ELEC590 - RISC-V research

Fredrick Chiu

Introduction:

This report is the research report for the RISC-V research in the ELEC590 in Spring 2025, including a scalar core design and investigation of RISC-V vector extension (RVV).

Section 1. Scalar processor design

Design overview:

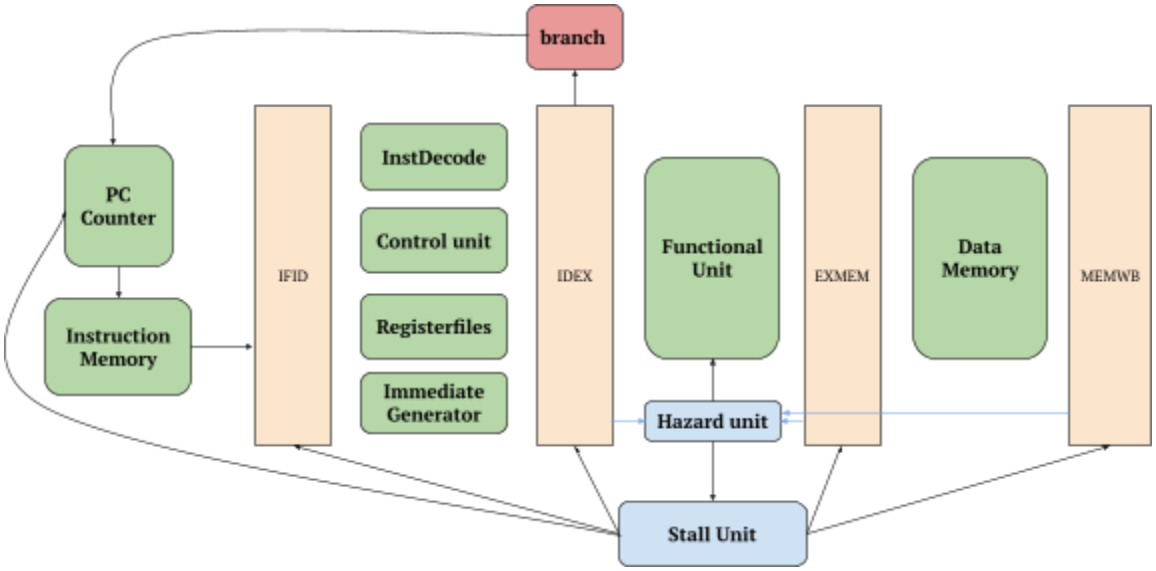

fig 1. A simplified overview of the scalar core design

The design is composed of the following modules:

Memory, Immediate, Control units:

- **Program counter**
- **Data memory**
- **Instruction memory**
- **Immediate generator**
- **Register files**
- **Control unit**
- **MUX**

Execution/Functional units:

- **ALU**
- **ALU control unit**
- Multiplier - designing

Pipeline registers:

- **pipeline registers**

Hazard handling units:

- **Stall unit**
- **Forwarding unit**
- **Load-Use hazard handler**

Other units:

- **Two include files for parameters and wire declarations**
- branch predictor - designed but not integrated
- Cordic - not designed
- Vector extension - designing

where the modules in bold are designed and integrated and at least partially verified. The remainder is either not integrated or not designed yet, or being designed now.

Simulation:

To avoid being too lengthy, please check out the slides and the testbench codes in github. Most of the module-wise simulation tests each module's functionality in a simple way, e.g. using a repeated pattern of inputs and expecting certain outputs.

The pipelined top module simulation was originally using Verilator, but now there are some problems due to shared usage of Verilator between this project and the other project. Instead, the pipelined top module was tested using a pre-loaded 6-instruction assembly which intentionally gives rise to a load-use hazard. The load-use hazard only arises when the pipelined core functions correctly. See load-use hazard in later subsection.

