This file contains instructions and notes about the Ion CPU core project. The core structure is briefly explained in section 3. The rest of this doc is mostly usage instructions for the test samples and custom utilities. In its present state it's more a reminder to myself than a proper explaination of the system.

Last modified: Jul/30/2011

1 Quick Start

The project includes a pre-generated, synthesizable demo ready to run on a DE-1 development board from Terasic: a besic 'Hello World' application. You can find instructions in 5.1.

The same application can be simulated in Modelsim (no other simulator is supported yet). Instructions can be found in section 4.1.

2 Introduction

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

2.1 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.

2.2 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).

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 (see xxx).

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 2.6 below. 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.

The code samples can be found in the /src directory (see section 6).

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

- MIPS-I things not implemented
    1. External 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).

2.3 Performance

In my main test system, a Cyclone-2 grade -7, I'm quite sure that 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, as I said before.

2.4 Next steps

- Implement efficient load interlock detection with no wasted cycles.
- Do whatever it takes to use standard C library functions (see xxx).
- 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.

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

3 CPU description

This section is about the 'core' cpu, excluding the cache.

3.1 Some general features

- Synchronized to rising edge of clk only; no latches.
- All inputs need to be synchronous to clk.
- Synchronous register bank, both read and write ports.

3.2 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 final 'top' module (called 'mcu') has its outputs registered to limit t_{co} to acceptable values.

3.2.1 Code and data read bus interface

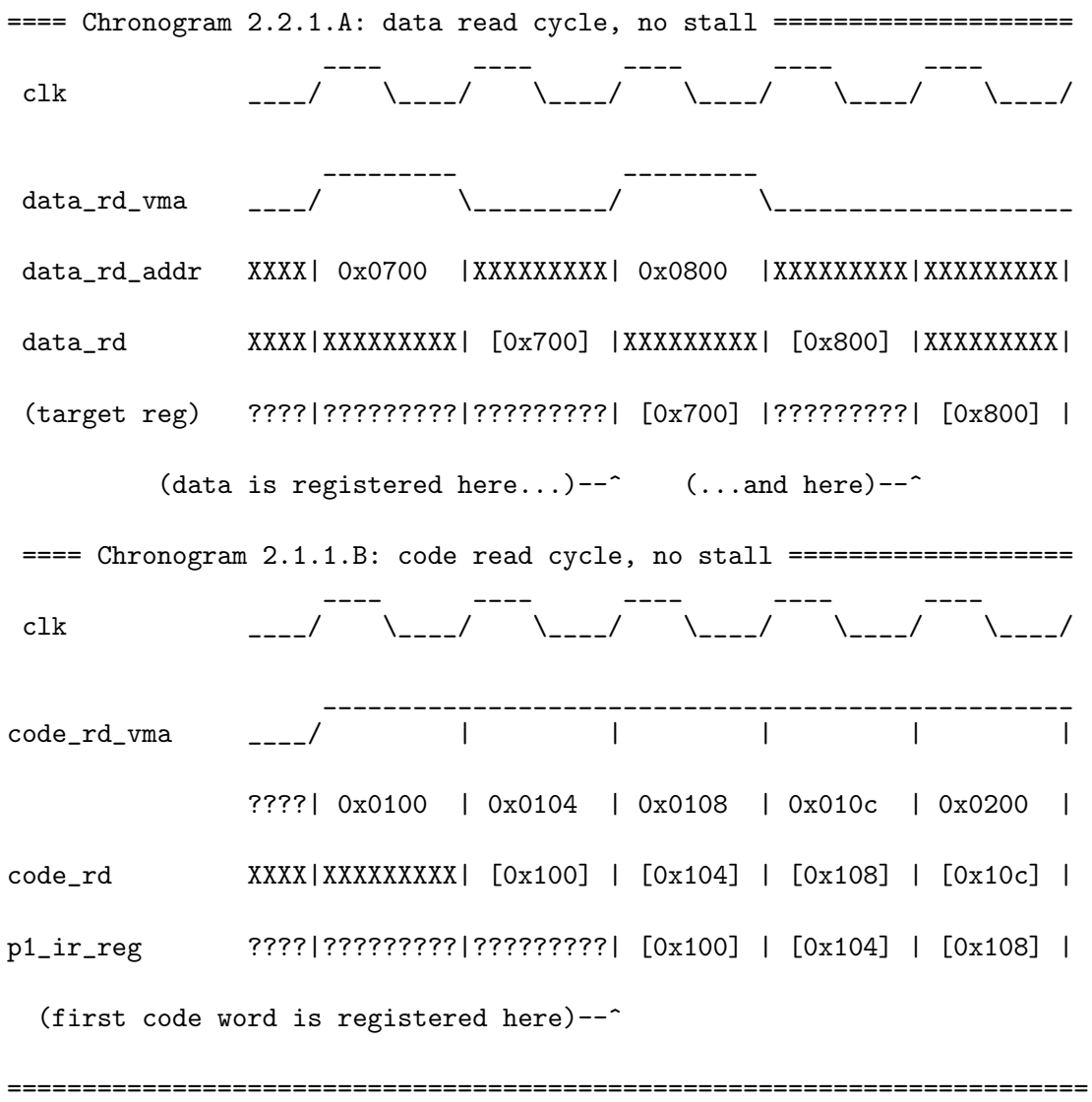
Both buses have the same interface:

⋆_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.



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.

3.2.2 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

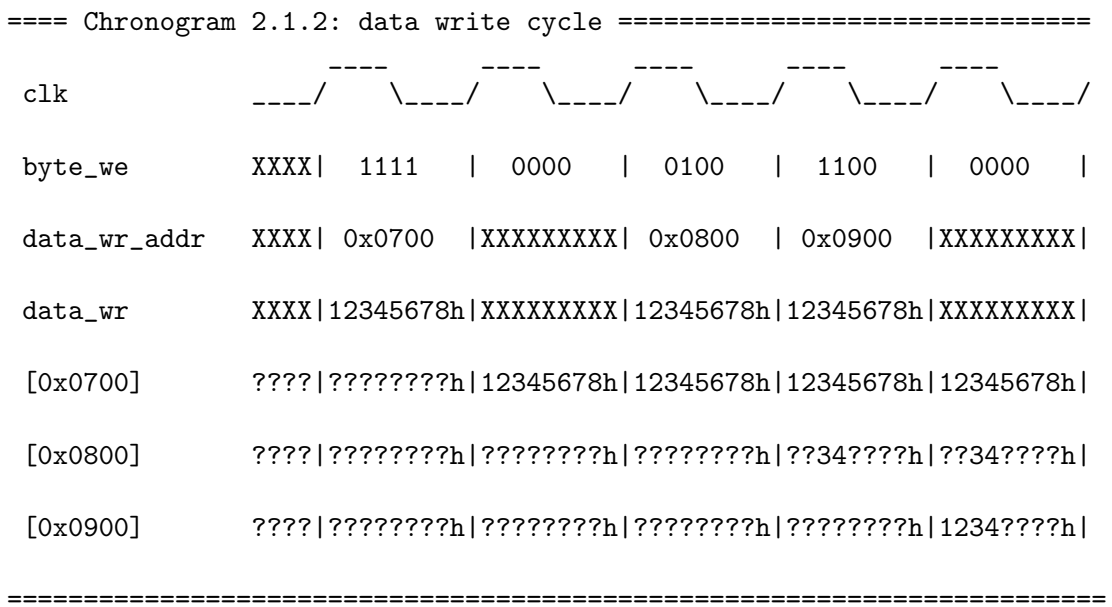
Write cycles are synchronous too. The four bytes in the word should be handled separately –

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):



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).

3.2.3 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 current implementation of the wait input and its logic is going to change. Eventually it will be described here.

In short, 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 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 xxx on the memory controller.

3.3 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.

