卒業論文

巡回冗長検査 CRC32 のハード/ソフト最適分割の検討

氏　　名 : 伊藤　大喜
学籍番号 : 2260050004-3
指導教員 : 山崎　勝弘　教授
提 出 日 : 2009年2月19日

立命館大学　理工学部電子情報デザイン学科

内容概要

本論文では、LSI 設計の主流となっているハードウェア記述言語の Verilog-HDL を用いて、CRC32 回路を設計することで Vreilog-HDL 記述の理解を深めるとともに、ハードウェア化における効果や性能などを検証する。Verilog-HDL は、ソフトウェア開発者にも受け入れられるように C 言語や Pascal の要素を取り入れて開発された言語であり、シミュレーション機能も充実している。

本論文では CRC32 回路のアルゴリズムを理解した後、実際に C 言語での設計、モジュール分割案を提案し、実際に Verilog-HDL を用いて設計する。またソフトウェア言語の C 言語でのソフトウェア記述での実現も行う。また、実験で得られたデータなどを元に、それぞれの利点などを考慮し、機能分割を行うことにより最適な分割案を考察する。

目次

図目次

表目次

1. はじめに

1970 年代以降、LSI は劇的なスピードで微細化、高性能化をしてきた。これらは、家電用品、携帯電話、自動車など、様々なものに利用され、生活になくてはならない物となっている。今日では VLSI,ULSI などと呼ばれる、さらに集積度の高い物も発明されている。こうした環境下で、ハードウェアの開発環境も劇的に進歩している。ハードウェア記述言語 HDL、FPGA、シミュレーションアクセラレータなどの発明がそれに当たるだろう。これらによって開発はより高速化、簡易化、低コスト化を実現し、大きな役割を担っている。しかしながらこれらを使用しても、間に合わないという現実があり、これらを打破する策としての 1 つがハード/ソフト協調設計である。これは何かを設計する場合、機能毎に予め役割を決めておき、それらの優先させる事柄にあわせてハードウェア、ソフトウェアどちらで実現するかを検討する。例えばハードウェアの処理速度の利点を生かして、速度を重視した回路の実現、ソフトウェアの柔軟性を生かし、バージョンアップなどを容易にした回路の実現などがある[1][2][3]。本研究ではこの回路設計の方法の一つであるハード/ソフト協調設計という技術について学ぶ。

本研究では通信で使用される誤り検出を行う CRC という技術に着目し、その中の代表として CRC32 を扱った。CRC を始めとした誤り検出などは、情報の送受信を行う過程などで常に使用される技術なので検出する作業にかかる処理時間は直接情報の送受信にかかる時間に関わってくる。また、送受信する情報の規模によっても処理時間に影響が出る[5][8]。そこで本研究では、これらの回路をハードウェア化して処理時間の短縮を図る。また、作成した回路について、ハードウェア、ソフトウェアの最適な分割案を検討する。

本研究ではこうした LSI 設計の高度化が著しいという時代背景と、誤り検出での実行時間はファイル送信時間に大きく影響するということから、Verilog-HDL を使用し、CRC32 回路の設計を実際に行う。また、一方で CRC32 回路を C 言語を用いた方法でもこれらの設計を行う。後にそれぞれについて動作検証、評価を行う。ハードウェア設計では回路規模、遅延、実行クロック数などを求め、ソフトウェア設計では負荷割合などを求める。また、実験で得たモジュールの負荷割合などを参考に、ハードウェア、ソフトウェアの持つそれぞれのメリットやデメリットなどを考慮し、各自の特性を生かして、ハードウェアとソフトウェアの最適な分割案を検討する。さらに、実験を行った上での考察を行う。

本研究室では本研究とは別で「C ソースコード解析によるハード/ソフト最適分割システムの構築」という研究が行われている。[4]この研究ではハード/ソフト協調設計を行うにあたり、早期段階でそれらの最適な分割パターンを解析するといったものである。この研究の一環として、本研究で作成する回路を利用し、本研究を別の研究にも有効活用するということも今後考えることができるだろう。

各章の構成について、２章では CRC32 についての概要、アルゴリズムの説明からはじまり、C 言語を用い、実際に CRC32 回路の設計を行う。３章では実際にハードウェア化を考慮し、モジュールを分割し、ハードウェア記述を行い、CRC32 回路を設計していき、分割した各モジュールの説明や、関係などについても触れる。また、完成した回路を検証、評価する。4 章では実際に作成した回路をもとにして、ハードウェアとソフトウェア間で機能の分割の検討を行っていく。

2. 巡回冗長検査 CRC32 の概要

2.1 CRC32 とは

CRC とは Cyclic Redundancy Check の略であり、巡回冗長検査というデータの誤りを検出するための誤り検出符号の一種の事である。データの誤りが発生したときに対応する技術は、誤りを検出するのみで、訂正する能力は備えていない誤り検出方式と、誤りを検出し、訂正する能力を備えている誤り訂正方式に大別される。CRC は誤りを検出するのみの誤り検出方式である。CRC は他の冗長符号の方式であるパリティチェック方式よりも冗長度に比較して高い誤り検出能力を持ち、バースト誤りにも強いという利点を持つ。

CRC は一定の生成多項式による除数の余りを検査用の冗長ビットとする方法である。ここで言う一定の生成多項式とは、唯一の標準規格があるわけではない。生成多項式の一部の名称と用途を表 1 に示す。

誤り検出能力や、CRC 値の衝突などにも関わってくることがあるので、生成多項式の選択は CRC を実装する上で非常に重要である。

表 1：CRC の生成多項式

名称	生成多項式	主な用途
CRC-1	x^1+1	パリティビット
CRC-4-ITU	x^4+x^1+1	ITU G.704
CRC-5-ITU	$x^5+x^4+x^2+1$	ITU G.704
CRC-5-USB	x^5+x^2+1	USB トークンパケット
CRC-8-ATM	x^8+x^2+x+1	ATM
CRC-12	$x^{12}+x^{11}+x^3+x^2+x^1+1$	通信系
CRC-16	$x^{16}+x^{15}+x^2+x^1+1$	SDLC、USB
CRC-30	$x^{30}+x^{29}+x^{21}+x^{20}+x^{15}+x^{13}+x^{12}+x^{11}+x^8+x^7+x^6+x^2+x^1+1$	CDMA
CRC-32	$x^{32}+x^{26}+x^{23}+x^{22}+x^{16}+x^{12}+x^{11}+x^{10}+x^8+x^7+x^5+x^4+x^2+x^1+1$	V.42, MPEG-2, PNG

