ASIPSolutions

ASIP Meister Users' Manual

Version 2.3
July 2008

ASIP Solutions, Inc.

sanko-osaka-honmachi Bldg. 6F, 2-3-8 Honmachi, Chuo-ku, Osaka, 541-0053 Japan
ASIP Solutions, Inc.
support@asip-solutions.com

Table of Contents

List of Figures

Introduction

1. ASIP Meister Manual Structure

Table 1 shows the documents list of **ASIP Meister**. The table describes the name of each document, its contents and its path from **ASIP Meister** directory. If you follow the "Install Guide", **ASIP Meister** would be installed under "/usr/local/ASIPmeister/" and the PDF file of Tutorial for example would be located at,
"/usr/local/ASIPmeister/share/tutorial/AM_Tutorial_en.pdf"
The documents shown provide supplementary explanation to this manual. So please refer to them as necessary.

Table 1 ASIP Meister **Documents List**

Document & File name	*Description*
ASIP Meister User Manual share/AM_usersmanual_en.pdf	This manual
Install Guide InstallGuide.txt	This guide explains the operation environment (HW/SW platform) required to run **ASIP Meister**, how to install and setup **ASIP Meister**.
ASIP Meister Tutorial share/tutorial/AM_tutorial_en.pdf	This tutorial document provides [small_RISC Processor] and compiler generation process tutorials to help you familiarize yourself with **ASIP Meister**.
FLEXnet License User Guide share/LicensingEndUserGuide.pdf	This Guide explains License server setting operation. For all details related to licensing, please refer to this guide.

2. This Manual Usage and Content

This manual is for beginner users who want to use **ASIP Meister**, and users who would like to confirm the specific functionality of **ASIP Meister**. This manual consists of 3 main sections【**ASIP Meister** Outline】【**ASIP Meister** Operation】 and 【Appendix】. Please refer to each accordingly.

3. Manual Conventions

This manual has the following conventions;

- User interfaces such as menus, buttons, text boxes, parameters, and checkboxes are enclosed within the brackets [ ].
- $ symbol means the command prompt.
- The name of files and directories are enclosed with double quotation marks " ".
- ＜Note＞ refers to notes or restriction items to consider, so please read carefully.
- ＜Related＞ lists up related explanation references.

ASIPSolutions

I. ASIP Meister Outline

1. What is ASIP Meister

ASIP Meister is a tool for developing Application Specific Instruction Set Processors (ASIPs) from high level specification description. The functionality of the tool is listed below,

- Automatic generation of the processor HDL description from Micro Op. description.
- Fast Estimation of processor design quality at an early stage of design process.

For the above functionality, it is possible to examine and compare different architecture implementations using **ASIP Meister**.

The input/output of **ASIP Meister** is roughly shown in the figure below. Using the GUI of **ASIP Meister**, the user inputs the design parameters and Micro Op. description. **ASIP Meister** generates an estimation report of the processor design quality and its RTL description files. Furthermore, **ASIP Meister** also generates a development environment composed of a C compiler, assembler and linker. The synthesis model and the simulation model are automatically generated in HDL. These models then become the input to logical synthesis tools and functional simulation tools.

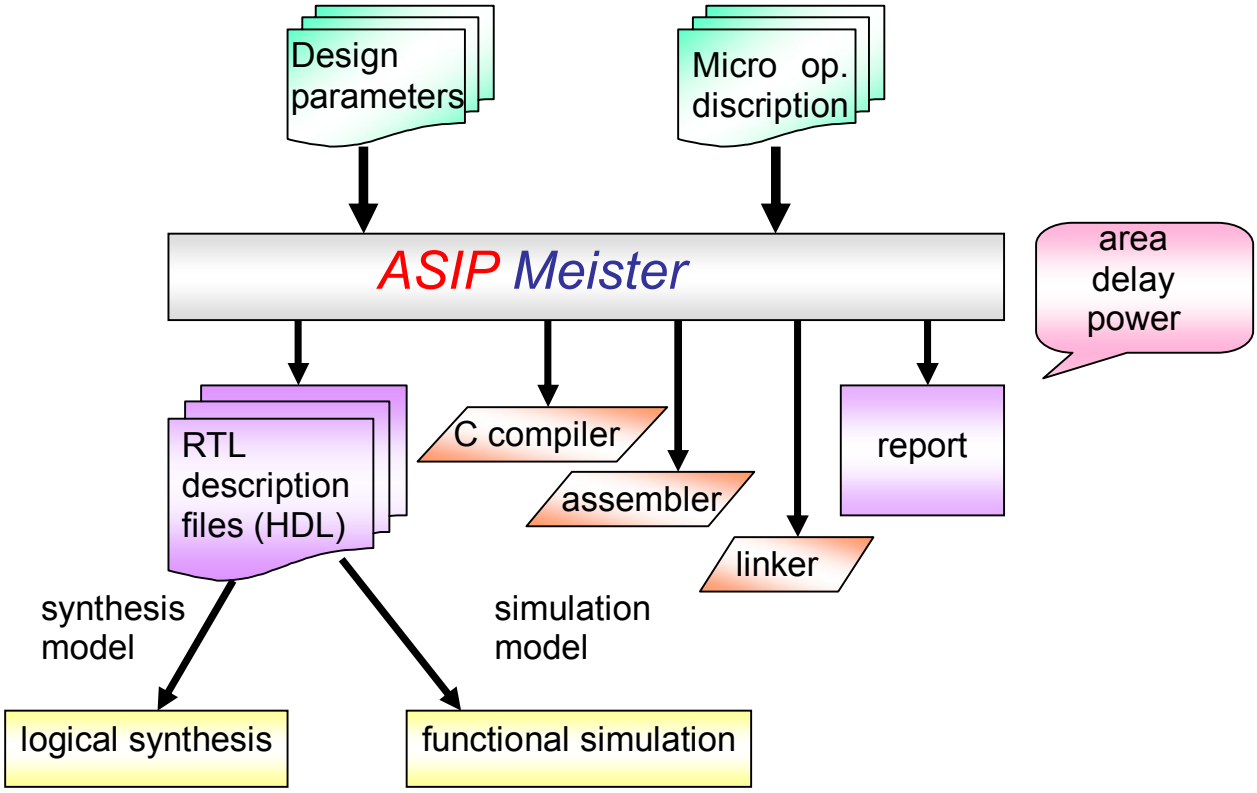


Figure 1 ASIP Meister **Input and Output**

2. Design Data Input Flow

When **ASIP Meister** starts, the window (shown in Figure 2) opens.

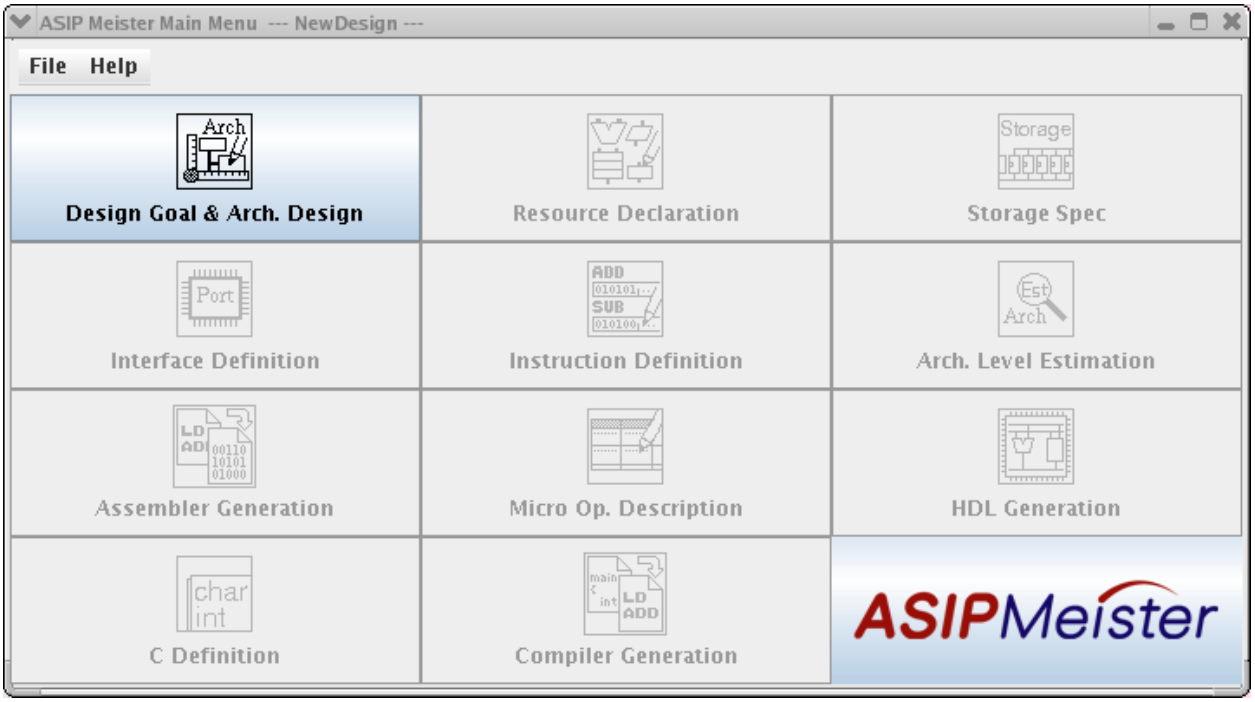


Figure 2 ASIP Meister **Main Window**

In **ASIP Meister**, to design a processor core, eleven design steps should be followed;

[Design Goal & Arch. Design]:	Processor Spec. Settings
[Resource Declaration]:	Declaration of used Resources (Computational, Registers, etc.)
[Storage Spec]:	Storage Specification
[Interface Definition]:	Processor Interface Design
[Instruction Definition]:	Instruction Type and Format Definition
[Arch. Level Estimation]:	Processor Design Performance Estimation
[Assembler Generation]:	Assembler File Generation
[Micro Op. Description]:	Instruction Micro Op. Design
[HDL Generation]:	HDL Description Generation
[C Definition]:	Definition of C code for extended instructions that are not supported by the compiler.
[Compiler Generation]:	Generation of compiler and Binutils

Each design step is represented by a rectangular button in the Main window. When clicking on this button a sub-window opens and the processor design step of this part can be specified.

Since there is dependency between design steps, it is necessary to consider the order among design steps as shown below.

Input to [Resource Declaration], [Instruction Definition] is possible after [Design Goal & Arch Design]

Input to [Arch. level Estimation], [Assembler Generation], [C Definition] is possible after [Design Goal & Arch Design], [Resource Declaration], [Interface Definition], [Instruction Definition]

Input to [Micro Op. Description] is possible after [Arch. Level Estimation]
Execution of [HDL Generation] is possible after [Micro Op. Description].
Execution of [Compiler Generation] is possible after [C Definition]

II. ASIP Meister Operation

1. Basic Operation

This chapter explains the basic operation of **ASIP Meister**.

1.1 ASIP Meister Startup

When **ASIP Meister** is used, the proper software license is needed. Please install the license file or start the license server beforehand in accordance with the installation guide. Verify also that the path to the license file or the license server name is properly set in LM_LICENSE_FILE environment variable. As for the starting method of the license server, please refer to the installation guide. In case you want to use the compiler generation function, the directory specified for the application program development generation tool at installation time should also be specified in the ASIP_APDEV_SRCROOT environement variable.

```
$ export LM_LICENSE_FILE=/usr/local/ASIPmeister/ASIPmesiter.lic
$ export ASIP_APDEV_SRCROOT=/home/xxx/ASIP_APs(The path of application
  program development generation tool)
```

ASIP Meister requires Java™ 2 environment to be installed first to run. Please refer to Java™ home page and follow the normal procedure to install Java™ 2 environment on your machine. When you start **ASIP Meister**, the path to **ASIP Meister** and the Java™2 environment needs to be specified. The sample file (“ASIPmeister.setup.sample”) is provided at the ”/usr/local/ASIPmeister/share/” directory for reference. This sample setup file assumes that your Java™ 2 running environment is installed at ”/usr/java/jre1.5.0_05/” directory. If it is installed in a different directory, please modify the settings according to your environment.

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

The sample file is written for bash shell. Therefore, if you are using csh or tcsh, please change your settings using setenv as below;

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

Figure 3 Settings Using `setenv`

The **ASIP Meister** start command is “ASIPmeister”. As in the following example, you can specify the input data file as an argument as well.

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

＜Note＞ ”.pdb” is the extension of the data file of **ASIP Meister**.

Default design data including the definition of the memory port

In **ASIP Meister**, it is necessary to define the connection ports of the processor with the instruction and data memory. The ports include standard ports, i.e the address bus and data

bus, there are also special ports for access memory method and control. In **ASIP Meister** "share/basefile" directory, predesigned files are prepared for the following memory structures.

Design File Name	Instruction Memory Access	Data Memory Access
InstSingle-DataSingle.pdb	Single Cycle Access	Single Cycle Access
InstSingle-DataMulti.pdb	Single Cycle Access	Multi-Cycle Access
InstMulti-DataSingle.pdb	Multi-Cycle Access	Single Cycle Access
InstMulti-DataMulti.pdb	Multi-Cycle Access	Multi-Cycle Access

1.2 Main Window

When you start **ASIP Meister**, [**ASIP Meister** Main Menu] window (Figure 4) opens.

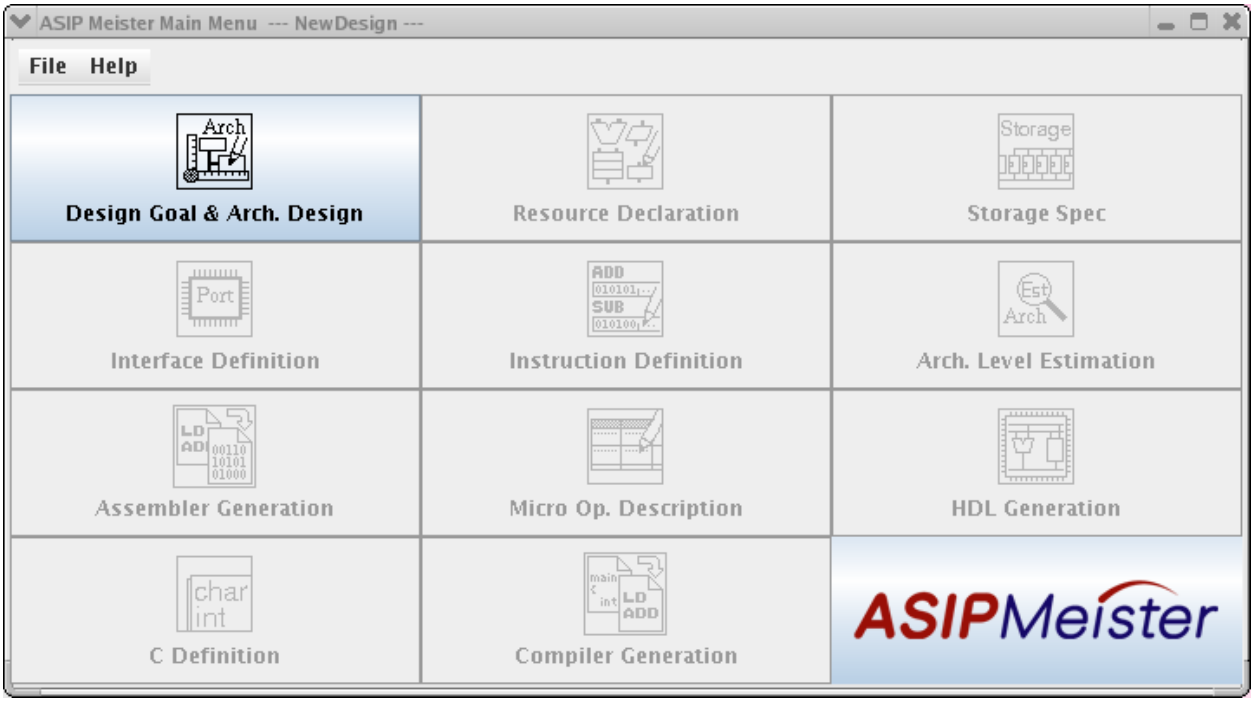


Figure 4 [ASIP Meister Main Menu] Window

In this window, all processor design steps are listed. The input to each design step would be explained next starting from 2. [Design Goal & Arch. Design].

The title bar of the main menu window displays the data file currently under design. For example, when you create a new design, "---New Design---" is displayed on the title bar.
The details of menus of [ASIP Meister Main Menu] window are described next.

(1) [File] Menu

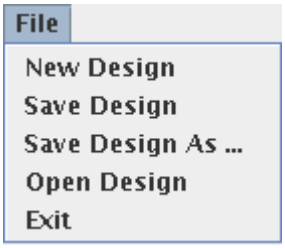


Figure 5 Main Window > File Menu

[New Design]
Functionality: Starts a new design.

[Save Design]
Functionality: Saves the data of a currently opened design with the same name.

[Save Design As …]
Functionality: Saves the data of a currently opened design under a different name.
When selecting this item, [Save] window opens. First, click [Save in] pull down menu to select the directory to save the data file. Then, enter the data file name to [File name] field and click [Save] button to save the data file. The extension ".pdb" is automatically added to the data file name.

<note> Using **[Save Design]** or **[Save Design As …]** when a sub-window is open does not save the settings specified inside the sub-window. First, from the sub-window [File] menu select [Commit], then select **[Save Design]** or **[Save Design As …]** in order to save all your settings.

[Open Design]
Functionality: Loads a saved design data.
If you click this menu command, the window to select the input data file opens. Select the data file to enter and click [Open] button.

[Exit]
Functionality: Exits **ASIP Meister**.

(2) [Help] Menu

Figure 6 Main Window > Help Menu

[Version]
Functionality: Displays version information of **ASIP Meister**.

Table 2 Menu List of [ASIP Meister Main Menu] Window

Menu	Sub-Menu	Functionality
File	New Design	Starts a new design
	Save Design	Saves Design data using Same name
	Save Design As …	Saves Design Data using different name

Menu	Sub-Menu	Functionality
	Open Design	Opens a Saved Design Data
	Exit	Exits **ASIP Meister**
Help	Version	Displays Version Information

1.3 Sub-Windows

To open the sub-window of each item in [Main Menu] window, simply click the item in the window. The detail of how to set and enter the value for each sub-window is explained starting from chapter 2. (Design Goal & Arch. Design). To close the sub-window, click [Close] from [File] menu. In order to save your settings as you progress, choose [Commit] submenu from [File] menu, then choose either [Save Design] or [Save Design As …]. If you have completed setting all the options in the sub-window, click [Complete] checkbox (Figure 7) before closing it.

Checking [Complete] checkbox implies you have completed setting the options in the sub-window. Then you can proceed and set options in another sub-window. If you close the sub-window without checking [Complete] checkbox, you cannot proceed to next sub-window.

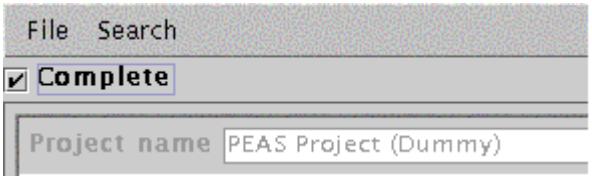


Figure 7 Sub-Window [Complete] Button

: Not Completed

: Completed

: Not Completed (READ ONLY)

: Completed (DISPLAY ONLY)

Figure 8 Variations of [Complete] Button

1.4 Working Directory

ASIP Meister stores its working files, auto generated HDL description files and the assembler assemble rule description file to the "meister" directory. This working directory exists in the current directory. If it does not exist, **ASIP Meister** creates the directory automatically.

＜Related＞10. HDL Generation

1.5 Compiler Directory

In **ASIP Meister** Standard edition, the compiler or the Binutils can be generated. However, before generation, you can designate a target directory. Setting the environment variable ASIP_GNU_INSTALL_DIR with the target directory path will cause the compiler and the Binutils to be generated in the designated directory. If no directory is designated, the files will be generated automatically under the working directory "meister/swgen".

2. Design Goal & Arch. Design

When clicking [Design Goal &Arch. Design] in Main Window, the sub-window in Figure 9 opens. Within this sub-window, you can set the processor type and architecture parameters.

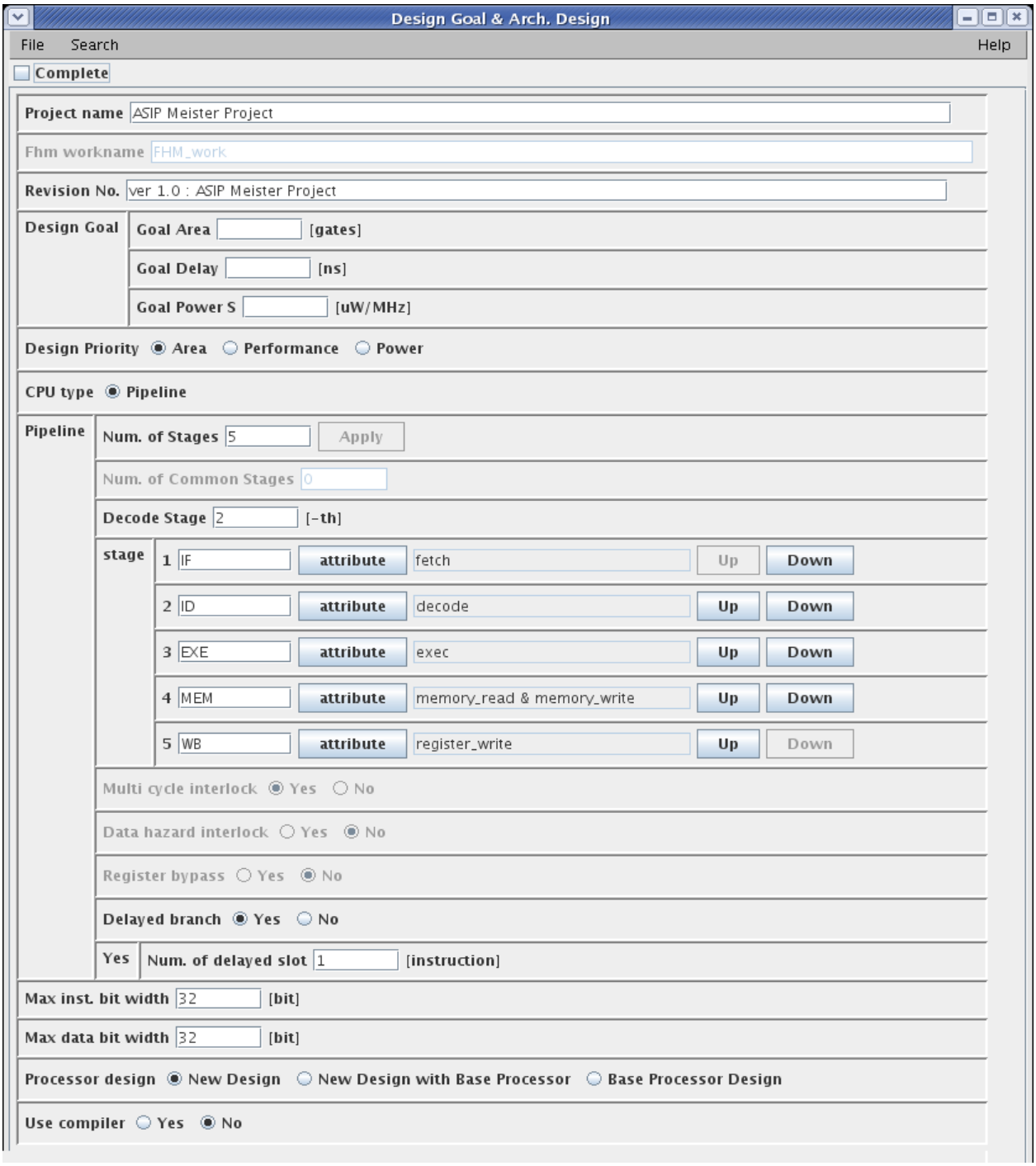


Figure 9 [Design Goal & Arch. Design] Window

Mainly the outline of the processor core that will be designed can be determined, by specifying the processor design data management information (Project name, FHM database, revision information), design goals (area, performance, power consumption), processor type, pipeline

structure (stage number, stage functionality), instruction and data bit width, processor design course and compiler usage.

The [Design Goal] including ([Goal Area], [Goal Delay], [Goal Power S]) in the [Design Goal & Arch. Design] window will be compared later against the actual estimates of the designed processor core in [Arch. Level Estimation] design step.

＜Reference＞7. Arch. Level Estimation

[Project name]: Project Name

Enter the project name for the current design data to this field. You can use any characters for this field. So use a project name that is most convenient for your design.

＜Note＞The project name entered to this field is not the entity name for the processor. You can specify the entity name for the processor in [Interface Definition] window. (for details refer to chapter 5. Interface Definition)

[Fhm workname]: Resource Database name

This field shows the database name of the FHM resource used by **ASIP Meister**.

＜Note＞ In the current version, "FHM_work" is the usually offered and the only workname enabled.

[Revision No.]: Revision number

Enter the version information of the design data to this field. You can use any characters for this field. Use this field to manage the version information of the design.

[Design Goal]: Target value of the design

Enter the target design qualities of the processor to these fields. Enter the target area of the processor, the target delay of the data path (combination circuit) and the target static power consumption to [Goal Area], [Goal Delay] and [Goal Power S] fields respectively. Architecture-level estimation engine in **ASIP Meister** will use these values. (for details refer to chapter 7. Arch. Level Estimation)

[Design Priority]: Priority of the design target

Select which option ([Area], [Performance] or [Power]) to prioritize in the design of the target processor. If you click [Area] radio button, the area of the processor will have the highest priority when architecture-level estimation engine estimates the design quality. Likewise, if you click either [Performance] or [Power] radio button, the priority of the estimation changes accordingly.

[CPU type]: Processor type to design

This field shows the architecture type of the processor to design.
In this version, **ASIP Meister** only supports pipeline architecture.

[Num. of Stages]: number of pipeline stages

Enter the number of the pipeline stages to this field. Click [Apply] button. The number of the rows that appear in [stage] field corresponds to the value entered in [Num. of Stages] field.

＜Note＞Rows representing stage name TMP are added when you increase the number of stages. When you decrease the number of stages, rows will be deleted from the bottom up. After you have reduced the number of stages, from [File] menu, choose [Commit] menu, and a message window like the one in Figure 10 will appear.

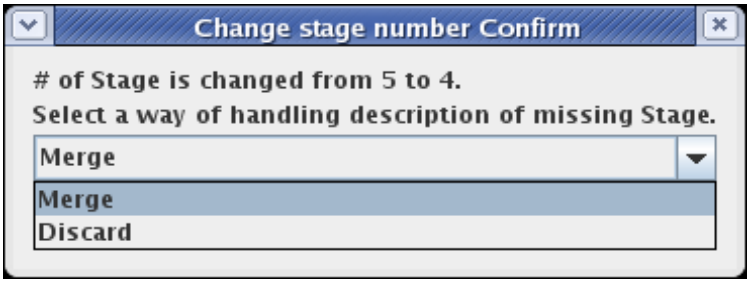


Figure 10 [Change stage number Confirm] Window

In the window shown, you can decide how to deal with the Micro Op. description of the window that will be eliminated. After you choose "Merge", choosing "Select" will merge the Micro Op. description of the deleted stage to the present last stage. However, if you choose "Discard", choosing "Select" will discard the Micro Op. description of the deleted stage.

<Note> After changing the number of stages, until you choose [Commit] from the [File] menu, you cannot change the order of the stages using "Up" and "Down" buttons in the [Design Goal & Arch. Design] Window.

[Num. of Common Stages]: Number of common stages in the pipeline

This field shows the number of common stages.
Common stage means the same stage that is used by the Micro Op. of all instructions. Here, the number of common stages from stage 1 is input. When stage 1 is not a commmon stage, the num. of common stages becomes 0. In this version, the value is always "0".

[Decode Stage]: Decoding stage of the pipeline

Enter the stage number, which executes the instruction decoding to this field.

