Brownie STD 32

Reference Manual

---------- ***Ver. 1.1***

Change Log:

2008/07/22 v.1.1 : Released

2008/04/01 v.1.0 : Released

〒541-0053　大阪市中央区本町2-3-8 三甲大阪本町ビル6階
エイシップ・ソリューションズ株式会社
info@asip-solutions.com

Release Notes

v.1.0:

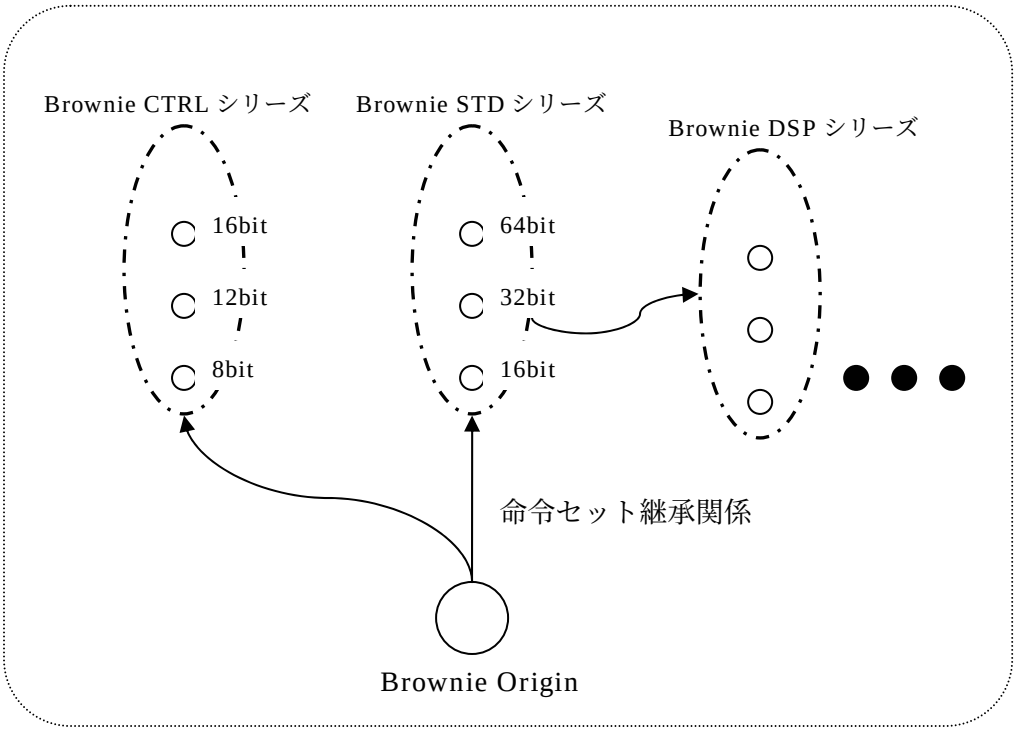
・リリース

Introduction

ASIP（Application Specific Instruction-set Processor）は各種用途向けに特化されたアーキテクチャを持つプロセッサであり，低面積・低消費電力・高性能なプロセッサを実現できるとあって組み込みシステム向けのプロセッサとして注目を集めている．ASIP を用いることで効率の高い組み込みシステムを設計できるが，一方で ASIP の設計自体に多大な設計期間がかかるという問題もはらんでいる．この ASIP の多大な設計期間のうち，大半が基本的なプロセッサの仕様策定であったり，またその実装であったりと，設計者が本来力を注ぐべき特化命令の設計とは関係のない部分で労力を浪費していると言える．ASIP 設計においては，設計者が目的の特化命令の設計に注力できることが理想であり，そのためにはベースとなるプロセッサに特化命令を追加する形で開発するのが望ましい．このようなベース・プロセッサ拡張による ASIP 開発は効率的な設計方法として一般的に良く用いられているが，いくつかの問題点も指摘されている．

一般的なベース・プロセッサ拡張による ASIP 設計の問題点のひとつとして，ベース・プロセッサの選択肢がそれほど多くないことがあげられる．目標に出来るだけ近い仕様が備わっているベース・プロセッサを利用することが，設計者の余分な負担の削減や，全体の開発期間短縮，ASIP の性能向上のために重要である．しかし一般的なベース・プロセッサを用いた設計では，ベース・プロセッサの仕様が目標仕様から離れていることが多く，機能過剰や機能不足などから結局余分な労力を浪費してしまいがちであった．ASIP 設計においては，多数の目的・要求に応じた多数のアーキテクチャ候補が存在し，設計者がそこから自由にベース・プロセッサを選択できることが望ましい．

もうひとつのよく知られたベース・プロセッサ拡張による ASIP 開発の問題点として，ベースとするプロセッサのアーキテクチャに強く制約をうけてしまうため，特化命令の柔軟な設計が望めないこともあげられる．設計の柔軟性は効率的な ASIP を設計する際に必要不可欠なものである．ASIP を柔軟に，かつ容易に設計できるツールとして ASIP Meister がよく知られている．ASIP Meister はプロセッサの仕様記述からプロセッサの HDL 記述，およびアセンブラを自動的に生成するツールであり，リソースの追加やアーキテクチャの変更，命令の拡張などを柔軟に行うことが出来る．しかしこのように ASIP Meister を利用した ASIP 開発の場合でも，基本的なプロセッサ仕様の策定といった労力は避けらない．ASIP の柔軟な開発を短期間で行うためには，ASIP Meister で利用できるベース・プロセッサが必要である．



Brownie ファミリ

Brownie はこのような要求をうけて設計されたベース・プロセッサ・ファミリである．Brownie ファミリの概念を上図に示す．Brownie ファミリには目的に応じたアーキテクチャの命令セット・シリーズがあり，制御用途向け命令セットを持った BrownieCTRL シリーズ，一般用途向け命令セットを持った BrownieSTD シリーズ，信号処理用途向けを持った BrownieDSP シリーズなどがある．これらの命令セット・シリーズの中で, さらにデータバス bit 幅が異なるプロセッサが多数存在しており（Std シリーズならば 16, 32, 64bit），設計者はこれらのベース・プロセッサ候補の中から開発のベースに最も適した Brownie を選択できる．また，Brownie の各命令セット・シリーズ間には命令セット互換性がある．たとえば，BrownieDSP シリーズは BrownieSTD シリーズの命令セットの上位互換にあたる．Brownie の特徴のひとつとして， ASIP Meister との高い親和性があげられる．すなわち，設計者は Brownie をベースにして ASIP Meister を用いることで直ちに特化命令の設計を始められ, アーキテクチャやリソースの変更を伴うような複雑な特化命令の設計も容易に行えるようになる．また，Brownie は必要最低限の命令セットを実装しており，ASIP 設計者の命令セット拡張を妨げない.

ベース・プロセッサへの要求はこのようなアーキテクチャのバリエーションやベース・プロセッサのカスタマイズの柔軟性・容易性だけではない，ASIP 設計にまつわるもうひとつの大きな課題として，ソフトウェア開発環境の作成がある．特化命令を追加した ASIP を設計してもコンパイラ等のソフトウェア開発環境がなければ実際のシステムで利用するのは困難である．BrownieSTD シリーズ以上の Brownie には GNU 開発環境（gcc，binutils, gdb, newlib）が付属され，利用可能である． Brownie は単

純な命令セット，シンプルなアーキテクチャを取ることや，Brownie ファミリ内での命令セット互換性を保つことで，コンパイラの拡張のしやすさも重視している．命令セットの単純化はプロセッサの性能向上や，コンパイラの作成上重要である．Brownie をベースに ASIP を開発した場合，Brownie 付属の GNU 環境を拡張する形で容易にソフトウェア開発環境を構築できる．

このように Brownie は ASIP のためのベース・プロセッサとして十分な要求を満たしたベース・プロセッサであり，ASIP Meister と組み合わせて利用することで高い開発効率と設計自由度を実現し，ASIP 設計者を強力に支援できる．

Brownie STD は Brownie ファミリの中でも汎用的に作られた命令セット・シリーズである．整数・論理演算，ロード・ストア，分岐を備え，外部，内部割込みに対応する．一般的な整数演算やペリフェラル操作を行い，システムの中核として利用できるプロセッサである．

1. Architecture

1.1. Architecture Overview

BrownieSTD は RISC 型パイプラインプロセッサ・コアである．基本的なアーキテクチャ・パラメータを以下の表にまとめる．

表．アーキテクチャ・パラメータ

基本アーキテクチャ	RISC
メモリアーキテクチャ	ハーバード
命令語長	32
データ語長	32
アドレス指定	バイトアドレス
レジスタ方式	汎用レジスタ
汎用レジスタ数	32
パイプライン段数	4
遅延分岐スロット	0
浮動小数点ユニット	なし
フォワーディング	あり（完全）

BrownieSTD のメモリ・アーキテクチャはハーバード・アーキテクチャである．命令長は 32bit 固定，データ語長は 32bit である．アドレスの指定はバイト単位で行える．整数汎用レジスタを 32 本搭載する．Brownie は 4 段パイプライン構造を持っており，各パイプラインステージには IF，ID，EXE，WB と名前がつけられている．パイプラインには完全フォワーディングがなされるため，全ての命令の演算結果はただちに使用可能となる．ただし，ロード系命令に関しては遅延ロードを行い，遅延スロットは 1 である．なお，BrownieSTD は遅延分岐スロットを持たない（遅延分岐を行わない）．また，BrownieSTD は浮動小数点演算を持たない．浮動小数点演算命令はカスタム命令として追加可能である．

1.2. Registers

BrownieSTD は整数レジスタを 32 本搭載する．各レジスタは GPR0～GPR31 としてプログラマが使用可能である．これらの 32 本のレジスタはプログラマが自由に使用可能であるが，幾つかのレジスタはハードウェア的に予約されている．ハードウェア的に予約されたレジスタはプロセッサが自動的に値を更新する可能性がある．

Zero Register:

GPR0 は常に値 0 を保持しておくためのレジスタとして予約されており，常に 0 が出力される．なお，このレジスタへの値の書き込みは禁止である．

Status Register:

GPR1 はプロセッサのステータスを反映するためのレジスタ（Status Register）として予約されている．Status Register はフラグ，および割り込みマスクとプロセッサモードを保持する．フラグはフラグ変化を伴う演算が実行されると自動的に更新される．割り込み許可フラグは'1'で許可，'0'で不許可を表す．実行モードフラグは"01"でユーザーモード，"00"でカーネルモードをそれぞれ示す．ALU フラグへの書き込みは禁止である．実行モードフラグ，割り込み許可フラグへの書き込みはカーネルモード時のみ, 許可される. カーネルモード以外のモード時の書き込みは禁止である. なお, Status Register の読み出し・書き込み，およびその更新タイミングに関しては「Appendix B. ステータスレジスタの扱い」を参照.

Status Register の各ビットの意味および次に示す通りである.

Status Register [3:0]:	ALU フラグ（Carry, Zero, Sign, Overflow）
Status Register [8]:	外部割込み許可フラグ
Status Register [9]:	内部割込み許可フラグ
Status Register [15:14]:	実行モードフラグ

Interrupt Return Register:

GPR2 は外部および内部割り込みが発生した命令のアドレスを保持する．このレジスタの値は割り込み発生時に自動的に更新される.