Hazard handling mechanism:

This implementation deals with hazards which occur when the following conditions are satisfied:

- MEM hazard
  - if (MEM/WB.RegWrite
    and (MEM/WB.RegisterRd ≠ 0)
    and not(EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd ≠ ID/EX.RegisterRs1))
    and (MEM/WB.RegisterRd = ID/EX.RegisterRs1)) ForwardA = 01
  - if (MEM/WB.RegWrite
    and (MEM/WB.RegisterRd ≠ 0)
    and not(EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd ≠ ID/EX.RegisterRs2))
    and (MEM/WB.RegisterRd = ID/EX.RegisterRs2)) ForwardB = 01

fig 2. The conditions of forwarding due to hazard

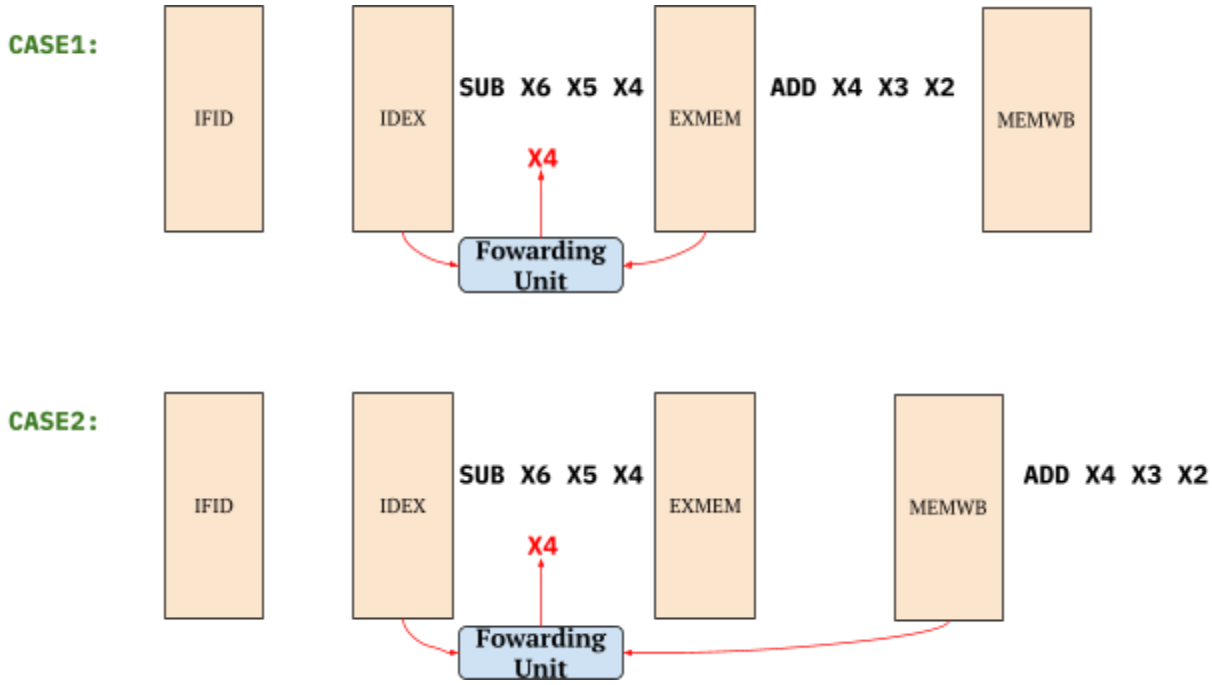

fig 3. A demonstration of the conditions of forwarding

The above figure shows two circumstances where we need to deal with hazards by forwarding. A forwarding unit is inserted so there is no need to stall the pipeline stages.

