KIẾN TRÚC MÁY TÍNH

ET4270

TS. Nguyễn Đức Minh

[Adapted from Computer Organization and Design, 4th Edition, Patterson & Hennessy, © 2008, MK]
[Adapted from Computer Architecture lecture slides, Mary Jane Irwin, © 2008, PennState University]

Tổ chức lớp

Số tín chỉ	3 (3-1-1-6)
Giảng viên	TS. Nguyễn Đức Minh
Văn phòng	C9-401
Email	minhnd1@gmail,com
Website	https://sites.google.com/site/fethutca/home • Username: ca.fet.hut@gmail.com • Pass: dungkhoiminh
Sách	*Computer Org and Design*, 3rd Ed., Patterson &Hennessy, ©2007 *Digital Design and Computer Architecture*, David Money Harris
Thí nghiệm	3 bài
Bài tập	Theo chương, đề bài xem trên trang web

Điểm số

Điều kiện thi **Lab**

Bài thi giữa kỳ **30%**

Bài tập 20% (Tối đa 100 điểm)

Tiến trình 10%

Tối đa: 100 điểm,

Bắt đầu: 50 điểm

Tích lũy, trừ qua trả lời câu hỏi trên lớp và đóng góp tổ chức lớp

Bài thi cuối kỳ **70%**

Lịch học

- Thời gian:
  - Từ 14h00 đến 17h20
  - Lý thuyết: 11 buổi x 135 phút / 1 buổi
  - Bài tập: 4 buổi x 135 phút / 1 buổi
  - Thay đổi lịch (nghỉ, học bù) sẽ được thông báo trên website trước 2 ngày

Kết luận chương 1

- Hệ thống máy tính được xây dựng **từ phân cấp các lớp trừu tượng**. Các chi tiết triển khai lớp dưới bị che khuất khỏi lớp trên.
- **Kiến trúc tập lệnh** – lớp giao tiếp giữa phần cứng và phần mềm mức thấp – là lớp trừu tượng quan trọng trong hệ thống máy tính.
- Phần cứng máy tính gồm **5 thành phần**: **đường dữ liệu, khối điều khiển, bộ nhớ, khối vào, và khối ra**. 5 thành phần đó kết nối với nhau bằng hệ thống bus theo mô hình **vonNeumann** hoặc mô hình **Havard**.
- Phương pháp đánh giá hiệu năng một hệ thống máy tính là dùng **thời gian thực hiện 1 chương trình**. Thời gian thực hiện chương trình được tính bằng công thức:

$$T_{cpu} = I \times CPI \times T_c$$

Nội dung

1. Hệ đếm và biểu diễn số trong máy tính (nhắc lại)
2. Kiến trúc tập lệnh
    1. Yêu cầu chức năng máy tính vonNeumman
    2. Yêu cầu chung của kiến trúc tập lệnh
    3. Kiến trúc tập lệnh MIPS
    4. Biên dịch
3. Các phép toán và cách thực hiện trong máy tính

Hệ đếm cơ số r

- Một số biểu diễn trong hệ đếm cơ số r gồm m chữ số trước dấu phẩy và n chữ số sau dấu phẩy:

$$D = (d_{m-1}d_{m-2}\ldots d_1 d_0, d_{-1}d_{-2}\ldots d_{-n})_r$$

Trong đó

  - $0 \le d_i \le r\text{-}1$ là các chữ số
  - $d_{m\text{-}1}$: chữ số có ý nghĩa lớn nhất
  - $d_{\text{-}n}$: chữ số có ý nghĩa nhỏ nhất

- Giá trị của D trong hệ cơ số 10:

$$D = \sum_{i=-n}^{m-1} d_i \cdot r^i$$

Các hệ đếm thông dụng

- Hệ cơ số 10: $r = 10; 0 \leq d_i \leq 9$
- Hệ cơ số 2: $r = 2; d_i \in (0,1)$; d_i được gọi là các bit
- Hệ cơ số 8: $r = 8; 0 \leq d_i \leq 7$
- Hệ cơ số 16: $r = 16; d_i \in (0,\ldots,9,A,B,C,D,E,F)$

- Máy tính dùng hệ cơ số 2, và 16

Chuyển từ thập phân sang nhị phân

- **Bước 1 - Phần nguyên:** Chia số cần đổi cho 2 lấy phần dư; Lấy thương chia tiếp cho 2 lấy phần dư; Lặp lại cho đến khi thương bằng 0; Phần dư cuối cùng là bit có giá trị lớn nhất (MSB), phần dư đầu tiên là bit có giá trị nhỏ nhất (trước dấu phẩy)
- **Bước 2 - Phần thập phân:** Nhân số cần đổi với 2, lấy phần nguyên của tích; Lấy phần thập phân của tích nhân tiếp với 2, lấy phần nguyên; Lặp lại đến khi tích bằng 0 hoặc tích bị lặp lại; Phần nguyên đầu tiên là bit đầu tiên, phần nguyên cuối cùng là bít cuối cùng (sau dấu phẩy).

Chuyển từ nhị phân sang hệ 16

- Nhóm số thập phân thành các nhóm 4 bít, lần lượt từ phải sang trái.
- Nhóm cuối cùng có thể có số bit nhỏ hơn 4

$$(\underbrace{b_{m-1}, b_{m-2}, b_{m-3}, b_{m-4}}_{h_{m/4-1}}, \cdots \underbrace{b_7, b_6, b_5, b_4}_{h_1}, \underbrace{b_3, b_2, b_1, b_0}_{h_0})$$

- Chuyển mỗi nhóm 4 bít thành 1 chữ số hệ 16

Hệ 2	Hệ 16	Hệ 2	Hệ 16	Hệ 2	Hệ 16	Hệ 2	Hệ 16
0001	1	0101	5	1001	9	1101	D
0010	2	0110	6	1010	A	1110	E
0011	3	0111	7	1011	B	1111	F
0100	4	1000	8	1100	C		

Ví dụ 2.1 – Chuyển đổi hệ đếm

- Chuyển đổi các số sau giữa các cơ số 10, 2 và 16
  - $(241{,}625)_{10}$
  - $(1101\ 0101{,}1001)_{2}$
  - $(4A{,}3F)_{16}$

Biểu diễn số nguyên không dấu

Hex	Binary	Decimal
0x00000000	0…0000	0
0x00000001	0…0001	1
0x00000002	0…0010	2
0x00000003	0…0011	3
0x00000004	0…0100	4
0x00000005	0…0101	5
0x00000006	0…0110	6
0x00000007	0…0111	7
0x00000008	0…1000	8
0x00000009	0…1001	9
	…	
0xFFFFFFFC	1…1100	$2^{32} - 4$
0xFFFFFFFD	1…1101	$2^{32} - 3$
0xFFFFFFFE	1…1110	$2^{32} - 2$
0xFFFFFFFF	1…1111	$2^{32} - 1$

2^{31} 2^{30} 2^{29} . . . 2^3 2^2 2^1 2^0 trọng số

31 30 29 . . . 3 2 1 0 vị trí

1 1 1 . . . 1 1 1 1 bit

1 0 0 0 . . . 0 0 0 0 - 1

$2^{32} - 1$

- Các số dương ➔ không cần bít dấu
- Khoảng biểu diễn: [0, 2^m-1]

Biểu diễn số nguyên bằng 1 bít dấu và độ lớn

$$(b_{m-1}, b_{m-2,} \cdots, b_1, b_0) = (-1)^{b_{m-1}} \sum_{i=0}^{m-2} b_i \times 2^i$$

- Trong đó:
  - Bít MSB b_{m-1} là bít dấu; $b_{m-1} = 0$ biểu diễn số dương, $b_{m-1} = 1$ biểu diễn số âm
  - Các bít còn lại biểu diễn 1 số nhị phân không dấu
- Có 2 biểu diễn số 0: 10…0 (-0) và 00…0(+0)
- Khoảng biểu diễn $[-(2^{m-1}-1), 2^{m-1}-1]$

Biểu diễn số nguyên bằng mã bù 2

$$(b_{m-1}, b_{m-2,} \cdots, b_1, b_0)_{\text{bù 2,m bit}} = -b_{m-1} \times 2^{m-1} + \sum_{i=0}^{m-2} b_i \times 2^i$$

- Trong đó:
  - Bít MSB b_{m-1} là bít dấu; $b_{m-1} = 0$ biểu diễn số dương, $b_{m-1} = 1$ biểu diễn số âm
  - Các bít còn lại biểu diễn giá trị của số ở dạng mã bù 2
- Có 1 biểu diễn số 0: 00…0
- Khoảng biểu diễn $[-2^{m-1}, 2^{m-1}-1]$
- Quy tắc tìm biểu diễn bù 2, m bít :
  - Đổi số dương sang mã nhị phân, thêm các bít 0 để đủ m bít. (Bít lớn nhất là bít dấu = 0)
  - Tìm mã bù 1 bằng cách đảo các bít
  - Mã bù 2 = mã bù 1 + 1

Biểu diễn số nguyên bằng mã bù 2

❖ Quy tắc tìm biểu diễn bù 2, 4 bít :

1000	-8	$=-2^3$
1001	-7	$=-(2^3 - 1)$
1010	-6	
1011	-5	
1100	-4	
1101	-3	
1110	-2	
1111	-1	
0000	0	
0001	1	
0010	2	
0011	3	
0100	4	
0101	5	
0110	6	
0111	7	$=2^3 - 1$

Biểu diễn số nguyên mã lệch (bias)

$$(b_{m-1}, b_{m-2,} \cdots, b_1, b_0)_{2,bias} = \sum_{i=0}^{m-1} b_i \times 2^i - bias$$

- Trong đó: bias là độ lệch và thường bằng 2^{m-1}
- Không có bit dấu
- Khoảng biểu diễn: $[-2^{m-1}, 2^{m-1}-1]$

Ví dụ 2.2 – Biểu diễn số nguyên

- Chuyển các số sau sang dạng mã bù 1, mã bù 2 và mã 2 lệch 127 độ dài 8 bít
  - 123
  - -8
  - -3
  - -126
  - 129
  - -129

Biểu diễn số thực chuẩn IEEE-754

❖ Độ chính xác đơn dùng 32 bit nhị phân

31	30 ... 23	22 ... 0
S	**Mũ exp: số nguyên lệch 127**	**Độ lớn chuẩn hóa M**
s	$e_7e_6\ldots e_0$	$f_{22}f_{21}\ldots f_0$

- 1 bít dấu s: 0 số dương; 1 số âm
- 8 bít biểu diễn số mũ theo mã lệch 127:

$$\exp = (e_7e_6\ldots e_0)_{2,bias127} = \sum_{i=0}^{7} 2^i \times e_i - 127$$

- 23 bít biểu diễn phần độ lớn được chuẩn hóa $1 \le M < 2$

$$M = (1,f_{22}f_{21}\ldots f_0)_2$$

$$R_{IEEE-754} = (-1)^s \times 2^{\exp} \times M$$

Biểu diễn số thực chuẩn IEEE-754

❖ Biểu diễn các số đặc biệt

Số mũ lệch 127	Độ lớn	Số được biểu diễn
0	0	0
1 to 254	Bất kỳ	Số chuẩn hóa
255	0	$\pm\infty$
255	Số khác 0	NaN
0	Số khác 0	Số không chuẩn hóa

Biểu diễn số thực chuẩn IEEE-754

❖ Khoảng biểu diễn

	Độ chính xác đơn	Độ chính xác kép
Machine epsilon Độ chính xác	2^{-23} or 1.192×10^{-7}	2^{-52} or 2.220×10^{-16}
Smallest positive Số dương nhỏ nhất	2^{-126} or 1.175×10^{-38}	2^{-1022} or 2.225×10^{-308}
Largest positive Số dương lớn nhất	$(2- 2^{-23}) 2^{127}$ or 3.403×10^{38}	$(2- 2^{-52}) 2^{1023}$ or 1.798×10^{308}
Decimal Precision Độ chính xác thập phân	6 significant digits 6 chữ số sau dấu phảy	15 significant digits 15 chữ số sau dấu phảy

Ví dụ 2.3 – Biểu diễn số thực dạng IEEE-754

- Đổi các số sau thành biểu diễn số thực dấu phẩy động độ chính xác đơn
  - 125,50
  - -56,25
- Đổi các biểu diễn số thực dấu phẩy động độ chính xác đơn sau thành số thực ở hệ 10
  - 1 1101 1001 11000…0 (tổng cộng 32 bít)
  - 0 1001 1101 10100…0 (tổng cộng 32 bít)

Biểu diễn ký tự, chữ số

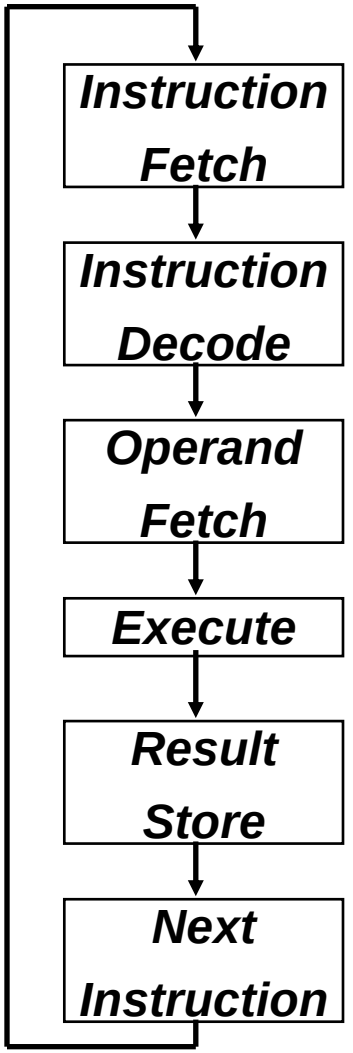
- Mã ASCII: 7 bit hoặc 8 bít
- Mã Unicode: 16 bít
- Mã BCD

b3b2b1b0	000	001	010	011	100	101	110	111
0000	NUL	DLE	SP	0	@	P	‘	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	“	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	V
0111	BEL	ETB	‘	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	|
1101	CR	GS	-	=	M	]	m	}
1110	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	_	o	DEL

Decimal digit	BCD
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

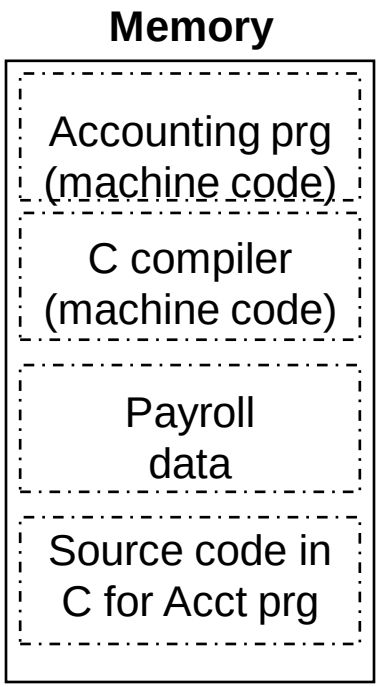
Máy tính vonNeumann: Hoạt động cơ bản

- Nạp chỉ thị từ bộ nhớ chương trình
- Xác định hành động và kích thước chỉ thị
- Định vị và nạp dữ liệu toán hạng
- Tính giá trị kết quả hoặc trạng thái
- Lưu dữ liệu vào bộ nhớ để sử dụng sau
- Xác định lệnh tiếp theo



Máy tính vonNeumann: Yêu cầu chức năng

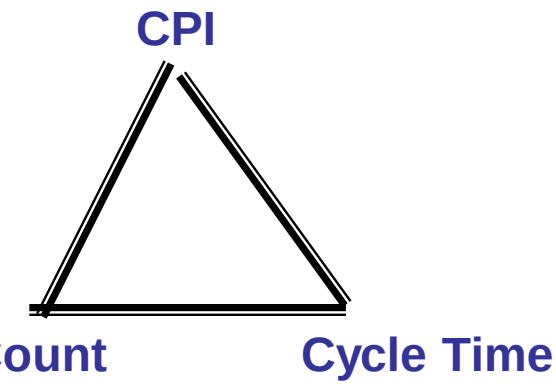
- Yêu cầu nguyên tắc:
  - Máy tính hoạt động theo các chỉ thị máy
  - Chỉ thị được biểu diễn bằng các số không phân biệt với dữ liệu
  - Chương trình được lưu trữ trong bộ nhớ
- Khái niệm về chương trình được lưu trữ (*eng. stored-program*):
  - Chương trình được cung cấp như là 1 tệp các số nhị phân.
  - Máy tính có thể chạy các chương trình đã tạo sẵn nếu chúng tương thích với 1 kiến trúc tập lệnh
  - Số lượng ít các kiến trúc tập lệnh
- ➔ Xác định yêu cầu chức năng: Kiến trúc tập lệnh



Kiến trúc tập lệnh: Đánh giá

- Thông số khi thiết kế:
  - Có thể triển khai được không? Mất bao lâu? Giá thành?
  - Có thể lập trình được không? Có dễ biên dịch?
- Thông số tĩnh:
  - Độ lớn chương trình trong bộ nhớ?
- Thông số động:
  - Số lượng chỉ thị được thực hiện? Số lượng byte cần nạp để chạy chương trình?
  - Số chu kỳ đồng hồ cần cho mỗi chỉ thị?
  - Tốc độ đồng hồ?

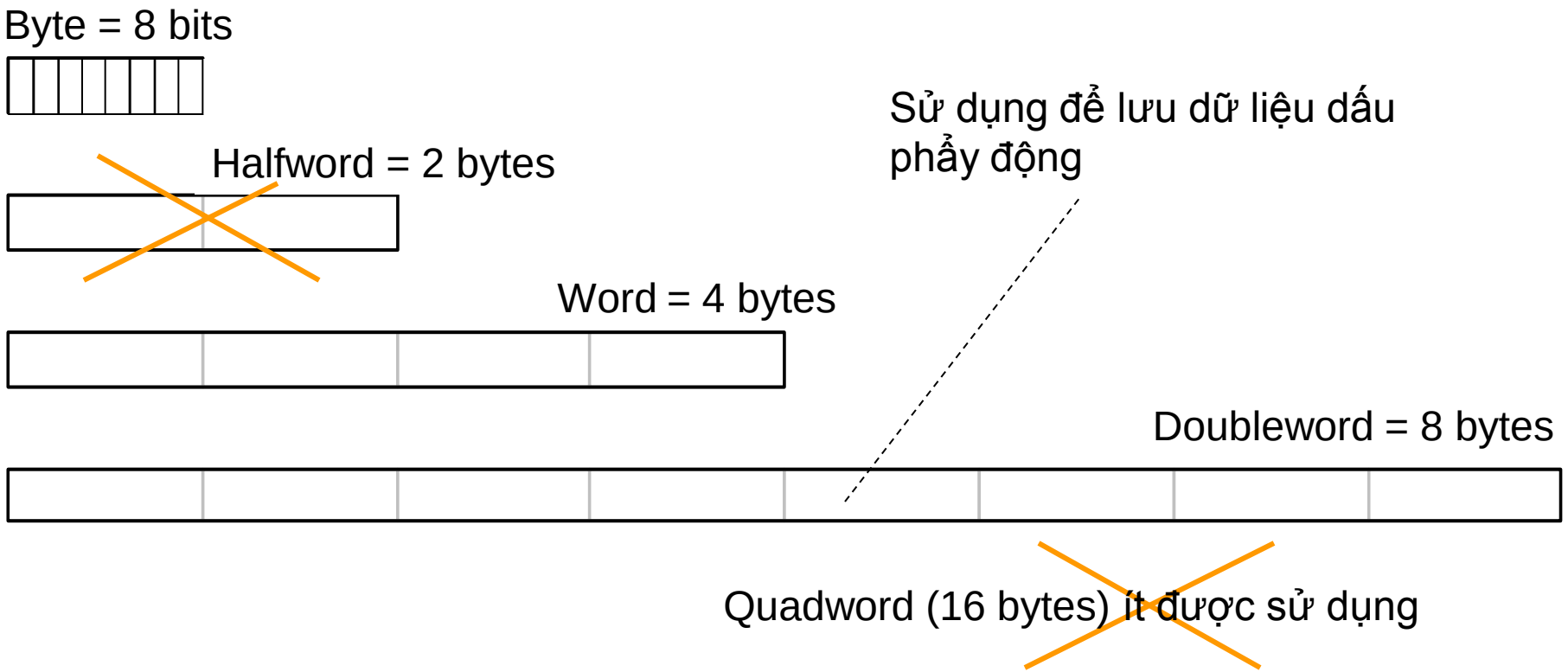
Thông số tốt nhất: Thời gian thực hiện!



Kiến trúc tập lệnh: Yêu cầu

- Kích thước và kiểu dữ liệu
- Phép toán: loại nào được hỗ trợ
- Định dạng và mã hóa chỉ thị:
  - Chỉ thị được giải mã thế nào?
- Vị trí toán hạng và kết quả
  - Số lượng toán hạng?
  - Giá trị toán hạng được lưu ở đâu?
  - Kết quả được lưu ở vị trí nào?
  - Các toán hạng bộ nhớ được định vị thế nào?
- Chỉ thị tiếp theo: nhẩy, điều kiện, rẽ nhánh

Dữ liệu: Kiểu & kích thước



Kiến trúc tập lệnh: Yêu cầu

- Kích thước và kiểu dữ liệu
- Phép toán: loại nào được hỗ trợ
- Định dạng và mã hóa chỉ thị:
  - Chỉ thị được giải mã thế nào?
- Vị trí toán hạng và kết quả
  - Số lượng toán hạng?
  - Giá trị toán hạng được lưu ở đâu?
  - Kết quả được lưu ở vị trí nào?
  - Các toán hạng bộ nhớ được định vị thế nào?
- Chỉ thị tiếp theo: nhẩy, điều kiện, rẽ nhánh

Các phép toán

- *Load/Store*: Đọc và ghi bộ nhớ
- *Computational*: Tính toán số học và logic, so sánh
  - Có lệnh nhân chia hay không?
  - Các lệnh so sánh nào?
- *Jump and Branch*: Nhẩy và rẽ nhánh
- *Floating Point*: Lệnh dấu phẩy động
  - *coprocessor*
- *Memory Management*: Quản lý bộ nhớ
- *Special*: Lệnh đặc biệt

Các phép toán

Dịch chuyển dữ liệu	Đọc (từ bộ nhớ), Ghi (tới bộ nhớ) Chuyển giữa các ô nhớ Chuyển giữa các thanh ghi Vào (từ thiết bị I/O), Ra (tới thiết bị I/O) push, pop (từ/tới ngăn xếp)
Số học	Số nguyên (nhị phân, thập phân), Số thực dấu phẩy động. Cộng, trừ, nhân chi
Dịch	Dịch trái/phải, Quay trái/phải
Logic	not, and, or, set, clear
Điều khiển (nhảy, rẽ nhánh)	Không điều kiện, Có điều kiện
Liên kết với thủ tục	call, return
Ngắt	trap, return
Đồng bộ	test & set
Chuỗi	search, translate
Đồ họa (MMX)	Phép toán song song

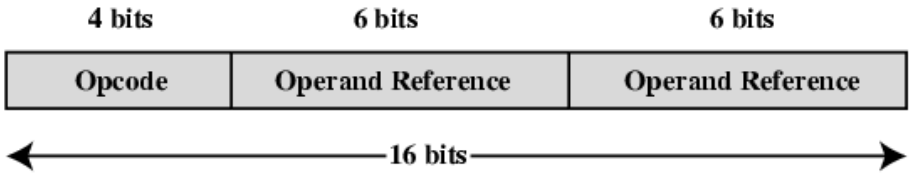
Các phép toán

- Các phép toán đơn giản được sử dụng nhiều và chiếm đa số trong các chỉ thị của chương trình
- Cần tập trung vào các phép toán:
  - *load, store*
  - *move register-register*
  - *add, subtract, and, shift*
  - *compare equal, compare not equal*
  - *branch, jump, call, return*

Kiến trúc tập lệnh: Yêu cầu

- Kích thước và kiểu dữ liệu
- Phép toán: loại nào được hỗ trợ
- Định dạng và mã hóa chỉ thị:
  - Chỉ thị được giải mã thế nào?
- Vị trí toán hạng và kết quả
  - Số lượng toán hạng?
  - Giá trị toán hạng được lưu ở đâu?
  - Kết quả được lưu ở vị trí nào?
  - Các toán hạng bộ nhớ được định vị thế nào?
- Chỉ thị tiếp theo: nhẩy, điều kiện, rẽ nhánh

Định dạng lệnh: các trường



- Mã lệnh chỉ ra nhiệm vụ (chức năng) của lệnh
- Tham chiếu toán hạng nguồn chỉ ra các toán hạng được xử lý bởi lệnh
- Tham chiếu kết quả chỉ ra nơi lưu trữ kết quả của lệnh
- Tham chiếu lệnh kế tiếp chỉ ra cách tính toán hoặc nơi lưu trữ lệnh sẽ được thực hiện tiếp theo
  - Thường không được chỉ ra rõ ràng trong lệnh mà được ngầm coi là lệnh liền sau lệnh hiện tại trong chuỗi lệnh
  - Trong một số loại lệnh, địa chỉ của lệnh tiếp theo sẽ được chỉ ra

Kiến trúc tập lệnh: Yêu cầu

- Kích thước và kiểu dữ liệu
- Phép toán: loại nào được hỗ trợ
- Định dạng và mã hóa chỉ thị:
  - Chỉ thị được giải mã thế nào?
- Vị trí toán hạng và kết quả
  - Số lượng toán hạng?
  - Giá trị toán hạng được lưu ở đâu?
  - Kết quả được lưu ở vị trí nào?
  - Các toán hạng bộ nhớ được định vị thế nào?
- Chỉ thị tiếp theo: nhẩy, điều kiện, rẽ nhánh

Số lượng toán hạng (1)

