ASIPSolutions

ASIP Meister チュートリアル

2008 年 10 月

バージョン 2.3

エイシップ・ソリューションズ株式会社

〒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

ASIPSolutions

目次

図表目次

1. 表示上の約束

本マニュアルでの表記上の約束事を説明します。

1. 文中、[]で囲まれた名称はメニュー、ボタン、テキストボックス、チェックボックスなどの名称を表しています。
2. $ はコマンドプロンプトを表しています。
3. 文中、""で囲まれた名称はファイル、ディレクトリなどの名称を表しています。

2. ASIP Meister をスタートする前に（準備）

2.1. インストールとセットアップ

2.1.1. ASIP Meister のセットアップファイルの作成

ASIP Meister と java2 の実行ファイルのパスと ASIP Meister の環境変数を設定したファイル"ASIPmeister.setup"を作成します。インストールガイドに従ってインストールされている場合には、"ASIPmeister.setup"を次のように記述します。

記述例:

```
PATH=/usr/java/jre1.6.0_05/bin/:$PATH
PATH=/usr/local/ASIPmeister/bin/:$PATH
ASIP_APDEV_SRCROOT=/home/xxx/ASIPS_APs
ASIPmeister_HOME=/usr/local/ASIPmeister
export PATH
export ASIP_APDEV_SRCROOT
export ASIPmeister_HOME
```

2.2. チュートリアル設計例題

このチュートリアルでは"small_RISC"と呼ばれる小規模のプロセッサを、仕様書に従って設計していきます。仕様書が巻末にありますので参照しながら作業をすすめてください。

3. ASIP Meister での共通動作

この節ではASIP Meisterが生成するディレクトリやファイルについて簡単に説明しながら、ASIP Meister の起動に必要な設定と、ASIP Meister の起動、保存、終了、再開に関する操作をしてもらいます。

本チュートリアルでは設計データを"tutorial"というディレクトリに保存します。あらかじめディレクトリを作成しておいてください。

なお、この節で行う作業はファイルなどの操作に慣れてもらうために行います。”small_RISC”の設計に直接必要な作業ではありませんので、理解されている方は飛ばしていただいて結構です。

3.1. 起動

3.1.1. パスの設定

まず、ASIP Meister を適切に起動するために、開いているターミナルで環境変数を設定し、java2 と ASIP Meister へのパスを有効にします。”ASIPmeister.setup”ファイルのあるディレクトリで次のコマンドを実行してください。

```
$ source ASIPmeister.setup
```

これで、ASIP Meister を起動することができます。

新しいターミナルで ASIP Meister を起動するには、ターミナルの起動ごとにこの操作をする必要があります。ターミナル起動時にこの操作を自動的に実行するよう設定することをお勧めします。

3.1.2. ASIP Meister 起動

ASIP Meister を起動するには、ターミナル上でコマンドを入力する必要があります。新規に設計をするか、設計を更新するかで入力するコマンドが異なります。

3.1.3. 新規設計スタートの場合

新規に設計を行う場合は次のようにして ASIP Meister を起動します。この場合、はじめからプロセッサを設計するメインウインドウが開きます。

```
$ ASIPmeister &
```

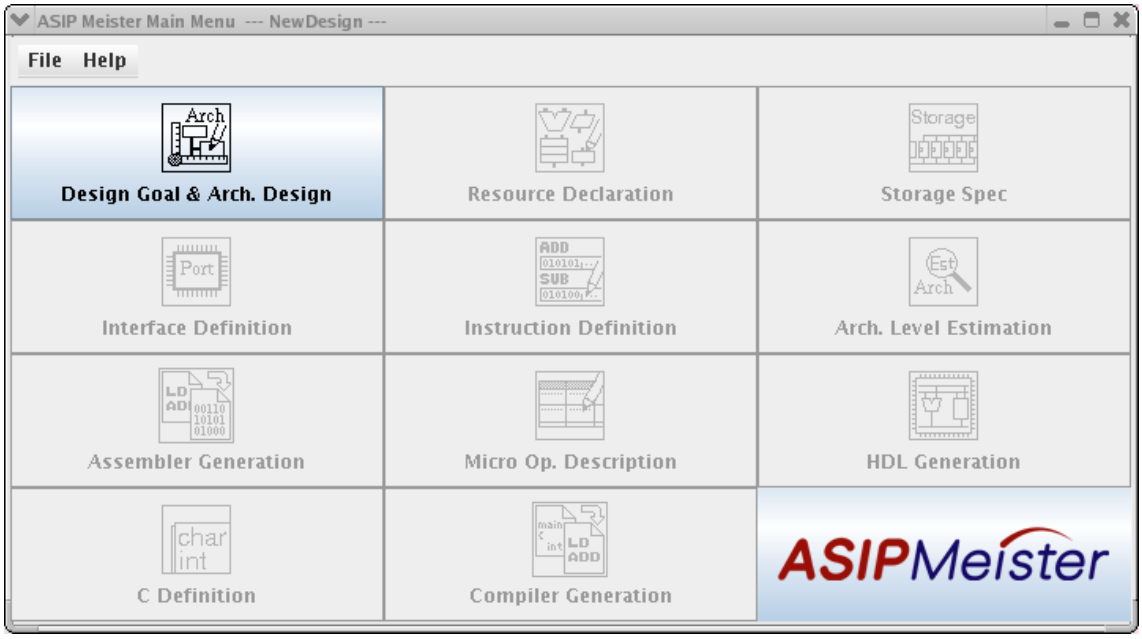

図 1 ASIP Meister の新規設計画面

3.1.4. 設計更新の場合

既存の設計データを更新する場合は新規設計と同様に ASIP Meister を起動し、起動後に[File]メニューの[Open Design]から更新する設計データを選ぶことになります。

```
$ ASIPmeister &
```

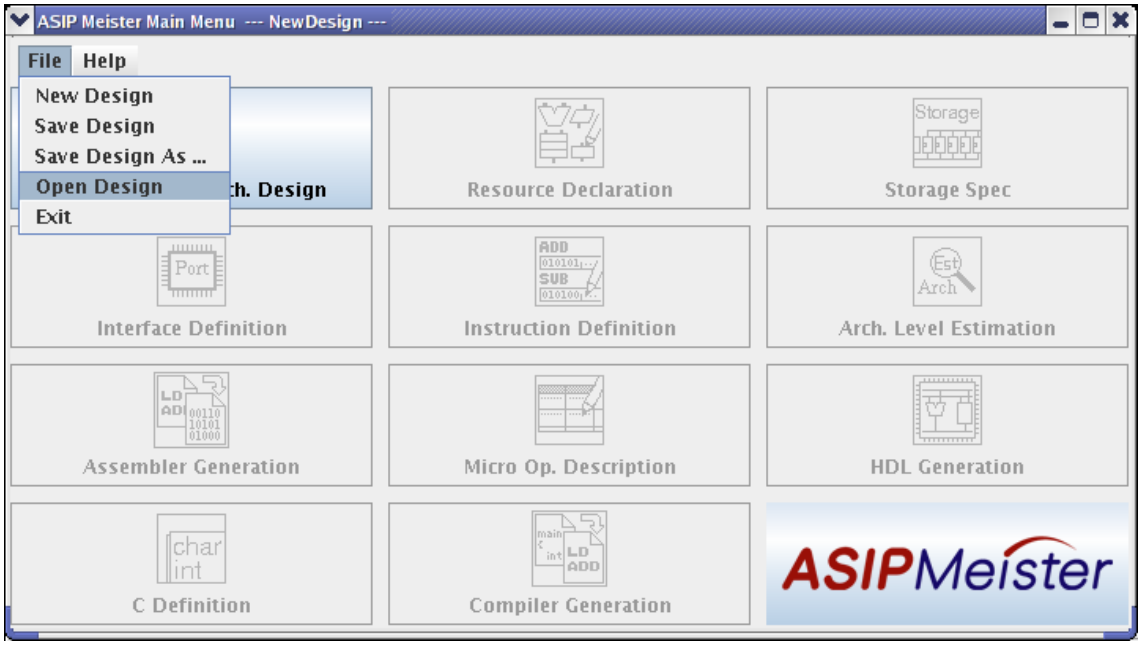

図 2 ASIP Meister の File メニュー

また、起動時に引数として、更新する設計データを指定することができます。この場合、設計途中のプロセッサを更新するメインウインドウが開き、設計データを更新することができます。図 3 ASIP Meister の設計途中の画面は Storage Spec のステップまで設計が進んだ、プロセッサの設計データを開いたところです。

```
$ ASIPmeister データ名.pdb &
```

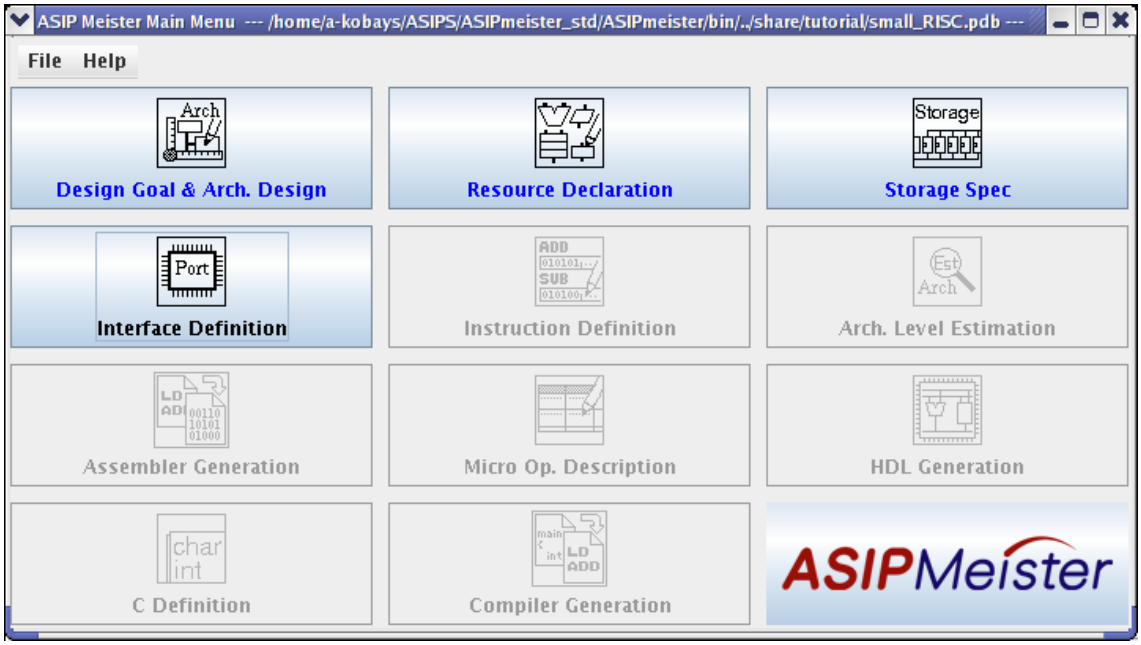

図 3 ASIP Meister の設計途中の画面

3.2. ワーキングディレクトリ”meister”と設計データ”.pdb”

3.2.1. meister ディレクトリ

ASIP Meister はカレントディレクトリに”meister”というワーキングディレクトリを作成します。meister ディレクトリには ASIP Meister の作業用のファイル、または生成された HDL 記述ファイルが格納されます。

3.2.2. 設計データ

ASIP Meister では入力データを保存するときにファイル名を指定します。その指定されたファイル名に拡張子”.pdb”がついたファイルが ASIP Meister の設計データになります。

3.2.3. 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。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。よく使用される組み合わせのメモリ構成についてはポートの定義を含むデフォルトの設計データを用意していますので、これらの設計データを使用することができます。

3.3. ASIP Meister の終了、設計データの保存

3.3.1. 新規設計データを保存する場合、またはデータ名を変更して保存する場合

メインウインドウで[File]メニューから[Save Design]または[Save Design As …]を選択するとデータ名入力ウインドウが開きます。

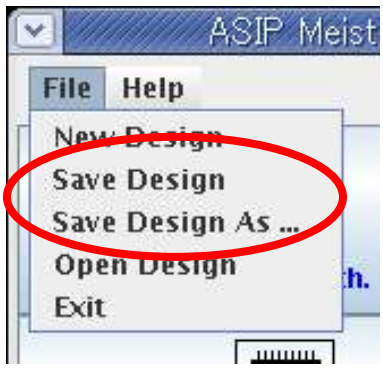

図 4 [File]メニューの保存項目

データ名入力ウインドウで、[Save In]の▼プルダウンメニューから、".pdb"データを保存する先のディレクトリ名を選択します。チュートリアルでは"tutorial"ディレクトリを選択します。

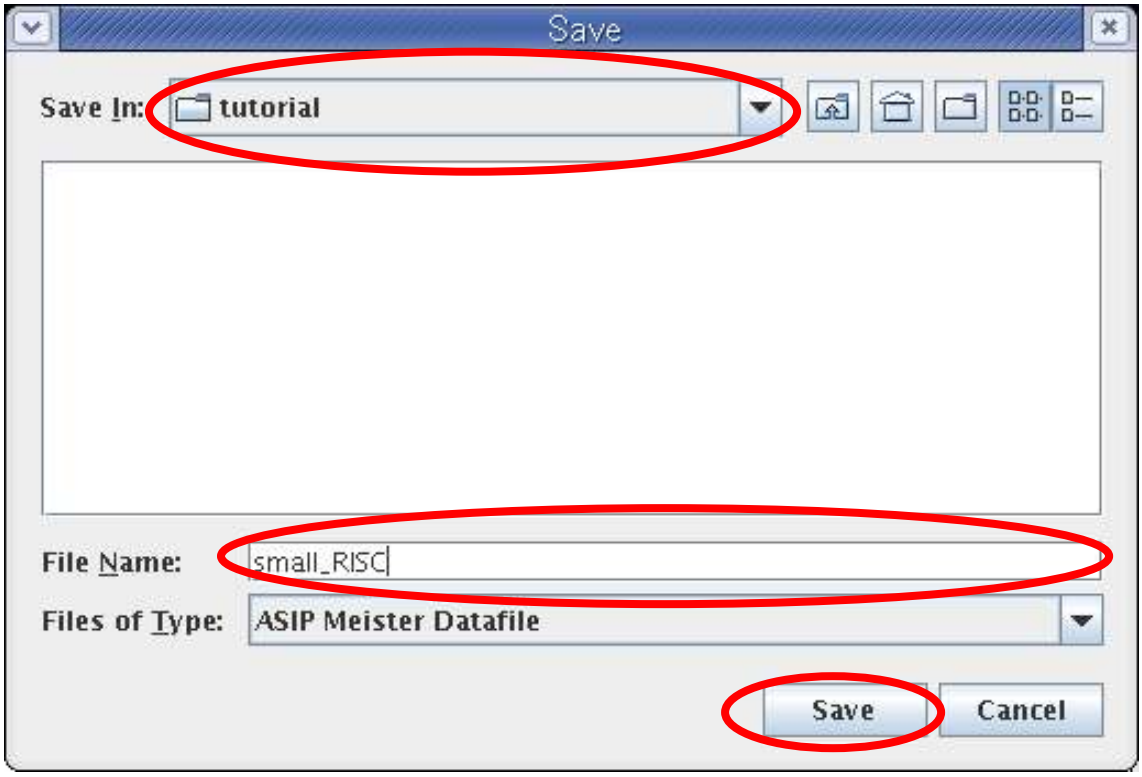

図 5 保存ダイアログボックス

次に、[File name]に設計データ名を入力します。この名前に拡張子".pdb"がついて、"small_RISC.pdb"というファイル名で設計データが保存されます。最後に、[Save]をクリックしてデータを保存し、終了します。

メモ:ASIP Meister を起動するとき、コマンドライン引数に設計データ名を指定する場合、ここでつけた名前を指定することになります。

$ ASIPmeister small_RISC.pdb &

3.3.2. 設計更新の場合（設計データ名を変更しない場合）

[File]メニューから[Save Design]を選択します。そうすると、現在の設計データが保存できます。

3.4. サブウインドウを閉じてメインウインドウへ戻る

ASIP Meister では設計するプロセッサの情報をサブウインドウごとに入力していきます。サブウインドウで必要な項目を全て設定し終えたら、[Complete]ボックスをチェックし、[File]から[Close]を選択してメインウインドウに戻ります。未入力の項目があると[Complete]チェックボックスはチェックできません。

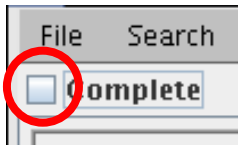

図 6 complete チェックボックス

メモ：[Complete]ボックスと ASIP Meister の推奨の作業フロー

[Complete]ボックスをチェックすると、サブウインドウでの設定が完了となり、次のサブウインドウの設定作業に進むことができます。一方、[Complete]チェックボックスをチェックしないでサブウインドウを閉じた場合、次のサブウインドウの作業に進むことはできません。

4. ASIP Meister での設計作業の流れ

ASIP Meister では、サブウインドウを順番に開き、必要な情報を入力していきます。次に、ASIP Meister での設計の典型的な作業順序を示します。

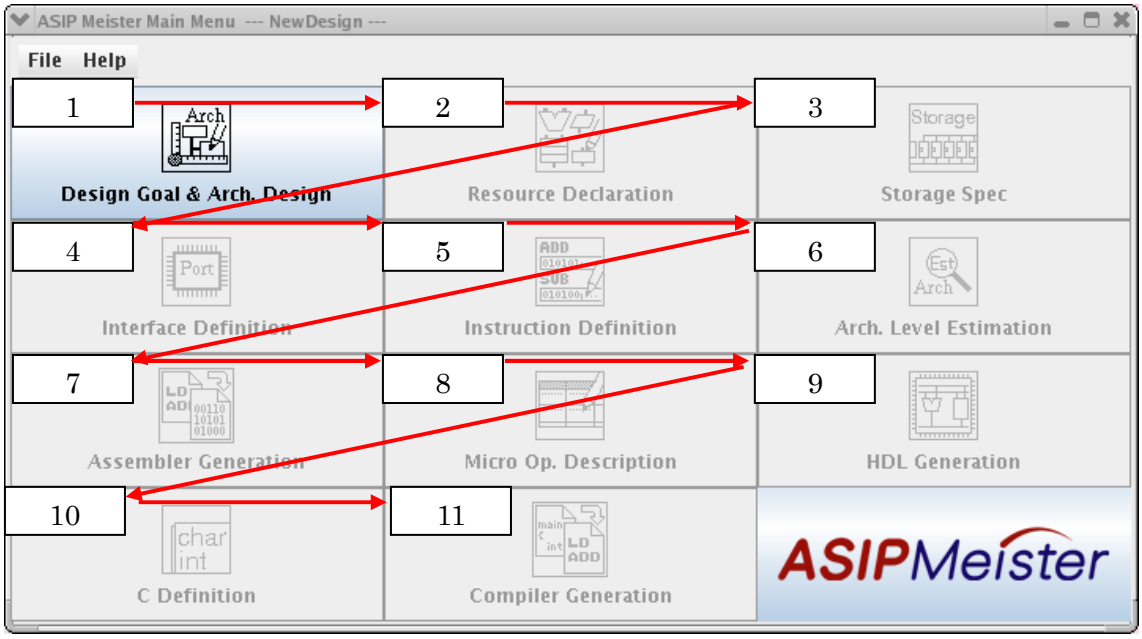

図 7 ASIP Meister の設計順序

以下、この順番に従って small_RISC を設計していくことにします。

1. Design Goal & Arch. Design　　設計目標値とパイプラインステージの設定
2. Resource Declaration　　リソース宣言
3. Storage Spec　　ストレージ定義
4. Interface Definition　　入出力ポートの定義
5. Instruction Definition　　命令形式・命令コードの定義
6. Arch. Level Estimation　　概略見積もり
7. Assembler Generation　　アセンブラ記述の生成
8. Micro Op. Description　　マイクロ動作記述
9. HDL Generation　　HDL 生成
10. C Definition　　C 記述の定義
11. Compiler Generation　　コンパイラ, Binutils の生成

<注意> [10. C Definition]と[11. Compiler Generation]に関しては、ASIP Meister Standard で提供されているベースプロセッサ ***Brownie*** にのみ有効です。***Brownie*** を拡張したプロセッサ以外では、コンパイラ生成できません。

4.1. メモリモデル定義ファイルの読み込み

はじめにメモリモデル定義ファイルを読み込みます。今回作成する small_RISC は、命令メモリがシングルサイクルアクセス、データメモリがマルチサイクルアクセスとなります。このメモリモデルに対応するメモリモデル定義ファイルは

“basefile/InstSingle-DataMulti.pdb”

となります。設計に先立ってこのファイルを読み込むことにより、後の Interface Definition での作業を簡約化することができます。

メモリモデル定義ファイルの開き方は、通常の設計ファイルを開く方法と同じで、コマンドラインの引数に与えるか、メニューから指定することができます。

4.2. 設計目標値とアーキテクチャ・パラメータの設定: Design Goal & Arch. Design

このステップでは、プロセッサタイプやアーキテクチャ・パラメータを設定し、プロセッサの大枠を設定します。

まず、ASIP Meister のメインウインドウ内で[Design Goal & Arch. Design]をクリックします。

図 8 Design Goal & Arch. Design ボタン

[Design Goal & Arch. Design] サブウインドウが開きました。

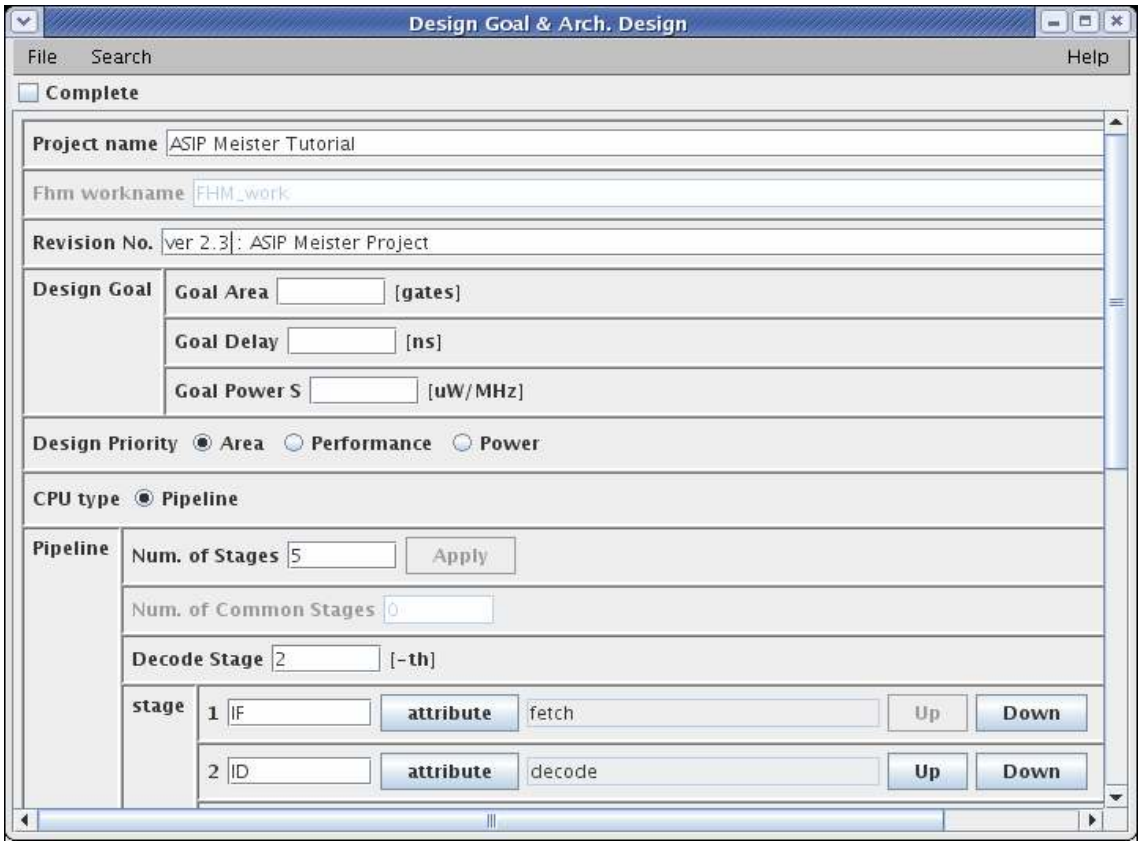

図 9 [Design Goal & Arch. Design]サブウインドウ

4.2.1. 設計データの管理情報

では実際にデータを入力していきます。データを入力するにはテキストボックスをクリックし、カーソルが有効になった状態でキーボードから入力をしていきます。

初めに、設計データの管理情報を入力します。下図のように入力してください。

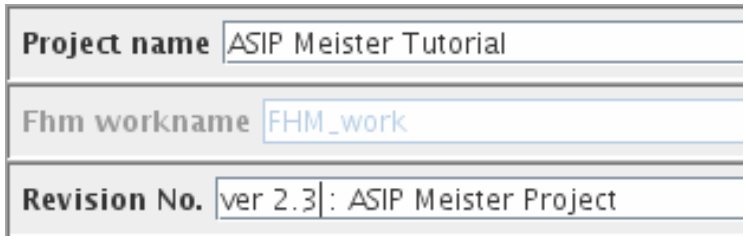

図 10 設計管理情報の入力

Project name	設計プロジェクト名を入力します。任意の文字列が入力できますが、ここでは”ASIP Meister Tutorial”と入力します。
Fhm workname	Fhm workname の設定ができます。現バージョンでは無効で、何も入力することはできません。
Revision No.	設計のバージョン管理に用いる任意の文字列を入力することが出来ま

	す。バージョン管理はユーザーに委ねられていますが、今回は”ver 2.3：ASIP Meister Tutorial”とします

4.2.2. 設計の目標値

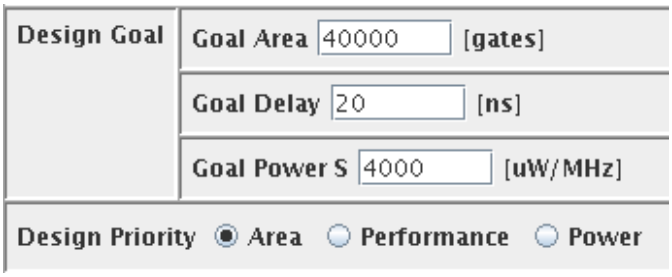

図 11 設計目標の入力

次に設計するプロセッサの規模の目標値を設定します。

Design Goal	Goal Area	目標面積です（単位：[gates]）。40000[gate]とします
	Goal Delay	組み合わせ回路の最大パス遅延時間です(単位：[ns])。20[ns]とします。
	Goal Power S	静止時消費電力です（単位：[μW/MHz]）。4000[μW/MHz]とします。
Design Priority	回路生成で優先処理される設計目標を面積、最大遅延時間、静止時消費電力（Area/Delay/Power）から一つだけ選択します。今回は Area を選択します。	

