

ASIP Meister ユーザーズマニュアル

第 2.3 版
2008 年 10 月

エイシップ・ソリューションズ株式会社



©2008, ASIP Solutions, Inc.

ALL RIGHTS RESERVED

不許複製

本ドキュメントの一部または全部を許可無く複製することを禁じます。
複製する必要がある場合には、下記にお問合せ下さい。

〒541-0053 大阪市中央区本町 2-3-8 三甲大阪本町ビル 6F

エイシップ・ソリューションズ株式会社

support@asip-solutions.com

目次

はじめに	1
1. ASIP Meister マニュアル構成	1
2. 本マニュアルの内容と使い方	1
3. 表記上の約束事	1
I. ASIP Meister 概要	2
1. ASIP Meister とは	2
2. 設計データ入力フロー	3
II. ASIP Meister の操作方法	5
1. 基本操作	5
1.1 <i>ASIP Meister</i> の起動	5
1.2 メインウィンドウ	6
1.3 サブウィンドウ	8
1.4 ワーキングディレクトリ	8
1.5 コンパイラディレクトリ	8
2. Design Goal & Arch. Design	10
3. Resource Declaration	16
3.1 [<i>Resource Declaration</i>] ウィンドウのメニュー一覧	17
3.2 リソースの構成要素	18
4. Storage Spec	22
4.1 [<i>Register File</i>] タブ	22
4.2 [<i>Register</i>] タブ	24
4.3 [<i>Memory</i>] タブ	25
5. Interface Definition	26
6. Instruction Definition	29
6.1 命令タイプの定義: [<i>Instruction Type Definition</i>]	30
6.2 命令の定義: [<i>Instruction Declaration</i>]	36
6.3 割り込み(例外)の定義: [<i>Exception Declaration</i>]	37
7. Arch.Level Estimation	40
8. Assembler Generation	42
9. Micro Op. Description	44
9.1 [<i>Micro Op. Description</i>] ウィンドウ(命令動作定義)	45
9.2 [<i>Micro Op. Description</i>] ウィンドウ(割り込み動作定義)	46
9.3 [<i>Micro Op. Description</i>] ウィンドウ(マクロ定義)	47
9.4 [<i>Macro Expansion</i>]: マクロ展開の確認	49
10. HDL Generation	49
11. C Definition	53
11.1 [<i>Data Type</i>] タブ	53
11.2 [<i>Ckf Prototype</i>] タブ	54
12. Compiler Generation	58
付録	60
1. 設計可能なプロセッサモデル	60
1.1 制限事項	60
2. 使用上の制限	61

ASIP Meister ユーザーズマニュアル

3. マイクロ動作記述の記述方法	61
3.1 マイクロ動作記述の基礎	61
3.2 マイクロ動作記述の構文要素	62
3.3 マイクロ動作を記述する際の注意点	64
3.4 マイクロ動作記述の BNF	64
4. 予約語	65
4.1 ASIP Meister 設計ファイルの予約語	66
4.2 VHDL の予約語	66
5. 外部インタフェース仕様	67
5.1 mifu を命令メモリアクセスユニットとして使用する場合	67
5.2 mifu をデータメモリアクセスユニットとして使用する場合	67
6. 割込み記述方法	69
6.1 想定割込みモデル	69
6.2 内部割込みの概要	70
6.3 外部割込みの概要	71
6.4 NMI 割込みの概要	72
6.5 割込み仕様入力手順	73
6.6 割込みの入力	73
6.7 割込みハンドラへのジャンプ、割込み受理通知信号のアサーション	76
6.8 割込み処理における割込み発生時における PC の値の取得方法	78
7. PAS - メタアセンブラユーザーズマニュアル	79
7.1 PAS の概要	80
7.2 PAS の起動方法	80
7.3 PAS の実行例	80
7.4 アセンブル規則記述ファイル	82
7.5 アセンブリ言語の文法	87
7.6 アドレッシングモード	90
7.7 アセンブラ制御命令	91
8. 登録されているリソースモデルとパラメータ	100
8.1 adder	100
8.2 alu, mini_alu	100
8.3 barrelshifter	111
8.4 divider	112
8.5 extender	113
8.6 multiplexor	113
8.7 multiplier	114
8.8 shifter	115
8.9 register	116
8.10 registerfile	117
8.11 fwu	118
8.12 mifu	120
8.13 pcu	121
8.14 wire_in	122
8.15 wire_out	122
8.16 wire_inout	123
9. 応用プログラム開発環境として生成されるファイルとその使い方	124
9.1 出力されるプログラム一式	124
9.2 出力されるプログラムの主な使用法	124

図表目次

図 1 ASIP Meister の入出力.....	2
図 2 メインウィンドウ全体図.....	3
図 3 setenv による設定	5
図 4 [ASIP Meister Main Menu]ウィンドウ.....	6
図 5 メインウィンドウのファイルメニュー.....	7
図 6 メインウィンドウのヘルプメニュー.....	7
図 7 サブウィンドウの[Complete]ボタン	8
図 8 [Complete]ボタンのバリエーション	8
図 9 [Design Goal & Arch. Design]ウィンドウ.....	10
図 10 [Change stage number Confirm]ウインドウ	12
図 11 パイプラインステージの属性選択ウィンドウ	13
図 12 [Resource Declaration]ウィンドウ	16
図 13 リソースを追加した状態の[Resource Declaration]ウィンドウ.....	17
図 14 FHM Browser	20
図 15 [Port Set]: リソースモデルの入出力インターフェース	21
図 16: レジスタ・ファイル定義.....	22
図 17: 拡大されたストレージ一覧.....	24
図 18: レジスタ定義.....	25
図 19: メモリ定義	25
図 20 [Interface Definition]ウィンドウ.....	26
図 21 [Instruction Definition]ウィンドウ	29
図 22 [New Instruction Type Confirm]ウィンドウ	33
図 23 [Instruction Type Definition]ウィンドウ	34
図 24 [Instruction Type Definition]ウィンドウ	35
図 25 命令名入力ウィンドウ	36
図 26 新命令定義	36
図 27 見積もりシステム選択ウィンドウ	40
図 28 見積もり結果ウィンドウ.....	40
図 29 [Generation Confirm]ウィンドウ.....	42
図 30 メタアセンブラ用記述記述生成完了ウィンドウ	43
図 31 [Micro Op. Description]ウィンドウ	44
図 32 [Micro Op. Description]ウィンドウ	46
図 33 [Micro Op. Description]ウィンドウ	47
図 34 [New Macro Confirm]ウィンドウ.....	48
図 35 [Micro Op. Description]ウィンドウ	49
図 36 [Generation Confirm]ウィンドウ	50
図 37 [Generation Confirm]ウィンドウ	50
図 38 HDL 記述生成完了ウィンドウ	51
図 39 (例)シミュレーションモデルが生成されたディレクトリのファイル一覧.....	52
図 40 (例)論理合成モデルが生成されたディレクトリのファイル一覧.....	52
図 41 [C Definition]ウインドウ	53
図 42 [Ckf Prototype]タブ	55
図 43 [New CKF]ウインドウ	55
図 44 [Generation Confirm]ウインドウ	58
図 45 割込みコントローラとプロセッサの接続関係.....	70
図 46 内部割込みのタイミングチャート	71

図 47 外部割込みのタイミングチャート	72
図 48 NMI 割込みのタイミングチャート	72

ASIP Meister ユーザーズマニュアル

表 1 ASIP Meister マニュアル一覧	1
表 2 [ASIP Meister Main Menu]ウィンドウのメニュー一覧	7
表 3 [Design Goal & Arch. Design]ウィンドウの項目一覧	14
表 4 [Resource Declaration]のメニュー一覧	17
表 5 Use as で選択できる用途の一覧	19
表 6 属性一覧	19
表 7: Storage Spec: Register File	22
表 8: ストレージ使用方法	23
表 9 外部ポート属性一覧表	27
表 10 [Interface Definition]で設定する項目の一覧	28
表 11 [Field Attr]パラメータ選択条件	30
表 12: Addressing Mode	31
表 13 [Instruction Type Definition]パラメータ	32
表 14 [Instruction Declaration]パラメータ	36
表 15 割込み(例外)定義パラメータ	38
表 16 [Micro Op. Description]ウィンドウ	48
表 17 DataType 一覧	54
表 18 CKF で選択する情報	55
表 19 割込みの優先順位	69
表 20 割込み終了命令タイプの例	76
表 21 割込み終了命令の例	76

はじめに

1. ASIP Meister マニュアル構成

ASIP Meister に含まれるドキュメントを表 1 に示します。表 1 では **ASIP Meister** をインストールしたディレクトリからの相対パスを示しています。付属の Install Guide に従ってインストールした場合、**ASIP Meister** は、`"/usr/local/ASIPmeister/"` にインストールされます。したがって、チュートリアルファイルは、

`"/usr/local/ASIPmeister/share/tutorial/AM_tutorial_ja.pdf"`

にあることになります。本マニュアルを補完するものとしてこれらのドキュメントを参照してください。

表1 ASIP Meister マニュアル一覧

ドキュメント名 & ファイル名	記載内容
ASIP Meister ユーザーズマニュアル <code>share/AM_usersmanual_ja.pdf</code>	本書
インストールガイド <code>InstallGuide.txt</code>	ASIP Meister の動作環境要件 ASIP Meister のインストールとセットアップ方法
ASIP Meister チュートリアル <code>share/tutorial/AM_tutorial_ja.pdf</code>	「プロセッサ small_RISC」題材によるチュートリアルとコンパイラ生成についてのチュートリアル。 ASIP Meister を一通り体験することができます。
FLEXnet ライセンス ユーザーガイド <code>share/LicensingEndUserGuide.pdf</code>	ライセンスサーバの設定方法など、ライセンスについての詳細が必要な際に参照してください。

2. 本マニュアルの内容と使い方

本マニュアルは、**ASIP Meister** を初めてお使いになる方、あるいは、**ASIP Meister** の特定の機能について確認したい方などを対象に記述しています。

本マニュアルは【**ASIP Meister** 概要】【**ASIP Meister** の操作方法】【付録】で構成されます。必要に応じてお読みください。

3. 表記上の約束事

本マニュアルの表記上の約束事を説明します。

- 文中、`[]` で囲んだ名称はメニュー、ボタン、テキストボックス、チェックボックス、パラメータなどの名称を表しています。
- `$` はコマンドプロンプトを表しています。
- 文中、`"` で囲んだ名称はファイル、ディレクトリなどの名称を表しています。
- <注意> は、注意事項・制限事項が書いてありますので、必ずお読みください。
- <ポイント> は、知っておくと便利な情報が書かれています。
- <関連> は、関連する説明事項をリストアップしています。

I. ASIP Meister 概要

1. ASIP Meister とは

ASIP Meister は、設計の上流工程における特定用途向きプロセッサ(ASIP)開発ツールです。次のような機能を備えています。

- マイクロ動作記述からプロセッサの HDL 記述を自動生成
- 上流工程における、短時間での設計品質見積もり

これらの機能により上流工程での設計短 TAT 化が図れ、**ASIP Meister** に使用により、多数の実装方法の比較検討が可能になります。

ASIP Meister の入出力を図 1 に示します。ユーザは **ASIP Meister** の GUI を使って、設計パラメータとマイクロ動作記述を入力します。**ASIP Meister** は設計品質見積もりレポートと RTL 設計記述ファイルを出力します。また、C コンパイラ、アセンブラ、リンカ等の応用プログラム開発環境を生成します。RTL 設計記述は HDL で記述され、論理合成モデルとシミュレーションモデルが自動生成されます。これらの HDL ファイルは、論理合成ツールと機能シミュレーションツールの入力となります。

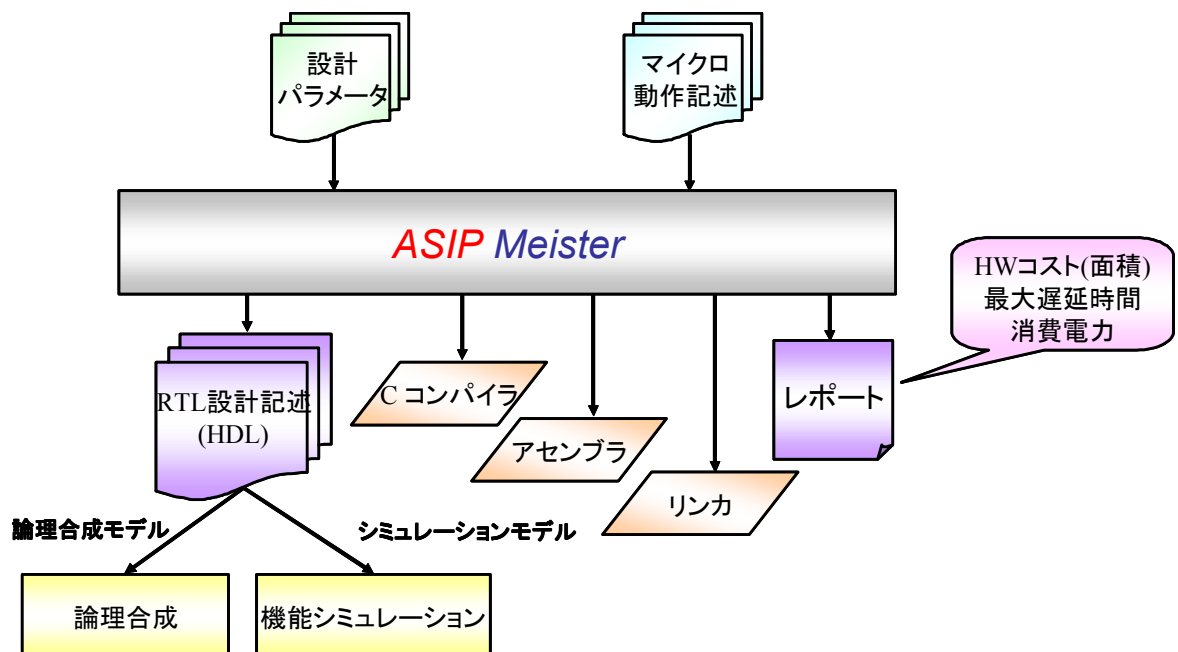


図1 ASIP Meister の入出力

2. 設計データ入力フロー

ASIP Meister を起動すると図 2 に示すメインウィンドウが開きます。

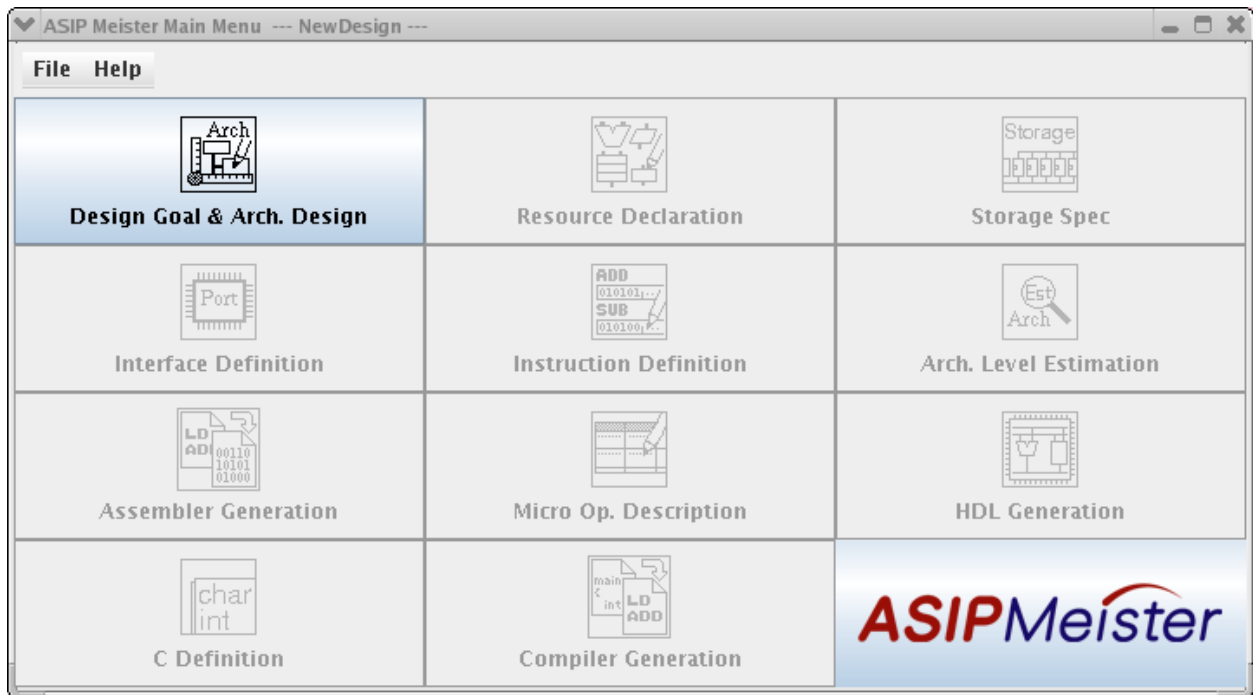


図2 メインウィンドウ全体図

ASIP Meister では、ひとつのプロセッサを、以下の 11 個の項目に分割して設計します。

[Design Goal & Arch. Design]:	プロセッサスペックの設定
[Resource Declaration]:	プロセッサに使用する演算器やレジスタなどの宣言
[Storage Spec]:	ストレージの仕様
[Interface Definition]:	プロセッサのインターフェースの設計
[Instruction Definition]:	命令形式及び命令コードの定義
[Arch. Level Estimation]:	プロセッサの設計品質見積もり
[Assembler Generation]:	メタアセンブラ用ファイルの生成
[Micro Op. Description]:	命令のマイクロ動作の設計
[HDL Generation]:	HDL 記述の生成
[C Definition]:	コンパイラが対応していない拡張命令に対する C 記述の定義
[Compiler Generation]:	コンパイラ, Binutils の生成

各項目は、メインウィンドウの 11 個の大きな四角の部分に対応しています。クリックすることにより、サブウィンドウが開き、プロセッサの各部分の設計をすることができます。

各項目間には依存関係があり、依存のある項目でのデータ入力に先に終了している必要があります。項目間の依存関係を下に示します。

[Resource Declaration], [Instruction Definition] : [Design Goal & Arch Design]終了後に入力可能

[Arch. level Estimation], [Assembler Generation], [C Definition]: [Design Goal & Arch Design], [Resource Declaration], [Interface Definition], [Instruction Definition]終了後に入力可能



[Micro Op. Description] : [Arch. Level Estimation]終了後に入力可能
[HDL Generation] : [Micro Op. Description]終了後に実行可能
[Compiler Generation] : [C Definition]終了後に実行可能

II. ASIP Meister の操作方法

1. 基本操作

ここでは、**ASIP Meister** の基本的な操作方法について説明します。

1.1 ASIP Meister の起動

ASIP Meister を使用するには正規のソフトウェアライセンスが必要です。インストールガイドに従いあらかじめライセンスファイルをインストールまたはライセンスサーバの起動を行ってください。**ASIP Meister** 起動に際し、環境変数 `LM_LICENSE_FILE` にライセンスファイルのパスまたはライセンスサーバ名を設定されていることを確認してください。ライセンスサーバの起動方法はインストールガイドを参照してください。また、コンパイラ生成機能を使用する場合は、インストールの際に作成した応用プログラム開發生成ツールのディレクトリ場所を環境変数 `ASIP_APDEV_SRCROOT` に設定してください。

```
$ export LM_LICENSE_FILE=/usr/local/ASIPmeister/ASIPmesiter.lic
$ export ASIP_APDEV_SRCROOT=/home/xxx/ASIP_APs (応用プログラム開發生成ツールの場所)
```

ASIP Meister を起動するにはJava™ 2環境が必要です。Java™ 2のインストールに関しては、Java™ のホームページを参照してください。ここではJava™ 2が正常にインストールされているとします。

ASIP Meister を起動する際に、**ASIP Meister** とJava™2の実行環境へのパスを指定する必要があります。設定用のサンプルファイル ("`ASIPmeister.setup.sample`") を `/usr/local/ASIPmeister/share/` ディレクトリに用意しています。このサンプルファイルはJava™ 2の実行環境が `/usr/java/jre1.6.0_05/` にインストールされていることを想定していますので、自分の環境に合わせて、適宜、編集して使用してください。

```
$ source ASIPmeister.setup.sample
```

またこのサンプルファイルは `bash` シェルを仮定して書かれていますので、`csh` や `tcsh` をお使いの方は `setenv` による設定に変更してください。

```
% setenv PATH /usr/java/jre1.6.0_05/bin/:$PATH
% setenv PATH /usr/local/ASIPmeister/bin/:$PATH
```

図3 `setenv` による設定

ASIP Meister の起動コマンドは `ASIPmeister` です。引数として入力データファイルを指定することができます。

```
$ ASIPmeister [input_data_file.pdb]
```

<ポイント> "`.pdb`" は **ASIP Meister** 入力データファイルの拡張子です。

メモリポートの定義を含むデフォルト設計データ

ASIP Meister ではプロセッサと命令メモリやデータメモリを接続するポートを定義する必要があります。

ります。ポートにはアドレスバスやデータバスなどといった標準的なものがありますが、メモリへのアクセス方法や書き込みの有無により別途必要になるポートもあります。**ASIP Meister** では“share/basefile”ディレクトリに次のメモリ構成について設計データを用意しています。

設計ファイル名	命令メモリへのアクセス方法	データメモリへのアクセス方法
InstSingle-DataSingle.pdb	シングルサイクルアクセス	シングルサイクルアクセス
InstSingle-DataMulti.pdb	シングルサイクルアクセス	マルチサイクルアクセス
InstMulti-DataSingle.pdb	マルチサイクルアクセス	シングルサイクルアクセス
InstMulti-DataMulti.pdb	マルチサイクルアクセス	マルチサイクルアクセス

1.2 メインウィンドウ

ASIP Meister を起動すると、図 4に示す[ASIP Meister Main Menu]ウィンドウが開きます。

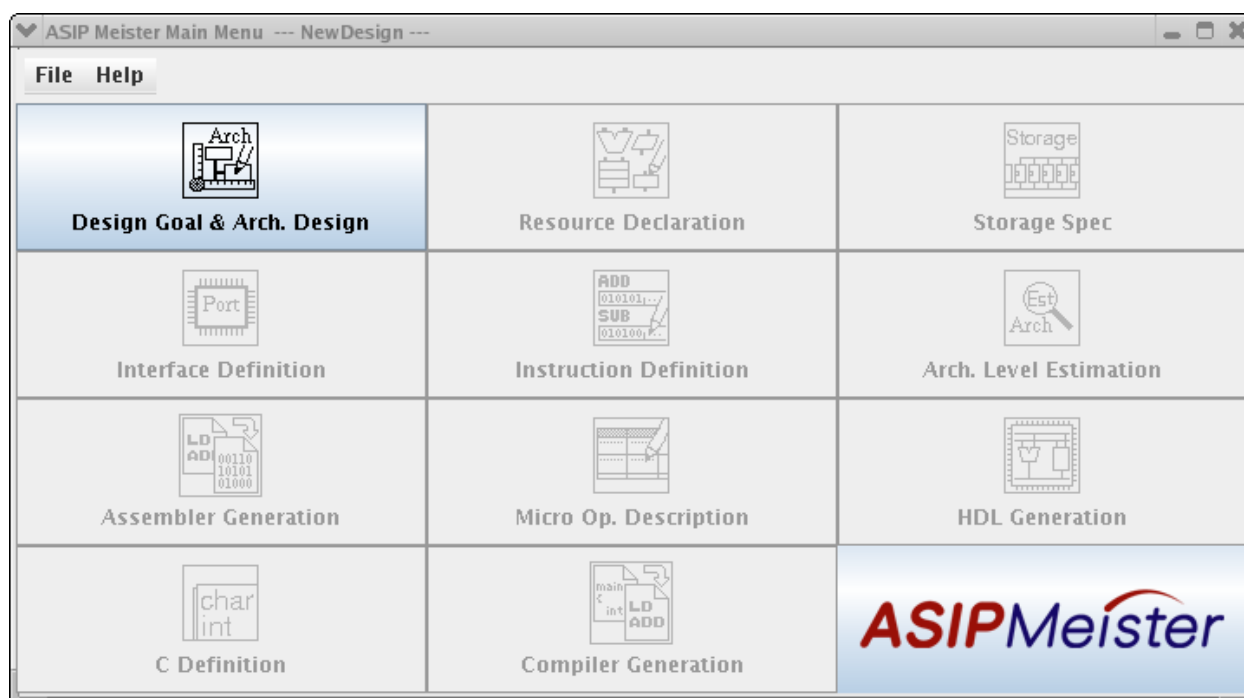


図4 [ASIP Meister Main Menu]ウィンドウ

メインウィンドウの各フローでデータを入力することで、プロセッサを設計していきます。各ステップでの入力方法などについては2. Design Goal & Arch. Design以降で説明します。

メインウィンドウのタイトルバーには設計中のデータファイルの名称が表示されます。新規設計の場合は“--- NewDesign ---”と表示されます。

次に、メインウィンドウの各メニューについて説明します。

(1) [File]メニュー

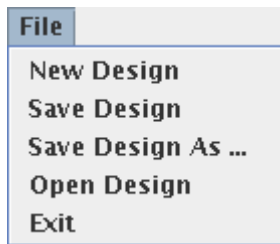


図5 メインウインドウのファイルメニュー

[New Design]

機能: 新規に設計を開始します。

[Save Design]

機能: 設計中のデータを上書き保存します。

[Save Design As ...]

機能: 設計中のデータを別名で保存します。

この項目を選択すると、データファイル名入力ウィンドウが開きます。まず[Save in]の▼プルダウンメニューとウィンドウからデータファイルを保存するディレクトリを選択してください。[File name]テキストボックスにデータファイル名を入力して、[Save]ボタンをクリックすることでデータファイルが保存されます。このとき指定したファイル名に拡張子".pdb"が付加されていない場合は、自動的に".pdb"が付加されます。

<注意> サブウインドウでの作業中に設計データを保存したい場合は、サブウインドウの[File]メニューから[Commit]メニューを選択し、[Save Design]または[Save Design As ...]を選択してください。[Commit]メニューを選択しなければ、サブウインドウを開いてからの変更が反映されません。

[Open Design]

機能: 保存されている設計データを読み込みます。

この項目を選択すると、入力データファイルの選択ウィンドウが開きます。ここで入力したいデータファイルを選択し、[Open]ボタンをクリックしてください。

[Exit]

機能: **ASIP Meister** を終了します。

(2) [Help]メニュー



図6 メインウインドウのヘルプメニュー

[Version]

機能: **ASIP Meister** のバージョン情報を表示します。

表2 [ASIP Meister Main Menu]ウィンドウのメニュー一覧

メニュー	サブメニュー	機能
File	New Design	新規設計の開始

メニュー	サブメニュー	機能
	Save Design	設計データの上書き保存
	Save Design As ...	設計データの別名保存
	Open Design	設計データを開く
	Exit	ASIP Meister の終了
Help	Version	バージョンの表示

1.3 サブウィンドウ

各サブウィンドウはメインウィンドウにおいてそれぞれの項目を選択することで表示されます。各サブウィンドウでの入力方法などは2. Design Goal & Arch. Design以降で説明します。サブウィンドウを閉じる場合は、[File]メニューから[Close]メニューを選択してください。作業途中に保存する場合は、[File]メニューから[Commit]メニューを選択し、[Save Design]または[Save Design As ...]を選択してください。全ての項目設定作業が終了している場合は、サブウィンドウを閉じる前に[Complete]ボタンをチェックしてください。(図 7)

[Complete]ボックスをチェックするとサブウィンドウでのデータ入力が完了となり、依存するサブウィンドウでのデータ入力に進むことができます。[Complete]ボックスをチェックしていない場合は依存するサブウィンドウのデータ入力に進むことが出来ません。

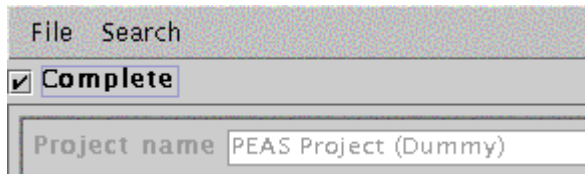


図7 サブウィンドウの[Complete]ボタン

- ☐ **Complete**: 非完了
- ☒ **Complete**: 完了
- ☐ **Complete**: 非完了 (READ ONLY)
- ☒ **Complete**: 完了 (表示のみ)

図8 [Complete]ボタンのバリエーション

1.4 ワーキングディレクトリ

ASIP Meister はカレントディレクトリの“meister”というワーキングディレクトリに、**ASIP Meister** のワークファイル及び自動生成された HDL 記述ファイルやメタアセンブラ用アセンブル規則記述ファイルを格納します。カレントディレクトリに“meister”ディレクトリが存在しない場合は自動的に作成します。

< 関連 > 10. HDL Generation

1.5 コンパイラディレクトリ

ASIP Meister の Standard 版では、コンパイラや Binutils を生成できます。その際、生成するディレクトリを指定することができます。環境変数として、ASIP_GNU_INSTALL_DIR を設定すると、コンパイラや Binutils のファイルが、ASIP_GNU_INSTALL_DIR で指定したフォルダに生成され

ます。環境変数として `ASIP_GNU_INSTALL_DIR` を設定しなければ、ワーキングディレクトリ `"meister/swgen"` 以下に生成されます。

2. Design Goal & Arch. Design

メインウィンドウから[Design Goal & Arch. Design]をクリックすると、図 9のサブウィンドウが開きます。このサブウィンドウでは、プロセッサのタイプやアーキテクチャパラメータを設定します。

The screenshot shows the 'Design Goal & Arch. Design' window with the following fields and options:

- Project name:** ASIP Meister Project
- Fhm workname:** FHM_work
- Revision No.:** ver 1.0 : ASIP Meister Project
- Design Goal:**
 - Goal Area: [] [gates]
 - Goal Delay: [] [ns]
 - Goal Power S: [] [uW/MHz]
- Design Priority:** ☒ Area ☐ Performance ☐ Power
- CPU type:** ☒ Pipeline
- Pipeline:**
 - Num. of Stages: 5 [Apply]
 - Num. of Common Stages: 0
 - Decode Stage: 2 [-th]
 - stage table:

stage	IF	ID	EXE	MEM	WB
1	IF	ID	EXE	MEM	WB
2	attribute	attribute	attribute	attribute	attribute
3	fetch	decode	exec	memory_read & memory_write	register_write
4	Up	Down	Up	Down	Up
5	Down	Up	Down	Up	Down
- Multi cycle interlock:** ☒ Yes ☐ No
- Data hazard interlock:** ☐ Yes ☒ No
- Register bypass:** ☐ Yes ☒ No
- Delayed branch:** ☒ Yes ☐ No
- Yes:** Num. of delayed slot: 1 [instruction]
- Max inst. bit width:** 32 [bit]
- Max data bit width:** 32 [bit]
- Processor design:** ☒ New Design ☐ New Design with Base Processor ☐ Base Processor Design
- Use compiler:** ☐ Yes ☒ No

図9 [Design Goal & Arch. Design]ウィンドウ

このウィンドウでは、プロセッサの設計データの管理情報(プロジェクト名、FHM データベース名、リビジョン情報)、設計目標(面積、実行速度、消費電力)、設計するプロセッサのタイプ、パイプライ

ンの概要(ステージ数、ステージの機能)、命令及びデータのビット幅、プロセッサの設計方針、コンパイラの使用決定などを入力して、設計するプロセッサの概要を決定します。

[Design Goal & Arch. Design]ウィンドウで入力した[Design Goal]に関する項目([Goal Area]、[Goal Delay]、[Goal Power S]の値)は、設計目標値として使われ、[Arch. Level Estimation]において、設計中のプロセッサの面積、遅延、消費電力の見積もり値との比較に使われます。

<関連>7. Arch.Level Estimation

[Project name]: プロジェクト名

設計データにつけるプロジェクト名を入力します。ここでは、任意の文字列が使用でき、他に設計中のプロセッサとの混同を防ぐ目的で使用できます。

<注意>ここで指定したプロジェクト名は生成されるプロセッサのエンティティ名ではありません。プロセッサのエンティティ名は[Interface Definition]ウィンドウで入力します。(5. Interface Definition参照)

[Fhm workname]: 使用するデータベース名

ASIP Meister が使用する FHM のリソースモデルのデータベース名を入力します。

<注意>現バージョンでは”FHM_work”が標準で提供されており、”FHM_work”のみが使用可能となっています。

[Revision No.]: リビジョン情報

設計データのリビジョン情報を入力します。この項目は任意の文字列が使用できます。設計データのリビジョン管理にお使いください。

[Design Goal]: 設計の目標値

プロセッサの設計目標を設定します。**ASIP Meister** では、プロセッサ全体の面積、データパス(組合せ回路部)の遅延時間、消費電力の設計目標値をそれぞれ[Goal Area]、[Goal Delay]、[Goal Power S]に入力します。ここで入力した設計目標値は、[Arch. Level Estimation]において、設計中のプロセッサの面積、遅延、消費電力の見積もり値との比較に使われます(7. Arch.Level Estimation参照)。

[Design Priority]: 優先設計目標

プロセッサの設計目標のうち面積、遅延時間、消費電力のどの設計目標を優先するかを指定します。[Design Priority]で”Area”を指定した場合、[Arch. Level Estimation]で、生成された HDL 記述から面積優先で論理合成した場合の面積、遅延、消費電力を予測します。”Performance”や”Power”を選択した場合も同様に、遅延および消費電力を優先して論理合成した場合の面積、遅延、消費電力を予測します。

[CPU type]: 設計するプロセッサのアーキテクチャタイプ

”CPU type”では、設計するプロセッサのアーキテクチャを選択します。現バージョンでサポートしている CPU アーキテクチャは”Pipeline”のみとなっています。

[Num. of Stages]: パイプラインのステージ数

パイプラインのステージ数を入力します。ステージ数を入力後、右にある[Apply]ボタンを押すと、下にある[stage]の部分のステージパラメータ入力行の行数が、[Num. of Stages]に入力した値に変更されます。

<注意 1>ステージ数を増やした場合は、ステージ名 **TMP** という行が追加されます。また、ステージ数を減らした場合は、後段のステージから削除されます。ステージ数を減らした後に、[File]メニューの[Commit]メニューを選択すると、図 10に示すメッセージウィンドウが表示されます。

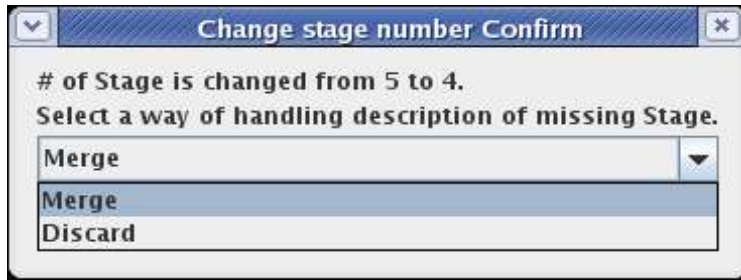


図 10 [Change stage number Confirm] ウィンドウ

このウィンドウでは、削除されるステージのマイクロ動作記述をどのように扱うかを選択します。“Merge”を選択した後、“Select”を選択すると、変更後の最終ステージに削除されたステージのマイクロ動作記述が追加されます。一方、“Discard”を選択した後、“Select”を選択すると、削除されたステージのマイクロ動作記述は破棄されます。

<注意 2> ステージ数を変更すると、[File]メニューから[Commit]メニューを選択するまで、後述する“Up”, “Down”ボタンを使用してのステージの入れ替えを行うことは出来ません。

[Num. of Common Stages]: パイプラインにおける共通ステージ数

共通ステージ数を入力します。共通ステージとは、すべての命令においてマイクロ動作が同じステージを意味します。ここでは、ステージ 1 から連続する共通ステージ数を入力します。ただし、ステージ 1 が共通ステージでない場合は、共通ステージ数は 0 となります。現バージョンでは 0 に固定されています。

