RVV: Variable Length, Variable Pain

Кущенко Андрей

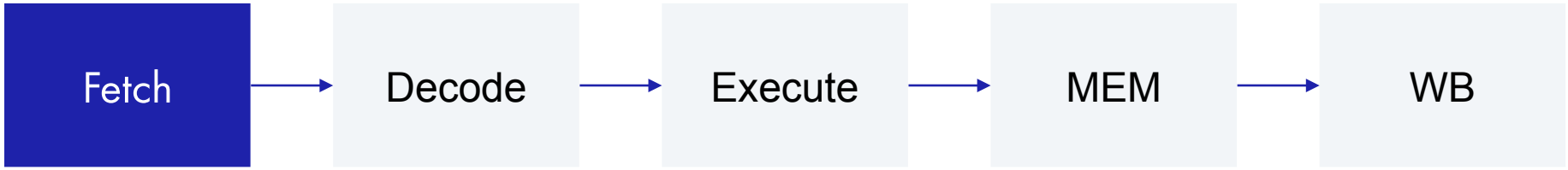
Базовое введение в конвейер CPU

Противоречивая функциональность RVV

Проблемы разработки и верификации реализации RVV

Выводы

Типовой пятистадийный конвейер RISC-V



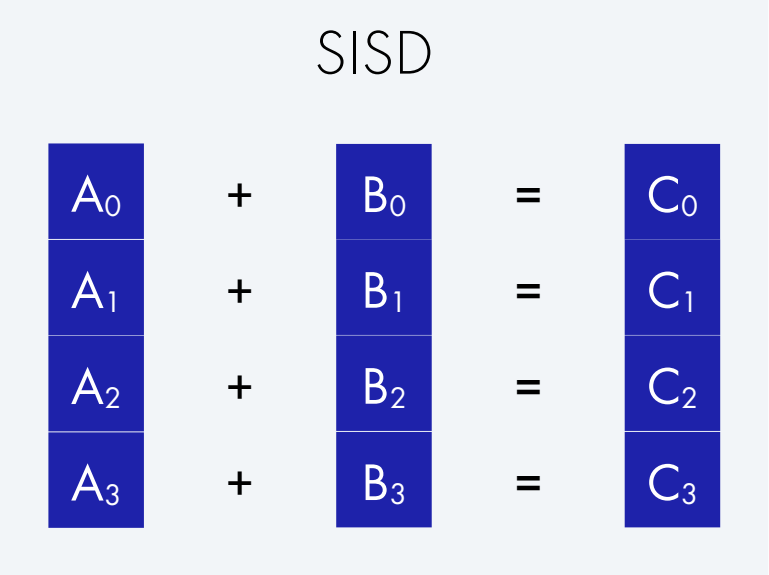
Типовой пятистадийный конвейер RISC-V

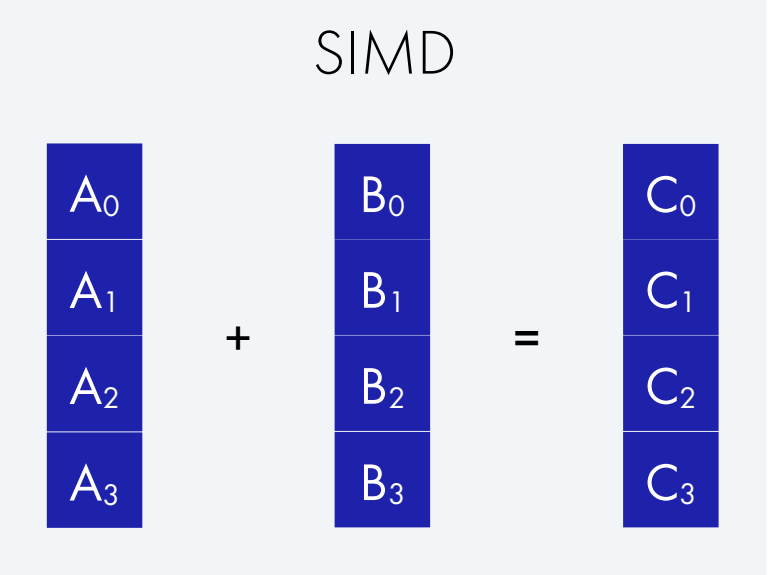
Типовой пятистадийный конвейер RISC-V

Типовой пятистадийный конвейер RISC-V

Типовой пятистадийный конвейер RISC-V

SIMD


SISD
A0 + B0 = C0
A1 + B1 = C1
A2 + B2 = C2
A3 + B3 = C3


SIMD
A0 B0 C0
A1 + B1 = C1
A2 B2 C2
A3 B3 C3

Базовое введение в конвейер CPU

Противоречивая функциональность RVV

Проблемы разработки и верификации реализации RVV

Выводы

RVV: variable length, variable pain

- Программист не знает (не обязан знать) длину регистра (VLEN = 128, 256, 1024...).
- CPU самостоятельно определяет, сколько элементов может обработать за раз.
- Маски хранятся непосредственно в векторных регистрах, а не отдельно.
- Нет необходимости в «эпилоге» при размере вектора не кратному размеру регистра.
- Поддержка режима дробных размеров векторных регистров.
- Неочевидные проблемы с производительностью
- Недетерменизм поведения некоторых инструкций и режимов работы
- Сложность в реализации и верификации

Динамическое изменение размера векторных регистров и элементов

LMUL (Lane Multiplier) inside {1/8, 1/4, 1/2, 1, 2, 4, 8}
SEW (Selected Element Width) inside {8, 16, 32, 64}

