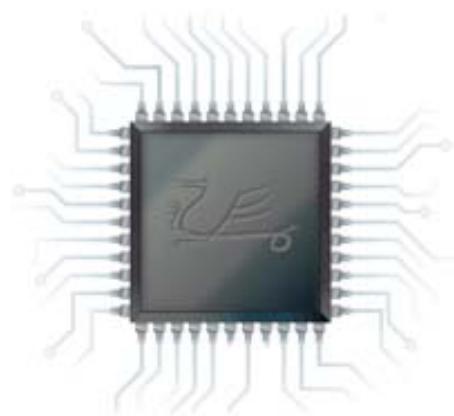


Reference Manual for the *rv32_cpu* RISC-V Verilog softcore CPU

(version 0.6 draft, 23nd Nov 2021)



Simon Southwell

September 2021

Copyright

Copyright © 2021 Simon Southwell (Wyvern Semiconductors)

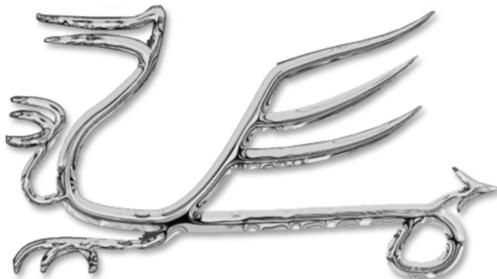
This document may not, in whole or part, be copied, photocopied, reproduced, translated, or reduced to any electronic medium or machine-readable form without prior written consent from the copyright holder.

Disclaimers

No warranties: the information provided in this document is “as is” without any express or implied warranty of any kind including warranties of accuracy, completeness, merchantability, non-infringement of intellectual property, or fitness for any particular purpose. In no event will the author be liable for any damages whatsoever (whether direct, indirect, special, incidental, or consequential, including, without limitation, damages for loss of profits, business interruption, or loss of information) arising out of the use of or inability to use the information provided in this document, even if the author has been advised of the possibility of such damages.

Simon Southwell (simon@anita-simulators.org.uk)

Cambridge, UK, September 2021



Introduction

This document details the design and use of the open-source `rv32i_cpu_core` RISC-V softcore. It is a Verilog based design targeted at FPGAs.

It is meant as an exercise in implementing a modern RISC based processor with the aim to demonstrate the operations of such a device, and some of the design problems encountered in constructing such a component. The code itself is also designed to be accessible and expandable, as an education tool and a starting point for customisation. As such, this document, as well as usage information, will give some details of the internal design of the source code sufficient to navigate and modify the code, as desired. At this time it's fixed at implementing the RV32I features, with the extensions being a possible future expansion.

The implementation trades off between performance, resource usage and clarity of design. It was designed to be an efficient implementation running at a usefully high clock frequency, with a modest FPFA resource requirement—but where clarity of operation would be unnecessarily obfuscated, a more practical route is taken. The design can be used without the need to understand the code, but if one desires to delve deeper and make modifications, then the intent is that the code is as easy to understand as possible, consistent with implementing the desired features, and design details are well documented in this manual, along with some of the design decisions that were made along the way.

This being said, an initial specification, when targeting the chosen example platform (a DE10-nano with a Cyclone V 5CSEBA6U23I7 FPGA) is for a 100MHz clock, less than 1000ALMs (~2600 LEs) and a pipelined Harvard architecture.

Along with the implementation is an example simulation test bench, targeting the ModelSim simulator from Mentor Graphics, and the synthesis setup to target the DE10-nano, using the Intel/Altera Quartus Prime tools. These two environments are documented within this manual as well.

***rv32i_cpu_core* Supported Features**

The overall features of this RV32I initial delivery are listed below

- All RV32I instructions implemented
 - Configurable for RV32E
 - Single HART
- Separate instruction and data memory interfaces (Harvard architecture)
- 5 deep pipeline architecture
- 1 cycle operations for all instructions except branch, jump and load
 - Regfile update bypass feedback employed
- Branch instructions take 1 cycle when not branching, 4 when branching
 - ‘never take’ branch prediction policy employed, with pipeline cancellation on branch
 - Jump instruction takes 4 cycles
- Load instructions take a minimum of 3 cycles, plus any additional wait states
- Register file configurable between register or RAM based
 - defaults to RAM based, using 2 × M10K RAM blocks
 - Register based costs approximately 700ALMs (~1900 LEs)
- Example simulation test bench provided
 - Targets ModelSim
- Example FPGA target platform using the terasic [DE10-nano](#) development board (employing the Intel [Cyclone V 5CSEBA6U23I7](#) FPGA).
 - Targeting 100MHz clock operation

- < 1000 ALMs (~2600 LEs) when also employing Zicsr extensions (RV32I implementation currently around 700 ALMs, ~1600 LEs).

The initial delivery instantiate the design in a core component, suitable for adding into the FPGA environment as a “Platform Designer” component. This core also instantiates some local FPGA RAM for the instruction memory (imem) and data memory (dmem). A future enhancement is to add a memory management unit for use of the, up to 1GB DRAM available on the DE10-nano.

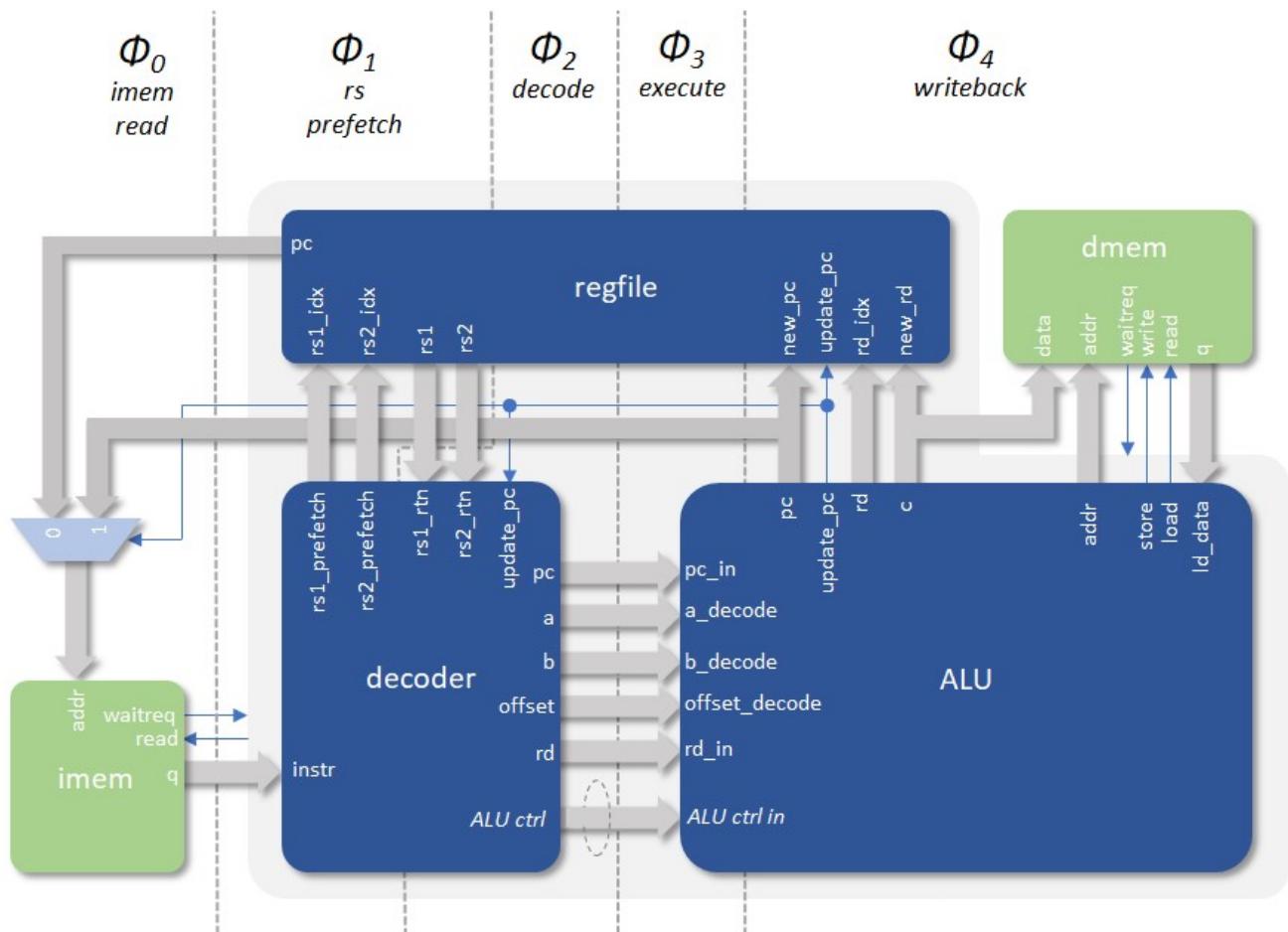
RV32I Core Implementation

The base core implements the RISC-V RV32I specification (see [1]). The idea is to implement a ‘classic’ RISC processor architecture, with multi-stage pipelining, to allow all instructions that have inputs and outputs only within the core, and do not alter the flow of the running program, to execute in a single cycle. That is to say, each instruction takes as many cycles as the pipeline is deep, but that the pipeline stages each have a partially executed instruction at that stage, so that the rate of processing is 1 cycle per instruction—with just an initial latency to fill it.

The instructions that do not have single cycle operations are jumps, branches (if taken), and memory load instructions. Memory store instructions will also have wait states when a memory sub-system is added but, for now, the example design that instantiates the core, uses FPGA memory blocks, which can be written in a single cycle.

Top Level

The implementation of the `rv32i_cpu_core` consists of three modules, which implement a 5-stage pipeline. The diagram below shows a simplified view of these components, which are instantiated within a top level.



The source code for the CPU can be found in the repository at [HDL/src](#), and consists of the following files

- `rv32i_cpu_core.v` : Top level CPU core
- `rv32i_regfile.v`: Register file, containing the CPUs registers and program counter (the first phase of instruction reads)
- `rv32i_decode.v` : Instruction decoder implementing the next two phases of the pipe, source register pre-fetch and instruction decode
- `rv32i_alu.v` : Arithmetic logic unit executing the instruction and register write-back (that last two phases)

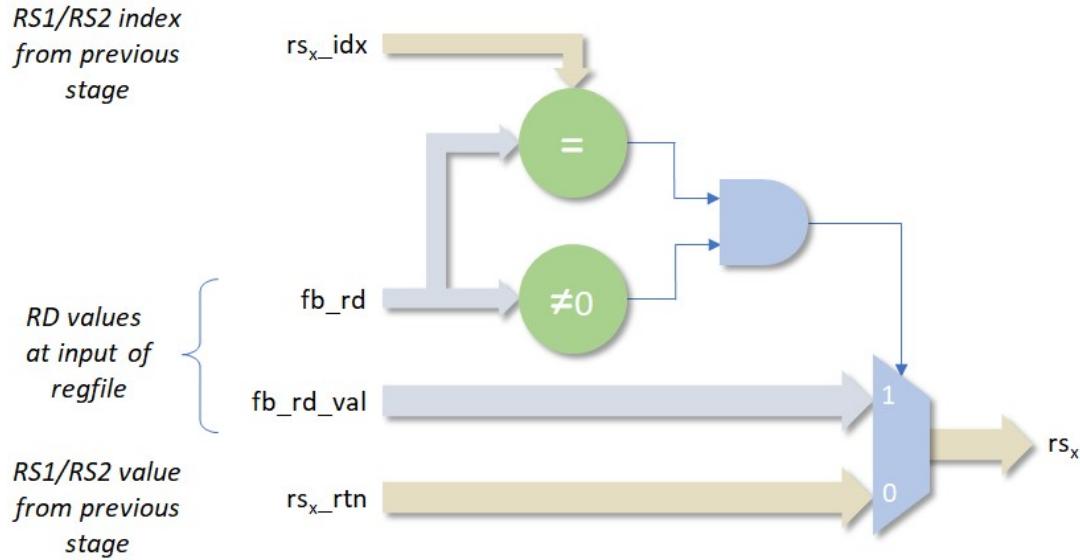
Pipeline Overview

The 5-stage pipeline allows for the efficient implementation of instructions, by breaking up the individual calculations required for the instructions into smaller pieces. It would be possible to execute an instruction in a single clock cycle, but that clock cycle would have to be long enough for all the calculations to be completed before allowing the next instruction to proceed. The aim of the pipeline is to divide the calculations into smaller ‘chunks’, with the output of one stage, feeding its results to the next stage. Since each stage’s calculations are smaller than the whole, the clock cycle speed may be increased, limited by the slowest of the stages. In an ideal design, the stages would all take the same time, and the pipeline would be balanced, and run as fast as is possible.

