CES Chair for Embedded Systems

Prof. Dr. J. Henkel

Karlsruhe Institute of Technology

Fakultät für Informatik

Laboratory: "Designing Application Specific Embedded Processors"

This work uses ASIP Meister developed by PEAS Project, Osaka University.

6/29/2014

Table of contents

1 Introduction

This document was written for the participants of the students laboratory "Developing Embedded Application Specific Processors", that is given at the Chair for Embedded Systems [CES] at the University of Karlsruhe. This document assumes a tool- and environment-setup specific to this laboratory. Many parts are written with our development environment in mind and cannot be applied to other setups without change, but users of such setups can get an impression about the tool chains for specific tasks.

1.1 Application specific instruction set processors

Application Specific Instruction-set Processors (ASIPs) are a good trade-off between Application Specific Integrated Circuits (ASICs) and General Purpose Processors (GPPs). ASICs show the best performance in energy and speed, but on the other hand, they have the highest development costs and therefore are only reasonable for high volume products. Unlike ASICs, where everything is executed in hardware, GPPs execute everything in software. This makes them extremely flexible, but on the other hand, they show a bad performance in energy and speed terms, especially compared to ASICs.

ASIPs are processors with an application specific instruction set. So in contrast to the GPPs they are optimized for a specific application or for a group of applications. For this group of applications they achieve better energy and speed results than GPPs, as they have hardware support for these applications. However, contrary to ASICs they are still very flexible and can execute any kind of application, although they do not have the energy and speed benefits for other applications. The customization of ASIPs typically addresses three architectural levels that vary depending on the platform vendor [Henkel03]:

- *Instruction extension*: The designer can define customized instructions by specifying their functionality. The extensible processor platform will then generate the extended instructions that then coexist with the base instruction set.
- *Inclusion/exclusion of predefined blocks*: The designers can choose to include or exclude predefined blocks as part of the extensible processor platform. Block examples include special function registers, built-in self-test, multiply-and-accumulate operation blocks, and caches.
- *Parameterization*: The designer can fix extensible processor parameters such as instruction and data cache sizes, the number of registers, and so on.

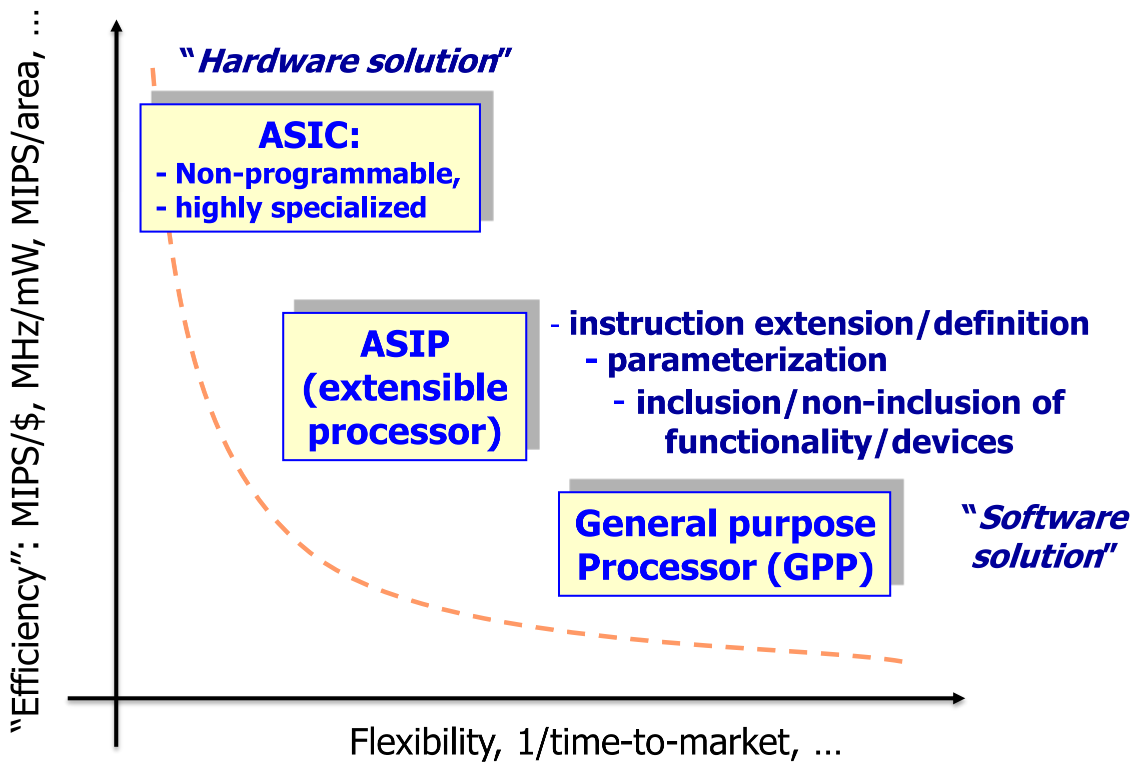

"Hardware solution"
ASIC:
- Non-programmable,
- highly specialized
ASIP (extensible processor)
- instruction extension/definition
- parameterization
- inclusion/non-inclusion of functionality/devices
General purpose Processor (GPP)
"Software solution"
"Efficiency": MIPS/$, MHz/mW, MIPS/area, …
Flexibility, 1/time-to-market, …

Figure 1-1
GPPs, ASIPs and ASICs [Henkel06]

ASIPs represent a good trade-off between Application Specific Integrated Circuits (ASICs) and General Purpose Processors (GPPs), as shown in Figure 1-1. ASICs have the highest efficiency due to the fact, that they are often manually optimized for a specific task and therefore only the necessary elements are included. This has a high impact to the power consumption and the execution speed, but it causes a high time-to-market and high development costs. Nevertheless, these development costs can amortize when selling a huge amount of units, due to the lower costs per unit. The GPPs are less efficient due to the fact, that they are usable for many different kinds of applications and therefore often contain blocks that are not needed for a certain task. Whenever an application domain changes frequently due to e.g. changing standards, then the GPPs are capable to adapt to these changes, whereas the ASIC would need to be redesigned.

1.2 ASIP design flow

Designing an ASIP typically starts with analyzing and profiling the targeted application or application domain, as shown in Figure 1-2. After this analysis, an ASIP is defined by e.g. specifying its instructions set, embedding blocks that are required to implement the instructions, and further configuring the architecture. Traditionally, the step of defining the ASIP is done manually. However, recent research activities have focused on automating this process within the range of a manually defined search space.

After the ASIP is defined, a synthesizable hardware description (for implementing the ASIP) and a tool chain (e.g. compiler, simulator, etc.) are created automatically. Using these tools and the hardware description, the ASIP is simulated and benchmarked. This might lead to a refinement of the ASIP to further optimize it or to keep the constraints, e.g. the area- or power budget. After the ASIP fulfils the requirements, a prototype (e.g. FPGA-based) is created and finally, the ASIP is taped out.

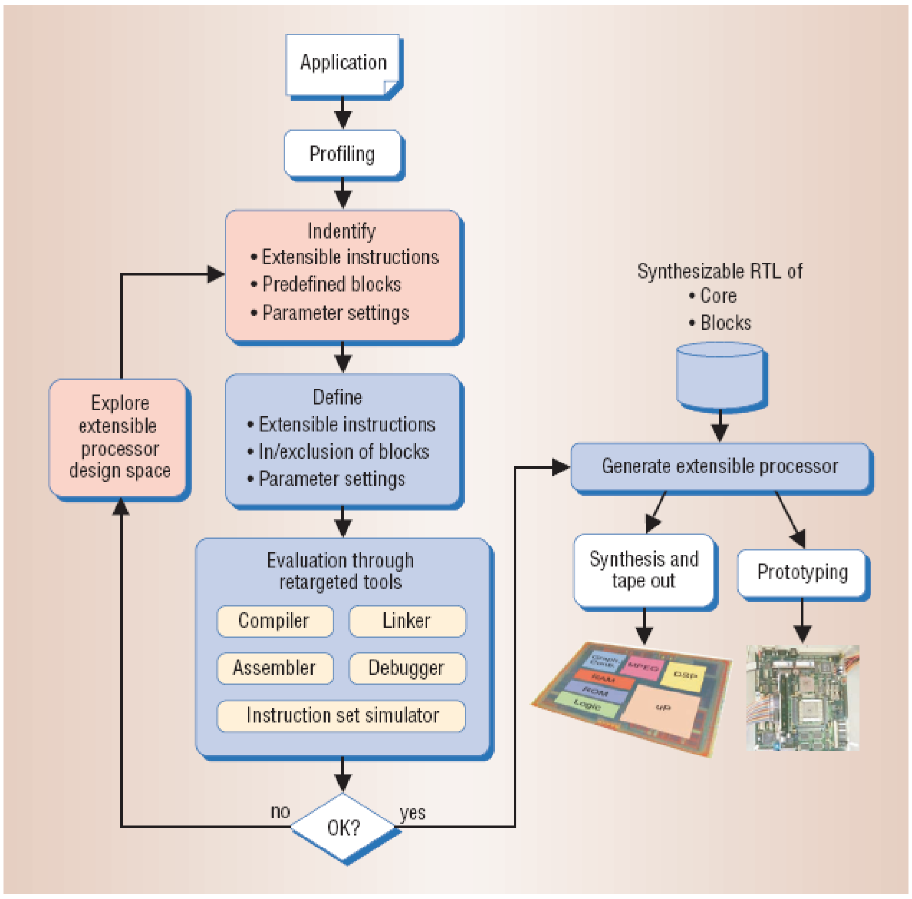


Figure 1-2
Typical ASIP Design Flow [Henkel03]

1.3 Custom Instruction identification

Custom Instructions often combine often-executed functionality in one dedicated assembler instruction. That allows executing this functionality faster and/or more efficient, etc. Generally, there are two ways to improve the performance by using Custom Instructions:

- Parallelism: Execute multiple operations that are (data-wise) independent from each other in parallel (i.e. at the same time)
- Chaining: Execute multiple operations that are (data-wise) dependent from each other sequentially (i.e. after each other) but within the same cycle. This might affect the frequency of the ASIP (to assure that all operations complete in the same cycle).

Both, parallelism and chaining can be applied together in the same Custom Instruction. Furthermore, sometimes it may also be beneficial to consider a very different implementation (compared to the given software implementation of the application) that results in the same functionality. For instance, to exchange two Bytes within the same Word, a software implementation has to use *and*, *shift*, and *or* operations. However, in hardware this corresponds to a simple re-wiring without any computation and thus can be implemented very efficient.

There are some constraints for defining Custom Instructions. A Custom Instruction is typically an unbreakable operation, i.e. it is executed either completely or not at all. Therefore, the same property has to hold for the part of the application that is considered to become a Custom Instruction. An application consist of a certain control flow and a data flow. The control flow is realized by jumps, loops, etc. The data flow instead corresponds to the input- and output dependencies of the operations. An application can be represented as a so-called Base-Block graph. A Base Block thereby represents a set of operations that are always executed together (more formally: a sequence of operations that does not contain a *jump* except at its end and where no *jump* may enter the sequence except at its beginning). A Base Block fulfills the above-mentioned requirements of an unbreakable operation and therefore is a good candidate for a Custom Instruction. However, sometimes it is possible to embed control flow within a Custom Instruction. For instance in the case of a MAX operation (i.e. determining the maximum of two values and assigning it to a result variable), both control-flow parts (the *then* and the *else* part) can be embedded inside the Custom Instruction.

There are further constraints that are important when identifying Custom Instruction. They will be explained by using the example shown in Figure 1-3. Part a) shows a given data flow within a Base Block (i.e. no further control flow limits the selection of a Custom Instruction) that is executed very frequently in an application (and thus is an interesting candidate for a Custom Instruction). Each node in this graph corresponds to a certain operation (e.g., addition) and the edges correspond to data dependencies. The first approach is to try to implement the whole data flow as one rather larger Custom Instruction. If the critical path of this data flow is too long (and thus would affect the frequency of the ASIP) the data flow can be implemented as a so-called multi-cycle operation, i.e. it may require multiple cycles to execute this instruction (the CPU pipeline is then typically stalled until the execution completes).

One typical constraint for a Custom Instruction is the amount of inputs/outputs that are read from/written to the register file. If we assume that our ASIP provides four read ports and two write ports then the shown Custom Instruction exceeds these limits. Therefore, we exclude some nodes of the data flow in part b) of the figure to fulfill the input/output requirements. Furthermore, there might be so-called forbidden nodes, i.e. nodes in the data flow that may not be part of a Custom Instruction. In our example, we have a *load* node, i.e. an access to the data memory. If we assume that our instruction executes in the EXE stage of a simple processor pipeline then it might be complicated to access the resources of the MEM stage to provide access to the data memory. Therefore, load/store nodes might be forbidden within a Custom Instruction. In part c) we therefore exclude the *load* node. However, our Custom Instruction would then no longer be convex, i.e. we have an edge leaving our Custom Instruction towards a sub graph (the single *load* node in our example) and another edge coming from this sub graph and entering our Custom Instruction. Therefore, our Custom Instruction has to be executed <u>before</u> the sub graph (to provide its input) and <u>after</u> the sub graph (to receive its output) which is obviously not possible at the same time.

a) Initial Custom Instruction
b) Reduced I/O connectivity
c) Excluded *forbidden* nodes
d) Convex graph
e) Reduced size (e.g. due to constraint for area, power, frequency, …)

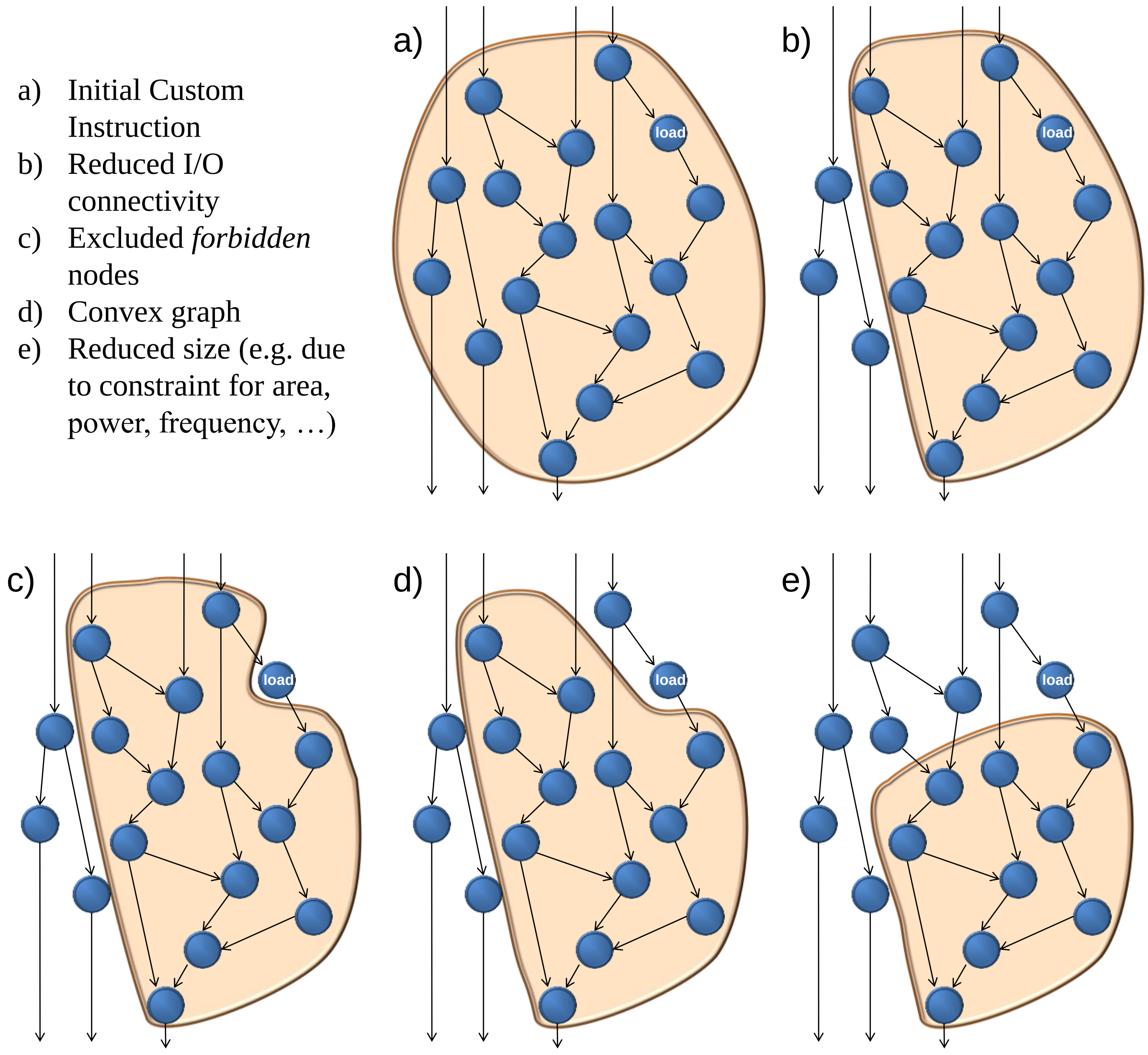


Figure 1-3
Constraints for *valid* Custom Instructions

To establish the property of a convex Custom Instruction we exclude further nodes and reach part d) of Figure 1-3. This Custom Instruction now is convex, does not contain *forbidden* nodes, and respects the maximal input/output ports of the register file. Therefore, it is a *valid* Custom Instruction. Further constraints (e.g. the maximum area, power consumption, or latency) might lead to a further reduction of the data flow graph, as shown in part e).

1.4 Goal of the laboratory

This laboratory shall teach the creation of ASIPs from the design, over the high-level simulation to the final prototype on FPGA hardware. Benchmarks of speed, needed area and power/energy consumption shall be performed and compared among different created ASIPs. For this purpose the usage of the different tools have to be practiced and the connection of those tools to form a tool chain has to be understood. The main goal is creating new ASIPs for special applications, to benchmark those ASIPs to find out their benefits and drawbacks and finally to interpret the benchmark results.

2 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 set up in a wrong way.

2.1 Network structure

You will work with the workstations in room 268. The programs needed to solve the tasks will run under Linux. Therefore, you have to use SuSE Linux, which is installed via Dual-Boot (SuSE Linux / Windows XP) on every workstation. For certain tasks, it will be necessary to change the machine in order to meet the requirements of computing power.

There is one dedicated application workstations:

- i80pc06.ira.uka.de – CoSy Compiler system:
  You will use this PC for building your own customized 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.

In most cases, you do not have to manually login into those machines or to copy data to or from those machines. We will provide scripts that will do the work for you. Just be aware of the fact that it will sometimes take time to complete everything. If some errors will occur, try to find out what is wrong by reading the script output carefully. That will show what went wrong. Start again or if it fails, again ask your tutor. The most important thing is to know or to recognize that the results of the script task are right or wrong. That 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. The names of the client PCs are similar to i80pcXX.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. 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.

2.2 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 manpages, i.e.: `man ls` calls up 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 "`.`". The parent directory is "`..`". 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 of the few he has write access to. 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 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: `./ASIPMeisterProjects/dlx_basis` which refers to the file or directory "dlx_basis" of the subdirectory ASIPMeister-

Projects of the current working directory. An example for an absolute path is `/home/asip01/test`.

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.:

   i. `cd /home/asip01`

   ii. `cd ..`

   iii. `cd ../asip02/ASIPMeisterProjects`

   iv. `cd ./dlx_basis/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). Examples:

   i. `mkdir ~/ASIPMeisterProjects/dlx_basis/Applications`

   ii. `rmdir ../dlx_basis/test1`

   iii. `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`” give a detailed listing with the last file modification time and sort by it (actually these are three parameters). Examples:

   i. `ls`

   ii. `ls -l ../ASIPMeisterProjects`

   iii. `ls -ltc ASIPMeisterProjects/dlx_basis/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. Examples:

   i. `mv Applications/tset Applications/test`

   ii. `rm /home/asip01/oldfile`

   iii. `rm -r ../dlx_basis`

i. Copying files: the `cp` command copies files and directories. Its arguments are: optional switches, then the source and the target. The "`-r`" switch enables recursive copying. Examples:

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

   ii. `cp -r dlx_basis dlx_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 truncate). Redirection is also possible to/from other processes via pipes: use "`program1 | program2`" to direct the output of program1 to the input of program2. Examples:

     i. `ls -ltc >filelisting`

     ii. `ls | sort`

     iii. `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 "`&`" to the 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 (more on that further down).

- Basic 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 regards 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 in. Once in less, 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). Examples:

   i. `less ModelSim/TestData.DM`

   ii. `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 prepended with a "-" and files that were not in the old file and are in the new file prepended 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). Examples:

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

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

d. Show only beginning/end of a file/output. The `head` and `tail` programs show only the first/last lines (respectively) of a file or 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 logfiles of a running program). This is requested with the "`-f`" option. Examples:

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

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

e. Show machine activity summary: while the "ps" command gives a detailed overview of the running processes, sometimes a summary is more useful. `top` shows an 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 operate (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.

   i. Create the DSA-keys:

      Log into any machine and invoke:

      ```
      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. 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 having to enter the password.

The complete documentation can be found in the web: http://www.schlittermann.de/ssh

2.3 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 work. On the other hand, those script files depend on this directory structure. You have to completely 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 ASIP Meister Projects including your applications and all other files will be placed in this directory. Inside this "ASIPMeister-

Projects”-folder every project has an own subdirectory (e.g. “dlx_basis” or “AnotherProject”). We recommended that you create a new subdirectory/project for each changed CPU. Among the projects the “ASIPMeisterProjects”-folder also includes a “TEMPLATE_PROJECT”. This directory gives you an empty ASIP Meister Project, where only the script files are included. 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 better to start from the scratch.

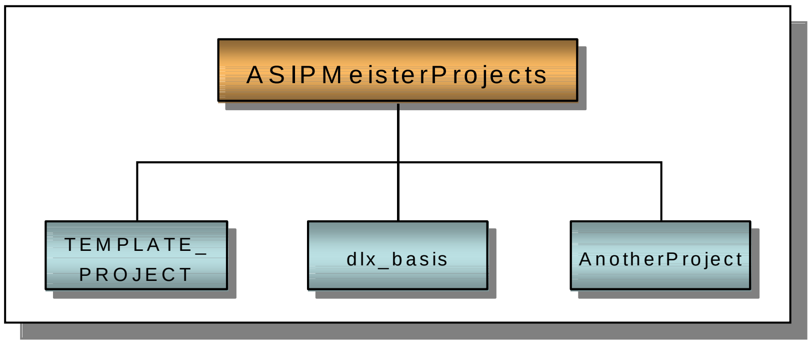


Figure 2-1
The directory structure for all ASIP Meister Projects

The biggest part of the directory structure is placed inside each ASIP Meister project directory, e.g. “dlx_basis”, as shown in Figure 2-2. Every project directory contains three subdirectories and a set of local files. Those directories and files will be explained in the remainder of this section.

The **“Applications”** directory contains all your user applications that you want to run and simulate on the CPU, and the scripts that are needed to support those tasks. Every application is placed inside a specific subdirectory for this application. When you want to work with the scripts, then you always do so from this specific subdirectory. For example, you might want to work with “Application 1”, as shown in Figure 2-2. Your first step is to compile this application. Inside the “Application 1” directory, you have provided the source code “application1.c”. To compile it you have to enter the “Application 1” directory and to execute

```
../mkimg application1.c
```

The reference “../” to access the mkimg-script is important. The script expects to be called from the place, where the results shall 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 script (mkimg_fromS, mkall, …) are explained in Chapter 8.3. The concept of calling the scripts from the specific application subdirectory is also important for the scripts dlxsim and initMem, which are explained in Chapter 3.3 and Chapter 6.3 respectively.

The scripts inside the “Applications” directory are only wrappers for the real scripts. Those wrappers first read the “env_settings” file from your project directory (e.g. “dlx_basis” in Figure 2-2). This file contains all information about your current project, e.g. which compiler to use and which dlxsim to call. Afterwards the wrapper scripts call the real scripts. The real scripts are placed at a global position. This enables us to make changes to those scripts without the need to copy those changes to all your projects. The env_settings file is explained in a later paragraph of this subchapter.

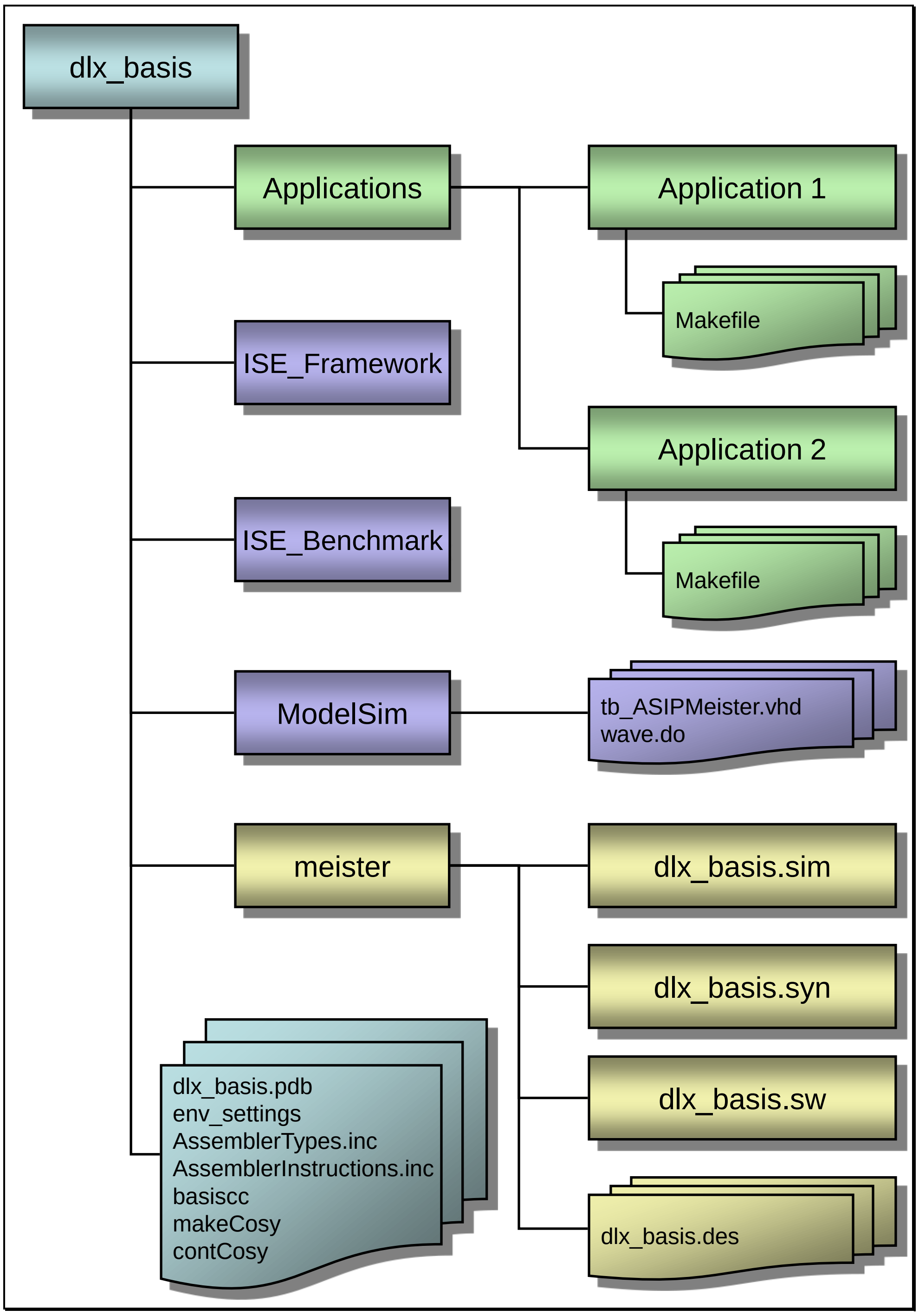


Figure 2-2
The directory structure for a specific ASIP Meister Project

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 two files already placed in this directory: the test bench and a configuration for watching some CPU internal signals. The details (e.g. how to use those files) are explained in Chapter 5.1.

ASIP Meister automatically creates the “meister” directory and this directory contains the ASIP Meister output, like the VHDL files. This meister directory is always created at the place, where ASIP Meister is started. So execute ASIP Meister inside your project directory (dlx_basis 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 meister itself directories contains three subdirectories for the simulation VHDL files, the synthesis VHDL files and the software description respectively. Some additionally files are placed inside the meister directory itself. The most important file here is the “dlx_basis.des” file. This file contains all information that is needed 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 2-3. From the subdirectories, you will mostly need the VHDL files from “dlx_basis.syn” for simulation in ModelSim as explained in Chapter 5 and for synthesis as explained in Chapter 6. Inside the “dlx_basis.sw” directory, which is needed for creating the CoSy 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 shall support, but sometimes this file has to be manually edited before starting the automatic compiler generation.

