

2023年度版

Ver. 2023-04-13a

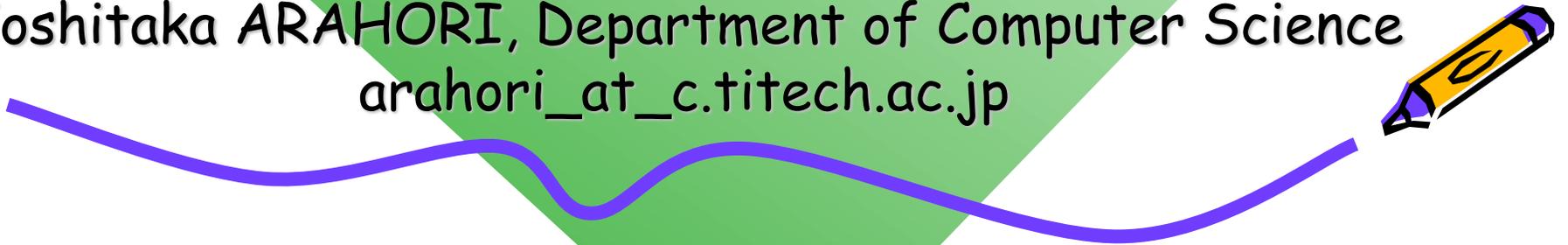


Course number: CSC.T341

コンピュータ論理設計 演習(2) Computer Logic Design Exercise(2)

情報工学系 荒堀喜貴

Yoshitaka ARAHORI, Department of Computer Science
arahori_at_c.titech.ac.jp



Computer Logic Design support page <https://www.arch.cs.titech.ac.jp/lecture/CLD/>

重要

- 演習は 8:50~10:30 です. 8:45までに学術国際情報センター3階 **情報工学系 計算機室** に集まってください.
- ACRIルームのサイトで, 演習の日の **6:00~9:00** と **9:00~12:00** の時間帯を **予約**をしてください.
- vs から始まるサーバを選択して予約すること. 予約する2つの時間帯で**同じサーバ**を予約すること.
 - 各マシンの負荷を下げるために, 仮想マシンの名前の最後の2文字が12~15は使わない. 具体的には **vs001~vs011, vs101~vs111, vs201~vs211, vs301~vs311, vs401~vs411, vs501~vs511, vs601~vs610** から選ぶこと.
- <https://gw.acri.c.titech.ac.jp/wp/>

ACRI ルームへようこそ!

ようこそ. ACRI ルームは, 100枚を超える FPGA ボードや **FPGA StarterBOX** を含むサーバ(計算機)をリモートからアクセスして利用できる FPGA 利用環境です.

[New] 日別スケジュールをリニューアルしました. ページ遷移をすることなく全てのサーバの予約状況をチェックできます. (2021-01-14)

日別スケジュール

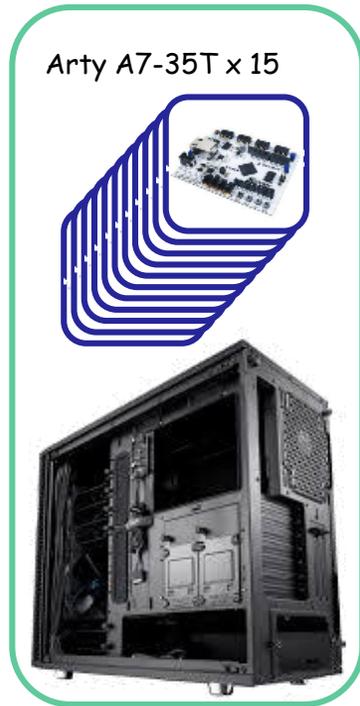
<前日> 2021-04-13 * >翌日> 移動

サーバ	vs001 (U200)	vs001 (U200)	vs002 (U250)	vs003 (U290-E51)	vs004 (U50)	vs001	vs002	vs003
00:00	Close	Close	Close	Close	Close	Close	Close	Close
03:00	Close	Close	Close	Close	Close	Close	Close	Close
06:00	Close	Close	Close	Close	Close	Close	Close	Close
09:00	Close	Close	Close	Close	Close	Close	Close	Close
12:00	Open	Open	Open	Open	Open	Open	Open	Close
15:00	Open	Open	Open	Open	Open	Open	Open	Open
18:00	Open	Open	Open	Open	Open	Open	Open	Open
21:00	Open	Open	Open	Open	Open	Open	Open	Open



ACRiルームのサーバー計算機

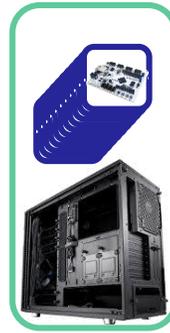
各マシンの負荷を下げるために、仮想マシンの名前の最後の2文字が12~15は使わない。
具体的には **vs001~vs011**, **vs101~vs111**, **vs201~vs211**, **vs301~vs311**, **vs401~vs411**,
vs501~vs511, **vs601~vs610** から選ぶこと。



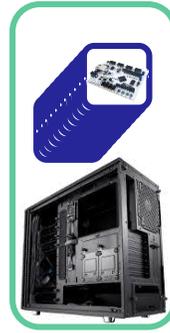
vs001~
vs015



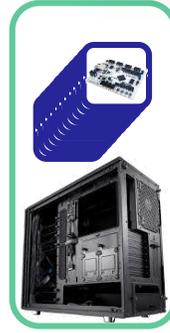
vs101~
vs115



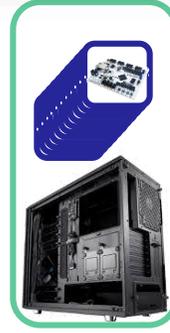
vs201~
vs215



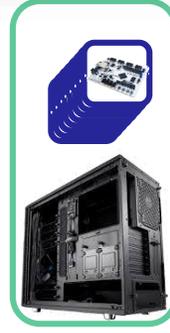
vs301~
vs315



vs401~
vs415



vs501~
vs515



vs601~
vs610



コンピュータ論理設計 演習(Exercise)の注意点

- 演習はACRiルームを利用します。
- 3~4人のグループを作成します. そのグループ内で情報を共有しながら演習を進めてください.
- 問題はグループ内で相談して解決する, あるいは, 担当のTA(Teaching Assistant)や教員に質問してください.
- 演習には出席点があります. 休まずにきちんと出席しましょう.
- 演習スライドにチェックポイントの図がある場所は, 作業を確認してもらう場所です. すべてのチェックポイントをクリアしましょう.