LOAD-USE Hazard Handling mechanism:

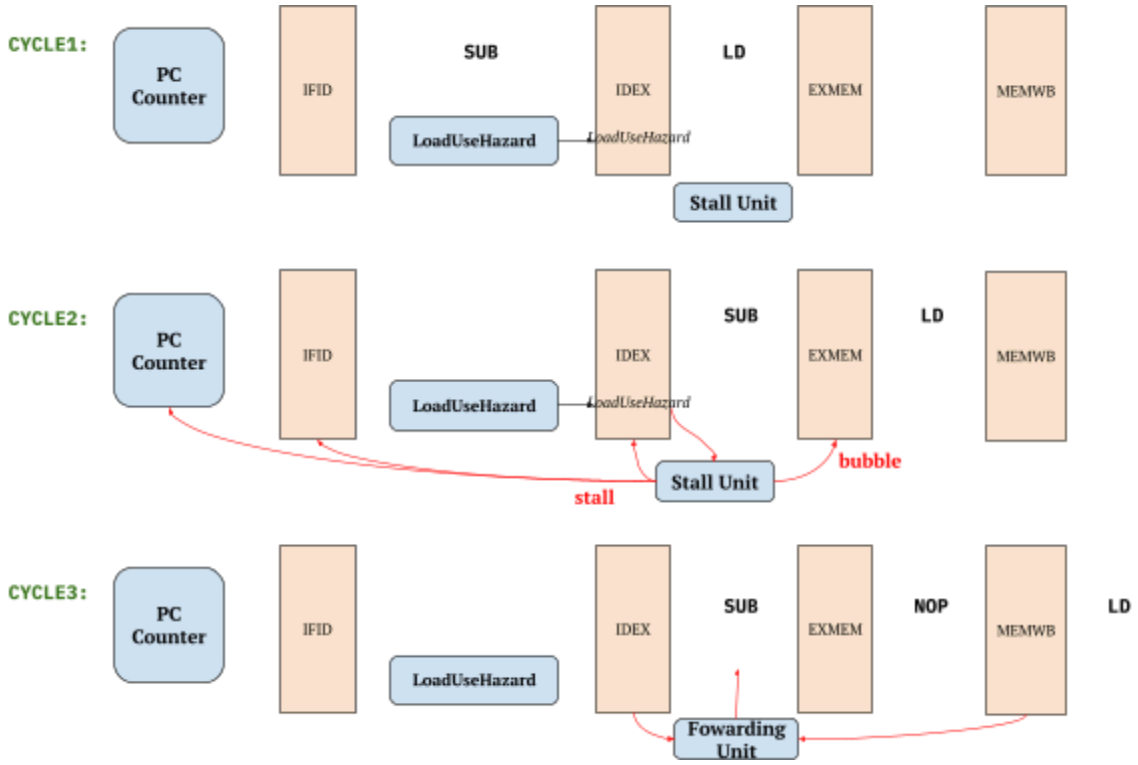

fig 4. The mechanism of load-use hazard handling in consecutive cycles

Load-Use hazard happens when we access a data one cycle right after loading it from data memory. Since the critical path of keeping the pipeline stages going on might be too long, capping the maximum clock frequency, a strategy is required. Stalling and Bubbling come in

handy to make the hazards addressable by the normal hazard handling unit.

To do this, two modules called LoadUseHazard and stall are used. The former detects the Load-Use Hazard and passes a flag to the next stage, and the stall unit stalls or bubbles the stages based on the flag.

Section 2. RISC-V Vector investigation

`Vector machine background:`

The RISC-V vector draft was released nearly 10 years ago, but was ratified around 2021. The contents were not stable and constantly changed until the ratified version came out. After it was released, works from different institutions and companies sprang up, but to an extent the knowledge of how to efficiently and effectively implement it remains inaccessible to amatuer makers and developers without a team. This section aims to cover the fundamental concepts and terminology of vector processing.

`RISC-V vector extension status quo:`

RVV ratified version 1.0 defined V-extension and other subextensions, along with various instructions and hardware requirements. For example, Zve* sub-extensions are designed for embedded systems.

Now, the RISC-V organization is working version 1.1, which includes additional sub-extensions for specific applications.

Now, many of the open-source RISC-V toolchains, e.g. clang, LLVM, have already supported the compilation of RVV. However, the RVV intrinsics remain the best way to run programs on RVV units due to its low-level wrapping of vector assembly.

