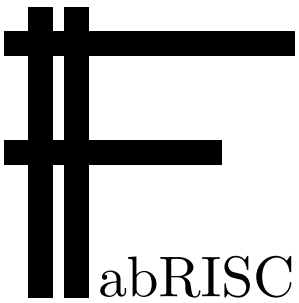

Instruction Set Architecture

Enrico Gatto Monticone

10/03/2022

Contents

List of Figures

1 Introduction

FabRISC is a simple RISC-like, register-register load-store architecture composed of xxx instructions with variable length encodings of two, four and six bytes. This specification is designed to be modular allowing a simple and straightforward implementation from basic designs up to high performance systems. The ISA includes vector, floating-point, compressed as well as privileged instructions supporting 32 and 64-bit multithreaded microarchitectures. This specification is completely free, open-source and available to anyone interested in the project (Creative Commons license). This document is divided into multiple sections each explaining the architecture in detail with the help of tables, figures and implementation specific suggestions in order to aid the hardware and software designers in creating an efficient realization of the FabRISC ISA.

Commentary in this document is formatted in this way and communication will be more colloquial. If the reader is only interested in the specification, these sections can be skipped without hindering the understanding of the document. This project tries to be more of a hobby learning experience rather than a new super serious industry standard, plus the architecture borrows many existing concepts from the most popular and iconic ISAs like x86, RISC-V, MIPS, ARM and openRISC. I chose to target FPGAs as the primary platform for two main reasons: one is that ASICs are out of the question for me and most people because of cost. Two is that using discrete components makes little sense from a sanity and practicality point of view given the complexity of the project, however, software simulators can be a good platform for simpler implementations. The main idea here is the use of variable length encoding of four and six byte instruction size along with an optional compressed module to increase code density. Flags are also another important and optional aspect that i emphasize because they can be a handy tool when it comes to efficiently control some unavoidable side effects as well as easing multi-word arithmetic. One final aspect of the ISA is the fact that all instructions can specify the length of the data type in order to more precisely control edge cases such as overflow and underflows (and orthogonality reasons). This is not achieved via register sub addressing but rather by simply masking the unnecessary portion of the word while leaving the masked bits in place.

1.1 Terminology

The FabRISC architecture uses the following terminology throughout the document:

- *"Architecture"* is used to refer to the set of abstractions that the hardware must provide to the software.
- *"Atomic"* is used to refer to any operation that must, either be completely executed, or not at all.
- *"Architectural state"* is used to refer to the state of the processor that can directly be observed by the programmer.
- *"Coherence"* is used to refer to the ability of a system to be coherent, that is, ensuring the uniformity of shared resources across the entire system. In particular, it defines the ordering of accesses to a single memory location for systems that implement caching.
- *"Consistency"* is used to refer to the ability of a system to be consistent, that is, defining a particular order of operations across all memory locations that is obeyed by everyone within the system.
- *"Consistency model"* is used to refer to a particular model or protocol of consistency within a particular system.
- *"Event"* is used to generically refer to any extra-ordinary situation that needs to be taken care of as soon as possible.
- *"Exception"* is used to refer to any non severe internal, synchronous event.
- *"Fault"* is used to refer to any severe internal, synchronous event.
- *"Hardware thread"* and *"thread"* are used to refer to a particular physical instance of a software thread running, specifically, on the central processing unit (CPU).
- *"Interrupt"* is used to refer to any external, asynchronous event.

- *"Memory fence"* and *"fence"* are used to refer to particular instructions that have the ability to enforce a specific ordering of other memory instructions.
- *"Memory transaction"* and *"transaction"* are used to refer to a particular series of operations that behave atomically within the system.
- *"Micro architectural state"* is used to refer to the actual state of the processor that might not be directly visible by the programmer.
- *"Micro architecture"* is used to refer to the particular physical implementation of a given architecture.
- *"Page"* is used to refer to a logical partition of the main memory.
- *"Promotion"* is used to refer to the automatic switch from user mode to supervisor mode by the processor.
- *"Transparent"* is used to refer to something that is (mostly) invisible to the programmer.
- *"Unaligned"* and *"misaligned"* are used to refer to any memory item that is not naturally aligned, that is, the address of the item modulo the referenced element size, is not equal to zero.

