修 士 論 文

RTL とゲートレベルを混在させた最適な論理回路設計に関する研究

北陸先端科学技術大学院大学
情報科学研究科情報科学専攻

ZHANG, Zhifei

2014 年 3 月

修 士 論 文

RTL とゲートレベルを混在させた最適な論理回路設計に関する研究

指導教官　田中　清史　准教授

審査委員主査　田中　清史　准教授
審査委員　井口　寧　教授
審査委員　金子　峰雄　教授

北陸先端科学技術大学院大学
情報科学研究科情報科学専攻

ZHANG, Zhifei

提出年月：2014 年 2 月

概要

ASIC や FPGA 内に実現される論理回路の設計において、かつては回路図入力方式が一般的であったが、回路集積化技術の向上に伴って、手作業による回路図作成は限界に達しつつあり、近年は Verilog HDL や VHDL などのハードウェア記述言語による HDL 設計が主流になっている。この流れは、更に C 言語などのより高位に位置する言語を使用する設計へと移行する傾向がある。ハードウェア記述言語による設計は、設計者にとって、かつての煩雑な作業フローを大幅に改善でき、回路の細かい部分を考慮せずに動作のみを記述すればよく、開発効率の観点からは高効率であることは間違いない。しかし、言語を使用することは高位合成や論理合成など、回路図入力方式においては存在しなかった工程の追加を強いられ、これにより、物理的に実現不可能な回路を記述すればエラーを起こす。あるいは、設計者の深い知識／経験に基づいた低階層での高度な回路作成が不可能であるといった点がある。このことから、ハードウェア記述言語の使用は回路設計において必ずしも有利とは限らない。

本研究は、Xilinx 社の FPGA を開発するために不可欠な総合的な開発ソフトウェアである ISE Design Suite 13.2 を使用して、デザイン入力、シュミレーション、論理合成、マッピング、配置・配線、プログラミングというステップで回路作成を行う。HDL 設計方式と回路図入力設計方式を使用して以下のタイプの回路を設計し、Spartan-3E Starter Kit Board[7]をターゲットとしたインプリメントを行い、スライス数、LUT 数、最大遅延で比較し、どちらの設計方式がより優れるかを評価することで考察を行う。

(1)加算器　(2)マルチプレクサ　(3) 7 セグメントデコーダー　(4)トライステート
(5)シフトレジスタ　(6)カウンタ　(7)ステートマシン　(8)CPU 回路

評価では、各対象回路に対して複数の HDL 設計、複数の回路図設計を用意し、HDL 設計間の比較、回路図設計間の比較、HDL 設計と回路図設計間の比較を行う。更に、規模の大きな回路として CPU を HDL と回路図の両方を使用して階層的に設計し、HDL と回路図の階層的な組合せをいくつか用意し、比較を行う。

生成された回路を比較した結果、HDL 設計方式を使用する大きな利点は、設計抽象度を引き上げることで、HDL 記述を合成ツールで最適化する余地を確保できる点であることがわかった。ゲートレベル記述へ変換された回路は、回路図設計方式で生成されるものと比較して、より小規模な回路が生成される傾向があるほか、高速演算専用のキャリーロジックなどが多用されていることから、特に設計規模の大きな回路において、回路図設計方式よりも最大遅延が短くなる傾向があった。一方で、設計規模が小さい回路では、逆に回路図設計が高い評価の回路を生成する可能性もある。HDL 設計と回路図設計を適切に混在させ、最適なシステム設計を実現することが重要である。

同一の回路に対して、異なる HDL 設計方式は、設計抽象度の差異により最適化の余地と使用される高速演算ロジックの数が異なるため、評価値に差が出た。異なる回路図設計方式でも、構造の差異により論理規模が異なるため、評価も異なる結果となった。さらに、

混在設計に関する評価結果から、評価が高い部分回路を含めることにより、回路全体の評価が高くなることがわかった。複雑な大規模回路の設計を行う際には、極力多くのモジュール化と階層化を行い、一つのモジュールの規模を小さくすることで、モジュール毎にHDL設計、回路図設計の最適なものを選択できる幅が広がる。各モジュールを組み合わせることにより、全体の回路の最適性を向上させることが期待できる。

最適的な回路の作成が求められる場合に、両設計方式でそれぞれ設計した結果を比較した上で決定することが可能となるが、設計期間や人的な余力を考慮すると、どのような複雑さと規模の回路に対して、どちらの設計方式を採用するべきかについての基準を構築することが今後の課題である。

目次

図目次

表目次

第１章　はじめに

1.1　背景と目的

ASICやFPGA内に実現される論理回路の設計において、かつては回路図入力方式が一般的であったが、回路集積化技術の向上に伴って、手作業による回路図作成は限界に達しつつあり、近年はVerilog HDLやVHDLなどのハードウェア記述言語（Hardware Description Language: HDL）による設計が主流になっている。この流れは、更にC言語などのより高位に位置する言語を使用する設計へと移行する傾向がある。ハードウェア記述言語による設計は、設計者にとって、かつての煩雑な作業フローを大幅に改善でき、回路の細かい部分を考慮せずに動作のみを記述すればよく、開発効率の観点からは高効率であることは間違いない。その他、言語を使用する設計の利点として、通常のエディタによる入力・編集が可能、論理式レベルや真理値表で考える必要が少ない、コンポーネント単位での再利用の容易さ、ゲートレベルのライブラリに非依存な記述が基本であるためターゲットデバイスの変更が問題とならないなどがある。しかし、言語を使用することは高位合成や論理合成など、回路図入力方式においては存在しなかった工程の追加を強いられ、これにより、物理的に実現不可能な回路を記述すればエラーを起こす。あるいは、設計者の深い知識／経験に基づいた低階層での高度な回路作成が不可能であるといった点がある。このことから、ハードウェア記述言語の使用は回路設計において必ずしも有利とは限らない。

本研究は、論理回路設計をターゲットとして、ゲートレベル設計（回路図入力設計）方式とRTL（HDL）設計方式を使用して以下のタイプの回路を設計し、生成される回路を各種評価項目で比較し、最適な論理設計の指針を与えることを目的とする。

(1)加算器　(2)マルチプレクサ　(3) 7セグメントデコーダー　(4)トライステート
(5)シフトレジスタ　(6)カウンタ　(7)ステートマシン　(8)CPU回路

すなわち、設計する回路のタイプによって、RTL（HDL）で記述すべき回路、ゲートレベル（回路図入力）で設計すべき回路を明らかにすることが狙いである。これにより、大規模のハードウェアを設計する際に、HDL設計と回路図設計を適切に（階層的に）混在させることによる最適なシステム設計の実現を可能とすることを目的とする。

1.2　ターゲットデバイス－ Field Programmable Gate Array (FPGA)

FPGA は内部回路をプログラミングできる集積回路であり、現在多くの電子機器分野で

使用されている。その最大の特徴として、内部が基本的にメモリで構成されており、そのメモリの内容を書き替えることにより、ロジックが変更され、様々な回路を構成することができることが挙げられる。本研究はFPGAをターゲットとして回路を設計し、設計方式間の評価比較を行う。(FPGA の主要ベンダの一つである Xilinx 社の FPGA をターゲットとする。)

FPGAをターゲットとする回路の設計において、設計方式の選択肢としては回路図入力、HDLの使用、および高級言語による設計が存在する。近年は回路図入力よりもHDLの使用が一般的であり、さらに高位に位置する C 言語などの高級言語の使用は今後の有力な設計方式として期待されている。

現在の主流である HDL を使用する方式では、まずは全体の回路を機能ごとにいくつかのモジュールに分けて設計する。HDL言語で記述されたモジュールの正当性を検証するために、論理シュミレーションを行う。エラーや予想外の波形図が出る場合、そのモジュールを特定し修正を行い、問題がないモジュールを階層的に結合して回路を構成することができる。続いて論理合成によりゲートレベル回路に変換し、回路全体が予想通り動作するかについて再度テストベンチを利用してシュミレーションを行う。入力、出力などの波形図を考察し、問題が消化するまで各モジュールの修正を繰り返して行う。(ただし、FPGAをターゲットとする開発では、実際のデバイスを使用した動作テストが容易であるため、このゲートレベルシミュレーションはしばしば省略される。) 最後にターゲット FPGA に対するマッピングおよび配置・配線を行い、タイミング遅延、実際のFPGA上で動作などを検証する。一方、回路図入力方式によって設計する場合は、上記のゲートレベル回路に相当するものを回路図として作成する。その後のフローは同様である。

ASIC 分野においては、大規模化にともないHDLによる設計が一般的になってきたが、FPGA をターゲットとして設計した場合、HDL 方式と回路図入力方式とで生成される回路がどのように異なるか、さらに、それらの間でどの程度の性能の差があるのかは明らかにされていないため、本研究で両方式による生成回路の特徴を明らかにする。

1．3　研究方法

1.3.1　研究ツール

本研究で使用するツールは、Xilinx社のFPGAを開発するために不可欠な総合的な開発ソフトウェア、ISE Design Suite 13.2であり、ゲートレベル設計方式やRTL設計方式、論理合成､回路検証､配置配線というFPGA開発フローでの各工程の機能をサポートしている。このツールでFPGAを開発する場合、いくつかのステップに分けられる。

①デザイン入力：HDL言語や回路図入力により回路をデザインし、ソースファイルを作成す

る。
②シュミレーション：テストベンチを利用してデザインの機能を検証する。一般的には波形図を観察し、入力信号や出力信号を確かめる方法で行う。
③論理合成：作成されたソースをコンパイルし、ゲートレベル回路記述に変換する。HDL 言語で作成されたソースが対象であり、回路図入力デザインの場合は単純な変換処理がなされるのみである。
④マッピング：ゲートレベル回路の FPGA の内部資源（ルックアップテーブルやフリップフロップ）への論理的な割当を行う。
⑤配置・配線：使用するルックアップテーブル、フリップフロップの配置の決定、およびそれらの間の配線を決定する。
⑥プログラミング：生成された最終ファイルをデバイスにダウンロードし、実行する。

1.3.2　評価手法

本研究では、HDL 設計と回路図入力の両設計方式で作成したそれぞれの回路に対して、以下のように回路規模を示すスライス数やルックアップテーブル（LUT）数、最大遅延という内部指標を比較しながら、どちらの設計方式がより優れるかを評価することで考察を行う。
①スライス数：一定数の LUT、フリップフロップ（FF）からなる単位である。
②LUT 数： 4 入力（あるいは 6 入力）1 出力の論理を実現する単位であり、SRAM で構成される。([5][6])
③最大遅延：FPGA 内部リソースへのマッピング、および配置配線後、入力（あるいは FF の出力）信号から、出力（あるいは FF の入力）信号までの遅延時間が一番長くなるパスである。

評価では、各対象回路に対して複数の HDL 設計、複数の回路図設計を用意し、HDL 設計間の比較、回路図設計間の比較、HDL 設計と回路図設計間の比較を行う。更に、規模の大きな回路として CPU を HDL と回路図の両方を使用して階層的に設計し、HDL と回路図の階層的な組合せをいくつか用意し、比較を行う。

1.4　本研究の貢献

HDL 設計方式は、設計期間の短縮が期待できることに加え、不具合修正の効率化、ライブラリ化可能などのメリットが挙げられる。かつて回路図設計方式を利用していた設計者は HDL 設計に移行する傾向が高まっている。一方、論理回路設計において回路図設計方式の利点がいくつか挙げられる。回路図設計方式は回路構造の細部まで指定可能であり、HDL

設計方式のように論理合成工程の追加を強いられず、回路の形態・規模・性能的に設計者の意図に近い回路を作成することが可能である。以上から、HDL 設計方式と回路図設計方式にはメリット、デメリットが存在しており、一概にどちらが有利、不利とは言えない。本研究では、両方式で作成した回路に対して、規模と最大遅延などの内部指標に関して比較し、ぞれぞれがどのような特徴を持つのかを明らかにすることを目的とする。それらの結果に基づいて、最適な回路の設計を実現するために、HDL 設計を行うべきか、回路図設計を行うべきか、あるいは両設計方式をどのように階層的に組み合わせるかによって最適な回路となるかが導かれる。

1.5　本論文の構成

本論文の構成は以下の通りである。

第 2 章は、本研究で選定したタイプの回路を HDL 設計、回路図設計で作成した結果と、シュミレーションによる動作確認について説明する。第 3 章は、設計した回路に対して、HDL 設計と回路図設計間で回路規模および動作性能について比較を行う。第 4 章は、本論文をまとめる。

第 2 章　回路設計

本章では、本研究で選定した回路タイプ(加算器、マルチプレクサ、7 セグメントデコーダー、トライステート、シフトレジスタ、カウンタ、ステートマシン、CPU 回路)を HDL 設計方式、回路図設計方式を用いて設計した結果について、それぞれ回路の概要を説明する。

2.1　組み合せ回路の設計

組み合わせ回路は、過去の入力と依存がなく、現在の入力のみによって出力を決定する、情報を記憶しない回路である。

2.1.1　加算器

2.1.1.1　加算器の概要

加算器は計算機で使用される基本的な演算装置の一つである。最も簡単な加算器は半加算器と全加算器を使用して構成することができる。1 ビット半加算器は、二つの 1 ビット 2 進数(通常 2 つのオペランドの最下位の桁同士)を加算し、その桁の加算値と桁上げを出力する回路である。1 ビット全加算器は、二つの 1 ビット 2 進数(通常 2 つのオペランドの最下位以外の桁同士)と下位桁からの桁上げを加算し、その桁の加算値と桁上げを出力する回路である。32 ビットの加算器を構成するためには、最下位桁のための半加算器 1 個と、その他の桁のための全加算器 31 個を組み合わせることになる。この他に、HDL 言語の提供する演算子を使用する簡単な設計方法が存在する。以下では HDL による加算器の 2 種類の設計方法、および回路図による設計について説明する。

2.1.1.2　加算器の HDL 設計

HDL 設計①

最も簡単な加算器の HDL 設計は、Verilog HDL が提供する“+” 演算子を利用する方法である。2 つの入力信号(IN_A、IN_B)に対して+演算子で加算を行い、加算値出力信号(OUT_S)および桁上げ出力信号 (CARRY_OUT) に代入する記述により、加算器が実現可能である。以下に Verilog HDL コードを示す。

```
module count(
    input [31:0] IN_A,
    input [31:0] IN_B,
    output [31:0] OUT_S,
    output CARRY_OUT
    );
assign {CARRY_OUT, OUT_S} = IN_A + IN_B;
endmodule
```

HDL 設計②

1 ビット半加算器モジュールと 1 ビット全加算器モジュールを Verilog HDL で記述し、上位モジュールによってそれらを接続することにより、加算器が実現可能である。以下に Verilog HDL コードを示す。adder_half が半加算器モジュール、adder_full が全加算器モジュール、count が上位モジュールである。

1 ビット半加算器モジュール

```
module adder_half(
    input A,
    input B,
    output S,
    output C
    );
    assign S = A ^ B;
    assign C = A & B;
endmodule
```

1 ビット全加算器モジュール

```
module adder_full(
    input A,
    input B,
    input Cin_in,
    output S,
    output Cin_out
    );
    wire Z1,C1,C2;
```

```
    adder_half adder_half1(A,B,Z1,C1);
    adder_half adder_half2(.A(Z1),.B(Cin_in),.S(S),.C(C2));
    assign Cin_out = C2 ^ C1;
endmodule
```

上位モジュール

```
module count(
    input [31:0] A,
    input [31:0] B,
    output [31:0] S,
    output C
    );
    wire U0_C,U1_C,U2_C,U3_C,U4_C,U5_C,U6_C,U7_C,U8_C,U9_C,U10_C,
    U11_C,U12_C,U13_C,U14_C, U15_C,U16_C,U17_C,U18_C,U19_C,U20_C,
    U21_C,U22_C,U23_C,U24_C,U25_C,U26_C,U27_C,U28_C,U29_C,U30_C;
    adder_half S0(.A(A[0]),.B(B[0]),.S(S[0]),.C(U0_C));
    adder_full S1(.A(A[1]),.B(B[1]),.Cin_in(U0_C),.S(S[1]),.Cin_out(U1_C));
    adder_full S2(.A(A[2]),.B(B[2]),.Cin_in(U1_C),.S(S[2]),.Cin_out(U2_C));
    adder_full S3(.A(A[3]),.B(B[3]),.Cin_in(U2_C),.S(S[3]),.Cin_out(U3_C));
    //  省略
    adder_full S31(.A(A[31]),.B(B[31]),.Cin_in(U30_C),.S(S[31]),.Cin_out(C));
endmodule
```

2.1.1.3　加算器の回路図設計

1 ビット半加算器の真理値表は、入力信号を A と B、和を S、キャリーを C として表 2.1 のように表わされる。

表 2.1　1 ビット半加算器の真理値表

A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

半加算器の真理値表から論理式を導くと、S=A⊕B、C=A*B となり、それにしたがって、

半加算器の回路図は図 2.1 のようになる。

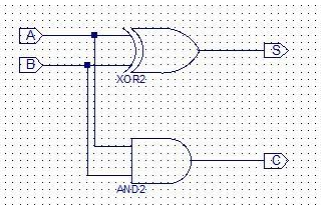

図 2.1　1 ビット半加算器回路図

1 ビット全加算器の真理値表は、入力信号を A と B、下位からの桁上げ入力信号を X、和を S、上位への桁上げ信号を C として表 2.2 のように表わされる。

表 2.2　1 ビット全加算器の真理値表

A	B	X	S	C
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

全加算器の真理値表から論理式を導くと、S=A⊕B⊕X、C=A*X+B*X+A*B となり、それにしたがって、全加算器の回路図は図 2.2 のように、2 個の半加算器と 1 個の OR 論理ゲートから構成することができる。

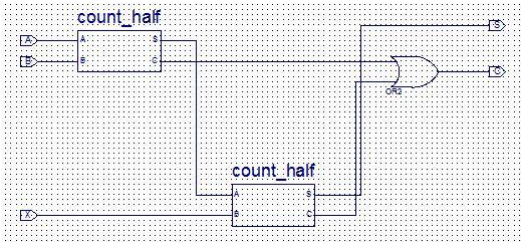

図 2.2　1 ビット全加算器回路図

32 ビット加算器は、最下位の桁同士の加算は半加算器を利用し、ほかの桁同士の加算は全加算器を利用し、それらを図 2.3 のように組み合わせることで実現される。

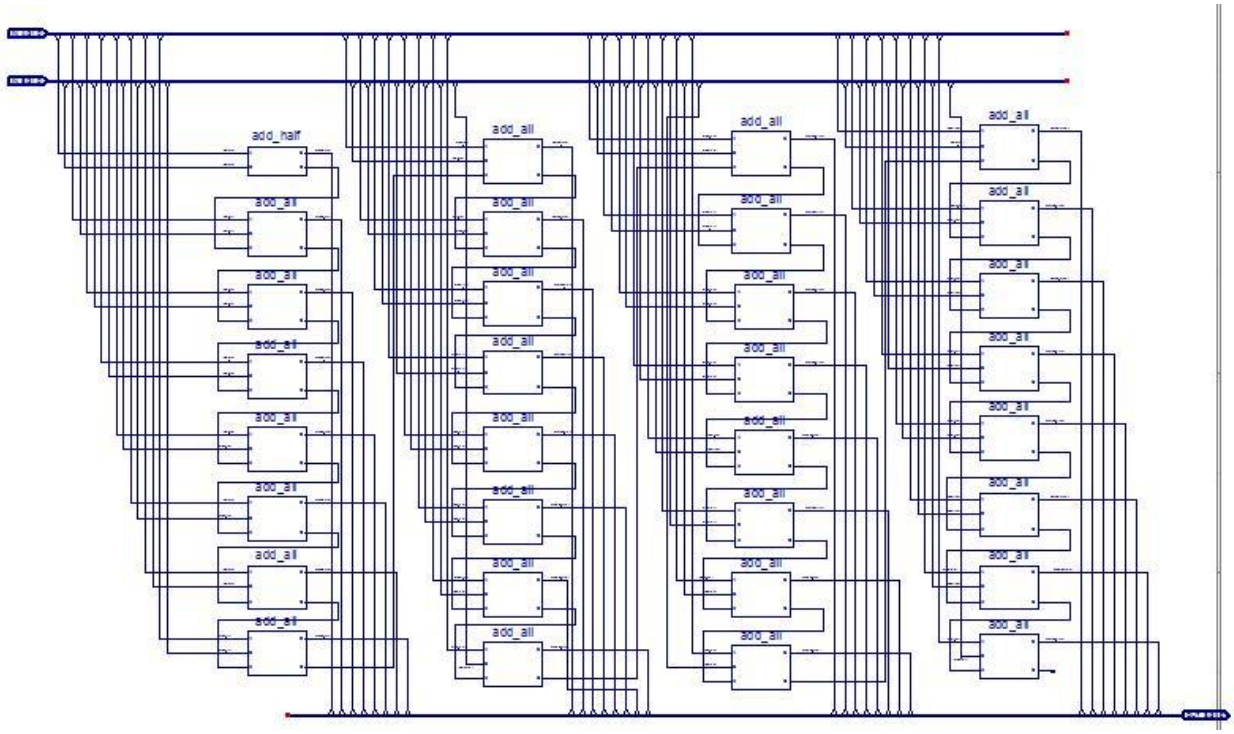

図 2.3　32 ビット加算器回路

2.1.1.4　加算器のシュミレーション

加算器の入力信号と出力信号を下記のように設定し、波形図を観察する。

入力信号 A：11

入力信号 B: 8

出力信号 S：19

HDL 設計①のシュミレーション

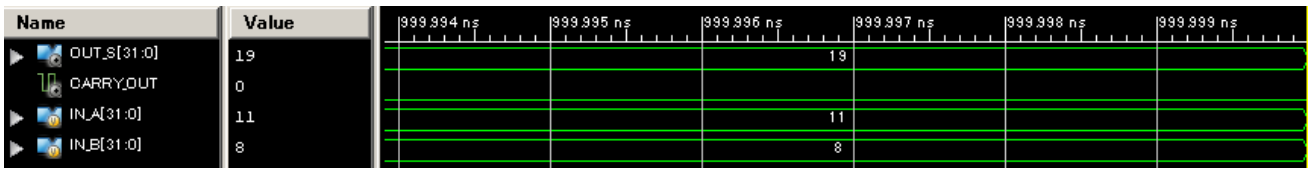

図 2.4　HDL 設計①のシュミレーション

HDL 設計②のシュミレーション

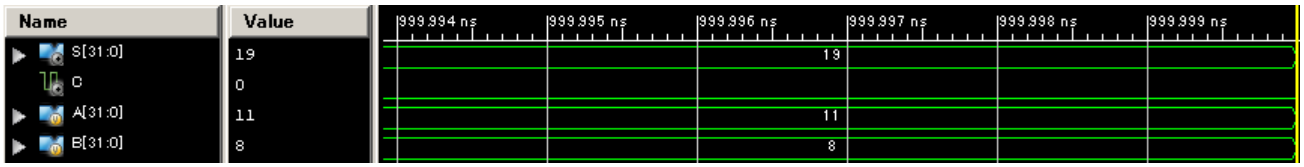

図 2.5　HDL 設計②のシュミレーション

回路図設計のシュミレーション

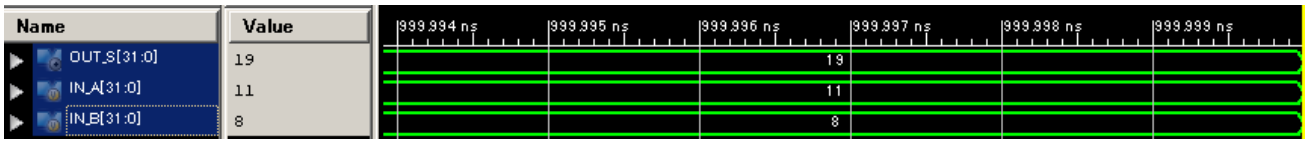

図 2.6　回路図設計のシュミレーション

2.1.1.5　キャリー先読み加算器の構成

前述のように、必要な個数の半加算器、全加算器を接続することにより、任意の桁数の加算器を構成することができるが、桁数が増加するにしたがって、ゲート数とゲート段数が多くなる。演算は最下位の桁から順次に桁上げ計算を行うため、回路全体の合計遅延時間が大きくなり、演算速度が低下することになる。したがって、このような順次桁上げ加算器は高性能計算機ではほとんど使用されない。高速化を目的とする加算器は、キャリー先読みと呼ばれる構造で設計することができ、幅広く使用されている。