Link Register:

GPR3 は JPL や JPRL 命令（2.2. Instructions 参照）から復帰すべきアドレスを保持する．すなわち，JPL および JPRL 命令の次の命令のアドレスが保存される．この値は JPL や JPRL が実行されると自動的に更新される.

さらに GPR4，GPR5，GPR6 はそれぞれコンパイラ用に Return Register，Frame Pointer，Stack Pointer と予約されているが，ハードウェア的に予約はされていないので，コンパイラを利用しない場合はプログラマが自由に使用できる．以下の表に予約レジスタの一覧を載せる.

表．予約レジスタ

予約レジスタ	用途	予約
GPR0	Zero Register	ハードウェア
GPR1	Status Register	ハードウェア
GPR2	Interrupt Return Register	ハードウェア
GPR3	Link Register	ハードウェア
GPR4	FramePointer	コンパイラ
GPR5	Stack Pointer	コンパイラ
GPR6	Return Value for Integer or Float/Double	コンパイラ
GPR7	Return Value for Double	コンパイラ
GPR8-15	Register Passing	コンパイラ
GPR16-31	Free	なし

1.3. Memory Model

Addressing

BrownieSTD のアドレッシングにはバイトアドレスを用いる.

Alignment

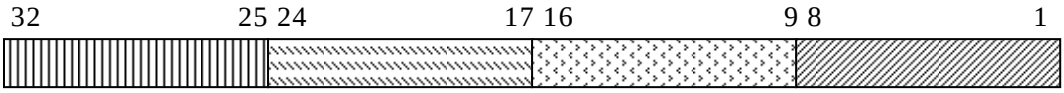
BrownieSTD のアライメントは，バイトアクセスの場合は 1 バイト，ハーフワードアクセスの場合は 2 バイト, ワードアクセスの場合は 4 バイト境界をもつ. BrownieSTD はこれらのアライメントに違反するアクセスを切り捨て補正し，実効アドレスを計算する．例えば，ワードアクセスをする場合，アライメントは

0x0000
0x0004
0x0008
0x000C
:

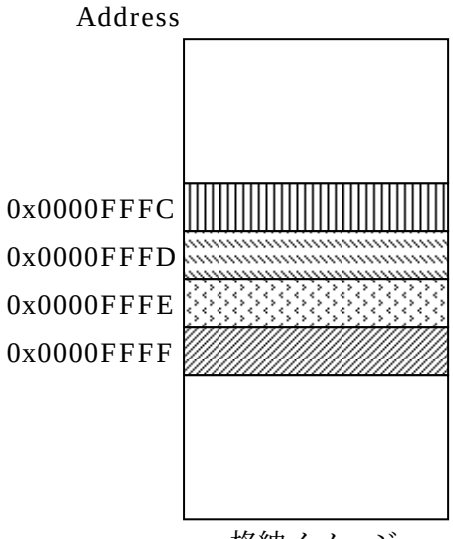
となるが，もしこのとき 0x000A 番地アドレスが指定されたときは，下 2 桁を切り捨て，0x0008 と補正したアドレスを実効アドレスとする.
BrownieSTD は，アライメント違反に対して例外を発生させない.

Endian

BrownieSTD のバイトオーダーはビッグ・エンディアンである.
32bit データを 0x0000FFFC 番地に格納した場合の図を以下に示す.



32bit データ



格納イメージ

Memory Space

BrownieSTD の命令メモリ空間を次の表に示す.

メモリの先頭から 0x0fffffff まではカーネル空間として確保され，ユーザ空間は 0x10000000 から 0xffffffff までである．割り込みベクタはカーネル空間中の 0x0ffe0000 から 0x0fff0000 に配置される.

表 1 命令メモリ空間

アドレス	領域
0x00000000 - 0x0FFFFFFF	カーネル領域
0x0FFE0000 - 0x0FFF0000	割り込みベクタ配置領域
0x10000000 - 0xFFFFFFFF	ユーザ領域

1.4. Instruction Set

Instruction Type

BrownieSTD は以下の 7 タイプの命令を持つ.

RR Type
RI Type
MA Type
BR Type
JP Type
JPR Type
SP Type
RT Type

RR (Register and Register) Type は 2 つのソースレジスタを用いて演算を行い，レジスタに書き戻す命令を示す． RI (Register and Immediate) Type はレジスタと即値を用いて演算を行い，レジスタに書き戻す命令を示す．即値は 16bit で表現される．MA (Memory Access) Type は Load/Store 系の命令タイプである．メモリ・アクセスにはレジスタ間接アドレッシングを用いることができる．BR (BRanch) Type は指定したレジスタの値を評価して分岐する．JP (JumP) Type は無条件即値相対ジャンプを，JPR (JumP Register) Type はレジスタ間接絶対ジャンプの命令をそれぞれ指す．SP (SPecial) Type はソースレジスタや出力レジスタを指定しない命令（NOP など）を示す．RT（Register Transfer） Type は 1 つのソースレジスタを用いて演算を行い，レジスタに書き戻す命令を示す.

Addressing Mode

BrownieSTD の命令セットは次の 2 つのアドレッシングモードをサポートする.

- 即値アドレッシング
- レジスタ間接アドレッシング

レジスタ間接アドレッシングにはオフセットが指定できる．オフセットもバイトアドレスで表される，実効アドレスは「レジスタの値＋オフセット」である．オフセットは符号付の値として評価される.

1.5. Interrupts

BrownieSTD がサポートする割り込みは以下のとおりである.

優先度	タイプ	割り込み名
Highest	RESET	RESET
:	Internal	TRAP
Lowest	External	EXT

RESET 割り込みは外部リセット信号によって起動される割り込みである．RESET 割り込みは最も優先度が高く，マスク不能である．次に優先度が高いのが TRAP 割り込みである．TRAP 割り込みはソフトウェア的に発生させられる．最も優先度が低いのが EXT 割り込みである. EXT 割り込みは外部からの割り込み要求によって発生する．BrownieSTD は外部割込み要求ポートを一つ持ち，プロセッサ・コアの外にある割り込みコントローラから割り込み要因を特定できる．割り込みコントローラはメモリ空間に配置され，データバスからアクセス可能であるとする．内部割込み，および外部割込みは Status Register 内の適当なフラグを設定することによって有効化･無効化できる.

Brownie は割り込みベクタ方式で割り込みを処理する．割り込みベクタは割り込みベクタ配置領域に以下のように配置される.

表 2 割り込みベクタ領域

アドレス	領域
0x0FFE0000 - 0x0FFF000(	割り込みベクタ配置領域
0x0FFE0000 - 0x0FFE03F	リセット割り込みベクタ
0x0FFE0400 - 0x0FFE07F	外部割込みベクタ
0x0FFE0800 - 0x0FFE0BF	内部割込みベクタ

このうち TRAP 例外用の空間は以下のように予約されている． TRAP の詳細は 2.2.2. Instruction Descriptions を参照.

アドレス	領域
0x0FFE0800 - 0x0FFE0900	TRAP

1.6. Interface

BrownieSTD のインターフェースを以下の表に示す.

ポート名	方向	ビット幅	用途
CLK	IN	1	クロック
RESET	IN	1	リセット
DMEM_ADDR_OUT	OUT	32	データメモリアドレスバス
DMEM_ADDRERR_IN	IN	1	データメモリアドレスエラー
DMEM_DATA_OUT	OUT	32	データメモリバス (in方向)
DMEM_DATA_IN	IN	32	データメモリバス (out方向)
DMEM_RW_OUT	OUT	1	データメモリR/Wモード
DMEM_WMODE_OUT	OUT	2	データメモリアクセスモード
DMEM_EMODE_OUT	OUT	1	符号拡張モード出力
DMEM_CANCEL_OUT	OUT	1	データメモリキャンセル
DMEM_REQ_OUT	OUT	1	データメモリReq
DMEM_ACK_IN	IN	1	データメモリAck
IMEM_ADDR_OUT	OUT	32	命令メモリアドレス
IMEM_ADDRERR_IN	IN	1	命令メモリアドレスエラー
IMEM_DATA_IN	IN	32	命令メモリバス
EXTINT_IN	IN	1	外部割込み要求
EXTCATCH_OUT	OUT	1	外部割込み処理開始通知

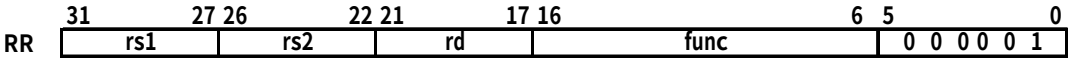
CLK はクロックを，RESET は外部リセット信号を示す．Brownie は命令メモリ，データメモリに対してそれぞれのメモリアクセス・インターフェースを持つ．ただし，BrownieSTD はフェッチが必ず 1 サイクルで完了することを想定しているため，命令メモリインターフェースには Req, Ack 信号が用意されていない．外部からの割込み要求は EXTINT_IN へ入力される．BrownieSTD は要求された割り込み処理を開始した場合はその旨を，EXTCATCH_OUT を通じて外部に通知する． 詳しいインターフェースのプロトコルについては 3. Interrupt と 4. Memory Access を参照.

2. Instruction Set

2.1. Instruction Format

BrownieSTD の各命令タイプのフォーマット，およびその命令タイプに対する拡張可能な命令数を示す.

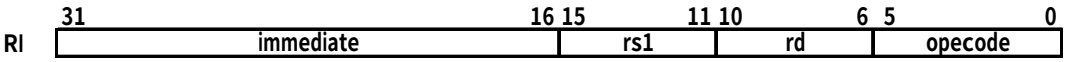
RR Type



rs1 および rs2 はソースレジスタのインデックスを，rd は演算結果を格納するレジスタのインデックスを示す． func は RR Type の種類をさす．下位 6bit は opecode であるが，RR Type の場合は”000001”に固定される.

RR Type として BrownieSTD は 15 個の命令を持つ. したがって RR Type に追加可能な拡張命令数は 2033 個である.

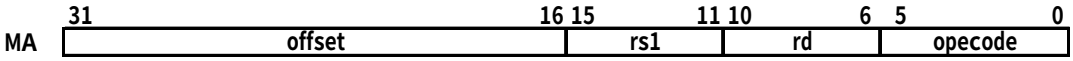
RI Type



rs1 はソースレジスタのインデックスを，rd は演算結果を格納するレジスタのインデックスを示す. Immediate は即値データである. 下位 6bit は opecode であるが, RI Type の場合は”100000”～”111111”まで使用可能である.

RI Type として BrownieSTD は 9 個の命令を持つ．したがって RI Type に追加可能な拡張命令数は 23 個である.

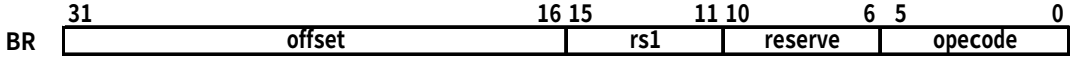
MA Type



rs1，rd はレジスタのインデックスを示す．rs1，rd はロード系，ストア系で示す内容が異なり，ロード系の場合 rs1，rd が，それぞれソースアドレス，保存レジスタのインデックスを，ストア系の場合は rs1，rd は，それぞれ保存先アドレスを格納したレジスタ，ソースレジスタを示す．offset は即値データである．下位 6bit は opecode であるが，MA Type の場合は”000010”～”000111”まで使用している．アドレッシングモードはレジスタ間接アクセスである.