- 3 toán hạng:
  - Địa chỉ của 2 toán hạng, và kết quả đều được chứa trong mã lệnh
  - OP A, B, C ⇔ A ← B OP C
  - Dễ biên dịch từ ngôn ngữ bậc cao sang lệnh máy
  - Giá trị toán hạng lưu trong các thanh ghi
  - Lệnh chỉ ra chỉ số thanh ghi

- 2 toán hạng: (giá trị trong các thanh ghi, hoặc địa chỉ ô nhớ)
  - Một địa chỉ được dùng cho toán hạng và kết quả
  - OP A, B ⇔ A ← A OP B
  - Biên dịch đòi hỏi thêm lệnh và thanh ghi để lưu trữ tạm thời
  - Giá trị toán hạng lưu trong thanh ghi hoặc trong ô nhớ.
  - Lệnh chỉ ra chỉ số thanh ghi hoặc địa chỉ ô nhớ

Số lượng toán hạng (2)

- Một toán hạng: lệnh tích lũy (accumulator)
  - Một toán hạng và kết quả được quy định ngầm được lưu trong 1 thanh ghi đặc biệt (Accumulator – AC)
  - Toán hạng còn lại lưu trong thanh ghi
  - OP A ⇔ AC ← AC OP A
  - Thông dụng trong bộ xử lý tín hiệu số

- Không toán hạng: lệnh ngăn xếp (stack)
  - Tất cả các địa chỉ được quy định ngầm
  - Kết quả và toán hạng thứ hai nằm ở địa chỉ đỉnh của stack: T
  - Toán hạng thứ nhất nằm ở địa chỉ thứ 2 của stack: T-1
  - OP ⇔ T ← T-1 OP T

- Số lượng toán hạng quyết định: độ dài lệnh, I và CPI

Ví dụ 2.4: So sánh số lượng toán hạng

Xét câu lệnh ở ngôn ngữ bậc cao: Y = (A – B)/(C+D*E)

Biên dịch thành hợp ngữ:

3 địa chỉ

```
SUB   Y, A, B
MUL   T, D, E
ADD   T, T, C
DIV   Y, Y, T
```

2 địa chỉ

```
MOV   Y, A
SUB   Y, B
MOV   T, D
MUL   T, E
ADD   T, C
DIV   Y, T
```

1 địa chỉ

```
LOAD    D
MUL     E
ADD     C
STORE   Y
LOAD    A
SUB     B
DIV     Y
STORE   Y
```

0 địa chỉ

Chuyển sang dạng toán tử sau:

Y = AB-CDE*+/

```
PUSH    A
PUSH    B
SUB
PUSH    C
PUSH    D
PUSH    E
MUL
ADD
DIV
POP     Y
```

Kiến trúc tập lệnh: Yêu cầu

- Kích thước và kiểu dữ liệu
- Phép toán: loại nào được hỗ trợ
- Định dạng và mã hóa chỉ thị:
  - Chỉ thị được giải mã thế nào?
- Vị trí toán hạng và kết quả
  - Số lượng toán hạng?
  - Giá trị toán hạng được lưu ở đâu?
  - Kết quả được lưu ở vị trí nào?
  - Các toán hạng bộ nhớ được định vị thế nào?
- Chỉ thị tiếp theo: nhẩy, điều kiện, rẽ nhánh

Giá trị toán hạng – Chế độ địa chỉ

Register	Add R4,R3	R4← R4+R3
Immediate	Add R4,#3	R4 ← R4+3
Displacement	Add R4,100(R1)	R4 ← R4+Mem[100+R1]
Register indirect	Add R4,(R1)	R4 ← R4+Mem[R1]
Indexed / Base	Add R3,(R1+R2)	R3 ← R3+Mem[R1+R2]
Direct or absolute	Add R1,(1001)	R1 ← R1+Mem[1001]
Memory indirect	Add R1,@(R3)	R1 ← R1+Mem[Mem[R3]]
Auto-increment	Add R1,(R2)+	R1 ← R1+Mem[R2]; R2 ← R2+d
Auto-decrement	Add R1,–(R2)	R2 ← R2–d; R1 ← R1+Mem[R2]
Scaled	Add R1,100(R2)[R3]	R1 ← R1+Mem[100+R2+R3*d]

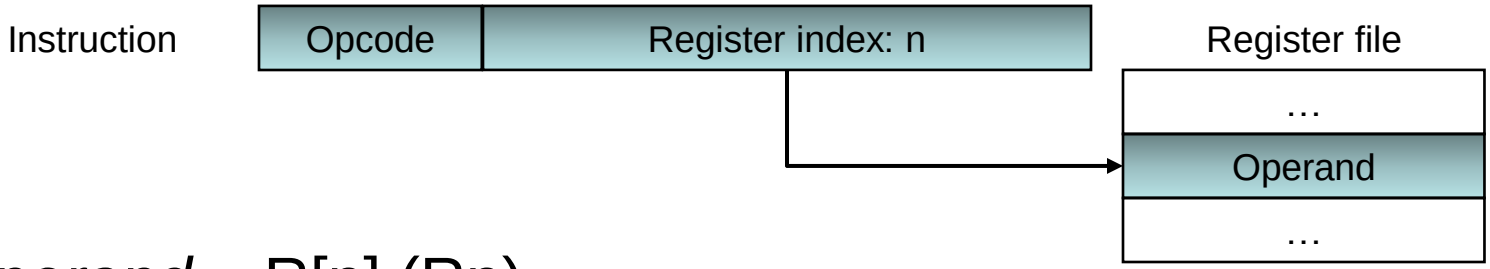
Chế độ địa chỉ tức thì (*Immediate*)

- Giá trị của toán hạng (toán hạng) là trường toán hạng của câu lệnh

- *Operand* = Operand field
- Không tham chiếu đến bộ nhớ để nạp dữ liệu
- Toán hạng luôn là hằng số trong khi chạy chương trình
- Tốc độ cao
- Khoảng giá trị của toán hạng nhỏ
- Ví dụ: ADD R4, #3: R4 ← R4+3

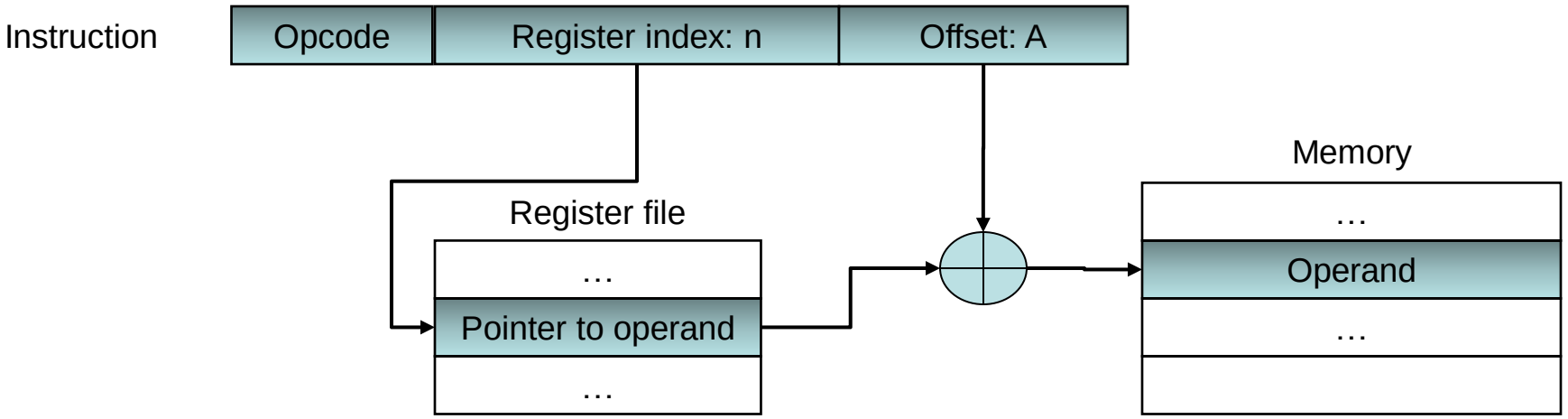
Chế độ địa chỉ thanh ghi (*Register*)

- Toán hạng được chứa trong thanh ghi chỉ ra bởi trường địa chỉ



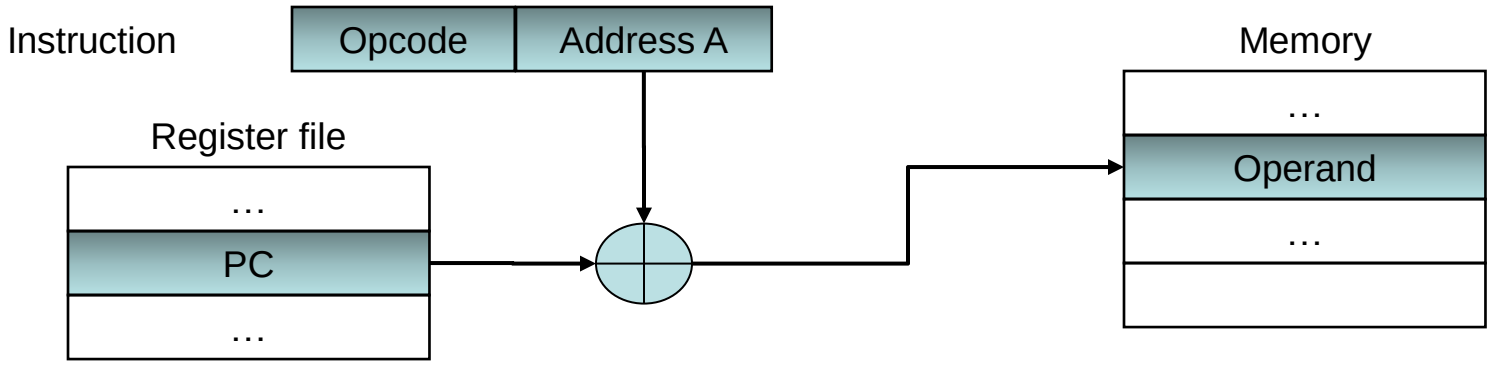
- *Operand* = R[n] (Rn)
- Không truy cập bộ nhớ
- Thực thi nhanh
- Trường địa chỉ dùng ít bit
  - Lệnh ngắn hơn
  - Nạp lệnh nhanh hơn
- Số lượng thanh ghi bị hạn chế

Chế độ địa chỉ dịch chuyển (*Displacement*)



- Trường địa chỉ chứa gồm 2 phần cơ sở và độ lệch
  - A chứa giá trị được sử dụng trực tiếp
  - n chứa chỉ số của thanh ghi để sử dụng gián tiếp
  - A, Rn có thể là cơ sở và độ lệch hoặc ngược lại
- Địa chỉ của toán hạng EA = R[n] + A
- Operand = MEM[EA]

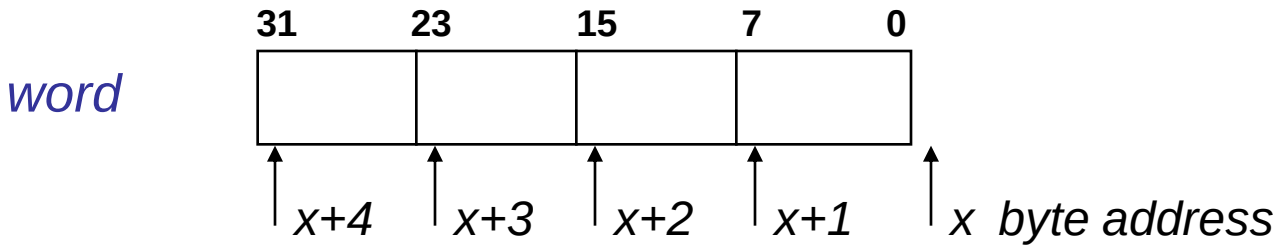
Chế độ địa chỉ tương đối (*Relative*)



- Phiên bản của địa chỉ dịch chuyển
- R = PC, được ngầm định trong mã lệnh (opcode)
- operand = MEM[A + PC]
- Lấy toán hạng từ địa chỉ cách vị trí chương trình hiện tại A ô nhớ
- Dùng để truy cập các hằng số, biến, địa chỉ địa phương

Địa chỉ bộ nhớ

- Địa chỉ bộ nhớ:
  - Địa chỉ byte: đánh địa chỉ cho các ô nhớ kích thước 1 byte
  - Địa chỉ word: đánh địa chỉ cho các ô nhớ kích thước 1 word
- 2 câu hỏi khi thiết kế ISA:
  - Các kiểu dữ liệu lớn hơn byte được lưu trữ thế nào?
  - Địa chỉ khác byte được tính toán thế nào?
  - Các kiểu dữ liệu lớn có được lưu trữ ở vị trí địa chỉ byte bất kỳ hay không? (Vấn đề *alighment*)



Địa chỉ bộ nhớ: Endianess và Alignment

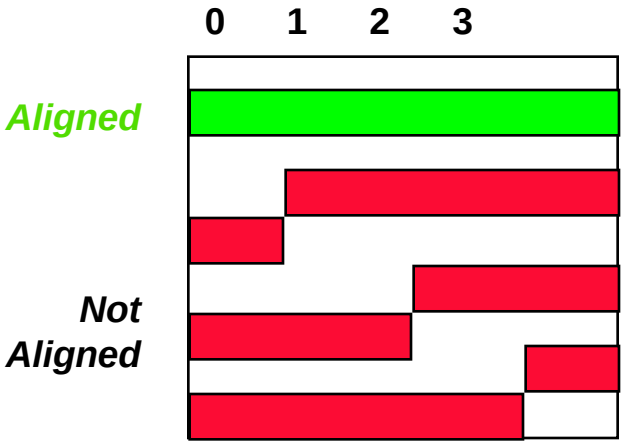
- Big Endian:
  - Địa chỉ word = địa chỉ của byte có ý nghĩa lớn nhất trong word (*Most Significant Byte*)
  - IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA
- Litle Endian:
  - Địa chỉ word = địa chỉ của byte có ý nghĩa nhỏ nhất trong word (*Least Significant Byte*)
  - Intel 80x86, DEC Vax, DEC Alpha
- Alignment:
  - Dữ liệu được lưu trữ ở địa chỉ byte chia hết cho kích thước.

Địa chỉ bộ nhớ: Endianess và Alignment

- Big Endian:
  - Địa chỉ word = địa chỉ của byte có ý nghĩa lớn nhất trong word (*Most Significant Byte*)
  - IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA
- Litle Endian:
  - Địa chỉ word = địa chỉ của byte có ý nghĩa nhỏ nhất trong word (*Least Significant Byte*)
  - Intel 80x86, DEC Vax, DEC Alpha
- Alignment:
  - Dữ liệu được lưu trữ ở địa chỉ byte chia hết cho kích thước.

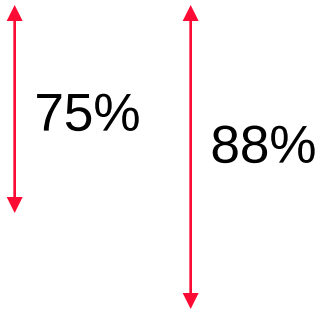
Địa chỉ bộ nhớ: Endianess và Alignment

- Big Endian:
  - Địa chỉ word = địa chỉ của byte có ý nghĩa lớn nhất trong word (*Most Significant Byte*)
  - IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA
- Litle Endian:
  - Địa chỉ word = địa chỉ của byte có ý nghĩa nhỏ nhất trong word (*Least Significant Byte*)
  - Intel 80x86, DEC Vax, DEC Alpha
- Alignment:
  - Dữ liệu được lưu trữ ở địa chỉ byte chia hết cho kích thước.



Sử dụng chế độ địa chỉ

- Displacement: 42% avg, 32% to 55%
- Immediate: 33% avg, 17% to 43%
- Register deferred (indirect): 13% avg, 3% to 24%
- Scaled: 7% avg, 0% to 16%
- Memory indirect: 3% avg, 1% to 6%
- Misc: 2% avg, 0% to 3%



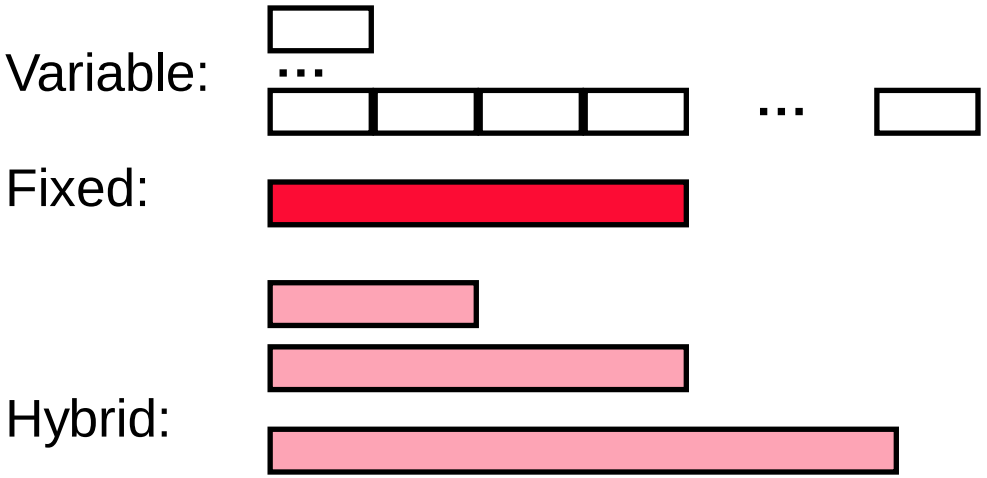
- 75% displacement & immediate
- 88% displacement, immediate & register indirect

Thống kê chế độ địa chỉ

- Kích thước trường toán hạng trực tiếp
  - 8 bit: 50-60%
  - 16 bit; 75%-80%
- Các chế độ địa chỉ quan trọng
  - Dịch chuyển (*Displacement*)
  - Trực tiếp (*Immediate*)
  - Thanh ghi gián tiếp (*Register indirect*)
- Kích thước độ lệch trong chế độ địa chỉ: 12-16 bít
- Kích thước toán hạng trực tiếp: 8-16 bít

Định dạng chỉ thị

- Độ dài chỉ thị:
  - Cố định
  - Thay đổi
  - Lai: gồm 1 vài loại chỉ thị có độ dài cố định khác nhau
- Khi kích thước chương trình quan trọng: dùng chỉ độ dài thay đổi
- Khi hiệu năng quan trọng: dùng độ dài cố định



Một số kiến trúc tập lệnh

	Complex Instruction Set (CISC)Computer			Reduced Instruction Set (RISC) Computer		Superscalar		
Characteristic	IBM 370/168	VAX 11/780	Intel 80486	SPARC	MIPS R4000	PowerPC	Ultra SPARC	MIPS R10000
Year developed	1973	1978	1989	1987	1991	1993	1996	1996
Number of instructions	208	303	235	69	94	225		
Instruction size (bytes)	2–6	2–57	1–11	4	4	4	4	4
Addressing modes	4	22	11	1	1	2	1	1
Number of general-purpose registers	16	16	8	40 - 520	32	32	40 - 520	32
Control memory size (Kbits)	420	480	246	—	—	—	—	—
Cache size (KBytes)	64	64	8	32	128	16-32	32	64

Kiến trúc RISC

- Reduce Instruction Set Computer
- DEC Alpha, AMD 29k, ARC, ARM, Atmel AVR, MIPS, PA-RISC, Power (PowerPC), SuperH, và SPARC.
- Định dạng lệnh và độ dài lệnh cố định, đơn giản
  - Dễ giải mã lệnh
- Các thanh ghi chung mục đích có thể sử dụng trong nhiều ngữ cảnh
  - Dễ thiết kế phần mềm biên dịch
  - Có thể cần các thanh ghi dấu phẩy động riêng biệt
- Chế độ địa chỉ đơn giản
  - Các chế độ địa chỉ phức tạp được thực hiện thông qua chuỗi lệnh số học và lệnh nạp/ghi
- Ít hỗ trợ các loại dữ liệu phức tạp

Kiến trúc CISC

- Complex Instruction Set Computer
- System/360, z/Architecture, PDP-11, VAX, Motorola 68k, và x86.
- Lệnh có độ dài thay đổi, phức tạp
  - Có thể bao gồm 1 vài phép toán nhỏ
  - Gần ngôn ngữ lập trình bậc cao
- Nhiều chế độ địa chỉ phức tạp
- Hỗ trợ các loại dữ liệu phức tạp

CISC vs. RISC

RISC	CISC
- Tập lớn các thanh ghi - Tập lệnh đơn giản - Tập trung trao đổi dữ liệu giữa thanh ghi - Các lệnh thực hiện trong một chu kỳ máy - Các lệnh LOAD/STORE đơn giản để truy cập bộ nhớ - Giới hạn chế độ địa chỉ - Từ mã có chiều dài cố định	- Giới hạn số thanh ghi - Tập lệnh phức tạp - Nhấn mạnh vào các hoạt động truy cập bộ nhớ - Lệnh có thể được thực hiện trong nhiều chu kỳ máy - Một lệnh có thể tương đương với nhiều lệnh của RISC - Nhiều chế độ địa chỉ - Mã lệnh có chiều dài thay đổi tùy vào từng lệnh

CISC vs. RISC

RISC	CISC
- Mã lệnh thực hiện nhanh - Đơn vị điều khiển đơn giản - Giải mã nhanh - Xử lý song song đường ống hiệu suất cao - Thiết kế, phát triển và kiểm tra nhanh - Hỗ trợ trình dịch tăng cường sự tối ưu - Giảm các lỗi rẽ nhánh của đường ống - Tăng tốc truyền tham số cho các thủ tục	- Ngôn ngữ lập trình assembly mạnh - Giảm các yêu cầu khi thiết kế trình dịch - Các tính năng với dấu phẩy động mạnh - Tăng khả năng của cache

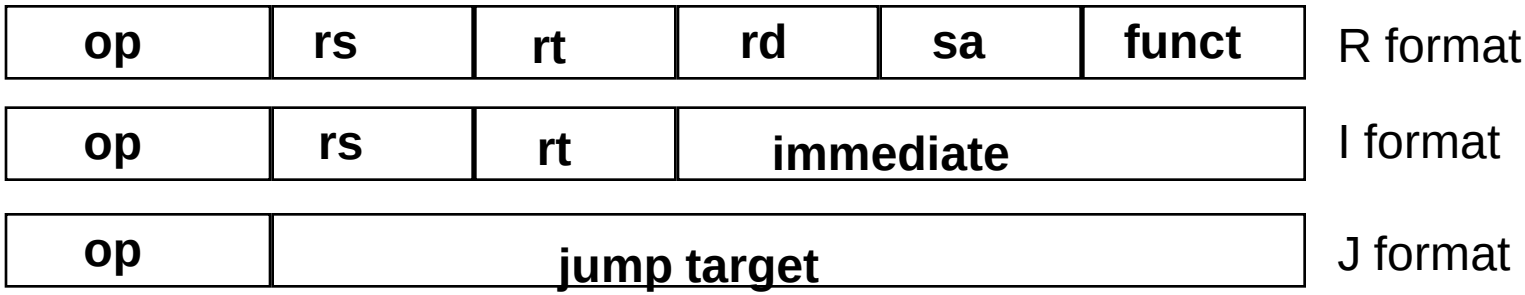
Ví dụ 2.4. So sánh hiệu năng RISV vs. CISC

Kiến trúc tập lệnh ISA có 2 lớp chỉ thỉ phức tạp (C) và đơn giản (S). Trong 1 chương trình thời gian thực hiện chỉ thị S chiếm 95%. Để triển khai ISA bằng kiến trúc RISC ta sẽ triển khai các chỉ thị S bằng phần cứng và chỉ thị C bằng phần mềm (dùng đoạn chỉ thị S và coi như 1 chỉ thị giả C). So sánh với kiến trúc CISC, các chỉ thị S sẽ được thực hiện nhanh hơn 20% và các chỉ thị CISC bị chậm đi 3 lần. Kiến trúc nào có hiệu năng cao hơn và cao hơn bao nhiêu lần?