[Stage]: Stage information

These fields have rows to enter the parameter for each pipeline stage. The number of the row displayed corresponds to the value entered in [Num. of Stages] field. Each row displays the stage number, the stage name, and its attributes.
You can set the stage attribute with the attribute selection window (Figure 11). That window opens by clicking [attribute] button. Click the checkbox of the attribute to use for the stage and click [OK] button to close the window. The attributes setting appears in stage specification field. You can set multiple attributes to each stage. You can set the following seven attribute types;

- [fetch]: Instruction fetch stage
- [decode]: Instruction decoding stage
- [register_read]: Register reading stage
- [exec]: Operation execution stage
- [memory_read]: Memory reading stage
- [memory_write]: Memory writing stage
- [register_write]: Register writing stage

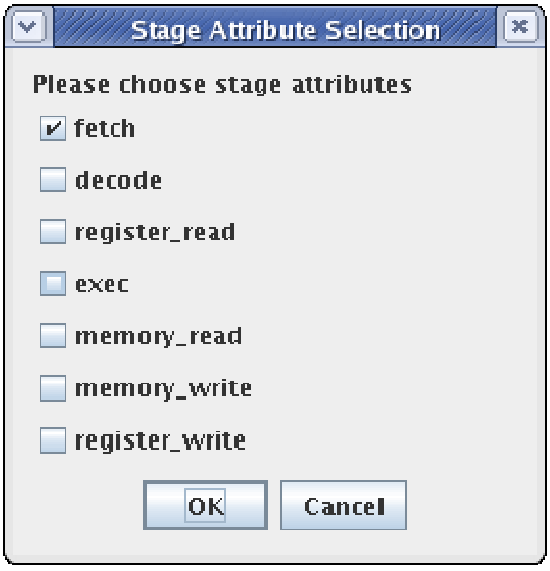


Figure 11 Pipeline Stage Attribute Selection Window

Furthermore, Using "Up" or "Down" buttons, the stage name and the attribute will swap with the upper stage or with the lower stage respectively.

<note> In the case of performing stage swap using the "Up" and "Down" buttons, until you choose [Commit] menu from [File] menu, the number of stages cannot be changed.

[Multi cycle interlock]: Interlock setting for multi cycle operation

This field shows the interlock setting for multi cycle operation.
In this version, you cannot modify this value. When multi cycle operation is detected, it is possible to select to stall the succeeding instructions or not. When stalling the succeeding instructions, please select "Yes".

＜Note＞ In the Current version, "Yes" is always specified.

[Data hazard interlock]: Interlock setting for data hazard

This field shows the interlock setting for data hazard. In this version, you cannot modify the value. When data hazard is detected, it is possible to select to stall the succeeding instructions or not. When stalling the succeeding instructions, please select "Yes".

＜Note＞ In the Current version, "No" is always specified.

[Register bypass]: Register bypass setting

This field shows the register bypass (forward) setting. In this version, you cannot modify the value. "yes" means that register bypass is enabled.

＜Note＞ In Current version, "No" is always specified.

[Delayed branch]: Delay branch setting

Click [Yes] radio button, if you wish to use the delay branch.

[Num. of delayed slot]: Number of delay branch slots

Enter the number of delayed slots to this field. This field becomes available only when you have selected [Yes] radio button at [Delayed Branch] field. When the Num. of delayed slot is zero, is the same as if there is no delayed branch.

[Max inst. bit width]: Maximum instruction bit width

Enter the maximum bit width of the instruction to this field.

[Max data bit width]: Maximum data bit width

Enter the maximum bit width of data to this field.

[Processor design]: Processor Design Course

In the processor design course, you can select 1 out of 3 entries as follows.

1. **[New Design]:** A new Design course of a processor.
2. **[New Design with Base Processor]:** A design course that takes the Base processor as base and extends it. In case you choose **2.[New Design with Base Processor]**, choose to have Base processor design file open.
3. **[Base Processor Design]:** A new Design course of a base processor.

<Note 1> The difference between **1. [New Design]** and **3.[Base Processor Design]** is that the instructions of a processor designed with **1. [New Design]** can be modified or new instructions can be added, however, the instructions of a processor designed with **2.[New Design with Base Processor]** can not be modified, and only new instructions can be added. In case that the instructions were modified, the design data file is not allowed to be overwritten.

<Note 2> According to the processor design course flow, the constraints in the Phase will change. Each constraint will be explained in the Phase of the target.

<Note 3> During the processor design course, if you want to change from **3.[Base Processor Design]** to **2.[New Design with Base Processor]**, a window confirming whether you want to register the current design file as base processor will apprear. If you click [Yes], the file will be registered as base processor. In the case you want change from **2.[New Design with Base Processor]** to **1.[New Design]**, a window confirming if you want to discard the base processor information will appear.

[Use compiler]: Compiler Usage

If "Yes" is selected, base processor ***Brownie*** is will be used as base, and the compiler that will be generated will support instruction extended instruction set. On the other hand, if "No" is selected, it will assume that no compiler will be used and both [C Definition] and [Compiler Generation] phases cannot be used.

<.Note> In the case you chose [New Design with Base Processor] when selecting the [Processor Design], when selecting [Use Compiler] options, if you change your selection from "no" to "yes" a message box will apprear asking whether you have modified the base processor instruction set or not yet modified.

Table 3 Items list of [Design Goal & Arch. Design] Window

Item	Description
Project name	Design Project Name
Fhm workname	Used Resources Database Name
Revision No.	Design Data Version Information
Goal Area	Processor Area Unit [gates]
Goal Delay	Maximum Delay Unit [ns]
Goal Power S	Static Power Consumption Unit [uW/MHz]
Design Priority	Priority of design goals
CPU Type	Processor Type
Number of Stages	Pipeline stage number

Item	Description
Number of Common Stages	Number of common stages in the pipeline
Decode Stage	Decoding stage of the pipeline
Stage Name	Each Stage Name
Attribute	Stage Attribute
Multi cycle interlock	Interlock setting for multi cycle operation
Data hazard interlock	Interlock setting for data hazard
Register bypass	Register bypass setting
Delayed branch	Delay branch setting
Num. of delayed slot	Number of delay branch slots
Max inst. Bit width	Maximum instruction bit width
Max data bit width	Maximum data bit width
Processor design	Processor design course
Use compiler	Compiler usage

3. Resource Declaration

When clicking [Resource Declaration] in Main Window, sub-window in Figure 12 opens. You can declare the resources of the processor in this sub-window.

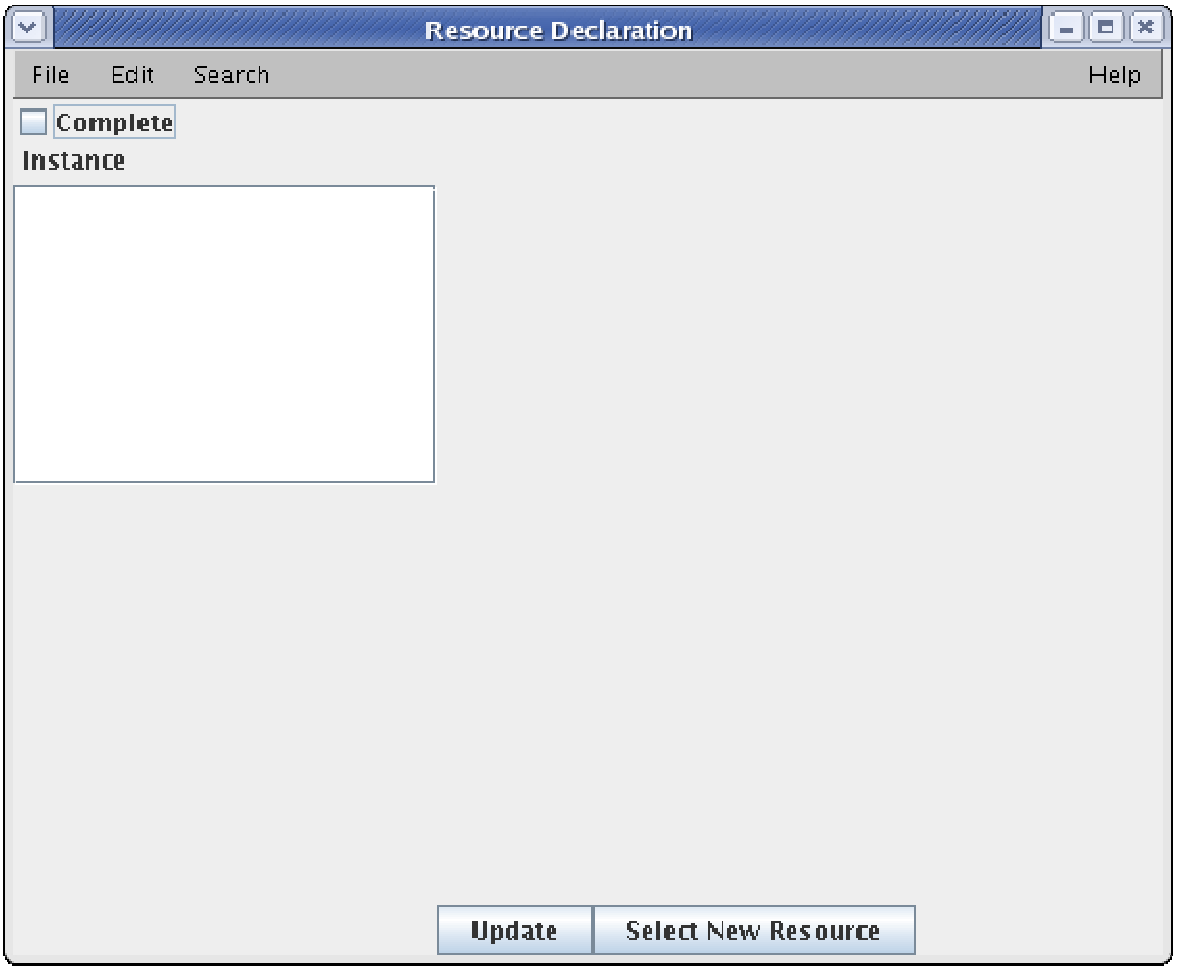


Figure 12 [Resource Declaration] Window

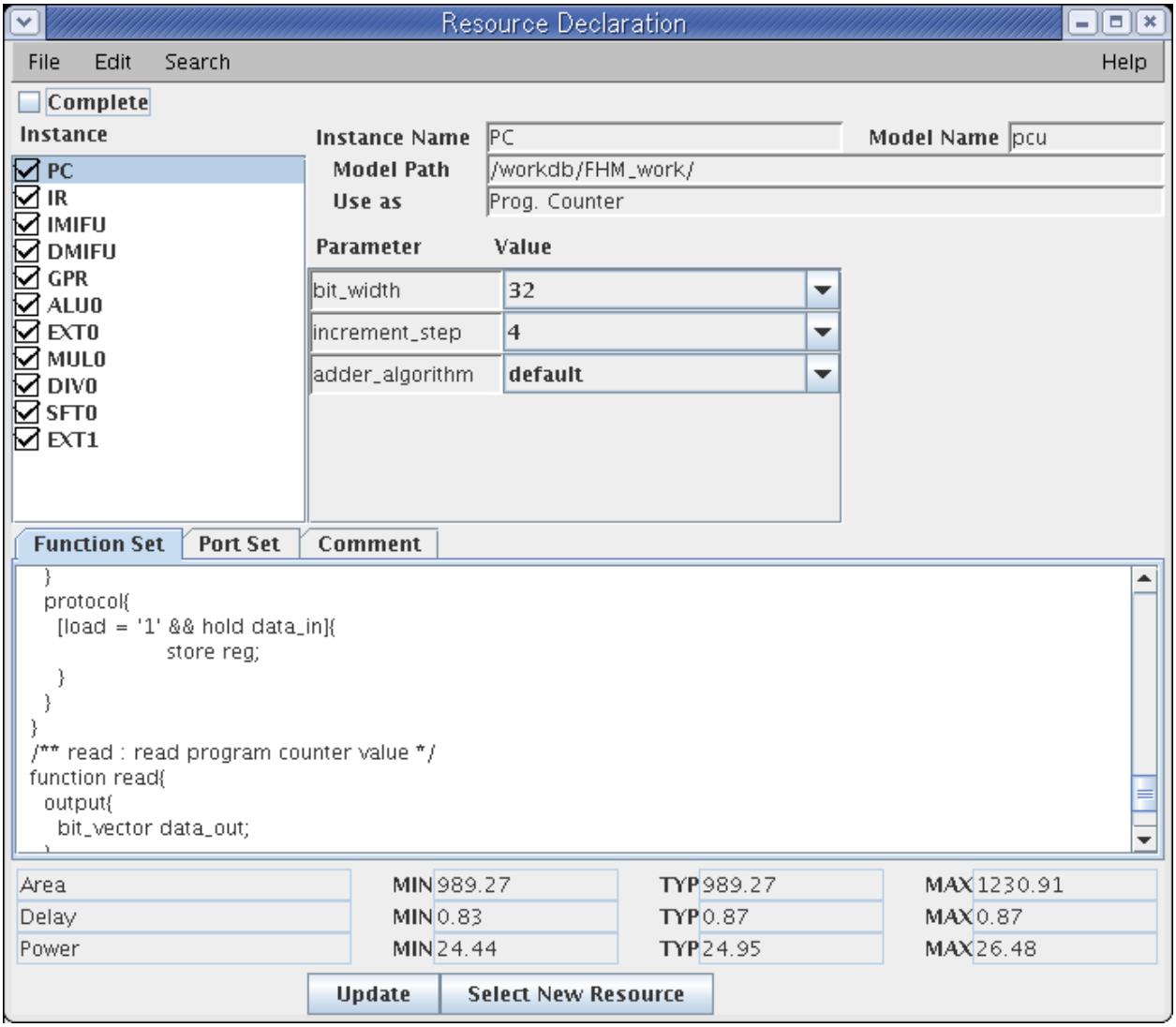


Figure 13 [Resource Declaration] Window *(when Resources are added)*

Figure 13 shows [Resource Declaration] window when resources are added.

When clicking on [Update] button, the selected resource operation information, and the resource estimation is shown in lower part window.

3.1 [Resource Declaration] Window Menu

Table 4 shows the menu in the [Resource Declaration] window.

Table 4 [Resource Decration] window menu list

Menu	Functionality
Edit	**Edit menu**
Insert	Designate the location to insert resource
Delete	Delete resource
Copy	Copy resource
Move	Move resource
Mark	Mark resource

Mark Clear	Ummark resource
Update	Update resource information
Valid	Permit resource usage
Invalid	Disallow resource usage
Search	**Search menu**
Instance	Search for resource

Each menu will be explained in order.

3.1.1 Insert

Use **Insert** to designate a location to insert a resource when a resource is added. Checking the [Insert] check box will add a new resource next to the location of the selected resource. Unchecking the check box will add a new resource to the bottom end of the [Instance] column.

3.1.2 Delete

Use **Delete** to remove a resource. The deleted resources are the resources checked in the [Instance] column. When removing a resource, no confirmation window will appear, so make sure that only the resouces you want to delete are the checked ones.

3.1.3 Copy

Use **Copy** to copy a resource. Selecting the resource you want to copy and choosing the copy option from the [Copy] menu, a window asking to insert the new resource name will appear. After you insert the name and click "OK", a copy of the selected resource will be generated carrying the newly inserted name, and will be added to the list.

3.1.4 Move

Use **Move** to move a resource. When you want to change the order of the resources in the [Instance] column, use **Move**. First, select [Mark] menu, then select the resource you want to move. Then, select the location to where you want to move the resource, and finally, select [Move] menu and the resource will move to the wanted location.

3.1.5 Mark

Use **Mark** to select a resource. When you want to use [Move] menu, use this command.

3.1.6 Mark Clear

Use Mark Clear to unmark the marked resources.

3.1.7 Update

Use **Update** to update the resource information. It updates the estimation information of the selected resource.

3.1.8 Valid

Use **Valid** to declare the usage of a resource. Using Valid will cause the selected resource in the [Instance] column to be used in the processor.

3.1.9 Invalid

Use **Invalid** to disallow a usage of a resource. Using **Invalid** for a resource selected from the [Instance] column will disallow the usage of this resource in the processor. Consequently, Invalidating a resource with not include this resource in the processor when the processor is generated.

3.1.10 Instance

Use **Instance** to search for a resource. From [Search] menu, if you choose [Instance] menu, [Research Search] window will appear. By inserted the string and click "Search" button, the name of the resource and will be searched for.

3.2 Resource Components

When clicking on [Select New Resource] in [Resource Declaration] window, the used resources selection window [FHM Browser] is displayed. The list of resources registered in [FHM Browser] is shown. Each fields of this window will be explained next.

[Model]: Resource model list

This field lists the available resources. If you click the resource in the list, [Parameter] field appears on the right side of [FHM Browser].

＜Note＞Types of used resource models in FHM can change with the change of **ASIP Meister** release version.

[Parameter],[Value]: Parameter item and its value of the resource

[Parameter] field displays the available parameter options, according to the resource selected in [Model] field. Select the value from [Value] list for each parameter.

[Use as]: Usage of the resource

Click the pull-down menu to select the role of the resource. For example, if you use register model as a program counter, select [Prog. Counter] from the list. Table 5 shows the options in the pull-down menu.

Table 5 Use as Pull-down menu Selection List

Option	Description
(unspecified)	Select this option, when any other options in the list do not apply to your usage.
Inst. Register	Instruction register
Register File	Register file
Prog. Counter	Program counter
Inst. Memory	Instruction memory
Data Memory	Data memory
Plain Register	Plain register
Mask Register	Mask register (for interrupt)

＜Note＞ Each processor must include at least one resource as [Inst. Register], [Prog. Counter], [Inst. Memory] and [Data Memory].

[Function Set]: Function of the resource

Click [Function Set] tab to check the function of the resource selected in the list on [Model] field. Use this option to check whether the selected resource model provides the intended function that you wish or not.

[Port Set]: I/O interface of the resource

Click [Port Set] tab to view the port name, direction of I/O, data type, bit width and attribute of the signal shown in Table 6.

Table 6 Attribute List

Attribute	Description
Clock	Clock Signal
Ctrl	Control Signal
Reset	Reset Signal
Data	Data Signal
Mode	Mode Selection Signal

[Estimate]: Estimation of resource

If you click [Estimate] button, the value for the area, delay and power consumption are estimated based on the parameters you have set. These estimated values are displayed in

[Area], [Delay], and [Power] rows respectively. Each row displays three different estimated values. [MIN] field shows the minimum estimated value and [MAX] shows the maximum estimated value. [TYP] shows the estimated value with the design priority that you set in [Design Goal & Arch. Design] step.

[Instantiate]: Create resource instance

Click [Instantiate] button, and then [New Instance] window opens. You can set the instance name in the window. Setting the instance name and clicking [OK] button, the instance will appear in [Resource Declaration] window.

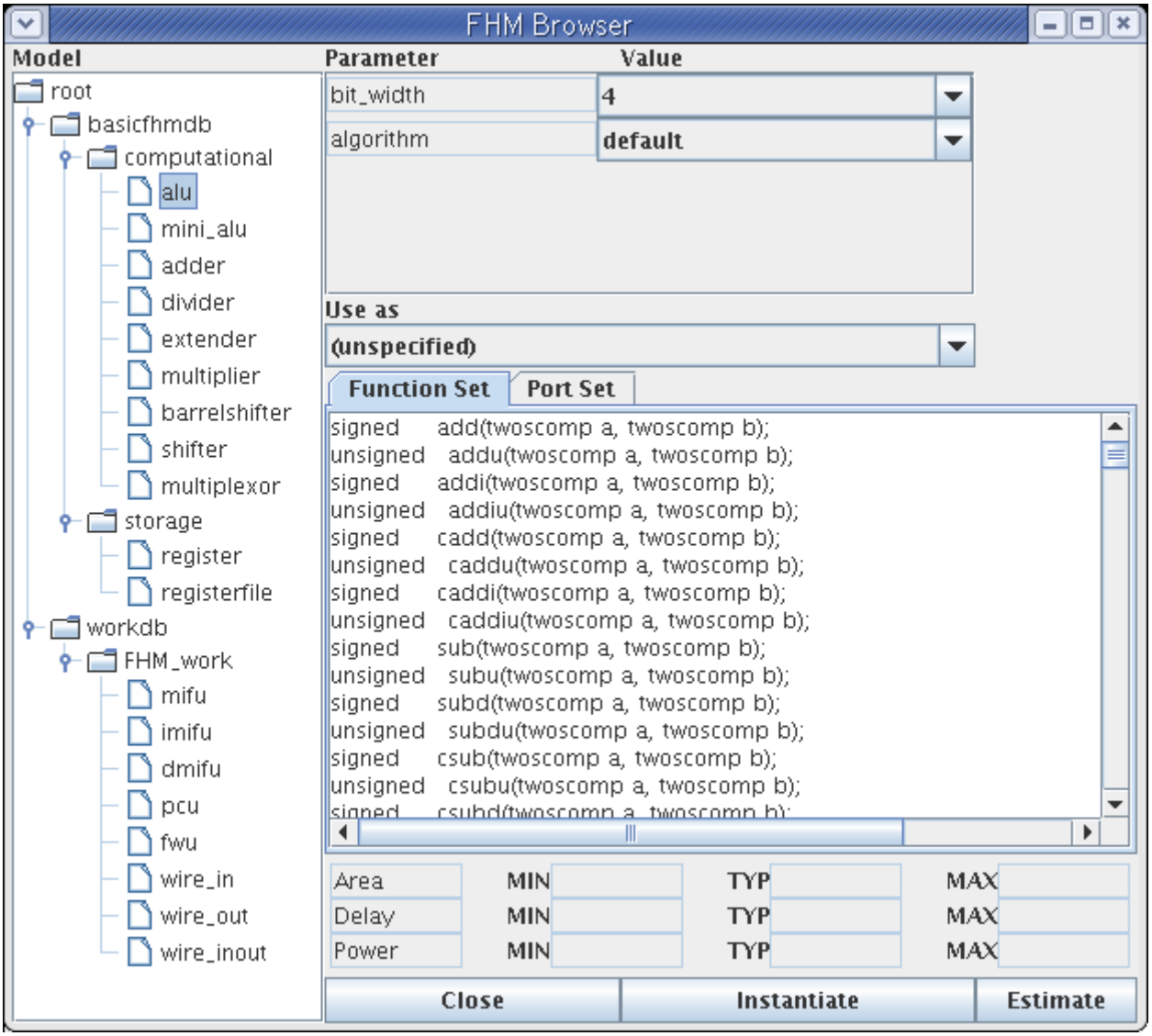


Figure 14 FHM Browser

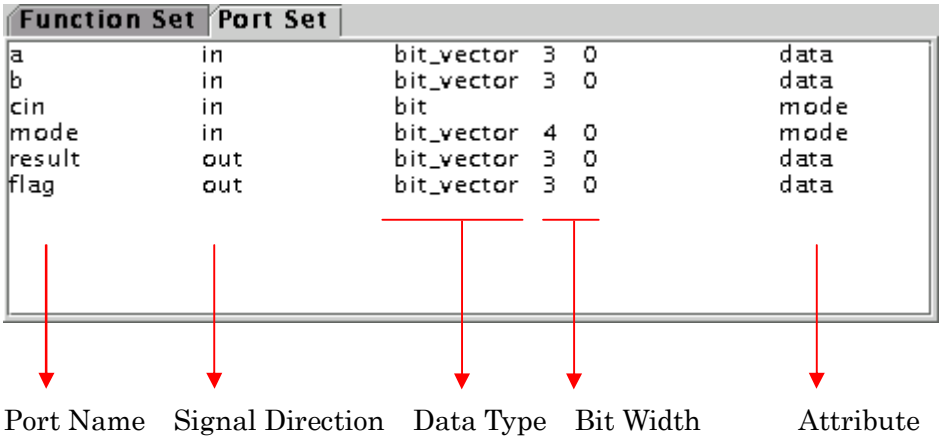


Figure 15 [Port Set]: Resource Model Input/Output Interface

4. Storage Spec

From [Storage Spec] window, for register file, registers, memory and other storage you can set different specification such as name, data bit width, and the register uses definiton.

The defined storage information here will be used by the generated meta-assembler assemble grammar.

When you click [Storage Spec.] button, [Storage Spec] window opens. The storage specification window consists of three parts; [Register File], [Register], and [Memory]. Explanation of the input to each is presented next.

4.1 [Register File] Tab

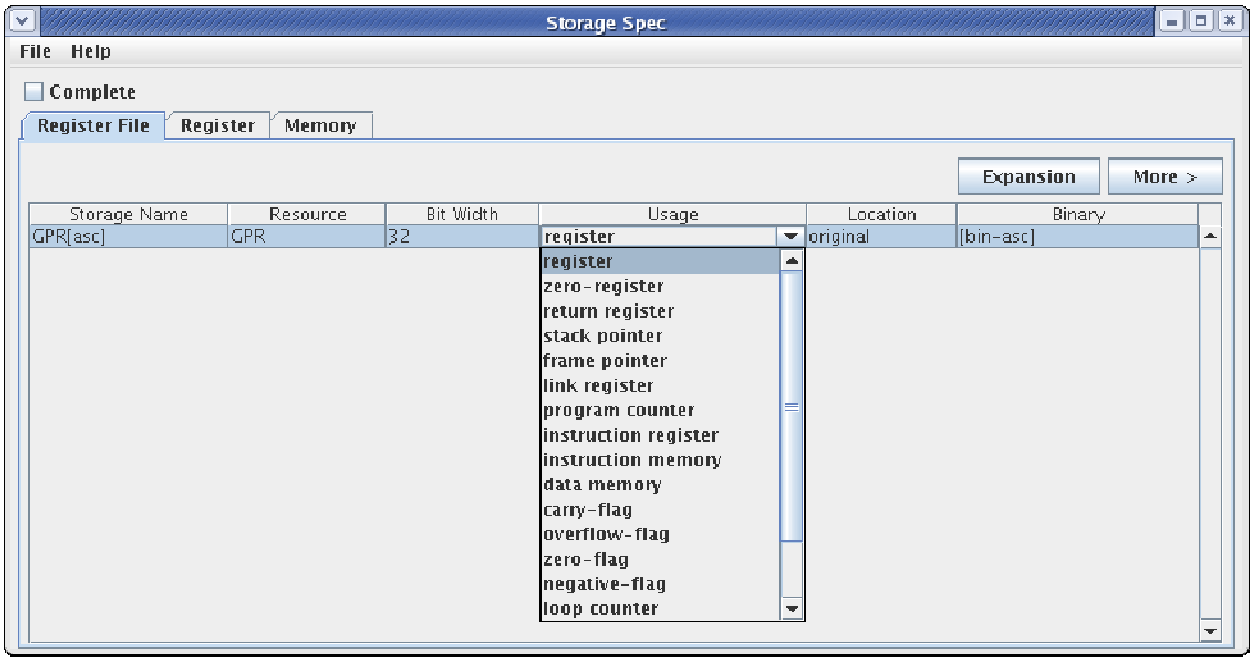


Figure 16: Register File Definition

When you select [Register File] tab in [Storage Spec] window, you can define the specification of the register files. Table 7 shows the register file specification.

Table 7: Storage Spec: Register File

Storage Name	Storage Name Used by Assembly Code
Resource	Resource name declared in [Resource Declaration] step
Bit Width	Storage data bit width
Usage	Storage usage that specify how to handle the storage by the compiler
Location	Storage location
Binary	Binary representation for machine code

Since register file has two or more registers, the number of registers can be specified using the following key words;

[asc]: When register file has n registers, register index 0 to n-1 can be specified in ascending

order.
[des]: When register file has n registers, register index n-1 to 0 can be specified in descending order.

Moreover, binary representation of each register can be specified using the following words;

[binary-asc]: binary representation in ascending order
[binary-dsc]: binary representation in descending order

Storage usage can be selected from the list box. Table 8 shows the storage usage different selections.

Table 8: Storage Usage

Register	Data register used for general calculation
Zero register	Register with zero value
Return register	Register that keeps the return value of the functional call
Stack pointer	Register used as the stack pointer
Frame pointer	Register that keeps the head address of local variables when a function is called.
Link register	Register that keeps the return address
Program Counter	Register that keeps the memory address of instruction to fetch next.
Instruction register	Register that keeps the fetched instruction
Instruction memory	Memory that contains instructions
Data memory	Memory that contains data to handle by processor
Carry flag	Register that contains carry flag data
Overflow flag	Register that contains overflow flag data
Zero flag	Register that contains zero flag data
Negative flag	Register that contains negative flag data
Loop counter	Loop counter for zero-overhead loop (future extension)
Start address	Start address for zero-overhead loop (future extension)
End address	End address for zero-overhead loop (future extension)
Instruction number	Register that keeps the number of instructions that specifies for zero-overhead loop block (future extension)

You can specify overlapped registers using [Location] field. When specified, the storage name is described in [Location] field, if not necessary, “original” is written. In the register file specification, just write “original” to [Location] field.

In register file definition, designers can specify the register specification to each register. When you click [Expansion] button, the window with the expanded storage list opens (Figure 17). Usage of each register and other features can be specified in this window. [More] button also shows the expanded lists but it does not reflect the change of the register file specification. If you want to check the previous setting, click [More] button, otherwise, click [Expansion] button.