MA Type への追加可能命令数は他 BR，JP，JPR と共有され，opcode フィールドの”010000”～”011111”まで使用でき，計 16 個である．

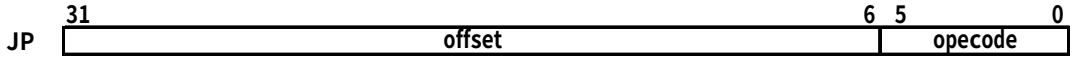
BR Type



rs1 はソースレジスタのインデックスを示す．offset は即値データである．下位 6bit は opecode であるが，BR Type の場合は”001001”～”001010”まで使用している．アドレッシングモードは即値アクセスである．

BR Type への追加可能命令数は他 MA，JP，JPR と共有され，opcode フィールドの”010000”～”011111”まで使用でき，計 16 個である．

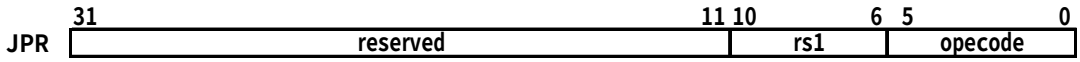
JP Type



offset は即値データである．下位 6bit は opecode であるが，JP Type の場合は”001011”～”001101”まで使用している．アドレッシングモードは即値アクセスである．

JP Type への追加可能命令数は他 MA，BR，JPR と共有され，opcode フィールドの”010000”～”011111”まで使用でき，計 16 個である．

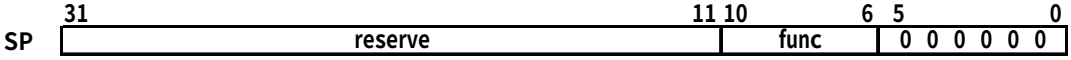
JPR Type



rs1 はアドレスが格納されたレジスタのインデックスである．下位 6bit は opecode であるが，JPR Type の場合は”001110”～”001111”まで使用している．アドレッシングモードはレジスタ間接アクセスである．

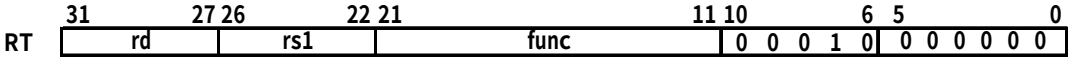
JPR Type への追加可能命令数は他 MA，BR，JP と共有され，opcode フィールドの”010000”～”011111”まで使用でき，計 16 個である．

SP Type



func は命令の種類を示す. 下位 7bit は opecode であるが, SP Type の場合は”0000000”に固定される.

RT Type



func は命令の種類を示す. 下位 11bit は opecode であるが, RT Type の場合は”000100000000”に固定される.

2.2. Instructions

2.2.1. Behavior Notation

BrownieSTD の命令ビヘイビアを記述するための記法を以下の表に定義する．
なお，比較演算子に(unsigned)が付いた場合は符号無し比較であることを示す．

表 3 ビヘイビア記法と意味

Behavior記法	意味
A + B	算術加算
A - B	算術減算
A * B	算術積
A / B	算術除算
A % B	算術剰余算
A & B	Bitwise AND
A | B	Bitwise OR
A ^ B	Bitwise XOR
~ A	Bitwise NOT
A >> B	AをBの値だけ論理右シフト
A << B	AをBの値だけ論理左シフト
A >>> B	AをBの値だけ算術右シフト
A <<< B	AをBの値だけ算術左シフト
zero(A, B)	データAをBビットまでのゼロ拡張
sign(A, B)	データAをBビットまでの符号拡張
conj(A, .., Z)	AからZの連接（Aが上位）
min(A, B)	{A, B}の最小値
max(A, B)	{A, B}の最大値
A == B	AとBが等価であるかの真偽値
A != B	AとBが非等価であるかの真偽値
A < B	AがBよりも小であるかの真偽値
A > B	AがBよりも大であるかの真偽値
A >= B	AがB以上であるかの真偽値
A <= B	AがB以下であるかの真偽値
A && B	論理積
A || B	論理和
! A	論理否定
C ? A : B	Cが真であればA，偽であればB
A = B	右辺Bから左辺Aへのデータ転送
if(exp)	式expが真であれば文を実行
PC	プログラムカウンタの値
LinkRegister	リンクレジスタの値（GPR3）
StatusRegister	ステータスレジスタの値（GPR1）
GPR(n)	インデックスがnの汎用レジスタの値
Data(addr)	addr番地に格納されているのデータメモリの値（バイト単位
A[B]	データAのBビット目の値
A[B..C]	データAのBビットからCビット目までの切り出し
CURADDR	その命令が配置されている命令メモリのアドレス
TRPBASE	TRAP命令のベースアドレス
rs1, rs2, rd	ソースレジスタ番号
immediate	即値
offset	オフセット値

2.2.2. Instruction Descriptions

BrownieSTD の各命令の詳細を示す. 命令の詳細フォーマットは以下のとおりである.

Instruction Type: *命令が属するタイプ*
Instruction Format: 命令のアセンブリフォーマット
Flag Change: 変化するフラグの種類 (C, Z, S, V), (-, -, -, V)等
Exception: この命令により例外が発生する場合は発生ステージ番号
Behavior:
ビヘイビア記法を用いた命令の動作記述

Description:
命令の概要

Flag Change 欄に記載される例は

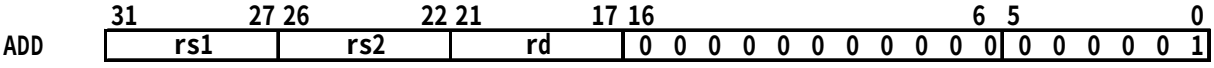
C: Carry（キャリー）
Z: Zero（ゼロ）
S: Sign（符号）
V: Overflow（オーバーフロー）
ー: それまでの値を保持
x: 不定

を示している.

なお, Flag Change は 2 命令後にステータスレジスタに反映される. Flag Change をチェックする場合は 2 命令後以降で参照しなければならない. Flag Change のタイミングについては「Appendix B.3. フラグレジスタの更新タイミング」を参照.
また, キャリーフラグ(C)は加算時と減算時において以下のような意味を持っている.

- 加算時
  - C = 1：キャリー発生
  - C = 0：キャリー未発生
- 減算時
  - C = 1：ボロー未発生
  - C = 0：ボロー発生

ADD — *Addition*



Instruction Type: *RR Type*

Instruction Format: ADD *rd*, *rs, rs2*

Flag Change: (C, Z, S, V)

Exception: No

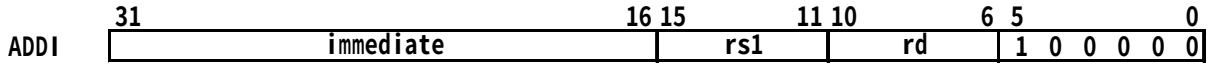
Behavior:

```
GPR(rd) = GPR(rs1) + GPR(rs2);
```

Description:

GPR(rs1) と GPR(rs2) の値を符号付算術加算し，結果を GPR(rd) に書き込む．この演算によるフラグ変化（Carry, Zero, Sign, Overflow）は Status Register に反映される．

ADDI — *Addition with Immediate*



Instruction Type: *RI Type*

Instruction Format: ADDI *rd, rs1, immediate*

Flag Change: (C, Z, S, V)

Exception: No

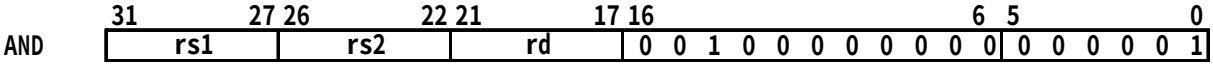
Behavior:

```
GPR(rd) = GPR(rs1) + sign(immediate, 32);
```

Description:

GPR(rs1) と immediate の値を符号付算術加算し，結果を GPR(rd) に書き込む．この演算のフラグ（Carry, Zero, Sign, Overflow）は Status Register に反映される．immediate の値は符号付の値として扱われる．

AND — *Logical AND*



Instruction Type: *RR Type*

Instruction Format: AND *rd, rs1, rs2*

Flag Change: (-, -, -, -)

Exception: No

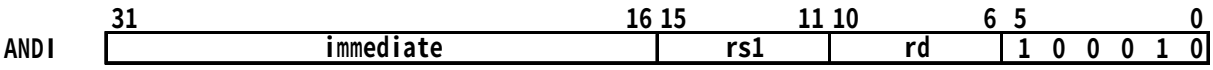
Behavior:

```
GPR(rd) = GPR(rs1) & GPR(rs2);
```

Description:

GPR(rs1) と GPR(rs2) の論理積を計算し，結果を GPR(rd) に書き込む.

ANDI — *Logical AND with Immediate*



Instruction Type: *RI Type*

Instruction Format: ANDI *rd, rs1, immediate*

Flag Change: (-, -, -, -)

Exception: No

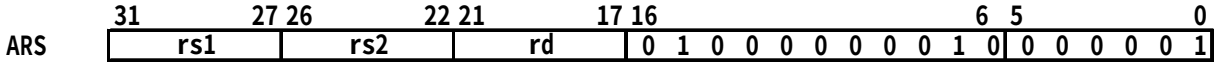
Behavior:

```
GPR(rd) = GPR(rs1) & zero(immediate, 32);
```

Description:

GPR(rs1) と immediate の論理積を計算し，結果を GPR(rd) に書き込む.
immediate の値は符号なしの値として扱われる.

ARS — *Arithmetic Right Shift*



Instruction Type: *RR Type*

Instruction Format: ARS *rd, rs1, rs2*

Flag Change: (-, -, -, -)

Exception: No

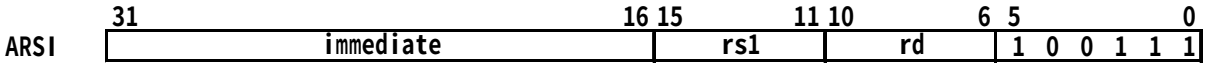
Behavior:

```
GPR(rd) = GPR(rs1) >>> GPR(rs2)[4..0];
```

Description:

GPR(rs1) の値を GPR(rs2)だけ算術右シフトし結果を GPR(rd) に書き込む. GPR(rs2)の値は有効な下位ビットのみ評価され，残りは無視される.

ARSI — *Arithmetic Right Shift with Immediate*



Instruction Type: *RI Type*

Instruction Format: ARSI *rd, rs1, immediate*

Flag Change: (-, -, -, -)

Exception: No

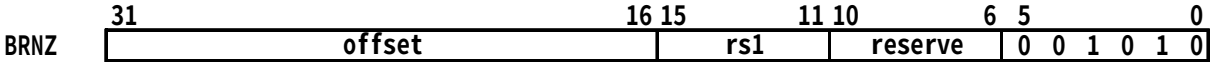
Behavior:

```
GPR(rd) = GPR(rs1) >>> immediate[4..0];
```

Description:

GPR(rs1) を immediate の値で算術右シフトし，結果を GPR(rd) に書き込む. immediate の値は有効な下位ビットのみ評価され，残りは無視される.