- 演習時間でなくてもACRiルームを利用できます. 現在は, 1日に4枠(3時間 × 4枠 = 12時間)を利用できます. 独自のハードウェア設計などに挑戦しましょう.

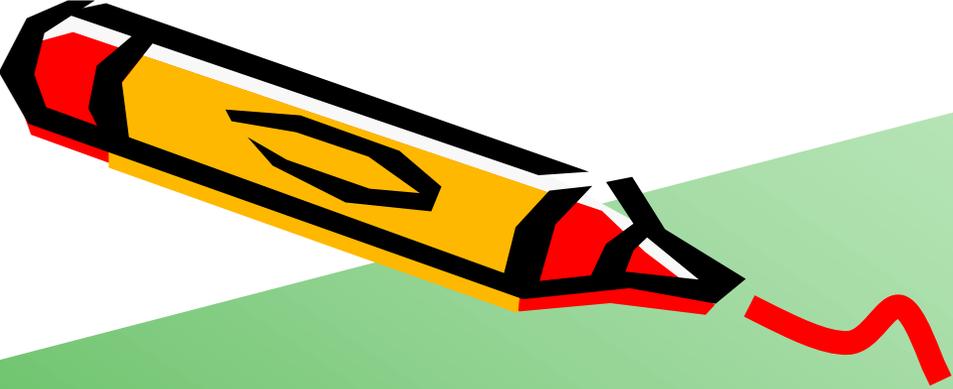


Exercise(2)

- Project_2 と Project_2b

- **加算器**をFPGAに実装して, それらの遅延を計測する.
これによって, デジタル回路がFPGAでどのくらいの動作周波数で動くか把握する.
- code073.v, code074.v, code078.v, code080.v, main11.xdc は **/home/tu_kise/cld/2023/** にあるので, 適切なフォルダにコピーして使うこと.





Project_2
Project_2b



新しい Vivado プロジェクトの作成とファイルの登録

- **前回の演習を参考**に, Vivado で新しいプロジェクト `project_2` を作成する.
- Ubuntu で起動したターミナルで, 次のコマンドを実行してファイルをコピーする.
 - `/home/tu_kise` は automount のディレクトリなので, アクセスしないとファイルが見えない. `tab`キーによる補完がうまく動作しないことがあるので注意する.
 - 最後の `ls` コマンドで, `code073.v`, `code074.v`, `code078.v`, `main11.xdc` が表示されることを確認.

```
$ ls /home/tu_kise
$ cd ~/cld/project_2
$ cp /home/tu_kise/cld/2023/code073.v .
$ cp /home/tu_kise/cld/2023/code074.v .
$ cp /home/tu_kise/cld/2023/code078.v .
$ cp /home/tu_kise/cld/2023/main11.xdc .
$ ls
```

- Vivado で, `project_2` の制約ファイルとして **`main11.xdc`** を登録する.
 - **`main11.xdc` ファイルの内容を変更する必要はない.**
- Vivado で, `project_2` の Verilog HDL ファイルとして **`code078.v`** を登録する.



FPGA constraint file, XDC (Xilinx Design Constraints)

- main11.xdc の内容を Ubuntu のターミナルあるいは Vivado で確認すること。
- 拡張子が xdc のファイルは, 制約 (constraint) を与えるために利用する。
- 制約ファイル main11.xdc の1行目では, w_clk という信号を E3 というピン (100MHzのクロック信号) に割り当てる制約を追加する。
 - w_clk は論理合成のためのトップモジュール m_main として Verilog HDL 記述で列挙した信号名
- 信号をピンに割り当てる制約が無い場合, その信号は Vivado によって自動的に適切なピンに割り当てられる。
- 2行目で入力ピン w_clk が, 10.00ns (100MHz) のクロックであることを指定する。
- このピンを LVCMOS33 (low voltage CMOS 3.3V) とする制約を追加している。この制約について, 本演習では詳細を理解する必要はない。

main11.xdc の最初の2行

```
set_property -dict { PACKAGE_PIN E3 IOSTANDARD LVCMOS33 } [get_ports { w_clk }];  
create_clock -add -name sys_clk -period 10.00 [get_ports {w_clk}];
```

```
module m_main (w_clk);  
    input wire w_clk;  
    .  
    .  
    .
```

