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

Roberto André Hexsel

Departamento de Informática

Universidade Federal do Paraná

81.531-990 – Curitiba, PR, Brazil

roberto@inf.ufpr.br

May 29, 2015

1   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 a model 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. The model contains a mixture of behavioral and RTL code.

One of my objectives was for the model to run code compiled by GCC and thus it contains an almost 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. The integer instructions are supported, with floating point as a future development.

Besides the processor, there are models for simple instruction and data caches[1] – both are direct mapped, and the data cache is write-through with no block allocation on write-misses. The data cache and memory support references to words, half-words and bytes.

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*. There is enough machinery to support a full blown Unix-like operating system.

The simulation version of the testbench comprises 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 1. 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 was modeled with multiplexers.

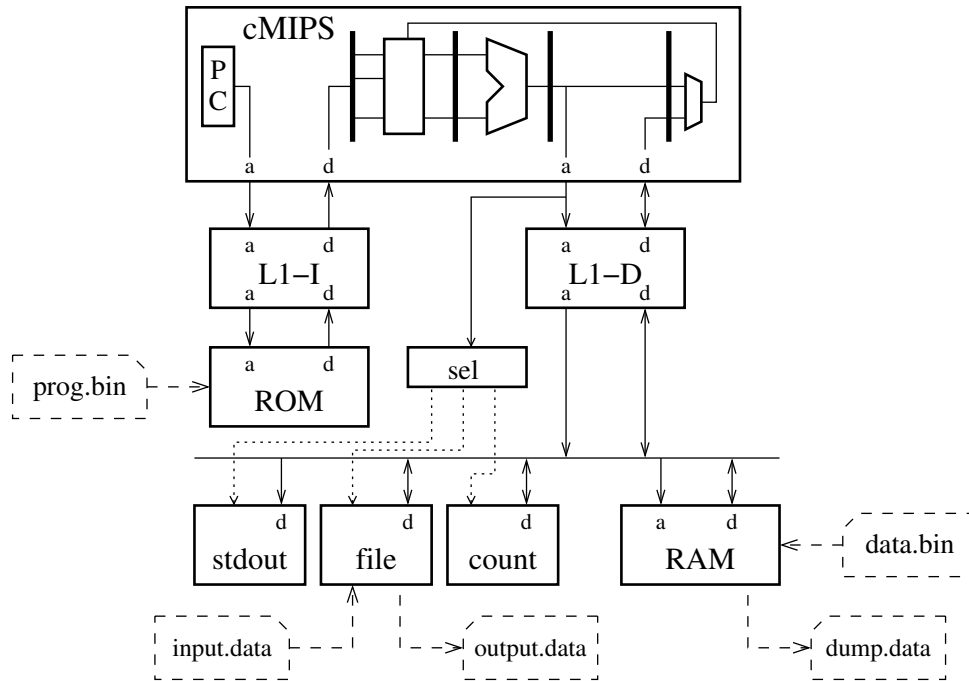[1]These models are not synthesis ready.

Figure 1: 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 designed 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 `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 input/output file operations; it supports a 12 key keypad, LCD display, and serial interface (UART). The interfaces to the SDRAM controller, microSD card reader and Ethernet connection will be added soon.

My VHDL sorcery book is the excellent [Ashenden08].

2 Pipeline Model

Figure 2 shows a block diagram of the first two pipeline stages, instruction fetch (IF), instruction decode and register fetch (RF), and Figure 3 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 pipe. Core interface signals are shown in red.

This space intentionally left blank.

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 first half of the clock cycle, and are updated on the rising edge of the clock.

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 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 29, 2015.

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.

Memory initialization on the synthesis version is done from files ROM.mif and RAM.mif and it is done by the synthesis tool.

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. There is a signal from the processor that defines 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.

3 Coprocessor 0

Figure 4 shows a block diagram of the first three stages of the control pipeline, *viz* exception instruction fetch (EXCP_IF), exception decode (EXCP_RF), and exception execution (EXCP_EX).

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_100, EXPC_180 or EXPC_200, or the non-mask-able interrupt/reset address, EXPC_000.

