ASIP Meister

ASIP Meister [ASIPMeister] is a development environment for creating application specific instruction set processors (ASIPs). It is not the purpose of this chapter to explain the benefits or the usage of ASIP Meister. To learn the usage of ASIP Meister you have to work through the user manual and the tutorial, which are available in the “share” subdirectory of ASIP Meister. ASIP Meister itself is located in the directory /AM/ASIPmeister/ in our laboratory environment. The purpose of this chapter is to summarize some typical challenges with ASIP Meister and some typical but hard to understand error messages, that might appear while using the software. Chapter 4.4 will afterwards give a tutorial about the so-called ‘Flexible Hardware Model’ (FHM) of ASIP Meister, as this part is missing in the official tutorial.

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.

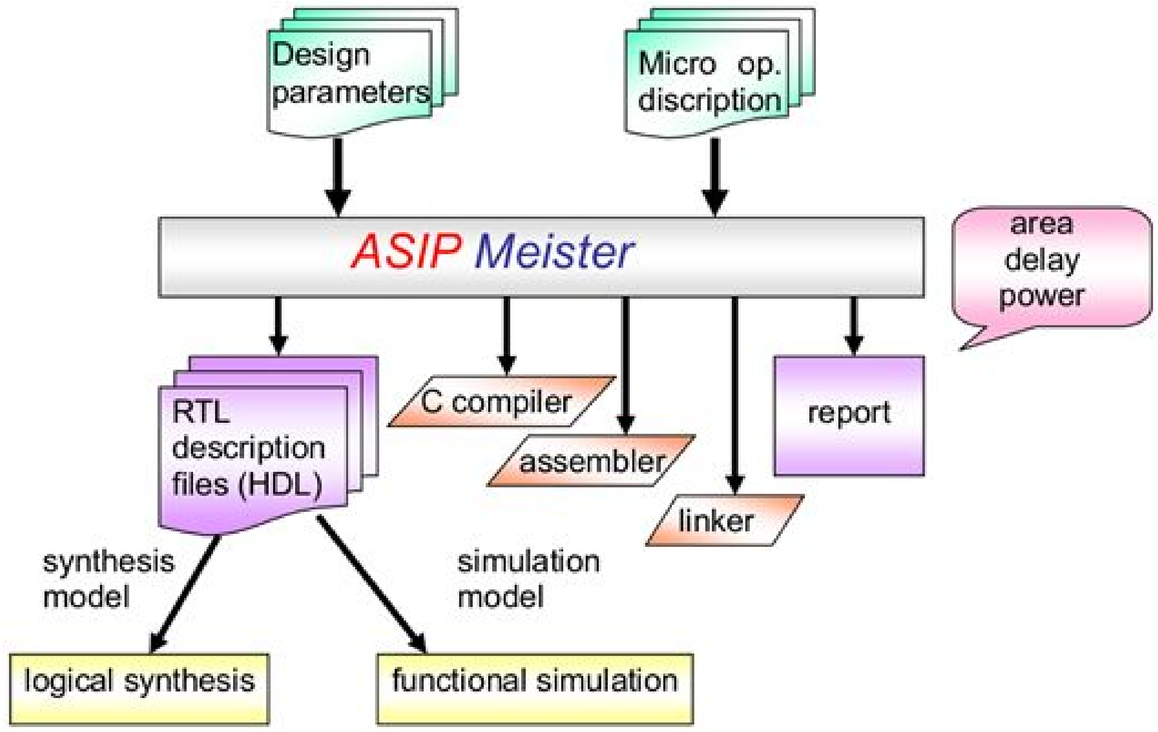

Figure 4-1: ASIP Meister Input and Output

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.

Processor Design Flow Using ASIP Meister

Open each sub-window in ASIP Meister, in the order specified in Figure 4-2. This would be typically the order of operation that should be performed to design and generate a processor core using ASIP Meister.