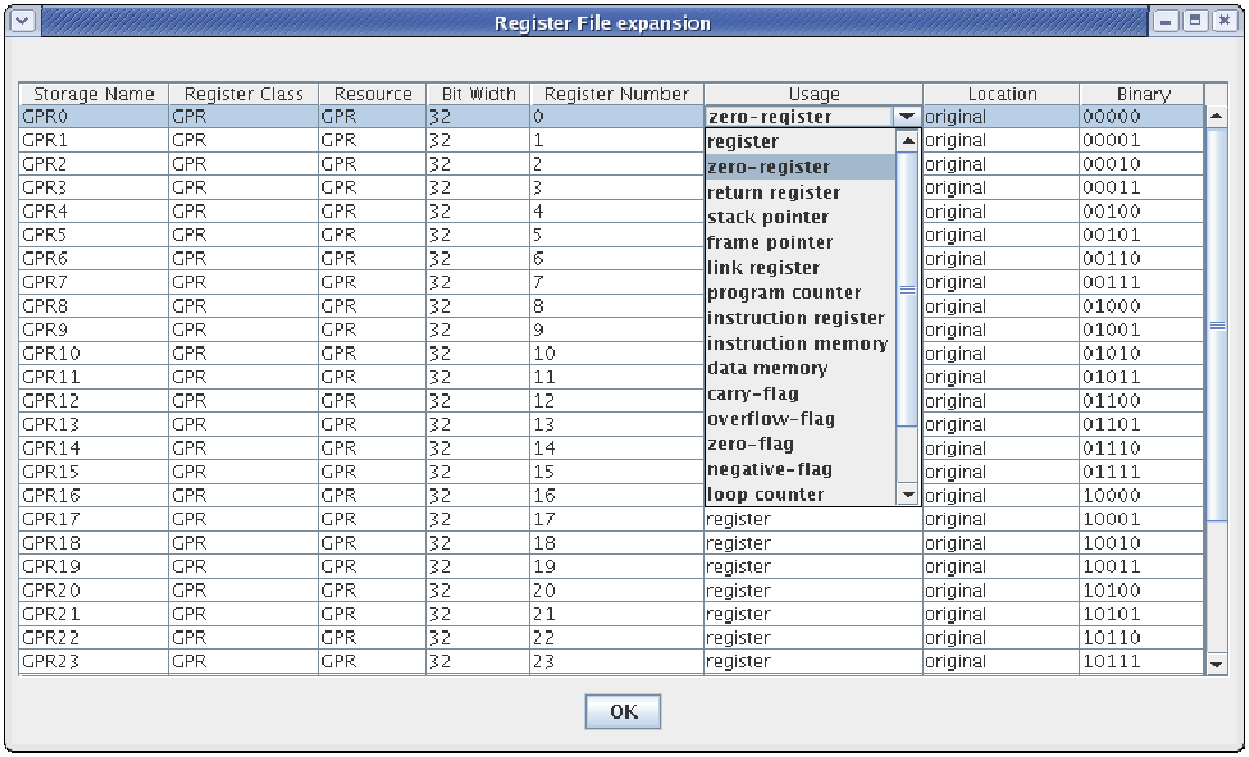


Figure 17: Expanded Storage List

4.2 [Register] Tab

When you select [Register] tab in [Storage Spec] window, you can define the specification of the registers including storage name, resource name, bit width, resource usage, and location. (see Figure 18).

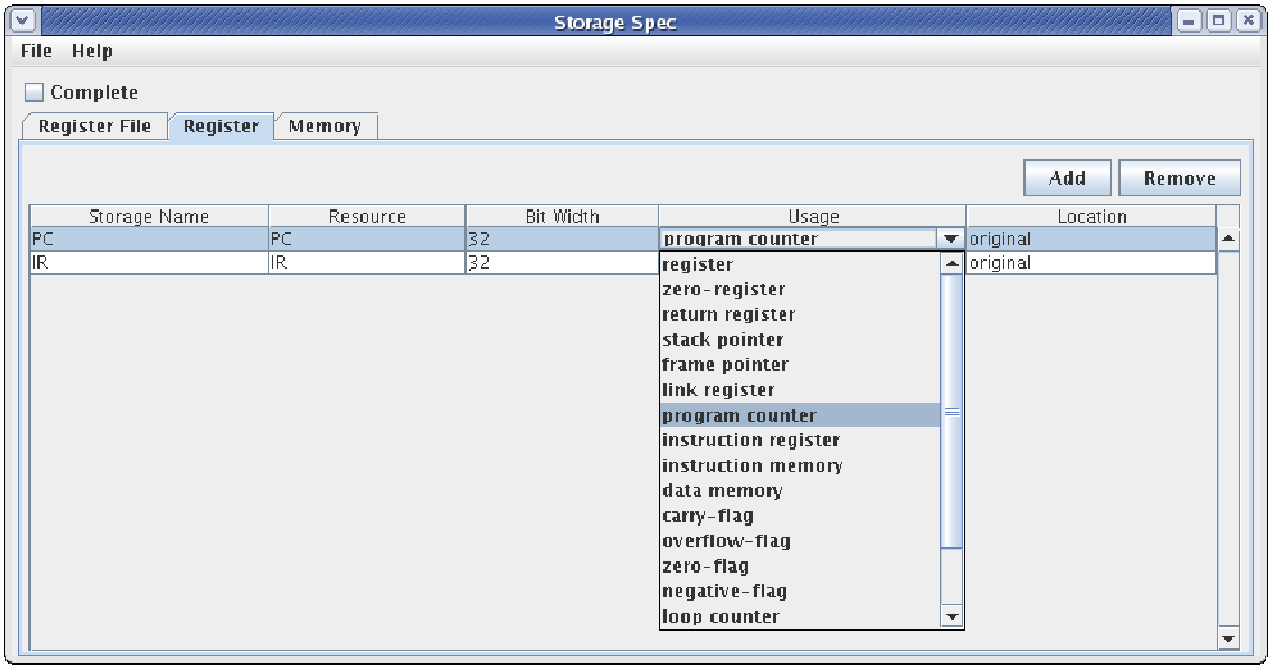


Figure 18: Register Definition

4.3 [Memory] Tab

When you select [Memory] tab in [Storage Spec] window, you can define the specification of the memories including storage name, resource name, bit width, usage, and access bit (see Figure 19). Access bit is the minimum access bit width when the processor accesses the memory.

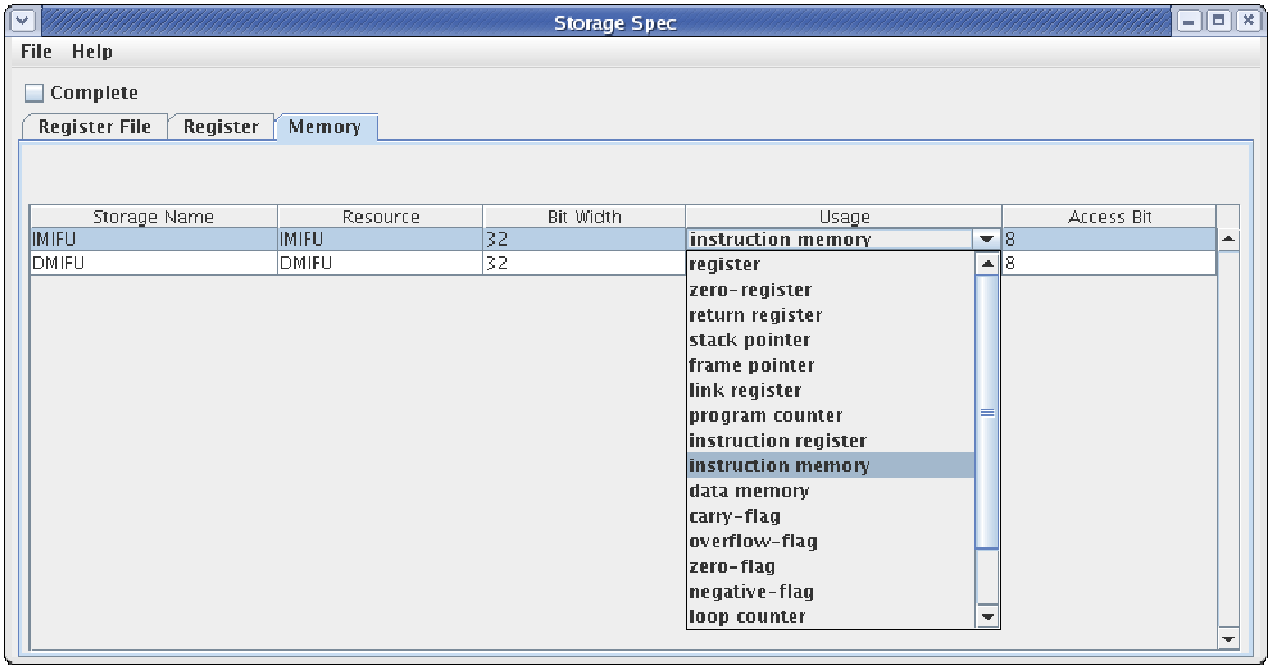


Figure 19: Memory Definition

5. Interface Definition

When clicking [Interface Definition] in Main Window, [Interface Definition] window (Figure 20) opens. In this window, you can enter the instance name of the processor and I/O information to determine the external interface of the processor.

Under [Complete] button, the processor entity name is displayed. Below is the defined ports list. Each line corresponds to one port. The validity of the port, attribute, name, direction, bit-width is also shown. On the right side of window, there is a [New Port] button to add new ports.

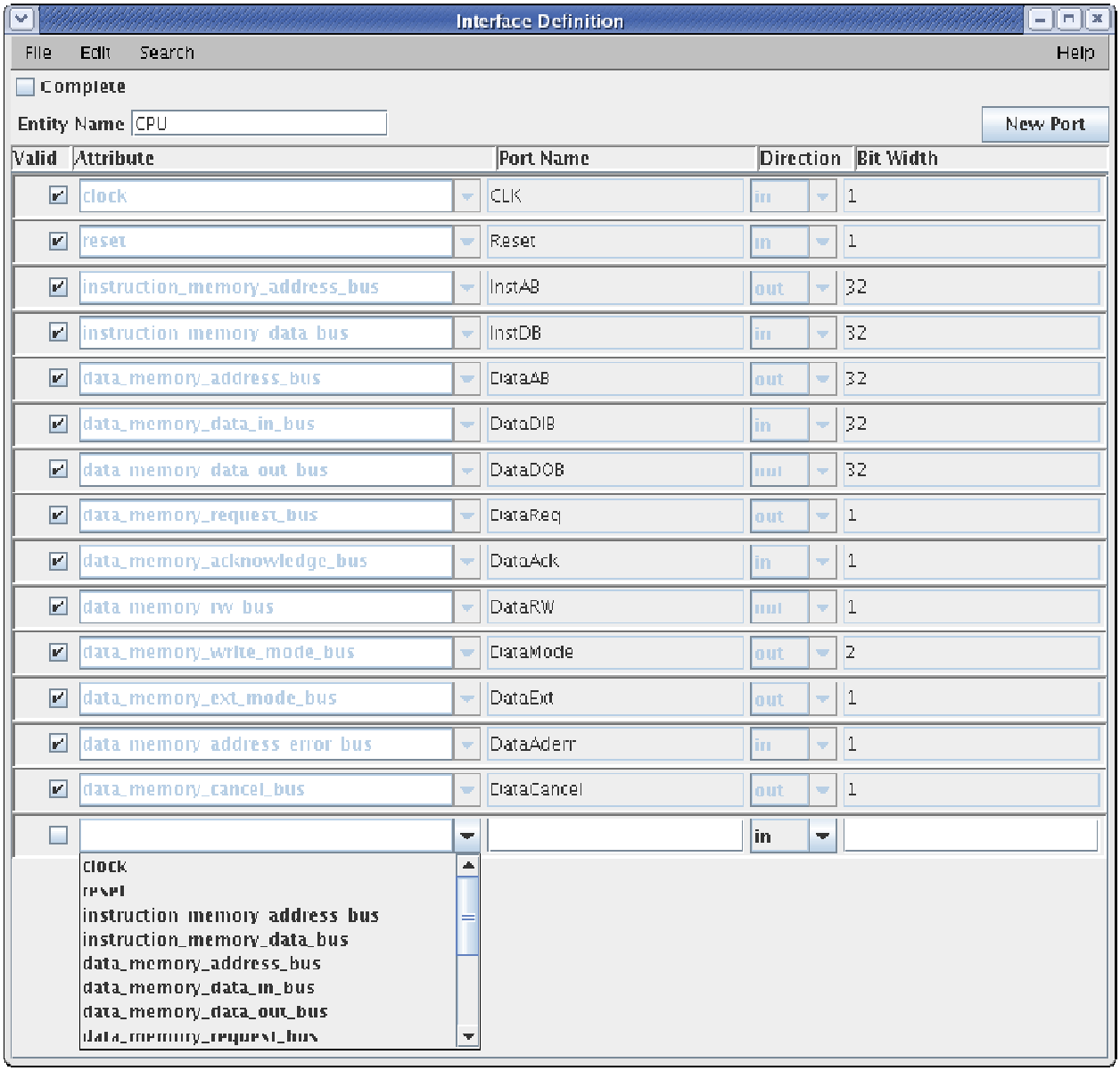


Figure 20 [Interface Definition] Window

＜Note＞In **ASIP Meister**, only the memory interface unit (MIFU) and WIRE resources can connect to the external ports. According to the port attributes, HDL generation engine connects I/O ports to MIFU or WIRE ports.
ASIP Meister has a design data file where the memory ports are defined. Using this design data file for the memory structure, it is possible to skip the setting of external port attribute.

[Entity Name]: Processor Name

This field shows the instance name of the target processor. In [HDL Generation] step, the generation engine uses the name as the entity name of the target processor. The name with ".vhd" or ".v" extension becomes the file name of the main HDL description.

.

＜Note＞ Because the name used here would be the processor entity name in [HDL Generation]. The name should follow the naming rules of the simulation or synthesis tool that would be used after HDL generation.

＜Reference＞ 4. Reserved Words

[Valid]: Port Valid/Invalid setting

This option specifies to implement the port or not. When the checkbox is checked, the port is implemented in the processor. When the checkbox is unchecked, the port is not implemented as an external port of the processor.

[Attribute]: Attribute of the port

This field specifies the port attribute. According to the attributes specified here, HDL generation engine connects ports and their internal signals. Table 9 lists the attributes and their description.

Table 9 External Port Attribute List

Attribute	Description
Clock	Clock
Reset	Reset
instruction_memory_address_bus	Instruction Memory Address Bus
instruction_memory_data_in_bus	Instruction Memory Data In Bus
instruction_memory_data_out_bus	Instruction Memory Data Output Bus
instruction_memory_request_bus	Instruction Memory Request Bus
instruction_memory_acknowledge_bus	Instruction Memory Acknowledge Bus
instruction_memory_rw_bus	Instruction Memory Read/Write Bus
instruction_memory_write_mode_bus	Instruction Memory Access Mode Bus
instruction_memory_ext_mode_bus	Instruction Memory Read Sign Extension
instruction_memory_address_error_bus	Instruction Memory Address Error Bus
instruction_memory_cancel_bus	Instruction Memory Cancel Bus
data_memory_address_bus	Data Memory Address Bus
data_memory_data_in_bus	Data Memory Data In Bus
data_memory_data_out_bus	Data Memory Data Out Bus
data_memory_request_bus	Data Memory Request Bus
data_memory_acknowledge_bus	Data Memory Acknowledge Bus
data_memory_rw_bus	Data Memory Read/Write Bus
data_memory_access_mode_bus	Data Memory Access Mode Bus
data_memory_ext_mode_bus	Data Memory Read Sign extension
data_memory_address_error_bus	Data Memory Address Error Bus
data_memory_cancel_bus	Data Memory Cancel Bus

Attribute	Description
interrupt	Interrupt Bus
Unspecified	User defined WIRE Resource

[Port Name]: Port name

You cannot change the port name when the checkbox in [Valid] field is checked. To change the external port name, first uncheck the checkbox and change the name. You must not use VHDL reserved words as the port name.

＜Reference＞4. Reserved Words

[Direction]: Direction of the signal

Pull down the menu to select the direction of the external signal. You can select [in] for input signal, [out] for output signal and [inout] for input and output signal.

[Bit Width]: Bit width of the port

This field specifies the bit width of the port.

[New Port]: Adding port

Clicking [New Port] button, a new blank row appears at the end of the list to declare a new port.

＜Note＞A new line cannot be added after the last line where no port definition is shown. i.e. ([PortName] and [Bit Width]) are empty.

Table 10 [Interface Definition] Setting Items List

Parameter	Description
Entity Name	Processor Name (Entity Name)
Attribute	External Port Attribute
Port Name	External Port Name
Direction	Signal Direction - Input[in], Output [out], Bi-directional [inout]
Bit Width	Bit Width

6. Instruction Definition

When clicking [Instruction Definition] in Main Window, [Instruction Definition] window (Figure 21) opens.

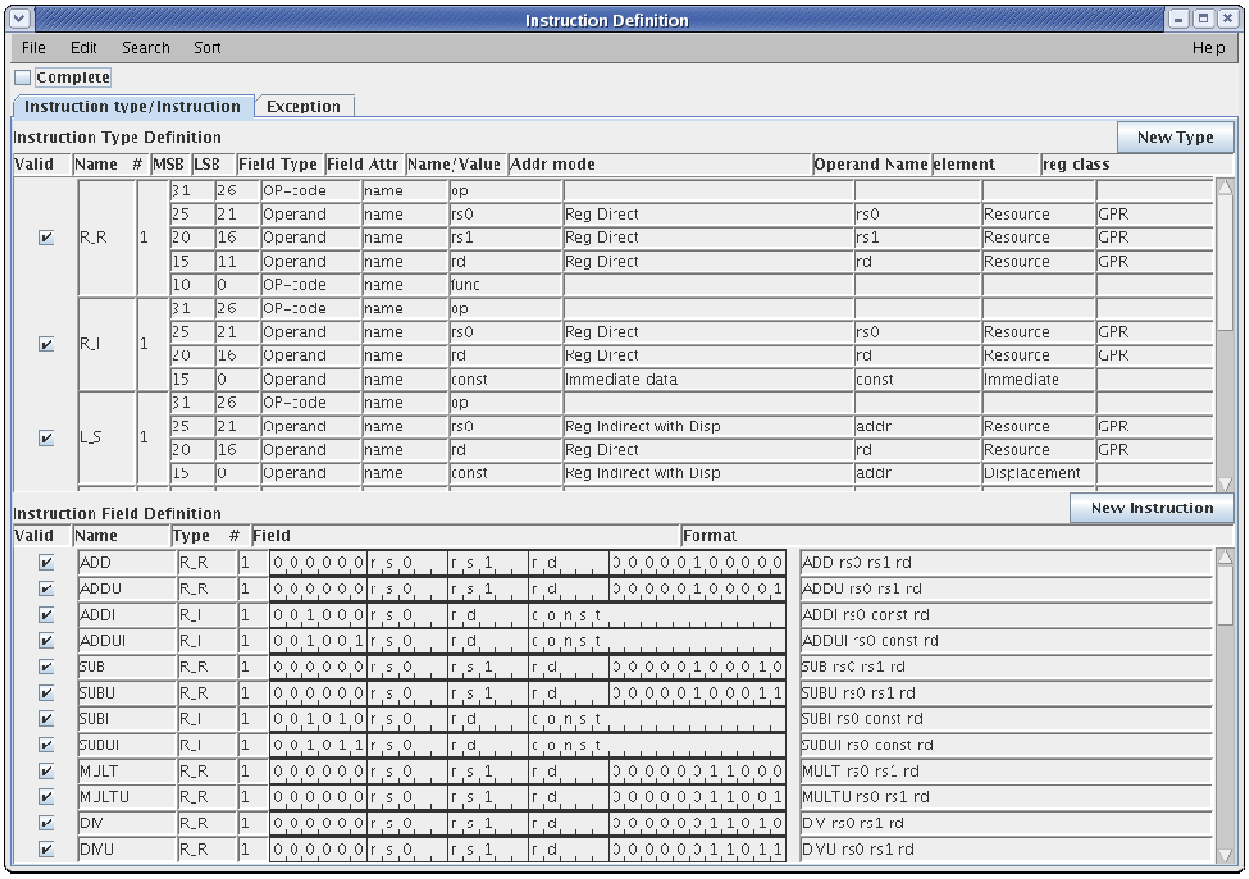


Figure 21 [Instruction Definition] Window

In [Instruction Definition] window, you can define the instruction types, instruction formats and interrupt conditions in **ASIP Meister**. To define the instruction type and instruction format, click [Instruction type/Instruction] tab. You cannot define the instruction format without the instruction type, so, you must define the instruction type first. To define the interrupt, click [Exception] tab.

In [Design Goal & Arch. Design] phase, the limits in [Instruction Definition] changes according to the settings of [Processor Design] and [Use Compiler]. These limits will be explained next.

- [New Design] : No limits.
- [New Design with Base Processor]
  - When [Use Compiler] is set to "yes"
    - Concerning the base processor, instruction type modification or deletion are not possible.
    - Addition of New instruction types and addition of new instructions are possible.
  - When [Use Compiler] is set to "no"
    - No limits
- [Base Processor Design] : No limits

6.1 Instruction Type Definition

[Instruction Type Definition] field describes the instruction type information with the following values;

[Valid]: valid/invalid instruction type

This checkbox specifies the validity of using the instruction type or not. Check the checkbox to allow using the instruction type and uncheck it to disable using the instruction type.

[Name]: Instruction Type Name

[#]: Identification number of the instruction type

You can use an identification number to distinguish instruction types with the same name. Note that, the instruction type with the same name but a different identification number ([#]) cannot be used simultaneously. If you try to check [Valid] checkbox of instruction type "IT (2)" with some instructions with the instruction type "IT (1)" used, a confirmation dialog appears. When you select [OK] in the dialog, the [Valid] checkbox of all instructions with the instruction type "IT (1)" becomes unchecked (not to implement).

[MSB]: Most Significant Bit Field

This field displays the most significant bit (MSB) of each instruction type field.

[LSB]: Least Significant Bit Field

This field displays the least significant bit (MSB) of each instruction type field.

[Field Type]: Field Type

[Field Type] box displays the type of the each field. This field shows one of four selections [OP-code], [Operand], [Reserved], and [Dont_care].

[Field Attr]: Sharing/un-sharing of the instruction type field

When using a given instruction type to define instructions, this field can be used to specify that a field type can be shared among all instructions in case [binary] is selected, while on the other hand, it can be different for all instructions in case [name] is selected. When the [Field Type] is [Operand], nothing but [name] can be selected.

Table 11 [Field Attr] Parameter Selection States

Field Type	Available [Field Attr]	Description
OP-code	Binary	If you would like to use the same value for all instructions of a given instruction type, use this field attribute.
	Name	If you would like to set different values in the field for each instruction of a given instruction type, use this field attribute.
Operand	Name	When the [Field Type] is [Operand], you can select only [name].
Reserved	Binary	If you would like to reserve the field for future extension, and this field must be the specific binary, select this pair.
Dont_care	Name	If you do not mind what value appears in this field, select this pair.

[Name / Value]: Field Name/Value

This field displays the name or value of the field. When [binary] is displayed in [Field Attr], this field should be filled with the binary value, and when [name] is displayed, this field displays the name.

[Addr Mode]: Addressing Modes

Operand usage is specified with the addressing mode. Table 12 describes the addressing modes.

Table 12 Addressing Modes

Addressing Mode	Element Type	Description
Reg Direct	Resource	The data is in the register specified by the value.
Indirect	Immediate	The data is in the register specified by the data of the register specified by the value.
Reg Indirect	Resource	The data is in the memory specified by the data of the register specified by the value.
Reg Indirect with Pre Decrement	Resource and Displacement	The data is in the memory specified by the data of the register specified by the value. The data of the register is decremented before memory access.
Reg Indirect with Pre Increment	Resource and Displacement	The data is in the memory specified by the data of the register specified by the value. The data of the register is incremented before memory access.
Reg Indirect with Post Decrement	Resource and Displacement	The data is in the memory specified by the data of the register specified by the value. The data of the register is decremented after memory access.
Reg Indirect with Post Increment	Resource and Displacement	The data is in the memory specified by the data of the register specified by the value. The data of the register is incremented after memory access.
Reg Indirect with Disp	Resource and Displacement	The data is in the memory specified by the data of the register specified by the value. The data of the register is displaced by the value after memory access.
Reg Indirect with Disp and Increment	Resource and Displacement	The data is in the memory specified by the data of the register specified by the value. The data of the register is displaced by the value and incremented after memory access.
Reg Indirect with Index	Resource and Displacement	The data is in the memory that address equals to the sum of the data of the register specified by the value and the index value.
Reg Indirect with Index and Increment	Resource and Displacement	The data is in the memory that address equals to the sum of the data of the register specified by the value and the index value. The data of the register is incremented after memory access.
Reg Indirect with Scaled Index	Resource and Displacement and Scale	The data is in the memory that address equals to the sum of the data of the register specified by the value and the scaled index value.
Reg Indirect with Disp and Scaled Index	Resource and Displacement and Scale	The data is in the memory that address equals to the sum of the data of the register specified by the value and the scaled index value.
PC relative address	Symbol	PC Relative addressing

Addressing Mode	Element Type	Description
Absolute address	Symbol	Absolute addressing
Immediate data	Immediate	Immediate data

[Operand Name]

In the assembler language, operands are specified with mnemonic format. In **ASIP Meister**, mnemonic format is described using string and operand name. Designers can declare the name of the operand in this field. Please note that when you select the addressing mode that consists of two elements such as "Reg Indirect with Disp" addressing and you would like to make an operand with two fields, please describe the same operand name at "Operand Name" section.

[element]

The addressing mode element type is specified here. The element is selected from a pull-down menu. Based on Table 12 Addressing Mode, the designer should describe the element type of each operand. For example, when "Reg Indirect with Disp" addressing mode is selected, "resource and displacement" should be also selected.

[reg class]: Register class

When you select the type of element as "Resource", you have to specify register class. The register class specifies the accessible registers of the operand. For example, if you select "GPR" from the list, the operand can access "GPR" register file. If you select "GPR0", that is one of the register file elements, the operand can only access "GPR0" register. The register class is declared in the storage declaration step. Note that this register class has been declared at the previous design phase "Storage Specification".

Table 13 [Instruction Type Definition] Parameter

Parameter	Description
Instruction Type Name	Instruction Type Name
Number of Fields	Number of Fields
MSB	Most Significant Bit
LSB	Least Significant Bit
Field Type	Field Type Specification Operation Code [OP-code] Operand [Reserved] Field [Dont_care]
Field Attr	Selected as either [binary] The field value is shared among instructionts. [name] The field value is set differently for each instruction.
Name / Value	Field Value Operation Code, Reserved Field Binary Value Operation Code Name, Operand Name, Don't Care
Addr mode	Operand Addressing Mode
Operand Name	Operand Name
Element	Specifies Addressing Mode Element
reg class	Specifies Addressing Mode Register Class

[New Type]: New Instruction Type Definition

Clicking [New Type] button, [New Instruction Type confirm] window (Figure 22) will open. In [New Instruction Type confirm] window, enter the instruction type name and the number of fields.

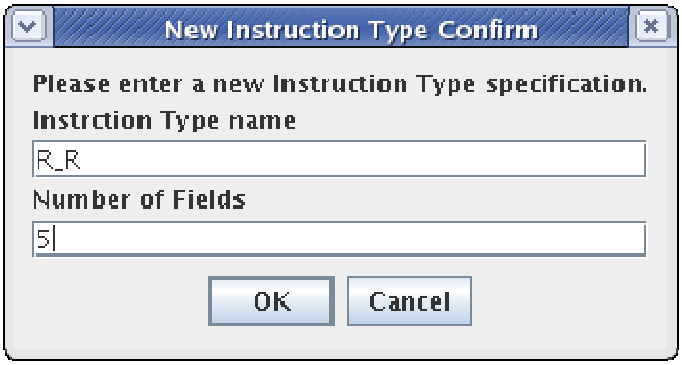


Figure 22 [New Instruction Type Confirm] Window

If you click [OK] button in [New Instruction Type Confirm] window, [Instruction Type Declaration Window] window (Figure 23) opens.

