ET4 171 - Processor Design Project Guide

George Voicu
g.r.voicu@tudelft.nl

Changlin Chen
c.chen-2@tudelft.nl

Mihai Lefter
m.lefter@tudelft.nl

Nicoleta Cucu-Laurenciu
n.cuculaurenciu@tudelft.nl

Sorin Cotofana
s.d.cotofana@tudelft.nl

TU Delft Delft University of Technology

Contents

1

Overview

To be able to fulfill the ET4 717 course requirements, i.e., improve the performance of the Plasma baseline implementation and evaluate your proposal, you have to make use of a certain methodology and a set of specific tools. In this section we present an overview of the generic design and implementation methodology and the associated infrastructure.

To accomplish this experiment, the first question we need to address is "What do we need in order to carry it on?"

Starting from one of your targets, e.g., to improve the computational performance of an arithmetic core, you need at least an EDA tool to describe your design by using VHDL. Please note that an EDA tool is not just a text editor for VHDL, even though most of the time you'll be struggling in VHDL, it creates the premises to *translate your VHDL description into "real" hardware.*

Then you need to evaluate your design, to check its correctness and efficiency, so you need a way to simulate your design and to map it into real hardware. Moreover you need benchmarks to evaluate its correctness and performance.

Since an arithmetic core cannot run standalone, you need a supporting platform (in other words, a CPU) to embed it in, and on this platform you can execute the benchmark programs. This platform should be easily implementable in the same way in which your designs are going to be implemented. Concerning the benchmarks, you can develop them by yourself to verify your design, or you can use standard generally acceptable sets to evaluate your design then to compare it with equivalent counterparts. The benchmark programs are usually written in a high level programming language, e.g., C. Generally speaking, the supporting system on which the implementation tools and the benchmark compilation are run has another instruction set than the general-purpose processor for which you develop your project. Thus you need a dedicated compiler (a cross-platform compiler since the target platform is different than the host platform) to compile the benchmark programs for the supporting platform.

Once you get the EDA tools, the supporting processor, implement the required hardware, generate the benchmarks binary via compilation, you can start evaluating your design. If you're lucky enough you can simulate your design and synthesize it without any errors or bugs. In that case you can carry on the evaluation and if the obtained performance is satisfactory you can start writing your report. However this is quite unlikely to happen after the first attempt. Most of the time, you'll get fatal errors at compile time, soft errors at execution time, or some strange errors you don't have a clue where they are coming from. In any of these cases, you need some debug tools, for hardware or software, and/or for both of them.

So far, what we are talking about is just a simple system, which runs bare binary programs. If you want to run more complicated applications, you have to port an operating system on the system.

In section 2 we particularize the above-described generic methodology in the context of the provided project development framework. Afterwards, the supporting Plasma platform is described in section 3.

2

Project Workflow

To design and evaluate your improved Plasma processor you have to synthesize it on an FPGA board. For that you should follow the general implementation flow depicted in Figure 2.1. In addition, following the next suggestions should prove very helpful:

- Make a clear plan for your design before you start writing VHDL code.
- Run a module level simulation every time you modify something in your VHDL code, before you start running a system level simulation.
- Run a system simulation every time you change your design before you synthesize it!
- Make a backup of your source code before you start a major revision on it, or even better, use a version control software tool!

2.1. Setup of the Development Environment

The employed EDA software is installed on any public computer belonging to the CE laboratory, and in consequence accessible via individual CE accounts. Alternatively, for convenience, one may set up the development environment on its personal computer, e.g., by following the subsequent installation and configuration flow.

Prerequisite: TU Delft VPN connection

When running the EDA tools (e.g., Modelsim and Xilinx ISE) outside the TUD Campus, a VPN connection is required, in order to be able to access the licenses. Further information is available at `http://vpn.tudelft.nl/`. A newer version of the Cisco VPN software can be downloaded from here.

2.1.1. Mentor Graphics Modelsim HDL simulator

It is recommended to use Modelsim, for behavioral (pre-synthesis) RTL simulation, and potentially for timing (post-route) simulation, in order to verify that the design is functioning as intended.

- Installation.
  Upon Modelsim installation has completed, two folders are created: `modeltech64_10.1c` (if the default path settings are used during installation), and the install folder `MentorGraphics`. In the following, we assume an installation of Modelsim SE 10.1c. For other versions, substitute the version in the install folder with your version. Note that licenses for the latest 10.3 version are not available on TUD Campus.

- License setup.
  As concerns the licensing, the `LM_LICENSE_FILE` user environment variable is required to be set, as follows:
  - Access the environment variables by right-clicking on `My Computer`⟶`Properties`, click `Advanced system settings`⟶`Environment Variables`.
  - Under *User variables*, click *New* and enter `LM_LICENSE_FILE` in the *Variable name* field. For the variable value field, enter `27017@flexserv1.tudelft.nl`. If a variable called `LM_LICENSE_FILE` already exists, make sure to append to the existing values, with semicolon as separator between the multiple values.

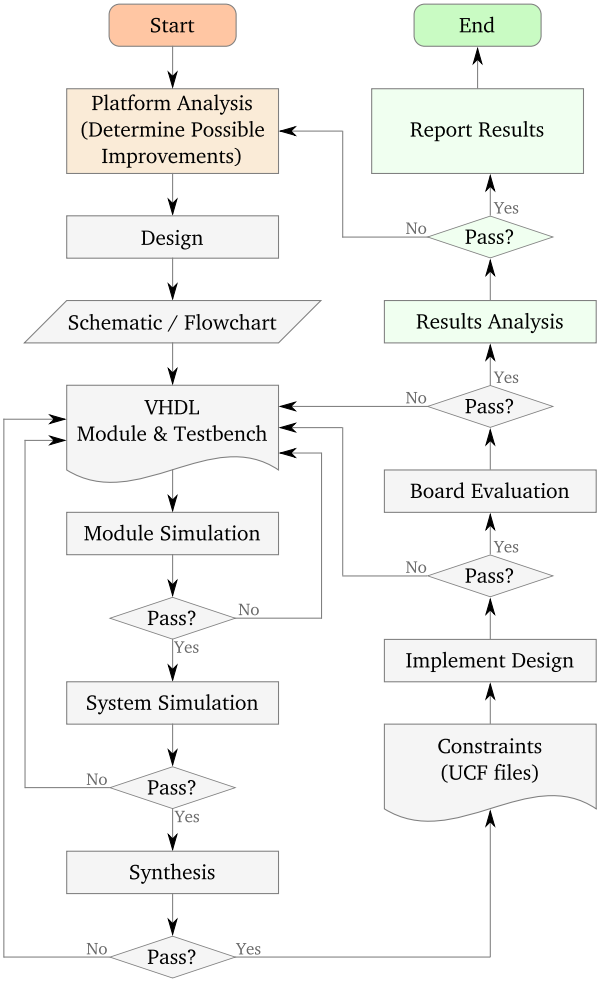

Figure 2.1: Flow.

2.1.2. Xilinx ISE

- Download and installation.
  It is recommended to download and install the latest version of Xilinx ISE Design Suite, specifically version 14.7, which is available at `http://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/design-tools.html`. *Note:* During the installation process, when prompted to select the ISE edition to be installed, choose *ISE Design Suite*, as it is the minimal installation which supports the Virtex 4 XC4VFX60 FPGA present on the ML410 board used for this project.

- License setup.
  After the installation has finished, the Xilinx License Configuration Manager will automatically open. In the *Acquire a License* tab, select the option *Get My Purchased License(s)*. Subsequently, in the *Manage Licenses* tab, append to the LM_LICENSE_FILE user environment variable field the following license servers: `29002@flexserv11.tudelft.nl`; `29002@flexserv12.tudelft.nl`; `29002@flexserv13.tudelft.nl`, with semicolon `;` as separator between the multiple locations/licenses. Then click *Set* and close.

2.1.3. Compile and map HDL Xilinx libraries for Modelsim

The preponderant corpus of a design RTL code is using technology independent constructs, for design portability and simulation speed considerations. However, in certain situations, it is necessary/beneficial to instantiate device specific (e.g., Virtex 4) components in order to achieve the desired functionality. Specifically, for the present project purpose, the following library(ies) shall be employed:

- UNISIM library - for functional simulation of Xilinx primitives (e.g., clock components such as buffers and digital clock managers, RAM components such as RAMB16_S9), and potentially
- XilinxCoreLib - for functional simulation of IP cores created with the Xilinx Core Generator software.

Prior to simulating in Modelsim a design which employs such libraries, two steps are required: (i) the libraries must be compiled for the HDL simulator (i.e., for Modelsim), and (ii) the mapping between the compiled logical design libraries and their physical locations on the disk has to be defined in the initialization file of Modelsim (i.e., the `modelsim.ini` file).
For the compiled libraries mappings to be updated automatically - in the `modelsim.ini` file - after the libraries compilation step, environment variables are required to be set-up *prior to the libraries compilation step*, as follows:

- The first step is to change the write permissions of the `modelsim.ini` file. Browse to the ModelSim install directory (i.e., `C:\modeltech64_10.1c`), right-click on `modelsim.ini`, select *Properties* and uncheck *Read-only*.
- Make a backup of the `modelsim.ini` file for safety reasons.
- Access the environment variables by right-clicking on `My Computer`⟶`Properties`, click `Advanced system settings`⟶`Environment Variables`.
- Under *System variables*, click *New* and enter `modelsim` in the *Variable name* field. For the variable value field, enter `C:\modeltech64_10.1c\modelsim.ini`.
- In a similar manner, add another variable named `Path` with the value `C:\modeltech64_10.1c\win64`. If a variable called `path` already exists, then make sure you append to the existing values (note that multiple values can be separated by semicolon).