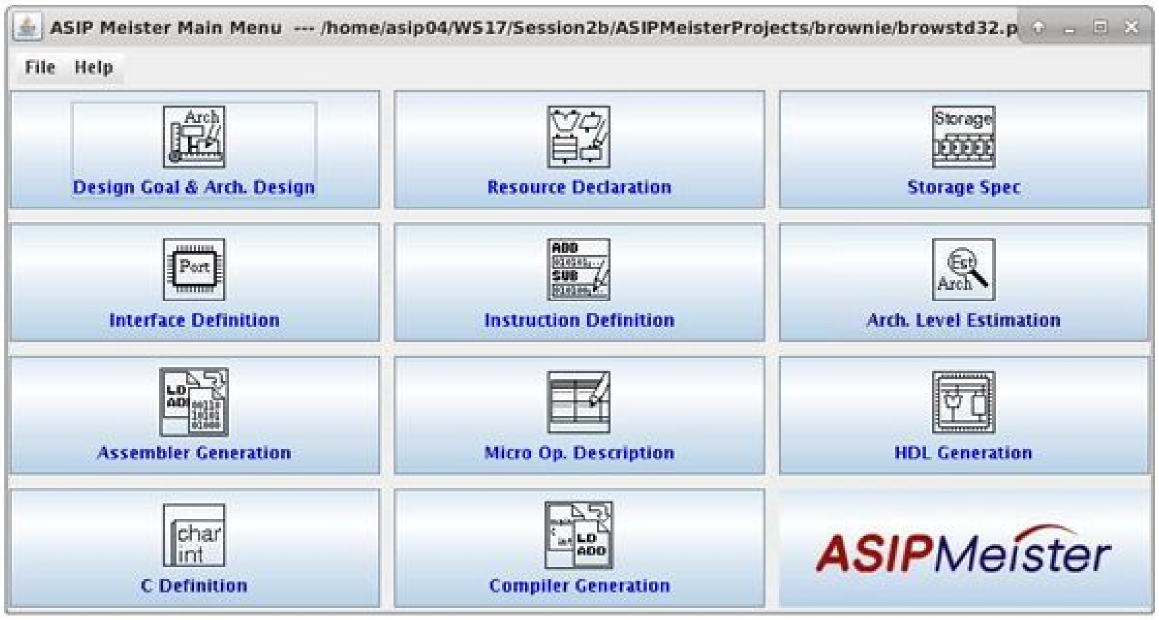

Figure 4-2: ASIPmeister main window

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

For details of the individual sub-windows, see the corresponding sections in [TUT] and [UM].

Typical Challenges while Working with ASIP Meister

- The **register r0** is hardwired to zero in our ASIP Meister CPU. Nevertheless, this is only a hint for the compiler. The compiler will never try to write a value different from zero to this register and the compiler will use this register when a zero value is needed. However, this register can be written with any value, like all the other registers. The reason for this behavior is that the register file is created by the FHM description (flexible hardware model) in the "*Resource Declaration*", and the configurations in the "*Storage Specifications*" are only used for the compiler and assembler, but are not used for creating the hardware.

- Do not use the "*meister/xxx.sim*" directory like "*browstd32.sim*", when simulating or synthesizing the VHDL code. There is a problem with the register file. Instead, use the "*meister/xxx.syn*" directory like "*browstd32.syn*". The same VHDL-code will be used for synthesis as explained in Chapter 6.
- ASIP Meister can only be used once on each computer. Therefore, if two different groups want to work with ASIP Meister, they have to use different computers. To make sure, that ASIP Meister is at most started once on each PC, the starting script tests, whether the file "**/tmp/fhm_server.log**" exists in the temporary directory. ASIP Meister creates this file while starting and it is removed after ASIP Meister terminates. If this file exists, then someone else might already use ASIP Meister on that PC. However, it can also happen, that this file is not automatically deleted, after ASIP Meister terminates. If you think, that no one else is using ASIP Meister on this PC, then remove this file.
- When writing **MicroOp** code, be careful with macros. A macro that is defined as three stage long and started in the ID phase will execute in the ID, the EX and the MEM phases, not just in the ID phase. If you want to view an instruction's *MicroOp* code without macros, select this instruction and hit the "*Macro Expansion*" button in the upper right corner.