[Decode Stage]: デコードステージ

プロセッサがフェッチした命令をデコードするステージを入力します。

[Stage]: ステージ情報行

[stage]には、パイプラインの各ステージのパラメータを入力するパラメータ入力行が[Num. Of Stages]で指定したステージ数と同数表示されています。一番上の行が第 1 ステージに対応しており、一番下の行が最終ステージに対応しています。各行の先頭には、ステージ番号が表示されており、その右側のボックスにステージ名を入力します。

各ステージの属性を、[attribute]ボタンをクリックすることにより表示されるウィンドウ(図 11)から選択します。ステージに持たせたい属性に該当するボックスをチェックし、[OK]ボタンを押しウィンドウを閉じます。すると[attribute]ボタンの右側に、先ほど選択した属性が表示されます。1 ステージで複数の属性を選択することができます。選択できる属性は以下の 7 種類です。

- [fetch]: 命令フェッチステージ
- [decode]: 命令デコードステージ
- [register_read]: レジスタ読み込みステージ
- [exec]: 演算ステージ
- [memory_read]: メモリ読み込みステージ
- [memory_write]: メモリ書き込みステージ
- [register_write]: レジスタ書き込みステージ

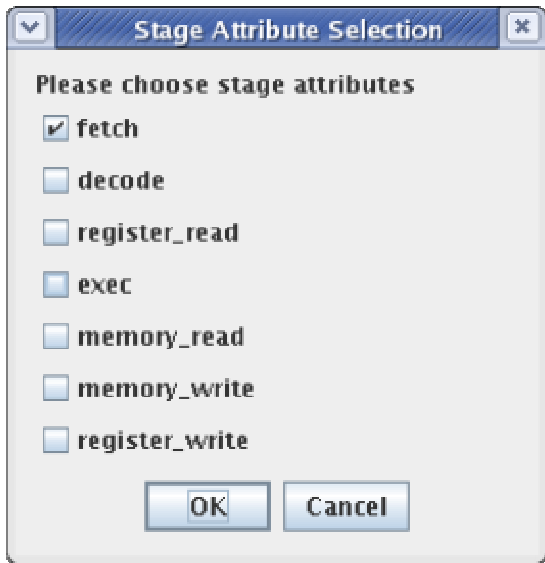


図11 パイプラインステージの属性選択ウィンドウ

また、“Up”，“Down”ボタンを使用することで、ステージ名とその属性を1つ上のステージや1つ下のステージのステージ名や属性とスワップできます。

＜注意＞“Up”，“Down”を選択し、ステージ名やその属性をスワップした場合、[File]メニューの[Commit]を選択するまで、ステージ数(Num. of Stages)を変更することができません。

[Multi cycle interlock]: マルチサイクル演算時のインタロックの有無

マルチサイクル演算を行うときに、後続命令をストールさせるかどうかを選択します。マルチサイクル演算時に後続命令をストールさせる場合は“Yes”を選択してください。

＜注意＞現在のバージョンでは“Yes”に固定です。

[Data hazard interlock]: データハザード時のインタロックの有無

データハザードが発生したときに、後続命令をストールさせるかどうかを選択します。データハザード時に、後続命令をストールさせる場合は“Yes”を選択してください。

＜注意＞現在のバージョンでは“No”に固定です。

[Register bypass]: レジスタバイパスの有無

レジスタのバイパス(レジスタフォワーディング)をするかどうかを選択します。レジスタバイパスをする場合は、“Yes”を選択してください。

＜注意＞現在のバージョンでは“No”に固定です。

[Delayed branch]: 遅延分岐の有無

遅延分岐を使用するか、しないかを選択します。遅延分岐を使う場合は、[Yes]を選択してください。

[Num. of delayed slot]: 遅延分岐スロット数

遅延分岐スロット数を入力します。[Num. of delayed slot] 入力フィールドは[Delayed branch]で[Yes]を選択した場合に開きます。

遅延分岐スロット数0とは、遅延分岐しない場合と同じとなります。

[Max inst. bit width]: 命令最大ビット幅

命令の最大ビット幅を入力します。

[Max data bit width]: データ最大ビット幅
データの最大ビット幅を入力します。

[Processor design]: プロセッサの設計方針

プロセッサの設計方針を選択します。3つの項目から1つを選択します。

1. **[New Design]:** 新規にプロセッサを設計する方針です。
2. **[New Design with Base Processor]:** ベースプロセッサを基本とし、ベースプロセッサに対して拡張する設計方針です。そのため、**2.[New Design with Base Processor]**を選択する場合は、ベースプロセッサの設計ファイルを開いた状態で選択してください。
3. **[Base Processor Design]:** 新規にベースプロセッサを設計する方針です。

<補足> 1. **[New Design]**と 3.**[Base Processor Design]**の相違点は、設計したプロセッサを 2.**[New Design with Base Processor]**で選択するベースプロセッサとして、使用する際に表れます。1. **[New Design]**で設計したプロセッサは、プロセッサの命令の変更や追加ができますが、3.**[Base Processor Design]**で設計したプロセッサの命令の変更はできず追加のみとなり、変更するとその設計データファイルは上書きできなくなります。

<注意 1>プロセッサの設計方針に基づいて、5. Interface Definition、9. Micro Op. Descriptionフェーズでの制限が変わります。それぞれの制限については、対象のフェーズで説明します。

<注意 2>プロセッサの設計方針を 3.**[Base Processor Design]** から 2.**[New Design with Base Processor]**に変更した場合、その時点の設計ファイルをベースプロセッサ登録するか確認ウィンドウが表示されます。**[Yes]**を選択すると、ベースプロセッサとして登録します。また 2.**[New Design with Base Processor]**から 1.**[New Design]**に変更した際は、ベースプロセッサの情報を失うことを確認するウィンドウが表示されます。

[Use compiler]: コンパイラの使用

コンパイラの使用を選択します。”Yes”を選択した場合は、ベースプロセッサ **Brownie** を基本とし、拡張命令に対応したコンパイラを作成することを前提とします。”No”を選択すると、コンパイラを使用しないことを前提とした設計になり、**[C Definition]**フェーズと**[Compiler Generation]**フェーズは使用できません。

<補足> **[Processor Design]**の選択が**[New Design with Base Processor]**の際、**[Use Compiler]**を”no”から”yes”に変更すると、ベースプロセッサの命令を変更していないかどうかの確認メッセージが表示されます。

表3 **[Design Goal & Arch. Design]**ウィンドウの項目一覧

項目	意味
Project name	設計プロジェクト名
Fhm workname	使用するリソースデータベース名
Revision No.	設計データのリビジョン情報
Goal Area	プロセッサ全体の面積 単位は[gates]
Goal Delay	最大遅延時間 単位は[ns]
Goal Power S	静止時消費電力 単位は[uW/MHz]
Design Priority	優先目標
CPU Type	プロセッサのタイプ

項目	意味
Number of Stages	パイプライン段数
Number of Common Stages	共通ステージ数
Decode Stage	デコード命令のステージ番号
ステージ名	各ステージの名前
Attribute	ステージの属性
Multi cycle interlock	マルチサイクル演算時のパイプラインインタロックの有無
Data hazard interlock	データハザード時のパイプラインインタロックの有無
Register bypass	レジスタバイパスの有無
Delayed branch	遅延分岐の有効 / 無効
Num. of delayed slot	遅延分岐スロット数
Max inst. bit width	命令の最大ビット幅
Max data bit width	データの最大ビット幅
Processor design	プロセッサの設計方針
Use compiler	コンパイラの使用

3. Resource Declaration

[Resource Declaration]は、プロセッサで使用するリソースの宣言をします。メインウィンドウから[Resource Declaration]をクリックすると、図 12の[Resource Declaration]ウィンドウが開きます。

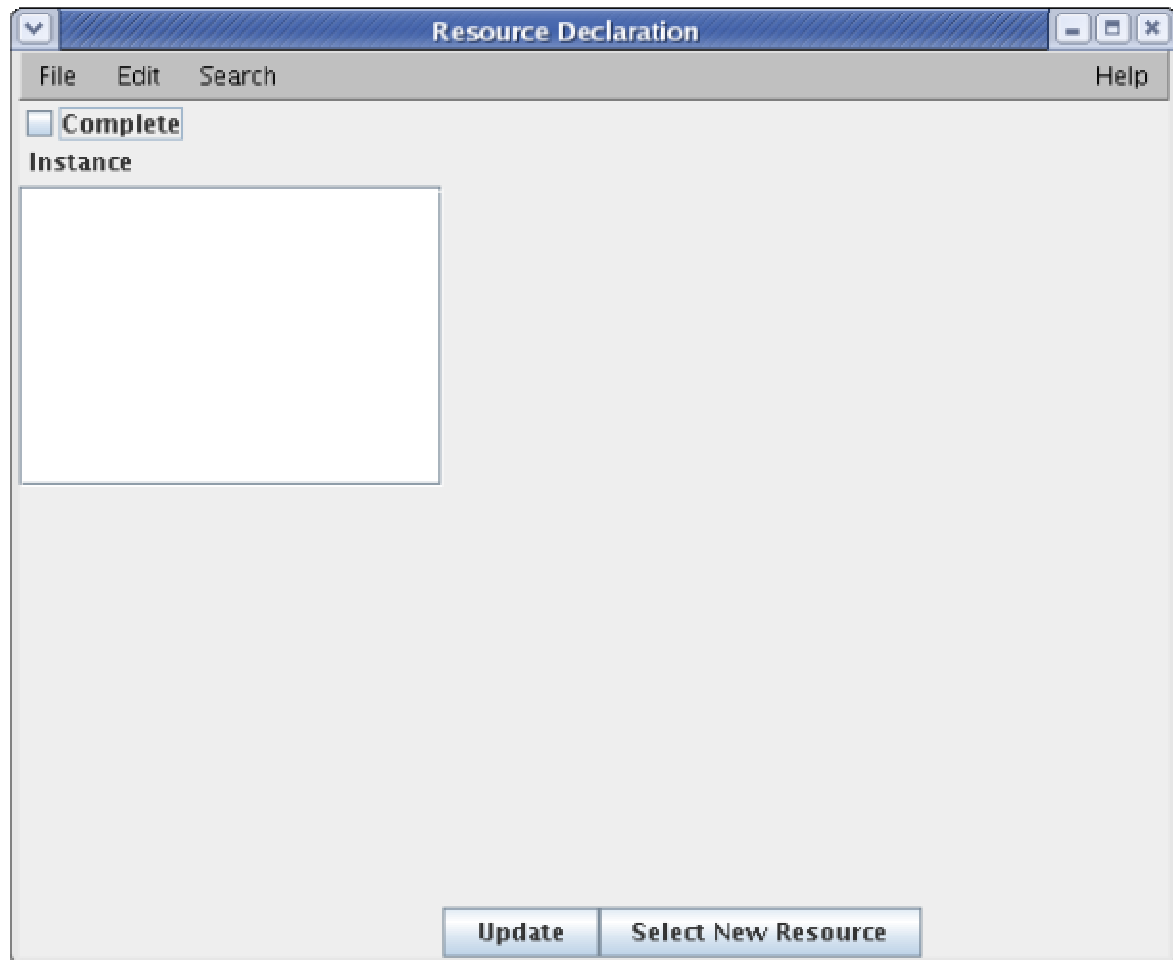


図12 [Resource Declaration]ウィンドウ

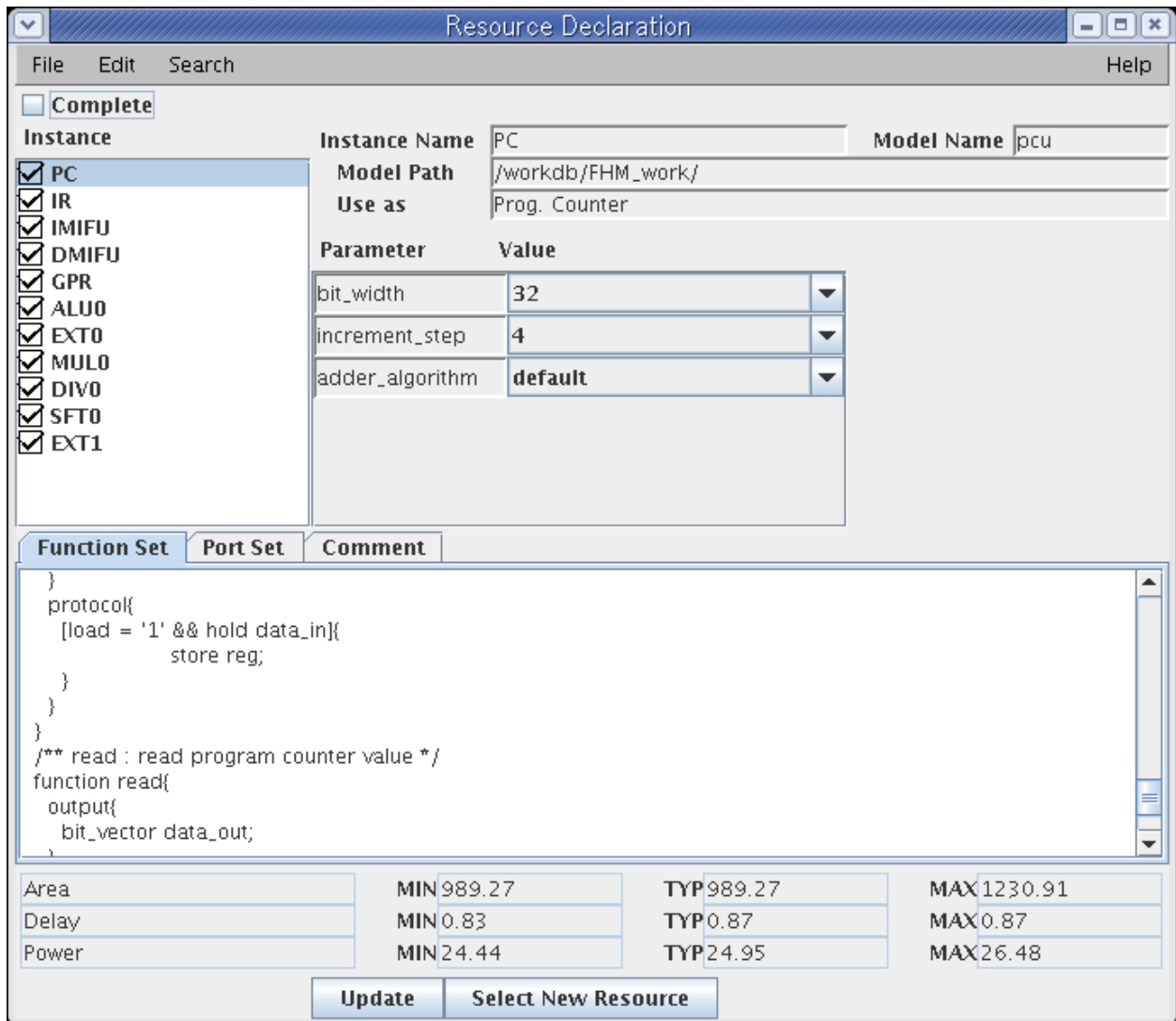


図13 リソースを追加した状態の[Resource Declaration]ウィンドウ

図 13に、[Resource Declaration]ウィンドウにおいて、リソースを追加した例を示します。

[Update]ボタンをクリックすることで、選択しているリソースの機能情報、見積もり値がウィンドウ下部に表示されます。

3.1 [Resource Declaration]ウィンドウのメニュー一覧

[Resource Declaration]ウィンドウにおけるメニュー一覧を表 4に示します。

表4 [Resource Declaration]のメニュー一覧

項目	動作
Edit	Edit メニュー
Insert	リソースの挿入場所の指定
Delete	リソースの削除
Copy	リソースのコピー

Move	リソースの移動
Mark	リソースの選択
Mark Clear	リソースの選択解除
Update	リソース情報の更新
Valid	リソースの使用を許可
Invalid	リソースの使用を許可しない
Search	Search メニュー
Instance	リソース名の検索

それぞれのメニュー項目について順に説明します。

3.1.1 Insert

リソースを追加する際の挿入場所を指定します。[Insert]のチェックボックスをオンにすると、リソースを追加する際に選択しているリソースの位置に、新規のリソースが追加されます。[Insert]のチェックボックスをオフにすると、[Instance]欄の最下位に新規のリソースが追加されます。

3.1.2 Delete

リソースを削除します。削除するリソースは、[Instance]欄で選択されているリソースです。削除する際に確認するウインドウは表示されないので、注意してください。

3.1.3 Copy

リソースをコピーします。コピーしたいリソースを選択し、[Copy]メニューを選択すると、新しいリソースの名前を入力するウインドウが表示されます。名前を入力し、"OK"を選択すると、入力した名前のリソースがコピーされ、リソース一覧に追加されます。

3.1.4 Move

リソースを移動します。[Instance]欄のリソース一覧の順番を変更する際に使用します。まず[Mark]メニューを選択し、移動したいリソースを選択します。そして、移動させたい場所のリソースを選択し、[Move]メニューを選択することで、リソースを移動できます。

3.1.5 Mark

リソースを選択します。[Move]メニューを使用する際に使用します。

3.1.6 Mark Clear

Mark したリソースを解除します。

3.1.7 Update

リソース情報を更新します。[Instance]欄で選択したリソースの見積もり値を更新します。

3.1.8 Valid

リソースの使用を宣言します。[Instance]欄で選択したリソースをプロセッサで使用するとします。

3.1.9 Invalid

リソースを使用しないようにします。[Instance]欄で選択したリソースに対して、[Invalid]メニューを選択すると、対象のリソースをプロセッサで使用しないとします。よって Invalid の状態のリソースは、生成されるプロセッサに含まれません。

3.1.10 Instance

リソースを検索します。[Search]メニューの[Instance]メニューを選択すると、[Resource Search]ウインドウが表示されます。検索したい文字列を入力し、"Search"ボタンを選択することで、リソースの名前を検索できます。

3.2 リソースの構成要素

[Resource Declaration]ウインドウの[Select New Resource]ボタンをクリックすると、図 14の使

用するリソースを選択するウィンドウ[FHM Browser]が表示されます。[FHM Browser]には登録されているリソースモデルの一覧が表示されます。このウィンドウに表示されている設定項目について説明します。

[Model]: リソースモデル一覧

使用可能なリソースモデルの一覧をツリー表示しています。ツリーは左側の丸印をクリックすることで表示を展開できます。リソースモデル一覧から使用したいリソースをクリックして選択することで、選択したリソースモデルに必要なパラメータ設定画面が表示されます。

<注意>使用可能なリソースモデルの種類は **ASIP Meister** のリリースバージョンにより変更される場合があります。

[Parameter],[Value]: リソースモデルのパラメータの項目と値

ビット幅やアルゴリズムなどの選択可能なパラメータはリソースモデルによって異なります。選択したリソースモデルに応じて選択可能なパラメータの項目が表示されますので、表示された全ての項目について[Value]からパラメータを設定してください。

[Use as]: リソースの使用用途

選択したリソースモデルをどのような用途に用いるかをプルダウンメニューから選択します。たとえばレジスタモデルをプログラムカウンタとして使用する場合は、[Prog. Counter]を選択します。表 5に用途の一覧を示します。

表5 Use as で選択できる用途の一覧

選択肢	意味
(unspecified)	以下のどの用途にも該当しない場合に設定
Inst. Register	命令レジスタ
Register File	レジスタファイル
Prog. Counter	プログラムカウンタ
Inst. Memory	命令メモリ
Data Memory	データメモリ
Plain Register	プレーンレジスタ
Mask Register	マスクレジスタ

<注意>1 プロセッサ内に[Inst. Register]、[Prog. Counter]、[Inst. Memory]及び[Data Memory]の用途で用いられるリソースは必ず1つでなければなりません。

[Function Set]: リソースモデルの機能

[Function Set]には、選択したリソースモデルの機能が表示されます。

[Function Set]をクリックすると、左の[Model]ウィンドウで選択されているリソースモデルの機能が表示されます。選択したリソースモデルが目的の機能を持っているかをここで確認します。

[Port Set]: リソースモデルの入出力インターフェース

リソースモデルのポート名、入出力の方向、データタイプ、ビット幅、及び属性を表示します。表 6に属性一覧を示します。

表6 属性一覧

属性	意味
Clock	クロック
Ctrl	制御信号
Reset	リセット信号
Data	データ

属性	意味
Mode	モード信号

[Estimate]: リソースの概略見積もり

[Estimate]をクリックすると、ビット幅などの所定のパラメータを設定した状態での面積、遅延、消費電力を見積もり、それぞれ[Area]、[Delay]及び[Power]について、最小値[MIN]、標準値[TYP]、及び最大値[MAX]の3種類の値が表示されます。

[Instantiate]: リソースの追加

[Instantiate]をクリックすると、リソース名を入力するウィンドウが開きます。開いた入力ボックスにリソースの名前を入力します。

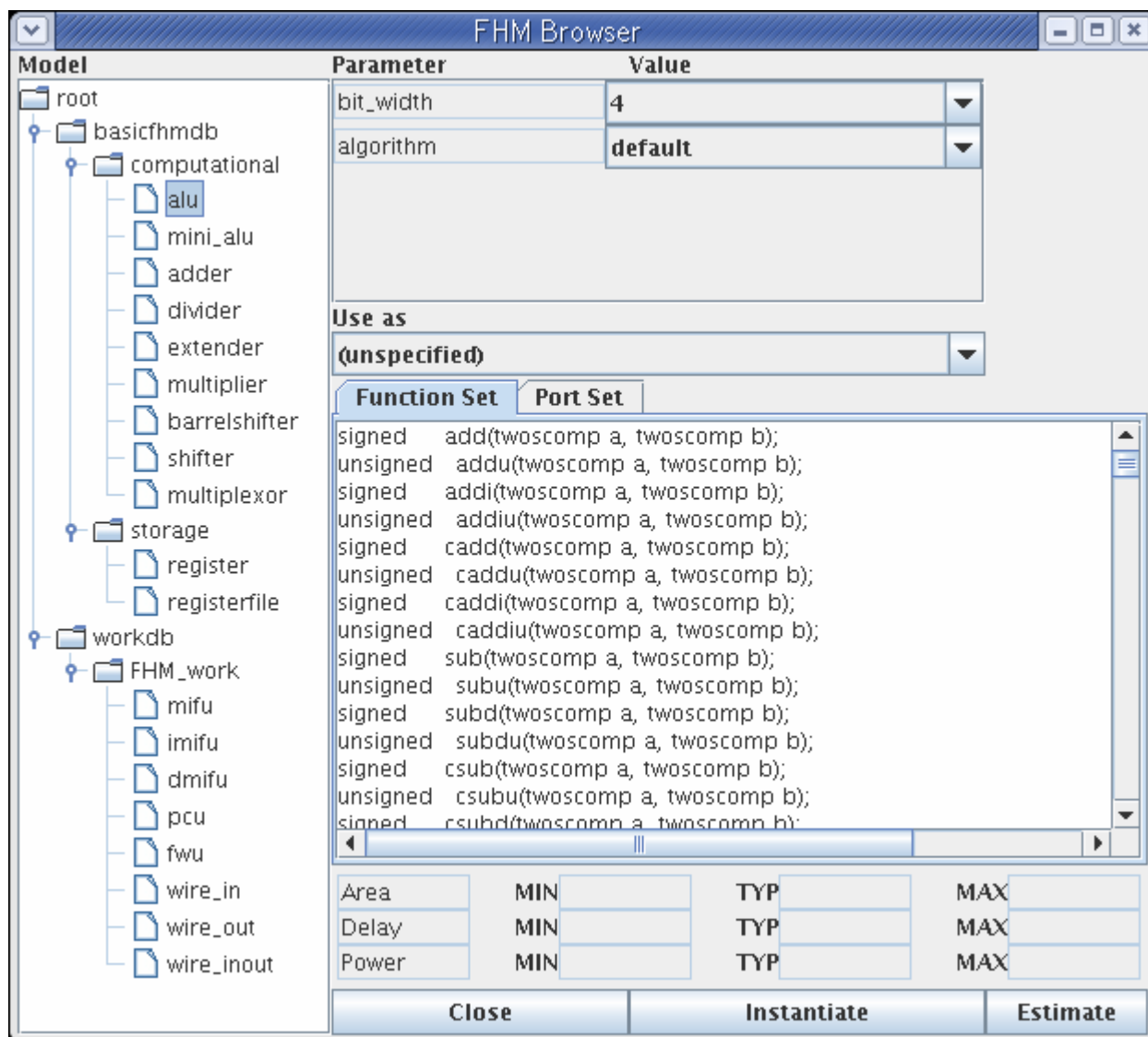




図14 FHM Browser


Function Set	Port Set				
a	in	bit_vector	3	0	data
b	in	bit_vector	3	0	data
cin	in	bit			mode
mode	in	bit_vector	4	0	mode
result	out	bit_vector	3	0	data
flag	out	bit_vector	3	0	data




ポート名




信号方向



データタイプ



ビット幅



属性

図15 [Port Set]: リソースモデルの入出力インターフェース

4. Storage Spec

[Storage Spec]はレジスタ・ファイル、レジスタそしてメモリなどのストレージに関して、ストレージの名前、ビット幅、そしてストレージの使用用途を定義します。ここで定義した情報はメタアセンブラ用アセンブル規則記述生成で使用されます。以下、レジスタファイル情報の入力、レジスタ情報の入力、メモリ情報の入力について説明します。

[Storage Spec]ボタンをクリックすると、[Storage Spec]ウィンドウが開きます。図 16に示す [Storage spec]ウィンドウは、[Register File]、[Register]、そして[Memory]の3つのタブから成ります。

4.1 [Register File]タブ

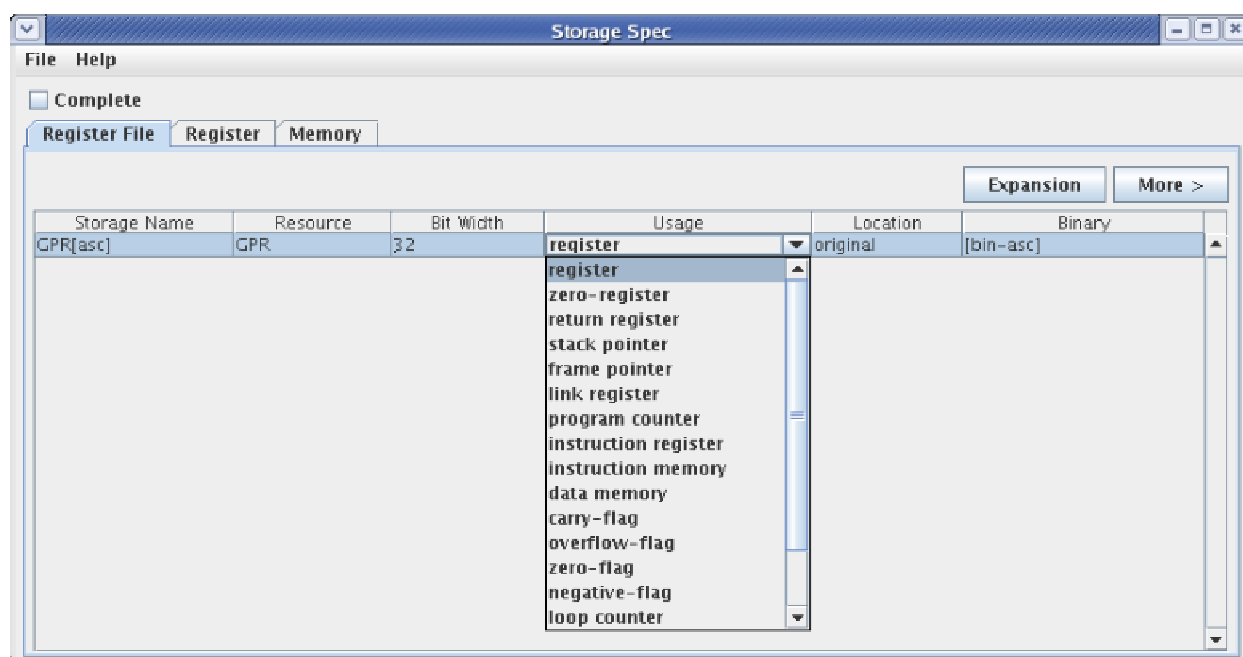


図16: レジスタ・ファイル定義.

[Register File]タブではレジスタ・ファイルに対するストレージ情報が定義できます。レジスタ・ファイルに対するストレージ情報は表 7に示す6つの要素から成ります。

表7: Storage Spec: Register File

Storage Name	アセンブリ・コードで使用するストレージ名
Resource	Resource Declaration フェーズで宣言されたリソース名
Bit Width	ビット幅
Usage	コンパイラ中でストレージがどのように扱われるかについて指定するストレージの使用方法
Location	ストレージ位置
Binary	機械語用バイナリ表現

レジスタ・ファイルは 2 つ以上のレジスタを有しますが、レジスタ・ファイル内の各レジスタに番号をつける必要があります。レジスタの番号付けは次のキーワードを用いて指定します:

[asc]: レジスタ・ファイルが n 個レジスタを持つ場合、0 から $n-1$ の昇順で番号付けを行います。
[des]: レジスタ・ファイルが n 個レジスタを持つ場合、 $n-1$ から 0 の降順で番号付けを行います。

さらに、それぞれのレジスタの機械語用バイナリ表現は次のキーワードを用いて指定します:

[bin-asc]: 2 進数でオール 0 からオール 1 の昇順のバイナリ表現を用います。
[bin-des]: 2 進数でオール 1 からオール 0 の降順のバイナリ表現を用います。

これらのキーワードを用いると、レジスタ・ファイル中のレジスタの定義が容易になります。ストレージ使用法はリストボックスを用いて選択します。 選択可能な使用法を表 8 に示します。

表8: ストレージ使用方法.

Register	データ・レジスタ：一般の演算で用いるレジスタです。
Zero register	ゼロ・レジスタ：常に 0 を保持するレジスタです。
Return register	リターン・レジスタ：関数呼び出しの戻り値を保持するレジスタです。
Stack pointer	スタック・ポインタ：スタックの先頭アドレスを保持するレジスタです。
Frame pointer	フレーム・ポインタ：関数呼び出し時にローカル関数の変数領域の先頭アドレスを保持するレジスタです。
Link register	リンク・レジスタ：戻りアドレスを保持するレジスタです。
Program Counter	プログラム・カウンタ：命令フェッチ時にアクセスするメモリアドレスを保持するレジスタです。
Instruction register	命令レジスタ：フェッチした命令を格納するレジスタです。
Instruction memory	命令メモリ：命令列を格納するメモリです。
Data memory	データメモリ：プロセッサによって処理されるデータを格納するメモリです。
Carry flag	キャリーフラグ格納用レジスタ
Overflow flag	オーバーフローフラグ格納用レジスタ
Zero flag	零フラグ格納用レジスタ
Negative flag	負号フラグ格納用レジスタ
Loop counter	ゼロ・オーバーヘッド・ループ(ZOL)拡張のためのループ・カウンタ（将来拡張予定）
Start address	ループの開始アドレス(ZOL 用)（将来拡張予定）
End address	ループの終了アドレス(ZOL 用)（将来拡張予定）
Instruction number	ループ・ブロックの命令数を示す(ZOL 用)。(将来拡張予定)

オーバーラップド・レジスタを、[Location]フィールドを用いて指定することができます。 オーバラップド・レジスタを指定する場合は、[Location]フィールドにストレージ名を記述します。 もし必要なければ“original”と書いてください。

レジスタ・ファイルの定義では、設計者はそれぞれのレジスタのレジスタ仕様を指定することができます。 [expansion]ボタンをクリックすると、図 17 のように展開されたストレージ一覧が表示さ

れます。 各レジスタの使用方法は、このウィンドウを用いて指定します。

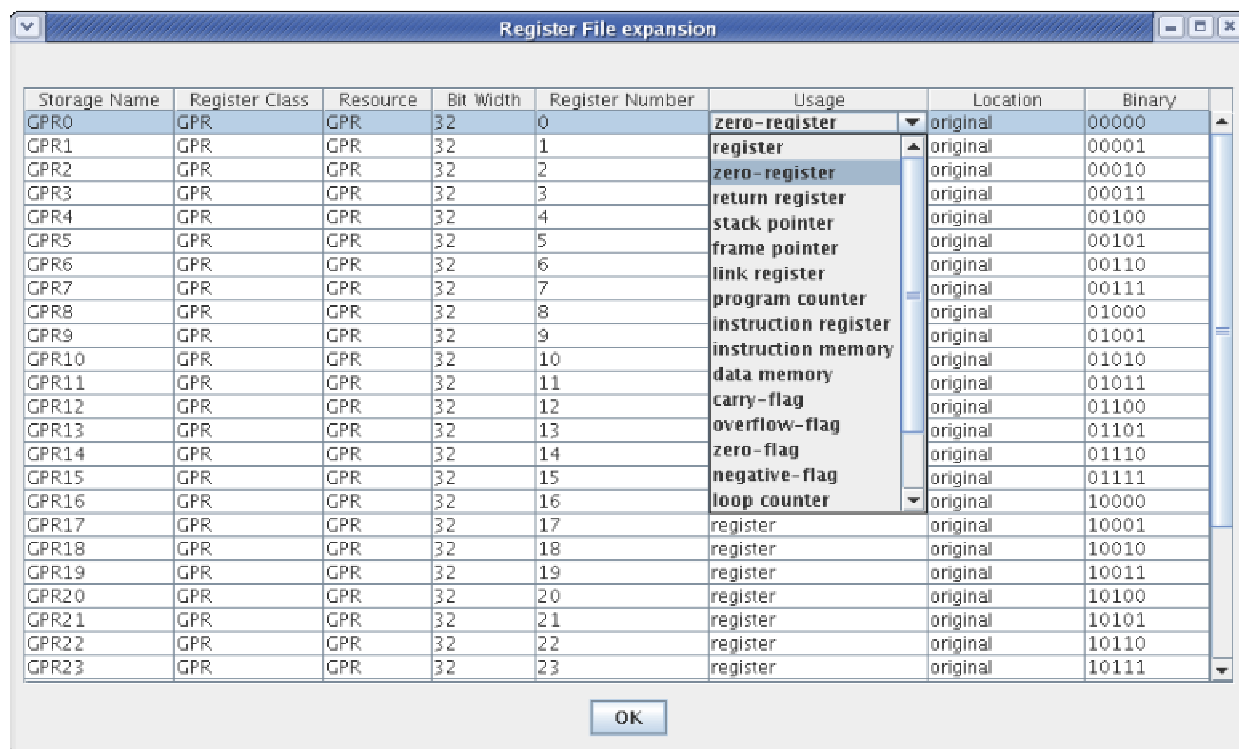


図17: 拡大されたストレージ一覧

4.2 [Register]タブ

レジスタ定義では、ストレージ名、リソース名、ビット幅と使用方法、ストレージ位置を指定します。(図 18).

