ION – MIPS Compatible Core

José A. Ruiz

December 2011

Contents

List of Figures

List of Tables

This file contains usage instructions and notes about the Ion CPU core project. The core structure is briefly explained in sections 1 to 5. The rest of this doc describes other aspects of the project: code samples, utility scripts, etc.

This document is not yet a full reference on the Ion core or a data sheet. Instead, it should be taken as a companion and commentary to the source code.

This document assumes you know in some depth the MIPS-I architecture. Terms and concepts from [1, 'See MIPS Run'] and [2, 'IDT R3K Reference Manual'] are used throughout the text.

Last modified: June 20, 2012

Section 1

Introduction

1.1 Features

This is a MIPS-I compatible CPU, aiming at compatibility with IDT's R3000 MIPS derivative:

Key features

1. Binary compatible to R3000 series of CPUs.
   Can use regular R3000 toolchains despite a few unimplemented opcodes listed below.
2. Kernel/user mode operation as per the architecture definition.
3. Exception handling compatible to MIPS-I standard.
4. 4KB direct-mapped code cache.
5. 4KB direct-mapped, writethrough data cache.
6. Simplified CP0, mostly compatible to R3000.
7. All unimplemented opcodes trigger the proper traps.
8. Includes minimalistic memory handler with interfaces for external SRAM (or FLASH) on 8- and 16-bit data bus.
9. Size and speed comparable to other free MIPS cores.
10. Fully sinchronous (rising clock edge only). No latches.
11. Source HDL is fully vendor independent (Only tested on Xilinx and Altera synthesis tools).

The implementation departs from the standard R3000 in some points:

Non-standard features

1. No MMU and no TLB – meaning no memory address translation at all.
2. Cache management simplified in an incompatible way.
3. A number of R3000 opcodes are unimplemented or implemented in an incompatible way:
   Unaligned load/store instructions (formerly patented).
   All CPA instructions.
   All CP0 instructions related to TLB and the cache.
4. Some other R3000 CP0 features have been omitted.
   This includes the CP0 timer.
5. Interrupt mask handling simplified in an incompatible way.

Finally, there are some flaws and missing features meant to be fixed in subsequent versions:

Missing features to be implemented eventually

1. Hardware interrupts not implemented.
2. Memory handler does not support dynamic RAM.
3. Caches are not configurable or parametrizable.
4. Real documentation (specs doc & data sheet) missing.

Section 2

Usage

2.1 Main Modules

The core is split in three main modules:

1. The CPU (mips_cpu.vhdl).
2. The cache+memory controller (mips_cache.vhdl).
3. A 'SoC' entity which combines CPU+Cache (mips_soc.vhdl).

The entity you should use in your projects is the SoC module. The project includes a 'hardware demo' built around this module (see section 9.1) which is meant as an usage example.

2.2 Bootstrap Code

Though the core is meant to run mostly from off-chip memory, the current version of the SoC module includes a small ROM implemented as FPGA BRAM and called 'bootstrap BRAM'. In the current version of the core, this BRAM can be loaded with arbitrary code and its size can be configured by using generics, but it can't be removed from the SoC. Even though the memory map can be modified to boot from external FLASH and not use a BRAM at all, a BRAM will still be inferred within the SoC – subsequent versions will fix this.

As can be seen in table 3.3, the internal BRAM is mirrored all over a wide area starting at `0xb8000000`. In practice, this means the BRAM will be mapped at the CPU reset address (`0xbfc00000`) and thus the bootstrap code should be placed there. Unless the bootstrap BRAM is very small, it will span over the interrupt vector address too (`0xbfc00180`).

For example, the 'Adventure' demo included with the project uses bootstrap code included in file `/src/common/bootstrap.s`. This bootstrap code is fairly incomplete (interrupt response code is mostly a stub) yet it's enough to boot most applications. Note that the C startup code, which deals with things like initializing the static variables on the data segment, etc. is not part of this bootstrap code. It can be found in file `/src/common/c_startup.s`

So, in short, the code loaded onto the startup BRAM should include the most basic system initialization (cache initialization at least) and the entry point for the interrupt response code; plus a jump to the main program entry address.

Anyone trying to build some application on this core is advised to use the code samples as starting points, specially the makefiles.

Loading Bootstrap Code on the SoC Module

Once the code that is to be loaded on the bootstrap BRAM has been built, you need to load it onto the bootstrap BRAM within the FPGA.

As you probably already know, there are several possible ways to deal with this and most of them involve using *'Memory Initialization Files'* of some sort. This project is different.

So far, this project does not include any support for using IMF files of any kind. Instead, the bootstrap BRAM is inferred and initialized using regular VHDL constructs and a constant passed to the SoC module as a generic.

This scheme has a big drawback: every time the object code within the FPGA changes, the whole synthesis needs to be re-run. This drawback is manageable as long as the core is not used in any big project or if the bootstrap code does not change often.

On the other hand, I see some big advantages in using regular BRAM inference in this stage of the project:

1. The whole scheme is totally vendor agnostic.
2. Object code embedded on VHDL constants can very easily be used in both simulation and synthesis.

So, whatever object code is to be used to initialize the SoC bootstrap BRAM has to be passed to the SoC module instance as a generic constant (see section 3.1). The constant must be of type `t_obj_code`, which is defined in package *mips_pkg*.

Building the Bootstrap Initialization Constant

The project includes a python script (`/tools/build_pkg/build_pkg.py`) whose purpose is to build an VHDL `t_obj_code` constant out of a *binary* object code file.

This script will read one or more big-endian, binary object files and will produce a VHDL package file that will contain initialization constants for the bootstrap BRAM and for some other memories that are only used in the simulation test bench. The package can optionally include too some simulation and synthesis configuration constants – such as the size of the bootstrap BRAM.

The makefiles included in the code samples invoke this script twice: once to generate a package called *sim_params_pkg* and used in the simulation test bench; and once to build a package called *bootstrap_code_pkg* used for synthesis.

Please refer to the makefiles for usage examples, and read the script source for more detailed usage instructions.

Section

3

SoC Module

The main purpose of the SoC module is to encapsulate the somewhat complex interconnection between the CPU and the Cache/Memory Controller module.

If some project demands that some piece of hardware be directly connected to the CPU, bypassing the cache, this is where it should be – an MMU comes to mind.

Any peripherals deemed common enough that they will be present in all projects might be placed in the SoC module too.

In the current version of the SoC module, there is only one peripheral included in it – a hard-wired UART module. There is no penalty for placing peripherals ouside the SoC module, so there is no incentive to place them inside. This is an implementation option of yours.

Bear in mind that, in its current state, the SoC module is little more than a vehicle for building demos around the ION CPU. It is not meant as a real-world SoC, though it might be deloped into one eventually.

3.1 SoC Generics

The SoC needs to be configured upon instantiation by setting the following generics:

Table 3.1: SoC module generics

Name	Type	Default value	Description
`BOOT_BRAM_SIZE`	integer	1024	Bootstrap BRAM size in 32-bit words.
`OBJ_CODE`	t_obj_code	(void code)	Bootstrap BRAM contents.
`CLOCK_FREQ`	integer	50e6	Main clock rate.
`BAUD_RATE`	integer	19200	UART baud rate.
`SRAM_ADDR_SIZE`	integer	17	Size of SRAM address bus.

The current version of the SoC is not very strict in the enforcement of limits for the generics. You are advised to use only 'reasonable' values. This will be fixed, eventually.

Generic `CLOCK_FREQ` is only needed in order to compute the default baud period for the internal UART (from the value of generic `BAUD_RATE`).

Generic `BOOT_BRAM_SIZE` will determine the size of the internal bootstrap BRAM. This generic *can't be zero*; in the current version of the SoC, the BRAM can't be disabled or omitted.

Note that if the size of the bootstrap BRAM is not enough to hold the whole bootstrap code provided in generic `OBJ_CODE`, the code *will be sineltly truncated!*. Usually this will result in an early crash.

Generic `OBJ_CODE` is used at synthesis time to initialize the bootstrap BRAM. This generic is meant to contain boostrap code, as seen in section 2.2). It can be omitted, in which case the bootstrap BRAM will be initialized to all zeros.

3.2 SoC Ports

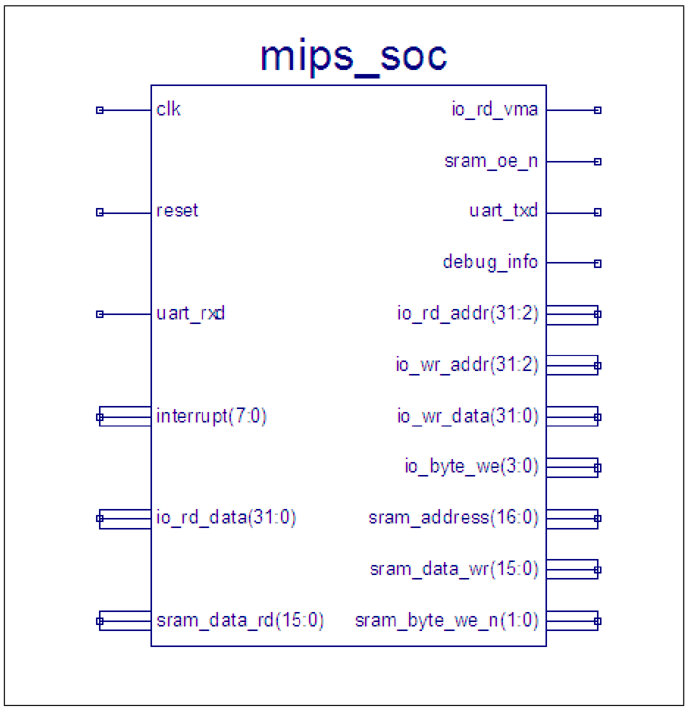

Figure 3.1: SoC module interface

As you can see in figure 3.1 (symbol generated by Xilinx ISE), the SoC has the following interfaces:

1. Interface to external static asynchronous memory (SRAM, FLASH...).

Table 3.2: SoC module interface ports

Name	Type	Width	Description
clk	in	1	Clock input, active rising edge.
reset	in	1	Synchronous global reset.
sram_address	out	16	Memory word address (bit 0 absent).
sram_data_wr	out	16	Memory write data. Only valid when one of the memory byte write enable outputs is active.
sram_data_rd	in	16	Memory read data. Latched when xxx.
sram_byte_we_n	out	2	Memory byte write enable, active low. (0) enables the low byte (7 downto 0) (1) enables the high byte (15 downto 8).
io_rd_addr	out	30	I/O port read address (bits 1..0 absent). Only valid when io_rd_vma is high.
io_wr_addr	out	30	I/O port write address (bits 1..0 absent).
io_wr_data	out	32	I/O write data. Only valid when one of the i/o byte write enable outputs is active.
io_rd_data	in	32	I/O read data. Latched when xxx.
io_byte_we	out	4	I/O byte write enable, active high. (0) enables the low byte (7 downto 0) (3) enables the high byte (31 downto 24).
io_rd_vma	out	1	Active high on i/o read cycles.
uart_rxd	in	1	RxD input to internal UART.
uart_txd	out	1	TxD output from internal UART.
interrupt	in	8	Interrupt request inputs, active high.

