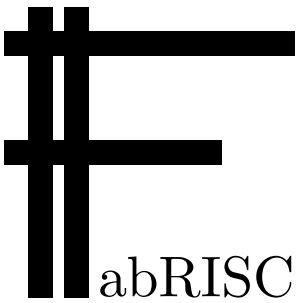

Instruction Set Architecture

Enrico Gatto Monticone

11th, May 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. With very simple implementations, it's possible to realize processors of 16 and 8-bit width as well. The ISA can be further expanded and enriched by allocating unused opcode combinations to custom application specific instructions. FabRISC is completely free, open-source and available to anyone interested in the project via the official GitHub: `https://github.com/Clamentos/FabRISC` (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 shorter "compressed" ones 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 in both the documentation and the software running. These parameters are actually stored in some regions of memory so that programs can gather information about various characteristics of the system (see section 5 for more information):

- ***Word Length*** *(WLEN): this parameter indicates the natural scalar word length of the processor in bits, for example, a 64-bit CPU will have WLEN of 64.*
- ***Maximum Vector Length*** *(MXVL): this parameter indicates the maximum vector length of the processor 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 currently 43 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 stations and ROB 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 WLEN 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. The following, is the list of all modules:

1. ***Computational****: these modules include instructions that perform arithmetic or logic operations:*
    - 1) ***CISB****: computational-integer-scalar-basic.*
    - 2) ***CISA****: computational-integer-scalar-advanced.*
    - 3) ***CIVB****: computational-integer-vector-basic.*
    - 4) ***CIVA****: computational-integer-vector-advanced.*
    - 5) ***CIVR****: computational-integer-vector-reduction.*
    - 6) ***CFSB****: computational-FP-scalar-basic.*
    - 7) ***CFSA****: computational-FP-scalar-advanced.*
    - 8) ***CFVB****: computational-FP-vector-basic.*
    - 9) ***CFVA****: computational-FP-vector-advanced.*
    - 10) ***CFVR****: computational-FP-vector-reduction.*
    - 11) ***CCIB****: computational-compressed-integer-basic.*
    - 12) ***CCIA****: computational-compressed-integer-advanced.*
    - 13) ***CCFB****: computational-compressed-FP-basic.*
    - 14) ***CCFA****: computational-compressed-FP-advanced.*
    - 15) ***CSM****: computational-scalar-mask.*
2. ***Data transfer****: these modules include instructions that perform memory load, store, register move and swap operations:*
    - 16) ***DSB****: data transfer-scalar-basic.*
    - 17) ***DSA****: data transfer-scalar-advanced.*
    - 18) ***DVB****: data transfer-vector-basic.*
    - 19) ***DVA****: data transfer-vector-advanced.*
    - 20) ***DVG****: data transfer-vector-gather scatter.*

- 21) ***DC**: data transfer-compressed.*
- 22) ***DB**: data transfer-block.*

3. ***Flow transfer**: these modules include instructions that perform conditional branch, jump, call and return operations:*
   - 23) ***FISB**: flow transfer-integer-scalar-basic.*
   - 24) ***FISA**: flow transfer-integer-scalar-advanced.*
   - 25) ***FISM**: flow transfer-integer-scalar-mask.*
   - 26) ***FIV**: flow transfer-integer-vector.*
   - 27) ***FFSB**: flow transfer-FP-scalar-basic.*
   - 28) ***FFSA**: flow transfer-FP-scalar-advanced.*
   - 29) ***FFSM**: flow transfer-FP-scalar-mask.*
   - 30) ***FFVB**: flow transfer-FP-vector.*
   - 31) ***FC**: flow transfer-compressed.*
4. ***System**: these modules include instructions that perform system various related operations:*
   - 32) ***SB**: system-basic.*
   - 33) ***SA**: system-advanced.*
5. ***Miscellaneous**: these modules include miscellaneous components such as extra instructions, register banks, events and more:*
6. 34) ***FNC**: fences.*
7. 35) ***TM**: transactional-memory.*
8. 36) ***AM**: atomic-memory.*
9. 37) ***EXC**: exceptions.*
10. 38) ***FLT**: faults.*
11. 39) ***IOINT**: IO interrupts.*
12. 40) ***IPCINT**: IPC interrupts.*
13. 41) ***DBGR**: debugging registers.*
14. 42) ***PERFC**: performance counters.*
15. 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. This method makes possible the implementation of silly things such as having no flow transfer instructions, thus it kind of relies on the common sense of the hardware designers when it comes to realize sensible microarchitectures.

---

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 in *little-endian* order.

3.1 Integer types

Integers are arguably the most common data types. They 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.*

Values that serve as pointers are also manipulated as signed two's complement integers. Although the concept of sign doesn't make much sense for addresses, signed arithmetic can still be applied without many problems. Addition and subtraction will always yield the same exact bit pattern regardless of the interpretation of the operands. Multiplication can still produce the same pattern as well but only if the result is WLEN long, which means ignoring the upper WLEN bits. The only comparisons that do not depend on sign are equality and inequality because they simply amount to checking if each individual bit of one operand is equal or not to the ones in the other operand. Other comparisons, such as less than or greater than are risky if the object is close to the sign boundary from 0x7FFFFFFF ... F to 0x80000000 ... 0.

3.2 Floating point types

[coming soon...]

The low level data types are more or less the usual ones. Interesting and, perhaps, annoying situations can arise when manipulating addresses because all integer values are treated as signed. Usually though, pointer arithmetic is not something that should be heavily relied on because some operations are deemed illegal such as multiplication, division, modulo and bitwise logic. The sign boundary on 64 bit systems can be considered a non issue since the address space is absolutely huge and everything could fit into one of the two partitions. Systems with smaller WLEN might encounter some difficulties but some amount of pointer arithmetic can still be done without problems. [FP types comment...]

---

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 6 for more information). When it comes to instructions, it's mandatory to have fetch engines that support accesses aligned at the 16-bit boundary. 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.

4.2 Synchronization

FabRISC provides dedicated atomic instructions via the AM 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 and SC pair, as briefly mentioned above, necessitates of a transparent register that is used to hold the address of the fetched element as well as its size. Once the reservation is completed and marked as valid, accesses performed by other cores must be monitored via snooping or directory coherence. 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, 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 operation.

