memo

ディジタル電子回路

授業開始までしばらくお待ちください。

オンライン視聴できない人へ。

オンラインで受講する人も基本的に一緒です。

自宅ネットワークの事情により、授業のストリーミング配信の視聴が困難な学生は以下の対応をしてください。

1. この授業のスライドをよく読んで、不明な点は自分で調べるなどして、わかる範囲で内容を理解する。
2. このページも含め、**必要な部分がすべて理解できたと思うまで以下の 2 ステップを繰り返す**。
   - わからない部分を e-mail 等で質問する。(宛先は `hiroyuki.kobayashi@oit.ac.jp`)
   - e-mail 等による返信をよく読んで理解する。
3. この資料の末尾にある課題を行い、この資料内の方法で (Google Forms で) 提出する。

授業の受講に関して

- 講義資料（スライド等）は**COMMON**に置く。
- 講義は**Google Meet** で行い、録画した講義は**Goole Drive**に置く。

  https://stream.meet.google.com/stream/1d1866da-5bff-4881-96b2-3745413fe31a

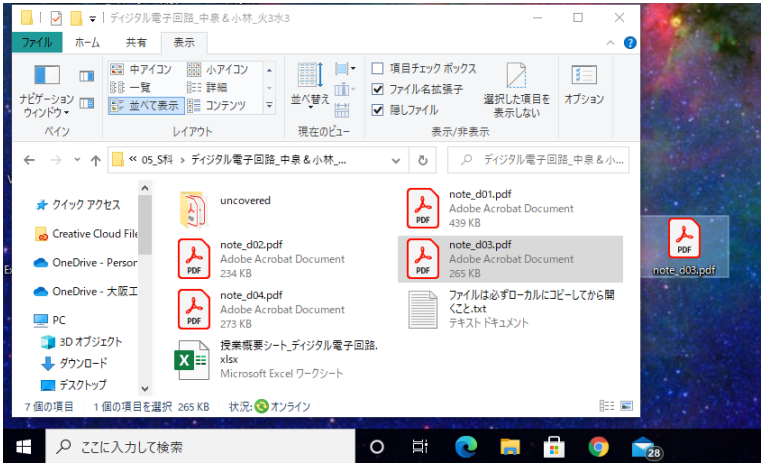
  https://drive.google.com/drive/folders/1bT-z3ICQyMYC_5JvlL29UZYqbOhVG492
- 出席確認レポートは**Google Forms** で提出。(毎回同一 URL)

  https://forms.gle/9ruwtfJg5LQgQNpU7
- **Slack**を補助的な連絡チャネルとする。必須ではないので使いたくなければ使わなくてもいい。授業に関連したちょっとした（重要でない）追加説明をする。気楽な質問手段としても活用されたい。登録は大学の e-mail アドレスで行うこと。

  https://oitkobayashi.slack.com

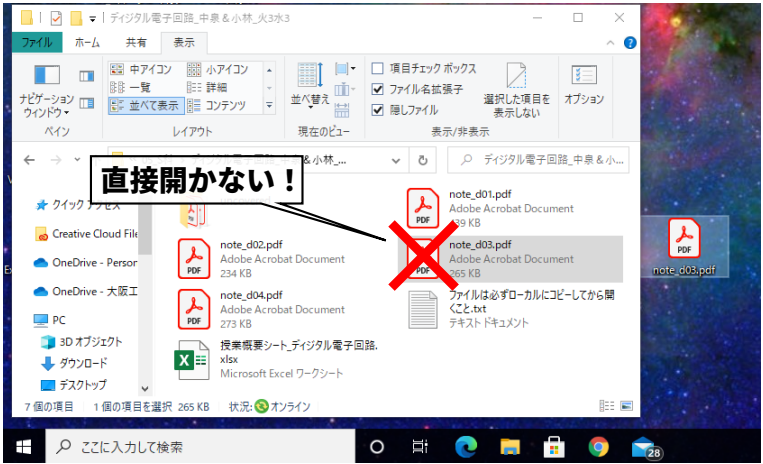
COMMON フォルダの注意事項 (全授業共通)

根源的に悪いのは Windows の仕様なのですが、ご協力ください。



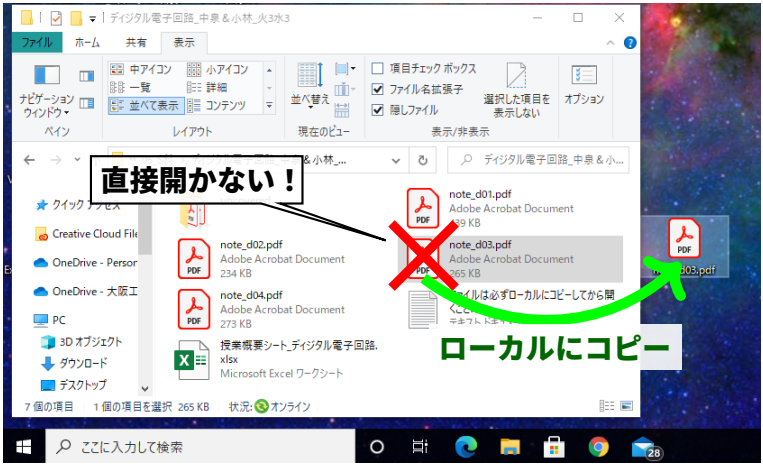
COMMON フォルダの注意事項 (全授業共通)

根源的に悪いのは Windows の仕様なのですが、ご協力ください。



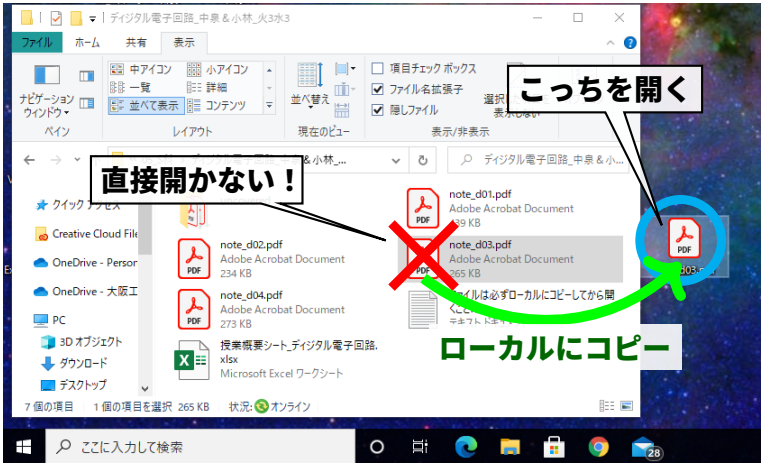
COMMON フォルダの注意事項 (全授業共通)

根源的に悪いのは Windows の仕様なのですが、ご協力ください。



COMMON フォルダの注意事項 (全授業共通)

根源的に悪いのは Windows の仕様なのですが、ご協力ください。



R/S 科ディジタル電子回路
Digital Electronics
『足し算・引き算』

Google Meet

小林裕之・中泉文孝

大阪工業大学 RD 学部システムデザイン工学科・ロボット工学科

OSAKA INSTITUTE OF TECHNOLOGY

13 of 14

a LaTeX + Beamer slideshow

演算回路～加算器と減算器～

電子回路に計算をさせるには…?

- _____ がすべての基本。
- _____ も基本。
- ハードウェアで足し算のできないコンピュータはほとんどない。
- ハードウェアで引き算のできないコンピュータもほとんどない。
- ハードウェアで掛け算や割り算ができないコンピュータはいくらでもある。

演算回路～加算器と減算器～

電子回路に計算をさせるには…?

- **足し算** がすべての基本。
- _____ も基本。
- ハードウェアで足し算のできないコンピュータはほとんどない。
- ハードウェアで引き算のできないコンピュータもほとんどない。
- ハードウェアで掛け算や割り算ができないコンピュータはいくらでもある。

演算回路～加算器と減算器～

電子回路に計算をさせるには…?

- **足し算** がすべての基本。
- **引き算** も基本。
- ハードウェアで足し算のできないコンピュータはほとんどない。
- ハードウェアで引き算のできないコンピュータもほとんどない。
- ハードウェアで掛け算や割り算ができないコンピュータはいくらでもある。

足し算をするには

(ふつうの) 人間の場合

- 10 進数 1 桁の計算を暗算する
- 2 桁以上の計算は筆算（もしくは脳内筆算）

⬇

『1 桁 +1 桁』つまり　とおりの足し算の暗記（@小学校）。

計算機の場合

- 2 進数 1 桁の計算を一発で回路で演算する
- 2 桁以上の計算はそれを {多数並べる or 繰り返し使う}

⬇

『1 桁 +1 桁』つまり　とおりの足し算を一発で演算する回路（→ 加算器）。

足し算をするには

(ふつうの) 人間の場合

- 10 進数 1 桁の計算を暗算する
- 2 桁以上の計算は筆算（もしくは脳内筆算）

⬇

『1 桁 +1 桁』つまり100とおりの足し算の暗記（@小学校）。

計算機の場合

- 2 進数 1 桁の計算を一発で回路で演算する
- 2 桁以上の計算はそれを {多数並べる or 繰り返し使う}

⬇

『1 桁 +1 桁』つまり とおりの足し算を一発で演算する回路（→ 加算器）。

足し算をするには

(ふつうの) 人間の場合

- 10 進数 1 桁の計算を暗算する
- 2 桁以上の計算は筆算（もしくは脳内筆算）

⬇

『1 桁 +1 桁』つまり100とおりの足し算の暗記（@小学校）。

計算機の場合

- 2 進数 1 桁の計算を一発で回路で演算する
- 2 桁以上の計算はそれを {多数並べる or 繰り返し使う}

⬇

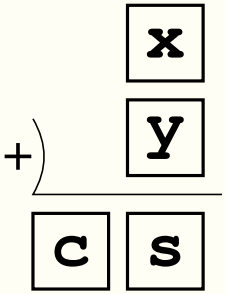
『1 桁 +1 桁』つまり4とおりの足し算を一発で演算する回路（→ 加算器）。

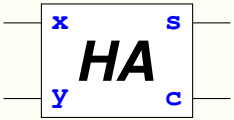
こんがらがらないように

- 2 進数は “0” と “1” の 2 つのディジットを並べて数値を表現する。
- この “0” と “1” それぞれに、ブール代数の**0 元**と**1 元**を対応させる。
- たまたまこの授業では 2 進数のディジットとブール代数の元に割り当てている文字が “0” と “1” で同じだっただけで、本来は全然無関係なものを紐づけているだけ。

半加算器 (HA;)

2 進数 1 桁の『足し算』





入力		出力	
x	y	c	s
0	0		
0	1		
1	0		
1	1		

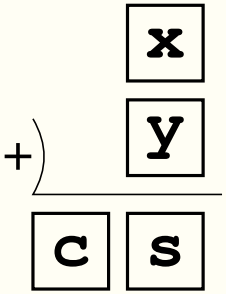
- s: 和 (sum)
- c: 繰り上がり (carry)

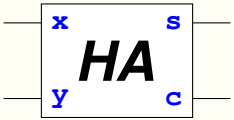
s =

c =

半加算器 (HA; half adder)

2 進数 1 桁の『足し算』





入力		出力	
x	y	c	s
0	0		
0	1		
1	0		
1	1		

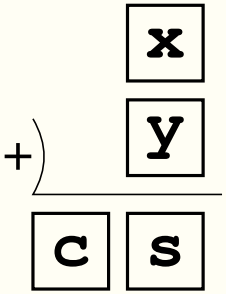
- s: 和 (sum)
- c: 繰り上がり (carry)

