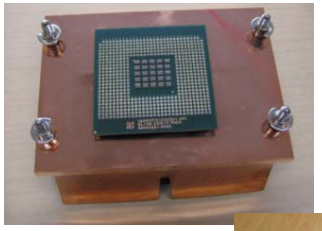
Course number: CSC.T341

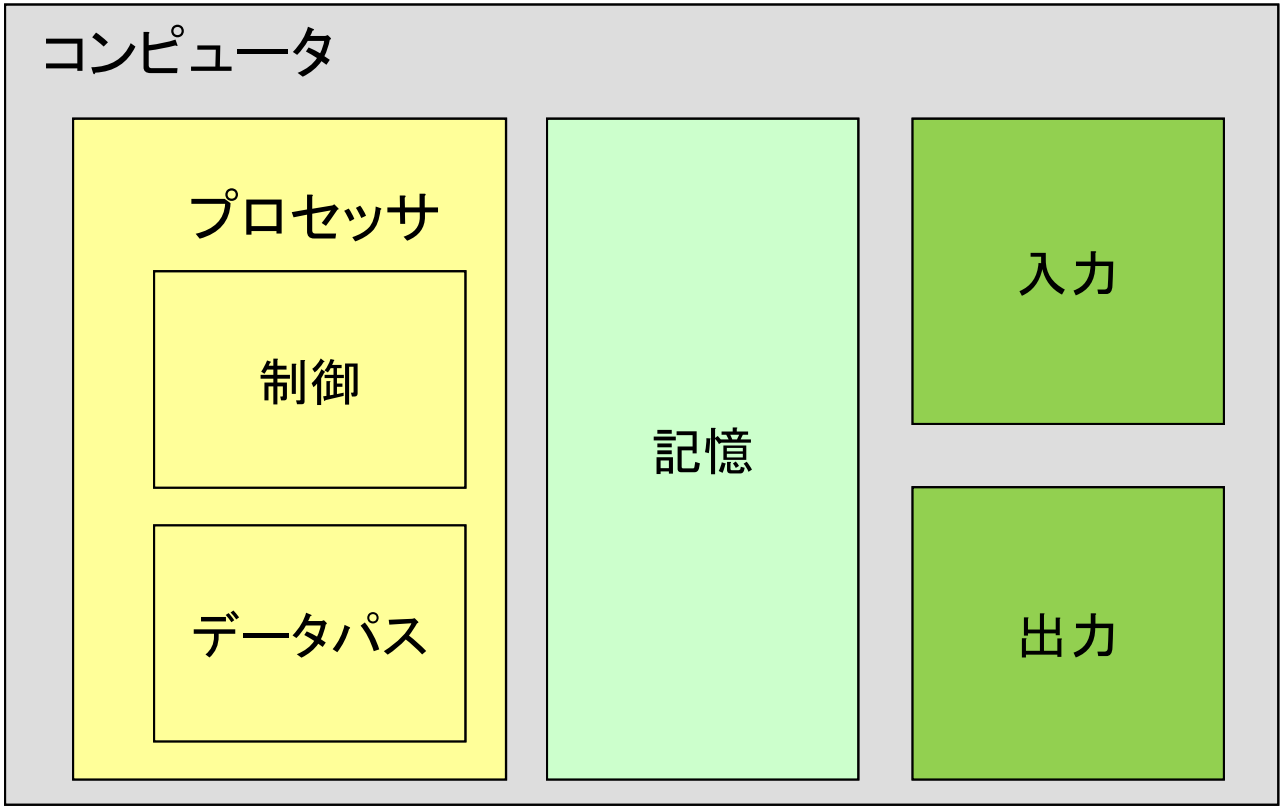
コンピュータ論理設計
Computer Logic Design

6. 命令セットアーキテクチャ:データ表現とアドレス指定形式
Instruction Set Architecture: Data Representation and Addressing

吉瀬 謙二　情報工学系
Kenji Kise, Department of Computer Science
kise _at_ c.titech.ac.jp　www.arch.cs.titech.ac.jp/lecture/CLD/
W621 講義室　　月 10:45-12:15, 木 9:00-12:15