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

Filename	**Explanation**
dlx_basis.pdb	The ASIP Meister project file, i.e. your CPU design. If you use this filename as parameter when starting ASIP Meister then the design will immediately be loaded. It is important that you always start ASIP Meister inside your current project directory, as otherwise the meister directory will be created at the wrong place, i.e. at a place where the scripts don’t expect it.
env_settings	This file contains all settings for your project. Every script that you call evaluates this file, so you have to take care that the information in this file is correct. After you create a new project directory, your first task is to adapt this file. In the Figure 2-4 the entries in this file will be explained. The first 3 settings are the most important ones; the other settings will rarely be changed.
AssemblerTypes.inc / AssemblerInstructions.inc	The content of these files will be automatically added to the description file .des, which is used to explain the assembler, which instructions shall be assembled to which binary-code. Every time you call mkimg (Chapter 8.3), this extension will be performed. With these files, you can create dummy-instruction, like mapping an instruction like NOP (which is not known to the CPU itself) to another instruction, which behaves like a NOP.
basiscc	The compiler binary, which is created by “makeCoSy” and which

Filename	Explanation
	is used by "mkimg", as explained in Chapter 8.2. The name of the compiler binary might change, but it always ends with "cc".
makeCoSy	A script for automatically creating the CoSy compiler. The usage is explained in Chapter 8.2.
contCoSy	A script for continuing the creation of the CoSy compiler, as explained in Chapter 8.2. In some cases, it is necessary to perform some manual changes, if "makeCoSy" aborts with an error message. Then this script can continue the creation of the compiler after the errors have been fixed.

Figure 2-3
The files in a project directory

Setting	Explanation
PROJECT_NAME = dlx_basis	This is the name of your project directory ("dlx_basis" in Figure 2-2 or for example "AnotherProject" in Figure 2-1). Whenever you create a new project with a new directory, then you have to adapt this line.
CPU_NAME = dlx_basis	This is the name of the ASIP Meister project file ("dlx_basis.pdb" in Figure 2-2). You do not need to change this value, as we place different ASIP Meister projects in different project directories. However, if you rename the ASIP Meister project file, then you have to change this setting too, as the directory names inside the "meister" directory depend on the name of the ASIP Meister project file.
COMPILER_PREFIX = basisSinas	This is the prefix of the name from your compiler binary. The real compiler name gets a "cc" attached. The mkimg scripts will always use the compiler, that is specified by this line and the makeCoSy script will name the created binary as configured here. Therefore, you can easy switch between different compiler versions, by just changing this line.
DLXSIM_DIR = /Software/epp/dlxsim	The full directory name for the dlxsim simulator, as explained in Chapter 3.3. If you want to use a modified version of dlxsim, then you can just copy this directory into your home, make your changes and adapt this setting to use your modified version.
ISE_NAME = ISE_basis	This is the name for your ISE project where you can synthesize your CPU for the hardware platform, as explained in Chapter 6. This directory setting is used to combine the synthesis result with an application, as explained in Chapter 6.3.

Setting	Explanation
ASIPMEISTER_PROJECTS_DIR = ~/ASIPMeisterProjects	This is the directory name for all your ASIP Meister projects, e.g. "ASIPMeisterProjects" in Figure 2-1.
PROJECT_DIR = ${ASIP-MEIS-TER_PROJECTS_DIR}/${PROJECT_NAME}	This is the full directory name for your current project. You do not need to change this. The only thing you need to do is to change the settings for the project name, as mentioned above.
MEISTER_DIR = ${PROJECT_DIR}/meister	The full directory name of the "meister" subfolder. You do not have to change this value.
MODELSIM_DIR = ${PRO-JECT_DIR}/ModelSim	The full directory name of the ModelSim directory. When compiling an application, as explained in Chapter 8.3, the created binary will automatically be copied to this directory.
ISE_DIR = ${PROJECT_DIR}/${ISE_NAME }	The full directory name of the ISE project. You do not have to change this value.
MKIMG_DIR = /Software/epp/mkimg	The full directory name for the different mkimg scripts, as explained in Chapter 8.3. You do not have to change this value.
COSY_DIR = /Software/epp/CoSy	The full directory name for the scripts for the CoSy compiler generation, as explained in Chapter 8.2. You do not have to change this value.
COMPILER_NAME = ${COMPILER_PREFIX}cc	The name of the compiler binary. You do not have to change this value. To change the compiler name you can adapt the COMPILER_PREFIX setting, as explained above.

Figure 2-4
The configurable settings for an ASIP Meister project

2.4 Text editors

Reference cards for vi and emacs (or just google for e.g. "emacs reference filetype:pdf" if the links are no longer valid):

http://www.digilife.be/quickreferences/QRC/Vi%20Reference%20Card.pdf

http://inst.eecs.berkeley.edu/~cs3/fa06/emacsreference.pdf

3 Dlxsim

Dlxsim [DLX-Package] is an instruction accurate simulator for DLX assembly code. In this laboratory, we will use a modified version of dlxsim, which is changed in such a way, that it is behaving like the ASIP Meister specific implementation of the DLX Processor, which will be created and used in the later steps of the laboratory. In the first subchapter, some basic ideas about the DLX architecture and the DLX instruction set will be introduced. Afterwards the basic usage of dlxsim will be explained. In the last subchapter, it is shown how dlxsim can be extended to support new assembly instructions, which will be added to the DLX processor with ASIP Meister.

3.1 The DLX architecture

The DLX architecture [Hennessy96] is designed for an easy and fast **pipeline processor**. It is a **Load-/Store-architecture**, which means that there are dedicated commands for accessing the memory and that all the other commands only work on registers, but not on memory addresses. As an implication, the DLX architecture has a **big uniform register file** that consists of 32 registers with 32 bits each, where only the register r0 has a special meaning, as it is hard wired to zero. The DLX architecture also consists of registers for floating point operations, but as those operations are not available in our ASIP Meister implementation, they will not be discussed here any further. The pipeline stages for the DLX processor are the following:

1. Instruction Fetch (**IF**): This phase reads the command on which the program counter (PC) points from the instruction memory into the instruction register (IR) and increases the PC.
2. Instruction Decode (**ID**): Here the instruction format is determined and the respectively needed parameters are prepared; e.g. reading a register from the register file or sign extending an immediate value.
3. Execute (**EXE**): The specific operation is executed in this phase for the parameters, which have been prepared in the preceding stage.
4. Memory Access (**MEM**): If the command is a memory access, then the access will be executed in this phase. Every non-memory access command will pass this stage without any activity.
5. Write Back (**WB**): in the last stage, the result that has been computed or loaded will be written back to the register file.

An overview of the pipeline is given in Figure 3-1.

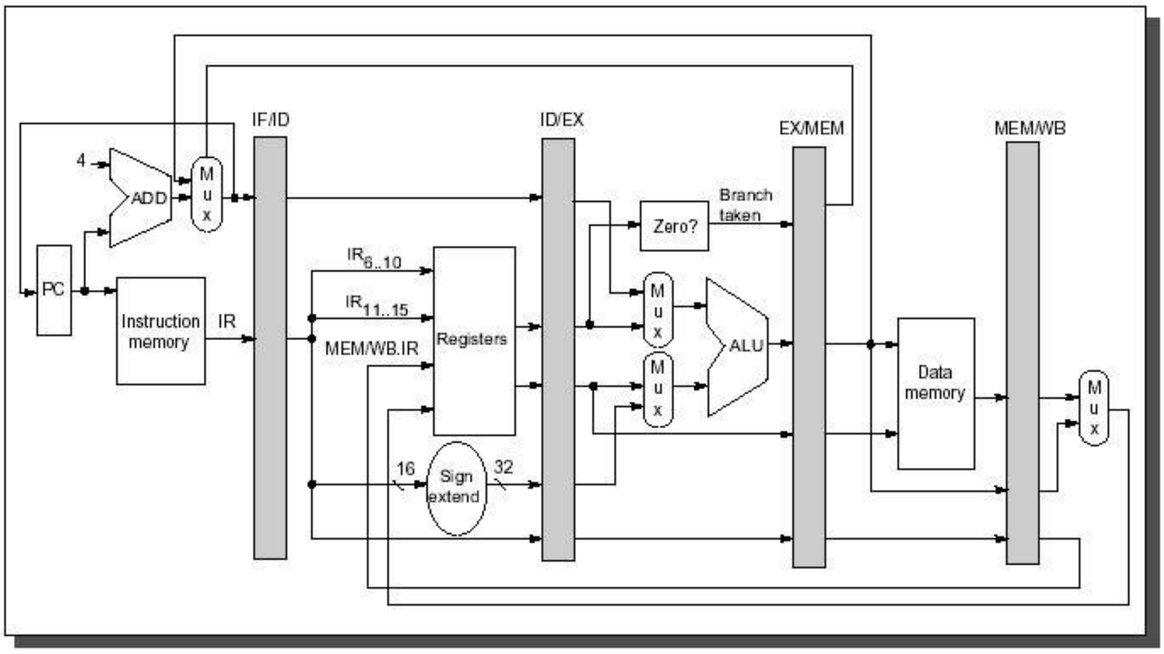


Figure 3-1
DLX-like pipeline [Shaaban01]

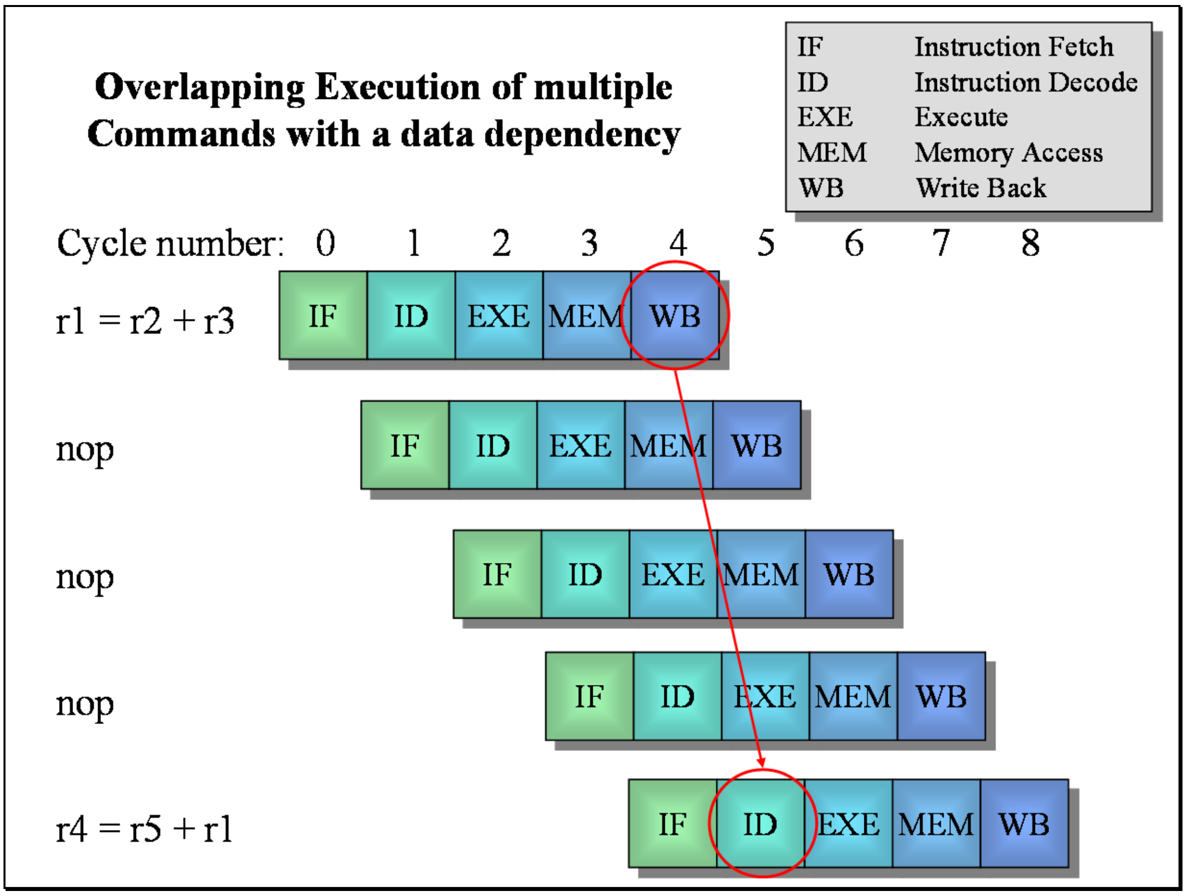


Figure 3-2
DLX pipeline example with a data dependency

In Figure 3-2 an example for some overlapping commands in the described pipeline is shown. In the first command, the values from the registers r2 and r3 are added and the result is stored into register r1. The write back to r1 is done in clock cycle 4, so it cannot be read earlier than in clock cycle 5. The last command of the shown pipeline example is using r1 as input and it is scheduled in such a way, that it is reading r1 in clock cycle 5, so that it is using the latest value that has just been written back by the first command. The example shows, that three successive NOPs are enough to resolve a data dependency. The DLX processor, as described by Hennessy and Patterson, is using a **data forwarding** technique to resolve such data dependencies without the in-between NOPs, but ASIP Meister does not support data forwarding in the current release 1.1. Therefore, for ASIP Meister generated processors the NOPs are needed to resolve the data dependencies (as shown in Chapter 3.2.1 you can configure dlxsim to behave in both ways, i.e. with or without data forwarding).

Under some special circumstances, less than three NOPs might be enough to resolve a data dependency. Such an example is shown in Figure 3-3. In this example, the multicycle instruction LW (Load Word) has been used and this instruction might force the pipeline to stall. It relies on the speed of the used data memory how long the pipeline is stalled and whether it is stalled at all. If the cache can handle a load instruction, then the pipeline might not stall at all, but if slow memory is used, then the pipeline might stall for some dozen cycles.

Figure 3-4 shows how important it is to know, when exactly the pipeline is going to stall. The only difference between Figure 3-3 and Figure 3-4 is the order of the NOP and the LW instruction. With the LW instruction as first instruction, the stall happens just in time to resolve the data dependency. With the NOP instruction first, the stall comes too late. This shows how carefully one has to deal with pipeline stalling when manually optimizing assembly code.

The **instruction set** of the DLX architecture is separated into four **instruction classes** (arithmetic for integer, arithmetic for float, load/store and branch), which are implemented in three **instruction formats**, as shown in Figure 3-5. The arithmetic instructions use either an instruction format for three registers or an instruction format for two registers and an immediate value. The load/store instructions always use the format with two registers and one immediate, where the effective address is compute as the sum of one register (as base address) and the immediate. The other register is used either as value to store in memory or as register where to place the loaded value. The branch instructions are divided into conditional branches and unconditional jumps. The jumps use an instruction format with a 26-bit immediate as a PC-relative jump-target. The *jump register* instructions instead use the I-Format to declare in which register the absolute jump target is placed. The second register and the immediate are not used by these instructions. The conditional branches need a field for a register that contains the condition, so they use the I-Format as well. They use the 16-Bit immediate as PC relative jump target. The second available register is not used.

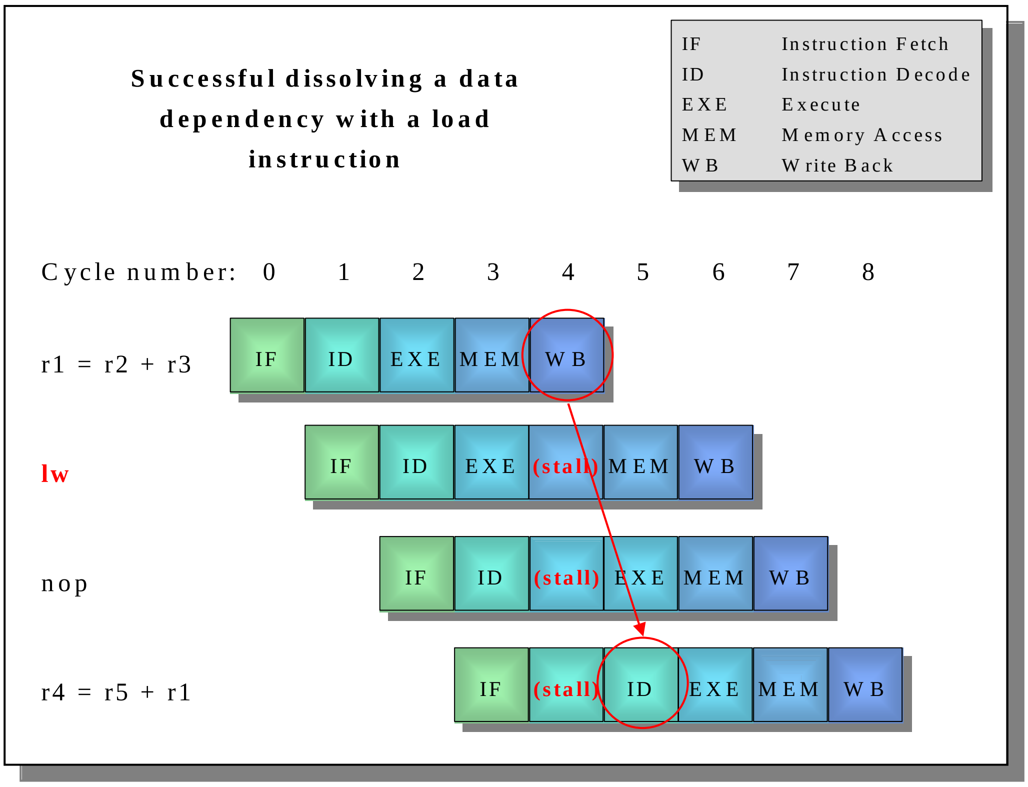


Figure 3-3
Successful dissolving a data dependency with a load instruction

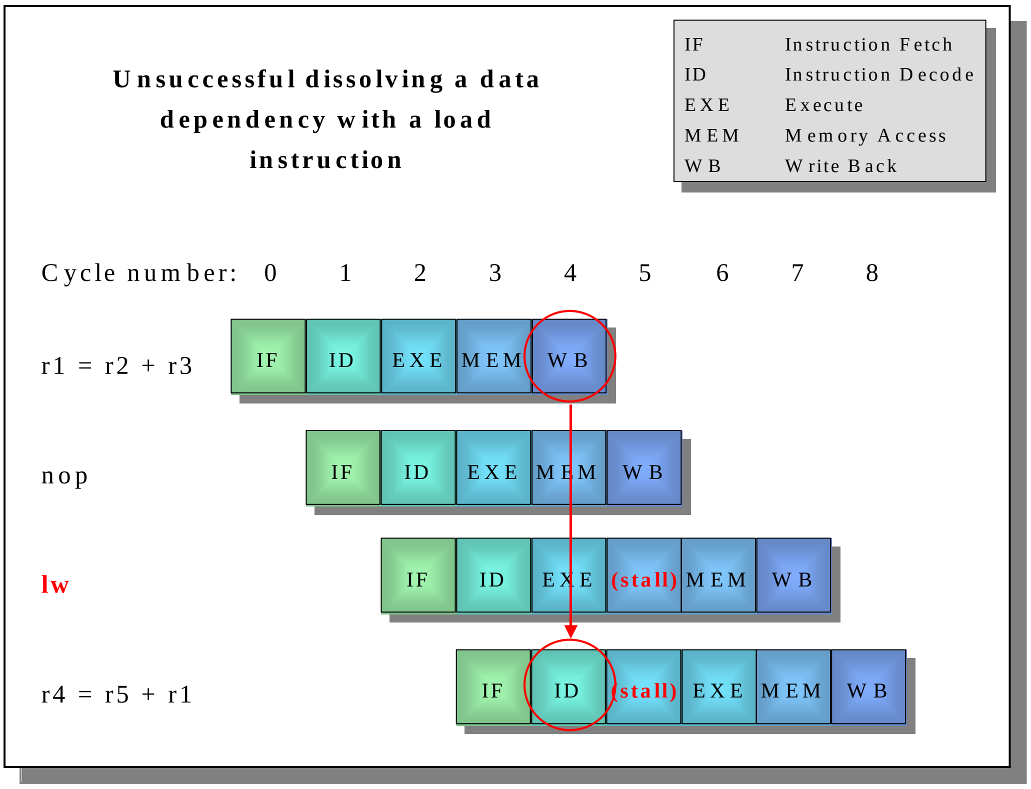


Figure 3-4
Unsuccessful dissolving a data dependency with a load instruction

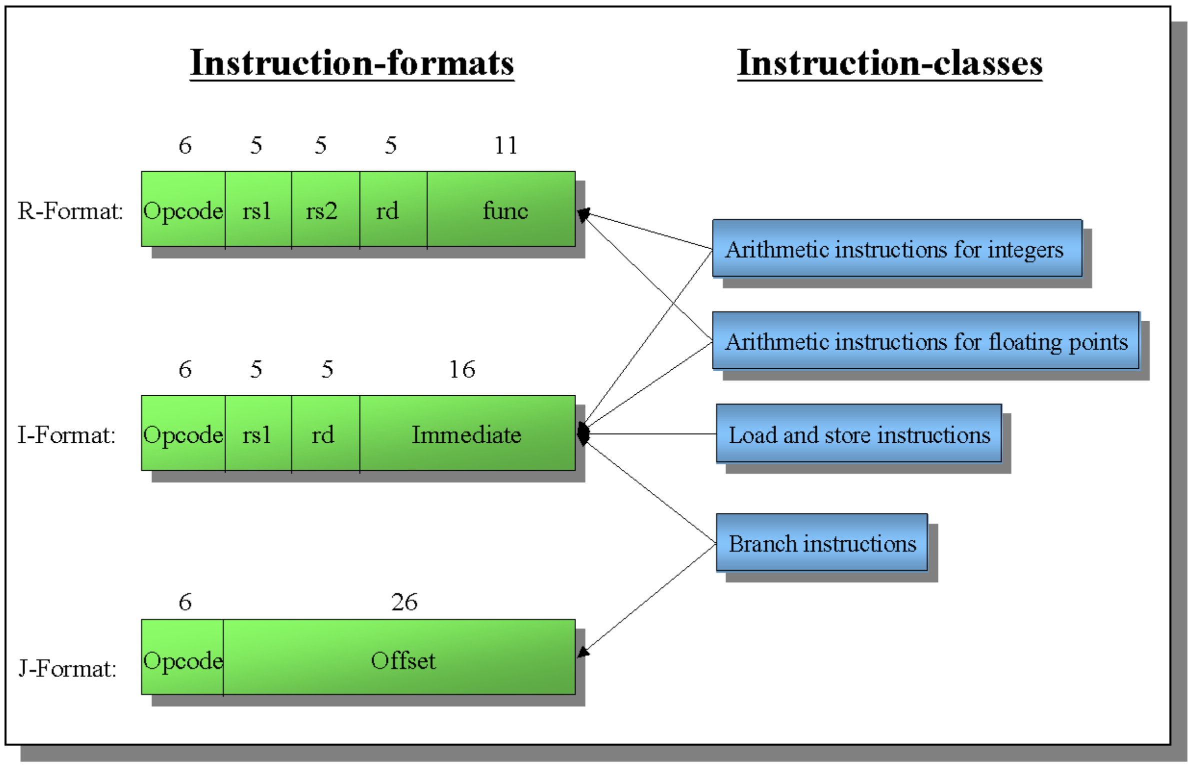

Instruction-formats
Instruction-classes
6 5 5 5 11
R-Format: Opcode rs1 rs2 rd func
6 5 5 16
I-Format: Opcode rs1 rd Immediate
6 26
J-Format: Opcode Offset
Arithmetic instructions for integers
Arithmetic instructions for floating points
Load and store instructions
Branch instructions

Figure 3-5
Instruction formats and classes of the DLX architecture

For many assembly-commands, there are special versions for dealing with unsigned values and for using immediate values as seconds input parameters. Those versions have an attached “i” for “immediate” and/or an attached “u” for “unsigned” as suffix (e.g. addui). A summary of all assembly instructions that are available in the ASIP Meister specific implementation of the DLX processor that is used in the laboratory (i.e. dlx_basis) is shown in Figure 3-6. For a more detailed description of the assembly-commands have a look into [Sailer96].

Instruction	**Description**	**Instruction**	**Description**
`add, addu, addi, addui`	Add; Syntax: `add rd rs0 rs1;` rs1 can alternatively be the immediate	`sub, subu, subi, subui`	Subtract
`mult, multu`	Multiply	`div, divu`	Divide
`mod, modu`	Modulo	`and, andi`	And
`or, ori`	Or	`xor, xori`	Xor
`sll, slli`	Shift left logical	`srl, srli`	Shift right logical
`sra, srai`	Shift right arithmetic	`slt, sltu, slti`	Set “less than”; Syntax: `slt rd rs0 rs1;` Compare rs0 and rs1 and set rd to 1 if and only if rs0 is “less than” rs1

Instruction	Description	Instruction	Description
`sgt, sgtu, sgti`	Set “greater than”	`sle, slue, slei`	Set “less or equal”
`sge, sgeu, sgei`	Set “greater or eaqual”	`seq, seqi`	Set “equal”
`sne, snei`	Set “not equal”	`lhi`	Load high immediate; Syntax: `lhi rd imm;` Load an 16-bit immediate into the 16 high bits of an register; use together with “ori” to load an 32-bit immediate
`lb, lbu`	Load Byte	`lh, lhu`	Load High
`lw`	Load Word; Syntax: `load rd, imm(rs);` the effective address is computed by adding the immediate to rs	`sb`	Store Byte; Syntax: `store imm(rd), rs;` the effective address is computed by adding the immediate to rd
`sh`	Store High	`sw`	Store Word
`beqz`	Branch if “equal zero” Syntax: `branch rs, imm;` the branch target is a PC-relative address, given by the immediate	`bnez`	Branch if “not equal zero”
`j (jr)`	Jump (Register); Syntax: `j imm (jr rs);` The jump target is a PC-relative (absolute) address, given by the immediate (register)	`jal (jalr)`	Jump and link (Register); Similar to j/jr, but also writes the address from the following command to the link register; used for subroutine calls

Figure 3-6
Summary of all assembly instructions in the dlx_basis processor

Finally, some specialties in the architecture need to be mentioned:

- **Delay slots:** An instruction, that is placed right after an unconditional jump or a conditional branch instruction is always executed. In fact, there are two instructions, that enter the pipeline, but only the first one is executed, the second one will not be allowed to write the computed result back to the register file.
- **Multicycle operations:** The operations mult, div and mod in their different versions are multicycle operations. That means, that those operations will not stay for one cycle in the execute phase, but for multiple cycles. The pipelined is stalled until the instructions finished their work in the execute stage.

- **Stalling:** The communication to the data memory is controlled by Request/Acknowledge-Signals to deal with memories of different speed. The corresponding assembly instructions might stay for multiple cycles in the memory stages, until the memory access is finished.
- **Special Registers:** Some registers in the general-purpose register file usually handle some special cases. However, contrary to the hard-wired zero-register r0 any register, as long it is done consistent in the assembly file, can handle those other special cases. Those special cases are the stack pointer (r29), the frame pointer (r30) and the link register (r31), which is then used as return address.

3.1.1 Assembler optimizations for DLX

In this chapter, we present some optimizations techniques with the help of examples. The main foundation for our optimizations is the fact, that our DLX implementation does not have a Data Forwarding Unit and therefore needs 3 cycles after a written register can be read (as motivated in Figure 3-2).

Example 1: Redundant Computations

```
addi r7, r7, $4
nop
nop
nop
lw r9, 0(r7)
```

The upper code can be transformed to:

```
addi r7, r7, $4
lw r9, 4(r7) ; r7'OLD
```

This example explicitly uses the old value of r7 (write back is not finished yet) and therefore uses the immediate field of the load instruction to recomputed the addition.

Example 2: Early register overwriting

```
sge r1, r2, r3
nop
nop
nop
beqz r1, label
nop
; if branch not taken, then exchange the content of r2 and r3
add r1, r0, r3
add r3, r0, r2
nop
nop
add r2, r0, r1
```

The upper code can be transformed to:

```
sge r1, r2, r3
nop
```

```
nop
add r1, r0, r3
beqz r1, label ; r1'OLD
nop
add r3, r0, r2
add r2, r0, r1
```

This example is overwriting the comparison result in r1 to speedup the exchange of r2 and r3 if the branch is not taken. Overwriting r1 does not affect the branch instruction, as it will use the old value of r1. It just has to be made clear that the old value of r1 is not needed in the case the branch is taken. Alternatively, any other unused register can be used instead of r1.

Example 3: Pre-Computations

```
LoopStart: r1 = compare(i, 42)
nop nop nop
if (r1 > 0) then goto LoopEnd
i--;
x += b;
nop nop nop
x *= 3;
goto LoopStart:
LoopEnd: ...
```

The upper code can be transformed to:

```
r1 = compare(i, 42);
nop nop nop
LoopStart: if (r1 > 0) then goto LoopEnd
i--;
x += b;
nop nop
r1 = compare(i, 42);
x *= 3;
goto LoopStart:
LoopEnd: ...
```

In this example, the original loop starts with a data dependency that introduces many nops to the loop. This is avoided by performing a pre-computation outside the loop and splitting the needed computation inside the loop to the beginning and the end of the loop body, which strongly reduces the number of altogether executed nops inside the loop.

3.2 Extending dlxsim

3.2.1 Startup parameters for dlxsim

Some parameters can be adapted when starting dlxsim. All mentioned parameters in Figure 3-7 do not allow blanks between a parameter and a possible value, e.g. “-pd1” instead of “-pd 1”.

Option	**Description**
-f*filename*	This parameter will load an assembly file after initialization. This parameter is equivalent to the instruction "load {filename}" in Figure 3-9.
-sf*filename*	This parameter loads a script file that can contain any command that you can type within dlxsim. These commands are then executed one after the other. For simulation automation, you can forward the output of dlxsim to a file (`../dlxsim […] > foo.txt`) and then extract the needed information out of this file (`grep "Total cycles" foo.txt`).
-dbb# (Debug Base Blocks)	This option only has an effect, if the before mentioned option "Debug Assembly" is enabled too. If both options are activated, then a register snapshot will automatically be printed at every base block start. This snapshot only includes registers, for which the value has changed since the last snapshot. By default, this option is turned off to avoid the enormous amount of output. To turn it on you have to enter "-dbb1".
-da# (Debug Assembly)	This option helps you debugging the simulated assembly code. A debugging output is printed for all load/store and jump/branch instructions, including in which cycle the message was printed and from which address it was triggered. By default, this option is turned on. To turn it off you have to enter "-da0".
-cdd# (Check Data Depend-ency)	With this option, you enable a warning message that appears when an unresolved data dependency is found in the executed assembly code. This means you will get a warning if for example, r5 is read in cycle 10, but an earlier command has made a write access to r5, that cannot be read back before cycle 12. Therefore, you would read the 'old' value. By default, this option is turned on. To turn it off you have to enter -cdd0.
-wsdo# (Warn Specific Dependency Once)	This option belongs to the before mentioned option Check Data Dependency. If there is an unresolved data dependency in a loop, then the warning message would usually be printed for every time the loop is executed. With this define the warning will only be printed the first time the unresolved data dependency is noticed. By default, this option is turned on. To turn it off you have to enter –wsdo0.

