ET4 171 - Processor Design Project Guide

Nicoleta Cucu-Laurenciu
n.cuculaurenciu@tudelft.nl

Sorin Cotofana
s.d.cotofana@tudelft.nl

TU Delft Delft University of Technology

Contents

1 Overview **1**

2 Project Workflow **2**

2.1 Setup of the Development Environment . . . 2
2.2 Functional Verification . . . 3
2.2.1 Custom Testbenches . . . 4
2.2.2 Instruction Set Architecture Modifications . . . 4
2.3 FPGA Implementation . . . 5
2.4 Processor Evaluation . . . 6
2.4.1 Area Evaluation . . . 6
2.4.2 Timing Analysis . . . 7
2.4.3 Benchmarks . . . 7
2.4.4 FPGA Benchmark Execution . . . 8
2.4.5 Power Consumption Evaluation . . . 9
2.5 How to Report the Results . . . 10

3 Plasma Platform **13**

3.1 Plasma CPU . . . 13
3.1.1 Program Counter Generation . . . 13
3.1.2 Memory Interface Unit . . . 14
3.1.3 Decode and Control Unit . . . 15
3.1.4 Signal Multiplexing Unit . . . 16
3.1.5 Register File (Register Bank) . . . 17
3.1.6 Functional Units . . . 18
3.1.7 Pipeline . . . 19
3.2 Cache . . . 19
3.3 DDR Controller . . . 19
3.4 Clock Generation . . . 19

4 List of Files **25**

5 Frequently Asked Questions **26**

Bibliography **27**

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 and 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 as 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. In the best scenario 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 will get fatal errors at compilation 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 debugging tools, for hardware or software, and/or for both of them.

The previous discussion holds true for 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 Plasma platform is described in Section 3.

2

Project Workflow

To design, debug, 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. First, the baseline design should be analyzed and its weak point(s) determined by means of, e.g., profiling, and subsequently avenues for improvement should be identified. Then, for each and every to-be-changed/novel processor component, a pen-and-paper plan capturing its input and output ports and RTL design for the intended component functionality should be made. The next step is to write the corresponding HDL code for that component and test whether it performs its function as intended or not (behavioral RTL simulation). Generally speaking, it is advisable to write a HDL testbench and test that component individually, before integrating it inside the system. Otherwise, the testbench provided for the overall system can be used directly (facilitated by the fact that based on either assembly or C code, all the necessary means for generating the needed testcases with custom sequence of to be executed instructions are provided). Once the RTL verification of the system (which integrates the modified component) is successfully passed, the system design can be implemented and tested on the FPGA. Logic synthesis is first carried out, converting the RTL constructs from the HDL code into a design implementation in terms of generic resources (LUTs, MUXes, memory, flip-flops, adders, etc.). The next step is to map the design generic resources into the specific resources available on the Virtex 4 FPGA (e.g., slices, IOBs). The place and route step then decides where in the die the specific resources are to be placed, and wires them together such that the design criteria imposed (such as pin constraints for components ports and timing constraints from the **.ucf** file) are being met. The end result is a bit file, which can be downloaded into the FPGA in order to configure it according to the placed and routed design. The system timing performance, area, and consumed power can now be evaluated and appropriate further actions undertaken. A detailed description of the Figure 2.1 flow steps is presented in the remainder of the chapter. In addition, we strongly advise you to make use of the following policy:

- 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

A virtual machine running Ubuntu 16.04.2, which contains the project files and the employed EDA software (QuestaSim 10.5c_4 and Xilinx ISE 14.7), is provided. A listing of the directory structure and provided files is given in Section 4.

Prerequisite: TU Delft IP address.
When running the EDA tools (QuestaSim and Xilinx ISE) outside the TUD Campus, a VPN connection is required in order to be able to access the licenses (see [1] for instructions on how to set up a VPN connection).

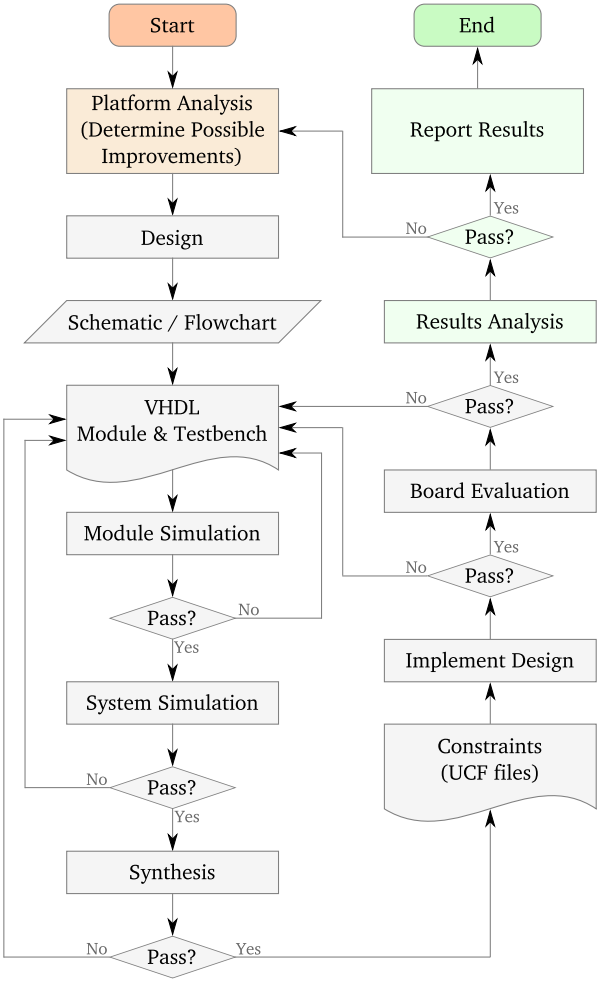

Figure 2.1: Flow Steps.

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 file `sim/sim_tb_top.vhd`.

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

```
1  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 running on the processor can be selected by configuring the *`sdramfile`* constant in the `testbench.vhd` file, with values equal to the file names in the folder `sim/ddr_content/`.

Note that the simulation time is quite long for the evaluation benchmarks (see Section 2.4.3). You can check the information in the QuestaSim console or the UART output.txt file to track the simulation progress. If you want to resume the simulation after a stop, use the command:

```
1  run -all
```

or the command:

```
1  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 is compiled. This means that you should place this command before the one that compiles the higher level module.

By default, the signal waveforms in the plasma_top module and ddr_ctrl_top are monitored. You can add signals you want to observe directly to the script in a similar manner, or using the QuestaSim menu (please refer to the QuestaSim documentation).

If your design passes this verification then you can continue to further steps, otherwise you have to revisit your design and re-do the required design flow steps until you pass the formal verification milestone.

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 processor functionality using this opcodes test, as it allows for easier fault isolation, and faster overall debugging, when compared to the benchmarks evaluation case.

To get more inside into the processor behaviour during the debugging step you may also develop your own custom testbenches and to this end a MIPS compiler and cross-assembler are provided.

You can create your own testbench in assembly (by e.g., modifying benchmarks/opcodes.asm). In the folder benchmarks/opcodes.asm, you can also find a sample Makefile. Running `make clean opcodes` in that folder will re-generate the corresponding opcodes.srec file needed for behavioral RTL simulation in QuestaSim.

Alternatively, you can also write your own testbench in c code (folder benchmarks/custom) and execute:

```
cd benchmarks/custom
make clean image
```

The corresponding custom.srec file will be re-generated automatically in the sim/ddr_content folder.

2.2.2. Instruction Set Architecture Modifications

The MIPS cross-compiler (`mips-pdp-elf-`) and the tool employed to build it (crosstool-NG), are provided in case you decide to increase the Plasma processor performance by means of MIPS ISA modifications. In such a case apart of adding new custom instruction(s) and designing the afferent hardware supporting its/their execution you also need to enable the appropriate compiler support as otherwise the new instruction(s) will not be utilized into the generated code. The required compiler modifications should be made in the form of patches which should be placed in the folder toolchain-ctng/ctng-bin/lib/crosstool-ng-1.22.0/patches/my_patches. The source packages used to build the cross-compiler are available in the folder toolchain-ctng/ctng_toolchain_src.
It is advisable not to overwrite the original compiler, but to run

```
ct-ng menuconfig
```

in the toolchain-ctng folder, and modify `Tuple's vendor string` in the `Toolchain options` menu. This will update the `.config` file further used for toolchain building. The MIPS toolchain can then be re-built using in the same folder (toolchain-ctng):

```
ct-ng clean
ct-ng build
```

Upon finishing, the path should be updated in benchmarks/Makefile.common, and the benchmarks re-compiled e.g., for benchmark pi

```
cd benchmarks/pi
make clean image
```

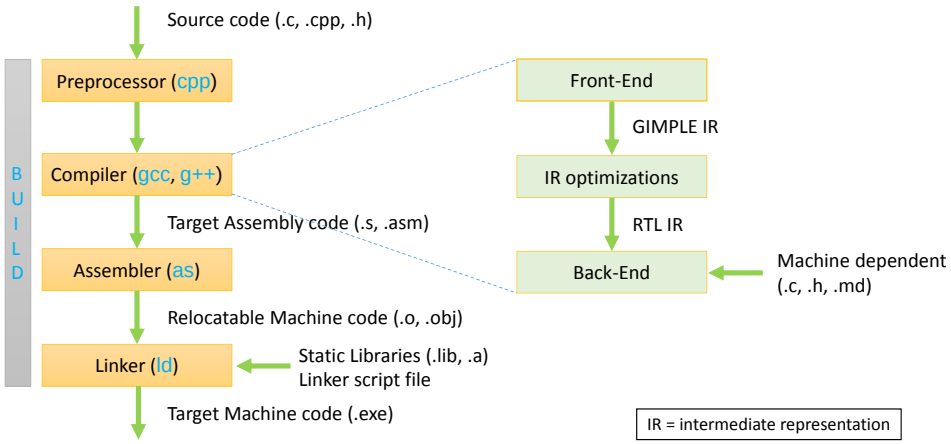

Figure 2.2: C Compilation Flow.

The corresponding `pi.srec` file will be re-generated automatically in the `sim/ddr_content` folder.

Subsequently, we describe the main steps for creating GCC compiler support for new custom instructions added to the existing MIPS ISA. Figure 2.2 depicts the compilation flow for a C program. Preprocessing is the first pass, where the include-files, conditional compilation instructions, and macros are being processed. The second pass is compilation, which takes the output of the preprocessor and the C source code, and generates assembly instructions specific to the target processor architecture. Assembly is the third stage of compilation, which produces the machine code (object code). Linking is the final stage of compilation, which combines one or multiple object files or libraries provided as input, into a single executable file. The link process is controlled by a linker script (see for example `benchmarks/elf32ebmip.xc`), which describes how the sections should be mapped into the output file, and controls the memory layout of the output file.