Kiến trúc tập lệnh MIPS

Định dạng chỉ thị:

- 32 bit
- 3 loại định dạng:
  - R-chỉ thị thanh ghi: 2 toán hạng nguồn thanh ghi, 1 toán hạng đích thanh ghi
  - I-chỉ thị trực tiếp: 1 toán hạng nguồn thanh ghi, 1 toán hạng nguồn trực tiếp, 1 toán hạng đích thanh ghi
  - J-chỉ thị nhảy: 1 toán hạng nguồn trực tiếp



Nguyên tắc thiết kế MIPS (RISC)

- Tính đơn giản quan trọng hơn tính quy tắc(Simplicity favors regularity)
  - Chỉ thị kích thước cố định (32 bit)
  - Ít định dạng chỉ thị (3 loại định dạng)
  - Mã lệnh ở vị trí cố định (6 bit đầu)
- Nhỏ hơn thì nhanh hơn
  - Số chỉ thị giới hạn
  - Số thanh ghi giới hạn
  - Số chế độ địa chỉ giới hạn
- Tăng tốc các trường hợp thông dụng
  - Các toán hạng số học lấy từ thanh ghi (máy tính dựa trên cơ chế load-store)
  - Các chỉ thị có thể chứa toán hạng trực tiếp
- Thiết kế tốt đòi hỏi sự thỏa hiệp
  - 3 loại định dạng chỉ thị

Chỉ thị số học của MIPS

❖ Mã hợp ngữ của chỉ thị số học

```
add     $t0, $s1, $s2
sub     $t0, $s1, $s2
```

- Mỗi chỉ thị số học thực hiện một phép toán
- Mỗi chỉ thị chứa chính xác ba chỉ số của các thanh ghi trong tệp thanh ghi của đường dữ liệu (`$t0,$s1,$s2`)

destination ← source1 op source2

- Định dạng chỉ thị loại thanh ghi (R format)

0	17	18	8	0	0x22

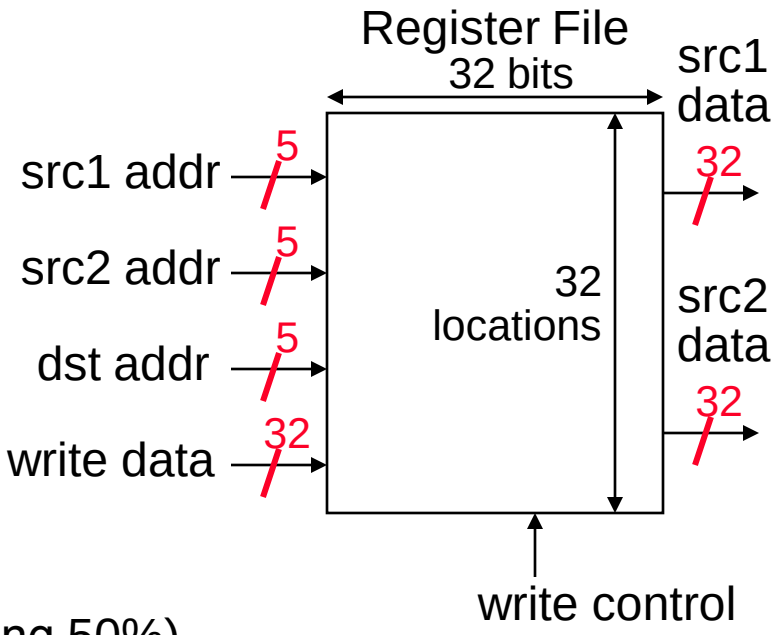
Các trường trong chỉ thị MIPS

Các trường trong 1 chỉ thị MIPS được đặt tên:

op	6-bits	mã lệnh xác định phép toán (opcode)
rs	5-bits	chỉ số thanh ghi chứa toán hạng nguồn 1 trong tệp thanh ghi
rt	5-bits	chỉ số thanh ghi chứa toán hạng nguồn 2 trong tệp thanh ghi
rd	5-bits	chỉ số thanh ghi sẽ lưu kết quả trong tệp thanh ghi
shamt	5-bits	số lượng dịch (cho chỉ thị dịch)
funct	6-bits	mã chức năng thêm cho phần mã lệnh

Tệp thanh ghi của MIPS

- Gồm 32 thanh ghi 32-bit
  - 2 cổng đọc
  - 1 cổng ghi
- Thanh ghi:
  - Nhanh hơn bộ nhớ chính
    - Nhiều thanh ghi sẽ chậm hơn
      (VD., 1 tệp gồm 64 thanh ghi word sẽ chậm hơn tệp gồm 32 thanh ghi khoảng 50%)
    - Số lượng cổng đọc ghi ảnh hưởng bậc 2 đến tốc độ
  - Dễ biên dịch
    - VD., (A*B) – (C*D) – (E*F) có thể thực hiện phép nhân theo thứ tự bất kỳ, không giống như ngăn xếp
  - Chứa biến chương trình
    - cải thiện độ lớn mã chương trình



Các thanh ghi MIPS

Tên	Chỉ số	Công dụng	Preserve on call?
$zero	0	constant 0 (hardware)	n.a.
$at	1	reserved for assembler	n.a.
$v0 - $v1	2-3	returned values	no
$a0 - $a3	4-7	arguments	yes
$t0 - $t7	8-15	temporaries	no
$s0 - $s7	16-23	saved values	yes
$t8 - $t9	24-25	temporaries	no
$gp	28	global pointer	yes
$sp	29	stack pointer	yes
$fp	30	frame pointer	yes
$ra	31	return addr (hardware)	yes

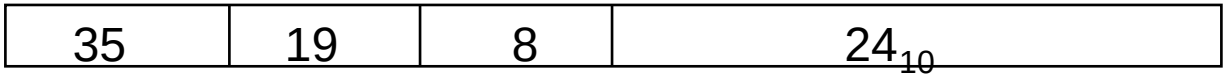
Truy cập bộ nhớ

- 2 chỉ thị dịch chuyển dữ liệu để truy cập bộ nhớ
  - `lw  $t0, 4($s3)  #đọc 1 từ từ bộ nhớ`
  - `sw  $t0, 8($s3)  #ghi 1 từ vào bộ nhớ`
- Dữ liệu được đọc vào (lw) hoặc ghi ra từ (sw) 1 thanh ghi trong tệp thanh ghi – 5 bit chỉ số thanh ghi
- 32 bit địa chỉ bộ nhớ được tạo ra bằng cách cộng giá trị thanh ghi cơ sở (*base register*) với giá trị offset
  - Trường offset rộng 16 bit sẽ giới hạn các ô nhớ trong khoảng $\pm 2^{13}$ hay 8,192 words ($\pm 2^{15}$ hay 32,768 bytes) tính từ giá trị của thanh ghi cơ sở

Định dạng lệnh truy cập bộ nhớ

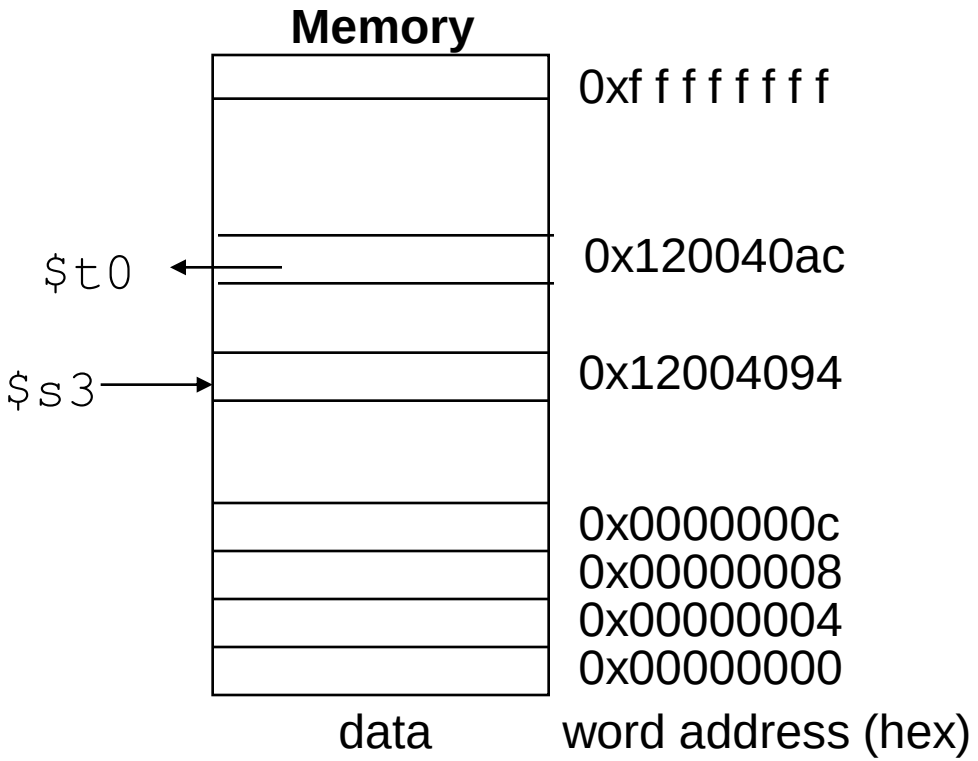
- Định dạng chỉ thị Load/Store (Định dạng I):

```
lw $t0, 24($s3)
```



24_{10} + $s3 =

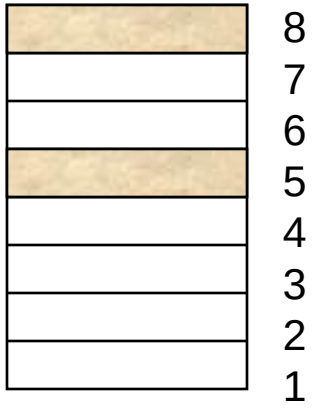
. . . 0001 1000
+ . . . 1001 0100
. . . 1010 1100 =
0x120040ac



Ví dụ 2.5. Truy cập bảng (*array*)

- Cho A[ ] = là 1 mảng bắt đầu tại địa chỉ cơ sở lưu trong thanh ghi $s3;
- Biến *h* được gắn với thanh ghi $s2;
- Dịch: A[5] = h + A[8]
- Thành mã hợp ngữ MIPS:

```
lw   $t0, 32 ($s3)      # $t0 ← A[8]
add $t0, $s2, $t0       # $t0 ← h+$t0
sw  $t0, 20 ($s3)       # A[5] ← $t0
```



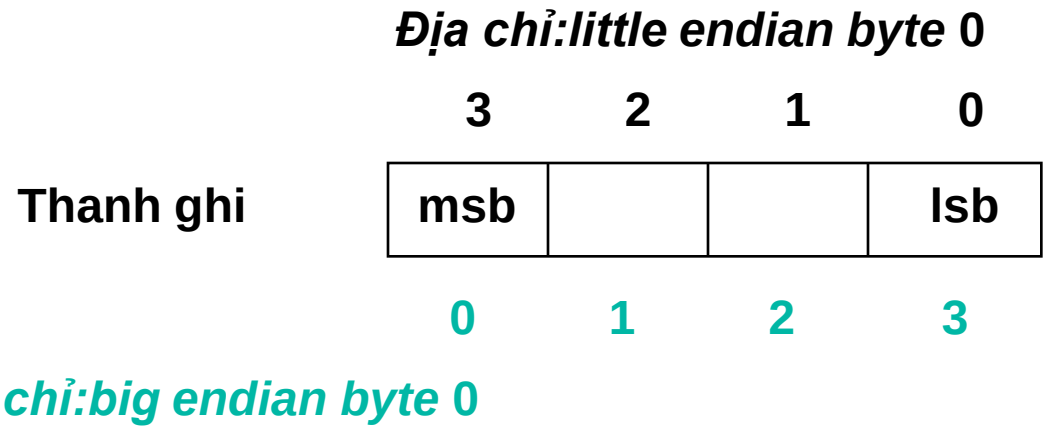
Ví dụ 2.6. Truy cập mảng với chỉ số thay đổi

- A[ ] = array with *base address* in $s3;
- *variables g, h, i* associated with registers $s1, $s2, $s4
- Compile: g = h + A[i]
- into MIPS instructions:

```
add  $t1, $s4, $s4      # $t1 ← i+i = 2i
add  $t1, $t1, $t1      # $t1 ← 2i+2i = 4i
add  $t1, $t1, $s3      # $t1 ← address of A[i]
lw   $t0, 0 ($t1)       # $t0 ← A[i]
add  $s1, $s2, $t0      # $s1 ← h + A[i]
```

Lưu trữ byte: Endianess (Nhắc lại)

- Big Endian: leftmost byte is word address
- Little Endian: rightmost byte is word address



Đọc và ghi byte

- MIPS các lệnh đặc biệt để dịch chuyển bytes

```
lb   $t0, 1($s3)   #load byte from memory
sb   $t0, 6($s3)   #store byte to  memory
```

- Các byte 8bit nào được đọc và ghi?
  - Lệnh đọc đưa byte được đọc vào 8 bit ngoài cùng bên phải từ bộ nhớ vào thanh ghi đích
    - Các bit khác của thanh ghi?
  - Lệnh ghi lấy 8 bit ngoài cùng bên phải của thanh ghi nguồn và ghi vào bộ nhớ
    - Giữ các byte khác trong từ nhớ không thay đổi

Ví dụ 2.7. Đọc ghi byte

- Cho đoạn mã sau và trạng thái bộ nhớ. Xác định trạng thái bộ nhớ sau khi thực hiện đoạn mã

```
add   $s3, $zero, $zero
lb    $t0, 1($s3)
sb    $t0, 6($s3)
```

- Giá trị lưu trong $t0?

Memory	Word Address (Decimal)
0x 0 0 0 0 0 0 0 0	24
0x 0 0 0 0 0 0 0 0	20
0x 0 0 0 0 0 0 0 0	16
0x 1 0 0 0 0 0 1 0	12
0x 0 1 0 0 0 4 0 2	8
0x F F F F F F F F	4
0x 0 0 9 0 1 2 A 0	0
Data	

- Từ nào được ghi vào bộ nhớ ở vị trí nào?
- Điều gì xảy ra khi máy tính là loại little Endian?

Đọc ghi nửa từ

- MIPS also provides special instructions to move half words

```
lh   $t0, 1($s3)   #load half word from memory
sh   $t0, 6($s3)   #store half word to  memory
```

- What 16 bits get loaded and stored?
  - load half word places the half word from memory in the rightmost 16 bits of the destination register
    - what happens to the other bits in the register?
  - store half word takes the half word from the rightmost 16 bits of the register and writes it to the half word in memory
    - leaving the other half word in the memory word unchanged

Lệnh trực tiếp

- Các hằng số trong chương trình thường có giá trị nhỏ
- Các phương pháp lưu trữ và sử dụng hằng số:
  - Lưu các hằng thường dùng trong bộ nhớ và đọc chúng
  - Tạo 1 thanh ghi kết nối cứng (như $zero) để lưu hằng số
  - Dùng các lệnh đặc biệt có chứa hằng số

```
addi  $sp, $sp, 4     #$sp = $sp + 4

slti $t0, $s2, 15     #$t0 = 1 if $s2<15
```

- Định dạng mã máy (Định dạng I):

- Hằng số được chứa trong lệnh!
  - Định dạng trực tiếp giới hạn giá trị trong khoảng $+2^{15}–1$ to -2^{15}

Lệnh dịch

- Shifts move all the bits in a word left or right

```
sll $t2, $s0, 8     #$t2 = $s0 << 8 bits
srl $t2, $s0, 8     #$t2 = $s0 >> 8 bits
```

- Instruction Format (R format)

0		16	10	8	0x00

- Such shifts are called logical because they fill with zeros
  - Notice that a 5-bit shamt field is enough to shift a 32-bit value $2^5 - 1$ or 31 bit positions

Lệnh logic

- There are a number of bit-wise logical operations in the MIPS ISA

```
and $t0, $t1, $t2  #$t0 = $t1 & $t2
or  $t0, $t1, $t2  #$t0 = $t1 | $t2
nor $t0, $t1, $t2  #$t0 = not($t1 | $t2)
```

- Instruction Format (R format)


```
andi $t0, $t1, 0xFF00   #$t0 = $t1 & ff00
ori  $t0, $t1, 0xFF00   #$t0 = $t1 | ff00
```

- Instruction Format (I format)

Sử dụng các hằng số lớn

- Đưa 1 hằng số 32 bit vào 1 thanh ghi
- Sử dụng 2 lệnh:
  - Lệnh nạp vào phần cao "load upper immediate“

```
lui $t0, 0xaaaa
```

16	0	8	1010101010101010

  - Lệnh nạp vào phần thấp:

```
ori $t0, $t0, 0xaaaa
```

1010101010101010	0000000000000000
0000000000000000	1010101010101010

1010101010101010	1010101010101010

Lệnh điều khiển dòng chương trình

- Lệnh rẽ nhánh có điều kiện:

```
bne $s0, $s1, Lbl #go to Lbl if $s0≠$s1
beq $s0, $s1, Lbl #go to Lbl if $s0=$s1
```

  - Ex:

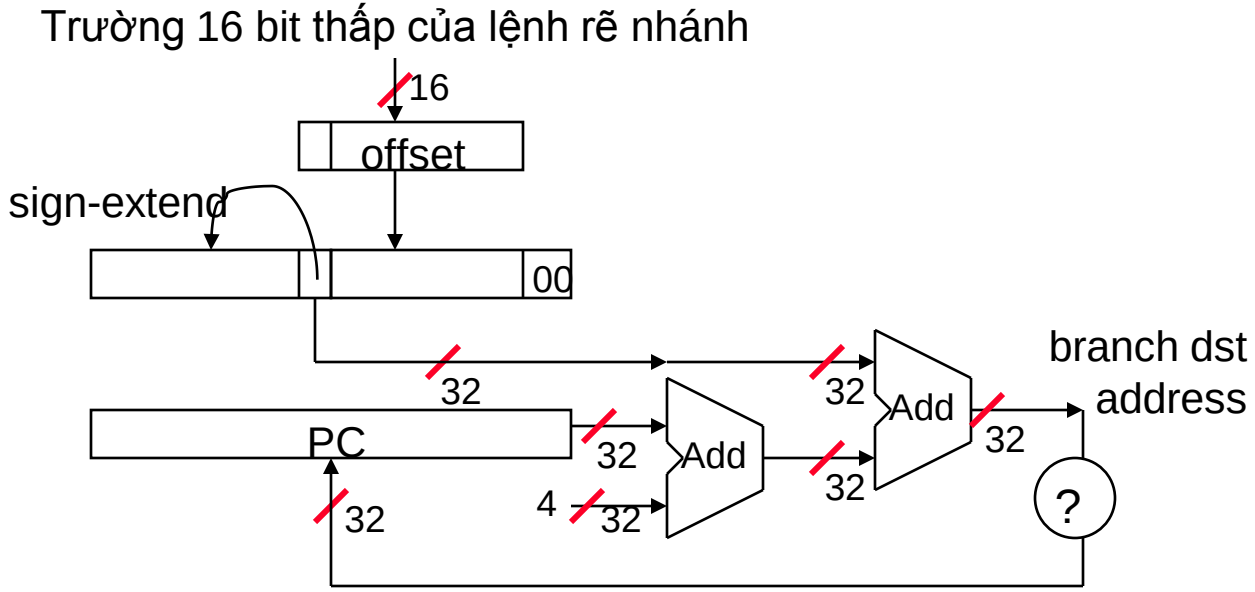
```
        if (i==j) h = i + j;

        bne $s0, $s1, Lbl1
        add $s3, $s0, $s1
Lbl1:   ...
```

- Định dạng lệnh (Định dạng I):

- Địa chỉ đến được xác định như thế nào ?

Xác định địa chỉ rẽ nhánh đến

- Sử dụng 1 thanh ghi (giống như lw và sw) cộng với 16-bit offset
  - Thanh ghi địa chỉ lệnh PC (*Instruction Address Register*)
    - Việc sử dụng PC được tự động bao hàm trong lệnh
    - PC được cập nhật (PC+4) khi lệnh được nạp vì vậy khi tính toán nó chứa giá trị địa chỉ của lệnh kế tiếp
  - giới hạn khoảng cách rẽ nhánh trong khoảng -2^{15} đến $+2^{15}-1$ (word) lệnh kể từ lệnh sau lệnh rẽ nhánh. Tuy nhiên phần lớn các rẽ nhánh là địa phương.



So sánh hỗ trợ lệnh rẽ nhánh

- Có lệnh `beq`, `bne`, các loại điều kiện khác? (VD., rẽ nhánh nếu nhỏ hơn)? Cần 1 lệnh so sánh khác: `slt`
- Set on less than:

```
slt $t0, $s0, $s1     # if $s0 < $s1     then
                      # $t0 = 1          else
                      # $t0 = 0
```

- Instruction format (R format):

- Các phiên bản khác của `slt`

```
slti $t0, $s0, 25     # if $s0 < 25 then $t0=1 ...
sltu $t0, $s0, $s1    # if $s0 < $s1 then $t0=1 ...
sltiu $t0, $s0, 25    # if $s0 < 25 then $t0=1 ...
```

Sử dụng slt

- Dùng `slt, beq, bne`, và giá trị 0 trong thanh ghi `$zero` để tạo ra các điều kiện rẽ nhánh khác
  - less than `blt $s1, $s2, Label`

```
slt  $at, $s1, $s2      #$at set to 1 if
bne  $at, $zero, Label  #$s1 < $s2
```

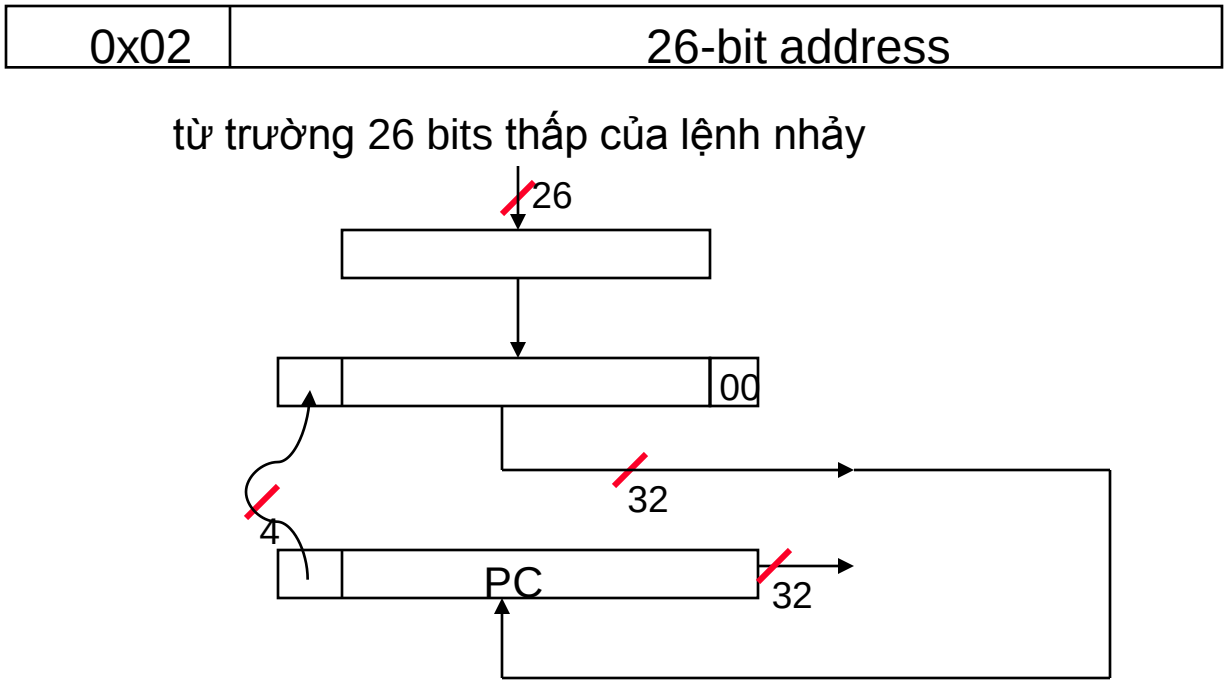
  - less than or equal to `ble $s1, $s2, Label`
  - greater than `bgt $s1, $s2, Label`
  - great than or equal to `bge $s1, $s2, Label`
- Các lệnh rẽ nhánh được thêm vào tập lệnh như các lệnh giả, được nhận dạng và mở rộng bằng trình dịch assembler
  - Trình dịch assembler cần thanh ghi riêng (`$at`)

Lệnh nhảy không điều kiện

- Lệnh nhảy không điều kiện:

```
j   label          #go to label
```

- Định dạng lệnh (J Format):



Nhảy đến địa chỉ ở xa

- Khi địa chỉ nhảy đến ở xa hơn, và không thể biểu diễn bằng 16 bits?