2. Interface to on-chip peripherals.
3. Interrupt inputs.
4. Debug port.

These interfaces will be explained in the following subsections. The top module for the demo supplied with the project (c2sb_demo.vhdl) will be used for illustration.

NOTE: This section needs a lot of elaboration – ideally this should be equivalent to a datasheet in thoroughness and detail. This work, like many other parts of this project, will have to wait.

SoC interface to static memory

The interface to external memory in the SoC module is essentially that of the internal cache/memory controller. Its timing is described in section 5.4.

The SoC inputs are meant to be connected straight to the FPGA i/o pins. The only trick is the bidirectional memory data bus: as you can see, the SoC data buses are unidirectional and thus you will need to provide an interconnection external to this module. This interconnection shall include the requisite 3-state buffers:

```
sram_databus <= sram_data_wr when sram_byte_we_n/="11" else (others => 'Z');
```

The top level *c2sb_demo* module can be used as a fully tested example of how to use this interface to connect to a common 16-bit-wide SRAM chip (ISSI IS61LV25616).

In reviewing the top module source, note that I had to adapt the dual byte-write-enable outputs to the SRAM configuration of a single write-enable plus dual byte-enable inputs.

Note too that the static memory bus of the SoC module is used to access both the 16-bit wide SRAM and an 8-bit wide FLASH. These chips are connected to separate buses on the target board, so the top c2sb_demo module needs to conflate both buses before connecting them to the SoC. This is why a multiplexor is used in the `mpu_sram_data_rd` bus. A real-world board would probably have the SRAM and the FLASH connected to the same bus, simplifying the interface logic.

SoC interface to peripherals

Every CPU access to an area designated as I/O (see 3.3, memory map) will trigger a read/write cycle on this interface.

I/O ports are synchronous, byte accesible registers meant to be implemented within the FPGA. I/O ports do not support wait states.

The I/O interface has separate input and output buses.

In an output cycle, one or more lines of signal `io_byte_we` will be asserted for one clock cycle. Signals `io_wr_addr` and `io_wr_addr` will be valid as long as `io_byte_we` is asserted.

In an input cycle, `io_rd_vma` will be asserted for one cycle and the input data should be present at `io_rd_data` at the end of the following clock cycle. The full read operation extends over two clock cycles.

SoC interrupt inputs

The present version of the CPU does not have support for hardware interrupts and therefore these signals are not used yet and are unconnected. Hardware interrupts will be implemented in some future version as time permits.

SoC debug port

The debug port is a VHDL record (`t_debug_info`, defined in package *mips_pkg*), which holds some internal CPU status flags that can be useful while debugging the core. It is not meant to be useful for a real application.

Currently the record holds only two flags:

- `cache_enabled`, asserted when the cache is enabled.
- `unmapped_access`, asserted when some access to an unmapped address is made.

The current version of the demo connects these signals to some on-board LEDs.

3.3 SoC Memory Map

The *memory map* determines the type of memory that is connected to each of a number of predefined address rangess (see section 5.3). It is defined in package *mips_pkg* and it is implemented in the *mips_cache* module.

Table 3.3: SoC module memory map

Address range	Type	Wait States	Intended usage
`0xb8000000-0xbfffffff`	BRAM	0	SoC internal boot BRAM.
`0x00000000-0x07ffffff`	SRAM-16	2	Off-chip SRAM.
`0x80000000-0x87ffffff`	SRAM-16	2	Off-chip SRAM.
`0x20000000-0x27ffffff`	I/O	0	On-chip I/O registers.
`0xb0000000-0xb7ffffff`	SRAM-8	7	Off-chip SRAM or FLASH.

3.4 SoC UART

The current revision of the SoC includes a single peripheral, a hardwired 8-bit UART (file *uart.vhdl*).

This UART is an 8-bit module built for some other unrelated project of mine and commandeered to serve on this SoC. Therefore, it has some features (like its 8-bit interface) which are sub-optimal for this application and/or are not used.

The UART is 'hardwired' because some of its operational parameters are hardcoded and can't be changed even at synthesis time. Namely:

- Stop bits: 1.

- Parity: None.
- Bits per character: 8.

All other parameters can at least be configured at synthesis time, and under some conditions can be configured at run time too. The interested user must read the module source for a better explaination of these features. This document will only deal with the UART module as it is instantiated in the SoC.

These are the UART control registers:

Table 3.4: UART control registers

Byte Address	Word Address	Register
`0x20000003`	`0x20000000`	Tx/Rx Buffer
`0x20000007`	`0x20000004`	Status.
`0x2000000b`	`0x20000008`	Baud rate period, low byte.
`0x2000000f`	`0x2000000c`	Baud rate period, high byte.

All of these registers are mapped to the byte address given in the table, that is, they are mapped on the *low* byte of the 32-bit word they belong to – you don't have to worry about this unless you use a word pointer to access these registers.

UART Usage

Until hardware interrupts are implemented, you have to rely on polling to use the UART.

When you want to transmit, you wait until flag TxRdy is '1' and then write to the Tx Buffer. That will clear TxRdy until the transmission is done. Writing to the Tx Buffer will NOT clear flag TxIrq.

Writing to the Tx Buffer while TxRdy is '0' will have no effect.

When you want to read received data, you wait until RxRdy is '1' and then read the Rx Buffer. Reading the Rx Buffer will clear flag RxRdy until a new byte arrives. Reading the RxBuffer while RxRdy is '0' will return undefined data. Reading the Rx Buffer will NOT clear flag RxIrq.

Of course, once hardware interrupts are implemented, you will use them instead of polling, but this is the basic mechanics. Same as any old UART, really.

UART Status Register

These are the flags present in the status register:

```
UART Status Register

    7       6       5       4       3       2       1       0
+-------+-------+-------+-------+-------+-------+-------+-------+
|   0   |   0   | RxIrq | TxIrq |   0   |   0   | RxRdy | TxRdy |
+-------+-------+-------+-------+-------+-------+-------+-------+
    h       h      W1C     W1C      h       h       r       r
```

Bits marked ’h’ are hardwired and can’t be modified.

Bits marked ’r’ are read only; they are set and clear by the UART core.

Bits marked W1C (’Write 1 Clear’) are set by the UART core when an interrupt has been triggered and must be cleared by the software by writing a ’1’.

- Status bit TxRdy is high when there isn’t any transmission in progress. It is cleared when data is written to the transmission buffer and is raised at the same time the transmission interrupt is triggered.
- Status bit RxRdy is raised at the same time the receive interrupt is triggered and is cleared when the data register is read.
- Status bit TxIrq is raised when the transmission interrupt is triggered and is cleared when a 1 is written to it.
- Status bit RxIrq is raised when the reception interrupt is triggered and is cleared when a 1 is written to it.

When writing to the status/control registers, only flags TxIrq and RxIrq are affected, and only when writing a ’1’ as explained above. All other flags are read-only.

UART Baud Rate Registers

When the UART module generic ’HARDWIRED’ is set to ’false’, these registers can be written to in order to configure the baud rate – see the source for details.

When the UART module generic ’HARDWIRED’ is set to ’true’, these registers are frozen and can’t be modified. This is how the module is instantiated in the current version of the SoC.

In either case, these are write-only registers: reading them will return the contents of the status register (simplified multiplexor).

The baud rate is configured by loading these registers with the baud period in clock cycles.

UART Interrupt

The UART can trigger an interrupt (i.e. assert its interrupt output for one clock cycle) whenever a character is received or transmitted. The UART source explains in detail exactly when these interrupts are triggered.

The interrupt status is kept in two flags on the status register that can be used for interrupt polling. Note there's no way to tell what kind of interrupt we got other than looking at those flags.

Since the current CPU revision does not support hardware interrupts, this feature is still unused and the interrupt line is unconnected. Again, details can be found in the UART module source.

Section
4

CPU Description

This section is about the 'core' cpu (mips_cpu.vhdl), excluding the cache.