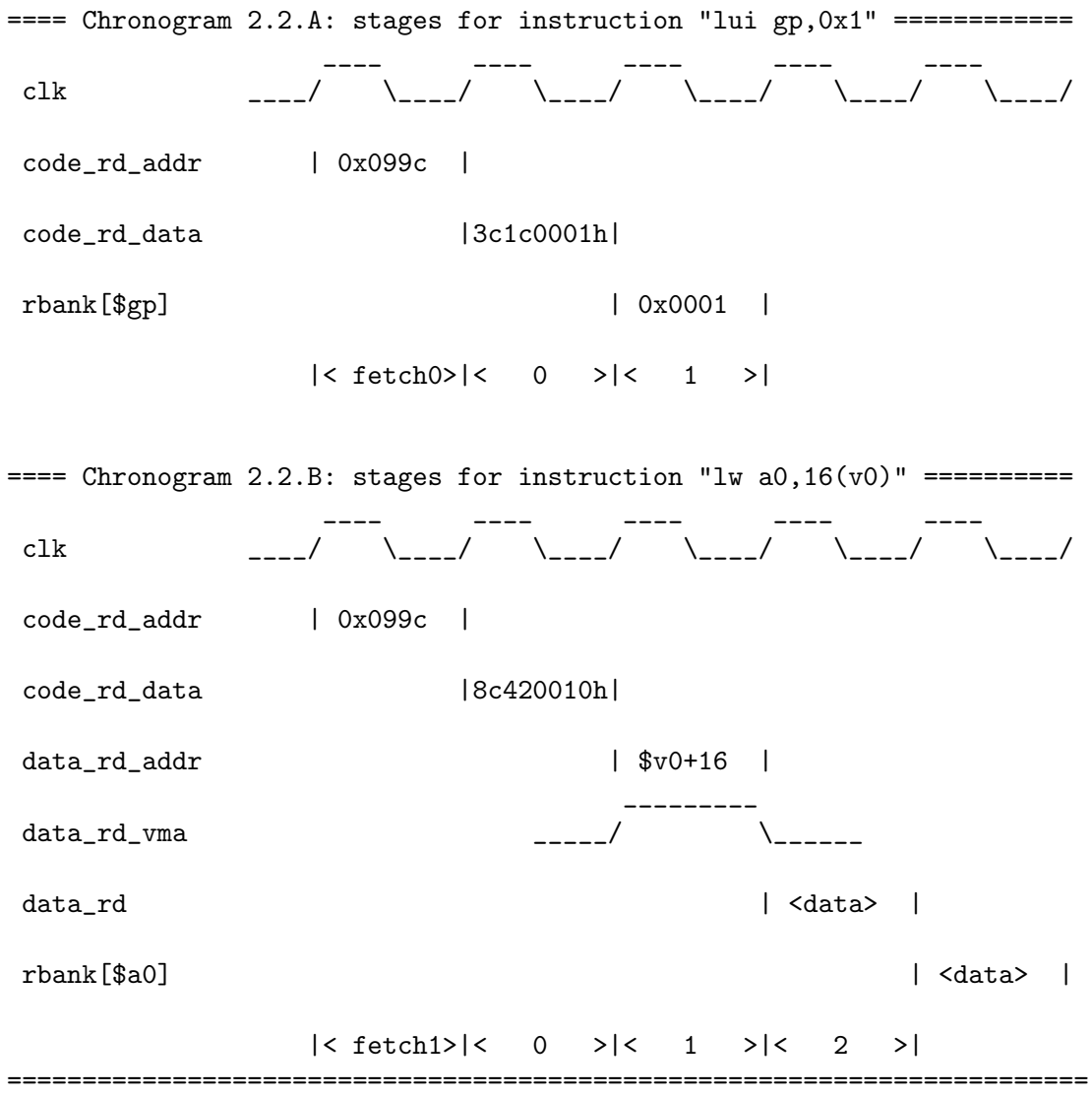
A computational instruction of the I- or R- type goes through the following stages during execution:

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_
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:



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 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. Bearing in mind that the code 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 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.

