卒業論文
SFL 開発環境を用いた VAX-11/780 命令互換 32 ビットプロセッサの開発

近藤信行 †

† 東海大学コミュニケーション工学科
Email:1adt2415@keyaki.cc.u-tokai.ac.jp

概要

コンピュータシステムの包括的教育材料として VAX-11/780 命令互換 32 ビットプロセッサコアの開発と評価を行った。SFL を用いて論理設計を行い、Icarus-Verilog と VAX 用 gcc,DHRYSTONE BENCHMARK を用いて評価した。本稿では、今回の開発の過程と性能評価の詳細を報告するとともに、一連の開発による教育目的について述べる。

Development of the VAX-11/780 instruction compatible 32bit processor using SFL development environment

Nobuyuki Kondoh†

†School of Information Technology and Electronics, Tokai University.
Email:1adt2415@keyaki.cc.u-tokai.ac.jp

Abstruct

I developed the VAX-11/780 instruction compatible 32bit processor as teaching materials of a computer system , and evaluated the performance of the processor. I designed logic using SFL and simulated with Icarus-Verilog. The binary data for debugging was generated with gcc for VAX-11,and evaluated the developed processor using DHRYSTONE BENCHMARK. I report the development of the processor in detail, and the performance of the processor.

目次

1 はじめに 3
- 1.1 開発目的 . . . 3

2 開発環境と開発手順 3
- 2.1 開発ツールについて . . . 3
- 2.2 開発手順 . . . 3

3 VAX11/780 のアーキテクチャ 4
- 3.1 データタイプ . . . 4
- 3.2 レジスタ . . . 5
- 3.3 命令ストリーム形態 . . . 6
  - 3.3.1 可変長命令 . . . 6
  - 3.3.2 オペコード . . . 6
  - 3.3.3 オペランド . . . 6
  - 3.3.4 アドレッシングモード . . . 6
- 3.4 メモリ管理機構 (MMU: Memory Manegement Unit) . . . 10
  - 3.4.1 MMU とは . . . 10
  - 3.4.2 メモリマッピング (アドレス変換) 11
  - 3.4.3 TLB . . . 11
  - 3.4.4 メモリ保護 . . . 11
- 3.5 命令セット . . . 11

4 開発互換プロセッサのシステムアーキテクチャ 12
- 4.1 設計思想 . . . 12
  - 4.1.1 システム構成 . . . 12
  - 4.1.2 データパス . . . 12
- 4.2 互換プロセッサファイル構成 . . . 13
- 4.3 命令キュー (Instruction Queue) . . . 14
  - 4.3.1 命令キューとは . . . 14
  - 4.3.2 実装理由 . . . 14
  - 4.3.3 キューのサイズ . . . 14
  - 4.3.4 動作仕様 . . . 15
- 4.4 メモリ管理機構 (MMU: Memory Manegement Unit) . . . 16
  - 4.4.1 動作仕様 . . . 16
- 4.5 処理フロー . . . 16
  - 4.5.1 (1). 初期化 (Initialize) 動作 . . . 16
  - 4.5.2 (2). 命令フェッチ . . . 16
  - 4.5.3 (3). 命令デコード . . . 17
  - 4.5.4 (4). オペランドフェッチ . . . 17
  - 4.5.5 (5). 命令実行 . . . 17
  - 4.5.6 (6). ライトバック . . . 17
- 4.6 命令フェッチ動作 . . . 17
  - 4.6.1 命令フェッチとは . . . 17
  - 4.6.2 動作仕様 . . . 17
- 4.7 命令デコード動作 . . . 17
  - 4.7.1 命令デコードとは . . . 17
  - 4.7.2 動作仕様 . . . 18
- 4.8 オペランドフェッチ動作 . . . 18
  - 4.8.1 オペランドフェッチとは . . . 18
  - 4.8.2 アドレッシングモードデコードとは . . . 19
  - 4.8.3 オペランドフェッチ動作について 19
  - 4.8.4 仮オペランドの取得 . . . 19
  - 4.8.5 アドレッシングモードデコードの処理フロー . . . 20
  - 4.8.6 アドレッシングモードデコードにおける各ステートの概要説明 20
  - 4.8.7 各アドレッシングモードごとのデコード説明 . . . 22
- 4.9 命令実行処理 . . . 24
  - 4.9.1 命令実行とは . . . 24
  - 4.9.2 動作仕様 . . . 24
  - 4.9.3 命令実装状況 . . . 26
- 4.10 ライトバックステージの処理について . 26
  - 4.10.1 ライトバックとは . . . 26
  - 4.10.2 動作仕様 . . . 26

5 性能評価 26
- 5.1 Quartus2 による論理合成結果 . . . 26
- 5.2 DHRYSTONE BENCHMARK による命令処理効率の評価 . . . 27
- 5.3 命令キューのエントリ数によるメモリアクセス効率の評価 . . . 28

6 開発フロー 29

7 今後の開発予定 30

8 教育的効果 30

9 まとめ 31

1　はじめに

1.1　開発目的

　コンピュータシステム教育において重要なのは、包括的にシステム全体を理解させることにある。システム全体の理解というのは

- プロセッサがもつ各種命令の動作
- プロセッサを含めたハードウェア動作
- 高級言語からマシン語に至るまでの過程
- OS の動作

などを理解することある。我々の研究室では従来からプロセッサ設計をエンジニア教育の一環として行ってきており、ある程度の複雑性をもつプロセッサコアの開発はエンジニア教育という点において、優れた効果を挙げることが実証できたと考えている。そこで今回は、ＯＳを含めたコンピュータシステムの構築を目的とし、VAX-11/780 命令互換 32 ビットプロセッサの開発を行った。VAX-11/780 命令互換とした主な理由として以下のことが挙げられる。

1.VAX-11/780 は 32 ビット CISC プロセッサであること。
複雑な命令やアドレッシングモードを多数もつ 32 ビット CISC プロセッサは多様な命令動作やアドレッシングモードデコードの仕組みを理解させることができる。

2. 高級言語からマシン語に変換する手段がすでに存在する。
gcc において VAX はサポートされているため、gcc を用いてＣ言語→アセンブリ言語→バイナリデータにすることが可能である。これにより高級言語からマシン語に至る過程を知ることができ、シミュレーションデバッグの際の命令データを容易に用意することができる。

3. 動作が確認されている OS が存在する。BSD 系 UNIX はもともと PDP-11 後継である VAX-11 ファミリーをターゲットとして作られていたため、VAX 上でネイティブに動作する。特に GCC が動くというのは、デバッグ時におけるエラー箇所の特定を行いやすいという点からみて非常に有益である。

2　開発環境と開発手順

2.1　開発ツールについて

　プロセッサコアの開発については以下のツールを使用した。

- SFL
  - 純国産かつ習得が比較的容易なハードウェア記述言語
- sfl2vl
  - SFL から論理合成可能な Verilog ソースコードに変換するプログラム
- Icarus Verilog
  - オープンソースの Verilog コンパイラ＆シミュレータ
- GCC 3.3.4 for VAX
  - VAX 用にコンパイルした GCC3.3.4
- Quartus2 ver4.1
  - ALTERA 社の FPGA 用論理合成ツール

2.2　開発手順

　開発は以下の手順で行った。

　1.SFL を用いて論理設計を行う。

　2.sfl2vl に通して VerilogHDL に変換する。

　3.Icarus Verilog を用いて、シミュレーションを行う。シミュレーション時に利用する命令データは C 言語で記述したプログラムを VAX 用 GCC にてコンパイルし、それを awk スクリプトでシミュレーションの際に

データタイプ	サイズ	範囲	
整数データ		符号付き	符号無し
バイト	8 ビット	-128〜127	0〜255
ワード	16 ビット	-32767〜32767	0〜65535
ロングワード	32 ビット	-2^{31}〜2^{31}	0〜$2^{32}-1$
クォドワード	64 ビット	-2^{63}〜2^{63}	0〜2^{63-1}
オクタワード	128 ビット	-2^{127}〜2^{127}	0〜2^{127-1}
浮動小数点データ		精度	
F浮動小数点	32 ビット	10 進数で 7 桁	
D浮動小数点	64 ビット	10 進数で 16 桁	
G浮動小数点	64 ビット	10 進数で 15 桁 (指数部に多くのビットを割り当てるため)	
H浮動小数点	128 ビット	10 進数で 33 桁	
特殊タイプデータ		範囲と備考	
パック 10 進ストリング	0〜16 バイト (31 桁)	数値は 1 バイトあたり 2 桁使用。 符号は下位 4 ビットで表現	
キャラクタストリング	0〜65535 バイト	1 バイトあたり 1 文字	
可変長ビットフィールド	0〜32 ビット	解釈方法により異なる	
パック 10 進ストリング	0〜31 バイト (0〜31 桁まで)	$-10^{31}-1$〜$10^{31}-1$	
パック 10 進ストリング	$\geq$ 2 つのロングワード ÷ 待ち行列要素	要素数は 0〜20 億まで	

表 1: VAX11/780 のデータタイプ一覧

利用可能なメモリデータに整形したものを使用した。

4.Quartus2 にてターゲットを Stratix として論理合成して回路規模と動作周波数を確認し、ソースに対して修正を加えて回路規模の縮小と動作周波数の向上を図る。

3 VAX11/780 のアーキテクチャ

ここでは現在までに設計が終了している部分に限定し、VAX11/780 のアーキテクチャについて説明する。

3.1 データタイプ

VAX が扱うデータタイプは、整数データであるバイト, ワード, ロングワード, クォドワードの 4 種類 1 バイトは 8 ビットであり、1 ワードは 2 バイト、1 ロングワードは 4 バイト、1 クォドワードは 8 バイトとなる。浮動小数点データは F 浮動小数点, D 浮動小数点の 2 種類存在する。そして特殊タイプとして、以下の 5 種類がある。

パック 10 進数字列 ....
数字の各桁を 1 ニブル (4 ビット) を使って表すデータ

キャラクタストリング ....
文字を ASCII コードにし、1 バイトを使って表すデータ

可変長ビットフィールド ....
大きなデータ構造の中でパッキングされた一種の小さな整数データ実際は単なるビットデータの集合体で、利用する際にアドレスを指定することで目的の値を取得する必要がある。

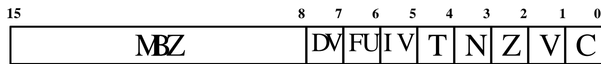

図 1: プロセッサステータスワード (PSW)

数字列 ‥‥

数字を ASCII コードにし、1 バイトを使って表すデータ

待ち行列 ‥‥

2 重にリンクされた循環リストからなるデータ構造

これらのデータタイプをまとめ、表 1 に示す。表 1 に載せた整数データのオクタワード、浮動小数点データのＧ浮動小数点とＨ浮動小数点についてはＶＡＸファミリのプロセッサオプションによって使用可能となる。プロセッサオプションによってサポートされるデータタイプについては、その特殊性や使用用途、これらのデータタイプをサポートするコンパイラなどが限定されることから、今回の互換プロセッサではこれらのオプションデータタイプは未実装とした。

3.2 レジスタ

VAX11/780 は 32 ビット汎用レジスタとしてＲ 0〜Ｒ 15 の 16 個をもつ。この中で特にＲ 12〜Ｒ 15 までをそれぞれ引数ポインタ (AP), フレームポインタ (FP), スタックポインタ (SP), プログラムカウンタ (PC) として利用する。この他にプロセッサステータスロングワード (PSL) というプロセッサの状態を示す 32 ビットレジスタが 1 つある。PSL の下位 15 ビットを特にプロセッサステータスワード (PSW) と呼び、特権がなくとも自由に参照可能な情報が格納されている。図 1 に PSW のビット構成を示す。PSW 内には条件コードビットやトラップビットなどが存在する。それらの詳細について以下に示す。

ビット＜３：０＞

PSW のビット 0 からビット 3 までは条件コードと呼ばれている。これらのビットは一般的に最後に実行された命令の結果を示す。条件コードは条件つきの分岐命令などで使用される。

- **Ｎビット**

  Ｎビットは負条件コードである。一般的にこのコードは命令の実行結果が負の場合にセットされる。結果が正またはゼロのときはクリアされるこのビットはオーバーフローが発生して実行結果の符号が誤った結果となっても、正しい結果に従ってセット, クリアされる。

- **Ｚビット**

  Ｚビットはゼロ条件コードである。基本的にこのコードは命令の実行結果が正確にゼロの場合にセットされその他の場合にはクリアされる。このビットもＮビットと同じくオーバーフローが発生して実際の結果がおかしい場合でも、本当の実行結果がゼロであるような場合にはセットされる。

- **Ｖビット**

  Ｖビットはオーバーフロー条件コードである。算術演算でオーバーフローを起こした際にセットされる。オーバーフローがおきなかった場合はクリアされる。オーバーフローが発生しないもしくはオーバーフローが意味を持たない算術演算命令、算術演算以外の命令ではクリアするしないは命令次第である。このビットがセットされ、かつ後述のオーバーフロートラップ許可ビットがセットされている場合トラップ (例外) が発生する。