The two components that we are interested in modifying to add support for a new instruction are the compiler and the assembler.

The compiler, GCC, consists of a language front-end, and a back-end, as illustrated in Figure 2.2. The front-end parses the source code into an Intermediate Representation (IR). High-level optimizations are performed on the IR code by the GCC middle-end, followed by conversion to a different IR representation (RTL), which shall be used by the GCC bak-end. The back-end performs a series of low-level optimizations (such as common sub-expression elimination, data flow analysis, combine instruction patterns, register allocation, instruction scheduling), and generates the assembly code as end-result. A file of interest is `gcc/config/mips/mips.md`, as it defines all the target instructions patterns which are used by GCC, and is subject to target specific customizations. These instructions are defined using Machine Description (MD) constructs [2]. The anatomy of a 3-operand 32-bit addition instruction is shown in Figure 2.3 as an example. The expression `match_operand:DI 0 "register_operand" "d,d"` will try to match operand 0 having machine mode Double Integer (DI) (64-bits). The predicate function to use is `register_operand` and the constraint to be tested is `"=d"` for operand 0 (`"="` means that operand 0 is the output operand, `"d"` means register defined by mips). For this instruction, there are 2 alternative constraints (1) `"d"` for operand 0, 1 and 2, and (2) `"d"` for operand 0 and 1 and `"Q"` for operand 3. For details see [2].

The assembler, as part of the `binutils` package requires modifications as well, as it has to recognize the existence of the new custom instruction. The file `binutils/opcodes/mips-opc.c` contains all the built-in MIPS instructions used to assemble (and file `binutils/opcodes/mips-dis.c` to disassemble the target instructions). Figure 2.4 illustrates the `addi` instruction entry. More about the particular fields structure can be found in `include/opcode/mips.h`.

2.3. FPGA Implementation

To synthesize and implement your design, open the Xilinx ISE project file in the folder `pdp_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 `Generate Programming File`

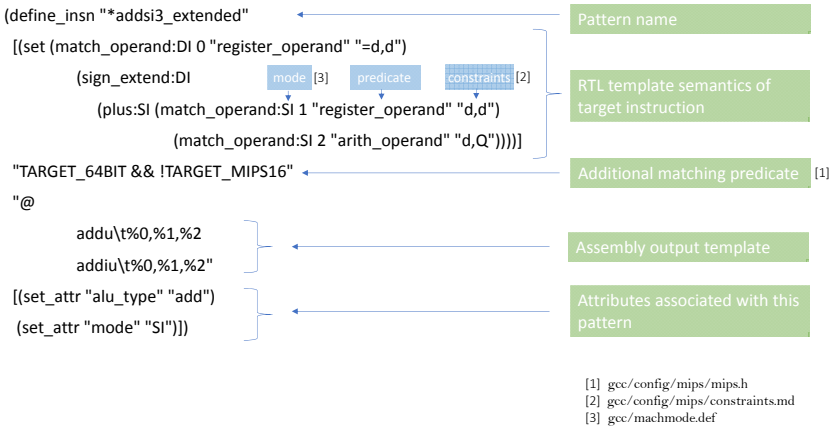

Figure 2.3: Instruction RTL Template.

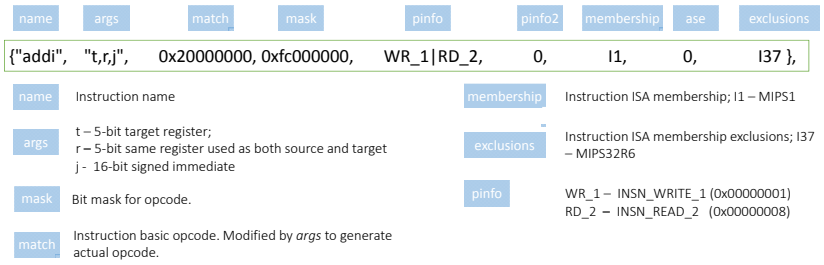

Figure 2.4: "Addi" Instruction Entry.

in the `Processes` panel, all the implementation steps (logic synthesis, place and route, generate programming file) are automatically executed. Upon a successful completion the bitstream file `top_ml410.bit` is available in this folder (`pdp_ISE`).

To create your own ISE project from scratch, you can follow 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 folder `rtl` to your project.
- Add the constraint file `pdp_ISE/ml410.ucf` to your design.

2.4. Processor Evaluation

Once you obtain a bug free design you may pursue to its evaluation. Part of the relevant information, i.e., maximum operating frequency and area, can be derived from ISE reports, while the rest requires benchmarks execution on the FPGA platform. In this section we discuss the key aspects of the processor evaluation procedure.

2.4.1. Area Evaluation

Figure 2.5 illustrates a floorplan for the baseline design. Configurable Logic Blocks (CLBs) are the main logic resource for combinatorial and sequential circuits implementation. Each CLB consists of 4 interconnected slices (see [3] Chapter 5 for details). For memory implementation, besides distributed RAM (from CLBs), a number of 18Kb Block RAMs (BRAMs) are available (see [3], Chapter 4 for details, and [4] for the BRAMs instantiation primitives).

To report the obtained area utilization, a compound value should be derived from the post place-and-route Device Utilization Summary in ISE, based on the number of occupied slices and the number of FIFO16/RAMB16s, considering the following relations:

- $A_{FIFO16/RAMB16} = A_{CLB}$,

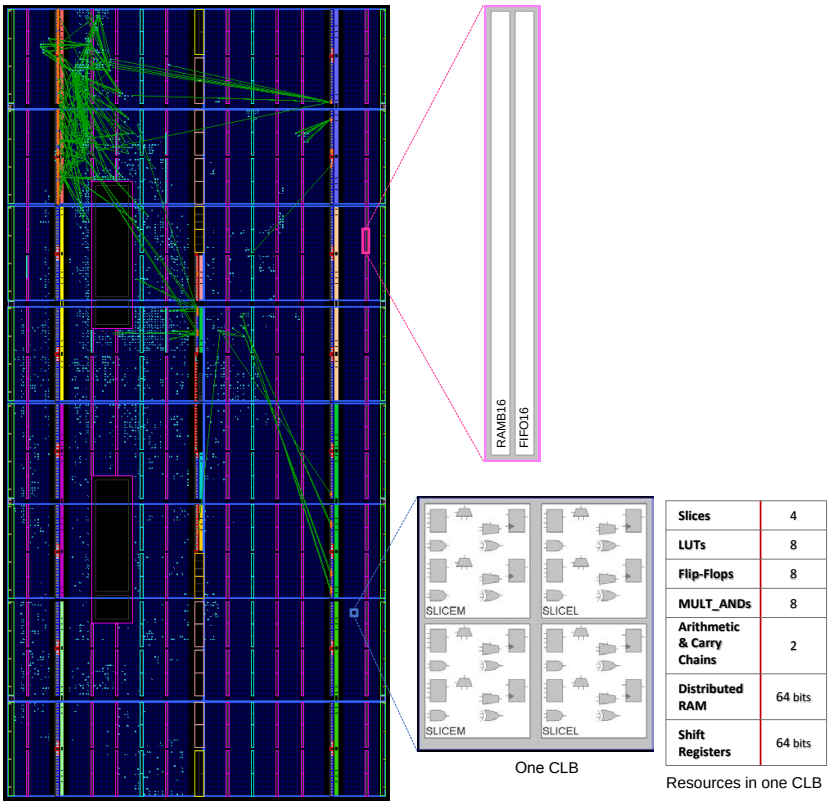

Figure 2.5: Baseline Design Floorplan.

- $A_{SLICE} = A_{CLB}/4$.

The baseline design area, for instance, can be computed using the previous relations as $9 \cdot A_{CLB} + 2022 \cdot A_{CLB}/4 = 514.5 \cdot A_{CLB}$.

2.4.2. 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. Note that this analysis provides a maximum attainable operation frequency estimate, but in practice due to FPGA operating conditions, e.g., FPGA temperature, your design might fail to operate at the corresponding clock period. In view of this a certain safety margin could be advisable for the FPGA experimental evaluation.

For the baseline design, the maximum frequency reported by ISE is 42.219MHz. The information related to the critical path can be found in the post Place & Route (PAR) report. Note that certain paths from the timing report can be false paths, and should be ignored. Figure 2.6 presents timing related information for a path in the timing report, among which the path start point and end point, the number of levels of logic, the logical resources included in the path, and the slack with respect to the maximum frequency. Further information on how to read a timing report or set timing constraints can be found in the ISE documentation.

2.4.3. Benchmarks

To evaluate the impact of your improvements on the Plasma platform performance you are given a set of benchmarks to be remotely run 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 the corresponding Benchmark Score - the execution cycle count in million cycles - on

```
Slack (setup path):     0.788ns (requirement - (data path - clock path skew + uncertainty))
  Source:               u1_plasma_top/u1_plasma/u1_cpu/u2_mem_ctrl/opcode_reg_4 (FF)
  Destination:          u1_plasma_top/u2_ddr/u2_ddr/data_write2_28 (FF)
  Requirement:          12.631ns
  Data Path Delay:      7.113ns (Levels of Logic = 8)
  Clock Path Skew:      -4.306ns (0.166 - 4.472)
  Source Clock:         u1_plasma_top/clk rising at 0.310ns
  Destination Clock:    u1_plasma_top/clk_2x falling at 12.941ns
  Clock Uncertainty:    0.424ns

  Clock Uncertainty:          0.424ns  ((TSJ^2 + TIJ^2)^1/2 + DJ) / 2 + PE
    Total System Jitter (TSJ):  0.000ns
    Total Input Jitter (TIJ):   0.000ns
    Discrete Jitter (DJ):       0.607ns
    Phase Error (PE):           0.120ns

  Maximum Data Path: u1_plasma_top/u1_plasma/u1_cpu/u2_mem_ctrl/opcode_reg_4 to u1_plasma_top/u2_ddr/u2_ddr/data_write2_28
    Location             Delay type         Delay(ns)  Physical Resource
                                                       Logical Resource(s)
    -------------------------------------------------  -------------------
    SLICE_X39Y147.YQ     Tcko                  0.291   u1_plasma_top/u1_plasma/u1_cpu/u2_mem_ctrl/opcode_reg<5>
                                                       u1_plasma_top/u1_plasma/u1_cpu/u2_mem_ctrl/opcode_reg_4
    SLICE_X37Y146.G1     net (fanout=15)       0.708   u1_plasma_top/u1_plasma/u1_cpu/u2_mem_ctrl/opcode_reg<4>
    SLICE_X37Y146.Y      Tilo                  0.165   u1_plasma_top/u1_plasma/u1_cpu/u3_control/alu_func<0>144
                                                       u1_plasma_top/u1_plasma/u1_cpu/u3_control/exception_out_or0000_SW0_SW0
    ...
    SLICE_X35Y155.G1     net (fanout=4)        0.566   u1_plasma_top/data_write<28>
    SLICE_X35Y155.CLK    Tgck                  0.239   u1_plasma_top/u2_ddr/u2_ddr/data_write2<29>
                                                       u1_plasma_top/u2_ddr/u2_ddr/data_write2_28_mux00011
                                                       u1_plasma_top/u2_ddr/u2_ddr/data_write2_28
    -------------------------------------------------  ---------------------------
    Total                                      7.113ns (2.008ns logic, 5.105ns route)
                                                       (28.2% logic, 71.8% route)
```

Figure 2.6: Post-PAR Timing Related Information for a Path.

the baseline Plasma platform are given in Table 2.1. The C source code and the assembly listings can be found in the `benchmarks` folder. Note that if your Plasma performance improvement strategy relies (also) on MIPS ISA modification/augmentation benchmarks compilation is required before launching their remote FPGA execution.

