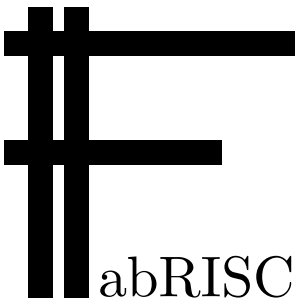

Instruction Set Architecture

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.*

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 Memory

This section is dedicated to the memory model used by FabRISC including data alignment, synchronization, consistency, as well as possible cache coherence directives. All primitive data types must have one of the following sizes:

- ***Byte**: a grouping of continuous 8 bits.*
- ***Short**: a grouping of continuous 16 bits.*
- ***Word**: a grouping of continuous 32 bits.*
- ***Doubleword**: a grouping of continuous 64 bits (for 64 bit machines only).*

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. 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 4 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 compressed instructions, is 16 and the programmer must ensure that the code is aligned at said boundary, if not, the MISI 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 program counter (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, 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 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 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.*

FabRISC also provides optional instructions 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.*

Transactions can generate exceptions, 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.

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

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.

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, 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.

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 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 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 section 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 special implementation specific CACOP instruction as briefly 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, potential DMA behavior 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 hart 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 and 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, is not paged in the traditional sense, that is, the virtual to physical mapping will always be the same, however it must still have page table entries for protection bits. The ISA splits this MMIO address space in two segments:

- ***CPU segment****: this portion, starting from address zero, is composed of 128 bytes and should be used to hold CPU information, such as implemented ISA extensions, cache sizes as well as other CPU capabilities and characteristics.*
- ***IO segment****: this portion, starting from address 128, is composed of 65408 bytes and should be used to communicate with external devices via MMIO.*

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. This region still needs page table entries because of the protection bits, however, i suggest leaving the mappings of virtual to physical the same all the time for simplicity.

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 between the processor and 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.*

Address	Name	Size	Category
00	CPU ID	8	CPU info
07	Implemented ISA modules	4	CPU info
0B	Number of cores	1	CPU info
0C	SMT degree	1	CPU info
0D	WLEN and MXVL	1	CPU info
0E	CPU clock speed	4	CPU info
11	CPU temperature	2	CPU info
13	CPU voltage	2	CPU info
15	TLB information	32	CPU info
35	TLB hierarchy	4	CPU info
38	Cache information	32	CPU info
58	Cache hierarchy	4	CPU info
5B	IO channels	1	IO info
5C	IO clock speed	4	IO info
60	IO voltage	2	IO info
62	Available memory	8	MEM info
6A	Memory channels	1	MEM info
6B	CAS latency	1	MEM info
6C	RAS to CAS latency	1	MEM info
6D	RAS precharge latency	1	MEM info
6E	CMD latency	1	MEM info
6F	Memory clock speed	4	MEM info
73	Memory temperature	2	MEM info
75	Memory voltage	2	MEM info
77	Reserved	9	Reserved

Table 1: CPU segment table.

The DMA protocol or scheme implemented by the hardware designer must also take consistency into account since memory operations to different addresses are allowed to be done out-of-order. This means that fencing instructions must retain their effect from the point of view of the hart and IO devices, which must provide similar fencing features as well.

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, including the devices themselves.

---

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 harts, processes, IO devices, signal system faults or simply trigger software 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 of the executing program and, because of this, the handling must done in program order. This category is further sub divided in two.

- ***Exceptions****: These events are user handled, non promoting and with a global priority level of 0. They are generated by the 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 the 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 and then branching, usually under the form of a case statement, to the proper code.*

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 preemptively without having to perform time consuming checks over and over. Examples of this can be seamless array boundary checks via the use of debugging registers (see section 6 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.

- ***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 device request, which can be achieved in a similar fashion to faults.*
- ***IPC interrupts****: These events are supervisor handled, promoting with a global priority level of 1. They are generated by other harts 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 in a similar fashion to faults and 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 programmer.

4.3 Event handling

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

1. *write the event identifier into the cause register (CAUSE).*
2. *save the value of the program counter (PC) and status register (CSR) into the appropriate buffer registers depending on the privilege level of the event:*
    - *for non-promoting events, simply copy the PC into the exception buffer 0 (EB0) and the CSR into the user exception buffer 1 (UEB1).*
    - *for promoting events, simply copy the PC, into the supervisor event buffer 0 (SEB0), the CSR into the supervisor event buffer 1 (SEB1) and the CAUSE register into the supervisor event buffer 2 (SEB2). The reason for the copy of the cause register is to avoid state corruption since the currently interrupted hart might already be handling an exception.*
3. *mask any other interrupt if the triggered event is promoting, otherwise skip this step. The masking can be accomplished by simply setting the xxx and xxx bits in the CSR to one.*
4. *set the xxx bit of the CSR to one in order to signify that the hart is handling an event.*
5. *switch to supervisor mode if the triggered event is promoting by setting the xxx bit in the CSR, otherwise skip this step.*
6. *jump to the handler location with the use of the appropriate pointer register. Depending of the privilege level of the event this step will involve:*
    - *for non-promoting events, a simple copy of the value of the user event handler pointer (UEHP) into the program counter suffices. From there a case statement can be used in combination with the CAUSE register to determine which event was generated.*
    - *for promoting events, a simple copy of the value of the supervisor event handler pointer (SEHP) into the program counter suffices. From there a case statement can be used in combination with the CAUSE register to determine which event was generated.*

