Memoirs of the Faculty of Science
Kochi University (Information Science)
Vol. 26 (2005), No. 3

３２ビット浮動小数点演算器の機能検証と改善
（制御部とアキュムレータ）

城間周博　田原学　田部哲也　國信茂郎

高知大学理学部数理情報科学科

Abstract

近年の集積回路の高集積度による VLSI（大規模集積回路）の出現により、HDL（ハードウェア記述言語）を用いた機能設計からトップダウン設計が重要になってきている。

本研究では、HDL を用いて浮動小数点プロセッサ(Floating Point Unit)の設計を行った。また、コンピュータ上での回路シミュレーションを行った。

1　序論

我々の身の回りにある電気・電子機器にはLSIが多数搭載されている。例えばパソコンやゲーム機にはCPUやメモリ、さらには周辺機器制御用LSIなどの様々なLSIが搭載されている。このような状況が出現したのは、半導体のプロセス技術の進展によりLSIに搭載できる回路の規模が増大し、高度で多様な機能を実現できるようになったからである。

その背景になるコンピュータの中核テクノロジの世代区分を説明すると、第一世代は1951年から1959年で真空管が使われ、第二世代（1960年から1968年）に入るとトランジスタが使われた。トランジスタとは電気的なオン/オフ動作をするスイッチである。第三世代（1969年から1977年）に移ると集積回路すなわちチップが登場する。集積回路は数十から数百のトランジスタを１チップにまとめて接続したものである。その後、１チップに載せられるトランジスタ数は、数百から数百万へと驚異的に増大した。このような集積回路のことを超大規模集積回路と呼ぶ。この超大規模集積回路が第四世代（1978年から現在）の主役である。この超大規模集積回路のことを略してVLSIと呼ぶ。

ここで世代ごとに単位コストあたりの相対性能比を比べてみる。第一世代の値を1とすると、第二世代は35、第三世代は900、そして第四世代は2,400,000となり驚異的に高性能化されている。ここで分かるように年代を重ねるごとに急激に高集積、高性能化されてきている。もっとも回路が大規模で複雑なものになれば回路設計の困難は増大する。そこでLSIの大規模化・複雑化に対応する設計技術にも革新が起こった。すなわち、RTL回路をHDLと言うハードウェアを設計するための設計言語で記述し、LSI をトップダウン的に設計する手法が開発され、広く用いられるようになった。

ここで設計技術の進展を見てみるとLSI設計のシミュレーション技術は、1980年代に入りゲートレベルシミュレーションが自動化され、ボード上でのシミュレーションからコンピュータ上でのシミュレーションが可能になった。しかし、近年の設計規模の拡大により従来の設計方法である回路図入力ではゲートレベルシミュレーションを用いても大規模回路を要求通りに短期間で開発することが困難になってきたので設計時間の短縮とコスト削減の要求性、また優れた論理合成ツールの出現によりゲートレベル設計から HDL 記述設計へ変異している。

HDLとは回路の動作や構造を記述するための言語で、回路設計では従来のように回路図入力ではなくHDLによって回路動作を記述する。HDLは主にC言語のような記述性をもつVerilog-HDLとIEEE Std-1076規格に基づくVHDLがある。HDLの出現により、以前までのセル部から設計を始め、最終的にターゲットの製品を作成する方法（ボトムアップ設計）から、機能の検証からアプローチする方法（トップダウン設計）に移行している。トップダウン設計では設計工程を機能検証工程とゲートへの具体化作業工程の二つに分けることにより、設計者ははじめからタイミングなどの考慮をする必要がなく、機能設計に関するバグを早い段階で見つけることが出来、開発期間を短縮している。

本研究では、「ＨＤＬ（Hardware Description Language）を用いたマイクロプロセッサの構築」をテーマとして研究を行っている。一昨年（２００２年)、同研究室ではVerilog-HDL を用いて MIPS 命令を扱う３２ビット CPU の設計を行った。昨年度は一昨年の継続として３２ビット浮動小数点プロセッサの構築を行った。本年度はその３２ビット浮動小数点プロセッサ（FPU）の再構築を目標とした。

本論文は、３２ビット浮動小数点プロセッサ(FPU)の再構築に取り組んだものである。FPU を専用プロセッサとして追加することによって、マトリックス演算を実行し、3次元物体を移動および回転することが可能となるなど数値演算、グラフィックスといったより高度な処理が可能になる。我々は HDL 設計の特徴と手順を追っていきながら、VLSI 設計への理解を高めていくと共に、実際に Verilog-HDL による記述を行ってハードウェアの動作の流れを理解し、FPU のコアを設計することを具体的な研究とした。

２　浮動小数点プロセッサの設計

本章では、浮動小数点の説明、仕様設計、データパス設計、verilogHDL による記述を行っていく。

2.1　浮動小数点とは

　符号なし整数および符号付き整数の他に、プログラミング言語では小数を含む数値を取り扱う。そのような数値を実数（real）形式という。例えば以下のようなものである。

３．１４１９５２６５...

１．０×１０$^{-9}$

特に２つ目の例に使われている表記法は科学表記法と呼ばれ、コンピュータで扱うときには、浮動小数点形式と呼ばれる。通常、浮動小数点形式の数を書き表す場合、小数点の左側には数字を１つしか書かない。一般形は以下のようになる。

$1.\mathrm{xxxxxxxxxx}_2 \times 2^{\mathrm{yyy}}$

2.1.1　３２ビット浮動小数点数の表現形式

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	……	1	0
s	指数								仮数									
１ビット	８ビット								２３ビット									

表２．１　IEEE754 での浮動小数点数の表現形式（単精度）

- S は浮動小数点数の符号を表す（１は負を意味する)。
- 指数は８ビットの指数フィールドを表す（指数の符号を含む)。・・・(*1)
- 仮数は 23 ビットの小数を表す。

（＊）この表現形式は数値と符号が別々に保持されているため、符号付き絶対値表現と呼ばれる。

（*1）最も小さな負の指数を 00000000 と表し，最も大きい正の指数を 1111111 と表すゲタばき表現をとる。（IEEE754 では単精度用のゲタに 127 を使用。）
－１ ⇒ －１＋１２７(10)＝１２６(10) ⇒ ０１１１１１１０(2)
＋１ ⇒ ＋１＋１２７(10)＝１２８(10) ⇒ １００００００００(2)
以上のように指数にゲタをはかせた場合、浮動小数点数の実際の値は下記の式によって表されることになる。

$$(-1)^{s} \times (1+\text{仮数}) \times 2^{(\text{指数}-\text{ゲタ})}$$

2.1.2　３２ビット浮動小数点プロセッサの特長

本３２ビット浮動小数点プロセッサは IEEE754 の単精度（基本形式）の規格に準拠した高速浮動小数点プロセッサである。このプロセッサの特長をまとめると次の通りである。

（a）冗長２進アルゴリズムの採用（李研究）

冗長 2 進表現を用いた乗算器、および除算器により、LSI 化に適した演算器を実現している。

（b）独立したハードウェア加減算器、乗算器、および除算器

加減算、乗算、および除算をそれぞれの専用ハードウェアによりサポートすることにより、加減算、乗算、および除算を並列に実行でき、浮動小数点プロセッサとして高速演算を可能としている。また、冗長２進型除算器を用いればチップ面積を有効に利用できるが、独立に除算器を設けることにより簡単な制御機構により浮動小数点プロセッサを構成することが出来る。

（c）独立バスの採用

データ入力、データ入出力バスを各々独立して設けた。これにより、プロセッサとして高性能化を図ることが出来る。

（d）３２ビット×３２ワードのマルチポートレジスタの採用

図２．１にマルチポートレジスタのブロック図を示す。

加減算器、乗算器、および除算器の三つの演算器を読み出し４ポート、書き込み３ポート計７ポートのレジスタにより結合し、データ入出力、および演算器の並列動作を可能としている。このことにより多様な計算を行う場合でも演算器の性能を常に最高性能に近い状態で実行できる。各ポートで使用するレジスタのアドレスは、図２．２に示す命令コード中のアドレス指定領域で指定する。なお、アドレス E は、E(R)ポート（読み出しポート）と E（W）ポート（書き込みポート）の指定を兼ねている。

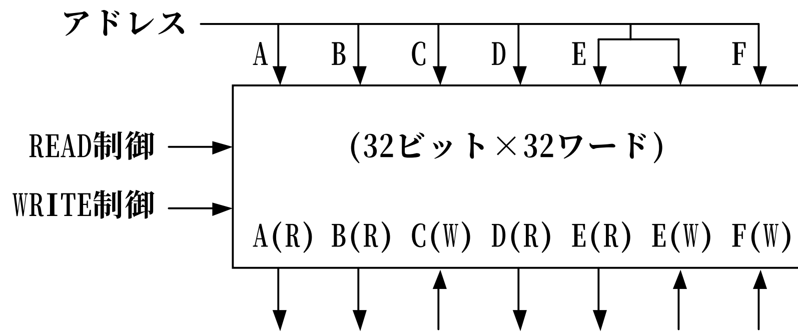


図 2.1　マルチポートレジスタ

2.2　仕様設計

仕様設計はアーキテクチャ設計、インプリメンテーション設計、および外部インタフェース設計の３つからなる。

アーキテクチャ設計とはレジスタセットや命令セットなどを決定することである。外部インタフェース設計とは周辺ＬＳＩとの通信のためのデータやアドレスや制御信号を定義し、信号のやり取りの順番やタイミングを定義することである。

ここでは、アーキテクチャ設計と外部インタフェース設計を行っていく。インプリメンテーションについては、パイプラインやキャッシュなどの技法を用いなかったため省略する。

2.2.1　アーキテクチャ設計

（1） レジスタセット

レジスタセットとは使用するレジスタの仕様を決めることである。以下にＦＰＵのレジスタの仕様を示す。

- ＦＰＵのレジスタは読み出しに 4 ポート、書き込みに 3 ポートを持つレジスタとする。以下これをマルチポートレジスタと呼ぶ。
- マルチポートレジスタは 32bit×32words とする。
- レジスタの 0 番地は 0 レジスタとする。

（2） 命令セット

命令セットとは個々のマイクロプロセッサにおける命令の集合である。以下にＦＰＵの命令セットの特徴を示す。また、アセンブリ言語と機械語を用いて各命令の説明を順にしていく。

（ｉ）命令セットの特徴

- FPU の命令はすべて 41 ビットである。
- FPU の命令の種類は、加算（add）と Acc 加算、減算（subtract）と Acc 減算、乗算（multiply）と Acc 乗算、除算の開始（divide start)、除算の終了（divide end)、メモリとのデータ転送のための命令ロード（load word)、ストア（store word）とする。

（ii）命令形式（instruction format）

本浮動小数点プロセッサは高速動作を実現するために、マイクロコードで演算ユニットを制御する CISC(Complex Instruction Set Computer)型ではなく、RISC(Reduced Instruction Set Computer)型を採用している。命令長は４１ビットの水平型の命令形式で、互いに独立したフィールドで構成している。下図に命令フォーマットを示す。図に示すように、浮動小数点レジスタ（FREG）用の６種の５ビット・アドレスフィールド、３種の演算器、および入出力に対応する独立したフィールドを持つ。

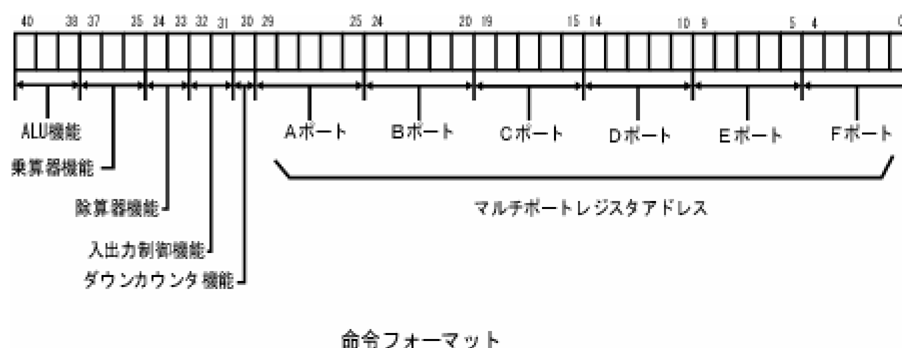


図 2.2　　命令フォーマット図

ALU 機能		
ニモニック	コーディング	機能
ADD	１００	加算
SUB	１０１	減算

除算器機能		
ニモニック	コーディング	機能
DIVS	０１	除算の開始
DIVE	１１	除算の終了

乗算器機能		
ニモニック	コーディング	機能
MUL	１００	乗算

入出力制御機能		
ニモニック	コーディング	機能
IN	０１	データの入力
OUT	１１	データの出力

表 2.1　命令フォーマットに対応する各演算のコーディング

2.2.2　外部インタフェース設計

FPU と通信を行うものとしては命令が格納された命令メモリとデータが格納されたデータメモリがある。以下に FPU とメモリとのインタフェースを図 2.3 に、入出力信号の概要を表 2.2 に示す。

（1）インタフェース

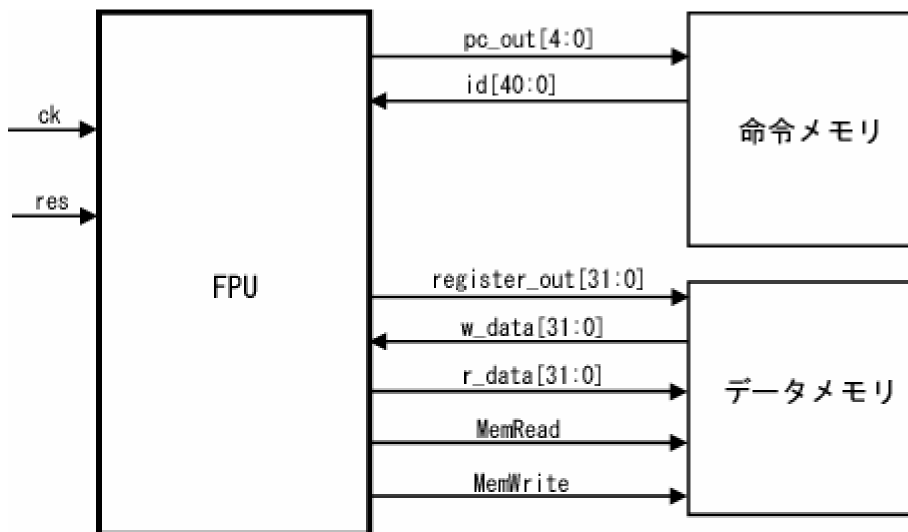


図 2.3　ＦＰＵ/メモリインタフェース

（2）入出力信号

入出力	ビット幅	信号名	説明
input	1	ck	クロック信号
input	1	res	リセット信号（1 のときリセット）
input	41	id	命令メモリからのデータバス
input	32	w_data	データメモリからのデータバス
output	5	pc_out	プログラムカウンタのアドレスバス
output	5	register_out	データメモリへのアドレスバス
output	32	r_data	データメモリへの読み出しデータバス
output	1	MemRead	メモリから読み込む時 1
output	1	MemWrite	メモリに書き込む時 1

表 2.2　ＦＰＵの入出力信号の概要

2.3　データパス設計

データパスを設計するにあたって、まずは使用する各命令を実行するのに必要な構成要素を洗い出し、それらの要素に基づいて各命令タイプ別にデータパスを設計していく。最後にそれらを結合して全体のデータパスを作ることにする。

2.3.1　各データパスの設計

各データパスごとに構成要素、データの流れを順に見ていき、それらをもとにデータパスを図 2.4～図 2.9 に示す。

（1）命令フェッチ

（ｉ）構成要素

命令メモリ	プログラムの命令を格納する場所。入力された命令アドレスによって命令を出力する。
プログラムカウンタ	必要な命令を取り出す為のアドレスを保持する。

（ii）データの流れ

① プログラムカウンタの値を命令メモリに送る
② プログラムカウンタの値が指すアドレスにある命令を出力する。
③ 次の命令を実行するためにプログラムカウンタの値を１繰り上げる。

（iii）データパス

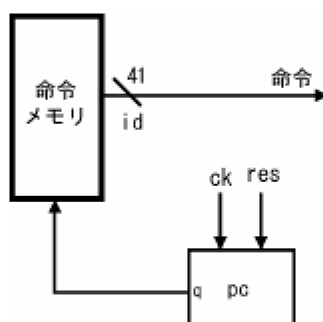


図 2.4　命令フェッチにおけるデータパス

（２）加算（add)、減算（sub）命令

（ｉ）構成要素

マルチポートレジスタ （ＭＰＲ）	加減算命令ではレジスタを 3 ポート使用する。 2 ポートはデータを読み出し、残りはデータを書き込む。読み出しは読み出し制御信号 RegRead で、書き込みは書き込み制御信号 RegWrite で制御する。
加減算器 （ＦＡＵ）	ＭＰＲから読み出したデータを対象に演算を行う。どちらの演算をするかは 1 ビットの制御信号を用いて行う。

（ii）データの流れ

① 命令が命令メモリからフェッチされ、PC が繰り上げられる
② 読み出しレジスタＤポートとＥポートで指定された２つのデータをレジスタから読み出させる
③ 読み出されたデータが加減算器に入力され、制御信号 FAUin=0 のとき加算を、FAUin=1 のとき減算を行う
④ Regwrite=1 のとき、結果をＦポートが指定するアドレスに書き込む

（iii）データパス

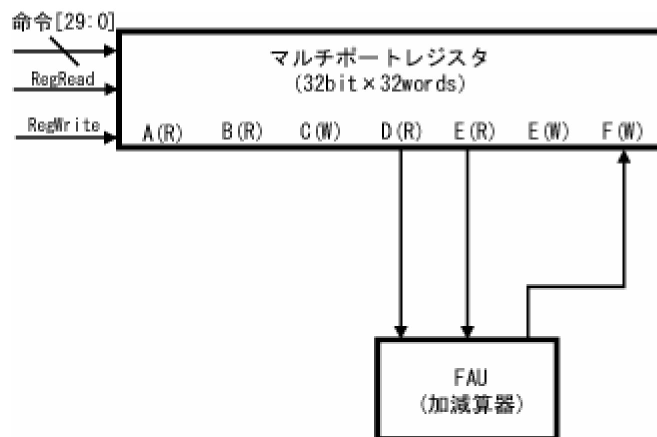


図 2.5　加減算命令のデータパス

（3）乗算（mul）命令

（ｉ）構成要素

マルチポートレジスタ（MPR）	乗算命令ではレジスタを 3 ポート使用する。2 ポートはデータを読み出し、残りはデータを書き込む。読み出しは読み出し制御信号 RegRead で、書き込みは書き込み制御信号 RegWrite で制御する。
乗算器（FMUL）	MPRから読み出したデータを対象に演算を行う。

（ii）データの流れ

① 命令が命令メモリからフェッチされ、PC が繰り上げられる
② 読み出しレジスタＡポートとＢポートで指定された２つのデータをレジスタから読み出させる
③ 読み出されたデータが乗算器に入力され、乗算を行う
④ Regwrite=1 のとき、結果をＣポートが指定するアドレスに書き込む、またはアキュムレータ（ACC）の制御信号 ACCin=1 のとき、ACC に書き込む

（iii）データパス

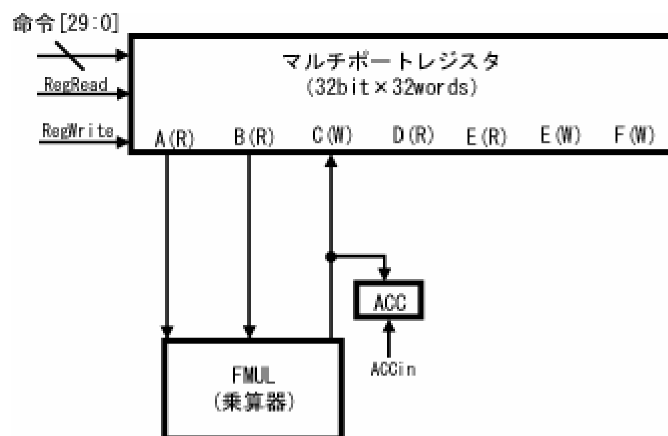


図 2.6　乗算命令のデータパス

（4）除算の開始（divs）命令

（ｉ）構成要素

マルチポートレジスタ （ＭＰＲ）	除算の開始命令ではレジスタを単独命令では2 ポート、乗算命令と並行する場合は 4 ポート使用する。 2 ポート（4 ポート）はデータを読み出す。 読み出しは読み出し制御信号 RegRead で制御する。
除算器 （ＦＤＩＶ）	ＭＰＲから読み出したデータを対象に演算を行う。

（ii）データの流れ

① 命令が命令メモリからフェッチされ、PC が繰り上げられる
② 読み出しレジスタＡポートとＢポート（乗算命令が並行する場合はＤポートとＥポート）で指定された２つのデータをレジスタから読み出させる
③ 読み出されたデータが除算器に入力され、除算を行う

（iii）データパス

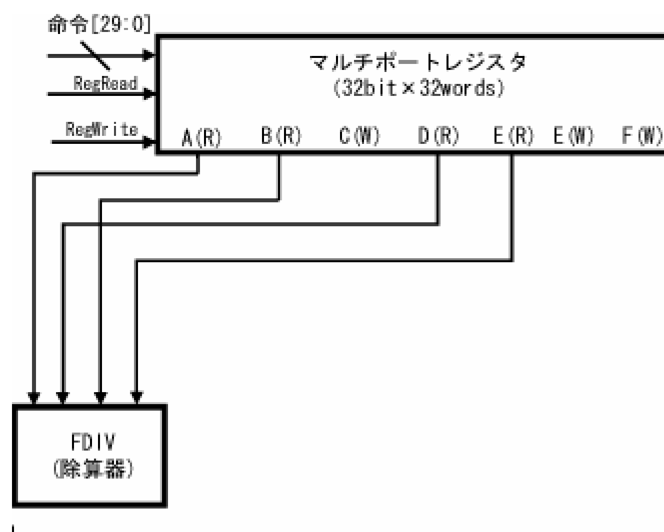


図 2.7　除算の開始命令のデータパス

（5）除算の終了（dive）命令

（ｉ）構成要素

マルチポートレジスタ（ＭＰＲ）	除算の終了命令では並行する命令によってレジスタをＣポートかＦポートを使い分ける。書き込みは書き込み制御信号 RegWrite で制御する。
除算器（ＦＤＩＶ）	入力される値に対して除算を行い、演算結果を出力する。

（ii）データの流れ

（乗算命令がある場合）
① 除算の結果がＦポートの指定されたアドレスに書き込まれる
（上記以外）
① 除算の結果がＣポートの指定されたアドレスに書き込まれる

（iii）データパス

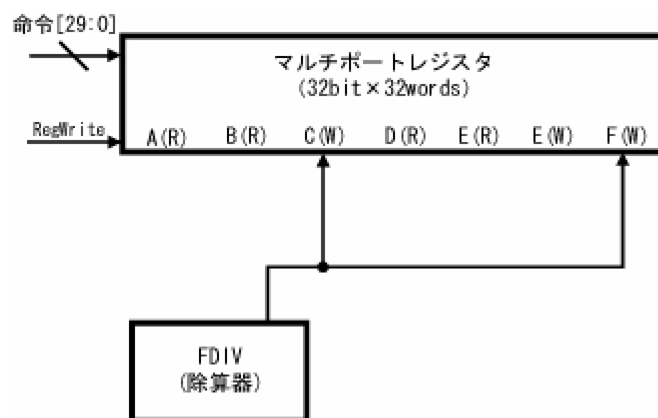


図 2.8　除算の終了命令のデータパス

（6）ロード／ストア命令（lw,sw）

（ｉ）構成要素

マルチポートレジスタ（ＭＰＲ）	ロード命令ではレジスタを並行する命令によってＣポートかＥポートを使い分け、ストア命令ではレジスタをＢポートかＤポートを使い分ける。ロードのときはロードした値をレジスタに書き込み、ストアのときはストアする値をレジスタから読み出す。書き込み、読み出しはそれぞれ制御信号 RegWrite,RegRead で制御する。
データメモリ	データが格納されているメモリ。ロードのときは指定されたアドレスによってメモリからデータが読みだされ、ストアのときは指定されたアドレスによって、メモリへデータを書き込む。書き込み、読み込みはそれぞれ MemWrite,MemRead 信号によって制御される。

