

Universidade Federal do Paraná

Departamento de Informática

Roberto A Hexsel

cMIPS – a synthesizable VHDL model
for the classical five stage pipeline

Relatório Técnico
RT_DINF 001/2016

Curitiba, PR
2016

cMIPS – a synthesizable VHDL model for the classical five stage pipeline

Roberto A Hexsel
Universidade Federal do Paraná
Departamento de Informática
roberto@inf.ufpr.br

Introduction

I have been teaching Computer Architecture for nearly twenty years on courses that make heavy use of Patterson & Hennessy's *Computer Organization & Design: The Hardware/Software Interface* [PH09], and I always felt the need for some form of the processor for the students to play with. In the last few years we introduced VHDL in the two courses that precede CA and thus our students can now study and play with models for the implementation of the MIPS instruction set.

Some months ago I started to write a model for the processor with the intention that the model should resemble the design presented in the book as closely as possible. One of my objectives was for the model to run code compiled by GCC – hence it is a complete implementation of the MIPS32r2 instruction set [MIPS05b], with all the attending complexities of a real-life instruction set. The data path has all the interlocks and forwarding paths needed for correct and efficient execution of compiled C code. All the user-level integer instructions are supported.

The result of the development effort described here is shown in Figure 1, which is gtkwave's screen from a simulation run. The colored rectangles highlight the progress of one instruction down the pipeline stages. In our environment, the students are presented with the simplest development tools: the ghdl VHDL compiler [ghdl12] and gtkwave to display the timing diagrams that result from the simulations [gtkwave12]. C and assembly programs are compiled/assembled with gcc and binutils [gcc99, binutils98].

The control processor, or the Coprocessor 0 (COP0) [MIPS05c] is partially implemented: the six hardware interrupts, two software interrupts, and the non mask-able interrupts are implemented, in *Interrupt Compatibility Mode*. The Memory Management Unit comprises an 8 entry fully associative TLB. There is enough machinery to support a full blown Unix-like operating system.

This space intentionally left blank.

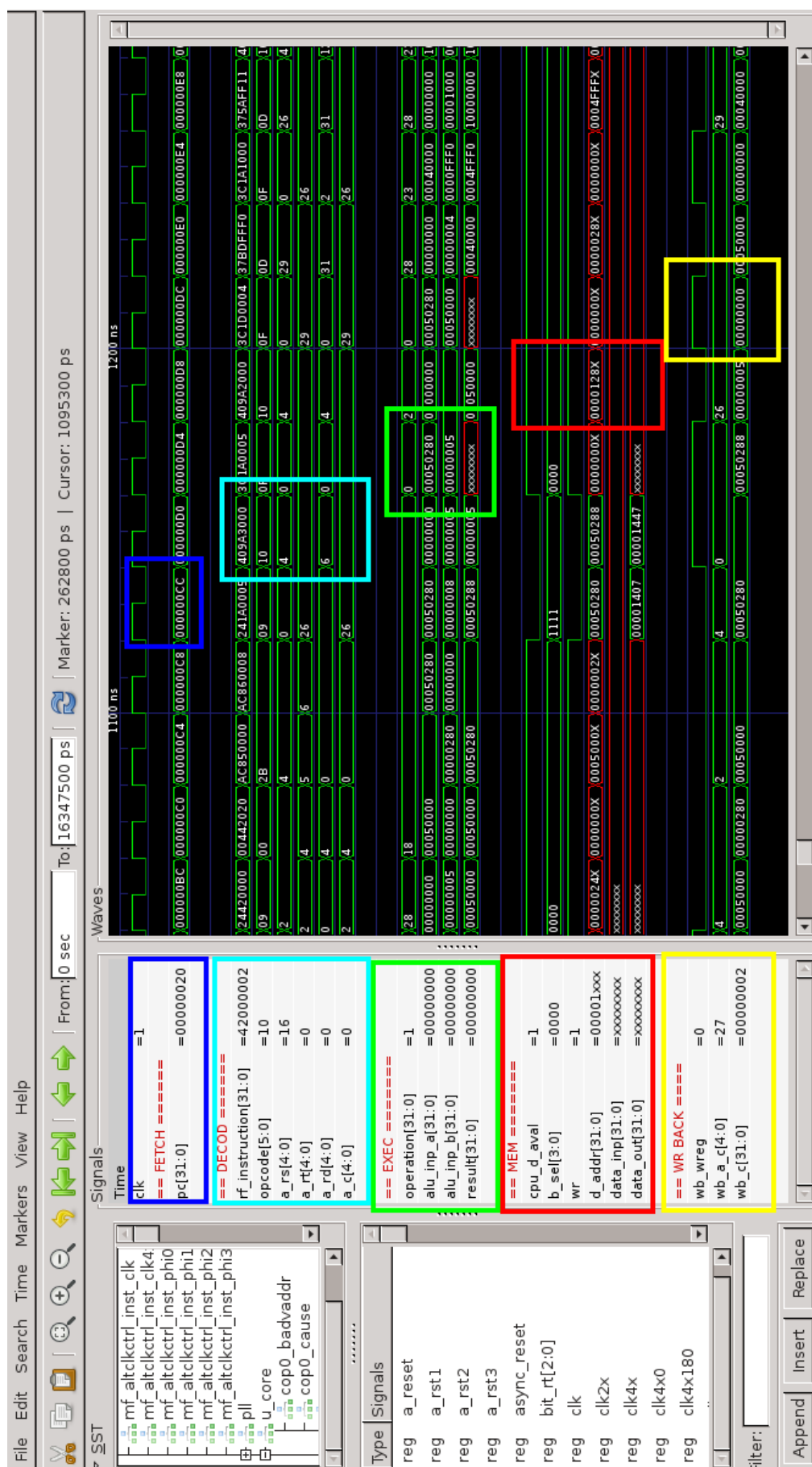


Figure 1: Sample simulation run displayed with gtkwave.

The simulation version of the testbench is a “simple computer” with the processor core, instruction and data caches, RAM and ROM and five ‘peripherals’. A block diagram of the ‘computer’ is shown in Figure 2. The peripherals are: one to print on the simulator’s standard output; one for reading from an input file; one for writing to an output file; a counter that generates an interrupt after a specified number of clock cycles; and a simple UART along with a remote UART it can communicate with, not shown in the diagram. What appears on the diagram as a memory bus is a set of multiplexers.