FabRISC also provides optional instructions, via the TM 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 TM instruction module necessitates the presence of exceptions (consult section 6 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 TM module requires the presence of the same special buffer registers as the EXC module. The TM module adds the following extra exception events to the architecture:

- ***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 05.*
- ***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 06.*
- ***Depth overflow abort*** *(OABT): the current transaction was aborted because it exceeded the upper transaction depth limit. This exception has an id of 07.*
- ***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 08.*
- ***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 09.*
- ***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 0A.*

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, via the SA instruction module, implementation-dependent instructions, such as CACOP, that can be sent to the cache controller directly to manipulate its operation (see section 9 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 FNC 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 and 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 FNC module also requires the ability for the hart to switch between the release consistency and the more stringent sequential consistency model 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.

---

5 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.

5.1 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 of the MMIO space, 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 of the MMIO space, is composed of 65408 bytes and should be used to communicate with external devices via MMIO.*

Address	Name	Size	Category
00	CPU ID	8	CPU info
08	Implemented ISA modules	8	CPU info
10	Number of cores	1	CPU info
11	SMT degree	1	CPU info
12	WLEN and MXVL	1	CPU info
13	CPU clock speed	4	CPU info
17	CPU temperature	2	CPU info
19	CPU voltage	2	CPU info
1B	TLB information	32	CPU info
3B	TLB hierarchy	4	CPU info
3F	Cache information	32	CPU info
5F	Cache hierarchy	4	CPU info
63	IO channels	1	IO info
64	IO clock speed	4	IO info
68	IO voltage	2	IO info
6A	Available memory	8	MEM info
72	Memory channels	1	MEM info
73	CAS latency	1	MEM info
74	RAS to CAS latency	1	MEM info
75	RAS precharge latency	1	MEM info
76	CMD latency	1	MEM info
77	Memory clock speed	4	MEM info
7B	Memory temperature	2	MEM info
7D	Memory voltage	2	MEM info
7F	Reserved	1	Reserved

Table 1: CPU segment table.

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.

5.2 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.*

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 via the use of atomic and fencing operations.

---

6 Events

This section is dedicated to the specification of exceptions, faults and interrupts which are needed to implement the EXC, FLT, IOINT, IPCINT, DBGR and TM modules. 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.

6.1 Synchronous events

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 unless noted otherwise. This category is further sub divided in two:

- ***Exceptions****: these events are user handled, non promoting and with a global priority level of 0 through 2. 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 all within the address space of the process in question.*
- ***Faults****: these events are supervisor handled, promoting and with a global priority level of 4. They are generated by the executing instructions and each process will have the same handler dictated 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 part of the OS 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 8 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.

6.2 Asynchronous events

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 5. They are generated by external IO devices and can have a dynamic 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 3. They are generated by other harts in the system in order to communicate with each other (inter-process communication) and can have a dynamic 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. IPC interrupts are intended for communication between different processes or threads. Conceptually, they work in a similar fashion to IO interrupts with the only difference being that IPC interrupts are generated by other processes or threads.

6.3 Event modules requirements

In order to be able to handle any kind of event, the hardware must provide the CAUSE register which is used to store the unique identifier of the triggered event. If multiple event modules are implemented, multiple instances of the same requirements are, obviously, needed just once. A list containing all the possible events is provided at the bottom of this section.

- ***EXC module****: this module adds exceptions to the architecture. The hardware must implement the EB0, EB1, EB2 and UEHP registers that are used as buffers when an exception is triggered.*
- ***TM module****: this module adds aborts code to the architecture. The hardware must implement the EB0, EB1, EB2 and UEHP registers that are used as buffers when an exception is triggered. Other requirements for this module can be found in section 4.*
- ***DBGR module****: this module adds breakpoints to the architecture. The hardware must implement the EB0, EB1, EB2 and UEHP registers that are used as buffers when an exception is triggered. Other requirements for this module can be found in section 8.*
- ***FLT module****: this module adds faults to the architecture. The hardware must implement the SEB0, SEB1, SEB2 and SEHP registers that are used as buffers when a fault is triggered. Faults are a promoting event and relies on the presence of the supervisor.*
- ***IOINT module****: this module adds IO interrupts to the architecture. The hardware must implement the SEB0, SEB1, SEB2 and SEHP registers that are used as buffers when an IO interrupt is triggered. IO interrupts are a promoting event and relies on the presence of the supervisor.*
- ***IPCINT module****: this module adds IPC interrupts to the architecture. The hardware must implement the SEB0, SEB1, SEB2 and SEHP registers that are used as buffers when an IPC interrupt is triggered. IPC interrupts are a promoting event and relies on the presence of the supervisor.*

If the supervisor is not decided to be implemented, the FLT, IOINT and IPCINT modules can still be implemented without it by hard-coding the handlers in memory locations that are always the same. This alleviates the requirements for simpler machines such as micro controllers which don't need or require any privileged ISA capability because they, usually, do not run any operative system. Some events of the FLT module are mandatory, specifically the ILLI, MALI, MISI and INCI events are mandatory and must always be implemented.

6.4 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. *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), the CSR into the exception buffer 1 (EB1) and the CAUSE register into the exception buffer 2 (EB2).*
    - *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).*
2. *write the event identifier into the cause register (CAUSE).*
3. *disable the handling of any further event depending on the type:*
    - *if the event is an exception, disable the auto exception mode by setting the AEXC and the DBGF bits in the CSR to zero and one respectively in order to suppress any further exception event.*
    - *if the event is an interrupt, mask any other potential incoming interrupt by setting the IPCM, IOIM, DBGF and CNTF bits in the CSR to one for all of them in order to suppress any other unwanted event, as well as freezing any potential active performance counter.*
4. *set the TRAP 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 SUPB 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.*

Step number 3 is critical in avoid potential corruption of the state while the handler is saving important registers to memory. In order to allow nesting of exceptions, once the critical user registers (exception buffers) have been saved to the call stack in main memory, simply re-enabling the AEXC and disabling the DBGF bits will allow any pending exception to trap the hart again re-triggering the handling procedure. Exceptions triggered during this period of "blindness" might be queued to avoid loss of information. A similar picture can be painted for interrupts and the queuing might be performed by the IO controller or in the core internally as well. The supervisor is allowed to interrupt the user at any time without corrupting the user state since the critical registers are saved into the supervisor event buffers. Nesting faults can be also accomplished in the same way as exceptions, however it's possible to come across the "double fault" edge case which is going to be explained in more detail shortly. After the successful handling of an event, the returning phase can

be performed by executing the ERET instruction (see section 9 for more information). The returning phase will move the exception buffers or supervisor event buffers back into their corresponding locations according to the promotion type of the event: an ERET from a non promoting event will move the exception buffers back, while an ERET from a promoting event will move the supervisor event buffers back. It is worth noting that for exceptions generated by arithmetic instructions, such as overflows or division by zero, the hart must replay the offending instruction after the handling of the event with now, hopefully, correct parameters.

