制約とデザインをよくする Tips

KEK IPNS E-sys
本多良太郎

取り扱う内容

配置制約
- IOB制約, ASYNC_REG制約 (外界との信号やり取り)
- 相対配置制約 (RLOC)

タイミング制約
- クロック定義とグルーピング
- multicycle_path制約
- クロックをMUXで切り替える場合

制約の説明≒Vivadoの説明であり関連UGの種類がとても多いです。
- 直接関係のあるUG: UG903, UG912。
- Vivado関係のUG: UG894, UG895, UG896, UG899, UG901, UG904, UG905, UG906, UG908

全部合わせると1500ページ程度になります。
とても扱いきれないのでよくあるケースに絞って解説します。

デザインをよくするためのTIPS
- リセット配布
- 巨大なファンアウトとドライバの複製
- CLBレジスタ挿入について再考
- マジックナンバーはやめましょう
- 巨大なIPには注意
- タイミング違反が出たら

余談
制約の熟練度がどれだけ凝ったFWを生成できるかを大体決めているように感じます。

外界との信号やり取り

Electronics System Group

タイミング解析について

FPGA内部の信号やり取り
- Vivadoが全てのパラメータを知っている(Jitterなどは正しく与えたうえで) 。通常Vivadoに任せればよいはず。
- FPGA内部信号の遅延量制御を行うような回路は中級レベルではありません。

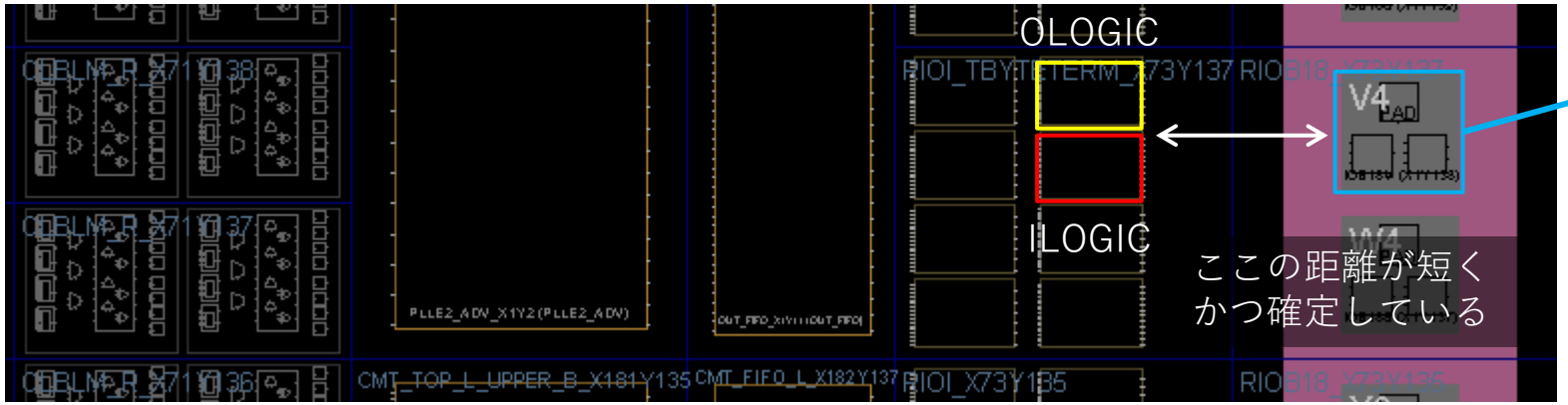
FPGA外部との信号やり取り
出力
- 同期信号はすべて同じタイミングでFPGA PADから出力されてほしい。
- IO block上にレジスタを配置すればよい。
  - 最後のレジスタからPADまでの経路は基板上の配線パターンを延長したことに相当。
  - IO blockに最後のレジスタを固定しておかないと合成のたびにパターン長が変わる回路となってしまう。

入力
- IO blockにレジスタを配置したとしても信号間の入力タイミングばらつきが気になる。
  - 信号間の時間差を制約で与える?与えた時間差の見積もりって本当に正確?
  - 今回はこれは省略することにします。
- やはりIO block上の素子を使って取り込む方法を考えます。

⇨ 必要な場合UG903 IO遅延の制約を参照

低速の場合 (50-100 MHz程度まで)と高速の場合について考えみます。
いずれの場合でもスキュー調整のためクロックバッファに頼らないといけない事を想定しており、
1 MHz以下など高速クロックで如何様にでもタイミング調整できるケースは除外します。

外界への出力

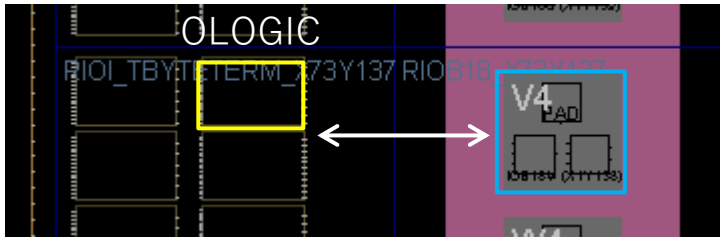


低速 (50-100 MHz程度まで)のバス信号などの場合
データであれば
- 最後のレジスタをOLOGIC上に配置する => **IOB制約**を使う

クロックであれば
- OLOGICをODDRプリミティブに構成し出力する

高速, もしくはシリアル通信の場合
- 素直にクロックフォワード付きのOSERDESを利用する。

外界への出力



OLOGIC上にレジスタが配置される。
HDLのコード上はただのFFに見え、CLBの
レジスタと区別はつかない。

XDCによるIOB制約

- HDLでコード内制約も行えますが再利用性が悪くなるので私はIOBはXDCでかけるようにしています。
- 最上位ポートを駆動している素子がFFでなければいけません。

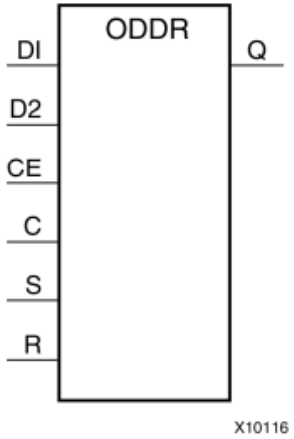
```
set_property IOB TRUE [get_ports SIGIN]
set_property IOB TRUE [get_ports SIGOUT]
```

最上位のモジュールポート名

外界への出力

ODDR

- Double-data-rateでデータを出力するためのプリミティブ。
- D1にHIGH, D2にLOWを入力しておけば同位相, 逆にすれば逆位相のクロックが出力される。



ポートの説明

ポート名	方向	幅	機能
Q	出力	1	データ出力 (DDR)。IOB パッドに接続されます。
C	入力	1	クロック入力
CE	入力	1	クロック イネーブル入力。High になると、ポート C のクロック入力がイネーブルになります。
D1 : D2	入力	1 (それぞれ)	データ入力。DDR データを ODDR モジュールに入力するピンです。
R	入力	1	リセット。SRTYPE の設定によって異なります。
S	入力	1	アクティブ High の非同期セット ピン。SRTYPE 属性の設定により、同期にもなります。

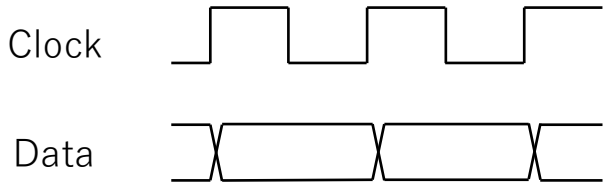
なぜODDRプリミティブを使ってクロックを出力するか

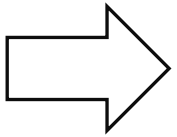
- OLOGICにはクロック出力専用のプリミティブが (恐らく) 無い。
- ODDRを使う事でIOB制約を使った他のデータとの間でスキュー調整されたクロックを外部へ出力することが出来る。