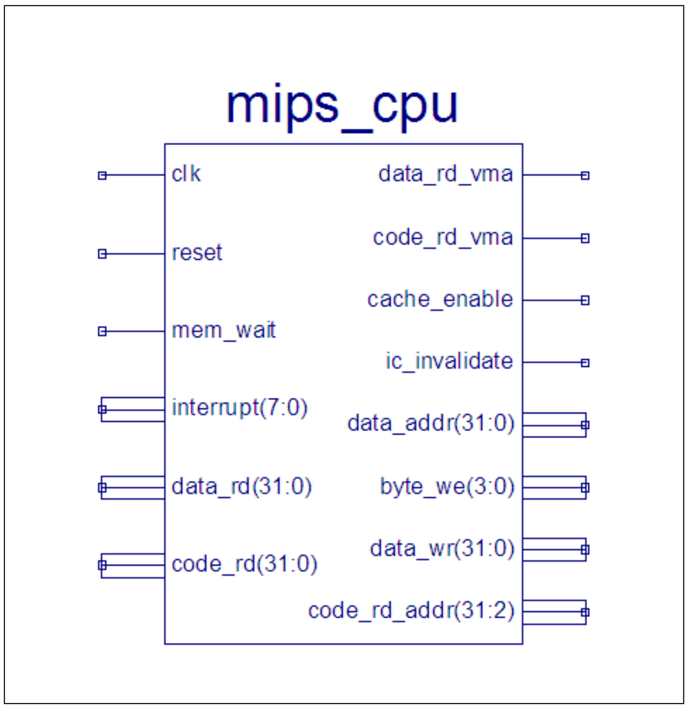

Figure 4.1: CPU module interface

the CPU module is not meant to be used directly. Instead, the SoC module described in chapter 3 should be used.

The following sections will describe the CPU structure and interfaces.

4.1 Bus Architecture

The CPU uses a Harvard architecture: separate paths for code and data. It has three separate, independent buses: code read, data read and data write. (Data read and write share the same address bus).

The CPU will perform opcode fetches at the same time it does data reads or writes. This is not only faster than a Von Neumann architecture where all memory cycles have to be bottlenecked through a single bus; it is much simpler too. And most importantly, it matches the way the CPU will work when connected to code and data caches.

(Actually, in the current state of the core, due to the inefficient way in which load interlocking has been implemented, the core is less efficient than that – more on this later).

The core can't read and write data at the same time; this is a fundamental limitation of the core structure: doing both at the same time would take one more read port in the register bank – too expensive. Besides, it's not necessary given the 3-stage pipeline structure we use.

In the most basic use scenario (no external memory and no caches), code and data buses have a common address space but different storages and the architecture is strictly Harvard. When cache memory is connected the architecture becomes a Modified Harvard – because data and code ultimately come from the same storage, on the same external bus(es).

Note that the basic cpu module (mips_cpu) is meant to be connected to internal, synchronous BRAMs only (i.e. the cache BRAMs). Some of its outputs are not registered because they needn't be. The parent module (called 'mips_soc') has its outputs registered to limit t_{co} to acceptable values.

Code and data read bus interface

Both buses have the same interface:

⋆_rd_addr	Address bus
⋆_rd	Data/opcode bus
⋆_rd_vma	Valid Memory Address (VMA)

The CPU assumes SYNCHRONOUS external memory (most or all FPGA architectures have only synchronous RAM blocks):

When ⋆_rd_vma is active ('1'), ⋆_rd_addr is a valid read address and the memory should provide read data at the next clock cycle.

The following ascii-art waveforms depict simple data and code read cycles where there is no interlock – interlock is discussed in section 2.3.

```
==== Chronogram 3.1.A: data read cycle, no stall =====================
                     ----      ----      ----      ----      ----
 clk            ____/    \____/    \____/    \____/    \____/    \____/

                     ---------           ---------
 data_rd_vma    ____/         \_________/         \____________________

 data_rd_addr   XXXX| 0x0700  |XXXXXXXXX| 0x0800  |XXXXXXXXX|XXXXXXXXX|

 data_rd        XXXX|XXXXXXXXX| [0x700] |XXXXXXXXX| [0x800] |XXXXXXXXX|

 (target reg)   ????|?????????|?????????| [0x700] |?????????| [0x800] |

          (data is registered here...)--^    (...and here)--^

 ==== Chronogram 3.1.B: code read cycle, no stall ====================
                     ----      ----      ----      ----      ----
 clk            ____/    \____/    \____/    \____/    \____/    \____/

                     --------------------------------------------------
code_rd_vma     ____/         |         |         |         |         |

                ????| 0x0100  | 0x0104  | 0x0108  | 0x010c  | 0x0200  |

code_rd         XXXX|XXXXXXXXX| [0x100] | [0x104] | [0x108] | [0x10c] |

p1_ir_reg       ????|?????????|?????????| [0x100] | [0x104] | [0x108] |

  (first code word is registered here)--^

========================================================================
```

The data address bus is 32-bit wide; the lowest 2 bits are redundant in read cycles since the CPU always reads full words, but they may be useful for debugging.

Data write interface

The write bus does not have a vma output because byte_we fulfills the same role:

⋆_wr_addr	Address bus
⋆_wr	Data/opcode bus
byte_we	WE for each of the four bytes

Write cycles are synchronous too. The four bytes in the word should be handled separately – the CPU will assert a combination of byte_we bits according to the size and alignment of the store.

When byte_we(i) is active, the matching byte at data_wr should be stored at address data_wr_addr. byte_we(0) is for the LSB, byte_we(3) for the MSB. Note that since the CPU is big endian, the MSB has the lowest address and LSB the highest. The memory system does not need to care about that.

Write cycles never cause data-hazard stalls. They would take a single clock cycle except for the inefficient cache implementation, which stalls the CPU until the writethrough is done.

This is the waveform for a basic write cycle (remember, this is without cache; the cache would just stretch this diagram by a few cycles):

```
==== Chronogram 3.2: data write cycle =================================
                     ----      ----      ----      ----      ----
 clk            ____/    \____/    \____/    \____/    \____/    \____/

 byte_we        XXXX|  1111    |  0000    |  0100    |  1100    |  0000    |

 data_wr_addr   XXXX| 0x0700   |XXXXXXXXX| 0x0800   | 0x0900   |XXXXXXXXX|

 data_wr        XXXX|12345678h|XXXXXXXXX|12345678h|12345678h|XXXXXXXXX|

 [0x0700]       ????|????????h|12345678h|12345678h|12345678h|12345678h|

 [0x0800]       ????|????????h|????????h|????????h|??34????h|??34????h|

 [0x0900]       ????|????????h|????????h|????????h|????????h|1234????h|

==========================================================================
```

Note the two back-to-back stores to addresses 0x0800 and 0x0900. They are produced by two consecutive S⋆ instructions (SB and SH in the example), and can only be done this fast because of the Harvard architecture (and, again, because the diagram does not account for the cache interaction).

CPU stalls caused by memory accesses

In short, the 'mem_wait' input will unconditionally stall all pipeline stages as long as it is active. It is meant to be used by the cache at cache refills.

The cache/memory controller stops the cpu for all data/code misses for as long as it takes to do a line refill. The current cache implementation does refills in reverse order (i.e. not 'target address first').

Note that external memory wait states are a separate issue. They too are handled in the cache/memory controller. See section 5 on the memory controller.

4.2 Pipeline

Here is where I would explain the structure of the cpu in detail; these brief comments will have to suffice until I write some real documentation.

This section could really use a diagram; since it can take me days to draw one, that will have to wait for a further revision.

This core has a 3-stage pipeline quite different from the original architecture spec. Instead of trying to use the original names for the stages, I'll define my own.

A computational instruction of the I- or R- type goes through the following stages during execution:

FETCH-0	Instruction address is in code_rd_addr bus
FETCH-1	Instruction opcode is in code_rd bus
ALU/MEM	ALU operation or memory read/write cycle is done OR Memory read/data address is on data_rd/wr_address bus OR Memory write data is on data_wr bus
LOAD	Memory read data is on data_rd bus

In the core source (mips_cpu.vhdl) the stages have been numbered:

FETCH-1	= stage 0
ALU/MEM	= stage 1
LOAD	= stage 2

Here's a few examples:

```
==== Chronogram 3.3.A: stages for instruction "lui gp,0x1" ============
                     ----      ----      ----      ----      ----
 clk            ____/    \____/    \____/    \____/    \____/    \____/

 code_rd_addr       | 0x099c  |

 code_rd_data                 |3c1c0001h|

 rbank[$gp]                             | 0x0001  |

                    |< fetch0>|<   0   >|<   1   >|
=======================================================================
==== Chronogram 3.3.B: stages for instruction "lw a0,16(v0)" ==========
                     ----      ----      ----      ----      ----
 clk            ____/    \____/    \____/    \____/    \____/    \____/

 code_rd_addr       | 0x099c  |

 code_rd_data                 |8c420010h|

 data_rd_addr                           | $v0+16  |
                                         ---------
 data_rd_vma                       _____/         \______

 data_rd                                          | <data>  |

 rbank[$a0]                                                 | <data>  |

                    |< fetch1>|<   0   >|<   1   >|<   2   >|
========================================================================
```

In the source code, all registers and signals in stage <i>are prefixed by "p<i>_", as in p0_*, p1_* and p2_*. A stage includes a set of registers and all the logic that feeds from those registers (actually, all the logic that is between registers p0_* and p1_* belongs in stage 0, and so on). Since there are signals that feed from more than one pipeline stage (for example p2_wback_mux_sel, which controls the register bank write port data multiplexor and feeds from p1 and p2), the naming convention has to be a little flexible.

FETCH-0 would only include the logic between p0_pc_reg and the code ram address port, so it has been omitted from the naming convention.

All read and write ports of the register bank are synchronous. The read ports belong logically to stage 1 and the write port to stage 2.

IMPORTANT: though the register bank read port is synchronous, its data can be used in stage 1 because it is read early (the read address port is loaded at the same time as the instruction opcode). That is, a small part of the instruction decoding is done on stage FETCH-1, by feeding the source register index field straight from the code bus to the register bank BRAM.

Bearing in mind that the code cache ram is meant to be the exact same type of block as the register bank (or faster if the register bank is implemented with distributed RAM), and we will cram the whole ALU delay plus the reg bank delay in stage 1, it does not hurt moving a tiny part of the decoding to the previous cycle.

