

32ビット仮想コンソール



システム仕様

パート3:プロセッサー(CPU)

資料作成日2023.01.22年Carra著

これは何だ?

このドキュメントは、Vircon32システム仕様のパート3です。この一連のドキュメントは、Vircon32システムを定義し、その機能と動作を詳細に説明する完全な仕様を提供します。

この仕様の主な目的は、Vircon32システムとは何か、およびゲームシステムが準拠していると見なされるために実装される必要がある方法の標準を定義することです。また、Vircon32は仮想システムであるため、これらのドキュメントの重要な2番目の目的は、独自のVircon32実装を作成するための知識を誰にでも提供することです。

Vircon32について

Vircon32プロジェクトはCarraによって独自に作成されました。Vircon32システムとその関連資料(ドキュメント、ソフトウェア、ソースコード、アート、およびその他の関連要素を含む)は、元の作成者が所有しています。

Vircon32は無料のオープンソースプロジェクトで、誰でもゲーム機をプレイし、そのためのソフトウェアを開発できるようにすることを目的としている。この詳細については、使用可能な各ソフトウェアに含まれているライセンステキストを参照してください。

このドキュメントについて

このドキュメントは、Creative Commons Attribution 4.0 License(CC BY 4.0)に基づいて提供されています。ライセンスの全文は、Creative CommonsのWebサイトで読むことができます。

<https://creativecommons.org/licenses/by/4.0/>

まとめ

仕様のパート3では、コンソールプロセッサ、つまり中央処理装置(CPU)を定義しています。このドキュメントでは、このチップの動作、内部要素、命令セットアーキテクチャ、および実行プロセスについて説明します。

[1 Introduction 3](#_Toc83493794)

[2 CPU Registers 4](#_Toc83493795)

[3 The stack 4](#_Toc83493796)

[4 String operations 5](#_Toc83493797)

[5 Control flags 6](#_Toc83493798)

[6 Responses to control signals 6](#_Toc83493799)

[7 Instruction format 7](#_Toc83493800)

[8 Instruction set 7](#_Toc83493801)

[9 The CPU processing cycle 11](#_Toc83493802)

[10 Instruction processing 12](#_Toc83493803)

[11 Detecting hardware errors 36](#_Toc83493804)

[12 Processing hardware errors 37](#_Toc83493805)

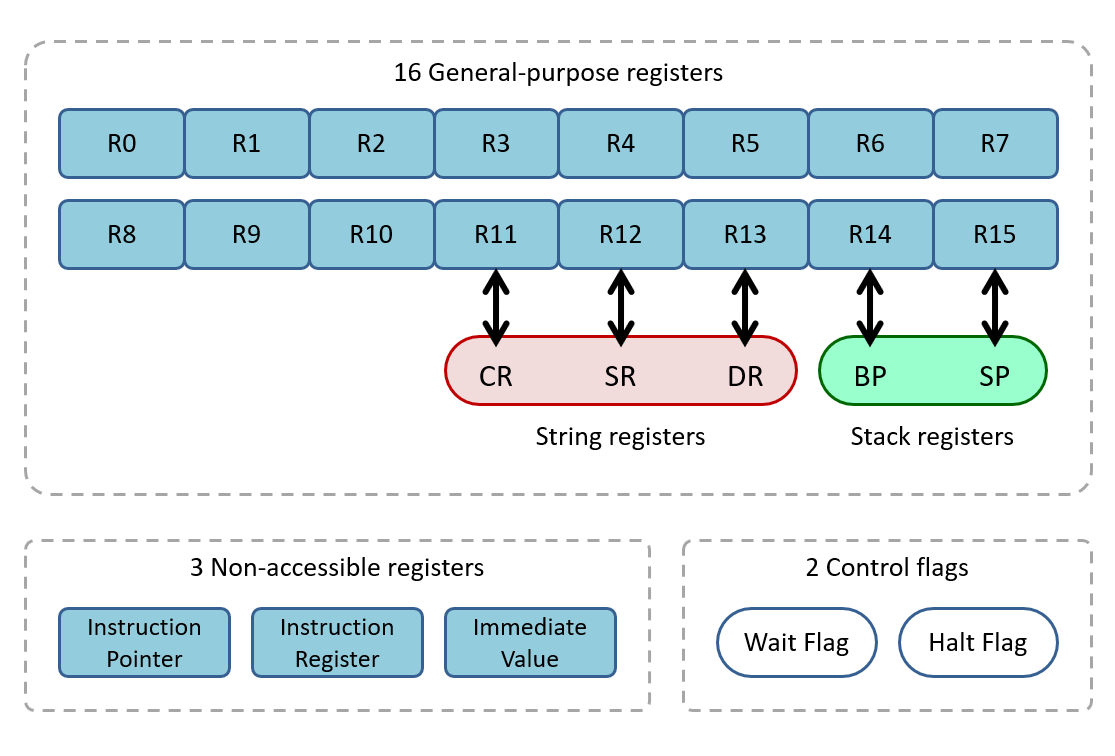
## 1はじめに

CPUは、Vircon32コンソールのメインチップです。その機能は、プログラム命令を読み込み、それらを順番に実行して、プログラムの記述された動作を生成することです。この実行の一部として、CPUは他のコンソールコンポーネントとも通信し、他のチップにコマンドを送信してその機能を制御します。CPUは常に通信を開始するチップであるため、メモリバスと制御バスの両方にマスターデバイスとして接続されます。

このプロセッサは、純粋な32ビットアーキテクチャを特徴としています。すべてのVircon32コンポーネントと同様に、32ビットデータでのみ動作します。その命令セットは、x86プロセッサファミリに大まかに基づいていますが、Vircon32の機能セットに適合しています。また、64命令のみとなるように簡素化されています。これらの命令は、他のCPU設計と比較して、バリエーションも少なくなっています。

### 1.1 CPU内部エレメント

この図は、CPUに存在する内部要素の概要を示しています。次の各項では、これらの各要素について説明し、CPUがスタック(ここでは緑色で表示されている領域)または文字列処理(赤色の領域)に関連する操作を実行している場合に、一般的なレジスタの一部が特定の機能を受け取る方法を示します。



## 2つのCPUレジスタ

CPUは、32ビットレジスタのセットを特徴としています。その機能は、実行に必要なデータを格納することです。レジスタは、2つの異なるグループに分けることができます。

### 2.1汎用レジスタ

汎用レジスタは、プログラムの実行をサポートするためのものであり、プログラマが直接アクセスできる唯一のレジスタです。CPUには、R0～R15という名前の16個の汎用レジスタの配列があります。

これらのレジスタの最後の5つは、次の章で説明するいくつかのCPU機能(スタックおよび文字列操作)でも使用されます。そのため、これらの役割を反映した代替名も与えられます。

### 2.2内部レジスタ

これらのレジスタはCPU自体の動作のために予約されているため、プログラムから直接アクセスすることはできません。3つの内部レジスタは次のとおりです。

#### 命令ポインタ

CPUが次のプログラム命令を読み取るメモリアドレスを保持します。

#### 命令レジスタ

最後の読み取り命令、つまり現在実行されている命令を格納します。

#### 即時価値

最後の読み取り命令が即値を使用している場合は、その値が読み取られ、ここに格納されます。

## 3スタック

CPUは、Last-In、First-Outポリシーに従って値を格納および抽出するために使用されるハードウェアスタックを実装します。その主な機能は、プログラムフローを指示するための呼び出しおよびリターン命令によって使用されます。上位レベルの言語も通常、スタックを使用して「スタックフレーム」を格納し、言語の関数呼び出しコンテキストを保存します。プログラマは、スタックから値を明示的にプッシュおよびポップすることもできます。

### 3.1スタック操作

スタックはRAMメモリの最も高い位置に配置され、下に向かって成長します。スタック操作では、最後の2つの汎用レジスタが使用されます。通常の名前に加えて、スタック内の役割から次のエイリアスとして参照することもできます。

R14=BP(ベースポインタ)

R15=SP(スタック・ポインタ)