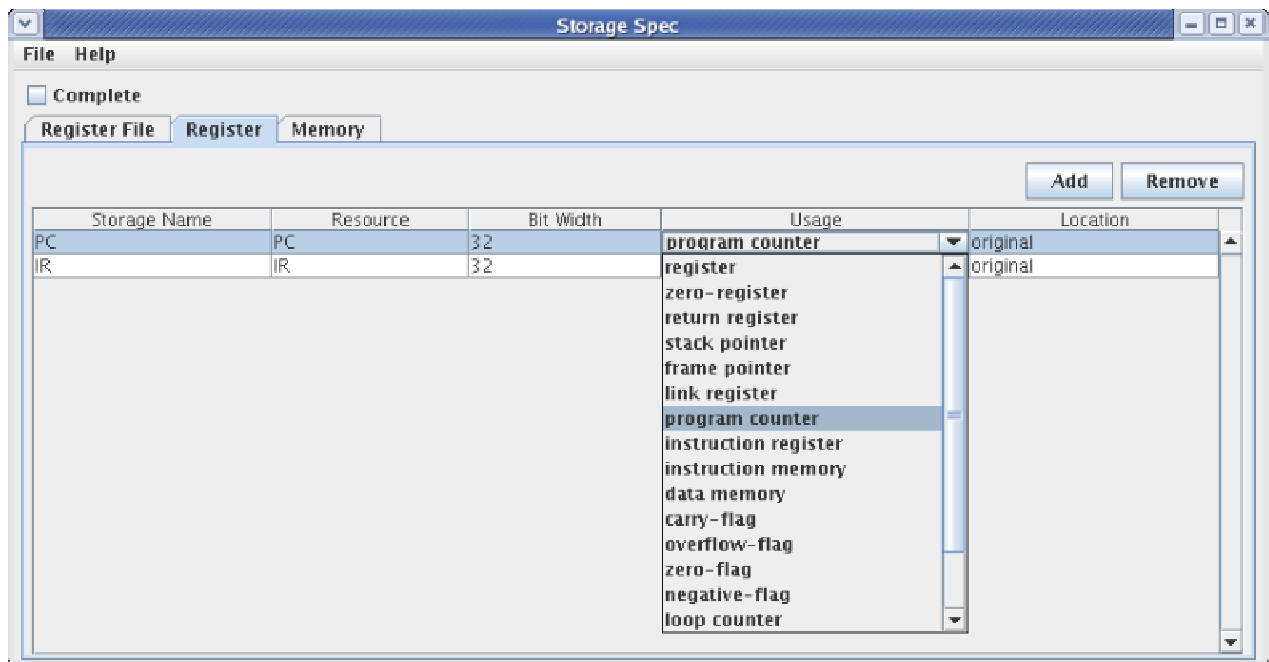


図18: レジスタ定義

4.3 [Memory]タブ

メモリ定義では、ストレージ名、リソース名、ビット幅、使用方法、アクセス・ビットを指定します(図 19)。 アクセス・ビットはプロセッサがメモリにアクセスする際の最小ビット幅です。

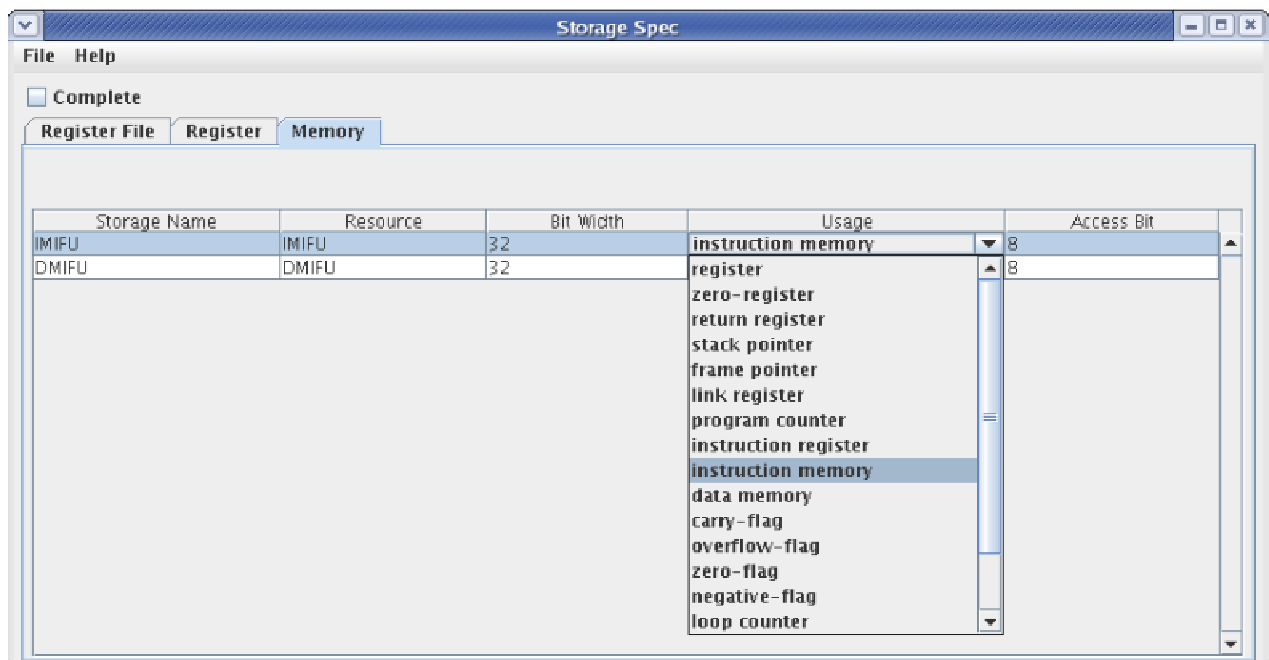


図19: メモリ定義

5. Interface Definition

メインウィンドウから[Interface Definition]をクリックすると図 20に示す[Interface Definition]ウィンドウが開きます。

[Interface Definition]では、設計するプロセッサの名前と入出力ポートに関する情報を入力し、プロセッサの外部とのインターフェースを決定します。

[Complete]ボタンの下には、プロセッサのエンティティ名が表示され、その下に、宣言されているポートの一覧が表示されます。各行がそれぞれポートに対応しており、そのポートの有効 / 無効、属性、名前、方向、ビット幅が表示されています。

ウィンドウの右上には、新たにポートを追加するための[New Port]ボタンがあります。

The screenshot shows the 'Interface Definition' window. At the top, there is a menu bar with 'File', 'Edit', 'Search', and 'Help'. Below the menu bar is a 'Complete' checkbox. Underneath, the 'Entity Name' is set to 'CPU'. To the right of the entity name is a 'New Port' button. The main area contains a table with the following columns: 'Valid', 'Attribute', 'Port Name', 'Direction', and 'Bit Width'.

Valid	Attribute	Port Name	Direction	Bit Width
<input checked="" type="checkbox"/>	clock	CLK	in	1
<input checked="" type="checkbox"/>	reset	Reset	in	1
<input checked="" type="checkbox"/>	instruction_memory_address_bus	InstAB	out	32
<input checked="" type="checkbox"/>	instruction_memory_data_bus	InstDB	in	32
<input checked="" type="checkbox"/>	data_memory_address_bus	DataAB	out	32
<input checked="" type="checkbox"/>	data_memory_data_in_bus	DataDIB	in	32
<input checked="" type="checkbox"/>	data_memory_data_out_bus	DataDOB	out	32
<input checked="" type="checkbox"/>	data_memory_request_bus	DataReq	out	1
<input checked="" type="checkbox"/>	data_memory_acknowledge_bus	DataAck	in	1
<input checked="" type="checkbox"/>	data_memory_rw_bus	DataRW	out	1
<input checked="" type="checkbox"/>	data_memory_write_mode_bus	DataMode	out	2
<input checked="" type="checkbox"/>	data_memory_ext_mode_bus	DataExt	out	1
<input checked="" type="checkbox"/>	data_memory_address_error_bus	DataAderr	in	1
<input checked="" type="checkbox"/>	data_memory_cancel_bus	DataCancel	out	1
<input type="checkbox"/>			in	

Below the table, there is a list box containing the following attributes: clock, reset, instruction_memory_address_bus, instruction_memory_data_bus, data_memory_address_bus, data_memory_data_in_bus, data_memory_data_out_bus, and data_memory_request_bus. The list box has a scrollbar and a search icon.

図20 [Interface Definition]ウィンドウ

＜注意＞**ASIP Meister** では、メモリアンタフェースユニット(MIFU)とユーザが定義した WIRE リソースを外部ポートに接続できます。[Interface Definition]で宣言したポートと同名の WIRE リソースを追加することで外部と接続できます。

ASIP Meister はメモリポートが定義されている設計データファイルを持っています。よく使用されるメモリ構成についてはこの設計データファイルを使用して、外部ポート属性の設定を省略することができます。

[Entity Name]: プロセッサの名前

ここで入力した名前がプロセッサのエンティティ名となります。[HDL Generation]では、ここで入力したエンティティ名に拡張子“.vhd”もしくは“.v”が付いた HDL ファイルがトップレベルコンポーネントとして生成されます。

＜注意＞ここで入力した名前は[HDL Generation]で生成されるプロセッサ記述にもそのまま使われますので、HDL 生成後に使用する合成ツールやシミュレータのエンティティ名の制限を満たしておく必要があります。

＜参照＞4. 予約語

[Valid]: ポートの有効 / 無効

宣言されているポートを実際に使用するかどうかを指定します。その行のポートを使用するときは、チェックボックスをチェックします。チェックされていない行のポートは、プロセッサの外部ポートとして使用されなくなります。

[Attribute]: 外部ポートの属性

外部ポートの属性を入力します。ここで指定した属性に従って、[HDL Generation]では、外部ポートと内部の信号線を接続していきます。表 9に、指定できる属性とその意味について示します。

表9 外部ポート属性一覧表

属性	プロセッサ内部の接続先
Clock	クロック
Reset	リセット
instruction_memory_address_bus	命令メモリのアドレスバス
instruction_memory_data_in_bus	命令メモリのデータ入力バス
instruction_memory_data_out_bus	命令メモリのデータ出力バス
instruction_memory_request_bus	命令メモリのリクエストバス
instruction_memory_acknowledge_bus	命令メモリのアクノリッジバス
instruction_memory_rw_bus	命令メモリのリード/ライト指定
instruction_memory_write_mode_bus	命令メモリのアクセス・モード指定バス
instruction_memory_ext_mode_bus	命令メモリのリード符号拡張指定
instruction_memory_address_error_bus	命令メモリのアドレスエラーバス
instruction_memory_cancel_bus	命令メモリのキャンセルバス
data_memory_address_bus	データメモリのアドレスバス
data_memory_data_in_bus	データメモリのデータ入力バス
data_memory_data_out_bus	データメモリのデータ出力バス
data_memory_request_bus	データメモリのリクエストバス
data_memory_acknowledge_bus	データメモリのアクノリッジバス
data_memory_rw_bus	データメモリのリード/ライト指定
data_memory_access_mode_bus	データメモリのアクセス・モード指定バス
data_memory_ext_mode_bus	データメモリのリード符号拡張指定

属性	プロセッサ内部の接続先
data_memory_address_error_bus	データメモリのアドレスエラーバス
data_memory_cancel_bus	データメモリのキャンセルバス
interrupt	割り込みバス
unspecified	ユーザ定義 WIRE リソース

[Port Name]: ポート名

外部ポート名を入力します。[Valid]列がチェックされているときはポート名の変更はできません。外部ポート名を変更するときは、[Valid]列のチェックをはずしてください。ポート名は、VHDL のネーミングルールに準拠しておく必要があります。

<参照>4. 予約語

[Direction]: 信号方向

外部ポートの信号の向きを、プルダウンメニューにより、入力[in]、出力[out]及び双方向[inout]から選択します。

[Bit Width]: ビット幅

外部ポートのビット幅を指定します。

[New Port]: ポートの追加

[New Port]ボタンを押すと、ポート一覧の一番下に新しい行が加わり、新しいポートが宣言できます。

<注意>一番下の行が未定義([Port Name]や[Bit Width]が空白)の時は新しい行は追加されません。

表10 [Interface Definition]で設定する項目の一覧

パラメータ	機能及び意味
Entity Name	プロセッサの名前(エンティティ名)
Attribute	外部ポートの属性
Port Name	外部ポート名
Direction	信号方向。入力[in]、出力[out]、双方向[inout]から選択
Bit Width	ビット幅

6. Instruction Definition

メインウインドウから[Instruction Definition]をクリックすると[Instruction Definition]ウインドウが開きます。(図 21)

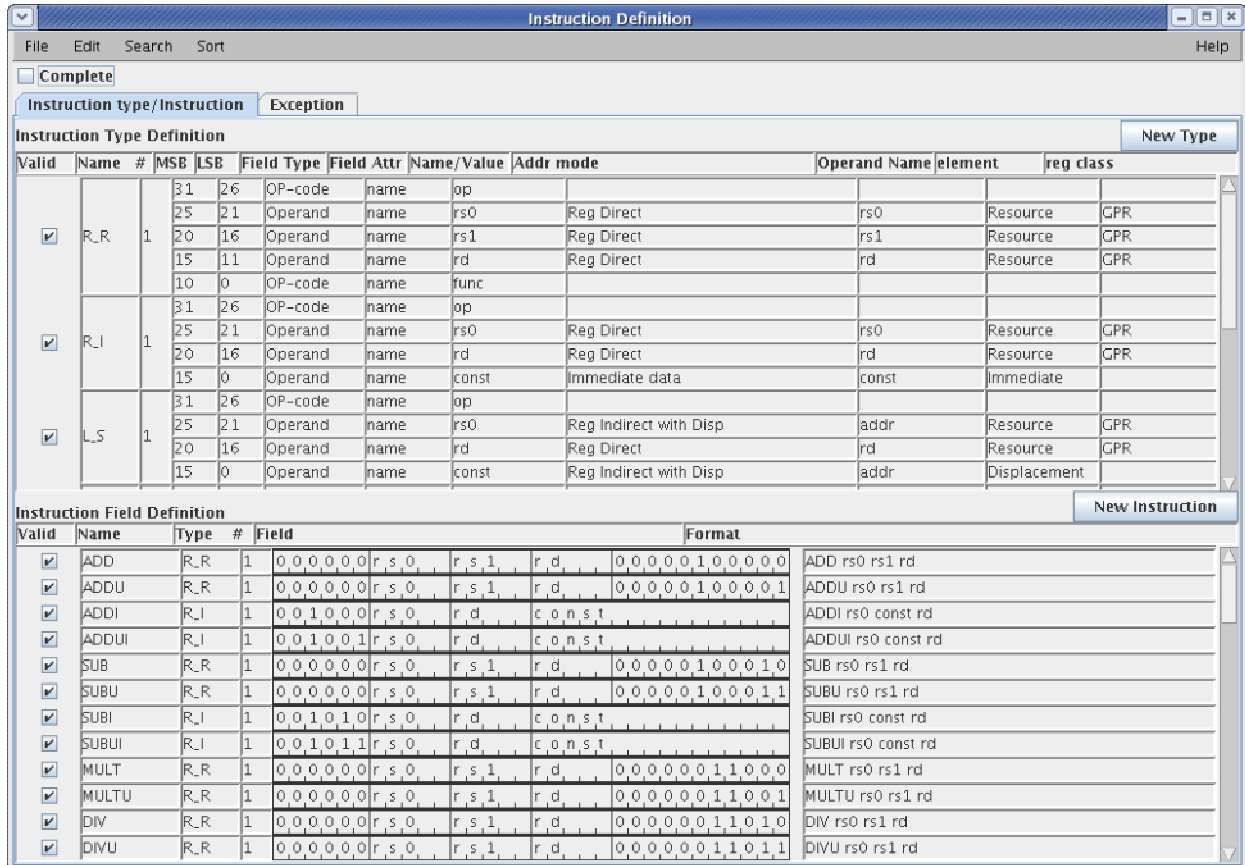


図21 [Instruction Definition]ウインドウ

[Instruction Definition]では **ASIP Meister** の命令タイプと命令及び割込みを定義します。命令タイプと命令形式は、[Instruction type / Instruction]タブから定義します。命令は命令タイプを使って定義しますので、必要な命令タイプを先に定義しておく必要があります。割込みは、[Exception]タグから定義します。

[Design Goal & Arch. Design]フェーズの[Processor Design], [Use Compiler]項目の指定により、[Instruction Definition]における制限が異なります。以下にその制限を示します。

- [New Design]：制限なし
- [New Design with Base Processor]
 - [Use Compiler]が”yes”のとき
 - ◇ ベースプロセッサが持つ命令タイプ、命令の編集や削除は不可
 - ◇ 新たな命令タイプの追加、命令の追加は可能
 - [Use Compiler]が”no”のとき
 - ◇ 制限なし
- [Base Processor Design]：制限なし

6.1 命令タイプの定義: Instruction Type Definition]

[Instruction Type Definition]で設定する項目について説明します。

[Valid]: 命令タイプの有効 / 無効

定義されている命令タイプを実際使用するかどうかを指定します。その命令タイプを使用するときは、チェックボックスをチェックします。チェックされていない命令タイプは、命令タイプに使用できなくなります。

[Name]: 命令タイプの名前

[#]: 命令タイプの識別番号

同一名の命令タイプの識別番号は、多重定義された命令タイプを区別するために使います。定義済みの命令タイプと同じ名前、異なる命令タイプを定義したい場合に、異なる識別番号を与えることで定義します。識別番号を変えることで同一名の命令タイプを複数定義し、[Valid]チェックボックスでどの命令タイプを用いるかを指定します。同一名の命令タイプを複数個記述できますが、異なる識別番号を持つ同一名の命令タイプを複数個有効にすることはできません。ある命令タイプが有効になっているときに、識別番号だけが異なる同一名の命令タイプを有効にすると、確認画面が表示され、今まで有効になっていた命令タイプが無効となり、必ずひとつの命令タイプのみが有効となります。この命令タイプを使って定義された命令も、新しい命令タイプのものへと変更されます。

[MSB]: フィールドの Most Significant Bit

フィールドの最上位ビット。命令の識別番号の隣には、その命令の各フィールドが行毎に表示されています。MSB は、そのフィールドの最上位ビットです。

[LSB]: フィールドの Least Significant Bit

フィールドの最下位ビット。

[Field Type]: フィールドのタイプ

各フィールドのタイプが表示されています。そのフィールドが、オペコードの時は[OP-code]、オペランドの時は[Operand]、予約フィールドの時は[Reserved]、ドントケアの時は[Dont_care]が表示されます。

[Field Attr]: 命令フィールドの共有 / 非共有

この命令タイプを使用して複数の命令を定義するときに、このフィールドが各命令において共通の場合は[binary]となり、命令毎に異なる場合は[name]となります。尚、[Field Type]が[Operand]の時は[name]しか選択できません。

表11 [Field Attr]パラメータ選択条件

Field Type	選択できる [Field Attr]	選択基準
OP-code	Binary	この命令タイプを使用する命令が、ここで定義したフィールドの値を継承する場合、つまり、命令セットの定義の時に、このフィールドの値を命令毎に変更しない場合
	Name	この命令タイプを使用する命令が、ここで定義したフィールドの値を変更する場合
Operand	Name	この命令タイプを使用する命令が、ここで定義したフィールドの値を変更する場合。[Field Type]で[Operand]を選択した場合は、ここでは[name]しか選択できません
Reserved	Binary	予約フィールドとして値を固定する場合。
Dont_care	Name	未使用フィールドとして値をドントケアとする場合。

[Name / Value]: フィールドの名前 / 値

そのフィールドの名前や値です。[Field Attr]が[binary]の時はバイナリ値となり、[name]の時は文字列となります。

[Addr Mode]: アドレッシング・モード

オペランドの使用方法はアドレッシング・モードで指定します。アドレッシング・モードは表 12 から選択します。アドレッシング・モードはオペランド・フィールドを含みます。表 12 にはアドレッシング・モード名、要素タイプ、説明を示します。要素は **element field** で指定します。それぞれのオペランドはアドレッシング・モードを持っています。もし 2 つの要素を含むようなアドレッシング・モードを使用したい場合は、同じオペランド名を指定します。

表12: Addressing Mode

アドレッシング・モード	要素タイプ	説明
Reg Direct	Resource	オペランド・レジスタの値
Indirect	Immediate	即値で指定する値をアドレスとしたメモリの値。
Reg Indirect	Resource	オペランド・レジスタの値をアドレスとしたメモリの値
Reg Indirect with Pre-Decrement	Resource and Displacement	オペランド・レジスタの値をアドレスとしたメモリの値 オペランド・レジスタの値はメモリアクセス前にデクリメントされます。
Reg Indirect with Pre-Increment	Resource and Displacement	オペランド・レジスタの値をアドレスとしたメモリの値 オペランド・レジスタの値はメモリアクセス前にインクリメントされます。
Reg Indirect with Post-Decrement	Resource and Displacement	オペランド・レジスタの値をアドレスとしたメモリの値 オペランド・レジスタの値はメモリアクセス後にデクリメントされます。
Reg Indirect with Post-Increment	Resource and Displacement	オペランド・レジスタの値をアドレスとしたメモリの値 オペランド・レジスタの値はメモリアクセス後にインクリメントされます。
Reg Indirect with Disp	Resource and Displacement	オペランド・レジスタの値とディスプレースメントを加算した値をアドレスとしたメモリの値
Reg Indirect with Disp and Increment	Resource and Displacement	オペランド・レジスタの値とディスプレースメントの和をアドレスとしたメモリの値 オペランド・レジスタはメモリアクセス後にディスプレースメントが加えられます。
Reg Indirect with Index	Resource and Displacement	オペランド・レジスタとインデックス・レジスタの値を加算した値をアドレスとするメモリの値
Reg Indirect with Index and Increment	Resource and Displacement	オペランド・レジスタとインデックス・レジスタの値の和をアドレスとするメモリの値 インデックス・レジスタの値はオペランド・レジスタに加えられます。
Reg Indirect with Scaled Index	Resource and Displacement and	オペランド・レジスタの値と、スケールで指定した値とインデックス・レジスタの値の積の和をアド

アドレッシング・モード	要素タイプ	説明
	Scale	レスとするメモリの値
Register Indirect with Disp and Scaled Index	Resource and Displacement and Scale	オペランド・レジスタの値、スケールで指定した値とインデックス・レジスタの値の積、ディスプレイースメントの三つの和をアドレスとするメモリの値
PC relative address	Symbol	PC 相対アドレッシング
Absolute address	Symbol	絶対アドレッシング
Immediate data	Immediate	即値

[Operand Name]

アセンブラ言語においては、オペランドはニモニック形式を用いて表現されます。ASIP Meisterでは、ニモニック形式は文字列とオペランド名を用いて記述されます。設計者はこのフィールドでオペランド名を宣言します。“Reg Indirect with Disp”のような2つの要素を含むアドレッシング・モードを選択する場合、オペランドに2つのビット・フィールドを持たせるようにし、“Operand Name”セクションには同じオペランド名を記述するようにしてください。

[element]

アドレッシング・モードの要素はここで指定します。要素はプルダウン・メニューから選択します。表12: Addressing Modeによって、設計者はそれぞれのオペランドの要素を記述しなければなりません。例として、“Reg Indirect with Disp”アドレッシング・モードを選択した場合、リソース要素か、さもなければディスプレイースメント要素を指定しなければなりません。

[reg class]: レジスタ・クラス

“Resource”として要素のタイプを選択した場合、レジスタ・クラスを指定しなければなりません。レジスタ・クラスはオペランドのアクセス可能なレジスタを指定します。例として、もし、一覧から“GPR”を選択すると、オペランドは“GPR”にアクセスできるようになります。もし、レジスタ・ファイル要素のひとつである“GPR0”を選択すると、オペランドは“GPR0”しかアクセスできません。レジスタ・クラスは宣言段階中で宣言されます。このレジスタ・クラスは以前の設計フェーズ“Storage Specification”で宣言されたものであることに注意してください。

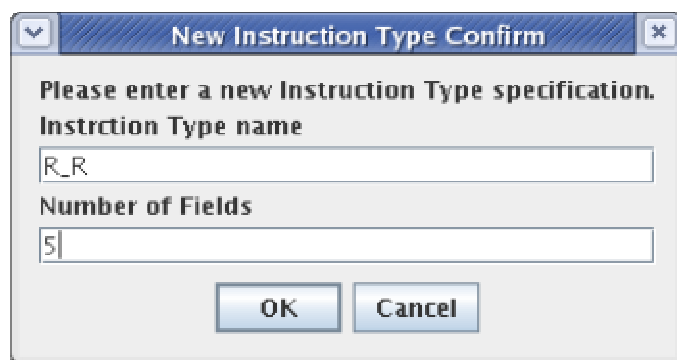
表13 [Instruction Type Definition]パラメータ

項目	機能及び意味
Instruction Type Name	命令タイプ名
Number of Fields	フィールド数
MSB	フィールドの最上位ビット
LSB	フィールドの最下位ビット
Field Type	フィールドタイプを指定 オペレーションコード[OP-code] オペランド[Operand] 予約フィールド[Reserved] ドントケア[Dont_care]

項目	機能及び意味
Field Attr	次から選択 [binary] 命令タイプの定義で設定したフィールド値を共通使用 [name] 命令毎にフィールド値を定義
Name / Value	フィールド値 オペレーションコード、予約フィールドのバイナリ値 オペレーションコード名、オペランド名、ドントケアの文字列
Addr mode	オペランドのアドレッシングモード
Operand Name	オペランド名
Element	アドレッシングモードの要素を指定
reg class	アドレッシングモードのレジスタクラスを指定

[New Type]: 新しい命令タイプの定義

[New Type]をクリックすると新しい命令タイプを追加できます。[New Instruction Type Confirm]ウィンドウで、命令タイプ名とフィールド数を入力します。



New Instruction Type Confirm

Please enter a new Instruction Type specification.

Instruction Type name
R_R

Number of Fields
5

OK Cancel

図22 [New Instruction Type Confirm]ウィンドウ

[OK]ボタンを押すと、[Instruction Type Definition]ウィンドウが開きます。

MSB	LSB	Field Type	Field Attr	Value	Addr mode	Operand Name	element	reg class
31		OP-code	binary		Reg Direct		Resource	
		OP-code	binary		Reg Direct		Resource	
		OP-code	binary		Reg Direct		Resource	
		OP-code	binary		Reg Direct		Resource	
	0	OP-code	binary		Reg Direct		Resource	

図23 [Instruction Type Definition]ウィンドウ

[Instruction Type Definition]ウィンドウで、各フィールドの情報を定義します。

(1) フィールドのビット位置: [MSB],[LSB]

各フィールドの最上位ビット、最下位ビットを入力します。

(2) フィールドタイプ: [Field Type]

フィールドが、オペレーションコード、オペランド、予約フィールド、またはドントケアかを指定します。

▼をプルダウンして[OP-code]、[Operand]、[Reserved]、[Dont_care]から選択します。

MSB	LSB	Field Type	Field Attr	Value	Addr mode	Operand Name	element	reg class
31		OP-code	binary		Reg Direct		Resource	
		OP-code	binary		Reg Direct		Resource	
		Operand	binary		Reg Direct		Resource	
		Reserved	binary		Reg Direct		Resource	
		Dont_care	binary		Reg Direct		Resource	
	0	OP-code	binary		Reg Direct		Resource	

(3) フィールド値の継承属性: [Field Attr]

フィールド値が命令定義に継承されるのか、継承されずに命令毎に定義されるのかを指定します。
▼をプルダウンして[name]、[binary]から選択します。

The screenshot shows the 'Instruction Type Definition' window with the 'Field Attr' column set to 'binary' for all fields. The 'Value' column is empty for all fields.

MSB	LSB	Field Type	Field Attr	Value	Addr mode	Operand Name	element	reg class
31		OP-code	binary		Reg Direct		Resource	
		OP-code	binary		Reg Direct		Resource	
		OP-code	binary		Reg Direct		Resource	
		OP-code	binary		Reg Direct		Resource	
	0	OP-code	binary		Reg Direct		Resource	

(4) フィールド値の設定: [Value]

オペレーションコードの値やオペランドの名前を定義します。[Field Attr]が[binary]のときはバイナリ値を、[name]のときは文字列を入力します。

The screenshot shows the 'Instruction Type Definition' window with the 'Value' column set to 'name' for all fields. The 'Value' column contains the following values: 'op', 'rs0', 'rs1', 'rd', and 'func'.

MSB	LSB	Field Type	Field Attr	Value	Addr mode	Operand Name	element	reg class
31	26	OP-code	name	op	Reg Direct		Resource	
25	21	Operand	name	rs0	Reg Direct	rs0	Resource	GPR
20	16	Operand	name	rs1	Reg Direct	rs1	Resource	GPR
15	11	Operand	name	rd	Reg Direct	rd	Resource	GPR
10	0	OP-code	name	func	Reg Direct		Resource	

図24 [Instruction Type Definition]ウィンドウ

(5) 全ての設定が終了しましたら[OK]ボタンを押します。

6.2 命令の定義: [Instruction Declaration]

[Instruction Declaration]では、[Instruction type]で定義した命令タイプを使って命令を定義します。

(6) 命令定義パラメータ

下表に[Instruction Declaration]パラメータを示します。

表14 [Instruction Declaration]パラメータ

パラメータ	機能及び意味	入力データまたは入力方法
Instruction Name	命令名(ニーモニック)	文字列 <参照>4. 予約語
Instruction Type	命令タイプ名	定義済みの命令タイプ一覧から選択
Value	オペレーションコードまたはオペランド。 命令タイプの[Field Type]での[Field Attr]が[name]の場合にのみ値を書き換え可能です。 それ以外の場合は、命令タイプのフィールドの値が継承されるので、変更できません。	オペレーションコードのバイナリ値か、オペランド名を入力します。

[New Instruction]: 新規命令の追加

[Instruction Definition]ウィンドウの[New Instruction]ボタンを押すと新しい命令名(ニーモニック)を入力するウィンドウ[Input]が開きますので(図 25参照)、命令の名前を入力し、[OK]を押すと、[Instruction Declaration Window]ウィンドウが開きます(図 26参照)。

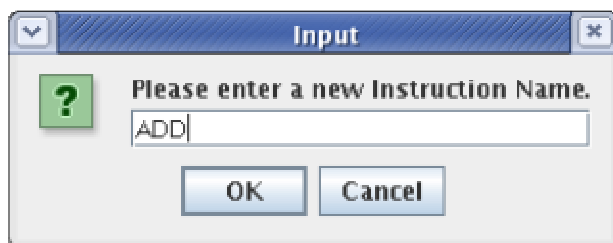


図25 命令名入力ウィンドウ

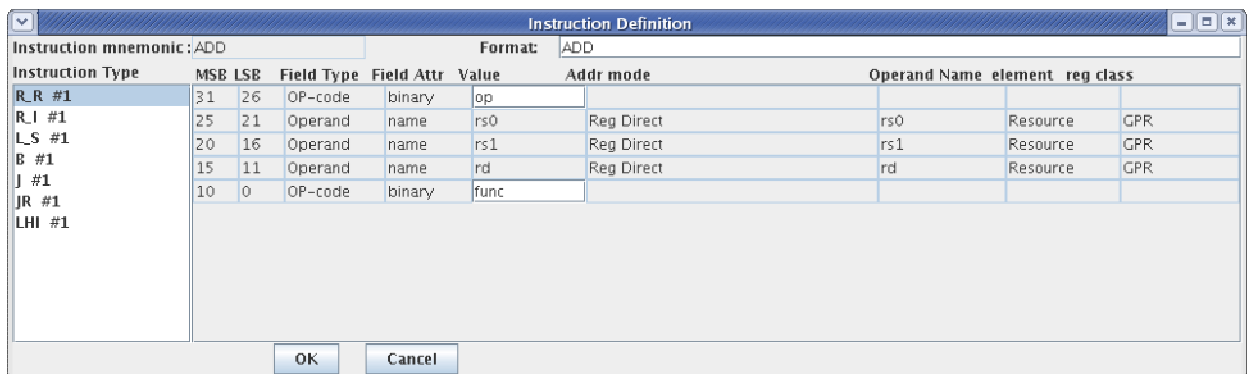


図26 新命令定義

[Instruction mnemonic]: 命令のニーモニック

[input]ウィンドウで入力した名前が表示されています。ここでは命令名を変更することはできません。

[Instruction Type]: 命令タイプの選択

定義した命令タイプとその識別番号が表示されていますので、使用する命令タイプを選択します。選択した命令タイプによって、[MSB]、[LSB]、[Field Type]、[Field Attr]、[Value]などの項目がフィールド数分表示されます。

[Value]: フィールドの値

各フィールドの値を表示しています。

命令タイプの[Field Attr]が[binary]となっているフィールドの[Value]が入力ボックスになりますので、そのフィールドの値を入力します。命令タイプの[Field Attr]が[binary]でないときは、命令タイプのフィールドの値が継承されます。

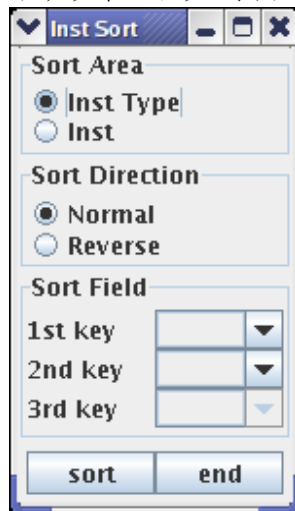
[Format]: 命令のアセンブリフォーマット

命令のアセンブリフォーマットを入力します。アセンブリフォーマットは、命令名と全てのオペランドを記述します。また、オペランドの記述の順番によって、生成されるコンパイラ、メタアセンブラの仕様がことなります。

<注意> [Field Type]が”Operand”かつ[Field Attr]が”name”のオペランドは必ず[Format]に記述する必要があります。

[Sort]メニュー：命令タイプ・命令の並び替え

[Instruction Definition]ウィンドウのメニューバーの[Sort]から[Sort]をクリックすると、下の図に示すウィンドウが表示され、命令タイプおよび命令の並び替えが可能です。



6.3 割込み(例外)の定義: [Exception Declaration]

[Exception]をクリックすると、割込み(例外)定義サブウィンドウ [Internal / Exception Declaration]が開きます。リセット割込みは必ず定義してください。

[Interrupt Name]: 割込み(例外)名

[Type]: 割り込みタイプ

割込みタイプをプルダウンメニューから選択します。External, Internal, NMI, Reset から選択します。

[Condition]: 発生条件を論理式で記述

割り込み発生条件を記述します。外部ポートと信号値の対応で、条件を記述します。外部ポートをプルダウンメニューから選択し、割り込み時の値を[Active Value]欄に入力します。内部割込みの場合のみ Type を指定します。”Unselected”, “decoder_error”, “instr_specific”の3つから選択します。

[Valid]: 割り込みの有効・無効

定義された割り込みの有効・無効を選択します。

表15 割込み(例外)定義パラメータ

パラメータ	機能及び意味	入力データ及び入力方法
Interrupt Name	割込み(例外)名	任意の文字列
Type	割り込みタイプを指定	▼プルダウンメニューから選択。 External: 外部割込み Internal: 内部割込み NMI: マスク不可割込み Reset: リセット割込み (必須)
Condition	発生条件を論理式で記述	記述方法は構文、演算子割り込み発生条件を記述します。外部ポートと信号値の対応で、割り込み発生時の値を記述します。
Valid	設定の有効・無効を指定	チェックボックスをクリック

<注意>

- ・外部割込みは1つでなければなりません.
- ・リセット割込みは必ず必要です.