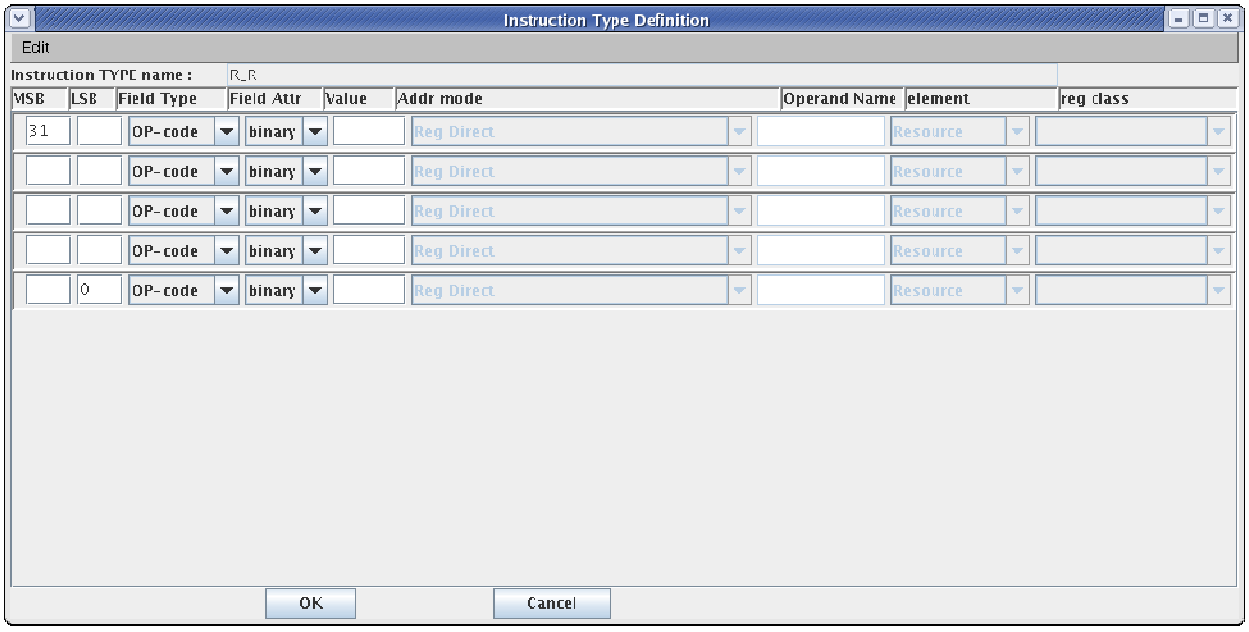


Figure 23 [Instruction Type Definition] Window

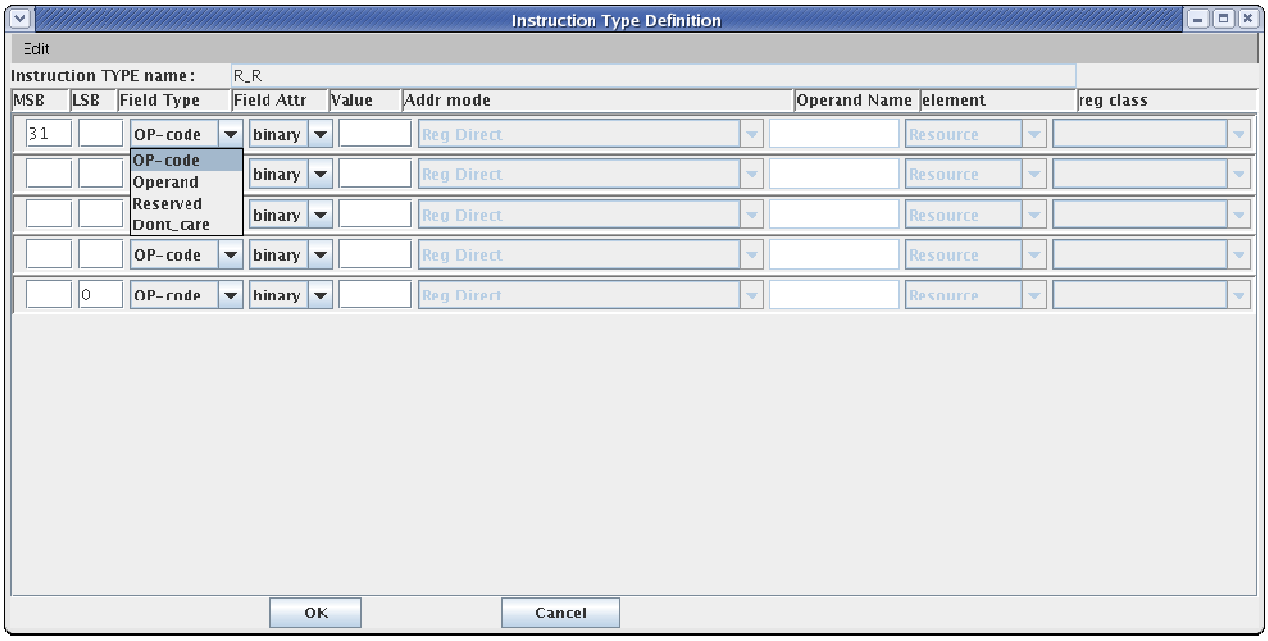
In the [Instruction Type Definition] window each field information is defined.

(1) Field bound: [MSB],[LSB]

Enter the MSB and LSB of each field.

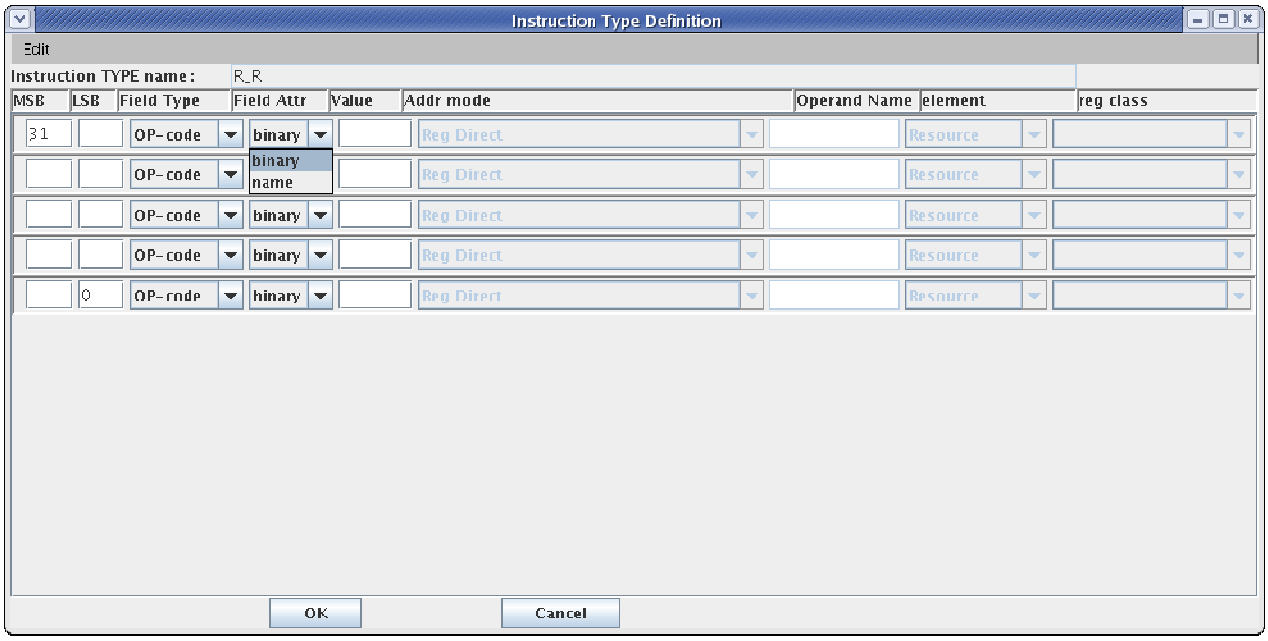
(2) Field Type: [Field Type]

Select the field type from the pull down menu ▼ (See figure below). That menu consists of [Op-code] (operation code), [Operand] (operand), [Reserved] (reserved field) and [Dont_care] (don't care field) types.



(3) Field attribute: [Field Attr]

Select the field attribute [name] or [binary] from the pull down menu ▼ (See Figure below)



(4) Field Value: [Value]

Set the field value. If [Field Attr] is [binary], set the binary value, and if [Field Attr] is [name], set the field name.

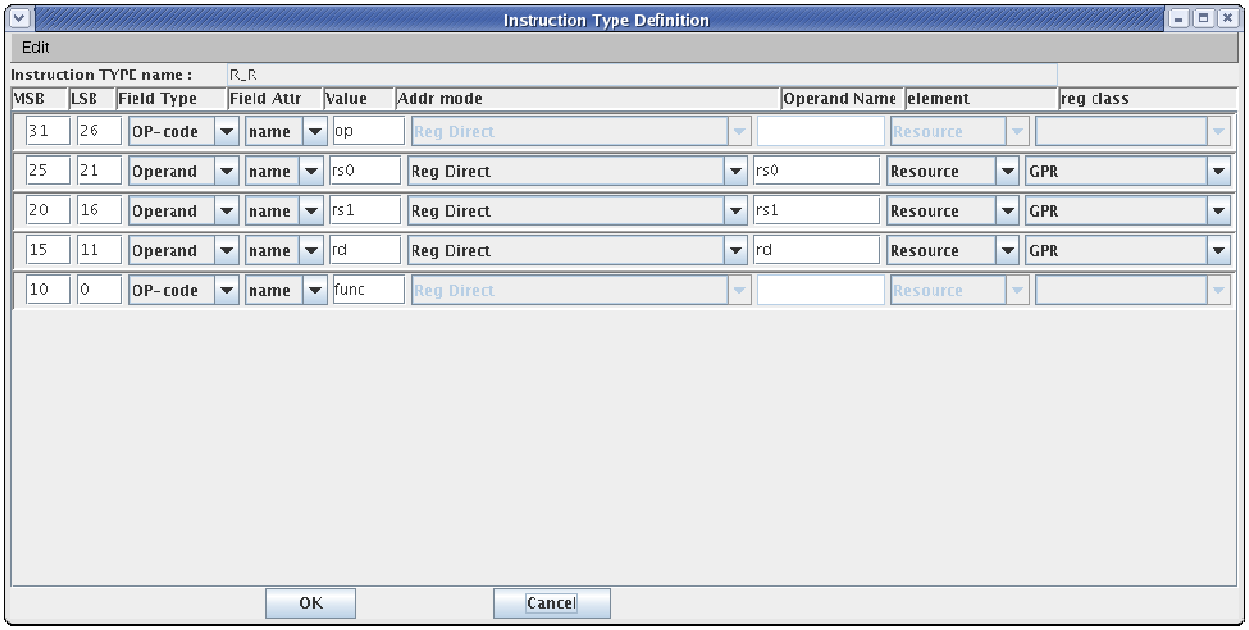


Figure 24 [Instruction Type Definition] Window

(5) After all settings are completed press [OK] button.

6.2 Instruction Declaration

In [Instruction Declaration], the [Instruction type] is used to define each instruction type.

(6) Instruction Declaration Parameter

In the table below, [Instruction Declaration] parameters are shown.

Table 14 [Instruction Declaration] Parameter

Parameter	Description	Input Data or Input Method
Instruction Name	instruction Name (mnemonic)	Character string < Reference > 4. Reserved Words
Instruction Type	Instruction Type	Select from predefined instruction type list
Value	Operation Code or Operand. This field is only enabled when the [Field Attr] field is set to [name] in the instruction type definition. Otherwise, the field value of the instruction type is inherited.	Enter the binary value of the Operation Code or the name of the Operand .

[New Instruction]: Adding New Instructions

If you click [New Instruction] button on [Instruction Field Definition] field, [Input] window (Figure 25) opens. Enter the instruction name and click [OK] button, and then, [Instruction Definition] window (Figure 26) will open.

Figure 25 Instruction Name Input Window

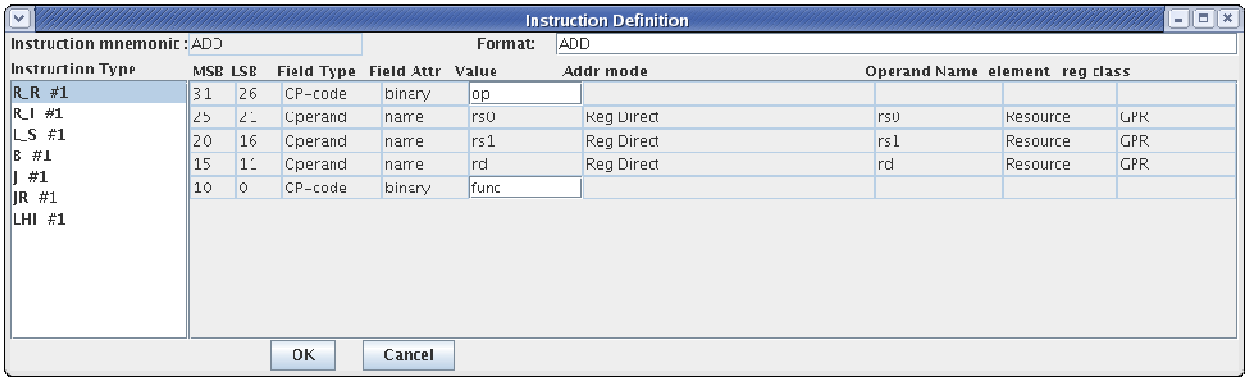


Figure 26 New Instruction Definition

[Instruction mnemonic]: Instruction name

This field displays the instruction name that has been entered in [input] window. This name can not be changed.

[Instruction Type]: Select the instruction type

This field lists the defined instruction type and its identification number. Select the instruction type to use from this list. After you select the instruction type, the right pane of the window displays number of rows corresponding to the number of the fields. Note that each row corresponds to each field and each row includes the value of [MSB], [LSB], [Field Type], [Field Attr] and [Value].

[Value]: Value of field

This field displays the value of each field. [Value] field of the instruction is only enabled when [Field Type] and [Field Attr] are set to [OP-code] and [name] respectively in the instruction type definition. Enter the value of the field to [value] option. Otherwise, the field value of the instruction type is inherited.

When the [Field Attr] of the instruction type is [binary], there will be an input box to enter the value. Otherwise, the value of the field type is inherited.

[Format]: Instruction assembly format

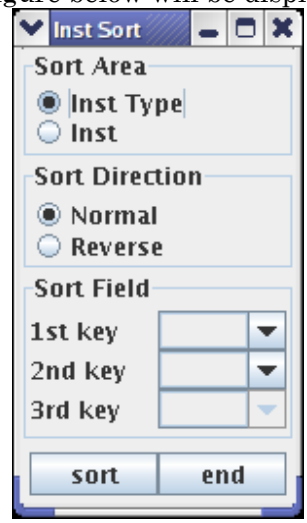
This field displays the instruction assembly format. With the assembly format, you can set the name along with all the operands of the instruction. Changing the order of the description will change also the specifications of the generated meta assembler and the compiler.

＜Note＞ If you set the [Field Type] to “Operand” and setting the [Field Attr] to ”name”, then it is mandantory to set the [Format] field as well.

[Sort] Menu: Instruction Type Sorting

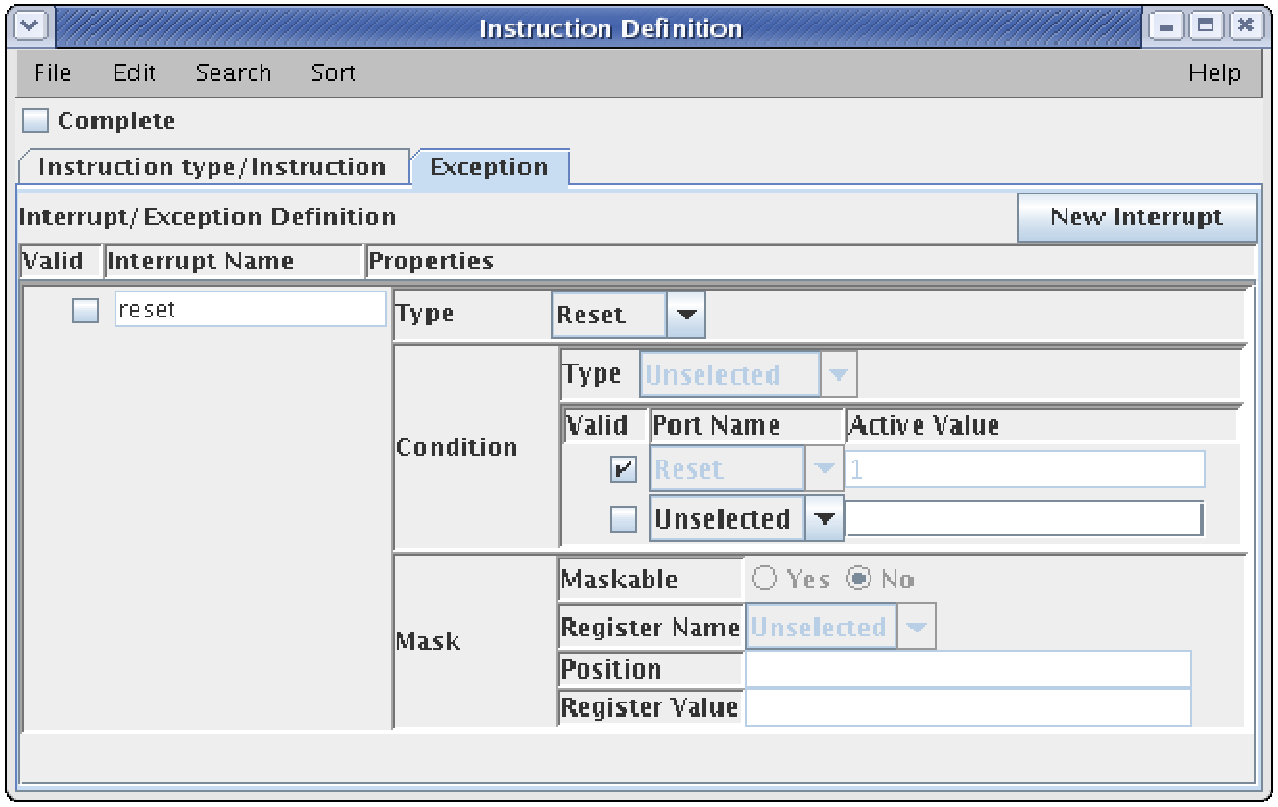
When clicking on [Sort] in [Instruction Definition] window menu bar, the window shown in

figure below will be displayed. In this window, it is possible to sort instruction types.



6.3 Interrupt/Exception Declaration

By clicking [Exception] tab in [Instruction Definition] window, you switch to [Interrupt/Exception Declaration] pane. Note that the reset interrupt definition is MANDATORY.



[Interrupt Name]: Interrupt (Exception) Name

[Type]: Interrupt Type

Select the interrupt type from: [External] for the external interrupt, [Internal] for the internal interrupt, [NMI] for the non-maskable interrupt or [Reset] for the reset interrupt.

[Condition]: Describe the condition of interrupt generation

Set the conditions of the interrupt.

Based on external port and signal value, describe the condition. From the pulldown menu choose an external port, and set the [Active Value] column with the signal value. In the case of internal interrupt only, you have to set the type field. Choose one of the 3 values "Unselected", "decoder_error" and "instr_specific".

[Valid]: Enable/Disable interrupt

Use this checkbox to enable/disable the defined interrupt.

Table 15 Interrupt (Exception) Definition Parameter

Parameter	Description	Input Data and Input Method
Interrupt Name	Specify Interrupt (Exception) Name	Optional character string
Type	Specify Interrupt Type	▼Selection from Pull down Menu External: External Interrupt Internal: Internal Interrupt NMI: Non Maskable Interrupt Reset: Reset Interrup (Mandatory)
Condition	Describe the condition of interrupt generation	Based on external port and signal value, describe the condition. Corresponding to the set external port and the signal value, the time that the interrupt occurs is described.
Valid	Enable/Disable Interrupt	Click on Check-box

＜Note＞

・There is only one external interrupt.

・Reset interrupt is mandatory.

7. Arch. Level Estimation

In [Arch. Level Estimation], you can check the estimated performance of the processor (area, delay, and static power consumption). Since this is the estimation before finalizing the details of the design, the estimated values are not exact values, but these values can help you in the following design steps. If the estimation result shows lower values than your expectation, you can change the design settings (e.g. Used resource models or parameters) and re-estimate the design quality.

[Estimation Confirm]: Estimation System Selection Window

When clicking on [Arch. Level Estimation], [Estimation Confirm] window opens to select the estimation system. (See Figure 27)

Use the pull down menu to select the estimation system and click on [Execute] button, architecture level estimation is performed and results are shown as in Figure 28.

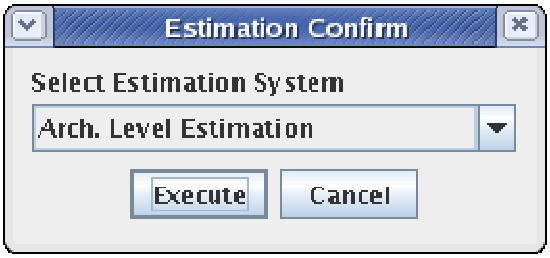


Figure 27 Estimation System Selection Window

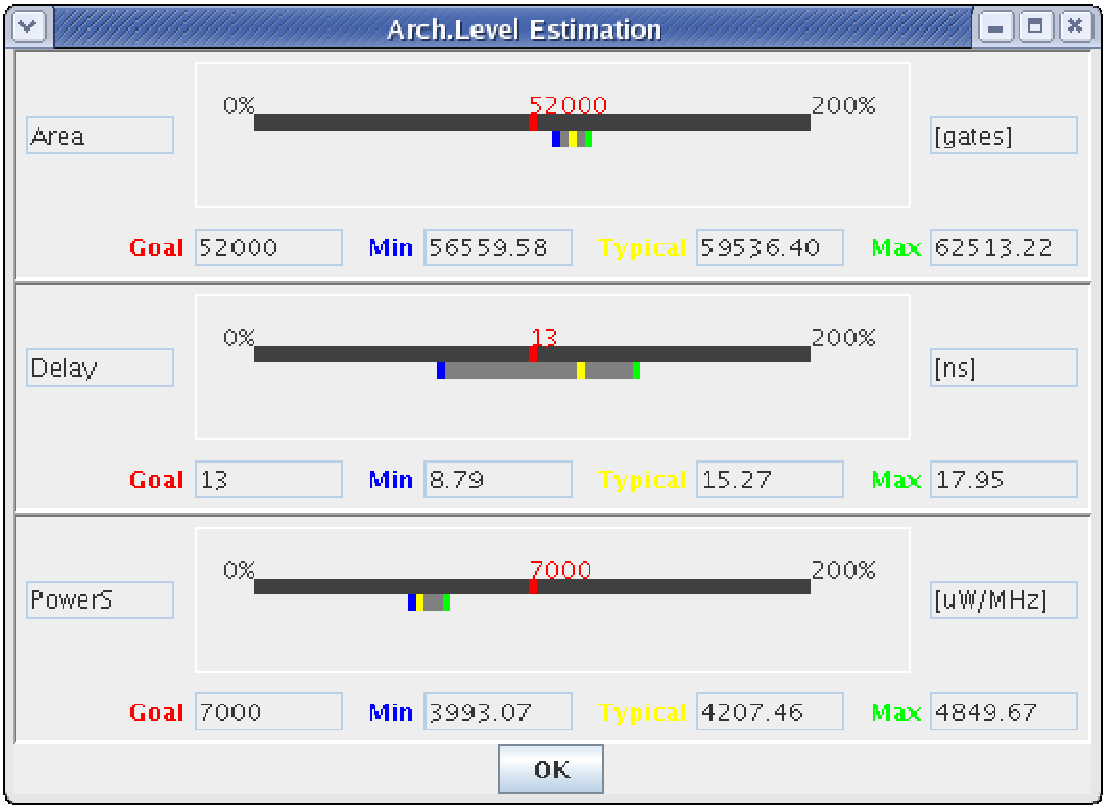
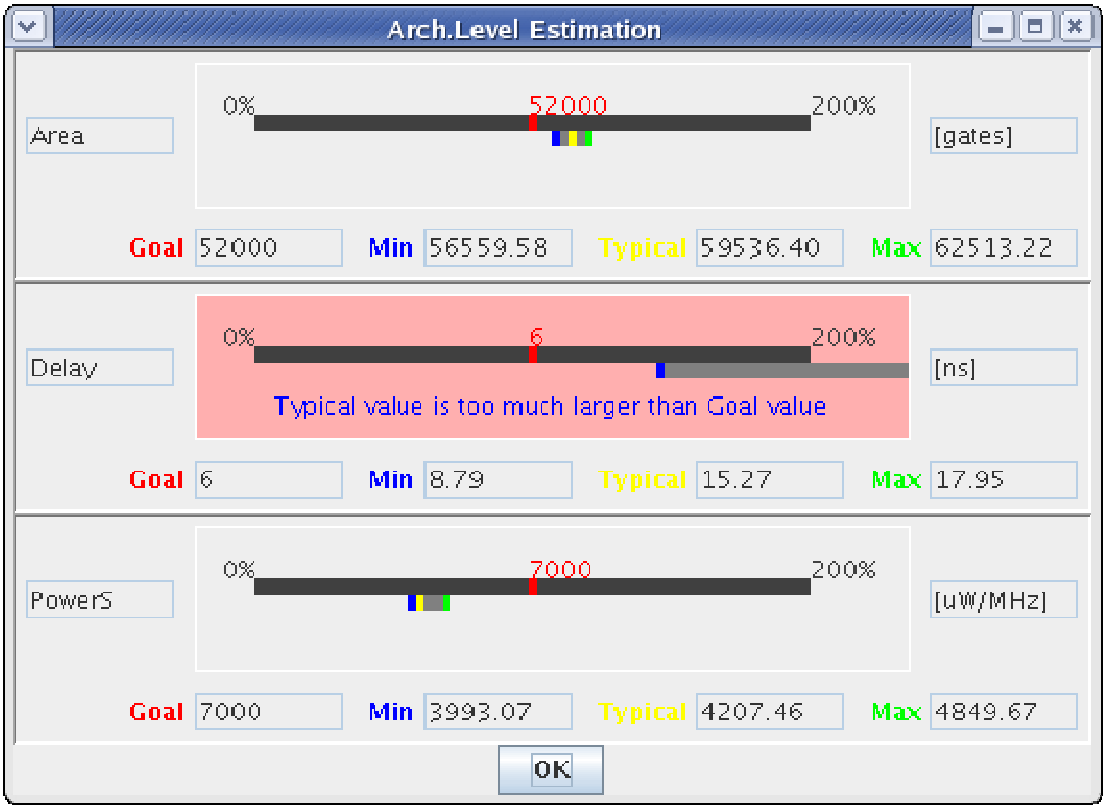


Figure 28 Estimation Results Window

The processor core [Area], maximum path delay [Delay] and power consumption [Power S] are computed by the estimation engine. The set point which is set with [Design Goal & Arch. Design] is also displayed on a (%) scale showing the minimum [Min], standard [Typical], and maximum [Max] estimates calculated by the estimation engine,

When the estimate values are substantially larger or smaller than the set point which is set with [Design Goal & Arch. Design], a warning will be indicated.



8. Assembler Generation

When clicking [Assembler Generation] in Main Window, [Generation Confirm] window (Figure 29) opens. In this design step, the description for the meta-assembler can be automatically generated.

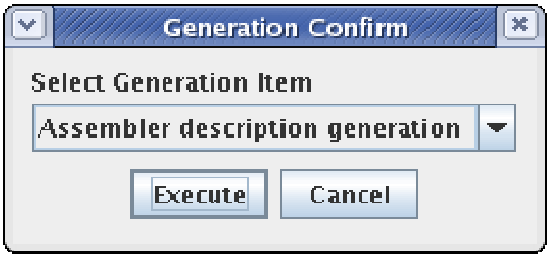


Figure 29 [Generation Confirm] Window

In [Generation Confirm] window, click on [Execute] button to generate the assembler description file.

After the meta-assembler description generation is completed, the window in Figure 30 would open. If there are no errors, click on [OK] button to return to main window.