ベースポインタはスタックの現在の最高位置(一般にスタックの最下部と呼ばれる)を指し、スタックポインタはその最低位置(スタックの最上部)を追跡します。

#### スタックにプッシュするプロセス

CPUがスタックに値をプッシュするときは、次の手順に従います。

以降のセクションでは、この操作は次のように表されます。Stack.Push(値)

(1)メモリ[StackPointer]=Value

(2)StackPointer-=1;

#### スタックからポップするプロセス

CPUがスタックからレジスタに値をポップするとき、CPUはこれらのステップに従う。

以降のセクションでは、この操作は次のように表されます。レジスタ=Stack.Pop()

(1)StackPointer+=1

(2)レジスタ=メモリ[StackPointer]

## 4つの文字列操作

CPU命令の中には、単一のプログラム命令が一連の連続したメモリアドレスに対して動作できるように設計されているものがあります。CPUの中には、テキストを表さない場合でも、これらのデータセットを「文字列」と呼ぶものがあります。

文字列操作では、スタックレジスタの直前にある3つの汎用レジスタを使用します。通常の名前に加えて、文字列操作での役割から、次のエイリアスとして参照することもできます。

R11=CR(カウントレジスタ)

R12=SR(ソースレジスタ)

R13=DR(デスティネーションレジスタ)

命令が複数のアドレスを処理できるようにするために、CPUはカウントレジスタが0に達するまで同じ命令を自動的に繰り返します。このようにして、CPUはより効率的なバルク処理を可能にする高速内部ループを実装します。命令が処理されるたびに、カウンタは自動的に減少します。それに加えて、最大2つのポインタレジスタ(送信元アドレスと宛先アドレス用)が自動的に増加します。

プログラムが文字列演算を使用する方法は、最初にSRとDRに正しいソースアドレスと宛先アドレスを配置し、次にCRに反復回数を書き込み、最後に文字列命令を使用することです。

## 5制御フラグ

CPUには、特定の状況でCPUの動作を一時停止するために使用される2つの制御フラグがあります。これらはそれぞれ1ビットであり、1の場合は設定されていると解釈され、0の場合は設定されていないと解釈されます。これらのフラグの意味と使用方法は次のとおりです:

#### 停止フラグ

設定すると、次にリセットするか電源を入れるまでCPUの動作を停止します。

#### 待機フラグ

設定すると、次のフレームが開始されるまでCPUの動作を停止します。

プログラマは、これらのフラグの両方を異なる目的で設定できます。Waitフラグを使用すると、プログラムは独自の実行速度を制御できます。一方、プログラムが終了した場合は、終了メカニズムとしてCPUを停止できます。

## 6制御信号に対する応答

コンソール内のすべてのコンポーネントと同様に、制御信号がトリガーされるたびに、CPUはそれを受信し、そのイベントを処理するための応答を生成します。各制御信号に対して、CPUは次のアクションを実行することによって応答します:

#### リセット信号:

* HaltフラグとWaitフラグは0にクリアされます。
* すべてのレジスタは、整数値0にクリアされる。
* スタックは、BPおよびSPを最後のRAMアドレス、すなわち0x003FFFFFに設定することによってリセットされる。
* 命令ポインタは、BIOSの起動アドレス(0x10000004)を指すように設定されます。

#### フレーム信号:

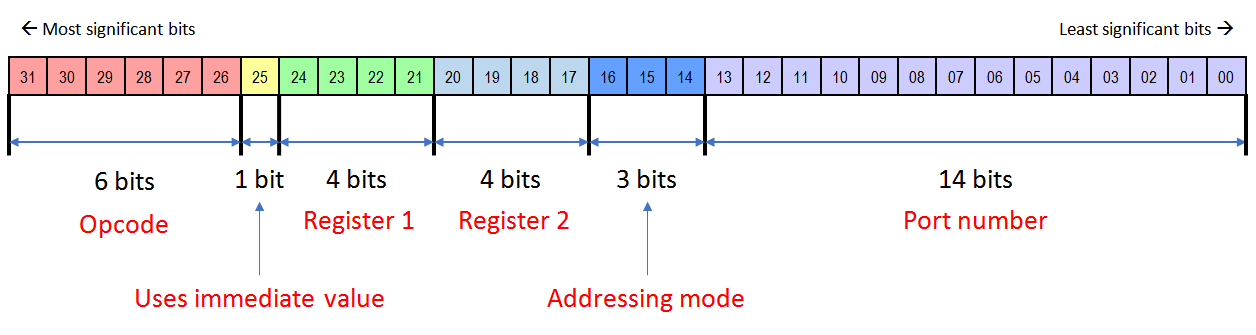
* Waitフラグが0にクリアされます。

#### サイクル信号:

* いずれかの制御フラグが設定されている場合、CPUは何も行いません。
* それ以外の場合、CPUは第9章で説明する処理サイクルを実行します。

## 7命令形式

CPU命令は、CPUによってフィールドのセットとして解釈される単一の32ビット値です。この図は、各フィールドの位置とサイズを示しています。



これらのフィールドはすべて符号なし整数として解釈されます。フィールド「レジスタ1」および「レジスタ2」は、命令によって使用される場合、汎用CPUレジスタの配列内のレジスタインデックス(0から15)を指定する。

あるフィールドが特定の命令によって使用されない場合、それは0に設定されるべきであるが、これは命令の正しい処理に影響を与えない。

一部の命令では、イミディエイト値と呼ばれる2番目の32ビットワードを使用する必要があります。これは、[イミディエイト値を使用]フィールドを1に設定することによって指定されます。この場合、2番目のワードが命令自体とともにメモリから読み取られます。この2番目の値はイミディエイト値レジスタに格納され、命令処理中に使用されます。

不正な形式の命令を検出するために、Vircon32システムは必要ないことに注意してください。不正な命令を処理すると、実装依存の動作が発生します。

## 8命令セット

CPUによってサポートされる異なる命令は、6ビットの符号なし整数であるオペレーションコード(命令内のオペコードフィールド)によって表されます。Vircon32 CPUにはちょうど64の異なる命令があるので、未使用のオペコードはありません。

Vircon32 CPUで使用されている命令の多くはx86命令に基づいており、ほとんどの場合同じ名前を使用しています。アセンブリ命令は、このドキュメントとVircon32アセンブラプログラムの両方で、インテルの構文を使用して表されています。

目的に応じていくつかの異なる命令グループを区別することができます。次の表は、64個の命令を機能別にまとめたものです。理解を容易にするために、これらのグループを考慮してオペコードに番号が付けられていることに注意してください。

|  |  |  |
| --- | --- | --- |
| CPU制御 | | |
| オペコード | 指示 | 簡単な説明 |
| 00 | ハイレベル | リセットするまでCPUの動作を停止 |
| 01 | 待って | 次のフレームを待機 |

|  |  |  |
| --- | --- | --- |
| ジャンプ命令 | | |
| オペコード | 指示 | 簡単な説明 |
| 02 | 日本プラスチック工業会 | ジャンプinconditionally |
| 03 | コール | 呼び出しサブルーチン |
| 04 | リセット | サブルーチンから戻る |
| 05 | ジェイティ | 真の場合にジャンプ |
| 06 | JFキー | Falseの場合にジャンプ |

|  |  |  |
| --- | --- | --- |
| 整数の比較 | | |
| オペコード | 指示 | 簡単な説明 |
| 07 | アイ・イー・キュー | 等しい整数 |
| 08 | いん | 等しくない整数 |
| 09 | アイジーティー | より大きい整数 |
| 10個 | アイジー | 以上の整数 |
| 11人 | 集合研修 | より小さい整数 |
| 12個 | イール | 整数以下 |

|  |  |  |
| --- | --- | --- |
| Floatの比較 | | |
| オペコード | 指示 | 簡単な説明 |
| 13個 | フェク | 同等の浮動小数点 |
| 14個 | んっ | 等しくない浮動小数点 |
| 15個 | フル | 浮動小数点以上 |
| 16個 | ふげ | 浮動小数点以上 |
| 17歳 | FLT[尺度係数] | より小さい浮動小数点 |
| 18歳 | フ | 浮動小数点以下 |

