# FaVOR ISA

Benjamin Wall
*Department of Computer Science*
*George Mason University*
Fairfax, Virginia
bwall4@gmu.edu

***Abstract*—I present a new instruction set architecture, the FaVOR ISA, and an associated GNU binutils toolchain [1] (including an assembler, linker, and simulator). FaVOR is a load-store architecture, and is designed to support three features in particular: fixed-point arithmetic, vector operations (including on individual vector components), and address masking (intended to efficiently support pointer tagging schemes).**

**The ISA includes integer and floating point instructions, load and store instructions, and branching operations. The design currently does not include privilege levels, MMU operations, or interrupts (besides the existence of a syscall instruction that nominally functions as some sort of software interrupt). These details are important but not considered relevant to the main initial goals of the ISA, and so are left as future work.**

***Index Terms*—Computer architecture, ISA, Fixed point, Vector, Pointer tagging, RISC**

## I. Introduction

C remains one of the programming languages closest to the metal. Modern C programming uses a variety of newer techniques thanks to the huge 64-bit virtual address spaces of modern CPUs, as well as its relationship to other languages and domains such as GLSL and graphics programming, embedded systems, game development, and many more.

Much as C remains close to the hardware, the hardware remains close to C. But modern hardware doesn't support all the niceties of modern C, due to backward compatibility requirements with respect to ABI and instruction encoding itself. The instruction set architecture (ISA) in particular represents the semantics available to any high level language that wishes to run on a particular set of processors. These semantics often match C decently well, but the mismatches are difficult or unproductive to fix.

I present a new ISA that intends to semantically support modern C programming more closely, especially three particular features of my own C programming style: 1) vector operations, 2) pointer tagging, and 3) fixed point numerical operations. Moreover, the ISA has been designed in general to remain close to C, including good support for various-sized data types and with an eye towards useful ABI features.

Designing an ISA from the ground up is a huge project. To improve the ISA design ideally requires iterating on it with the help of a C compiler. Compiling suites of benchmarks and tracking improvements and regressions with different versions of an ISA is the best way to ensure that it actually helps the compiler generate efficient code. Unfortunately, such an in-depth project is beyond the scope of this paper.

So, I focus on low-hanging fruit. Ideas that can be added into the instruction coding at little or no cost to its general expressiveness, and generally building the ISA to match my desired semantics in obvious and efficient ways.

As such, I start from my motivating list of features–vectors, pointer tagging, fixed point–and build the ISA from there. Every decision about data types, ABI specifications, and architectural state is made to make the ISA more closely match the C semantics I care about. My development of a GNU binutils toolchain [1] for the ISA helps me test my ideas with real code, and iterate on them that way even without a full C compiler. I also keep in mind the way decisions will affect the microarchitecture, especially if minor changes to instruction encoding lead to an obviously "useful" bit pattern that could be exploited in hardware implementation.

In this paper, I present:

- A detailed description of the FaVOR ISA, including its instruction encodings, ABI design, and architectural state.
- The GNU binutils toolchain for the FaVOR ISA, which supports a partial set of instructions.

## II. Background & Motivation

C programming remains one of the "close to the metal" options for modern programming, despite the extreme amount of optimizations performed by modern compilers and the differences between the C abstract machine and actual hardware implementation. Nonetheless, modern C programming, especially for consumer hardware, and even more especially in the context of games, is different from the past in several notable ways:

- We have huge virtual memory spaces with 64 bit pointers, and most consumer applications will only use a small number of those pointers to actually represent program memory. This leads to the idea of "pointer tagging"–using unused bits in a pointer to represent other data. However, these bits must always be masked off of the pointer value when it is actually used as a pointer.
- Vector operations are widespread, but are not well supported in the C language. SIMD operations are decently well supported (both in language extensions and in hardware) but are not relevant for many kinds of vector math that operate on one component at a time (e.g. the acceleration of a player character in a 2D platformer).

- Fixed point math is not well supported in C, while floating point math is. This corresponds with most consumer hardware, such as x86, which has many floating point instructions but not many fixed point ones. This is despite the continuing usefuless of fixed point arithmetic in many possible kinds of applications.

These three points are the core motivating features of the FaVOR ISA. As the project grew, several additional elements of C programming and systems programming in general began to influence the ISA.

- In C-like languages (including C and Rust), we do not have exceptions and instead have to manually propagate errors up the call stack. This necessitates checking the return value of each function that might return an error and returning from the function from there. The ABI and calling convention could be reworked to make these checks cost less (and even to support microarchitectural features like keeping track of these kinds of coding patterns in the shadow stack).
- Calling conventions in general are complicated and unnecessarily restrained. Some architectures are not particularly register-rich (such as x86) requiring struct values to be tightly packed in registers and then extracted out again or simply requiring more pushes to the stack.
- Almost no calling convention uses all of its caller-saved registers as return values, for no apparent reason.
- ISA register sizes often do not match particularly directly to C types. This is both a blessing and a curse–enabling extremely natural and fast operations on the machine's native size, but sometimes requiring additional masking or other operations to maintain C semantics in non-native sized cases.

All of these points are relevant to the design of both the C language and the hardware. The FaVOR ISA addresses them in the context of hardware. The FaVOR ISA was designed from the ground up to address the first three, the "motivating problems" for FaVOR.

- To support pointer tagging, FaVOR simply does not support unaligned loads/stores. Unaligned loads/stores are not that common or useful, and so simply treating the last few bits of each pointer as 0 is a no-brainer: it enables entirely free pointer tagging by using those last few bits for extra data.
- FaVOR enables treating groups of registers as vectors of 1, 2, 3, or 4 components, corresponding to the most common types in games. Each register, however, can still be accessed individually, allowing easy switching between vector operations and single-component operations.
- FaVOR supports fixed point operations alongside floating point operations. This should enable efficient usage of fixed point formats by software.

FaVOR also address the latter four points simply by nature of its design.

- FaVOR's predication flag registers are used as part of the ABI to make error chains efficient and possibly mappable to a shadow stack.
- FaVOR's calling convention is defined to use as many registers as possible, with the exact data types that parameters expect.
- FaVOR very simply uses all of the caller saved registers for both arguments and return values, as there's no reason not to.
- FaVOR's instruction encoding supports all 4 of the main C data type sizes: 8-bit, 16-bit, 32-bit, and 64-bit. This keeps it nicely matching the C semantics.

## III. Methodology

The design of the ISA was driven by many competing forces, but primarily two: first: the core design principles and features I wanted for the instruction set, and second: additional useful properties that became more apparent as I started implementing the instruction set as a new backend inside GNU binutils.

The starting point for the design was the bit requirements for the fixed point arithmetic operations in particular.

The earliest decisions I made for the ISA included: it should have 32 64-bit general purpose registers, and support any 64-bit fixed point format (e.g. 48.12, 1.63, etc).

The reasons for these decisions are as follows.

- Register-rich architectures are generally better. Reducing the need to re-use registers should reduce false dependencies and make compilation easier.
- When treating groups of registers as vectors of 4 components, 32 registers is only 8 different vectors, and one of them is used for special registers, leaving only 7. It seems difficult to support the desired vector features with any fewer registers.
- 64-bit arithmetic is generally the highest that most applications need, while 32-bit is too few.
- My personal experience with fixed point numbers shows that almost every format is useful. For example, I have previously written DSP code using a 4.28 format and game code using an 8.8 format. Other formats also seem useful such as 1.31 or 1.63 for DSP, and many variations such as 56.8 and 48.16 and 24.8 for video game code.

These two decisions (32 registers, any fixed point format) fundamentally drive the requirements for the ISA. 32 registers requires 5-bit register operands in the instruction format, and the fixed point format flexibility requires a 6-bit shift operand for the fixed point instructions. This means that:

- A 1-register instruction would require a minimum of 11 bits.
- A 2-register instruction would require a minimum of 16 bits.
- A 3-register instruction would require a minimum of 21 bits.

Due to these extreme bit requirements for the fixed point formats, two principles were clear about the design of the ISA:

- Not every instruction could use the same format as the fixed point instructions. There is simply too much overhead in bits.
- 2-registers would be much better for the fixed point instructions then 3-registers, as that would leave many more bits for other parts of the instruction.