当然ですが、設計では各項目がこの値以下になることを目標とします。

4.2.3. アーキテクチャ・パラメータ

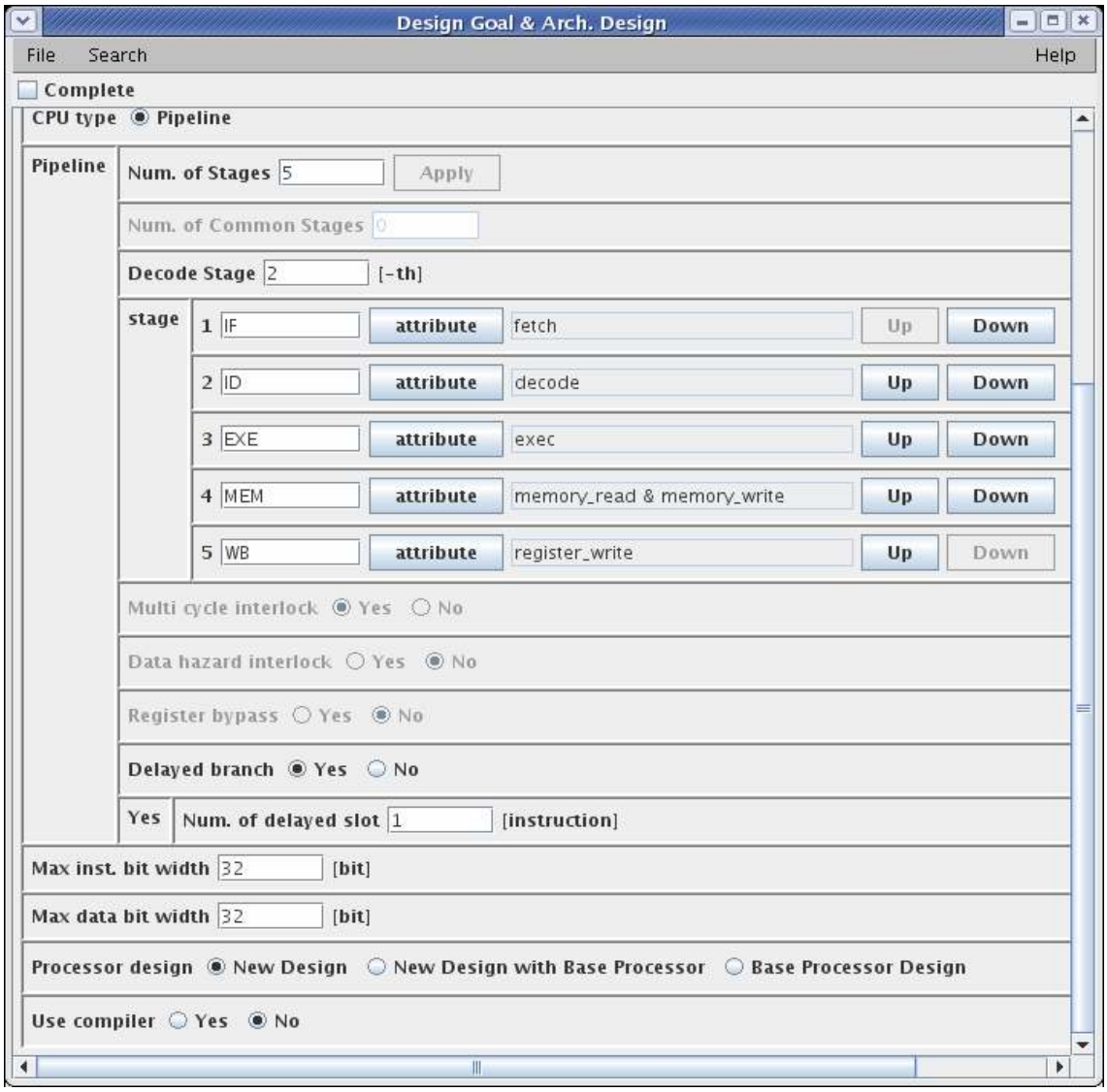

図 12 アーキテクチャ・パラメータの入力

設計するプロセッサの種類やパイプラインの段数などを入力します。設定する項目が少し多いので、数回に分けて説明していきます。[CPU Type]は現バージョンでは[pipeline]しか選択肢がありませんのでこれを選択します。

CPU Type	CPU の種類を設定します。現バージョンでは”pipeline”のみ設定可能です。
Pipeline	パイプラインの各ステージの情報を設定します。（後述）
Max inst. Bit width	命令の最大ビット幅を設定します。（後述）
Max data bit width	データの最大ビット幅を設定します。（後述）
Processor design	プロセッサの設計方針を”New Design”, “New Design with Base

	Processor”, “Base Processor Design”から選択します。今回は”New Design”を選択します。
Use compiler	コンパイラを作成するかを選択します。今回は”No”を選択します。

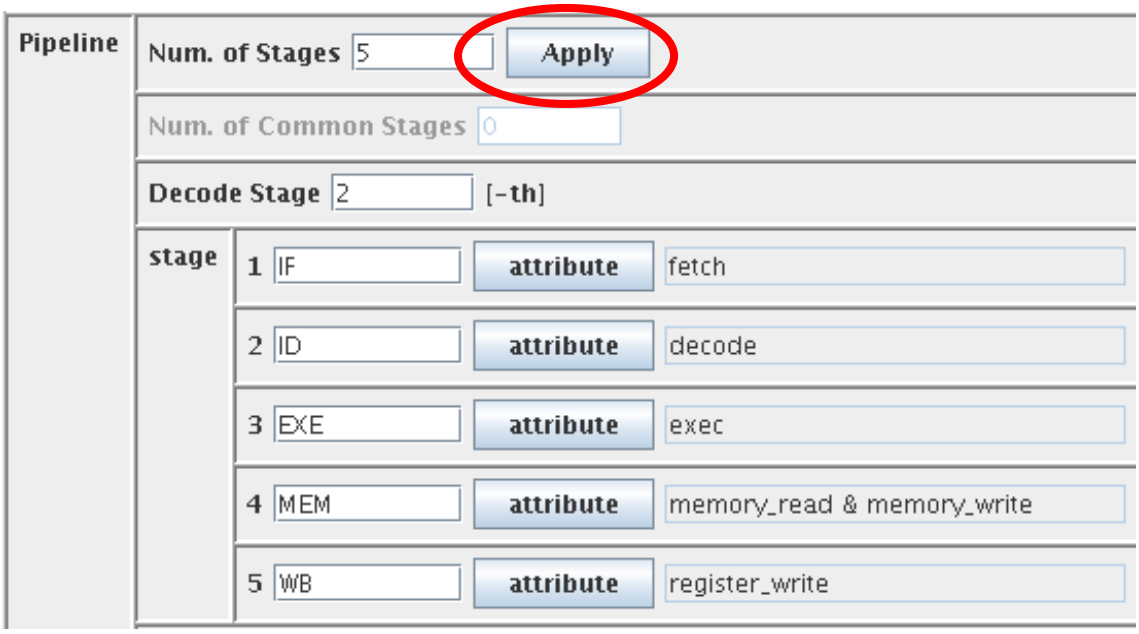

図 13 パイプラインステージの設定

次にパイプラインの情報を入力していきます。入力する情報は次の表のとおりとなっています。

最初に[Num. of Stages]を入力して、その後、[Apply]ボタンを押してから他の値を入力するようにしてください。この順序は守るよう注意してください。

Num. of Stages	パイプラインの段数を設定します。ここでは 5 とします。入力後[Apply]ボタンを押してください。
Num. of Common Stages	共通の動作をするステージ数を設定します。現バージョンでは設定することができません。
Decode Stage	デコード処理をするステージを設定します。
stage	各ステージの動作を設定します。

まずパイプラインの段数を上記のように設定して[Apply]ボタンを押します。[Stage]行に 5 ステージ分のパラメータ入力行が現れます。

次にパイプラインの各ステージについて設定します。このチュートリアルではデフォルトのステージ名を使用しますが、設定の方法も説明しておきます。ステージ名は入力フィ

ールドに入力し、ステージ属性は[Attribute]ボタンをクリックしてそこで現れるウインドウ内で設定します。

次ではパイプラインの特性を設定することができますが、現バージョンでは遅延分岐に関してのみ設定することができます。

Multi cycle interlock	現バージョンでは設定できません。
Data hazard interlock	現バージョンでは設定できません。
Register bypass	現バージョンでは設定できません。
Delayed branch	遅延分岐を有効にするかどうかを指定します。
Num. of delayed slot	遅延分岐が有効である場合、遅延スロット数を設定します。

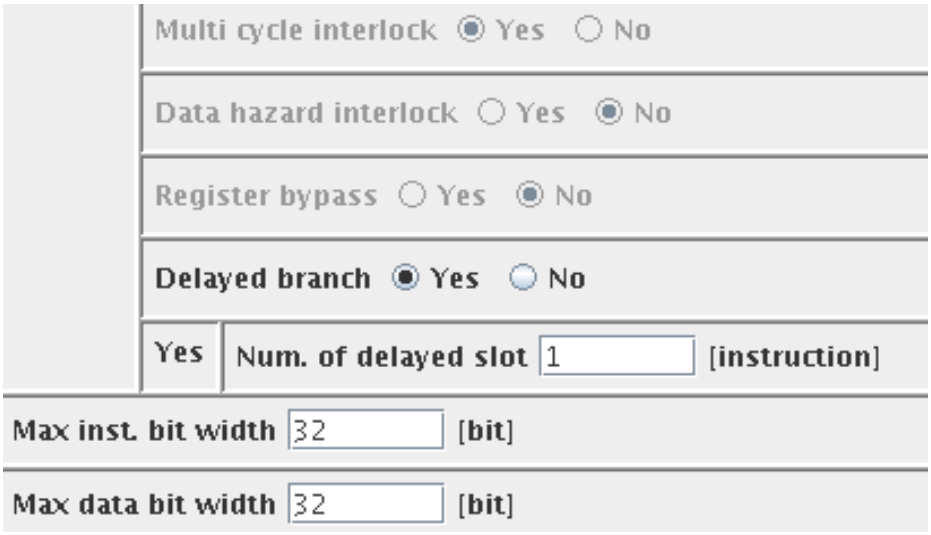

図 14 遅延分岐とビット幅の設定

遅延分岐とデータ幅の設定をします。[Delayed branch]を[Yes]に設定してください。遅延分岐スロット数を設定する[Num. of delayed slot]バーが現れたと思います。この値を 1 とします。

その下で最大命令ビット幅と最大データ幅が指定しますが、両方とも 32 とします。

最後にプロセッサの設計方針とコンパイラの使用を指定しました。前述したとおり、それぞれ”New Design”, “No”を選択します。

以上で[Design Goal & Arch. Design]のパラメータが設定できました。設定を確定するために画面上部の[Complete]チェックボックスをチェックし、[File]→[Close]で設計入力画面を閉じます。

もし設定の間違いに気がついたら、[Complete]チェックボックスのチェックを外してか

ら修正し、完了したらまたチェックしなおしてから終了します。この操作は他のステップにおいても同様です。

4.3. リソース宣言: Resource Declaration

[Design Goal & Arch. Design] のステップが終了した時点でメインウインドウの[Resource Declaration]ボタンが有効になります。[Interface Definition]ボタン有効になっていますが、このチュートリアルではリソースの宣言を先に行います。

図 15 Resource Declaration ボタン

[Resource Declaration]のステップでは設計で使用するリソースモデルを FHM-DBMS から選択し、リソースモデルにパラメータを設定し、設計で使用するリソースを宣言します。

ASIP Meister はハードウエアモデルを FHM-DBMS という ASIP Meister のオリジナルデータベースによって提供しています。また、FHM ではハードウエアリソースの機能が function set として記述されており、後続のマイクロ動作記述 [Micro Op. description] で使用することが出来ます。それらを使用するための設定をここで行います。

はじめに、[Resource Declaration]ボタンをクリックして、[Resource Declaration]ウインドウを開いてください。

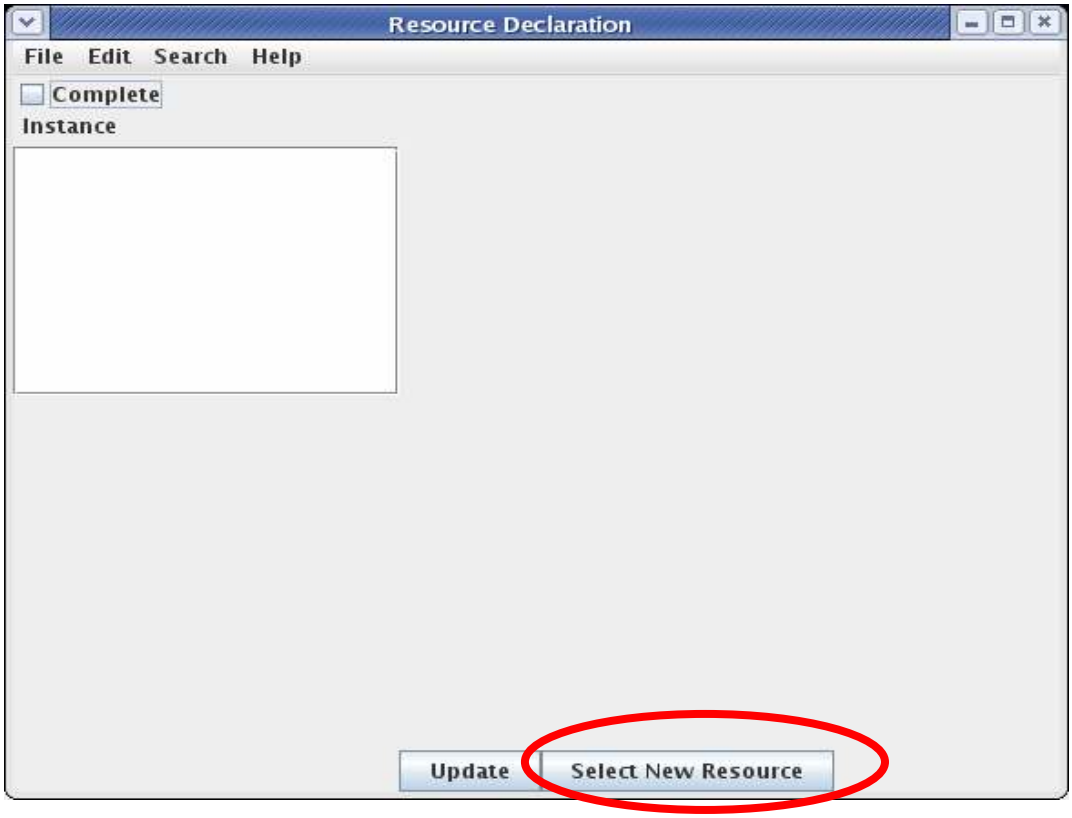

図 16 新規のリソース設計画面

上図のようなウインドウが開きます。この状態は新規の状態でリソースが何も登録されていない状態です。

次に、リソースを登録します。[Select New Resource]ボタンをクリックし、[FHM Browser]ウインドウを開いてください。

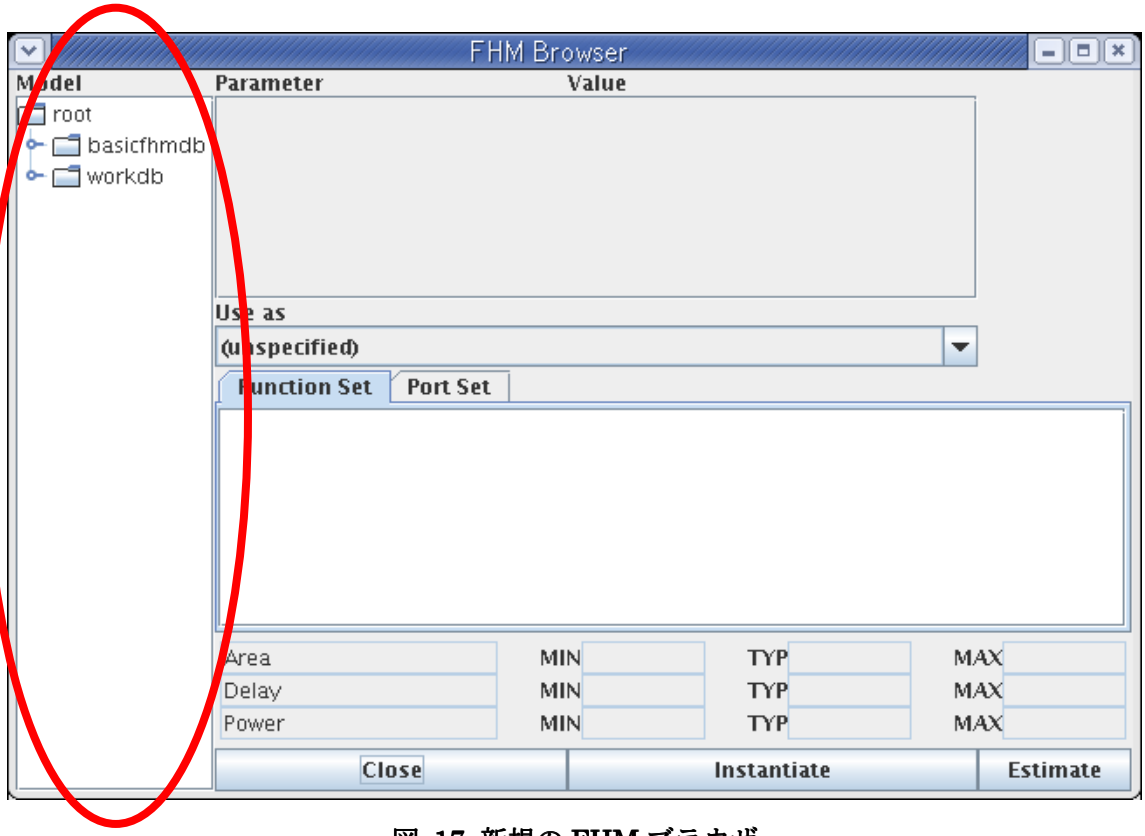

図 17 新規の FHM ブラウザ

[FHM Browser]ウインドウを開くと、左の[Model]フレームウインドウに ASIP Meister で使用することができるリソースモデルの一覧がディレクトリ形式で表示されます。一覧から使用するリソースモデルを選択すると、リソースモデルのパラメータを設定できるようになります。設定するパラメータは選択したリソースモデルによって異なります。

はじめに、[pcu]モデルを使用したプログラムカウンタを登録します。[Model]フレームウインドウから[workdb]をクリックしてディレクトリを展開します。続いて[FHM_work]ディレクトリも展開し、その中から[pcu]選択します。

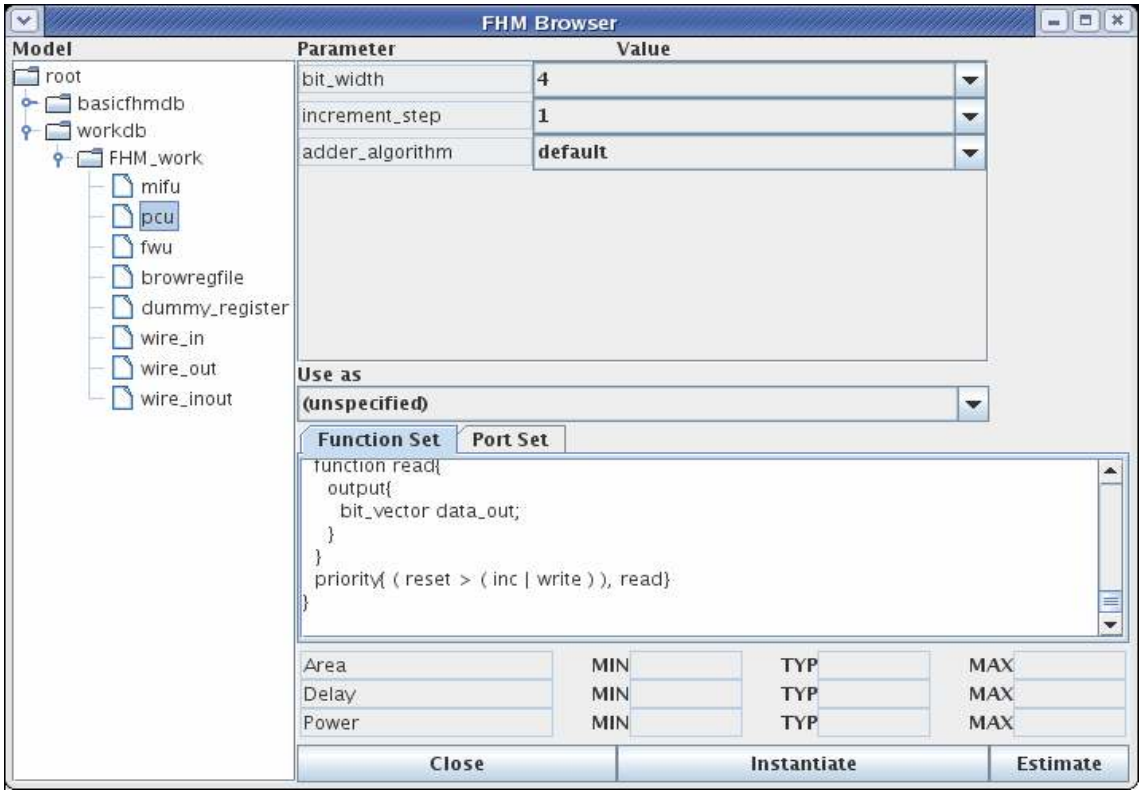

図 18 [pcu]の選択画面

[pcu]を選択すると、上図のような[pcu]のパラメータ設定画面が現れます。[pcu]のモデルで設定するパラメータは次の通りです。これらのパラメータは▼プルダウンメニューから所望のものを選択します。

bit width	カウンタのビット幅。
increment step	加算ステップ数
adder algorithm	加算アルゴリズム
Use as	リソースの使用目的

値を以下のように設定します。

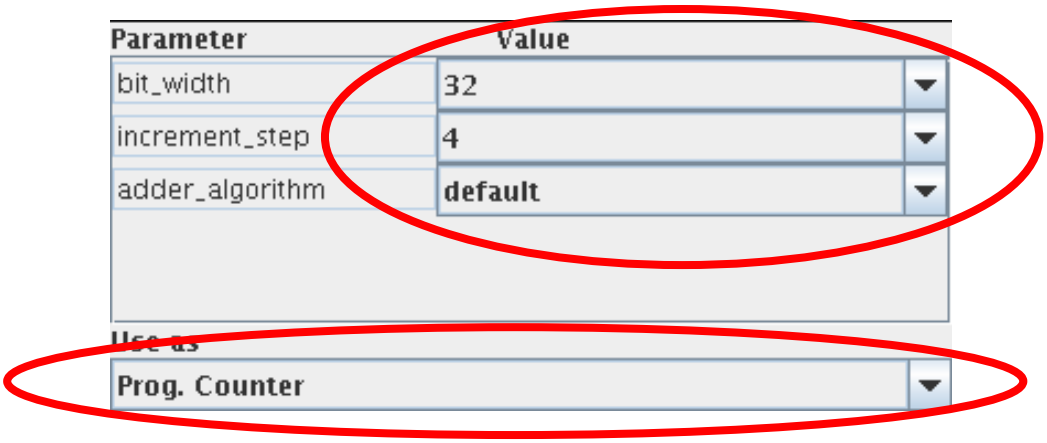

図 19 [pcu]で設定するパラメータ

[Use as]はそのリソースがプロセッサ内部で特定の役割を持つ場合に指定します。リソース[pcu]はプログラムカウンタとして使用するので、”Prog. Counter”を選択します。

次に、設定したパラメータを持つリソースの見積もり値を確認します。[Estimate]ボタンを押すと Area、Delay、Power の見積もり値が算出されます。

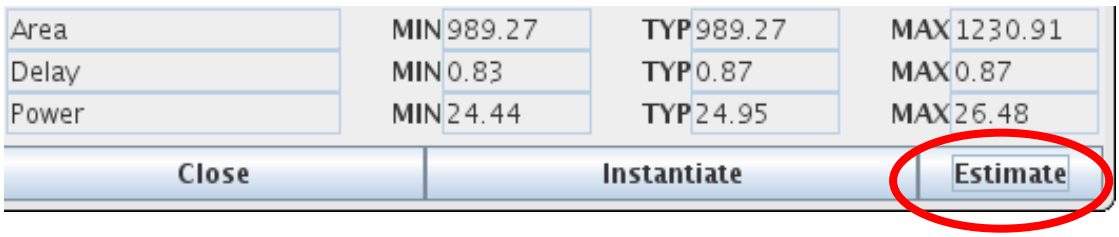

図 20 [pcu]リソースの見積もり規模

最後にリソースに名前をつけて登録します。[Instantiate]ボタンをクリックすると、リソース名を入力するウインドウが開きます。

図 21 新しいインスタンスの宣言

「small_RISC 仕様書」に従い、プログラムカウンタリソースの名前を”PC”として、[OK]ボタンを押してください。”PC”という名前でリソースが登録されます。

リソースを登録し終えたら、いったん[FHM Browser]ウインドウを閉じます。ウインドウを閉じるためには、図の[Close]ボタンを押します。

登録されたことは [Resource Declaration]ウインドウで確認することができます。ウインドウの左フレームの[Instance]に、”PC”リソースが表示されていることを確認してください。

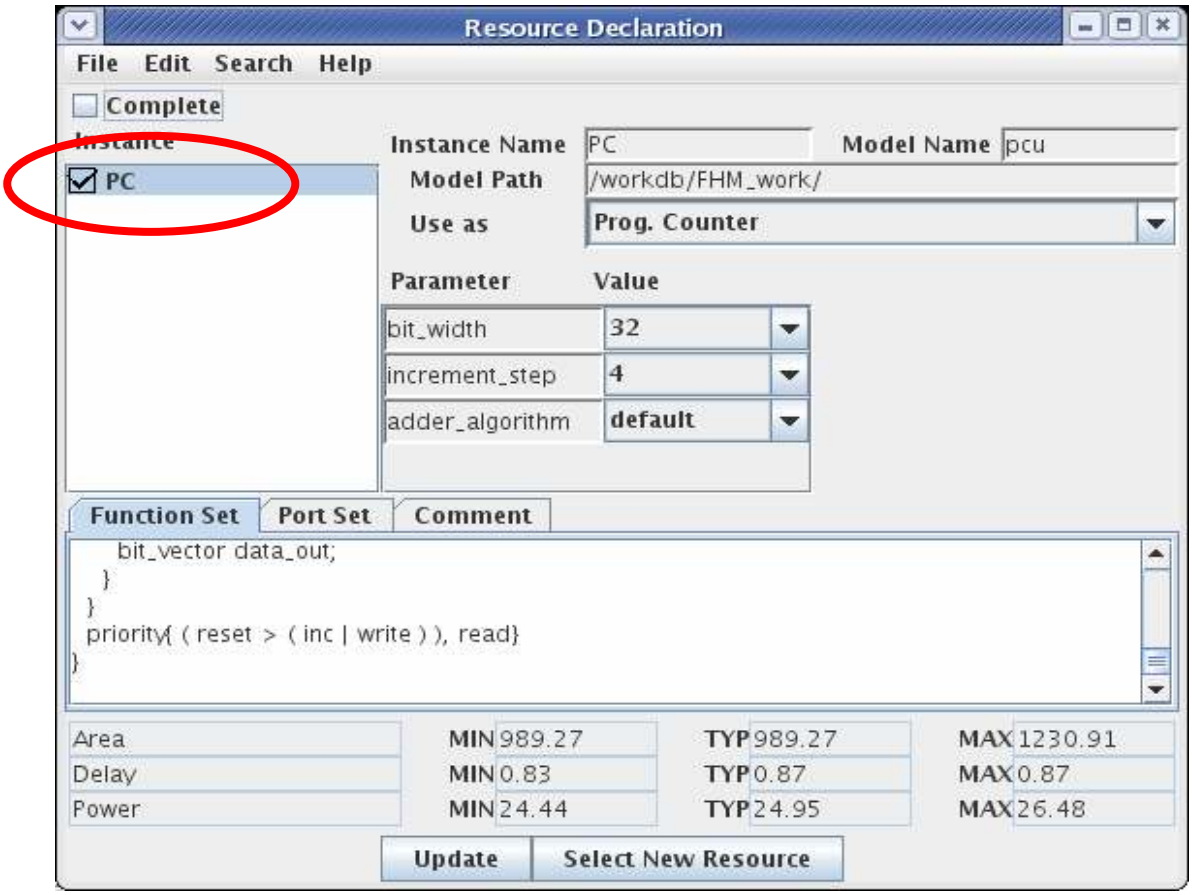

図 22 PC リソースを宣言し終えた状態

“PC”リソース宣言のための入力はこれで終了ですが、”PC”の[Function Set]と[Port Set]を確認しておきます。[Function Set]の記述は[Micro Op. description]で使用します。[Function Set]サブウインドウの記述を確認しておいてください。また、”PC”のポートは次のように定義されています。ポート名、信号方向、VHDL のデータタイプおよび信号属性を確認してください。