- Phần mềm assembler hỗ trợ – nó chèn 1 lệnh nhảy không điều kiện đến địa chỉ nhảy đến và đảo điều kiện rẽ nhánh

```
        beq   $s0, $s1, L1
```

trở thành

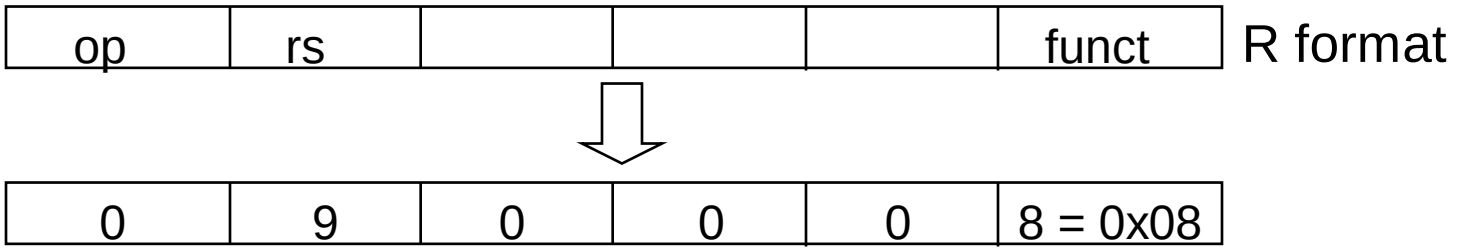
```
        bne   $s0, $s1, L2
        j     L1
L2:
```

Nhảy đến địa chỉ thay đổi

- Các ngôn ngữ bậc cao có các lệnh như `case` hay `switch` cho phép lựa chọn trong nhiều trường hợp phụ thuộc vào 1 biến
- Lệnh:

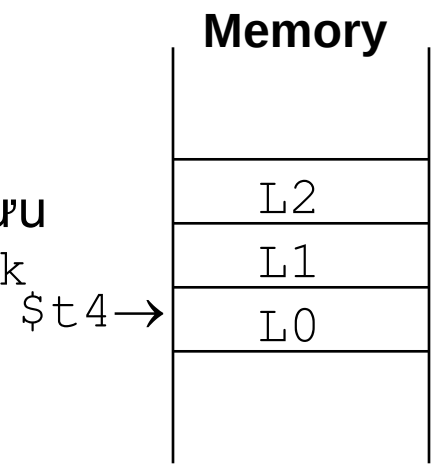
```
jr  $t1      #go to address in $t1
```

- Mã máy:



Lệnh case (switch) ở ngôn ngữ bậc cao

```
switch (k) {
   case 0:  h=i+j;  break; /*k=0*/
   case 1:  h=i+h;  break; /*k=1*/
   case 2:  h=i-j;  break; /*k=2*/
```



- Giả sử 3 từ liên tiếp trong bộ nhớ bắt đầu từ địa chỉ lưu trong `$t4` chứa giá trị của các nhãn L0, L1, và L2 và `k` lưu trong `$s2`

```
        add   $t1, $s2, $s2      #$t1 = 2*k
        add   $t1, $t1, $t1      #$t1 = 4*k
        add   $t1, $t1, $t4      #$t1 = addr of JumpT[k]
        lw    $t0, 0($t1)        #$t0 = JumpT[k]
        jr    $t0                #jump based on $t0
L0:     add   $s3, $s0, $s1      #k=0 so h=i+j
        j     Exit
L1:     add   $s3, $s0, $s3      #k=1 so h=i+h
        j     Exit
L2:     sub   $s3, $s0, $s1      #k=2 so h=i-j
Exit:   . . .
```

- Các từ lưu địa chỉ các nhãn như trên gọi là bảng địa chỉ nhảy *(jump address table*)
- Bảng này chứa dữ liệu nhưng thường nằm chung với đoạn mã chương trình

Gọi hàm hoặc thủ tục

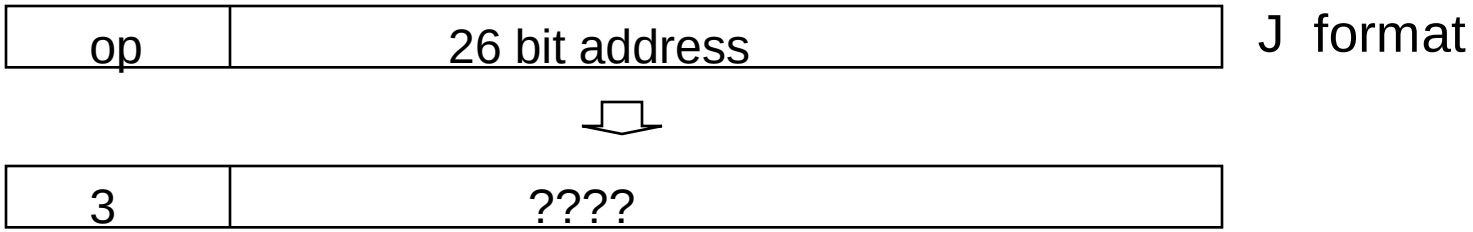
1. Hàm chính (hàm gọi, caller) đặt các tham số vào vị trị mà thủ tục (hàm bị gọi, callee) có thể truy cập
   - `$a0` - `$a3`: 4 thanh ghi tham số
2. Hàm gọi chuyển quyền điều khiển cho hàm bị gọi
3. Hàm bị gọi được cấp chỗ lưu trữ cần thiết
4. Hàm bị gọi thực hiện công việc mong muốn
5. Hàm bị gọi đặt kết quả vào vị trí hàm gọi có thể truy cập
   - `$v0` - `$v1`: 2 thanh ghi kết quả
6. Hàm bị gọi trả điều khiển cho hàm gọi
   - `$ra`: 1 thanh ghi địa chỉ trở về để quay về vị trí xuất phát

Lệnh để gọi 1 hàm

- MIPS procedure call instruction:

```
jal    ProcAddress  #jump and link
```

- Lưu PC+4 vào thanh ghi `$ra` như là đường dẫn đến lệnh kế tiếp khi trở về từ hàm
- Định dạng mã máy:

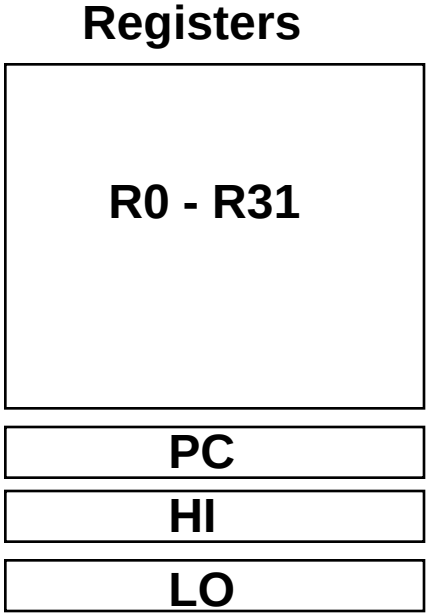


- Hàm sẽ trở về hàm gọi bằng:

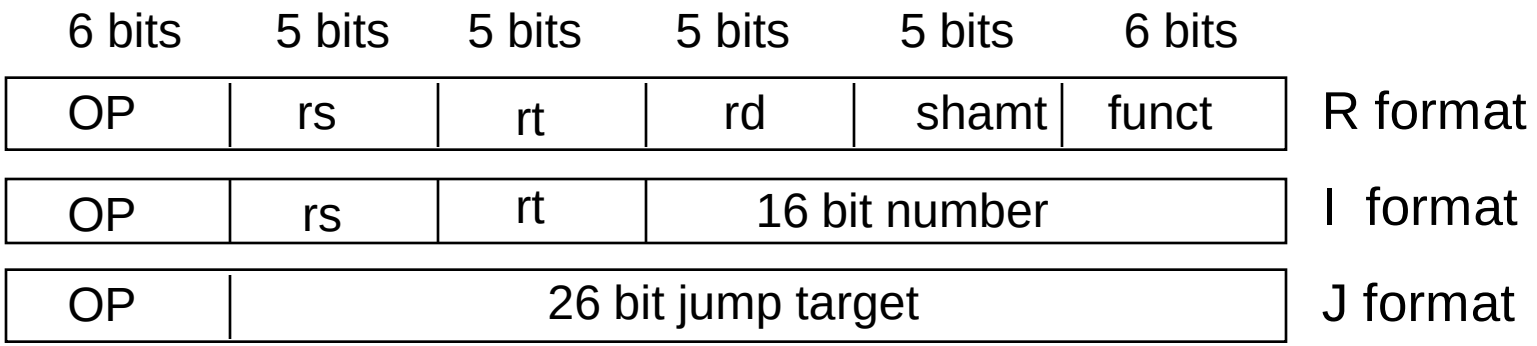
```
jr   $ra                #return
```

Tổng kết MIPS

- Các loại lệnh
  - Load/Store
  - Computational
  - Jump and Branch
  - Floating Point
    - coprocessor
  - Memory Management
  - Special



- 3 định dạng lệnh: độ rộng 32 bit



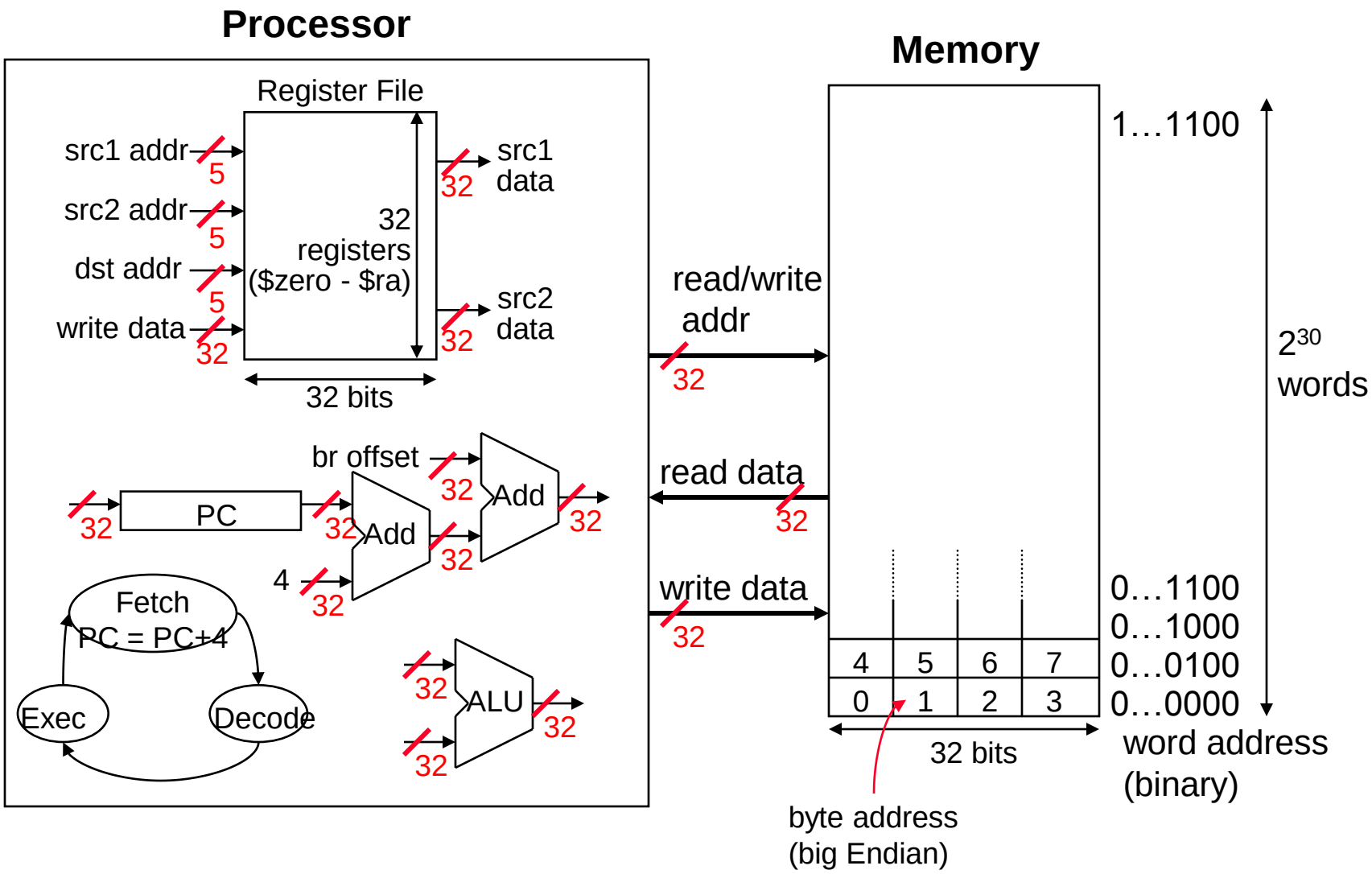
Tổng kết các lệnh MIPS

Category	Instr	OpC	Example	Meaning
Arithmetic (R & I format)	add	0 & 20	add $s1, $s2, $s3	$s1 = $s2 + $s3
	subtract	0 & 22	sub $s1, $s2, $s3	$s1 = $s2 - $s3
	add immediate	8	addi $s1, $s2, 4	$s1 = $s2 + 4
	shift left logical	0 & 00	sll $s1, $s2, 4	$s1 = $s2 << 4
	shift right logical	0 & 02	srl $s1, $s2, 4	$s1 = $s2 >> 4 (fill with zeros)
	shift right arithmetic	0 & 03	sra $s1, $s2, 4	$s1 = $s2 >> 4 (fill with sign bit)
	and	0 & 24	and $s1, $s2, $s3	$s1 = $s2 & $s3
	or	0 & 25	or $s1, $s2, $s3	$s1 = $s2 | $s3
	nor	0 & 27	nor $s1, $s2, $s3	$s1 = not ($s2 | $s3)
	and immediate	c	and $s1, $s2, ff00	$s1 = $s2 & 0xff00
	or immediate	d	or $s1, $s2, ff00	$s1 = $s2 | 0xff00
	load upper immediate	f	lui $s1, 0xffff	$s1 = 0xffff0000

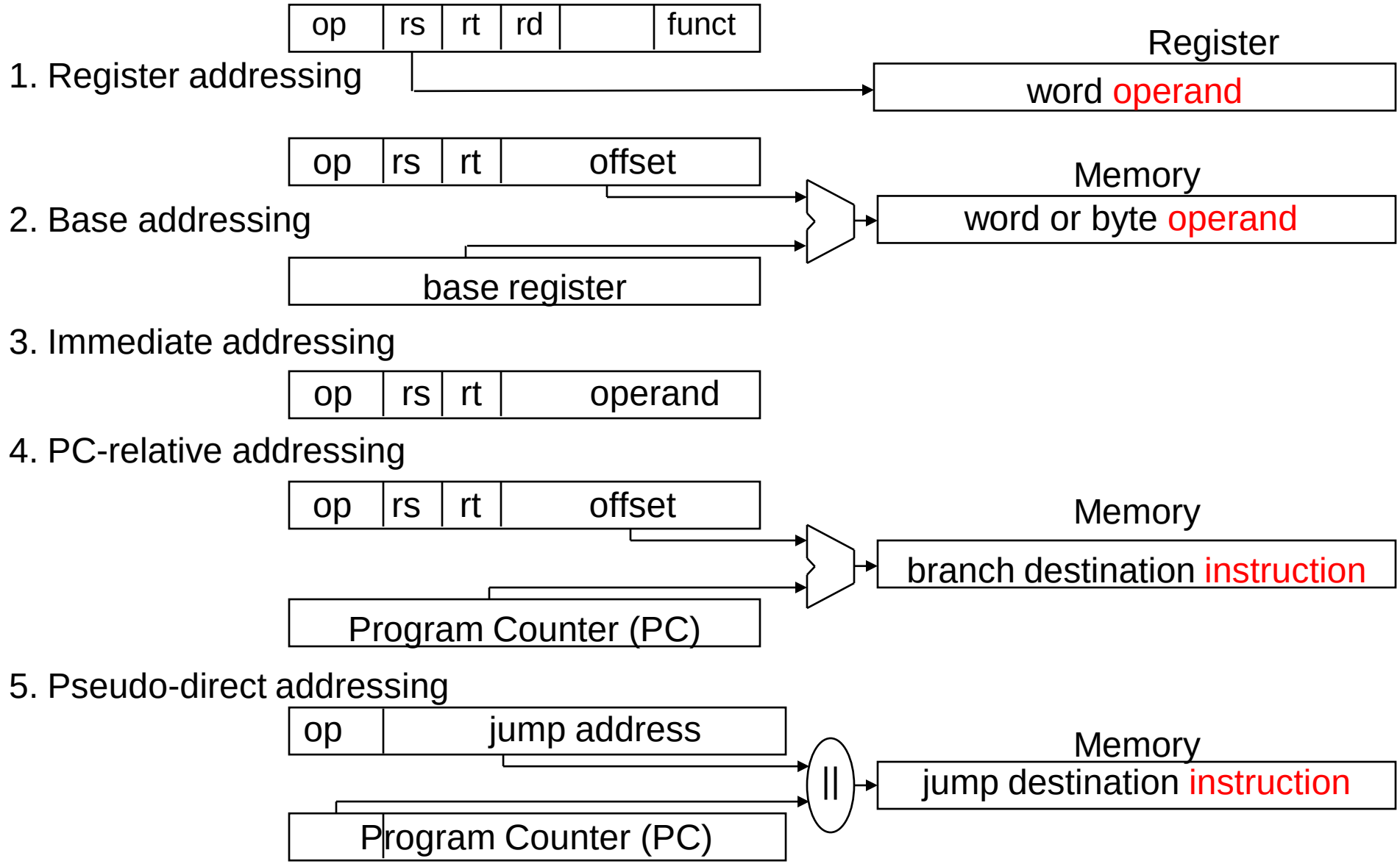
Tổng kết các lệnh MIPS

Category	Instr	OpC	Example	Meaning
Data transfer (I format)	load word	23	lw $s1, 100($s2)	$s1 = Memory($s2+100)
	store word	2b	sw $s1, 100($s2)	Memory($s2+100) = $s1
	load byte	20	lb $s1, 101($s2)	$s1 = Memory($s2+101)
	store byte	28	sb $s1, 101($s2)	Memory($s2+101) = $s1
	load half	21	lh $s1, 101($s2)	$s1 = Memory($s2+102)
	store half	29	sh $s1, 101($s2)	Memory($s2+102) = $s1
Cond. branch (I & R format)	br on equal	4	beq $s1, $s2, L	if ($s1==$s2) go to L
	br on not equal	5	bne $s1, $s2, L	if ($s1 !=$s2) go to L
	set on less than immediate	a	slti $s1, $s2, 100	if ($s2<100) $s1=1; else $s1=0
	set on less than	0 & 2a	slt $s1, $s2, $s3	if ($s2<$s3) $s1=1; else $s1=0
Uncond. jump	jump	2	j 2500	go to 10000
	jump register	0 & 08	jr $t1	go to $t1
	jump and link	3	jal 2500	go to 10000; $ra=PC+4

Tổ chức máy tính MIPS



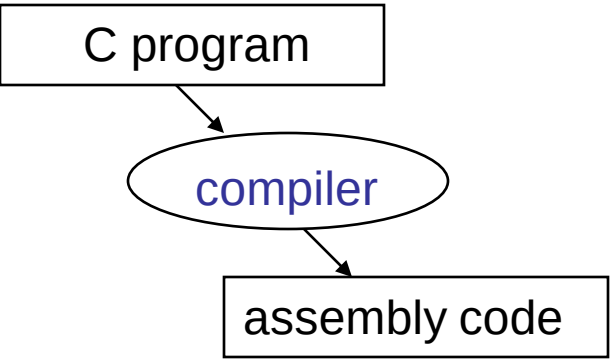
Chế độ địa chỉ MIPS



Nguyên tắc thiết kế RISC

- Simplicity favors regularity
  - fixed size instructions – 32-bits
  - small number of instruction formats
- Smaller is faster
  - limited instruction set
  - limited number of registers in register file
  - limited number of addressing modes
- Good design demands good compromises
  - three instruction formats
- Make the common case fast
  - arithmetic operands from the register file (load-store machine)
  - allow instructions to contain immediate operands

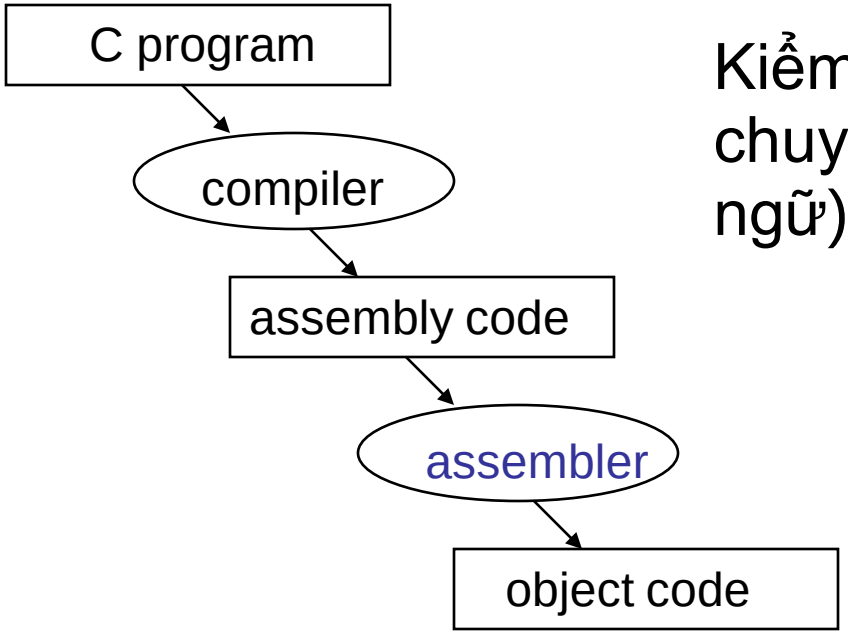
Biên dịch



Biến đổi chương trình C thành hợp ngữ

- Các ưu điểm của chương trình ngôn ngữ bậc cao
  - Số dòng mã ít hơn rất nhiều
  - dễ hiểu dễ biên dịch
  - …
- Ngày nay các trình biên dịch tối ưu có thể tạo ra mã hợp ngữ tốt như chuyên gia lập trình và thường tốt hơn khi dịch các chương trình lớn
  - Kích thước mã nhỏ hơn, tốc độ nhanh hơn

Assembler



Kiểm tra cú pháp mã hợp ngữ và chuyển đổi mã biểu tượng (mã hợp ngữ) thành mã đối tượng (mã máy).

- Ưu điểm của assembler
  - Dễ nhớ
  - Sử dụng nhãn địa chỉ - giá trị địa chỉ được tính bởi assembler
  - Sử dụng chỉ thị giả
    - VD., “move $t0, $t1” chỉ có trong assembler và sẽ được chuyển thành chỉ thị “add $t0,$t1,$zero”)

- Chú ý: Khi xác định hiệu năng cần đếm số chỉ thị được thực thi chứ không phải kích thước mã chương trình.

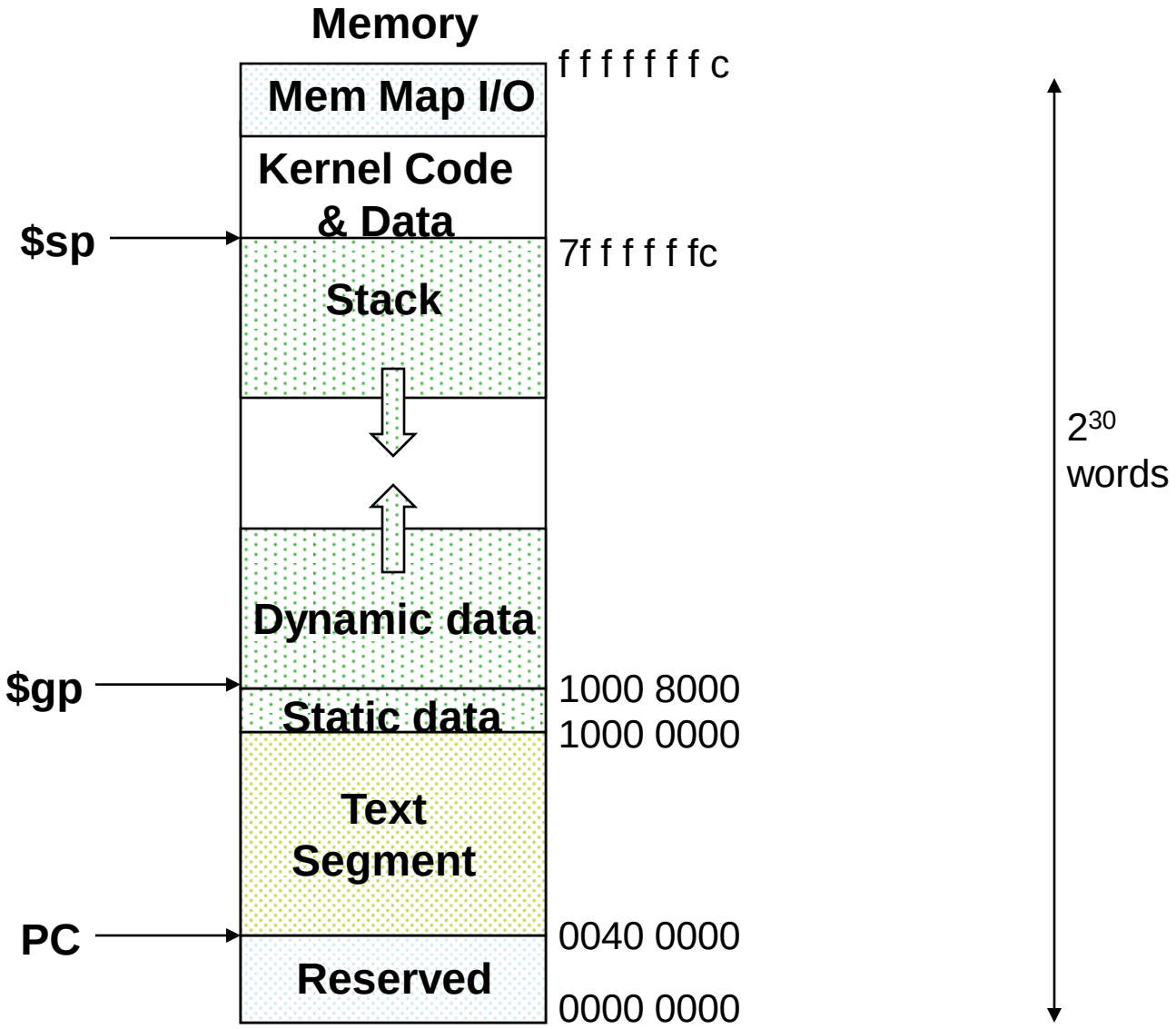
Nhiệm vụ chính của assembler

1. Tạo bảng biểu tượng (*symbol table*) chứa tên biểu tượng (nhãn) và địa chỉ tương ứng
   - 1 biểu tượng địa phương được sử dụng trong tệp nó được định nghĩa. Biểu tượng được quy ước mặc định là địa phương.
   - 1 biểu tượng toàn cục (ngoại) *tham chiếu/được tham chiếu* đến mã hoặc dữ liệu ở 1 tệp. Các biểu tượng toàn cục được khai báo rõ ràng là toàn cục (VD., `.globl main`)
2. Dịch các lệnh ở mã hợp ngữ thành ngôn ngữ máy bằng cách "lắp ghép" các giá trị số tương ứng với mã lệnh (*opcode*), chỉ số thanh ghi (*register specifiers*), số bít dịch (shift amounts), và độ lệch các lệnh jump/branch.

