Working Environment

This chapter explains the technical environment for the laboratory. This includes the usage of the computers and the directory structure for this laboratory. It is very important to understand completely the directory structure, as many scripts rely on this special structure and will not work at all or create an unexpected output if the directory structure is set up in a wrong way.

Network Structure

The programs needed to perform the laboratory tasks run under Linux. You will work with the computers in the lab 308.2. This room has 10 computers with Ubuntu 14.04, 32bit installed and are named as follow:

- i80labpcXX.ira.uka.de where XX=01, 02, ... 10.

These computers can also be accessed remotely via SSH or via X2Go Client, see Chapter 2.2.1 and 2.2.2. The user name like asipXX where XX=01, 02..., 10 and password to login to these computers for different student groups will be distributed at the beginning of the lab. You can also access these computers from student lab (Room 312.4) or from your personal Laptop. X2Go Client is already installed on all computers in the student lab (Room 312.4) while you have to install and configure X2Go Client on your Laptop at your own. It is recommended to use SSH instead of X2Go as it has some conflicts with some settings.

For ASIPmeister and GCC compiler, it is necessary to change the machine in order to meet the software requirements. There is one dedicated application workstations:

- i80pc57.ira.uka.de – ASIPmeister system:

You will use this PC for building your own customized CPU and compiler that is able to use the instructions you added to the instruction set of the basic processor. This task also needs a powerful machine to complete in a reasonable amount of time.

To run, this lab, you need to export the following variables, or you can add it the your /home/.bashrc.user.

```
export ASIPS_LICENSE=29000@i80asip.ira.uka.de
export PATH=/AM/ASIPmeister/bin:$PATH
export ASIP_APDEV_SRCROOT=~/AM_tools
export PATH=/usr/java/jre1.6.0_45/bin:$PATH
export ASIPmeister_Home=/AM/ASIPmeister
export ASIPmeister_HOME=/AM/ASIPmeister
```

source /home/adm/modelsim_66d.setup

source /home/adm/xilinx_13.2_32bit.setup

In most cases, you have to manually login into this machine to work with the ASIPmeister and GCC Compiler. We will provide you the scripts that will do most of the work for you. Just be aware of the fact that it will sometimes take time to complete everything. If some errors occur, try to find out what is wrong by reading the script output carefully. This will show what went wrong. Start again and if it fails again then ask your tutor. The most important thing is to know or to recognize that the results of the script task are right or wrong. This depends on the output, so read it carefully before starting the next task.

The lab program directory is mounted via NFS to your client. Thus, almost every program will work on your local machine. Data supplied to the groups or created by them will be stored on a server directory that is always available for a client machine. All client machines are configured as NFS clients, which mount the corresponding home directory via NFS. This means you can work wherever you want. The hostname will change, but your home directory will show the same content on all clients.

To switch between the machines mentioned above you have to use SSH (recommended). The names of the client PCs are similar to i80labpcXX.ira.uka.de with the 'XX' replaced by the individual number of the PC. Login with SSH requires authentication, which can be done with passwords or public keys (see Chapter 2.2.1). In order to minimize the efforts for changing machines we recommend public keys, so you do not have to type your password for every remote login.

Basic UNIX Commands/Programs

If you are familiar with the Unix/Linux environment, you can skip this section. In this section, "Unix" will refer to GNU/Linux mostly.

- **Command Line Interpreter**: Interaction with the system usually is done via the command line, although some file system operation can be done with graphical or text based file managers.

  The command line interpreter (generally called "shell") runs on a terminal and primarily provides the user with the means to start programs. It also has several built in commands as well as a scripting language. The default shell in the lab is *bash* (Bourne Again Shell).

  Some programs and commands require command line arguments (parameters), which can be supplied in a space-separated list to the program, i.e.:

  cp file1 file2 executes the program "*cp*" with the arguments "*file1*" and "*file2*".