外界からの入力

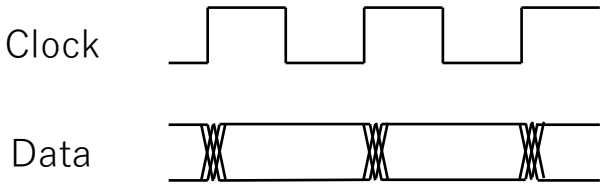
データとソースクロック両方の入力がある場合
(データ取り込みを外来クロックで行う)

データ出力時





データ入力時



配線遅延が揃えられていない回路では、
タイミングがずれている**かも**しれない。
(きちんと等長が取られている回路なら起こらない。)

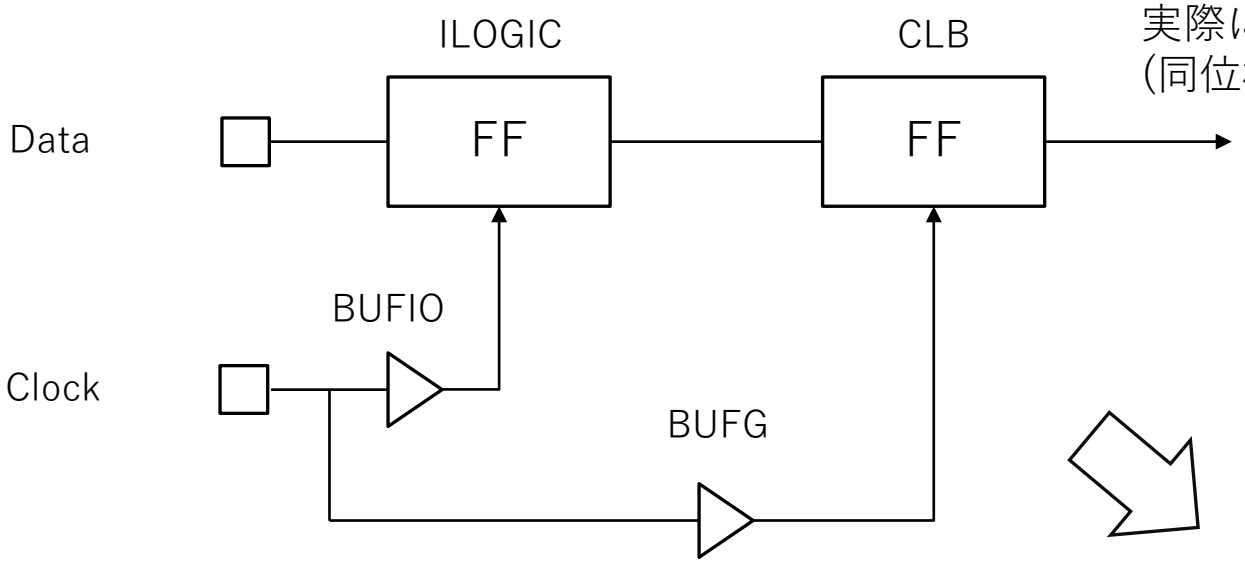
同位相は危ない場合でも、50-100 MHz程度までなら
逆位相で取り込めば十分余裕がある。

それ以上の速度の場合素直にISERDESを使った方が
よいだろう。

外界からの入力

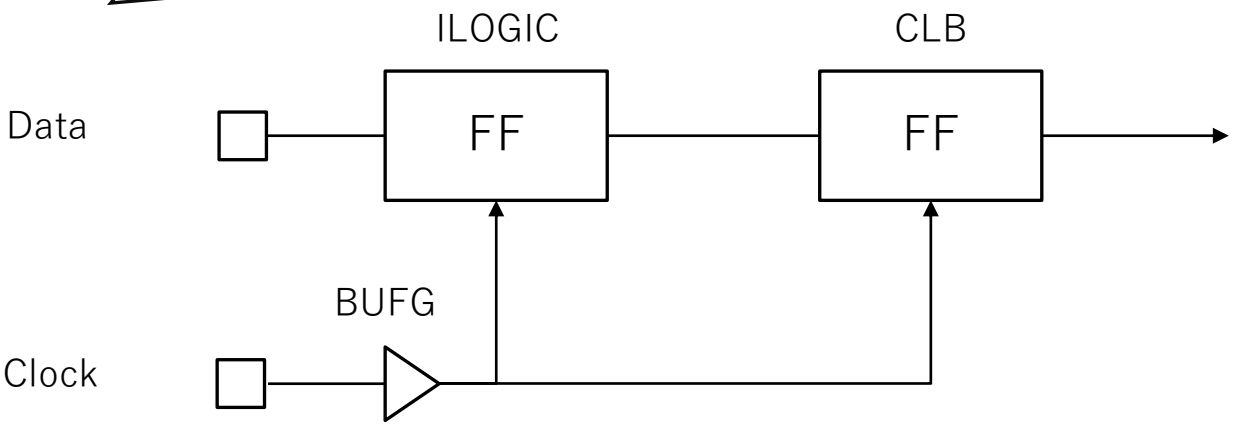
逆位相クロックをソースが送ってくれる場合。

IOB制約で最初のFFをILOGICに置く



説明のために書いた冗長な回路
実際にはBUFIOとBUFGが出力するクロックは同一
(同位相になるように調整される) のでBUFGだけで良い。

現実的な配線

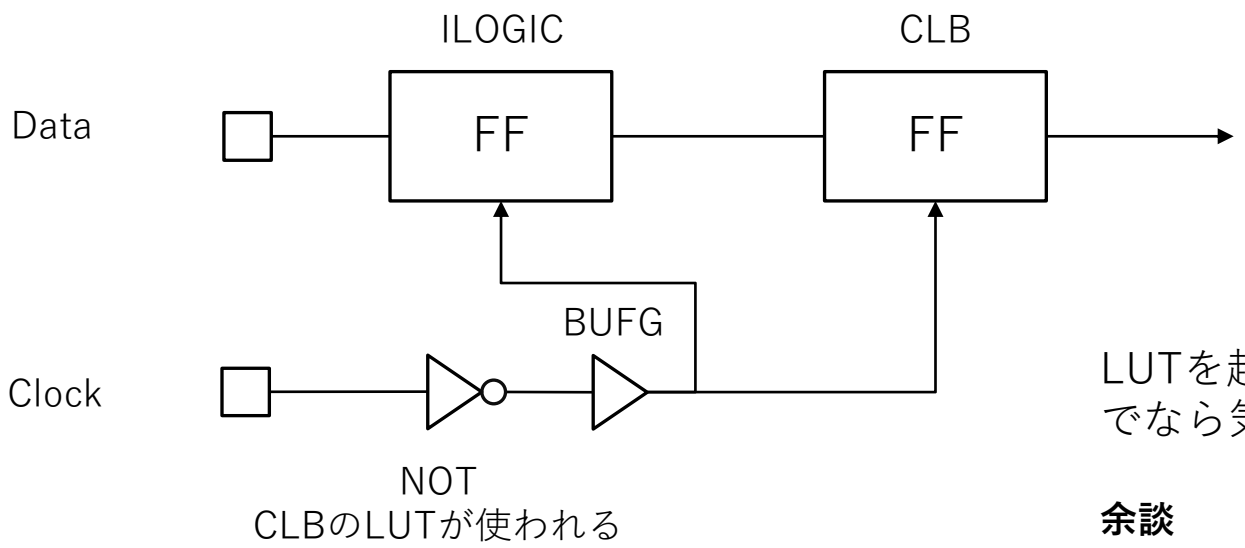


クロック入力はCCピンでないといけない。
(恐らくエラーになる)