Table 2.1: Benchmarks Description and Baseline Performance.

Name	Benchmark Score (BS)	Description
opcodes	for verification only	Tests all MIPS I instructions
cjpeg	29.740140	JPEG compression
divide	264.820001	Large number (192Kb) integer division using GMP library
multiply	149.492876	Large number (64Kb) integer multiplication using GMP library
pi	845.277552	Computes 1000 digits of PI using basic arithmetic operations
fir	187.411039	Length-63 bandpass FIR filter applied to 50000 input samples
rsa	563.890532	RSA message signing using GMP library
ssd	797.170350	Pattern matching using Sum-of-Squared-Differences
ssearch	457.727967	String search using look-up tables
susan	910.498879	Gaussian image smoothing
bench_all	4323.096258	All Benchmarks in one run

2.4.4. FPGA Benchmark Execution

The evaluation of your enhanced processor on the FPGA board encompasses 2 main steps: (1) the FPGA is first programmed with your bitstream (`.bit`) file, and (2) the benchmark to be executed by the processor (`.bin` file), is then sent via UART to the FPGA board, loaded in the DDR memory available on the ML410 board, and its execution launched. After benchmark execution completion, the benchmark results are sent from the FPGA board to the computer via UART. Subsequently, we detail the procedure underlying the two previous steps.

The communication with the remote FPGA platform is realized via Dropbox shared folders [5]. Specifically, one common shared folder will be created for each and every group. To this end, each and every group should send one e-mail per group, which contains: the group number, the name, student id, and e-mail address linked to own Dropbox account, for all the members in the group. Based on the provided emails, shared folders invitations will be sent to each group member.

Once the bitstream file has been generated in ISE, you should rename it with the name of the corresponding benchmarks name, e.g., rename the bitsteam `top_ml410.bit` to `susan.bit` if you would like to evaluate target benchmark `susan`) and place it (e.g., `susan.bit`) in the group Dropbox folder. The FPGA will be programmed automatically with your bitstream, after which the `.bit` file will be automatically deleted from the Dropbox folder. The corresponding benchmark binary image (e.g., `susan.bin`) will be sent automatically via UART to the FPGA and loaded into the ML410 board DDR memory.

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`).
Notes:

- 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.
- There are two cases when you should upload in the Dropbox folder both the `.bit` and `.bin` files: (1) if the MIPS ISA is extended and implicitly compiler modification are carried out (and hence the 11 benchmarks are recompiled), and (2) if a new custom benchmark is created and compiled. In these 2 cases, the bit file `top_ml410.bit` and the bin file (e.g., `susan.bin`) should be renamed to `custom.bit` and `custom.bin` before placing them both in the Dropbox folder.

2.4.5. Power Consumption Evaluation

This section describes the steps that need to be followed to get a power consumption estimation of the Plasma Platform on an FPGA. ISE XPower Analyzer (XPA) tool is employed to estimate the post implementation - post Place-And-Route (PAR) - power consumption. Note that by default, the XPA tool does not require the user to specify any information related to the switching profile of the design nets, but uses instead a vectorless estimation algorithm, which assigns default activity rates to the design nets. However, to perform an accurate power analysis, the signals switching activity profile should be provided (e.g., via a **.vcd** file which contains information about value changes and their timestamps on selected signals in the design). Subsequently, the flow for power estimation of `u1_plasma` is provided.

Post PAR Simulation Model Generation (ISE)

Before you can estimate power your design has to be fully implemented (i.e., all steps up until, and including `Place & Route` are performed). After all components are placed into the actual FPGA device resources and all the details about used routing resources and exact timing information for each path in the design are defined, a VHDL simulation model of the implemented design (together with an **.sdf** file containing true timing delay information) can be generated. This VHDL model will be further used in QuestaSim to generate the activity file **.vcd**. To generate the simulation model, the following ISE steps are required:

- In `Hierarchy` pane right click and set `u1_plasma` as the new top module.
- In the `Processes` pane, expand `Implementation` and then `Place & Route`.
- Double-click `Generate Post-Place & Route Simulation Model`.

As a result, `plasma_timesim.vhd` and `plasma_timesim.sdf` are generated in the folder `netgen/par`.

Post PAR Simulation for .vcd Generation (QuestaSim)

To generate the .vcd file, a script `power.do` is provided in the simulation directory. The VHDL model of the plasma component and its submodules from the RTL behavioral simulation (`sim.do` script) are now replaced by the post-PAR VHDL model (`plasma_timesim.vhd`). The upper hierarchy components and the testbench file remain the same. Additionally, the script `power.do` includes signals activity dumping related commands

```
vcd file ../pdp_ISE/power.vcd
vcd add -ports -internal sim_tb_top/u1_plasma_top/u1_plasma/*
```

which instruct to monitor all ports (in and out) and internal signals for the plasma entity.
Note: The longer the simulation time, the more accurate the power estimates, but at the expense of increased filesize (may reach in the order of GBs).

Running `power.do` (`do power.do` in the QuestaSim command line) will generate the file `power.vcd` in the `pdp_ISE` folder. After that, the shell script `vcd_hierarchy_edit.sh` is required to be run in order to limit the scope to parts of the design hierarchy for further usage in ISE (the scopes of upper hierarchy modules `sim_tb_top` and `u1_plasma_top` are removed).

Power Estimation (ISE)

After completing the previous steps the XPA tool can be invoked by double clicking `Analyze Power Distribution (XPower Analyzer)` from the `Place & Route` expanded menu in the `Processes` pane. The tool receives as input (Select `File` → `Open Design`):

- the design file (`plasma.ncd`)
  (the placed and routed design database file which contains all the logic configuration and routing information)
- the physical constraints file (`plasma.pcf`)
  (contains the settings for all of the design logic and I/Os plus activity for specific nets such as clock networks)
- the simulation activity file (`power.vcd`)
  (contains the switching activity profile for the post-PAR design).

During analysis, the XPA tool will try to match the design nets in the post-PAR design (**.ncd**) with the simulation nets from the **.vcd** file. As a result, two percentages are being reported: design nets matched and simulation nets matched. For the remaining of the design nets that were not matched (and thus for which no switching activity is user provided), default activity rates are being assumed. Hence, the higher the nets matching percentages, the better quality the power estimates. After XPA has finished running, the dynamic power consumption will appear in the appropriate cells of the power report. Make sure all the clock frequencies have been correctly extracted.

Evaluating the power consumption for the baseline design running benchmark `pi` for 1ms, generates the values summarized in Figure 2.7. From these values, a 60% dynamic power estimation accuracy increase is observed when using the activity file `.vcd`.

Note: After finishing the power estimation, do not forget to switch back the top module to `top_ml410` in the ISE `Hierarchy` pane.

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 provide a short summary of the entire work while covering the following items:

- 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;
- Changes that you have performed to the core, together with their implications;

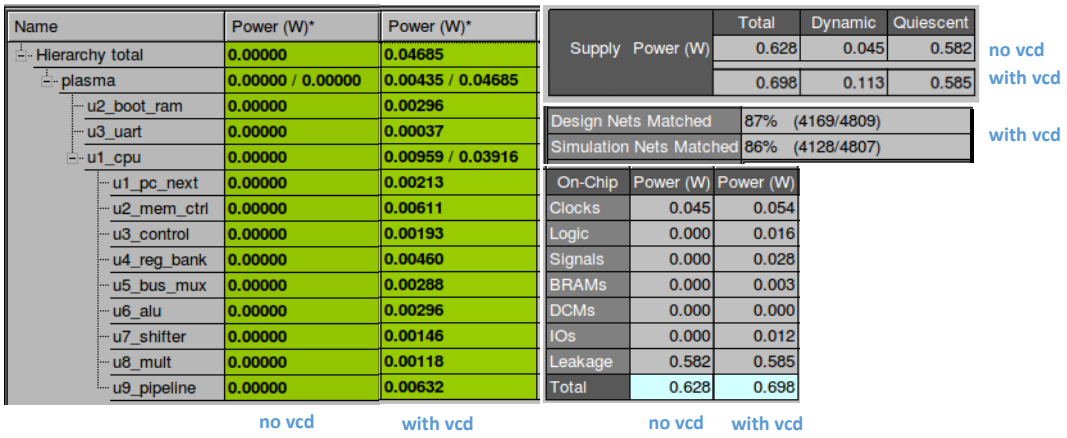

Figure 2.7: Baseline Power Consumption Values.

- Obtained main results;
- Your main conclusions.

This section should not include details about your design. At the end of the introduction add the organization (outline) of the report.

Next, include a separate section in which you motivate your choices for improvement in a way which is much more descriptive and comprehensive than what you mentioned in the introduction. To this end you need to detail:

- The way you analyzed the baseline processor in order to identify its weak points;
- Your findings based on which you have decided to make the improvements;
- The to be pursued improvement avenues;
- What do you expect from implementing your changes.