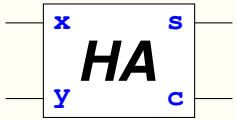
s =

c =

半加算器 (HA; half adder)

2 進数 1 桁の『足し算』





入力		出力	
x	y	c	s
0	0		**0**
0	1		**1**
1	0		**1**
1	1		**0**

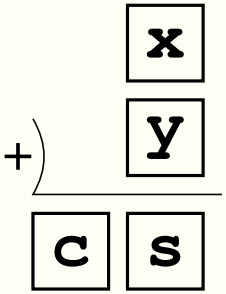
- s: 和 (sum)
- c: 繰り上がり (carry)

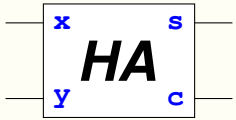
s =

c =

半加算器 (HA; half adder)

2 進数 1 桁の『足し算』





入力		出力	
x	y	c	s
0	0	**0**	**0**
0	1	**0**	**1**
1	0	**0**	**1**
1	1	**1**	**0**

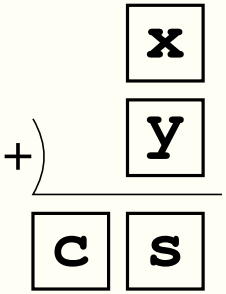
- s: 和 (sum)
- c: 繰り上がり (carry)

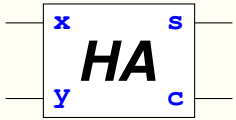
s =

c =

半加算器 (HA; half adder)

2 進数 1 桁の『足し算』





入力		出力	
x	y	c	s
0	0	**0**	**0**
0	1	**0**	**1**
1	0	**0**	**1**
1	1	**1**	**0**

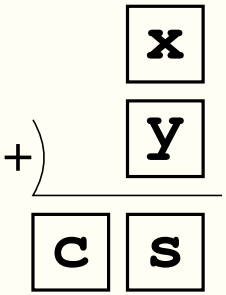
- s: 和 (sum)
- c: 繰り上がり (carry)

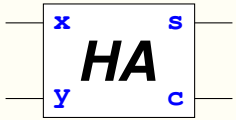
$$s = (\bar{x} \cdot y) + (x \cdot \bar{y})$$

$$c =$$

半加算器 (HA; half adder)

2 進数 1 桁の『足し算』





入力		出力	
x	y	c	s
0	0	**0**	**0**
0	1	**0**	**1**
1	0	**0**	**1**
1	1	**1**	**0**

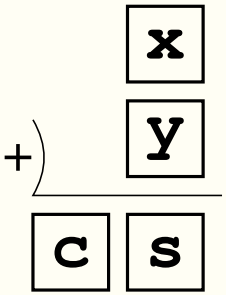
- s: 和 (sum)
- c: 繰り上がり (carry)

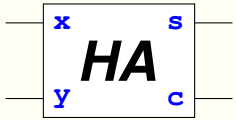
$$s = (\overline{x} \cdot y) + (x \cdot \overline{y}) = x \oplus y$$

$$c =$$

半加算器 (HA; half adder)

2 進数 1 桁の『足し算』





入力		出力	
x	y	c	s
0	0	**0**	**0**
0	1	**0**	**1**
1	0	**0**	**1**
1	1	**1**	**0**

- s: 和 (sum)
- c: 繰り上がり (carry)

$$s = (\bar{x} \cdot y) + (x \cdot \bar{y}) = x \oplus y$$

$$c = x \cdot y$$

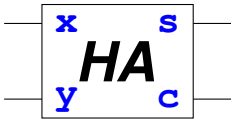
でも、HA は所詮 “half” 加算器でした。

- **半加算器では 2 桁以上の足し算は** _____。なぜならば、_____ (carry) を処理できないから。
- 完全な足し算をするには、次の 3 つの入力が必要。
  - 2 つの足す数 (x, y)
  - 1 つの (c_0)

2 入力 2 出力の
『半加算器』

でも、HA は所詮 “half” 加算器でした。

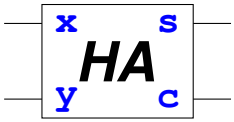
- **半加算器では 2 桁以上の足し算は できない**。なぜならば、(carry) を処理できないから。
- 完全な足し算をするには、次の 3 つの入力が必要。
  - 2 つの足す数 (x, y)
  - 1 つの (c_0)



2 入力 2 出力の
『半加算器』

でも、HA は所詮 “half” 加算器でした。

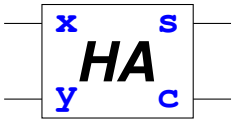
- **半加算器では 2 桁以上の足し算は できない**。なぜならば、下位の桁からの繰り上がり(carry) を処理できないから。
- 完全な足し算をするには、次の 3 つの入力が必要。
  - 2 つの足す数 (x, y)
  - 1 つの (c_0)



2 入力 2 出力の
『半加算器』

でも、HA は所詮 “half” 加算器でした。

- **半加算器では 2 桁以上の足し算は できない**。なぜならば、下位の桁からの繰り上がり(carry) を処理できないから。
- 完全な足し算をするには、次の 3 つの入力が必要。
  - 2 つの足す数 (x, y)
  - 1 つの**繰り上がり**(c_0)

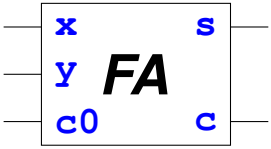


2 入力 2 出力の
『半加算器』

でも、HA は所詮 “half” 加算器でした。

- **半加算器では 2 桁以上の足し算は できない**。なぜならば、下位の桁からの繰り上がり(carry) を処理できないから。
- 完全な足し算をするには、次の 3 つの入力が必要。
  - 2 つの足す数 (x, y)
  - 1 つの**繰り上がり**(c_0)

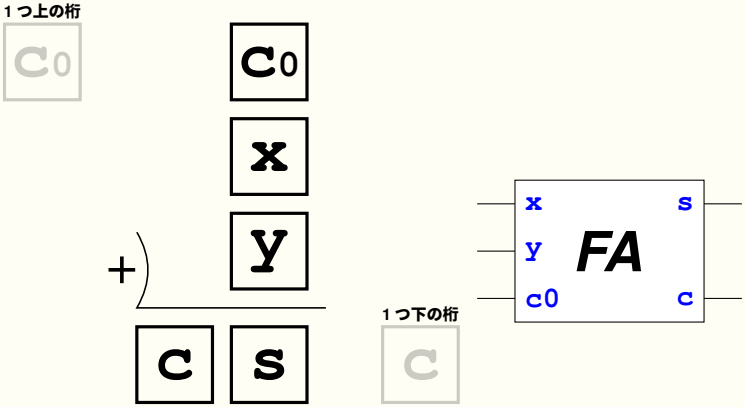
2 入力 2 出力の
『半加算器』



3 入力 2 出力の
『全加算器』

全加算器 (FA;)

繰り上がり入力つき 2 進数 1 桁の『足し算』

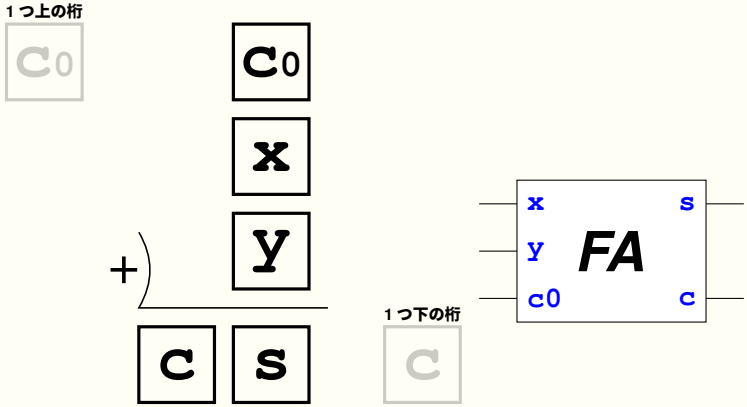


入力			出力	
x	y	c_0	c	s
0	0	0		
0	1	0		
1	0	0		
1	1	0		
0	0	1		
0	1	1		
1	0	1		
1	1	1		

s: 和 (sum), c: 繰り上がり (carry), c_0: 下位からの繰り上がり

全加算器 (FA; full adder)

繰り上がり入力つき 2 進数 1 桁の『足し算』

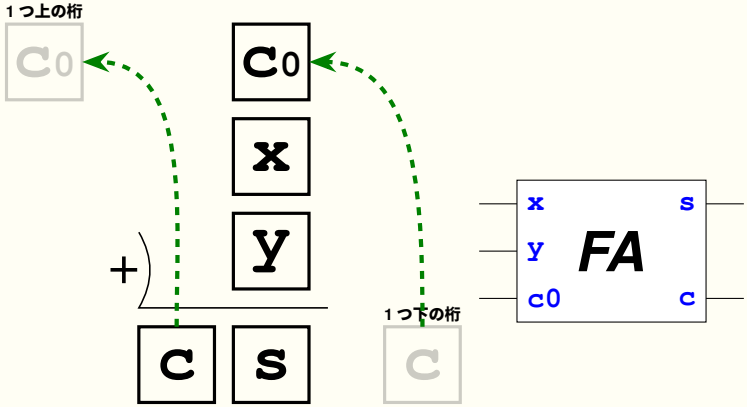


入力			出力	
x	y	c_0	c	s
0	0	0		
0	1	0		
1	0	0		
1	1	0		
0	0	1		
0	1	1		
1	0	1		
1	1	1		

s: 和 (sum), c: 繰り上がり (carry), c_0: 下位からの繰り上がり

全加算器 (FA; full adder)

繰り上がり入力つき 2 進数 1 桁の『足し算』

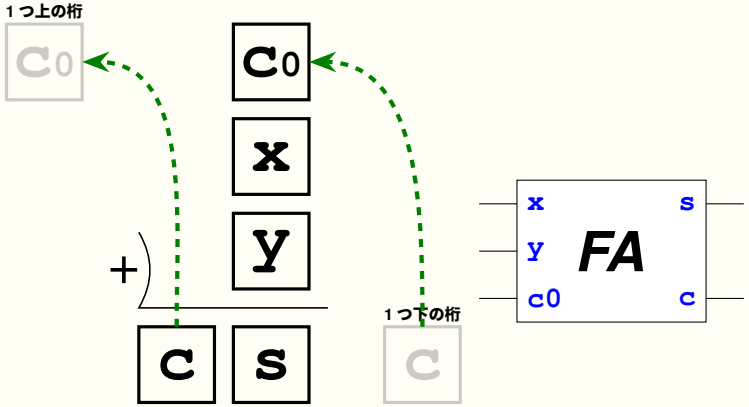


入力			出力	
x	y	c_0	c	s
0	0	0		
0	1	0		
1	0	0		
1	1	0		
0	0	1		
0	1	1		
1	0	1		
1	1	1		

s: 和 (sum), c: 繰り上がり (carry), c_0: 下位からの繰り上がり

全加算器 (FA; full adder)