LMUL=1/2

v0	1	0
v1	3	2
v2	5	4
v3	7	6

LMUL=1

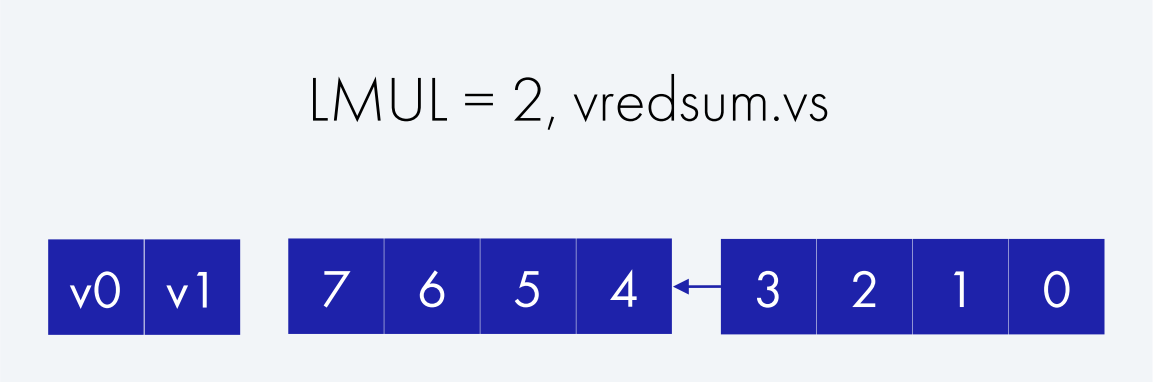
v0	3	2	1	0
v1	7	6	5	4

LMUL=2

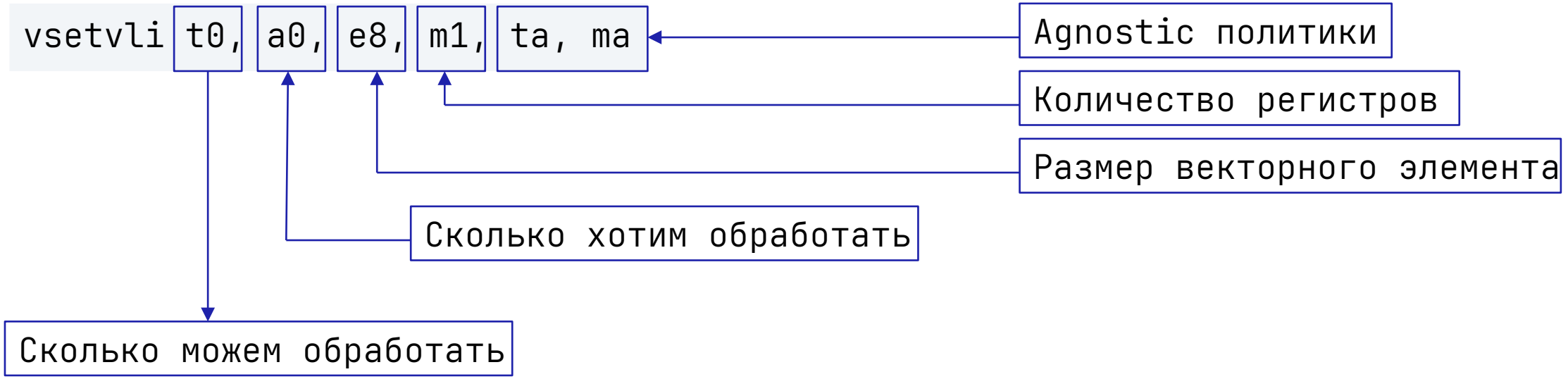
v0	v1	7	6	5	4	3	2	1	0

Падение производительности в случае зависимостей

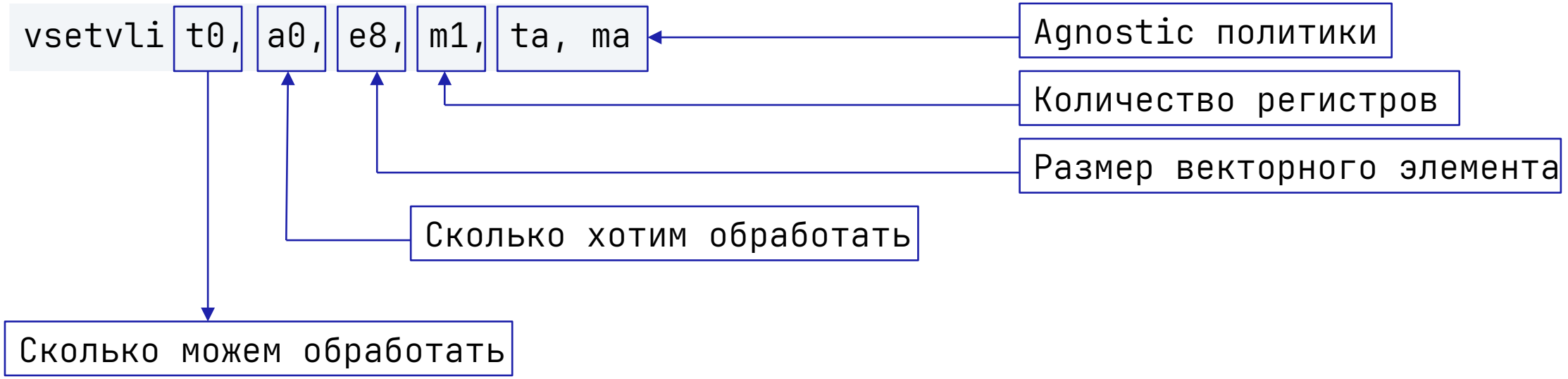
- Явная зависимость между результатом обработки первого и второго регистров
- Возможные решения:
  - “пузыри” в конвейере
  - Сложная логика байпасов
  - Аппаратура для максимальной конфигурации LMUL=8

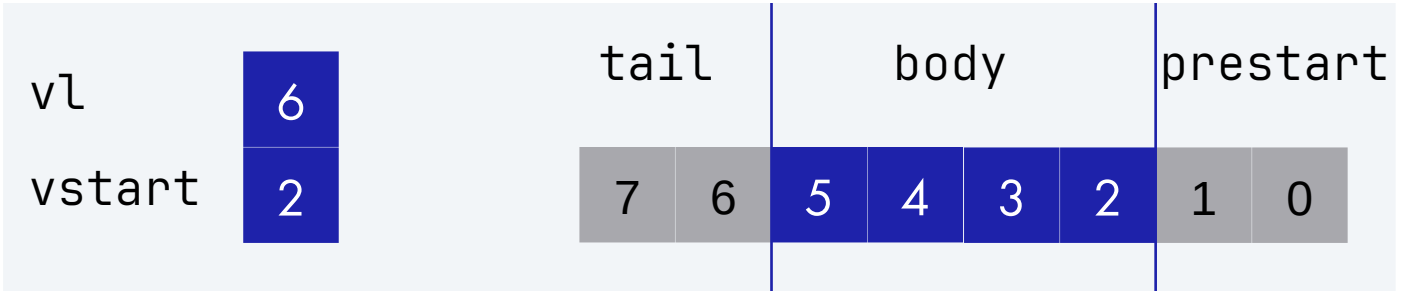


vsetvli


vsetvli t0, a0, e8, m1, ta, ma
Agnostic политики
Количество регистров
Размер векторного элемента
Сколько хотим обработать
Сколько можем обработать

vsetvli


vsetvli t0, a0, e8, m1, ta, ma
Agnostic политики
Количество регистров
Размер векторного элемента
Сколько хотим обработать
Сколько можем обработать


vl
6
vstart
2
tail
body
prestart
7
6
5
4
3
2
1
0

vsetvli и скалярный цикл

```
loop:
    vsetvli t0, a0, e32, m1, ta, ma # Сколько обработаем?
    vle32.v v0, (a1)                # Загрузили
    vadd.vv v2, v0, v1              # Сложили
    vse32.v v2, (a2)                # Сохранили
    sub a0, a0, t0                  # Уменьшили счетчик
    add a1, a1, t0                  # Сдвинули указатель (x4)
    bnez a0, loop
```

vsetvli и скалярный цикл

```
loop:
    vsetvli t0, a0, e32, m1, ta, ma # Сколько обработаем?
    vle32.v v0, (a1)                # Загрузили
    vadd.vv v2, v0, v1              # Сложили
    vse32.v v2, (a2)                # Сохранили
    sub a0, a0, t0                  # Уменьшили счетчик
    add a1, a1, t0                  # Сдвинули указатель (x4)
    bnez a0, loop
```