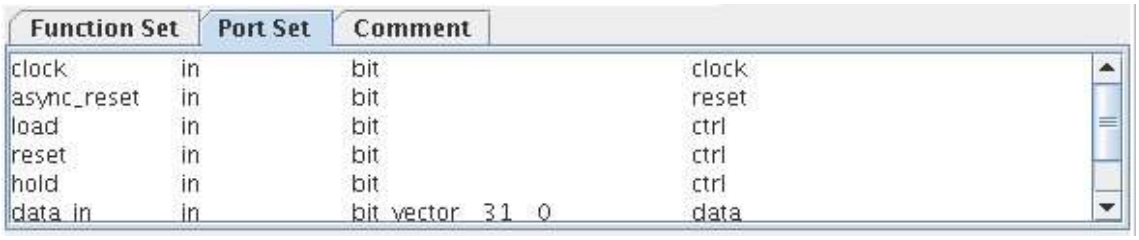

図 23 PC リソースのポート情報

以下、同様にして命令レジスタ、命令メモリ、データメモリ、レジスタファイル、ALU を宣言します。次に示す図の通り必要な設定を行ってください。

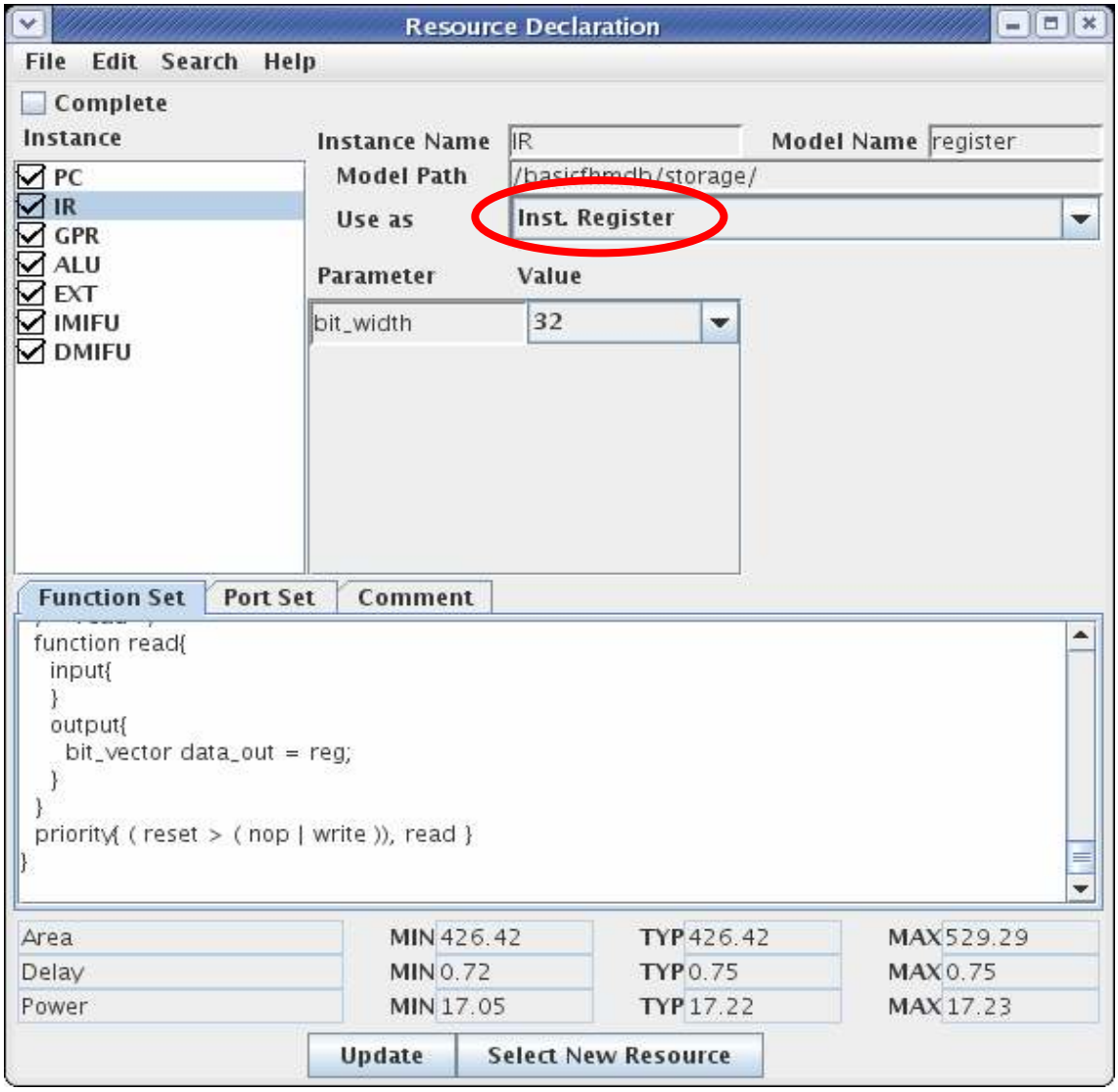

図 24 命令レジスタを宣言し終えた状態

命令レジスタは[register]という名前の FHM を使用します。ビット幅は 32 ビットに指定し、Use as は[Inst. Register]を指定します。また、名前は”IR”とします。

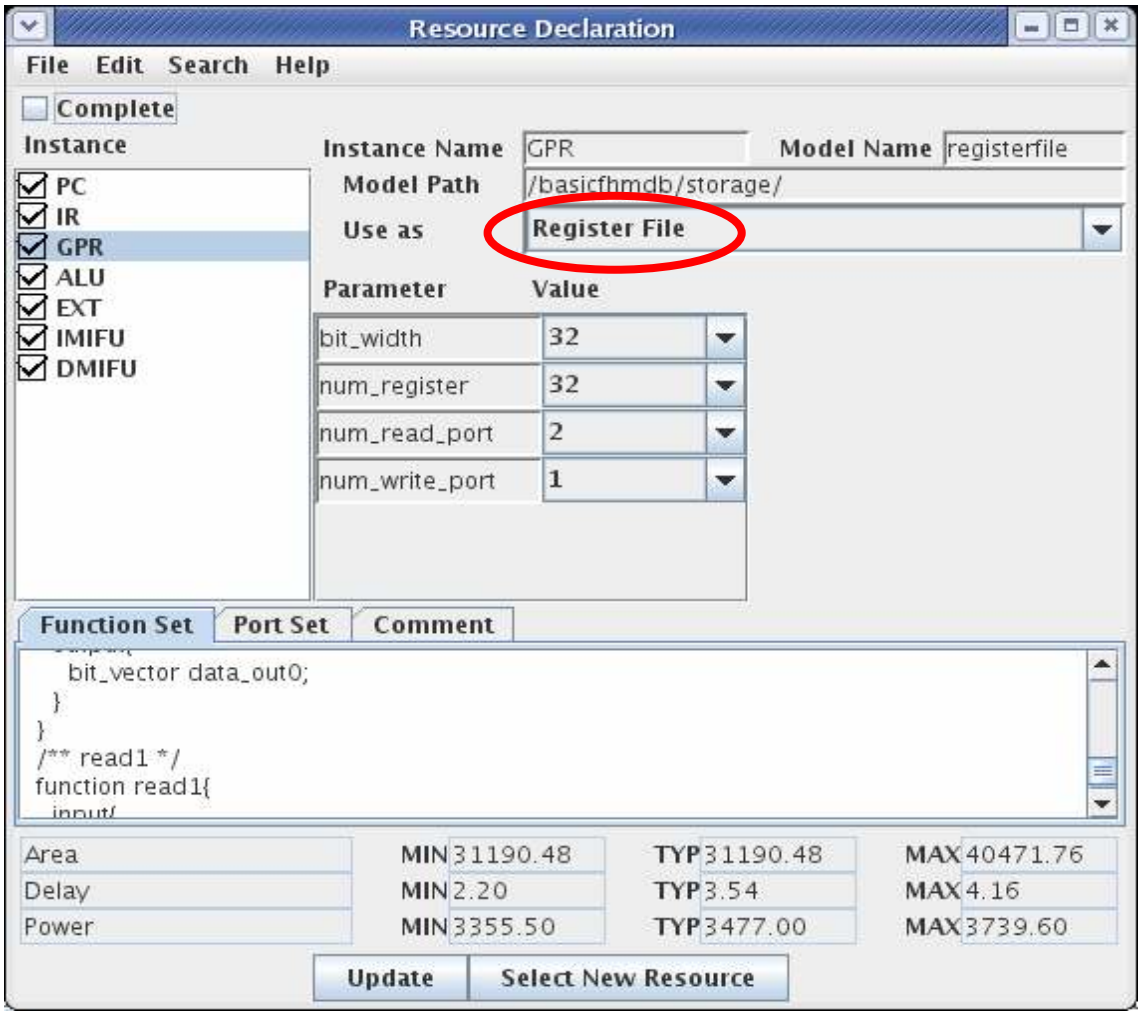

図 25 汎用レジスタファイルを宣言し終えた状態

汎用レジスタファイルは[registerfile]という名前の FHM を使用します。ビット幅、内部のレジスタの数、読み出しポート数、書き込みポート数は図 25 のように設定します。リソースの名前は”GPR”とし、Use as は[Register File]とします。

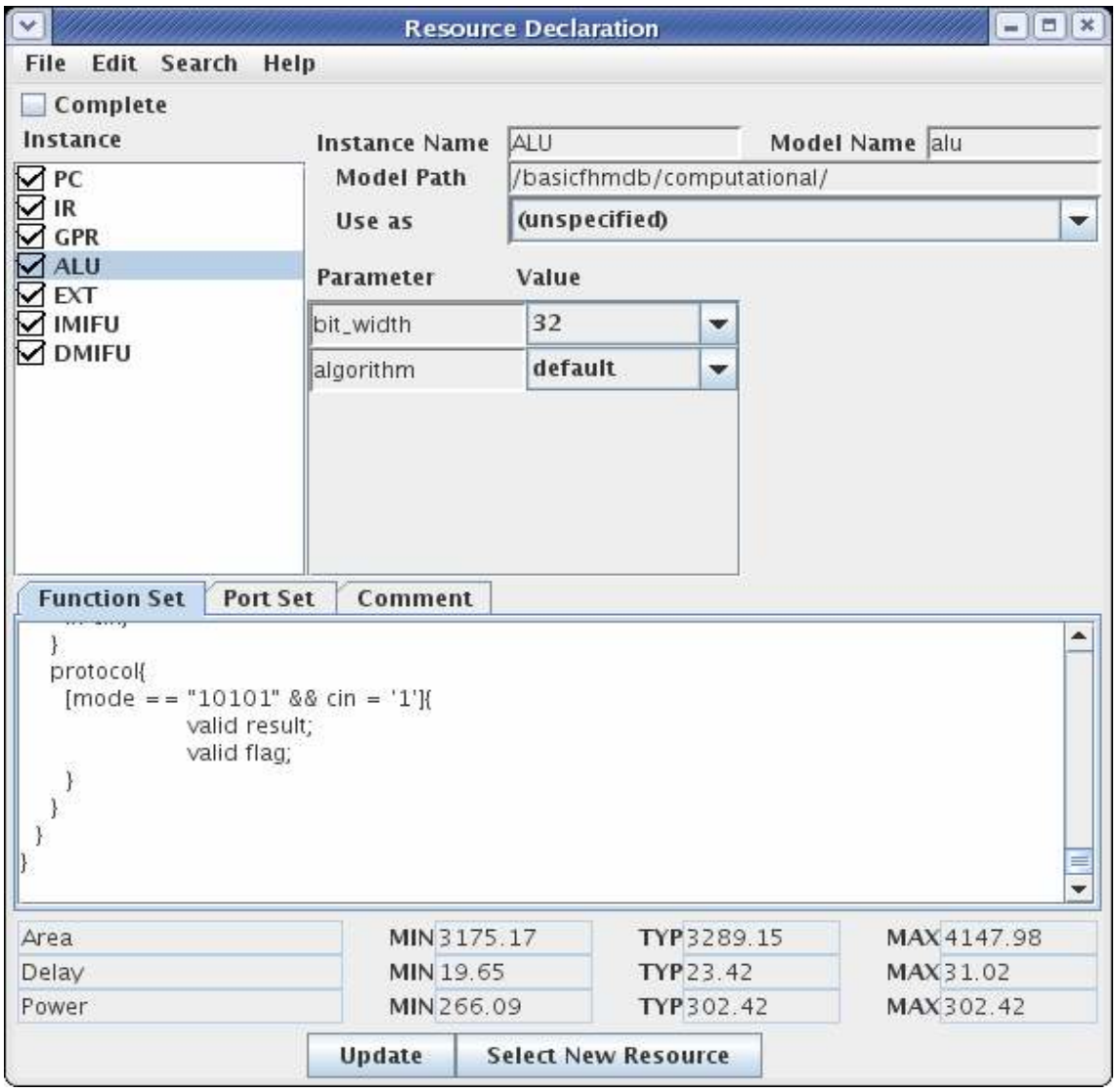

図 26 算術論理演算器を宣言し終えた状態

算術論理演算器は[alu]という名前の FHM を使用します。ビット幅と演算アルゴリズムは図 26 のように設定します。Use as はここでは[Unspecified]とします。リソースの名前は”ALU”と設定します。

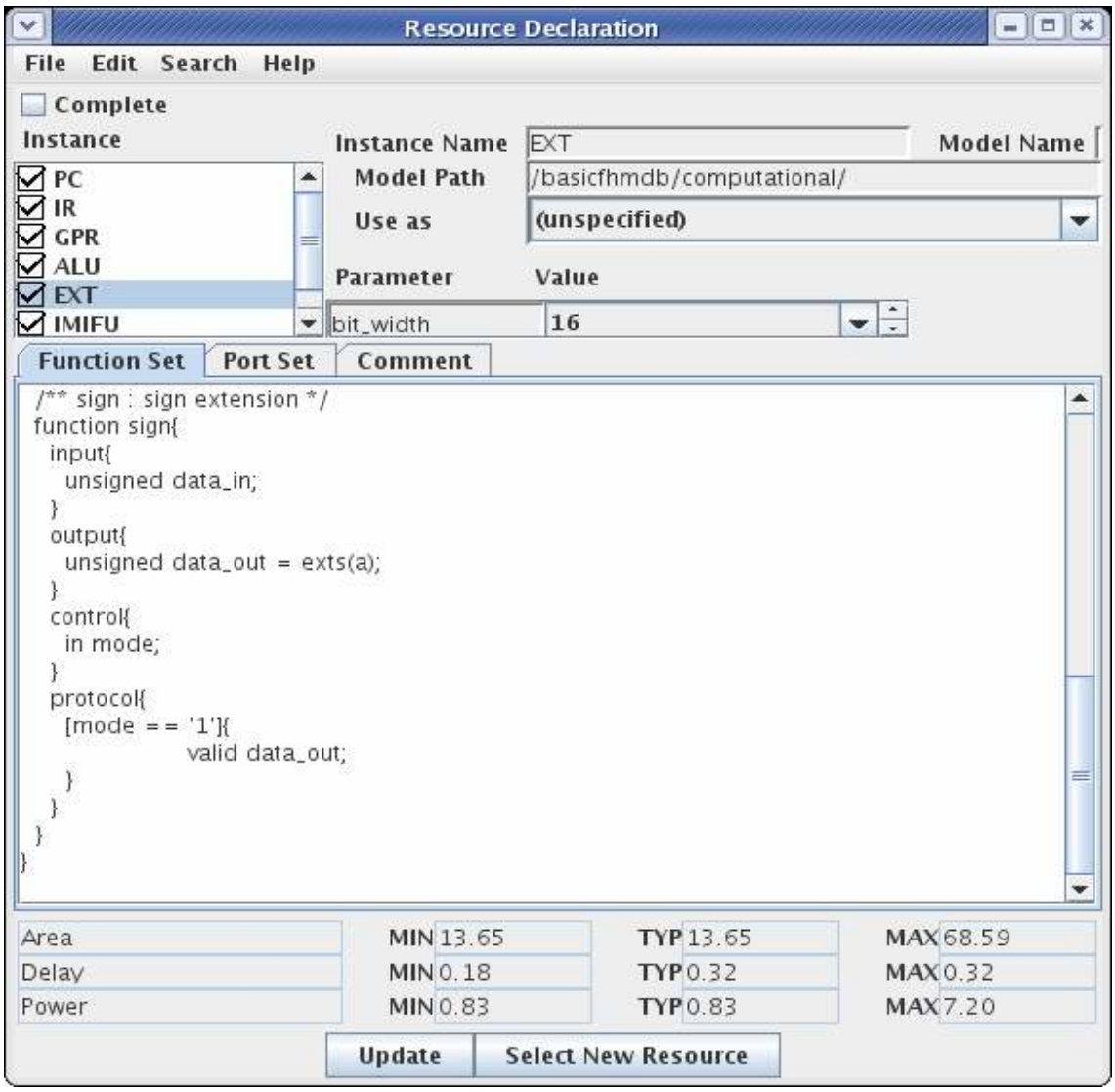

図 27 符号拡張器を宣言し終えた状態

符号拡張器には[extender]という名前の FHM を使用します。入力ビット幅は 16 ビット、出力ビット幅は 32 ビットに設定し、Use as は[Unspecified]と設定します。また、リソースの名前は”EXT”とします。

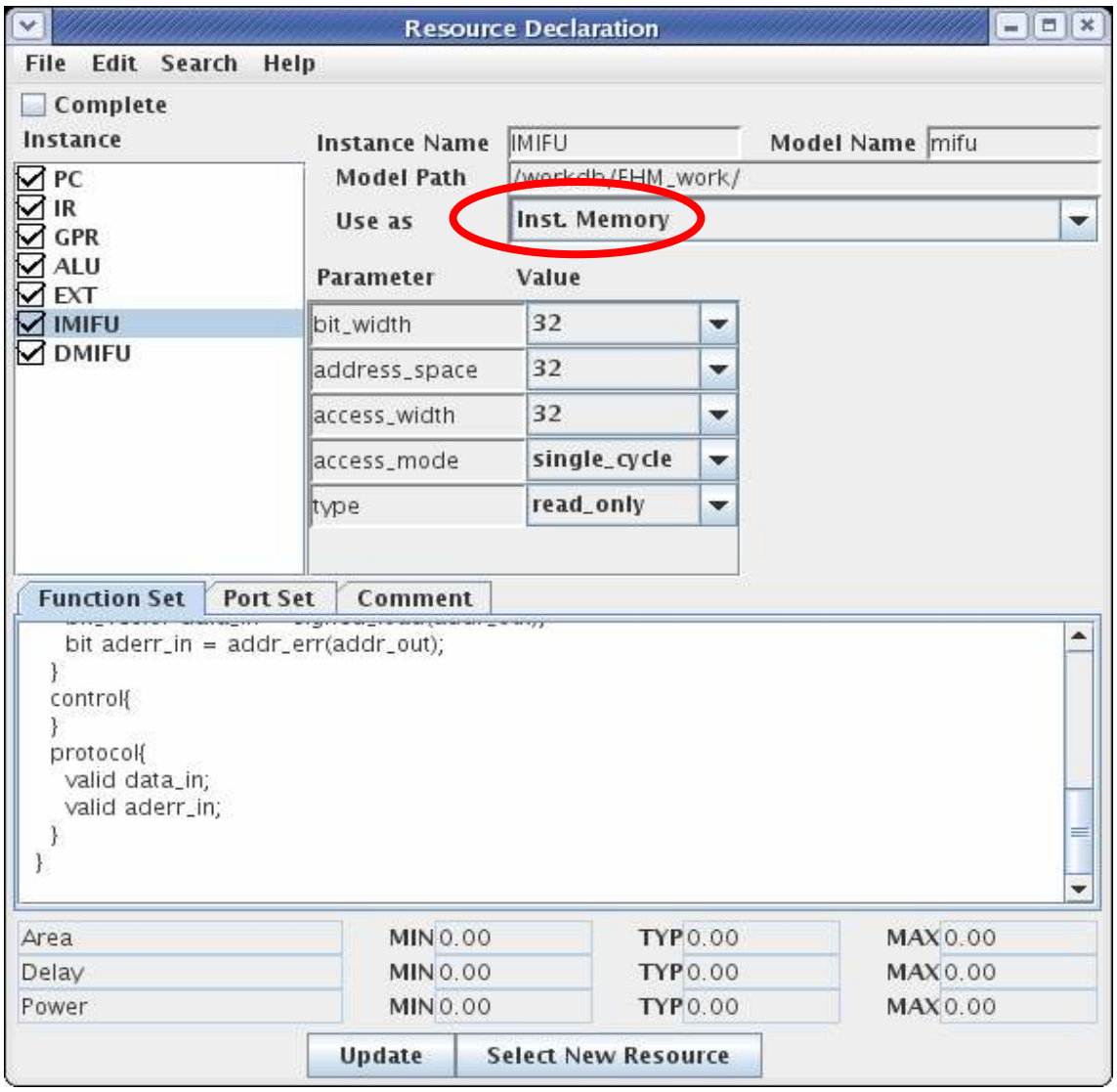

図 28 命令メモリインターフェースユニットを宣言し終えた状態

命令メモリにアクセスするインターフェースを宣言します。これには[mifu]という名前の FHM を使用します。ビット幅とアドレス空間、および、アクセス幅は図 28 のように全て 32 ビットと設定します。また、アクセスモードはシングルサイクルとし、読み込みのみを行うようにします。Use as の項目は[Inst. Memory]と設定します。また、リソースの名前は”IMIFU”とします。

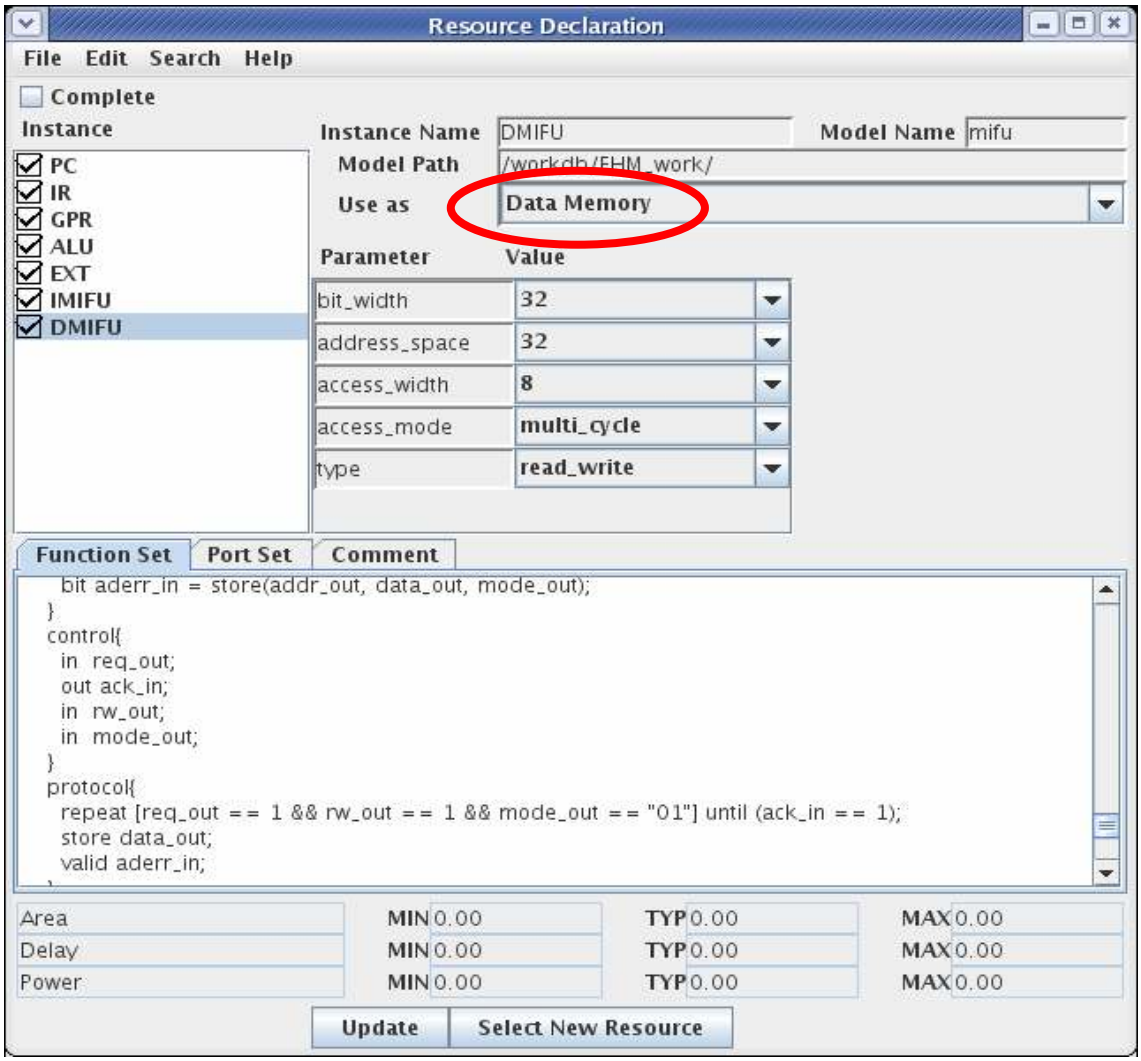

図 29 データメモリインターフェースユニットを宣言し終えた状態

データメモリにアクセスするインターフェースを宣言します。これには命令メモリへのインターフェースと同様に、[mifu]という名前の FHM を用います。ビット幅とアドレス空間は共に 32 ビットに設定し、アクセスビット幅は 8 ビットとします。また、データメモリのアクセスモードはマルチサイクルとします。最後に、データメモリは読み書き両方を行う必要があるため、タイプを”read_write”に設定します。Use as の項目は[Data Memory]を選択します。また、リソースの名前は”DMIFU”とします。