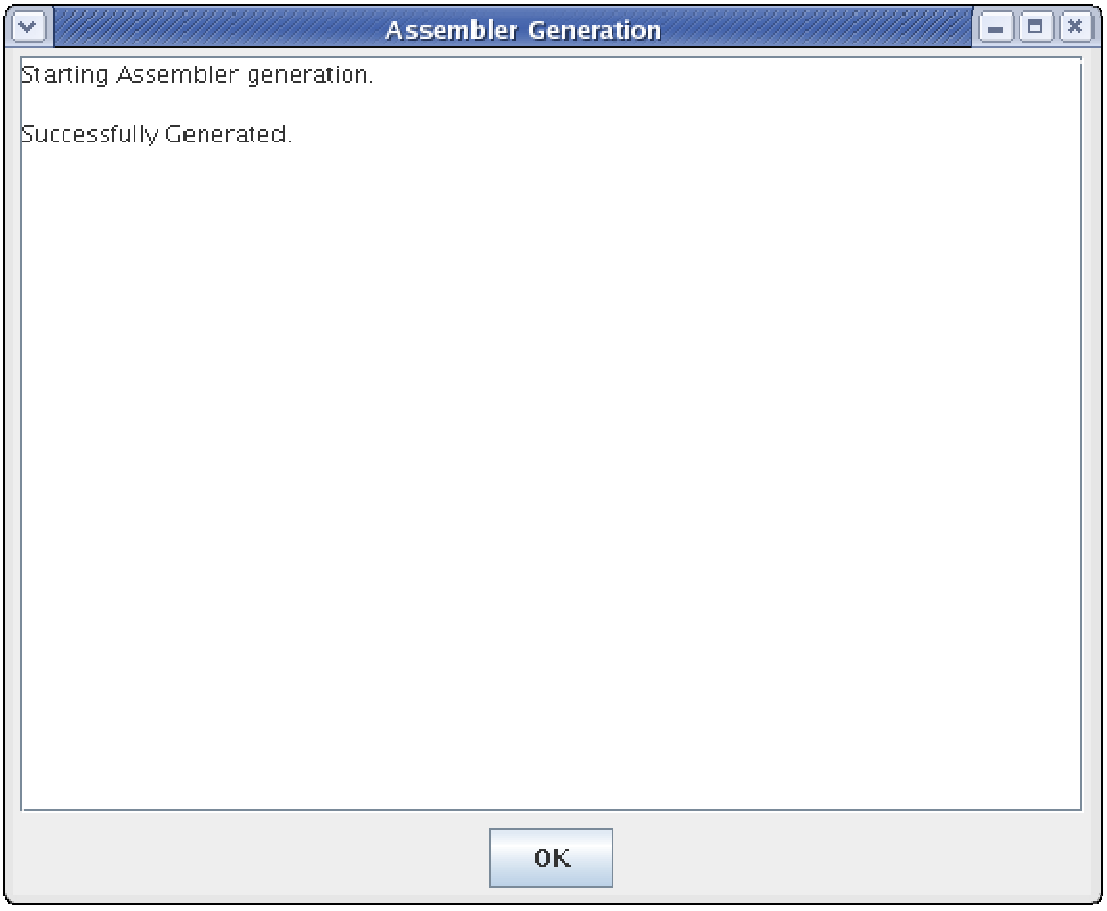


Figure 30 Meta-Assembler Generation Complete Window

When **ASIP Meister** successfully generates the description for the meta-assembler, the description is created under the "meister" directory in the current directory. For example, if you have a design data file named "model.pdb", the description file for the meta-assembler will be "model.des".

9. Micro Op. Description

When clicking [Micro Op. Description] button, [Micro Op. Description] window (Figure 31) opens. In this window, you can describe the behavior of all instructions (in each pipeline stage) and interrupts (exceptions) defined in [Instruction Definition] window. The behavior description consists of declaration of variable section and description for each pipeline stage.

The features of Micro Op. description in **ASIP Meister** are as follows;

- Insensitive to changes in used resources or pipeline operation
- Description of instruction functionality and behavior
- Description of H/W operation when an external interrupt is generated.

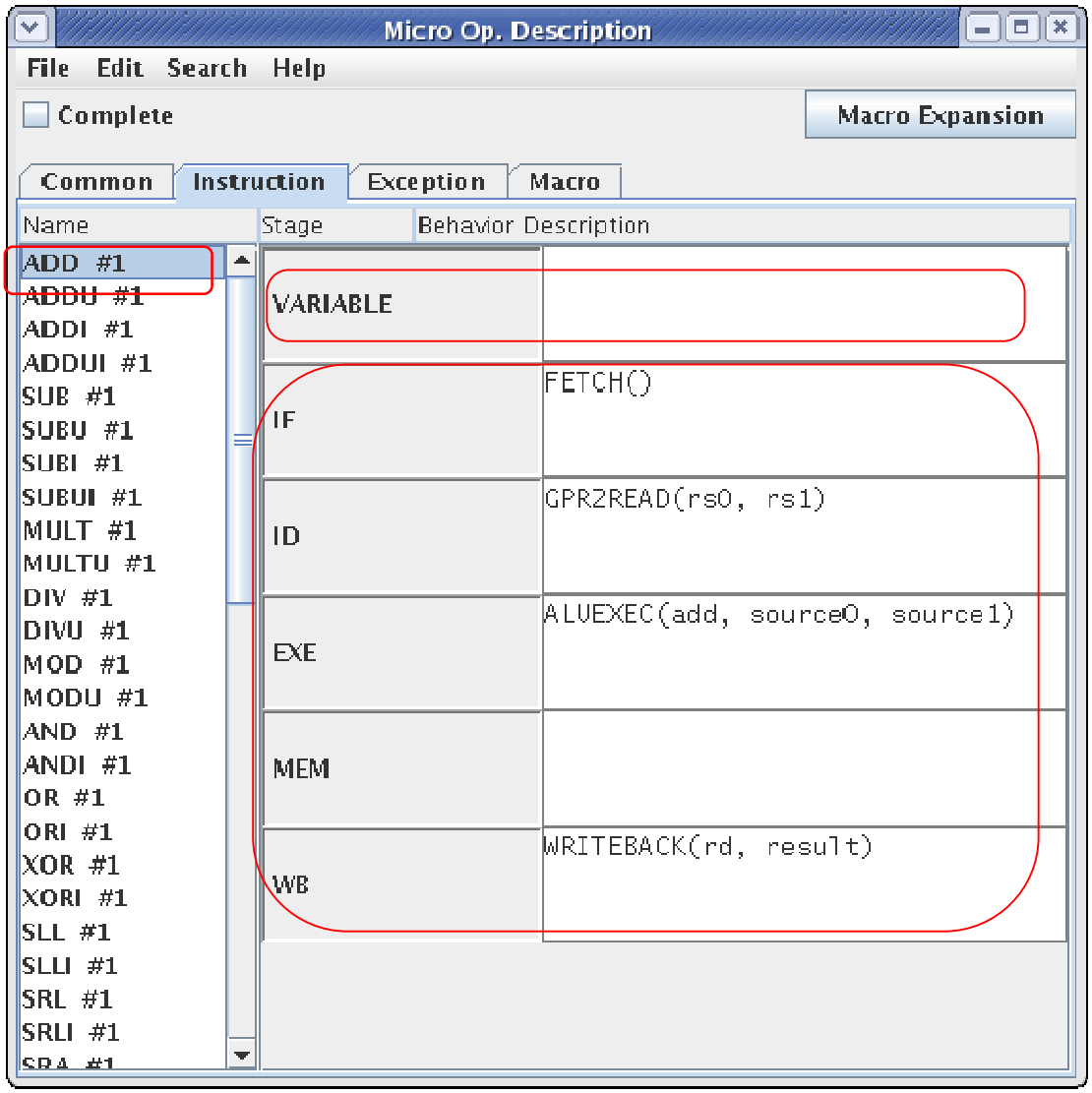


Figure 31 [Micro Op. Description] Window

"Micro-operation Description Coding Guide" describes the detail of the syntax and behavior of the micro operation description language. Please refer to the document for further details on the topic.

[Micro Op. Description] window has four tabs; [Common], [Instruction], [Exception] and [Macro]. Select [Instruction] to describe the behavior of the instruction, [Exception] to describe the behavior of interrupt and [Macro] to define the macro.

＜Note＞The current version does not support [Common].

In [Design Goal & Arch. Design] phase, the limits in [Instruction Definition] changes according to the settings of [Processor Design] and [Use Compiler]. These limits will be explained next.

- [New Design] : No limits.
- [New Design with Base Processor]
  - When [Use Compiler] is set to "yes"
    - Concerning the base processor, instruction types modification or deletion are not possible.
    - Addition of New instruction types and addition of new instructions are possible.
  - When [Use Compiler] is set to "no"
    - No limits
- [Base Processor Design] : No limits

9.1 [Micro Op. Description] Window *(Instruction Operation Definition)*

To describe the behavior of instructions, please click [Instruction] tab in [Micro Op. Description] window and [Micro Op. Description] window will show the description of the instructions' behavior. [Name] column lists the instruction names defined in [Instruction Definition] step. If you select the instruction name from the list, its behavior is displayed in the right side of the window.

[Stage] column includes [VARIABLE] as a first item and the stage names defined in [Design Goal & Arch Design] step. The variables used in several stages must be defined in [VARIBALE] row and local variables used in one stage can be defined in each stage description.

＜Note＞ Refer to [Micro-operation Description Coding Guide」

When writing Micro Op. description, it is possible to use [Copy], [Cut], [Paste], [Undo], [Redo], [CopyDescription] (Micro operation description copy) and [PasteDescription] (Micro operation description paste) from [Edit] menu.

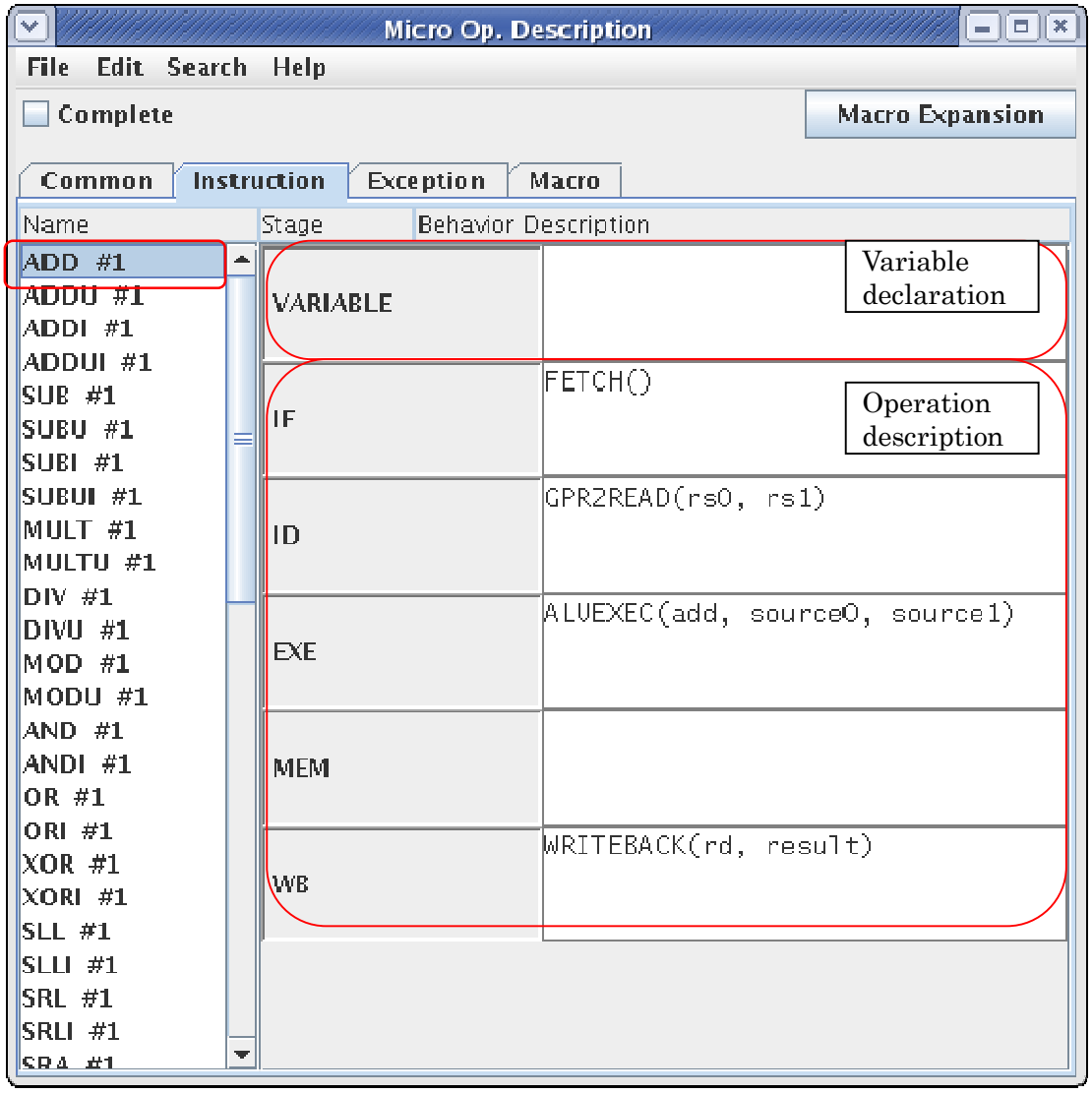


Figure 32 [Micro Op. Description] Window

9.2 [Micro Op. Description] Window *(Interrupt Operation Definition)*

To describe the behavior of interrupts and exceptions, click [Exception] tab in [Micro Op. Description] window and the window will show the description of behavior of the interrupts and exceptions (Figure 33).

[Name] column lists the interrupt and exception names defined in [Instruction Definition] step. If you select the interrupt (exception) name from the list, its behavior is displayed in the right side of the window. [Stage] column includes [VARIABLE] as the first row and [STAGE 1]. Please define variables in [VARIABLE] field and the behavior of the interrupt in [Behavior Description] field of [STAGE 1].

＜Note＞ Refer to [Micro-operation Description Coding Guide]

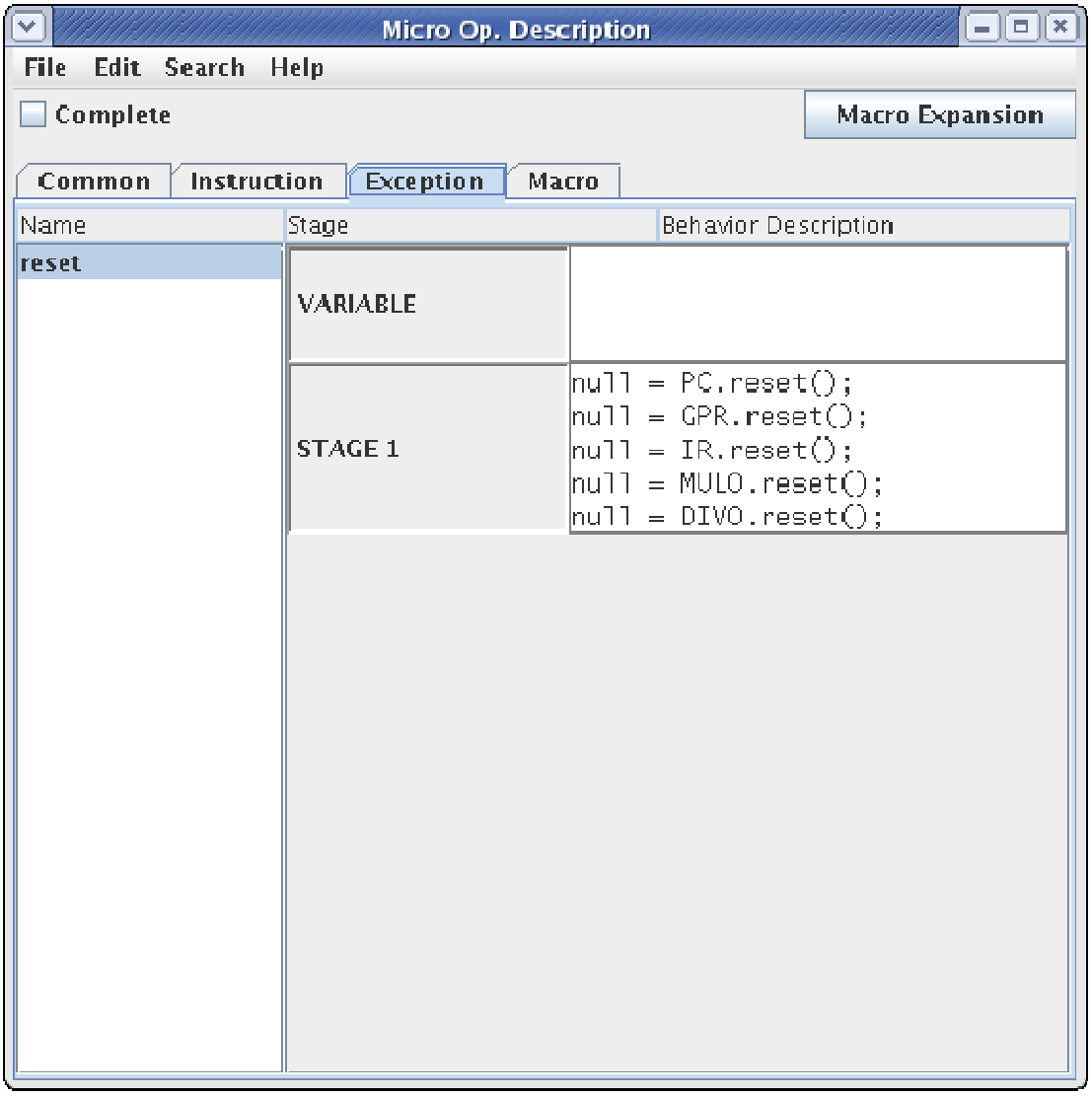


Figure 33 [Micro Op. Description] Window

9.3 [Micro Op. Description] Window *(Macro Definition)*

You can simplify the micro behavior description by defining the macro for the behavior that is commonly used among multiple instructions.

To define new macros, please click [New Macro] button in the window and [New Macro Confirm] window will open (Figure 34) then press [OK] button.

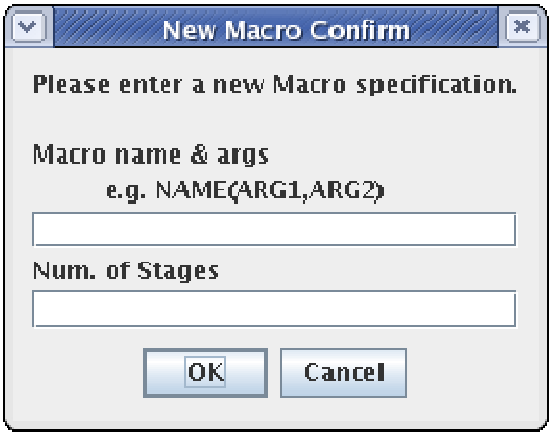


Figure 34 [New Macro Confirm] Window

Because [Micro Op. Description] window opens, the description is similar to the instruction Micro Op. description.

＜Note＞ Refer to [Micro-operation Description Coding Guide」

Table 16 [Micro Op. Description] Window

Parameter	Description
Macro name	Macro Name
args	Arguments
Num. of Stages	Execution stage number
Behavior Description	Micro operation description

9.3.1 Macro Expansion Rules

In the Micro Op. description of instructions, when you call a macro it can be expanded. Macro calling format is shown below

Macro Name(Argument 1, Argument 2, …, Argument n)

When a macro is expanded, the following description parts can be added:

- Declaration of variables common to instructions in all pipeline stages.
- Declaration of variables within and after the stage where the macro is called.
- Processing part within and after the stage where the macro is called.

Which stage the macro will be expanded, after the stage which calls a macro depends on the stage number which called the macro. If the stage number is 1, the instruction variable declaration part will be expanded in the stage which called the macro. If the stage is 2, the instruction variable declaration part will be expanded in the stage which called the macro and the following stages.

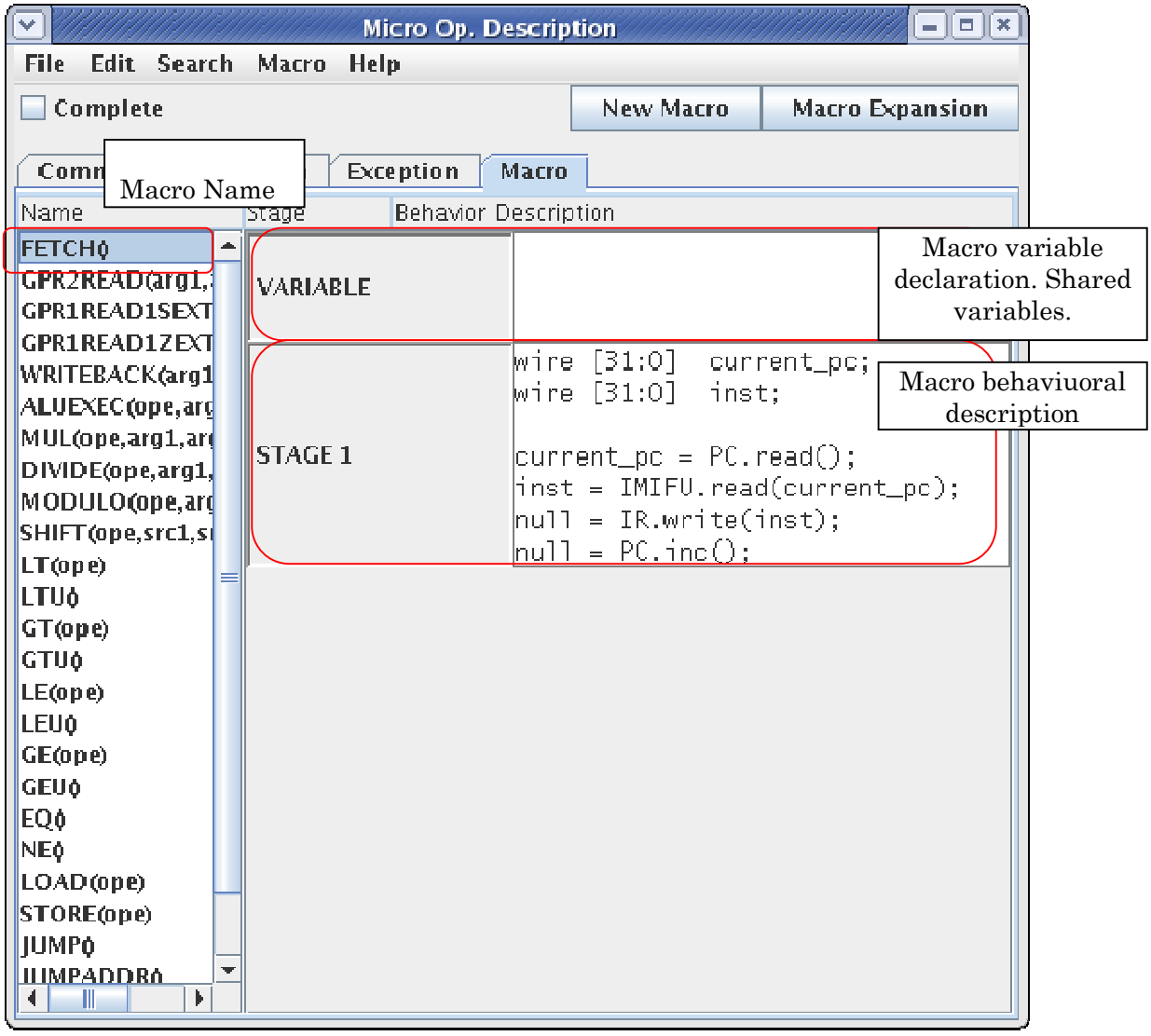


Figure 35 [Micro Op. Description] Window

9.4 [Macro Expansion]

When the macro is used to describe the behavior of the instruction, interrupt, and macro, **ASIP Meister** has a function to let you check the macro expanded description. To check the macro expanded description, press [Macro Expansion] button. This expands the macro to display its details. Argument strings will be replaced with actual argument values.

10. HDL Generation

By clicking on [HDL Generation] in Main Window, [Generation Confirm] window (Figure 36) opens. In [HDL Generation] design step, the HDL description of the processor can be automatically generated. This includes the HDL description of the simulation model used to validate the functionality of the designed processor core and that of the logical synthesis model.

The HDL description of each resource is described with the abstraction level specified in [Description Style of HDL] settings of [Resource Declaration] step.

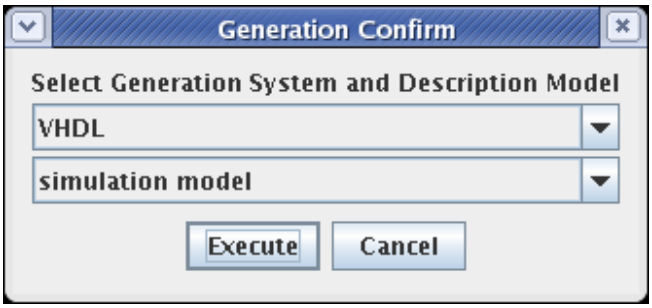


Figure 36 [Generation Confirm] Window

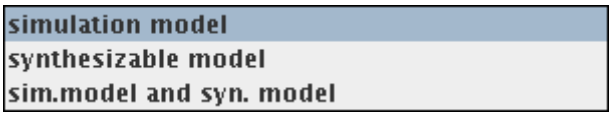


Figure 37 [Generation Confirm] Window

Please select the generation model from the pull down menu and click on [Execute] button to automatically generate the HDL files. The details of each engine are as follows;

1. [simulation model]: Generates the simulation model only.
2. [synthesizable model]: Generates the logical synthesis model only.
3. [sim. model and syn. model]: Generates both simulation model & logical synthesis model.

After HDL description generation is completed, the window in (Figure 38) opens. If there are no errors, click on [OK] button to return to main window.

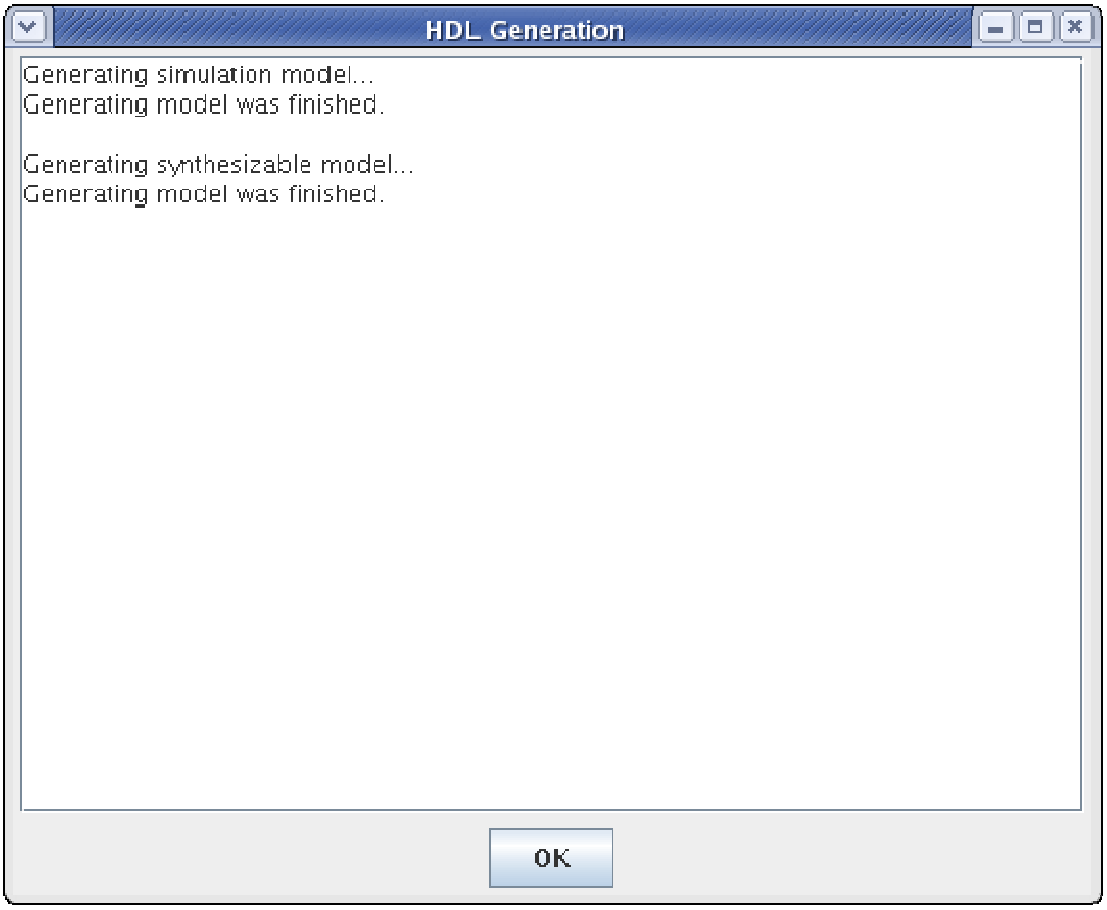


Figure 38 VHDL Description Complete Window

When **ASIP Meister** generates the HDL description files successfully, the directory that contains the description files is created under the "meister" directory in the current directory. For example, if you have a design data file named "model.pdb", the description for the simulation model is stored under "meister/model.*lang*.sim/" directory, and the description for the logical synthesis model is stored under "meister/model.*lang*.syn/" directory. "*lang*" is "VHDL" or "Verilog" according to the choice.