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

Exception Execution – EXCP_EX   The control registers are updated at this stage. If an exception or interrupt is taken, the pipeline is stalled for two cycles to clear all control and instruction hazards, and the instructions at IF and RF are nullified. If the exception is an *overflow*, then the instruction at EX is also nullified.

Virtual addresses are computed on a separate adder and presented to the TLB during EX so that an exception can be signaled at this stage, rather than at MM.

3.1 COP0 resources

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

The instructions break, syscall, trap, mfc0, mtc0, eret, ei and di are implemented. These provide enough functionality for executing a full-blown Unix-like operating system on the processor model. The wait instruction aborts 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 not updated until it is 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.

See [MIPS05c] for examples of assembly code that references the COP0 registers. Simpler code fragments can be found in tests/*.s, these files have fairly obvious names.
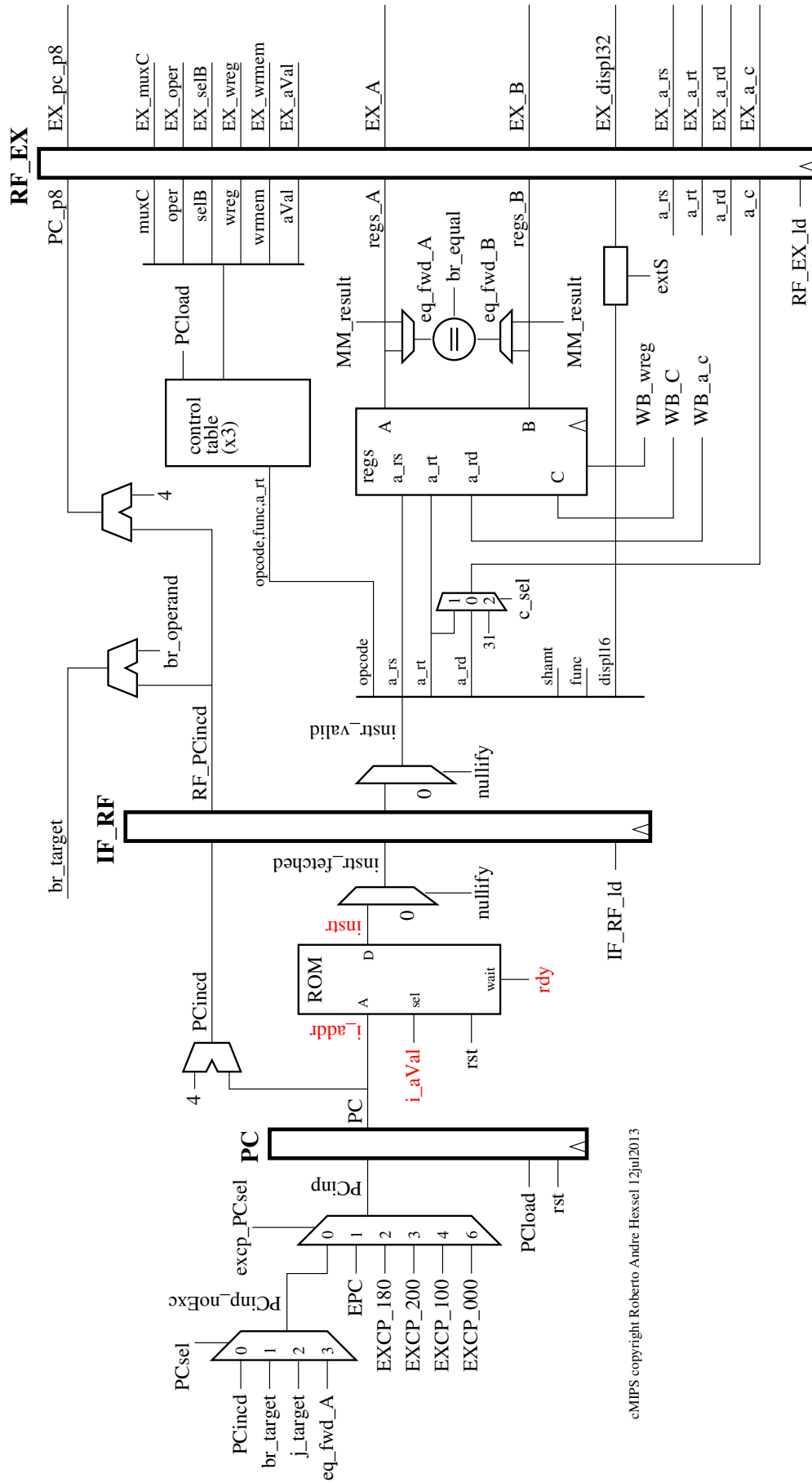
Figure 2: Block diagram of the front end: instruction fetch and decode.
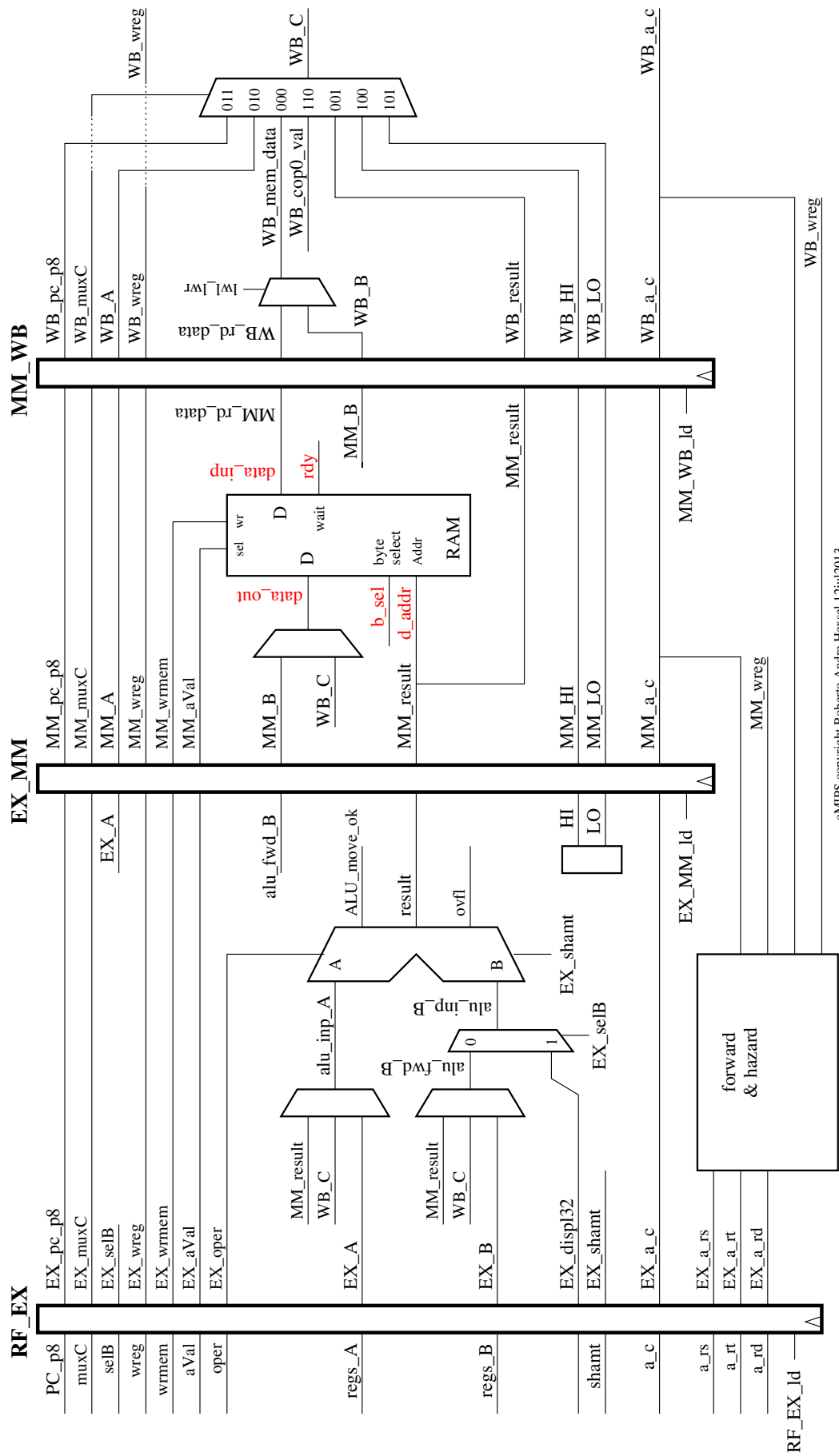
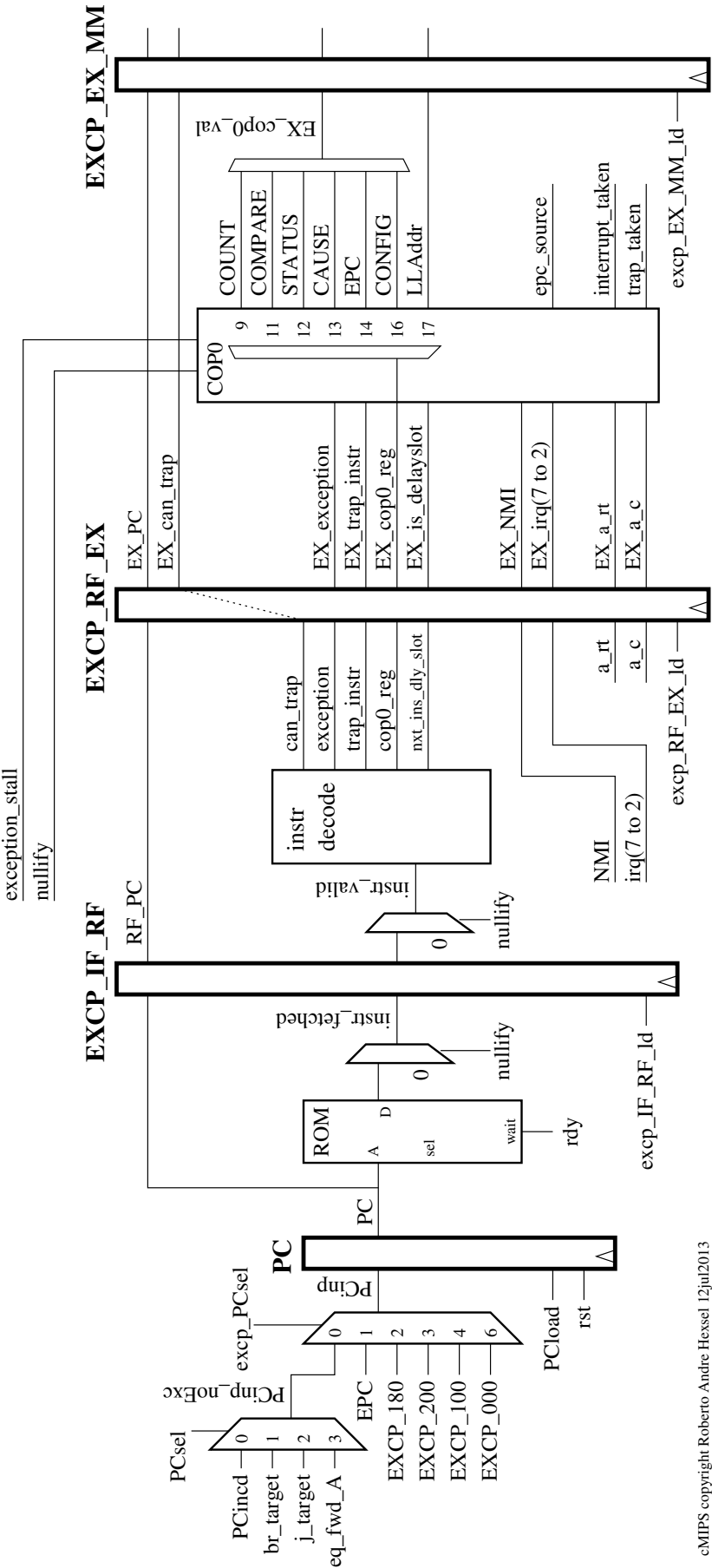Figure 3: Block diagram of the back end: execution, memory and write-back.