- **Online Help System**: Most UNIX commands come with documentation in the manual page format (*man*-pages). They usually give a detailed description of the program, all command line parameters, and some examples. Use the *man* command to read *man*-pages, i.e.: "*man ls*" calls the *man*-page for the "*ls*" command. Use the up and down arrow keys (or Page Up/Down) to navigate in the man page viewer and "*q*" to quit.
- **Interacting with the File System**
  a. **File System Structure**: UNIX organizes all files (regular and special files) in a tree structure. The root of the tree is called "/", directories are the nodes (leaf or non-leaf) and files the leaf nodes of the tree. The file system does not have a concept of drives – new data from external media (networked or removable) is accessed by attaching the sub tree of the new file system to the UNIX file system tree somewhere. Usually ordinary users are not permitted to do this.

  b. **Working Directory**: Every process has a current working directory – a pointer to a directory node in the file system tree. The current working directory is referred to as "." and the parent directory as "..". To print the current working directory of the shell, use the command "*pwd*" (print working directory).

  c. **Home Directory**: Every user has a home directory – one for which he has write access. All your data will be saved in subdirectories of your home directory. Home directories are mounted via NFS from a remote server. Home directories have the form of "*/home/asip01*" but a user can refer to his home directory with "~". Hence, "*/home/asip01/test*" and "*~/test*" (executed by the user "*asip01*") refer to the same.

  d. **File Paths**: To specify a file or a directory, a path name must be provided. There are two kinds of paths: absolute and relative. An absolute path starts at the root directory, a relative path starts at the current working directory. File system nodes (path name components) are separated by "/" (the same as the backslash in Windows). A relative path may be like "*./ASIPMeister-Projects/browstd32*" which refers to the file or directory "*browstd32*" of the subdirectory ASIPMeisterProjects of the current working directory. An example for an absolute path for the same relative path is "*/home/asip01/ASIPMeisterProjects/browstd32*".

  e. **Changing the Current Working Directory**: To change the current working directory use the *cd* command with a path name as the argument, i.e.:

```
cd /home/asip01

cd ..

cd ../asip02/ASIPMeisterProjects
cd ./browstd32/Applications
```

f. **Creating/Removing Directories**: the "*mkdir*" command takes a pathname as an argument and creates a new directory accessible via this path. The "*-p*" parameter tells mkdir to create any missing subdirectories as well. "*rmdir*" deletes empty directories. To remove non-empty directories use the "*rm*" command with the "*-r*" option (see below). Some of the examples are::

```
mkdir ~/ASIPMeisterProjects/browstd32/Applications
```

```
rmdir ../browstd32/test1
```

```
mkdir -p ~/some/non_existing/directory
```

g. **Listing Directory Contents**: To examine directory contents use the "*ls*" command. Without parameters, it will show the file and directory names of the current working directory. The "*ls*" command has many parameters, all described in the man-page. Typical ones are "*-l*" to provide a detailed listing, "*-a*" to show hidden files (filenames starting with a "."), "*-ltc*" gives a detailed listing with the last file modification time and sort by it (actually these are three parameters). Some of the examples are:

```
ls
```

```
ls -l ../ASIPMeisterProjects
```

```
ls -ltc ASIPMeisterProjects/browstd32/ModelSim
```

h. **Moving and Removing Files**: To move or rename a file use the "*mv*" command with the filename or directory as the argument. To remove a file use "*rm*"; the "*-r*" option causes recursive removal of subdirectories and their contents. Some of the examples are:

```
mv Applications/tset Applications/test
```

```
rm /home/asip01/oldfile
```

```
rm -r ../browstd32
```

i. **Copying Files**: The "*cp*" command copies files and directories. Its arguments are: optional switches, then the source and the target file/directory respectively. The "*-r*" switch enables recursive copying. Some of the examples are:

```
cp TestData.IM TestData.IM-backup
```

```
cp -r browstd32 browstd32_custom
```

- **Shell Operation:**

  a. **Input/Output Redirection**: Some programs read data on their standard input file descriptor (stdin), some write output to the standard output (stdout). Usually stdin is linked to the keyboard and stdout to the terminal screen. However, you can change this when calling a program. "< *somefile*" redirects stdin to read from "*somefile*" and not from keyboard, likewise *"> someotherfile"* will redirect stdout to write to "*someotherfile*" (if it doesn't exist, it will be created, if it does, it will be truncated first – use "*» someotherfile*" to append instead of truncating the previous contents. Redirection is also possible to/from other processes via pipes: use "*program1 | program2*" to direct the output of "*program1*" to the input of "*program2*". Some of the examples are:

ls -ltc >filelisting

ls | sort

cat <file1 »file2 (this appends the contents of file1 to file2).