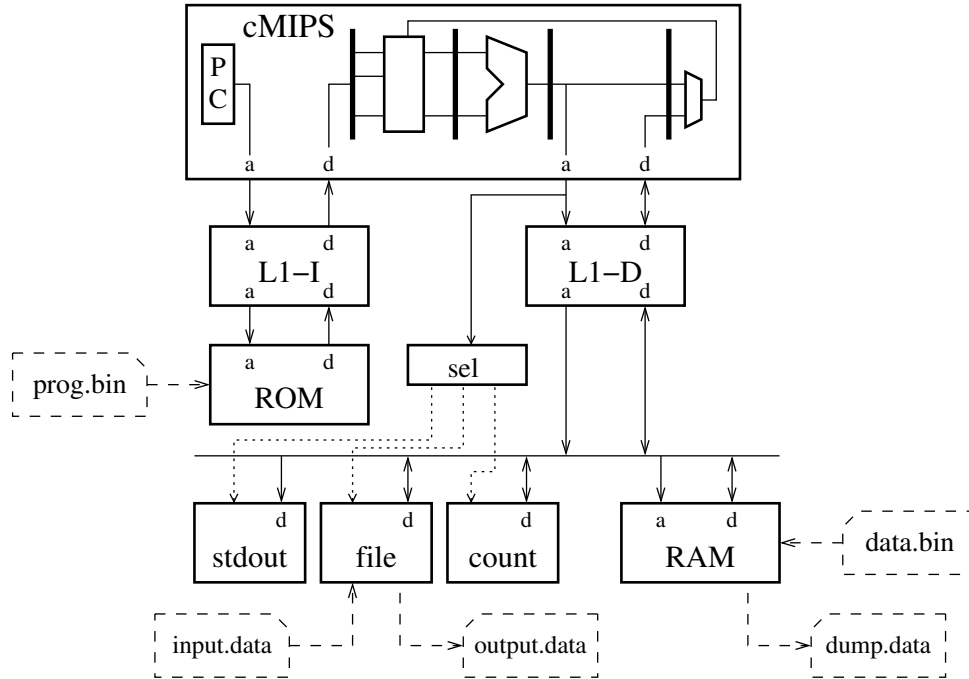


Figure 2: Block diagram of the “simple computer” in the testbench.

The diagram also shows the files needed for the model to run C/assembly programs. The code is input into the simulation ROM as file `prog.bin` and the RAM may be initialized with the contents of file `data.bin`. These two files are generated by the compilation and assembly scripts, which are described later. The simulator can read data from a binary input file (`input.data`) and can write binary results to `output.data`. The contents of the RAM can be copied to file `dump.data`. These files are shown as dashed boxes.

The model was conceived primarily as an aid to teaching through simulation and was later adapted for synthesis on a Altera Cyclone IV FPGA. The assembly/compilation scripts generate the files `ROM.mif` and `RAM.mif`, as needed by Altera’s tools, and these two are replacements for `prog.bin` and `data.bin`. The synthesis model does not provide for input/output file operations; it supports a 12 key keypad, LCD display, and serial interface (UART). The interfaces to the SDRAM controller, VGA output, microSD card reader and Ethernet connection will be added soon.

The source code for the simulation version is available at <https://github.com/rhexsel/cmips>. Some files for the synthesis version are missing from the repository – please do request these from the author as needed.

1 Pipeline Model – User Instructions

Figure 3 shows a block diagram of the processor core, with the user-level pipeline at the top, and the control pipeline (Coprorocessor 0) at the bottom. The user-level pipeline stages are described in what follows, and the control, or system-level, pipestages are described in Section 2.

The datapath for the user-level instructions has all the interlocks and forwarding paths for the correct and efficient execution of assembly and C code.

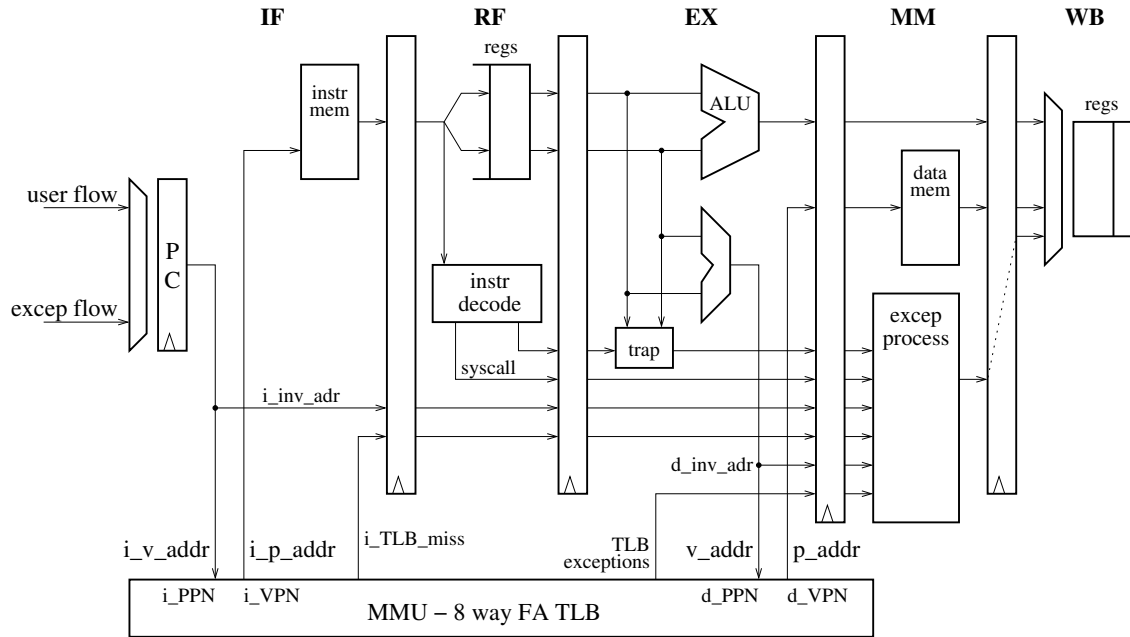


Figure 3: cMIPS user level (top) and Coprocessor 0 (bottom) pipeline.

Figure 4 shows a block diagram of the first two pipeline stages, instruction fetch (IF), instruction decode and register fetch (RF), and Figure 5 shows a block diagram of the last three pipe stages, Execution (EX), access to memory (MM) and write-back (WB). The diagrams show the names of the ‘important’ signals, which are those needed for debugging, or to follow an instruction as it progresses down the pipeline. Core interface signals are shown in red. The diagram in Figure 7 shows the signals connecting pipeline and TLB.

Instruction Fetch – IF The memory interface stalls the pipeline if the cache/ROM asserts the wait/ready signal (*rdy*). Access to instructions is aligned to word boundaries.

Instruction Decode and Register Fetch – RF The decoding logic employs three tables and combinational logic. There is forwarding through the register bank; register contents are read on the falling edge of the clock signal (ϕ_2), and are updated on the rising edge of the clock (ϕ_0).

All branches and jumps are resolved in the second stage and there are forwarding paths from the memory stage to the comparator inputs, and an interlock for values that should come from the execution stage. There are interlocks for jump and branch delay-slots, and for load delay-slots.

Execution – EX There is a complete set of forwarding paths onto the Arithmetic and Logic Unit (ALU) inputs, and a hazard detection unit stalls the pipeline until all data dependencies are cleared.

The HI and LO registers are implemented inside the ALU and there is no need for interlocking between mult/div and mflo or mfhi as the operations complete in one clock cycle – *notice that this is not the behavior specified by MIPS32r2*. In the synthesis model division takes four clock cycles and this instruction has not been implemented, as of May 21, 2021.

Memory – MM During reset, both the RAM and the ROM are initialized from the files data.bin and prog.bin, respectively. These files are read from the current directory, with respect to the GHDL simulator.

In the synthesis version, memory is initialized from files ROM.mif and RAM.mif, generated with the compilation scripts.

The ROM is word-addressed and read-only. The RAM is byte-addressed and supports partial-word writes. Partial-word reads are handled by the memory pipe-stage. The signal byte_select defines, to the external interface, the width of the reference, as well as which portion of the word is to be updated.

The RAM memory interface is similar to that of the IF stage and the pipeline is stalled when the I-cache/ROM and/or D-cache/RAM assert their wait/ready signals.

Write Back – WB This stage is comprised by a multiplexer that selects the value to be written to the register bank.

This space intentionally left blank.