Các nhiệm vụ khác của Assembler

- Thay mã giả lệnh bằng mã hợp ngữ hợp lệ
  - Thanh ghi `$at` được dành riêng cho assembler để làm việc này
- Thay lệnh rẽ nhánh xa bằng 1 lệnh rẽ nhánh gần theo sau bởi 1 lệnh nhảy
- Thay lệnh với giá trị tức thời lớn bằng lệnh `lui` theo sau bởi 1 lệnh `ori`
- Đổi các số ở dạng thập phân và hệ 16 thành các số ở dạng nhị phân và ký tự thành mã ASCII tương ứng.
- Xử lý các dẫn hướng sắp xếp dữ liệu (e.g., `.asciiz`)
- Triển khai các macro thành các chuỗi chỉ thị

Sơ đồ bộ nhớ MIPS



Cấu trúc 1 tệp mã máy

- Object file header: kích thước và vị trí các phần sau trong tệp
- Text (code) segment (`.text`) : mã máy
- Data segment (`.data`) : dữ liệu đi kèm với mã
  - Dữ liệu tĩnh (*static data*) – được cấp phát trong toàn bộ quá trình chạy
  - Dữ liệu động (*dynamic data*) – cấp phát khi cần thiết
- Relocation information: xác định các lệnh (dữ liệu) sử dụng (nằm tại vị trị) địa chỉ tuyệt đối – *không* liên quan đến 1 thanh ghi (kể cả PC)
  - Trên MIPS các lệnh `j`, `jal`, và 1 số lệnh đọc ghi (VD., `lw $t1, 100($zero)`) sử dụng địa chỉ tuyệt đối
- Symbol table: các nhã toàn cục cùng với địa chỉ (nếu được định nghĩa cùng trong đoạn mã) hoặc không cùng địa chỉ (nếu được định nghĩa ngoài đoạn mã)
- Debugging information

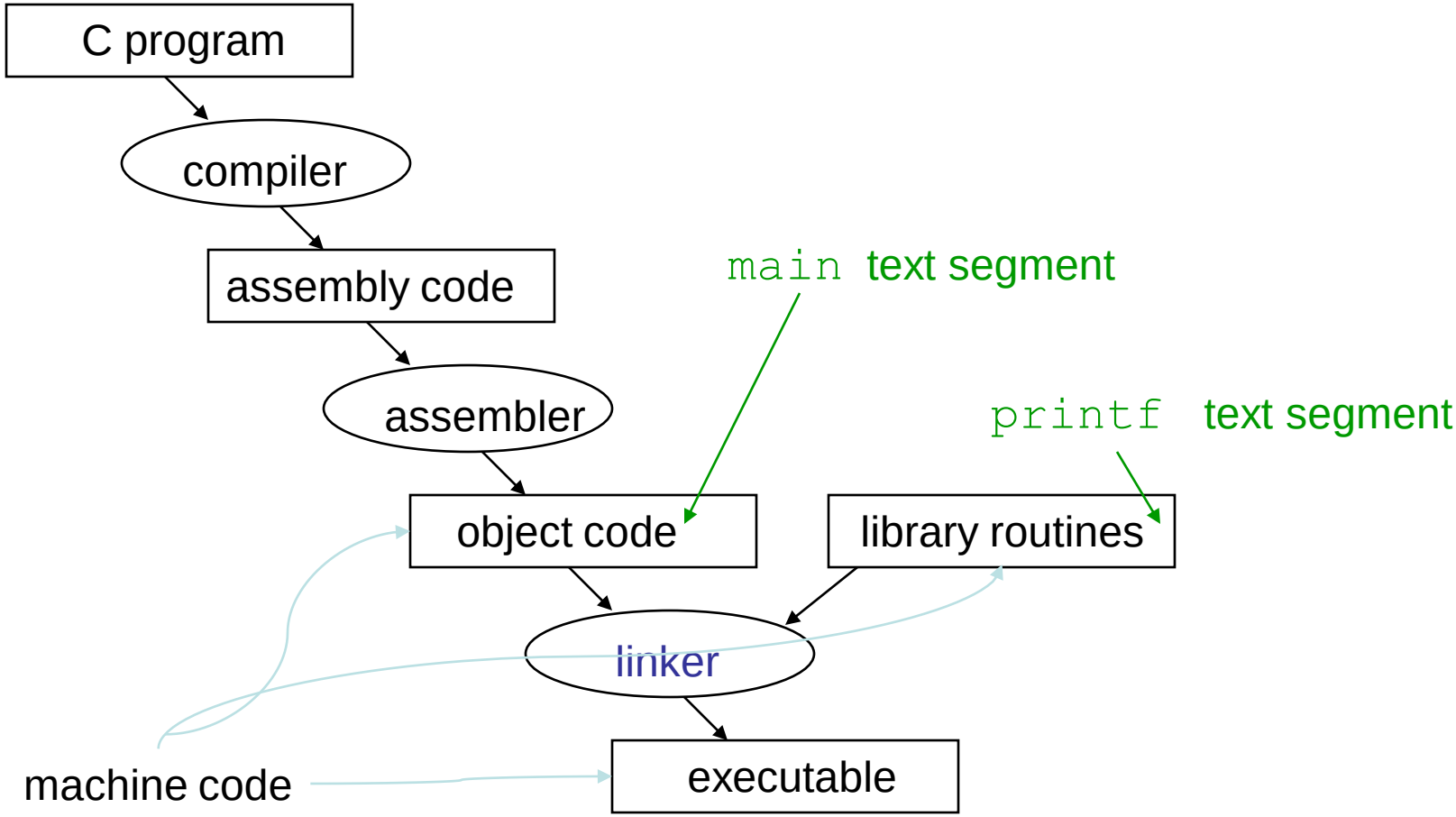
Ví dụ 2.8. Cấu trúc tệp mã máy

Gbl?	Symbol	Address
	str	1000 0000
	cr	1000 000b
yes	**main**	**0040 0000**
	loop	0040 000c
	brnc	0040 001c
	done	0040 0024
yes	**printf**	**???? ????**

Relocation Info	
Address	Data/Instr
1000 0000	str
1000 000b	cr
0040 0018	j loop
0040 0020	j loop
0040 0024	jal printf

```
        .data
        .align 0
str:    .asciiz "The answer is "
cr:     .asciiz "\n"
        .text
        .align 2
        .globl main
        .globl printf
main:   ori     $2, $0, 5
        syscall
        move    $8, $2
loop:   beq     $8, $9, done
        blt     $8, $9, brnc
        sub     $8, $8, $9
        j       loop
brnc:   sub     $9, $9, $8
        j       loop
done:   jal     printf
```

Liên kết (*linker*)



Liên kết

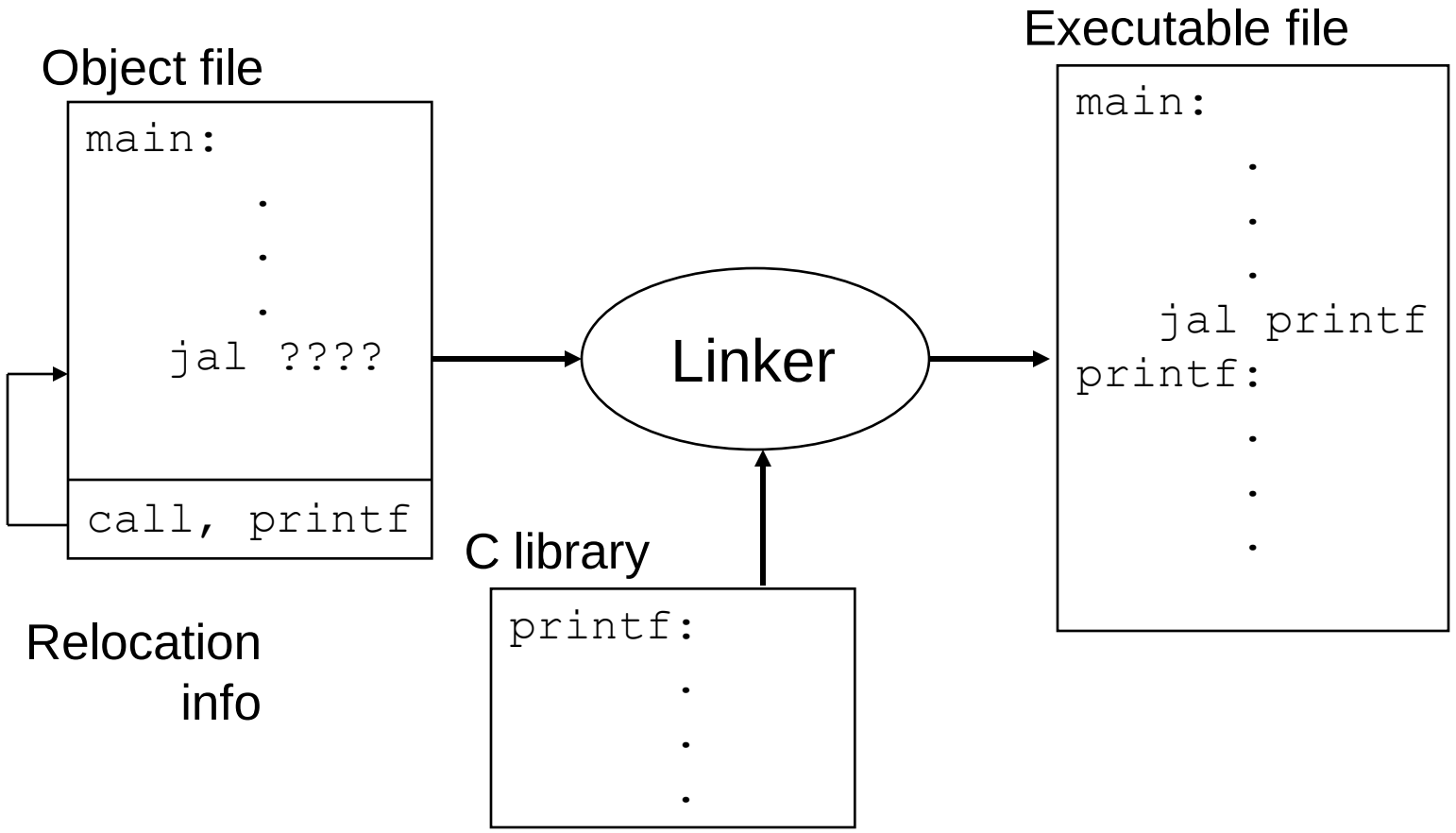
Liên kết các đoạn mã máy độc lập với nhau

- Chỉ cần biên dịch và assemble lại các đoạn mã có thay đổi: nhanh hơn

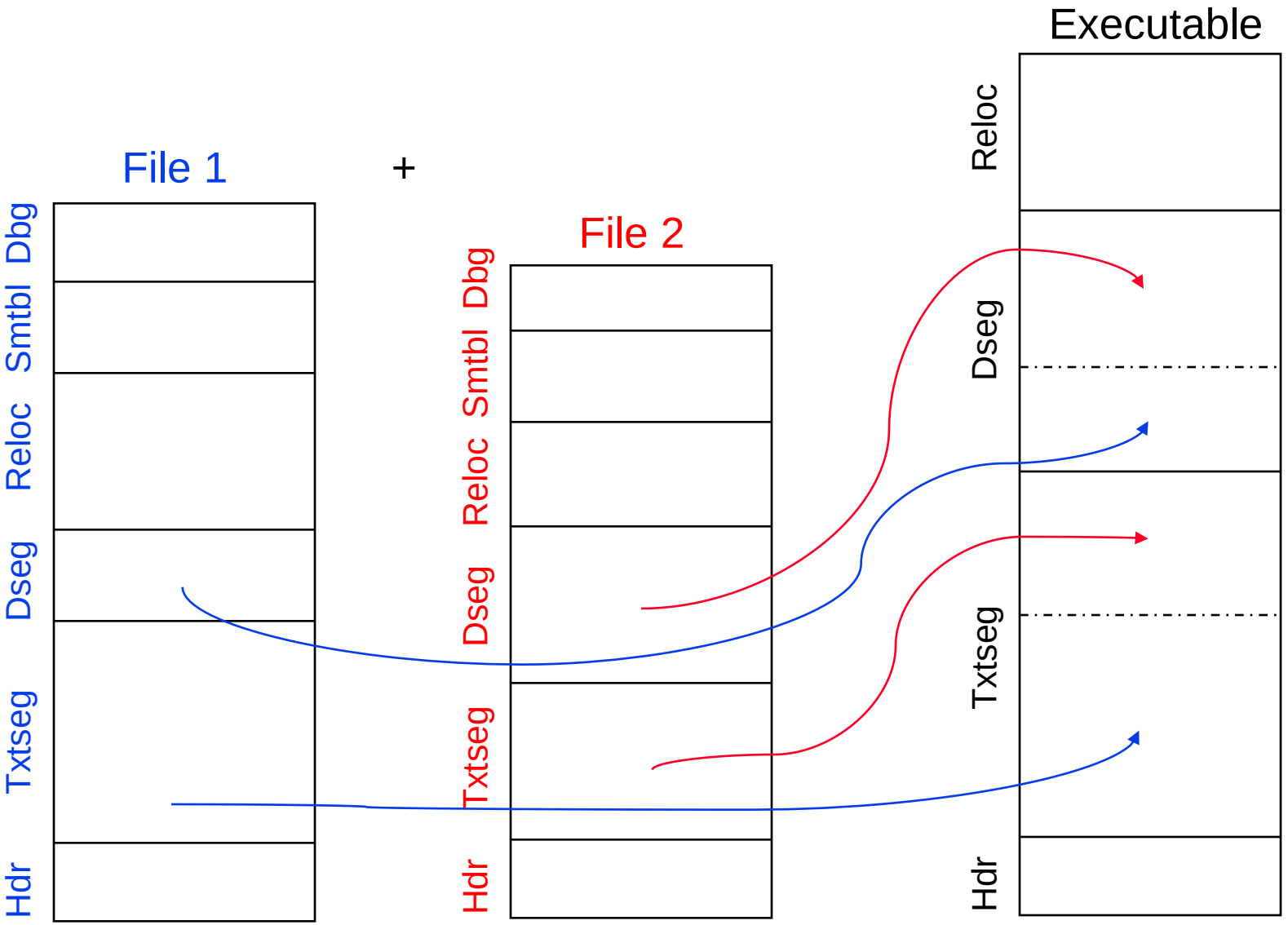
1. Quyết định mẫu cấp phát bộ nhớ cho đoạn mã và đoạn dữ liệu của từng mô đun.
   - Chú ý: Các mô đun được assemble độc lập, **mỗi mô đun** đều có đoạn mã bắt đầu tại 0x0040 0000 và dữ liệu tĩnh bắt đầu tại 0x1000 0000
2. Cấp phát lại địa chỉ tuyệt đối để phản ánh đúng địa chỉ bắt đầu của đoạn mã và đoạn dữ liệu
3. Sử dụng bảng biểu tượng để xác định các nhãn chưa được xác định
   - Các địa chỉ dữ liệu, rẽ nhánh nhảy tới các mô đun ngoài

- Linker tạo ra tệp thực hiện được

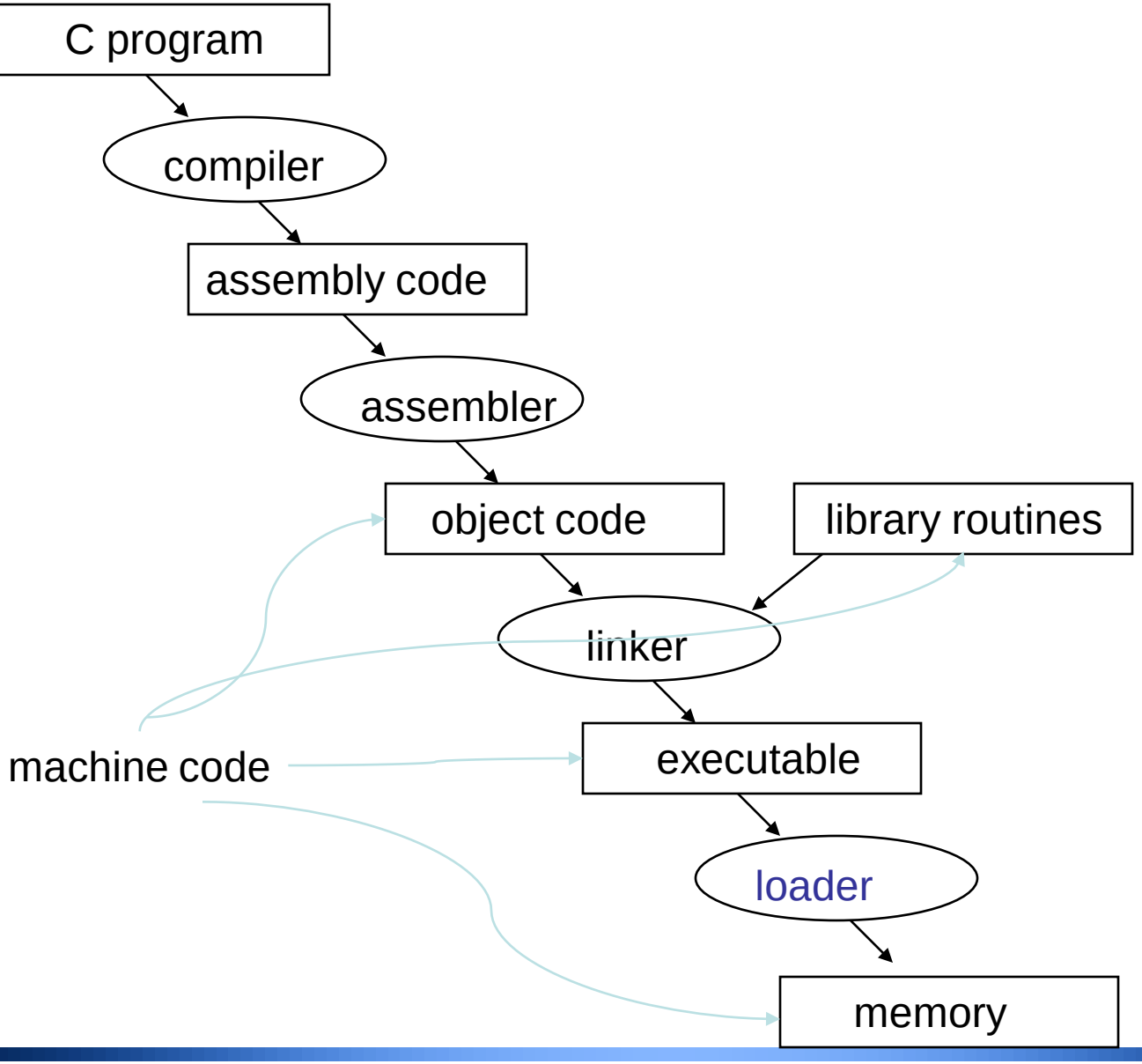
Liên kết


Object file
main:
.
.
.
jal ????
call, printf
Relocation info
C library
printf:
.
.
.
Linker
Executable file
main:
.
.
.
jal printf
printf:
.
.
.

Liên kết 2 tệp mã lệnh



Nạp chương trình



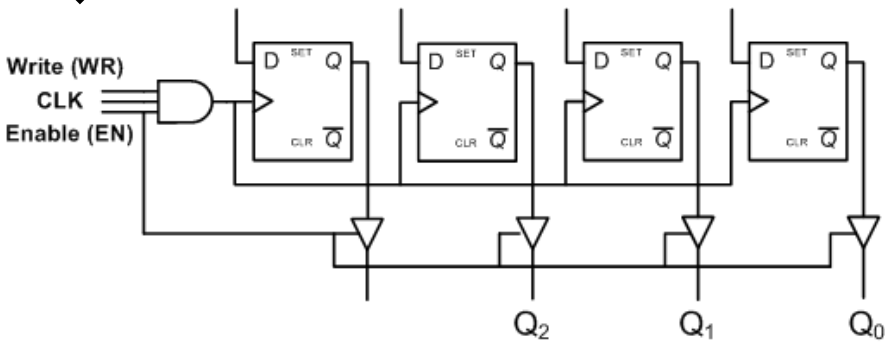
Nạp chương trình

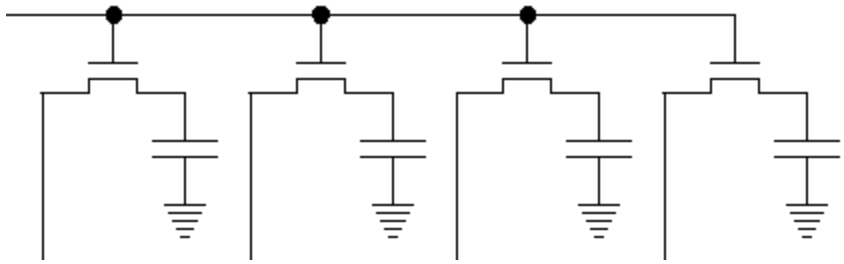
- Nạp (sao chép) mã thực hiện từ đĩa vào bộ nhớ tại địa chỉ bắt đầu được xác định bởi hệ điều hành
- Sao chép các tham số (nếu có) của hàm chính vào ngăn xếp
- Khởi tạo các thanh ghi và đặt con trỏ ngăn xếp vào vị trí trống (0x7fff fffc)
- Nhảy đến hàm khởi tạo (tại PC = 0x0040 0000). Hàm khởi tạo sẽ chép các tham số vào thanh ghi tham số và gọi hàm chính bằng lệnh `jal main`

Phép toán và cách thực hiện

- Phép toán dịch
- Phép toán số học
  - Cộng, trừ
  - Nhân
  - Chia
- Phép toán dấu phẩy động

Dữ liệu máy tính: Vector bit

- Lưu trữ trong thanh ghi hoặc từ nhớ
- 


- Truyền dẫn trên đường bus
- Xử lý bằng phép toán
  - Phép toán dịch
  - Kiểm tra 1 bit, đặt 1 bit, xóa 1 bit
  - Sao chép các bit
- Hiện tượng tràn

Phép toán dịch

❖ **Dịch logic**

- ❑ Các chữ số trống được điền bằng 0
- ❑ Sang phải1 bit: $srl_1(a_{n-1},a_{n-2},\ldots,a_0) = (0,a_{n-1},a_{n-2},\ldots,a_1)$
- ❑ Sang trái 1 bit: $sll_1(a_{n-1},a_{n-2},\ldots,a_0) = (a_{n-2},a_{n-3},\ldots,a_0,0)$
- ❑ Dùng để triển khai bộ nhân và chia không dấu

0	a_{n-2}	…	a_1	a_0
a_{n-1}	a_{n-2}	…	a_1	a_0

❖ **Dịch số học**

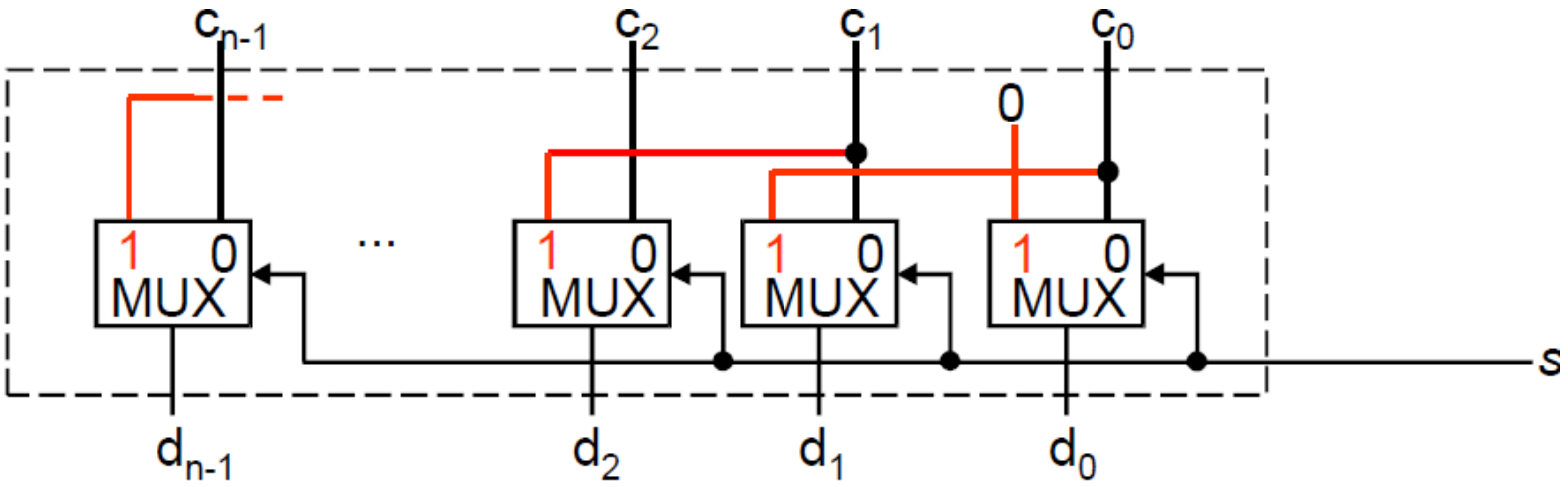
- ❑ Bít dấu (MSB) không được thay đổi
  - dịch phải sao chép bít dấu vào các chữ số trống
  - dịch trái không dịch bít dấu
- ❑ Sang phải 1 bit: $sra_1(a_{n-1},a_{n-2},\ldots,a_0) = (a_{n-1},a_{n-1},a_{n-2},\ldots,a_1)$
- ❑ Sang trái 1 bit: $sla_1(a_{n-1},a_{n-2},\ldots,a_0) = (a_{n-1},a_{n-3},\ldots,a_0,0)$
- ❑ Dùng để triển khai bộ nhân và chia có dấu

a_{n-1}	a_{n-2}	…	a_1	a_0
a_{n-1}	a_{n-2}	a_{n-3}	…	a_0

❖ Kết quả sai và xảy ra hiện tượng tràn nếu: $a_{n-1} \neq a_{n-2}$

Bộ dịch

- Dịch trái 0 hoặc 1 bít

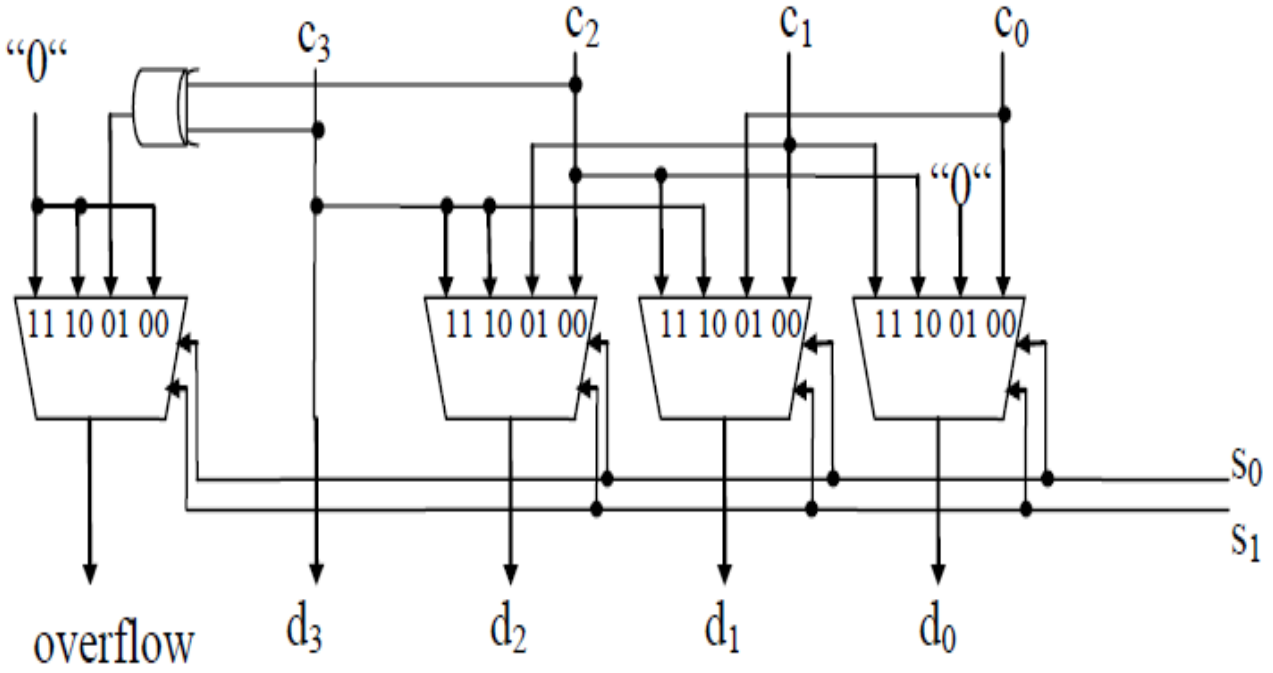


- Có thể thiết kế bộ dịch cả trái và phải

Bộ dịch

❖Bộ dịch trái 1 bít, và dịch phải 2 bít

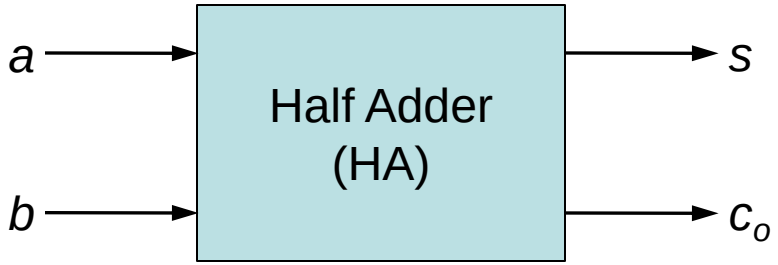
shift by	s_1	s_0
1	0	1
0	0	0
-1	1	1
-2	1	0



