FabRISC

Instruction Set Architecture

Enrico Gatto Monticone

17/04/2022

Contents

List of Tables

1 Introduction

FabRISC is a feature rich, register-register, load-store architecture with variable length encodings of two, four and six bytes. This specification is designed to be highly modular allowing a simple and straightforward implementation from basic designs up to high performance systems by picking only the desired modules. The ISA includes scalar, vector, floating-point, compressed, atomic as well as privileged instructions supporting 32 and 64-bit multithreaded microarchitectures. It is also possible to further enrich and enhance the ISA by allocating unused opcode combinations to custom application specific instructions. This specification is completely free, open-source and available to anyone interested in the project via the official GitHub: xxx (license details can be found at the very end of the document). The specification 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 instruction set architecture.

Commentary in this document will 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 core ideas here are the use of variable length encoding of four and six byte instruction size along with an optional compressed module to increase code density. Another 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 overflows and underflows as well as orthogonality reasons. This is not achieved via register sub addressing but rather by simply masking the unnecessary portion of the word while leaving the ignored bits in place. This ISA, all though not a "pure" RISC design with basic instructions and few addressing modes, resembles that philosophy for the most part skewing away from it in specific (and optional) areas only, such as, being able to load, store, move and swap multiple registers with a single instruction.

1.1 Terminology

The FabRISC architecture uses the following terminology throughout the document in order to more accurately define technical concepts and vocabulary:

- ***"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, or a single core or hardware thread, 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 techniques.*
- ***"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.*
- ***"Core"*** *is used to refer to a fully functional and complete sub-CPU within a bigger entity. Advanced processors often aggregate multiple similar sub-CPUs in order to be able to schedule different programs each working on it's own stream of data. It is important to note that each core can implement a completely different microarchitecture, as well as, instruction set.*

- ***"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"*** *or simply* ***"hart"*** *are used to refer to a particular physical instance of a software thread running, specifically, on the central processing unit (CPU).*
- ***"Instruction set architecture"*** *or simply* ***"ISA"*** *are used to refer to the architecture that the central processing unit provides to the software under the form of instructions.*
- ***"Interrupt"*** *is used to refer to any external, asynchronous event.*
- ***"Memory fence"*** *or simply* ***"fence"*** *are used to refer to particular instructions that have the ability to enforce a specific ordering of other memory instructions.*
- ***"Memory transaction"*** *or simply* ***"transaction"*** *are used to refer to a particular series of operations that behave atomically within the system.*
- ***"Microarchitectural state"*** *is used to refer to the actual state of the processor, or a single core or hardware thread, that might not be visible by the programmer in its entirety.*
- ***"Microarchitecture"*** *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 system memory.*
- ***"Promotion"*** *is used to refer to the automatic switch from user mode to supervisor mode by the processor or a single core or hardware thread, caused by an event.*
- ***"Transparent"*** *is used to refer to something that is, mostly, invisible to the programmer.*
- ***"Trap"*** *is used to refer to the transition from a state of normal execution to the launch of an event handler.*
- ***"Unaligned"*** *or* ***"misaligned"*** *are used to refer to any memory item that is not naturally aligned, that is, the address of the item modulo its size, is not equal to zero.*

1.2 Implementation specific parameters

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

- ***Word Length*** *(WLEN): this parameter indicates the processor's natural scalar word length in bits, for example, a 64-bit CPU will have WLEN of 64.*
- ***Maximum Vector Length*** *(MXVL): 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 64.*
- ***ISA Modules*** *(ISAMOD): this parameter indicates the implemented instruction set modules of the processor. This parameter is xxx bits long and works as a checklist where each bit indicates the desired module. The list of all possible modules is presented in the next section.*

I consider 512 bits for 64 bit machines and 256 for 32 bit machines to be a good maximum limit since i doubt that even advanced architectures would benefit in a practical way from a wider vector pipeline except, perhaps, in some very specific situations. This length can already be a bit of a challenge because of the shear width, necessitating a very large number of bits for the reservation station entries in out-of-order machines. Other than a practical limit, there is also a physical limit to the vector length which is given by the length of the vector mask register (VMSK) in the special purpose register file. This is because VMSK must be VLEN bits wide which, in the case of 64 bit machines can allocate up to 64 mask bits (one per byte element). The same logic applies for 32 bit machines which are limited to 32 mask bits.