Option	Description
-pd# (Pipeline Delay)	With this option, you can configure, whether a delayed register write back shall be emulated and if yes, how long the delay between register write and register read shall be. The internal usage of this value is like this: If a value is written in cycle c, then it cannot be read back before cycle c + PipelineDelay. For the laboratory there are two useful values: With the value 1 you configure, that only the elementary delay shall be assumed. That means, that the result of an instruction can immediately be read back by the succeeding instruction. This can be seen as a pipeline model with a data-forwarding concept. Remember, that the delay-slots for branches are still used! With the value 4 you configure the pipeline model that is used for the ASIP Meister generated processor with the register read in stage 2 and the register write in stage 5. This can be seen as a pipeline model without a data-forwarding concept. Other values except 1 and 4 are basically working too, but the instructions, that imply a pipeline stall (e.g. load, mult, …) are personalized to the values 1 and 4 and might behave unexpected for different values. The default value is 4.
-ms# (Memory Size)	The size of the memory that is available within dlxsim. It is the common memory for instructions and data.
-lf*filename*	With this parameter, all print instructions for the LCD (as shown in Chapter 8.5.1) are written to a file instead of the screen.
-uf*filename*	With this parameter, all print instructions for the UART (as shown in Chapter 8.5.2) are written to a file instead of the screen.
-af*filename*	With this parameter, all outputs to the AudioOut IP-Core (as shown in Chapter 8.5.2) are written to a file instead of the screen.

Figure 3-7
Summary of helpful dlxsim starting parameters
Legend: *cursive*: replace with appropriate option
#: replace with a number

3.2.2 How to add a new instruction

The following list shows you the needed steps to add a new instruction for a given instruction-format. The given line numbers are rounded and might change during time, as the source code is partially under development. Every needed part of the source code is marked with a comment that includes the string "ASIP NEW_INSTRUCTIONS".

1. dlx.h:235 Add a new define for **OP_{CommandName}** with a unique number.
2. sim.c:365 Add the **name** for your assembly-command into the operationNames-array. The position of the name in this array has to correspond with the defined number in dlx.h from the preceding step.

3. asm.c:350 Add a new entry with your instruction name, instruction format and opcode into the opcodes-table. The **instruction name** has to be the same like in the previous step and completely written in lower-case! The already available **instruction formats** are shown in Figure 3-8. There are two different possibilities for the **opcode**. Either you use the 6 highest bits or you use the 6 lowest bits for your opcode. The unused bits always stay 0 in the opcode field. These two possibilities differ in how dlxsim is internally handling the instruction. This will become clearer in the following step. You should use the 6 highest bits for the branch instruction formats and the 6 lowest bits for the arithmetic instruction formats. Have a look at the following step to find free opcodes. The following values in the opcode table are usually not that important and might be filled out by copy-and-past from a similar instruction. Only the flags are important if you use instructions with immediate values as parameter. The flags are explained in asm.c.
4. sim.c:335 This step depends on your choice of the preceding step. You have used either the 6 highest bits or the 6 lowest bits for the opcode. If you have used the 6 highest bits, then you have to modify the **opTable** in sim.c otherwise you have to modify the **specialTable**. In both cases you replace the table entry at the position that corresponds to the chosen 6-bit opcode with your own OP_{CommandName}. So if you have chosen the opcode value 5, then you replace the 5th entry (start counting with 0) in the array with your own command.
5. sim.c:1350 Implement a new **case** for the big "switch (wordPtr→opcode)"-statement for your command. A good start is a copy-and-paste from a similar instruction. There are 2 different variables for the parameters. For example, there is a "wordPtr→rs1" and a "rs1". In "wordPtr→rs1" the number of the first source register is stored, while in "rs1" the value of this source register is stored. At the beginning of an implementation of one specific 'case' some macros like "LoadRegisterS1" are called. Those macros take care, that "rs1" is initialized with the current value of the register with the number "wordPtr→rs1".
6. Compiling To test your modified version of dlxsim you have to re-compile it. Simply type "make" in the dlxsim directory.

When the error "Unknown Opcode" occurs while loading the assembly file, then the corresponding instructions was not accepted. If you are sure, that it is no typing error in the assembly file (register names, immediate values, …) then have a look into points 1 – 4.

Instruction Format	**Parameters**
NO_ARGS	no operands
LOAD	(register, address)
STORE	(address, register)
LUI	(dest, 16-bit expression)
ARITH_2PARAM	(dest, src) OR (dest, 16-bit immediate) OR "dest" replaced by "dest/src1"

Instruction Format	Parameters
ARITH_3PARAM	(dest, src1, sr2c) OR (dest/src1, src2) OR (dest, src1, 16-bit immediate) OR (dest/src1, 16-bit immediate)
ARITH_4PARAM	(rd, rs1, rs2, rs3) OR (rd, rs1, rs2, 5-bit immediate)
BRANCH_0_OP	(label) the source register is implied
BRANCH_1_OP	(src1, label)
BRANCH_2_OP	(src1, src2, label)
JUMP	(label) OR (src1)
SRC1	(src1)
LABEL	(label)
MOVE	(dest,src1)

Figure 3-8
Summary of available dlxsim instruction formats

3.2.3 How to add a new instruction-format

Adding a new instruction format is much more difficult than adding a new instruction for a given format. To add a new format you have to take care about how the parameters for your new format are to be stored in a 32-bit instruction word and how they are extracted out of it. Every needed part of the source code is marked with a comment that includes the string "ASIP NEW_FORMAT".

1.	asm.c:140	Define a unique number for your instruction-format
2.	asm.c:150	Add the min and max numbers of arguments, that your instruction format accepts in the arrays "minArgs" and "maxArgs". Use the position in the array, that corresponds to the defined number in the previous step.
3.	asm.c:765	Implement how the parameters shall be stored in the 32-bit instruction word within the "switch (insPtr->class)"-statement.
4.	asm.c:1080	Implement what shall be written if the instruction is disassembled in the "switch (opPtr->class)"-statement.
5.	sim.c:2215	In the method "compile" you have to implement how the 32-bit instruction word shall be expanded.
6.	Compiling	To test your modified version of dlxsim you have to re-compile it. Simply type "make" in the dlxsim directory.

3.3 Using dlxsim

Dlxsim is distributed as source code, so before using it you have to compile it. Usually this is done, by just typing `make` in the tcl subfolder and afterwards in the dlxsim folder. Then you can start the program by typing `dlxsim`. If you want to use dlxsim for an ASIP Meister Project, then you call it by typing "/Software/epp/dlxsim_Laboratory/dlxsim ".

Figure 3-9 shows some typical dlxsim commands. The list is not exhaustive, but it covers all the usual suspects.

Command	**Description**
load *filename*$^+$	Load a file, parse its content, and place the translated content to the simulated memory. Every address that is not mentioned in the file remains it old value.
get *address* {*options*}	This parameter gets a value from the simulated memory or the registers. Examples for "address" are: r0, …, r31, pc, npc (next pc), 10 (memory address 10), 0x10 (memory address 16), _main (if _main is a label). Examples for "options" are: u (unsigned), d (decimal), x (hexadecimal, default), B (binary), i (instruction), v (don't read a value, but print the address itself; as example this can be used to translate from decimal to binary), s (interpret the upcoming sequence as 0-terminated string). In the options, you can also request to get the succeeding addresses from the determined base address. As an example, "get _loop 10i" would deliver the 10 first commands from the label _loop interpreted as instruction memory. This also works, if the address is a register, e.g. get r1 5u.
put *address data*	Places some data at the given address. The address might be a register too, like in the get command.
step {*number*}	Executes the "number" next instructions. The number might be omitted; the default value is 1. The printed assembly command at the end of "step" is the next assembly command that is to be executed.
go {*address*}	Executes the assembly program, until an error occurs or a trap instruction is executed. Starts executing at "address". If no "address" is given, then it continues executing where it stopped (e.g. after some single steps). If no instruction was executed yet, then the default address is 0.
stats {*options*}	Prints some statistics about the simulated assembly program. The options are explained in more detail in Chapter 3.3.1.
quit exit	Terminates the program
stop {*options*}	Manages break points: "stop at *address* {*command*}": Executes "command" whenever the given "address" is touched in any way. The default "command" is to abort the execution. "stop info": prints the list of break points. "stop delete *number*$^+$": deletes some specific break points. The numbers are according to the "stop info" list.

Command	Description
asm "*command*" {*pc*}	Returns the opcode for "*command*". If the command needs to know the address where it is to be stored (e.g. pc-relative jumps), then the current pc can be given. Default pc is 0.

Figure 3-9
Summary of typical dlxsim commands
Legend: *cursive*: replace with appropriate option
{braces}: optional
$^+$: one or multiple times

Some more things have to be mentioned about dlxsim:

- You can use the cursor buttons to navigate in your command history, i.e. in the previously entered commands. The up- and down-arrow-keys let you navigate inside the selected command, e.g. to correct typing errors.

- When you just enter an empty command in dlxsim (i.e. just press enter without having entered a command), then the last executed command will be repeated. This is for example useful, when you want to step through your code. You just have to execute the step instruction one time manually by typing "step" in the shell and afterwards you can repeatedly press enter to execute the next step instructions.

- When you press the Tabulator key, you will get an auto completion for filenames. The offered files are the files in the directory from where you started dlxsim. Although this auto completion will support the abbreviation `~` for your home directory, a load instruction with this abbreviation will fail. The same holds for loading files with the "-f" starting parameter, as shown in Figure 3-9.

- After you have simulated an assembly code, you have to restart dlxsim to simulate another assembly code. The "load" instruction will not reset everything to its default value.

- Every assembly command that accepts an immediate as parameter can also handle a label as immediate. This is especially useful for the load/store branch/jump and lhi commands.

- The ASIP Meister unit for data memory access does not accept an immediate change from a load command to a store command or vice versa. Dlxsim can handle this situation, but will print a warning to indicate, that this assembly code might produce a different result if it is simulated with the VHDL-code from an ASIP Meister CPU.

3.3.1 Statistics

There are many statistics available for the executed assembly code. You can get the statistics by typing "stats {*options*}", as shown in Figure 3-9. The different available options are shown in Figure 3-10. The different options can be combined; the default option is "all".

Option	Description
hw	Shows the memory size.
stalls	Shows the pipeline stalls (i.e. number of elapsed cycles, where the pipeline did not proceed) for the different categories.
all	Shows all before mentioned statistics in the same order as they appear in this figure.
reset	Resets all statistics to their initial values. This is useful if you want to see statistics for a specific part of the program and you want to mask the statistic results that are computed before you reach the beginning of this specific program part.
imcount imcount2 imcount3	Shows the number of executions for memory addresses. The output is organized into three columns. The first column shows how often a specific part of the instruction memory was executed. The second column shows the starting address of the specific memory part and the third column shows the size of the memory part. The different spellings of imcount (e.g. imcount2) refer to different sortings for the columns. This statistic merges neighbored memory addresses that are executed for nearly the same number into a single memory portion for which one output line is printed. The deviation to the average value is printed in the first column (e.g. "# of executions: 2 ± 1").
baseblocks	Shows the separation from the program into base blocks. This statistic is separated into 4 columns. The first one shows the start address and the reason why a base block starts there (e.g. a label, the command after a branch command, …). The second row shows the end address together with the reason why the base block ends there. The third row shows the size of the base block and the last row shows how often it was executed.

Figure 3-10
Summary of available dlxsim statistics

3.3.2 Debugging with dlxsim

This chapter assumes, that you are not only used to the assembler code and dlxsim, but that you are also used to the compiler (Chapter 8.3) and inline assembly (Chapter 8.4).

General Points:

- **Compare the results** from the CoSy compiled version and the gcc-compiled version. For the gcc compiled version you have to add printf statements for all essential variables, like:

```
#ifndef COSY
  printf("temp1: %i\n", temp1);
#endif
```

For the CoSy-compiled version, you have to make the essential variables global, for example moving the variable “temp1” from inside the main method to a global part outside the main method. All global variables will get an own label in the assembly code with an underscore before the name, e.g. the variable “temp1” will get the label “_temp1”. In dlxsim you can see the value of such global variables with the “get” instruction, e.g. “get _temp1 i”.

Debugging with dlxsim:

- Sometimes the **dlxsim simulation aborts** with an error message, e.g. when a load instruction is trying to access an address that is outside the simulated memory range. In such cases, you first have to find out, which instruction is causing this crash. With the instruction “get pc” you can see the address which is currently executed. With the instruction “get {address} i” you can see which instruction is placed at this address and at which label this instruction can be found. With the instruction “get {Address}-0x10 20i” you can see the context of this instruction.

- Getting more debugging information from dlxsim is very helpful to understand, what the assembly code is doing. Therefore, the dlxsim starting parameters “-da#” and “-dbb#” are useful. You have to replace the “#” with either a “1” to turn the option on or with a “0” to turn it off.
  - da#: Debug Assembly. This option is turned *on* by default and it will print status information on the screen for every jump/branch/load/store-instruction. You can print additional status information by adding your needed information into the sim.c of dlxsim.
  - dbb#: Debug Base Blocks. When you turn *on* this option then at every start of a base-block all changed register values will be printed. A base block is an elementary block of assembly code that is only executed sequentially. This means, that either all instructions of a base block are executed (one after the other) or none of them is executed at all. The borders of a base block (beginning/end) are jumps and labels. The simulation will create a huge amount of output on the screen if you turn this option on. Therefore it is recommended, that you copy the output to a text file for easier reading. You can automatically print everything into a text file if you start dlxsim like: `"../dlxsim -fassembly.dlxsim -dbb1 | tee output.txt"` The “tee” program will copy all output to the screen and to the file.

- Always have a look at the printed warnings when dlxsim runs the simulation. At the end of every simulation the warnings are summarized, i.e. the number of the printed warnings is shown. If the simulations aborts before its usual end this summary is not printed. To see the summary you can see them in the statistics with `"stats warnings"`.

Changing the source code to find the bug:

- When your CoSy compiled program is not running as expected then there are different possible steps to find the problem. First, you should try to locate which code is causing the problem. Therefore, you should find a minimal C-code that still causes the problem. The further steps are much easier when the created assembly code is as small as possible.

- You have to understand, what the assembly code is doing to find and understand the error. The output of a compiler is usually very difficult to read and to understand. Sometimes it is very helpful to use SINAS (Support for Inline Assembly) to make the code more readable. With SINAS, you can add your own assembler instructions or comments to the created assembly code. With SINAS, you can also force the compiler not to rearrange the created assembly code for optimizations. For example:

```
asm int addComment() {
     @[.barrier]
     ; COMMENT: Between the i- and the j-loop
}
void main() {
     for (int i=0; i<42; i++) { ... }
     dummyFunction1();
     addComment();  // SINAS
     dummyFunction2();
     for (int j=0; j<23; j++) { ... }
}
```

- This code will add the printed comment between the two loops. In addition, the ".barrier" directive will make sure, that no code moves across this barrier, i.e. the dummyFunction1 will always be completed before the addComment function and the dummyFunction2 will always be started after the addComment function.

- When you want to add many SINAS directives with different comments, then it can be very time consuming to create all the SINAS functions. Therefore, it can be helpful to use macros to create the SINAS functions. For Example:

```
#define ADD_COMMENT_NUMBER(NUMBER) \
     asm void addComment ## NUMBER() { \
          @[.barrier] \
          ; COMMENT: This is comment number NUMBER \
     }

ADD_COMMENT_NUMBER(1) // This will create the SINAS function "addCom-
ment1()"
ADD_COMMENT_NUMBER(2) // This will create the SINAS function "addCom-
ment2()"
void main() {
     functionX();
     addComment1();
     functionY();
     addComment2();
}
Mighty things can be done with those macros. Another example:
#define ADD_COMMENT_STRING (NUMBER, STRING) \
     asm void addComment ## NUMBER() { \
          @[.barrier] \
          ; COMMENT: STRING \
     }

ADD_COMMENT_STRING(3, "between X and Y")
ADD_COMMENT_STRING(4, “after Y”)
void main() {
     functionX();
     addComment3();
     functionY();
     addComment4();
}
```

4 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 /Software/epp/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 4.3 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.

4.1 Typical challenges while working with ASIP Meister

- The **register r0** is said to be 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 "**dlx_basis.sim**" directory, when simulating or synthesizing the VHDL code. There is a problem with the register file. Instead, use the "dlx_basis.syn" directory. The same VHDL-code will be used for synthesis (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 the 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 call a tutor to remove this file.

- The ASIP Meister main GUI shows an ASIP Meister logo in the bottom-right corner. Do not click on this logo. If you click on this logo, then you will not be able to save your design anymore and you will not be able to terminate ASIP Meister. If you try, then you will the following error message: *„Following windows are opened. Version information"*. To terminate ASIP Meister in such cases you have to execute "`killall java`" in a shell.

- If you manually want to write **assembly** code, then you have to use the special syntax of ASIP Meister's assembler, i.e. “pas”. Register names have to be written like “%r5”and immediate values have to be written like $42. The general structure of an assembly program is like the following:

```
.addressing   Word
.section      .text
_main:    nop
          ...
.section      .data
_array:   .data.32  42
          ...
```

- At a taken **jump** instruction, the instruction after the jump instruction will also be executed. This is the so-called delay slot, as explained in Chapter 3.1. However, in our current implemented CPU not only the first instruction, but instead the two first instructions after a jump will enter the pipeline. The first instruction will be executed and the result will be written to the register file, while the second instruction will not be executed and nothing will be written to the register file. The reason for this behavior is, that the jump target is written to the program counter in the execute stage. At the end of the execute stage the second instruction after the jump instruction has already entered the pipeline (see Figure 3-2). Instead of flushing the pipeline, the second instruction will not be allowed to stay for multiple cycles in any of its stages and it will not be allowed to write anything to the register file.

- When writing **MicroOp** code, be careful with macros. A macro that is defined as three stages 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.

4.2 Typical Problems, Error Messages and their solutions

When a new Linux is installed on a machine, the service port number to the /etc/services needs to be added.

```
# port for fhm ( ASIP Meister )
fhm   55555/tcp
```

Typical problems while using the GUI or the MicroOp etc.:

- There are some bugs in the GUI when you use copy-and-past. Often it works, but some combinations just let ASIP Meister crash. So you'd better save your project before copy-and-paste or you can directly do the copy-and-past in the human-readable ASIP Meister project file with a text editor

- When you use conditional instructions (e.g. a conditional write-back in the WB stage) then do not forget that you typically always want to write back something. For instance, a conditional write back in case a condition is true is often not, what you want. Typically (though not always) you also want to specify what shall be written to this register when the condition is wrong.

- Conditional instructions just work for Resource usages but not for e.g. signal assignments. To achieve this you have to use the additional resource MUX and you have to specify the conditions and the values that shall be written in the different cases. This also allows using more cases then just true or false. When you use a conditional Resource usage then ASIP Meister will internally create a MUX out of it, so explicitly using a MUX does not imply extra hardware.

Error Messages while Hardware Generation:

- “Error Token: wire”: Make sure, that the organization of the MicroOp description of the pipeline stages is like the following: first, the wire definitions, then at least one blank line, then instructions without further wire declarations.

- “Error Token: wire”: Make sure, that in the Instruction Definition each Instruction Format has at least one Opcode field which is set to "name", not to "binary".

- “rtgen error: code 26 : dlx_basis.ppd(254)
  syntax error: instruction: , stage: 6, line: 255,
  error token: instruction.”: This message is referring to stage 6 in a 5 stage CPU. This message appears, if you have created a CPU without a reset exception in the Instruction Definition.

Error Messages while Software Generation:

- “13
  1 or 0 required in const value”: (the value “13” is just an example) Make sure, that your signal assignments in the MicroOp Description contain space characters, like: “flag␣=␣'0';”, with ␣ standing as a placeholder for a space character.

- “The format of the instruction "slli" does not have all of elements. "none" is missed.”: (the instruction name “slli” is an example) In the Instruction Definition for an Instruction Type do not use the Field Type "Reserved" with the Field Attribute "binary".

Error Messages in Hardware OR Software Generation OR anywhere else in ASIP Meister:

- “Out of Memory”: Make sure, that your signal assignments in the MicroOp Description contain space characters, like: “flags␣=␣"0000";”, with ␣ standing as a placeholder for a space character.

- “Keyword error: "don't_care" is not name or binary”: This message can appear in the “Arch. Level Estimation”. In the Instruction Definition for an Instruction Type, do not use the Field Type "Reserved" with the Field Attribute "don’t care".

Error Messages outside ASIP Meister, but inside our tool chain

- You have to write blanks in the MicroOp description, e.g. "result␣=␣ALU.add", with ␣ standing as a placeholder for a space character. If you write it like "result=ALU.add", meistercg will complain about something like "LU.add" (the “A” is missing in this error message), while creating the CoSy compiler.

- “ERROR: IOException while reading from inputFile or writing to outputFile.
  "null"
  Aborting execution now.”: This message can appear while running the tool “ASIP_asm”, which is a part of the mkimg scripts. The reason for this message is an unusual character in the input file of ASIP_asm, i.e. the *.out file from the compiler_temp directory. To remove this unusual sign, just open the *.out file in a text editor and copy everything into a new file. This new file should no longer contain the unusual sign. A ‘diff’ will show you the line in which the sign was placed. Now you manually have to give this modified *.out file as input to ASIP_asm to receive the TestData files. Usually this happens, if you work with hand written assembly code and the unusual sign already existed in the original assembly file. Then you should this sign with the described method, like for the *.out file.

4.3 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

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 /Software/epp/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 rc 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 ASIPmeister`" 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` shall be the base directory, i.e.: `share/fhmdb/workdb/-peas/rotator.fhm` would 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 4.5 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 according 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 following lines a 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 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, 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 those 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

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

you are writing statements that span multiple lines you have to encapsulate them in high commas (“”) 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 the 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 - those 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, those 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 it 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 can 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 6.4, and Chapter 7.2. Leave the estimation data there though, as ASIP Meister will complain without them. Should you 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/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

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 Chapter 8.3), 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/dlx_yourcpu/meister/-dlx_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.

4.4 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. Should a *cancel* signal arrive, 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.

4.5 General hints about FHMs

- Do not take the code copy and paste out of 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 implementation.

- The exemplary FHM 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.

