



ASIP Meister Tutorial

July 2008

Version 2.3

ASIP Solutions, Inc.



©2008, ASIP Solutions, Inc.

ALL RIGHTS RESERVED

Reproduction or distribution of this document is prohibited without prior permission.

In case of copy or duplication, please contact the following address:

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

1. Tutorial Conventions	1
2. Before Using ASIP Meister	1
2.1. Installation and Setup	1
2.2. Tutorial Design Sample	1
3. Common Operations Using ASIP Meister	1
3.1. Startup	2
3.1.1. Path Setting	2
3.1.2. ASIP Meister Startup	2
3.1.3. Starting with a New Design	2
3.1.4. Starting with an Existing Design	3
3.2. Working Directory “meister” and Design Data File “.pdb”	5
3.3. ASIP Meister Exit and Design Data Save	5
3.3.1. Saving New Design Data (or Saving Design Data under Different Name)	5
3.3.2. Updating Existing Design Data (Without Changing File Name)	7
3.4. Closing Sub-Windows and Return to Main Window	7
4. Processor Design Flow Using ASIP Meister	7
4.1. Reading Memory Model Definition File	9
4.2. Setting Design Goal & Architecture Parameters	9
4.2.1. Design Data Management Information	10
4.2.2. Design Goal Values	11
4.2.3. Architecture Parameters	11
4.3. Resource Declaration	15
4.4. Storage Specification Definition	28
4.4.1. Register File Specification	28
4.4.2. General Register Specification	31
4.4.3. Memory Specification	32

4.5. Interface Definition	33
4.5.1. Entity Name Setting	34
4.5.2. I/O Ports Settings	35
4.6. Instruction Set Definition	37
4.6.1. Instruction Set Definition in ASIP Meister	37
4.6.2. Instruction type definition	38
4.6.3. Instruction Definition	41
4.6.4. Interrupt/Exception Setting	43
4.7. Arch. Level Estimation	46
4.8. Assembler Generation	48
4.8.1. Assembler File Generation	50
4.9. Micro Operation Description	50
4.9.1. Macro definition: [Macro]	51
4.9.2. Instruction Operation Description: [Instruction]	53
4.9.3. Interrupt/Exception Description: [Exception]	60
4.10. HDL Generation	61
4.10.1. Confirming Generated HDL	63
4.11. C Definition (ASIP Meister Standard Environment Required)	65
4.12. Compiler Generation, Binutils Generation: (ASIP Meister Standard environment is required)	71
<i>A. Design Goals</i>	<i>1</i>
<i>B. Memory Access Method</i>	<i>1</i>
<i>C. Pipeline Structure</i>	<i>1</i>
<i>D. Resource Usage</i>	<i>2</i>
<i>E. Storage Definition</i>	<i>3</i>
E.1. Register File/Storage Definition	3
E.1.1. Register File Setting	3
E.1.2. Register File (Register Usage)	3
E.2. Register/Storage Definition	4
E.2.1. Instruction Register Setting	4
E.2.2. Program Setting	4

E.3. Memory/Storage Definition	4
E.3.1. Instruction Memory Interface Unit	4
E.3.2. Data Memory Interface Unit	5
F. Input/Output Ports	5
G. Instruction Set List	5
G.1. Instruction Type	5
G.1.1. Register-Register Type (R_R Type)	6
G.1.2. Register-Immediate Type (R_I Type)	6
G.1.3. Conditional Branch Type (B Type)	6
G.1.4. Un-Conditional Branch Type (JP_T Type)	7
G.2. Instruction List	7
G.2.1. ADD Instruction	7
G.2.2. SUB Instruction	8
G.2.3. LOAD Instruction	8
G.2.4. STORE Instruction	9
G.2.5. BEZ Instruction	10
G.2.6. Instruction	10
H. Interrupt/Exception Handling	11

List of Figures

FIGURE 1 ASIP MEISTER MAIN WINDOW (STARTING A NEW DESIGN)	3
FIGURE 2 [FILE] MENU	4
FIGURE 3 ASIP MEISTER MAIN WINDOW (DESIGN IN PROGRESS)	5
FIGURE 4 SAVE DESIGN	6
FIGURE 5 DATA SAVE DESIGN BOX	6
FIGURE 6 COMPLETE CHECK BOX	7
FIGURE 7 DESIGN FLOW USING ASIP MEISTER	8
FIGURE 8 DESIGN GOAL & ARCH. DESIGN BUTTON	9
FIGURE 9 [DESIGN GOAL & ARCH. DESIGN] SUB-WINDOW	10
FIGURE 10 DESIGN DATA MANAGEMENT INFO	10

FIGURE 11 DESIGN GOAL INPUT	11
FIGURE 12 ARCHITECTURE PARAMETER INPUT.....	12
FIGURE 13 PIPELINE STAGES SETTING	13
FIGURE 14 DELAYED BRANCH AND BIT-WIDTH SETTING	14
FIGURE 15 RESOURCE DECLARATION BUTTON.....	15
FIGURE 16 RESOURCE DECLARATION WINDOW.....	16
FIGURE 17 FHM BROWSER	17
FIGURE 18 [FHM BROWSER] WITH [PCU] MODEL INFO	18
FIGURE 19 [PCU] PARAMETER SETTING	19
FIGURE 20 [PCU] RESOURCE ESTIMATE BUTTON.....	19
FIGURE 21 INSTANCE NAME INPUT	19
FIGURE 22 PC RESOURCE DECLARATION COMPLETE STATE.....	20
FIGURE 23 PC RESOURCE PORTS INFO.....	21
FIGURE 24 DECLARATION OF INSTRUCTION REGISTER.....	21
FIGURE 25 DECLARATION OF REGISTER FILE	22
FIGURE 26 DECLARATION OF ALU	23
FIGURE 27 DECLARATION OF EXTENDER	24
FIGURE 28 DECLARATION OF INSTRUCTION MEMORY INTERFACE UNIT.....	25
FIGURE 29 DECLARATION OF DATA MEMORY INTERFACE UNIT.....	26
FIGURE 30 RESOURCE DECLARATION COMPLETED	27
FIGURE 31 STORAGE SPEC. BUTTON.....	28
FIGURE 32 STORAGE SPEC. WINDOW.....	28
FIGURE 33 REGISTER FILE SPEC. SETTING (BEFORE).....	29
FIGURE 34 REGISTER FILE SPEC. SETTING (AFTER).....	29
FIGURE 35 REGISTER FILE EXPANSION DATA CONFIRM MESSAGE.....	30
FIGURE 36 DETAILED INFO. OF REGISTER FILE SPEC.	31
FIGURE 37 REGISTER SPEC. SETTING.....	32
FIGURE 38 MEMORY SPEC. SETTING.....	32
FIGURE 39 INTERFACE DEFINITION BUTTON.....	33
FIGURE 40 INTERFACE DEFINITION WINDOW	34
FIGURE 41 ENTITY NAME SETTING.....	35
FIGURE 42 I/O PORTS SETTINGS	36
FIGURE 43 I/O PORT VALIDITY.....	36
FIGURE 44 INSTRUCTION DEFINITION BUTTON	37
FIGURE 45 INSTRUCTION TYPE AND INSTRUCTION DEFINITION.....	37
FIGURE 46 NEW TYPE BUTTON	38

FIGURE 47 [NEW INSTRUCTION TYPE CONFIRM] WINDOW.....	38
FIGURE 48 NEW INSTRUCTION TYPE SETTING.....	39
FIGURE 49 R_R TYPE SETTING	40
FIGURE 50 [INSTRUCTION DEFINITION] WINDOW (R_R TYPE)	40
FIGURE 51 [INSTRUCTION DEFINITION] WINDOW (ALL TYPES).....	41
FIGURE 52 NEW INSTRUCTION BUTTON	42
FIGURE 53 REGISTRATION OF INSTRUCTION NAME	42
FIGURE 54 NEW INSTRUCTION SETTING	42
FIGURE 55 SMALL_RISC INSTRUCTIONS	43
FIGURE 56 INTERRUPT/EXCEPTION DEFINITION PANEL	44
FIGURE 57 INTERRUPT/EXCEPTION TYPE SETTING	45
FIGURE 58 RESET SIGNAL SETTING.....	45
FIGURE 59 INTERRUPT/EXCEPTION VALIDITY.....	46
FIGURE 60 ARCH. LEVEL ESTIMATION BUTTON.....	47
FIGURE 61 ESTIMATION EXECUTION.....	47
FIGURE 62 ESTIMATION EXECUTION RESULTS	48
FIGURE 63 ASSEMBLER GENERATION BUTTON.....	49
FIGURE 64 ASSEMBLER DESCRIPTION GENERATION.....	49
FIGURE 65 ASSEMBLER DESCRIPTION GENERATION RESULTS	49
FIGURE 66 GENERATED ASSEMBLER DESCRIPTION FILE.....	50
FIGURE 67 MICRO OP. DESCRIPTION BUTTON	50
FIGURE 68 MICRO OP. DESCRIPTION WINDOW	51
FIGURE 69 MACRO WINDOW	52
FIGURE 70 NEW MACRO CONFIRM	53
FIGURE 71 FETCH MACRO INPUT	53
FIGURE 72 INSTRUCTION MICRO OPERATIONS DESCRIPTION WINDOW	54
FIGURE 73 ADD INSTRUCTION MICRO OP.....	55
FIGURE 74 SUB INSTRUCTION MICRO OP.....	56
FIGURE 75 LOAD INSTRUCTION MICRO OP.....	57
FIGURE 76 STORE INSTRUCTION MICRO OP.....	58
FIGURE 77 BEZ INSTRUCTION MICRO OP.....	59
FIGURE 78 J INSTRUCTION MICRO OP.....	60
FIGURE 79 RESET INTERRUPT DESCRIPTION	61
FIGURE 80 HDL GENERATION BUTTON	62
FIGURE 81 [GENERATION CONFIRM] WINDOW	62
FIGURE 82 HDL GENERATION EXECUTION RESULTS	63

FIGURE 83 HDL FILES USED FOR SIMULATION	64
FIGURE 84 HDL FILES USED FOR SYNTHESIS.....	64
FIGURE 85 C DEFINITION BUTTON.....	65
FIGURE 86 [C DEFINITION] WINDOW	66
FIGURE 87 [CKF PROTOTYPE] TAB.....	67
FIGURE 88 [NEW CKF] WINDOW	67
FIGURE 89 MICRO OP. DESCRIPTION OF NXOR INSTRUCTION	70
FIGURE 90 [CKF PROTOTYPE] WINDOW AFTER THE REGISTRATION OF NXOR INSTRUCTION.....	71
FIGURE 91 COMPILER GENERATION BUTTON	71
FIGURE 92 [GENERATION CONFIRM] WINDOW	72

1. Tutorial Conventions

This tutorial is written according to the following conventions.

1. User interfaces such as menus, buttons, text boxes, and check boxes are enclosed with brackets ([and]).
2. “\$” Symbol denotes command prompt.
3. The name of files and directories are enclosed within double quotation marks.

2. Before Using ASIP Meister

2.1. Installation and Setup

Prepare a file “ASIPmeister.setup” to set the right paths to the executable files of ASIP Meister and JAVA™ 2 and the ASIP Meister environment variables. If the installation is completed according to the Installation Guide, describe “ASIPmeister.setup” as follows

Setup File Example:

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

2.2. Tutorial Design Sample

In this tutorial, you are going to generate a small processor core (small_RISC) using ASIP Meister. While referring to the specification of the processor core in this tutorial appendix, please proceed with your tutorial and enjoy till the end!

3. Common Operations Using ASIP Meister

In this section, you can accustom yourself to different operations that are common to



all design projects enabled with ASIP Meister.

The design data used in this tutorial will be saved under a directory named “tutorial”. Create a directory with this name in advance.

3.1. Startup

3.1.1. Path Setting

Please open a terminal and set the path for JAVA™ 2 and ASIP Meister. This tutorial assumes that users use the sample ASIP Meister setup file. Execute the following command in the directory where “ASIPmeister.setup” file is located.

```
$ source ASIPmeister.setup
```

3.1.2. ASIP Meister Startup

When starting ASIP Meister, the user will need to enter a starting command in a terminal. The command differs according to whether the user intends to start a new design or open an existing design.

3.1.3. Starting with a New Design

When starting a new design, ASIP Meister should be started as below.

```
$ ASIPmeister &
```

In this case, the main window shown in Figure 1 will open.

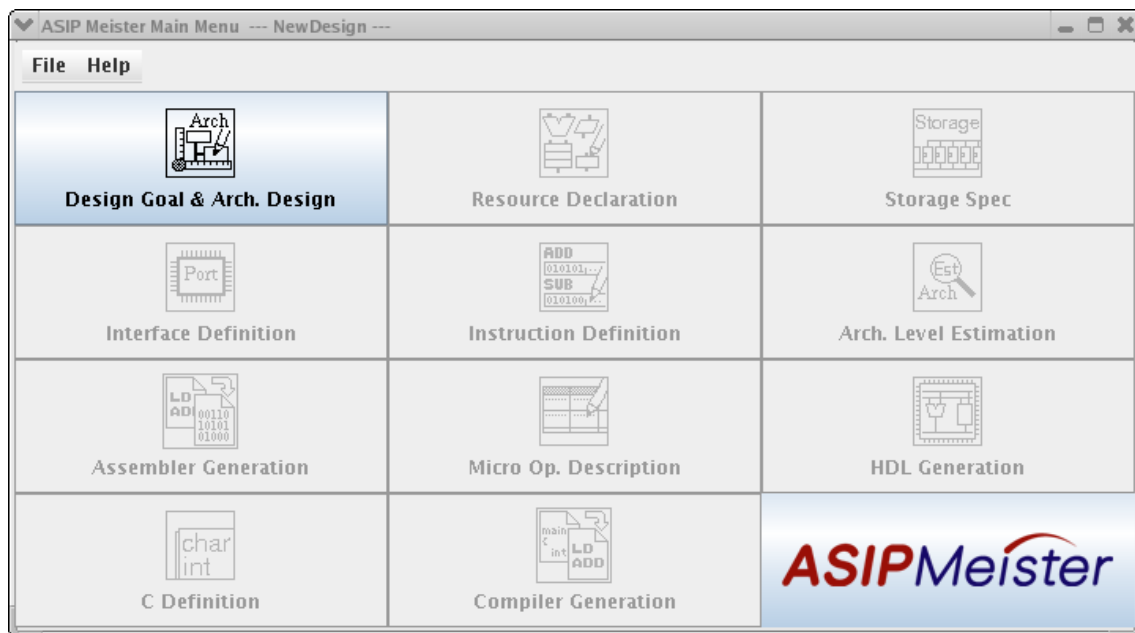


Figure 1 ASIP Meister Main Window (Starting a New Design)

3.1.4. Starting with an Existing Design

When updating an existing design data, ASIP Meister should be started as already explained in previous section. Then, from [File] Menu, click [Open Design] to select the design data file to open.

\$ ASIPmeister &

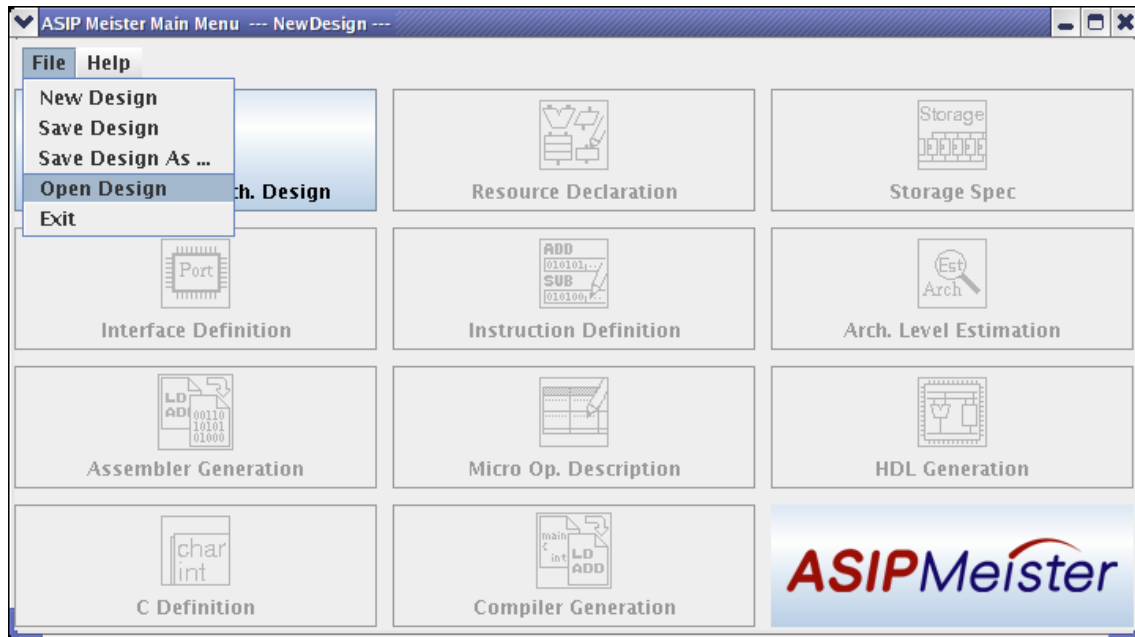


Figure 2 [File] Menu

Otherwise, the name of a design data file can be specified as an argument when starting ASIP Meister. In such a case, the main window will open to allow the user to update the existing design or complete remaining design steps. Figure 3 shows ASIP Meister main window with design data updated till Storage Spec. step when starting ASIP Meister using the following command,

```
$ ASIPmeister small_RISC.pdb &
```

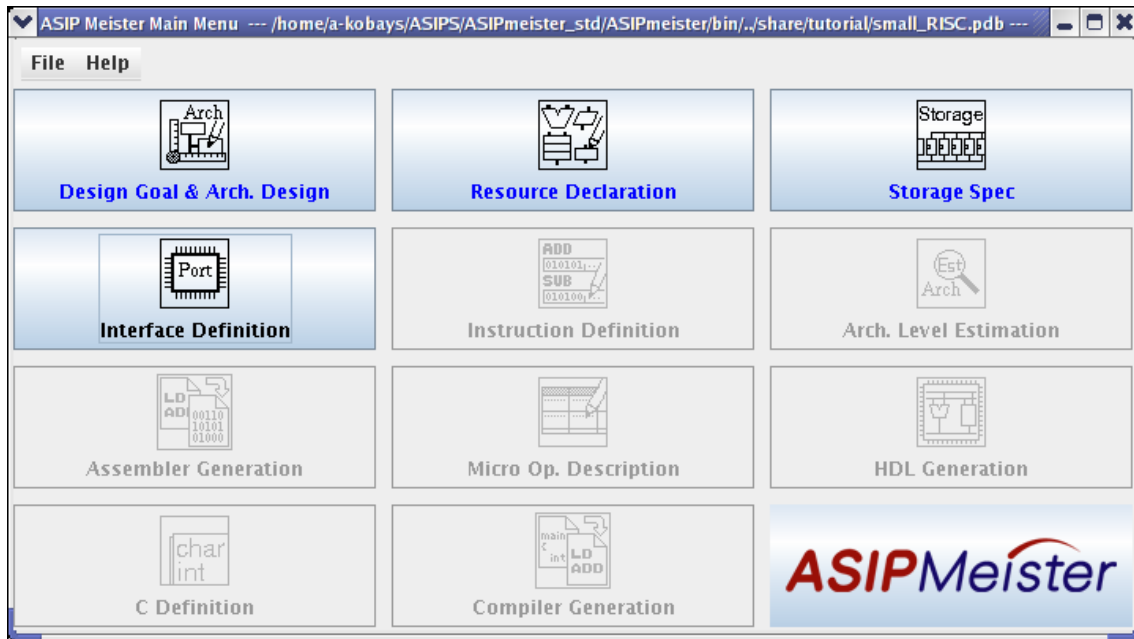


Figure 3 ASIP Meister Main Window (Design in Progress)

3.2. Working Directory “meister” and Design Data File “.pdb”

3.2.1. Working Directory “meister”

ASIP Meister creates a working directory named “meister” in the current directory. Working files of ASIP Meister and generated HDL description files are stored in “meister” directory.

3.2.2. Design Data File

The extension of the design data file of ASIP Meister is “.pdb”. To save the data with ASIP Meister, please specify the design data name with “.pdb” extension.

3.3. ASIP Meister Exit and Design Data Save

3.3.1. Saving New Design Data (or Saving Design Data under Different Name)

In [ASIP Meister Main Menu] window, click [File] → [Save Design] or [Save Design As ...] to open [Save] window as shown in Figure 4.

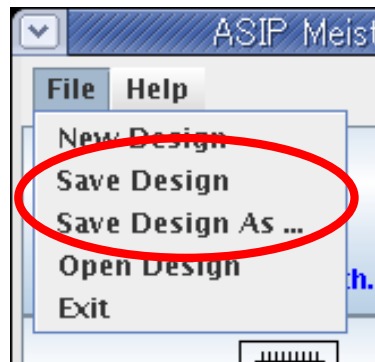


Figure 4 Save Design

Enter the design data name in [File name] field. From [Save In] pull-down menu, select the destination directory “tutorial”. Click [Save] button, to save the data and close the window.

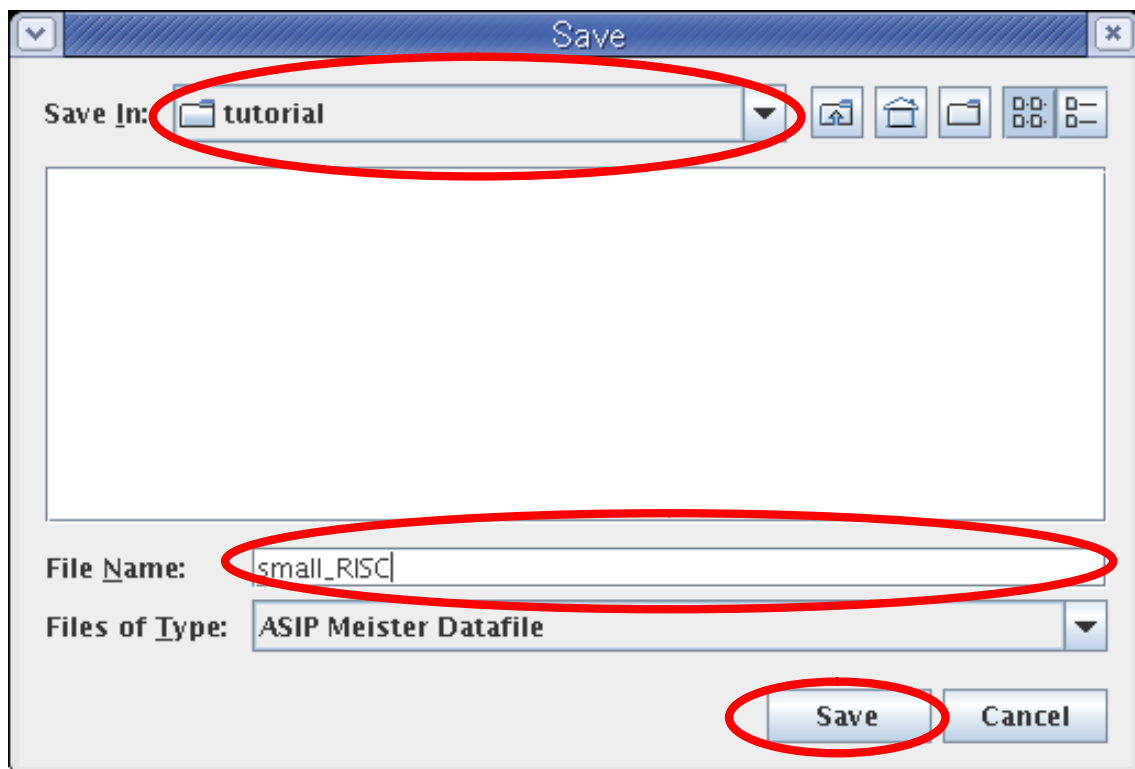


Figure 5 Data Save Design Box

The extension “.pdb” follows the saved file name. In this case, a file with the name “small_RISC.pdb” would be saved.