The VHDL description files has the extension ".vhd". The VHDL file of the top module is named as the entity name defined in [Interface definition] with the extension ".vhd". When the selected output language is VHDL, the generated files would be as shown in Figure 39 and Figure 40.

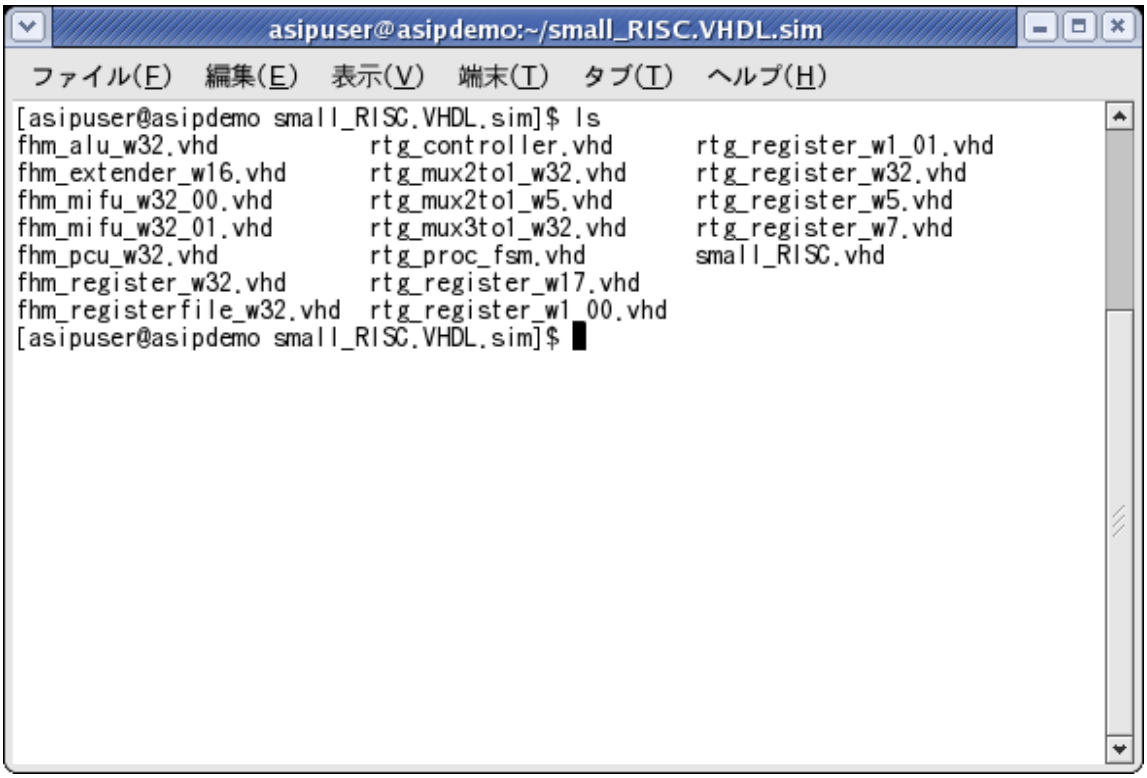


Figure 39 (Example) Generated Directory File List of Simulation Model

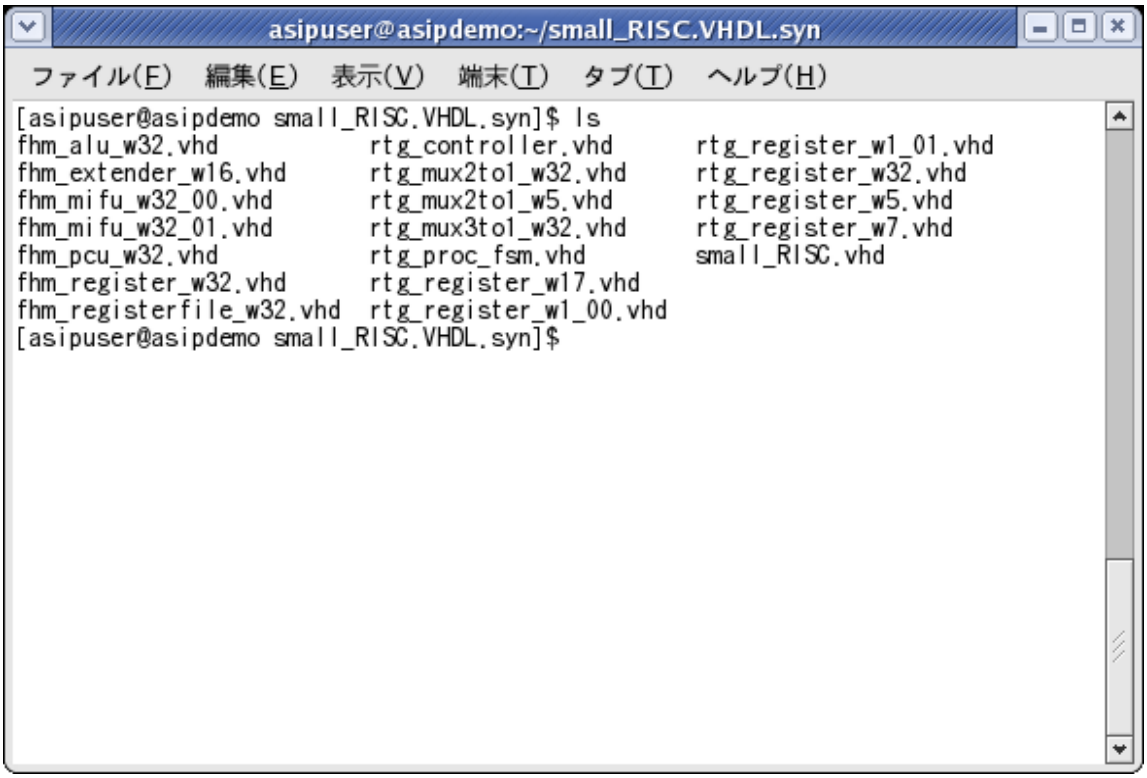


Figure 40 (Example) Generated Directory File List of Synthesis Model

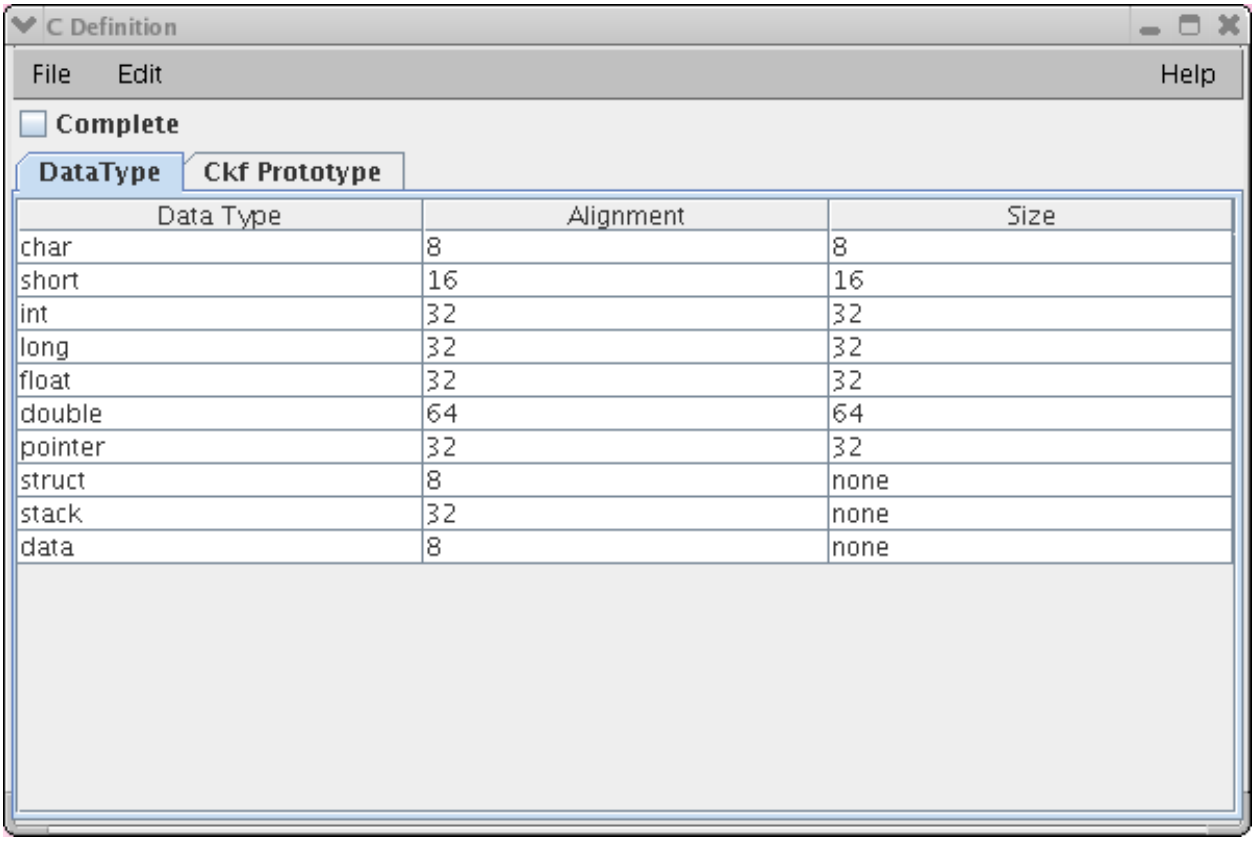
11. C Definition

If you click on [C Definition] in the main window, a [C Definiton] window like the one in Figure 41 will appear.

The **ASIP Meister** complier generation supports the base processor ***Brownie***. In order for the compiler to generate instructions that are extended from the ***Brownie*** processor, some definitions are necessary.

Using [C Definitions], you can define the C description specifications that are supported by the **ASIP Meister** compiler and the C description variables that support the instructions extended from the ***Brownie*** processor.

<Reference> Brownie Specifications



Data Type	Alignment	Size
char	8	8
short	16	16
int	32	32
long	32	32
float	32	32
double	64	64
pointer	32	32
struct	8	none
stack	32	none
data	8	none

Figure 41 [C Definition] window

In [C definition] at the [DataType] tab, you can define the memory usage constraints and the bit width of the C language variable different models. However at the [Ckf Prototype] tab, you can define the variables that support the extended instructions.

11.1 [DataType] Tab

At [DataType] tab (Figure 41), the model of the used variables are described in the **Data Type** column, and it includes defining the memory usage domain constraints (Alignment), and the bit width for the used variables,

Table 17 DataTypes

Model Name	Usage	Default Value	
		Alignment	Size
Char	1 byte integer	8	8
Short	2 byte integer	16	16
Int	4 byte integer	32	32
Long	4 byte integer	32	32
Float	4 byte integer	32	32
double	8 byte integer	64	64
pointer	pointer	32	32
Struct	Object Composition	8	none
Stack	stack	32	none
Data	Initialization constant	8	none

Because the memory addressing is byte addressing, the memory area that can be used changes according to the Alignment. In case Alignment is 8, all the address can be used. While in the case of 16, 32 and 64, the beginning address of the variable can only be incremented by multiples of 2,4 and 8 respectively.

＜Note＞In this version, the information in [DataType] tab cannot be modified.

11.2 [Ckf Prototype] Tab

At [Ckf Prototype] tab, you can declare the extended instructions that support the C language defined variables. Figure 42 shows the [Ckf Prototype] of the [C Definition] window.

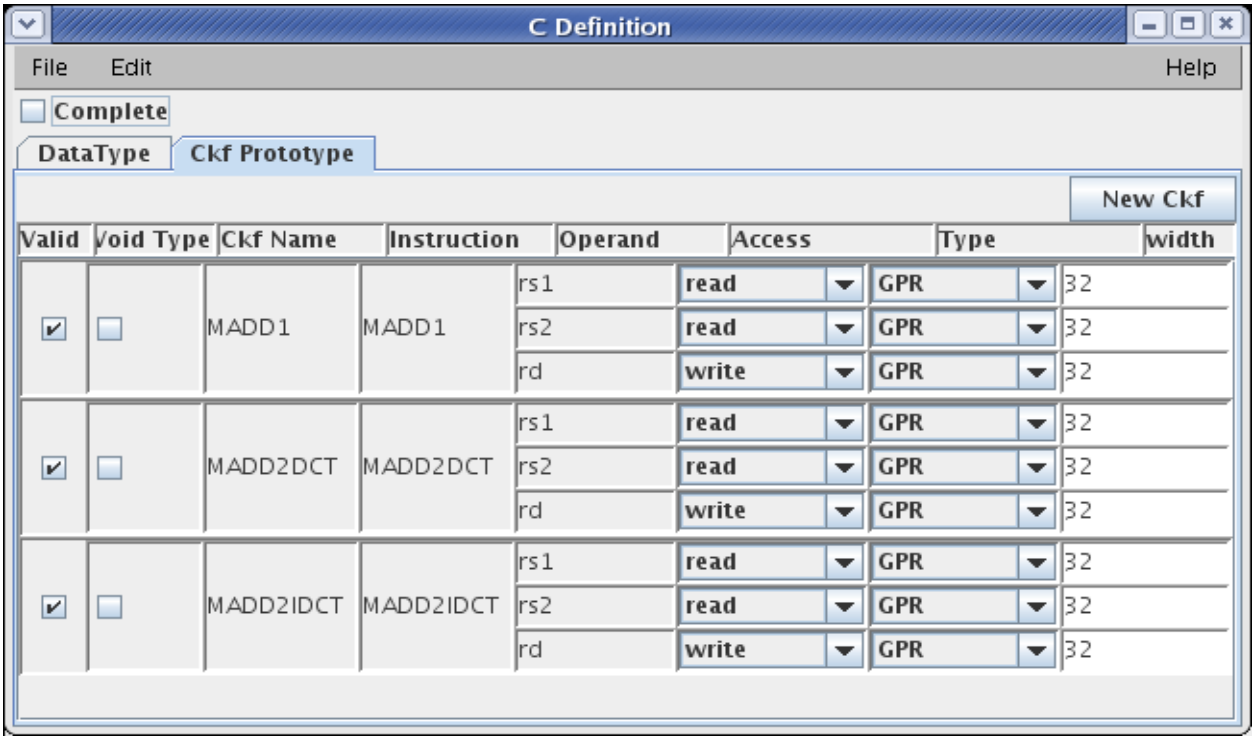


Figure 42 [Ckf Prototype] Tab

In the initialization state, Ckf is not registered. By selecting [New Ckf], you can register Ckf. After you select [New Ckf], the window shown in Figure 43 will appear.

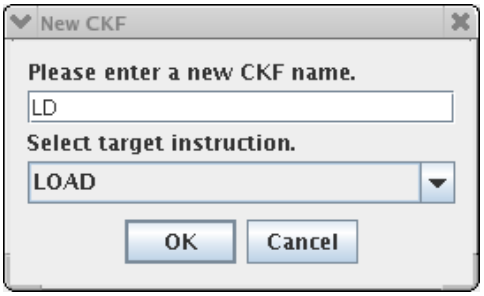


Figure 43 [New CKF] Window

In [New CKF] window, input the variable name, and select the target instruction that corresponds to it and then click OK. Hence the registration of the Ckf. In Table 18, the information of Ckf is shown.

Table 18 CKF Options

Item	Description	Default Value
Valid	Decides usage of Ckf	Checked
Void Type	Return value existance	Unchecked

Ckf Name	Ckf name	Same name set at [New CKF]
Instruction	Instructions that correspond to Ckf	Instructions slected at [New CKF]
Description of each operand		
Operand	Instruction operand	[Instruction Definition]で定義
Access	IO to/from the variable	Read
Type	Operand model	GPR and singed
Width	Operand bit width	Defined in [Instruction Definition]

Valid : Decides usage of Ckf

Decide whether to use Ckf or not Ckf. If you do not check [Valid], the generated compiler willnot support the Ckf.

Void Type : Return value existance

Decide whether there is a return value or not. For one extended instruction, if the ”Access” field is set to ”Write”, and its ”Type” is set to ”GPR” or if only one item in ”Operand” has an “Access” set to ”Write” then you do not check this checkbox. In all other cases you have to check this checkbox. In case it is checked, the output operand in the CKF is dealt with as a by reference operand. Examples of a definition of CKF called Ckf1, with both cases where ”Void Type” is not checked and is checked are demonstrated in the following C language description.

< case is not checked>

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

< case is checked>

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

<Note> In the current version, you have to include ”_ _builtin_brownie32_” before the extended instruction.

Ckf Name : Ckf name

Shows the name of the selected Ckf. The same name set in [New CKF] is shown here. The names of the already registered Ckf cannot be modified. In the case you want to modify the name, once the target Ckf is deleted, you can make a new Ckf entry with the new name.

Instruction : Instructions that correspond to Ckf

Shows the instruction that corresponds to the selected Ckf. Like Ckf Name, after setting itcannot be modified. In the case you want to change the settings, you need to make a new Ckf entry.

Operand : Instruction operand

Shows the operands of the instruction. In the [Instruction Definition], all the Fields that have ”Operand” set in its ”Field Type” are shown here.

Access : Access direction of operands

Shows the access direction to and from the operand. When you set the Ckf input variables, choose ”read”, and when you set the Ckf output variables, choose ”write”.

Type : Operand type

Shows the operand type. At [Instruction Definition], if "Addr mode" field is set to "Reg Direct", this item is fixed to "**GPR**".However if "Addr mode" field is set to "Immediate data"、then you can select either "**signed**" or "**unsigned**".

Width : Operand bit width

Shows the bit width of the operand. In the case that Type is set to "**GPR**"、then the bit width of the GPR resource is used by default, however, if in the case that Type is set to "**signed**" or "**unsigned**"、then the bit width of the Field set at [Instruction Definition] is used by default.

Delete CKF(menu)

Use to delete an existing entry of Ckf. Only the Ckf that is Marked will be deleted.

Mark(menu)

Use to select Ckf. After you Mark a Ckf it will become blue, and commands such as "**Delete**"will become valid to use.

Mark Clear(menu)

Remove a Mark off of a Ckf.

<Note> The Micro Op. of Brownie extended instructions should be already defined.

12. Compiler Generation

At [Compiler Generation] stage, the processor compiler and the Binutils can be generated. The generated compiler and Binutils supports the Ckf defined at [C Definition]. If you select [Compiler Generation], a [Generation Confirm] window will appear as shown in Figure 44.

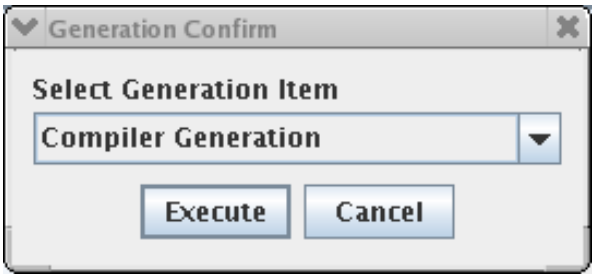


Figure 44 [Generation Confirm] Window

At [Generation Confirm] window, choose one of the displayed items in the drop down list that will next be automatically generated. After you click [Execute] button, the compiler related components automatic generation will start.

- Input Description Generation : Compiler extended description generation
- GNU Tools Generation : Using the compiler extended disxription, the compiler and binutils are generated.

The processes involved in the [Compiler Generation] phase are as follows.

1. Input Description Generation select and run.
2. GNU Tools Generation select and run.

If you select [Input Description Generation] from the drop down list, the compiler extended description will be generated. After the generation terminates successfully, if the design data file was called "model.pdb", a new directory called "model.swgen" will be generated inside "meister" directory, and inside this directory the compiler extended description is generated in a file named "model.xml".

<Note> Based on the compiler extended description, the compiler and binutils will be generated. Hence before you choose [GNU Tools Generation] for generation, you have to choose [Input Description Generation].

＜Note＞ The compiler generation will not terminate successfully if the ASIP_APDEV_SRCROOT environment variable was not set with the installation path of the application program development generation tool during the installation phase.

After you select [GNU Tools Generation] and the generation terminates successfully, the compiler and the Binutils will be generated in the directory path set in the environment variable ASIP_GNU_INSTALL_DIR. If the environment variable ASIP_GNU_INSTALL_DIR was not set, the compiler and the Binutils will be generated by default in a folder

called ”model.swgen” in the meister directory.

Using the generated compiler, the following demonstrates a method for cross compiling a C program sample file called “sample.c”.

First, you have to add the path**”model.swgen/bin”** to the $PATH environment variable.

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

Executing the command shown above will lead to the generation of the processor object code sample.

<Note 1> The link script “link.sc”, contains the memory mapping description information. A reference example is shown below.

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

Viewing the description above, the program is allocated starting from address 0x0, while the data is allocated starting from address 0x100000000.

<Note 2> When you want to write the extended instruction description in C code, you have to add ”__builtin_brownie32_” to the ckf definition. An example is demonstrated in the following.

Ex.) : ckf : MAC, case of 3 parameters and 1 output

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

The extended instructions cannot be successfully compiled if the description does not include ”__builtin_brownie32_”.

ASIPSolutions

Appendix

This appendix describes the following items.

1. Supported processor models for design
2. Use limitation
3. How to write the micro operation description
4. Reserved words
5. External interface specification
6. How to describe interrupts
7. PAS meta-assembler usage
8. Registered resource models and their parameters

1. Supported Processor Models For Design

The available processor models that can be designed with **ASIP Meister** are listed below.

- Pipeline structure
- Fixed length instruction
- Single Cycle instruction fetch
- Multi-cycle instruction
- In-Order completion
- Interlocks for structural hazards (Interlocks for data hazard are not supported)
- Harvard architecture
- External interrupts (single level)
- Internal interrupts (exception)

Please contact the supporting group for details of other models not stated above.

1.1 Restricted Items

- An interrupt cannot be generated while processing an instruction within a delayed branch slot.
- Operation of stages before the decode stage must be common in all instructions.
- Divergance stage (The stage where the PC value is written) is only one.
- The processor state must not change before the divergence stage.
- The processor state must not change before the stage where internal interrupt occurs.
- Multi-cycle operation (e.g. multi-cycle fetch) must not be performed before a divergence stage.
- External interruption is one at most.
- The reset interruption by all means is necessary.

2. Use Limitation

ASIP Meister does not support multi-user feature. It is developed for single user only. Therefore, when used by multiple users, each user must install his/her own **ASIP Meister** on the computer.

3. Micro Op. Description Method

3.1 Basics of Micro Op. Description

In micro-operation description, you can describe the following behaviors for each pipeline stage by using the resources specified in “Resource Declaration”.

- Data transfer
- Processing using a resource (operation, read, write etc.)

Data transfer is denoted by the sign “=”. It is similar to substituting the right side for left side in general programming language. Processing using a resource is denoted by <resource name>“.”<function name>"("<input lining>")". For the <function name>, you can use functions described in "Function Set" displayed in "Resource Declaration" window. The following is a part of "Function Set" in model alu.

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

----------<Some lines are omitted>----------

  /** 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;
      }
    }
  }

----------<Some lines are omitted>----------

}
```

For example, if you use the function add of resource ALU instance, you can describe it as

shown belwo using previously declared variables (result, flag, a, and b).

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

where a and b are inputs to ALU and result and flag are outputs from ALU. The input to a resource is considered similar to argument when calling function in programming language.

The order of the input to resource is corresponding to the order of the port name that appears in "input" of "Function Set". When the output from a resource is more than one, variables are listed with the sign "< >". The order of the output is corresponding to the order of the port name that appears in "input" of "Function Set".

3.2 Syntax Elements of Micro Op. Description

3.2.1 Variable

In micro operation description, a variable is used to transfer data. The Variable should be previously declared in a variable declaration part. About the format of variable declaration, please refer to BNF in this appendix. The variables in micro operation description are treated as follows.

- A Variable declared for each instruction
  This variable is a variable used in several pipeline stages. Namely, it works as a pipeline register.

- Variable declared within a pipeline stage
  This variable is a variable used only in the stage where it is declared. Namely, it works as a signal line to transfer data between resources.

3.2.2 Binary Constant

The available constant in micro operation description is only a binary constant. A binary constant, is a bit literal denoted by 0 or 1 surrounded with " ' " and a vector literal denoted by more than one occurrence of 0 or 1 surrounded with " " ". A constant of 1 bit is indicated with the bit literal and a constant of more than 1 bit is indicated with the vector literal.

3.2.3 Concatenation

Concatenation of bits is denoted by "< >". Only concatenation of variables is available. For example, if you wish to use a variable d of 9 bits by concatenating a variable a of 3 bits, a variable b of 2 bits and a variable c of 4 bits, you should describe as follows.

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

where the most significant bit of the variable d becomes the most significant bit of the variable a and the least significant bit of the variable d becomes the least significant bit of the variable c.

3.2.4 Bit Operator

There are three kinds of bit operators AND, OR, INVERT. The operand must be a variable.

- AND operation of bits
  The operator "&" is used.

  ```
  a = “010”;
  b = “110”;
  c = a & b;
  ```

  In the above example, the value of c becomes "010".

- OR operation of bits
  The operator "|" is used.

  ```
  a = “010”;
  b = “110”;
  c = a | b;
  ```

  In the above example, the value of c becomes "110".

- INVERT operation of bits
  The operator "~" is used.