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 5 will be considered first, all the way to global priority level 0. Local priority simply specifies the order in which events of the same type must be handled and only concerns asynchronous events. Local priority can be dynamically changed by the supervisor with the help of the IO controller, which should be responsible for scheduling the actual interrupt to be sent to the target hart. It is important to note that it is forbidden to have more than one IO or IPC interrupt with the same local priority. This is because there shouldn't be any ambiguity on the scheduling order. Synchronous events from the EXC and TM modules do not have local priority because those exceptions are generated from the instructions themselves and should, therefore, be handled in program order. The DBGR module, however, allows the triggering of multiple events needing an internal priority between the said events.

Performance counters and debugging registers, if present, must be freezed if the triggered event is promoting which is accomplished via the DBGF and the CNTF bits. If the supervisor needs said resources, it must save them first before utilizing them. The supervisor must restore the proper values back before actually scheduling a user process for execution. This ensures the retention of values across context switches and event handling.

6.5 Double fault

During the handling of an exception, if another fault event is triggered before having saved the supervisor buffer registers to memory the state might become corrupt because those registers will be overwritten, thus, corrupting the previous values. This situation can happen if the second event is either: ILLI, MALI, MISI, PGFT or INCI fault. Those faults might afflict the supervisor code responsible of scheduling the handlers if it contains an: illegal, malformed, misaligned, incompatible instruction or a page fault. If said problematic instruction is executed before saving the supervisor event buffers to memory, corruption of state is unavoidable. If such situation is detected, the hart must be immediately halted and the DBF bit in the CSR must be set to one, signifying the presence of a double fault. Avoiding a double fault caused by the PGFT event can be done by always having that portion of code in memory.

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. Events are not perfect mechanisms and can result in the awful double fault situation. Luckily this situation can only arise if the supervisor code responsible for launching the handlers themselves contains erroneous instructions. In order to get rid of the double fault, that particular supervisor code must never cause the mentioned faults and must always be loaded into memory to avoid page faults.

ID	Name	Mnemonic	Type	Requirement	GP	LP
00	Carry over exception	COVRE	Exception	EXC module	2	-
01	Carry under exception	CUNDE	Exception	EXC module	2	-
02	Overflow exception	OVFLE	Exception	EXC module	2	-
03	Underflow exception	UNFLE	Exception	EXC module	2	-
04	Division by 0 exception	DIV0E	Exception	EXC module	2	-
05	Conflict abort	CABT	Exception	TM module	1	-
06	Event abort	EABT	Exception	TM module	1	-
07	Depth overflow abort	OABT	Exception	TM module	1	-
08	Replacement abort	RABT	Exception	TM module	1	-
09	Size abort	SABT	Exception	TM module	1	-
0A	Depth underflow abort	UABT	Exception	TM module	1	-
0B	Trap on execute address	EXT	Exception	DBGR module	0	14
0C	Trap on read address	RDT	Exception	DBGR module	0	13
0D	Trap on write address	WRT	Exception	DBGR module	0	12
0E	Trap on read write address	RWT	Exception	DBGR module	0	11
0F	Trap on read write execute address	RWET	Exception	DBGR module	0	10
10	Trap on execute range	EXRT	Exception	DBGR module	0	9
11	Trap on read range	RDRT	Exception	DBGR module	0	8
12	Trap on write range	WRRT	Exception	DBGR module	0	7
13	Trap on read write range	RWRT	Exception	DBGR module	0	6
14	Trap on read write execute range	RWERT	Exception	DBGR module	0	5
15	Trap on COVR flag	COVRT	Exception	DBGR module	0	4
16	Trap on CUND flag	CUNDT	Exception	DBGR module	0	3
17	Trap on OVFL flag	OVFLT	Exception	DBGR module	0	2
18	Trap on UNFL flag	UNFLT	Exception	DBGR module	0	1
19	Trap on DIV0 flag	DIV0T	Exception	DBGR module	0	0
1A	Reserved	-	Exception	-	-	-
...	...	...	Exception	...	...	...
FF	Reserved	-	Exception	-	-	-

Table 2: User event table. GP and LP are abbreviations for Global Priority and Local Priority respectively.

ID	Name	Mnemonic	Type	Requirement	GP	LP
00	Switch interrupt	SWITCH	IO interrupt	SUPER module	5	max
01	Interrupt 0	INT0	IO interrupt	SEVNT module	5	cstm
...	...	...	...	...	...	...
40	Interrupt 63	INT63	IO interrupt	SEVNT module	5	cstm
41	Interrupt 64	INT64	IPC interrupt	SEVNT module	5	cstm
...	...	...	...	...	...	...
80	Interrupt 127	INT127	IPC interrupt	SEVNT module	5	cstm
81	System call 0	SYSCL0	Fault	SEVNT module	4	-
...	...	...	...	...	...	
E4	System call 99	SYSCL99	Fault	SEVNT module	4	-
E5	Illegal instruction	ILLI	Fault	mandatory	4	-
E6	Malformed instruction	MALI	Fault	mandatory	4	-
E7	Misaligned instruction	MISI	Fault	mandatory	4	-
E8	Illegal execute address	IEXE	Fault	SEVNT module	4	-
E9	Illegal read address	IRDA	Fault	SEVNT module	4	-
EA	Illegal write address	IWRA	Fault	SEVNT module	4	-
EB	Misaligned address	MISA	Fault	mandatory	4	-
EC	Page fault	PGFT	Fault	SEVNT module	4	-
ED	Incompatible instruction	INCI	Fault	mandatory	4	-
EE	Reserved	-	-	-	-	-
...	...	...	...	...	...	...
FF	Reserved	-	-	-	-	-

Table 3: Supervisor event table. GP and LP are abbreviations for Global Priority and Local Priority respectively. The term "cstm" stands for custom because it can be decided on the fly by the supervisor, while "max" simply means the highest possible value.