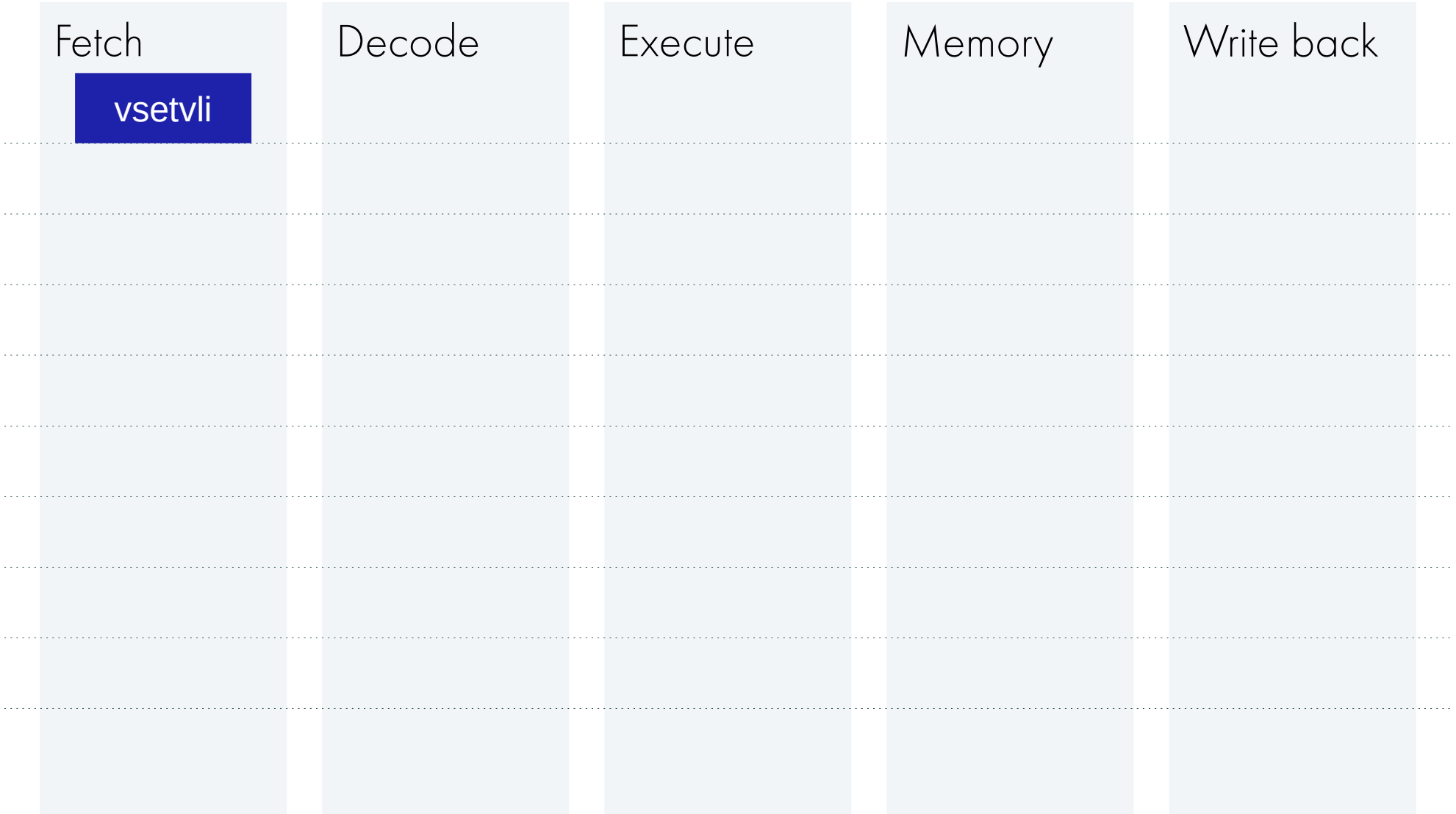
vsetvli и скалярный цикл

```
loop:
    vsetvli t0, a0, e32, m1, ta, ma # Сколько обработаем?
    vle32.v v0, (a1)                # Загрузили
    vadd.vv v2, v0, v1              # Сложили
    vse32.v v2, (a2)                # Сохранили
    sub a0, a0, t0                  # Уменьшили счетчик
    add a1, a1, t0                  # Сдвинули указатель (x4)
    bnez a0, loop
```

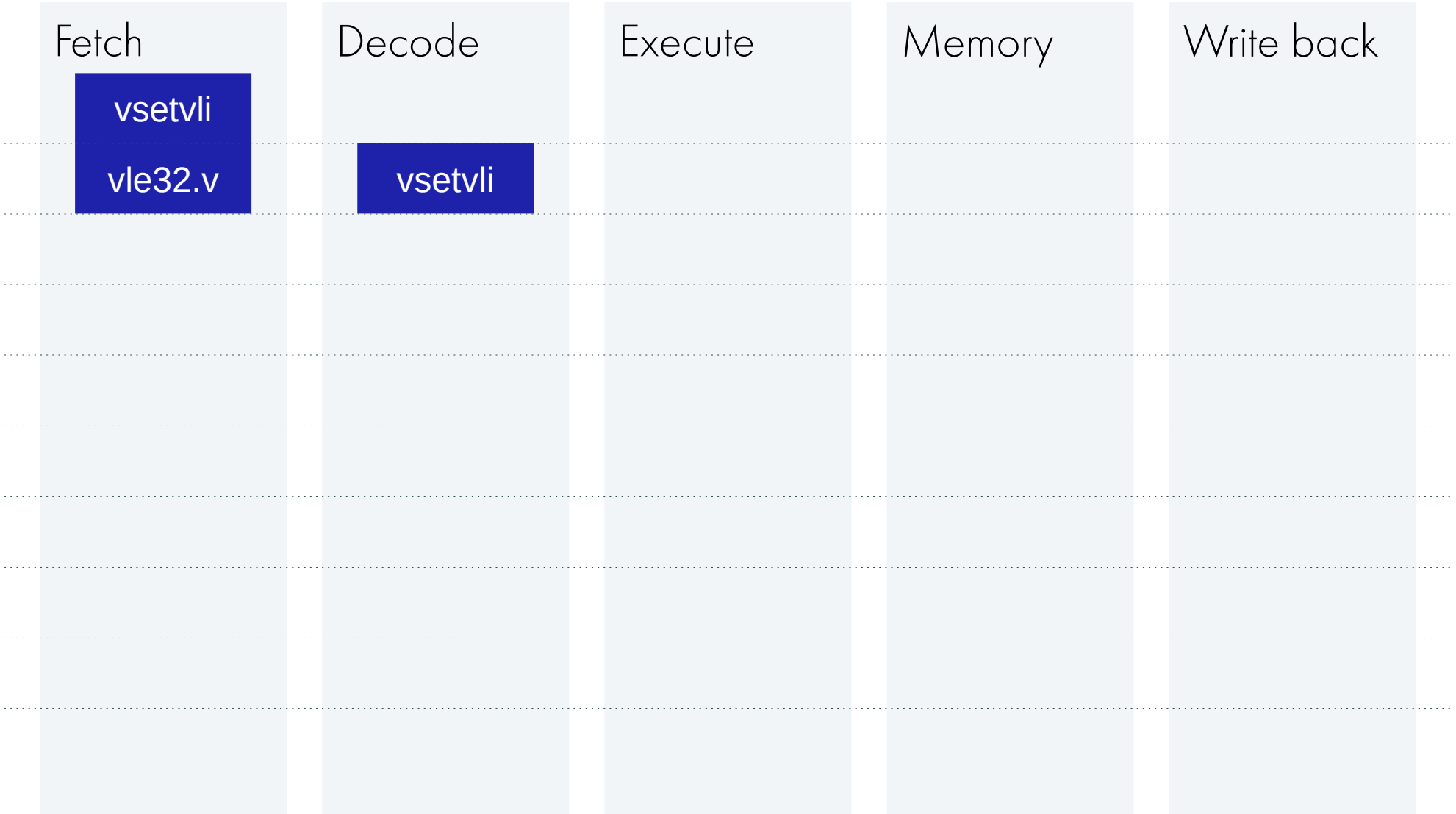
vsetvli и скалярный цикл

```
loop:
    vsetvli t0, a0, e32, m1, ta, ma # Сколько обработаем?
    vle32.v v0, (a1)                # Загрузили
    vadd.vv v2, v0, v1              # Сложили
    vse32.v v2, (a2)                # Сохранили
    sub a0, a0, t0                  # Уменьшили счетчик
    add a1, a1, t0                  # Сдвинули указатель (x4)
    bnez a0, loop
```