（ii）データの流れ

（ロード命令）

① 命令が命令メモリからフェッチされ、PC が繰り上げられる
② レジスタからメモリアドレスを算出
③ Memread=1 のとき、メモリアドレスが指すデータを読み出す
④ Regwrite=1 のとき、メモリアドレスが指すデータを書き込みレジスタで指定されたレジスタに書き込む

（ストア命令）

① 命令が命令メモリからフェッチされ、PC が繰り上げられる
② レジスタからメモリアドレスを算出
③ Memwrite=1 のとき、メモリアドレスが指す場所にストアするデータを書き込む

（iii）データパス

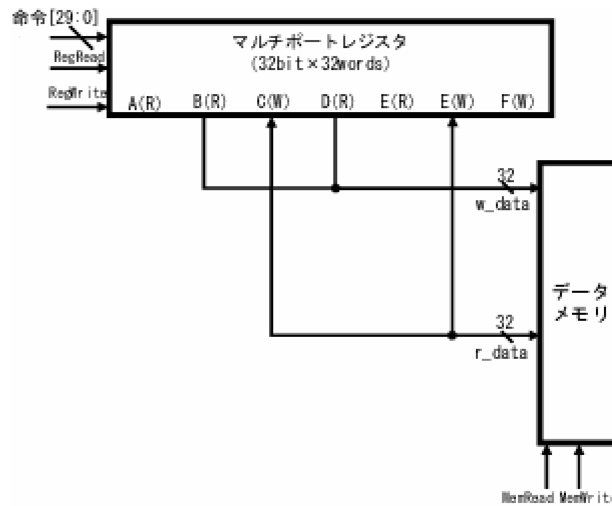


図 2.9　ロード/ストア命令のデータパス

2.3.2　データパス全体部

以上で各命令別にデータパス設計を行ってきた。次に各データパスを合わせて、データパス全体を構成する。ここで個々のデータパスを組み合わせるにあたって、共有できる要素は共有する必要がある。共有するためには、複数入力を接続して、その中から制御信号によって１つの入力を選択できるようにすればよい。この操作はマルチプレクサ、3 to１セレクタと呼ばれる回路を用いて行われる。

（１）共有可能部分

- 除算器に入力されるデータは除算の開始命令の場合、ＡポートとＢポートから出されたデータであり、除算の開始命令と乗算命令が並行する場合、ＤポートとＥポートから出されたデータである。
- 加減算器に入力されるデータは加減算の命令の場合、Ｄポートから出されたデータであり、アキュムレータ（ＡＣＣ）データとの加算命令の場合は、ＡＣＣから出されたデータである。

- Ｆポートに書き込まれるデータは、加減算命令の場合は加減算の結果で、除算の終了命令の場合は、除算の結果である。
- データメモリに書き込まれるデータは、加算命令などが並行する場合Ｄポートが使えないので、Ｂポートの読み出しデータで、乗算命令が並行する場合Ｂポートが使えないので、Ｄポートの読み出しデータである。

- Ｃポートに書き込まれるデータは、乗算命令の場合は乗算の結果で、除算の終了命令の場合は除算の結果で、ロード命令の場合はロードするデータである。

- E ポートに書き込まれるデータは、RegWrite=1 のときロード命令がないにもかかわらずロードしたデータに書き換わってしまうので、MemRead=0 の場合はＥポートのデータで、MemRead=1 の場合はロードするデータになるようにマルチプレクサを用いる。

（２）共有後のデータパス

共有可能部分はマルチプレクサや３to１セレクタを使って回路を共有できる。５つのマルチプレクサ（ＭＵＸ）ではそれぞれの選択制御信号によって 2 つのデータから 1 つのデータを選択する。3 to１セレクタでは選択制御信号によって、3 つのデータから 1 つのデータを選択する。各選択制御信号をそれぞれ、AorD,BorE,FAUin,Fin,Dmin,Cin,Ein とする。

以上で今回使用するすべての命令を満たすデータパスが設計できた。そのデータパスを図に示し、データパス設計を終了する。

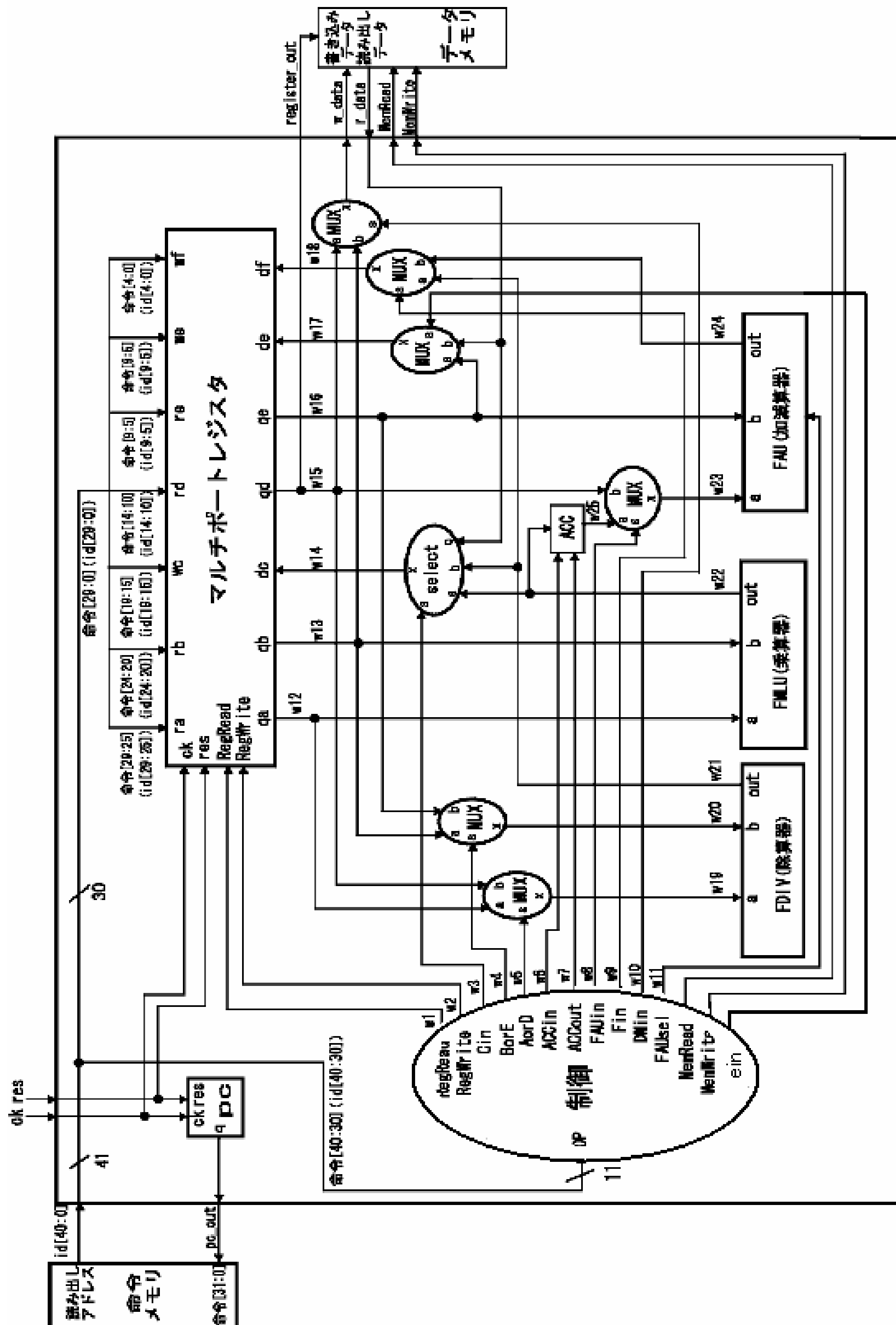


図 2.10　データパス全体図

2.4 VerilogHDL による記述

以上でデータパス設計、制御回路設計を行ってきた。これらを用いれば FPU の RTL 回路図を作成できる。HDL 記述の特徴は回路の動作そのまま表した回路の記述（動作記述）によってゲートレベルを意識することなく容易に記述することができる。また上位の階層と下位の回路のインスタンスを接続して回路を記述（構造記述）することによって階層構造を表現することができるということであった。ここでは RTL 回路を VerilogHDL で記述する。まず RTL 回路図をもとに各部品をそれぞれ VerilogHDL で記述する。次に上位階層と接続することで RTL 回路全体を VerilogHDL で記述していく。

2.4.1 各部品の HDL 記述

各部品における機能、シンボル図、および使用する信号を明らかにし、それらをもとに VerilogHDL で記述する。各シンボル図は図 2.11～図 2.14 に、使用する信号の機能は表 2.3～表 2.6 に示す。

（１）レジスタ

(i) 機能

・同期リセット付き 32 ビット×32 のレジスタファイルである。
・0 番地には常に 0 の値が保持されているゼロレジスタとする。
- 書き込みの時、クロックの立ち上がりエッジに同期して、指定された番地に書き込む。書き込みポートは 3 つ。
- 読み出しの時、指定された番地の値を読み出す。読み出しポートは 4 つ。

（ii）シンボル図と各信号の機能

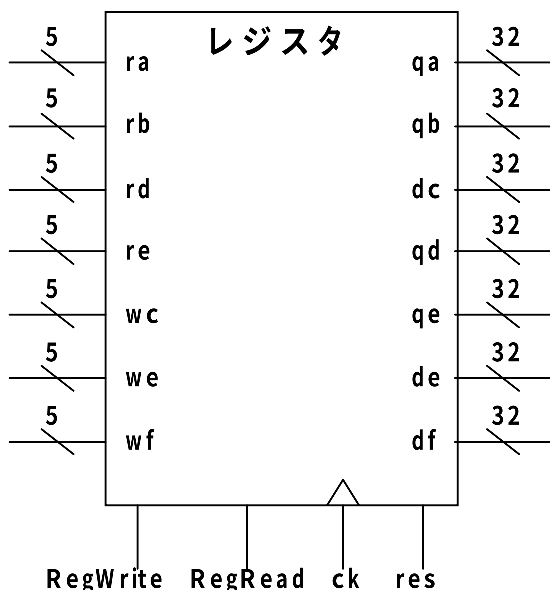


図 2.11 レジスタのシンボル図

信号	機能	補足
ck	クロック信号	
res	リセット信号	クロックの立ち上がりに同期して値が 0 の時 32 個のレジスタのデータを 0 とする
ra,rb,rd,re	読み出しレジスタ	
qa,qb,qd,qe	読み出しデータ	
wc,we,wf	書き込みレジスタ	
dc,de,df	書き込みデータ	
RegRead	読み出し制御信号	1 の時、読み出しレジスタで指定する番地のデータを読み出す
RegWrite	書き込み制御信号	1 の時、書き込みレジスタで指定する番地にデータを書き込む

表 2.3　レジスタで使用する信号の機能

（iii）ＨＤＬ記述

/*レジスタの HDL 記述*/

```
module   multiport_register(dc,de,df,ra,rb,rd,re,wc,we,wf,
                             RegRead,RegWrite,ck,res,qa,qb,qd,qe);
   input [31:0]   dc,de,df;
   input [4:0]    wc,we,wf;
   input [4:0]    ra,rb,rd,re;
   input              ck,res,RegRead,RegWrite;
   output[31:0]   qa,qb,qd,qe;
   reg     [31:0]   regf[0:31];
   integer          i;


   function [4:0]   read;                        //読み出しアドレス関数
          input            RegRead;
          input [4:0]   RegAddress;

          if(RegRead==1'b1)
               read=RegAddress;
          else if(RegRead==1'b0)
               read=5'bxxxxx;
   endfunction


   always   @(posedge ck)begin
           if(res==1'b1)
                for(i=0; i<32; i=i+1)
                     regf[i]<=32'h00000000;

           else if(RegWrite==1'b1)begin        //書き込み
                        if(wc!=5'b0)
                             regf[wc]<=dc;
                        if(we!=5'b0 && we!=wc)
                             regf[we]<=de;
```

```
                    if(wf!=5'b0 && wf!=wc && wf!=we)
                        regf[wf]<=df;
            end
        end


    assign qa=regf[read(RegRead,ra)];        //読み出し
    assign qb=regf[read(RegRead,rb)];
    assign qd=regf[read(RegRead,rd)];
    assign qe=regf[read(RegRead,re)];

endmodule
```

（２）マルチプレクサ

(i) 機能

- 制御信号ｓが０であればａを選択し、ｓが１であればｂを選択するマルチプレクサである。

(ii) シンボル図と各信号の機能

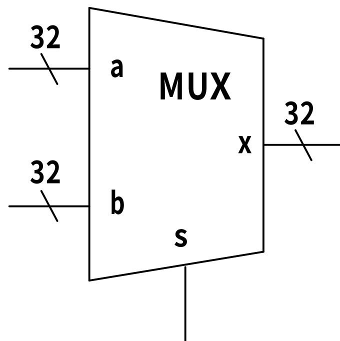


図 2.12　マルチプレクサのシンボル図

信号	機能	補足
a,b	32 ビットの入力	
x	32 ビットの出力	
s	制御信号	a か b かを選択する

表 2.4　マルチプレクサで使用する各信号の機

(iii) ＨＤＬ記述

```
/*マルチプレクサのＨＤＬ記述*/

module   mux32(a,b,s,x);

input   [31:0]     a,b;
input              s;
output [31:0]     x;
```

```
reg       [31:0]     x;

   always   @(a or b or s) begin
      if(s==1'b0)
         x <= a;
      else
         x <= b;
    end
endmodule
```

（３）セレクタ

(i) 機能

- 制御信号ｓが００であればａを，０１であればｂを，１０であればｃを選択するマルチプレクサである。

(ii) シンボル図と各信号の機能

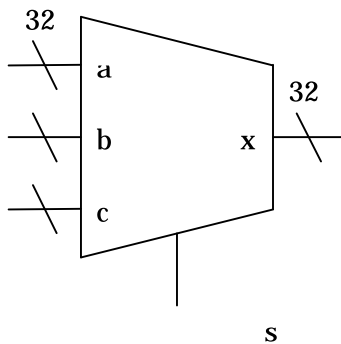


図 2.13　セレクタのシンボル図

信号	機能	補足
a,b,c	32 ビットの入力	
x	32 ビットの出力	
s	制御信号	a か b か c を選択する

表 2.5　セレクタで使用する信号の機能

（iii）ＨＤＬ記述

/*セレクタのＨＤＬ記述*/

```
module sel_3to1(a,b,c,Cin,x);
    input   [31:0]     a,b,c;
    input   [1:0]      Cin;
    output [31:0]     x;

  assign x = (Cin==2'b00)? a:
                 (Cin==2'b01)? b:
                 (Cin==2'b10)? c: 32'hxxxxxxxx;
endmodule
```

（４）プログラムカウンタ

（i）機能

- 同期リセットの４１ビットカウンタである。
- クロックの立ち上がりに同期して、１カウントアップする。

（ii）シンボル図と各信号の機能

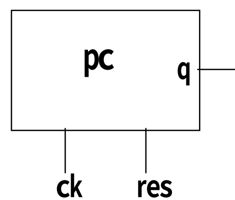


図 2.14　プログラムカウンタのシンボル図

信号	機能	補足
q	41 ビットの出力	
ck	クロック信号	
res	リセット信号	クロックの立ち上がりに同期して値が１の時 プログラムカウンタの値を０にリッセトする

表 2.6　プログラムカウンタで使われる各信号の機能

（iii）HDL 記述

```
/*プログラムカウンタのＨＤＬ記述*/
module    pc(ck,res,q);
    input              ck,res;
    output    [40:0]    q;
    reg       [40:0]    q;

    always @(posedge ck)begin
      if(res==1'b1)                         /*リセット*/
          q <= 41'b00000000000000000000000000000000000000000;
      else                                  /*PC の基本動作*/
          q <= q + 41'b00000000000000000000000000000000000000001;
    end
endmodule
```

2.5　FPU 全体の HDL 記述

以上で各部品を Verilog-HDL で記述することができた。最後に先ほどに記述した各部品を上位階層の“ＦＰＵ”と接続する。上位階層回路を Verilog-HDL で記述し、ＨＤＬ記述の完成とする。なおこの記述の信号などは図 3.14 示す。

```
/*ＦＰＵのＨＤＬ記述*/

module FPU(ck,res,pc_out,id,register_out,w_data,r_data,MemRead,MemWrite);

    input              ck,res;
    input    [40:0]    id;
    input    [31:0]    r_data;
    output   [31:0]    w_data,register_out;
    output   [40:0]    pc_out;
    output             MemRead,MemWrite;

    wire               w1,w2,w4,w5,w6,w7,w8,w9,w10,w11;
    wire     [1:0]     w3;
    wire     [31:0]    w12,w13,w14,w15,w16,w17,w18,w19,w20,w21,w22,w23,w24,w25;

    assign   w15 = register_out;

    fpu_ctrl_unit i0(.op(id[40:30]),.RegRead(w1),.RegWrite(w2),.Cin(w3)
                     ,.BorE(w4),.AorD(w5),.ACCin(w6),.ACCout(w7),.FAUin(w8)
                     ,.Fin(w9),.DMin(w10),.FAUsel(w11)
                     ,.MemWrite(MemWrite),.MemRead(MemRead));

    multiport_register i1(.ck(ck),.res(res),.ra(id[29:25]),.rb(id[24:20]),.rd(id[14:10]),
                          .re(id[9:5]),.wc(id[19:15]),.we(id[9:5]),.wf(id[4:0]),
                          .qa(w12),.qb(w13),.qd(register_out),.qe(w16),.dc(w14),
                          .de(w17),.df(w18),.RegRead(w1),.RegWrite(w2));

    fau i2(.a(w23),.b(w16),.out(w24),.FAUsel(w11));

    divide i3(.a(w19),.b(w20),.out(w21));
```

```
    fmulti32 i4(.a(w12),.b(w13),.out(w22));

    pc i5(.ck(ck),.res(res),.q(pc_out));

    mux32 i6(.a(w12),.b(w15),.s(w5),.x(w19));

    mux32 i7(.a(w13),.b(w16),.s(w4),.x(w20));

    mux32 i8(.a(w25),.b(w15),.s(w8),.x(w23));

    mux32 i9(.a(w21),.b(w24),.s(w9),.x(w18));

    mux32 i10(.a(w15),.b(w13),.s(w10),.x(w_data));

    mux32 i13(.a(w16),.b(r_data),.s(MemRead),.x(w17));

    sel_3to1 i11(.a(w22),.b(w21),.c(r_data),.s(w3),.x(w14));

    acc i12(.a(w22),.ACCin(w6),.ACCout(w7),.ck(ck),.res(res),.q(w25));

endmodule
```

3　浮動小数点プロセッサの再構築

私たちは昨年の研究で未完成だった３２ビット浮動小数点プロセッサの再構築を目標に研究した。研究内容は未完成部分だった加減算器、制御の訂正。新たにアキュムレータを付け加えること。まずここでは簡単に、未完成部分だったところの説明、新たに付け加えた目的を述べ、次から詳しく説明する。

加減算器・・・・・・減算が正常にできていなかった。例えば５－（－７）＝１２にならないといけないところが、－２になっていた。

アキュムレータ・・・アキュムレータは従来の浮動小数点プロセッサには付いていなかった。効率よく演算を行えるようにするために作った。

制御・・・・・・・・足りなかった制御信号の追加。間違っていた制御のＨＤＬ記述の手直し。アキュムレータを付け加えた事による制御の変更。

除算器・・・・・・・除算の開始、終了命令が正しく動作できない。
（これについてはまだ研究中である）

3.1　加減算器

従来の加減算器は減算が正常にできていなかった。例えば５－（－７）＝１２にならなければならないところが、－２になった。それはアルゴリズムが間違っていたために起きていたので、それを変更しＨＤＬ記述を書き直し正常に減算できるようにした。
詳細は増田、中薗の卒業論文に書かれている。

3.2　アキュムレータ

アキュムレータとは、内部に ALU を備えた特別なレジスタです。このレジスタには、単にデータを記憶するだけでなく、入力データと現在記憶しているデータとの間で演算を行うことができます。アキュムレータは、別名、Ａレジスタと呼ばれることもある。Ａレジスタは古い呼び名で、初期のプログラム内臓方式のコンピュータにおいてはハードウェアは貴重であった。したがって、コンピュータの開拓者たちは、今日のマシンのようにレジスタをいくつも備えることはできなかった。実際、当時のマシンは演算用意のレジスタを１つしか備えていなかった。すべての演算結果は単一のレジスタに累積（accumulate）されるので、このレジスタはアキュムレータ(accumulator)又はＡレジスタと呼ばれた。

3.2.1　本研究のアキュムレータの概略

本研究で使用するアキュムレータはＦＭＵＬ（乗算器）の演算結果をアキュムレータに格納し、その格納された値をＦＡＵ（加減算器）に送り加減算をすることができる。つまり乗算結果をマルチポートレジスタを通さずに加減算器に送ることができるのである。

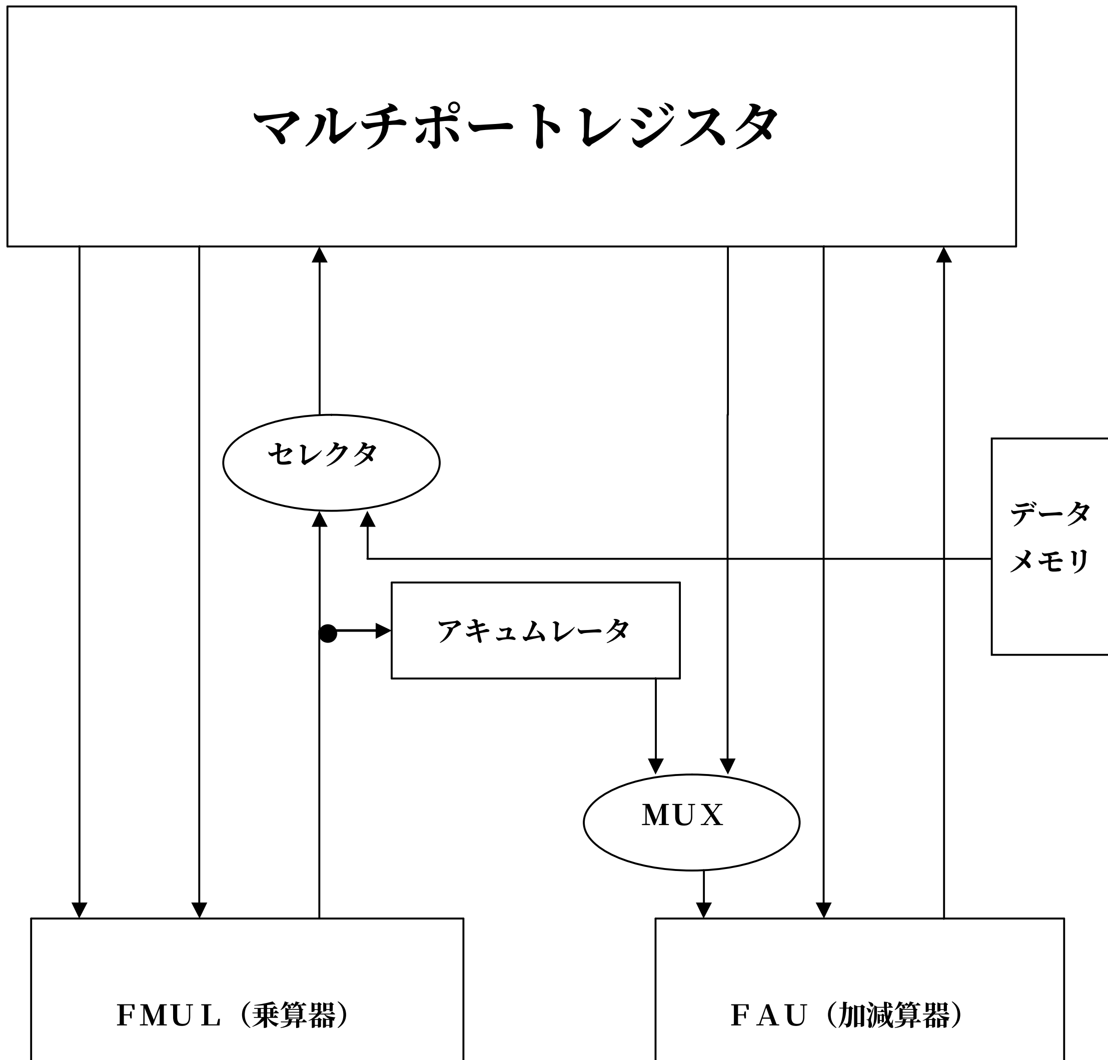


図 3.1　アキュムレータ周辺の概略図

3.2.2　アキュムレータの有無の比較

積和演算の流れ

・アキュムレータを付ける前（乗算の演算結果をマルチポートレジスタを通して加減算する場合）

ⅰ.乗算の演算結果をマルチポートレジスタの何番地に保存するかを指定して、保存させる。

ⅱ.さきほどの乗算の演算結果が保存されている番地を指定して加減算する。

・アキュムレータを付けた後（乗算の演算結果をアキュムレータを通して加減算する場合）

ⅰ.ACCin=1 にして乗算を行う。乗算の演算結果はアキュムレータに書き込まれる。

ⅱ.ACCout=1 にして加減算を行う。乗算の演算結果がアキュムレータからロードされ加減算に使用される。

3.2.3　アキュムレータのメリット

ⅰ.乗算の値をアキュムレータにいれることができるので、Ｃポートへのロードと乗算が一度にできる。

ⅱ.マトリクス演算を行うときは、アキュムレータを使用すると乗算と加算とロードを一度にできるので時間を短縮することができる。

3.2.4　アキュムレータの機能、シンボル図、ＨＤＬ記述

（i）機能

・FMUL（乗算器）の演算結果を格納する。格納できる値は 1 つとする。
・命令制御ユニットにより値の格納､出力を制御する。

（ii）シンボル図と各信号の機能

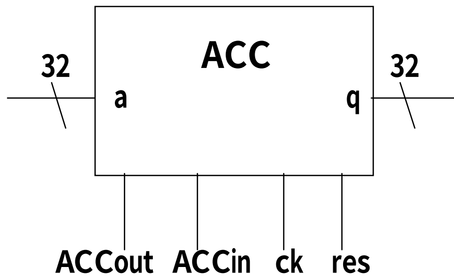


図3.2　ACCのシンボル図

信号	機能	補足
a	32 ビットの入力	
q	32 ビットの出力	
ck	クロック信号	
res	リセット信号	クロックの立ち上がりに同期して値が１の時２個のレジスタのデータを０とする
ACCin	書き込み制御信号	１の時書き込み
ACCout	読み出し制御信号	１の時読み出し