7 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.

7.1 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 6 and 9 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 hart active to allow the loading of the OS itself and the creation of all the required data structures. This is indicated by the BOOT bit in the control and status register. After the booting phase is done, this bit can simply be set to zero causing the processor to execute normally. Only one hart is required to have this bit while the others can ignore this requirement and simply start in a halted state waiting to be unlocked by an IPC interrupt.

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.

7.2 Virtual memory

FabRISC provides privileged customizable and implementation specific instructions (MMUOP) to interface directly with the MMU, if the processor has one, allowing the OS to modify and manage that particular unit (see section 9 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 and thread id in the PID and TID registers respectively to avoid flushing during context switches. 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 and thread ids can help reducing the burst of TLB misses caused by context switches especially with frequent system calls in microkernel based solutions.

7.3 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.

---

8 ISA specification

In this section the register file organization, vector model, processor modes and the instruction modules are presented. FabRISC is a modular ISA composed of four macro instruction modules each divided into several micro modules that the hardware designer can choose. The macro modules, as discussed in section 2, concern computational, data transfer, flow transfer and system instructions. Processors that don't support certain modules must generate the INCI fault whenever an unimplemented instruction is fetched. Processors must also generate the ILLI fault whenever a combination of all zeros or all ones is fetched in order to increase safety against buffer overflows exploits or accidental access of uninitialized memory locations.

8.1 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 any vector execution. All of these registers are non privileged.*

4. ***Helper registers*** *(HLPRs): this bank is composed of 32 registers which can be used for debugging, performance counters, memory safety and more. These registers can WLEN or 32 bits wide and are not privileged.*

Scalar	Special	Vector	Helper
P0	EB0	V0	DBG0
P1	EB1	V1	DBG1
P2	EB2	V2	DBG2
P3	UEHP	V3	DBG3
P4	VMSK	V4	DBG4
P5	PC	V5	DBG5
S0	CAUSE	V6	DBG6
S1	CSR	V7	DBG7
S2	SEB0	V8	DBG8
S3	SEB1	V9	DBG9
S4	SEB2	V10	DBG10
S5	SEHP	V11	DBG11
S6	PTP	V12	DBG12
S7	PID	V13	DBG13
S8	TID	V14	DBG14
S9	FU0	V15	DBG15
S10	FU1	V16	DBGM0
S11	FU2	V17	DBGM1
S12	FU3	V18	PCNT0
S13	K0	V19	PCNT1
S14	K1	V20	PCNT2
S15	K2	V21	PCNT3
S16	K3	V22	PCNT4
S17	WDT	V23	PCNT5
N0	N5	V24	PCNT06
N1	N6	V25	PCNT07
N2	N7	V26	PCNT8
N3	N8	V27	PCNT9
N4	N9	V28	PCNT10
GP	N10	V29	PCNT11
SP	N11	V30	PCNTM0
RA	N12	V31	PCNTM1

Table 4: Register file.

8.1.1 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 9 for more information).

8.1.2 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:

- ***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), EB1 will store the status register (CSR) and EB2 will store the cause register (CAUSE). 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.*
- ***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. Event identifiers are small 8 bit values that uniquely identify each individual event: 256 possible for the user and another 256 possible for the supervisor. The remaining 23 bits can be used to hold extra information about the triggered event. CAUSE is 32 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.*
- ***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.*
- ***Thread identifier**: this register, called "TID", holds the thread identifier for the currently executing thread. The MMU can use TID to avoid flushing the TLB during context switches because TID serves as a discriminant. If the CPU transitions from user to supervisor mode, TID must hold the identifier of the privileged thread via the use of hardwired values. TID 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.*
- ***Watchdog timer**: this register, called "WDT", is a counter that periodically counts down from a set value towards zero. Once at zero, the counter will generate the SWITCH IO interrupt and reset itself to the previously set value. The generated interrupt must be directed to the hart itself in order to invoke the supervisor to perform a context switch. This timer is controlled via the WDAB bit in the CSR. WDT is 32 bits wide and is privileged.*
- ***Extra volatile registers**: these registers are simply extra volatile registers for the user.*

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 20 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.*
  - ***Debugging registers freeze bit** (DBGF): this bit indicates if the debugging registers are active or not for this particular hart. This bit serves to suppress exception events triggered by debugging registers during the handling of events.*
  - ***Counter registers freeze bit** (CNTF): this bit indicates if the performance counter registers are active or not for this particular hart. This bit serves to suppress any counter during the handling of events in order to preserve its value for the preempted process.*

- ***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 in order to not skew the load.*
- ***Double fault bit*** *(DBF): this bit indicates if the hart generated a double fault. This bit can be fed to the IO controller in order to be red by the master core.*
- ***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.*
- ***Watchdog active bit*** *(WDAB): this bit indicates if the watchdog timer WDT is active or not. In order to operate the WDT register, the supervisor must write said register with a value greater than zero. Once the value has been written, the timer can be enabled by asserting this bit to one.*
- ***reserved****: the remaining bit is unused.*

8.1.3 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 CSR 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.*

- ***Trapped mode**: this CSR bit dictates if a particular hart is handling an event or not. The only usage for this bit is to notify other harts within the system who is doing what.*
- ***Halted mode**: this CSR bit dictates if the hart is halted or not. The halted state simply disables the hart in question. The hart can be still be "unlocked" by waking IPC interrupts sent by other harts.*

8.1.4 Helper register bank

The fourth bank is dedicated to helper registers, such as debugging and performance counting registers. This bank is optional and is only needed by the DBGR and PERFC modules. The bank is divided into two sections, the first holds debugging registers while the second holds the performance counters.