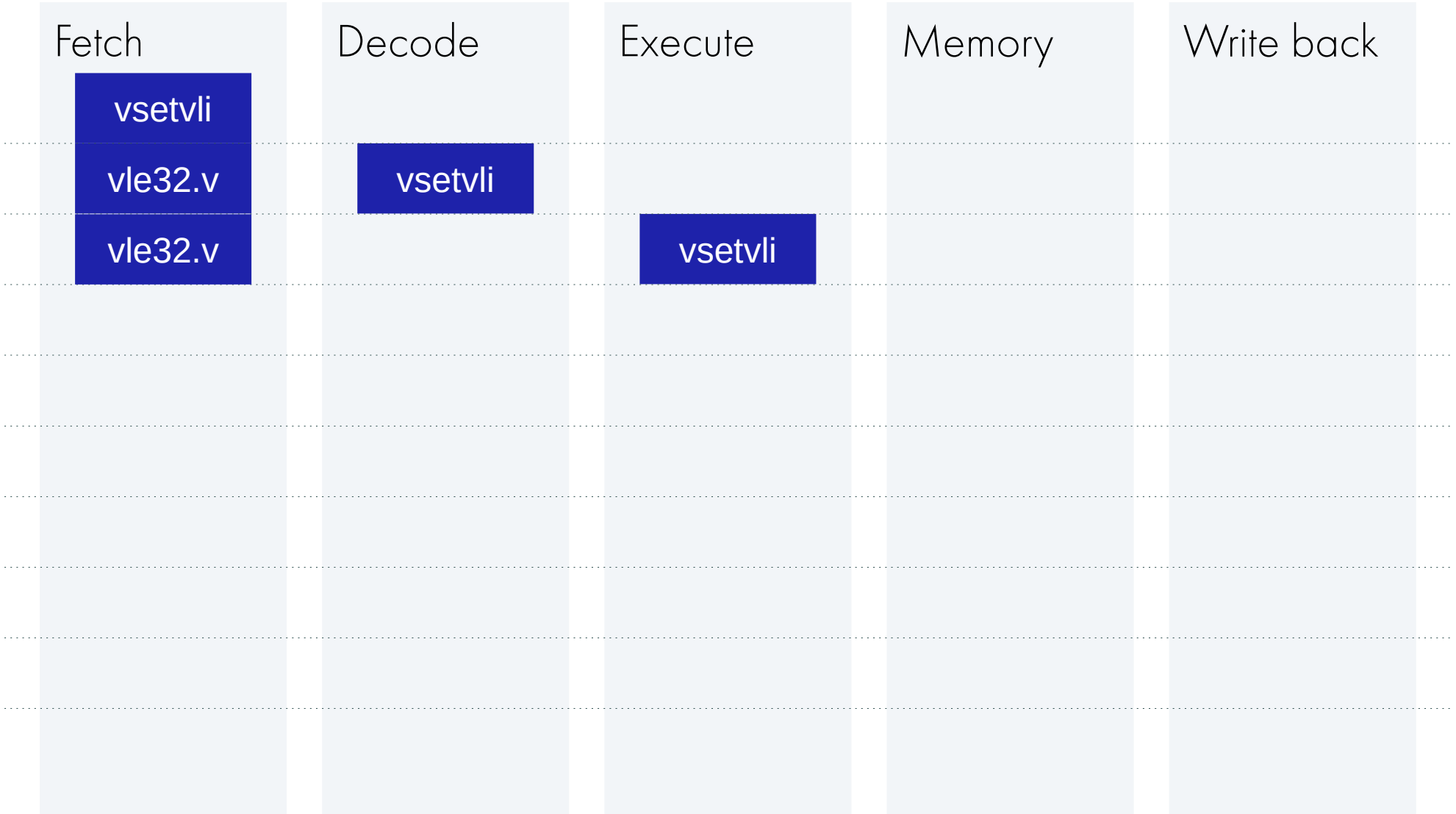
Неявная зависимость от состояния CSR



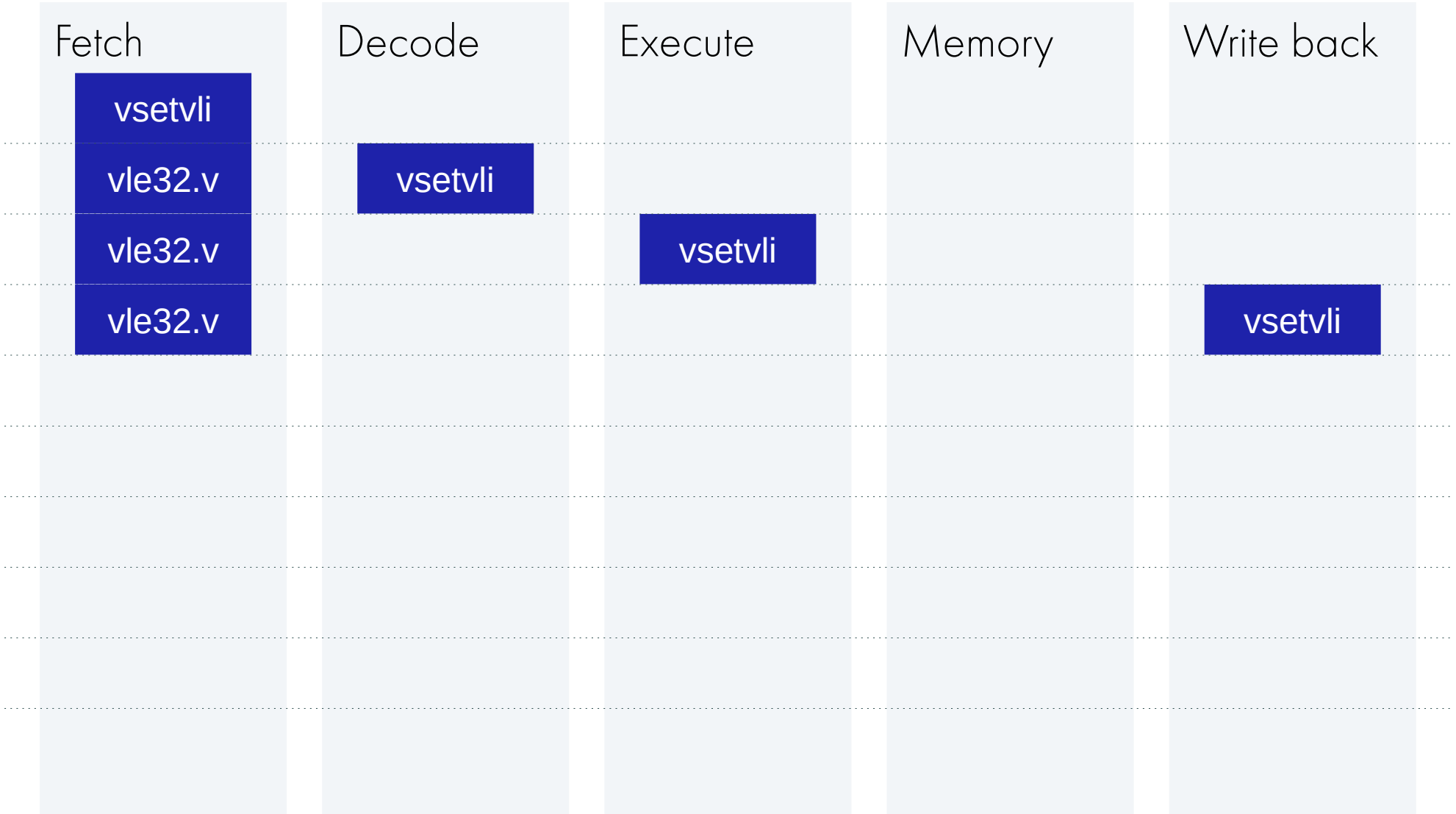
Неявная зависимость от состояния CSR



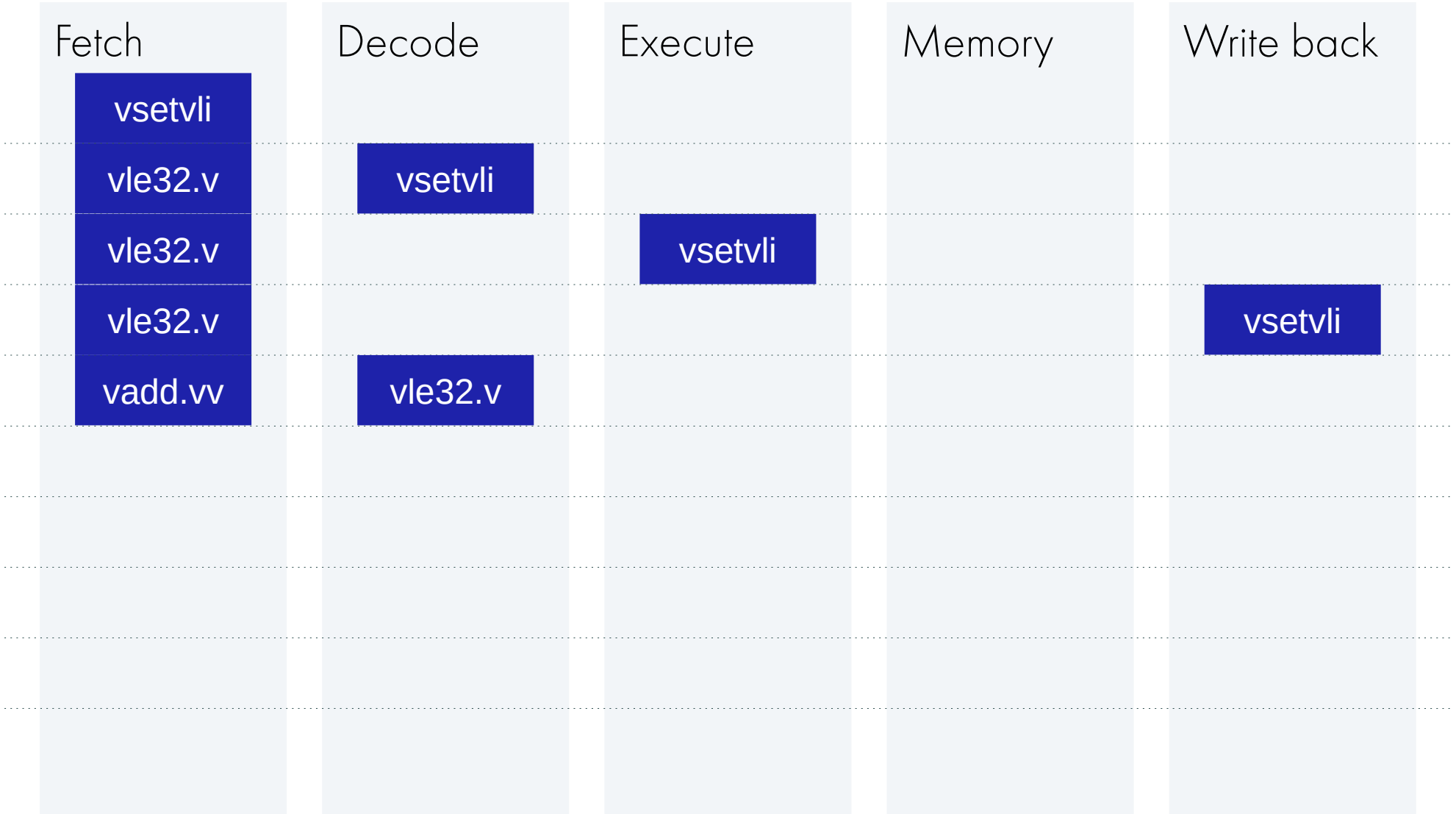