b. **Job Control**: Processes started from the shell are often referred to as "*jobs*". Usually, when the shell launches a program it will take control of the terminal and the shell will be suspended until the process terminates – the job "is running in the foreground". To launch a job "in the background" – start it, but give control back to the shell immediately, append "&" at the end of parameter list. To list any jobs launched from this shell (they all have to be in the background), type "*jobs*" – this will give you the job-IDs with the program names and parameters of the jobs. To bring a job to the foreground, use "*fg %job-ID*" (substitute job-ID for the actual job ID). Sometimes you will want to temporarily stop a job – if the program is in the foreground, hit "CTRL-Z" (job suspension). To continue job execution in the foreground use "*fg*" as described above, or "*bg %job-ID*" to continue execution in the background.

   To terminate a job in the foreground, hit "CTRL-C" (send keyboard interrupt). Note that if a process is unresponsive (due to bugs – i.e. endless loop and signal processing disabled) it can ignore this; the only way to terminate it is to send a KILL signal as explained below).

- Some important programs and commands

  a. **Process/Activity Listing**. To view a complete list of running processes, use the "*ps*" program. It is mostly used with the "*auxw*" options to provide a complete and detailed listing. The output is usually quite long, so it is often piped into "*less*" or "*head*".

  b. **Text Viewer**. "*less*" is a program to view text (you can use it to view binary data as well, but it treats it as a byte stream without any special formatting, except for control characters – they are substituted). The argument is the file to view; otherwise, data can be piped if used without any arguments. To scroll through the "*less*" output use the up and down arrow keys (or Page Up/Down) to navigate, "*q*" to quit. To jump to a specific line (current line and byte numbers are displayed at the bottom), type the line number followed by *G* (i.e. *123G* to jump to line 123). Some of the examples are:

```
less ModelSim/TestData.DM
```

```
ps axuw | less
```

c. **Difference between Files**: The "*diff*" program examines two files for differences and displays these. Useful to compare outputs of programs and see if and what changes occurred. Common options are "*-u*" (unified output) to have verbose output. Unified output lists the lines that were in the old file, but are not in the new file preceded with a "-" and files that were not in the old file and are in the new file preceded with a "+". Lines that start with neither are just context to make orientation a bit easier. File sections where changes were detected start with @@, followed by the line ranges of the sections. A graphical diff-tool that visualizes the output of "*diff*" is "*kompare*" (the 'k' instead of 'c' for 'compare' denotes that it is meant for kde; so this is not a typing error). Some of the examples are:

```
diff -u TestData.DM-OLD TestData.DM | less
```

```
kompare EvilFile.txt GoodFile.txt &
```

d. **Show only the Beginning/End of a File/Output**. The "*head*" and "*tail*" programs show only the first and last lines (respectively) of a file or the data that are piped in. By default they show 10 lines, but the "*-n x*" option sets it to x lines. Tail can also monitor a file for future appends and display them (useful when watching log files of a running program). This is requested with the "*-f*" option. Some of the examples are:

```
ls -ltc Applications/bubblesort | head -n 5
```

```
tail -n 0 -f /tmp/logfile
```

e. **Show Machine's Activity Summary**: while the "*ps*" command gives a detailed overview of the running processes, sometimes a summary is more useful. "*top*" shows the current number of running processes, the CPU(s), memory and swap usage and the active processes. It updates every three seconds and can be exited with "*q*".