Tutorial for the "Flexible Hardware Model" (FHM)

To add a new instruction to an ASIP Meister CPU one would usually write a *MicroOp* description for the instruction, using the facilities of an existing hardware module (ALU, shifter, adder, etc.) The ability to use multiple hardware modules during one stage and to create more than one instance of a specific module combined with the operators provided by the *MicroOp* description language is sufficient for most basic instructions. However, this method has several shortcomings that make it impossible to do the following (among others):

- Working with not program-defined wire ranges which depend on register contents or immediate values
- Implementing multi-cycle instructions

This tutorial will show how to write a new hardware module to provide a "*rotate left*" instruction.

Setting up ASIP Meister to add new FHM

In our current setup, ASIP Meister is installed globally. To modify FHMs you will need to set it up locally:

1. Copy the ASIP Meister directory tree to your home directory:

   cp -r /AM/ASIPmeister// ~

2. Update the PATH environment variable to use the local ASIP Meister directory by editing the file *~/.bashrc.user* and adding the following line at the end of the file:

   PATH=$HOME/ASIPmeister/bin/:$PATH

3. Force the shell to re-read the *bashrc* file to use the updated PATH variable:

   source ~/.bashrc or logout and login again

4. Verify that you are using the correct ASIP Meister copy: *"which ASIP-meister"* should print the path to the local copy

5. Edit *~/ASIPmeister/bin/ASIPmeister* (line 25) and make sure your local ASIP Meister path (*$HOME/ASIPmeister*) is assigned to the variable *ASIP_MEISTER_HOME*.

From now on whenever relative paths (not starting with a /) are mentioned, *$HOME/ASIPmeister* should be the base directory, i.e.: *share/fhmdb/workdb/-peas/rotator.fhm* should be */home/asipXX/ASIPmeister/share/fhmdb/workdb/-peas/rotator.fhm*

FHM structure

The directory *share/fhmdb* has two subdirectories and the file *fhmdbstruct*. ASIP Meister reads this file to determine which FHM files to use. Most of the modules necessary for the basic functions of the CPU are in the directory *basicfhmdb*; they are further divided into the categories *computational* and *storage*. New FHMs should be added to the *share/fhmdb/workdb/peas* directory.

FHMs are written in XML (eXtensible Markup Language). ASIP Meister processes their data in several stages, most importantly "*HDL Generation*" which creates the CPU VHDL files. Usually some embedded Perl is used in the FHMs to customize VHDL code. An FHM can be divided roughly into two parts: *behavior* and *synthesis*. We are not going to differ between them, though.

Header and Function description

First, we need to make ASIP Meister aware of our new FHM. Edit *share/fhmdb/-fhmdbstruct*. Go to the tag *<library name="workdb">* and add another model to the peas class by adding the line *<model>rotator</model>*. Save and close the file, as this is the only change needed for this file.

To make your task a bit easier, we have prepared a template file with many mandatory sections already done. Copy the file *share/fhmdb/workdb/peas/skeleton-1p.fhm* to *share/fhmdb/workdb/peas/rotator.fhm* and edit this file. TAKE CARE: Small errors in this file will lead to very general error messages that are hard to find. Consider the general hint in Chapter 4.6 and double-check your changes for typing errors and missing spaces.

Name your FHM "*rotator*" by editing the name in the *<model_name>* tag. Rename the author in the *<author>* tag. The <parameter> section is used to allow FHM customization from the *Resource declaration* section in ASIP Meister. The parameter *bit_width* for the input and output vector is already defined. Leave it as it is.

Next, we have the *function description*, which is generated with an embedded Perl script. Functions are used by ASIP Meister to interface the MicroOp description and the actual VHDL code. The program generates the necessary registers/multiplexers and control signals to address the corresponding hardware module. The Perl script uses the Perl *print* function to output the *function description*. The *«* and the following string start a so called "*here document"*, which instructs the Perl interpreter to treat all the following lines as single character string (performing variable substitution) until it finds the string after the *«* on a single line.