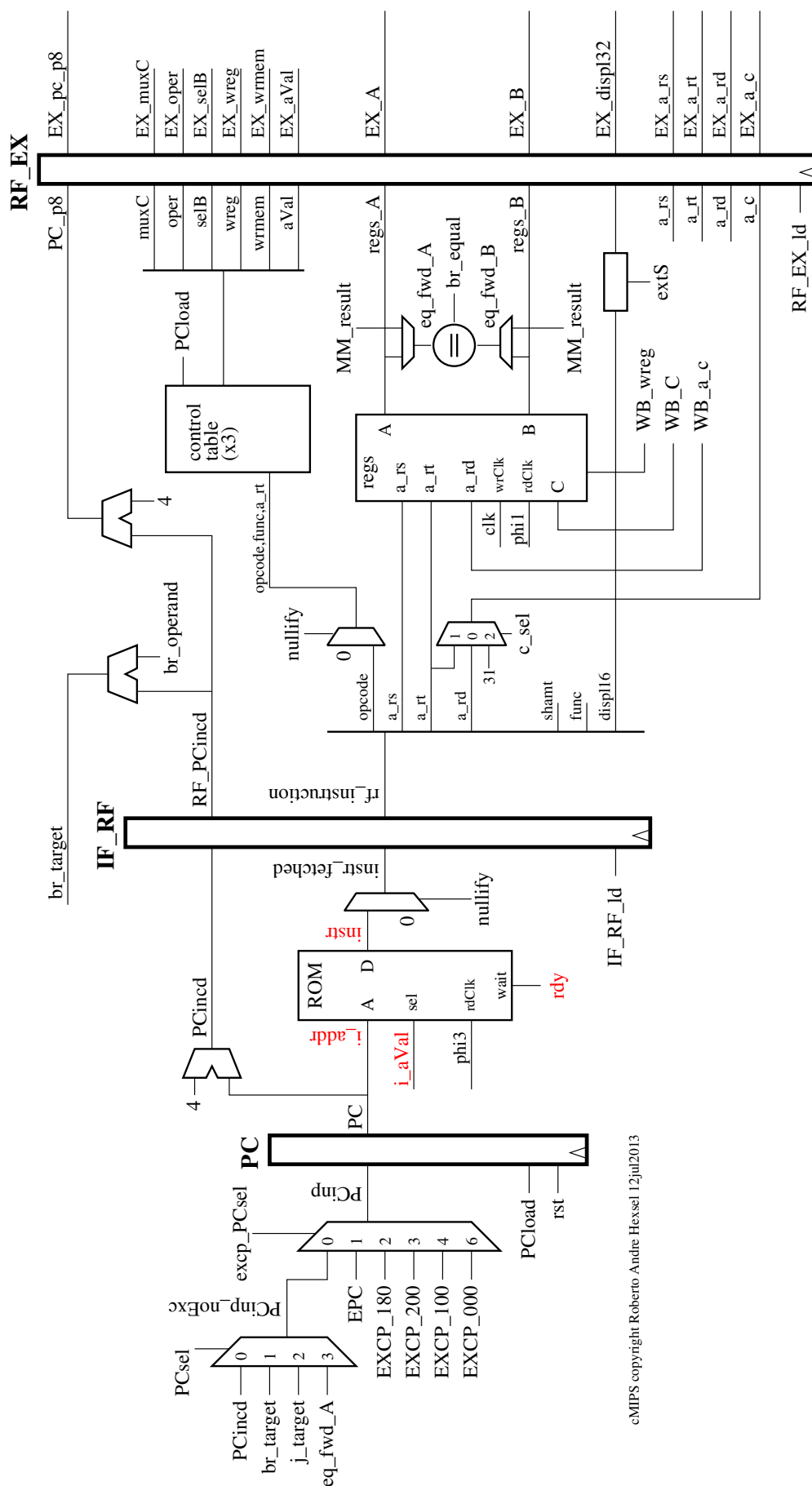
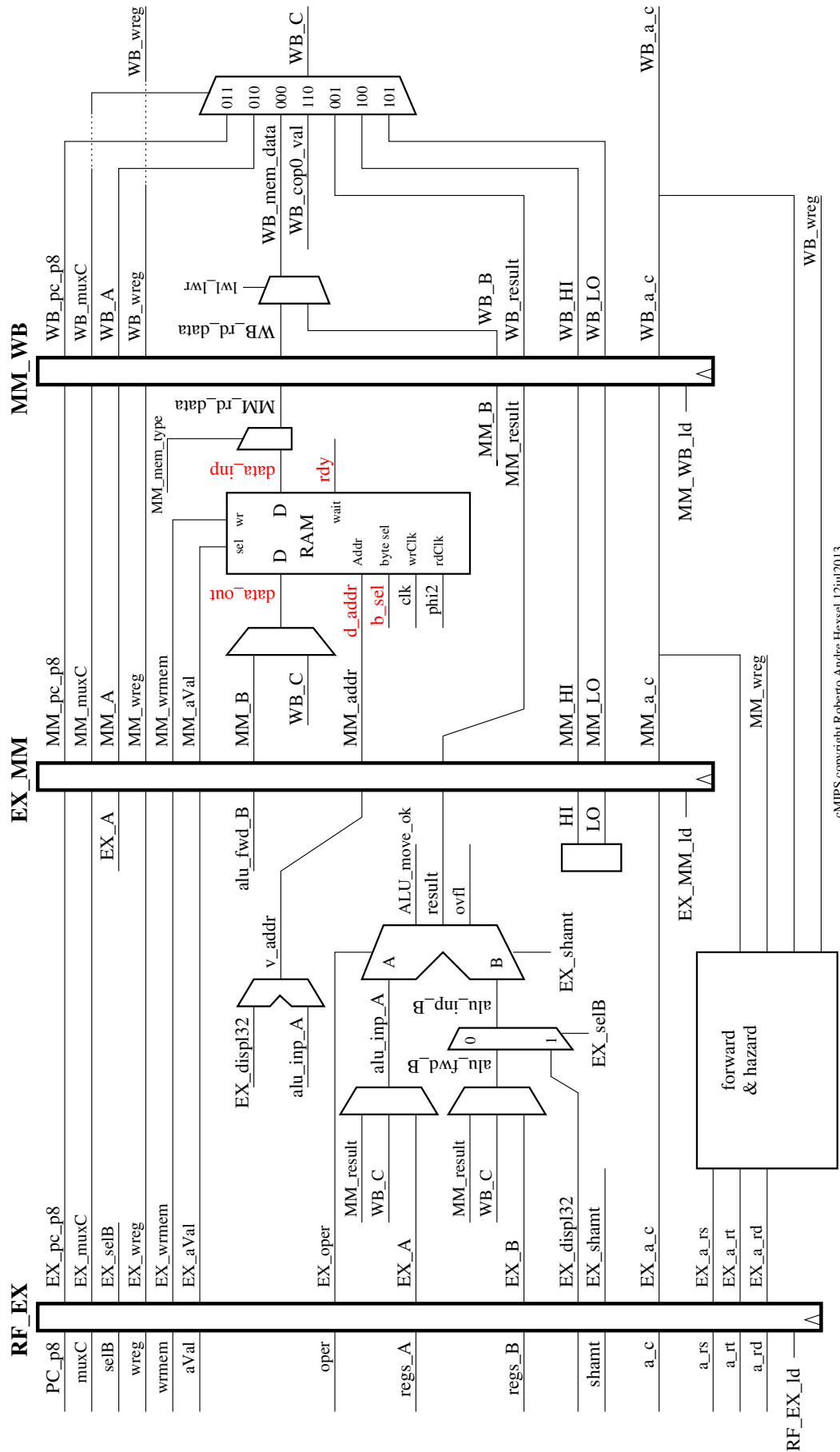


Figure 4: Block diagram of the front end: instruction fetch and decode.



cMIPS copyright Roberto Andre Hessel 12jul2013

Figure 5: Block diagram of the back end: execution, memory and write-back.

