FPGA/HDLを活用したソフトウェア並列処理の構築

goyoki

@並列プログラミングカンファレンス

自己紹介

- goyoki(hatena/twitter)
  - 千里霧中http://d.hatena.ne.jp/goyoki/
- 組込みエンジニア
- Doxygen日本語メンテナ
- 主にテスト関連コミュニティで情報発信
  - yomite.swtest、xUnit Test Patterns読書会等

概要

- 並列処理のための組込みアーキテクチャをソフトエンジニアを対象に紹介

- 組み込みでの並列処理の構築例
  - アーキテクチャの工夫でFPGA/HDLをソフトウェア的に扱う
  - ソフトウェアとFPGA/HDLの協調により、並列処理の設計容易性を改善させる

目次

- 概要
  - 組み込み並列処理とFPGA/HDLの関わり
- FPGA/HDL
  - FPGA/HDL概要
  - FPGA/HDLをソフトウェア的に扱うためのアーキテクチャ
- 並列設計
  - FPGA/HDLでの並列処理設計
  - ソフトウェア-FPGA協調処理（通信処理編）
  - 協調処理の設計プロセス

組み込み並列処理とFPGA/HDLの関わり

組込み並列処理の現状

- 組込み並列処理の需要
  - 処理データの増大
    - 画素数、FPS、通信料、ストレージ容量・・・
  - インタフェースの複雑化

- 組込み並列処理のジレンマ
  - 組み込みで要求されること
    - リアルタイム性
    - ハードウェアのコストダウン

リアルタイム性

- 十分な実行時間の時間予測性
- 入力に対する即時反応

- 時間の粒度はデジタル回路信号レベル

```
//必ず100ns
for (i = 0; i < 100; i++){}

//必ず100ns
for (i = 0; i < 100; i++){}
```

ハードウェアのコストダウン

- ハードウェアコストが製品コストに直結
- CPUパワーやメモリが一般的に貧弱
  - 量産効果、配置面積、電源等の要因

- スペックダウンが製品の強みに

組み込み並列処理のジレンマ

- 少ないコア（ロースペック）で、シングルタスク（RT処理）
  - シングルタスク＋割り込み
  - RTOS
- 並行処理が複雑化すると設計が爆発
  - 割り込みでリアルタイム性が損なわれる
  - タイミングの重複でタスク漏れ

- FPGA/HDLへの処理委譲で、並列処理の設計容易性を劇的に改善できることがある
  - FPGAをソフトウェア的に扱えるアーキテクチャを構築
  - タスク分割で処理をFPGAに委譲

FPGA/HDL概要

FPGA

- Field Programmable Gate Array
- 任意のロジック回路を構築できるLSI
  - EDA/論理合成ツールによりHDLコードをデジタル回路として展開できる
  - IPコアやペリフェラルで機能を提供
- ICとして提供される

HDL

- Hardware Description Language
  ハードウェア記述言語
- Verilog HDL、VHDLが主流
- 論理回路の設計やふるまいを記述するための開発言語
  - PLD等の設計、実装
  - ASIC等の設計
  - 論理回路のシミュレーションやモデリング
- プログラミング言語とみなされる場合も

HDL

Verilog HDLコード

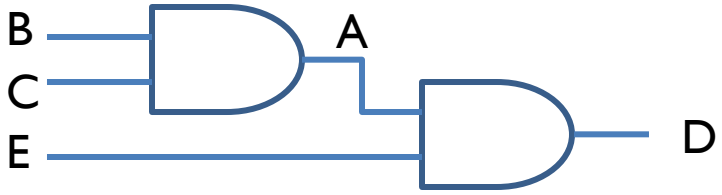
```
module simple_ff (clk, nreset, d, q, enable);
        input clk, nreset, d, enable;
        output q;

        reg q_reg;

        assign q = q_reg;

        always @(posedge clk or negedge nreset)
                if (nreset == 1'b0)
                        q_reg <= 0;
                else if (enable == 1'b0)
                        q_reg <= q;
                else
                        q_reg <= d;
endmodule
```

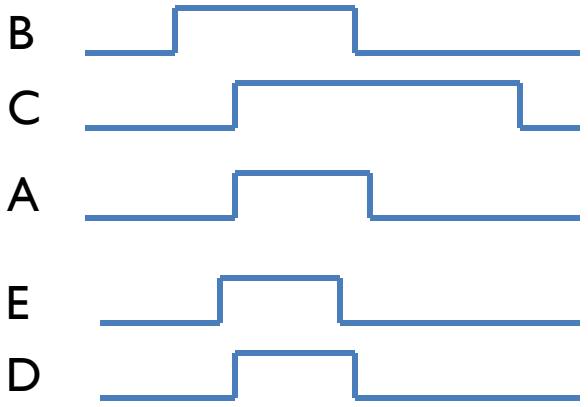
HDLコードの特徴

- コードがデジタル回路として動作
  - 常時評価
  - 並列動作
  - 電気的遅延

Verilog HDLの記述例

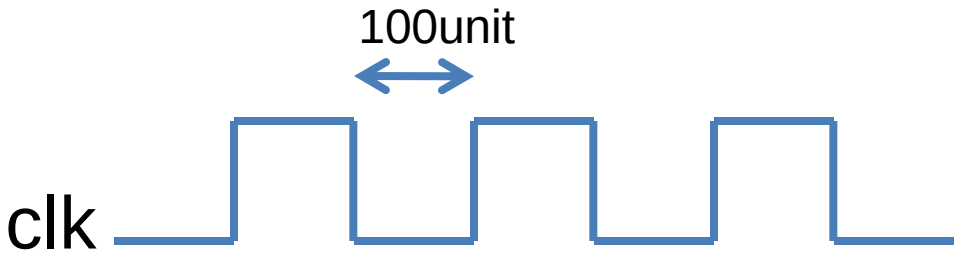
```
wire A, B, C, D, E;

assign A = B & C;
assign D = A & E;
```





Verilog HDLの記述例2

```
reg clk;

always #100
        clk <=~ clk;

initial clk <= 0;
```



HDLの特徴

- ソフトウェアとの違い
  - ソフトウェアプログラミング言語
    - デフォルトで直列実行
    - 特殊な構文で並列実行
  - HDL
    - デフォルトで並列実行
    - 特殊な構文で直列実行

Verilog HDLの基本文法

- 今回は触りとして基本的なものを紹介

データ型、代入文の扱い

- 基本型
  - ネット型（wire）：配線
  - レジスタ型（reg）：値を保持するデータ型

- 代入文の扱い
  - ネット型（assign文）：回路の接続

```
wire X, Y;
assign X = Y;
```

データ型、代入文の扱い

- レジスタ型への代入
  - 「=」：ブロッキング代入。順序を守って代入
  - 「<=」：ノンブロッキング代入。並列的に代入

```
always #500 begin
        A <= B;
        C <= D;
end

always #300 begin
        A = B;
        C = D;
end
```

数値表現

- 基本表現
  - 「0」「1」「x」「z」
  - 上記の連なった値（「4’b101x」「4’d5」）
- シミュレーション用にハイレイヤーなデータ型も持つ（Integer型など）

- 高位な数値表現は規約的に実現
  - 小数：固定小数点を設定
  - 負値：2の補数処理を導入

演算

- C、Pascalの多くの演算子を使用可
  - 演算子（「+」「-」「/」「*」「%」「~」「&」「|」「^」）
  - 論理演算、比較(「!=」「&&」「||」「==」「<」「<=」)
  - シフト演算など(「>>」「<<」)

```
assign C = A + B;
assign A = (B == 4'b0101) ? D : E;
```

手続き型処理の記述

- initial文：指定ステートメントを一度だけ実行
- always文：指定ステートメントを繰り返し実行
  - イベントトリガを設定可能
  - initial文はシミュレーション用

```
Initial begin
   …
end

always #100 begin
   …
end
```

構造化

- if文、for文、while文使用可
- 多くはシミュレーション用途
  - プログラミング言語と見なされることも

- モジュール単位
  - module、function、task
- `include構文で参照構造を形成

```
for (i = 0; i < 100; i = i + 1) begin
....
end
```