After the execution of the desired handler code, the last instruction will trigger the returning phase. During that phase, it is sufficient to restore the value of the program counter, the status register and, if the event was promoting, the CAUSE register by storing into them the values of the appropriate buffer registers previously written. It is important to note that the processor must always trap preemptively, that is, before the trapping instruction writes the result and, after the return phase, the processor must replay the instruction that caused the event.

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.

Performance counters must be freezed if the triggered event is promoting. If the supervisor needs the performance monitoring counters, it must save them first before unfreezing them. The supervisor must restore the proper values of the counters before actually scheduling a user process for execution. This ensures the retention of values across context switches and event handling.

Handling events should be quite a straight forward process. The launching phase explains all the needed steps to successfully branch to the target location. Once the handler is executed the ERET instruction can be executed to bring back the old state. This mechanism can be used to schedule another process entirely, by setting all the registers to the desired values and executing an ERET instruction which causes the PC and CSR to be written with the values held in the supervisor event buffers. When an event of any kind is triggered, it's necessary to flush or invalidate any in-flight instruction including the very last stage (write-back / commit). This is because it's possible to perform (absolute) branches by simply writing to the program counter directly with a standard move instruction, which means that the control flow can erroneously be steered back causing the execution of wrong instructions if the last stage is left unaltered.

ID	Name	Mnemonic	Type
00	Interrupt 0	INT0	IO interrupt
...	...	...	...
3F	Interrupt 63	INT63	IO interrupt
40	Interrupt 64	INT64	IPC interrupt
...	...	...	...
7F	Interrupt 127	INT127	IPC interrupt
80	System call 0	SYSCL0	Fault
...	...	...	...
E3	System call 99	SYSCL99	Fault
E4	Illegal instruction	ILLI	Fault
E5	Malformed instruction	MALI	Fault
E6	Misaligned instruction	MISI	Fault
E7	Illegal execute address	IEXE	Fault
E8	Illegal read address	IRDA	Fault
E9	Illegal write address	IWRA	Fault
EA	Misaligned address	MISA	Fault
EB	Page fault	PGFT	Fault
EC	Reserved	-	-
...	...	...	...
FF	Reserved	-	-

Table 2: Supervisor event table.

ID	Name	Mnemonic	Type
00	Carry over exception	COVRE	Exception
01	Carry under exception	CUNDE	Exception
02	Overflow exception	OVFLE	Exception
03	Underflow exception	UNFLE	Exception
04	Division by 0 exception	DIV0E	Exception
05	Associativity abort	AABT	Exception
06	Conflict abort	CABT	Exception
07	Event abort	EABT	Exception
08	Depth overflow abort	OABT	Exception
09	Size abort	SABT	Exception
A0	Depth underflow abort	UABT	Exception
A1	Break on instruction address EQ	BIAEQ	Exception
A2	Break on instruction address LE	BIALE	Exception
A3	Break on instruction address GE	BIAGE	Exception
A4	Break on read address EQ	BRAEQ	Exception
A5	Break on read address LE	BRALE	Exception
A6	Break on read address GE	BRAGE	Exception
A7	Break on write address EQ	BWAEQ	Exception
A8	Break on write address LE	BWALE	Exception
A9	Break on write address GE	BWAGE	Exception
AA	Break on instruction 16	BIN16	Exception
AB	Break on instruction 32	BIN32	Exception
AC	Break on instruction 48	BIN48	Exception
AD	Break on carry over	BCOVR	Exception
AE	Break on carry under	BCUND	Exception
AF	Break on overflow	BOVFL	Exception
B0	Break on underflow	BUNFL	Exception
B1	Break on division by 0	BDIV0	Exception

Table 3: User event table.

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 the appropriate fault. The hart should only enter the supervisor mode if an appropriate event is triggered and a dedicated instruction (ERET) is provided to transfer the control back to the user (consult section 7 for more information). If the system makes use of a multicore / multithreaded 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 the xxx bit in the status register. After the booting phase is done, this bit can simply be set to zero causing the processor to execute normally.