2 Coprocessor 0 – System Control Instructions

Figure 6 shows a block diagram of the four stages of the control pipeline, *viz* exception instruction fetch (EXCP_IF), exception decode (EXCP_RF), exception execution. and exception memory–result (EXCP_MM). The Coprocessor 0 (COP0) comprises several registers and extensive control and interlocking logic.

Exception Instruction Fetch – EXCP_IF The control logic steers the appropriate value to the PC input, which can be a ‘normal’ next instruction address, or an exceptional address, which can be the Exception PC (EPC), or one of the exception entry addresses EXPC_000 (TLBrefill), EXPC_100 (CacheError), EXPC_180 (general exception handler), EXPC_200 (interrupt handler), or the non-mask-able interrupt/reset address, EXPC_BFC0.

The address generated by PC is used to probe the TLB. If there is a miss, the exception is signalled and handled when the faulting instruction arrives at (EXCP_MM). The three instructions that follow the faulting instruction are nullified.

Exception Decode – EXCP_RF The control instructions are decoded at this stage.

Exception Execution – EXCP_EX Overflow exceptions and the trap instruction are evaluated at this stage. If an exception is taken, it is handled at EXCP_MM, and the instructions at IF, RF and MM are nullified.

The virtual address for memory references is computed and the TLB is probed for a valid mapping. If there is a TLB miss, the memory reference is nullified, as are the three instructions that follow the faulting memory reference.

Exception Memory-Result – EXCP_MM The COP0 registers are updated at this stage. If an exception or interrupt is taken, the pipeline is stalled for three cycles to clear all control and instruction hazards, and the instructions at IF, RF and MM are nullified.

The six hardware interrupts, two software interrupts, and the non mask-able interrupt are implemented in *Interrupt Compatibility Mode*. A software interrupt, generated with a mtc0 instruction, takes effect 3 cycles after the mtc0.

2.1 COP0 resources

A subset of the Coprocessor 0 resources are implemented. The subset was implemented as specified in [MIPS05c] and that document should be consulted for writing code to access COP0 registers.

The control instructions break, syscall, trap, mfc0, mtc0, eret, ei, di, tlbr, tlbp, tlbbi, tlbbw, and ehb are implemented. These provide enough functionality for executing a full-blown Unix-like operating system on the processor model. The wait instruction is used to abort the simulation, and is therefore not implemented as specified in [MIPS05c].

The COUNT register counts clock cycles, and the COMPARE register can be written with a 32 bit number to generate a periodical interrupt (hardware interrupt level 5, int_req(7)) when the value in COUNT equals the value in COMPARE.

The STATUS register determines the execution mode (user/kernel), whether interrupts are enabled etc.

The CAUSE register indicates the cause of the exception or interrupt. Once an interrupt or exception is taken, the CAUSE register is only updated after being read by an `mfc0 $r,$13` instruction, thus holding information on the event which caused the disruption of the normal execution flow.

The EPC register holds the address of the instruction that was not executed because of an exception or interrupt.

The CONFIG register holds the configuration of the processor and caches.

The LLaddr holds the effective address of the last `ll` (load linked) instruction.

The MMU/TLB registers Index, Random, EntryHi, EntryLo0, EntryLo1, BadVAddr and Context are implemented as part of the MMU.

See [MIPS05c] for examples of assembly code that references the COP0 registers. Simpler code fragments can be found in `tests/*.s` – the files have fairly obvious names.

This space intentionally left blank.

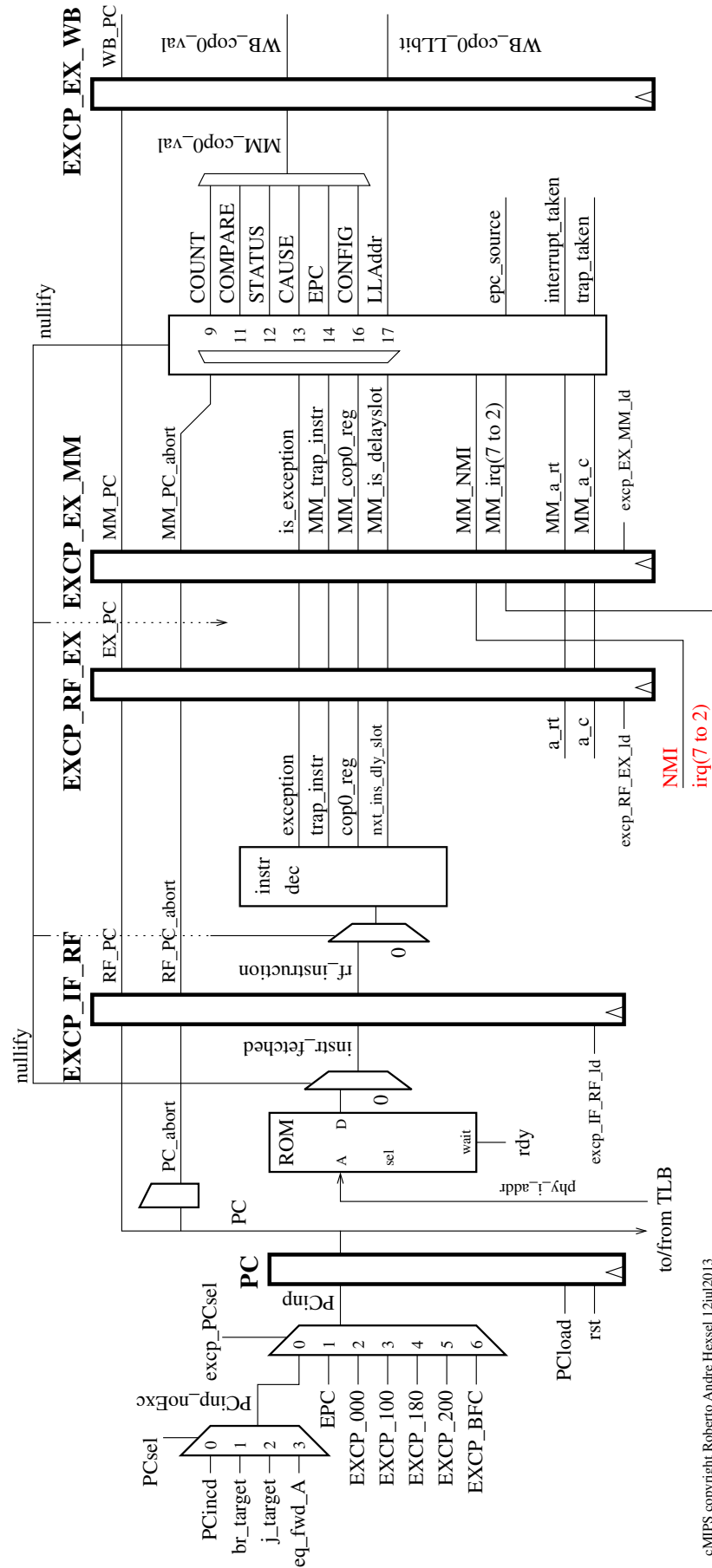


Figure 6: Block diagram of the control pipeline (COP0).