Figure 4: Block diagram of the control pipeline (COP0).

3.2 Memory Management Unit and the TLB

The simulation model includes an 8 entry fully associative TLB that supports 4 Kbytes 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 5 shows a block diagram of the signals that comprise the virtual to physical translations.
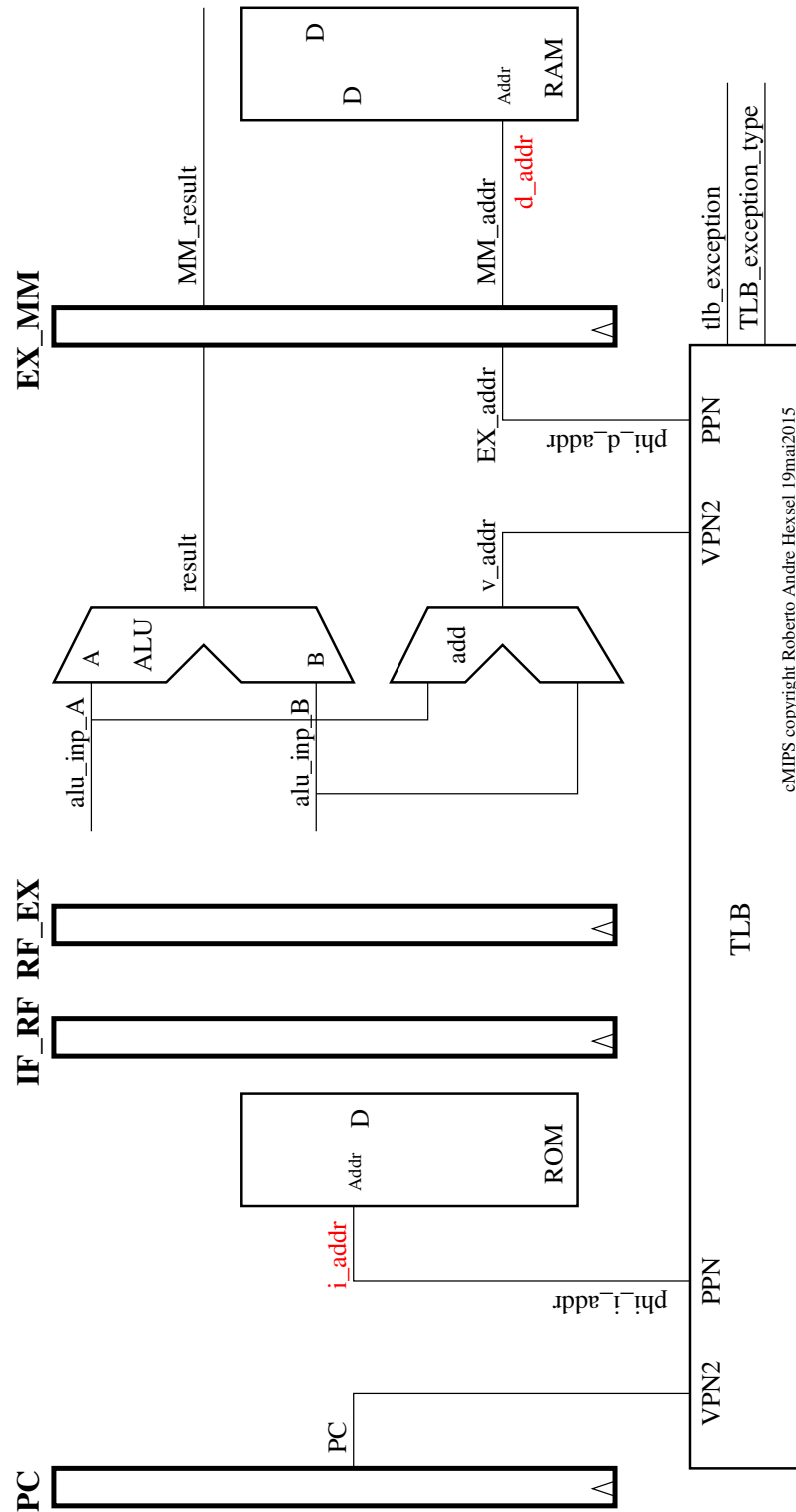
Figure 5: TLB–pipeline connections.

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.