2.2　CRC32 アルゴリズム

図 1 にデータの送受信という立場で CRC の概要を示す。CRC を使用したファイルの送受信の場合、送信側は 1 伝送単位ごとにファイルのビット列を 2 進数とみなし生成多項式で割った余りをチェックビットとして付加して送信する。一方、受信側は伝送された情報を同じ多項式で割る。すると送信した情報に誤りが無い限りは割り切れるのは明白である。よって余りが 0 になれば誤りが発生していないと判断することができる。

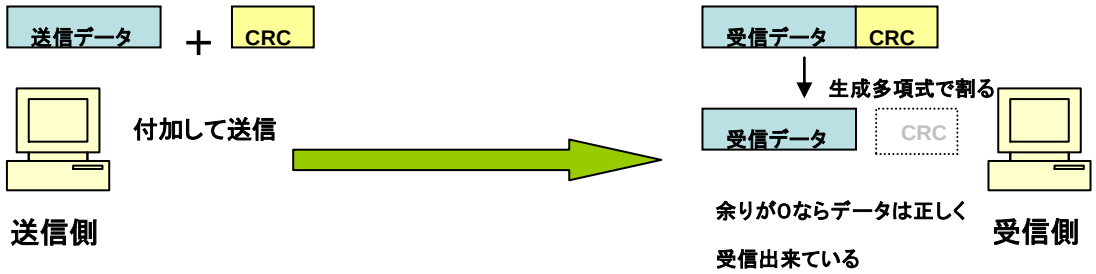


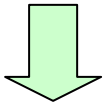
図 1：ファイルの送受信における CRC

以下に CRC の演算方法を入力ビット列が 11010011101100 で生成多項式が 1011 の場合について説明する。以下のように入力ビット列の左端に除数を表す生成多項式を並べる。まずは除数の左端のビットの上部にあるビットを判定する。0 である場合、そのまま除数のビット列を右に 1 つシフトする。1 である場合、入力ビットと出力ビットそれぞれのビットを EX-OR 演算を行い、除数のビット列を右に 1 つシフトする。今回の例では、判定は 1 なので EX-OR 演算を行っている。

```
入力ビット列　11010011101100
除数ビット列　1011
              ----------------
結果ビット列　01100011101100
```

どちらかの演算が 1 度完了すれば、その結果ビット列を次の入力ビット列に代入し、同じように除数で割っていく。

以前の結果ビット列　01100011101100



入力ビット列　01100011101100
除数ビット列　1011
結果ビット列　00111011101100

これらの作業を除数ビット列の右端が入力ビット列の右端に到達するまで繰り返す。すると最終的には以下の結果が得られる。

入力ビット列　00000000001110
除数ビット列　1011
結果ビット列　00000000000101

この結果ビット列が今回行った除算の余りとなる。一方でこれが誤り検出に使用される CRC の値となる。

CRC32 の場合、ここで使用される除数ビット列、つまり生成多項式は表 1 にある通り、$x^{32}+x^{26}+x^{23}+x^{22}+x^{16}+x^{12}+x^{11}+x^{10}+x^{8}+x^{7}+x^{5}+x^{4}+x^{2}+x^{1}+1$ つまりビット列にすると、100000100110000010001110110110111 である。計算は同様に行うことができる。

2.3　CRC32 回路の C 言語による実現

CRC32 のデータ構造、アルゴリズムに関してより理解を深めるためにまずは C 言語を用いて CRC32 回路の実現を行った。

始めに入力ビット列の終端を知る必要があり、それを取得するためループする。取得すればそれを使用して多項式がそのビット列の終端に達していないかを判別し、達していれば演算を終了する。達していなければ次の処理へ、進む。次の処理では判定用のビットに着目し、１の場合、ループし１ビットずつ EX-OR を行う。０の場合、演算は行わない。その後、判定用のビットのアドレスを一つ進め、次のビットに着目する。これらの処理を繰り返し、多項式が終端に達したところで演算を終える。演算を終えれば結果が出力される。概念図を図 2 に示す。

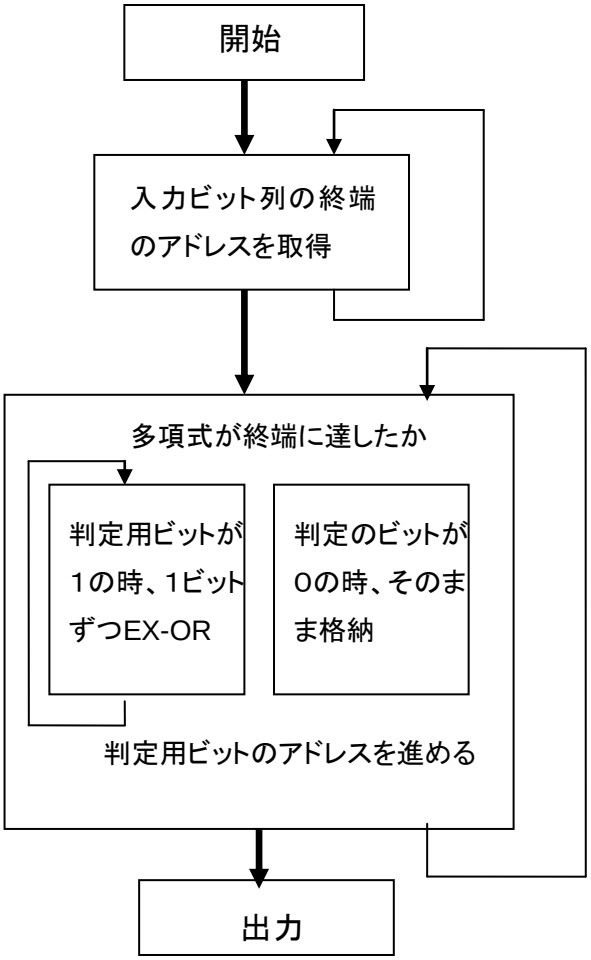


図 2：CRC32 回路の C 言語による実現

3. CRC32 回路のハードウェア設計

3.1 CRC32 回路の設計

CRC32 回路の設計を行うに当たって、使用する全てのモジュールをハードウェア設計をするため、Verilog-HDL での記述を行った。

CRC32 の冗長ビット列を生成する上ではまず、検査対象のビット列が必要となる。このビット列は入力データによって全く異なり、様々な値に変化する。よって、頻繁に書き換わることが考えられるため、検査対象のビット列のデータを収め、送信する機能を持たせたモジュールを用意し、このモジュールを memory と名付けた。

また、入力データを受け取り、その値によって EX-OR 演算などの演算を行い、実際に CRC32 の冗長ビット列を生成するモジュールを用意し、このモジュールには calculator と名付けた。図 3 に全体のモジュールの構成を示す。