Register Hazards

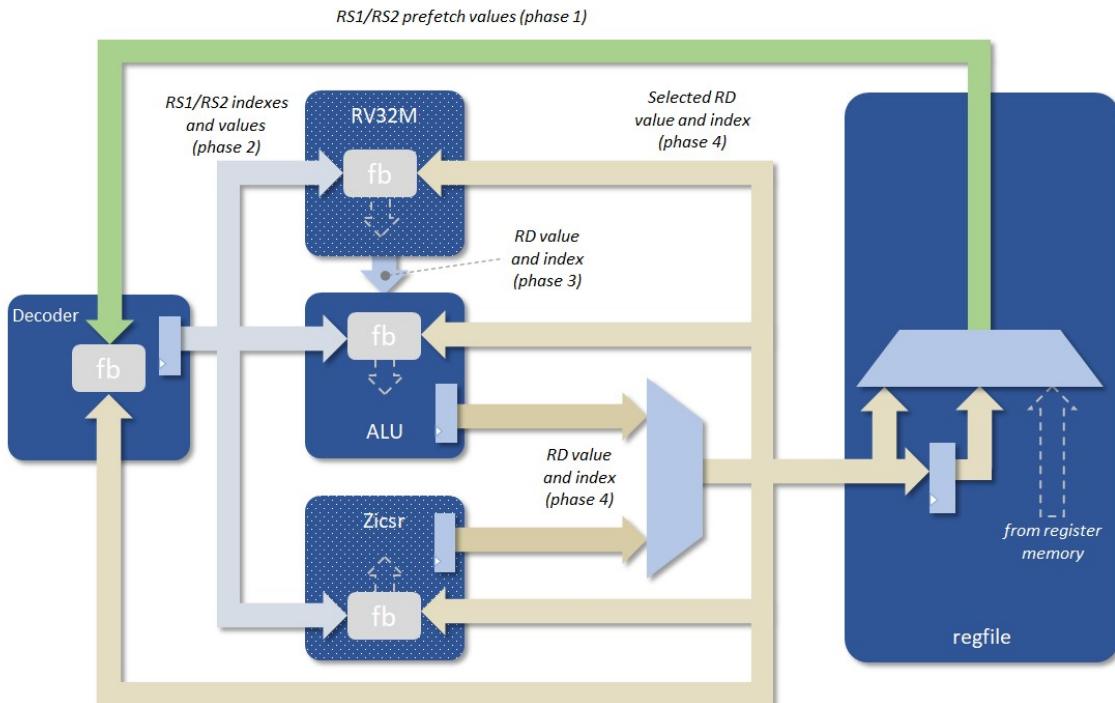
Having a pipeline architecture cause its own problems, however. If an instruction’s input relies on a result of a neighbouring instructions before it, but it takes several cycles before that result is available (in a register, say), then that instruction may get stale data if it does not wait for the result of the instruction in front of it to be available. This gets worse, the deeper the pipeline. Several options are available. Conceptually, the simplest system might be that the issuing of instructions is halted whilst an instruction is in the pipeline, and is only issued when the writeback phase is reached for the instruction in front. This, though, gets back to the single cycle operation, as only one instruction is active at a time. Overheads would probably make this worse than single cycle operations. A better approach might be to know which registers (or memory locations, possibly) are to be updated by the instructions in the pipeline, and issue a new instructions only if there are no hazards—that is, its source registers do not match any of the destination registers of instructions in the pipe. This would require keeping and monitoring destination registers (and, possibly, store addresses), at each stage.

The approach take in this design it to allow instructions to proceed regardless. The values of the registers indexed by the instruction’s RS1 and RS2 indexes (as appropriate), are accessed very early on, with the possibility of retrieving stale value. At each subsequent stage, except writeback, if a destination register is being written back, the stage will swap its source value with the writeback value, if its index matches that being written (and is not $x0$, which is never written). Since the ALU’s execute phase immediately precedes the writeback phase, there is opportunity to update the values at all the phases up to that point. The details of destination feedback will be discussed on the sections for each block but, in summary each block has a circuit similar to the diagram shown below (though possibly with different signal names).



Each module in the pipeline's path is feedback the same RD index and RD value that is being written to the regfile module. One slight variation is that the decoder is not fed back the register indexes for the prefetch RS1 and RS2 values, as it has these internally, decoded from the instruction and used to generate the prefetch register addressing. The index value is non-zero when a write to the regfile is active, so the circuit inspects for non-zero values, and compares the index to the index value of the RS registers it is processing (RS1 and RS2, with a circuit for each). If there is a match it selects the feedback value in preference to the RS values forwarded from the previous phase.

The diagram below shows the regfile bypass feedback paths.



In this design there are two sources of RD updates—one from the ALU and one from the Zicsr module (if enabled). It is the mux'ed version of this that is routed back to each of the other blocks, as well as forward to the regfile module. The feedback logic (shows as fb in the diagram), takes the RD values and indexes and compares with the module's source of the RS indexes, forwarding to the internal logic the appropriate value.

This system allows a pipelined architecture to get the correct RS value (i.e. the most up-to-date) without stalling the path. However, it creates critical paths in the design. In this design, this is from the registered RD output of the ALU, through the mux with the Zicsr RD output, back to the RS inputs of the ALU and through the feedback logic. Then it follows the ALUs logic path to the register for the RD (see the diagram in the *Arithmetic Logic Unit* section below, with the RS inputs are marked as a and b, and the RD value output is marked as c). For the de10-nano example platform, this is the critical path.

A similar long path exists from the registered ALU's RD output, through the mux into the regfile. Internally to the regfile module is a short pipeline of the value being written to allow for the latency of storing and the immediately accessing the data from the register file's memory. A similar feedback logic circuit is used to choose whether to use the incoming value being written, the one in the previous cycle, or that from memory. This adds a multiplexer to the path for selecting the returned prefetch RS values, with the fetch from memory likely to be the longest of the data paths. The prefetch values then have to go through the decoder's feedback logic before being processed by the decoder (see diagram in the *Decoder* section below) and ultimately registered as the RS outputs. In the example platform this isn't the critical path, but might easily become so with additional decoding (e.g. for more extension support) or on a different targeted platform.

The trade-off here is the advantage of issuing instructions once a cycle for maximising compute cycle efficiency, against the maximum speed at which the logic can be clocked. Given that pipeline is 5 phases deep, it is likely that the clocking would not be increased 4 or 5 fold by removing the bypass feedback functionality.

Branch Hazards

Another issue with a pipeline that allows instructions to always be issued is the problem of jumps and branches. If a jump is issued, or a branch that will be taken, the PC is updated from a linear increment to some arbitrary address to start fetching instructions from that location. When the jump is issued, though, it will be several cycles before the PC is updated at the writeback phase. In the meantime, the pipeline is fetching instructions beyond the jump, as if the jump has not happened. Those are not meant to be executed, and can alter state, with dire consequences.

In this design, the instructions proceed until an update to the PC is flagged at writeback. At this point, all the stages in the pipeline are 'cancelled'—effectively they are turned to nops (no-operation), and the pipeline continues. Since state is only updated at the writeback phase, by forcing the stages to nop, none of the erroneously fetched instructions have any effect, even though the pipeline proceeds. What this does do, though, is introduce cycles in the pipeline that do nothing. This is why, in the sections above, it has been stated that branches take 1 cycle if the branch is not taken but 4 cycles if taken. This is a function of the pipeline depth, and it will be seen in other implementations as well. Since the jump instructions are effectively a branch with an 'always taken' status, they are handled no differently, and the pipeline is 'stuffed' with the next instructions. The new PC value will not be known until the writeback stage, so early decode of jumps to stop issuing instructions until the new PC is available would just introduce additional logic to handle these. This might have an advantage in a power-conscious design, when one would not want to clock stages that do nothing, but this design does not attempt to be that sophisticated.

Stalling Mechanism

Another hazard with a pipeline architecture is for instructions that can cause “wait-state” or stalls. This boils down accesses to memory (and memory mapped I/O). In a general system, a memory management unit (MMU) might be employed which gives access to the DRAM (say) for a large storage space. Accessing that space can take tens of cycles (or more if I/O). A cache can be employed, with local fast access memory, with, perhaps, single cycle accesses. But, eventually, even the cache will need to be updated from main memory, and the access by the CPU will have to stall. This applies to both reading instructions from memory as well as load and stall instructions.

The current design treats instruction reads and load/store instructions differently. The example design using the rv32i_cpu_core, has local FPGA memory, with single cycles writes, and a delayed cycle read. The design, however, has the ability to handle arbitrary wait-states for both instruction reads and memory access instructions.

imem reads

For instruction reads the, design policy is to issue nop instructions for all cycles where instruction read data is not valid. This takes care of situations where the memory sub-system stalls on instruction fetches, and in the initial latency of fetching the first instructions.

As mentioned before, this is not perhaps the best policy from a power consumption point of view (if a design has features to support this), but, for this design, it is the simplest solution.

dmem loads

For memory accesses, the example design, with local memory will not stall on store instructions. On load instructions, however, the reading from memory takes a single cycle. In addition, the address to be read is not available until the execution phase output, as it is calculated during this phase. So wait states must be introduced to allows issue the read, wait for the return value, and then writeback to the destination register.

Conceptually, the easiest thing to do might be to stop issuing new instructions after a load (or stuff with nops), until the load writeback is reached. Decoding for a load instruction takes a couple of phases, and so a latency on stopping the PC increment exists, and so some sort of wind-back might be needed to pick up at the correct address when the load is complete. The problem with this approach is that it empties the pipeline behind the load instruction, adding additional cycles over-and-above the wait states.

In this design, the approach is to allow the pipeline to proceed behind the load instruction, until the execute phase, when the pipe is stalled. The writeback phase will hold with the memory read accesses for as long as necessary to get the returned data, and route it to the destination register. At this point the pipeline is allowed to proceed, and no added latency was introduced by stuffing the pipe with nop instructions.

The spreadsheet fragment below is an attempt to show what happens for a load access stall (and was used for the signalling design)

additional waitstates										
	ϕ_0	Id (addr+0)	addr+1	addr+2	addr+3	addr+3	...	addr+3	addr+4	addr+5
	imem read (pc)	ϕ_0	-	Id	I1	I2	I2	...	I2	I3
	rs prefetch (instr)	ϕ_1	-	Id	I1	I2	...	I2	I2	I4
	decode(instr_reg)	ϕ_2	-	-	Id	I1	...	I2	I2	I3
	execute (a, b)	ϕ_3	-	-	Id	I1	...	I1	I1	I2
	writeback(c, rd)	ϕ_4	-	-	-	Id	...	Id	Id	I1
dmem signals	read	0	0	0	0	1	...	1	0	-
	wait	0	0	0	0	1	...	0	0	-
ctrl signals	hold PC	0	0	0	1	1	1	0	0	-
	hold instr_reg, a+b, c, rd	0	0	0	0	1	1	1	0	-
	clear load	0	0	0	0	0	0	1	0	-

Normal (unstalled) pipeline state
dmem wait state
hold state
mem access data valid
Set low by 'clear load' (as held load state would otherwise set it, starting a new access)

The spreadsheet shows how the PC and phases proceed until stalled. A stall is ‘initiated’ when a dmem access is activated (read signal = 1), but the wait signal from the memory is also high. This is shown in read, and this may be the state for a number of cycles. Now it is known that a load instruction will have at least one wait state, and so early warning on stalling the PC can be affected when the load reaches the execution phase. This needs so as not to push instructions into the pipeline with no space left, and hence lose them. The ‘hold PC column shows this signal, asserted before the others.

The other stages in the pipeline need to stall on the first memory access wait state, and continue to stall until valid data is returned (when ‘wait’ is de-asserted—green). This is shown in the row below ‘hold PC’). Finally, the next cycle after the memory read valid data is where the writeback to a register is performed. Since the load instruction is still active, the dmem ‘read’ must be cleared in this phase (which would otherwise do because of the still active load instruction), so a new access is not started.

After this, the pipeline continues, with the instructions behind the load already to go. Thus only two additional cycles are added.

Parameters and Configuration

In general, the logic will work as intended without the need for configuration. However, the CPU core (rv32i_cpu_core), and the top level wrapper (core) have some parameters to alter their configuration, and these are documented below.

Parameters for the rv32i_cpu_core

The following lists the parameters available for the CPU core module, which are also routed to the core top level to make available to test benches and synthesis instantiations.

- RV32I_TRAP_VECTOR
- RV32I_RESET_VECTOR
- RV32I_REGFILE_USE_MEM
- RV32I_LOG2_REGFILE_ENTRIES (5 for RV32I, 4 for RV32E)
- RV32I_ENABLE_ECALL

The initial RV32I softcore implementation does not have CSR registers, as per the Zicsr extension specification, and so does not have an mvtec register defining the address for traps. The RV32I_TRAP_VECTOR parameter defines this address and has a default value of

0x00000004. Since there is no mvtec register, this cannot be changed programmatically. When the Zicsr extensions are added, this will become the default value of the mvtec register.