3.4 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.

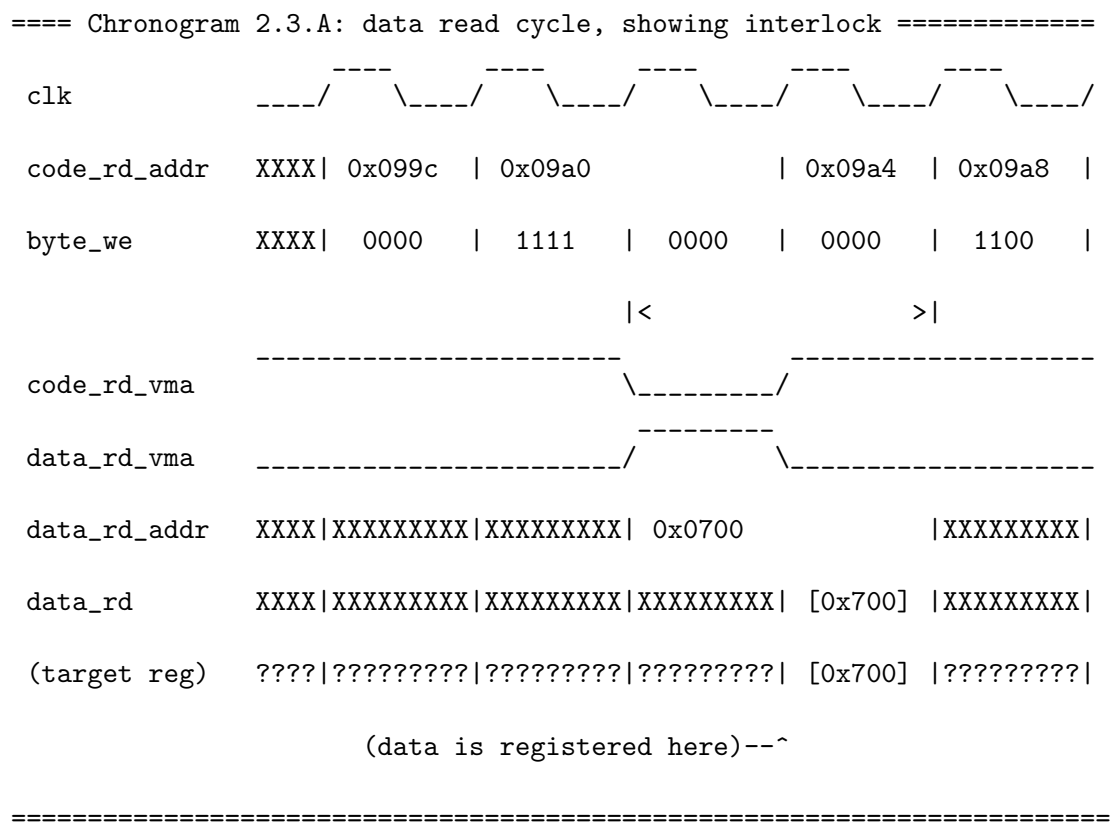
Conflict (a) is solved with some data forwarding logic: if we detect the data hazard, the register bank uses a 'feed-forward' value instead of the value read from the memory file.
In file mips_cpu.vhdl, see process 'data_forward_register' and the following few lines, where the hazard detection logic 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], pag. 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], pag. 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.



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...).

3.5 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 access to CP0 trap.
Memory provileges 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], pag. 64.

Plasma used to save in epc the address of the instruction after break or syscall, and used an unstandard vector address (0x03c). This core will use 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.

3.6 Multiplier

The core includes a multiplier/divider module from revision 18.

It uses a slightly modified version of Plasma's multiplier unit. Changes have been commented in the source code.

The main difference is the 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.

3.7 Memory map definition

The system memory map is hardcoded in function 'decode_address_mips1', defined in package 'mips_pkg.vhdl'.