BRNZ — *Branch Not Zero*



Instruction Type: *BR Type*

Instruction Format: BRNZ *rs1, offset*

Flag Change: (-, -, -, -)

Exception: No

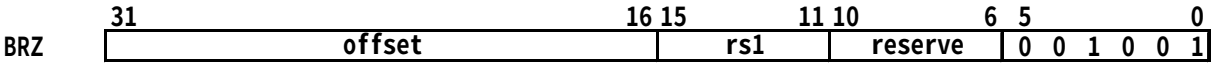
Behavior:

```
if(GPR(rs1) != 0)
  PC = CURADDR + 4 + sign (offset, 32);
```

Description:

もし GPR(rs1)の値が 0 でなければ，offset の値を符号拡張して PC の値に加算する（相対アドレスジャンプ）．この時使用されるジャンプの基準点はこの命令の次の命令のアドレスである．

BRZ — *Branch Zero*



Instruction Type: *BR Type*

Instruction Format: BRZ *rs1, offset*

Flag Change: (-, -, -, -)

Exception: No

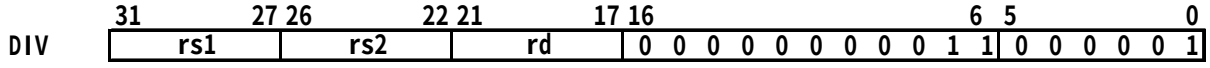
Behavior:

```
if(GPR(rs1) == 0)
  PC = CURADDR + 4 + sign (offset, 32);
```

Description:

もし GPR(rs1)の値が 0 ならば, offset の値を符号拡張して PC の値に加算する（相対アドレスジャンプ）．この時使用されるジャンプの基準点はこの命令の次の命令のアドレスである．

DIV — *Division*



Instruction Type: *RR Type*

Instruction Format: DIV *rd, rs1, rs2*

Flag Change: (-, -, -, -)

Exception: No

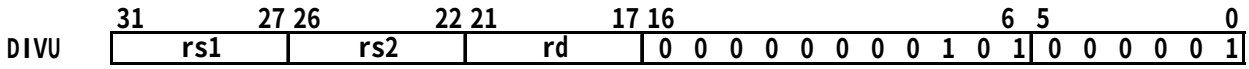
Behavior:

```
GPR(rd) = GPR(rs1) / GPR(rs2);
```

Description:

GPR(rs1) と GPR(rs2) の値を符号付算術除算し，結果を GPR(rd) に書き込む．ただし，0 除算，またはオバーフローが発生するような演算（0x80000000 / 0xFFFFFFFF）が行われた場合の結果は不定である．

DIVU — *Unsigned Division*



Instruction Type: *RR Type*

Instruction Format: DIVU *rd, rs1, rs2*

Flag Change: (-, -, -, -)

Exception: No

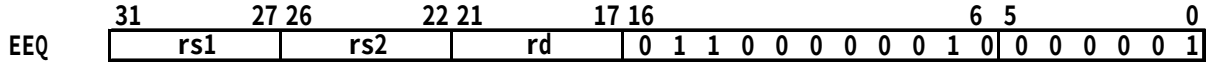
Behavior:

```
GPR(rd) = GPR(rs1) /(unsigned) GPR(rs2);
```

Description:

GPR(rs1) と GPR(rs2) の値を符号なし算術除算し, 結果を GPR(rd) に書き込む.

EEQ — *Evaluate Equal to*



Instruction Type: *RR Type*

Instruction Format: EEQ *rd, rs1, rs2*

Flag Change: (-, -, -, -)

Exception: No

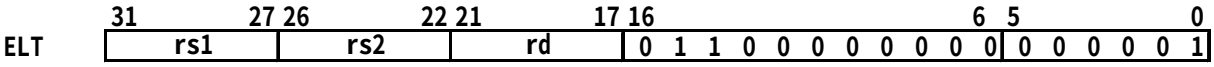
Behavior:

```
GPR(rd) = (GPR(rs1) == GPR(rs2)) ? 1:0;
```

Description:

GPR(rs1) の値が GPR(rs2)の値と等しい場合, GPR(rd) に 1 を書き込む. それ以外の場合は 0 を書き込む. このとき, GPR(rs1)と GPR(rs2)の値は符号付の値として評価される.

ELT — *Evaluate Less Than*



Instruction Type: *RR Type*

Instruction Format: ELT *rd, rs1, rs2*

Flag Change: (-, -, -, -)

Exception: No

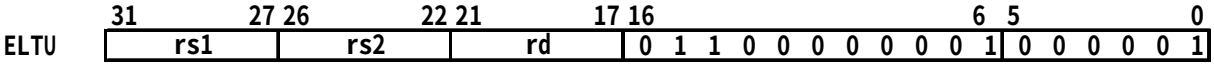
Behavior:

```
GPR(rd) = (GPR(rs1) < GPR(rs2)) ? 1:0;
```

Description:

GPR(rs1) の値が GPR(rs2)の値よりも小さい場合, GPR(rd) に 1 を書き込む. それ以外の場合は 0 を書き込む. このとき, GPR(rs1)と GPR(rs2)の値は符号付の値として評価される.

ELTU — *Evaluate Less Than Unsigned*



Instruction Type: *RR Type*

Instruction Format: ELTU *rd, rs1, rs2*

Flag Change: (-, -, -, -)

Exception: No

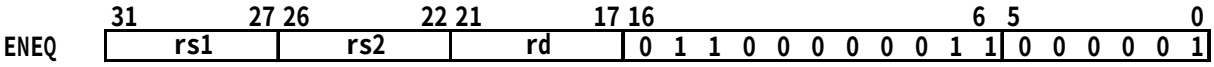
Behavior:

```
GPR(rd) = (GPR(rs1) <(unsigned) GPR(rs2)) ? 1:0;
```

Description:

GPR(rs1) の値が GPR(rs2)の値よりも小さい場合, GPR(rd) に 1 を書き込む. それ以外の場合は 0 を書き込む. このとき, GPR(rs1)と GPR(rs2)の値は符号無しの値として評価される.

ENEQ — *Evaluate Not Equal to*



Instruction Type: *RR Type*

Instruction Format: ENEQ *rd, rs1, rs2*

Flag Change: (-, -, -, -)

Exception: No

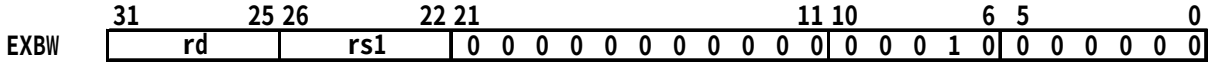
Behavior:

```
GPR(rd) = (GPR(rs1) != GPR(rs2)) ? 1:0;
```

Description:

GPR(rs1) の値が GPR(rs2)の値と等しくない場合, GPR(rd) に 1 を書き込む. それ以外の場合は 0 を書き込む. このとき, 書く GPR(rs1)と GPR(rs2)の値は符号付の値として評価される.

EXBW — *Extend Byte to Word*



Instruction Type: *RT Type*

Instruction Format: EXBW *rd, rs1*

Flag Change: (-, -, -, -)

Exception: No

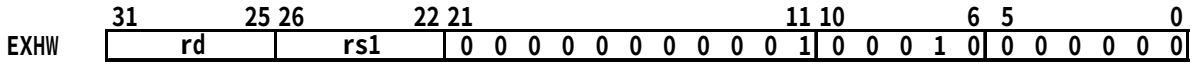
Behavior:

```
GPR(rd) = sign(GPR(rs1)[7:0], 32);
```

Description:

GPR(rs1) の下位 8 ビットを 32 ビットデータに符号拡張する.

EXHW — *Extend Half word to Word*



Instruction Type: *RT Type*

Instruction Format: EXHW *rd, rs1*

Flag Change: (-, -, -, -)

Exception: No

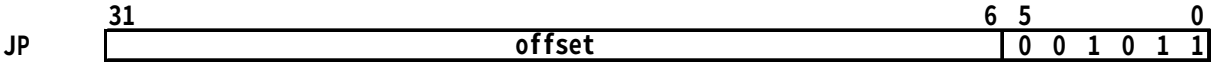
Behavior:

```
GPR(rd) = sign(GPR[15:0], 32);
```

Description:

GPR(rs1) の下位 16 ビットを 32 ビットデータに符号拡張する.

JP — *Jump*



Instruction Type: *JP Type*

Instruction Format: JP *offset*

Flag Change: (-, -, -, -)

Exception: No

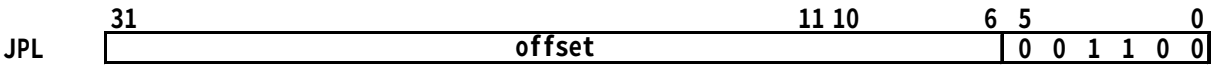
Behavior:

```
PC = CURADDR + 4 + sign(offset, 32);
```

Description:

無条件に offset の値を符号拡張して PC の値に加算する(相対アドレスジャンプ). この時使用されるジャンプの基準点はこの命令の次の命令のアドレスである. offset の値は符号付の値として扱われる.

JPL — *Jump and Link*



Instruction Type: *JP Type*

Instruction Format: JPL *offset*

Flag Change: (-, -, -, -)

Exception: No

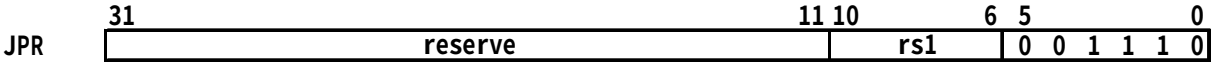
Behavior:

```
LinkRegister = CURADDR + 4;
PC           = CURADDR + 4 + sign(offset, 32);
```

Description:

LinkRegister に次の命令のアドレス(PC + 4)を書き込み，無条件に offset の値を符号拡張して PC の値に加算する（相対アドレスジャンプ). この時使用されるジャンプの基準点はこの命令の次の命令のアドレスである. offset の値は符号付の値として扱われる.

JPR — *Jump with Register*



Instruction Type: *JPR Type*

Instruction Format: JPR *rs1*

Flag Change: (-, -, -, -)

Exception: No

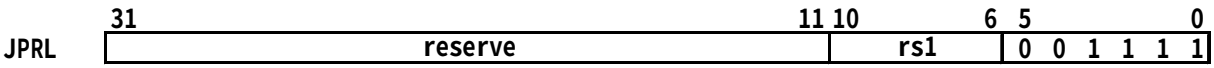
Behavior:

```
PC = GPR(rs1);
```

Description:

無条件に GPR(rs1)の値を PC に書き込む（絶対アドレスジャンプ）.

