CS535 Project Proposal

RISC V[ECTOR]

Siddarth Suresh, Benjamin Dunaisky

March 9th, 2025

General overview:

Our architecture is *general purpose* and a clean-sheet design. Its *distinguishing features* are that it will include special registers and instructions for operating on *vector* data types, as well as a scriptable simulator.

The *word size* is 32 bits (4 bytes). As typical for RISC architectures, our ISA will use a *fixed* instruction encoding and a *load/store architecture* for categorizing instructions.

Types and type operations

All integer types are *signed* and *32 bits*. Signed bits are represented using *two's complement.*

The *arithmetic operations to be supported on the integer type* are: add, subtract, multiply, divide, modulo, arithmetic right shift, left shift, and the *logical operations to be supported are:* bitwise and, or, not, xor.

The *operations to be supported on the vector type* are: vector load, vector store, element-wise add, element-wise subtract, element-wise multiply (dot product), and element-wise divide.

Registers

To support the integer and vector data types, the architecture includes 24 general-purpose registers: 16 integer registers (x0 to x15) and 8 vector registers (v0 to v7). Vector registers are 256 bits (8 words).

Register	Description
x0	hardwired 0
x1	link register

x2	stack pointer
x3	condition code register
x4	vector length register
x5…x15	integer general purpose
v0…v7	vector general purpose

A special register which holds the program counter is also provided.

The condition code register

The condition code register maintains a set of four bits that the arithmetic and compare operations set:

Bit 0: *greater than*
Bit 1: *equal to*
Bit 2: *underflow*
Bit 3: *overflow*
Bits 4-31: empty

CMP and CMPV is the only instruction that sets the greater than and equal to bits. Most arithmetic operations set the underflow/overflow bits. There is a separate conditional branch instruction for each bit. This is also documented below.

Instructions & addressing modes

Memory model and addressing modes

The maximum number of operands per instruction is three, fetching a *single instruction per word*. It uses a Princeton *memory organization* and its *address unit* is a full word. The address range is 64 kilobytes. (16384 words)

Similar to RISC-V, our architecture supports *three addressing modes: register, immediate, and displacement*. The displacement mode contains a 16-bit displacement field to access the entire 64 kilobytes address range.

Register indirect addressing is accomplished by placing a value of 0 in the 16-bit displacement field, and *absolute addressing* is accomplished by using the constant 0 register as the base register. The addressing modes used for each instruction are encoded into the opcode.

Instruction Types

Being a *load/store architecture*, the supported instructions for ALU operations, loads, stores, and jumps can be grouped into four types: register to register (R-TYPE), ALU immediates and loads/stores (I-TYPE), and jumps/subroutine management (J-TYPE).

	31 22	21 17	16 12	11 7	6 2	1 0
R-TYPE 00	empty	destination 1	source 2	source 1	opcode	type

	31 16	15 11	10 6	5 2	1 0
I-TYPE 01	immediate	destination 1	source 1	opcode	type

	31 11	10 6	5 2	1 0
J-TYPE 10	displacement	base address	opcode	type

Immediate fields and accessing invalid memory addresses

All immediate fields are treated as a signed integer. In the case an instruction (jump, load, etc.) attempts to access an address larger than the address space, the memory address that is *actually* accessed is:

```
REQUESTED_ADDRESS mod TOTAL_ADDRESS_SPACE.
```

In the case an instruction attempts to access an address lower than zero, the address that is actually accessed is

```
((REQUESTED_ADDRESS mod TOTAL_ADDRESS_SPACE) + TOTAL_ADDRESS_SPACE) mod
REQUESTED_ADDRESS.
```

Instructions supported

A comprehensive list of instructions is listed below. *All undocumented invalid instructions* (including non-existent opcodes or other nonsensical instructions) *will be treated as a no-op*.