<Tip> When starting ASIP Meister, the user can specify the design data file name as a command line argument, to load design data previously saved in this file.

```
$ ASIPmeister small_RISC.pdb &
```

3.3.2. Updating Existing Design Data (Without Changing File Name)

In [ASIP Meister Main Menu] window, click [File] → [Save Design] menu. The current design data would be saved.

3.4. Closing Sub-Windows and Return to Main Window

After you have finished the design at the sub-window, click [Complete] check box and click [File] → [Close] menu to return to the main window. The check box cannot be checked if the information required for the completion of the sub-window is missing.

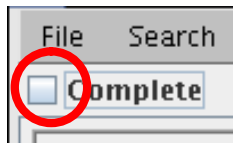


Figure 6 Complete Check Box

<Tip> Once the user check the [Complete] check box, the design at the sub-window is completed and the user can proceed to the next design stage sub-window. If the user closed the sub window without checking the [Complete] check box, the user will not be able to proceed to next design stage sub-window.

4. Processor Design Flow Using ASIP Meister

Open each sub-window in ASIP Meister, in the order specified in Figure 7. This would be typically the order of operation that should be performed to design and generate a processor core using ASIP Meister.

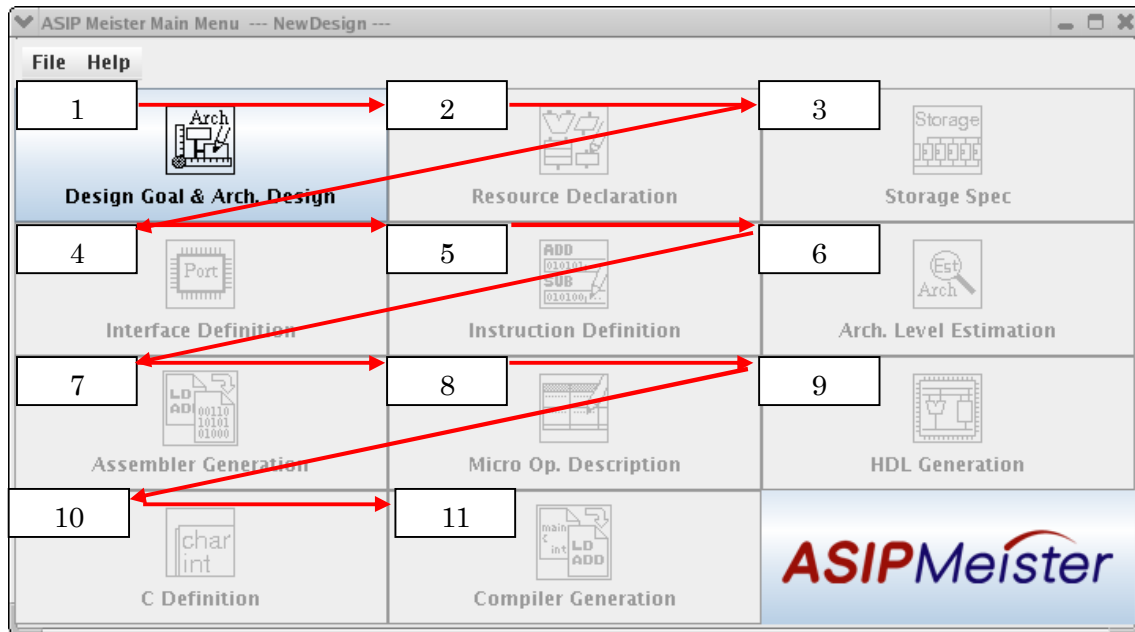


Figure 7 Design Flow Using ASIP Meister

Please follow the order below when designing the processor core in this tutorial (small_RISC).

- | | |
|-------------------------------------|--|
| 1. Design Goal & Arch. Design | Setting design goal values and pipeline stages |
| 2. Resource Declaration | Declaring resources |
| 3. Storage Specification Definition | Specifying storages |
| 4. Interface Definition | Defining I/O interfaces of the target processor |
| 5. Instruction Definition | Defining instruction types and instruction set |
| 6. Arch. Level Estimation | Estimating design quality of target processor |
| 7. Assembler Generation | Generating Assembler description files |
| 8. Micro Op. Description | Describing micro-operations for each instruction |
| 9. HDL Generation | Generating HDL files of target processor |
| 10. C Definition | Defining C code |
| 11. Compiler Generation | Generating compiler and Binutils |

<Note> [10. C Definition] and [11. Compiler Generation] are available exclusively for base processor **Brownie** provided in ASIP Meister Standard. You cannot generate a compiler without a processor extending **Brownie**.

4.1. Reading Memory Model Definition File

First, read the memory model definition file. In this tutorial, small_RISC is a processor whose instruction memory is single access cycle and data memory is multi-cycle access. The memory definition file appropriate to small_RISC is

“basefile/InstSingle-DataMulti.pdb”

By reading this file before starting design, you can save efforts when designing Interface Definition step later.

You can open a memory model definition file either by indicating it as a command line option or by using menu.

4.2. Setting Design Goal & Architecture Parameters

This section explains how to use [Design Goal & Arch. Design] window to set the processor type and architecture parameters.

Click [Design Goal & Arch. Design] in [ASIP Meister Main Menu] window to open the sub-window shown in Figure 9.



Figure 8 Design Goal & Arch. Design Button

Figure 9 [Design Goal & Arch. Design] Sub-window

4.2.1. Design Data Management Information

First, input the following information for managing design data;

Figure 10 Design Data Management Info.

Project name	Enter the design project name. Any letters can be used. For example, “ASIP Meister Tutorial” is used here.
Fhm workname	Current version does not support this parameter. FHM (Flexible Hardware Model) is a resource data base.

Revision No.	You can manage the version of the design data by entering information to this field. Any letters may be used. For example, {ver 1.0 : ASIP Meister Tutorial} is used here.
--------------	--

4.2.2. Design Goal Values

Design Goal	Goal Area	<input type="text" value="40000"/>	[gates]
	Goal Delay	<input type="text" value="20"/>	[ns]
	Goal Power S	<input type="text" value="4000"/>	[uW/MHz]
Design Priority <input checked="" type="radio"/> Area <input type="radio"/> Performance <input type="radio"/> Power			

Figure 11 Design Goal Input

Next, input the following design goals and identify the priority of design quality.

Design Goal	Goal Area	Area (Unit: [gates]) e.g. 40000[gate]
	Goal Delay	Maximum delay of internal circuits (Unit: [ns]) e.g. 20[ns]
	Goal Power S	Static power consumption (Unit: [μ W/MHz]) e.g. 4000 [μ W/MHz]
Design Priority	Select the design goal that has the highest priority among (Area/Delay/Power). <i>Area</i> is selected here.	

Finally, input the following architecture parameters;

4.2.3. Architecture Parameters

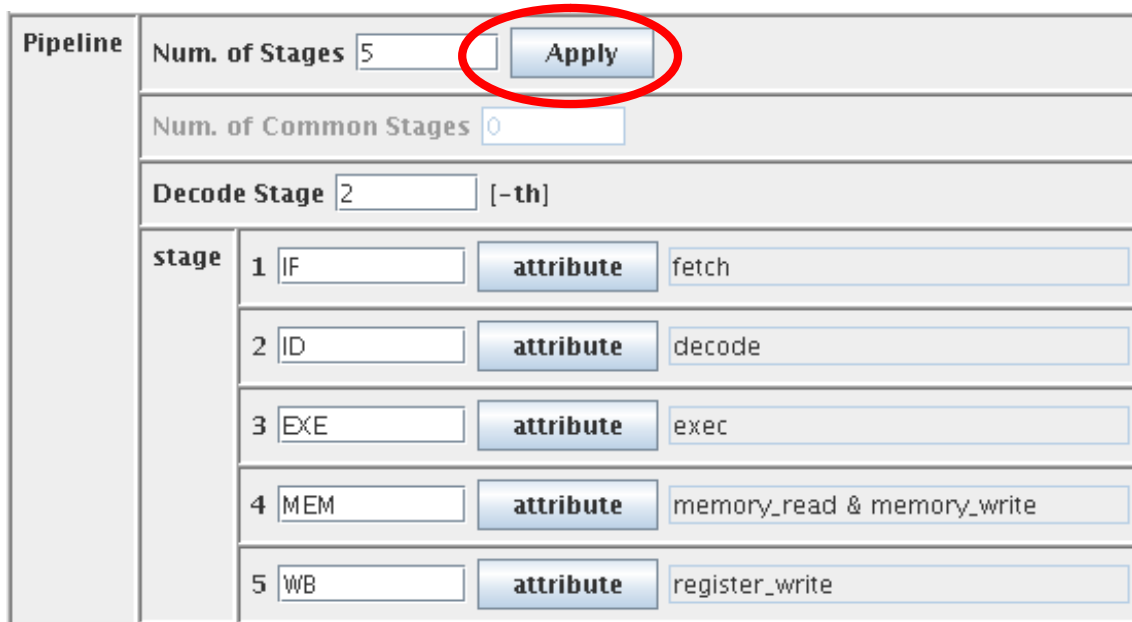
The screenshot shows a software window titled "Design Goal & Arch. Design". It contains a "File" menu, a "Search" field, and a "Help" button. Below these is a "Complete" checkbox. The main section is titled "CPU type" with a radio button selected for "Pipeline". Under the "Pipeline" section, there are several input fields and buttons: "Num. of Stages" (set to 5) with an "Apply" button, "Num. of Common Stages" (set to 0), "Decode Stage" (set to 2) with a "[-th]" label, and a table of stages. The table has 5 rows, each with a stage number, a name (IF, ID, EXE, MEM, WB), an "attribute" button, a description (fetch, decode, exec, memory_read & memory_write, register_write), and "Up" and "Down" buttons. Below the table are four rows of interlock and bypass settings, each with a label and two radio buttons (Yes/No): "Multi cycle interlock" (Yes selected), "Data hazard interlock" (No selected), "Register bypass" (No selected), and "Delayed branch" (Yes selected). Below these is a "Yes" label and a "Num. of delayed slot" (set to 1) with a "[instruction]" label. At the bottom are "Max inst. bit width" (set to 32) and "Max data bit width" (set to 32), both with "[bit]" labels. The last two rows are "Processor design" (radio buttons for "New Design" selected, "New Design with Base Processor", and "Base Processor Design") and "Use compiler" (radio buttons for "Yes" and "No" selected, with "No" selected).

Figure 12 Architecture Parameter Input

Enter the type of processor core and the number of pipeline stages to be designed. The explanation will be broken into several parts for simplicity. In the current version, you can select only [pipeline] as [CPU Type].

CPU Type	Current version only supports {pipeline} processors.
Pipeline	Specify the parameters for the pipeline processor.
Max inst. Bit width	Enter the maximum bit width of instruction.
Max data bit width	Enter the maximum bit width of data.
Processor design	Select processor design policy from "New Design", "New Design with Base Processor" and "Base Processor Design." Select "New

	Design” this time.
Use compiler	Choose whether you wish to generate a compiler. Select “No” this time.



Pipeline	Num. of Stages		5	Apply	
	Num. of Common Stages		0		
	Decode Stage		2	[-th]	
	stage	1	IF	attribute	fetch
		2	ID	attribute	decode
		3	EXE	attribute	exec
		4	MEM	attribute	memory_read & memory_write
		5	WB	attribute	register_write

Figure 13 Pipeline Stages Setting

Enter pipeline information as shown in the following table:

Number of Stages	Set the number of pipeline stages. For example, it is set to 5 here.
Number of Common Stages	This version does not support this setting.
Decode Stage	Set the position of decode handling stage.
Stage	Set the attribute for each stage.

First, enter {5} to [Num. Of Stages] field and click [Apply] button. The number of [Stage] fields will be enabled based on the number entered in [Num. of Stages] field. In this case, five rows will be enabled in [Stage] field.

Then, follow these procedures;

- Name each stage. This tutorial uses default names. However, when [attribute] button is clicked for each stage, a window is opened to select the stage attribute.

Click the necessary check boxes and click [OK] button.

Next, you can set pipeline characteristics. This version supports only delayed branch feature.

Multi cycle interlock	This version does not support this selection.
Data hazard interlock	This version does not support this selection.
Register bypass	This version does not support this selection.
Delayed branch	Enable/Disable Delayed branch.
Num. of delayed slot	Set the number of delayed slots (Delayed Branch Enabled).

The screenshot shows a configuration window with the following settings:

- Multi cycle interlock:** ☒ Yes ☐ No
- Data hazard interlock:** ☐ Yes ☒ No
- Register bypass:** ☐ Yes ☒ No
- Delayed branch:** ☒ Yes ☐ No
- Yes** (selected) **Num. of delayed slot:** [instruction]
- Max inst. bit width:** [bit]
- Max data bit width:** [bit]

Figure 14 Delayed Branch and Bit-width Setting

- Click [Yes] in [Delayed branch] field to enable [Num. of delayed slot] field. Enter {1} in [num. of delayed slot] field.
- Enter {32} to the [Max. inst. Bit width] and [Max. data bit width] fields.

You have now specified the processor design policy and the specification of the compiler. As describe above, select "New Design" and "No" respectively.

Now we have set all the necessary parameters for [Design Goal & Arch. Design] window. Click [Complete] button on the left top of the window and click [File] → [Close] to return to the main window.

In case you have an incorrect setting after closing the [Design Goal & Arch. Design]

window, you can click again on [Design Goal & Arch. Design] button to reopen the window, click on [Complete] check box to enable making modifications. The same procedure is applied to any other window throughout the design flow.

4.3. Resource Declaration

After the [Design Goal & Arch. Design] step is completed. The [Resource Declaration] button in the main window becomes activated.

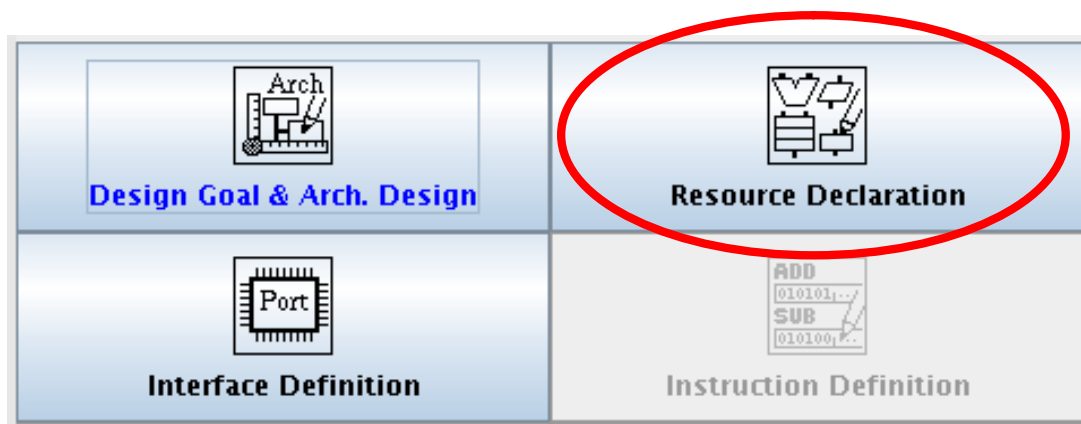


Figure 15 Resource Declaration Button

In the [Resource Declaration] step, you can select resources from FHM-DB (Flexible Hardware Model Database) and set parameters for these resources.

ASIP Meister original database, FHM-DB is used to provide hardware models. Functions of each hardware resource are described as a function set and it will be used in the later [Micro Op. description] window.

Click [Resource Declaration] button in the main window to open [Resource Declaration] window.

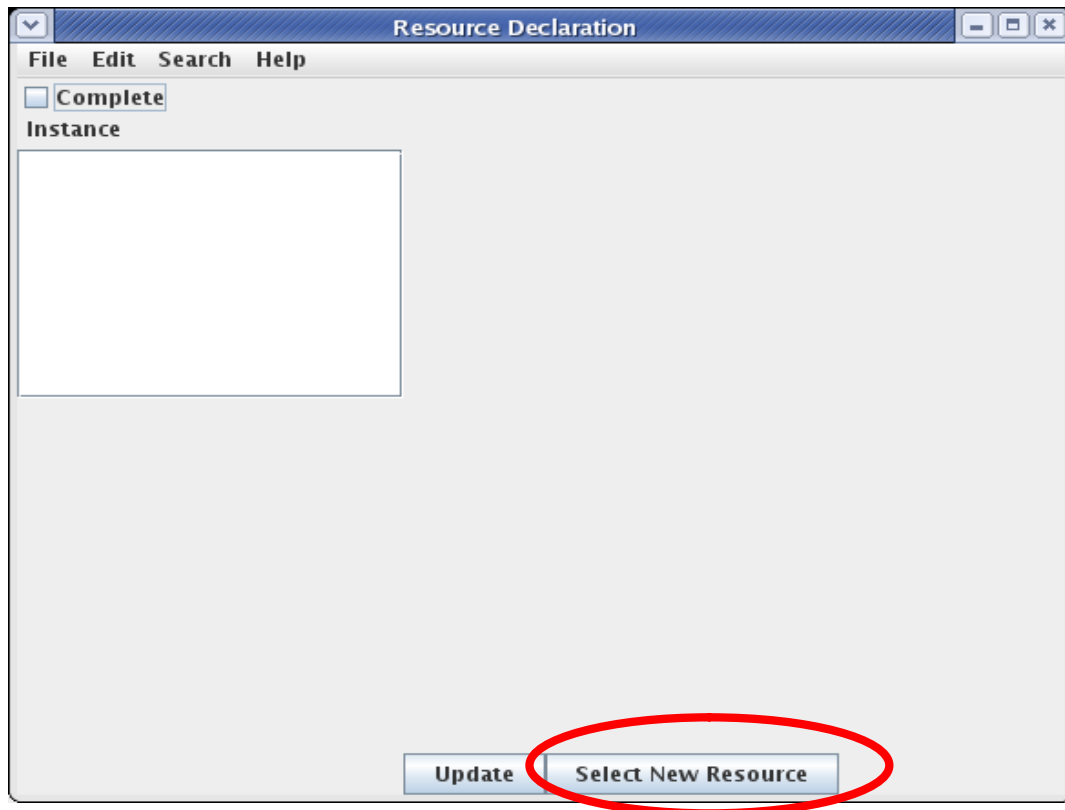


Figure 16 Resource Declaration Window

The above window shows that no resources have been registered yet. To select a resource from FHM-DB, click [Select New Resource] button to open [FHM Browser] window.

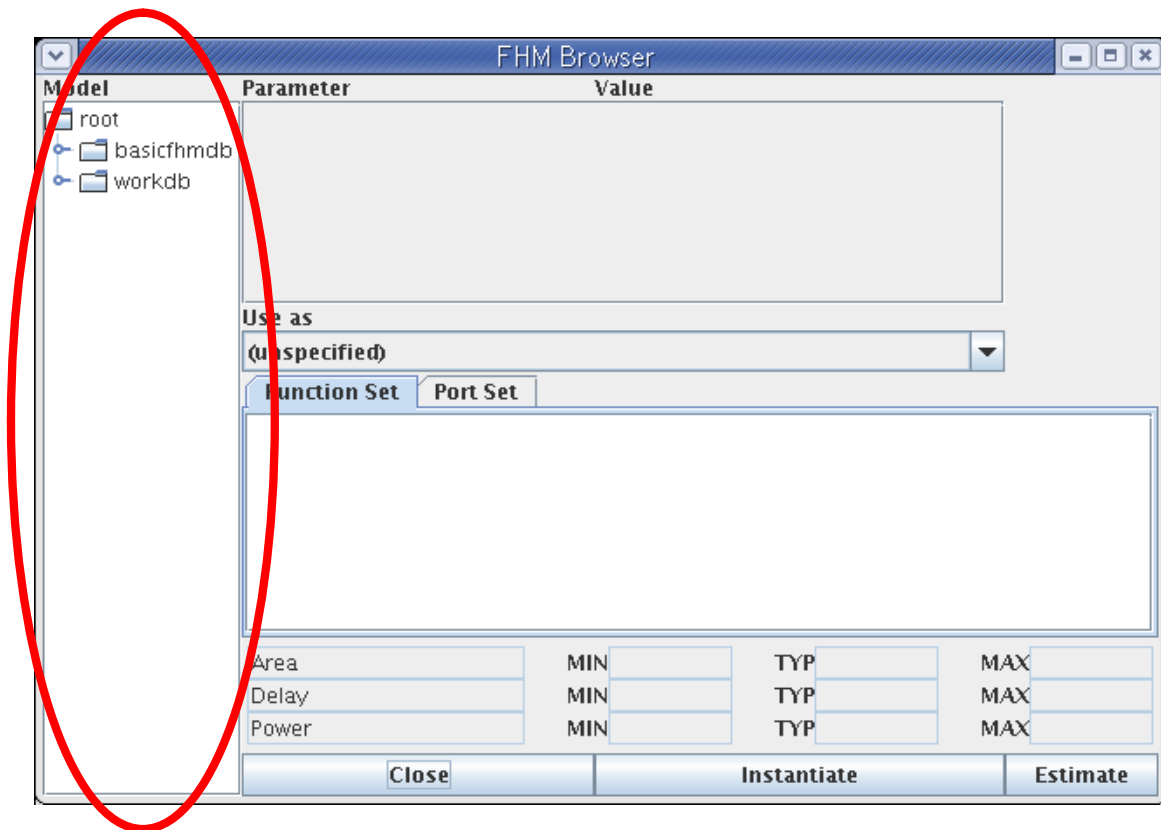


Figure 17 FHM Browser

When the [FHM Browser] is opened, the list of available FHMs will be displayed in the left side of [Model] list window.

When you select a resource, the parameters of this resource will be displayed in the window. The required parameters depend on the resource model. For the details of each resource in FHM-DB, refer to each [Function Set] tab.

For example, when you click [workdb] → [FHM_work] in [Model] list, a tree of available resources would be displayed. In this tutorial, we use [pcu] model for the program counter. When you click [pcu], [pcu] setting window opens as shown in Figure 18.