Remote Operation

a. **Remote login**. To log onto a different machine, the SSH program is available (Secure Shell). Communication between the two machines is encrypted by SSH. To simply log onto a different machine with the current user, use "*ssh hostname*". If your account username is not the same as on the local machine use "*ssh otheruser@hostname*", where "*otheruser*" is the username of the account on the remote machine. SSH will ask for a password on the remote machine, and then log in and start a shell. "*exit*" will end the remote shell and terminate the SSH connection leaving you on your local machine.

b. **Copying via SSH**: Not all machines have NFS mounted home directories, so you will sometimes have to copy files to a remote machine. The SSH package provides "*scp*" for this, which can copy files or directories from one machine to another (if you have an account on the remote machine). The syntax is "*scp somefile remotehost:*" (note the colon at the end, omitting it will cause scp to copy "somefile" to the file "remotehost" on the local machine). To copy directory trees use "*scp -r directory remotehost:*"

c. **X11 Forwarding**: GUI programs can be run on remote machines as well. Log in on a remote machine with "*ssh -X remotehost*" and start a program installed on remotehost. It will be displayed on your machine, but will run (access files and use resources – CPU, memory, etc.) on remotehost.

d. **Public Key Authentication**. An alternative to providing passwords for every login is public key authentication mode. A key pair is created on your local machine and you can copy the public key to any machine you want to log on later. Once set up, authentication is done automatically and no interactive passwords are necessary. The steps to setup a public key authentication is as follows:

   i. First, create the DSA-keys by logging into any machine and invoking:

ssh-keygen –t dsa

Confirm the default destination for the keys. Afterwards enter your pass phrase, for example, your group name or you can just leave it empty. Sometimes you will be asked for this, so remember what you entered. Public and private keys will be stored in your "*~/.ssh*" directory.

   ii. Then copy the public key to the remote machine:

ssh-copy-id –i ~/.ssh/id_dsa.pub user@remote-machine

Enter your password to confirm this step. Afterwards log out and try to log in again. This should work without asking you to enter the password.

X2Go Client

X2Go is a program used to run graphical applications on Linux machines remotely. This uses a technology, which results in better performance. This also allows for suspending and resuming sessions and programs, while they are running. This allows the use of long-running graphical applications.

- Installation

    a. The X2Go Client software is already installed on all computers in out student lab.

   b. For your Laptop, you can download and install the X2Go Client from: http://wiki.x2go.org/doku.php/download:start

- Configuration

  When you first run the X2Go client, a "*New session*" dialog should appear otherwise you can create new session using "*Session*" menu and then clicking on "*New Session...*" You should fill this in with the following information in "*Session*" tab:

a. Session name - Any name you like to identify the session - if you're connecting to the PC "*i80labpc01*", you might just want this to be " *i80labpc01*"

b. Host - Full name of the computer you're connecting to, e.g. "*i80labpc01.ira.uka.de*"

c. Login - Your user ID, for example "*asip01*"

d. Session type - Select XFCE (recommended) - This is a low-power window manager that is the only one supported in the current version of Ubuntu.

e. Keep all the other setting to default.

- Connecting

   a. To start the session, click on it and enter your password when prompted

   b. After you click OK, it will connect to the server and start your session. Watch the Status line to see what is happening. Once the status is "running," your session should launch.

   c. Sometimes "*i80labpcXX*" cannot be accessed remotely using X2Go, while it works fine with SSH. May be there is a conflict between some settings in your ~/.bashrc.user and X2Go. For example one temporary solution is to comment out the line ". /home/adm/xilinx_13.2__32bit.setup" in ~/.bashrc.user. This makes X2Go work properly. If Xilinx ISE is required, you can source it manually.

Directory Structure

In development environments it is very useful to force developers to meet certain rules how program data is stored. Therefore, we provide a template that makes it easier for the tutors to find the results of every group and enables us to write script files that work on fixed locations to speed up the work. On the other hand, these script depend on this directory structure. You have to understand and use this directory structure to avoid problems.

The main directory structure is shown in Figure 2-1. Your home directory contains one special directory called "*ASIPMeisterProjects*". All your

ASIPMeister Projects including your applications and all other files will be placed in this directory. Inside this "*ASIPMeisterProjects*"-directory every project has a subdirectory (e.g. "*browstd32*" or "*AnotherProject*"). We recommended that you create new subdirectory/project for each changed version of the CPU. Among the projects the "*ASIPMeisterProjects*"-directory also includes a "*TEMPLATE_PROJECT*". This directory gives you a basic ASIP Meister Project with minimum director structure and files required to start your ASIP projects. This template is a good starting point to create a new project from the scratch. Usually you can copy from your last project to create a new one, but sometimes it is recommended to start from the scratch.