繰り上がり入力つき 2 進数 1 桁の『足し算』

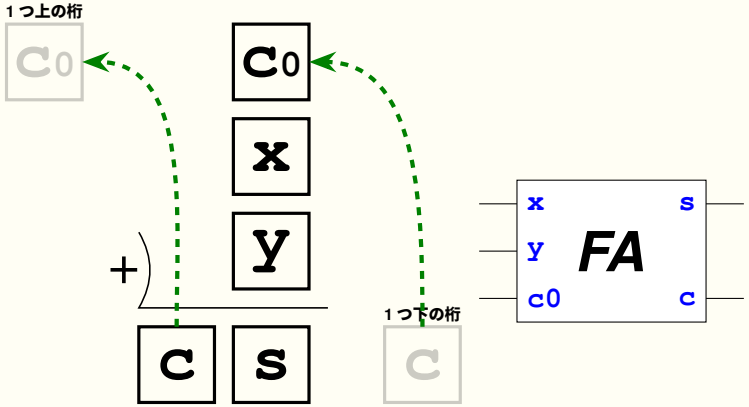


入力			出力	
x	y	c_0	c	s
0	0	0		**0**
0	1	0		**1**
1	0	0		**1**
1	1	0		**0**
0	0	1		**1**
0	1	1		**0**
1	0	1		**0**
1	1	1		**1**

s: 和 (sum), c: 繰り上がり (carry), c_0: 下位からの繰り上がり

全加算器 (FA; full adder)

繰り上がり入力つき 2 進数 1 桁の『足し算』



入力			出力	
x	y	c_0	c	s
0	0	0	**0**	**0**
0	1	0	**0**	**1**
1	0	0	**0**	**1**
1	1	0	**1**	**0**
0	0	1	**0**	**1**
0	1	1	**1**	**0**
1	0	1	**1**	**0**
1	1	1	**1**	**1**

s: 和 (sum), c: 繰り上がり (carry), c_0: 下位からの繰り上がり

全加算器の論理式

問: 全加算器を論理設計せよ。

$$s =$$

$$c =$$

全加算器の論理式

問: 全加算器を論理設計せよ。

$$s = (\overline{x} \cdot y \cdot \overline{c}_0) + (x \cdot \overline{y} \cdot \overline{c}_0) + (\overline{x} \cdot \overline{y} \cdot c_0) + (x \cdot y \cdot c_0)$$

$$c =$$

全加算器の論理式

問: 全加算器を論理設計せよ。

$$s = (\overline{x} \cdot y \cdot \overline{c}_0) + (x \cdot \overline{y} \cdot \overline{c}_0) + (\overline{x} \cdot \overline{y} \cdot c_0) + (x \cdot y \cdot c_0)$$

$$c = (x \cdot y) + (x \cdot c_0) + (y \cdot c_0)$$

全加算器の論理式

問: 全加算器を論理設計せよ。

$$
\begin{aligned}
s &= (\overline{x} \cdot y \cdot \overline{c}_0) + (x \cdot \overline{y} \cdot \overline{c}_0) + (\overline{x} \cdot \overline{y} \cdot c_0) + (x \cdot y \cdot c_0) \\
&= \text{『3 入力偶数パリティ生成器』と考えると…} = \\
c &= (x \cdot y) + (x \cdot c_0) + (y \cdot c_0)
\end{aligned}
$$

全加算器の論理式

問: 全加算器を論理設計せよ。

$$
\begin{aligned}
s &= (\overline{x} \cdot y \cdot \overline{c}_0) + (x \cdot \overline{y} \cdot \overline{c}_0) + (\overline{x} \cdot \overline{y} \cdot c_0) + (x \cdot y \cdot c_0) \\
&= \text{『3 入力偶数パリティ生成器』と考えると…} = x \oplus y \oplus c_0 \\
c &= (x \cdot y) + (x \cdot c_0) + (y \cdot c_0)
\end{aligned}
$$

全加算器の論理式

問: 全加算器を論理設計せよ。

$$
\begin{aligned}
s &= (\overline{x} \cdot y \cdot \overline{c}_0) + (x \cdot \overline{y} \cdot \overline{c}_0) + (\overline{x} \cdot \overline{y} \cdot c_0) + (x \cdot y \cdot c_0) \\
&= \text{『3 入力偶数パリティ生成器』と考えると…} = x \oplus y \oplus c_0 \\
c &= (x \cdot y) + (x \cdot c_0) + (y \cdot c_0)
\end{aligned}
$$

(一度主加法標準系に戻してみる)

全加算器の論理式

問: 全加算器を論理設計せよ。

$$
\begin{aligned}
s &= (\overline{x} \cdot y \cdot \overline{c}_0) + (x \cdot \overline{y} \cdot \overline{c}_0) + (\overline{x} \cdot \overline{y} \cdot c_0) + (x \cdot y \cdot c_0) \\
&= \text{『3 入力偶数パリティ生成器』と考えると…} = x \oplus y \oplus c_0 \\
c &= (x \cdot y) + (x \cdot c_0) + (y \cdot c_0) \\
&\quad \text{(一度主加法標準系に戻してみる)} \\
&= \underbrace{(\overline{x} \cdot y \cdot c_0) + (x \cdot \overline{y} \cdot c_0)} + \underbrace{(x \cdot y \cdot \overline{c}_0) + (x \cdot y \cdot c_0)}
\end{aligned}
$$

全加算器の論理式

問: 全加算器を論理設計せよ。

$$
\begin{aligned}
s &= (\overline{x} \cdot y \cdot \overline{c}_0) + (x \cdot \overline{y} \cdot \overline{c}_0) + (\overline{x} \cdot \overline{y} \cdot c_0) + (x \cdot y \cdot c_0) \\
&= \text{『3 入力偶数パリティ生成器』と考えると…} = x \oplus y \oplus c_0 \\
c &= (x \cdot y) + (x \cdot c_0) + (y \cdot c_0) \\
&\quad \text{(一度主加法標準系に戻してみる)} \\
&= \underbrace{(\overline{x} \cdot y \cdot c_0) + (x \cdot \overline{y} \cdot c_0)}_{(x \oplus y) \cdot c_0} + \underbrace{(x \cdot y \cdot \overline{c}_0) + (x \cdot y \cdot c_0)}
\end{aligned}
$$

全加算器の論理式

問: 全加算器を論理設計せよ。

$$
\begin{aligned}
s &= (\overline{x} \cdot y \cdot \overline{c}_0) + (x \cdot \overline{y} \cdot \overline{c}_0) + (\overline{x} \cdot \overline{y} \cdot c_0) + (x \cdot y \cdot c_0) \\
&= \text{『3 入力偶数パリティ生成器』と考えると…} = x \oplus y \oplus c_0 \\
c &= (x \cdot y) + (x \cdot c_0) + (y \cdot c_0) \\
&\quad \text{(一度主加法標準系に戻してみる)} \\
&= \underbrace{(\overline{x} \cdot y \cdot c_0) + (x \cdot \overline{y} \cdot c_0)}_{(x \oplus y) \cdot c_0} + \underbrace{(x \cdot y \cdot \overline{c}_0) + (x \cdot y \cdot c_0)}_{x \cdot y}
\end{aligned}
$$

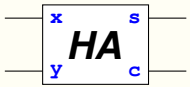
全加算器の論理式

問: 全加算器を論理設計せよ。

$$
\begin{aligned}
s &= (\overline{x} \cdot y \cdot \overline{c}_0) + (x \cdot \overline{y} \cdot \overline{c}_0) + (\overline{x} \cdot \overline{y} \cdot c_0) + (x \cdot y \cdot c_0) \\
&= \text{『3 入力偶数パリティ生成器』と考えると…} = x \oplus y \oplus c_0 \\
c &= (x \cdot y) + (x \cdot c_0) + (y \cdot c_0) \\
&\quad \text{(一度主加法標準系に戻してみる)} \\
&= \underbrace{(\overline{x} \cdot y \cdot c_0) + (x \cdot \overline{y} \cdot c_0)}_{(x \oplus y) \cdot c_0} + \underbrace{(x \cdot y \cdot \overline{c}_0) + (x \cdot y \cdot c_0)}_{x \cdot y} \\
&= ((x \oplus y) \cdot c_0) + (x \cdot y)
\end{aligned}
$$

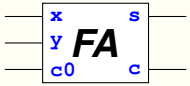
FA の作り方・別解

(ダメだと思っていた)HA がここにきて再登場

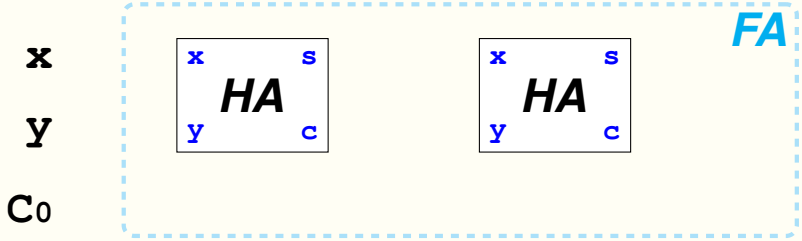


$$s = x \oplus y,\ c = x \cdot y$$

FA はこうやっても作れる!

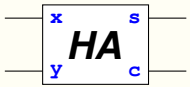


$$s = x \oplus y \oplus c_0,\ c = (x \oplus y) \cdot c_0 + x \cdot y$$



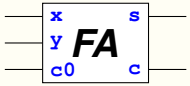
FA の作り方・別解

(ダメだと思っていた)HA がここにきて再登場

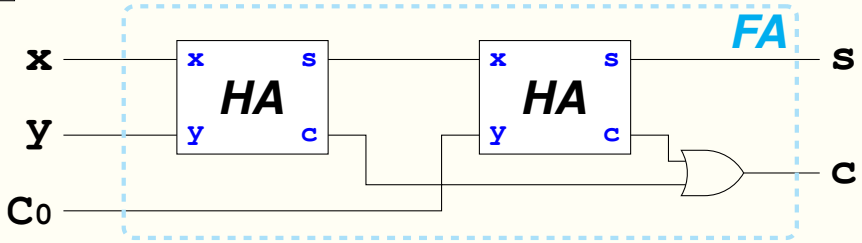


$$s = x \oplus y,\ c = x \cdot y$$

FA はこうやっても作れる!



$$s = x \oplus y \oplus c_0,\ c = (x \oplus y) \cdot c_0 + x \cdot y$$



n ビットの足し算

一桁ができたら、あとは（筆算と同様に）並べるだけ。

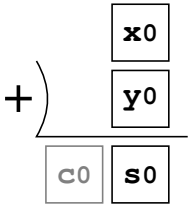
$(x_3\ x_2\ x_1\ x_0)_2 + (y_3\ y_2\ y_1\ y_0)_2 = (c_3\ s_3\ s_2\ s_1\ s_0)_2$ の計算:

このやり方を　　方式という。

n ビットの足し算

一桁ができたら、あとは（筆算と同様に）並べるだけ。

$(x_3\ x_2\ x_1\ x_0)_2 + (y_3\ y_2\ y_1\ y_0)_2 = (c_3\ s_3\ s_2\ s_1\ s_0)_2$ の計算:

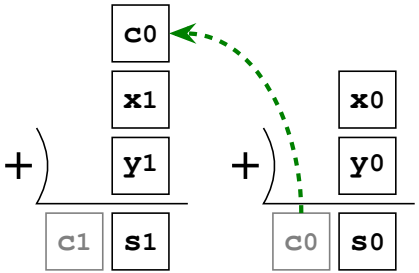


このやり方を　　方式という。

n ビットの足し算

一桁ができたら、あとは（筆算と同様に）並べるだけ。

$(x_3\ x_2\ x_1\ x_0)_2 + (y_3\ y_2\ y_1\ y_0)_2 = (c_3\ s_3\ s_2\ s_1\ s_0)_2$ の計算:

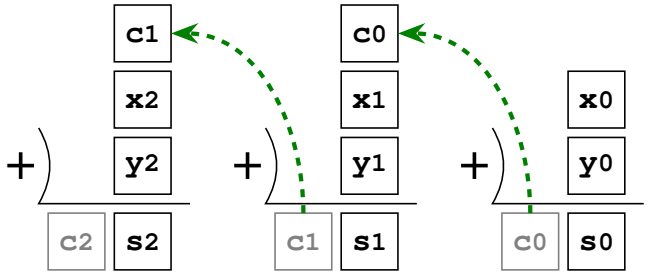


このやり方を　　方式という。

n ビットの足し算

一桁ができたら、あとは（筆算と同様に）並べるだけ。

$(x_3\ x_2\ x_1\ x_0)_2 + (y_3\ y_2\ y_1\ y_0)_2 = (c_3\ s_3\ s_2\ s_1\ s_0)_2$ の計算:

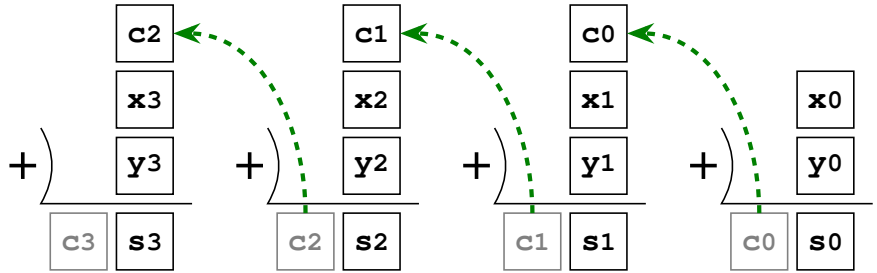


このやり方を　　方式という。

n ビットの足し算

一桁ができたら、あとは（筆算と同様に）並べるだけ。

$(x_3\ x_2\ x_1\ x_0)_2 + (y_3\ y_2\ y_1\ y_0)_2 = (c_3\ s_3\ s_2\ s_1\ s_0)_2$ の計算:

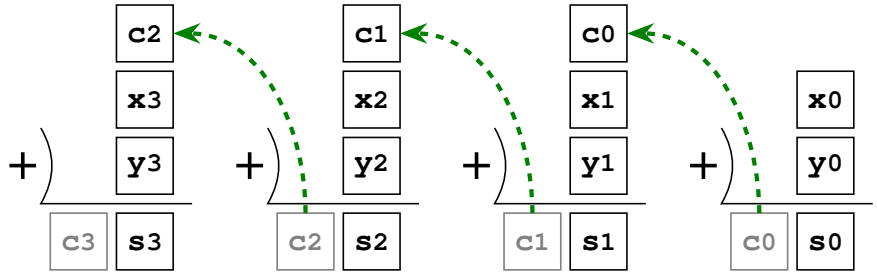


このやり方を　　方式という。

n ビットの足し算

一桁ができたら、あとは（筆算と同様に）並べるだけ。

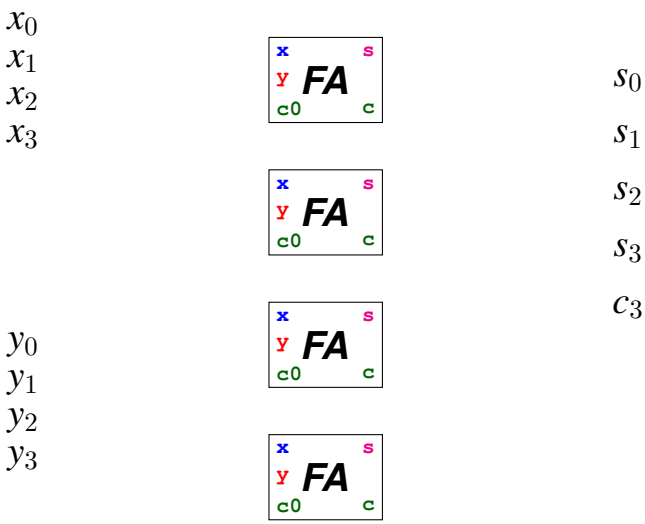
$(x_3\ x_2\ x_1\ x_0)_2 + (y_3\ y_2\ y_1\ y_0)_2 = (c_3\ s_3\ s_2\ s_1\ s_0)_2$ の計算:



このやり方を**リプルキャリー**方式という。

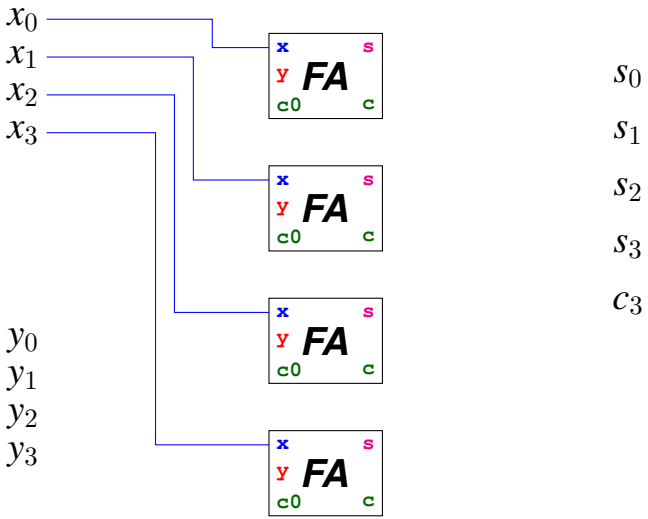
リプルキャリー方式加算回路 (前ページの計算をする回路)

『回路』というほどご大層な図でもありませんけど……。



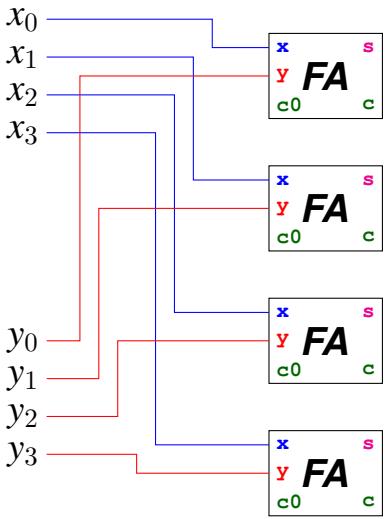
リプルキャリー方式加算回路 (前ページの計算をする回路)

『回路』というほどご大層な図でもありませんけど……。



リプルキャリー方式加算回路 (前ページの計算をする回路)

『回路』というほどご大層な図でもありませんけど……。



s_0

s_1

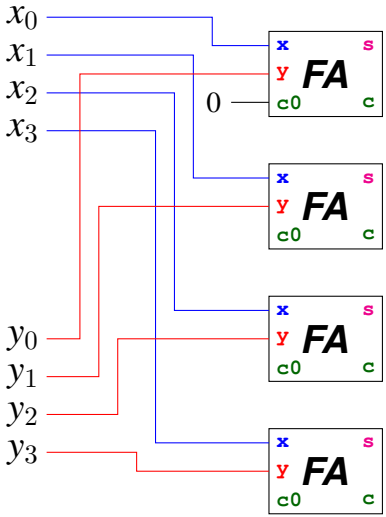
s_2

s_3

c_3

リプルキャリー方式加算回路 (前ページの計算をする回路)

『回路』というほどご大層な図でもありませんけど……。



s_0

s_1

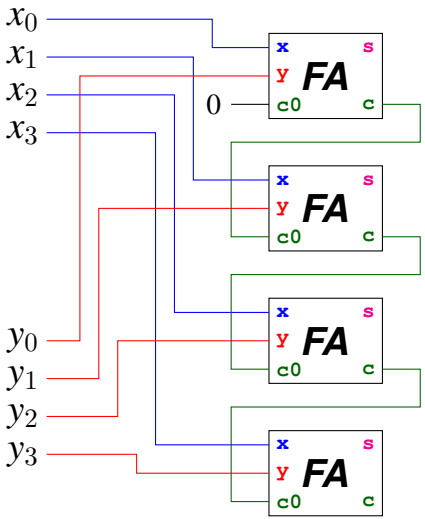
s_2

s_3

c_3

リプルキャリー方式加算回路 (前ページの計算をする回路)

『回路』というほどご大層な図でもありませんけど……。



s_0

s_1

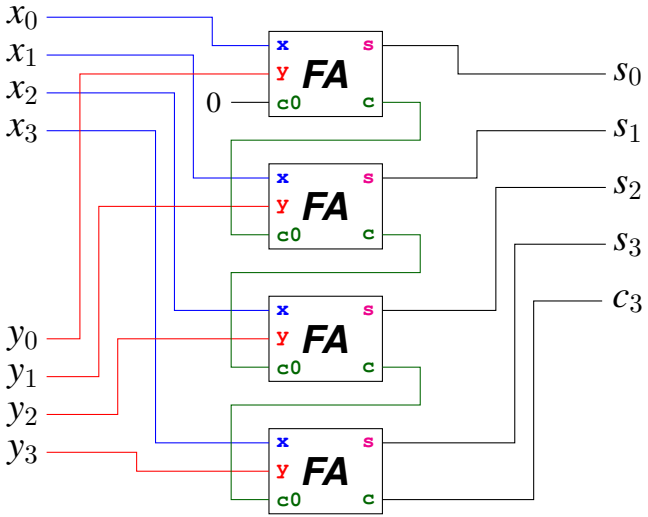
s_2

s_3

c_3

リプルキャリー方式加算回路 (前ページの計算をする回路)

『回路』というほどご大層な図でもありませんけど……。



リップルキャリー方式の加算器の性能

問: n ビットのリップルキャリー方式の加算器の演算速度のオーダーと回路の複雑さのオーダーを考えよ。

解答

- **速度**:
  キャリーが n 回遅延するので…
  速度のオーダー:
- **複雑さ:**
  FA が n 個必要（で、ほかに特に増える要素もなさそう）なので…
  複雑さのオーダー:

リップルキャリー方式の加算器の性能

問: n ビットのリップルキャリー方式の加算器の演算速度のオーダーと回路の複雑さのオーダーを考えよ。

解答

- **速度**:
  キャリーが n 回遅延するので…
  $$\text{速度のオーダー: } \mathcal{O}(n)$$
- **複雑さ:**
  FA が n 個必要（で、ほかに特に増える要素もなさそう）なので…
  $$\text{複雑さのオーダー:}$$

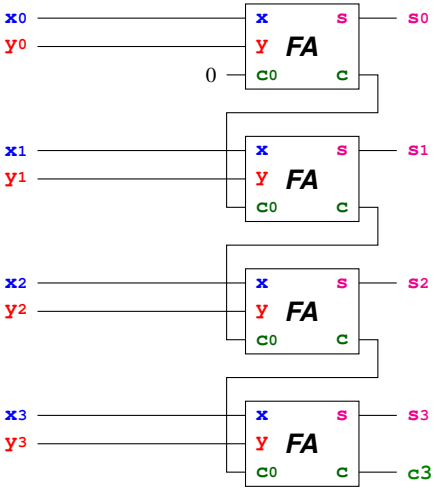
リップルキャリー方式の加算器の性能

問: n ビットのリップルキャリー方式の加算器の演算速度のオーダーと回路の複雑さのオーダーを考えよ。

解答

- **速度**:
  キャリーが n 回遅延するので…
  $$\text{速度のオーダー: } \mathcal{O}(n)$$
- **複雑さ:**
  FA が n 個必要（で、ほかに特に増える要素もなさそう）なので…
  $$\text{複雑さのオーダー: } \mathcal{O}(n)$$