Another motivating factor of the design was the address masking system. The core idea is that unaligned load/stores in this architecture should simply not be supported, with any unaligned pointer simply being truncated when used, supporting extremely efficient pointer tagging.

However, if the architecture is designed primarily around aligned loads/stores, then it seems obvious that the instruction format itself should be a fixed size. As such, an early decision was also to make the instruction format 32 bits for every instruction, similar to RISC-V and other RISC architectures.

Combining the previous two ideas gives the following constraints:

- If using 2-register fixed point instructions, there will be 16 bits left over for the rest of the instruction.
- If using 3-register fixed point instructions, there will only be 11 bits left over for the rest of the instruction, which is enough for a particularly limited instruction format–but is not enough for the other features of the ISA.

As such, the core idea from near the beginning of the design process was that instructions would be 32 bits, and that fixed point instructions would have 2 register operands, leaving 16 bits for the rest of the instruction format.

The rest of the instruction format went through many, many iterations. Besides the fixed point instructions, and ideas about address masking, there was still one motivating point that strongly affected the encoding: the ability to use registers as vectors.

The idea from the beginning was to allow treating groups of registers as 1, 2, 3, or 4 component vectors (as these are the most common vectors used in computer graphics & game programming). These four varieties of vectors can be encoded in 2 bits in all the instructions that use them. Where those two bits were located in the instruction format changed significantly over time.

Another 2-bit field that many instruction formats used was a "size" field. The idea was to support varieties of instructions such as in Listing 1.

```
add.u8  a0, a1, a2
add.u16 a0, a1, a2
add.u32 a0, a1, a2
add.u64 a0, a1, a2
```

Listing 1: 8, 16, 32, and 64-bit unsigned adds.

This enables C code using different data types to be efficiently compiled to single instructions, rather than having to perform additional masking operations to keep data the correct size. Similar to the vector field, this field moved around a lot during the design process.

Besides the core constraints imposed by the various bit requirements for the different fields, there were additional forces driving the design of the ISA.

- The ISA should be reasonably efficient on the microarchitecture side. This means that fields that do the same thing should, whenever possible, be placed at the same location.
- Similarly, whenever there is a sign-extended immediate value, the bit controlling the sign should be located at the same location in the encoding. This idea comes from the same approach in RISC-V [2].
- Anywhere else some functionality in the ISA can be made more reusable, it should be. These often did not become obvious until that part of the ISA was implemented in the toolchain, necessitating more design churn.
- The main implementation of the ISA was in the GNU binutils toolchain, including the simulator there. This meant that conveniences for the toolchain seemed sometimes appropriate; in particular, keeping relocated fields in the encoding as contiguous fields made them much easier to implement in the toolchain.

All told, these various factors meant that the ISA went through several revisions, including various ways to delineate the various instruction encodings, various ways to locate the common fields, and other variations, until a sufficient amount of design information was gathered to nail down one reasonably final encoding.

The encoding eventually got to a state, based partially on RISC-V, with the following properties:

- The 4 least significant bits form an "opcode" field, choosing between various sub-encodings of instructions.
- Most instructions have a vector field and a size field.
- All instructions have a "predicated" field.

These properties would remain till the final encoding. After enough design and iteration on the various sub-encodings and the instructions available within them, it was possible to come up with a final arrangement of fields inside the instruction that kept as many things as possible in the same place, and that kept the opcode space reasonably open for further expansion.

This process of iteration included much testing in GNU binutils and re-writing of existing code. Implementing features in GNU binutils would reveal possible commonalities that could be refactored–in the encoding itself–to be smarter and more reusable.

Perhaps the best example of this is the set of condition codes. As with most ISAs, FaVOR has a set of condition flags that are set by comparisons and arithmetic operations. These condition flags can then be combined into specific *conditions codes*, such as "unsigned greater than", which is equal to `carry && !zero`.

However, there are many instructions in FaVOR that re-use the set of condition codes, and so ideally they can re-use the same ALU logic when possible for computing them. This means that whenever some instructions might use condition codes, the order of those condition codes should be the same, and should *ideally* be aligned to start at `0000` somewhere inside the encoding (so that the specific condition code can be extracted simply as a wire, not even requiring a subtraction).

This means that the ordering of the condition codes is important, as it is pervasive throughout the ISA. But, while working on the condition codes, I realized various properties that would make them much better.

First, the jump instructions can only encode 9 different kinds of condition codes, so the decision of which condition codes should be left out is important and changes how they are arranged.

Later, however, it also became extremely obvious that the last bit of the condition code should simply flip the condition. For the condition "equals" and "not equals" this was already the case (they were one bit apart), and for the condition "negative" and "non-negative" this was *also* already the case. So the rest of the instructions–the greater than, less than, greater than-equal, and less-than-equal, were rearranged to fit this same pattern. This meant rearranging a number of instructions inside the "misc" category, the "jump" category, and the "2-int" category (all discussed later).

These kinds of useful instruction encoding properties that only become apparent once the implementation starts to be realized cropped up throughout the project, including through some prototypes in Verilog as well as through the main iteration on the GNU binutils toolchain.

## IV. Results

The FaVOR ISA is now a partially-complete ISA defining:

- Basic integer & system instructions
- The general architectural state
- Formats for future instructions

With the associated binutils toolchain able to assemble and link reasonable small programs, but missing some features including:

- Floating point numbers
- General purpose load/store instructions
- Integer instructions with immediate operands

### A. Architectural state

The architectural state of the FaVOR ISA is defined as 32 general purpose integer registers, 32 floating point registers, 4 "predication flag" registers, 16 "condition flag" registers, and a 62 bit program counter.

The integer and floating-point registers are both 64 bits. They can be used to store values of various sizes. For the integer registers, smaller values are stored from least-significant byte to most-significant byte, meaning that unsigned casts from smaller values to larger values are free. For the floating-point registers, only 2 sizes of values are supported: 32-bit "singles" and 64-bit "doubles." 32-bit "singles" are stored in the upper 32 bits of the register, so that the sign bit of both kinds of floating point numbers overlaps inside the register.

The predication flag registers are 1-bit registers that can be set as the result of comparison instructions or as a result of condition flag values generated by a previous arithmetic instruction. There are 4 predication flag registers to match the 4-component vectors that may be operated on in parallel. The predication flag registers enable specific components of each vector to be enabled/disabled in particular instructions, called "predicated" instructions, so that multiple copies of the same instruction can effectively conditionally execute in parallel.

Relatedly, there are four 1-bit condition flag registers, replicated four times to create the 16 condition flag registers. The four condition flags are:

- `zero`: represents whether the previous arithmetic result was 0
- `carry`: carry bit from previous instruction, or represents unsigned overflow of previous arithmetic operation
- `sign`: sign bit of the previous arithmetic result
- `overflow`: represents signed overflow of the previous arithmetic operation

These condition flags are relatively standard. The instruction encoding defines 12 different condition codes based on these flags, with 4 reserved slots for future conditions, outlined in Table I.

TABLE I: Condition codes

| Index | ASM | Condition | Definition |
|---|---|---|---|
| 0000 | eq | equals | zero |
| 0001 | ne | not equals | !zero |
| 0010 | le.u | unsigned $\leq$ | !carry \|\| zero |
| 0011 | g.u | unsigned $>$ | carry && !zero |
| 0100 | l.u | unsigned $<$ | !carry |
| 0101 | ge.u | unsigned $\geq$ | carry |
| 0110 | le.s | signed $\leq$ | zero \|\| (sign != overflow) |
| 0111 | g.s | signed $>$ | !zero && (sign == overflow) |
| 1000 | l.s | signed $<$ | sign != overflow |
| 1001 | ge.s | signed $\geq$ | sign == overflow |
| 1010 | neg | negative | sign |
| 1011 | nne | non-negative | !sign |
| 1100 | - | reserved | - |
| 1101 | - | reserved | - |
| 1110 | - | reserved | - |
| 1111 | - | reserved | - |

There are 12 condition codes currently defined, but the ISA reserves an additional 4 slots of condition codes so that they can be encoded nicely into instructions. In particular, using a single 16-bit value to choose the condition code should enable microarchitectural optimizations such as using a single condition code decoder that can be fed from multiple sources all providing the same formatted 4-bit value.