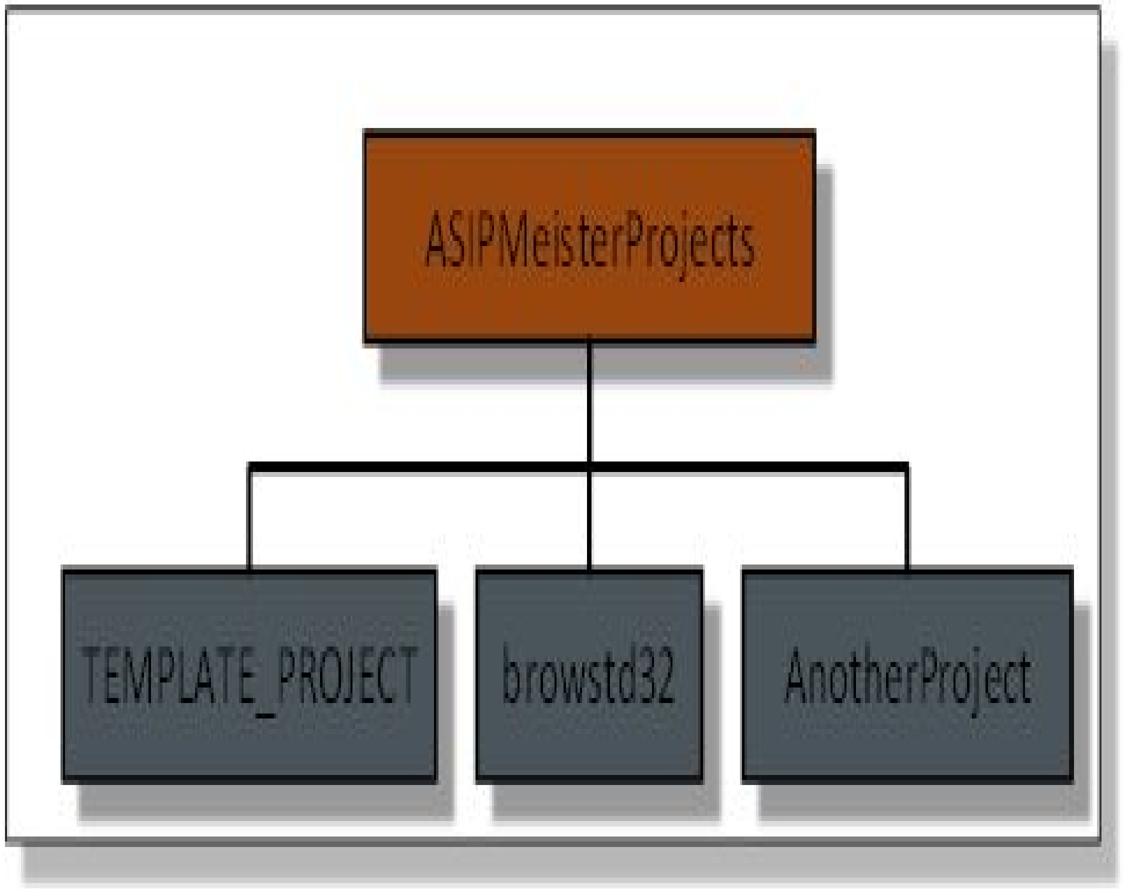

Figure 2-1: The Directory Structure for all ASIP Meister Projects

The biggest part of the directory structure is placed inside each ASIP Meister project directory, e.g. "*browstd32*", as shown in Figure 2-1. Every project directory contains **five** subdirectories and a set of local files. These directories and files are explained in the following sections.