- **Ｃビット**

  Ｃビットはキャリー条件コードである。一般的にこのコードは算術演算で桁あふれを起こした場合にセットされる。算術演算で桁あふれしなかった場合はクリアされる。算術演算以外の命令ではクリアするしないは命令次第である。このビットは分岐命令のほか、倍精度の演算の際に入力として利用される。

ビット＜７：４＞

ビット 4 から 7 はトラップ許可フラグである。これらのフラグがセットされていると、特殊な環境でトラップを発生させることができる。

- **ＤＶビット**

  ＤＶビットは 10 進オーバーフロートラップ許可ビットである。このビットがセットされていると命令実行で求められた 10 進数の結果の絶対値が大

きすぎて、指定した宛先アドレスに表現できないときに 10 進オーバーフロートラップが発生する。

- **ＦＵビット**
  ＦＵビットは浮動小数点アンダーフロートラップ許可ビットである。このビットがセットされていると、浮動小数点命令を実行して求められた浮動小数点データの値が小さすぎて表現できない場合に浮動小数点アンダーフロートラップが発生する。

- **ＩＶビット**
  ＩＶビットは整数オーバーフロートラップ許可ビットである。このビットがセットされていると命令を実行してオーバーフローが発生した場合において整数オーバーフロートラップが発生する。

- **Ｔビット**
  Ｔビットはトレースビットである。このビットがセットされていると、次の命令を実行した後で、トレーストラップが発生する。この機能はデバッグソフトや性能解析ソフトなどでプログラムを 1 命令づつ実行するために使用される。

ビット＜ 15：8 ＞
ビット 8 からビット 15 までは現在使用されておらず、常にゼロとなっている。

3.3　命令ストリーム形態

3.3.1　可変長命令

VAX11 は固定命令長ではなく可変長命令で命令ストリームが構成されている。可変長命令とはオペコードとオペランドの長さが一定ではない命令のことである。可変長命令を用いることで命令ストリームの無駄を無くし、メモリ空間を有効に使うことができる

3.3.2　オペコード

VAX11 は命令コードとして 1 バイト命令と 2 バイト命令の 2 種類を持つ。2 バイト命令の 9 割以上が拡張プロセッサオプション使用時、つまりオクタワードやG浮動小数点やH浮動小数点用の命令であり、のこりの 1 割にしてもデバッグ用の命令であるため、実際には 1 バイト命令でほとんどのプログラムが動作する。1 バイト命令と 2 バイト命令の区別は命令コードが FC(1111 1100) から FF(1111 1111) までの間の値で始まる場合は 2 バイト命令という判別できるようになっている。図 2 に命令コードの形式を示す。

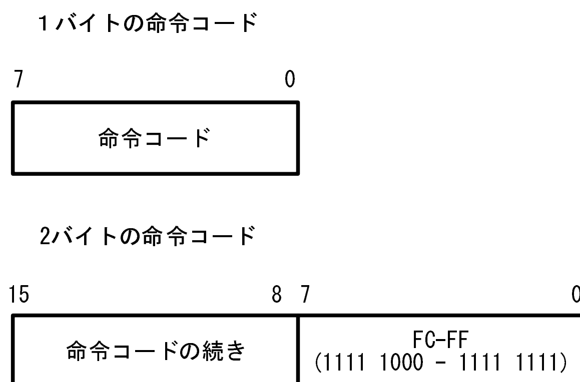

図 2: 命令コードの形式

3.3.3　オペランド

命令ストリーム中のオペランドは”アドレッシングモード識別子＋レジスタ番号”で構成され、その数は命令によって決定される。

3.3.4　アドレッシングモード

VAX11 において命令オペランドはメインメモリ, 汎用レジスタ, 命令ストリームの中の 3 つの場所に存在することができる このため、命令オペランドがどこにあるかを示す方法が必要である。この方法のことをオペランドアドレッシングモードという。アドレッシングモードは命令ストリームの中にある 8 ビットのオペランド対応部の上位 4 ビットを利用して指定される。図 3 にアドレッシングモードのパターンをまとめた図をを示す。この、上位 4 ビットのアドレッシングモードを指定する部分をモード指定子という。

VAX11 のアドレッシングモードの一覧を表 2 に示す。VAX11 のアドレッシングモードは大きく分けて、汎用レジスタアドレッシングモードとプログラムカウンタアドレッシングモードの 2 種類に分類される。汎用アドレッシングモードとは PC 以外の汎用レジスタを利用できるモード。その逆でプログラムカウンタア

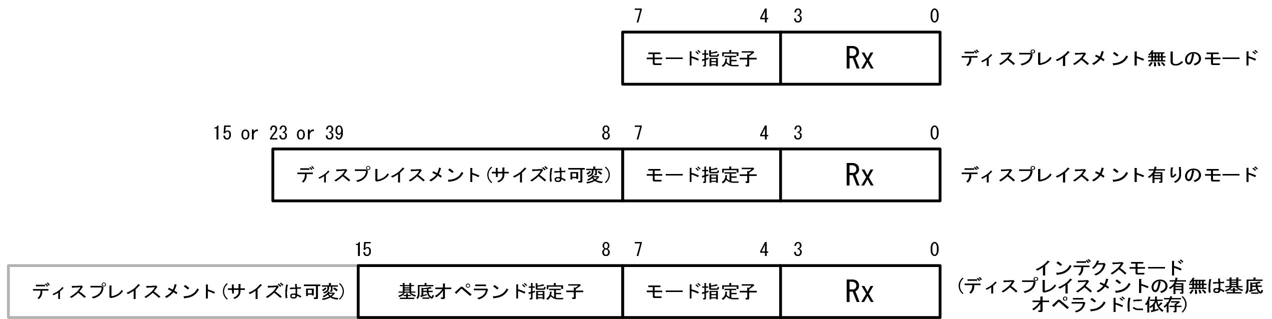

図 3: アドレッシングモードタイプ

ドレッシングモードはPC以外のレジスタは利用することができないモードである。細かい分類については説明を加えながら以下に列挙する。

1. **汎用アドレッシングモード**

   (a) **レジスタモード**
   指定されたレジスタの内容がオペランドとなる。動作例を図4に示す。

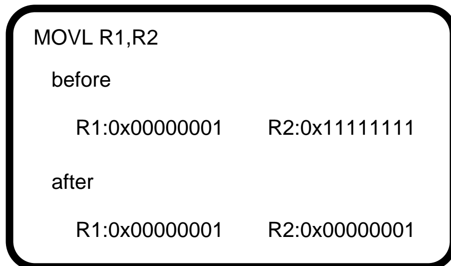

図 4: レジスタモード

   (b) **レジスタディファードモード**
   指定されたレジスタの内容をアドレスするメモリデータがオペランドとなる。図5に動作例を示す。

   (c) **オートインクリメントモード**
   基本となる動作はレジスタディファードと同じである。加えて、このモードではオペランド取得後指定レジスタの値に命令で指定されたオペランドのサイズに応じた値を加算するバイトなら1バイト、ワードなら2バイト、ロングなら4バイトという加算方法となる。図6に動作例を示す。

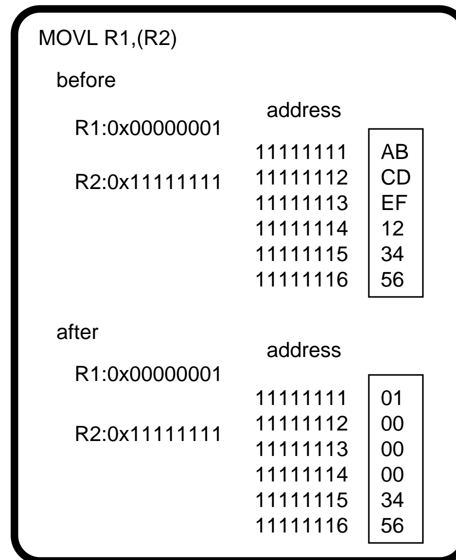

図 5: レジスタディファードモード

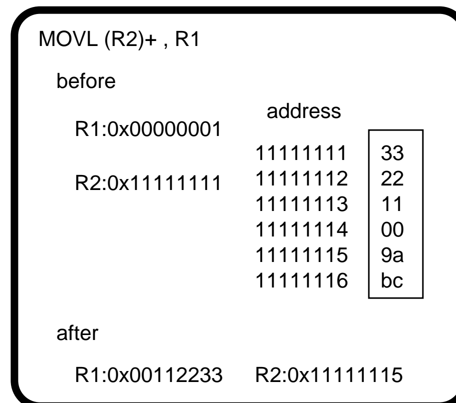

図 6: オートインクリメントモード

汎用アドレッシングモード	
モード指定子	モード名
0x0 - 0x3	Literal
0x4	Index
0x5	Register
0x6	Register Deferred
0x7	Autodecriment
0x8	Autoincrement
0x9	Autoincrement Deferred
0xa	Byte Displacement
0xb	Byte Displacement Deferred
0xc	Word Displacement
0xd	Word Displacement Deferred
0xe	Longword Displacement
0xf	Longword Displacement Deferred
プログラムカウンタアドレッシングモード	
モード指定子	モード名
0x8	Immidiate
0x9	Absolute
0xa	Byte Relative
0xb	Byte Relative Deferred
0xa	Word Relative
0xb	Word Relative Deferred
0xa	Longword Relative
0xb	Longword Relative Deferred

表 2: VAX-11/780 アドレッシングモード一覧

(d) **オートデクリメントモード**
オートインクリメントの逆で最初に命令で指定されたオペランドサイズに応じた値を指定レジスタから減算し、その値をアドレスとしたメモリデータがオペランドとなる。図 7 に動作例を示す。

(e) **オートインクリメントディファードモード**
指定されたレジスタの内容をアドレスとするメモリデータを取得し、さらにその値をアドレスとするメモリデータがオペランドとなる。つまりメモリを 2 回読むことでオペランドが求められる。オペランド取得後はオートインクリメントモードと同じく、指定レジスタの値に加算を行う 図 8 に動作例を示す。

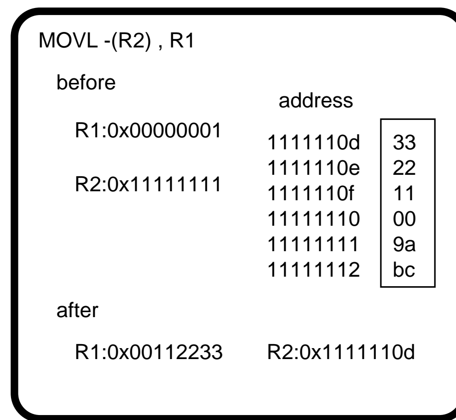

図 7: オートデクリメントモード

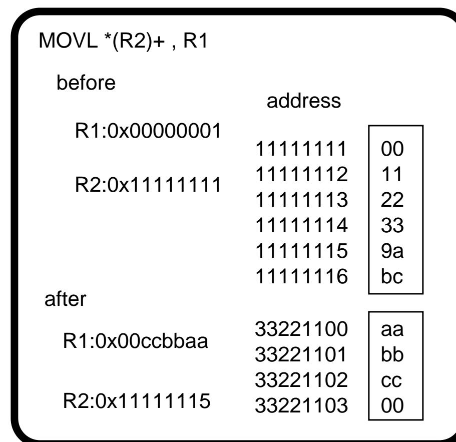

図 8: オートインクリメントディファードモード

(f) **リテラルモード**
このモードはオペランド指定子の変わりに即値を用いる。このモードが使用できる即値の範囲は 0〜63 までである。実際には使用できる即値範囲が 0〜15,16〜31,32〜47,48〜63 の 4 通りに分類できる。図 9 に動作例を示す。

(g) **ディスプレイメントモード**
命令ストリーム中に存在するディスプレイメント (変位) を指定レジスタの内容と加算し、その結果をアドレスとしたメモリデータがオペランドとなるディスプレイスメントのサイ

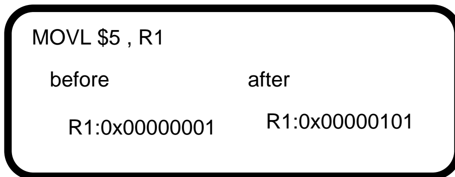

図 9: リテラルモード

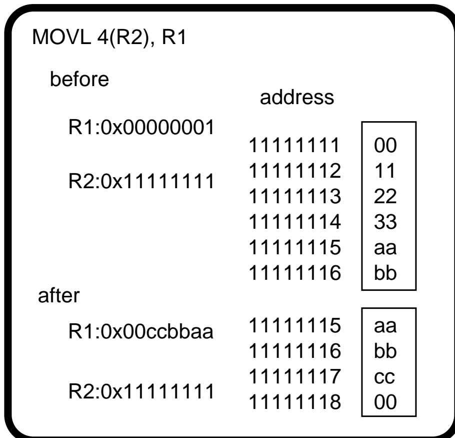

図 10: ディスプレイスメントモード

ズは 1 バイト、2 バイト、4 バイトの 3 種類があり、それぞれに対応するモード指定子が用意されている。
図 10 に動作例を示す。

(h) **ディスプレイスメントディファードモード**
ディスプレイスメントモードにて求めたオペランドをアドレスとしたメモリデータをオペランドとする。このモードもディスプレイスメントのサイズに応じてモード指定子が異なる。図 11 に動作例を示す。