Bộ cộng nửa, cộng 2 số 1 bit

Tín hiệu vào: *a*, *b*

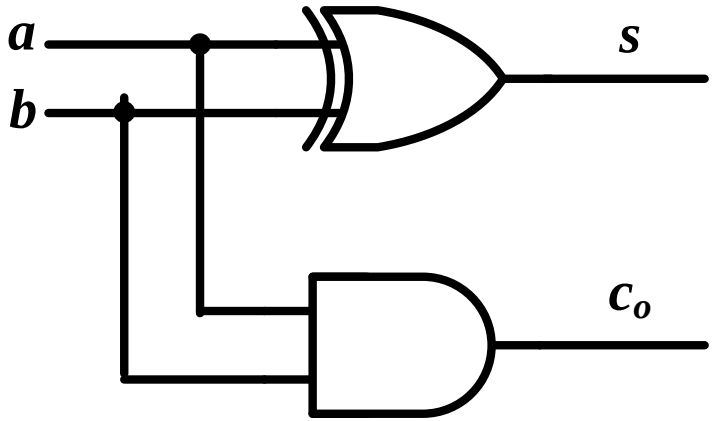
Tín hiệu ra: *s* (sum), c_o (carry out)



a	*b*	*s*	c_o
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Câu hỏi:

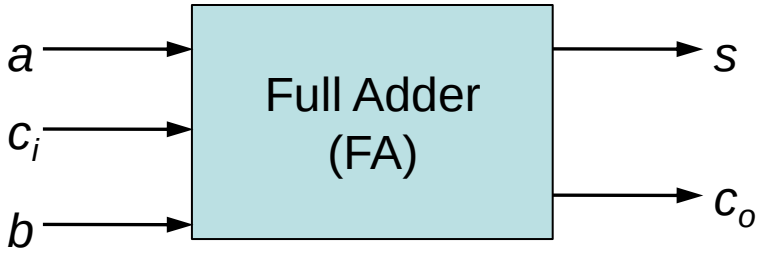
Xác định biểu thức Bool cho *s*, và c_o



Bộ cộng đầy đủ, cộng 3 số 1 bit

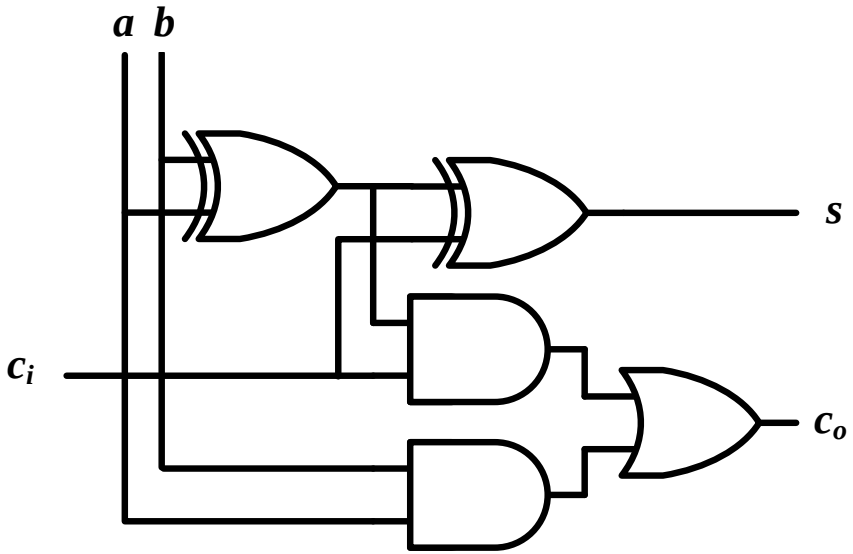
Tín hiệu vào: a, b, c_i (carry in)

Tín hiệu ra: s (sum), c_o (carry out)



Câu hỏi:

Xác định biểu thức Bool cho s, và c_o



a	b	c_i	s	c_o
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Phép cộng, trừ 2 số có dấu

- 2 số biểu diễn ở dạng mã bù 2.
- Cộng từng bit từ bit LSB đến bit MSB; Nhớ sang cột kế tiếp

	0	1	1	1	
5		0	1	0	1
3		0	0	1	1
-8		1	0	0	0

Tràn

	1	0	0	0	
-7		1	0	0	1
-2		1	1	1	0
7		0	1	1	1

Tràn

	0	0	0	0	
5		0	1	0	1
2		0	0	1	0
7		0	1	1	1

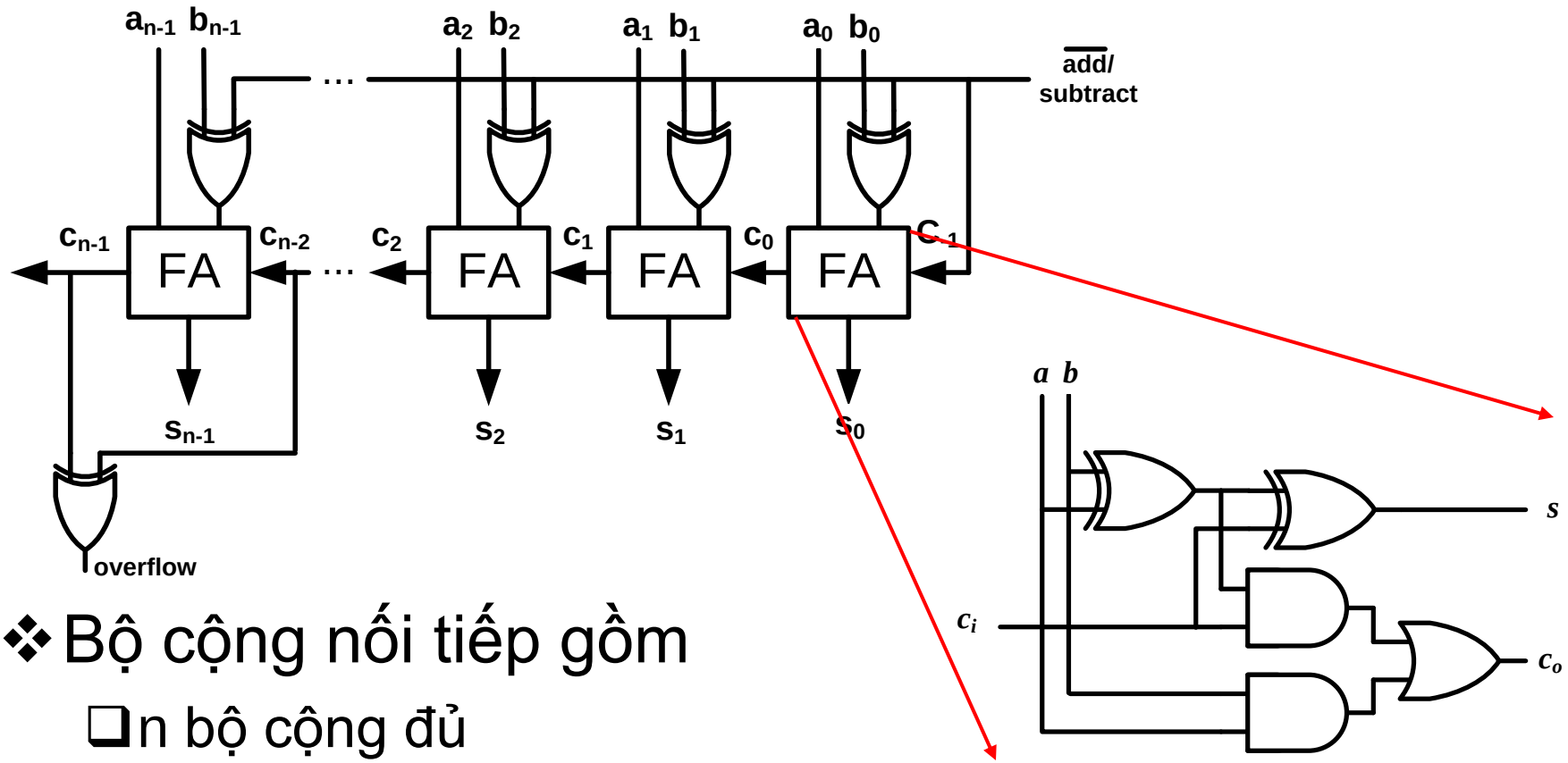
Không tràn

	1	1	1	1	
-3		1	1	0	1
-5		1	0	1	1
-8		1	0	0	0

Không tràn

- Kết quả sai và tràn xảy ra khi 2 bit nhớ cuối cùng khác nhau: $c_{o,3} \neq c_{i,3}$
- Cộng 2 số khác dấu luôn không xảy ra tràn
- Phép trừ là phép cộng với số đảo (dùng mã bù 2)

Bộ cộng 2 số n bít dạng bù 2



- Bộ cộng nối tiếp gồm
  - n bộ cộng đủ
  - n cổng logic xor để tính số đảo khi trừ
  - Cổng logic xor để phát hiện tràn

Tốc độ bộ cộng

- Bộ cộng nối tiếp
  - Tín hiệu nhớ lan truyền (*ripples*) qua tất cả các bộ cộng "*ripple carry adder*"
  - Độ trễ tăng tuyến tính với số lượng bộ cộng (số bit của mỗi toán hạng)
    - ✓ Bít nhớ: $t_{ar}(c_n) = t_{ar}(c_{n-1}) + 2 = 3 + 2*(n-1)$
    - ✓ Bít tổng: $t_{ar}(s_n) = t_{ar}(c_n) + 1 = 4 + 2*(n-1)$
- Tăng tốc bằng bộ cộng tính bit nhớ trước (*Carry lookahead Adder*)

Bộ cộng CLA

- Với Ripple-Carry Adder, bit nhớ được tính dựa trên các bít nhớ trước đó ➔ tốc độ chậm
- Tăng tốc độ, tính bit nhớ ở mỗi cột trực tiếp từ tín hiệu đầu vào

$$s_i = a_i \oplus b_i \oplus c_{i-1}$$

$$\begin{aligned} c_i &= a_i b_i + a_i c_{i-1} + b_i c_{i-1} \\ &= a_i b_i + c_{i-1}(a_i + b_i) \\ &= a_i b_i + c_{i-1}(a_i \oplus b_i) \end{aligned}$$

Tín hiệu tạo nhớ: $g_i = a_i b_i$
Tạo ra c_i khi $a_i = b_i = 1$

Lan truyền nhớ: $p_i = (a_i \oplus b_i)$
Truyền nhớ từ đầu vào đến đầu ra khi $a_i \oplus b_i = 1$

- Bit nhớ đầu ra của cột *i* được tính từ tín hiệu tạo nhớ và tín hiệu lan truyền nhớ $c_i = g_i + c_{i-1}\, p_i$

Bộ cộng CLA

- Tính toán bit nhớ

$$c_0 = g_0 + p_0 c_{-1}$$

$$c_1 = g_1 + p_1 c_0 = g_1 + p_1 g_0 + p_1 p_0 c_{-1}$$

$$c_2 = g_2 + p_2 c_1 = g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_{-1}$$

$$c_3 = g_3 + p_3 c_2 = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0 + p_3 p_2 p_1 p_0 c_{-1}$$

- Mỗi công thức nhớ trên có thể được triển bởi một mạch logic 2 mức
- Để tính toán p_i, g_i ta cần mạch logic 1 mức từ đầu vào
- Vậy cần tối đa 3 mức từ đầu vào để tính được tín hiệu nhớ

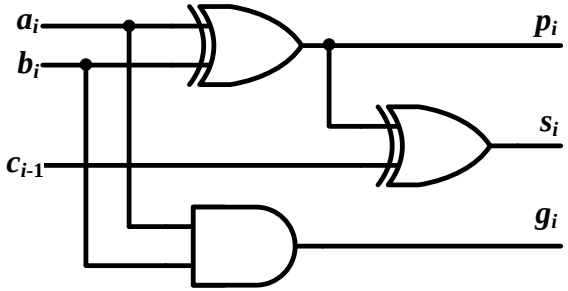
➔Tăng tốc độ

➔Cần cổng AND n+2 đầu vào cho c_n!

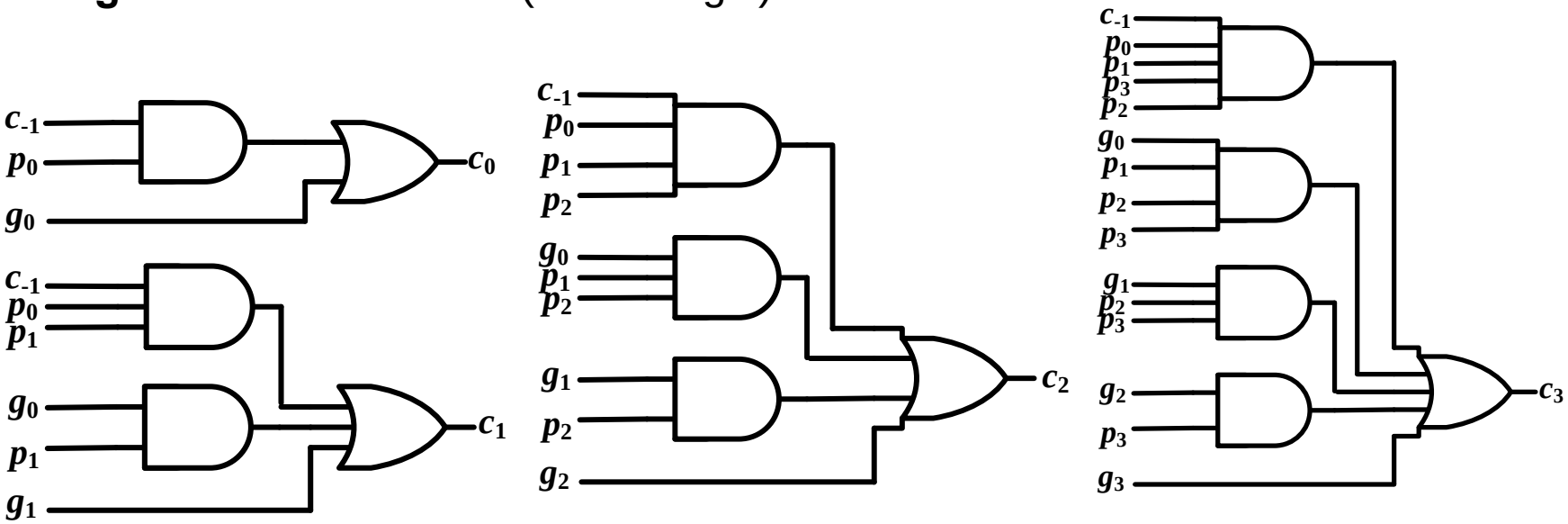
Bộ cộng CLA

Gồm 2 tầng

Tầng 1: Tính toán tổng, tính hiệu tạo nhớ và truyền nhớ (1 mức logic) - PFA



Tầng 2: Tính toán bit nhớ (2 mức logic) - CLA



Phép nhân không dấu

- Nhân lần lượt các cột của số bị nhân và số nhân được tích cục bộ
- Các tích cục bộ được cộng với nhau theo cột

a_3	a_2	a_1	a_0	*	b_3	b_2	b_1	b_0
					a_3b_0	a_2b_0	a_1b_0	a_0b_0
+				a_3b_1	a_2b_1	a_1b_1	a_0b_1	
+			a_3b_2	a_2b_2	a_1b_2	a_0b_2		
+		a_3b_3	a_2b_3	a_1b_3	a_0b_3			
	s_7	s_6	s_5	s_4	s_3	s_2	s_1	s_0

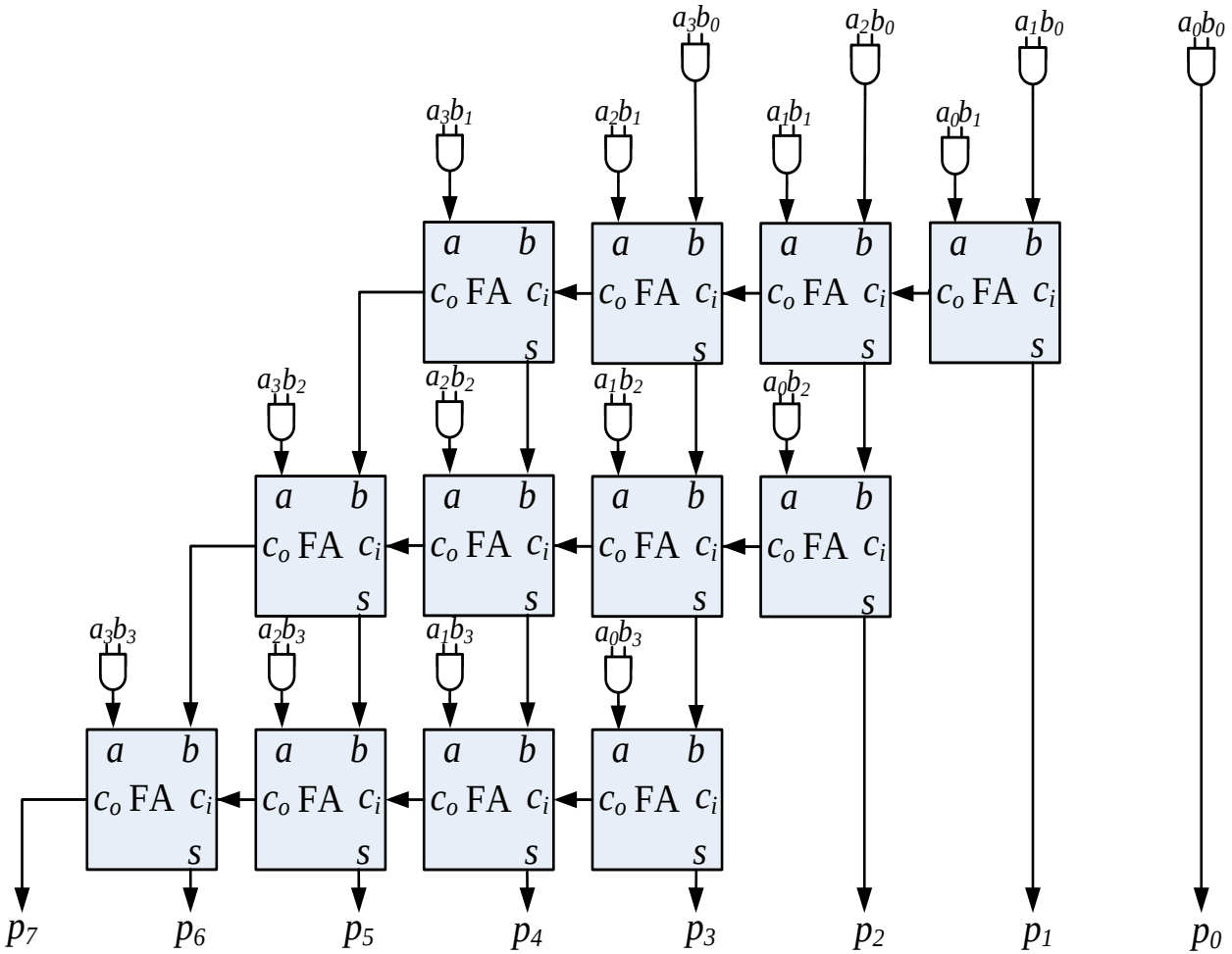
a	b	a*b
0	0	0
0	1	0
1	0	0
1	1	1

- Ví dụ

1	0	1	1	*	0	0	1	1	11*3
					1	0	1	1	
+				1	0	1	1		
+			0	0	0	0			
+		0	0	0	0				
	0	0	1	0	0	0	0	1	33

Bộ nhân không dấu song song

- Sử dụng logic tổ hợp



Phép nhân có dấu

- Mở rộng bít dấu cho các tích cục bộ
- Với tích cục bộ của bit dấu số b_3, cần lấy số đối (số bù 2)

a_3	a_2	a_1	a_0	*	b_3	b_2	b_1	b_0
	a_3b_0	a_3b_0	a_3b_0	a_3b_0	a_3b_0	a_2b_0	a_1b_0	a_0b_0
+	a_3b_1	a_3b_1	a_3b_1	a_3b_1	a_2b_1	a_1b_1	a_0b_1	
+	a_3b_2	a_3b_2	a_3b_2	a_2b_2	a_1b_2	a_0b_2		
+	$\overline{a_3b_3}$	$\overline{a_3b_3}$	$\overline{a_2b_3}$	$\overline{a_1b_3}$	$\overline{a_0b_3}$			
+					1			
	p_7	p_6	p_5	p_4	p_3	p_2	p_1	p_0

Ví dụ 2.9 – Phép nhân có dấu

1	0	1	1	*	0	0	1	1	-5*3
	1	1	1	1	1	0	1	1	
+	1	1	1	1	0	1	1		
+	0	0	0	0	0	0			
+	1	1	1	1	1				
					1				
						1			
					1				
			1	0					
			1						
	1	0							
	1								
	1	1	1	1	0	0	0	1	-15

Phép nhân có dấu

❖ Đơn giản hóa

a_3	a_2	a_1	a_0	*	b_3	b_2	b_1	b_0
	a_3b_0	a_3b_0	a_3b_0	a_3b_0	a_3b_0	a_2b_0	a_1b_0	a_0b_0
+	a_3b_1	a_3b_1	a_3b_1	a_3b_1	a_2b_1	a_1b_1	a_0b_1	
+	a_3b_2	a_3b_2	a_3b_2	a_2b_2	a_1b_2	a_0b_2		
+	$\overline{a_3b_3}$	$\overline{a_3b_3}$	$\overline{a_2b_3}$	$\overline{a_1b_3}$	$\overline{a_0b_3}$			
					1			
	p_7	p_6	p_5	p_4	p_3	p_2	p_1	p_0

a_3	a_2	a_1	a_0	*	b_3	b_2	b_1	b_0
						a_2b_0	a_1b_0	a_0b_0
+					a_2b_1	a_1b_1	a_0b_1	
+				a_2b_2	a_1b_2	a_0b_2		
+			$\overline{a_2b_3}$	$\overline{a_1b_3}$	$\overline{a_0b_3}$			
+					1			
-					a_3b_0			
-				a_3b_1				
-			a_3b_2					
-		$\overline{a_3b_3}$						
	p_7	p_6	p_5	p_4	p_3	p_2	p_1	p_0

Phép nhân có dấu

❖ Đơn giản hóa

a_3	a_2	a_1	a_0	*	b_3	b_2	b_1	b_0
						a_2b_0	a_1b_0	a_0b_0
+					a_2b_1	a_1b_1	a_0b_1	
+				a_2b_2	a_1b_2	a_0b_2		
+			$\overline{a_2b_3}$	$\overline{a_1b_3}$	$\overline{a_0b_3}$			
+					1			
-		$\overline{a_3b_3}$	a_3b_2	a_3b_1	a_3b_0			
	p_7	p_6	p_5	p_4	p_3	p_2	p_1	p_0

a_3	a_2	a_1	a_0	*	b_3	b_2	b_1	b_0
						a_2b_0	a_1b_0	a_0b_0
+					a_2b_1	a_1b_1	a_0b_1	
+				a_2b_2	a_1b_2	a_0b_2		
+			$\overline{a_2b_3}$	$\overline{a_1b_3}$	$\overline{a_0b_3}$			
+					1			
-					a_3b_0			
-				a_3b_1				
-			a_3b_2					
-		$\overline{a_3b_3}$						
	p_7	p_6	p_5	p_4	p_3	p_2	p_1	p_0