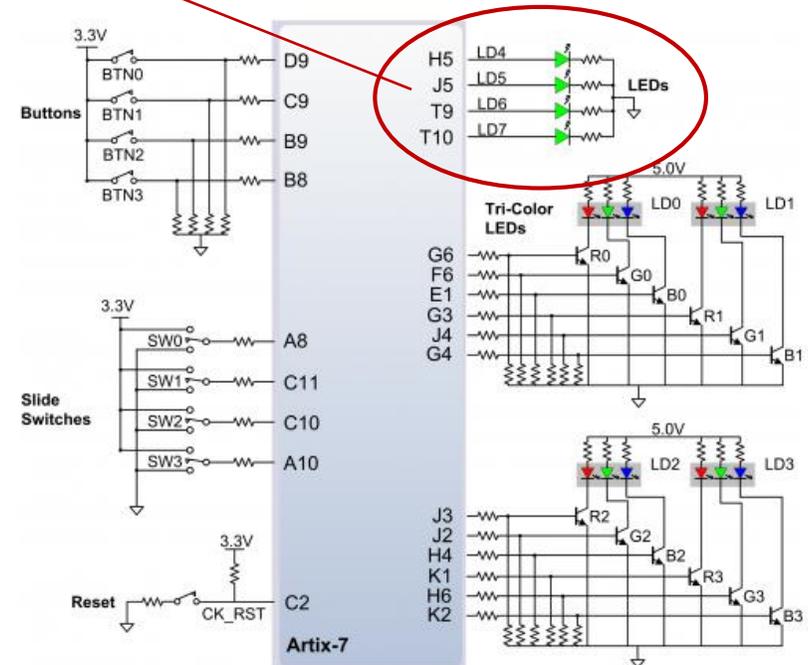
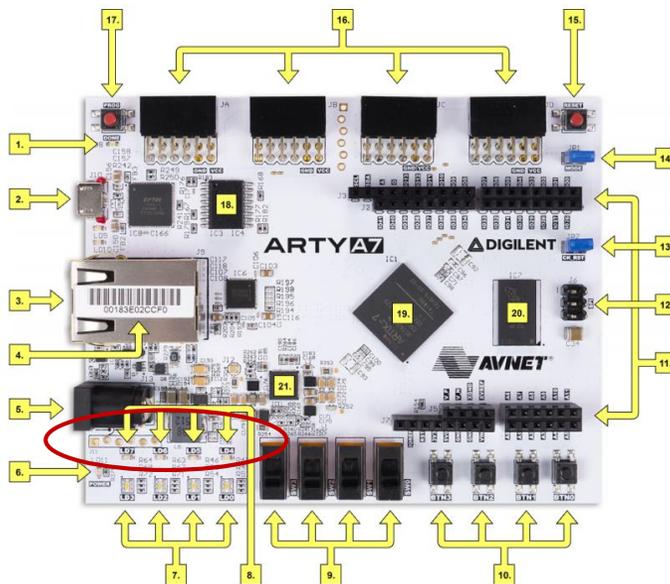
FPGA constraint file, XDC (Xilinx Design Constraints)

- main11.xdc の2行目以降では, $w_led[0]$ の信号を H5 のピンに割り当てる制約を追加する。同様に, $w_led[1]$, $w_led[2]$, $w_led[3]$ に, J5, T9, T10 のピンを割り当てる。

main11.xdc

```
set_property -dict { PACKAGE_PIN E3 IOSTANDARD LVCMOS33 } [get_ports { w_clk }];
create_clock -add -name sys_clk -period 10.00 [get_ports {w_clk}];

set_property -dict { PACKAGE_PIN H5 IOSTANDARD LVCMOS33 } [get_ports { w_led[0] }];
set_property -dict { PACKAGE_PIN J5 IOSTANDARD LVCMOS33 } [get_ports { w_led[1] }];
set_property -dict { PACKAGE_PIN T9 IOSTANDARD LVCMOS33 } [get_ports { w_led[2] }];
set_property -dict { PACKAGE_PIN T10 IOSTANDARD LVCMOS33 } [get_ports { w_led[3] }];
```



jt	Description	Callout	Description	Callout	Description
	FPGA programming DONE LED	8	User RGB LEDs	15	chipKIT processor reset

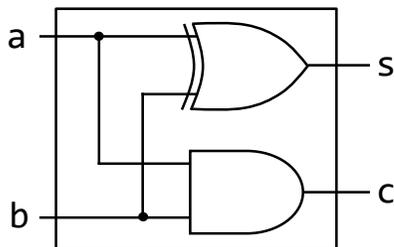
code073.v 半加算器 (Half Adder)

- code073.v をシミュレーションして, その表示を確認すること.
- Half Adder, HA (半加算器)の回路とその記述の例を示す.
 - 1ビットの入力 a, b の加算をおこなう回路.
 - 入力 a, b と出力 c (carry out), s (sum) とするtruth table(真理値表)を table073 に示す.

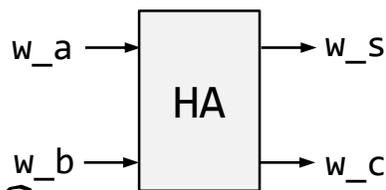
```
$ iverilog code073.v  
$ ./a.out
```

table073

a	b	c	s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0



HA



code073.v

```
`default_nettype none  
  
module m_top ();  
    reg r_a, r_b;  
    wire w_c, w_s;  
    initial begin  
        #10 r_a <= 0; r_b <= 0;  
        #10 r_a <= 0; r_b <= 1;  
        #10 r_a <= 1; r_b <= 0;  
        #10 r_a <= 1; r_b <= 1;  
    end  
    always@(*) #1  
        $write("%2d: %d %d -> %b %b\n", $time, r_a, r_b, w_c, w_s);  
    m_HA m_HA0 (r_a, r_b, w_s, w_c);  
endmodule  
  
module m_HA (w_a, w_b, w_s, w_c);  
    input wire w_a, w_b;  
    output wire w_s, w_c;  
    assign w_c = w_a & w_b;  
    assign w_s = w_a ^ w_b;  
endmodule
```

```
11: 0 0 -> 0 0  
21: 0 1 -> 0 1  
31: 1 0 -> 0 1  
41: 1 1 -> 1 0
```

code074.v 全加算器 (Full Adder)