After having set the environment variables, the libraries compilation process can be started. To this end, for convenience, two options are presented: a) compilation via GUI mode, or *equivalently* b) compilation via command line mode.
a) Libraries compilation via GUI

- Open the *Simulation Library Compilation Wizard*. This can be accessed from `Start Menu` ⟶ `Xilinx Design Tools` ⟶ `ISE Design Suite 14.7` ⟶ `ISE Design Tools` ⟶ `64/32-bit Tools`.
- The *Select Simulator* window opens up. Select the appropriate simulator (Modelsim SE, 64/32 bits), and unless filled in by default, enter as follows:
  - Simulator Executable Location: e.g., `C:\modeltech64_10.1c\win64`
  - Configuration File: `compxlib.cfg`
  - Compxlib Log File: `compxlib.log`
- Next select the HDL used for simulation. It is recommended to select both VHDL and Verilog.
- Then select the Virtex 4 device family.
- The next window is for *Selecting Libraries for Functional and Timing Simulation*. Different libraries are required for different types of simulation (behavioral, post-route, etc.). It is recommended to select All Libraries as the default option.
- Finally, the window for *Output Directory for Compiled Libraries* is shown. It is recommended to leave the default values that Xilinx provides. Then select *Launch Compile Process*.

b) Libraries compilation via command line Alternatively, instead of using the GUI mode (option a)), one may opt for the command line functionally equivalent counterpart, whose steps are described subsequently:

- Open a command prompt. In the command window change the directory to the location of the `compxlib.exe`: (for 32-bit systems, change to directory `nt` instead)

```
cd C:\Xilinx\14.7\ISE_DS\ISE\bin\nt64
```

- Invoke `compxlib` with the switches as follows:

```
compxlib -s mti_se -64bit -l all -arch virtex4 -lib all -w -exclude_superseded -verbose
```

The library compilation process may contain a lot of warnings, but it should be error-free. Upon compilation completion, the mapping between the logical design libraries and their physical locations should be automatically updated in the `modelsim.ini` file, e.g.:

```
[LIBRARY]
...
unisim = C:/Xilinx/14.7/ISE_DS/ISE/vhdl/mti_se/10.1c/nt64/unisim
xilinxcorelib = C:/Xilinx/14.27/ISE_DS/ISE/vhdl/mti_se/10.1c/nt64/xilinxcorelib
...
```

2.2. Functional Verification

To verify the correctness of your design, you have to run both module level and system level simulation. For module level verification, you have to write a testbench by yourself. For system level verification, you have a top level testbench described in the `sim/sim_tb_top.vhd` file.

To run system level simulation open Modelsim, change the working directory to the `sim` folder in the provided lab package. A *sim.do* file is offered to compile the source code and start the simulation process. Type the following command in the Modelsim console:

```
do sim.do
```

The commands in *sim.do* will be executed line by line to compile the source codes and invoke the simulation. The application run on the processor is selected by configuring the *sdramfile* constant in the testbench file, with values equal to files in the `sim/ddr_content/` folder.

Note that the simulation time is quite long for the evaluation benchmarks (see Section 2.4.2). You can check the information in the Modelsim console or the UART output.txt file to track the simulation progress. If you want to resume the simulation after a stop, use command:

```
run -all
```

or,

```
run 5000000
```

to run another 5000ns (simulation time). Note that the simulation timescale is 1ps.

If you create a new module, you have to add it to the *sim.do* file. For example, if your new module is called my_design.vhd, you need to add:

```
vcom -work work path_to_your_design/my_design.vhd
```

to the *sim.do* file. Your design should be compiled before the higher level module in which your design is instantiated. This means you should place this command before the one that compiles the higher level module.

The waveforms of signals in the `plasma_top` module and `ddr_ctrl_top` are illustrated by default. You can add signals you want to monitor directly to the script in a similar manner, or using the Modelsim menu (please refer to the Modelsim documentation).

If your design passes this verification then you can continue to further steps, otherwise you have to repeat the design flow in order to correct your design.

2.2.1. Custom testbenches

The assembly code corresponding to a test of all ISA opcodes is provided in `benchmarks/opcodes.asm` in order to facilitate the design debugging. We suggest to first verify (using behavioral RTL simulation) the design functionality using the opcodes test, as it allows for easier faults isolation, and faster overall debugging, when compared to the evaluation benchmarks case.

For custom testbench developement, a MIPS cross-compiler and assembler are provided in the `compiler` folder. In the `benchmarks/opcodes.asm` folder you can also find a sample Makefile.

2.3. FPGA Implementation

To synthesize and implement your design, open the Xilinx ISE project file in `plasma_ISE`. The "Processes" panel (in the middle left of the screen) contains all the available implementation steps. Select the top design file (`top_ml410`) in the upper left panel. By double-clicking the "Generate Programming File" in the "Processes" panel, all the implementation steps are automatically executed. Upon a successful completion the bitstream file `top_ml410.bit` is available in this folder.

The current ISE project is created with Xilinx ISE 14.7. If you are using a different version, you can just migrate the project or you can create your own ISE project by following the next procedure.

- File → New project. The *Evaluation Development Board* should be "Virtex 4 ML410 Evaluation Platform". Choose your preferred language, i.e., VHDL. Finish the New Project Wizard.
- Add all the files from the `rtl` folder to your project.
- Add the constraint file `plasma_ISE/ml410.ucf` to your design.

2.4. Evaluation

2.4.1. Timing analysis

To check that the implemented design works at the requested frequency double-click in ISE on the "Analyze Post-Place & Route Static Timing" in the "Processes" panel. A report opens up detailing all timing constraints (usually one for each clock signal in the design) with the top slowest propagation paths and any potential timing violations.

2.4.2. Benchmarks

To evaluate the impact of your improvements on the Plasma platform performance you are given a set of pre-built benchmarks that run remotely on the FPGA board. After benchmark execution on the board the message "CORRECT!" is displayed if the benchmark results are as expected. Otherwise, an "ERROR" message is displayed. In addition, to evaluate the performance of your solution, the number of CPU cycles (in million cycles) consumed by the benchmark execution is measured and displayed. The benchmarks and their execution cycle count (in million cycles) are given in Table 2.1. The C source code and assembly listings can be found in the `benchmarks` folder.

Table 2.1: Benchmark description.

Name	Baseline Performance	Description
opcodes	for verification only	Tests all MIPS I instructions
cjpeg	28.170021	JPEG compression
divide	271.132626	Large number (192Kb) integer division using GMP library
multiply	152.532326	Large number (64Kb) integer multiplication using GMP library
pi	845.357613	Computes 1000 digits of PI using basic arithmetic operations
rsa	560.226687	RSA message signing using GMP library
ssd	798.280788	Pattern matching using Sum-of-Squared-Differences
ssearch	464.961143	String search using look-up tables
susan	783.253743	Gaussian image smoothing
bench_all	3996.180351	All Benchmarks in one run

2.4.3. Remote FPGA verification

The remote access flow for verifying the FPGA implemented design consists of the following steps:

- Dropbox folder setup.
  The communication of input/output files for remote FPGA access, will be performed via Dropbox shared folders. For each group, a shared folder will be created.
  Prerequisites: In order to access the group folder, you are required to have Dropbox (www.dropbox.com) account(s) and provide us with the email address(es) linked to your Dropbox account(s).

  Based on provided emails, shared folders invitations will be sent to the group members; upon the invitations acceptance, the group shared folder will be mapped to the individual Dropbox accounts. The shared folder can then be accessed for upload/download either locally (if the Dropbox client is locally installed), or online via the Dropbox website.

- Place the .bit file(s) in the Dropbox folder.
  Bitstream consistency requirement: every bitstream must be named identical to the name of the benchmark to be executed (e.g., bitstream **susan.bit** for target benchmark **susan**).

- The FPGA will be programmed automatically with the bitstream, after which the bitstream will be automatically deleted from the Dropbox folder. Subsequently, the binary image of the targeted benchmark will be sent automatically via UART to the board and loaded in the DDR memory.

  Note: The scheduling time for programming each bitstream on the FPGA is compliant with a round robin scheduling policy, relative to the remaining Dropbox groups' folders and the individual bitstream files timestamp. Thus placing at the same time multiple bitstream files (corresponding to different benchmarks) in the Dropbox folder, will not necessarily result in their consecutive programming on the FPGA.

- For each programmed bitstream, two output files will be generated in the Dropbox folder:
  - *a .txt file with the benchmark execution related results* (e.g., status of the benchmark results correctness, figures of merit), and
  - *a simulation log .txt file* consisting of ERRORS/INFO/WARNINGS concerning the status of the entire simulation (e.g., bitstream consistency, FPGA programming, UART receive/transmit).