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 from damage.

5.2 Virtual memory

FabRISC provides privileged customizable and 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 7 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 CSR 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 or threads. With IPC interrupts it possible to perform:

- ***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 / multithreaded systems. Starting a thread can be achieved by sending a dedicated IPC interrupt that has the ability to wake up the destination core or hardware thread from the halted state. A similar and symmetric picture can employed to stop running cores or threads via a dedicated halting IPC interrupt. An alternative would be to directly talk to the IO controller, which in turn, enables or disables threads.*

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, data transfers as well as thread initialization and termination. 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 micro modules that the hardware designer can choose. The macro modules concern computational, data transfer, control transfer and system instructions:

- ***Computational****: this macro module contains all the instructions that perform arithmetic or logic operations.*
  - *CSIB: computational-scalar-integer-basic*
  - *CSIA: computational-scalar-integer-advanced*
  - *CSIR: computational-scalar-integer-reduced*
  - *CSFB: computational-scalar-FP-basic*
  - *CSFA: computational-scalar-FP-advanced*
  - *CSFR: computational-scalar-FP-reduced*
  - *CVIB: computational-vector-integer-basic*
  - *CVIA: computational-vector-integer-advanced*
  - *CVFB: computational-vector-FP-basic*
  - *CVFA: computational-vector-FP-advanced*
- ***Data transfer****: this macro module contains all the instructions that perform data transfer operations.*
  - *DSB: data-scalar-basic*
  - *DSA: data-scalar-advanced*
  - *DST: data-scalar-atomic*
  - *DSM: data-scalar-multiple*
  - *DSR: data-scalar-reduced*
  - *DVB: data-vector-basic*
  - *DVA: data-vector-advanced*
- ***Jumps****: this macro module contains all the instructions that perform control transfer operations.*
  - *JSIB: jump-scalar-integer-basic*
  - *JSIA: jump-scalar-integer-advanced*
  - *JSIR: jump-scalar-integer-reduced*
  - *JSFB: jump-scalar-FP-basic*
  - *JSFA: jump-scalar-FP-advanced*
  - *JSFR: jump-scalar-FP-reduced*
  - *JVIB: jump-vector-integer-basic*
  - *JVIA: jump-vector-integer-advanced*
  - *JVFB: jump-vector-FP-basic*
  - *JVFA: jump-vector-FP-advanced*
- ***System****: this macro module contains all the instructions that perform specific system operations.*
  - *YB: system-basic*
  - *YA: system-advanced*
  - *YX: system-transactional*

To indicate which ISA modules a particular implementation is composed of, a simple binary number can be used similarly to a checklist. FabRISC allows the hardware designer to choose any combination of micro module, however, for each macro module at least one micro module must be chosen. With this is perfectly legal, for example, to have almost no scalar instructions rendering the underlying processor a potential vector-heavy machine.

6.1 Register file

Depending on which modules are chosen as well as WLEN and MXVL values, the register file can be composed of up to three different banks of variable width. Registers are declared in the following presented lists in order:

1. ***Scalar general purpose registers*** *(SGPRs): this bank is composed of 32 registers which can be used to hold program variables during execution. The registers are all WLEN bits wide and are used by scalar integer and floating point instructions. All of these registers are non privileged.*
2. ***Special purpose registers*** *(SPRs): this bank is composed of 32 registers which are internally used to keep track of state, modes, flags and other system related things. The registers can be WLEN or 32 bits wide with some being privileged resources.*
3. ***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 MXVL bits wide and are used by vector integer and floating point instructions. This bank is only necessary if the machine in question supports vector execution. All of these registers are non privileged.*

6.1.1 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:

- ***Parameter registers****: 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-down, while returning values are stored starting from the bottom-up.*
- ***Persistent registers****: 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.*
- ***Volatile registers****: 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.*
- ***Global pointer*** *(GP): this register is used to point to the global variable area.*
- ***Stack pointer*** *(SP): this register is used as a pointer to the call-stack.*
- ***Return address*** *(RA): this register is used to hold the return address for the currently executing function call.*

Vector registers are all considered persistent, which means that the callee save scheme must be utilized since it's assumed that their value will be retained across function calls. Special instructions are also provided to move vector registers, or part of them, to and from the scalar bank (see section 7 for more information).