- 全加算器として動作するように code074.v の青色の部分を変更し、シミュレーションで確認すること。
- Full Adder, FA (全加算器) の回路とその記述の一部を示す。
- 次のスライドにヒントあり。

code074.v

table074

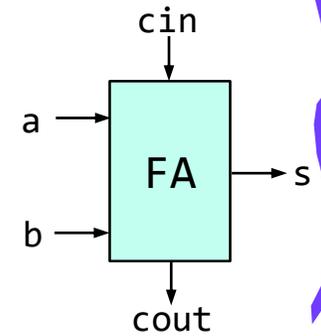
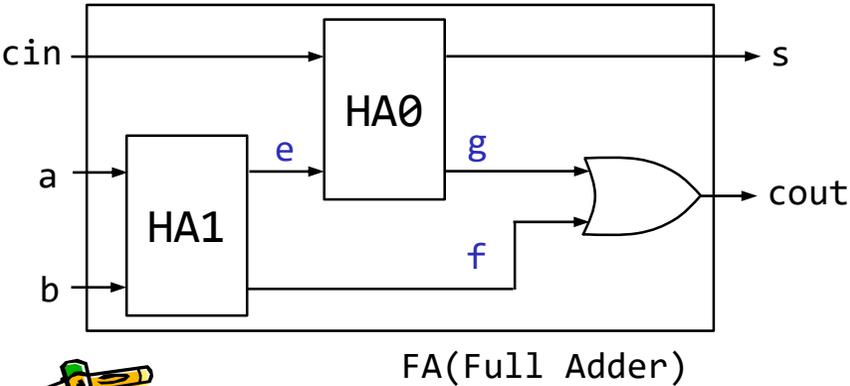
a	b	cin	cout	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

```
module m_top ();
  reg r_a, r_b, r_cin;
  wire w_s, w_cout;
  initial begin
    #10 r_a <= 0; r_b <= 0; r_cin <= 0;
    #10 r_a <= 0; r_b <= 0; r_cin <= 1;
    #10 r_a <= 0; r_b <= 1; r_cin <= 0;
    #10 r_a <= 0; r_b <= 1; r_cin <= 1;
    #10 r_a <= 1; r_b <= 0; r_cin <= 0;
    #10 r_a <= 1; r_b <= 0; r_cin <= 1;
    #10 r_a <= 1; r_b <= 1; r_cin <= 0;
    #10 r_a <= 1; r_b <= 1; r_cin <= 1;
  end
  always@(*) #1 $write("%d %d %d -> %b %b\n",
    r_a, r_b, r_cin, w_cout, w_s);
  m_FA m_FA0 (r_a, r_b, r_cin, w_s, w_cout);
endmodule

module m_FA (w_a, w_b, w_cin, w_s, w_cout);
  /* Please describe here by yourself */
endmodule

module m_HA (w_a, w_b, w_s, w_c);
  input wire w_a, w_b;
  output wire w_s, w_c;
  assign w_c = w_a & w_b;
  assign w_s = w_a ^ w_b;
endmodule
```

0	0	0	->	0	0
0	0	1	->	0	1
0	1	0	->	0	1
0	1	1	->	1	0
1	0	0	->	0	1
1	0	1	->	1	0
1	1	0	->	1	0
1	1	1	->	1	1



ヒント code074.v 全加算器 (Full Adder)



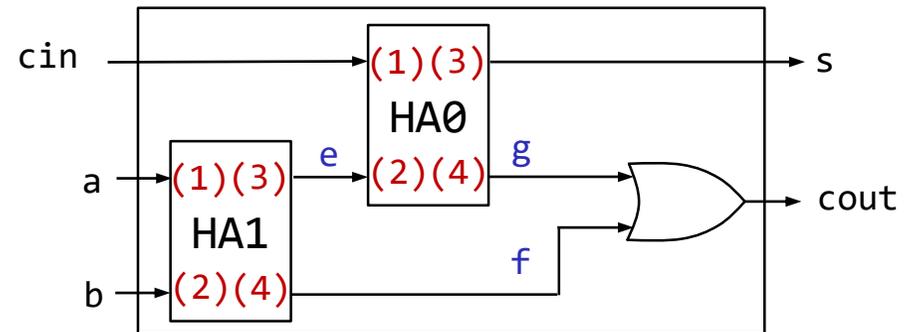
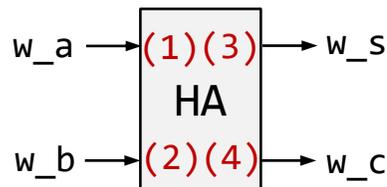
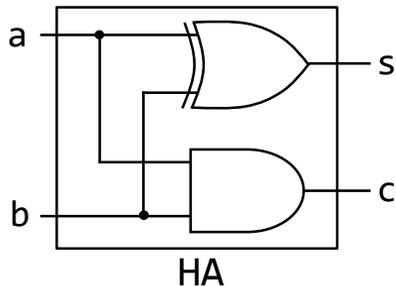
- 全加算器として動作するように code074.v の青色の部分を変更し、シミュレーションで確認すること。
- Full Adder, FA (全加算器) の回路とその記述の一部を示す。

code074.v の一部

```
module m_FA (w_a, w_b, w_cin, w_s, w_cout);  
    /* Please describe here by yourself */  
endmodule  
  
    (1) (2) (3) (4)  
module m_HA (w_a, w_b, w_s, w_c);  
    input wire w_a, w_b;  
    output wire w_s, w_c;  
    assign w_c = w_a & w_b;  
    assign w_s = w_a ^ w_b;  
endmodule
```

ヒント: 少し記述を追加した code074.v の一部

```
module m_FA (w_a, w_b, w_cin, w_s, w_cout);  
    input wire w_a, w_b, w_cin;  
    output wire w_s, w_cout;  
    wire w_e, w_f, w_g;  
    m_HA HA0 ( /* connect wires here */ );  
    m_HA HA1 ( /* connect wires here */ );  
    assign w_cout = w_f | w_g;  
endmodule  
  
module m_HA (w_a, w_b, w_s, w_c);  
    input wire w_a, w_b;  
    output wire w_s, w_c;  
    assign w_c = w_a & w_b;  
    assign w_s = w_a ^ w_b;  
endmodule
```