Vector speedup:

Vector processors achieve speedup by exploiting data-level parallelism. Different from typical SIMD, where multiple data from different sources are executed in a single instruction, similarly, vector processors use SIMD in time, where a single instruction acts on multiple data from some vector sources (see binary and ternary vector instructions).

However, vector processors cannot achieve unlimited speedup. This can be easily proved by Amdahl's law, where parallel speedup is capped by a sequential part of the program.

Zve32x:

Zve32x is a vector sub-extension for embedded systems, and it can be integrated with scalar cores with XLEN = 32 or 64. The 32 in Zve32x means the minimum VLEN = 32, and the x means it only executes

integer data. The Zve32x are required to execute the following instructions:

- all vector load and store instructions
- all vector integer instructions
- all vector configuration instructions
- all vector fixed-point arithmetic instructions
- all vector integer single-width and widening reduction instructions.
- all vector mask instructions
- all vector permutation instructions

Vector terminology & parameter:

The following list some of the most important parameters of RVV:

- **VLEN** - number of bits in power of 2 in each of the vector registers
  - ELEN ⩽ VLEN ⩽ 2^16
- **ELEN** - The maximum size in power of 2 in bits of a vector element that any operation can produce or consume
  - The maximum width of a lane
  - ELEN ⩾ 8
- **DLEN** - total data path widths in lanes in bits (ELEN × #lanes)
- **SEW** - Selected Element Width (vsew[2:0] = 8 to 64)
  - **width of the currently selected element** within the vector register
- **LMUL**: Vector Register Group Multiplier (1/8 to 8)
  - To **group multiple vector registers together** and perform operations.
- **VLMAX**: Maximum Operable Elements
  - **maximum number of elements** that can be operated on **with a single vector instruction** given SEW and LMUL
  - **VLMAX = LMUL × VLEN / SEW**
  - **Total vector length > VLMAX => Strip mining** (software)
- **VL**: Vector Length - vl CSR
  - An unsigned integer specifying the **total number of elements to be updated (operated)** with results from a vector instruction

- **VSTART**: Starting Element - vstart CSR
    - the **index of starting element** for a particular instruction's operation within the vector register.

Typical Vector components:

Typically, a vector consists of vector registers that store vector data for vector instructions, vector functional units (VFU) to execute computing instructions, and vector load-store unit to carry out vector load/store instructions.

An RVV unit is required to implement **32 vector data registers**, each with VLEN-bit width, and **7 unprivileged Control & Status Registers (CSRs)**.