```
a = “010”;
b = ~a;
```

  In the above example, the value of c becomes "101".

3.2.5 Relational Operator

A relational operator is used to compare a variable with a given constant. There are two kinds of relational operators as follows.

- Equal sign

```
b = a == “01”;
```

  In the above example, the value of b becomes '1' if the value of a is "01" , and it becomes '0' otherwise.

- Unequal sign

```
b = a != “01”;
```

  In the above example, the value of b becomes '0' if the value of a is "01" , and it becomes '1' otherwise.

3.2.6 Bit Range Specifying

A bit range specifying is used to specify a bit range of a variable to be extracted.

- 1 bit specifying case

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

  In the above example, the third bit of the variable a is substituted for a variable b. The variable b must be declared as a 1 bit variable.

- More than 1 bit specifying case

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

  In the above example, from the first to the third bit of the variable a are substituted for a variable b. The variable b must be declared as a 3 bit variable.

3.2.7 Conditional Operation

A conditional operation is described as follows.

- Conditional data transfer case

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

  This is similar to a ternary operator "?:" in C language. In the above example, the value of a is substituted for a variable result if the value of the conditional variable condition is '1' and the value of b is substituted for a variable result if the value of the conditional variable condition is '0' . A conditional variable must be a 1bit variable.

- Conditional execution of a resource function case

The following example shows that a write function is executed with data set for the input of the resource REG if the value of the conditional variable condition is '1'. A conditional variable must be a 1 bit variable.

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

3.3 Points to Notice in Micro Op. Description

Points to notice in micro operation description are shown as below.

- Operation of stages before the decode stage specified in "Design Goal & Arch. Design" must be common in all instructions.

- It is an error when there exists a combination of instructions which generate resource competition.
- The name of the operand defined by the instruction type definition is treated as a declared variable in an instruction based on this instruction type.
- There must be one null line between a variable declaration part and a micro operation description part.

3.4 BNF of Micro Op. Description

```
< variable declaration part > ::= { < variable declaration > }
< variable declaration > ::= < wire declaration>
< wire declaration > ::=   'wire' [<bit range specifying>] < wire name> ';'
< wire name > ::= < identifier>

< micro operation description part > ::= { < micro operation description> }
< micro operation description > ::= < stage variable declaration part> <row of statement>

< stage variable declaration part > ::= < variable declaration part >
< row of statement > ::= { < statement > }

< statement > ::= < simple assignment statement> |
          < conditional assignment statement> |
          < conditional function execution statement>

< simple assignment statement > ::= <left side> '=' < right side > ';'
< left side > ::= < variable name > |<group of variable name> |   'null'
< right side > ::= < bit AND operation expression> |   < bit OR operation expression> |   < bit invert
operation expression> |   < comparison expression> |
          < set > |   < resource reference> |   < part selection expression> |
          < binary constant> |   < variable reference>

< bit AND operation expression> ::=     < variable reference> '&' < variable reference>
< bit OR operation expression> ::=     <variable reference> '|' <variable reference>
< bit invert operation expression> ::=     '~' <variable reference>
< comparison expression> ::=     <variable reference> < relational operator> < binary constant >
< relational operator > ::= '==' |'!='

<variable reference> ::= <variable name>
<variable name> ::= <wire name>
<variable name set> ::=     '<' <variable name> { ',' <variable name> } '>'
<set> ::=     '<' <variable reference> { ',' <variable reference> } '>'

< resource reference > ::=   < resource name> '.'     < function name> '(' [<parameter line>] ')'
< function name > ::= < identifier >
< parameter line > ::=     < parameter > { ',' < parameter > }
< parameter > ::= <variable reference>

< part selection expression > ::= <variable reference> < part selector>
< part selector > ::=     '[' <index> [':' < index >] ']'
```

< index > ::= < nonnegative integer>

< conditional assignment statement > ::= < left side > '=' '(' < conditional variable reference> ')' '?' <variable reference> ':' <variable reference> ';'
<conditional variable reference> ::= <variable reference>
< conditional function execution statement > ::= < left side > '=' '[' < conditional variable reference> ']' < resource function specifying> ';'
< resource function specifying > ::= < resource name > '.' < function name > '(' [<parameter line >] ')'

< identifier > ::= <English letter> { < English alphanumeric character> |'_' }
< English letter > ::= < English capital letter> |< English small letter >
< English capital letter > ::= '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'
< English small letter > ::= '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'
< English alphanumeric character > ::= < English letter > |<digit>
<digit > ::= '0' |< digit except 0>
< digit except 0> ::= '1' |'2' |'3' |'4' | '5' |'6' |'7' |'8' |'9'

< natural number> ::= < digit except 0> { < digit > }
< nonnegative integer > ::= '0' |< natural number >

< binary constant > ::= < bit literal> |< vector literal >
< bit literal > ::= ' ' ' < binary character > ' ' '
< vector literal > ::= ' " ' < binary character> {< binary character >} ' " '
< binary character > ::= '0' |'1'

< string> ::= "" { < arbitrary letter except line feed mark> } ""
< comment part> ::= < same as the syntax of comment in C/C++ （from // to the end of line, or between /* and */） >

4. Reserved Words

This section describes reserved words that cannot be used in **ASIP Meister** design files and VHDL files.

4.1 Reserved Words in ASIP Meister Design File

The following words cannot be used as data file name, storage name, resource instance name, instruction type name, instruction field name, instruction name, entity name of a target processor, port name, interrupt & exception name and variable name in micro operation description.

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	opecode	operand	out
parameter	plain_register	port	program_counter
register_file	register_read_stage	register_write_stage	reserved
reset_interrupt	reset_port	resource	saved_pc
stage	status_register	throw	top_module
wire			

4.2 Reserved Words in VHDL

The following words cannot be used as entity name of a target processor and port name.

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. External Interface Specification

The memory interface unit selected in Resource Declaration phase is implemented in the processor generated by **ASIP Meister**. This section describes a memory interface unit and an external interface specification.

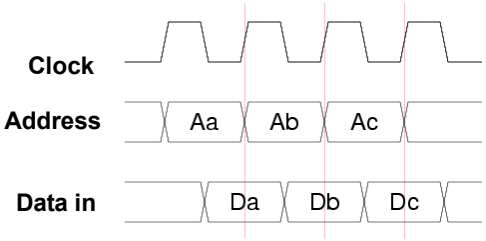
5.1 mifu Used as Instruction Memory Access Unit

Memory interface unit (mifu) supports both single cycle access and multi cycle access. In the case that mifu is used as a single cycle instruction memory access unit, you should define the following ports in Interface Definition phase.

Use As	*Direction*	*Attribute*
address	out	instruction_memory_address_bus
Data input to CPU	in	instruction_memory_data_in_bus

The operating condition is as follows.

- It is necessary to give the data specified by address to the data input port until the next rising of clock (refer to the next figure).



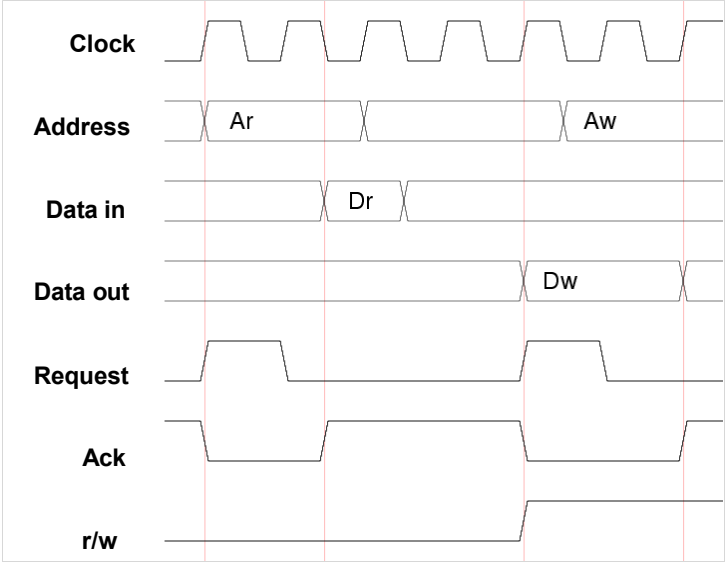
5.2 mifu Used as Data Memory Access Unit

In the case that mifu is used as a multi cycle data memory access unit, you should define the following ports in Interface Definition phase. The asterisk(*) of Attribute becomes instruction if instruction memory or becomes data if data memory.

Use As	*Direction*	*Attribute*
address	out	*_memory_address_bus
Data input to CPU	in	*_memory_data_in_bus
Data output from CPU	out	*_memory_data_out_bus
Access request from CPU	out	*_memory_request_bus
Acknowledge from memory	in	*_memory_acknowledge_bus
Write/Read control	out	*_memory_rw_bus
Access cancel signal	out	*_memory_cancel_bus

Operating conditions are as follows.

- The memory access should begin when the access request signal becomes '1'. As for address, the data input to CPU and write/read control signal, their values are maintained until the acknowledge signal is obtained (refer to the next figure).
- It is necessary to execute reading process when write/read control signal is '0' and writing process when write/read control signal is '1'.
- The access request signal becomes '1' during one cycle. When the request signal becomes '1' again before the acknowledge signal becomes '1', it is necessary to cancel the previous access and process access again.
- The acknowledge signal from memory should be '0' during the period when access is executed and the access request signal is '0' , and should be '1' during other period (refer to the next figure).
- It is necessary to cancel the access under processing when the access cancel signal becomes '1'.



Moreover, when the value of parameter access_width of dmifu is smaller than the value of bit_ width, the following ports are needed.

Use as	*Direction*	*Attribute*
Access bit number control	out	*_memory_access_mode_bus
Reading-case zero/sign extension control	out	*_memory_ext_mode_bus

Operating conditions are as follows.

- The access bit number control outputs the value of $(\frac{access\ bit\ number}{access\_width} - 1)$ in a binary number. For example, when the bit width is 32 bits and the access width is 8 bits, it is necessary to execute 32bit access for "11", 16bit access for "01" and 8 bit access for "00" respectively. When writing, it is assumed that effective data is stored in the lower bits of the data input signal.
- The zero/sign extension control signal shows whether the sign extension of data is executed when the reading bit number is less than the bit width. It is necessary to transfer the zero extension data for '0' and the sign extension data for '1'.

The function of mifu offers the address error as an output. When the following port is defined, the address error output at using of the function of mifu becomes effective. When the port is not declared, the address error output is indeterminate.

Use As	*Direction*	*Attribute*
Address Error Signal	in	*_memory_address_error_bus

The operating condition is as follows.

- The address error signal should become '1' when an address is in error.

6. Interrupt Description Method

6.1 Interrupt Spec. Presumed Interrupt Model

The following interrupt models are presumed in **ASIP Meister**.

- Corresponding to the following interrupts (ordered with high priority first)
  - Reset interrupt
  - NMI(Non-maskable interrupt)New!
  - Internal interrupt
  - External interrupt
    - Corresponding to masking of interrupt (internal interrupt, external interrupt)
    - Having an interrupt controller outside processor (supporting only one path for external interrupt)
    - Corresponding to multi interrupt from the software side

Table 19 shows all presumed interupts. In the figure, N is the pipeline stage number. When an interrupt whose priority is high occurs simultaneously with a low priority interrupt, the interrupt whose priority is low is cancelled, and the interrupt whose priority is high is processed.

Table 19 Interrupt Priority Precedence

Priority Level	Detection Stage	Name
High	N	Reset Interrupt
:	N	NMI(Non Maskable Interrupt)
:	N	Stage N Internal Interrupt
:	N-1	Stage N-1 Internal Interrupt
:	:	:
:	2	Stage 2 Internal Interrupt
:	1	Stage 1 Internal Interrupt
Low	1	External Interrupt

In Figure 45, the connection with an outside interrupt controller is shown. Supporting only one path for external interrupt, it is assumed to put an interrupt controller outside the processor core when several external interrupts are to be supported.

When an external interrupt is generated, the interrupt controller asserts the **interrupt** signal in figure and keeps the signal asserted untill the **catch** signal is raised. The processor will be notified with the interrupt signal assertion, and enters into the state of waiting interrupt processing.

In this state, the following instruction would be discarded but processing of all instructions inside the pipeline is awaited for completion. Then, the state of the processor moves to the state of interrupt processing. In this state, the processor will assert the catch signal for 1 cycle to notify the controller of the acceptance of interrupt. At the same time, it will jump to the address of interrupt handler, and from the next cycle, the handler will start execution. When the interrupt handler is completed, the processor will assert the end_of_interrupt signal for 1 cycle to notify the controller of the end of interrupt handling. When multiple external interrupts exist, a general purpose data bus can be used to distinguish the kind of interrupt and read the interrupt vector. Furthermore, the precedence of external interrupts and mask setting inside the interrupt controller can be accessed through the same general purpose data bus.

In case of reset interrupt and NMI interrupt, the interrupt handling will immediately be processed.

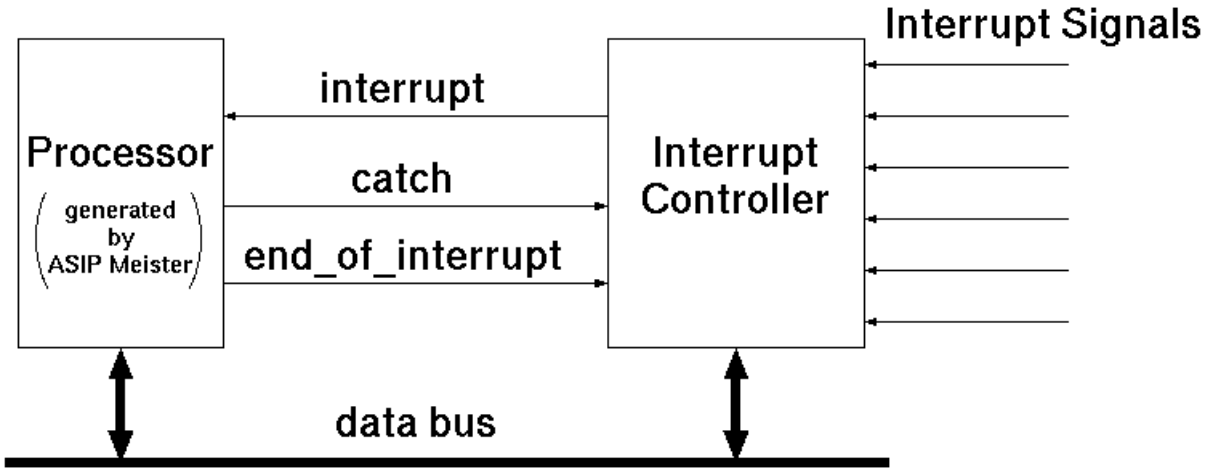


Figure 45 Interrupt Controller and Processor Connectivity

6.2 Internal Interrupt Outline

The timing chart when an internal interrupt is generated is shown in Figure 46. when detecting the internal interrupt signal in an instruction, the following instruction would be discarded and processing of previous instructions is awaited. The interrupt processing described by the micro operation description is executed according to the kind of the generated interrupt at the next cycle when the pipeline is flushed. It moves to the interrupt handler's processing (instruction **w - z** in figure) by jumping to an appropriate address during the interrupt processing. The figure shows the example of instruction **h** generating the decode error exception during cycle 5 in the second stage (**H** in figure). The interrupt processing of decode error exception (**D** in figure) is executed when the following instruction **i** is discarded by the exception generation, the execution of the previous instruction ends, and the pipeline is flushed. In the interrupt processing (**D** in figure), the address of instruction h where the internal interrupt is generated can be acquired.

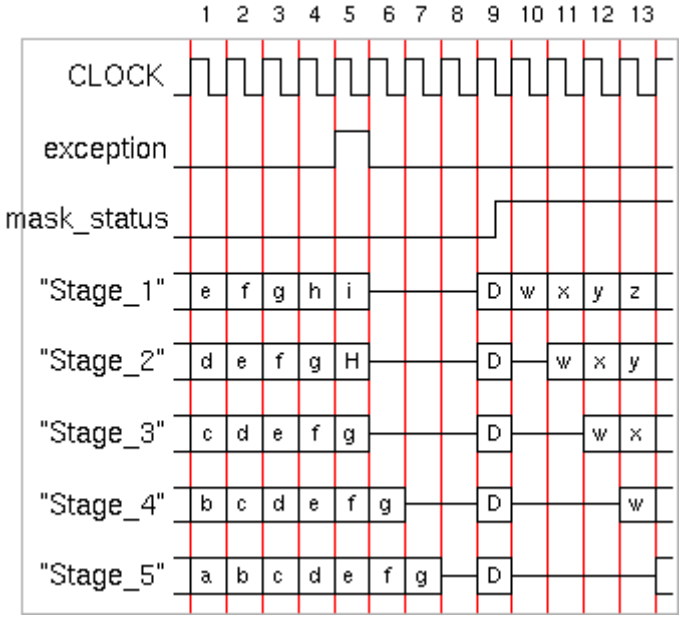


Figure 46 Internal Interrupt Timing Chart

6.3 External Interrupt Outline

The timing chart when an external interrupt is generated is shown in Figure 47. The fetch is stopped from the next cycle after the one when the external interrupt signal is detected. When detecting an external interrupt, the instruction at the first stage of the pipeline is discarded. The interrupt processing of the external interrupt described by the micro operation description is executed at the next cycle when the pipeline is flushed (It is the same timing as in the case that an internal interrupt is generated in the first stage).

The interrupt controller can be notified of the acceptance of the interrupt using the catch signal assertion by micro operation description. (The interrupt signal must be asserted at least until this cycle and should wait for catch signal assertion. If instructions whose processing is awaited to complete are branch or make another internal interrupt, the external interrupt handling is postponed. Therefore, the interrupt signal should be continued to be asserted until the assertion of the catch signal in order not to miss the external interrupt.).

In the interrupt processing, it moves to the interrupt handler's processing (instruction **w**-**z** in figure) by jumping to an appropriate address. The instruction **z** in figure is an interrupt handler end instruction, and asserts the **end_of_interrupt** signal in stage 3, then it jumps to the value of PC saved when the interrupt was generated and the instruction sequence that has been suspended by the interrupt is restarted. In the interrupt processing (**Cycle 8** in figure), since the address (of instruction **g**) when the internal interrupt was generated can be acquired, micro operation description which saves that address to some appropriate place should be written.

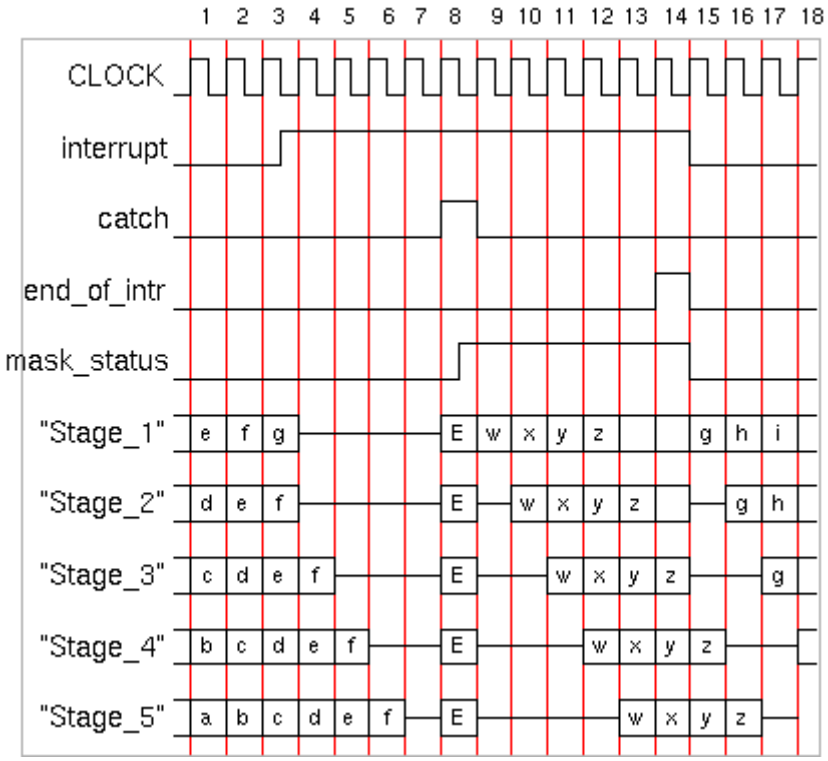


Figure 47 External Interrupt Timing Chart

6.4 NMI Interrupt Outline

The timing chart when NMI is generated is shown in Figure 48. When the NMI signal is detected at the clock edge, the interrupt processing of NMI is executed in the next cycle. The NMI signal should be asserted during one or two cycles, and if it is asserted during three cycles

or more, the interrupt processing of NMI is executed again. In the figure, all instructions c-g are discarded by the NMI signal asserted during fetching the instruction **g** (cycle 3) and executing instruction **c-f**, and the interrupt processing **N** of NMI is executed at the next cycle (cycle 4).

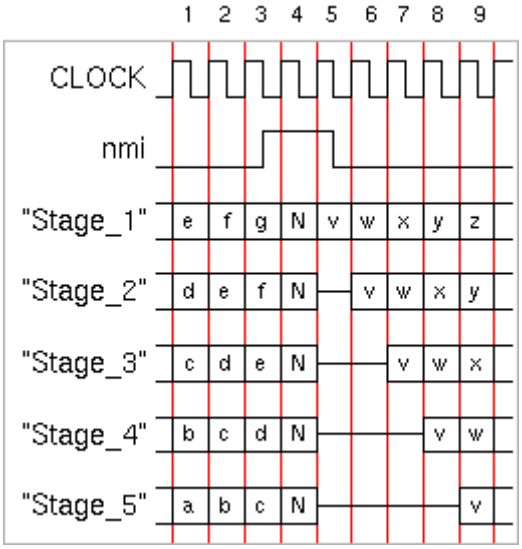


Figure 48 NMI Interrupt Timing Chart

6.5 Interrupt Spec. Input Procedure

In this page, the following interrupts are assumed in explanation,

- Reset interrupt
- NMI
- External interrupt
- Decode error interrupt（internal interrupt)
- Overflow interrupt of ALU (internal interrupt)

Moreover, masking will be done for an interrupt other than the reset interrupt and NMI.

To describe an interrupt, information is input according to the following procedure.

1. Setting of ports for interrupt and user definition port　(Interface Definition phase)
2. Wire resource declaration for user definition port (Resource Declaration phase)
3. Declaration of interrupt (Instruction Definition phase)
4. Declaration of interrupt end instruction (Instruction Definition phase)
5. Micro operation description of interrupt (Micro Op. Description phase)
6. Micro operation description of interrupt generation (Micro Op. Description phase)
7. Micro operation description of interrupt end instruction (Micro Op. Description phase)

The user definition port means a port other than for the interrupt input signal. In that case, it is necessary to declare a wire resource in the Resource Declaration phase (wire_in, wire_out, or wire_inout). Only when an internal interrupt is generated in the instruction, "Micro operation description of interrupt generation" is a necessary step.

6.6 Interrupt Input

6.6.1 Setting Interrupt and User defined Ports

The port for an external interruption is declared.

Attribute	Port Name	Direction	Signal Type
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" is an external interrupt signal, "NMI_IN" is an NMI input signal, "CATCH_OUT" is interrupt acceptance notification signal and "EOI_OUT" is an interrupt end signal.

6.6.2 Wire Resource Declaration for User defined Ports

It is necessary to create a wire resource corresponding to a user definition port.

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

Here, the name of the wire resource and the name of the user definition port should be made the same. If these are the same, the connection between the user definition port and the wire resource is automatically performed.

Moreover, it is necessary to create a resource corresponding to a mask register.

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

Here, Use as should be Mask Register. The resource specified with Mask Register can be selected at the following external interrupt declarations.

6.6.3 Interrupt Declaration

Five Types of interrupts can be decalred.

Reset Interrupt

Interrupt Name	*Properties*		
reset	*Type*	**Reset**	
	Condition	*Type*	Unselected
		Port Name	*Active Value*
		RESET	**1**
	Mask	*Maskable*	No
		Register Name	Unselected
		Position	
		Register Value	

NMI

Interrupt Name	*Properties*		
nmi	*Type*	**NMI**	
	Condition	*Type*	Unselected
		Port Name	*Active Value*
		NMI_IN	**1**
	Mask	*Maskable*	No

		Register Name	Unselected
		Position	
		Register Value	

External interrupt

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

Decode error interrupt（internal interrupt)

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**

Overflow interrupt of ALU (internal interrupt)

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**

Mask Condition Setting

Mask	*Maskable*	**Yes**

	Register Name	MASKREG
	Position	0
	Register Value	1

When the LSB of MASKREG is 1, interrupt generation can be masked.
In summary, as below.

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 Interrupt End Instruction Declaration

The instruction that gives the EOI_OUT(end of interrupt) signal when the interrupt processing routine ends is created beforehand. If this instruction is issued at interrupt handler's end, an interrupt end signal is transmitted to the external interrupt controller from the EOI_OUT port.

Table 20 shows an example of interrupt end instruction type “type0”. Table 21 shows an example of interrupt end instruction (eoi).

Table 20 Interrupt End Instruction Type Example

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

Table 21 Interrupt End Instruction Example

Name	*Type*	*Field*	*Format*
eoi	type0	011111 11111111111111111111111111	eoi

6.6.5 Interrupt Micro operation description

The Interrupt micro operation description is described in the “Exception” tab of the “Micro Op. Description” Window. This description should end within one cycle (pipeline and multi-cycle handling cannot be described). In general, the catch signal is asserted, interrupt mask is set, PC value is saved, and jump to to interrupt handler takes place.

BNF

< interrupt behavior statement> ::= < simple assignment statement> | < conditional assignment statement> | < saved PC assignment statement>
< saved PC assignment statement> ::= < left side> '=' 'saved_pc' ';'
(cf.) < statement > ::= < simple assignment statement> |
< conditional assignment statement> |

< conditional function execution statement> | <internal interrupt generation statement> | < conditional internal interrupt generation statement>

User Defined Port Handling

When reading or wirting to user defined ports, the read or write function of the corresponding wire resource is used.

6.7 Jump to Interrupt Handler and Assertion of Interrupt Acceptance Signal

6.7.1 When interrupt handler's address is fixed

This is a description of jumping to 0x10000000 when the interrupt occurs.

```
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 When the interrupt handler's address is taken (from an external input)

This is a description of jumping to the address read from the specific port **ADR_IN** when the interrupt enters.

```
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 When the interrupt handler's address is taken (from an interrupt vector)

When the interrupt handler's address is taken (from an interrupt vector) in the vectored interrupt, it might be good to describe it as follows, because there is especially no interrupt such as interrupt vector.

This is a description of jumping to the address specified with **intr_vector0-3** according to the interrupt vector read from the specific port **INTR_VECTOR_IN** when the interrupt enters.

```
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 Reset Interrupt

In reset interrupt, registers and instruction registers are reset. PC is reset, or set with the address in which the program read at the starting is stored.

Here, **GPR** and **IR** are reset, and PC is set with the address in which the program read at the starting is stored (0x30000).

```
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 Acquisition of PC value when Interrupt is Generated

In general, after the end of interupt handling, in order to return to the same location in program where it was interrupted the PC (Program counter) value when the interupt occured should be saved. In case of an internal interrupt, the available value of the PC is the instruction address when the internal interrupt occurs. On the other hand, in case of an external interrupt, the available value of the PC is the next instruction address of the last processing instruction. Therefore, when returning to the program before interrupt, the available PC value is the start address after interrupt in case of external interrupt. In reset interrupt and NMI the value of the PC when the interrupt occured cannot be saved.

In the next Micro Op. description, the value of PC is saved when an interrupt is generated. The value is saved in the register file (20th register).

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

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

The "**saved_pc**" is treated as a reserved word. At present, use of it is permitted only at the interrupt operation description. Moreover, when using **saved_pc** for the argument of the **FHM** resource function, it is necessary to intervene by another wire variable (variable “**spc**” in above-mentioned example) because the only form of using it at the right of the assignment statement.

6.8.1 Micro operation description of interrupt generation

For an internal interrupt of "**Instr_specific**" type, an interrupt is thrown within the micro operation description of a general instruction. (As for an internal interrupt of "**decode_error**" type, the detection logic is inserted automatically)

An example that "overflow" internal interrupt is generated if the overflow flag stands after executing addition with ALU is shown.

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

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

6.8.2 Micro operation description of interrupt end instruction

The instruction issued when the interrupt ends is described.

```
wire eoi_signal;
wire zero;

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

7. PAS – Meta-Assembler User Manual

This section describes how to use PAS meta-assembler. The following corresponds to PAS version 0.3.

7.1 PAS Outline

PAS Meta-assembler is a retargetable assembler, where the configuration can be easily changed. The configuration parameters consist of the register class, the instruction format, the number of bits of one byte, endianity, addressing unit (byte or word), etc. These parameters can be specified by using a file describing the instruction format, that is called an assemble rule description file.

For details of the assemble rule description file, please refer to [7.4 Assemble Rule Description File]. For the assembler control instructions, please refer to [7.7 Assembler Control Instructions]. For the command line options, please refer to [7.2 PAS Startup Method]

7.2 PAS Startup Method

In this section, the method of starting PAS is explained.
The form of starting PAS is as below.

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

The meaning of each option is as follows.

-des *descfile*
: The assemble rule description file is specified by the argument *descfile*. When **-help** or **-version** is not specified, this option is indispensable.

-src *srcfile*
: The assembly source file is specified by the argument *srcfile*. When **-help** or **-version** is not specified, this option is indispensable.

-a *listingfile*
: The output file of an assemble listing is specified by the argument *listingfile*. When this option is omitted, the output is sent to standard output.

-endian *endian*
: Endianity is specified by the argument *endian*. **Big** or **Little** can be specified.

-help
: A simple summary of the PAS usage is displayed in a standard output.

-version
: A version of PAS is displayed in a standard output.

-debug
: In addition to usual listing display, the dump of the result of analyzing the assemble rule file and the section contents are displayed.

7.3 PAS Execution Example

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

Here, **foo.des** is an assemble rule description file and **a.s** is an assembly source file. If **a.s** is as below:

```
        .data.24 -1
        .data.32 128
        .data.16 5
L4:
L5:
        .data.24 0123
        .data.24 0xabcd
```

then the following listing output is obtained.

```
****** 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 Assemble Rule Description File

The following three sections are described in the assemble rule description file.

- Register class definition

  Registers are divided into groups and their classes are defined. In each respective class, the correspondence between the name of the registers belonging to the class and the representation of registers in machine language code is described.

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

  This is an extract from the assemble rule description file for a processor. In this example, the register class named *gpr* is defined and it is shown that the registers named GPR0, GPR1, ... GPR31 belong to this class. For instance, the representation of the resister named GPR1 in the machine language code is 00001.

- Instruction format definition

  The Instruction format is defined. The instruction length, the number of operands and operands are defined. An example of the instruction format definition is shown as follows.

```
Instruction_type{
     type{
         "Itype"{
```

```
                width{"31","0"},
                otype{
                    "Itype0"{
                        "RDirect"{"Resource"{"rt","gpr"}},
                        "RDirect"{"Resource"{"rs","gpr"}},
                        "Immediate_data"{"Immediate"{"immediate",""}}
                    },
                    "Itype02"{
                        "RDirect"{"Resource"{"rt","gpr"}},
                        "RDirect"{"Resource"{"rs","gpr"}},
                        "Absolute_address"{"Symbol"{"immediate",""}}
                    },
                                ...
                }
            }
        }
    }
```

The instruction format named **Itype** is defined in this part. The instruction length of the instruction format named **Itype** is 32 bits of **width** {**"31","0"**}. A set of the operands that can be used by **Itype** form is defined in **otype**. In this case, the operand form named **Itype0** consists of three operands: general register, general register, and immediate data. In PAS, the addressing mode is defined beforehand. **Rdirec**t shows direct reference to register and **Immediate_data** shows immediate. **gpr** is a register class name defined in the register class definition.

- Machine Instruction Definition

The machine instruction for each individual instruction is defined. To which instruction format it belongs, field width, position, and usage (an operation code or an operand?) etc. are described.

The example definition of **ADDI** instruction is presented as follows.