外界からの入力

逆位相クロックをソースが送ってくれない場合。

現実的な配線



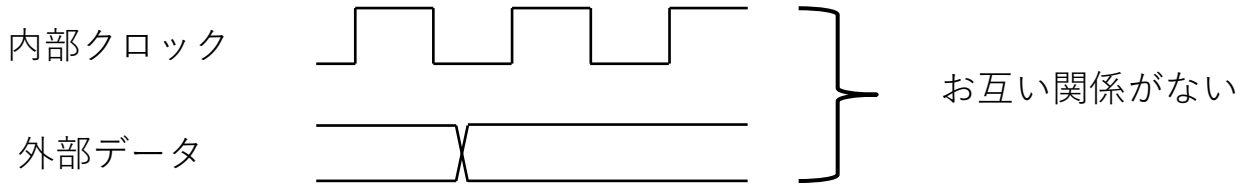
LUTを超えるために数百psの遅延が出るが100 MHz程度までなら気にならないだろう。

余談

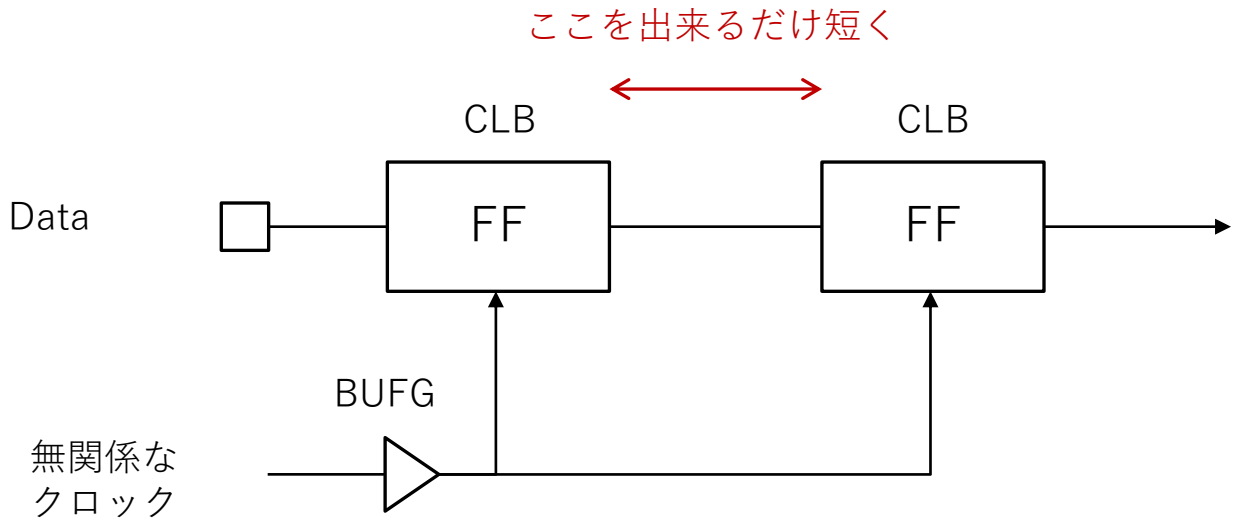
- PADとBUFIO間は専用配線でファブリックリソースが介在できないため、NOT入りで先ほどの冗長な構成は作れない。

外界からの入力

ソース側から同期クロックの入力が無い場合。
(無関係のシステムクロックで取り込まれる)
このケースではデータはただのタイミング入力か非常にゆっくり遷移するデータだろう。

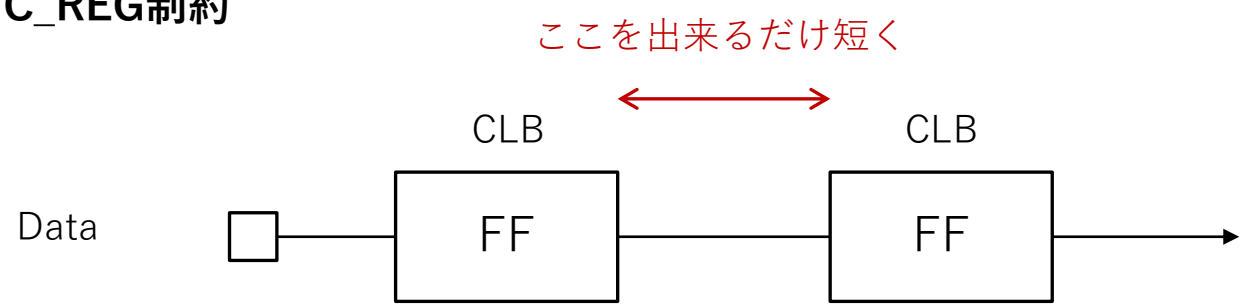


まず考慮しないといけない事はメタステーブル対策である。
二重なり三重のFFを使って



外界からの入力

ASYNC_REG制約



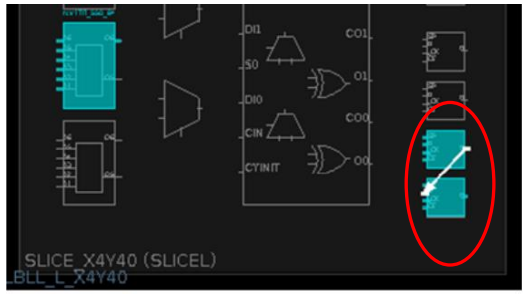


図 3-2: レジスタのグループ化

ASYNC_REG制約を用いると可能な限り
同一スライス内のレジスタ素子に配置しようとする。

Verilog-HDL

```
(* ASYNC_REG = "TRUE" *) reg sync_0, sync_1;
always @(posedge clk) begin
sync_1 <= sync_0;
sync_0 <= en;
...
```

VHDL

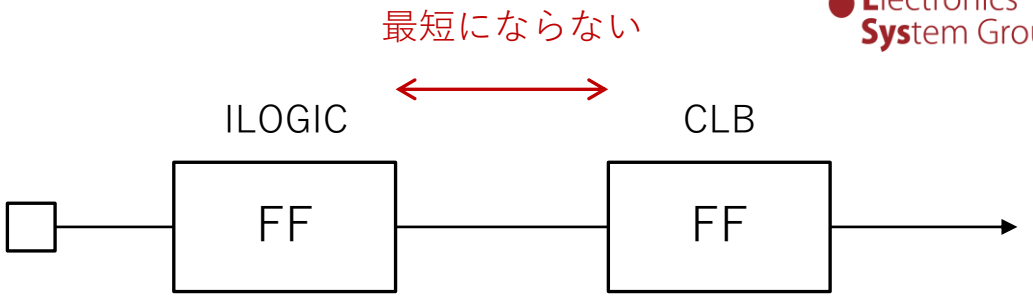
```
  attribute ASYNC_REG : string;
  attribute ASYNC_REG of sync_ff : label is "true";

begin

  sync_ff : process(clk)
  begin
    if(clk'event and clk = '1') then
      reg_sync0  <= data;
      reg_sync1  <= reg_sync0;
    end if;
```

外界からの非同期入力に限らずFPGA内部で非同期クロックドメインを
乗り換えるときにも利用する。
非同期信号を受け取る時は常に使ってよい。

外界からの入力

ASYNC_REGとIOB制約

- ASYNC_REGとIOBを同時に制約するとIOBが優先される。



しかし、優先すべきはメタステーブル対策。
そのIOBは本当に必要か考えましょう。

- ➢ FPGA内部での配線遅延は多くとも数ns。
- ➢ 非同期クロックで取り込んでいる時点で通常無視できる値だろう。

どうしても同時に使いたい場合 (本当に必要かまず考えてほしい)

- 三重FFを用意して最初のFFにIOB制約を適用し、残り2つのFFをLOC制約で手動で近くのスライスにおいてしまう方法がある。
  - ASYNC_REG制約だけではILOGCの近くのスライスを使ってくれるとは限らないため。