コンピュータ(ハードウェア)の古典的な要素



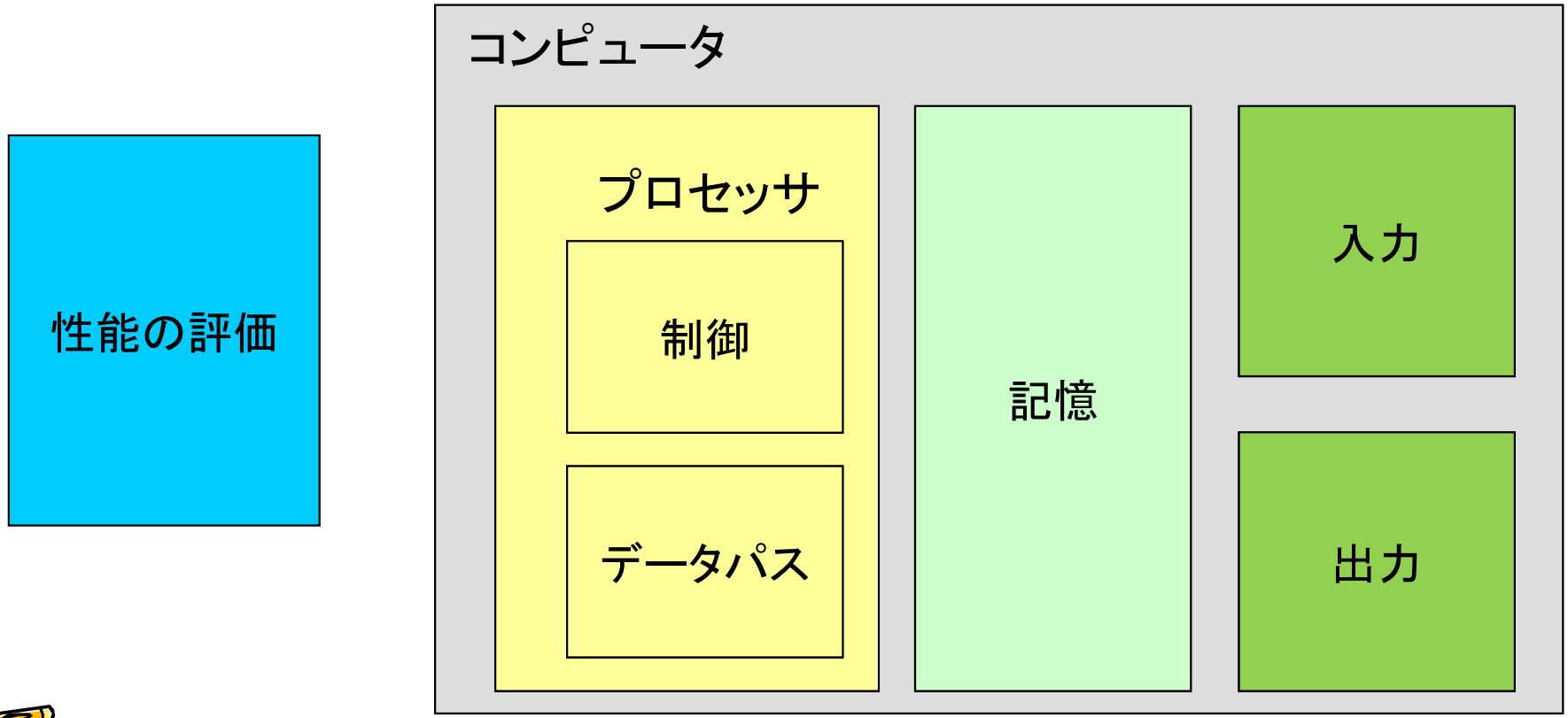


プロセッサは記憶装置から命令とデータを取り出す。入力装置はデータを記憶装置に書き込む。出力装置は記憶装置からデータを読みだす。制御装置は、データパス、記憶装置、入力装置、そして出力装置の動作を指定する信号を送る。