One other microarchitectural consideration for the condition codes is that every code comes in a pair, such as "equals" and "not equals" or "unsigned $\leq$" and "unsigned $>$." This means that the least significant bit of the condition code simply flips the condition, which should enable microarchitectures to implement the condition codes as only half of the actual codes plus an XOR on the output with the least significant bit. Of course, a truly parallel implementation of the ISA would need 4 copies of the condition code logic, one for each vector component.

The FaVOR program counter is only defined to be 62 bits because every instruction in FaVOR is aligned to 4 bytes. As such, the last two bits of the program counter are simply redundant and do not need to be stored.

Of the 32 general-purpose registers, 4 of them are special registers. These 4 registers are:

- Register 28: frame pointer
- Register 29: link address

- Register 30: stack pointer
- Register 31: zero register

The frame pointer and stack pointer are ABI related, with the stack pointer being the most important–always pointing to the bottom of the stack (the FaVOR stack is defined to grow downwards). The frame pointer can be optionally used to implement a linked list of call frames, which can be useful for debuggability. The two registers most directly important to the ISA, however, are the link address and the zero register.

The zero register, much like the similar register in MIPS and RISC-V, is defined to always have the value 0. Writes to this register do not do anything, while reads always return a value of 0. This register is particularly useful for the load/store instructions, which always take a pointer register and an offset register–the zero register may be used when the offset is not important.

The link address register stores the previous value of the program counter when a function is called. That is, it stores the address of the jump instruction that called a function, rather than the address after the jump instruction. When the link address register is updated in this way, its two least significant bits are always set to 0, as the program counter is always implicitly aligned to 4 bytes. Similarly, when the link address is restored to the program counter through a return instruction, the last 2 bits are simply ignored.

### *B. ABI & calling convention*

The registers in FaVOR are named as shown in Table II and Table III. The registers are logically grouped into groups of 4, so as for usage with the vector-based instructions in FaVOR.

There are 3 main categories of registers are far as the ABI is concerned:

- **a** registers: caller-saved, used for both arguments and return values.
- **t** registers: callee-saved, used for any callee-saved values.
- **r** registers: same as **t** registers, but are conceptually reserved for usage by the programming language implementation or kernel implementation. That is, if there is a register that kernel code or a programming language implementation wants to reserve outside of normal ABI purposes, by convention it should use the **r** registers first.

TABLE II: General purpose register names

| | | | |
|---|---|---|---|
| a0 | a1 | a2 | a3 |
| a4 | a5 | a6 | a7 |
| a8 | a9 | a10 | a11 |
| t0 | t1 | t2 | t3 |
| t4 | t5 | t6 | t7 |
| t8 | t9 | t10 | t11 |
| r0 | r1 | r2 | r3 |
| fp | la | sp | zero |

TABLE III: Floating-point register names

| | | | |
|---|---|---|---|
| fa0 | fa1 | fa2 | fa3 |
| fa4 | fa5 | fa6 | fa7 |
| fa8 | fa9 | fa10 | fa11 |
| fa12 | fa13 | fa14 | fa15 |
| ft0 | ft1 | ft2 | ft3 |
| ft4 | ft5 | ft6 | ft7 |
| ft8 | ft9 | ft10 | ft11 |
| ft12 | ft13 | ft14 | ft15 |

All of the **a** registers are used for both function parameters and return values, allocated in order without any packing of values inside the registers. For example, if there are 12 single-byte values passed to a function, all 12 would go inside the `a0`..`a11` registers without anything needing to be pushed to the stack. Similarly, if a function returns a tuple of several integers, they should go in all the **a** registers first before anything is pushed to the stack.

The idea is that when a register is caller-saved there is no real reason *not* to use it as both an argument register and a return register. Any caller is allowed to arbitrarily clobber it anyway, so it might as well put something useful (i.e. the return value) in it instead of wasting time pushing that useful information to the stack. Similarly, the caller might as well use the register for additional arguments, as it can't do anything else with it anyways.

The condition flags are, unsurprisingly, also effectively caller-saved–any callee may clobber them.

The predication flags are also caller-saved, but serve additional purposes in the calling convention. In particular, the 0th predication flag is used to communicate properties of certain kinds of return values.

- For a boolean return value, the flag bit is equal to the return value (and the return value is stored in `a0`).
- For a pointer return value, the flag bit is 1 if the pointer is NULL, and 0 otherwise. This enables NULL pointer values to be quickly unwound up the call stack using predicated instructions.
- For an `Option` or `Result` value such as in Rust, the flag bit is similarly set to 1 in the `None` or `Err` cases and 0 otherwise, also to quickly enable unwinding of these values up the call stack.
- For certain kinds of error pointers, such as in the Linux kernel with its `ERR_PTR` approach [3], the flag bit can be set to 1 in the case of an error and 0 otherwise, again for supporting efficient unwinding.

A key property of the ISA that enables this predication flag as an additional part of the calling convention is the encoding of MISC instructions. Any MISC instruction can also act as a return instruction, meaning that it is possible to set the predication flag to the relevant condition in a single instruction.

```
ret.chkptr  a0      # check for non-null
ret.chkbool a0
ret.test    a0, 63 # test a bit
```

Listing 2: Various ret instructions useful for predication setting or chains.

```
j.l          subroutine
ret?
...
ret.chkptr a0
```

Listing 3: How to call a function then return if it returned null.

The exact ABI for additional arguments and large structs has not been decided yet. The stack grows downward, with the stack pointer (`sp` register) always pointing at the bottom.

When a function call is made, with a `j.l` instruction, the previous program counter is stored in the `la` register. Similarly, any `ret` instruction copies the value in `la` back to the program counter.

The system call convention currently puts the call number into `a11`, and the system call itself is allowed to clobber `a0` (where it writes the return value), but should restore all other registers (except for condition flags and predicate flags).

### C. Instruction encoding & semantics

FaVOR instructions are always 32 bits long (and are always aligned to 4 bytes when stored in memory). A compressed instruction set was considered but discareded due to the many bit-heavy instructions that exist in the encoding–there are not many spare bits.

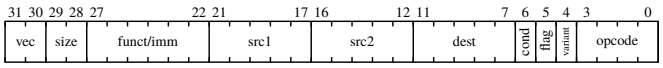


Fig. 1: Basic instruction layout

Fig. 1 shows the common layout of fields in FaVOR. This fields are not common to every instruction, but when possible, FaVOR does locate fields in identical places in different encodings to keep the microarchitecture simpler.

Every instruction in FaVOR has an `opcode` field, and every instruction except for JUMP has a `cond` field. The `opcode` field distinguishes between 16 sub-encodings of instructions. The `cond` field determines whether an instruction is predicated.

The predicated instructions have two possible semantics:

1) In the case of instructions that operate on a vector of registers or memory locations, every component of the destination is only written to if the matching component of the vector of predication flags is true. Examples of this category include arithmetic instructions and loads/stores.
2) In the case of instructions that do not meaningfully operate on a vector, the instruction is only executed if the 0th predication flag is true. Examples of this category include jumps and load-immediate instructions.

For any instruction with a `cond` flag (i.e. everything but JUMP), the suffix `?` indicates that the instruction is predicated. So, for example, `add.u32?` indicates a conditional add instruction. JUMP instructions also support the same suffix with the same meaning but encode the information differently.

Besides the `cond` flag, there are two additional flags located next to the `opcode` field. The `variant` flag is effectively an extension of the `opcode` field: many instructions have 2 related variants, so the `variant` field can switch between them. The `flag` field is used either as an additional flag that controls the behavior of the instruction, or simply as an extension of the `funct` field. The reason for the layout this way is that many instructions in FaVOR include a single 1-bit flag that controls some aspect of the instruction:

- MISC instructions include an `is_return` flag
- JUMP instructions include an `and_link` flag
- INT and FLOAT instructions include a `splat` flag
- LOAD and STORE instructions include an `fp` flag

So, due to the overwhelmingly prevalence of single-bit flags in this fashion, it was decided to locate it in the same place in as many instructions as possible.

The `vec` and `size` fields are, respectively, common to any instructions that support varied-component vectors, and to any instructions that support the various data sizes.