Phép nhân có dấu

❖ Đơn giản hóa

a_3	a_2	a_1	a_0	*	b_3	b_2	b_1	b_0
						a_2b_0	a_1b_0	a_0b_0
+					a_2b_1	a_1b_1	a_0b_1	
+				a_2b_2	a_1b_2	a_0b_2		
+			$\overline{a_2b_3}$	$\overline{a_1b_3}$	$\overline{a_0b_3}$			
+					1			
-	0	$\overline{a_3b_3}$	a_3b_2	a_3b_1	a_3b_0			
	p_7	p_6	p_5	p_4	p_3	p_2	p_1	p_0

a_3	a_2	a_1	a_0	*	b_3	b_2	b_1	b_0
						a_2b_0	a_1b_0	a_0b_0
+					a_2b_1	a_1b_1	a_0b_1	
+				a_2b_2	a_1b_2	a_0b_2		
+			$\overline{a_2b_3}$	$\overline{a_1b_3}$	$\overline{a_0b_3}$			
+					1			
+	1	a_3b_3	$\overline{a_3b_2}$	$\overline{a_3b_1}$	$\overline{a_3b_0}$			
					1			
	p_7	p_6	p_5	p_4	p_3	p_2	p_1	p_0

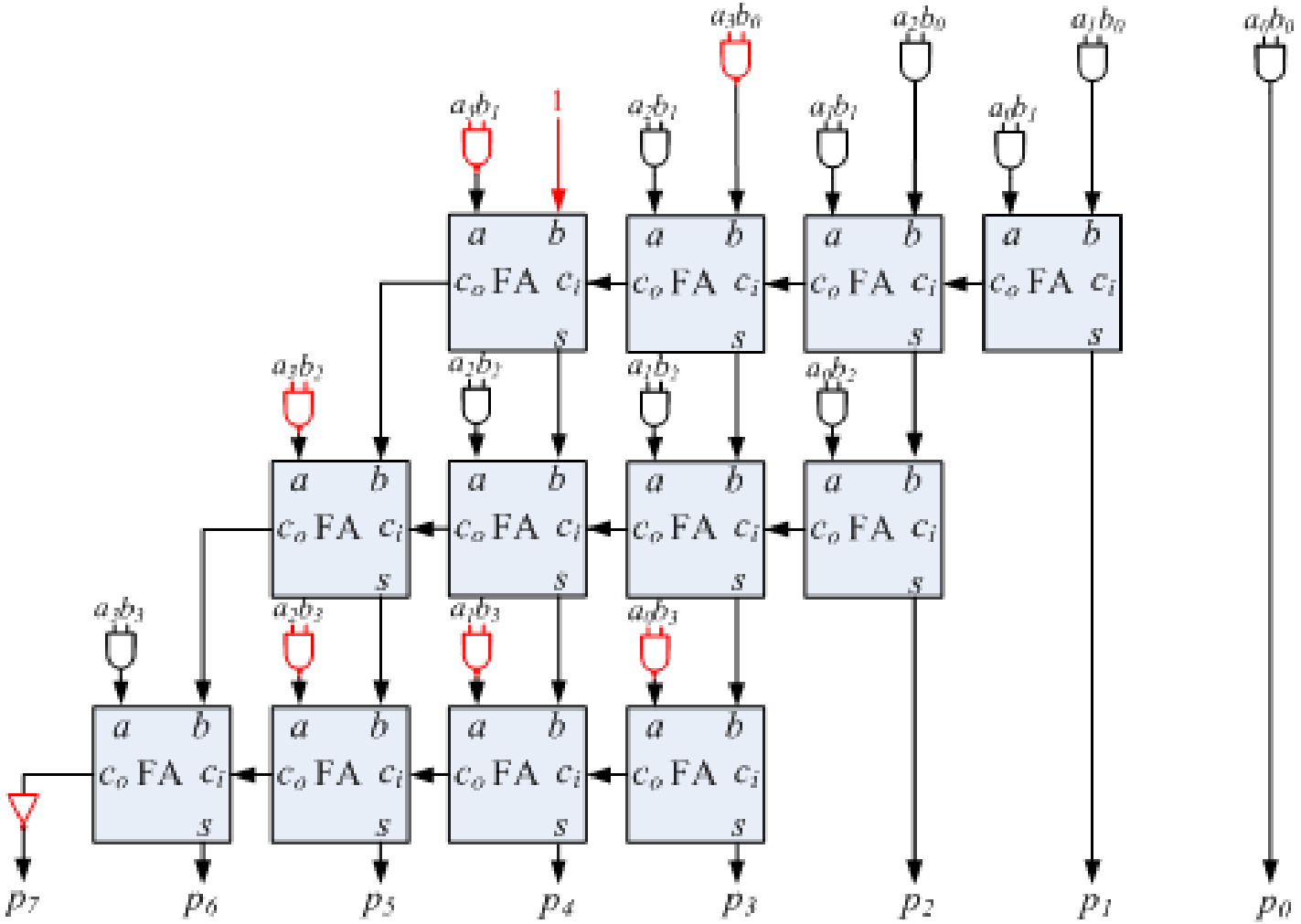
Phép nhân có dấu

❖ Đơn giản hóa

a_3	a_2	a_1	a_0	*	b_3	b_2	b_1	b_0
						a_2b_0	a_1b_0	a_0b_0
+					a_2b_1	a_1b_1	a_0b_1	
+				a_2b_2	a_1b_2	a_0b_2		
+			a_2b_3	a_1b_3	a_0b_3			
+					1			
+	1	a_3b_3	a_3b_2	a_3b_1	a_3b_0			
					1			
	p_7	p_6	p_5	p_4	p_3	p_2	p_1	P_0

a_3	a_2	a_1	a_0	*	b_3	b_2	b_1	b_0
					a_3b_0	a_2b_0	a_1b_0	a_0b_0
+				a_3b_1	a_2b_1	a_1b_1	a_0b_1	
+			a_3b_2	a_2b_2	a_1b_2	a_0b_2		
+		a_3b_3	a_2b_3	a_1b_3	a_0b_3			
+	1			1				
	p_7	p_6	p_5	p_4	p_3	p_2	p_1	p_0

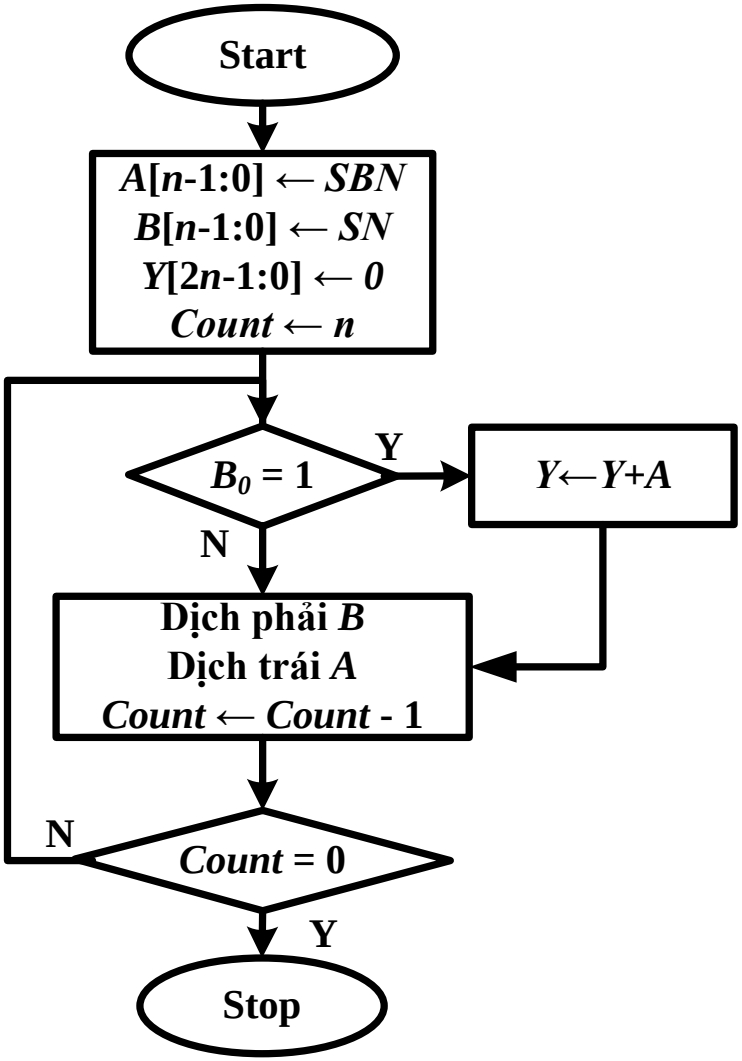
Bộ nhân có dấu



Bộ nhân nối tiếp

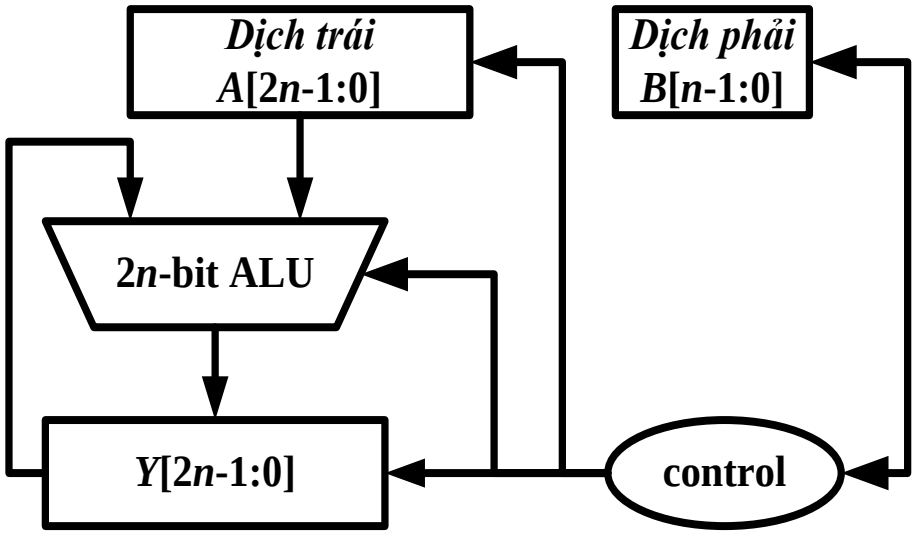
- Sử dụng bộ cộng để cộng các tích cục bộ
- Thực hiện phép nhân trong vài chu kỳ đồng hồ
- Lưu số bị nhân, số nhân và kết quả tạm thời trong các thanh ghi
- Với mỗi bit b_i của số nhân B (từ phải qua trái)
  - Nhân b_i với số bị nhân A và cộng tích với kết quả tổng tạm thời Y ⇔ Nếu $b_i = 1$ thì cộng A vào Y
  - Dịch A sang trái 1 bit

Bộ nhân nối tiếp



Triển khai gồm:

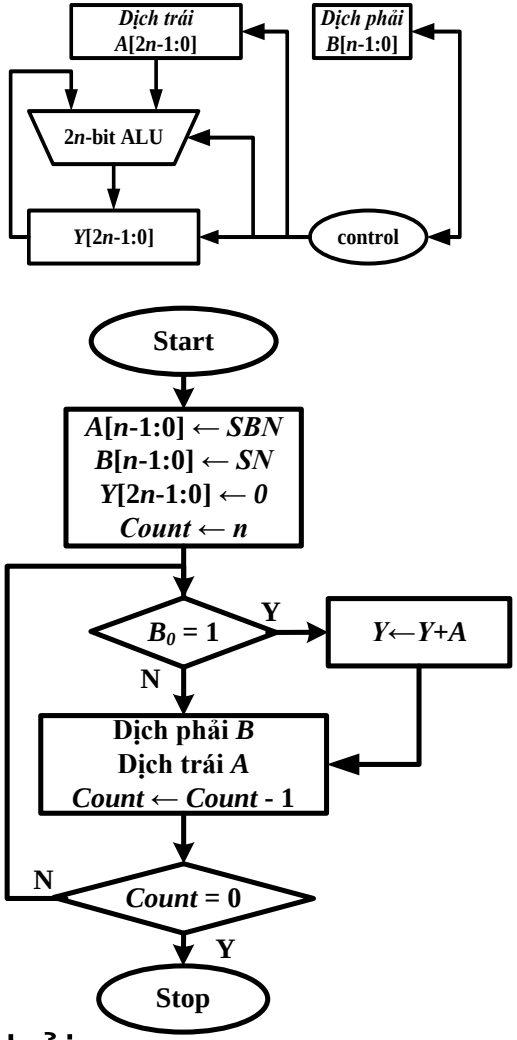
- 2 thanh ghi 2n bit
- 1 thanh ghi n bit
- 1 bộ cộng 2n bit
- 1 khối điều khiển



Ví dụ 2.10 - Bộ nhân nối tiếp

Y	0	0	0	0	0	0	0	0
A	0	0	0	0	1	1	0	1
Y	0	0	0	0	1	1	0	1
A	0	0	0	1	1	0	1	0
Y	0	0	1	0	0	1	1	1
A	0	0	1	1	0	1	0	0
Y	0	0	1	0	0	1	1	1
A	0	1	1	0	1	0	0	0
Y	1	0	0	0	1	1	1	1
A	1	1	0	1	0	0	0	0

B	1	0	1	1	Counter=4
B	0	1	0	1	Counter=3
B	0	0	1	0	Counter=2
B	0	0	0	1	Counter=1
B	0	0	0	0	Counter=0

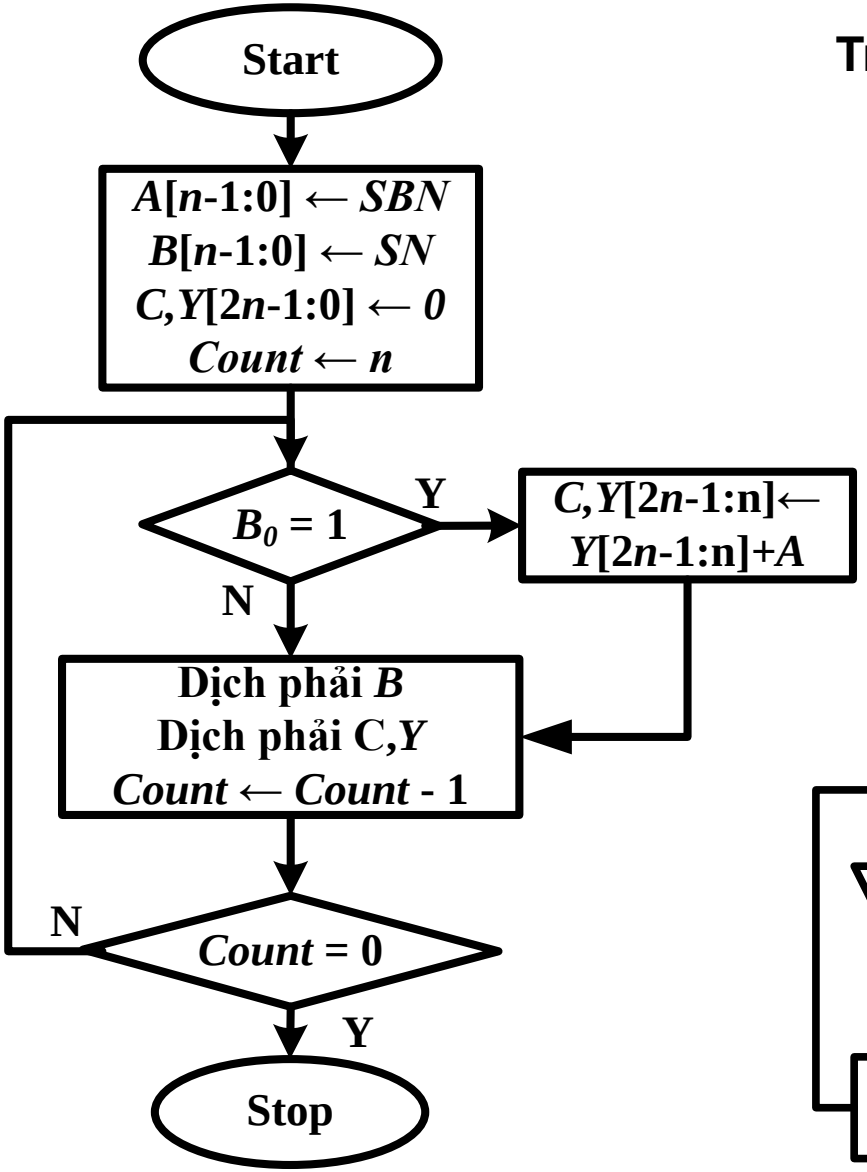


Nhận xét:

- ✓ Một nửa số bit của A luôn bằng 0
- ✓ Khi A dịch trái, bit 0 được thêm vào bên phải
- ➔ các bit LSB của tích không bị ảnh hưởng

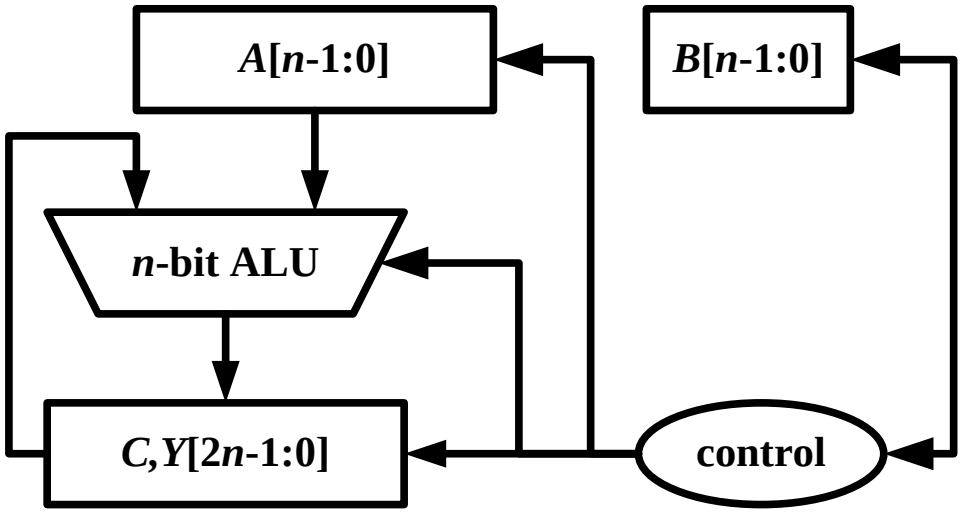
Ý tưởng: Giữ A ở phía trái của tích và dịch tích sang phải

Bộ nhân nối tiếp – Dùng *n*-bit ALU



Triển khai gồm:

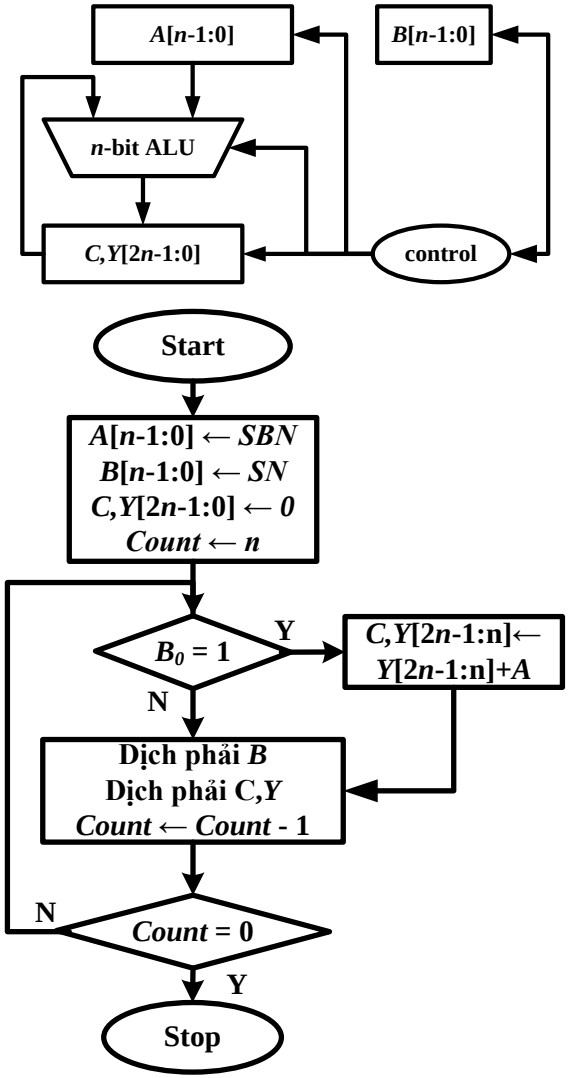
- 2 thanh ghi n bit
- 1 thanh ghi 2n+1 bit
- 1 bộ cộng n bit
- 1 khối điều khiển



Ví dụ 2.11 – Bộ nhân nối tiếp

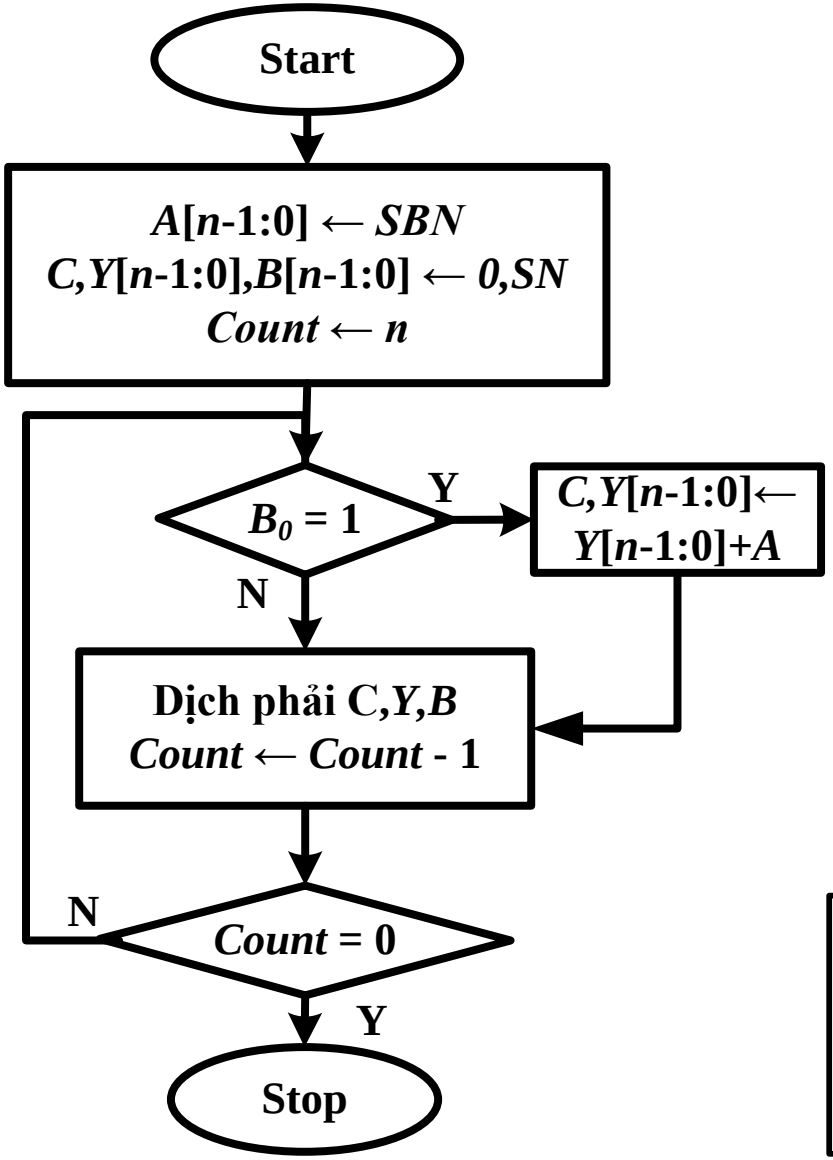
Y	0	0	0	0	0	0	0	0	0
A		1	1	0	1				
Y	0	1	1	0	1	0	0	0	0
Y	0	0	1	1	0	1	0	0	0
A		1	1	0	1				
Y	1	0	0	1	1	1	0	0	0
Y	0	1	0	0	1	1	1	0	0
A		1	1	0	1				
Y	0	1	0	0	1	1	1	0	0
Y	0	0	1	0	0	1	1	1	0
A		1	1	0	1				
Y	1	0	0	0	1	1	1	1	0
Y	0	1	0	0	0	1	1	1	1

B	1	0	1	1	*Counter=4*
B	0	1	0	1	*Counter=3*
B	0	0	1	0	*Counter=2*
B	0	0	0	1	*Counter=1*
B	0	0	0	0	*Counter=0*



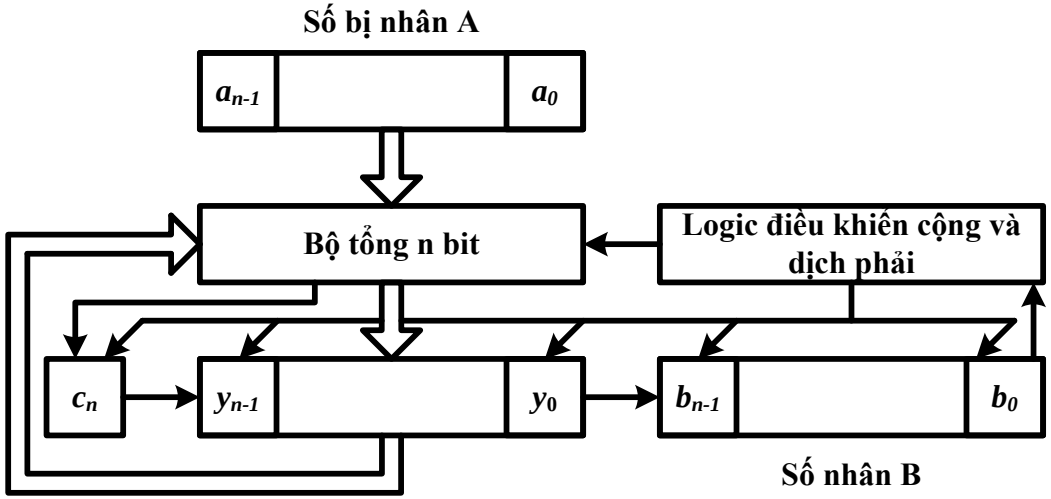
Nhận xét: Trong quá trình nhân chỉ một số bit của *Y* có ý nghĩa với kết quả