  Both output files will be named relative to the benchmark name (e.g., **susan.txt** and **susan_log.txt** for target benchmark **susan**).

2.4.4. Power Consumption Evaluation

This section describes the steps that need to be followed to get an estimation of the power consumption of the Plasma Platform on an FPGA. The power consumption of the platform can be estimated with the help of both Modelsim and ISE.

Signal Dump in Modelsim

Before you can estimate power in ISE you must first create a signal dump with Modelsim, i.e., a .vcd file that contains the transition values of monitored signals. For this you can use the Modelsim project and execute a script with the following commands:

```
1  vsim -L unisim work.sim_tb_top
2  vcd file power.vcd
3  vcd add -r sim_tb_top/*
4  run 2ms
5  quit -sim
```

You can place this script in the simulation directory. If you name the script power.do, just type do power.do in the command line of Modelsim to execute it. It may take both time and disk space to finish. The longer the simulation time, the more accurate the results is, but at the expense of increased filesize (may reach in the order of GBs).

The meaning of the commands is as follows:

- The vsim command specifies the top-level unit of the design in order to initialize the simulation.
- The vcd command specifies how the .vcd file should be called.
- The vcd add instructs which signals should be monitored.

It should be noted that in principle you can export any waveform in .vcd format. A good suggestion for you is that you do power evaluation both on top level and module level, which will give you a good way to understand how much power/energy involves in your design and what is the fraction it contributes to the entire system.

Power Estimation in ISE

Before you can estimate power be sure that your design is fully implemented, i.e., all steps up until, and including "Place & Route" are performed (double click Processes → Implement Design).

Go to the Processes panel and expand Implement Design and then Place & Route. Click Analyze Power Distribution (XPower Analyzer). This will open the XPower utility, with the .ncd file from your ISE project already loaded. Thus, the estimated values are less accurate as the switching activity from the .vcd file is not taken into account.

To perform an accurate power estimation go to File → Open Design. Select `top_ml410.ncd` as the design file and for the Simulation file select the in Modelsim generated `power.vcd` file. XPower will analyze the file and the dynamic power consumption will appear in the appropriate cells of the power report (inside XPower).

2.5. How to Report the Results

Your report should have a clear flow from the beginning to the end. Sections should nicely follow each other logically with connecting links from one to the other.

In the introduction you should include a short summary of the entire work:

- Motivation for your approach: what is the general idea behind your optimization proposal, how have you decided to make the changes to the core, and why those and no other changes;
- What are the changes that you have performed to the core, together with their implications;
- What results have you obtained;
- What are your conclusions.

This should be really a summary, not detailed. At the end of the introduction add also the organization (outline) of the report.

Next, it would be nice to include a separate section in which you motivate your choices for improvement - much more descriptive and comprehensive than what you said in the introduction. Detail:

- How did you analyze the baseline processor in order to see its weak points;
- What were your findings based on which you have decided to make the improvements;
- What do you expect from implementing your changes.

Next, a description of all the performed changes should follow. Here it is important that you first motivate your choices. Let us presume that you chose to improve the multiplier. Before you go into details regarding the architecture and the implementation of the improved version, you should include a small survey of the different types of multipliers and present the reasons for your choice. Do not forget to refer to the literature. In the actual description include figures to present the architecture of the new blocks that you introduce in the design. Start from the top level and go down to each important component, i.e., component that you have implemented in a special manner, in such a way that it is clear where it is placed and why.

Handling the interface signals properly is important for you to make the processor with new designs working. So please describe how you integrate your design into the processor. You can draw the action of handshaking signals in the form of timing chart or state machine. Discuss also the verification aspects

of the designs: set of testbenches that cover a wide range of possibilities, test coverage. Attach the system simulation result to show that your design works in Modelsim.

The most important part of the report is the analysis.

Summarize your work and add your conclusions and possible future work plans in a final section. You should also put things into perspective and include the following:

- what was your initial plan;
- what were your expectations;
- what are the results and why are the results not according to the expectations in case they are not.

Conclusions can also include some feedback to help us to improve the project for the next generations (not mandatory but it is more than welcome).

The report should also be consistent, in structure, language and formatting styles. The entire report should be at the present tense, with the exception of the conclusion section, where past tense should be employed. In a technical report the headings are written with capitals, and usually numbered to be easier to read it and reference parts of it. Also, the text is usually justified. The references should be placed combined at the end of the entire report.

3

Plasma Platform

The Plasma platform (see Figure 3.1) is a minimal System-on-a-Chip (SoC) design written in VHDL which consists of the following components:

- a 32-bit MIPS processor core - Plasma CPU;
- an unified 4-KB cache;
- a 4-KB boot memory;
- a DDR SDRAM controller;
- an Universal Asynchronous Receiver/Transmitter (UART) unit.

The platform is based on the open-source Plasma project [1] available on the OpenCores website. The original design was tailored for the Xilinx ML410 board (Revision E) [2], depicted in Figure 3.2. Technical details about all the available hardware on the board can be found via the following link [3].

The use scenario of the SoC in the remote benchmark evaluation procedure is the following:

1. The board is programmed with the generated SoC bitstream.
2. A boot loader software which resides in the `boot_ram` starts execution on the Plasma CPU after reset. Its purpose is to receive on the UART the application (the benchmark binary image) to be executed, place it in the DDR memory, and redirect the execution to the address that points to the beginning of the application.
3. The application executes and sends the output on the UART.
4. The remote access server captures the application output from UART and saves it in a file.

Thus, do not modify the `boot_ram` component, otherwise the remote evaluation on the FPGA board will not function properly.

3.1. Plasma CPU

The Plasma CPU is a small synthesizable 32-bit RISC microprocessor that executes all MIPS I [4] user mode instructions except unaligned load and store operations. In the code the top module of the CPU is represented by the `mlite_cpu` unit (see Figure 3.3). The components are briefly described next.

3.1.1. Program Counter Generation

The `pc_next` unit (see Figure 3.4) generates the address of the next instruction on its *pc_future* output port. The value of *pc_future* can either:

- be the incremented value of the previous program counter (stored locally in *pc_reg*);
- come directly from the `mem_ctrl` unit (on the *opcode25_0* input port), in case of an unconditional branch;
- be computed by the `alu` unit and received on the *pc_new* input port, in case of a conditional branch.

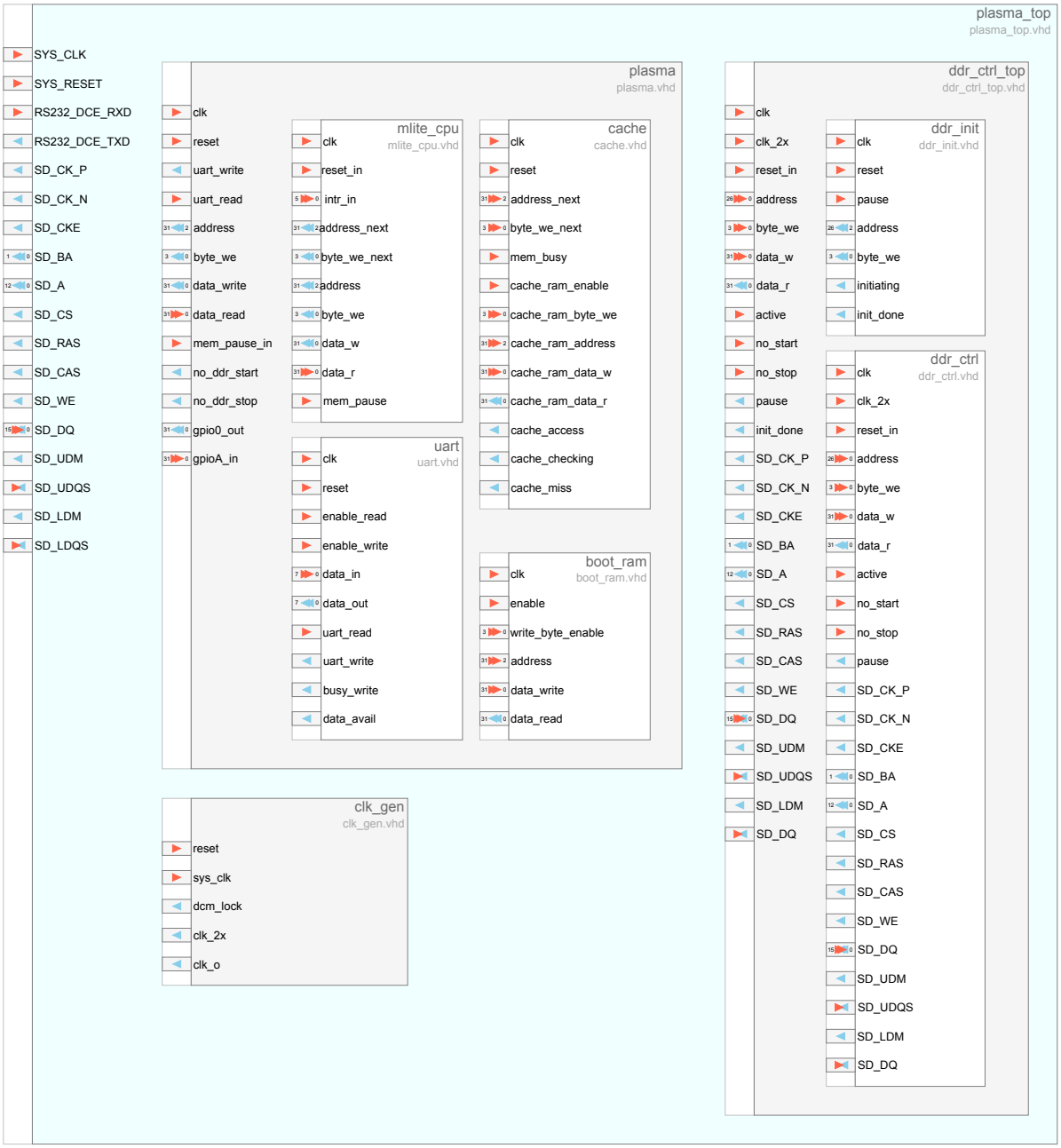

Figure 3.1: Plasma SoC Platform.