4.6 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;
```

5 ModelSim

ModelSim is a full-featured VHDL simulator. With VHDL, you can create logic designs consisting of any size. To verify functional operation you need to simulate your design to see whether it works like expected. Therefore, test benches are used. They contain so-called stimuli or test vectors, simply said: values for the input signals. Those values are applied to the input signals and then the corresponding output signals are compared with the expected values.

To simulate a specific design one has to provide two kinds of files to the ModelSim simulator: VHDL files that contain your processor design and a testbench file that creates the needed environment. ASIP Meister is creating the VHDL files for your processor design automatically and the designer does not have to take care about the VHDL implementation of the processor. We provide the testbench and you can find it in the TEMPLATE_PROJECT/ModelSim directory (directory structure is explained in Chapter 2.3). This testbench is providing a reset and a clock signal to the CPU. Furthermore, it contains simulated data- and instruction-memory for the CPU. Those simulated memories have to be initialized with memory images that are read from TestData.DM and TestData.IM, before the CPU can start executing the application. The mkimg script is creating those memory images and it automatically copies them to the ModelSim directory of your current ASIP Meister project (so you are always simulating the application that was compiled last). After the simulation of the application is finished, you will find a memory image of the final data memory (TestData.OUT) that contains the results of the application if they are stored in data memory.

5.1 Tutorial

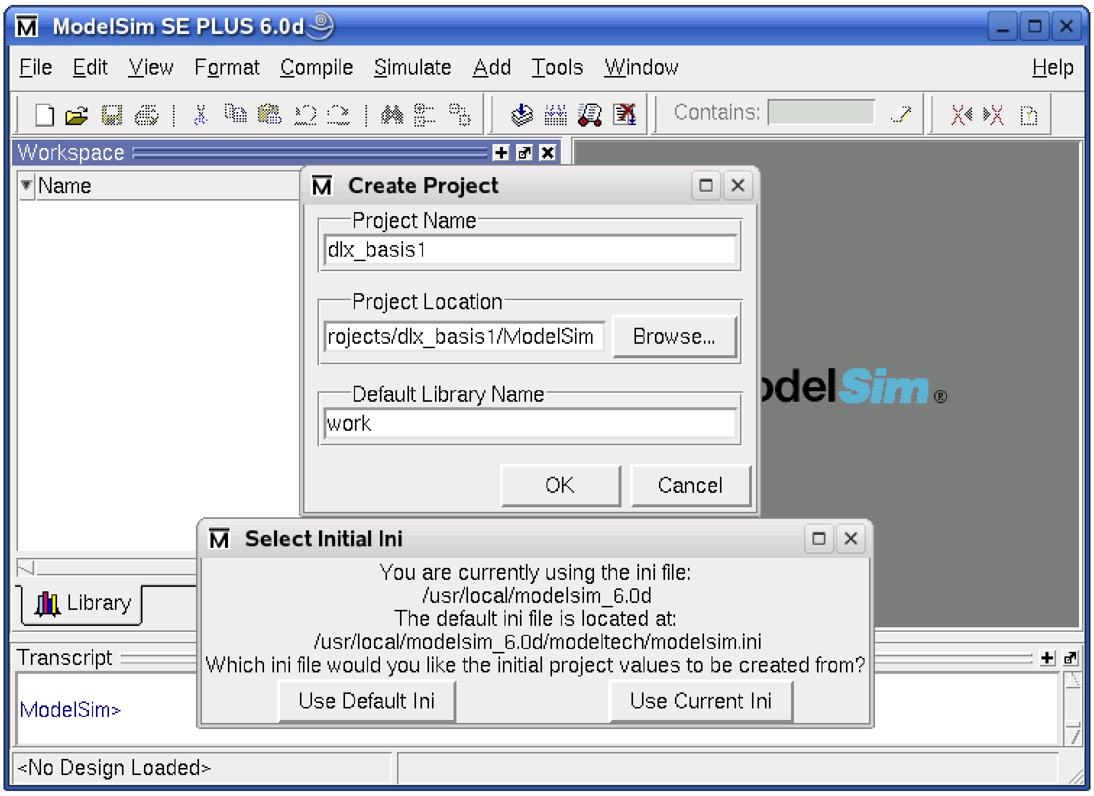
5.1.1 Create a new ModelSim Project

Please be aware that it is important that you start ModelSim in the ModelSim directory of your ASIP Meister project, as it is shown in Chapter 2.3.

On the shell change to the ModelSim directory of your project and invoke “`vsim &`”. If ModelSim asks for *modelsim.ini* choose the d*efault.ini* to proceed.

Menu: File > New > Project

Enter a project name and change the project location to the ModelSim directory in your project directory. Confirm the dialog with the OK button.



Create Project

Project Name: dlx_basis1

Project Location: rojects/dlx_basis1/ModelSim

Browse...

Default Library Name: work

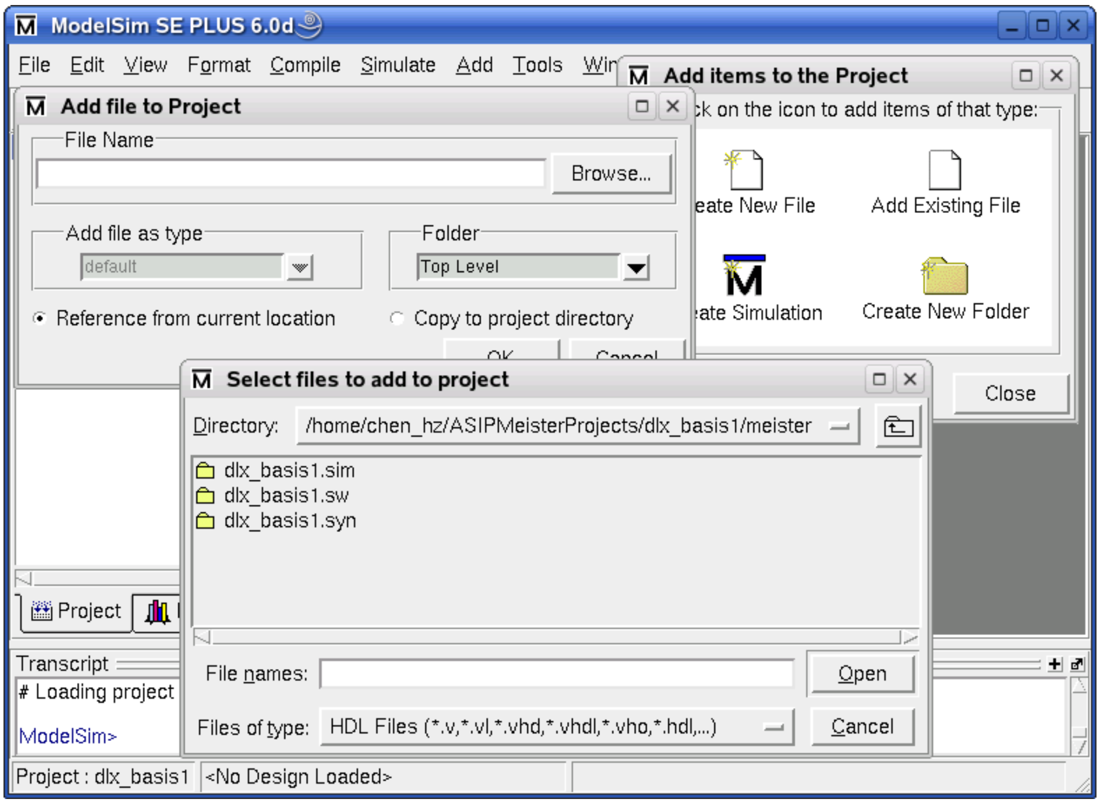
OK Cancel

Select Initial Ini

You are currently using the ini file:
/usr/local/modelsim_6.0d
The default ini file is located at:
/usr/local/modelsim_6.0d/modeltech/modelsim.ini
Which ini file would you like the initial project values to be created from?

Use Default Ini Use Current Ini

Figure 5-1
Creating a ModelSim Project



Loading project

Project : dlx_basis1 <No Design Loaded>

Add items to the Project

ck on the icon to add items of that type:

eate New File — Add Existing File — eate Simulation — Create New Folder

Close

Add file to Project

File Name:

Browse...

Add file as type: default

Folder: Top Level

Reference from current location — Copy to project directory

OK Cancel

Select files to add to project

Directory: /home/chen_hz/ASIPMeisterProjects/dlx_basis1/meister

dlx_basis1.sim
dlx_basis1.sw
dlx_basis1.syn

File names:

Open

Files of type: HDL Files (*.v,*.vl,*.vhd,*.vhdl,*.vho,*.hdl,...)

Cancel

Figure 5-2
Adding files to a ModelSim Project

5.1.2 Include the testbench and ASIP Meister CPU files

Choose the icon *Add Existing File*. Browse to the meister/dlx_basis.syn directory of your ASIP Meister project. Here you will find the VHDL files for synthesis. There is also a VHDL model of the processor in the .sim directory, but do not use the files of the .sim directory, as they do not work properly. Mark all the files and confirm the dialog with *open*. Once again, choose the icon *Add Existing File* to add the testbench files: tb_ASIPmeister.vhd, MemoryMapperTypes.vhd, MemoryMapper.vhd, and Helper.vhd from the ModelSim directory of your current project.

5.1.3 Compile the project

Menu: Compile > Compile Order > Auto Generate

Every file is compiled and you can see the result of the compile process. ModelSim determines automatically the compile order of the VHDL files. After closing the dialog every file should have a green mark behind its name, showing that the compilation was successful (instead of the question marks, shown in Figure 5-3. A green mark with a yellow dot is a warning that usually is a problem. So take care about the warnings.

IMPORTANT: When you edit your CPU in ASIP Meister and execute *HDL Generation*, then the VHDL files are regenerated and have to be recompiled. ASIP Meister might even generate new files, that have not been there in the previous set of VHDL files, e.g. for new pipeline registers that are needed for modified *MicroOp Descriptions*. These new VHDL files are not included into your ModelSim project yet. Also previously needed VHDL files might no longer be needed for a modified ASIP Meister CPU and ModelSim will complain about these files while recompiling. To avoid manually checking every VHDL file, whether it already was included into your project or whether it is a new file, you can do the following: Delete the meister/dlx_basis.syn directory before executing the *HDL Generation*; this will make sure that no files that are no longer needed exist. Remove all ASIP Meister VHDL files out of your ModelSim project and after executing the *HDL Generation* just add the new created VHDL files from the meister/dlx_basis.syn directory, like explained in the previous point. Instead of the sub window shown in Figure 5-2 when creating a new project, you can use the menu: File > Add to Project > Existing File.

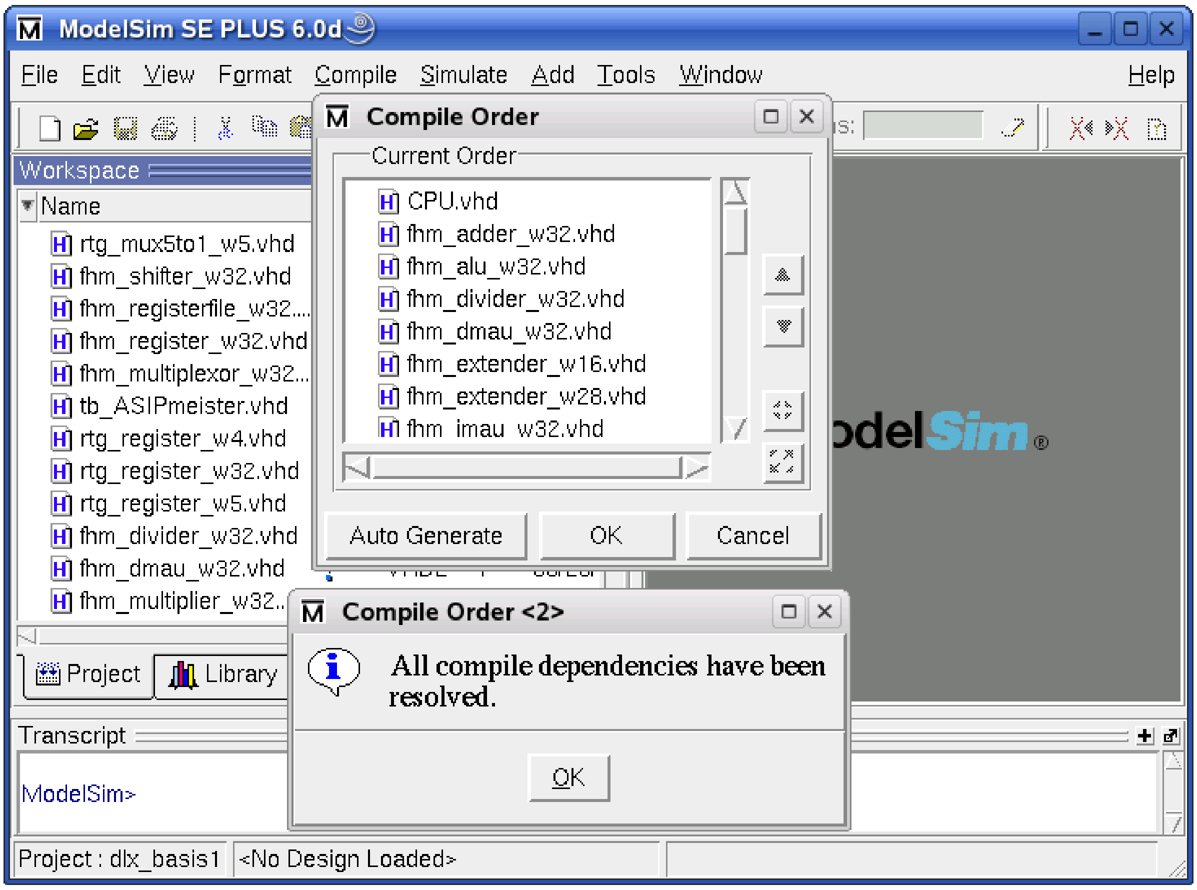
5.1.4 Run the simulation

Menu: Simulate > Start Simulation

Open the work library, mark the entry *cfg* (that is the VHDL configuration for the testbench) in the list (as shown in Figure 5-4) and press OK. That will start the simulation and you will get another two tabs attached to the Workspace window (sim / Files).

IMPORTANT: Make sure that no *Component Unbound* Message is printed while starting the simulation. If such a message is printed, then this is a serious problem within the simulation. Usually it helps to recompile everything and to start the simulation again (menu: Compile > Compile All). However, a new VHDL file, that was automatically created by ASIP Meister, but that is not included into your project yet, can also cause such a message.

Menu: Tools > Execute Macro



Name
rtg_mux5to1_w5.vhd
fhm_shifter_w32.vhd
fhm_registerfile_w32....
fhm_register_w32.vhd
fhm_multiplexor_w32...
tb_ASIPmeister.vhd
rtg_register_w4.vhd
rtg_register_w32.vhd
rtg_register_w5.vhd
fhm_divider_w32.vhd
fhm_dmau_w32.vhd
fhm_multiplier_w32...

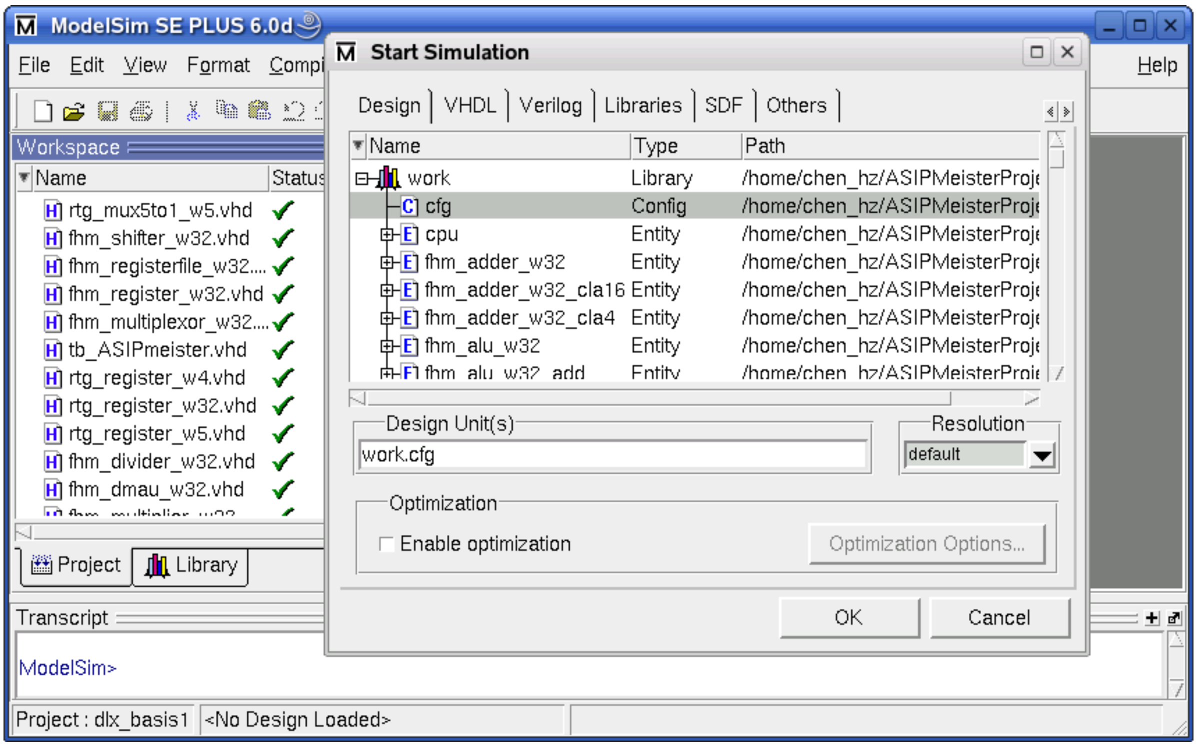
CPU.vhd
fhm_adder_w32.vhd
fhm_alu_w32.vhd
fhm_divider_w32.vhd
fhm_dmau_w32.vhd
fhm_extender_w16.vhd
fhm_extender_w28.vhd
fhm_imau_w32.vhd

All compile dependencies have been resolved.

ModelSim>

Project : dlx_basis1 <No Design Loaded>

Figure 5-3
Compiling the project



Name	Status
rtg_mux5to1_w5.vhd	✓
fhm_shifter_w32.vhd	✓
fhm_registerfile_w32....	✓
fhm_register_w32.vhd	✓
fhm_multiplexor_w32....	✓
tb_ASIPmeister.vhd	✓
rtg_register_w4.vhd	✓
rtg_register_w32.vhd	✓
rtg_register_w5.vhd	✓
fhm_divider_w32.vhd	✓
fhm_dmau_w32.vhd	✓

Name	Type	Path
work	Library	/home/chen_hz/ASIPMeisterProje
cfg	Config	/home/chen_hz/ASIPMeisterProje
cpu	Entity	/home/chen_hz/ASIPMeisterProje
fhm_adder_w32	Entity	/home/chen_hz/ASIPMeisterProje
fhm_adder_w32_cla16	Entity	/home/chen_hz/ASIPMeisterProje
fhm_adder_w32_cla4	Entity	/home/chen_hz/ASIPMeisterProje
fhm_alu_w32	Entity	/home/chen_hz/ASIPMeisterProje
fhm_alu_w32_add	Entity	/home/chen_hz/ASIPMeisterProje

Design Unit(s): work.cfg
Resolution: default
☐ Enable optimization

ModelSim>

Project : dlx_basis1 <No Design Loaded>

Figure 5-4
Starting the ModelSim simulation

Mark the *wave.do* file in your ModelSim directory and press OK to load it. The wave-window is filled with certain signals that are useful to evaluate the simulation of the program running on the processor. Those signals are explained in Chapter 5.1.5.

Press the button *Run all* to run the simulation until it aborts. At the end of a simulation the message "Failure: Simulation End" is printed. The type "Failure" is only used to automatically abort the simulation. This is not a real failure. At the simulation end, the file TestData.OUT is created in your ModelSim directory. It contains the content of the simulated memory after the CPU finished working. Therefore, if your algorithm is storing the result in the memory you can find the values here.

If you want to run another simulation with a changed program or with a changed initial data memory, then all you have to do is to run mkimg to create the new TestData.IM and TestData.DM and afterwards you have to press the buttons *restart* and *run all*, like shown in Figure 5-5.

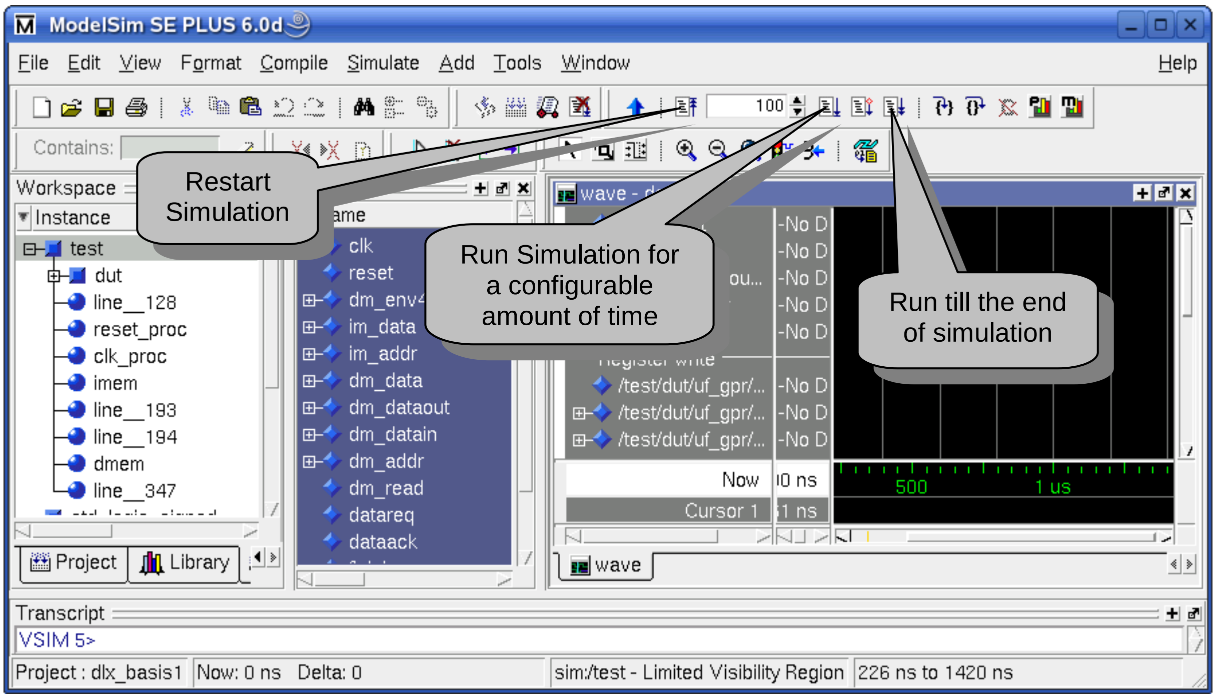


Figure 5-5
Running the ModelSim simulation

5.1.5 Statistics of the simulation

During simulation time, the testbench prints status messages about memory access (Load/Store operations) into the workspace status window. Thus, you can see which operation is being executed and which values are stored and loaded. The following examples show a read and a write access:

```
# ClockCycle:23 InstrAddr:0x00000042 DMemAddr:0x0000FFF0 --> 0 (Read)
# ClockCycle:30 InstrAddr:0x00000049 DMemAddr:0x0000FFF4 <-- 0 (Write
          32-Bit) (Old value was 45)
```

The read access is performed in cycle 23 while the currently requested instruction memory address (im-addr) is 0x42. This does not mean, that the corresponding load instruction is fetched into the pipeline in cycle 23 or that this load instruction is placed at im-addr 0x42. Instead, this means that the MEM-Phase of the corresponding load instruction is executed in cycle 23 and that the instruction at address 0x42 is fetched into the pipeline, while this load instruction is performing its MEM phase. The corresponding load instruction is usually placed some instructions before the printed im-addr, unless there was a jump in between. The loaded value is zero in the printed example and this value comes from address 0xFFF0. This is a stack operation, as the stack is starting at address 0xFFFF and growing downwards in our cases (but the starting address of the stack might be a subject of changes). The loaded value is zero in this example. The afterwards printed write-example additionally shows which value was placed in the memory location that is to be overwritten.

A more detailed kind of statistics for the simulation is the wave diagram as shown in Figure 5-6. The waves show the internal details of the VHDL model that is simulated. The waves are grouped into 5 parts, which are explained in Figure 5-7.

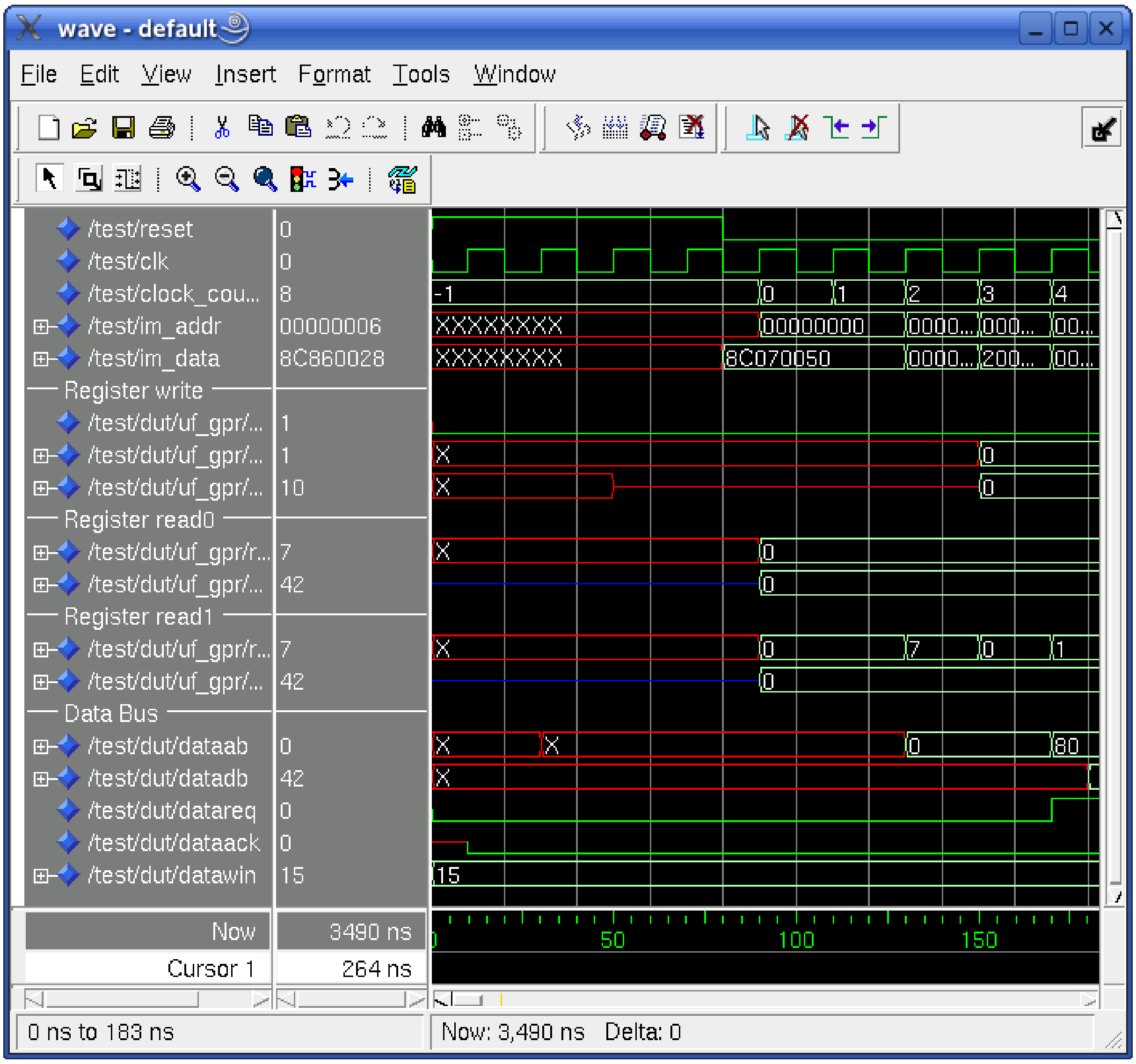


Figure 5-6
ModelSim waves

Signal	Explanation
reset	The reset signal of the CPU. This signal is active at the beginning of every simulation to initialize the CPU.
clk	The clock signal of the CPU. This signal is helpful to see, when other signal changes are really taken into the CPU, as they are only sampled at the rising edge of the clock.
clock_counter	The clock counter counts the number of executed clock cycles since the CPU started running after the initial reset.
im_addr	This is the Instruction Memory Address. It shows the address of the instruction that the CPU wants to fetch into its pipeline.
im_data	This is the Instruction Memory Data that corresponds to the previous shown im_addr. Therefore, it is the binary representation of the assembly instruction that is fetched by the CPU.
w_enb0	This is the write-enabled signal. It controls, whether the data at the input port of the register file will be written to the specified register or not. For example while jump or multicycle instructions this value is partially disabled (i.e. value '0'). Especially the jump instructions create a typical pattern for this signal. The jump instruction itself does not write to the register file, so this signal is '0' when the jump instruction comes to its WB phase. The 2 instructions after the jump instruction enter the pipeline, but only the first one will be allowed to write its result back to the register file.
w_sel0	This signal selects to which register the value at the input port of the register file shall be written.
data_in0	This is the data value that is going to be written into the register file, if the write-enabled signal is active.
r_selX	This signal selects which register of the register file shall be read.
data_outX	This is the data value that was read from the requested register.
dataab	This is the Address Bus for memory accesses. This bus either contains the address to which some data shall be written or the address from which some data shall be read.
datadb	The Data Bus contains the value that shall be written or that was read.
datareq	This is the request signal with which the CPU triggers a read or a write access.
dataack	This is the acknowledge signal, that is activated by the memory controller after a requested write access is finished or after the data bus contains the result of a requested read access.

Signal	Explanation
datawin	This signal determines the write mode. The usual values are “1111” for “read word” and “0000” for “write word”, but there are also values for writing 8- and 16-bit values.

Figure 5-7
Explanation of the signals in the ModelSim wave

To add additional signal to the Wave window (which is very helpfull for debugging) you need to enable two views in ModelSim (both are enabled by default):

Menu:	View > Workspace
Menu:	View > Debug Windows > Objects

In the Workspace you can choose which instance of you design shall be shown in the Objects windows. From the Objects window, you can then drag-and-drop signals to the Wave window.

5.2 General hints

- Change to your ModelSim directory inside your project directory tree and verify that the memory images TestData.IM and TestData.DM are present. They were built by mkimg. Furthermore, you need the testbench file tb_ASIPmeister.vhd and the ModelSim configuration script wave.do, which are available in the TEMPLATE_PROJECT/ModelSim directory. Please check their existence in your ModelSim directory before you start the work. It keeps you away from trouble.

- Always invoke ModelSim in the specific ModelSim directory of your current project by executing `vsim &` in this directory. ModelSim is working with project directories and is searching for information in the directory where it was started. After creating new projects all settings will be saved in the project file projectname.mpf (MPF = ModelSim Project file). To speed up the starting process you can invoke vsim with an option for your project file that you have created in an earlier simulation: ”`vsim projectname.mpf &`“.

- When you compile VHDL files ModelSim creates a local library in the subdirectory *work* to store the compilation results. This is the main reason why it is important to start ModelSim in the right place. It looks for last project information and for the local library.

- Sometimes you might get compiler errors from the ModelSim VHDL compiler. This is usually not the fault of ASIP Meister or the testbench. Very often, a *recompile all* solves this problem. However, sometimes you will have to create a new project from the scratch to get it working.

- If you open the waves before the simulation is started, then the signals will not be displayed. First start the simulation, and then open the waves.

6 Validating the CPU in Prototyping Hardware

In this step, we will test the CPU and application, which were generated in the previous steps, on a FPGA prototyping system. For this purpose, we will use the XUPV5-LX110T Prototyping Board from Xilinx shown in Figure 6-1.

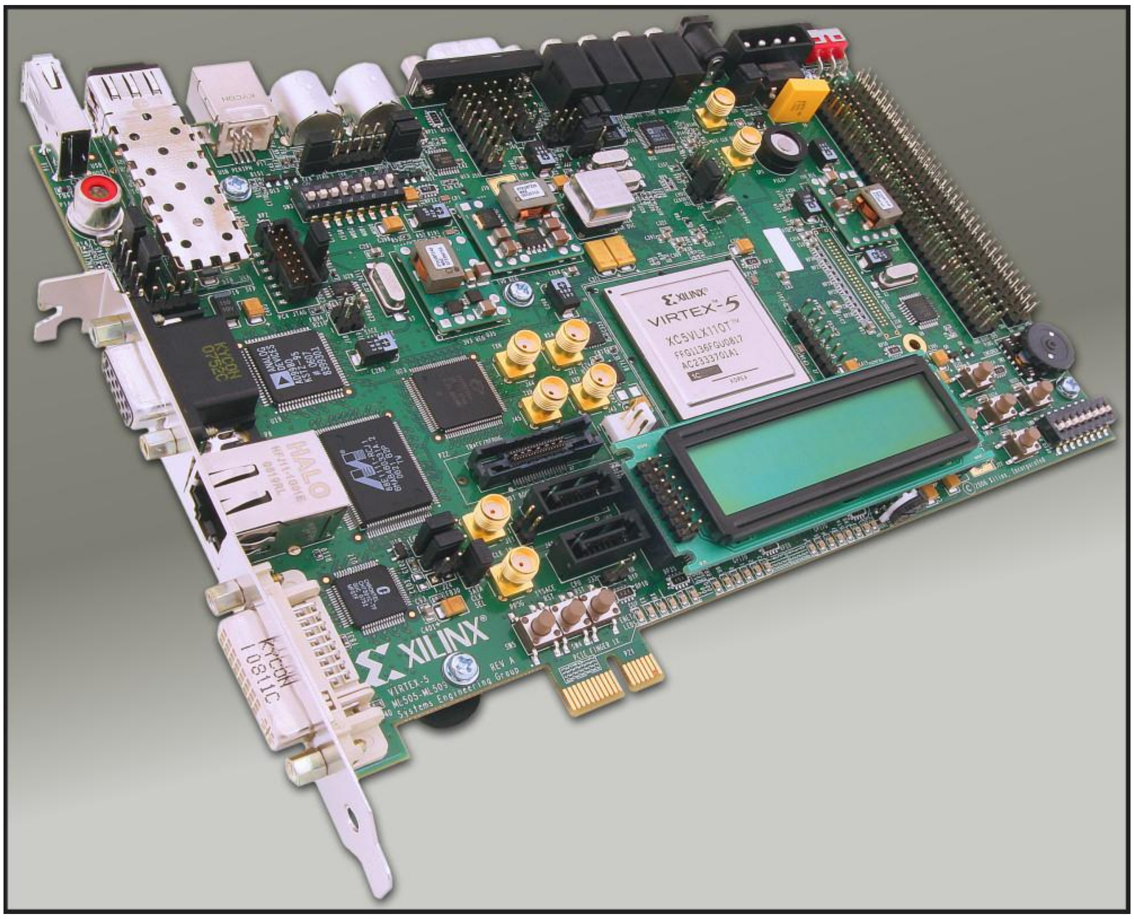


Figure 6-1
XUPV5-LX110T Prototyping Board from Xilinx [HWAFX]

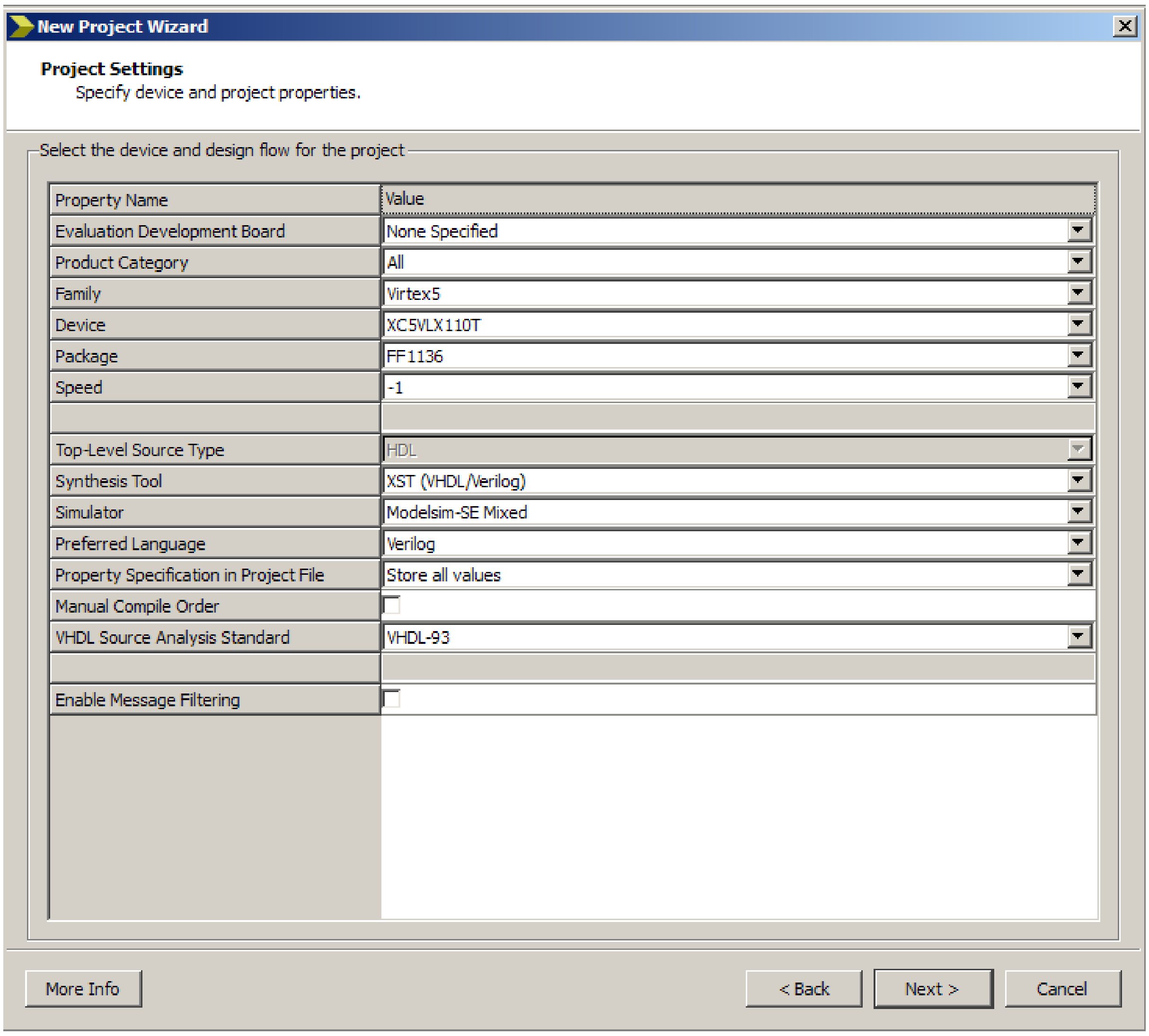
6.1 Creating the ISE Project

ISE (also called Project Navigator) is a program from Xilinx to support the whole tool-flow from managing your source files over synthesis, map, and place & route until finally uploading the design to the FPGA board.

If your local workstation is rather slow, then you should instead connect to a faster pc (e.g. i80pc86, i80pc125; all of them are placed in our server room and online 24/7) and run ISE there: `ssh -X i80pcXX`.

Start ISE by just executing `ise`

If you do not already have a project for your current CPU, then create a new one: “File” – “New Project”. As Project Location you should choose your ASIP Meister Project Directory (e.g. ASIPMeisterProjects/dlx_basis/) and as Project Name you can choose something like “ISE_...”. This Project Name will then automatically be added as new subdirectory to your chosen Project Directory. In the upcoming window “Device Properties”, you have to adjust the values to the data shown in Figure 6-2. Afterwards just press “Next” – “Next” – “Finish” to create an empty project for your CPU. The device settings are needed to make sure, that the map, place and route tools know exactly the type of the target FPGA.


New Project Wizard
Project Settings
Specify device and project properties.
Select the device and design flow for the project
More Info
< Back
Next >
Cancel

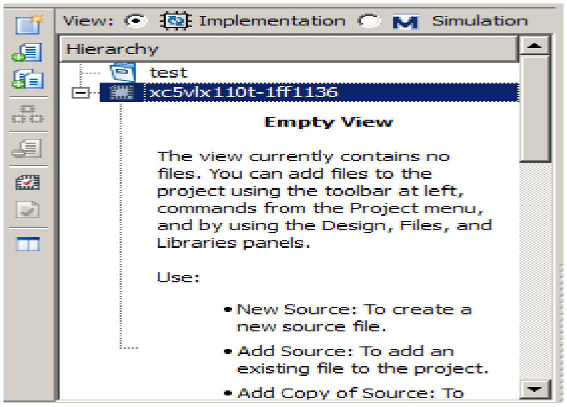
Property Name	Value
Evaluation Development Board	None Specified
Product Category	All
Family	Virtex5
Device	XC5VLX110T
Package	FF1136
Speed	-1
Top-Level Source Type	HDL
Synthesis Tool	XST (VHDL/Verilog)
Simulator	Modelsim-SE Mixed
Preferred Language	Verilog
Property Specification in Project File	Store all values
Manual Compile Order	☐
VHDL Source Analysis Standard	VHDL-93
Enable Message Filtering	☐

Figure 6-2
ISE Device Properties

Now you have to add the needed VHDL and constraint files to your ISE project, by right clicking on the XC5VLX110T-1ff1136 entry of your Sources View (at the upper left inside your ISE Window, see Figure 6-3) and choosing “Add Copy of Source”. After ISE has analyzed the type of the files, press OK. Adding a copy of the source has the advantage, that you can locally modify the files and that the CPU files in your ISE project are not overwritten, when you modify your ASIP Meister Project for testing purpose.

There are three types of files that are needed for a hardware implementation:

- CPU VHDL Files
- Framework Files
- Framework IP cores