高級言語機能

- ファイル操作（$fopne、$fread、$fwrite…）
- 標準入出力（$monitor、$display…）
- 時間関連（$time、$stime…）
  - 基本シミュレーション用途

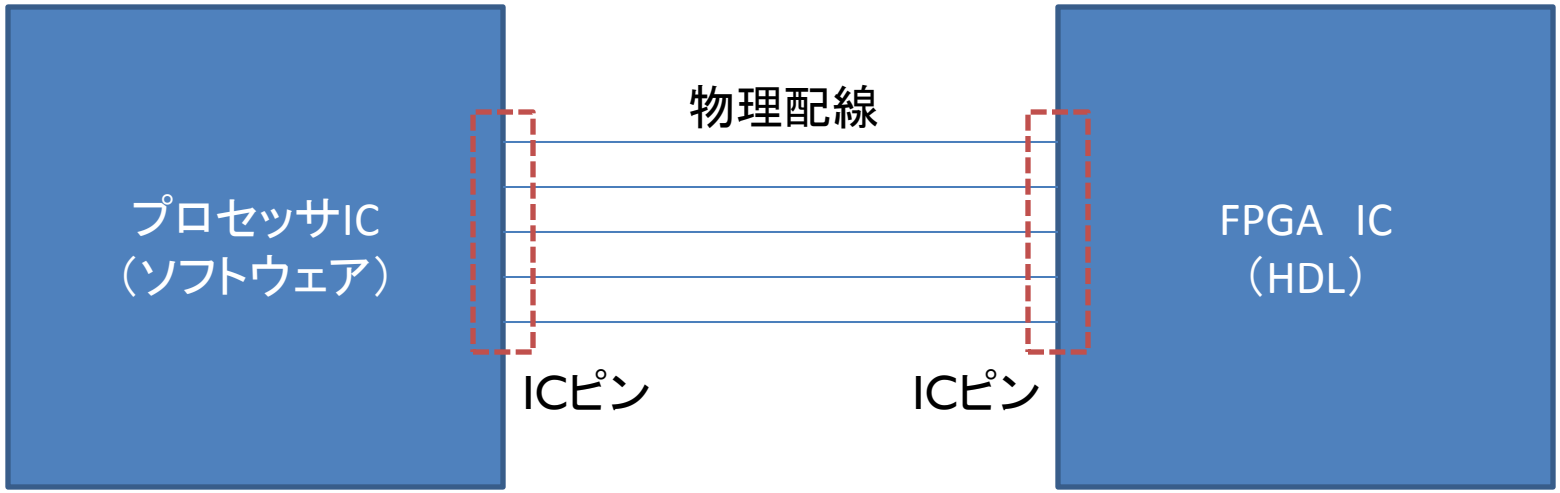
HDLまとめ

- 3つの特徴
  - 並列実行
  - 常時評価
  - 電気的遅延
- 高級言語機能も持つマルチパラダイム言語
- 並列処理を容易に記述できる

FPGA/HDLをソフトウェア的に扱うためのアーキテクチャ

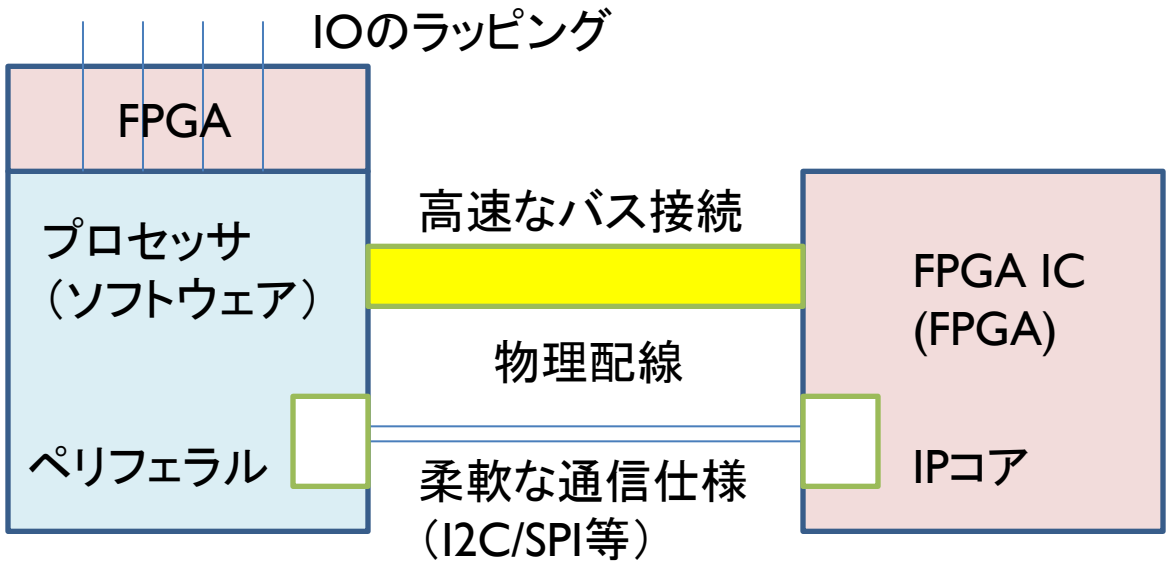
一般的な設計

- FPGAとプロセッサは個別のICとして扱う
- 物理配線を介して外部IOで通信する
- FPGAはデジタル回路デバイスとして扱われる



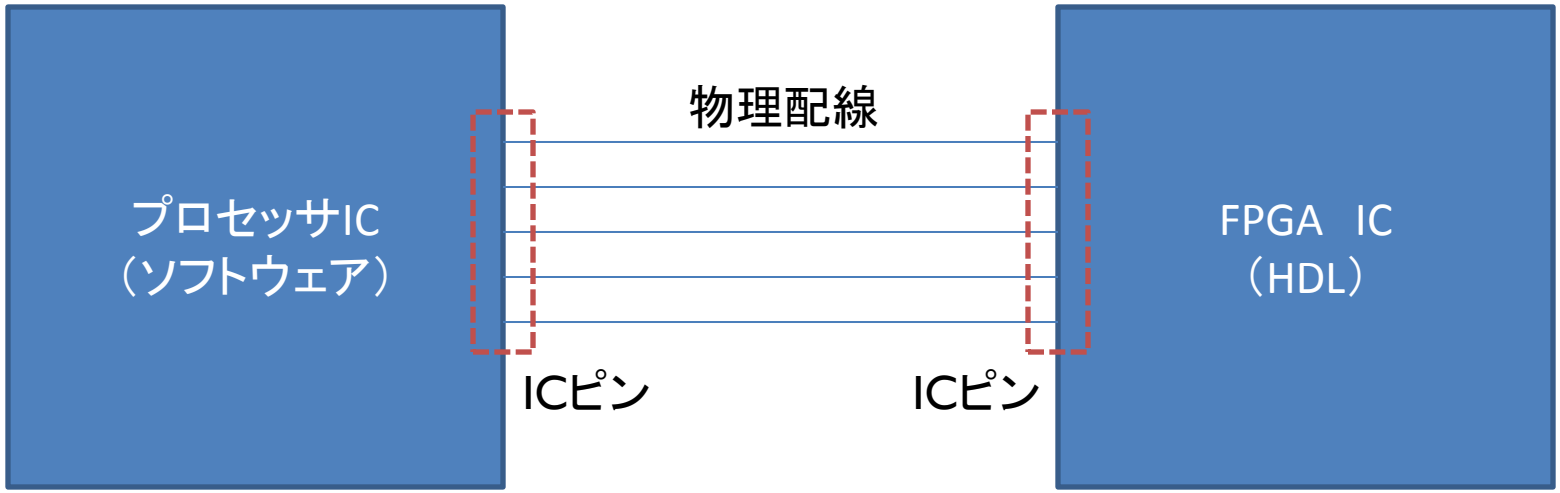
プロセッサ/FPGA統合ボード

- プロセッサ/FPGA統合ボード
  - ex)プロセッサとFPGAを高速バスで接続する
  - ex)〃を柔軟な通信規格で接続する
  - ex)プロセッサIOをFPGAでラッピングする