In a similar manner, the reset vector is defined using the RV32I_RESET_VECTOR parameter, and defaults to 0x00000000. Unlike the trap vector, there is no CSR register to change this, and the specifications leave this as an implementation choice. The two default values for the vectors, for this design, have been chosen to match the riscv-tests expectations.

The RV32I_REGFILE_USE_MEM parameter is used to select between a register based register file array when 0, or a RAM based register file array when 1. The default is 1 for RAM based. The choice is based on whether register or RAM is a limiting factor in a design using the core. In the current design, with a register based regfile, the FPGA resources used for the array is around that used for the rest of the core, doubling the requirements. If RAM is chosen, the two M-RAM blocks are used. For the example target platform this is two M10K blocks, with only 1024 bits of the 10240 bits used in each block. There are a few hundred blocks in the target FPGA, however, so the percentage usage is very low. If the core were targeted at an ASIC development, though, RAM may be more of an overhead than gates, so both implementations are available.

The RV32I_LOG2_REGFILE_ENTRIES parameter defines the ‘address’ width for the register file register array, which, thus, defines the number of entries. The default value is 5, giving the RV32I specification of 32 registers (x0 to x31). If an RV32E specification is required, the this may be altered to 4, for 16 registers (x0 to x15). When selecting a RAM based register file array, switching to an RV32E specification will save no RAM bits, and so is less useful.

The RV32I_ENABLE_ECALL parameter is purely for test purposes. It is there to allow the riscv-tests code to be run without the need for Zicsr extensions. These tests, as part of initialization set up some CSR registers and make tests on others. Normally, tests are made and will skip part of the initialisation, including executing an ecall instruction. With the parameter set to 0 (it defaults to 1), the ecall instruction is treated as a nop. With the implementation’s default values, and setting this parameter, the test code drops through to the main test code with sufficient initialisation to execute without issue. The intent is to remove this parameter once Zicsr extensions are added.

Additional Parameters for the top level Core

As well as the parameters for the CPU core, which are available at the top level core, some additional parameters are added, as listed below.

- RV32I_IMEM_ADDR_WIDTH
- RV32I_DMEM_ADDR_WIDTH
- RV32I_IMEM_INIT_FILE
- RV32I_DMEM_INIT_FILE
- RV32I_IMEM_SHADOW_WR

The RV32I_IMEM_ADDR_WIDTH and RV32I_DMEM_ADDR_WIDTH parameters specify the address widths of the internal instruction and data memories, and hence their size. By default they are 12 bits, for 4096 x 32 giving 16Kbytes for each RAM.

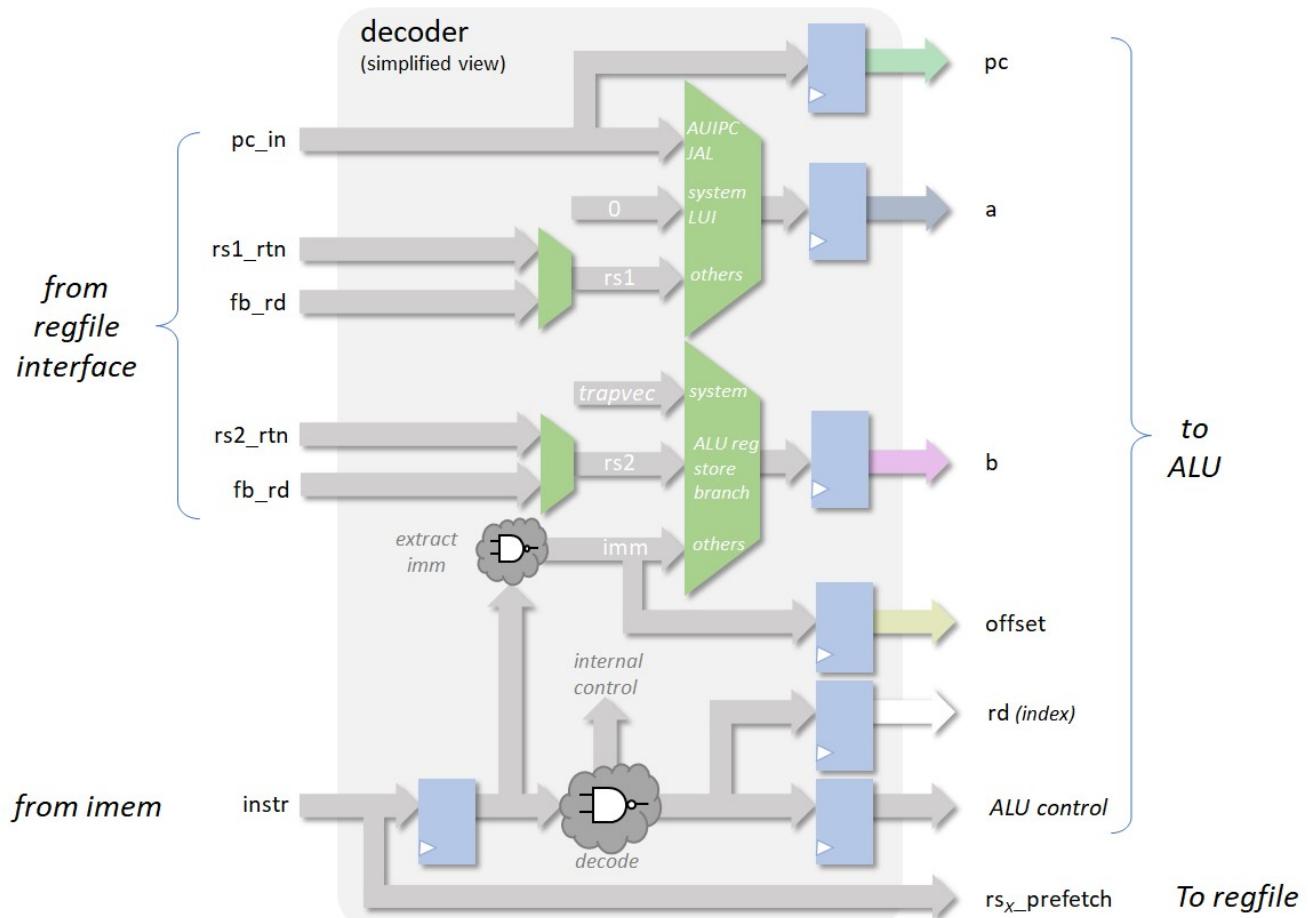
Memory initialisation files can be specified for the RAMs with the RV32I_IMEM_INIT_FILE and RV32I_DMEM_INIT_FILE parameters. The format for these are specific to the targeted example platform, and can either be in intel hex format (.hex), or in a ‘memory initialisation file’ (.mif) format. It is expected that, for different targets, the RAM modules will be replaced with the appropriate substitutes. The parameters can still be routed to these new blocks and specify an

initialisation file of the appropriate type, if required. The default value for these parameters is "UNUSED", indicating no initialisation file is specified for the RAM.

The RV32I_IMEM_SHADOW_WR parameter is meant for test purposes only. By default it is 0, with the logic's behaviour as normal. When set to 1, logic is enabled where all writes to the data memory, will also be written to the instruction memory. This was added specifically to allow the riscv-tests fence_i.s test to write, where stores to code locations are made for self-modifying purposes to test the fence.i instruction. Since IMEM and DMEM are separate memories, writes to code locations will be routed to DMEM. By enabling this feature, these writes are shadowed in the IMEM as well. It is expected that in an implementation this parameter is left at its default value. The example synthesis environment has modification of this parameter disabled.

Decoder

The decoder block, implemented in rv32i_decode.v, is responsible for processing raw instructions values and splitting out the instruction type and their various relevant fields of the instruction for sending to the arithmetic logic unit (ALU), as well as instigating source register prefetching. Thus it implements the pipeline stages 1 (rs prefetch) and 2 (decode). The diagram below shows a simplified view of the decoder module.



Instructions arrive from IMEM on the instr input, and the logic immediately routes the RS1 and RS2 fields to the register file block to start retrieving the values. This is regardless if what type the instruction will decode to be. The RISC-V specification has been constructed in such a way as to position the RS indexes in the same place for each instruction that specifies a source

register. By routing the bits in phase 1 to the register file block for all instructions, without further decode, the values become available for phase 2, if required. Otherwise the returned values are ignored. For a power sensitive design, additional decode in phase 1 might allow only fetching of data when relevant. Since the `instr` input is from memory, then it may arrive late in the cycle, and so needs to be only lightly processed where possible.

The instruction input is registered for the phase 1 processing. There are three main sections to the decode. Firstly, all possible opcode/function, register indexing and immediate value fields are extracted from the instruction. This is just wiring, and so is a convenience only.

The second stage is to use the extracted opcode and function fields to generate bits flagging the different instruction types, such as arithmetic, jump, memory accesses etc. With the operation bits, the various immediate fields are multiplexed to a single immediate value, based on the decode instruction type.

The final stage is to generate the outputs for routing to the ALU. The main outputs are an A and B value for inputs to the main ALU calculations. The values set on these are dependent on the instructions type. A can be the RS1 value, the PC or 0. The B value can be the RS2 value, the trap vector value or the selected immediate value. With appropriate A and B values the ALU calculations can take these common inputs and generate its output. It will still need to know the operation to perform on these value, and so the operation bit values are sent to the ALU (shown as ALU control on the diagram).

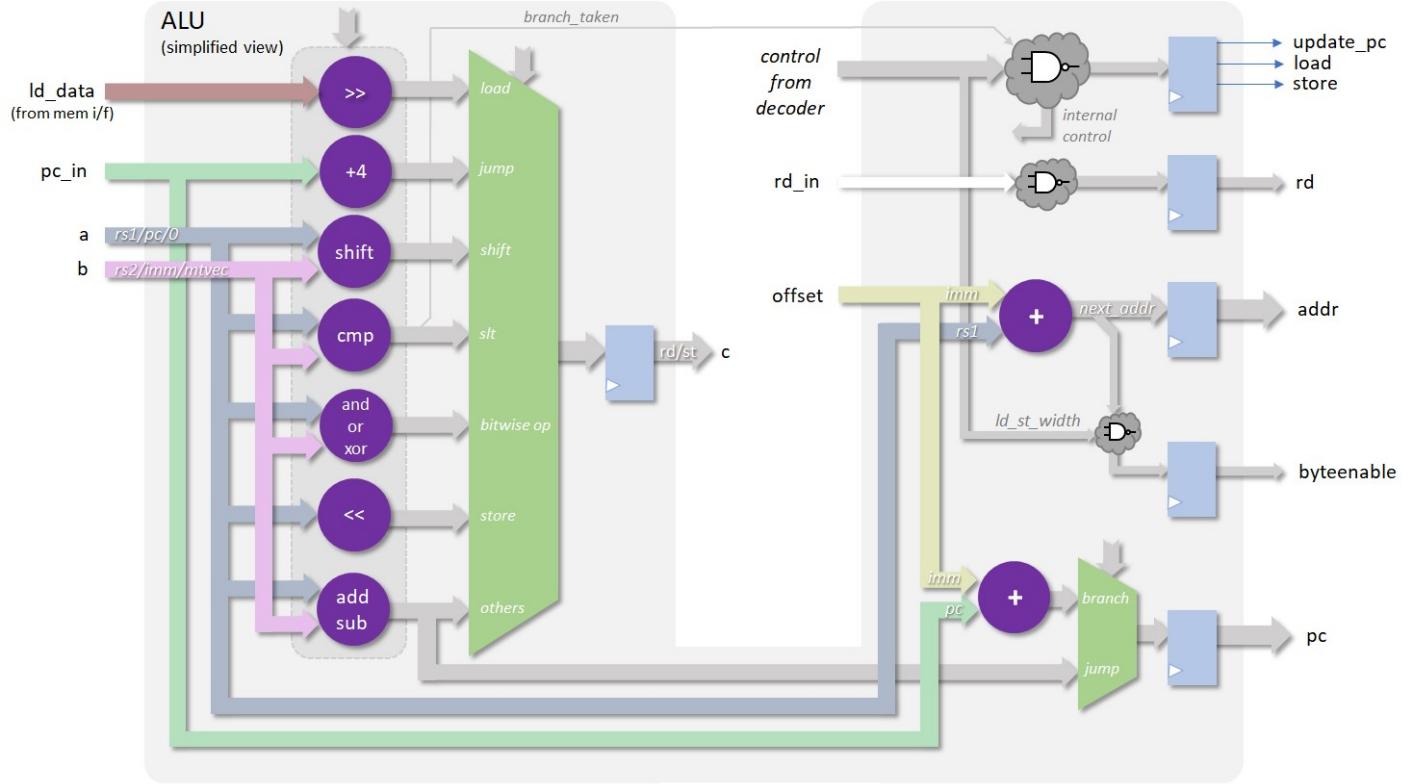
In addition to the A and B values, the source register indexes for the A and B values are sent to the ALU as well for instructions that use them (not shown in the diagram). This is purely for the pipeline destination (RD) feedback mechanism to allow for updating stale RS fetches (see *Pipeline Overview* section above). The decode block itself monitors for matching RS and RD indexes, and substitutes the register file returned values for the RD value update when a match occurs. This is not shown in the diagram. The RD index is also sent to the ALU to allow the ALU result to be routed to the appropriate destination register.