Неявная зависимость от состояния CSR



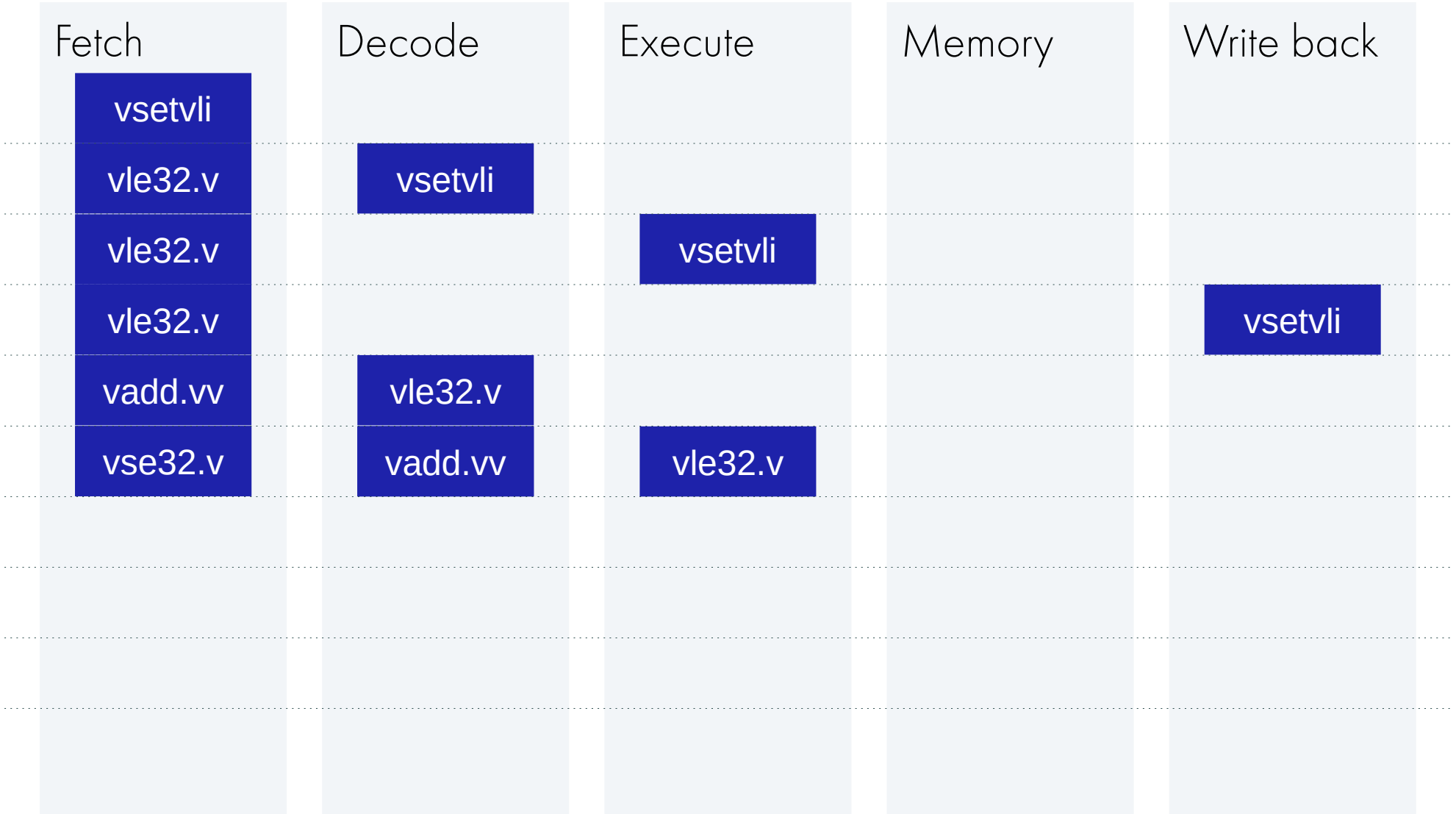
Неявная зависимость от состояния CSR



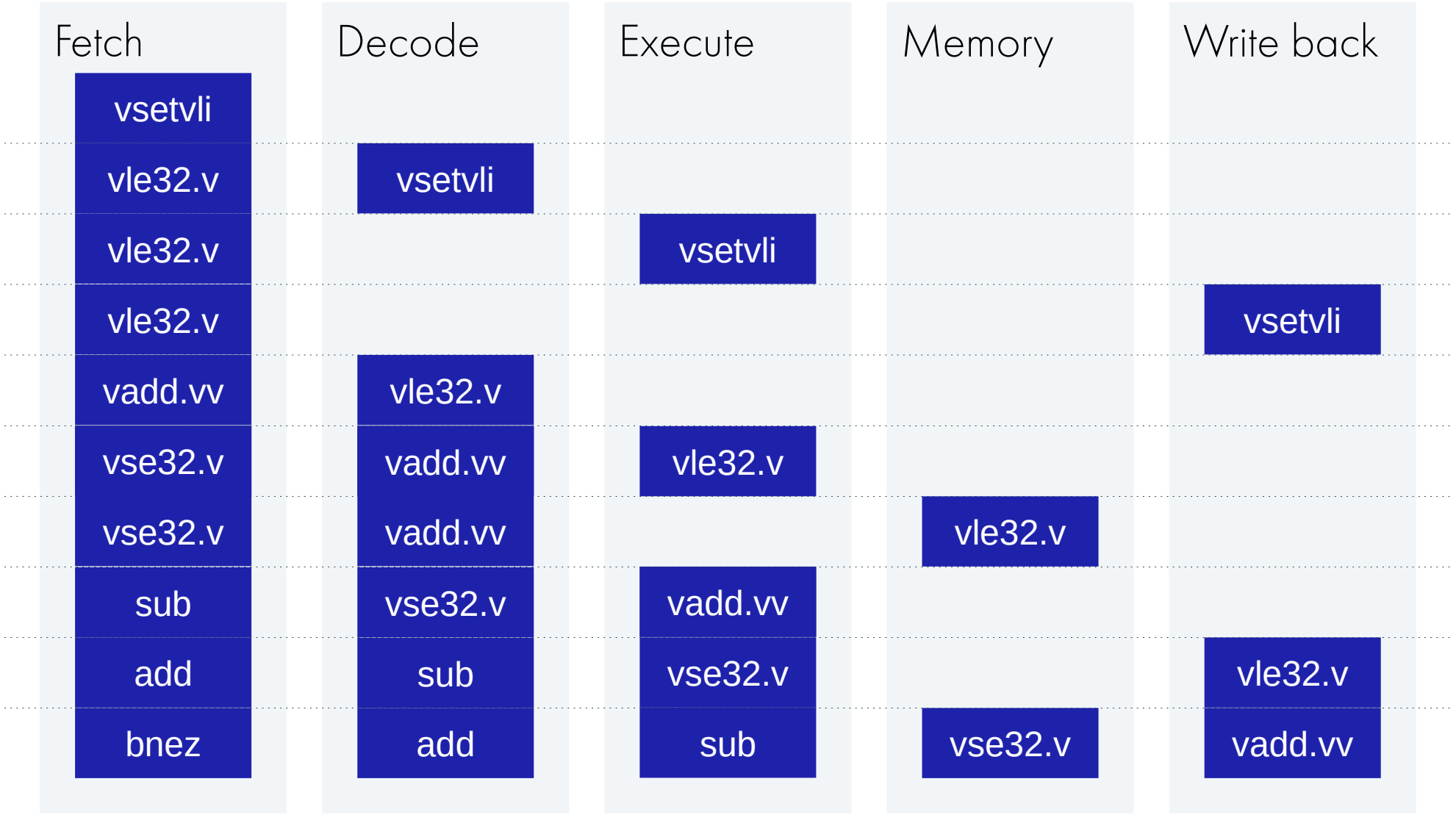
Неявная зависимость от состояния CSR



Неявная зависимость от состояния CSR