より高速な足し算: 方式



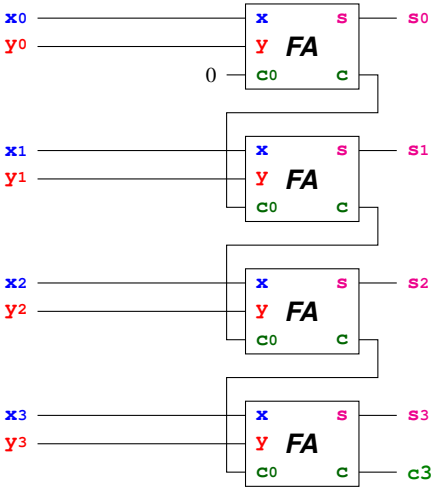
- 最初 (最下位) のキャリーが発生する条件は明らかに $x_0 = 1$ かつ $y_0 = 1$ のときだから…

$$CLA_1 =$$

- 同様に、
  $CLA_2 =$
- 同様に も作れるはず。

速度: , 複雑さ:

より高速な足し算: キャリールックアヘッド方式



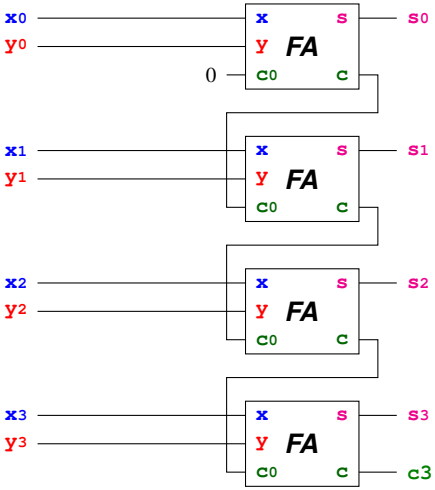
- 最初 (最下位) のキャリーが発生する条件は明らかに $x_0 = 1$ かつ $y_0 = 1$ のときだから…

$$CLA_1 =$$

- 同様に、
  $CLA_2 =$
- 同様に　　も作れるはず。

速度:　　, 複雑さ:

より高速な足し算: キャリールックアヘッド方式



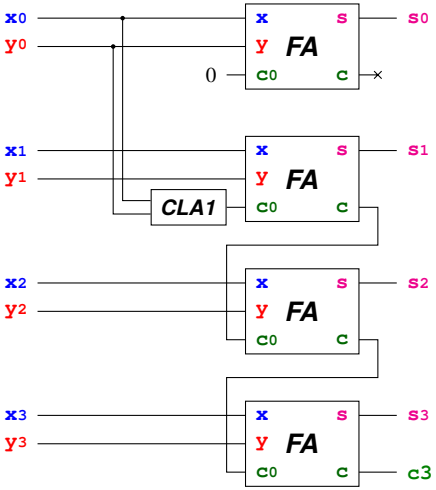
- 最初 (最下位) のキャリーが発生する条件は明らかに $x_0 = 1$ かつ $y_0 = 1$ のときだから…

$$CLA_1 = x_0 \cdot y_0$$

- 同様に、
  $CLA_2 =$
- 同様に　　も作れるはず。

速度:　　, 複雑さ:

より高速な足し算: キャリールックアヘッド方式



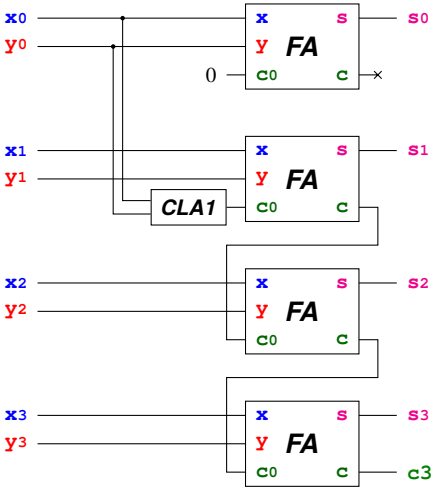
- 最初 (最下位) のキャリーが発生する条件は明らかに $x_0 = 1$ かつ $y_0 = 1$ のときだから…

$$CLA_1 = x_0 \cdot y_0$$

- 同様に、
  $CLA_2 =$
- 同様に　　も作れるはず。

速度:　　,　複雑さ:

より高速な足し算: キャリールックアヘッド方式



- 最初 (最下位) のキャリーが発生する条件は明らかに $x_0 = 1$ かつ $y_0 = 1$ のときだから…

$$CLA_1 = x_0 \cdot y_0$$

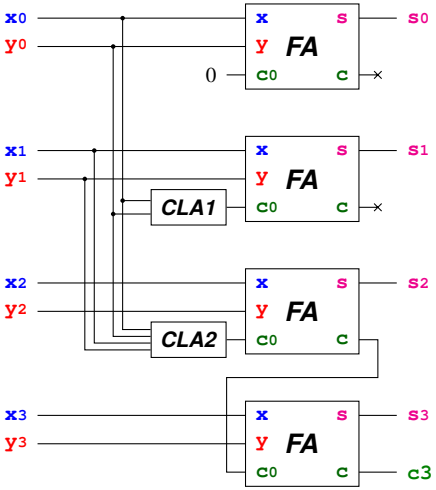
- 同様に、

$$CLA_2 = (x_1 \cdot y_1) + (x_1 \cdot x_0 \cdot y_0) + (x_0 \cdot y_1 \cdot y_0)$$

- 同様に　　も作れるはず。

速度:　　,　複雑さ:

より高速な足し算: キャリールックアヘッド方式



- 最初 (最下位) のキャリーが発生する条件は明らかに $x_0 = 1$ かつ $y_0 = 1$ のときだから…

$$CLA_1 = x_0 \cdot y_0$$

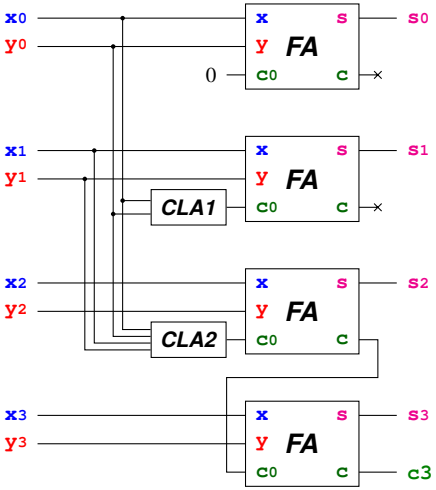
- 同様に、

$$CLA_2 = (x_1 \cdot y_1) + (x_1 \cdot x_0 \cdot y_0) + (x_0 \cdot y_1 \cdot y_0)$$

- 同様に　　も作れるはず。

速度:　　，　複雑さ:

より高速な足し算: キャリールックアヘッド方式



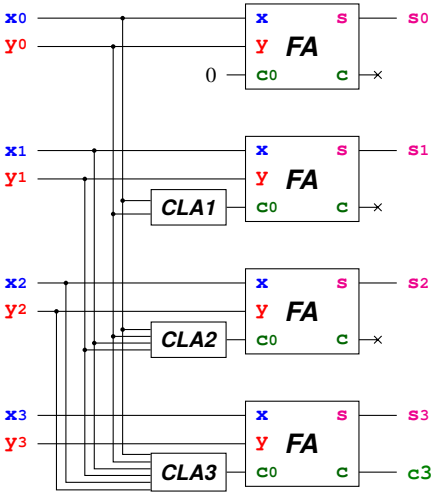
- 最初 (最下位) のキャリーが発生する条件は明らかに $x_0 = 1$ かつ $y_0 = 1$ のときだから…

$$CLA_1 = x_0 \cdot y_0$$

- 同様に、
$CLA_2 = (x_1 \cdot y_1) + (x_1 \cdot x_0 \cdot y_0) + (x_0 \cdot y_1 \cdot y_0)$
- 同様に$CLA_3(x_0, y_0, x_1, y_1, x_2, y_2)$も作れるはず。

速度: , 複雑さ:

より高速な足し算: キャリールックアヘッド方式



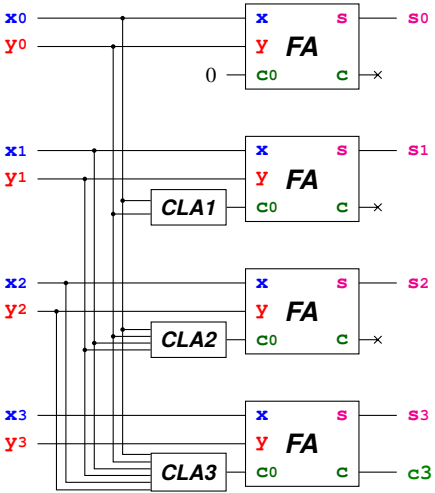
- 最初 (最下位) のキャリーが発生する条件は明らかに $x_0 = 1$ かつ $y_0 = 1$ のときだから…

$$CLA_1 = x_0 \cdot y_0$$

- 同様に、
$CLA_2 = (x_1 \cdot y_1) + (x_1 \cdot x_0 \cdot y_0) + (x_0 \cdot y_1 \cdot y_0)$
- 同様に$CLA_3(x_0, y_0, x_1, y_1, x_2, y_2)$も作れるはず。

速度: , 複雑さ:

より高速な足し算: キャリールックアヘッド方式



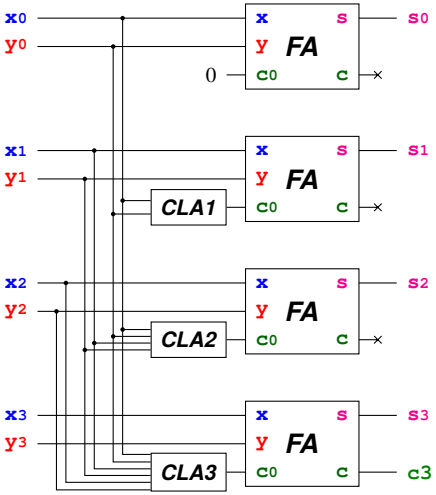
- 最初 (最下位) のキャリーが発生する条件は明らかに $x_0 = 1$ かつ $y_0 = 1$ のときだから…

$$CLA_1 = x_0 \cdot y_0$$

- 同様に、
$CLA_2 = (x_1 \cdot y_1) + (x_1 \cdot x_0 \cdot y_0) + (x_0 \cdot y_1 \cdot y_0)$
- 同様に$CLA_3(x_0, y_0, x_1, y_1, x_2, y_2)$も作れるはず。

速度: $\mathcal{O}(1)$, 複雑さ:

より高速な足し算: キャリールックアヘッド方式



- 最初 (最下位) のキャリーが発生する条件は明らかに $x_0 = 1$ かつ $y_0 = 1$ のときだから…

$$CLA_1 = x_0 \cdot y_0$$

- 同様に、
$CLA_2 = (x_1 \cdot y_1) + (x_1 \cdot x_0 \cdot y_0) + (x_0 \cdot y_1 \cdot y_0)$
- 同様に$CLA_3(x_0, y_0, x_1, y_1, x_2, y_2)$も作れるはず。