キャリー先読み加算器において、n 桁目の桁上がりは次の式で与えられる。

Cn=An*Bn+(An+Bn)*Cn-1

ここで、Cn は n 桁目からの桁上がり、An と Bn は n 桁目の入力信号、Cn-1 は n-1 桁目からの桁上がりとなる。Gn= An*Bn、Pn= An+Bn とすると、4 ビットキャリー先読み加算器の各桁上がり信号群は以下のようになる。

C0=G0+P0*C-1

C1=G1+P1*C0=G1+P1*(G0+P0*C-1)=G1+P1*G0+P1*P0*C-1

C2=G2+P2*C1=G2+P2*(G1+P1*C0)=G2+P2*G1+P2*P1*G0+P2*P1*P0*C-1

C3=G3+P3*C2= G3+P3*(G2+P2*C1)=G3+P3*G2+P3*P2*G1+P3*P2*P1*G0+P3*P2*P1*P0*C-1

各桁上がり C0 から C3 は、相互関係が無く同時に生成することが可能となる。これにより、各桁の加算を同時に演算することができ、前述の順次桁上げ加算器に比べて高速な処理となる。4 ビットキャリー先読み加算器を図 2.7 に示す。

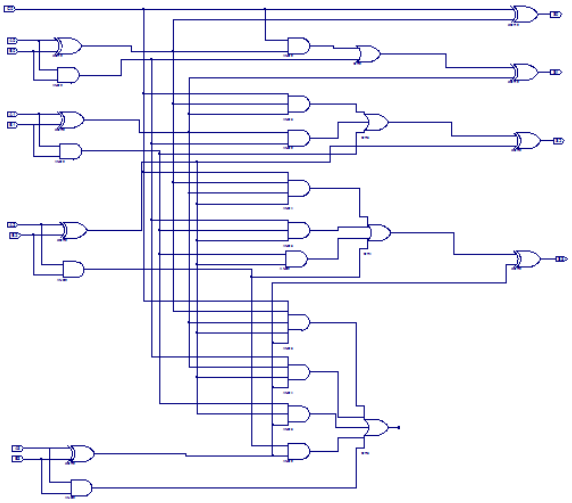

図 2.7　キャリー先読み加算器

32 ビットキャリー先読み加算器は、モジュール化した 4 ビットキャリー先読み加算器を並列に 8 個並べることにより実現される。

2.1.2　マルチプレクサ

2.1.2.1　マルチプレクサの概要

マルチプレクサは、複数の入力信号から一つの信号を選択して出力する回路である。32 ビット 4 入力のマルチプレクサは、2 ビット選択制御信号の組み合わせで入力のうちの一つを選択する。以下では HDL によるマルチプレクサの 2 種類の設計方法、および回路図による設計について説明する。

2.1.2.2　マルチプレクサの HDL 設計

HDL 設計①

2 ビット選択制御信号の 4 通りの組合せ(00、01、10、11)表現で入力信号 in_A、in_B、in_C、in_D のいずれかを選択する。本設計は Verilog HDL が提供する CASE 文を利用することにより、選択制御信号の値(Y)にしたがって、対応する入力信号を選択する。以下に Verilog HDL コードを示す。

```
module multiplex4to1(
    input [31:0] in_A,
    input [31:0] in_B,
    input [31:0] in_C,
```

```
    input [31:0] in_D,
    input [1:0] Y,
    output [31:0] out_S
    );

    function [31:0] MLT;
        input [1:0] Y;

        case (Y)
            2'b00: MLT = in_A;
            2'b01: MLT = in_B;
            2'b10: MLT = in_C;
            2'b11: MLT = in_D;
        endcase
    endfunction

    assign out_S = MLT(Y);
endmodule
```

HDL 設計②

本設計は、Verilog HDL の論理演算を入力データの桁毎に使用するものである。桁毎（i 桁目）の四つの入力信号を in_A[i]、in_B[i]、in_C[i]、in_D[i]、選択制御信号を Y[1:0]、出力を out_S[i]として、論理式で表すと次のようになる。

$$out\_S[i] = (in\_A[i] * \overline{Y[1]} * \overline{Y[0]}) + (in\_B[i] * \overline{Y[1]} * Y[0]) + (in\_C[i] * Y[1] * \overline{Y[0]}) + (in\_D[i] * Y[1] * Y[0])$$

以下に Verilog HDL コードを示す。

```
module multiplex4to1(
    input [31:0] in_A,
    input [31:0] in_B,
    input [31:0] in_C,
    input [31:0] in_D,
    input [1:0] Y,
    output [31:0] out_S
    );
```

```
assign out_S[0] = (in_A[0] & ~Y[1] & ~Y[0]) | (in_B[0] & ~Y[1] & Y[0]) | (in_C[0]
& Y[1] & ~Y[0]) | (in_D[0] & Y[1] & Y[0]);
assign out_S[1] = (in_A[1] & ~Y[1] & ~Y[0]) | (in_B[1] & ~Y[1] & Y[0]) | (in_C[1]
& Y[1] & ~Y[0]) | (in_D[1] & Y[1] & Y[0]);
assign out_S[2] = (in_A[2] & ~Y[1] & ~Y[0]) | (in_B[2] & ~Y[1] & Y[0]) | (in_C[2]
& Y[1] & ~Y[0]) | (in_D[2] & Y[1] & Y[0]);
assign out_S[3] = (in_A[3] & ~Y[1] & ~Y[0]) | (in_B[3] & ~Y[1] & Y[0]) | (in_C[3]
& Y[1] & ~Y[0]) | (in_D[3] & Y[1] & Y[0]);
//  省略
assign out_S[31] = (in_A[31] & ~Y[1] & ~Y[0]) | (in_B[31] & ~Y[1] & Y[0]) | (in_C[31]
& Y[1] & ~Y[0]) | (in_D[31] & Y[1] & Y[0]);
endmodule
```

2.1.2.3　マルチプレクサの回路図設計

まず、1ビットの 4To1(四つの入力信号から一つの信号を選択し出力する)マルチプレクサを作成する。真理値表は表 2.3 のように表わされる。

表 2.3　1 ビット 4To1 マルチプレクサの真理値表

in_A	in_B	in_C	in_D	Y1	Y0	out_S
in_A	in_B	in_C	in_D	0	0	in_A
in_A	in_B	in_C	in_D	0	1	in_B
in_A	in_B	in_C	in_D	1	0	in_C
in_A	in_B	in_C	in_D	1	1	in_D

表 2.3 にしたがって、回路図は図 2.8 のように構成することができる。

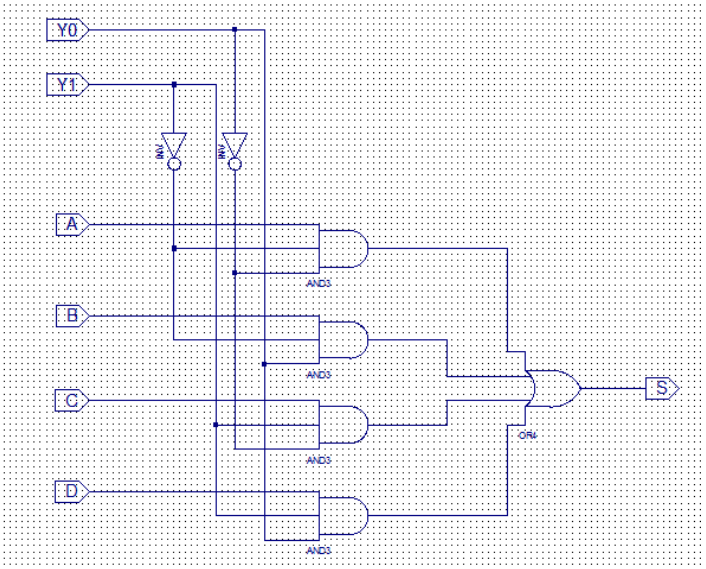

図 2.8　1 ビット 4To1 マルチプレクサ回路図

続いて、32 ビットの 4To1 マルチプレクサは、モジュール化した 1 ビットの 4To1 マルチプレクサを並列に 32 個並べ、それらを図 2.9 のように接続することで実現される。

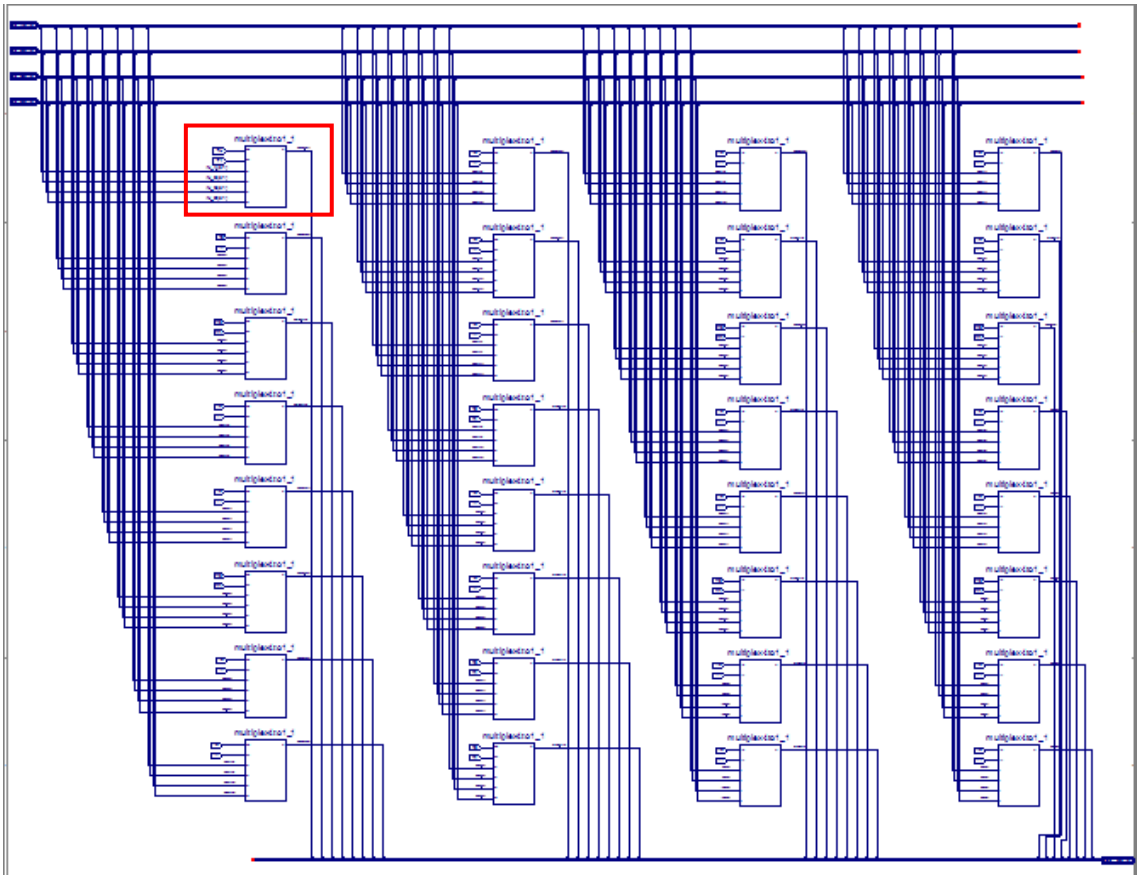

図 2.9　32 ビット 4To1 マルチプレクサ回路図

図 2.9 の赤枠を拡大したものが、図 2.10 である。

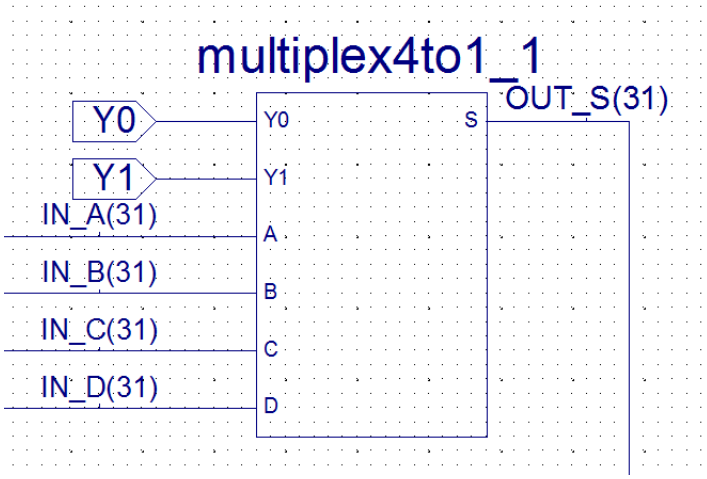

図 2.10　32 ビット 4To1 マルチプレクサ回路図の一部

2.1.2.4　マルチプレクサのシュミレーション

マルチプレクサの入力信号、制御選択信号と出力信号を下記のように設定し、波形図を観察する。

入力信号 in_A：1(00000000000000000000000000000001)

入力信号 in_B：3(00000000000000000000000000000011)

入力信号 in_C：7(00000000000000000000000000000111)

入力信号 in_D：15(00000000000000000000000000001111)

制御選択信号 Y：01

出力信号 out_S：3(00000000000000000000000000000011)

HDL 設計①のシュミレーション

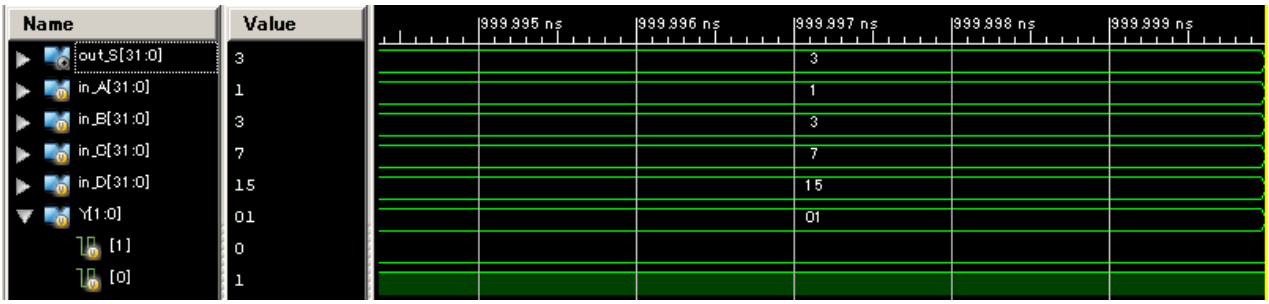

図 2.11　HDL 設計①のシュミレーション

HDL 設計②のシュミレーション

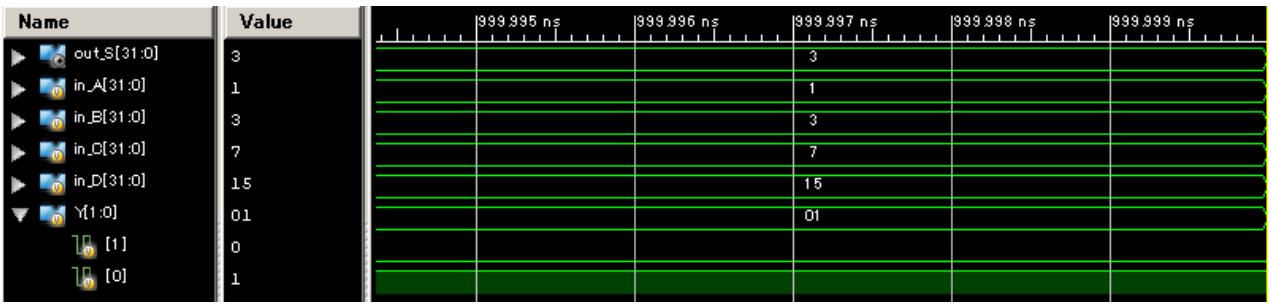

図 2.12　HDL 設計②のシュミレーション

回路図設計のシュミレーション

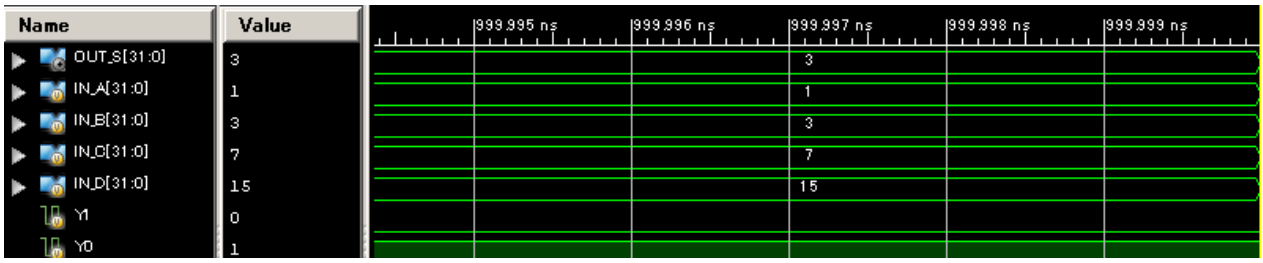

図 2.13　回路図設計のシュミレーション

2.1.3　7 セグメントデコーダ

2.1.3.1　7 セグメントデコーダの概要

7 セグメントデコーダは、図 2.14 のように 7 本の線分（Z0～Z6）から成る図形により、一桁の数字を表示するために使用される回路である。横縦の線分の部分集合を（点灯させる線分として）選択することにより、アラビア数字 0～9 まで表示することができる。以下では HDL による 7 セグメントデコーダの設計方法、および回路図による設計について説明する。

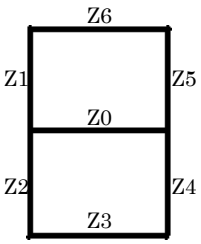

図 2.14　7 セグメントデコーダによる数字の表現

7 セグメントデコーダは 4 つの入力(A、B、C、D)のパターンにより、対応する出力(Z6、

Z5、…Z0)が決まる。表 2.4 が真理値表である。

表 2.4　7 セグメントデコーダの真理値表

A	B	C	D	Z6	Z5	Z4	Z3	Z2	Z1	Z0	表示
0	0	0	0	1	1	1	1	1	1	0	0
0	0	0	1	0	1	1	0	0	0	0	1
0	0	1	0	1	1	0	1	1	0	1	2
0	0	1	1	1	1	1	1	0	0	1	3
0	1	0	0	0	1	1	0	0	1	1	4
0	1	0	1	1	0	1	1	0	1	1	5
0	1	1	0	1	0	1	1	1	1	1	6
0	1	1	1	1	1	1	0	0	0	0	7
1	0	0	0	1	1	1	1	1	1	1	8
1	0	0	1	1	1	1	1	0	1	1	9

