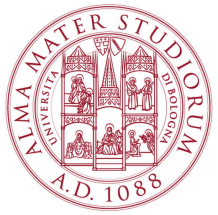
An MLIR Backend for Linear Algebra Microkernel Compilation

Sasha Lopoukhine
Anton Lydike
Alban Dutilleul
Chris Vasiladiotis
Tobias Grosser

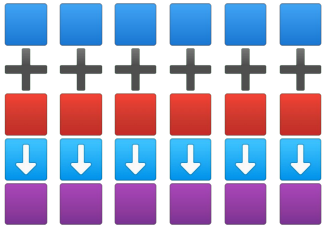
Federico Ficarelli

Josse Van Delm



Accelerated Linear Algebra Microkernels

$Z_i = X_i + Y_i$



Restricted Domain

Structured Input

High-Level Semantic Information

Optimal Hardware Utilisation

Control Flow and Memory Access is Costly

```
void fadd64(float* x, float* y, float* z) {


  for (uint32_t i = 0; i < 128; ++i) {
    z[i] = x[i] + y[i];
  }
}
```

Control Flow and Memory Access is Costly

```
void fadd64(float* x, float* y, float* z) {

  for (uint32_t i = 0; i < 128; ++i) {
    z[i] = x[i] + y[i];
  }
}
```

Costly Loop Management

Control Flow and Memory Access is Costly

```
void fadd64(float* x, float* y, float* z) {

  for (uint32_t i = 0; i < 128; ++i) {

    z[i] = x[i] + y[i];

  }

}
```

Costly Loop Management

Hard-To-Predict Address Offsets

RISC-V *(1202 Cycles)*

```
  li a5, 0
  li a6, 1024

.loop:
  add a4, a1, a5
  add a3, a0, a5

  flw ft1, 0(a4)
  flw ft0, 0(a3)

  add a4, a2, a5

  addi a5, a5, 8

  fadd.d ft2, ft0, ft1

  fsw ft2, 0(a4)

  bne a5, a6, .loop
```

1. Loop control:

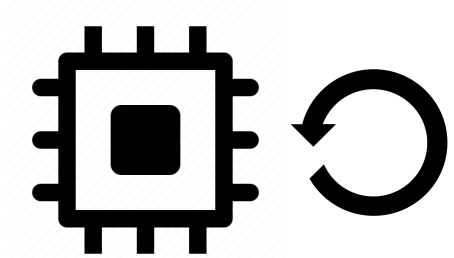
- load bounds
- iterator increment
- branching condition

2. Compute memory addresses for I/O

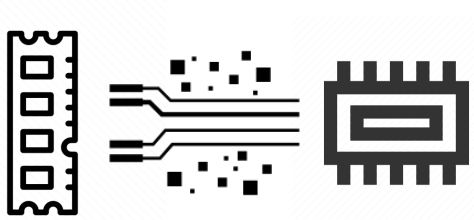
3. Load/Store data

Snitch: Hardware Acceleration

Open-Source ISA



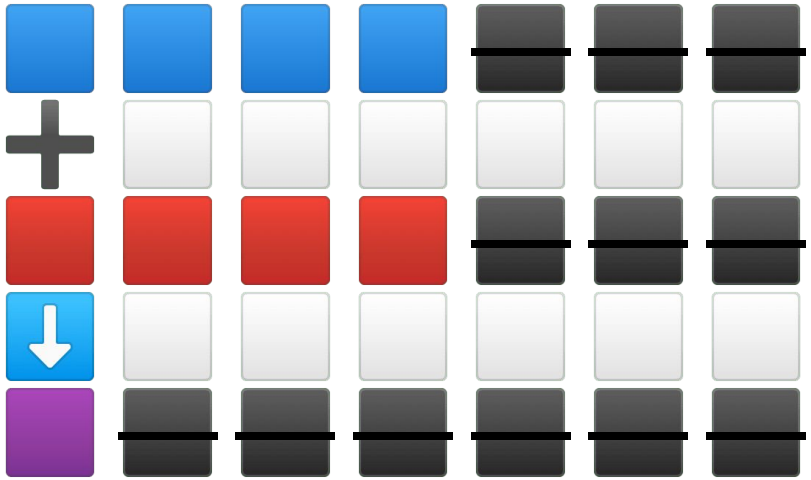
Hardware Loops ⟶ reduced loop management