両モジュール間では 2 つの値のやり取りを行う。1 つ目は data である。data は検査対象であるビット列で memory モジュール内に用意されている情報を 1 ビットずつ calculator モジュールに送り出す時の値である。検査対象のビット列は、データによって、桁数も異なるため、ビットの終端も異なってくる。そこで、bit_count というものを用意し、これにより入力されるビット列の終端を判断する。

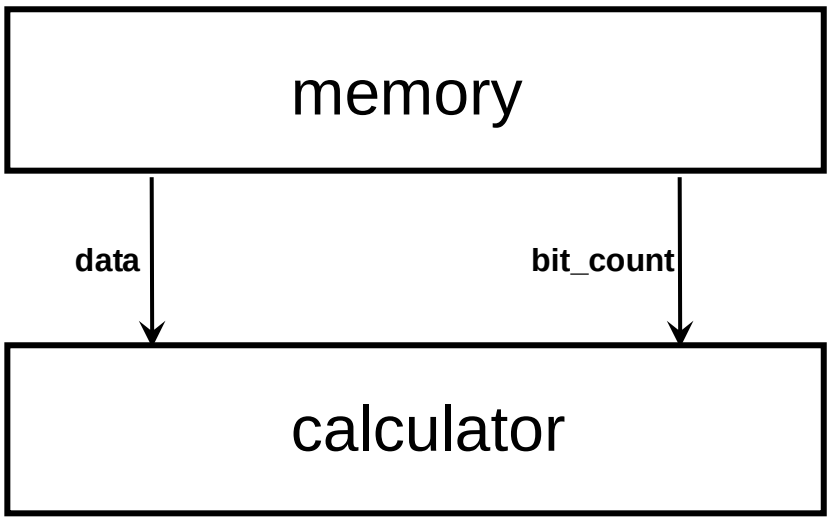


図 3：CRC32 モジュール構成

3.2　各モジュールの説明

3.2.1　memory モジュール

memory モジュールの主な役割は検査対象であるビット列を calculator モジュールに 1 ビットずつ送信していく。また、それらのビット列がすべて送信出来れば、calculator にビット列の終了を報告する。モジュールの構成を図 4、入出力の各信号線の名称、ビット幅、機能を表 2 示した

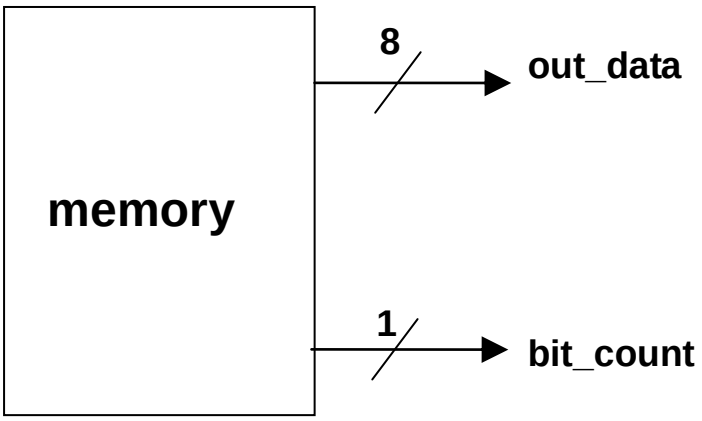


図 4：memory モジュール

表 2：memory の信号線

信号名	方向	幅(bit)	詳細
out_data	output	8	送信データ
bit_count	output	1	データ受信停止信号

memory モジュールにはあらかじめ値を格納する配列のレジスタが用意されている。そこに検査対象となるビット列である、2 または 0 のビット列を入力し、配列の値の無い部分には目印として 2 を入力する設計にした。このモジュールでは値を配列の先頭から 1 つずつ送信し、2 の値が検出されれば、送信ビット列の終了とみなし目印として bit_count を送信する。ビット列の送信の概念を図 5 に示す。

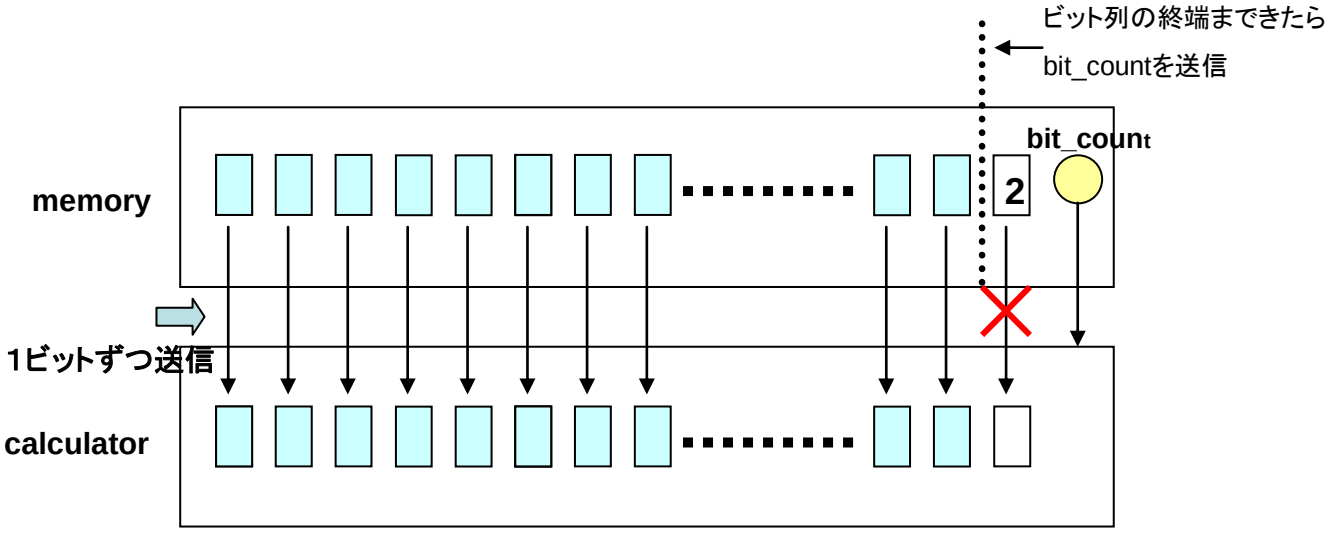


図 5：ビット列の更新

3.2.2　calculator モジュール

calculator モジュールの役割は主に剰余の演算である。

memory レジスタから送られてくるデータを受信し、それらに対して 2.2 で述べた演算を行い検査用のビット列を算出する。モジュールの構成を図 6、入出力の各信号線の名称、ビット幅、機能を表 3 に示した。