The selection is performed by the *pc_source* signal, generated by the control unit. In case of a conditional branch, the *take_branch* signal is also utilized in the selection: *pc_future* takes the *pc_new* value when *take_branch* is '1', and the previously incremented program counter values when *take_branch* is '0'.

3.1.2. Memory Interface Unit

The mem_ctrl unit (see Figure 3.5) assures the core to memory communication: it sends addresses to and receives data from the memory subsystem. The unit is controlled through the *mem_source* signal, issued by the control unit, to perform one of the following tasks:

- Instructions fetch:
  - it sends the appropriate instruction address, received from the pc_next on the *address_pc*

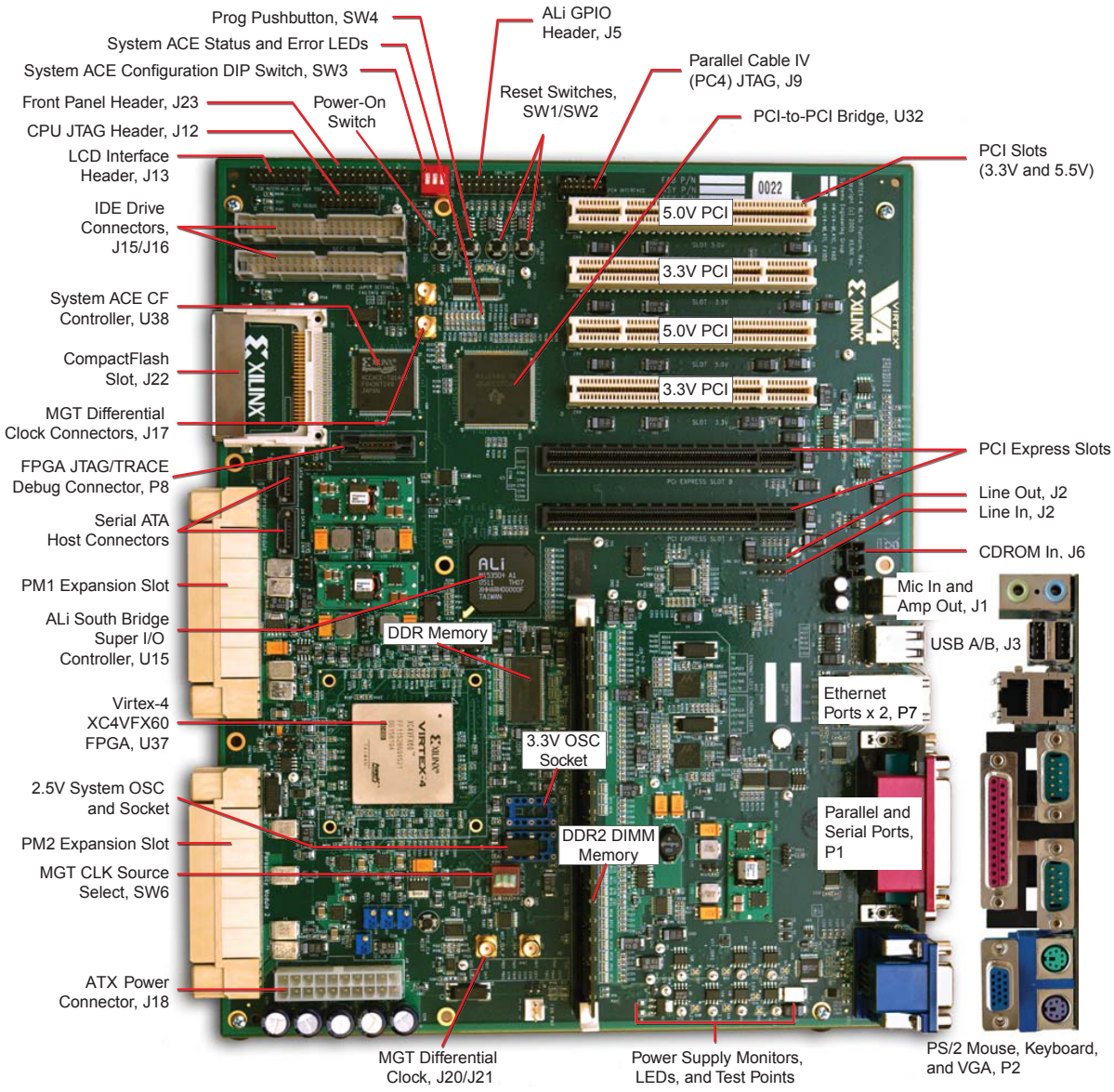

Figure 3.2: Xilinx ML410 Board and Front Panel Detail.

port, to the memory;

- it receives the instruction opcode on its *data_r* port;
- it delivers the instruction opcode to the control unit on its *opcode_out* port.

- Data memory read (for load operations):
  - it sends to the memory the address of the datum to be loaded;
  - it receives the datum from memory and it passes it to the bus_mux unit;
- Data memory write (for store operations):
  - it sends to the memory the datum and the address where to be written.

3.1.3. Decode and Control Unit

The control unit (see Figure 3.6) performs instruction decode, based on which it generates the control signals for all the other units. The instruction encodings are detailed in Figure 3.7.

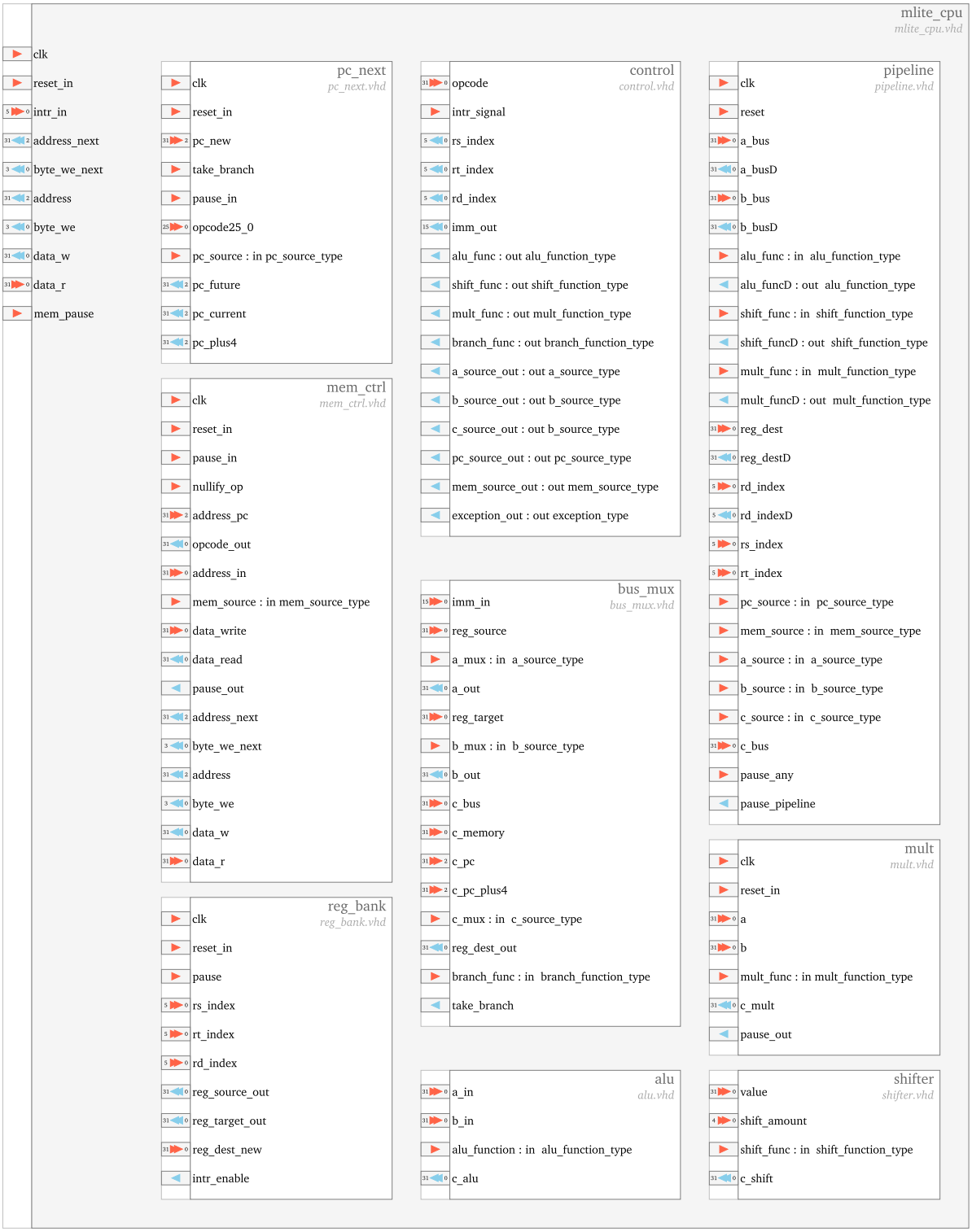

Figure 3.3: Plasma CPU components.