1.2 Implementation specific parameters

This document makes use of some implementation specific microarchitecture parameters to clear potential misunderstandings:

- *WLEN (Word Length)*: this parameter indicates the processor's natural scalar word length in bits, for example a 64 bit CPU will have WLEN of 64.
- *MXVL (Maximum Vector Length)*: this parameter indicates the processor's maximum vector length in bits. Possible values can be chosen from:
  - 64 bit: for processors with WLEN of 32
  - 128 bit: for processors with WLEN of 32 and 64
  - 256 bit: for processors with WLEN of 32 and 64
  - 512 bit: for processors with WLEN of 32 and 64

  In short, the possible values can be any power of two that is at least twice the size of WLEN and less than 512 bits.

2 Memory

This section is dedicated to the memory model used by FabRISC including data alignment, synchronization, consistency, as well as possible cache coherence directives are explained here.

2.1 Data alignment

FabRISC, overall, treats the main memory and the MMIO regions as collections of byte addressable locations in *little endian* order with a range of 2^{WLEN} addresses in total. The specification leaves to the hardware designer the choice of supporting aligned or unaligned memory accesses or both for data, however, if aligned is decided to be the only supported scheme, the processor must generate the appropriate fault every time the constraint is violated (consult section xxx for more information). When it comes to instructions, it's mandatory to have fetch engines that support accesses at the 16-bit boundary alignment. This is because the greatest common denominator of the instruction sizes, with or without the compressed module, is 16 and the programmer must ensure that the code is aligned at said boundary, if not, a fault must be generated. Branch offsets, as a result of this, are logically shifted by one place to the left before being added to the PC. This means that said offsets will specify 16-bits as the smallest addressable object, effectively doubling the range in terms of bytes.

Data alignment issues can arise when the processor wants to read or write an item whose size is greater than the smallest addressable thing. This problem is tricky to design hardware for, especially caches, because misaligned items can cross cache line boundaries as well as page boundaries. Alignment networks and more complex caches are needed which can increase complexity and slow down the critical path too much for simple designs. For already complex multicore out-of-order superscalar machines, however, i believe that supporting unaligned accesses can be handy so that the software writer can make decisions freely without having to worry about this problem and potentially decrease the memory footprint.

2.2 Synchronization

FabRISC provides dedicated atomic instructions to achieve proper synchronization in order to protect critical sections and to avoid data races in threads that share memory with each other. The proposed instructions behave atomically to each other and can be used to implement atomic *"read-modify-write"* operations for locks, semaphores and barriers. It is important to note that if the processor is a simple single core and single threaded machine, then this section can be skipped since the problem can be solved by the OS.

- *Load Linked* (LL) is an atomic memory operation that loads an item from memory into a register and performs a *"reservation"* of the fetched location. The reservation can simply be storing the physical address and size of the object into an internal transparent register.
- *Store Conditional* (SC) is an atomic instruction that stores an item from a register to a memory location if and only if the reservation matches and is marked as valid, that is, the physical address and size are the same plus a valid bit set. In the case of a mismatch, or an invalidity, SC must not perform the store and must return a zero in its register destination as an indication of the failure. If STC succeeds, the item is written to memory, a one is returned into its register destination and all reservations must be invalidated.

FabRISC also provides optional instructions to support basic transactional memory that can be employed instead of locks to exploit parallelism in a more *"optimistic"* manner. Multiple transactions can happen in parallel as long as no conflict is detected by the hardware, however when such situations occur, the offended transaction must aborted, that is, it must discard all the changes and restore the architectural state immediately before the start of the transaction itself. If a transaction detects no conflict it is allowed to commit the changes and the performed operations can be considered atomic. Transactions can be nested inside each other up to a depth of 256, beyond this, an exception must be generated to notify the programmer.