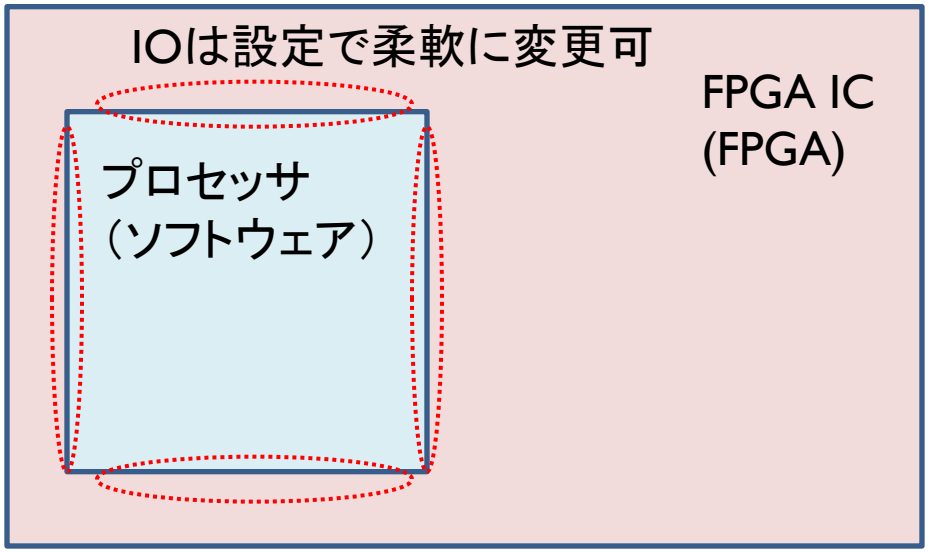
問題

- データ共有が外部IO、物理配線で縛られる
  - 上流工程で機能分割がFix
  - 下流でのタスク分割・委譲が制限される
  - FPGAは非ソフトなハードウェアとして扱われる



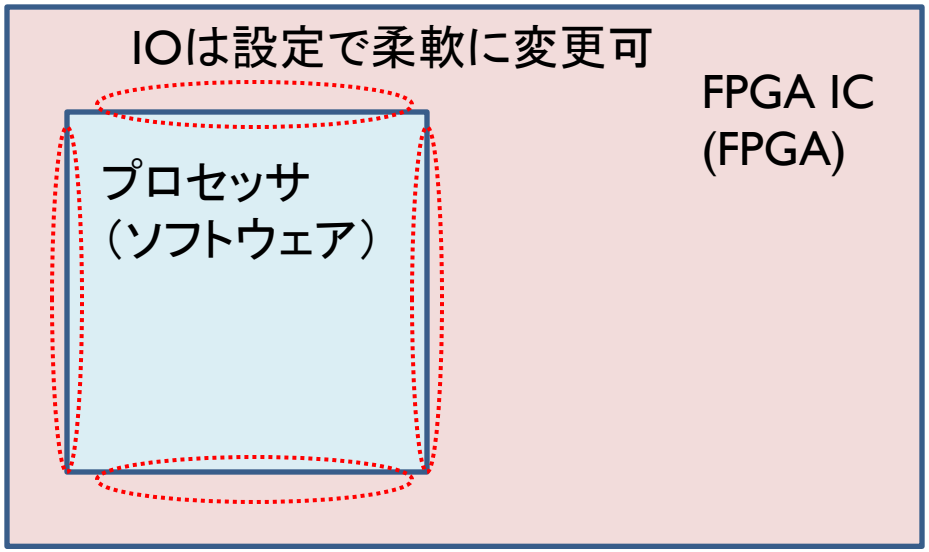
改善策：ソフトプロセッサ

- ソフトプロセッサでソフトを駆動する
  - FPGA上に展開可能なプロセッサ
  - コードで記述される
  - ツールで柔軟にカスタマイズ・拡張可能
  - MicroBlaze、Nios II等



改善策：ソフトプロセッサ

- ソフトウェア-HDLのIFを柔軟に設定可
  - 抽象化されたデジタルデータをAPIで共有
  - プロセッサ機能カスタマイズ化
  - 接続設定カスタマイズ・拡張可
- ライブラリのAPIのような扱いでHDL側を操作



アーキテクチャまとめ

- 通信・データ共有方法がH.W.に依存すると上流で機能分割をFixしなければならない
- ソフトプロセッサによりHDLをソフトウェア的に扱えるようになる

FPGA/HDLでの並列処理設計

HDLでの並列処理の設計

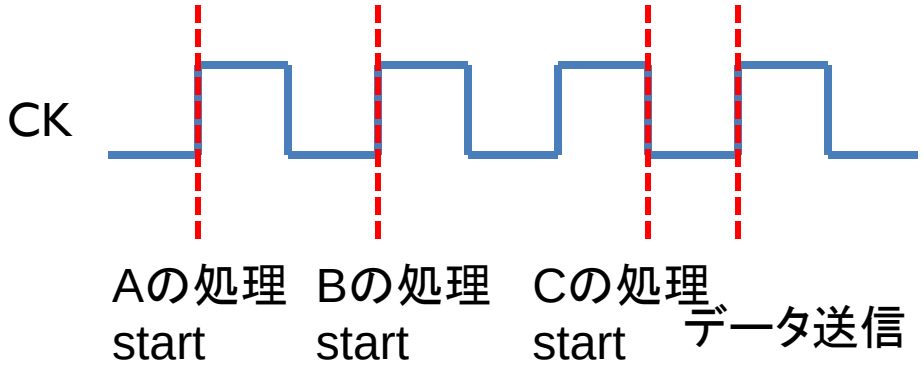
- 並列処理設計の障害
  - 常時評価
  - 並列実行
  - 電気的遅延

- HDLでは時間予測性の確保が重要
- 一般的なアプローチ
  - クロック同期設計

クロック同期設計

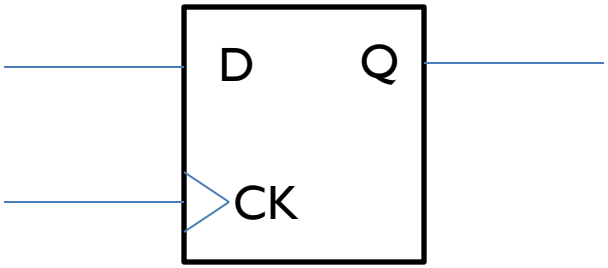
- 高精度、高周波数の基準クロックを確保
- 基準クロックに合わせて信号伝達や演算を行う

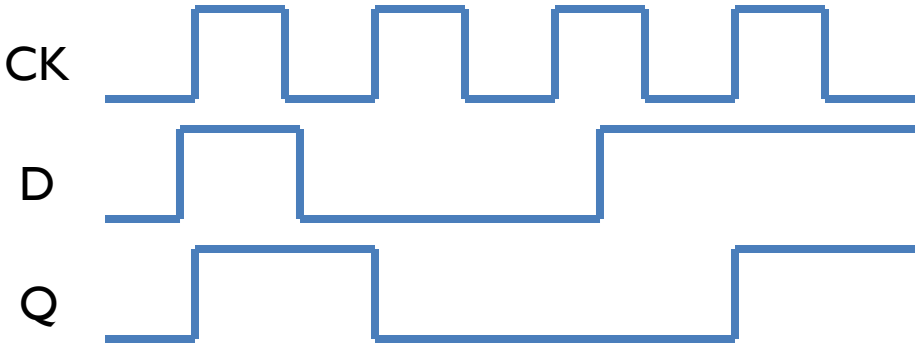
- クロック同期設計の構成要素
  - クロック同期回路（クロック同期型FF）
  - タイミング解析