2.2 Memory Management Unit and the TLB

The simulation model includes an 8 entry fully associative TLB that supports 4 Kbyte pages. The MMU does not check the 3 most significant bits that divide the address space into user/kernel mapped/unmapped regions. Thus, the address space is flat and user-to-kernel protection violations are not checked. Figure 7 shows a block diagram of the signals that comprise the virtual to physical translations.

The TLB entries are initialized to map 6 pages of ROM, 8 pages of RAM and 2 pages for the peripherals. Table 1 shows the contents of the TLB after initialization. The register Wired should be initialized to 2 to ensure that the first two pages of ROM and the peripherals are always mapped and do not generate TLB misses. If these pages become unmapped, the code at the bottom of the ROM addresses, which handles TLB and other exceptions, will not be available to fix the faulting address maps, and an infinite loop ensues: fault \rightarrow fault ...

Table 1: Initial page mappings on the TLB.

<i>Entry</i>	<i>type</i>	<i>contents</i>	<i>addresses</i>
0	ROM	ROM pages 0,1	0x0000.0000–0x0000.1fff
1	I/O	I/O pages 0,1	0x3c00.0000–0x3c00.1fff
2	ROM	ROM pages 2,3	0x0000.2000–0x0000.3fff
3	ROM	ROM pages 4,5	0x0000.4000–0x0000.5fff
4	RAM	RAM pages 0,1	0x0004.0000–0x0004.1fff
5	RAM	RAM pages 2,3	0x0004.2000–0x0004.3fff
6	RAM	RAM pages 4,5	0x0004.4000–0x0004.5fff
7	RAM	RAM pages 6,7	0x0004.6000–0x0004.7fff

The addresses shown in Table 1 are defined in file `vhdl/packageMemory.vhd`: the base of the RAM pages is at `x_DATA_BASE_ADDR`, and the base of the I/O addresses is in `x_IO_BASE_ADDR`. If these addresses are changed, the TLB will be initialized with the new values once the VHDL code is recompiled.

The register Context has 16 bits to point to the top of the page table, rather than the 9 bits prescribed in [MIPS05c], pg. 67. This is needed to support address spaces smaller than 4 Gbytes, so that the PageTable may reside in low(ish) addresses.

The initialization code at `include/start.s` sets the top of the stack at address $(x_DATA_BASE_ADDR + x_DATA_MEM_SZ - 16)$. The constants `x_DATA_BASE_ADDR` and `x_DATA_MEM_SZ` are defined in file `vhdl/packageMemory.vhd`.

This space intentionally left blank.

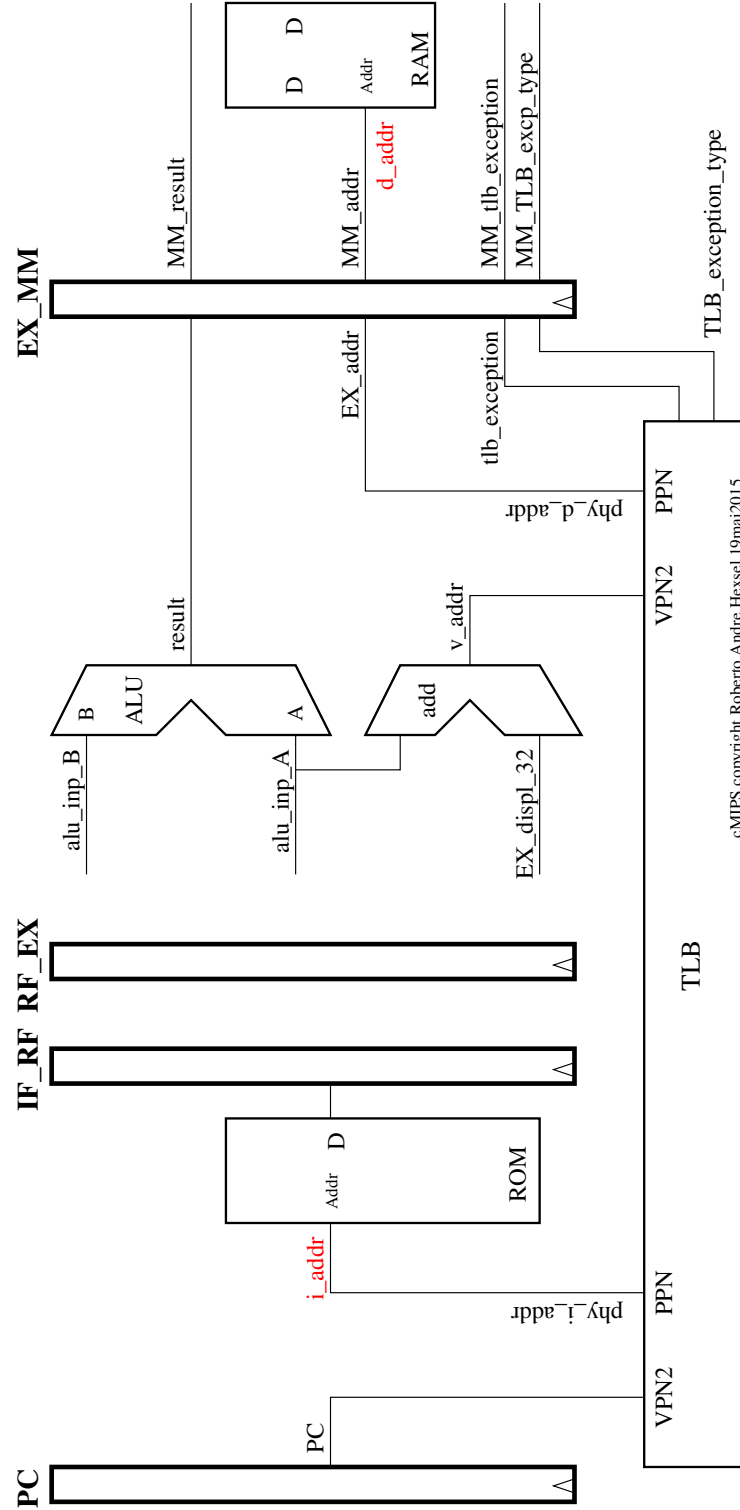


Figure 7: TLB-pipeline connections.

3 The Entities

In the comments, $act=\{0,1\}$ means that the signal is active in 0 or in 1, respectively. `regN` is a width N `std_logic_vector`.