2 ISA modules list

This section is dedicated to provide a brief but full description of all the different modules that the FabRISC ISA offers. There are no mandatory modules in this specification in order to maximize the flexibility, however, once a particular extension is chosen, the hardware must provide all the features and abstractions of said extension. The requirements for each and every module will be extensively explained in the next sections. Modules are divided into two main categories:

1. ***Instruction modules****: these modules add instructions only.*
2. ***Miscellaneous modules****: these modules can add things other than instructions such as registers, counters, events, processor operating modes, etc...*

2.1 Instruction modules

Instruction modules only concern instructions and their implementation. These modules are further divided into four general sub-categories each of which contains the actual instruction modules:

1. ***Computational****: these instructions perform arithmetic or logic operations.*
    - 1) ***CNISB****: computational-normal-integer-scalar-basic.*
    - 2) ***CNISA****: computational-normal-integer-scalar-advanced.*
    - 3) ***CNIVB****: computational-normal-integer-vector-basic.*
    - 4) ***CNIVA****: computational-normal-integer-vector-advanced.*
    - 5) ***CNIVR****: computational-normal-integer-vector-reduction.*
    - 6) ***CNISB****: computational-normal-integer-scalar-basic.*
    - 7) ***CNISA****: computational-normal-integer-scalar-advanced.*
    - 8) ***CNFVB****: computational-normal-floating point-vector-basic.*
    - 9) ***CNFVA****: computational-normal-floating point-vector-advanced.*
    - 10) ***CNFVR****: computational-normal-floating point-vector-reduction.*
    - 11) ***CCIB****: computational-compressed-integer-basic.*
    - 12) ***CCIA****: computational-compressed-integer-advanced.*
    - 13) ***CCFB****: computational-compressed-floating point-basic.*
    - 14) ***CCFA****: computational-compressed-floating point-advanced.*
    - 14) ***...****: computational-...*
    - 15) ***...****: computational-...*
    - 16) ***...****: computational-...*
    - 17) ***...****: computational-...*
2. ***Data transfer****: these instructions perform memory load, store, register move and swap operations.*
    - 18) ***DNSB****: data transfer-normal-scalar-basic.*
    - 19) ***DNSA****: data transfer-normal-scalar-advanced.*
    - 18) ***DNVB****: data transfer-normal-vector-basic.*
    - 19) ***DNVA****: data transfer-normal-vector-advanced.*
    - 20) ***DNVG****: data transfer-normal-vector-gather scatter.*
    - 21) ***DA****: data transfer-atomic.*
    - 22) ***DB****: data transfer-block.*
    - 23) ***DC****: data transfer-compressed.*

3. ***Flow transfer****: these instructions perform conditional branch, jump, call and return operations.*

    - 24) ***FNISB****: flow transfer-normal-integer-scalar-basic.*
    - 25) ***FNISA****: flow transfer-normal-integer-scalar-advanced.*
    - 26) ***FNISM****: flow transfer-normal-integer-scalar-mask.*
    - 27) ***FNIVB****: flow transfer-normal-integer-vector-basic.*
    - 28) ***FNIVA****: flow transfer-normal-integer-vector-advanced.*
    - 29) ***FNFSB****: flow transfer-normal-floating point-scalar-basic.*
    - 30) ***FNFSA****: flow transfer-normal-floating point-scalar-advanced.*
    - 31) ***FNFSM****: flow transfer-normal-floating point-scalar-mask.*
    - 32) ***FNFVB****: flow transfer-normal-floating point-vector-basic.*
    - 33) ***FNFVA****: flow transfer-normal-floating point-vector-advanced.*
    - 34) ***FC****: flow transfer-compressed.*