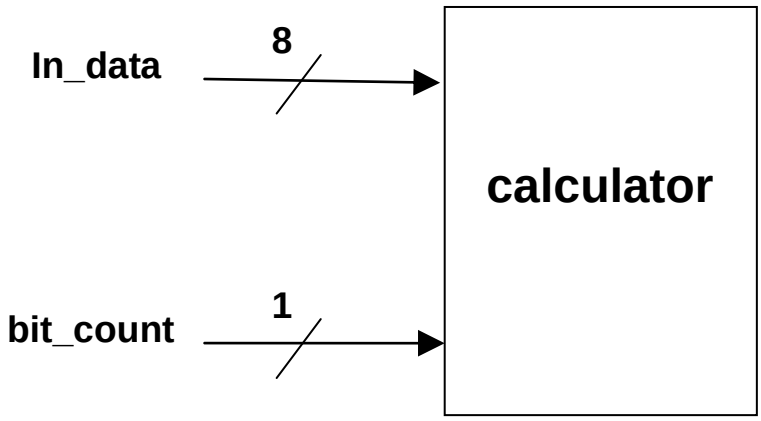


図 6：calculator モジュール

表 3：calculator の信号線

信号名	方向	幅(bit)	詳細
in_data	input	8	受信データ
bit_count	input	1	データ受信停止信号

calculator モジュールは memory モジュールから 1 ビットずつ data 配列に値を受信していく。memory モジュールで送信ビット列の終端が検出されれば bit_count に 1 を受信する。bit_count を受信すれば calculator モジュールはビット列の値の格納を終了し、次の動作に移る。

CRC32 の生成多項式は一定であるため、その値は calculator モジュール内に記憶している。これらの値を使用して演算を行う。

(1) 判定ビットが 1 の場合は data 配列と crc32 配列を 1 ビットずつ EX-OR し data 配列に格納する。data 配列に格納する際、1 ビットずつ格納が行われるため、途中で判定ビットも書き換わり、判定に支障をきたす場合がある。

そこで一度、結果を別に用意した配列に格納する。EX-OR 演算が完了すれば、用意した配列のデータを data 配列に格納する。この際、判定ビットが格納されている場所の演算結果は別の場所に退避させ、他の場所すべてに結果の格納が完了した後、最後に退避させた値が格納される。図 7 に判定ビットが 1 の時の計算の概念を示す。

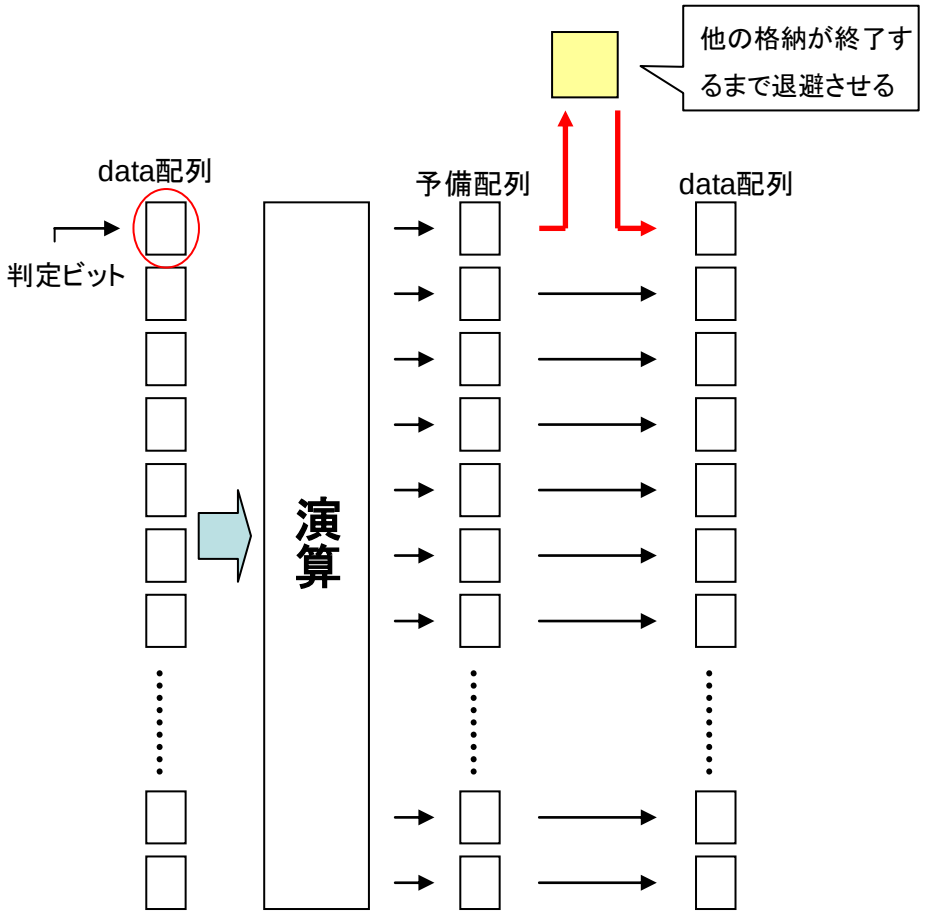


図 7：判定ビットが 1 の時の結果の格納

(2) 判定ビットが 0 の場合は EX-OR 演算を行わずに、そのまま同じ場所に同じ値を格納していく。これを配列の終端まで行う。

(3) 配列の始めからそれぞれ判定を繰り返していき、data 配列の終端まで crc32 配列の終端が達したら、つまり data 配列の終端のアドレスが crc32 配列の終端を表す値と一致したら、そこで演算は終了。そのときの data 配列の値が求めるべき剰余のビット列となる。
crc32 配列は 33 ビットであるので終端を表すアドレスの初期値は 33 である。この値は判定を繰り返す毎にシフトするので 1 ずつ増加していく。最終的にこの値と、calculator モジュールが memory モジュールからの data 配列の値の受信を停止したときのアドレスとが一致したときに演算は終了する。
図 8 に演算の終了時の概念を示す。