TABLE IV: Meaning of the vec and size fields

| vec | ASM | meaning |
|---|---|---|
| 00 | - | 1 component / scalar |
| 01 | x2 | 2 components |
| 10 | x3 | 3 components |
| 11 | x4 | 4 components |

| size | ASM (unsigned) | ASM (signed) | meaning |
|---|---|---|---|
| 00 | u8 | s8 | 8 bit |
| 01 | u16 | s16 | 16 bit |
| 10 | u32 | s32 | 32 bit |
| 11 | u64 | s64 | 64 bit |

The `dest`, `src2`, and `src1` fields represent the up-to-three register operands that instructions may have. The most common forms of instructions are shown in Listing 4.

```
op dest, src1, src2 # dest <- src1 <op> src2
op dest, src2       # dest <- dest <op> src2
```

Listing 4: Meaning of dest, src1, src2

Except for `store` instructions, `dest` is the destination register written to by an instruction. For any instruction that uses at least 1 register operand, that first operand will be located at the `dest` field. `src2` is next to `dest`, and will be present for any instruction with at least 2 operands. That is because `src2` is effectively the right-hand side of an arithmetic operation, so it is common to both 2-register and 3-register operations. Then, if there is a third register operand, it will be stored in `src1`.

The least consistent field among all encodings is the `funct/imm` field. The `funct` field in each encoding determines *which specific* instruction it is, e.g. add or subtract. Then, the `imm` or `immediate` field determines any immediate value for that instruction. Many instructions have no such immediate value.

The `funct` field may be located almost anywhere within the instruction, including at the location in Fig. 1 but also in various other locations. For example, for LI instructions, the `funct` field is located in bits 28-31, i.e. the location where

`vec` and `size` are usually located. For many instruction that don't have a `flag` bit, the `flag` bit is instead used as the least significant bit of the `funct` field.

However, because there is no real need for consistent `funct` field location in instructions from a micro-architectural perspective, this is not considered (at least by me) to be a significant issue.

The `immediate` field is layed out in a more organized way. In particular, the most-significant bit of it is *always* at the same location, bit 27. This means that any sign-extension hardware can be fed by the same bit of the encoding. Nonetheless, instructions have hugely varying values for the size of the `immediate` field.

### D. MISC instructions (opcode 0000)

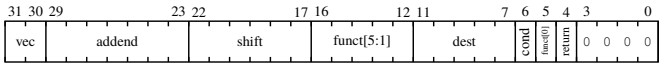


Fig. 2: MISC encoding

The MISC encoding is used to represent various miscellaneous instructions. There are two main categories of MISC instruction: SINGLETON, and the rest.

Any MISC instruction can be made into a return instruction by setting the `return` field of the encoding to 1. Otherwise, it does not have many standardized fields. Some instructions need a register operand and/or vector operand (such as `chkptr`), but the exact layout of these is subject to change.

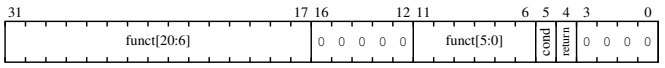


Fig. 3: SINGLETON encoding

SINGLETON is a subset of MISC where the (MISC) `funct` field is all 0. This gives a new 21-bit `funct` field for distinguishing between many, many "one-off" instructions. These instructions take no arguments so must do exactly one thing.

The four existing singleton instructions are `ill`, `nop`, `halt`, and `syscall`. `ill` has funct 0, so its encoding is 32 zeroes. It is simply an illegal instruction, nominally causing some kind of fault, so that if the processor ever tries to execute zeroed-out memory it instantly causes an issue.

`nop` is a simple no-op, `halt` halts the processor (preventing it from doing anything else), and `syscall` nominally performs some kind of syscall.

The exact definition of software interrupts for FaVOR has not been defined yet. For now, the `ill` and `syscall` instructions exist to make simulating the instruction set doable in the GNU binutils infrastructure.

There are also a series of `set` SINGLETON instructions, such as `seteq`. These take the condition flags and set the predication flags based on the corresponding condition. The first `set` instruction has `funct` 16 with the rest coming in order after it so that their `funct`s are nicely aligned for any condition code microarchitecture.

### E. JUMP instructions (opcode 0001)

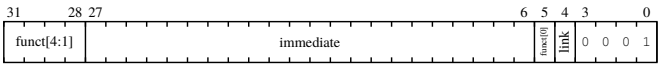


Fig. 4: JUMP encoding

JUMP instructions perform a pc-relative jump. That is, they add the immediate to the program counter. Because the pc is semantically aligned to 4 bytes, the jump can only be in 4-byte increments, so there are no redundant bits inside the encoding.

The `link` or `and_link` field determines whether the jump instruction also updates the `la` register. It is essentially used to turn a jump instruction into a call instruction. In assembly, it is represented by adding the suffix `.l` to a jump instruction.

There are 3 primary varieties of JUMP. JUMP does not directly have a `cond` field, but every JUMP instruction does have a `cond` variant (i.e. one that is only executed if the 0th predication flag is true). However, unlike other instructions, every JUMP instruction *also* has an inverse-cond variant, that is, an instruction that is only executed if the 0th predication flag is false. These exist because JUMPs are the most common kind of conditional instruction, and so building useful boolean logic into them makes sense.

The other kind of variation between JUMP instructions is the various branch instructions. Besides being itself conditional, JUMP instructions can look at the condition flags for an additional condition. The instructions are organized so that predicated JUMPs span `funct` 0-9, inverse predicated span `funct` 10-19, and unpredicated span `funct` 20-31.

In each span, the first instruction is the unconditional jump (`j` in assembly), while the next 9 are conditional branches, organized in the same order as the condition codes. (`beq`, `bne`, etc).

In assembly, the `?` suffix is used to mark a jump predicated, and `~` to mark it inverse-predicated.

Combining a predication suffix with a conditional branch enables jump instructions to essentially perform a logical and, jumping only if the predication flag is true (or false), and only if the branch condition is also fulfilled.

### F. Vector semantics and notation

Because "vectors" in FaVOR are really just combinations of registers, anytime a vector is referred to in a FaVOR instruction it will be simply be a single register name. The rest of the components of the vector are the registers that come sequentially after it in the register file, wrapping around to 0.

This does mean that to locate all the components of a register microarchitecturally requires multiple 5-bit adders. Early designs of FaVOR rejected this in favor of requiring vector registers to be "aligned" to the nearest power-of-two inside the register file. However, this leaves the FaVOR encoding with 6 unused bits–the least-significant bits of the 3 or 4 component vector operands in every single instruction.

It seems much more useful to simply support any register, despite the microarchitectural costs, than to waste bits in the encoding, especially because FaVOR is mostly supposed to

target consumer-level CPUs rather than embedded systems. This also makes the vector instructions extremely flexible, particularly the swizzle instructions.

To denote what is happening with various components in the vectors throughout the rest of this paper, I will use an "indexed" notation such as `dest[0]`. What this refers to is the first component of the `dest` register, so if `dest` is `a3`, then `dest[0] = a3`, `dest[1] = a4`, and so forth. `dest[0]` is equivalent to `dest` in this notation.

The assembler itself does not support such a notation. Instead, registers are simply always addressed by name (e.g. `a0`, `a1`) and the rest of the components are implicitly affected by the instruction. So an instruction like `add.u32x3 a0, a3, a11` is equivalent to the three instructions in Listing 5.

```
add.u32 a0, a3, a11
add.u32 a1, a4, t0
add.u32 a2, a5, t1
```

Listing 5: Semantics of a vector instruction.

### G. INT instructions (opcode 0010)

Opcode `0010` is used for integer instructions with register operands. There are two varieties of instructions, chosen by the `variant` flag: the 3-register variant, and the 2-register variant.

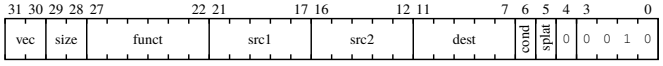


Fig. 5: INT-3 encoding

For 3-operand instructions, the encoding follows the general pattern from Fig. 1. These instructions have 3 registers: a destination, a left-hand side, and a right-hand side. The operation is performed on `src1` and `src2` with the result stored in `dest`.

The `vec`, `size`, and `cond` fields work as discussed previously, and the `funct` field chooses between instructions.

The `splat` field is a new field that enables a second mode for vector operations.