```
test
xc5vlx110t-1ff1136
```

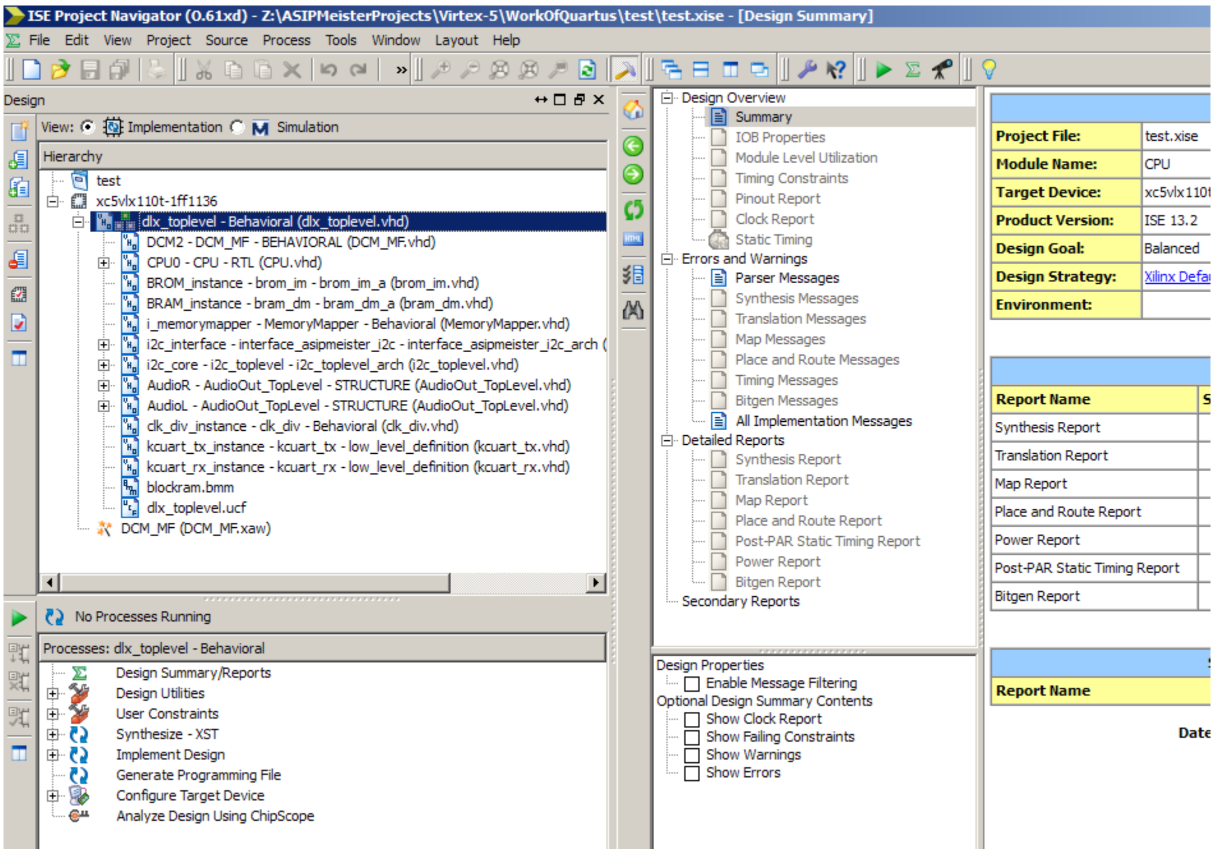
Empty View

The view currently contains no files. You can add files to the project using the toolbar at left, commands from the Project menu, and by using the Design, Files, and Libraries panels.

Use:

- New Source: To create a new source file.
- Add Source: To add an existing file to the project.
- Add Copy of Source: To

Figure 6-3
Add Sources


ISE Project Navigator (0.61xd) - Z:\ASIPMeisterProjects\Virtex-5\WorkOfQuartus\test\test.xise - [Design Summary]
File Edit View Project Source Process Tools Window Layout Help
Design
View: Implementation Simulation
Hierarchy

```
test
xc5vlx110t-1ff1136
  dlx_toplevel - Behavioral (dlx_toplevel.vhd)
    DCM2 - DCM_MF - BEHAVIORAL (DCM_MF.vhd)
    CPU0 - CPU - RTL (CPU.vhd)
    BROM_instance - brom_im - brom_im_a (brom_im.vhd)
    BRAM_instance - bram_dm - bram_dm_a (bram_dm.vhd)
    i_memorymapper - MemoryMapper - Behavioral (MemoryMapper.vhd)
    i2c_interface - interface_asipmeister_i2c - interface_asipmeister_i2c_arch (
    i2c_core - i2c_toplevel - i2c_toplevel_arch (i2c_toplevel.vhd)
    AudioR - AudioOut_TopLevel - STRUCTURE (AudioOut_TopLevel.vhd)
    AudioL - AudioOut_TopLevel - STRUCTURE (AudioOut_TopLevel.vhd)
    clk_div_instance - clk_div - Behavioral (clk_div.vhd)
    kcuart_tx_instance - kcuart_tx - low_level_definition (kcuart_tx.vhd)
    kcuart_rx_instance - kcuart_rx - low_level_definition (kcuart_rx.vhd)
    blockram.bmm
    dlx_toplevel.ucf
  DCM_MF (DCM_MF.xaw)
```

No Processes Running

Processes: dlx_toplevel - Behavioral

```
Design Summary/Reports
Design Utilities
User Constraints
Synthesize - XST
Implement Design
Generate Programming File
Configure Target Device
Analyze Design Using ChipScope
```

```
Design Overview
  Summary
  IOB Properties
  Module Level Utilization
  Timing Constraints
  Pinout Report
  Clock Report
  Static Timing
Errors and Warnings
  Parser Messages
  Synthesis Messages
  Translation Messages
  Map Messages
  Place and Route Messages
  Timing Messages
  Bitgen Messages
  All Implementation Messages
Detailed Reports
  Synthesis Report
  Translation Report
  Map Report
  Place and Route Report
  Post-PAR Static Timing Report
  Power Report
  Bitgen Report
Secondary Reports
```

Design Properties
- [ ] Enable Message Filtering

Optional Design Summary Contents
- [ ] Show Clock Report
- [ ] Show Failing Constraints
- [ ] Show Warnings
- [ ] Show Errors

Project File:	test.xise
Module Name:	CPU
Target Device:	xc5vlx110t
Product Version:	ISE 13.2
Design Goal:	Balanced
Design Strategy:	Xilinx Defa
Environment:	

Report Name	**S**
Synthesis Report	
Translation Report	
Map Report	
Place and Route Report	
Power Report	
Post-PAR Static Timing Report	
Bitgen Report	

Report Name

Date

Figure 6-4
ISE project overview

CPU VHDL Files have been generated using ASIP Meister and they can be found in the meister/{CPU-Name}.syn directory, as explained in Chapter 2.3.

Framework Files are important for the connection between CPU, Memory, UART, and all other components. They are predesigned for this laboratory and they are available in ~asip00/-

ASIPMeisterProjects/TEMPLATE_PROJECT/ISE_Framework. The framework consists of the following 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, 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 is then used to initialize the created bitstream with the application data, as explained in Chapter 6.3.

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 files (*.edn, *.ngc) are available in ~asip00/ASIPMeisterProjects/TEMPLATE_PROJECT/ISE_Framework/IP_-Cores. Note: The files <u>inside</u> the IP_Cores directory have to be copied to your ISE Project Directory. It is not sufficient to copy the full IP_Cores directory!

After you have added/copied all needed files to your ISE project/directory, your main window should look similar to the screenshot shown in Figure 6-4. In the sources sub window you can see all the source files and how they are structured, i.e. which file instantiates which other file.

6.2 Synthesizing and implementing the ISE project

In the Processes sub window in the lower left corner of Figure 6-4 the possible actions for each kind of file is shown. To synthesize and to implement your project you have to choose your VHDL-Toplevel in the Sources sub window ("dlx_Toplevel" in the figure) and after-wards you have to double click "Generate Programming File" in the Processes sub window. This will finally create the .bit file that can then be initialized with your CPU instruction- and data-memory and afterwards be uploaded to the FPGA prototyping board.

The whole process of synthesizing and implementing the design is subdivided into several steps that can be seen, if you click on the plus sign in the processes sub window. After the completion of each step, an update of the FPGA Design Summary will be shown in the corresponding sub window in the upper right corner. For example, the device utilization (i.e. the size of your CPU plus the framework) will be shown for the different types of elementary hardware available on the FPGA (e.g. clocks, logic, BRAMs). This is a first hint, how big your CPU actually is, but as it will be explained in Chapter 6.4, these values are not complete-ly accurate, as they not only include the CPU, but also include the Framework, which consists of many different components.

While synthesizing and implementing the design, many warnings will be printed. These warnings (unless created by a user modification, e.g. in the CPU) can be ignored. However, it should be mentioned, that it is very helpful to understand the meaning of these warnings when you are looking for a reason why something is working unexpected in hardware. The challenge here is, that the CPU and the IP cores create plenty of warnings, thus it is hard to locate the serious warnings.

6.3 Initializing the software and uploading to the prototyping board

After you have finished the synthesizing and implementation step, you receive a bitstream of your ISE project that includes your CPU connected to an internal memory inside the FPGA. Now you have to initialize this FPGA internal memory (called BlockRAM or BRAM) with your application instruction- and data memory to execute your program on your CPU. This initialization can be done insight the bitstream itself, i.e. before uploading the bitstream to the FPGA board.

To initialize the bitstream with your application, you need your application as TestData.IM and TestData.DM files, created with mkimg, mkimg_fromS or mkall, like explained in Chapter 8.3. However, compared to simulation with dlxsim or ModelSim you have to consider, that you have a limited amount of memory on the FPGA and therefore you have to adjust the position were the stack starts. For the usual simulation, the stack can start at address 0xFFFFC and is growing downwards. For hardware execution, this address is too big. For the current hardware prototype, you should use 0xEFFC. You can adjust the place where the stack shall start in the file "mkimgSettings". This file has to be located in the directory of your application and it is evaluated every time you call a version of mkimg or mkall. If this file does not exist, some default values are used for all settings. You can find a version of this file in ~asip00/ASIPMeisterProjects/TEMPLATE_PROJECT/Applications/TestPrint/.

The file "mkimgSettings" contains three parameters:

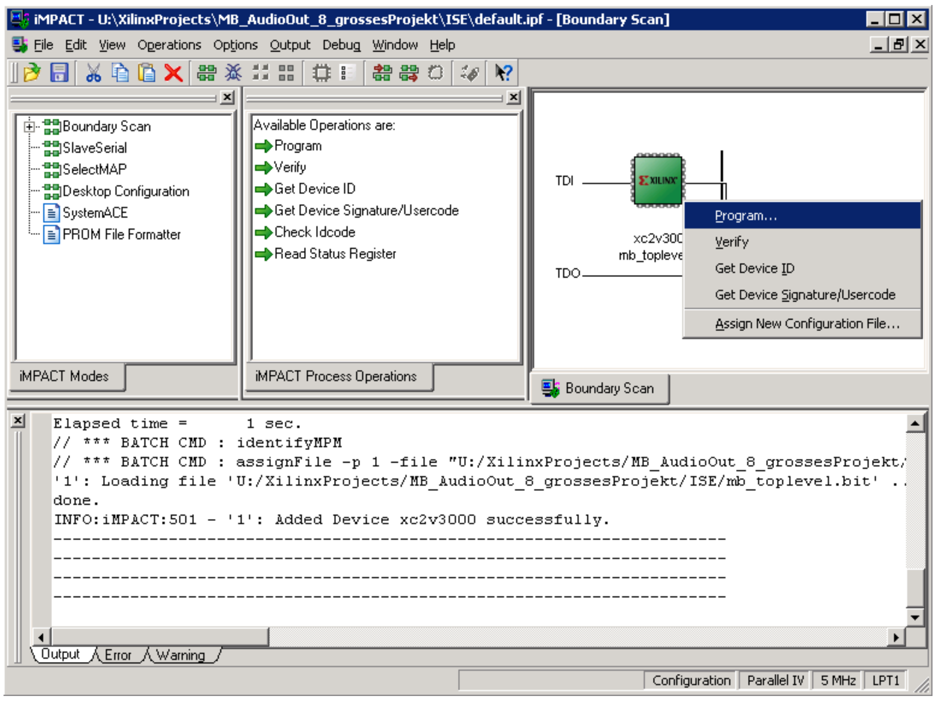
- STACK_START: Here you can configure the address of the stack start. To work in hardware this value depends of the size of the available memory in hardware. For our current prototype, "0xEFFC" is the correct value.
- ADD_NOPS: With this parameter, you can add additional NOPs between the real instructions of the application. For running in hardware, you need a value of 4 here, as our current CPU has some internal problem in some specific cases (due to some problems with the automatically created VHDL code of the CPU), which do not appear in ModelSim simulation!
- MAX_ALLOWED_SUCCESSIVE_NOPS: With this parameter, you can restrict the number of successive NOPs. Sometimes there are much more than three successive NOPs issued to the code, although three is the absolute maximum that the current pipeline structure needs for resolving pipeline dependencies, as explained in Chapter 3.1. For hardware, you have to use a value of 4 (due to some problems with the automatically created VHDL code of the CPU) and for dlxsim/ModelSim execution a value of 3 is sufficient. A value of 0 means that all NOPs are removed, except one NOP right after a Jump/Branch instruction (to fill the delay slot). This mode mimics the behavior that the CPU offers a data-forwarding unit that resolves all data dependencies. A value smaller than 0 (e.g. -1) means that the corresponding script will not be called at all and thus no NOPs are removed by the script.

The "mkimgSettings" contains a switch HARDWARE_PROFILE to easily change between the settings for FPGA prototype and dlxsim. After you have adjusted the above parameters in the file “mkimgSettings”, you have to recompile your application to let the changes take effect on the created TestData files.

After the TestData-files for your application are created (see above and Chapter 8.3) and the bitstream of your ISE project is created (see Chapter 6.1 and Chapter 6.2) you can initialize the bitstream with your application data by executing “`../initMem`” in your application subdirectory. This script will create two new files in your application subdirectory: *DirectoryName*.mem and *DirectoryName*.bit. The .mem file contains a memory dump for data and instruction memory and is only a temporary file. The bit file is the final bitstream of your specific CPU, of the surrounding framework, and with your application initialized to the BRAM. You have to configure the ISE project that you want to use in the “env_settings” as “ISE_NAME”. The bitstream in this directory will be used to create the application-initialized bitstream.

6.3.1 Turning-On the FPGA Board and uploading the Bitstream

After you have created the final bitstream with the initialized BRAM, you can upload this bitstream to the FPGA Board. At first, you have to turn the FPGA Board on.


iMPACT - U:\XilinxProjects\MB_AudioOut_8_grossesProjekt\ISE\default.ipf - [Boundary Scan]
File Edit View Operations Options Output Debug Window Help
Boundary Scan
SlaveSerial
SelectMAP
Desktop Configuration
SystemACE
PROM File Formatter
iMPACT Modes
Available Operations are:
Program
Verify
Get Device ID
Get Device Signature/Usercode
Check Idcode
Read Status Register
iMPACT Process Operations
TDI
TDO
xc2v300
mb_topleve
Program...
Verify
Get Device ID
Get Device Signature/Usercode
Assign New Configuration File...
Boundary Scan
Output
Error
Warning
Configuration
Parallel IV
5 MHz
LPT1

```
Elapsed time =      1 sec.
// *** BATCH CMD : identifyMPM
// *** BATCH CMD : assignFile -p 1 -file "U:/XilinxProjects/MB_AudioOut_8_grossesProjekt/
'1': Loading file 'U:/XilinxProjects/MB_AudioOut_8_grossesProjekt/ISE/mb_toplevel.bit' ..
done.
INFO:iMPACT:501 - '1': Added Device xc2v3000 successfully.
----------------------------------------------------------------------
----------------------------------------------------------------------
----------------------------------------------------------------------
----------------------------------------------------------------------
```

Figure 6-5
Uploading the Bitstream with iMPACT

After the FPGA Board is running, you have to connect the FPGA to the PC and then initialize it with your Bitstream. Therefore, the software iMPACT from Xilinx, shown in Figure 6-5 is used. Create a copy of the iMPACT link on your desktop. You can find it in Start menu → Xilinx ISE 8.1i → Accessories → iMPACT). If you try to start iMPACT you will receive an error message, complaining about the project-/ and the working directory (because you are no administrator on this PC). Therefore, right-click the symbol and choose ‘properties’. Then configure the working directory to “U:” (this is your mounted Linux-server home). Afterwards you can start iMPACT. Create a new project (no need to ‘save’ and ‘load’ a project) and choose the default point “Configure Device using Boundary Scan (JTAG)”. You should see a JTAG chain with one device (the FPGA) as shown in Figure 6-5. Assign the bitstream of your application subdirectory (U:\ASIPMeisterProjects\...) to the Xilinx FPGA device. To reactivate this menu point in a later run without restarting iMPACT, just right-click on the Xilinx device and choose "Assign New Configuration File". To program the Xilinx device (i.e. the FPGA) choose "Program" as shown in Figure 6-5 and confirm the dialog without any changes with "OK".

6.3.2 Initializing and using the external SRAM

The size of the BlockRAM (i.e. FPGA-internal RAM; at most 192 KByte for our FPGA, but something is used in the Framework for FIFOs etc.) is limited. This is especially problematic if a huge amount of input-data has to be provided to the application. Therefore, we have provided external SRAM to the Board, altogether four MB SRAM for IM and four MB SRAM for DM respectively. Our provided ISE Framework (see Chapter 6.1) provides the connection between the CPU and the SRAM. However, before the SRAM can be used it has to be initialized. Currently the SRAM initialization is unexpected slow. Therefore, whatever you want to test on the FPGA board, test a BlockRAM version first. The provided ISE Framework still provides the connection to the BlockRAM additionally. You just have to follow the above tutorials (Chapter 6.3 and 6.3.1) but do not forget to configure the StackStart (in the mkimgSettings) to 0xEFFC. To use the SRAM, follow the following tutorial:

- You have to compile the bootloader.c application (you can find it in ~asip00/-ASIPMeisterProjects/TEMPLATE_PROJECT/Applications/Bootloader/) with mkall and initMem. The resulting bit file contains the bootloader application in the FPGA internal BlockRAM.

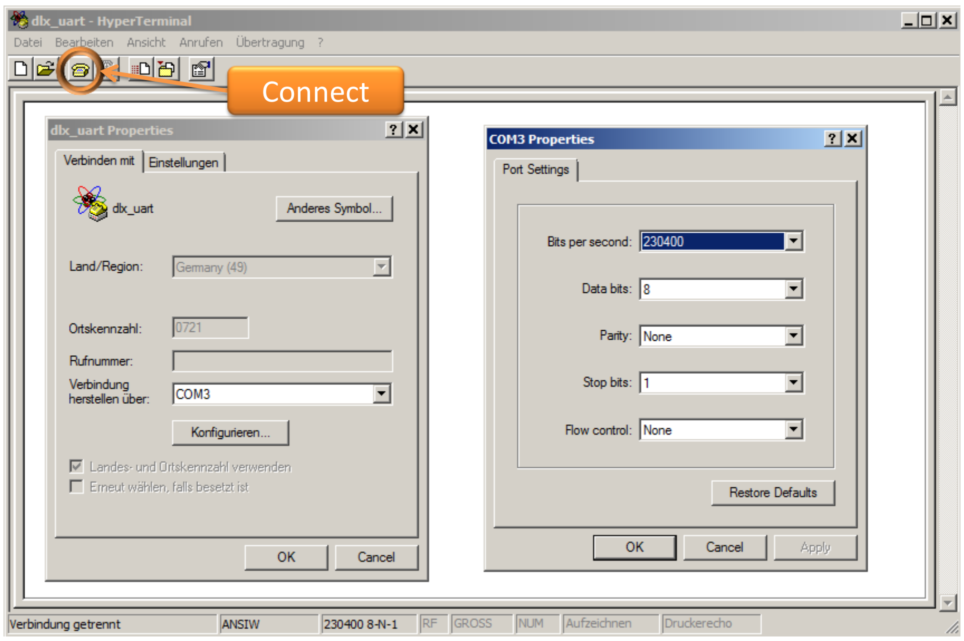
- You have to compile the user application (i.e. the one that you actually want to run from the SRAM) with the StackStart configured to 0xFFFFC (in mkimgSettings; in-stead of 0xEFFC for BRAM). Besides the normal TestData.IM and DM additionally the file TestData.IM_uart.txt and TestData.DM_uart.txt are created.

- You have to start the "dlx_uart.ht" file under windows, which will open the MS Windows HyperTerminal with the correct settings (you can find it in ~asip00/-ASIPMeisterProjects/TEMPLATE_PROJECT/Applications/Bootloader/). However, you may have to adapt the configured COM port, depending on which port you are connected to the FPGA (see Figure 6-8). After that, you have to click the 'connect' button to open the connection. Opening the connection always works without error message, but you have to make sure that the FPGA board is connected to the PC where you are running the HyperTerminal via UART.

- Now configure the FPGA Board to run with 40 MHz (see point (9) in Figure 6-6) and upload the bootloader bit-file (created in first step) with iMPACT. The UART port on the FPGA is configured to work with 40 MHz and if you use a faster or slower frequency, then you will not see the correct output via UART. The bootloader will prompt on the UART-Console for initializing the SRAM. It will ask for three things (you have to answer with pressing 'y' or 'n'): Initialize IM, Initialize DM, and Start Application.

- To initialize IM or DM you have to select 'y' and then use the HyperTerminal menu "Übertragung" → "Textdatei senden" to upload the file TestData.IM_uart or TestData.DM_uart.

- If you do not start the application in the IM-SRAM (pressing 'n' when asked) OR when the IM-SRAM application is finished OR when you press the reset button THEN you will come back to the bootloader.



dlx_uart

Land/Region: Germany (49)

Ortskennzahl: 0721

Rufnummer:

Verbindung herstellen über: COM3

☑ Landes- und Ortskennzahl verwenden

☐ Erneut wählen, falls besetzt ist

Bits per second: 230400

Data bits: 8

Parity: None

Stop bits: 1

Flow control: None

Verbindung getrennt | ANSIW | 230400 8-N-1 | RF | GROSS | NUM | Aufzeichnen | Druckerecho

Figure 6-5
HyperTerminal settings

6.3.3 Hardware specific limitations of the application

Many kinds of applications can be simulated with dlxsim or ModelSim, but to be able to be executed in hardware, there are some further limitations, which have to be considered. The most obvious point is the limitation of the available memory for instructions and data on the hardware prototype. Currently there are 16 KB for instruction and 16 KB for data memory available. A part of the initMem script will test, whether the your application binary including program and static data memory fits into these memory portions, but for dynamic requested data memory, like the stack which is growing with the number of nested function calls cannot be tested at compile time. If you need more memory, you have to use the SRAM instead (see Chapter 6.3.2).

Another point is that there are more NOP instructions needed for hardware execution than for simulation. Therefore the "mkimgSettings" file can be adjusted, like explained in Chapter 6.3.

The current connection to the data memory does only support word access to the data memory. Thus, the only supported memory access assembly instructions are lw and sw. The assembly instructions lb, lbu, lh, lhu actually work as well, because the load a full word from the memory and extract the required part of it inside the CPU. However, the instructions sb (store byte) and sh (store half word) will not work in the current hardware prototype and thus have to be avoided! A part inside the mkimg script will test, whether some of these unsupported instructions are used and it will generate a warning that this application will not run in the current hardware prototype (though it works fine in dlxsim and ModelSim). As workaround for accessing bytes (e.g. for strings to be printed on the LCD) some special functions are provided in the StdLib directory (see Chapter 8.5):