(i) **インデックスモード**
このモードのオペランド指定子は少なくとも 2 バイトから構成される このモードは以下の規則に従って構成される。

i. 第一次オペランド指定子は命令ストリームのビット 0 からビット 7 である。
ii. 第二次オペランド指定子は命令ストリームのビット 8 からビット 15 に表されている。第二次オペラントは基底オペランドとも呼ばれるため、その指定子は基底オペランド指定子といわれる。

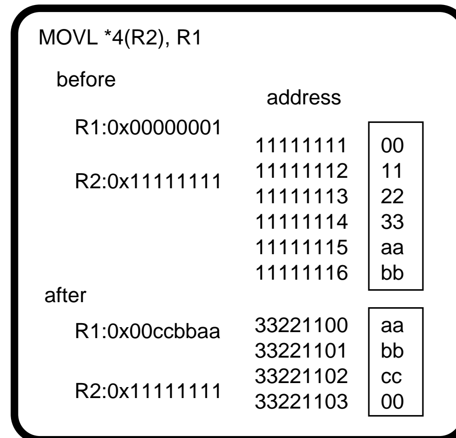

図 11: ディスプレイスメントディファードモード

第一次オペランドは指定レジスタの内容に対して命令で指定されたオペランドサイズに応じた値を乗算した値をアドレスとするメモリデータである バイトの場合には 1、ワードの場合には 2、ロングワードの場合には 4 というように自身のデータ型のバイト数を乗算する。

次に基底オペランド指定子を処理する基底オペランド指定子にはリテラルモード, インデックスモード, レジスタモードを除いた汎用レジスタアドレッシングモードのすべてのアドレッシングモードをもつオペランドが指定できる。また、後述するプログラムカウンタアドレッシングモードもすべて指定可能である。ただし、オートインクリメントモード, オートデクリメントモード, オートインクリメントディファードモードの場合は一次オペランド指定子で示されたレジスタと同じレジスタを基底オペランド指定子が使っている場合, 結果が予測できない。

最後に、基底オペランド指定子によって求められた基底オペランドと第一次オペランドを加算した結果をアドレスとするメモリデータが最終的なオペランドとなる。インデックスモードは配列を非常に効率よくアクセスすることができるため VAX11 の専用コンパイラ

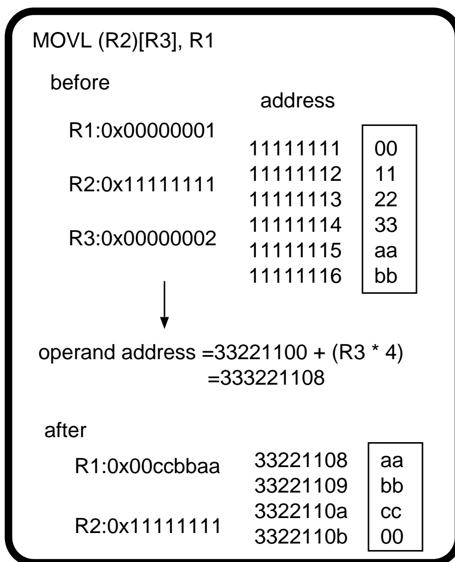

図 12: インデックスモード
レジスタディファードタイプ

では高速化のためによく使用されていたようである。配列アクセスのためにインデックスモードを用いた例を図 12 に示す。図 12 では第一オペランドがレジスタディファードインデックスモードであり、配列のベースアドレスをポインタとして R2 が持ち、求めたい配列の要素アドレスへの変位を R3 によって与えられている。よって、両者を加算することで求めたい配列要素のアドレスが求まる。そのアドレスの先がオペランドデータとなる。

2. **プログラムカウンタアドレッシングモード**

(a) **イミディエイトモード**
イミディエイトモードは PC を汎用レジスタとして使用したオートインクリメントモードである。つまり命令ストリームの中に即値が含まれており、そのサイズは命令によって指定されるその即値がオペランドとなる。オペランド取得後 PC に即値のサイズに応じた変位を加算する。動作例については図 6 の対象レジスタが PC となるだけであるため、省略する。

(b) **アブソリュートモード**
このモードはＰＣを汎用レジスタとして使用したオートインクリメントディファードモードである。命令ストリーム中の即値をアドレスとしてメモリのデータを取得し、それをオペランドとする。即値のサイズは命令によって指定される。 オペランド決定後,PC の値は即値のサイズに応じて定数を加算し変更する。動作例については図 8 の対象レジスタが PC となるだけであるため、省略する。

(c) **リラティブモード**
このモードはＰＣを汎用レジスタとして使用したディスプレイスメントモードである。命令ストリームの中のディスプレイスメント (変位) とＰＣの値を加算して、その値をアドレスとするメモリデータをオペランドとする。このモードはオペランドアドレスがＰＣに対して常に相対的な位置にあるため, リラティブモードという。動作例については図 10 の対象レジスタが PC となるだけであるため、省略する。

(d) **リラティブディファードモード**
このモードはＰＣを汎用レジスタとして使用したディスプレイスメントディファードモードである。命令ストリーム中のディスプレイスメントがＰＣに加算され、その値をアドレスとしたメモリデータをさらにアドレスとしたメモリデータがオペランドとなる。動作例については図 11 の対象レジスタが PC となるだけであるため、省略する。

3.4 メモリ管理機構 (MMU: Memory Manegement Unit)

3.4.1 MMU とは

MMU(Memory Management Unit) はメモリを管理する処理機構である。管理とは主にメモリのマッピングや保護のことを指す。その他にキャッシュの管理なども行う。

3.4.2 メモリマッピング (アドレス変換)

メモリマッピングとは、仮想アドレスを物理アドレスへと変換する機能のことである。

アプリケーションなどソフトウェア側から使用されるアドレス空間は、仮想的なものであり物理的なメモリを直接扱っているわけではない。これは物理的に用意できるメモリ容量が、アプリケーションプログラムなどが要求するメモリ容量に対して不足する事態が起こるという問題を解決するために考え出された仕組みである。プログラムの性質を考えると、ある瞬間瞬間に実行されているのは全体のプログラムの一部でしかない。そこでプログラムをいくつかのブロックに分割し、必要な部分だけを実メモリにロードして実行させ、不要になったらそのブロックをハードディスクなどの補助記憶装置に退避させて、空いたスペースに別の必要なブロックをロードするというしくみをＯＳに持たせることでユーザーは物理メモリの量を意識することなく巨大なプログラムを実行できることになる。これが仮想記憶の仕組みとなる。仮想記憶は後述のメモリ保護に対しても非常に有効に利用することができる。VAX11 の場合、この仮想記憶における 1 ブロックを 1 ページという単位で呼び、1 ページのサイズは 512 バイトと規定されている。この仮想記憶のアドレスのことを仮想アドレスと呼び、物理アドレスとはまったく別のものとなる。しかし実際には物理メモリのどこかと仮想記憶は関連付けられているため、仮想アドレスは比較, 加算などの処理を行うことで物理アドレスに変換することができる。

3.4.3 TLB

先に述べたメモリマッピング処理において、メモリアクセスがあるたびに変換処理を行うことは処理速度および処理効率の観点からあまり望ましくない。そこで実際には一度変換したアドレスを仮想アドレス：実アドレスといった 1 対 1 のテーブルにして保存しておき、そちらのテーブルをチェックして、変換アドレスがない場合のみ変換処理を行う。このテーブルのことを TLB(Translation Look-aside Buffer) という。TLB はメモリアクセス時の性能を決める重要な要素となる。TLB にアドレスが有るか無いかでメモリデータを得るのにかかる時間がかなり変わるため、極力 TLB ミスをさせないようにすることが大切となる。

3.4.4 メモリ保護

典型的なマルチタスクシステムなどでは複数のプログラムがメインメモリに同時に存在することがある。マルチタスクとは複数のタスク (プログラム) を同時に物理メモリ上に置き、ある決められた順番に従って少しずつ (多くの場合は時分割的に) 実行しているものである。この際に、あるプロセスが他のプロセスやＯＳに影響を与えないようにするにはメモリの保護が必要となる。つまり、あるプログラムが動作すると、ＯＳや他のプログラムが使用しているメモリデータを破壊してしまうというようなことを起こさせないということである。これを実現するためにページごとにどのプロセスが使用しているかのタグをつける。タグにはプロセスのモードと保護コードが記述されている。プロセスは大きく分けて 4 種類のモードに分けて管理される。カーネルモード, エグゼクティブモード, スーパーバイザモード, ユーザーモードの 4 つで、先にあげたほうから順に高い特権をもつ。高い特権をもつものは低い特権で管理されたプロセスのページに対してアクセスが可能である。どのようなアクセスが可能であるかについては保護コードに記されている。アクセスの種類はリード/ライト, リードオンリーアクセス不可の 3 つである。このタグに従って低い特権のプロセスが高い特権もしくは同レベルの特権をもつプロセスのメモリデータを破損しないように管理することをメモリ保護という。

3.5 命令セット

VAX11 の命令セット数を表 3.5 に一覧として示す。表 3.5 からわかるとおり、VAX11 は非常に豊富な命令を持つ。加算 (例:"ADD")、データ移動 (例:"MOVE") などどのようなプロセッサでも大抵持っている命令に加えてアセンブラを書く際に書き手が楽をできるように用意された命令が数多く存在する。1 例をあげると、"CRC" 命令などである。これは CRC(巡回冗長検査) を 1 命令で行うという命令である。

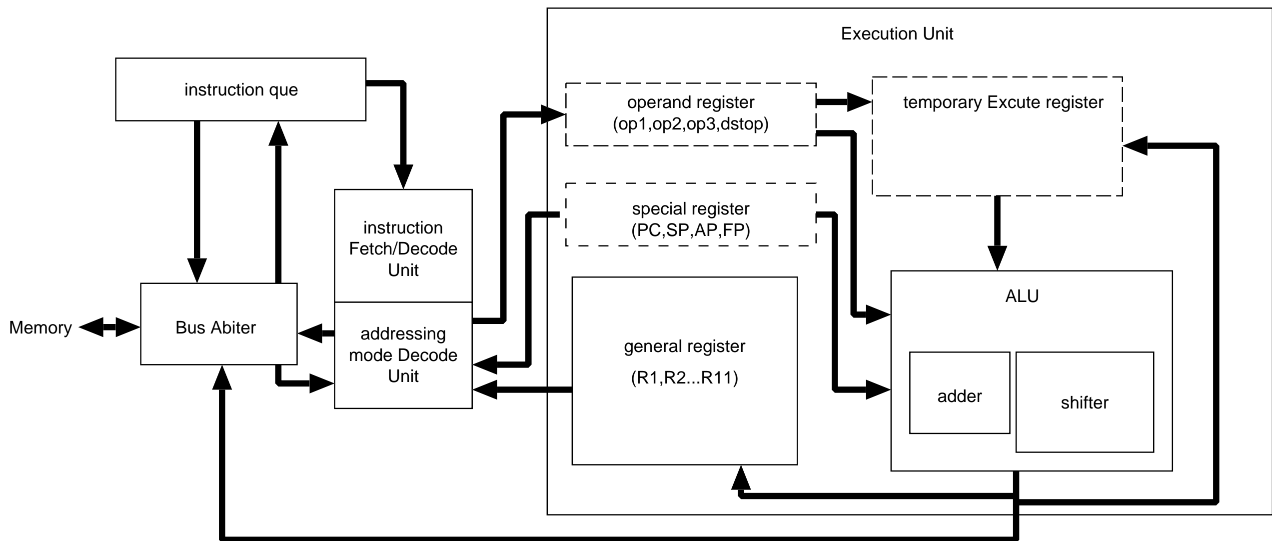

図 14: データパス

整数, 浮動小数点命令	133 命令
特殊命令	27 命令
制御命令	44 命令
文字列命令と巡回冗長検査命令	12 命令
10 進数字列命令	16 命令
編集命令	1 命令
トータル	233 命令

表 3: VAX-11/780 命令数一覧
＊注：拡張データタイプ用の命令は省く

4 開発互換プロセッサのシステムアーキテクチャ

4.1 設計思想

今回設計した互換プロセッサは動作周波数と回路規模のバランスを重視して開発を行った。その上で比重としては動作周波数に多少の重きを置いた。これは、アドレッシングモードデコードの関係でどうしても 1 命令に必要なクロック数が増加する傾向にあるため、性能をあげるためには動作周波数を上げる必要があると考えたためである。FPGA の場合、動作周波数を向上させせようとして共通化できる処理を共通化せずに独自のハードウェアで構成していくと、最終的には回路規模の増大に伴う配線遅延の増加で結果として動作周波数が低下してしまう。この為、今回の開発においては共通化できる部分は共通化することで回路規模の縮小を図り、その一方でレジスタは惜しまず使うことでファンイン、ファンアウトを減らした。以上のような工夫をすることで動作周波数の向上を図った。