メタステーブルとトラブル事象

経験上メタステーブルに一番弱いのはFSM (Finite State Machine)
ステート制御に使っている信号がメタステーブル状態になると…
- 未定義状態に落ち込む。
- デッドロックする。
- 処理が実行されない。

などなど…沢山トラブルにあいました。

直接FSMの制御を行う信号に気を付けるのは当然ですが、
メタステーブルは伝搬しうるので非同期信号は必ずメタステーブル対策を施してください。

FSMは非同期リセットにも弱いです。

Electronics System Group

LOC, RLOC

openit
Open source consortium of Instrumentation

LOC制約とBEL制約

LOC

- 絶対位置を指定する配置制約。SITEにロジックセルを固定する。
  - IOの最も近くに特定のセルを置きたい（先ほどのIOBとASYNC_REGが例）
  - デザインが複雑で自由度が多すぎて近くに配置するべきセルが離れてしまうようなとき
    - ただそこまで行く場合PBLOCKの採用を検討するべきかと思う。
- コード内制約をするよりもフロアプランナでGUIから位置を指定することが多いのでは。
  - 後程やってみます。

BEL

- CLB SLICEに対して内部リソースのどれを使うか指定する。
  - BEL制約が必要な事は普通はない。
  - psオーダーの遅延量制御のために私は使うが、非同期的な使い方。

原則FPGA内部の配置配線はVivadoに任せてよい。
何でもかんでもLOC制約をしないようにしましょう。

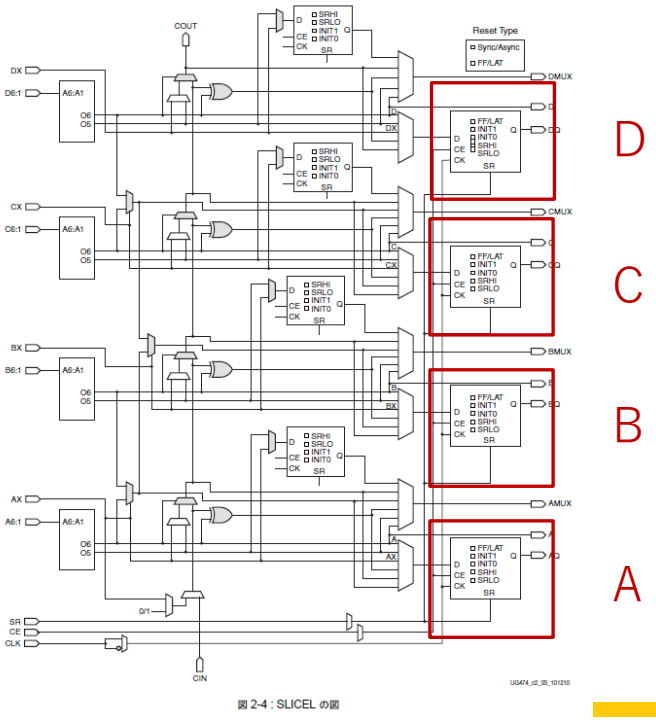


図 2-4 : SLICEL の図

RLOC制約

RLOC

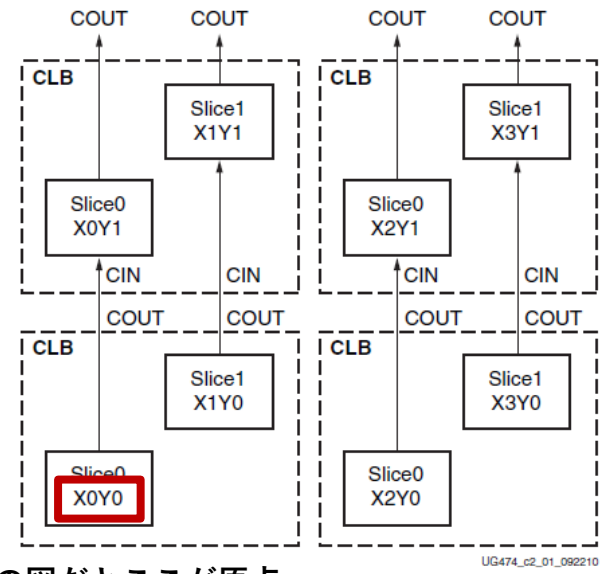
- セルの相対的な位置を指定する配置制約。原点がどこに来るかはVivadoが決める。
  - 原点を設計者が決めたい場合RLOC_ORIGINを追加で使用する。
- XDCで指定ができずコード内制約のみ。
- 私は4相クロックを使った1 ns精度程度のTDCのタイミング制御はこれで済ませてしまいます。
  - **相対的に近いところにいる ⇔ 配線遅延量が大体同じ**

VHDL

```
attribute rloc        : string;
attribute rloc      of u_FDCE_A     : label is "X0Y0";
attribute rloc      of u_FDCE_B     : label is "X1Y0";
attribute rloc      of u_FDCE_C     : label is "X0Y1";
attribute rloc      of u_FDCE_D     : label is "X1Y1";
```

Verilog-HDL

```
(* RLOC = "X0Y0", HU_SET = "h0" *) FD sr0 (.C(clk), .D(sr_1n), .Q(sr_0));
(* RLOC = "X0Y0", HU_SET = "h0" *) FD sr1 (.C(clk), .D(sr_2n), .Q(sr_1));
(* RLOC = "X0Y1", HU_SET = "h0" *) FD sr2 (.C(clk), .D(sr_3n), .Q(sr_2));
(* RLOC = "X0Y1", HU_SET = "h0" *) FD sr3 (.C(clk), .D(sr_4n), .Q(sr_3));
(* RLOC = "X0Y0", HU_SET = "h1" *) FD sr4 (.C(clk), .D(sr_5n), .Q(sr_4));
(* RLOC = "X0Y0", HU_SET = "h1" *) FD sr5 (.C(clk), .D(sr_6n), .Q(sr_5));
(* RLOC = "X0Y1", HU_SET = "h1" *) FD sr6 (.C(clk), .D(sr_7n), .Q(sr_6));
(* RLOC = "X0Y1", HU_SET = "h1" *) FD sr7 (.C(clk), .D(inrn), .Q(sr_7));
```



この図だとここが原点

IOB, LOCについて
実際にやってみます。
(RLOCはタイミング制約とあわせて)

1つのHDL内に複数原点がある場合HU_SETでグループを指定する。
RLOCはHU_SETで指定したグループへの制約なので、
未指定の場合でも暗黙で生成される。

Electronics System Group

タイミング制約

openit
Open source consortium of Instrumentation

クロック定義とクロックグループ

```
create_clock -period 10 -name clk_sys -waveform {0.000 5.000} [get_ports CLKP]

create_generated_clock -name tdc0 [get_pins u_mmcm/inst/mmcm_adv_inst/CLKOUT0]
create_generated_clock -name tdc1 [get_pins u_mmcm/inst/mmcm_adv_inst/CLKOUT1]
create_generated_clock -name tdc2 [get_pins u_mmcm/inst/mmcm_adv_inst/CLKOUT2]
create_generated_clock -name tdc3 [get_pins u_mmcm/inst/mmcm_adv_inst/CLKOUT3]
```

ソースクロック

ソースクロックからMMCMで生成したクロック

- 利用しているクロックは全て定義しないといけない。
- MMCMやPLLを利用するとソースクロックと生成クロック両方ともVivadoが自動定義してくれる。
  - ただし分かりずらい通し番号がついてしまうので私は自分で再定義する。

```
create_generated_clock -name clk_div2 -source [get_ports CLKP] -divide_by 2 [get_pins instance_name/pin_nDame]
```

- FFで2分周したようなユーザー定義のクロックも忘れずに定義する。

Vivadoが自動生成したクロックのリストが欲しい場合
- 合成済みデザインを読み込みTcl Consoleでreport_clocksを実行する。
- ユーザー定義のクロックが漏れていないかチェックにも使える。

クロック定義とクロックグループ

何故定義するか?