2.1.3.2　7 セグメントデコーダの HDL 設計

HDL 設計

本設計は Verilog HDL が提供する IF 文を利用することにより、入力信号 A、B、C、D にしたがって、対応する出力パターンを選択する。以下に Verilog HDL コードを示す。

```
Module decode(
    input A,
    input B,
    input C,
    input D,
    output [6:0] Z
    );
    assign Z[6:0] = decode_fuc(A,B,C,D);

    function[6:0] decode_fuc;
    input A,B,C,D;
        if (A == 0 && B == 0 && C == 0 && D == 0) begin
            decode_fuc[6:0] = 7'b1111110;
        end else if (A == 0 && B == 0 && C == 0 && D == 1) begin
            decode_fuc[6:0] = 7'b0110000;
        end else if (A == 0 && B == 0 && C == 1 && D == 0) begin
```

```
            decode_fuc[6:0] = 7'b1101101;
        end else if (A == 0 && B == 0 && C == 1 && D == 1) begin
            decode_fuc[6:0] = 7'b1111001;
        end else if (A == 0 && B == 1 && C == 0 && D == 0) begin
            decode_fuc[6:0] = 7'b0110011;
        end else if (A == 0 && B == 1 && C == 0 && D == 1) begin
            decode_fuc[6:0] = 7'b1011011;
        end else if (A == 0 && B == 1 && C == 1 && D == 0) begin
            decode_fuc[6:0] = 7'b1011111;
        end else if (A == 0 && B == 1 && C == 1 && D == 1) begin
            decode_fuc[6:0] = 7'b1110000;
        end else if (A == 1 && B == 0 && C == 0 && D == 0) begin
            decode_fuc[6:0] = 7'b1111111;
        end else if (A == 1 && B == 0 && C == 0 && D == 1) begin
            decode_fuc[6:0] = 7'b1111011;
        end else begin
            decode_fuc[6:0] = 7'b0000000;
        end
    endfunction
endmodule
```

2.1.3.3　7セグメントデコーダの回路図設計

回路図設計

真理値表により、各出力の論理式を導き出し、さらにカルノー図を利用し、式を簡略化すると、以下の論理式が得られる。

$Z6=(A+B+C+\overline{D})(\overline{B}+C+D)$;

$Z5=(B+C+\overline{D})(\overline{B}+C+\overline{D})$;

$Z4=B+\overline{C}+D$;

$Z3=(A+B+C+\overline{D})(\overline{B}+C+D)(\overline{B}+\overline{C}+\overline{D})$;

$Z2=(\overline{B}+C)\overline{D}$;

$Z1=(A+B+\overline{D})(\overline{B}+C)(\overline{C}+D)$;

$Z0=(A+B+C)(\overline{B}+\overline{C}+\overline{D})$;

この論理式にしたがい、回路図は図 2.15 のように構成することができる。

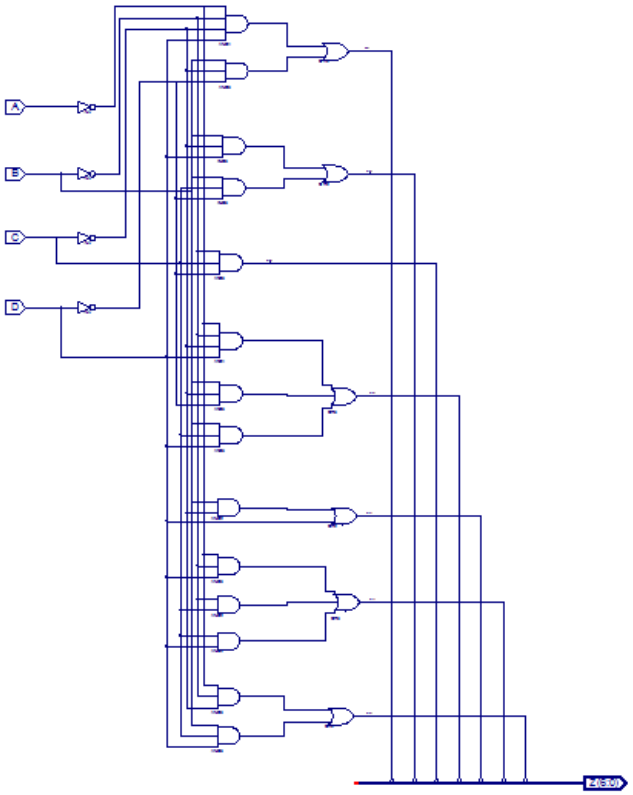

図 2.15　7 セグメントデコーダ回路図

2.1.3.4　7 セグメントデコーダのシュミレーション

7 セグメントデコーダの入力信号、出力信号を下記のように設定し、波形図を観察する。

入力信号 A：0

入力信号 B：1

入力信号 C：0

入力信号 D：0

出力信号 Z：0110011

HDL 設計のシュミレーション

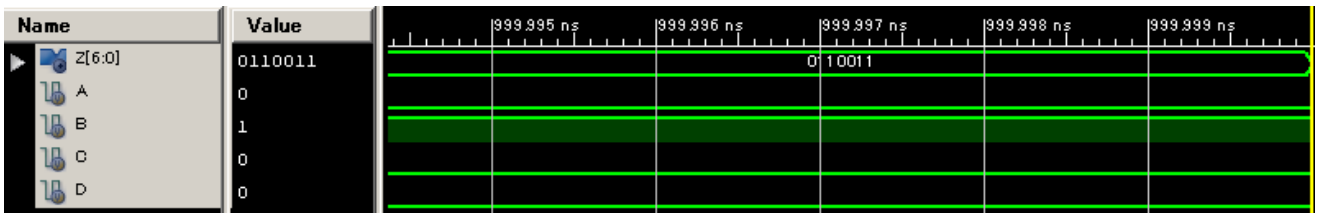

図 2.16　HDL 設計①のシュミレーション

回路図設計のシュミレーション

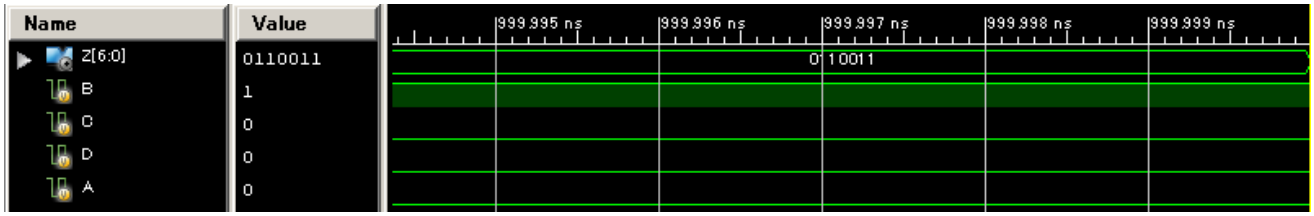

図 2.17　回路図設計のシュミレーション

2.1.4　トライステート

2.1.4.1　トライステートの概要

トライステートは、出力として 0 と 1 以外に、何も出力しない状態(ハイインビーダンス)を持つ論理回路素子である。バス上に複数の出力回路が存在する場合に、出力同士の衝突を回避するために、各出力部分にトライステート素子を配置し、コントロール信号により各出力の有無を制御する。

本研究では 4 つの入力(X1、X2、X3、X4)から、制御信号(T0、T1、T2、T3)にしたがって、表 2.5 のようなパターンの出力を生成する回路を設計した。T0～T3 の中で一つのみが 1 となる場合に対応する入力値（X1～X4）を選択して出力する。その他の場合は全てのトライステート素子が Z(ハイインビーダンス)出力となる。

表 2.5　トライステートを含む回路の真理値表

X1	X2	X3	X4	T0	T1	T2	T3	出力
X1	X2	X3	X4	1	0	0	0	X0
X1	X2	X3	X4	0	1	0	0	X1
X1	X2	X3	X4	0	0	1	0	X2
X1	X2	X3	X4	0	0	0	1	X3

以下では X1～X4 として 8 ビットデータを用いた場合の HDL による設計方法、および回路図による設計について説明する。

2.1.4.2　トライステートを含む回路の HDL 設計

HDL 設計

本設計は Verilog HDL が提供する IF 文を利用することにより、対応する出力パターンを選択する。以下に Verilog HDL コードを示す。

```
module tryState(T0, T1, T2, T3, X0, X1, X2, X3, Y);
    input T0;
    input T1;
    input T2;
    input T3;
    input [7:0] X0;
    input [7:0] X1;
    input [7:0] X2;
    input [7:0] X3;
    output [7:0] Y;

    assign Y = STATE(T0,T1,T2,T3);
    function[7:0] STATE;
        input T0,T1,T2,T3;
        if (~T3 & ~T2 & ~T1 & T0) begin
            STATE = X0;
        end else if (~T3 & ~T2 & T1 & ~T0) begin
            STATE = X1;
        end else if (~T3 & T2 & ~T1 & ~T0) begin
            STATE = X2;
        end else if (T3 & ~T2 & ~T1 & ~T0) begin
            STATE = X3;
        end else begin
            STATE = 8'bZZZZZZZZ;
        end
    endfunction
endmodule
```

2.1.4.3　トライステートを含む回路の回路図設計

回路図設計

真理値表にしたがって、8 ビットトライステートバッファ (BUFT8) を利用し、それらを図 2.18 のように接続することで実現される。

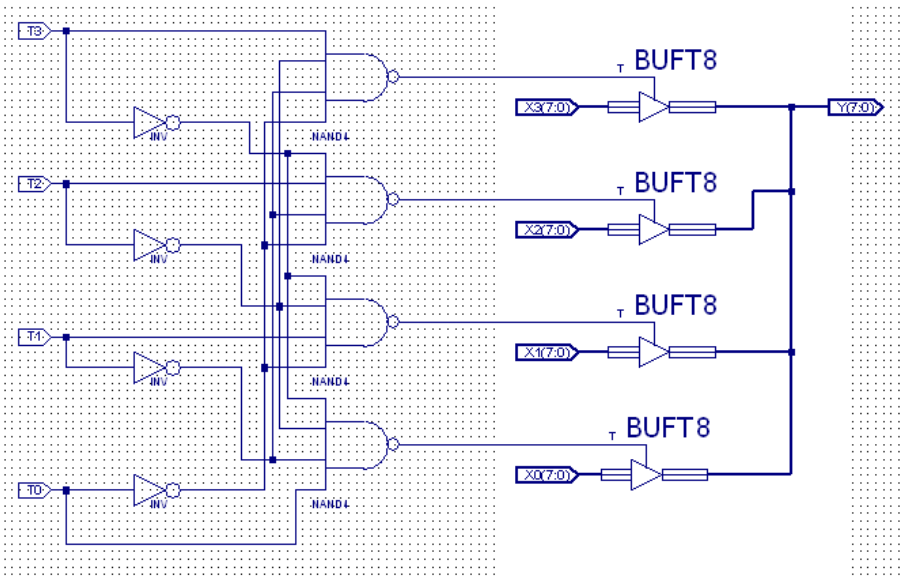

図 2.18　4 入力 8 ビットのトライステート

2.1.4.4　トライステートを含む回路のシュミレーション

トライステートの入力信号、出力信号を下記のように 2 パターン(出力がハイインピーダンスと非ハイインピーダンス)を設定し、波形図を観察する。

パターン①

入力信号：T0=1, T1=0, T2=0, T3=0

出力信号：X0

パターン②

入力信号：T0=0, T1=1, T2=0, T3=1

出力信号：Z(ハイインピーダンス)

HDL 設計のシュミレーション

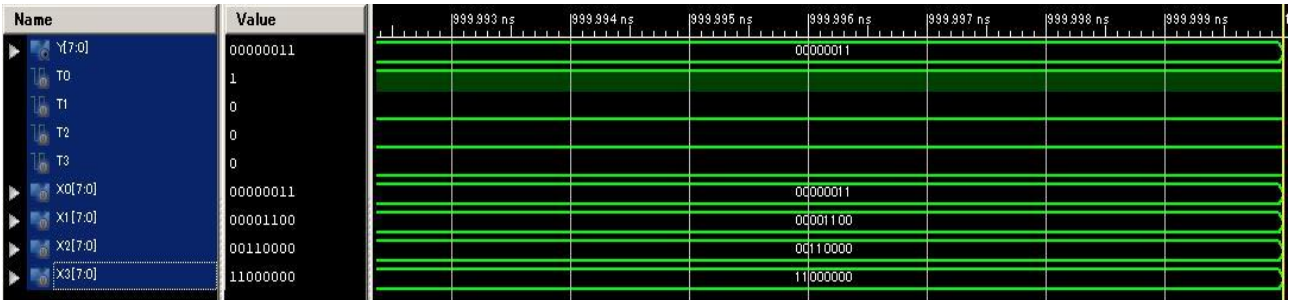

図 2.19　HDL 設計のシュミレーション(パターン①)

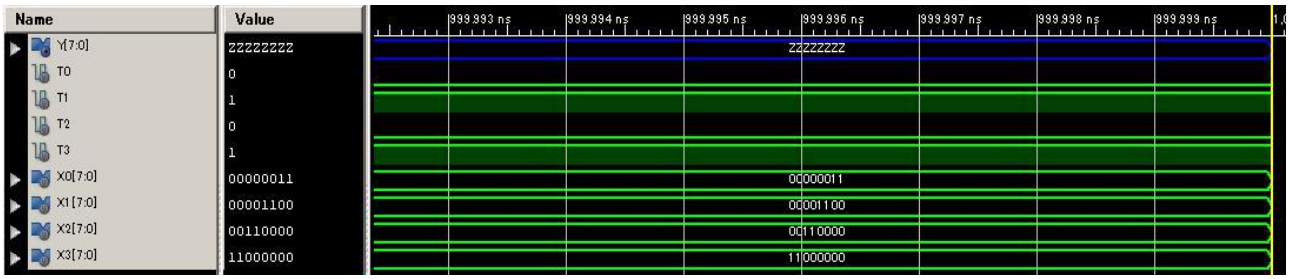

図 2.20　HDL 設計のシュミレーション(パターン②)

回路図のシュミレーション

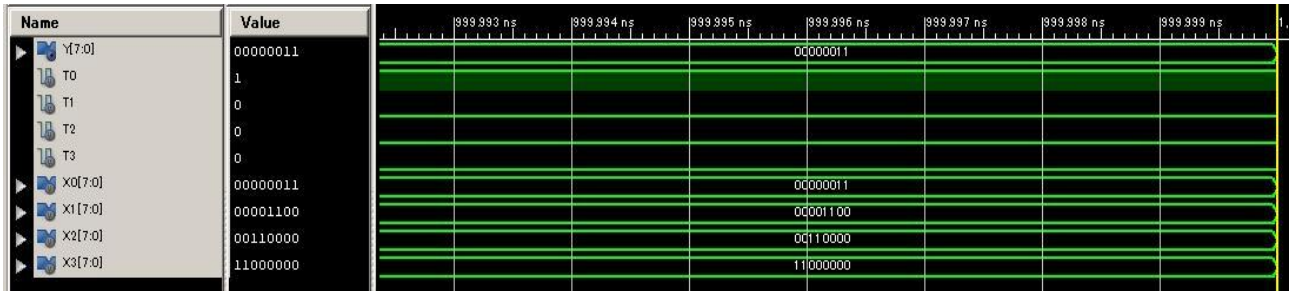

図 2.21　回路図設計のシュミレーション(パターン①)

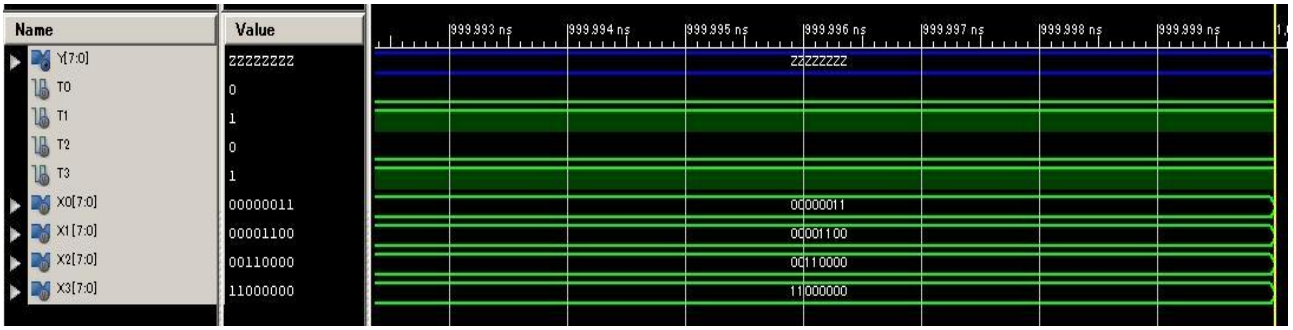

図 2.22　回路図設計のシュミレーション(パターン②)

2.2　順序回路の設計

順序回路とは、現在の状態と現在の入力に依存して、次の状態や出力を決定する回路である。順序回路には同期式と非同期式の 2 種類が存在する。

同期式回路：全ての内部ラッチが同じクロックのタイミングで動作する回路である。

非同期式回路：内部ラッチ毎に入力信号にしたがって非同期に（クロック信号とは無関係に）動作する回路である。

本研究では順序回路のうち同期式回路を対象とする。多くの同期式順序回路ではフリップフロップと呼ばれる 1 ビットのデータ記憶機能を備えている素子が使われる。フリップフロップにはいくつか種類があるが、その中で最も単純な論理を実現するものが D 型フ

リップフロップ（D-FF）である。クロック入力の立ち上がり時に入力値を状態として記憶し、次のクロックの立ち上がりタイミングまでその値を保持する。出力値は状態の値となる。真理値表を表 2.6 に示す。

表 2.6　D-FF の真理値表

入力	CLOCK	出力
0	↑	0
1	↑	1
任意	立ち上がり以外	保持（不変）

2.2.1　シフトレジスタ

2.2.1.1　シフトレジスタの概要

シフトレジスタ回路は、クロック入力に同期してレジスタ内でデータを左右に移動させる回路である。ここでは、D-FF で構成した同期式 32 ビットシフトレジスタ（左シフト）の HDL による２種類の設計方法、および回路図による設計について説明する。

2.2.1.2　シフトレジスタの HDL 設計

HDL 設計①

クロックが立ち上がるたびに、1 ビットの入力値を最下位桁にセットし、他の桁を 1 ビットずつ上位に移動させる。以下に Verilog HDL コードを示す。

```
module sift(
    input clk,
    input rst,
    input in,
    output reg [31:0] out = 32'h0
    );

    always @(posedge clk) begin
        if (rst == 1'b1)
            out <= 32'h0;
        else  begin
            out[0] <= in;
            out[31:1] <= out[30:0];
```

```
        end
    end
endmodule
```

HDL 設計②

D-FF モジュールを利用したシフトレジスタを示す。クロックが立ち上がるたびに、1 ビットの入力値を最下位の D-FF にセットし、他の D-FF に 1 桁下の D-FF の値をセットする。32 個の D-FF を接続することにより、32 ビットシフトレジスタが実現可能である。以下に Verilog HDL コードを示す。

D-FF モジュール

```
module D_FF(
    input D,
    input CLK,
    input RST,
    output reg Q = 1'b0
    );

    always @(posedge CLK) begin
        if(RST) begin
            Q <= 1'b0;
        end else begin
            Q <= D;
        end
    end
endmodule
```

シフトレジスタモジュール

```
Module sift(
    input INPUT,
    input CLK,
    input RST,
    output [31:0] OUTPUT
    );

    D_FF Q0(.D(INPUT),.CLK(CLK),.RST(RST),.Q(OUTPUT[0]));
```

```
    D_FF Q1(.D(OUTPUT[0]),.CLK(CLK),.RST(RST),.Q(OUTPUT[1]));
    D_FF Q2(.D(OUTPUT[1]),.CLK(CLK),.RST(RST),.Q(OUTPUT[2]));
    //  省略
    D_FF Q31(.D(OUTPUT[30]),.CLK(CLK),.RST(RST),.Q(OUTPUT[31]));
Endmodule
```