Mnemonic	Opcode	Format	Operation
ADD	00001	R	Adds the value in source register 1 to source register 2. Sets the result to the destination register. Sets both the overflow and underflow condition codes to either 0 or 1 depending on if one occurs.
SUB	00010	R	Subtracts the value of source register 2 from source register 1. Sets the result to the destination register. Sets both the overflow and underflow condition codes to either 0 or 1 depending on if one occurs.
MUL	00011	R	Multiplies the value in source register 1 to source register 2. Sets the result to the destination register. The destination register holds the least significant 32 bits in case of overflow. The upper 32 bits are discarded. Sets both the overflow and underflow condition codes to either 0 or 1 depending on if one occurs.
QUOT	00100	R	Divides the value in source register 1 with the value in source register 2. Sets the destination register with the *quotient* of the result. Division by 0 sets all the bits in the destination register to 1 and sets the overflow condition code in the condition register. Always sets the underflow condition to 0. *Division by the zero register results in a halt.*
REM	00101	R	Divides the value in source register 1 with the value in source register 2. Sets the destination register with the *remainder* of the result. Division by 0 sets all the bits in the destination register to 1 and sets the overflow condition code in the condition register. Always sets the underflow condition to 0. *Division by the zero register results in a halt.*
SFTR	00110	R	Arithmetic right shifts the value in source register 1 by the value stored in source register 2. Sets the result to the destination register. Shifts in the right direction shifts all of the bits to the right and fills the upper bits with a copy of the previous most significant bit.

			Shifting by a negative value is treated as a shift in the opposite direction. Shifting by a value larger than 32 bits will effectively clear the register to the smallest positive number (0, in two's complement representation) or -1 in case source register 1 was negative. If the shift causes the most significant bit to change from a 0 to a 1, then the overflow condition is set to 1 and the underflow condition is set to 0. If the shift causes the most significant bit to change from a 1 to a 0, then the underflow condition is set to 1 and the overflow condition is set to 0.
SFTL	00111	R	Arithmetic left shifts the value in source register 1 by the value stored in source register 2. Sets the result to the destination register. Shifts in the left direction shifts all of the bits to the left and fills the lower bits with a zero. Shifting by a negative value is treated as a shift in the opposite direction. Shifting by a value larger than 32 bits will effectively clear the destination register. If the shift causes the most significant bit to change from a 0 to a 1, then the overflow condition is set to 1 and the underflow condition is set to 0. If the shift causes the most significant bit to change from a 1 to a 0, then the underflow condition is set to 1 and the overflow condition is set to 0.
AND	01000	R	Performs a bitwise AND between the values in source register 1 and source register 2. Sets the result in the destination register. Always sets the overflow and underflow conditions to 0.
OR	01001	R	Performs a bitwise OR between the values in source register 1 and source register 2. Sets the result in the destination register. Always sets the overflow and underflow conditions to 0.
NOT	01010	R	Performs a bitwise NOT for the value in source register 1. Sets the result in the destination register. Values in source register 2 are ignored. Always sets the overflow and underflow conditions

			to 0.
XOR	01011	R	Performs a bitwise XOR between the values in source register 1 and source register 2. Sets the result in the destination register. Always sets the overflow and underflow conditions to 0.
ADDV	01100	R	Performs element-wise addition between the vector stored in source vector register 1 and the vector stored in source vector register 2. Sets the result in the destination vector register. The exact number of words operated on is determined by the vector length register. If the vector length register is zero, the destination register is cleared. If the value in the vector length register is greater than 8, it is capped at 8. Sets both the overflow and underflow condition codes to either 0 or 1 depending on if one occurs in any of the elements.
SUBV	01101	R	Performs element-wise subtraction of the vector stored in source vector register 1 from the vector stored in source vector register 2. Sets the result in the destination vector register. The exact number of words operated on is determined by the vector length register. If the vector length register is zero, the destination register is cleared. If the value in the vector length register is greater than 8, it is capped at 8. Sets both the overflow and underflow condition codes to either 0 or 1 depending on if one occurs in any of the elements.
MULV	01110	R	Performs element-wise multiplication between the vector stored in source vector register 1 and the vector stored in source vector register 2. Sets the result in the destination vector register. The exact number of words operated on is determined by the vector length register. If the vector length register is zero, the destination register is cleared. If the value in the vector length register is greater than 8, it is capped at 8. *Each resulting word stored in the destination register is limited to its least significant 32 bits in the event of an overflow.*