- When `splat` is 0, each component is operated on separately. That is, `dest[i] <- src1[i] op src2[i]`.
- When `splat` is 1, each component is operated on separately, *except* `src2` is sourced from the 0th component every time. That is, `dest[i] <- src1[i] op src2[0]`.

`splat` is indicated in assembly with a `...` suffix on the last operand to the instruction.

```
add.u64x4 a0, a4, a8...
add.u64x4 a0, a4, a8
```

Listing 6: Splat and non-splat vector instructions.

Some examples of 3-operand instructions include `add`, `sub`, `lsh`, `xor`, and `min`. For instructions like `min` where the behavior depends on whether the numbers are interpreted as signed or unsigned, there are two varieties of the instruction, where the `funct` for the signed version equals the `funct` for the unsigned version plus 1. Selecting between the two is done in assembly based on the type suffix of the instruction, e.g. `min.u32` versus `min.s32`.

For instructions where the interpretation of unsigned vs signed does not matter, the type suffix must be specified as `u`.

Most of the arithmetic instructions will update the condition flags when executed. This means, for example, a `sub` instruction can simply be followed by a conditional branch to create a loop, rather than needing a separate instruction to update the condition flags.

The `size` field often can be used as nothing more than a mask: for example, an 8 bit add can be implemented as a 64 bit add that is then truncated to 8 bits. However, for some instructions, such as `rol` or `ror`, the size field meaningfully affects how the instruction functions, and has to be respected in the ALU.

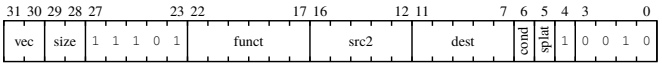


Fig. 6: INT-2 encoding

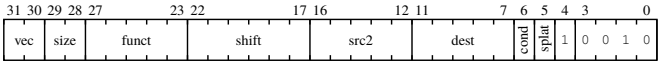


Fig. 7: FIX-2 encoding

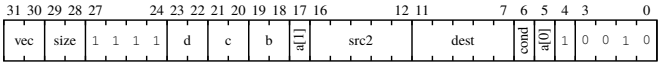


Fig. 8: INT-SWIZZLE encoding

For the `variant = 1` version of the INT opcode, there are three sub-encodings. The main idea is that the `src1` field from INT-3 has been subsumed into a longer `funct` field, spanning bits 17-27.

However, if bits 24-27 of this longer `funct` field are `1111`, then the instruction is actually an INT-SWIZZLE instruction. Similarly, if bits 23-27 are `11101`, the instruction is an INT-2 instruction. Finally, if bits 23-27 are anything else (i.e. not `11101`, `11110`, or `11111`), then the instruction is a FIX-2 instruction.

This is where the fixed point instructions finally come in. These instructions require the 6-bit `shift` field, which essentially determines the format of the fixed point number. For example, for a multiply, the result is shifted right by the `shift` value after the multiply.

Notice that instructions such as `add` and `sub` are not considered fixed point instructions. This is because a fixed point addition is identical to an integer one, so the integer instructions are re-used wherever possible (which also enables using 3 register operands instead of only 2). There are only 29 possible `funct` for the fixed point instructions after subtracting the special ones for INT-2 and INT-SWIZZLE, so they need to be carefully chosen.

Unfortunately, no specific operations for fixed point numbers have been implemented yet. Tentatively, `funct` 0 is a multiply and `funct` 1 is a divide. The fixed point operations may eventually also include vector-specific operations such as dot product.

The INT-2 encoding is used for integer based instructions that only need two arguments. Unlike fixed point numbers, there is no need for the 6 bit `shift` field so many integer instructions can simply be implemented as INT-3 instructions. But some instructions, such as `not` (binary not) and `negate` do not need two operands, so they might as well use this other encoding.

One big set of INT-2 instructions is the comparison instructions. As usual, these come in condition code order, and do two things: compare a vector of values through subtraction, setting the condition flags, and then update the corresponding predication flags based on the condition code specified (e.g. `cmpeq`, `cmpne`). There is also a `cmp` instruction that updates the condition flags but not the predication flags.

Finally, the INT-SWIZZLE instruction performs a swizzle operation. This is essentially a superpowered `mov`, where a destination and source vector are specified, and the components in the destination are updated based on the source, but rearranged in any way.

```
swiz.u64x4 a0, a4, 0123 # copy a4 to a0
swiz.u64x4 a0, a4, 0000 # copy a4... to a0
swiz.u64x4 a0, a0, 3210 # reverse a0
```

Listing 7: Examples of the swizzle instruction.

For INT-SWIZZLE, the `a` field is the offset of the first component in the destination, the `b` field is the second, and so forth. That is, `dest[0] <- src2[a]`, `dest[1] <- src2[b]`, and so forth.

Besides that, INT-SWIZZLE still supports the `vec` field for only swizzling 1, 2, or 3 components if wanted, and it still supports the `size` field which simply truncates the value if it's too big.

### H. FLOAT instructions (opcode 0011)

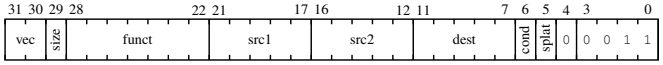


Fig. 9: FLOAT-3 encoding

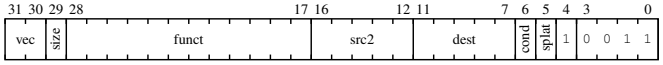


Fig. 10: FLOAT-2 encoding

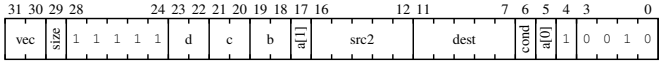


Fig. 11: FLOAT-SWIZZLE encoding

The FLOAT instructions are very similar to the INT instructions, with 2 primary differences:

1) The `size` field is only 1 bit, as the only floating-point formats supported are 32-bit and 64-bit floats.
2) There is no fixed point encoding here, so the FLOAT-2 only has to compete with FLOAT-SWIZZLE for encoding space in the `variant = 1` variant of the instruction.

The `funct` field for FLOAT is 1 bit larger than for INT due to `size` being 1 bit smaller. This also means that FLOAT-SWIZZLE has a prefix of `11111` rather than `1111`, leaving more room in FLOAT-2 for other instructions.

The float instructions have not been nailed down in detail, but they will consist of many arithmetic operations like the integer instructions. Due to the ample space in the FLOAT-2 encoding, it will likely be used for any integer to floating-point converting instructions (and vice versa), as well as for conversions involving fixed point numbers (which could quickly eat up some of the encoding space).

One note is that the register fields for all these instructions are only 5 bits because they each operate on a separate set of registers that is unambiguous due to the nature of the instruction. For example, the FLOAT instruction operands will all be one of the 32 floating-point registers, while the INT instruction operands will all be one of the 32 general registers.

### I. LOAD instructions (opcode 0100)

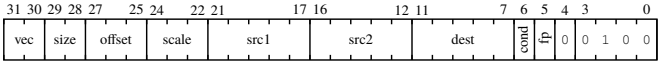


Fig. 12: LOAD encoding (variant 0)

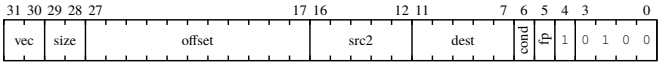


Fig. 13: LOAD encoding (variant 1)

The LOAD instruction very simply loads memory as such:

```
dest <- mem[#src2 + ((src1 << scale) + offset) << size]
```

That is, `src2` is always treated as a base pointer. Then, an offset is computed based on the `src1` operand and the `offset` and `scale` fields. The `src1` offset is scaled by `scale`, such as to index into an array of structs. The `offset` addend (a small signed number) is then added to the scaled `src1`. Finally, the whole computed offset is scaled *again* based on the size of the values we're trying to load–for 1 byte, it is shifted by 0; for 2 bytes, it is shifted by 1, and so forth.

This is, of course, because all loads are aligned–so there is no reason, generally, to specify an offset that is smaller than the kind of value we are trying to load.

Finally, the last few significant bits of the address expression are ignored depending on the alignment of the type. This is notated by the # character which was chosen arbitrarily to represent "this address ignores any bits that are outside the alignment of the type." This feature is the primary way that FaVOR supports pointer tagging.