```
Instruction{
        "ADDI"{
                type{"Itype"{otype{"Itype0"}}},
                "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"}}
        },
        ...
}
```

Refering to above example, it is understood that the instruction format of **ADDI** is **Itype** and the operand format is **Itype0**. **OP-code** and **Operand** describe each field of the instruction. The operation code is shown by 31 - 26 bit of the instruction, and its value in binary becomes 00100 for **ADDI**. There are three operands and the first is shown by 25 - 21 bit. The name of **rs** appears in the instruction format definition, it has direct

reference to register, and you will find that the register to be used belongs to the **gpr** class. The second operand, shown by 20 - 16 bit, indicates also that the register belongs to the **gpr** class. The third operand, shown by 15 - 0 **bit**, indicates immediate data of 16 bits.

Note that the assemble rule description file does not show what operation each machine instruction concretely does. It is limited only to information necessary to execute assembling. Hereafter, details of each definition are presented.

7.4.1 Register Class Definition

The description format of the register class definition is shown below.

```
Resource_tbl {
        " register class name " {
                {" register name", "code"},
                {" register name", " code "},
                ...
                {" register name", " code "}
        },
        " register class name " {
                {" register name", " code "},
                {" register name", " code "},
                ...
                {" register name", " code "}
        },
        ...
}
```

The register class definition starts by the key word named **Resource_tbl**. After that, the definition of each register class is shown between braces delimited by comma. Then, Register names and their code (binary number) are arranged within a register class. Note that, it is necessary to enclose the register class name, the register name, and the code with "" (double quart).

7.4.2 Instruction Format Definition

The description format of the instruction format definition is shown below.

```
Instruction_type {
  type {
    " instruction format name" {
      width{" most significant bit ", " least significant bit"},
      operand specification
    },
    " instruction format name " {
      width{" most significant bit ", " least significant bit"},
      operand specification
    },
    ...
  }
}
```

The instruction format definition starts by the key word named **Instruction_type**. After that, descriptions of each instruction format are arranged. Each instruction format consists of the combination of the instruction format name, the instruction length specification, and the operand specification. The **width** specifies the instruction length of each instruction format, for example in the case of 32 bit length, **width** is specified by **width{"31","0"}**. The operand specification specifies the number and the format of the operand. The description format of the operand specification is shown as follows.

```
otype{
  ″ operand format name″ {
     ″addressing mode name″ {
        ″ component name″, {″ field name″, ″ register class name″}
     },
     ″addressing mode name″ {
        ″ component name″, {″ field name″, ″ register class name″}
     }
     ...
  },
  ″ operand format name″ {
     ″addressing mode name″ {
        ″ component name″, {″ field name″, ″ register class name″}
     },
     ″addressing mode name″ {
        ″ component name″, {″ field name″, ″ register class name″}
     }
     ...
  },
  ...
}
```

The operand specification starts by the key word named **otype**. After that, the specifications of the operand that can be used in the belonging instruction format are arranged. The operand format name is a name of the operand format. The addressing mode name shows the addressing mode of the operand and is defined in PAS beforehand. The component name shows the meaning of this operand. The field name appearing in the field definition – to be explained in the next section, is used to specify which field of the machine language code the operand appears. When the operand is to use a register, the register class name is specified. When the register is not used, empty ("") is specified.

The name that can be used as an addressing mode name is as follows.

RDirect
Indirect
RIndirect
RIwPreDec
RIwPreInc
RIwPostDec
RIwPostInc
RIwDisp
RIwDisp_Up

RIwIndex
RIwIndex_Up
RIwIndex_Up
RIwScaledIndex
RIwDispScaledIndex
PCrelative_address
Absolute_address
Immediate_data

About details of the addressing mode, see section [7.6 Addressing Modes]. The component name can be one of the following.

Resource
: This shows that it is a register.

Displacement
: This shows that it is displacement of the addressing mode such as displacement modification register indirect etc.

Immediate
: This shows that it is immediate.

Scale
: This shows that it is a scale of the addressing mode such as scaling index modification register indirect etc.

Symbol
: This shows that it is a symbol.

Instruction Definition

```
Instruction {
   " Instruction name" {
      type{" Instruction format name"{otype{" operand type name"}}},
      " field type name"{
         " field attribute"{" field value"}, width{"MSB", "LSB"}
      },
      " field type name"{
         " field attribute"{" field value"}, width{"MSB", "LSB"}
      },
      ...
   },
   " Instruction name" {
      type{" Instruction format name"{otype{" operand type name"}}},
      " field type name"{
         " field attribute"{" field value"}, width{"MSB", "LSB"}
      },
      " field type name"{
         " field attribute"{" field value"}, width{"MSB", "LSB"}
      },
      ...
```

```
    },
    ...
}
```

The Instruction definition starts by the key word named **Instruction**. Each instruction consists of the instruction name, the instruction format specification and the description of each field.
The instruction format specification is specified by using the instruction format name and the operand type name defined in the instruction format definition.
The specification of the field consists of the field type name, the field attribute, the field value, and the positional specification. The field type name shows the distinction among the operation code and the operand etc. The field type name that can be specified is as follows.

OP-code
: This shows that the field is an operation code.

Operand
: This shows that the field is an operand.

Reserved
: This shows that the field is reserved and not being used now.

The field attribute is binary or name. In the case of binary, the value that should go into the field is specified by using a binary number for the following field value. In the case of name, the field name defined in the instruction format definition is written in the field value and the field is combined with the description of the operand as a result.
The positional specification is done by specifying most significant bit position (MSB) of the field and least significant bit position (LSB) in braces after the width key word.

7.5 Assembly Language Grammar

How to write an assembly source code is shown below.

7.5.1 Statement

A source program is composed of sentences. One sentence is described in one line. The composition of the sentence is as follows.

[label] <operation [operand]> [comment]

` **label** '
: This is a name labeling the sentence.
The label begins from the head of line and : (colon) is applied just behind the label name. For characters that can be used for the label, first character should be one of English capital letter, English small letter, underscore (_), or period (.). Number and dollar mark ($) are added to these characters as subsequent characters to the first character.

` **operation** '
: The mnemonic of an execution instruction and an assembler control instruction is described. The execution instruction is CPU instruction, and the one described in the assemble rule description file. The assembler control instruction is an instruction that controls the assembler.
The operation is written from column 2 when there is no label. When there is a label, it is written after one or more blanks or tabs after the label. The label might not be described according to the operation. About operations for which the label cannot be described by the assembler control instruction, this effect is written clearly in the explanation of each assembler control instruction.

` **operand** '
: The execution object of the operation and the like are described. The number and the kind of the operand are different according to the operation.
Start to write beyond one or more blanks or tabs after the operation.

` **comment** '
The annotation is described. It has no influence on the execution of a program.
It is a comment from semicolon (;) to the end of line.

7.5.2 Reserved Words

A Reserved word is a name that the assembler uses with a special meaning.
A Reserved word can be an assembler control instruction, an operator, or a location counter.

7.5.3 Symbols

The symbol is a name that the programmer defines, and it plays the following role.
`address symbol'
This shows an address of a stored position and a branch target.
`constant symbol'
This shows a constant.
For characters that can be used for the symbol name, first character should be one of English capital letter, English small letter, underscore (_), or period (.). Number and dollar mark ($) are added to these characters as subsequent characters to the first character.
Reserved word can not be used as a symbol.

7.5.4 Constants

There are an integer constant and a character-string constant in the constant.
There are the following kinds of integer constants.
`binary integer'
It starts by "0b" or "0B", and "0" or "1" continues.
`octal integer'
It starts by "0", and "0-7" continues.
`decimal integer'
It starts by the integer that is not "0", and "0-9" continues.
`hexadecimal integer'
It starts by "0x" or "0X", and "0-9,a-f,A-F" continues.
The character-string constant is an arrangement of characters treated as a string of data of the corresponding ASCII code value. It is shown enclosed with double quotes. The following can be used as a special notation besides usual ASCII printable characters.
¥b
This shows back space BS(0x08).
¥f
This shows form feed FF(0x0C).
¥n
This shows line feed LF(0x0A).
¥r
This shows carriage return CR(0x0D).
¥t
This shows horizontal tab HT(0x09).
¥v
This shows vertical tab VT(0x0B).
¥¥
This shows back slash.
¥″
This shows double quote.
Example:

"hello¥t¥"world¥"¥n"

Moreover, ASCII code value can be expressed by the octal notation and the hexadecimal notation.

` octal notation '

The figure of 0 - 7 is arranged after ¥0.

Example:

"¥012¥033"

` hexadecimal notation '

The figure of 0 - 9,a - f and A - F is arranged after ¥0x or ¥0X.

"¥0xa¥0xb"

7.5.5 Location Counter

The location counter indicates the address where the object code is located. The value of the location counter changes automatically according to the output of the object code. Moreover, it is also possible to change it to a specified value by the assembler control instruction.
The value of a present location counter can be referred to by dot (.). Dot (.) is a reserved word.

7.5.6 Expression

The expression is obtained by combining constants, symbols, and operators. It is used as an operand of an instruction or assembler control instruction.
The elements of an expression is shown below.

7.5.6.1 Term

The term consists of the following.

- Constant
- Location counter (.)
- Symbol
- Operation result by the above-mentioned term and operator

A single term is a kind of expression.

7.5.6.2 Operator

The kind and the priority of the operator are as follows. The smaller the priority number, the higher the priority becomes.

Priority 0

-

A single – executes reversing sign or getting a 2's complement number.

~

NOT of each bit, i.e. getting a 1's complement number.

Priority 1

*

multiplication

/

division

%

remainder

Priority 2

|
OR of each bit
&
AND of each bit
^
XOR of each bit
Priority 3
+
addition
–
subtraction

7.5.6.3 Brackets

The priority of operation can be changed by round brackets ().

When two or more operations are included in one expression, the order by which the operation is processed is determined by the priority of operator and the brackets specification.
The operation is processed in PAS according to the following rules.

`Rule 1'
Processing starts from the operation surrounded by brackets. When brackets are multiple, priority is given to the operation surrounded by the inside brackets.

`Rule 2'
Processing is done from the operation with higher priority of operation.

7.6 Addressing Modes

In PAS, addressing modes generally used are prepared beforehand and can be used in standard without depending on architecture.
Addressing modes defined in PAS are shown as follows.

`%Rn ：Direct reference to register'
Register is referred.

`(expression) ：Indirect addressing'
The value of the expression shows the memory address.

`(%Rn) ： Indirect reference to register'
The value of the register Rn shows the memory address.

`–(%Rn) ： Pre-decrement indirect reference to register'
The value of register Rn after decrement shows the memory address.

`+(%Rn) ： Pre-increment indirect reference to register'
The value of register Rn after increment shows the memory address.

`(%Rn)– ： Post-decrement indirect reference to register'
The value of register Rn before decrement shows the memory address.

`(%Rn)+ ： Post-increment indirect reference to register'
The value of register Rn before increment shows the memory address.

`disp(%Rn) ： Displacement modification indirect reference to register '
The value of "Value of Rn + disp" shows the memory address. The value of register Rn is not updated.

`(%Rbase + disp) ： Displacement modification indirect reference to register with updating addressing'
The value of "Value of Rn + disp" shows the memory address. The value of register Rn is updated to the value of " Value of Rn + disp "".

`(%Rbase, %Rindex) : Index modification indirect reference to register '

The value of "Value of Rbase + value of Rindex " shows the memory address. The value of register Rbase and the value of Rindex are not updated.

`(%Rbase + %Rindex) : Index modification indirect reference to register with updating addressing'

The value of "Value of Rbase + value of Rindex " shows the memory address. The value of register Rbase is updated to the value of "Value of Rbase + value of Rindex ".

`(%Rbase, %Rindex, scale) : Scaling index modification indirect reference to register '

The value of "Value of Rbase + value of Rindex × scale " shows the memory address. The value of register Rbase and the value of Rindex are not updated.

`disp(%Rbase, %Rindex, scale) Displacement, scaling index modification indirect reference to register'

The value of "Value of Rbase + value of Rindex × scale +disp" shows the memory address. The value of register Rbase and the value of Rindex are not updated.

`symbol : PC relative address by symbol specification'

Symbol shows the memory address of a branch target. The value of "Value of Symbol- PC" is set as displacement relative to PC.

`*symbol : Absolute address by symbol specification'

Symbol shows the absolute memory address of a branch target.

`$imm: Immediate'

The value of imm is used as it is.

7.7 Assembler Control Instructions

7.7.1 .accum *"accum_num"*

The fixed point data with an integer part is reserved. The fixed point number of the decimal point notation specified by *accum_num* is converted according to the form specified by the **accum_spec** control instruction and arranged in the memory.

The value of the data that can be specified is different depending on the number of binary digits of integer part specified by **.accum_spec**. If the number of binary digits of integer part is four, it reaches the value within the range of -16<= *accum_num*<16.

Example:

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

7.7.2 .accum.s *"accum_num"*

The fixed point data with a short-precision integer part reserved. The fixed point number of the decimal point notation specified by *accum_num* is converted according to the form specified by the **.short_accum_spec** control instruction and arranged in the memory.

The value of the data that can be specified is different depending on the number of binary digits of integer part specified by **.short_accum_spec**. If the number of binary digits of integer part is four, it reaches the value within the range of -16<= *accum_num*<16.

Example:

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

7.7.3 .accum.l *"accum_num"*

The fixed point data with a long-precision integer part is reserved. The fixed point number of the decimal point notation specified by *accum_num* is converted according to the form specified by the **.long_accum_spec** control instruction and arranged in the memory.

The value of the data that can be specified is different depending on the number of binary digits of integer part specified by **.long_accum_spec**. If the number of binary digits of integer part is four, it reaches the value within the range of -16<= *accum_num*<16.
Example:

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

7.7.4 .accum_spec *"spec"*

The fixed point data with an integer part is defined. Though there are a signed form and an unsigned form, both are specified. The size of data, the number of integer part binary digits and the scale (the number of fraction part binary digits) are specified by *spec*. The unsigned form is specified in the former half part of *spec* and the signed form is specified in the latter half part.
The specification method of *spec* is as follows.

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

`*size1*
Size of data for an unsigned type
`*integral1*
Number of digits in binary number of integer part for an unsigned type
`*scale1*
Number of digits in binary number of fraction part for an unsigned type
`*size2*
Size of data for a signed type
`*integral2*
Number of digits in binary number of integer part for a signed type
`*scale2*
Number of digits in binary number of fraction part for a signed type

The default is set as follows.

20:4:15,20:4:15

Label cannot be described.

7.7.5 .addr_space *"num"*

The size of an address space is specified. The argument *num* specifies an integer value. The size of the address space becomes 2^*num*. The default value is 16. Label cannot be described.

7.7.6 .addressing *"addrunit"*

The unit of addressing of a memory is set. **Byte** or **Word** can be specified for *addrunit*. If **Byte** is specified, the addressing is done in a byte unit. If Word is specified, the addressing is done in a word unit. The default is **Byte**. Label cannot be described.
Example:

```
.addressing Byte
.addressing Word
```

7.7.7 .align *"num"*

The argument *num* is an integer value and specifies the alignment number of the location counter value. The value of the location counter is corrected to multiples of the alignment number. Example:

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

In this example, the constant secured by the **.data.32** control instruction is aligned to four byte boundary by **.align 4**.

7.7.8 .ascii *"string"*

The character-string constant data specified by argument *string* is reserved in memory. Zero terminal character string is not added. Label can be described. Example:

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

7.7.9 .asciz *"string"*

The character-string constant data specified by argument *string* is reserved in memory. Zero terminal character string is added to the end of the character-string data. Label can be described. Example:

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

7.7.10 .bits_per_byte *"num"*

The number of bits of one byte is specified. The argument *num* specifies an integer value. Default is eight. Label cannot be described. Example:

```
        .bits_per_byte 12       ; 12 bits is specified per one byte.
```

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

The integer data is reserved. The *width* specifies the size of data by the number of bits. The *num* is an integer value and specifies the value of the integer data. Example:

```
        .data.24 10     ; The data of 24 bit length is secured.
        .data.16 0x1234 ; The data of 16 bit length is secured.
```

7.7.12 .double *"hex_num"*

The double precision floating point data type is reserved. The data type is prescribed by the **.double_spec** control instruction. The encoded data is specified for argument *hex_num* with the hexadecimal number. Example:

```
.double 0x3ff0000000000000     ; 1.0 (for the default format setting)
```

7.7.13 .double_spec *"spec"*

The double precision floating point data type is defined. The specification of the argument *spec* is as follows.

size:*sign*:*exponet*:*mantissa*

Each meaning is as below. All the sizes are specified with the number of bits.

`size'

This is an entire size of the data type.

`sign'
This specified the number of bits in which the sign is shown.
`exponent'
This is a size of the exponent part.
`mantissa'
This is a size of the mantissa.
The default is set as follows.

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

Label cannot be described.

7.7.14 .end

The end of the source program is declared.
Label cannot be described.

7.7.15 .endian *"endian"*

Endianity is specified. For the argument *endian,* **Big** is specified in the case of the Big endian and **Little** is specified in the case of the Little endian. The default is **Big**.
Label cannot be described.

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

The value specified by the argument *value* is set to the symbol specified by the argument *symbol*. The value of the symbol set with **.equ** cannot be set again. Further settings are discarded.
Label cannot be described.
Example:

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

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

The value specified by the argument *value* is set to the symbol specified by the argument *symbol*. The value of the symbol set with **.equiv** can be set again.
Label cannot be described.
Example:

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

7.7.18 .fixed *"fixed_num"*

The fixed point data is reserved. The fixed point number of the decimal point notation specified by *fixed_num* is converted according to the form specified by the **.fixed_spec** control instruction and arranged on a memory.
The value of the data that can be specified reaches the value within the range of -1 <= *fixed_num* < 1.
Example:

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

7.7.19 .fixed.s *"fixed_num"*

The short-precision fixed point data is reserved. The fixed point number of the decimal point notation specified by *fixed_num* is converted according to the form specified by

the **.short_fixed_spec** control instruction and arranged on a memory.

The value of the data that can be specified reaches the value within the range of -1 <= *fixed_num* < 1.

Example:

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

7.7.20 .fixed.l *"fixed_num"*

The long-precision fixed point data is reserved. The fixed point number of the decimal point notation specified by *fixed_num* is converted according to the form specified by the **.long_fixed_spec** control instruction and arranged in the memory.

The value of the data that can be specified reaches the value within the range of -1 <= *fixed_num* < 1.

Example:

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

7.7.21 .fixed_spec *"spec"*

The fixed point data is defined. Though there are a signed form and an unsigned form, both are specified. The size of data and the scale (the number of fraction part binary digits) are specified by *spec*. The unsigned form is specified in the former half part of *spec* and the signed form is specified in the latter half part.

The specification method of *spec* is as follows.

size1:*scale1*,*size2*:*scale2*

`*size1*
: Size of data for an unsigned type

`*scale1*
: Number of digits in binary number of fraction part for an unsigned type

`*size2*
: Size of data for a signed type

`*scale2*
: Number of digits in binary number of fraction part for a signed type

The default is set as follows.

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

Label cannot be described.

7.7.22 .float *"hex_num"*

The single precision floating point data type is reserved. The data type is prescribed by the **.float_spec** control instruction. The encoded data is specified for argument *hex_num* with the hexadecimal number.

Example:

```
.float 0x3f800000       ; 1.0 (for the default format setting)
```

7.7.23 .float_spec *"spec"*

The single precision floating point data type is defined. The specification of the argument *spec*

is as follows.

size:*sign*:*exponet*:*mantissa*

Each meaning is as below. All the sizes are specified with the number of bits.

`size'
This is an entire size of the data type.

`sign'
This specified the number of bits in which the sign is shown.

`exponent'
This is a size of the exponent part.

`mantissa'
This is a size of the mantissa.

The default is set as follows.

```
.float_spec 32:1:8:23
```

Label cannot be described.

7.7.24 .long_accum_spec *"spec"*

The fixed point data with a long precision integer part is defined. Though there are a signed form and an unsigned form, both are specified. The size of data, the number of integer part binary digits and the scale (the number of fraction part binary digits) are specified by *spec*. The unsigned form is specified in the former half part of *spec* and the signed form is specified in the latter half part.

The specification method of *spec* is as follows.

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

`*size1*'
Size of data for an unsigned type

`*integral1*'
Number of digits in binary number of integer part for an unsigned type

`*scale1*'
Number of digits in binary number of fraction part for an unsigned type

`*size2*'
Size of data for a signed type

`*integral2*'
Number of digits in binary number of integer part for a signed type

`*scale2*'
Number of digits in binary number of fraction part for a signed type

The default is set as follows.

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

Label cannot be described.

7.7.25 .long_fixed_spec

The long-precision fixed point data is defined. Though there are a signed form and an unsigned form, both are specified. The size of data and the scale (the number of fraction part binary digits) are specified by *spec*. The unsigned form is specified in the former half part of *spec* and the signed form is specified in the latter half part.

The specification method of *spec* is as follows.

size1:*scale1*,*size2*:*scale2*

`*size1*'

Size of data for an unsigned type

`*scale1*'

Number of digits in binary number of fraction part for an unsigned type

`*size2*'

Size of data for a signed type

`*scale2*'

Number of digits in binary number of fraction part for a signed type

The default is set as follows.

```
.long_fixed_spec 32:31,32:31
```

Label cannot be described.

7.7.26 .org "*lc*"

The value of a location counter is set. The *lc* is an integer with which the value of the location counter is specified. When the value of the location counter increases by *.org*, zeros are buried between the former value of the location counter and the new value of the location counter.
Label cannot be described.
Example:

```
.org 0x100
```

7.7.27 .section "*section*"

The section specified by the argument *section* is declared. The section that can be specified is as follows.

.text

Text section

.data

Data section

It is possible to declare again the declared section and restart.
The default section is prepared for any of the following.

- An execution instruction is described before the section is declared.
- An assembler control instruction that reserves data is described before the section is declared.
- The **.align** or the **.org** or the **.space** assembler control instruction is described before the section is declared.
- A location counter is referred to before declaring the section.
- Only the line of the label is described before the section is declared.

The default section is the text section of first address **0** and alignment number **2.** .
Label cannot be described.

7.7.28 .short_accum_spec "*spec*"

The fixed point data with a short precision integer part is defined. Though there are a signed form and an unsigned form, both are specified. The size of data, the number of integer part binary digits and the scale (the number of fraction part binary digits) are specified by *spec*. The

unsigned form is specified in the former half part of *spec* and the signed form is specified in the latter half part.
The specification method of *spec* is as follows.

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

`*size1*
Size of data for an unsigned type
`*integral1*
Number of digits in binary number of integer part for an unsigned type
`*scale1*
Number of digits in binary number of fraction part for an unsigned type
`*size2*
Size of data for a signed type
`*integral2*
Number of digits in binary number of integer part for a signed type
`*scale2*
Number of digits in binary number of fraction part for a signed type

The default is set as follows.

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

Label cannot be described.

7.7.29 .short_fixed_spec "*spec*"

The short-precision fixed point data is defined. Though there are a signed form and an unsigned form, both are specified. The size of data and the scale (the number of fraction part binary digits) are specified by *spec*. The unsigned form is specified in the former half part of *spec* and the signed form is specified in the latter half part.
The specification method of *spec* is as follows.

size1:*scale1*,*size2*:*scale2*

`*size1*
Size of data for an unsigned type
`*scale1*
Number of digits in binary number of fraction part for an unsigned type
`*size2*
Size of data for a signed type
`*scale2*
Number of digits in binary number of fraction part for a signed type

The default is set as follows.

```
.short_fixed_spec 8:7,8:7
```

Label cannot be described.

7.7.30 .space "*num*"

The value specified by the argument *num* is added to the current value of location counter. An integer value is specified for the *num*. When the value of the location counter increases by **.space**, zeros are buried between the former value of the location counter and the new value of the location counter.

```
.space 16          ; The location counter is advanced by 16.
```

7.7.31 .uaccum *"accum_num"*

The unsigned fixed point data with an integer part is reserved. The fixed point number of the decimal point notation specified by *accum_num* is converted according to the form specified by the **accum_spec** control instruction and arranged in memory.
The value of the data that can be specified is different depending on the number of binary digits of integer part specified by **.accum_spec**. If the number of binary digits of integer part is four, it reaches the value within the range of 0<= *accum_num*<16.
Example:

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

7.7.32 .uaccum.s *"accum_num"*

The unsigned fixed point data with a short-precision integer part is reserved. The fixed point number of the decimal point notation specified by *accum_num* is converted according to the form specified by the **.short_accum_spec** control instruction and arranged in memory.
The value of the data that can be specified is different depending on the number of binary digits of integer part specified by **.short_accum_spec**. If the number of binary digits of integer part is four, it reaches the value within the range of 0<= *accum_num*<16.
Example:

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

7.7.33 .uaccum.l *"accum_num"*

The unsigned fixed point data with a long-precision integer part is reserved. The fixed point number of the decimal point notation specified by *accum_num* is converted according to the form specified by the **.long_accum_spec** control instruction and arranged in memory.
The value of the data that can be specified is different depending on the number of binary digits of integer part specified by **.long_accum_spec**. If the number of binary digits of integer part is four, it reaches the value within the range of 0<= *accum_num*<16.
Example:

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

7.7.34 .ufixed *"fixed_num"*

The unsigned fixed point data is reserved. The fixed point number of the decimal point notation specified by *fixed_num* is converted according to the form specified by the **.fixed_spec** control instruction and arranged in memory.
The value of the data that can be specified reaches the value within the range of 0<= *fixed_num* < 1.
Example:

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

7.7.35 .ufixed.s *"fixed_num"*

The unsigned short-precision fixed point data is reserved. The fixed point number of the decimal point notation specified by *fixed_num* is converted according to the form specified by

the **.short_fixed_spec** control instruction and arranged in memory.
The value of the data that can be specified reaches the value within the range of 0<= *fixed_num* < 1.
Example:

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

7.7.36 .ufixed.l *"fixed_num"*

The unsigned long-precision fixed point data is secured. The fixed point number of the decimal point notation specified by *fixed_num* is converted according to the form specified by the **.long_fixed_spec** control instruction and arranged on a memory.
The value of the data that can be specified reaches the value within the range of 0<= *fixed_num* < 1.
Example:

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

7.7.37 .word_alignment *"num"*

The word boundary is aligned to the value specified by the argument *num*.
Label cannot be described.
Example:

```
.word_alignment 8        ; aligned to 8 byte boundary
```

8. Registered Resource Models and Parameters

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

<result, flag> = 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

<result, flag> = 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

<result, flag> = 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
<result, flag> = 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
<result, flag> = 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
<result, flag> = 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
<result, flag> = 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
<result, flag> = 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
<result, flag> = 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
<result, flag> = 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
<result, flag> = 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

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

8.2.13 Function : csub

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

<result, flag> = 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

<result, flag> = 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

<result, flag> = 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

<result, flag> = 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

<result, flag> = 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

<result, flag> = 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

<result, flag> = 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

<result, flag> = 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

<result, flag> = 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

<result, flag> = 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

<result, flag> = 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
<result, flag> = 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
<result, flag> = 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
<result, flag> = 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
<result, flag> = 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
<result, flag> = 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 = 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* $\geqq$ *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* ≧ *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* ≦ *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> = 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* ≦ *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> = 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 << 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 >> 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>* = div*(a, b)*}
- divu : Unsigned division {*<q, r, flag>* = divu*(a, b)*}
- diva : Absolute division {*<q, r, flag>* = 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> = 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> = 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
<q, r, flag> = 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* = zero*(data_in)*}
- sign : Sign extension {*data_out* = 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 = 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 = 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*])

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*i*(*data_in_i, w_sel_i*)}
- read*i* : Data read {*data_out_i* = read(*r_sel_i*)}

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*i*(*data_in_i, w_sel_i*)
Attention:
For all *0* ≦ *i* ≦ *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_out_i = read*i*(*r_sel_i*)
Attention:
For all 0 ≦ *i* ≦ *num_read_port*, read*i* 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) }
- forward*n*: Set forwarding data { null = forward*n*(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 forward*n* function, otherwise data from register file.
Format
data = forward(register_number, data_from_register)

8.11.2 Function : Forward*n*

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 = forward*n*(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* : Minimul 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
<*data_out, addr_err*>= ldu_*i*(*addr*)

Attention:
For all *i* ≠ *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()

索引

N

O

P

R

S

T

U

V

W