All registers but a few exceptions belong squarely to one of the pipeline stages:

Stage 0:	
p0_pc_reg	PC
(external code ram read port register)	Loads the same as PC
Stage 1:	
p1_ir_reg	Instruction register
(register bank read port register)	
p1_rbank_forward	Feed-forward data (hazards)
p1_rbank_rs_hazard	Rs hazard detected
p1_rbank_rt_hazard	Rt hazard detected
Stage 2:	
p2_exception	Exception control
p2_do_load	Load from data_rd
p2_ld_*	Load control
(register bank write port register)	

Note how the register bank ports belong in different stages even if it's the same physical device. No conflict here, hazards are handled properly (logic defined with explicit vendor-independent vhdl code, not with synthesis pragmas, etc.).

There is a small number of global registers that don't belong to any pipeline stage:

pipeline_stalled	Together, these two signals...
pipeline_interlocked	...control pipeline stalls

And of course there are special registers accessible to the CPU programmer model:

mdiv_hi_reg	register HI from multiplier block
mdiv_lo_reg	register LO from multiplier block
cp0_status	register CP0[status]
cp0_epc	register CP0[epc]
cp0_cause	register CP0[cause]

These belong logically to pipeline stage 1 (can be considered an extension of the register bank) but have been spared the prefix for clarity.

Note that the CP0 status and cause registers are only partially implemented.

Again, this needs a better explaination and a diagram.

4.3 Interlocking and Data Hazards

There are two data hazards we need to care about:

a) If an instruction needs to access a register which was modified by the previous instruction, we have a data hazard – because the register bank is synchronous, a memory location can't be read in the same cycle it is updated – we will get the pre-update value.

b) A memory load into a register Rd produces its result a cycle late, so if the instruction after the load needs to access Rd there is a conflict.

Conflict (a) is solved with some data forwarding logic: if we detect the data hazard, the register bank uses a 'feed-forward' value instead of the value read from the memory file.
In file mips_cpu.vhdl, see process 'data_forward_register' and the following few lines, where the hazard detection logic and data register and multiplexors are implemented. Note that hazard is detected separately for both read ports of the reg bank (p0_rbank_rs_hazard and p0_rbank_rt_hazard).
Note too that this logic is strictly regular vhdl code – no need to rely here on the synthesis tool to add the bypass logic for us. This gets us some measure of vendor independence.

As for conflict (b), in the original MIPS-I architecture it was the job of the programmer to make sure that a loaded value was not used before it was available – by inserting NOPs after the load instruction, if necessary. This is what I call the 'load delay slot', as discussed in [2, p. 13-1].

The C toolchain needs to be set up for MIPS-I compliance in order to build object code compatible with this scheme.
But all succeeding versions of the MIPS architecture implement a different scheme instead, 'load interlock' ([1, p. 28]). You often see this behavior in code generated by gcc, even when using the -mips1 flag (this is probably due to the precompiled support code or libc, I have to check).
In short, it pays to implement load interlocks so this core does, but the feature should be optional through a generic.

Load interlock is triggered in stage 1 (ALU/MEM) of the load instruction; when triggered, the pipeline stages 0 and 1 stall, but the pipeline stage 2 is allowed to proceed. That is, PC and IR are frozen but the value loaded from memory is written in the register bank.

In the current implementation, the instruction following the load is UNCONDITIONALLY

stalled; even if it does not use the target register of the load. This prevents, for example, interleaving read and write memory cycles back to back, which the CPU otherwise could do.
So the interlock should only be triggered when necessary; this has to be fixed.

```
==== Chronogram 3.4.A: data read cycle, showing interlock =============
                     ----      ----      ----      ----      ----
 clk            ____/    \____/    \____/    \____/    \____/    \____/

 code_rd_addr   XXXX| 0x099c   | 0x09a0            | 0x09a4   | 0x09a8   |

 byte_we        XXXX|  0000   |  1111   |  0000   |  0000   |  1100   |

                                        |<                 >|

                ------------------------           --------------------
 code_rd_vma                            \_________/
                                         ---------
 data_rd_vma    ________________________/         \____________________

 data_rd_addr   XXXX|XXXXXXXXX|XXXXXXXXX| 0x0700            |XXXXXXXXX|

 data_rd        XXXX|XXXXXXXXX|XXXXXXXXX|XXXXXXXXX| [0x700]  |XXXXXXXXX|

 (target reg)   ????|?????????|?????????|?????????| [0x700]  |?????????|

                       (data is registered here)--^

========================================================================
```

Note how a fetch cycle is delayed.

This waveform was produced by this code:

```
...
998:    ac430010    sw  v1,16(v0)
99c:    80440010    lb  a0,16(v0)
9a0:    a2840000    sb  a0,0(s4)
9a4:    80440011    lb  a0,17(v0)
9a8:    00000000    nop
...
```

Note how read and write cycles are spaced instead of being interleaved, as they would if interlocking was implemented efficiently (in this example, there was a real hazard, register $a0, but that's coincidence – I need to find a better example in the listing files...).

4.4 Exceptions

The only exceptions supported so far are software exceptions, and of those only the instructions BREAK and SYSCALL, the 'unimplemented opcode' trap and the 'user-mode access to CP0' trap. Memory privileges are not and will not be implemented. Hardware/software interrupts are still unimplemented too.

Exceptions are meant to work as in the R3000 CPUs except for the vector address.
They save their own address to EPC, update the SR, abort the following instruction, and jump to the exception vector 0x0180. All as per the specs except the vector address (we only use one).

The following instruction is aborted even if it is a load or a jump, and traps work as specified even from a delay slot – in that case, the address saved to EPF is not the victim instruction's but the preceding jump instruction's as explained in [1, p. 64].

Plasma CPU used to save in epc the address of the instruction after break or syscall, and used an unstandard vector address (0x03c). This core will go the standard R3000 way instead.

Note that the epc register is not used by any instruction other than mfc0. RTE is implemented and works as per R3000 specs.

4.5 Multiplier

The core includes a multiplier/divider module which is a slightly modified version of Plasma's multiplier unit. Changes have been commented in the source code.

The main difference is that Plasma does not stall the pipeline while a multiplication/division is going on. It only does when you attempt to get registers HI or LO while the multiplier is still running. Only then will the pipeline stall until the operation completes.
This core instead stalls always for all the time it takes to do the operation. Not only it is simpler this way, it will also be easier to abort mult/div instructions.

The logic dealing with mul/div stalls is a bit convoluted and coud use some explaining and some ascii chronogram. Again, TBD.

Section

5

Cache/Memory Controller Module

The project includes a cache+memory controller module from revision 114.

Both the I- and the D-Cache are implemented. But the parametrization generics are still mostly unused, with many values hardcoded. And SDRAM is not supported yet. Besides, there are some loose ends in the implementation still to be solved, exlained in section 5.5.

5.1 Cache Initialization and Control

The cache module comes up from reset in an indeterminate, unuseable state. It needs to be initialized before being enabled.
Initialization means mostly marking all D- and I-cache lines as invalid. The old R3000 had its own means to achieve this, but this core implements an alternative, simplified scheme.

The standard R3000 cache control flags in the SR are not used, either. Instead, two flags from the SR have been repurposed for cache control.

Cache control flags

Bits 17 and 16 of the SR are NOT used for their standard R3000 purpose. Instead they are used as explained below:

- Bit 17: Cache enable [reset value = 0]
- Bit 16: I- and D-Cache line invalidate [reset value = 0]

You always use both these flags together to set the cache operating mode:

- Bits 17:16='00'
  Cache is enabled and working.
- Bits 17:16='01'
  Cache is in D- and I-cache line invalidation mode.
  Writing word X.X.X.N to ANY address will invalidate I-Cache line N (N is an 8-bit word and X is an 8-bit don't care). Besides, the actual write will be performed too; be careful...

  Reading from any address will cause the corresponding D-cache line to be invalidated; the read will not be actually performed and the read value is undefined.
- Bits 17:16='10'
  Cache is disabled; no lines are being invalidated.
- Bits 17:16='11'
  Cache behavior is UNDETERMINED – i.e. guaranteed crash.

Now, after reset the cache memory comes up in an undetermined state but it comes up disabled too. Before enabling it, you need to invalidate all cache lines in software (see routine cache_init in the memtest sample).

5.2 Cache Tags and Cache Address Mirroring

In order to save space in the I-Cache tag table, the tags are shorter than they should – 14 bits instead of the 20 bits we would need to cover the entire 32-bit address:

```
            _________ <-- These address bits are NOT in the tag
           /         \
31 ..   27| 26 .. 21  |20 ..          12|11  ..        4|3:2|
+---------+-----------+-----------------+---------------+---+---+
| 5       |           | 9               | 8             | 2 |   |
+---------+-----------+-----------------+---------------+---+---+
^                     ^                 ^               ^- LINE_INDEX_SIZE
5 bits                9 bits            LINE_NUMBER_SIZE
```

Since bits 26 downto 21 are not included in the tag, there will be a 'mirror' effect in the cache. We have effectively split the memory space into 32 separate blocks of 1MB which is obviously not enough but will do for the initial versions of the core.

In subsequent versions of the cache, the tag size needs to be enlarged AND some of the top bits might be omitted when they're not needed to implement the default MIPS memory map (namely bit 30 which is always '0').

5.3 Memory Controller

The cache functionality and the memory controller functionality are so closely related that I found it convenient to bundle both in the same module: I have experimented with separate modules and was unable to come up with the same performance with my synthesis tools of choice. So, while it would be desirable to split the cache from the memory controller at some later version, for the time being they are a single module.

The memory controller interfaces the cache to external memory, be it off-core or off-chip. The memory controller implements the refill and writethrough state machines, that will necessarily be different for different kinds of memory.

Memory Map Definition

The MIPS architecture specs define a memory map which determines which areas are cached and which is the default address translation [2, p. 2-8].
Neither the memory translation nor the cacheable/uncacheable attribute of the standard MIPS architecture have been implemented. In this core, program addresses are always identical to hardware addresses.