コンピュータの古典的な要素

コンパイラ

インタフェース　Instruction Set Architecture (ISA), 命令セットアーキテクチャ

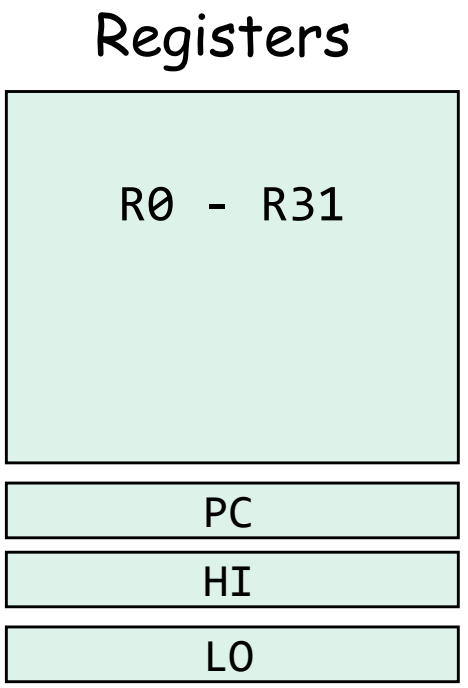


RISC vs CISC

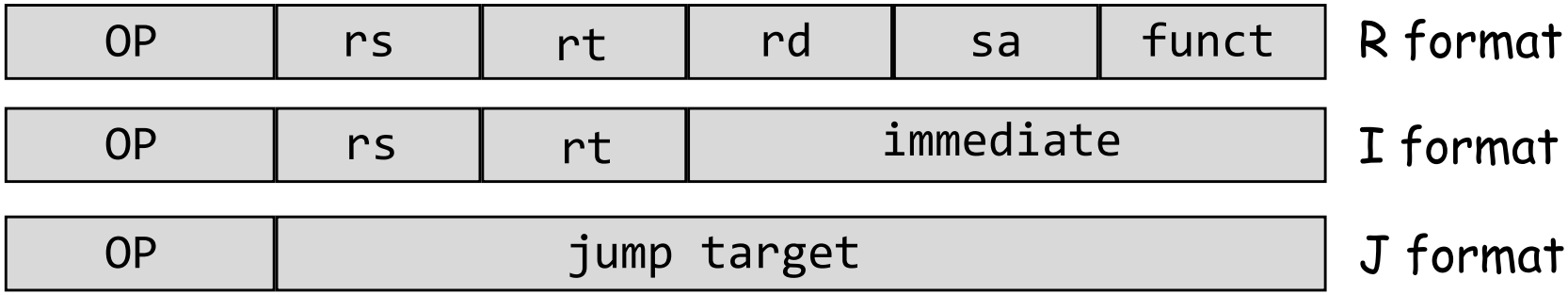
- RISC (Reduced Instruction Set Computer) philosophy
  - fixed instruction lengths
  - load-store instruction sets
  - limited addressing modes
  - limited operations
  - RISC: MIPS, Alpha, ARM, …
- CISC (Complex Instruction Set Computer) philosophy
  - ! fixed instruction lengths
  - ! load-store instruction sets
  - ! limited addressing modes
  - ! limited operations
  - CISC : DEC VAX11, Intel 80x86, …

MIPS R3000 Instruction Set Architecture (ISA)

- Instruction Categories
  - Computational
  - Load/Store
  - Jump and Branch
  - Floating Point
    - coprocessor
  - Memory Management
  - Special



3 Instruction Formats: all 32 bits wide



MIPS Register Convention, ABI(Application Binary Interface)

Name	Register Number	Usage	Preserve on call?
$zero	0	The Constant Value 0	N.A.
$at	1	Assembler Temporary	No
$v0 - $v1	2-3	Returned Values	No
$a0 - $a3	4-7	Arguments	No
$t0 - $t7	8-15	Temporaries	No
$s0 - $s7	16-23	Saved Temporaries	Yes
$t8 - $t9	24-25	Temporaries	No
$k0 - $k1	26-27	Reserved for OS Kernel	No
$gp	28	Global Pointer	Yes
$sp	29	Stack Pointer	Yes
$fp	30	Frame Pointer	Yes
$ra	31	Return Address	Yes