Scalar	Special	Vector
P0	DBG0	V0
P1	DBG1	V1
P2	DBG2	V2
P3	DBG3	V3
P4	DBG4	V4
P5	DBG5	V5
S0	DBG6	V6
S1	DBG7	V7
S2	DBG8	V8
S3	DBG9	V9
S4	DBGM0	V10
S5	DBGM1	V11
S6	EB0	V12
S7	EB1	V13
S8	UEHP	V14
S9	VMSK	V15
S10	PC	V16
S11	CSR	V17
S12	CAUSE	V18
S13	SEB0	V19
S14	SEB1	V20
S15	SEB2	V21
S16	SEHP	V22
S17	PTP	V23
N0	PID	V24
N1	FU0	V25
N2	FU1	V26
N3	FU2	V27
N4	K0	V28
GP	K1	V29
SP	K2	V30
RA	K3	V31

Table 4: Register file.

6.1.2 Special purpose register bank

The special purpose register bank, as mentioned earlier, is composed or 32 registers of different width 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 independent registers rather than an actual file. The SPRs are organized as follows:

- ***Debugging registers****: registers prefixed with "DBG" are used for debugging and other support features such as real-time counters and performance metrics. 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. It's worth noting that when used as counters, DBG registers will only be real-time for the current running process.*
- ***Debugging mode registers****: 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 six bit code reserved in its corresponding DBGM register to indicate the operating mode. Registers DBG0 to DBG4 will map to DBGM0 while registers DBG5 to DBG9 will map to DBGM1. All DBGM registers are always 32 bits wide and are not privileged.*

- ***Exception buffers**: registers prefixed with "EB" are used as temporary buffers for exception handling. They are transparent registers that are automatically managed by the hart 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.*
- ***User event handler pointer**: this register, called "UEHP", is used to hold the logical address of the exception handler for the current executing process. During the launching phase, UEHP will be used to perform an absolute branch (copy of UEHP value plus exception identifier into the PC). UEHP is WLEN bits wide and is not privileged.*
- ***Vector mask**: this register, called "VMSK" is used to hold the vector mask. A single bit signifies a mask for its corresponding byte of the vector result. Dedicated instructions are provided to directly set and manipulate the value of the VMSK register. The VMSK register is always WLEN bits wide and is not privileged.*
- ***Program counter**: this register, called "PC", is the program counter. It is used to point to the currently executing instruction. PC is WLEN bits wide and is not privileged.*
- ***Control and status register**: this register, called "CSR", is used to hold the hart supervisor and system state (see table below for more information). CSR is 32 bits wide and is privileged.*
- ***Event cause register**: this register, called "CAUSE", is used to hold the identifier of the latest occurring event. This register can be used inside the handler code to perform a case statement in order to execute the correct action. CAUSE is 32 bits wide and is privileged.*
- ***Supervisor event buffers**: 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) and SEB2 the event cause register (CAUSE). All SEB register are WLEN bits wide and are privileged.*
- ***Supervisor event handler pointer**: this register, called "SEHP", is used to hold the logical address of the supervisor event branch table. During the launching phase, SEHP will be used to perform an absolute branch (copy the value into the PC). SEHP is WLEN bits wide and is privileged.*
- ***Page table pointer**: this register, called "PTP", 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.*
- ***Process identifier**: this register, called "PID", 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. PID is WLEN bits wide and is privileged.*
- ***File usage**: registers prefixed with "FU" are used to hold information about which register in which bank was modified since the scheduling of the process. FU registers are used by specific instructions to store or load only the necessary registers for context switches. This is achieved by the use of a single bit per register to indicate if the target register is modified or not. FU0 is reserved for bank 0, FU1 for bank 1, FU2 for bank 2 and FU3 for bank 3. All FU registers are 32 bits and are privileged.*
- ***Kernel register**: registers prefixed with "KR" are kernel reserved general purpose registers. KR registers are all WLEN bits wide and are privileged.*

The CSR register is divided into several bits and flags fields in order to provide granular control over the supervisor mode state. If the micro-architecture doesn't support certain features, then the corresponding bits for those features can be ignored. The first 18 bits can be accessed in user mode while the remaining bits are privileged only. The bits are listed from 0 to 31 in the order that they appear:

- ***Instruction behavior section**: reserved for storing instruction behavior bits such as vector length, floating point rounding modes, etc. . . :*

- ***Vector length*** *(VLEN): six bits that indicate how many elements a vector instruction will operate on. A value of zero will signify only one element, while a value of 127 will signify all of the elements.*
- ***Rounding modes*** *(RMD): two bits that specify the floating point rounding modes:*
  1. *round towards $+\infty$.*
  2. *round towards $-\infty$.*
  3. *round towards zero.*
  4. *round towards even.*
- ***Scalar mask*** *(SMSK): single bit that determines the mask for scalar instructions. A value of zero will allow the result to be written, while a value of one will prevent the instruction to alter any state.*
- ***Auto exceptions bit*** *(AEXC): this bit indicates if the hart should trap every time a trapping arithmetic flag is set or not.*