クロック同期型FF

- デジタル信号を保持する
- 保持信号はクロックに合わせて更新する



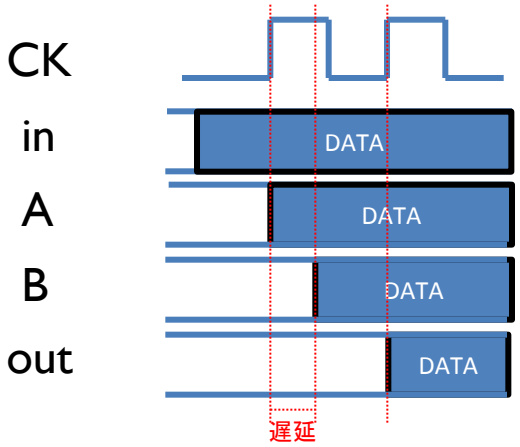


FFのHDL記述

```
module simple_ff (clk, nreset, d, q, enable);
        input clk, nreset, d, enable;
        output q;
        reg q_reg;

        assign q = q_reg;

        always @(posedge clk or negedge nreset)
                if (nreset == 1'b0)
                        q_reg <= 0;
                else if (enable == 1'b0)
                        q_reg <= q;
                else
                        q_reg <= d;
endmodule
```

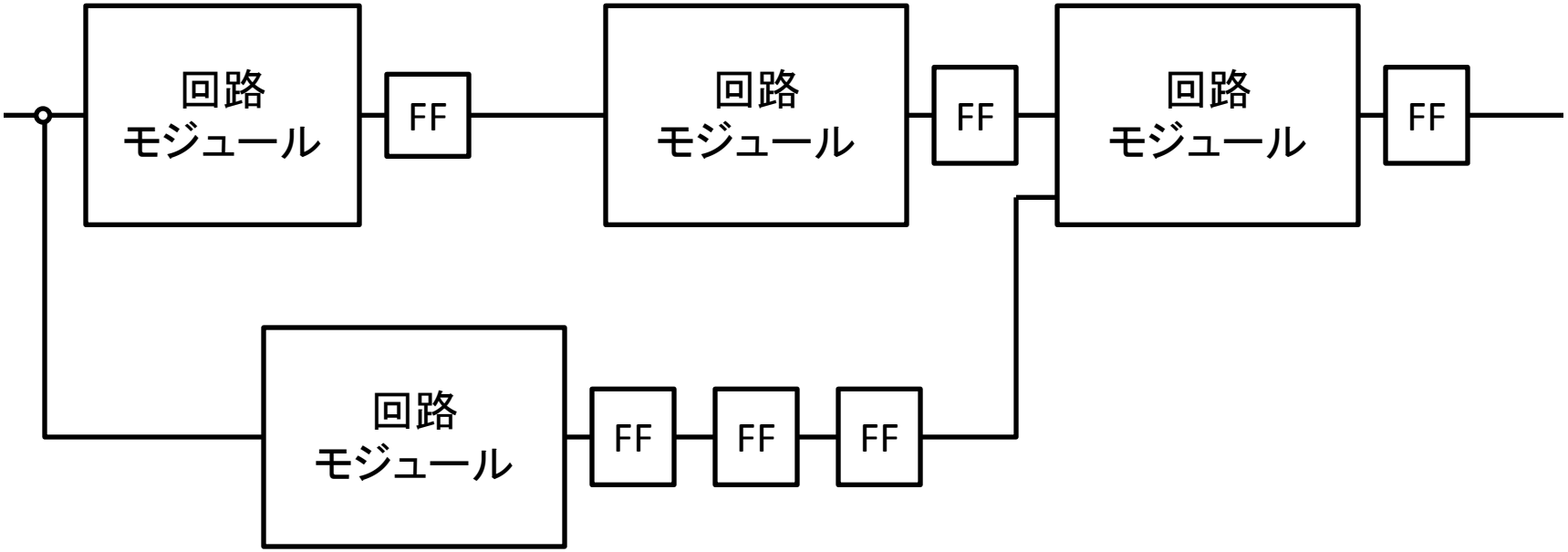
FFによるタイミング調整

- 回路をFFで囲む
  - クロックのedgeでFFの保持データ更新
  - 遅延がクロック幅を超えない限り、値の更新タイミングが固定
- 十分な時間予測性が確保される



FFによるモジュール化

- FFをIOとして回路を連結する
- クロック遅延の差はFF・トリガ信号で調整
- 並列処理の同期はFFの数で調整

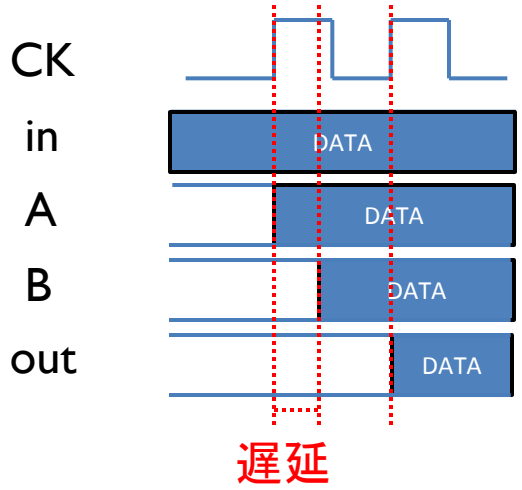


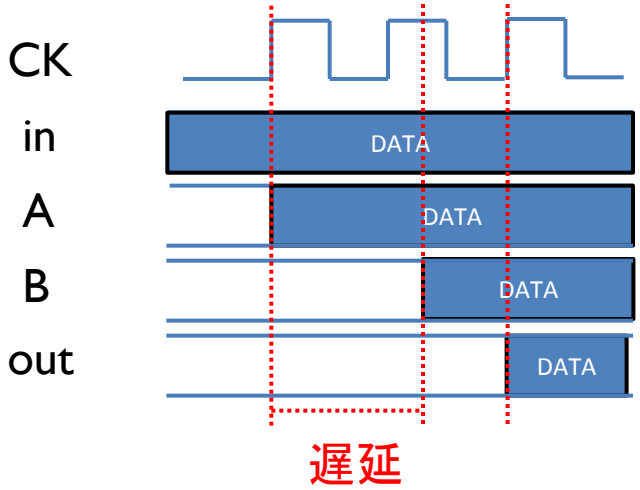
まとめ

- クロック同期型FFによるタイミング調整
  - 有効データの入出力をクロックエッジに限定
    - 電気的遅延をクロック間で吸収する
    - 十分な時間予測性が確保される
    - モジュール間の同期が可能となる
  - 並列処理設計はタイミング設計で実現
    - 例外：外部デバイスの排他制御、非同期信号との連携等

タイミング解析

- 意図したタイミングで信号を処理できるか





タイミングの遅延や誤差

- 一般的なタイミング誤差の原因
  - 電気的誤差
    - 外部デバイスの制御遅延
  - クロックの誤差
  - 多相クロックの合成誤差
  - 非同期信号、等々

- 予測は困難
  - 論理合成、配置配線のやり方で遅延が変化
  - 電圧や温度でも変化
  - 予測不能な非同期信号も存在

タイミング解析の手段

- 静的解析
  - パスベースの遅延評価
  - 論理合成、配置配線の評価
- 動的解析
  - モデルベーステスト
  - RTLシミュレーション
  - ゲートレベルシミュレーション
- 実機検証
  - ロジックアナライザ
  - セルフチェッキング

タイミング解析・設計の課題

- 定番の課題
  - 多相クロック
  - 非同期信号
  - 外部デバイスの誤差
  - 大規模回路のプローブ困難な内部信号

- FPGA設計は様々な課題を持つ
  - タイミング設計、ゲート数、消費電力、ノイズ、IPコア使用数…
- 現在はFPGAの高集積化が進んでいる背景からタイミング設計が一番のネックになりつつある

FPGA/HDLでの並列処理設計のまとめ

- 大事なこと：クロック同期設計で時間予測性を確保する
  - クロック同期型FF
  - タイミング解析
- 時間予測性を保障できれば、並列処理設計はクロック遅延ベースのタイミング設計で済む
  - タスク分割
    - モジュールの追加で実現。モジュール間の遅延はFFで調整
  - データ分割
    - 信号の幅を調整する、複数バスにするといった形で実現