|  |  |  |
| --- | --- | --- |
| データの移動 | | |
| オペコード | 指示 | 簡単な説明 |
| 19歳 | モフ | 値を移動 |
| 20個 | リー | メモリ位置の実効アドレスをロード |
| 21歳 | プッ | スタックの最上位に値をプッシュ |
| 22人 | ポッ | スタックの先頭から値をポップする |
| 23人 | イン | I/Oポートからの値の読み取り |
| 24時間 | 出力 | I/Oポートへの値の書き込み |

|  |  |  |
| --- | --- | --- |
| 文字列操作 | | |
| オペコード | 指示 | 簡単な説明 |
| 25人 | モブ | 文字列を移動(ハードウェアmemcpy) |
| 26人 | 設定 | 文字列を設定(ハードウェアmemset) |
| 27人 | コンポーネント | 文字列の比較(ハードウェアmemcmp) |

|  |  |  |
| --- | --- | --- |
| データ変換 | | |
| オペコード | 指示 | 簡単な説明 |
| 28人 | 運賃保険料込み値段 | 整数を浮動小数点に変換 |
| 29個 | CFI社 | 浮動小数点を整数に変換 |
| 30人 | CIB群 | 整数をブールに変換 |
| 31人 | CFBカード | 実数をブール値に変換 |

|  |  |  |
| --- | --- | --- |
| バイナリ操作 | | |
| オペコード | 指示 | 簡単な説明 |
| 32個 | しない | ビット方式のNOT |
| 33人 | かつ | ビット単位のAND |
| 34個 | または | ビット単位のOR |
| 35歳 | 排他的論理和 | ビット単位のXOR |
| 36個 | しない | ブールNOT |
| 37人 | スッ | ビットシフト左 |

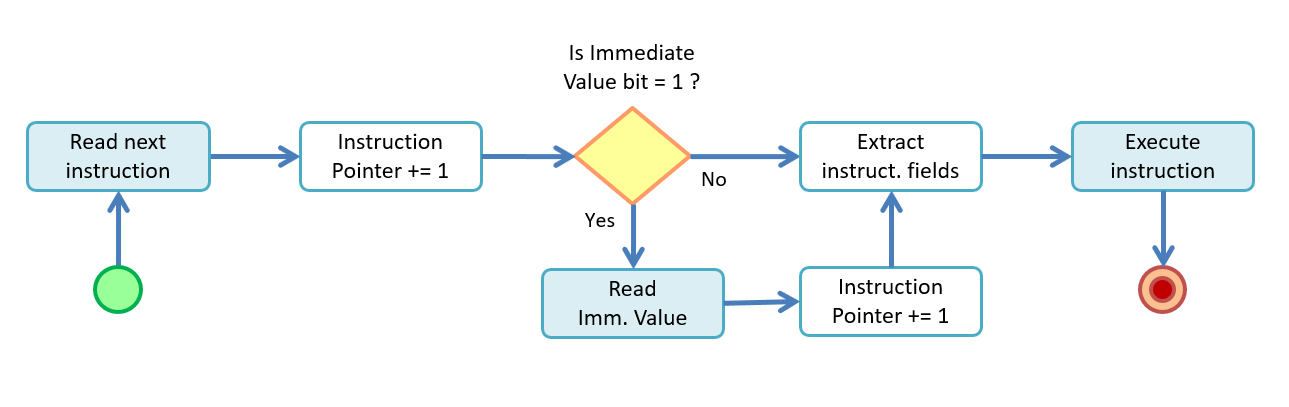
|  |  |  |
| --- | --- | --- |
| 整数演算 | | |
| オペコード | 指示 | 簡単な説明 |
| 38個 | IADD[外部参照追加] | 整数加算 |
| 39個 | ISUB[サブサーフェス] | 整数の減算 |
| 40人 | IMUL社 | 整数乗算(Integer Multiply) |
| 41人 | IDIV[イメージエディタ] | 整数除算 |
| 42人 | IMOD[イメージ管理I | 整数の係数 |
| 43人 | ISGN | 整数の符号の変更 |
| 44個 | アイミン | 整数の最小値 |
| 45歳 | アイマックス | 整数最大値 |
| 46個 | 国際航空運送協会 | 整数絶対値 |

|  |  |  |
| --- | --- | --- |
| 浮動小数点演算 | | |
| オペコード | 指示 | 簡単な説明 |
| 47人 | ぺた | 浮動小数点加算 |
| 48人 | FSUB[サブサブ] | 実数減算 |
| 49人 | フムル | 浮動小数点乗算 |
| 50人 | FDIV[ファイルディビジョン] | 浮動小数点除算 |
| 51人 | FMOD[フィーチャ編集] | フロート係数 |
| 52歳 | FSGN | 浮動小数点符号の変更 |
| 53人 | FMIN[フレーム間隔] | 浮動小数点の最小値 |
| 54個 | 最大 | 浮動小数点の最大値 |
| 55歳 | FABS社 | 浮動小数点絶対値 |

|  |  |  |
| --- | --- | --- |
| 拡張浮動小数点演算 | | |
| オペコード | 指示 | 簡単な説明 |
| 56人 | フ… | 床(丸め) |
| 57人 | CEIL[画面 | 天井(切り上げ) |
| 58歳 | まる | 最も近い整数に丸める |
| 59個 | シン | サイン |
| 60人 | アコス | アークコサイン |
| 61人 | ATAN2(航空機) | yおよびxからのアークタンジェント |
| 62人 | ログ | 自然対数 |
| 63人 | バシッ | 累乗 |

## 9 CPU処理サイクル

CPUは、そのプログラム処理をサイクルに分割します。各サイクルの目的は、プログラムからの新しい個別の命令を処理することです。CPU処理サイクルは、次のアルゴリズムによって記述されます:



青色でマークされたステップは、ハードウェア・エラーが発生する可能性のあるステップです。このような場合、このプロセスは中断され、関連するエラー処理が実行されます。この処理サイクルからのアクションは、次のように実行されます:

#### 次の命令を読む:

CPUは、命令ポインタが指すメモリアドレスを読み出し、その内容を命令レジスタに格納する。

#### 即時値の読み取り:

CPUは、命令ポインタが指すメモリアドレスを読み込み、その内容を即値に格納する。

#### 指示フィールドの抽出:

命令レジスタに格納された値は、命令ビットに符号化される異なる整数値に分割される。実装がビットフィールドを直接操作できる場合、この手順は実際には必要ない場合があります。

#### 命令の実行:

CPUは、命令のオペコードに応じて適切な処理を選択し、次のセクションで説明する64の可能なケースのいずれかを適用します。

CPUは、「Uses immediate value」フィールドを常に尊重し、そのビットを使用して別のワードを読み取るかどうかを決定する必要があることに注意してください。これは、命令が適切に形成されていない場合でも実行する必要があります。その種類の命令を処理すると、未定義の動作が発生します。

## 10命令処理

このセクションでは、各命令の構造と実行プロセスについて詳しく説明します。次のサブセクションでは、使用可能な64個のオペコードすべてについて説明します。

命令MOVを除き、すべての命令には1つまたは2つのバリアントがあることに注意することが重要です。すべての場合において、2つのバリアントを持つ命令は、「即時値を使用」ビットの値によってどちらが使用されるかを決定します。

### 00指示HLT(Halt)

#### 構造およびバリアント:

ハイレベル

#### 処理アクション:

HaltFlag=1

#### 説明:

HLTはCPUのHaltフラグをアクティブにします。これにより、次のコンソールの電源投入時またはリセット時にフラグがクリアされるまで、CPUは実行を停止します。他のコンポーネントは機能し続けることに注意してください。たとえば、SPUが音楽を再生していた場合、SPUは音楽を再生し続けます。