The `src1` and `src2` registers are always general-purpose registers. The `dest` register is either a general-purpose register or a floating-point register based on the value of the `fp` field.

For vector loads, neither `src2` nor `src1` are treated as vectors, only `dest` is. Essentially, a vector load is treated as follows (for a 4-component vector):

```
dest[0] <- mem[#src2[0] + ((src1[0] << scale) + offset + 0) << size]
dest[1] <- mem[#src2[0] + ((src1[0] << scale) + offset + 1) << size]
dest[2] <- mem[#src2[0] + ((src1[0] << scale) + offset + 2) << size]
dest[3] <- mem[#src2[0] + ((src1[0] << scale) + offset + 3) << size]
```

The second variant of the LOAD instruction enables swapping out the `src1` offset for a much larger immediate offset. In this case, `src1` is effectively treated as `zero`.

The `zero` register itself is very useful for LOAD instructions as it enables loads that do not have any variable offset amount.

Finally, loads are subject to change in future versions of the encoding, as these two variants have not been tested in any C compiler context. Depending on what kinds of offsets and scales are most useful, the load instruction (especially the second variant) may be reworked to be more useful.

### *J. STORE instructions (opcode 0101)*

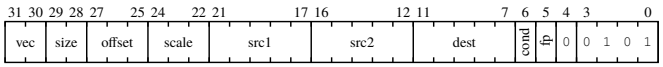


Fig. 14: STORE encoding (variant 0)

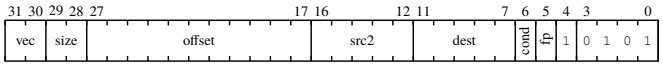


Fig. 15: STORE encoding (variant 1)

STORE instructions work exactly the same way as LOAD instructions, except the data flows from the `dest` register into main memory, rather than flowing from main memory into the `dest` register.

Note that the opcode for STORE is the same as the opcode for LOAD except the least-significant bit is 1.

The encoding for STORE will be updated in lockstep with LOAD if there are future changes. Essentially, LOAD and STORE are only put into separate opcodes to give the fields more useful bits. However, given the dubious usefulness of the current variants, one possibility is that LOAD and STORE will be combined into one opcode with `variant` switching between them in the future.

### *K. LOAD-SPECIAL instructions (opcode 0110)*

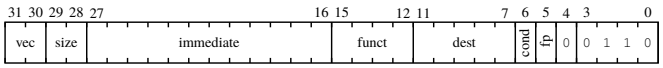


Fig. 16: LOAD-SPECIAL encoding


Fig. 17: LI encoding

The LOAD-SPECIAL opcode is used for two variants of instructions: variant 0, simply called LOAD-SPECIAL, and variant 1, called LI (load immediate).

The LOAD-SPECIAL variant is not yet fleshed out, but is intended for special pc-relative and perhaps `sp`-relative loads. In particular, although the LI instructions enable reasonably low-overhead encoding of constants into the instruction stream, that overhead becomes much more significant when the immediate values are re-used multiple times.

For example, there are some 32-bit constants that require two LI instructions to load. If such a constant is re-used three times, then instead of generated 6 instructions worth of loads (2 each time), we could instead generate 3 instructions worth of loads ("load from pc-relative address") plus 4 bytes for the constant itself. The constant value can be encoded into the instruction stream, and placed after a conditional jump or return statement. This saves 4 bytes of space overall, at the cost of a more complicated instruction.

So, LOAD-SPECIAL is intended for pc-relative loads, for the specific purpose of de-duplicating constant value loads when there are space savings for doing so.

It may also eventually include instructions for loading relative to the stack pointer, as having as large of a range of possible for `sp`-relative loads is useful for function preludes and postludes.

The LI instructions are significantly more flushed out. LI instructions load a 16-bit immediate into any of the registers, with the destination specified by `dest` and `fp` as in LOAD/STORE.

The `funct` chooses between 16 different functionalities. These take the form of three main kinds of instructions: `li_u`, `li_s`, and `li_o`.

- `li_u` loads a value into one of the halfwords (16 bit portions) of the destination register, and sets the rest of the register to 0.
- `li_s` loads a value into one of the halfwords of the destination register, sets any bits more significant than that halfword to 1, and any bits less significant to 0.
- `li_o` or's a value with one of the halfwords of the destination register.

In total, there are 13 implemented LI instructions, outlined in Table V. These instructions can be used to load any 64-bit constant in no more than 4 instructions, but they can also often load a value in fewer.

In particular, any value that has all 0's in the upper bits can be loaded with a sequence that consists of a `li_u` followed by some number of `li_o`. Similarly, any value that has all 1's in the upper bits can be loaded with a `li_s` followed by some number of `li_o`.

32-bit values can also be loaded efficiently thanks to `li0s32`. Essentially, a 32-bit signed value such as −1 can be loaded as `li0s32 0xFFFF`, rather than as a `li0u 0xFFFF ; li0o 0xFFFF`, because `li0s32` will only sign extend to the 31st bit rather than all 64.

TABLE V: LI instructions

| | |
|---|---|
| `li3u` | `dest <- (imm << 48)` |
| `li3o` | `dest <- dest \| (imm << 48)` |
| `li2u` | `dest <- (imm << 32)` |
| `li2s` | `dest <- (imm << 32) \| 0xFFFF000000000000` |
| `li2o` | `dest <- dest \| (imm << 32)` |
| `li1u` | `dest <- (imm << 16)` |
| `li1s` | `dest <- (imm << 16) \| 0xFFFFFFFF00000000` |
| `li1o` | `dest <- dest \| (imm << 16)` |
| `li0u` | `dest <- imm` |
| `li0s` | `dest <- imm \| 0xFFFFFFFFFFFF0000` |
| `li0o` | `dest <- dest \| imm` |
| `li0s32` | `dest <- imm \| 0x00000000FFFF0000` |
| `li0opc` | `dest <- pc + (dest \| imm)` |

One special LI instruction is `li0opc`. Because jump instructions are intrinsically position-independent (their offset is pc-relative), FaVOR is interested in supporting other features of position-independent code. To load a pointer value in a position-independent way, the pointer must be encoded in the instruction stream as an offset from the current program counter. `li0opc` enables this–any LI sequence ending in `li0opc` essentially encodes `dest <- pc + immediate`.

In the encoding itself, the `funct` for `li_u` and `li_s` is always different only by bit 28 (if bit 28 is 1, the instruction is `s`, otherwise it is `u`). This is important for relocation purposes. In particular, to load a pointer value always requires a 64-bit offset from the 62-bit program counter, but it is useful to support pointer loads that only take 2 instructions or 8 bytes (resulting in 32 bits of effective offset value). Most pointers are close enough to the program counter for this to be sufficient (especially given that jumps have even fewer bits than this for their offset). So, 32-bit pointer load sequences will always start with a `li1u` or `li1s`, but need to dynamically switch between them at link time based on whether the pointer value is $<$ the program counter (resulting in a negative offset and `li1s`), or $\geq$ the program counter (resulting in a non-negative offset and `li1u`). This swap between `li1s` and `li1u` will always load a 64-bit offset value, which results in truly position independent code. Putting the distinction between these two instructions in bit 28 allows this to be very easily implemented in the linker as a 1-bit relocation based on bit 31 of the pointer.

The assembler is setup to transform a psuedo-instruction, `li`, into sequences of these LI instructions. There are two main kinds of `li` instruction:

1) `li.u8, li.u16`, etc–these load an absolute value into the destination register. Because the value is known at assemble time, the assembler will dynamically determine how many instructions are actually needed for the constant value and keep it as few as possible.
2) `li.r32, li.r64`–these two are specifically set up for loading pointer values. `li.r32` means to generate a 2-instruction sequence, while `li.r64` means to generate a 4-instruction sequence. Because the pointer values are not known until link time, it is not possible to generate fewer instructions. The existence of both `li.r32` and `li.r64` allows the user to choose which size will be sufficient for their use case.
    - In the future, it is possible that a `li0upc` instruction will be added, allowing for single-instruction pointer loads if the offset fits within a 16-bit immediate. This would result in a new `li.r16` in the assembler.
    - Future work also includes checking for symbols that are known at assemble time and optimizing them similarly to the absolute case.