Table 1: Initial page mappings on the TLB.

Entry	type	contents	addresses
0	ROM	ROM pages 0,1	0x0000.0000–0x0000.1fff
1	I/O	I/O pages 0,1	0x0f00.0000–0x0f00.1fff
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 register Context has 14 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.

The initialization code at include/start.s sets the top of the stack at address (x_DATA_BASE_ADDR + x_DATA_MEM_SZ − 8). The constants x_DATA_BASE_ADDR and x_DATA_MEM_SZ are defined in file vhdl/packageMemory.vhd.

4   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.

Listing 1: Entity for the processor core.

```vhdl
entity core is
  port (
    rst      : in     std_logic; -- reset, act=0
    clk      : in     std_logic; -- pipeline clock
    phi2     : in     std_logic; -- 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;
```

Listing 2: Entity for the synchronous ROM.

```vhdl
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;
```

Listing 3: Entity for the synchronous RAM.

```vhdl
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;
```

The entities for the instruction and data caches are very similar and only the data cache is shown below. In the testbench there are two *fake caches*, one for instructions and one for data; these models only forward the signals from core to memory and can be used for simulations that do not (need to) make use of the caches.

Listing 4: Entity for the data cache.

```vhdl
entity D_CACHE is            -- cache models are not adapted for synthesis
  generic (DC_LATENCY : time);
  port (rst           : in    std_logic; -- reset, act=0
        clk4x         : in    std_logic; -- state machine's clock
        cpu_sel       : in    std_logic; -- CPU side address valid, act=0
        cpu_rdy       : out   std_logic; -- CPU side data not ready, act=0
        cpu_wr        : in    std_logic; -- CPU side write, act=0
        cpu_addr      : in    reg32;     -- CPU side address
        cpu_data_inp  : in    reg32;     -- CPU side data
        cpu_data_out  : out   reg32;
        cpu_xfer      : in    reg4;      -- CPU side byte/half/word select
        mem_sel       : out   std_logic; -- MEM side address valid, act=0
        mem_rdy       : in    std_logic; -- MEM side data not ready, act=0
        mem_wr        : out   std_logic; -- MEM side write, act=0
        mem_addr      : out   reg32;     -- MEM side address
        mem_data_inp  : in    reg32;
        mem_data_out  : out   reg32;     -- MEM side data
        mem_xfer      : out   reg4;      -- MEM side byte/half/word select
        ref_cnt       : out   integer;   -- reference counter
        rd_hit_cnt    : out   integer;   -- read hit counter
        wr_hit_cnt    : out   integer);  -- write hit counter
end entity D_CACHE;
```

5 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 an adder, 32-bit register, ring-counter to generate the four-phases clock, flip-flops.

core.vhd Processor core. The code follows "the book" [PH09] closely. 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 of the processor core.

cache.vhd Contains models for the instruction and data caches. There are two "fake caches" that just pass along all the signals, for simulations without caches. All design parameters are computed from definitions for capacity and block size, defined in packageMemory.vhd.

instrcache.vhd Contains an FPGA-friendly model for the instruction cache. All design parameters are computed from definitions for capacity and block size, defined in packageMemory.vhd. This model has a state machine that initializes all cache tags after system reset, and during this interval – one clock cycle per cache block – the reset signal to the processor must be kept asserted.

memory.vhd Contains simulation models for ROM and RAM memory.

ram.vhd Contains a synthesizable model for RAM memory.

r0m.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: a 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.

pipe-stages.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`, `remota.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.

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.

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.

6 Scripts

Several scripts were written to build the cMIPS model, cross-compile the test programs and run the simulations. Yes, one day these should all be merged into one big fat Makefile[2]. 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.

`build.sh` (executable) Usage: `bin/build.sh`
Compiles all VHDL sources and builds the simulator/model.

`assemble.sh` (executable) Usage: `bin/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: `bin/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.

---

[2]It is easier for me to update the scripts as I change things, than to shepherd a Makefile. Sorry. By the way, some Makefiles I've seen hide the error messages produced by GHDL – surely not a good thing.