2.2.1.3　シフトレジスタの回路図設計

回路図設計

HDL 設計②と同様に、32 個の D-FF を利用し、それらを図 2.23 のように組み合わせることで実現される。

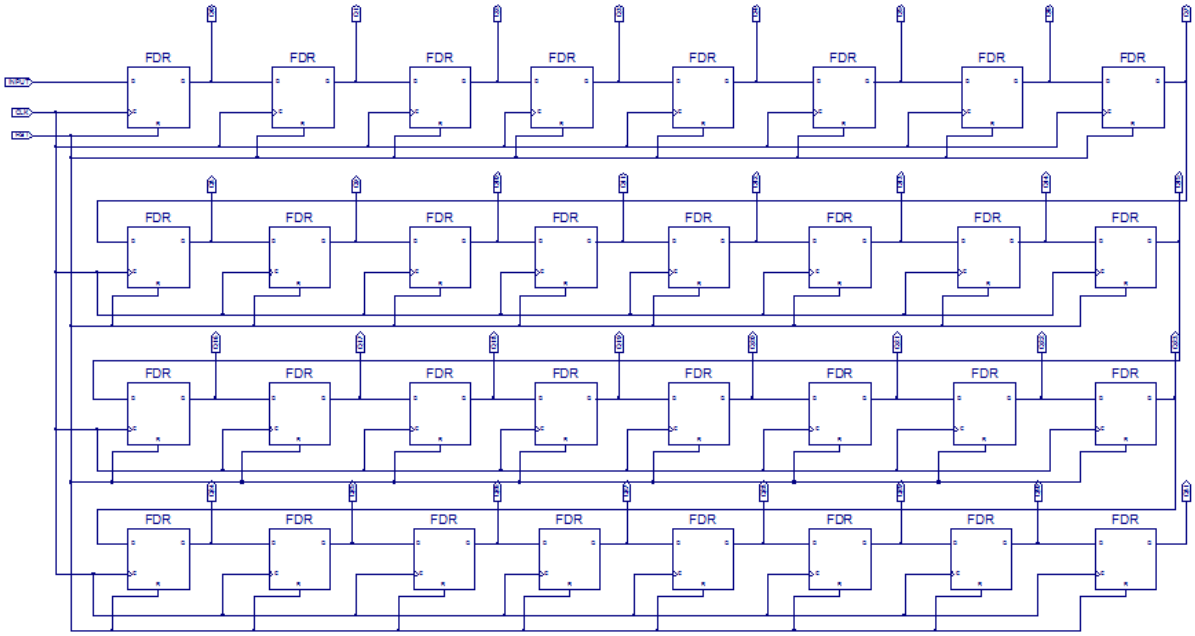

図 2.23　32 ビットのシフトレジスタ

2.2.1.4　シフトレジスタのシュミレーション

シフトレジスタの入力信号を下記の様に設定し、波形図を観察する。

入力信号：

0→0→1→1→0→0→1→1→0→0→1→1…

出力信号：

0000000000 → 0000000000 → 0000000001 → 0000000011 → 0000000110 → 0000001100 →
0000011001→0000110011→0001100110→0011001100→0110011001→1100110011…

HDL 設計①のシュミレーション

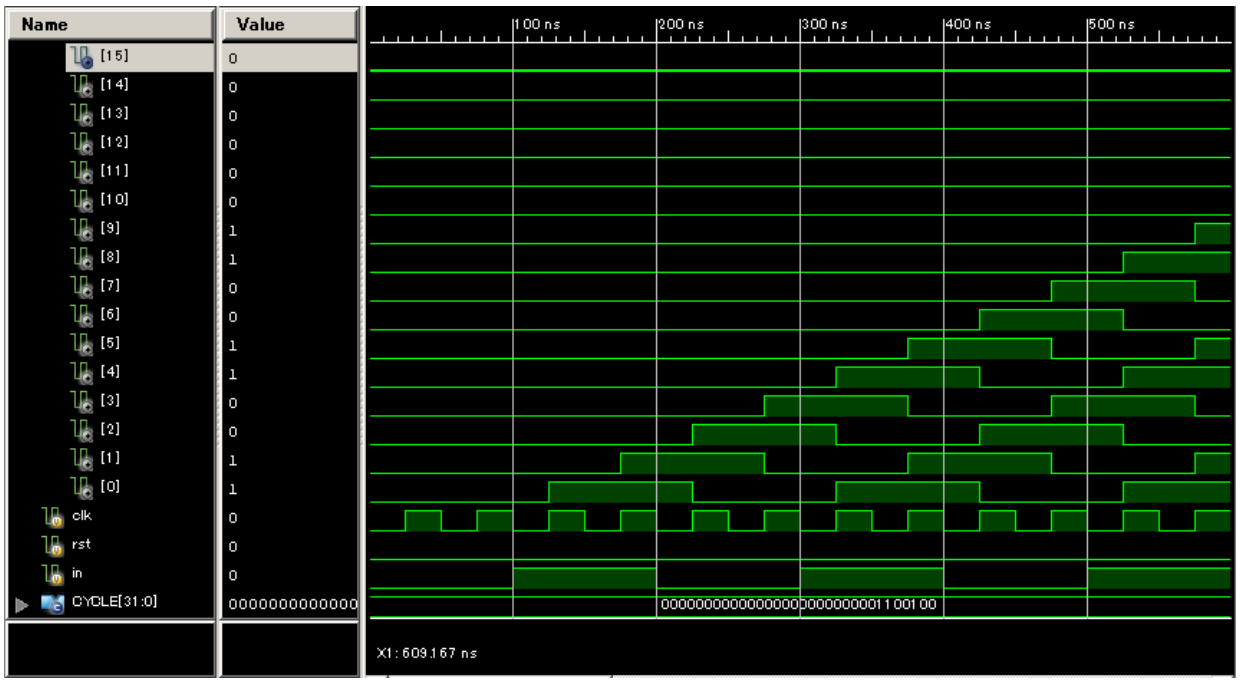

図 2.24　HDL 設計①のシュミレーション

HDL 設計②のシュミレーション

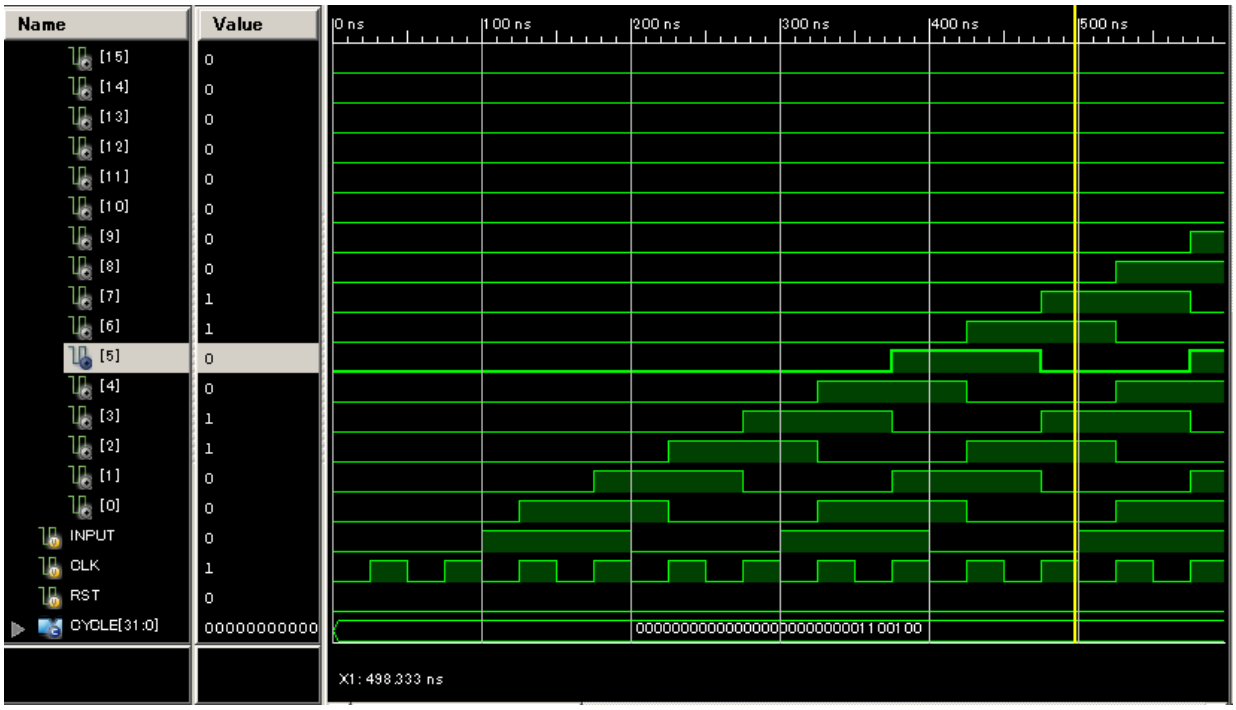

図 2.25　HDL 設計②のシュミレーション

回路図設計のシュミレーション

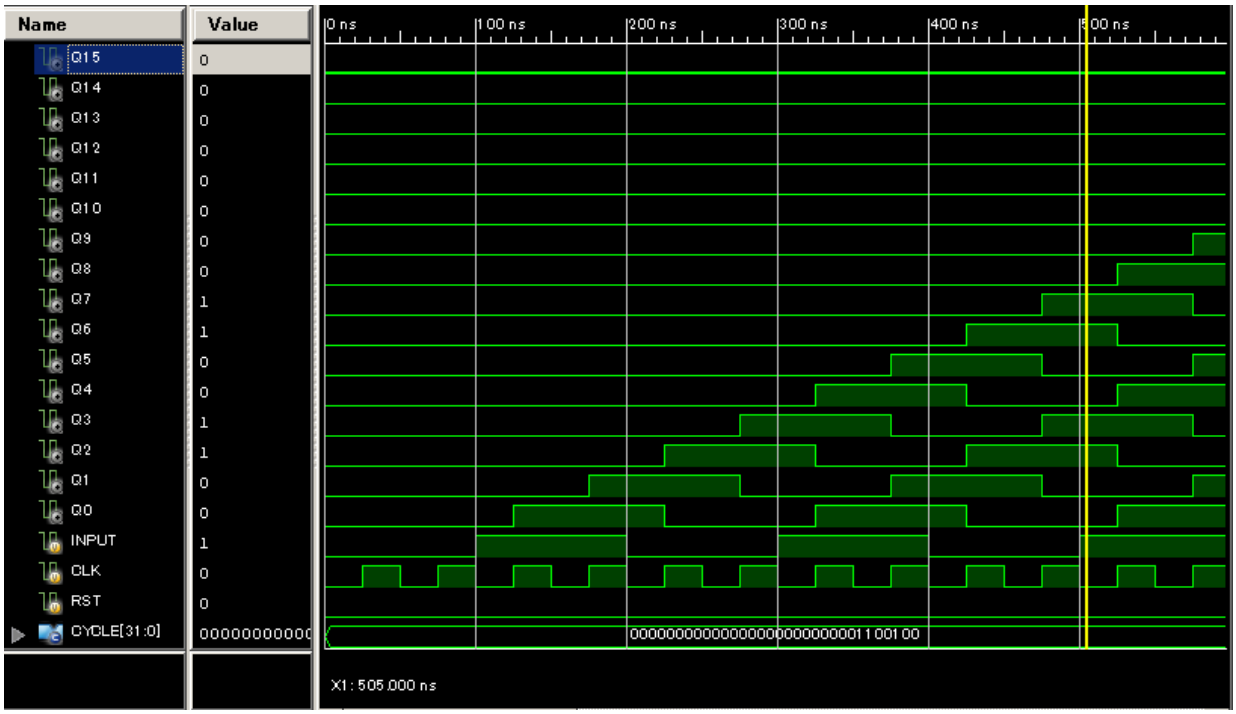

図 2.26　回路図設計のシュミレーション

2.2.2　カウンタ

2.2.2.1　カウンタの概要

カウンタ回路は、数え上げを行う回路である。同期式 32 ビットカウンタの HDL による２種類の設計方法、および回路図による設計について説明する。

2.2.2.2　カウンタの HDL 設計

HDL 設計①

Verilog HDL が提供する“+” 演算子を利用する方法である。クロックの立ち上がり時、前回までの計数結果(COUNT)に対して+演算子で 1 との加算を行い、COUNT を更新する。以下に Verilog HDL コードを示す。

```
module counter(
    input RESET,
    input CLK,
    output reg [31:0] COUNT = 32'h0
    );
```

```
    always @(posedge CLK) begin
        if (RESET == 1'b1)
            COUNT <= 32'h0;
        else begin
            COUNT <= COUNT + 32'h1;
        end
    end
endmodule
```

HDL 設計②

半加算器モジュールと D-FF モジュールを Verilog HDL で記述し、上位モジュールによってそれらを接続することにより、カウンタが実現可能である。回路の構成は回路図設計と同様である。以下に Verilog HDL コードを示す。HALF が半加算器モジュール、DFF が D-FF モジュール、counter が上位モジュールである。

半加算器モジュール

```
module HALF(
    input A,
    input B,
    output S,
    output Cout
    );

    assign {Cout, S} = A + B;
Endmodule
```

D-FF モジュール

```
module DFF(
    input D,
    input CLK,
    input RST,
    output reg Q = 1'b0
    );
   always @(posedge CLK) begin
       if(RST == 1)
           Q <= 1'b0;
```

```
        else begin
            Q <= D;
        end
    end
endmodule
```

カウンタモジュール

```
module counter(
    input A,
    input CLK,
    input RST,
    output [31:0] OUTPUT
    );

    wire S0,C0,S1,C1,S2,C2,S3,C3,S4,C4,S5,C5,S6,C6,S7,C7,S8,C8,S9,C9,
         S10,C10,S11,C11,S12,C12,S13,C13,S14,C14,S15,C15,S16,C16,S17,
         C17,S18,C18,S19,C19,S20,C20,S21,C21,S22,C22,S23,C23,S24,C24,
         S25,C25,S26,C26,S27,C27,S28,C28,S29,C29,S30,C30,S31,C31;

    HALF H0(.A(A),.B(OUTPUT[0]),.S(S0),.Cout(C0));
    DFF Q0(.D(S0),.CLK(CLK),.RST(RST),.Q(OUTPUT[0]));

    HALF H1(.A(OUTPUT[1]),.B(C0),.S(S1),.Cout(C1));
    DFF Q1(.D(S1),.CLK(CLK),.RST(RST),.Q(OUTPUT[1]));

    HALF H2(.A(OUTPUT[2]),.B(C1),.S(S2),.Cout(C2));
    DFF Q2(.D(S2),.CLK(CLK),.RST(RST),.Q(OUTPUT[2]));
    //  省略
    HALF H31(.A(OUTPUT[31]),.B(C30),.S(S31),.Cout(C31));
    DFF Q31(.D(S31),.CLK(CLK),.RST(RST),.Q(OUTPUT[31]));
endmodule
```