```
int storeByte(char* address, int value);
int storeShort(char* address, int value);
```

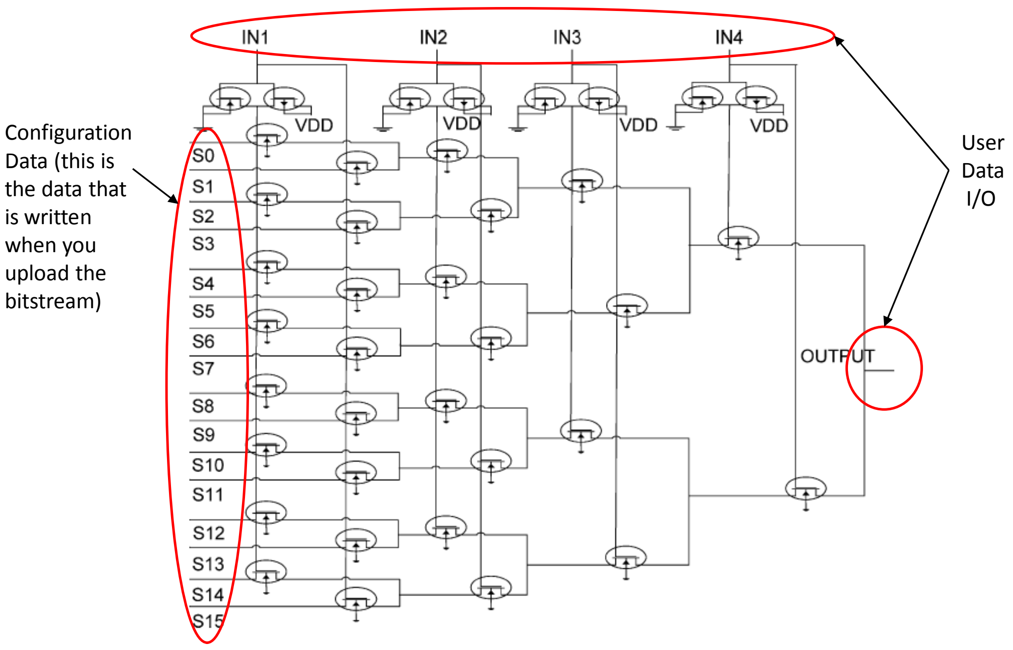
With these functions, you can indirect access specific bytes and half-words by only using lw and sw assembly instructions.

6.4 Getting accurate reports for area, delay, and critical path

The results for area and speed for your synthesis result like created with the tutorial in Chapter 6.1 are not accurate. This is, because the provided framework contains main additions like a state machine to communicate with the LCD via the I^2C bus or the data- and instruction memory and its connection to the CPU. These additions, which are needed for running the CPU on the hardware prototype, have a big impact on the measured size and speed of your CPU. Therefore, if you try to compare two different CPUs with the provided framework, you will mainly compare the framework with itself and it is hard to separate, which change in e.g. the CPU frequency is due to a change in the CPU or a more efficient optimization of the synthesis program due to a better interaction of CPU and framework.

To come around the above-mentioned problems, one might suggest synthesizing the plain CPU without any kind of framework to get accurate data without any impact of other components. Nevertheless, when you look at the output of this synthesis, you will notice, that all internal connections of the CPU will be automatically mapped to I/O-Pins of the FPGA, which has an impact of the size and speed of the synthesis result as well. This impact is due to the fact, that the I/O pins are rather slow compared to the FPGA-internal computation. However, you cannot force the synthesis tools to let the CPU connections unconnected, because then the synthesis tool would notice that there is no input to the CPU and that the output of the CPU is not used at all and thus it would remove the whole CPU for optimization reasons.

These two examples shall give you an impression how difficult it is to measure your results and how difficult it is to interpret the results of your measurements or even more: to compare two different measurements with each other. However, first we have to understand the unit in which the area is measured for FPGAs, i.e. a *Slice*.



IN1
IN2
IN3
IN4
VDD
VDD
VDD
VDD
Configuration Data (this is the data that is written when you upload the bitstream)
S0
S1
S2
S3
S4
S5
S6
S7
S8
S9
S10
S11
S12
S13
S14
S15
OUTPUT
User Data I/O

Figure 6-6
4-Input 1-Output Look-Up Table (4-LUT) [Kalenteridis04]

The basic block of a fine-grained configurable hardware (as used in FPGAs) is a Look-Up-Table (LUT) as shown in Figure 6-9. The shown example (using 4 inputs at the top and 1 output at the right side) can realize every Boolean function with four inputs. For each input combination (2^4=16) a dedicated configuration bit (S0-S15) can be programmed with the corresponding answer. The transistors feed the value of the selected configuration bit to the output. For instance, if S0-S14 are programmed to contain the value ‘0’ and just S15 is programmed to contain the value ‘1’, then this LUT behaves like a 4-input and-gate.

Many of the 4-LUTs are required to e.g. implement a 32-bit adder and additionally these LUTs have to be connected. FPGAs typically cluster their logic, i.e. they have local blocks with a strong interconnect, but less strong global interconnects. Figure 6-10 shows how Xilinx clusters the LUTs into so-called Slices (including a 1-bit register per LUT) and Configurable Logic Blocks. They have strong interconnects and especial logic for e.g. carry-chains to implement adders. Figure 6-11 shows how the CLBs are connected with configurable switches and dedicated connections.

Altogether, FPGAs provide a huge amount of these logic resources, e.g. the Virtex-II 3000 offers 14,336 Slices. The biggest Virtex-II FPGA (i.e. 8000) offers 46,592 Slices and the biggest Virtex-5 FPGA even offers 51,840 Slices (note: Virtex-5 additionally offers more logic per slice). Additionally they offer dedicated IP cores, e.g. multi-standard I/O ports, Digital Clock Managers, BlockRAMs, multipliers, and even PowerPC cores.

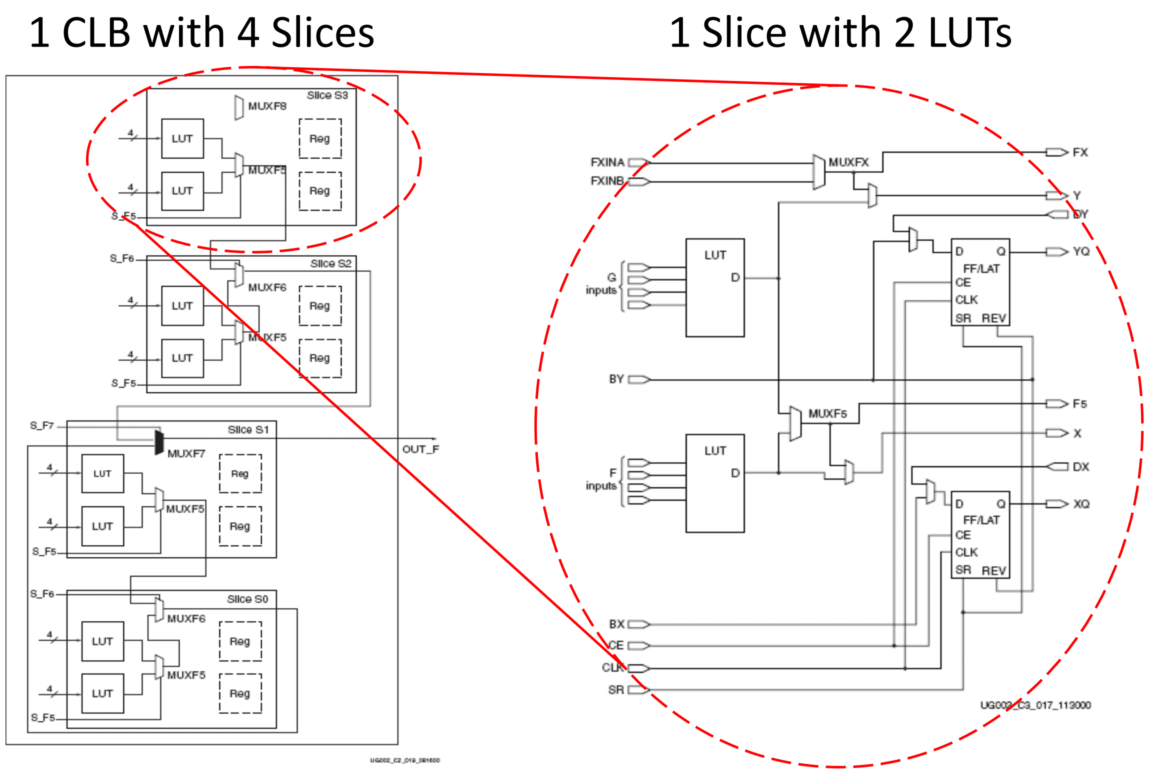


Figure 6-7
CLBs, Slices, and LUTs in a Virtex II FPGA [XUG002]

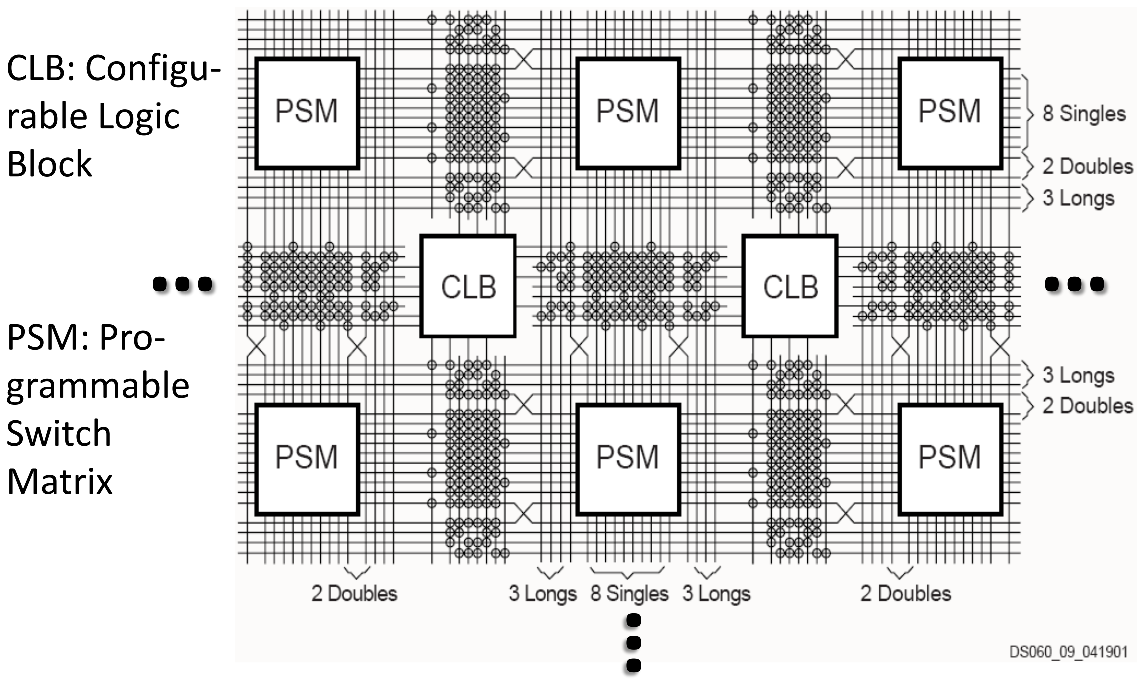
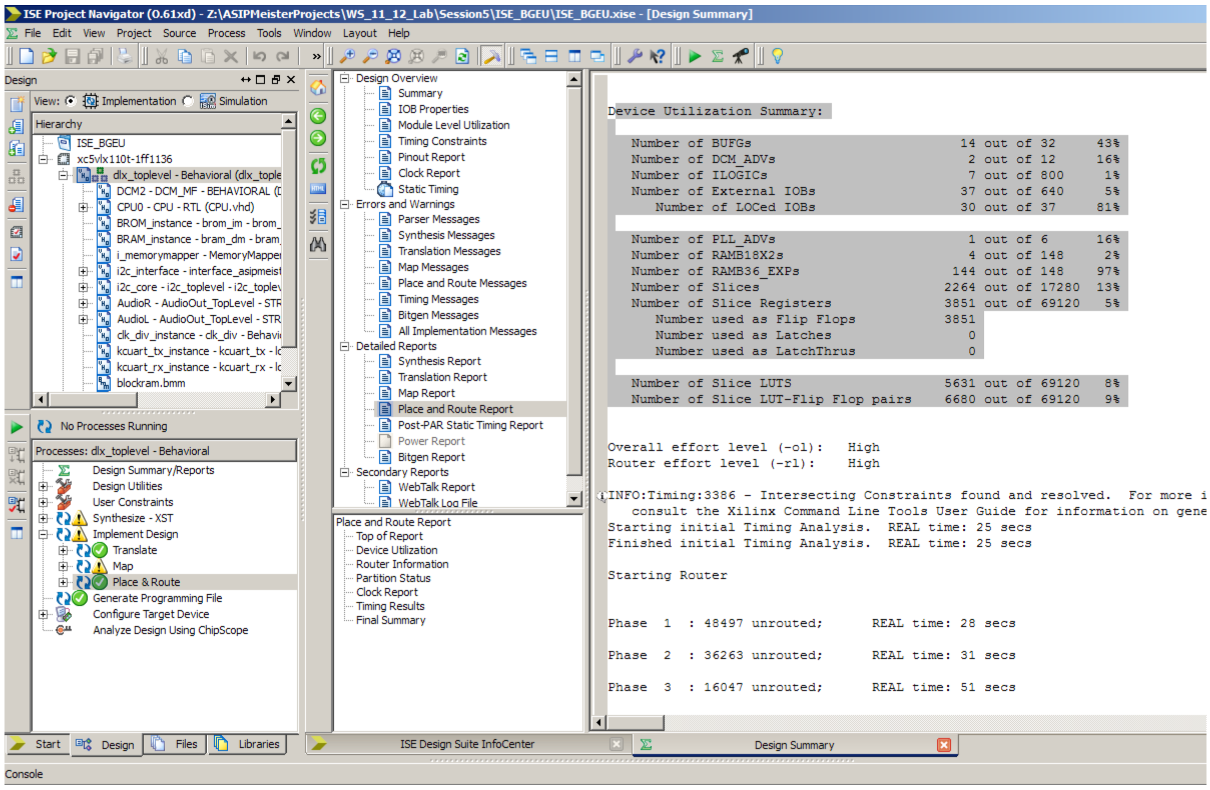

CLB: Configu-rable Logic Block
PSM: Pro-grammable Switch Matrix
PSM
CLB
8 Singles
2 Doubles
3 Longs
DS060_09_041901

Figure 6-8
Array of CLBs and PSMs [XDS060]

6.4.1 Getting Area and Delay report

To accurately measure only your CPU we designed a new framework, which consists of four files: bram_dm.edn, brom_im.edn, dlx_benchmark.vhd, and dlx_benchmark.ucf, which can be found in the directory ~asip00/ASIPMeisterProjects/TEMPLATE_PROJECT/ISE_-Benchmark/. 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 is the file, which contains the timing constraints and pins location for the design (in this step, we specify only the clock and reset constraints).

By using this framework, all the CPU connections will be mapped to the FPGA-internal BRAM memory and this will decrease the area needed to implement the project and will give more accuracy to compute the processor area and speed. To obtain the area and speed results for your CPU, your CPU files, and this special framework have to be synthesized and implemented as explained in Chapter 6.1 and Chapter 6.2.


ISE Project Navigator (O.61xd) - Z:\ASIPMeisterProjects\WS_11_12_Lab\Session5\ISE_BGEU\ISE_BGEU.xise - [Design Summary]
Design Overview
Summary
Errors and Warnings
Detailed Reports
Place and Route Report
Processes: dlx_toplevel - Behavioral
Place & Route
Design Summary

```
Device Utilization Summary:

   Number of BUFGs                          14 out of 32      43%
   Number of DCM_ADVs                        2 out of 12      16%
   Number of ILOGICs                         7 out of 800      1%
   Number of External IOBs                  37 out of 640      5%
      Number of LOCed IOBs                  30 out of 37      81%

   Number of PLL_ADVs                        1 out of 6       16%
   Number of RAMB18X2s                       4 out of 148      2%
   Number of RAMB36_EXPs                   144 out of 148     97%
   Number of Slices                       2264 out of 17280   13%
   Number of Slice Registers              3851 out of 69120    5%
      Number used as Flip Flops           3851
      Number used as Latches                 0
      Number used as LatchThrus              0

   Number of Slice LUTS                   5631 out of 69120    8%
   Number of Slice LUT-Flip Flop pairs    6680 out of 69120    9%


Overall effort level (-ol):   High
Router effort level (-rl):    High

INFO:Timing:3386 - Intersecting Constraints found and resolved.  For more i
   consult the Xilinx Command Line Tools User Guide for information on gene
Starting initial Timing Analysis.  REAL time: 25 secs
Finished initial Timing Analysis.  REAL time: 25 secs

Starting Router


Phase  1  : 48497 unrouted;      REAL time: 28 secs

Phase  2  : 36263 unrouted;      REAL time: 31 secs

Phase  3  : 16047 unrouted;      REAL time: 51 secs
```

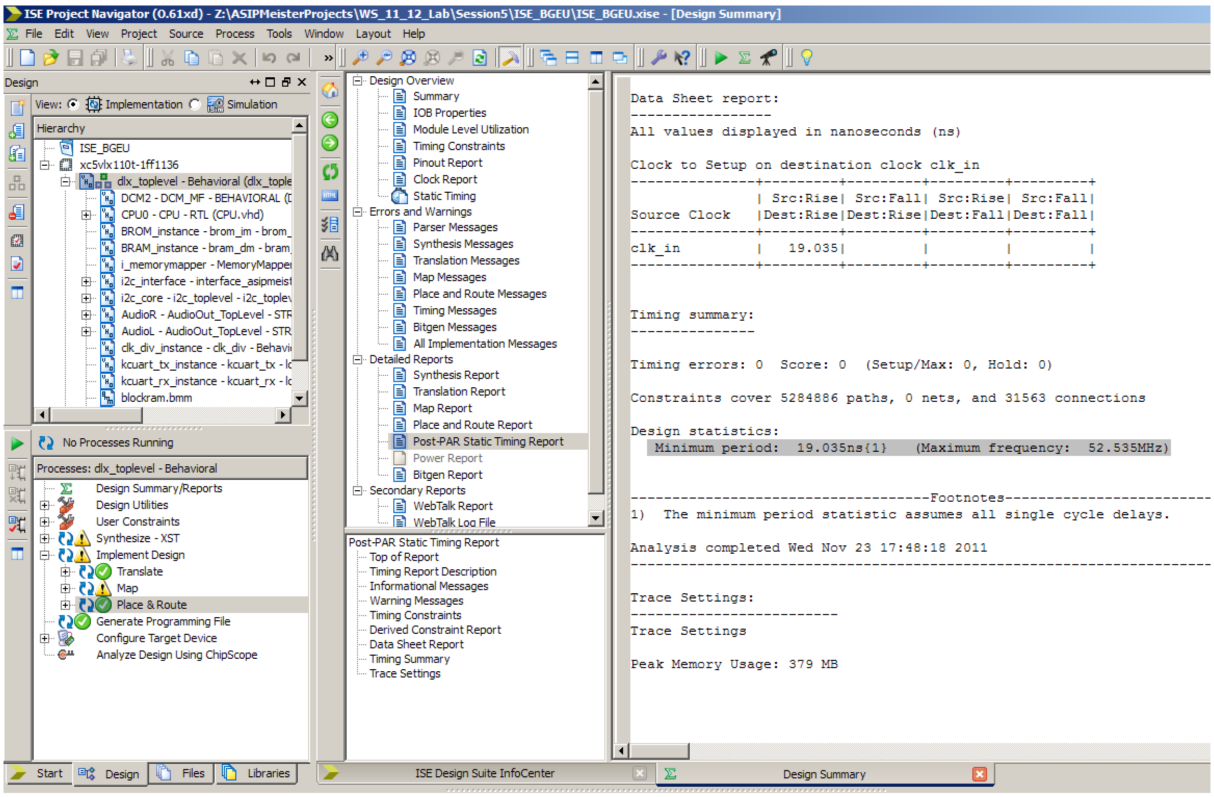
Figure 6-9
Area Report

Area Report:

In the Process sub window, expand "Place & Route" process. Double click on "Place & Route Report will open new window containing the results needed. Try to find the number of slices in the "Device Utilization Summary" as shown in Figure 6-9.

Delay Report:

In the Process sub window, expand "Place & Route" process and expand "Generate Post-Place & Route Static Timing Report". Double click on "Text-based Post-Place & Route Static Timing Report" will open new window containing the results needed. Try to find Minimum Period in the Design Statistics as shown in Figure 6-10.

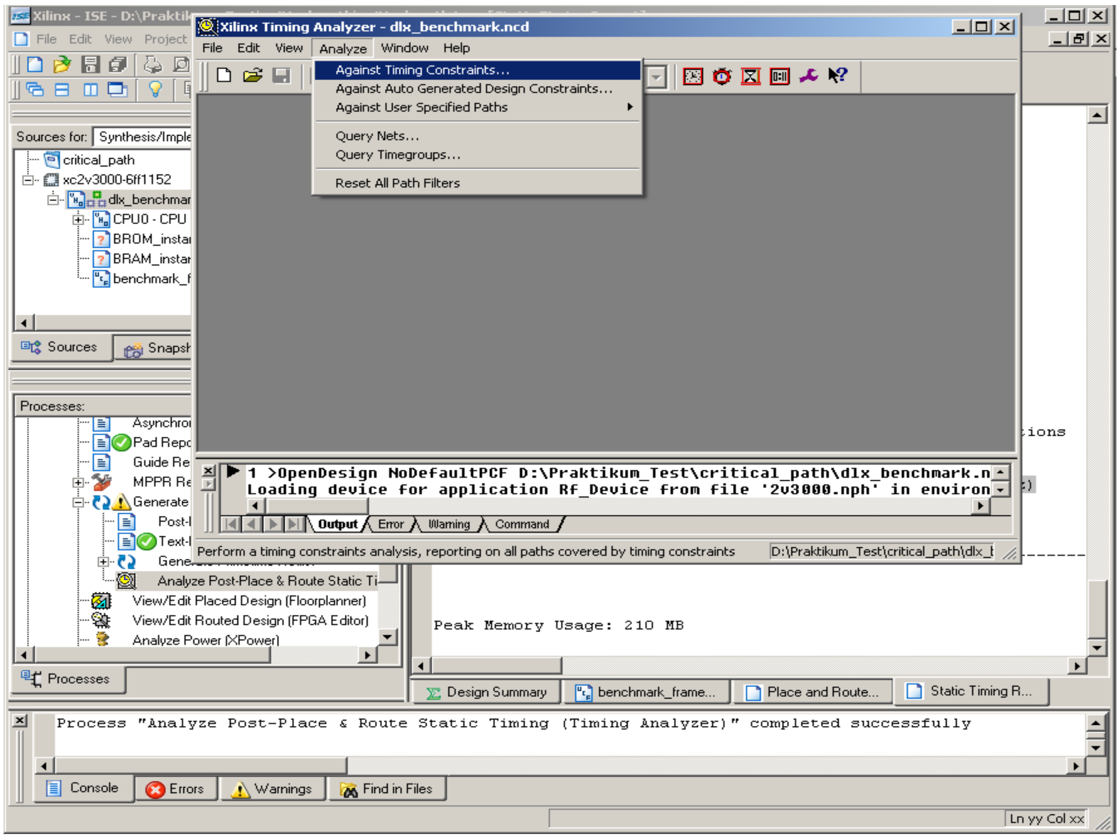


```
Data Sheet report:
-----------------
All values displayed in nanoseconds (ns)

Clock to Setup on destination clock clk_in
---------------+---------+---------+---------+---------+
               | Src:Rise| Src:Fall| Src:Rise| Src:Fall|
Source Clock   |Dest:Rise|Dest:Rise|Dest:Fall|Dest:Fall|
---------------+---------+---------+---------+---------+
clk_in         |   19.035|         |         |         |
---------------+---------+---------+---------+---------+


Timing summary:
---------------

Timing errors: 0  Score: 0  (Setup/Max: 0, Hold: 0)

Constraints cover 5284886 paths, 0 nets, and 31563 connections

Design statistics:
   Minimum period:  19.035ns{1}   (Maximum frequency:  52.535MHz)


------------------------------------Footnotes-----------------------------
1)  The minimum period statistic assumes all single cycle delays.

Analysis completed Wed Nov 23 17:48:18 2011
--------------------------------------------------------------------------

Trace Settings:
-------------------------
Trace Settings

Peak Memory Usage: 379 MB
```

Figure 6-10
Delay Report

6.4.2 Getting Critical Path report

To get the critical path, you have to analyze against the timing constraints. To do that, expand the "Place & Route" process in the sub window Process, and then expand "Generate Post-Place & Route Static Timing Report". A double click on "Analyze Post-Place& Route Static Timing (Timing Analyzer)" will open a new window for analyzing the timing. In this window, click on "Analyze → Analyze against timing constraints" as shown in Figure 6-11.


Xilinx Timing Analyzer - dlx_benchmark.ncd
File Edit View Analyze Window Help
Against Timing Constraints...
Against Auto Generated Design Constraints...
Against User Specified Paths
Query Nets...
Query Timegroups...
Reset All Path Filters
Sources for: Synthesis/Imple
Processes:
Analyze Post-Place & Route Static Ti
View/Edit Placed Design (Floorplanner)
View/Edit Routed Design (FPGA Editor)
Analyze Power (XPower)
Design Summary
Static Timing R...
Console
Errors
Warnings
Find in Files

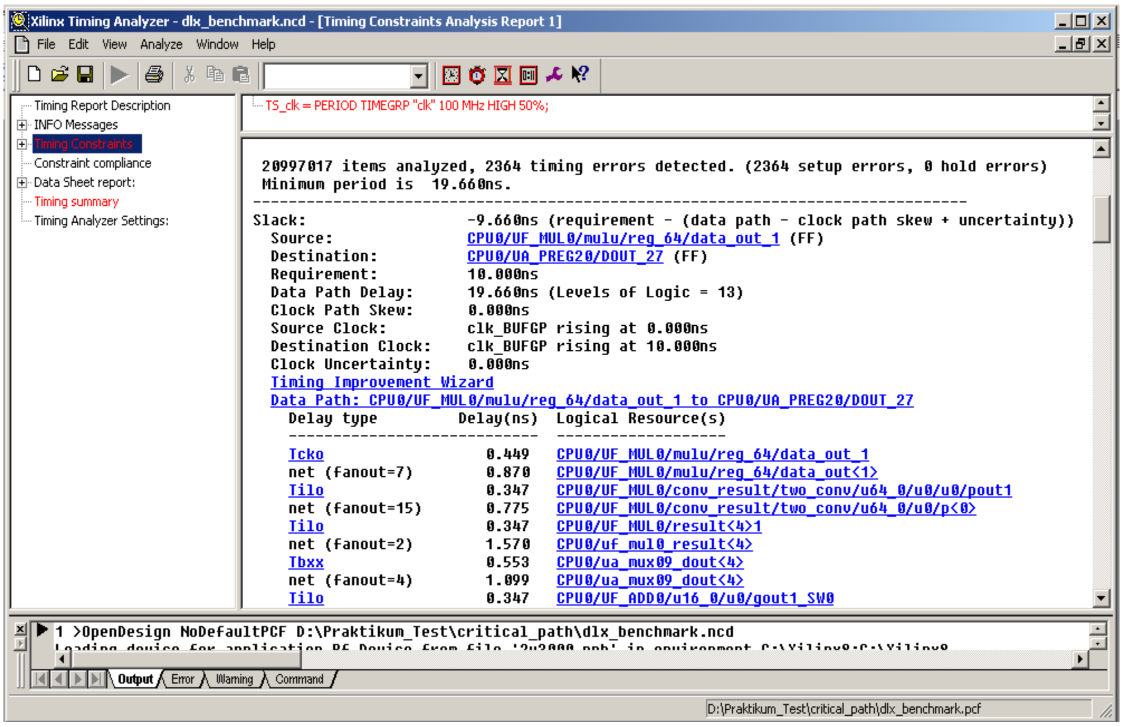
```
1 >OpenDesign NoDefaultPCF D:\Praktikum_Test\critical_path\dlx_benchmark.n
Loading device for application Rf_Device from file '2v3000.nph' in environ
```

Perform a timing constraints analysis, reporting on all paths covered by timing constraints

```
Peak Memory Usage: 210 MB
```

```
Process "Analyze Post-Place & Route Static Timing (Timing Analyzer)" completed successfully
```

Figure 6-11
Timing Analyzer Window


Xilinx Timing Analyzer - dlx_benchmark.ncd - [Timing Constraints Analysis Report 1]
File Edit View Analyze Window Help
Timing Report Description
INFO Messages
Timing Constraints
Constraint compliance
Data Sheet report:
Timing summary
Timing Analyzer Settings:
Output
Error
Warning
Command

TS_clk = PERIOD TIMEGRP "clk" 100 MHz HIGH 50%;