- *Transaction Begin* (TBEG): causes the thread that executed this instruction to start monitoring accesses by other threads via the coherence protocol as well as incrementing the nesting counter by one. This instruction effectively starts a transaction.

- *Transaction End* (TEND): causes the thread that executed this instruction to stop monitoring accesses by other threads and commit the changes as well as decrementing the nesting counter by one. This instruction effectively terminates a transaction.
- *Transaction Abort* (TABT): causes the thread that executed this instruction to stop monitoring accesses by other threads as well as generate an *"Explicit abort"* exception within the thread and cause it to restore the architectural state immediately before the latest TBEG and it will also cause the transaction nesting level counter to be decremented by one. This instruction effectively aborts a transaction.
- *Transaction Check* (TCHK): causes the thread that executed this instruction to return, in a specified register, the status of the current running transactional execution. This instruction effectively checks if the thread is in a transaction as well as its depth.

Transactions can generate exceptions called *"Abort codes"* that can be used by the programmer to take action in case the transaction was aborted. Each abort code encodes the reason why the current transaction was aborted:

- *Associativity abort*: the current transaction was aborted because a cache line was evicted for not enough associativity.
- *Conflict abort*: the current transaction was aborted because a write on shared variables was detected by the coherence protocol.
- *Capacity abort*: the current transaction was aborted because a cache line was evicted for not enough space.
- *Depth limit abort*: the current transaction was aborted because it exceeded the transaction depth limit.
- *Event abort*: the current transaction was aborted because an event, beside the ones in this list, got triggered.

Memory synchronization is extremely important in order to make shared memory communication even work at all. The problem arises when a pool of data is shared among different processes or threads that compete for resources and concurrent access to this pool might result in erroneous behaviour and must, therefore, be arbitrated. This zone is called "Critical section" and special atomic primitives can be used to achieve this protection. Many different instruction families can be chosen such as "Compare and swap", "Test and set", "Read-modify-write" and others. I decided to provide in the ISA the LL and SC pairs, as described above, because of its advantages and popularity among other RISC-like instruction sets. Two important advantages of this pair is that it is pipeline friendly (LL acts as a load and SC acts as a store) compared to others that try to do both. Another advantage is the fact that the pair doesn't suffer from the "ABA" problem. It is important to note, however, that this atomic pair doesn't guarantee forward progress and weaker implementations can reduce this chance even more. I decided to also provide basic transactional memory support because, in some situations, it can yield great performance compared to mutual exclusion without losing atomicity. This is completely optional and up to the hardware designer to implement or not simply because it can significantly complicate the design. Transactional memory seems to be promising in improving performance and ease of implementation when it comes to shared memory programs, but debates are still ongoing to decide which exact way of implementing is best.

2.3 Coherence

FabRISC leaves to the hardware designer the choice of which coherence system to implement. On multicore systems cache coherence must be ensured by choosing a coherence protocol and making sure that all the cores agree on the current sequence of accesses to the same memory location. That can be guaranteed by serializing the operations via the use of a shared bus or via a distributed directory and *Write-update* or *Write-invalidate* protocols can be employed without any issues. Software coherence can also be a valid option but it will rely on the programmer to explicitly flush or invalidate the cache of each core separately. Nevertheless, FabRISC provides implementation-dependent instructions that can be sent to the cache controller directly

to manipulate its operation (see section xxx for details). If the processor makes use of a separate instruction cache, potential complications can arise for self modifying code which can be solved by employing one of the above options, furthermore the ISA includes implementation specific instructions such as CACOP and MMUOP to directly manage caches and the MMU in these particular cases.

Cache coherence is a big topic and is hard to get right because it can hinder performance in both single core and multicore significantly. I decided to give as much freedom as possible to the designer of the system to pick the best solution that they see fit. Another aspect that could be important, if the software route is chosen, is the exposure to the underlying microarchitecture implementation to the programmer which can be yield unnecessary complications. Generally speaking though write-invalidate seems to be the standard approach in many modern designs because of the way it behaves in certain situations, especially when a process is moved to another core. Simple shared bus can be a good choice if the number of cores is small (lots of cores means lots of traffic), otherwise a directory based approach can be used to ensure that all the cores agree on the order of accesses. From this, the protocol can be picked: MSI, MESI, MOSI or MOESI, the latter being the most complex but most powerful.