MIPS Arithmetic Instructions

- MIPS assembly language arithmetic statement

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

- Each arithmetic instruction performs only one operation
- Each arithmetic instruction fits in 32 bits and specifies exactly three operands

```
destination <- source1 op source2
```

- Operand order is fixed (destination first)
- Those operands are all contained in the datapath's register file ($t0,$s1,$s2) – indicated by $

Example（例題）

- `f = ( g + h ) - ( i + j )`

  f, g, h, i, j をそれぞれレジスタ $s0, $s1, $s2, $s3, $s4 に割り付けるとする．上のステートメントをコンパイルした結果のMIPSコードはどうなるか．

Answer

- `f = ( g + h ) - ( i + j )`

  f, g, h, i, j をそれぞれレジスタ $s0, $s1, $s2, $s3, $s4 に割り付けるとする. 上のステートメントをコンパイルした結果のMIPSコードはどうなるか.

```
add $t0, $s1, $s2  # $t0 = (g + h)
add $t1, $s3, $s4  # $t1 = (i + j)
sub $s0, $t0, $t1  # f    = $t0 - $t1
```

Machine Language - Add Instruction

- Instructions, like registers and words of data, are 32 bits long
- Arithmetic Instruction Format (R format):

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

R format

op 6-bits *opcode* that specifies the operation

rs 5-bits register file address of the first source operand

rt 5-bits register file address of the second source operand

rd 5-bits register file address of the result's destination

shamt 5-bits shift amount (for shift instructions)

funct 6-bits function code augmenting the opcode

Example（例題）

- 次のMIPS命令列の機械語コードはどうなるか．2進数と16進数で示せ．

```
add $t0, $s1, $s2   # $t0 = (g + h)
add $t1, $s3, $s4   # $t1 = (i + j)
sub $s0, $t0, $t1   # f    = $t0 - $t1
```

Answer

- 次のMIPS命令列の機械語コードはどうなるか．2進数と16進数で示せ．

```
add $t0, $s1, $s2   # $t0 = (g + h)
add $t1, $s3, $s4   # $t1 = (i + j)
sub $s0, $t0, $t1   # f    = $t0 - $t1
```

add $t0, $s1, $s2


```
module m_top();
  reg [31:0] r_i1, r_i2, r_i3;
  initial begin
    r_i1 = {6'h0, 5'd17, 5'd18, 5'd8,  5'h0, 6'h20};
    r_i2 = {6'h0, 5'd19, 5'd20, 5'd9,  5'h0, 6'h20};
    r_i3 = {6'h0, 5'd8,  5'd9,  5'd16, 5'h0, 6'h22};
    $write("i1: %b %x¥n", r_i1, r_i1);
    $write("i2: %b %x¥n", r_i2, r_i2);
    $write("i3: %b %x¥n", r_i3, r_i3);
  end
endmodule
```

```
i1: 00000010001100100100000000100000 02324020
i2: 00000010011101000100100000100000 02744820
i3: 00000001000010011000000000100010 01098022
```