4.1.1 システム構成

図 13 に互換プロセッサのシステム構成図を示す。互換プロセッサは大きく分けて制御系とデータパス系に分かれており、MMU, 命令キュー, メインメモリと制御系はコアモジュールを介して接続されている。データパス系はレジスタ群とＡＬＵを持ち、制御系によって制御される。データパスと制御系の切り分けをすることでデータフローを集中させ、配線遅延及びセレクタの増加を抑えることで、回路規模の増大化を防ぐとともに動作速度の向上を図っている。

4.1.2 データパス

設計したプロセッサのデータパスを図 14 に示す。基本方針としてレジスタのファンイン、ファンアウトを減らすような設計をした。これは、”システム構成”の項目で述べたと同じく、回路規模の増大化を防ぐとと

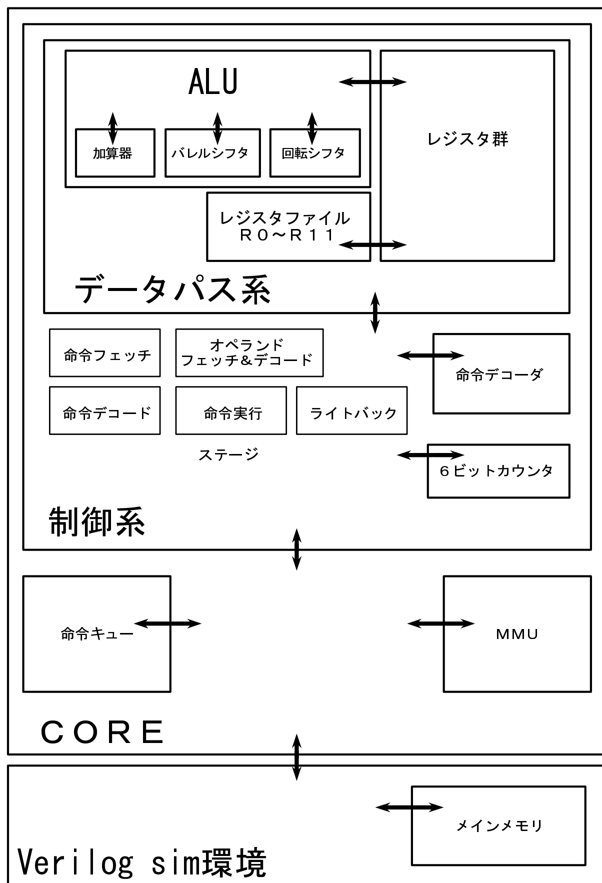

図 13: VAX11/780 互換プロセッサ システム構成図

もに動作速度の向上を狙いとしている。

4.2 互換プロセッサファイル構成

互換プロセッサを構成するＳＦＬファイルの説明を以下に示す。
本互換プロセッサは 12 ファイルで構成されており、トータルの記述量は 4000 行弱となっている。図 15 にファイルの階層的な関連と各ファイルの行数を示すとともに、以下に各ファイル内容についての概要を示す。ファイル名は内部に記述されているモジュール名と 1 対 1 で対応している。

- **core.sfl**
  トップモジュールとして、各モジュール繋ぐ役割をする。

- mmu.sfl

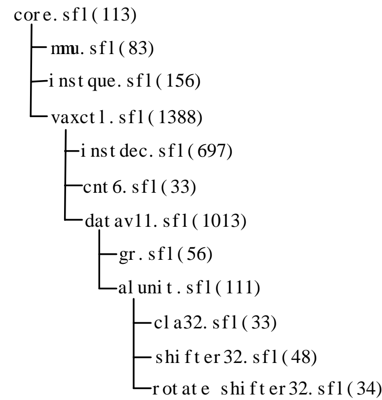

図 15: SFL ファイル構成 (カッコ内はファイルの行数)

  MMU(Memory Management Unit) となるモジュールであり、メモリアクセス管理を行う。

- instque.sfl
  命令キュー (Instruction Queue) として動作するモジュールとなる。

- vaxctl.sfl
  制御モジュールとなる。互換プロセッサの命令, 割り込みなどの大半の処理はこのモジュールを中心として行われる。このモジュールは処理フローに関係する 5 つのステージをもつ。各ステージはそれぞれ命令フェッチ、命令デコード、オペランドフェッチ＆デコード、命令実行、ライトバックの処理を行う 各ステージの細かい内容については後述する処理ごとの説明において行う。

- instdec.sfl
  命令デコードのためのテーブル引きをするためのモジュールとなる。

- cnt6.sfl
  乗算命令のループ回数をカウントするための専用 6 ビットカウンタモジュールである。

- datav11.sfl
  データパスモジュールである。処理に使うレジスタを含めデータ保存はほぼこのモジュール内で完結する。

- gr.sfl
  汎用レジスタモジュールである。R0 から R11 まで

のレジスタが用意されている。FPGA にフィッティングする際に EAB を使って汎用レジスタをフィッティングすることを考慮し、モジュール"datav11"とはあえて別のモジュールとした。

- alunit.sfl
  ALU(Arithmetic Logic Unit) となるモジュールである。各下位の演算モジュールから出力された条件ビットを判別して出力する処理などを行う。詳しくは後述する。

- cla32.sfl
  32 ビット全加算器となるモジュールである。

- shifter32.sfl
  両方向対応 32 ビットバレルシフタとなるモジュールである。

- rotateshifter32.sfl
  両方向対応 32 ビットローテートバレルシフタとなるモジュールである。

4.3 命令キュー (Instruction Queue)

4.3.1 命令キューとは

命令キューとは、命令ストリームを格納する先入れ先出し (FIFO:First In,First Out) のデータバッファのことである。本互換プロセッサでは、データバッファそのものではなく、

- メインメモリからの命令ストリーム取得
- 取得命令データの管理 (データのクリア, 必要に応じた再取得など)
- サイズ可変データ要求に対する命令データの提供

という大別して 3 つの機能を満たすモジュールを命令キューと呼ぶ。

4.3.2 実装理由

アーキテクチャの項で述べたとおり、VAX11 は可変長命令をサポートする。そのため、1 命令の命令長はオペコード及びオペランドをすべてデコードするまで決定できない。加えてアドレッシングモードの中には、アドレッシングモードデコードに必要な即値が命令ストリーム中に存在するモードが存在する。このような条件の中、必要に応じてその都度ごとに命令ストリームをメインメモリから取得していたのでは、メモリレイテンシに依存するメモリの応答待ちが頻繁に発生することになり、1 命令あたりに必要なクロック数 (CPI: Clock Per Instrunction) が増加することとなる。つまり、結果として命令処理速度の低下を招く。

上記の問題を解決するためには、メモリとプロセッサとの間の速度差を埋めるように命令キューを用意してやることが考えられる。まず、あらかじめメインメモリから命令ストリームを一定量読み出して命令キューに保存しておく。命令キューはキュー内にデータが存在する場合、データ要求を受けたら即座にそのデータを出力することができる。よって、命令, オペランドフェッチなどを行う際に、命令キューからオペコードやアドレッシングモードデコードに必要な即値を取得すれば、メインメモリとプロセッサの間の速度差の緩衝をはかり処理速度の低下を避けることが可能となる。

以上のようなことから、命令キューを実装することとした。

4.3.3 キューのサイズ

命令キューのキューサイズをどのぐらいの大きさにするかについて以下の項目を考慮した

- **(a).** 汎用的に使用する命令のオペコード長はほとんどが 8 ビットであり、命令長が 16 ビットの命令は拡張オプション装着時の特殊命令ばかりであること
- **(b).** アドレッシングモードデコードに使用される即値およびディスプレイスメント (変位) は最大でも 64 ビットであり、大半が 32 ビット以下であること
- **(c).** 分岐命令などによってプログラムカウンタの値が更新された場合、キューは自身のデータをフラッシュ(消去) する必要があること
- **(d).**CPI は特殊命令を除き、最低でも 9 クロック程度であるため、1 エントリ空になっても他の動作中にメモリからデータを取得できると予測したこと

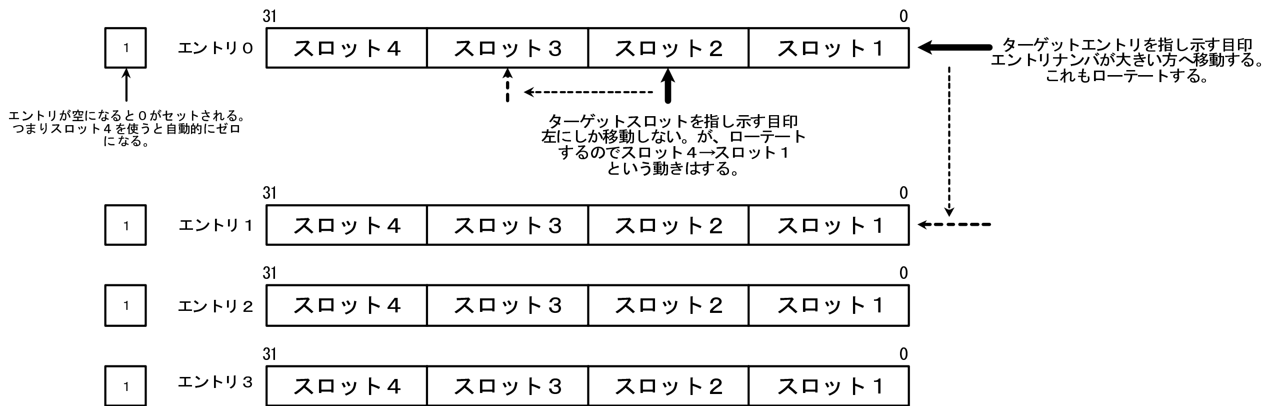

図 16: 命令キューの動作概要図

- **(e).** 命令実行中に命令キュー以外がメインメモリにアクセスすることがあること

(a),(b) については 1 命令の命令長が平均でどの程度になるか考える際の材料とした。(c),(d),(e) については取得の際のメモリ占有と分岐直後のメモリ占有について考慮する際の材料とした。これは、キューがメモリの応答待ちに入ってい場合、MMU によって他のメインメモリへの要求は受け付けない仕様としたため、頻繁に命令キューがメモリアクセスを繰り返すと、命令実行部がメインメモリへの要求受付待ち状態となり命令キューを実装した意味が薄れてしまう。したがって命令キューは自身のキューが埋まるまでメモリアクセスを繰り返すためキューサイズを大きくすればするほど、メモリアクセス増えることになる。また、分岐命令などで分岐が確定した場合には、キューをフラッシュして分岐先の命令ストリームを新たに取得しなければならないため、キューサイズを大きくして大量に命令ストリームを保存しておいても、結果的に保存データを破棄してしまい、徒労に終わることも考えられる。

以上の観点から総合的に考え、命令キューは 32 ビット 4 エントリのトータル 128 ビットとした。

4.3.4 動作仕様

命令キューの動作を模式的表した図を図 16 に示す。

動作開始時、命令キューは当然ながら空であるため、メモリに大してデータ要求をだす。この際、命令キュー独自のプログラムカウンタ (以降キューカウンタとする) をアドレスとして用いる。これは、命令実行などの際に使用されるプログラムカウンタを利用してしまうと、命令データの先読みを行うのが困難になるためである。命令キューはメモリからデータを 32 ビット単位で取得するため、一回のデータ取得処理で命令キュー 1 エントリ (32 ビット) 分が埋まることになる。データ取得後、キューカウンタはデータ取得が完了すると＋4 される。

キューの各エントリに有効な命令データが存在するかを判断するため、1 エントリごとにデータの有無を判断するためのフラグビットを用意し、データが存在する場合にフラグビットをオンにする。その後、データが使用されるなどで 1 エントリすべて空になった場合にフラグをオフにする。命令キューは、フラグオフとなったエントリが存在するかぎりメインメモリにデータ要求を出し続け、命令データを取得し続ける。メインメモリより取得したデータをどのエントリに格納するかについては、格納エントリを示すフラグを用意しており該当エントリにデータが格納された後、次のエントリを指すようになっている。

命令キューは要求として 8 ビット、16 ビット、32 ビットの出力要求を受け付ける仕様とした。これは VAX11 のアーキテクチャを調査した結果、命令ストリームから取得するデータのサイズが、先の 3 サイズの組み合わせてすべて実現できるためである。また、単独で 32 ビット以上の出力要求についてはその出現頻度を考慮した結果必要ないと判断した。8 ビットおよび 16 ビットの出力要求を処理するため、1 エントリを 4 ブロック

にセパレートし、出力要求時にどのブロックからデータを取り出せば良いか判別するフラグを用意した。

出力要求を受けた命令キューは要求データサイズを満たすだけのデータを自身が保持しているかどうかチェックし、要求データサイズを満たすデータを保持している場合、ACK を返し命令データを出力する。

分岐命令などによりプログラムカウンタが更新された場合、制御部から命令キューに対してキューデータのフラッシュ要求が来る。フラッシュ要求が来た場合、直ちに命令キュー内のキューデータはフラッシュされる。同時にキューカウンタとプログラムカウンタの同期が行われ、命令ストリームの再取得が開始される。フラッシュ要求が来た際にメインメモリからデータを取得中であった場合は、取得したデータを破棄したのち、上記の処理が行われる。