2.2.2.3　カウンタの回路図設計

回路図設計

32 個の半加算器(図 2.27)と 32 個の D-FF を利用し、それらを図 2.28 のように組み合

わせることで実現される。

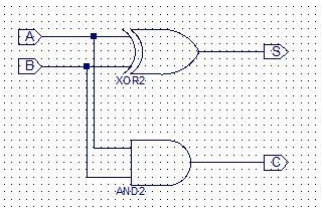

図 2.27　半加算器回路図

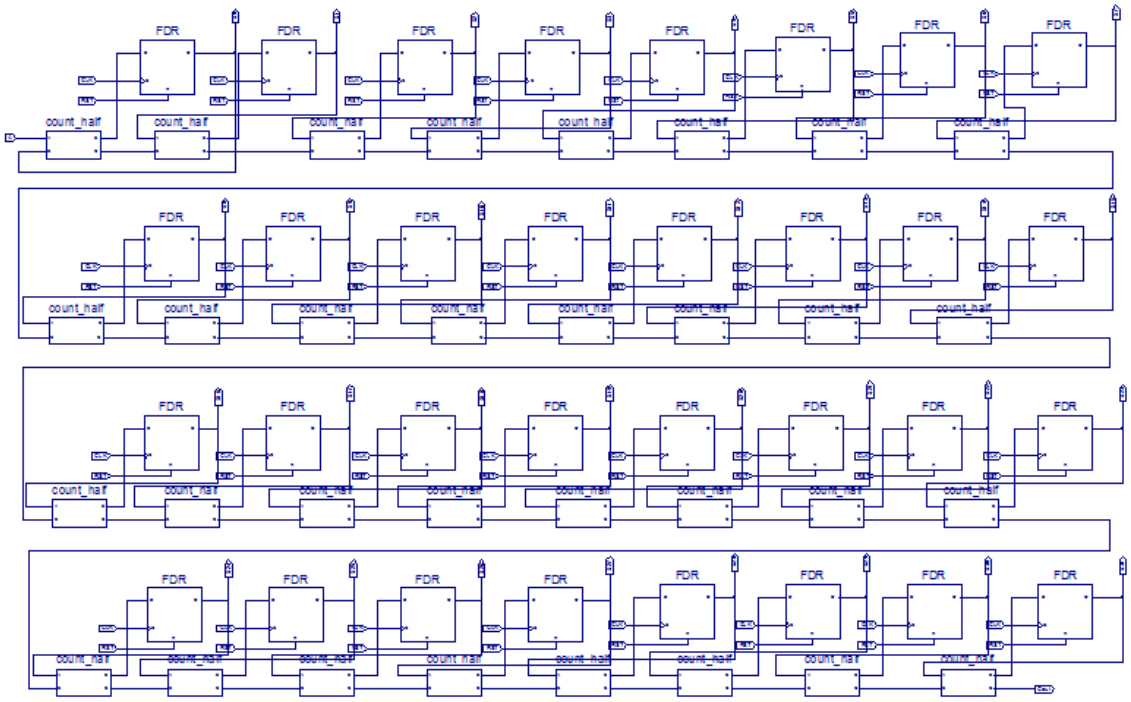

図 2.28　カウンタ回路図

2.2.2.4　カウンタのシュミレーション

カウンタの出力信号が下記の様なパターンとなる波形図を観察する。

出力信号：

0000→0001→0010→0011→0100→0101→0110→0111→1000→1001→1010→1011→1100…

HDL 設計①のシュミレーション

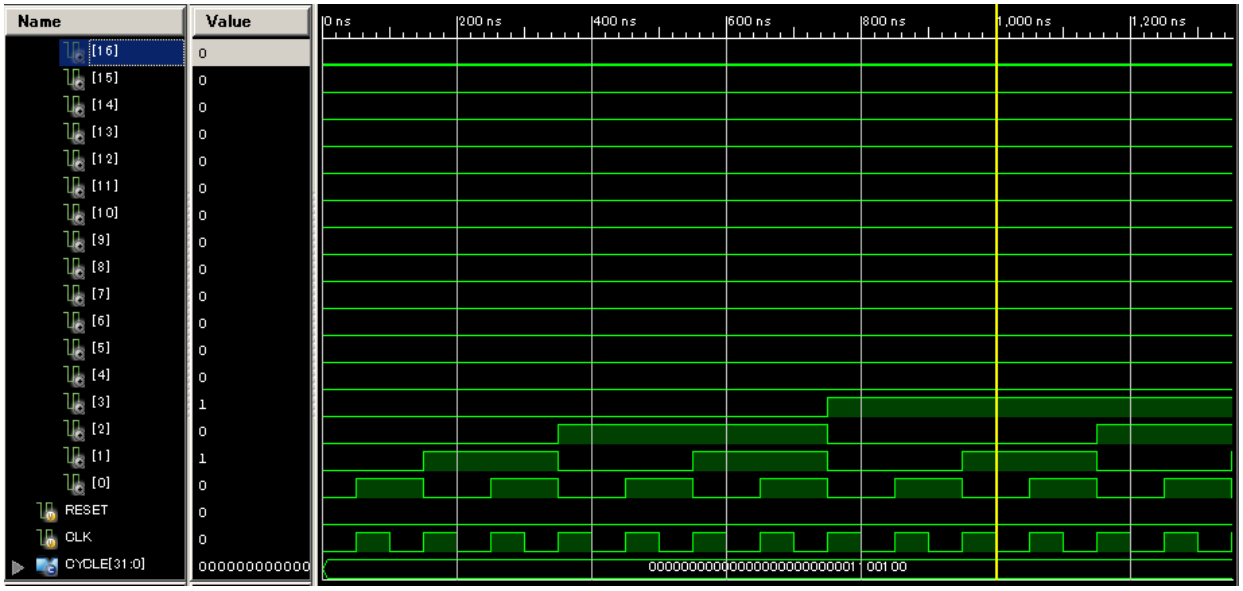

図 2.29　HDL 設計①のシュミレーション

HDL 設計方式②のシュミレーション

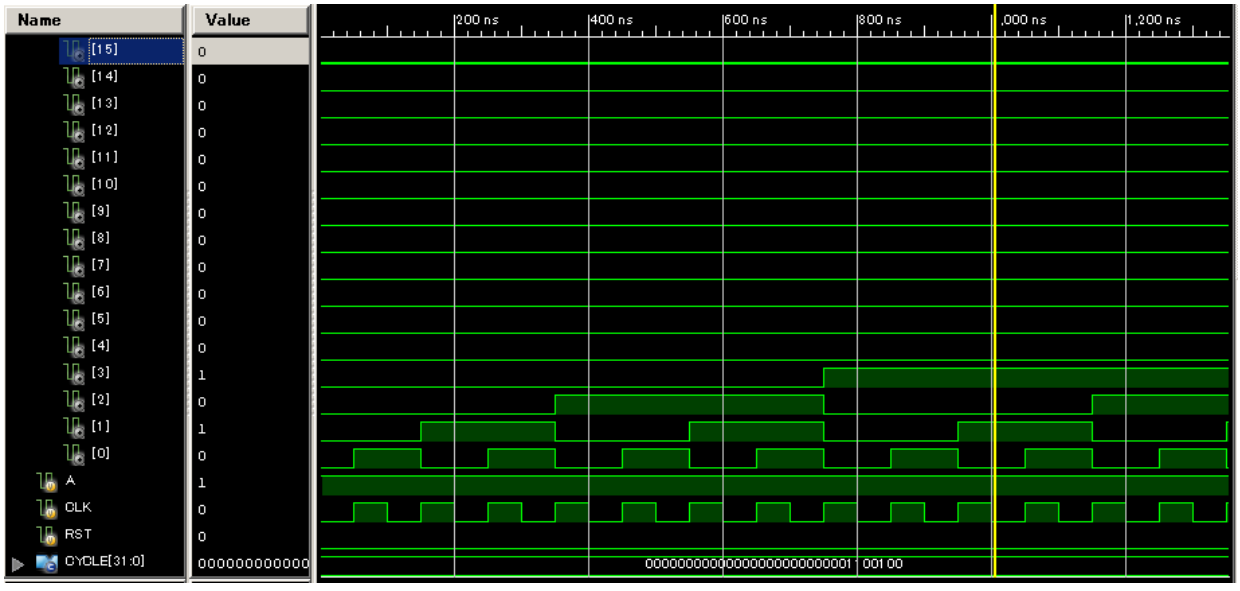

図 2.30　HDL 設計②のシュミレーション

回路図設計のシュミレーション

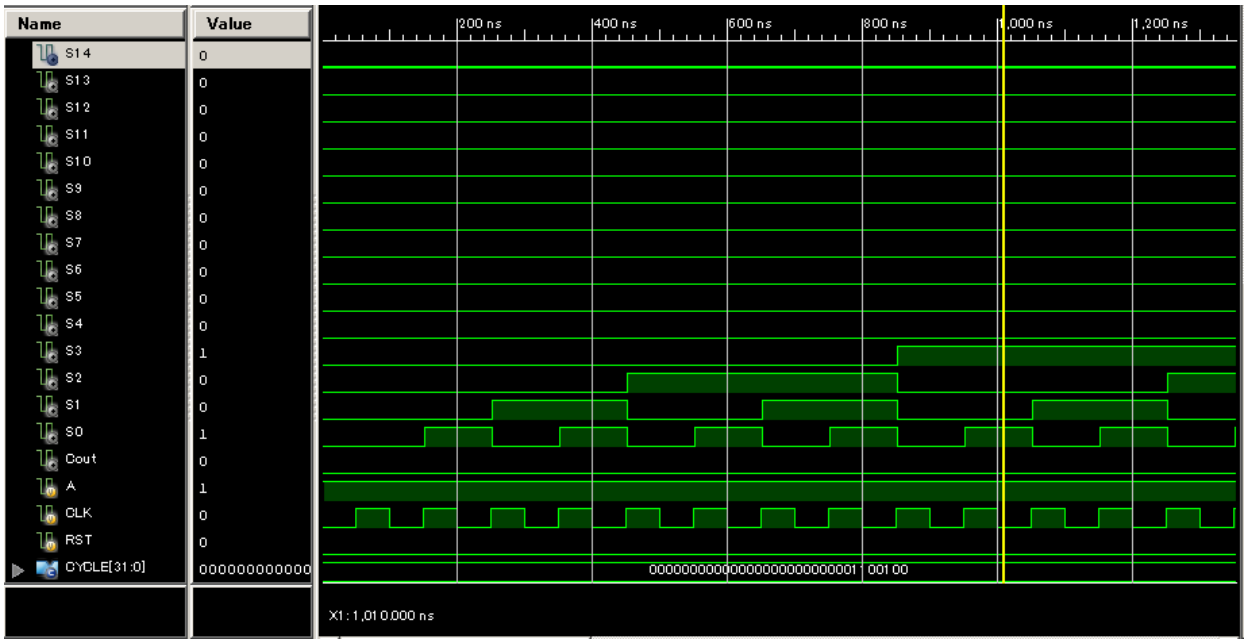

図 2.31　回路図設計のシュミレーション

2.2.3　ステートマシン

2.2.3.1　ステートマシンの概要

ステートマシンは、入力信号と現在の状態により出力信号と次の状態が決まる回路である。ここでは後述する CPU 回路の部分機能である MAIN_CTRL 回路を対象とする。

クロック信号(CLK)が立ち上がるたびに、リセット(RST)信号の値により、状態を表す内部信号 S[3:0]が”0000”に初期化されるか、あるいは次の状態を表す内部信号 NS[3:0]の値になるかが、PROCESS モジュールによって決定する。S の値にしたがって、各モジュール(ALUOP、ALUSRCA、ALUSRCB など)を通じて、同名の各出力信号(ALUOP、ALUSRCA、ALUSRCB など)が生成される。内部信号 NS は、S と入力信号 OP から NS_GEN モジュールによって生成される。NS は次のクロックの立ち上がり時に PROCESS モジュールへの入力となる。本回路のモジュール間接続を図 2.32 に示す。

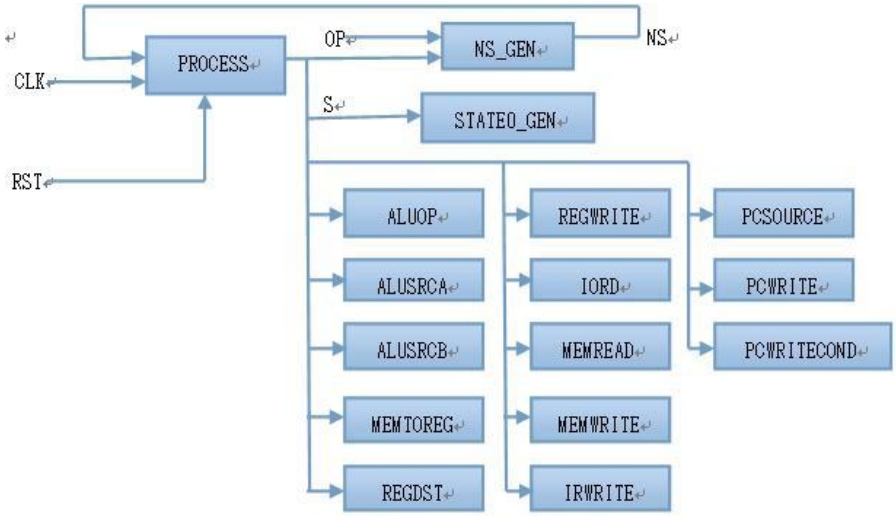

図 2.32　ステートマシンイメージ図

2.2.3.2　ステートマシンの HDL 設計

図 2.32 にしたがって、各モジュールを HDL で作成し、上位モジュールで結合することより実現される。Verilog ソースは付録に添付する。

2.2.3.3　ステートマシンの回路図設計

各モジュールを回路図で作成し、それらを図 2.32 にしたがって接続することで実現される。上位階層回路図を図 2.33 に示す。(各モジュール内の回路図は紙面の都合上省略する。)

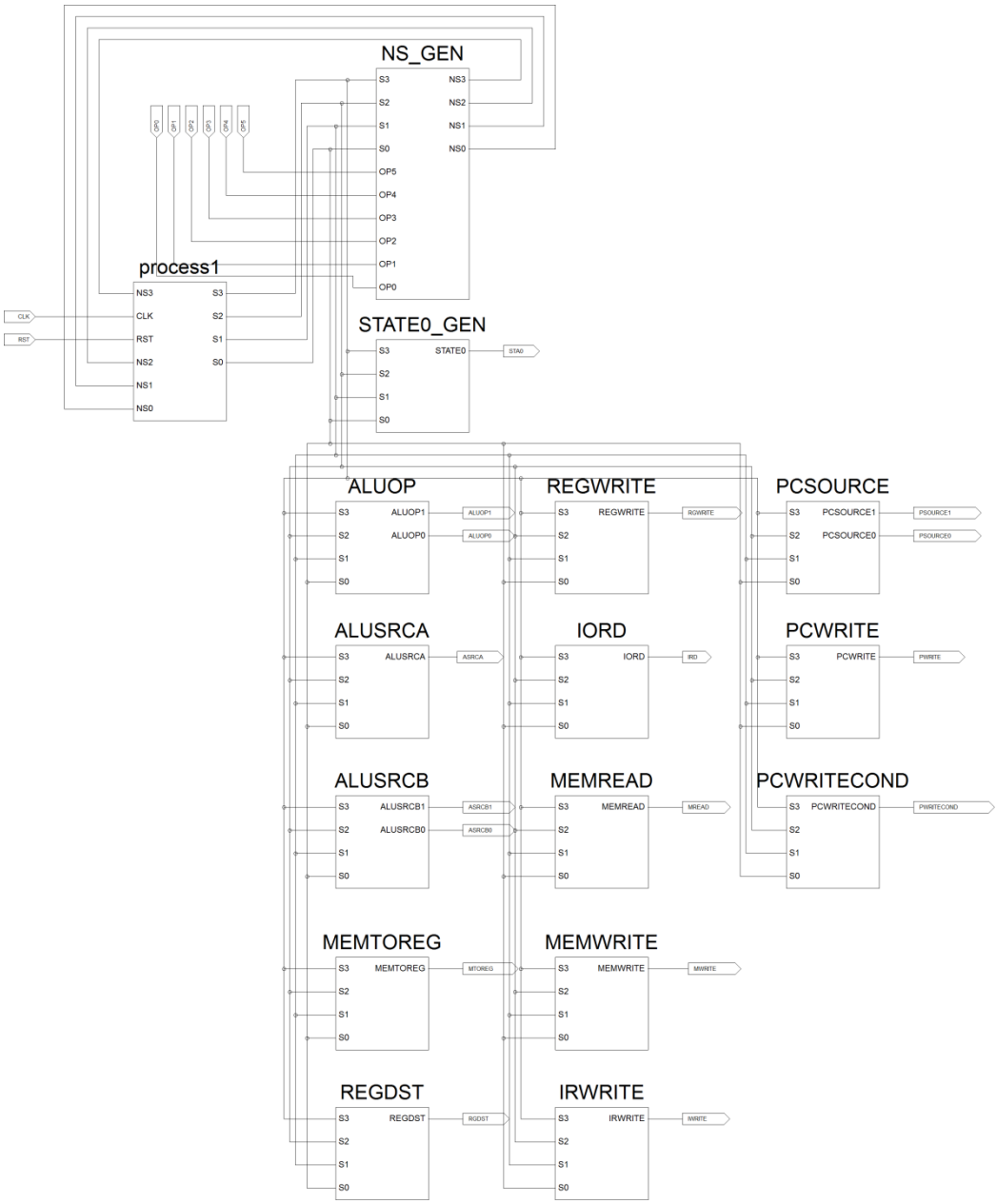

図 2.33　ステートマシンの上位階層回路図

2.2.3.4　ステートマシンのシュミレーション

ステートマシンの初期入力信号を下記のように設定し、波形図を観察する。

初期入力信号

CLK:0

RST:1

OP:100011

CLK の周期：50ns

RST 変化

100ns 経過: RST 1→0

250ns 経過: RST 0→1

150ns 経過: RST 1→0

HDL 設計のシュミレーション

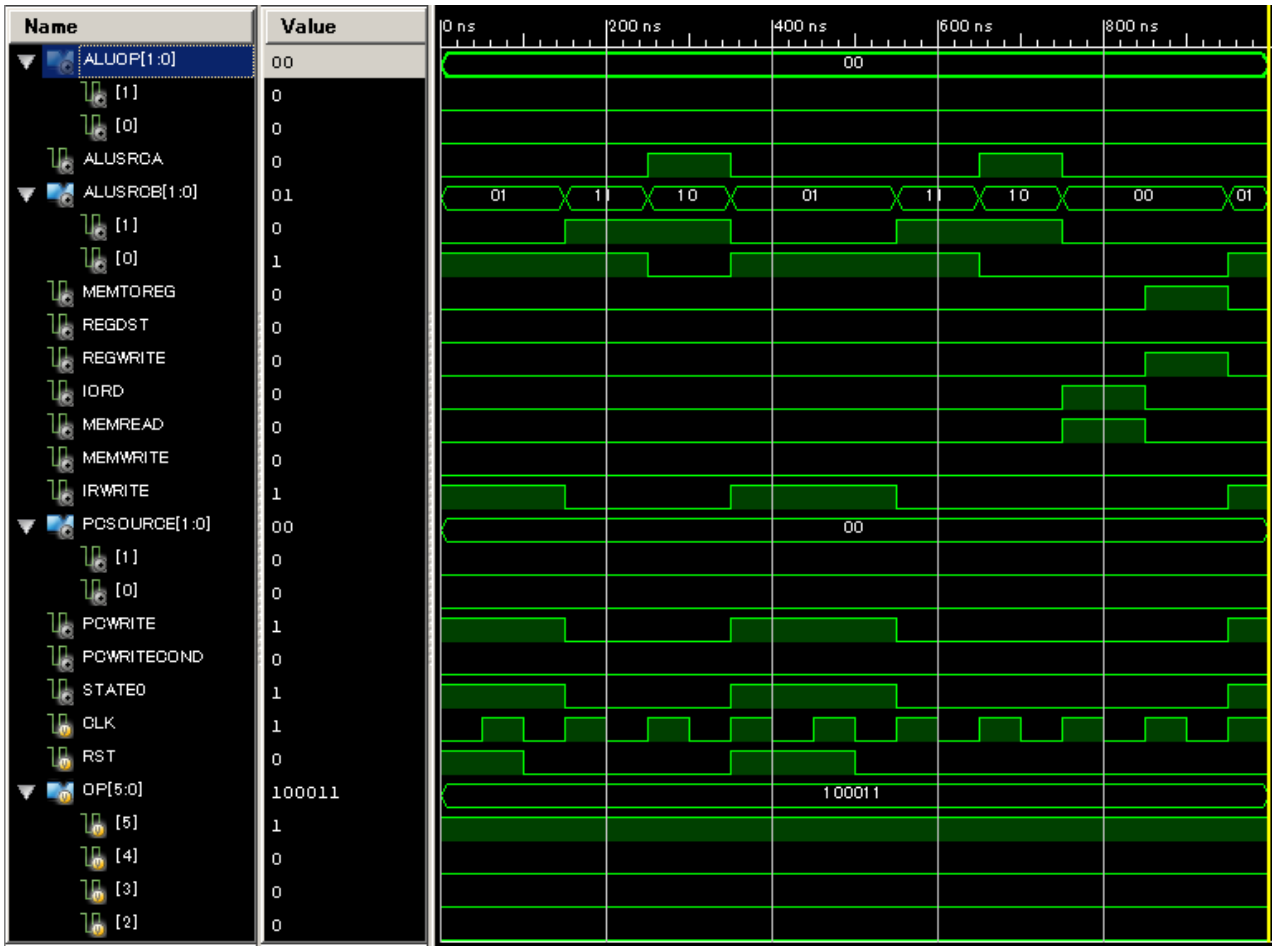

図 2.34　HDL 設計のシュミレーション

回路図設計のシュミレーション

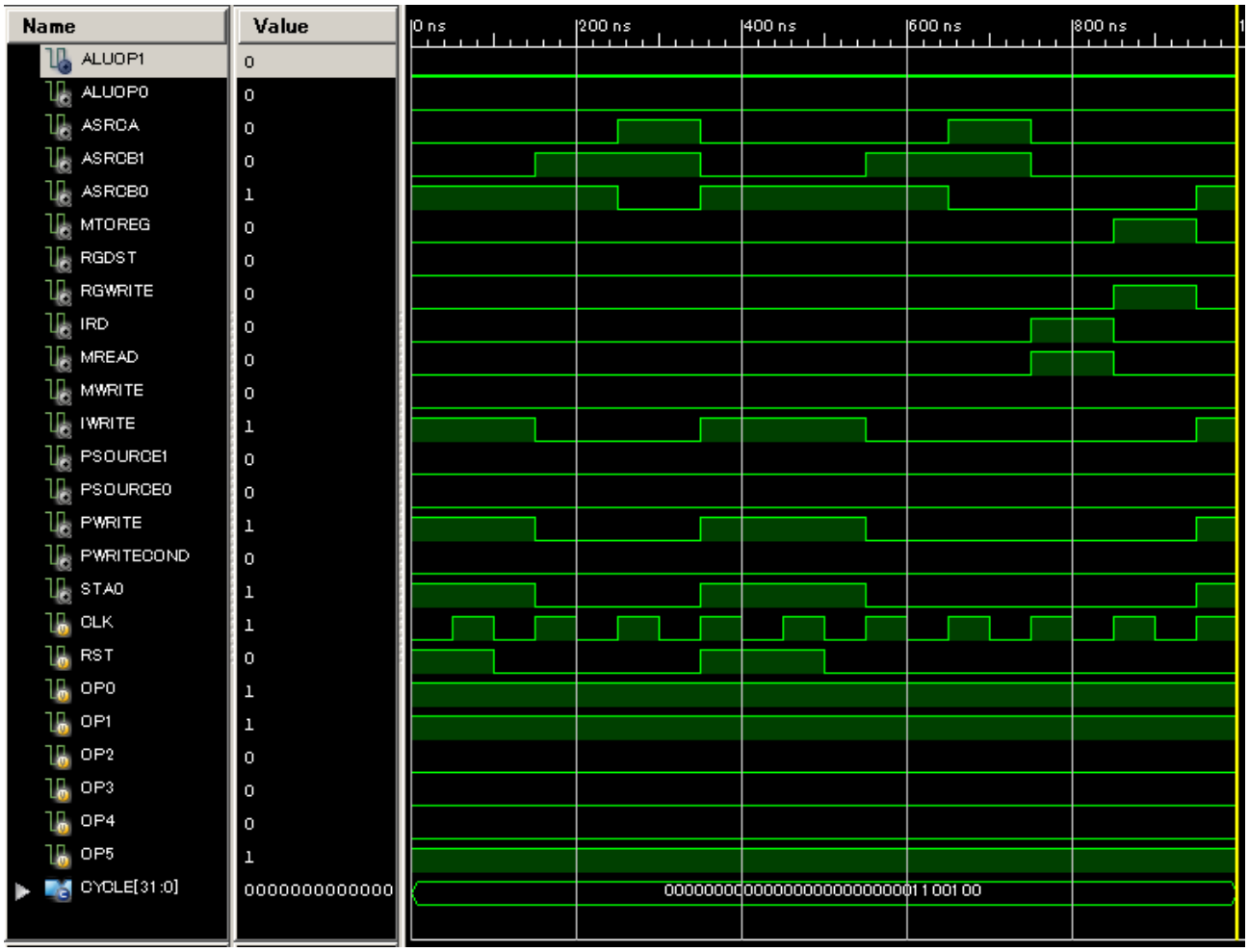

図 2.35　回路図設計のシュミレーション

2.3　総合回路の設計

2.3.1　CPU 回路

組み合わせ回路、順序回路を含め、様々な論理を含む CPU 回路を総合回路の対象とする。本回路のイメージを図 2.36 に示す。

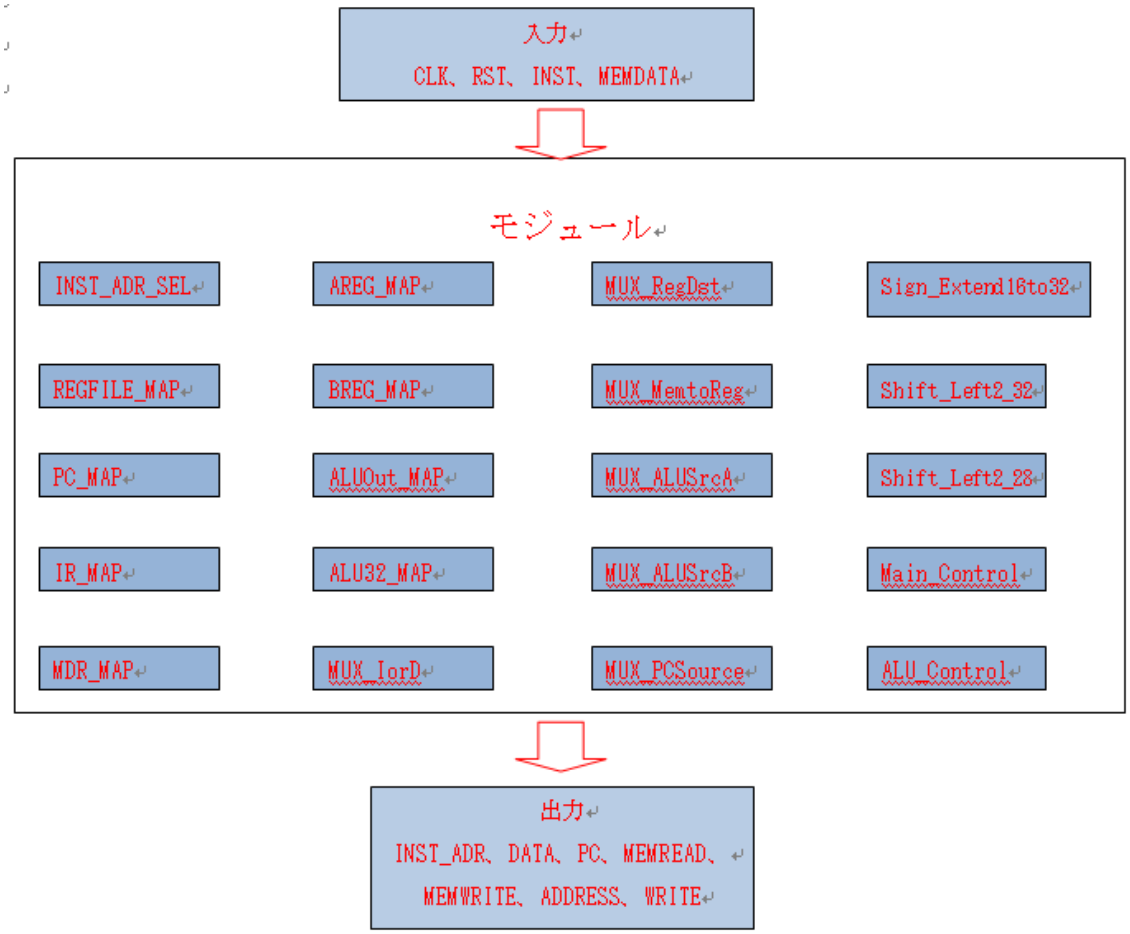

図 2.36　CPU イメージ図

2.3.1.1　CPU 回路の概要

本研究で対象とする CPU 回路は MIPS 命令セット[4]のサブセットを実行する 32 ビットアーキテクチャ構成である。(前節のステートマシン回路はこの CPU の一部である。) 各モジュールの機能は以下の通りである。