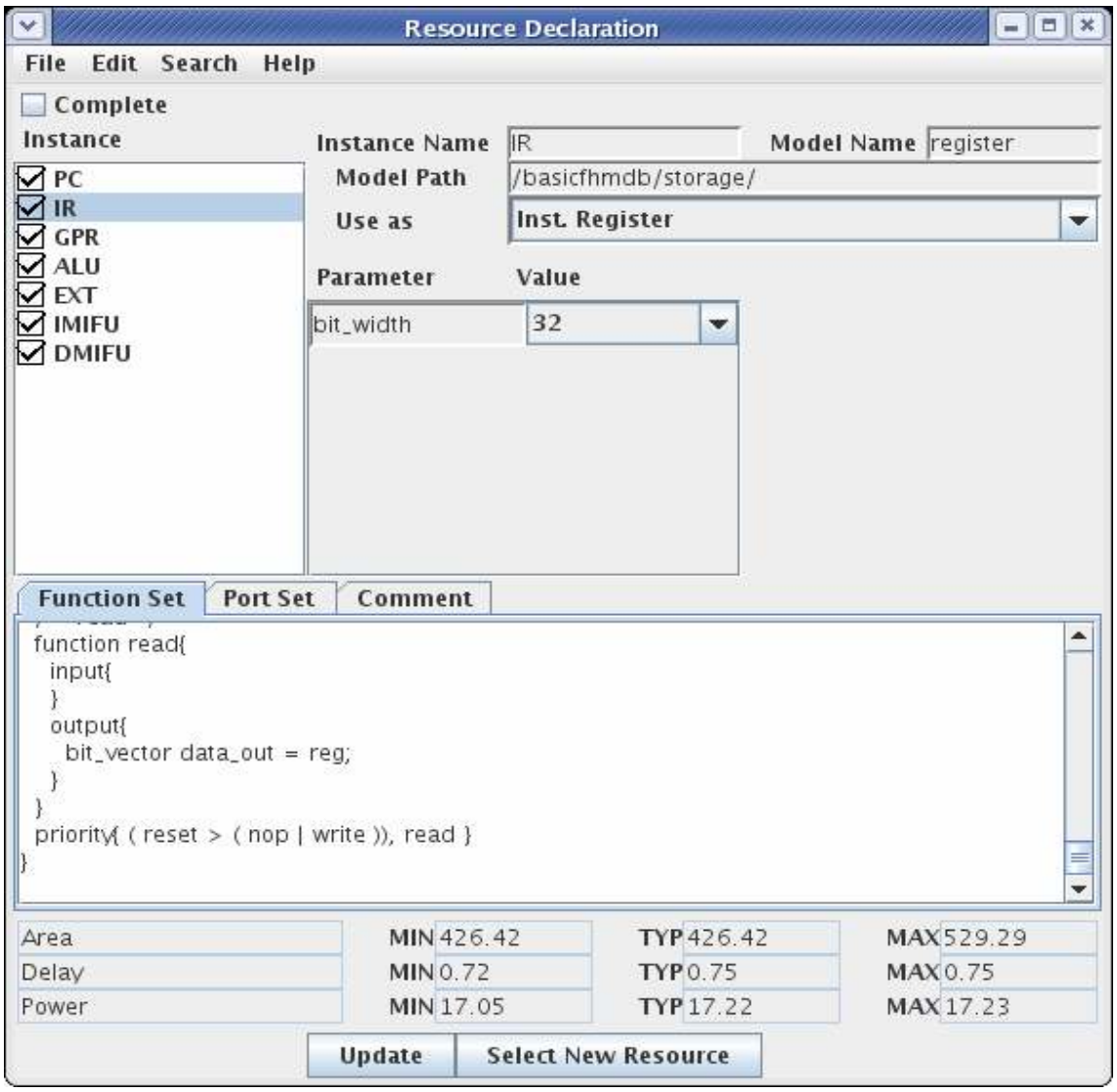

図 30 全てのリソースを宣言し終えた状態

すべて設定し終わって、上図のようになれば終了です。なお、[Instance]欄のチェックボックスの on/off は宣言されたリソースの有効・無効をコントロールします。今回は全てチェックされたままとします。

全てのリソースを登録し終えたら、[Complete]チェックボックスをクリックして、[File]→[Close]で[Resource Declaration]ウインドウを閉じます。

4.4. ストレージ仕様定義: Storage Specification Definition

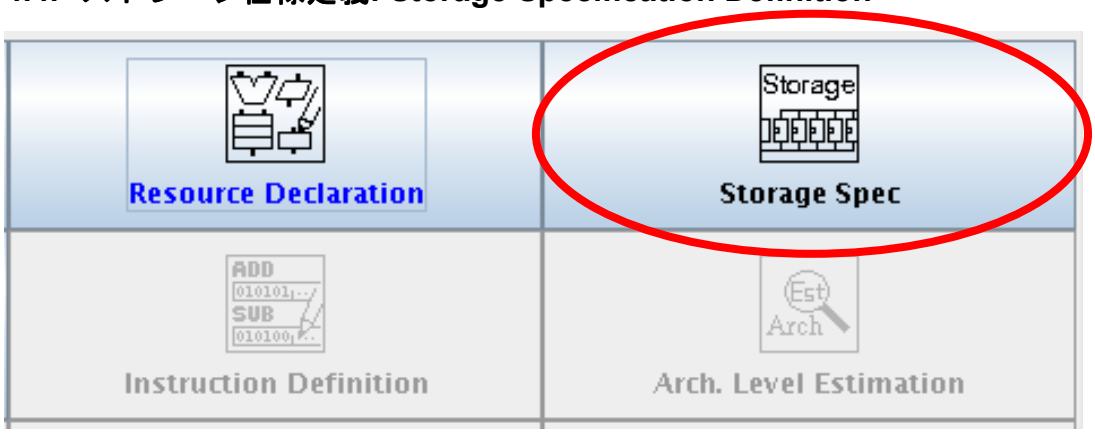

図 31 Storage Spec ボタン

[Resource Declaration]のステップが終了した時点で[Storage Spec]ボタンが有効になります。このボタンを押して[Storage Spec]ウインドウを開いてください。

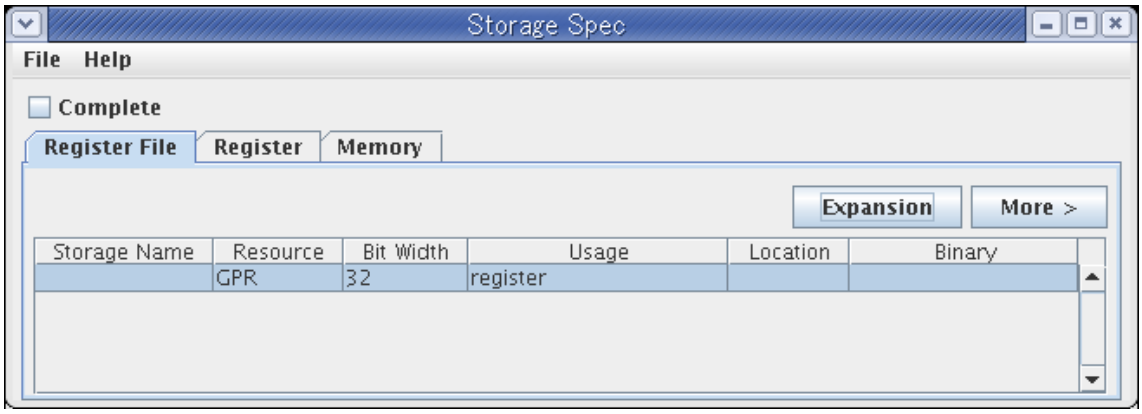

図 32 Storage Spec ウインドウ

このステップでは、設計者は汎用レジスタ、プログラムカウンタ、命令レジスタといったプロセッサで使用するストレージを定義します。ストレージの情報は命令定義で使用されます。

ストレージ仕様は次の 3 つから成ります。レジスタファイル仕様、レジスタ仕様、メモリ仕様です。以下、それぞれの仕様を設定していきます。

4.4.1. レジスタファイル仕様

ここでは、レジスタファイル仕様を定義します。メインウインドウでストレージ仕様ボタンをクリックしたときに、レジスタファイルを見ることができますが、これはレジスタ

ファイル・リソースを既にリソース宣言のステップで宣言していたためです。

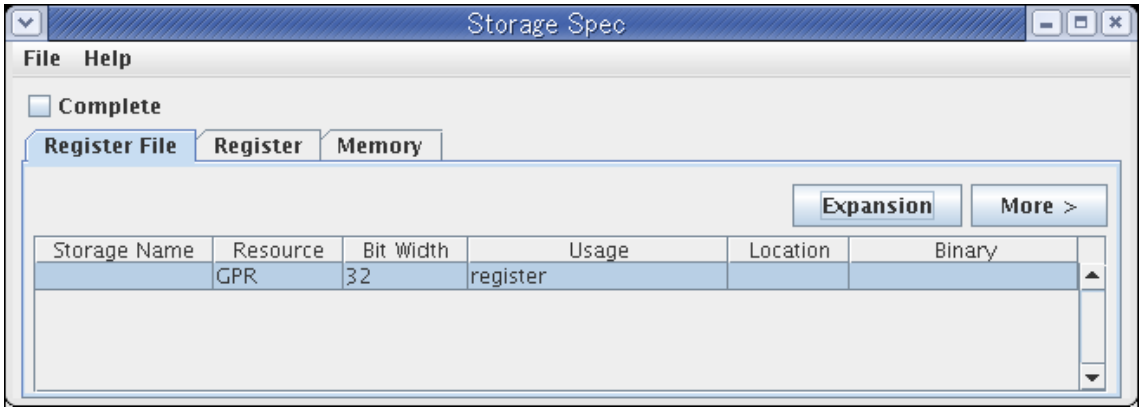

図 33 レジスタファイルの仕様設定

ここではストレージ名、ビット幅、使用方法、位置、バイナリ表現を設定します。次のようにレジスタファイル仕様を定義します。

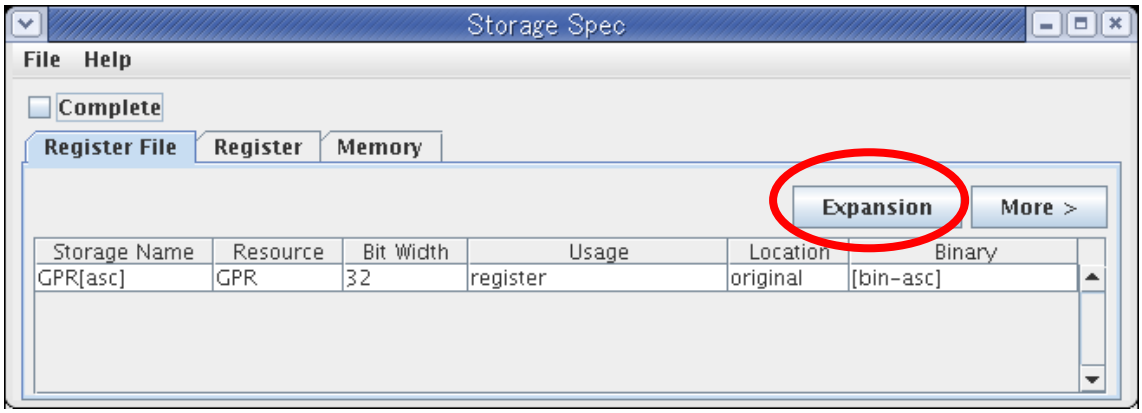

図 34 レジスタファイルの仕様を設定し終えた状態

各項目の意味は次の通りです。

Storage Name	ストレージ名としてアセンブリコードで用いられます。
Resource	ストレージとして用いられるリソースを指定します。
Bit width	ストレージのビット幅を指定します。
Usage	ストレージの使用方法を指定します。例えば、プログラムカウンタやスタックポインタといった用途を指定します。
Location	レジスタの位置を指定します。オーバーラップド・レジスタを定義する場合に用いられます。
Binary	レジスタファイル中のそれぞれのレジスタのバイナリ表現を表します。アセンブラでこの情報を用います。

レジスタファイル中には多くのレジスタがありますが、これらのレジスタを一括して表現するためのキーワードがあります。名前の表現としては次の 2 つを用いることができます。

[asc]	昇順に並べたレジスタを表します。GPR[asc]というように使用します。
[des]	降順に並べたレジスタを表します。GPR[des]というように使用します。

また、バイナリの表現としては次の 2 つを用いることができます。

[bin-asc]	昇順に並ぶバイナリ表現を表します。
[bin-des]	降順に並ぶバイナリ表現を表します。

情報を記述し終えたあと、[expansion]ボタンをクリックします。

[expansion]ボタンをクリックすると、確認ダイアログが現れます。

図 35 レジスタファイルの拡張を確認するダイアログ

[OK]をクリックすると、次のような[Register File expansion]ウインドウが現れます。

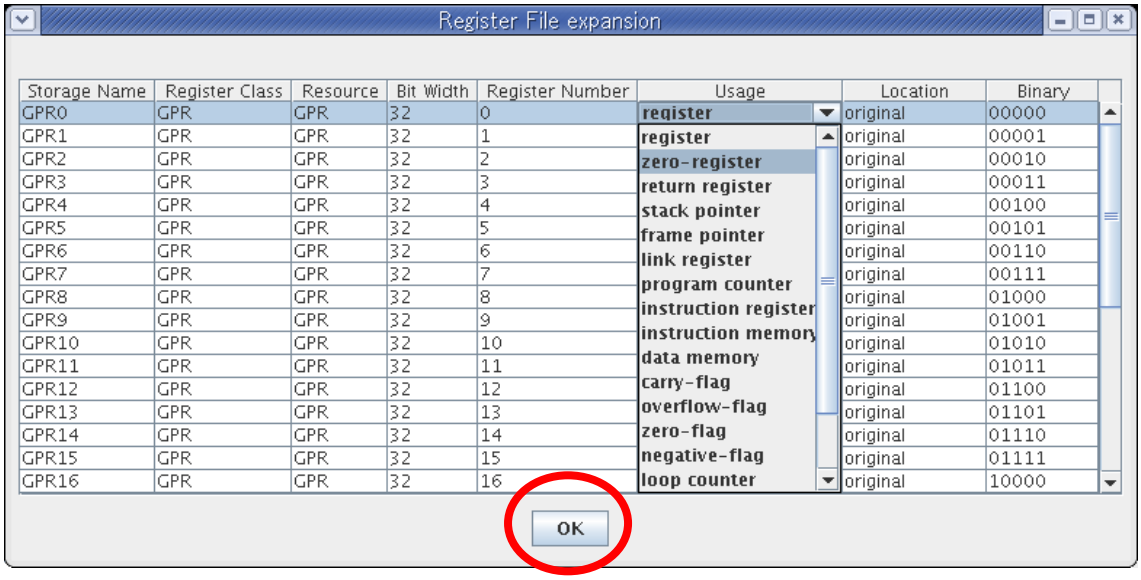

図 36 拡張されたレジスタファイルの一覧とそれぞれのレジスタの使用方法の設定

このウインドウでは先ほどの設定により拡張されたレジスタファイルが表示され、拡張された個々のレジスタの設定を行うことができます。

それぞれのレジスタは[Register File expansion]ウインドウに表示されます。ここでは、[Usage]の欄でそれぞれのレジスタの使用方法を設定します。

”small_RISC”では次のように変更します。

レジスタ	Usage
GPR0	zero register
GPR28	stack pointer
GPR29	frame pointer
GPR30	link register
GPR31	return register

レジスタの使用方法を設定し終えたら[OK]ボタンを押して、[Register File expansion]ウインドウを閉じます。これで[Register File]での設定は終了です。

4.4.2. レジスタ仕様

次にレジスタ仕様を定義します。[Storage Spec]ウインドウで[Register]タブをクリックすると、レジスタ仕様定義部が表示されます。

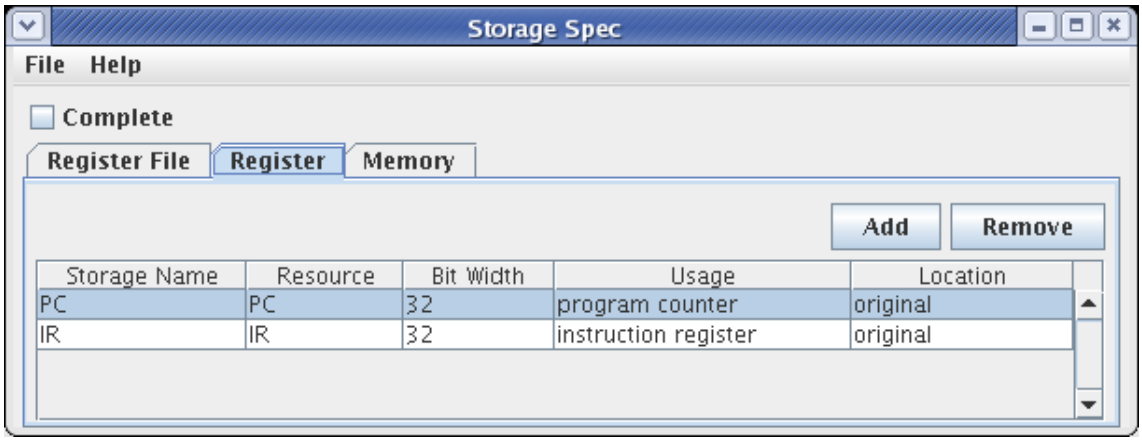

図 37 レジスタ仕様の設定

このウインドウでは、ストレージ名、リソース、ビット幅、使用方法といった情報を設定します。この情報はレジスタファイル仕様と同様に設定し、命令定義で使用されます。ここでは図 37 のように設定します。

4.4.3. メモリ仕様

最後にメモリ仕様を定義します。[Storage Spec] ウインドウで[Memory]タブをクリックすると、メモリ仕様定義部が表示されます。

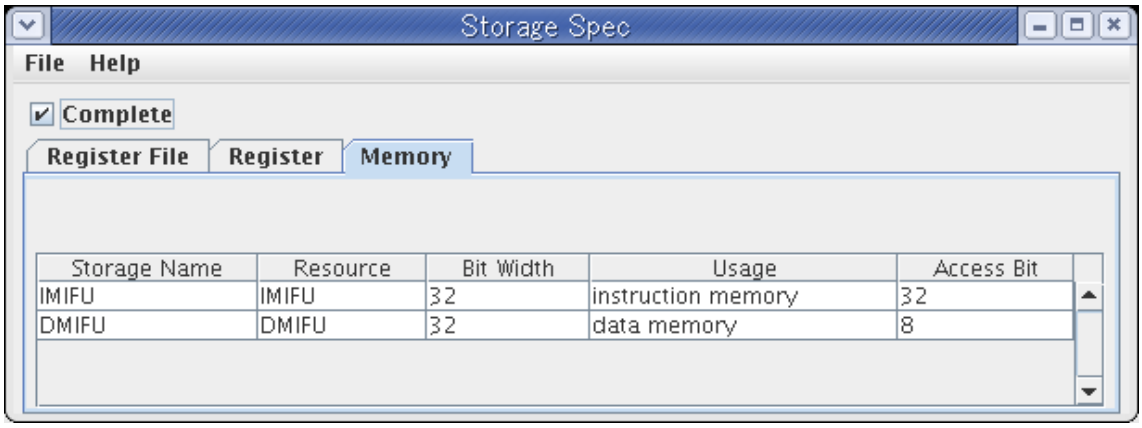

図 38 メモリ仕様の設定

この部分では、ストレージ名、リソース、ビット幅、使用方法、アクセスビット、といった情報を設定します。なお、アクセスビットではプロセッサがメモリにアクセスするときに必要となる最小のビット幅を指定します。命令メモリとデータメモリを図 38 のように定義します。

これでストレージ仕様の記述は完了です。[Complete]チェックをクリックし、それから[File]→[Close]としてストレージ仕様ウインドウを閉じてください。

4.5. 入出力インターフェース定義: Interface Definition

続いて、設計するプロセッサのエンティティ名と入出力ポート名の設定をします。ASIP Meister では、クロック信号とリセット信号、割り込み信号のほか、メモリアクセスを行う信号のみが外部と接続されているという形をとっています。[Interface Definition]のステップではこれらの外部と接続している信号線の名前や向き、ビット幅を設定します。

メインウインドウで[Interface Definition]ボタンをクリックしてください。

図 39 Interface Definition ボタン

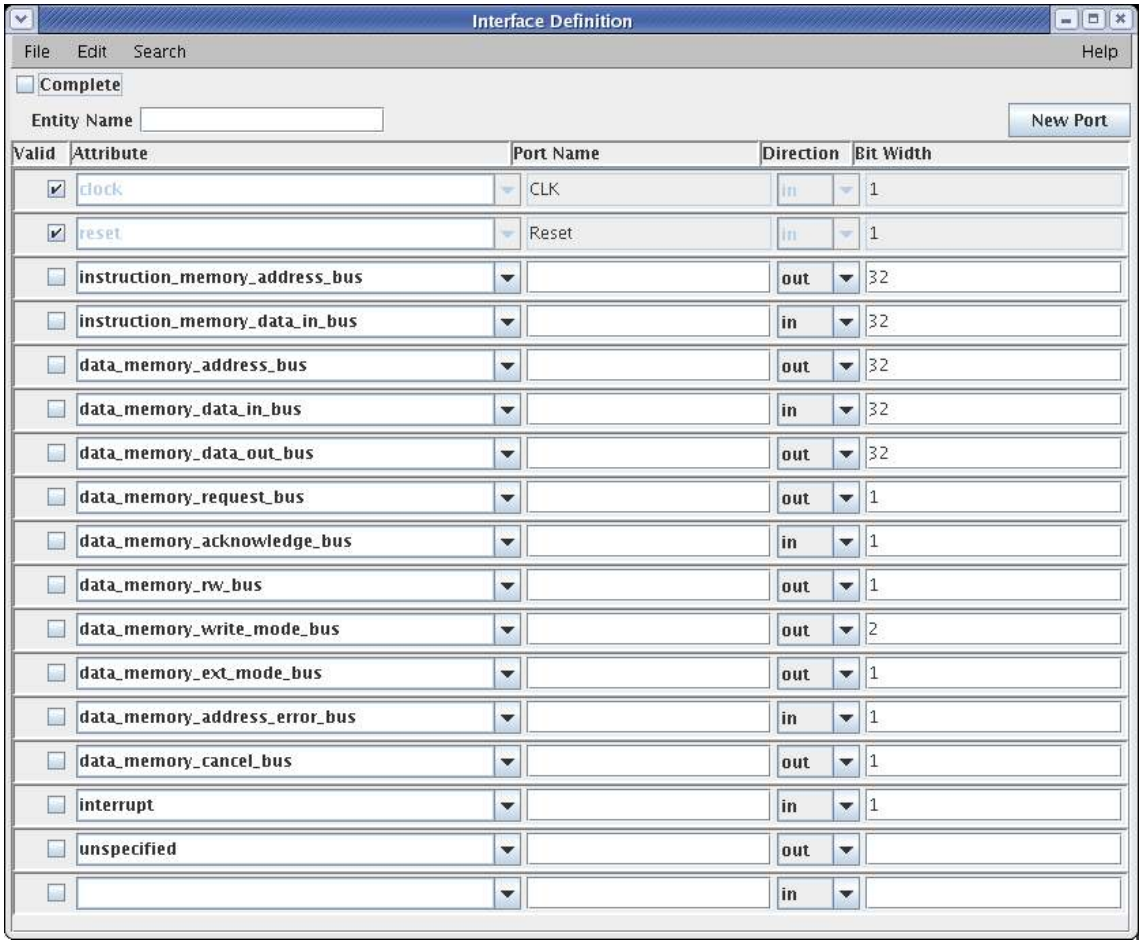

図 40 新規のインターフェース設定画面

上図のようなウインドウが開きます。このウインドウで新しい信号を追加することもありますが、基本的にはデフォルトの信号線の名前と向き、ビット幅を指定することになります。仕様書の内容に従い、信号線の各項目を設定していきます。

4.5.1. エンティティ名の設定

ASIP Meister が生成するプロセッサのトップレベルのエンティティ名を設定します。プロセッサのトップレベルの HDL 記述は”エンティティ名.vhd”というファイル名で生成されます。プロセッサの名前は”small_RISC”なので、エンティティ名も”small_RISC”とします。

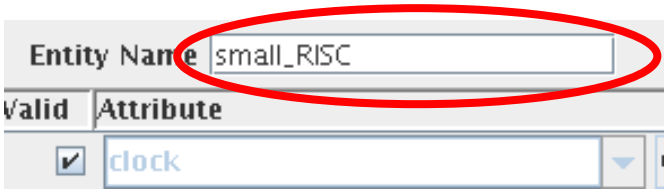

図 41 エンティティ名の設定

4.5.2. 信号線の設定

信号線の名前と向き、ビット幅を指定します。このチュートリアルでは次の図のように指定します。ところどころ、名前を設定していない信号がありますが、これらの信号は今回のチュートリアルでは使用しません。

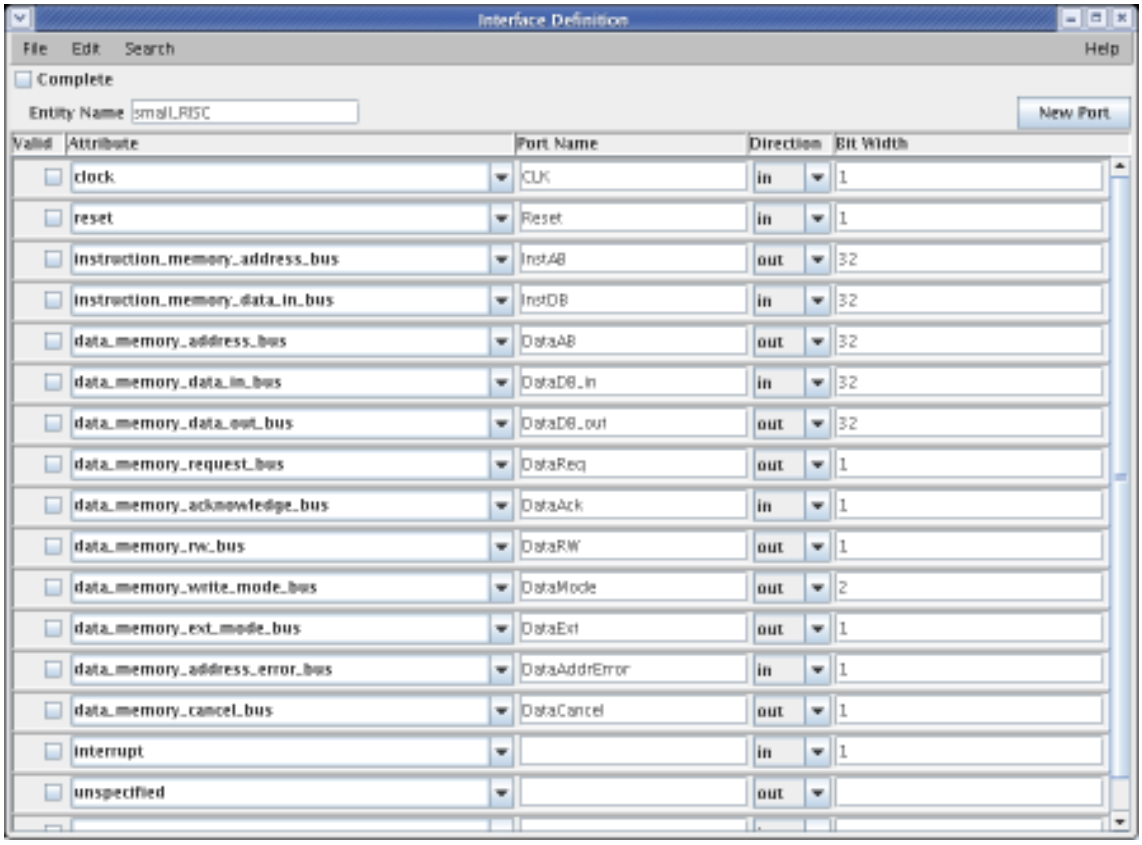

図 42 信号線の設定

信号の設定をし終えたら、その信号を有効にするために Valid チェックボックスをチェックします。使用する信号は、前のステップで名前を設定した信号だけとなります。したがって、名前を設定した信号のみ Valid チェックボックスをチェックしていきます。