命令キューはプロセッサが HALT 命令を実行する、もしくはなんらかのエラーで停止するまで動作し続ける。

4.4 メモリ管理機構 (MMU: Memory Manegement Unit)

4.4.1 動作仕様

現在は

- メモリに大してデータ出力要求を出し、データが取得でき次第 ACK を返す処理

- 命令キューと命令動作制御系のメモリデータ要求競合時の処理

の２通りの処理のみをサポートしている。アドレス変換機能、保護機能などは実装予定であるが未実装である。

4.5 処理フロー

互換プロセッサの処理フローを説明する。まず、動作のフローチャートを図 17 に示す。

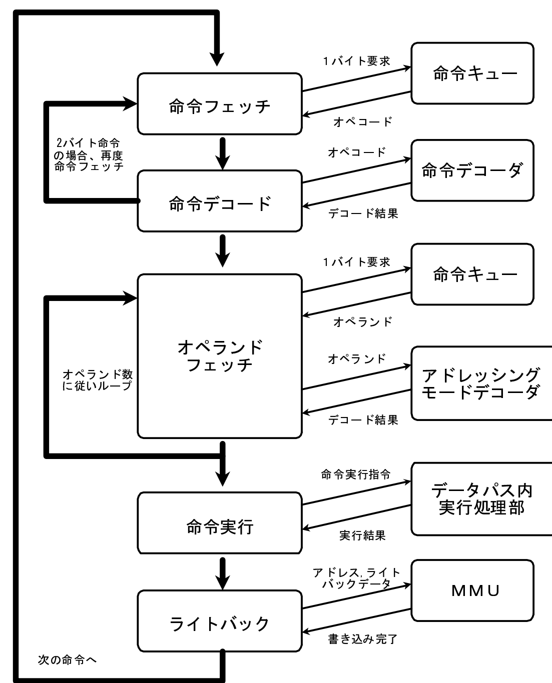

図 17: 処理フローチャート

4.5.1 (1). 初期化 (Initialize) 動作

プロセッサが起動すると、まず初期化 (Initialize) 動作に入る。初期化プロセスとして現在は、

- スタックポインタの値を設定

- レジスタの値を初期化

- 命令キュー及び命令フェッチステージの起動

を行っている。本来であれば、スタートアップルーチンを読み込んで行うものであるが、現在までにまだスタートアップルーチンの読み込み処理部が完成していないため、内部で自動的に処理を行う使用となっている。

4.5.2 (2). 命令フェッチ

初期化が完了すると命令フェッチステージが動きだす。命令フェッチステージは命令キューからオペコードを取得した後、命令デコートステージへ遷移する。

4.5.3　(3). 命令デコード

命令デコードは命令デコードステージで行う。(2) で取得したオペコードをデコードを行い、命令の種類, オペランド数, オペランドタイプ (オペランドサイズ, 属性) に対応するフラグを生成する。フラグ生成後、オペランドフェッチステージへ遷移する。一部の特殊命令 (”HALT などオペランドの無いもの”) についてはオペランドフェッチステージではなく、特殊実行ステージへ遷移する。

4.5.4　(4). オペランドフェッチ

オペランドフェッチはオペランドフェッチステージで行う。(3) でデコードして生成したフラグに基づいて、オペランドを取得する。まずアドレッシングモード指定子を含む仮オペランドを命令キューから取得する。次にアドレッシングモードデコードを行い、そのデコード結果に基づいて最終的なオペランドをメモリ, レジスタ, 命令ストリームのいずれかから取得する。上記の動作をオペランド数に基づき複数回行った後、命令実行ステージへと遷移する。例外的にある種のオペランドについてはアドレッシングモードを持たないものも存在する。このため、命令キューからの取得した仮オペランドがそのままオペランドとなるものもある。また、状態遷移する前に PC(プログラムカウンタ) の値を更新する。この理由などについては後述する。

4.5.5　(5). 命令実行

命令実行は命令実行ステージで行う。(3) での命令種類に基づき、(4) で求めたオペランドを用いて命令の実行を行う。命令実行に要するクロック数は命令によって異なるが、頻出する命令 (”MOVE,ADD,SUB,CLR” 系統など) に必要なクロック数はクルティカルパスなどを考慮しつつなるべく少なくなるように設計した。命令実行後、ライトバックに適したデータへとデータの整形を行った後、ライトバックステージへと遷移する。命令実行後にデータを書き戻す必要がない命令もあるが、現在はライトバックステージに遷移させている。

4.5.6　(6). ライトバック

ライトバックはライトバックステージで行う。(5) での結果に基づいてメモリもしくはレジスタに対してデータの書き戻し (ライトバック) を行う。データ書き出しの必要が無い命令については、現在は素通りという形をとっている。ライトバックが完了すると、現在の命令処理の後始末を行ったあと命令フェッチステージへと遷移する。

4.6　命令フェッチ動作

4.6.1　命令フェッチとは

命令フェッチとはオペコードをメインメモリなどからプロセッサ内に取得することである。本互換プロセッサでは命令キューからオペコードを取得することを指す。

4.6.2　動作仕様

命令フェッチステージが起動されると、命令キューに対して 1 バイトのデータ要求を発行する。命令キューの ACK を待ち、ACK が帰ってくるとデータパス内に命令キューから出力されたオペコードを保存し、命令デコードステージへと遷移する。

命令フェッチステージでは 1 バイトしかフェッチをしない。これは、2 バイト命令の判別は取得した 1 バイトを判別する必要があることから、命令デコードステージで行うほうがシンプルだと考えたからである。また、汎用的に利用される命令の大半は 1 バイト命令という観点からも考え、命令デコードステージで判別させることとした。

4.7　命令デコード動作

4.7.1　命令デコードとは

命令デコードとは命令フェッチをして取得したオペコードのデコードを行い、命令の判別を行うことである。本互換プロセッサではオペコードをデコードし、判別結果に基づいて次の処理の準備をする一連の動作を命令デコードと呼ぶ。

4.7.2 動作仕様

命令デコードステージが起動すると命令デコーダに対してオペコードを渡す。命令デコーダはオペコードをデコードし、判別した命令に従い、複数の制御信号を返す。制御信号の種類としては以下の 3 種類を用意した。

- **命令種類**
  命令の種類のフラグ (例:MOV,ADD など) となる。扱うオペランドサイズのみ異なり、命令の基本動作自体は同じ命令種類 (例：”MOVB”と”MOVW”) については命令種類フラグとしては”MOVE”というフラグで統一処理できるように命令実行ステージを設計した。また、2 バイト命令についても 2 バイト命令フラグを設定する。

- **オペランド数**
  オペランドの数を示すフラグとなる。0 個から 4 個までの 5 つ。オペランド数が 4 個を超える命令も存在するが、その類の命令は種類が少なく発生頻度も稀なため、独自のルートで処理を行わせることにした。

- **オペランドタイプ (サイズ)**
  オペランドのサイズを示すフラグとなる。バイト, ワード, ロングワード, クォドワード, フロート, ダブルフロートなどのサイズを示す。また、命令によってはオペランドごとにサイズが異なる命令などもあるため、最高 4 つまでオペランドサイズに応じたフラグを用意できるように設計した。ただし、オペランドがいくつあろうとそのサイズがすべて同じような命令の場合は最初のオペランドサイズのフラグだけを設定すればよい。つまり、n 番目のオペランドのサイズが n-1 番目のオペランドサイズと異なる場合にのみ n 番目のオペランドタイプフラグを設定すればよい。オペランドフラグが設定されていない場合、一つ前のオペランドサイズとみなすようにオペランドフェッチステージの処理を設計した。

- **オペランドタイプ (属性)**
  オペランドの属性を示すフラグとなる。VAX11 アーキテクチャでは各オペランドに命令に応じた属性が存在する。属性は”読み込みのみ (Read Only)”,”書き込みのみ (Write Only)”,”変更 (Read and Write)” ”アドレス (メインメモリ実効アドレス)”,”分岐変位”,”特殊アドレス”の 6 つの属性の組み合わせて表される。

命令デコードステージは命令デコーダが発行した制御信号によって命令実行およびオペランド取得の準備を行う。

まず、命令種類フラグから命令実行ステージを各命令のタスクで起動する。これは命令実行の際に再度オペコードをデコードするのを回避するためである。命令実行ステージが起動された際の詳細な処理内容は処理命令実行ステージの動作仕様にて解説する。同じように、オペランド数フラグに従ったタスクでオペランド数フラグステージを起動する。オペランドタイプ (サイズ, 属性) についても、オペランドタイプフラグに従いオペランドタイプフラグステージを起動する。オペランド数フラグステージ及びオペランドタイプフラグステージはタスクをフラグとして使用するためだけに存在する。フラグレジスタを用意した場合、自ら ON,OFF のための値を代入したり ON,OFF のための関数のようなものを用意する必要がある。しかしステージ記述のタスクをフラグレジスタとみなせば、generate,finish の二つの予約語を用いることでフラグレジスタを変更したことになり非常に都合が良い。このため、フラグステージを用意した。

すべての準備が終了すると 2 バイト命令の 1 バイト目である場合を除き、オペランドフェッチステージへと遷移する。2 バイト命令の 1 バイト目であった場合は、命令フェッチステージへ遷移する。

4.8 オペランドフェッチ動作

4.8.1 オペランドフェッチとは

オペランドフェッチとはレジスタやメインメモリ上からオペランドを取得することである。本互換プロセッサでは命令キューから仮オペランドを取得し、アドレッシングモードデコードなどを行いオペランドを取得することを指す。

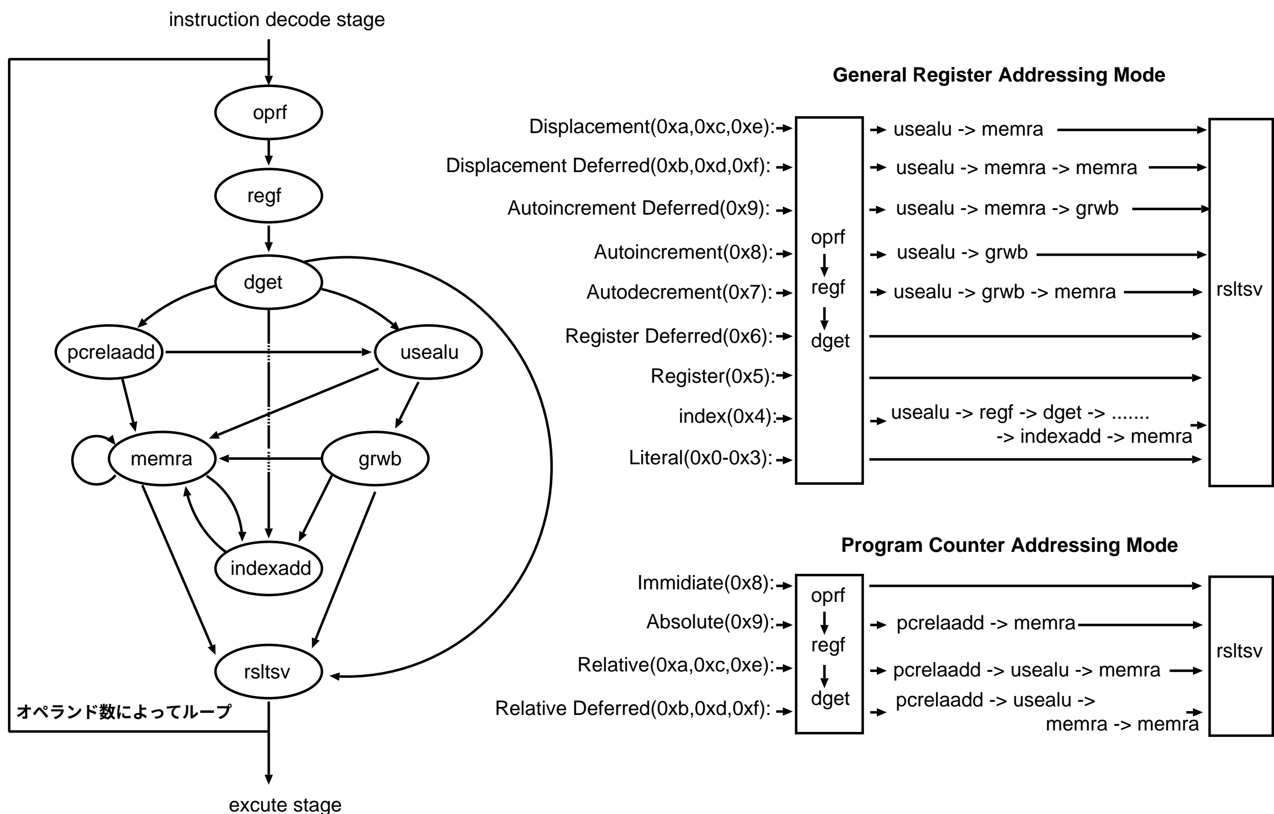

図 18: オペランドフェッチの状態遷移図及び各アドレッシングモードの状態遷移一覧

4.8.2 アドレッシングモードデコードとは

アドレッシングモードデコードとは、仮オペランドのアドレッシングモードを判別してオペランドを求めることを指す。

4.8.3 オペランドフェッチ動作について

オペランドフェッチは命令キューから仮オペランドを取得し、必要があればアドレッシングモードデコードをしてオペランドを求めることで完了する。一連の動作をオペランド数に基づいて繰り返すことで、全てのオペランドが取得できる。