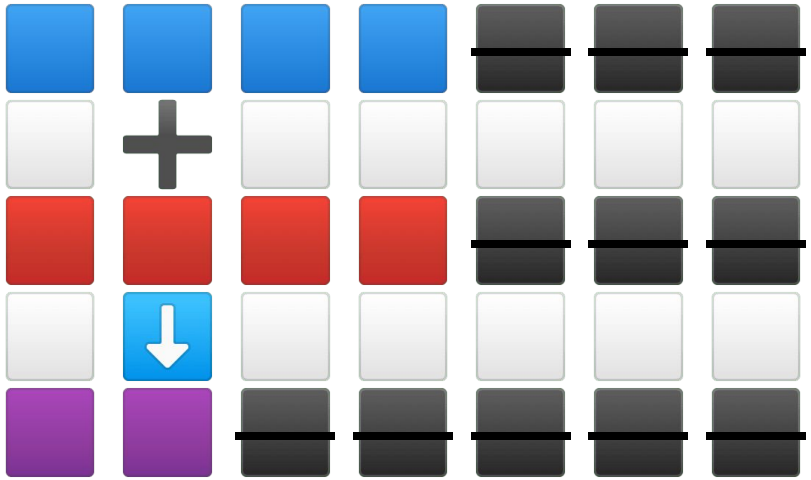
Streaming Registers ⟶ accelerate memory

No Compiler to Target Snitch Hardware

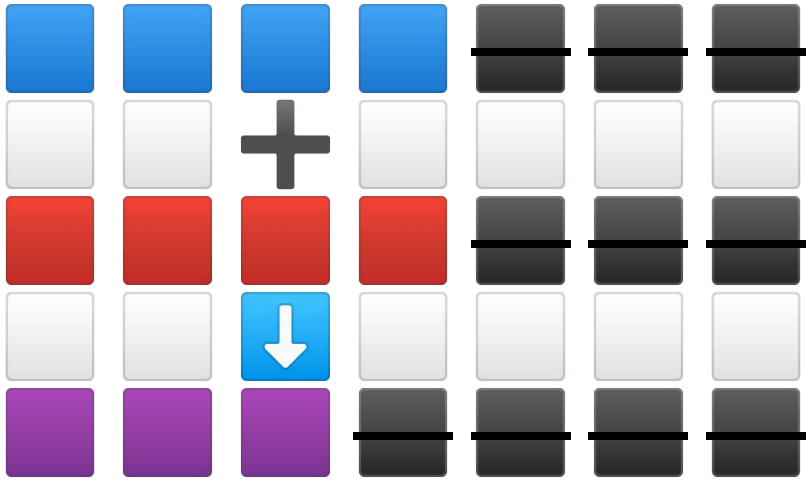
Explicit Store/Load at Offset



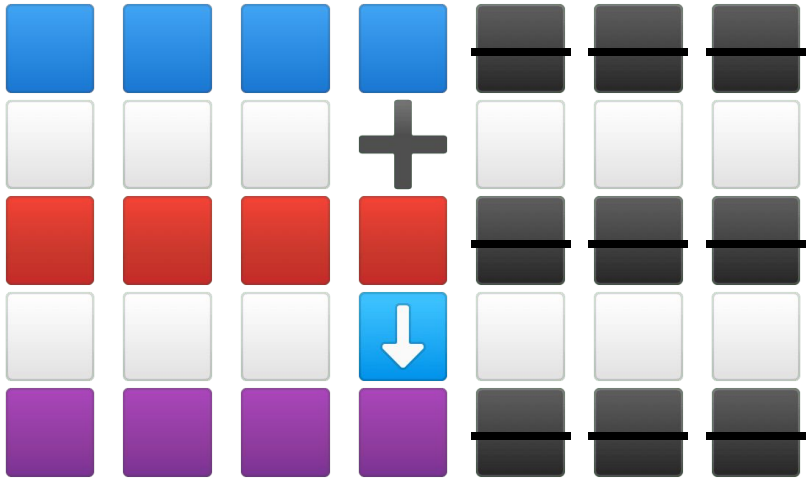
Explicit Store/Load at Offset



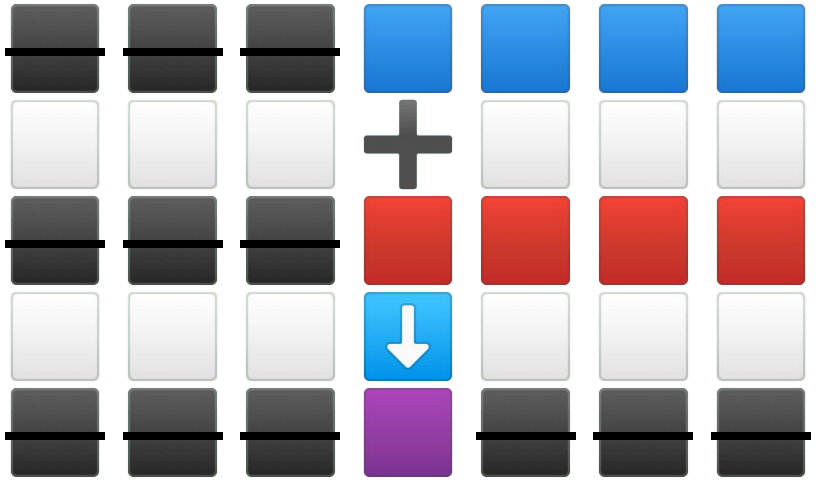