表 4.1　ACC で使用する各信号の機能

（iii）ＨＤＬ記述

```
/*ＡＣＣのＨＤＬ記述*/

module acc(a,ACCin,ACCout,ck,res,q);

    input   [31:0]    a;
    input             ACCin,ACCout,ck,res;
    output  [31:0]    q;
    reg     [31:0]    regf[0:1];        //値を格納できるレジスタファイルは１つとする
    integer           i;
```

```
    function [31:0]   ACCread;       //読み出し関数
      input           ACCout;
      input   [31:0] regfdata;

      if(ACCout==1'b1)begin
            ACCread=regfdata;
      end else begin
            ACCread=32'hxxxxxxxx;
      end
    endfunction

    assign q = ACCread(ACCout,regf[1]);        //読み出し

    always @(posedge ck)begin                  //書き込み
     if(res==1'b1)begin
       for(i=0; i<2; i=i+1)
            regf[i]<=32'h00000000;
     end else begin
       if(ACCin==1'b1)begin
            regf[0]=a;
            regf[1]=regf[0];
       end
     end
   end
endmodule
```

3.3　命令制御ユニット

命令制御ユニットの機能

制御回路とはデータパスに制御信号を与えデータの流れや演算の種類を制御する回路であり、データパスを思い通りに動作させるには、適切なタイミングで制御信号を与える必要がある。

ＦＰＵのシュミレーションを繰り返すと、制御回路に間違いがあることが判明した。そこで制御部を再検討し、改良前の制御回路と改良後の制御回路の比較をする。

3.3.1　制御信号の追加

（ｉ）新たに制御信号 Ein を取り付けることにした。

改良前は MemRead という１つの制御信号でメモリからの読み出しとＥポートへの書き込みを同時に制御している。これは回路の複雑化を防ぐのにとても重要なことである。

しかしそれでは加算とロードを並行して計算する時は、加算で使用するデータが勝手に別のデータに変わってしまう。（加算とロードはそれぞれＥポートを使用するため。）

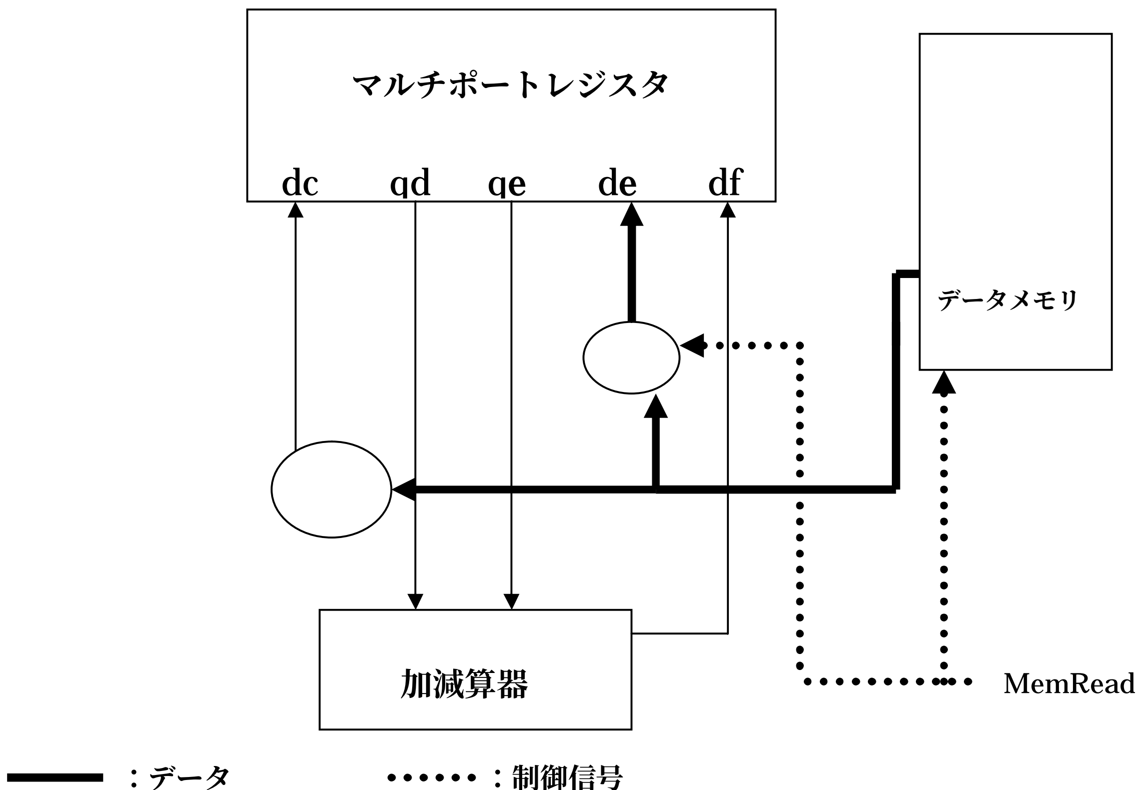


図 3.3　Ein 追加前のデータの遷移図

そこで MemRead はデータメモリの読み出しのみを制御するようにして Ein で E ポートへの入力を制御し、加算とロード命令を並行に行う場合は C ポートのみにロードするようにした。これにより加算器とロードを同時に行うときにも、不具合が生じなくなる。

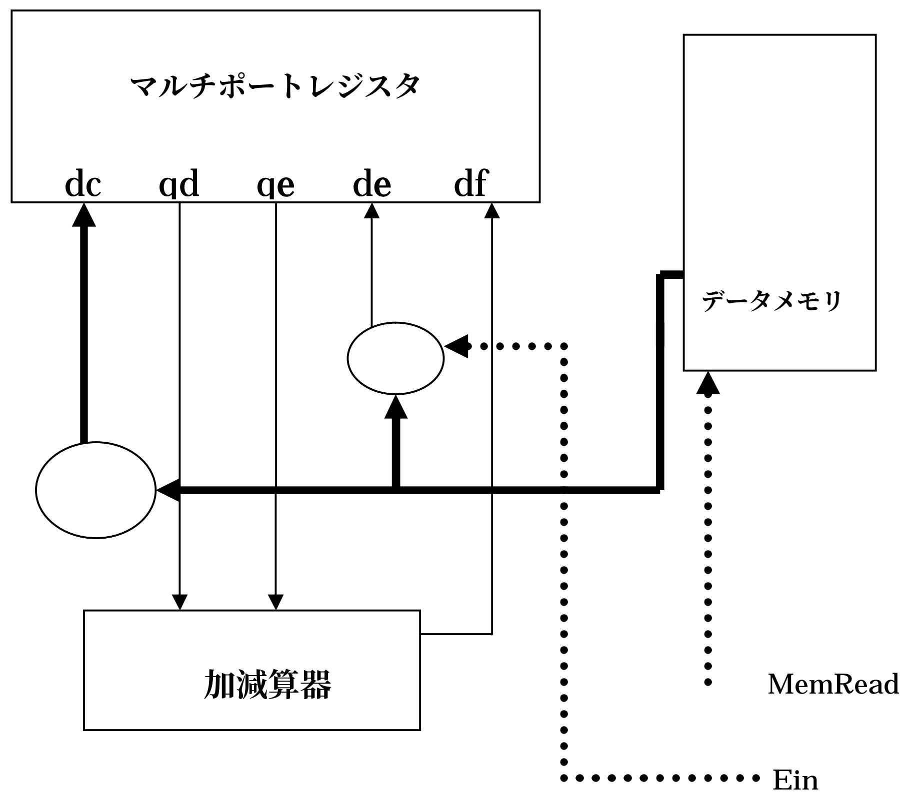


図 3.4　Ein 追加後のデータ遷移図

3.3.2　命令タイプと制御信号の関係

命令タイプと制御信号の関係にも誤りがあったため、改良した。

（ｉ）加減算の制御信号

改良前

命令名 / 制御信号	加算（減算）							
	乗算	除算の開始		除算の終了		ロード	ストア	
		ロード		ストア				
RegRead	1	1	1	1	1	1	1	1
RegWrite	1	1	1	1	1	1	1	1
AorD	x	0	0	x	x	x	x	x
BorE	x	0	0	x	x	x	x	x
Cin	00	10	xx	01	10	10	xx	xx
Fin	1	1	1	1	1	x	1	1
Dmin	x	x	x	1	x	x	1	x
ACCin	x	x	x	x	x	x	x	x
ACCout	x	x	x	x	x	x	x	x
FAUin	1	1	1	1	1	1	1	1
FAUsel	0(1)	0(1)	0(1)	0(1)	0(1)	0(1)	0(1)	0(1)
MemRead	0	1	0	0	0	1	0	0
MemWrite	0	0	0	1	0	0	1	0

表　3.2　加減算命令と制御信号の関係

命令名 ＼ 制御信号	加算　（減算）									
	乗算	ACC 乗算			除算の開始		除算の終了		ロード	ストア	
		ロード	除算の終了		ロード		ストア				
RegRead	1	1	1	1	1	1	1	1	1	1	1
RegWrite	1	1	1	1	1	1	1	1	1	1	1
AorD	x	x	x	x	0	0	x	x	x	x	x
BorE	x	x	x	x	0	0	x	x	x	x	x
Cin	00	10	01	xx	10	xx	01	01	10	xx	xx
Fin	1	1	1	1	1	1	1	1	1	1	1
Dmin	x	x	x	x	x	x	1	x	x	1	x
ACCin	0	1	1	1	0	0	0	0	0	0	0
ACCout	x	x	x	x	x	x	x	x	x	x	x
FAUin	1	1	1	1	1	1	1	1	1	1	1
FAUsel	0(1)	0(1)	0(1)	0(1)	0(1)	0(1)	0(1)	0(1)	0(1)	0(1)	0(1)
Ein	0	0	0	0	0	0	0	0	0	0	0
MemRead	0	1	0	0	1	0	0	0	1	0	0
MemWrite	0	0	0	0	0	0	1	0	0	1	0

表 3.3　改良後の加算命令と制御信号の関係

(1)表 4.1 の Cin の除算の終了が 10 になっている。10 は除算の結果を選択するのではなくデータメモリからのデータを選択なので、除算の結果を選択するように 01 に変更した。

(2)ACC をつけたので加減算と ACC 乗算とロード、加減算と ACC 乗算と乗算の終了、加減算と ACC 乗算を新たに追加した。

（ii）乗算の制御信号

制御信号	乗算						
	除算の開始	除算の終了			ロード	ストア	
		ロード	ストア				
RegRead	1	1	1	1	1	1	1
RegWrite	1	1	1	1	1	1	1
AorD	1	x	x	x	x	x	x
BorE	1	x	x	x	x	x	x
Cin	00	00	00	00	00	00	00
Fin	x	0	0	0	x	x	x
Dmin	x	0	0	x	x	0	x
ACCin	x	x	0	x	x	x	x
ACCout	x	x	x	x	x	x	x
FAUin	x	x	x	x	x	x	x
FAUsel	x	x	x	x	x	x	x
MemRead	0	1	0	0	1	0	0
MemWrite	0	0	1	0	0	1	0

表 3.4　　乗算と制御信号の関係（改良前）

命令名 / 制御信号	乗算						
	除算の開始	除算の終了			ロード	ストア	
		ロード	ストア				
RegRead	1	1	1	1	1	1	1
RegWrite	1	1	1	1	1	1	1
AorD	1	x	x	x	x	x	x
BorE	1	x	x	x	x	x	x
Cin	00	00	00	00	00	00	00
Fin	x	0	0	0	x	x	x
Dmin	x	x	0	x	x	0	x
ACCin	0	0	0	0	0	0	0
ACCout	x	x	x	x	x	x	x
FAUin	x	x	x	x	x	x	x
FAUsel	x	x	x	x	x	x	x
Ein	0	1	x	x	1	x	x
MemRead	0	1	0	0	1	0	0
MemWrite	0	0	1	0	0	1	0

表 3.5　乗算と制御信号の関係　（改良後）

（1）乗算と除算の終了とロードを同時に実行する時、B レジスタ、D レジスタは使用しないため Dmin=x とした。

（2）ACCin の列をすべて０にした。これは x だと１クロック前の制御信号を受け継いでしまうためである。例えば１クロック前に ACCin=1 にして ACC に乗算結果を保存しておきたいのに、x のままだと ACCin=1 が引き継がれて値が書き換えられてしまう不具合が生じるためである。

（iii）除算の制御信号

命令名 / 制御信号	除算の開始			除算の終了			ロード	ストア
	ロード	ストア		ロード	ストア			
RegRead	1	1	1	1	0	0	0	1
RegWrite	1	0	0	1	1	1	1	0
AorD	0	0	0	x	x	x	x	x
BorE	0	0	0	x	x	x	x	x
Cin	xx	00	xx	01	01	01	10	xx
Fin	x	x	x	x	x	x	x	x
Dmin	x	x	x	x	x	x	x	1
ACCin	x	x	x	x	x	x	x	x
ACCout	x	x	x	x	x	x	x	x
FAUin	x	x	x	x	x	x	x	x
FAUsel	x	x	x	x	x	x	x	x
MemRead	1	0	0	1	0	0	1	0
MemWrite	0	1	0	0	1	0	0	1

表 3.6　除算と制御信号の関係　（改良前）

命令名 / 制御信号	除算の開始			除算の終了			ロード	ストア
	ロード	ストア		ロード	ストア			
RegRead	1	1	1	1	1	0	1	1
RegWrite	1	0	0	1	1	1	1	0
AorD	0	0	0	x	x	x	x	x
BorE	0	0	0	x	x	x	x	x
Cin	xx	xx	xx	01	01	01	xx	xx
Fin	x	x	x	x	x	x	x	x
Dmin	x	0	x	x	1	x	x	1
ACCin	0	0	0	0	0	0	0	0
ACCout	x	x	x	x	x	x	x	x
FAUin	x	x	x	x	x	x	x	x
FAUsel	x	x	x	x	x	x	x	x
Ein	1	x	x	1	x	x	1	x
MemRead	1	0	0	1	0	0	1	0
MemWrite	0	1	0	0	1	0	0	1

表 3.7　除算と制御信号の関係　（改良後）

（1）除算の終了とストアを同時に実行する時、ロード命令の時 RegRead=1 にした。

（2）ACCin の列を０にした。理由は（ii）乗算の制御信号と同じである。

命令名 / 制御信号	ＡＣＣ加算　（ＡＣＣ減算）													
	乗算		ACC 乗算				除算の開始			除算の終了		ロード	ストア	
	ストア		除算の終了	ロード	ストア		ロード	ストア		ストア				
RegRead	1	1	1	1	1	1	1	1	1	1	1	1	1	1
RegWrite	1	1	1	1	1	1	1	1	1	1	1	1	1	1
AorD	x	x	x	x	x	x	0	0	0	x	x	x	x	x
BorE	x	x	x	x	x	x	0	0	0	x	x	x	x	x
Cin	00	00	01	10	xx	xx	10	xx	xx	01	01	10	xx	xx
Fin	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Dmin	0	x	x	x	0	x	x	0	x	1	x	x	1	x
ACCin	0	0	1	1	1	1	0	0	0	0	0	0	0	0
ACCout	1	1	1	1	1	1	1	1	1	1	1	1	1	1
FAUin	0	0	0	0	0	0	0	0	0	0	0	0	0	0
FAUsel	0(1)	0(1)	0(1)	0(1)	0(1)	0(1)	0(1)	0(1)	0(1)	0(1)	0(1)	0(1)	0(1)	0(1)
Ein	0	0	0	0	0	0	0	0	0	0	0	0	0	0
MemRead	0	0	0	1	0	0	1	0	0	0	0	1	0	0
MemWrite	1	0	0	0	1	0	0	1	0	1	0	0	1	0

表　3.8　ＡＣＣ加算と制御信号の関係

ACC 加減算を行いながら同時にできるすべての命令を表にした。

3.3.3　命令タイプと op コードの関係比較

命令名	命令操作コード										
	op10	op9	op8	op7	op6	op5	op4	op3	op2	op1	op0
加算	1	0	0	x	x	x	x	x	x	x	x
減算	1	0	1	x	x	x	x	x	x	x	x
乗算	x	x	x	1	0	0	x	x	x	x	x
除算の開始	x	x	x	x	x	x	0	1	x	x	x
ロード	x	x	x	x	x	x	x	x	1	0	x
ストア	x	x	x	x	x	x	x	x	0	1	x

表　3.9　　op コードと命令タイプとの関係　（改良前）

（1）ACC を付け加えたため、命令名に ACC 加算、ACC 減算、ACC 乗算を追加する。ACC 加算、ACC 減算の op コードはそれぞれ加減算器機能 110,111 とおく。

ＦＡＵ機能	
機能	コーディング
加算	１００
減算	１０１
ACC 加算	１１０
ACC 減算	１１１

乗算器機能	
機能	コーディング
乗算	１００
ACC 乗算	００１

命令名	命令操作コード										
	op10	op 9	op 8	op 7	op 6	op 5	op 4	op 3	op 2	op 1	op 0
ACC 加算	1	1	0	x	x	x	x	x	x	x	x
ACC 減算	1	1	1	x	x	x	x	x	x	x	x
加算	1	0	0	x	x	x	x	x	x	x	x
減算	1	0	1	x	x	x	x	x	x	x	x
乗算	x	x	x	1	0	0	x	x	x	x	x
ACC 乗算	x	x	x	0	0	1	x	x	x	x	x
除算の開始	x	x	x	x	x	x	0	1	x	x	x
除算の終了	x	x	x	x	x	x	1	1	x	x	x
ロード	x	x	x	x	x	x	x	x	0	1	x
ストア	x	x	x	x	x	x	x	x	1	0	x