Неявная зависимость от состояния CSR



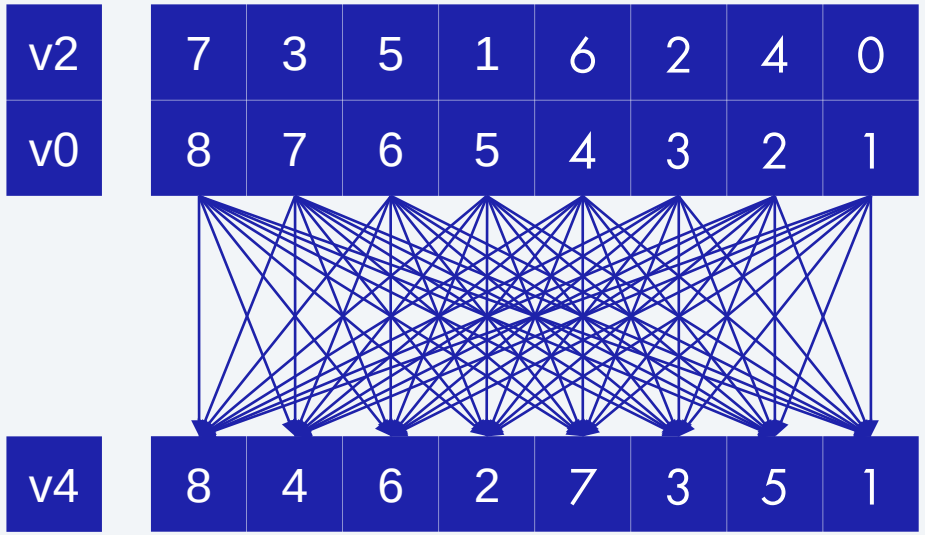
Полный кроссбар через vrgather

```
# Таблица перестановки (8 элементов)
li t0, 0x0703050106020400
vmv.v.x v2, t0
vrgather.vv v4, v0, v2
```