### 01命令WAIT

#### 構造およびバリアント:

待って

#### 処理アクション:

待機フラグ=1

#### 説明:

WAITは、CPUのWaitフラグをアクティブにします。これにより、タイマーが次のフレームの開始を通知したときにフラグがクリアされるまで、CPUは実行を一時停止します。リセットまたは電源投入アクションも、新しいフレームを開始するため、CPUの実行を再開します。

新しいフレームが開始されると、CPUは通常の順序に従って実行を再開します。つまり、WAITの直後に命令を処理します。他のコンポーネントは機能し続けることに注意してください。

### 02命令JMP(ジャンプ)

#### 構造およびバリアント:

(バリアント1):JMP{即時値}

(バリアント2):JMP{レジスタ1}

#### 処理アクション:

(バリアント1):InstructionPointer=ImmediateValue

(バリアント2):InstructionPointer=レジスタ1

#### 説明:

JMPは、オペランドで指定されたアドレスに無条件でジャンプします。この命令を処理した後、CPUは新しいアドレスで実行を継続します。

### 03 CALL命令

#### 構造およびバリアント:

(バリアント1):CALL{即時値}

(バリアント2):CALL{レジスタ1}

#### 処理アクション:

(バリアント1):  
Stack.Push(InstructionPointer)  
InstructionPointer=ImmediateValue

(バリアント2):  
Stack.Push(InstructionPointer)  
InstructionPointer=レジスタ1

#### 説明:

CALLは、指定されたアドレスに対してサブルーチン呼び出しを実行します。この命令を処理するとき、現在の命令ポインタ(すでに増分されています)はスタックの先頭に保存され、新しいアドレスで上書きされます。実行は新しいアドレスで続行されます。

### 04指示RET(リターン)

#### 構造およびバリアント:

リセット

#### 処理アクション:

InstructionPointer=Stack.pop()

#### 説明:

RETは、以前に呼び出されたサブルーチンから戻ります。この命令を処理する場合、現在の命令ポインタはスタック内の最上位の値で上書きされます。その後、実行は以前に保存されたアドレスで続行されます。

### 05命令JT(真の場合はジャンプ)

#### 構造およびバリアント:

(バリアント1):JT{レジスタ1}、{ImmediateValue}

(バリアント2):JT{レジスタ1}、{レジスタ2}

#### 効果:

(バリアント1):もしレジスタ1!=0次にInstructionPointer=ImmediateValue

(バリアント2):もしレジスタ1!=0次にInstructionPointer=レジスタ2

#### 説明:

JTがジャンプを実行するのは、最初のオペランドがtrueの場合、つまり、整数として取得されたときにゼロ以外の場合だけです。その場合、その動作は無条件ジャンプと同じです。それ以外の場合は効果がありません。

### 06命令JF(偽の場合はジャンプ)

#### 構造およびバリアント:

(バリアント1):JF{レジスタ1},{即値}

(バリアント2):JF{レジスタ1}、{レジスタ2}

#### 処理アクション:

(バリアント1):もしレジスタ1==0次にInstructionPointer=ImmediateValue

(バリアント2):もしレジスタ1==0次にInstructionPointer=レジスタ2

#### 説明:

JFは、最初のオペランドがfalseの場合にのみジャンプを実行します。つまり、整数として取得された場合は0です。その場合、その動作は無条件ジャンプと同じです。それ以外の場合は効果がありません。

### 07命令IEQ(Integer Equal)

#### 構造およびバリアント:

(バリアント1):IEQ{レジスタ1},{即値}

(バリアント2):IEQ{レジスタ1}、{レジスタ2}

#### 処理アクション:

(バリアント1):もしレジスタ1==即値次にレジスタ1=1それ以外レジスタ1=0

(バリアント2):もしレジスタ1==レジスタ2次にレジスタ1=1それ以外レジスタ1=0

#### 説明:

IEQは、整数として解釈される2つのオペランドを取り、それらが等しいかどうかをチェックします。ブール値の結果は、常にレジスタである最初のオペランドに格納されます。

### 08命令INE(Integer Not Equal)

#### 構造およびバリアント:

(バリアント1):INE{レジスタ1},{即値}

(バリアント2):INE{レジスタ1}、{レジスタ2}

#### 処理アクション:

(バリアント1):もしレジスタ1!=ImmediateValue次にレジスタ1=1それ以外レジスタ1=0

(バリアント2):もしレジスタ1!=レジスタ2次にレジスタ1=1それ以外レジスタ1=0

#### 説明:

INEは、2つのオペランドを整数として解釈し、それらが異なるかどうかをチェックします。ブール値の結果は、常にレジスタである最初のオペランドに格納されます。

### 09命令IGT(より大きい整数)

#### 構造およびバリアント:

(バリアント1):IGT{レジスタ1},{即値}

(バリアント2):IGT{レジスタ1}、{レジスタ2}

#### 処理アクション:

(バリアント1):もしレジスタ1>即値次にレジスタ1=1それ以外レジスタ1=0

(バリアント2):もしレジスタ1>レジスタ2次にレジスタ1=1それ以外レジスタ1=0

#### 説明:

IGTは、整数として解釈される2つのオペランドを取り、最初のオペランドが2番目のオペランドより大きいかどうかをチェックします。ブール値の結果は、常にレジスタである最初のオペランドに格納されます。

### 10命令IGE(以上の整数)

#### 構造およびバリアント:

(バリアント1):IGE{レジスタ1}、{ImmediateValue}

(バリアント2):IGE{レジスタ1}、{レジスタ2}

#### 処理アクション:

(バリアント1):もしレジスタ1>=ImmediateValue次にレジスタ1=1それ以外レジスタ1=0

(バリアント2):もしレジスタ1>=レジスタ2次にレジスタ1=1それ以外レジスタ1=0

#### 説明:

IGEは整数として解釈される2つのオペランドを取り、最初のオペランドが2番目のオペランドより大きいか等しいかどうかをチェックします。ブール値の結果は、常にレジスタである最初のオペランドに格納されます。

### 11命令ILT(より小さい整数)

#### 構造およびバリアント:

(バリアント1):ILT{レジスタ1}、{ImmediateValue}

(バリアント2):ILT{レジスタ1}、{レジスタ2}

#### 処理アクション:

(バリアント1):もしレジスタ1<即値次にレジスタ1=1それ以外レジスタ1=0

(バリアント2):もしレジスタ1<レジスタ2次にレジスタ1=1それ以外レジスタ1=0

#### 説明:

ILTは、整数として解釈される2つのオペランドを取り、最初のオペランドが2番目のオペランドより小さいかどうかをチェックします。ブール値の結果は、常にレジスタである最初のオペランドに格納されます。

### 12命令ILE(整数以下)

#### 構造およびバリアント:

(バリアント1):ILE{レジスタ1},{即値}

(バリアント2):ILE{レジスタ1}、{レジスタ2}

#### 処理アクション:

(バリアント1):もしレジスタ1<=即値次にレジスタ1=1それ以外レジスタ1=0

(バリアント2):もしレジスタ1<=レジスタ2次にレジスタ1=1それ以外レジスタ1=0

#### 説明:

ILEは、整数として解釈される2つのオペランドを取り、最初のオペランドが2番目のオペランド以下であるかどうかをチェックします。ブール値の結果は、常にレジスタである最初のオペランドに格納されます。