表　3.10　op コードと命令タイプとの関係　（改良後）

3.3.4　制御のＨＤＬ記述

（ｉ）新しい制御ユニットの機能

制御信号 Ein を追加したので１１ビットの入力 Op（命令のオペコード）を１５ビットの出力（各制御信号）とするデコード回路になった。{改良前は出力１４ビット}

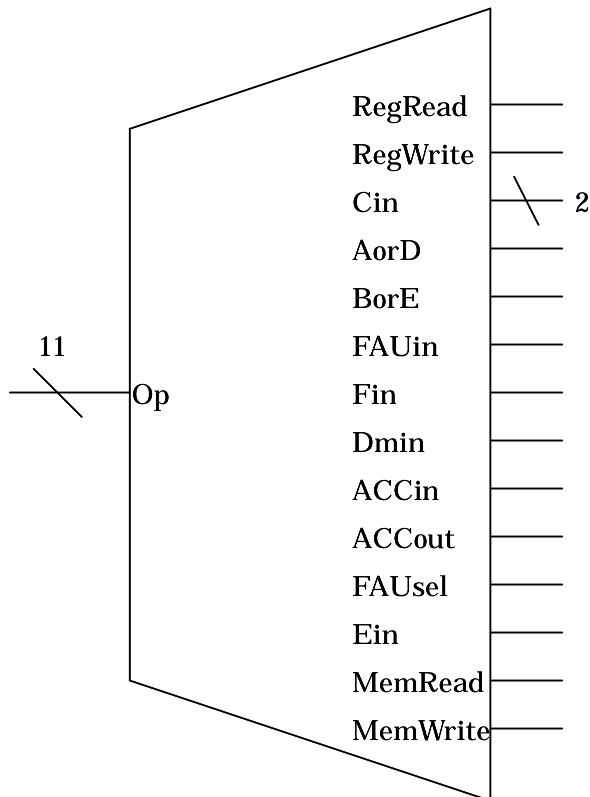


図 3.5　　シンボル図

（ii）ＨＤＬ記述

制御信号の追加（Ein)、制御回路の修正、命令の追加（ACC）したのでそれに伴い、ＨＤＬ記述を改良した。また、記述の間違いを見つけやすくするために case 文を使用した。

```
        /*制御ユニットの HDL 記述*/
module fpu_ctrl_unit(op,RegRead,RegWrite,AorD,BorE,Fin,DMin,
                     ACCin,ACCout,FAUin,FAUsel,Ein,MemRead,MemWrite,Cin);

  input   [10:0] op;
  output [1:0]   Cin;
  output           RegRead,RegWrite,AorD,BorE,Fin,DMin,
                   ACCin,ACCout,FAUin,FAUsel,Ein,MemRead,MemWrite;
  reg      [1:0]   Cin;
  reg               RegRead,RegWrite,AorD,BorE,Fin,DMin,
                   ACCin,ACCout,FAUin,FAUsel,Ein,MemRead,MemWrite;


  always @(op)begin
    case(op)
      11'b10010000000:{RegRead,RegWrite,AorD,BorE,Cin,Fin,DMin,      /*加算・乗算*/
                       ACCin,ACCout,FAUin,FAUsel,Ein,MemRead,MemWrite}
<=15'b11xx001x0x10000;

      11'b10000100010:{RegRead,RegWrite,AorD,BorE,Cin,Fin,DMin,      /*加算・ACC 乗算・ロード*/
                       ACCin,ACCout,FAUin,FAUsel,Ein,MemRead,MemWrite}
<=15'b11xx101x1x10010;

      11'b10000111000:{RegRead,RegWrite,AorD,BorE,Cin,Fin,DMin,      /*加算・ACC 乗算・除算の終了*/
                       ACCin,ACCout,FAUin,FAUsel,Ein,MemRead,MemWrite}
<=15'b11xx011x1x10000;

      11'b10000100010:{RegRead,RegWrite,AorD,BorE,Cin,Fin,DMin,      /*加算・ACC 乗算*/
                       ACCin,ACCout,FAUin,FAUsel,Ein,MemRead,MemWrite}
```

```
<=15'b11xxxx1x1x10000;

        11'b10000001010:{RegRead,RegWrite,AorD,BorE,Cin,Fin,DMin,       /*加算・除算の開始・ロード*/
                         ACCin,ACCout,FAUin,FAUsel,Ein,MemRead,MemWrite}
<=15'b1100101x0x10010;

        11'b10000001000:{RegRead,RegWrite,AorD,BorE,Cin,Fin,DMin,       /*加算・除算の開始*/
                         ACCin,ACCout,FAUin,FAUsel,Ein,MemRead,MemWrite}
<=15'b1100xx1x0x10000;

        11'b10000011100:{RegRead,RegWrite,AorD,BorE,Cin,Fin,DMin,       /*加算・除算の終了・ストア*/
                         ACCin,ACCout,FAUin,FAUsel,Ein,MemRead,MemWrite}
<=15'b11xx01110x10001;

        11'b10000011000:{RegRead,RegWrite,AorD,BorE,Cin,Fin,DMin,       /*加算・除算の終了*/
                         ACCin,ACCout,FAUin,FAUsel,Ein,MemRead,MemWrite}
<=15'b11xx011x0x10000;

        11'b10000000010:{RegRead,RegWrite,AorD,BorE,Cin,Fin,DMin,       /*加算・ロード*/
                         ACCin,ACCout,FAUin,FAUsel,Ein,MemRead,MemWrite}
<=15'b11xx101x0x10010;

        11'b10000000100:{RegRead,RegWrite,AorD,BorE,Cin,Fin,DMin,       /*加算・ストア*/
                         ACCin,ACCout,FAUin,FAUsel,Ein,MemRead,MemWrite}
<=15'b11xxxx110x10001;

        11'b10000000000:{RegRead,RegWrite,AorD,BorE,Cin,Fin,DMin,       /*加算*/
                         ACCin,ACCout,FAUin,FAUsel,Ein,MemRead,MemWrite}
<=15'b11xxxx1x0x10000;

        11'b10110000000:{RegRead,RegWrite,AorD,BorE,Cin,Fin,DMin,       /*減算・乗算*/
                         ACCin,ACCout,FAUin,FAUsel,Ein,MemRead,MemWrite}
<=15'b11xx001x0x11000;

        11'b10100100010:{RegRead,RegWrite,AorD,BorE,Cin,Fin,DMin,       /*減算・ACC 乗算・ロード*/
                         ACCin,ACCout,FAUin,FAUsel,Ein,MemRead,MemWrite}
```

```
<=15'b11xx101x1x11010;

        11'b10100111000:{RegRead,RegWrite,AorD,BorE,Cin,Fin,DMin,       /*減算・ACC 乗算・除算の終了*/
                          ACCin,ACCout,FAUin,FAUsel,Ein,MemRead,MemWrite}
<=15'b11xx011x1x11000;

        11'b10100100010:{RegRead,RegWrite,AorD,BorE,Cin,Fin,DMin,       /*減算・ACC 乗算*/
                          ACCin,ACCout,FAUin,FAUsel,Ein,MemRead,MemWrite}
<=15'b11xxxx1x1x11000;

        11'b10100001010:{RegRead,RegWrite,AorD,BorE,Cin,Fin,DMin,       /*減算・除算の開始・ロード*/
                          ACCin,ACCout,FAUin,FAUsel,Ein,MemRead,MemWrite}
<=15'b1100101x0x11010;

        11'b10100001000:{RegRead,RegWrite,AorD,BorE,Cin,Fin,DMin,       /*減算・除算の開始*/
                          ACCin,ACCout,FAUin,FAUsel,Ein,MemRead,MemWrite}
<=15'b1100xx1x0x11000;

        11'b10100011100:{RegRead,RegWrite,AorD,BorE,Cin,Fin,DMin,       /*減算・除算の終了・ストア*/
                          ACCin,ACCout,FAUin,FAUsel,Ein,MemRead,MemWrite}
<=15'b11xx01110x11001;

        11'b10100011000:{RegRead,RegWrite,AorD,BorE,Cin,Fin,DMin,       /*減算・除算の終了*/
                          ACCin,ACCout,FAUin,FAUsel,Ein,MemRead,MemWrite}
<=15'b11xx011x0x11000;

        11'b10100000010:{RegRead,RegWrite,AorD,BorE,Cin,Fin,DMin,       /*減算・ロード*/
                          ACCin,ACCout,FAUin,FAUsel,Ein,MemRead,MemWrite}
<=15'b11xx101x0x11010;

        11'b10100000100:{RegRead,RegWrite,AorD,BorE,Cin,Fin,DMin,       /*減算・ストア*/
                          ACCin,ACCout,FAUin,FAUsel,Ein,MemRead,MemWrite}
<=15'b11xxxx110x11001;

        11'b10100000000:{RegRead,RegWrite,AorD,BorE,Cin,Fin,DMin,       /*減算*/
                          ACCin,ACCout,FAUin,FAUsel,Ein,MemRead,MemWrite}
```

```
<=15'b11xxxx1x0x11000;

        11'b00010001000:{RegRead,RegWrite,AorD,BorE,Cin,Fin,DMin,       /*乗算・除算の開始*/
                          ACCin,ACCout,FAUin,FAUsel,Ein,MemRead,MemWrite}
<=15'b111100xx0xxx000;

        11'b00010011010:{RegRead,RegWrite,AorD,BorE,Cin,Fin,DMin,       /*乗算・除算の終了・ロード*/
                          ACCin,ACCout,FAUin,FAUsel,Ein,MemRead,MemWrite}
<=15'b11xx000x0xxx110;

        11'b00010011100:{RegRead,RegWrite,AorD,BorE,Cin,Fin,DMin,       /*乗算・除算の終了・ストア*/
                          ACCin,ACCout,FAUin,FAUsel,Ein,MemRead,MemWrite}
<=15'b11xx00000xxxx01;

        11'b00010011000:{RegRead,RegWrite,AorD,BorE,Cin,Fin,DMin,       /*乗算・除算の終了*/
                          ACCin,ACCout,FAUin,FAUsel,Ein,MemRead,MemWrite}
<=15'b11xx000x0xxxx00;

        11'b00010000010:{RegRead,RegWrite,AorD,BorE,Cin,Fin,DMin,       /*乗算・ロード*/
                          ACCin,ACCout,FAUin,FAUsel,Ein,MemRead,MemWrite}
<=15'b11xx00xx0xxx110;

        11'b00010000100:{RegRead,RegWrite,AorD,BorE,Cin,Fin,DMin,       /*乗算・ストア*/
                          ACCin,ACCout,FAUin,FAUsel,Ein,MemRead,MemWrite}
<=15'b11xx00x00xxxx01;

        11'b00010000000:{RegRead,RegWrite,AorD,BorE,Cin,Fin,DMin,       /*乗算*/
                          ACCin,ACCout,FAUin,FAUsel,Ein,MemRead,MemWrite}
<=15'b11xx00xx0xxxx00;

        11'b00000101010:{RegRead,RegWrite,AorD,BorE,Cin,Fin,DMin, /*ACC 乗算・除算の開始・ロード*/
                          ACCin,ACCout,FAUin,FAUsel,Ein,MemRead,MemWrite}
<=15'b111110xx1xxx010;

        11'b00000101000:{RegRead,RegWrite,AorD,BorE,Cin,Fin,DMin,       /*ACC 乗算・除算の開始*/
                          ACCin,ACCout,FAUin,FAUsel,Ein,MemRead,MemWrite}
```

```
<=15'b1011xxxx1xxx000;

        11'b00000111010:{RegRead,RegWrite,AorD,BorE,Cin,Fin,DMin,   /*ACC 乗算・除算の終了・ロード*/
                         ACCin,ACCout,FAUin,FAUsel,Ein,MemRead,MemWrite}
<=15'b11xxxx0x1xxx101;

        11'b00000111100:{RegRead,RegWrite,AorD,BorE,Cin,Fin,DMin,   /*ACC 乗算・除算の終了・ストア*/
                         ACCin,ACCout,FAUin,FAUsel,Ein,MemRead,MemWrite}
<=15'b11xxxx001xxxx01;

        11'b00000111000:{RegRead,RegWrite,AorD,BorE,Cin,Fin,DMin,   /*ACC 乗算・除算の終了*/
                         ACCin,ACCout,FAUin,FAUsel,Ein,MemRead,MemWrite}
<=15'b11xxxx0x1xxxx00;

        11'b00000100010:{RegRead,RegWrite,AorD,BorE,Cin,Fin,DMin,      /*ACC 乗算・ロード*/
                         ACCin,ACCout,FAUin,FAUsel,Ein,MemRead,MemWrite}
<=15'b11xxxxxx1xxx110;

        11'b00000100100:{RegRead,RegWrite,AorD,BorE,Cin,Fin,DMin,      /*ACC 乗算・ストア*/
                         ACCin,ACCout,FAUin,FAUsel,Ein,MemRead,MemWrite}
<=15'b11xxxxx01xxxx01;

        11'b00000100000:{RegRead,RegWrite,AorD,BorE,Cin,Fin,DMin,      /*ACC 乗算*/
                         ACCin,ACCout,FAUin,FAUsel,Ein,MemRead,MemWrite}
<=15'b11xxxxxx1xxxx00;

        11'b00000001010:{RegRead,RegWrite,AorD,BorE,Cin,Fin,DMin,      /*除算の開始・ロード*/
                         ACCin,ACCout,FAUin,FAUsel,Ein,MemRead,MemWrite}
<=15'b1100xxxx0xxx110;

        11'b00000001100:{RegRead,RegWrite,AorD,BorE,Cin,Fin,DMin,      /*除算の開始・ストア*/
                         ACCin,ACCout,FAUin,FAUsel,Ein,MemRead,MemWrite}
<=15'b1000xxx00xxxx01;

        11'b00000001000:{RegRead,RegWrite,AorD,BorE,Cin,Fin,DMin,      /*除算の開始*/
                         ACCin,ACCout,FAUin,FAUsel,Ein,MemRead,MemWrite}
```

```
<=15'b1000xxxx0xxxx00;

      11'b00000011010:{RegRead,RegWrite,AorD,BorE,Cin,Fin,DMin,       /*除算の終了・ロード*/
                        ACCin,ACCout,FAUin,FAUsel,Ein,MemRead,MemWrite}
<=15'b11xx01xx0xxx110;

      11'b00000011100:{RegRead,RegWrite,AorD,BorE,Cin,Fin,DMin,       /*除算の終了・ストア*/
                        ACCin,ACCout,FAUin,FAUsel,Ein,MemRead,MemWrite}
<=15'b11xx01x10xxxx01;

      11'b00000011000:{RegRead,RegWrite,AorD,BorE,Cin,Fin,DMin,       /*除算の終了*/
                        ACCin,ACCout,FAUin,FAUsel,Ein,MemRead,MemWrite}
<=15'b01xx01xx0xxxx00;

      11'b00000000010:{RegRead,RegWrite,AorD,BorE,Cin,Fin,DMin,       /*ロード*/
                        ACCin,ACCout,FAUin,FAUsel,Ein,MemRead,MemWrite}
<=15'b11xxxxxx0xxx110;

      11'b00000000100:{RegRead,RegWrite,AorD,BorE,Cin,Fin,DMin,       /*ストア*/
                        ACCin,ACCout,FAUin,FAUsel,Ein,MemRead,MemWrite}
<=15'b10xxxxx10xxxx01;

      11'b11010000100:{RegRead,RegWrite,AorD,BorE,Cin,Fin,DMin,       /*ACC 加算・乗算・ストア*/
                        ACCin,ACCout,FAUin,FAUsel,Ein,MemRead,MemWrite}
<=15'b11xx00100100001;

      11'b11010000000:{RegRead,RegWrite,AorD,BorE,Cin,Fin,DMin,       /*ACC 加算・乗算*/
                        ACCin,ACCout,FAUin,FAUsel,Ein,MemRead,MemWrite}
<=15'b11xx001x0100000;

      11'b11000111000:{RegRead,RegWrite,AorD,BorE,Cin,Fin,DMin,       /*ACC 加算・ACC 乗算・除算の
終了*/
                        ACCin,ACCout,FAUin,FAUsel,Ein,MemRead,MemWrite}
<=15'b11xx011x1100000;

      11'b11000100010:{RegRead,RegWrite,AorD,BorE,Cin,Fin,DMin,   /*ACC 加算・ACC 乗算・ロード*/
```

```
                    ACCin,ACCout,FAUin,FAUsel,Ein,MemRead,MemWrite}
<=15'b11xx101x1100010;

    11'b11000100100:{RegRead,RegWrite,AorD,BorE,Cin,Fin,DMin,    /*ACC 加算・ACC 乗算・ストア*/
                    ACCin,ACCout,FAUin,FAUsel,Ein,MemRead,MemWrite}
<=15'b11xxxx101100001;

    11'b11000100000:{RegRead,RegWrite,AorD,BorE,Cin,Fin,DMin,    /*ACC 加算・ACC 乗算*/
                    ACCin,ACCout,FAUin,FAUsel,Ein,MemRead,MemWrite}
<=15'b11xxxx1x1100000;

    11'b11000001010:{RegRead,RegWrite,AorD,BorE,Cin,Fin,DMin,    /*除算の開始・ロード・ACC 加算*/
                    ACCin,ACCout,FAUin,FAUsel,Ein,MemRead,MemWrite}
<=15'b1100101x0100010;

    11'b11000001010:{RegRead,RegWrite,AorD,BorE,Cin,Fin,DMin,    /*除算の開始・ストア・ACC 加算*/
                    ACCin,ACCout,FAUin,FAUsel,Ein,MemRead,MemWrite}
<=15'b1100xx100100001;

    11'b11000001000:{RegRead,RegWrite,AorD,BorE,Cin,Fin,DMin,    /*除算の開始・ACC 加算*/
                    ACCin,ACCout,FAUin,FAUsel,Ein,MemRead,MemWrite}
<=15'b1100xx1x0100000;

    11'b11000011100:{RegRead,RegWrite,AorD,BorE,Cin,Fin,DMin,    /*除算の終了・ストア・ACC 加算*/
                    ACCin,ACCout,FAUin,FAUsel,Ein,MemRead,MemWrite}
<=15'b11xx01110100001;

    11'b11000011000:{RegRead,RegWrite,AorD,BorE,Cin,Fin,DMin,    /*除算の終了・ACC 加算*/
                    ACCin,ACCout,FAUin,FAUsel,Ein,MemRead,MemWrite}
<=15'b11xx011x0100000;

    11'b11000000010:{RegRead,RegWrite,AorD,BorE,Cin,Fin,DMin,    /*ロード・ACC 加算*/
                    ACCin,ACCout,FAUin,FAUsel,Ein,MemRead,MemWrite}
<=15'b11xx101x0100010;

    11'b11000000100:{RegRead,RegWrite,AorD,BorE,Cin,Fin,DMin,    /*ストア・ACC 加算*/
```

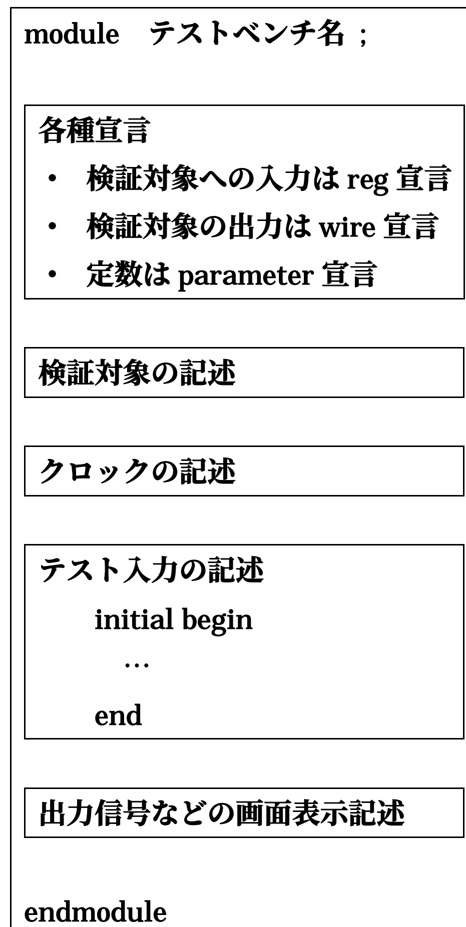
```
                    ACCin,ACCout,FAUin,FAUsel,Ein,MemRead,MemWrite}
<=15'b11xxxx110100001;

    11'b11000000000:{RegRead,RegWrite,AorD,BorE,Cin,Fin,DMin,    /*ACC 加算*/
                    ACCin,ACCout,FAUin,FAUsel,Ein,MemRead,MemWrite}
<=15'b11xxxx1x0100000;

    11'b11110000100:{RegRead,RegWrite,AorD,BorE,Cin,Fin,DMin,    /*ACC 減算・乗算・ストア*/
                    ACCin,ACCout,FAUin,FAUsel,Ein,MemRead,MemWrite}
<=15'b11xx00100101001;

    11'b11110000000:{RegRead,RegWrite,AorD,BorE,Cin,Fin,DMin,    /*ACC 減算・乗算*/
                    ACCin,ACCout,FAUin,FAUsel,Ein,MemRead,MemWrite}
<=15'b11xx001x0101000;

    11'b11100111000:{RegRead,RegWrite,AorD,BorE,Cin,Fin,DMin,    /*ACC 減算・ACC 乗算・除算の
終了*/
                    ACCin,ACCout,FAUin,FAUsel,Ein,MemRead,MemWrite}
<=15'b11xx011x1101000;

    11'b11100100010:{RegRead,RegWrite,AorD,BorE,Cin,Fin,DMin,  /*ACC 減算・ACC 乗算・ロード*/
                    ACCin,ACCout,FAUin,FAUsel,Ein,MemRead,MemWrite}
<=15'b11xx101x1101010;

    11'b11100100100:{RegRead,RegWrite,AorD,BorE,Cin,Fin,DMin,   /*ACC 減算・ACC 乗算・ストア*/
                    ACCin,ACCout,FAUin,FAUsel,Ein,MemRead,MemWrite}
<=15'b11xxxx101101001;

    11'b11100100000:{RegRead,RegWrite,AorD,BorE,Cin,Fin,DMin,    /*ACC 減算・ACC 乗算*/
                    ACCin,ACCout,FAUin,FAUsel,Ein,MemRead,MemWrite}
<=15'b11xxxx1x1101000;

    11'b11100001010:{RegRead,RegWrite,AorD,BorE,Cin,Fin,DMin,  /*除算の開始・ロード・ACC 減算*/
                    ACCin,ACCout,FAUin,FAUsel,Ein,MemRead,MemWrite}
<=15'b1100101x0101010;
```

```
        11'b11100001010:{RegRead,RegWrite,AorD,BorE,Cin,Fin,DMin,   /*除算の開始・ストア・ACC 減算*/
                         ACCin,ACCout,FAUin,FAUsel,Ein,MemRead,MemWrite}
<=15'b1100xx100101001;

        11'b11100001000:{RegRead,RegWrite,AorD,BorE,Cin,Fin,DMin,       /*除算の開始・ACC 減算*/
                         ACCin,ACCout,FAUin,FAUsel,Ein,MemRead,MemWrite}
<=15'b1100xx1x0101000;

        11'b11100011100:{RegRead,RegWrite,AorD,BorE,Cin,Fin,DMin,   /*除算の終了・ストア・ACC 減算*/
                         ACCin,ACCout,FAUin,FAUsel,Ein,MemRead,MemWrite}
<=15'b11xx01110101001;

        11'b11100011000:{RegRead,RegWrite,AorD,BorE,Cin,Fin,DMin,       /*除算の終了・ACC 減算*/
                         ACCin,ACCout,FAUin,FAUsel,Ein,MemRead,MemWrite}
<=15'b11xx011x0101000;

        11'b11100000010:{RegRead,RegWrite,AorD,BorE,Cin,Fin,DMin,       /*ロード・ACC 減算*/
                         ACCin,ACCout,FAUin,FAUsel,Ein,MemRead,MemWrite}
<=15'b11xx101x0101010;

        11'b11100000100:{RegRead,RegWrite,AorD,BorE,Cin,Fin,DMin,       /*ストア・ACC 減算*/
                         ACCin,ACCout,FAUin,FAUsel,Ein,MemRead,MemWrite}
<=15'b11xxxx110101001;

        11'b11100000000:{RegRead,RegWrite,AorD,BorE,Cin,Fin,DMin,       /*ACC 減算*/
                         ACCin,ACCout,FAUin,FAUsel,Ein,MemRead,MemWrite}
<=15'b11xxxx1x0101000;


        default : {RegRead,RegWrite,AorD,BorE,Cin,Fin,DMin,       /*その他*/
                   ACCin,ACCout,FAUin,FAUsel,Ein,MemRead,MemWrite} <=15'bxxxxxxxxxxxxxxx;

        endcase

    end
```

4　テストベンチの記述とシミュレーション

ＨＤＬ設計ではＲＴＬ回路をＨＤＬで記述するだけでなく、正しく動作するかをＨＤＬシミュレータで確認しなければならない。このとき、テスト対象であるＲＴＬ回路のみでなく、回路にテスト信号を与える記述が必要になる。これをテストベンチという。そこでここではまずＦＰＵに必要な構成要素をそれぞれＨＤＬによるテストベンチを記述、そしてシミュレーションを用いて、動作確認を行う。最後にそれぞれの部品を接続してＦＰＵのテストベンチを記述、シミュレーションを用いてＦＰＵ全体の回路の動作確認を行っていく。

4.1 テストベンチのＨＤＬ記述構造



この記述構造をもとに以下、ＦＰＵに必要な構成要素のテストベンチをＨＤＬ記述していく。

（テストベンチは同様の形式のため、ここでは本研究で作成したアキュムレータ、再構築した命令制御ユニットについてのみの説明に限る）

4.2　各構成要素のテストベンチ記述とシミュレーション実行

通常、回路のシミュレーションは真理値表をもとに正しく動作しているかを確認する必要がある。たとえば、二つの入力を加算し出力する半加算器と呼ばれるものがある。これは、1 ビットの入力 a,b を加算し、和 s と桁上がり信号 co が出力される。これら入出力の動作を真理値表で表すと、表 4.1 のようになる。

a	b	s	co
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

表 4.1　半加算器の真理値表

テストベンチでは入力 a,b を (a,b) = (0,0),(0,1),(1,0),(1,1) と入力し、s,co が (s,co)= (0,0),(1,0),(1,0),(0,1)と出力されることを確認できるよう記述する必要がある。こうして、シミュレーションを実行し真理値表と同じ動作をすれば、半加算器として間違いなく動作するということが確認できる。

4.2.1　アキュムレータのシミュレーション

（ｉ）テストベンチ記述

```
module acc_test;
   reg  [31:0]  a;
   reg          ACCin,ACCout,ck,res;
   wire [31:0]  q;

  acc A(a,ACCin,ACCout,ck,res,q);

  always begin
    ck=0; #150;
    ck=1; #150;
  end

   initial begin
          a=32'h00000001; ACCin=1'b1; ACCout=1'b0; res=1'b1;
     #600  a=32'h00000001; ACCin=1'b1; ACCout=1'b0; res=1'b0;
     #300  a=32'h00000101; ACCin=1'b1; ACCout=1'b1; res=1'b0;
     #300  a=32'h00000001; ACCin=1'b0; ACCout=1'b1; res=1'b0;
     #300  a=32'h00000011; ACCin=1'b1; ACCout=1'b0; res=1'b0;
     #300  a=32'h00000001; ACCin=1'b0; ACCout=1'b1; res=1'b0;
     #300  a=32'h00000001; ACCin=1'b1; ACCout=1'b1; res=1'b0;
     #300  a=32'h00000001; ACCin=1'b0; ACCout=1'b1;  res=1'b0;
     #500  $finish;

   end

  initial begin
     $monitor ($stime," q=%h",q);
  end
endmodule
```