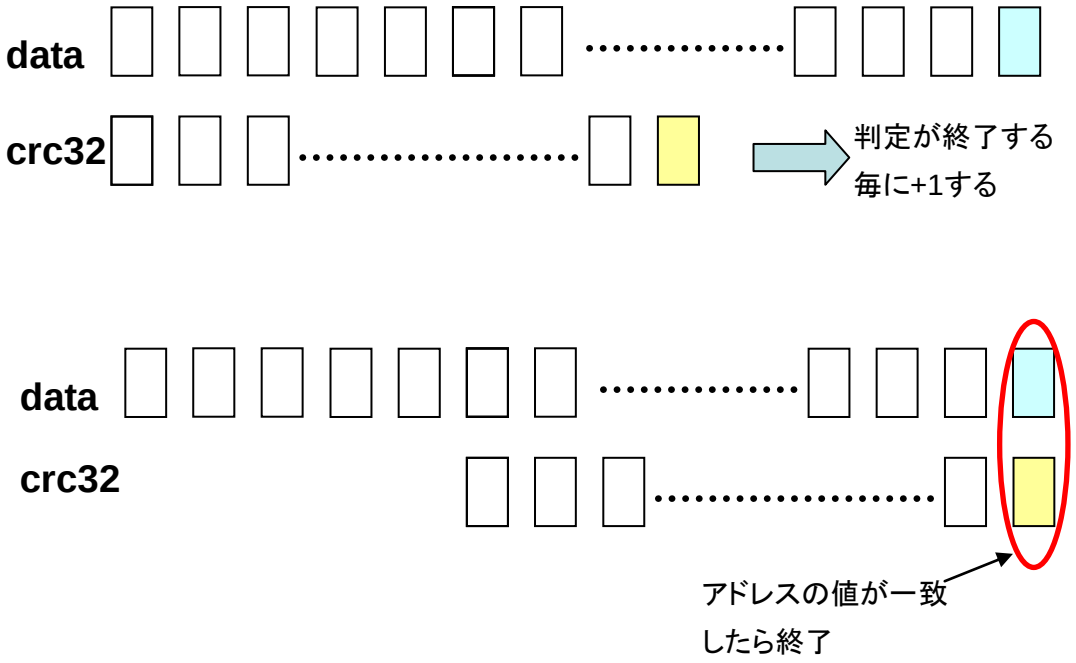


図 8：演算の終了

3.3　検証と評価

完成した Verilog 記述の CRC32 を Xilinx 社の ModelSim XEⅢを使用し、シミュレーションを行った。ModelSim はデザインのファンクションおよびタイミング モデル、HDL ソース コードを検証することができるシミュレーションツールである。

今回実験を行った動作環境を表 4 に示す。

表 4：動作環境

名称	IntelliStation Z Pro
CPU	Xeon
動作周波数	3.00GHz

検証を行うに当たって、入力ビット列には任意に決めたビット列である、10101000010000101100101010010101000011 の 38 ビットを用意した。この入力ビット列に対して、正しく出力されているかの観測、また、要する時間、クロック数などを観測した。計算結果を観測すると、手計算を行った時の結果と同一な、00000010100111000110111100010101110101 という結果が出力された。よって、演算は正しく行えているものであると予想できる。一方、今回シミュレーションを行うに当たってのテストベンチの記述ではクロックは 50ps 毎に立ち上がり、立ち下がりを繰り返す。ModelSim 上でシミュレーションを行い、波形を出力し観測した結果、演算が終了するまでには 32.45ns かかっていた。よって、1 クロックに要する時間は 100ps であることから、32.4ns / 100ps を計算すると 324 となり、この回路では演算を終了するまでには 324 クロックを要した。

記述した CRC32 回路の動作が正しく行えていることが確認できたため、それらの回路の論理合成を行った。今回、論理合成を行うことによって各モジュールの回路規模、遅延時間を得た。そのときの Verilog 記述行数も併せて表 5 に示す。表 5 によると calculator モジュールが memory モジュールよりも回路規模が大幅に上回っている。これは演算量やレジスタ数などの記述が多く素子や配線を多く必要とするからであると考えられる。

表 5：CRC32 回路のハードウェア性能

モジュール名	回路規模 [スライス	遅延 [ns]	記述行数
memory	457	7.05	135
calculator	1999	6.08	162

4. ハード/ソフト最適分割の検討

4.1 ハード/ソフト協調設計の検討

ハード/ソフト協調設計とはシステムを構成するハードウェア、ソフトウェアをそのシステム自体の性能やコストなどを考慮して最適と思われる構成に設計を行うことである。これらを実現するには図 9 に示した流れで実現することができる。