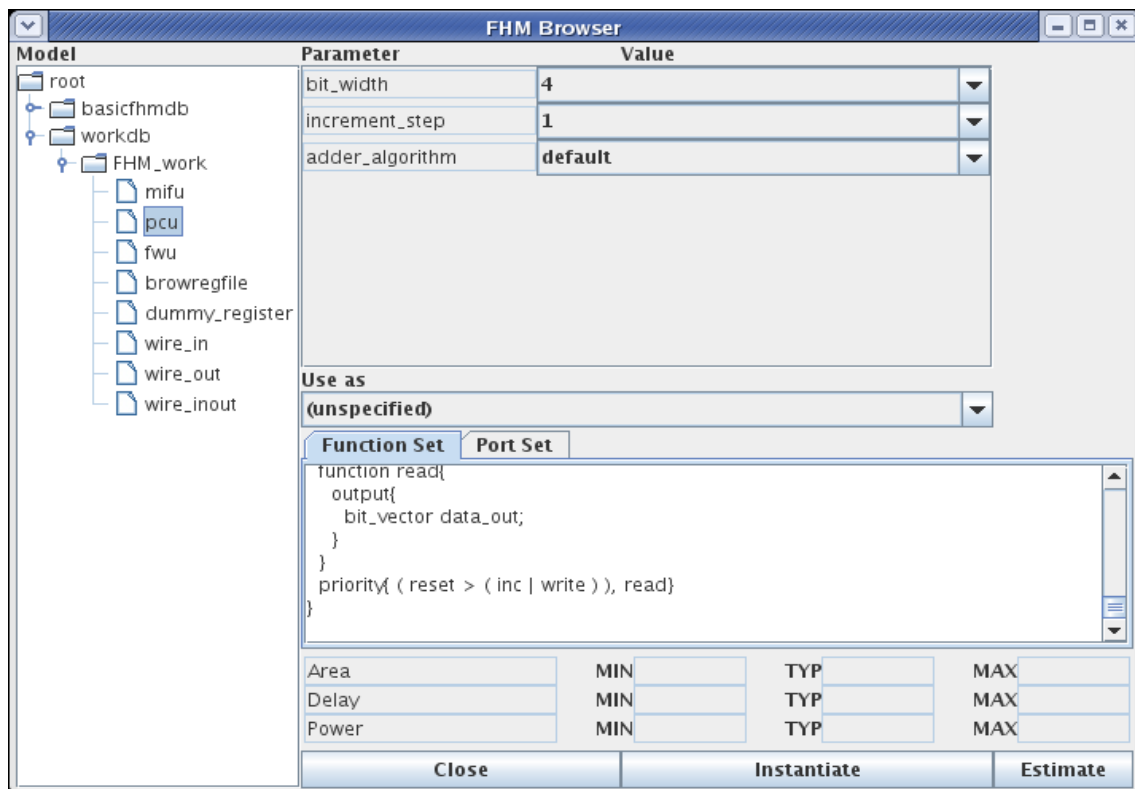


Figure 18 [FHM Browser] with [pcu] model info

You need to set the following parameters for the program counter [pcu]. To set the parameter, click the pull-down menu and select appropriate values from the menu.

bit width	Counter bit width
increment step	Number of increment steps
adder algorithm	Algorithm for increment
Use as	Model usage of resource

Figure 19 shows the value of each parameter for [pcu].

Parameter	Value
bit_width	32
increment_step	4
adder_algorithm	default
Use as	
	Prog. Counter

Figure 19 [pcu] Parameter Setting

Select a model usage from [Use as] pull-down menu, when the resource has a specific role in the processor. If the values in the pull-down menu do not match the role of the resource, always select [(unspecified)]. For example, the Use as for [pcu] is {Prog. Counter}.

After you have set the parameters for the resource, you can confirm the estimated values for this resource. Click [Estimate] button and check the estimated values for [Area], [Delay], and [Power].

Area	MIN 989.27	TYP 989.27	MAX 1230.91
Delay	MIN 0.83	TYP 0.87	MAX 0.87
Power	MIN 24.44	TYP 24.95	MAX 26.48
Close		Instantiate	Estimate

Figure 20 [pcu] Resource Estimate Button

Next, click [Instantiate] button to register the resource and [New Instance] window will open.

New Instance

Please input a new instance name.

PC

OK

Cancel

Figure 21 Instance Name Input

Based on {small_RISC} design specification, enter {PC} as a name of [pcu] and click [OK] button. The [Instance] frame in the left side of [Resource Declaration] window should show [PC] as a registered resource (See Figure 22).

The description in [Function Set] tab would be used later in [Micro-Op description] window, so confirm the content of [Function Set] sub window.

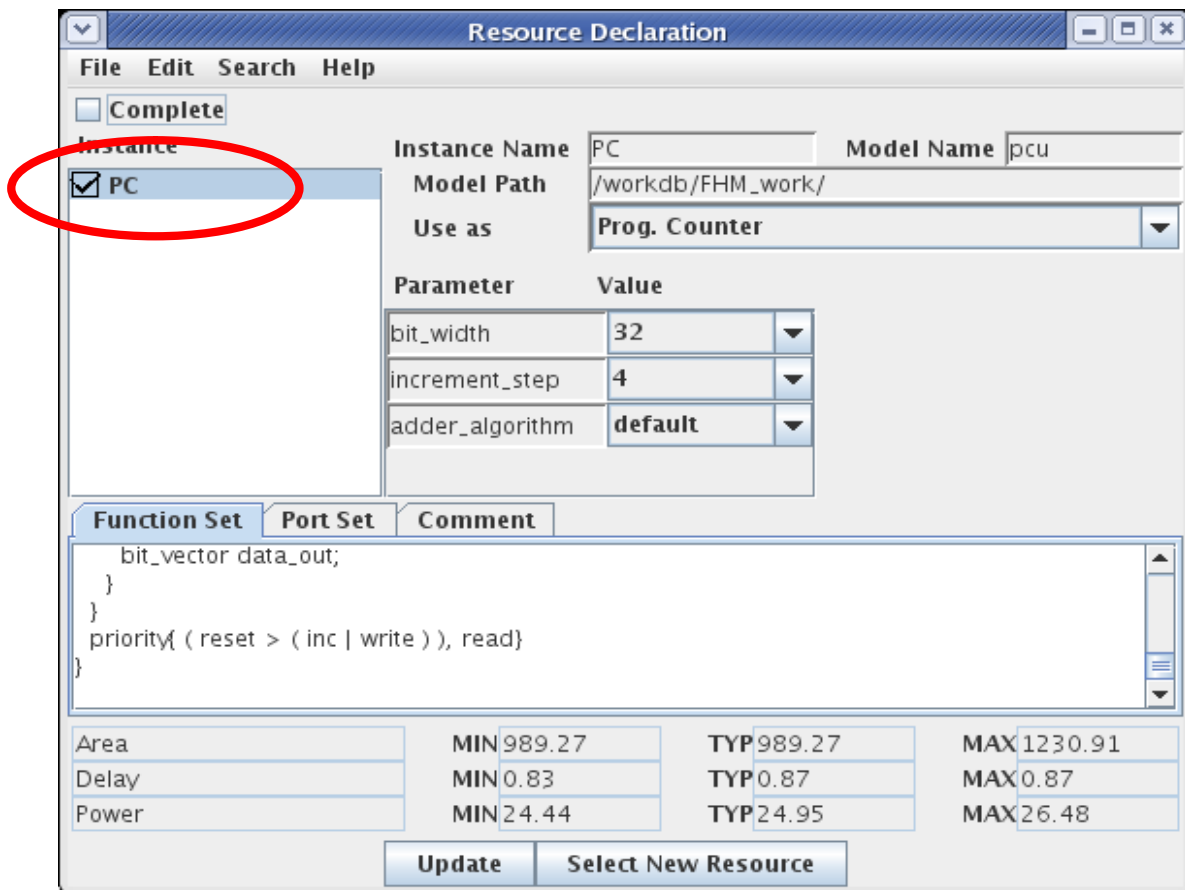


Figure 22 PC Resource Declaration Complete State

Confirm also the content of [Port Set] tab, including port names, signal directions, data types, and signal attributes.

Function Set	Port Set	Comment	
clock	in	bit	clock
async_reset	in	bit	reset
load	in	bit	ctrl
reset	in	bit	ctrl
hold	in	bit	ctrl
data_in	in	bit vector 31 0	data

Figure 23 PC Resource Ports Info.

The above declaration procedure applies for the remaining resources, i.e. Instruction register, register file, ALU, extender, instruction memory interface unit, and data memory interface unit.

Resource Declaration

File Edit Search Help

☐ Complete

Instance

- ☒ PC
- ☒ IR
- ☒ GPR
- ☒ ALU
- ☒ EXT
- ☒ IMIFU
- ☒ DMIFU

Instance Name IR **Model Name** register

Model Path /basicfmdh/storage/

Use as Inst. Register

Parameter **Value**

bit_width	32
-----------	----

Function Set **Port Set** **Comment**

```

function read{
  input{
  }
  output{
    bit_vector data_out = reg;
  }
}
priority{ ( reset > ( nop | write )), read }
}

```

Area	MIN 426.42	TYP 426.42	MAX 529.29
Delay	MIN 0.72	TYP 0.75	MAX 0.75
Power	MIN 17.05	TYP 17.22	MAX 17.23

Update **Select New Resource**

Figure 24 Declaration of Instruction Register

In Figure 24, FHM model name [register] has been declared as an Instruction register. Bit width is set to 32 bits. Model usage [Use as] is set to {Inst. Register} and Instance Name to {IR}.

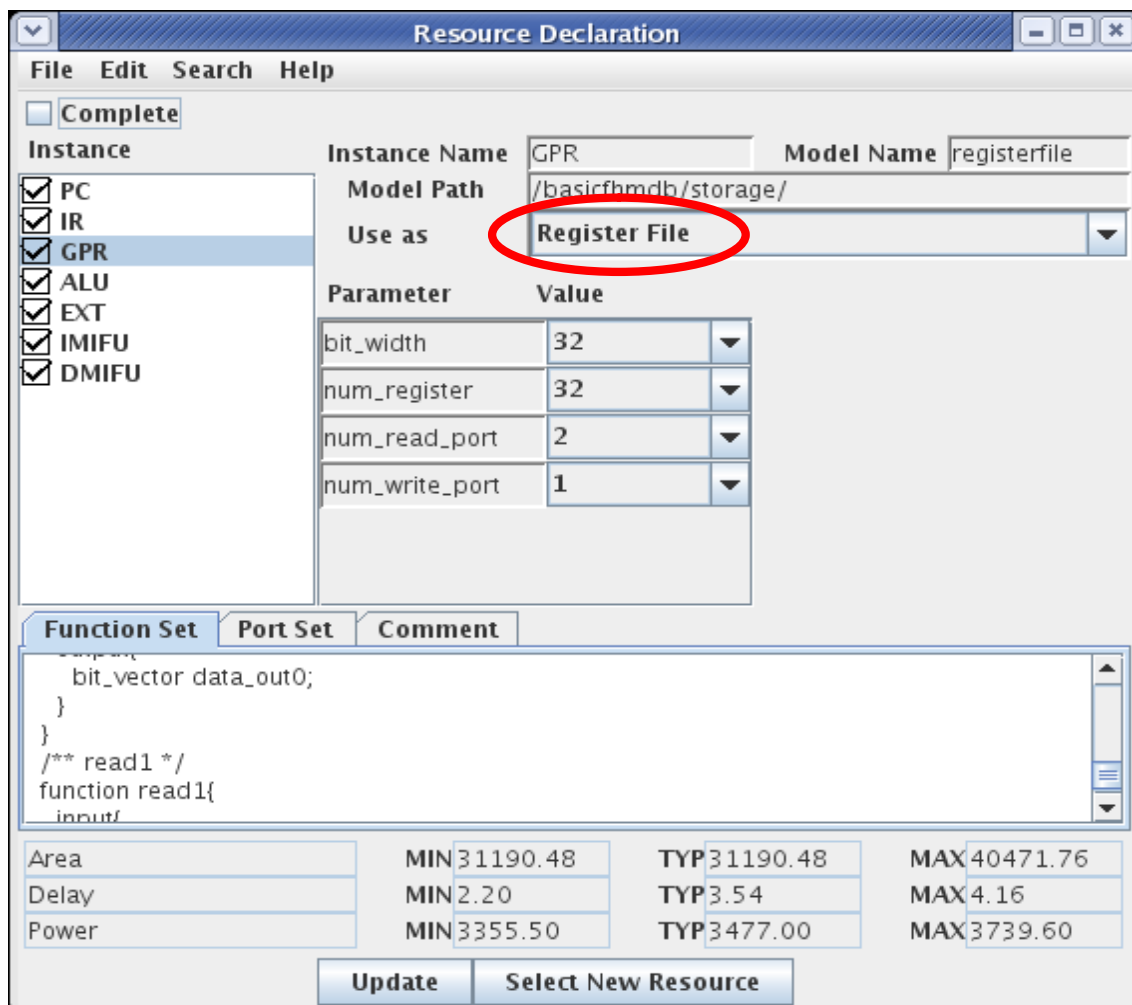


Figure 25 Declaration of Register File

In Figure 25, FHM model name [registerfile] has been declared as Register File. Bit width, number of registers, number of read ports, and write ports are set as shown in the above figure. Model usage [Use as] is set to {Register File} and Instance Name to {GPR}.

The screenshot shows the 'Resource Declaration' dialog box. The 'Instance' list on the left has checkboxes for PC, IR, GPR, ALU (selected), EXT, IMIFU, and DMIFU. The 'Instance Name' is 'ALU' and the 'Model Name' is 'alu'. The 'Model Path' is '/basicfhmdb/computational/'. The 'Use as' dropdown is set to '(unspecified)'. The 'Parameter' table shows 'bit_width' as 32 and 'algorithm' as default. The 'Function Set' tab is active, showing a code snippet. At the bottom, a table displays performance metrics for Area, Delay, and Power, with columns for MIN, TYP, and MAX values. 'Update' and 'Select New Resource' buttons are at the bottom right.

Parameter	Value
bit_width	32
algorithm	default

```

}
protocol{
  [mode == "10101" && cin = '1']{
    valid result;
    valid flag;
  }
}
}

```

Area	MIN	3175.17	TYP	3289.15	MAX	4147.98
Delay	MIN	19.65	TYP	23.42	MAX	31.02
Power	MIN	266.09	TYP	302.42	MAX	302.42

Figure 26 Declaration of ALU

In Figure 26, FHM model name [ALU] is declared as an arithmetic and logic unit. Bit width and operational algorithm are set as shown in the above figure. Model usage [Use as] is set to {Unspecified} and Instance Name to {ALU}.

Resource Declaration

File Edit Search Help

☐ Complete

Instance Name: EXT Model Name: []

Model Path: /basicfhmdb/computational/

Use as: (unspecified)

Parameter	Value
bit_width	16

Function Set | Port Set | Comment

```

/** sign : sign extension */
function sign{
  input{
    unsigned data_in;
  }
  output{
    unsigned data_out = exts(a);
  }
  control{
    in mode;
  }
  protocol{
    [mode == '1']{
      valid data_out;
    }
  }
}

```

Area	MIN 13.65	TYP 13.65	MAX 68.59
Delay	MIN 0.18	TYP 0.32	MAX 0.32
Power	MIN 0.83	TYP 0.83	MAX 7.20

Update Select New Resource

Figure 27 Declaration of Extender

In Figure 27, FHM model name [extender] is declared as an extender. Input bit width is set to {16} bits and output (extended) bit width is set to {32} bits. Model usage [Use as] is set to {Unspecified} and Instance Name to {EXT}.

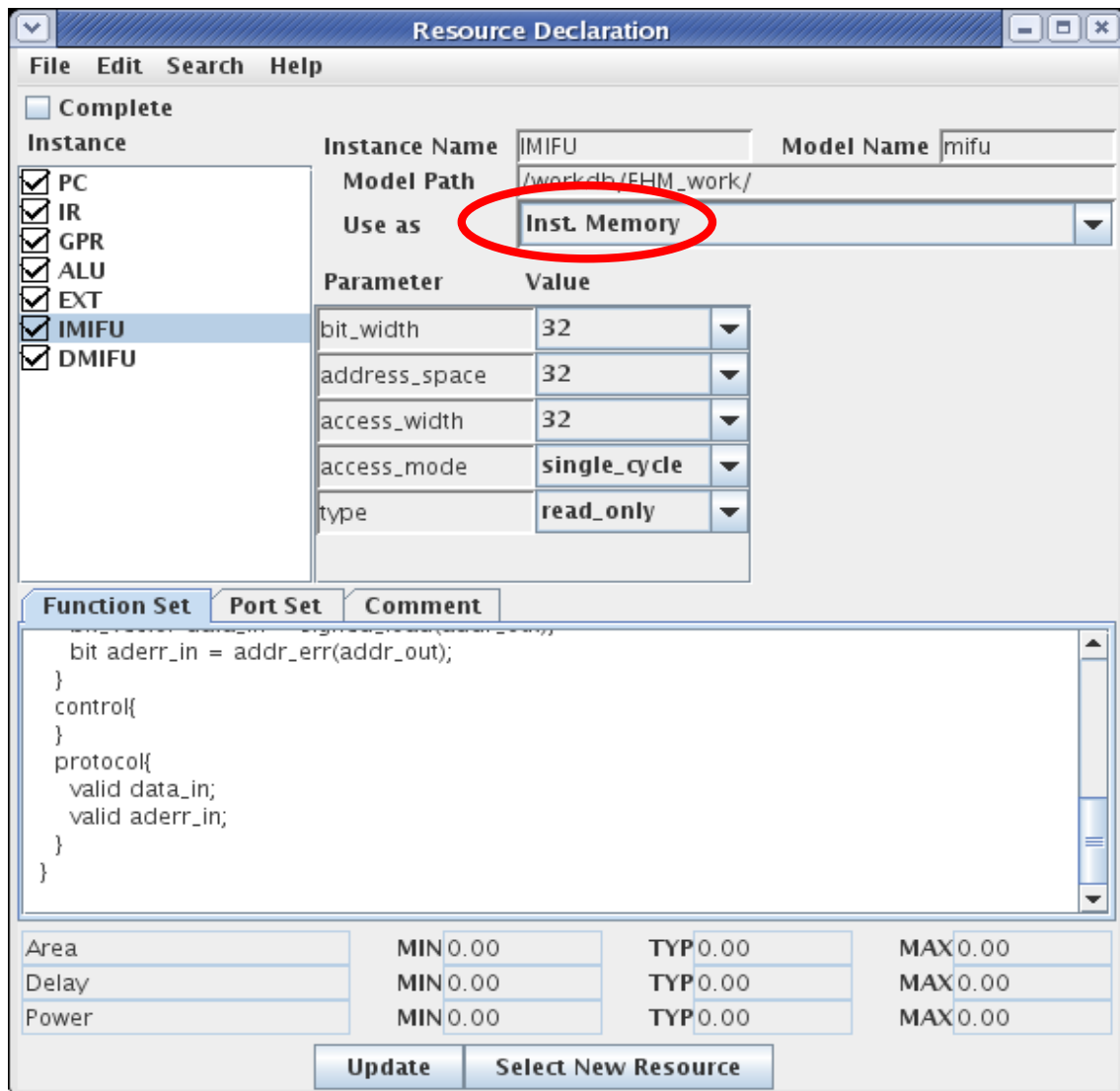


Figure 28 Declaration of Instruction Memory Interface Unit

In Figure 28, FHM model name [mifu] is declared as an instruction memory interface unit. Bit width, address space, access width, access mode, and type are set as shown in the above figure. Model usage [Use as] is set to {Inst. Memory} and Instance Name to {IMIFU}.

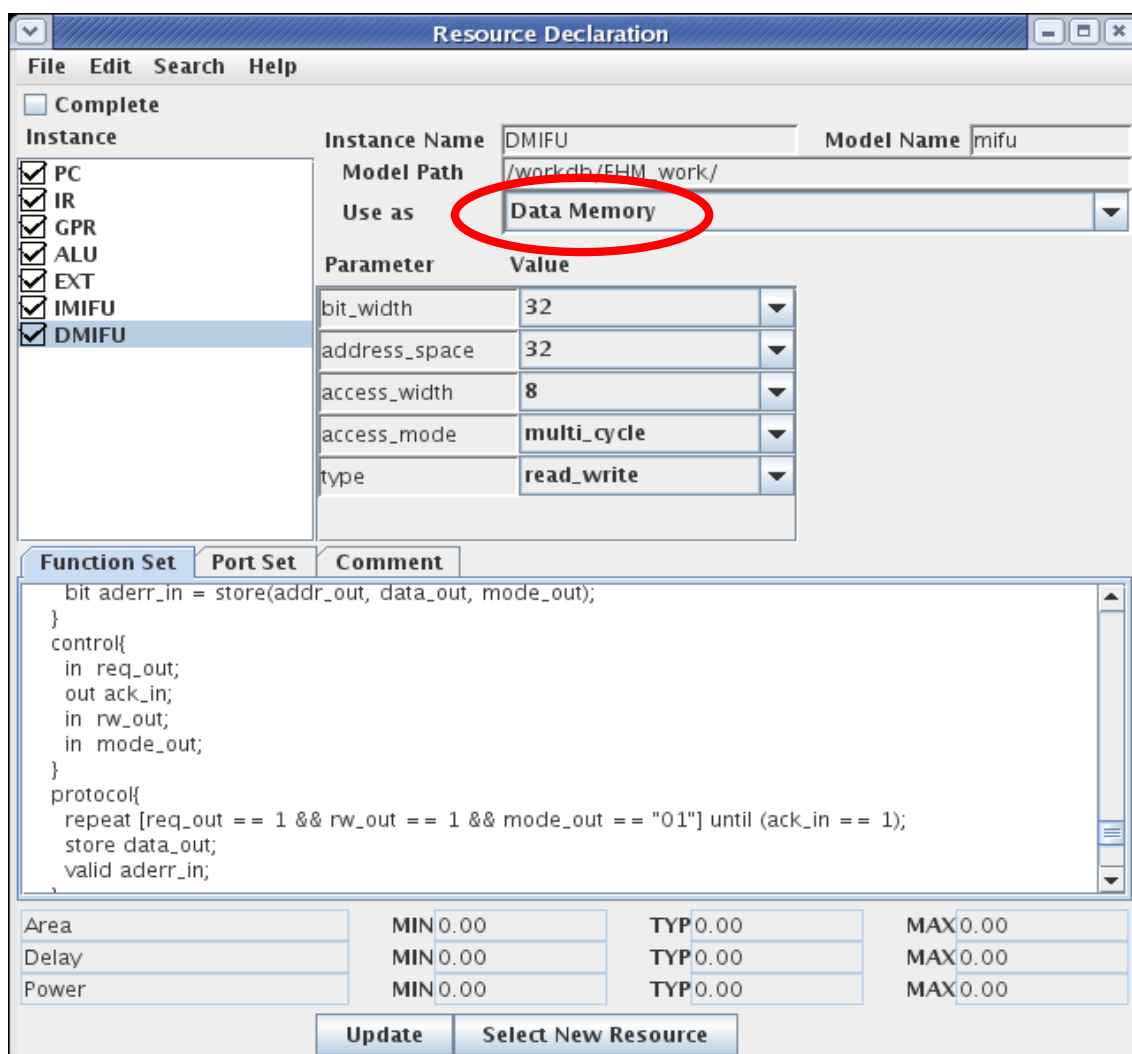


Figure 29 Declaration of Data Memory Interface Unit

In Figure 29, FHM model name [mifu] is declared as a data memory interface unit. Bit width, address space, access width, access mode, and type are set as shown in the above figure. Model usage [Use as] is set to {Data Memory} and Instance Name to {DMIFU}.