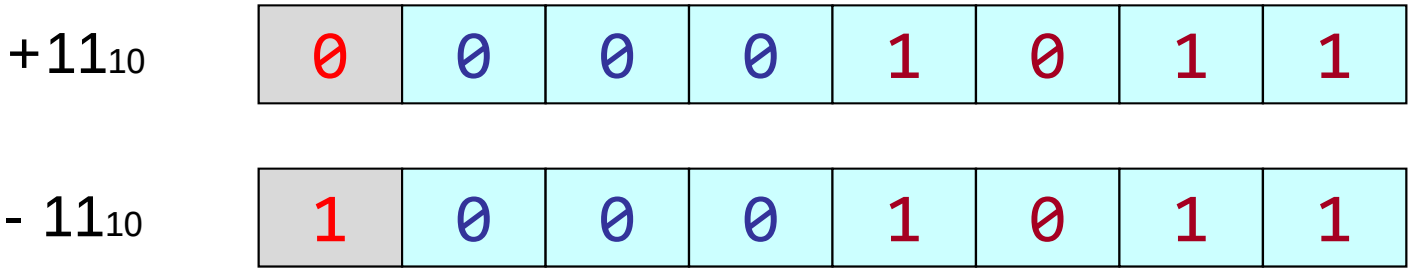
Integer（整数）Representation

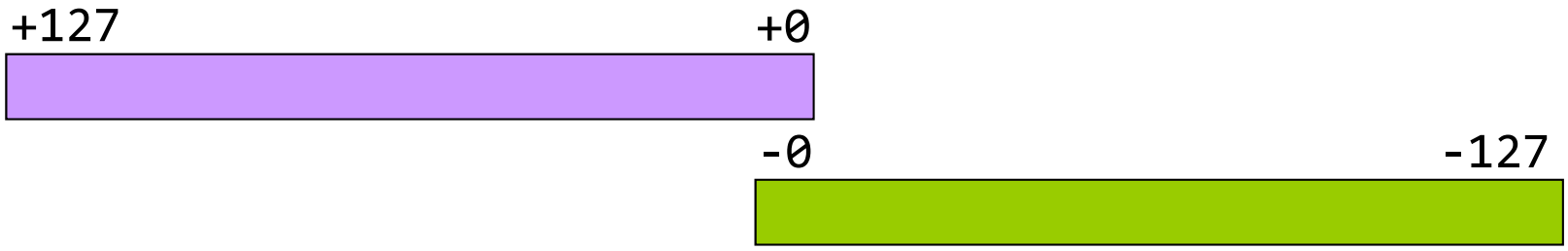
- 符号なし数，負数を表現できない．
- 整数を2進数で表現する．
  - 例えば，11_{10} であれば，1011_2 として下位ビットを決める．
  - 上位の残ったビットを0で埋める．
  - 8ビットであれば，0～255 の256個の整数を表現できる．

0	0	0	0	1	0	1	1

Integer（整数）Representation

- 符号つき絶対値 (sign and magnitude) 表現
  - 11_{10} であれば，1011_2 として下位ビットを決める．
  - ただし，最上位ビットを用いて符号を表す（符号ビット）．符号ビットが0であれば正数，1であれば負数とする．
  - 残ったビットを0で埋める．
  - 8ビットであれば，- 127 ～127 までの 255個の整数を表現できる





Integer（整数）Representation

- 2の補数 (two's complement) による符号付き数の表現
  - 正数のビットを反転させ，1を加えたものを負数とする．
  - 8ビットであれば，- 128 ～127 までの 255個の整数を表現できる

$0000\ 0000_2 = +0_{10}$	$1111\ 1111_2 = -0_{10}$	
$0000\ 0001_2 = +1_{10}$	$1111\ 1110_2 = -1_{10}$	$1111\ 1111_2 = -1_{10}$
$0000\ 0010_2 = +2_{10}$	$1111\ 1101_2 = -2_{10}$	$1111\ 1110_2 = -2_{10}$
…	…	…
$0111\ 1101_2 = +125_{10}$	$1000\ 0010_2 = -125_{10}$	$1000\ 0011_2 = -125_{10}$
$0111\ 1110_2 = +126_{10}$	$1000\ 0001_2 = -126_{10}$	$1000\ 0010_2 = -126_{10}$
$0111\ 1111_2 = +127_{10}$	$1000\ 0000_2 = -127_{10}$	$1000\ 0001_2 = -127_{10}$
		$1000\ 0000_2 = -128_{10}$
0～127の正数	0～127の正数のビット反転（1の補数表現）	ビット反転に1を加えて得る負数

Integer（整数）Representation

- 2の補数 (two's complement) 表現の特徴
  - 全てのビットを反転させて1を加えると，正負が反転する．
  - 最上位ビットは符号ビットとよばれ，0であれば正数，1であれば負数．
  - 符号拡張(sign extension)と呼ばれるビット長を増やす処理は，符号ビットを複製して補填すればよい．

$$x + \overline{x} = -1$$

$$\overline{x} + 1 = -x$$

Example（例題）

- 16ビットの2進数の 2_{10} と -2_{10} を32ビットの2進数に変換せよ．これらは2の補数で表現されているとする．

Answer

- 16ビットの2進数の 2_{10} と -2_{10} を32ビットの2進数に変換せよ. これらは2の補数で表現されているとする.