(ii)　シミュレーションの波形表示

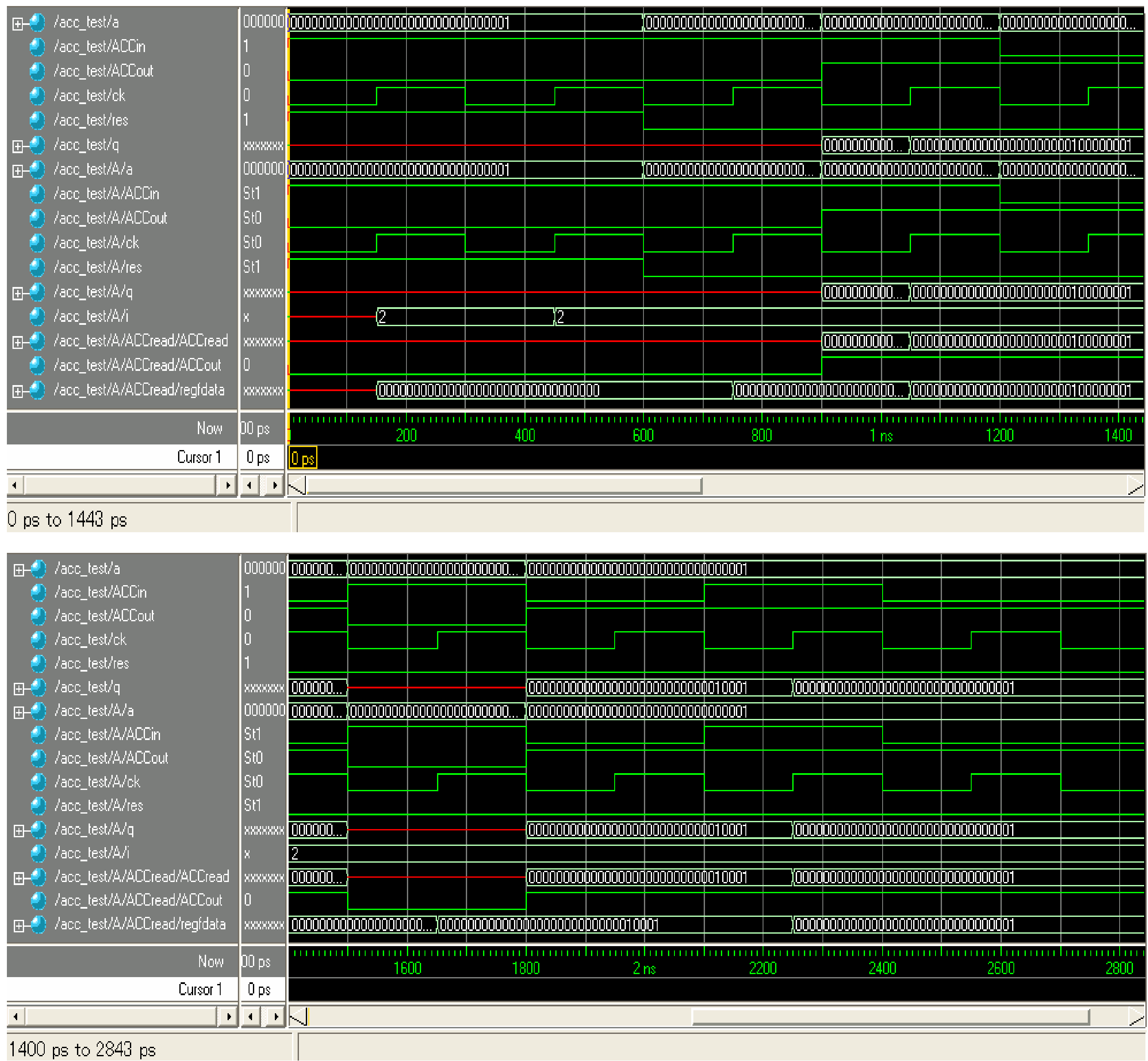


図 4.1　アキュムレータのシミュレーション結果の波形表示

(iii)　シミュレーション結果の説明

図 4.1 はテストベンチの波形表示である。
ck や res などの名称はＦＰＵ全体図に対応してある。

まず、初期状態を見てみると、600ns までは ck の変化にかかわらず res=1 となっているので、初期リセットとしてレジスタファイルが 0 にリセットされている。

600ns のとき、ACCin=1 となるが、ck=0 より何も働かない。
750ns のとき、ACCin=1,ck=1 より、入力データを ACC に書き込んでいる。
ここでは a=00000000000000000000000000010001 が regfdata に書き込まれている。

900ns のとき、ACCout=1 となるので ACC に格納されていた値(regfdata)が出力されている。出力の値はｑである。
1050ns のとき、ck の立ち上がりにより、新たな a の値が ACC に書き込まれる。
ここでは a=00000000000000000000000100000001 が regfdata に書き込まれている。

1200ns のとき、ACCin=0 となり、ACCout は変化なし。
1350ns のとき、ACCin=0 より、書き込みは行われず、格納されているデータに変化はなし。ACCout=1 は変わらず、1050ns のときと同じデータ a を出力している。

1500ns のとき、ACCout=0 より、出力はされない。ACCin=1 となるが、ck=0 より書き込みは行われない。
1650ns のとき、ck の立ち上がりで、ACCin=1 より、新たな入力データが ACC に書き込まれている。ここでは a=00000000000000000000000000010001 が regfdata に書き込まれている。

1800ns のとき、ACCout=1 となり、1650ns のときに新たに ACC に書き込まれた値が出力されている。
1950ns のとき、ck=1 となるが、ACCin=0 より ACC に書き込みは行われず、格納されているデータに変化はなし。

2100ns のとき、ACCin=1 となるが、ck=0 より書き込みは行われない。ACCout=1 は変わらず、同じ値を出力している。
2250ns のとき、ck の立ち上がりで、ACCin=1 より新たな入力データが ACC に書き込

まれている。ここでは a=00000000000000000000000000000001 が regfdata に書き込まれている。

2400ns のとき、ACCout=1 より、2250ns のとき新たに ACC に書き込まれたデータが出力されている。
2550ns のとき、ck=1 となるが、ACCin=0 より、ACC に書き込みは行われず、格納されているデータに変化はなし。

以上で、アキュムレータのシミュレーションを完了した。

4.2.2　制御回路ユニットのシミュレーション

（ｉ）テストベンチ記述

```
module fpu_ctrl_unit_test;

reg     [10:0] op;
wire            RegRead,RegWrite,AorD,BorE,Fin,DMin,
                ACCin,ACCout,FAUin,FAUsel,MemRead,MemWrite;
wire    [1:0]   Cin;

parameter STEP=300;

fpu_ctrl_unit f1(op,RegRead,RegWrite,AorD,BorE,Fin,DMin,
                    ACCin,ACCout,FAUin,FAUsel,MemRead,MemWrite,Cin);

initial begin
   #STEP       op=11'b10010000000;
   #STEP       op=11'b00010011010;
   #STEP       op=11'b00000001010;
   #STEP       op=11'b00000011010;
   #STEP       op=11’b11010000000;
   #STEP $finish;
end
```

```
initial begin
$monitor  ($time,"op=%b  RegRead=%b  RegWrite=%b  AorD=%b  BorE=%b  Fin=%b
           DMin=%b   ACCin=%b   ACCout=%b   FAUin=%b   FAUsel=%b   Ein=%b
           MemRead=%b  MemWrite=%b
           Cin=%b",op,RegRead,RegWrite,AorD,BorE,Fin,DMin,ACCin,ACCout,FAUin,
           FAUsel,Ein,MemRead,MemWrite,Cin);
end
endmodule
```

（ii）シミュレーションの波形表示

図 4.2　　命令制御ユニットのシミュレーション結果の波形表示

（iii）シミュレーション結果の説明
制御回路ユニットのＨＤＬ記述によると、op コードに対応する制御信号は以下のようにならなければならない。

Op	10010000000	00010011010	00000101010	00000011010	11010000000
RegRead	1	1	1	1	1
RegWrite	1	1	1	1	1
AorD	x	x	1	x	x
BorE	x	x	1	x	x
Cin	00	00	10	01	00
Fin	1	0	x	x	1
Dmin	x	x	x	x	x
ACCin	0	0	1	0	0
ACCout	x	x	x	x	1
FAUin	1	x	x	x	0
FAUsel	0	x	x	x	0
Ein	0	1	0	1	0
MemRead	0	1	1	1	0
MemWrite	0	0	0	0	0

表 4.2　op コードと制御信号の関係図

上記の表と、シミュレーション結果とを照らし合わせると一致するのでＨＤＬ記述が正しいことが証明できた。

4..3　ＦＰＵ全体の回路のテストベンチ記述とシミュレーション実行

前章では、ＦＰＵの各構成要素のシミュレーションを行ってきた。本章では、すべてのデータパスを結合しＦＰＵとしてシミュレーションを行う。改良した制御により、加減乗除・ロード・ストア命令とこれらの複数の命令を同時に実行する並列処理が正しく動作するか、そして今回のＦＰＵの改良による性能の評価を検討していく。

4.3.1　行列演算の実行

本浮動小数点プロセッサの性能を現実的なプログラムで評価するため、グラフィック処理でよく用いられる４×４の行列による座標変換に適用した。さらに、行列演算は積和演算を多用するため、ＦＰＵへのアキュムレータ取り付けによる性能評価に適していると考えたからである。

図 4.3 に座標ベクトル（X, Y, Z, W）の変換のための４×４の行列式を示す。入力座標ベクトル（X, Y, Z, W）に対して、出力としてW’、および出力座標ベクトル（X’/W’、Y’/W’、Z’/W’、１）を得る。

$$\begin{vmatrix} a11 & a12 & a13 & a14 \\ a21 & a22 & a23 & a24 \\ a31 & a32 & a33 & a34 \\ a41 & a42 & a43 & a44 \end{vmatrix} \begin{vmatrix} X \\ Y \\ Z \\ W \end{vmatrix} = \begin{vmatrix} X' \\ Y' \\ Z' \\ W' \end{vmatrix} = W' \begin{vmatrix} X'/W' \\ Y'/W' \\ Z'/W' \\ 1 \end{vmatrix}$$

入力：座標ベクトル（X, Y, Z, W）
出力：W’、座標ベクトル（X’/W’、Y’/W’、Z’/W’、１）

図4.3　　４行×４列の行列計算

次ページ図 4.4 に示すように内部のマルチポートレジスタに行列要素をもたせ、変換すべき座標データを入力バスから入力しつつ、変換後のデータを出力バスから出力することにより実行する。

A11 ： R1	a12 ： R2	a13 ： R3	a14 ： R4

A21 ： R5	a22 ： R6	a23 ： R7	a24 ： R8
A31 ： R9	a32 ： R10	a33 ： R11	a34 ： R12
A41 ： R13	a42 ： R14	a43 ： R15	a44 ： R16

1

図 4.4　行列要素の格納レジスタ（Ｒｎはｎ番地のレジスタを示す）

以下に４×４の行列演算の命令ステップ毎の具体的な動作と、ＦＰＵが認識可能な機械語に翻訳したものをそれぞれ示す。

/*メモリ０番地にはデータ 01000000100000000000000000000000 が入っている*/
/*0*/000_000_00_00_0_00000_00000_00000_00000_00000_00000
/*1*/000_000_00_01_0_00000_00000_00000_00000_00001_00000
メモリからＲ１にロード
/*2*/000_000_00_01_0_00000_00000_00000_00000_00010_00000
メモリからＲ２にロード
/*3*/000_000_00_01_0_00000_00000_00000_00000_00011_00000
メモリからＲ３にロード
/*4*/000_000_00_01_0_00000_00000_00000_00000_00100_00000
メモリからＲ４にロード
/*5*/000_000_00_01_0_00000_00000_00000_00000_00101_00000
メモリからＲ５にロード
/*6*/000_000_00_01_0_00000_00000_00000_00000_00110_00000
メモリからＲ６にロード
/*7*/000_000_00_01_0_00000_00000_00000_00000_00111_00000
メモリからＲ７にロード
/*8*/000_000_00_01_0_00000_00000_00000_00000_01000_00000
メモリからＲ８にロード
/*9*/000_000_00_01_0_00000_00000_00000_00000_01001_00000
メモリからＲ９にロード

/*10*/000_000_00_01_0_00000_00000_00000_00000_01010_00000
メモリからＲ１０にロード
/*11*/000_000_00_01_0_00000_00000_00000_00000_01011_00000
メモリからＲ１１にロード
/*12*/000_000_00_01_0_00000_00000_00000_00000_01100_00000
メモリからＲ１２にロード
/*13*/000_000_00_01_0_00000_00000_00000_00000_01101_00000
メモリからＲ１３にロード
/*14*/000_000_00_01_0_00000_00000_00000_00000_01110_00000
メモリからＲ１４にロード
/*15*/000_000_00_01_0_00000_00000_00000_00000_01111_00000
メモリからＲ１５にロード
/*16*/000_000_00_01_0_00000_00000_00000_00000_10000_00000
メモリからＲ１６にロード
/*17*/000_000_00_01_0_00000_00000_00000_00000_10001_00000
ロード（Ｘ）
/*18*/000_100_00_01_0_01101_10001_10101_00000_10010_00000
ロード（Ｙ）、乗算：R21←a41*X
/*19*/000_001_00_01_0_01110_10010_00000_00000_10011_00000
ロード（Ｚ）、ACC 乗算：ACC←a42*Y
/*20*/110_001_00_01_0_01111_10011_10100_00000_10101_11110
ロード（W）、ACC 加算：R30←ACC+R21、ACC 乗算：ACC←a43*Z
/*21*/110_001_00_00_0_10000_10100_00000_00000_11110_11111
ACC 加算：R31←ACC+R30、ACC 乗算：ACC←a44*W
/*22*/110_100_00_00_0_00001_10001_10110_00000_11111_10101
ACC 加算：R21←ACC+R31、乗算：R22←a11*X
/*23*/000_001_00_00_0_00010_10010_00000_00000_00000_00000
ACC 乗算：ACC←a12*Y
/*24*/110_001_00_00_0_00011_10011_00000_00000_10110_11110
ACC 加算：R30←ACC+R22、ACC 乗算：ACC←a13*Z
/*25*/110_001_00_00_0_00100_10100_00000_00000_11110_11111
ACC 加算：R31←ACC+R30、ACC 乗算：ACC←a14*W
/*26*/110_100_00_00_0_00101_10001_10111_00000_11111_10110
ACC 加算：R22←ACC+R31、乗算：R23←a21*X
/*27*/000_001_01_00_0_00110_10010_00000_10110_10101_11001
ACC 乗算：ACC←a22*Y、除算の開始：R25←X’/W’

/*28*/110_001_00_00_0_00111_10011_00000_00000_10111_11110
ACC 加算：R30←ACC+R23、ACC 乗算：ACC←a23*Z
/*29*/110_001_00_00_0_01000_10100_00000_00000_11110_11111
ACC 加算：R31←ACC+R30、ACC 乗算：ACC←a24*W
/*30*/110_000_00_00_0_00000_00000_00000_00000_11111_10111
ACC 加算：R23←ACC+R31
/*31*/000_100_01_00_0_01001_10001_11000_10111_10101_11010
乗算：R24←a31*X、除算の開始：R26←Y’/W’
/*32*/000_001_00_00_0_01010_10010_00000_00000_00000_00000
ACC 乗算：ACC←a32*Y
/*33*/110_001_00_00_0_01011_10011_00000_00000_11000_11110
ACC 加算：R30←ACC+R24、ACC 乗算：ACC←a33*Z
/*34*/110_001_00_00_0_01100_10100_00000_00000_11110_11111
ACC 加算：R31←ACC+R30、ACC 乗算：ACC←a34*W
/*35*/110_000_00_00_0_00000_00000_00000_00000_11111_11000
ACC 加算：R24←ACC+R31
/*36*/000_000_01_10_0_11000_10101_11011_10101_00000_00000
除算の開始：R27←Z’/W’、R21 の値をストア
/*37*/000_000_00_10_0_00000_11001_00000_00000_00000_00000
R25 の値をストア
/*38*/000_000_01_10_0_10101_10101_11100_11010_00000_00000
除算の開始：R28←W’/W’、R26 の値をストア
/*39*/000_000_00_10_0_00000_11011_00000_00000_00000_00000
R27 の値をストア
/*40*/000_000_00_10_0_00000_11100_00000_00000_00000_00000
R28 の値をストア

表 4.3　行列演算の命令ステップ毎の動作とその機械語表記

この命令によると、図 4.5 のような値が出ることが予想される。

$$\begin{vmatrix} 4 & 4 & 4 & 4 \\ 4 & 4 & 4 & 4 \\ 4 & 4 & 4 & 4 \\ 4 & 4 & 4 & 4 \end{vmatrix} \begin{vmatrix} 4 \\ 4 \\ 4 \\ 4 \end{vmatrix} = \begin{vmatrix} 64 \\ 64 \\ 64 \\ 64 \end{vmatrix} = 64 \begin{vmatrix} 1 \\ 1 \\ 1 \\ 1 \end{vmatrix}$$

図 4.5　行列演算の結果予想

次に、テストベンチ記述である。冒頭にもふれたように、テストベンチは HDL 記述でかかれた RTL 回路にテスト信号を与える記述である。これに含まれる内容は、以下の通りである。

- 上記プログラムを保存、読み出しする命令メモリと、データの保存、読み書きするデータメモリを作成する。また、プログラムを命令メモリにロードする。
- クロック入力 ck に、1 クロック 10ns とし、シミュレーション開始から 10ns 後にたち上がり、シミュレーション終了までクロック信号を与え続ける。
- リセット信号 res を初期リセット（res=1）とし、プログラムカウンタとレジスタを 0 にリセットする。その後は、0 にして、リセットはしない。
- 命令メモリからプログラムカウンタで指定された命令を読み出し、実行する。
- プログラムカウンタが命令を命令メモリにないアドレスを指定し読み出しすると、つまりプログラムが終了するとシミュレーション終了とする。

以下に、そのＨＤＬ記述を示す。

```
module fpu_simulation;

   reg          ck,res;
   reg   [40:0] id;
   reg   [31:0] r_data;
   wire  [31:0] w_data,register_out;
   wire  [40:0] pc_out;
   wire         MemRead,MemWrite;
   integer      i;

   reg   [40:0] id_mem[0:50];     //命令メモリの生成
   reg   [31:0] dt_mem[0:100];    //データメモリの生成

   parameter  STEP = 10;

   always begin
         #(STEP/2) ck = 0;
         #(STEP/2) ck = 1;
   end
```

```
  always @(pc_out) begin              //命令メモリの動作記述
          id <= id_mem[pc_out];
  end

  always @(register_out) begin         //データメモリの動作記述
       if(MemWrite == 1'b1)
           dt_mem[register_out] <= w_data;
       if(MemRead == 1'b1)
           r_data <= dt_mem[register_out];
  end

  FPU i0(ck,res,pc_out,id,register_out,w_data,r_data,MemRead,MemWrite);

  initial begin
        $readmemb("id.dat",id_mem);              //命令データの読み込み
        $readmemb("dt.dat",dt_mem);              //メモリの初期データの読み込
み

        #STEP        res = 1'b1;                 //PC,レジスタのリセット
        #(STEP/4)    res = 1'b0;

        while (id !==
41'bxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx)begin        //命令数だけクロック
を進める
             #STEP;
        end

        $stop;
  end

endmodule
```

以上のようにして、作成したテストベンチによって、シミュレーションを実行する。
以下にはシミュレーション結果を示し、クロックごとに説明していく。

次ページの図 4.6 からは

加減算器
fpu_simulation/i0/i2/a（入力）
fpu_simulation/i0/i2/b（入力）
fpu_simulation/i0/i2/out（出力）

除算器
fpu_simulation/i0/i3/a（入力）
fpu_simulation/i0/i3/b（入力）
fpu_simulation/i0/i3/out（出力）

乗算器
fpu_simulation/i0/i4/a（入力）
fpu_simulation/i0/i4/b（入力）
fpu_simulation/i0/i4/out（出力）

アキュムレータ（ＡＣＣ）
fpu_simulation/i0/i12/a（入力）
fpu_simulation/i0/i12/q（出力）

に対応しており、その他はＦＰＵ全体図に対応している。

初期状態は、ck は不定値から０を始めとして動作を開始している。それ以外は不定値である。それより次の状態からは、以下よりクロックごとに図 4.6 から示し、それぞれについて詳しくみていく。