- ***Debugging registers**: registers prefixed with "DBG" are used to hold a value that is constantly checked every cycle against a particular condition. If the condition is met, the register will trigger an exception causing a subsequent jump to the handler for that exception. DBG registers are all WLEN bits wide and are not privileged.*
- ***Debugging mode registers**: registers prefixed with "DBGM" are used to hold the configuration bits for each DBG register in this bank. Each DBGM register will hold a 4 bit code for each DBG register that specifies its operating mode. DBGM0 will hold the codes for DBG0 through DBG7, while DBGM1 will hold the codes for DBG8 through DBG15. DBGM registers are all 32 bits wide and are not privileged.*
- ***Performance counters**: registers prefixed with "PCNT" are used as general purpose performance counters or, alternatively, as real-time counters. It is important to note that PCNT registers are only active during the lifetime of the hart that uses them and are freezed during context switches. PCNT registers are all 64 bits wide and are not privileged.*
- ***Performance counters mode**: registers prefixed with "PCNTM" are used to hold configuration bits for each PCNT register in this bank. Each PCNTM register will hold a 5 bit code for each PCNT register that specifies its operating mode. PCNTM0 will hold the codes for PCNT0 through PCNT5, while PCNTM1 will hold the codes for PCNT6 through PCNT11. PCNTM registers are all 32 bits wide and are not privileged.*

8.1.5 Debugging and Performance measurement modules

The DBGR and PERFC modules can be included in the design in order to provide to the programmer the ability to perform debugging via breakpoints, zero-cost memory boundary checks as well as various performance and time monitoring operations. The two mentioned modules will require the implementation of some specific instructions (see section 9 for more information) as well as the previously mentioned helper registers. FabRISC defines some standard configuration modes for the DBG registers:

- ***Disabled**: this mode, code 0000, doesn't do anything and simply ignores any value written in the corresponding DBG register.*
- ***Trap on execute address**: this mode, code 0001, will cause the corresponding DBG register to generate the EXT exception as soon as the hart tries to fetch the instruction at the specified address.*
- ***Trap on read address**: this mode, code 0010, will cause the corresponding DBG register to generate the RDT exception as soon as the hart tries to read data at the specified address.*
- ***Trap on write address**: this mode, code 0011, will cause the corresponding DBG register to generate the WRT exception as soon as the hart tries to write data at the specified address.*
- ***Trap on read or address**: this mode, code 0100, will cause the corresponding DBG register to generate the RWT exception as soon as the hart tries to read or write data at the specified address.*
- ***Trap on read or write or execute address**: this mode, code 0101, will cause the corresponding DBG register to generate the RWET exception as soon as the hart tries to read or write data or fetch the instruction at the specified address.*

- ***Trap on execute range****: this mode, code 0110, will cause the corresponding DBG register to generate the EXRT exception as soon as the hart tries to fetch instructions outside the specified range. If this mode is selected, the value of the specified DBG register will be considered the starting address of the range. The terminating address of the range will be held in the DBG register after the specified one and its mode will be ignored.*
- ***Trap on read range****: this mode, code 0111, will cause the corresponding DBG register to generate the RDRT exception as soon as the hart tries to read data outside the specified range. If this mode is selected, the value of the specified DBG register will be considered the starting address of the range. The terminating address of the range will be held in the DBG register after the specified one and its mode will be ignored.*
- ***Trap on write range****: this mode, code 1000, will cause the corresponding DBG register to generate the WRRT exception as soon as the hart tries to write data outside the specified range. If this mode is selected, the value of the specified DBG register will be considered the starting address of the range. The terminating address of the range will be held in the DBG register after the specified one and its mode will be ignored.*
- ***Trap on read or write range****: this mode, code 1001, will cause the corresponding DBG register to generate the RWRT exception as soon as the hart tries to read or write data outside the specified range. If this mode is selected, the value of the specified DBG register will be considered the starting address of the range. The terminating address of the range will be held in the DBG register after the specified one and its mode will be ignored.*
- ***Trap on read or write or execute range****: this mode, code 1010, will cause the corresponding DBG register to generate the RWERT exception as soon as the hart tries to read or write data or fetch instructions outside the specified range. If this mode is selected, the value of the specified DBG register will be considered the starting address of the range. The terminating address of the range will be held in the DBG register after the specified one and its mode will be ignored.*
- ***Trap on COVR flag****: this mode, code 1011, will cause the corresponding DBG register to generate the COVRT exception as soon as the COVR flag is raised at the instruction address held in the specified DBG register.*
- ***Trap on CUND flag****: this mode, code 1100, will cause the corresponding DBG register to generate the CUNDT exception as soon as the CUND flag is raised at the instruction address held in the specified DBG register.*
- ***Trap on OVFL flag****: this mode, code 1101, will cause the corresponding DBG register to generate the OVFLT exception as soon as the OVFL flag is raised at the instruction address held in the specified DBG register.*
- ***Trap on UNFL flag****: this mode, code 1110, will cause the corresponding DBG register to generate the UNFLT exception as soon as the UNFL flag is raised at the instruction address held in the specified DBG register.*
- ***Trap on DIV0 flag****: this mode, code 1111, will cause the corresponding DBG register to generate the DIV0T exception as soon as the DIV0 flag is raised at the instruction address held in the specified DBG register.*

Performance counter modes are left as implementation specific since the microarchitecture can vary quite dramatically across different designs. The hardware designer has at his disposal up to 32 different possible modes.

It is worth noting that when a DBG register triggers an event, the upper 23 bits of the CAUSE registers can be used to write which specific DBG caused the exception, allowing the programmer to distinguish between DBG registers that check for the same condition.

If multiple DBG register specify multiple address ranges, those ranges must be AND-ed together in order to allow proper automatic boundary checking inside the process address space. This can be useful for high-level languages that provide memory-safe operations without hindering performance at all.

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 event handling 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. The last bank includes a collection of "helper" registers that can be used for debugging, performance monitoring and memory safety.