4. ***System****: these instructions perform system various related operations.*

    - 35) ***SB****: system-basic.*
    - 36) ***SA****: system-advanced.*
    - 37) ***SF****: system-fences.*
    - 38) ***ST****: system-transactional.*

2.2 Miscellaneous modules

Miscellaneous modules are more flexible, they can provide useful features such as debugging, performance counters, user events, system events, etc.... Supervisor modules allow FabRISC to become a privileged architecture to better help with OS implementation. The list of modules is the following:

- 39) ***UEVNT****: user events.*
- 40) ***SEVNT****: system events.*
- 41) ***BPREG****: breakpoint registers.*
- 42) ***PERFC****: performance counters.*
- 43) ***SUPER****: supervisor.*

The high number of modules and extensions is very useful because it allows the hardware designers to only implement what they really want and very little extra. The fact that there is no explicit mandatory subset of the ISA helps can with specialized systems. With this, it becomes perfectly possible to create, for example, a floating-point only processor with no integer instructions to alleviate overheads.

---

3 Low-level data types

This section is dedicated to explain the various proposed low-level data types including integer and floating point. The smallest addressable object in FabRISC is the "byte", that is, eight consecutive bits. Longer types are constructed from multiple bytes side by side following powers of two: one, two, four or eight bytes.

3.1 Integer types

Integer types are arguably the most common data types. Integers are always signed using 2's complement notations and depending on their length, they can have various names:

- ***Byte****: 8 consecutive bits.*
- ***Short****: 16 consecutive bits, or alternatively, 2 bytes.*
- ***Word****: 32 consecutive bits, or alternatively, 4 bytes.*
- ***Long****: 64 consecutive bits, or alternatively, 8 bytes.*

Integer types are manipulated by integer instructions which, by default, behave in a modular fashion. Edge cases, such as wraps-around or overflows can be happen in particular situations:

- ***Carry over****: this situation arises when the absolute value of the result is too big to fit in the desired data type.*
- ***Carry under****: this situation arises when the absolute value of the result is too small to fit in the desired data type.*
- ***Overflow****: this situation arises when the signed value of the result is too big to fit in the desired data type.*
- ***Underflow****: this situation arises when the signed value of the result is too small to fit in the desired data type.*

3.2 Floating point types

...

...

---

4 Memory

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

4.1 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. If aligned is decided to be the only supported scheme, the hart must generate the MISA fault every time the constraint is violated (consult section xxx for more information). Alignment on instructions is the result of the greatest common denominator between the implemented instruction lengths and this constraint must be satisfied by the programmer or compiler, if not, the MISI fault must be generated. Because the possible lengths are only 2, 4 or 6 bytes, the greatest common denominator can only result in a value of 2 or 4. With this it's possible to shift branch offsets by one or two bits respectively to the left before being added to the program counter (PC) which means that said offsets will specify 16 or 32 bits as the smallest addressable object, effectively doubling or quadrupling 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, potentially decrease the memory footprint.

4.2 Synchronization

FabRISC provides dedicated atomic instructions via the DA instruction module 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 and can be used to implement atomic *read-modify-write* and *test-and-set* operations for locks, semaphores and barriers. It is important to note that if the processor can only run one hart at any given moment, then this section can be skipped since the problem can be solved by the operative system. Below is a description of the atomic instructions, which are divided in two categories:

- ***Read-modify-write*** *instructions:*
  - ***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 and marking it as valid.*
  - ***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 the 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 destination register as an indication of the failure. If SC succeeds, the item is written to memory, a one is returned into its register destination and all reservations must be then invalidated.*
- ***Test-and-set*** *instructions:*
  - ***Compare and swap*** *(CAS) is an atomic instruction that conditionally and atomically swaps two values in memory if a particular and specified condition is met. The conditions are equality or less than.*

The LL & SC pair, as briefly mentioned above, necessitate of a transparent register that is used to hold the address of the fetched element. Once the reservation is completed and marked as valid, accesses performed by other cores must be monitored by snooping any other memory access made by other cores. If a memory write request to the matching address is snooped by another core, then that particular reservation must be immediately invalidated. The CAS instruction, in order to be atomic, it must perform it's job all in the same bus request. This ensures that no other core or device has access to the memory potentially changing the target variable in the middle of the CAS instruction.