The following section should provide a description of all the performed changes in which you need to motivate your detailed design choices. For example: 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 you could choose and present the reason(s) for your choice. Do not forget to refer to the literature. While completing this section give attention to the following:

- Include figures to present the architecture of the new blocks that you introduce in the design. It is advisable to 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.
- Describe the way you embed your designs into the processor. You can draw the action of hand-shaking signals in the form of timing chart and/or state machine. Bear in mind that handling the interface signals properly is essential in order to allow for a smooth integration of your new modules within the Plasma processor.
- Discuss the design verification aspects you considered.
- If relevant, attach QuestaSim simulation results.

The next section is dedicated to reporting and commenting the experimental results and it can be perceived as the most important part of the report. In this section you should:

- Provide experimental results and analyze them separately for each modification you did, as well as for combination of improvements. In this way, you can compare the improvements in terms of cost and benefit against the baseline, against each other, and see how the results add up when you combine different improvements.
- Report the detailed results (including timing information, critical path information, resource utilization information from ISE reports, and power consumption figures) for all considered designs.
- Take into account in your analysis basic relevant performance metrics, e.g., area (A), critical path delay (deduced from clock frequency) (D), Benchmarks Scores (BSs), power consumption (P), as well as compound metrics, e.g., A*D, A*BS, P*D, and P*BS products.
- Comment on the obtained results and try to identify which improvement is contributing to which figure of merit and which proposed improvement is the most effective.

Finally, summarize your work and add your conclusions and possible future work plans in the last report section. In this context 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).

Some other issues you need to think about when writing your report:

- The report should be consistent, in structure, language, and formatting style.
- It is a common practice to make use of the present tense, with the exception of the conclusion section, where past tense should be employed.
- Headings are written with capitals and numbered.
- The text is usually justified.
- References should be placed in a dedicated section at the end of the 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;
- a unified 4-KB cache;
- a 4-KB boot memory;
- a DDR SDRAM controller;
- a Universal Asynchronous Receiver/Transmitter (UART) unit.

The platform is based on the open-source Plasma project [6] available on the OpenCores website. The original design was updated for our purpose and tailored for the Xilinx ML410 board (Revision E) [7], depicted in Figure 3.2. Technical details about the on board available hardware can be found in [8].

The SoC use scenario for 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 the 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, as otherwise the FPAG remote evaluation will not function properly.

3.1. Plasma CPU

The Plasma CPU is a small synthesizable 32-bit RISC microprocessor that executes all MIPS I [9] user mode instructions except unaligned load and store operations. In the VHDL code the CPU top module is represented by the `mlite_cpu` unit (see Figure 3.3) and its 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.

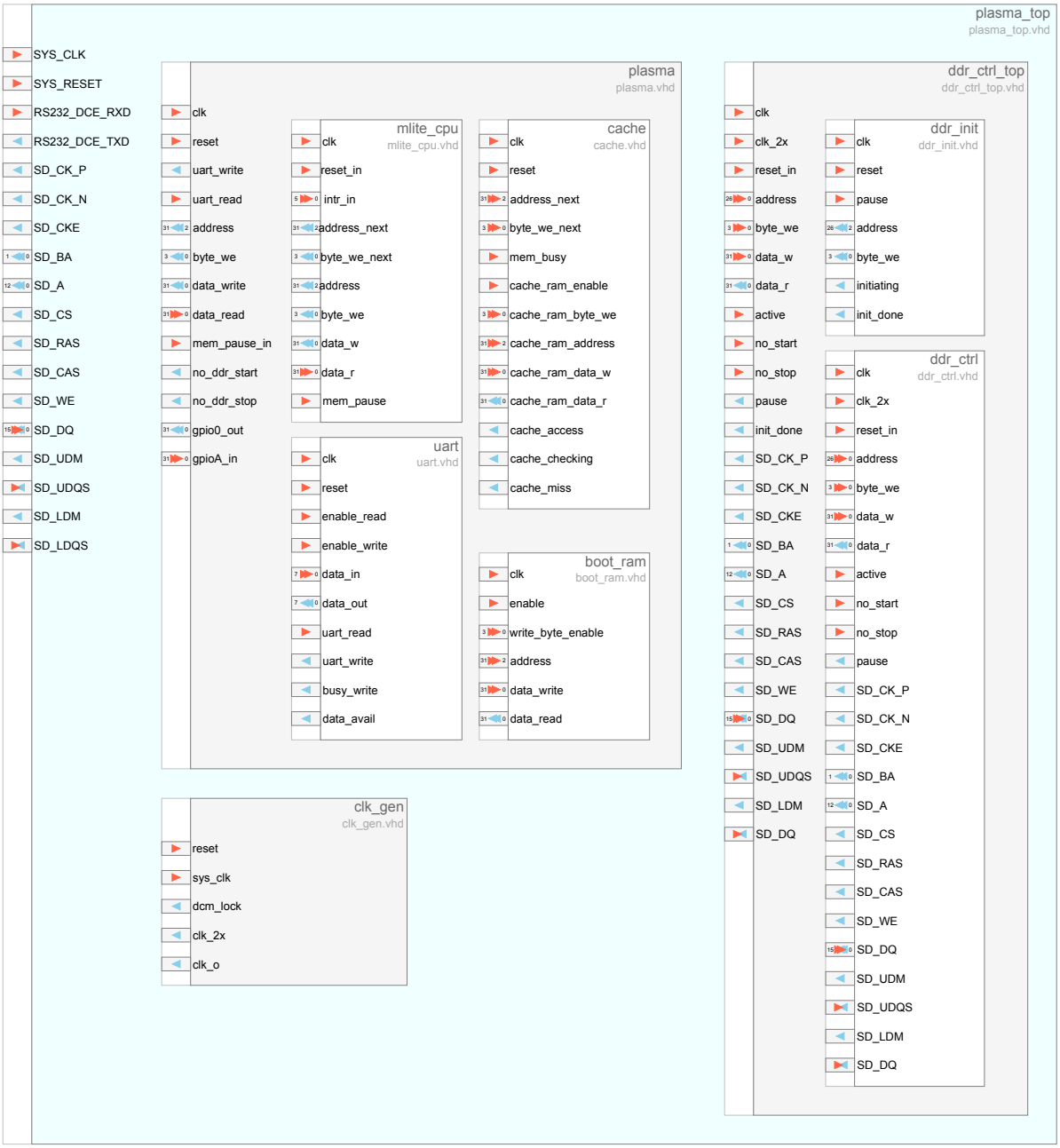

Figure 3.1: Plasma SoC Platform.

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) is managing the core to memory communication: it sends addresses to and receives/send data from/to 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*

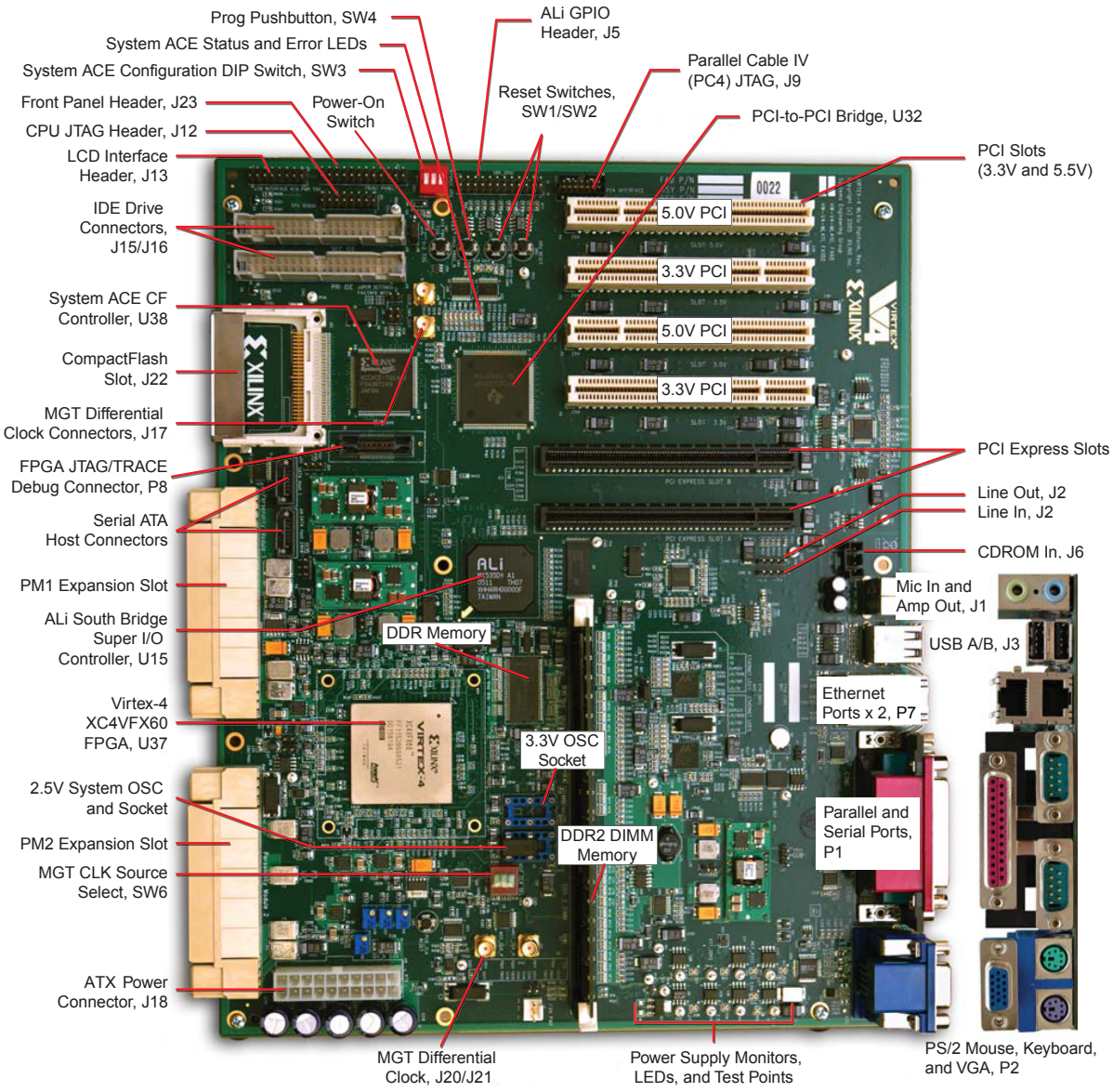

Figure 3.2: Xilinx ML410 Board and Front Panel Detail.

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 where to be written address.

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 encoding is detailed in Figure 3.7.