2.4 Consistency

FabRISC utilizes a fully relaxed memory consistency model formally known as *Release consistency* that allows all possible orderings in order to give processors the freedom to reorder memory instructions to different addresses in any way it wants. For debugging and for specific situations the stricter *Sequential consistency* model can be utilized and the processor must be able to switch between the two at any time via a dedicated bit in the control and status register. Special instructions, called *"Fences"*, are provided to let the programmer impose an order on memory operations when the relaxed model is in use. If the processor doesn't reorder memory operations this section can be skipped.

- *Fence Loads* (FNCL): this instruction forbids the processor to reorder any load instruction across the fence.
- *Fence Stores* (FNCS): this instruction forbids the processor to reorder any store instruction across the fence.
- *Fence Loads and Stores* (FNCLS): this instruction forbids the processor to reorder any load or store instruction across the fence.

The fences can be used on any memory type of instruction, including the LL & SC pair to forbid reordering when acquiring or releasing a lock for critical sections and barriers. Writes to portions of memory where the code is stored can be made effective by issuing a command to the cache controller via the special implementation specific CACOP instruction as discussed above.

The memory consistency model i wanted to utilize was a very relaxed model to allow all kinds of performance optimization to take place inside the system. However one has to provide some sort of restrictions, effectively special memory operations, to avoid absurd situations. Even with those restrictions debugging could be quite difficult because the program might behave very weirdly, so i decided to include the sequential model that forbids reordering of any kind of memory instruction. If a program is considered well synchronized (data race-free and all critical sections are protected) consistency becomes less of an issue because there will be no contention for resources and, therefore, the model can be completely relaxed without any side effects. Achieving this level of code quality is quite the challenge and so these consistency instructions can be employed in making sure that everything works out.

---

3 Input Output

This section is dedicated to the specification that FabRISC uses for communicating with external devices as well as other cores and hardware threads if present. The architecture defines IO mappings and potential DMA behaviour and, in the next section, OS support and inter-process communication schemes are discussed.

3.1 Memory mapped IO

FabRISC reserves a portion of the high memory address space to *Memory mapped IO*. This region, of the size of 2^{16} bytes, is not cached nor paged and byte addressable in little endian order. If a thread wants to transfer data to an IO device it can simply execute a memory operation to this section without further complications. The IO device must map all of its internal registers and state to this region, multiple channels or buses can potentially be employed to reduce the latency in case another transfers are already taking place as well as increasing the bandwidth. It is important to note that this region, although not paged in the traditional sense, (the virtual to physical mapping will always be the same) it must still have page table entries for protection. The ISA roughly splits this address space in two segments:

- *CPU segment*: this portion, starting from address zero, is composed of xxx bytes and should be used to hold CPU information, such as implemented ISA extensions, cache sizes and other CPU capabilities and characteristics.
- *IO segment*: this portion, starting from address xxx, is composed of xxx bytes and should be used to communicate with external devices.

I decided to go with memory mapped IO because of its flexibility and simplicity compared to port based solutions. The IO region can be considered plain memory by the processor internally, which allows for advanced and fancy operations that use locks, barriers, fences and transactions to be done by multiple threads to the same device. I don't recommend caching or paging this region because it can yield potential inconsistencies and unnecessary complexities.