- 各クロック間の同期・非同期の指定をVivadoに教える必要があるから。

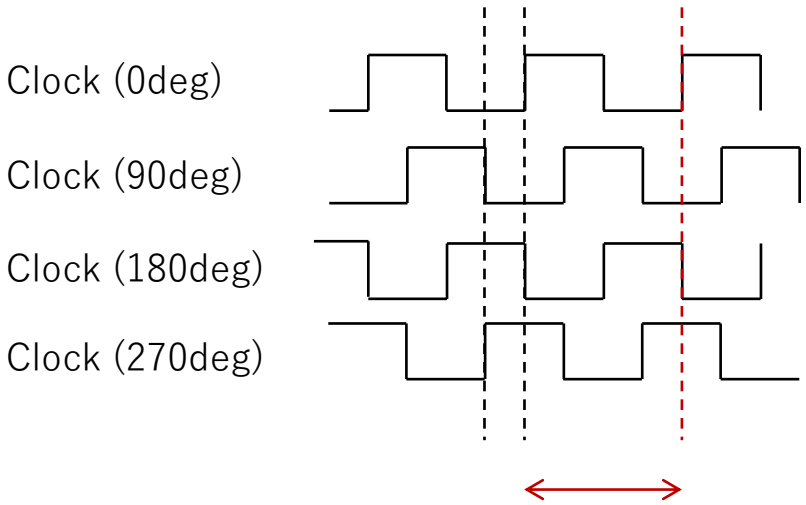
set_clock_groupでasynchronous制約する

- デフォルトでは定義済みクロックは全て同期であるとしてタイミング解析で計算される。
- そもそも同期関係がないクロック間、同期関係にはあるがタイミング解析から除外してもよいクロック間は非同期指定する。

```
set_clock_groups -name async_sys -asynchronous \
    -group {clk_tdc_0 clk_tdc_90 clk_tdc_180 clk_tdc_270} \
    -group clk_sys \
    -group clk_gtx \
    -group clk_spi \
    -group clk_icap \
    -group clk_10mhz
```

グループでまとめられたクロック間はタイミング解析に含まれる。
グループ間は非同期として解析から除外される。

multicycle_path



multii-cycle: 2

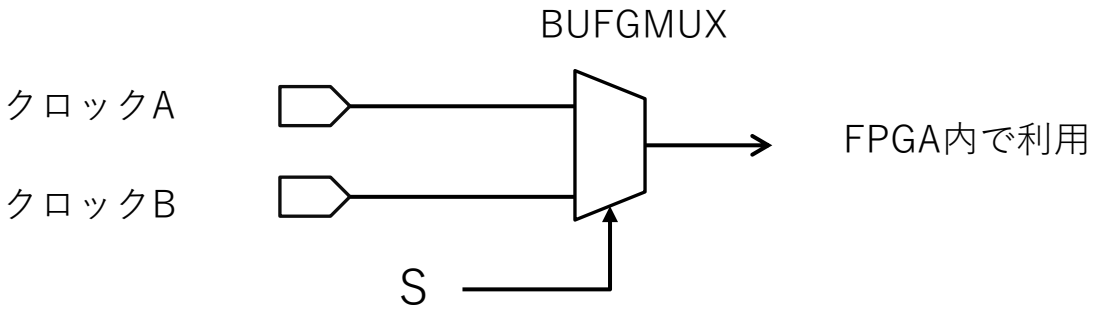
- 例えば左のような周波数が同じで90度ごとに位相の異なるクロックを利用している時、270度から0度のドメインへの乗り換えを考える。
- この時許される時間は元の周波数の1/4しかなく大体の場合でタイミングを満たさない。

```
set_multicycle_path -setup -from [get_clocks tdc3] -to [get_clocks tdc0] 2
```

- そこでmulticycle_path制約を利用して270度から0度への乗り換えの際は次のクロックエッジではなく、複数回あとのエッジを利用する事にする。
  - 上の例では2つ目のエッジで正しく取り込まれるようにタイミング解析される。
  - 許容レイテンシは5/4周期分。