7. Arch.Level Estimation

[Arch. level Estimation]では、現在設計しているプロセッサの性能(面積、遅延時間、消費電力)を見積もります。マイクロ動作記述を書く前なので、見積もりの精度は粗くなりますが、設計早期の段階で見積もることができます。ここでの見積もりにより所望の性能が得られそうに無いことがわかったときは、これまでの設計内容(使用するリソースモデルやパラメータ)を変更してください。

[Estimation Confirm]: 見積もりシステムの選択ウィンドウ

メインウィンドウで[Arch. Level Estimation]をクリックすると、概略見積もりエンジン選択ウィンドウ[Estimation Confirm]が開きます。(図 27参照)

プルダウンメニューから、見積もりエンジンを選択し、[Execute]を押して、プロセッサの性能を見積もります。プロセッサ性能を見積もった結果がウィンドウ(図 28)に表示されます。

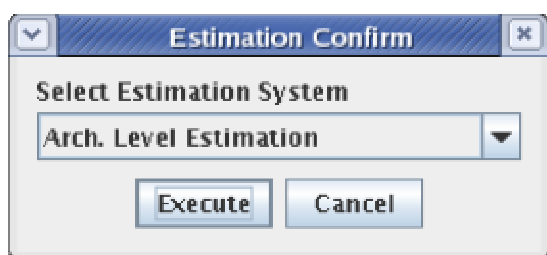


図27 見積もりシステム選択ウィンドウ

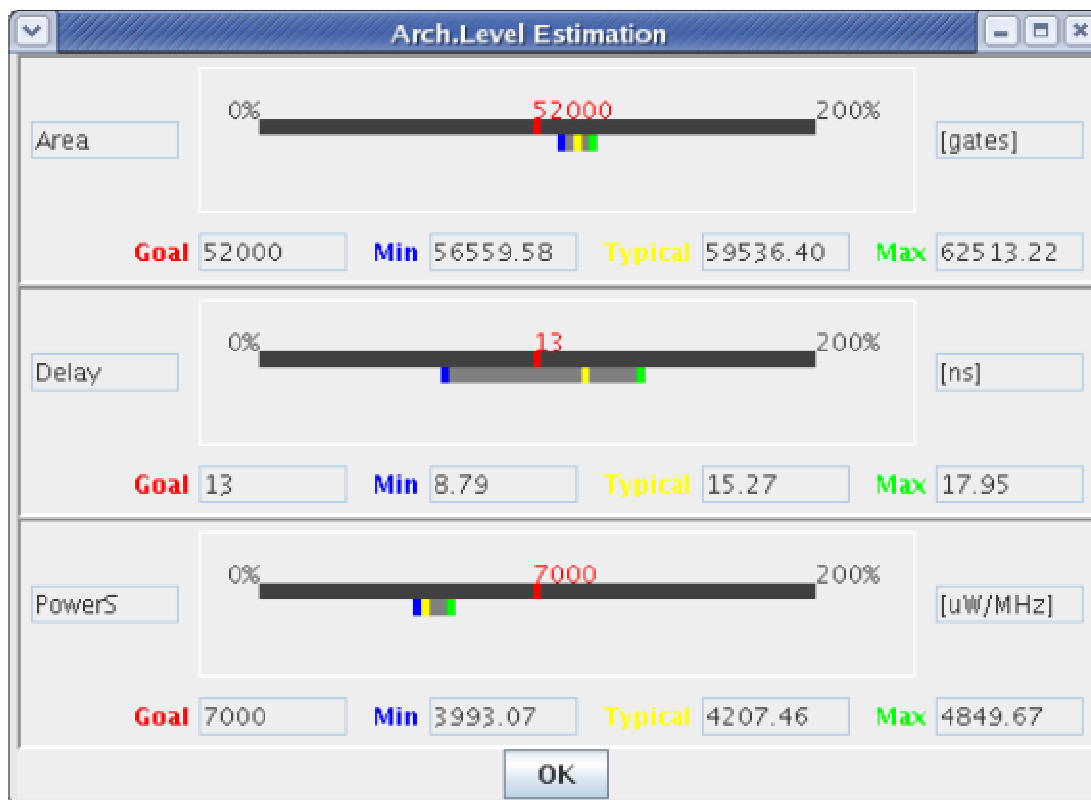
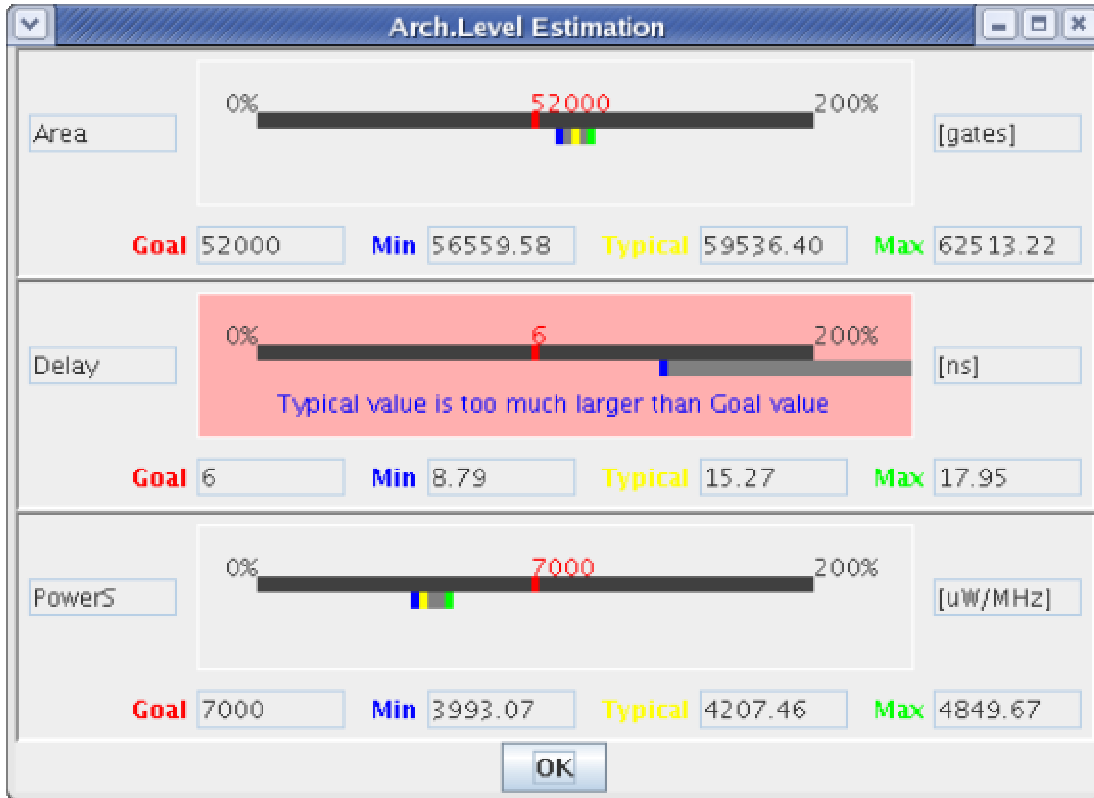


図28 見積もり結果ウィンドウ

見積もりエンジンによる面積[Area]、最大パス遅延[Delay]、消費電力[Power S]の算出結果がそれ

ぞれ次の項目に、また、[Design Goal & Arch. Design]で設定した目標値[Goal]に表示され、[Goal]との差分割合(%)がグラフに表示されます。また、[Min]、[Typical]、[Max]には、見積もりエンジンが算出した各性能の最大値、標準値、最小値が表示されます。

見積もり値が[Design Goal & Arch. Design]で設定した目標値よりも大幅に大きいか小さいときは警告が表示されます。



8. Assembler Generation

メインウィンドウで[Assembler Generation]をクリックすると、図 29 [Generation Confirm]ウィンドウの[Generation Confirm]ウィンドウが開きます。

[Assembler Generation]では、メタアセンブラ用の記述を自動生成することができます。

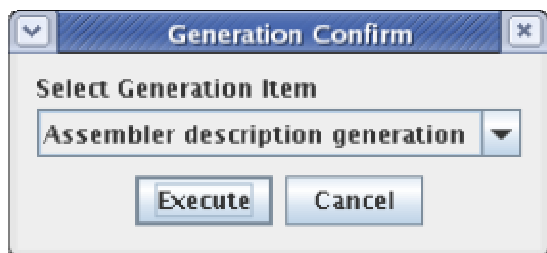


図29 [Generation Confirm]ウィンドウ

[Generation Confirm]ウィンドウにおいて[Execute]を押すと、メタアセンブラ用記述の自動生成が行われます。

メタアセンブラ用記述の生成が完了すると、図 30のウィンドウが開きます。エラーが無いことを確認し、[OK]ボタンを押すとメインウィンドウに戻ります。



図30 メタアセンブラ用記述記述生成完了ウィンドウ

メタアセンブラ用記述の自動生成が正常に終了すると、カレントディレクトリの“meister”ディレクトリの下にファイルが作成されます。設計データファイル名が“model.pdb”だった場合、“model.des”ファイルが生成されます。

9. Micro Op. Description

[Micro Op. Description] では、[Instruction Definition] で定義した命令と割込み(例外)について、ステージ毎に動作記述をします。動作記述は、命令名の選択、変数の宣言、及びステージ毎の動作記述から構成されます。**ASIP Meister** のマイクロ動作記述には次のような特徴があります。

- 使用リソースやパイプライン動作を意識しない
- 命令の機能や振る舞いを記述する
- 割込み(例外)発生時のハードウェア動作を記述する

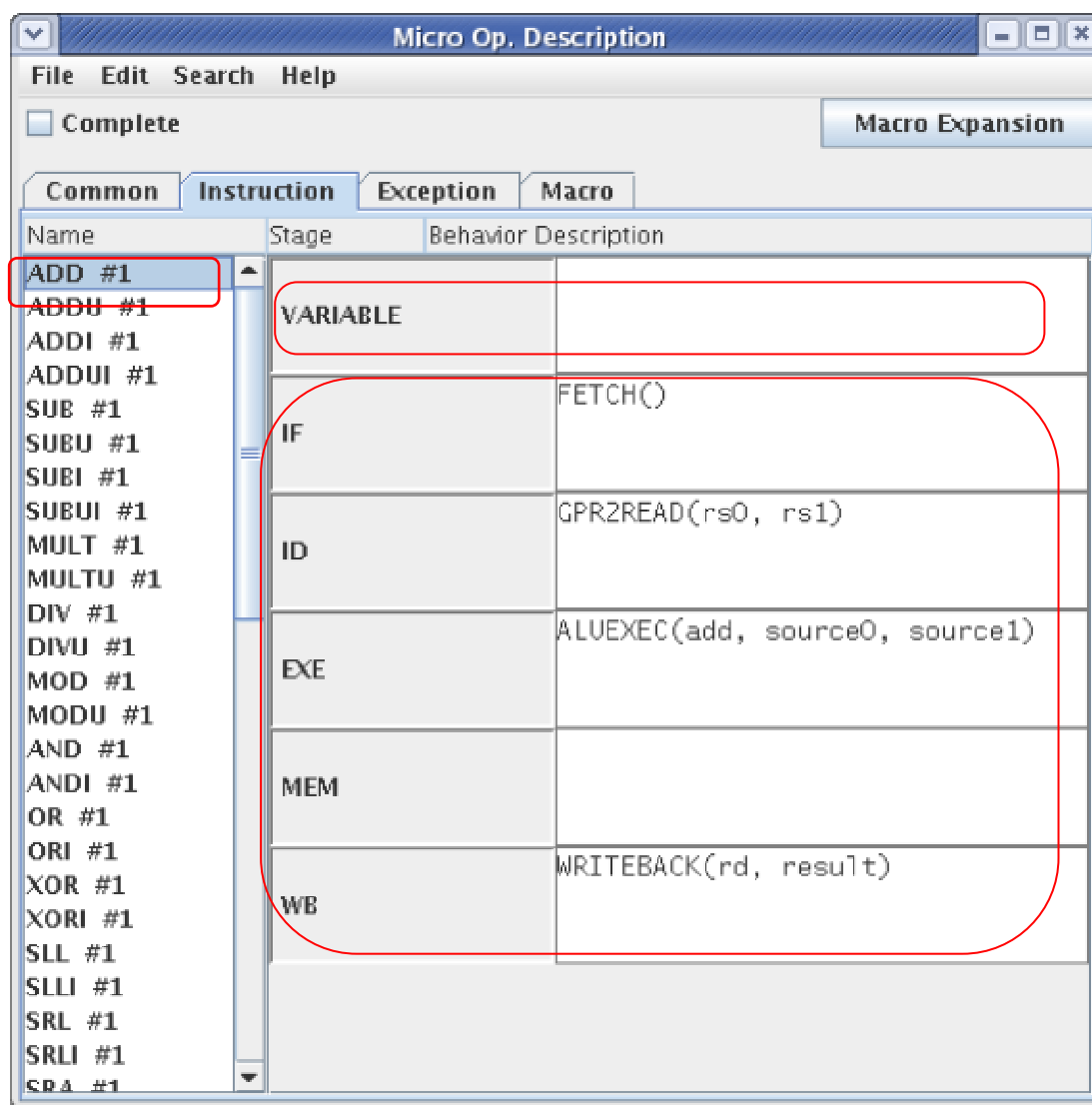


図31 [Micro Op. Description] ウィンドウ

文法や構文などマイクロ動作記述の書き方の詳細は、「マイクロ動作記述の書き方」を参照してください。

[Micro Op. Description] ウィンドウから、[Instruction]、[Exception]、[Macro]を選択し、命令の動作記述、割込みの動作記述、またはマクロ定義を選択します。

<注意> 現行バージョンでは[Common]はサポートしていません。

また[Design Goal & Arch. Design]フェーズの[Processor Design], [Use Compiler]項目の指定により、[Micro Op. Description]における制限が異なります。以下にその制限を示します。

- [New Design]：制限なし
- [New Design with Base Processor]
 - [Use Compiler]が”yes”のとき
 - ◇ ベースプロセッサが持つ命令のマイクロ動作記述は編集不可
 - ◇ 追加命令のマイクロ動作記述は編集可能
 - [Use Compiler]が”no”のとき
 - ◇ 制限なし
- [Base Processor Design]：制限なし

9.1 [Micro Op. Description]ウィンドウ(命令動作定義)

[Micro Op. Description]ウィンドウから[Instruction]を選択すると、命令の動作記述ウィンドウがオープンします。

[Name]欄に[Instruction Definition]で定義した命令名の一覧が表示されます。命令名をひとつ選択すると、その命令の動作が右側に表示されます。初期状態では、何も表示されていないので、ここに動作を書いていくことになります。

[Stage]欄には[VARIABLE]と[Design Goal & Arch. Design]で定義したステージ名が表示されます。[Behavior Description]欄に、各ステージの動作を記述します。

[VARIABLE]欄には、各ステージで共通に使用する変数を宣言します。

<注意> 「マイクロ動作記述の書き方」を参照

マイクロ動作記述入力の際には、[Edit]メニューから、テキストの[Copy] (コピー), [Cut] (切り取り), [Paste] (貼り付け), [Undo] (やり直し) と [Redo] (やり直しのやり直し) および [CopyDescription] (マイクロ動作記述のコピー), [PasteDescription] (マイクロ動作記述の貼り付け) が可能です。

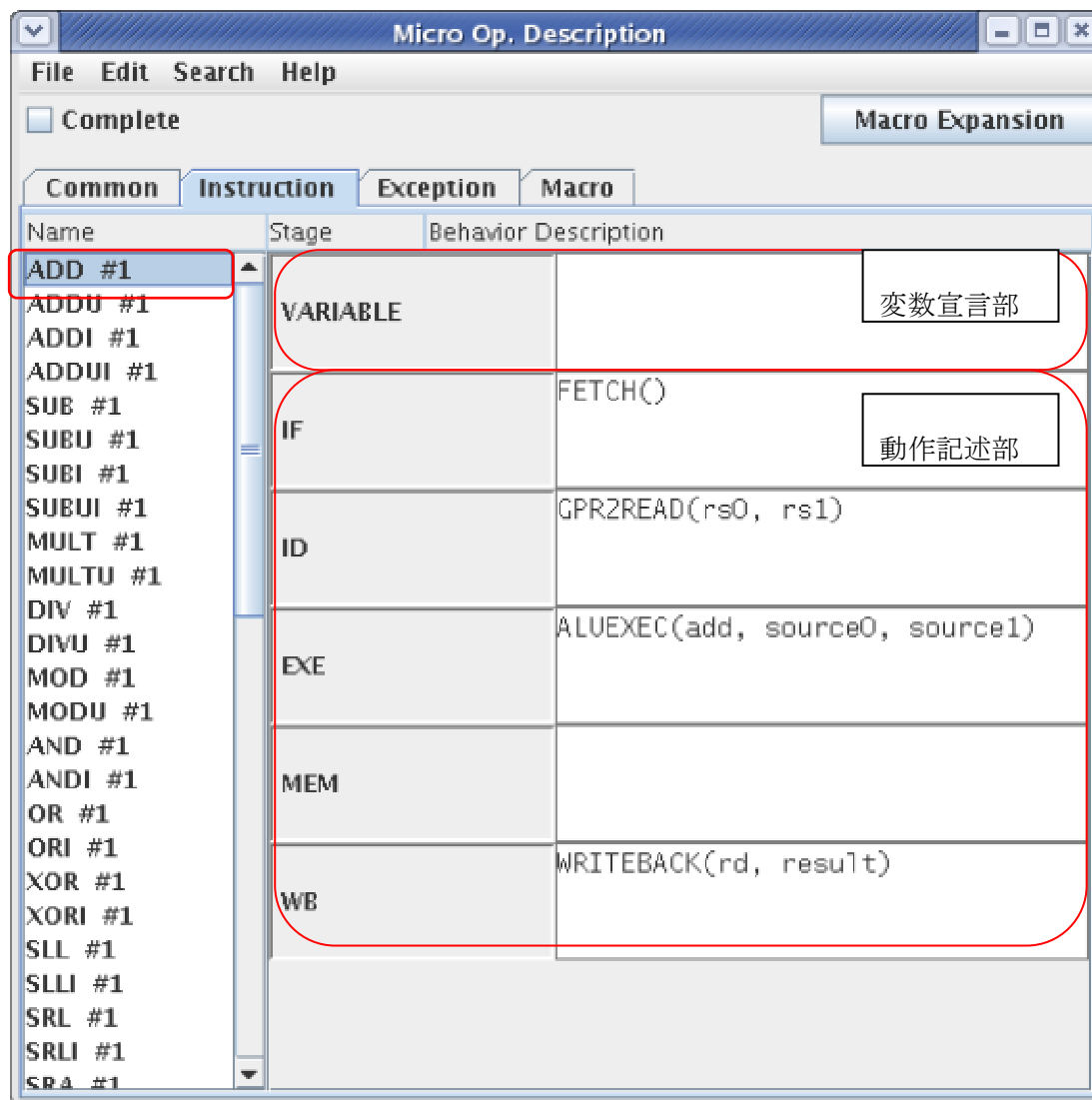


図32 [Micro Op. Description]ウィンドウ

9.2 [Micro Op. Description]ウィンドウ(割り込み動作定義)

[Exception]を選択すると、割り込み(例外)処理の記述ウィンドウがオープンします。

[Name]欄に[Instruction Definition]で定義した割り込み(例外)名の一覧が表示されます。割り込み(例外)名をひとつ選択すると、その割り込みの動作が右側に表示されます。

[Stage]欄には[VARIABLE]と[Design Goal & Arch. Design]で定義したステージ名が表示されます。[Behavior Description]欄に各ステージの動作を記述します。

[VARIABLE]には、各ステージで共通に使用する変数を宣言します。

<注意> 「マイクロ動作記述の書き方」を参照

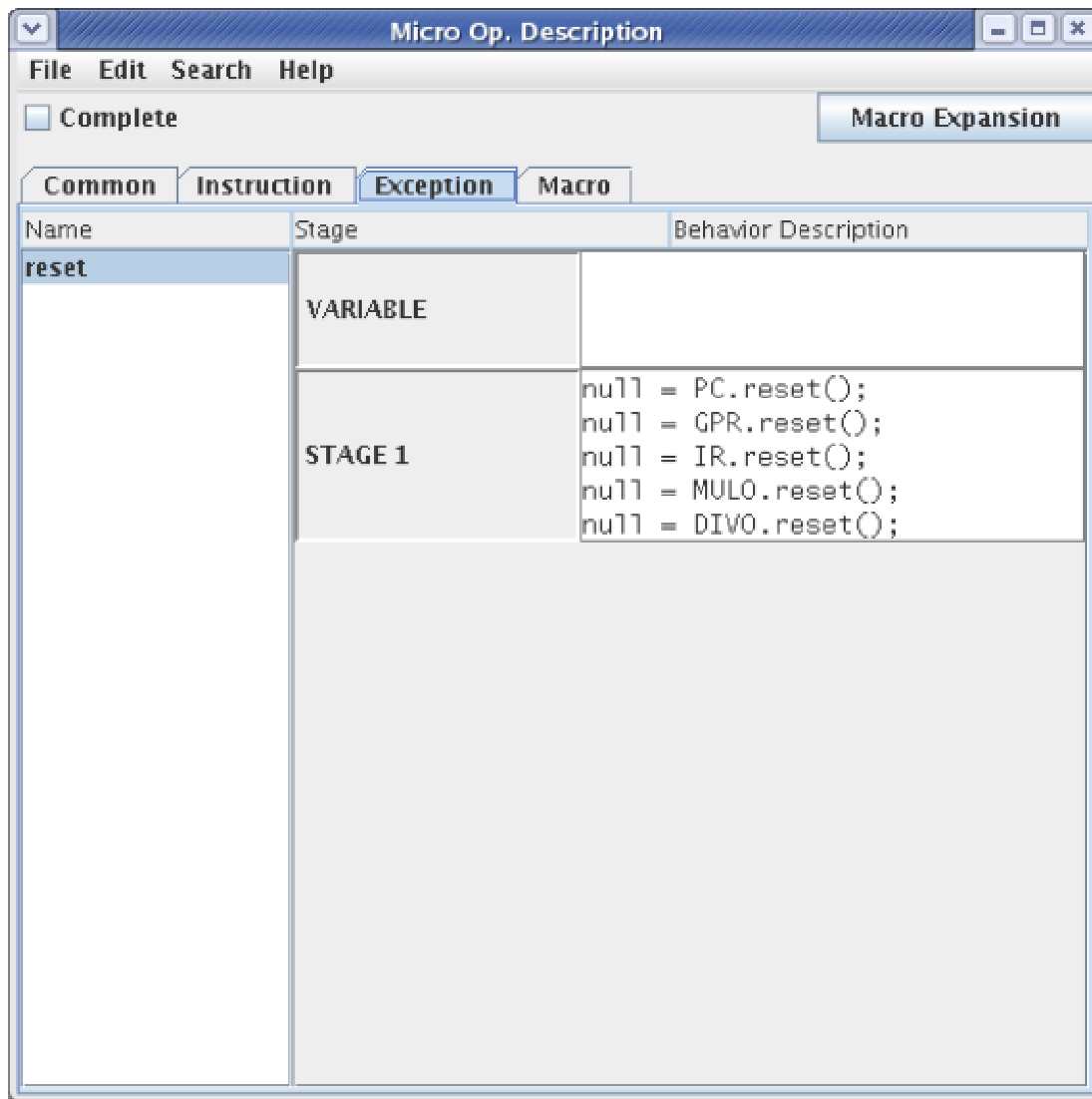


図33 [Micro Op. Description]ウィンドウ

9.3 [Micro Op. Description]ウィンドウ(マクロ定義)

複数命令に共通の動作について、マクロ定義をすることで、マイクロ動作記述を簡素化することができます。

[Macro]を選択すると、[New Macro Confirm]ウィンドウが開きますので、マクロの名前、引数、及びそのマクロのステージ数を入力し、[OK]を押します。



図34 [New Macro Confirm]ウィンドウ

[Micro Op. Description]ウィンドウが開きますので、命令のマイクロ動作記述と同様に動作を記述していきます。

<注意>「マイクロ動作記述の書き方」を参照

表16 [Micro Op. Description]ウィンドウ

パラメータ	意味
Macro name	マクロ名
args	引数
Num. of Stages	実行ステージ数
Behavior Description	マクロの動作記述

9.3.1 マクロ展開のルール

命令のマイクロ動作記述でマクロを呼び出して展開できます。マクロの呼び出しは次のような文字列で表されます。

マクロ名(引数 1, 引数 2, ..., 引数 n)

命令のマイクロ動作記述でマクロが展開された場合、次の記述部にマクロの記述が追加されます。

- 命令の各ステージに共通する変数宣言部
- マクロを呼び出したステージとその後のステージの変数宣言部
- マクロを呼び出したステージとその後のステージの実行部

マクロを呼び出したステージの後、どれだけのステージにマクロが展開されるかは、呼び出したマクロのステージ数に依ります。呼び出したマクロのステージ数が 1 であれば、マクロを呼び出したステージと命令の変数宣言部にマクロが展開されます。呼び出したマクロのステージ数が 2 であれば、マクロを呼び出したステージと次のステージおよび、命令の変数宣言部にマクロが展開されます。

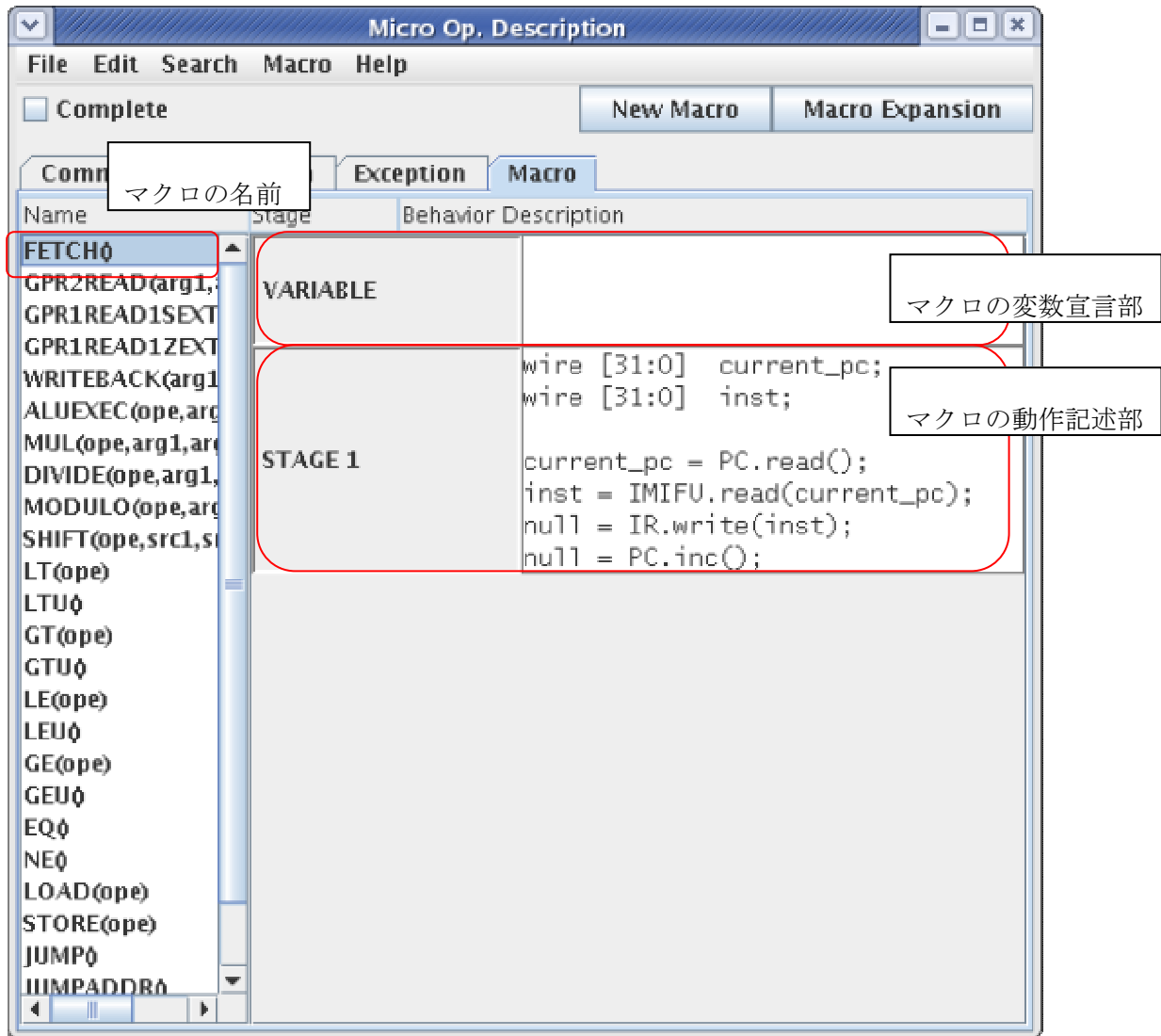


図35 [Micro Op. Description]ウィンドウ

9.4 [Macro Expansion]: マクロ展開の確認

命令や、割り込み、マクロの動作記述でマクロを使用している場合、[Macro Expansion]を押すと、該当するマクロが展開され、マクロを使って記述した動作を細かく確認することができます。尚、仮引数文字列は与えられた実引数文字列で置き換えます。

10. HDL Generation

メインウィンドウで[HDL Generation]をクリックすると、図 36の[Generation Confirm]ウィンドウが開きます。

[HDL Generation]では、プロセッサの HDL 記述を生成します。機能検証・動作検証を行うためのシミュレーションモデル、及び論理合成モデルの HDL 記述を自動生成することができます。各リソースの HDL 記述は[Resource Declaration]の[Description Style of HDL]で指定された抽象度の記述となります。

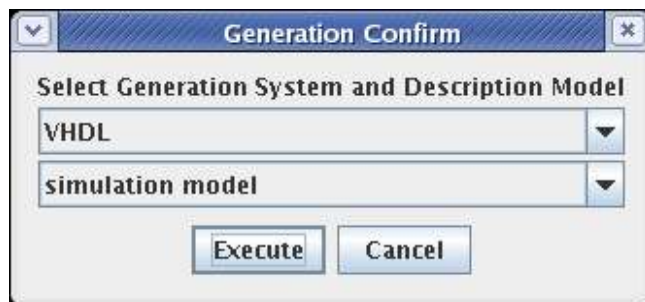


図36 [Generation Confirm]ウィンドウ



図37 [Generation Confirm]ウィンドウ

[Generation Confirm]ウィンドウにおいて、次の自動生成エンジン選択肢の中からひとつを選択して[Execute]を押すと、HDL 記述の自動生成が行われます。

1. [simulation model]: シミュレーションモデルのみの生成
2. [synthesizable model]: 論理合成モデルのみの生成
3. [sim. model and syn. model]: シミュレーションモデルと論理合成モデルの生成

HDL 記述の生成が完了すると、図 38のウィンドウが開きます。エラーが無いことを確認し、[OK] ボタンを押すとメインウィンドウに戻ります。

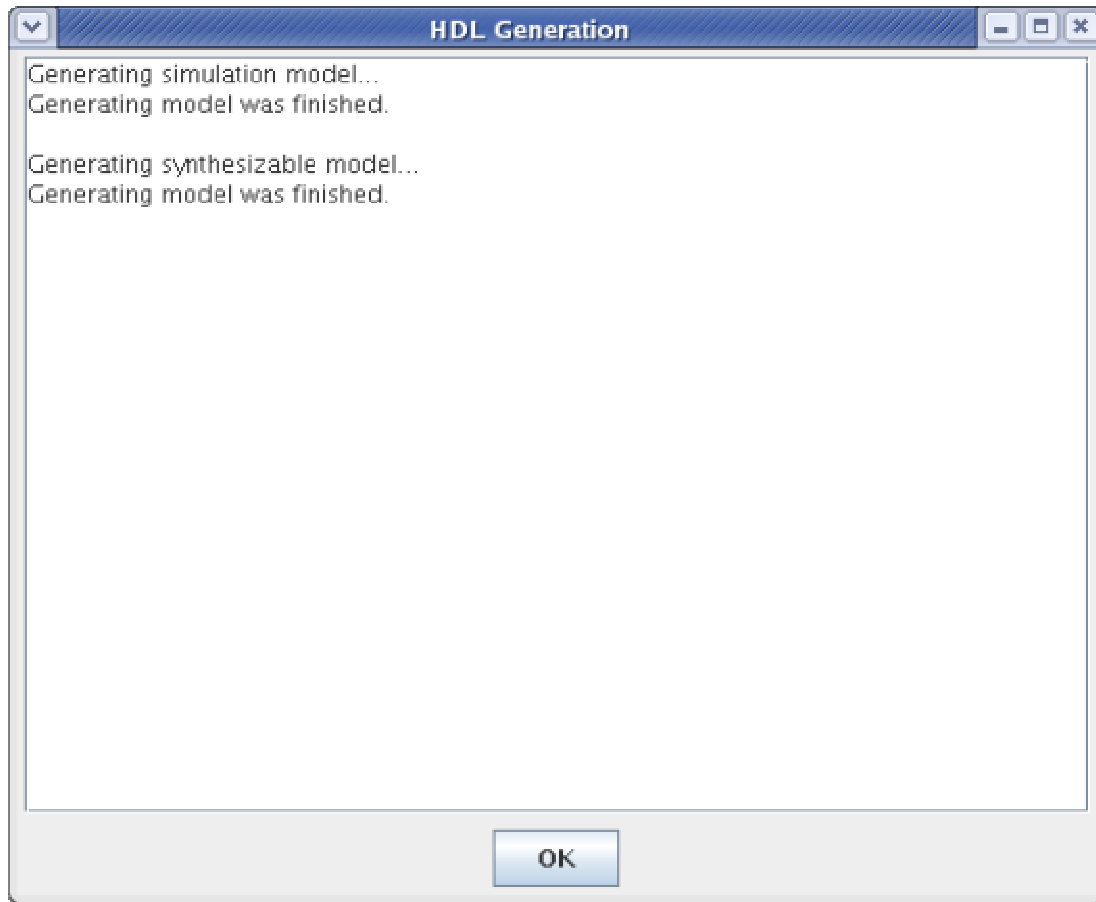


図38 HDL 記述生成完了ウィンドウ

HDL 記述の自動生成が正常に終了すると、カレントディレクトリの“meister”ディレクトリの下に、シミュレーションモデル用ディレクトリと論理合成モデル用ディレクトリが作成され、各々のディレクトリの下にファイルが作成されます。設計データファイル名が“model.pdb”だった場合、シミュレーションモデル記述は“meister/model.**lang**.sim/”ディレクトリ下に、論理合成モデルは“meister/model.**lang**.syn/”ディレクトリ下に生成されます。“**lang**”は“VHDL”もしくは“Verilog”で設定に依存します。

各ディレクトリにはプロセッサの HDL 記述が保存されます。[Design Goal & Arch. Design] で設定した“エンティティ名.vhd”、もしくは“エンティティ名.v”のファイルがプロセッサの最上位階層の記述となります。

出力言語として VHDL を選択した場合のファイルの生成例をファイルの生成例を図 39から図 40に示します。



図39 (例)シミュレーションモデルが生成されたディレクトリのファイル一覧

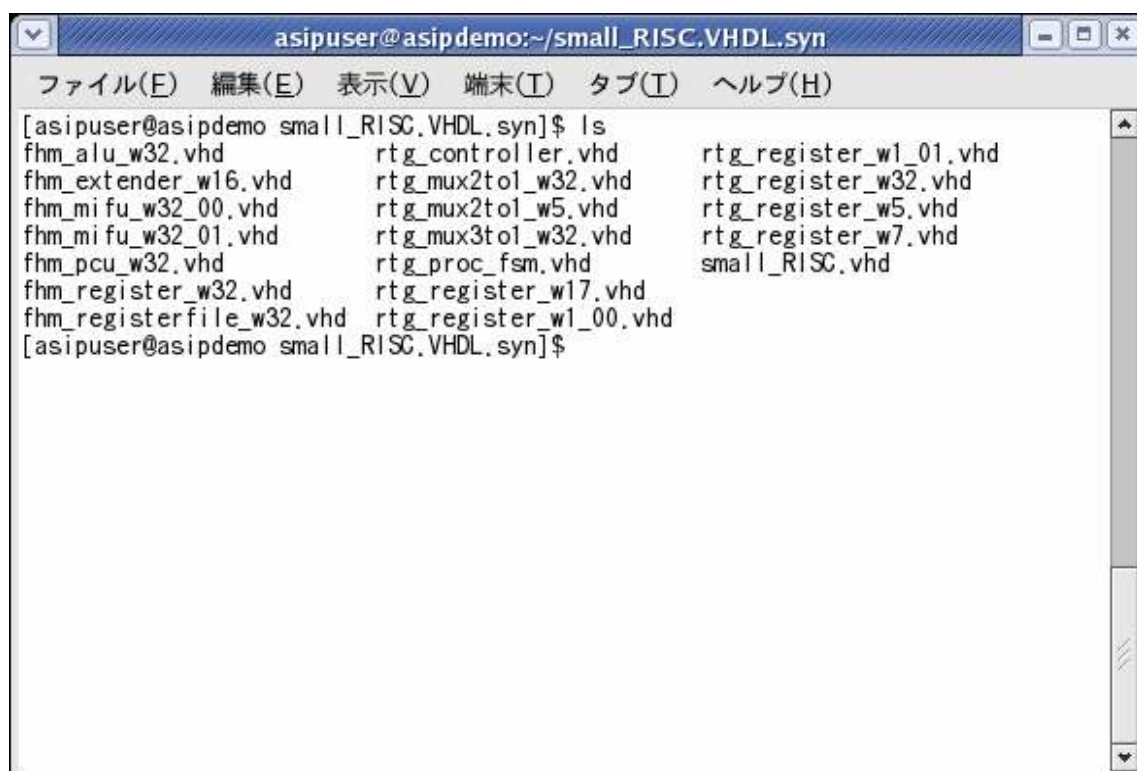


図40 (例)論理合成モデルが生成されたディレクトリのファイル一覧

11. C Definition

メインウィンドウで[C Definition]をクリックすると、図 41に示す[C Definition]ウィンドウが開きます。

ASIP Meister のコンパイラ生成は、ベースプロセッサ **Brownie** に対応しています。そこで、ベースプロセッサ **Brownie** から拡張した命令をコンパイラが生成できるように定義する必要があります。