3.1.4. Signal Multiplexing Unit

The main task of the `bus_mux` unit (see Figure 3.8) is to perform the functional units input signals multiplexing. In addition, the `bus_mux` unit also performs the comparison required by the conditional branch instructions, and it generates the branch taken/not taken signal on its *`take_branch`* port.

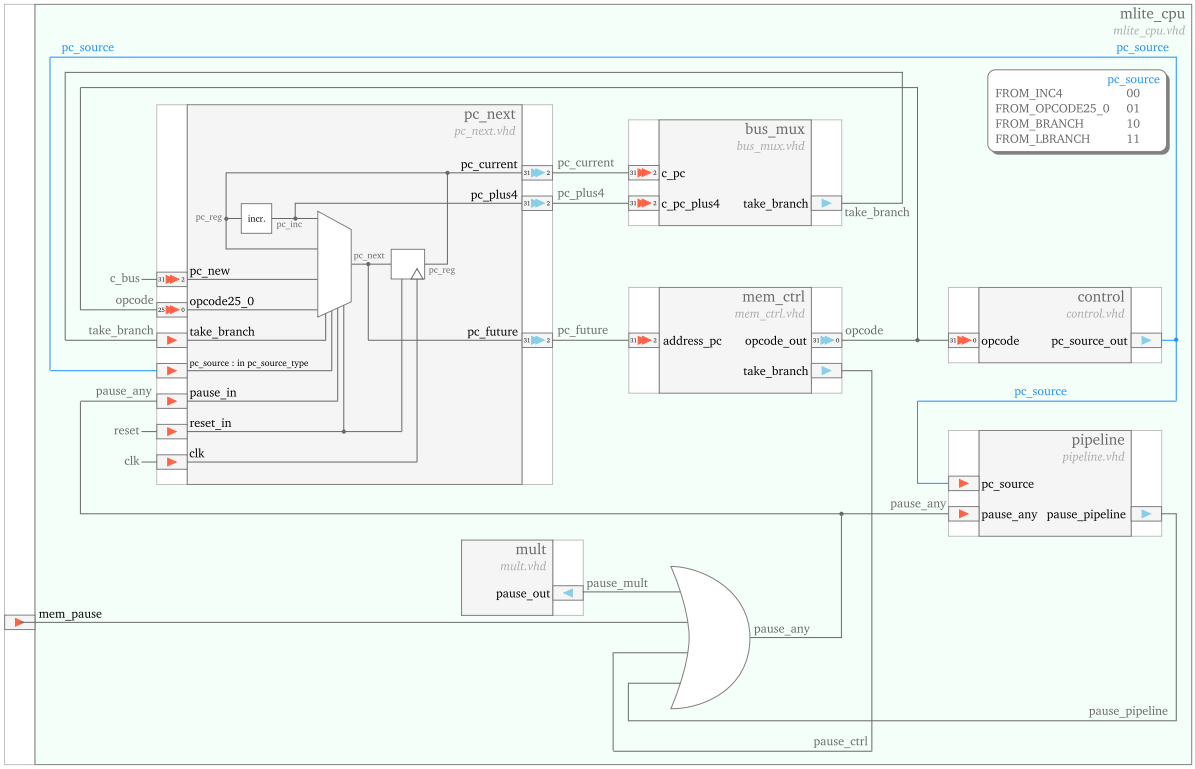

Figure 3.4: Program counter generation.

3.1.5. Register File (Register Bank)

The Plasma CPU is based on the MIPS I instruction set, hence there are 32-bit general purpose registers, each 32-bit wide. From the compiler perspective, each register has a specific function, detailed in Table 3.1. You can use the table to find the mapping between the software register name (the one present in the benchmark assembly listing) and the hardware address of the register in the register bank. The only functions that also need support from the hardware implementation are the following: the value of register R0 is always zero, while R31 is used as the link register to return from a subroutine.

Table 3.1: List of registers.

Register	Name	Function
R0	zero	Always contains 0
R1	at	Assembler temporary
R2-R3	v0-v1	Function return value
R4-R7	a0-a3	Function parameters
R8-R15	t0-t7	Function temporary values
R16-R23	s0-s7	Saved registers across function calls
R24-R25	t8-t9	Function temporary values
R26-R27	k0-k1	Reserved for interrupt handler
R28	gp	Global pointer
R29	sp	Stack Pointer
R30	s8	Saved register across function calls
R31	ra	Return address from function call
HI-LO	hi-lo	Multiplication/division results
PC	Program Counter	Points at 8 bytes past current instruction
EPC	Exception PC	Exception program counter return address

In addition to the general purpose registers there are 4 special registers, also detailed in Table 3.1. HI

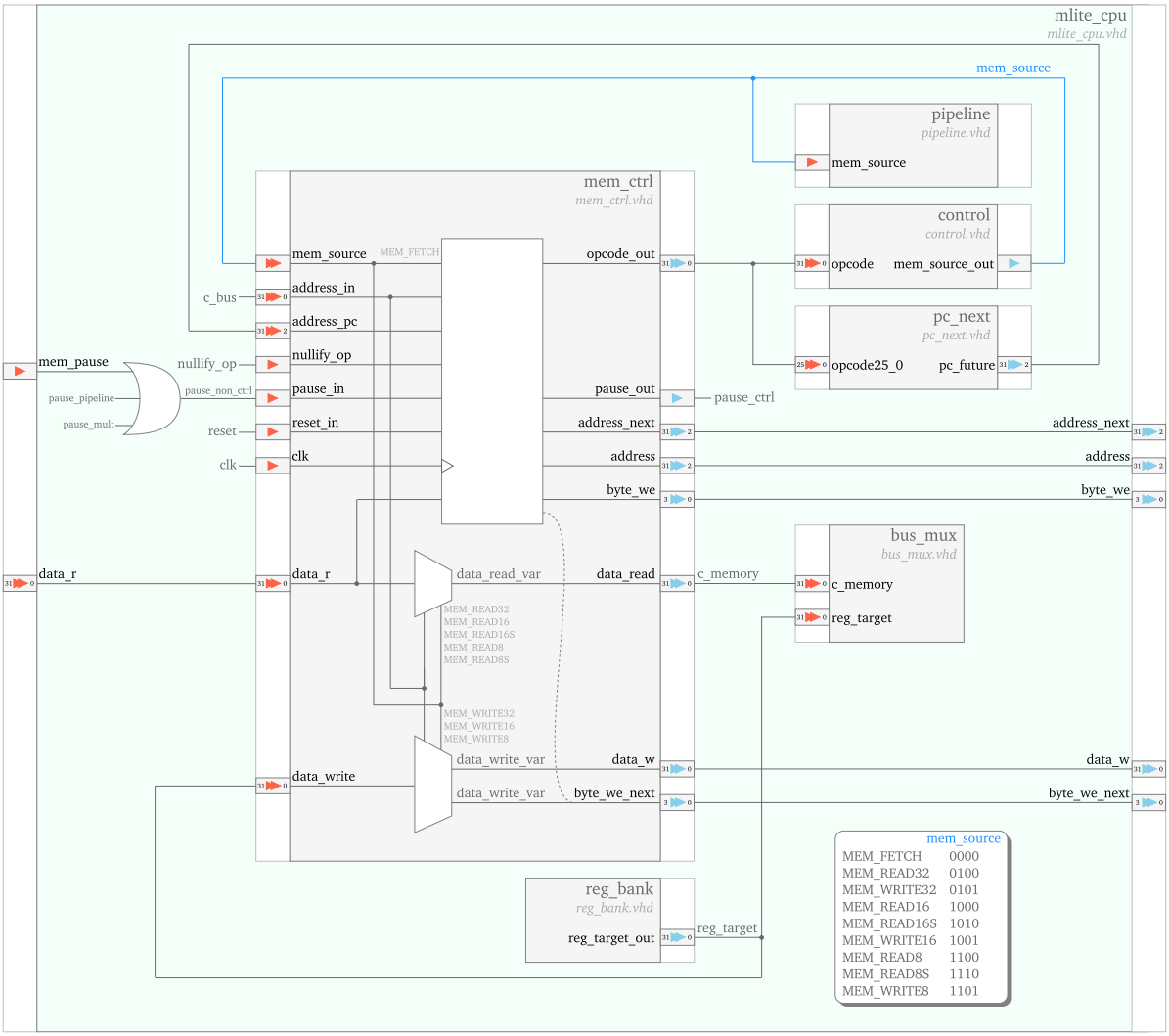

Figure 3.5: Internal CPU Memory Controller.

and LO registers contain the 32-bit MSB and LSB part, respectively, of a 64-bit multiplication/division result. The Program Counter (PC) specifies the address of the next instruction in the program. The Exception Program Counter (EPC) register remembers the program counter when there is an interrupt or exception. There is no status register. Instead, the results of a comparison set a register value, and the branch then tests this register value.