複数のクロックをBUFGMUXで切り替える場合

クロックAとBは周波数が同じだがソースが異なる。
- 例: Aはローカル発振器、デバッグ用。Bは上流から配られるマスタークロック、本番用。

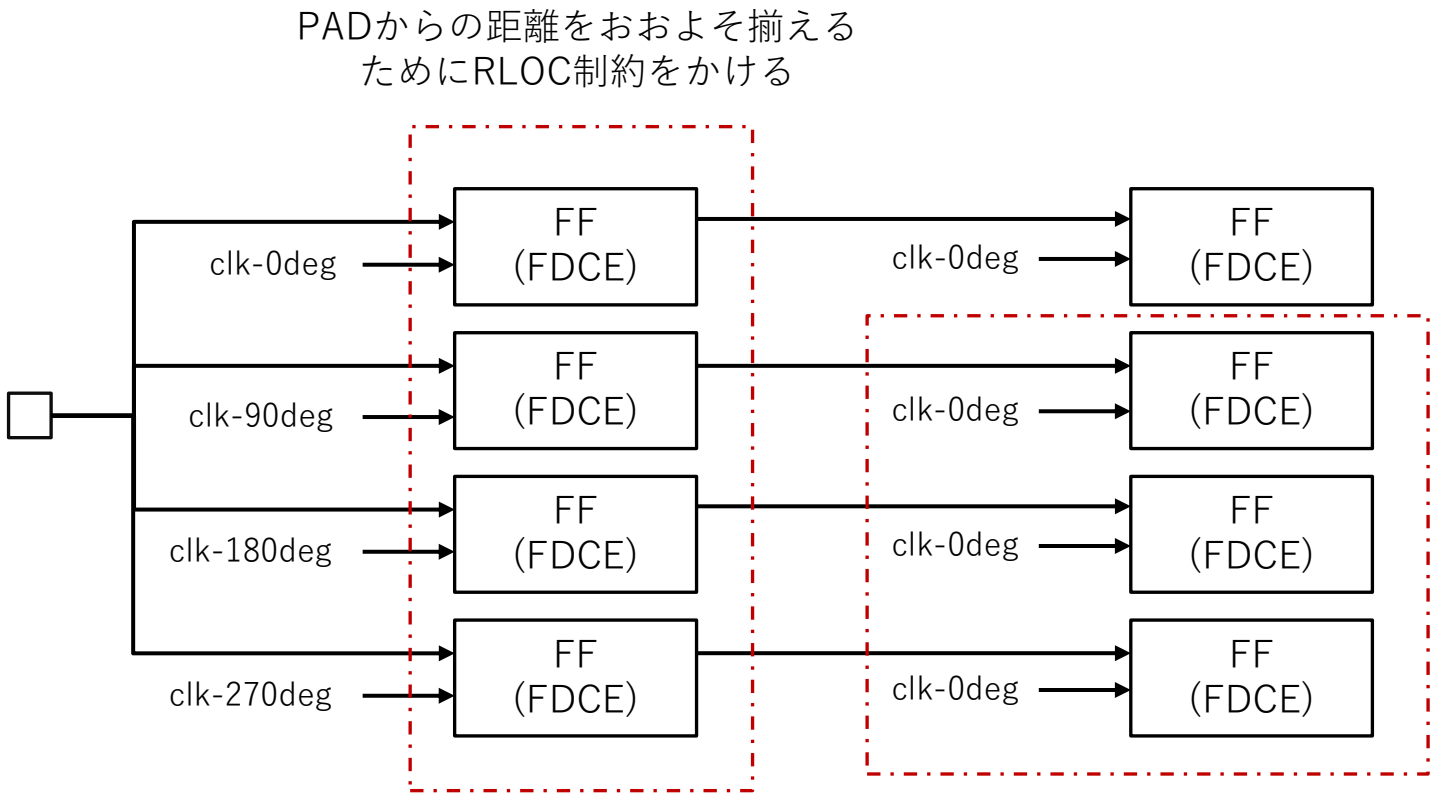


- どういうわけかVivadoはクロックAとBの間のタイミング解析をしようとしてタイミング違反を出してくる。
- これら2つは同一かつ排他的でありどちらかだけ解析すればよいことを伝えないといけない。

```
# Clock definition
create_clock -period 10.000 -name clk_osc -waveform {0.000 5.000} [get_ports CLKOSC_P]
set_input_jitter clk_osc 0.030

create_clock -period 10.000 -name clk_hul -waveform {0.000 5.000} [get_ports CLKHUL_P]
set_input_jitter clk_hul 0.030

set_case_analysis 0 [get_pins u_BUFGMUX_inst/S]   MUXのS入力。S=0のケースだけ解析しなさいの意。

set_clock_groups -name async_input -physically_exclusive -group [get_clocks clk_osc] -group [get_clocks clk_hul]
```

set_clock_groupで物理的に排他的なことを指示

ここで再びRLOC, multicycle_path, clock_groupについて実際にやってみます。
ここでは4相クロックを利用してTDCを作る際の最初の4つのFFと、
その次のクロック乗り換えについて実装してみます。



クロックドメイン乗り換え
が発生する。
270度から0度への乗り換えに
multicycle_pathを指定する。

Electronics System Group

デザインをよくするためのTips

openit
Open source consortium of Instrumentation

リセット配布について

前提知識

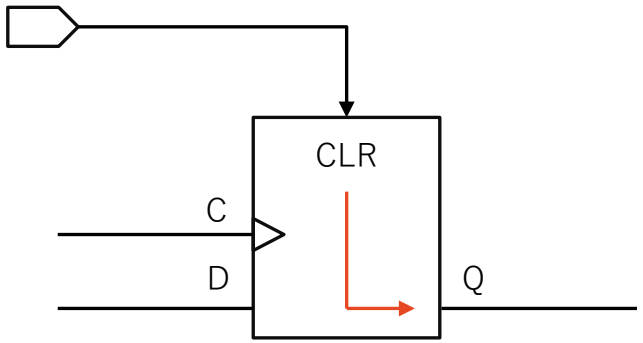
XilinxではCLBレジスタの非同期 (同期)リセット (セット) について以下の表記を用いている。

- 非同期リセット: (Asynchronous) Clear (CLR)
- 非同期セット: (Asynchronous) Preset (PRE)
- 同期リセット: (Synchronous) Reset (R)
- 同期セット: (Synchronous) Set (S)

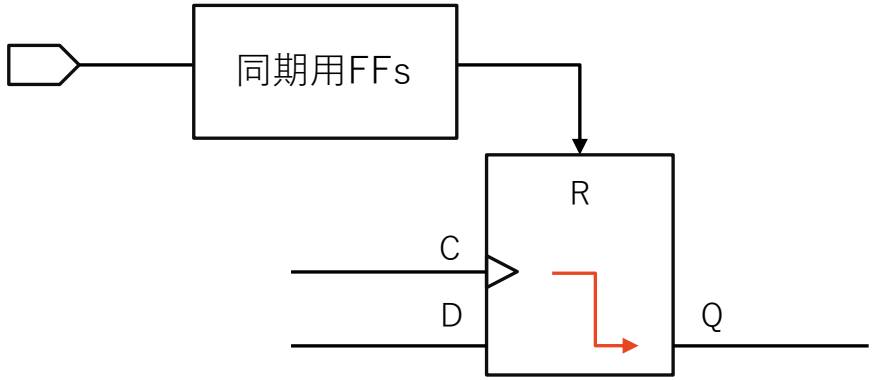
⇨ Primitive名のFDCE
D-typeのFFでclock Enableと非同期リセット(Clear)がついた素子

これまでの例題でいくつかのリセット配布の仕方を使ってみました。
端的に書くと以下の2種類

非同期リセット



同期リセット



Qの遷移タイミングが不明な非同期リセット回路が危険なことは明白。
原則として非同期リセット (セット) は使ってはいけません。
これだけなら簡単ですがリセット配布にはもっと複雑な事情があります。

リセット配布について

ここで話すことはXilinxのホワイトペーパー（WP231）の抜粋です。
かなり古いドキュメントで現在のXilinx FPGAではそのまま通用しない事も書いてありますが、
一般論としてとてもいいことが書いてあります。

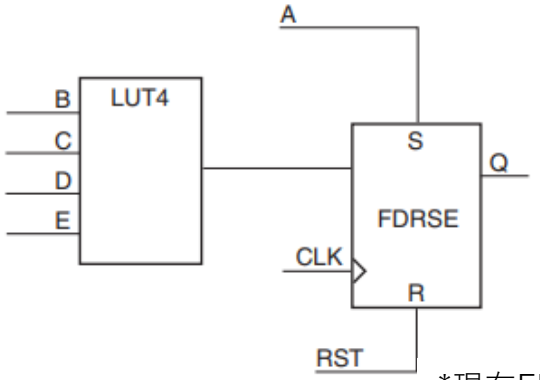
非同期リセット（セット）と同期リセット（セット）ではCLBリソースを利用する時の最適化が違う

- QはCLKに同期して遷移しなければならない。
- CLKに同期しているRとSは組み合わせ回路の一部として使える。

VHDL	Verilog

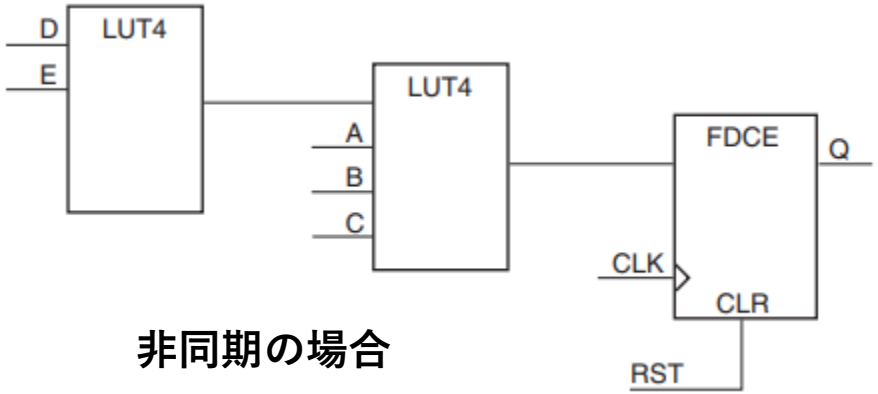
process (CLK)
begin
  if (rising_edge(clk)) then
  if (RST = '1') then
  Q <= '0';
  else
  Q <= A or (B and C and D and E);
  end if;
  end if;
end process;
	always @(posedge CLK)
	  if (RESET)
	  Q <= 1'b0;
	  else
	  Q <= A | (B & C & D & E);

同期の場合



*現在FDRSEはプリミティブとして廃止されている

*現在のFPGAではLUTは6入力なのでこの通りにはならない



非同期の場合

リセット配布について

そのリセット信号は本当に必要？不必要なリセットは最適化を阻害する

- シリーズFF (CLBのレジスタ) はSRL16 (LUTで実装) もしくは分散RAMで置き換える事が出来る。
  - SRLとRAMにはリセット入力がない。リセットのビヘイビア記述が存在すると最適化されない。

- 1つのスライスが受け付けられるリセット信号は1種類。
  - 複数のリセットを配っているとリセットが異なるリーフセルは必ず別のスライスに分割されてしまい、ファブリック面積を余分に消費する。
  - この事はクロックでも同様。ファブリックの利用効率を高めるためにはなるべくクロックドメインの数を減らした方がよい。