JPRL — *Jump with Register and Link*



Instruction Type: *JPR Type*

Instruction Format: JPRL *rs1*

Flag Change: (-, -, -, -)

Exception: No

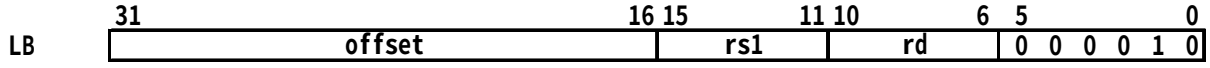
Behavior:

```
LinkRegister = CURADDR + 4;
PC           = GPR(rs1);
```

Description:

LinkRegister に次の命令のアドレス(PC + 4)を書き込み, 無条件に GPR(rs1)の値を PC に書き込む（絶対アドレスジャンプ）.

LB — *Load Byte*



Instruction Type: *MA Type*

Instruction Format: LB *rd, offset(rs1)*

Flag Change: (-, -, -, -)

Exception: No

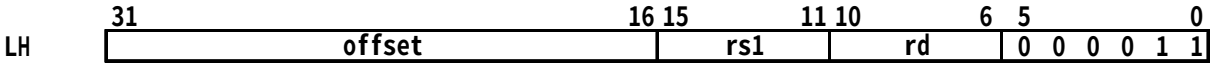
Behavior:

```
GPR(rd) = sign(Data(GPR(rs1) + offset), 32);
```

Description:

GPR(rs1)の値と offset の値を加算した値を実効アドレスとしてデータメモリにアクセスする． offset は符号付の値として評価される．

LH — *Load Half word*



Instruction Type: *MA Type*

Instruction Format: LH *rd, offset(rs1)*

Flag Change: (-, -, -, -)

Exception: No

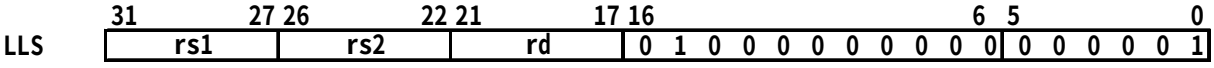
Behavior:

```
GPR(rd) = sign( conj(Data(GPR(rs1) + offset),
                     Data(GPR(rs1) + offset + 1), 32);
```

Description:

GPR(rs1)の値と offset の値を加算した値を実効アドレスとしてデータメモリにアクセスする．offset は符号付の値として評価される．

LLS — *Logical Left Shift*



Instruction Type: *RR Type*

Instruction Format: LLS *rd, rs1, rs2*

Flag Change: (-, -, -, -)

Exception: No

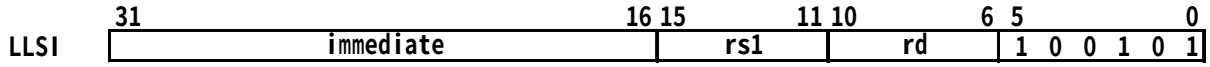
Behavior:

```
GPR(rd) = GPR(rs1) << GPR(rs2)[4..0];
```

Description:

GPR(rs1) の値を GPR(rs2)だけ論理左シフトし結果を GPR(rd) に書き込む. GPR(rs2)の値は有効な下位ビットのみ評価され，残りは無視される.

LLSI — *Logical Left Shift with Immediate*



Instruction Type: *RI Type*

Instruction Format: LLSI *rd, rs1, immediate*

Flag Change: (-, -, -, -)

Exception: No

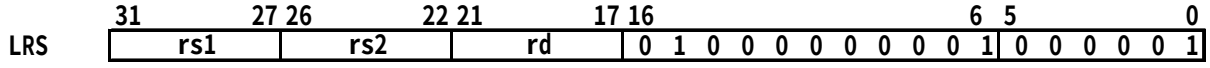
Behavior:

```
GPR(rd) = GPR(rs1) << immediate[4..0];
```

Description:

GPR(rs1) を immediate の値で論理左シフトし，結果を GPR(rd) に書き込む. immediate の値は有効な下位ビットのみ評価され，残りは無視される.

LRS — *Logical Right Shift*



Instruction Type: *RR Type*

Instruction Format: LRS *rd, rs1, rs2*

Flag Change: (-, -, -, -)

Exception: No

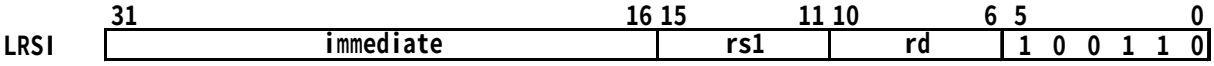
Behavior:

```
GPR(rd) = GPR(rs1) >> GPR(rs2)[4..0];
```

Description:

GPR(rs1) の値を GPR(rs2)だけ論理右シフトし結果を GPR(rd) に書き込む.
GPR(rs2)の値は有効な下位ビットのみ評価され，残りは無視される.

LRSI — *Logical Right Shift with Immediate*



Instruction Type: *RI Type*

Instruction Format: LRSI *rd, rs1, immediate*

Flag Change: (-, -, -, -)

Exception: No

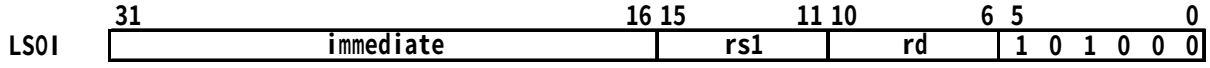
Behavior:

```
GPR(rd) = GPR(rs1) >> immediate[4..0];
```

Description:

GPR(rs1) を immediate の値で論理右シフトし，結果を GPR(rd) に書き込む.
immediate の値は有効な下位ビットのみ評価され，残りは無視される.

LSOI — *Left Shift and OR with Immediate*



Instruction Type: *RI Type*

Instruction Format: LSOI *rd, rs1, immediate*

Flag Change: (-, -, -, -)

Exception: No

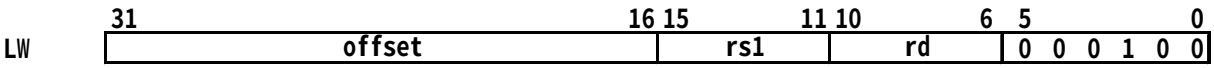
Behavior:

```
GPR(rd) = (GPR(rs1) << 16) | zero(immediate, 32);
```

Description:

GPR(rs1) を 16bit 左シフトした後，immediate との論理和を計算し，結果を GPR(rd) に書き込む.

LW — *Load Word*



Instruction Type: *MA Type*

Instruction Format: LW *rd, offset(rs1)*

Flag Change: (-, -, -, -)

Exception: No

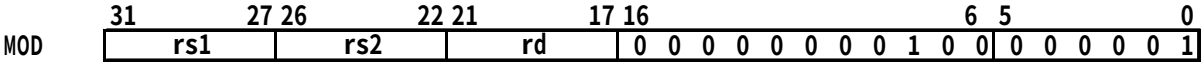
Behavior:

```
GPR(rd) = conj(Data(GPR(rs1) + offset),
               Data(GPR(rs1) + offset + 1),
               Data(GPR(rs1) + offset + 2),
               Data(GPR(rs1) + offset + 3));
```

Description:

GPR(rs1)の値と offset の値を加算した値を実効アドレスとしてデータメモリにアクセスする． offset は符号付の値として評価される.

MOD — *Modulo arithmetic*



Instruction Type: *RR Type*

Instruction Format: MOD *rd, rs1, rs2*

Flag Change: (-, -, -, -)

Exception: No

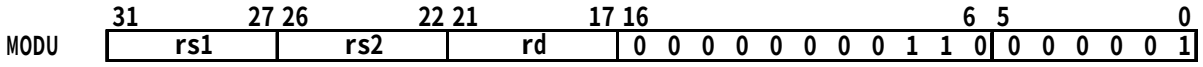
Behavior:

```
GPR(rd) = GPR(rs1) % GPR(rs2);
```

Description:

GPR(rs1)と GPR(rs2)の値を符号付算術剰余算し，結果を GPR(rd) に書き込む．この演算の結果は必ず正の値となる．ただし，0 除算，またはオバーフローが発生するような演算（0x80000000 % 0xFFFFFFFF）が行われた場合の結果は不定である．

MODU — *Unsigned Modulo arithmetic*



Instruction Type: *RR Type*

Instruction Format: MODU *rd, rs1, rs2*

Flag Change: (-, -, -, -)

Exception: No

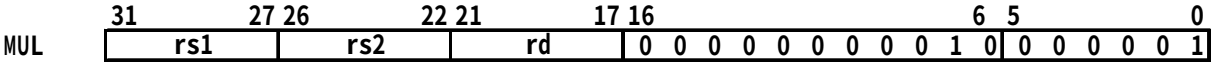
Behavior:

```
GPR(rd) = GPR(rs1) %(unsinged) GPR(rs2);
```

Description:

GPR(rs1)と GPR(rs2)の値を符号なし算術剰余算し, 結果を GPR(rd) に書き込む．この演算の結果は必ず正の値となる．

MUL — *Multiplication*



Instruction Type: *RR Type*

Instruction Format: MUL *rd, rs1, rs2*

Flag Change: (-, -, -, -)

Exception: No

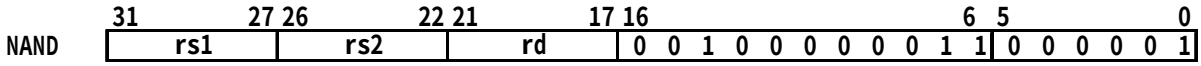
Behavior:

```
GPR(rd) = GPR(rs1) * GPR(rs2);
```

Description:

GPR(rs1) と GPR(rs2) の値を符号付算術乗算し，結果を GPR(rd) に書き込む.

NAND — *Logical NAND*



Instruction Type: *RR Type*

Instruction Format: NAND *rd, rs1, rs2*

Flag Change: (-, -, -, -)

Exception: No

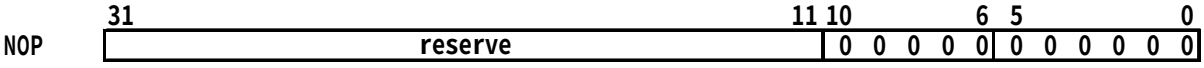
Behavior:

```
GPR(rd) = ~(GPR(rs1) & GPR(rs2));
```

Description:

GPR(rs1) と GPR(rs2) の論理否定積を計算し，結果を GPR(rd) に書き込む.

NOP — *No Operation*



Instruction Type: *SP Type*

Instruction Format: NOP

Flag Change: (-, -, -, -)

Exception: No

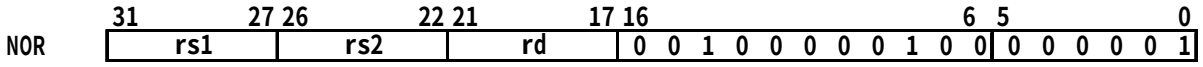
Behavior:

```
(Nothing)
```

Description:

NOP 命令は何も有効な動作をしない.

NOR — *Logical NOR*



Instruction Type: *RR Type*

Instruction Format: NOR *rd, rs1, rs2*

Flag Change: (-, -, -, -)

Exception: No

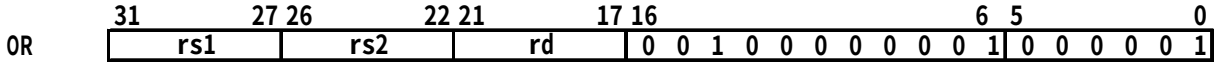
Behavior:

```
GPR(rd) = ~(GPR(rs1) | GPR(rs2));
```

Description:

GPR(rs1) と GPR(rs2) の論理否定和を計算し，結果を GPR(rd) に書き込む.