The load sequence optimization performed by the assembler does a few main things:

- Any all-0 or all-1 halfwords starting from the most significant bit of the instruction are completely skipped and replaced with a `u` or `s` flag for the eventual first instruction in the sequence as appropriate.
- For every other non-zero halfword in the instruction, a `li_o` is generated to load that halfword. This means all-0 halfwords are skipped, resulting in space savings for constants such as `0xAAAA0000BBBB0000` (only requires 2 instructions).
- For 0 values, the assembler still has to generate at least one instruction, so it generates `li0u 0`.

The one optimization opportunity that is not used at all by the assembler is for constants such as `0xA000B000FFFFC000`. This could be generated using Listing 8, but is a very niche optimization (it only applies to constants that have 0xFFFF in their 2nd halfword, and that also have some non-zero data in their other three halfwords, *and* that do not have 0xFFFF anywhere else). As such, this is left for future work.

```
li0s32 0xC000
li2o   0xB000
li3o   0xA000
```

Listing 8: Missed optimization.

Alongside supporting integer values, the LI instruction set also nicely supports floating-point values. In particular, because the floating-point registers are conceptually overlaid on the high bits, the high half-word of the floating-point registers can be loaded with `li3u` for small constants.

For example, the high 16 bits of the double value 1.0 are 0x3ff0, while the rest are all 0, and the high 16 bits of the float value 1.0 are 0x3f80, while the rest are all zero. This means that small floating point values can be loaded in a single instruction: `li3u fa0, 0x3ff0` to load a double value of 1.0, and `li3u fa0, 0x3f80` to load a float value of 1.0. More generally, the nature of the floating-point format where the first few bits of the mantissa are located in the first 16 bits, as well as the exponent and sign bits, mean that many "simple" values can be loaded in only 1 instruction. More LI instructions are only needed to fill out the later bits of the mantissa. This will, of course, be necessary for some common floating-point values, such as "blend factors" between 0 and 1 that are often set as manual constants in game development. But, the design of the LI instructions still manages to keep a number of floating-point constants very easy to load, including essentially any power of two including the very important 1.0.

### L. INT-IMM instructions (opcode 1010)

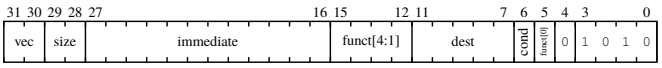


Fig. 18: INT-IMM encoding


Fig. 19: INT-IMM-ADD encoding

These instructions are used to perform arithmetic with immediate values. They only have a single register operand, but do have a vector operand, so they are semantically `dest[i] <- dest[i] op immediate`. This gives room in the encoding for a larger immediate value, while still enabling

vector instructions, which is useful for certain kinds of SIMD code (e.g. tracking a vector of loop indices, where every index would need to be incremented by 1).

Note that the `opcode` for INT-IMM equals the INT opcode plus 8. A similar relationship will exist for an FLOAT-IMM opcode in the future.

The variant 0 encoding supports 32 different operations with its 5 bit `funct` field. The exact list of operations has not been nailed down yet, but will definitely include shifts and other bitwise operations.

The variant 1 encoding is specifically for addition and subtraction. Its `funct` field has been reduced to only 1 bit, giving 4 more bits to the immediate value for a total of 16 bits. For add and subtract, this immediate is always treated as unsigned, as to flip the sign is accomplished by simply switching to the other instruction. Integer add and subtract with immediate are particularly important for managing the `sp` value.

INT-IMM instructions have not yet been implemented in the FaVOR binutils.

### *M. Other opcodes*

All other opcodes are currently unimplemented and reserved. It is important to keep the ISA flexible for future expansion, although in its current state it will likely see changes even to its existing opcodes as well.

There are currently 8 opcodes used plus 1 for the not-yet-designed FLOAT-IMM instructions, leaving 7 opcodes for future expansion.

### *N. GNU binutils & demos*

The GNU binutils implementation for FaVOR is available online [1]. It currently consists of assembler support, (`gas/config/tc-favor.c`, `opcodes/favor-opc.c`), disassembler support (`opcodes/favor-dis.c`), some various relocations for the GNU linker (`bfd/elf32-favor.c`), and support for GNU sim (`sim/favor/interp.c`), although this particular simulation infrastructure

```
1   .data
2   string:
3   .asciz "hello, world\n"
4   .equ strlen, (. - string)
5
6   .text
7   .global _start
8   _start:
9       li.u64 a11, 5     # syscall number: write(fd, buf, len)
10      li.u64 a0, 1      # fd  = 1 (stdout)
11      li.r32 a1, string # buf points to hello world string
12      li.u64 a2, strlen # len = length of string
13
14      syscall           # perform syscall
15      halt

PROBLEMS 14  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

emdash@LabTop:~/favor-isa/tests$ favor-elf-as hello.S -o hello.o
emdash@LabTop:~/favor-isa/tests$ favor-elf-ld hello.o -o hello
emdash@LabTop:~/favor-isa/tests$ favor-elf-run hello
hello, world
emdash@LabTop:~/favor-isa/tests$
```

Fig. 20: assembling & running hello.S

```
1   .data
2   string:
3   .asciz "hello, world\n"
4   .equ strlen, (. - string)
5
6   .text
7   .global _start
8   _start:
9       li.u64 a11, 5       # syscall number: write(fd, buf, len)
10      li.r32 a1,  string  # buf points to hello world string
11      li.u64 a2,  strlen  # len = length of string
12
13      li.u64 t0,  10      # loop counter
14      li.u64 t1,  1       # constant 1 (don't have addi/subi yet)
15  loop:
16      # Syscall puts return value in a0, so we must re-load it each time.
17
18      li.u64 a0, 1        # fd of 1 = stdout
19      syscall
20
21      # Make the string shorter each loop.
22      add.u64 a1, a1, t1
23      sub.u64 a2, a2, t1
24
25      # Decrement loop counter then jump.
26      sub.u64 t0, t0, t1
27      bne loop
28      halt

PROBLEMS 14  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

emdash@LabTop:~/favor-isa/tests$ favor-elf-as hello-loop.S -o hello-loop.o
emdash@LabTop:~/favor-isa/tests$ favor-elf-ld hello-loop.o -o hello-loop
emdash@LabTop:~/favor-isa/tests$ favor-elf-run hello-loop
hello, world
ello, world
llo, world
lo, world
o, world
, world
 world
world
orld
rld
emdash@LabTop:~/favor-isa/tests$
```

Fig. 21: assembling & running hello-loop.S

is not widely used (as compared to e.g. QEMU [4]) or well documented.

The implementation currently contains support for SINGLETON, JUMP, INT-3, INT-2, and LI instructions, as well as number of relocations, including the relocations necessary for JUMP instructions as well as for pc-relative LI sequences.

This enables some basic demo programs that can print strings (using the `write` system call, as the GNU sim enables easy system call support).

Fig. 20 shows usage of the `li.r32` instruction to load a pointer to a string, as well as usage of other load instructions and `syscall`.

Fig. 21 shows usage of `sub` plus the `bne` instruction to implement a loop. This could also be done using `cmpne` followed by `j?`, but that would require one additional instruction. This demonstrates how the additional conditional branches instructions can make code more compact.

Fig. 22 shows usage of `j.l` as well as `ret` to implement a function call. The function argument is loaded into `a0` as per the calling convention.

Fig. 23 shows how the psuedo-instruction `li` is assembled into an optimized sequences of LI instructions.

## V. Related Work

FaVOR is perhaps most closely related to RISC-V [2], which was a primary source of reading for existing architectural practice. Some of the features that FaVOR inherited from

RISC-V include an opcode + funct encoding for the various kinds of instructions, as well as the idea of placing every sign bit for immediate values in the same place in the instruction encoding to reduce the cost of sign extension architecture.

The Clockhands [5] ISA presents an approach for instruction encoding that completely obviates the need for register renaming in the microarchitecture. An approach like this was considered for FaVOR, due to the apparent efficiency of the method, but was ultimately rejected as it doesn't fit together well with the vector system of FaVOR.

The ARM architecture includes a "top byte ignore" [6] feature that has some capabilities for pointer tagging. Unlike FaVOR, this feature requires some environmental support for the program trying to use it–that is, it requires both kernel and compiler support. FaVOR is in a similar boat where compiler support is necessary to easily use the semantics of the tagged pointers, but FaVOR does not require any kernel support whatsoever for its pointer tagging features.