- 同期リセットしか受け付けない専用リソースの内蔵レジスタへ非同期リセットを配ろうとすると、一部機能がファブリックに染み出すことがある。

（恐らくHDLでビヘイビア記述により専用リソースを推定させる場合の話だと思われる）

リセット配布について

不必要なリセット配布を削除する

- バスやシーケンサ制御信号はリセットのタイミングで正しく0に遷移しないとまずいが、例えばデータ線などは制御信号が0になれば流れて消えていくだけなのでリセットを配らなくてもよい
  - （かもしれない。ケースによる。）

- ところでリセットがないと電源投入時にはどうなる？
  - Xilinx FPGAにはGlobal Set Reset (GSR) 信号という電源投入時に全てのプリミティブに配布される信号が存在する。
  - GSR入力があると各プリミティブの出力はINITで指定した値に初期化される。

リセット配布について

完全に同期リセット（セット）だけで対応できたとして気を付ける事は？

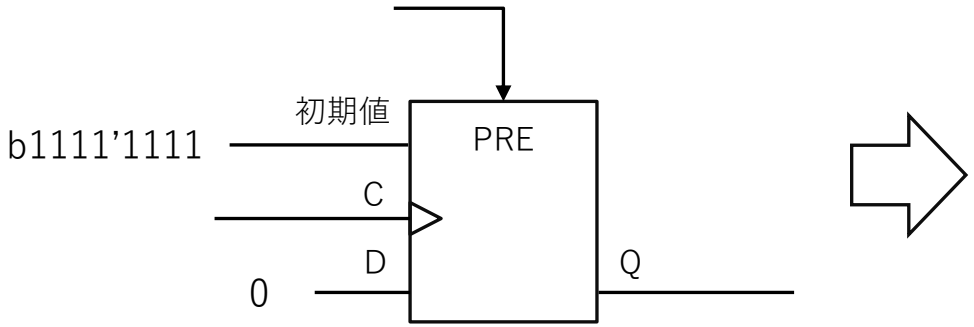
- 全てのリセット経路に対してタイミング解析が必要。巨大なFWでは動作速度を落とす原因となるかも。
- 同一信号を複数の場所へ到達させるため巨大なファンアウトが必要になる
  - 不必要なリセットは削除する。
  - BUFG (グローバルクロックライン) を使ってリセット配布を行う。
    - BUFGが余っているなら試す価値あり。
- 非同期クロックドメイン間では必ずリセット信号の再同期を取る事。

実はGSRを能動的に配る方法も存在する。
そのためにはFPGAのコンフィグレーションを制御するSTARTUP~というプリミティブを利用する。
かなりマニアックな解決方法なのでもし利用を検討する場合各UGを熟読して利用してください。

リセット配布について

どうしても非同期リセットを使わないといけない場合
(例えばリセットによりクロックを喪失する場合など)

- リセットの入りは非同期、抜けは同期にするというテクニックが存在する。



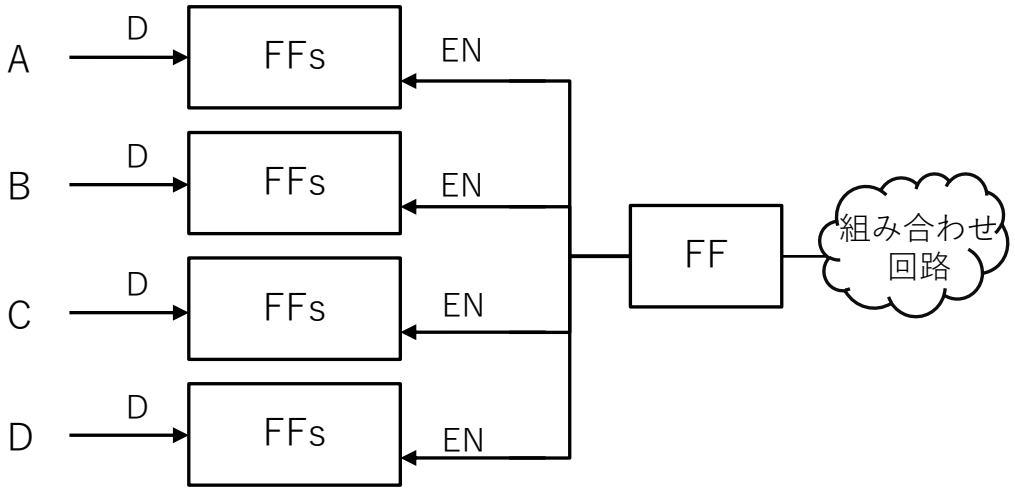

```
-- Reset sequence --
sync_reset  <= reset_shiftreg(kWidthResetSync-1);
u_sync_reset : process(rst, clk)
begin
  if(rst = '1') then
    reset_shiftreg  <= (others => '1');
  elsif(clk'event and clk = '1') then
    reset_shiftreg  <= reset_shiftreg(kWidthResetSync-2 downto 0) & '0';
  end if;
end process;
```

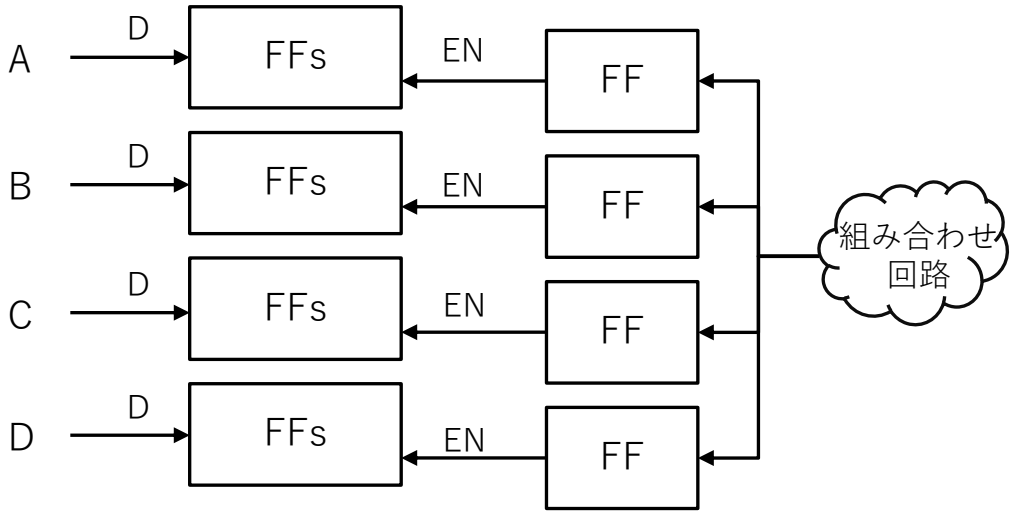
非同期セットが入ると
b1111'1111をプリセットする
長さ8のシフトレジスタ
(長さは必要なクロックサイクル数に依存)

Qを非同期リセットへ配布する

巨大なファンアウトとドライバの複製

同一の信号をたくさんのエンドポイントに配る場合どちらがファブリックを広く使えるだろうか?
(信号伝達には必ず配線遅延が存在する)





- 右の方が良さそうに見えるが、このようにHDLを記述しても最適化により1つのファンアウトにまとめられてしまう事が多い。（ドライバの複製と削除は最適化の一種）
- **EQUIVALENT_DRIVER_OPT制約**でKEEPを指定すると同一信号のドライバ統合を阻害できる。
  - HDLでは制約できずXDCでセルに対して行う。