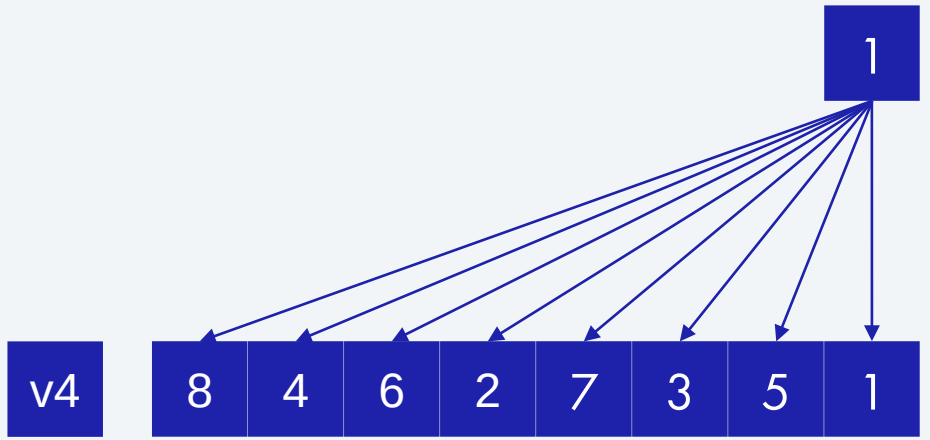
Полный кроссбар через vrgather

```
# Таблица перестановки (8 элементов)
li t0, 0x0703050106020400
vmv.v.x v2, t0
vrgather.vv v4, v0, v2
```

Полный кроссбар всех элементов

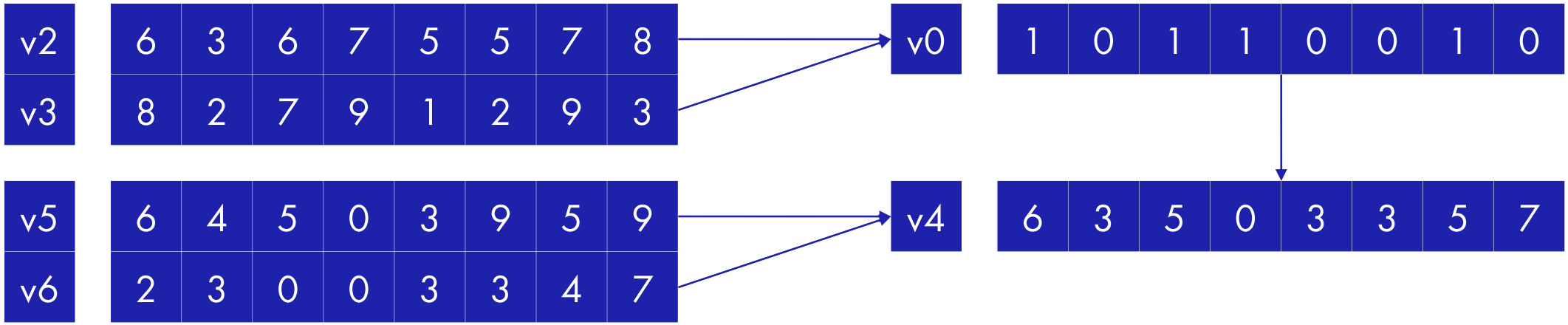


Кроссбар для одного элемента



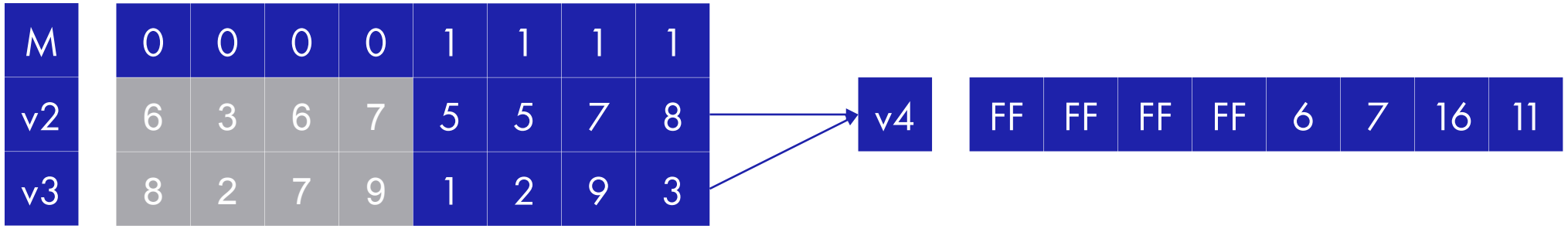
Маскирование

```
# if (a[i] < b[i]) z[i] = c[i] else z[i] = d[i]
vmslt.vv v0, v2, v3          # v0.mask[i] = (v2[i] < v3[i])
vmerge.vvm v4, v6, v5, v0    # v4[i] = v0.mask ? v5[i] : v6[i]
```



Неактивные элементы не ускоряют!

Data path width = 64
SEW = 32
VLEN = 256



Сегментные операции работы с памятью

```
struct Pixel { uint8_t r, g, b; };

vlseg3e8.v v8, (a0)    # v8=R, v9=G, v10=B
vadd.vi v8, v8, 10     # Добавили яркость красному
vsseg3e8.v v8, (a0)    # Сохранили обратно
```

Сегментные операции работы с памятью

```
struct Pixel { uint8_t r, g, b; };

vlseg3e8.v v8, (a0)   # v8=R, v9=G, v10=B
vadd.vi v8, v8, 10    # Добавили яркость красному
vsseg3e8.v v8, (a0)   # Сохранили обратно
```

Сегментные операции работы с памятью

```
struct Pixel { uint8_t r, g, b; };

vlseg3e8.v v8, (a0)   # v8=R, v9=G, v10=B
vadd.vi v8, v8, 10    # Добавили яркость красному
vsseg3e8.v v8, (a0)   # Сохранили обратно
```

Сегментные операции работы с памятью

```
struct Pixel { uint8_t r, g, b; };

vlseg3e8.v v8, (a0)   # v8=R, v9=G, v10=B
vadd.vi v8, v8, 10    # Добавили яркость красному
vsseg3e8.v v8, (a0)   # Сохранили обратно
```