3.2 Direct memory access

FabRISC provides the ability for IO devices to access the main system memory directly via DMA without passing through the processor. A dedicated centralized controller can be utilized to achieve this, but the hardware designer is free to choose another alternative if considered appropriate and, if this method of communication is chosen to be used, cache coherence must be ensured to the IO devices too. Some possible options can be (as discussed earlier):

- *Non cacheable memory region*: with this configuration coherence isn't a problem because no caching is performed by the CPU and the IO device in question. The system, however, needs to be able to dynamically declare which portion of memory is cacheable and which isn't which can lead to unnecessary complexities.
- *Software IO coherence*: with this configuration the CPU and the device are required to flush or invalidate the cache explicitly with no extra hardware complexity, however, this option requires the exposure of the underlying organization to the programmer.
- *Hardware IO coherence*: with this configuration, both the CPU and the IO device, will monitor each other's accesses via a common bus or a directory and proper actions are automatically taken according to a coherence protocol which can be the already existent one in the processor.

The DMA protocol / scheme implement by the hardware designer must also take consistency into account since memory operations to different addresses are allowed be done out-of-order. This means that fencing instructions must retain their effect.

For more bandwidth demanding devices, DMA can be used to transfer data at very high speeds in the order

of several Gb/s without interfering with the CPU. This scheme however, is more complex than plain MMIO because of the special arbiter that handles and grants the requests. IO coherence, as well as its consistency, is actually the main reason of this subsection as a remainder that it needs to be considered during the development of the underlying microarchitecture.

---

4 Events

This section is dedicated to the specification of exceptions, faults and interrupts. FabRISC uses the term *event* to indicate the generic categorization of these kinds of situations. Events can be used to communicate with other threads, processes, IO devices, signal system faults or simply trigger software caused exceptions. Events are sub divided into two main categories: Synchronous and Asynchronous.

4.1 Synchronous

Synchronous events are internally generated and are considered deterministic, that is, if they happen, they will always happen in the same location in the executing program and, because of this, the handling is done in program order. This category is further sub divided in two, consisting of:

- *Exceptions*: These events are user handled, non promoting and with a global priority level of 0. They are generated by executing instructions and each process can have it's own private handler with the help of a dedicated pointer register. From a higher level, the handling of exceptions looks like an automatic function call to the specified handler address.
- *Faults*: These events are supervisor handled, promoting and with a global priority level of 2. They are generated by executing instructions and each process will have the same handler specified by the supervisor. From a high level, the handling of faults looks like an automatic function call performed by the supervisor to a location that is always the same, usually under the for of a branch table, to the proper handler.

Exceptions and faults are used to identify software related problems and edge cases such as overflows, divisions by zero, memory accesses to protected sections and more. I believe that exceptions can be valuable and useful for handling special cases quickly and pre-emptively without having to perform time consuming checks. Examples of this can be seamless array boundary checks via the use of breakpoint registers (see section xxx for more information) that trigger the appropriate exception. Arithmetic edge cases can be similarly treated via the use of said special registers or by operating the processor in a "safe state", which will trigger the appropriate exception every time something went wrong with the executing software. Faults are similar to exceptions but for more delicate problems such as page faults, illegal or malformed instructions and accesses to protected memory areas. All in all, i believe that one cannot really design a hardware system where these kinds of problems never occur, which is why i put emphasis on being able to gracefully recover when such cases happen.

4.2 Asynchronous

Asynchronous events are externally generated and are not considered deterministic, that is, they can happen at any time regardless of what the CPU is doing. Asynchronous events can be masked but only by the supervisor and should be handled as soon as possible in order to keep latency low. This category is further sub divided in two, consisting of:

- *IO interrupts*: These events are supervisor handled, promoting with a global priority level of 3. They are generated by external IO devices and can have an internal priority level (to decide which one to handle when multiple are triggered at the same time). From a high level, the handling of IO interrupts, can be considered as a context switch to the desired process that will handle the IO device request, which can be achieved with the use of a branch table (similar to faults).
- *IPC interrupts*: These events are supervisor handled, promoting with a global priority level of 1. They are generated by other cores in the system in order to communicate with each other (inter-process communication) and can have an internal priority level (to decide which one to handle when multiple are triggered at the same time). From a high level, the handling of IPC interrupts, can be considered as a context switch to the desired process that will handle the IPC request, which can be achieved with the use of a branch table (similar to faults & IO interrupts).