Explicit Store/Load at Offset



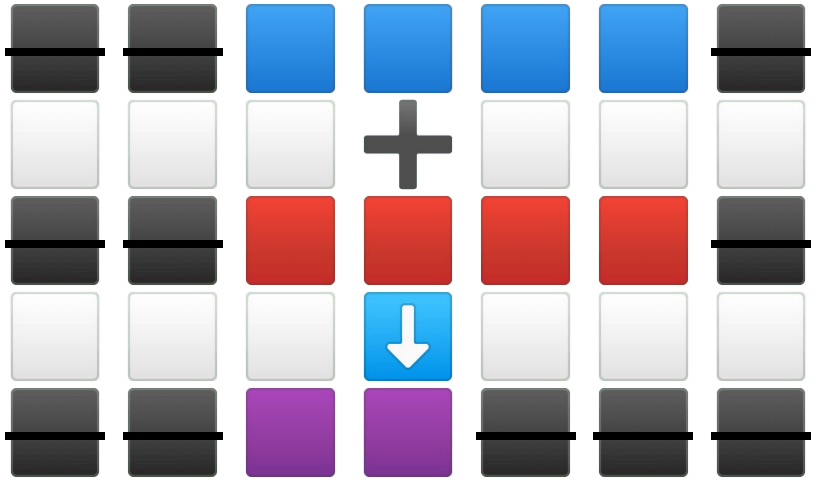
Explicit Store/Load at Offset



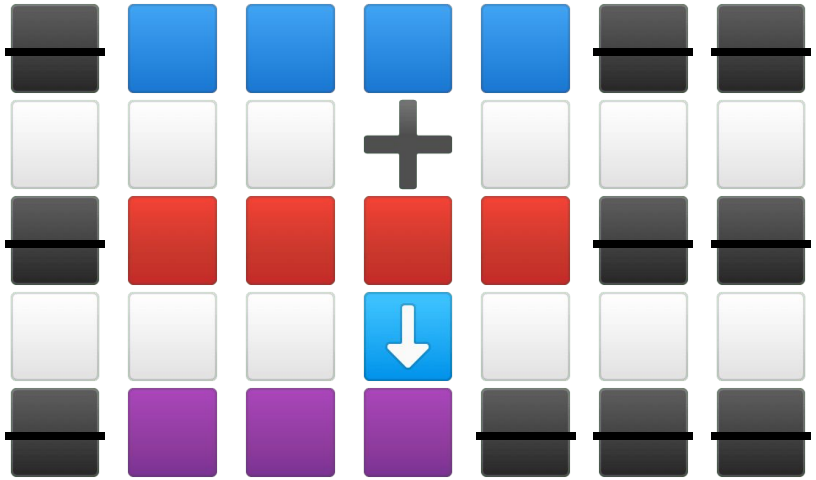
Snitch **S**treaming **S**emantic **R**egisters (**SSR**)



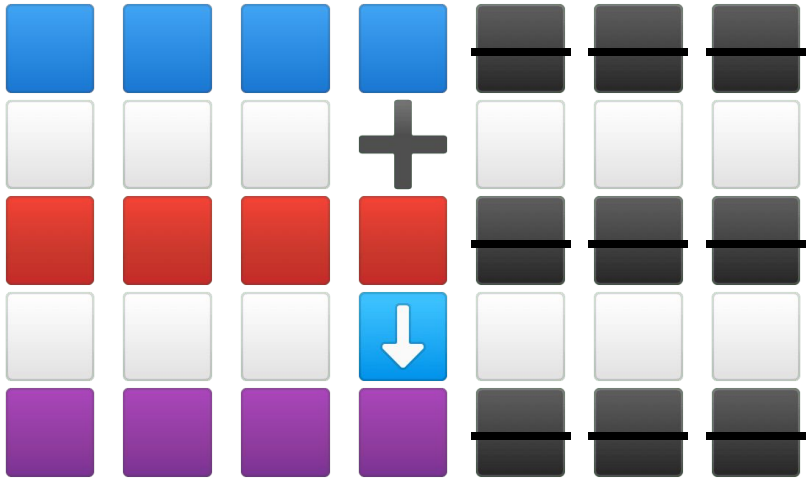
Snitch **S**treaming **S**emantic **R**egisters (**SSR**)