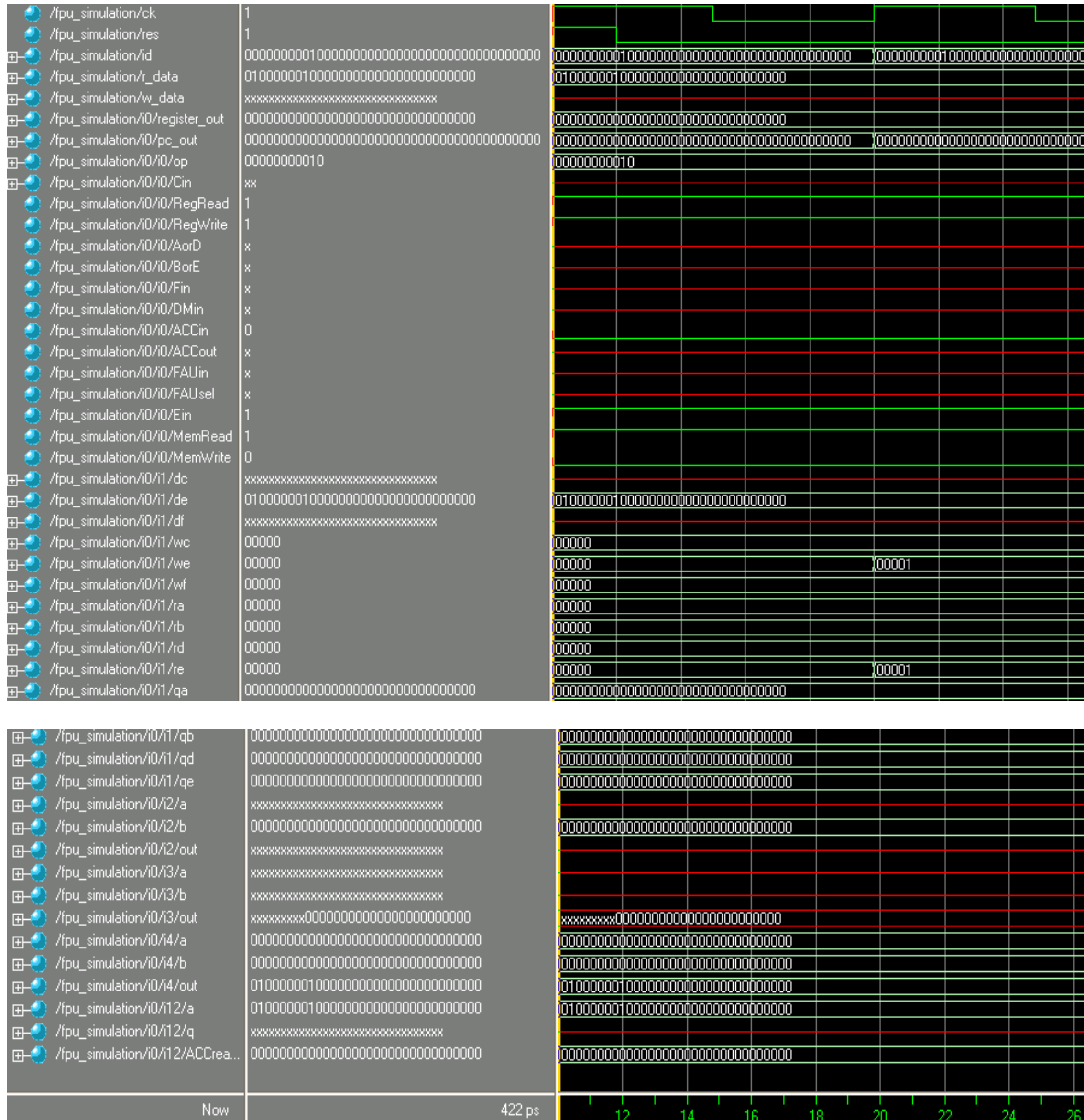


図 4.6　テストベンチのシミュレーション結果（10ns 時）

10ns のとき、res が 1 になっているので、初期リセットとしてＰＣ（プログラムカウンタ）と、regf(レジスタファイル)が 0 にリセットされる。

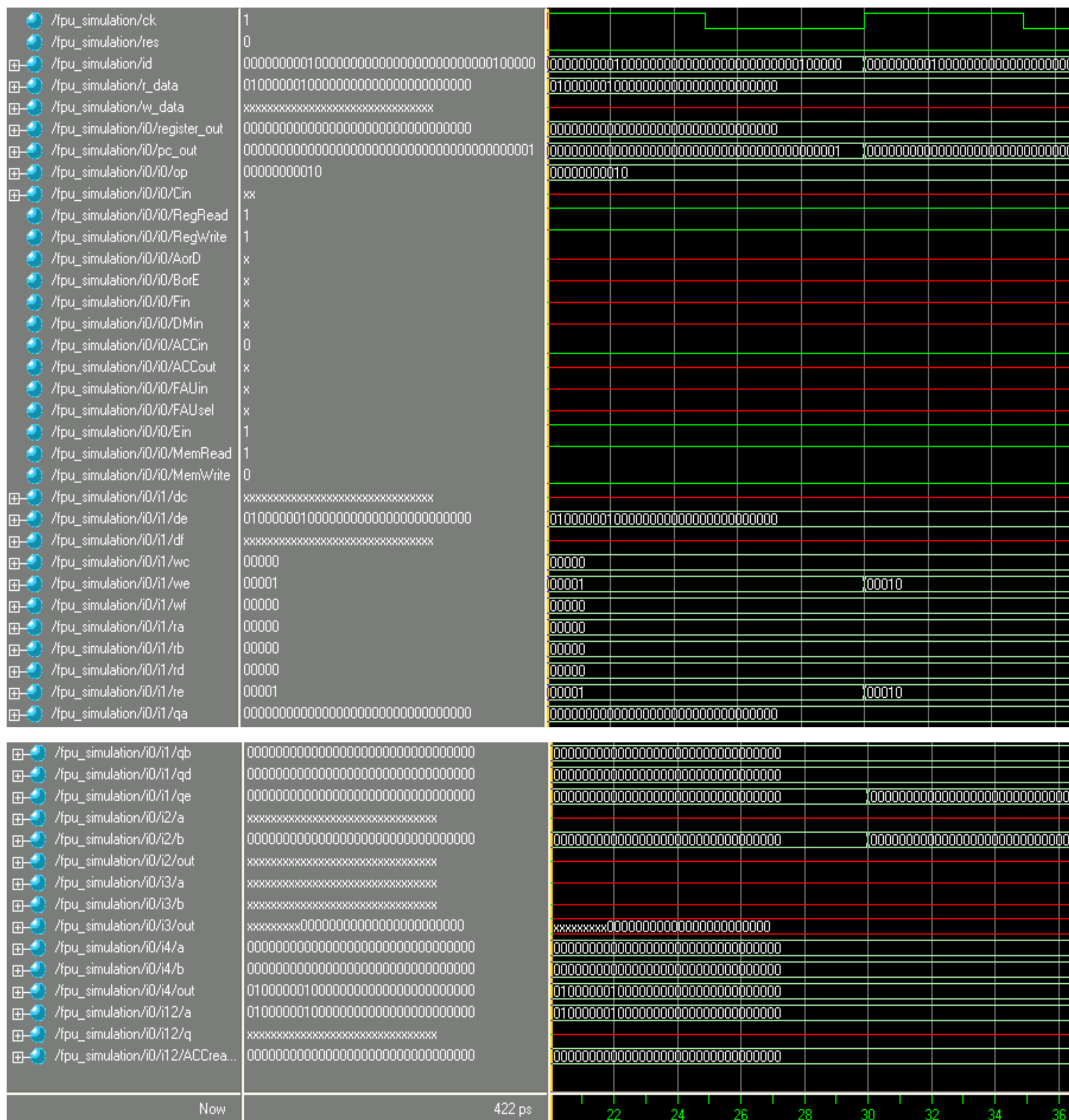


図 4.7　テストベンチのシミュレーション（20ns 時）

20ns のとき、ＰＣが１を指定したため、命令メモリから id[40:0]に対応した命令、ロード命令が実行される。

　ロード命令は読み出すデータのあるアドレスを、データメモリよりレジスタへ転送する命令である。

　データの流れは、id[14:10]→qd→w15→データメモリ→r_data→w17→we（書き込み）となる。

　ここでは行列の要素 a11 をデータメモリからレジスタ１へロードしている。

　以下、180ns までは行列の要素 a12～a44、そしてＸをロードしており、同じロード命令となるため、省略し、次は 190ns から示す。

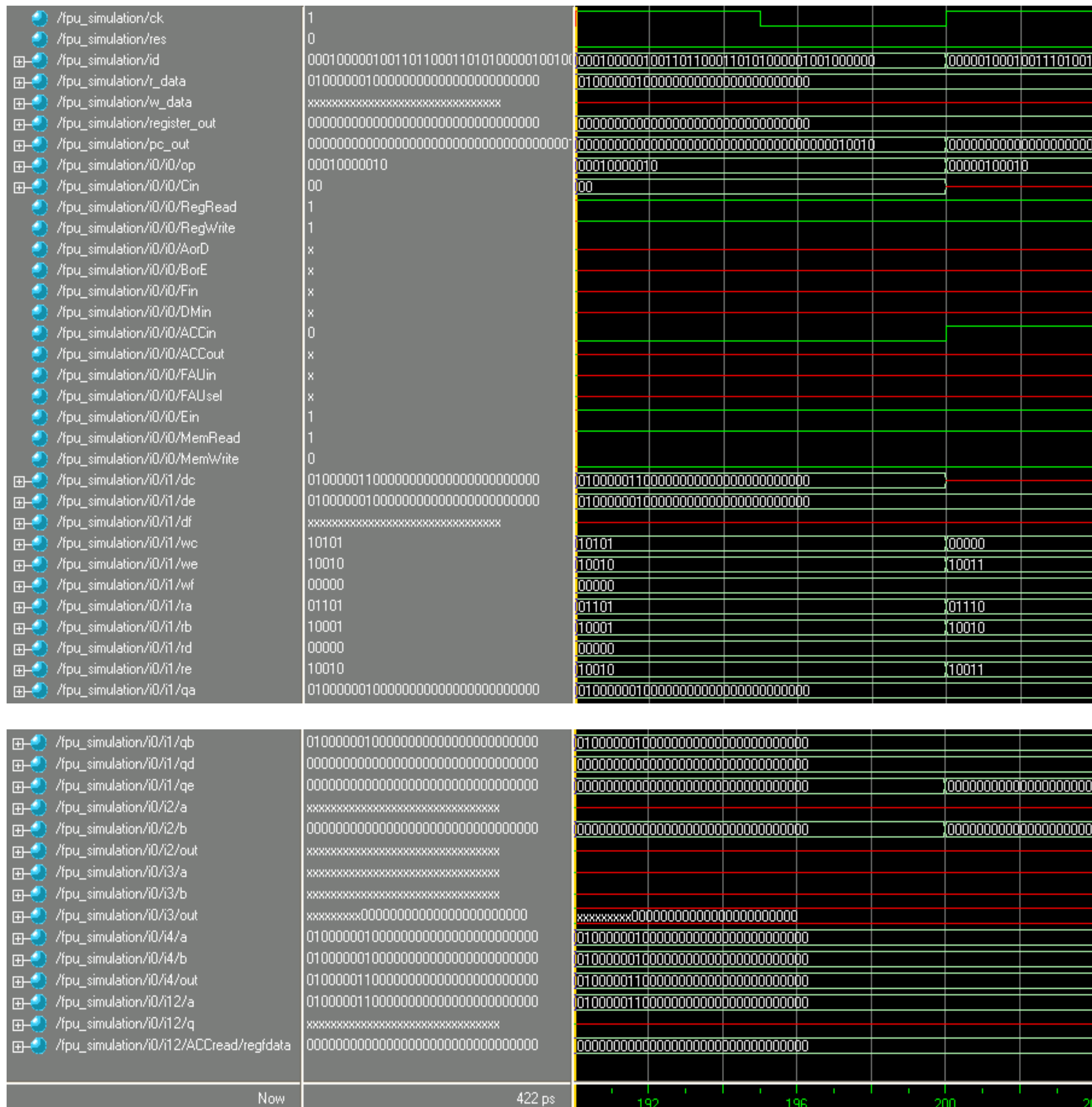


図 4.8　テストベンチのシミュレーション（190ns 時）

190ns のときロード、乗算命令を並行させて行っている。

乗算命令は演算する２つの値を指定されたレジスタより得て、乗算器に入力し、乗算された結果を指定されたレジスタに書き込む命令である。

データの流れは id[24:20]→w12→乗算器、もう一方が id[19:15]→w13→乗算器となり、　演算結果→w22→セレクタ→w14→wc（書き込み）となる。

ここでの乗算は、R21←a41*X を行っている。計算結果は 16、すなわち浮動小数点表示では 01000001100000000000000000000000 がレジスタ２１に入ることとなる。図 4.8 の、dc（書き込みデータ）、wc（書き込みレジスタ）を見ると、正しく出来ていることがわかる。

ロード命令は前フェーズのロードと同様である。

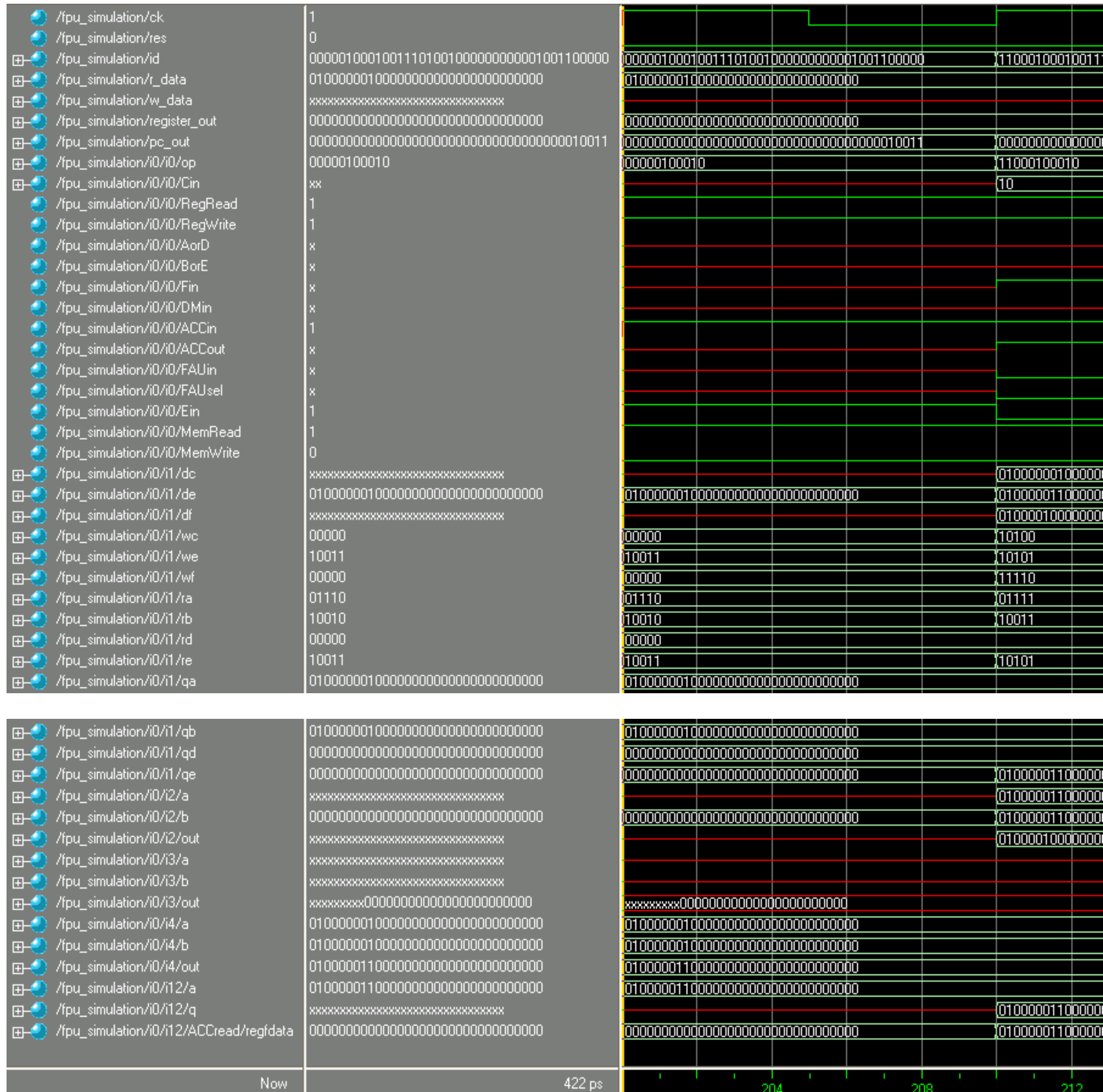


図 4.9　テストベンチのシミュレーション（200ns 時）

200ns のとき、ACC 乗算とロード命令を行っている。

ACC 乗算命令は乗算器に入力するまでは乗算命令と同じだが、乗算結果はレジスタへ入れず、ACC に書き込むという命令である。

データの流れは id[24:20]→w12→乗算器、もう一方が id[19:15]→w13→乗算器となり、　演算結果→w22→ACC（ACC への書き込み）となる。

ここでは　ACC←a42*Y　を行っている。計算結果は 16、すなわち浮動小数点表示で、01000001100000000000000000000000 が ACC に入ることとなる。

図 4.9 より、ACCin=1 で、ACC の入力 a の値を見ると、正しく出来ていることがわかる。

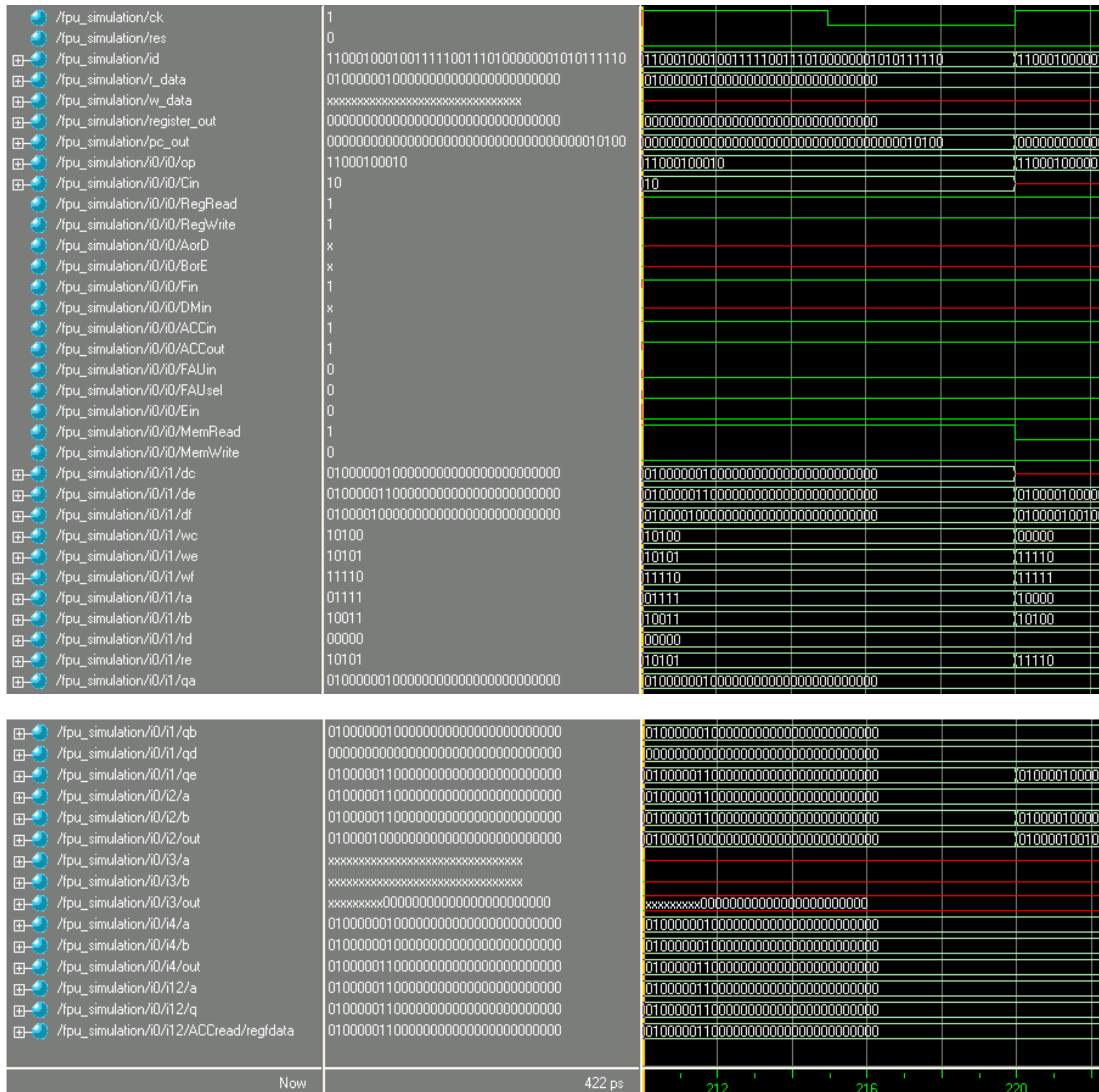


図 4.10　テストベンチのシミュレーション（210ns 時）

210ns のとき、ロード、ACC 加算、ACC 乗算命令を行っている。

ACC 加算命令は演算する２つの値をレジスタより得て、加減算器に入力し、加算された結果を指定されたレジスタに書き込む命令である。

データの流れは ACC→w25→マルチプレクサ→w23→加減算器、もう一方が id[9:5]→w16→加減算器となり、加算結果→w24→マルチプレクサ→w18→wf（書き込み）となる。

ここでは ACC 加算で R30←ACC+R21 を行い、計算結果は 32、すなわち浮動小数点示で 01000010000000000000000000000000 がレジスタ 30 に入ることになる。また、ACC 乗算で ACC←a43*Z を行い、計算結果は 16、すなわち浮動小数点表示で 01000001100000000000000000000000 が ACC に入ることになる。それぞれ図 4.10 で正しいことが確認できる。

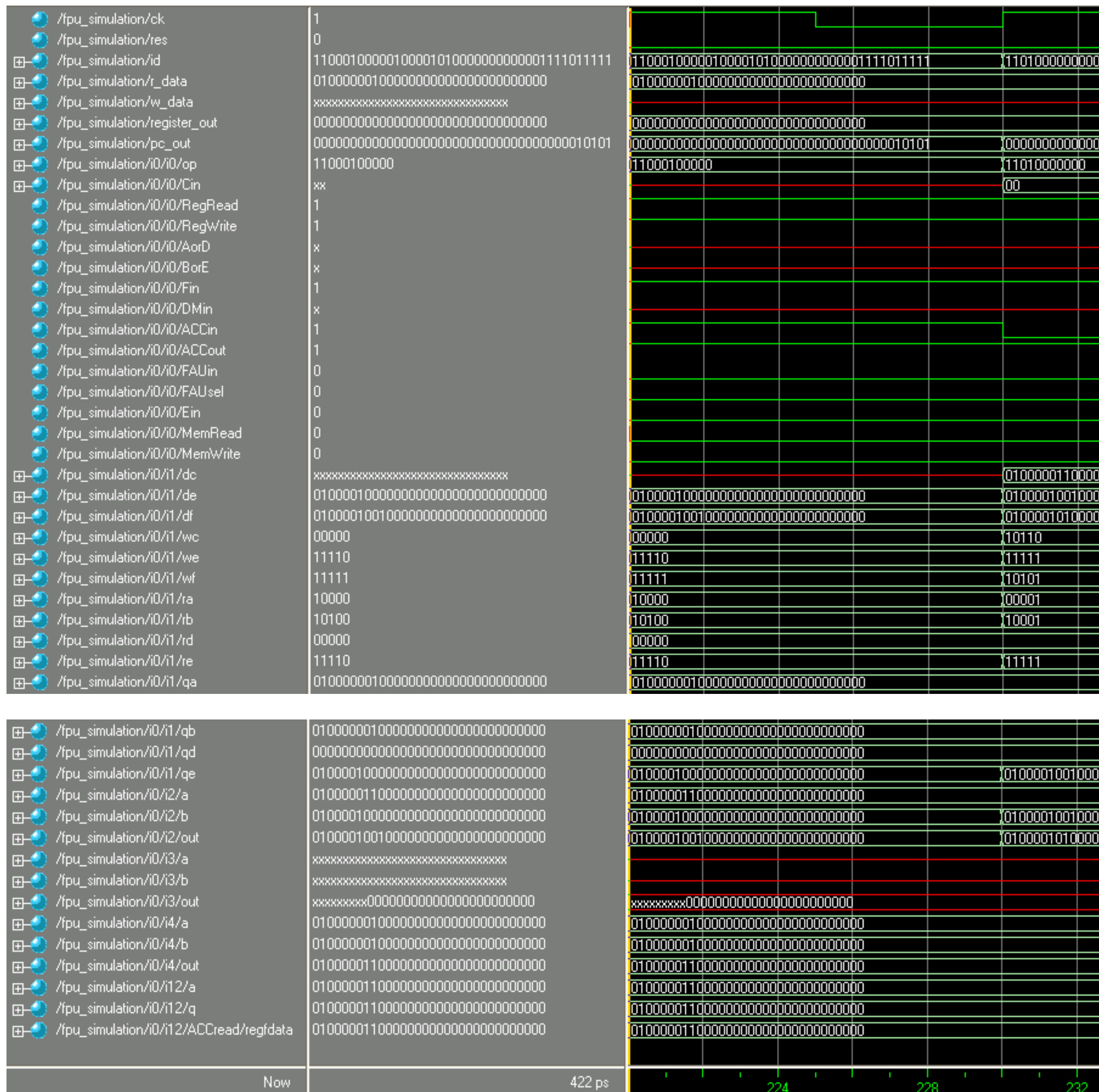


図 4.11　テストベンチのシミュレーション（220ns 時）

220ns のとき、ACC 加算と ACC 乗算を行っている。

ACC 加算命令では R31←ACC+R30 を行っており、計算結果として 48、すなわち浮動小数点表示で 01000010010000000000000000000000 がレジスタ 31 に入ることになる。また、ACC 乗算命令では ACC←a44*W を行っており、計算結果として 16、すなわち浮動小数点表示で 01000001100000000000000000000000 が ACC に入ることになる。

それぞれの計算結果は、ACC 加算命令では df（書き込みデータ）と wf（書き込みレジスタ）を、ACC 乗算命令では ACC の入力 a の値を、それぞれ図 4.11 より確認することで、正しくできていることがわかる。