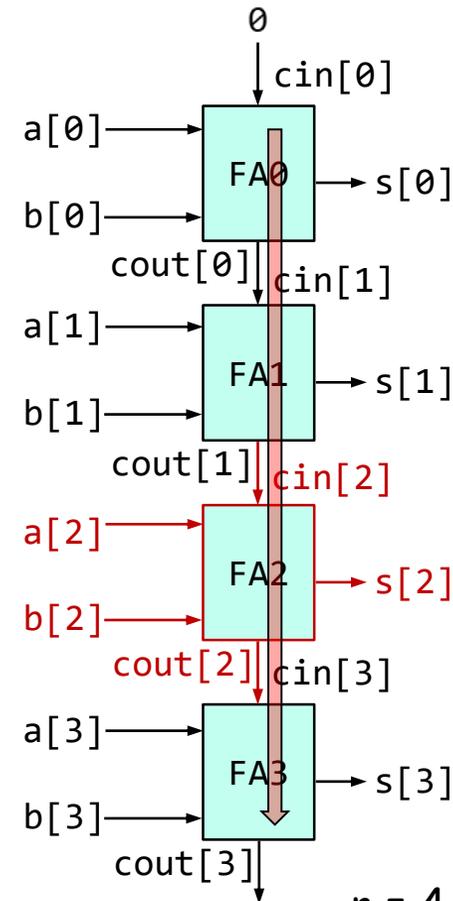
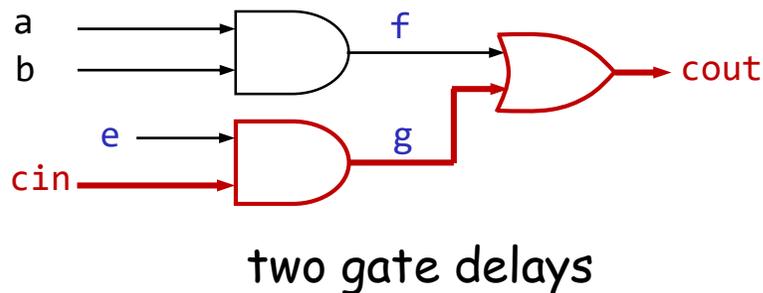
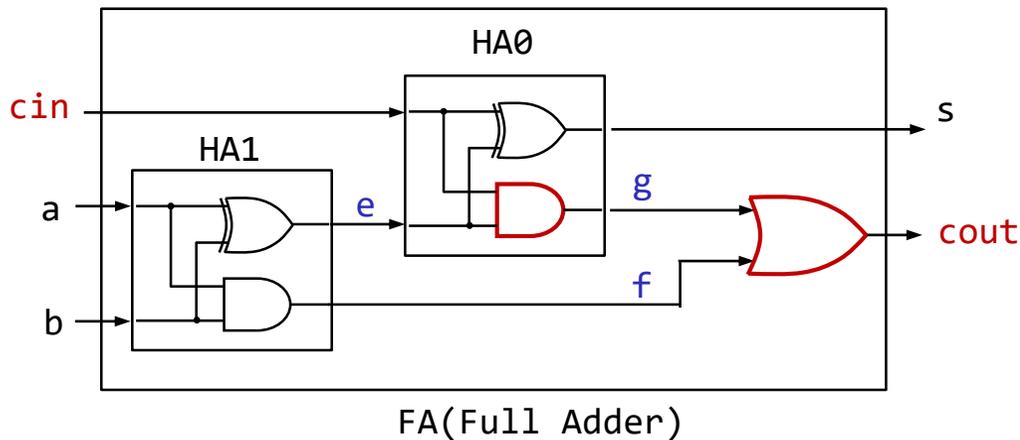


FA(Full Adder)

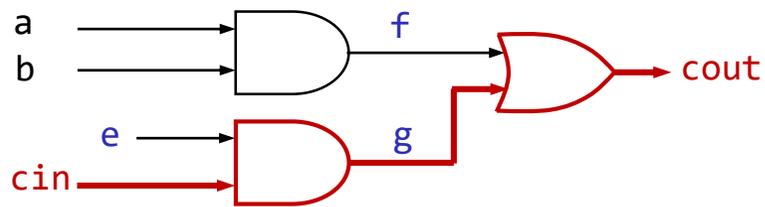
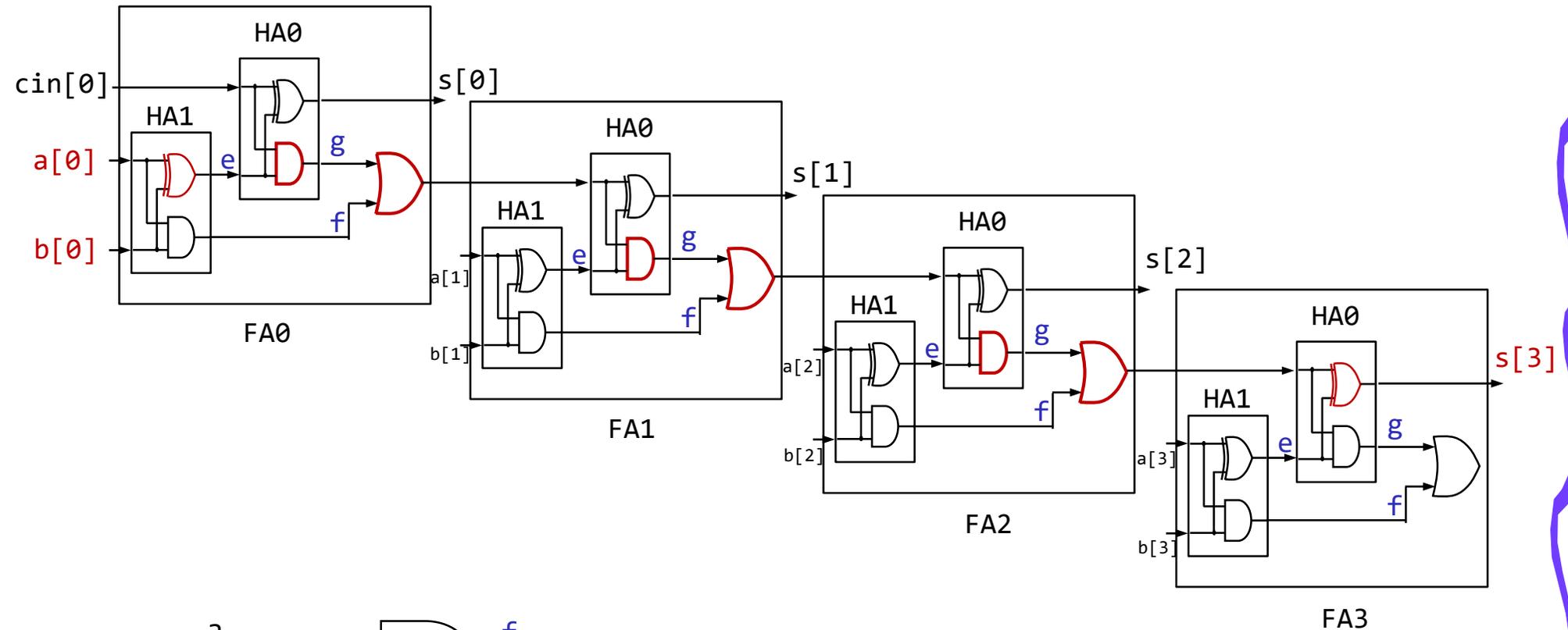