OR — *Logical OR*



Instruction Type: *RR Type*

Instruction Format: OR *rd, rs1, rs2*

Flag Change: (-, -, -, -)

Exception: No

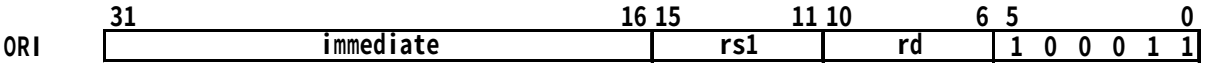
Behavior:

```
GPR(rd) = GPR(rs1) | GPR(rs2);
```

Description:

GPR(rs1) と GPR(rs2) の論理和を計算し，結果を GPR(rd) に書き込む.

ORI — *Logical OR with Immediate*



Instruction Type: *RI Type*

Instruction Format: ORI *rd, rs1, immediate*

Flag Change: (-, -, -, -)

Exception: No

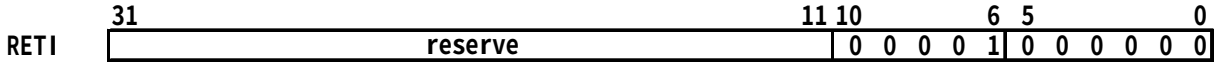
Behavior:

```
GPR(rd) = GPR(rs1) | zero(immediate, 32);
```

Description:

GPR(rs1) と immediate の論理和を計算し，結果を GPR(rd) に書き込む.
immediate の値は符号なしの値として扱われる.

RETI — *Return from Interrupt*



Instruction Type: *SP Type*

Instruction Format: RETI

Flag Change: (-, -, -, -)

Exception: No

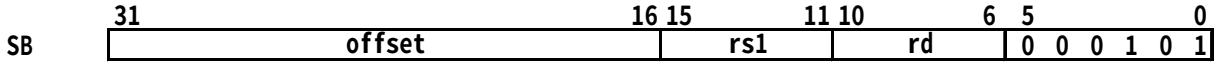
Behavior:

```
Status Register[8]          = 1;
Status Register[9]          = 1;
Status Register[15:14]      = "01";
PC                          = Interrupt Return Register;
```

Description:

RETI 命令は割り込みを全て有効にして Interrupt Return Register の値を PC に書き込む.

SB — *Store Byte*



Instruction Type: *MA Type*

Instruction Format: SB *offset (rd), rs1*

Flag Change: (-, -, -, -)

Exception: No

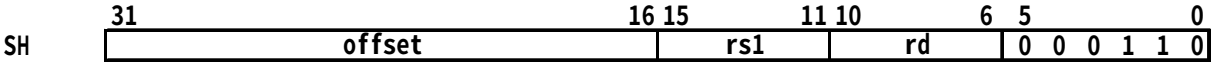
Behavior:

```
Data(GPR(rd) + offset) = GPR(rs1)[7..0];
```

Description:

GPR(rd)の値と offset の値を加算した値を実効アドレスとして GPR(rs1)の値をデータメモリに書き込む. offset は符号付の値として評価される.

SH — *Store Half word*



Instruction Type: *MA Type*

Instruction Format: SH *offset(rd), rs1*

Flag Change: (-, -, -, -)

Exception: No

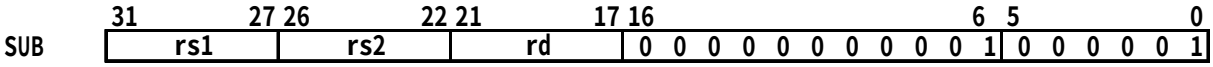
Behavior:

```
Data(GPR(rd) + offset)     = GPR(rs1)[15..8];
Data(GPR(rd) + offset + 1) = GPR(rs1)[7..0];
```

Description:

GPR(rd)の値と offset の値を加算した値を有効アドレスとして GPR(rs1)の値をデータメモリに書き込む．offset は符号付の値として評価される．

SUB — *Subtraction*



Instruction Type: *RR Type*

Instruction Format: SUB *rd, rs1, rs2*

Flag Change: (C, Z, S, V)

Exception: No

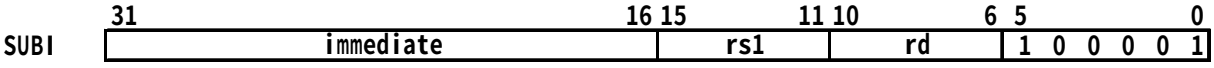
Behavior:

```
GPR(rd) = GPR(rs1) - GPR(rs2);
```

Description:

GPR(rs1) と GPR(rs2) の値を符号付算術減算し，結果を GPR(rd) に書き込む．この演算によるフラグ変化（Carry, Zero, Sign, Overflow）は Status Register に反映される．

SUBI — *Subtraction with Immediate*



Instruction Type: *RI Type*

Instruction Format: SUBI *rd, rs1, immediate*

Flag Change: (C, Z, S, V)

Exception: No

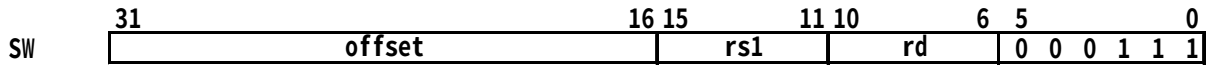
Behavior:

```
GPR(rd) = GPR(rs1) - sign(immediate, 32);
```

Description:

GPR(rs1) と immediate の値を符号付算術減算し, 結果を GPR(rd) に書き込む. この演算によるフラグ変化（Carry, Zero, MSB, Overflow）は Status Register に反映される． immediate の値は符号付の値として扱われる.

SW — *Store Word*



Instruction Type: *MA Type*

Instruction Format: SW *offset(rd), rs1*

Flag Change: (-, -, -, -)

Exception: No

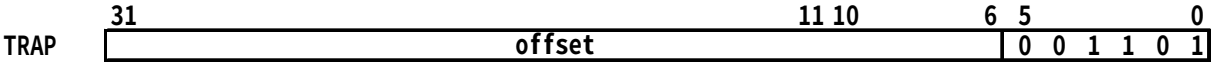
Behavior:

```
Data(GPR(rd) + offset)     = GPR(rs1)[31..24];
Data(GPR(rd) + offset + 1) = GPR(rs1)[23..16];
Data(GPR(rd) + offset + 2) = GPR(rs1)[15..8];
Data(GPR(rd) + offset + 3) = GPR(rs1)[7..0];
```

Description:

GPR(rd)の値と offset の値を加算した値を有効アドレスとして GPR(rs1)の値をデータメモリに書き込む．offset は符号付の値として評価される.

TRAP *Trap*



Instruction Type: *JP Type*

Instruction Format: TRAP *offset*

Flag Change: (-, -, -, -)

Exception: Stage 3

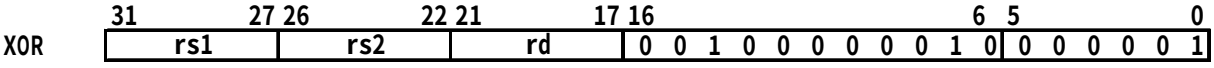
Behavior:

```
if(Status Register[9] == 1){
  Status Register[8]              = 0;
  Status Register[9]              = 0;
  Status Register[15:14]          = "00";
  Interrupt Return Register       = CURADDR;
  PC                              = TRPBASE + offset;
}
```

Description:

ソフトウェア割り込みを発生させる．オペランドの offset はソフトウェア割り込みの種類（ハンドラアドレス）を示す．

XOR — *Logical Exclusive OR*



Instruction Type: *RR Type*

Instruction Format: XOR *rd, rs1, rs2*

Flag Change: (-, -, -, -)

Exception: No

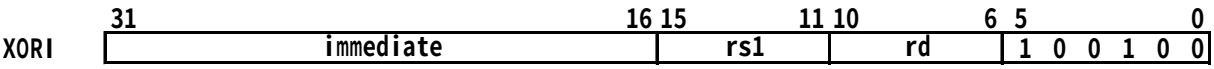
Behavior:

```
GPR(rd) = GPR(rs1) ^ GPR(rs2);
```

Description:

GPR(rs1) と GPR(rs2) の排他的論理和を計算し，結果を GPR(rd) に書き込む.

XORI — *Logical Exclusive OR with Immediate*



Instruction Type: *RI Type*

Instruction Format: XORI *rd, rs1, immediate*

Flag Change: (-, -, -, -)

Exception: No

Behavior:

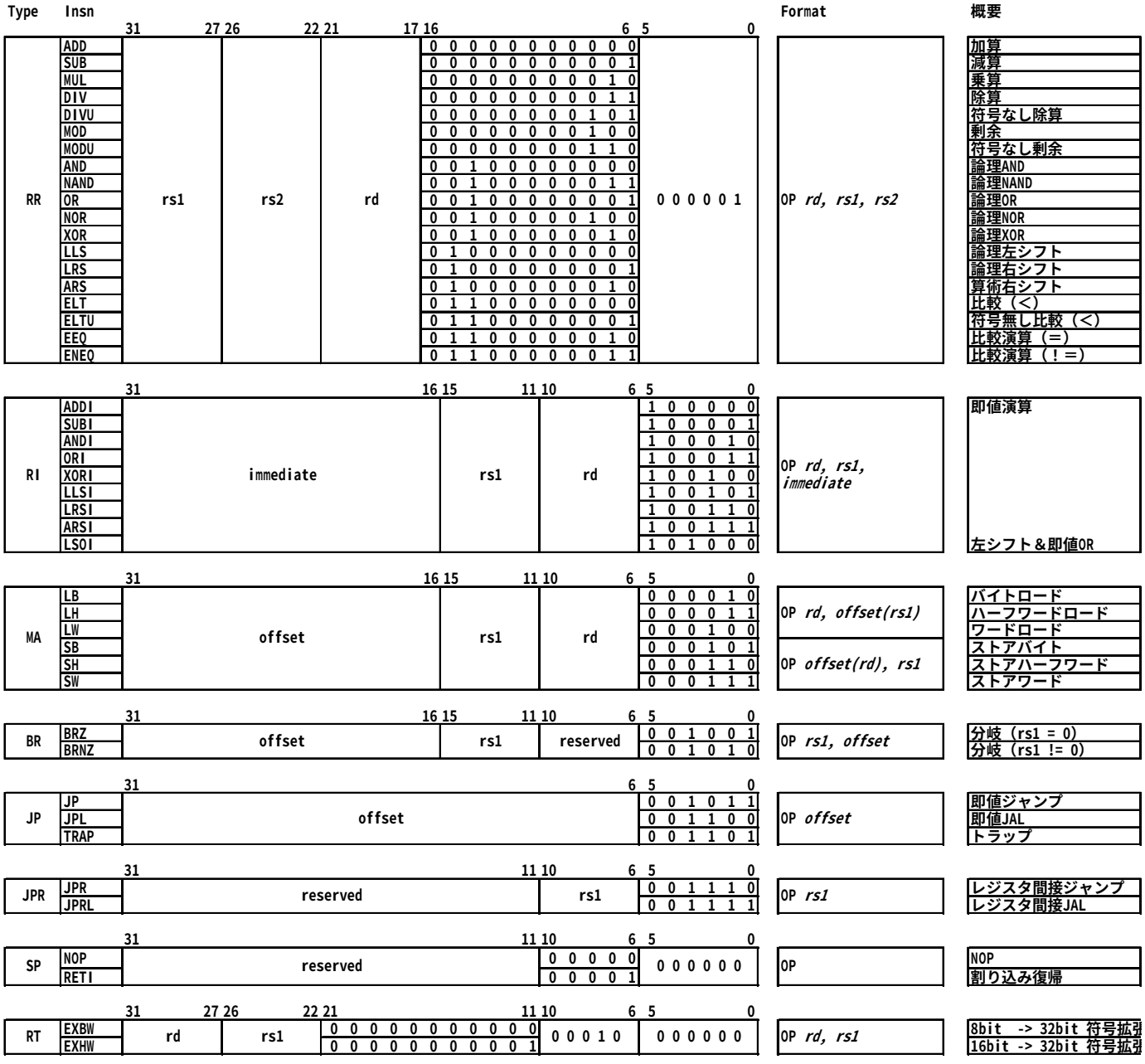
```
GPR(rd) = GPR(rs1) ^ zero (immediate, 32);
```