When requested to perform a refill or a writethrough, the memory controller needs to know what type of memory it is to be dealing with. The type of memory for each memory area is defined in a hardcoded memory map implemented in function 'decode_address_mips1', defined in package 'mips_pkg.vhdl'. This function will synthesize into regular, combinational decode logic.

For each address, the memory map logic will supply the following information:

1. What kind of memory it is.
2. How many wait states to use.
3. Whether it is writeable or not (ignored in current version).

4. Whether it is cacheable or not (ignored in current version).

In the present implementation the memory map can't be modified at run time.

These are the currently supported memory types:

Identifier	Description
MT_BRAM	Synchronous, on-chip FPGA BRAM
MT_IO_SYNC	Synchronous, on-chip register (meant for I/O)
MT_SRAM_16B	Asynchronous, off-chip static memory, 16-bit wide
MT_SRAM_8B	Asynchronous, off-chip static memory, 8-bit wide
MT_DDR_16B	Unused yet
MT_UNMAPPED	Unmapped area

Invalid memory accesses

Whenever the CPU attempts an invalid memory access, the 'unmapped' output port of the Cache module will be asserted for 1 clock cycle.

The accesses that will raise the 'unmapped' output are these:

1. Code fetch from address decoded as MT_IO_SYNC.
2. Data write to memory address decoded as other than RAM or IO.
3. Any access to an address decoded as MT_UNMAPPED.

The 'unmapped' output is ignored by the current version of the parent MCU module – it is only used to raise a permanent flag that is then connected to a LED for debugging purposes, hardly a useful approach in a real project.

In subsequent versions of the MCU module, the 'unmapped' signal will trigger a hardware interrupt.

Note again that the memory attributes 'writeable' and 'cacheable' are not used in the current version. In subsequent versions 'writeable' will be used and 'cacheable' will be removed.

Uncacheable memory areas

There are no predefined 'uncacheable' memory areas in the current version of the core; all memory addresses are cacheable unless they are defined as IO (see mips_cache.vhdl).
In short, if it's not defined as MT_UNMAPPED or MT_IO_SYNC, it is cacheable.

5.4 Cache Refill and Writethrough Chronograms

The cache state machine deals with cache refills and data writethroughs. It is this state machine that 'knows' about the type and size of the external memories. when DRAM is eventually implemented, or when other widths of SRAM are supported, this state machine will need to change.

The following subsections will describe the refill and writethrough procedures. Bear in mind that there is a single state machine that handles it all – refills and writethroughs can't be done simultaneously.

SRAM interface read cycle timing – 16-bit interface

The refill procedure is identical for both D- and I-cache refills. All that matters is the type of memory being read.

```
==== Chronogram 4.1: 16-bit SRAM refill -- DATA ============================

              --    --    --    --    --    --     -     --    --    --    --
clk         _/  \__/  \__/  \__/  \__/  \__/  \__/ ..._/  \__/  \__/  \__/

cache/ps    ?| (1)             | (2)             | ... | (2)             |??

refill_ctr  ?| 0                                 | ... |  3              |??

chip_addr   ?|  210h           |  211h           | ... |  217h           |--

data_rd     -XXXXX  [218h]     XXXXX  [219h]     | ... XXXXX  [217h]     |--
             |<- 2-state sequence              ->|
            -                                                             --
sram_oe_n    \____________________________________ ... __________________/
             |<- Total: 24 clock cycles to refill a 4-word cache line  ->|
============================================================================
```

(NOTE: signal names left-clipped to fit page margins)

In the diagram, the data coming into bram_data_rd is depicted with some delay.

Signal *cache/ps* is the current state of the cache state machine, and in this chronogram it takes the following values:

1. data_refill_sram_0
2. data_refill_sram_1

Each of the two states reads a halfword from SRAM. The two-state sequence is repeated four times to refill the four-word cache entry.

Signal *refill_ctr* counts the word index within the line being refilled, and runs from 0 to 4.

SRAM interface read cycle timing – 8-bit interface

The refill from an 8-bit static memory is essentially the same as depicted above, except we need to read 4 bytes (over the LSB lines of the static memory data bus) instead of 2 16-bit halfwords. The operation takes correspondingly longer to perform and uses an extra address line but is otherwise identical.

TODO: 8-bit refill chronogram to be done.

16-bit SRAM interface write cycle timing

The path of the state machine that deals with SRAM writethroughs is linear so a state diagram would not be very interesting. As you can see in the source code, all the states are one clock long except for states *data_writethrough_sram_0b* and *data_writethrough_sram_1b*, which will be as long as the number of wait states plus one. This is the only writethrough parameter that is influenced by the wait state attribute.

A general memory write will be 32-bit wide and thus it will take two 16-bit memory accesses to complete. Unaligned, halfword or byte wide CPU writes might in some cases be optimized to take only a single 16-bit memory access. This module does no such optimization yet. For simplicity, all writethroughs take two 16-bit access cycles, even if one of them has both we_n signals deasserted.

The following chronogram has been copied from a simulation of the 'hello' sample. It's a 32-bit wide write to address 00000430h. As you can see, the 'chip address' (the address fed to the SRAM chip) is the target address divided by 2 (because there are 2 16-bit halfwords to the 32-bit word). In this particular case, all the four bytes of the long word are written and so both the we_n signals are asserted for both halfwords.

In this example, the SRAM is being accessed with 1 WS: WE_N is asserted for two cycles. Note how a lot of cycles are used in order to guarantee compliance with the setup and hold times of the SRAM against the we, address and data lines.

```
==== Chronogram 4.3: 16-bit SRAM writethrough, 32-bit wide =================

                     --    --    --    --    --    --    --    --    --    -
clk                _/  \__/  \__/  \__/  \__/  \__/  \__/  \__/  \__/  \__/

cache/ps           ?| (1) | (2) | (3)       | (4) | (5) | (6)       | (7) |?

sram_chip_addr     ?|          218h               |       219h            |?

sram_data_wr       -------|       0000h           |         044Ch         |-
                   -------------             -----------             -------
sram_byte_we_n(0)               \___________/           \___________/
                   -------------             -----------             -------
sram_byte_we_n(1)               \___________/           \___________/

                   ---------------------------------------------------------
sram_oe_n
============================================================================
```

Signal *cache/ps* is the current state of the cache state machine, and in this chronogram it takes the following values:

1. data_writethrough_sram_0a
2. data_writethrough_sram_0b
3. data_writethrough_sram_0c
4. data_writethrough_sram_1a
5. data_writethrough_sram_1b
6. data_writethrough_sram_1c

5.5 Known Problems

The cache implementation is still provisional and has a number of acknowledged problems:

1. All parameters hardcoded – generics are almost ignored.
2. SRAM read state machine does not guarantee internal FPGA T_{hold}. In my current target board it works because the FPGA hold times (including an input mux in the parent module) are far smaller than the SRAM response times, but it would be better to insert an extra cycle after the wait states in the sram read state machine.
3. Cache logic mixed with memory controller logic.

Section

6

Logic Simulation

The project has been simulated using Modelsim 6.3g. The test bench uses some features not present in earlier versions (namely library Signal Spy) so if you use some other simulator or some earlier version of Modelsim, see section on Modelsim dependencies (6.5) below.

In short, the simulation test bench is meant to run any of the code samples provided in directory /src, under a controlled environment, while logging the cpu state to a text log file.

This log file can then be compared to a log file generated by a software simulator for the same code sample (see section 7.1). The software simulator is the 'golden model' against which the cpu is tested, so any difference between both log files means trouble.

This method is far easier than building a fully automated test bench, and much more convenient and reliable than a visual inspection of the simulation state.

In addition to the main log file, there is a console log file to which all data written to the UART is logged (see section 6.4).

The simulation test bench can be found in file '/vhdl/tb/mips_tb.vhdl'. This test bench is meant to be used with all the code samples.

Each of the code samples configures the simulation test bench with certain parameters (such as simulation length or memory sizes) and of course each sample has a different object code to be run. The way to pass these parameters to the simulation is through a simulation package, in file '/vhdl/tb/sim_params_pkg.vhdl'.

This file is generated from a template whenever you 'make' each code sample (see section 8). The package is built using oe of the provided tools, 'build_pkg', explained in section **??**.

Note that all code samples share the same vhdl files: you need to run the makefile target 'sim' for the sample you want to simulate; that will overwrite the package file mentioned above. So there's no vhdl file that is specific to a particular code sample.

While the test benches and sample code are good enough to catch MOST errors in the full system (i.e. cache included) they don't help with diagnostic; once you know there's an error, and the approximate address where it's triggered (approximate because of the cache) you have to dig into the simulation waveforms to find it.

6.1 Running the Simulation

A simulation script can be found at '/sim/mips_tb.do'. This script will simulate the test bench entity in file '/vhdl/tb/mips_tb.vhdl'.

All the code samples are run with the same script.

The test bench files mentioned in the previous section are automatically generated for each of the sample programs. This is automatically done by the sample code makefile, assuming you have a MIPS cross-toolchain in your computer (see section 8).

For convenience, a pre-generated file 'sim_params_pkg.vhdl' is included so you can launch a simulation without having to install toolchains, etc. The code is that of the 'hello world' sample.

I guess that if you are interested in this sort of stuff then you probably know more about Modelsim than I do. Yet, here's a step-by-step guide to simulating the 'hello world' sample:

1. Run 'make hello_sim' from directory '/src/hello'. This will compile the program sources, build the necessary binary object files and then create the package file mentioned above. Read the makefile and comments in the python script '/src/bin2hdl.py' for details.

   ALTERNATIVELY, if you don't have a toolchain you can just skip this step and use the default vhdl files provided, which are those of the 'hello world' sample.