16ビットの2進数の 2_{10} 0000 0000 0000 0010

16ビットの2進数の -2_{10} 1111 1111 1111 1110

32ビットの2進数の 2_{10} 0000 0000 0000 0000 0000 0000 0000 0010

32ビットの2進数の -2_{10} 1111 1111 1111 1111 1111 1111 1111 1110

Verilog HDLで2の補数表現として表示

- ワイヤ型の信号の定義を wire signed とすることで，2の補数表現の整数として扱われる．

```
module m_top();
  reg [15:0] r_data = 16'b1111111111111110;
  wire signed [15:0] w_data = r_data;

  initial #1 begin
    $write("%6d¥n", r_data);
    $write("%6d¥n", w_data);
  end
endmodule
```

```
 65534
    -2
```

Verilog HDLで2の補数表現の符号拡張

- 符号拡張(sign extension)と呼ばれるビット長を増やす処理は，符号ビットを複製して補填すればよい．
- 2の補数表現の16ビットの整数を32ビットの整数に変換する例

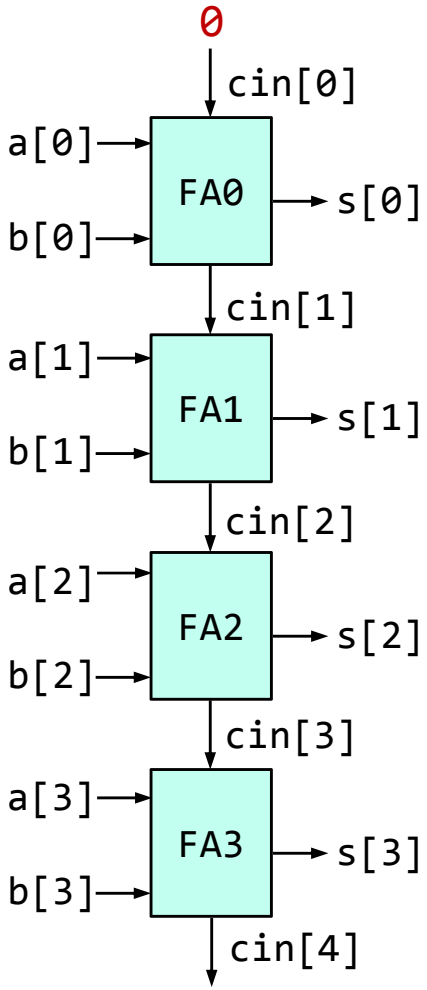
```
module m_top();
  wire signed [15:0] w_data1 = 16'b1111111111111110;
  wire signed [31:0] w_data2 = {{16{w_data1[15]}}, w_data1};

  initial #1 begin
    $write(“%5d %32b¥n", w_data1, w_data1);
    $write("%5d %32b¥n", w_data2, w_data2);
  end
endmodule
```

```
   -2                  1111111111111110
   -2 11111111111111111111111111111110
```