[C Definition]では、**ASIP Meister** のコンパイラが対応する C 記述の仕様とベースプロセッサ **Brownie** から拡張した命令に対応する C 記述の関数を定義します。

<参考> Brownie の仕様書

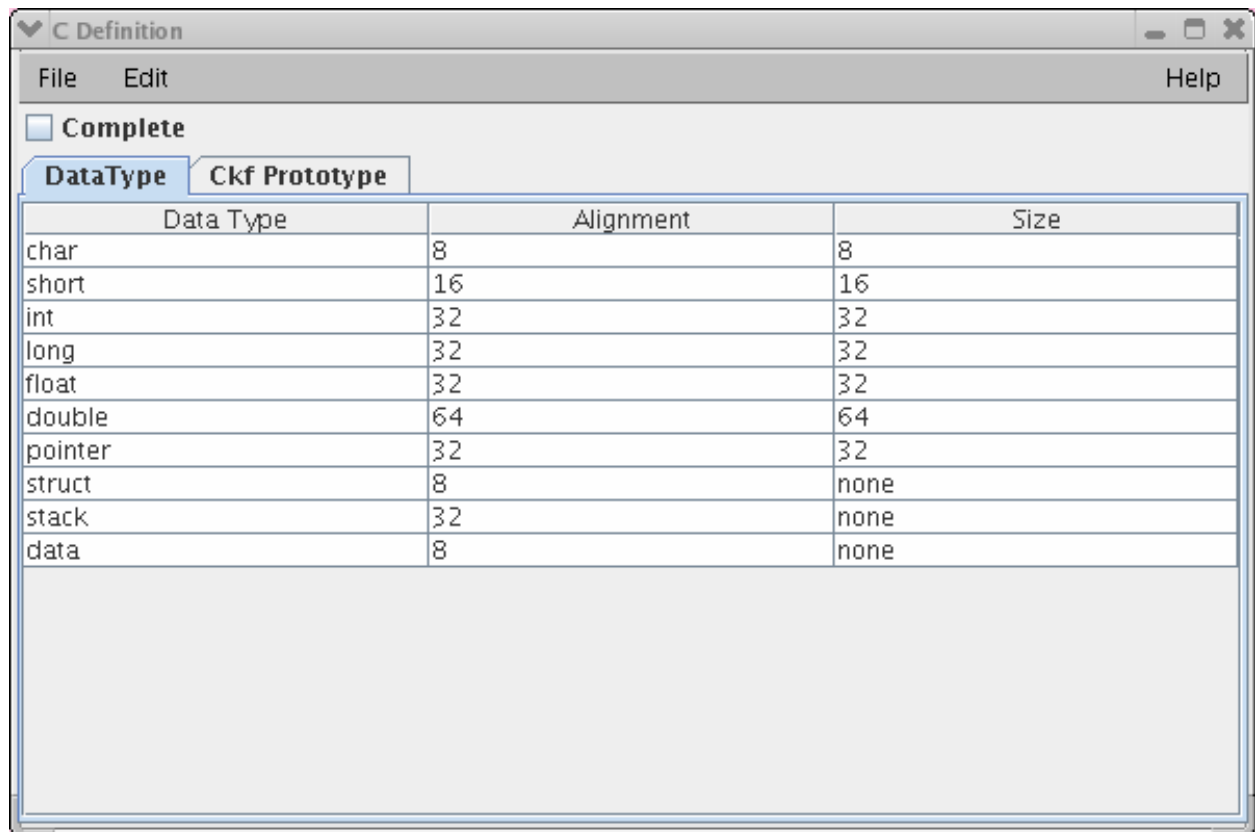


図41 [C Definition]ウィンドウ

[C definition]において、[DataType]タブでは C 言語の変数の型に対応するメモリ使用領域制約とビット幅を定義し、[Ckf Prototype]タブで、拡張命令に対応する関数を定義します。

11.1 [DataType]タブ

[DataType]タブ(図 41)では、使用する変数の型が Data Type 欄に記述されており、それに対して、メモリ使用領域制約(Alignment)とビット幅(Size)を定義します。

それぞれの型の一覧を表 17に示します。

表17 DataType 一覧

型名	用途	デフォルト値	
		Alignment	Size
Char	1 バイトの整数型	8	8
Short	2 バイトの整数型	16	16
Int	4 バイトの整数型	32	32
Long	4 バイトの整数型	32	32
Float	4 バイトの実数型	32	32
double	8 バイトの実数型	64	64
pointer	ポインタ	32	32
Struct	構造体	8	none
Stack	スタック	32	none
Data	初期代入の定数	8	none

メモリはバイトアドレスを想定しているため、**Alignment** によって使用できるメモリ領域が異なります。**Alignment** が 8 の場合は、全てのアドレスを使用できますが、16, 32, 64 では、変数の先頭アドレスが、それぞれ 2 の倍数, 4 の倍数, 8 の倍数の領域しか使用できません。

<注意> 現バージョンでは、DataType タブの情報は編集できません。

11.2 [Ckf Prototype]タブ

[Ckf Prototype]タブでは、C 言語で定義した関数に対応する拡張命令を宣言します。図 42に[C Definition]ウインドウの[Ckf Prototype]タブを示します。

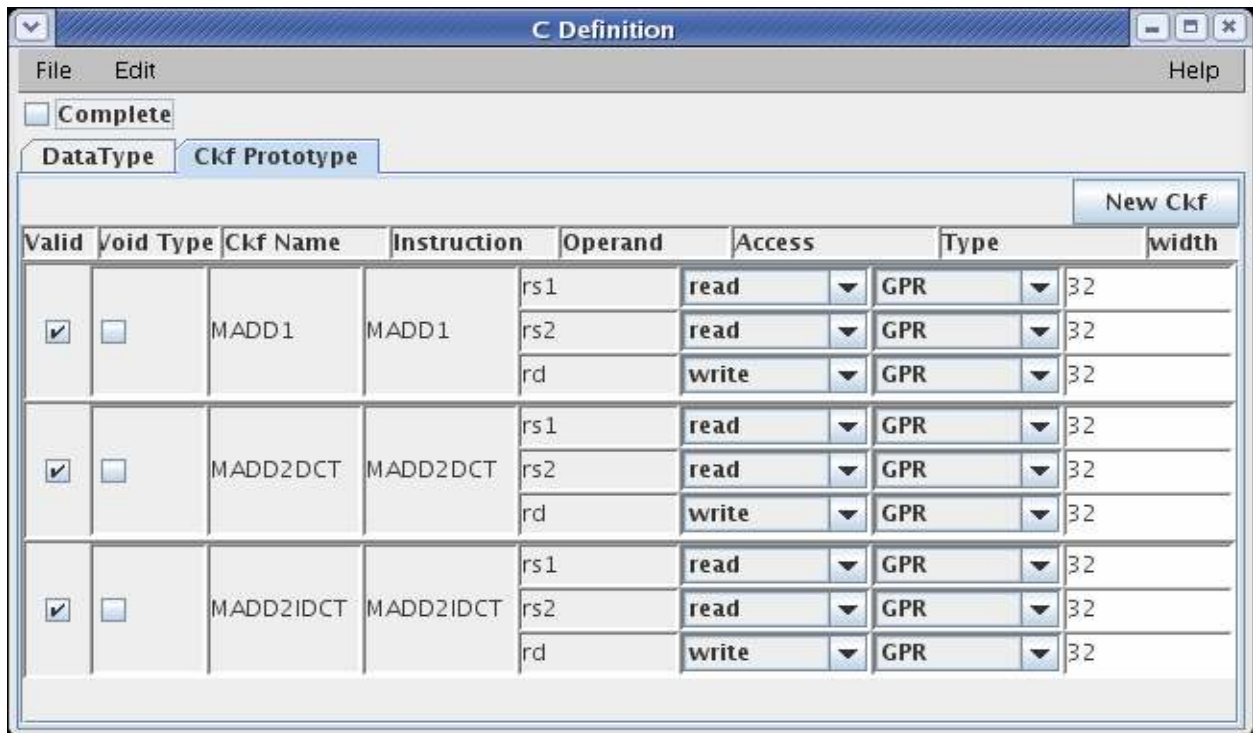


図42 [Ckf Prototype]タブ

初期状態では、Ckfは登録されていません。そこで、[New Ckf]を選択することで、Ckfを登録することが出来ます。[New Ckf]を選択すると、図 43に示す[New CKF]ウインドウが開きます。

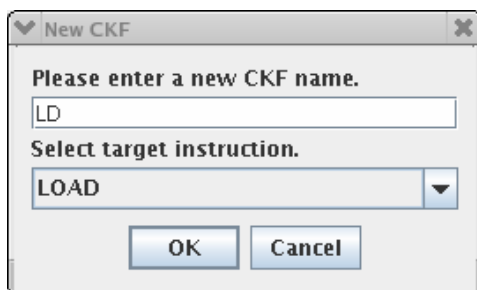


図43 [New CKF]ウインドウ

[New CKF]ウインドウで、関数名を上の入力欄に入力し、対応する命令を選択し、OK を選択します。すると、CKF が登録されます。登録された CKF は表 18に示す情報を持ちます。

表18 CKF で選択する情報

項目	意味	デフォルト値
Valid	Ckf を使用するか判定	チェック
Void Type	返り値の有無	チェックなし
Ckf Name	Ckf の名前	[New CKF]で入力した名前

Instruction	Ckf に対応する命令	[New CKF]で選択した命令
以下オペランドごとに記述		
Operand	命令のオペランド	[Instruction Definition]で定義
Access	関数への入出力	Read
Type	オペランドの形式	GPR または signed
Width	オペランドのビット幅	[Instruction Definition]で定義

Valid : Ckf を使用するか判定

Ckf を使用するか決定します。[Valid]をチェックしなければ、生成するコンパイラはこの Ckf に対応しません。

Void Type : 返り値の有無

Ckf の返り値が存在するかを表します。拡張命令の”Operand”で”Access”に”Write”が指定してあり、その”Type”が”GPR”で、さらに Access に”Write”が指定してある”Operand”の数が 1 つのみであるとき、チェックしません。それ以外の場合は、必ずチェックする必要があります。チェックする場合、出力オペランドは CKF において参照渡し引数として扱われます。Ckf1 という名前の CKF を定義した場合に、”Void Type”をチェックしない場合とチェックする場合の C 言語記述例を以下に示します。出力オペランドに対応する引数が先に現れ、その後に入力オペランドに対応する引数が続きます。出力オペランド同士、入力オペランド同士の引数の順番は、”Operand”タブの上から下に表示されている順で登録されます。

<チェックしない場合>

```
A = __builtin_brownie32_Ckf1(B, C);
```

<チェックする場合>

```
__builtin_brownie32_Ckf1(A, B, C);
```

<注意> 現バージョンでは拡張命令の前に”__builtin_brownie32_”をつける必要があります。

Ckf Name : Ckf の名前

Ckf の名前です。[New CKF]ウインドウで登録した名前が表示されます。登録されている Ckf の名前は変更することが出来ません。Ckf の名前を変更したい場合は、一旦対象の Ckf を削除してから、新たに Ckf を登録しなおす必要があります。

Instruction : Ckf に対応する命令

Ckf に対応する命令です。Ckf Name と同様に一度設定した命令は変更することができません。変更する場合は、新たに Ckf を登録する必要があります。

Operand : 命令のオペランド

命令のオペランドです。[Instruction Definition]において、”Field Type”を”Operand”とした Field が全て表示されます。

Access : オペランドのアクセス方向

オペランドのアクセス方向を表します。Ckf の入力変数とする場合は”read”を選択し、Ckf の出力変数とする場合は”write”を選択します。

Type：オペランドのタイプ

オペランドのタイプを表します。[Instruction Definition]において、“Addr mode”を”Reg Direct”にした場合は、“**GPR**”で固定です。“Addr mode”を”Immediate data”にした場合は、“**signed**”と“**unsigned**”を選択できます。

Width：オペランドのビット幅

オペランドのビット幅を表します。Type が”**GPR**”の場合は、GPR リソースのビット幅がデフォルトとして代入され、Type が”**signed**”や”**unsigned**”の場合は、[Instruction Definition]での Field のビット幅がデフォルト値として代入されます。

Delete CKF(メニュー)

Ckf を削除します。削除する Ckf は、現在 Mark している Ckf です。

Mark(メニュー)

Ckf を選択します。Mark すると Ckf 名が青くなり、“**Delete**”などの機能を使用できます。

Mark Clear(メニュー)

Mark を解除します。

<注意> Brownie の拡張命令のマイクロ動作記述は、すでに定義されている必要があります。

12. Compiler Generation

[Compiler Generation]では、プロセッサのコンパイラ, Binutils を生成します。生成されたコンパイラ, Binutils は、[C Definition]で宣言した Ckf に対応します。[Compiler Generation]を選択すると、図 44の[Generation Confirm]ウインドウが表示されます。

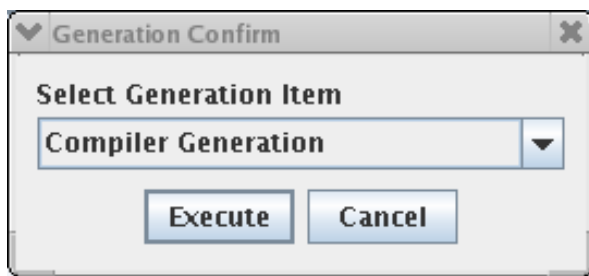


図44 [Generation Confirm]ウインドウ

[Generation Confirm]ウインドウにおいて、次の自動生成エンジン選択肢の中からひとつを選択して[Execute]を押すと、コンパイラ関連の自動生成が行われます。

- Input Description Generation：コンパイラ拡張記述の生成
- GNU Tools Generation：コンパイラ拡張記述からコンパイラ, binutils の生成

つまり、[Compiler Generation]フェーズでのコンパイラ, binutils の生成手順は、

1. Input Description Generation を選択実行
2. GNU Tools Generation を選択実行

となります。

[Input Description Generation]を選択すると、コンパイラ拡張記述が生成されます。生成が正常に終了すると、設計データファイル名が”model.pdb”だった場合、”meister”の中に”model.swgen”というフォルダができ、その中に”model.xml”というコンパイラ拡張記述が生成されます。

<補足>コンパイラ拡張記述を元にコンパイラや binutils を生成しますので、[GNU Tools Generation]の前に[Input Description Generation]を選択する必要があります。

<注意>インストールの際に作成した応用プログラム開發生成ツールのディレクトリ場所を、環境変数 ASIP_APDEV_SRCROOT として設定していないと正しくコンパイラは作成されません。

[GNU Tools Generation] を 選 択 後 , 生 成 が 正 常 に 終 了 す る と 、 環 境 変 数 ASIP_GNU_INSTALL_DIR で指定されたフォルダ以下にコンパイラ, Binutils のファイルが生成されます。環境変数 ASIP_GNU_INSTALL_DIR が設定されていない場合は、meister 以下に”model.swgen”フォルダが作成され、その下にコンパイラ, Binutils のファイルが作成されます。

以下に生成されたコンパイラを使用して、C プログラム”sample.c”をクロスコンパイルする方法を

示します。

まず、環境変数\$PATH に生成された”**model.swgen/bin**”を追加します。

1. \$ brownie32-elf-gcc -S sample.c -o sample.s
2. \$ brownie32-elf-as -o sample.o sample.s
3. \$ brownie32-elf-ld -o sample -T link.sc sample-a.o

以上のコマンドを順に入力することで、プロセッサのオブジェクトコード **sample** が生成されます。

<補足 1> link.sc はリンカスクリプトと呼ばれ、メモリマップを記述するファイルです。記述の参考例を以下に示します。

```
-----link.sc の中身-----
SECTIONS
{
    . = 0x0;
    .text : { *(.text) }
    . = 0x100000000
    .data : { *(.data) }
    .bss : { *(.bss) }
}
----- link.sc file end -----
```

上記の記述は、プログラムが 0x0 番地から、データが 0x100000000 番地から配置されるように指定しています。

<補足 2> 拡張命令の C コードを記述する際の注意点として、ckf に”__builtin_brownie32_”を追加して記述する必要があります。以下に例を示します。

例) : ckf : MAC, 引数 : 3 個, 出力 : 1 個の場合.

C 記述 : y = __builtin_brownie32_MAC(a, b, c);

このように記述しないと、拡張命令を正しくコンパイルできないので注意してください。

付録

ここでは、次の項目について説明します。

1. 設計可能なプロセッサモデル
2. 使用上の制限
3. マイクロ動作記述の書き方
4. 予約語
5. 外部インタフェース仕様
6. 割込み記述方法
7. PAS メタアセンブラの使用法
8. 登録されているリソースモデルとパラメータ

1. 設計可能なプロセッサモデル

ここでは、**ASIP Meister** で設計することができるプロセッサモデルを挙げています。**ASIP Meister** は次に示すモデルのプロセッサを設計することができます。

- パイプライン
- 固定長命令
- シングルサイクル命令フェッチ
- マルチサイクル命令
- In-Order 完了
- 構造ハザードによるインタロック(データハザードによるインタロックには未対応)
- ハーバード・アーキテクチャ
- 外部割込(単一レベル)
- 内部割込(例外)

これ以外のモデルについてはサポート部門までお問い合わせください。

1.1 制限事項

- 遅延分岐スロット中の命令は内部割込みを発生させてはいけません
- デコードステージより前のステージのマイクロ動作記述はすべての命令で一致させなければなりません
- 分岐ステージ (プログラムカウンタに直接値を書き込むステージ) は1つにしなければなりません
- 分岐ステージより前でプロセッサの状態を変えてはいけません
- 内部割り込みが発生するステージより前でプロセッサの状態を変えてはいけません
- 分岐ステージより前でマルチサイクル演算 (例: マルチサイクル命令フェッチ) を行ってはいけません
- 外部割り込みは最大1つでなければなりません
- リセット割り込みは必ず必要です

2. 使用上の制限

ASIP Meister はシングルユーザによる使用を前提としています。複数のユーザが使用する場合は、各自がインストールした **ASIP Meister** を使用するようにしてください。

3. マイクロ動作記述の記述方法

3.1 マイクロ動作記述の基礎

マイクロ動作では、[Resource Declaration]で指定されたリソースを用いて、パイプライン・ステージ単位に以下の動作を記述することができます。

- データ転送
- リソースを用いた処理(演算, 読み出し, 書き込み等)

データ転送は、記号 "=" を用いて表します。これは、一般的なプログラミング言語において、左辺に右辺の値を代入することと同様の概念です。リソースを用いた処理は、<リソース名> "." <関数名> "(" <入力の並び> ")" で表します。この<関数名>には、[Resource Declaration] ウィンドウで表示される "Function Set" で記述された関数を使用できます。以下に、モデル alu の "Function Set" の一部を示します。

```
model alu32{
  port {
    in a[31:0], b[31:0], cin, mode[4:0];
    out result[31:0], flag[3:0];
  }
}
```

-----<中略>-----

```
/** add : signed add, flag(3):C, flag(2):Z, flag(1):S, flag(0):V */
function add{
  input{
    unsigned a, b;
  }
  output{
    unsigned result = add(a,b);
    bit_vector flag = alu_flag(mode, a, b, cin);
  }
  control{
    in mode;
    in cin;
  }
  protocol{
    [mode == "01001" && cin == '0']{
      valid result;
      valid flag;
    }
  }
}
```

```
    }
  }
}
```

-----<中略>-----

```
}
```

例えば、モデル `alu` をインスタンス化したリソース `ALU` の `add` という関数を使用したい場合は、予め宣言された変数 `result`, `flag`, `a`, `b` を用いて

```
<result, flag> = ALU.add(a, b);
```

と記述します。ここでは、`a`, `b` は `ALU` への入力、`result`, `flag` は `ALU` からの出力となります。リソースへの入力は、プログラミング言語の関数呼び出しにおける引数と同様に考えることができます。リソースの入力を列挙する順序は、"Function Set" の "input" に現れるポート名の順序と対応しています。リソースからの出力が複数となる場合は、記号 "<>" を用いて変数を列挙します。列挙する順序は "Function Set" の "output" に現れるポート名の順序と対応しています。

3.2 マイクロ動作記述の構文要素

3.2.1 変数

マイクロ動作を記述する際、データの受け渡しには変数を用います。変数を用いる場合は、変数宣言部で予め宣言する必要があります。変数宣言の方式については付録の BNF を参照してください。マイクロ動作記述における変数は、次の 2 通りあります。

命令毎に宣言される変数

この変数は、複数のパイプライン・ステージにおいて用いられる変数です。つまり、パイプライン・レジスタとして機能します。

パイプライン・ステージ内で宣言される変数

この変数は、宣言されたステージでのみ用いられる変数です。つまり、リソース間のデータ転送を行うための信号線として機能します。

3.2.2 二進定数

マイクロ動作記述で利用できる定数は二進定数のみです。二進定数には、"`'`" で囲まれた 0 あるいは 1 で表されるビットリテラルと "`"`" で囲まれた 0 あるいは 1 の 2 回以上並びで表されるベクタリテラルがあります。1 ビットの定数はビットリテラルで表し、2 ビット以上の定数はベクタリテラルで表します。

3.2.3 接続

ビットの接続は変数の接続のみ可能で、"<>" を用いて表します。例えば、3 ビットの変数 `a`, 2 ビットの変数 `b`, 4 ビットの変数 `c` を接続して 9 ビットの変数 `d` として扱いたい場合は、

```
d = <a, b, c>;
```

と記述します。この時、変数 `d` の最上位ビットは `a` の最上位ビット、最下位ビットは `c` の最下位ビットとなります。

3.2.4 ビット演算子

ビット演算子には、AND, OR, 反転の 3 種類があります。被演算子は変数でなければなりません。

- ビット AND 演算
演算子 "&" を用います。

```
a = "010";
```

```
b = "110" ;
```

```
c = a & b;
```

この例では, c の値は "010" となります.

- ビット OR 演算

演算子 "|" を用います.

```
a = "010" ;
```

```
b = "110" ;
```

```
c = a | b;
```

この例では, c の値は "110" となります.

- ビット反転演算

演算子 "~" を用います.

```
a = "010" ;
```

```
b = ~a;
```

この例では, b の値は "101" となります.

3.2.5 関係演算子

ある変数と与えられる定数を比較する場合に用います. 次の 2 通りを使用することができます.

- 等号

```
b = a == "01" ;
```

この例では, a の値が "01" であれば b の値に '1' が, そうでなければ b の値に '0' が代入されます.

- 不等号

```
b = a != "01" ;
```

この例では, a の値が "01" であれば b の値に '0' が, そうでなければ b の値に '1' が代入されます.

3.2.6 ビット範囲指定

ある変数のうち抽出したいビット範囲を指定する場合に用います.

- 1 ビットを指定する場合

```
b = a[3];
```

この例では, 変数 a の第 3 ビットが b に代入されます. b は 1 ビットの変数として宣言されなければなりません.

- 2 ビット以上を指定する場合

```
b = a[3:1];
```

この例では, 変数 a の第 1 ビットから第 3 ビットが b に代入されます. b は 3 ビットの変数として宣言されなければなりません.

3.2.7 条件付き処理

条件付き処理は以下のように記述します.

- 条件付きデータ転送の場合

```
result = (condition) ? a : b ;
```

C 言語の三項演算子 "?" と同様に考えることができます. この例では, 条件変数 condition が '1' であれば a を '0' であれば b を result に代入します. なお, 条件変数は 1 ビットの

変数でなければなりません。

- 条件付きリソース機能実行の場合
以下に示すのは、条件変数 `condition` が '1' であれば リソース REG の `write` 関数を変数 `data` を入力として実行する、という例です。なお、条件変数は 1 ビットの変数でなければなりません。

```
null = [condition] REG.write(data);
```

3.3 マイクロ動作を記述する際の注意点

以下にマイクロ動作を記述する際の注意点を挙げます。

- [Design Goal & Arch. Design] で指定されたデコード・ステージより前のステージの動作は、全命令で共通してはなりません。
- リソース競合が発生する命令の組み合わせが存在する場合は、エラーになります。
- 命令タイプ定義で定義されたオペランドの名前は、その命令タイプを継承した命令において宣言済みの変数として扱われます。
- 変数宣言部とマイクロ動作記述部の間には、1 行の空行が必要となります。

3.4 マイクロ動作記述の BNF

<変数宣言部> ::= { <変数宣言> }

<変数宣言> ::= <ワイヤ宣言>

<ワイヤ宣言> ::= 'wire' [<ビットレンジ指定>] <ワイヤ名> ';'

<ワイヤ名> ::= <識別子>

<マイクロ動作記述部> ::= { <マイクロ動作記述> }

<マイクロ動作記述> ::= <ステージ変数宣言部> <文の並び>

<ステージ変数宣言部> ::= <変数宣言部>

<文の並び> ::= { <文> }

<文> ::= <単純代入文> |

<条件付き代入文> |

<条件付き機能実行文>

<単純代入文> ::= <左辺> '=' <右辺> ';'

<左辺> ::= <変数名> | <変数名集合> | 'null'

<右辺> ::= <ビット AND 演算式> | <ビット OR 演算式> | <ビット反転演算式> | <比較式> |
<集合体> | <リソース参照> | <部分選択式> |
<二進定数> | <変数参照>

<ビット AND 演算式> ::= <変数参照> '&' <変数参照>

<ビット OR 演算式> ::= <変数参照> '|' <変数参照>

<ビット反転演算式> ::= '~' <変数参照>

<比較式> ::= <変数参照> <関係演算子> <二進定数>

<関係演算子> ::= '==' | '!='

<変数参照> ::= <変数名>

<変数名> ::= <ワイヤ名>
 <変数名集合> ::= '<' <変数名> {'<' <変数名>}'>
 <集合体> ::= '<' <変数参照> {'<' <変数参照>}'>

 <リソース参照> ::= <リソース名> '!' <機能名> '(' [<パラメタの並び>] ')'
 <機能名> ::= <識別子>
 <パラメタの並び> ::= <パラメタ> {'<' <パラメタ>}'
 <パラメタ> ::= <変数参照>

 <部分選択式> ::= <変数参照> <部分選択子>
 <部分選択子> ::= '[' <添字> ':' <添字> ']'
 <添字> ::= <非負整数>

 <条件付き代入文> ::= <左辺> '=' '(' <条件変数参照> ')' '?' <変数参照> ':' <変数参照> ';' <条件変数参照> ::= <変数参照>
 <条件付き機能実行文> ::= <左辺> '=' '(' <条件変数参照> ')' <リソース機能指定> ';' <リソース機能指定> ::= <リソース名> '!' <機能名> '(' [<パラメタの並び>] ')'

 <識別子> ::= <英文字> {'<英数字> | '_' }
 <英文字> ::= <英大文字> | <英小文字>
 <英大文字> ::= 'A' | 'B' | 'C' | 'D' | 'E' |
 'F' | 'G' | 'H' | 'I' | 'J' |
 'K' | 'L' | 'M' | 'N' | 'O' |
 'P' | 'Q' | 'R' | 'S' | 'T' |
 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z'
 <英小文字> ::= 'a' | 'b' | 'c' | 'd' | 'e' |
 'f' | 'g' | 'h' | 'i' | 'j' |
 'k' | 'l' | 'm' | 'n' | 'o' |
 'p' | 'q' | 'r' | 's' | 't' |
 'u' | 'v' | 'w' | 'x' | 'y' | 'z'
 <英数字> ::= <英文字> | <数字>
 <数字> ::= '0' | <0 以外の数字>
 <0 以外の数字> ::= '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

 <自然数> ::= <0 以外の数字> {'<数字> }
 <非負整数> ::= '0' | <自然数>

 <二進定数> ::= <ビットリテラル> | <ベクタリテラル>
 <ビットリテラル> ::= ''' <二進文字> '''
 <ベクタリテラル> ::= ' ' <二進文字> {'<二進文字>}' ' ' <二進文字> ::= '0' | '1'

 <文字列> ::= "" { <改行以外の任意の文字> } ""
 <コメント部> ::= <C/C++のコメント構文と同じ (//から行末まで, または, /*から*/の間) >

4. 予約語

この節では ASIP Meister 設計ファイルおよび VHDL ファイルで使用できない予約語について説

明します。

4.1 ASIP Meister 設計ファイルの予約語

以下の語は、データファイル名、ストレージ名、リソース・インスタンス名、命令タイプ名、命令フィールド名、命令名、ターゲットプロセッサのエンティティ名、ポート名、割り込み及び例外名、マイクロ動作記述の変数名として使用できません。

active_value	catch_interrupt	cause_condition	cause_condition_type
clock_port	connect_to	data_memory	decode_error
decode_stage	design_level	direction	dont_care
exec_stage	external_interrupt	fetch_stage	flag_register
for_simulation	for_synthesis	in	inout
instr_memory	instr_register	instr_specific	instr_type
instruction	internal_controller	internal_interrupt	mask_bitpos
mask_condition	mask_register	mask_register	memory_read_stage
memory_write_stage		mod	model null
num_stages	opcode	operand	out
parameter	plain_register	port	program_counter
register_file	register_read_stage	register_write_stage	reserved
reset_interrupt	reset_port	resource	saved_pc
ds_offset	stage	status_register	throw
top_module	wire		

4.2 VHDL の予約語

以下の語はターゲットプロセッサのエンティティ名およびポート名として使用できません。

abs	access	after	alias	all
and	architecture	array	assert	attribute
begin	block	body	buffer	bus
case	component	configuration	constant	disconnect
downto	else	elsif	end	entityexit
file	for	function	generate	generic
group	guarded	if	impure	in
inertial	inout	is	label	library
linkage	literal	loop	map	mod
nand	new	next	nor	not
null	of	on	open	or
others	out	package	port	postponed
procedure	process	pure	range	record
register	reject	rem	report	return
rol	ror	select	severity	shared
signal	sla	sll	sra	srl
subtype	then	to	transport	type
unaffected	units	until	use	variable
wait when	while	with	xnor	
xor				

5. 外部インタフェース仕様

ASIP Meister の生成するプロセッサには、Resource Declaration フェーズで選択されたメモリインタフェースユニットが実装されます。本節では、メモリインタフェースユニットと外部インタフェースの仕様について説明します。

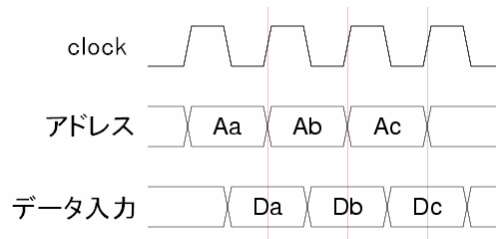
5.1 mifu を命令メモリアクセスユニットとして使用する場合

mifu はシングルサイクルアクセス/マルチサイクルアクセスの両方に対応しています。mifu をシングルサイクルの命令メモリアクセスユニットとして使用する場合は、以下のポートを Interface Definition フェーズで定義する必要があります。

用途	入出力方向	属性名
アドレス	out	instruction_memory_address_bus
CPU へのデータ入力	in	instruction_memory_data_in_bus

動作条件は次のとおりです。

- データ入力ポートには、次のクロックの立ち上がりまでにアドレスが示すデータが与えられる必要があります(図参照)。



5.2 mifu をデータメモリアクセスユニットとして使用する場合

mifu をマルチサイクルのデータメモリアクセスユニットとして使用する場合は、以下のポートを Interface Definition フェーズで定義する必要があります。属性名の*は、命令メモリの場合 instruction、データメモリの場合 data となります。

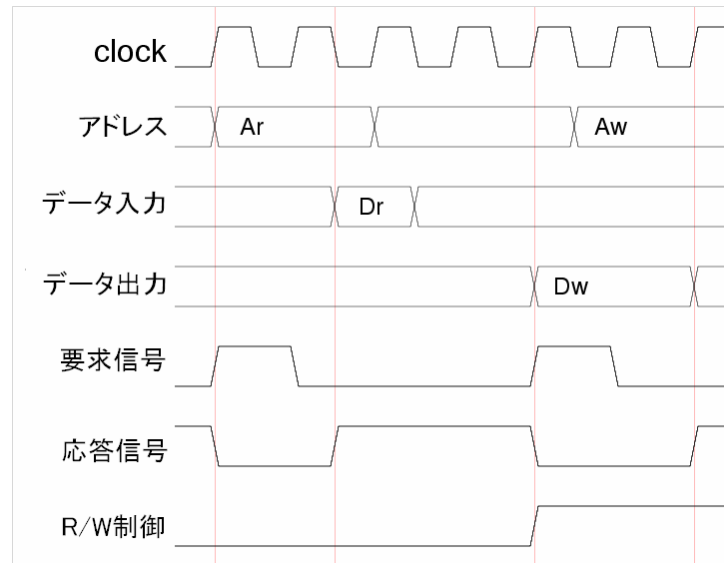
用途	入出力方向	属性名
アドレス	out	*_memory_address_bus
CPU へのデータ入力	in	*_memory_data_in_bus
CPU からのデータ出力	out	*_memory_data_out_bus
CPU からのアクセス要求	out	*_memory_request_bus
メモリからの応答	in	*_memory_acknowledge_bus
書き込み/読み込み制御	out	*_memory_rw_bus
アクセスキャンセル信号	out	*_memory_cancel_bus

動作条件は以下のとおりです。

- アクセス要求信号が '1' となったとき、メモリアクセスを開始する必要があります。アドレス、CPU へのデータ入力、書き込み/読み込み制御信号は、応答信号が得られるまで値が保持されま

す(図参照)。

- 書き込み/読み込み制御信号が‘0’の場合は読み込み、‘1’の場合は書き込み処理を行う必要があります。
- アクセス要求信号は 1 サイクルの間 ‘1’ となります。応答信号が ‘1’ となる前に、要求信号が再び ‘1’ となった場合は、前のアクセスをキャンセルし、再度アクセスを処理する必要があります。
- メモリからの応答信号は、アクセス中および要求信号が ‘1’ の間は ‘0’、それ以外は ‘1’ となっている必要があります(図参照)。
- アクセスキャンセル信号が ‘1’ となった場合、処理中のアクセスをキャンセルする必要があります。



また、mifu のパラメタ `access_width` の値が `bit_width` の値よりも小さい場合、次のポートが必要となります。

用途	入出力方向	属性名
アクセスビット数制御	out	*_memory_access_mode_bus
読み込み時ゼロ/符号拡張制御	out	*_memory_ext_mode_bus

動作条件は以下のとおりです。

- アクセスビット数制御信号は、 $(\frac{bit_width}{access_width} - 1)$ を二進数で表した値が出力されます。例えば、`bit_width` が 32 ビット、`access_width` が 8 ビットの時、アクセスビット制御信号の値が “11” の場合は 32 ビットアクセス、“01” の場合は 16 ビットアクセス、“00” の場合は 8 ビットアクセスを行います。書き込みの場合、有効データはデータ入力信号の下位ビットに格納されているとします。
- ゼロ/符号拡張制御信号は、読み込みビット数が `bit_width` よりも小さい場合、データを符号拡張するかを示します。‘0’ の場合はゼロ拡張、‘1’ の場合は符号拡張したデータが渡される必要があります。

mifu の機能は、アドレスエラーを出力として提供します。次のポートを定義すると、mifu の機能使用時のアドレスエラー出力が有効となります。ポートを宣言しない場合、アドレスエラー出力は不定値となります。