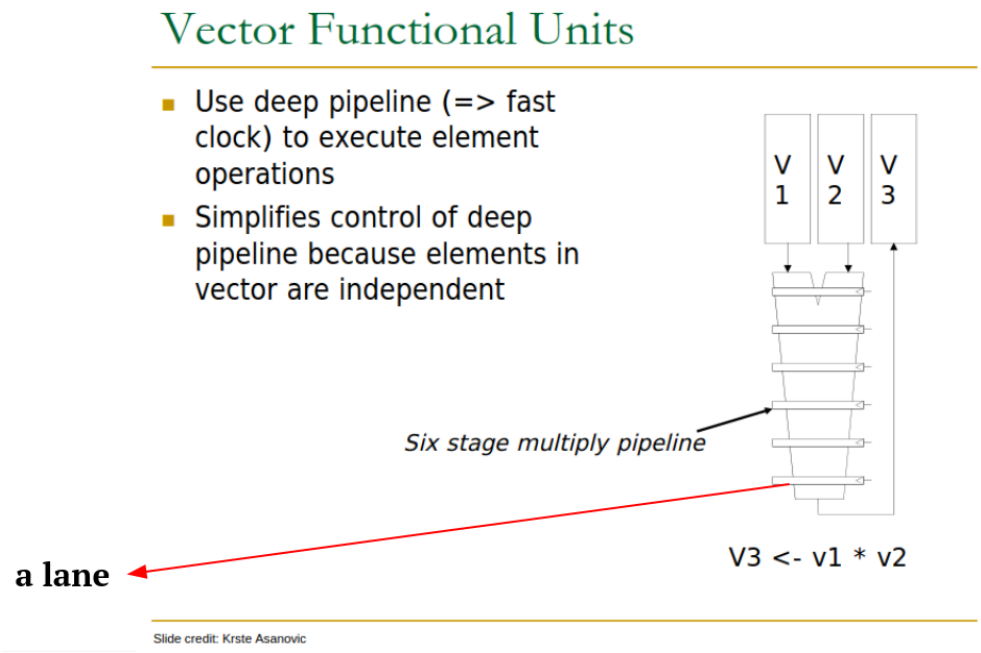

fig 5. A lane in a vector functional unit [0]

A VFU is composed of multiple lanes, where a lane carries out an element-wise operation every cycle.

RVV CSRs:

Table 1. New vector CSRs

Address	Privilege	Name	Description
0x008	URW	vstart	Vector start position
0x009	URW	vxsat	Fixed-Point Saturate Flag
0x00A	URW	vxrm	Fixed-Point Rounding Mode
0x00F	URW	vcsr	Vector control and status register
0xC20	URO	vl	Vector length
0xC21	URO	vtype	Vector data type register
0xC22	URO	vlenb	VLEN/8 (vector register length in bytes)

fig 6. The list of CSRs required by RVV, where URW means read/write and URO means read-only

The vstart register specifies the index of the starting element of the vector register that a vector instruction acts on. The vtype register sets up LMUL, SEW, and hence VLMAX (since VLMAX = LMUL * VLEN/SEW). The vl register determines the number of data an instruction operates, and it can be updated by CSR instructions such as vsetvli.

Key features of vector processor:

The following are some key features of typical vector processing units:

- Pipelining
  - higher clock rate
- Chaining
  - forwarding data
- Vector register grouping
  - Use vlmul[2:0] grouping register with **LMUL**
  - allow instructions to act on more elements (if fixed SEW & VLEN)
- Vector strip mining
  - Total vector length (AVL) > vlmax e.g. use loop
  - Use register vl for first iteration, and then VLMAX
  - what if **ELEN > VLEN**
    - RVV now doesn't support this
- Scatter/gather operations
  - when vector data are distributed across memory
  - combine elements into vector registers
- Vector mask instructions
  - only operates on a portion of elements