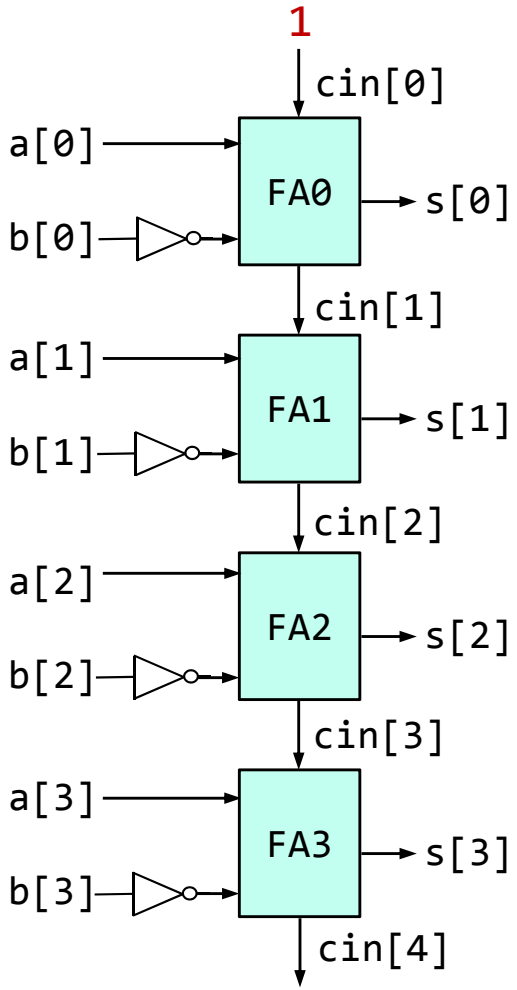
Adder and Subtractor

- 4-bit Ripple Carry Adder

s = a + b



s = a - b

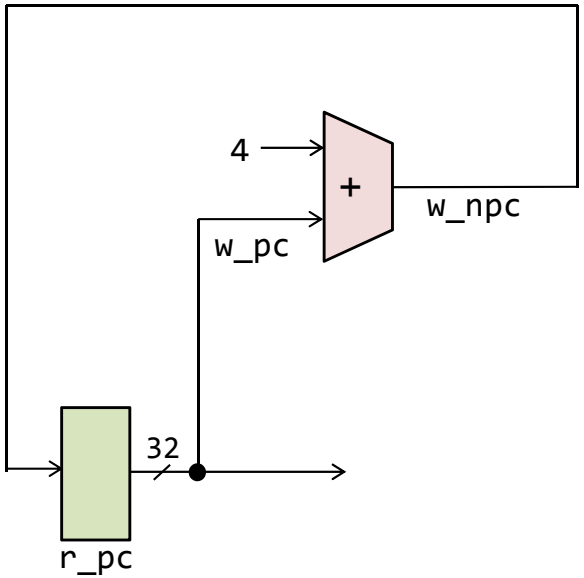


References

- Computer Logic Design support page
  - http://www.arch.cs.titech.ac.jp/lecture/CLD/
- 情報工学系計算機室
  - http://www.csc.titech.ac.jp/
- Xilinx Vivado Design Suite
  - https://japan.xilinx.com/products/design-tools/vivado.html
- Digilent Nexys 4 DDR Atrix-7 FPGA
  - https://reference.digilentinc.com/reference/programmable-logic/nexys-4-ddr/start
  - https://store.digilentinc.com/nexys-4-ddr-artix-7-fpga-trainer-board-recommended-for-ece-curriculum/
- Verilog HDL
  - https://ja.wikipedia.org/wiki/Verilog
- Frix (Feasible and Reconfigurable IBM PC Compatible SoC)
  - http://www.arch.cs.titech.ac.jp/a/Frix/
- 7シリーズFPGAデータシート：概要
  - https://japan.xilinx.com/support/documentation/data_sheets/j_ds180_7Series_Overview.pdf
- 7シリーズFPGA コンフィギャラブルロジックブロック ユーザーガイド
  - https://japan.xilinx.com/support/documentation/user_guides/j_ug474_7Series_CLB.pdf

Sample Circuit 1

```
module m_main (w_clk, w_pc);
  input  wire w_clk;
  output wire [31:0] w_pc;
  reg [31:0] r_pc = 0;

  assign w_pc = r_pc;
  wire [31:0] w_npc = w_pc + 4;
  always@(posedge w_clk) r_pc <= w_npc;
endmodule
```



Sample Circuit 2

```
module m_main (w_clk, w_ifpc);
  input  wire w_clk;
  output wire [31:0] w_ifpc;
  reg [31:0] r_pc = 0;
  reg [31:0] r_ifpc = 0;

  assign w_pc = r_pc;
  assign w_ifpc = r_ifpc;
  wire [31:0] w_npc = w_pc + 4;
  always@(posedge w_clk) r_pc   <= w_npc;
  always@(posedge w_clk) r_ifpc <= w_pc;
endmodule
```