### 13命令FEQ(Float Equal)

#### 構造およびバリアント:

(バリアント1):FEQ{レジスタ1}、{ImmediateValue}

(バリアント2):FEQ{レジスタ1}、{レジスタ2}

#### 処理アクション:

(バリアント1):もしレジスタ1==即値次にレジスタ1=1それ以外レジスタ1=0

(バリアント2):もしレジスタ1==レジスタ2次にレジスタ1=1それ以外レジスタ1=0

#### 説明:

FEQは、2つのオペランドを浮動小数点として解釈し、それらが等しいかどうかをチェックします。ブール値の結果は、常にレジスタである最初のオペランドに格納されます。

### 14命令FNE(Float Not Equal)

#### 構造およびバリアント:

(バリアント1):FNE{レジスタ1}、{ImmediateValue}

(バリアント2):FNE{レジスタ1}、{レジスタ2}

#### 処理アクション:

(バリアント1):もしレジスタ1!=ImmediateValue次にレジスタ1=1それ以外レジスタ1=0

(バリアント2):もしレジスタ1!=レジスタ2次にレジスタ1=1それ以外レジスタ1=0

#### 説明:

FNEは、2つのオペランドを浮動小数点として解釈し、それらが異なるかどうかをチェックします。ブール値の結果は、常にレジスタである最初のオペランドに格納されます。

### 15命令FGT(Float Greater Than)

#### 構造およびバリアント:

(バリアント1):FGT{レジスタ1}、{ImmediateValue}

(バリアント2):FGT{レジスタ1}、{レジスタ2}

#### 処理アクション:

(バリアント1):もしレジスタ1>即値次にレジスタ1=1それ以外レジスタ1=0

(バリアント2):もしレジスタ1>レジスタ2次にレジスタ1=1それ以外レジスタ1=0

#### 説明:

FGTは、2つのオペランドを浮動小数点として解釈し、最初のオペランドが2番目のオペランドよりも大きいかどうかをチェックします。ブール値の結果は、常にレジスタである最初のオペランドに格納されます。

### 16命令FGE(浮動小数点以上)

#### 構造およびバリアント:

(バリアント1):FGE{レジスタ1}、{ImmediateValue}

(バリアント2):FGE{レジスタ1}、{レジスタ2}

#### 処理アクション:

(バリアント1):もしレジスタ1>=ImmediateValue次にレジスタ1=1それ以外レジスタ1=0

(バリアント2):もしレジスタ1>=レジスタ2次にレジスタ1=1それ以外レジスタ1=0

#### 説明:

FGEは、整数として解釈される2つのオペランドを取り、最初のオペランドが2番目のオペランドより大きいか等しいかどうかをチェックします。ブール値の結果は、常にレジスタである最初のオペランドに格納されます。

### 17命令FLT(Float Less Than)

#### 構造およびバリアント:

(バリアント1):FLT{レジスタ1}、{即値}

(バリアント2):FLT{レジスタ1}、{レジスタ2}

#### 処理アクション:

(バリアント1):もしレジスタ1<即値次にレジスタ1=1それ以外レジスタ1=0

(バリアント2):もしレジスタ1<レジスタ2次にレジスタ1=1それ以外レジスタ1=0

#### 説明:

FLTは、浮動小数点として解釈される2つのオペランドを取り、最初のオペランドが2番目のオペランドより小さいかどうかをチェックします。ブール値の結果は、常にレジスタである最初のオペランドに格納されます。

### 18命令FLE(Float Less or Equal)

#### 構造およびバリアント:

(バリアント1):FLE{レジスタ1},{即値}

(バリアント2):FLE{レジスタ1}、{レジスタ2}

#### 処理アクション:

(バリアント1):もしレジスタ1<=即値次にレジスタ1=1それ以外レジスタ1=0

(バリアント2):もしレジスタ1<=レジスタ2次にレジスタ1=1それ以外レジスタ1=0

#### 説明:

FLEは、2つのオペランドを浮動小数点として解釈し、最初のオペランドが2番目のオペランドより小さいか等しいかどうかをチェックします。ブール値の結果は、常にレジスタである最初のオペランドに格納されます。

### 19命令MOV(移動)

#### 構造およびバリアント:

(バリアント1):MOV{レジスタ1},{即時値}

(バリアント2):MOV{レジスタ1}、{レジスタ2}

(バリアント3):MOV{レジスタ1},[{ImmediateValue}]

(バリアント4):MOV{レジスタ1},[{レジスタ2}]

(バリアント5):MOV{レジスタ1},[{レジスタ2}+{ImmediateValue}]

(バリアント6):MOV[{ImmediateValue}],{レジスタ2}

(バリアント7):MOV[{レジスタ1}],{レジスタ2}

(バリアント8):MOV[{レジスタ1}+{即値}],{レジスタ2}

#### 処理アクション:

(バリアント1):レジスタ1=即値

(バリアント2):レジスタ1=レジスタ2

(バリアント3):レジスタ1=メモリ[ImmediateValue]

(バリアント4):レジスタ1=メモリ[レジスタ2]

(バリアント5):レジスタ1=メモリ[レジスタ2+ImmediateValue]

(バリアント6):メモリ[ImmediateValue]=レジスタ2

(バリアント7):メモリ[レジスタ1]=レジスタ2

(バリアント8):メモリ[レジスタ1+即値]=レジスタ2

#### 説明:

MOVは、2番目のオペランドに示された値を、最初のオペランドに示されたレジスタまたはメモリアドレスにコピーします。MOVは、8つの異なるアドレッシングモードを区別する必要があるため、処理が最も複雑な命令です。

この命令は、「アドレッシングモード」フィールドで使用する8つのモードを指定します。可能な値は次のように解釈されます。

|  |  |  |
| --- | --- | --- |
| MOVアドレッシングモード | | |
| 値 | 宛先 | [ソース] |
| 0 | レジスタ1 | 即時価値 |
| 1個 | レジスタ1 | レジスタ2 |
| 2個 | レジスタ1 | メモリ[即値] |
| 3個 | レジスタ1 | メモリ[レジスタ2] |
| 4個 | レジスタ1 | メモリ[レジスタ2+即値] |
| 5個 | メモリ[即値] | レジスタ2 |
| 6個 | メモリ[レジスタ1] | レジスタ2 |
| 7個 | メモリ[レジスタ1+即値] | レジスタ2 |

### 20命令LEA(ロード有効アドレス)

#### 構造およびバリアント:

(バリアント1):LEA{レジスタ1},[{レジスタ2}]

(バリアント2):LEA{レジスタ1},[{レジスタ2}+{ImmediateValue}]

#### 処理アクション:

(バリアント1):レジスタ1=レジスタ2

(バリアント2):レジスタ1=レジスタ2+即値

#### 説明:

LEAは2番目のオペランドとしてメモリアドレスを取ります。そのアドレス(内容ではない)を、最初のオペランドとして指定されたレジスタに格納します。最も有用なケースは、アドレスがポインタ+オフセットの形式で指定された場合です。これは、加算が自動的に実行されるためです。

### 21命令PUSH

#### 構造およびバリアント:

PUSH{レジスタ1}

#### 処理アクション:

Stack.Push(レジスタ1)

#### 説明:

PUSHは、CPUハードウェアスタックを使用して、スタックの最上部にある指定されたレジスタに含まれる値を追加します。

### 22命令POP

#### 構造およびバリアント:

POP{レジスタ1}

#### 処理アクション:

レジスタ1=Stack.Pop()

#### 説明:

POPは、CPUハードウェアスタックを使用して、スタックの最上位から値を削除し、指定されたレジスタに書き込みます。

### 23命令IN

#### 構造およびバリアント:

IN{レジスタ1}、{ポート番号}

#### 処理アクション:

レジスタ1=ポート[PortNumber]

#### 説明:

INは、制御バスを使用して別のチップのI/Oポートから読み取り、戻り値を指定されたレジスタに格納します。この読み取り要求は、指定されたポートによっては副作用を引き起こす可能性があります。

### 24命令OUT

#### 構造およびバリアント:

(バリアント1):OUT{PortNumber},[{ImmediateValue}]の値

(バリアント2):OUT{ポート番号}、{レジスタ1}

#### 処理アクション:

(バリアント1):ポート[PortNumber]=ImmediateValue

(バリアント2):ポート[PortNumber]=レジスタ1

#### 説明:

OUTはコントロールバスを使用して、指定された値を別のチップのI/Oポートに書き込みます。この書き込み要求は、指定されたポートによっては副作用を引き起こす可能性があります。

### 25命令MOVS(移動文字列)

#### 構造およびバリアント:

モブ

#### 処理アクション:

メモリ[DR]=メモリ[SR]

DR+=1の場合

SR+=1の場合

CR剤-=1の場合

もしCR>0の場合次にInstructionPointer-=1の場合

#### 説明:

MOVSは、SRで指定されたメモリアドレスからDRで指定されたメモリアドレスに値をコピーし(想定されるMOV[DR],[SR]のように)、CRのカウンタが0になるまでローカルループを繰り返し、連続するアドレスを処理します。