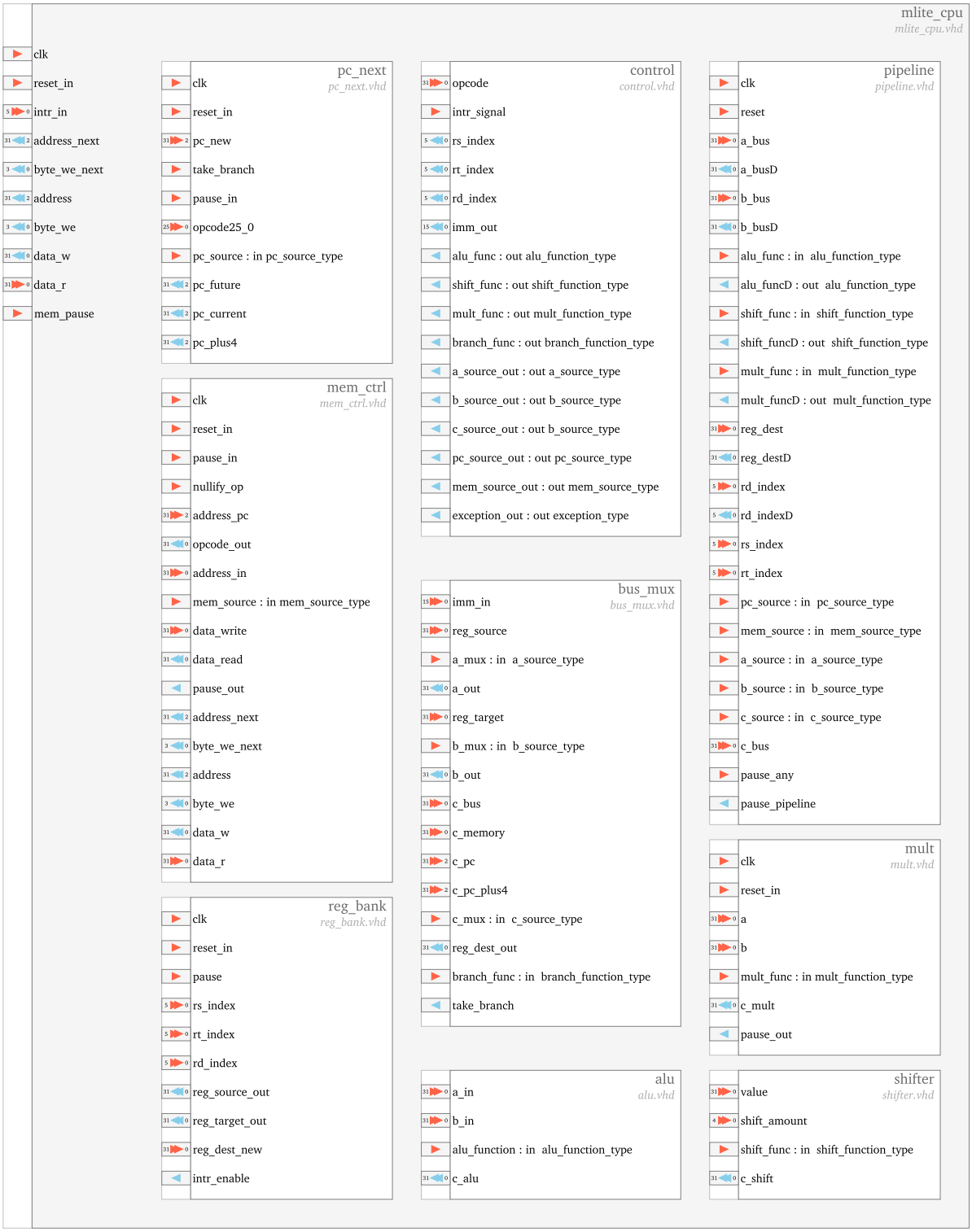

Figure 3.3: Plasma CPU components.

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.

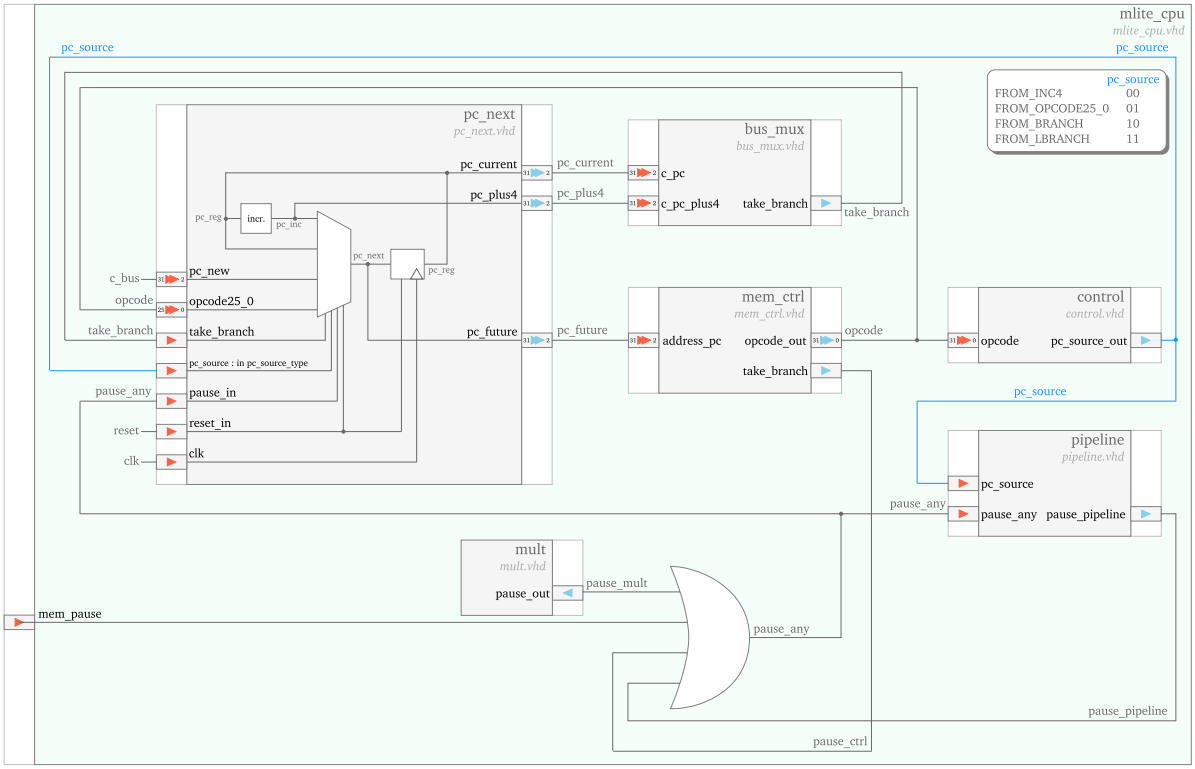

Figure 3.4: Program counter generation.

3.1.5. Register File (Register Bank)

The Plasma CPU is based on the MIPS I instruction set, hence it embeds 32 32-bit general purpose registers and from the user (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

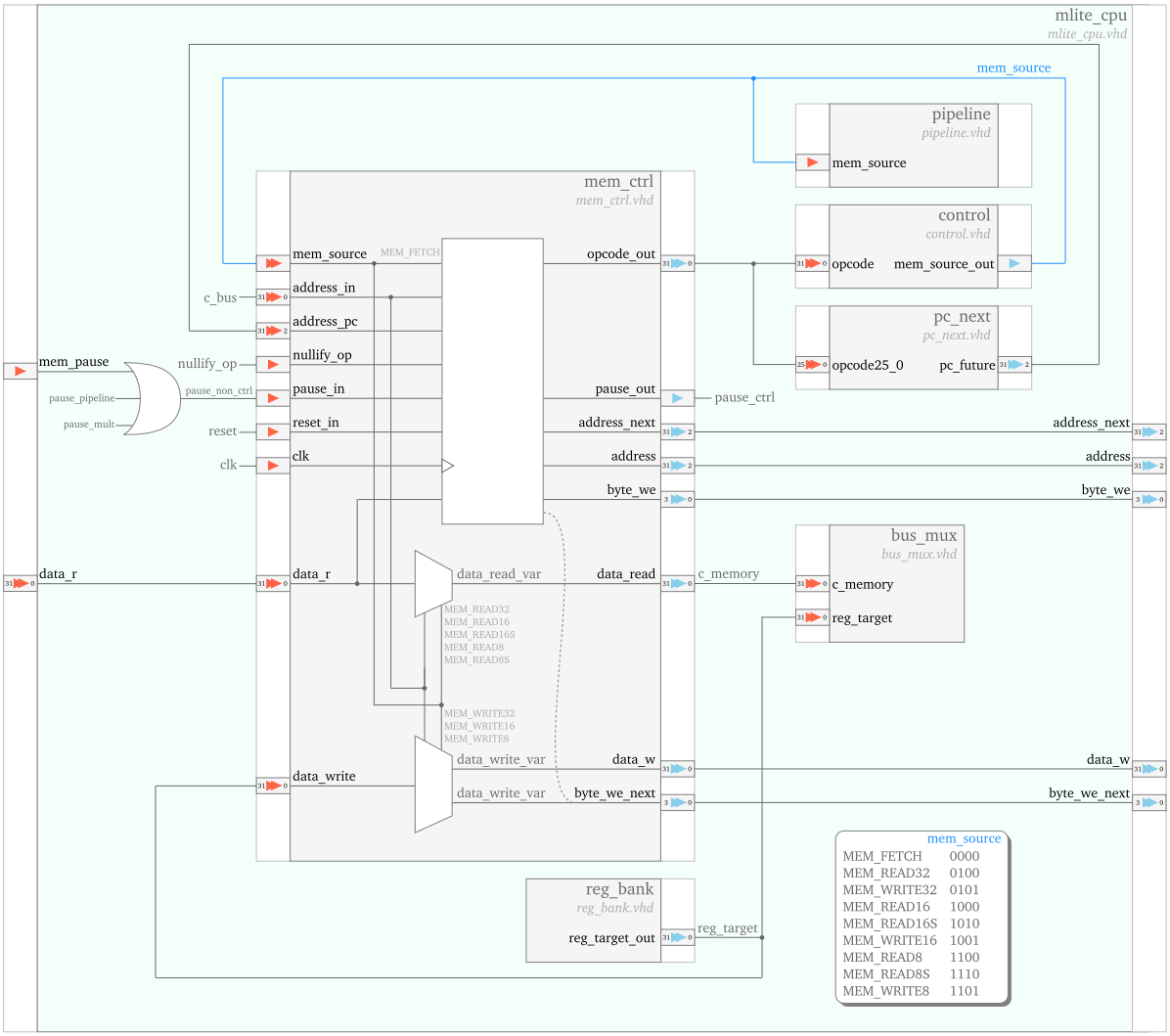

Figure 3.5: Internal CPU Memory Controller.