UG912

- 検出器セグメント間のマトリックスコインシデンスを取るような回路だと、同一信号を複数のエンドポイントに配る事が多発する。

CLBレジスタの挿入について考察

- CLB以外のリソースはFPGA内で特定の列に格納されており動かせない。
- 専用リソース間の信号伝達は配線遅延が大きく動作周波数を下げる要因になりえる。
  - 専用リソース間には適度にレジスタを挿入しファブリック内に中継地点を設けた方が、スループットが向上する（レイテンシとはトレードオフ）。

設計の勘所として
- パイプラインレジスタを後付けしてもロジックが破綻しないように設計する。
- 追加の遅延を許容しておく。

ソースコードの書き方として
- 信号線名の振り替えを利用する（これは賛否あると思うのでコーディングルールを決めておきましょう）

```
din_moduleB    <= dout_moduleA;
```

ただの信号線名の振り替え
(Xilinxの例題コードで良く見る記述)

```
process(clk)
begin
  if(clk'event and clk= '1') then
    din_moduleB    <= dout_moduleA;
  end if;
end process;
```

新しい信号線の定義なしにレジスタの追加が出来る。
モジュールを追加する場合でも書き直しになる部分の量が減る。

一方で…
定義する信号線数が増えるので可読性が悪くなる。
自分のコーディングスタイルを決めましょう。

マジックナンバーはやめましょう

マジックナンバーとはべた書きの数字で一見すると何を意味するのか分からない物です。

HDLではバス幅の記述やプリセット値でやってしまいがちですが、
他の人が見た時の可読性を大幅に悪くします。

Verilog-HDLにしろVHDLにしろ定義済みの定数をインクルードする方法があるので、
マジックナンバーはさけましょう。

巨大なIP（ロジック）について

例えば以下のような巨大なFIFOを実装したいとします。

- データ入力幅: 32-bit
- データ出力幅: 8-bit
- 書き込み深さ: 65536
- 書き込みクロック: 125 MHz
- 読み出しクロック: 250 MHz

データ幅がNon-symmetric aspect ratiosなのでbuilt-in FIFOは利用できない。

さて、これを実装して入力をVIO、出力をILAへ配線してみましょう。
とても簡単な（というかほぼ何もしない）回路だし周波数もそれほど高くありません。
ただ1つ…読み出し深さが非常に大きくなることが予想できます。

配置配線は通るでしょうか？

巨大なIP（ロジック）について

タイミングエラーが出ました（Vivado 2020.1, Windows 10で実行）

Timing	Setup | Hold | Pulse Width
Worst Negative Slack (WNS):	-0.571 ns
Total Negative Slack (TNS):	-4.788 ns
Number of Failing Endpoints:	16
Total Number of Endpoints:	8558
Implemented Timing Report	

到達地点がhiddenになっており何の信号か分からない。
いずれにせよIP内部なので設計を変形することは不可能。

Summary

Name	Path 21
Slack	-0.571ns
Source	u_fifo0/U0/inst_fifo_gen/gconvfifo.rf/grf.rf/gntv_or_sync_fifo.mem/gbm.gbmg.gbmga.ngecc.bmg/inst_blk_mem_gen/gnbram.gnativebmg.native_blk_mem_gen/valid.cstr/ramloop[56].rar
Destination	<hidden> (rising edge-triggered cell FDRE clocked by clk_sys {rise@0.000ns fall@2.000ns period=4.000ns})
Path Group	clk_sys
Path Type	Setup (Max at Slow Process Corner)
Requirement	4.000ns (clk_sys rise@4.000ns - clk_sys rise@0.000ns)
Data Path Delay	4.234ns (logic 2.162ns (51.066%) route 2.072ns (48.934%))
Logic Levels	5 (LUT3=1 LUT6=2 MUXF7=1 MUXF8=1)
Clock Path Skew	-0.307ns
Clock Un...rtainty	0.065ns

巨大なIP（ロジック）について

これは意図的に出したタイミングエラー。
非対称データ幅により出力側の回路が巨大になったことが原因。

ではどうするか？
データサイズ的にFIFOの深さも変えられないし非対称データ幅も必須だとすると…

- データのバッファ機能は巨大なbuilt-in FIFO
- データ幅の変換は小さなBRAM FIFO

に分割して実装してみましょう。

巨大なIP（ロジック）について

タイミングエラーが消えて必要な機能もすべて実装できました。

今回はあくまで1つの起こりうるケースですが、
巨大で複雑な回路は動作周波数がどうしても上がりません。

特に下位ビットから上位ビットへの伝搬遅延の存在する
巨大なカウンタを内蔵している回路は要注意です。
（例えばRAMのポインタインクリメントはこれに相当）

なるべくシンプルにかつ1つ1つの回路規模が小さくなるように設計するようにしましょう。

タイミング違反が出たら

日々Worst Negative Slack (WNS) 戦っている人も少なくないでしょう。
万能の処方箋はありませんが、いくつかのケースで考えてみます。
(false_pathやclock_groupの制約が正しく行われている前提で)

ファブリックの使用量がそれほど大きくない場合（空いているクロック領域が多い）
配置配線の自由度が低いので動作周波数限界に近いことが予想できる。

- まずはtiming summaryで遅延を生んでいる原因を探る。
  - Logicが主であればパイプラインレジスタを挿入する余地があります。
- リセット信号が要求時間内に到達できない事もあり得ます。
  - これまでに説明したテクニックを組み合わせて対処します。

Summary

Name	Path 21
Slack	-0.571ns
Source	u_fifo0/U0/inst_fifo_gen/gconvfifo.rf/grf.rf/gntv_or_sync_fifo.mem/gb
Destination	<hidden> (rising edge-triggered cell FDRE clocked by clk_sys {ris
Path Group	clk_sys
Path Type	Setup (Max at Slow Process Corner)
Requirement	4.000ns (clk_sys rise@4.000ns - clk_sys rise@0.000ns)
Data Path Delay	4.234ns (logic 2.162ns (51.066%) route 2.072ns (48.934%))
Logic Levels	5 (LUT3=1 LUT6=2 MUXF7=1 MUXF8=1)
Clock Path Skew	-0.307ns
Clock Un...rtainty	0.065ns

タイミング違反が出たら

ファブリックの使用量が高い（200T程度のFPGAでスライス使用量7-8割程度以下）
配置配線の自由度が高く正解にたどり着いてない可能性がある
巨大なFPGAだともっと低使用率から難しくなるだろう

- パイプラインレジスタ挿入や同一信号を広く配るためのドライバの複製など、ファブリックを広く使う工夫を試してみる。
- それでもダメな場合ロジックや配線、特にリセット配線の削除を検討する。
- CLBで実装している機能でDSPやBRAMなど専用ブロックへ振り替え可能な物がないか検討する。

足す方向の修正より（自由度を増やしてしまう）、
引く方向の修正の方がいい結果を生むことがある。
（過剰なパイプラインレジスタを削除する事も時に必要となる。）
ファブリックの最適化を促進して使用率を下げられるとよい。

スライス使用量が7-8割を超える場合
このあたりから動作周波数を維持することが爆発的に難しくなる。

- **そもそもこういう状況にならないようにFPGAは選定したい。**
- PBLOCK等の制約を駆使してロジックセルが利用できる領域を制限したりして、自由度を下げてあげないとVivadoが正解にたどり着けなくなる。
- このあたりは脱初心者を目指す人には荷が重すぎる。

タイミング違反が出たら

- どうしてもだめな場合配置配線（および合成）のストラテジーを変えるという方法があります。
- 省スペース化、省電力化、高パフォーマンス化など色々なストラテジーがあり、タイミングを後少しで満たせそうなときに有効。

私は原則デフォルトストラテジーで開発します。
（この業界のFWは何度も小改良をすることが多いため）
売り切りの場合最初から目的のストラテジーで開発してもいいでしょう。

コード修正のたびにタイミング違反が出たりでなかったりするような危うい状況ではなく、確定的にタイミングを満たすようになるまでデザインを修正します。