Rename the name of the *function* from "*foo*" to "*rotl*" and change the comment in the line above to something sensible (e.g. "*rotate left*").

Functions are divided into 4 blocks:

input: declaration of *function parameters*. We need the actual data and the amount by which to rotate, so write the following two lines between the curly braces of *input*:

bit [$msb:0] data_in;

bit [7:0] amount;

> Note that *$msb* is a Perl variable that will be substituted by the actual value (assigned above, *$bit_width - 1*)

output: declaration of the *function output*. The rotated value is of the same type as the input value, so add the following between the braces of the *output* block:

bit [$msb:0] data_out;

control: *control variables* used by the RTG controller of the CPU. These signals are not accessible from the MicroOp description, but can be used in the VHDL code in the FHM. For our hardware to know which direction (left or right) to shift, we will use a 1 bit signal ('0' for left, '1' for right). Add the following for the *control* section:

in direction;

The *in* keyword means that the module will be able to access the signal read-only. *out* and *inout* are the other two possibilities.

protocol: Describes what the *function* should do once a condition is met. In this case, we will use the following simple protocol:

[direction == ’0’] {

valid data_out;

}

That is, it is for the *function description*. The *function convention* is next. The content is identical to the *function description* except for the following two changes:

Write the following into the protocol block:

single_cycle_protocol {

direction = ’0’;

}

Write the following into the *control* block:

in bit direction;

To declare additional *functions*, simply add their descriptions to the “*here document*” (or use a new print statement), do not start a new XML *<function_description>* or *<function_conv>* block.

Ports, Instance and Entity

The *<function_port>* section declares the signals that will be connected to our module. As with the *function convention* and *function description* it is an embedded Perl script, and as before the output is done with the “*here document”*. We mentioned all the needed ports in the *function convention* and *description* already: *direction*, *data_in*, *amount*, *data_out*. Use the following lines for the declaration (make sure you write them between the “*print «FHM_DL_PORTS;”* and the “*FHM_DL_PORTS lines*” in the *<function_port>* part):

direction in bit mode

data_in in bit_vector $msb 0 data

amount in bit_vector 7 0 data

data_out out bit_vector $msb 0 data

You will notice two things: First, there are two types of ports: mode and data ports[1] - use *data* for your input and output signals and *mode* for control signals. Second, one bit signals are declared as bit and don’t have a range specification,

[1]There is another type: ctrl which is used for multi cycle instructions

while signals wider than one bit (vectors) are of type *bit_vector* and have a range (width) - in this case from the most significant bit down to 0.

The *<instance>* is the core of the module - the actual architecture VHDL code. Embedded Perl is used here as well. Go to the *$signals* variable. As you can see here, documents can also be used to assign values to variables. No additional signals are necessary for our rotator, so we will leave *$signals* as it is.

The next variable, *$vhdl* is more interesting: Our module should be sensitive to changes of input data, shifting amount and shifting direction (so it should re-compute the output data if one of these three values changes), hence we define a process with these three values in its sensitivity list. We also need one integer variable to hold the value of the shifting amount (*amount* is a signal, not a variable) and one signal for the temporary value of the result. After the *begin* keyword we can write the process code. Remember that this variable (as all the others) will later be used to create the actual VHDL output (the variable is not placed at the correct position for the VHDL output, but instead it is later used at the correct position). If you are writing statements that span multiple lines you have to encapsulate them with double quotes (“ ”) to assign them to the variable.

First, we need to convert the signal *amount* to integer and assign it to *a*. Next we check if *a* is within range, if it is not, we set the result to *undefined*, otherwise we can rotate. A case switch is used to decide into which direction to rotate. The *others* case is necessary, as the type *std_logic* has more states than just ’0’ and ’1’, so don’t delete it when you add the code for "rotate right".