2. Within Modelsim, change directory to /syn. Modelsim will create its stuff in this directory. This includes the log file, which by default will be '/syn/hw_sim_log.txt', and the console log file '/syn/hw_sim_console.log'.
   (You could use any other directory, this is just a convenient place to put modelsim data out of the way. Just remember where the log files are.).

3. Run script '/sim/mips_tb.do' (menu tools->tcl->execute macro) The simulation will run to completion and print a message in Modelsim's transcript window when it's done. You can open the console log file to see the program output, in this case the 'Hello world' message.

The test bench terminates the simulation when:

1. It detects two consecutive code fetches from the same address.
2. The simulation timeout is reached.

Condition 1 is meant to detect single-instruction loops such as those commonly found after the invocation of the main() function in a C program. This is a convenient way for the software to signal its termination.

The timeout is one of the simulation parameters which is defined in the makefiles. It is arbitrarily fixed for each sample by trial and error so that the program has time to execute. Change them if necessary.

6.2 Simulation File Logging

The simulation test bench will log any of the following events:

- Changes in the register bank.
- Changes in registers HI and LO (implemented even if mul/div is not).
- Changes in registers EPC and SR.
- Data loads (any resulting register change is logged separately).
- Data stores.

Note that changes in other internal registers, including PC, are not logged. This means that for example a long chain of NOPs, or MOVEs that don't change register values, will not be seen in the log file. This is on purpose.

Events are logged with the address of the instruction that triggered the change. This holds true even for load instructions.

The simulation log file is stored by default in modelsim's working directory (see above). I don't provide any automated script to do the comparison, you should use whatever diff tool you like best.

6.3 Log File Format

There is a text line for each of the following events:

- Register change

  "(pc) [reg_num]=value"

  Where:

pc	=>PC value (8-digit hex)
reg_num	=>Register index (2-digit hex), or any of LO,HI,EP
value	=>New register value (8-digit hex)

- Write cycle (store)

  "(pc) [address] \|mask\|=value WR"

  Where:

pc	=>PC value (8-digit hex)
address	=>Write address
mask	=>Byte-enable mask (2-digit hex)
value	=>Write data

  The PC value is the address of the instruction that caused the logged change, NOT the actual value of the PC at the time of the change. This is so to make the hardware logger's life easier – the SW simulator and the real HW don't work exactly the same when the cache starts stalling the cpu (which the SW does not simulate) and the best reference point for all instructions is their own adddress.

  The mask will have a '1' at bits 3..0 for each byte write-enabled. MSB is bit 3, LSB is bit 0. Note that the data is big endian, so the MSB is actually the LOWER address. The upper nibble of the mask is always 0.

  The value will match the behavior of the ion cpu; the significant byte(s) will have the actual write data and the other bytes will not be relevant but will behave exactly as the real hardware (so that the logs are directly comparable).

  The WR at the end of the line is for visual reference only.

- Read cycle (load)

  "(pc) [address] <** >=value RD"

  Where:

pc	=>PC value (8-digit hex)
address	=>Read address
<** >	=>Padding (ignore)
value	=>Read data

  Note that in the real machine, the data is read into the cpu one cycle after the address bus is output (because the memory is synchronous) so that the full read cycle spans 2 clock cycles (when proper interlocking is implemented, the load will overlap the next instruction; right now it just stalls the pipeline for 1 cycle). This is simplified in the log files for readability.

  Note that the size of the read (LH/LB/LW) instruction is not recorded: the CPU always reads 32-bit words.

  The RD at the end of the line is for visual reference only.

For example, these are lines 1153-1162 of the simulation log for the default 'hello world' test program:

```
...
(BFC009AC) [05]=20000000
(BFC009B0) [20000020] <**>=00000003 RD
(BFC009B0) [03]=00000003
(BFC009B8) [03]=00000002
(BFC009C0) [03]=20000000
(BFC009C4) [20000000] |0F|=00000070 WR
(BFC00E74) [12]=00000004
(BFC00E78) [10]=BFC01048
(BFC00E7C) [BFC01048] <**>=00000069 RD
(BFC00E7C) [05]=00000069
...
```

(NOTE: this example taken from revision 176, yours may vary)

The read cycle at pc=0xbfc009b0 modifies register 0x03; that's why there are two lines with the same pc value.

The code that produced that log is this (from hello.lst):

```
...
bfc009ac:    3c052000     lui   a1,0x2000
bfc009b0:    8ca30020     lw    v1,32(a1)
bfc009b4:    00000000     nop
bfc009b8:    30630002     andi  v1,v1,0x2
bfc009bc:    1060fffc     beqz  v1,0xbfc009b0
bfc009c0:    3c032000     lui   v1,0x2000
bfc009c4:    ac620000     sw    v0,0(v1)
bfc009c8:    03e00008     jr    ra
bfc009cc:    00000000     nop
...
bfc00e74:    26520001     addiu  s2,s2,1
bfc00e78:    26100001     addiu  s0,s0,1
bfc00e7c:    92050000     lbu    a1,0(s0)
bfc00e80:    00000000     nop
...
```

(Remember the register numbers: \$v0=0x02, \$v1=0x03, \$a1=0x05, \$s0=0x10, \$s2=0x12)

Note that, unlike previous versions of this project, all changes are logged with the address of the instruction that caused them.

The log file format is hardcoded into vhdl package mips_sim_pkg and the software simulator C source that implement it. It will be probably modified as the project moves on so it is best if you verify all of this yourself with the project version you intend to use before using this information.

Note that the software simulation log and the modelsim log need not be the same size even if both CPUs behave identically; the one that spans a longer simulated time will be longer.
The point is that both need to be identical up to the last line of the shortest file.

6.4 Console Output Logging

Every byte written to the UART TX register is logged (in ascii) to a text file which by default is '/syn/hw_sim_console.log'. Apart from the automatic insertion of a CR after every LF, the data is logged verbatim.

Though the UART is included in the test bench, the actual UART operation is bypassed: The test bench forces the 'tx ready' high so that the CPU never has to wait for a character to be transmitted. This is a simplification that saves me the trouble to do a cycle-accurate simulation of the UART in the software simulator.

The UART input is not simulated at all, for simplicity. So, for example, the Adventure sample, which does read the console input, can't be properly simulated past the first console input – there is plenty of code to simulate before that so this is no problem for the moment.

6.5 Use of Modelsim Features

Apart from the format of the simulation scripts, which would be easy to port to any other simulation tool, the simulation test bench uses a feature of Modelsim 6.3 that is not even present in earlier versions – SignalSpy.

The test bench uses SignalSpy to examine internal cpu signals from the top entity, including the whole register bank. There is no other way to examine those signals in vhdl, unless you want to add them to the module interface.

The test bench needs to access those signals in order to detect changes in the internal cpu state that should be logged. That is, it really needs to look at those signals if it is to be of any use.

If you are using any other simulation tool, look for an alternative method to get those internal signals or just add them to the core interface. I would suggest adding a debug port of type record to mips_cpu – and hope the synthesis tool does not choke on it. Adding individual debug ports would be a PITA.
I guess this is why Mentor people took the trouble to write SygnalSpy.

I plan to move to Symphony EDA eventually, so I'll have to fix this.

Using GHDL would be an option, except because it only supports vhdl. The project will use a SDRAM model in verilog for which I could not find a vhdl replacement. If the project is to be ported to GHDL (a very desirable goal even if only because not everybody has access to Modelsim) this will have to be worked around.

Section

7

Tools

Directory '/tools' of the project includes a few tools – small C or Python programs purpose-built for this project.

What follows is a brief description of each of the tools. This document won't go into the implementation or usage details. The tools themselves have brief usage instructions and for any further details the user must read the source code.

7.1 MIPS Software Simulator

Plasma project includes a MIPS-I simulator made by Steve Rhoads, called 'mlite.c'. According the the author, it was used as a golden model for the construction of the cpu, the same as I have done.
I have made some modifications on Rhoads' code, mostly for logging, and called the new program 'slite' ('/tools/slite/src/slite.c').

The most salient features are:

- Logs CPU state to a text file. The format is identical to that of the vhdl test bench log. You can select the code address that triggers the logging.
- Echoes CPU UART output to the host console or to a log file.
- Can be run in interactive mode (like a monitor).
  Step by step execution, breakpoints, that kind of thing.
- Can be run in batch (unattended) mode.
  So that you can easily run a program to compare logs with the vhdl test bench.
- Does not simulate the cache at all.

Each code sample includes a DOS batch file named 'swsim.bat' that runs the simulator in batch mode. Note that the BAT file invokes a windows binary which is included in the SVN repository and should be immediately useable after checkout.

The program includes usage help (a short description of the command line parameters). The source code (very simple and straighforward) is included in the project. The BAT files provide an usage example. And anyone who is interested and finds trouble can always contact me.

For all these reasons I think it is not necessary to explain the simulator in detail. Nothing to do with laziness, as you can see.

Many system parameters are hardcoded, including the log file name, the simulated memory sizes and the code and data addresses.

The hardcoded log file name is "sw_sim_log.txt" and it is generated in the same directory from which the simulator is run.

7.2 Configuration Package Builder Script build_pkg.py

This tools is used to build a simulation and synthesis configuration package.

The generated package contains configuration constants used by the simulation test bench *'mips_tb.vhdl'* and by the hardware demo *'c2sb_demo.vhdl'*.

It too includes memory initialization constants containing object code, used to initialize simulated and inferred memories, both in simulation and in synthesis.

In the code samples, this script is used to generate two separate packages for simulation and synthesis. Please refer to the makefiles for detailed usage examples.

7.3 Conversion Script bin2hdl.py

NOTE: This script was used in previous versions of the project – it came in handy to initialize byte-sliced memories when the caches were under development.
It has been abandoned because it was far too complicated and no longer necessary. The VHDL templates it refers to and the script itself have been moved from the /src directory to their own subdirectory in /tools.
It is being retained in case it becomes useful again but it is no longer used.