Programa 1: Entity for the processor core.

```

entity core is
  port (
    rst      : in      std_logic; — reset, act=0
    clk      : in      std_logic; — pipeline clock
    phi1     : in      std_logic; — 1/4 cycle out of phase pipeline clock
    phi2     : in      std_logic; — 2/4 cycle out of phase pipeline clock
    phi3     : in      std_logic; — 3/4 cycle out of phase pipeline clock
    i_aVal   : out     std_logic; — instruction address valid, act=0
    i_wait   : in      std_logic; — instr memory not ready, act=0
    i_addr   : out     reg32;   — instruction address
    instr    : in      reg32;   — the instruction proper
    d_aVal   : out     std_logic; — data address valid, act=0
    d_wait   : in      std_logic; — data memory not ready, act=0
    d_addr   : out     reg32;   — data address
    data_inp : in      reg32;   — data input
    data_out : out     reg32;   — data output
    wr       : out     std_logic; — write, act=0
    b_sel    : out     reg4;    — byte selection, for LB,SB,LH,SH
    nmi      : in      std_logic; — non mask-able interrupt
    irq      : in      reg6);   — 6 hardware interrupts
  end core;

```

Programa 2: Entity for the synchronous ROM.

```

entity ROM is
  generic (LOAD_FILE_NAME : string); — ROM initialization file
  port (rst      : in      std_logic; — reset, act=0
        clk      : in      std_logic; — wait state machine's clock
        sel      : in      std_logic; — chip select, act=0
        rdy      : out     std_logic; — data not ready (waiting), act=0
        phi2     : in      std_logic; — address strobe
        addr     : in      reg32;   — address
        data     : out     reg32);  — instruction
  end ROM;

```

Programa 3: Entity for the synchronous RAM.

```

entity RAM is
  generic (LOAD_FILE_NAME : string); — RAM initialization file
  port (rst      : in      std_logic; — reset, act=0
        clk      : in      std_logic; — wait state machine's clock
        sel      : in      std_logic; — chip select, act=0
        rdy      : out     std_logic; — data not ready (waiting), act=0
        wr       : in      std_logic; — write, act=0
        phi2     : in      std_logic; — address strobe
        addr     : in      reg32;   — address
        data_inp : in      reg32;
        data_out : out     reg32;
        byte_sel : in      reg4);   — byte/half/word select
  end RAM;

```

4 VHDL Sources

The VHDL source files are described below. The VHDL source files are stored in directory `vhdl`.

- `packageWires.vhd` Several constants and data types are defined in this file – the exception being cache/TLB parameters and values related to COP0. A few functions to display `std_logic_vectors` on the terminal are provided to help in debugging.
- `packageMemory.vhd` Defines the addresses for RAM, ROM, and peripherals, plus TLB and cache design parameters.
- `packageExcp.vhd` Defines the addresses and constants needed to access COP0 resources.
- `altera.vhd` Simulation models for Altera’s own components (PLLs and clock drivers).
- `aux.vhd` Auxiliary models, such as adder, 32-bit register, ring-counter to generate the four-phases clock, flip-flops.
- `core.vhd` Processor core. The code attempts to follow “the book” [PH09]. Each pipeline stage is delimited by a pair of pipeline registers and all the combinational circuits (and some state) are contained “within” the pipe-stage.
- `exception.vhd` Contains the registers for the control pipeline. These carry the information regarding the exceptions that arise as an instruction travels down the pipeline. The entities and architecture are very large yet simple. These were put on a separate file to ease the navigation though the code.
- `cache.vhd` Contains models for the instruction and data caches. There are two “fake caches” that just pass along all the signals.
- `memory.vhd` Contains simulation models for ROM and RAM memory.
- `ram.vhd` Contains a synthesizable model for RAM memory.
- `rom.vhd` Contains a synthesizable model for ROM memory.
- `io.vhd` Models for the peripherals, both simulation and synthesis.

For simulation the models are: one that writes an integer to the simulator’s standard output, one that writes a character to the simulator’s standard output, one that reads one integer from a file, and one that writes one integer to a file, and one that displays hit/miss statistics from the caches.

For simulation and synthesis the models are: an external 30 bit counter that generates an interrupt after a programmable number of clock cycles, a two-digit seven-segment LED display, an LCD display controller interface, a keyboard-/switches interface, and the UART’s bus interface.

See `include/cMIPSio.c` for the API to the peripherals.

- `pipestages.vhd` The pipeline registers are defined in this file. The entities and architecture are very large yet simple. These were put on a separate file to ease the navigation though the code of the processor core.
- `units.vhd` Models for the main functional units are defined in this file, namely the ALU and the register bank.
- `uart.vhd` Models for the UART and for the “remote computer”. The “remote computer” can read the file `serial.inp` and send its contents to the UART, or write the file `serial.out` with characters received from the UART.

The UART model is synthesizable whereas the “remote computer” is for simulation only.

`tb_cMIPS.vhd` The testbench declares and instantiates all components of the system, as well as reset and clock generator processes. The addresses for ROM, RAM and I/O are decoded in the testbench.

There is a simulation version (`vhdl/tb_cMIPS.vhd`) and a synthesis version, in `altera/tb_cMIPS.vhd`. The other files in the `altera` directory are those needed for synthesis.

5 Scripts

Several scripts were written to build the cMIPS model, cross-compile the test programs and run the simulations. Yes, maybe, one day these should all be merged into one big fat Makefile. The scripts are in the `bin` directory.

If the command line argument `-h` is given to any of the scripts, an usage message is printed on the screen, and the script exits.

Do not forget to add `/path/to/cMIPS/bin:.` to your `$PATH` variable.

You need `mips-gcc` and `Binutils` to compile your programs and run the test programs. See `docs/installCrosscompiler` for instructions on building and installing the cross compiler and toolchain.

`build.sh` (executable) Usage: `build.sh`

Compiles all VHDL sources and builds the simulator/model.

In case of trouble *not caused by poor VHDL coding*, remove the files `vhdl/.last_import` and `vhdl/work-obj93.cf`, then run `bin/build.sh` again.

`assemble.sh` (executable) Usage: `assemble.sh [-v] [-O 2] file.s`

The path to GCC and binutils is (automagically) set at the top of the script.

Given an assembly source file, produces `prog.bin` and `data.bin` to be input by the VHDL model. `mips-objcopy` is used to produce the binaries.

If the command line argument `-v` (verbose) is given, `objdump` prints the `.text` and `.data` sections of the ELF, as well as the memory map produced by `mips-ld` in a file with same prefix as source, and suffix `.map`. If the argument `-O {0,1,2,3}` is given, `mips-as` uses that number as the optimization level, defaults to `-O1`.

The argument `-mif` generates the initialization files for synthesis.

Care must be exercised in assembling the test programs written in assembly: if the assembly code is optimized, the instructions might be reordered by the assembler and results may appear to be incorrect.

`compile.sh` (executable) Usage: `compile.sh [-O 2] [-v] file.c`

The path to GCC and binutils is (automagically) set at the top of the script.

Given a C source file, produces `prog.bin` and `data.bin` to be input by the VHDL model. See below for a description of the compilation/linking process.

If the command line argument `-v` (verbose) is given, `objdump` prints the `.text` and `.data` sections of the ELF, as well as the memory map produced by `mips-ld` in a file with same prefix as source, and suffix `.map`. If the argument `-O {0,1,2,3}` is given, `mips-gcc` uses that number as the optimization level, defaults to `-O1`.

The argument `-mif` generates the initialization files for synthesis.

Care must be exercised in compiling test programs written in C that reference the peripherals. If the source code is optimized, the C commands that reference the I/O addresses might be optimized away, as for instance, a loop that continuously tests the same address is deemed useless by the compiler and removed from the executable. Not nice at all.

`include/cMIPS.ld` This ld script contains the definition of the memory map employed by `assemble.sh` and `compile.sh` to link the executables. The address definitions must be kept consistent with those in `packageMemory.vhd` since the addresses of memory are hardwired in the testbench. See `edMemory.sh`.

`edMemory.sh` (executable) Usage: `edMemory.sh [-v]`

This script modifies the header files so the address ranges defined in `packageMemory.vhd` are propagated to all appropriate files which are thus kept consistent. The script is not very intelligent: any changes to the address range definitions must keep the spacing, and naming, exactly as they are in `packageMemory.vhd` and in `include/cMIPS.{ld,h,s}`.