速度: $\mathcal{O}(1)$, 複雑さ: $\mathcal{O}(2^n)$

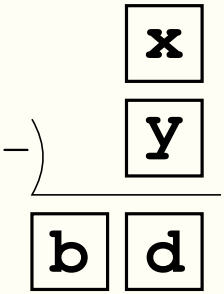
引き算をするには

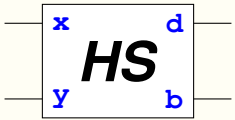
引き算をするには

足し算とほぼ同じ

半減算器 (HS;)

2 進数 1 桁の『引き算』





入力		出力	
x	y	b	d
0	0		
0	1		
1	0		
1	1		

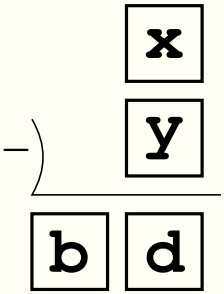
- d: 差 (difference)
- c: 桁下がり (borrow)

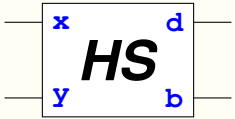
d =

b =

半減算器 (HS; half subtractor)

2 進数 1 桁の『引き算』





入力		出力	
x	y	b	d
0	0		
0	1		
1	0		
1	1		

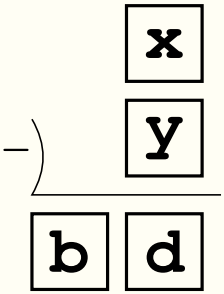
- d: 差 (difference)
- c: 桁下がり (borrow)

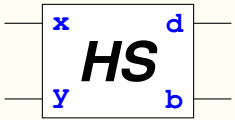
d =

b =

半減算器 (HS; half subtractor)

2 進数 1 桁の『引き算』





入力		出力	
x	y	b	d
0	0		**0**
0	1		**1**
1	0		**1**
1	1		**0**

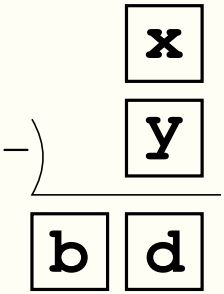
- d: 差 (difference)
- c: 桁下がり (borrow)

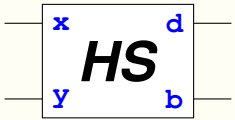
d =

b =

半減算器 (HS; half subtractor)

2 進数 1 桁の『引き算』





入力		出力	
x	y	b	d
0	0	**0**	**0**
0	1	**1**	**1**
1	0	**0**	**1**
1	1	**0**	**0**

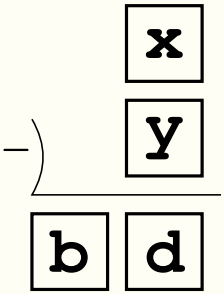
- d: 差 (difference)
- c: 桁下がり (borrow)

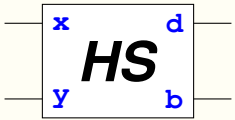
d =

b =

半減算器 (HS; half subtractor)

2 進数 1 桁の『引き算』





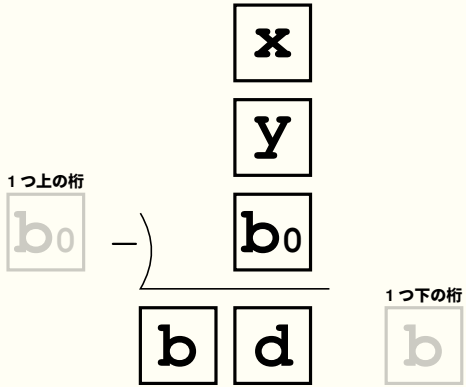
入力		出力	
x	y	b	d
0	0	**0**	**0**
0	1	**1**	**1**
1	0	**0**	**1**
1	1	**0**	**0**

- d: 差 (difference)
- c: 桁下がり (borrow)

$$d = x \oplus y$$
$$b = \overline{x} \cdot y$$

全減算器 (FS;)

桁下がり入力つき 2 進数 1 桁の『引き算』

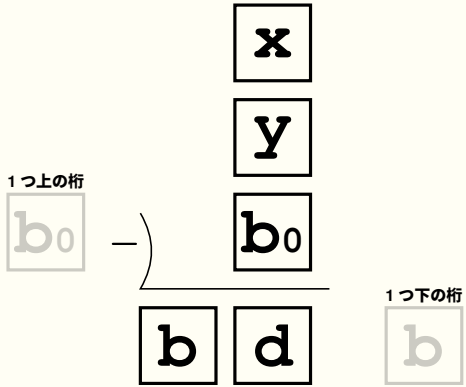


入力			出力	
x	y	b_0	b	d
0	0	0		
0	1	0		
1	0	0		
1	1	0		
0	0	1		
0	1	1		
1	0	1		
1	1	1		

d: 差 (difference), b: 桁下がり (borrow), b_0: 上位からの桁下がり

全減算器 (FS; full subtractor)

桁下がり入力つき 2 進数 1 桁の『引き算』

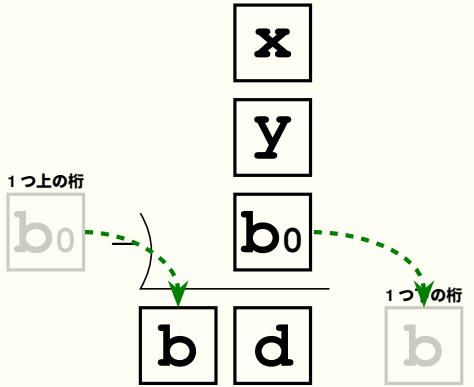


入力			出力	
x	y	b_0	b	d
0	0	0		
0	1	0		
1	0	0		
1	1	0		
0	0	1		
0	1	1		
1	0	1		
1	1	1		

d: 差 (difference), b: 桁下がり (borrow), b_0: 上位からの桁下がり

全減算器 (FS; full subtractor)

桁下がり入力つき 2 進数 1 桁の『引き算』

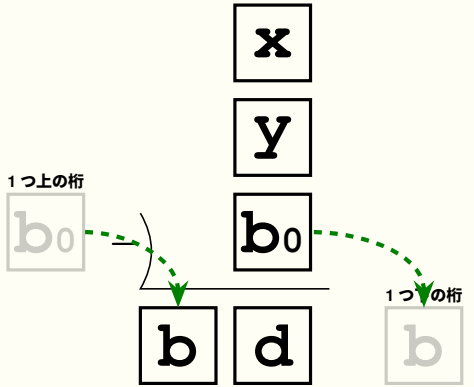


入力			出力	
x	y	b_0	b	d
0	0	0		
0	1	0		
1	0	0		
1	1	0		
0	0	1		
0	1	1		
1	0	1		
1	1	1		

d: 差 (difference), b: 桁下がり (borrow), b_0: 上位からの桁下がり

全減算器 (FS; full subtractor)

桁下がり入力つき 2 進数 1 桁の『引き算』

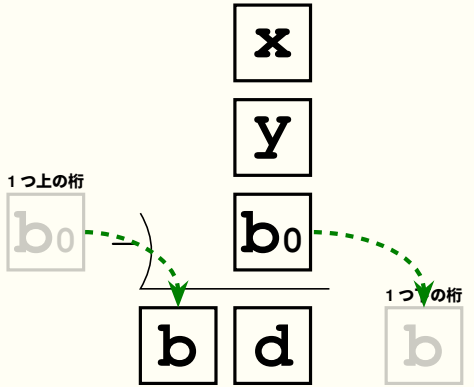


入力			出力	
x	y	b_0	b	d
0	0	0		**0**
0	1	0		**1**
1	0	0		**1**
1	1	0		**0**
0	0	1		**1**
0	1	1		**0**
1	0	1		**0**
1	1	1		**1**

d: 差 (difference), b: 桁下がり (borrow), b_0: 上位からの桁下がり

全減算器 (FS; full subtractor)

桁下がり入力つき 2 進数 1 桁の『引き算』

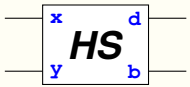


入力			出力	
x	y	b_0	b	d
0	0	0	**0**	**0**
0	1	0	**1**	**1**
1	0	0	**0**	**1**
1	1	0	**0**	**0**
0	0	1	**1**	**1**
0	1	1	**1**	**0**
1	0	1	**0**	**0**
1	1	1	**1**	**1**

d: 差 (difference), b: 桁下がり (borrow), b_0: 上位からの桁下がり

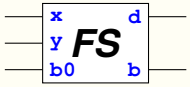
FS の作り方・別解

(ダメだと思っていた)HS がここにきて再登場

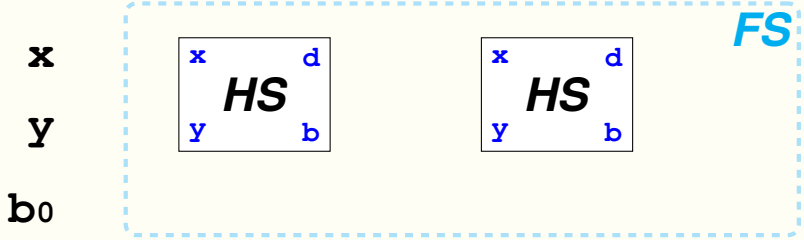


$$d = x \oplus y,\ b = \overline{x} \cdot y$$

FS はこうやっても作れる!

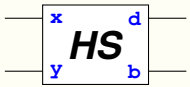


$$d = x \oplus y \oplus b_0,\ b = \left(\overline{x \oplus y}\right) \cdot b_0 + \overline{x} \cdot y$$



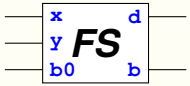
FS の作り方・別解

(ダメだと思っていた)HS がここにきて再登場

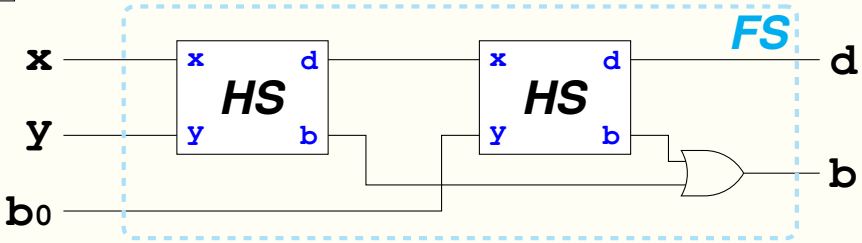


$$d = x \oplus y,\ b = \overline{x} \cdot y$$

FS はこうやっても作れる!



$$d = x \oplus y \oplus b_0,\ b = \left(\overline{x \oplus y}\right) \cdot b_0 + \overline{x} \cdot y$$



もう一つの冴えたやり方 (減算器)

復習

2 進数で $a - b$ の計算は、

$$a + (\qquad\qquad\qquad)$$

のように、足し算として計算できる。

もう一つの冴えたやり方 (減算器)

復習

2 進数で $a - b$ の計算は、

$$a + (b\text{の} \qquad\qquad)$$

のように、足し算として計算できる。

もう一つの冴えたやり方 (減算器)

復習

2 進数で $a - b$ の計算は、

$$a + (b\text{の}\mathbf{2\ \text{の補数}})$$

のように、足し算として計算できる。

もう一つの冴えたやり方 (減算器)

復習

2 進数で $a - b$ の計算は、

$$a + (b\text{の}\mathbf{2\ \text{の補数}})$$

のように、足し算として計算できる。

- 2 の補数の作り方:

もう一つの冴えたやり方 (減算器)

復習

2 進数で $a - b$ の計算は、

$$a + (b\text{の}\mathbf{2\ \text{の補数}})$$

のように、足し算として計算できる。

- 2 の補数の作り方: **1 の補数に 1 を足す**

もう一つの冴えたやり方 (減算器)

復習

2 進数で $a - b$ の計算は、

$$a + (b\text{の}\mathbf{2\ \text{の補数}})$$

のように、足し算として計算できる。

- 2 の補数の作り方: **1 の補数に 1 を足す**
- 1 の補数の作り方:

もう一つの冴えたやり方 (減算器)

復習

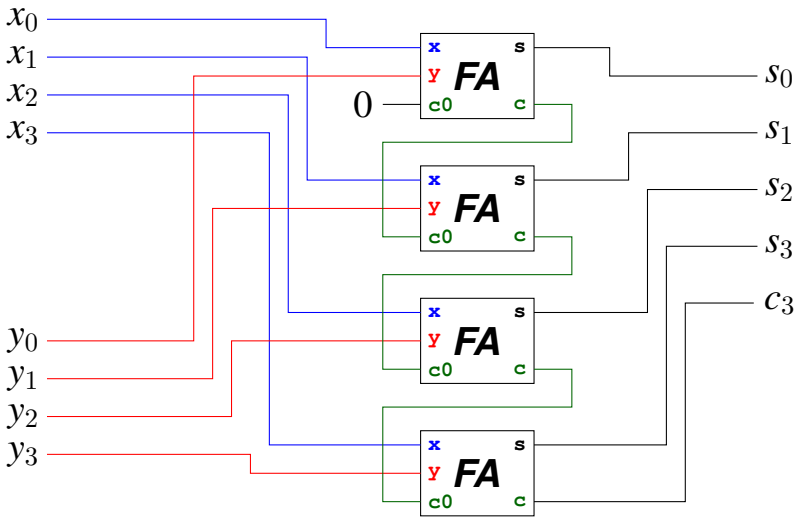
2 進数で $a - b$ の計算は、

$$a + (b\text{の}\mathbf{2\ \text{の補数}})$$

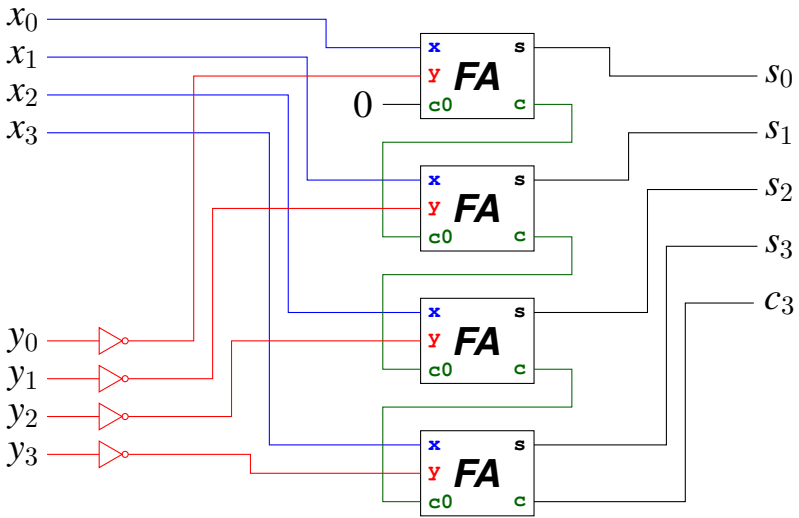
のように、足し算として計算できる。

- 2 の補数の作り方: **1 の補数に 1 を足す**
- 1 の補数の作り方: **ビット反転**

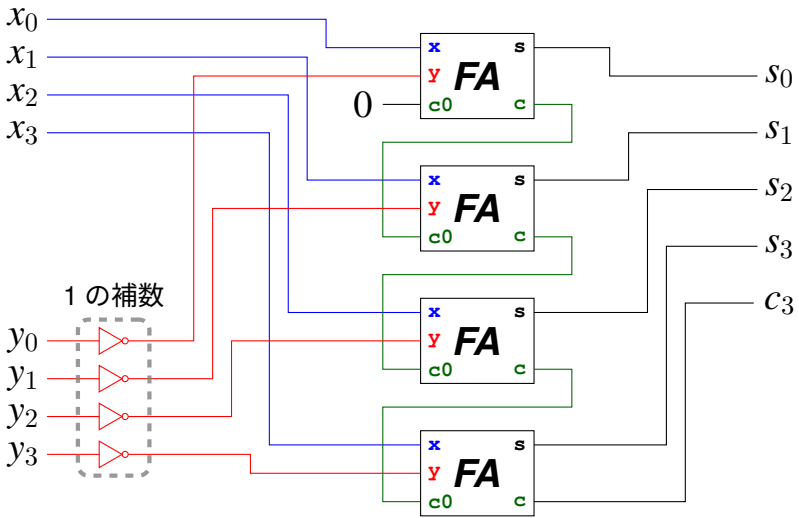
加算器で作る減算器



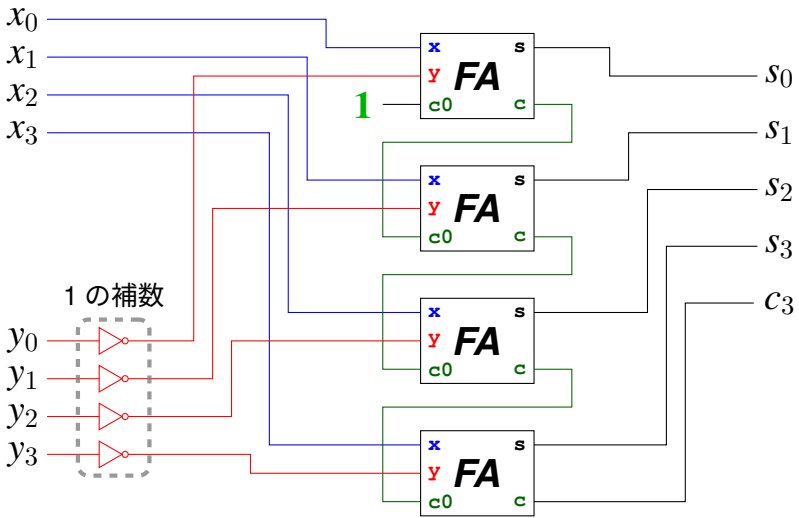
加算器で作る減算器



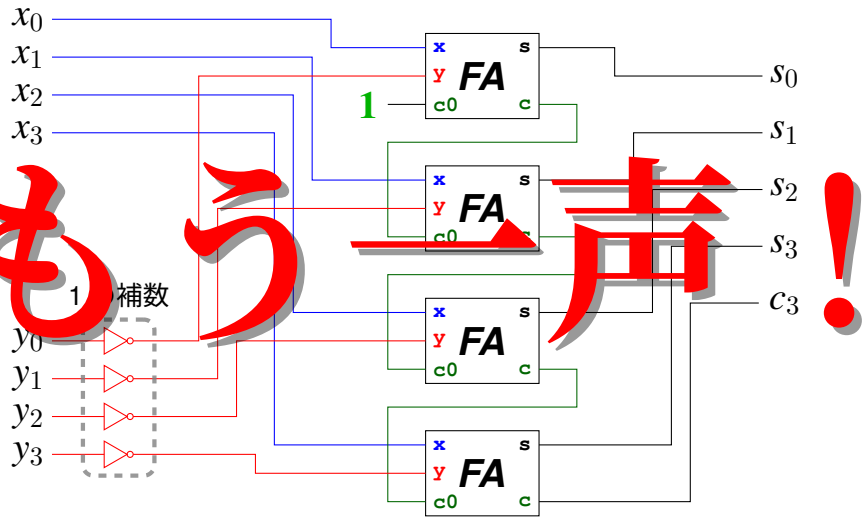
加算器で作る減算器



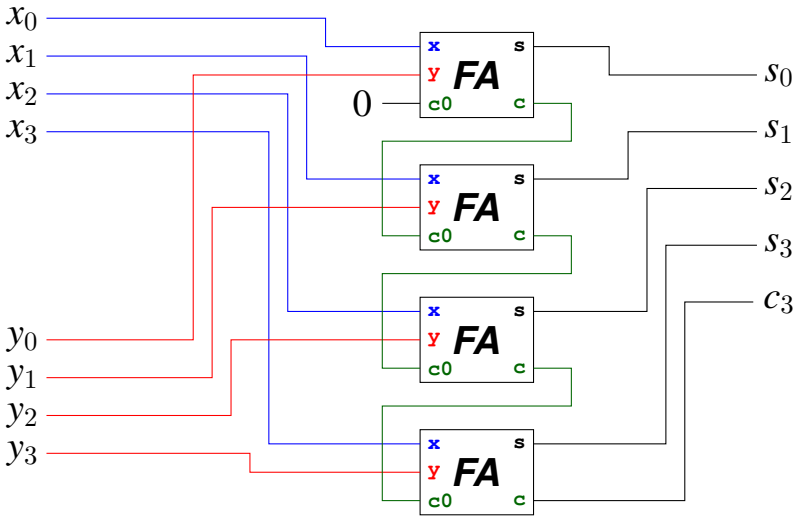
加算器で作る減算器



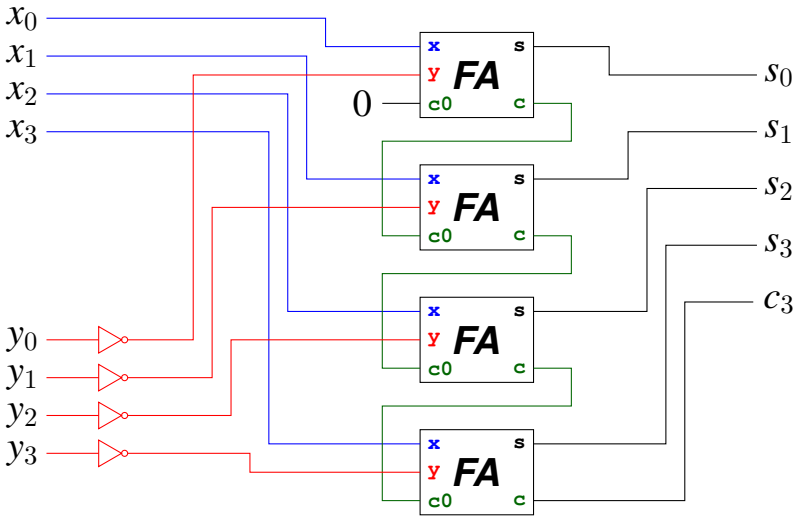
加算器で作る減算器



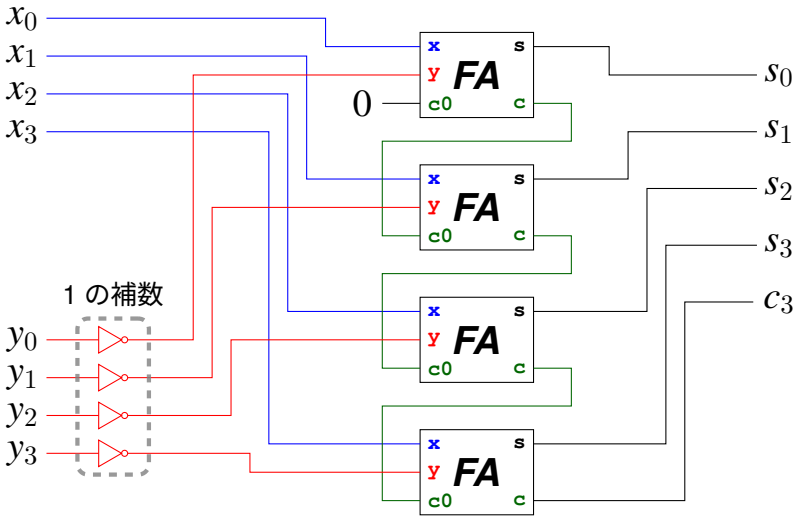
加算器で作る加減算器



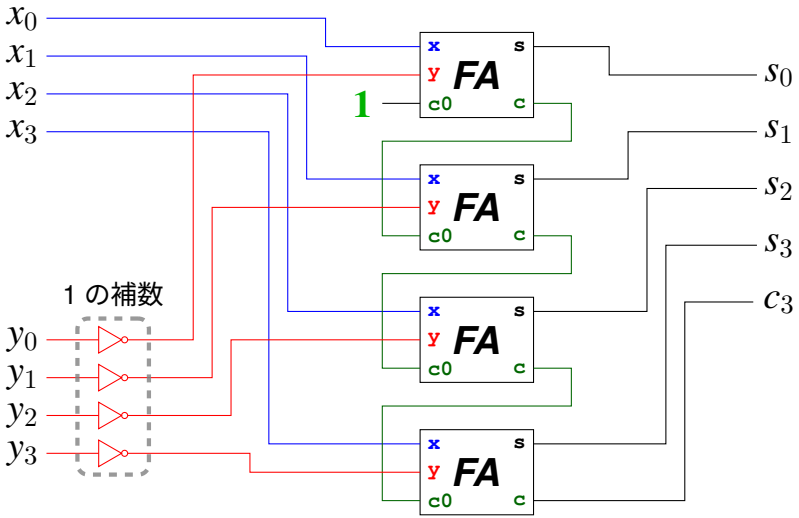
加算器で作る加減算器



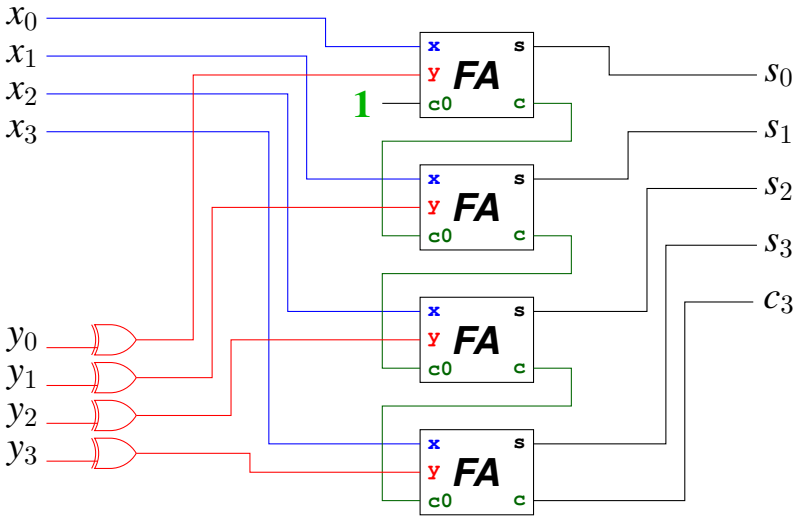
加算器で作る加減算器



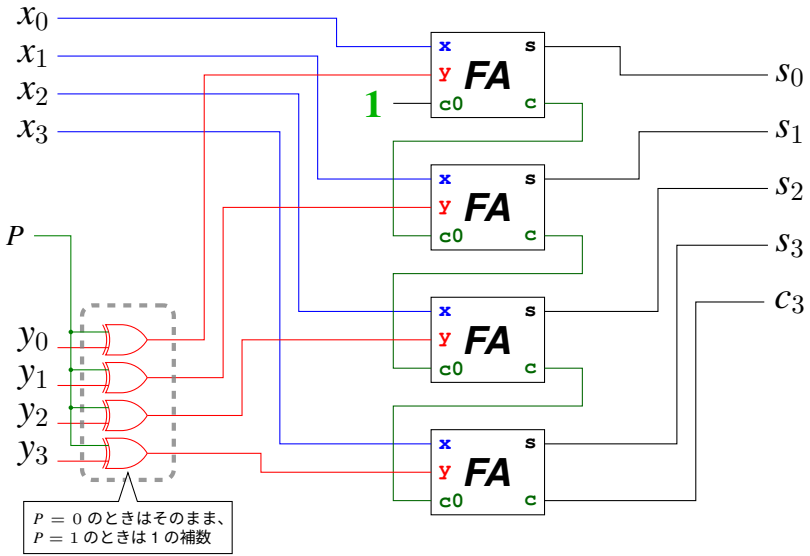
加算器で作る加減算器



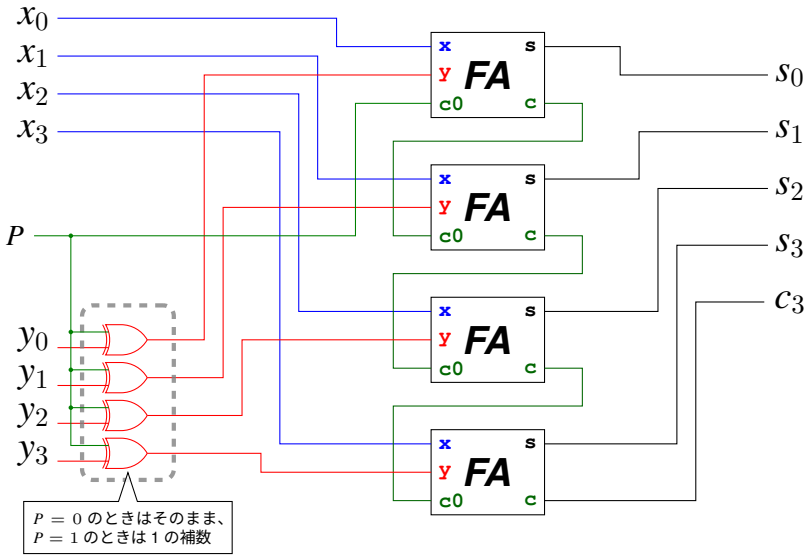
加算器で作る加減算器



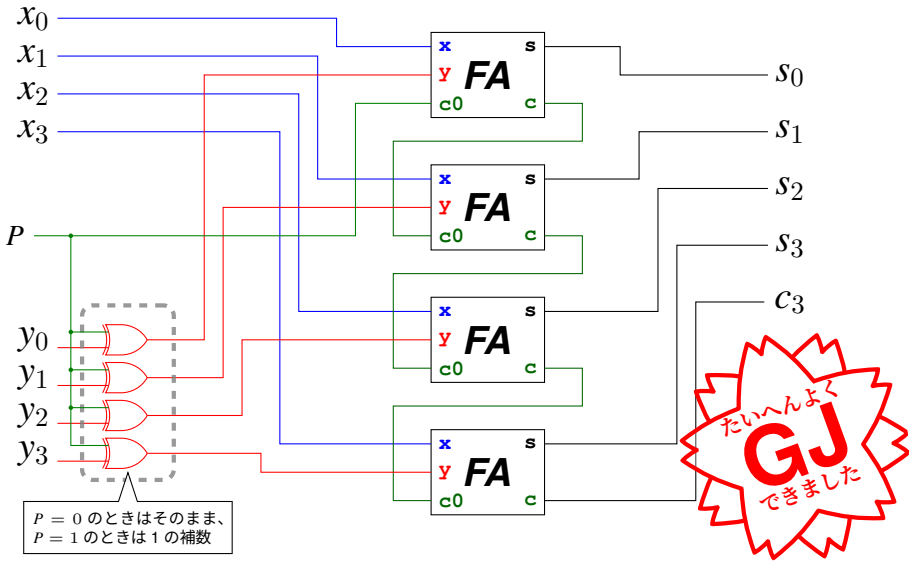
加算器で作る加減算器


x_0
x_1
x_2
x_3
P
y_0
y_1
y_2
y_3
1
x
y
c0
s
c
FA
s_0
s_1
s_2
s_3
c_3
$P = 0$ のときはそのまま、
$P = 1$ のときは 1 の補数

加算器で作る加減算器


x_0
x_1
x_2
x_3
P
y_0
y_1
y_2
y_3
x
y
c0
s
c
FA
s_0
s_1
s_2
s_3
c_3
$P = 0$ のときはそのまま、
$P = 1$ のときは 1 の補数

加算器で作る加減算器


x_0
x_1
x_2
x_3
P
y_0
y_1