This Python script reads one or more binary files and 'inserts' them in a vhdl template. It makes the conversion from binary to vhdl strings and slices the data in byte columns, as required by the RAM implementation (in which each byte in a word is stored in a different RAM with a separate WE, 4 blocks in all).

The 3 binary files the script can read are the object code image, the data image (initialized data sections) and a FLASH image.

The script inserts a number of simulation parameters in the template file, as illustrated by the makefiles.

The makefiles of the code samples can be used as an example. The script code is a bit convoluted but it is understandable if you do know Python, and includes some usage instructions.

The vhdl templates (/src/*_template.vhdl) have placeholder 'tags' that are replaced with real application data by this script.

Some of the tags are these:

"@code0@"	: Contents of RAM block for slice 0 (lsb) of code
...	
"@code3@"	: Contents of RAM block for slice 3 (msb) of code
"@code31@"	: Contents of RAM block for slices 3 & 1 (odd) of code
"@code20@"	: Contents of RAM block for slices 2 & 0 (odd) of code
"@data0@"	: Contents of RAM block for slice 0 (lsb) of data
...	
"@data3@"	: Contents of RAM block for slice 3 (msb) of data
"@data31@"	: Contents of RAM block for slices 3 & 1 (odd) of data
"@data20@"	: Contents of RAM block for slices 2 & 0 (odd) of data
"@flash@"	: Contents of simulated FLASH
"@data-32bit@"	: Contents of 32-bit-wide RAM block of data
"@entity_name@"	: Name of entity in target vhdl file
"@arch_name@"	: Name of architecture in target vhdl file
"@code_table_size@"	: Size of RAM block to be used for code, in words
"@code_addr_size@"	: ceil(log2(@code_table_size@))
"@data_table_size@"	: Size of RAM block to be used for data, in words
"@data_addr_size@"	: ceil(log2(@data_table_size@))

There's a few more tags; they are described in the script source and the usage help.

These placeholders will be replaced with object code or with data values provided by the script command line (see makefiles).

The script has been used with Python 2.6.2. It should work with earlier or later versions but I haven't tested.

Note: all of the above info is in the script itself, and can be shown with command line option –h. Since it will be more up to date than this doc, you're advised to read the script.

Section 8

Code samples

Directory /src directory contains a few test applications that can be simulated and run on real hardware, except for the opcode test which can only be simulated. See the readme file and the makefile for each program.

Please read the /src/reame.txt file for information that will probably be more up-to-date than this doc.

The makefiles have been tested with the CodeSourcery toolchain for windows (that can be downloaded from www.codesourcery.com) and with the Buildroot toolchain for GNU/Linux.

Most makefiles have two targets, to create a simulation test bench and a synthesizable demo.

Target 'sim' will build the simulation test bench package files as described in section 6.

Target 'demo' will build a synthesizable demo; it will compile the sample sources and place the resulting object code in file '/vhdl/SoC/bootstrap_code_pkg.vhdl'.

The build process will produce two or more binary files ('*.code' and '*.data', or '*.bin') that can be run on the software simulator, plus a listing file (*.lst) handy for debugging.

All projects include a DOS batch file 'swsim.bat' that invokes the software simulator with the proper parameters. As an example, these are the contents of the 'swsim.bat' file for the 'hello' demo:

```
@rem Run software simulator in hands-off mode
..\..\tools\slite\slite\bin\Debug\slite.exe ^
    --bram=hello.code ^
    --trigger=bfc00000 ^
    --noprompt ^
    --nomips32 ^
    --map=hello.map ^
    --trace_log=trace_log.txt
```

As you can see, the simulator is invoked in 'batch' or 'hands-off' mode, so the simulated program will be run to completion, generating a simulation log. The point of this is comparing that log to the log generated by the Modelsim simulation of the same program, as has already been explained.

The python script 'bin2hdl.py' is used to insert binary data on vhdl templates. Assuming you have Python 2.5 or later in your machine, call the script with:

```
python bin2hdl.py --help
```

to get a short description (see section 7.3).

8.1 Support Code

Bootstrap Code

File *'/src/common/bootstrap.s'* contains the reset code and the stub interrupt handler.

This code is meant to be placed at the reset vector address; in the present revision of the project, the bootstrap code would be placed in the bootstrap BRAM of the SoC module (see section 2.2). It can be placed in external memory if the SoC memory map and the makefiles are altered accordingly.

The reset code initializes both I-Cache and D-Cache and jumps to symbol `'entry'` with the CPU in kernel mode and interrupts disabled.

The interrupt code is somewhat more elaborate but nevertheless is still a stub. The interrupt code will find out the cause of the interrupt and will proceed.

Table 8.1 shows interrupt sources recognized in the current version, and how the interrupt code deals with them.

Please note that the interrupt code does not even *know* there is such a thing as a hardware interrupt – hardware interrupts are not implemented yet in the CPU.

Table 8.1: Interrupt handling

Cause	Processing
SYSCALL instruction	Return immediately (STUB).
BREAK instruction	Return immediately (STUB).
Invalid opcode	Return immediately (STUB).
Unimplemented MIPS-32 opcode	Emulate opcode.

As can be seen, most of the interrupt processing is missing, replaced by stubs. Eventually, those stubs will be replaced with calls to C functions that will be defined in the supporting libraries. I have yet to work out exactly how to do it without reinventing the wheel.

The only interrupt processing that is performed (even if only partially) is the emulation of *some* MIPS-32 opcodes.

MIPS-32 Opcode Emulation

I have found out that most MIPS toolchains target the MIPS-32 architecture and can only be made to work with the MIPS-I architecture with some difficulty. This applies specially to the C support libraries, where MIPS-32 opcodes are used occasionally. Other reasons, such as the availability of MIPS-32 ports of uClinux, make at least partial compatibility to MIPS-32 very desirable.

On the other hand, extending the core to implement the full MIPS-32 specs (as opposed to the far simpler MIPS-I) might not be possible without running into a patent minefield – I may be wrong in this.

Therefore, I have chosen to just emulate whatever subset of the MIPS-32 ISA is enough to overcome the above obstacles. I have run some experiments with uClinux and have found out that only a few MIPS-32 opcodes need to be emulated – see table 8.2.

Opcode emulation is implemented in file */src/common/opcode_emu.s*.

The emulation code has been tested (code sample *'opcodes'*) but is still unfinished: not only are there opcodes to be emulated, as can be seen in table 8.2, but the emulation does NOT work when the MIPS-32 opcode is in a delay slot; or more precisely, the jump instruction of the delay slot is not emulated as it should – this is just work to be done, nothing specially difficult.

If the trapped MIPS-32 opcode is not one of the emulated opcodes, it is ignored exactly as if it was a NOP. A flag is raised in a special, reserved area of the BSS segment – see the source for details. Eventually the trap handling will be complete and unimplemented MIPS-32 opcodes will be dealt with as undefined opcodes.

Note that opcode emulation can be disabled in the makefiles, by defining symbol `NO_EMU_MIPS32` when assembling *bootstrap.s*.

Table 8.2: Emulated MIPS-32 opcodes

Opcode	Emulation
EXT	Fully emulated.
INS	Fully emulated.
CLO	Fully emulated.
CLZ	Fully emulated.
MUL (3-reg version)	To Be Done.
LWL	To Be Done.
LWR	To Be Done.
SWL	To Be Done.
SWR	To Be Done.

C Startup Code

File *'/src/common/c_startup.s'* contains the C startup code used in all C code samples.

This startup code does what's usual in these cases, namely:

1. Initialize the stack at the end of the bss area.
2. Clear the bss area.
3. Move the data section from FLASH to RAM (if applicable).
4. Call main().
5. Freeze in endless loop after main() returns, if it does.

See the makefile of any of the C code samples for examples on how to assemble, configure and link the startup code.

C Support Library Replacement

The core will eventually be tested with one of the regular, free libc replacement libraries available. It isn't yet because building those libraries for a MIPS-I target (i.e. with no MIPS-32 stuff mixed in) has turned out to be much more difficult than I had anticipated.

In order to be able to use the C toolchain and provide some code samples in C, I have built a minimalistic libc replacement called 'libsoc'. The source code can be found in */src/common/libsoc/src*.

This mini-libc provides only those C support functions I have found necessary so far; it will be extended with new functionality as I add more code samples, until I can finally use some real C library.

Apart from a number of utility functions that the C runtime needs, libsoc provides the following:

- Floating point support (SW, float and double).
- `putchar` and `getchar` using the SoC UART.
- Replacement for the built-in `printf` provided by gcc.

That's it. Yet, even this little is enough to run the Adventure demo...

All the source code has been lifted from the original Plasma supporting code or has been scavenged from the net – see credits in the file headers.

Section

9

Hardware Demo

9.1 Pre-generated demo

The project includes a few synthesizable code samples, including a 'Hello world' demo and a memory tester. Only the 'hello' demo is included in pre-generated form, the others have to be built using the included makefiles – assuming you have a mips toolchain.

'Pre-generated' in this context means that all the vhdl files necessary for building the demo are already included with the project, including the configuration package that contains the program's object code, and the only tool needed is the synthesis tool.

The pregenerated demo is included just for convenience, so that you can launch some small application on hardware without installing a C toolchain.

A constraints file is provided ('/vhdl/demo/c2sb_demo.csv') which includes all the pin constraints for the default target board, in CSV format. This constraints file is shared by all demos targeted to the DE-1 board.

The default target board is TerasIC's DE-1, with a Cyclone-II FPGA (EP2C20F484C7). This is the only hardware platform the core has been tested in, so far.

I have used the free Altera IDE (Altera Quartus II 9.0). This version of Quartus does not even require a free license file and can be downloaded for free from the altera web site. But if you have a DE-1 board on hand I guess you already know that.

I assume you are familiar with Altera tools but anyway this is how to set up a project using Quartus II:

1. Create new project with the new project wizard. Top entity should be c2sb_demo. Sug-

gested path is /syn/altera/(project name).