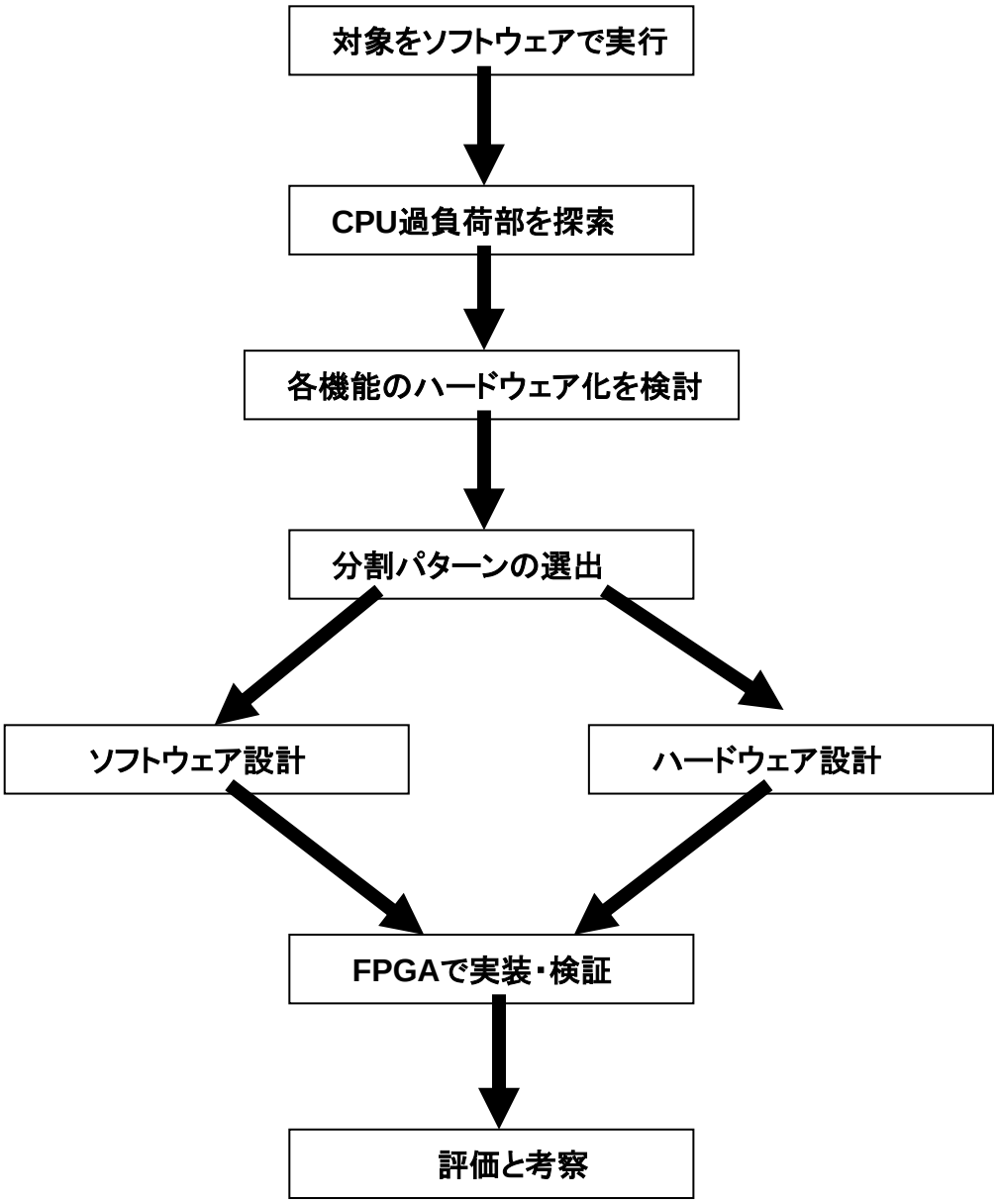


図 9：ハード/ソフト分割の流れ

(1) 対象をソフトウェアで実行

CRC32 についてアルゴリズムなどを理解しつつ、C 言語で記述し、実際にソフトウェアでの実行を行う。ここで動作の理解を深める。

(2) CPU 過負荷部を探索

CPU 負荷を計測。過負荷部を探索し、ブロックごとに負荷を把握する。

(3) 各機能のハードウェア化を検討

機能、負荷などの観点から各モジュール毎のハードウェア化を検討する。

(4) 分割パターンの選出

各モジュールについて性能を考慮し分割パターンを選出する。

(5) ソフトウェア設計、ハードウェア設計

ハードウェア、ソフトウェアの双方で各モジュールを設計、動作を確認。

(6) FPGA で実装・検証

完成した回路を FPGA 上に実装し、検証を行う。

本実験ではここまで至らなかった。

(7) 評価と考察

計測結果から評価、考察。

これらを踏まえて、CRC32 回路の協調設計の検討を行うに当たり、本実験では以下の流れで実験を行った。まず、C 言語による CRC32 回路のソフトウェア設計である。これにより動作の理解を深めた。次に、ハードウェア化の検討を行い、各機能でモジュールの分割案を考察した。次に、それらを実際に Verilog-HDL でハードウェア設計を行い、ModelSim 上で波形を観測、検証を行った。各モジュール毎のハードウェア化が完了すると、協調設計を考慮したソフトウェア設計を行った。それらから得られた情報を元に、評価、考察を行う。

4.2　CRC32 回路のモジュール分割の検討と考察

モジュール分割を考慮する上で必要となるので、ハードウェア設計で、正しい動作が確認された後、モジュール分割案の通りに関数を用意したタイプのソフトウェア設計を考慮し、それらを C 言語を使用して記述した。動作も確認し正しく動作したことが確認できた後、完成したプログラムを元に、それぞれの関数について、負荷割合を算出した。結果は図 10 に示す。

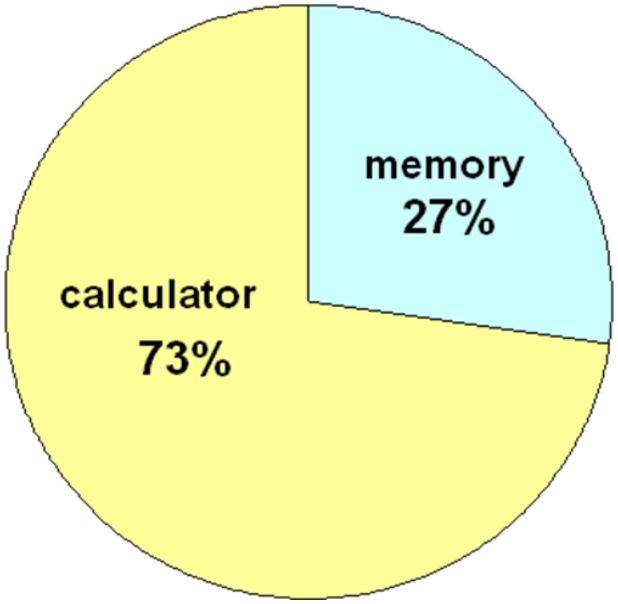


図 10：CRC32 回路の負荷割合

図を見ると、calculator モジュールの負荷割合が memory モジュールの負荷割合と比べて大きいことがわかる。これは、calculator モジュールは memory モジュールよりも、より複雑な演算を行っているからだと考えられる。これらを参考に、ハード・ソフトの分割を考慮する。今回、これらを機能ブロック単位で分割する。表 6 に機能ブロック単位での考えられる分割パターンを示す。

表 6：パターン別性能

分類	ハードウェア処理部	ソフトウェア処理部	回路規模	実行クロック数
A		memory, calculator	0	126597
B	memory	calculator	457	
C	calculator	memory	1999	
D	memory, calculator		2456	324

表を見ると、ABCD の計 4 パターンの組み合わせが考えられる。これらの組み合わせそれぞれについて検討を行う。A の分割案の場合、memory モジュール、calculator モジュールのどちらもソフトウェアでの実現なので、専用のハードウェアを用意する必要ない。しかし、ソフトウェアでの実現なので実行時間はかかってしまうという欠点がある。B の分割案の場合、負荷の大きい calculator モジュールがソフトウェア実現、負荷の小さい memory モジュールがハードウェア実現であり、あまり効率がいい分割案とは言えない。C の分割案の場合、負荷の大きい calculator モジュールがハードウェア実現、負荷の小さい memory モジュールはソフトウェア実現であり、最もバランスが取れた分割案だと考えられる。D の分割案の場合、memory モジュール、calculator モジュールのどちらもハードウェアでの実現であり、この分割案は最もスピードに特化した分割案であると考えられる。しかしながら回路規模は大きくなる。

今回の例では、検証の都合上、計算する桁数は限られていたが、calculator モジュールで行う計算による負荷は、入力桁数を増やせば増加すると考えられる。これは桁数が増加すれば計算量も増加するためである。つまり、calculator モジュールに関しては扱うデータ量が大きくなった場合の処理速度などを考慮すると、ハードウェアでの実現が理想的だと考えられる。memory モジュールに関しては、扱うメモリ数が大きくそれが原因で必要以上に回路規模が増加していると考えられる。実際の動作自体は軽いものなので、コスト面など、様々なことを考慮に入れると、memory モジュールはソフトウェアでの実現が適していると考えられる。

本実験ではモジュール分割は値を送信する memory モジュールと、受信し、演算を行う calculator モジュールを提案し設計したが、calculator モジュールをもう少し細かく分割するタイプの分割案も考えられる。例として図 11 に示すように、calculator モジュールを 3 つに分割する案を考えた。1 つ目は data モジュールで、役割は memory モジュールから値を受け取る、後に説明する 2 つのモジュールからの演算結果を受け取る、演算の終了の判定である。2 つ目は calcu1 モジュールで判定ビットが 1 の時の動作を行い、data モジュールに結果を送る。3 つ目は calcu0 モジュールで判定ビットが 0 の時の動作を行い、data モジュールに結果を送る。最終的に data モジュールに演算結果が格納されることになる。このように、少ない機能でも分割することによって、ハード/ソフト機能分割案はより多様になり、分割による効果をより得やすくなると考えられる。