- ***Transactional memory section****: reserved for storing the eight bit transaction nesting depth counter (TND). Every time a transactional memory instruction is executed, this counter may be modified by either incrementing or decrementing. If the counter overflows or underflows, the OABT or UABT exceptions must be triggered accordingly.*
- ***Supervisor bit*** *(SUPB): this bit indicates if the hart is in supervisor mode or not.*
- ***Boot bit*** *(BOOT): this bit indicates if the hart is in boot mode or not.*
- ***Consistency bit*** *(CONS): this bit indicates what memory consistency model to use.*
- ***Paging bit*** *(PAGB): this bit indicates if the hart should perform address translation via paging.*
- ***IPC interrupts mask*** *(IPCM): this bit indicates if the IPC interrupts are masked or not.*
- ***IO interrupts mask*** *(IOIM): this bit indicates if the IO interrupts are masked or not.*
- ***Trapped bit*** *(TRAP): this bit indicates if the hart is currently handling a promoting event. This bit can be red by the IO controller to better understand where to route incoming interrupts.*
- ***Halt bit*** *(HLTB): this bit indicates if the hart is halted or not.*
- ***Cache bit*** *(CACB): this bit indicates if the hart can access cache or not.*
- ***reserved****: the remaining bits are unused.*

6.1.3 Processor operating modes

FabRISC defines multiple operating modes specified in CSR and FR which can be changed by simply manipulating the value of said register. This is a further explanation of the above lists:

- ***Supervisor mode****: this CSR bit is automatically set by the hardware when an appropriate event is triggered and it dictates if the hart is in supervisor or user mode. For simpler implementations that don't want to support privilege levels, this can be omitted.*
- ***Boot mode****: this CSR bit dictates which memory device to fetch instructions from to make boot loaders possible with the help of an external ROM. For simpler implementations that don't want to support this mechanism, this bit can be omitted.*
- ***Consistency mode****: this CSR bit dictates the memory consistency model currently being employed and, if set, it causes the hart to execute memory instructions in program order effectively forcing the sequential consistency model. For simpler implementations that don't reorder memory instructions, this bit can be omitted.*

- ***Paged memory mode****: this CSR bit dictates the memory management model currently being employed and, if set, it causes the hart to perform virtual to physical translation, effectively enabling paged memory. For simpler implementations that don't support paging, this bit can be omitted.*
- ***Auto exceptions mode****: this FR bit dictates the exception handling mechanism currently being employed and, if set, it causes the hart to branch to the handler whenever an exception is triggered. If the bit is not set, the hart discards any exception limiting itself to setting the flags only.*
- ***Cached mode****: this CSR bit dictates if a particular hart is allowed to use caches as part of it's operation. A value of zero will cause the hart to bypass the whole cache hierarchy.*

The special purpose bank, as explained above, contains transparent debug registers which can be written with a value that is constantly being tested at every cycle. DBGM registers hold the six bit configuration code for each individual debugging register for a total of 64 possible modes. There are 16 proposed modes to help with debugging, exception handling and boundary checks. The remaining combinations are left as microarchitecture specific performance monitoring modes.

1. ***Disabled****: The value of won't be used for anything. If a DBG register is in this state, it can be used as an extra volatile GPR.*
2. ***instruction address EQ****: Trap if an instruction with an address equal to the stored value in the corresponding DBG register is fetched. Exception code is xxx.*
3. ***instruction address LE****: Trap if an instruction with an address less or equal to the stored value in the corresponding DBG register is fetched. Exception code is xxx.*
4. ***instruction address GE****: Trap if an instruction with an address greater or equal to the stored value in the corresponding DBG register is fetched. Exception code is xxx.*
5. ***read address EQ****: Trap if a memory read is performed at an address equal to the stored value in the corresponding DBG register. Exception code is xxx.*
6. ***read address LE****: Trap if a memory read is performed at an address less or equal to the stored value in the corresponding DBG register. Exception code is xxx.*
7. ***read address GE****: Trap if a memory read is performed at an address greater or equal to the stored value in the corresponding DBG register. Exception code is xxx.*
8. ***write address EQ****: Trap if a memory write is performed at an address equal to the stored value in the corresponding DBG register. Exception code is xxx.*
9. ***write address LE****: Trap if a memory write is performed at an address less or equal to the stored value in the corresponding DBG register. Exception code is xxx.*
10. ***write address GE****: Trap if a memory write is performed at an address greater or equal to the stored value in the corresponding DBG register. Exception code is xxx.*
11. ***instruction trap 16****: Trap if an instruction whose 16-bit pattern corresponds to the stored value in the corresponding DBG register is fetched. Exception code is xxx.*
12. ***instruction trap 32****: Trap if an instruction whose 16-bit pattern corresponds to the stored value in the corresponding DBG register is fetched. Exception code is xxx.*
13. ***instruction trap 48****: Trap if an instruction whose 16-bit pattern corresponds to the stored value in the corresponding DBG register is fetched. Exception code is xxx.*
14. ***COVR trap****: Trap if an instruction whose address corresponds to the stored value in the corresponding DBG register generates a COVR flag. Exception code is xxx.*
15. ***CUND trap****: Trap if an instruction whose address corresponds to the stored value in the corresponding DBG register generates a CUND flag. Exception code is xxx.*