用途	入出力方向	属性名
アドレス不正信号	in	*_memory_address_error_bus

動作条件は以下のとおりです。

- アドレス不正信号は、アドレスが不正の場合に‘1’となる必要があります。

6. 割込み記述方法

6.1 想定割込みモデル

現バージョンの ASIP Meister では、次の割込みモデルを想定しています。

- 対応する割込みの種類
 - リセット割込み
 - NMI(ノン・マスカブル割込み)
 - 内部割込み
 - 外部割込み
 - ◇ 割込みのマスクに対応（内部割込み，外部割込み）
 - ◇ 割込みコントローラをプロセッサ外部に持つ事を想定（外部割込みは 1 系統のみサポート）
 - ◇ 多重割込みにはソフトウェア側で対応する（ハードウェアとしては非対応）

割込みの優先順位について、す。

表 19 にまとめます。表中、N はパイプラインステージ数を表わします。優先度の高い割込みと低い割込みが同時に発生した場合、優先度の低い割込みはキャンセルされ、優先度の高い割込みのみが処理されます。

表 19 割込みの優先順位

優先度	検出ステージ	名称
高	N	リセット割込み
：	N	NMI(ノン・マスカブル割込み)
：	N	ステージ N の内部割込み
：	N-1	ステージ N-1 の内部割込み
：	：	：
：	2	ステージ 2 の内部割込み
：	1	ステージ 1 の内部割込み
低	1	外部割込み

図 45に外部の割込コントローラとの推奨接続関係図を示します。

外部割り込みは 1 系統のみで、複数の外部割り込みがあるシステムの場合には、プロセッサの外に割込みコントローラ(図中 Interrupt Controller)を設けることを想定しています。外部割り込みが発生した場合には、割込みコントローラは図中の interrupt 信号をアサートし、catch 信号が上がる

までアサートし続けます。プロセッサ(図中 Processor)は interrupt 信号がアサートされたことを検知し、割込み処理待ちの状態に遷移します。この状態では、パイプライン中に残っている命令の実行が完了するまで待ち、パイプライン中のすべての命令が完了し、割込み処理が行う準備が整ったら、1 サイクルの割込み処理状態に遷移します。この状態において、プロセッサは、catch 信号を 1 サイクルの間アサートし、割込みコントローラに割込みの受理を通知します。同時に割込みハンドラの存在するアドレスにジャンプし、次のサイクルから割込みハンドラの処理を開始します。割込みハンドラの処理が終了したら、end_of_interrupt 信号を 1 サイクルの間アサートし、割込みコントローラに割込みハンドラの終了を通知します。複数の外部割込みが存在する場合、汎用データベースを介して、メモリマップされた割込みコントローラから割込みの種類などの情報を取得することもできます。また、割込みコントローラ内に、外部割込み間の優先度やマスク設定の機能がある場合、同様に汎用データベースを用いてこれにアクセスすることができます。なお、リセット割込み、NMI 割込みの場合には、パイプライン中の命令の実行完了を待たずに割込み処理に移ります。

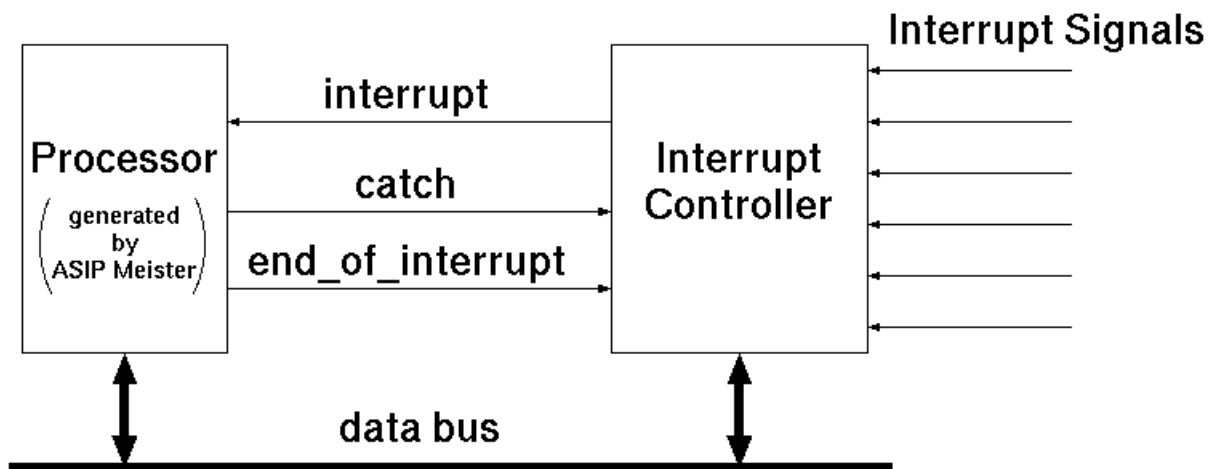


図45 割込みコントローラとプロセッサの接続関係

6.2 内部割込みの概要

図 46に、内部割込み発生時のタイミングチャートを示します。内部割込み信号を検知した命令の後続命令は破棄され、先行命令の処理が終了するまで待ちます。パイプラインが空になった次のサイクルで、発生した割込みの種類に応じて、マイクロ動作記述で記述した割込み処理が実行されます。割込み処理中で適切なアドレスにジャンプするなどして、割込みハンドラの処理(図中、命令 w~z)にうつります。図は、命令 h が、サイクル 5 に第 2 ステージにおいてデコードエラー例外を発生(図中 H)させている例です。例外発生により後続命令 i は破棄され、先行命令の実行が終了し、パイプラインが空になった時点で、デコードエラー例外の割込み処理(図中 D)を行います。なお、割込み処理(図中 D)では、内部割込みが発生した命令 h のアドレスを取得することができます。

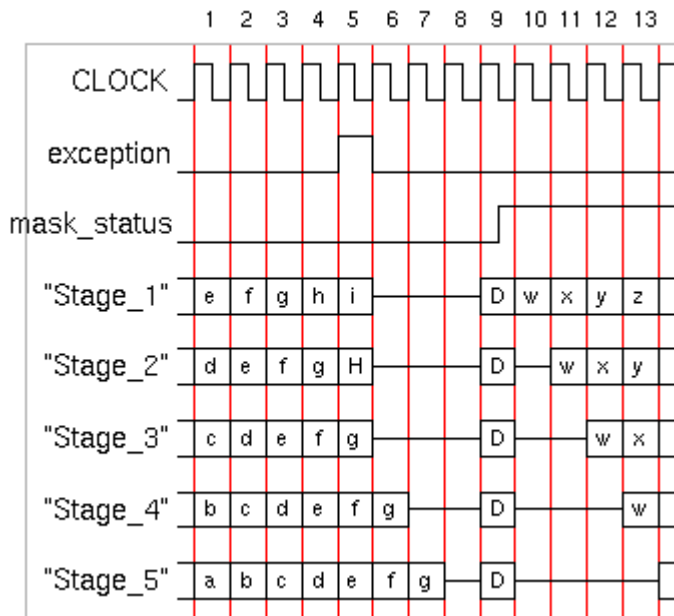


図46 内部割込みのタイミングチャート

6.3 外部割込みの概要

図 47に、外部割込み発生時のタイミングチャートを示します。外部割込み信号を検知した次のサイクルから新規の命令フェッチは停止されます。また、外部割込みを検知したサイクルに第 1 ステージに存在していた命令も破棄されます。パイプラインが空になった次のサイクルで、マイクロ動作記述で記述した外部割込みの割込み処理が実行されます（第 1 ステージで内部割込みが発生した場合と同じタイミングです）。**catch** 信号をアサートする記述をマイクロ動作記述に含めておくことで、割込みコントローラに割込みの受理を通知することができます。（少なくともこのサイクルまで、外部割込み信号 **interrupt** はアサートされている必要があります。実行完了を待っている命令が分岐や内部割込みを起こした場合には、割込みの受理が延期されますので、割込みを取りこぼさないためには、**catch** 信号がアサートされるまで **interrupt** 信号はアサートされている必要があります。）割込み処理では、適切なアドレスにジャンプし、割込みハンドラの処理（図中、命令 **w**～**z**）にうつります。図中、命令 **z** は、割込みハンドラ終了命令であり、ステージ 3 において **end_of_interrupt** 信号をアサートし、割込み処理時に保存してあった PC の値にジャンプし、割込みによって中断していた命令シーケンスを再開します。なお、割込み処理(図中サイクル 8)では、外部割込みが発生した時点のアドレス(命令 **g** のアドレス)を取得することができますので、これを適切な場所に保存する処理をマイクロ動作で記述する必要があります。

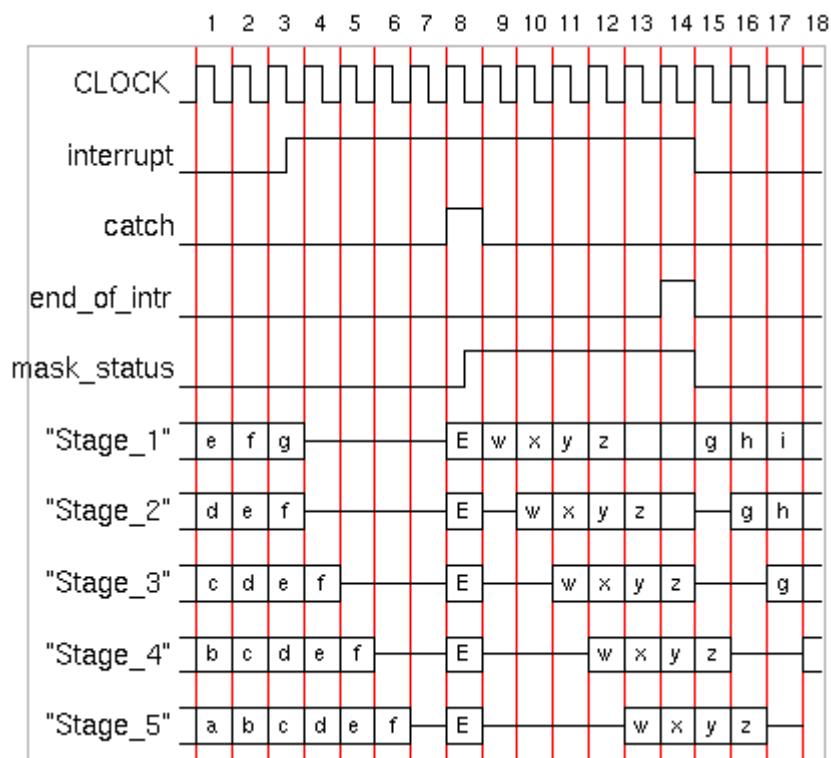


図47 外部割込みのタイミングチャート

6.4 NMI 割込みの概要

図 48に、NMI 発生時のタイミングチャートを示します。NMI 信号はクロックエッジで検出され、次のサイクルで NMI の割込み処理が実行されます。NMI 信号は、1 ないし 2 サイクルの間アサートされるべきであり、3 サイクル以上アサートされていると、再び NMI の割込み処理が実行されます。図中、サイクル 3 にアサートされた NMI 信号により、命令 c-g はすべて破棄され、次のサイクル(サイクル 4)に NMI の割込み処理 N が実行されます。

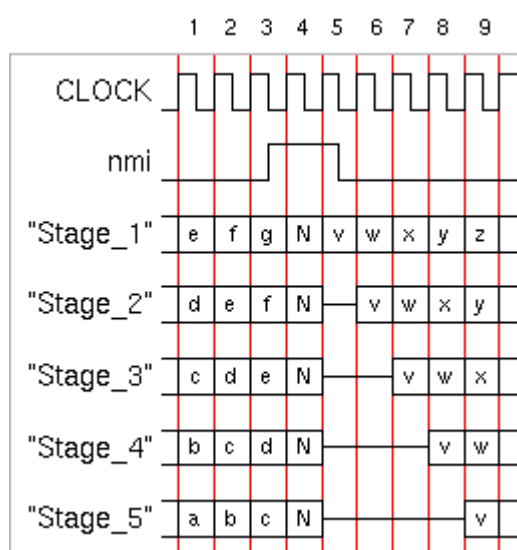


図48 NMI 割込みのタイミングチャート

6.5 割込み仕様入力手順

このページでは、次の割込みを作成することを想定して割り込みの入力手順について説明します。

- リセット割込み
- NMI
- 外部割込み
- デコードエラー割込み(内部割込み)
- ALU のオーバーフロー割込み(内部割込み)

リセット割込みと NMI 以外に対しては、割込みのマスクを行うことにします。

割込みの仕様を記述するためには次の手順で情報を入力します。

1. 割込み用ポート，ユーザ定義ポートの設定 (Interface Definition フェーズ)
2. ユーザ定義ポート用の wire リソース宣言 (Resource Declaration フェーズ)
3. 割込みの宣言 (Instruction Definition フェーズ)
4. 割込み終了命令の宣言 (Instruction Definition フェーズ)
5. 割込みのマイクロ動作記述 (Micro Op. Description フェーズ)
6. 割込み発生のマイクロ動作記述 (Micro Op. Description フェーズ)
7. 割込み終了命令のマイクロ動作記述 (Micro Op. Description フェーズ)

ユーザ定義ポートとして、割込み入力信号以外のポートを宣言します。その場合 Resource Declaration フェーズで同名のワイヤリソース(wire_in, wire_out, 又は wire_inout)を宣言しておく必要があります。また、"割込み発生のマイクロ動作記述"は、命令内で発生する内部割込みに対してのみ必要なステップです。

6.6 割込みの入力

6.6.1 割込み用ポート，ユーザ定義ポートの設定

外部割込み用のポートを宣言します。

<i>Attribute</i>	<i>Port Name</i>	<i>Direction</i>	<i>Signal Type</i>
interrupt	EXTINT_IN	in	std_logic
interrupt	NMI_IN	in	std_logic
unspecified	CATCH_OUT	out	std_logic
unspecified	EOI_OUT	out	std_logic

“EXTINT_IN”は外部割込み要求入力信号，“NMI_IN”は NMI 要求入力信号，“CATCH_OUT”は割込み受理通知信号，“EOI_OUT”は割込み終了通知信号です。

6.6.2 ユーザ定義ポート用の wire リソース宣言

ユーザ定義ポート(Attribute=unspecified のポート)には対応しては、wire リソースを作成する必要があります。

<i>Name</i>	<i>Path</i>	<i>width</i>	<i>default_output</i>
CATCH_OUT	/workdb/FHM_work/wire_out	1	fixed_to_0
EOI_OUT	/workdb/FHM_work/wire_out	1	fixed_to_0

ここで、wire リソースの名前とユーザ定義ポートの名前は同じにしておく必要があります。同じであれば、ユーザ定義ポートと wire リソースが自動的に接続されます。

また、マスクレジスタのリソースを作成します。

<i>Name</i>	<i>Path</i>	<i>width</i>	<i>Use as</i>
MASKREG	/basicfhmdb/storage/register	1	Mask Register

ここで、Use as は Mask Register にしておく必要があります。Mask Register で指定されたリソースが、割込み宣言時にマスクレジスタとして選択できます。

6.6.3 割込みの宣言

上記の 5 つの割込みを宣言します。

リセット割込み

<i>Interrupt Name</i>	<i>Properties</i>		
reset	<i>Type</i>	Reset	
	<i>Condition</i>	<i>Type</i>	Unselected
		<i>Port Name</i>	<i>Active Value</i>
		RESET	1
	<i>Mask</i>	<i>Maskable</i>	No
		<i>Register Name</i>	Unselected
		<i>Position</i>	
		<i>Register Value</i>	

NMI

<i>Interrupt Name</i>	<i>Properties</i>		
nmi	<i>Type</i>	NMI	
	<i>Condition</i>	<i>Type</i>	Unselected
		<i>Port Name</i>	<i>Active Value</i>
		NMI_IN	1
	<i>Mask</i>	<i>Maskable</i>	No
		<i>Register Name</i>	Unselected
		<i>Position</i>	
		<i>Register Value</i>	

外部割込み

<i>Interrupt Name</i>	<i>Properties</i>		
extint	<i>Type</i>	External	
	<i>Condition</i>	<i>Type</i>	Unselected
		<i>Port Name</i>	<i>Active Value</i>
		EXTINT_IN	1
	<i>Mask</i>	<i>Maskable</i>	Yes
		<i>Register Name</i>	MASKREG
		<i>Position</i>	0
		<i>Register Value</i>	1

デコードエラー割込み(内部割込み)

Interrupt Name	Properties		
dec_err	Type	Internal	
	Condition	Type	decode_error
		Port Name	Active Value
	Mask	Maskable	Yes
		Register Name	MASKREG
		Position	0
		Register Value	1

ALU のオーバーフロー割込み(内部割込み)

Interrupt Name	Properties		
overflow	Type	Internal	
	Condition	Type	instr_specific
		Port Name	Active Value
	Mask	Maskable	Yes
		Register Name	MASKREG
		Position	0
		Register Value	1

マスク条件の設定

Mask	Maskable	Yes
	Register Name	MASKREG
	Position	0
	Register Value	1

は、「レジスタ MASKREG の 0 ビット目(LSB)の値が 1 である場合は、割込みの発生を抑制する」ことを表します。

以下にまとめます。

Interrupt Name	Properties							
	Type	Condition			Mask			
	Type	Type	Port Name	Active Value	Maskable	Register Name	Position	Register Value
reset	Reset		RESET	1				
nmi	NMI		NMI_IN	1				
extint	External		EXTINT_IN	1	Yes	MASKREG	0	1
overflow	Internal	instr_specific			Yes	MASKREG	0	1

dec_err	Internal	decode_error		Yes	MASKREG	0	1
---------	----------	--------------	--	-----	---------	---	---

6.6.4 割込み終了命令の宣言

割込み処理ルーチンが終了したときに EOI_OUT(end of interrupt)信号を出す命令を作っておきます。割込みハンドラの最後でこの命令を発行すれば、EOI_OUT ポートから、外部の割込みコントローラに割込み終了を通知することができます。表 20 に割込み終了命令のタイプ “type0” の定義を、表 21 に割込み終了命令 “eoi” の定義の例を示します。

表 20 割込み終了命令タイプの例

Name	MSB	LSB	Field Type	Field Attr	Name/Value
type0	31	26	OP-code	binary	011111
	25	0	OP-code	name	op

表 21 割込み終了命令の例

Name	Type	Field	Format
eoi	type0	011111 111111111111111111111111	eoi

6.6.5 割込みのマイクロ動作記述

割込みのマイクロ動作記述は、“Micro Op. Description”ウインドウの“Exception”タブ内に記述します。この記述は 1 サイクルで終わらなければいけません。すなわち、パイプライン処理、マルチサイクル処理は記述できません。一般的には、catch 信号をアサートし、割込みマスクの設定をするか、割込み発生時の PC の値を保存し、割込みハンドラへジャンプする処理を書きます。

BNF

<割込み動作文> ::= <単純代入文> | <条件付き代入文> | <保存 PC 代入文>

<保存 PC 代入文> ::= <左辺> ‘=’ ‘saved_pc’ ‘;’

(cf.) <文> ::= <単純代入文> | <条件付き代入文> | <条件付き機能実行文> | <内部割込み発生文> | <条件付き内部割込み発生文>

ユーザ定義ポートの扱い

ユーザ定義ポートの読み書きには、対応する wire リソースの read 又は write ファンクションを使用します。

6.7 割込みハンドラへのジャンプ、割込み受理通知信号のアサーション

6.7.1 固定アドレスへジャンプする割込みの記述例

割込みハンドラのアドレスが固定の場合は次のように書きます。これは、割込みが入ったときに、0x10000000 番地にジャンプするという記述です。

```
wire [31:0] jump_addr;
wire       one;

jump_addr = “00010000000000000000000000000000” ;
null      = PC.write(jump_addr);
```

```

one      = '1' ;
null     = MASKREG.write(one);
null     = CATCH_OUT.write(one);

```

6.7.2 外部からのアドレス入力に応じてジャンプする割込みの記述例

割込みコントローラとプロセッサの間に，外部割込みの種類を通知する専用ポート(ADR_IN)を設けて，その値によって割込みハンドラのアドレスを可変にする場合は次のように記述します．これは，割込みが入ったときに，ADR_IN ポートから読み出したアドレスに飛ぶという記述です．

```

wire [31:0] jump_addr;
wire [3:0]  intr_vec;
wire [3:0]  h0;
wire [7:0]  h00;
wire        one;

h0          = "0000" ;
h00         = "00000000" ;
intr_vec    = ADR_IN.read();
jump_addr   = <h00, h00, h00, h00, h00, h00, h00, intr_vec, h0>;
null        = PC.write(jump_addr);
one         = '1' ;
null        = MASKREG.write(one);
null        = CATCH_OUT.write(one);

```

6.7.3 外部からの割込みベクタ入力に応じてジャンプする割込みの記述

割込みコントローラとプロセッサの間に専用ポートを設けて，その値に応じて割込みハンドラのアドレスを表引きする場合は次のように記述します．これは，割込みが入ったときに，INTR_VECTOR_IN ポートから読み出した割込みベクタによって，変数 intr_vector0-3 で指定されているアドレスにジャンプするという記述です．

```

wire [31:0] jump_addr;
wire [31:0] intr_vector0;
wire [31:0] intr_vector1;
wire [31:0] intr_vector2;
wire [31:0] intr_vector3;
wire [31:0] tmp_iv_0;
wire [31:0] tmp_iv_1;
wire [1:0]  vector;
wire        vector0;
wire        vector1;
wire        one;

// Interrupt Vector Table
intr_vector0 = "00001000000000000000000000000000" ;

```

```

intr_vector1 = "00001100000000000000000000000000" ;
intr_vector2 = "00001110000000000000000000000000" ;
intr_vector3 = "00001111000000000000000000000000" ;
// read Interrupt Vector from Input Port for Interrupt Vector
vector      = INTR_VECTOR_IN.read() ;
// determine 'jump_addr' according to 'vector'
vector0     = vector[0] ;
vector1     = vector[1] ;
tmp_iv_0    = (vector0)? Intr_vector1 : intr_vector0;
tmp_iv_1    = (vector0)? Intr_vector3 : intr_vector2;
jump_addr   = (vector1)? Tmp_iv_1 : tmp_iv_0;

// jump to 'jump_addr'
null        = PC.write(jump_addr);

// assert mask register
one         = '1' ;
null        = MASKREG.write(one);
null        = CATCH_OUT.write(one);

```

6.7.4 リセット入力の記述例

リセット割込みではレジスタや命令レジスタをリセットします。PC はリセットするか、起動時に読みこむプログラムがあるアドレスをセットします。ここでは、GPR と IR と MASKREG をリセットし、PC にアドレス 0x30000 をセットしています。

```

wire [31:0] start_addr;
wire [7:0] h00; wire [7:0] h03;

h00      = "00000000" ;
h03      = "00000011" ;
start_addr = <h00, h00, h00, h00, h00, h03, h00, h00>;
null      = PC.write(start_addr);
null      = GPR.reset();
null      = IR.reset();
null      = MASKREG.reset();

```

6.8 割込み処理における割込み発生時における PC の値の取得方法

一般的に、割込みハンドラ終了後に元のプログラムに復帰するために、割込み信号が入ったときの PC(プログラムカウンタ)の値を取得して保存しておく必要があります。取得できる PC の値は、内部割込みの場合には、その内部割込みが発生した命令のアドレス、外部割込みの場合には、最後に実行が完了した命令の次に実行すべきアドレスを表します。すなわち、元のプログラムへの復帰時には、取得したアドレスから実行を再開すれば重複実行なしに処理を再開することができます。なお、リセット割込み、NMI については、割込み発生時の PC の値は正しく取得できません。

次に示すマイクロ動作記述で、割込み発生時の PC の値を取得できます。ここでは、割込みが入

ったときの PC の値を, 割込み処理の中で直接レジスタファイルに書き込んでいます (ここでは第 20 番レジスタに書き込んでいます).

```
wire [31:0] spc;
wire [4:0] sel;

sel = "10100" ; // 20th
spc = saved_pc;
null = GPR.write0(sel, spc);
```

“saved_pc”は, 予約語扱いです. 割込み動作記述の時のみ使用が許されていますので, 一般命令のマイクロ動作記述内で用いることはできません. また, saved_pc は, 代入文の右辺に単独で用いる形式のみサポートされていますので, FHM リソースの機能実行文の引数などに用いるときには, 別の wire 変数(上記の例では変数“spc”)に一旦代入する必要があります.

6.8.1 割込み発生 of the マイクロ動作記述

内部割込みで, “instr_specific”タイプの内部割込みは一般命令のマイクロ動作記述内で割込みを throw します. (“decode_error”タイプの内部割込みは, 検出論理が自動挿入されます.) ALU で加算を行い, オーバーフローフラグが立っていれば“overflow”内部割込みを発生させる例を示します.

```
wire [3:0] flag;
wire cond;

<result, flag> = ALU.add(source0, source1);
cond           = flag[0];
[cond] throw overflow;
```

6.8.2 割込み終了命令 of the マイクロ動作記述

割込み終了時に発行する命令について記述します.

```
wire eoi_signal;
wire zero;

eoi_signal = '1' ;
null       = EOI_OUT.write(eoi_signal);
zero       = '0' ;
null       = MASKREG.write(zero);
```

7. PAS - メタアセンブラユーザズマニュアル

本書は, PAS メタアセンブラのユーザズマニュアルです. PAS バージョン 0.3 に対応しています.

7.1 PAS の概要

メタアセンブラ PAS は、構成変更可能なアセンブラです。変更可能なパラメータは、レジスタクラス、命令形式などの命令フォーマット、1 バイトのビット数、エンディアン、アドレッシング単位(バイト、ワード) などからなります。これらのパラメータは、アセンブル規則記述ファイルと呼ばれる、命令フォーマットを記述するファイルや、アセンブラの制御命令(疑似命令)、コマンド行オプションなどで指定することができます。

アセンブル規則記述ファイルについての詳細は「7.4 アセンブル規則記述ファイル」を参照してください。アセンブラ制御命令については「7.7 アセンブラ制御命令」を参照してください。コマンド行オプションについては「7.2 PAS の起動方法」を参照してください。

7.2 PAS の起動方法

pas の起動方法について説明します。起動形式は以下の通りです。

```
pas [-des descfile] [-src srcfile] ¥  
    [-a listingfile] [-endian endian] ¥  
    [-help] [-version] [-debug]
```

各オプションの意味は以下の通りです。

-des *descfile*

引数 *descfile* でアセンブル規則記述ファイルを指定します。このオプションは、**-help** または **-version** が指定されていない場合は必須です。

-src *srcfile*

引数 *srcfile* でアセンブリソースファイルを指定します。このオプションは、**-help** または **-version** が指定されていない場合は必須です。

-a *listingfile*

引数 *listingfile* でアセンブルリスティングの出力ファイルを指定します。このオプションを省略した場合は、標準出力に出力されます。

-endian *endian*

引数 *endian* でエンディアンを指定します。指定できるのは、**Big** または **Little** です。

-help

pas の使用方法の簡単な要約を標準出力に表示して終了します。

-version

pas のバージョンを標準出力に表示して終了します。

-debug

通常のリスティング表示に加えて、アセンブル規則ファイルを解析した結果のダンプやセクションの内容を表示します。

7.3 PAS の実行例

```
Pas -des foo.des -src a.s
```

ここで、foo.des がアセンブル規則記述ファイル、a.s がアセンブリソースファイルです。a.s が

```
.data.24 -1  
.data.32 128  
.data.16 5
```

L4:

L5:

```
.data.24 0123
.data.24 0xabcd
```

となっていた場合、次のようなリスティング出力が得られます。

***** Source Program List *****

LineNo	LC	Code	Source Program
1	0000	ffffff	.data.24 -1
2	0003	00000080	.data.32 128
3	0007	0005	.data.16 5
4			L4:
5			L5:
6	0009	000053	.data.24 0123
7	000c	00abcd	.data.24 0xabcd

***** Cross Reference List *****

Defined Symbol

name	section	lc	attr	value	lineno
L4	.text	0009	Label	9	4
L5	.text	0009	Label	9	5

Undefined Symbol

name	section	lc	attr	value	lineno
------	---------	----	------	-------	--------

Multiple Defined Symbol

name	section	lc	attr	value	lineno
------	---------	----	------	-------	--------

***** Section Data List *****

Sec	Attr	Size
.text	Writable	f
addr_space : 16		
addressing : Byte		
bitwidth per byte : 8		
word alignment : 4		
.data	Data	0
addr_space : 16		
addressing : Byte		
bitwidth per byte : 8		
word alignment : 4		

7.4 アセンブル規則記述ファイル

アセンブル規則記述ファイルには以下の三つを記述します。

- レジスタクラス定義

レジスタをグループ分けし、それをクラスを定義します。クラス毎に、そのクラスに属するレジスタの名前と、そのレジスタの機械語コード中の表現の対応を記述します。

```
Resource_tbl {
    "gpr" {
        {"GPR0", "00000"},
        {"GPR1", "00001"},
        ...
        {"GPR31", "11111"}
    }
}
```

これは、あるプロセッサのアセンブル規則記述ファイルから抜き出したものです。この例では、*gpr* というレジスタクラスを定義しており、このクラスには、GPR0, GPR1, ..., GPR31 というレジスタが属していることを表しています。例えば、GPR1 というレジスタの機械語コード中での表現は、00001 となります。

- 命令形式定義

機械語命令の形式を定義します。命令長、オペランドの数、オペランド形式を定義します。命令形式定義の例を以下に示します。

```
Instruction_type {
    type {
        "ltype" {
            width {"31", "0"},
            otype {
                "ltype0" {
                    "RDirect" {"Resource" {"rt", "gpr"}},
                    "RDirect" {"Resource" {"rs", "gpr"}},
                    "Immediate_data" {"Immediate" {"immediate", ""}}
                },
                "ltype02" {
                    "RDirect" {"Resource" {"rt", "gpr"}},
                    "RDirect" {"Resource" {"rs", "gpr"}},
                    "Absolute_address" {"Symbol" {"immediate", ""}}
                },
                ...
            }
        }
    }
}
```

}

この部分では、**Itype** という名前の命令形式を定義しています。 **Itype** という命令形式の命令長は、**width{"31","0"}** により、32 ビットになります。 **otype** に、**Itype** 形式で使用できるオペランドの組を定義します。 この場合、**ItypeO** というオペランド形式は、三つのオペランドからなり、汎用レジスタ、汎用レジスタ、即値という組合せになります。 **PAS** では、アドレッシングモードについては、一般に良く使用されている形式をあらかじめ定義してあります。 **RDirect** はレジスタ直接、**Immediate_data** は、即値を表します。 **gpr** は、レジスタクラス定義で定義されているレジスタクラス名です。

- 命令定義

個々の機械語命令を定義します。 どの命令形式に属するか、フィールドの幅、位置、用途(オペコードかオペランドか)などを記述します。 **ADDI** 命令の定義を以下に示します。

```
Instruction{
    "ADDI"{
        type{"Itype"{otype{"ItypeO"}}},
        "OP-code"{"binary"{"001000"},width{"31","26"}},
        "Operand"{"name"{"rs"},width{"25","21"}},
        "Operand"{"name"{"rt"},width{"20","16"}},
        "Operand"{"name"{"immediate"},width{"15","0"}}
    },
    ...
}
```

これを見ると、**ADDI** の命令形式は **Itype** であり、オペランド形式は **ItypeO** であることが分かります。 **OP-code** や **Operand** は命令の各フィールドを記述します。 命令の 31 ビット～26 ビットがオペコードを表し、**ADDI** の場合はその値が二進法で 00100 になります。 オペランドは三つ取り、一番目が 25 ビット～21 ビットで表されます。 **rs** という名前は、命令形式定義に現れており、それを見ると、レジスタ直接で、使用するレジスタは **gpr** クラスに所属するということになります。 二番目のオペランドは 20 ビット～16 ビットで、やはり **gpr** クラスのレジスタを表します。 三番目は、15 ビット～0 ビットで、16 ビットの即値を表すということが分かります。

アセンブル規則記述ファイルには、それぞれの機械語命令が具体的にどのような動作をするかまでは記述しません。 あくまでもアセンブルを行うのに必要な情報に限られます。

以下、各定義の詳細を見ていきます。

7.4.1 レジスタクラス定義

レジスタクラス定義の記述形式を以下に示します。

```
Resource_tbl {
    "レジスタクラス名" {
        {"レジスタ名","コード"},
        {"レジスタ名","コード"},
        ...
        {"レジスタ名","コード"}
    },
    "レジスタクラス名" {
```

```

        {"レジスタ名", "コード"},
        {"レジスタ名", "コード"},
        ...
        {"レジスタ名", "コード"}
    },
    ...
}

```

`Resource_tbl` というキーワードでレジスタクラス定義の開始を表します。以下、中括弧の中に、各レジスタクラスの定義をカンマで区切って並べます。各レジスタクラスは、レジスタ名とそのコード(二進数)を並べたものになります。レジスタクラス名、レジスタ名、コードは ""(ダブルクォート)で囲む必要があります。

7.4.2 命令形式定義

命令形式定義の記述形式を以下に示します。

```

Instruction_type {
    type {
        "命令形式名" {
            width{"最上位ビット", "最下位ビット"},
            オペランド指定
        },
        "命令形式名" {
            width{"最上位ビット", "最下位ビット"},
            オペランド指定
        },
        ...
    }
}

```

`Instruction_type` キーワードで命令形式定義を開始します。以下、必要な分だけ各命令形式の記述を並べます。各命令形式は、命令形式名、命令長指定、オペランド指定の組からなります。

`width` は各命令形式の命令長を指定する構文で、例えば 32 ビット長の場合、`width{"31","0"}` のように、最上位ビットのビット位置、最下位ビットのビット位置の組合せで指定を行います。オペランド指定は、オペランドの数と形式を指定するものです。以下にオペランド指定の記述形式を示します。

```

otype{
    "オペランド形式名" {
        "アドレッシングモード名" {
            "構成要素名", {"フィールド名", "レジスタクラス名"}
        },
        "アドレッシングモード名" {
            "構成要素名", {"フィールド名", "レジスタクラス名"},
        }
        ...
    },
    "オペランド形式名" {

```

```

"アドレッシングモード名" {
    "構成要素名", {"フィールド名", "レジスタクラス名"}
},
"アドレッシングモード名" {
    "構成要素名", {"フィールド名", "レジスタクラス名"},
}
...
},
...
}

```

otype キーワードでオペランド指定を開始します。以下に、所属する命令形式で使用可能なオペランドの組合せ指定を並べて書きます。オペランド形式名は、オペランドの組合せの名前です。アドレッシングモード名は、オペランドのアドレッシングモードを表し、PAS であらかじめ定義されています。構成要素名は、このオペランドの意味を表します。フィールド名は、次のセクションで説明する命令定義に現れるもので、機械語コードのどのフィールドにオペランドが現れるかを指定するために使います。オペランドがレジスタを使用するものである場合、レジスタクラス名を指定します。レジスタを使用しない場合は、空("")にします。

アドレッシングモード名として使用可能な名前は以下の通りです。

```

RDirect
Indirect
RIndirect
RlwPreDec
RlwPreInc
RlwPostDec
RlwPostInc
RlwDisp
RlwDisp_Up
RlwIndex
RlwIndex_Up
RlwIndex_Up
RlwScaledIndex
RlwDispScaledIndex
PCrelative_address
Absolute_address
Immediate_data

```

アドレッシングモードの詳細については、「7.6 アドレッシングモード」を参照してください。

また、構成要素名は以下のいずれかになります。

Resource

レジスタであることを表します。

Displacement

ディスプレイースメント修飾レジスタ間接などのアドレッシングモードのディスプレイースメントであることを表します。

Immediate

即値であることを表します。

Scale

スケール付インデックス修飾レジスタ間接などのアドレッシングモードのスケールであることを表します。

Symbol

シンボルであることを表します。

7.4.3 命令定義

```
Instruction {
  "命令名" {
    type{"命令形式名"{otype{"オペランドタイプ名"}}},
    "フィールドタイプ名"{
      "フィールド属性"{"フィールド値"}, width{"MSB", "LSB"}
    },
    "フィールドタイプ名"{
      "フィールド属性"{"フィールド値"}, width{"MSB", "LSB"}
    },
    ...
  },
  "命令名" {
    type{"命令形式名"{otype{"オペランドタイプ名"}}},
    "フィールドタイプ名"{
      "フィールド属性"{"フィールド値"}, width{"MSB", "LSB"}
    },
    "フィールドタイプ名"{
      "フィールド属性"{"フィールド値"}, width{"MSB", "LSB"}
    },
    ...
  },
  ...
}
```