I think that interrupts are a great tool that allow the IO devices themselves to start the communication, as opposed to the time consuming polling. This option is especially useful when low latency is required and can be used in conjunction with the regular low speed memory mapped IO transfers or the faster DMA. For devices that do not need any kind of bandwidth or responsiveness, polling can still be a valid choice without utilizing any extra resources. A shadow register file can potentially be utilized to reduce the latency to a minimum, however, it must behave transparently to the architecture.

4.3 Handling

Handling events involves a *"launching"* phase and a *"returning"* phase. The steps of each phase must be performed by the CPU in a single cycle in order to avoid corruption of the internal state. Launching phase is composed of the following steps:

1. save the value of the program counter (PC) and status register (CSR) into appropriate buffer register depending on the privilege level of the event.
2. mask any other interrupt if the triggered event is promoting, otherwise skip this step. The masking can be accomplished by simply setting a bit in the CSR to the appropriate value.
3. switch to supervisor mode if the triggered event is promoting, otherwise skip this step.
4. jump to the handler location. Depending of the privilege level of the event this step will involve:
    - for non-promoting events, simply copy the value of the user event table pointer (UETP) into the program counter (PC).
    - for promoting events, simply copy the value of the supervisor event tablepointer (SETP) into the program counter (PC).

After the execution of the desired handler code, the last instruction will trigger the returning phase. During that phase, the CPU will essentially do the opposite of what the launching phase did:

1. restore the value of the program counter (PC) and status register (CSR) by storing into them the values of the appropriate buffer registers previously written.
2. ...

Global priority, introduced in the previous subsections, is used to give a further discriminant in case different types of events are triggered in the same clock cycle. Events with global priority level of 3 will be considered first, all the way to priority level 0. Local priority simply specifies the order in which events of the same type must be handled. Local priority concerns asynchronous events only, since synchronous events must be handled in program order as discussed in the previous sections.

Comment goes here...

5 OS support

This section is dedicated to the supported OS primitives by the FabRISC architecture. They are divided into different categories explained below and should be, ideally, handled by the hardware in order to give the operative system a solid foundation. The features allow the realization of any kind of OS, from monolithic designs, microkernel, exokernel and hybrids.

5.1 Supervisor mode

FabRISC is a privileged architecture and makes use of the supervisor to create a border between the OS and the user. With supervisor privileges the executing code has complete and total control over the hardware and any access to protected resources in user mode, such as particular instructions, registers, or addresses must generate a fault. The processor should only enter the supervisor mode if an appropriate event is triggered and a dedicated instruction is provided to transfer the control back to the user (consult section xxx for more information). If the system makes use of a multicore processor then each core must have the ability to be in supervisor mode or not independently. When booting up, the system should start in supervisor mode with only one core active to allow the loading of the OS itself and the creation of all the required data structures. This is indicated by a special bit in the status register (CSR).

Having a supervisor mode is essential for implementing a working operative system. This special mode allows the executing code to have complete and total control over the CPU and is used, by the OS, to protect itself from other processes that could, intentionally or not, compromise the system. When the processor isn't running in supervisor mode it will run in user mode with restrictions, mainly in manipulating certain hardware state including some special registers, addresses and instructions. This, along with virtual memory, will effectively confine any user level process to its own sandbox protecting itself and others.

5.2 Virtual memory

FabRISC provides privileged customizable (implementation specific) instructions to interface directly with the MMU, if the processor has one, allowing the OS to modify and manage that particular unit (see section xxx for more details), alternatively, the managing can be automatic if the hardware supports it. A special purpose register called PTP is also provided to hold the physical address of the page table to perform the page table walk in case of a TLB miss. PTP must be set by the supervisor only with the appropriate address every time a context switch happens and a write operation on PTP in user mode must cause a fault. The TLB can also make use the process ID in the PID register to avoid flushing during context switches and page swapping can be supported via the generation of the *Page fault* event that can be handled by the OS.