y_2
y_3
x
y
c0
s
c
FA
s_0
s_1
s_2
s_3
c_3
$P = 0$ のときはそのまま、
$P = 1$ のときは 1 の補数
たいへんよく
GJ
できました

出席確認レポート課題 (次の月曜の 12 時締め切り)

問: キャリールックアヘッド方式の $CLA_2 = (x_1 \cdot y_1) + (x_1 \cdot x_0 \cdot y_0) + (x_0 \cdot y_1 \cdot y_0)$ を導出せよ。以下をすべて記述すること。

- 言葉による説明
- 真理値表
- カルノー図などによる簡単化のプロセス

提出は下記 URL の Google Forms。歪んでいない、開いた時に横倒しになっていない、コントラストが読むに耐えうる PDF で提出すること。

https://forms.gle/9ruwtfJg5LQgQNpU7

Epilogue: CPU の機械語命令

プログラミング言語と機械語

コンパイルの正体見たり

こんな**C 言語**のコード (注: 話を単純化するための無意味なコードです。) をコンパイルすると…

```
int a, b, c;
b = a + c;
b = a - c;
```

こんな**アセンブリ言語**のコードになり、さらにアセンブル・リンクを経て、

```
mov   eax, [ebp-0x10]
add   eax, [ebp-0x18]
mov   [ebp-0x14], eax
mov   eax, [ebp-0x10]
sub   eax, [ebp-0x18]
mov   [ebp-0x14], eax
```

このような数値列になる。これが CPU が本来実行可能な**機械語**。

```
8B 45 F0
03 45 E8
89 45 EC
8B 45 F0
2B 45 F8
89 45 EC
```

- これは Intel 8086 という 16-bit CPU の例。
- `eax` や `ebp` というのが**レジスタ**。
- 変数 `a` の値は**アドレス**が `[ebp-0x10]` のメモリに格納されている。
- $(\mathrm{F0})_{16}$, $(\mathrm{E8})_{16}$, $(\mathrm{EC})_{16}$ はそれぞれ $(-10)_{16}$, $(-18)_{16}$, $(-14)_{16}$ の**16 の補数表現**。

2 進数で見る加算・減算命令の例

8086 でも十分複雑なので単純な 8-bit CPU で……。

6502 の **ADC #** (加算命令)	0	1	1	0	1	0	0	1
6502 の **SBC #** (減算命令)	1	1	1	0	1	0	0	1
Z80 の **ADD A, n** (加算命令)	1	1	0	0	0	1	1	0
Z80 の **SUB A, n** (減算命令)	1	1	0	1	0	1	1	0

加減算回路の P 入力に機械語命令が関与している雰囲気[1]は感じられるかな？

[1]実際の CPU はこのあとにもいろいろなビット列の変換があるのでこのまま単純に演算回路に直結するわけではありません。