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 dual port memories [4].

3.1.6. Functional Units

The Plasma CPU has three functional units: an ALU, a Multiplier/Divider, and a Shifter.

The ALU (`alu.vhd`) executes arithmetic and logic operations, with a delay of 1 clock cycle. The adder in the ALU is described in behavioral VHDL as a ripple-carry adder.

The serial Multiplier/Divider (`mult.vhd`) takes 32 cycles to compute the 64-bit multiplier result, or the 32-bit quotient and the he 32-bit remainder. The pipeline is stalled during the mul/div operation by asserting the *`pause_out`* signal.

The Shifter (`shifter.vhd`) performs left and right bit-shifting in 1 clock cycle.

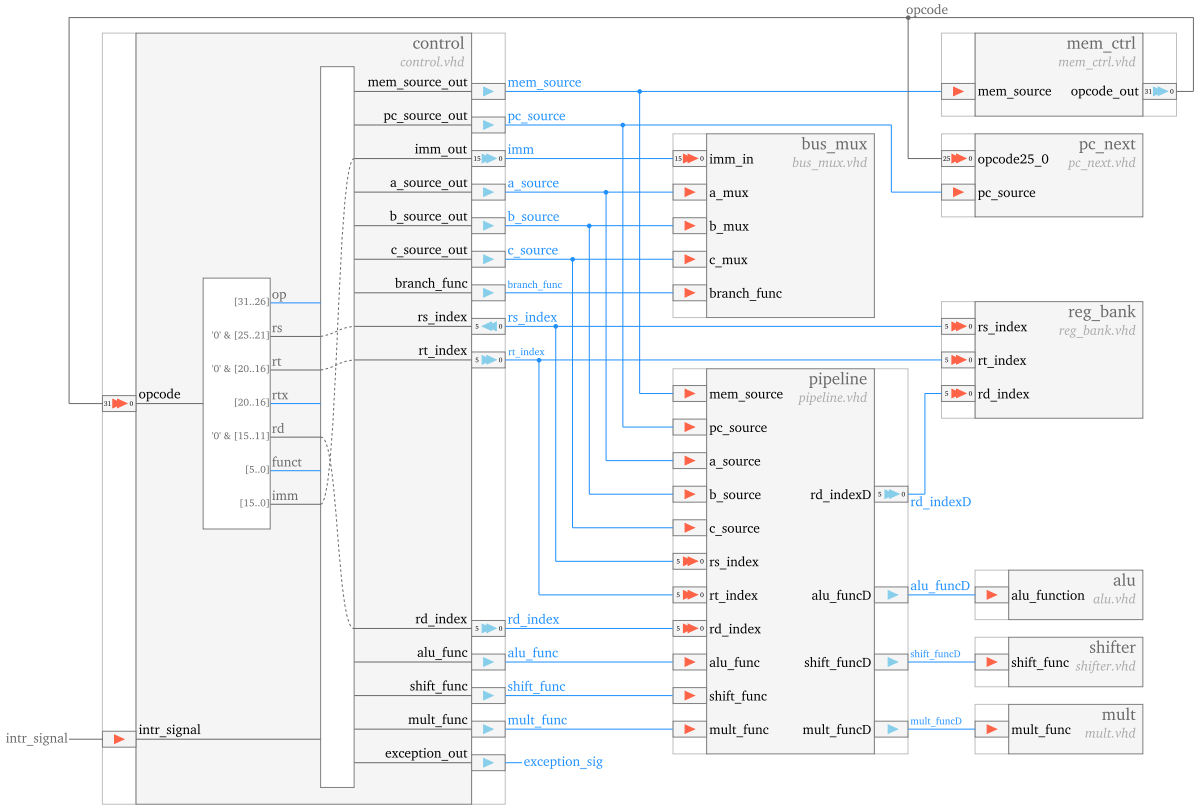

Figure 3.6: Decode and Control Logic Unit.

3.1.7. 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, as described by 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.

3.3. DDR Controller

The Plasma platform has access to an MT46V16M16-5B 32MB DDR SDRAM memory chip [10] through the DDR controller descried within 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) [4] 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 by 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 according to this relation: $COUNT\_VALUE$ = core frequency (Hz) / 230400Hz.