2. Set target device as EP2C20F484C7. This choice tells the synth tool what speed grade and chip package we'll be targetting.
3. 'Next' your way out of the new project wizard.
4. Add to the project all the vhdl files in /vhdl and /vhdl/demo, except mips_cache_stub.vhdl and sdram_controller.vhdl.
5. Add to the project all the vhdl files in /vhdl/SoC.
6. Select file c2sb_demo.vhdl as top.
7. Import pin constraints file (assignments->import assignments).
8. Create a clock constraint for signal clk (51 MHz or some other suitable speed which gives us some minimal slack).
9. In the device settings window, click "Device and pin options...".
10. Select tab "Dual-Purpose pins".
11. Double-click on nCEO value column and select "use as regular I/O". IMPORTANT: otherwise the synthesis will fail; we need to use a FPGA pin that happens to be dual-purpose (programming and regular).
12. Select 'speed' optimization.
13. Save the project and synthesize.
14. Make sure the clock constraint is met (timing analyzer report). There is a random element to the synthesis process, as you know, but the core as shipped should pass the constraint.
15. Program the FPGA from Quartus-2
16. If you have a terminal hooked to the serial port (19200/8/N/1) you should see a welcome message after depressing the reset button. (by default this is pushbutton 2).

In the present version, the synthesis will produce a lot of warnings. The ugliest are about unused pins and an undeclared clock line. None of them should be really scary.

Note that none of the on-board goodies are used in the demo except as noted in section 9.2 below.

In order to generate the demos (not using the pre-generated file) you have to use the makefiles provided with the code samples. Please see the sample readme files and the makefiles for details. In short, provided you have a MIPS toolchain installed and Python 2.5+, all you have to do is run make (which will automatically build all the vhdl files where they need to be, etc.) and run the synthesis.

9.2 Porting to other dev boards

I will only deal here with the 'hello' demo, the process is the same for all other samples that don't involve external FLASH.

The 'hello' demo should be easily portable to any board which has all of this:

- An FPGA capable enough (the demo uses internal memory for code).
- At least 4KB of 16-bit wide external, asynchronous, old-fashioned SRAM.
- A reset pin (possibly a pushbutton).
- A clock input (uart modules assume 50MHz, see below).
- RXD and TXD UART pins, plus a connector, header or whatever.

The only module that care at all about clock rate is the UART embedded into the SoC module. It's hardwired to 19200 bauds when clocked at 50MHz, so if you use a different frequency you must edit the generics in the demo entity accordingly – the demo generics are passed all the way down to whatever module needs them.
The UART has hardly been tested at clock rates other than 50MHz and has not passed any independent test bench; try the core first at 50 MHz.

Though there is no reset control logic, the reset input is synchronized internally, so you can use a raw pushbutton – you may trigger multiple resets if your pushbutton isn't tight but you'll never cause metastability trouble.

Assuming you take care of all of the above, the easiest way I see to port the demo is just editing the top module ports ('/vhdl/demo/c2sb_demo.vhdl') to match your board setup. The only tricky part is the interface to FLASH and SDRAM.

All the code in this project is vendor agnostic (or should be, I have only tried it on Quartus and ISE). Specifically, it does not instantiate memory blocks (relying instead on memory inference) or clock managers or buffers. This has its drawbacks but is an stated goal of the project – in the long run it pays, I think, and it certainly makes the porting easier.

9.3 'Adventure' demo

There is another demo targeting the same hardware as the 'hello' demo above: a port of 'Adventure'. The C source (included) has been slightly modified to not use any library functions nor any filesystem (instead uses a built-in constant string table).

Build steps are the same as for the hello demo (the make target is 'demo').

Since the binary executable is too large to fit internal BRAM, it has to be executed from the DE-1 onboard flash. You need to write file 'adventure.bin' to the start of the FLASH using the 'Control Panel' tool that came with your DE-1 board. That's the only salient difference. That and the amount of SRAM; The 512KB present on the DE-1 are enough but I don't remember right now what is the minimum, please look at the map file. This should only matter if you want to port to another board.

The game will offer you an auto-walkthrough option. Answer 'y' and it will play itself for about 250 moves, leaving you at an intermediate stage of the game from which you can play on.

Now, admittedly 'Adventure' is no standard benchmark and even running it to completion does not guarantee that there are no bugs hidden in the cache or any of the opcodes. On the other hand, when you get to the *maze of twisty little passages* you know you have a computer, finished or not. The 'Adventure' demo is great as a confidence builder.

Besides, running Adventure on a computer built by myself is something I've always wanted to do :)

Section

10

Design Notes

10.1 Project Goals

The first iteration of the project will be deemed finished when it can do the following:

1. Run a minimal set of MIPS-I opcodes.
   Excluding unaligned load/store (formerly patented).
   Excluding all CPA instructions.
   Excluding all CP0 instructions related to TLB.
   Cache instructions will not be implemented as defined.

2. Catch all undefined opcodes (and trigger exception).

3. Operate in kernel/user mode as per the architecture definition.

4. Handle exceptions in a manner compatible to MIPS-I standard.

5. Code cache and data cache, even if not standard.
   No MMU and no TLB, and no cache-related instructions.

6. Implement as much of CP0 as necessary for the above goals.

7. Interface to external SRAM (or FLASH) on 8- and 16-bit data bus.

8. Be no bigger than Plasma in a Spartan-3 or Cyclone-2 device, and no slower – Plasma is used as a reference in many ways.
   Speed measured in raw clock frequency for the time being. (I.e. don't not consider stalls, interlocks, etc. yet)

9. Interlock behavior of MUL/DIV and L* compatible to toolchain.
   That is, interlock loads instead of relying on a delay slot.

Unaligned load/stores are excluded not because of patent concerns (the patents already expired) but because they're not essential for a first version of the core. The same goes for all other exclusions.

As of rev. 154 all the 1st block goals have been accomplished (but not very heavily tested; many bugs remain, probably).

For a second iteration I plan on the following:

1. Proper interlocking of load cycles (with no wasted cycles).
2. External interrupt support.
3. Trap handlers (instruction emulation) for unaligned load and store instructions.
4. Trap handlers (instruction emulation) for the most usual MIPS32 instructions.
5. Some much needed optimization of the caches.

None of these things have been done.

Note that 32-bit memory interfaces are not to be implemented any time soon, mainly because I don't have any actual hardware with which to test it.

10.2 Development status

The CPU is already able to execute almost any MIPS-I code (excluding some unimplemented instructions such as cache control).
It can pass a basic opcode test and can execute some basic applications compiled with standard gcc tools (specifically, it can run an 'Adventure' demo and a tiny 'hello world' program, see section 6).

Hardware interrupt support is entirely missing.

The most important limitations are the very basic memory interface, with no support for SDRAM, and the absence of MIPS32 trap handlers – which means that the ubiquitous MIPS32 toolchains can't be easily used with this core.

The memory controller can already access external static memory (SRAM or FLASH) on 8-bit and/or 16 bit buses. Still does not support SDRAM, nor static RAM in other bus widths. My main development target is a DE-1 board from Terasic (Cyclone-2) and I have focused in the kind of memory it has.

Wait states can be configured at synthesis, see section 5.3. Code sample 'memtest' takes advantage of this to do a basic test of the external SRAM, and code sample 'Adventure' uses both Flash and SRAM. All the code samples habe been tested with the cache enabled and disabled, and they ship with the cache enabled (i.e. with C startup code that initializes and enables the cache).

The code samples can be found in the /src directory (see section 8).

This is a summary of the state of the CPU at this time:

- MIPS-I things not implemented
    1. External hardware interrupts.
- Things implemented but not fully tested.
    1. Rte instruction.
    2. Kernel/user modes.
- Things with provisional implementation
    1. Load interlocks: the pipeline is stalled for every load instruction, even if the target register is not used in the following instruction. So that every load takes two cycles.
       The interlock logic should check register indices and stall only if there is a data hazard.
       Note that all that's needed is a better identification of stall conditions; the logic to enable a load instruction that does not stall to overlap the next instruction is already in place.
       The interlock logic needs a stronger test bench anyway.
    2. Documentation is too sparse and source code is barely commented.
    3. The D-Cache handles RAW hazards in a very inefficient way.
       Data refills in a SW+LW sequence should only be triggered when the SW invalidates the same line the LW is loading. Instead, the current cache triggers the data refill always (for a SW+LW sequence, that is).
       This performance drag has to be fixed without ruining the clock rate (that's the catch).

10.3 Performance

In my main test system, a Cyclone-2 grade -7, the core with caches and with mul/div and all other necessary functionality, plus a barebones UART, will be below 2500 LEs + 18 BRAMs, running at least at 50 MHz (with 'balanced optimization' on Quartus-II).

As soon as the core is in a stable state I will include a few synthesis performance numbers for common configurations.

As soon as I can build a dhrystone benchmark I will post results (and commit the code). The core needs a timer before I can do that.

My first performance target will be a real R3000 at the same clock rate. I can anticipate that performance will be MUCH lower than that (by a factor of 4 or more) due to the bus widths and the wait states, AND the inefficient cache implementation. I'll work on that as soon as the basic stuff is done.

10.4 Next steps

- Implement efficient load interlock detection with no wasted cycles.
- Do whatever it takes to use standard C library functions.
- Alternatively, build a small C library replacement.
- Add a couple of benchmarks, including one with FP arithmetic.
- Modify the software simulator so it can boot uClinux.
- Make a uClinux port suitable for a R3000 derivative, from BuildRoot.
- Make a freeRTOS port suitable for a R3000 derivative.

Some of the above items are done, others are in progress and others are pipe dreams at this point.

Bibliography

[1] Dominic Sweetman, *See MIPS Run*. Elsevier, 2nd Edition, 2007.

[2] Dominic Sweetman, *IDT R30xx Family Software Reference Manual*. IDT, (freely available in PDF format), 1994.