At the end of the process, we assign the value of the *res* signal to the *data_out* signal. See below listing for the architecture VHDL:

```
process (data_in, amount, direction)
variable a : integer;
variable res : std_logic_vector($msb downto 0);
begin
a := TO_INTEGER(UNSIGNED(amount));
if (a > 0 and a < $bit_width) then
case direction is
when ’0’ => -- rotate left
res($msb downto a) :=
data_in($msb - a downto 0);
res(a - 1 downto 0) :=
data_in($msb downto $bit_width - a);
```

when others => -- not reached

res := (others => 'X');

end case;

else

res := (others => 'X');

end if;

data_out <= res;

end process;

Leave the comment section untouched and look closer at the FHM_DL_TOP_2 document. First, three libraries are included - these are necessary for the std_logic data types and several functions and macros. Next, the entity is declared, which states all input and output ports of our VHDL module. Although we already defined the ports of our FHM, these were interpreted by ASIP Meister - the port declaration for the entity, as well as the rest of the VHDL code will be used verbatim, without any error checking by ASIP Meister (although we can still check for errors in ModelSim). Use the code in the below listing (between the "*print «FHM_DL_PORTS;"* and the "*FHM_DL_PORTS lines*" in the <*instance*> part) for the entity ports:

data_in : in std_logic_vector($msb downto 0);

direction : in std_logic;

amount : in std_logic_vector(7 downto 0);

data_out : out std_logic_vector($msb downto 0)

That is for the <*instance*> block. Next, we have an <*entity*> section. It defines an *entity* in a different file, which is why a new block is needed, but the ports are the same, so use the code from the above listing.

Estimation and the Synthesis Model

The <*testvector*> section may be left empty, the <*synthesis*> script contains instructions to process the FHM file - we leave that untouched as well.

The <*estimation*> block has data relevant for area, power and delay estimations, but we use more accurate tools, that consider the actual application execution, as shown in Chapter 6.5, and Chapter 7.2. Leave the estimation data there though, as ASIP Meister will complain without them. You should change FHM parameters however, you will have to adjust the estimation section if you want to get rid of the warnings.

That is it for the *behavior* model. As mentioned in the beginning, we won't differ between the *behavior* and the *synthesis* model, so just copy-and-paste the complete <*model*> block and change the value of the <*design_level*> tag from behavior to synthesis.

Your FHM file should now have a structure similar to the following listing:

```
<?xml version="1.0" encoding="Shift_JIS" ?>

<FHM>

<model_name> rotator </model_name>

<model>

<design_level> behavior </design_level>

[...]

</model>

<model>

<design_level> synthesis </design_level>

[...]

</model>

</FHM>
```

You can now use the module in ASIP Meister. Instantiate the FHM resource in the *Resource Declaration* and write the new instruction *rotl*. Set the operands to the correct Addressing Mode, DataType, etc in the *Behavior Description* window, but leave the behavior description itself out. Use the syntax

```
result = ROT0.rotl(source0, am);
```

where *result* and *source0* are 32-bit wires and *am* is a 8-bit wire.

Testing the new FHM

When HDL and SWDev Generation run without errors (SWDev will probably print some warnings about setting throughput to 1 - that is alright), proceed testing your module/instruction. Write a small assembly code, compile it (as shown in Figure 2-2), create a new project in ModelSim, load your design and compile it. If there are errors during compilation of the VHDL files (especially in *fhm_rotator_w32.vhd)*, then you have made a mistake in your VHDL code. Now you could go back to the FHM and correct it there, but a faster way is to edit the mentioned VHDL file in your *~/ASIPMeisterProjects/browstd32_YourCPU/-meister/browstd32_YourCPU.syn/* directory and try to correct the code there. Once you get it running, make the corrections in the FHM file as well (important: in the behavior and synthesis sections), as ASIP Meister will overwrite the VHDL files every time you recreate the CPU.

Once your CPU VHDL files are compiled, run your test program. Check the results carefully! Experiment with large values as well (i.e. use *lhi %r2, $65535* to set the upper 16 bits to '1'). Once you verified your design, backup your FHM (important!) and implement the "*rotr* " - right rotation instruction by making the necessary changes to your FHM.