				Opcode Bitfields															
	Opcode	Name	Action	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Arithmetic Logic Unit	ADD rd,rs,rt	Add	rd=rs+rt	0	1	0	1	0	1	rs	rs	rs	rs	rs	rt	rt	rt	rt	rt	rd	rd	rd	rd	rd	0	0	0	0	0	1	0	0	0	0
	ADDI rt,rs,imm	Add Immediate	rt=rs+imm	0	0	1	0	0	0	rs	rs	rs	rs	rs	rt	rt	rt	rt	rt	imm	imm	imm	imm	imm	imm	imm	imm	imm	imm	imm	imm	imm	imm	imm	imm
	ADDIU rt,rs,imm	Add Immediate Unsigned	rt=rs+imm	0	0	1	0	0	1	rs	rs	rs	rs	rs	rt	rt	rt	rt	rt	imm	imm	imm	imm	imm	imm	imm	imm	imm	imm	imm	imm	imm	imm	imm	imm
	ADDU rd,rs,rt	Add Unsigned	rd=rs+rt	0	1	0	1	0	1	rs	rs	rs	rs	rs	rt	rt	rt	rt	rt	rd	rd	rd	rd	rd	0	0	0	0	0	1	0	0	0	1
	AND rd,rs,rt	And	rd=rs&rt	0	1	0	1	0	1	rs	rs	rs	rs	rs	rt	rt	rt	rt	rt	rd	rd	rd	rd	rd	0	0	0	0	0	1	0	0	1	0
	ANDI rt,rs,imm	And Immediate	rt=rs&imm	0	0	1	1	0	0	rs	rs	rs	rs	rs	rt	rt	rt	rt	rt	imm	imm	imm	imm	imm	imm	imm	imm	imm	imm	imm	imm	imm	imm	imm	imm
	LUI rt,imm	Load Upper Immediate	rt=imm<<16	0	0	1	1	1	1	rs	rs	rs	rs	rs	rt	rt	rt	rt	rt	imm	imm	imm	imm	imm	imm	imm	imm	imm	imm	imm	imm	imm	imm	imm	imm
	NOR rd,rs,rt	Nor	rd=~(rs|rt)	0	1	0	1	0	1	rs	rs	rs	rs	rs	rt	rt	rt	rt	rt	rd	rd	rd	rd	rd	0	0	0	0	0	1	0	0	1	1
	OR rd,rs,rt	Or	rd=rs|rt	0	1	0	1	0	1	rs	rs	rs	rs	rs	rt	rt	rt	rt	rt	rd	rd	rd	rd	rd	0	0	0	0	0	1	0	0	1	0
	ORI rt,rs,imm	Or Immediate	rt=rs|imm	0	0	1	1	0	1	rs	rs	rs	rs	rs	rt	rt	rt	rt	rt	imm	imm	imm	imm	imm	imm	imm	imm	imm	imm	imm	imm	imm	imm	imm	imm
	SLT rd,rs,rt	Set On Less Than	rd=rs<rt	0	1	0	1	0	1	rs	rs	rs	rs	rs	rt	rt	rt	rt	rt	rd	rd	rd	rd	rd	0	0	0	0	0	1	0	1	0	1	0
	SLTI rt,rs,imm	Set On Less Than Immediate	rt=rs<imm	0	0	1	0	1	0	rs	rs	rs	rs	rs	rt	rt	rt	rt	rt	imm	imm	imm	imm	imm	imm	imm	imm	imm	imm	imm	imm	imm	imm	imm	imm
	SLTIU rt,rs,imm	Set On	rt=rs<imm	0	0	1	0	1	1	rs	rs	rs	rs	rs	rt	rt	rt	rt	rt	imm	imm	imm	imm	imm	imm	imm	imm	imm	imm	imm	imm	imm	imm	imm	imm
	SLTU rd,rs,rt	Set On Less Than Unsigned	rd=rs<rt	0	1	0	1	0	1	rs	rs	rs	rs	rs	rt	rt	rt	rt	rt	rd	rd	rd	rd	rd	0	0	0	0	0	1	0	1	0	1	1
	SUB rd,rs,rt	Subtract	rd=rs-rt	0	1	0	1	0	1	rs	rs	rs	rs	rs	rt	rt	rt	rt	rt	rd	rd	rd	rd	rd	0	0	0	0	0	1	0	0	0	1	0
	XOR rd,rs,rt	Subtract Unsigned	rd=rs-rt	0	1	0	1	0	1	rs	rs	rs	rs	rs	rt	rt	rt	rt	rt	rd	rd	rd	rd	rd	0	0	0	0	0	1	0	0	0	1	1
	SUBU rd,rs,rt	Exclusive Or	rd=rs^rt	0	1	0	1	0	1	rs	rs	rs	rs	rs	rt	rt	rt	rt	rt	rd	rd	rd	rd	rd	0	0	0	0	0	1	0	0	1	1	0
	XORI rt,rs,imm	Exclusive Or Immediate	rt=rs^imm	0	0	1	1	1	0	rs	rs	rs	rs	rs	rt	rt	rt	rt	rt	imm	imm	imm	imm	imm	imm	imm	imm	imm	imm	imm	imm	imm	imm	imm	imm
Shifter	SLL rd,rt,sa	Shift Left Logical	rd=rt<<sa	0	1	0	1	0	1	rs	rs	rs	rs	rs	rt	rt	rt	rt	rt	rd	rd	rd	rd	rd	sa	sa	sa	sa	sa	0	1	0	1	0	1
	SLLV rd,rt,rs	Shift Left Logical Variable	rd=rt<<rs	0	1	0	1	0	1	rs	rs	rs	rs	rs	rt	rt	rt	rt	rt	rd	rd	rd	rd	rd	0	0	0	0	0	0	0	1	0	0	0
	SRA rd,rt,sa	Shift Right Arithmetic	rd=rt>>sa	0	1	0	1	0	1	0	0	0	0	0	rt	rt	rt	rt	rt	rd	rd	rd	rd	rd	sa	sa	sa	sa	sa	0	0	0	0	1	1
	SRAV rd,rt,rs	Shift Right Arithmetic Variable	rd=rt>>rs	0	1	0	1	0	1	rs	rs	rs	rs	rs	rt	rt	rt	rt	rt	rd	rd	rd	rd	rd	0	0	0	0	0	0	0	1	1	1	1
	SRL rd,rt,sa	Shift Right Logical	rd=rt>>sa	0	1	0	1	0	1	0	0	0	0	0	rt	rt	rt	rt	rt	rd	rd	rd	rd	rd	sa	sa	sa	sa	sa	0	0	0	0	1	0
	SRLV rd,rt,rs	Shift Right Logical Variable	rd=rt>>rs	0	1	0	1	0	1	rs	rs	rs	rs	rs	rt	rt	rt	rt	rt	rd	rd	rd	rd	rd	0	0	0	0	0	0	0	1	1	0	0
Multiply/Division	DIV rs,rt	Divide	HI=rs%rt; LO=rs/rt	0	1	0	1	0	1	rs	rs	rs	rs	rs	rt	rt	rt	rt	rt	0	1	0	1	0	1	0	0	0	0	0	0	0	1	1	0	1	0
	DIVU rs,rt	Divide Unsigned	HI=rs%rt; LO=rs/rt	0	1	0	1	0	1	rs	rs	rs	rs	rs	rt	rt	rt	rt	rt	0	1	0	1	0	1	0	0	0	0	0	0	0	1	1	0	1	1
	MFHI rd	Move From HI	rd=HI	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	0	0	0	0	0	0	rd	rd	rd	rd	rd	0	0	0	0	0	0	1	0	0	0	0
	MFLO rd	Move From LO	rd=LO	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	0	0	0	0	0	0	rd	rd	rd	rd	rd	0	0	0	0	0	0	1	0	0	1	0
	MTHI rs	Move To HI	HI=rs	0	1	0	1	0	1	rs	rs	rs	rs	rs	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	0	0	0	0	0	0	1	0	0	0	1
	MTLO rs	Move To LO	LO=rs	0	1	0	1	0	1	rs	rs	rs	rs	rs	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	0	0	0	0	0	0	1	0	0	1	1
	MULT rs,rt	Multiply	HI,LO=rs*rt	0	1	0	1	0	1	rs	rs	rs	rs	rs	rt	rt	rt	rt	rt	0	1	0	1	0	1	0	0	0	0	0	0	0	1	1	0	0	0
	MULTU rs,rt	Multiply Unsigned	HI,LO=rs*rt	0	1	0	1	0	1	rs	rs	rs	rs	rs	rt	rt	rt	rt	rt	0	1	0	1	0	1	0	0	0	0	0	0	0	1	1	0	0	1
Branch	BEQ rs,rt,offset	Branch On Equal	if(rs==rt) pc+=offset*4	0	0	0	1	0	0	rs	rs	rs	rs	rs	rt	rt	rt	rt	rt	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset
	BGEZ rs,offset	Branch On >= 0	if(rs>=0) pc+=offset*4	0	0	0	0	0	1	rs	rs	rs	rs	rs	0	0	0	0	1	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset
	BGEZAL rs,offset	Branch On >= 0 And Link	r31=pc; if(rs>=0) pc+=offset*4	0	0	0	0	0	1	rs	rs	rs	rs	rs	1	0	0	0	1	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset
	BGTZ rs,offset	Branch On > 0	if(rs>0) pc+=offset*4	0	0	0	1	1	1	rs	rs	rs	rs	rs	0	0	0	0	0	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset
	BLEZ rs,offset	Branch On	if(rs<=0) pc+=offset*4	0	0	0	1	1	0	rs	rs	rs	rs	rs	0	0	0	0	0	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset
	BLTZ rs,offset	Branch On	if(rs<0) pc+=offset*4	0	0	0	0	0	1	rs	rs	rs	rs	rs	0	0	0	0	0	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset
	BLTZAL rs,offset	Branch On	r31=pc; if(rs<0) pc+=offset*4	1	0	0	0	0	0	rs	rs	rs	rs	rs	0	0	0	0	0	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset
	BNE rs,rt,offset	Branch On Not Equal	if(rs!=rt) pc+=offset*4	0	0	0	1	0	1	rs	rs	rs	rs	rs	rt	rt	rt	rt	rt	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset
	BREAK	Breakpoint	epc=pc; pc=0x3c	0	1	0	1	0	1	code	code	code	code	code	code	code	code	code	code	code	code	code	code	code	code	code	code	code	code	0	0	1	1	0	1
	J target	Jump	pc=pc_upper|(target<<2)	0	0	0	0	1	0	target	target	target	target	target	target	target	target	target	target	target	target	target	target	target	target	target	target	target	target	target	target	target	target	target	target
	JAL target	Jump And Link	r31=pc; pc=target<<2	0	0	0	0	1	1	target	target	target	target	target	target	target	target	target	target	target	target	target	target	target	target	target	target	target	target	target	target	target	target	target	target
	JALR rs	Jump And Link Register	rd=pc; pc=rs	0	1	0	1	0	1	rs	rs	rs	rs	rs	0	0	0	0	0	rd	rd	rd	rd	rd	0	0	0	0	0	0	0	1	0	0	1	1	0	0	1
	JR rs	Jump Register	pc=rs	0	1	0	1	0	1	rs	rs	rs	rs	rs	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	0	0	0	0	0	0	1	0	0	0	0
	MFC0 rt,rd	Move From Coprocessor	rt=CPR[0,rd]	0	1	0	0	0	0	0	0	0	0	0	rt	rt	rt	rt	rt	rd	rd	rd	rd	rd	0	1	0	1	0	0	0	0	0	1	0	0	0	0	0	0
	MTC0 rt,rd	Move To Coprocessor	CPR[0,rd]=rt	0	1	0	0	0	0	0	0	1	0	0	rt	rt	rt	rt	rt	rd	rd	rd	rd	rd	0	1	0	1	0	0	0	0	0	1	0	0	0	0	0	0
	SYSCALL	System Call	epc=pc; pc=0x3c	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	0	1	1	0	0	1	1	0	0	1	1
Memory	LB rt,offset(rs)	Load Byte	rt=*(char*)(offset+rs)	1	0	0	0	0	0	rs	rs	rs	rs	rs	rt	rt	rt	rt	rt	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset
	LBU rt,offset(rs)	Load Byte Unsigned	rt=*(Uchar*)(offset+rs)	1	0	0	1	0	0	rs	rs	rs	rs	rs	rt	rt	rt	rt	rt	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset
	LH rt,offset(rs)	Load Halfword	rt=*(short*)(offset+rs)	1	0	0	0	0	1	rs	rs	rs	rs	rs	rt	rt	rt	rt	rt	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset
	LHU rt,offset(rs)	Load Halfword Unsigned	rt=*(Ushort*)(offset+rs)	1	0	0	1	0	1	rs	rs	rs	rs	rs	rt	rt	rt	rt	rt	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset
	LW rt,offset(rs)	Load Word	rt=*(int*)(offset+rs)	1	0	0	0	1	1	rs	rs	rs	rs	rs	rt	rt	rt	rt	rt	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset
	SB rt,offset(rs)	Store Byte	*(char*)(offset+rs)=rt	1	0	1	0	0	0	rs	rs	rs	rs	rs	rt	rt	rt	rt	rt	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset
	SH rt,offset(rs)	Store Halfword	*(short*)(offset+rs)=rt	1	0	1	0	0	1	rs	rs	rs	rs	rs	rt	rt	rt	rt	rt	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset
	SW rt,offset(rs)	Store Word	*(int*)(offset+rs)=rt	1	0	1	0	1	1	rs	rs	rs	rs	rs	rt	rt	rt	rt	rt	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset	offset

Figure 3.7: Plasma Instruction Encoding.

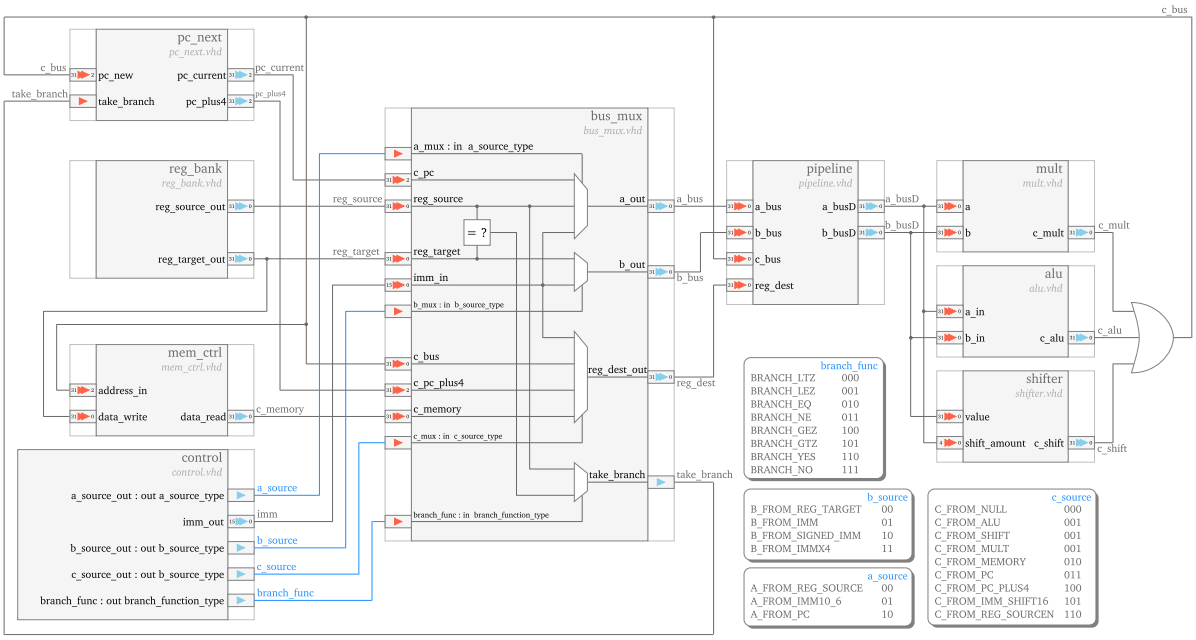

Figure 3.8: Internal CPU bus.