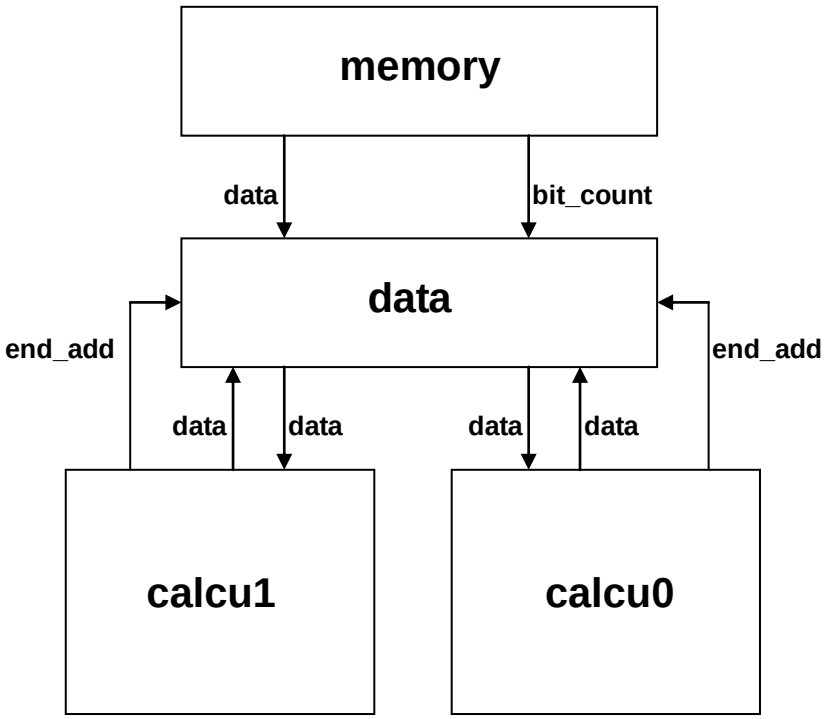


図 11：別のモジュール分割例

本実験の掲げた目的は CRC32 回路のハード/ソフト最適分割の検討である。本実験では実際に FPGA 上での実装、検証にまでは至らなかったが、本来 MicroBlaze を用いて、FPGA 上にシステムを実装し、より正確な結果を検証、考察することが重要である。

5. おわりに

本研究では現在の LSI 開発環境下で求められているハード/ソフト最適分割の技術について考察するため、代表として CRC32 回路を例にとり、回路設計を行った。始めに CRC32 についてのアルゴリズムの理解、C 言語での設計による動作理解、モジュール分割、ハードウェア設計、ソフトウェア設計などを行い、分割案の提案を行った。

今後の課題はソフト・マクロ CPU である MicroBlaze を用いて、FPGA 上にシステムを実装し、ソフトウェアプログラムの動的命令実行数を観測して、より精密な予測を行うことがあげられる。

今後も LSI 設計技術の大幅な向上は予想するのにたやすい。そういった中でハードウェア、ソフトウェア両方の技術の習得がますます期待される。本研究室ではそれらの実現を目的とした学生などが多く、研究に必要とされる設備も整っており、非常に恵まれた環境である。今後、こういった環境を生かし、本研究室での研究が技術の進歩に貢献することを期待している。

謝辞

本研究の機会を与えてくださり、貴重な助言ご指導をいただきました山崎勝弘教授に深く感謝いたします。

また、色々と助言や励ましを下さった高性能計算研究室の皆様に心より深く感謝いたします。

参考文献

[1] 和田智行:Misty1 暗号回路の設計とハード/ソフト最適分割の検討,卒業論文,2007,03.

[2] 梅原直人:ハード/ソフト最適分割を考慮したAES暗号システムとJPEGエンコーダの設計と検証,卒業論文,2005,03.

[3] 的場督永:ハード／ソフト最適分割を考慮した JPEGエンコーダの協調設計,卒業論文,2005,03.

[4] C ソースコード解析によるハード/ソフト最適分割システムの構築,2009,03

[5] 情報処理技術者試験ソフトウェア開発技術者完全教本,日本経済新聞社,2005

[6] やさしい C,ソフトバンクパブリッシング,2002

[7] 改訂・入門 Verilog HDL 記述,CQ 出版,2007

[8] Wikipedia(巡回冗長検査)
http://ja.wikipedia.org/wiki/%E5%B7%A1%E5%9B%9E%E5%86%97%E9%95%B7%E6%A4%9C%E6%9F%BB

付録
CRC32 回路のハードウェア記述

・ memory モジュール

```
module data_crc32(clk,rst,out_data,bit_count);

    input               clk,rst;
    output      [7:0]   out_data;
    output              bit_count;
    reg         [7:0]   data_r   [0:63];
    reg         [7:0]   address_r;
    reg                 bit_count_r;

    assign  out_data  =  (bit_count_r==1'b0)?     data_r[address_r]:
                                                  data_r[address_r];
    assign  bit_count   =    bit_count_r;

    always @(posedge clk or negedge rst) begin
        if (rst==1'b0) begin
            address_r <= 8'h00;
        end
        else if (data_r[address_r] != 8'h02) begin
            address_r <= address_r +1;
        end
        else begin
           address_r <= address_r;
        end
    end

    always @(posedge clk or negedge rst) begin
        if (rst==1'b0) begin
            bit_count_r <= 1'b0;
        end
        else if (rst==1'b1) begin
            if (data_r[address_r] == 8'h02) begin
                bit_count_r <= 1'b1;
```

```
            end
            else begin
                bit_count_r <= 1'b0;
            end
        end
        else begin
            bit_count_r <= 1'b0;
        end
    end

always @(posedge clk or negedge rst) begin
        if (rst==1'b0) begin
            data_r[0] <= 1'b1;
            data_r[1] <= 1'b0;
            data_r[2] <= 1'b1;
            data_r[3] <= 1'b0;
            data_r[4] <= 1'b1;
            data_r[5] <= 1'b0;
            data_r[6] <= 1'b0;
            data_r[7] <= 1'b0;
            data_r[8] <= 1'b0;
            data_r[9] <= 1'b1;
            data_r[10] <= 1'b0;
            data_r[11] <= 1'b0;
            data_r[12] <= 1'b0;
            data_r[13] <= 1'b0;
            data_r[14] <= 1'b1;
            data_r[15] <= 1'b0;
            data_r[16] <= 1'b1;
            data_r[17] <= 1'b1;
            data_r[18] <= 1'b0;
            data_r[19] <= 1'b0;
            data_r[20] <= 1'b1;
            data_r[21] <= 1'b0;
            data_r[22] <= 1'b1;
            data_r[23] <= 1'b0;
```

```
data_r[24] <= 1'b1;
data_r[25] <= 1'b0;
data_r[26] <= 1'b0;
data_r[27] <= 1'b1;
data_r[28] <= 1'b0;
data_r[29] <= 1'b1;
data_r[30] <= 1'b0;
data_r[31] <= 1'b1;
data_r[32] <= 1'b0;
data_r[33] <= 1'b0;
data_r[34] <= 1'b0;
data_r[35] <= 1'b0;
data_r[36] <= 1'b1;
data_r[37] <= 1'b1;
data_r[38] <= 8'h02;
data_r[39] <= 8'h02;
data_r[40] <= 8'h02;
data_r[41] <= 8'h02;
data_r[42] <= 8'h02;
data_r[43] <= 8'h02;
data_r[44] <= 8'h02;
data_r[45] <= 8'h02;
data_r[46] <= 8'h02;
data_r[47] <= 8'h02;
data_r[48] <= 8'h02;
data_r[49] <= 8'h02;
data_r[50] <= 8'h02;
data_r[51] <= 8'h02;
data_r[52] <= 8'h02;
data_r[53] <= 8'h02;
data_r[54] <= 8'h02;
data_r[55] <= 8'h02;
data_r[56] <= 8'h02;
data_r[57] <= 8'h02;
data_r[58] <= 8'h02;
data_r[59] <= 8'h02;
```

```
            data_r[60] <= 8'h02;
            data_r[61] <= 8'h02;
            data_r[62] <= 8'h02;
            data_r[63] <= 8'h02;
        end
        else begin
            data_r[address_r] <= data_r[address_r];
        end
    end

endmodule
```