ソフトウェア-FPGA協調処理（通信処理編）

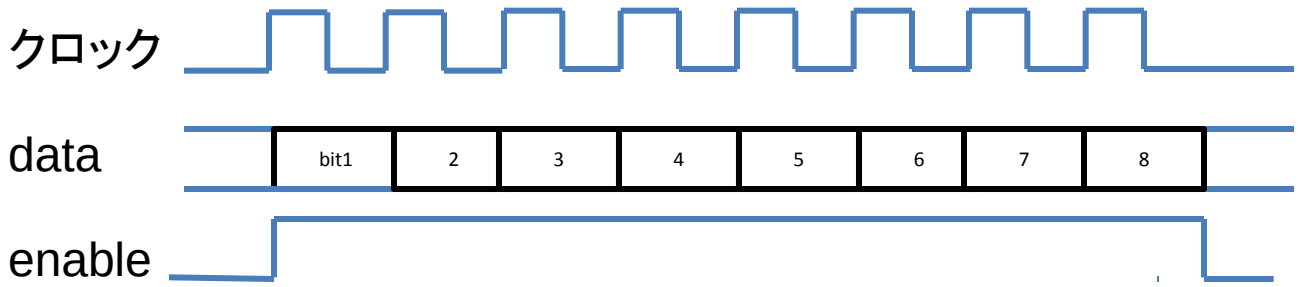
ソフトウェア/FPGAの協調処理

- 直列世界のソフトウェア
- 並列世界のHDL
  - 協調により、組込みソフトウェア並列処理の設計容易性を劇的に改善できる場合がある

- 今回は通信処理のタスク分割→並列処理化の例を紹介

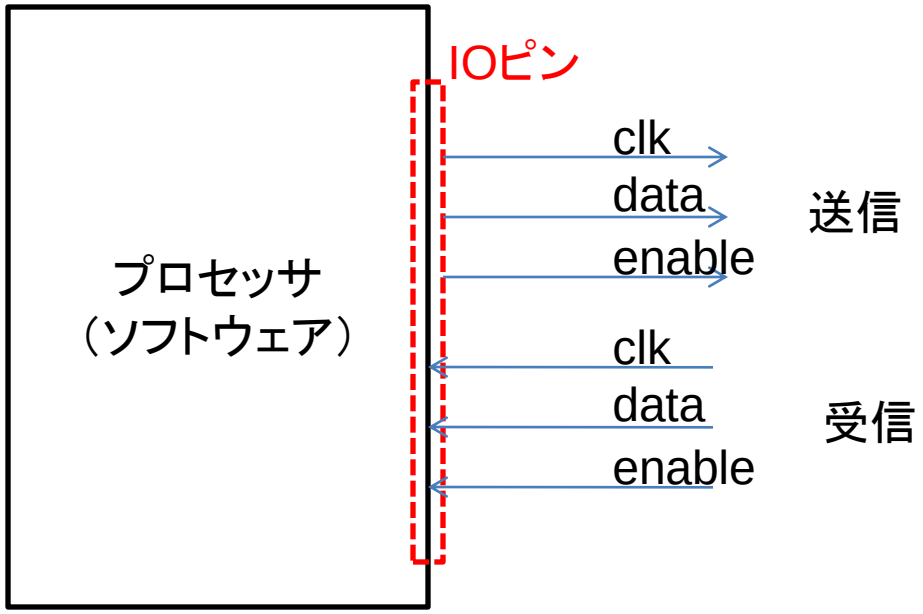
例題：通信仕様AAA

- クロックにあわせてシリアル伝送
  - クロックは絶対に1KHz
  - バイト単位で送受信。途切れることはない
  - データ送信中はenable信号を1に



ソフトウェアで開発する場合

- IOピンをファームウェアが直接制御

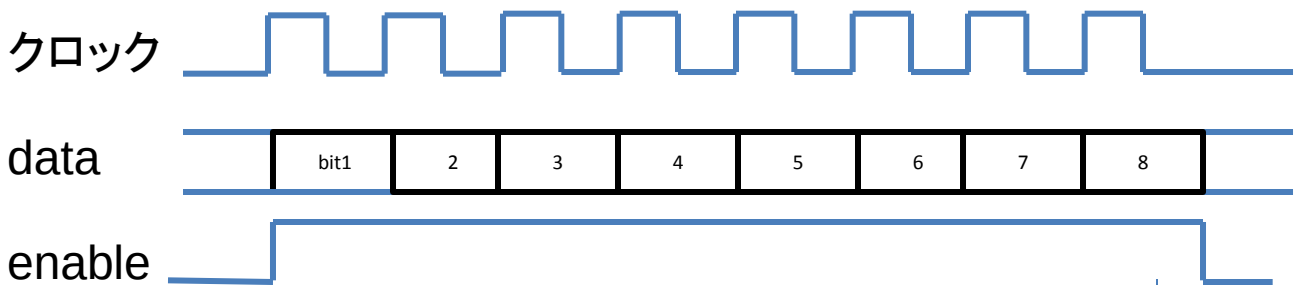


送信部

```
void AAA_send(unsigned char send_data)
{
        int i;
        clk_set(0);
        wait_us(100);
        enable_set(1);

        for (i = 7; i >= 0; i--) {
                clk_set(1);
                data_set((send_data >> i) & 1);
                wait_us(500);
                clk_set(0);
                wait_us(500);
        }
        wait_us(100);

        enable_set(0);

        wait_us(100);
}
```

受信部

```
unsigned char AAA_receive_isr(void)
{//enable信号割り込み
        unsigned char data = 0;
        int i;
        for (i = 7; i >= 0; i--) {
                while(clk_get() == 1);
                while(clk_get() == 0);

                data += (unsgined char)(get_data() << i);
        }
        return data;
]
```