The MIPS architecture specs define a memory map which determines which areas are cached and which is the default address translation ([2], page 2-8).

The core as of now does not support any kind of memory translation: program addresses are always identical to hardware addresses.
The memory map attributes are only relevant to the cache module (which, in the current version, includes the memory controller); the cache/memory controller is the only module responsible for accessing external storage; it needs to know how to do it. For each address, the cache needs to know:

1. What kind of memory it is
2. How many wait states to use
3. Whether it is readable or not
4. Whether it is cacheable or not

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

The cache module uses 'decode_address_mips1' to determine what to do for each cache refill – the refill state mechine is different for each kind of memory, see section 3.8.

Note that the cache stub implements only points 1, 2 and 3.

(NOTE: the cache module includes the memory controller, which is what actually uses all this information. The I- and D-Cache logic don't care about memory types or mappings).

3.8 Cache/memory controller module

The project includes a cache+memory controller module from revision 114. Earlier revisions used a 'stub' cache described in section 3.9.

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.

As time permits I will add timing diagrams to this section. For now, I will only say that the timing diagrams will be nearly identical to those of the stub cache with only one important difference: In the stub cache, each refill operation only reads a single word from memory, whereas in the real cache each refill reads 4 words.

The following section includes chronograms that may

3.8.1 Cache initialization and control

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]

  When '0', both caches are disabled. All memory accesses will trigger a cache refill (even sucessive accesses to the same line).
  When '1', caches are enabled and work as usual.

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

  When bits 17:16='01', 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...

  When bits 17:16='01', reading from any address will cause the corresponding data cache line to be invalidated; the read will not be actually performed and the read value is undefined.

  When bit 16 is '0', the cache will work as usual. When bits 17:16='11' cache behavior is UNDETERMINED.

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).

Note that there are no 'uncached' memory areas in the current version of the core; all memory addresses are cacheable unless they are defined as IO (see mips_cache.vhdl).

3.8.2 Cache tags, cache address mirroring and uncacheable blocks

FIXME This is explained in the cache source; should be explained here too

3.9 'Stub' cache

In revisions up to 106 the project included a 'stub cache. that module was meant as a placeholder until the rela cache was made, and worked as a memory interface plus a simple cache with a line size of 1 word.

Earlier revisions of the project include some documentation on the stub cache (file /doc/ion_project.txt). Please refer to those earlier revisions for details on this module.

4 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 3.3 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 5.1). 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 4.4).

There are a few simulation test bench templates in the /src directory, which are used by all the code samples.
The only ones actually used are '/src/code_rom_template.vhdl' and '/src/sim_params_template.vhdl'. The others are remnants of previous versions that will be removed ASAP.

The template in file '/src/code_rom_template.vhdl' is filled with object code meant to be run from internal FPGA BRAM. This is how we load bootstrap code into our FPGA. The resulting file is '/vhdl/demo/code_rom_pkg.vhdl' and is used by both the simulation test bench and the synthesizable MCU.

The template in file '/src/sim_params_template.vhdl' is filled with simulation parameters (such

as the simulation length, etc.) and the resulting file is written as '/vhdl/tb/sim_params_pkg.vhdl'. This file is only used by the simulation test bench.

All of this template filling is done by a python script (/src/bin2hdl.py) which is invoked from the makefiles.

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 two files mentioned above. So there's no vhdl file that is specific to a particular code sample.

The actual test bench entity is at '/vhdl/tb_tb.vhdl' and is shared by all the code samples.

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. It's easier than it seems.

4.1 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 xxx).

For convenience, a pre-generated mips_tb.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 thw two package files 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. In 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 end 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.

4.2 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.
Note that early versions of the project logged the address of the preceding instruction – it was confuse and I have fixed it.

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.

4.3 Log file format

FIXME: the log examples below are from an early version of the logger that did not use the instruction address but the previous address. They are very misleading and should be updated.