8.2 Instruction formats

FabRISC specifies 14 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	G
							opc	opc	ra	ra	opc	H
							imm	imm	imm	ra	opc	I
							imm	imm	imm	imm	opc	J
							opc	opc	opc	opc	opc	K
	rd	mod	mod	mod	rc	rb	rb	rb	ra	ra	opc	A
	opc	mod	mod	mod	rc	rb	rb	rb	ra	ra	opc	B
	opc	opc	mod	opc	mod	rb	rb	rb	ra	ra	opc	C
	imm	imm	imm	mod	rc	rb	rb	rb	ra	ra	opc	D
	imm	mod	mod	mod	imm	rb	rb	rb	ra	ra	opc	E
	imm	imm	mod	mod	imm	imm	imm	imm	ra	ra	opc	F
imm	imm	imm	imm	mod	rc	rb	rb	rb	ra	ra	opc	D.l
imm	imm	mod	mod	mod	imm	rb	rb	rb	ra	ra	opc	E.l
imm	imm	imm	mod	mod	imm	imm	imm	imm	ra	ra	opc	F.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 two instruction modifier classes:*

    1. ***tt m uu**: this class is used by scalar memory instructions (R,R,R,R). '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. ***-a m uu**: this class is used by vector memory instructions (R,R,R,R). '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.*

- ***B**: this format has three instruction modifier classes:*

    1. ***tt m vv**: this class is used by computational instructions (R,R,R). 'tt' encodes the data type length: 1, 2, 4 or 8 bytes, 'm' states if the instruction is masked or not and 'vv' dictates the instruction mode: scalar, vector-vector or vector-scalar.*
    2. ***tt m –**: this class is used by vector gather and scatter memory instructions (R,R,R). '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 cc**: this class is used by the CAS atomic memory instructions (R,R,R). 'tt' encodes the data type length: 1, 2, 4 or 8 bytes, 'm' states if the instruction is masked or not and 'cc' dictates the condition: EQ, NE, LT, or LE.*

- ***C**: this format has five instruction modifier classes:*

    1. ***tt m – v**: this class is used by computational instructions (R,R). 'tt' encodes the data type length: 1, 2, 4 or 8 bytes, 'm' states if the instruction is masked or not and 'v' dictates the instruction mode: scalar or vector*
    2. ***tt m – d**: this class is used by reduction instructions (R,R). 'tt' encodes the data type length: 1, 2, 4 or 8 bytes, 'm' states if the instruction is masked or not and 'd' dictates the direction for non commutative operations.*
    3. ***tt m - tt**: this class is used by conversion instructions (R,R). 'tt' encodes the data type length: 1, 2, 4 or 8 bytes and 'm' states if the instruction is masked or not.*
    4. ***tt m nnn**: this class is used by move and swap instructions (R,R). '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 to act on from one to eight inclusive.*
    5. ***tt m - cc**: this class is used by compare based mask setting instructions (R,R). 'tt' encodes the data type length: 1, 2, 4 or 8 bytes, 'm' states if the instruction is masked or not and 'cc' dictates the condition: EQ, NE, LT, or LE.*

- ***D**: this format has two instruction modifier classes:*

    1. ***tt m**: this class is used by scalar load and store memory instructions (R,R,R,I). 'tt' encodes the data type length: 1, 2, 4 or 8 bytes and 'm' states if the instruction is masked or not.*
    2. ***- a m**: this class is used by vector load and store memory instructions (R,R,R,I). 'a' specifies the addressing mode: standard or striding and 'm' states if the instruction is masked or not.*

- ***D.l**: this format has two instruction modifier classes:*

    1. ***tt m uu**: this class is used by scalar load and store memory instructions (R,R,R,I). 'tt' encodes the data type length: 1, 2, 4 or 8 bytes, 'm' states if the instruction is masked or not and 'uu' specifies the auto-update mode: nothing, post-increment, post-decrement or pre-decrement.*
    2. ***- a m uu**: this class is used by vector load and store memory instructions (R,R,R,I). 'a' specifies the addressing mode: standard or striding, 'm' states if the instruction is masked or not and 'uu' specifies the auto-update mode: nothing, post-increment, post-decrement or pre-decrement.*

- ***E**: this format has five instruction modifier classes:*

1. ***tt m v o**: this class is used by computational instructions (R,R,I). 'tt' encodes the data type length: 1, 2, 4 or 8 bytes, 'm' states if the instruction is masked or not, 'v' dictates the instruction mode: scalar or vector and 'o' is simply and extra opcode bit.*
2. ***tt m ii**: this class is used scalar load and store memory instructions (R,R,I). 'tt' encodes the data type length: 1, 2, 4 or 8 bytes, 'm' states if the instruction is masked or not and 'ii' are simply extra immediate bits.*
3. ***- a m ii**: this class is used by vector load and store instructions (R,R,I). this class is used by vector load and store memory instructions. 'a' specifies the addressing mode: standard or striding, 'm' states if the instruction is masked or not and 'ii' are simply extra immediate bits.*
4. ***tt p ii**: this class is used by compare based conditional branch instructions (R,R,I). 'tt' encodes the data type length: 1, 2, 4 or 8 bytes, 'p' dictates the instruction mode: integer or floating point and 'ii' are simply extra immediate bits.*
5. ***iiiii**: this class is used by indirect function calls and jump instructions (R,R,I). 'iiiii' are simply extra immediate bits.*

- ***E.l**: this format has five instruction modifier classes:*

1. ***tt m v o**: this class is used by computational instructions (R,R,I). 'tt' encodes the data type length: 1, 2, 4 or 8 bytes, 'm' states if the instruction is masked or not, 'v' dictates the instruction mode: scalar or vector and 'o' is simply and extra opcode bit.*
2. ***tt m uu**: this class is used scalar load and store memory instructions (R,R,I). 'tt' encodes the data type length: 1, 2, 4 or 8 bytes, 'm' states if the instruction is masked or not and 'uu' specifies the auto-update mode: nothing, post-increment, post-decrement or pre-decrement.*
3. ***- a m uu**: this class is used by vector load and store instructions (R,R,I). 'a' specifies the addressing mode: standard or striding, 'm' states if the instruction is masked or not and 'uu' specifies the auto-update mode: nothing, post-increment, post-decrement or pre-decrement.*
4. ***tt p cc**: this class is used by compare based conditional branch instructions (R,R,I). 'tt' encodes the data type length: 1, 2, 4 or 8 bytes, 'p' dictates the instruction mode: integer or floating point and 'cc' dictates the condition: EQ, NE, LT, or LE.*
5. ***iiiii**: this class is used by indirect function calls and jump instructions (R,R,I). 'iiiii' are simply extra immediate bits.*

- ***F and F.l**: this format has four instruction modifier classes:*

1. ***- m ff**: this class is used by block load and store memory instructions (R,I). 'ff' specifies the target register file: SGPRs, VGPRs, SPRs or HLPRs and 'm' states if the instruction is masked or not.*
2. ***tt cc**: this class is used by test based branches (R,I). 'tt' encodes the data type length: 1, 2, 4 or 8 bytes and 'cc' dictates the condition: EQ, NE, LT, or LE.*
3. ***iiii**: this class is used by direct function calls and jump instructions (R,I). 'iiii' are simply extra immediate bits.*
4. ***tt m c**: this class is used by bit manipulation instructions (R,I). ...*

- ***G**: this format has one instruction modifier class:*

1. ***tt p**: this class is used by compressed computational instructions (R,R). 'tt' encodes the data type length: 1, 2, 4 or 8 bytes and 'p' dictates the instruction mode: integer or floating point.*

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.