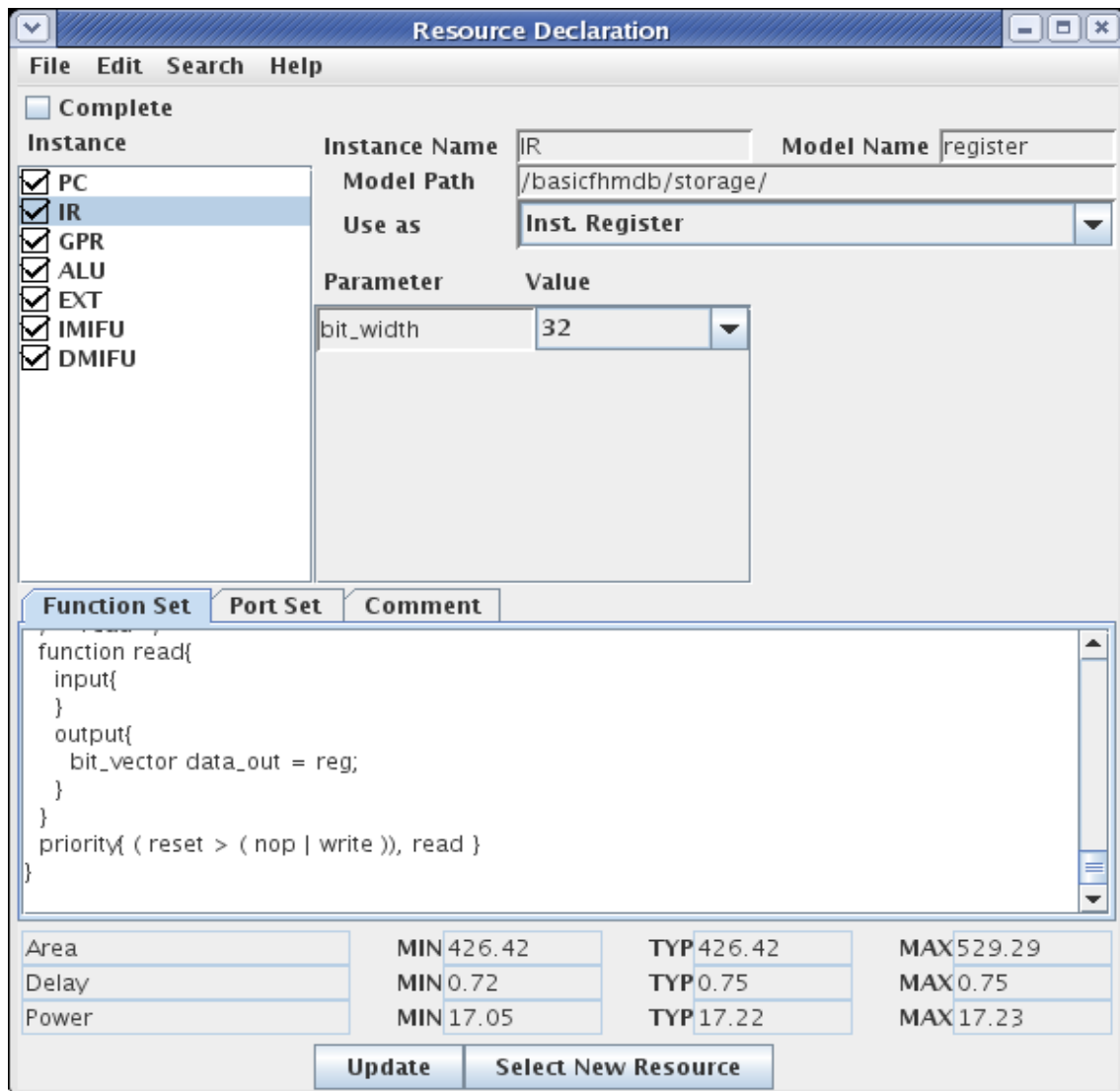


Figure 30 Resource Declaration Completed

After declaring all resources, the [Resource Declaration] window would be as shown in Figure 30. In [Instance] list, you can click on each resource check box, to activate/deactivate the resource module. Now all resource modules are activated.

After declaring all resource, click on [complete] check box. Click [File] → [Close] to close the [Resource Declaration] window.

4.4. Storage Specification Definition

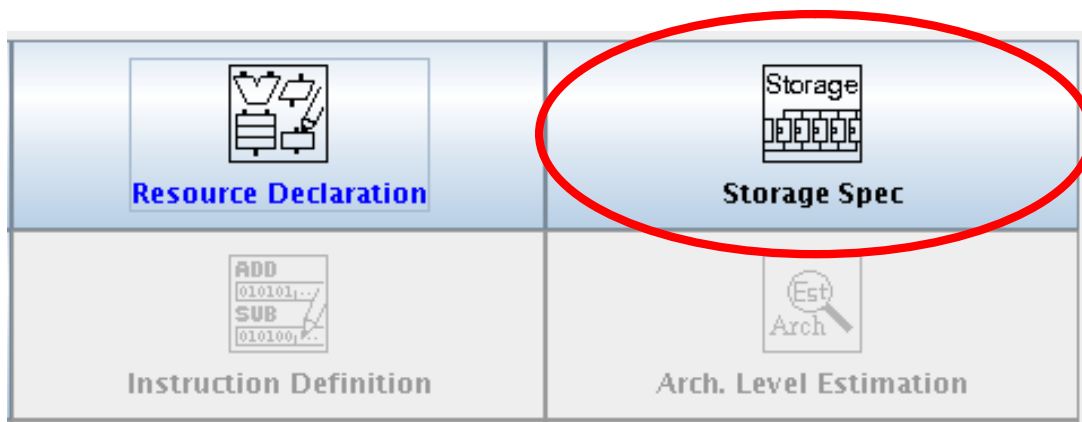


Figure 31 Storage Spec. Button

After completing [Resource Declaration] step, [Storage Spec] button becomes activated. Click on this button to open [Storage Spec] window as shown in Figure 32.

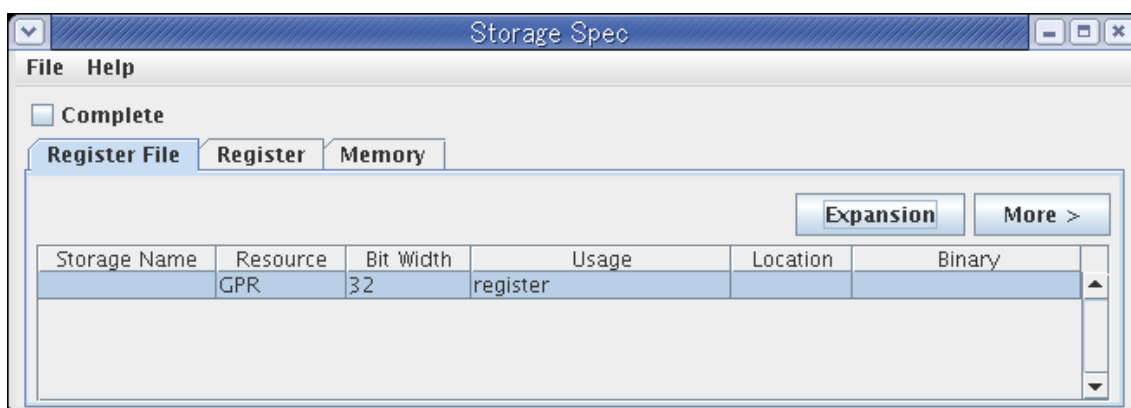


Figure 32 Storage Spec. Window

In this design phase, designers define storages for the processor such as general purpose registers, program counter, instruction register, and so on. The information of storage is used for instruction definition. Storage specification consists of three windows selected by tabs; register file specification, register specification, and memory specification. The following sections explain each window.

4.4.1. Register File Specification

In this window, register file specification is defined. You should find a register file resource when you click storage specification button in the main window, because the register file resource has already been declared in the previous step [Resource Declaration].

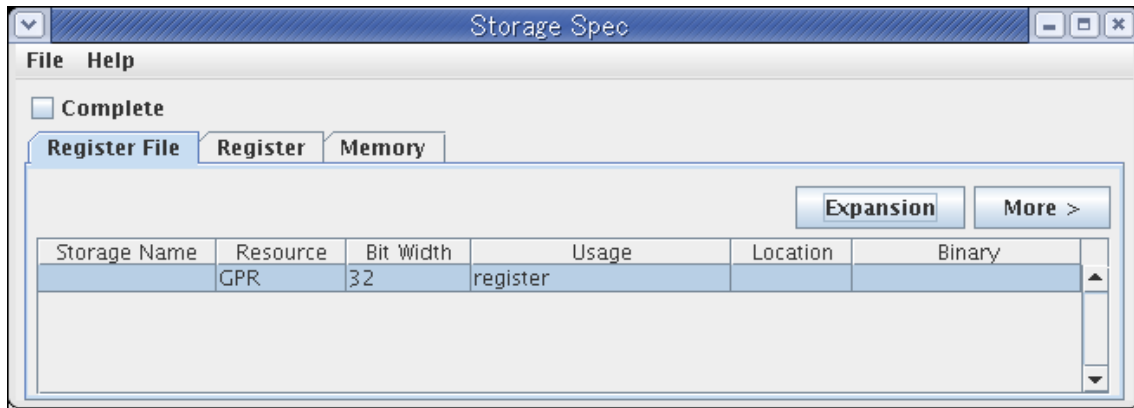


Figure 33 Register File Spec. Setting (Before)

Specify storage name, bit width, usage, location, and binary representation in this tab. The following table outlines the register file specification.

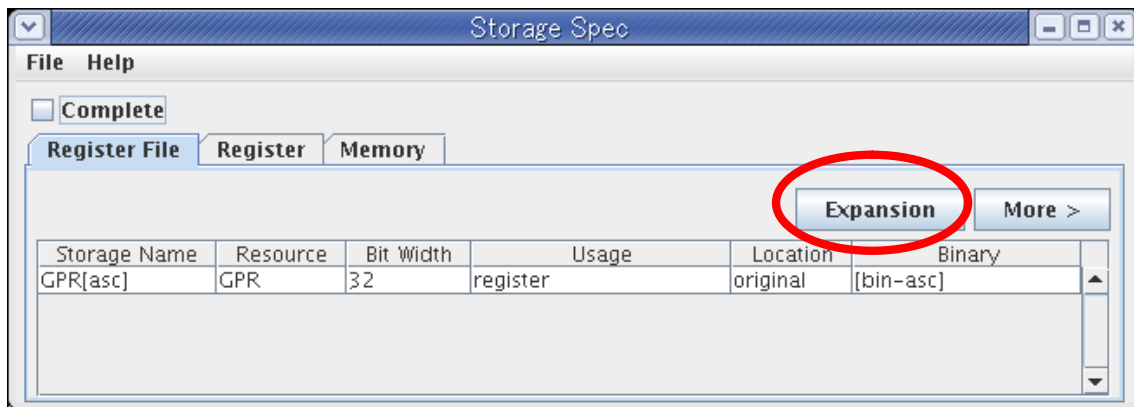


Figure 34 Register File Spec. Setting (After)

Storage Name	Storage Name used in Assembly Code
Resource	Resource instance name. The instance name must be chosen from the instance names declared in Resource Declaration step.
Bit width	The bit width of storage. Use the same width declared in Resource Declaration step.
Usage	The usage of each register. For example, program counter, zero register, stack pointer, and so on.

Location	Location of each register. This part is used when you would like to define overlapped registers.
Binary	Binary representation of each register in a register file. This information is used in the assembler.

A register file consists of multiple registers indexed using a GPR[] notation. Indexing can be either in ascending or descending order, as outlined below.

[asc]	Ascending order: GPR[asc]
[des]	Descending order: GPR[des]

Also a binary representation of each register can be specified in either ascending or descending order, as outlined below.

[bin-asc]	Binary ascending order.
[bin-des]	Binary descending order.

After declaring above information, click [expansion] button. The following message dialog will be displayed. Press [OK] to open [Expansion Frame] window, shown in Figure 36.

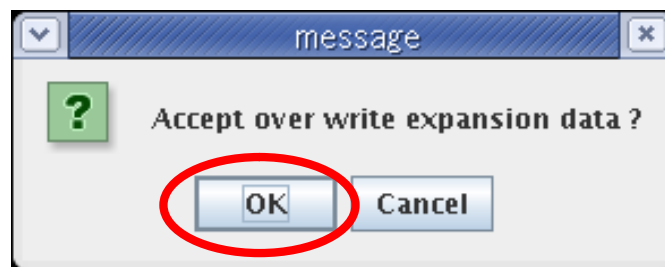


Figure 35 Register File Expansion Data Confirm Message

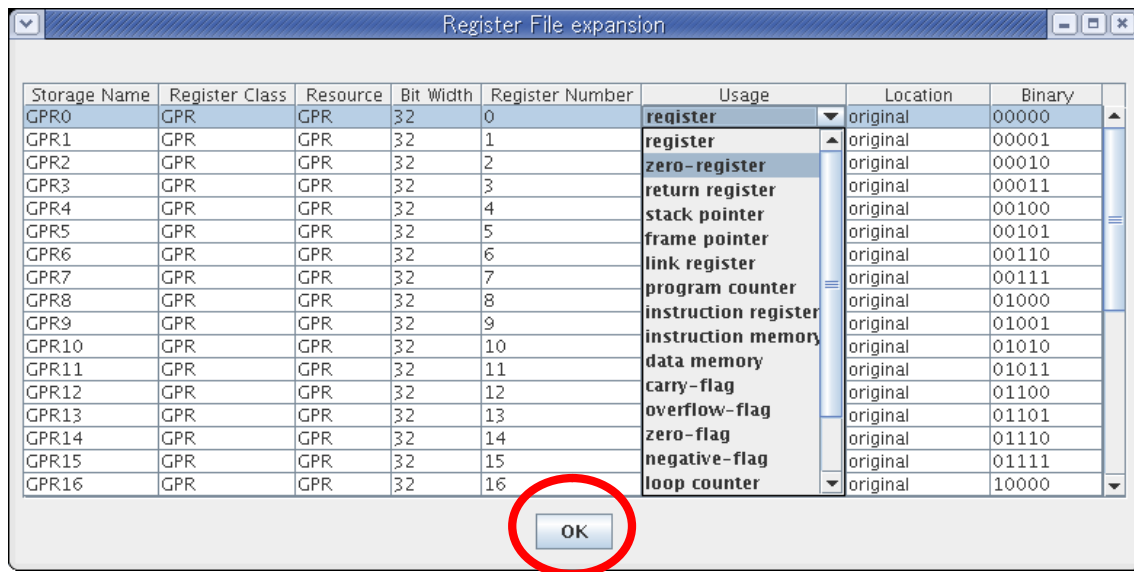


Figure 36 Detailed Info. of Register File Spec.

Each register can be seen in expansion frame window. You can specify usage of each register in this part. In this tutorial, usage of {small_RISC} registers will be set as follows;

Register	Usage
GPR0	zero register
GPR28	stack pointer
GPR29	frame pointer
GPR30	link register
GPR31	return register

After setting register usage, click [OK] button to close [Register File expansion] window.

4.4.2. General Register Specification

When you click [Register] tab, register definition window (Figure 37) will be displayed. In this window, you can specify the following information; storage name, resource, bit width, usage and location. This information is the same as register file definition.

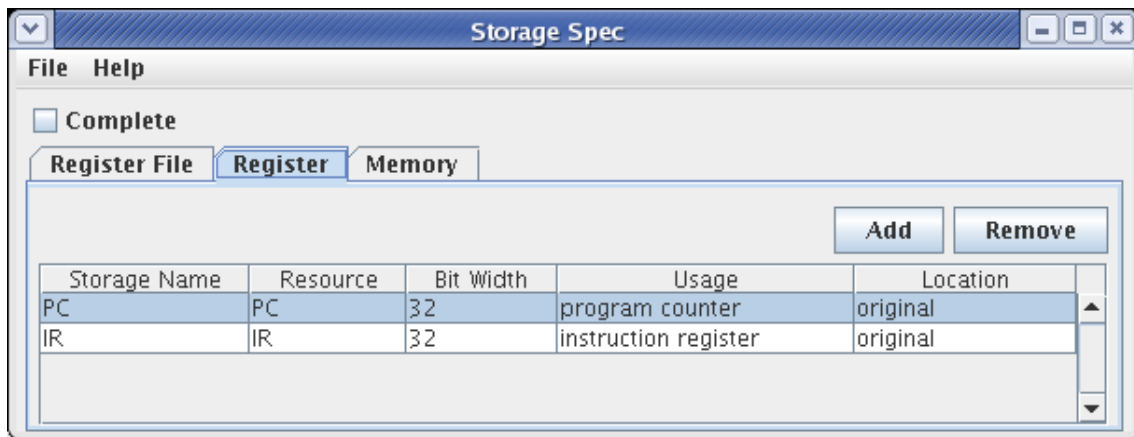


Figure 37 Register Spec. Setting

Fill in the information for program counter and instruction register. For the program counter, use the following specification; {*Storage name*: PC, *Resource*: PC, *Bit width*: 32, *Usage*: program counter, *Location*: original}. For the instruction register, use the following specification; {*Storage name*: IR, *Resource*: IR, *Bit width*: 32, *Usage*: instruction register, *Location*: original}

4.4.3. Memory Specification

When you click [Memory] tab, memory specification window (Figure 38) will be displayed. In this window, you can specify the following information; storage name, resource, bit width, usage, and access bit. Access bit indicates minimum bit width when a processor accesses memory.

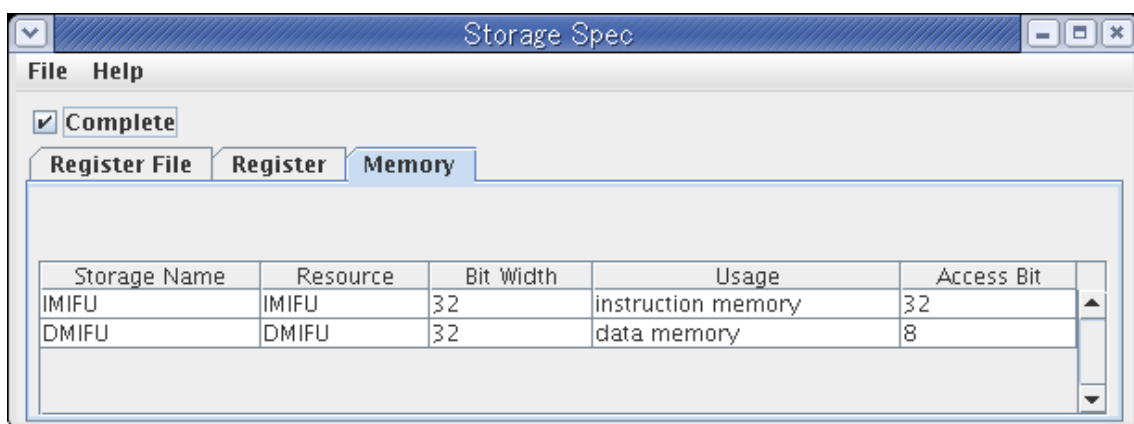


Figure 38 Memory Spec. Setting

Fill in information for instruction memory and data memory interface units. For instruction memory interface unit, use the following specification; {*Storage name*:

IMIFU, *Resource*: IMIFU, *Bit width*: 32, *Usage*: instruction memory, *Access bit*: 32}.

For data memory interface unit, use the following specification; {*Storage name*: DMIFU, *Resource*: DMIFU, *Bit width*: 32, *Usage*: data memory, *Access bit*: 32}.

When you finish describing the storage specification, click the complete button and close the [Storage Spec.] window.

4.5. Interface Definition

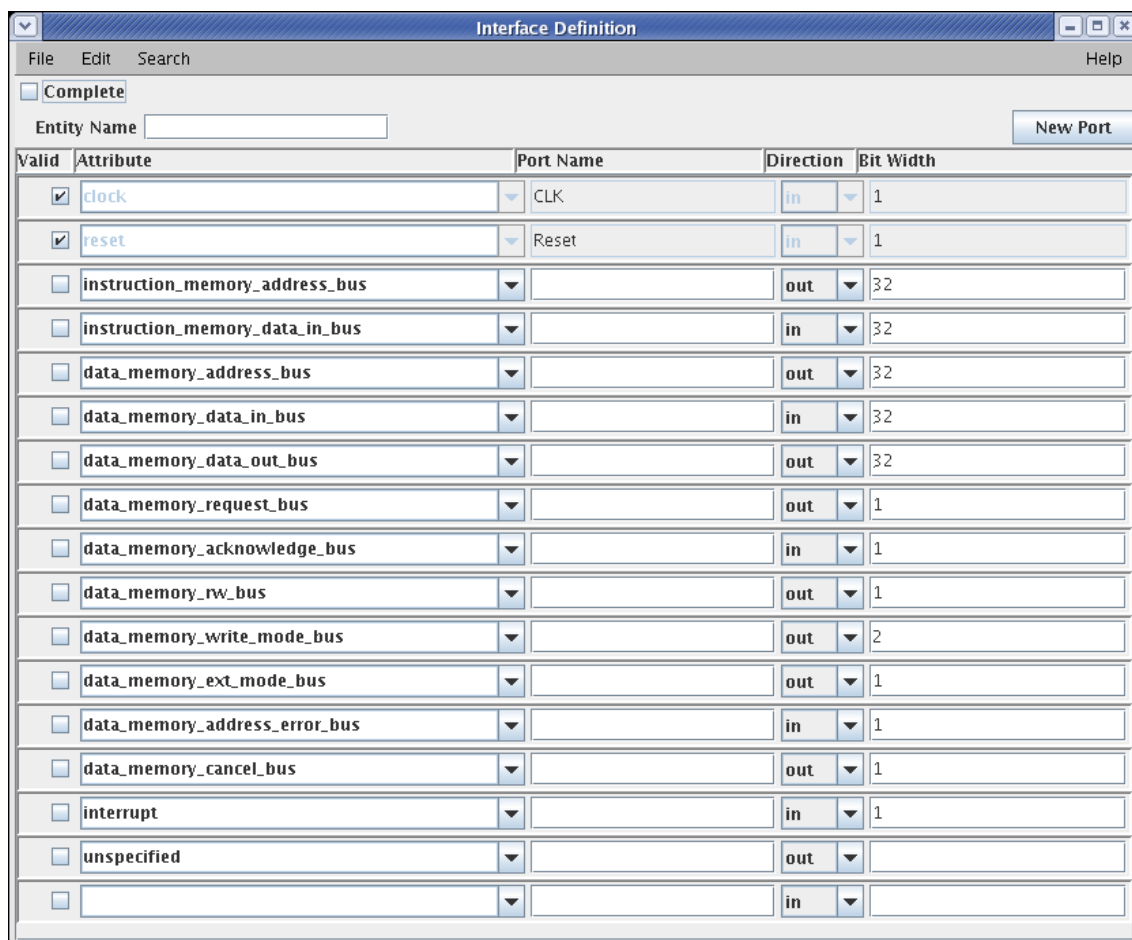
This section explains how to set the entity name of the processor and I/O ports information. There are several types of ports you can declare:

- (1) External access ports for IMIFU and DMIFU.
- (2) Clock and Reset ports.
- (3) Ports for interrupts.