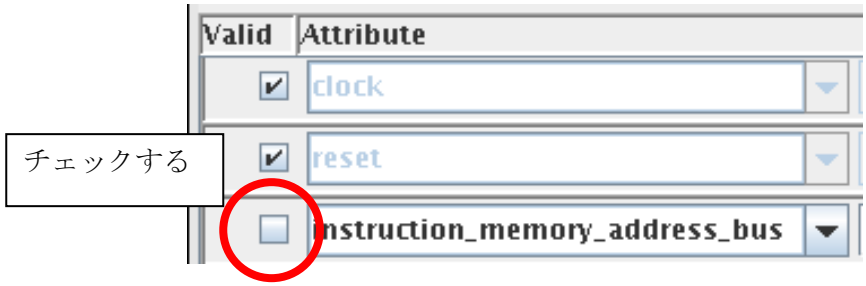

図 43 信号線の有効化

これでこのステップは終了です。Complete チェックボックスをチェックして設計を確定

して、[File]→[Close]でウインドウを閉じてください。

4.6. 命令タイプ・命令セット・例外の定義: Instruction Definition

このステージでは命令セットと、例外・割り込みを定義します。

図 44 Instruction Definition ボタン

メインウインドウで[Instruction Definition]ボタンをクリックして、[Instruction Definition]ウインドウを開いてください。

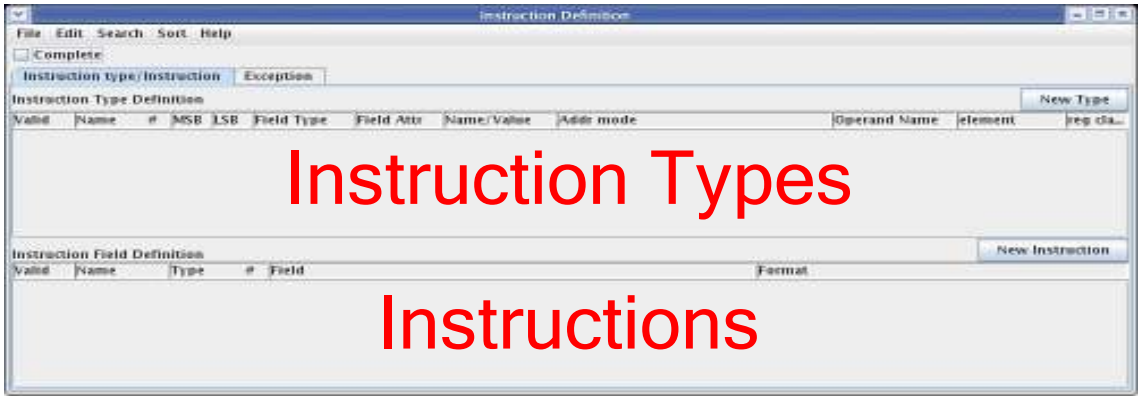

図 45 命令タイプと命令の定義

上図のウインドウが現れます。これから命令セットと例外・割り込みを定義します。

4.6.1. ASIP Meister における命令セットの定義

まずは命令セットを定義します。[Instruction Definition]ウインドウの[Instruction type /Instruction]タブを選択してください。

ASIP Meister の命令定義ではまず命令タイプを定義し、命令コードのフィールドの区切り方を決めます。次に各命令タイプに属する命令を宣言し、同時にその命令に固有の値（オ

ペコード）を割り当てます。

4.6.2. 命令タイプの定義

はじめに、画面上部の[Instruction type Definition]で命令タイプを定義します。仕様書の命令セットの説明に沿って、ここでは 4 つの命令タイプ”R_R”、”R_I”、”B”、”JP_T”を定義します。

まず、命令タイプ”R_R”を定義します。新しい命令タイプを定義するためには、[New Type]ボタンをクリックします。すると、新しい命令のタイプ名とフィールド数を入力するためのウインドウが開きます。

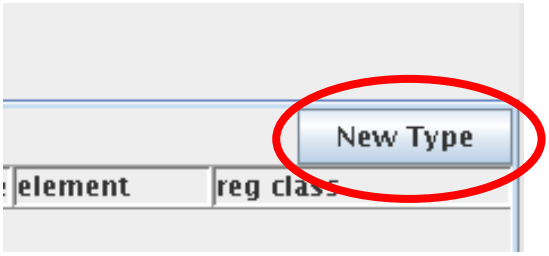

図 46 New Type ボタン

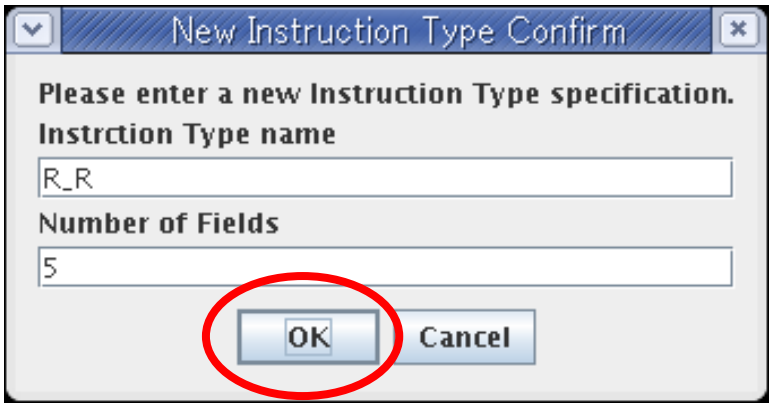

図 47 新しい命令タイプの名前とフィールド数の設定

ここで、命令タイプ名に”R_R”を、フィールド数に 5 を入力し、[OK]ボタンを押します。そうすると、次のようにフィールド数 5 の命令タイプ定義ウインドウ[Instruction Type Definition]ウインドウが開きます。

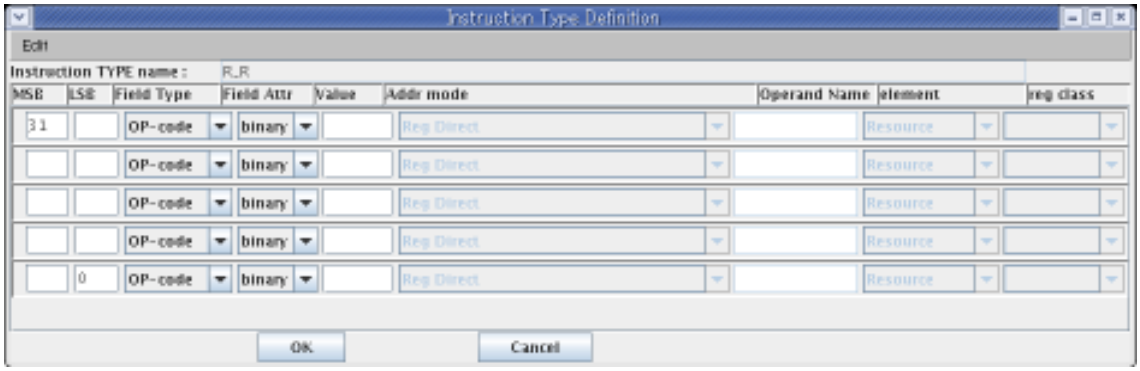

図 48 新しい命令タイプの設定

各フィールドの意味を次に説明します。

MSB、LSB	フィールドの範囲を指定します。MSB がフィールドの最上位ビット、LSB が最下位ビットをそれぞれ表します。
Field Type、 Field Attr	後述。
Value	このフィールドがとるビット値、またはビット名を指定します。
Addr mode	アドレッシングモードを指定します。フィールドタイプが[Operand]のときのみ指定可能です。
Operand Name	アセンブラが使用するオペランドの名前を指定します。
element、 reg class	フィールドのオペランドがどのようなリソースを用いるのかという情報を指定します。フィールドタイプが[Operand]であり、アドレッシングモードでレジスタを使用したときのみこの項目を指定できます。

[Field Type]、[Field Attr]ではフィールドタイプと属性をそれぞれ指定します。指定はボックス横の▼をプルダウンして行います。なお、フィールド属性はフィールドタイプによって選択できる属性が異なっています。

Field Type	選択できる[Field Attr]	選択基準
Opecode	Binary	この命令タイプを使用する命令は、ここで定義したフィールドの値を変更しない場合
	Name	この命令タイプを使用する命令は、命令ごとにフィールドの値を変更する場合
Operand	Name	[name]しか変更できません

仕様書によると、R_R 形式（レジスタ-レジスタ形式）の命令タイプは次のようになって

います。

31　　26	25　　21	20　　16	15　　11	10　　0
000000	rs0	rs1	rd	func
オペコード	入力レジスタ 0	入力レジスタ 1	出力レジスタ	命令固有のオペコード

31 ビットから 26 ビットまではオペコードとなっており、値も”000000”と固定値です。したがって、この範囲では[Opecode]-[Binary]と設定した上で”000000”と指定します。

次に、25 ビットから 11 ビットまでの 3 つのレジスタ指定オペランドがありますが、これは一つ一つ範囲を指定してフィールドタイプを[Operand]と指定します。また、全て GPR の汎用レジスタに直接アクセスするため、アドレッシングモードに[Reg Direct]を指定、リソースには”GPR”を指定します。

最後に、10 ビットから 0 ビットの範囲ですが、これは命令ごとにオペコードを割り当てるため、ここでは名前のみを設定します。したがって、[OP-code]-[name]を指定します。

仕様書に従って値を入力すると次の図のようになります。

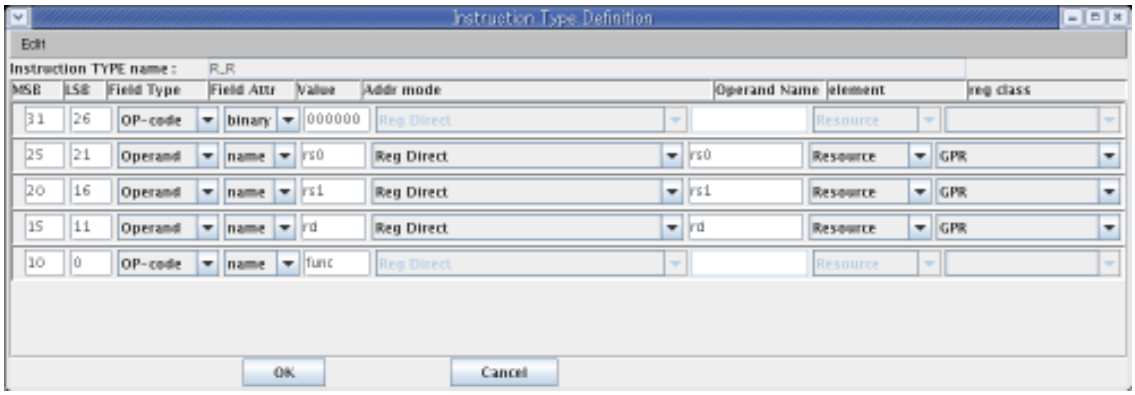

図 49 R_R タイプ入力後の[Instruction Type Definition]ウインドウ

最後にウインドウの[OK]ボタンを押すと[Instruction Definition]ウインドウに戻ります。

図 50 R_R タイプ入力後の[Instruction Definition]ウインドウ

同様にして、命令タイプ”R_I”、”B”、”JP_T”を定義します。

図 51 R_I, B, JP_T タイプの入力後の[Instruction Definition]ウインドウ

メモ：命令タイプの変更

命令タイプを定義し終えたあと、定義を変更する場合は、命令タイプの名前のところをダブルクリックしてください。そうすれば[Instruction Type Definition]ウインドウが開いて、命令タイプを変更できます。

メモ：命令タイプの削除

一度定義した命令タイプを削除する場合は、命令タイプをマークする必要があります。マークをするためには、命令タイプの名前のところをクリックして、名前の文字列を反転させたあと、[Edit]メニューから[Mark]を選択するか、名前の文字列のところでマウスの真

ん中のボタンをクリックします（真ん中のボタンがなければ、左右のボタンを同時に押します)。すると、名前の文字列の背景色が青くなります。この状態で、[Edit]メニューから[TypeDelete]を選択するとマークした命令タイプを削除できます。

<注意> Valid をチェックしていると削除できません.

4.6.3. 命令の定義

命令タイプを全て設定し終わったら、次は命令を定義していきます。命令定義は[Instruction Definition]ウインドウ下部の[Instruction Field Definition]で行います。

small_RISC の仕様書に従い、それぞれ次のようにの各命令を割り当てます。

ADD（加算命令）	R_R
SUB（減算命令）	R_R
LOAD（ロード命令）	R_I
STORE（ストア命令）	R_I
J（無条件分岐命令）	JP_T
BEZ（条件分岐命令）	B

まず"R_R"命令タイプを使用して、加算命令 ADD を定義します。[Instruction Definition]ウインドウの[New Instruction]ボタンを押してください。命令名を入力するウインドウが現れます。

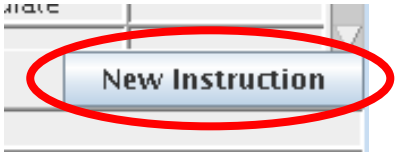

図 52 New Instruction ボタン

図 53 新しい命令の名前の設定ウインドウ

”ADD”と入力して[OK]ボタンをクリックしてください。すると、命令の設定ウインドウが開きます。

図 54 新しい命令の設定

左のフレームで命令タイプを設定し、右のフレームで ADD 命令に固有な情報を設定します。左のフレームでは”R_R”を指定してください。次に右に表示されるフィールドでは第 5 フィールドの値を設定します。仕様書の通り[Value]を” 00000100000”とします。ウインドウ上部の[Format]にも入力を忘れないようにしてください。”ADD rd rs0 rs1”と入力します。

以上を入力し終えたら[OK]ボタンを押し、ADD 命令の定義を終了します。[Instruction Definition]ウインドウに戻ります。

同様にして、その他の命令を定義します。

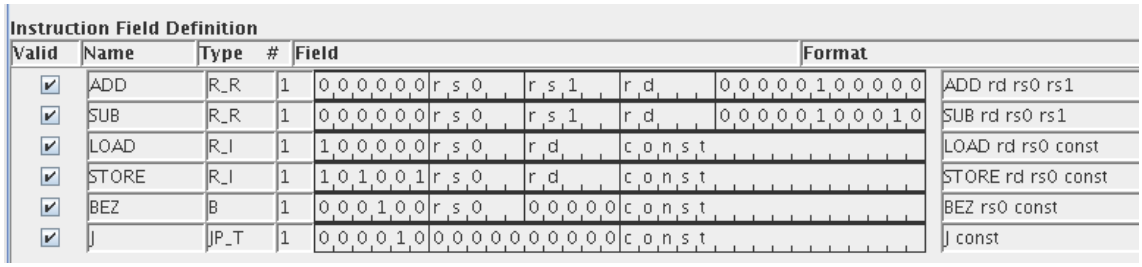

図 55 全ての命令

上図のようになれば命令定義は完了です。

メモ：命令の変更

命令を定義し終えたあと、定義を変更する場合は、命令の名前のところをダブルクリックしてください。そうすれば、命令を定義するウインドウが開き、定義を変更できます。

メモ：命令の削除

一度定義した命令を削除するためには、命令をマークする必要があります。マークするためには、命令の名前のところをクリックし、名前の文字列を反転させたあと、[Edit]メニューから[Mark]を選びます。もしくは、命令の名前のところでマウスの真ん中のボタンを押します（真ん中のボタンがなければ左右のボタンを同時に押します）すると、命令の名前の背景色が青くなります。この状態で[Edit]メニューから[InstDelete]を選択するとマークした命令を削除できます。

<注意> Valid をチェックしていると削除できません.

4.6.4. 割り込み（例外）の設定

命令セットの定義がすべて完了したら、次は例外・割り込みの定義を行います。[Instruction Definition]ウインドウの[Exception]タブをクリックしてください。

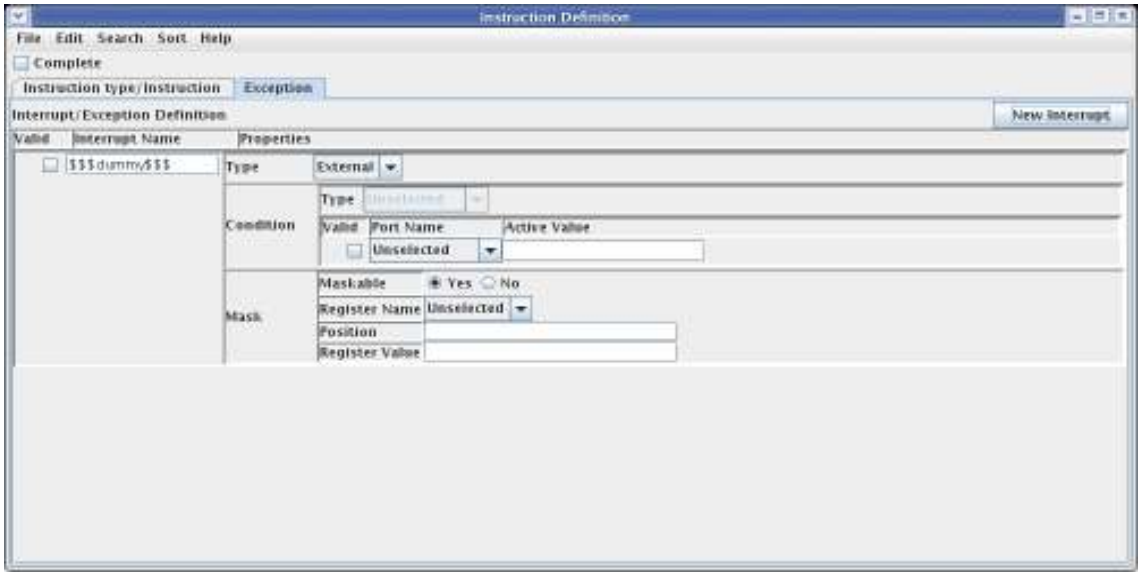

図 56 例外・割り込みの新規設定画面

ここに例外・割り込みを入力していきます。画面にはすでにひとつの割り込み定義バーがありますが、さらに例外・割り込みを定義するには[New Interrupt]ボタンをクリックします。ボタンをクリックすると、新しい入力バーが現れます。

それぞれの項目の意味は次のとおりです。

Valid	例外・割り込みの設定の有効・無効を指定します
Interrupt Name	例外・割り込みの名前を設定します
Type	▼プルダウンメニューから割り込みの種類を指定します
Condition	割り込みの発生条件を指定します

Mask	割り込みがマスク可能である場合、マスクの設定を行います

ここでは”dummy”を改変して、リセット割り込み”reset”を定義することにします。まずは[Interrupt Name]を”reset”に変更します。

次に、割り込みのタイプを”Reset”にします。

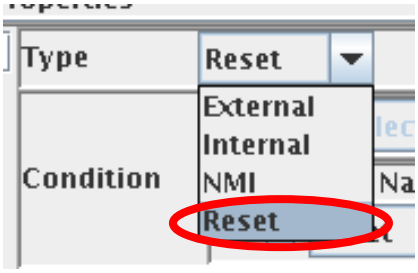

図 57 例外・割り込みの種類の設定

次は[Condition]でリセット信号の信号線を指定し、その信号の値がどんな値になったときにリセットが有効になるのかを設定します。仕様どおり、リセット信号として[Port Name]に”Reset”を設定し、この信号の値が”1”になったときにリセットが働くようにします。また、この信号でのリセットを有効にするために[Valid]チェックボックスにチェックをつけます。

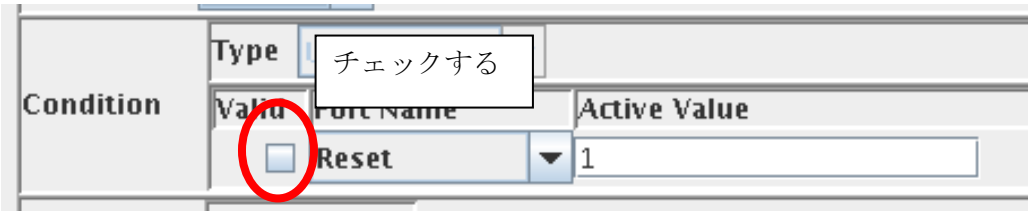

図 58 リセット信号線の設定

リセット割り込みの情報を全て設定し終えると次のような画面になります。ここで、最後にこのリセット割り込みを有効にするために、[Valid]チェックボックスをチェックします。

メモ：例外・割り込みの削除

一度定義した例外・割り込みを削除するためには、例外をマークする必要があります。マークするためには、例外の名前のところでマウスの真ん中のボタンを押します（真ん中のボタンがなければ左右のボタンを同時に押します）。すると、例外の名前の背景色が青になります。この状態で[Edit]メニューから[ExptDelete]を選択するとマークした例外を削除できます。

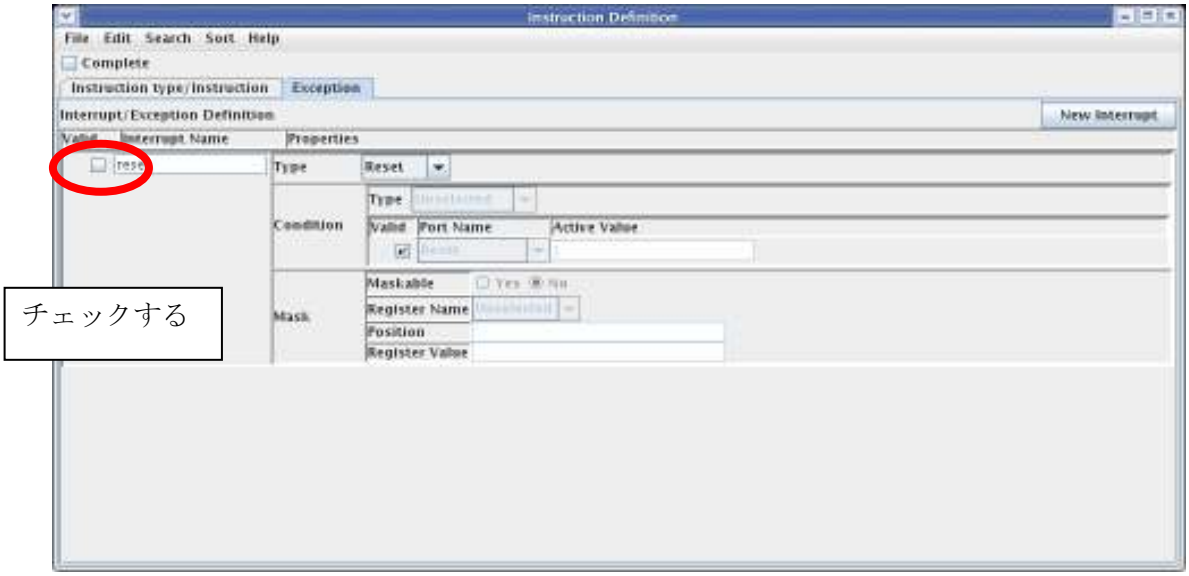

図 59 例外・割り込みの有効化

以上で命令セット、例外・割り込みの設定は終了です。設計を確定するために[complete]チェックボックスをチェックし、[File]→[Close]でウインドウを閉じてください。

4.7. 概略見積もり: Arch. Level Estimation

このステージでは、これまでの設計情報から HDL 生成を行う前に設計品質指標（面積、遅延時間、消費電力）を見積もります。設計の詳細が決定する前の見積もりであり、精度は粗いですが、高速な見積もりが可能です。概略見積もりの結果が所望の品質を得ていない場合は、これまでの設計内容を変更して、再度見積もりを行います。

今回は設計が見積もりを満足するように作られていますので確認の意味でこの作業を行います。

図 60 Arch. Level Estimation ボタン

メインウインドウの[Arch. Level Estimation]ボタンをクリックしてください。

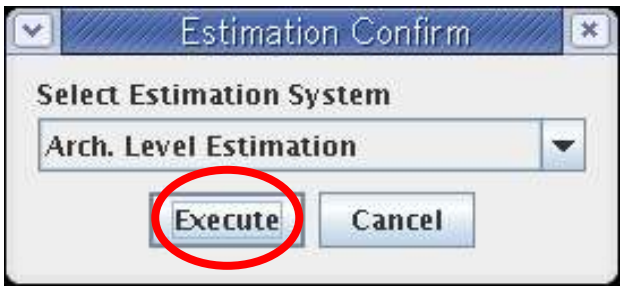

図 61 見積もりの実行

上図のような[Estimation Confirm]ウインドウが現れ、見積もりができるようになります。見積もりはこのウインドウの[Execute]ボタンを押すことで開始します。

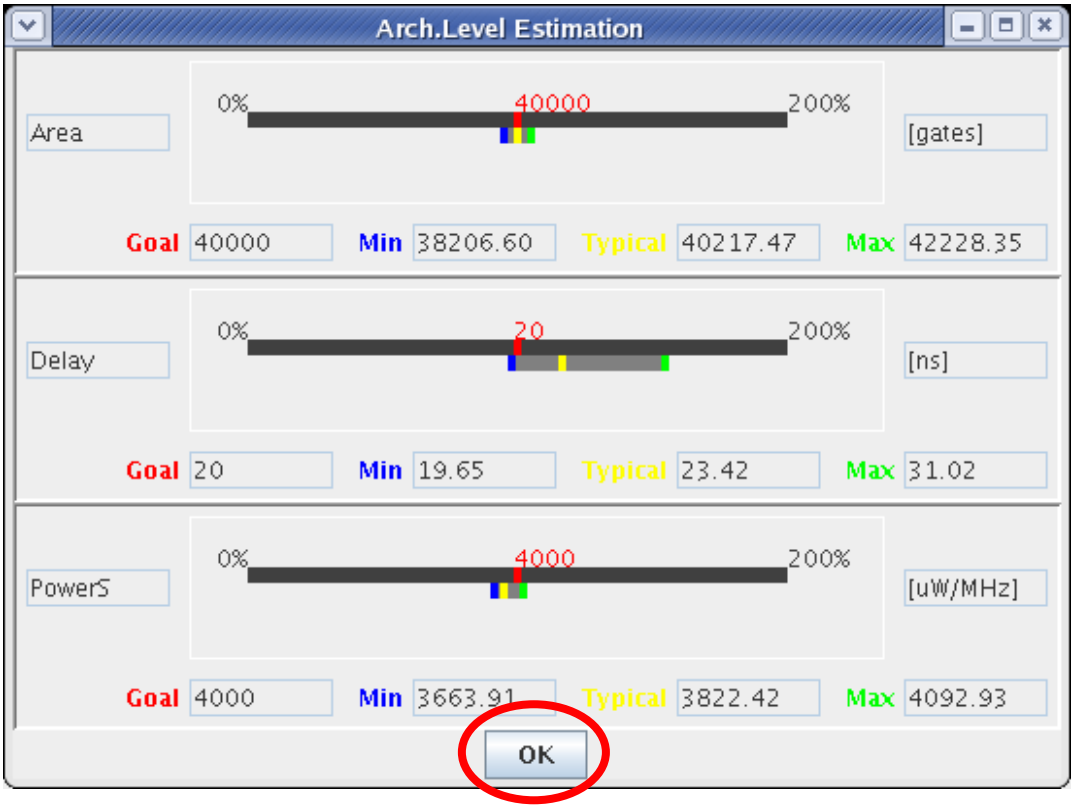

図 62 見積もりの実行結果

今までの設定が正しく行われていれば見積もり結果は上図のような画面になります。この見積もりでは、設計目標を満たしているといえるでしょう。

これでこのステージは終了です。見積もりを確認したら[OK]ボタンを押して画面を閉じてください。

4.8. アセンブラ生成: Assembler Generation

このステージでは、設計するプロセッサのアセンブラ記述を生成します。

ASIP Meister にはどのような命令体系にも対応できるメタアセンブラが付属しています。メタアセンブラを用いて設計したプロセッサの命令列を生成するためには、プロセッサや命令の情報といった、アセンブラ記述が必要となります。[Assembler Generation]ではメタアセンブラに渡すアセンブラ記述を出力します。