In addition to the A and B values, some instruction require calculations on the PC. For example, branch instructions must do a comparison of two registers, and A and B are used for these values, but a new PC is required if the branch is taken and a calculation is required on the PC value an offset value. The PC and the extracted offset are also routed from the decoder to the ALU.

Arithmetic Logic Unit

The ALU is where all the calculations for an instruction are executed, and where the results are written back to registers or memory, where applicable. New address values are also written back to the PC when necessary. Thus it implements phases 3 and 4 of the pipeline.

The diagram below shows a simplified view of the ALU logic.



At the heart of the design is the main execution calculations, which take the A and B values from the decoder to provide a result in a C output. For the most part, A and B are fed to the various possible operation types, and the appropriate result mux'ed to the C register, selected from the instruction type bit controls from the decoder. The categories of instruction in this mux are:

- load
- store
- jump
- shift
- comparison (not branches)
- bitwise operations
- arithmetic (add and subtract)

Of these categories all use the A and B inputs, except the following. The load instruction takes the returned data and aligns the data dependent on the data width and address. Similarly, the store instructions have the source data aligned to the appropriate byte lanes, ready for writing to memory. Jump instructions take the PC value for the instruction and adds 4. This is for the link address that will then be written to the indexed destination register. The actual new PC value is calculated separately and is discussed below. All the other types of instruction take the A and B values, as selected by the decoder, whether from a source register, and immediate value, or whatever. Not shown in the diagram is the writeback feedback mechanism where the A or B values from the decoder are overridden by the C output if their index matches the destination register index (and not 0). This is the phase 4 stage's 'stale rs data' mechanism,

As mentioned before, some of the instructions require additional calculations. For loads and stores a memory address is calculated. This takes the an address base value on the A input and an offset value sent from the decoder and extracted from the instruction's immediate value. Normally an immediate value appears on the B input, but the main calculation logic is

busy with either aligning memory read data (loads) or aligning write data (stores). An addr output sends the address to memory, and with byte enables calculated from the bottom bits.

For jumps and branches, a new PC value must be calculated, and this comes from the offset mentioned above and the PC value input to generate a new PC value.

All the c, addr and pc calculations proceed regardless of instruction type being executed. Whether they affect state, or not, depends on the control outputs. A writeback to a destination register only happens if the rd index output is non-zero. Also, an update_pc output flags to write the pc output to the PC register. A load and store output control initiating accesses to memory. All of these are functions of the instruction type control inputs.

Stalling

In the pipeline section, stalling was detailed from an overall perspective. The calculations on when to stall each stage of the pipeline are done at the top level. For the ALU, a single stall input controls stalling of the phase 3 and 4 stages. When stall is asserted the ALU will hold key state with its previous value. These include the following outputs

- addr
- load
- store
- rd
- update_rd
- pc
- update_pc

In addition, some internal state, ld_width, is also held which the byte enable outputs are calculated from.

Register File

The register file module is where all the main CPU state is implemented---the main integer registers x0 to x31, and the PC program counter. At its simplest, it is an array of 32 bit registers that can be written and read. As it holds the program counter, it implements phase 0 of the pipeline.

Registers versus RAM

In this implementation, the registers can be configured to be implemented as an array of logic registers or as memory blocks under the control of a parameter, RV32I_REGFILE_USE_MEM.

When registers are chosen, the an array of 32 bit registers is instantiated, which allows them to be written, selected with an rd_idx input and a new_rd value to be written. Two values need to be read simultaneously, for RS1 and RS2 values, and rs1_idx and rs2_idx index the registers to be read, with data returned on rs1 and rs2 outputs in the following cycle. Note that all three ports can be active simultaneously.

If memory blocks are used, to save on register usage, some problems arise. A two port memory block can have, at most two active accesses, where three are required. Secondly, write-through (where a location being read that's also being written sees the new value) is not possible without additional logic. These problems need to be solved if RAM blocks are to be used.

The multiple accesses issue is solved by instantiating two RAM blocks, which both get written simultaneously when a new RD value is being written, thus they always contain the same data.

When two reads are required, one is directed to the first RAM, the second to the other RAM, giving simultaneous read access. This, however, does not solve the write through problem.

Pipeline Feedback

When writing to the RAM, the inputs are latched, and then the value is written in the next cycle. If reading in the same cycle as the write, these inputs will also be latched, and data read in the next cycle, but this will be the old data, as the state is updating in the same cycle. So, the regfile module has a short queue of recently updated values to bypass the RAM in these cases.

If a read access matches the input write location, the value is simply passed back as the read value, just as for pipeline feedback in other stages. If the read index matches the previous write location, then a stored write value is routed back as the read value, to compensate for the cycles needed to update memory. In this manner, the correct value is returned, however close read and write accesses to the same location are.

This mechanism is employed whether registers or RAMs are used so as to keep the logic the same in both cases, thus simplifying it. It also allows the register implementation to register its returned value outputs, relieving timing for the decoder.

PC and Stalling

For the regfile module, stalling is just a matter of holding the PC state. Two PC values are kept in the register file block. The pc output is that used to fetch the instructions from IMEM. An additional value, last_pc, is the value of the PC in the last cycle (normally PC - 4). This value is fed to the decoder as the PC value of the instruction sent, since there is a delay in instruction fetches, and is

Memory Interfaces

The two memory interfaces, for the instructions (IMEM) and load/store data (DMEM) are modelled on the Avalon memory mapped master bus specification from Intel/Altera. This is because the example target platform uses the Intel/Altera Cyclone V FPGA, and this maps well to a solution based on these devices. Mapping to a more generic protocol, such as the open source Wishbone interface should be quite simple, with the main difference being a 'cycle' signal to indicate a bus access, and a write enable indicating a write (and not read), instead of separate read and write strobes.

The signalling is not fixed cycle but has a wait request input back to the master buses of the CPU core, to introduce wait states as discussed before for loads and instruction reads.

The current example design uses local memories, but the longer term aim is to have a memory sub-system with caches and access to main DRAM memory. The use of wait request inputs allows for this enhancement, with arbitrary wait states that may occur, without alteration of the CPU core logic.

Zicsr Extensions

The Zicsr extensions add the following features to the base implementation:

- clock counter
- Timer
 - 1µs resolution
 - Scales with system clock (on CLK_FREQ_MHZ parameter)
 - Timer interrupts
- Counters/timers mapped to user addresses as read-only registers
- Retired instruction counter
- External interrupts
- Vectored and non-vectored interrupts
- Support for software interrupts
- Support for memory mapped timer register access

Configuration Parameters

The Zicsr extensions can be enabled and disabled with the core's RV32_ZICSR_EN parameter. In addition, certain features of the CSR registers can be disabled to save on area.

The RV32_DISABLE_TIMER parameter excludes the real-time timer counter, when the Zicsr logic is included. When non-zero, the 64 bit counter, and its 'tick' circuitry, is removed. Since the processor still needs a real-time clock, the cycle counter is used instead, and it is this that is read when accessing the user timer CSR registers at 0xC01 and 0xC81. Since the timer can't stop, the CY bit of the mcountinhibit CSR register ([2], sec 3.1.13) has no effect when set if configured for no timer. For software to be able to schedule real-time events in this mode, the value of the CLK_FREQ_MHZ parameter is made available in the custom read-only CSR register at 0xFFFF, to indicate the rate at which the cycle counter is counting.

The RV32_DISABLE_INSTRET parameter excludes the retired instruction 64 bit counter logic when set. When this parameter is non-zero, reads of the low and high counter bits (at 0xC02 and 0xC82) return 0.

Additional Instructions

With the Zicsr logic enabled, the relevant addition instructions are available, as defined in [1], sec 9.1:

- csrrw
- csrrs
- csrrc
- csrrwi
- csrrsi
- csrrci
- mret

Supported Control and Status Registers

A subset of all possible machine level CSR registers has been chosen to balance the requirements of a real-time system or OS, and the need to minimize the resources used in the logic. See [2], Sec 3.1, 3.2 and [1], Sec 10.1. The supported machine level registers are:

- mstatus

- misa
- mie
- mtvec
- mcounteren
- mcountinhibit
- mscratch
- mepc
- mcause
- mtval
- mip
- mcycle, mcycleh
- minstret, minstreth (reads 0 when RV32_DISABLE_INSTRET non-zero)
- mvendor (fixed at 0 at present)
- marchid (fixed at 0 at present)
- mimpid (fixed at 0 at present)
- mhartid (fixed at 0)

In addition, the user view of the counters is also added:

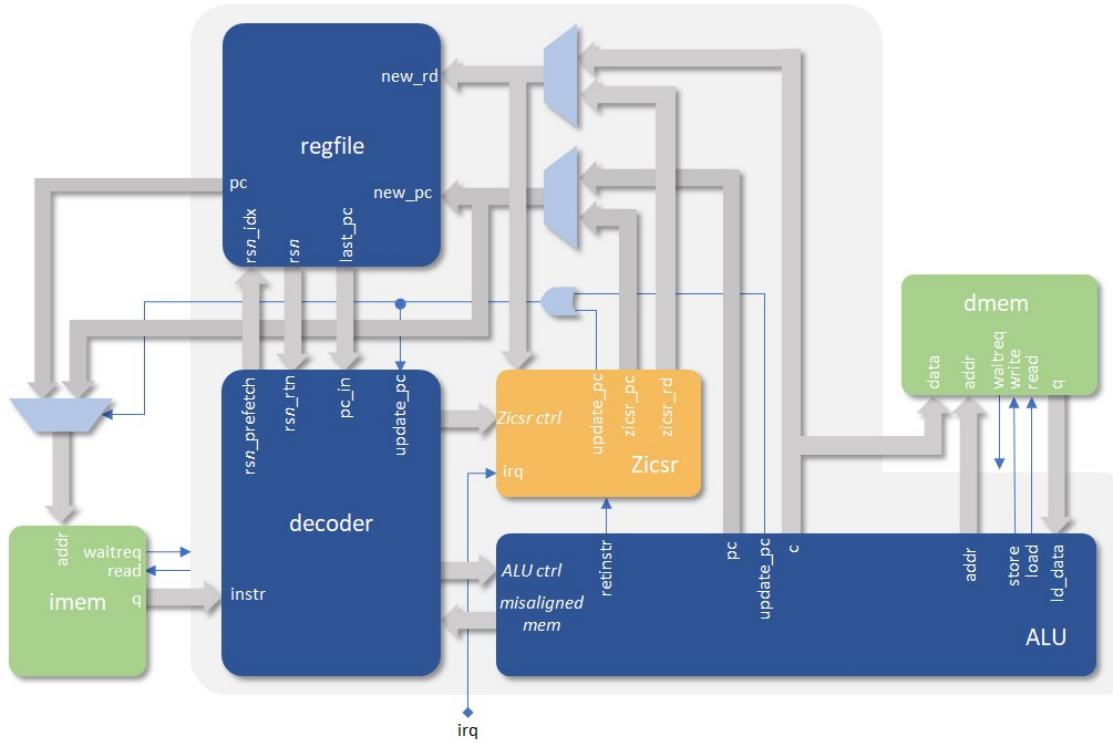
- ucycle, ucycleh
- utimer, utimerh (reads cycle counter when RV32_DISABLE_TIMER non-zero)
- uinstret, uinstreth (reads 0 when RV32_DISABLE_INSTRET non-zero)

Finally, a custom machine CSR register is added at 0xFFFF for reading the configured timer tick frequency (see [2], table 2.1), where it is set to 1 (MHz) when the timer is configured, but reads the CLK_FREQ_MHZ parameter when the timer is not configured, and the cycle counter is doubled-up and used for the timer instead.

Logic Design

The logic design for adding the Zicsr functionality, both in terms of the addition instructions and the CSR registers themselves, was done to balance making the logic configurable for adding or removing the resources, and making the design simple and efficient.

The diagram below is an updated top level diagram showing the blocks when Zicsr extensions are included, with the appropriate additional data path connections.



The design adds the decoding of instructions directly to the original base decoder unit, but this then farms out the processing of the instructions to a separate `rv32_zicsr` module. This not only executes the instructions, but also includes the implemented CSR registers, as described in the previous section.

The decode interface to the Zicsr block (marked as `Zicsr ctrl` in the diagram) is two separate paths. One flags when a CSRxxx or mret instruction is decoded, with parameters, whilst the other flags when synchronous exceptions happen, and includes exception type and PC address of instruction. The ALU provides memory access misalignment to the decoder to route as exception information, along with the other types of exceptions.