With argument `-v` prints the differences between new and old versions of modified files. This script is invoked automatically by `build.sh`, `run.sh`, `assemble.sh` and `compile.sh`, and normally there would be no need to invoke it directly.

`run.sh` (executable) Usage: `run.sh [-n] [-w] [-v v.sav]`

Builds the model and then runs the simulation. If given the argument `-n` sends the output of the simulator to `/dev/null`, discarding the (very large) file with timing information that would be input to `gtkwave`. If given the argument `-w`, it starts `gtkwave`. There is no (much) reason to supply `-n` and `-w` simultaneously. If given the arguments `-w -v v.sav`, then `gtkwave` is invoked with the visualization definitions in `v.sav`.

Notice that the simulator produced by `ghdl` expects `prog.bin`, `data.bin`, `input.data` and `output.data` to be in the current directory. The last two may be empty files.

`v.sav` Contains definitions for visualization with `gtkwave` such as the timescale and signals to be displayed. This can be a symbolic link to one of the save files supplied: one to ‘watch’ the pipeline, one for COP0, one for the TLB, one each for the transmission or reception by the UART.

`tests/doTests.sh` (executable) Usage: `./doTests.sh`

Performs all functional tests on the cMIPS simulator/model. See Section 7.

5.1 What about compilation?

The run time support for cMIPS is rather small and primitive, and no libraries are provided with the code (yet). Thus, all the code to be compiled must be self contained. `bin/compile.sh` takes a *single file* that must contain the function `main()` along with everything else that might be needed.

The C code for the I/O functions (`include/cMIPSio.c`) is compiled and linked with your “main” file. It contains functions to access the simulated peripherals such as for reading and writing to the simulator’s standard input and output, reading and writing files, making a dump of the RAM, and accessing the external counter. The code in this file *cannot* be

optimized because GCC will happily eliminate all those meaningless references to ‘memory’ – memory is in quotes because the I/O registers are memory mapped.

All initialization code must go into `include/start.s`. As of now, this file initializes the STATUS register, the stack pointer and pins down on the TLB the page with the stack. It also contains some interrupt/exception dispatch code. The function `exit()` flushes the pipeline and stops the simulation. The instruction `wait` is not implemented in the processor and is used solely to stop the simulation in an orderly fashion.

The interrupt handlers must be collected into file `include/handlers.s`. See the definition of signal `irq` in `vhdl/tb_cMIPS.vhd` for what device interrupts on which interrupt line/priority. Do a search for “`irq <=`”.

The symbol `_end` marks the highest RAM address used by your code. This symbol/variable is useful to determine the size of RAM. Of course, the top of the stack is allocated at the topmost RAM address, which should be at a safe distance above `_end`.

The file `include/cMIPS.ld` is a simple ‘driver’ for `mips-ld` and maps the executable sections into file `prog.bin`, and assorted data sections onto file `data.bin`.

6 File I/O from the VHDL model

The I/O address ranges are defined in `packageMemory.vhd`. Models for a few ‘peripherals’ are provided: one that writes an integer to VHDL’s simulator standard output, one that reads from file `input.data`, and one that writes to file `output.data`. These two files must be binary files, filled with 32-bit integers. To check the contents of `input.data` and `output.data` try

```
od -tx4 output.data | cut -d' ' -f 2-5 | sed -e '$d'.
```

The I/O functions should go into a library at some point; they are declared in `include/cMIPS.h` and their behavior is briefly described below.

```
// orderly end of simulation — strictly speaking, not a peripheral
extern void exit(int);
```

```
// prints an integer on VHDL's simulator standard output (stdout)
extern void print(int);
```

```
// prints a character on simulator's standard output (stdout)
// output is only displayed after a '\0' or an '\n' is sent out
extern void to_stdout(char c);
```

```
// writes an integer to file output.data
extern void writeInt(int);
```

```
// close file output.data
extern void writeClose(void)
```

```
// reads an integer from file input.data; returns 1 at EOF, 0 oth
extern int readInt(int*);
```

```
// dumps the contents of the entire RAM to file dump.data
extern void dumpRAM(void);
```

7 Tests

The tests directory contains several assembly and C source files, some scripts to automate the tests, and files with the expected results for the simulation runs.

Each assembly file tests some specific instructions or features of the processor, caches, or memory. The C files are simple benchmarks that perform more extensive testing of the processor/memory.

The script `doTests.sh` assembles (almost all) the assembly files and runs them on the VHDL model. Then it compiles (almost all) the C files and also runs them on the VHDL model. It should complete in a couple of minutes.

Each C file was compiled and run on a Linux desktop to generate the “correct” output. For a C source file *TST.c*, a file *TST.expected* was produced and stored in directory tests. Similarly for the assembly files. *TST.c* is then compiled with `mips-gcc` and run on `cMIPS`. The output of the test programs is normally sent to the simulator’s standard output.

`doTests.sh` runs the VHDL simulator and compares the simulated output *TST.simout* to the expected output *TST.expected*. If they are equal, the result is deemed to be correct; otherwise, the script aborts and the *diff* between *TST.simout* and *TST.expected* is printed on the screen.

The files `include/cMIPS.{ld,s,h}` contain the address ranges for instructions, data and IO devices. These files must be kept consistent with `packageMemory.vhd` since these addresses are hardwired in `tb_cMIPS.vhd`. All test files import one of `cMIPS.{s,h}`, and `assemble.sh` and `compile.sh` import `cMIPS.ld` to link the `cMIPS` executables. The script `edMemory.sh` automatically edits the pertinent files – do not mess with these files unless you understand the risks and know what you are trying to do.

How to run a test To run a test you shall:

1. add `/path/to/cMIPS/bin` and `/path/to/crosscompilers/bin` to your PATH shell variable – the scripts will do this automatically if the components of your pathnames do not have any spaces or weird characters;
2. export the pathname to your `cMIPS` installation to the shell scripts:
`cd /path/to/cMIPS ; export tree=$PWD`
or edit all scripts in `cMIPS/bin` to change the installation directory.
If the components of your pathnames do not have any spaces or weird characters, the scripts will do this automatically;
3. `cd tests` and pick a program to simulate/test, for instance, `insert.s`;
4. the source must be assembled and copied to the directory where the testbench will execute:
`assemble -v insert.s && mv prog.bin data.bin ..`
5. return to the top directory (`cd ..`) and perform the simulation:
`run.sh -w &`
this command rebuilds the `cMIPS` simulator, and starts `gtkwave` because of the `-w` argument;
6. if the model is correctly built, the simulation is run and `gtkwave` started. The standard output shows the program’s output, which should be identical with `tests/insert.expected`.

At the end of a (correct) simulation run the simulator prints a message similar to

the one below to standard error output, after printing out any (potentially) correct results to the standard output.

```
home/roberto/cMIPS/vhdl/core.vhd:808:7:@5387500ps:(assertion failure):
cMIPS BREAKPOINT at PC=0000011c opc=010000 fun=100000 brk=10000000000000000000
SIMULATION ENDED (correctly?) AT exit();
/home/roberto/cMIPS/tb_cmips:error: assertion failed
/home/roberto/cMIPS/tb_cmips:error: simulation failed
```

How to run all tests To perform all functional tests on the model you shall:

1. `cd tests;`
2. `./doTests.sh.` If all the tests produce the expected results, the terminal shows a list of the tests performed that produced “the expected results”. If any of the programs yields an output that differs from the expected, the script stops and displays the offending output on the simulator’s standard output.