8.3 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.

8.3.1 Opcode extraction

Instruction opcodes can be of variable length of 7, 11, 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 → 1111011xxxxxxxxx with a total of 124 combinations.*
- ***11 bits*** *11111000000xxxxx → 11111001111xxxxx with a total of 16 combinations.*
- ***12 bits*** *111110100000xxxx → 111111011111xxxx with a total of 64 combinations.*
- ***16 bits*** *1111111000000000 → 1111111001111111 with a total of 128 combinations.*
- *there are 384 unused opcode combinations going from: 11111101xxxxxx → 11111111xxxxxx. These combinations can be used by custom application specific instructions.*

The total size of the encoding space amounts to 332 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...]

8.3.2 Micro-operation format

Micro-operation formats should be few with regular and easy to decode fields. Having one format with fixed length will simplify the logic that handles the micro-op at the cost of more space required to store it. Variable length formats can be an good choice to increase density, especially in the micro-op cache at the cost of more complex logic. Finding a good balance is can get tricky, but the following fixed length format can be a good starting point:

[to be continued...]

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. This sequencing however is still quite simple because it often just amounts to moving or swapping multiple registers. A single micro-instruction can be normally emitted and the later stages can be left doing the heavy lifting of actual sequencing the operation. This second slightly different approach can reduce the complexity of the main control unit at the expense of having a simpler and smaller "micro control unit" later in the pipeline. At the end of the day, it's all about compromises.

---

9 Instruction list

This section is dedicated to a full and extensive list of all the proposed instructions in the ISA which are divided into their corresponding module. Computational instructions can generate flags but, since there is no flags register, they will be ignored unless the AEXC bit is set for automatic arithmetic exceptions.

9.1 Computational-integer-scalar-basic

- ***(ADD) Addition**: this instruction computes ($ra = rb + rc$); belongs to the B format; uses the modifier class 1; has an opcode of xxxxxxxxxxx and can generate the COVR, OVFL and UNFL flags. This instruction is scalar.*
- ***(SUB) Subtraction**: this instruction computes ($ra = rb - rc$); belongs to the B format; uses the modifier class 1; has an opcode of xxxxxxxxxxx and can generate the COVR, OVFL and UNFL flags. This instruction is scalar.*
- ***(AND) Bitwise AND**: this instruction computes ($ra = rb \wedge rc$); belongs to the B format; uses the modifier class 1; has an opcode of xxxxxxxxxxx and doesn't generate any flags. This instruction is scalar.*
- ***(NAND) Bitwise NAND**: this instruction computes ($ra = \neg(rb \wedge rc)$); belongs to the B format; uses the modifier class 1; has an opcode of xxxxxxxxxxx and doesn't generate any flags. This instruction is scalar.*
- ***(OR) Bitwise OR**: this instruction computes ($ra = rb \vee rc$); belongs to the B format; uses the modifier class 1; has an opcode of xxxxxxxxxxx and doesn't generate any flags. This instruction is scalar.*
- ***(NOR) Bitwise NOR**: this instruction computes ($ra = \neg(rb \vee rc)$); belongs to the B format; uses the modifier class 1; has an opcode of xxxxxxxxxxx and doesn't generate any flags. This instruction is scalar.*
- ***(XOR) Bitwise XOR**: this instruction computes ($ra = rb \oplus rc$); belongs to the B format; uses the modifier class 1; has an opcode of xxxxxxxxxxx and doesn't generate any flags. This instruction is scalar.*
- ***(XNOR) Bitwise XNOR**: this instruction computes ($ra = \neg(rb \oplus rc)$); belongs to the B format; uses the modifier class 1; has an opcode of xxxxxxxxxxx and doesn't generate any flags. This instruction is scalar.*
- ***(ALSH) Arithmetic left shift**: this instruction computes ($ra = rb <<< rc$); belongs to the B format; uses the modifier class 1; has an opcode of xxxxxxxxxxx and can generate the xxx flags. This instruction is scalar.*
- ***(ARSH) Arithmetic right shift**: this instruction computes ($ra = rb >>> rc$); belongs to the E format; uses the modifier class 1; has an opcode of xxxxxxxxxxx and can generate the xxx flags. This instruction is scalar.*

- ***(LLSH) Logical left shift**: this instruction computes ($ra = rb << rc$); belongs to the E format; uses the modifier class 1; has an opcode of xxxxxxxxxxx and can generate the xxx flags. This instruction is scalar.*

- ***(LRSH) Logical right shift**: this instruction computes ($ra = rb >> rc$); belongs to the E format; uses the modifier class 1; has an opcode of xxxxxxxxxxx and can generate the xxx flags. This instruction is scalar.*

- ***(IADD) Immediate addition**: this instruction computes ($ra = rb + signxt(imm_8)$); belongs to the E format; uses the modifier class 1; has an opcode of xxxxxxxxxxx and can generate the xxx flags. This instruction is scalar.*

- ***(ISUB) Immediate subtraction**: this instruction computes ($ra = rb - signxt(imm_8)$); belongs to the E format; uses the modifier class 1; has an opcode of xxxxxxxxxxx and can generate the xxx flags. This instruction is scalar.*

- ***(IAND) Immediate bitwise AND**: this instruction computes ($ra = rb \wedge zeroxt(imm_8)$); belongs to the E format; uses the modifier class 1; has an opcode of xxxxxxxxxxx and can generate the xxx flags. This instruction is scalar.*

- ***(INAND) Immediate bitwise NAND**: this instruction computes ($ra = \neg(rb \wedge zeroxt(imm_8))$); belongs to the E format; uses the modifier class 1; has an opcode of xxxxxxxxxxx and can generate the xxx flags. This instruction is scalar.*