FabRISC also provides optional instructions, via the ST instruction module, to support basic transactional memory that can be employed instead of the above seen solutions to exploit parallelism in a more "optimistic" manner. Multiple transactions can happen in parallel as long as no conflict is detected by the hardware. when such situations occur, however, the offended transaction must be 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, the OABT exception must be generated to notify the programmer.

- ***Transaction Begin*** *(TBEG): causes the hart that executed this instruction to checkpoint its microarchitectural state and start monitoring accesses by other harts via the coherence protocol as well as incrementing the nesting counter by one. This instruction effectively starts a transaction.*
- ***Transaction End*** *(TEND): causes the hart that executed this instruction to stop monitoring accesses by other harts and commit the changes as well as decrementing the nesting counter by one. This instruction effectively terminates a transaction. The updates to memory can be considered atomic and permanent after the completion of this instruction.*
- ***Transaction Abort*** *(TABT): causes the hart that executed this instruction to stop monitoring accesses by other harts as well as generate an* ***Explicit abort*** *exception within the hart and cause it to restore the microarchitectural state immediately before the latest TBEG as well as decrementing the transaction nesting level counter by one. This instruction effectively aborts a transaction.*
- ***Transaction Check*** *(TCHK): causes the hart 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.*

The ST instruction module necessitates the presence of exceptions (consult section xxx for more information), as briefly mentioned above, called *abort codes* that can be used by the programmer to take the appropriate actions in case the transaction was aborted. Each abort code specifies the reason why the current (most nested) transaction was aborted. As a result of this, the ST module requires the UEVNT module to also be implemented. The ST module extends the UEVNT module adding the following exceptions:

- ***Conflict abort*** *(CABT): the current transaction was aborted because a write on shared variables was detected by the coherence protocol. This exception has an id of xxx.*
- ***Event abort*** *(EABT): the current transaction was aborted because an event, beside the ones in this list, got triggered. This exception has an id of xxx.*
- ***Depth overflow abort*** *(OABT): the current transaction was aborted because it exceeded the upper transaction depth limit. This exception has an id of xxx.*
- ***Replacement abort*** *(RABT): the current transaction was aborted because a cache line was evicted back to memory for not enough associativity. This exception has an id of xxx.*
- ***Size abort*** *(SABT): the current transaction was aborted because a cache line was evicted for back to memory not enough space. This exception has an id of xxx.*
- ***Depth underflow abort*** *(UABT): this abort code is only generated if a TEND instruction is executed and the depth counter is zero. This exception has an id of xxx.*

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 behavior 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. The CAS atomic instruction, even though it suffers from the ABA problem, it guarantees forward progress, rendering this instruction stricter. 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.

4.3 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, such as CACOP, that can be sent to the cache controller directly to manipulate its operation (see section 7 for more 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. All the harts that map to the same core don't need to worry about coherence since the caches are shared between those harts. This argument holds true for whole cores that share bigger pools of cache, such as L2 or L3.

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 and confusions. 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.

4.4 Consistency

FabRISC utilizes a fully relaxed memory consistency model formally known as *release consistency* that allows all possible orderings in order to give harts the freedom to reorder memory instructions to different addresses in any way they want. For debugging and specific situations the stricter *sequential consistency* model can be utilized and the hart 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 by the SF instruction module to let the programmer impose an order on memory operations when the relaxed model is in use. If the hart doesn't reorder memory operations this module is not necessary and can be skipped. The proposed fencing instructions are:

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

The fences can be used on any memory type of instruction, including the LL & SC pair and CAS 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 dedicated implementation specific CACOP instruction as briefly discussed above. The SF module also requires the ability for the hart to switch between the release consistency and the more stringent *sequential consistency* models via the use of a dedicated bit in the control and status register (CSR).

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.