16. ***OVFL trap**: Trap if an instruction whose address corresponds to the stored value in the corresponding DBG register generates a OVFL flag. Exception code is xxx.*

17. ***UNFL trap**: Trap if an instruction whose address corresponds to the stored value in the corresponding DBG register generates a UNFL flag. Exception code is xxx.*

18. ***DIV0 trap**: Trap if an instruction whose address corresponds to the stored value in the corresponding DBG register generates a DIV0 flag. Exception code is xxx.*

The entirety of the register file is divided into three banks with several registers each. This can sound like a lot of state (and it kinda is), but other ISAs such as full RISC-V implementation would arguably contain more because of the way CSRs are implemented. In FabRISC, the first bank offers a 32 entry flat register file in order to efficiently support modern graph-coloring driven compiler register allocation. Integer and floating-point registers are shared to ease conversions and the extra GPRs found in the special purpose bank can alleviate some of the pressure caused by the sharing. Because there is no hardware stack pointer nor return address register, standard calling conventions can be used to give a "purpose" to all the GPRs in the bank. Compressed instructions, because of the size limitations, are only able to access eight registers of the bank. The second bank is where the "magic" happens: special registers are present to help with debugging, memory safety as well as keeping track of state and resource usage to reduce the overhead on context switches. File usage registers help with that, for example, if no vector register was written during the time quantum of a particular process, then there is no need to save them again.

6.2 Instruction formats

FabRISC specifies 15 different instruction formats to allocate as much space as possible to the operands, while leaving some opcode encoding space to implement extra application specific instructions. Formats can be variable length from two up to six bytes in order to accommodate larger immediate constants for addresses and data. Formats are divided into several bit fields that are scrambled around in order to reduce decoding logic. Many also carry with them a "modifier" field, called mod, that can provide extra information such as mask, data type length, vector mode and more. All instructions treat data as signed unless explicitly noted.

[47:32]	[31:27]	[26]	[25]	[24:22]	[21:17]	[16]	[15]	[14:12]	[11:10]	[9:7]	[6:0]	Format
							mod	rb	mod	ra	opc	H
							imm	imm	imm	ra	opc	I
							imm	imm	imm	imm	opc	J
							opc	opc	opc	opc	opc	K
	opc	mod	mod	mod	rc	rb	rb	rb	ra	ra	opc	A
	imm	opc	mod	mod	imm	rb	rb	rb	ra	ra	opc	B
	opc	opc	mod	opc	mod	rb	rb	rb	ra	ra	opc	C
	imm	imm	imm	mod	imm	rb	rb	rb	ra	ra	opc	D
	imm	imm	imm	mod	rc	rb	rb	rb	ra	ra	opc	E
	rd	mod	mod	mod	rc	rb	rb	rb	ra	ra	opc	F
	imm	imm	mod	mod	imm	imm	imm	imm	ra	ra	opc	G
imm	imm	opc	mod	mod	imm	rb	rb	rb	ra	ra	opc	B.l
imm	imm	mod	mod	mod	imm	rb	rb	rb	ra	ra	opc	D.l
imm	imm	mod	mod	mod	rc	rb	rb	rb	ra	ra	opc	E.l
imm	imm	imm	mod	mod	imm	imm	imm	imm	ra	ra	opc	G.l

Table 5: Formats table.

- ***opc**: short for instruction opcode.*
- ***mod**: short for instruction modifier.*
- ***ra, rb, rc, rd**: short for register a, b, c, d which are register specifiers.*
- ***imm**: short for immediate.*

The instruction modifier bit field depends on the particular class of instructions in question. Each format can have different modifier classes to best fit particular groups of instructions:

- ***A**: this format has three instruction modifier classes:*
  1. ***tt m vv**: this class is used by scalar computational instructions. 'tt' encodes the data type length: 1, 2, 4 or 8 bytes, 'm' states if the instruction is masked or not and 'vv' indicates the instruction mode: scalar, vector-vector or vector-scalar.*
  2. ***tt m –**: this class is used by vector gather and scatter instructions. 'tt' encodes the data type length: 1, 2, 4 or 8 bytes and 'm' states if the instruction is masked or not.*
  3. ***tt m ff**: this class is used by the CAS instruction. 'tt' encodes the data type length: 1, 2, 4 or 8 bytes, 'm' states if the instruction is masked or not and 'ff' states the condition to check: EQ, NE, LT or LE.*
- ***B and B.l**: this format has one instruction modifier class:*
  1. ***tt m v**: this class is used by scalar and vector computational instructions. 'tt' encodes the data type length: 1, 2, 4 or 8 bytes, 'm' states if the instruction is masked or not and 'v' indicates the instruction mode: scalar or vector-scalar.*
- ***C**: this format has four instruction modifier classes:*
  1. ***tt m —**: this class is used by scalar computational instructions. 'tt' encodes the data type length: 1, 2, 4 or 8 bytes and 'm' states if the instruction is masked or not.*
  2. ***tt m - tt**: this class is used by casts and conversion instructions. 'tt' encodes the data type length: 1, 2, 4 or 8 bytes, 'm' states if the instruction is masked or not.*
  3. ***tt m nnn**: this class is used by move and swap multiple instructions. 'tt' encodes the data type length: 1, 2, 4 or 8 bytes, 'm' states if the instruction is masked or not and 'nnn' specifies how many registers are acted on, from one to eight inclusive.*
  4. ***tt m s ff**: this class is used by mask setting on compare instructions. 'tt' encodes the data type length: 1, 2, 4 or 8 bytes, 'm' states if the instruction is masked or not, 's' states if the instruction operates on signed or unsigned data and 'ff' states the condition to check: EQ, NE, LT or LE.*
- ***D and E**: these formats have two instruction modifier classes:*
  1. ***tt m**: this class is used by scalar memory instructions. 'tt' encodes the data type length: 1, 2, 4 or 8 bytes and 'm' states if the instruction is masked or not.*
  2. ***- m a**: this class is used by vector memory instructions. 'a' encodes the addressing mode: standard or striding and 'm' states if the instruction is masked or not.*
- ***D.l, E.l and F**: these formats have two instruction modifier classes:*
  1. ***tt m uu**: this class is used by scalar memory instructions. 'tt' encodes the data type length: 1, 2, 4 or 8 bytes, 'm' states if the instruction is masked or not and 'uu' specifies what auto update mode to use: nothing, post-increment, post-decrement or pre-decrement.*
  2. ***– m a uu**: this class is used by vector memory instructions. 'a' encodes the addressing mode: standard or striding, 'm' states if the instruction is masked or not and 'uu' specifies what auto update mode to use: nothing, post-increment, post-decrement or pre-decrement.*
- ***G and G.l**: this format has four instruction modifier classes:*

1. ***rr m -****: this class is used by block memory instructions. 'rr' specifies which register file: SGPRs, SPRs or VGPRs and 'm' states if the instruction is masked or not*
2. ***ff p -****: this class is used by test branch instructions. 'ff' specifies the condition: EQ, NE, LT or LE and 'p' states if the instruction mode: integer or floating point.*
3. ***iiii****: this class is used by direct function calls and jump instructions. 'iiii' extends the offset by four bits.*
4. ***e h m s****: this class is used by special bit-field manipulation instructions. 'e' specifies how the immediate is filled: zero-filled or one-filled. 'h' specifies the immediate highlighting mode: zeros ignore and ones select or zero clear and ones select. 'm' states if the instruction is masked or not and 's' signifies if the immediate is shifted by 32 positions to the left or not.*

- ***H****: this format has xxx instruction modifier classes: ...*

The moderate number of formats arise from the fact that i wanted to "best-fit" several groups of instructions in order to leave as much space as possible to the operands without hindering the opcode space. I believe, however, that the decoding complexity is still very manageable due to the high degree of similarity among the formats. Some times is just a few bits changing purpose and or position. The register specifiers are always in the same position and the MSB of immediates are also always in the same position to reduce decoding complexity. The first seven bits, although sometimes not enough for the full opcode, will always be enough to encode the format and, therefore, the length of the instruction.

6.3 Decoding guidelines

This sub section is dedicated to give hints and suggestions to how to decode instructions for this particular ISA. It is important to remember that this is only a suggestion and it's not mandatory to strictly perform the decoding in the presented manner. Nevertheless, the proposed decoding strategy for a one instruction is divided into multiple steps:

1. *retrieve the opcode.*
2. *use the retrieved opcode to index the microcode read only memory.*
3. *use some of the microcode output to route the operand fields.*
4. *use some of the microcode output to insert the correct operation codes.*
5. *emit the composed micro-instruction.*

The micro-instruction is supposed to be a more "digested" macro-instruction that is much more hardware friendly while still retaining some information density in order to reduce it's size. The micro-instruction can be further decoded down the pipeline to extract the raw control lines for the individual functional units.