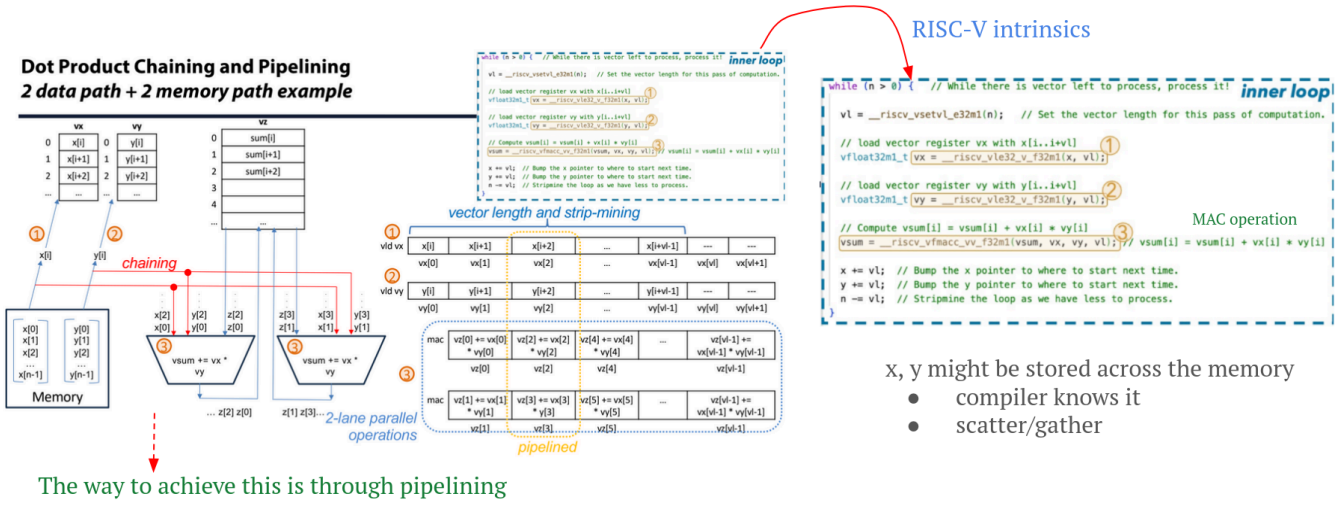

fig 7. key features of a VPU

Figure credit: Ray Simar, RVR, Rice University

Vector instruction types:

In general, vector instructions fall into three categories as follows:

- Memory type
  - e.g. load/store
  - allows scatter/gather by segment/index stride load/store
  - vmem-format.adoc
- Configuration setting type
  - CSR instructions, e.g. vsetvli rd rs1 vtypei sets up vl, AVL, and vtype
  - vcfg-format.adoc
- Arithmetic type
  - e.g. +-×/
  - **suffix vv, vx , vi** indicate vector-vector, vector-scalar, vector-immediate respectively
  - **binary type** operates on elements from two source registers; **ternary type** operates on elements from three source registers
  - can obtain double-width vector operation results by **widening instruction**, and single-width results the other way around by **narrowing**
  - **Reduction instructions** allow operation on vector data with a scalar value using a binary operator (&, |, sum, min/max), to get or reduce to a scalar result. For example, sum-reduction.
  - **Permute instructions** are used to move data from vector register to scalar register, or from within a vector register to different position in itself or to other vector registers
  - valu-format.adoc

Section 3. Future Plan

The following are figures of the future design goals of the scalar core as well as the RVV unit:

`Scalar core design:`

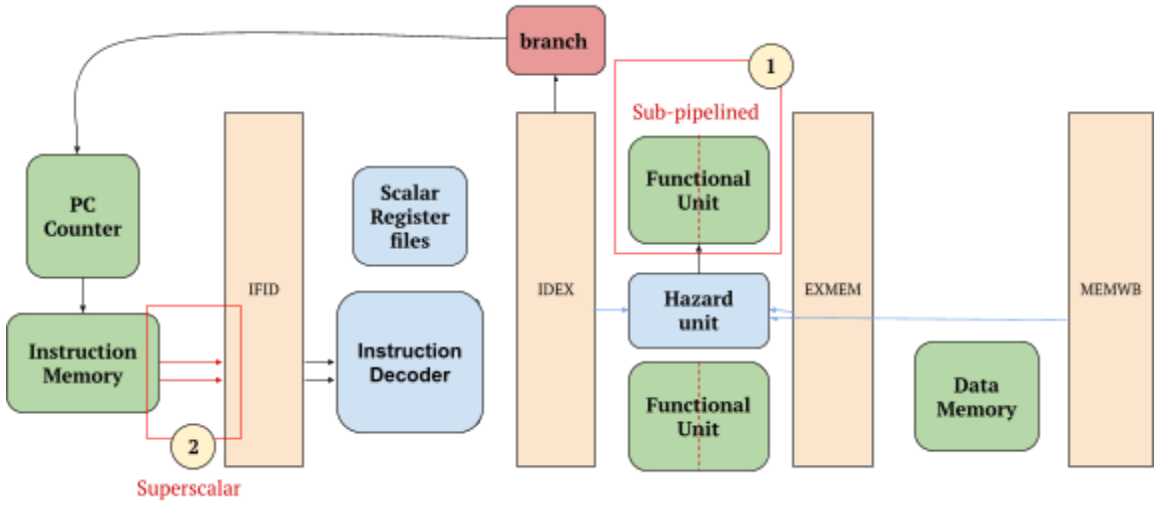

fig 8. Design plan for the scalar core

Currently, the multiplier is implemented brutally using the * operator in verilog, which typically incurs a significant overhead by synthesizer, limiting the maximum frequency. Therefore, designing a good multiplier is important to optimize the EX-stage or the whole scalar core. To this end and give the design more flexibility, now I plan to pipeline the EX-stage or only the multiplier, but this is expected to complicate the design and timing might be an issue as discussed with Prof. Yu Kee.