Multi Cycle FHMs

As mentioned at the beginning of the tutorial, one of the reasons to write custom FHMs is to implement multi-cycle (stalling) instructions. Examples of stalling instructions are the multiplication and division operations, which use dedicated multiplier and divider hardware blocks. Stalling hardware is usually implemented with *State Machines.* This section will show you how to write a FHM for a multi-cycle instruction. We will extend the rotator from the previous sections - the multi-cycle version will rotate only one bit per cycle (obviously the performance will be inferior to the single-cycle variant; it is just for demonstration).

ASIP Meister provides three signals to control multi-cycle instructions: *start*, *fin* and *cancel.* The *start* signal will be set by the RTG controller once the instruction is started, and our hardware will set the *fin* signal once the operation is finished, to signal the CPU that normal pipeline operation can be resumed. If a *cancel* signal arrives, the hardware should abort its operation. We will disregard the cancel signal here, but reset our hardware on the CPU *reset* signal into its default state.

<function_description>

We need to define the three control signals in the *control* block of our function. As the *control* block of your function use

in start, cancel, direction;

out fin;

We also need a protocol for our stalling instruction. For the *protocol* block use:

repeat [start == 1] until (fin == 1 || cancel == 1);

if (fin == 1 && direction == 0) {

valid data_out;

}

<function_conv>

Add the three signals to the *control* block of your function. It should be now:

in bit direction;

in bit start;

in bit cancel;

out bit fin;

and for the *protocol* block use:

multi_cycle_protocol {

start_signal start = ’1’;

fin_signal fin = ’1’;

cancel_signal cancel = ’1’;

direction = ’0’;

}

<function_port>

The three signals as well as the CPU *clock* and *reset* signals need to be added to the port declaration of our FHM, so use:

direction in bit mode

data_in in bit_vector $msb 0 data

amount in bit_vector 7 0 data

data_out out bit_vector $msb 0 data

clock in bit clock

reset in bit reset

start in bit ctrl

fin out bit ctrl

cancel in bit ctrl

Note that the port type is *ctrl*, not *mode* for the three multi-cycle control signals, and *clock* and *reset* for the two CPU signals respectively.

<instance>

The VHDL implementation uses one process for the state machine, but this time we only use *clock* and *reset* in its sensitivity list. In the process body, we first handle the reset case (synchronously), and then depending on the current state we either start the state machine from the idle state (*st0*), execute a one-bit rotation (*st1*), or assign the result (*st2*). The following code is provided as text file as well:

$vhdl = "process (clock, reset)

type t_s is (st0, st1, st2);

variable state : t_s;

variable a : integer;

```
variable res : std_logic_vector($W1 downto 0);
variable tmp_data : std_logic_vector(31 downto 0);
begin
if rising_edge(clock) then
-- handle reset (synchronous)
if reset = ’1’ then
state := st0;
data_out <= \"00000000000000000000000000000000\";
fin <= ’1’;
else
case state is
when st0 =>
if start = ’1’ then
state := st1;
a := TO_INTEGER(UNSIGNED(amount));
tmp_data := data_in;
end if;
fin <= ’0’;
when st1 =>
if (a > 0 and a < $bit_width) then
case direction is
when ’0’ => -- rotate left one bit
res($W1 downto 1) := tmp_data($W1 - 1 downto 0);
res(0) := tmp_data($W1);
when others => -- not reached
res := (others => ’X’);
end case;
a := a - 1;
tmp_data := res;
else
if a /= 0 then
```

```
res := (others => ’X’);
end if;
state := st2;
end if;
fin <= ’0’;
when st2 =>
data_out <= res;
state := st0;
fin <= ’1’;
end case;
end if;
end if;
end process;";
```

Further down, in the *entity* declaration, we will need to add our three control signals as well as the clock and reset signals. The entity should now have the following signals:

```
data_in : in std_logic_vector($W1 downto 0);
direction : in std_logic;
amount : in std_logic_vector(7 downto 0);
data_out : out std_logic_vector($W1 downto 0);
clock : in std_logic;
reset : in std_logic;
start : in std_logic;
cancel : in std_logic;
fin : out std_logic);
```