①モジュール REGFILE_MAP、PC_MAP、IR_MAP、MDR_MAP、AREG_MAP、BREG_MAP、ALUOut_MAP：一時的なデータ、命令、アドレスを保存する一時レジスタの役割。

②モジュール ALU32_MAP：加算、論理和、論理積などの演算を行う算術論理演算ユニット。

③モジュール MUX_IorD、MUX_RegDst、MUX_MemtoReg、MUX_ALUSrcA、MUX_ALUSrcB、MUX_PCSource：二つ以上の入力から一つを選択するマルチプレクサロジック。

④モジュール Sign_Extend16to32、Shift_Left2_32、Shift_Left2_28：符号拡張あるいはシフト演算を行うロジック。

⑤モジュール Main_Control：CPU の制御信号を生成するロジック。

⑥モジュール ALU_Control：算術論理演算ユニット内の演算器を選択する制御信号を生成す

るロジック。

2.3.1.2　CPU 回路の HDL 設計

各モジュールを HDL で作成し、上位モジュールで結合することより実現される。Verilog ソースを付録に添付する。

2.3.1.3　CPU 回路の回路図設計

各モジュールを回路図で作成し、それらを上位回路図として接続することで実現される。（図 2.37 に上位回路図の主要部分を示す。各モジュール内の回路図は紙面の都合上省略する。）

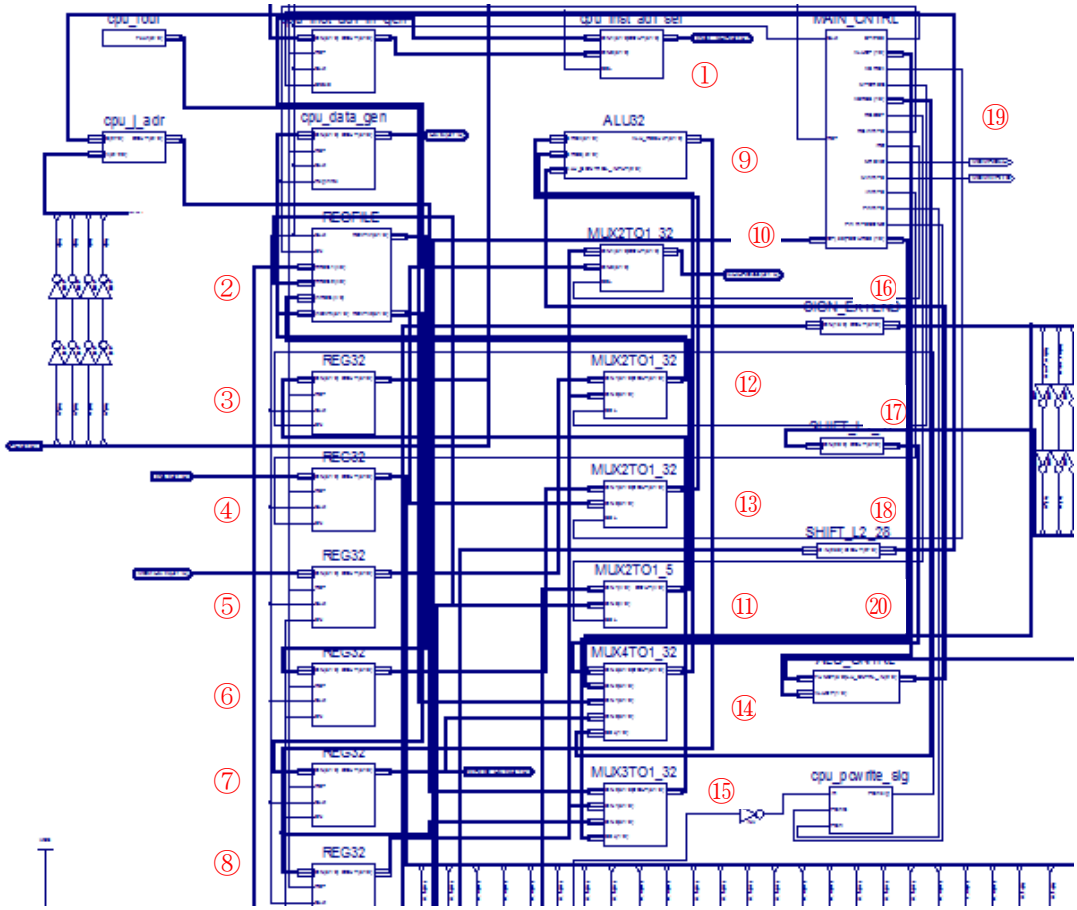

図 2.37　CPU の上位階層回路図の主要部分

①INST_ADR_SEL②REGFILE_MAP③PC_MAP④IR_MAP⑤MDR_MAP⑥AREG_MAP⑦BREG_MAP
⑧ALUOut_MAP⑨ALU32_MAP⑩MUX_IorD⑪MUX_RegDst⑫MUX_MemtoReg⑬MUX_ALUSrcA
⑭MUX_ALUSrcB⑮MUX_PCSource⑯Sign_Extend16to32⑰Shift_Left2_32
⑱Shift_Left2_28⑲Main_Control⑳ALU_Control

2.3.1.4　CPU 回路の HDL、回路図を混在した設計

2.3.1.2 の HDL 設計をベースにして、MAIN_CTRL モジュールを 2.3.1.3 の回路図設計のものに置き換えることにより、HDL と回路図の両設計が混在する形で回路を構成する。

2.3.1.5　CPU 回路のシュミレーション

（シュミレーションは紙面の都合上省略する。）

第 3 章　評価と考察

本章では、２章で扱った HDL 設計と回路図入力の両設計方式で作成したそれぞれの回路に対して、Spartan-3E Starter Kit Board[7]をターゲットとしたインプリメントを行い、回路規模および最大遅延を比較することにより、設計方式による生成回路の傾向を評価する。なお、具体的な指標は以下の通りである。

①スライス数：一定数の LUT、フリップフロップ（FF）からなる単位である。

②LUT 数： 4 入力 1 出力の論理を実現する単位であり、SRAM で構成される。

③最大遅延：FPGA 内部リソースへのマッピング、および配置配線後の、入力（あるいは FF の出力）信号から、出力（あるいは FF の入力）信号までの遅延時間が一番長くなるパスである。

評価では、各対象回路要素に対して複数の HDL 設計、複数の回路図設計を用意し、HDL 設計間の比較、回路図設計間の比較、HDL 設計と回路図設計間の比較を行う。更に、規模の大きい回路として CPU を HDL と回路図の両方を使用して階層的に設計し、HDL と回路図の階層的な組合せをいくつか用意し、比較を行う。

3.1　各回路の評価結果

3.1.1　加算器(32 桁)

	HDL 設計①	HDL 設計②	回路図設計①	回路図設計②(キャリー先読み)
スライス数	16	46	47	104
LUT 数	32	63	91	200
最大遅延(マッピング後)	9.931ns	29.572ns	29.572ns	23.614ns
最大遅延(配置配線後)	13.146ns	42.513ns	40.914ns	42.516ns

加算器(4 桁)

	HDL 設計①	HDL 設計②	回路図設計①	回路図設計②(キャリー回路)
スライス数	4	4	5	8
LUT 数	8	7	8	16
最大遅延(マッ	6.291ns	7.060ns	7.060ns	7.864ns

ピング後)				
最大遅延(配置配線後)	8.285ns	9.371ns	8.750ns	9.964ns

HDL 設計方式同士は、論理合成ツールによってゲートレベル記述へ変換する処理がある。HDL 設計①は設計抽象度が高いため、論理合成で最適化する余地があるため、規模がより小さく、最大遅延も短くなった。(実際には、ISE ツールにより加算器のハードマクロが利用されている。) HDL 設計②は回路図設計①と同じ構造となり、最適化の余地が少ないため、HDL 設計①より規模、遅延が大きくなった。

回路図設計方式同士は、異なる構造で異なる結果となった。回路図設計②はキャリー先読みと呼ばれる構造で設計され、回路図設計①より高速化が期待できる方式である。しかし、4 桁の加算器は桁数が少ないため、キャリー先読み構造の利点が小さく、回路規模／面積の増大による配線遅延の影響が相対的に大きくなり、回路図設計①が優位に立つ。32 桁の回路図設計②は、キャリー先読み構造の効果が大きくなり、マッピング後の最大遅延は比較的小さいが、配線遅延の影響から、配置配線後の最大遅延は回路図設計①よりも大きくなった。

HDL 設計方式と回路図設計方式は、HDL 設計①が設計抽象度の高さのため、最適化余地があることに加え、ゲートレベル記述へ変換された回路を調査した結果、高速演算専用のキャリーロジックが多用されていることがわかった。このことが回路図設計に対する最も大きな優位性となっている。一方、同じ論理構造を採用する HDL 設計②と回路図設計①を比較すると、最大遅延(マッピング後)が同一であるほか、前者のほうが規模が小さくなるが、最大遅延(配置配線後)は大きくなった。さらに、32 桁回路図設計②の最大遅延(マッピング後)が HDL 設計②より小さくなっている。以上のことから、HDL 設計は記述の抽象度にしたがって、生成される回路のサイズおよび遅延において大きな相違をもたらし、場合によっては回路図設計よりも遅延が大きくなることが示された。

3.1.2　マルチプレクサ

	HDL 設計①	HDL 設計②	回路図設計
スライス数	32	32	128
LUT 数	64	64	160
最大遅延(マッピング後)	5.773ns	5.773ns	6.256ns
最大遅延(配置配線後)	12.025ns	12.759ns	16.195ns

HDL 設計方式同士は、回路規模が同一であるが、HDL 設計①の抽象度が高いことから、論理合成で最適化する余地が HDL 設計②より若干大きくなることが予想される。しかし、結果においてマッピング後の最大遅延が同一であることから、マルチプレクサ回路のように規則性がありかつ段数の多くない組合せ回路については、抽象度の相違による生成回路の遅延への影響は大きくないといえる。ゲートレベル記述へ変換された回路を調査した結果、両方において高速演算専用のキャリーロジックを含め、回路構造も類似しているため、評価値の差は大きくない。

HDL 設計方式と回路図設計方式は、HDL 設計が設計抽象度の高さのため、最適化余地があることに加え、高速演算専用のキャリーロジックが多用されていることにより、回路図設計に対して優位性が表れている。そのため、HDL 設計の規模が小さくなり、最大遅延は小さくなるという評価結果となった。

3.1.3　セグメントデコーダ

	HDL 設計	回路図設計
スライス数	4	4
LUT 数	7	7
最大遅延(マッピング後)	5.452ns	5.452ns
最大遅延(配置配線後)	7.556ns	7.716ns

HDL 設計、回路図設計ともに、使用スライス数、LUT 数、マッピング後の最大遅延において同一の結果となった。配置配線後の最大遅延については、HDL 設計の評価値が若干高くなるが、差は大きくない。マルチプレクサ回路と同様に、セグメントデコーダ回路は規則性を持ち、かつ論理段数が小さい回路であるため、設計方式の相違によって生成される回路に大きな差異は確認されなかった。

3.1.4　トライステート

	HDL 設計	回路図設計
スライス数	15	2
LUT 数	29	4
最大遅延(マッピング後)	8.973ns	8.096ns
最大遅延(配置配線後)	13.930ns	13.233ns

HDL 設計方式において、論理合成でゲートレベルへ変換された回路は、論理ゲート(AND、OR など)のみから構成されることが確認された。一方、回路図設計方式では、論理ゲートの

ほか、トライステートバッファ(BUFT)を四つ組み合わせて、データ通信を切り替えることができるように制御している。そのため、回路図設計に対して優位性が表れ、規模が小さくなり、最大遅延は小さくなるという評価結果となった。

3.1.5　シフトレジスタ

	HDL 設計①	HDL 設計②	回路図設計
スライス数	16	32	32
LUT 数	0	0	0
最大遅延(マッピング後)	6.896ns	6.892ns	6.892ns
最大遅延(配置配線後)	8.319ns	8.902ns	8.350ns

HDL 設計、回路図設計ともに、使用スライス数、LUT 数、最大遅延において大きな差は無い。(ただし、スライス数に関しては HDL 設計①が少なくなっている。) シフトレジスタはフリップフロップの単純な接続構造で構成され、規則性を持つため、設計方式の相違によって生成される回路に大きな差異は確認されなかった。

3.1.6　カウンタ

	HDL 設計①	HDL 設計②	回路図設計
スライス数	16	33	26
LUT 数	1	62	43
最大遅延(マッピング後)	6.896ns	6.896ns	6.896ns
最大遅延(配置配線後)	8.840ns	9.059ns	8.965ns

HDL 設計方式同士は、HDL 設計①の抽象度が高いことから、論理合成で最適化する余地が HDL 設計②より大きくなる。ゲートレベル記述へ変換された回路を調査した結果、両方とも高速演算専用のキャリーロジックが含まれるが、HDL 設計①のほうがキャリーロジックを多く使用し、その代わりに論理ゲート(AND、OR など)の数が押さえられている。そのため、スライス数、LUT 数、最大遅延(配置配線後)については、HDL 設計①が優位に立つことがわかった。

ゲートレベル記述へ変換された HDL 設計方式②はキャリーロジックを使用しているが、

論理ゲートの数は回路図設計方式より上回っているため、スライス数、LUT 数と配置配線後の最大遅延は回路図設計より大きくなった。

3.1.7　ステートマシン

	HDL 設計	回路図設計
スライス数	14	49
LUT 数	25	89
最大遅延(マッピング後)	7.700ns	7.700ns
最大遅延(配置配線後)	9.827ns	10.560ns

マッピング後の最大遅延は同様であるが、スライス数、LUT 数および配置配線後の最大遅延は HDL 設計方式が優位に立つことがわかった。本回路は構造が不規則であり、構成される各コンポーネントに対する最適化の余地があり、さらにゲートレベル記述へ変換された回路には、高速演算専用のキャリーロジックが使用されているため、HDL 設計の規模が小さくなり、配置配線後の最大遅延は小さくなるという評価結果となった。

3.1.8　CPU 回路

	HDL 設計	回路図設計	HDL・回路図設計混在①	HDL・回路図設計混在②
スライス数	389	1522	389	1488
LUT 数	452	1845	452	1777
最大遅延(マッピング後)	7.700ns	8.504ns	8.504ns	7.700ns
最大遅延(配置配線後)	13.161ns	15.722ns	16.179ns	14.071ns

HDL 設計と回路図設計を比較した場合、対象の CPU 回路のように、回路の複雑度が増加するほど、HDL 設計による設計抽象度の高さの優位性が顕著となり、最適化余地が多くなるため、ゲートレベル記述へ変換された回路には、高速演算専用のキャリーロジックが数多く使用されていることがわかった。このことが回路図設計に対する最も大きな優位性となっている。そのため、HDL 設計の規模がより小さく、最大遅延がより短くなることがわかった。

HDL・回路図設計混在①(以下では混在①)は HDL 設計をベースに、一部のステートマシン回路(2.3.3 節の回路)のみ回路図設計に置き換えたもの、HDL・回路図設計混在②(以下では混在②)は回路図設計をベースに、一部のステートマシン回路のみ HDL 設計に置き換えた

ものである。

HDL 設計と比較し混在①は、部分回路のステートマシンとして低い評価の回路図設計を使用しているため、回路全体の評価が低くなることがわかった。また、回路図設計と比較し、混在②は、ステートマシンとして高い評価の HDL 設計を使用しているため、回路全体の評価が高くなることがわかった。

3.2 評価まとめ

3.2.1 各回路に対する HDL 設計と回路図設計

複雑な回路になるほど、HDL 設計と比較し、回路図設計によって生成される回路の規模が大きく、最大遅延が長くなる傾向があることがわかった。HDL 設計では、設計抽象度の高い HDL 記述から、ツールによる論理合成を通して、回路図設計相当のゲートレベル回路への変換が行われる。この変換の際の最適化によって、高速演算専用のキャリーロジックが埋め込まれ、このことが回路図設計に対する最も大きな優位性となるため、よい評価結果となる。

しかし、HDL 設計における記述抽象度の相違(3.1.1 節の加算器)、回路図設計における意図的な高速論理ゲートの使用(3.1.4 節のトライステート)、回路の規則性(3.1.3 節のセグメントデコーダ、3.1.5 節のシフトレジスタ)、配線遅延などの影響により、HDL 設計、回路図設計の両設計方式には大きな差異が確認されないこともあり、あるいは逆に回路図設計より HDL 設計が規模が大きく、最大遅延が長くなることがある。このことから、生成回路の評価は対象回路のタイプや設計スタイルに依存するため、HDL 設計が一概に優位に立つとは限らない。

3.2.2 同一回路に対する異なる HDL 設計

加算器、マルチプレクサ、シフトレジスタ、カウンタに対して、二種類の HDL 設計を行った。記述抽象度が高い設計は、論理合成で最適化する余地が大きいため、より最適化された回路を生成できるほか、高速演算専用のキャリーロジックもより多く使用されるため、規模がより小さく、最大遅延も短くなることがわかった。

3.2.3 同一回路に対する異なる回路図設計

加算器に対して、二種類の回路図設計方式で設計を行った。構造の差異により論理ゲート数が大きく影響することがわかった。また、一般的には桁上げ伝搬方式よりも桁上げ

先読み方式が高速であることが知られているが、今回の評価では、桁上げ先読み方式の大きな論理サイズが原因となり配線遅延が増大したため、逆に低速になる結果が得られた。（この配線遅延の影響は、論理ブロックへのマッピング後の結果と、配置・配線後の結果が大きく異なることからわかる。）これにより、論理サイズを減少させることを配慮することにより配線遅延の影響が小さくなり、高速化が期待できる可能性がある。

3.2.4　同一回路に対する HDL 設計、回路図設計を混在する設計方法

CPU 回路に対して、HDL 設計をベースに、一部の回路を回路図設計に置き換えたものと、回路図設計をベースに、一部の回路を HDL 設計に置き換えたものと、2 種類の混在設計を行った。その一部の回路であるステートマシン回路は、単体では 3.1.7 の評価により回路図設計より HDL 設計が優位であることがわかっている。この性質は混在設計においても保存されており、HDL 設計ベースに評価の低い回路図によるステートマシンを混在させた場合は全体の評価値が低くなり、逆に評価の低い回路図設計ベースに評価の高い HDL によるステートマシンを混在させた場合は全体の評価が高くなっている。このことから、基本的には評価値の高い部分回路を組み合わせることにより、全体の回路の性能が高くできることが予想できる。

第 4 章　結論

4.1　本研究結果のまとめ

3 章の評価で示したように、HDL 設計方式を使用する大きな利点は、設計抽象度を引き上げることで、HDL 記述を合成ツールで最適化する余地を確保できることである。ゲートレベル記述へ変換された回路は、回路図設計方式で生成されるものと比較して、より小規模な回路が生成される傾向があるほか、高速演算専用のキャリーロジックが多用されていることから、回路図設計方式よりも最大遅延が短くなる傾向があることがわかった。ただし、設計規模の小さい回路では、設計方式の相違によって生成される回路に大きな差異は確認されなかった。

論理の簡単化などにより回路サイズを減少させることや、キャリーロジックなどの高速な論理ゲートを明示的に使用することにより、回路図設計により高い評価の回路を得る可能性もあるが、今日の LSI 設計では回路の規模が大きくなる傾向があり、場合によっては数千万ゲート規模の回路を設計する必要もあり、回路図設計はその描画効率と論理検証効率の観点から限界に達していると言える。このことから、今後は更に HDL 設計が優位に立つことが予想される。

異なる HDL 設計方式に対して、設計抽象度の差異により最適化の余地と使用される高速演算ロジックの数が異なるため、評価値に差が出た。異なる回路図設計方式でも、構造の差異により論理規模が異なるため、評価も異なる結果となる。さらに、3 章における混在設計に関する評価結果から、全体回路に評価が高い部分回路を含めることにより、回路全体の評価が高くなることがわかった。回路を設計する際には、回路を構成する各部分に対して、その回路タイプにしたがって優位性の高い設計方式(HDL 設計、回路図設計)で設計を行い、それらを組み合わせることにより高性能な回路の実現が可能となることが予想される。

4.2　結論

複雑な大規模回路を設計する際には、極力多くのモジュール化と階層化設計を行い、一つのモジュールの規模を小さくすることで、モジュール毎に HDL 設計、回路図設計の最適なものを選択できる幅が広がる。各モジュールを組み合わせることにより、全体の回路の最適性を向上させることが期待できる。また、小規模のモジュールは再利用が容易であり、設計効率化の有効な手段と成りうる。

4.3　今後の課題

現在の計算機ハードウェア業界で、HDL 設計は製造期間の短縮、設計資源の再利用、そのほかにも本研究で示したように、抽象度の高さからより規模が小さく最大遅延が短くなる利点があるため、回路図設計の代替として主流になり、多くの設計現場で使われるようになった。その一方で、小規模の回路では、明示的な高速論理要素の使用やカルノー図などの利用によるゲート数削減により、HDL 設計方式より最適な回路を作成することが可能となる場合がある。しかし、大規模の回路に対しては、手動による回路の最適化には限界があるため、HDL 方式と論理合成ツールの利用に優位性がある。

理想的な回路の作成が求められる場合に、両設計方式でそれぞれ設計した結果を比較した上で決定することが有効であるが、設計期間や人的な余力を考慮すると、どのような複雑さと規模の回路に対して、どちらの設計方式を採用するべきかについての基準を構築することが今後の課題である。

参考文献

[1] 設計言語 Verilog - HDL 入門　培風館　ISBN-13: 978-4563035020

[2] Verilog HDL 数字システム設計と応用　西安電子技術大学出版社. ISBN 7-5606-1165-6

[3] Verilog HDL プログラミング　人民郵電出版社.ISBN 7-115-11939-2

[4] MIPS32 Architecture For Programmers: Volume II: The MIPS32 Instruction Set. MIPS Technologies, Inc.