The **"Applications"** directory should contain all your user applications that you want to run and simulate on the CPU, each in a separate subdirectory. Every application should be placed inside a separate subdirectory for this application along with a *Makefile* that is needed to compile, simulate and implement the application. When you want to compile an application, simulate it (dlxsim or ModelSim), generate its bitstream for FPGA, or upload its bitstream into FPGA, you run the *Makefile* in that particular application subdirectory. To see different options that *Makefile* offers, execute "***make***" in an application subdirectory, as shown below:

```
asip04@i80labpc04:~/ASIPMeisterProjects/browstd32/Applications/Application1:$make

/--------

| USAGE:

\--------

'make sim': compile for dlxsim/Modelsim

'make dlxsim': compile for dlxsim and directly start simulation

'make fpga': compile for FPGA and update bitstream

'make upload': upload the existing bitstream to the FPGA (note: this command
does not generate a new bitstream)

'make all': compile for dlxsim/Modelsim and for FPGA

'make clean': delete the BUILD directories

/--------------------

| PASSING PARAMETERS:

\--------------------

'DLXSIM_PARAM=...'

Example: 'make dlxsim DLXSIM_PARAM="-da0 -pf1"'

Note: These are the default parameters. Don't forget the double high-commas
(i.e.: ") when passing multiple parameters.

'GCC_PARAM =...'

Example: 'make sim GCC_PARAM =-O3'

Example: 'make dlxsim GCC_PARAM =-O3 DLXSIM_PARAM="-da0 -pf1"'

Note: If you want to enfore re-compilation with different parameters then you
have to 'make clean' to make sure that all files are re-compiled
```

For example, if you want to work with "*Application1*", as shown in Figure 2-3. Your first step is to compile this application. Inside the "*Application1*" directory containing source code "*application1.c*" or "*application1.s*". To compile it you have to execute the following in the "*Application 1*" directory:

```
asip04@i80labpc04:~/ASIPMeisterProjects/browstd32/Applications/Application1:$make
sim
```

The *Makefile* is expected to be executed from the directory, where the results will be placed, so always execute the scripts from the application specific subdirectory, never from the "*Applications*" directory itself! The details about the parameters, the created output files and the different versions of the Makefile scripts are explained in Figure 2-2. The concept of calling the scripts from the specific application subdirectory is also important for the Makefile script while executing "*make sim*", "*make dlxsim*", "*make fpga*" and "*make upload*" as explained in Figure 2-2.

Script	**Description**
make sim	It will compile your assembly or C-file file in your current application directory and "*BUILD_SIM*" subdirectory is created in your current directory containing different files like "*TestData.IM*" and "*TestData.DM*" file for the ModelSim. The output assembly file named "NameOfTheApplication-Directory.dlxsim" will also be generated. You can pass different parameters to this Makefile as follows: i. *GCC_PARAM* is used to set optimization level for GCC Compiler. The compiler options are directly forwarded to the compiler binary e.g. "-O0" or "-O4" for different optimization levels. for example: make sim GCC_PARAM =-O3

Script	**Description**
make dlxsim	It will start the dlx simulator to simulate the compiled file generated from the previous stage. Similar to previous command, "*BUILD_SIM*" containing different files like "*TestData.IM*", "*TestData.DM*" and "NameOfTheApplication-Directory.dlxsim" is created. Here, you have to pass some parameters to dlxsim mentioned in the Figure. You can also pass *GCC_PARAM* with this command. Another parameter that is required with this command is DLXSIM_PARAM, which is used to pass different dlxsim parameters, for example: make dlxsim DLXSIM_PARAM="-fAppName.s -da0 –pf1 -lfputc.out " In the command, "*AppName.s*" is loaded into dlxsim for simulation, without debugging and with pipeline forwarding. Moreover, "*putc.out*" is the file containing the printed output. Executing "*make dlxsim*" without any parameters, will execute the file "*BUILD_SIM/AppName.dlxsim*" with "*-da0*" and "*-pf1*".
make fpga	It will compile your assembly or C-file, generate the required DM/IM
make all	It will compile for dlxsim and for ModelSim and generate final bitstre
make upload	It will upload the existing bitstream to the FPGA BlockRAM. Note t
make clean	It cleans your current project directory by deleting "*BUILD_SIM*".