8 Memory Interface

The timing of a memory access, both for ROM and RAM, is controlled by two signals, `aVal` and `wait`. In what follows the ROM interface is described; the interface of the RAM is similar. Figure 8 shows the timing diagram for a reference *without* wait-states, and a reference with *one* wait-state.

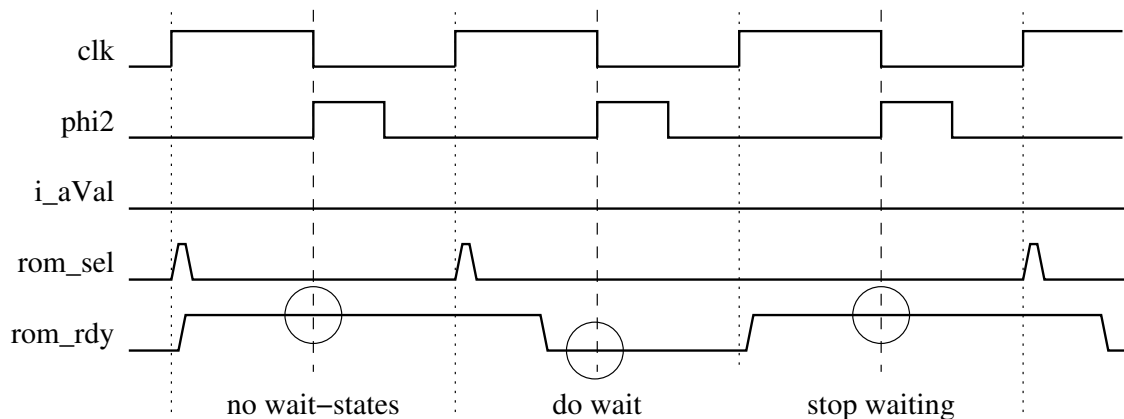


Figure 8: ROM access timing diagram with wait-states.

On the rising edge of `clk` the processor asserts `i_aVal` to signal the start of a new reference, while the content of the PC is output onto the address lines. During the first half of the cycle the address decoder compares the address range, and if no wait-state is needed, the signal `rom_rdy` must be deasserted – it must be deasserted prior to the rising edge of `phi2`.

When one or more wait-states are needed in order for the access to complete, the signal `rom_rdy` must be asserted prior to the rising edge of `phi2`, and is kept asserted, for as many processor cycles as necessary. The signal `rom_rdy` is sampled by the processor at the rising edge of `phi2`, and when it is deasserted, the reference completes on the next rising edge of `clk`.

On the ROM, the address is latched on the rising edge of `phi2` and the instruction is captured by the IF_RF pipeline register on the rising edge of `clk`.

On the RAM, the address is latched on the rising edge of `phi2`. On a store, the memory is updated on the rising edge of `clk`. On a read, the datum is captured by the `MM_WB` pipeline register on the rising edge of `clk`.

The instruction memory port (ROM) state machine freezes while there is a wait-state request to the data memory port (RAM); likewise, the data memory port also freezes while there is a wait-state request to the ROM port. The pipeline is fully interlocked to the memory ports, as well as to data and control dependencies.

The peripherals communicate with the processor through the data memory bus and the addresses of RAM or peripherals must be decoded from the data address bus. See the `data_addr_decode` and `inst_addr_decode` entities/architectures in the testbench.

8.1 Synthesis model

The model was conceived primarily as an aid to learning through simulation and was later adapted for synthesis on an Altera Cyclone IV FPGA. The synthesis model runs at 50Mhz, which is the rated clock speed of the Mercurio IV development kit [Macnica14].

The assembly and compilation scripts generate the files `ROM.mif` and `RAM.mif`, as needed by Altera's tools, and these two are replacements for `prog.bin` and `data.bin`, as described earlier. The synthesis model does not provide for input/output file operations; it does support a 12 key keypad, LCD display, and serial interface (UART). The interfaces to the SDRAM controller, VGA output and microSD card reader are under development.

8.2 The road ahead

On the software front, the obvious “next step” in the development of any processor implementation is the porting of an operating system (OS). As cMIPS was primarily intended as a tool for education, a fairly obvious OS choice would be XINU [Comer15]. Some may argue that any of the embedded versions of Linux *ought* to be the porting choice¹. A production system, whatever technical merits it may possess, is far too complex for the beginner and therefore not a conducive instructional medium.

On the hardware development front, at the top of the “to do list” is the synthesizable memory hierarchy, which includes caches, an SDRAM controller, and a microSD controller. These components make up a three-level memory hierarchy which can support a complete virtual memory system on the development kit.

This space intentionally left blank.

¹The susceptible reader can rest assured that the author has been a user and advocate for Linux for nearly one quarter of a century.

9 Ownership and Rights of Use

```
-- ++++++
--  cMIPS, a VHDL model of classical the five stage MIPS pipeline.
--  Copyright (C) 2013-2016  Roberto André Hexsel
--
--  This program is free software: you can redistribute it and/or modify
--  it under the terms of the GNU General Public License as published by
--  the Free Software Foundation, version 3.
--
--  This program is distributed in the hope that it will be useful,
--  but WITHOUT ANY WARRANTY; without even the implied warranty of
--  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
--  GNU General Public License for more details.
--
--  You should have received a copy of the GNU General Public License
--  along with this program.  If not, see <http://www.gnu.org/licenses/>.
-- ++++++
```

References

- [PH09] *Computer Organization & Design: The Hardware/Software Interface*. David A Patterson, John L Hennessy, 2009, 4th ed, Morgan Kaufmann, ISBN 9780123744937.
- [ghdl12] *GHDL – G Hardware Design Language*, Tristan Gingold *et. alli*, 2012, <<http://ghdl.free.fr>>, 11/7/2012.
- [gtkwave12] *GTKWave – Visualization tool for VCD, LXT, LXT2, VZT, FST, and GHW files*, Anthony Bybell, 2012, <<http://gtkwave.sourceforge.net>>, 11/7/2012.
- [gcc99] *GCC, the GNU Compiler Collection*, Richard Stallman *et. alli*, 1999, Free Software Foundation, <<http://gcc.gnu.org>>, 28/7/2013.
- [binutils98] *GNU Binutils*, Richard Stallman *et. alli*, 1999, Free Software Foundation, <<http://ftp.gnu.org/gnu/binutils>>, 28/7/2013.
- [MIPS05a] *MIPS32 Architecture for Programmers, Volume I: Introduction to the MIPS32 Architecture*, MIPS Technologies, Rev. 2.50, 2005.
- [MIPS05b] *MIPS32 Architecture for Programmers, Volume II: The MIPS32 Instruction Set*, MIPS Technologies, Rev. 2.50, 2005.
- [MIPS05c] *MIPS32 Architecture for Programmers, Volume III: The MIPS32 Privileged Resource Architecture*, MIPS Technologies, Rev. 2.50, 2005.
- [Ashenden08] *The Designer’s Guide to VHDL*, Peter J Ashenden, 2008, 3rd ed, Morgan Kaufmann, ISBN 9780120887859.
- [Macnica14] *Mercurio IV: Manual do Usuário*, DHW Macnica, 2014, <<http://www.macnicadhw.com.br/products/mercurion-4-devkit-board>>.
- [Comer15] *Operating System Design – The XINU Approach*, Douglas E Comer, 2015, 2nd ed, CRC Press, ISBN 9781498712439.