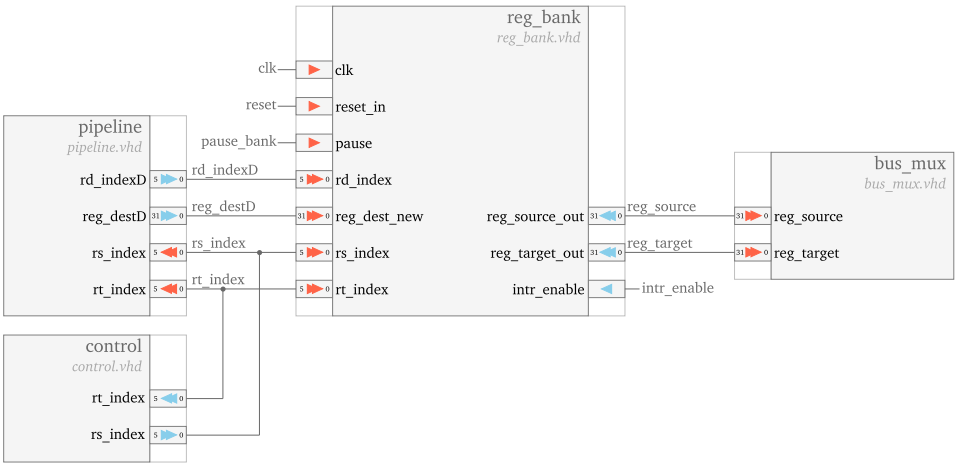

Figure 3.9: Register file interconnection.

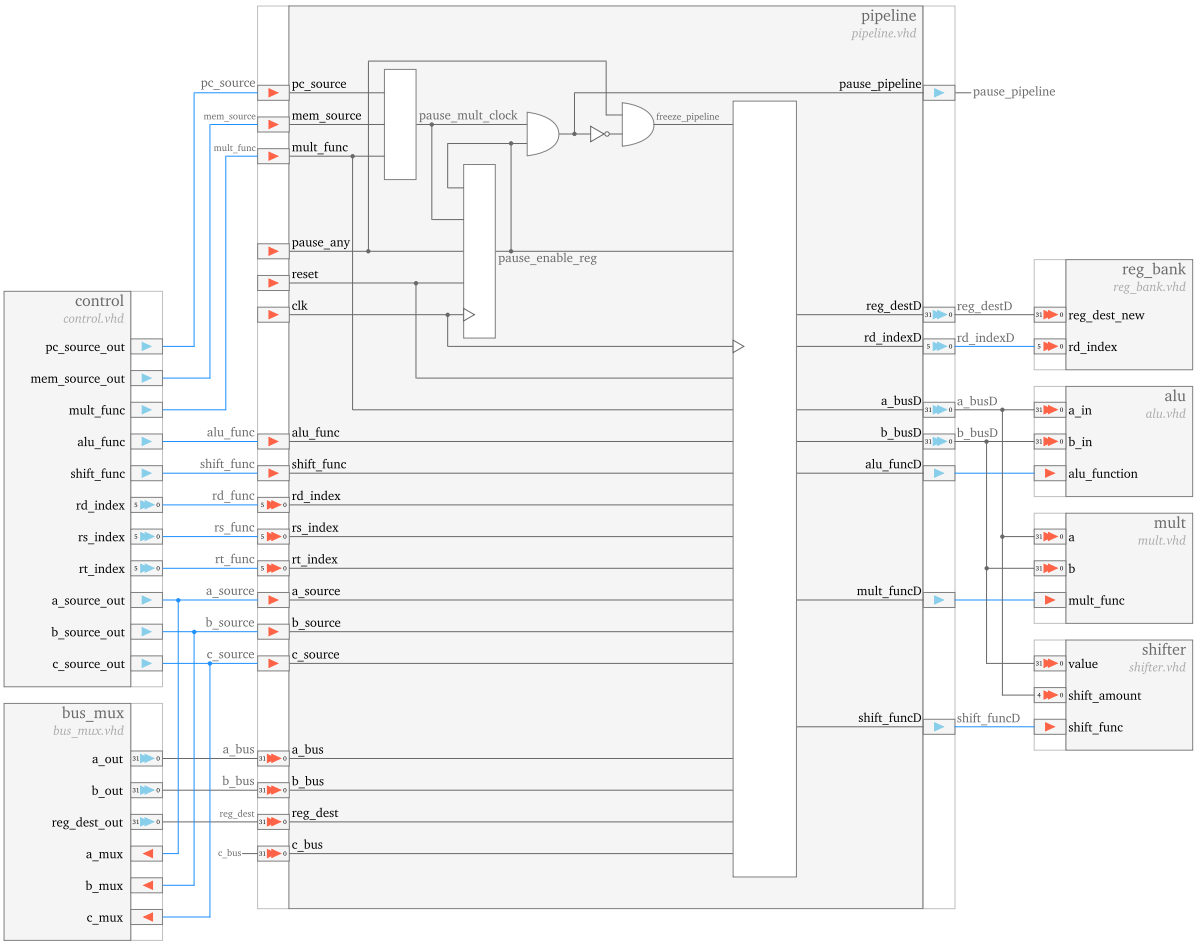

Figure 3.10: Pipeline entity.

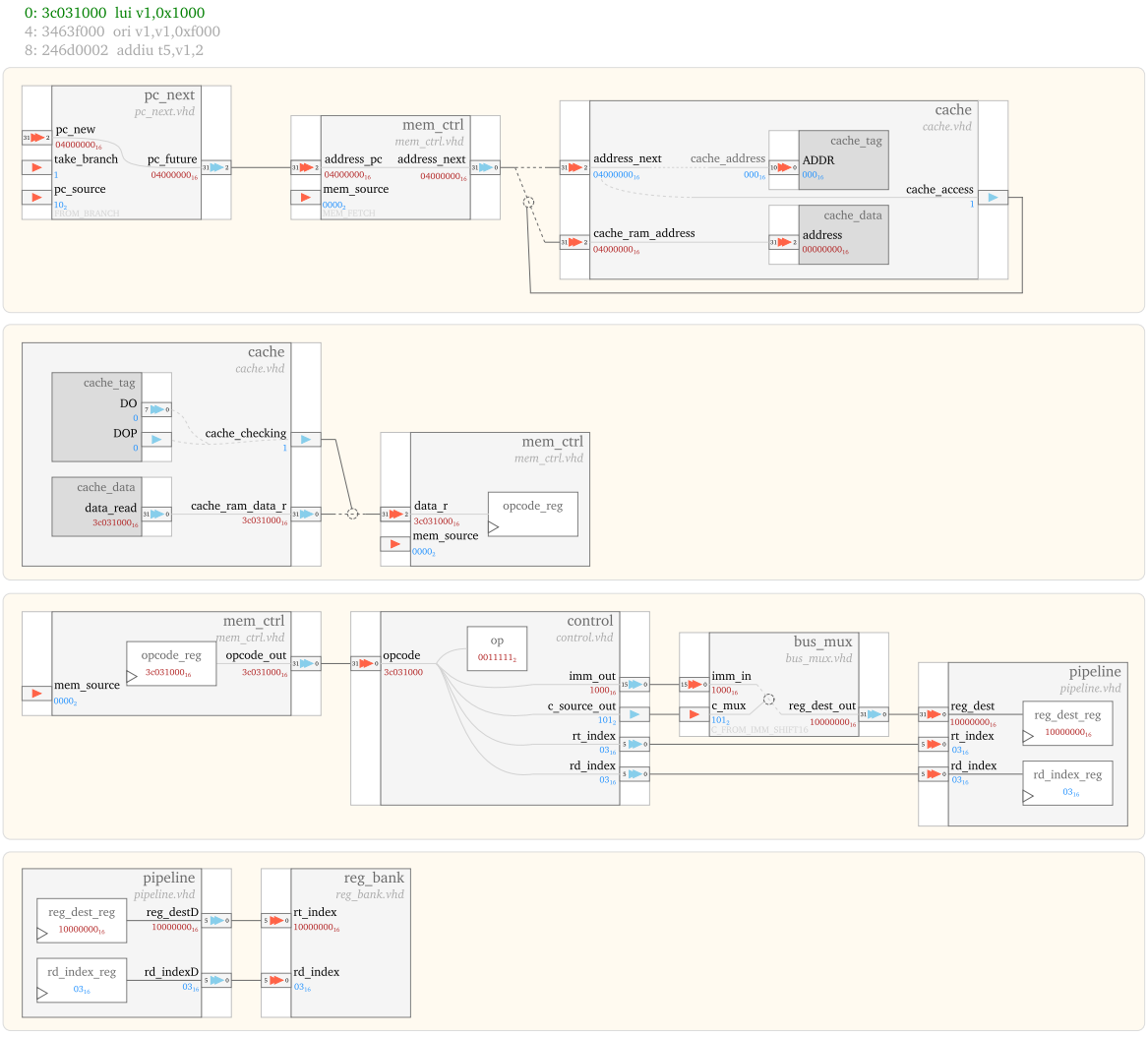

Figure 3.11: Instruction life-cycle.

4

List of Files

File	Purpose
pdp_ISE	
ml410.ucf	Board constraints file (FPGA pin mappings, timing constraints)
pdp_ise.xise	Xilinx ISE project file
vcd_hierarchy_edit.sh	Script for .vcd post processing
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 decoding
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
power.do	Simulation script for power evaluation
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 an 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 and you have a clear plan and view on what you want to achieve. Creating checkpoints for your design can be quite helpful as they can allow 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 is an intermediate milestone meeting scheduled on May 16^{th} or 17^{th} in which those things should be discussed. Prior and after that, you may consult the Processor Design team members (rooms HB10.080 - HB10.090). Please bear in mind that abusing that service may have negative consequences on your grade. Our strategy is to encourage you to have initiative, to be creative, and autonomously solve the problems you may encounter during this project.

Bibliography

[1] *TUD vpn manuals,* http://studenten.tudelft.nl/en/students/ict/help/manuals/ict-manuals-staff-students/virtual-private-network/.

[2] *GCC Machine Descriptions,* https://gcc.gnu.org/onlinedocs/gccint/Machine-Desc.html.

[3] Xilinx UG070, *Virtex-4 FPGA User Guide,* https://www.xilinx.com/support/documentation/user_guides/ug070.pdf.

[4] Xilinx UG619, *Virtex-4 Libraries Guide for HDL Designs,* http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/virtex4_hdl.pdf.

[5] *Dropbox,* https://www.dropbox.com/.

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

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

[8] Xilinx UG085, *ML410 Embedded Development Platform User Guide,* http://www.xilinx.com/support/documentation/boards_and_kits/ug085.pdf.

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

[10] *MT46V16M16-5B DDR SDRAM Datasheeet,* http://media.digikey.com/pdf/Data%20Sheets/Micron%20Technology%20Inc%20PDFs/MT46V64M4,32M8,16M16.pdf.