```
 20997017 items analyzed, 2364 timing errors detected. (2364 setup errors, 0 hold errors)
 Minimum period is  19.660ns.
--------------------------------------------------------------------------------
Slack:                  -9.660ns (requirement - (data path - clock path skew + uncertainty))
  Source:               CPU0/UF_MUL0/mulu/reg_64/data_out_1 (FF)
  Destination:          CPU0/UA_PREG20/DOUT_27 (FF)
  Requirement:          10.000ns
  Data Path Delay:      19.660ns (Levels of Logic = 13)
  Clock Path Skew:      0.000ns
  Source Clock:         clk_BUFGP rising at 0.000ns
  Destination Clock:    clk_BUFGP rising at 10.000ns
  Clock Uncertainty:    0.000ns
  Timing Improvement Wizard
  Data Path: CPU0/UF_MUL0/mulu/reg_64/data_out_1 to CPU0/UA_PREG20/DOUT_27
    Delay type         Delay(ns)  Logical Resource(s)
    ----------------------------  -------------------
    Tcko                  0.449   CPU0/UF_MUL0/mulu/reg_64/data_out_1
    net (fanout=7)        0.870   CPU0/UF_MUL0/mulu/reg_64/data_out<1>
    Tilo                  0.347   CPU0/UF_MUL0/conv_result/two_conv/u64_0/u0/u0/pout1
    net (fanout=15)       0.775   CPU0/UF_MUL0/conv_result/two_conv/u64_0/u0/p<0>
    Tilo                  0.347   CPU0/UF_MUL0/result<4>1
    net (fanout=2)        1.570   CPU0/uf_mul0_result<4>
    Tbxx                  0.553   CPU0/ua_mux09_dout<4>
    net (fanout=4)        1.099   CPU0/ua_mux09_dout<4>
    Tilo                  0.347   CPU0/UF_ADD0/u16_0/u0/gout1_SW0
```

```
1 >OpenDesign NoDefaultPCF D:\Praktikum_Test\critical_path\dlx_benchmark.ncd
```

D:\Praktikum_Test\critical_path\dlx_benchmark.pcf

Figure 6-12
Critical Path in the Xilinx Timing Analyzer

In the “Analyze against timing constraints” window, keep everything, as it is and click OK. After a while, the timing analyzer window will be opened. In this window (shown in Figure 6-12), click on the “Timing constraint” and you will get the critical paths ordered from the longest to the shorter paths. For each path, you will find the total delay and the sub-delays for the signals this path consists of.

In the shown example of Figure 6-12 you can see, that the path has it's "source" and "destination" inside the CPU0 (i.e. the instantiation of the ASIP Meister CPU, as it is named in the dlx_benchmark.vhd). When looking at the detailed signals this path consists of, you can see, that it starts in the multiplier of the CPU (MUL0, as it is named in the "Resource Declaration" in ASIP Meister), afterwards goes through a multiplexer (mux09) and then goes to the adder (ADD0, as it is named in the "Resource Declaration" in ASIP Meister).

When comparing the maximal frequency of two different CPUs you have to look at the critical paths of both CPUs to understand why maybe the one CPU is slower than the other or vice versa.

7 Power estimation

In this tutorial we will learn how to estimate the power by using ModelSim to generate the switching activity of the design, and xPower (from Xilinx) to analyze the results and generate the final power report and CosmosScope to visualize the results.

7.1 Different types of power

Typically, power dissipation in a cell is subdivided into two different groups, dynamic power, and static power. **Dynamic power** is the power dissipated in a cell when the input voltage is actively transitioning. Dynamic power is further subdivided into two components switching power and internal power. **Switching power** is the power required to charge the capacitive load on the output pins of the cell. Switching power is shown in Figure 7-1 as the I_{sw} switching current charging up the C_{load} capacitor. This component of power is calculated using the familiar ½ CV^2 equation.

For switching power, all we need to know is V_{dd} and the capacitive load that is driven by the cell. Therefore, the library does not need to be characterized for this component of power dissipation.

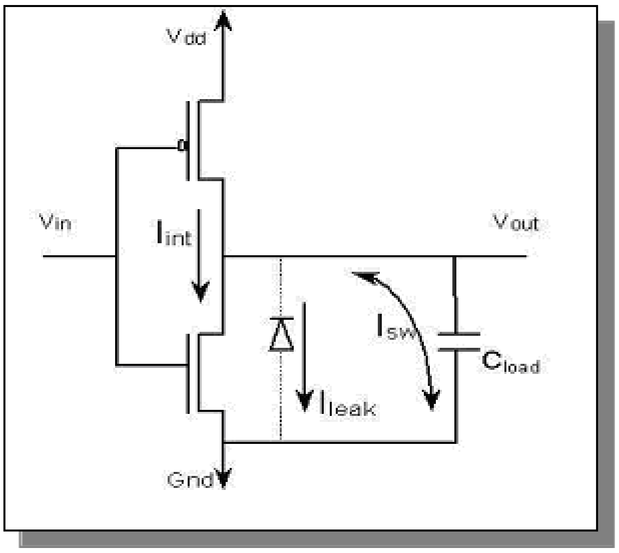


Figure 7-1
Switching Power dissipation

Internal power consists of short-circuit power and power dissipated by charging the capacitive loads that are internal to the cell (not shown in the above circuit). Short-circuit power is the power that is dissipated due to the short period that paths in the cell are essentially short circuits. In the circuit shown above, I_{int}, that is the current path when the device is shortcircuited, shows internal power. Every time the input V_{in} toggles, there is a short period of time where both transistors are turned on and there is a path from V_{dd} to ground. The longer both transistors are active, the higher the power dissipation. The circuit above is quite simple (an inverter) and there is only one path from V_{dd} to ground. For complex circuits, you could have dozens of potential short circuit paths. Internal power is dependent on the transition time of the input voltage V_{in} and the output capacitive load. These factors determine how long the short circuit is active. There is no easy formula to determine the power dissipated due to internal power, so the cell must be characterized for it.

The final component of power analysis is **leakage power**. In the circuit diagram in Figure 7-1 it is shown as leak current. This is the power dissipated when the circuit is in a steady state and it is due to the following factors inherent in transistors: reverse bias leakage current, sub-threshold current, or other second-order leakage power. Leakage power has become a major factor in power analysis.

At 180 nm and above, leakage power was typically less than 1% of the total power dissipation in a circuit. With 130 nm and below, the leakage power becomes a much larger factor, up to 50% of dissipated power in some cases. Leakage power should be characterized for each cell in the library.

To report the switching power, we need the switching activities of the design. The switching activity contains information about the static probability and toggle rate. The static probability can be calculated during a simulation by comparing the time a signal is at a certain logic state (state 0 or state 1) to the total time of simulation. The toggle rate is the number of transitions between logic-0 and logic-1 (or vice versa) of a design object per unit of time.

The previous power definitions can be concluded by these two equations:

Total Power	=	Dynamic power	+	Leakage power
Dynamic Power	=	Switching power	+	Internal power

7.2 Estimating the power consumption

To estimate the power consumption for a specific application on a specific CPU you need to generate the switching activities, which are saved as a "value change dump" file (.vcd). This file is generated when the project is simulated (the application is executed on the CPU) using ModelSim. Then the .vcd file will be used as input to the xPower tool, which finally will generate the Power report. This report can be visualized using the CosmosScope visualization tool, as shown in Chapter **Error! Reference source not found.**.

7.2.1 Generate the "value change dump" file using ModelSim

To generate the .vcd file, start ModelSim and create a project with the following files (you may also reuse an existing project if it contains the same files):

- All CPU VHDL files (from .syn folder of ASIP Meister)
- The application files (TestData.IM and TestData.DM)
- Testbench VHDL file (from the typical ModelSim simulation)

(1) Configure the CPU Frequency for which you want to run the power estimation. Open the ModelSim testbench (tb_ASIPmeister.vhd), search for CLK_HALF_PERIOD, and change the value accordingly. Take care: For simulation-reasons this value corresponds to the time (in nano seconds) of a half clock period. For example, 10 ns half period correspond to 20 ns clock period, which is 50 MHz frequency.
(2) Compile the project Compile → Compile Order → Auto Generate
(3) Start the simulation: VSIM > **vsim -t 1ns work.cfg**
(4) Store the activities during the program execution in the test.vcd file by typing: VSIM > **vcd file test.vcd**
(5) Add all signals of the CPU instance: VSIM > **vcd add -r test/dut/***
The entity name of the tutorial example testbench is *test* and the instance name of the device under test is *dut*. Using the -r switch with ModelSim's 'vcd add' command will result in a large but significantly more accurate VCD file. VCD files can grow quite large for larger designs or even for smaller designs if the simulation run time is long.
(6) Run the simulation: VSIM > **run -all**
The activities will be saved as test.vcd file.

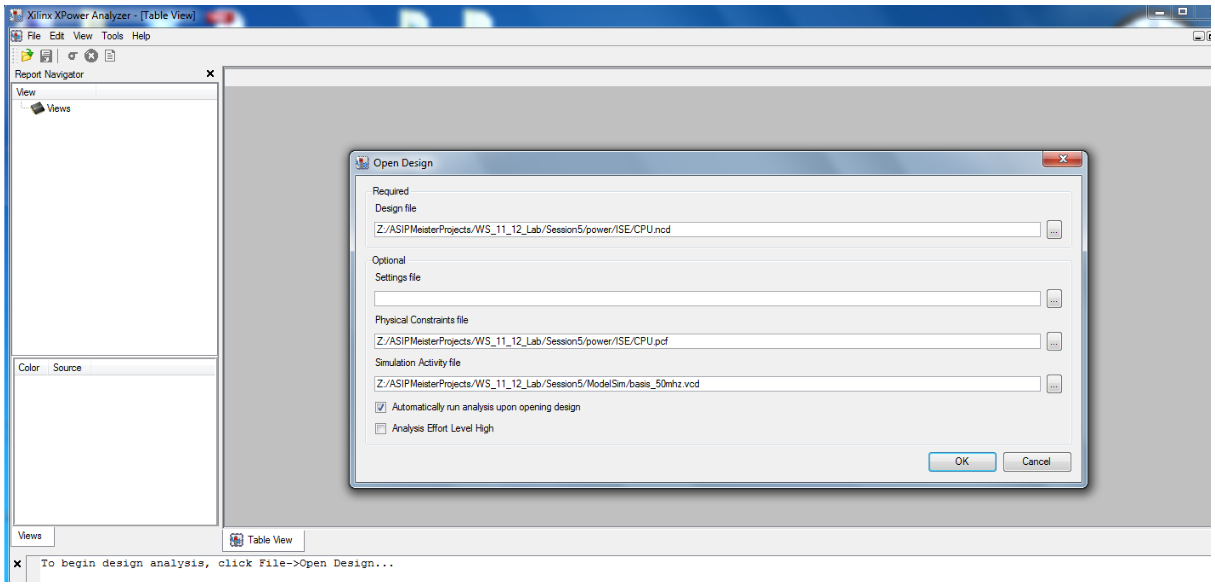
7.2.2 Generating the power report using xPower

The second step is to create an ISE project that you want to analyze. Make a new project and add only the CPU files from the ASIP Meister .syn directory to it. In the "Processes"-tree of your top-level and under "Implement Design/Place & Route", click on "Analyze Power Distribution (xPower Analyzer)". This will open the xPower window. Now click on "File/open Design", in the Design file, you have to add your .ncd file "e.g. toplevel.ncd" and in Simulation Activity file, you have to add the .vcd file that generated from the previous step. Finally, in the Physical Constraint file, you can add the .pcf file for you project "e.g. toplevel.pcf" as shown in Figure 7-2. As soon as you click ok, the Xpower tool will simulate the files and gives you a summary about the power consumption in your design (like the Leakage power and the total power). On the right side, you find "By Clock Domain". Here you see the frequency in (MHz). This should be the same frequency that you prepared when the .vcd file generated as shown in Figure 7-3.

Important Note:

It is worthy to know that the total power number that you get from Xpower is the Dynamic power plus the leakage power. When you read the leakage power consumption, you will see that is much higher than the dynamic power. This is because the leakage power is total leakage power consumed in the whole FPGA chip even if your design is small and it occupies a very small part in the chip. Since Vitrex5 does not have a power gating, the leakage power is always consumed and high.

For that, when you take the power number for your design, you have to consider only the dynamic power in order to make your results comparable and see clearly how much you have to pay for your custom instructions in terms of power.



Required

Design file: Z:/ASIPMeisterProjects/WS_11_12_Lab/Session5/power/ISE/CPU.ncd

Optional

Settings file:

Physical Constraints file: Z:/ASIPMeisterProjects/WS_11_12_Lab/Session5/power/ISE/CPU.pcf

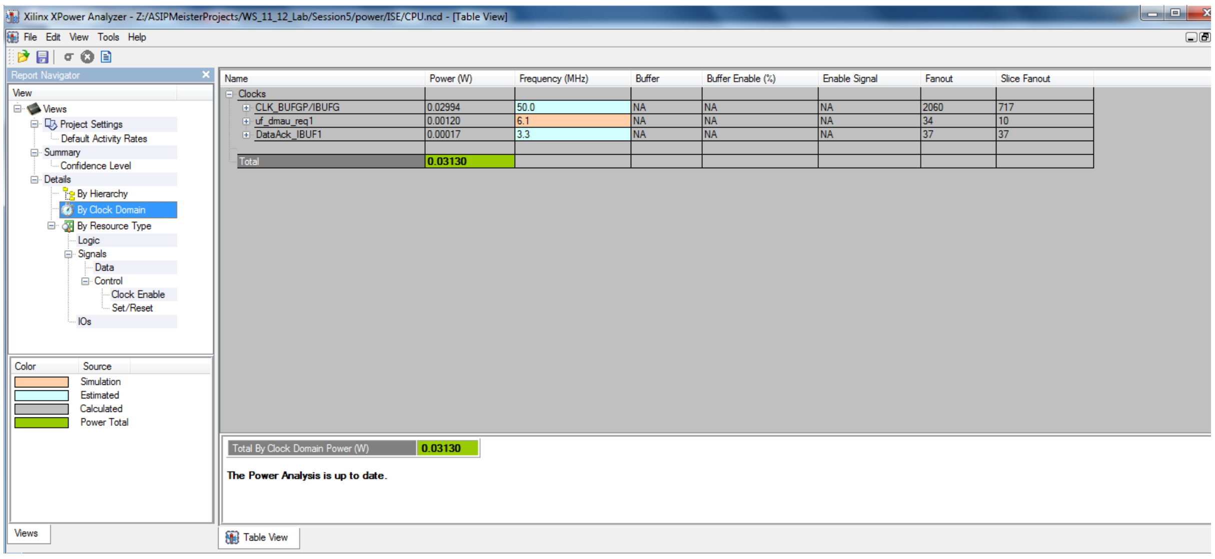
Simulation Activity file: Z:/ASIPMeisterProjects/WS_11_12_Lab/Session5/ModelSim/basis_50mhz.vcd

[✓] Automatically run analysis upon opening design

[ ] Analysis Effort Level High

```
To begin design analysis, click File->Open Design...
```

Figure 7-2
Preparing Xpower Tool



Color	Source
🟧	Simulation
🟦	Estimated
⬜	Calculated
🟩	Power Total

Name	Power (W)	Frequency (MHz)	Buffer	Buffer Enable (%)	Enable Signal	Fanout	Slice Fanout
Clocks							
CLK_BUFGP/IBUFG	0.02994	50.0	NA	NA	NA	2060	717
uf_dmau_req1	0.00120	6.1	NA	NA	NA	34	10
DataAck_IBUF1	0.00017	3.3	NA	NA	NA	37	37
Total	0.03130						

Total By Clock Domain Power (W)	0.03130

The Power Analysis is up to date.

Figure 7-3
Checking the CPU Frequency

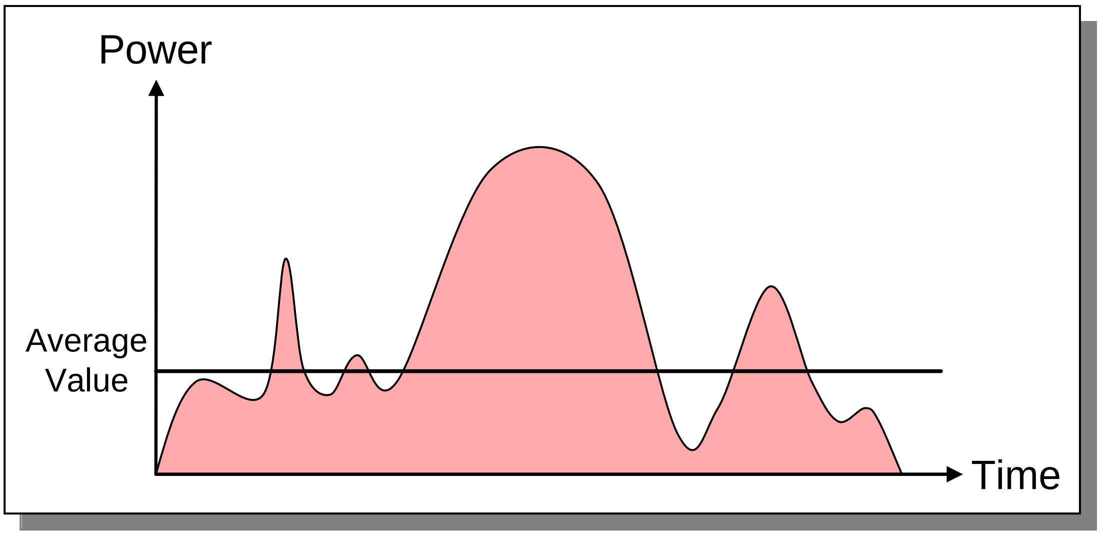


Figure 7-4
Power dissipation as average value

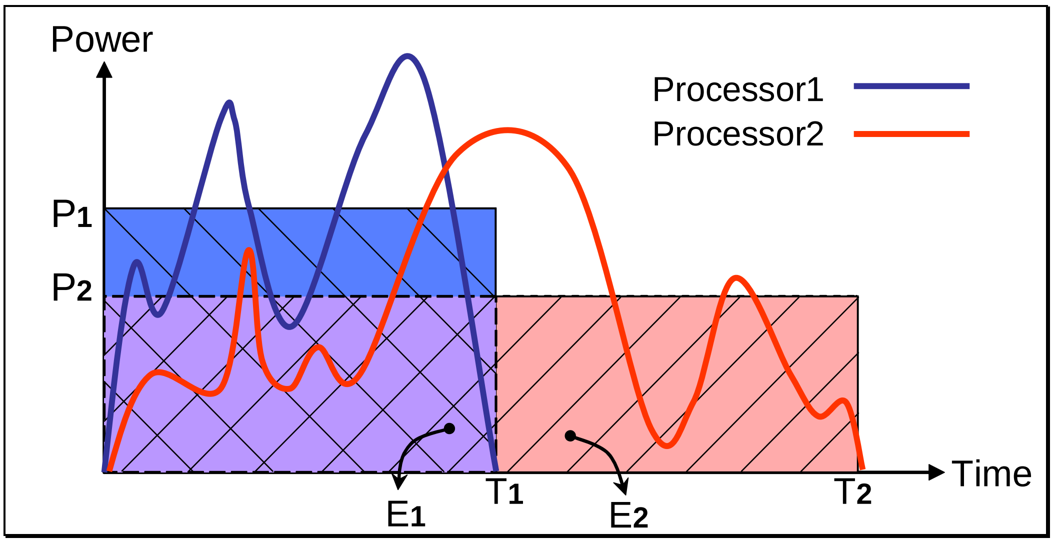


Figure 7-5
Energy comparison between two processors

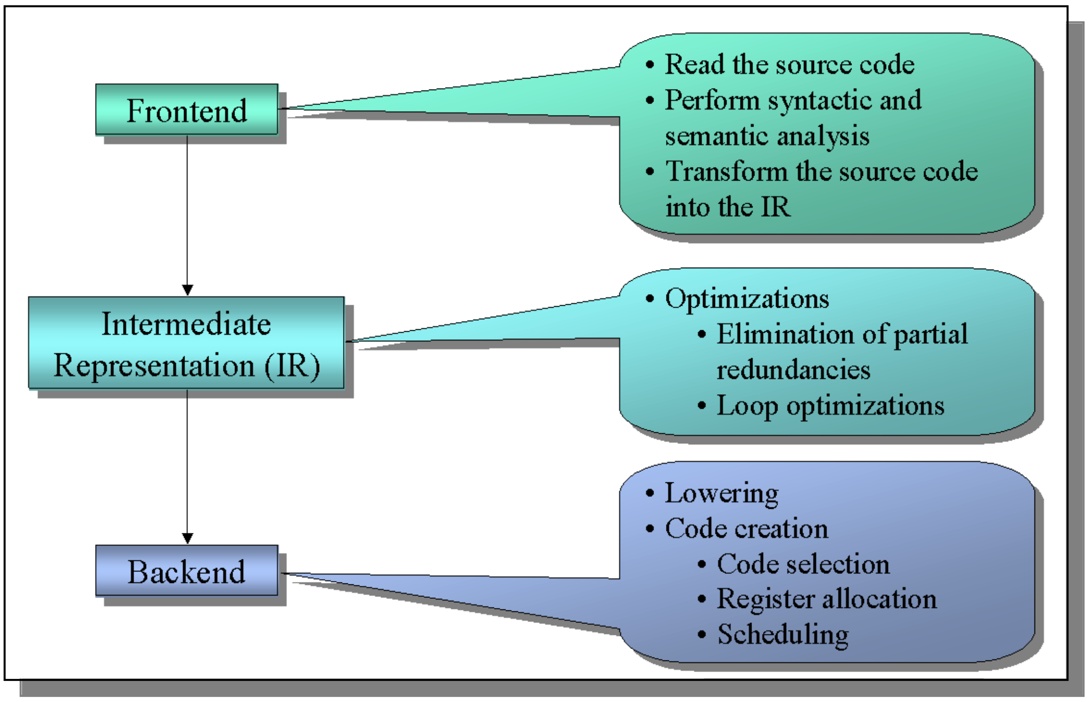
To compare the power consumption between two processors, the average value will not give correct information because every processor can execute the same application with different execution time, as shown in Figure 7-5. In this case, we need to consider the energy. The energy is defined as the power consumption during the execution time, and given as:

$$E(t) = \int_0^T P(t).dt \Rightarrow E = P * T$$

8 CoSy compiler

The CoSy compiler development system [CoSy] is a compiler generator that automatically creates an executable compiler out of an architecture description. In our case, ASIP Meister automatically creates this architecture description itself and an afterwards running program provided by the developers of ASIP Meister. This chapter will explain some basics about the buildup of retargetable compilers und in the following subchapters the creation and usage of a CoSy compiler for our specific ASIP Meister CPU's will be explained.

8.1 Basics about retargetable compilers



Frontend
• Read the source code
• Perform syntactic and semantic analysis
• Transform the source code into the IR
Intermediate Representation (IR)
• Optimizations
• Elimination of partial redundancies
• Loop optimizations
Backend
• Lowering
• Code creation
• Code selection
• Register allocation
• Scheduling

Figure 8-1
A typical buildup for a retargetable compiler

A typical retargetable compiler is separated into 3 phases, as shown in Figure 8-1. The first stage is architecture independent, but source language dependent. This phase reads the source code, inspects it for syntactic and semantic correctness, and transforms it into the second phase, the intermediate representation (IR). This IR is as well source language independent as architecture independent and it is used to perform the optimizations. This implies, that the optimizations can be easily reused, if the source language or the architecture changes. The third phase is source language independent, but highly architecture dependent. This backend first transforms the IR into a low-level form. That means, that high-level structures, like polymorphic procedure calls are replaced by jump tables or that complex data structures are disassembled into elementary memory accesses. Afterwards the final assembly code has to be created. This part is separated into three steps. In the first step, code has to be selected out of the lowered IR. This code selection is not unique, as there are always different possibilities to represent some statements in assembly language. This code selection works with virtual registers, which are replaced by real registers in the second step. This register allocation might lead to additional stack accesses for swapping values out, if no free real register can be found to hold the value of a virtual register. In the third step, the code is scheduled, to minimize penalties for data dependencies. Every step from this code selection has a great influence on the outcome of the other steps. The sequence of these steps is not determined and different compilers work with different sequences. The above given order is just exemplary.

8.2 Creating the CoSy compiler

For creating your own CoSy compiler that is dedicated to your individual CPU, you have to invoke a script called "makeCoSy" inside your project directory (e.g. "dlx_basis" in Figure 2-2). It is important, that the script is executed inside the project directory, as it relies on relative positioned files. Before executing "makeCoSy", there are some points that need to be considered:

- You have to create the hardware and software description for your current ASIP Meister CPU. Although one might think, that it is enough to only create the software description it is important to also create the hardware, as some of the created files are needed for creating the CoSy compiler.

- The file "instruction_set.arch" in the "meister/dlx_basis.sw" directory contains information about all assembly instructions of your current CPU. This information also includes the behavior description of your instructions, as you have entered them in ASIP Meister. The only difference of the behavior description is that every kind of brackets is lost in this instruction_set.arch file. Therefore, you have to edit this file and you have to manually insert the missing brackets again, otherwise the created compiler might use a wrong definition of an assembly instruction and thus will not use this instruction in the expected way. As shown in Chapter 8.2.2 there are some kinds of assembly instructions, that will not be used by the compiler at all, but that create error messages while creating the compiler. If you are sure, that you don't want those instructions to be included into the compiler and you want to use inline assembly instead (as shown in Chapter 8.4), then you can remove the complete instruction definition out of the "instruction_set.arch" file.

- You have to adjust your project configurations in the "env_settings" file, as explained in Chapter 2.3 and especially in Figure 2-4. The COMPILER_PREFIX value has to be changed to your preferred name. This way you can create a new compiler and keep the old one usable. To switch back to the old compiler you just have to change this setting back, as "mkimg" is always using the compiler binary, that is chosen in the "env_settings" file, as explained in Chapter 8.3.

- The makeCoSy script copies all needed files to i80pc06, as this is our dedicated workstation, where the CoSy tool chain is installed. Whenever there are some error-messages while the generation, they rely to files that were copied to this workstation.

When you manually have to fix some falsely generated rules then you have to do it on this workstation.

- One step of the compiler generation has to be finalized by pressing <STRG>+C. This step will print an information to let you know when to press <STRG>+C.

- Except the “makeCoSy” script, there also exist a “contCoSy” script (speak: Continue CoSy). This script is meant for continuing the creation of a CoSy compiler, if the initial creation with “makeCoSy” was aborted with an error message, as shown in Chapter 8.2.2. In such cases, you have to manually fix the printed error message (usually on i80pc06) and afterwards you can continue the compiler creation with the “contCoSy” script. In fact, the contCoSy script will tell you the specific steps that have to be performed for manually continuing the compiler generation. Those steps are single instructions, that are printed related to your current project name and that can be copied and pasted to the Linux-shell to perform the needed steps.

8.2.1 Structure of a CoSy rule

To understand the typical problems while creating a CoSy compiler it is important to understand how the backend is created for the CoSy compiler generator. Every assembly instruction that shall be used in the backend is described by a rule. This rule explains the prerequisites and the effect of this instruction. In this chapter, the basics for such rules will be explained. An exemplary rule might look like the following:

```
RULE [add_0] add:mirPlus(rs0:GPR, rs1:GPR) -> rd:GPR;
CONDITION {
     (IS_CHAR(add.Type) || IS_SHORT(add.Type) || IS_SHORT(add.Type) ||
     IS_LONG(add.Type) || IS_LONG(add.Type))
}
COST 1;
EMIT {
     addInsnInfo_RSRSR( state, SCHED_OP_add,
          rd, SCHED_STRING_0, rs0, SCHED_STRING_0, rs1, gcg_cycle );
}
```

Every rule starts with the keyword **RULE** and continues with an optional name ([add]_0). Afterwards a source-sub tree of the lowered intermediate representation (LIR) is described (add:mirPlus(rs0:GPR, rs1:GPR)), followed by a target sub tree (rd:GPR). The goal of this rule is to perform a replacement of the source sub tree by the target sub tree in the LIR, if the afterwards described condition evaluates to true. The source-sub tree in this example contains a node of the LIR (mirPlus), which is named “add” for later references. This node has two inputs, which are settled to be registers (GPR) in this case. Both registers have a symbolic name (rs0 and rs1) for later references. Instead of an elementary register, another node can be used as the input for a node, this way sub trees can be described. The target in this example is only a single register, but it can also be a sub tree.

The next keyword in this example is the **CONDITION**. For every sub tree in the LIR that matches the pattern in the source-sub tree, this condition is evaluated to see, whether the replacement can be performed. The condition in this example uses some macros (IS_CHAR, …) and it refers to a node of the source sub tree (add) and accesses some fields of this node (Type).

With the **COST** keyword, the user can define how expensive a rule shall be. For example, the cost can reflect the execution time of the code that has to be executed to implement the behavior of this rule. While compilation of the application the code selector determines all rule combinations that are able to cover the whole LIR with rules and then it chooses the cheapest coverage concerning the COST clauses.

The last keyword in the example rule is **EMIT**. If a coverage of the LIR is found, chosen and scheduled, then the emit parts of all rules are considered. The emit parts contain plain C-code, but they can also access the properties of the named parts in the rule header. A usual emit part for the above given rule would look like:

```
EMIT {
     fprintf(outfile, "add %s, %s, %s\n", rd, rs0, rs1);
}
```

The parameter *outfile* is expected to be a file, that was opened elsewhere and the parameters *rd*, *rs0* and *rs1* will be replaced by the real register names, after the register allocator has selected them. In our ASIP Meister examples the emit part looks different, because the assembly instructions are not immediately written into a file, but they are written into a temporary data structure, that takes care about inserting the NOP instructions before the result is written to a file. This data structure can be accessed by the different versions of the addInsInfo (speak: Add Instruction Information) methods, like shown in the example above.

8.2.2 Typical problems while creating the CoSy compiler

While creating the CoSy compiler a tool chain is executed systematically. In this chapter only the three main point of this tool chain will be mentioned, as those three main points are also the main point for the typical problems. The entry to this tool chain is ASIP Meister, as it creates the software description for your current CPU. In the next step, this software description is transformed into CoSy rules by the tool "meistercg", which is created by the developers of ASIP Meister. The last step is compiling the CoSy compiler with the automatically created rules.

The typical problems in the first step (ASIP Meister software creation) are explained in Chapter 4. For the second step with "meistercg", there are very few problems. One example is an if-then-else construction in the behavior description of an assembly instruction. While ASIP Meister accepts this, it is not implemented in "meistercg" yet. Newer versions might support such behavior descriptions, but the current version will abort. The error message refers to a specific file (usually the instruction_set.arch in the dlx_basis.sw directory) that manually has to be changed (i.e. delete the corresponding assembly instruction with the if-then-else) to solve the problem.

The most problems will appear in the final stage, the compilation of the CoSy compiler itself. The reasons for those problems are typically falsely created rules from "meistercg". The most kinds of those falsely created rules are meanwhile automatically detected and fixed by a part of the makeCoSy script. If an error happens in this stage, then the related file and the line number will be printed in the error message. As the compiler generation is done on the workstation i80pc06, as explained in Chapter 8.2 you have to log in to this workstation and you have to manually correct the error in the rule and afterwards you can continue the compiler generation with the contCoSy script, as explained in Chapter 8.2. For correcting the rules you should look at the composition of these rules, as it is explained in Chapter 8.2.1.

There is a known bug that often appears in the arith.cgd: “ERROR: Identifier 'imm_2' not declared”. Might be another immediate, depending on what immediate you have used in your behavior description. There is an already existing imm_16, where you can look, how the imm_2 addition has to be added. Here are the steps for the solution:

- In ccmir.cgd (same directory as arith.cgd) add the following parts (both times in the same area, as the corresponding entry for the imm_16 is located):
    - RULE [imm2] o:mirInstConst -> imm_2;
      CONDITION {
      UnivInt_Is2(o.Value)
      }
    - #define UnivInt_Is2(t) (UnivInt_to_int(t) == 2)
- In nonTerm.cgd (same directory as arith.cgd) add the following part:
  imm_2 ADDRMODE;

8.3 Using the CoSy compiler

After you have created the CoSy compiler for your specific CPU like explained in Chapter 8.2, the binary file (i.e. the compiler) will be placed in your ASIP Meister project directory. The name of this binary is defined in the “env_settings” file, as explained in Chapter 2.3. This binary is now used to compile your applications.

Instead of using this binary directly, we provide a script that is not only invoking the binary but also doing some additional steps that are either needed or helpful. Those scripts are the different versions of the “mkimg” script (speak: “make image”, i.e. *memory* image that shall be executed by the CPU). To invoke one of those scripts you have to enter the specific application subdirectory (e.g. dlx_basis/Applications/TestApp/) and then execute `“../mkimg {fileName} {parameters}”`. The parameters will be explained for the different versions of the script in the remainder of this chapter. It is very important to remember to only invoke those scripts from the inside of your application directory, as it is explained in Chapter 2.3.

Altogether three different versions of mkimg exist. Those versions mostly differ in the kind of input files they accept and in the place where the assembler is executed. You can use the file “mkimgSettings” in your application subdirectory to adjust some behaviors of the mkimg scripts, as it is explained in Chapter 6.3. The details of the different mkimg-versions are shown in Figure 8-2. The similarities of the five versions are the created output files. On the one hand, they create a temporary directory “compiler_temp” in your application directory for temporary data and on the other hand, they create the actual output. If everything works fine, then you can ignore the compiler_temp directory. However, if an error occurs in the mkimg execution, then the compiler_temp directory includes temporary files that might help to understand the problem.

Script	Description
mkimg	This version is expecting C-Code as input. The syntax is: `../mkimg application.c {CompilerOptions}` The compiler options are directly forwarded to the compiler binary. Examples are: "-O0" or "-O4" for different optimization levels. When you want to manually change the created assembly code, then you can copy the .asm file from the created "compiler_temp" subdirectory to the application directory and change the suffix from .asm to .s. Then you can do your modifications in this file and use mkimg_froms to create the TestData.IM and DM out of this file.
mkimg_fromS	Similar to mkimg, except that not C-Code is expected as input file, but an assembly file. The syntax is: `../mkimg_fromS application.s`
mkall	Similar to mkimg, except that not only one single C-file is compiled, but all C-Files and S-Files in the current directory will be compiled and they will be linked together with all assembly files in the current directory to form one common binary. The syntax is: `../mkall {CompilerOptions}` The output assembly file will be called "*NameOfTheApplication-Directory*.dlxsim". The other versions of mkimg create an output filename that is similar to the input filename.

Figure 8-2
Different versions of "mkimg"

The following files are created by the different versions of the mkimg scripts as the actual output:

- *ApplicationName*.dlxsim: A file for simulating your application in dlxsim, as explained in Chapter 3.3.
- TestData.IM and TestData.DM: These files contain the memory images for the instruction and data memory. Those images are needed for the VHDL simulation with ModelSim, as explained in Chapter 5.1.4 and they are needed for synthesis, as explained in Chapter 6.3.
- TestData.IM_uart.txt and TestData.DM_uart.txt: Special versions of the above files that are required when running your application on the FPGA-Prototype and using the external SRAM as IM and DM. A bootloader will then use these files to initialize the SRAM with the actual IM and DM data via UART interface (see Chapter 6.3.2).

The different versions of the mkimg scripts do more than only creating the actual output. In fact there are some problems in the automatically generated compiler descriptions and thus in the automatically generated compiler, as shown in Chapter 8.3.2. For the known problems, we wrote a script that is invoked by mkimg to overcome the specific problem. That is why mkimg is creating so much output on the screen and in the temporary directory. However, generally everything is fine, when the last printed message says: FINISHED "mkimg".

The mkimg scripts for C-files will automatically add a `"#define COSY"` line at the beginning of the C-file and names the result "*ApplicationName*.cc". This way you can use `"#ifdef COSY … #else … #endif"` constructions to handle a gcc-version and a CoSy-version in the same file. For example, you can add debugging output in the gcc-version or you can implement gcc-versions of CoSy specific inline assembly functions, as shown in Chapter 8.4. This automatically added line will also affect the error messages, as the error message refers to the generated "*ApplicationName*.cc" file instead of your "*Application-Name*.c" file. This means, that every printed line number in the error message corresponds to the preceding line in your "*ApplicationName*.c" file.

8.3.1 Typical reasons why a special instruction is not automatically used

There are some cases, where an assembly instruction will not be used, although it should be usable for an application. For example, the branch instructions are not automatically made known to the compiler, by the meistercg tool from ASIP Meister, which automatically creates the rules for the compiler generation. Branch instructions need to be added manually to the set of rules, which is not a trivial task, as you have to understand the ASIP Meister specific concepts of rules for branch instructions to do so. Another reason for an unused assembly instruction is a behavior description of this instruction that does not match the application you are compiling. For example, you might have defined a shift instruction and you are using a division by four in your application. Now you might expect, that the shift instruction is used for this division, as a division by four can be expressed with a shift instruction. However, unless there is a special rule for this shift instruction that matches a division node with a power-of-2 immediate as input, the shift instruction will not be used.

Except the above-mentioned cases, where the CoSy compiler cannot automatically use an instruction without some manual changes to the set of rules, there are cases where an instruction really should be automatically applicable. For such cases, we provide a checklist that you should manually inspect to find the problem:

- Write an elementary C-Code to force the compiler to use your instruction. Make sure, that the data types are correct. Do not mix signed with unsigned values. An immediate value for the assembly instruction should always be a constant in the C-code. A register should always be a variable.
- Review the instruction_set.arch file in the meister/dlx_basis.sw directory of your current project. Verify the behavior description of your instruction. Sometimes necessary brackets are missing in this file, as explained in Chapter 8.2.
- Review the arith.cgd file on i80pc06. This file is automatically created by meistercg and it contains all CoSy rules for arithmetic instructions. You can find this file in `~/buildtrees/{COMPILER_PREFIX}cc/build/engines/{COMPILER_PREFIX}cg/cgd/`
  Inspect the rule for your instruction that is not used and make sure, that the combination of nodes (e.g. mirAdd) and the condition match your assembly instruction and that this rule can really be used in your elementary C-code. Details about the CoSy rules can be found in Chapter 8.2.1.

8.3.2 Typical problems while using the CoSy compiler

There are some known problems with the created compiler binary, due to faulty rule descriptions. Sometimes it even can happens, that the compiler binary aborts the compilation of your application with a run time error message, as some cases of an application seems not to be covered by the created set of rules at all. Here we provide a list of workarounds that you really should consider, when your program is working incorrect, or the compiler is crashing.

- Every function has to return a value, i.e. you may not use 'void' functions! The reason for this is, that the compiler sometimes uses the 'return register' e.g. as loop counter, when in this loop a void function is called, as it thinks, this void function will not need the return register. However, if this void function itself calls a function, which overwrites the return register, then the loop counter is lost.

- You cannot use structs as parameters (or return value) of functions. This needs copying all members of the struct to the function. Instead, you can use pointers to structs.

- Together with inline assembly, (see Chapter 8.4) there are additional problems with structs. Sometimes the compiler aborts with a run-time error message. In such cases, it often helps to modify the C-code such, that there are some independent instructions between the struct usage and the inline assembly call.

- Sometimes there are problems with initializing arrays that are declared locally in functions, e.g. "int a[] = {23, 42}". This can result in a run-time error message, as the compiler needs a rule for a parallel copy from the initial array ({23, 24}) to the memory space of the array (a), but such a rule was not defined. As workaround you can make the array global (or static) or you can separate declaration and definition of the array, i.e. "int a[2]; a[0]=23; a[1]=42;".

8.4 Support for Inline Assembly (SINAS)

It sometimes happens that a custom instruction is implemented correct in the compiler, but it is anyway not used, although it should be usable concerning the compiled C-Code. This happens very often in cases, where the custom instruction contains control flow structures, like an if-then-else construct. The reason for this behavior is that the compiler was not able to match the specific instruction in his internal representation of the application.

In such situations, there are two possibilities to overcome the problem. The first possibility is to extend the compiler with an optimization engine that is dedicated to finding places in the internal representation where the specific custom instruction can be used. This possibility is very time-consuming while creating the compiler and has to be redone for every new custom instruction, as different custom instructions usually need different strategies for identifying them in the internal representation. The second possibility is to use inline assembly. Inline assembly is a porting of assembly code that is explicit written into the C-Code at the place, where it should be used. This way we can insert any custom instruction right into the C-Code without modifying the compiler. We can also insert small subroutines in a hand-optimized assembly version into the C-code. This possibility is time consuming while preparing the application, but for very complex custom instructions, it is much faster than extending the compiler to automatically identify the instruction.

First, we will give a small example for the basic syntax for the CoSy specific version of inline assembly and later we will go into further details how the embedded assembly code can be parameterized.

```
asm int example(int parameter1, int parameter2)
{
     @[
            // Some later explained directives can be placed here
     ]
            ; This comment is written to the created assembly file
     nop
     nop
     nop
     add    @{}, @{parameter1}, @{parameter2}
     nop
     nop
     nop
            ; The @{parameter1} and @{parameter2} will be replaced by
            ; the used registers for those parameters. The @{} refers
            ; to the register for the return value.
}

void main() {
     int a=5, b=7, c;
     c = example(a, b);
}
```

The created assembly file would look similar to the following (changes by the register allocations are marked):

```
...
addi    r1, r0, $5
addi    r2, r0, $7
        ; This comment is written to the created assembly file
nop
nop
nop
add     r3, r1, r2
nop
nop
nop
        ; The r1 and r2 will be replaced by
        ; the used registers for those parameters. The r3 refers
        ; to the register for the return value.

...
```

The *asm* directive to create inline assembly is CoSy specific and so it will not compile e.g. with gcc. To write a C-Code, that can be used with gcc and with CoSy and that includes CoSy specific inline assembly, one can use a "`#ifdef COSY` … `#else` … `#endif`" constructions to handle a gcc-version and a CoSy-version in the same file. As explained in Chapter 8.3, a "`#define COSY`" directive is automatically added to the source code, if it is compiled with the mkimg script. To use inline assembly with the CoSy compiled version and an equivalent function call with the gcc-compiled version one can write something like:

```
#ifdef COSY
    asm int example(int parameter1, int parameter2)
    {
            ; ...
    }
#else
    int example(int parameter1, int parameter2)
    {
            // ...
    }
#endif
```

The most important directives are explained in Figure 8-3. They concern restrictions to the parameters and reservations of further registers.

Directive	Explanation
.scratch	With this directive, temporary registers can be requested. For example: `.scratch temp1, temp2` would reserve 2 registers which you can address in the assembly code as @{temp1} and @{temp2}. For every scratch register you need a restrict directive (see below). The name of a scratch register needs to be at least 2 characters long.

.restrict	With the restrict directive a parameter can be restricted to a special register class or a special register. In our ASIP Meister CPU there is only one register class, i.e. reg (but there might be different register classes, e.g. for floating point values). For example (upper case 'R' is important):
	`.restrict temp1:reg, temp2:reg<R17>`
	restricts temp1 to the register class reg and temp2 to the specific register R17 in the register class reg. Instead of restricting to a single register, you can also specify a register range. For example
	`.restrict  temp2:reg<R1-R5, R7, R9-R10>`
	forces temp2 to be in any register between R1 and R10 except R6 and R8.

Directive	Explanation
.change	With the change directive you can inform the CoSy scheduler, that a specific register is changed by the inline assembly. For example `.change R30, R31` announces, that the registers R30 and R31 might not contain the same values as they did before the inline assembly was executed. This is similar to requesting two scratch register and then explicit restricting those registers to R30 and R31.
.barrier	This directive is used to prevent the scheduler from moving instructions across the inline assembly instructions. This means, that every computation in the C-Code that is placed before the inline assembly part will be finished before the inline assembly code is executed and every computation in the C-Code that is placed behind the inline assembly part has not been started before the inline assembly code is finished.
.clobber	With the clobber directive you can inform the CoSy scheduler, that an input parameter is changed within the function. For example `.clobber parameter` informs the scheduler, that the register that is assigned to the parameter patameter1 will be changed by the inline Assembly. If the value of this register is needed after the inline assembly call, then the compiler will automatically create a backup of this register before the inline assembly is executed.
.interfere	With this directive you can inform the CoSy scheduler, that two input parameter should be located into different registers. For example `.interfere (parameter1, parameter2)` will make sure, that two different registers will be allocated for the both parameters. This is useful together with the clobber directive. If you want to use the both parameters as different temporary values you have to make sure, that they are in different registers
.unique	This directive makes sure, that all assigned registers for the parameters will be different. So it is similar to interfere, but it is treating all parameters, including the return value (@{}).

Figure 8-3
Explanation of the SINAS directives

It is also possible to handle inline assembly that returns more than one value. In this case, a structure can be returned and inside the SINAS-Function the members of this structure can be accessed like parameters.

```
struct myStruct { int div; int mod; };

asm struct myStruct example2(int parameter1, int parameter2)
{
    divmod @{div}, @{mod}, @{parameter1}, @{parameter2}
}
```

```
int main() {
     struct myStruct temp = example2(5, 7);
     return temp.div + temp.mod;
}
```

Sometimes the compiler aborts with an “internal fatal” error message when working with structs and SINAS. This is a problem in the compiler description (which is automatically generated by ASIPMeister). To avoid this problem (work around; no solution) you have to do some computation / assignment between the structure assignment and structure access (in the above example between “temp=example2(5, 7): and “temp.div”). You have to make sure that this computation / assignment cannot be removed by the compiler, for instance you can declare a variable “int foo;” at the beginning of the program, assign it a value “foo = 42;” between the structure assignment and usage, and return the value “return foo;” at the end of the program.

8.5 Library with standard functions for ASIP Meister / CoSy / Hardware Prototype

Many applications use some standard library calls like printf, malloc or atoi that are not declared in the standard of the C programming language, but which are nevertheless declared in the C standard library. Now, CoSy is a compiler for the C language and does not provide an implementation for the standard library (in fact there are some huge fragments that would have to be adapted to our specific environment, what has not been done due to the complexity of a full run-time implementation).

To close the gap between a plain C compiler and the wish of letting complex algorithms run in hardware and produce understandable output we are providing some basic stdlib functionality, which is dedicated to the environment of ASIP Meister, the CoSy compiler and our hardware prototype. This basic stdlib functionality will be extended on demand to reflect the latest changes of our environment. Thus, it will not be explained exhaustive or in high detail. Instead, the underlying concepts and the needed steps for using the basic stdlib library will be explained here, plus some of the main functions for using our touch screen LCD with some examples.

All typical functionalities of our stdlib implementation are available in the directory /Software/epp/StdLib/. The functionality is encapsulated into a header file that is providing the interface and a documentation of the functionality and a C file for implementing the header. You can use these files by linking them into your application project subdirectory and using “mkall” to compile your application and the stdlib files into one binary and simulation file, as it will be shown in the example below. Linking to the files instead of copying them has the advantage, that you always have the latest version of these files in your project. Some of the files in the stdlib directory have dependencies to other files of this directory. Thus, you can get a compiler error, that a specific header file was not found when you try to compile your project. You then have to manually link to the dependent files as well. Just linking to all available files is generally not a good idea, as this will make the compilation process take longer and it will increase the needed memory size of your application.

Now we will demonstrate how you can create a small application that is using the LCD of the hardware prototype:

- Create a new subdirectory inside the application directory of your ASIP Meister project and change into this new directory

- Link to lib_lcd, loadStorByte and string, by executing:

  ```
  ln -s /Software/epp/StdLib/lib_lcd.* .
  ln -s /Software/epp/StdLib/loadStoreByte.* .
  ln -s /Software/epp/StdLib/string.* .
  ```

  lib_lcd has a dependency to loadStoreByte and string, that is why you need both.

  The lib_lcd exists in different C implementations, depending on your target LCD. One C file is for the LCD with a 240x128 resolution, the other one is for the 320x240 resolution, and the last one is for the dlxsim simulation. Make sure that at most one of these C files is available in your application subdirectory; otherwise, the linker will complain about multiple implementations (one per C file) of the LCD functionality.

- Prepare a "mkimg_settings" file, as shown in Chapter 6.3.

- Add a new C file that contains you main method. This main method shall contain the following code as example:

  ```
  t_print("Hello World\r\n");
  t_printInt(42);
  ```

  The LCD needs the "\r\n" in the string, as it is handling carriage return (\r) and line feed (\n) independently.

- Compile your project by executing `../mkall`.

The resulting program can be simulated with dlxsim or ModelSim or it can be uploaded to the hardware prototype (explained in Chapter 6.3) depending on the selected LCD library.

After you have once compiled a C-Code that rarely changes (e.g. the libraries), you can reuse the created assembly code for future compilations. That will enormously speed up the whole compilation process. This is especially good for all applications in the StdLib directory. Just compile them one time with mkall and then copy the created .asm file from the compiler temp directory to the directory of your other application files, but name it .s instead of .asm. Then remove the C code (but not the header) to make sure the code does not exist twice (in the C code and in the Assembly file). For instance:

```
cp compiler_temp/loadStoreByte.asm loadStoreByte.s
rm loadStoreByte.c
```

However, when you change the mkimgSettings, e.g. moving from dlxsim to FPGA implementation then you have to delete these assembly files, link to the C files, and recompile them. If you are often switching between dlxsim and FPGA, then it may be beneficial to create two application directories with the different libraries and mkimgSettings specified for dlxsim and FPGA respectively and using the same application-specific source code in both directories (e.g. with a link).

As mentioned above, the "lib_lcd" exists in different variants. The files "lib_lcd_240.c" and "lib_lcd_320.c" are for the FPGA prototype. They implement the real protocol for communicating with the I^{2}C bus and thus with the LCD. For a simulation, we cannot use this implementation, as it is waiting for certain responses from I^{2}C/LCD, but in dlxsim/ModelSim simulation, these answers will never appear. We have therefore implemented “lib_lcd_-dlxsim.c” which simply writes every single character that is send to the LCD to an output file. This is a very good debugging possibility for printing text, although it is not helpful for any graphic output to the LCD (e.g. lines, bars, …), as these control words are hard to understand manually. For dlxsim, the characters are either printed to the console whenever they appear or they are collected and printed to a file (parameter “pf” in Chapter 3.2.1). For ModelSim the characters are printed to a file “lcd.out” in the ModelSim directory. The header file “lib_lcd.h” is valid for all C files, but you have to make sure, that at most one of the both C files is available in the directory when compiling with “mkall”. Otherwise, you will get two implementations of the LCD functions and the assembler will complain, that he cannot decide which one to choose.

8.5.1 Functions of the LCD library

This chapter will summarize some of the available functions to access the features of the touch screen LCD of our hardware prototyping board. The library also contains some high-level functionality that is useful, but introduces a dependency to “string.c”. Therefore, this library has to be provided as well when using lib_lcd.

The most important basic I/O instructions are: t_print(char*), t_printInt(int), and t_printHex(int value, int digits). You can use them to print strings and numbers, for instance:

```
char tempString[] = "\r\n\t\t";
t_print("Hello World!");
t_print(tempString);
t_printInt(23);
t_print(" = ");
t_printHex(23, 0);
t_print(tempString)
t_printInt(42);
t_print(" = ");
t_printHex(42, 4);
```

The printHex() function can trim the output to a given number of digits (4 in this case). To not trim the number of digits you have to give ‘0’ as parameter. The output of the above example looks like:

```
Hello World!
        23 = 0x17
        42 = 0x002A
```

In the following, we will describe further functions of the LCD. Some of them are generally sending a command to the LCD (you have to use the LCD manual [eDIP] to find the available commands) and some of them are offering typical commands (e.g. drawline) as convenience functions.

```
int checkbuffer()
```

This function returns the number of available bytes in the send buffer of the LCD. It can be used to wait for a return value of the LCD. For example when pressing a button on the touch panel, a value of this button will be written to the LCD send buffer.

```
int getbytes (char* dest, int bytes_to_read)
```

This reads a specific number of bytes from the send buffer. With the *checkbuffer* function, you can test how many bytes are available in the buffer.

```
int sendcommand (const char cmd0, const char cmd1,
                 const int options[], const char text[],
                 int intcount, int charcount, int address)
```

This is a general function for sending commands to the LCD. The following commands internally use *sendcommand* to realize their functionality. The parameters of *sendcommand* are:

- Two chars, specifying the command
- Depending on the type of the command, some options
- Depending on the type of the command a string with a predefined size
- The address of the LCD (defined in lib_lcd.c)

```
int t_print (const char* str)
```

This function writes a string to the LCD terminal. The "t_" indicates a command for the console mode of the LCD, compared to the graphic mode (g_) of the LCD, which will be explained later.

```
int t_cursor (int onoff)
```

Turns on (1) or off (0) the blinking cursor of the terminal.

```
int t_enable (int onoff)
```

Turns the display on (1) or off (0). When the display is turned off, all submitted data will be ignored. Previously sent data (when the display was on) will be buffered and will become visible again, after the display will become turned on again.

```
int g_print (const char* str, int x, int y)
```

Writes a string to the coordinates (x, y). You must not send control signals like \n in this function. They are only available for the t_print function.

```
int g_drawrect (int x1, int y1, int x2, int y2)
int g_drawline (int x1, int y1, int x2, int y2)
```

Draws a rectangle/line.

```
int g_makebar (int x1, int y1, int x2, int y2, int low_val,
               int high_val, int init_val, int type,
               int fill_type, int touch)
```

Creates a bar graph at the defined coordinates. `low_val` and `high_val` describe the minimal and maximal (at most 254) value of the bar graph. `init_val` defines the initial value and `type` and `fill_type` adjust the appearance of the graph. Type=1 will draw a bar in a box and the fill_type (in the range of 1 to 15) then defines the fill pattern. Type=3 will draw a line in a box and the fill_type will then define the thickness of the line. For more details refer too [LCD].

With `touch` you can define, whether the bar graph shall be user changeable by the touch screen. Every bar graph gets a unique number, which is returned by the function. At most 32 bar graphs are supported by the display. When the touch screen functionality is activated and the user changes the value of the bar graph, the LCD automatically writes the number of the changed bar graph and its modified value to the buffer, from where it can be received with the checkbuffer and getbytes function.

```
int g_setbar (int barnum, int value)
```

Sets an existing bar graph to a specific value.

```
int g_makeswitch ( const char* str, int x1, int y1, int x2, int y2,
                   int down, int up)
```

Creates a switch (button) with a label. The parameter `str` contains this label, preceded by a control char which defines the alignment of the label. “C” means centered, “L” means left- and “R” means right-aligned. `down` and `up` define the code, which the LCD shall write into the send buffer, when the button is pressed or released. When 0 is provided as parameter for `up` or `down`, then nothing will be written to the buffer for the specific activity.

```
int g_makeradiogroup (int group_number)
```

A new defined switch can be assigned to a specific radio group. A radio group automatically makes sure, that at most one switch of the group is pressed.

```
int g_makemenubutton (const char* str, int x1, int y1, int x2, int y2,
                      int down, int up, int select, int space)
```

A menu item is created. With `str` you can define the appearance and the menu entries. The first character defines the direction in which the menu shall open (“L”: left, “R”: right, “O” up, “u”: down). The second char defines the alignment of label on the menu item (“C”: centered, “L”: left aligned, “R”: right aligned). The labels of the menu entries follow afterwards and are separated by a “|” sign. `down` defines the value that shall be written to the buffer, when the menu item is pressed and `up` defines the value that shall be written to the buffer, when the menu is closed, without choosing a specific entry (aborted). `select` defines the base value that is used to compute the value that will be written to the LCD buffer, when a menu entry is chosen. This value is computed as: base value + entry number – 1. `space` defines the gap in pixels between the menu entries. This is a global value, thus it will also change the appearance of already existing menus.

8.5.2 Functions of further libraries

Besides the LCD, further functionality and helping functions are available. They will be summarizes in this section to give an overview of the available features. You should look into the libraries to find out more.

lib_uart: besides printing to LCD, you may also print to the UART port using the u_print, u_printInt, and u_printHex functions (similar to t_print etc.). You can read from the UART port with u_getbytes. However, you need a HyperTerminal running at the pc to which the UART is connected (see Chapter 6.3.2 for details about the HyperTerminal).

lib_audio: you can write uncompressed PCM audio data to an audio port using the writeToAudio{L|R} functions. The corresponding sample width of the corresponding IP core in the VHDL-framework (default 16 Bit) can be configured in AudioOut_Types.vhd and the sampling rate (default 40 KHz) can be configured in dlx_Toplevel.vhd (search for 'i_audio_out').

clockCounter: you can use the functions writeClockCounter and readClockCounter to access a special register that automatically increments by one in each cycle. You can use this to benchmark the performance of a certain C-Code by e.g. writing a '0' to it to start the benchmark and to read it again at the end of the benchmark. Make sure that the code that shall be benchmarked does not contain any unnecessary I/O instructions (e.g. t_print) as they are extremely slow.

String: contains some typical string manipulation functions, e.g. strlen, strcat, strcpy, … It additionally contains two functions to convert an integer number to a string representation, as it is used in the I/O libraries, e.g. t_printInt or u_printHex.

8.5.3 Changing the Frequency

In order to change the frequency of your design and see if it still works or not (required for the last Session), the Framework contains a DCM (Digital Clock Manager) that generates five different clocks. By changing the knop on the external PCB, a multiplexer (in the toplevel vdhl file) selects the corresponding frequency that you want for to apply on your design.

Knop	**Frequency**
--------	---------------
0	100 MHz
1	80 MHz
2	66 MHz
3	50 MHz
4	40 MHz
5	25 MHz
Else	100 MHz

Table of Figures

References

[CES] Homepage of the Chair for Embedded Systems at the University of Karlsruhe: http://ces.univ-karlsruhe.de/

[Henkel03] Joerg Henkel "Closing the SoC Design Gap"; IEEE Computer Volume 36, Issue 9, September 2003, Pages: 119-121.

[Henkel06] Joerg Henkel "Design and Architectures for Embedded Systems (ESII)", University of Karlsruhe, WS06/07

[ASIPMeister] ASIP Meister Homepage (note: this is the new homepage from the founded company; the old academic homepage no longer exists): http://www.asip-solutions.com/

[HWAFX] Virtex-II FF1152 Prototype Board: http://www.xilinx.com/univ/xupv5-lx110t.htm

[Hennessy96] John L. Hennessy, David A Patterson "Computer Architecture – A Quantitative Approach" 2nd edition 1996; Morgan Kaufmann Publishers, Inc.

[Shaaban01] Muhammad Shaaban "The DLX Architecture", lecture on 'Computer Architecture' http://meseec.ce.rit.edu/eecc551-winter2001/551-12-5-2001.pdf

[Sailer96] Philip M. Sailer, David R. Kaeli "The DLX Instruction Set Architecture Handbook", 1996 Morgan Kaufmann Publishers, Inc.

[DLXsim] Handbook for DLXsim: http://heather.cs.ucdavis.edu/~matloff/dlx.html

[DLX-Package] A composition of the used programs DLXsim, DLXview, DLXgui and DLXcc: http://www.ra.informatik.uni-stuttgart.de/~rainer/Lehre/ROTI01/DLX/

[Kalenteridis04] Kalenteridis et al. "A complete platform and toolset for system implementation on fine-grained reconfigurable hardware", Microprocessors and Microsystems 2004

[XUG002] Xilinx Corp. "UG002: Virtex-II Platform FPGA User Guide" http://www.xilinx.com/support/documentation/user_guides/ug002.pdf

[XDS060] Xilinx Corp. "DS060: Spartan and Spartan-XL Families Field Programmable Gate Arrays" http://www.xilinx.com/support/documentation/data_sheets/ds060.pdf

[FSM] Tutorial for VHDL Final State Machines with synchronous and asynchronous process: www.gaisler.com/doc/vhdl2proc.pdf

[CoSy] Homepage from ACE, the developers of the CoSy compiler: www.ace.nl

[eDIP] Data Sheets for the eDIP LCDs http://www.lcd-module.de/deu/dip/edip.htm