<entity>

Again, our five signals need to be added to the *entity* declaration. The new *entity* should have:

```
data_in : in std_logic_vector($W1 downto 0);
direction : in std_logic;
amount : in std_logic_vector(7 downto 0);
data_out : out std_logic_vector($W1 downto 0);
```

clock : in std_logic;

reset : in std_logic;

start : in std_logic;

cancel : in std_logic;

fin : out std_logic);

Estimation, Synthesis, ASIP Meister usage and Testing

No changes for estimation or synthesis but make sure to copy your changes to the behavior *<model>* section of the FHM.

Instantiate and use the resource in ASIP Meister just as with a singly-cycle FHM - no changes needed. Write a test program and check in ModelSim whether the pipeline actually stalls (the *im_addr* value shouldn't change during stalling) and whether the result is the same as with the single-cycle instruction.

General Hints about FHMs

- Do not copy and paste the code from this *pdf* file. Often, this manifests in wrong code (e.g. wrong blanks) and the effort to debug the code afterwards is bigger, than manually transcribing the code.
- ASIP Meister has only very few debug facilities for FHMs (e.g. */tmp/fhm_server.log* contains some more information compared to the popup windows). Therefore, you usually have to use the trial and error scheme. The syntax for FHMs is very restrictive. Sometimes a missing blank can cause a problem. Sometimes (!) you can ignore error messages in the *Resource Declaration*, as long as the resource was successfully instantiated. These error messages are meant for the *Estimation,* which is not needed for the implementation.
- The FHM example in this tutorial is constructed for a module with one parameter. When you need more parameters, have a look at the other existing FHMs.
- As soon as a new FHM is successfully constructed, you can make further changes directly in the VHDL code for simplicity. Whenever the VHDL code is finalized, you should include it into the FHM file again, as the created VHDL files are overwritten every time you regenerate your CPU.
- Create backups very frequently. Due to the difficult debugging, it is difficult to find and fix bugs and thus it is often simpler to go back to a slightly earlier version and to implement the changes again.

Synthesizable VHDL code

Here are some hints for improving the chance, that your code is synthesizable. These are not general statements that are true under all circumstances or that guarantee that your code will be synthesizable.

- VHDL procedures often make problems for synthesizing (aborting with error message). Avoid them unless you know what you do.
- Within a process only use the *'event* modifier once, e.g. if *clk='1'* and *clk'event.*
- Avoid *wait* statements.
- Avoid the data types '*bit*' and '*bit_vector*', but use '*std_logic*' and '*std_logic_vector*' instead.
- Typically everything inside a process should be synchronous, e.g.:

```
MyProcess : process (clk)
begin
if rising_edge(clk) then
-- rising_edge(clk) is a macro for clk'event and clk='1'
if reset = '1' then
...
else
...
end if;
end if;
end process;
```

- Initialize all signals/variables in the reset statement. The initialize-statement (e.g. "*variable foo : integer := 42;*") is either ignored or only evaluated when the FPGA is configured, but certainly it is not evaluated when you press the reset button.
- In VHDL simulation, the output of a process is only recomputed, if any signal in the sensitivity list had an activity/event. In hardware, the processes run all the time, as they are implemented in dedicated hardware. This can lead to correct simulation results that are not reproducible in the FPGA prototype. Therefore, the sensitivity list is only meant for simulation and has no impact to the synthesis. If everything inside your process is synchronous (like in the above example), then the *clk* is the only signal you need in your sensitivity list. Otherwise, all signals that are read need to be considered in this list!

- Often a final state machine (FSM) is needed. Below is an example for it. A different approach that also supports VHDL procedures can be found in [FSM].

```
MyStateMachine : process(clk)
type stateType is (state0, state1);
variable state : stateType;
begin
if rising_edge(clk) then
if reset = '1' then
state := state0;
else
case state is
when state0 =>
...
when state1 =>
...
when others =>
...
end case;
end if;
end if;
end process;
```