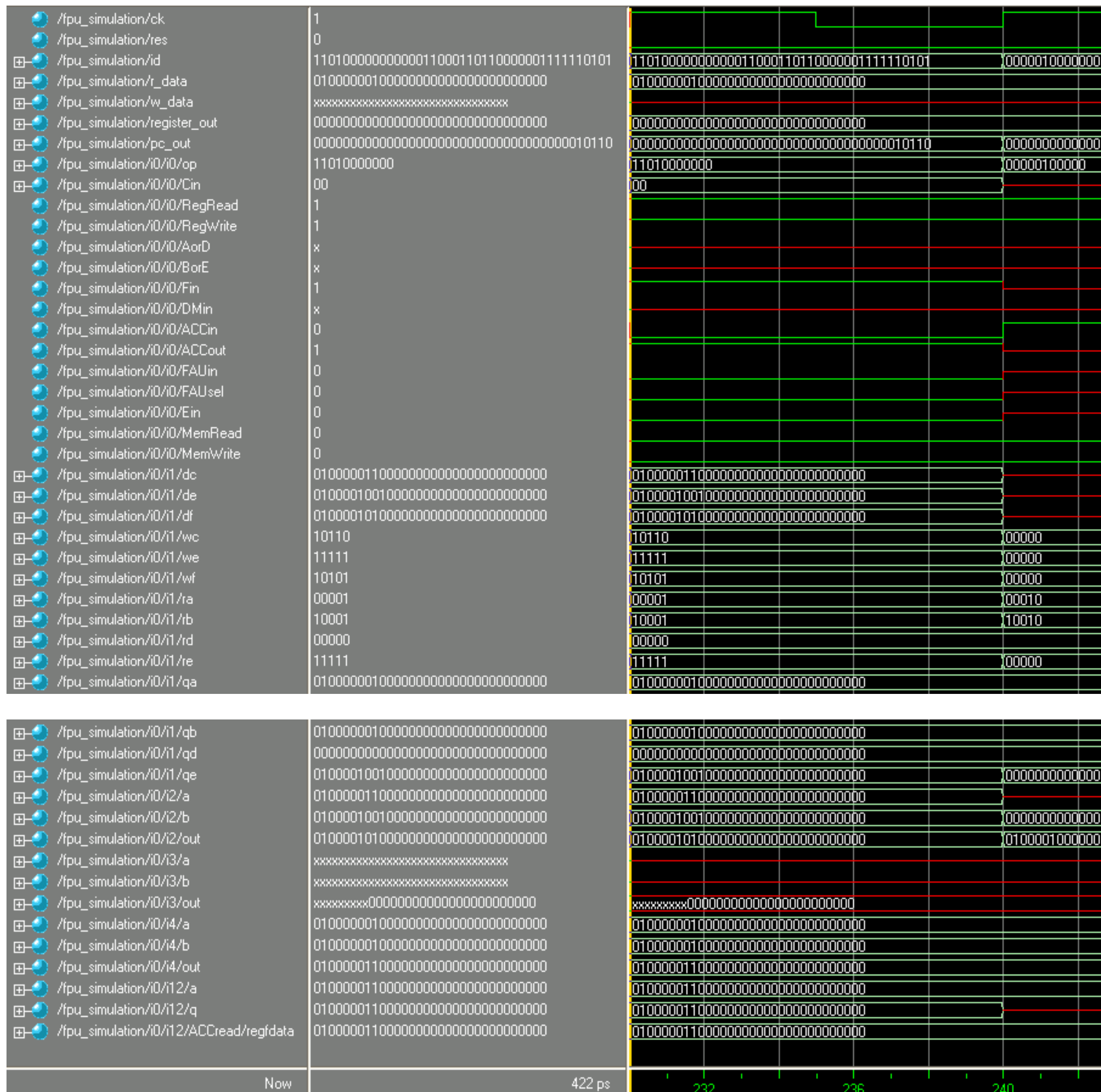


図 4.12　テストベンチのシミュレーション（230ns 時）

230ns のとき、ACC 加算命令と乗算命令を行っている。

ACC 加算命令では R21←ACC+R31 を行っており、計算結果として 64、すなわち浮動小数点表示で 01000010100000000000000000000000 がレジスタ 21 に入ることになる。この値は行列演算の W ’の値となる。また乗算命令では R22←a11*X を行っており、ここからは今までの W ’を求めた計算と同じように X ’を求めていく計算となる。基本的には W ’を求めることと同じことなので、次からの 240ns～260ns までの説明は省略する。

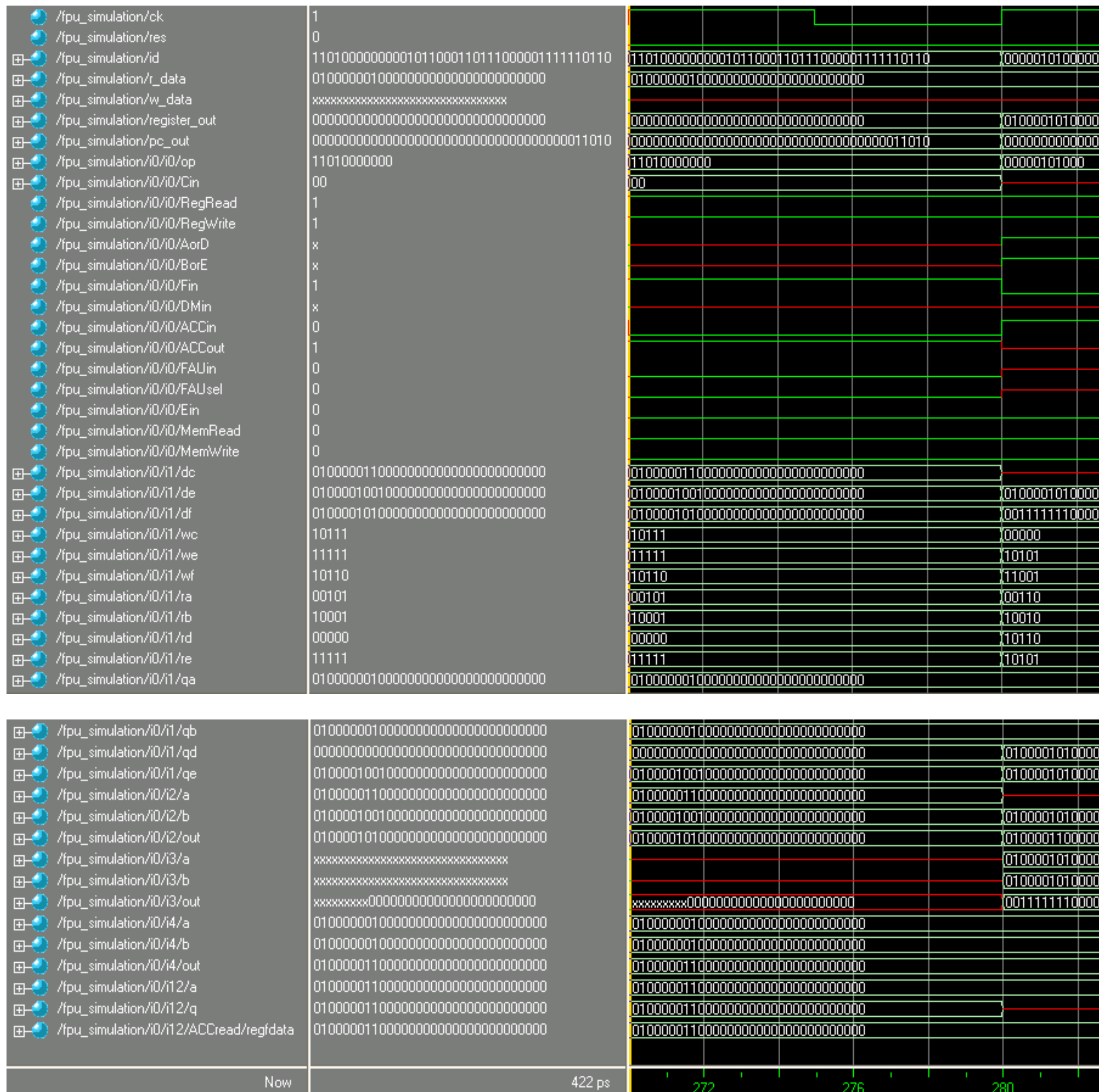


図 4.13　テストベンチのシミュレーション（270ns 時）

270ns のとき、ACC 加算命令と乗算命令を行っている。

ACC 加算命令は R22←ACC+R31 を行っており、計算結果は 64 で、この値が X’の値となる。図 4.13 の df、wf の値より、正しく出来ていることが確認できる。

また、乗算命令では R23←a21*X を行っており、W’、X’と同様に次は Y’を求めていく計算となる。これもまた同じ計算となるので 290ns、300ns の説明は省略している。

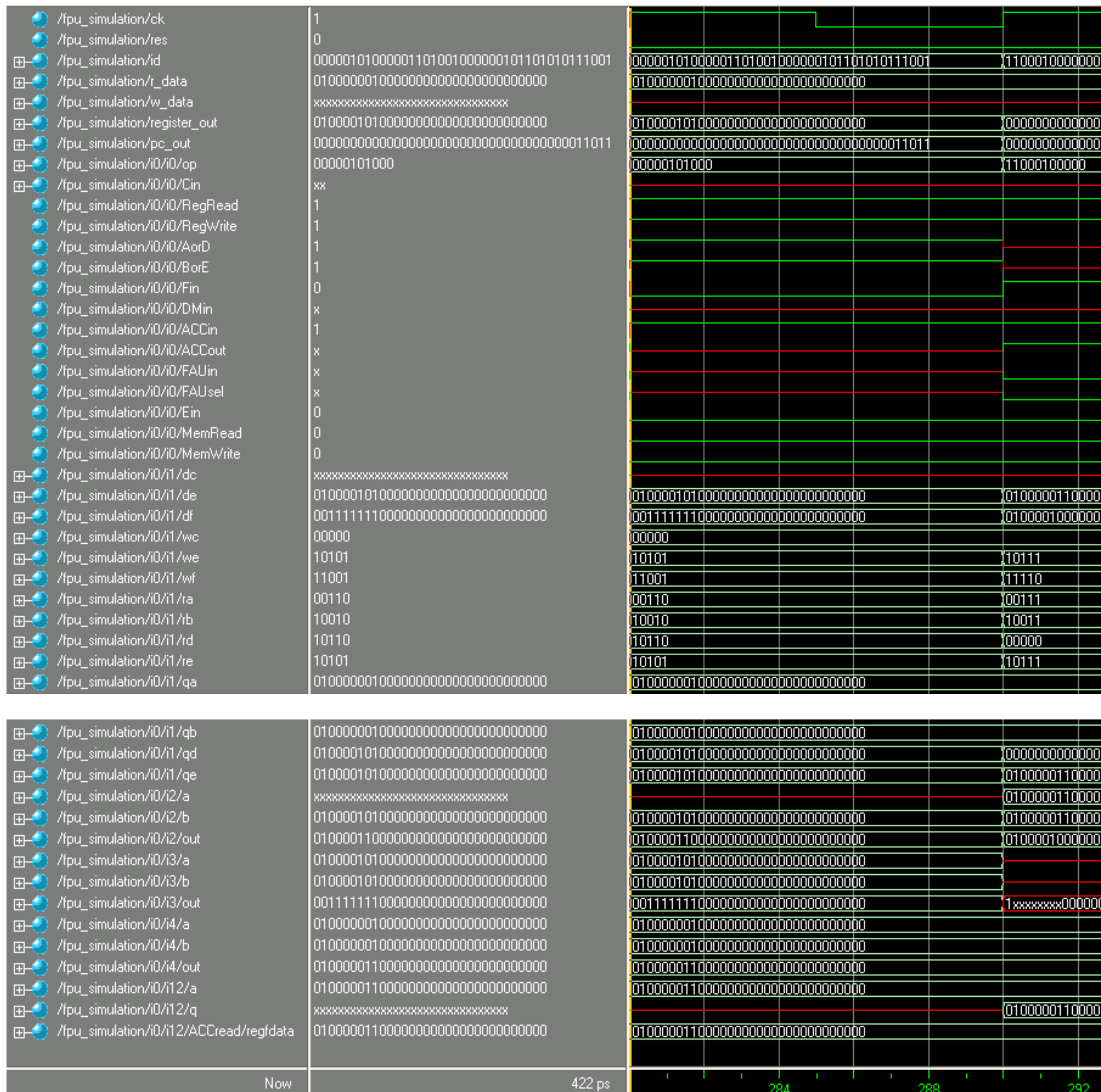


図 4.14　テストベンチのシミュレーション（280ns 時）

280ns のとき、ACC 乗算と除算の開始命令を行っている。

除算の開始命令では W'、X' が計算されたことにより R25←X'/W' を計算している。前章で述べたように、除算の開始、終了命令は正しく動作しない。そこで、ここでは除算の開始命令で除算をし、除算されたデータをレジスタに入れられるようにした。

除算の開始命令のデータの流れは id[14:10]→w15→マルチプレクサ→w19→除算器、もう一方が id[9:5]→w16→マルチプレクサ→w20→除算器、そして、計算結果→w21→マルチプレクサ→w18→wf（書き込み）となる。

計算結果は 1、浮動小数点表示では 00111111100000000000000000000000 が df に入っていることが、図 4.14 より確認できる。

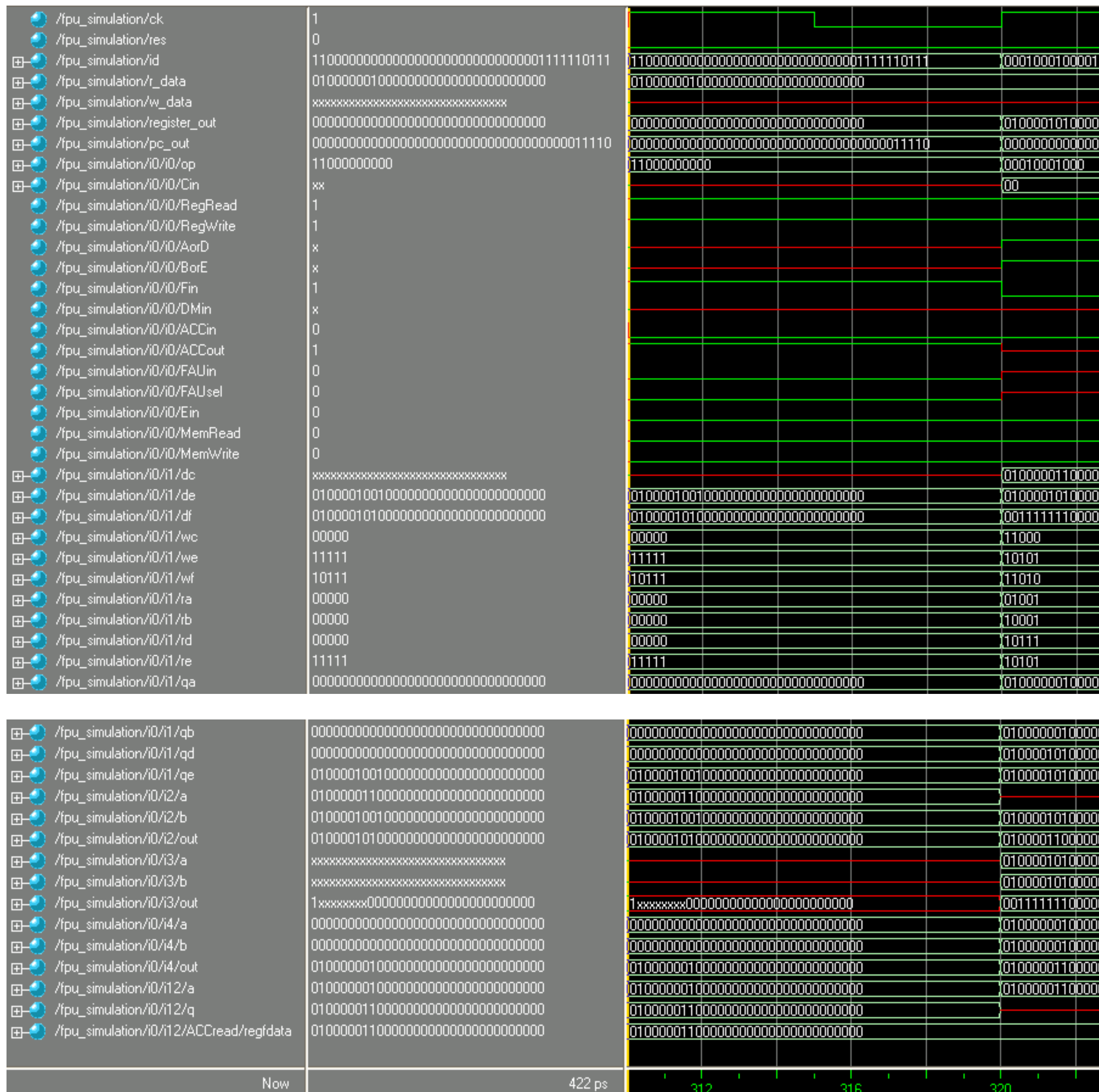


図 4.15　テストベンチのシミュレーション（310ns 時）

310ns のとき、ACC 加算のみ行っている。

ここでは R23←ACC+R31 を計算しており、計算結果は 64、すなわち浮動小数点表示で 01000010100000000000000000000000 がレジスタ 23 に入ることとなる。

図 4.15 より、df（書き込みデータ）、wf（書き込みレジスタ）を見ると正しくできていることが確認できる。この値が Y'の値となる。

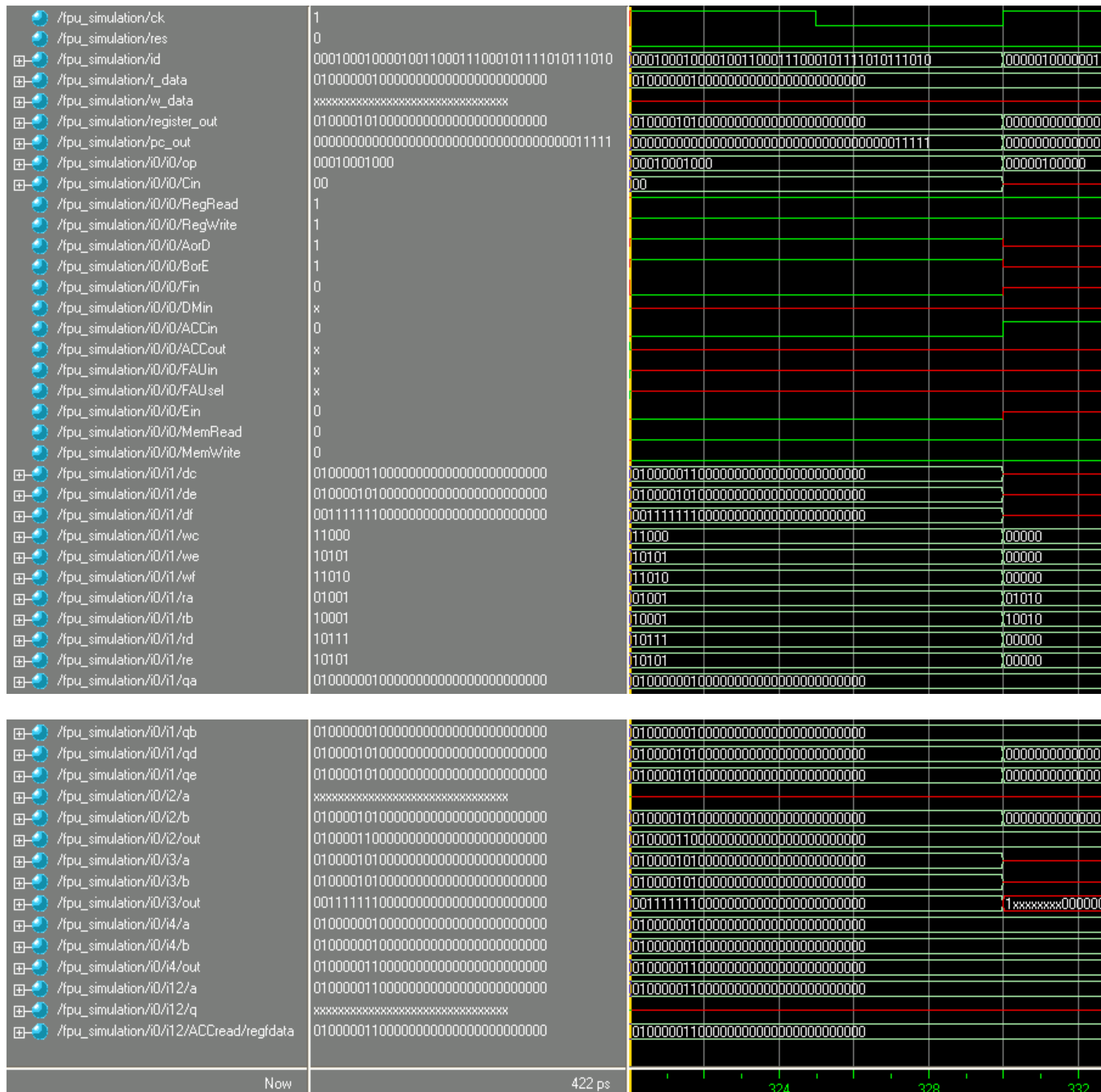


図 4.16　テストベンチのシミュレーション（320ns 時）

320ns のとき、乗算命令と除算の開始命令を行っている。

除算の開始命令は R26←Y’/W’を行っており、計算結果は 1、浮動小数点表示で 00111111100000000000000000000000 がレジスタ 26 に入ることになる。

図 4.16 より、df（書き込みデータ）、wf（書き込みレジスタ）の値を見ることで、正しく出来ていることが確認できる。

乗算命令は R24←a31*X を行っており、これは Z’を求めていく過程となる。これも W’と同じ計算を行うこととなるので 330～350ns の説明は省略する。

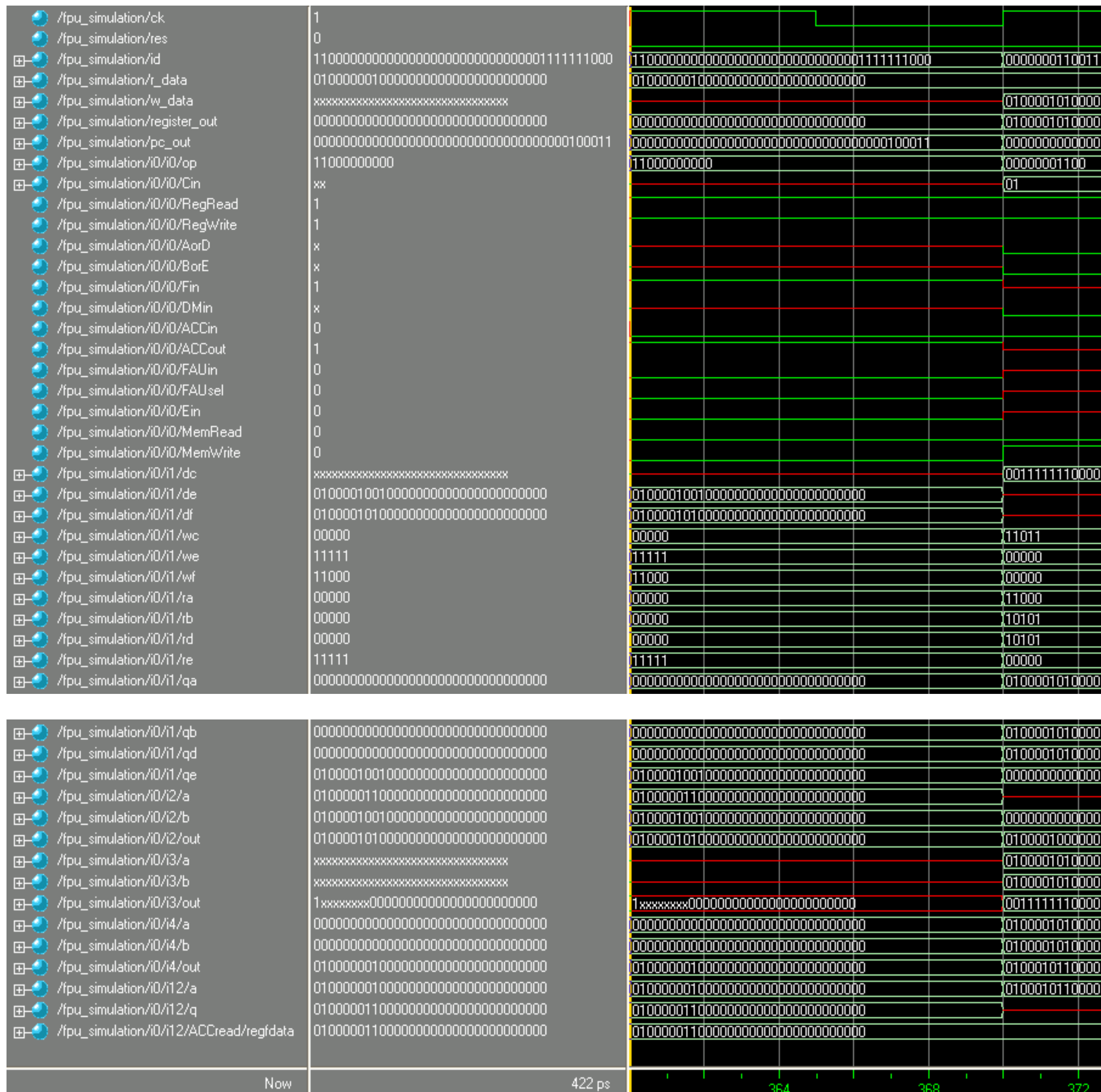


図 4.17　テストベンチのシミュレーション（360ns 時）

360ns のとき、ACC 加算を行っている。

ここでは R24←ACC+R31 を行っており、計算結果は 64、すなわち浮動小数点表示で 01000010100000000000000000000000 がレジスタ 24 に入ることとなる。