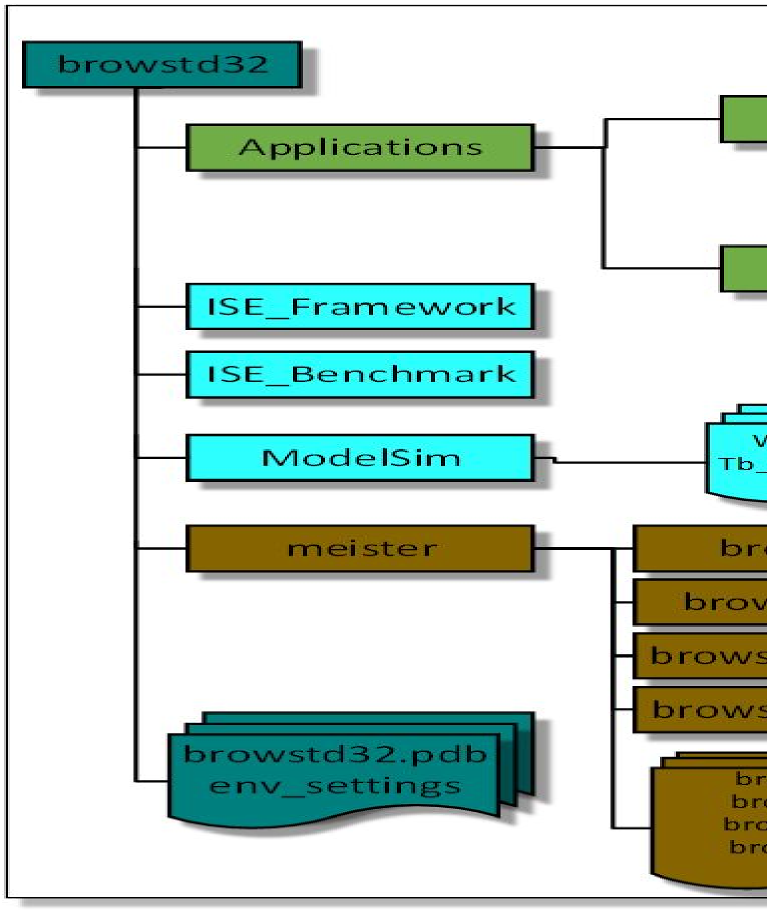

Figure 2-2: Makefile Options and Parameters
Figure 2-3: The Directory Structure for a Specific ASIP Meister Project

The *Makefile* script inside the "*Applications*" directory is only wrapper for the real scripts. This wrapper first read the "*env_settings*" file from your project directory (e.g. "*browstd32*" in Figure 2-3). This file contains all the information about your current project, e.g. which compiler to use and which dlxsim to call. Afterwards the wrapper script calls the real scripts. The real scripts are placed at a global position. This enables us to make changes to these scripts without the need to copy these changes to all your projects. The "*env_settings*" file is explained in Figure 2-5.

The ***"meister"*** directory contains the ASIP Meister output, like the VHDL and compiler generation files. ASIP Meister automatically creates this "*meister*" directory. This meister directory is always created at the place, where ASIP Meister is started. So execute ASIP Meister inside your project directory (*browstd32* in this example)! It is very important, that you always start ASIP Meister in your current project directory. Otherwise, the other scripts will not find the meister subdirectory or even worse: they work with an old version of the meister directory. The directory "*meister*" contains three subdirectories for the simulation VHDL files (*browstd32.sim*), the synthesis VHDL files (*browstd32.syn*) and the software description (*browstd32.sw*) respectively for the project "*browstd32*". Some additional files are placed inside the meister directory itself. The most important file here is the "*browstd32.des*" file. This file contains all information that is required to create a binary file out of an assembly file, i.e. to assemble an assembly file. This file can be automatically extended with user instructions, as explained in Figure 2-4:. From the subdirectories, you will mostly need the VHDL files from "*browstd32.syn*" for simulation in ModelSim as explained in Chapter 5 and for synthesis as explained in Chapter 6. Inside the "*browstd32.sw*" directory, which is needed for creating the GCC compiler, you will mostly need the "*instruction_set.arch*" file, as explained in Chapter 8.2. This file contains the summary of all assembly instructions that the compiler will support, but sometimes this file has to be manually edited before starting the automatic compiler generation.