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

For example, these are lines 274-281 of the simulation log for the default 'hello world' test bench:

```
...
(00000230) [04]=00010000
(00000234) [1D]=00010230
(00000238) [00010244] |0F|=00000038 WR
(0000023C) [1F]=00000240
(00000288) [00010000] <**>=636F6D70 RD
(00000288) [05]=00000063
(00000294) [07]=0000000A
(00000298) [03]=20000000
...
```

(NOTE: this example taken from revision 1, yours will probably vary)

The read cycle at pc=0x288 modifies register 0x05; that's why there are two lines with the same pc value.

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

```
...
22c:    3c040001    lui a0,0x1
230:    27bdffe8    addiu   sp,sp,-24
234:    afbf0014    sw  ra,20(sp)
```

```
238:    0c0000a1    jal 0x284
23c:    24840000    addiu   a0,a0,0
...
284:    90850000    lbu a1,0(a0)
288:    00000000    nop
...
```

(Remember the register numbers: $a0=4, $sp=1d and $ra=1f)

Please note the addiu instruction in the jal delay slot (which is effectively a nop)and how all changes are logged with the 'next' pc value: the jal change to register 0x1f is logged at the jump target address, and not at the delay slot address. I hope this is not too confusing. It'd be great if we could see the log for a delay-slot instruction other than nop, I'll have to fetch an example from the listing...

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.

Note that the software simulation log and the modelsim log need not be the same size; 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.

4.4 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 simulated by te 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.

4.5 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, this is no whim of mine.

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 because not everybody has access to Modelsim) this will have to be worked around.

5 Hardware demo

5.1 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.

This is just for convenience, so that you can launch some demo on hardware without installing the 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.

This demo has only been tested on a single dev board: terasIC's DE-1, with a Cyclone-II FPGA (EP2C20F484C7).

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. Suggested path is /syn/altera/¡project name¿.
2. Set target device as EP2C20F484C7. This choice determines speed grade and chip package.
3. 'Next' your way out of the new project wizard.
4. Add to the project all the vhdl files in /vhdl and /vhdl/demo. Select file c2sb_demo.vhdl as top.
5. Import pin constraints file (assignments-¿import assignments).
6. Create a clock constraint for signal clk (51 MHz or some other suitable speed which gives us some minimal slack).
7. In the device settings window, click "Device and pin options...".
8. Select tab "Dual-Purpose pins".
9. 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).
10. Select 'balanced' optimization.
11. Save the project and synthesize.
12. Make sure the clock constraint is met (timing analyzer report). There is a random element to the synthesis process, as you know, so it is possible that you need to repeat it if the first trial does not pass the constraints.
13. Program the FPGA from Quartus-2
14. 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 pusbutton 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 4.2 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.

5.2 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 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 modules that care at all about clock rate are the UART modules. They are 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.
Be aware that these uart modules have been used a lot in other projects but have seldom been tested with other than 50MHz clocks; they should work but you have been warned.

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.

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.

5.3 '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.

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.

Running Adventure on a computer built by myself is just something I wanted to do :) besides, it serves a useful purpose as a confidence builder.

6 Tools

6.1 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.
- 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.

The program includes usage help (a short description of the command line parameters). Between this and the BAT files I think you have enough information to use the program (and you can always contact me, of course).

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.

6.2 Conversion script bin2hdl.py

This Python script reads two binary files (for the data and code images, as explained above) 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 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 simple enough to be understandable even if you don't 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
"@data0@"	: Contents of RAM block for slice 0 (lsb) of data
...	
"@data3@"	: Contents of RAM block for slice 3 (msb) 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 lot 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 scrip 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.

7 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 4.

Target 'demo' will build a synthesizable demo; it will compile the sample sources and place the resulting object code in file '/vhdl/demo/code_rom_pkg.vhdl' (note that the 'sim' target has to do this too).

The build process will produce two or more binary files ('*.code' and '*.data', or '*.bin') that can be run on the software simulator:

```
slite hello.code hello.data
```

Plus a listing file (*.lst) handy for debugging.

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 6.2).