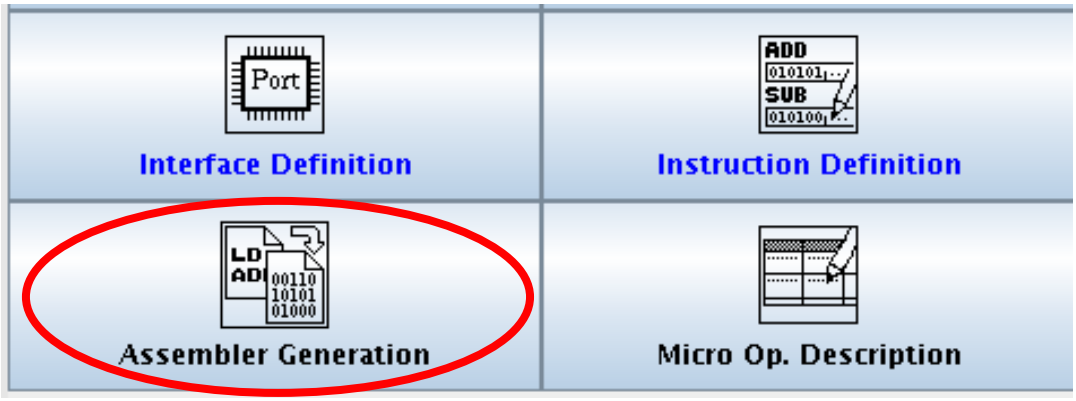

図 63 Assembler Generation ボタン

まずメインウインドウ内で[Assembler Generation]ボタンを押して、[Generation Confirm]ウインドウを開きます。

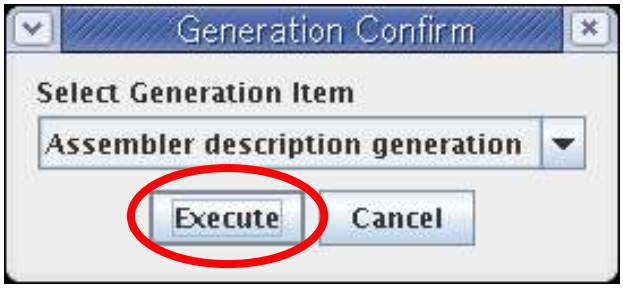

図 64 Assembler 記述の生成

このウインドウで[Execute]ボタンを押してください。アセンブラ記述生成プログラムが起動し、アセンブラ記述ファイルを生成します。

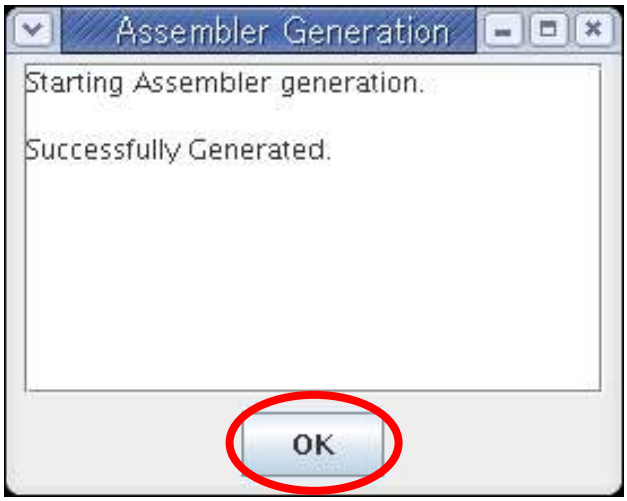

図 65 Assembler 記述の生成結果

生成プログラムが終了すると、上図のようなメッセージが表示されます。エラーがないことを確認し、[OK]ボタンを押してください。これでこのステージは終了です。

エラーがあった場合は、これより前のステージにおいて間違った記述があったということです。第 1 ステージからひとつひとつ確認してみてください。

4.8.1. 生成ファイルの確認

ASIP Meister では起動時にカレントディレクトリに"meister"というディレクトリを作成します。アセンブラ記述の生成に成功したら、この"meister"ディレクトリの下に small_RISC.des というアセンブラ記述ファイルが生成されているはずですのでご確認ください。

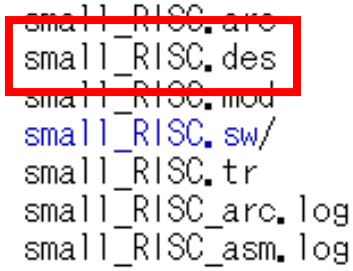

図 66 生成されたアセンブラ記述

4.9. マイクロ動作記述: Micro Op. Description

このステージでは、それぞれの命令と割り込みの動作をパイプラインのステージごとに記述します。記述においてはマクロを使用することができ、同じ記述を繰り返す場合、記

述内容を減らすことができます。ここでは、[Instruction Definition]で定義した 6 命令（ADD、SUB、LOAD、STORE、BEZ、J）の動作とリセット割り込みの動作を記述していきます。

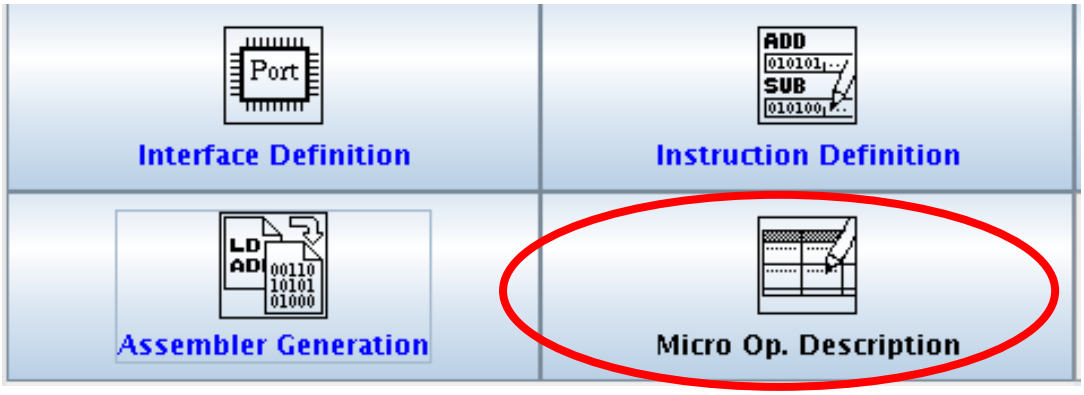

図 67 Micro Op. Description ボタン

まず、メインウインドウで[Micro Op. Description]ボタンを押して、[Micro Op. Description]ウインドウを開きます。

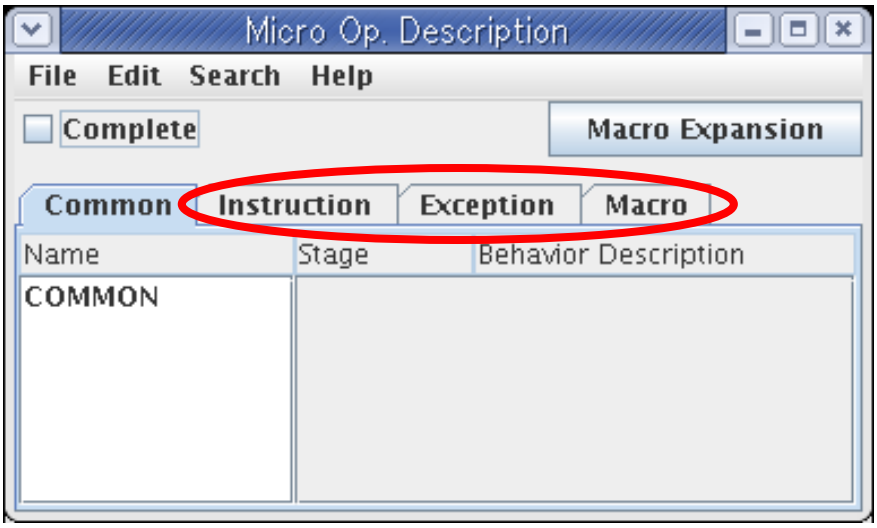

図 68 Micro Op. Description ウインドウの記述項目

ウインドウ内のタブを選択し、定義する動作を決めます。タブには次のようなものがあります。

Instruction	命令ごとの動作記述
Exception	例外・割り込み処理の動作記述
Macro	マクロ定義の記述

以降、[Macro]、[Instruction]、[Exception]の順に定義していきます。

4.9.1. マクロ定義: [Macro]

small_RISC のマイクロ動作記述では命令のフェッチを行うマクロ”FETCH”を定義することにします。まず、[Micro Op. Description]の[Micro]タブをクリックしてください。

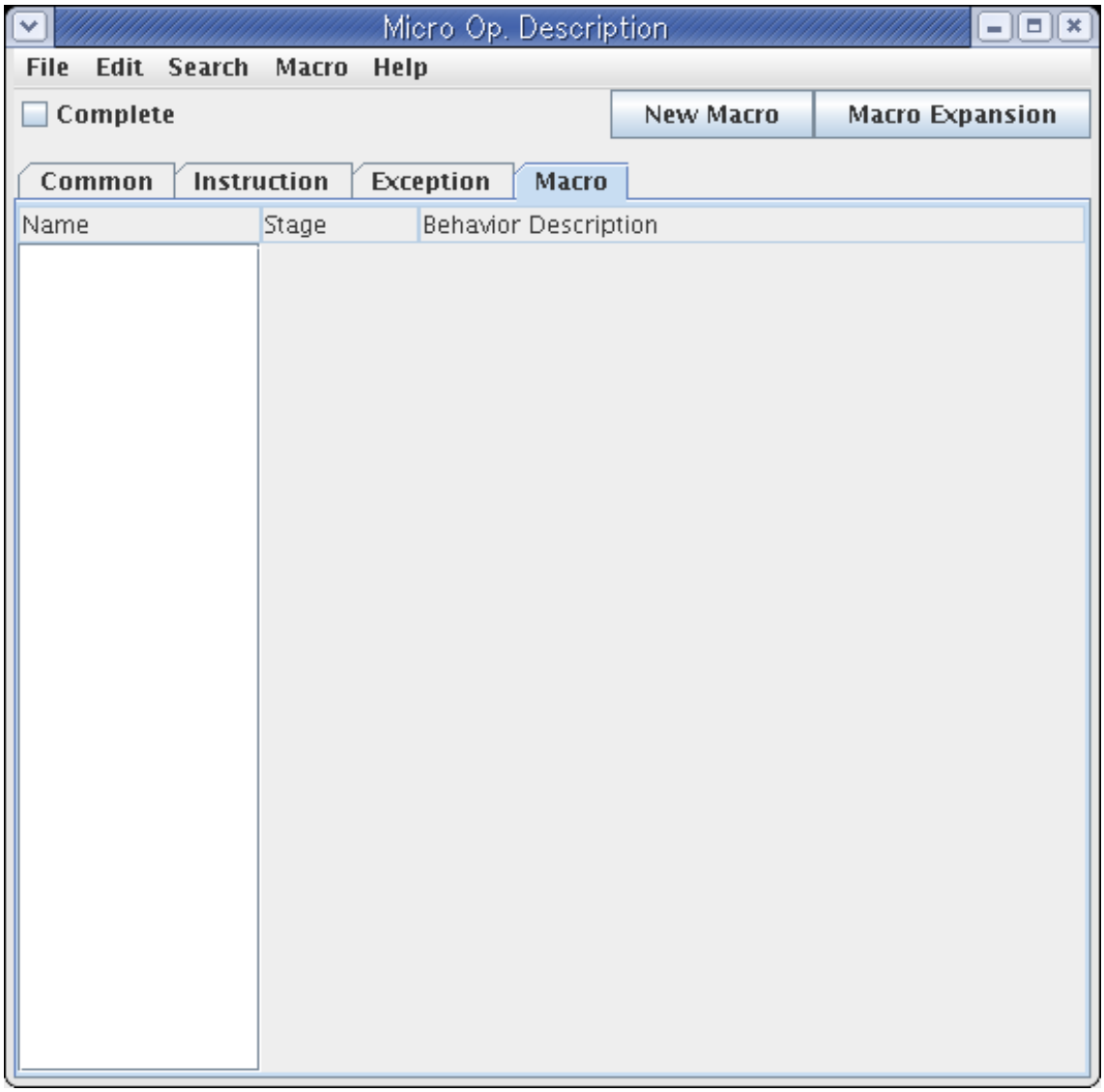

図 69 Macro タブ選択後のウインドウ

[Macro]メニューから[New Macro]を選択する、あるいは[New Macro]ボタンを押すと図のようなウインドウが開きます。

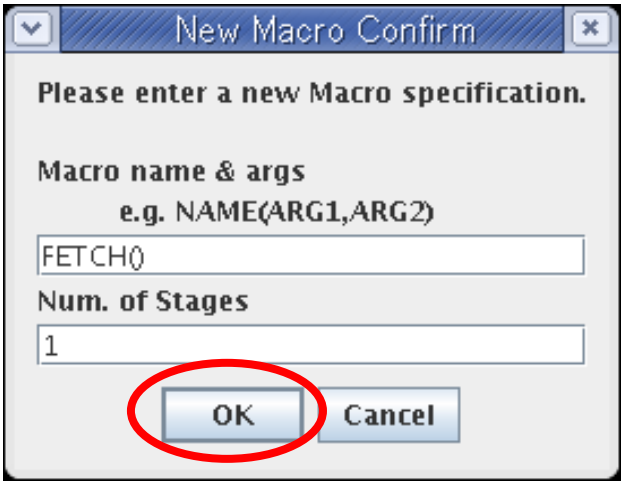

図 70 FETCH マクロの名前とステージ数

このウインドウで定義するマクロの名前、引数、ステージ数を指定します。”FETCH”マクロの場合は上図のように定義します。入力し終えたら[OK]ボタンを押してください。”FETCH”マクロの動作を記述するフィールドが現れます。

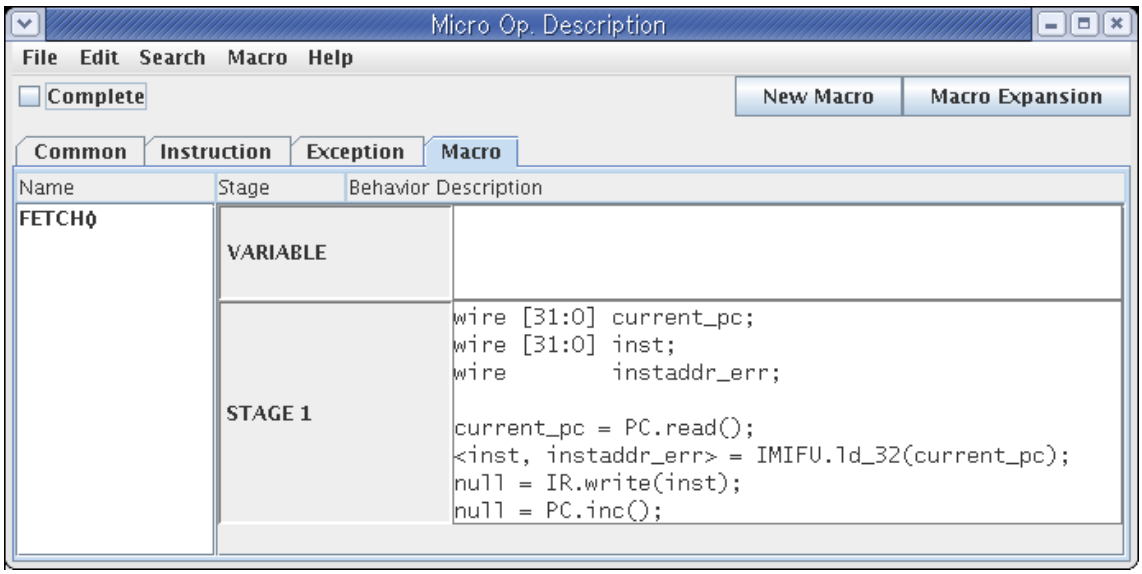

図 71 FETCH マクロの入力

ここで定義したマクロは全ての命令のマイクロ動作記述で使用します。FETCH マクロの場合上図のように記述します。これでマクロの記述は終了です。

4.9.2. 命令の動作記述: [Instruction]

[Micro Op. Description] ウインドウの [Instruction]タブを選択すると、左のフレームに[Instruction Definition]で定義した命令の一覧が現れ、右のフレームに命令のマイクロ動作記述を定義する画面が現れます。なお、中央に現れる[Stage]の欄は[Design Goal & Arch.

Design]で定義したステージ名を表します。

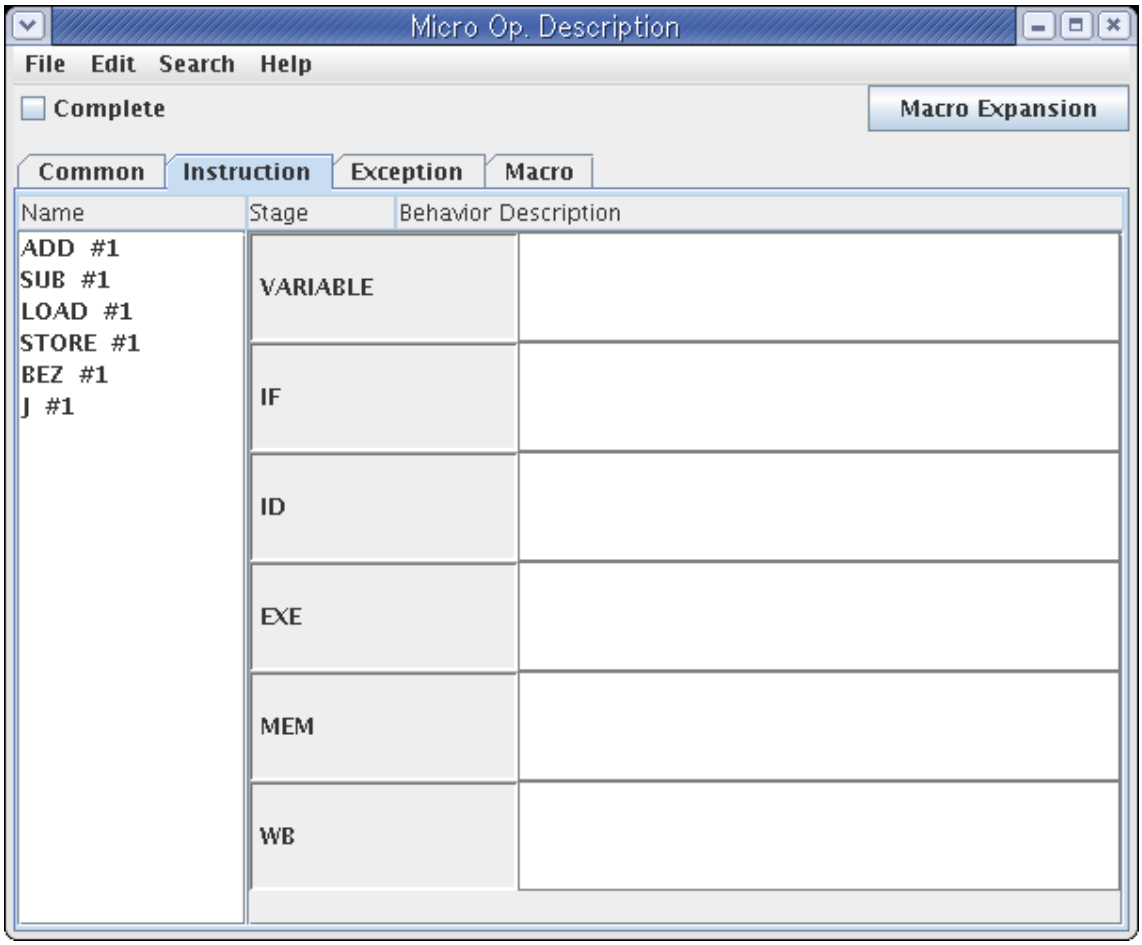

図 72 命令のマイクロ動作記述画面

以下、”ADD”、”SUB”、”LOAD”、”STORE”、”J”、”BEZ”命令の記述例を示します。

4.9.2.1. ADD 命令

ADD 命令は第 2 ステージでレジスタの内容を読み込み、第 3 ステージで演算を行います。そして、第 5 ステージで演算結果をレジスタに書き戻します。この動作を行うマイクロ動作記述を入力した後の画面は次のようになります。

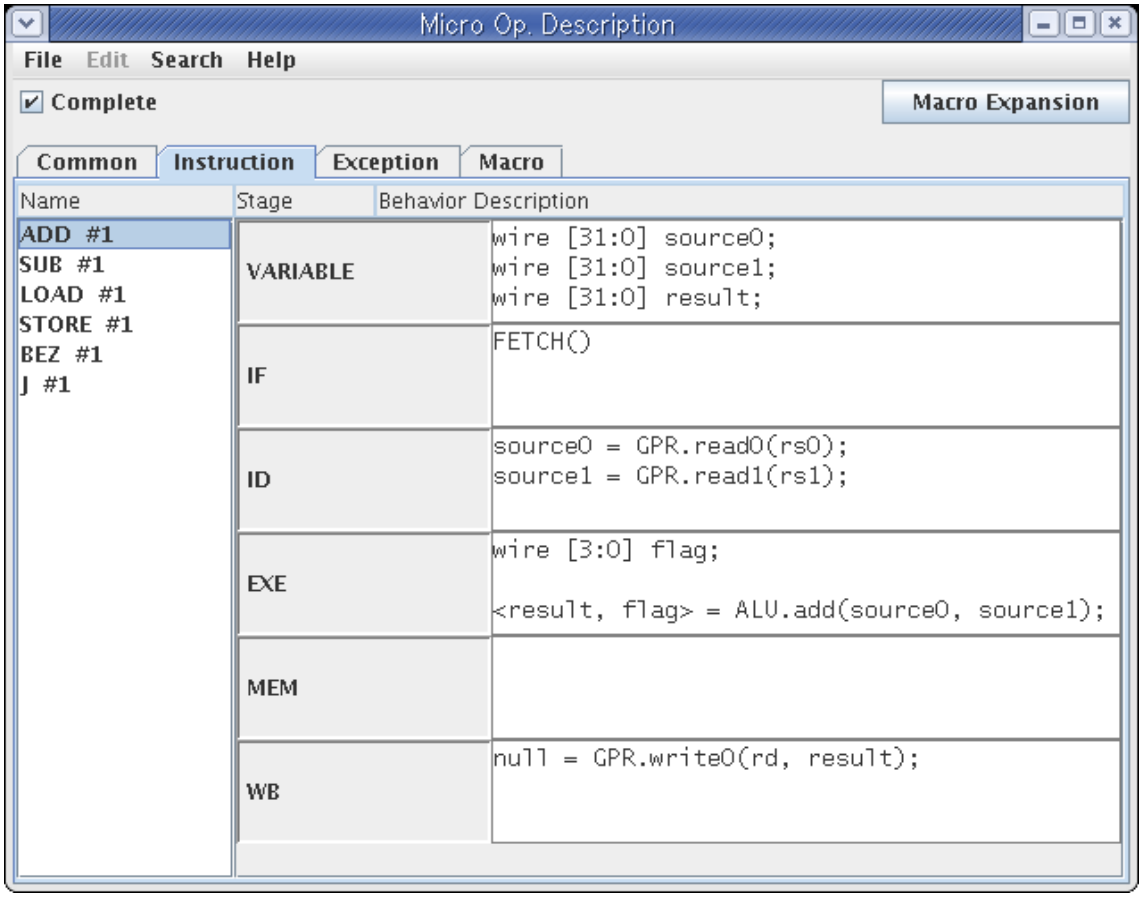

図 73 ADD 命令のマイクロ動作記述

4.9.2.2. SUB 命令

SUB 命令はほぼ ADD 命令と記述が同じです。入力後の画面は次のようになります。

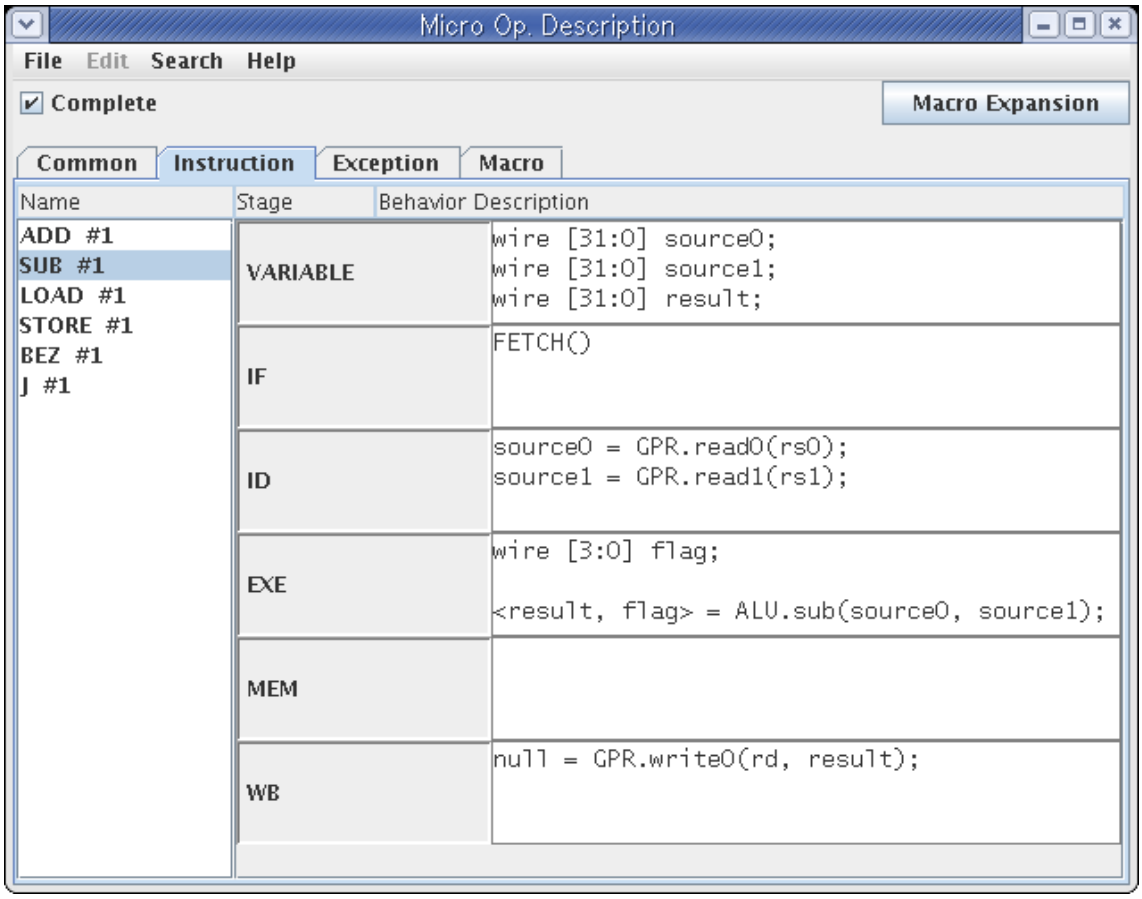

図 74 SUB 命令のマイクロ動作記述

4.9.2.3. LOAD 命令

LOAD 命令は第 2 ステージで読み込むメモリアドレスのベースアドレスとインデックスを取得し、第 3 ステージで読み込むメモリアドレスを計算します。そして、第 4 ステージで実際にメモリにアクセスしてメモリの内容をレジスタに読み込みます。この内容のマイクロ動作記述を入力した後の画面は次のようになります。