Virtual memory is a central concept in any operative system and i wanted to provide, at least, basic support without going too crazy (this is pretty much as simple as it gets). The SR1 also has a special bit to enable or disable paging by bypassing the TLB and avoid doing the virtual to physical translation all together. The process ID can help reducing the burst of TLB misses caused by context switches especially with frequent system calls in microkernel based solutions.

5.3 Hardware thread communication

Thanks to the already discussed IPC interrupts, it is possible to implement efficient low level hardware thread communication. Initialization and termination of threads, however, is mandatory since it effectively allows the creation and destruction of threads if the processor has multiple cores.

- *Thread signalling*: this form of communication is achieved by the use of dedicated interrupts as suggested earlier. The signal can be sent to an IO controller just like a request from an external IO device which can then forward the request to the destination thread that is interrupted.
- *Message passing*: this form of communication is achieved by the use of dedicated MMIO addresses to send and receive small messages through the IO controller. The message formats should use a common convention agreed by the hardware designer and the programmer for everybody.

- *Thread initialization* and *Thread termination*: this form of communication is vital to be able to start and stop threads at will in multicore systems. Starting a thread can be achieved by sending a dedicated IPC interrupt that has the ability to *wakeup* the destination core or hardware thread from a halted state. A similar and symmetric picture can employed to stop running cores or threads via a dedicated halting IPC interrupt.

Efficient communication is key to achieve good performance in any system. I decided to support simple message passing and signalling directly in hardware to lower the latencies as much as possible since an FPGA CPU will only have clock speeds in the hundreds of MHz. The only form of communication that isn't explicitly listed here is "Shared memory" simply because it is a natural consequence of virtual memory and paging. In this subsection i assumed the presence of a centralized IO controller that manages interrupts, IO requests and transfers. Any other solution that behaves similarly to that is completely possible and allowed, for example a more distributed kind of controller where each core has its own independent logic (or something along those lines) could be a possible alternative.

---

6 ISA specification

In this section the register file organization, vector model, processor modes and the ISA modules are presented. FabRISC is a modular ISA composed of four macro modules each divided into several sub-modules that the hardware designer can choose. The macro modules concern computational, data transfer, control transfer and system instructions:

- ...

To indicate which ISA modules a particular implementation is composed of, a simple binary number can be used similarly to a checklist.

6.1 Register file

Depending on which modules, WLEN and VLEN values are chosen for implementation, the register file can be composed of different banks (up to four) of variable width:

1. Scalar general purpose registers (SGPRs): this bank is composed of 32 registers which can be used for holding values during execution. The registers are all WLEN bits wide and are used by integer and floating point scalar instructions.
2. Vector general purpose registers (VGPRs): this bank is composed of 32 registers which can be used for holding values during execution. The registers are all VLEN bits wide and are used by integer and floating point vector instructions. This bank is only necessary if the processor supports vector execution.
3. Special purpose registers (SPRs): this bank is composed of 32 registers which are internally used to keep track of state, modes, flags and other things. The registers can be WLEN or 32 bits wide with some being privileged resources.
4. Vector flag register (VFLG): this bank is composed of $fracVLENWLEN$ registers which are used to keep trach of the vector flags. The registers are all WLEN bits wide and this bank is only necessary if the processor supports vector execution.

6.2 Register ABI

FabRISC specifies an ABI (application binary interface) for the SGPRs and VGPRs. It is important to note that this is just a suggestion on how the general purpose registers should be used in order to increase code compatibility. For scalar registers:

- P: registers prefixed with the letter 'P' are used for parameter passing and returning to and from function calls. Parameters are stored in these registers starting from the top and going down, while returning values are stored starting from the bottom and going up.
- S: registers prefixed with the letter 'S' are "persistent" registers, that is, registers whose value should be retained across function calls. This implies a "callee save" calling convention for these registers.
- N: registers prefixed with the letter 'N' are "volatile" registers, that is, registers whose value may not be retained across function calls. This implies a "caller save" calling convention for these registers.
- GP (global pointer): this register is used to point to the global variable area.
- SP (stack pointer): this register is used as a pointer to the call-stack.
- FP (frame pointer): ...
- RA (return address): this register is used to hold the return address of the currently executing function call.