code078.v n-bit Ripple Carry Adder のクリティカルパス

- The carry out signal (w_cout) from the carry in signal (w_cin) takes two gate delays per bit.



code078.v 4-bit Ripple Carry Adder のクリティカルパス



two gate delays per bit



加算器のクリティカルパスの遅延を計測する



- code078.v の m_FA の青色の部分を、code074.v と同様に変更すること。

```
`define D_N 32

module m_main (w_clk, w_a, w_b, w_dout);
  input  wire w_clk, w_a, w_b;
  output wire w_dout;
  reg [`D_N-1:0] r_a=0, r_b=0, r_s=0;
  wire [`D_N-1:0] w_s;
  assign w_dout = ^r_s;
  always@(posedge w_clk) begin
    r_a <= {w_a, r_a[`D_N-1:1]};
    r_b <= {w_b, r_b[`D_N-1:1]};
    r_s <= w_s;
  end
  m_ADDER m_ADDER0 (r_a, r_b, w_s);
endmodule

module m_ADDER (w_a, w_b, w_s);
  input  wire [`D_N-1:0] w_a, w_b;
  output wire [`D_N-1:0] w_s;
  wire [`D_N:0] w_cin;
  assign w_cin[0] = 0;
  generate genvar g;
  for (g = 0; g < `D_N; g = g + 1) begin : Gen
    m_FA m_FA0(w_a[g], w_b[g], w_cin[g], w_s[g], w_cin[g+1]);
  end
endgenerate
endmodule

module m_FA (w_a, w_b, w_cin, w_s, w_cout);
  /* Please describe here by yourself */
endmodule

module m_HA (w_a, w_b, w_s, w_c);
  input  wire w_a, w_b;
  output wire w_s, w_c;
  assign w_c = w_a & w_b;
  assign w_s = w_a ^ w_b;
endmodule
```

code078.v



加算器のクリティカルパスの遅延を計測する

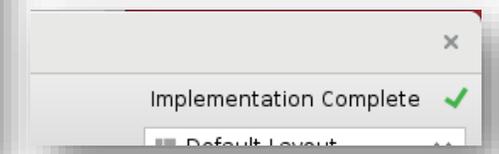
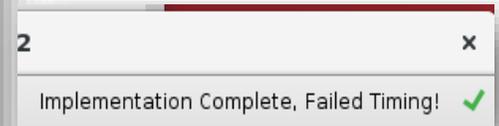
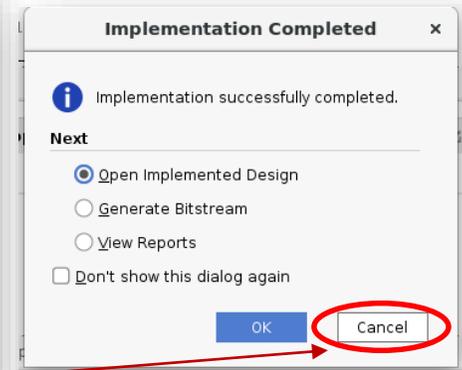
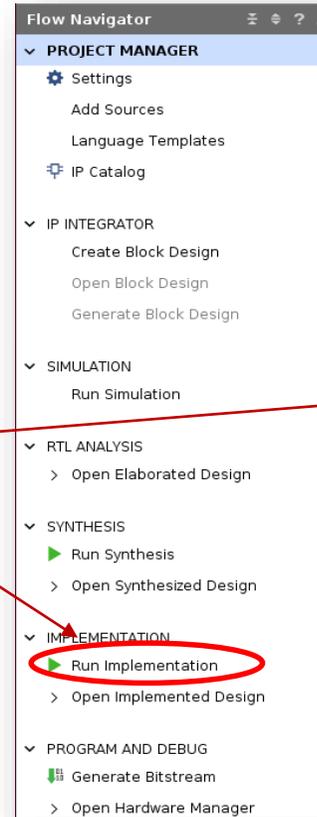
- code078.v を修正して, 100MHzの動作周波数の制約を満たす n-bit Adder の最大の n を求めること. ただし, n は5の倍数とする.
 - code078.v を用いて合成する(Run Implementation). Bitstreamは生成する必要はない.
 - 1行目の D_N の値を変化させて合成. Failed Timing! と出力された時は制約を満たしていない.
 - 1行目の D_N の値を小さくして合成. Implementation Complete が出力された時は満たしている.

```
code078.v
`define D_N 32
module m_main (w_clk, w_a, w_b, w_dout);
  input wire w_clk, w_a, w_b;
  output wire w_dout;
  reg [`D_N-1:0] r_a=0, r_b=0, r_s=0;
  wire [`D_N-1:0] w_s;
  assign w_dout = ^r_s;
  always@(posedge w_clk) begin
    r_a <= {w_a, r_a[`D_N-1:1]};
    r_b <= {w_b, r_b[`D_N-1:1]};
    r_s <= w_s;
  end
  m_ADDER m_ADDER0 (r_a, r_b, w_s);
endmodule

module m_ADDER (w_a, w_b, w_s);
  input wire [`D_N-1:0] w_a, w_b;
  output wire [`D_N-1:0] w_s;
  wire [`D_N:0] w_cin;
  assign w_cin[0] = 0;
  generate genvar g;
  for (g = 0; g < `D_N; g = g + 1) begin : Gen
    m_FA m_FA0(w_a[g], w_b[g], w_cin[g], w_s[g], w_cin[g+1]);
  end
endgenerate
endmodule

以降は省略
```