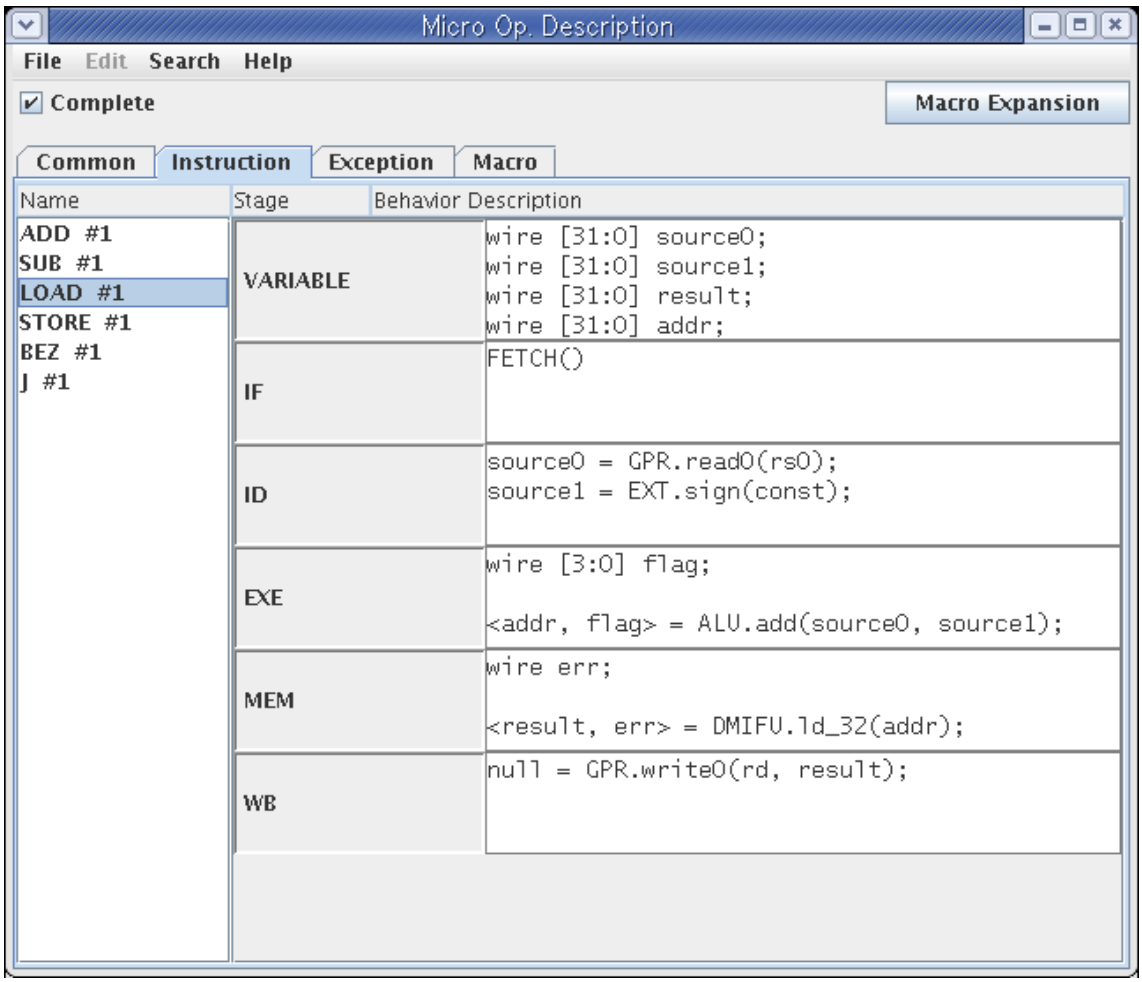

図 75 LOAD 命令のマイクロ動作記述

4.9.2.4. STORE 命令

STORE 命令は LOAD 命令と同様に、第 2 ステージで書き込むアドレスのベースアドレスのインデックスを得ていますが、さらに、書き込むデータも得ています。そして、第 3 ステージで書き込むアドレスを計算し、第 4 ステージで実際にメモリに書き込んでいます。この内容をマイクロコード記述した後の画面は次のようになります。

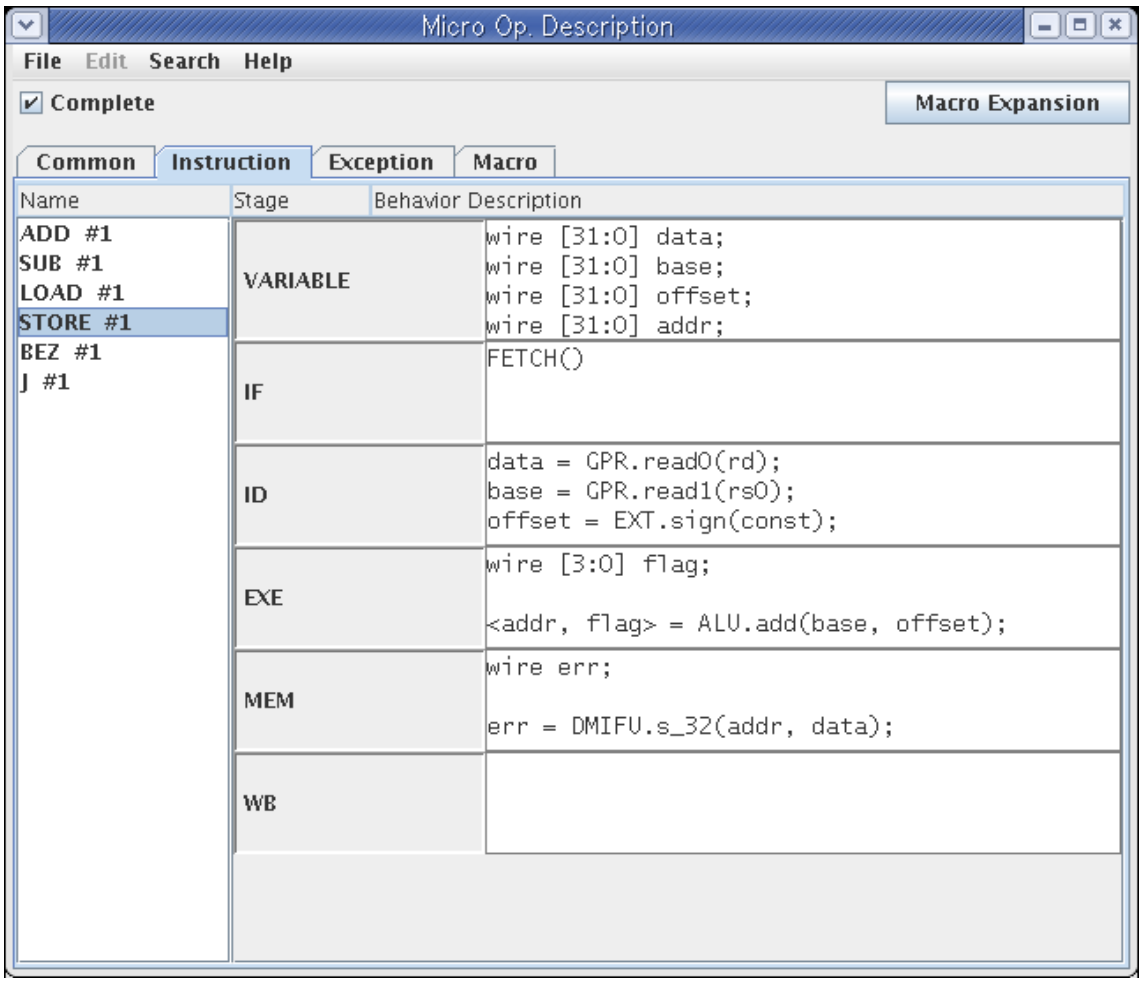

図 76 STORE 命令のマイクロ動作記述

4.9.2.5. BEZ 命令

BEZ 命令は、第 2 ステージで 0 と比較すべきレジスタの内容と現在の PC の値を取り出します。そして、第 3 ステージで分岐するアドレスを計算し、もし、レジスタの内容が 0 であれば実際に PC の値を書き換えて分岐します。この動作のマイクロ動作記述をした後の画面は次のようになります。

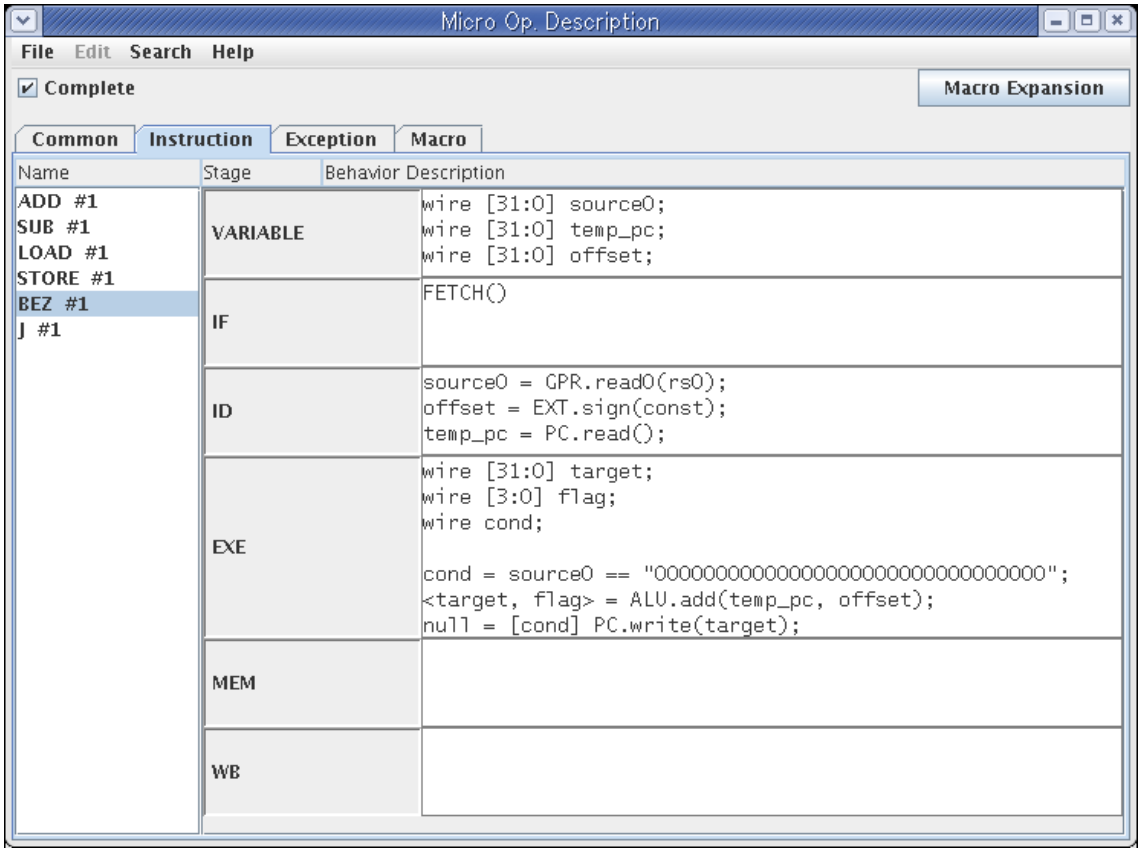

図 77 BEZ 命令のマイクロ動作記述

4.9.2.6. J 命令

J 命令は BEZ 命令と同じような動作をしますが、レジスタの比較を行わないことが違います。まず、第 2 ステージで現在の PC の値を読み込みます。そして、第 3 ステージで分岐先のアドレスを計算し、PC の値を書き換えて分岐します。この動作のマイクロ動作記述後の画面は次のようになります。

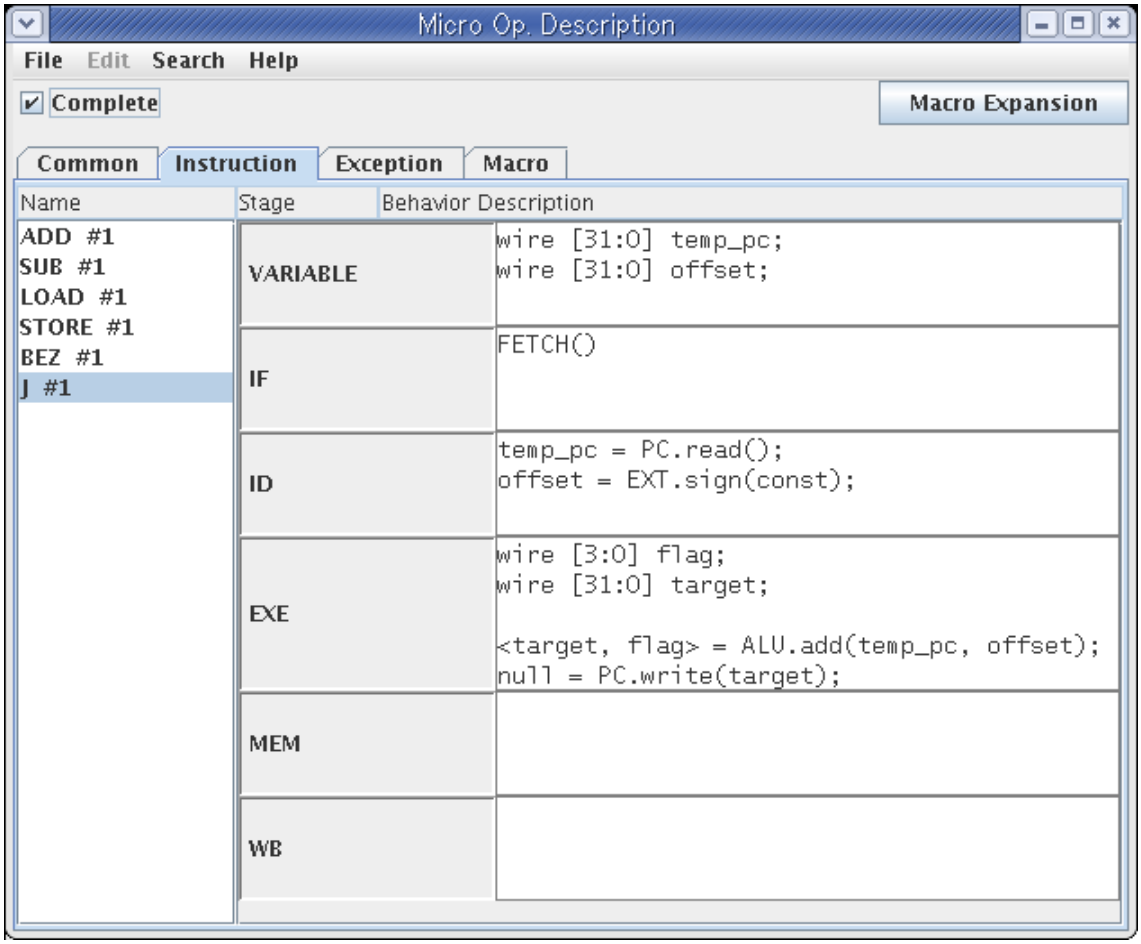

図 78 J 命令のマイクロ動作記述

4.9.3. 例外・割り込みの記述

[Micro Op. Description] ウインドウの [Exception] タブを選択すると、左のフレームに [Instruction Definition] で定義した割り込みの一覧が現れ、右のフレームに割り込みのマイクロ動作記述を定義する画面が現れます。

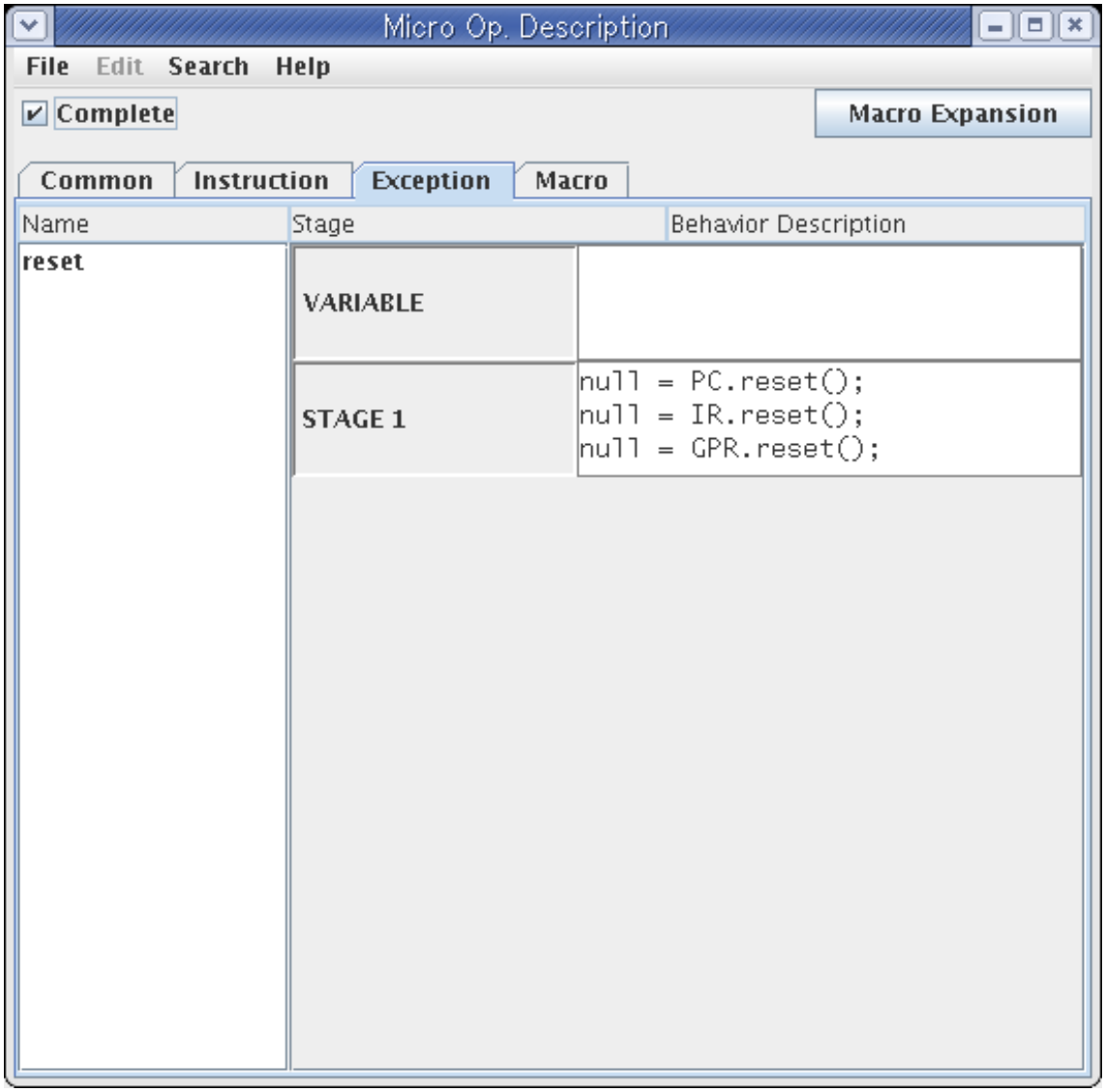

図 79 reset 割り込み記述後の画面

上図のようにリセット割り込み “reset” のマイクロ動作記述の記述してください。

以上でマイクロ動作の全ての記述が終了しました。記述を確定するために[Complete]チェックボックスをチェックし、[File]→[Close]でウインドウを閉じてください。

これで、入力すべき HDL 生成のための全てのデータが入力されました。

4.10. HDL 生成: HDL Generation

このステージでは機能検証・動作検証を行うシミュレーションモデル、および論理合成可能なモデルの HDL 記述を生成します。生成される HDL 言語は VHDL および Verilog を指定することができます。(ただし、Verilog での生成には別途ライセンスが必要です)

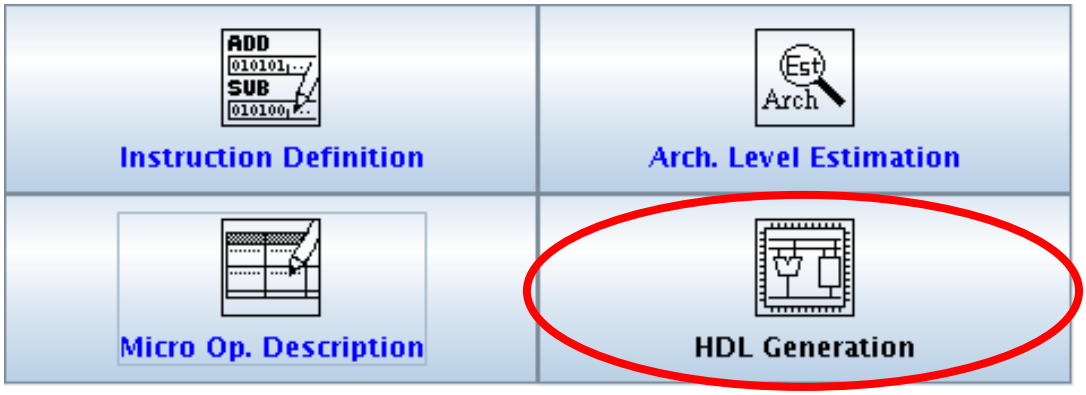

図 80 HDL Generation ボタン

メインウインドウから[HDL Generation]ボタンをおして、[Generation Confirm]ウインドウを開きます。

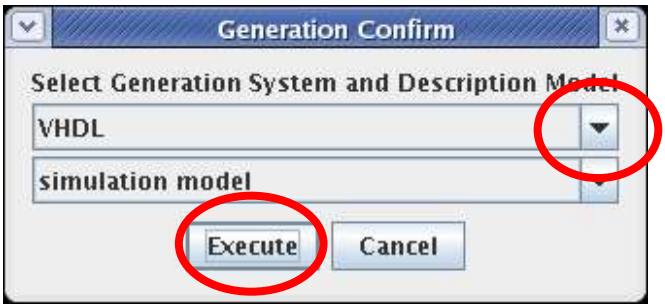

図 81 HDL 生成の開始

このウインドウで、生成言語と生成レベルを指定します。生成言語の指定は VHDL か Verilog かを指定します。生成レベルの指定はシミュレーションモデルの生成か、論理合成可能なモデルの生成か、あるいは両方のモデルの生成かを選択します。それぞれの選択は、▼のプルダウンメニューから選択します。メニューの内容は次のとおりです。

表 1 生成言語の指定

VHDL	VHDL 言語での生成
Verilog	Verilog 言語での生成(別途ライセンスが必要)

表 2 生成レベルの指定

simulation model	シミュレーションモデルのみ生成
synthesizable model	論理合成可能なモデルのみ生成
sim. model and syn. model	両方のモデルを生成

生成言語を指定し、生成レベルには両方のモデルを生成する[sym. model and syn. model]を選択し、[Execute]ボタンを押してください。すると、モデル生成プログラムが開始し、HDL 記述が生成されます。

プログラムの実行が終了すると、図のようなメッセージが表示されます。エラーがないことを確認し、[OK]ボタンを押すとメインウインドウに戻ります。

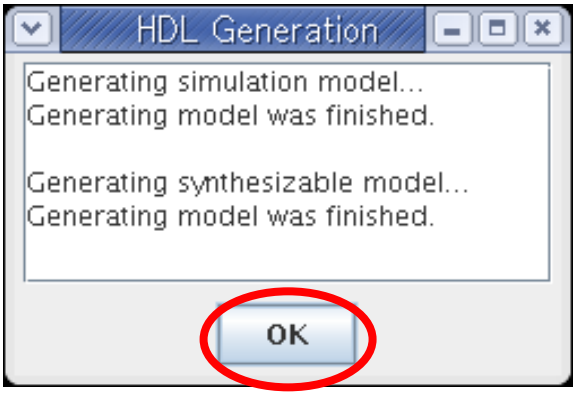

図 82 HDL 生成の実行結果

4.10.1. HDL 生成の確認

カレントディレクトリの下に"meister"というディレクトリが作られています。生成された HDL ファイルはモデルごとに別のディレクトリに格納されます。small_RISC の設計では、シミュレーションモデル用は meister/***lang***/small_RISC.sim に、論理合成可能なモデルは meister/***lang***/small_RISC.syn に保存されます。"***lang***"の部分には生成言語の名前が入り、vhdl か verilog になります。確認してください。

最後に、生成されたシミュレーションモデルの HDL ファイルの一覧と論理合成可能なモデルの HDL ファイルの一覧を示します。

図 83 シミュレーションモデル用の HDL ファイル

図 84 論理合成可能な HDL ファイル

以上で small_RISC の HDL 生成に関するチュートリアルは終了です。

続いて，ベースプロセッサ ***Brownie*** を拡張したプロセッサを使用し，[C Definition]と[Compiler Generation]についてのチュートリアルを行います．***Brownie*** は ASIP Meister Standard 環境でのみ使用することが出来ます。

4.11. C 記述の定義 ：C Definition (ASIP Meister Standard の環境が必要)

このステージでは、ベースプロセッサ ***Brownie*** に追加した拡張命令を C 記述でどのように表すかを定義します。C 記述での拡張命令を定義することで，コンパイラが拡張命令に対応したアセンブリコードを出力できます。