・calculator モジュール

```
module calculator(clk,rst,in_data,bit_count);

    input               clk,rst;
    input               bit_count;
    input       [7:0]   in_data;
    reg                 bit_count_r;
    reg         [7:0]   data_r   [0:100];
    reg         [7:0]   address_r;
    reg                 crc32_r [0:63];             //crc32 = 104C11DB7
    reg                 ans_r[0:100];           //最終的な答え
    reg         [7:0]   add_ans_r;
    reg         [7:0]   add_data_r;
    reg         [7:0]   add_data2_r;
    reg         [7:0]   add_crc_r;
    reg         [7:0]   end_crc_r,end_crc2_r;
    reg                 a;

    always @(posedge clk or negedge rst) begin
        if (rst==1'b0) begin
            crc32_r[0] <= 1'b1;
            crc32_r[1] <= 1'b0;
            crc32_r[2] <= 1'b0;
```

```
            crc32_r[3] <= 1'b0;
            crc32_r[4] <= 1'b0;
            crc32_r[5] <= 1'b0;
            crc32_r[6] <= 1'b1;
            crc32_r[7] <= 1'b0;
            crc32_r[8] <= 1'b0;
            crc32_r[9] <= 1'b1;
            crc32_r[10] <= 1'b1;
            crc32_r[11] <= 1'b0;
            crc32_r[12] <= 1'b0;
            crc32_r[13] <= 1'b0;
            crc32_r[14] <= 1'b0;
            crc32_r[15] <= 1'b0;
            crc32_r[16] <= 1'b1;
            crc32_r[17] <= 1'b0;
            crc32_r[18] <= 1'b0;
            crc32_r[19] <= 1'b0;
            crc32_r[20] <= 1'b1;
            crc32_r[21] <= 1'b1;
            crc32_r[22] <= 1'b1;
            crc32_r[23] <= 1'b0;
            crc32_r[24] <= 1'b1;
            crc32_r[25] <= 1'b1;
            crc32_r[26] <= 1'b0;
            crc32_r[27] <= 1'b1;
            crc32_r[28] <= 1'b1;
            crc32_r[29] <= 1'b0;
            crc32_r[30] <= 1'b1;
            crc32_r[31] <= 1'b1;
            crc32_r[32] <= 1'b1;
        end
    end
    always @(posedge clk or negedge rst) begin
        if (rst==1'b0) begin
            bit_count_r <= 1'b0;
            add_data2_r          <= 8'h00;
```

```
    add_crc_r           <= 8'h00;
    add_ans_r <= 8'h00;
    end_crc2_r <= 6'b100001;
    address_r <= 8'h00;
    add_data_r          <= 8'h00;
    end_crc_r <= 6'b100001;
end
else if (rst==1'b1) begin
    if (bit_count_r==1'b0) begin
        bit_count_r <= bit_count;
        if (in_data == 2'b10)begin
          data_r[address_r] <= 1'bz;
          ans_r[address_r] <= 1'bz;
        end
        else begin
          data_r[address_r] <= in_data;
          ans_r[address_r] <= in_data;
          address_r <= address_r + 8'h01;
        end
    end
    else if (bit_count_r==1'b1) begin
         bit_count_r <= bit_count;
        if(end_crc2_r < address_r) begin
                if(data_r[add_data_r]==1'b1) begin                  /
                    if(add_crc_r < end_crc_r) begin                          /
                        ans_r[add_data2_r] <= data_r[add_data2_r]^crc32_r[add_crc_r];
                        add_data2_r <= add_data2_r + 8'h01;
                        add_crc_r <= add_crc_r + 8'h01;
                    end
                    else begin
                        if((add_ans_r != add_data_r) && (add_ans_r <= address_r -1))begin
                            data_r[add_ans_r] <= ans_r[add_ans_r];
                            add_ans_r <= add_ans_r + 8'h01;
                        end
                        else if((add_ans_r == add_data_r) && (add_ans_r <= address_r -1))
                        begin
```

```
                        a <= ans_r[add_ans_r];
                        add_ans_r <= add_ans_r + 8'h01;
                    end
                    else begin
                        data_r[add_data_r] <= a;
                        add_ans_r <= 8'h00;
                        add_data_r <= add_data_r + 8'h01;
                        add_data2_r <= add_data_r + 8'h01;
                        add_crc_r <= 1'b0;
                        end_crc2_r <= end_crc2_r + 8'h01;
                    end
                end
            end
            else   begin
                if(add_crc_r < end_crc_r) begin
                    add_data2_r <= add_data2_r + 8'h01;
                    add_crc_r <= add_crc_r + 8'h01;
                end
                else begin
                    add_data_r <= add_data_r + 8'h01;
                    add_data2_r <= add_data_r + 8'h01;
                    add_crc_r <= 1'b0;
                    end_crc2_r <= end_crc2_r + 8'h01;
                end
            end
        end
    end
  end
endmodule
```