			Sets both the overflow and underflow condition codes to either 0 or 1 depending on if one occurs in any of the elements.
DIVV	01111	R	Performs element-wise division between the vector stored in source vector register 1 and vector stored in source vector register 2. Sets the *quotient* of the division (element-wise) in the destination vector register. The exact number of words operated on is determined by the vector length register. If the vector length register is zero, the destination register is cleared. If the value in the vector length register is greater than 8, it is capped at 8. *Division by 0 within a vector sets all the bits in the corresponding word in the destination vector register to 1 and sets the overflow condition code in the condition register. Always sets the underflow condition to 0.*
CMP	10000	R	Sets the greater than condition bit in condition register if value in source register 1 is greater than value in source register 2. Sets the equality condition bit in condition register if value in source register 1 is equal to value in source register 2.
CEV	10001	R	Sets the equality condition bit in the condition register if the value in source vector register 1 is equal to value in source vector register 2. The exact number of words operated on is determined by the vector length register. If the vector length register is zero, the instruction is treated as a no-op. If the value in the vector length register is greater than 8, it is capped at 8.

I Type

Mnemonic	Opcode	Format	Operation
LOAD	0001	I	Loads the value present at the memory address calculated by the sum of the value in the source register and the immediate value into the destination register.

LOADV	0010	I	Loads the value present at the memory address calculated by the sum of the value in the source register and the immediate into the destination vector register. The exact number of words copied is determined by the vector length register. If the value in the vector length register is greater than 8, it is capped at 8.
ADDI	0011	I	Adds value in the sign-extended immediate to the value in source register and stores the result of the sum in the destination register. Sets the overflow/underflow condition codes to either zero or one depending on if one occurs.
SUBI	0100	I	Subtracts value in the sign-extended immediate from the value in source register and stores the result of the difference in the destination register. Sets the overflow/underflow condition codes to either zero or one depending on if one occurs.
SFTRI	0101	I	Arithmetic right shifts the value in source register 1 by the sign-extended immediate. Sets the result to the destination register. Shifts in the right direction shifts all of the bits to the right and fills the upper bits with a copy of the previous most significant bit. Shifting by a negative value is treated as a shift in the opposite direction. Shifting by a value larger than 32 bits will effectively clear the register to the smallest positive number (0, in two's complement representation) or -1 in case source register 1 was negative. If the shift causes the most significant bit to change from a 0 to a 1, then the overflow condition is set to 1 and the underflow condition is set to 0. If the shift causes the most significant bit to change from a 1 to a 0, then the underflow condition is set to 1 and the overflow condition is set to 0.
SFTLI	0111	I	Arithmetic left shifts the value in source register 1 by the sign-extended immediate. Sets the result to the destination register. Shifts in the left direction shifts all of the bits to the left and fills the lower bits with a zero. Shifting by a negative value is treated as a shift in

			the opposite direction. Shifting by a value larger than 32 bits will effectively clear the destination register. If the shift causes the most significant bit to change from a 0 to a 1, then the overflow condition is set to 1 and the underflow condition is set to 0. If the shift causes the most significant bit to change from a 1 to a 0, then the underflow condition is set to 1 and the overflow condition is set to 0.
ANDI	1000	I	Performs a bitwise AND between the value in source register and sign-extended immediate and stores the result in the destination register.
ORI	1001	I	Performs a bitwise OR between the value in source register and sign-extended immediate and stores the result in the destination register.
XORI	1010	I	Performs a bitwise XOR between the value in source register and sign-extended immediate and stores the result in the destination register.
STORE	1011	I	Stores the value present in source register in memory at the address calculated by sum of value in base address register and value in displacement field in memory.
STOREV	1100	I	Stores the value present in the source vector in memory starting at the address calculated by sum of value in base address register and value in displacement field. The exact number of words copied is determined by the vector length register. If the value in the vector length register is greater than 8, it is capped at 8.