The Zicsr block is the 'ALU' block for the extensions, and can generate changes in the program counter. So it, like the RV32I ALU, generates destination register and PC update signaling. These are multiplexed with the ALU before sending to the register file block. To maintain some of the CSR register state, the Zicsr block needs to know when instructions are completed (retired), and a signal is sent from the ALU.

Lastly, an external IRQ input is synchronised and sent to the Zicsr block. Not shown on the diagram, but an external software interrupt input is also sent to the block. This is a pin on the `rv32_zicsr` module, but it is expected that this would be connected to a memory mapped register in the external memory mapped space.

Extensions to Decoder

The decoder block has additional logic added to support the Zicsr extensions, controlled with the `RV32_ZICSR_EN` parameter.

Firstly, the detection of illegal instructions is extended, with the `invalid_instr` signal set if a shift value of 32 or more is seen.

The system instruction decode signal (`system_instr`) is altered to remove the now defunct `RV32I_ENABLE_ECALL` parameter, disabling `ecall` instructions, as this is now fully supported. The a decode is added for `CSRxxx` instructions, with a flag to indicate whether immediate or not, and the `mret` instruction.

The `no_writeback` flag is updated to be set for system instructions only when the Zicsr extensions disabled. In addition, it is set when Zicsr instruction decoded.

Within the synchronous process, logic is added to set an exception pulse output (`exception`) when synchronous exceptions occur. These conditions are as follows:

- Misaligned IADDR: based on the input PC value not being 32 bit aligned
- Misaligned loads and store: uses input from the ALU
- Illegal instruction: based on the `instr_invalid` flag
- `ecall` or `ebreak` instruction

The exception pulse is accompanied by an exception type, an address (for load or store misalignments) and a PC of the instruction. For the load and store faults the PC is based on the last PC value as the exception was detected in the ALU, which is one phase behind the decoder. These exception signals go to the Zicsr module.

When a Zicsr instructions is decoded, the type of the instruction (read-write, read-set or read-clear) is output, along with the instruction destination register index. Similarly a signal output to when the `mret` instruction is decoded. These are also sent to the Zicsr block to take appropriate action. If the Zicsr instruction is an immediate instruction, then the `a` value output (usually for the ALU) is set to the RS1 index value signal, as these bits correspond to the instruction's immediate bits.

Finally, when there is no Zicsr extensions, updating the PC on system instructions is taken care of by the `rv32_zicsr` module, and so the `b` output, which for the base implementation is set to the trap vector parameter, does not select this when the Zicsr extensions are configured. Similarly for the RS index output (`a_rs_index`), which is zero on a system instruction when Zicsr is configured. This is so that ALU processes the system PC update when no Zicsr, but not when there is.

Extensions to ALU

The changes to the ALU are mainly passive, as the new functionality is implemented in the `rv32_zicsr` module. The main difference is the detection and output of load and store misalignment exceptions. These are sent out on a new interface (`misaligned_xxx`) and routed to the decoder, when merges these in with its own exception generation.

In addition, a signal is exported that is set for each instruction completed, and sent directly to the `rv32_zicsr` module for counting retired instructions.

rv32_zicsr Module

The functionality of this block is to implement the supported CSR register (as defined in the Supported Control and Status Registers section, above) and process the `CSRxxx` instructions that access and update them. In addition, it amalgamates exception, both synchronous from the decoder block interface, and asynchronous from the external interrupt, the software interrupt and the timer.

Within the block, there is logic to read and write the CSR registers, either from reads and writes via the `CSRxxx` instructions, or updates to the applicable registers when events occur. The actual indexing for the registers is done via an auto-generated module (`zicsr_rv32_regs`, implemented in `zicsr_rv32_regs_auto.v`, with a header `zicsr_auto.vh`), generated from a

JSON description. This has a read and a write port, with strobes, addresses (indexes) and data paths. The address/index decode logic is all generated internally. The implementation of the registers is configurable, via the JSON description, to be an internal or external register. For the CSR registers, if they are only updated via CSRxxx instructions, such as mstatus, then they are internally implemented as part of the logic file auto-generation, and there is no external port. Similarly, a register may be marked as a constant, such as mhartid, and implemented internally. If a register may be updated by both a CSRxxx instructions and external events, then it is implemented externally, and the auto-generated logic has a port with a pulse for that register, which pulses when written, and an output data port with write data. An input data port is also added for reading the external CSR register value. If a register is described in the JSON with bit fields, the input and output data ports will be separate for each described bit field. Since the CSRxxx instructions can be set or clear, as well as write, the rv32_zicsr logic uses the dual ports to read-modify-write the registers, so the clear and set functionality can be implemented on the register values.

The counters are implemented in the rv32_zicsr module, controlled via parameters, as described in the Configuration Parameters sub-section of this Zicsr Extension section. When the RT timer is enabled a microsecond counter is implemented to count the appropriate clock ticks. This is scaled dependent on the CLK_FREQ_MHZ parameter. The instruction retired counter is incremented on a signal from the ALU, but is removed if disabled, and will read 0.

The exceptions are processed, updating the PC based on type and mode and mtvec value, and updating the appropriate status, cause and interrupt registers, along with disabling further interrupts etc.

RV32M Extensions

The RV32M extensions add integer multiplication and division to the RV32I base specifications, as defined in [1], Chapter 7. In this design there is a separate module, `rv32_m`, that implements the functionality, and may be configured in various modes, depending on priorities, such as area, speed and availability of hardware DSP multiplication units (e.g. in an FPGA).

The logic adds the following 8 instructions, for signed and unsigned integer multiplication and division

- `mul`: signed/unsigned multiplication, returning lower 32 bit of result
- `mulh`: signed multiplication, returning upper 32 bits of result
- `mulhu`: unsigned multiplication, returning upper 32 bits of result
- `mulhsu`: signed × unsigned multiplication, returning upper 32 bits of result
- `div`: signed division, returning 32 bit quotient
- `divu`: unsigned division, returning 32 quotient
- `rem`: signed division, returning 32 bit remainder
- `remu`: unsigned division, returning 32 bit remainder

Configuration Parameters

The `rv32_m` module has three parameters, available at the `rv32i_cpu_core` top level to configure how it operates, as listed below:

- `RV32_M_EN`: default 1 (i.e. enabled)
- `RV32M_FIXED_TIMING`: default 1 (i.e. no optimization logic—always performs fixed timed calculation)
- `RV32M_MUL_INFERRRED`: default 0 (i.e. does not infer DSP multiplication units, and uses shift logic).

The `RV32_M_EN` parameter is used to control whether the RV32M functionality is instantiated or not. By default it is, but if an implementation does not need the features, then area can be saved by disabling the module. If the module is enabled, then performance and area can be traded off using two associated parameters.

Firstly the `RV32M_FIXED_TIMING` parameter, when non-zero (the default), will always perform a full calculation (multiply, divide or remainder) for each relevant instruction, fixing the calculation time for each instruction, as defined in the *Performance* section below. This is the cheapest option, in terms of logic resources, but, since each instruction only returns 32 bits, to get all 64 bits for a multiplication, or the division and remainder of a divide, two instructions must be executed. With the parameter non-zero the calculation will be done twice for each instruction pair (e.g. `mul`, `mulh` or `div`, `rem`). When the parameter is zero, optimization logic is instantiated that monitors for the same operation (i.e. one of the multiplication instructions or one of the division/remainder instructions) with the same inputs as the last successful calculation. If there is a match, the stored result is accessed without the need for running the entire calculation. Therefore, if a `mul` instruction is followed by a `mulh`, or a `div` is followed by a `rem`, with the same input values, then only the first instruction takes the calculation penalty, whilst the second returns in just two cycles. This is irrespective of the RS1 and RS2 registers used to source the inputs, or the RD register indexed to store the result.

Secondly, the `RV32M_MUL_INFERRRED` parameter is used to control how the multiplication logic operates. By default this is set to 0, meaning that the multiplication shift circuit is instantiated, incurring the penalty in logic resource, and in the calculation execution time. The benefit is that the logic is vanilla RTL and can be synthesized to any target. If the parameter is non-zero, then

the logic instantiated is a straight forward $C \leq A * B$; Verilog statement. This mode is intended, primarily, for FPGA targets that have hardware DSP elements with multipliers—usually with optional add or accumulate features. In this case the statement will infer one or more of these elements (depending on the underlying size of the hardware multipliers). For FPGA targets with DSP elements this saves logic resources and also speeds up the multiplication calculation time. Since not all targets (ASIC or FPGA) have DSP elements, the configuration allows the appropriate logic to be used.

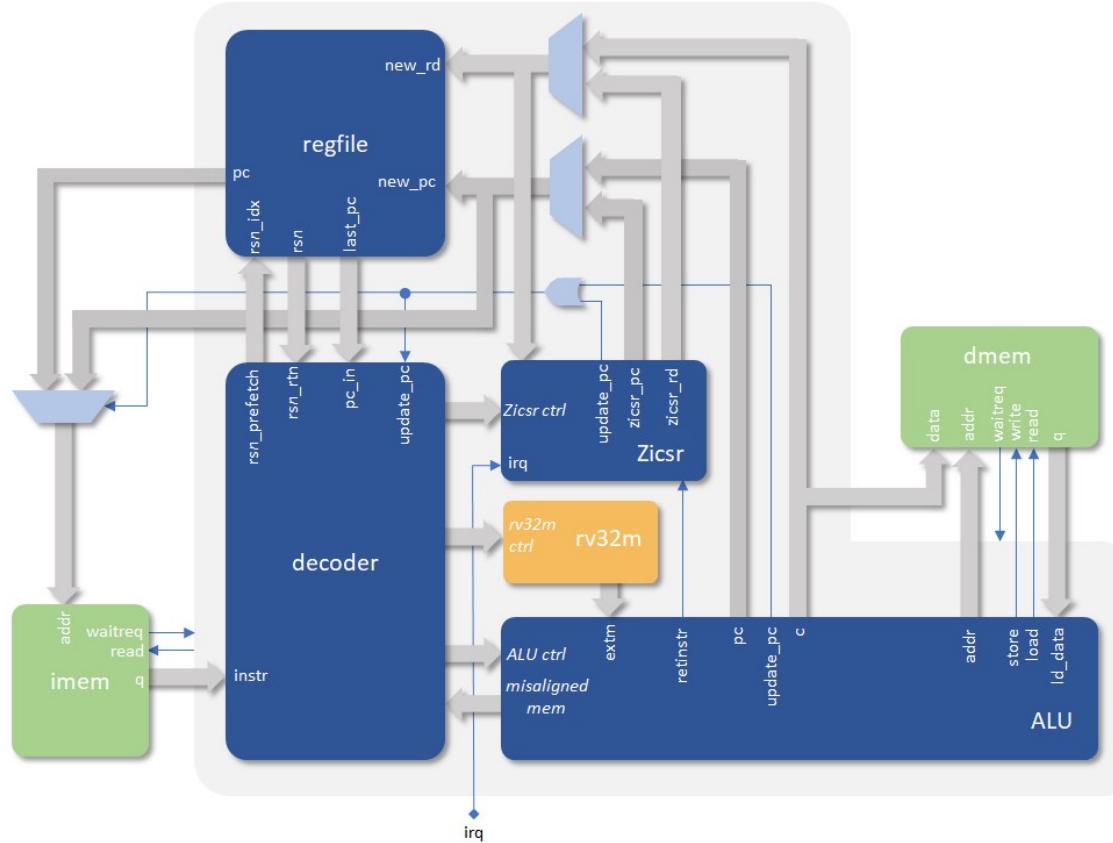
Performance

Below is shown the cycle count for the executions of the instruction types, as a function of the parameter settings, matching inputs to last calculation and (for the case of multiplication) the inference of hardware DSP elements

- Multiplication with RV32M_MUL_INFERRRED = 0: **34 cycles**
- Multiplication with RV32M_MUL_INFERRRED > 0: **3 cycles**
- Multiplication with RV32_FIXED_TIMING = 0: as above if no match, or **2 cycles** if match
- Division and remainder with RV32_FIXED_TIMING = 0: **34 cycles** if no match, or **2 cycles** if match
- Exceptions (divide-by-zero or divide signed overflow): **2 cycles**

Logic Design

The logic design for the RV32M extensions houses the execution (phase 3) logic in a separate `rv32_m` module. The decoder is extended to do initial decode of the added instructions, and the result is routed to the ALU. This is shown in the updated block diagram below:



The decoder does a partial decode and flags to the module of a dedicated interface. The calculation is done as a phase 3 execution, and the result routed to the ALU to send out as the phase 4 writeback.

Extensions to Decoder

As mentioned above, the execution of multiply and divide instructions is housed in the `rv32_m` module, but the decoder is extended, when the `RV32_M_EN` parameter is non-zero, to decode these instructions. A new interface is added, prefixed with `extm_`, which flags to the `rv32_m` module when a RV32M instruction is decoded (`extm_instr`), which of the eight instructions it is (`extm_funct`) and the RD destination register the result is destined for (`extm_rd`). These latter two value are simply extracted from already decoded and registered values, when the values occupy the same location in the instruction.