- ***(IOR) Immediate bitwise OR**: this instruction computes ($ra = rb \vee zeroxt(imm_8)$); belongs to the E format; uses the modifier class 1; has an opcode of xxxxxxxxxxx and can generate the xxx flags. This instruction is scalar.*

- ***(INOR) Immediate bitwise NOR**: this instruction computes ($ra = \neg(rb \vee zeroxt(imm_8))$); belongs to the E format; uses the modifier class 1; has an opcode of xxxxxxxxxxx and can generate the xxx flags. This instruction is scalar.*

- ***(IXOR) Immediate bitwise XOR**: this instruction computes ($ra = rb \oplus zeroxt(imm_8)$); belongs to the E format; uses the modifier class 1; has an opcode of xxxxxxxxxxx and can generate the xxx flags. This instruction is scalar.*

- ***(IXNOR) Immediate bitwise XNOR**: this instruction computes ($ra = \neg(rb \oplus zeroxt(imm_8))$); belongs to the E format; uses the modifier class 1; has an opcode of xxxxxxxxxxx and can generate the xxx flags. This instruction is scalar.*

- ***(IASH) Immediate arithmetic shift**: this instruction computes ($ra = rb <<<>>> imm_8$); belongs to the E format; uses the modifier class 1; has an opcode of xxxxxxxxxxx and can generate the xxx flags. This instruction is scalar and the direction of the shift is dictated by imm[7] bit.*

- ***(ILSH) Immediate logical shift**: this instruction computes ($ra = rb <<>> imm_8$); belongs to the E format; uses the modifier class 1; has an opcode of xxxxxxxxxxx and can generate the xxx flags. This instruction is scalar and the direction of the shift is dictated by imm[7] bit.*

- ***(LIADD) Long immediate addition**: this instruction computes ($ra = rb + signxt(imm_{26})$); belongs to the E.l format; uses the modifier class 1; has an opcode of xxxxxxxxxxx and can generate the xxx flags. This instruction is scalar.*

- ***(LISUB) Long immediate subtraction**: this instruction computes ($ra = rb - signxt(imm_{26})$); belongs to the E.l format; uses the modifier class 1; has an opcode of xxxxxxxxxxx and can generate the xxx flags. This instruction is scalar.*

- ***(LIAND) Long Immediate bitwise AND**: this instruction computes ($ra = rb \wedge zeroxt(imm_{26})$); belongs to the E format; uses the modifier class 1; has an opcode of xxxxxxxxxxx and can generate the xxx flags. This instruction is scalar.*

- ***(LINAND) Long Immediate bitwise NAND**: this instruction computes ($ra = \neg(rb \wedge zeroxt(imm_{26}))$); belongs to the E.l format; uses the modifier class 1; has an opcode of xxxxxxxxxxx and can generate the xxx flags. This instruction is scalar.*

- ***(LIOR) Long Immediate bitwise OR**: this instruction computes ($ra = rb \vee zeroxt(imm_{26})$); belongs to the E.l format; uses the modifier class 1; has an opcode of xxxxxxxxxxx and can generate the xxx flags. This instruction is scalar.*

- ***(LINOR) Long Immediate bitwise NOR**: this instruction computes ($ra = \neg(rb \vee zeroxt(imm_{26}))$); belongs to the E.l format; uses the modifier class 1; has an opcode of xxxxxxxxxxx and can generate the xxx flags. This instruction is scalar.*

- ***(LIXOR) Long Immediate bitwise XOR**: this instruction computes ($ra = rb \oplus zeroxt(imm_{26})$); belongs to the E.l format; uses the modifier class 1; has an opcode of xxxxxxxxxxx and can generate the xxx flags. This instruction is scalar.*
- ***(LIXNOR) Long Immediate bitwise XNOR**: this instruction computes ($ra = \neg(rb \oplus zeroxt(imm_{26}))$); belongs to the E.l format; uses the modifier class 1; has an opcode of xxxxxxxxxxx and can generate the xxx flags. This instruction is scalar.*
- ***(IIC) Integer-integer cast**: this isntruction computes ($ra = signxt_{mod}(rb)$); belongs to the C format; uses the modifier class 3; has an opcode of xxxxxxxxxxx and generates the xxx flags.*

9.2 Computational-integer-scalar-advanced

- ***(MUL)** : ...*
- ***(MAC)** : ...*
- ***(DIV)** : ...*
- ***(REM)** : ...*
- ***(IMP)** : ...*
- ***(NIMP)** : ...*
- ***(LRT)** : ...*
- ***(RRT)** : ...*
- ***(SWP)** : ...*
- ***(IMUL)** : ...*
- ***(IMAC)** : ...*
- ***(IDIV)** : ...*
- ***(IREM)** : ...*
- ***(IIMP)** : ...*
- ***(INIMP)** : ...*
- ***(IRT)** : ...*
- ***(ISWP)** : ...*
- ***(LIMUL)** : ...*
- ***(LIMAC)** : ...*
- ***(LIDIV)** : ...*
- ***(LIREM)** : ...*
- ***(LIIMP)** : ...*
- ***(LINIMP)** : ...*
- ***(CTO)** : ...*
- ***(CTLO)** : ...*
- ***(CTTO)** : ...*
- ***(CTZ)** : ...*
- ***(CTLZ)** : ...*
- ***(CTTZ)** : ...*
- ***(MWADD)** : ...*
- ***(MWSUB)** : ...*
- ***(MWMUL)** : ...*
- ***(MWDIV)** : ...*
- ***(MALSH)** : ...*
- ***(MARSH)** : ...*
- ***(MLLSH)** : ...*
- ***(MLRSH)** : ...*

[To be continued...]

9.3 Data transfer instructions

[Coming soon...]

9.4 Control transfer instructions

[Coming soon...]

9.5 System instructions

[Coming soon...]

10 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:

[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.

---

11 License and changelog

This is the FabRISC instruction set architecture version 0.2:

- 0.1 → 0.2:
    1. merged old B format into old D.
    2. renamed the formats.
    3. remade the format table.
    4. remade the mod classes.
    5. updated the opcode ranges.

Official GitHub page of the FabRISC project: `https://github.com/Clamentos/FabRISC`