On the other hand, if possible, in the far future I plan to build a cordic unit and make the core superscalar, and these should not affect the original design (not modifying much of the original design).

RVV unit design:

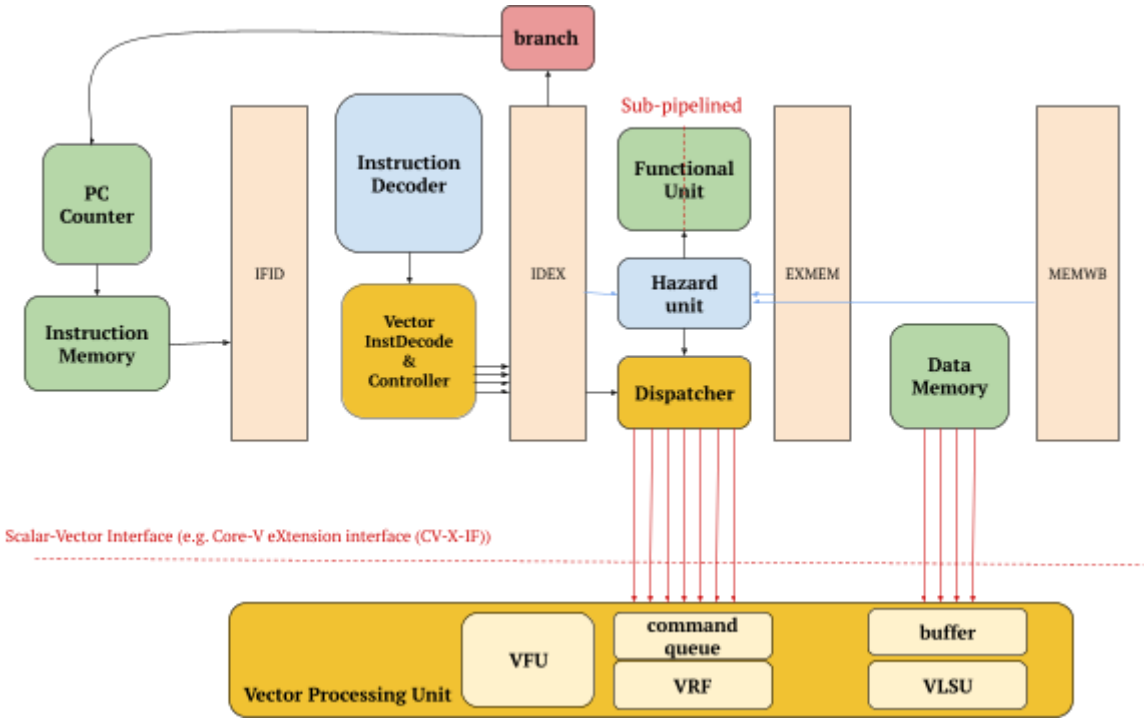

fig 9. Design plan for the RVV unit

On the original scalar core side, the RVV draft design is designed to integrate a vector instruction decoder, a controller (dedicated or integrated into the scalar one), a dispatcher to send commands and parameters to the VPU, and a memory interface.

For the interface between the Scalar unit and the VPU, a potential design can be using the CV-X-IF Interface, an interface whose purpose

is to extend custom or standardized extension without the need to modify the CPU design. It is an interface standard released by OpenHW group, and they even propose to make it a standard under RISC-V organization.

Additionally, some other parameters such as caches should also be taken into account.

Reference:

[0] Onur Mutlu, CMU, https://course.ece.cmu.edu/~ece740/f13/lib/exe/fetch.php%3Fmedia=seth-740-fall13-module5.1-simd-vector-gpu.pdf