Snitch **S**treaming **S**emantic **R**egisters (**SSR**)



Snitch **S**treaming **S**emantic **R**egisters (**SSR**)



RISC-V + SSR *(441 Cycles)*

```
  # Begin Streaming from/to ft0,ft1,ft2

  li a5, 0
  li a6, 128

.loop:
  addi a5, a5, 1

  fadd.d ft2, ft0, ft1

  bne a5, a6, .loop

  # End Streaming
```

SSR extension handles:

- Computing memory addresses
- Load/Store of operands

Snitch **F**P **Rep**etition Instruction (**FREP**)

*"Repeat the next **k** instructions **n** times"*

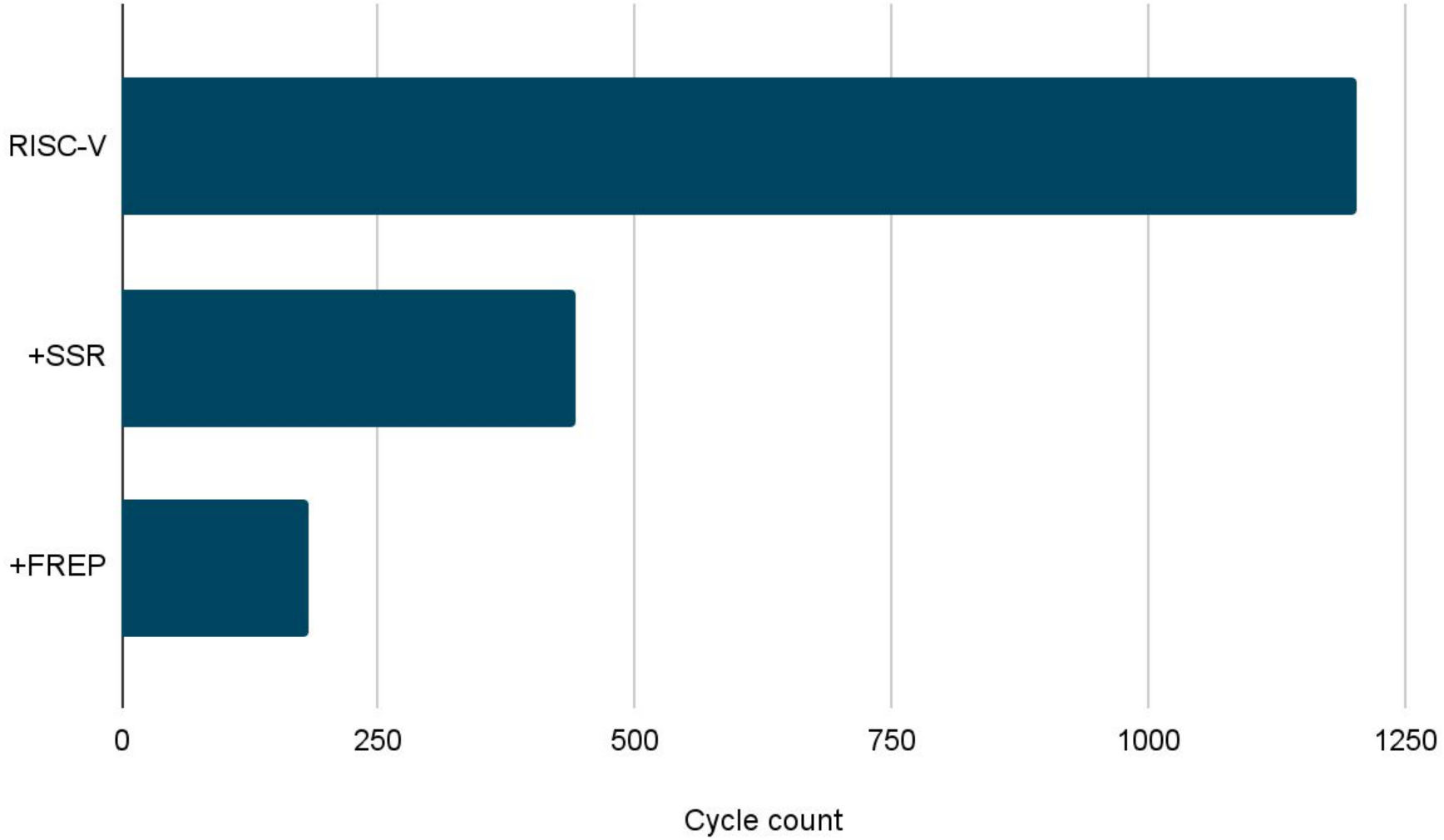
RISC-V + SSR + FREP *(183 Cycles)*

```
# Begin Streaming from/to ft0,ft1,ft2

li a5, 0
li a6, 127

frep.o a6, 1, 0, 0
fadd.d ft2, ft0, ft1

# End Streaming
```