The interconnection of the `reg_bank` unit with the rest of the units is pictured in Figure 3.9. It is implemented using four Xilinx RAM16X1D [5] dual port memories.

3.1.6. Pipeline

The `pipeline` unit (see Figure 3.10) contains the pipeline registers (flip-flops) that delay the inputs of the functional units and of the register file write port. Other separation registers between pipeline stages are placed in their corresponding modules. An example for an instruction flow starting with the fetch and ending with the result commit is depicted in Figure 3.11.

3.2. Cache

A 4-KB unified cache is present in the memory hierarchy of the Plasma platform, in the `cache` component. The cache is direct mapped, has a block size of 32 bits and caches only the lowest 2-MB of the main memory. It works at the same frequency as the processor and has a hit latency of 1 cycle.

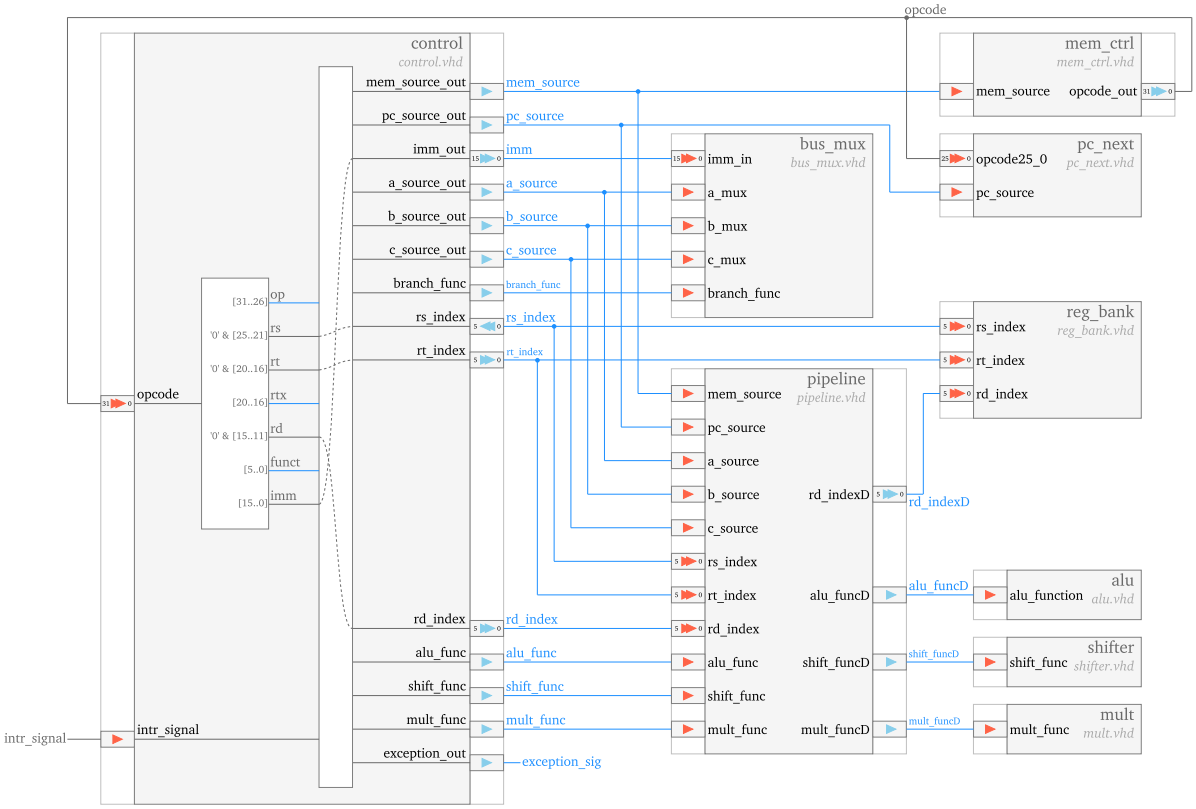

Figure 3.6: Decode and Control Logic Unit.

3.3. DDR Controller

The Plasma platform has access to a MT46V16M16-5B 32MB DDR SDRAM memory chip through the DDR controller contained in the `ddr_ctrl` component. For every 32-bit request issued by the processor, the DDR controller issues a request with a burst length of 2, thus transferring 2 16-bit words from/to the DDR memory. The memory initialization routine is performed by the `ddr_init` component.

3.4. Clock Generation

All the platform components use the same clock signal. In addition, the DDR controller needs another clock with a doubled frequency, to read/write data from/into memory at double data rates. The internal platform clock is generated by the `clk_gen` unit in the following manner. The Xilinx proprietary DCM (Digital Clock Manager) [5] from the *DCM_BASE0* instance synthesizes the *CLKFX* clock signal from the ML410 100MHz on-board oscillator. The frequency of the *CLKFX* signal is determined by the *CLKFX_DIVIDE* and *CLKFX_MULTIPLY* parameters. This clock signal is used as the double clock for the DDR controller and at the same time is further divided with a factor of 2 to obtain the internal platform clock.

The UART component uses a constant value, *COUNT_VALUE*, to ensure that the UART transmission speed is 230400bps. This value is dependent on the processor core internal clock frequency by this relation: *COUNT_VALUE* = core frequency (Hz) / 230400Hz.