Click [Interface Definition] button (Figure 39) to open [Interface Definition] window (Figure 40). The window opens with the default ports displayed.



Figure 39 Interface Definition Button



The screenshot shows the 'Interface Definition' window. It has a menu bar with 'File', 'Edit', 'Search', and 'Help'. Below the menu bar is a 'Complete' checkbox and an 'Entity Name' text field. A 'New Port' button is located on the right. The main area is a table with the following columns: 'Valid', 'Attribute', 'Port Name', 'Direction', and 'Bit Width'.

Valid	Attribute	Port Name	Direction	Bit Width
<input checked="" type="checkbox"/>	clock	CLK	in	1
<input checked="" type="checkbox"/>	reset	Reset	in	1
<input type="checkbox"/>	instruction_memory_address_bus		out	32
<input type="checkbox"/>	instruction_memory_data_in_bus		in	32
<input type="checkbox"/>	data_memory_address_bus		out	32
<input type="checkbox"/>	data_memory_data_in_bus		in	32
<input type="checkbox"/>	data_memory_data_out_bus		out	32
<input type="checkbox"/>	data_memory_request_bus		out	1
<input type="checkbox"/>	data_memory_acknowledge_bus		in	1
<input type="checkbox"/>	data_memory_rw_bus		out	1
<input type="checkbox"/>	data_memory_write_mode_bus		out	2
<input type="checkbox"/>	data_memory_ext_mode_bus		out	1
<input type="checkbox"/>	data_memory_address_error_bus		in	1
<input type="checkbox"/>	data_memory_cancel_bus		out	1
<input type="checkbox"/>	interrupt		in	1
<input type="checkbox"/>	unspecified		out	
<input type="checkbox"/>			in	

Figure 40 Interface Definition Window

To declare a new port, click [New Port] button. This adds a new row to the window. In this tutorial, refer to {small_RISC} specification and add the I/O ports using the following guidelines.

4.5.1. Entity Name Setting

The name entered in this field is used as an entity name of the processor in the VHDL description. The file name of VHDL description of the processor becomes the name with the extension “.vhd”. Note that the name should conform to the VHDL naming conventions.

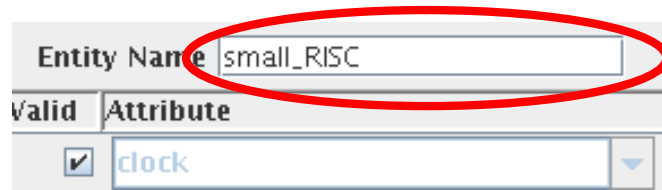


Figure 41 Entity Name Setting

4.5.2. I/O Ports Settings

This includes the following settings;

- [Port name]: Port name

The name entered in this field is used as an external port name for the processor in the VHDL description. The name should conform to the VHDL naming conventions.

- [Port name]: Port name

The name entered in this field is used as an external port name for the processor in the VHDL description. The name should conform to the VHDL naming conventions.

- [Direction]: Direction of the Port

Select [in] for input, [out] for output and [inout] for both.

- [Attribute]: Attribute keyword

Select a proper keyword based on the usage of the port

After setting all I/O Ports, the [Interface Definition] window should look as shown in Figure 42.

The 'Interface Definition' window shows the configuration for the 'small_RISC' entity. It includes a 'Complete' checkbox and a 'New Port' button. The main table lists various I/O ports with their attributes, names, directions, and bit widths.

Valid	Attribute	Port Name	Direction	Bit Width
<input type="checkbox"/>	clock	CLK	in	1
<input type="checkbox"/>	reset	Reset	in	1
<input type="checkbox"/>	instruction_memory_address_bus	InstAB	out	32
<input type="checkbox"/>	instruction_memory_data_in_bus	InstDB_in	in	32
<input type="checkbox"/>	data_memory_address_bus	DataAB	out	32
<input type="checkbox"/>	data_memory_data_in_bus	DataDB_in	in	32
<input type="checkbox"/>	data_memory_data_out_bus	DataDB_out	out	32
<input type="checkbox"/>	data_memory_request_bus	DataReq	out	1
<input type="checkbox"/>	data_memory_acknowledge_bus	DataAck	in	1
<input type="checkbox"/>	data_memory_rw_bus	DataRW	out	1
<input type="checkbox"/>	data_memory_write_mode_bus	DataMode	out	2
<input type="checkbox"/>	data_memory_ext_mode_bus	DataExt	out	1
<input type="checkbox"/>	data_memory_address_error_bus	DataAddrError	in	1
<input type="checkbox"/>	data_memory_cancel_bus	DataCancel	out	1
<input type="checkbox"/>	interrupt		in	1
<input type="checkbox"/>	unspecified		out	

Figure 42 I/O Ports Settings

After setting all I/O ports, click on the [valid] check box to indicate its validity. In case of modifying any of the settings, click again on [valid] check box to allow making the modification.

This close-up shows the 'Valid' column of the table. The 'clock' and 'reset' rows have their checkboxes checked, while the 'instruction_memory_address_bus' row has its checkbox unchecked. A red circle highlights the unchecked checkbox, and a callout box labeled 'Check' points to it.

Valid	Attribute
<input checked="" type="checkbox"/>	clock
<input checked="" type="checkbox"/>	reset
<input type="checkbox"/>	instruction_memory_address_bus

Figure 43 I/O Port Validity

After setting [Interface Definition] window, check [Complete] check box and click [File] → [Close] to return to main window.

4.6. Instruction Set Definition

This section explains the declaration process of instruction types and processor core instruction set in ASIP Meister.



Figure 44 Instruction Definition Button

Click [Instruction Definition] button in the main window (Figure 44) to open [Instruction Definition] window, as shown in Figure 45.

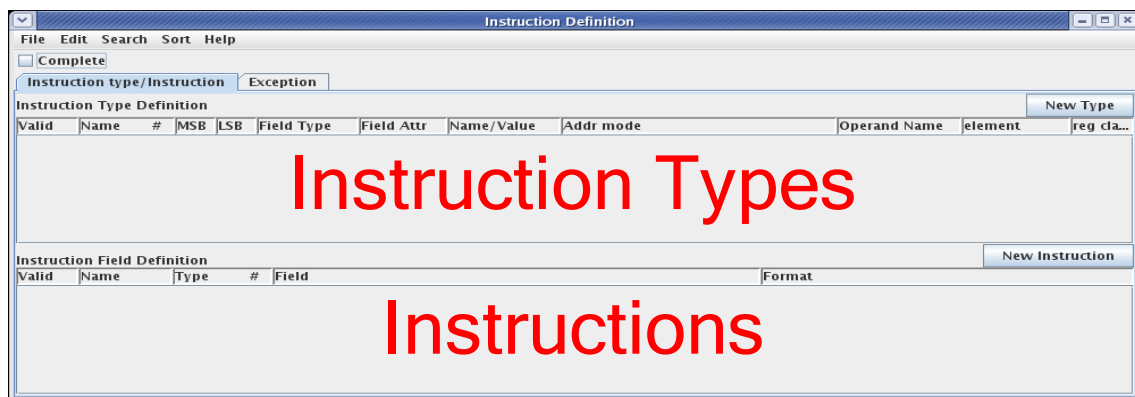


Figure 45 Instruction Type and Instruction Definition

4.6.1. Instruction Set Definition in ASIP Meister

Select [Instruction type /Instruction] tab from [Instruction Definition] window to define the instruction set.

In ASIP Meister instruction definition, define instruction types and specify how to separate the instruction code field. Declare instructions of each instruction type, simultaneously assigning the characteristic values (Op-code) to the instructions.

4.6.2. Instruction type definition

In this tutorial you will declare 4 instruction types: {R_R, R_I, B, and JP_T}.

To declare a new instruction type, click [New Type] button to open [New Instruction Type Confirm] window. This window has two fields to enter the name of a new instruction type and the number of the fields for the instruction type.

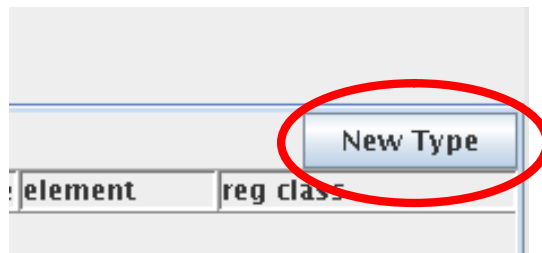


Figure 46 New Type Button

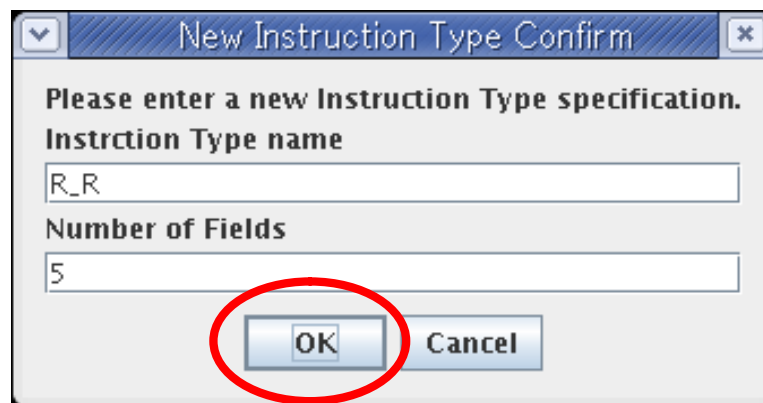


Figure 47 [New Instruction Type Confirm] Window

Enter {R_R} and {5} to each field and click [OK] button. Five rows would appear in [Instruction Type Definition Window] as shown in Figure 48. Within this window, you can declare bit fields of the instruction type and unique binaries for the instruction type as follows.

Figure 48 New Instruction Type Setting

MSB, LSB	Enter the value of most significant bit (MSB) and least significant bit (LSB).
Field Type, Field Attr	Description follows later.
Value	Enter binary value or name to each field.
Addr mode	Enter addressing mode of each operand.
Operand Name	Specify the operand name for assembly language.
Element, reg class	Specify which resource is assigned to each operand field, and when the addressing mode of the operand uses registers, specify the register class that the operand can use.

For [Field Type], click pull-down menu to select a value from (Op-code, Operand and Reserved) from the pull-down menu. Similarly for [Field Attr], click pull-down menu to select a value depending on the option selected in [Field Type]. The following table shows different attributes.

Field Type	[Field Attr]	Description
Opcode	Binary	The value is fixed binary pattern for all instructions of the same instruction type.
	Name	The value is variable and each instruction of the same instruction type must have a different value.
Operand	Name	The value is fixed name for all instructions of the same instruction type.

For example, based on {small_RISC} design specification, R_R (Register-Register) Type instruction would be defined as follows:

31	26	25	21	20	16	15	11	10	0
000000		rs0		Rs1		rd		func.	
Op-code		Source Register 0		Source Register 1		Destination Register		Instruction specific Op-code	

Based on the above instruction type definition, instruction type {R_R} should be defined as shown in Figure 49.

Figure 49 R_R Type Setting

Next, click [OK] button and return to [Instruction Definition] window. {R_R} type would be defined as shown in Figure 50.

Figure 50 [Instruction Definition] Window (R_R Type)

Similarly, instruction types {R_I, B, JP_T} can be defined, as shown in Figure 51.

<Tip> Changing an Instruction Type

After defining an instruction type, it would be possible to change the type definition. Double click on Instruction Type Name, to open [Instruction Type Definition], where you can change the instruction type.

<Tip> Deleting an Instruction Type

When deleting an instruction type defined previously, it is necessary to mark the instruction type. Click the name of the instruction type. After that, select [Mark] from the [Edit] menu or click the mouse middle button at the name's character string, (click right and left mouse's buttons at the same time, if there is no middle button). By doing it, the background color of the name's character string becomes blue. At this condition, the marked instruction type can be deleted by selecting [TypeDelete] from the [Edit] menu.

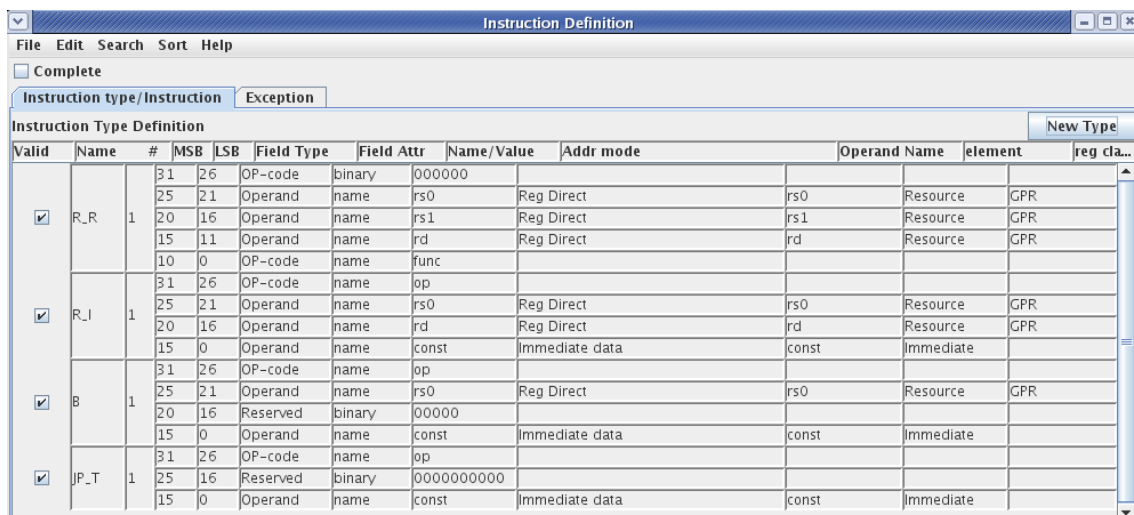


Figure 51 [Instruction Definition] Window (All Types)

<Note> The deletion cannot be performed with [Valid] checked.

4.6.3. Instruction Definition

After defining all instruction types, the next step is to define each instruction based on its type. Based on small_RISC design specification, the following instructions are defined.

ADD (Addition)	R_R
SUB (Subtraction)	R_R
LOAD (load from memory)	R_I

STORE (store in memory)	R_I
J (un-conditional branch)	JP_T
BEZ (conditional branch)	B

First, define the add instruction {ADD} by using {R_R} instruction type. Click [New instruction] button in [Instruction Definition] window to open [Input] window (Figure 53).



Figure 52 New Instruction Button

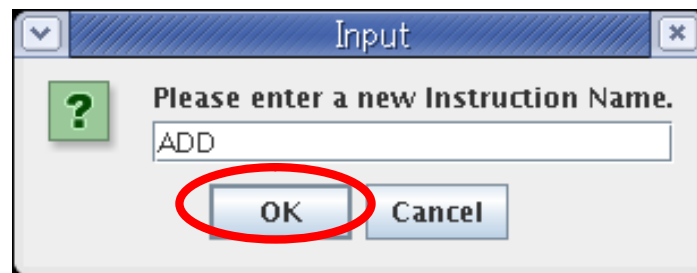


Figure 53 Registration of Instruction Name

Enter {ADD} to the field and click [OK] button and [Instruction Window] opens (Figure 54).

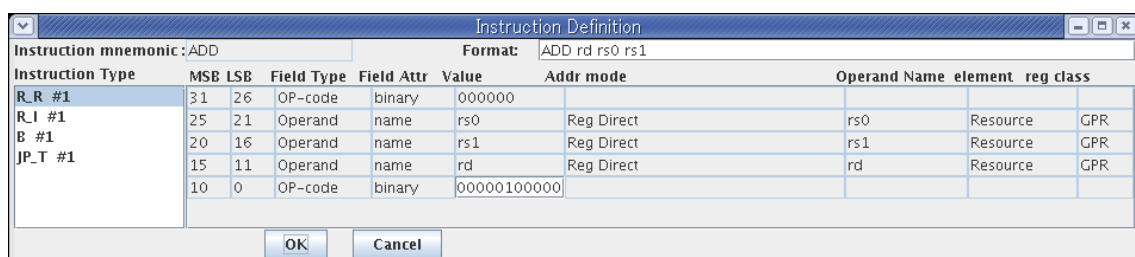


Figure 54 New Instruction Setting

From [Instruction Type] list in the left side of the window, select the instruction type {R_R}. This is used for the add instruction {ADD}. Only the first and fifth fields are enabled for entering the value.

This is a unique field for instructions of {R_R} instruction type. After entering the

binary pattern {00000100000} to the fifth field, and the [Format] {ADD rd rs0 rs1}, click [OK] button to end the {ADD} instruction definition and return to [Instruction Definition] window. The window shows {ADD} instruction definition. Likewise, define other instructions as shown in Figure 55.

Instruction Field Definition									
Valid	Name	Type	#	Field				Format	
<input checked="" type="checkbox"/>	ADD	R_R	1	0 0 0 0 0 0	r s 0	r s 1	r d	0 0 0 0 0 1 0 0 0 0 0	ADD rd rs0 rs1
<input checked="" type="checkbox"/>	SUB	R_R	1	0 0 0 0 0 0	r s 0	r s 1	r d	0 0 0 0 0 1 0 0 0 1 0	SUB rd rs0 rs1
<input checked="" type="checkbox"/>	LOAD	R_I	1	1 0 0 0 0 0	r s 0	r d	c o n s t		LOAD rd rs0 const
<input checked="" type="checkbox"/>	STORE	R_I	1	1 0 1 0 0 1	r s 0	r d	c o n s t		STORE rd rs0 const
<input checked="" type="checkbox"/>	BEZ	B	1	0 0 0 1 0 0	r s 0	0 0 0 0 0	c o n s t		BEZ rs0 const
<input checked="" type="checkbox"/>	J	JP_T	1	0 0 0 0 1 0	0 0 0 0 0 0 0 0 0 0	c o n s t			J const

Figure 55 small_RISC Instructions

<Tip> Changing an Instruction

After defining an instruction, to change the definition, double-click on the instruction name, to open [Instruction definition] window, where you can change the definition.

<Tip> Deleting an Instruction

When deleting an instruction type defined previously, it is necessary to mark the instruction type. Click the name of the instruction type. After that, select [Mark] from the [Edit] menu or click the mouse middle button at the name's character string, (click right and left mouse buttons at the same time, if there is no middle button). By doing it, the background color of the name's character string becomes blue. At this condition, the marked instruction type can be deleted by selecting [InstDelete] from the [Edit] menu.

<Note> The deletion cannot be performed with [Valid] checked.

4.6.4. Interrupt/Exception Setting

Now define the reset interrupt operation. If you click the [Exception] tab, [Interrupt/Exception Definition] panel appears as shown in Figure 56.

The screenshot shows the 'Instruction Definition' window with the 'Exception' tab selected. The window contains a table with columns: Valid, Interrupt Name, and Properties. The first row is for a dummy interrupt named '\$\$\$dummy\$\$\$'. The 'Type' is set to 'External'. The 'Condition' section has a 'Valid' checkbox and a 'Port Name' dropdown set to 'Unselected'. The 'Mask' section has a 'Maskable' radio button set to 'Yes', a 'Register Name' dropdown set to 'Unselected', and empty fields for 'Position' and 'Register Value'. A 'New Interrupt' button is in the top right.

Figure 56 Interrupt/Exception Definition Panel

To define a new interrupt/exception, click the [New Interrupt] button. This will create a new row for the new interrupt/exception. The following table describes the fields and their meanings.

Valid	Click to enable/disable the interrupt/exception.
Interrupt Name	Name of interrupt/exception.
Type	From pull-down menu, select Interrupt/Exception Type. You can select [External] or [Internal]. External: External signal causes the condition. Internal: Internal signal causes the condition.
Condition	Logical expression to cause the interrupt/exception.
Mask	Mask setting

Here change {dummy}, to {reset} in [Interrupt Name] field and change the interrupt type to {Reset}.

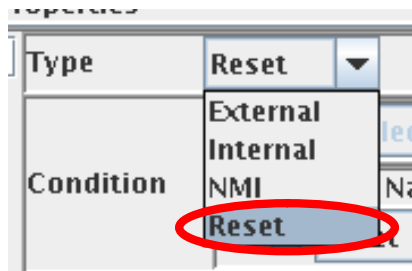


Figure 57 Interrupt/Exception Type Setting

Next, specify the signal line of the reset signal with [Condition], and then set a value for the signal. The reset becomes effective when the value of the signal is reached. Reset signal is made to work when {Reset} is set to [Port Name] as a reset signal according to the specification, and when the value of this signal becomes {1}.

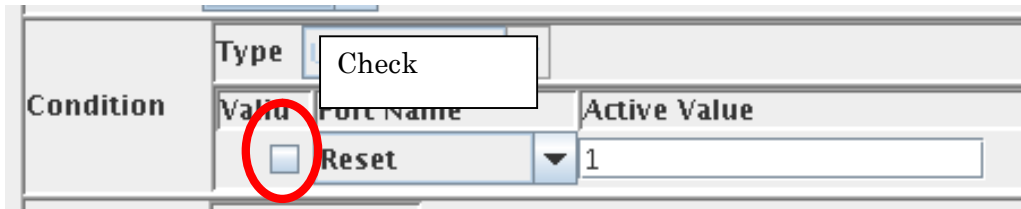


Figure 58 Reset Signal Setting

After setting all information on the reset interrupt, as shown in Figure 59. Check the [Valid] check box to activate this reset interrupt.

<Tip> Deleting an Interrupt/Exception

To delete a defined Interrupt/Exception, you need to mark it first. Click the name of the Interrupt/Exception. After that, select [Mark] from the [Edit] menu or click the mouse middle button at the name's character string, (click right and left mouse buttons at the same time, if there is no middle button). By doing it, the background color of the name's character string becomes blue. At this condition, the marked instruction type can be deleted by selecting [ExptDelete] from the [Edit] menu.