FREP extension handles:

- pipeline repetition of FP instruction

Performance of Vector Addition on Occamy



C with Assembly needed to Target Snitch

```
const uint32_t m = 8, n = 16;
const uint32_t niter = M * N;

snrt_ssr_loop_2d(SNRT_SSR_DM_ALL, m, n, sizeof(double) * n, sizeof(double));

snrt_ssr_read(SNRT_SSR_DM0, SNRT_SSR_2D, x);  // ft0
snrt_ssr_read(SNRT_SSR_DM1, SNRT_SSR_2D, y);  // ft1
snrt_ssr_write(SNRT_SSR_DM2, SNRT_SSR_2D, z); // ft2

snrt_ssr_enable();

asm volatile(
  "frep.o %[nfrep], 1, 0, 0 \n"
  "fadd.d ft2, ft0, ft1 \n"
  :
  : [nfrep] "r"(niter - 1)
  : "ft0", "ft1", "ft2", "memory");

snrt_ssr_disable();
```

1. extensive use of intrinsics
2. inline assembly

linalg dialect ($Z_i=X_i+Y_i$)

```
linalg.generic #fadd_attributes
ins(%X, %Y: memref<128xf64>, memref<128xf64>)
outs(%Z: memref<128xf64>) {
^bb0(%x: f64, %y: f64, %z: f64):
  %r0 = arith.addf %x, %y : f64
  linalg.yield %r0 : f64
}
```

stream dialect

```
%0 = stream.strided_read %X, [8, 16], [16, 1]
      : (memref<8x16xf64>) -> !stream.readable<f64>
%1 = stream.strided_read %Y, [8, 16], [16, 1]
      : (memref<8x16xf64>) -> !stream.readable<f64>
%2 = stream.strided_write %Z, [8, 16], [16, 1]
      : (memref<8x16xf64>) -> !stream.writable<f64>
stream.generic x 128
ins (%0, %1 : !stream.readable<f64>, !stream.readable<f64>)
outs (%2 : !stream.writable<f64>) {
^0(%x : f64, %y : f64):
  %r0 = arith.addf %x, %y : f64
  stream.yield %r0 : f64
}
```

snitch_stream dialect

```
%3 = snitch_stream.strided_read %0, [8, 16], [128, 8]
   : !stream.readable<!riscv.freg<ft0>>
%4 = snitch_stream.strided_read %1, [8, 16], [128, 8]
   : !stream.readable<!riscv.freg<ft1>>
%5 = snitch_stream.strided_write %2, [8, 16], [128, 8]
   : !stream.writable<!riscv.freg<ft2>>
snitch_stream.generic x 128
ins(%3, %4 : ...)
outs(%5 : ...) {
^0(%x : !riscv.freg<ft0>, %y : !riscv.freg<ft1>):
  %r0 = riscv.fadd.d %x, %y : !riscv.freg<ft2>
  snitch_stream.yield %r0 : !riscv.freg<ft2>
}
```

snitch_runtime dialect

```
snrt.ssr_loop_2d (%15, %16, %17, %18, %19)
snrt.ssr_write (%15, %20, %2)
snrt.ssr_enable
%22 = riscv.li 127 : !riscv.reg<>
riscv.frep_outer %22, 0, 0 {
^0(%x : !riscv.freg<ft0>, %y : !riscv.freg<ft1>):
    %r0 = riscv.fadd.d %x, %y : !riscv.freg<ft2>
    riscv.frep_yield %r0 : !riscv.freg<ft2>
}
snrt.ssr_disable
```