For vector registers:

- ...
- ...

6.3 Special purpose register bank

The special purpose register bank, as mentioned earlier, is composed or 32 registers with a specific purpose in mind. It is important to note that multiple registers may be updated every cycle, which means that, this bank should be seen more as a grouping of registers rather than an actual file.

- DBG (debugging): registers prefixed with "DBG" are used for debugging and other support features such as real-time and performance counters. If DBG registers are not used for any specific purpose, they can be used as extra scalar GPRs in which the ABI categorizes them as extra volatile registers. DBG registers are all WLEN bits wide and are not privileged.
- DBGM (debugging mode): registers prefixed with "DBGM" are used to specify the particular operating mode of the DBG registers (see table below for more information). Each DBG register has a five bit code reserved in its corresponding DBGM register to indicate the operating mode. Registers DBG0 to DBG5 will map to DBGM0 and while registers DBG6 to DBG10 will map to DBGM1. All DBGM registers are always 32 bits wide and are not privileged.
- EB (exception buffer): registers prefixed with "EB" are used as temporary buffers for exception handling. They are transparent registers that are automatically managed by the CPU during the launch and return phases of the handling. EB0 will store the program counter (PC), while EB1 will store the status register (CSR). All EB register are WLEN bits wide and are not privileged.
- ETP (exception table pointer): this register is used to hold the logical address of the exception branch table for the current executing process. During the launching phase, ETP will be used to perform an absolute branch (copy the value into the PC). ETP is WLEN bits wide and is not privileged.
- VMSK (vector mask): ...
- PC (program counter): this register is the program counter / instruction pointer. It is used to point to the currently executing instruction. PC is WLEN bits wide and is not privileged.
- CSR (control and status register): this register is used to hold processor state such as arithmetic flags, execution modes, etc (see table below for more information). CSR is 32 bits wide and is privileged.
- SEB (supervisor event buffer): registers prefixed with "SEB" are used as temporary buffers for supervisor event handling. They are transparent registers and are automatically managed by the CPU during the launch and return phase of the handling. SEB0 will store the program counter (PC), while SEB1 will store the status register (CSR). All SEB register are WLEN bits wide and are privileged.
- SETP (supervisor event table pointer): this register is used to hold the logical address of the supervisor event branch table. During the launching phase, SETP will be used to perform an absolute branch (copy the value into the PC). SETP is WLEN bits wide and is privileged.
- PTP (page table pointer): this register holds the physical address of the page tables for the current executing process. During a TLB miss, the MMU can use PTP to perform the page table walk. If the CPU transitions from user to supervisor mode, PTP must be set to point to the page tables of the OS with the use of a hardwired value. PTP is WLEN bits wide and is privileged.
- PID (process identifier): this register holds the process identifier for the currently executing process. The MMU can use PID to avoid flushing the TLB during context switches because PID serves as a discriminant. If the CPU transitions from user to supervisor mode, PID must hold the identifier of the privileged process (kernel) via the use of hardwired values.
- FU (file usage): ...
- KR (kernel register): registers prefixed with "KR" are kernel reserved general purpose registers. KR registers are all WLEN bits wide and are privileged.

6.4 Instruction formats

...

6.5 Decoding guidelines

...

7 Instruction list

This section is dedicated to a full and extensive list of all the proposed instructions in the ISA which are divided into their corresponding module.

7.1 Computational instructions

...

7.1.1 (ADD) Addition

Computes the binary addition between $rb and $rc and stores the result in $ra, as well as updating the arithmetic flags.

- Format: A.
- Opcode: 111110100000.
- Updated flags: ...
- Operation: $x = y + z$.
- Privilege: user.

...

7.2 Data transfer instructions

...

7.3 Control transfer instructions

...

7.4 System instructions

...

8 Macro-op fusion

...