> *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.*

`include/cMIPS.ld` Not really a script, but contains the definition of the memory map employed by `assemble.sh` and `compile.sh` to link the executables. The address definitions <u>must</u> 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: `bin/edMemory.sh [-v]`

This script changes 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 <u>must</u> 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: `bin/run.sh [-n] [-w]`

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.

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 8.

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

Performs all functional tests on the cMIPS simulator/model; the script varies the number of wait states (0 or 1), hence the memory access time, as it repeats the whole gamut of tests – see Section 8. Normally, the real caches, and not the *fake caches*, should be instantiated in the testbench for these tests to be of much interest.

6.1 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 provided 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 optimize away all those meaningless references to memory 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 and the stack pointer. 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 file `include/stop.s` does not contain code. It is the last file to be linked and declares a few variables, which are allocated in memory at addresses above all variables declared in your "main" file. In particular, variable _highmem marks the highest RAM address used by your code. This 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 _highmem.

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` .

7 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 -tx 4 output.data | cut -d' ' -f 2-5 | sed -e '$d'`.

There are a few examples that employ the functions to access the 'peripherals' from C code in file `tests/readFile.c`. These 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);

// reads a character from simulator's standard input (stdin)
extern int from_stdin(char *);

// 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 otw
extern int readInt(int*);
```

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

8 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.

The assembly files test some specific instructions or features of the processor, caches, or memory. The C files are simple benchmarks used to do more complete tests 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.

The script `testAllLatencies.sh` tests two combinations of ROM and RAM latencies (0 or 1 wait-states) – the script edits the file `../packageWires.vhd` and changes the number of wait-states, then invokes `doTests.sh`. It can take some minutes to test the combinations.

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 <u>must</u> 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 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 like 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=00000044 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 command:

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 shows the offending output.

How to check the memory hierarchy and the processor interlocks   To perform all tests on the model, with sensible combinations of ROM and RAM latencies, you shall order:

1. `cd tests`;
2. `./testAllLatencies.sh`. This script invokes `./doTests` for different RAM and ROM latencies. Of course, the testbench must instantiate the real instruction and data caches, and not the fake caches. As of May 29, 2015, caches are not functional.

9   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 6 shows the timing diagram for a reference *without* wait-states, and a reference with *one* wait-state.
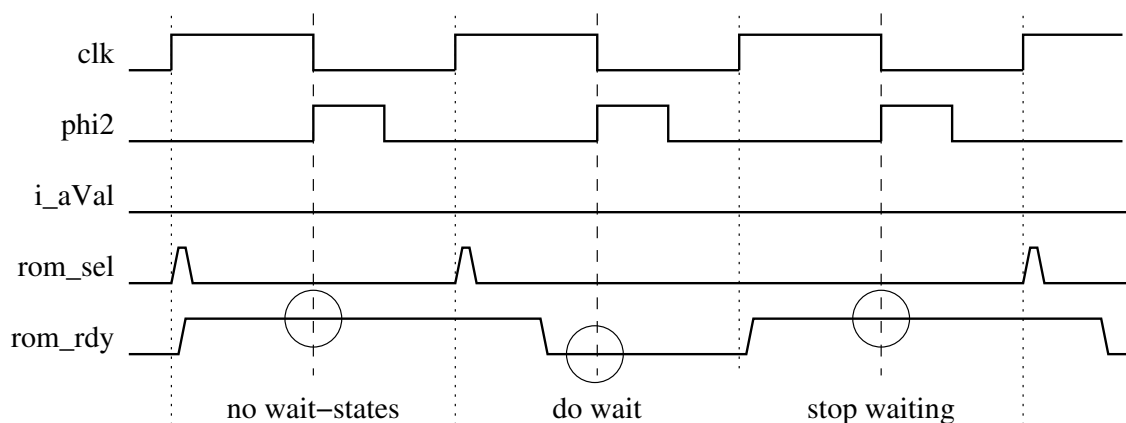

Figure 6: Timing diagram with wait-states.

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 finally deasserted, the reference completes on the next clock cycle.

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.

10   Ownership and Rights of Use

References

[PH09]   *Computer Organization & Design: The Hardware/Software Interface.* David A Patterson, John L Hennessy, 2009, 4th ed, Morgan Kaufmann, ISBN 9780123744937.

[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.