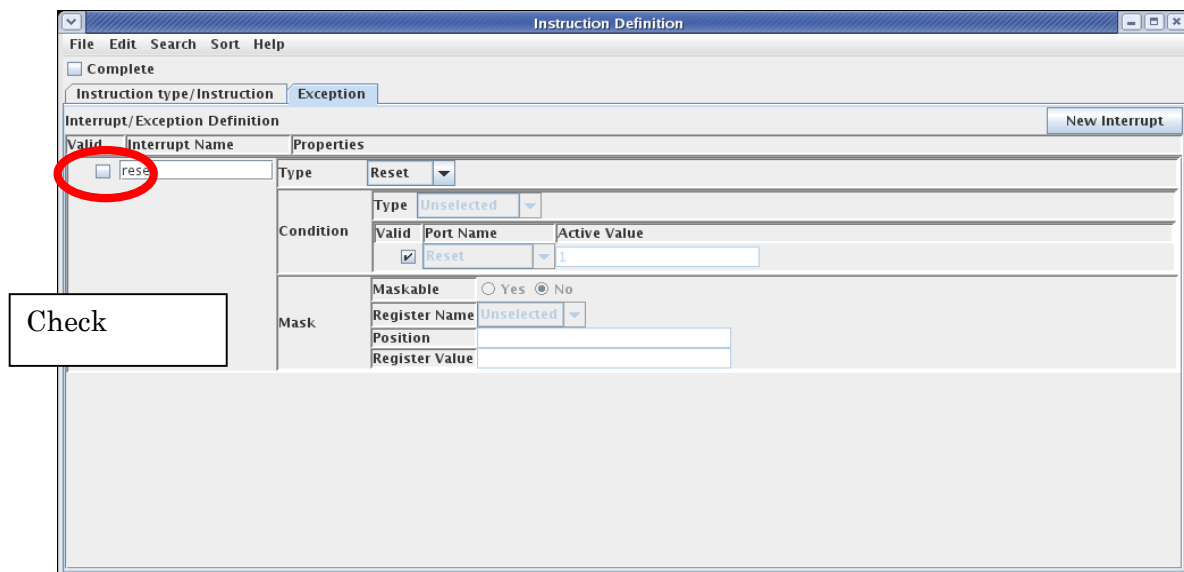


Figure 59 Interrupt/Exception Validity

By this we come to the end of defining {small_RISC} instruction set, reset, and interrupt settings. To confirm the design, check [complete] check box, then select [File] → [Close] to close the window.

4.7. Arch. Level Estimation

This section explains how to estimate the design quality of the target processor using the currently set information. ASIP Meister estimates design qualities (area, delay, and power consumption). Since this is the estimation before finalizing the design details, the estimated values are not exact values, but these values can help you in the following design steps. If the estimation result shows lower value than your expectation, you can change your current settings and re-estimate the processor core design quality.

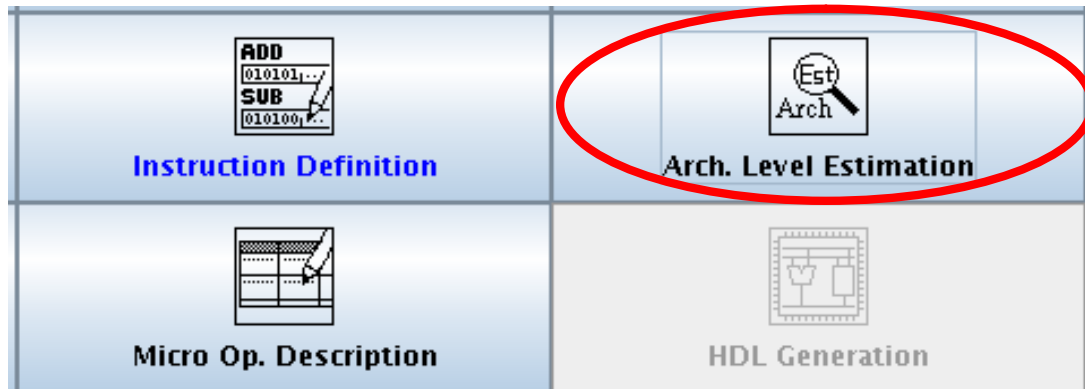


Figure 60 Arch. Level Estimation Button

Click [Arch. Level Estimation] in the main window to open [Estimation Confirm] window.

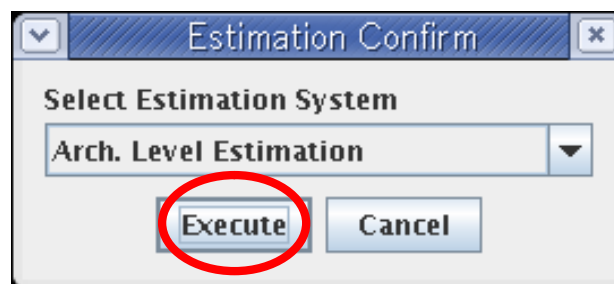


Figure 61 Estimation Execution

Click [Execute] button in the [Estimation Confirm] window. The estimation engine will estimate design qualities and show the results of the estimation in the [Arch. Level Estimation] window as shown in Figure 62.

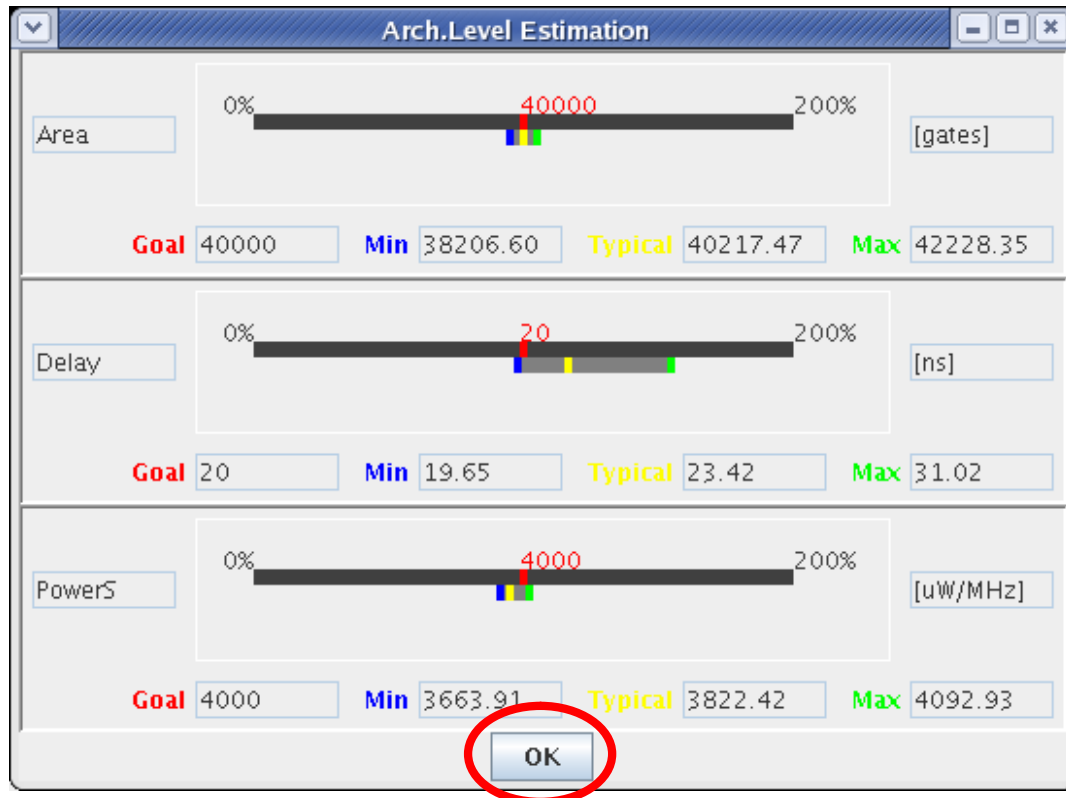


Figure 62 Estimation Execution Results

Confirm the displayed results and click [OK] button and select [File] → [Close] to return to the main window.

4.8. Assembler Generation

In this stage, the designed processor assembler would be generated. ASIP Meister has the ability to generate a meta-assembler based on the processor design description.

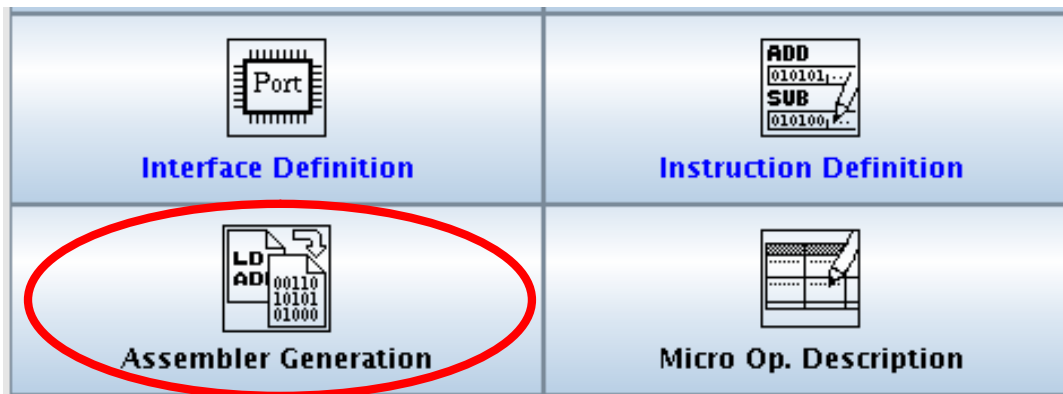


Figure 63 Assembler Generation Button

Click [Assembler Generation] button in main window to open [Generation Confirm] window.

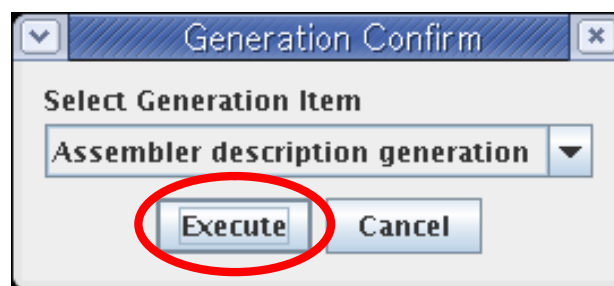


Figure 64 Assembler Description Generation

By clicking on [Execute] button in above window, the assembler description files would be generated. A successful generation should show the window in Figure 65.

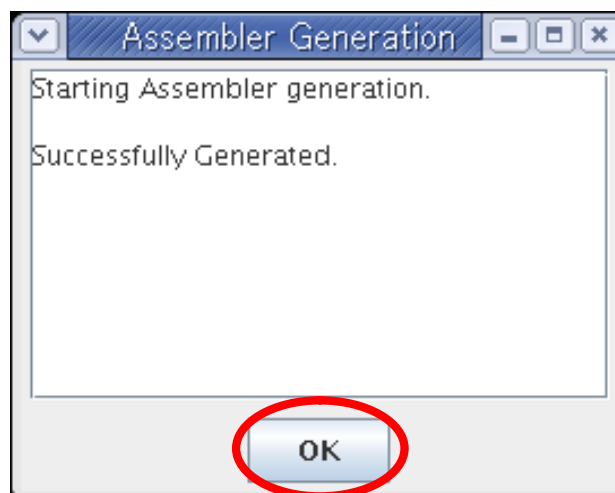


Figure 65 Assembler Description Generation Results

Click [OK] button, in case of successful generation. In case of error, return back to stage 1 and revise your processor design description, stage by stage.

4.8.1. Assembler File Generation

Within ASIP Meister “mesiter” directory, confirm the generation of the assembler file “small_RISC.des” in case of successful assembler generation.

```
small_RISC.arc  
small_RISC.des  
small_RISC.mod  
small_RISC.sw/  
small_RISC.tr  
small_RISC_arc.log  
small_RISC_asm.log
```

Figure 66 Generated Assembler Description File

4.9. Micro Operation Description

In this design stage, the micro-operation of each instruction is described. In this tutorial, you will define the micro operation description of 6 instructions (ADD, SUB, LOAD, STORE, BEZ, J) and reset interrupt.

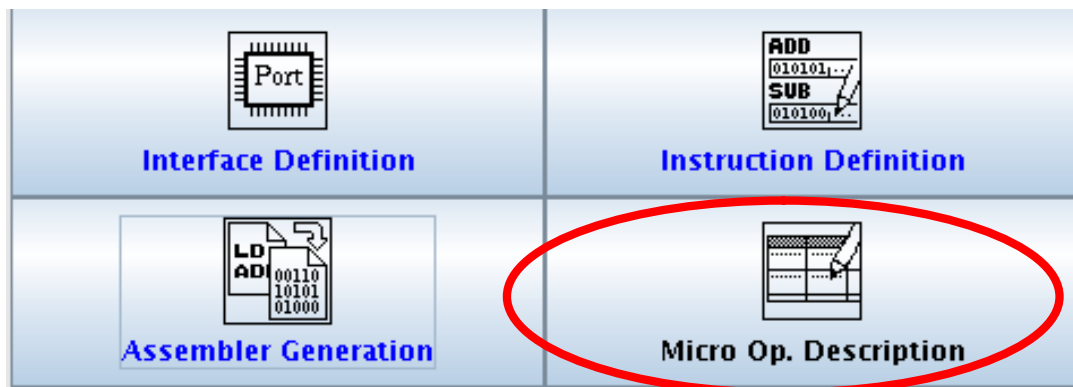


Figure 67 Micro Op. Description Button

Click the [Micro Op. Description] button in the main window to open [Micro Op. Description] window, as shown in Figure 68.

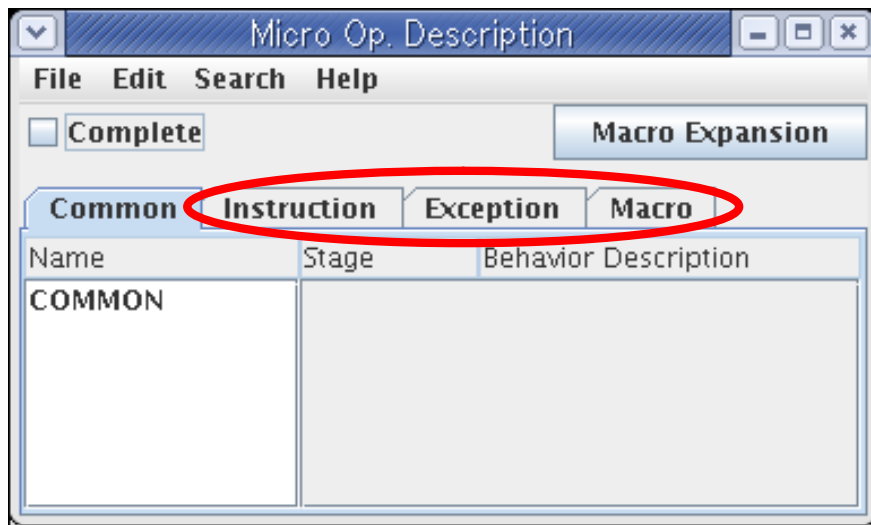


Figure 68 Micro Op. Description Window

In this window, click the tabs to select the sub windows;

Instruction	Operation description for each instruction.
Exception	Operation description for interrupt/exception process.
Macro	Macro description.

In this tutorial, you will follow the order of [Macro], [Instruction], and [Exception].

4.9.1. Macro definition: [Macro]

First, define {FETCH} macro. Click on [Micro] tab to open [Micro Op. Description].

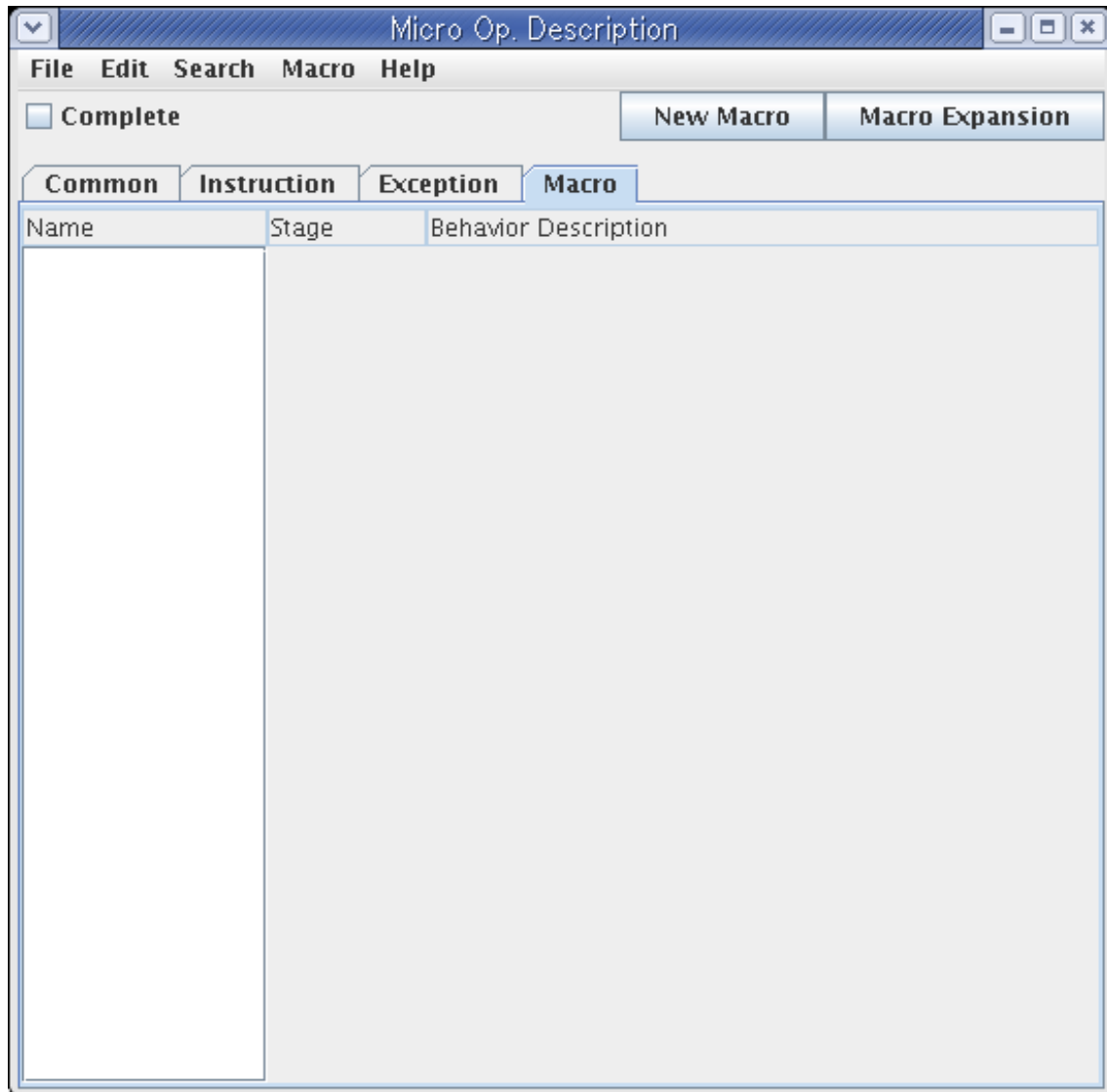


Figure 69 Macro Window

Click [Macro] → [New Macro] to open [New Macro Confirm] window (Figure 70).

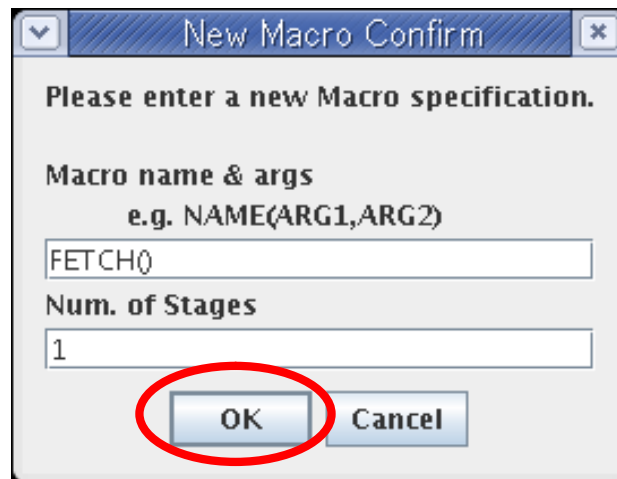


Figure 70 New Macro Confirm

Enter {FETCH0} to [Macro name & args] field and {1} to [Num. of Stages] field. Click [OK] button and the macro would be listed in [Micro Op Description] window. Next, enter the description of the macro. Figure 71 shows the description of FETCH macro.

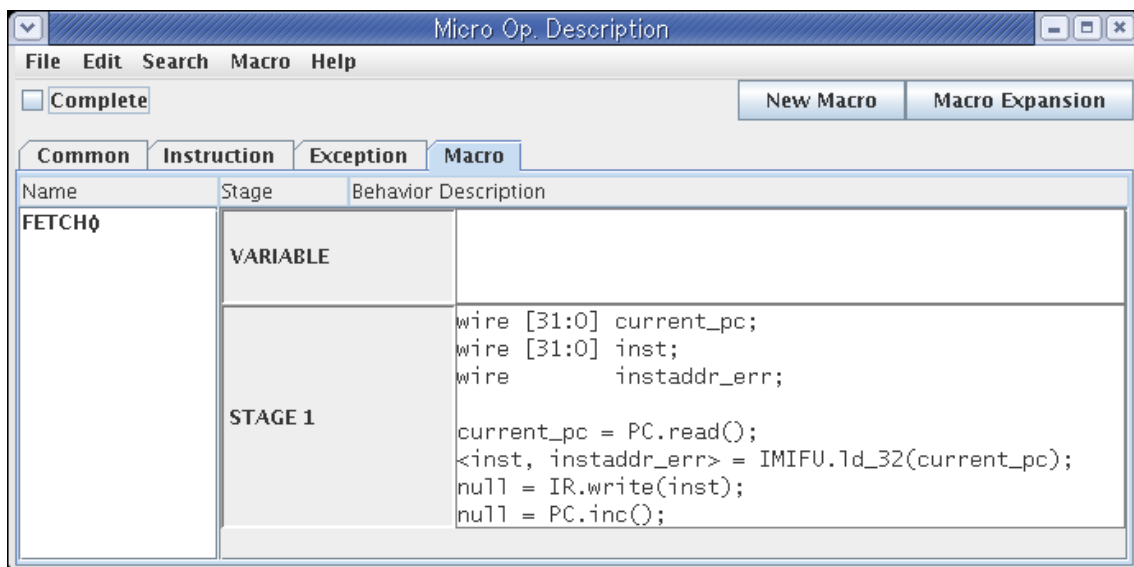


Figure 71 FETCH Macro Input

4.9.2. Instruction Operation Description: [Instruction]

To describe each instruction operation, click [Instruction]. The left side frame of [Micro Op. Description] window shows a list of defined instructions. [Stage] column shows the variable declaration (VARIABLE) and the stage names defined in [Design

Goal & Arch. Design] window. Write the description for each corresponding pipeline stage in [Behavior Description] field.

