệu suất và chi phí

Ví dụ: Khối có địa chỉ 12 được đưa vào cache



*** Ánh xạ trực tiếp:**

Ví dụ đề bài: Ánh xạ trực tiếp, 64 dòng cache (6 bits), kích thước một dòng 16 byte (4 bits) (do mỗi ô 1 byte) cho biết vị trí cần tham chiếu, bộ nhớ 1200 được ánh xạ vào cache



*** Ánh xạ liên kết tập hợp:**

Ví dụ: Ánh xạ tập hợp: Nếu có n byte kích thước trong một dòng, có 8 dòng, nếu chia thành 2 tập hợp (2 đường). Thì tức là trong một set sẽ có 2 dòng. Ta lấy số dòng cache/số đường là ra 4 set. Nếu có một địa chỉ ánh xạ vào thì có thể vào đường 1 hoặc đường 2 tùy vào thuật toán như:

Set 0	0	Mem[8]
	1	Mem[12]
Set 1	0	Mem[9]
	1	Mem[5]
Set 2	0	Mem[2]
	1	Mem[6]
Set 3	0	Mem[7]
	1	Mem[19]

+ LRU (Least Recently Used) thay thế dòng cache ít sử dụng,

+ LFU (Least Frequently Used) giữ lại khối có tần số sử dụng cao

MỤC LỤC

CHƯƠNG 1: MÁY TÍNH CÁC KHÁI NIỆM VÀ CÔNG NGHỆ 1

A. LÝ THUYẾT 1
1.1. Giới thiệu 1
1.2 Lịch sử phát triển 2
1.3. Bên dưới chương trình ứng dụng 2
1.4. Bên trong máy tính 4
1.5. Phân biệt kiến trúc RIST và CISC 8
1.5.1. RISC (Reduced Instruction Set Computer): 8
1.5.2. CISC (Complex Instruction Set Computer): 9
1.6. Sự khác nhau giữa kiến trúc máy tính và tổ chức máy tính 9
B. BÀI TẬP 10
1.5. Các hệ đếm cơ bản 10
1.6. Các phép tính trong hệ nhị phân 11
1.7. Biểu diễn số nguyên 12
1.8. Biểu diễn số thực có dấu chấm động 13
1.9. Hiệu suất máy tính 15

CHƯƠNG 2: KIẾN TRÚC BỘ LỆNH 16

2.1. Giới thiệu 16
2.2. Các phép tính 16
2.3. Toán hạng 16
2.3.1. Toán hạng thanh ghi (Register Operands) 16
2.3.2. Toán hạng bộ nhớ (Memory Operands) 17
2.3.3. Toán hạng bằng (Constant or Immediate Operands) 19
2.4. Số có dấu và không dấu 19
2.5. Biểu diễn lệnh 19
2.5.1 R- Format. 19
2.5.2 I – Format và J – Format 20
2.6. Các phép tính logic 20
2.7 Các lệnh điều kiện và nhảy 21
2.8 Thủ tục (Procedure) cho assembly MIPS 22

CHƯƠNG 3: PHÉP TOÁN SỐ HỌC TRÊN MÁY TÍNH 26

3.1. Phép cộng & Phép trừ 26
3.2. Phép nhân 27
3.3. Phép chia 29
3.4. Số chấm động 32

CHƯƠNG 4: BỘ XỬ LÝ PROCESSOR 34

4.1. Giới thiệu 34

4.2. Nhắc lại các quy ước thiết kế logic 34
4.3. Xây dựng đường dữ liệu (datapath) đơn giản..... 35
4.3.1. Giai đoạn tìm nạp lệnh (Instruction Fetch) 37
4.3.2. Giải mã và lấy các toán hạng cần thiết (Instruction Decode & Operand Fetch)..... 37
4.3.3. Giai đoạn sử dụng ALU hay giai đoạn thực thi (ALU)..... 39
4.3.4. Giai đoạn truy xuất vùng nhớ (Memory Access) 40
4.3.5. Giai đoạn ghi lại kết quả/ lưu trữ (Result Write)..... 40
4.4. Hiện thực datapath đơn chu kì..... 41

CHƯƠNG 5: KỸ THUẬT ỐNG DẪN (PINELINE)..... 42

5.1. Tổng quan về pipelining (Mục 4.5):..... 42
5.2. Thiết kế datapath và control cho pipelining (Mục 4.6): 42
5.3. Vấn đề về hazard trong pipelining: 44
5.3.1. Data Hazards (Xung đột dữ liệu): 44
5.3.2. Control Hazards (Xung đột điều khiển): 45
5.3.3. Structural Hazards (Xung đột cấu trúc):..... 46
5.3.4. Exception Hazards (Xung đột do ngoại lệ): 46

CHƯƠNG 6: HỆ THỐNG BỘ NHỚ PHÂN CẤP 47

6.1. Bộ nhớ chính 47
6.2. Phân cấp bộ nhớ 47
6.3. Nguyên lý hoạt động bộ nhớ đệm của bộ xử lý..... 48