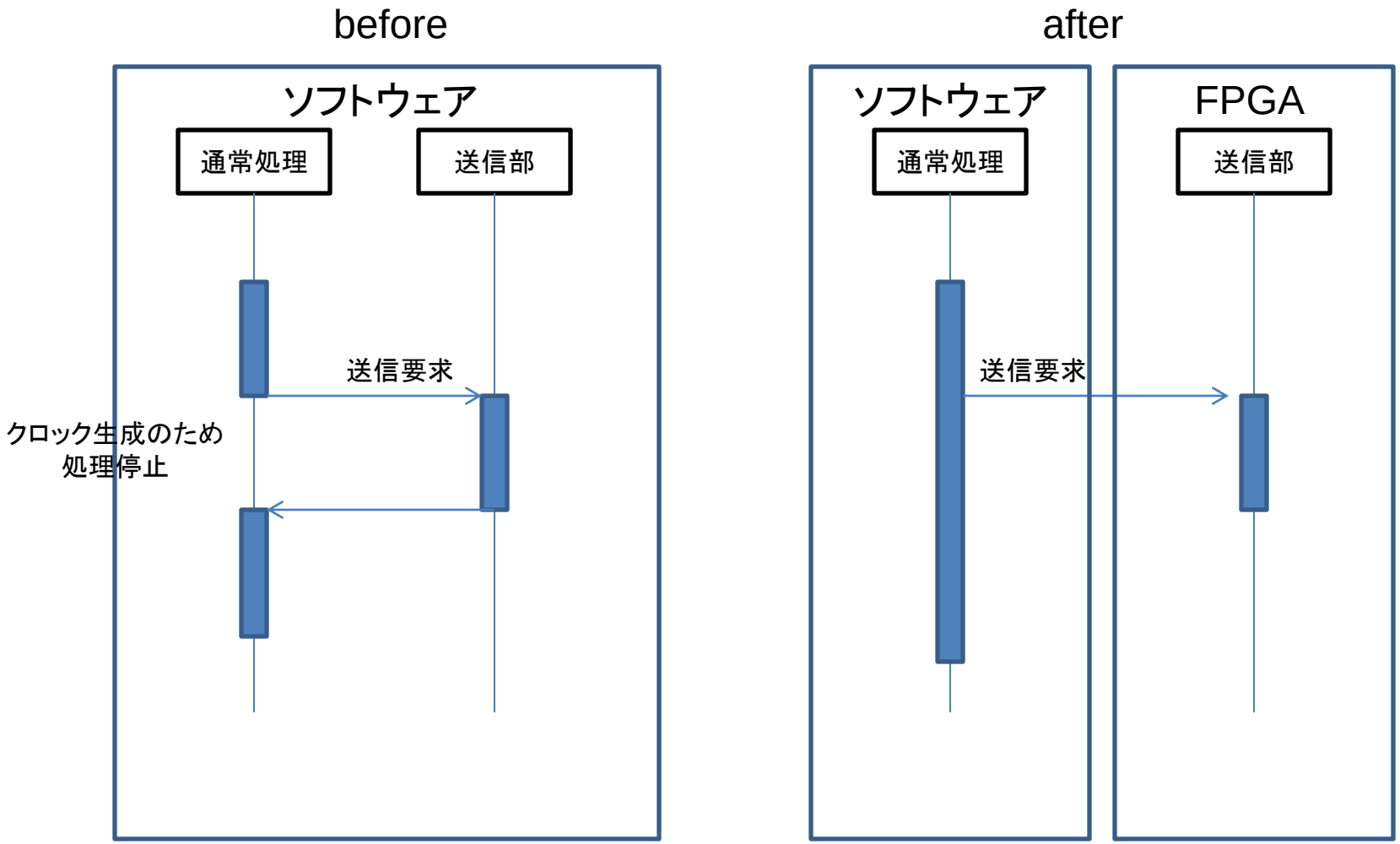
問題

- リアルタイム性の要求が厳しい
  - クロック生成処理保護→1コア1タスク＆割込み全ブロック
  - 受信漏れ防止→受信はenableエッジ検出後即時に開始
- 設計上のリスク
  - 長時間送受信を行う場合は？
  - 送信中に受信データが来た場合は？
- 複雑なタイミング調整が要求される
  - 設計、アーキテクチャが爆発
  - マルチコア化対応も非現実的
    - 最大同時送信数数、最大同時受信数分、コアを増やす？

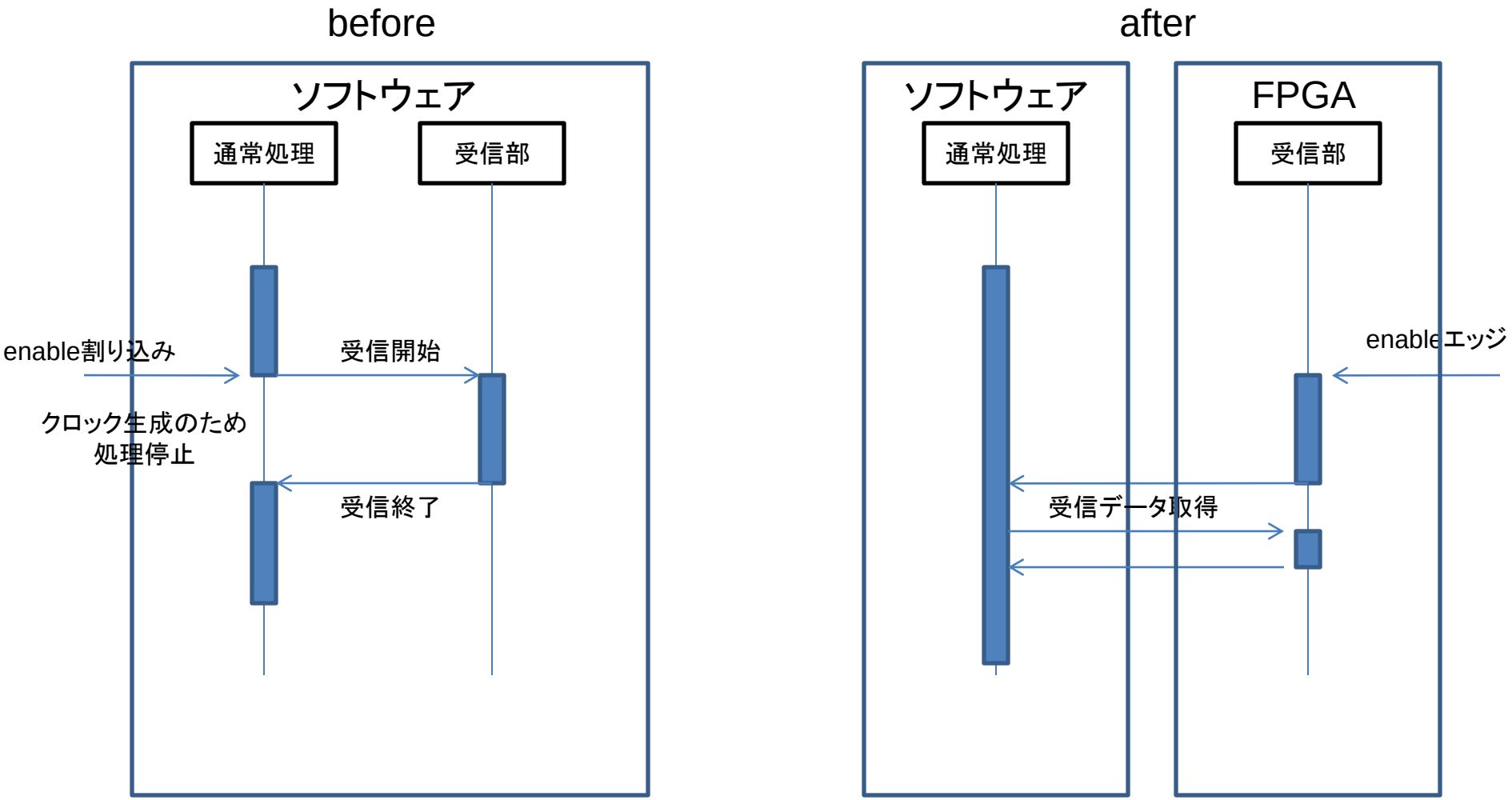
並列処理化

- ソフトウェア処理をタスク分割し、FPGAに委譲
- コアを増やさず処理を並列化する

FPGA/HDLへの処理の委譲（送信）

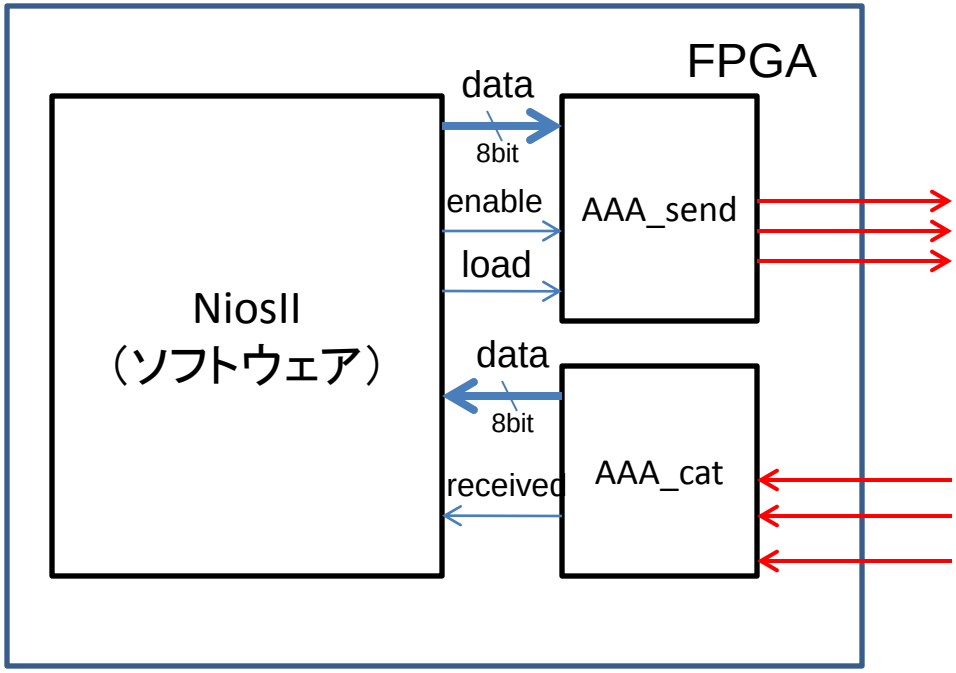


FPGA/HDLへの処理の委譲(受信)