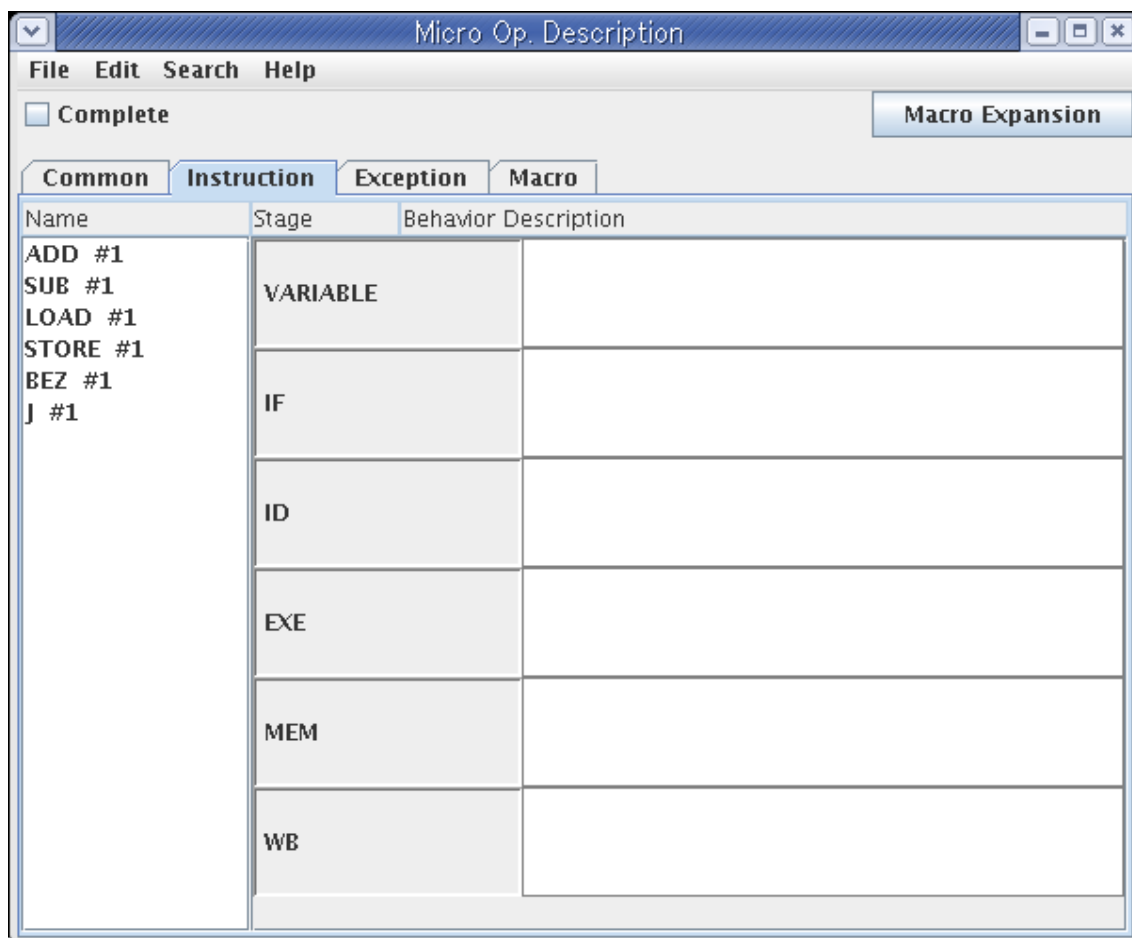


Figure 72 Instruction Micro Operations Description Window

Followings are the operation description of {ADD, SUB, LOAD, STORE, J, and BEZ}.

4.9.2.1. ADD Instruction

In the second stage of ADD instruction, two operands are read from source registers. In third stage, ALU addition operation takes place. In the fifth stage, the result is written back to destination register. This instruction behavior is described by the micro-operations description shown in Figure 73.

Micro Op. Description		
File Edit Search Help		
<input checked="" type="checkbox"/> Complete		Macro Expansion
Common	Instruction	Exception Macro
Name	Stage	Behavior Description
ADD #1	VARIABLE	wire [31:0] source0; wire [31:0] source1; wire [31:0] result;
SUB #1	IF	FETCH()
LOAD #1	ID	source0 = GPR.read0(rs0); source1 = GPR.read1(rs1);
STORE #1	EXE	wire [3:0] flag; <result, flag> = ALU.add(source0, source1);
BEZ #1	MEM	
J #1	WB	null = GPR.write0(rd, result);

Figure 73 ADD Instruction Micro Op.

4.9.2.2. SUB Instruction

SUB Instruction operation is similar to ADD. This instruction behavior is described by the micro-operations description shown in Figure 74.

Micro Op. Description		
File Edit Search Help		
<input checked="" type="checkbox"/> Complete		Macro Expansion
Common Instruction Exception Macro		
Name	Stage	Behavior Description
ADD #1	VARIABLE	wire [31:0] source0;
SUB #1		wire [31:0] source1;
LOAD #1		wire [31:0] result;
STORE #1		
BEZ #1		
J #1	IF	FETCH()
	ID	source0 = GPR.read0(rs0); source1 = GPR.read1(rs1);
	EXE	wire [3:0] flag; <result, flag> = ALU.sub(source0, source1);
	MEM	
	WB	null = GPR.write0(rd, result);

Figure 74 SUB Instruction Micro Op.

4.9.2.3. LOAD Instruction

In the second stage of LOAD instruction, the memory base address and index are extracted. In third stage, the effective address is calculated by adding the base to index. In fourth stage, memory is accessed to read data from memory referenced by the effective address. In the fifth stage the read data is written back to a destination register. This instruction behavior is described by the micro-operations description shown in Figure 75.

Micro Op. Description		
File Edit Search Help		
<input checked="" type="checkbox"/> Complete		Macro Expansion
Common	Instruction	Exception Macro
Name	Stage	Behavior Description
ADD #1	VARIABLE	wire [31:0] source0;
SUB #1		wire [31:0] source1;
LOAD #1		wire [31:0] result;
STORE #1		wire [31:0] addr;
BEZ #1		FETCH()
J #1		
	IF	
	ID	source0 = GPR.read0(rs0); source1 = EXT.sign(const);
	EXE	wire [3:0] flag; <addr, flag> = ALU.add(source0, source1);
	MEM	wire err; <result, err> = DMIFU.ld_32(addr);
	WB	null = GPR.write0(rd, result);

Figure 75 LOAD Instruction Micro Op.

4.9.2.4. STORE Instruction

In the second stage of STORE instruction, the data, memory base address and index are extracted. In third stage, the effective address is calculated by adding the base to index. In fourth stage, memory is accessed to write data into memory referenced by the effective address. This instruction behavior is described by the micro-operations description shown in Figure 76.

Micro Op. Description		
File Edit Search Help		
<input checked="" type="checkbox"/> Complete		Macro Expansion
Common Instruction Exception Macro		
Name	Stage	Behavior Description
ADD #1	VARIABLE	wire [31:0] data;
SUB #1		wire [31:0] base;
LOAD #1		wire [31:0] offset;
STORE #1		wire [31:0] addr;
BEZ #1		
J #1	IF	FETCH()
	ID	data = GPR.read0(rd); base = GPR.read1(rs0); offset = EXT.sign(const);
	EXE	wire [3:0] flag; <addr, flag> = ALU.add(base, offset);
	MEM	wire err; err = DMIFU.s_32(addr, data);
	WB	

Figure 76 STORE Instruction Micro Op.

4.9.2.5. BEZ Instruction

In the second stage of BEZ instruction, the condition, PC and index values are extracted. In third stage, the condition is checked against 0, and the effective branch address is calculated by adding the PC to index value. If condition is true, the new PC value is loaded with the branch address. This instruction's behavior is described by the micro-operations description shown in Figure 77.

Micro Op. Description		
File Edit Search Help		
<input checked="" type="checkbox"/> Complete		Macro Expansion
Common	Instruction	Exception Macro
Name	Stage	Behavior Description
ADD #1 SUB #1 LOAD #1 STORE #1 BEZ #1 J #1	VARIABLE	wire [31:0] source0; wire [31:0] temp_pc; wire [31:0] offset;
	IF	FETCH()
	ID	source0 = GPR.read0(rs0); offset = EXT.sign(const); temp_pc = PC.read();
	EXE	wire [31:0] target; wire [3:0] flag; wire cond; cond = source0 == "00000000000000000000000000000000"; <target, flag> = ALU.add(temp_pc, offset); null = [cond] PC.write(target);
	MEM	
	WB	

Figure 77 BEZ Instruction Micro Op.

4.9.2.6. J Instruction (unconditional branch)

In the second stage of J instruction, the PC and offset values are extracted. In third stage, the effective branch address is calculated by adding the PC to offset value and the result is loaded into PC to cause unconditional branching. This instruction behavior is described by the micro-operations description shown in Figure 78.

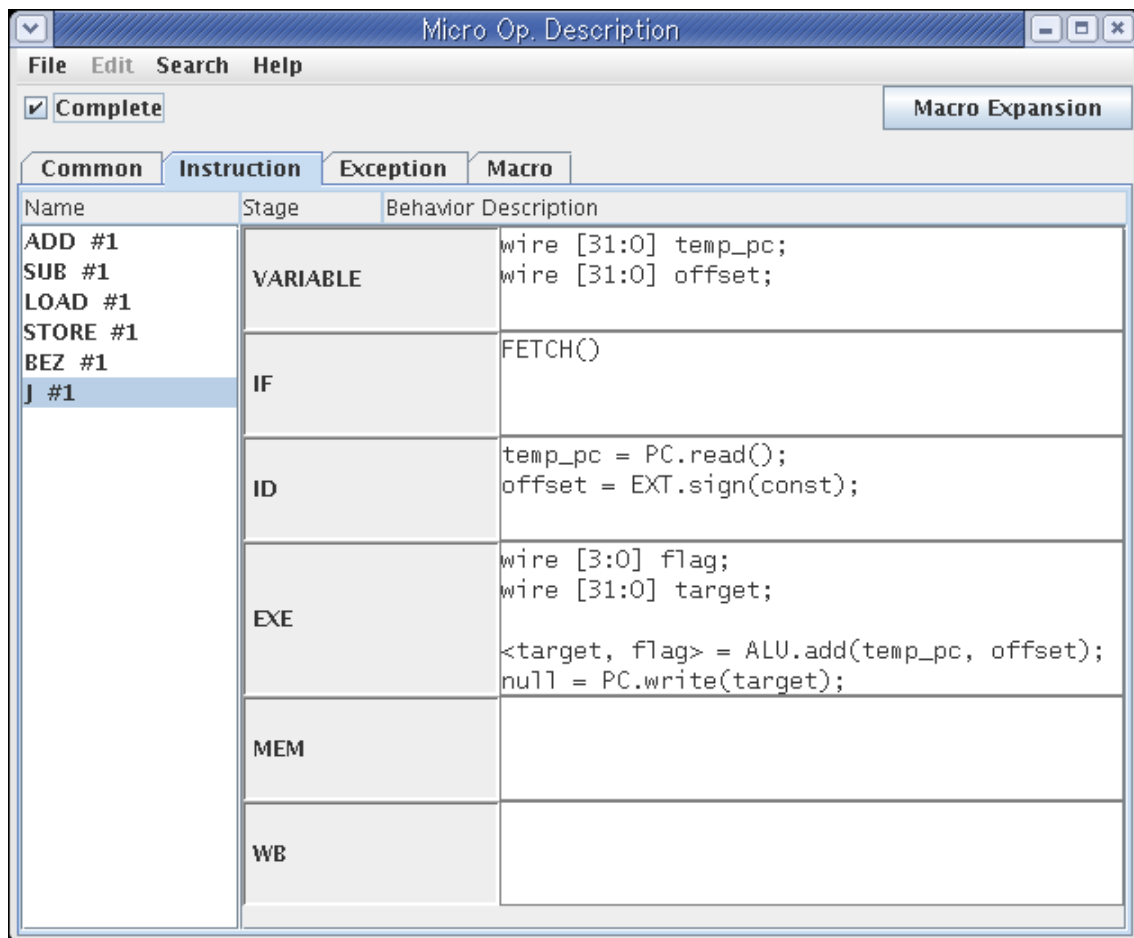


Figure 78 J Instruction Micro Op.

4.9.3. Interrupt/Exception Description: [Exception]

Next, you will describe the operation of {reset} exception defined in [Instruction Definition] window. Click [Exception] tab to define interrupt/exception processes. Figure 79 shows the description of {reset} interrupt.

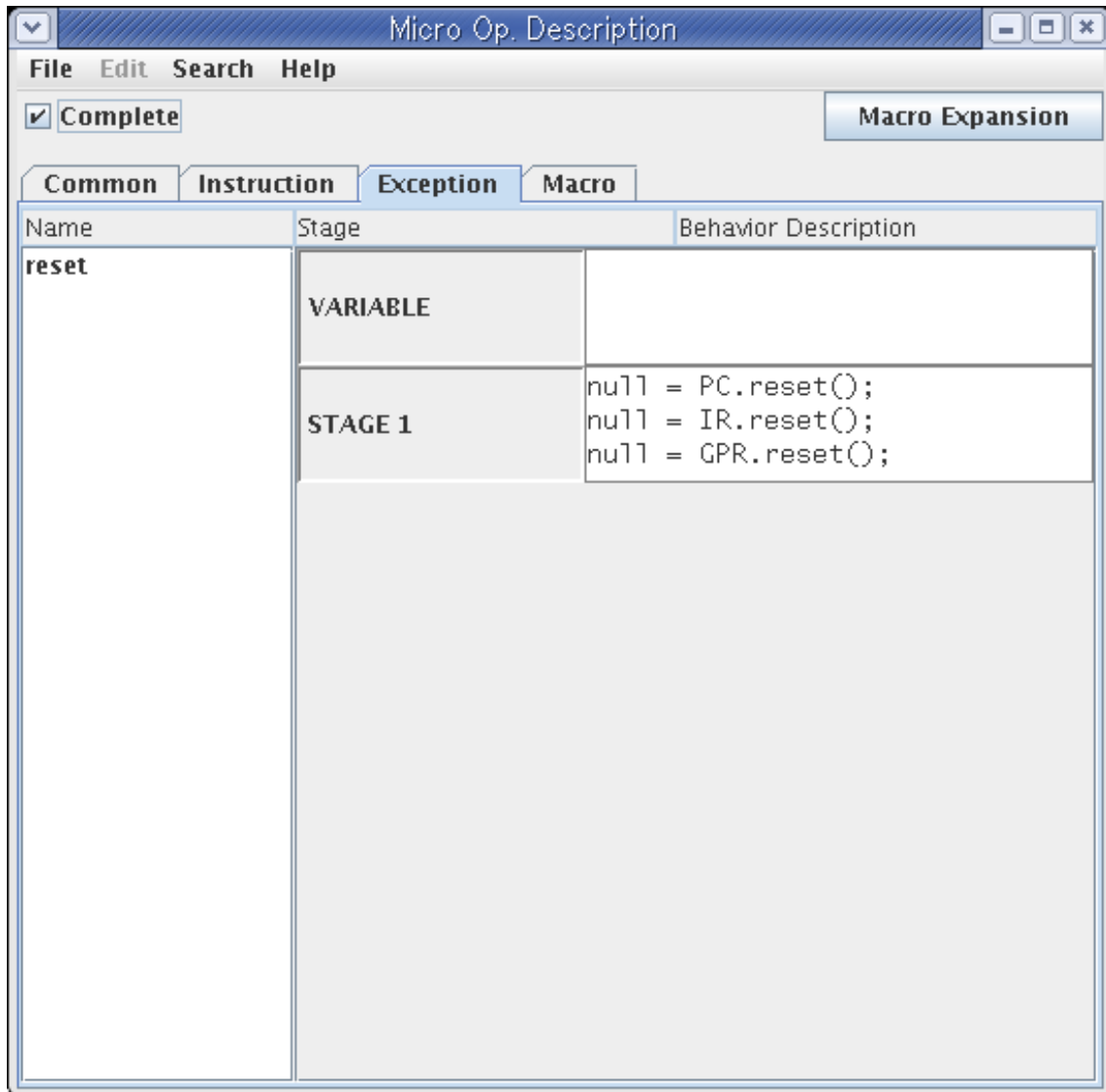


Figure 79 Reset Interrupt Description

After you have written all instruction micro operation descriptions, check [Complete] check box and go to [File] → [Close] to return to the main window.

You have entered all the data necessary for HDL generation described in the following section.

4.10. HDL Generation

This is the last step in processor design phases. In this step, the HDL description files for the simulation and logical synthesis of the designed ASIP core are generated. ASIP Meister uses the abstraction level that has been set earlier with the [Description Style

of HDL] radio button of [Resource Declaration] window.

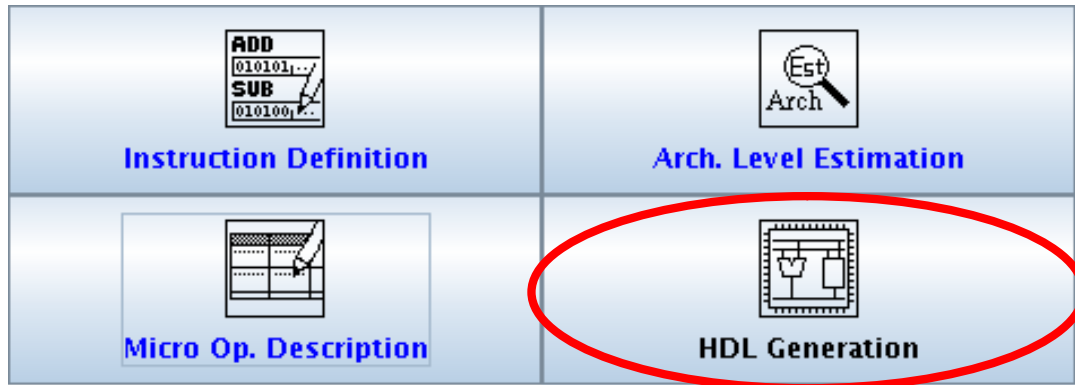


Figure 80 HDL Generation Button

Click [HDL Generation] button in the main window to open [Generation Confirm] window.

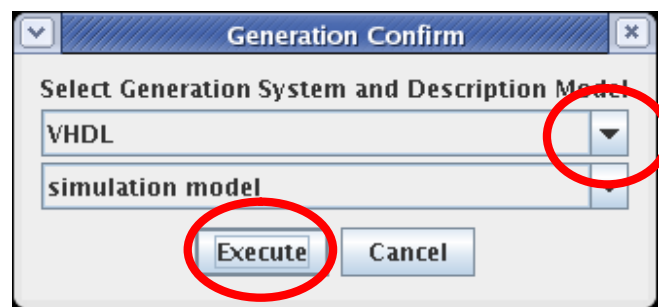


Figure 81 [Generation Confirm] Window

In this window, you can select generation language and generation level. By selecting the appropriate option from the pull-down menu, you can select VHDL or Verilog as a language and you can select to generate the simulation model, the synthesizable model, or both.

Table 1 Selectable Generating Language

VHDL	VHDL Generation
Verilog	Verilog Generation

Table 2 Selectable Generation Level

simulation model	Simulation Model Generation
synthesizable model	Synthesis Model Generation
sim. model and syn. model	Generation of both models

Select [sim. Model and syn. Model] to generate both models and click [Execute] button to start the HDL generation engine.

When the generation engine completes HDL generation, the following message window shown in Figure 82 will appear. Confirm that no error is displayed in the window and click [OK] button to return to the main window.

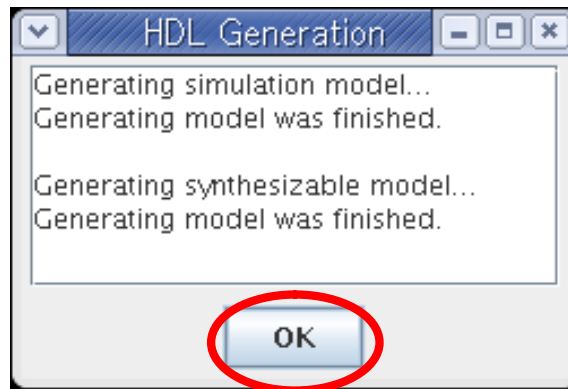


Figure 82 HDL Generation Execution Results

4.10.1. Confirming Generated HDL

Now the current directory has a new directory “meister”. The “meister” directory has two new directories for the simulation model and logical synthesizable model. Generated HDL Files are saved under each directory. For small_RISC design, confirm the generation of the following directories;

```
meister/lang/small_RISC.sim
meister/lang/small_RISC.syn
```

Ensure that “lang” is either vhdl or verilog.



```

[asipuser@asipdemo:~/small_RISC.VHDL.sim]
ファイル(E) 編集(E) 表示(V) 端末(T) タブ(T) ヘルプ(H)
[asipuser@asipdemo small_RISC.VHDL.sim]$ ls
fhm_alu_w32.vhd          rtg_controller.vhd      rtg_register_w1_01.vhd
fhm_extender_w16.vhd    rtg_mux2to1_w32.vhd    rtg_register_w32.vhd
fhm_mifu_w32_00.vhd     rtg_mux2to1_w5.vhd     rtg_register_w5.vhd
fhm_mifu_w32_01.vhd     rtg_mux3to1_w32.vhd    rtg_register_w7.vhd
fhm_pcu_w32.vhd         rtg_proc_fsm.vhd       small_RISC.vhd
fhm_register_w32.vhd    rtg_register_w17.vhd
fhm_registerfile_w32.vhd rtg_register_w1_00.vhd
[asipuser@asipdemo small_RISC.VHDL.sim]$
  
```

Figure 83 HDL Files Used for Simulation



```

[asipuser@asipdemo:~/small_RISC.VHDL.syn]
ファイル(E) 編集(E) 表示(V) 端末(T) タブ(T) ヘルプ(H)
[asipuser@asipdemo small_RISC.VHDL.syn]$ ls
fhm_alu_w32.vhd          rtg_controller.vhd      rtg_register_w1_01.vhd
fhm_extender_w16.vhd    rtg_mux2to1_w32.vhd    rtg_register_w32.vhd
fhm_mifu_w32_00.vhd     rtg_mux2to1_w5.vhd     rtg_register_w5.vhd
fhm_mifu_w32_01.vhd     rtg_mux3to1_w32.vhd    rtg_register_w7.vhd
fhm_pcu_w32.vhd         rtg_proc_fsm.vhd       small_RISC.vhd
fhm_register_w32.vhd    rtg_register_w17.vhd
fhm_registerfile_w32.vhd rtg_register_w1_00.vhd
[asipuser@asipdemo small_RISC.VHDL.syn]$
  
```

Figure 84 HDL Files Used for Synthesis

You have now come to the end of the tutorial on HDL generation of small_RISC.

Now you can proceed to the tutorial on [C Definition] and [Compiler Generation], using a processor extending a base processor ***Brownie***. ***Brownie*** can be used exclusively in the ASIP Meister Standard environment.

4.11. C Definition (ASIP Meister Standard Environment Required)

In this stage, you can define how to represent the extension instruction added to the base processor *Brownie* in C code. Extension instruction definition enables a compiler to output assembly code complying with the extension instruction.

<Note> Any instruction of *Brownie* must not be deleted.