<***注意***> ベースプロセッサ ***Brownie*** の命令を削ってはいけません。

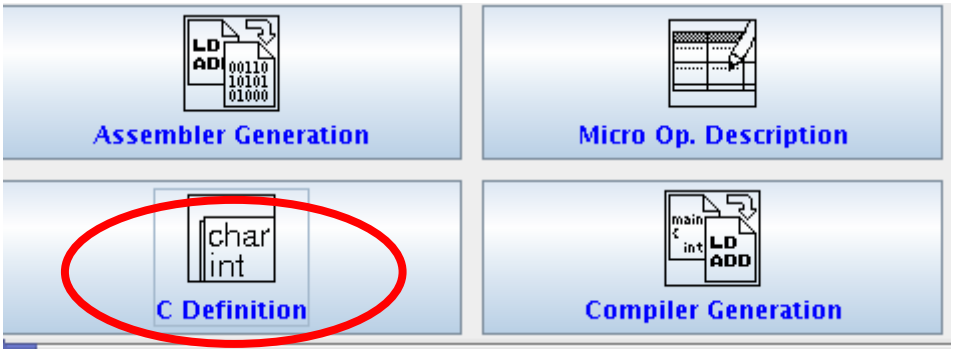

図 85 C Definition ボタン

初めに、***Brownie*** の設計ファイルを読み込み、前節までの方法で Brownie に追加命令を設計しておく必要があります。

メインメニューから[C Definition]ボタンを押すと、図 85 に示すウインドウが表示されます。

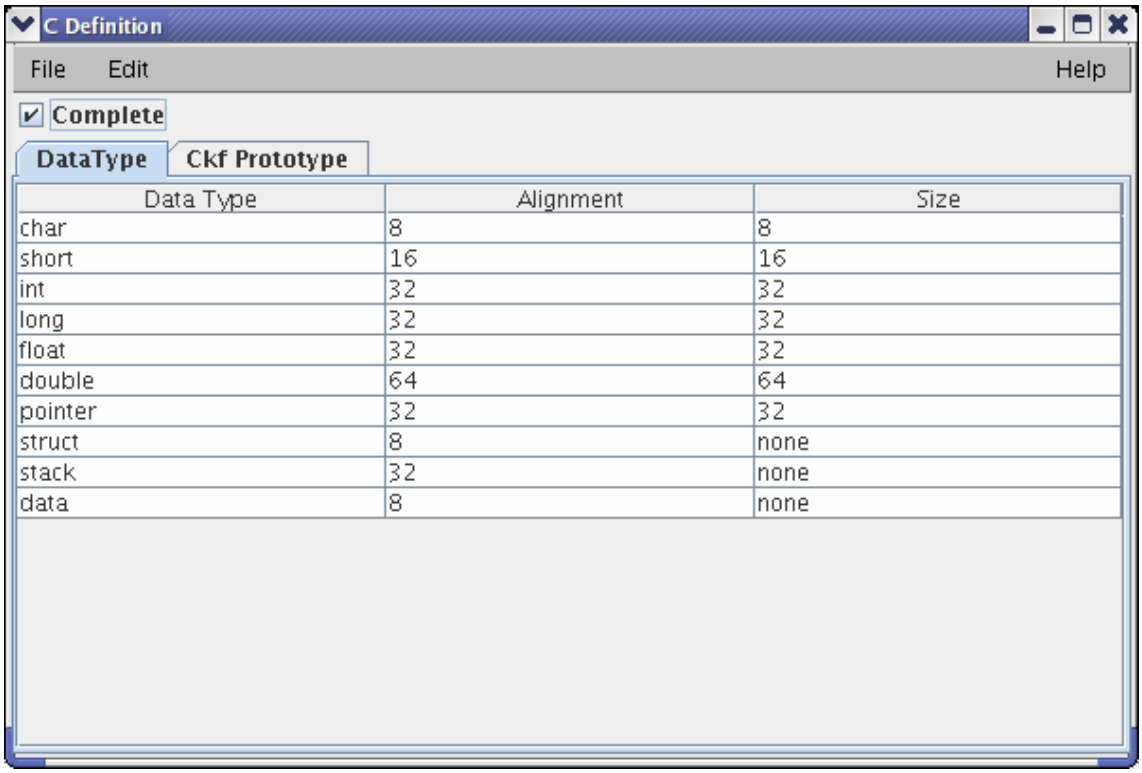

図 86 [C Definition]ウインドウ

[C Definition]は、以下の 2 つのタブから構成されます。

1. DataType タブ ：C 記述のデータタイプの定義
2. Ckf Prototype タブ ： 拡張命令の定義

現バージョンでは、DataType タブの内容は変更することができないため、DataType タブの内容に変更は加えません。次に、Ckf Prototype タブを選択します。Ckf Prototype タブを選択すると、図 87 に示すウインドウに変化します。

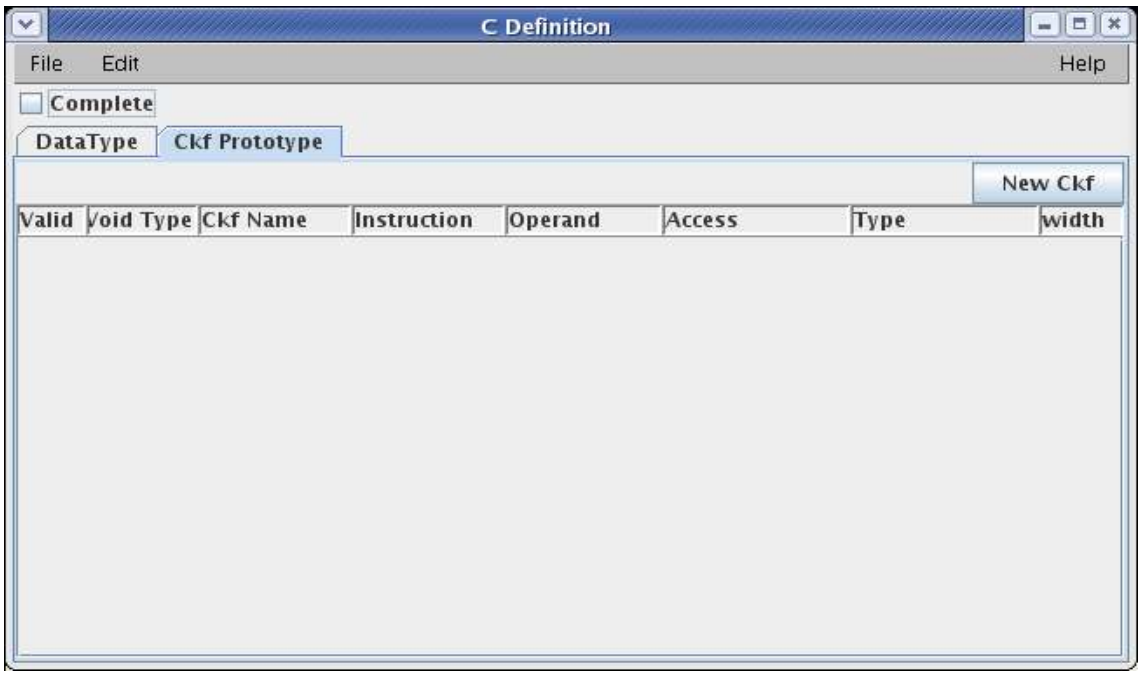

図 87 [Ckf Prototype]タブ

初期状態では、Ckf は登録されていません。[New Ckf]を選択することで、Ckf を登録することが出来ます。このチュートリアルでは、追加した命令として、Brownie が持たない”NXOR”命令を Ckf として登録します。”NXOR”命令の命令タイプとマイクロ動作記述はあらかじめ定義する必要があります。”NXOR”命令は 2 つのレジスタから値を読み込み、NXOR 演算（排他的論理和の否定の演算）を行った結果を 1 つのレジスタに書き込む命令です。最初に[New Ckf]を選択します。すると、図 88 に示す[New CKF]ウインドウが開きます。

図 88 [New CKF]ウインドウ

“NXOR”命令の Ckf 名を”NXOR”とします。OK ボタンを押すと，図 90 のように[Ckf

Prototype]ウインドウが変化します。
“NXOR”命令では、各項目をそれぞれ図 90 に示すように定義します。入力オペランドである rs1, rs2 については”Access”列を”Read”とし、出力オペランドである rd については”Access”列を”Write”とします。また、ビット幅を表す”Width”については、Brownie32 のレジスタ長に合わせて 32 とします。今回追加する命令では、出力オペランドが 1 つで、入力オペランドが 1 つ以上存在するために、”Void Type”についてはチェックを入れません。このように定義すると、C 記述で下記に示すような記述に対して、対応したアセンブリ記述を生成するようなコンパイラを生成できます。

<C 記述>

```
int A, B, C;
A = __builtin_brownie32_NXOR(B,C);
```

<**注意**> 拡張命令の前に”__builtin_brownie32_”をつける必要があります。

<変換後のアセンブリ記述>

```
NXOR B, C, A;
```

“Void Type”のチェックについては、以下の場合においては必ずチェックする必要があります。この時、出力オペランドに当たる引数は参照渡し引数として扱われます。また、引数の順番は、出力オペランドが先に来て、その後に入力オペランドが続きます。出力オペランド同士、入力オペランド同士では、”Operand”タブの上から順に登録されます。

- 出力オペランドが存在しない場合
  例：スタックポインタを暗黙的に参照し、スタックに値を PUSH する命令

```
PUSH        rs                          ;; rs の値をスタックへ
void __builtin_brownie32_PUSH(int rs);
```

- 出力オペランドが 2 つ以上存在する場合
  例：除算の商と剰余を別々のレジスタに同時に格納する命令

```
DIV         rs1, rs2, rd1, rd2          ;; rd1 = rs1 / rs2, rd2 = rs1 % rs2
void __builtin_brownie32_DIV(int *rd1, int *rd2, int rs1, int rs2);
```

- 出力オペランドが 1 つであるが、入力オペランドが存在しない場合
  例：スタックポインタを暗黙的に参照し、スタックの値を POP する命令

```
POP         rd                          ;;スタックの値を rd へ
```

void __builtin_brownie32_POP(int *rd);

例：ハードウェア乱数生成器から乱数を取得しレジスタに格納する命令

RND rd ;;乱数値を rd へ

void __builtin_brownie32_RND(int *rd);

[Ckf Prototype]ウインドウ中で編集できる各項目の詳しい説明に関しては、マニュアルを参照してください。

<補足> NXOR 命令のマイクロ動作記述

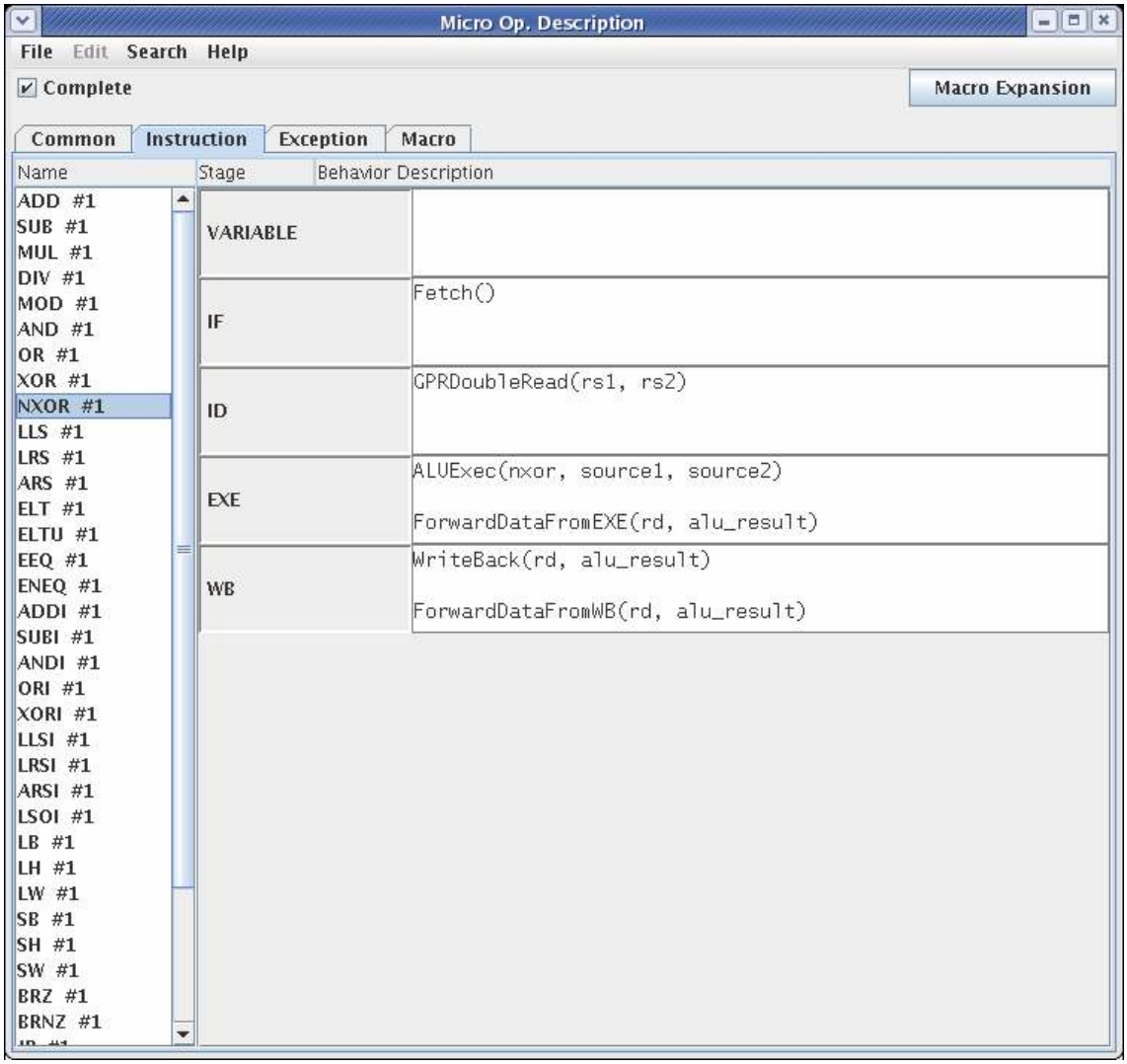

図 89 NXOR 命令のマイクロ動作記述

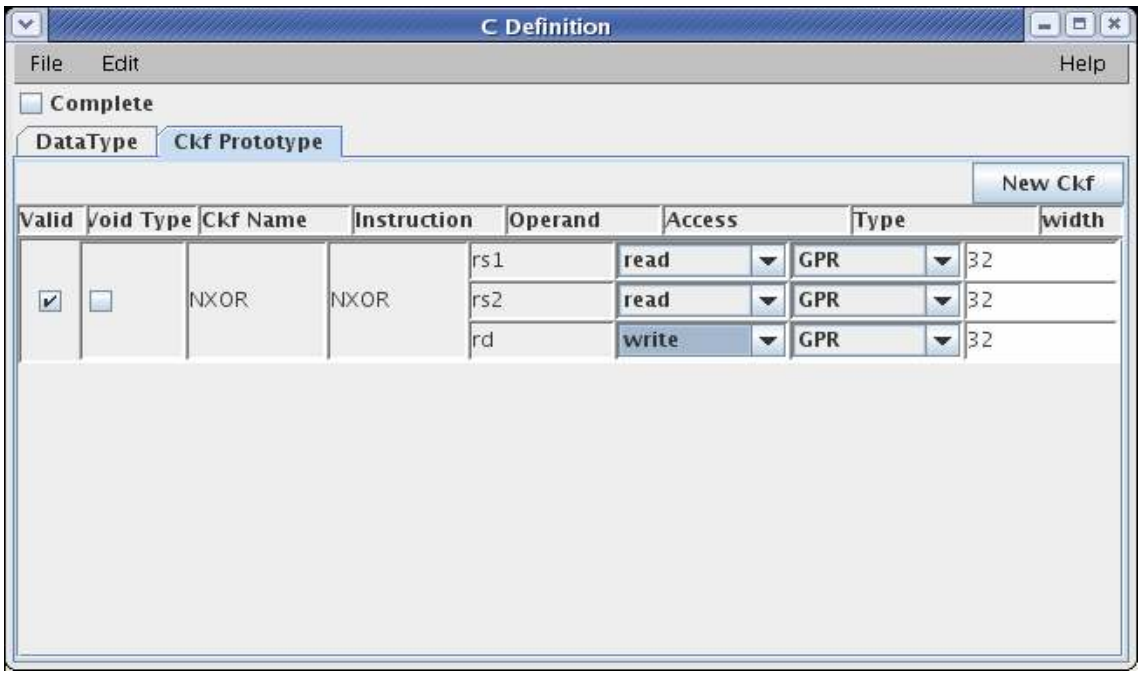

図 90 NXOR 命令登録後の[Ckf Prototype]ウインドウ

これで[C Definition]は終了です。Complete をチェックし、[C Definition]ウインドウを閉じてください。

4.12. コンパイラ拡張, Binutils 生成 : Compiler Generation (ASIP Meister Standard の環境が必要)

4.11[C Definition]で定義した Ckf に対応したコンパイラと Binutils を生成します。

図 91 Compiler Generation ボタン

まず図 92 に示す[Generation Confirm]ウインドウで[Input Description Generation]を

選択し”Execute”を選択します。

次に[GNU Tools Generation]を選択します。

<注意> [GNU Tools Generation]を選択後、十数分間時間がかかることがあります。

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。設計ファイル名が, **“Brownie.pdb”** であるため、Brownie.swgen の下にコンパイラ、Binutils のファイルが生成されます。

以上で作業は終了です。

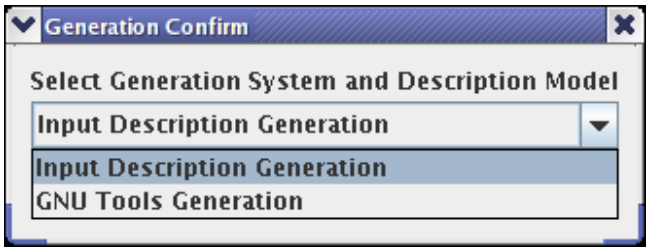

図 92 [Generation Confirm]ウインドウ

チュートリアルで説明されていない項目についてはマニュアルを参照してください。

プロセッサ small_RISC 仕様（付録）

2008 年 10 月

エイシップ・ソリューションズ株式会社

このドキュメントは、ASIP Meister のチュートリアル用に作成した、プロセッサ small_RISC の仕様書です。

A. 設計目標

small_RISC の設計目標は次の表に示すとおりとします。また、面積を優先して設計することにします。

面積	遅延	消費電力
45000 [gate]	20 [ns]	4000 [μW/MHz]

B. メモリのアクセス方式

small_RISC が想定している命令メモリ、およびデータメモリへのアクセス方式は次のとおりとなります。

	アクセス方式
命令メモリ	シングルサイクル
データメモリ	マルチサイクル

C. パイプライン構成

パイプラインの構成は 5 段とします。各パイプラインのステージの内容、および、遅延分岐、命令幅、データ幅に関しては次の表のとおりとします。

ステージ番号	ステージの動作内容
1	命令フェッチ
2	命令デコード/レジスタ読み出し
3	演算実行
4	メモリアクセス
5	レジスタ書き込み

遅延分岐	あり
遅延分岐のスロット数	1
命令幅	32 ビット
データ幅	32 ビット

D. 使用リソース

small_RISC は後に示すように、加算・減算命令、ロード・ストア命令、無条件・条件付分岐命令を持ちます。また、ロード・ストア命令や分岐命令では 16 ビットの即値オペランドをとり、32 ビットのレジスタ値と加算することになります。したがって、使用するリソースとして、算術論理演算器（ALU）の他に符号拡張用のエクステンダ（EXT）が必要となります。また、データを格納するレジスタや、メモリアクセスユニットも必要となります。次の表に FHM として選択するリソースを挙げます。

リソース	選択する FHM	リソースの名称	パラメータ
プログラムカウンタ	pcu	PC	増加ステップ幅 = 4 加算アルゴリズム = default Use as ＝ Prog. Counter
命令レジスタ	register	IR	ビット幅 = 32 ビット Use as ＝ Inst. Register
命令メモリ アクセスユニット	mifu	IMIFU	データのビット幅 = 32 ビット アドレス空間 = 32 ビット アクセス幅 ＝ 32 ビット アクセスモード ＝ シングルサイクル タイプ ＝ 読み込みのみ Use as ＝ Inst. Memory
データメモリ	mifu	DMIFU	データのビット幅 = 32 ビット

アクセスユニット			アドレス空間 = 32 ビット アクセス幅 = 8 ビット アクセスモード = マルチサイクル タイプ = 読み書き可能 Use as = Data Memory
レジスタファイル	registerfile	GPR	レジスタ本数 = 32 読み出しポート数 = 2 書き込みポート数 = 1 Use as = Register File
演算ユニット	alu	ALU	加算アルゴリズム = default Use as = Unspecified
符号拡張ユニット	extender	EXT	入力データ幅 = 16 ビット 出力データ幅 = 32 ビット Use as = Unspecified

E. ストレージの定義

先ほど宣言したリソースがプロセッサ内部でどのようなレジスタやメモリとしてみなされるのかということを設定します。

E.1. レジスタファイル・ストレージの定義

プロセッサが使用する汎用レジスタを定義します。レジスタファイルの展開に関する設定を行った後、展開した個々のレジスタの使用用途を設定します。

E.1.1. レジスタファイルの展開に関する設定

ストレージ名	GPR[asc]
リソース	GPC
ビット幅	32
用途	Register
Location	Original
Binary	[bin-asc]

E.1.2. 展開した個々のレジスタの使用用途

GPR0	zero register
GPR28	Stack pointer

GPR29	frame pointer
GPR30	link register
GPR31	return register

E.2. レジスタ・ストレージの定義

汎用レジスタ以外のレジスタとして、命令レジスタとプログラムカウンタを定義します。

E.2.1. 命令レジスタの定義

ストレージ名	IR
リソース	IR
ビット幅	32
用途	instruction register
Location	Original

E.2.2. プログラムカウンタの定義

ストレージ名	PC
リソース	PC
ビット幅	32
用途	program counter
Location	Original

E.3. メモリ・ストレージの定義

外部のメモリとして命令メモリとデータメモリを定義します。

E.3.1. 命令メモリ

ストレージ名	IMIFU
リソース	IMIFU
ビット幅	32
用途	instruction memory
アクセス幅	32

E.3.2. データメモリ

ストレージ名	DMIFU
リソース	DMIFU
ビット幅	32
用途	Data memory
アクセス幅	8

F. 入出力ポート

設計するプロセッサコアには、クロック信号とリセット信号やその他のメモリアクセスのための信号を供給する必要があります。次の表にプロセッサに接続する信号を示します。

信号名	信号の向き	ビット幅	信号の説明
CLK	In	1	クロック信号
Reset	In	1	リセット信号
InstAB	out	32	命令メモリアドレスバス
InstDB	In	32	命令メモリデータバス
DataAB	out	32	データメモリアドレスバス
DataDB_in	In	32	データメモリデータ入力バス
DataDB_out	out	32	データメモリデータ出力バス
DataReq	out	1	データメモリ request 信号
DataAck	in	1	データメモリ acknowledge 信号
DataRW	out	1	データメモリ リード・ライト信号
DataMode	out	2	データメモリライトモード信号
DataExt	out	1	データメモリ符号拡張信号
DataAddrError	in	1	データメモリアドレスエラー信号
DataCancel	out	1	データメモリキャンセル信号

G. 命令セット

本節では small_RISC の 4 種類の命令タイプを示した後、おのおのの命令の概要と命令コードの一覧を示します。

G.1. 命令形式

命令形式は次の 4 種類があります。

1. レジスタ-レジスタ形式（R_R 形式）

2. レジスタ-即値形式（R_I 形式）
3. 条件分岐形式（B 形式）
4. 無条件分岐形式（JP_T 形式）

G.1.1. レジスタ-レジスタ形式（R_R 形式）

レジスタ-レジスタ形式では、2 つのレジスタからデータを読み出して処理を行い、その結果を他のレジスタに書き込む命令で使用します。加算・減算命令で使用する命令タイプです。

31 26	25 21	20 16	15 11	10 0
000000	rs0	rs1	rd	func
オペコード	入力レジスタ 0	入力レジスタ 1	出力レジスタ	命令のオペコード

G.1.2. レジスタ-即値形式（R_I 形式）

レジスタ-即値形式では、1 つのレジスタ、1 つの即値からデータを読み出して処理を行い、その結果を他のレジスタに書き込む命令で使用します。ロード、ストアといったメモリアクセス命令で使用するタイプです。

31 26	25 21	20 16	15 0
op	rs0	rd	const
オペコード	入力レジスタ 0	出力レジスタ	オフセット

G.1.3. 条件分岐形式（B 形式）

条件分岐形式は、1 つのレジスタからデータを読み出して 0 との比較を行い、レジスタの値が 0 であれば（このとき、条件は真となる）、分岐する命令で使用します。分岐先アドレスは PC の値と与えられた 16 ビットの即値の和で表されます。

31 26	25 21	20 16	15 0
op	rs0	00000	const
オペコード	入力レジスタ	予約	オフセット

G.1.4. 無条件分岐形式（JP_T 形式）

無条件分岐形式は、無条件に分岐する命令で使用されます。分岐先アドレスは PC の値と与えられた 16 ビットの即値の和です。

31 26	25 16	15 0

op	0000000000	const
オペコード	予約	オフセット

G.2. 命令概要

G.2.1. ADD 命令

命令形式

レジスタ-レジスタ形式（R_R 形式）

命令フィールド

31 26	25 21	20 16	15 11	10 0
000000	rs0	rs1	rd	00000100000
オペコード	入力レジスタ 0	入力レジスタ 1	出力レジスタ	命令のオペコード

命令フォーマット

```
ADD rd rs0 rs1
```

命令の機能

（レジスタ rs0 の内容） + （レジスタ rs1 の内容）の計算結果をレジスタ rd に格納します。

```
rd = rs0 + rs1
```

G.2.2. SUB 命令

命令形式

レジスタ-レジスタ形式（R_R 形式）

命令フィールド

31 26	25 21	20 16	15 11	10 0
000000	rs0	rs1	rd	00000100010
オペコード	入力レジスタ 0	入力レジスタ 1	出力レジスタ	命令固有のオペコード

命令フォーマット

```
SUB rd rs0 rs1
```

命令の機能

（レジスタ rs0 の内容） – （レジスタ rs1 の内容）の計算結果をレジスタ rd に格納し

ます。

```
rd = rs0 - rs1
```

G.2.3. LOAD 命令

命令形式

レジスタ-即値形式（R_I 形式）

命令フィールド

31 26	25 21	20 16	15 0
100000	rs0	rd	const
オペコード	入力レジスタ 0	出力レジスタ	オフセット

命令フォーマット

```
LOAD rd rs0 const
```

命令の機能

（レジスタ rs0 の内容）と const との和で示されるメモリアドレスの内容（32 ビット分）をレジスタ rd に書き込みます。

```
rd = [rs0 + const]
```

G.2.4. STORE 命令

命令形式

レジスタ-即値形式（R_I 形式）

命令フィールド

31 26	25 21	20 16	15 0
101001	rs0	rd	const
オペコード	入力レジスタ 0	出力レジスタ	オフセット

命令フォーマット

```
STORE rd rs0 const
```

命令の機能

（レジスタ rd の内容）を（レジスタ rs0 の内容）と const との和で示されるメモリアドレスに 32 ビット分書き込みます。

```
[rs0 + const] = rd
```

G.2.5. BEZ 命令

命令形式

条件分岐形式（B 形式）

命令フィールド

31 26	25 21	20 16	15 0
000100	rs0	00000	const
オペコード	入力レジスタ	予約	オフセット

命令フォーマット

```
BEZ rs0 const
```

命令の機能

レジスタ rs0 の内容が 0 であれば、プログラムカウンタの値に const の値を加算します。

```
if (rs0 == 0){
    PC = PC + const
}
```

G.2.6. J 命令

命令形式

無条件分岐形式（JP_T 形式）

命令フィールド

31 26	25 16	15 0
000010	0000000000	const
オペコード	予約	オフセット

命令フォーマット

```
J const
```

命令の機能

無条件でプログラムカウンタの値に const の値を加算します。

```
PC = PC + const
```

H. 例外・割り込み

プロセッサ small_RISC では、リセット割り込みを備えています。割り込みの実装内容は次に示すとおりとします。

発生条件	reset ポートから信号”1”が入力されることで発生
リセット処理	プログラムカウンタ、命令レジスタ、汎用レジスタファイルをそれぞれリセット
処理サイクル数	１サイクル