For the `extm_instr` flag, and internal signal is added, `mul_div_instr`, which decode the `instr_reg` value to uniquely decode the RV32M group of instructions, if `RV32_M_EN` is non-zero, and this is then registered as `extm_instr`. The `mul_div_instr` signal is also used in the decoder to ensure that the ALU doesn't perform any calculation, by setting the `no_writeback` signal. The `rv32_m` module has the decoder's A and B outputs routed to it as its data inputs, along with their matching register indexes (for bypass feedback purposes). By default, the A input selects RS2, which is what is required, but the B default selection is the immediate value, but requires to be RS1. So the `mul_div_instr` is also used to update the b and `b_rs_idx` decoder outputs to be the values for RS2.

Note that the decoding to the individual multiply or divide instructions is done in the `rv32_m` module and not in the decoder. This minimizes the changes required to the common decoder, and the ease of including or excluding the relevant logic on the `RV32_M_EN` parameter.

Extensions to ALU

Some small extensions to the ALU were needed for the RV32M functionality. As mentioned before, in order not to introduce an additional multiplexing on the RD updating to the register file (as was done for Zicsr), which is a critical path on the feedback bypassing for all modules, the outputs of the `rv32_m` are routed to the ALU to re-use its c and rd outputs. This still adds a selection to the critical path, namely from rd updates to the ALU result (c) output, but experimentation showed that this gave the most favourable result.

A new interface (with `extm_` prefixes on the signals) is added with a flag indicating that the `rv32_m` module has an active result (`extm_update_rd`), the RD destination register index (`extm_rd_idx`), and the value being written (`extm_rd_val`). When the update signal is flagged, the internal `next_rd` signal is mux'd to the `extm_rd_idx` value. This takes priority over the stall signal, as this is set when the `rv32_m` module is active to prevent the ALU advancing whilst the multi-cycle calculations are proceeding. Then, in the synchronous process, the ALU's c output is updated (again as a priority) with the `extm_rd_val` if the `extm_update_rd` signal is active.

Note that the ALU does not have any parameters associated with the RV32M functionality. The `extm_update_rd` and `extm_rd` signals in the `rv32i_cpu_core` are tied off to 0 when `RV32_M_EN` is zero. In this case, the synthesis process will optimize away the logic in the ALU associated with the multiplication and division.

rv32_m Module

The `rv32_m` module has logic multiplication and division functionality, based on shift registers doing shift and add (for multiplication) and long division. These take a fixed number of cycles (see *Performance* section above), with the calculations always being done on unsigned values, and a cycle used at either end to convert the inputs to unsigned values and the result back to

signed value, as appropriate. This is not necessarily the most efficient way of doing this, and the two sign conversion cycles might be saved with more complex logic. The decision was made, however, to keep with the conversion implementation for the sake of clarity and simplicity in implementation. This aligns the core with similar FPGA softcores, such as the LatticeMico32 [3] from Lattice Semiconductors, which takes 34 cycles for its division instructions, or the NIOS II [4], from Intel, which takes 35 cycles.

When converting the division result sign the rule followed is that the dividend and the remainder have the same sign, whilst the result (quotient) is negative if the sign of the inputs (dividend and divisor) are different.

For divide-by-zero and overflow conditions, the logic detects these at the start of the calculation, and sets the outputs according to Table 7.1 in section 7.2 of [1]. In these cases the calculation is terminated, with the result returned after 2 cycles.

Verification

The verification strategy of the design follows that of the instruction set simulator. The [riscv-tests](#) supplied by RISC-V International, which are categorised into tests for the various base specifications and extensions. They are self-checking assembly code, and thus require a minimum amount of functionality in order to run correctly. These tests, in turn, rely on the [riscv-test-env](#) repository.

Some ‘turn-on’ test code was written in order to get to a useful standard, but this was not self-checking and is superseded by the riscv-tests code, and so is not part of the repository.

An example simulation test environment is provided to run the riscv-tests, as well as a synthesis environment to build for the DE10-nano board, with the ability to run these same tests. The following sections describe these environments and how they are used.

Simulation

The CPU core simulation environment can be found in the repository in the folder HDL/test. This contains all the files to compile and run a simulation for the core.

Compiling and Running the Simulation

Prerequisites

A provided makefile allows for compilation and execution of the example simulation tests. The current support for simulation is only for ModelSim at this time, and has some prerequisites before being able to be run. The following tools are expected to be installed

- ModelSim Intel FPGA Starter Edition. Version used is 10.5b, bundled with Quartus Prime 18.1 Lite Edition. The MODEL_TECH environment variable must be set to point to the simulator binaries directory
- MinGW and Msys 1.0: For the tool chain and for the utility programs (e.g. make). The PATH environment variable is expected to include paths to these two tools’ executables

The versions mentioned above are those that have been used to verify the design, but different versions will likely work just as well. If MODEL_TECH environment variable is set correctly, and MinGW and Msys bin folders added to the PATH, then the makefile should just use whichever versions are installed.

Compiling test code

Before running a simulation a test file must be created to load into the instruction memory. By default the memory is looking for a memory initialisation file named test.mif. This will be generated from one of the tests in the [riscv-tests](#) repository. The riscv-tests repository is expected to be checked out in the same directory as this repository, as well as the accompanying [riscv-test-env](#) repository, which riscv-tests relies on.

A make file (makefile.test) is provided to compile a test to an executable and then convert this to a memory initialisation file (.mif). There is a program supplied for the ELF to .mif conversion, which is in the test folder and called elf2vmif.cpp. The make file takes care of building this if it hasn’t been built already.

The make file itself is executed with some arguments to override variables to select a test to be compiled and the sub-directory in riscv-tests/isa where it is located. For example, from a command window, and in the test folder, one can execute, say:

```
make -f makefile.test SUBDIR=rv32ui FNAME=addi.S
```

This will compile the addi.S test to a file in the test folder called addi.exe. It then converts this to a file test.mif. If a simulation is now run, the compiled code will be loaded into the instruction memory before the simulation is run.

Running a Simulation

The main make file (makefile) is used to compile and then run a simulation, either as a batch simulation or a GUI simulation. Typing `make help` displays the following usage message:

```
make          Build code without running simulation
make compile    Build HDL code without running simulation (same as make)
make run/sim     Build and run batch simulation
make rungui/gui   Build and run GUI simulation
make runlog/log    Build and run batch simulation with signal logging
make waves        Run wave view in ModelSim (to view runlog signals)
make regression   Run the regression test
make clean         clean previous build artefacts
make help          Display this message
```

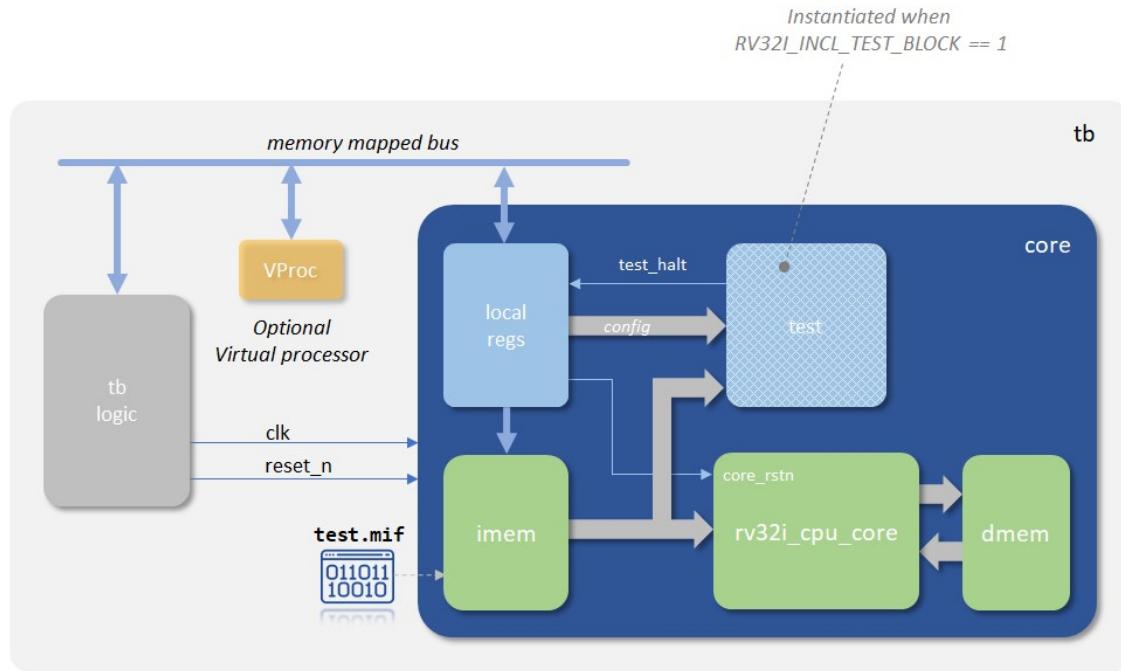
Giving no arguments compiles the Verilog code. The `compile` does the same thing. The various execution modes are for batch, GUI and batch with logging. The batch run with logging assumes there is a `wave.do` file (generated from the waveform viewer) that it processes into a `batch.do` with signal logging commands. When run, a `vsim.wlf` file is produced by the simulator, just as it would when running in GUI mode, with data for the logged signals. The `make wave` command can then be used to display the logged data without running a new simulation. Note that the `batch.do` generation is fairly simple at this point, and if the `wave.do` is constructed with complex mappings the translation may not work. The `make regression` command runs all the regression tests, as per the `run32i_tests.bat` script. The `make clean` command deletes all the artefacts from previous builds to ensure the next compilation starts from a fresh state.

The `riscv-tests` exit with the `gp` register containing a test number from bit 1 upwards, and the bottom bit set. If the test number is 0, then the test has passed (if the bottom bit is set). Otherwise the test number is the number of a test that failed (it halts after the first failure). The test bench, at the end of the test inspects the value and prints a message for pass or fail, with some information about the state.

All of the `rv32ui` tests can be run as a regression test using the `run32i_tests.bat` batch file. This goes through all of the RV32I unprivileged instruction tests, compiling them, running a batch simulation, and extracting the PASS/FAIL messages in a test log (`test.log`), which it prints to the screen at the end for inspection for failures.

Test Bench

The test bench (`tb.v`) is a straight forward Verilog module, in a traditional format. The top level core is instantiated as the UUT, and a clock and reset are generated locally and sent to the core. Not much else needs to be connected to the core, and the main other inputs to the design are the parameters. The test bench module has parameters, some of which are those of the instantiated top level core, allowing them to be overridden when running a test. The diagram below shows the general structure of the test bench and instantiated core.



The core has an optionally instantiated test block for monitoring the `imem` read bus, and halting the core when certain conditions are met, as defined by the control register bits in the local registers. The test bench sets the `RV32I_INCL_TEST_BLOCK` parameter to 1, so that this block is present during simulations.

The core has an Avalon memory mapped slave bus for reads and writes to registers/memory from an external source. In normal usage, this is mainly for loading a program to instruction memory, and then taking the core out of reset to execute the program. In simulation, the program is loaded from an initialisation file to save wasting time writing to memory during simulation. Since it is instantiated, the core's test block is configured using other registers, to select halt conditions (on unimplemented instruction reads, `ecall` instruction and/or specific address reads). When a halt condition occurs, the core will set a signal which can be read via the registers. The registers in the core are defined below:

core	This block is the core registers					
Register Name	Word Addr Offset	Field	Bits	Type	Reset	Description
CSR_CORE_CONTROL	0x0		4			Core control register
		CLR_HALT	0	w0	0	Clear test halt status
		HALT_ON_ADDR	1	w	0	Enable/disable halt on specific address
		HALT_ON_UNIMP	2	w	0	Halt on unimplemented instruction
		HALT_ON_ECALL	3	w	0	Halt on <code>ecall</code> instruction
CSR_CORE_STATUS	0x1		2			Status of core
		HALTED	0	r	0	Halted status
		RESET	1	r	0	Reset status
CSR_CORE_HALT_ADDR	0x2		32	w	0x00000040	Halt address, if <code>halt_on_addr</code> active
CSR_CORE_GP	0x3		32	r	0	gp register value
CSR_CORE_TEST_TIMER_LO	0x4		32	wp	0	Test register to write timer low 32 bits
CSR_CORE_TEST_TIMER_HI	0x5		32	wp	0	Test register to write timer high 32 bits
CSR_CORE_TEST_TIME_CMP_LO	0x6		32	wp	0	Test register to write timer comparator low 32 bits
CSR_CORE_TEST_TIME_CMP_HI	0x7		32	wp	0	Test register to write timer comparator high 32 bits
CSR_CORE_TEST_EXT_SW_INTERRUPT	0x8		1	w	0	Test register generate external software interrupt

Some probes into the core are made from the top level to extract signals for control of execution. In particular, the GP register (x3) and PC register, along with the test_halt signal from the test block. The test bench logic monitors these signals and determines when a halt condition occurs. It is when the test bench detects a halt condition that test status is inspected and a PASS/FAIL message is displayed.