CRの値が0以下で呼び出された場合でも、MOVSは常に、記述されたループを少なくとも1回実行することに注意してください。

この命令は、レジスタを経由せずにメモリ内の2つの場所から値を直接コピーするVircon32 CPUの唯一の方法です。

### 26命令SETS(Set String)

#### 構造およびバリアント:

設定

#### 処理アクション:

メモリ[DR]=SR

DR+=1の場合

CR剤-=1の場合

もしCR>0の場合次にInstructionPointer-=1の場合

#### 説明:

SETは、SRの値をDRが指すアドレスにコピーします(MOV[DR]、SRの場合と同様)。次に、CRのカウンタが0に達するまでローカルループを繰り返し、連続するアドレスに書き込みます。

CRの値が0以下で呼び出された場合でも、SETSは常に記述されたループを少なくとも1回は実行することに注意してください。

### 27命令CMPS(比較文字列)

#### 構造およびバリアント:

CMPS{レジスタ1}

#### 処理アクション:

レジスタ1=メモリ[DR]-メモリ[SR]

もしレジスタ1!=0次に末端処理

DR+=1の場合

SR+=1の場合

CR剤-=1の場合

もしCR>0の場合次にInstructionPointer-=1の場合

#### 説明:

CMPSは、DRで指定されたアドレスの値とSRで指定されたアドレスの値を減算して比較し、CRのカウンタが0になるまでローカルループを繰り返し、連続するアドレスを読み込みます。

比較結果は指定されたレジスタに格納され、等しい場合は0、[DR]の値がより大きい場合は正、[SR]の値がより大きい場合は負になります。

0以下のCRの値で呼び出された場合でも、CMPSは常に、記述されたループを少なくとも1回実行することに注意してください。

### 28命令CIF(Convert Integer to Float)

#### 構造およびバリアント:

CIF{レジスタ1}

#### 処理アクション:

レジスタ1=(浮動小数点)レジスタ1

#### 説明:

CIFは、指定されたレジスタaを整数値として解釈します。次に、その値を浮動小数点表現に変換し、その結果を同じレジスタに格納します。

浮動小数点表現の精度が制限されているため、32ビット整数の値が十分に高いと、浮動小数点として表現されたときに精度が失われることに注意してください。

### 29命令CFI(Convert Float to Integer)

#### 構造およびバリアント:

CFI{レジスタ1}

#### 処理アクション:

レジスタ1=(整数)レジスタ1

#### 説明:

CFIは、指定されたレジスタaを浮動小数点値として解釈します。次に、その値を整数表現に変換し、その結果を同じレジスタに格納します。変換は四捨五入ではなく、切り捨てによって行われます(小数点以下は破棄されます)。

float表現の範囲が非常に広いため、floatの値が十分に大きいと、32ビット整数として表現されたときに精度が失われることに注意してください。

### 30命令CIB(Convert Integer to Boolean)

#### 構造およびバリアント:

CIB{レジスタ1}

#### 処理アクション:

もしレジスタ1!=0次にレジスタ1=1

#### 説明:

CIBは、指定されたレジスタaを整数値として解釈します。次に、その値を標準のブール表現に変換し、結果を同じレジスタに格納します。これは、0以外のすべての値が1に変換されることを意味します。

### 31命令CFB(Convert Float to Boolean)

#### 構造およびバリアント:

CFB{レジスタ1}

#### 処理アクション:

もしレジスタ1!=0.0次にレジスタ1=1それ以外レジスタ1=0

#### 説明:

CFBは、指定されたレジスタaを浮動小数点値として解釈します。次に、その値を0(浮動小数点値0.0の場合)または1(その他の値の場合)に変換し、そのレジスタに格納します。

### 32命令NOT

#### 構造およびバリアント:

NOT{レジスタ1}

#### 処理アクション:

レジスタ1=レジスタ1ではない

#### 説明:

NOTは、指定されたレジスタ内のすべてのビットを反転することによって、バイナリ「not」を実行します。

### 33命令AND

#### 構造およびバリアント:

(バリアント1):AND{レジスタ1},{イミディエート値}

(バリアント2):AND{レジスタ1}、{レジスタ2}

#### 処理アクション:

(バリアント1):レジスタ1=レジスタ1 AND ImmediateValue

(バリアント2):レジスタ1=レジスタ1 ANDレジスタ2

#### 説明:

ANDは、指定された2つのオペランド内の対応するビットの各ペア間でバイナリの「and」を実行します。結果は、常にレジスタである最初のビットに格納されます。

### 34命令OR

#### 構造およびバリアント:

(バリアント1):OR{レジスタ1},{イミディエート値}

(バリアント2):OR{レジスタ1}、{レジスタ2}

#### 処理アクション:

(バリアント1):レジスタ1=レジスタ1 OR即値

(バリアント2):レジスタ1=レジスタ1 ORレジスタ2

#### 説明:

ORは、指定された2つのオペランド内の対応するビットの各ペア間でバイナリの「or」を実行します。結果は、常にレジスタである最初のビットに格納されます。

### 35命令XOR

#### 構造およびバリアント:

(バリアント1):XOR{レジスタ1},{即値}

(バリアント2):XOR{レジスタ1}、{レジスタ2}

#### 処理アクション:

(バリアント1):レジスタ1=レジスタ1 XOR即値

(バリアント2):レジスタ1=レジスタ1 XORレジスタ2

#### 説明:

XORは、指定された2つのオペランドの対応するビットの各ペア間で、バイナリの「排他的OR」を実行します。結果は、常にレジスタである最初のビットに格納されます。

### 36命令BNOT(ブールNOT)

#### 構造およびバリアント:

BNOT{レジスタ1}

#### 処理アクション:

もしレジスタ1==0次にレジスタ1=1それ以外レジスタ1=0

#### 説明:

BNOTは、指定されたレジスタをブール値として解釈し、それを逆のブール値に変換します。これは、最初にCIBを使用し、次にビット番号0を反転することと同じです。

### 37命令SHL(左シフト)

#### 構造およびバリアント:

(バリアント1):SHL{レジスタ1}、{ImmediateValue}

(バリアント2):SHL{レジスタ1}、{レジスタ2}

#### 処理アクション:

(バリアント1):レジスタ1=レジスタ1<<ImmediateValue

(バリアント2):レジスタ1=レジスタ1<<レジスタ2

#### 説明:

SHLは、指定されたレジスタでビットの左方移動を実行します。2番目のオペランドは、シフトする位置の整数として取得されます。0の位置をシフトしても効果はありませんが、負の値を指定すると右にシフトします。シフトのタイプは論理的です。左にシフトする場合、オーバーフローは破棄され、ゼロが最下位ビットとして導入されます。右にシフトする場合、アンダーフローは破棄され、ゼロが最上位ビットとして導入されます。

### 38命令IADD(Integer Add)

#### 構造およびバリアント:

(バリアント1):IADD{レジスタ1},{ImmediateValue}

(バリアント2):IADD{レジスタ1}、{レジスタ2}

#### 処理アクション:

(バリアント1):レジスタ1+=即値

(バリアント2):レジスタ1+=レジスタ2

#### 説明:

IADDは両方のオペランドを整数として解釈し、加算を実行します。結果は最初のオペランドに格納されます。このオペランドは常にレジスタです。オーバーフロービットは破棄されます。

### 39命令ISUB(整数減算)

#### 構造およびバリアント:

(バリアント1):ISUB{レジスタ1},{ImmediateValue}

(バリアント2):ISUB{レジスタ1}、{レジスタ2}

#### 処理アクション:

(バリアント1):レジスタ1-=ImmediateValue(即時値)

(バリアント2):レジスタ1-=レジスタ2

#### 説明:

ISUBは両方のオペランドを整数として解釈し、減算を実行します。結果は最初のオペランドに格納されます。このオペランドは常にレジスタです。オーバーフロービットは破棄されます。