Instruction キーワードで命令定義を開始します。各命令は、命令名、命令形式指定、各フィールドの記述からなります。

命令形式指定は、命令形式定義で定義した命令形式名とオペランドタイプ名を使って指定を行います。フィールドの指定は、フィールドタイプ名、フィールド属性、フィールド値、位置指定からなります。フィールドタイプ名は、オペコードやオペランドの区別を表します。指定できるのは以下の通りです。

OP-code

フィールドがオペコードであることを表します。

Operand

フィールドがオペランドであることを表します。

Reserved

フィールドが予約されており、現在使用されていないことを表します。

フィールド属性は、**binary** か **name** になります。**binary** の場合は、次のフィールド値にフィールドに入るべき値を二進数で指定します。**name** の場合は、フィールド値には命令形式定義で定義したフィールド名を書き、フィールドとオペランドの記述を結びつけます。

位置指定は、**width** キーワードの後の中括弧に、フィールドの最上位ビット位置(MSB)と最下位ビット位置(LSB)を指定することで行います。

7.5 アセンブリ言語の文法

アセンブリソースコードの書き方を以下に示します。

7.5.1 文

ソースプログラムは文の並びから構成されます。文は一行に一個記述します。文の構成は以下のようになります。

[ラベル] <オペレーション [オペランド]> [コメント]

‘ラベル’

文につける名前です。

ラベルは行頭から開始し、ラベル名の直後に : (コロン)をつけます。ラベルに使用できる文字は、先頭の一文字が 英大文字、英小文字、アンダースコア(_), ピリオド(.) のどれかでなければなりません。二文字目以降は、これらの文字に数字、ドル記号(\$)が加わります。

‘オペレーション’

実行命令、アセンブラ制御命令のニーモニックを記述します。実行命令は CPU の命令のことで、アセンブル規則記述ファイルに記述されているものです。アセンブラ制御命令は、アセンブラに指示を与える命令です。

オペレーションは、ラベルがない場合は、2 カラム目以降から書き始めます。ラベルがある場合は、ラベルの後に 1 つ以上の空白またはタブをおいて書き始めます。

オペレーションによっては、ラベルを記述できない場合があります。アセンブラ制御命令でラベルを記述できないものについては、各アセンブラ制御命令の解説のところで、その旨を明記してあります。

‘オペランド’

オペレーションの実行対象などを記述します。オペランドの個数と種類は、オペレーションによって異なります。

オペランドは、オペレーションの後に 1 つ以上の空白またはタブをおいて書き始めます。

‘コメント’

注釈を記述します。プログラムの実行には影響ありません。

コメントは、セミコロン(;)から行末までになります。

7.5.2 予約語

予約語は、特別な意味を持つ語としてアセンブラが用意している名前です。

予約語の種類には、アセンブラ制御命令、演算子、ロケーションカウンタがあります。

7.5.3 シンボル

シンボルは、プログラマが定義する名前であり、次の役割を果たします。

‘アドレスシンボル’

データの格納場所、分岐先などのアドレスを表します。

‘定数シンボル’

定数を表します。

シンボル名に使用できる文字は、先頭の一文字が 英大文字、英小文字、アンダースコア(_), ピリオド(.) のどれかでなければなりません。二文字目以降は、これらの文字に数字、ドル記号(\$)が加わります。

予約語は、シンボルとして使用できません。

7.5.4 定数

定数には整数定数と文字列定数があります。

整数定数には以下の種類があります。

`2 進整数`

「0b」または「0B」で始まり、「0」または「1」が続きます。

`8 進整数`

「0」で始まり、「0-7」が続きます。

`10 進整数`

「0」でない整数で始まり、「0-9」が続きます。

`16 進整数`

「0x」または「0X」で始まり、「0-9,a-f,A-F」が続きます。

文字列定数は、文字の並びを対応する ASCII コード値のデータの並びとして、扱うものです。ダブルクォートで囲って表します。通常の ASCII 印字可能文字の他に、特別な表記法として以下のものが使用できます。

¥b

バックスペース BS(0x08) を表します。

¥f

フォームフィード FF(0x0C) を表します。

¥n

ラインフィード LF(0x0A) を表します。

¥r

キャリッジリターン CR(0x0D) を表します。

¥t

水平タブ HT(0x09) を表します。

¥v

垂直タブ VT(0x0B) を表します。

¥¥

バックスラッシュを表します。

¥"

ダブルクォートを表します。

例:

"hello¥t¥"world¥"¥n"

また、ASCII コード値を 8 進表記、16 進表記を使って表すこともできます。

`8 進表記`

¥0 の後に、0~7 の数字を並べます。

例:

"¥012¥033"

`16 進表記`

¥0x または ¥0X の後に、0~9,a~f,A~F の数字を並べます。

"¥0xa¥0xb"

7.5.5 ロケーション・カウンタ

ロケーション・カウンタは、オブジェクトコードを配置するアドレスを指します。ロケーション・カ

カウンタの値は、オブジェクトコードの出力に応じて自動的に変化します。また、アセンブラ制御命令によって指定の値に変更することも可能です。

現在のロケーション・カウンタの値は、ドット(.)で参照できます。・は予約語です。

7.5.6 式

式は、定数やシンボルと演算子を組み合わせて演算結果を求めるものです。実行命令やアセンブラ制御命令のオペランドに使用します。

式の構成要素を以下に示します。

7.5.6.1 項

項は以下のものから成ります。

- 定数
 - ロケーション・カウンタ(.)
 - シンボル
 - 上記の項と演算子による演算結果
- 単独の項も式の一種です。

7.5.6.2 演算子

演算子の種類と優先度は以下ようになります。優先度の数字が小さいほど優先度が高くなります。

優先度 0

—

単項の - は、符号反転または 2 の補数

~

ビット毎の NOT, 1 の補数

優先度 1

*

乗算

/

除算

%

剰余

優先度 2

|

ビット毎の OR

&

ビット毎の AND

^

ビット毎の排他的論理和

優先度 3

+

加算

-

減算

7.5.6.3 括弧

丸括弧 () によって、演算の優先順位を変えることができます。

1 つの式の中に複数の演算が含まれる場合、演算子の優先順位と括弧指定によって、演算を処理する

順序が決まります。

PAS は以下の規則に従って演算を処理します。

`規則 1`

括弧で括られた演算から処理する。括弧が多重になっているときは、より内側の括弧で括られた演算を優先する。

`規則 2`

演算の優先順位が高いものから処理する。

7.6 アドレッシングモード

PAS では、一般的に使用されるアドレッシングモードをあらかじめ用意しており、アーキテクチャによらず統一的に使用することができます。

以下に PAS で定義されているアドレッシングモードを示します。

`%Rn : レジスタ直接`

レジスタを参照します。

`(式) : 間接アドレッシング`

式の値がメモリアドレスを表します。

`(%Rn) : レジスタ間接`

レジスタ Rn の値がメモリアドレスを表します。

`- (%Rn) : プリ・デクリメント・レジスタ間接`

デクリメントした後のレジスタ Rn の値がメモリアドレスを表します。

`+ (%Rn) : プリ・インクリメント・レジスタ間接`

インクリメントした後のレジスタ Rn の値がメモリアドレスを表します。

` (%Rn) - : ポスト・デクリメント・レジスタ間接`

デクリメントする前のレジスタ Rn の値がメモリアドレスを表します。

` (%Rn) + : ポスト・インクリメント・レジスタ間接`

インクリメントする前のレジスタ Rn の値がメモリアドレスを表します。

`disp (%Rn) : ディスプレースメント修飾レジスタ間接`

「Rn の値 + disp」の値がメモリアドレスを表します。レジスタ Rn の値は更新されません。

` (%Rbase + disp) : ディスプレースメント修飾レジスタ間接更新アドレッシング`

「Rn の値 + disp」の値がメモリアドレスを表します。レジスタ Rn の値は、「Rn の値 + disp」の値に更新されます。

` (%Rbase, %Rindex) : インデックス修飾レジスタ間接`

「Rbase の値 + Rindex の値」の値がメモリアドレスを表します。レジスタ Rbase, Rindex の値は更新されません。

` (%Rbase + %Rindex) : インデックス修飾レジスタ間接更新アドレッシング`

「Rbase の値 + Rindex の値」の値がメモリアドレスを表します。レジスタ Rbase の値は「Rbase の値 + Rindex の値」に更新されます。

` (%Rbase, %Rindex, scale) : スケール付インデックス修飾レジスタ間接`

「Rbase の値 + Rindex の値 × scale」の値がメモリアドレスを表します。レジスタ Rbase, Rindex の値は更新されません。

`disp (%Rbase, %Rindex, scale) : ディスプレースメント、スケール付インデックス修飾レジスタ間接`

「Rbase の値 + Rindex の値 × scale + disp」の値がメモリアドレスを表します。レジスタ Rbase, Rindex の値は更新されません。

`symbol : シンボル指定による PC 相対アドレス`

symbol が、分岐先メモリアドレスを表します。「symbol の値 - PC」の値を PC 相対のディスプレースメントとします。

`*symbol : シンボル指定による絶対アドレス'

symbol が、分岐先の絶対メモリアドレスを表します。

`\$imm: 即値'

imm の値がそのまま使われます。

7.7 アセンブラ制御命令

7.7.1 .accum "accum_num"

整数部つき固定小数点データを確保します。accum_num で指定される 10 進小数点表記の固定小数点数を .accum_spec 制御命令で指定された形式に従って変換し、メモリ上に配置します。

指定できるデータの値は、.accum_spec で指定した整数部の二進桁数により異なります。整数部二進桁数が 4 であれば、 $-16 \leq accum_num < 16$ の範囲になります。

例:

```
.accum -16.0
.accum 15.5
```

7.7.2 .accum.s "accum_num"

短精度整数部つき固定小数点データを確保します。accum_num で指定される 10 進小数点表記の固定小数点数を .short_accum_spec 制御命令で指定された形式に従って変換し、メモリ上に配置します。

指定できるデータの値は、.short_accum_spec で指定した整数部の二進桁数により異なります。整数部二進桁数が 4 であれば、 $-16 \leq accum_num < 16$ の範囲になります。

例:

```
.accum.s -16.0
.accum.s 15.5
```

7.7.3 .accum.l "accum_num"

長精度整数部つき固定小数点データを確保します。accum_num で指定される 10 進小数点表記の固定小数点数を .long_accum_spec 制御命令で指定された形式に従って変換し、メモリ上に配置します。

指定できるデータの値は、.long_accum_spec で指定した整数部の二進桁数により異なります。整数部二進桁数が 4 であれば、 $-16 \leq accum_num < 16$ の範囲になります。

例:

```
.accum.l -16.0
.accum.l 15.5
```

7.7.4 .accum_spec "spec"

整数部つき固定小数点データ形式を定義します。符号付の形式と符号なしの形式がありますが、その両方を指定します。spec でデータの大きさ、整数部二進桁数、スケール(小数部二進桁数)を指定します。spec の前半部で符号なしを、後半部で符号付を指定します。

spec の指定方法は以下のようになります。

size1: integral1: scale1, size2: integral2: scale2

`size1`

符号なし型の場合のデータの大きさ

`integral1`

符号なし型の場合の整数部の二進数での桁数

`scale1`

符号なし型の場合の小数部の二進数での桁数

`size2`

符号付き型の場合のデータの大きさ

`integral2`

符号付き型の場合の整数部の二進数での桁数

`scale2`

符号付き型の場合の小数部の二進数での桁数

デフォルトは

20:4:15, 20:4:15

になっています。

ラベルは記述できません。

7.7.5 **.addr_space "num"**

アドレス空間の大きさを指定します。引数 *num* は整数値を指定します。アドレス空間の大きさは、 2^{num} になります。デフォルト値は、16 です。

ラベルは記述できません。

7.7.6 **.addressing "addrunit"**

メモリのアドレッシング単位を設定します。*addrunit* に指定可能なのは、Byte または Word です。Byte を指定するとアドレッシングがバイト単位になります。Word を指定するとアドレッシングがワード単位になります。デフォルト値は、Byte です。

ラベルは記述できません。

例:

```
.addressing Byte
```

```
.addressing Word
```

7.7.7 **.align "num"**

引数 *num* は整数値であり、ロケーション・カウンタ値の境界調整数を指定します。ロケーション・カウンタの値を境界調整数の倍数に補正します。

例:

```
.align 4
.data.32 0x1234
```

この例では、*.align 4* で、*.data.32* 制御命令で確保する定数を 4 バイト境界に調整します。

7.7.8 **.ascii "string"**

引数 *string* で指定される文字列定数データをメモリ上に確保します。ゼロ終端文字列は追加しません。ラベルを記述可能です。

例:

```
.ascii "hello, world¥n¥0"
str1: .ascii "ABCDEF"
```

7.7.9 .asciz "*string*"

引数 *string* で指定される文字列定数データをメモリ上に確保します。文字列データの末尾にゼロ終端文字列を追加します。ラベルを記述することが可能です。

例:

```
.asciz "hello, world¥n"
L1: .asciz "abcdef"
```

7.7.10 .bits_per_byte "*num*"

1 バイトのビット数を指定します。引数 *num* は整数値を指定します。デフォルトは 8 です。ラベルは記述できません。

例:

```
.bits_per_byte 12 ; 1 バイトを 12 ビットに指定
```

7.7.11 .data."*width*" "*num*"

整数データを確保します。*width* で、データの大きさをビット数で指定します。*num* は整数値であり、整数データの値を指定します。

例:

```
.data.24 10 ; 24 ビット長のデータを確保
.data.16 0x1234 ; 16 ビット長のデータを確保
```

7.7.12 .double "*hex_num*"

倍精度浮動小数点型データを確保します。データ型は `.double_spec` 制御命令で規定されます。引数 *hex_num* にはエンコード済みのデータを 16 進数で指定します。

例:

```
.double 0x3ff0000000000000 ; 1.0 (デフォルトの形式設定の場合)
```

7.7.13 .double_spec "*spec*"

倍精度浮動小数点データ型を定義します。引数 *spec* の指定は以下のようになります。

size: sign: exponent: mantissa

それぞれの意味は以下の通りです。大きさは全てビット数で指定します。

``size``

データ型全体の大きさです。

``sign``

符号を表すビット数を指定します。

``exponent``

指数部の大きさです。

``mantissa``

仮数部の大きさです。

デフォルトでは,

```
.double_spec 64:1:11:52
```

になっています.

ラベルは記述できません.

7.7.14 .end

ソースプログラムの終わりを宣言します.

ラベルは記述できません.

7.7.15 .endian "*endian*"

エンディアンを指定します. 引数 *endian* には, ビッグエンディアンの場合は **Big** を, リトルエンディアンの場合は **Little** を指定します. デフォルトは **Big** です.

ラベルは記述できません.

7.7.16 .equ "*symbol*", "*value*"

引数 *symbol* で指定されるシンボルに, 引数 *value* で指定される値を設定します. **.equ** で設定したシンボルの値を再設定することはできません. 二回目以降の設定は無視されます.

ラベルは記述できません.

例:

```
.equ foo, 0x100
```

7.7.17 .equiv "*symbol*", "*value*"

引数 *symbol* で指定されるシンボルに, 引数 *value* で指定される値を設定します. **.equiv** で設定したシンボルの値は, 再設定することが可能です.

ラベルは記述できません.

例:

```
.equiv bar, 0xff
```

7.7.18 .fixed "*fixed_num*"

固定小数点データを確保します. *fixed_num* で指定される 10 進小数点表記の固定小数点数を **.fixed_spec** 制御命令で指定された形式に従って変換し, メモリ上に配置します.

指定できるデータは, $-1 \leq \textit{fixed_num} < 1$ の範囲になります.

例:

```
.fixed 0.1  
.fixed -0.5
```

7.7.19 .fixed.s "*fixed_num*"

短精度固定小数点データを確保します. *fixed_num* で指定される 10 進小数点表記の固定小数点数を **.short_fixed_spec** 制御命令で指定された形式に従って変換し, メモリ上に配置します.

指定できるデータの値は, $-1 \leq \textit{fixed_num} < 1$ の範囲になります.

例:

```
.fixed.s 0.1  
.fixed.s -0.5
```

7.7.20 `.fixed.l "fixed_num"`

長精度固定小数点データを確保します。 *fixed_num* で指定される 10 進小数点表記の固定小数点数を `.long_fixed_spec` 制御命令で指定された形式に従って変換し、メモリ上に配置します。

指定できるデータは、 $-1 \leq \text{fixed_num} < 1$ の範囲になります。

例:

```
.fixed.l 0.1
.fixed.l -0.5
```

7.7.21 `.fixed_spec "spec"`

固定小数点データを定義します。符号付の形式と符号なしの形式がありますが、その両方を指定します。 *spec* でデータの大きさ、スケール(小数部二進桁数)を指定します。 *spec* の前半部で符号なしを、後半部で符号付を指定します。

spec の指定方法は以下のようになります。

size1: scale1, size2: scale2

``size1'`

符号なし型の場合のデータの大きさ

``scale1'`

符号なし型の場合の小数部の二進数での桁数

``size2'`

符号付き型の場合のデータの大きさ

``scale2'`

符号付き型の場合の小数部の二進数での桁数

デフォルトは、

```
.fixed_spec 16:15, 16:15
```

になっています。

ラベルは記述できません。

7.7.22 `.float "hex_num"`

単精度浮動小数点型データを確保します。データ型は `.float_spec` 制御命令で規定されます。引数 *hex_num* にはエンコード済みのデータを 16 進数で指定します。

例:

```
.float 0x3f800000 ; 1.0 (デフォルトの形式設定の場合)
```

7.7.23 `.float_spec "spec"`

単精度浮動小数点データ型を定義します。引数 *spec* の指定は以下のようになります。

size: sign: exponent: mantissa

それぞれの意味は以下の通りです。大きさは全てビット数で指定します。

``size'`

データ型全体の大きさです。

``sign'`

符号を表すビット数を指定します。

``exponent'`

指数部の大きさです。

``mantissa'`

仮数部の大きさです。

デフォルトでは、

`.float_spec 32:1:8:23`

になっています。

ラベルは記述できません。

7.7.24 `.long_accum_spec "spec"`

長精度整数部つき固定小数点データを定義します。符号付の形式と符号なしの形式がありますが、その両方を指定します。 *spec* でデータの大きさ、整数部二進桁数、スケール(小数部二進桁数)を指定します。 *spec* の前半部で符号なしを、後半部で符号付を指定します。

spec の指定方法は以下のようになります。

`size1: integral1: scale1, size2: integral2: scale2`

``size1'`

符号なし型の場合のデータの大きさ

``integral1'`

符号なし型の場合の整数部の二進数での桁数

``scale1'`

符号なし型の場合の小数部の二進数での桁数

``size2'`

符号付き型の場合のデータの大きさ

``integral2'`

符号付き型の場合の整数部の二進数での桁数

``scale2'`

符号付き型の場合の小数部の二進数での桁数

デフォルトは

`.long_accum_spec 36:4:31, 36:4:31`

になっています。

ラベルは記述できません。

7.7.25 `.long_fixed_spec`

長精度固定小数点データを定義します。符号付の形式と符号なしの形式がありますが、その両方を指定します。 *spec* でデータの大きさ、スケール(小数部二進桁数)を指定します。 *spec* の前半部で符号なしを、後半部で符号付を指定します。

spec の指定方法は以下のようになります。

`size1: scale1, size2: scale2`

``size1'`

符号なし型の場合のデータの大きさ

``scale1'`

符号なし型の場合の小数部の二進数での桁数

``size2'`

符号付き型の場合のデータの大きさ

``scale2'`

符号付き型の場合の小数部の二進数での桁数

デフォルトは、

`. long_fixed_spec 32:31, 32:31`

になっています。

ラベルは記述できません。

7.7.26 `.org "/c"`

ロケーション・カウンタ値を設定します。 *lc* はロケーション・カウンタ値を指定する整数です。 `.org` によりロケーション・カウンタ値が増えた場合、元のロケーション・カウンタ値と新しいロケーション・カウンタ値の間は 0 で埋められます。

ラベルは記述できません。

例:

`.org 0x100`

7.7.27 `.section "section"`

引数 *section* で指定されるセクションを宣言します。指定可能なセクションは以下の通りです。

`.text`

テキスト・セクション

`.data`

データ・セクション

すでに宣言済みのセクションを再び宣言し、再開することができます。

次のいずれかの場合、デフォルト・セクションが用意されます。

- セクションを宣言しないうちに、実行命令を記述している
- セクションを宣言しないうちに、データを確保するアセンブラ制御 命令を記述している。
- セクションを宣言しないうちに、`.align` または `.org` または `.space` アセンブラ制御命令を記述している
- セクションを宣言しないうちに、ロケーション・カウンタを参照している
- セクションを宣言しないうちに、ラベルの行だけを記述している

デフォルト・セクションは、先頭アドレス 0、境界調整数 2 のテキスト・セクションです。

ラベルは記述できません。

7.7.28 `.short_accum_spec "spec"`

短精度整数部つき固定小数点データを定義します。符号付の形式と符号なしの形式がありますが、その両方を指定します。 *spec* でデータの大きさ、整数部二進桁数、スケール(小数部二進桁数)を指定します。 *spec* の前半部で符号なしを、後半部で符号付を指定します。

spec の指定方法は以下のようになります。

size1: integral1: scale1, size2: integral2: scale2

``size1``

符号なし型の場合のデータの大きさ

``integral1``

符号なし型の場合の整数部の二進数での桁数

``scale1``

符号なし型の場合の小数部の二進数での桁数

``size2``

符号付き型の場合のデータの大きさ

``integral2``

符号付き型の場合の整数部の二進数での桁数

``scale2`

符号付き型の場合の小数部の二進数での桁数
デフォルトは,

`. short_accum_spec 12:4:7, 12:4:7`

になっています.

ラベルは記述できません.

7.7.29 `. short_fixed_spec "spec"`

短精度固定小数点データを定義します. 符号付の形式と符号なしの形式がありますが, その両方を指定します. `spec` でデータの大きさ, スケール(小数部二進桁数)を指定します. `spec` の前半部で符号なしを, 後半部で符号付を指定します.

`spec` の指定方法は以下ようになります.

`size1: scale1, size2: scale2`

``size1`

符号なし型の場合のデータの大きさ

``scale1`

符号なし型の場合の小数部の二進数での桁数

``size2`

符号付き型の場合のデータの大きさ

``scale2`

符号付き型の場合の小数部の二進数での桁数

デフォルトは,

`. short_fixed_spec 8:7, 8:7`

になっています.

ラベルは記述できません.

7.7.30 `. space "num"`

現在のロケーション・カウンタ値に, 引数 `num` で指定された値を加算します. `num` は整数値を指定します. `. space` によりロケーション・カウンタ値が増えた場合, 元のロケーション・カウンタ値と新しいロケーション・カウンタ値の間は 0 で埋められます.

`. space 16 ;ロケーションカウンタを 16 進める`

7.7.31 `. uaccum "accum_num"`

符号なし整数部つき固定小数点データを確保します. `accum_num` で指定される 10 進小数点表記の固定小数点数を `. accum_spec` 制御命令で指定された形式に従って変換し, メモリ上に配置します.

指定できるデータの値は, `. accum_spec` で指定した整数部の二進桁数により異なります. 整数部二進桁数が 4 であれば, $0 \leq accum_num < 16$ の範囲になります.

例:

`. uaccum 0.0`

`. uaccum 15.5`

7.7.32 `. uaccum. s "accum_num"`

符号なし短精度整数部つき固定小数点データを確保します. `accum_num` で指定される 10 進小数

点表記の固定小数点数を `.short_accum_spec` 制御命令で指定された形式に従って変換し、メモリ上に配置します。

指定できるデータの値は、`.short_accum_spec` で指定した整数部の二進桁数により異なります。整数部二進桁数が 4 であれば、 $0 \leq accum_num < 16$ の範囲になります。

例:

```
.uaccum.s 0.0
.uaccum.s 15.5
```

7.7.33 `.uaccum.l "accum_num"`

符号なし長精度整数部つき固定小数点データを確保します。`accum_num` で指定される 10 進小数点表記の固定小数点数を `.long_accum_spec` 制御命令で指定された形式に従って変換し、メモリ上に配置します。

指定できるデータの値は、`.long_accum_spec` で指定した整数部の二進桁数により異なります。整数部二進桁数が 4 であれば、 $0 \leq accum_num < 16$ の範囲になります。

例:

```
.accum.l 0.0
.accum.l 15.5
```

7.7.34 `.ufixed "fixed_num"`

符号なし固定小数点データを確保します。`fixed_num` で指定される 10 進小数点表記の固定小数点数を `.fixed_spec` 制御命令で指定された形式に従って変換し、メモリ上に配置します。

指定できるデータは、 $0 \leq fixed_num < 1$ の範囲になります。

例:

```
.ufixed 0.1
.ufixed 0.9
```

7.7.35 `.ufixed.s "fixed_num"`

符号なし短精度固定小数点データを確保します。`fixed_num` で指定される 10 進小数点表記の固定小数点数を `.short_fixed_spec` 制御命令で指定された形式に従って変換し、メモリ上に配置します。

指定できるデータは、 $0 \leq fixed_num < 1$ の範囲になります。

例:

```
.ufixed.s 0.1
.ufixed.s 0.9
```

7.7.36 `.ufixed.l "fixed_num"`

符号なし長精度固定小数点データを確保します。`fixed_num` で指定される 10 進小数点表記の固定小数点数を `.long_fixed_spec` 制御命令で指定された形式に従って変換し、メモリ上に配置します。

指定できるデータは、 $0 \leq fixed_num < 1$ の範囲になります。

例:

```
.ufixed.l 0.1
.ufixed.l 0.9
```

7.7.37 .word_alignment "num"

語境界を引数 *num* で指定した値に設定します。
ラベルは記述できません。

例:

```
.word_alignment 8 ; 8 バイト境界に設定
```

8. 登録されているリソースモデルとパラメータ

Library basicfhmdb

Class computational

8.1 adder

Description

Adder

Parameter List

- *bit_width*: Bit-width of input and output data (1-64)
- *algorithm*: Addition algorithm (*default*: use operator, *rca*: ripple carry adder, *cla*: carry look ahead adder)

Function List

- *adc*: Addition with carry {<*result*, *cout*> = *adc*(*a*, *b*, *cin*)}
-

8.1.1 Function : *adc*

Description

Addition with carry

Input

- *a*: Data for addition
- *b*: Data for addition
- *cin*: Carry for addition

Output

- *result*: Result of $a + b$
- *cout*: Carry of $a + b$

Format

<*result*, *cout*> = *adc*(*a*, *b*, *cin*)

8.2 alu, mini_alu

Description

Arithmetic and logic unit

Parameter List

- *bit_width*: Bit-width of input and output data (1-64)
- *algorithm*: Addition algorithm (*default*: use operator, *rca*: ripple carry adder, *cla*: carry look ahead adder)

Function List

- *add*: Signed addition {<*result*, *flag*> = *add*(*a*, *b*)}

- `addu` : Unsigned addition $\{<result, flag> = addu(a, b)\}$
- `addi` : Signed add. with inc. $\{<result, flag> = addi(a, b)\}$
- `addiu` : Unsigned add. with inc. $\{<result, flag> = addiu(a, b)\}$
- `cadd` : Signed add. with sat. [1] $\{<result, flag> = cadd(a, b)\}$
- `caddu` : Unsigned add. with sat. [1] $\{<result, flag> = caddu(a, b)\}$
- `caddi` : Signed add. with inc. and sat. [1] $\{<result, flag> = caddi(a, b)\}$
- `caddiu` : Unsigned add. with inc. and sat. [1] $\{<result, flag> = caddiu(a, b)\}$
- `sub` : Signed subtraction $\{<result, flag> = sub(a, b)\}$
- `subu` : Unsigned subtraction $\{<result, flag> = subu(a, b)\}$
- `subd` : Signed sub. with dec. $\{<result, flag> = subd(a, b)\}$
- `subdu` : Unsigned sub. with dec. $\{<result, flag> = subdu(a, b)\}$
- `csub` : Signed sub. with sat. [1] $\{<result, flag> = csub(a, b)\}$
- `csubu` : Unsigned sub. with sat. [1] $\{<result, flag> = csubu(a, b)\}$
- `csubd` : Signed sub. with dec. and sat. [1] $\{<result, flag> = csubd(a, b)\}$
- `csubdu` : Unsigned sub. with dec. and sat. [1] $\{<result, flag> = csubdu(a, b)\}$
- `inc` : Signed increment $\{<result, flag> = inc(a)\}$
- `incu` : Unsigned increment $\{<result, flag> = incu(a)\}$
- `cinc` : Signed inc. with sat. [1] $\{<result, flag> = cinc(a)\}$
- `cincu` : Unsigned inc. with sat. [1] $\{<result, flag> = cincu(a)\}$
- `dec` : Signed decrement $\{<result, flag> = dec(a)\}$
- `decu` : Unsigned decrement $\{<result, flag> = decu(a)\}$
- `cdec` : Signed dec. with sat. [1] $\{<result, flag> = cdec(a)\}$
- `cdecu` : Unsigned dec. with sat. [1] $\{<result, flag> = cdecu(a)\}$
- `not` : Not operation $\{<result, flag> = not(a)\}$
- `and` : And operation $\{<result, flag> = and(a, b)\}$
- `or` : Or operation $\{<result, flag> = or(a, b)\}$
- `xor` : Xor operation $\{<result, flag> = xor(a, b)\}$
- `nor` : Nor operation $\{<result, flag> = nor(a, b)\}$
- `nxor` : Nxor operation $\{<result, flag> = nxor(a, b)\}$
- `nand` : Nand operation $\{<result, flag> = nand(a, b)\}$
- `cmp` : Comparison of signed data $\{flag = cmp(a, b)\}$
- `cmpu` : Comparison of unsigned data $\{flag = cmpu(a, b)\}$
- `cmpz` : Comparison with zero $\{flag = cmpz(a)\}$
- `max` : Signed greater data selection [1] $\{<result, flag> = max(a, b)\}$
- `maxu` : Unsigned greater data selection [1] $\{<result, flag> = maxu(a, b)\}$
- `min` : Signed less data selection [1] $\{<result, flag> = min(a, b)\}$
- `minu` : Unsigned less data selection [1] $\{<result, flag> = minu(a, b)\}$

[1]: Available only with alu.

8.2.1 Function : add

Description

Signed addition

Input

- `a` : Data for addition
- `b` : Data for addition

Output

- `result` : Result of $a + b$
- `flag` : Flags of operation (flag(3)= '0', flag(2): zero flag ('1' when result = "0..0"), flag(1): MSB of result, flag(0): overflow flag.)

Format

$\langle result, flag \rangle = add(a, b)$

8.2.2 Function : addu**Description**

Unsigned addition

Input

- a : Data for addition
- b : Data for addition

Output

- $result$: Result of $a + b$
- $flag$: Flags of operation (flag(3): carry, flag(2): zero flag ('1' when result = "0..0"), flag(1): MSB of result, flag(0)= '0'.)

Format

$\langle result, flag \rangle = addu(a, b)$

8.2.3 Function : addi**Description**

Signed addition with increment

Input

- a : Data for addition
- b : Data for addition

Output

- $result$: Result of $a + b + 1$
- $flag$: Flags of operation (flag(3): carry, flag(2): zero flag ('1' when result = "0..0"), flag(1): MSB of result, flag(0): overflow flag.)

Format

$\langle result, flag \rangle = addi(a, b)$

8.2.4 Function : addiu**Description**

Unsigned addition with increment

Input

- a : Data for addition
- b : Data for addition

Output

- $result$: Result of $a + b + 1$
- $flag$: Flags of operation (flag(3): carry, flag(2): zero flag ('1' when result = "0..0"), flag(1): MSB of result, flag(0): overflow flag.)

Format

$\langle result, flag \rangle = addiu(a, b)$

8.2.5 Function : cadd**Description**

Signed addition with saturation

Input

- a : Data for addition
- b : Data for addition

Output

- $result$: Result of $a + b$ with saturation
- $flag$: Flags of operation (flag(3): carry, flag(2): zero flag ('1' when result = "0..0"), flag(1): MSB

of result, flag(0): overflow flag.)

Format

$\langle result, flag \rangle = cadd(a, b)$

8.2.6 Function : caddu

Description

Unsigned addition with saturation

Input

- a : Data for addition
- b : Data for addition

Output

- $result$: Result of $a + b$ with saturation
- $flag$: Flags of operation (flag(3): carry, flag(2): zero flag ('1' when result = "0..0"), flag(1): MSB of result, flag(0): overflow flag.)

Format

$\langle result, flag \rangle = caddu(a, b)$

8.2.7 Function : caddi

Description

Signed addition with increment and saturation

Input

- a : Data for addition
- b : Data for addition

Output

- $result$: Result of $a + b + 1$ with saturation
- $flag$: Flags of operation (flag(3): carry, flag(2): zero flag ('1' when result = "0..0"), flag(1): MSB of result, flag(0): overflow flag.)

Format

$\langle result, flag \rangle = caddi(a, b)$

8.2.8 Function : caddiu

Description

Unsigned addition with increment with saturation

Input

- a : Data for addition
- b : Data for addition

Output

- $result$: Result of $a + b + 1$ with saturation
- $flag$: Flags of operation (flag(3): carry, flag(2): zero flag ('1' when result = "0..0"), flag(1): MSB of result, flag(0): overflow flag.)

Format

$\langle result, flag \rangle = caddiu(a, b)$

8.2.9 Function : sub

Description

Signed subtraction

Input

- a : Data for subtraction
- b : Data for subtraction

Output

- $result$: Result of $a - b$

- *flag*: Flags of operation (flag(3): carry, flag(2): zero flag ('1' when result = "0..0"), flag(1): MSB of result, flag(0): overflow flag.)

Format

$\langle result, flag \rangle = \text{sub}(a, b)$

8.2.10 Function : subu**Description**

Unsigned subtraction

Input

- *a*: Data for subtraction
- *b*: Data for subtraction

Output

- *result*: Result of $a - b$
- *flag*: Flags of operation (flag(3): carry, flag(2): zero flag ('1' when result = "0..0"), flag(1): MSB of result, flag(0): overflow flag.)

Format

$\langle result, flag \rangle = \text{subu}(a, b)$

8.2.11 Function : subd**Description**

Signed subtraction with decrement

Input

- *a*: Data for subtraction
- *b*: Data for subtraction

Output

- *result*: Result of $a - b - 1$
- *flag*: Flags of operation (flag(3): carry, flag(2): zero flag ('1' when result = "0..0"), flag(1): MSB of result, flag(0): overflow flag.)

Format

$\langle result, flag \rangle = \text{subd}(a, b)$

8.2.12 Function : subdu**Description**

Unsigned subtraction with decrement

Input

- *a*: Data for subtraction
- *b*: Data for subtraction

Output

- *result*: Result of $a - b - 1$
- *flag*: Flags of operation (flag(3): carry, flag(2): zero flag ('1' when result = "0..0"), flag(1): MSB of result, flag(0): overflow flag.)

Format

$\langle result, flag \rangle = \text{subdu}(a, b)$

8.2.13 Function : csb**Description**

Signed subtraction with saturation

Input

- *a*: Data for subtraction
- *b*: Data for subtraction

Output

- *result* : Result of $a - b$ with saturation
- *flag* : Flags of operation (flag(3): carry, flag(2): zero flag ('1' when result = "0..0"), flag(1): MSB of result, flag(0): overflow flag.)

Format

$\langle result, flag \rangle = csub(a, b)$

8.2.14 Function : csubu

Description

Unsigned subtraction with saturation

Input

- *a* : Data for subtraction
- *b* : Data for subtraction

Output

- *result* : Result of $a - b$ with saturation
- *flag* : Flags of operation (flag(3): carry, flag(2): zero flag ('1' when result = "0..0"), flag(1): MSB of result, flag(0): overflow flag.)