[5] Spartan-3 FPGA Family Data Sheet (Ver. 3.1). Xilinx, Inc. (2013).

[6] Spartan-6 Family Overview (Ver. 2.0). Xilinx, Inc. (2011)

[7] Spartan-3E FPGA Starter Kit Board User Guide(Ver 1.2). Xilinx, Inc.(2011)

謝辞

この研究を作成するにあたり，終始熱心なご指導をして下さった田中清史准教授と、適切な助言を賜り、励ましを頂きました井口寧教授に深く感謝の意を表します。そして研究を進めるにあたり多大な協力を頂きましたゼミ同輩に心から感謝いたします。

付録

ステートマシンの Verilog HDL ソース

```
module main_cntrl(
    input CLK,
    input RST,
    input [5:0] OP,
    output [1:0] ALUOP,
    output ALUSRCA,
    output [1:0] ALUSRCB,
    output MEMTOREG,
    output REGDST,
    output REGWRITE,
    output IORD,
    output MEMREAD,
    output MEMWRITE,
    output IRWRITE,
    output [1:0] PCSOURCE,
    output PCWRITE,
    output PCWRITECOND,
    output STATE0
    );

    reg [3:0] s;
    wire [3:0] ns;

    //********* combination logic circuit *********
    assign ns = NS_GEN(OP,s);
    assign STATE0 = STATE0_GEN(s);
    assign ALUOP = {ALUOP1(s), ALUOP0(s)};
    assign ALUSRCA = ALUSRCA_GEN(s);
    assign ALUSRCB = {ALUSRCB1(s), ALUSRCB0(s)};
    assign MEMTOREG = MEMTOREG_GEN(s);
    assign REGDST = REGDST_GEN(s);
    assign REGWRITE = REGWRITE_GEN(s);
    assign IORD = IORD_GEN(s);
```

```
assign MEMREAD = MEMREAD_GEN(s);
assign MEMWRITE = MEMWRITE_GEN(s);
assign IRWRITE = IRWRITE_GEN(s);
assign PCSOURCE = {PCSOURCE1(s), PCSOURCE0(s)};
assign PCWRITE = PCWRITE_GEN(s);
assign PCWRITECOND = PCWRITECOND_GEN(s);

//順序回路、reg sのロジック
always @(posedge CLK or posedge RST)
    begin
        if (RST==1'b1)
    begin
        s <= 4'b0000;
    end else
    begin
        s <= ns;
    end
end

//ファンクションNS_GEN
function [3:0] NS_GEN;
    input [5:0] OP;
    input [3:0] s;

    if( s == 4'b0000) begin
        NS_GEN[3:0] = 4'b0001;
    end else if( s == 4'b0001) begin
        if( OP == 6'b100011 || OP == 6'b101011) begin
        NS_GEN[3:0] = 4'b0010;
    end else if( OP == 6'b000000) begin
        NS_GEN[3:0] = 4'b0110;
    end else if( OP == 6'b000100 || OP == 6'b000101) begin
        NS_GEN[3:0] = 4'b1000;
    end else if( OP == 6'b000010) begin
        NS_GEN[3:0] = 4'b1001;
    end else if( OP == 6'b001000) begin
```

```
        NS_GEN[3:0] = 4'b1010;
    end else begin
        NS_GEN[3:0] = 4'b0000;
    end
    end else if( s == 4'b0010) begin
        if( OP == 6'b100011) begin
        NS_GEN[3:0] = 4'b0011;
    end else begin
        NS_GEN[3:0] = 4'b0101;
    end
    end else if( s == 4'b0011) begin
        NS_GEN[3:0] = 4'b0100;
    end else if( s == 4'b0110) begin
        NS_GEN[3:0] = 4'b0111;
    end else if( s == 4'b1010) begin
        NS_GEN[3:0] = 4'b1011;
    end else begin
        NS_GEN[3:0] = 4'b0000;
    end
endfunction

//ファンクション STATE0_GEN
function STATE0_GEN;
    input [3:0] s;

    if( s == 4'b0000) begin
        STATE0_GEN = 1'b1;
    end else begin
        STATE0_GEN = 1'b0;
    end
endfunction

//ファンクション ALUOP1
function ALUOP1;
    input [3:0] s;
```

```
    if( s == 4'b0110) begin
        ALUOP1 = 1'b1;
    end else begin
        ALUOP1 = 1'b0;
    end
endfunction

//ファンクション ALUOP0
function ALUOP0;
    input [3:0] s;

    if( s == 4'b1000) begin
        ALUOP0 = 1'b1;
    end else begin
        ALUOP0 = 1'b0;
    end
endfunction

//ファンクション ALUSRCA
function ALUSRCA_GEN;
    input [3:0] s;

    if( s == 4'b0010 || s == 4'b0110 || s == 4'b1000 || s == 4'b1010) begin
        ALUSRCA_GEN = 1'b1;
    end else begin
        ALUSRCA_GEN = 1'b0;
    end
endfunction

//ファンクション ALUSRCB1
function ALUSRCB1;
    input [3:0] s;

    if( s == 4'b0001 || s == 4'b0010 || s == 4'b1010) begin
        ALUSRCB1 = 1'b1;
    end else begin
```

```
        ALUSRCB1 = 1'b0;
    end
endfunction

//ファンクション ALUSRCB0
function ALUSRCB0;
    input [3:0] s;

    if( s == 4'b0000 || s == 4'b0001) begin
        ALUSRCB0 = 1'b1;
    end else begin
        ALUSRCB0 = 1'b0;
    end
endfunction

//ファンクション MEMTOREG
function MEMTOREG_GEN;
    input [3:0] s;

    if( s == 4'b0100) begin
        MEMTOREG_GEN = 1'b1;
    end else begin
        MEMTOREG_GEN = 1'b0;
    end
endfunction

//ファンクション REGDST
function REGDST_GEN;
    input [3:0] s;

    if( s == 4'b0111) begin
        REGDST_GEN = 1'b1;
    end else begin
        REGDST_GEN = 1'b0;
    end
endfunction
```

```
//ファンクション REGWRITE
function REGWRITE_GEN;
    input [3:0] s;

    if( s == 4'b0100 || s == 4'b0111 || s == 4'b1011) begin
        REGWRITE_GEN = 1'b1;
    end else begin
        REGWRITE_GEN = 1'b0;
    end
endfunction

//ファンクション IORD
function IORD_GEN;
    input [3:0] s;

    if( s == 4'b0011 || s == 4'b0101) begin
        IORD_GEN = 1'b1;
    end else begin
        IORD_GEN = 1'b0;
    end
endfunction

//ファンクション MEMREAD
function MEMREAD_GEN;
    input [3:0] s;

    if( s == 4'b0011) begin
        MEMREAD_GEN = 1'b1;
    end else begin
        MEMREAD_GEN = 1'b0;
    end
endfunction

//ファンクション MEMWRITE
function MEMWRITE_GEN;
    input [3:0] s;
```

```
    if( s == 4'b0101) begin
        MEMWRITE_GEN = 1'b1;
    end else begin
        MEMWRITE_GEN = 1'b0;
    end
endfunction

//ファンクション IRWRITE
function IRWRITE_GEN;
    input [3:0] s;

    if( s == 4'b0000) begin
        IRWRITE_GEN = 1'b1;
    end else begin
        IRWRITE_GEN = 1'b0;
    end
endfunction

//ファンクション PCSOURCE1
function PCSOURCE1;
    input [3:0] s;

    if( s == 4'b1001) begin
        PCSOURCE1 = 1'b1;
    end else begin
        PCSOURCE1 = 1'b0;
    end
endfunction

//ファンクション PCSOURCE0
function PCSOURCE0;
    input [3:0] s;

    if( s == 4'b1000) begin
        PCSOURCE0 = 1'b1;
    end else begin
```

```
            PCSOURCE0 = 1'b0;
        end
    endfunction

    //ファンクション PCWRITE
    function PCWRITE_GEN;
        input [3:0] s;

        if( s == 4'b0000 || s == 4'b1001) begin
            PCWRITE_GEN = 1'b1;
        end else begin
            PCWRITE_GEN = 1'b0;
        end
    endfunction

    //ファンクション PCWRITECOND
    function PCWRITECOND_GEN;
        input [3:0] s;

        if( s == 4'b1000) begin
            PCWRITECOND_GEN = 1'b1;
        end else begin
            PCWRITECOND_GEN = 1'b0;
        end
    endfunction
endmodule
```

CPU 回路の Verilog ソース

REGFILE モジュール

```
module REGFILE(
    input CLK,
    input EN,
    input [4:0] RREG1,
    input [4:0] RREG2,
    input [4:0] WREG,
    input [31:0] WDATA,
    output [31:0] RDATA1,
    output [31:0] RDATA2
    );

    wire [31:0] reg1, reg2, reg3, reg4, reg5, reg6, reg7, reg8,
    reg9, reg10, reg11, reg12, reg13, reg14, reg15, reg16,
    reg17, reg18, reg19, reg20, reg21, reg22, reg23, reg24,
    reg25, reg26, reg27, reg28, reg29, reg30, reg31;

    wire [31:1] enable;
    wire Disen;

    assign Disen = 1'b0;

    assign RDATA1 = process1(reg1, reg2, reg3, reg4, reg5, reg6, reg7,
                    reg8, reg9, reg10, reg11, reg12, reg13, reg14, reg15,
                    reg16, reg17, reg18, reg19, reg20, reg21, reg22, reg23,
                    reg24, reg25, reg26, reg27, reg28, reg29, reg30, reg31, RREG1);
    assign RDATA2 = process2(reg1, reg2, reg3, reg4, reg5, reg6, reg7,
                    reg8, reg9, reg10, reg11, reg12, reg13, reg14, reg15,
                    reg16, reg17, reg18, reg19, reg20, reg21, reg22, reg23,
                    reg24, reg25, reg26, reg27, reg28, reg29, reg30, reg31, RREG2);
    //ファンクション process1
    function [31:0] process1;
        input [31:0] reg1, reg2, reg3, reg4, reg5, reg6, reg7,
                     reg8, reg9, reg10, reg11, reg12, reg13, reg14, reg15,
                     reg16, reg17, reg18, reg19, reg20, reg21, reg22, reg23,
```

```
            reg24, reg25, reg26, reg27, reg28, reg29, reg30, reg31;
input [4:0] RREG1;

if( RREG1 == 5'b00001) begin
    process1[31:0] = reg1;
          end else if ( RREG1 == 5'b00010) begin
              process1[31:0] = reg2;
          end else if ( RREG1 == 5'b00011) begin
              process1[31:0] = reg3;
          end else if ( RREG1 == 5'b00100) begin
              process1[31:0] = reg4;
          end else if ( RREG1 == 5'b00101) begin
              process1[31:0] = reg5;
          end else if ( RREG1 == 5'b00110) begin
              process1[31:0] = reg6;
end else if ( RREG1 == 5'b00111) begin
              process1[31:0] = reg7;
          end else if ( RREG1 == 5'b01000) begin
              process1[31:0] = reg8;
          end else if ( RREG1 == 5'b01001) begin
              process1[31:0] = reg9;
          end else if ( RREG1 == 5'b01010) begin
              process1[31:0] = reg10;
          end else if ( RREG1 == 5'b01011) begin
              process1[31:0] = reg11;
          end else if ( RREG1 == 5'b01100) begin
              process1[31:0] = reg12;
          end else if ( RREG1 == 5'b01101) begin
              process1[31:0] = reg13;
          end else if ( RREG1 == 5'b01110) begin
              process1[31:0] = reg14;
          end else if ( RREG1 == 5'b01111) begin
              process1[31:0] = reg15;
          end else if ( RREG1 == 5'b10000) begin
              process1[31:0] = reg16;
          end else if ( RREG1 == 5'b10001) begin
```

```
                process1[31:0] = reg17;
            end else if ( RREG1 == 5'b10010) begin
                process1[31:0] = reg18;
            end else if ( RREG1 == 5'b10011) begin
                process1[31:0] = reg19;
            end else if ( RREG1 == 5'b10100) begin
                process1[31:0] = reg20;
            end else if ( RREG1 == 5'b10101) begin
                process1[31:0] = reg21;
            end else if ( RREG1 == 5'b10110) begin
                process1[31:0] = reg22;
            end else if ( RREG1 == 5'b10111) begin
                process1[31:0] = reg23;
            end else if ( RREG1 == 5'b11000) begin
                process1[31:0] = reg24;
            end else if ( RREG1 == 5'b11001) begin
                process1[31:0] = reg25;
            end else if ( RREG1 == 5'b11010) begin
                process1[31:0] = reg26;
            end else if ( RREG1 == 5'b11011) begin
                process1[31:0] = reg27;
            end else if ( RREG1 == 5'b11100) begin
                process1[31:0] = reg28;
            end else if ( RREG1 == 5'b11101) begin
                process1[31:0] = reg29;
            end else if ( RREG1 == 5'b11110) begin
                process1[31:0] = reg30;
            end else if ( RREG1 == 5'b11111) begin
                process1[31:0] = reg31;
    end else begin
        process1[31:0] = 32'b00000000000000000000000000000000;
    end
endfunction

//ファンクション process2
function [31:0] process2;
```

```
input [31:0] reg1, reg2, reg3, reg4, reg5, reg6, reg7,
             reg8, reg9, reg10, reg11, reg12, reg13, reg14, reg15,
             reg16, reg17, reg18, reg19, reg20, reg21, reg22, reg23,
             reg24, reg25, reg26, reg27, reg28, reg29, reg30, reg31;
input [4:0] RREG2;

if( RREG2 == 5'b00001) begin
    process2[31:0] = reg1;
          end else if ( RREG2 == 5'b00010) begin
              process2[31:0] = reg2;
          end else if ( RREG2 == 5'b00011) begin
              process2[31:0] = reg3;
          end else if ( RREG2 == 5'b00100) begin
              process2[31:0] = reg4;
          end else if ( RREG2 == 5'b00101) begin
              process2[31:0] = reg5;
          end else if ( RREG2 == 5'b00110) begin
              process2[31:0] = reg6;
end else if ( RREG2 == 5'b00111) begin
              process2[31:0] = reg7;
          end else if ( RREG2 == 5'b01000) begin
              process2[31:0] = reg8;
          end else if ( RREG2 == 5'b01001) begin
              process2[31:0] = reg9;
          end else if ( RREG2 == 5'b01010) begin
              process2[31:0] = reg10;
          end else if ( RREG2 == 5'b01011) begin
              process2[31:0] = reg11;
          end else if ( RREG2 == 5'b01100) begin
              process2[31:0] = reg12;
          end else if ( RREG2 == 5'b01101) begin
              process2[31:0] = reg13;
          end else if ( RREG2 == 5'b01110) begin
              process2[31:0] = reg14;
          end else if ( RREG2 == 5'b01111) begin
              process2[31:0] = reg15;
```

```
            end else if ( RREG2 == 5'b10000) begin
                process2[31:0] = reg16;
            end else if ( RREG2 == 5'b10001) begin
                process2[31:0] = reg17;
            end else if ( RREG2 == 5'b10010) begin
                process2[31:0] = reg18;
            end else if ( RREG2 == 5'b10011) begin
                process2[31:0] = reg19;
            end else if ( RREG2 == 5'b10100) begin
                process2[31:0] = reg20;
            end else if ( RREG2 == 5'b10101) begin
                process2[31:0] = reg21;
            end else if ( RREG2 == 5'b10110) begin
                process2[31:0] = reg22;
            end else if ( RREG2 == 5'b10111) begin
                process2[31:0] = reg23;
            end else if ( RREG2 == 5'b11000) begin
                process2[31:0] = reg24;
            end else if ( RREG2 == 5'b11001) begin
                process2[31:0] = reg25;
            end else if ( RREG2 == 5'b11010) begin
                process2[31:0] = reg26;
            end else if ( RREG2 == 5'b11011) begin
                process2[31:0] = reg27;
            end else if ( RREG2 == 5'b11100) begin
                process2[31:0] = reg28;
            end else if ( RREG2 == 5'b11101) begin
                process2[31:0] = reg29;
            end else if ( RREG2 == 5'b11110) begin
                process2[31:0] = reg30;
            end else if ( RREG2 == 5'b11111) begin
                process2[31:0] = reg31;
        end else begin
            process2[31:0] = 32'b00000000000000000000000000000000;
        end
    endfunction
```

```
        assign enable[1] = EN && (~WREG[4]) && (~WREG[3]) && (~WREG[2]) && (~WREG[1])
&& (WREG[0]);
        assign enable[2] = EN && (~WREG[4]) && (~WREG[3]) && (~WREG[2]) && (WREG[1])
&& (~WREG[0]);
        assign enable[3] = EN && (~WREG[4]) && (~WREG[3]) && (~WREG[2]) && (WREG[1])
&& (WREG[0]);
        assign enable[4] = EN && (~WREG[4]) && (~WREG[3]) && (WREG[2]) && (~WREG[1])
&& (~WREG[0]);
        assign enable[5] = EN && (~WREG[4]) && (~WREG[3]) && (WREG[2]) && (~WREG[1])
&& (WREG[0]);
        assign enable[6] = EN && (~WREG[4]) && (~WREG[3]) && (WREG[2]) && (WREG[1])
&& (~WREG[0]);
        assign enable[7] = EN && (~WREG[4]) && (~WREG[3]) && (WREG[2]) && (WREG[1])
&& (WREG[0]);
        assign enable[8] = EN && (~WREG[4]) && (WREG[3]) && (~WREG[2]) && (~WREG[1])
&& (~WREG[0]);
        assign enable[9] = EN && (~WREG[4]) && (WREG[3]) && (~WREG[2]) && (~WREG[1])
&& (WREG[0]);
        assign enable[10] = EN && (~WREG[4]) && (WREG[3]) && (~WREG[2]) && (WREG[1])
&& (~WREG[0]);
        assign enable[11] = EN && (~WREG[4]) && (WREG[3]) && (~WREG[2]) && (WREG[1])
&& (WREG[0]);
        assign enable[12] = EN && (~WREG[4]) && (WREG[3]) && (WREG[2]) && (~WREG[1])
&& (~WREG[0]);
        assign enable[13] = EN && (~WREG[4]) && (WREG[3]) && (WREG[2]) && (~WREG[1])
&& (WREG[0]);
        assign enable[14] = EN && (~WREG[4]) && (WREG[3]) && (WREG[2]) && (WREG[1])
&& (~WREG[0]);
        assign enable[15] = EN && (~WREG[4]) && (WREG[3]) && (WREG[2]) && (WREG[1])
&& (WREG[0]);
        assign enable[16] = EN && (WREG[4]) && (~WREG[3]) && (~WREG[2]) && (~WREG[1])
&& (~WREG[0]);
        assign enable[17] = EN && (WREG[4]) && (~WREG[3]) && (~WREG[2]) && (~WREG[1])
&& (WREG[0]);
        assign enable[18] = EN && (WREG[4]) && (~WREG[3]) && (~WREG[2]) && (WREG[1])
&& (~WREG[0]);
```

```
        assign enable[19] = EN && (WREG[4]) && (~WREG[3]) && (~WREG[2]) && (WREG[1])
&& (WREG[0]);
        assign enable[20] = EN && (WREG[4]) && (~WREG[3]) && (WREG[2]) && (~WREG[1])
&& (~WREG[0]);
        assign enable[21] = EN && (WREG[4]) && (~WREG[3]) && (WREG[2]) && (~WREG[1])
&& (WREG[0]);
        assign enable[22] = EN && (WREG[4]) && (~WREG[3]) && (WREG[2]) && (WREG[1])
&& (~WREG[0]);
        assign enable[23] = EN && (WREG[4]) && (~WREG[3]) && (WREG[2]) && (WREG[1])
&& (WREG[0]);
        assign enable[24] = EN && (WREG[4]) && (WREG[3]) && (~WREG[2]) && (~WREG[1])
&& (~WREG[0]);
        assign enable[25] = EN && (WREG[4]) && (WREG[3]) && (~WREG[2]) && (~WREG[1])
&& (WREG[0]);
        assign enable[26] = EN && (WREG[4]) && (WREG[3]) && (~WREG[2]) && (WREG[1])
&& (~WREG[0]);
        assign enable[27] = EN && (WREG[4]) && (WREG[3]) && (~WREG[2]) && (WREG[1])
&& (WREG[0]);
        assign enable[28] = EN && (WREG[4]) && (WREG[3]) && (WREG[2]) && (~WREG[1])
&& (~WREG[0]);
        assign enable[29] = EN && (WREG[4]) && (WREG[3]) && (WREG[2]) && (~WREG[1])
&& (WREG[0]);
        assign enable[30] = EN && (WREG[4]) && (WREG[3]) && (WREG[2]) && (WREG[1])
&& (~WREG[0]);
        assign enable[31] = EN && (WREG[4]) && (WREG[3]) && (WREG[2]) && (WREG[1])
&& (WREG[0]);

REG32 REG32_1(.CLK(CLK),.RST(Disen),.EN(enable[1]),.DIN(WDATA),.DOUT(reg1));
REG32 REG32_2(.CLK(CLK),.RST(Disen),.EN(enable[2]),.DIN(WDATA),.DOUT(reg2));
REG32 REG32_3(.CLK(CLK),.RST(Disen),.EN(enable[3]),.DIN(WDATA),.DOUT(reg3));
。。。(途中省略)。。。
REG32 REG32_31(.CLK(CLK),.RST(Disen),.EN(enable[31]),.DIN(WDATA),.DOUT(reg31));
endmodule
```