6.3.1 Opcode extraction

Instruction opcodes can be of variable length of 7, 8, 12 and 16 bits. The length of the opcode can be retrieved by looking at the first 7 seven bits. From there, with the help of a hard coded logic array, the full opcode can be obtained. Once fully composed, the opcode can then be used to index into the microcode ROM where information such as the format is held. Opcodes are distributed in the following manner:

- ***7 bits*** *0000000xxxxxxxxx → 1011011xxxxxxxxx with a total of 92 combinations.*
- ***8 bits*** *10111000xxxxxxxx → 11110111xxxxxxxx with a total of 64 combinations.*
- ***12 bits*** *111110000000xxxx → 111110111111xxxx with a total of 64 combinations.*
- ***16 bits*** *1111110000000000 → 1111110001111111 with a total of 128 combinations.*

The total size of the encoding space amounts to 348 instructions, however, only xxx opcodes are used in total. It's possible to reserve one entry in the microcode ROM for each possible instruction but opcodes are assigned in a specific way to reduce the size of the ROM. Instructions that fundamentally perform the same operation but on different data types such as integer and floating point will map to the same entry, compressing the encoding space to just under 256 entries. This compression requires a hard coded logic array that composes the target 8 bit ROM address. Some instructions might require some form of sequencing which can be achieved by either emitting multiple micro-instructions or letting the target functional unit handle the sequencing. It's important to note that the sequencing required is very simple because the instructions that need it effectively perform the same operation but multiple times, for example: block moves simply copy multiple registers. If it's decided to sequence the actual microcode, then the remaining entries of the ROM can be used for that very purpose.

[opcode compression table here...]

6.3.2 Micro-operation format

[Coming soon...]

The most expensive part of the decoding process is the micro-code ROM, which is going to roughly be 1024 bytes assuming a 32 bit entries. This might seem a lot, but most FPGAs have dedicated memory blocks that can cover the needed size. Block rams are quite fast and are often multi-ported to allow simultaneous access from different locations. This means that one block-ram is enough even for super-scalar micro-architectures where multiple instructions are decoded in parallel. The micro-instruction is the result of the decoding process, which holds all the information needed for later steps such as register and operation specifiers as well as other control signals all in one single object. As mentioned above, complex instructions might need to be "unrolled" or expanded into more than one micro-instruction before the processor advances to the next macro-instruction.

---

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 macro module and micro module. Each micro module will list its instructions in alphabetical order.

7.1 Basic scalar integer computational

This subsection is dedicated to simple scalar integer computational instructions. This module has the code name "BSIC" (basic scalar integer computational) and includes simple arithmetic and logic integer operations.

7.1.1 (ADD) Addition

This instruction computes $ra = rb + rc$. ADD is unprivileged; has an opcode of xxx and belongs to the A format. Updated arithmetic flags are the following: COVR, CUND, OVFL, UNFL, ZERO and SIGN. The remaining arithmetic flags are reset to zero. The applied modifier class is 1.

7.1.2 (SUB) Subtraction

This instruction computes $ra = rb - rc$. SUB is unprivileged; has an opcode of xxx and belongs to the A format. Updated arithmetic flags are the following: COVR, CUND, OVFL, UNFL, ZERO and SIGN. The remaining arithmetic flags are reset to zero. The applied modifier class is 1.

[to be continued...]

7.2 Data transfer instructions

[Coming soon...]

7.3 Control transfer instructions

[Coming soon...]

7.4 System instructions

[Coming soon...]

8 Macro-op fusion

This section is dedicated to give a brief explanation of the macro-op fusion technique. Macro-op fusion can help to increase performance by combining common instruction idioms, usually just pairs of instructions, into one micro-op that can be executed in a single cycle by the back-end of the processor. A fixed and hard-coded set of fusible instruction sequences is needed to tell the hardware when the fusion is applicable and when it is not, however, the only constraint is that a fused sequence must behave in the same way as a non fused sequence of the same instructions. In short, breaking a fusible sequence with a NOP or other non fusible instruction should not yield different architectural states. Some sequences are listed below, but more are possible:

1. ...

[to be continued...]

This is a very interesting technique that can help improve performance, especially in cases where the hardware is capable of doing certain operations in one cycle, but the ISA can't. The instruction decoder, though, needs to be designed with this capability in mind and the use of a micro-op cache greatly helps minimizing the pipeline bubbles created by the fusion. The micro-op cache also helps reducing power because it stores decoded instructions avoiding re decoding them every time the same code is executed. Fetching branches from this cache reduces their penalty in case of incorrect prediction.

---

9 License

Official GitHub page of the FabRISC project: ...