Description:

GPR(rs1) と immediate の排他的論理和を計算し,結果を GPR(rd) に書き込む.
immediate の値は符号なしの値として扱われる

2.2.3. Instruction Set Quick Reference

以下に BrownieSTD の命令，タイプおよびそのフォーマットとコード割付，およびニーモニック・フォーマットを示す.



Type	Insn	31–27	26–22	21–17	16–6	5–0	Format	概要
RR	ADD	rs1	rs2	rd	0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 1	OP *rd, rs1, rs2*	加算
	SUB				0 0 0 0 0 0 0 0 0 0 1			減算
	MUL				0 0 0 0 0 0 0 0 0 1 0			乗算
	DIV				0 0 0 0 0 0 0 0 0 1 1			除算
	DIVU				0 0 0 0 0 0 0 0 1 0 1			符号なし除算
	MOD				0 0 0 0 0 0 0 0 1 0 0			剰余
	MODU				0 0 0 0 0 0 0 0 1 1 0			符号なし剰余
	AND				0 0 1 0 0 0 0 0 0 0 0			論理AND
	NAND				0 0 1 0 0 0 0 0 0 1 1			論理NAND
	OR				0 0 1 0 0 0 0 0 0 0 1			論理OR
	NOR				0 0 1 0 0 0 0 0 1 0 0			論理NOR
	XOR				0 0 1 0 0 0 0 0 0 1 0			論理XOR
	LLS				0 1 0 0 0 0 0 0 0 0 0			論理左シフト
	LRS				0 1 0 0 0 0 0 0 0 0 1			論理右シフト
	ARS				0 1 0 0 0 0 0 0 0 1 0			算術右シフト
	ELT				0 1 1 0 0 0 0 0 0 0 0			比較（<）
	ELTU				0 1 1 0 0 0 0 0 0 0 1			符号無し比較（<）
	EEQ				0 1 1 0 0 0 0 0 0 1 0			比較演算（=）
	ENEQ				0 1 1 0 0 0 0 0 0 1 1			比較演算（！=）

		31–16	15–11	10–6	5–0	Format	概要
RI	ADDI	immediate	rs1	rd	1 0 0 0 0 0	OP *rd, rs1, immediate*	即値演算
	SUBI				1 0 0 0 0 1		
	ANDI				1 0 0 0 1 0		
	ORI				1 0 0 0 1 1		
	XORI				1 0 0 1 0 0		
	LLSI				1 0 0 1 0 1		
	LRSI				1 0 0 1 1 0		
	ARSI				1 0 0 1 1 1		
	LSOI				1 0 1 0 0 0		左シフト＆即値OR

		31–16	15–11	10–6	5–0	Format	概要
MA	LB	offset	rs1	rd	0 0 0 0 1 0	OP *rd, offset(rs1)*	バイトロード
	LH				0 0 0 0 1 1		ハーフワードロード
	LW				0 0 0 1 0 0		ワードロード
	SB				0 0 0 1 0 1	OP *offset(rd), rs1*	ストアバイト
	SH				0 0 0 1 1 0		ストアハーフワード
	SW				0 0 0 1 1 1		ストアワード

		31–16	15–11	10–6	5–0	Format	概要
BR	BRZ	offset	rs1	reserved	0 0 1 0 0 1	OP *rs1, offset*	分岐（rs1 = 0）
	BRNZ				0 0 1 0 1 0		分岐（rs1 != 0）

		31–6	5–0	Format	概要
JP	JP	offset	0 0 1 0 1 1	OP *offset*	即値ジャンプ
	JPL		0 0 1 1 0 0		即値JAL
	TRAP		0 0 1 1 0 1		トラップ

		31–11	10–6	5–0	Format	概要
JPR	JPR	reserved	rs1	0 0 1 1 1 0	OP *rs1*	レジスタ間接ジャンプ
	JPRL			0 0 1 1 1 1		レジスタ間接JAL

		31–11	10–6	5–0	Format	概要
SP	NOP	reserved	0 0 0 0 0	0 0 0 0 0 0	OP	NOP
	RETI		0 0 0 0 1			割り込み復帰

		31–27	26–22	21–11	10–6	5–0	Format	概要
RT	EXBW	rd	rs1	0 0 0 0 0 0 0 0 0 0 0	0 0 0 1 0	0 0 0 0 0 0	OP *rd, rs1*	8bit -> 32bit 符号拡張
	EXHW			0 0 0 0 0 0 0 0 0 0 1				16bit -> 32bit 符号拡張

3. Interrupts

この章では BrownieSTD の割り込み処理に関して解説する.

3.1. Reset Interrupt

リセット割り込みはリセット入力ポートに ‘ 1 ’ が入力されることにより，起動する. リセット割り込みが起動されると，プロセッサ内部のレジスタは全てクリアされ，プログラムカウンタはリセット割り込みベクタ・アドレスにセットされる.

3.2. External Interrupt

外部割込みは Status Register の外部割り込み許可フラグビットを'0'にセットすることで無効化できる. 無効化した場合，EXTINT_IN にアサートされる外部割込みは無視される. 外部割込みが検出され，ハンドラが起動されると BrownieSTD は現在の処理を中断し，復帰すべき命令のアドレスを Interrupt Return Register に保存して外部割り込みベクタ 0x0FFE0400 番地に分岐し，外部・内部割込みを無効化する. また，同時に Status Register の実行モードフラグをカーネルモードへ変更する. ユーザーモードへの復帰は RETI 命令を用いる.

BrownieSTD の外部割込み要求タイミングを図 1 外部割り込み要求通知タイミングに示す. BrownieSTD に EXTINT_IN を通じて割り込み要求が通知されると，プロセッサ・コアが割り込み処理の準備を完了し，割り込み処理が開始された時点で EXTCATCH_OUT をアサートする. その間，EXTINT_IN はアサートし続けなければならない. EXTINT_IN がアサートされてから外部割込み処理が開始されるまで最低 5 サイクルかかるが，これはプロセッサの状態により変動（増加）する. したがって EXTCATCH_OUT を確認せずに EXTINT_IN のアサートを止めた場合，割り込み処理が検出されない可能性がある.

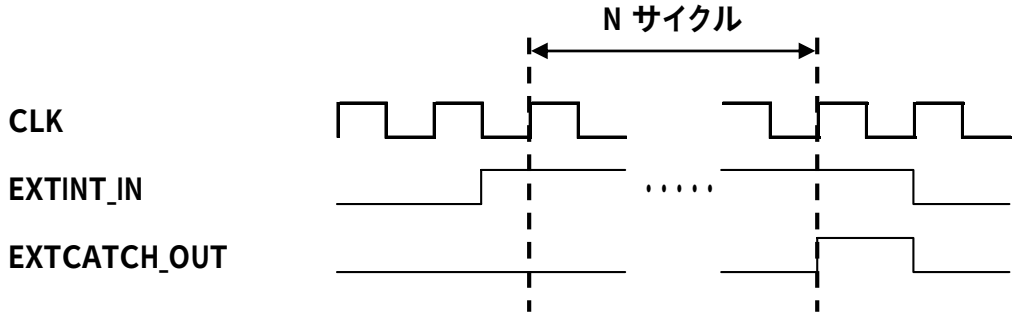


図 1 外部割り込み要求通知タイミング

3.2. Internal Interrupts

内部割込みは Status Register の内部割り込み許可フラグビットを'0'にセットすることで無効化できる．無効化した場合，全ての内部割込みは無視される．内部割込みハンドラが起動されると BrownieSTD は現在の処理を中断し，中断した命令のアドレスを Interrupt Return Register に保存して当該の内部割り込みベクタに分岐する．

3.2.1. TRAP

TRAP 命令を用いることでソフトウェア的に割り込みを発生させることが出来る．
TRAP 命令が実行されると Interrupt Return Register に TRAP 命令のアドレスを保存し，TRAP 割り込みベクタ（0x0FFE0800 + offset）に分岐する．また，同時に Status Register の実行モードフラグをカーネルモードへ変更する．ユーザーモードへの復帰は RETI 命令を用いる．
TRAP によって Interrupt Return Register に保存されるアドレスは例外を発生させた命令のアドレス，すなわち TRAP 自身のアドレスであるため，復帰するためには Interrupt Return Register の値を適切な値に設定しなさなければならない．

4. Memory Access

この章では BrownieSTD と外部メモリとの通信プロトコル，タイミングなどについて解説する．外部インターフェースの一覧については 1.6. Interface を参照.

4.1. Instruction Memory

BrownieSTD32 は命令メモリ・アクセスに対して読み込み専用である．データはアドレス送出サイクルに得られなければならない．命令メモリ・アクセスに使用される各信号線の意味を以下に示す.

IMEM_ADDR_OUT	：命令メモリ・アドレス送信バス
IMEM_ADDRERR_IN	：命令メモリ・アクセスエラー
IMEM_DATA_IN	：命令メモリ・データ受信バス

図 2 命令メモリ・リードアクセスにリードアクセスのタイミングを示す．フェッチアドレスが送信されたサイクルにそのアドレスのデータが確定されなければならない．何らかの要因でパイプラインがストールしている期間は，フェッチアドレスが固定されるので（図中 Addr(D)の期間），その間フェッチデータも保持し続ける必要がある.