REG32 モジュール

```
module REG32(
```

```
    input CLK,
    input RST,
    input EN,
    input [31:0] DIN,
    output reg [31:0] DOUT = 32'h0
    );

    always @(posedge CLK or posedge RST)
        begin
            if (RST == 1'b1) begin
                DOUT <= 32'h0;
            end else if (EN == 1'b1) begin
                DOUT <= DIN;
        end
    end
endmodule
```

ALU32 モジュール

```
module ALU32(
    input [31:0] SRCA,
    input [31:0] SRCB,
    input [2:0] ALU_CONTROL_INPUT,
    output [31:0] ALU_RESULT
    );

    wire [31:0] and_result;
    wire [31:0] or_result;
    wire [31:0] add_result;

    assign and_result = SRCA & SRCB;
    assign or_result = SRCA | SRCB;
    assign add_result = SRCA + SRCB;

    assign ALU_RESULT
        = process(ALU_CONTROL_INPUT, and_result, or_result, add_result);
```

```
    //ファンクション process3
    function [31:0] process;
        input [2:0] ALU_CONTROL_INPUT;
        input [31:0] and_result;
        input [31:0] or_result;
        input [31:0] add_result;

        if( ALU_CONTROL_INPUT == 3'b000) begin
            process = and_result;
                end else if ( ALU_CONTROL_INPUT == 3'b001) begin
                    process = or_result;
                end else if ( ALU_CONTROL_INPUT == 3'b010) begin
                    process = add_result;
        end else begin
            process = 32'b00000000000000000000000000000000;
        end
    endfunction
endmodule
```

MUX2TO1_32 モジュール

```
module MUX2TO1_32(
    input [31:0] DIN0,
    input [31:0] DIN1,
    input SEL,
    output [31:0] DOUT
    );

    assign DOUT = process(DIN0, DIN1, SEL);

    //ファンクション process
    function [31:0] process;
        input [31:0] DIN0, DIN1;
        input SEL;

        if( SEL == 1'b1) begin
            process[31:0] = DIN1;
```

```
        end else begin
            process[31:0] = DIN0;
        end
    endfunction
endmodule
```

MUX2TO1_5 モジュール

```
module MUX2TO1_5(
    input [4:0] DIN0,
    input [4:0] DIN1,
    input SEL,
    output [4:0] DOUT
    );

    assign DOUT = process(DIN0, DIN1, SEL);

    //ファンクション process
    function [4:0] process;
        input [4:0] DIN0, DIN1;
        input SEL;

        if( SEL == 1'b1) begin
            process[4:0] = DIN1;
        end else begin
            process[4:0] = DIN0;
        end
    endfunction
endmodule
```

MUX4TO1_32 モジュール

```
module MUX4TO1_32(
    input [31:0] DIN0,
    input [31:0] DIN1,
    input [31:0] DIN2,
    input [31:0] DIN3,
    input [1:0] SEL,
```

```
    output [31:0] DOUT
    );

    assign DOUT = process(DIN0, DIN1, DIN2, DIN3, SEL);

    //ファンクション process
    function [31:0] process;
        input [31:0] DIN0, DIN1, DIN2, DIN3;
        input [1:0] SEL;

        if (SEL == 2'b11) begin
            process[31:0] = DIN3;
        end else if (SEL == 2'b10) begin
            process[31:0] = DIN2;
        end else if (SEL == 2'b01) begin
            process[31:0] = DIN1;
        end else begin
            process[31:0] = DIN0;
        end
    endfunction
endmodule
```

MUX3TO1_32 モジュール

```
module MUX3TO1_32(
    input [31:0] DIN0,
    input [31:0] DIN1,
    input [31:0] DIN2,
    input [1:0] SEL,
    output [31:0] DOUT
    );

    assign DOUT = process(DIN0, DIN1, DIN2, SEL);

    //ファンクション process
    function [31:0] process;
        input [31:0] DIN0, DIN1, DIN2;
```

```
        input [1:0] SEL;

        if (SEL == 2'b10) begin
            process[31:0] = DIN2;
        end else if (SEL == 2'b01) begin
            process[31:0] = DIN1;
        end else begin
            process[31:0] = DIN0;
        end
    endfunction
endmodule
```

SIGN_EXTEND モジュール

```
module SIGN_EXTEND(
    input [15:0] DIN,
    output [31:0] DOUT
    );

    wire [15:0] sign;

    assign sign = { 16{DIN[15]} };
    assign DOUT = {sign, DIN};
endmodule
```

SHIFT_L2_32 モジュール

```
module SHIFT_L2_32(
    input [29:0] DIN,
    output [31:0] DOUT
    );

    assign DOUT = {DIN, 2'b00};
endmodule
```

SHIFT_L2_28 モジュール

```
module SHIFT_L2_28(
    input [25:0] DIN,
```

```
    output [27:0] DOUT
    );

    assign DOUT = {DIN, 2'b00};
endmodule
```

MAIN_CNTRL モジュール

```
module MAIN_CNTRL(
    input CLK,
    input RST,
    input [5:0] OP,
    output [1:0] ALUOP,
    output ALUSRCA,
    output [1:0] ALUSRCB,
    output MEMTOREG,
    output REGDST,
    output REGWRITE,
    output IORD,
    output MEMREAD,
    output MEMWRITE,
    output IRWRITE,
    output [1:0] PCSOURCE,
    output PCWRITE,
    output PCWRITECOND,
    output STATE0
    );

    reg [3:0] s = 4'h0;
    wire [3:0] ns;

    //******** combination logic circuit ********
    assign ns = NS_GEN(OP,s);
    assign STATE0 = STATE0_GEN(s);
    assign ALUOP = {ALUOP1(s), ALUOP0(s)};
    assign ALUSRCA = ALUSRCA_GEN(s);
    assign ALUSRCB = {ALUSRCB1(s), ALUSRCB0(s)};
```

```
assign MEMTOREG = MEMTOREG_GEN(s);
assign REGDST = REGDST_GEN(s);
assign REGWRITE = REGWRITE_GEN(s);
assign IORD = IORD_GEN(s);
assign MEMREAD = MEMREAD_GEN(s);
assign MEMWRITE = MEMWRITE_GEN(s);
assign IRWRITE = IRWRITE_GEN(s);
assign PCSOURCE = {PCSOURCE1(s), PCSOURCE0(s)};
assign PCWRITE = PCWRITE_GEN(s);
assign PCWRITECOND = PCWRITECOND_GEN(s);

//順序回路、reg sのロジック
always @(posedge CLK or posedge RST)
    begin
        if (RST == 1'b1)
            begin
                s <= 4'b0000;
            end
         else
            begin
                s <= ns;
          end
      end

//ファンクションNS_GEN
function [3:0] NS_GEN;
    input [5:0] OP;
    input [3:0] s;

    if( s == 4'b0000) begin
        NS_GEN[3:0] = 4'b0001;
    end else if( s == 4'b0001) begin
        if( OP == 6'b100011 || OP == 6'b101011) begin
            NS_GEN[3:0] = 4'b0010;
        end else if( OP == 6'b000000) begin
            NS_GEN[3:0] = 4'b0110;
```

```
        end else if( OP == 6'b000100 || OP == 6'b000101) begin
            NS_GEN[3:0] = 4'b1000;
        end else if( OP == 6'b000010) begin
            NS_GEN[3:0] = 4'b1001;
        end else if( OP == 6'b001000) begin
            NS_GEN[3:0] = 4'b1010;
        end else begin
            NS_GEN[3:0] = 4'b0000;
        end
    end else if( s == 4'b0010) begin
        if( OP == 6'b100011) begin
            NS_GEN[3:0] = 4'b0011;
        end else begin
            NS_GEN[3:0] = 4'b0101;
        end
    end else if( s == 4'b0011) begin
        NS_GEN[3:0] = 4'b0100;
    end else if( s == 4'b0110) begin
        NS_GEN[3:0] = 4'b0111;
    end else if( s == 4'b1010) begin
        NS_GEN[3:0] = 4'b1011;
    end else begin
        NS_GEN[3:0] = 4'b0000;
    end
endfunction

//ファンクション STATE0_GEN
function STATE0_GEN;
    input [3:0] s;

    if( s == 4'b0000) begin
        STATE0_GEN = 1'b1;
    end else begin
        STATE0_GEN = 1'b0;
    end
endfunction
```

```
//ファンクション ALUOP1
function ALUOP1;
    input [3:0] s;

    if( s == 4'b0110) begin
        ALUOP1 = 1'b1;
    end else begin
        ALUOP1 = 1'b0;
    end
endfunction

//ファンクション ALUOP0
function ALUOP0;
    input [3:0] s;

    if( s == 4'b1000) begin
        ALUOP0 = 1'b1;
    end else begin
        ALUOP0 = 1'b0;
    end
endfunction

//ファンクション ALUSRCA
function ALUSRCA_GEN;
    input [3:0] s;

    if( s == 4'b0010 || s == 4'b0110 || s == 4'b1000 || s == 4'b1010) begin
        ALUSRCA_GEN = 1'b1;
    end else begin
        ALUSRCA_GEN = 1'b0;
    end
endfunction

//ファンクション ALUSRCB1
function ALUSRCB1;
    input [3:0] s;
```

```
    if( s == 4'b0001 || s == 4'b0010 || s == 4'b1010) begin
        ALUSRCB1 = 1'b1;
    end else begin
        ALUSRCB1 = 1'b0;
    end
endfunction

//ファンクション ALUSRCB0
function ALUSRCB0;
    input [3:0] s;

    if( s == 4'b0000 || s == 4'b0001) begin
        ALUSRCB0 = 1'b1;
    end else begin
        ALUSRCB0 = 1'b0;
    end
endfunction

//ファンクション MEMTOREG
function MEMTOREG_GEN;
    input [3:0] s;

    if( s == 4'b0100) begin
        MEMTOREG_GEN = 1'b1;
    end else begin
        MEMTOREG_GEN = 1'b0;
    end
endfunction

//ファンクション REGDST
function REGDST_GEN;
    input [3:0] s;

    if( s == 4'b0111) begin
        REGDST_GEN = 1'b1;
    end else begin
```

```
        REGDST_GEN = 1'b0;
    end
endfunction

//ファンクション REGWRITE
function REGWRITE_GEN;
    input [3:0] s;

    if( s == 4'b0100 || s == 4'b0111 || s == 4'b1011) begin
        REGWRITE_GEN = 1'b1;
    end else begin
        REGWRITE_GEN = 1'b0;
    end
endfunction

//ファンクション IORD
function IORD_GEN;
    input [3:0] s;

    if( s == 4'b0011 || s == 4'b0101) begin
        IORD_GEN = 1'b1;
    end else begin
        IORD_GEN = 1'b0;
    end
endfunction

//ファンクション MEMREAD
function MEMREAD_GEN;
    input [3:0] s;

    if( s == 4'b0011) begin
        MEMREAD_GEN = 1'b1;
    end else begin
        MEMREAD_GEN = 1'b0;
    end
endfunction
```

```
//ファンクション MEMWRITE
function MEMWRITE_GEN;
    input [3:0] s;

    if( s == 4'b0101) begin
        MEMWRITE_GEN = 1'b1;
    end else begin
        MEMWRITE_GEN = 1'b0;
    end
endfunction

//ファンクション IRWRITE
function IRWRITE_GEN;
    input [3:0] s;

    if( s == 4'b0000) begin
        IRWRITE_GEN = 1'b1;
    end else begin
        IRWRITE_GEN = 1'b0;
    end
endfunction

//ファンクション PCSOURCE1
function PCSOURCE1;
    input [3:0] s;

    if( s == 4'b1001) begin
        PCSOURCE1 = 1'b1;
    end else begin
        PCSOURCE1 = 1'b0;
    end
endfunction

//ファンクション PCSOURCE0
function PCSOURCE0;
    input [3:0] s;
```

```
        if( s == 4'b1000) begin
            PCSOURCE0 = 1'b1;
        end else begin
            PCSOURCE0 = 1'b0;
        end
    endfunction

    //ファンクション PCWRITE
    function PCWRITE_GEN;
        input [3:0] s;

        if( s == 4'b0000 || s == 4'b1001) begin
            PCWRITE_GEN = 1'b1;
        end else begin
            PCWRITE_GEN = 1'b0;
        end
    endfunction

    //ファンクション PCWRITECOND
    function PCWRITECOND_GEN;
        input [3:0] s;

        if( s == 4'b1000) begin
            PCWRITECOND_GEN = 1'b1;
        end else begin
            PCWRITECOND_GEN = 1'b0;
        end
    endfunction
endmodule
```

ALU_CNTRL モジュール

```
module ALU_CNTRL(
    input [3:0] FUNCT,
    input [1:0] ALUOP,
    output [2:0] ALU_CNTRL_IN
    );
```

```
wire [2:0] r_frmt_cntrl;

assign ALU_CNTRL_IN = process1(ALUOP);
assign r_frmt_cntrl = process2(FUNCT);

//ファンクション process
function [2:0] process1;
    input [1:0] ALUOP;

    if( ALUOP == 2'b00) begin
        process1[2:0] = 3'b010;
    end else if ( ALUOP == 2'b01) begin
        process1[2:0] = 3'b110;
    end else if ( ALUOP == 2'b10) begin
        process1[2:0] = r_frmt_cntrl;
    end else begin
        process1[2:0] = 3'b110;
    end
endfunction

//ファンクション process
function [2:0] process2;
    input [3:0] FUNCT;

    if( FUNCT == 4'b1010) begin
        process2[2:0] = 3'b111;
    end else if ( FUNCT == 4'b0101) begin
        process2[2:0] = 3'b001;
    end else if ( FUNCT == 4'b0100) begin
        process2[2:0] = 3'b000;
    end else if ( FUNCT == 4'b0010) begin
        process2[2:0] = 3'b110;
    end else if ( FUNCT == 4'b0000) begin
        process2[2:0] = 3'b010;
    end else if ( FUNCT == 4'b1000) begin
```

```
            process2[2:0] = 3'b101;
        end else begin
            process2[2:0] = 3'b100;
        end
    endfunction
endmodule
```

上位 CPU モジュール

```
module CPU(
    input CLK,
    input RST,
    output [31:0] INST_ADR,
    output reg [31:0] DATA = 32'h0,
    output [31:0] PC,
    input [31:0] INST,
    output MEMREAD,
    output MEMWRITE,
    output [31:0] ADDRESS,
    output [31:0] WRITE_DATA,
    input [31:0] MEMDATA
    );

    wire [4:0] RegDst_sig;
    wire [31:0] MemtoReg_dat;
    wire [31:0] PC_in, PC_out;
    wire [31:0] IR_out;
    wire [31:0] MDR_out;
    wire [31:0] AREG_in, BREG_in;
    wire [31:0] AREG_out, BREG_out;

    wire Zero;
    wire [31:0] ALUSrcA_dat, ALUSrcB_dat;
    wire [31:0] ALUResult;
    wire [31:0] ALUOut_out;
    wire [31:0] SExtend_dat;
    wire [31:0] Shift_dat32;
```

```
wire [27:0] Shift_dat28;

wire [1:0] PCSource;
wire [1:0] ALUOp;
wire ALUSrcA;
wire [1:0] ALUSrcB;
wire RegWrite;
wire RegDst;
wire IRWrite;
wire MemtoReg;
wire IorD;
wire PCWrite;
wire PCWriteCond;
wire PCWrite_sig;
wire [2:0] ALU_CNTRL_IN;

wire En;
wire [31:0] Four;
wire [31:0] J_adr;

wire State0;
reg [31:0] inst_adr_in = 32'h0;

assign En = 1'b1;
assign Four = 32'b00000000000000000000000000000100;

assign PC = PC_out;

assign INST_ADR = INST_ADR_SEL(State0, PC_out, inst_adr_in);

always @(posedge CLK or posedge RST)
begin
    if (RST == 1'b1) begin
        inst_adr_in <= 32'h0;
    end else if (State0 == 1'b1) begin
        inst_adr_in <= PC_out;
```

```
        end
    end

    //ファンクション process
    function [31:0] INST_ADR_SEL;
        input State0;
        input [31:0] PC_out, inst_adr_in;

        if( State0 == 1'b1) begin
            INST_ADR_SEL[31:0] = PC_out;
        end else begin
            INST_ADR_SEL[31:0] = inst_adr_in;
        end
    endfunction

    always @(posedge CLK or posedge RST)
        begin
            if (RST == 1'b1) begin
                DATA <= 32'h0;
            end else if (RegWrite == 1'b1) begin
                DATA <= MemtoReg_dat;
        end
    end


REGFILE  REGFILE_MAP(
    .CLK(CLK),.EN(RegWrite),.RREG1(IR_out[25:21]),.RREG2(IR_out[20:16]),
    .WREG(RegDst_sig),.WDATA(MemtoReg_dat),.RDATA1(AREG_in),.RDATA2(BREG_in));
REG32 PC_MAP(.CLK(CLK),.RST(RST),.EN(PCWrite_sig),.DIN(PC_in),.DOUT(PC_out));
REG32 IR_MAP(.CLK(CLK),.RST(RST),.EN(IRWrite),.DIN(INST),.DOUT(IR_out));
REG32 MDR_MAP(.CLK(CLK),.RST(RST),.EN(En),.DIN(MEMDATA),.DOUT(MDR_out));
REG32 AREG_MAP(.CLK(CLK),.RST(RST),.EN(En),.DIN(AREG_in),.DOUT(AREG_out));
REG32 BREG_MAP(.CLK(CLK),.RST(RST),.EN(En),.DIN(BREG_in),.DOUT(BREG_out));
REG32 ALUOut_MAP(.CLK(CLK),.RST(RST),.EN(En),.DIN(ALUResult),.DOUT(ALUOut_out));

ALU32  ALU32_MAP(
```

```
    .SRCA(ALUSrcA_dat),.SRCB(ALUSrcB_dat),.ALU_CONTROL_INPUT(ALU_CNTRL_IN,
    .ALU_RESULT(ALUResult));

    assign Zero = 1'b0;

MUX2TO1_32 MUX_IorD(.DIN0(PC_out),.DIN1(ALUOut_out),.SEL(IorD),.DOUT(ADDRESS));
MUX2TO1_5 MUX_RegDst(
    .DIN0(IR_out[20:16]),.DIN1(IR_out[15:11]),.SEL(RegDst),.DOUT(RegDst_sig));
MUX2TO1_32 MUX_MemtoReg(
    .DIN0(ALUOut_out),.DIN1(MDR_out),.SEL(MemtoReg),.DOUT(MemtoReg_dat));
MUX2TO1_32 MUX_ALUSrcA(
    .DIN0(PC_out),.DIN1(AREG_out),.SEL(ALUSrcA),.DOUT(ALUSrcA_dat));
MUX4TO1_32 MUX_ALUSrcB(
    .DIN0(BREG_out),.DIN1(Four),.DIN2(SExtend_dat),
    .DIN3(Shift_dat32),.SEL(ALUSrcB),.DOUT(ALUSrcB_dat));

    assign J_adr = {PC_out[31:28], Shift_dat28};

MUX3TO1_32 MUX_PCSource(
  .DIN0(ALUResult),.DIN1(ALUOut_out),.DIN2(J_adr),.SEL(PCSource),.DOUT(PC_in));

SIGN_EXTEND Sign_Extend16to32(.DIN(IR_out[15:0]),.DOUT(SExtend_dat));
SHIFT_L2_32 Shift_Left2_32(.DIN(SExtend_dat[29:0]),.DOUT(Shift_dat32));
SHIFT_L2_28 Shift_Left2_28(.DIN(IR_out[25:0]),.DOUT(Shift_dat28));

MAIN_CNTRL Main_Control(
    .CLK(CLK),.RST(RST),.OP(IR_out[31:26]),
    .ALUOP(ALUOp),.ALUSRCA(ALUSrcA),.ALUSRCB(ALUSrcB),
    .MEMTOREG(MemtoReg),.REGDST(RegDst),.REGWRITE(RegWrite),
    .IORD(IorD),.MEMREAD(MEMREAD),.MEMWRITE(MEMWRITE),
    .IRWRITE(IRWrite),.PCSOURCE(PCSource),.PCWRITE(PCWrite),
    .PCWRITECOND(PCWriteCond),.STATE0(State0));

ALU_CNTRL ALU_Control(
    .FUNCT(IR_out[3:0]),.ALUOP(ALUOp),.ALU_CNTRL_IN(ALU_CNTRL_IN));
```

```
    assign WRITE_DATA = BREG_out;
    assign PCWrite_sig =
        PCWrite || (PCWriteCond && ((Zero && !IR_out[26]) || (!Zero && IR_out[26])));
endmodule
```