The **"ModelSim"** directory will contain your ModelSim project for simulating the CPU at VHDL level. ModelSim itself is explained in Chapter 5. It is important, that you start ModelSim inside this "*ModelSim*" directory, as it searches for specific files at the position where it was started. There are **five** files already placed in this directory containing the test bench and some configuration to monitor some CPU internal signals. The details (e.g. how to use these files) are explained in Chapter 5.1.

The **"ISE_Framework"** directory contains some predesigned framework files that are required for the connection between CPU, Memory, UART, and all other components. The framework consists of three types of file and all of them have to be added to the ISE project.

- The VHDL files describe how all components are connected together.
- The UCF file describes the user constraints (e.g., which I/O pins should be used and which clock frequency is requested).
- The BMM file contains a description of the memory buildup for instruction- and data-memory for the CPU. Out of this file "*..._bd.bmm*" file will be generated while implementation and this file will be used to initialize the created bitstream with the application data, as explained in Chapter 6.4.
- **IP cores** are used within the framework, e.g. memory blocks for instruction- and data-memory or FIFOs for the connection to the LCD. These IP cores are not available as VHDL source code, but instead they are available as pre-synthesized net lists. These files just have to be copied into the directory of your ISE project (no need to actually "*add*" them to the project) and then they will be used in the "*implementation*" step. The needed *.ngc files are available in "*/home/asip00/ASIPMeisterProjects/-TEMPLATE_PROJECT/ISE_Framework/IP_Cores*". Note: The files inside the IP_Cores directory have to be copied to your ISE Project Directory. It is not sufficient to copy the full IP_Cores directory!

The **“ISE_Benchmark”** directory contains some predesigned files to benchmark only your CPU more accurately as compared to “*ISE_Framework*”. The “*ISE_Benchmark*” consists of four files: “*bram_dm.ngc*”, “*brom_im.ngc*”, “*dlx_toplevel.vhd*”, and “*dlx_ toplevel.ucf*”. The first two files are BlockRAM netlist files for data and instruction memories. The VHDL file is the top level for the whole project. The UCF file specifies the timing constraints and pins location for the design (in this step, we specify only the clock and reset constraints).

Inside your **project directory** (“browstd32” in the example from Figure 2-3) are some local files that are explained in Figure 2-4:.

Filename	**Explanation**
browstd32.pdb	The ASIP Meister project file, i.e. your CPU design. If you use this filename as parameter w
env_settings	This file contains all settings for your project. Every script that you call evaluates this file, s

Figure 2-4: The Files in a Project Directory

The settings and paths for your project directory should be properly configured in the file “**env_settings**” as explained in Figure 2-5:.

Setting	**Explanation**
PROJECT_NAME = browstd32	This is the name of you
CPU_NAME = browstd32	This is the name of the
DLXSIM_DIR = /home/asip00/epp/dlxsim	This is the full path to
ISE_NAME = ISE_Framework	This is the name for yo
ASIPMEISTER_PROJECTS_ DIR = ~/ASIPMeisterProjects	This is the directory na
PROJECT_DIR = ${ASIPMEISTER_PROJECTS_DIR}/${PROJECT_NAME}	This is the full director
MEISTER_DIR = ${PROJECT_DIR}/meister	The full directory name
MODELSIM_DIR = ${PROJECT_DIR}/ModelSim	The full directory name
ISE_DIR = ${PROJECT_ DIR}/${ISE_NAME }	The full directory name
MKIMG_DIR = /home/asip00/epp/mkimg	The full directory name
COMPILER_NAME = brownie32-elf-gcc	The name of the compi
ASSEMBLER_NAME = brownie32-elf-as	The name of the assem
LINKER_NAME = brownie32-elf-ld	The name of the linker
COMPILER_DIR = ${MEISTER_DIR}/{CPU_NAME}.swgen/bin	The full directory name
PATH=${MEISTER_DIR}/${CPU_NAME}.swgen/bin:$PATH	Add the compiler binar

Figure 2-5: The Configurable Settings for an ASIP Meister Project