Bộ nhân nối tiếp



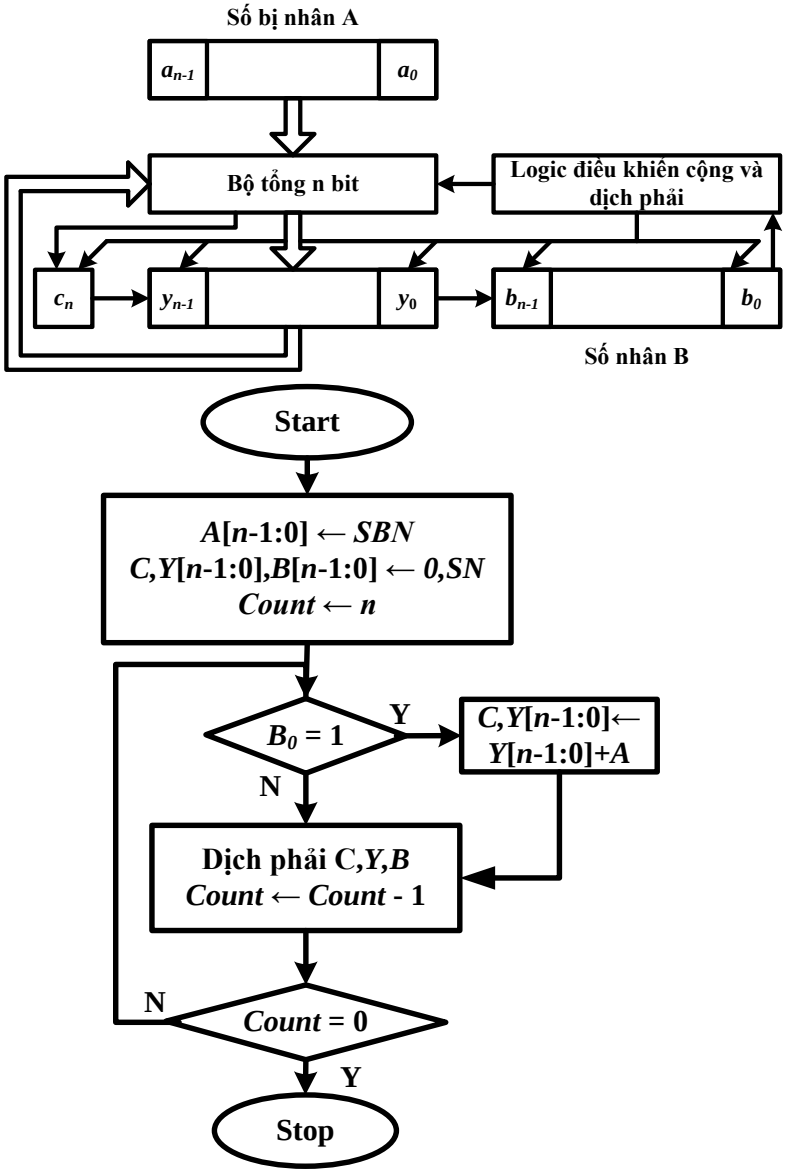
Triển khai gồm:

- 1 thanh ghi n bit
- 1 thanh ghi 2n+1 bit
- 1 bộ cộng n bit
- 1 khối điều khiển



Ví dụ 2.12 – Bộ nhân nối tiếp

Y	0	0	0	0	0	1	0	1	1	*Counter=4*
A		1	1	0	1					
Y	0	1	1	0	1	1	0	1	1	
Y	0	0	1	1	0	1	1	0	1	*Counter=3*
A		1	1	0	1					
Y	1	0	0	1	1	1	1	0	1	
Y	0	1	0	0	1	1	1	1	0	*Counter=2*
A		1	1	0	1					
Y	0	1	0	0	1	1	1	1	0	
Y	0	0	1	0	0	1	1	1	1	*Counter=1*
A		1	1	0	1					
Y	1	0	0	0	1	1	1	1	1	
Y	0	1	0	0	0	1	1	1	1	*Counter=0*



Nhân Booth

Nhân với một chuỗi số 1

$$A * 1111 = A * (2^4 - 2^0) = A * 2^4 - A$$

➔ Dịch A sang trái 4 bit và trừ đi A

Số bị nhân B chứa chuỗi số 1 từ bit vị trí *v* đến bit vị trí *u*

$$(b_{n-1}, b_{n-2}, \ldots b_{u+1}, b_u, \ldots, b_v, b_{v-1}, .., b_0) = (b_{n-1}, b_{n-2}, \ldots 0, 1, \ldots, 1, 0, .., b_0)$$

Chuỗi bit có thể thay thế bằng $2^{u+1} - 2^v$

➔Các phép nhân và cộng cho các bít b_u đến b_v có thể được thay bằng phép dịch trái và phép trừ

Ví dụ:

B = 001110 (14_{10}), u = 3, v = 2 ➔ A x B = A* 2^4 – A*2^1

Ví dụ 2.13 – Bộ nhân Booth

A
B
B

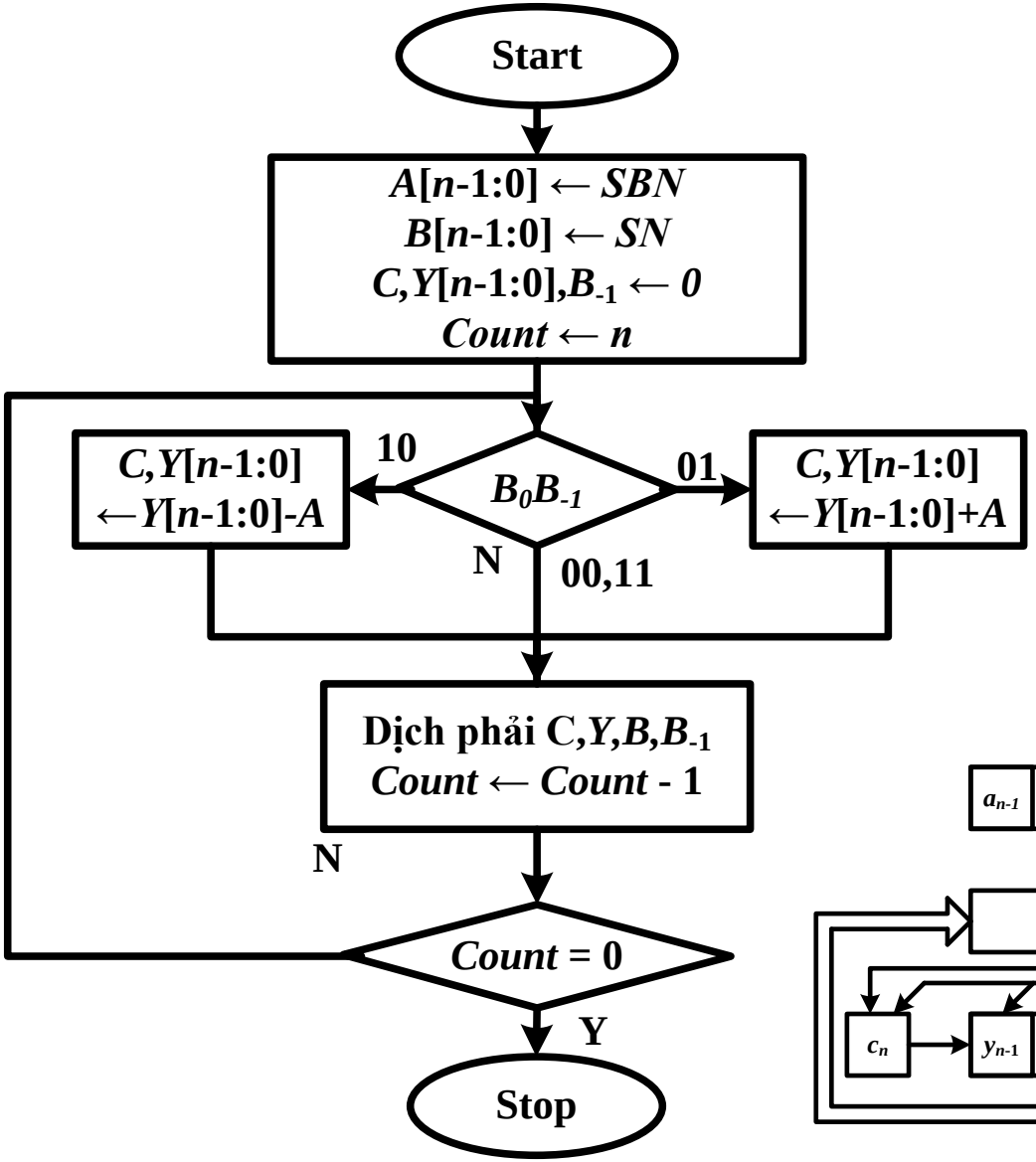
0	1	0	1	0	1
0	0	1	1	1	0
0	+1	0	0	-1	0

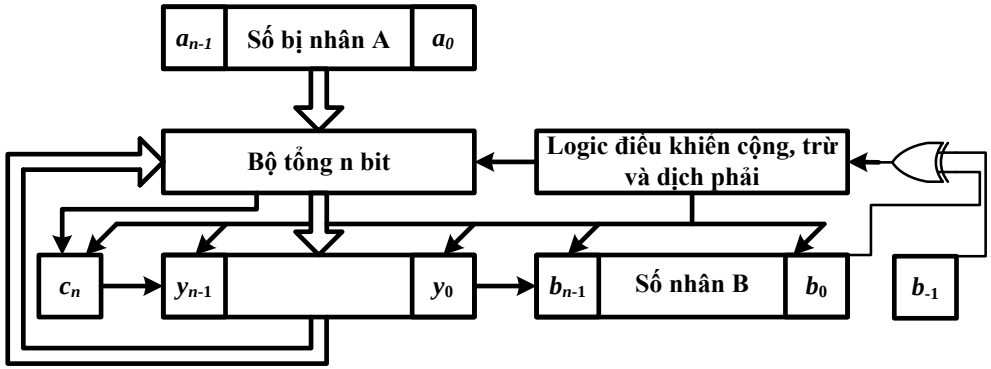
0	0	0	0	0	0	1	0	1	0	1	0
1	1	1	1	1	1	0	1	0	1	1	0
0	0	0	1	0	1	0	1	0	0	0	0
0	0	0	1	0	0	1	0	0	1	1	0

Thực hiện

1. **Bắt đầu chuỗi số 1 (chuyển từ 0 sang 1, hai bit liên tiếp là 01): trừ đi số bị nhân**
2. **Trong chuỗi số 0, hoặc chuỗi số 1 (2 bit liên tiếp là 00 hoặc 11): dịch trái số bị nhân**
3. **Kết thúc chuỗi số 1 (chuyển từ 1 sang 0, hai bit liên tiếp là 10): cộng với số bị nhân**

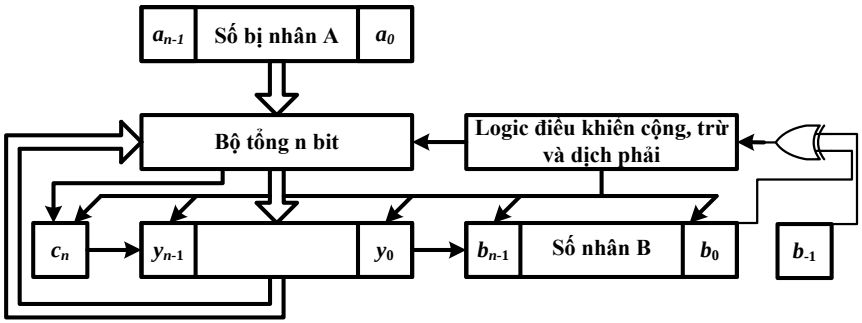
Thuật toán nhân Booth

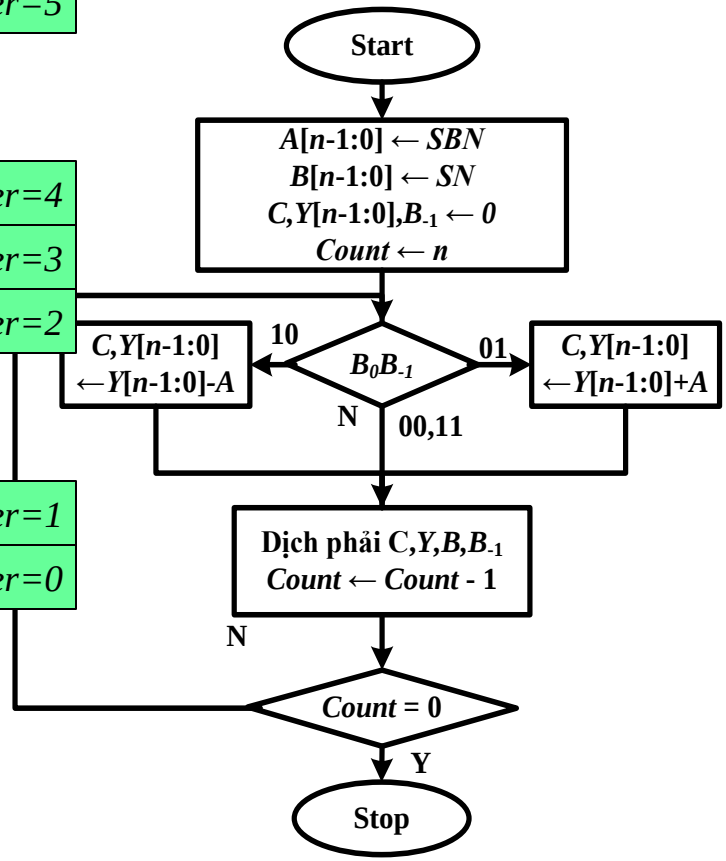




Ví dụ 2.14 – Minh họa thuật toán Booth

A		0	1	0	1	0	1								
Y	0	0	0	0	0	0	0	0	0	1	1	1	0	0	*Counter=6*
Y	0	0	0	0	0	0	0	0	0	0	1	1	1	0	*Counter=5*
-A		1	0	1	0	1	1								
Y	1	1	0	1	0	1	1	0	0	0	1	1	1	0	
Y	1	1	1	0	1	0	1	1	0	0	0	1	1	1	*Counter=4*
Y	1	1	1	1	0	1	0	1	1	0	0	0	1	1	*Counter=3*
Y	1	1	1	1	1	0	1	0	1	1	0	0	0	1	*Counter=2*
+A		0	1	0	1	0	1								
Y	0	0	1	0	0	1	0	0	1	1	0	0	0	1	
Y	0	0	0	1	0	0	1	0	0	1	1	0	0	0	*Counter=1*
Y	0	0	0	0	1	0	0	1	0	0	1	1	0	0	*Counter=0*





Nhân Booth: Nhân có dấu

Vì *a*, *b* là 2 số có dấu dạng bù 2:

$$a = -2^{n-1}*a_n + 2^{n-2}*a_{n-2} + ... + 2*a_1 + a_0$$

Xét 2 bit liên tiếp (a_i , a_{i-1}), hiệu của chúng và hoạt động nhân:

a_i	a_{i-1}	$a_i - a_{i-1}$	action
1	0	-1	Trừ b, và dịch
0	1	1	Cộng b và dịch
0	0	0	Bỏ qua
1	1	1	Bỏ qua

Giá trị được tính toán bởi bộ nhân Booth:

$$(0 - a_0) * b + (a_0 - a_1) * b * 2 + (a_1 - a_2) * b * 2^2 ... +$$

$$(a_{n-3} - a_{n-2}) * b * 2^{n-2} + (a_{n-2} - a_{n-1}) * b * 2^{n-1}$$

Triển khai các số hạng và tối giản:

$$b * (-2^{n-1}*a_n + 2^{n-2}*a_{n-2} + ... + 2*a_1 + a_0) = b * a.$$

which is exactly the product of a and b.

Phép chia

Phép nhân

Cộng với số bị nhân và dịch trái số bị nhân

Tối ưu phần cứng: cộng với số bị nhân và dịch phải tích

Phép chia

Trừ cho số chia và dịch phải số chia

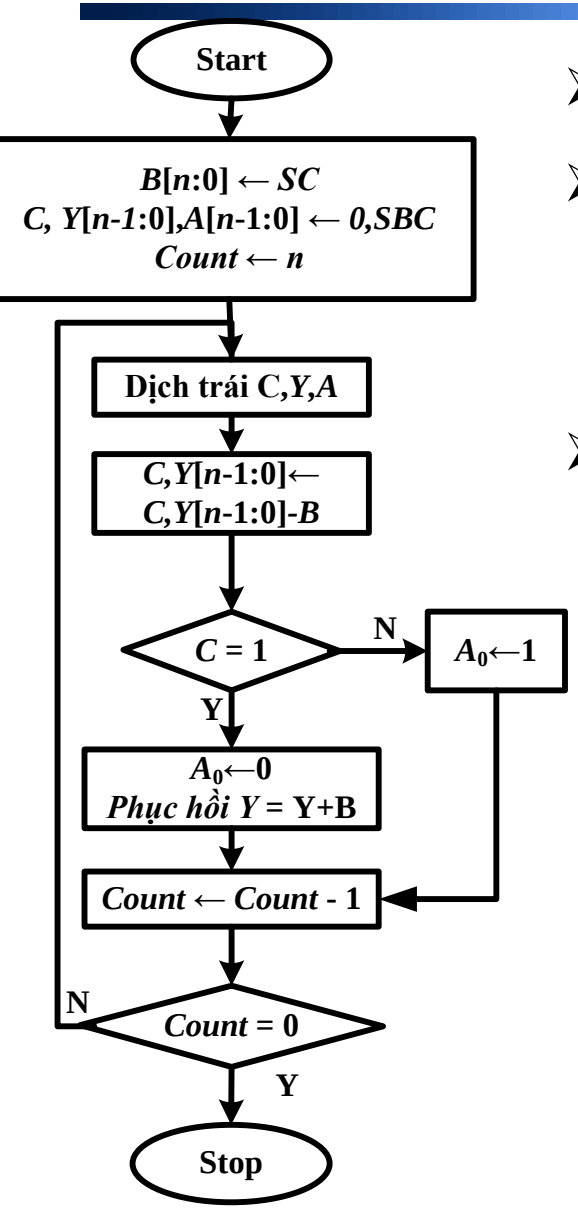
Ví dụ: Y = A / B

A	0	0	1	1	1
B	1	1			
B	0	1	1		
B	0	0	1	1	
A	0	0	0	0	1
B	0	0	0	1	1

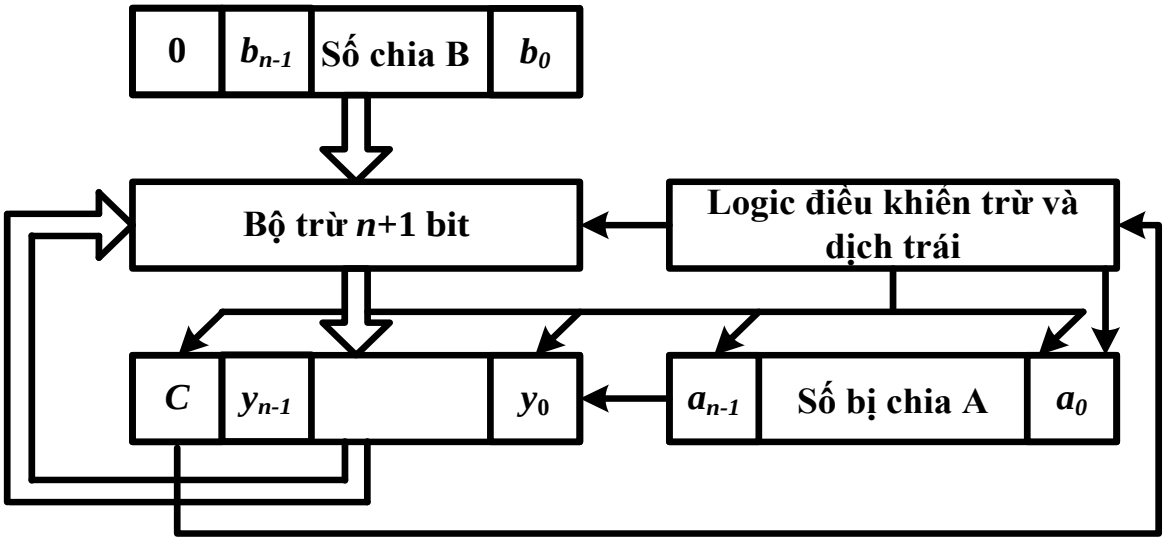
Y	0			
Y	0	0		
Y	0	0	1	
Y	0	0	1	0

Tối ưu phần cứng: trừ cho số chia và dịch trái phần dư

Thuật toán chia nối tiếp

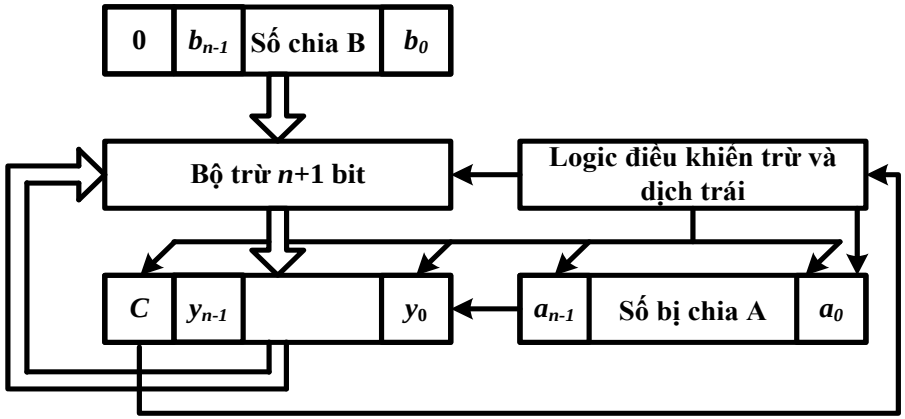


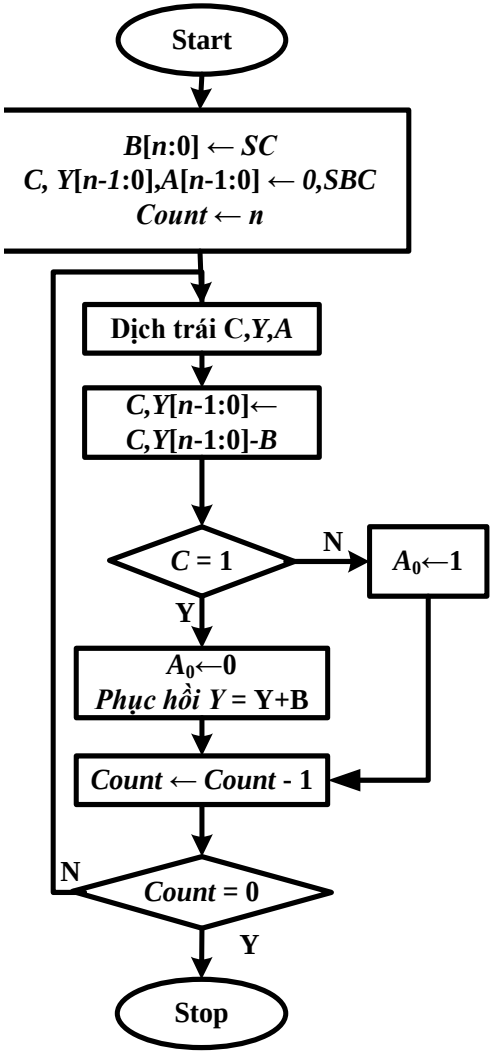
- Trừ Y = Y – B và kiểm tra bit dấu *C* của kết quả
- Nếu *C* = 1 (phép trừ kết quả âm) ➔
  - Bít kết quả a_0=0
  - Phục hồi số bị chia: Y = Y + B
- Nếu *C* = 0 (phép trừ kết quả dương) ➔
  - Bít kết quả a_0=1



Ví dụ 2.15 – Bộ chia nối tiếp

B		0	0	1	1					
Y	0	0	0	0	0	0	1	1	1	*Counter=4*
Y	0	0	0	0	0	1	1	1	0	
-B	1	1	1	0	1					
Y	1	1	1	0	1	1	1	1	0	
Y	0	0	0	0	0	1	1	1	0	*Counter=3*
Y	0	0	0	0	1	1	1	0	0	
-B	1	1	1	0	1					
Y	1	1	1	1	0	1	1	1	0	
Y	0	0	0	0	1	1	1	0	0	*Counter=2*
Y	0	0	0	1	1	1	0	0	0	
-B	1	1	1	0	1					
Y	0	0	0	0	0	1	0	0	0	
Y	0	0	0	0	0	1	0	0	1	*Counter=1*
Y	0	0	0	0	1	0	0	1	0	
-B	1	1	1	0	1					
Y	1	1	1	1	0	0	0	1	0	
Y	0	0	0	0	1	0	0	1	0	*Counter=0*





Chia có dấu

1. Chia phần giá trị tuyệt đối
2. Xác định dấu của kết quả
   - Dấu của thương:
     - Dương nếu số chia và số bị chia cùng dấu
     - Âm nếu số chia và số bị chia khác dấu
   - Dấu của phần dư: luôn cùng dấu với số bị chia
3. Đảo kết quả nếu cần thiết

Phép toán dấu phẩy động

- Số dấu phẩy động: $A = \pm M_A 2^{E_A}$; $B = \pm M_B 2^{E_B}$
- Phép cộng trừ: giả sử $E_A > E_B$

$A + B = \left(\pm M_A \pm M'_B\right)2^{E_A}$ trong đó $M'_B = M_B 2^{E_A - E_B}$

- Phép nhân

$A \cdot B = \left(\pm M_A\right) \cdot \left(\pm M_B\right)2^{E_A + E_B}$

- Phép chia

$A / B = \left(\pm M_A\right) / \left(\pm M_B\right)2^{E_A - E_B}$

- Chuẩn hóa kết quả: Đưa định trị về dạng chuẩn hóa và điều chỉnh số mũ tương ứng