本互換プロセッサではオペランドステージ内の 9 つのステートを遷移させながらオペランドフェッチを行う仕様とした。具体的には oprf，regf，getd，rsltsv，usealu，memra，grwb，idxadd，pcrelaadd の 9 つのステートである。9 つのステートをその処理目的より分けると

- **仮オペランドの取得**
  oprf
- **アドレッシングモードデコード**
  regf，getd，rsltsv，usealu，memra，grwb，idxadd，pcrelaadd

の 2 つに分かれる。図 18 左部分にステート遷移図を示す。

4.8.4 仮オペランドの取得

仮オペランドの取得は oprf ステートで行う。オペランドフェッチステージが起動されると、必ず oprf ステートから動作が始まる。oprf ステートは命令キューに対して 1 バイトのデータ要求を発行する。命令キューの ACK を待ち、ACK が帰ってくるとデータパス内に命令キューから出力された仮オペランドを保存する。

仮オペランド保存後、オペランド属性フラグをチェックする。オペランド属性が分岐変位かつ、オペランド

サイズがバイトだった場合は取得した仮オペランドがオペランドとなる。そのため、アドレッシングモードデコードを行わないように専用フラグをセットした後 regf ステートへ遷移する。オペランド属性が分岐変位でオペランドサイズがワードである場合は、仮オペランドを保存した上で専用フラグをセットし次のクロックで再度命令キューに対してデータ要求を発行し、仮オペランドを取得する。その後、2 回の取得データを結合した上で保存し、regf ステートへ遷移する。オペランド属性が分岐変位で無い場合は仮オペランド保存後、そのまま regf ステートへと遷移する。

4.8.5 アドレッシングモードデコードの処理フロー

アドレッシングモードデコードは regf，getd，rsltsv，usealu，memra，grwb，idxadd，pcrelaadd の 8 つのステートを遷移させることで行う。これらのステートの組み合わせによってすべてのアドレッシングモードに対応すると同時に、共通化できる処理を共通化した。これにより回路規模の縮小とそれによる動作周波数の向上を図った。図 18 の右部分に各アドレッシングモードの状態遷移フローを示す。

4.8.6 アドレッシングモードデコードにおける各ステートの概要説明

先述のアドレッシングモードデコード時に使用する 8 つのステートがそれぞれどのような処理をするかについて解説する

- **regf ステート**
  このステートは、oprf ステートで取得した仮オペランドのレジスタ指定子に従って、指定レジスタのデータを読みだし保存する。ただし、オペランド属性が分岐変位の場合は、仮オペランドを rsltsv ステートで利用するレジスタに移動させるだけである。データの保存が完了すると dget ステートへ遷移する。

- **dget ステート**
  このステートは、仮オペランドのアドレッシングモード指定子からアドレッシングモードを判別する。判別した結果に従って必要なデータ取得処理を行ったり、次のステートで使用する値を設定を行う。加えて、このステートでライトバックステージを起動する。これは、ライトバックの有無、ライトバック先 (メインメモリ or レジスタ) などは多くの場合ディスティネーションオペランドのアドレッシングモードと命令によって決まるためである。よって、このステートで判別したアドレッシングモードに従ったタスクでライトバックステージを起動することでライトバックの準備をする仕様とした。現在の仕様では、オペランドが複数存在する命令のデコード時にソースオペランドなど明らかにライトバック先ではないオペランドのアドレッシングモードに応じてライトバックステージが起動してしまう。しかしながら、VAX11 アーキテクチャではすべての命令において最終オペランドが必ずディスティネーションオペランドとなっている。このため、現在の仕様でも最終的には正しいタスクでライトバックステージが起動される仕組みとなっている。処理が終わると、それぞれのアドレッシングモードもしくはオペランド属性に合わせ各ステートへ遷移する。

- **rsltsv ステート**
  。大別して、このステートの処理は以下の 3 種類である。
  - オペランドの保存
  - オペランドアドレス (メモリアドレス, レジスタアドレスなど)
  - プログラムカウンタの更新

  オペランドの保存
  このステートに遷移した場合、基本的にオペランドフェッチが完了したことを意味する。オペランドの保存先レジスタとして op1，op2，op3，dstop という 4 本の 32 ニットレジスタを用意した。
  - 1 オペランドの命令は dstop が使用される。
  - 2 オペランドの命令の際には op3，dstop の 2 本が使用されることになる。
  - 3 オペランド命令の場合は op2，op3，dstop が使用される。

– 4 オペランド命令の場合には op1，op2，op3，dstop が使用される。
– 5 オペランド以上の命令は特殊なルートをとおり、汎用オペランドフェッチステートを通らないため、保存について考慮する必要は無い。

保存規則は上記のとおりである。保存先レジスタの決定に際しては、命令デコードステージにてセットしたオペランド数フラグを用いて判断をする。オペランド数フラグはオペランド保存後に変更される仕様とした。つまり、オペランド数フラグはこのステートを通る際に一つ少ないフラグへと変更されることになる。例えば、オペランド数 2 のフラグであった場合、オペランド数 1 のフラグへと変更される。そして、オペランド数 1 のフラグでこのステートに来た場合は、基本的にすべてのオペランドが求まったことを意味するため、命令実行ステージへと遷移することになる。

オペランドアドレスの保存

オペランドのアドレスについてもこのステートで保存を行う。fmemaop1，fmemaop2，fmemaop3，fmemadstop という 4 本の 32 ビットレジスタを用意した。上記のレジスタにはオペランドフェッチ時に使用した最後のメモリアクセス時のメモリアドレスを保存する。これは、ライトバックや一部の命令でオペランドのアドレスが必要となるためである。メモリアクセスを行わないアドレッシングモードであった場合は前回のメモリアドレスが保存されることになるが、そういった場合は確実にそのオペランドのオペランドアドレスを利用しない命令であるため、問題は出ない。オペランドアドレスを保存する理由は、アドレスを利用する命令があるからである。命令ごとに個別に保存するよりは常に保存をしておいて利用する側が勝手に利用したほうがトータルのセレクタは減少する。このためこのような仕様とした。同じ理由でレジスタ指定子で指定されたレジスタアドレスもここで wbreg というレジスタに保存する。

プログラムカウンタの更新

本互換プロセッサは命令ストリームから命令データを読み出す際に命令キューを利用するため、命令データ読み出しにはプログラムカウンタを利用する必要が無い。このため常にプログラムカウンタを更新してはいない。しかしながら、プログラムカウンタアドレッシングモードのデコード時や命令実行においてプログラムカウンタを利用することがあるため、命令実行の前やプログラムカウンタアドレッシングモードデコードの前にはプログラムカウンタを更新する必要がある。そこで、一部の特殊命令を除き命令実行の前には必ず通るこのステートにおいてプログラムカウンタの更新を行うこととした。

プログラムカウンタ更新をするため、命令キューに対してデータ要求を出すとその要求量に応じてプログラムカウンタの更新変位をカウントする機能を付加した。その更新変位をこのステートで加算することでプログラムカウンタを更新している。

- **usealu ステート**
  このステートは alu を利用してアドレッシングモードデコードに必要な加算、減算、シフトなどの演算を行う。処理終了後は memra ステートか grwb ステートへ遷移する。

- **memra ステート**
  このステートはメインメモリからデータを取得する。処理終了後は、もう一度 memra ステートの処理をするか idxadd ステート，rsltsv ステートのどちらかへ遷移する。

- **grwb ステート**
  このステートはアドレッシングモードに応じてレジスタの値を書き換える処理を行う。あくまで書き換えるだけであり、加減算などの処理は先に述べた usealu ステートで行う。オートインクリメント、オートデクリメント命令で使用する。処理終了後は idxadd ステートか rsltsv ステートへ遷移する。

- **idxadd ステート**
  このステートはインデックスモードにおける基底

オペランドと第一次オペランドの加算処理を行う。処理終了後は memra ステートへ遷移する。

- **pcrelaadd ステート**
このステートはプログラムカウンタを更新する。本互換プロセッサは先述のとおり命令の取得に命令キューを使用するため、プログラムカウンタの値を常に更新する必要が無い。このため、プログラムカウンタの変位を保存しておき、rsltsv ステートにてプログラムカウンタを更新している。ところが oprf ステートや dget ステートでも命令キューから命令データを取得するため、プログラムカウンタアドレッシングモードをデコードする際にプログラムカウンタの値が更新されず、ハザードが発生する。そこで、プログラムカウンタアドレッシングモードのデコード時には、プログラムカウンタの値を使用する前にこのステートを通すことでプログラムカウンタを更新することで、問題を回避している。

4.8.7 各アドレッシングモードごとのデコード説明

ここでは各アドレッシングモードがどのようなルートでステート遷移をしてオペランドを決定するかを簡単に説明する。各モード名のあとの数値はモード指定子である。

汎用レジスタアドレッシングモード

- **汎用レジスタモード** [0x5]

  処理フロー： **dget → rsltsv**

  このモードは指定レジスタの内容がオペランドとなる。regf ステートですでに指定レジスタの内容は読み出しているため dget ステートでは rsltsv ステート用のレジスタにデータを移動させ、rsltsv ステートへ遷移する。

- **レジスタディファードモード** [0x6]

  処理フロー: **dget → rsltsv**

  汎用レジスタモードと同じように見えるが dget ステートでは regf ステートにて取得したレジスタデータをアドレスとしてメモリからデータを取得する。取得したデータがオペランドとなるので、データを保存し rsltsv ステートへ遷移する。

- **オートインクリメントモード** [0x8]

  処理フロー: **dget → usealu → grwb → rsltsv**

  オペランドが求まった後、指定レジスタの値をオペランドサイズに従ってポストインクリメントする。オペランド自体は dget ステートでメモリリードを行うことで取得が完了する。この際にオペランドサイズを判断して、次の usealu ステートで指定レジスタに加算する値の設定をする。メモリからのオペランド取得が完了したら usealu ステートに遷移して加算処理を行う。加算を行った結果を次の grwb ステートにおいて指定レジスタに書き戻し、rsltsv ステートに遷移してデコード完了となる。

- **オートインクリメントディファードモード** [0x9]

  処理フロー: **dget → usealu → grwb → memra → rsltsv**

  このモードはオートインクリメントモードにおける最終オペランドをアドレスとしてメモリデータを取得すればよい。よって grwb ステートでオペランドレジスタの値を書き換えた後、memra ステートに遷移してメモリデータの取得を行う。データを取得した後 rsltsv ステートに遷移してデコード完了である。ただし、指定レジスタに加算する値はオペランドサイズに関係なく＋4 となる。

- **オートデクリメントモード** [0x7]

  処理フロー: **dget → usealu → grwb →**

memra → rsltsv

オートデクリメントモードはオートインクリメントモードと異なり、指定レジスタデータからオペランドサイズに応じた値をプリデクリメントした値をアドレスとしてメモリリードをする必要がある。このため dget ステートでは減算に使用する値の設定だけを行う。usealu ステートで減算処理を行い grwb ステートで結果データを書き戻したあと、結果データを用いて memra ステートでメモリリードを行ってオペランドを決定する。

- **リテラルモード** [0x0,0x1,0x2,0x3]

  処理フロー: **dget → rsltsv**

  リテラルモードはそのオペランドサイズに応じて 4 種類存在するが、どれもオペランドの元をそのまま 32 ビット拡張してやれば目的の即値となる。dget ステートで仮オペランドを符号拡張して保存し、rsltsv ステートに遷移することでデコードは完了する。

- **ディスプレイスメントモード** [0xa,0xc,0xe]

  処理フロー: **dget → usealu → memra → rsltsv**

  このモードの場合、dget ステートではディスプレイスメントを命令キューから取得する。この際アドレッシングモードですでに取得サイズが決定しているのでそれに従ったサイズで取得する。そして usealu ステートで取得データと指定レジスタ内容とを加算し、その結果を持って memra ステートにてメモリリードをしてオペランド確定となる。

- **ディスプレイスメントディファードモード** [0xb,0xd,0xf]

  処理フロー: **dget → usealu → memra → memra → rsltsv**

  このモードはディスプレイスメントモードの最終オペランドをアドレスとしてメモリリードを行って取得できるメモリデータがオペランドとなるので rsltsv ステートへ遷移する前に再度 memra ステートを通ればよい。

- **インデックスモード** [0x4]

  処理フロー: **dget → usealu → regf → dget → 基底オペランドのアドレッシングモードに依存 → indexadd → memra → rsltsv**