	Opcode	Name	Action	Opcode bitfields					
Arithmetic Logic Unit	ADD rd,rs,rt	Add	rd=rs+rt	000000	rs	rt	rd	00000	100000
	ADDI rt,rs,imm	Add Immediate	rt=rs+imm	001000	rs	rt	imm		
	ADDIU rt,rs,imm	Add Immediate Unsigned	rt=rs+imm	001001	rs	rt	imm		
	ADDU rd,rs,rt	Add Unsigned	rd=rs+rt	000000	rs	rt	rd	00000	100001
	AND rd,rs,rt	And	rd=rs&rt	000000	rs	rt	rd	00000	100100
	ANDI rt,rs,imm	And Immediate	rt=rs&imm	001100	rs	rt	imm		
	LUI rt,imm	Load Upper Immediate	rt=imm<<16	001111	rs	rt	imm		
	NOR rd,rs,rt	Nor	rd=~(rs|rt)	000000	rs	rt	rd	00000	100111
	OR rd,rs,rt	Or	rd=rs|rt	000000	rs	rt	rd	00000	100101
	ORI rt,rs,imm	Or Immediate	rt=rs|imm	001101	rs	rt	imm		
	SLT rd,rs,rt	Set On Less Than	rd=rs<rt	000000	rs	rt	rd	00000	101010
	SLTI rt,rs,imm	Set On Less Than Immediate	rt=rs<imm	001010	rs	rt	imm		
	SLTIU rt,rs,imm	Set On	rt=rs<imm	001011	rs	rt	imm		
	SLTU rd,rs,rt	Set On Less Than Unsigned	rd=rs<rt	000000	rs	rt	rd	00000	101011
	SUB rd,rs,rt	Subtract	rd=rs-rt	000000	rs	rt	rd	00000	100010
	SUBU rd,rs,rt	Subtract Unsigned	rd=rs-rt	000000	rs	rt	rd	00000	100011
	XOR rd,rs,rt	Exclusive Or	rd=rs^rt	000000	rs	rt	rd	00000	100110
	XORI rt,rs,imm	Exclusive Or Immediate	rt=rs^imm	001110	rs	rt	imm		
Shifter	SLL rd,rt,sa	Shift Left Logical	rd=rt<<sa	000000	rs	rt	rd	sa	000000
	SLLV rd,rt,rs	Shift Left Logical Variable	rd=rt<<rs	000000	rs	rt	rd	00000	000100
	SRA rd,rt,sa	Shift Right Arithmetic	rd=rt>>sa	000000	00000	rt	rd	sa	000011
	SRAV rd,rt,rs	Shift Right Arithmetic Variable	rd=rt>>rs	000000	rs	rt	rd	00000	000111
	SRL rd,rt,sa	Shift Right Logical	rd=rt>>sa	000000	rs	rt	rd	sa	000010
	SRLV rd,rt,rs	Shift Right Logical Variable	rd=rt>>rs	000000	rs	rt	rd	00000	000110
Multiplier/Divider	DIV rs,rt	Divide	HI=rs%rt; LO=rs/rt	000000	rs	rt	0000000000		011010
	DIVU rs,rt	Divide Unsigned	HI=rs%rt; LO=rs/rt	000000	rs	rt	0000000000		011011
	MFHI rd	Move From HI	rd=HI	000000	0000000000		rd	00000	010000
	MFLO rd	Move From LO	rd=LO	000000	0000000000		rd	00000	010010
	MTHI rs	Move To HI	HI=rs	000000	rs	000000000000000			010001
	MTLO rs	Move To LO	LO=rs	000000	rs	000000000000000			010011
	MULT rs,rt	Multiply	HI,LO=rs*rt	000000	rs	rt	0000000000		011000
	MULTU rs,rt	Multiply Unsigned	HI,LO=rs*rt	000000	rs	rt	0000000000		011001
Branch	BEQ rs,rt,offset	Branch On Equal	if(rs==rt) pc+=offset*4	000100	rs	rt	offset		
	BGEZ rs,offset	Branch On >= 0	if(rs>=0) pc+=offset*4	000001	rs	00001	offset		
	BGEZAL rs,offset	Branch On >= 0 And Link	r31=pc; if(rs>=0) pc+=offset*4	000001	rs	10001	offset		
	BGTZ rs,offset	Branch On > 0	if(rs>0) pc+=offset*4	000111	rs	00000	offset		
	BLEZ rs,offset	Branch On	if(rs<=0) pc+=offset*4	000110	rs	00000	offset		
	BLTZ rs,offset	Branch On	if(rs<0) pc+=offset*4	000001	rs	00000	offset		
	BLTZAL rs,offset	Branch On	r31=pc; if(rs<0) pc+=offset*4	000001	rs	10000	offset		
	BNE rs,rt,offset	Branch On Not Equal	if(rs!=rt) pc+=offset*4	000101	rs	rt	offset		
	BREAK	Breakpoint	epc=pc; pc=0x3c	000000	code				001101
	J target	Jump	pc=pc_upper|(target<<2)	000010	target				
	JAL target	Jump And Link	r31=pc; pc=target<<2	000011	target				
	JALR rs	Jump And Link Register	rd=pc; pc=rs	000000	rs	00000	rd	00000	001001
	JR rs	Jump Register	pc=rs	000000	rs	000000000000000			001000
	MFC0 rt,rd	Move From Coprocessor	rt=CPR[0,rd]	010000	00000	rt	rd	00000000000	
	MTC0 rt,rd	Move To Coprocessor	CPR[0,rd]=rt	010000	00100	rt	rd	00000000000	
	SYSCALL	System Call	epc=pc; pc=0x3c	000000	00000000000000000000				001100
Memory Access	LB rt,offset(rs)	Load Byte	rt=*(char*)(offset+rs)	100000	rs	rt	offset		
	LBU rt,offset(rs)	Load Byte Unsigned	rt=*(Uchar*)(offset+rs)	100100	rs	rt	offset		
	LH rt,offset(rs)	Load Halfword	rt=*(short*)(offset+rs)	100001	rs	rt	offset		
	LBU rt,offset(rs)	Load Halfword Unsigned	rt=*(Ushort*)(offset+rs)	100101	rs	rt	offset		
	LW rt,offset(rs)	Load Word	rt=*(int*)(offset+rs)	100011	rs	rt	offset		
	SB rt,offset(rs)	Store Byte	*(char*)(offset+rs)=rt	101000	rs	rt	offset		
	SH rt,offset(rs)	Store Halfword	*(short*)(offset+rs)=rt	101001	rs	rt	offset		
	SW rt,offset(rs)	Store Word	*(int*)(offset+rs)=rt	101011	rs	rt	offset		

Figure 3.7: Plasma Instruction Encodings.

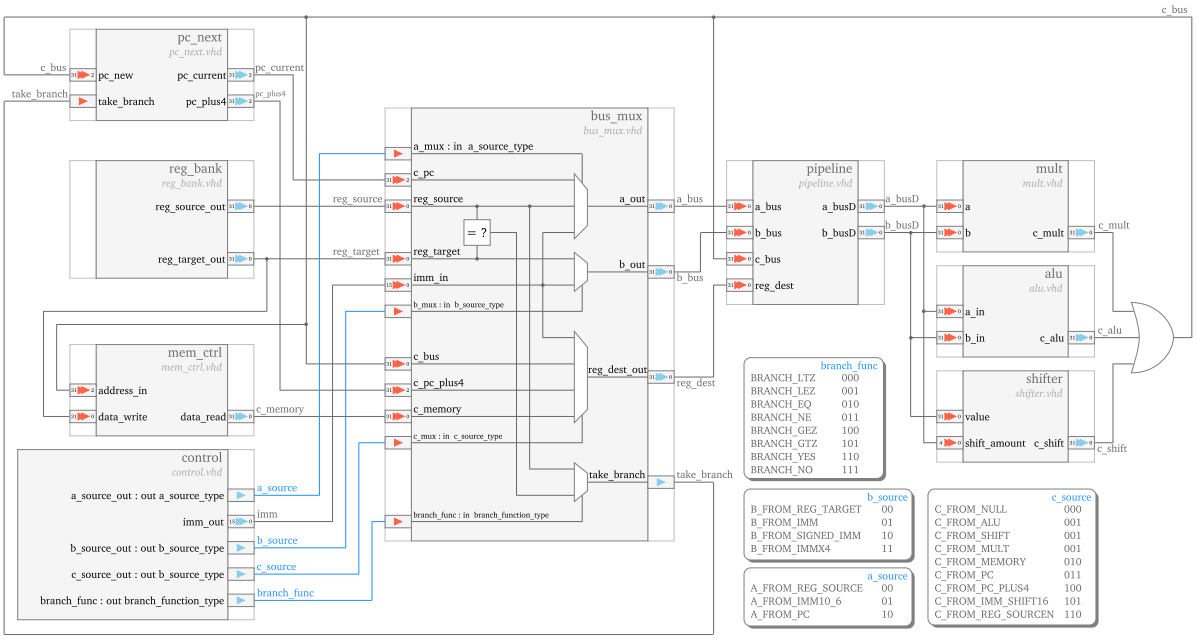

Figure 3.8: Internal CPU bus.

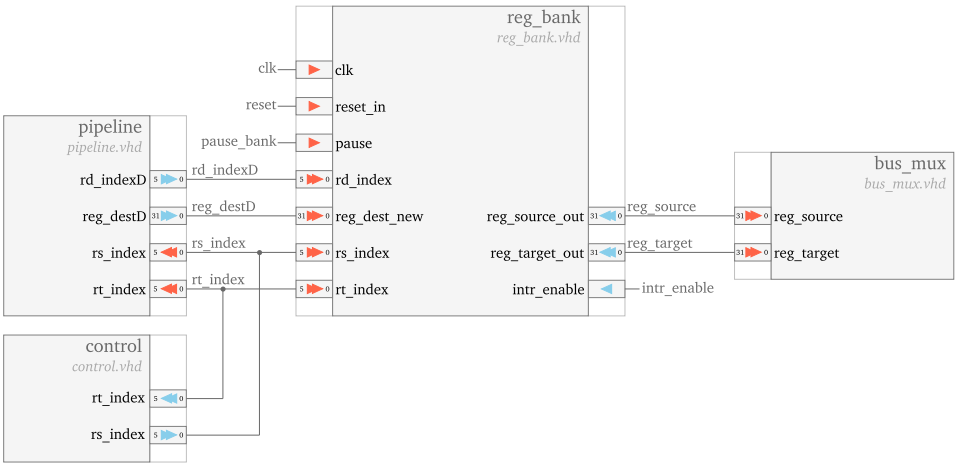

Figure 3.9: Register file interconnection.