v1	8	2	7	9	1	2	9	3
v2	6	3	6	7	5	5	7	8
v3	5	3	8	9	7	0	9	4

3	8	4	9	7	9	2	5	0

v1	8	2	7	9	1	2	9	3
v2	6	3	6	7	5	5	7	8
v3	5	3	8	9	7	0	9	4

3	8	4	9	7	9	2	5	0

Операции работы с памятью со страйдом

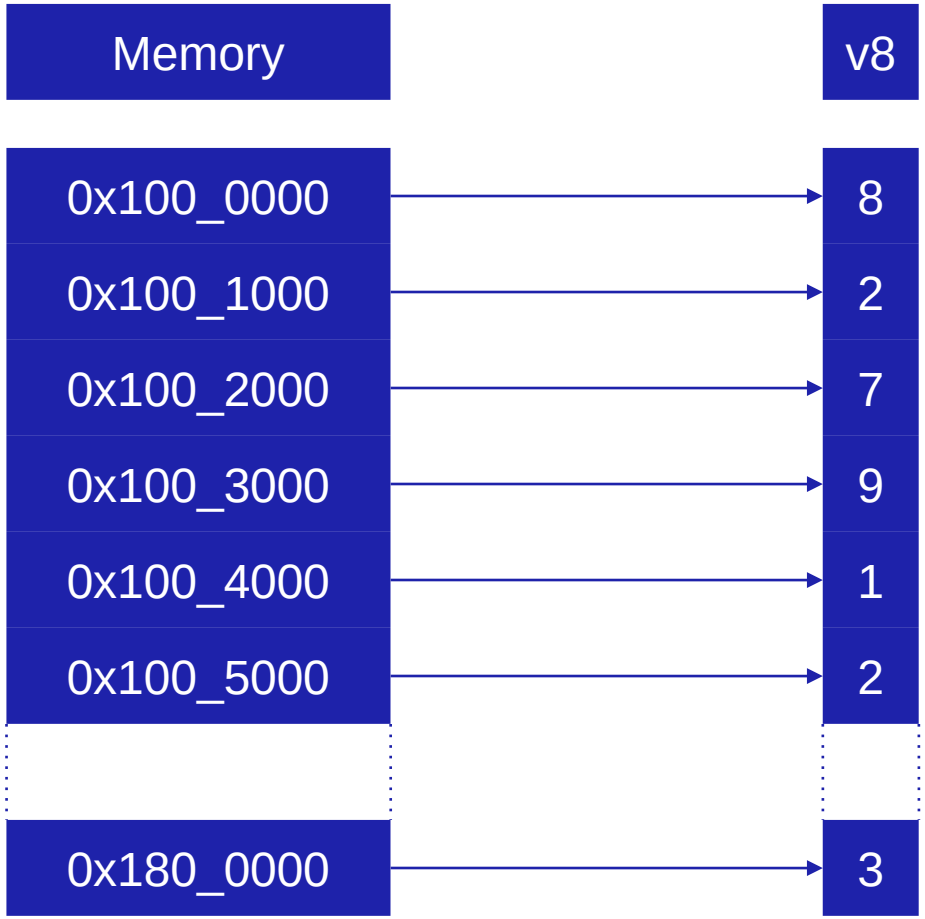
```
# VLEN = 128
# LMUL = 8
# VL = 128
# x11(stride)  = 0x1000
# X10(address) = 0x100_0000
vlse8.v v8 x10, x11
```

Получаем 128 обращений в память

Операции работы с памятью со страйдом

```
# VLEN = 128
# LMUL = 8
# VL = 128
# x11(stride)  = 0x1000
# X10(address) = 0x100_0000
vlse8.v v8 x10, x11
```

Получаем 128 обращений в память



Fault only first loads

```
# strlen()
loop:
  vsetvli a1, x0, e8, m8, ta, ma
  vle8ff.v v8, (a3)   # Load bytes
  csrr a1, vl         # Get bytes read
  vmseq.vi v0, v8, 0  # Set v0[i] where v8[i] = 0
  vfirst.m a2, v0     # Find first set bit
  add a3, a3, a1      # Bump pointer
  bltz a2, loop       # Not found?
```

Fault only first loads

```
# strlen()
loop:
  vsetvli a1, x0, e8, m8, ta, ma
  vle8ff.v v8, (a3)   # Load bytes
  csrr a1, vl         # Get bytes read
  vmseq.vi v0, v8, 0  # Set v0[i] where v8[i] = 0
  vfirst.m a2, v0     # Find first set bit
  add a3, a3, a1      # Bump pointer
  bltz a2, loop       # Not found?
```

Fault only first loads

```
# strlen()
loop:
  vsetvli a1, x0, e8, m8, ta, ma
  vle8ff.v v8, (a3)   # Load bytes
  csrr a1, vl         # Get bytes read
  vmseq.vi v0, v8, 0  # Set v0[i] where v8[i] = 0
  vfirst.m a2, v0     # Find first set bit
  add a3, a3, a1      # Bump pointer
  bltz a2, loop       # Not found?
```

Fault only first loads

```
# strlen()
loop:
  vsetvli a1, x0, e8, m8, ta, ma
  vle8ff.v v8, (a3)   # Load bytes
  csrr a1, vl         # Get bytes read
  vmseq.vi v0, v8, 0  # Set v0[i] where v8[i] = 0
  vfirst.m a2, v0     # Find first set bit
  add a3, a3, a1      # Bump pointer
  bltz a2, loop       # Not found?
```

Fault only first loads

```
# strlen()
loop:
  vsetvli a1, x0, e8, m8, ta, ma
  vle8ff.v v8, (a3)   # Load bytes
  csrr a1, vl         # Get bytes read
  vmseq.vi v0, v8, 0  # Set v0[i] where v8[i] = 0
  vfirst.m a2, v0     # Find first set bit
  add a3, a3, a1      # Bump pointer
  bltz a2, loop       # Not found?
```

Fault only first loads

```
# strlen()
loop:
  vsetvli a1, x0, e8, m8, ta, ma
  vle8ff.v v8, (a3)   # Load bytes
  csrr a1, vl         # Get bytes read
  vmseq.vi v0, v8, 0  # Set v0[i] where v8[i] = 0
  vfirst.m a2, v0     # Find first set bit
  add a3, a3, a1      # Bump pointer
  bltz a2, loop       # Not found?
```

Базовое введение в конвейер CPU

Противоречивая функциональность RVV

Проблемы разработки и верификации реализации RVV

Выводы

Проблемы разработки

- Баланс между производительностью и площадью
- Много краевых ситуаций
- Необходимость параметризации
- Динамическое изменение конфигурации
- Инструкции могут иметь разное количество данных на входе и на выходе
- Маленький кеш первого уровня
- Конфликты конвейера выходят на новый уровень (несколько регистров сразу)

Проблемы верификации

- Сотни инструкций
- Десятки режимов работы
- Множество ситуаций в рамках одной инструкции (VGATHER)
- Долгое время исполнения тестов из-за множества ситуаций
- Недетерменизм в исполнении некоторых инструкций

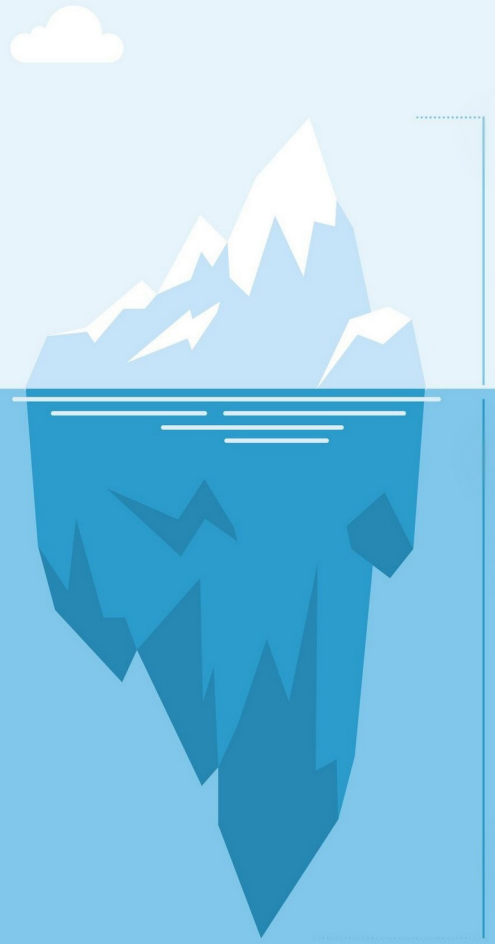
Базовое введение в конвейер CPU

Противоречивая функциональность RVV

Проблемы разработки и верификации реализации RVV

Выводы

Эффект "Айсберга"



- Элегантные циклы
- Мощные операции над структурами
- Нет «головной боли» с шириной регистра

- Динамическое переключение ширины (VL)
- Группировка регистров и борьба с перекрытиями
- Недетерминизм ta/ma
- Рестарт инструкций с произвольного индекса
- Работа с памятью
- Неочевидные проблемы с производительностью