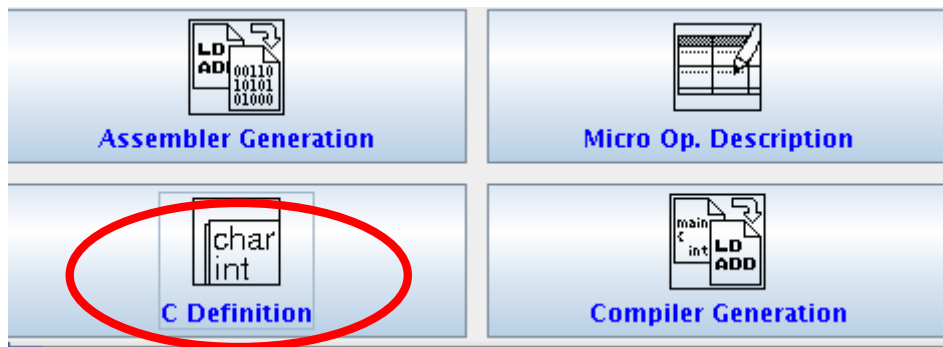


Figure 85 C Definition Button

A design file needs to be read and additional instructions need to be designed on Brownie according to the procedure explained above.

Click [C Definition] from the main menu to open the window shown in Figure **.

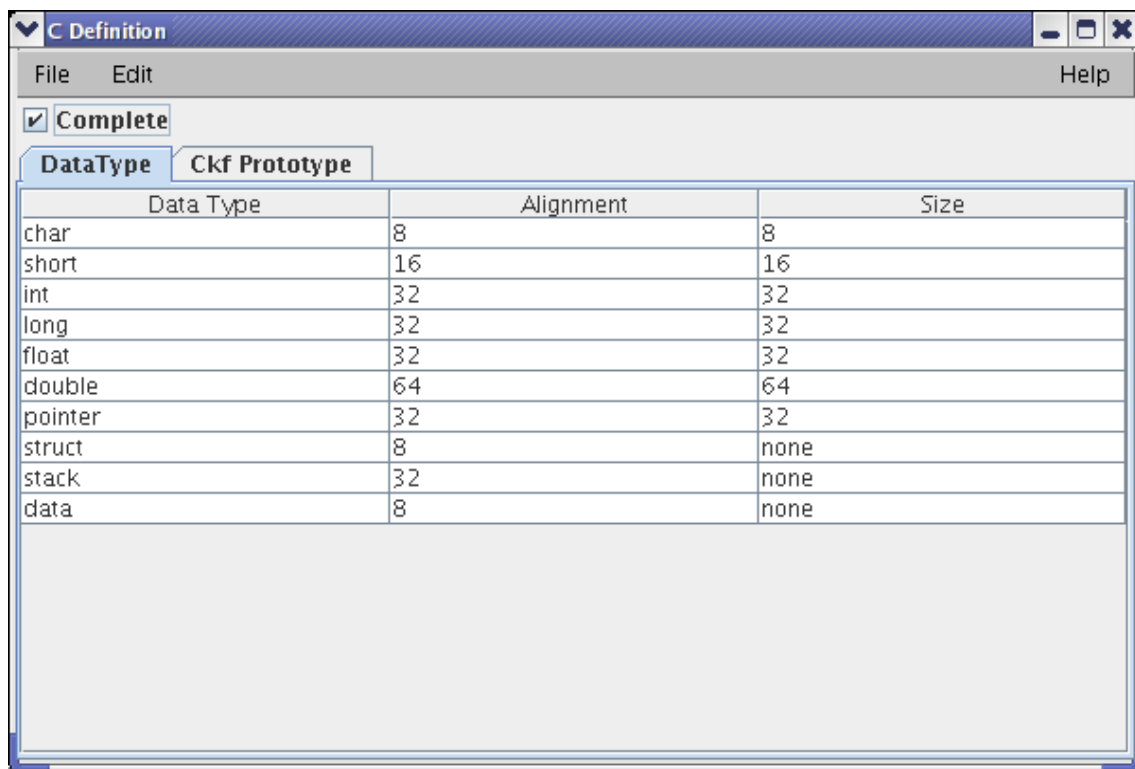


Figure 86 [C Definition] Window

[C Definition] consists of the following two tabs.

1. DataType tab: Definition of data type of C code.
2. Ckf Prototype tab: Definition of extension instruction.

The current version allows no alteration of the content of DataType tab. Select Ckf Prototype tab. The window shown in Figure ** will appear.

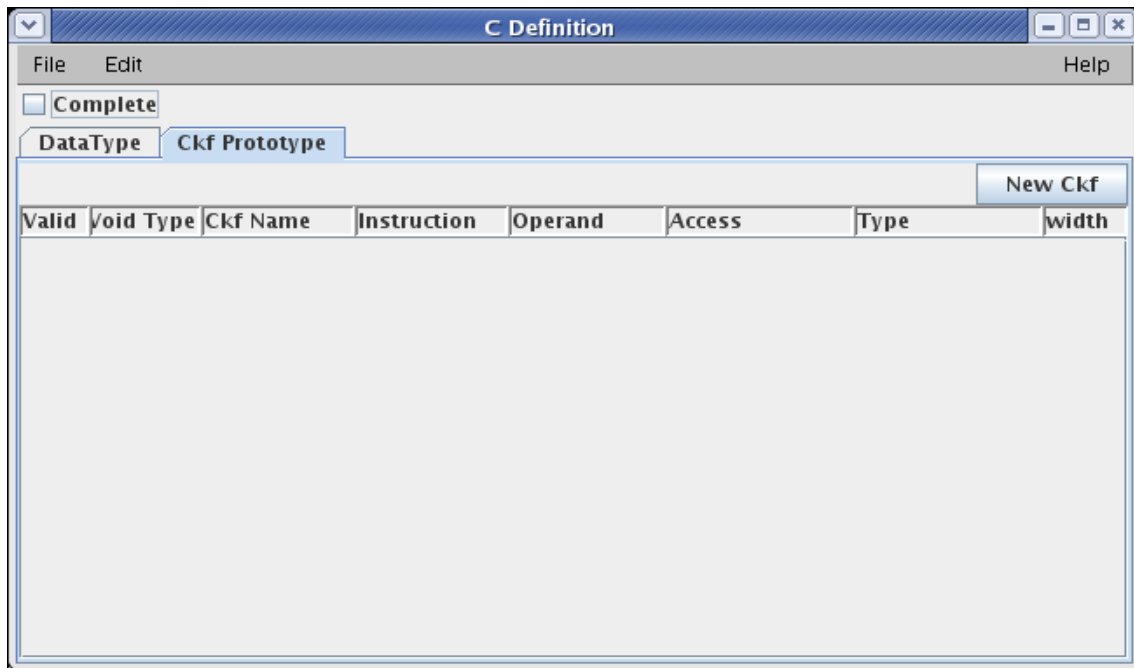


Figure 87 [Ckf Prototype] Tab

In the initial state, Ckf is not registered. You can select [New Ckf] to register Ckf. In this tutorial, an "NXOR" instruction which Brownie does not have will be registered as an added instruction. Instruction types of "NXOR" instruction and micro operation description need to be defined in advance. An "NXOR" instruction reads values from two registers and writes the result of NXOR (not exclusive-or) operation into one register. Select [New Ckf] to open [New CKF] window shown in Figure **.



Figure 88 [New CKF] Window

Set the Ckf name of an "NXOR" instruction as "NXOR." When OK button is clicked, [Ckf Prototype] window will look as shown in Figure **.

An "NXOR" instruction defines each item respectively as shown in Figure **.

Set "Access" to "Read" as for input operands rs1 and rs2; set "Access" to "Write" as for an output operand rd. Set the "Width" representing bit width to 32, in accordance with the register length of Brownie32. Since the instruction added this time has one output operand and more than one input operand, leave "Void Type" unchecked. By defining this way, you can generate a compiler which can generate assembly code complying with the description represented below in C code.

<C code>

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

<Note> You need to put "__builtin_brownie32_" before the extension instruction

<Assembly code after the conversion>

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

You must check "Void Type" in the following cases. In these cases, an output operand is handled as a call-by-reference argument.

- In case of no output operand

Instruction implicitly referring to stack pointer and PUSH values into the stack

```
PUSH      rs                      ;; Store rs value into the stack  
void __builtin_brownie32_PUSH(int x);
```

- In case of more than one output operand

e.g. Instructions to store the quotient and the residue of a division into separate registers,

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

- In case of one output operand and no input operand

e.g. Instructions implicitly referring to stack pointer and POP values into the stack

```
POP      rd                      ;;Store the value of a stack into rd  
void __builtin_brownie32_POP(int *x);
```

e.g. Instructions to acquire a number from the hardware random number

generator and store it into the register

```
RND      rd      ;;Store a random value into rd  
void __builtin_brownie32_RND(int *x);
```

For further details on each item editable in [Ckf Prototype] window, refer to ASIP Meister User Manual.

<Tip> micro operation description of NXOR instruction

Micro Op. Description		
File Edit Search Help		
<input checked="" type="checkbox"/> Complete		Macro Expansion
Common Instruction Exception Macro		
Name	Stage	Behavior Description
ADD #1	VARIABLE	
SUB #1		
MUL #1		
DIV #1		
MOD #1		
AND #1	IF	Fetch()
OR #1		
XOR #1	ID	GPRDoubleRead(rs1, rs2)
NXOR #1		
LLS #1	EXE	ALUExec(nxor, source1, source2)
LRS #1		ForwardDataFromEXE(rd, alu_result)
ARS #1	WB	WriteBack(rd, alu_result)
ELT #1		ForwardDataFromWB(rd, alu_result)
ELTU #1		
EEQ #1		
ENEQ #1		
ADDI #1		
SUBI #1		
ANDI #1		
ORI #1		
XORI #1		
LLSI #1		
LRSI #1		
ARSI #1		
LSOI #1		
LB #1		
LH #1		
LW #1		
SB #1		
SH #1		
SW #1		
BRZ #1		
BRNZ #1		

Figure 89 Micro Op. Description of NXOR Instruction

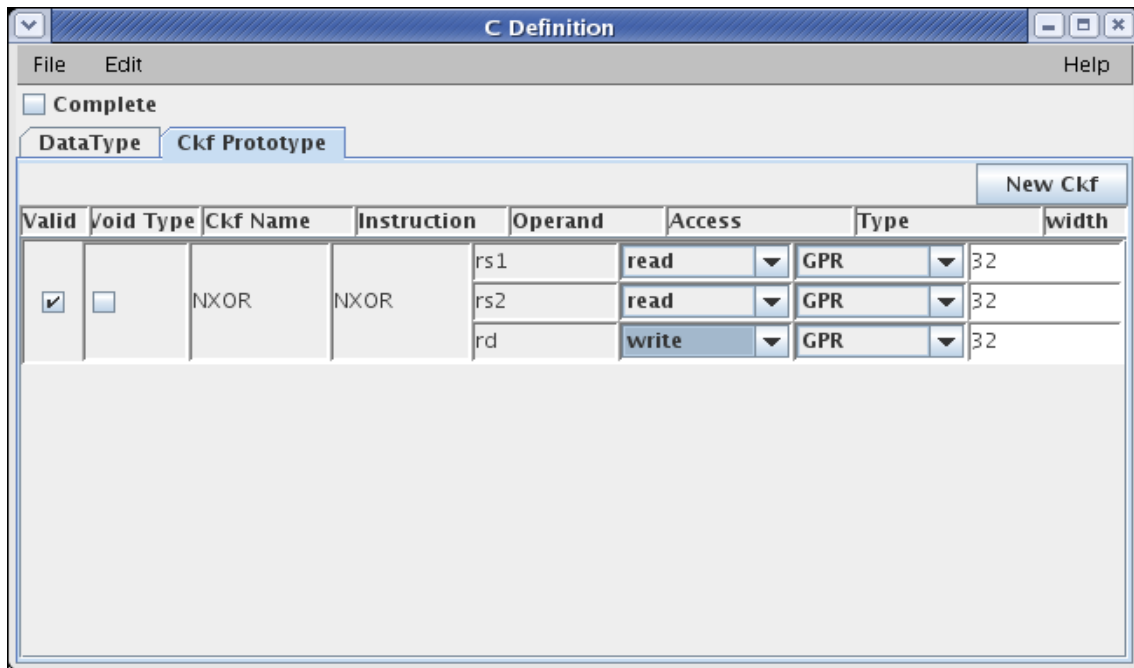


Figure 90 [Ckf Prototype] Window after the registration of NXOR instruction

You have now completed [C Definition]. Check [Complete] and close [C Definition] window.

4.12. Compiler Generation, Binutils Generation: (ASIP Meister Standard environment is required)

You can generate/create a compiler and a Binutils for Ckf defined in 4.11[C Definition].

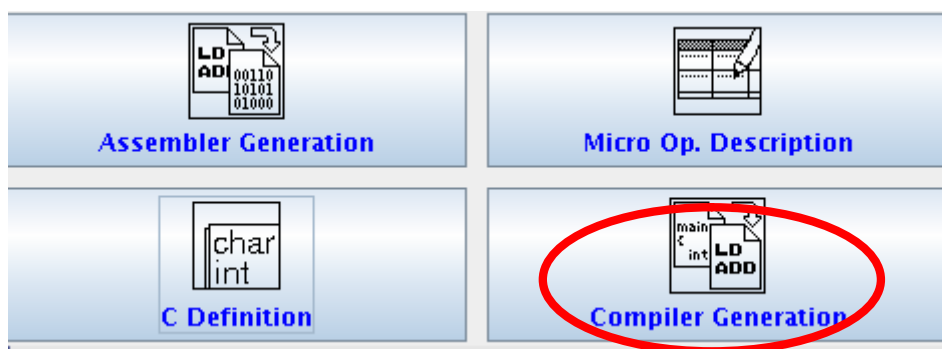


Figure 91 Compiler Generation Button

Select [Input Description Generation] → "Execute" in [Generation Confirm] window.

Then, select [GNU Tools Generation].

<Note> It can take more than ten minutes after selecting [GNU Tools Generation].

When the generation is completed successfully, compiler and Binutils files are generated in a folder designated with an environment variable *\$ASIP_GNU_INSTALL_DIR*. As design file name is "Brownie.pdb", compiler and Binutils files are generated under Brownie.swgen

You have now completed the tutorial!

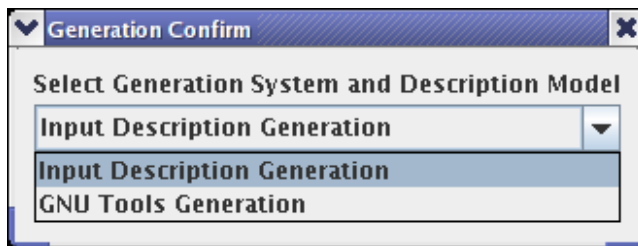


Figure 92 [Generation Confirm] Window

For further details not explained in this tutorial, refer to ASIP Meister User Manual.

small_RISC Processor Specification(Appendix)

March 2008

ASIP Solutions, Inc.

Below is a complete specification of (small_RISC) processor core used in this tutorial.

A. Design Goals

small_RISC design goals are given by the following values. Area is given highest priority among design goals.

Area	Delay	Power Consumption
45000 [gate]	20 [ns]	4000 [μ W/MHz]

B. Memory Access Method

The memory access methods that small_RISC assumes are described in the following table.

	Access Method
Instruction Memory	single-cycle
Data Memory	multi-cycle

C. Pipeline Structure

Pipeline structure consists of 5 stages as described below;

Stage Number	Stage Operation
1	Instruction Fetch
2	Instruction Decode/Register Read
3	Execute
4	Memory Access
5	Register Write Back

Delayed Branch	Yes
Delayed Branch Slot Number	1
Instruction Width	32 bit
Data Width	32 bit

D. Resource Usage

small_RISC, consists of a group of resources used by various instructions including load, store, unconditional, conditional branching and so on. Used resources include special and general purpose registers, memory access units, ALU, and extender (EXT).

Resource	Selected FHM	Resource Name	Parameter
Program Counter	pcu	PC	Increment Step width = 4 Increment Algorithm = default Use as = Prog. Counter
Instruction Register	register	IR	Bit width = 32 bit Use as = Inst. Register
Instruction Memory Interface Unit	mifu	IMIFU	Data Bit Width = 32 bit Address Space = 32 bit Access Width = 32 bit Access Mode = Single Cycle Type = Read Only Use as = Inst. Memory
Data Memory Interface Unit	mifu	DMIFU	Data Bit Width = 32 bit Address Space = 32 bit

			Access Width = 8 bit Access Mode = Multi-Cycle Type = Read/Write Use as = Data Memory
Register File	registerfile	GPR	Registers Number = 32 Number of Read Ports = 2 Number of Write Ports = 1 Use as = Register File
Arithmetic and Logic Unit	alu	ALU	Increment algorithm = default Use as = Unspecified
Extender Unit	extender	EXT	Input Data Width = 16 bit Output Data Width = 32 bit Use as = Unspecified

E. Storage Definition

This includes register file, register, and memory storage definition.

E.1. Register File/Storage Definition

Processor register file consists of a group of general purpose registers which are set as follows;

E.1.1. Register File Setting

Storage Name	GPR[asc]
Resource	GPR
Bit Width	32
Usage	register
Location	original
Binary	[bin-asc]

E.1.2. Register File (Register Usage)

GPR0	zero register
GPR28	stack pointer

GPR29	frame pointer
GPR30	link register
GPR31	return register

E.2. Register/Storage Definition

Special purpose registers include instruction register, program counter which are set as follows;

E.2.1. Instruction Register Setting

Storage Name	IR
Resource	IR
Bit width	32
Usage	instruction register
Location	original

E.2.2. Program Setting

Storage Name	PC
Resource	PC
Bit Width	32
Usage	program counter
Location	original

E.3. Memory/Storage Definition

Memory interface units include both instruction and data memory interface units.

E.3.1. Instruction Memory Interface Unit

Storage Name	IMIFU
Resource	IMIFU
Bit Width	32
Usage	instruction memory

Access Width	32
--------------	----

E.3.2. Data Memory Interface Unit

Storage Name	DMIFU
Resource	DMIFU
Bit Width	32
Usage	data memory
Access Width	8

F. Input/Output Ports

Processor core design includes clock, reset, and memory access signals, as shown in following table;

Signal Name	Direction	Bit Width	Description
CLK	in	1	Clock Signal
Reset	in	1	Reset Signal
InstAB	out	32	Instruction Memory Address Bus
InstDB	in	32	Instruction Memory Data Bus
DataAB	out	32	Data Memory Address Bus
DataDB_in	in	32	Data Memory Data Input Bus
DataDB_out	out	32	Data Memory Data Output Bus
DataReq	out	1	Data Memory Request Signal
DataAck	in	1	Data Memory Acknowledge Signal
DataRW	out	1	Data Memory Read/Write Signal
DataMode	out	2	Data Memory Write Mode Signal
DataExt	out	1	Data Memory Extension Signal
DataAddrError	in	1	Data Memory Address Error Signal
DataCancel	out	1	Data Memory Cancel Signal

G. Instruction Set List

G.1. Instruction Type

Four instruction types are defined within {small_RISC}:

1. Register-Register Type (R_R Type)
2. Register-Immediate Type (R_I Type)
3. Conditional Branch Type (B Type)
4. Un-conditional Branch Type (JP_T Type)

G.1.1. Register-Register Type (R_R Type)

In this type, data from two registers (Source) would be read, and operated, and the result would be written to another register (Destination). Addition and subtraction instructions belong to this type.

31	26	25	21	20	16	15	11	10	0
000000		rs0		rs1		rd		func	
Op-Code		Source Register 0		Source Register 1		Destination Register		Instruction Specific Code	

G.1.2. Register-Immediate Type (R_I Type)

In this type, data from one register (Source) and const value would be read, operated and the result is written to another register (Destination). Memory access instructions as Load and Store belong to this type.

31	26	25	21	20	16	15	0
op		rs0		rd		const	
Op-Code		Source Register 0		Destination Register		Offset	

G.1.3. Conditional Branch Type (B Type)

In this type, data from one register (Source) would be read, compared with 0, if condition is met branching takes place. Branching address is calculated by adding PC to

16 bit offset value.

31	26	25	21	20	16	15	0
op		rs0		00000		const	
Op-Code		Source Register 0		Reserved		Offset	

G.1.4. Un-Conditional Branch Type (JP_T Type)

In this type, branching takes place unconditionally, such that the branch address is the PC value added to the 16 bit offset value.

31	26	25	16	15	0
op		0000000000			const
Op-Code		Reserved			Offset

G.2. Instruction List

G.2.1. ADD Instruction

Instruction Type

Register-Register Type (R_R Type)

Instruction Field

31	26	25	21	20	16	15	11	10	0
000000		rs0		rs1		Rd		00000100000	
Op-Code		Source Register 0		Source Register 1		Destination Register		Instruction Specific Code	

Instruction Format

ADD rd rs0 rs1

Instruction Behavior

The content of register {rs0} + the content of register {rs1} would be calculated and saved in register {rd}.

$$rd = rs0 + rs1$$

G.2.2. SUB Instruction

Instruction Type

Register-Register Type (R_R Type)

Instruction Field

31	26	25	21	20	16	15	11	10	0
000000		rs0		rs1		rd		00000100010	
Op-Code		Source Register 0		Source Register 1		Destination Register		Instruction Specific Code	

Instruction Format

SUB rd rs0 rs1

Instruction Behavior

The content of register {rs0} – the content of register {rs1} would be calculated and saved in register {rd}.

$$rd = rs0 - rs1$$

G.2.3. LOAD Instruction

Instruction Type

Register-Immediate Type (R_I Type)

Instruction Field

31	26	25	21	20	16	15	0
100000		rs0		rd		const	

Op-Code	Source Register	Destination Register 0	Offset
---------	--------------------	---------------------------	--------

Instruction Format

LOAD rd rs0 const

Instruction Behavior

The content of memory address calculated by adding the content of register {rs0} and {const} would be loaded into register {rd}.

$$rd = [rs0 + const]$$

G.2.4. STORE Instruction

Instruction Type

Register-Immediate Type (R_I Type)

Instruction Field

31	26	25	21	20	16	15	0
101001		rs0		rd		const	
Op-Code		Source Register		Destination Register 0		Offset	

Instruction Format

STORE rd rs0 const

Instruction Behavior

The content of {rd} register {32 bit} would be saved in memory address calculated by adding the content of {rs} register and {const} values.

$$[rs0 + const] = rd$$

G.2.5. BEZ Instruction

Instruction Type

Conditional Branch Type (B Type)

Instruction Field

31	26	25	21	20	16	15	0
000100		rs0		00000		const	
Op-Code		Source Register		Reserved		Offset	

Instruction Format

BEZ rs0 const

Instruction Behavior

If content of register {rs0} equals 0, PC value will be incremented by {const} value and branching takes place.

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

G.2.6. Instruction

Instruction Type

Un-conditional branch (JP_T Type)

Instruction Field

31	26	25	16	15	0
000010		0000000000		const	
Op-Code		Reserved		Offset	

Instruction Format

J const

Instruction Behavior

Branching takes place unconditionally. The branch address is calculated by adding a {const} value to PC.

$$PC = PC + \text{const}$$

H. Interrupt/Exception Handling

{small_RISC} includes a reset interrupt. Reset condition and handling is explained as shown in the following table.

Condition	Reset port receives {1} at its input port
Reset Handling	Program counter, instruction register, and GPR are reset.
Handling Cycle No.	1 Cycle.