The Simple-V system [7] enables using scalar registers as vector registers in a manner similar to FaVOR, although it is innovative in that the vector length is not fixed as it is in FaVOR. Other similar features include the existence of a swizzle instruction.

FaVOR owes much of its toolchain's completeness to the GGX [8] or Moxie ISA, and the tutorial series created alongside it. FaVOR does not take much direct inspiration from Moxie, although some of its earlier designs shared some

```
.data
string:
.asciz "hello, world\n"
.equ strlen, (. - string)

.text
.global _start
_start:
    li.u64 a0, 0 # hello, world
    j.l hello_world
    li.u64 a0, 7 # world
    j.l hello_world
    li.u64 a0, 1 # ello, world
    j.l hello_world
    li.u64 a0, 12 # \n
    j.l hello_world
    halt

# hello_world(offset): Takes as an argument an offset into "hello world"
# then prints that string.
hello_world:
    # Load the pointer & add the offset
    li.r32 a1, string
    add.u64 a1, a1, a0

    # Subtract the offset from len
    li.u64 a2, strlen
    sub.u64 a2, a2, a0

    # Load the rest of system call arguments
    # Note that this clobbers a0
    li.u64 a11, 5
    li.u64 a0, 1
    syscall
    ret
```

```
PROBLEMS 14  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

emdash@LabTop:~/favor-isa/tests$ favor-elf-as hello-call.S -o hello-call.o
emdash@LabTop:~/favor-isa/tests$ favor-elf-ld hello-call.o -o hello-call
emdash@LabTop:~/favor-isa/tests$ favor-elf-run hello-call
hello, world
world
ello, world

emdash@LabTop:~/favor-isa/tests$
```

Fig. 22: assembling & running hello-call.S

```
tests > ASM li-opt.S
.global _start
_start:
    li.u64 a0, 0xFFFF # expect: 1
    li.u64 a1, 0xFFFFFFFF # expect: 2
    li.u64 a2, 0xFFFFFFFFFFFF # expect: 3
    li.u64 a3, 0xFFFFFFFFFFFFFFFF # expect: 1

    syscall

    li.u64 a0, 0xFFFF000000000000 # expect: 1
    li.u64 a1, 0xFFFFFFFF00000000 # expect: 2
    li.u64 a2, 0xFFFFFFFFFFFF0000 # expect: 3
    li.u64 a3, 0xFFFFFFFFFFFFFFFF # expect: 1

    syscall

    li.u64 a0, 0 # expect: 1
    li.u32 a1, 0 # expect: 1
    li.u16 a2, 0 # expect: 1
    li.u8  a3, 0 # expect: 1

    syscall

    li.u64 a0, 0xFFFF0000FFFF0000 # expect: 2

    syscall

    li.u32 a0, 0xFFFFFFFF # expect: 1 (lios32)

    syscall

    li.u32 a0, 0xFFFF0FFF # expect: 1
```

```
emdash@LabTop:~/favor-isa/tests$ favor-elf-as li-opt.S -o li-opt.o
emdash@LabTop:~/favor-isa/tests$ favor-elf-objdump -d li-opt.o

li-opt.o:     file format elf32-favor


Disassembly of section .text:

00000000 <_start>:
   0:   16 f0 ff 0f     li0u        a0, 0xffff
   4:   96 f0 ff 4f     li1u        a1, 0xffff
   8:   96 f0 ff 2f     li0o        a1, 0xffff
   c:   16 f1 ff 8f     li2u        a2, 0xffff
  10:   16 f1 ff 6f     li1o        a2, 0xffff
  14:   16 f1 ff 2f     li0o        a2, 0xffff
  18:   96 f1 ff 1f     li0s        a3, 0xffff
  1c:   80 01 00 00     syscall
  20:   16 00 00 20     li0o        a0, 0x0
  24:   96 00 00 20     li0o        a1, 0x0
  28:   16 01 00 10     li0s        a2, 0x0
  2c:   96 f1 ff 1f     li0s        a3, 0xffff
  30:   80 01 00 00     syscall
  34:   16 00 00 00     li0u        a0, 0x0
  38:   96 00 00 00     li0u        a1, 0x0
  3c:   16 01 00 00     li0u        a2, 0x0
  40:   96 01 00 00     li0u        a3, 0x0
  44:   80 01 00 00     syscall
  48:   16 f0 ff 6f     li1o        a0, 0xffff
  4c:   80 01 00 00     syscall
  50:   16 f0 ff ff     li0s32      a0, 0xffff
  54:   80 01 00 00     syscall
  58:   16 f0 ff f0     li0s32      a0, 0xfff
emdash@LabTop:~/favor-isa/tests$
```

Fig. 23: assembling & disassembling li-opt.S

features (e.g. labeling the opcode-like field of those early designs as a `type` or `kind` instead of an `opcode`).

Finally, perhaps the most obviously related feature is the SIMD feature [9] in one of the most mainstream consumer architectures: x86. x86 SIMD, however, serves different purposes than FaVOR–although FaVOR's vector instructions are decent at SIMD, SIMD is certainly not the focus. x86′s SIMD is powerful for exactly what it says it is–single instruction, multiple data–but doesn't have the component-by-component processing available to FaVOR.

## VI. Conclusion & Future Work

In this paper, I presented the FaVOR ISA, a new instruction set architecture supporting vector operations, address masking, and fixed point arithmetic, as well as an associated GNU binutils toolchain.

The FaVOR ISA is an extremely flexible ISA as it stands, with instructions able to operate on many different data types in various vector configurations (including "splat" configurations), all fitting within a 32 bit fixed-size instruction encoding. Some of this flexibility comes from sacrificing additional register operands, with its fixed point instructions only having 2 register operands and its immediate instructions only having 1. Nonetheless, FaVOR gets a lot done with a small number of instructions, and leaves almost half of the encoding space open to future expansion.

The ISA remains incomplete, as its implementation is missing several of the fundamental opcodes and several opcodes are subject to change.

Future work for FaVOR includes patching these missing features, as well as finalizing anything in the ISA that remains subject to change. Ideally this work occurs alongside development of a GCC backend for FaVOR so that the remaining pieces of the FaVOR design can be empirically designed through analysis of their impact on real-world code.

Additionally, a true microarchitectural implementation on FaVOR, such as on an FPGA, would reveal any obvious microarchitectural holes in its current design, enabling it to be a stronger ISA overall.

## References


[1] Benjamin Wall and GNU binutils contributors, "Fork of GNU's binutils-gdb repository with added favor ISA support.." Accessed: May 07, 2025. [Online]. Available: https://github.com/favor-isa/binutils-gdb/

[2] Derek Atkins *et al.*, *The RISC-V Instruction Set Manual*, vol. 1. pp. 10–25. Accessed: May 07, 2025. [Online]. Available: https://riscv.github.io/riscv-isa-manual/snapshot/unprivileged/

[3] James Orr, "Linux Kernel Code Error Checking." Accessed: May 07, 2025. [Online]. Available: https://classes.engineering.wustl.edu/cse422/code_pointers/05_kernel_code_error_checking.html

[4] Fabrice Bellard *et al.*, "QEMU." Accessed: May 07, 2025. [Online]. Available: https://www.qemu.org/

[5] Koizumi *et al.*, "Clockhands: Rename-free Instruction Set Architecture for Out-of-order Processors," *56th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 1–16, Oct. 2023, Accessed: May 07, 2025. [Online]. Available: https://dl.acm.org/doi/pdf/10.1145/3613424.3614272

[6] "Armv8.5-A Memory Tagging Extension," *ARM*. Accessed: May 07, 2025. [Online]. Available: https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Arm_Memory_Tagging_Extension_Whitepaper.pdf

[7] libreSOC contributors, "SV Overview." Accessed: May 07, 2025. [Online]. Available: https://libre-soc.org/openpower/sv/overview/

[8] Anthony Green, "How To Retarget the GNU Toolchain in 21 Patches." Accessed: May 07, 2025. [Online]. Available: https://github.com/atgreen/ggx

[9] "Intel® Intrinsics Guide." Accessed: May 07, 2025. [Online]. Available: https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html