  処理フローの中間部分が基底オペランドのアドレッシングモードに完全依存のため、まず最初にオペランドサイズに応じて決定される乗算で使用される乗数を dget ステートで決定する。このとき、乗数のとりうる値は"1,2,4,8,16"のいずれかであったため、本互換プロセッサでは乗算ではなく、シフト処理で置き換えることにした。そこで dget ステートではシフト量を設定している。usealu ステートでシフトを行った結果を保存し、regf ステートへ遷移する。この際にインデックスモードであるというフラグをセットしておく。本互換プロセッサの仕組みではアドレッシングモードデコードが完了すると必ず rsltsv ステートへ遷移することになるため、rsltsv 遷移かつインデックスモードフラグがセットされていた場合、indexadd ステートへ遷移するようにした。こうして第一次オペランドと基底オペランドをデコードして求めたオペランドとを加算し、その結果を持って memra ステートでメモリリードを行い、最終オペランドが決定できる。

- **プログラムカウンタアドレッシングモード**

  - **イミディエイトモード** [0x8]

    処理フロー: **dget → rsltsv**

    このモードはプログラムカウンタを汎用レジスタとして使用したオートインクリメン

トモードと同じである。しかし、命令キューがあるためオートインクリメントモードと同じようにメモリリードをする必要は無いオペランドサイズに従って定数を命令キューから取得した後、rsltsv へ遷移してデコード完了となる。

– **アブソリュートモード** [0x9]

処理フロー: **dget → pclrelaadd → memra → rsltsv**

このモードもイミディエイトモードと同じく、命令キューを利用することで効率よく処理できる。命令キューから取得するサイズは 4 バイト固定なので dget ステートで命令キューからデータを取得する。その後、pcrelaadd ステートでプログラムカウンタの値を更新した後、memra ステートへ遷移し、dget ステートで取得した値をアドレスとしてメモリリードをして得た値がオペランドとなる。

– **リラティブモード** [0xa,0xc,0xe]

処理フロー: **dget → pclrelaadd → usealu → memra → rsltsv**

基本的には汎用レジスタアドレッシングモードのディスプレイスメントモードとまったく同じデコードをすればよいのだが、プログラムカウンタの値を利用するため dget ステートの後、pcrelaadd ステートでプログラムカウンタを更新する。

– **リラティブディファードモード** **0xb,0xd,0xf**

処理フロー: **dget → pcrelaadd → usealu → memra → memra → rsltsv**

このモードも基本的にはディスプレイスメントディファードモードと同じデコードでよい。ただし、プログラムカウンタの値を利用するため、dget ステートのあと pcleraadd ステートにてプログラムカウンタの値を更新する必要がある。

4.9 命令実行処理

4.9.1 命令実行とは

命令に応じた処理を実行することである。

4.9.2 動作仕様

命令実行は当然のことながら各命令によって処理は様々である。ここでは各命令の詳細な処理を記述することはせず、本互換プロセッサの命令実行部の特徴的処理について解説する。

A. 起動方法

命令実行は命令実行ステージにおいて行われる。この命令実行ステージは 1 命令を処理する際に 1 回だけ起動するのではなく、2 回起動する仕様となっている。

まず、命令デコードが終わった際に命令デコードステージから命令実行ステージは起動される。この際に、命令実行ステージは初期ステートである命令判別ステートが動作するようになっている。このステートは、命令デコードステージが命令実行ステージを立ち上げる際に利用したタスクを判断し、現在デコードされている命令、つまり実行する命令がどのようなものであるかを判断する。そして、各命令ごとに用意された命令ステートへ遷移すると共に、自らのステージを終了させる。

次にこの命令実行ステージが起動するのはすべてのオペランドフェッチが完了したときである。すべてのオペランドフェッチが完了するとオペランドフェッチステージの rsltsv ステートから命令実行ステージは起動される。起動されると、1 回目の起動の際に移動しておいた命令ステートから動作を開始する。

以上のような１命令２回起動方法を用いることで、オペコードのデコード回数を減らすことができる。その代わりに命令実行ステージのステートレジスタのセレクタが増加する。

ライトバックするデータの前処理

２回目に起動された命令実行ステージは当然のことながら命令実行を行う。命令実行が完了した後はライトバックステージへと遷移するのだが、ライトバックするデータを整形してやる必要がある。VAX11 のアーキテクチャではレジスタに対してライトバックを行う際、オペランドサイズに指定された部分のみが変更されるということになっている。
わかりやすいようサンプルを用いて説明する。"0x11111111"というデータが格納されているレジスタがあったとして、そのレジスタがライトバック先であったとする。この際のある命令の実行結果が"0xffffffff"という値であった。ここで、命令のディスティネーションオペランドサイズがバイト指定であった場合、ライトバックした後のレジスタは"0x111111ff"とならなければならない。つまり"0xffffffff"になってはいけない。
上記のようなアーキテクチャのため、命令の実行結果をそのままレジスタやメモリに書き戻すことはできない。そこで本互換プロセッサではライトバック先がレジスタの場合命令実行ステージからライトバックステージに遷移する前に fixtype というステートを通るように設計した。fixtype ステートでは、オペランドサイズフラグに従ってライトバック用に実行結果データを整形する。メモリがライトバック先の場合については、メインメモリがバイト、ワード、ロングワードの３種類のデータライト方式に対応していると仮定しているため、整形は行わない。しかしながら、ロングワード書き換えにしか対応していないようなメインメモリを使用する可能性を考慮して、とりあえず fixtype ステートは通らせている。

fixtype ステートを通った後のライトバックデータは、ライトバックステージが整形をする必要なく単純にレジスタに対して書き戻し処理が行えることになる。メモリがライトバック先の場合はライトバックステージで下位ビットを切り出してライトバックする。

fixtype ステートに遷移させると、遷移するために１クロックかかるため、MOVE 命令など頻出命令については独自の整形処理部を持たせ共通化を行っていない。

条件ビット処理

加算命令や減算命令などにおけるオーバーフローやキャリー, ボローなどは演算に使用するオペランドサイズによって、検出に使用するビットが異なる。

サンプルとして、ADDB2 いう命令と ADDW2 という命令を例に解説する。ADDB2 という命令はサイズがバイトのオペランド２つを加算して、バイトオペランドとして結果をライトバックする命令である。ADDW2 という命令はサイズがワードのオペランド２つを加算して、ワードオペランドとして結果をライトバックする命令である。この状態で ADDB2 の加算においてキャリーが発生したとする。８ビットと８ビットを加算した結果を８ビットに格納できず、キャリーがでるという状態である。しかし加算結果を 16 ビットに格納するのであれば、キャリーはキャリーではなくただの９ビット目の値である。

本互換プロセッサではオペランドサイズが 32 ビット以下の場合、どのようなオペランドも演算前に 32 ビットのゼロ拡張が行われている。これは、加算器などの演算器はすべて 32 ビットのデータに対して演算を行うものとして構成したためである。このため、加算器などの演算器は現在処理している命令のオペランドサイズがどのようなものであるか知ることはできない。そこで本互換プロセッサにおいて、加算器などの演算器はバイト、ワード、ロングワードのすべての場合におけるオーバーフローやキャリー, ボローなどを検出し、すべてを出力する。制御部 (命令実行部) は、オペランドサイズフラグに従って必要な出力を選択し、条件ビットにセットする。

以上が本互換プロセッサの命令処理部の特徴的な部

分である。その他にも乗算アルゴリズムに改良したBooth のアルゴリズムを用いるなどの工夫をし、処理速度や動作速度の向上を狙っている。

4.9.3 命令実装状況

現在までに整数演算命令、制御命令については 9 割以上実装が完了している。特殊命令についても 5 割程度の命令が実装済みである。加えて、gcc のマシンディスクリプションファイル (md ファイル) をチェックし、gcc がコンパイル時に用いる命令についてはすべて実装をした。例として movc(文字列移動命令) などである。その他の命令についても逐一実装していく予定である。

4.10 ライトバックステージの処理について

4.10.1 ライトバックとは

ライトバックとは命令によって求められた結果をレジスタやメモリに書き戻すことを指す。

4.10.2 動作仕様

本互換プロセッサはライトバックステージにおいてライトバック処理を行う。VAX11 のアーキテクチャでは"ライトバック先がレジスタ", ライトバック先がメインメモリ","ライトバック無し"の 3 種類のライトバック処理が存在する。本互換プロセッサではライトバックステージを上記の 3 通りの処理に応じたタスクで起動することで、ライトバック処理を実現している。ライトバック先がレジスタの場合、ライトバック用に用意されたレジスタの内容をライトバック先レジスタに書き出せばよい仕様となっている。ライトバック先がメモリの場合、オペランドフェッチの際に求めて保存しておいたディスティネーションオペランドのメモリアドレスもしくはディスティネーションオペランドレジスタの値をアドレスとして、メインメモリにライトバック用レジスタの内容をライトバックする。この際、オペランドサイズフラグに従って、ライトバックする際のデータサイズを調整する、

5 性能評価

開発した互換プロセッサについて性能評価を行った。

5.1 Quartus2 による論理合成結果

開発した互換プロセッサを Quartus2 Ver4.1 WebEdition を用いて論理合成した。論理合成時のターゲットデバイスは Altera の Stratix EP1S10F780C5ES を使用した。論理合成結果を表 4 に示す。

論理合成ツール	Quartus2 Ver4.1 WebEdition
ターゲットデバイス	Stratix EP1S10F780C5ES
動作周波数	52.55Mhz
LEs	6118LEs

表 4: 開発した互換プロセッサの論理合成結果

Stratix の EP1S10 シリーズは最大ロジックエレメント数が 10570LEs であるため、約 6 割デバイスを使用していることになる。今後、未実装の機能を実装する上で、実質使えるロジックエレメント数は 3000LEs 程度であると考えられる。このため、回路規模については縮小を図り今後の未実装機能に利用できるロジックエレメント数を増加させたいと考えている。

動作周波数については開発当初の目標は 100Mhz と考えていたが、現在は 70Mhz を目標にしている。これは思ったよりも 100Mhz という目標値は厳しく、実現の可能性が低いためである。しかしながら、動作周波数の目標値を下げるかわりに、CPI を小さくするように設計を行うことで処理速度の低下を防ぎたい。開発途中で 70Mhz 以上の動作周波数は実現しているため、70Mhz を目標とした。

回路規模の縮小, 動作周波数の向上の 2 点を目指すために、汎用レジスタを EAB(Embedded Array Block) と呼ばれるメモリブロックを利用して論理合成することを考えている。汎用レジスタによるロジックエレメント数は約 600LEs であるため、EAB にマップすることである程度回路規模は縮小するのではないかと考えている。ただし、EAB はメモリブロックというようにアドレスの概念が存在する。このため、EAB からデータを読み込む際にはアドレスをラッチする必要がある。つまり、データ要求を出してから 1 クロック経

評価プロセッサ	DHRYSTONE VALUE	VAX MIPS	動作周波数 (Mhz)
開発したプロセッサ-最適化無し-	5764	3.4	52.55
開発したプロセッサ-O1-	18544	10.8	
開発したプロセッサ-O2-	18089	10.6	
開発したプロセッサ-O3-	18623	10.9	
開発したプロセッサ-Os-	19280	11.2	
VAX11/780	1714	1	5

表 5: DHRYSTONE BENCHMARK 結果

過しなければデータが取得できない。よって、EAB を利用する際には現在の処理プロセスを変更しなければならないため、現在は EAB に汎用レジスタをマップしていない。

5.2 DHRYSTONE BENCHMARK による命令処理効率の評価

開発した互換プロセッサを評価するため、DHRYSTONE BENCHMARK を VAX 用 gcc でコンパイルし、互換プロセッサ上にて動作させた際の DHRYSTONE VALUE を調査した。

DHRYSTONE BENCHMARK のコンパイルにあたっては最適化オプション"-O1"，"-O2","-O3","-Os" をそれぞれ付加した場合と、最適化オプションを使用しない場合の全部で 5 種類のバイナリを用意した。メモリレイテンシについては 1 クロックとしてベンチマークを行った。つまり、データ要求を MMU に対して発行すると次のクロックでデータが取得できる。これは、DHRYSTONE BENCHMARK のプログラムサイズから判断して、利用するメモリアドレスとそのデータは TLB 及びキャッシュ内に入るため、オリジナルの VAX11/780 でも 1 クロックあればメモリデータは取得できると考えたためである。

BENCHMARK 結果を表 5 に示す。開発した互換プロセッサはオリジナルの VAX11 と比較して最大で 11.2 倍の処理速度を持つことがわかった。

最適化オプションなしの場合に比べ、最適化オプションありの場合の DHRYSTONE VALUE のほうが約 3 倍程度良い値となっている。最適化オプション"-O1" 以上を付加した場合、変数のレジスタ割り当てや多重ジャンプの最適化などが行われる。上記の処理による高速化も確実に BENCHMARK 結果に好影響を与えていることは間違いない。しかしながら、今回の大幅な BENCHMARK 結果の上昇には VAX11 専用命令の使用が一番の要因ではないかと考える。