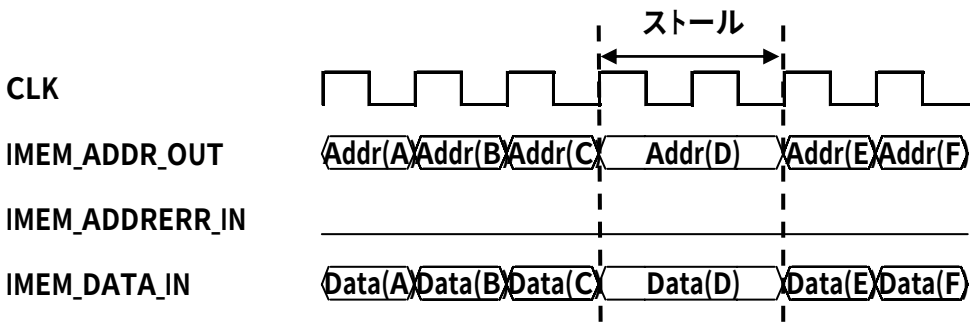


図 2 命令メモリ・リードアクセス

4.2. Data Memory

BrownieSTD32 はデータメモリに対して読み書きアクセス可能である．データメモリ・アクセスに使用される信号線の意味を以下に示す．

DMEM_ADDR_OUT　：データメモリ・アドレス
DMEM_ADDRERR_IN　：データメモリ・アクセスエラー
DMEM_DATA_OUT　：データメモリ・データ送信バス
DMEM_DATA_IN　：データメモリ・データ受信バス
DMEM_RW_OUT　：データメモリ・R/W モード
DMEM_WMODE_OUT　：データメモリ・アクセスモード
DMEM_EMODE_OUT　：データメモリ・符号拡張モード
DMEM_CANCEL_OUT　：データメモリ・アクセスキャンセル
DMEM_REQ_OUT　：データメモリ・アクセス要求
DMEM_ACK_IN　：データメモリ・アクセス完了

表 4 メモリ・アクセスモードと信号値の関係

	DMEM_RW_MODE_OUT	DMEM_WMODE_OUT	DMEM_EMODE_OUT
符号付バイト読み込み	0	"00"	1
符号付ハーフワード読み込み	0	"01"	1
符号付ワード読み込み	0	"11"	1
符号無しバイト読み込み	0	"00"	0
符号無しハーフワード読み込み	0	"01"	0
符号無しワード読み込み	0	"11"	0
バイト書き込み	1	"00"	0
ハーフワード書き込み	1	"01"	0
ワード書き込み	1	"11"	0

データメモリには幾つかのアクセスモードがある．各モードと信号値の関係を表 4 メモリ・アクセスモードと信号値の関係にまとめる．

データメモリへのアクセスタイミングを図 3 符号付ワード読み込みに示す．図は符号付ワード読み込みの例である．アクセス要求（REQ）と同時にアドレス，RW モード，アクセスモード，符号拡張モードが送信される．その後，BrownieSTD はアクセス完了（ACK）が通知されるまでその値を保持し続ける．メモリ側でデータが確定され，ACK が通知されると，Brownie はそのサイクルのデータを読み込み，実行を継続する．このほかのアクセスモードに関してもタイミングは全て同じであり，各制御信号値のみが異なる．図 4 符号付バイト書き込み例に符号付バイト書き込みの例を示す．書き込み時には書き込みアドレス，書き込みデータ，RW モード，アクセスモード，符号拡張モードが送信される．

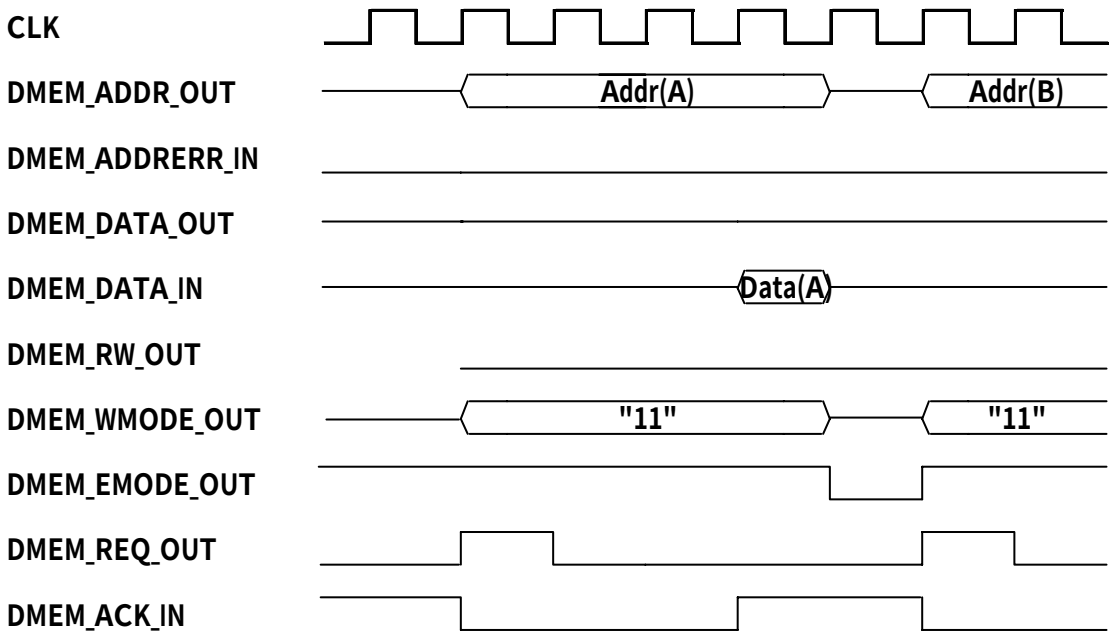


図 3 符号付ワード読み込み

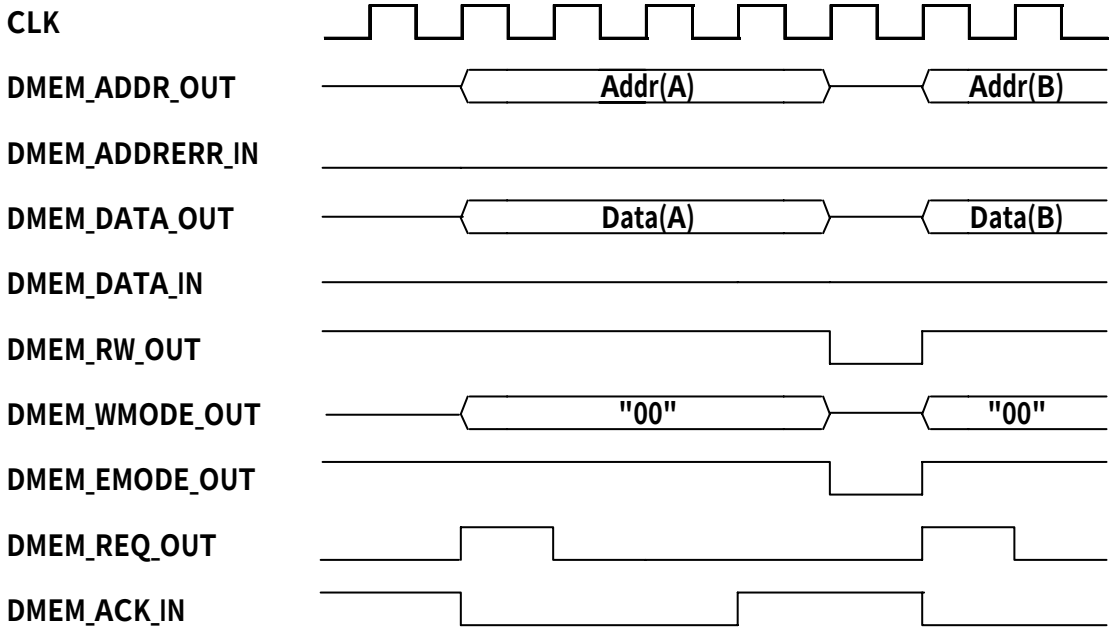


図 4 符号付バイト書き込み例

Appendix

Appendix A. 遅延ロード

遅延ロードとは，ロード命令によってロードされるデータが，ある一定期間（命令数）後に使用可能になるようなロード方式のことである．例えば，遅延ロード・スロットが 1 であるような場合で，以下のような命令列があった場合，

```
ope1    ----------------
ope2    ----------------
LW      %GPR1, 0(%GPR2)          ; ロード命令
ope3    ----------------         ; 遅延ロード・スロット
ope4    ----------------
```

ロード命令 LW によって GPR1 にデータがロードされるが, ロードされたデータ GPR1 は ope4 から有効（使用可能）であり，遅延ロード・スロット中の命令 ope3 は，それを利用できない．すなわち，遅延ロード・スロットには GPR1 を（ソースとしても，デスティネーションとしても）使用しない命令がスケジュール可能である．もし，遅延ロード・スロット中に GPR1 を使用する命令がスケジュールされると，以下のようなデータハザードが起こる．

```
ADD     %GPR2,%GPR0,%GPR1
LW      %GPR2,0(%GPR0)
ADD     %GPR3,%GPR0,%GPR2
```

の場合 GPR3 に入るのは GPR1 の値
(ロード直後の命令ではロードした値を利用できない)

```
LW      %GPR2,0(%GPR0)
ADD     %GPR2,%GPR0,%GPR1
ADD     %GPR3,%GPR0,%GPR2
```

の場合も GPR3 に入るのは GPR1 の値
(ロード直後にレジスタを上書きすると上書きした値が有効)

Appendix B. ステータスレジスタの扱い

Appendix B.1. ステータスレジスタの読み出し・書き込み

ステータスレジスタ（GPR1）に読み出し，または書き込みを行う場合，安全のため，事前に 2 命令 NOP を挿入すること．以下に例を示す．

```
ope1   ----------------
ope2   ----------------          ; should be NOP
ope3   ----------------          ; should be NOP
ope4   %GPR1, %GPRX, %GPRX       ; GPR1 is destination
ope5   ----------------
ope6   ----------------
```

上記のような命令列があり，ope4 が GPR1 をデスティネーション，またはソースに取るような命令だったとする（例中ではデスティネーションになっている），この場合，ope2 および ope3 は NOP とすべきである．もし，ope2，ope3 が NOP 以外の場合，ステータスレジスタの読み出し・書き込みが意図しない結果となるおそれがある．

Appendix B.2. ステータスレジスタへの書き込みの反映タイミング

ステータスレジスタ（GPR1）を更新した場合，ステータスレジスタの更新が反映されるのは 3 命令後である．以下に例を示す．

```
ope1   %GPR1, %GPRX, %GPRX       ; GPR1 is destination
ope2   ----------------
ope3   ----------------
ope4   ----------------
```

上記のような命令列があり，ope1 が GPR1 をデスティネーションに取る命令だとする．この場合，ope1 の結果が実際に GPR1 に反映されるのは ope4 の時点である．ope2，ope3 で GPR1 を読み出した場合，結果は不定である．

Appendix B.3. フラグレジスタの更新タイミング

フラグチェンジを伴うような命令を実行した場合，ステータスレジスタが更新される．ただし，フラグの更新は 2 命令後から有効である．以下に例を示す．

```
ope1   ----------------          ; change flag
ope2   ----------------
ope3   ----------------
ope4   ----------------
```

上記のような命令列があり，ope1 がフラグを更新する命令だとする．この場合，実際に GPR1 中のフラグレジスタに値反映されるのは ope3 の時点である．ope2, が GPR1 を読み出した場合，フラグレジスタの結果は不定である．