



Экосистема RISC-V

МФТИ
Весна 2024

Пару слов про docker



Docker – система упаковывания приложения вместе с окружением в «контейнер», а также управления такими контейнерами.

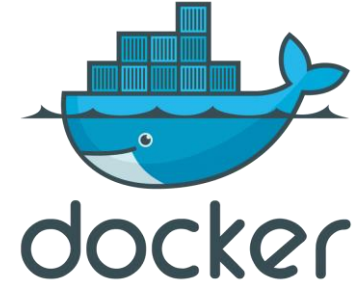
GameBoy:

- Просто вставь картридж и играй
- Хочешь поделиться с другом игрой – просто передай картридж



Docker позволяет передавать контейнер, в котором приложение будет сразу работать

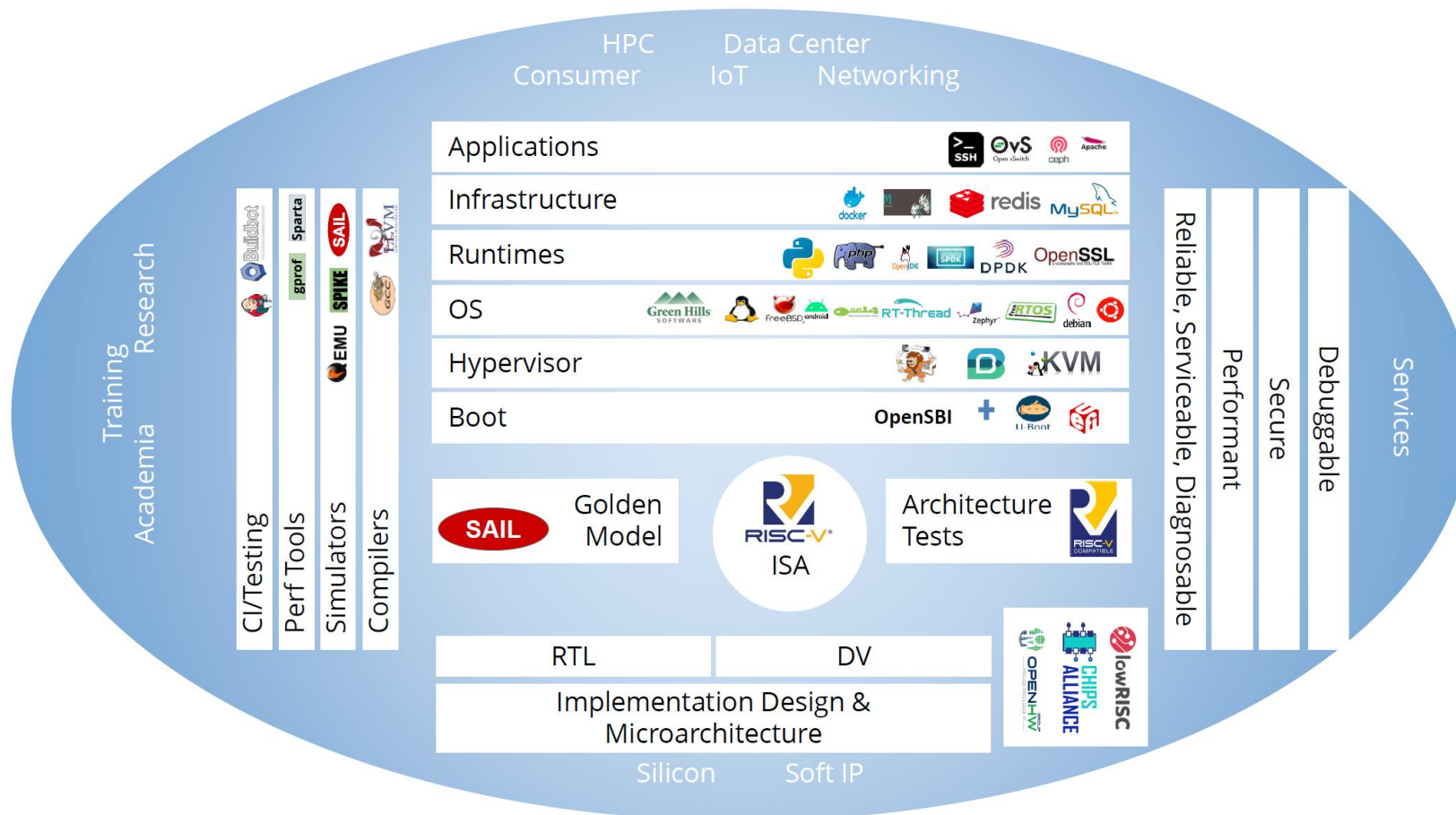
Пару слов про docker



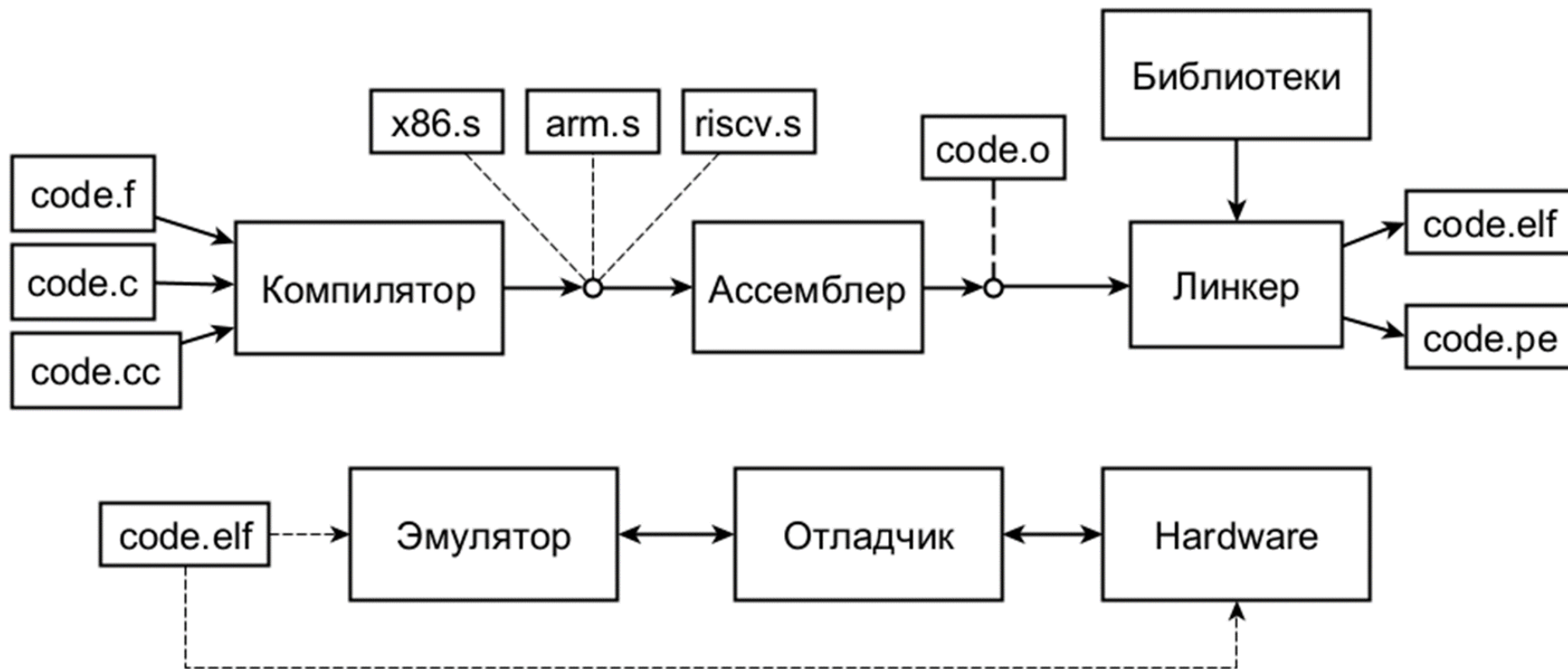
Окружение приложения:

- Бинарные зависимости (библиотеки, другие приложения)
- Структура файловой системы
- Конфигурация системы
- Переменные окружения

Другая сторона айсберга



Тулчейн – общий вид



Кросс-компиляция

Основная терминология:

- **build** – машина, где происходит сборка компилятора.
- **host** – машина, где происходит сборка приложения.
- **target** – машина, где происходит запуск приложения.

Обычно для RISCV: build = host, host ≠ target

```
$ riscv64-linux-gnu-gcc-12 -march=rv64 file.c -O2 -S
```