最適化オプション無しの場合、DHRYSTONE BENCHMARK 内の文字列操作 (比較, コピー, 移動など) には strcpy や strcmp などの文字列関数が利用される。この文字列関数内では 1 文字づつ文字をコピー, 比較をするという処理を複数回繰り返すことで、目的の処理を実現している。このため、処理に要するクロック数が増加し、DHRYSTONE VALUE に悪影響を与えていた。

一方、最適化オプションを付加した場合、strcpy や strcmp などの関数の変わりに movc,cmpc といった VAX11 専用を用いている。movc,cmpc は strcpy,strcmp に比べ非常に少ないクロック数で同じ処理を行うことができる。このため、VAX11 専用命令を使うことで 1DHRYSTONE LOOP に必要なクロック数が減少し、結果として DHRYSTONE VALUE が上昇したと考えている。

最適化オプションを付加した 4 種類のバイナリの実行結果もそれぞれ微妙に違いが存在する。ここで、"-O1" と"-O2" とを比較すると、本来より最適化されているはずの"-O2" のほうが DHRYSTONE VALUE がわずかに悪いことがわかる。

原因としてまず命令の変化による悪影響が考えられる。例えば、コードを最適化して 2 命令を 1 命令にしたとする。しかし必ずしもその処理で高速化するとは限らない。なぜならば、今回の命令実装は私独自のものであるため各命令に必要なクロック数は私にしかわからない。ところが、VAX11 用の gcc のマシンディスクリプションを記述したのは、私以外の方でありお

最適化オプション	キューエントリ数	メモリレイテンシ									
		1clk	2clk	3clk	4clk	5clk	6clk	7clk	8clk	9clk	10clk
無し	2	6406	5677	5088	4583	4156	3783	3451	3168	2929	2716
	4	5764	5181	4704	4162	3797	3477	3210	2914	2540	2470
	差分	642	496	384	421	359	306	241	254	389	246
”-O1”	2	18544	15593	13414	11720	10372	9271	8381	7616	6970	6429
	4	18544	15565	13350	11618	10232	9192	8302	7495	6840	6275
	差分	0	28	64	102	140	79	79	121	130	154
”-O2”	2	18089	15265	13171	11542	10234	9163	8301	7560	6930	6391
	4	18089	15239	13109	11435	10086	9080	8221	7430	6797	6233
	差分	0	26	62	107	148	83	80	130	133	158
”-O3”	2	18623	15746	13604	11934	10589	9486	8598	7833	7184	6627
	4	18623	15718	13541	11830	10435	9394	8510	7695	7044	6462
	差分	0	28	63	104	154	92	88	138	140	165
”-Os”	2	19280	16159	13871	12109	10705	9553	8631	7840	7172	6605
	4	19280	16129	13802	11988	10541	9452	8536	7691	7018	6434
	差分	0	30	69	121	164	101	95	149	154	171

表 6: 命令キューのメモリアクセス効率評価
命令キューのキューエントリ数とメモリレイテンシを変更した際の各 DHRYSTONE VALUE を調べた。
差分は 2 エントリ時の結果から 4 エントリ時の結果を減算したものである。

そらくその方はオリジナルの VAX11 用に記述を行っているであろうと推測される。当然、同じ命令だとしてもオリジナルの VAX11 と本互換プロセッサでは処理に必要なクロック数は変わってくる。オリジナルの VAX11 用に最適化をするようにマシンディスクリプションが記述されている以上、本互換プロセッサでは最適化によってトータルの処理に必要なクロック数が増加することは十分に考えられる。

次に、命令キューと命令実行部のメモリアクセスタイミングによる影響が考えられる。メモリアクセス要求が命令キューと命令実行部から発行された場合は、命令実行部が優先されるが命令キューがメモリからデータを取得している最中に命令実行部がメモリデータの読み書きの要求を出した場合、MMU によって要求が拒否され、命令実行はストール (一時停止) する。この場合、命令キューがメモリデータを取得するまでは停止していなくてはならないため、トータルの処理クロックは増加する。このメモリアクセスタイミングが同時に発生する回数がたまたま”-O2”のほうが多く、結果として DHRYSTONE VALUE が悪化した可能性は否定できない。

以上のようなことから”-Os”が一番良い結果を出したことについても、かならずしも命令コード量が減ったからではないと考えられる。

5.3 命令キューのエントリ数によるメモリアクセス効率の評価

命令キューのキューエントリ数とメモリアクセス効率の関係について評価を行った。まず、メモリレイテンシとキューエントリ数を変化させ、DHRYSTONE BENCHMARK にて DHRYSTONE VALUE を求めた。結果を表 6 に示す。

ベンチーマーク結果より、キューエントリ数が 2 エントリのほうがすべての組み合わせにおいて DHRYSTONE VALUE が良くなっている。このことから、キューサイズが 4 エントリの場合はキューのメモリアクセス回数が増加し、結果として命令実行部のメモリアクセスが妨げられていることがわかる。また、メモリレイテンシが大きくなるにつれ、DHRYSTONE VALUE

の開きも大きくなることからメモリレイテンシが大きい場合、2 エントリのほうがメモリアクセス効率が高いといえる。

6　開発フロー

今回の互換プロセッサの開発フローについて述べる。開発は 2004 年の 4 月から開始した。

- **2004 年 4 月**
  VAX11/780 のアーキテクチャの調査を開始すると同時に、命令キュー及びステージ構成の仕様を決定し、その開発を始めた。この時点での目標動作周波数は 100MHz に設定した。

- **2004 年 5 月**
  命令キュー及び各ステージの基本構成の開発が完了したため、アドレッシングモードデコーダ開発を開始した。
  デバッグ時にメモリからのデータ取得の必要性が生じたため、MMU(アドレス変換が未実装であるため、実質的には Bus Arbiter である) を実装した。

- **2004 年 6 月**
  アドレッシングモードデコーダの開発が終了した。これにより命令実行部の実装を始め、算術命令などのためにバレルシフタ、加算器を実装した。

- **2004 年 7 月**
  整数演算命令の実装がすべて完了した。この時点で動作周波数を確認するため、Altera の stratix をターゲットとして論理合成を行ったが、動作周波数は 32MHz となった。目的の動作周波数と比べ非常に低い動作周波数であったため、各レジスタやＡＬＵなどのファンイン, ファンアウトを見直し、大幅に回路を書き直す作業を始める。

- **2004 年 8 月**
  書き直したプロセッサコアを Altera の stratix をターゲットに再度論理合成を行った。結果として修正作業の効果があり 70MHz 程度の動作周波数で動作させることができた。このため、命令実装作業に戻り、分岐命令の実装を開始する。

- **2004 年 9 月**
  アドレッシングモードデコーダにおけるバグを発見する。プログラムカウンタアドレッシングモードのデコード時にＰＣの値が正常に更新されていなかったため、目的のオペランドを取得できないというバグであった。pcrelaadd ステートをアドレッシングモードデコードのルートに加えることで問題を解決した。上記のバグを修正したのがきっかけで、プログラムカウンタの更新場所をアドレッシングモードデコーダを抜ける箇所へ変更した。理由として、分岐命令の多くが分岐時にプログラムカウンタの値に変位を加えて分岐をするため、ライトバックステージでプログラムカウンタを更新していては遅いということがあげられる。したがって、分岐命令内でプログラムカウンタを更新した上で変位を加算することになるため、rsltsv ステートでプログラムカウンタを更新したほうが効率的であると判断したのが主な変更要因である。この月で分岐命令の大半の実装が完了した。

- **2004 年 11 月**
  第 1 週目で分岐命令の実装が完全に完了した。サブルーチンコール命令などの実装も完了したため、DHRYSTONE BENCHMARK を用いてプロセッサコアの評価を行う準備に入った。ここで、gcc を用いて DHRYSTONE BENCHMARK のバイナリを生成したが、分岐アドレスがきちんと設定されていないバイナリが生成されてしまった。この原因を調べた結果、文字列操作用ライブラリをリンクさせていなかったことが原因だと判明した。このため、文字列操作用ライブラリをリンクさせ、バイナリを生成して、正常に動作するバイナリデータを用意した。

  次に、DHRYSTONE STONE のアセンブリコードのチェックを行い、未実装命令を抽出した。未実装であったのは、文字列操作命令 (文字列移動命令)

と特殊操作命令 (case 文の動作をする命令) であった。それらの命令を実装したのち、実際に生成したバイナリの DHRYSTONE BENCHMARK を動作させながらデバッグを行ったところ、バグを発見した。このバグはオペランドの属性に起因するバグであった。VAX-11 のオペランドは命令によって、リードオンリー, ライトオンリー, リードライトなどの属性が存在する。この属性を無視して、どのオペランドもアドレッシングモードデコードをしてデータをリードしていたため、データが存在しない (Unknown な) ライトオンリーのアドレスのデータをフェッチし、ハザードが起きていた。そこで命令デコード時に各オペランドの属性に対するフラグを用意することで、このバグを解決した。

また、DHRYSTONE BENCHMARK のデバッグ時にＣ言語のソースコードとアセンブリコードを見ながらデバッグする方法をとった事でＣ言語のコードがどのようなアセンブリコードになるかの対応関係を把握することができた。デバッグを続け、gcc の最適化オプション無しの状態の DHRYSTONE BENCHMARK を処理したところ 1DHRYSTONE LOOP に 14000clock 程度かかった。そこで最適化オプション"-O1"でコンパイルし、再度評価すると 2387clock まで処理にかかったクロックが減少した。このことにより、最適化の効果を実感すると共に、どのような部分において最適化されたかの調査を行った。結果、文字列操作や CASE 文の処理を VAX-11 固有の命令に置き換えて大幅に処理に必要なクロックを削っていたことがわかった。更なる高速化を狙って最適化オプション"-O2"を試してみたが、実装していない命令が頻出したため、今回は評価に使用しないこととした。

- **2004 年 12 月**
  最適化オプション"-O2"でコンパイルした DHRYSTONE BENCHMARK を実行できるようにいくつか命令を実装した。

- **2005 年 1 月**
  gcc の VAX11 用マシンディスクリプションなどをチェックし、gcc が出力するアセンブリデータに利用される命令をリストアップし、それらすべての命令を実装した。
  最適化オプション"-O1"、"-O2"、"-O3"、"-O4"をそれぞれ付加してコンパイルした DHRYSTONE BENCHMARK を用いて開発したプロセッサの評価を行った。その結果、一部の分岐および文字列命令が性能を落としている原因と判断し改良を加えたうえで再度評価を行い、処理速度向上を確認した。

以上のような流れで開発を行ってきたが、プロセッサの開発を学部生 1 人が 6ヶ月で行えたのは、先に述べた SFL 言語を中心とする開発環境を用いたことによるものであるといえる。

7 今後の開発予定

今回の開発の最終目的はＯＳのブートである。よって今後は MMU の未実装機能, 割り込み処理機能, 未実装命令, 各種ＩＯなどの実装を行ったうえで NetBSD をブートさせるところまで開発を行う予定である。

8 教育的効果

現在まで開発を行ってきて、自分自身プロセッサに関する知識がかなり向上したと感じている。開発を始める前はあまり知らなかったアドレッシングモードデコードなどの方法であったり、関数コールなどの具体的な処理など、かなりの知識を得ることができた。また、DHRYSTONE BENCHMARK を動作させる際のデバッグにおいては、C 言語で書かれたコードと変換したアセンブリコードを対応させながらデバッグを行ったことで、C 言語での記述とアセンブリコードの対応関係を学ぶことができた。と同時にアセンブリコードと C 言語の対応関係を利用したデバッグ手法を学んだ。今後 OS のブートというところまで開発を続けることで、より一層知識を深めることができると確信している。以上のようなことから、プロセッサ開発による教育効果は非常に大きいといえる。

9　まとめ

SFL 言語をベースとした開発完了にて VAX11/780 命令互換 32 ビットプロセッサコアを開発した。プロセッサコア開発における教育的効果はハードウェア教育, ソフトウェア教育のどちらに対しても非常に大きい。今後は、NetBSD を動作させせることで、コンピュータシステム全体の構築を行う予定である。

参考文献

1) 日本ディジタルイクイップメント株式会社 教育部 :「ＶＡＸアーキテクチャハンドブック」, 共立出版 , 1984.

2) 中森 章 :「ＴＥＣＨＩマイクロプロセッサ・アーキテクチャ入門」, ＣＱ出版 , 2004

3) Compaq :「VAX MACRO and Instruction Set Reference Manual」, PDF release , 2001

4) 飯田佳洋 清水尚彦: 「ＰＤＰ 11 アーキテクチャへの GCC/GAS 及びリアルタイムカーネルの移植」, 情報処理学会 IPSJ-SIG-OS-92-12 , 2003

5) 飯田佳洋 : 「PARTHENON/SFL を用いた PDP-11 互換コンピュータシステムの開発」, 第 22 回パルテノン研究会 , 2003

6) 飯田佳洋 清水尚彦: 「組み込み向け PDP-11/40 互換プロセッサの開発」, SWEST , 2003

7) 大山将城 清水尚彦：「i8086 命令互換プロセッサ開発によるシステム設計教育」 , SWEST ,2004