モジュール接続

- 通信にFPGAモジュールを介する



送信部（FPGAモジュール）

```
module AAA_send(clk, enable, nreset, load, send_data, out_data, out_clk, out_enable);
  input clk, enable, load, nreset;
  input [7:0] send_data;
  output out_data, out_clk, out_enable;
  reg [7:0] reg_data;
  reg [3:0] send_cnt;

  wire sending, sending_d1;
  wire enable_d1;

  simple_ff enable_ff(.clk(clk), .nreset(nreset), .d(enable), .q(enable_d1), .enable(1));
  simple_ff sending_ff(.clk(clk), .nreset(nreset), .d(sending), .q(sending_d1), .enable(1));

  assign sending = enable_d1 & ~load;
  assign out_data = (enable_d1) ? reg_data[7] : 0;
  assign out_clk = clk;//今回は簡略化
  assign out_enable = (send_cnt == 4'h8) ? 0 : enable_d1;
```

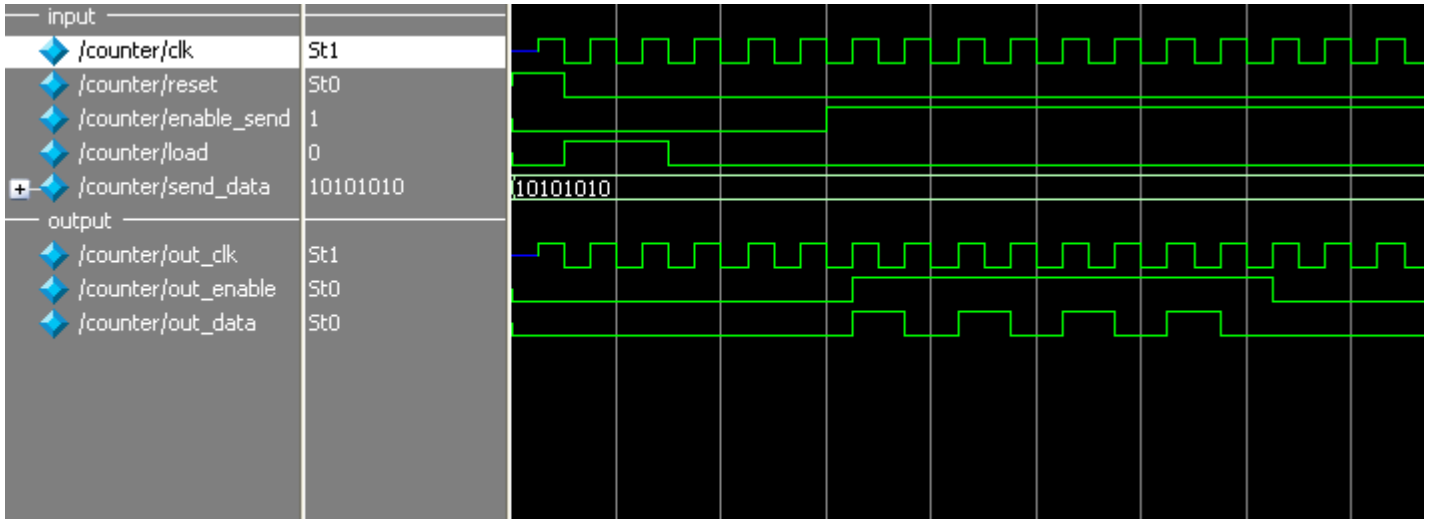
（続く）

送信部（FPGAモジュール）

（続き）

```
 always @(posedge clk or negedge nreset)
  if (nreset == 0) begin
    send_cnt <= 4'h8;
    reg_data <= 0;
  end else if (load) begin
    reg_data <= send_data;
    send_cnt <= 0;
  end else if (send_cnt == 4'h8) begin
    //何もしない
  end else if (sending) begin
    reg_data <= reg_data << 1;
    send_cnt = send_cnt + 4'b1;
  end

endmodule
```

送信部（FPGAモジュール）



送信部（ソフトウェア）

```
void AAA_send(unsigned char send_data)
{
        enable_set(0);
        wait_us(100);

        data_set(send_data);
        wait_us(100);

        load_set(1);
        wait_us(100);
        load_set(0);
        wait_us(100);

        enable_set(1);
        wait_us(100);
}
```

受信部(FPGA)

- FPGAモジュール

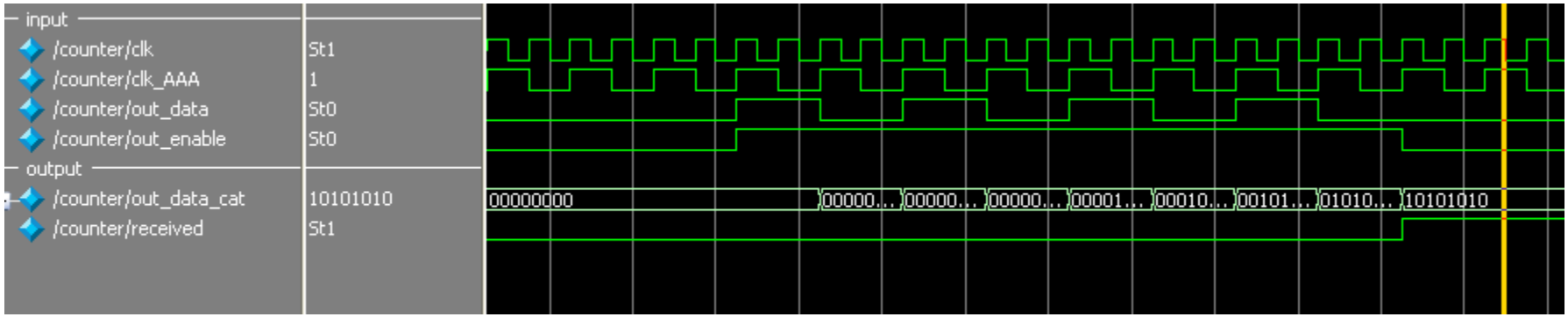
```
module AAA_cat(clk_AAA, nreset, enable, in_sig, out_data, received);
  input nreset, in_sig, enable, clk_AAA;
  output [7:0] out_data;
  output received;

  reg [3:0] receive_cnt;
  reg [7:0] receive_data;

  assign out_data = receive_data;
  assign received = (receive_cnt == 4'h8) ? 1'b1 : 1'b0;

  always @(posedge enable) begin
    receive_cnt <= 0;
    receive_data <= 0;
  end
（続く）
```

受信部(FPGA)

- FPGAモジュール

```
（続き）
 always @(posedge clk_AAA or negedge nreset) begin
   if (nreset == 0) begin
     receive_data <= 0;
     receive_cnt <= 0;
   end else if (receive_cnt == 4'd8) begin
    //何もしない
   end else if (enable) begin
     receive_data <= {receive_data, in_sig};
     receive_cnt <= receive_cnt + 1;
   end
 end
endmodule
```

受信部（FPGA）


input
/counter/clk
St1
/counter/clk_AAA
1
/counter/out_data
St0
/counter/out_enable
St0
output
/counter/out_data_cat
10101010
/counter/received
St1
00000000
00000...
00000...
00000...
00001...
00010...
00101...
01010...
10101010

受信部（ソフトウェア）

```
unsigned char AAA_get_isr(void)
{//received割り込み
        unsigned char data;

        data = data_read();

        return data;
}
```