In addition, the software interrupt pin can be controlled via a register, and the timer update port of the rv32i_cpu_core is hooked up to registers to allow its update, along with the comparator, for test purposes.

Core Parameters

The core has various parameters which configure the logic. In the test bench, these are configured in such a way as to allow the riscv-tests to be run. The table below shows the parameters, their default values and the values set by the test bench:

Core parameter name	rv32i_cpu_core?	Default	Test Bench	TB parameter name
CLK_FREQ_MHZ	yes	100	100	CLK_FREQ_MHZ
RV32I_RESET_VECTOR	yes	32'h00000000	32'h00000000	RESET_ADDR
RV32I_TRAP_VECTOR	yes	32'h00000004	32'h00000004	TRAP_ADDR
RV32I_LOG2_REGFILE_ENTRIES	yes	5	5	LOG2_REGFILE_ENTRIES
RV32I_REGFILE_USE_MEM	yes	1	1	REGFILE_USE_MEM
RV32_ZICSR_EN	yes	1	1	ZICSR_EN
RV32_DISABLE_TIMER	yes	0	0	-
RV32_DISABLE_INSTRET	yes	0	0	-
RV32_M_EN	yes	1	1	M_EN
RV32M_FIXED_TIMING	yes	1	1	M_FIXED_TIMING
RV32M_MUL_INFERRRED	yes	1	1	M_MUL_INFERRRED
RV32I_IMEM_ADDR_WIDTH	no	16	16	-
RV32I_DMEM_ADDR_WIDTH	no	16	16	-
RV32I_IMEM_INIT_FILE	no	"UNUSED"	"test.mif"	-
RV32I_DMEM_INIT_FILE	no	"UNUSED"	"UNUSED"	-
RV32I_ENABLE_ECALL	no	1	0	-
RV32I_IMEM_SHADOW_WR	no	0	1	IMEM_SHADOW_ER
RV32I_INCL_TEST_BLOCK	no	0	1	INCL_TEST_BLOCK

The table shows which parameters are also part of the rv32i_cpu_core parameters, and which configure the top level core only. Even though the test bench will give default values to all the core's parameters, as shown, only some are exposed at the test bench top level. These are shown in the right most column with the name used for the equivalent test bench. Thus, only these parameters can be altered from the command line when simulating.

Results

The simulations were run for each of the riscv-tests under the rv32ui folder—that is, the unprivileged instructions for RV32I base specification. The table below shows the tests that have been run and their status.

sub-test folder	test	status
rv32ui	simple.S	Passed
	add.S	Passed
	addi.S	Passed
	and.S	Passed
	andi.S	Passed
	auipc.S	Passed
	beq.S	Passed
	bge.S	Passed
	bgeu.S	Passed
	blt.S	Passed
	bltu.S	Passed
	bne.S	Passed
	jal.S	Passed
	jalr.S	Passed
	lb.S	Passed
	lbu.S	Passed
	lh.S	Passed
	luh.S	Passed
	lui.S	Passed
	lw.S	Passed
	or.S	Passed
	ori.S	Passed
	sb.S	Passed
	sh.S	Passed
	sll.S	Passed
	slli.S	Passed
	slt.S	Passed
	slti.S	Passed
	sltiu.S	Passed
	sltlu.S	Passed
	sra.S	Passed
	srai.S	Passed
	srl.S	Passed
	srlu.S	Passed
	sub.S	Passed
	sw.S	Passed
	xor.S	Passed
	xori.S	Passed

These are all the relevant tests for the current RV32I implementation. When the extension specifications are added, the additional tests will be run, as for the instruction set simulator.

Tests for Zicsr extensions

The simulations were run for each of the riscv-tests under the rv32mi folder—that is, the machine instructions for RV32I specification. The table below shows the tests that have been run and their status.

sub-test folder	test	status
rv32mi	breakpoint.S	Not yet supported [†]
	csr.S	Passed
	illegal.S	Passed
	ma_addr.S	Passed
	ma_fetch.S	Passed

	mcsr.S	Passed
	sbreak.S	Passed
	scall.S	Passed
	shamt.S	Passed

NOTES:

[†] Test relies on debug CSR registers to be implemented, which they currently aren't in this model.

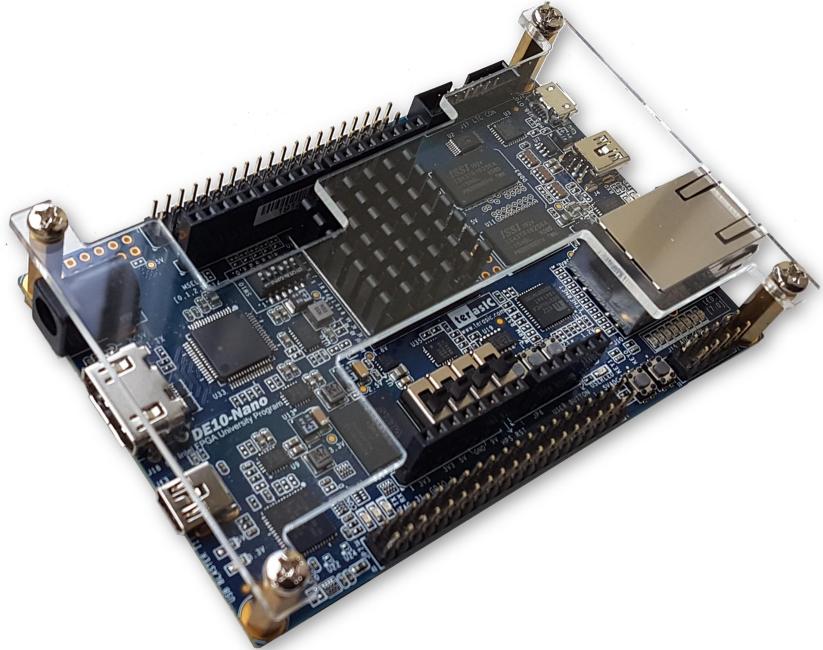
Tests for RV32M extensions

The simulations were run for each of the riscv-tests under the rv32um folder—that is, the machine instructions for RV32M extensions specification. The table below shows the tests that have been run and their status.

sub-test folder	test	status
rv32um	mul.S	Passed
	mulh.S	Passed
	mulhu.S	Passed
	mulhsu.S	Passed
	div.S	Passed
	divu.S	Passed
	rem.S	Passed
	remu.S	Passed

Synthesis

An example target platform environment is provided, targeting the DE10-nano board (shown below). Files are provided for the top level core to be instantiated in a Platform Designer infrastructure (.qsys file) and the setup and constraints for a Cyclone V BA6 device synthesis, suitable for the chosen platform. These files can be found in the HDL/de10-nano/build folder.

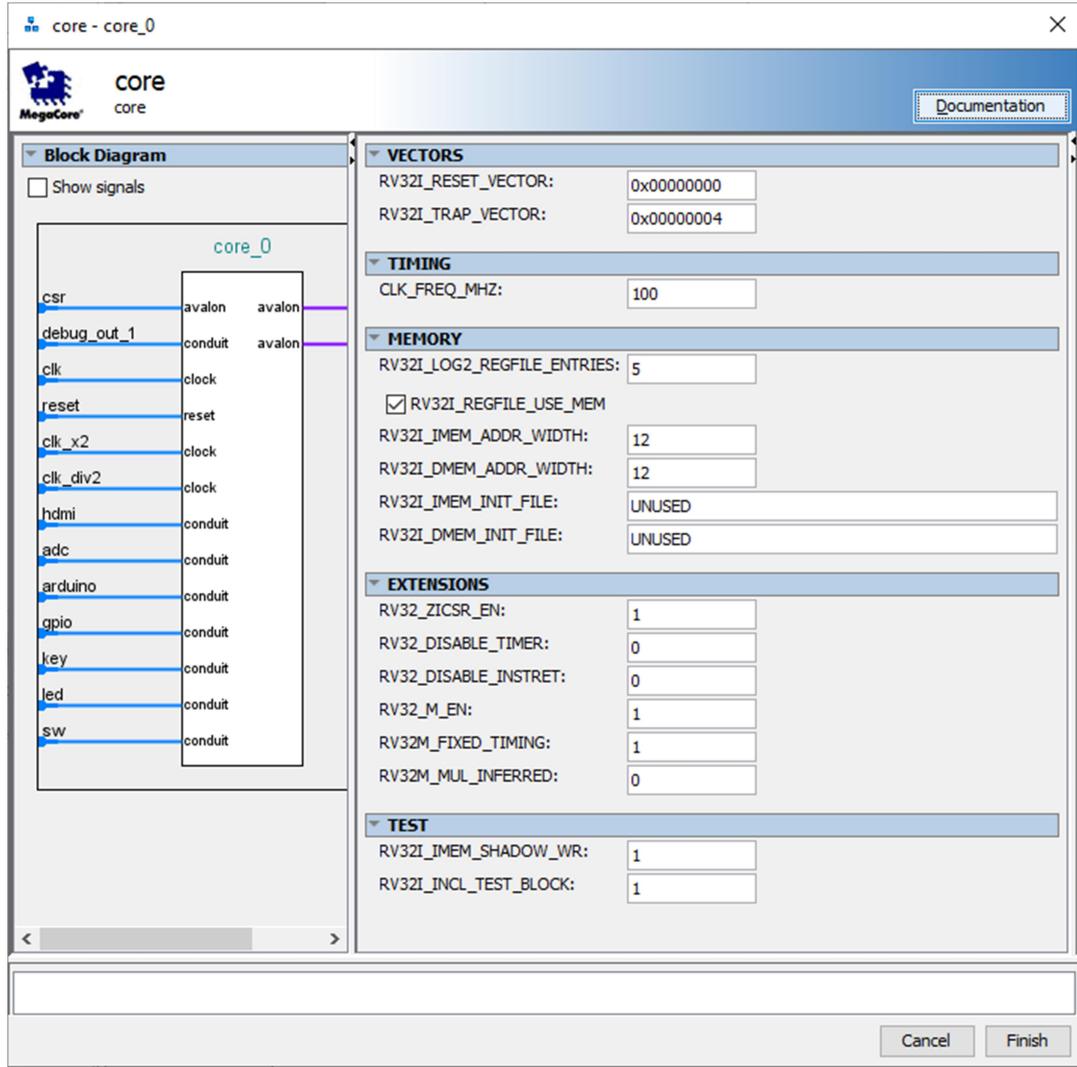


The environment assumes that the Quartus Prime tools have been installed on the host machine. The code has been tested on Quartus Prime Lite 18.1, but later versions should work just as well.

Core Component

The core logic is specific to the DE10-Nano target platform. Instantiates the rv32i_cpu_core and two Intel/Altera M-RAM memories. The source code is located in the HDL/de10-nano/src folder, along with a top.v file that is the synthesis top level for the target FPGA. The core has ports for all the possible interfaces for the DE10-Nano, but in this example most of these are not used.

The Quartus tools have been used to create a Platform Designer component from the core block that can be instantiated in the top_system, along with the Hard Processor System (HPS) and some other required peripherals. It is the top_system that is instantiated in the top.v code, and contains the core component. In the same folder as the core.v component file, the core_hw.tcl defines this component for use in the Platform Designer, listing all the files required and other attributes. The core parameters can be set in the platform designer by editing the instantiated core, and a window appears as shown below with the main interfaces and the parameters for altering.



Command Line Build

The design can be synthesised using the Quartus GUI tools in a standard manner, but a make based command line build option is provided. In the build directory a makefile can be used to synthesise the design by simply typing make from a command window from within the build folder.

When this is done, the script will construct the Platform Designer top_system from the top_system.qsys file, and then proceed to go through the build steps of synthesis. When complete, the timing report will be scanned for violations, and any found printed out for inspection.

The generated reports and FPGA image files are all generated in an output_files folder. Both a top.sof and a top.rbf file are generated for use, as appropriate, to configure the FPGA.

Results

Timing

The design was constrained for the target clock frequency of 100MHz via the Platform Designer's instantiated PLL components, with `outclk0` being this frequency and connected to the `clk` input of the core. No timing violations were reported and the estimated `fmax` frequency for the slowest corner was ~117.6MHz when compiled for RV32I only, reducing to 108.0MHz when the Zicsr extensions are enabled.