### 40命令IMUL(Integer Multiply)

#### 構造およびバリアント:

(バリアント1):IMUL{レジスタ1}、{ImmediateValue}

(バリアント2):IMUL{レジスタ1}、{レジスタ2}

#### 処理アクション:

(バリアント1):レジスタ1\*=即値

(バリアント2):レジスタ1\*=レジスタ2

#### 説明:

IMULは、両方のオペランドを整数として解釈し、乗算を実行します。結果は、常にレジスタである最初のオペランドに格納されます。オーバーフロービットは破棄されます。

### 41命令IDIV(整数除算)

#### 構造およびバリアント:

(バリアント1):IDIV{レジスタ1}、{ImmediateValue}

(バリアント2):IDIV{レジスタ1}、{レジスタ2}

#### 処理アクション:

(バリアント1):レジスタ1/=即値

(バリアント2):レジスタ1/=レジスタ2

#### 説明:

IDIVは、両方のオペランドを整数として解釈し、除算を実行します。結果は、常にレジスタである最初のオペランドに格納されます。

### 42命令IMOD(整数係数)

#### 構造およびバリアント:

(バリアント1):IMOD{レジスタ1}、{ImmediateValue}

(バリアント2):IMOD{レジスタ1}、{レジスタ2}

#### 処理アクション:

(バリアント1):レジスタ1=レジスタ1 mod ImmediateValue

(バリアント2):レジスタ1=レジスタ1 modレジスタ2

#### 説明:

IMODは、両方のオペランドを整数として解釈し、除算を実行します。その除算の残りは、常にレジスタである最初のオペランドに格納されます。

### 43命令ISGN(整数符号変更)

#### 構造およびバリアント:

ISGN{レジスタ1}

#### 処理アクション:

レジスタ1=-レジスタ1

#### 説明:

ISGNはオペランドレジスタを整数として解釈し、その符号を反転させる。

### 44命令IMIN(Integer Minimum)

#### 構造およびバリアント:

(バリアント1):IMIN{レジスタ1}、{ImmediateValue}

(バリアント2):IMIN{レジスタ1}、{レジスタ2}

#### 処理アクション:

(バリアント1):レジスタ1=min(レジスタ1,ImmediateValue)

(バリアント2):レジスタ1=min(レジスタ1、レジスタ2)

#### 説明:

IMINは、両方のオペランドを整数として解釈します。次に、両方の値の最小値を取得し、それを最初のオペランドに格納します。このオペランドは常にレジスタです。

### 45命令IMAX(Integer Maximum)

#### 構造およびバリアント:

(バリアント1):IMAX{レジスタ1},{即値}

(バリアント2):IMAX{レジスタ1}、{レジスタ2}

#### 処理アクション:

(バリアント1):レジスタ1=max(レジスタ1,ImmediateValue)

(バリアント2):レジスタ1=max(レジスタ1、レジスタ2)

#### 説明:

IMAXは、両方のオペランドを整数として解釈します。次に、両方の値の最大値を取得し、それを最初のオペランドに格納します。このオペランドは常にレジスタです。

### 46命令IABS(Integer Sign change)

#### 構造およびバリアント:

IABS{レジスタ1}

#### 処理アクション:

レジスタ1=abs(レジスタ1)

#### 説明:

IABSはオペランド・レジスタを整数として解釈し、その絶対値を取る。

### 47命令FADD(Float Add)

#### 構造およびバリアント:

(バリアント1):FADD{レジスタ1},{ImmediateValue}

(バリアント2):FADD{レジスタ1}、{レジスタ2}

#### 処理アクション:

(バリアント1):レジスタ1+=即値

(バリアント2):レジスタ1+=レジスタ2

#### 説明:

FADDは、両方のオペランドを浮動小数点として解釈し、加算を実行します。結果は、常にレジスタである最初のオペランドに格納されます。

### 48命令FSUB(Float Subtract)

#### 構造およびバリアント:

(バリアント1):FSUB{レジスタ1},{ImmediateValue}

(バリアント2):FSUB{レジスタ1}、{レジスタ2}

#### 処理アクション:

(バリアント1):レジスタ1-=ImmediateValue(即時値)

(バリアント2):レジスタ1-=レジスタ2

#### 説明:

FSUBは両方のオペランドを浮動小数点として解釈し、減算を実行します。結果は最初のオペランドに格納されます。このオペランドは常にレジスタです。

### 49命令FMUL(浮動小数点乗算)

#### 構造およびバリアント:

(バリアント1):FMUL{レジスタ1},{即値}

(バリアント2):FMUL{レジスタ1}、{レジスタ2}

#### 処理アクション:

(バリアント1):レジスタ1\*=即値

(バリアント2):レジスタ1\*=レジスタ2

#### 説明:

FMULは両方のオペランドを浮動小数点として解釈し、乗算を実行します。結果は最初のオペランドに格納されます。このオペランドは常にレジスタです。

### 50命令FDIV(浮動小数点除算)

#### 構造およびバリアント:

(バリアント1):FDIV{レジスタ1},{ImmediateValue}

(バリアント2):FDIV{レジスタ1}、{レジスタ2}

#### 処理アクション:

(バリアント1):レジスタ1/=即値

(バリアント2):レジスタ1/=レジスタ2

#### 説明:

FDIVは両方のオペランドを浮動小数点として解釈し、除算を実行します。結果は最初のオペランドに格納されます。このオペランドは常にレジスタです。

### 51命令FMOD(浮動小数点係数)

#### 構造およびバリアント:

(バリアント1):FMOD{レジスタ1},{ImmediateValue}

(バリアント2):FMOD{レジスタ1}、{レジスタ2}

#### 処理アクション:

(バリアント1):レジスタ1=fmod(レジスタ1,ImmediateValue)

(バリアント2):レジスタ1=fmod(レジスタ1、レジスタ2)

#### 説明:

FMODは、両方のオペランドを浮動小数点として解釈し、除算を実行します。次に、結果の小数部分が破棄されるときにその除算の剰余を取得し、それを最初のオペランドに格納します。このオペランドは常にレジスタです。

### 52指示FSGN(フロートサイン変更)

#### 構造およびバリアント:

FSGN{レジスタ1}

#### 処理アクション:

レジスタ1=-レジスタ1

#### 説明:

FSGNはオペランドレジスタを浮動小数点として解釈し、その符号を反転させる。

### 53命令FMIN(Float Minimum)

#### 構造およびバリアント:

(バリアント1):FMIN{レジスタ1},{ImmediateValue}

(バリアント2):FMIN{レジスタ1}、{レジスタ2}

#### 処理アクション:

(バリアント1):レジスタ1=min(レジスタ1,ImmediateValue)

(バリアント2):レジスタ1=min(レジスタ1、レジスタ2)

#### 説明:

FMINは、両方のオペランドを浮動小数点として解釈します。次に、両方の値の最小値を取得し、それを最初のオペランドに格納します。これは常にレジスタです。

### 54命令FMAX(Float Maximum)

#### 構造およびバリアント:

(バリアント1):FMAX{レジスタ1},{ImmediateValue}

(バリアント2):FMAX{レジスタ1}、{レジスタ2}

#### 処理アクション:

(バリアント1):レジスタ1=max(レジスタ1,ImmediateValue)

(バリアント2):レジスタ1=max(レジスタ1、レジスタ2)

#### 説明:

FMAXは、両方のオペランドを浮動小数点として解釈します。次に、両方の値の最大値を取得し、それを最初のオペランドに格納します。このオペランドは常にレジスタです。

### 55命令FABS(Float Sign change)

#### 構造およびバリアント:

FABS{レジスタ1}

#### 処理アクション:

レジスタ1=abs(レジスタ1)

#### 説明:

FABSはオペランドレジスタを浮動小数点として解釈し、その絶対値を取る。

### 56インストラクションFLR(フロア)

#### 構造およびバリアント:

FLR{レジスタ1}

#### 処理アクション:

レジスタ1=floor(レジスタ1)