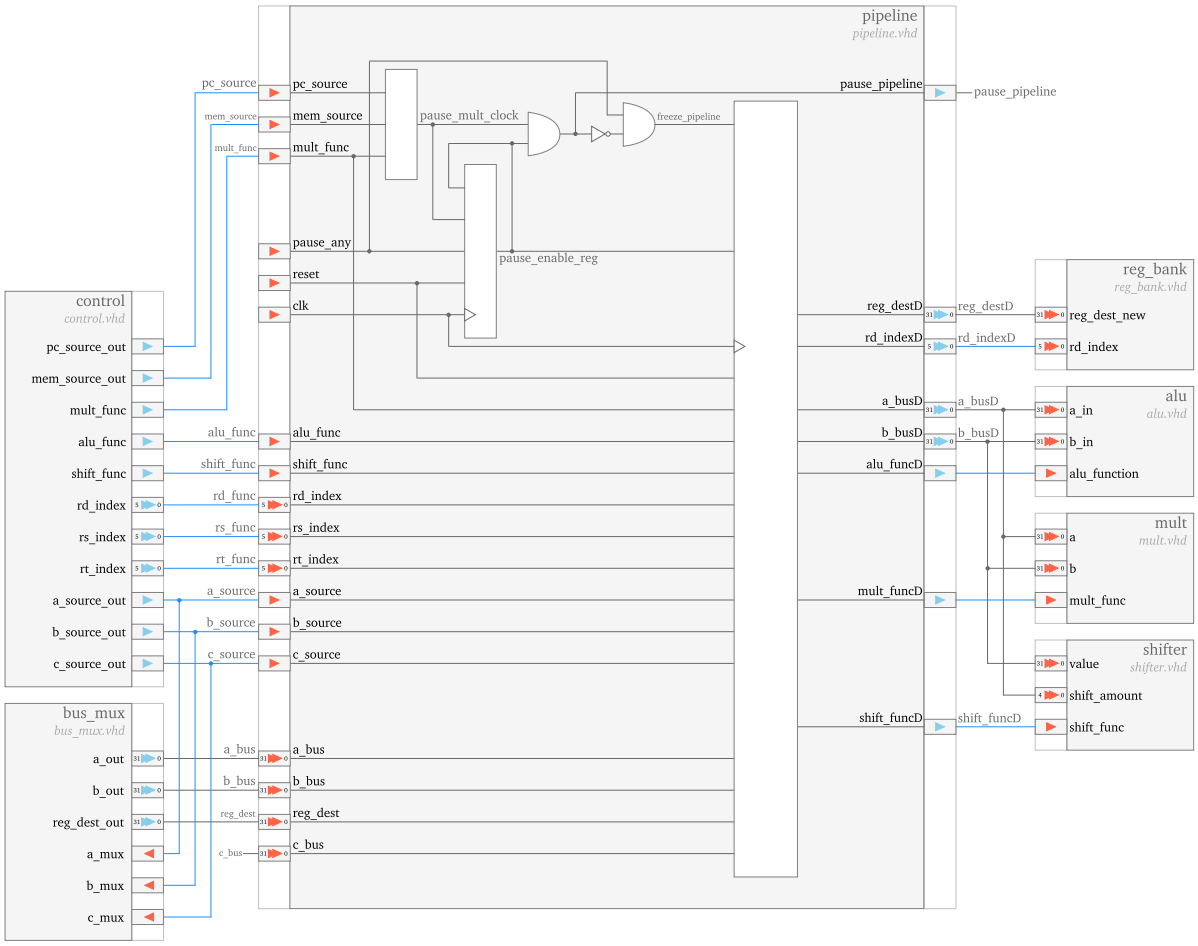

Figure 3.10: Pipeline entity.

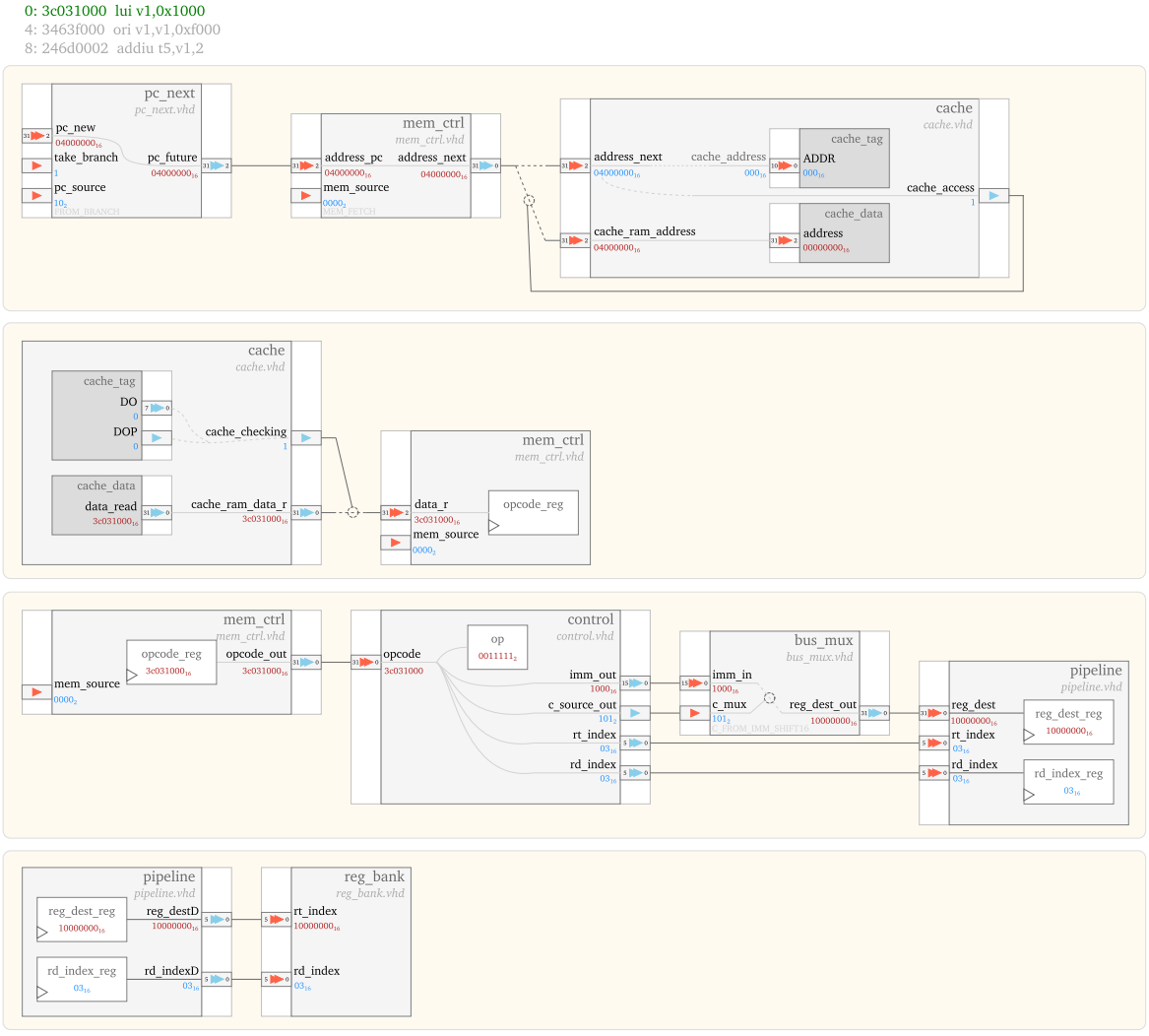

Figure 3.11: Instruction life-cycle.

4

List of Files

File	Purpose
plasma_ISE	
ml410.ucf	Board constraints file (FPGA pin mappings, timing constraints)
plasma_ise.xise	Xilinx ISE project file
rtl	
alu.vhd	Arithmetic Logic Unit
boot_ram.vhd	Boot memory
bus_mux.vhd	BUS Multiplex Unit
cache.vhd	4KB unified cache
cache_ram.vhd	Cache data memory
clk_gen.vhd	Clock generation
control.vhd	Instruction er
ddr_ctrl_top.vhd	DDR SDRAM controller wrapper
ddr_ctrl.vhd	DDR SDRAM controller
ddr_init.vhd	DDR SDRAM initialization
mem_ctrl.vhd	Memory Interface Unit
mlite_cpu.vhd	Top Level VHDL for CPU core
mlite_pack.vhd	Constants and Functions Package
mult.vhd	Multiplication and Division Unit
pc_next.vhd	Program Counter Unit
pipeline.vhd	Pipeline Delay Unit
plasma.vhd	CPU core with boot memory, cache and UART
plasma_top.vhd	SoC Platform Top Level VHDL
reg_bank.vhd	Register Bank for 32, 32-bit Registers
shifter.vhd	Shifter Unit
top_ml410.vhd	SoC Platform Top Level for interface Xilinx ML-410 board
uart.vhd	UART (can pause CPU if needed)
sim	
boot_ram_sim.vhd	Simulation boot memory
sim.do	Simulation script
sim_tb_top.vhd	Testbench module
sim\ddr_content	
*.srec	Benchmark simulation image
simlib	
.	DDR Memory Simulation Behavioral Model

5

Frequently Asked Questions

Q **Can I borrow a FPGA to my place?**

A No, it is not possible to borrow the FPGA.

Q **Can somebody take a look at my design? I have some errors that I do not manage to solve.**

A No, we can hardly do that. You should be able to solve the problems by yourself if you follow a good design discipline, you have a clear plan of what you want to achieve. Creating checkpoints for your design can be quite helpful as they can help you to go back to a previous solution, which didn't have the errors that you cannot fix.

Q **Can I have a discussion with you about the design? I have some ideas to improve it.**

A Yes. Actually there are two intermediate milestone meetings scheduled on April 30 and May 7 in which those things should be discussed. Prior and after that, you may consult the Processor Design team members (rooms HB10.070 - HB10.090). Please bear in mind that abusing that service may have negative consequences on your grade. In general, you are encouraged to have initiative, to be creative and solve the problems by yourself.

Bibliography

[1] S. Roades, *Plasma - most MIPS I opcodes,* http://opencores.org/project,plasma.

[2] Xilinx, *Xilinx ML410 Documentation and Tutorials,* http://www.xilinx.com/products/boards/ml410 ().

[3] Xilinx, *ML410 Embedded Development Platform User Guide,* http://www.xilinx.com/support/documentation/boards_and_kits/ug085.pdf ().

[4] C. Price, *MIPS IV Instruction Set,* http://math-atlas.sourceforge.net/devel/assembly/mips-iv.pdf.

[5] Xilinx, *Virtex-4 Libraries Guide for HDL Designs,* http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/virtex4_hdl.pdf ().