click this



新しい Vivado プロジェクトの作成とファイルの登録

- 前回の演習を参考に, Vivado で新しいプロジェクト **project_2b** を作成する.
- Ubuntu で起動したターミナルで, 次のコマンドを実行してファイルをコピーする.
 - /home/tu_kise は automount のディレクトリなので, アクセスしないとファイルが見えない. tabキーによる補完がうまく動作しないことがあるので注意する.
 - 最後の ls コマンドで, **code080.v**, **main11.xdc** が表示されることを確認.

```
$ ls /home/tu_kise
$ cd ~/cld/project_2b
$ cp /home/tu_kise/cld/2023/code080.v .
$ cp /home/tu_kise/cld/2023/main11.xdc .
$ ls
```

- Vivado で, **project_2b** の制約ファイルとして **main11.xdc** を登録する.
 - **main11.xdc** ファイルの内容を変更する必要はない.
- Vivado で, **project_2b** の Verilog HDLファイルとして **code080.v** を登録する.



code080.v シンプルな記述の n-bit Adder

- code080.v の D_N の値を変更して, 100MHzの動作周波数の制約を満たす n-bit Adder の最大の n を求めること. ただし, n は50の倍数とする.
- n-bit Adder の別のシンプルな記述例を code080.v に示す.
 - code078.v で記述した Ripple Carry Adder (順次桁上げ加算器)と, code080.v の記述のどちらが高速な回路を生成するか?
 - 桁上げ先見加算器 (Carry Lookahead Adder) について調べてみる.

code078.v

```
`define D_N 32

module m_main (w_clk, w_a, w_b, w_dout);
  input wire w_clk, w_a, w_b;
  output wire w_dout;
  reg [`D_N-1:0] r_a=0, r_b=0, r_s=0;
  wire [`D_N-1:0] w_s;
  assign w_dout = ^r_s;
  always@(posedge w_clk) begin
    r_a <= {w_a, r_a[`D_N-1:1]};
    r_b <= {w_b, r_b[`D_N-1:1]};
    r_s <= w_s;
  end
  m_ADDER m_ADDER0 (r_a, r_b, w_s);
endmodule

module m_ADDER (w_a, w_b, w_s);
  input wire [`D_N-1:0] w_a, w_b;
  output wire [`D_N-1:0] w_s;
  wire [`D_N:0] w_cin;
  assign w_cin[0] = 0;
  generate genvar g;
  for (g = 0; g < `D_N; g = g + 1) begin : Gen
    m_FA m_FA0(w_a[g], w_b[g], w_cin[g], w_s[g], w_cin[g+1]);
  end
endgenerate
endmodule
```

code080.v

修正するのはこの値だけ

```
`define D_N 32

module m_main (w_clk, w_a, w_b, w_dout);
  input wire w_clk, w_a, w_b;
  output wire w_dout;
  reg [`D_N-1:0] r_a=0, r_b=0, r_s=0;
  wire [`D_N-1:0] w_s;
  assign w_dout = ^r_s;
  always@(posedge w_clk) begin
    r_a <= {w_a, r_a[`D_N-1:1]};
    r_b <= {w_b, r_b[`D_N-1:1]};
    r_s <= w_s;
  end
  m_ADDER m_ADDER0 (r_a, r_b, w_s);
endmodule

module m_ADDER (w_a, w_b, w_s);
  input wire [`D_N-1:0] w_a, w_b;
  output wire [`D_N-1:0] w_s;
  assign w_s = w_a + w_b;
endmodule
```



Worst Negative Slack (WNS) & Critical Path

- From Vivado menu, select **Open Implemented Design**
- **Design Timing Summary** ウィンドウが表示される.
- WNS が正の値であれば, 生成された回路は制約を満たしている. また, 回路にはその値だけの余裕(slack)があることを示す.
 - 左図の $D_N = 32$ の例では, クロック周波数が 100MHz で 10 ns の制約に対して WNS は 1.796 ns となっており, これだけの余裕があることを示す. **つまり制約を満たしている**. この回路のクリティカルパスの遅延は $10 - 1.796 = 8.204$ ns となる.
 - 右図の $D_N = 80$ 例では, WNS は -3.527 であり, **制約を満たしていない**. この回路のクリティカルパスの遅延は $10 + 3.527 = 13.527$ ns となる.

Design Timing Summary

Setup	Hold
Worst Negative Slack (WNS): 1.796 ns	Worst Hold Slack (WHS): 0.166 ns
Total Negative Slack (TNS): 0.000 ns	Total Hold Slack (THS): 0.000 ns
Number of Failing Endpoints: 0	Number of Failing Endpoints: 0
Total Number of Endpoints: 94	Total Number of Endpoints: 94

All user specified timing constraints are met.

Design Timing Summary

Setup	Hold
Worst Negative Slack (WNS): -3.527 ns	Worst Hold Slack (WHS): 0.136 ns
Total Negative Slack (TNS): -60.012 ns	Total Hold Slack (THS): 0.000 ns
Number of Failing Endpoints: 29	Number of Failing Endpoints: 0
Total Number of Endpoints: 238	Total Number of Endpoints: 238

Timing constraints are not met.