改善点

- 改善点
  - (問題)シングルタスク、割り込み全ブロック
    →割り込みセーフに。非シングルタスクも化
  - (問題)割り込み後即時受信処理に
    →1バイト分まで保持可

- 拡張性も向上
  - 送信処理時間の拡大
    - FFやRAMで送信データをバッファリング。ソフトのアクセス数を削減
  - 大量の受信データ
    - バッファリング用のFFやRAMを用意。余裕を拡大

- クリティカルな処理が並列処理化
- 並列処理の設計容易性が改善

協調処理の設計プロセス

ソフトウェア/FPGA設計プロセス

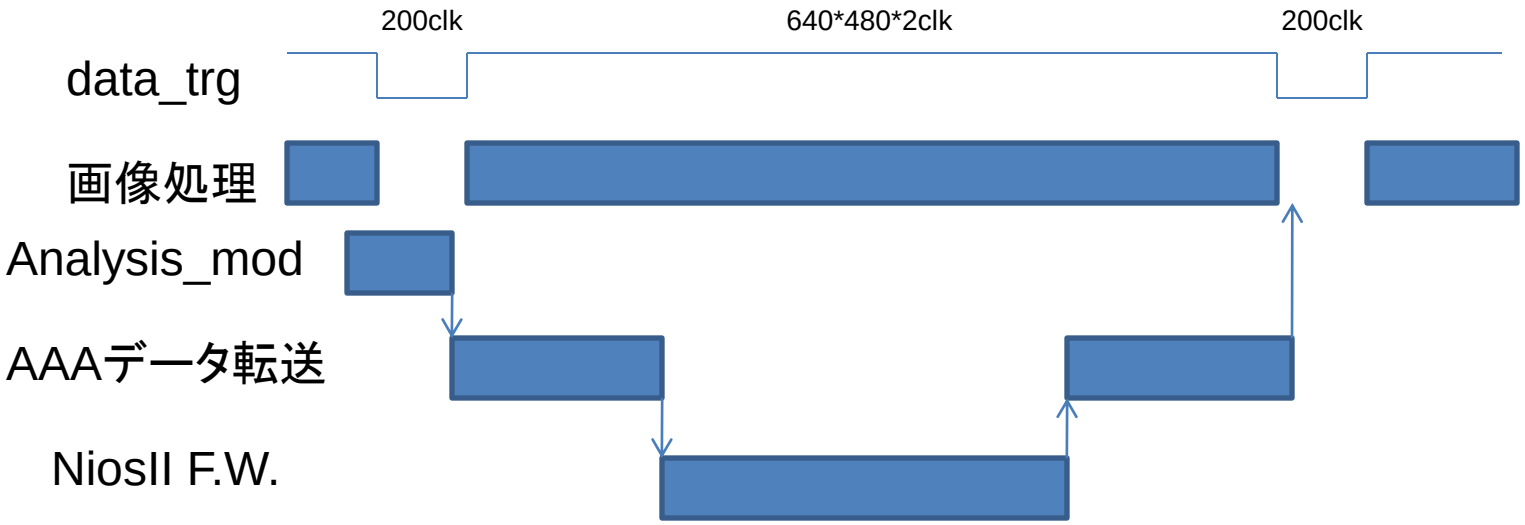
１　仕様設計
２　機能設計
３　機能分割
４　実装
５　テスト
６　QA

並列処理設計で特に大事なプロセス

- 機能設計、機能分割
  - アーキテクチャ設計
  - アルゴリズム設計

アーキテクチャ設計

- 協調処理の統合設計を行う
  - FPGA/プロセッサの必要条件を明らかにする
    - IPコア、ゲート数、クロック周波数…
  - アーキテクチャレベルのタイミング設計
    - ソフトウェア・FPGA間のタイミング設計
    - 外部デバイスとのタイミング設計



アーキテクチャ設計

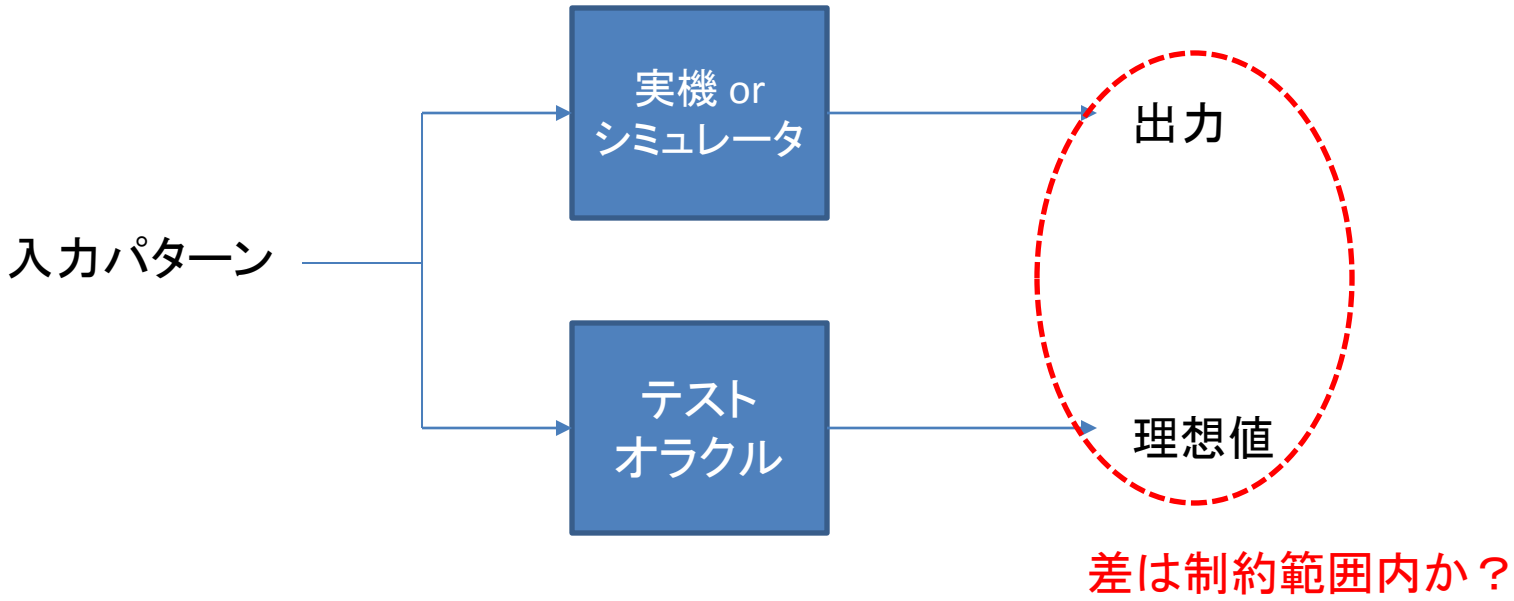
- アーキテクチャ・プロトタイピング
  - モデルや設計のシミュレータで検証
    - モデル駆動開発、モデルベーステスト
    - システム設計言語によるシミュレーション
      - SystemC、SystemVerilog
  - ソフトウェアやスクリプトでシステムの挙動を検証
    - Cベースによる実装
    - MATLAB等スクリプトによる実装

アルゴリズム設計

- 理想アルゴリズムと精度制約を設計する
  - FPGAへの委譲によってアルゴリズム精度は一般的に低下
    - ex)周波数フィルタ→FIRフィルタ
    - ex)浮動小数点→固定小数点
    - ex)32bit整数→8bitデータ
  - 上流設計でテストオラクルを確保
    - タスク分割、タスク委譲時はオラクルを使った動的解析を行う

アルゴリズム設計

- 動的解析
  - 品質を検証するのに十分な入力パターンを用意
  - その入力パターンで出力がテストオラクルと精度制約内で一致するか検証



まとめ

まとめ

- FPGA/HDLは並列処理と親和性が高い
- アーキテクチャの工夫で、FPGA/HDLをソフトウェア的に扱えるようになる
  - ソフトプロセッサ
- FPGA/HDLとの協調により、ソフトウェア並列処理の設計容易性が大きく改善することがある