Register Allocation in the riscv dialect

Leverage MLIR structure:

1. Walk IR and Nested Regions
2. Using the Static Single Assignment (SSA) form

   → Allocate register upon a use

   → Free register upon a definition

3. Structured Control Flow

   → Allocate loop operands before loop body

register allocation

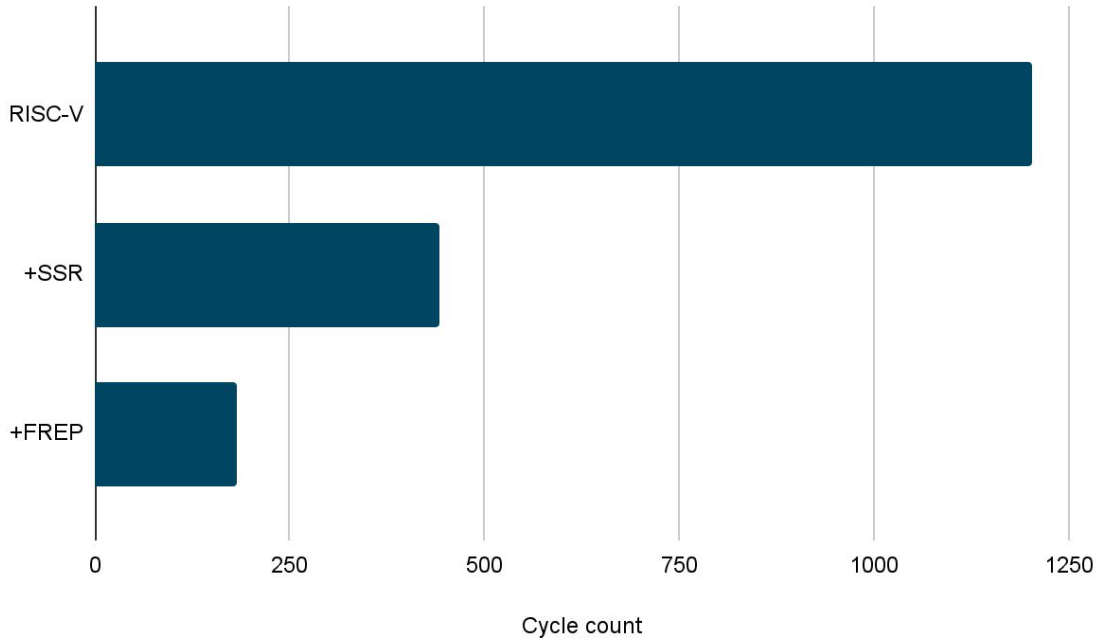
```
snrt.ssr_loop_2d (%15, %16, %17, %18, %19)
snrt.ssr_write (%15, %20, %2)
snrt.ssr_enable
%22 = riscv.li 127 : !riscv.reg<t0>
riscv.frep_outer %22, 0, 0 {
^0(%x : !riscv.freg<ft0>, %y : !riscv.freg<ft1>):
    %r0 = riscv.fadd.d %x, %y : !riscv.freg<ft2>
    riscv.frep_yield %r0 : !riscv.freg<ft2>
}
snrt.ssr_disable
```

Call runtime library

```
riscv_func.call @snrt_ssr_loop_2d (%15, %16, %17, %18, %19) : …
riscv_func.call @snrt_ssr_write(%15, %20, %2) : …
riscv_func.call @snrt_ssr_enable() : …
%22 = riscv.li 127 : !riscv.reg<t0>
riscv.frep_outer %22, 0, 0 {
^0(%x : !riscv.freg<ft0>, %y : !riscv.freg<ft1>):
    %r0 = riscv.fadd.d %x, %y : !riscv.freg<ft2>
    riscv.frep_yield %r0 : !riscv.freg<ft2>
}
snrt.ssr_disable
```

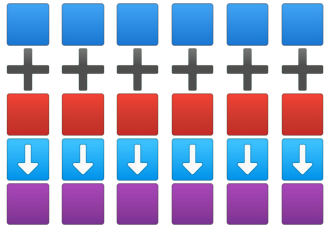
Emit assembly

```
call snrt_ssr_loop_2d
…
call snrt_ssr_write
…
call snrt_ssr_enable
riscv.li t0, 127
frep.o t0, 1, 0, 0
fadd.d ft2, ft0, ft1
call snrt_ssr_disable
```



Compiler backend design with MLIR

$Z_i = X_i + Y_i$



Preserve Semantic Information

Structured Register Allocation

Optimal Hardware Use