#### 説明:

FLRはオペランド・レジスタを浮動小数点として解釈し、整数値に切り捨てます。結果は整数に変換されませんが、浮動小数点のままであることに注意してください。

### 57命令CEIL(シーリング)

#### 構造およびバリアント:

CEIL{レジスタ1}

#### 処理アクション:

レジスタ1=ceil(レジスタ1)

#### 説明:

CEILはオペランドレジスタを浮動小数点として解釈し、それを整数値に切り上げます。結果は整数に変換されませんが、浮動小数点のままであることに注意してください。

### 58命令ROUND

#### 構造およびバリアント:

ROUND{レジスタ1}

#### 処理アクション:

レジスタ1=round(レジスタ1)

#### 説明:

ROUNDは、オペランドレジスタを浮動小数点として解釈し、最も近い整数値に丸めます。結果は整数に変換されませんが、浮動小数点のままであることに注意してください。

### 59命令SIN(Sine)

#### 構造およびバリアント:

SIN{レジスタ1}

#### 処理アクション:

レジスタ1=sin(レジスタ1)

#### 説明:

SINはオペランドレジスタを浮動小数点として解釈し、その値の正弦を計算します。正弦関数はその引数をラジアンで解釈します。

### 60命令ACOS(アークコサイン)

#### 構造およびバリアント:

ACOS{レジスタ1}

#### 処理アクション:

レジスタ1=acos(レジスタ1)

#### 説明:

ACOSはオペランドレジスタを浮動小数点として解釈し、その値のアークコサインを計算します。結果は[0,pi]の範囲のラジアンで示されます。

### 61命令ATAN2(逆正接2)

#### 構造およびバリアント:

ATAN2{レジスタ1}、{レジスタ2}

#### 処理アクション:

レジスタ1=atan2(レジスタ1、レジスタ2)

#### 説明:

ATAN2は、両方のオペランドレジスタを浮動小数点として解釈し、Vx=レジスタ2およびVy=レジスタ1のようにベクトルの角度を計算します。結果は最初のオペランドレジスタに格納され、[-pi,pi]の範囲のラジアンで示されます。角度の原点は(Vx>0、Vy=0)にあり、(Vx=0、Vy>0)に向かって回転すると角度が大きくなります。

### 62命令LOG(対数)

#### 構造およびバリアント:

LOG{レジスタ1}

#### 処理アクション:

レジスタ1=log(レジスタ1)

#### 説明:

LOGはオペランドレジスタを浮動小数点として解釈し、その値の底eの対数を計算します。

### 63命令POW(パワー)

#### 構造およびバリアント:

パカッ{レジスタ1}、{レジスタ2}

#### 処理アクション:

レジスタ1=pow(レジスタ1,レジスタ2)

#### 説明:

POWは、両方のオペランドレジスタを浮動小数点として解釈し、最初のオペランドを2番目のオペランドの累乗した結果を計算します。結果は最初のオペランドレジスタに格納されます。

## 11ハードウェア・エラーの検出

プログラムの実行では、ゼロによる除算など、ハードウェアが処理できない状況が発生することがあります。Vircon32では、これらの状況は常にCPU操作の結果として発生するため、プロセッサはエラー状況を検出し、対応するハードウェアエラーメカニズムをトリガーする責任を負うコンポーネントである必要があります。

次のサブセクションでは、ハードウェアエラーが発生する可能性のあるすべての状況をリストし、CPUが各タイプのエラーを検出するために使用するさまざまな基準について説明します。

### 11.1バスエラー

これらは、CPUが2つの通信バスのいずれかに要求を行ったときに発生する可能性のあるエラーです。バス自体が成功/失敗インジケータで要求に応答するため、CPUによるこれらの検出は自動的に行われます。

#### 無効なメモリの読み取り

メモリバスへの読み取り要求が失敗しました。

#### 無効なメモリ書き込み

メモリバスへの書き込み要求が失敗しました。

#### 無効なポート読み取り

制御バスへの読み取り要求が失敗しました。

#### 無効なポート書き込み

コントロールバスへの書き込み要求が失敗しました。

### 11.2スタックエラー

通常の動作では、スタックポインタはRAMメモリ内の有効なアドレス位置のみを指す必要があります。その範囲は0x00000000～0x003FFFFFです。プログラムエラーまたは異常に大きなコールスタックは、スタックが正常に機能し続けることができず、CPUが応答としてハードウェアエラーをトリガーする状況を引き起こす可能性があります。次の2つのエラー状況が考えられます:

#### スタックオーバーフロー

プッシュ操作の後、SPが負になった場合は、スタックオーバーフローが発生したと見なされます。この時点で、スタックはRAMメモリの最後まで成長しており、成長を続けることはできません。

#### スタックアンダーフロー

ポップ操作の後、SPが初期値の0x003FFFFF(RAMメモリ内の最後のワード)よりも大きくなった場合は、スタックアンダーフローが発生したと考えられます。この時点では、スタックは枯渇しているため、減少を続けることはできません。

### 11.3数学的エラー

数学演算を実行するCPU命令の中には、受け取った引数が関数ドメインの外にある場合にエラーが発生する一般的な原因があるものがあります。このような状況が検出されると、命令処理が中止され、対応するエラーがトリガーされます。数学関数が定義されていない場合は、次のようになります。

#### 除算エラー

このエラーは、4つの異なる命令で発生する可能性があります。除算とモジュラスで、整数と浮動小数点の両方のバージョンです。2番目の引数(除数)が0の場合に発生します。

#### アークコサインエラー

この命令は、引数が-1.0と+1.0の間にない場合(両端が有効範囲に含まれている場合)は処理できません。

#### 逆正接2エラー

受け取った引数が両方ともゼロの場合、この命令は処理できません。

#### 対数誤差

受信した引数がゼロ以下の場合、この命令は処理できません。

#### 電源エラー

この命令は、最初の引数(base)が負で、2番目の引数(exponent)が整数値でない場合には処理できません。

## 12ハードウェアエラーの処理

CPUが前のセクションで説明した状況の1つを検出した場合、対応するハードウェアエラー処理をトリガする必要があります。エラー処理の最後の段階はBIOSエラーハンドラルーチンによって行われますが、そのルーチンが呼び出される前に、CPUはそのための情報を準備する必要があります。

BIOSがエラーを処理するために必要な準備は、エラーコードを使用してエラーのタイプを識別し、内部レジスタを保存することです。

### 12.1エラー・コード

ハードウェア・エラーが発生すると、検出されたエラー状況のタイプが数値コードで表されます。次の表に、考えられるエラー・コードを示します:

|  |  |
| --- | --- |
| ハードウェアエラーコード | |
| コード | エラーの種類 |
| 0 | 無効なメモリの読み取り |
| 1個 | 無効なメモリ書き込み |
| 2個 | 無効なポート読み取り |
| 3個 | 無効なポート書き込み |
| 4個 | スタックオーバーフロー |
| 5個 | スタックアンダーフロー |
| 6個 | 除算エラー |
| 7個 | アークコサインエラー |
| 8個 | 逆正接2エラー |
| 9個 | 対数誤差 |
| 10個 | 電源エラー |

### 12.2エラーに対するCPUの応答

CPUがハードウェアエラーをトリガーすると、次の処理が実行されます。

* R0にエラーコードを書き込んでください。
* 命令ポインタをR1にコピーします。
* 命令レジスタをR2にコピーします。
* [Immediate Value]をR3にコピーします。
* BPとSPをRAMの最後の位置(アドレス0x003FFFFF)に設定します。
* Instruction Pointerを0x10000000(BIOSエラーハンドラアドレス)に設定します。
* そのアドレスから通常の実行を再開します。

その後、CPUは、エラー情報を画面に表示して実行を停止するタスクを最終的に完了するまで、BIOSエラーハンドラの実行を継続します。

スタックがリセットされ、以前のコールコンテキストが破棄されることに注意してください。これは、エラーがスタック自体によって引き起こされた可能性があるために必要です。また、BIOSが制御を取得した後は、以前のコンテキストはいずれにしても不要になります。

(パート3の終わり)