$D_N = 32$ とした時のRipple Carry Adderの合成結果

$D_N = 80$ とした時のRipple Carry Adderの合成結果

Vivado 2022.2 を利用

加算器のクリティカルパスの遅延を計測する (CP2)

- 以下の全てが終わってから、「担当の教員あるいはTA」に確認してもらうこと。
- `code078.v` を修正して Ripple Carry Adder を実装し、100MHzの動作周波数の制約を満たす n -bit Adder の最大の n を求めること。
 - 変更した `code074.v` を用いて、`module m_FA`のVerilog HDLコードとシミュレーション結果を「担当の教員あるいはTAに」示すこと。
 - 100MHzの動作周波数で制約を満たす n -bit Adder の最大の n (これを N とする) を求める。ただし、 n は5の倍数とする。
 - N の時の WNS (数値)を「担当の教員あるいはTAに」示す。
 - $N + 5$ の時(ぎりぎり制約を満たさない場合)の WNS を「担当の教員あるいはTAに」示す。
- `code080.v` のシンプルな記述の加算器について、100MHzの動作周波数の制約を満たす n -bit Adder の最大の n を求めること。
 - 100MHzの動作周波数で制約を満たす n -bit Adder の最大の n (これを N とする) を求める。ただし、 n は50の倍数とする。
 - N の時の WNS (数値)を「担当の教員あるいはTAに」示す。
 - $N + 50$ の時(ぎりぎり制約を満たさない場合)の WNS を「担当の教員あるいはTAに」示す。
- `code080.v` が高速になる理由を考えて、「担当の教員あるいはTAと」議論すること。



Check Point 2



ACRiブログ

- ACRiブログのFPGA関連の記事を読んでみよう。
 - <https://www.acri.c.titech.ac.jp/wordpress/>



HPC アプリを FPGA 上で加速! (1)

みなさん、こんにちは。広島市立大学の窪田と申します。今回、ACRi ブログを執筆する機会をいただきましたので、高位合成ツールである Xilinx Vitis を使って連立一次方程式の求解法である Conjugate Gradient (CG) 法を FPGA で動かす例題を使って、ソフトウェアの FPGA への移植について説明します。はじめに私自身は、主に HPC 向けのソフトウェアや並列化コンパイラを研究、開発してきましたが、最近は FPGA を使った処理の高速化にも手を広...

© 2021.04.15



ACRi ルームの FPGA で○×ゲームを作って遊んでみよう (1)

ACRi ルームではリモートで FPGA を利用できる環境が提供されています。セットアップ済みの開発ツールを使ってすぐに FPGA 開発をはじめることができます。...と言われても、さて何からはじめてみよう、という人もいるかもしれません。身近な問題から FPGA に向くものを選んで実装するというのは(残念ながら)簡単ではないので仕方ないことだと思います。そこで、楽しく FPGA 開発をはじめてみる題材としてゲームを作ってみるのはいかがでしょうか。この連載では、FPGA を使った○×ゲーム...

© 2021.04.14



FPGA for HPC : 宇宙物理アプリケーションをマルチ FPGA で高速化 (1)

どうも皆さんお久しぶりです。筑波大学計算科学研究センター所属の小林諒平です。昨年はスーパーコンピュータでも使われ始めた FPGA という記事を執筆させて頂き、そして今年また本ブログにて記事を執筆させて頂く機会を賜りましたので、今回は計算科学研究センター(以下、当センター)にて

そもそも「FPGA」って何なんですか? (1)



FPGA 使い、って社長に言われた。さあ、どうする? (1)



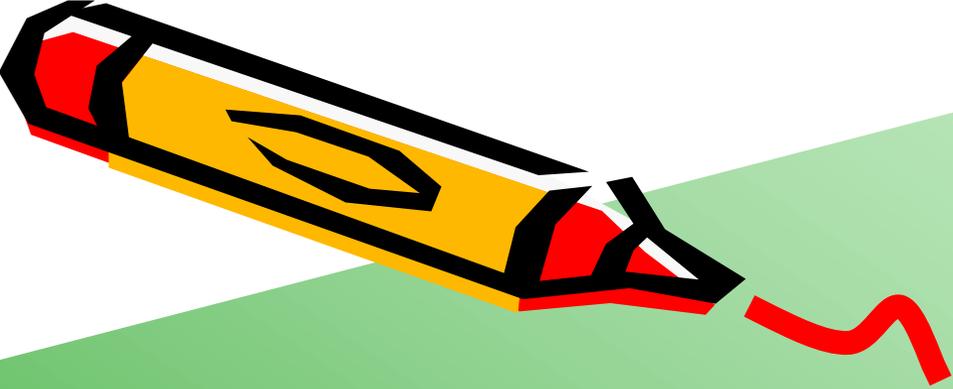
Vol. 1



参考資料

- 4ビットカウンタとシリアル通信ではじめるFPGA開発
ACRiブログシリーズ Kindle版
 - ハードウェア記述言語を用いたFPGA開発の入門書です。FPGAボードを購入することなく、無料で体験できるACRiルームの利用方法を説明しているので、かんたんにFPGAのためのハードウェア設計を始めることができます！
- ここまでの演習が難しいという場合は、こちらを読むと良い。
- ACRiブログの対応するコースを参考にしても良い。





References



References

- Computer Logic Design support page
 - <https://www.arch.cs.titech.ac.jp/lecture/CLD/>
- ACRi Room
 - <https://gw.acri.c.titech.ac.jp>
- ACRi Blog
 - <https://www.acri.c.titech.ac.jp/wordpress/>
- 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 Arty A7-35T
 - <https://reference.digilentinc.com/reference/programmable-logic/arty-a7/start>
- Verilog HDL
 - <https://ja.wikipedia.org/wiki/Verilog>