Format

$\langle result, flag \rangle = csubu(a, b)$

8.2.15 Function : csubd

Description

Signed subtraction with decrement and saturation

Input

- *a* : Data for subtraction
- *b* : Data for subtraction

Output

- *result* : Result of $a - b - 1$ with saturation
- *flag* : Flags of operation (flag(3): carry, flag(2): zero flag ('1' when result = "0..0"), flag(1): MSB of result, flag(0): overflow flag.)

Format

$\langle result, flag \rangle = csubd(a, b)$

8.2.16 Function : csubdu

Description

Unsigned subtraction with decrement with saturation

Input

- *a* : Data for subtraction
- *b* : Data for subtraction

Output

- *result* : Result of $a - b - 1$ with saturation
- *flag* : Flags of operation (flag(3): carry, flag(2): zero flag ('1' when result = "0..0"), flag(1): MSB of result, flag(0): overflow flag.)

Format

$\langle result, flag \rangle = csubdu(a, b)$

8.2.17 Function : inc

Description

Signed increment

Input

- *a* : Data for increment

Output

- *result*: Result of $a + 1$
- *flag*: Flags of operation (flag(3): carry, flag(2): zero flag ('1' when result = "0..0"), flag(1): MSB of result, flag(0): overflow flag.)

Format

$\langle result, flag \rangle = inc(a)$

8.2.18 Function : incu**Description**

Unsigned increment

Input

- *a*: Data for increment

Output

- *result*: Result of $a + 1$
- *flag*: Flags of operation (flag(3): carry, flag(2): zero flag ('1' when result = "0..0"), flag(1): MSB of result, flag(0): overflow flag.)

Format

$\langle result, flag \rangle = incu(a)$

8.2.19 Function : cinc**Description**

Signed increment with saturation

Input

- *a*: Data for increment

Output

- *result*: Result of $a + 1$ with saturation
- *flag*: Flags of operation (flag(3): carry, flag(2): zero flag ('1' when result = "0..0"), flag(1): MSB of result, flag(0): overflow flag.)

Format

$\langle result, flag \rangle = cinc(a)$

8.2.20 Function : cincu**Description**

Unsigned increment with saturation

Input

- *a*: Data for increment

Output

- *result*: Result of $a + 1$ with saturation
- *flag*: Flags of operation (flag(3): carry, flag(2): zero flag ('1' when result = "0..0"), flag(1): MSB of result, flag(0): overflow flag.)

Format

$\langle result, flag \rangle = cincu(a)$

8.2.21 Function : dec**Description**

Signed decrement

Input

- *a*: Data for decrement

Output

- *result*: Result of $a - 1$
- *flag*: Flags of operation (flag(3): carry, flag(2): zero flag ('1' when result = "0..0"), flag(1): MSB

of result, flag(0): overflow flag.)

Format

$\langle result, flag \rangle = dec(a)$

8.2.22 Function : decu

Description

Unsigned decrement

Input

- a : Data for decrement

Output

- $result$: Result of $a - 1$
- $flag$: Flags of operation (flag(3): carry, flag(2): zero flag ('1' when result = "0..0"), flag(1): MSB of result, flag(0): overflow flag.)

Format

$\langle result, flag \rangle = decu(a)$

8.2.23 Function : cdec

Description

Signed decrement with saturation

Input

- a : Data for decrement

Output

- $result$: Result of $a - 1$ with saturation
- $flag$: Flags of operation (flag(3): carry, flag(2): zero flag ('1' when result = "0..0"), flag(1): MSB of result, flag(0): overflow flag.)

Format

$\langle result, flag \rangle = cdec(a)$

8.2.24 Function : cdecu

Description

Unsigned decrement with saturation

Input

- a : Data for decrement

Output

- $result$: Result of $a - 1$ with saturation
- $flag$: Flags of operation (flag(3): carry, flag(2): zero flag ('1' when result = "0..0"), flag(1): MSB of result, flag(0): overflow flag.)

Format

$\langle result, flag \rangle = cdecu(a)$

8.2.25 Function : not

Description

Not operation

Input

- a : Data for operation

Output

- $result$: Result of not a
- $flag$: Flags of operation (flag(3): carry, flag(2): zero flag ('1' when result = "0..0"), flag(1): MSB of result, flag(0): overflow flag.)

Format

$\langle result, flag \rangle = not(a)$

8.2.26 Function : and

Description

And operation

Input

- *a* : Data for operation
- *b* : Data for operation

Output

- *result* : Result of *a* and *b*
- *flag* : Flags of operation (flag(3): carry, flag(2): zero flag ('1' when result = "0..0"), flag(1): MSB of result, flag(0): overflow flag.)

Format

<*result*, *flag*> = and(*a*, *b*)

8.2.27 Function : or

Description

Or operation

Input

- *a* : Data for operation
- *b* : Data for operation

Output

- *result* : Result of *a* or *b*
- *flag* : Flags of operation (flag(3): carry, flag(2): zero flag ('1' when result = "0..0"), flag(1): MSB of result, flag(0): overflow flag.)

Format

<*result*, *flag*> = or(*a*, *b*)

8.2.28 Function : xor

Description

Xor operation

Input

- *a* : Data for operation
- *b* : Data for operation

Output

- *result* : Result of *a* xor *b*
- *flag* : Flags of operation (flag(3): carry, flag(2): zero flag ('1' when result = "0..0"), flag(1): MSB of result, flag(0): overflow flag.)

Format

<*result*, *flag*> = xor(*a*, *b*)

8.2.29 Function : nor

Description

Nor operation

Input

- *a* : Data for operation
- *b* : Data for operation

Output

- *result* : Result of *a* nor *b*
- *flag* : Flags of operation (flag(3): carry, flag(2): zero flag ('1' when result = "0..0"), flag(1): MSB of result, flag(0): overflow flag.)

Format

$\langle result, flag \rangle = \text{nor}(a, b)$

8.2.30 Function : nxor**Description**

Nxor operation

Input

- a : Data for operation
- b : Data for operation

Output

- $result$: Result of a nxor b
- $flag$: Flags of operation (flag(3): carry, flag(2): zero flag ('1' when result = "0..0"), flag(1): MSB of result, flag(0): overflow flag.)

Format

$\langle result, flag \rangle = \text{nxor}(a, b)$

8.2.31 Function : nand**Description**

Nand operation

Input

- a : Data for operation
- b : Data for operation

Output

- $result$: Result of a nand b
- $flag$: Flags of operation (flag(3): carry, flag(2): zero flag ('1' when result = "0..0"), flag(1): MSB of result, flag(0): overflow flag.)

Format

$\langle result, flag \rangle = \text{nand}(a, b)$

8.2.32 Function : cmp**Description**

Comparison of signed data

Input

- a : Data for comparison
- b : Data for comparison

Output

- $flag$: Flags of operation (flag(3): carry, flag(2): zero flag ('1' when result = "0..0"), flag(1): MSB of result, flag(0): overflow flag.)

Format

$flag = \text{cmp}(a, b)$

8.2.33 Function : cmpu**Description**

Comparison of unsigned data

Input

- a : Data for comparison
- b : Data for comparison

Output

- $flag$: Flags of operation (flag(3): carry, flag(2): zero flag ('1' when result = "0..0"), flag(1): MSB of result, flag(0): overflow flag.)

Format

flag = cmp(*a*, *b*)

8.2.34 Function : cmpz**Description**

Comparison of signed data with zero

Input

- *a* : Data for comparison

Output

- *flag*: Flags of operation (flag(3): carry, flag(2): zero flag ('1' when result = "0..0"), flag(1): MSB of result, flag(0): overflow flag.)

Format

flag = cmpz(*a*)

8.2.35 Function : max**Description**

Signed greater data selection

Input

- *a* : Data for operation
- *b* : Data for operation

Output

- *result* : Max value (The value is *a* if ($a \geq b$) else *b*.)
- *flag*: Flags of operation (flag(3): carry, flag(2): zero flag ('1' when result = "0..0"), flag(1): MSB of result, flag(0): overflow flag.)

Format

<*result*, *flag*> = max(*a*, *b*)

8.2.36 Function : maxu**Description**

Unsigned greater data selection

Input

- *a* : Data for operation
- *b* : Data for operation

Output

- *result* : Max value (The value is *a* if ($a \geq b$) else *b*.)
- *flag*: Flags of operation (flag(3): carry, flag(2): zero flag ('1' when result = "0..0"), flag(1): MSB of result, flag(0): overflow flag.)

Format

<*result*, *flag*> = maxu(*a*, *b*)

8.2.37 Function : min**Description**

Signed less data selection

Input

- *a* : Data for operation
- *b* : Data for operation

Output

- *result* : Min value (The value is *a* if ($a \leq b$) else *b*.)
- *flag*: Flags of operation (flag(3): carry, flag(2): zero flag ('1' when result = "0..0"), flag(1): MSB of result, flag(0): overflow flag.)

Format

$\langle result, flag \rangle = \min(a, b)$

8.2.38 Function : minu

Description

Unsigned less data selection

Input

- a : Data for operation
- b : Data for operation

Output

- $result$: Min value (The value is a if $(a \leq b)$ else b .)
- $flag$: Flags of operation (flag(3): carry, flag(2): zero flag ('1' when result = "0..0"), flag(1): MSB of result, flag(0): overflow flag.)

Format

$\langle result, flag \rangle = \text{minu}(a, b)$

8.3 barrelshifter

Description

Rotater

Parameter List

bit_width : Bit-width of input and output data (4, 8, 16, 32, 64, 128)

Function List

- sl : Left rotation $\{data_out = sl(data_in, ctrl)\}$
 - sr : Right rotation $\{data_out = sr(data_in, ctrl)\}$
-

8.3.1 Function : sl

Description

Left rotation

Input

$data_in$: Data for operation

- $ctrl$: Shift amount

Output

- $data_out$: Result of $data_in \ll ctrl$

Format

$data_out = sl(data_in, ctrl)$

8.3.2 Function : sr

Description

Right rotation

Input

$data_in$: Data for operation

- $ctrl$: Shift amount

Output

- $data_out$: Result of $data_in \gg ctrl$

Format

$data_out = sr(data_in, ctrl)$

8.4 divider

Description

Divider

Parameter List

- *bit_width* : Bit-width of input and output data (Every 4 bits value between 4-64, and 128)
- *algorithm* : Division algorithm (*seq*: sequential type divider, *array*: array type divider)
- *adder_algorithm* : Addition algorithm (*default*: use operator, *rca*: ripple carry adder, *cla*: carry look ahead adder)
- *data_type* : Data type (*unsigned*: unsigned data, *abs*: absolute data, *two_complement*: twos complement data)

Function List

- *div* : Signed division $\{<q, r, flag> = \text{div}(a, b)\}$
- *divu* : Unsigned division $\{<q, r, flag> = \text{divu}(a, b)\}$
- *diva* : Absolute division $\{<q, r, flag> = \text{diva}(a, b)\}$

8.4.1 Function : div

Description

Signed division

Input

- *a* : Data for division
- *b* : Data for division

Output

- *q* : Result of a / b
- *r* : Remainder of a / b
- *flag* : Error flag (if input data *b* is 0, *flag* becomes '1')

Format

$<q, r, flag> = \text{div}(a, b)$

Attention:

This function can be use if *data_type* is *two_complement*.

8.4.2 Function : divu

Description

Unsigned division

Input

- *a* : Data for division
- *b* : Data for division

Output

- *q* : Result of a / b
- *r* : Remainder of a / b
- *flag* : Error flag (if input data *b* is 0, *flag* becomes '1')

Format

$<q, r, flag> = \text{divu}(a, b)$

Attention:

This function can be use if *data_type* is *unsigned* or *two_complement*.

8.4.3 Function : diva

Description

Absolute division

Input

- *a* : Data for division

- *b*: Data for division

Output

- *q*: Result of a / b
- *r*: Remainder of a / b
- *flag*: Error flag (if input data *b* is 0, *flag* becomes '1')

Format

$\langle q, r, \text{flag} \rangle = \text{diva}(a, b)$

Attention:

This function can be use if *data_type* is *abs*.

8.5 extender

Description

Extender

Parameter List

- *bit_width*: Bit-width of input data (1-63)
- *bit_width_out*: Bit-width of output data (8, 16, 32, 64)

Function List

- *zero*: Zero extension $\{data_out = \text{zero}(data_in)\}$
- *sign*: Sign extension $\{data_out = \text{sign}(data_in)\}$

Attention:

Value of parameter *bit_width* must be less than value of parameter *bit_width_out*.

8.5.1 Function : zero

Description

Zero extension

Input

- *data_in*: Data for extension

Output

- *data_out*: Result of zero extension

Format

$data_out = \text{zero}(data_in)$

8.5.2 Function : sign

Description

Sign extension

Input

- *data_in*: Data for extension

Output

- *data_out*: Result of sign extension

Format

$data_out = \text{sign}(data_in)$

8.6 multiplexor

Description

Multiplexor

Parameter List

- *bit_width*: Bit-width of input data (1-32, 64, 128)
- *number_of_ports*: Number of input ports (2-16)

Function List

- `sel` : Data selection $\{data_out = sel(data_in0, ..)\}$
-

8.6.1 Function : `sel`

Description

Data selection

Input

`data_in_i` : Data for selection

Output

- `data_out` : Result of zero extension

Format

- `data_out = sel(data_in0, ..)`

Attention:

The number of `data_in_i` is equal to `number_of_ports`.

8.7 multiplier

Description

Multiplier

Parameter List

- `bit_width` : Bit-width of input and output data (Every 4 bits value between 4-64, and 128)
- `algorithm` : Multiplication algorithm (*default*: use operator, *seq*: sequential type multiplier, *array*: array type multiplier)
- `adder_algorithm` : Addition algorithm (*default*: use operator, *rca*: ripple carry adder, *cla*: carry look ahead adder)
- `data_type` : Data type (*unsigned*: unsigned data, *abs*: absolute data, *two_complement*: twos complement data)

Function List

- `mul` : Signed multiplication $\{result = mul(a, b)\}$
- `mulu` : Unsigned multiplication $\{result = mulu(a, b)\}$
- `mula` : Absolute multiplication $\{result = mula(a, b)\}$

Attention:

If `algorithm` is *default*, `adder_algorithm` is ignored.

8.7.1 Function : `mul`

Description

Signed multiplication

Input

- `a` : Data for multiplication
- `b` : Data for multiplication

Output

- `result` : Result of $a * b$

Format

`result = mul(a, b)`

Attention:

This function can be use if `data_type` is *two_complement*.

8.7.2 Function : `mulu`

Description

Unsigned multiplication

Input

- *a* : Data for multiplication
- *b* : Data for multiplication

Output

- *result* : Result of $a * b$

Format

result = mulu(*a*, *b*)

Attention:

This function can be use if *data_type* is *unsigned* or *two_complement*.

8.7.3 Function : mula

Description

Absolute multiplication

Input

- *a* : Data for multiplication
- *b* : Data for multiplication

Output

- *result* : Result of $a * b$

Format

result = mula(*a*, *b*)

Attention:

This function can be use if *data_type* is *abs*.

8.8 shifter

Description

Shifter

Parameter List

- *bit_width* : Bit-width of input data (Every 4 bits value between 4-64)
- *amount* : Shift amount (*variable*, 1, 2, 4, 8, 16, 32)

Function List

- sll : Left logical shift {*data_out* = sll(*data_in*[, *mode*])}
- sla : Left arithmetic shift {*data_out* = sla(*data_in*[, *mode*])}
- srl : Right logical shift {*data_out* = srl(*data_in*[, *mode*])}
- sra : Right arithmetic shift {*data_out* = sra(*data_in*[, *mode*])}

Attention:

If *amount* is not *variable*, it must be less than *bit_width*.

8.8.1 Function : sll

Description

Left logical shift

Input

- *data_in* : Data for shift
- *mode* : Shift amount (*mode* exists only if *amount* is variable.)

Output

- *data_out* : Result of left logical shift

Format

data_out = sll(*data_in*[, *mode*])

8.8.2 Function : sla

Description

Left arithmetic shift

Input

- *data_in* : Data for shift
- *mode* : Shift amount (*mode* exists only if *amount* is variable.)

Output

- *data_out* : Result of left arithmetic shift

Format

data_out = sla(*data_in*[, *mode*])

8.8.3 Function : srl

Description

Right logical shift

Input

- *data_in* : Data for shift
- *mode* : Shift amount (*mode* exists only if *amount* is variable.)

Output

- *data_out* : Result of right logical shift

Format

data_out = srl(*data_in*[, *mode*])

8.8.4 Function : sra

Description

Right arithmetic shift

Input

- *data_in* : Data for shift
- *mode* : Shift amount (*mode* exists only if *amount* is variable.)

Output

- *data_out* : Result of right arithmetic shift

Format

data_out = sra(*data_in*[, *mode*])

Class storage

8.9 register

Description

Register

Parameter List

- *bit_width* : Bit-width of input data (1-80, 128)

Function List

- write : Data write {null = write(*data_in*)}
 - read : Data read {*data_out* = read()}
-

8.9.1 Function : write

Description

Data write

Input

- *data_in* : Data for write

Output

(No output)

Format

`null = write(data_in)`

8.9.2 Function : read

Description

Data read

Input

(No input)

Output

- *data_out*: Registered data

Format

`data_out = read()`

8.10 registerfile

Description

Registerfile

Parameter List

- *bit_width*: Bit-width of input data (Every 4 bits value between 4-64, and 128)
- *num_register*: The number of registers (4, 8, 16, 32)
- *num_read_port*: The number of read ports (1, 2, 4)
- *num_write_port*: The number of write ports (1, 2, 4)

Function List

- *write_i*: Data write {`null = write(data_ini, w_seli)`}
 - *read_i*: Data read {`data_outi = read(r_seli)`}
-

8.10.1 Function : write_i

Description

Data write with write port *i*

Input

- *data_in_i*: Data for write
- *w_sel_i*: Register number

Output

(No output)

Format

`null = write(data_ini, w_seli)`

Attention:

For all $0 \leq i \leq \text{num_write_port}$, *write_i* function is available.

8.10.2 Function : read_i

Description

Data read with read port *i*

Input

- *r_sel_i*: Register number

Output

- *data_out_i*: Registered data

Format

`data_outi = read(r_seli)`

Attention:

For all $0 \leq i \leq \text{num_read_port}$, `readi` function is available.

Library workdb

Class FHM_work

8.11 fwu**Description**

Forwarding (register-bypassing) unit.

Parameter List

- `bit_width`: Bit-width of data (4, 8, 16, 24, 32, 64, 128)
- `addr_width`: Bit-width of operand, as same as registerfile's one (1, 2, 3, 4, 5, 6, 7, 8, 12, 16, 24, 32, 64, 128)
- `stage_number`: the number of stages from an operand fetch stage to a write back stage. This number represents the number of stages to feed a data into the fwu (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

Function List

- `forward`: Get forwarded data { `data = forward(register_number, data_from_register)` }
- `forwardn`: Set forwarding data { `null = forwardn(register_number, data_to_forward)` } (*n*: 1 to `stage_number`)

8.11.1 Function : forward**Description**

Get forwarded data

Input

`register_number`: Id of register file

`data_from_register`: Data read from register file

Output

Forwarded data if any data forwarded by `forwardn` function, otherwise data from register file.

Format

`data = forward(register_number, data_from_register)`

8.11.2 Function : Forwardn**Description**

Send data to forwarding unit.

Input

`register_number`: Id of register file

`data_to_forward`: Data to send as forwarding value.

Output

(No output)

Format

`null = forwardn(register_number, data_to_forward)`

Example

A.1. Operations ADD, SUB, ...

stage 2:

`tmp0 = GPR.read0(rs0);`

`tmp1 = GPR.read1(rs1);`

`source0 = FWU0.forward(rs0, tmp0);`

```

    source1 = FWU1.forward(rs1, tmp1);
stage 3:
    wire flag[3:0];
    <result, flag> = ALU.add(source0, source1);
    null = FWU0.forward1(rd, result);
    null = FWU1.forward1(rd, result);
stage 4:
    null = FWU0.forward2(rd, result);
    null = FWU1.forward2(rd, result);
stage 5:
    null = GPR.write0(rd, result);
    null = FWU0.forward3(rd, result);
    null = FWU1.forward3(rd, result);

```

A.2. Operations ADDI, SUBI, ... (one operand)

```

stage 2:
    tmp0 = GPR.read0(rs);
    source0 = FWU0.forward(rs, tmp0);
    source1 = EXT0.sign(imm);
stage 3:
    wire flag[3:0];
    <result, flag> = ALU.add(source0, source1);
    null = FWU0.forward1(rd, result);
    null = FWU1.forward1(rd, result);
stage 4:
    null = FWU0.forward2(rd, result);
    null = FWU1.forward2(rd, result);
stage 5:
    null = GPR.write0(rd, result);
    null = FWU0.forward3(rd, result);
    null = FWU1.forward3(rd, result);

```

A.3. Operations LW, LH, LB (a result is ready in the 4th stage)

```

stage 2:
    tmp0 = GPR.read0(rs0);
    source0 = FWU0.forward(rs0, tmp0);
    source1 = EXT0.sign(const);
stage 3:
    wire flag[3:0];
    <addr, flag> = ALU.add(source0, source1);
stage 4:
    wire addr_err;
    <result, addr_err> = DMAU.ld_32(addr);
    null = FWU0.forward2(rd, result);
    null = FWU1.forward2(rd, result);
stage 5:
    null = GPR.write0(rd, result);
    null = FWU0.forward3(rd, result);
    null = FWU1.forward3(rd, result);

```

A.3. Operations SW, SH, SB (no data to be forwarded)

```

stage 2:
    tmp0 = GPR.read0(rs0);

```

```

    tmp1    = GPR.read1(rs1);
    data    = FWU0.forward(rs0, tmp0);
    base    = FWU1.forward(rs1, tmp1);
    offset = EXT0.sign(const);
stage 3:
    wire flag[3:0];
    <addr, flag> = ALU.add(base, offset);
stage 4:
    wire addr_err;
    addr_err = DMAU.s_32(addr, data);
stage 5:

```

8.12 mifu

Description

Memory interface unit

Parameter List

- *bit_width* : Bit-width of data (8, 16, 32, 64, 128)
- *address_space* : The size of address space (16, 32, 64, 128)
- *access_width* : Minimum access bit-width (8, 16, 32, 64, 128)
- *access_mode* : Access cycle (single_cycle, multi_cycle)
- *type* : Access type (read_only, read_write)

Function List

- *ld_i* : Signed data load {<*data_out*, *addr_err*>= *ld_i(addr)*}
- *ldu_i* : Unsigned data load {<*data_out*, *addr_err*>= *ldu_i(addr)*}
- *s_i* : Data store {*addr_err*= *s_i(addr, data_in)*}

Attention:

access_width must be less than or equal to *bit_width*.

8.12.1 Function : *ld_i*

Description

Signed *i*-bit data load

Input

- *addr* : Address of data memory

Output

- *data_out* : Loaded data
- *addr_err* : Error signal (If *addr* is outside range of data memory, *addr_err* becomes '1'.)

Format

<*data_out*, *addr_err*>= *ld_i(addr)*

Attention:

For all *i* where *i* is multiple of *access_width* and submultiple of *bit_width*, *ld_i* is available.

8.12.2 Function : *ldu_i*

Description

Unsigned *i*-bit data load

Input

- *addr* : Address of data memory

Output

- *data_out* : Loaded data
- *addr_err* : Error signal (If *addr* is outside range of data memory, *addr_err* becomes '1'.)

Format

$\langle data_out, addr_err \rangle = ldu_i(addr)$

Attention:

For all $i \neq bit_width$ where i is multiple of $access_width$ and submultiple of bit_width , ldu_i is available.

8.12.3 Function : s_i

Description

i -bit data store

Input

- $addr$: Address of data memory
- $data_in$: Data for store

Output

- $addr_err$: Error signal (If $addr$ is outside range of data memory, $addr_err$ becomes '1'.)

Format

$addr_err = s_i(addr, data_in)$

Attention:

For all i where i is multiple of $access_width$ and submultiple of bit_width , ld_i is available.

8.13 pcu

Description

Program counter unit

Parameter List

- bit_width : Bit-width of data (4, 8, 16, 32, 64, 128)
- $increment_step$: Increment amount (1, 2, 4, 8)
- $adder_algorithm$: Addition algorithm (*default*: use operator, *rca*: ripple carry adder, *cla*: carry look ahead adder)

Function List

- inc : Increment {null = $inc()$ }
- $write$: Data store {null = $write(data_in)$ }
- $read$: Data load { $data_out = read()$ }

8.13.1 Function : inc

Description

Increment

Input

(No input)

Output

(No output)

Format

null = $inc()$

8.13.2 Function : $write$

Description

Data store

Input

- $data_in$: Data for store

Output

(No output)

Format

`null = write(data¥¥in)`

8.13.3 Function : read**Description**

Data load

Input

(No input)

Output

- *data_out* : Loaded data

Format

`data_out = read()`

8.14 wire_in**Description**

Wire for input port

Parameter List

- *bit_width* : Bit-width of port (1-80, 128)

Function List

- `read` : Data read {*int_port* = read() }
-

8.14.1 Function : read**Description**

Data read

Input

(No input)

Output

- *int_port* : Read data

Format

`int_port = read()`

8.15 wire_out**Description**

Wire for output port

Parameter List

- *bit_width* : Bit-width of port (1-80, 128)
- *default_output* : Default output value (*fix_to_0*: ``0..0'', *fix_to_1*: ``1..1'', *keep*: keep previous output value)

Function List

- `write` : Data write {`null = write(int_port)` }
-

8.15.1 Function : write**Description**

Data write

Input

- *int_port* : Write data

Output

(No output)

Format

null = write(*int_port*)

8.16 wire_inout**Description**

Wire for input and output port

Parameter List

- *bit_width* : Bit-width of port (1-80, 128)

Function List

- write : Data write {null = write(*int_in_port*)}
 - read : Data read {*int_port* = read()}
-

8.16.1 Function : write**Description**

Data write

Input

- *int_port* : Write data

Output

(No output)

Format

null = write(*int_in_port*)

8.16.2 Function : read**Description**

Data read

Input

(No input)

Output

- *int_out_port* : Read data

Format

int_out_port = read()

9. 応用プログラム開発環境として生成されるファイルとその使い方

ASIP Meister Standard は、GNU C compiler と GNU binutils を生成するために、GCC(<http://gcc.gnu.org/>)および ArchC(<http://sourceforge.net/projects/archc>)用の記述を生成します。また、その記述を利用して、GNU C compiler と GNU binutils を生成できます。

生成されるプログラムの詳細な使用法、オプション等は、GNU のマニュアル(<http://www.gnu.org/manual/>)をご参照下さい。

9.1 出力されるプログラム一式

12. Compiler Generationフェーズが正常に終了すると、下記に示すファイルが生成されます。プログラム一式は、設計データファイル名が **"model.pdb"** だった場合、**"meister"** の中の **"model.swgen/bin"** というフォルダ中に生成されます。

- brownie32-elf-gcc
- brownie32-elf-addr2line
- brownie32-elf-ar
- brownie32-elf-as
- brownie32-elf-c++filt
- brownie32-elf-ld
- brownie32-elf-nm
- brownie32-elf-objcopy
- brownie32-elf-objdump
- brownie32-elf-ranlib
- brownie32-elf-readelf
- brownie32-elf-size
- brownie32-elf-strings
- brownie32-elf-strip

C 言語から実行形式 elf までの作成の流れは、12. Compiler Generationを参照してください。

9.2 出力されるプログラムの主な使用法

生成されたプログラムの詳細な使用法、オプション等は、GNU のマニュアルをご参照ください。

9.2.1 brownie32-elf-gcc

GNU C コンパイラです。C プログラムを入力として、そのアセンブリプログラムを出力します。(オプション **-S** を指定して下さい)。使用する時には、オプション **-o** で出力ファイル名を指定することができます。

また、最適化オプション **-O2** を指定することができます。

使用例：

```
$ brownie32-elf-gcc -S sample.c -o sample.s
```

sample.c という名前の C 言語で書かれたファイルをコンパイルし、sample.s という名前の brownie のアセンブリ言語を生成する。

```
$ brownie32-elf-gcc -S sample.c -O2 -o sample.c
```

sample.c という名前の C 言語で書かれたファイルをコンパイルし、sample.s という名前の brownie のアセンブリ言語を生成する。

9.2.2 brownie32-elf-addr2line

addr2line は、プログラムアドレスをファイル名と行番号に変換します。

9.2.3 brownie32-elf-ar

アーカイブの作成、変更、そして、それらの抽出を行います。

9.2.4 brownie32-elf-as

GNU アセンブラです。Brownie のアセンブリ言語で書かれたプログラムを入力として、オブジェクトファイルへ変換を行います。

使用例：

```
$ brownie32-elf-as -o sample.o sample.s
```

sample.s という名前のアセンブリ言語で書かれたファイルをコンパイルし、sample.o という名前のオブジェクトファイルを生成する。

9.2.5 brownie32-elf-c++filt

C++言語のための機能です。現在、ASIP Meister Standard から出力される C コンパイラは C++ には対応していないので、使用できません。

9.2.6 brownie32-elf-ld

複数のオブジェクトファイルやアーカイブファイルを結合し、シンボルの参照を結び付け、実行ファイルを生成します。

使用例：

```
$ brownie32-elf-ld -o sample -T link.sc sample.o
```

sample.o という名前のオブジェクトファイルを、リンカスクリプト link.sc の指示に従い、elf 形式の実行時ファイル sample を生成します。

9.2.7 brownie32-elf-nm

オブジェクトファイルのシンボルをリストアップします。

9.2.8 brownie32-elf-objcopy

オブジェクトファイルの内容を別のファイルにコピーします。

9.2.9 brownie32-elf-objdump

オブジェクトファイルに関する情報を表示します。引数として与えられたオブジェクトファイルの逆アセンブルを行います。

9.2.10 brownie32-elf-ranlib

アーカイブの内容の索引を生成し、それをアーカイブに保存します。

9.2.11 brownie32-elf-readelf

一つ以上の ELF フォーマットのオブジェクトファイルの情報を表示します。

9.2.12 brownie32-elf-size

引数として与えられたオブジェクトやアーカイブファイルに対し、セクションの大きさと全体の大きさをリストアップします。

9.2.13 brownie32-elf-strings

引数として与えられた file に対し、出力可能な文字が含まれている場合、それを出力します。

9.2.14 brownie32-elf-strip

オブジェクトファイルの中からシンボルを廃棄します。

索引

A

Arch. Level Estimation 3, 4, 11, 40
 Area 11, 20, 40
 ASIP Meister(製品名) 1, 2, 3, 5, 6, 7, 8, 11, 19, 27, 29,
 44, 61
 ASIPmeister(コマンド) 1, 5
 ASIPmeister.setup 5
 Attribute 12, 15, 27, 28

C

Clock 19
 Condition 38
 CPU 11, 14
 CPU Type 11, 14

D

Data hazard interlock 13, 15
 Data memory 23
 data_memory_acknowledge_bus 27
 data_memory_address_bus 27
 data_memory_data_bus 27
 data_memory_request_bus 27
 data_memory_write_mode_bus 28
 decode 12
 Decode Stage 12, 15
 Delay 20, 40
 Delayed branch 13, 15
 Design Goal 3, 6, 8, 10, 11, 14, 41, 45, 51
 Design Goal & Arch. Design 3, 6, 8, 10, 11, 14, 41,
 45, 51
 Design Priority 11, 14
 Direction 28
 DMAU 27

E

Entity Name 27, 28
 Exception 29, 37, 44, 46
 exec 12

F

fetch 12
 FHM 10, 11, 19, 20
 Fhm workname 11, 14
 Field Type 30, 32, 34, 36, 37
 Function Set 19

G

Goal Area 11, 14
 Goal Delay 11, 14
 Goal Power S 11, 14

H

HDL 2, 3, 4, 8, 11, 27, 42, 49, 53
 HDL Generation 3, 4, 9, 27, 42, 49, 53

I

IMAU 27
 Instruction... 3, 23, 29, 30, 32, 33, 34, 35, 36, 37, 44,
 45, 46
 Instruction Definition 3, 29, 36, 44, 45, 46
 Instruction memory 23
 Instruction register 23
 instruction_memory_address_bus 27
 instruction_memory_data_bus 27
 Interface Definition 3, 11, 26, 28
 Interrupt Name 38

L

LSB 30, 32, 34, 37

M

Macro 44, 47, 48, 49
 Max data bit width 14, 15
 Max inst. bit width 13, 15
 meister(ワーキングディレクトリ) 8, 43, 51
 memory_read 12
 memory_write 12
 Micro Op. Description 3, 4, 44, 45, 46, 47, 48, 49
 MSB 30, 32, 34, 37
 Multi cycle interlock 13, 15

N

Num. of delayed slot 13, 15
 Number of Common Stages 15
 Number of Stages 15

P

PATH 5
 Pipeline 11
 Port Name 28
 Power 11, 20, 40

Project name..... 11, 14

R

Register bypass..... 13, 15
register_read 12
register_write 12
Reset 19
Resource Declaration 3, 16, 17, 22, 49
Revision No..... 11, 14

S

Signal Type..... 26, 28
Stage 12, 45, 46

T

Type 30, 32, 33, 34, 35, 36, 37, 38

U

Use as..... 19

V

Valid 27, 28, 38
Value 19, 31, 33, 35, 36, 37
VHDL..... 2, 21, 27, 28, 42, 43, 49, 50, 51, 58

あ

アクリッジバス 27
アドレスバス 27

い

インストール 1, 5, 61
インストールガイド 1

え

エンジン選択肢 42, 50, 58
エンティティ名 11, 26, 27, 28, 51

か

カレントディレクトリ 8, 43, 51

く

クロック 19, 27

し

シミュレーション 2, 49, 50, 51, 52

す

ステージ番号 12
ステージ名 12, 15, 45
ストア 1

せ

セットアップ 1

て

データタイプ 19, 21
データバス 27
データメモリ 19
デコード命令 15
デコード命令のステージ番号 15

ね

ネーミングルール 28

は

バージョン管理 11
バージョン情報 10, 11
パイプライン段数 15
バス 1, 5, 11, 13, 15, 40

ひ

ビット幅 11, 13, 14, 15, 19, 20, 21

ふ

フィールド数 32, 33, 37
プログラムカウンタ 19

ほ

ポート名 28, 38

ま

マイクロ動作 3, 12
マイクロ動作記述 2, 40, 44, 47, 48
マイクロ動作記述の書き方 44, 45, 46, 48
マクロ定義 44, 47
マクロ名 48

も

モデル 2, 19, 43, 49, 50, 51, 52, 58, 60

り

リクエストバス	27
リセット	19, 27
リソースモデル	11, 19, 21, 40, 100

れ

レジスタファイル	19
レベル	60

わ

ワーキングディレクトリ	8
-------------------	---

漢字

引数	5, 47, 48, 49
回路の名前	27, 28
外部ポート	27, 28, 38
概略見積もり	3, 20
概略見積もりエンジン選択	40
拡張子	5, 7, 27
割込み	37, 38, 44, 46
割込み(例外)	37, 38, 44, 46
割込み(例外)処理	46
割込み(例外)定義	37, 38
機能検証	2, 49
見積もりエンジン	40, 41
現バージョン	11
実行ステージ	48
実行環境	5

出力[out]	28
書き込みモード指定バス	28
消費電力	10, 11, 20, 40
信号線	27
信号属性	21
信号方向	21, 28
静止時消費電力	14
設計の目標値	11
設計プロジェクト名	14
設計目標値	11
双方向[inout]	28
属性	12, 13, 15, 19, 26, 27, 28, 35
遅延時間	11, 14, 40
遅延分岐	13, 15
遅延分岐スロット数	13, 15
抽象度	49
動作検証	49
入出力ポート	26
入力[in]	28
発生条件	38
命令コード	3
命令セット	30
命令タイプ	29, 30, 32, 33, 36, 37
命令メモリ	19
命令メモリアクセスユニット	27
命令レジスタ	19
命令形式	3, 29, 30
命令名	36, 37, 44, 45
面積	10, 11, 14, 20, 40
例外	46
論理合成	2, 51
論理合成モデル	2, 49, 50, 51, 52
論理式	38