J Type:

Mnemonic	Opcode	Format	Operation
JMP	0001	J	Performs unconditional jump to the address in memory calculated by sum of the value in base address register and value in the immediate field.
JRL	0010	J	Performs unconditional jump to the address in

			memory calculated by sum of the current PC and value in the immediate field. The value in the base register is ignored.
JAL	0011	J	Sets the return address register to the memory address of the next instruction and performs an unconditional jump to the address in memory calculated by sum of the value in base address register and value in the immediate field.
BEQ	0100	J	Checks the EQ (equal) condition in the condition code register and branches to the memory address calculated by sum of value in PC and the value in the displacement field.
BGT	0101	J	Checks the GT (greater than) condition in the condition code register and branches to the memory address calculated by sum of value in PC and the value in the displacement field.
BUF	0110	J	Checks the UF (underflow) condition in the condition code register and branches to the memory address calculated by sum of value in PC and the value in the displacement field.
BOF	0111	J	Checks the OF (overflow) condition in the condition code register and branches to the memory address calculated by sum of value in PC and the value in the displacement field.
PUSH	1000	J	Increments the stack pointer by four bytes, writes the value in the base address register to the memory address pointed to by the stack pointer.
POP	1001	J	Loads the memory address pointed to by the stack pointer into the base address register, then decrements the stack pointer by four bytes.

Memory subsystem, stack management & subroutine support

The architecture will utilize a *single level cache*, the size of which is controlled by environment variables set by the user in the simulator. The size of the cache can only be modified after the simulator is reset through the GUI. It will be 2-way *set associative, write back, and write allocate on a write miss*. The replacement policy is least recently used. The size of the cache line will be 4 words.

Our architecture makes use of *stack-based memory allocation* and provides a special stack pointer (x1) register to support subroutine calls in conjunction with the push and pop instructions. *A single dedicated return register is provided* to support subroutine calls and returns with the JAL instruction. It is up to the programmer or compiler to manage pushing/popping the return register on the stack for deeper subroutine calls. The programmer also will likely want a frame pointer for this, but the convention used is up to them.

Special features of the simulation

As a stretch goal, and provided the simulator is sufficiently fast, we will *attempt to build a GUI that provides a full python scripting interface* for demoing purposes. The inspiration is the scripting functionality in GDB.

In the GUI, a 'load script' button will be provided while the simulator is not running. Scripts will be written in python, and consist of a list of functions corresponding to features of the simulator: load program, set breakpoint, run for X clock cycles, print address range, start simulation, and others. Each action will be run in sequence automatically, and will run to completion unless the simulator executes a HALT instruction or a special PAUSE function is called from the script. The demonstration can be resumed or aborted using a button in the GUI.

Management plan

The main program driver, memory simulation, and 5 stage pipeline will be *implemented in C++*. Separate modules for the simulation GUI will be written as Python extensions. The GUI itself will be *written using the Tkinter library*, and will display a scrollable list of the executing program's instructions, registers, cache, memory, and a control panel.

The GUI will be run concurrently on a separate thread to keep the application responsive, but the buttons for viewing and interacting with the state of the *simulator will be disabled while the simulator is running*. A button to send a halt signal to the simulation will be provided while it is running.

We believe this approach leverages both the speed of C++ while still benefiting from Python's efficient development cycle and rich libraries for GUI applications.

We will use a version control system and github for storage of project code and documentation files. The team will maintain a list of active issues to work on, and utilize unit testing using the Catch2 library. Pull requests will always be used to ensure merged code is always functional and peer reviewed. The team will communicate via email, text, and phone, and meet in person Tuesday and Saturday at 11am to review our status and plan work for the next week, or Zoom if that is not possible.

Initially, we will work together to create a set of github issues related to initial APIs for interfacing between the memory and pipeline in C++. We will also create issues relating to the API between the simulator and Python process responsible for the GUI, using the Python.h library to convert the minimal number of data structures necessary for displaying the memory layout. We plan to each claim issues on both sides, so that we each maintain a full understanding of how the simulator works together, and each part is built up simultaneously in pieces.

Finally, as Siddarth starts developing the assembler, Benjamin will continue building out the instruction set. Then we will divide the benchmark work between us. To show the effect of our vector extension, we will compare the benchmarking performed for matrix multiplication using the vector extension and non-vector baseline.

At the end, Siddarth will be responsible for the portion of the report that describes the simulator and using it through the UI while Benjamin focuses on reporting the effects of the benchmarking under the different modes. We will each write a section about what we learned, and who ended up doing what on the project.