Resource Usage

The table below shows the achieved resource usage for the synthesis of the base RV32I implementation, showing both the actual number of ALMs used, plus an estimate of the logic element (LE) equivalent, based on a synthesis run targeting a Cyclone 10LP device (10CL010YU256C6G). Also the number of M10K RAM blocks is given.

module	ALMs (~LEs)	M10Ks
rv32i_cpu_core	6 (31)	0
rv32i_decode	133 (412)	0
rv32i_alu	480 (1055)	0
rv32i_regfile (RAM)	64 (156)	2
Totals	683 (1654)	2

When the Zicsr extensions are added, the results become as shown in the table below:

module	ALMs (~LEs)	Disabled timer & instret	M10Ks
rv32i_cpu_core	37 (81)	37 (80)	0
rv32i_decode	174 (458)	170 (465)	0
rv32i_alu	506 (1120)	506 (1102)	0
rv32i_regfile (RAM)	81 (218)	81 (218)	2
rv32_zicsr	451 (996)	314 (687)	0
Totals	1249 (2873)	1108 (2552)	2

When the RV32M extensions are added, on top of the RV32I base and Zicsr extensions, the results become as shown in the table below, with `RV32M_FIXED_TIMING` set to zero, and `RV32M_MUL_INFERRRED` set to non-zero, in the first column, and `RV32M_FIXED_TIMING` set to non-zero, and `RV32M_MUL_INFERRRED` set to zero, in the second column. The DSP column gives the DSP multiplier counts for the two cases, with the Cyclone 10LP counts in brackets.

module	ALMs (~LEs)	ALMs (~LEs)	M10Ks (M9K)	DSP (C10LP)
rv32i_cpu_core	37 (85)	37 (85)	0	0/0 (0/0)
rv32i_decode	168 (457)	166 (457)	0	0/0 (0/0)
rv32i_alu	551 (1276)	534 (1275)	0	0/0 (0/0)
rv32i_regfile (RAM)	79 (218)	75 (219)	2	0/0 (0/0)
rv32_zicsr	456 (963)	458 (965)	0	0/0 (0/0)
rv32_m	306 (617)	262 (649)	0	3/0 (8/0)
Totals	1597 (3616)	1532 (3650)	2	3/0 (8/0)

Platform Testing

Testing on the example platform based around repeating the `riscv-tests` appropriate to the build—at this time, the `rv32ui` and, if Zicsr extensions enabled, `rv32mi` tests, as well as the `rv32um` tests if the RV32M extensions are enabled. In order to do this, the DE10-Nano FPGA needs to be configured with the image generated in synthesis (see *Command Line Build* section above). Some software is also required to load a program to the internal memory and then run a test.

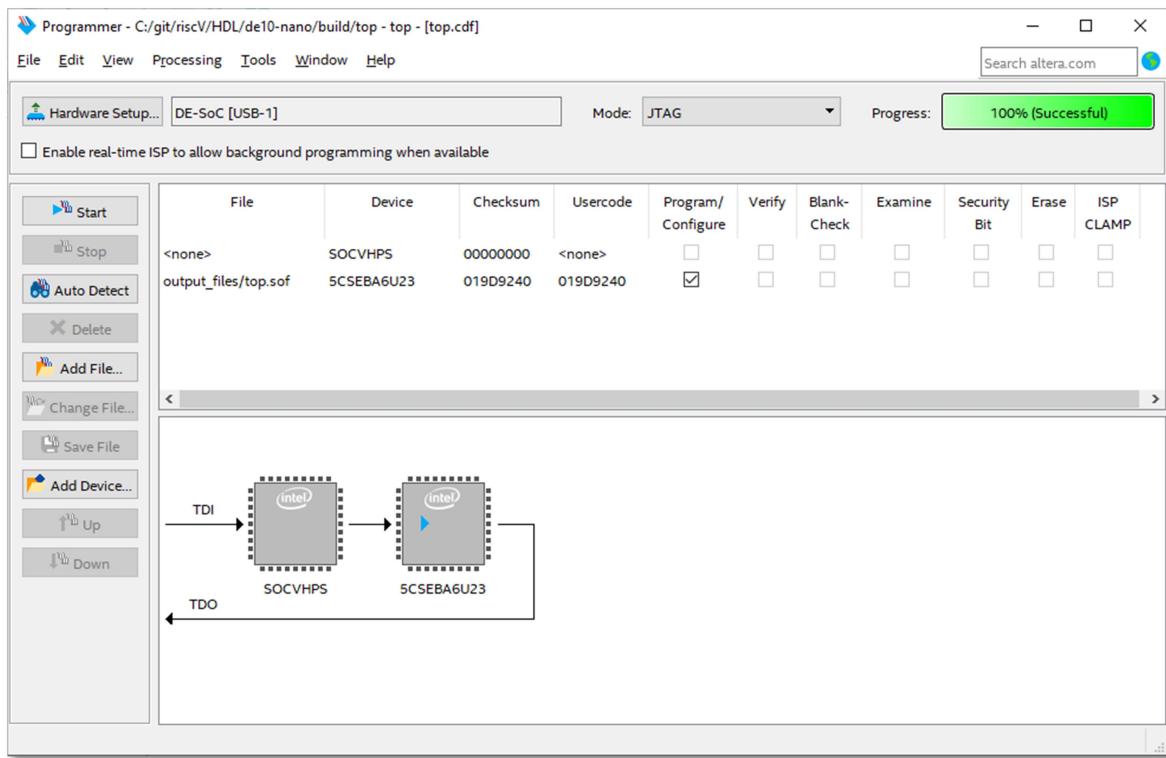
The core component is configured to instantiate the test logic (see diagram in *Test Bench* section above), which monitors for configured halt conditions. It is controlled via local core registers, accessed over a memory mapped control and status register bus (CSR) and also makes available the halt status and the value of the GP registers (see register table in the *Test Bench* section above). The test software accesses these registers to load a program, bring the core out of reset and then monitor for a halt status. It then determines the exit status and prints a PASS or FAIL message. This mimics the running of the tests in the simulation test bench, and the core parameters are set in synthesis to be the same as for simulation, apart from the `RV32I_IMEM_INIT_FILE`, which is left at the default of "UNUSED", as the code will be loaded by test program. This is shown in diagram in the Core Component section above.

Loading the FPGA Image

This document cannot go through a tutorial on the use of the Quartus tools, and the procedures are specific to the Intel FPGAs and the DE10-Nano board. It is supposed that familiarity with these tools, or their equivalent if targeting a different platform, is sufficient without further discussion. A brief summary is documented here for the example platform to indicate the steps required.

The DE10-Nano, for these tests is running a console version of Ångström Linux. At time of writing, this image can be found [here](#). With this setup, there are two ways to load an FPGA image on the DE10-Nano; over USB-Blaster JTAG or copy an image to the FAT32 partition so that it is loaded at boot time. The later needs the FAT32 partition mounted (e.g. `/mnt/fat32`), and the generated RBF (`top.rbf`) copied to that folder and named `soc_system.rbf`. The advantage of this is that the FPGA will be configured with the image every time the board is powered on.

Here we will concentrate on loading over the USB-Blaster. From the Quartus Prime GUI, the Programmer window can be launched. If connected correctly, the tool will detect the connection. If the first time connected, then the Auto Detect button should be pressed to inspect for the connected FPGA, but a `top.cdf` file is provided which already defines this chain.



In the diagram above is shown a programmer window with the correct chain configuration, and the `top.sof` file selected as the configuration image. Programming is then simply a matter of pressing the Start button. Programming this way means the image will be lost at power down, but it allows the image to be updated, without a reboot, when developing the logic.

Compiling the Test Program

The `HDL\de10-nano\test` folder contains the source files to compile a Linux ARM program to run `riscv-tests/rv32ui` tests on the DE10-nano platform.

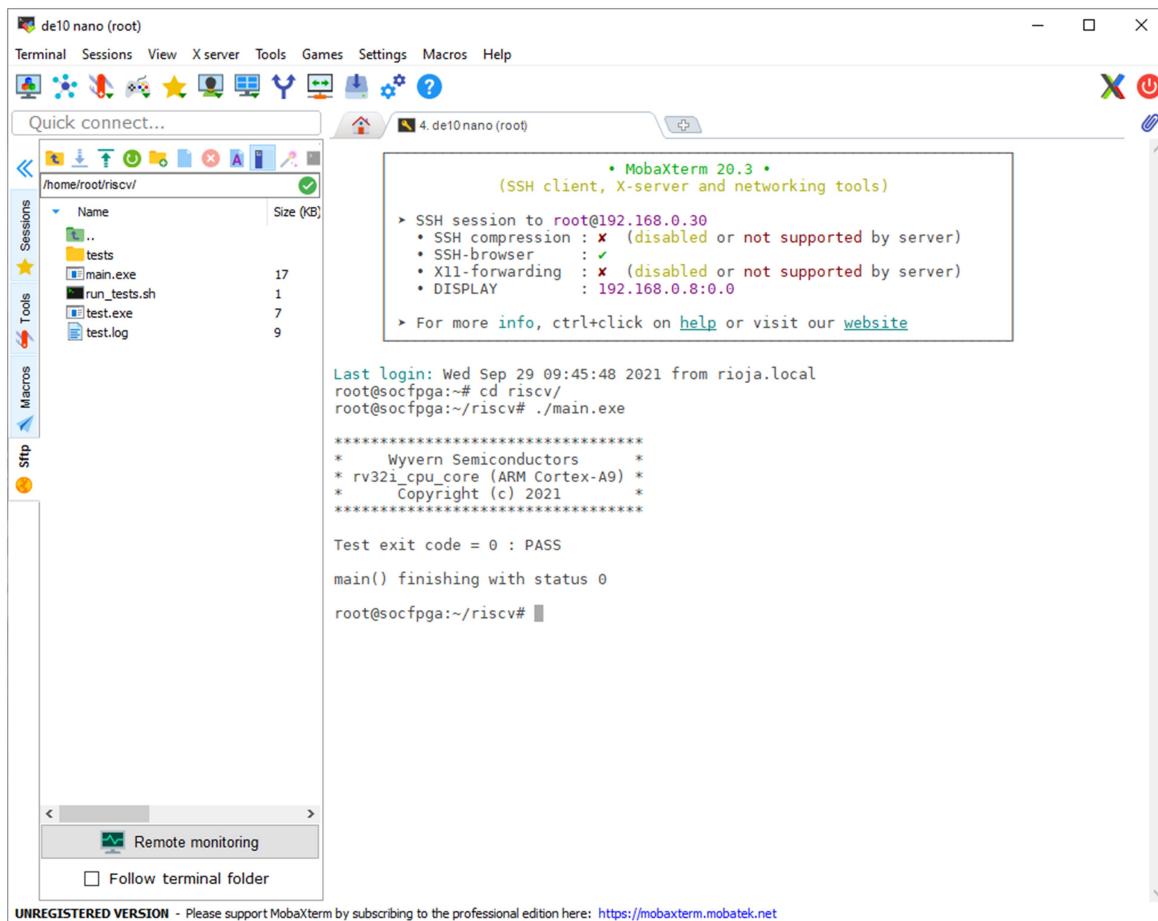
A makefile exists to build the test program. On a windows host, it assumes that the following tool chain is installed at the given location:

```
c:\Tools\gcc-linaro-4.9.4-2017.01-i686-mingw32_arm-linux-gnueabihf
```

If a toolchain is installed elsewhere, then the `TOOLPATH` variable should be modified, either on the command line, or the makefile updated. If no toolchain is available, a compiled version of the code is available in the folder (`make.exe`), but modifications to the program will not be possible. Alternatively, if the Linux running on the DE10-Nano has the GNU gcc toolchain installed, it may be compiled on platform. At the time of writing, the toolchain used in testing is available [here](#).

Uploading the Test Code

The `main.exe` executable should be copied to a suitable folder on the DE10-Nano. How this is done depends on the particular setup. For this development the DE10-Nano was connected to the local network (with a fixed MAC address—otherwise the allocated address must be discovered), and [MobaXterm](#) used to connect to the board as root. The diagram below shows a MobaXterm session and a single test run:



When run on the platform, `main.exe` program expects a RISC-V executable file called `test.exe`, which it will load to the internal memory and then execute. When the internal test block has halted, it will extract the state and print a PASS or FAIL message, as shown above.

Running the Tests

A script (`run_tests.sh`) is also provided to run the `riscv-tests/rv32ui` tests as a regression. A `test/` folder should be located in the same folder as the `main.exe` and the script, and contain all the compiled `rv32ui` tests. The script will run each executable in turn, and log the output in `test.log`. When complete all the PASS/FAIL messages are dumped to the screen for inspection.

The current version of the core has had all the tests run that were run in simulation (see table in the *Results* sub-section of the *Simulation* section), and all pass, verifying the logic on the example platform.

References

- [1] *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213*, Editors Andrew Waterman and Krste Asanović, RISC-V Foundation, December 2019.
- [2] *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 20190608-Priv-MSU-Ratified*, Editors Andrew Waterman and Krste Asanović, RISC-V Foundation, June 2019
- [3] *LatticeMico32 Processor Reference Manual*, Lattice Semiconductors, June 2012
- [4] *Nios II Processor Reference Guide Version 2020.10.22*, Intel, October 2020