図 4.17 より df（書き込みデータ）、wf（書き込みレジスタ）を見ると正しく出来ていることが確認できる。この値が Z’の値となる。

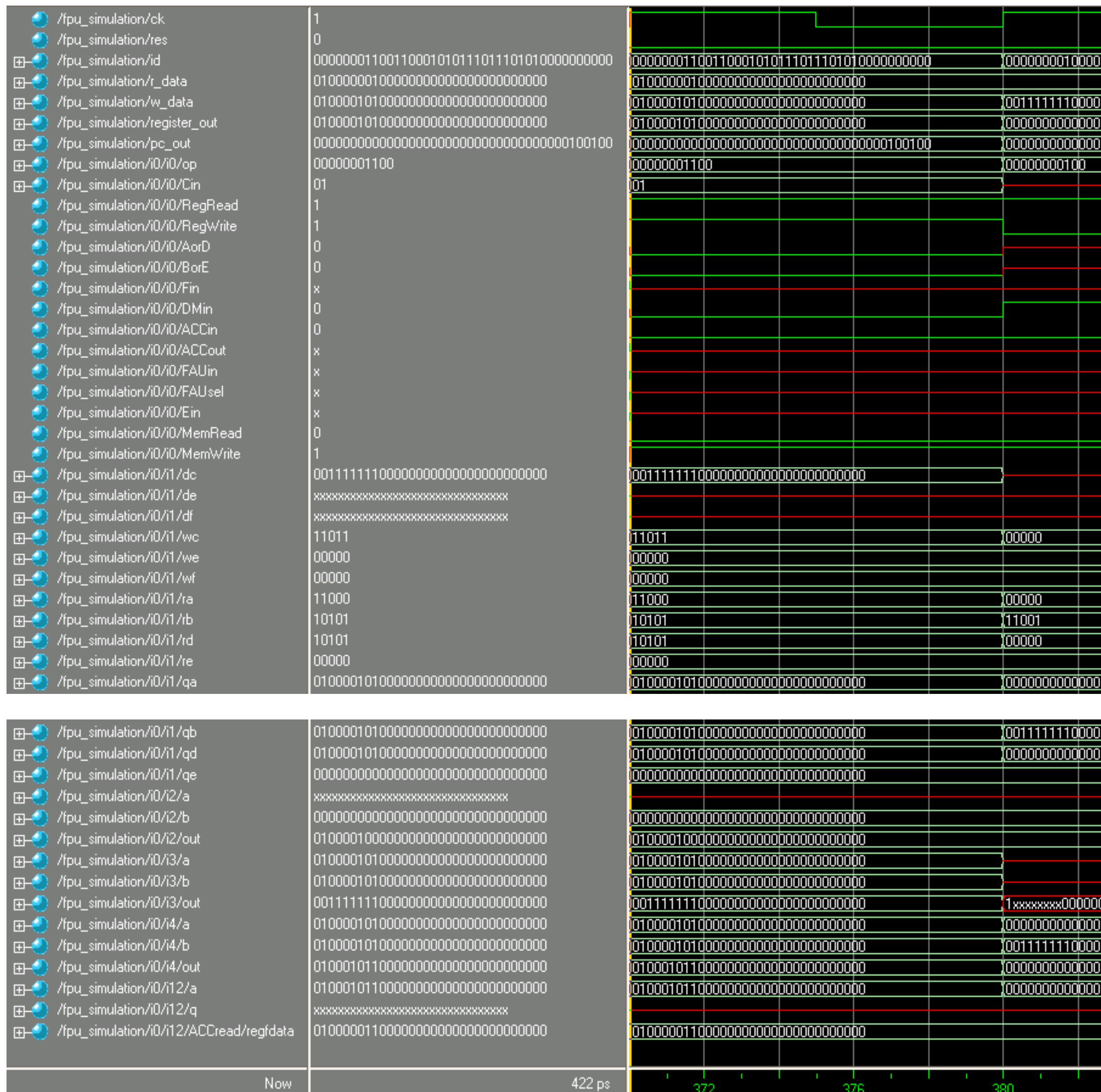


図 4.18　テストベンチのシミュレーション（370ns 時）

370ns のとき、除算の開始とストアを行っている。

除算の開始命令は、R27←Z’/W’を行っており、計算結果は 1、すなわち浮動小数点表示で 00111111100000000000000000000000 がレジスタ 27 に入ることとなる。

図 4.18 より dc、wc の値を見ることで正しく出来ていることが確認できる。

また、ストア命令とは、ストアするアドレスをレジスタからデータメモリへ転送する命令のことである。データの流れは、id[14:10]→w15→マルチプレクサ→w_data→データメモリ（書き込み）となる。

ここでは W’の値をストアしているので、浮動小数点表示で 01000010100000000000000000000000（10 進数：64）が w_data に正しく入っていることが、図 4.18 より確認できる。

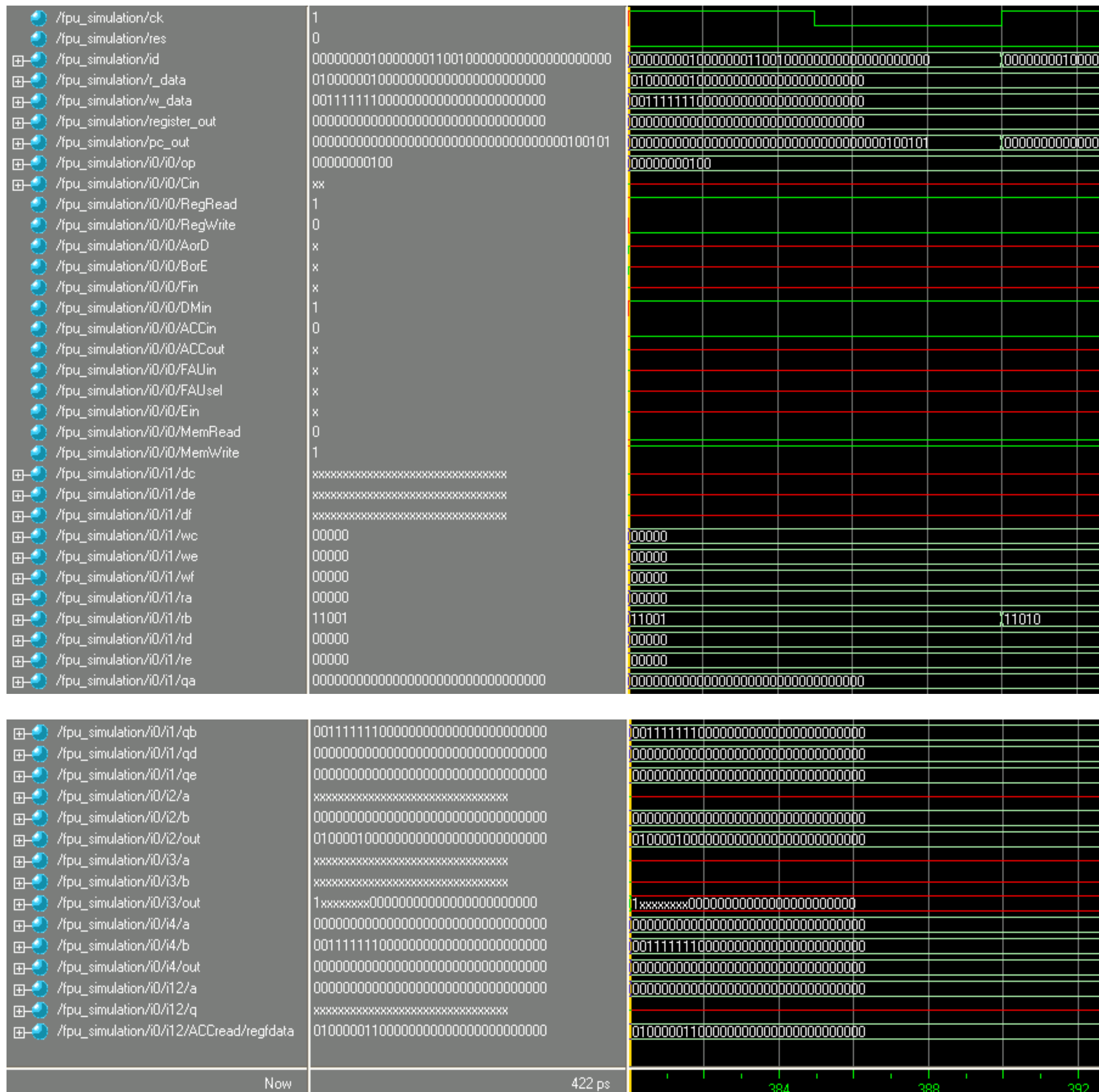


図 4.19　テストベンチのシミュレーション（380ns 時）

380ns のとき、ストア命令を行っている。

ここでは X’/W’の値、00111111100000000000000000000000（10 進数：1）をデータメモリにストアしている。

図 4.19 より w_data の値を見ると正しく出来ていることがわかる。

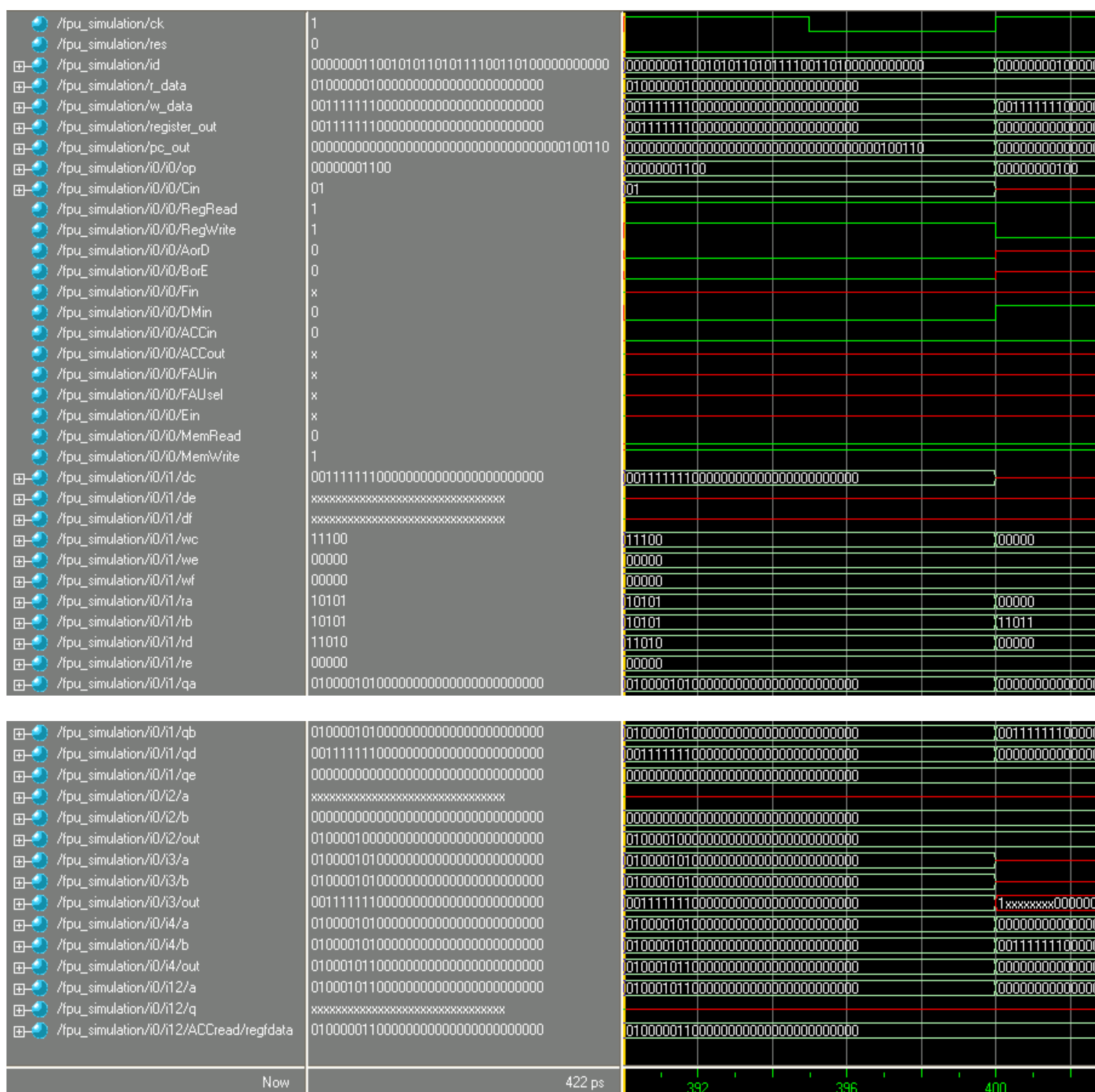


図 4.20　テストベンチのシミュレーション（390ns 時）

390ns のとき、除算の開始、ストア命令を行っている。

除算の開始命令は、R28←W’/W’を行っており、計算結果は 1、すなわち浮動小数点表示で 00111111100000000000000000000000 がレジスタ 28 に入ることとなる。

図 4.20 より dc（書き込みデータ）、wc（書き込みレジスタ）の値を見ることで正しく出来ていることが確認できる。

また、ストア命令はここでは Y’/W’の値をストアしているので、浮動小数点表示で 00111111100000000000000000000000（10 進数：1）がデータメモリにストアされる。

図 4.20 より w_data を見ると、正しく出来ていることがわかる。

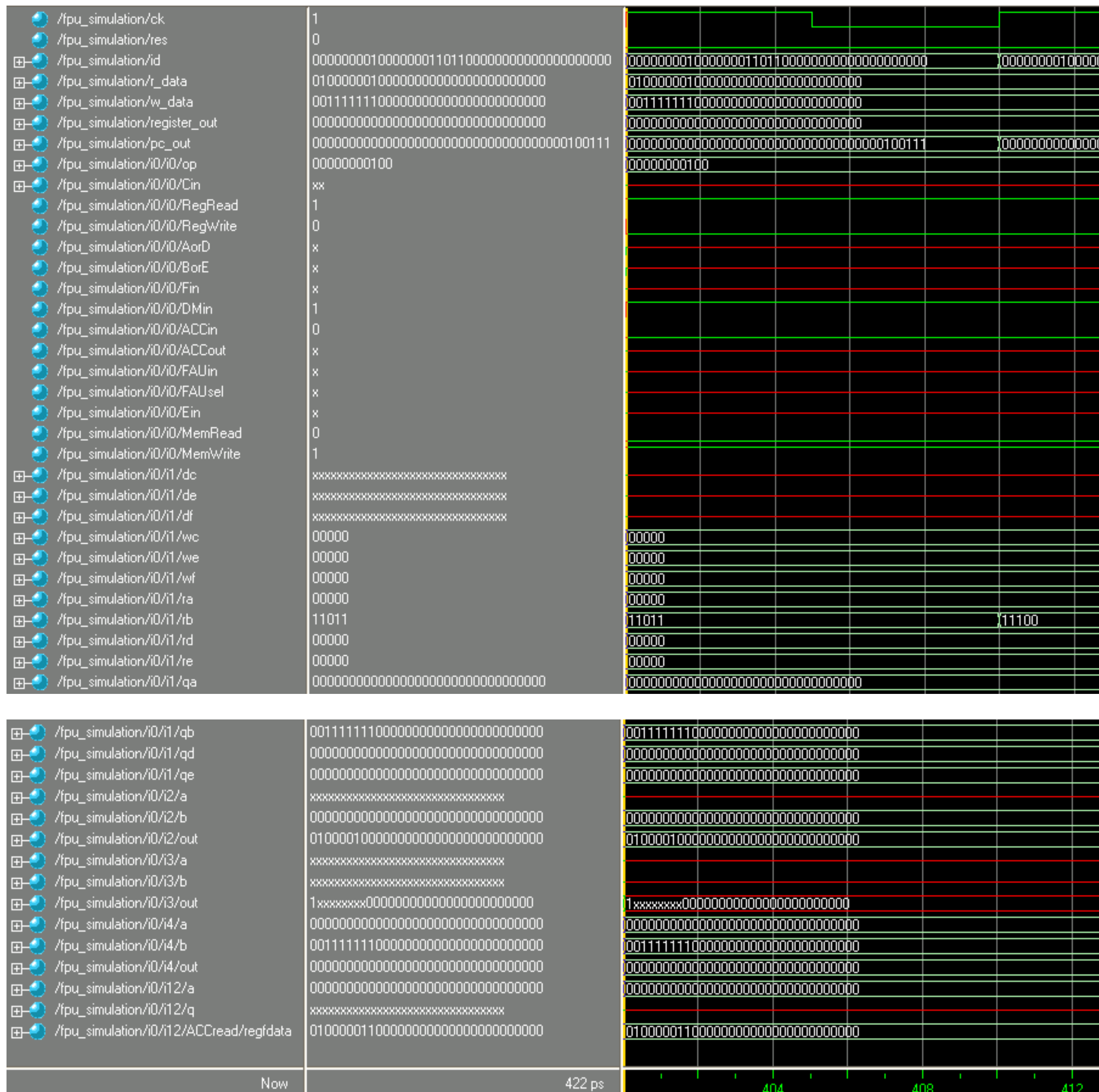


図 4.21　テストベンチのシミュレーション（400ns 時）

400ns のとき、ストア命令を行っている。

ここでは Z’/W’の値、浮動小数点表示で 00111111100000000000000000000000（10 進数：1）をデータメモリへとストアしている。

図 4.21 の w_data の値を見ると、正しく出来ていることが確認できる。

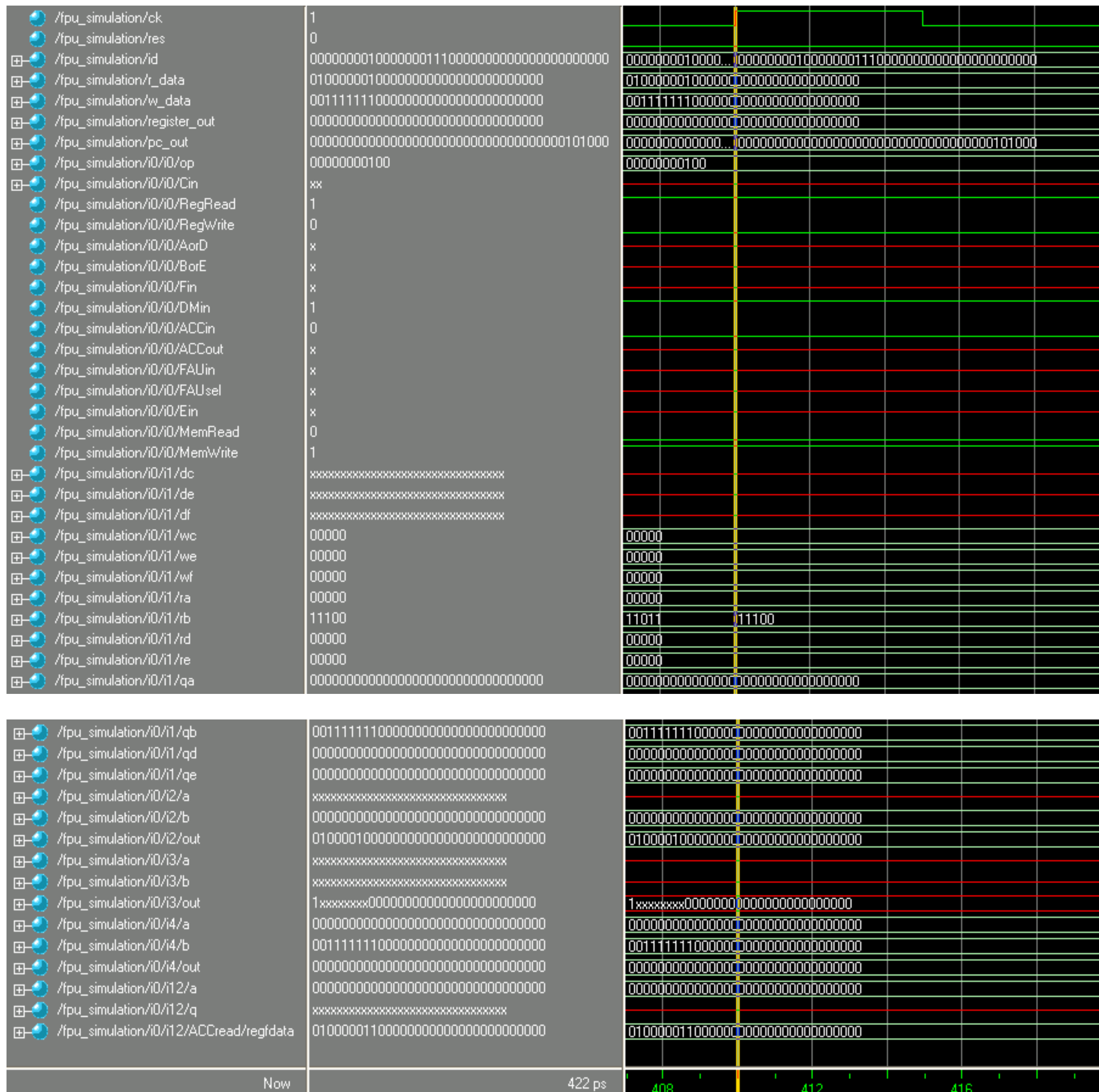


図 4.22　テストベンチのシミュレーション（410ns 時）

410ns のとき、ストア命令を行っている。

ここでは W’/W’の値、浮動小数点表示で 00111111110000000000000000000000（10 進数：1）をデータメモリへとストアしている。

図 4.21 の w_data の値を見ると、正しく出来ていることが確認できる。

以上、図 4.6～図 4.22 まで行列演算を行ってきたが、370ns～410ns までストアした値が、図 4.5 の結果と一致したことよりシミュレーション結果が正しいことが確認できた。

以上で、FPU の今回使用した命令のシミュレーションを完了した。

4.3.2　性能評価

　4×4 の行列演算をシミュレーションとして行った。アキュムレータの取り付けと制御信号 Ein の追加によってロード・ACC 加算・ACC 乗算命令の並行が可能になったこと、
また、積和演算を多用することによって、今回改良したＦＰＵでは５７個の浮動小数点演算を４０サイクルで実行できるという結果が出た。

おわりに

本研究では、HDL による FPU の設計ということで、FPU の HDL 記述、シミュレーション、FPGA での論理合成、そして配置配線を行い FPGA に書き込むデータ作成までをした。

今後の課題として、加減乗除・ロード・ストア以外の命令の追加、ハードウェアのリソース量を考慮し効率の良い回路、つまり if 文や case 文を論理式に変換することにより回路のコンパクト化をねらうこと、などがあげられる。同研究室の研究（アセンブラ解析）らも性能の向上が可能だと思われる。

参考文献

【１】林成憲　町田慎弥　李天慧　國信茂郎
HDL（ハードウェア記述言語）を用いた浮動小数点プロセッサの設計
【２】パターソン＆ヘネシー　日系 BP 社：
コンピュータの構成と設計　ハードウェアとソフトウェアのインターフェース
第 2 版　上・下
【３】株式会社エッチ・ディー・ラボ：
HDL Endeabor Verilog-HDL
【４】原田豊著　丸善株式会社：
論理回路と計算機ハードウェア
【５】國信茂郎
冗長２進演算アルゴリズムと高速プロセッサの実現に関する研究