Если host = target, то это нативная компиляция

Имя компилятора на хосте включает в себя target triple

```
<arch><sub>-<vendor>-<sys>-<env>
```

В рассматриваемом случае:


```
arch = riscv, sub = 64, vendor empty, sys = linux, env = gnu
```

Hello, RISC-V!

```
$ cat hello.c
#include <stdio.h>
int main() { printf("Hello, world!\n"); }
```

```
$ uname -m
x86_64
```

```
$ riscv64-linux-gnu-gcc hello.c -static -o hello.x
```

```
$  hello.x
Hello, world!
```

Hello, RISC-V!

```
$ cat hello.c
#include <stdio.h>
int main() { printf("Hello, world!\n"); }
```

```
$ uname -m
x86_64
```

```
$ riscv64-linux-gnu-gcc hello.c -static -o hello.x
```

```
$ qemu-riscv64 hello.x
Hello, world!
```


Концепция сжатых инструкций

```
// -march=rv64i
000000000000000000 <elt>:
  0:    00259593 slli    a1, a1, 0x2
  4:    00a58533 add     a0, a1, a0
  8:    00852503 lw      a0, 8(a0)
```

```
// -march=rv64ic
000000000000000002 <elt>:
  2:    058a      c.slli  a1, a1, 0x2
  4:    952e      c.add   a0, a0, a1
  6:    4508      c.lw    a0, 8(a0)
```

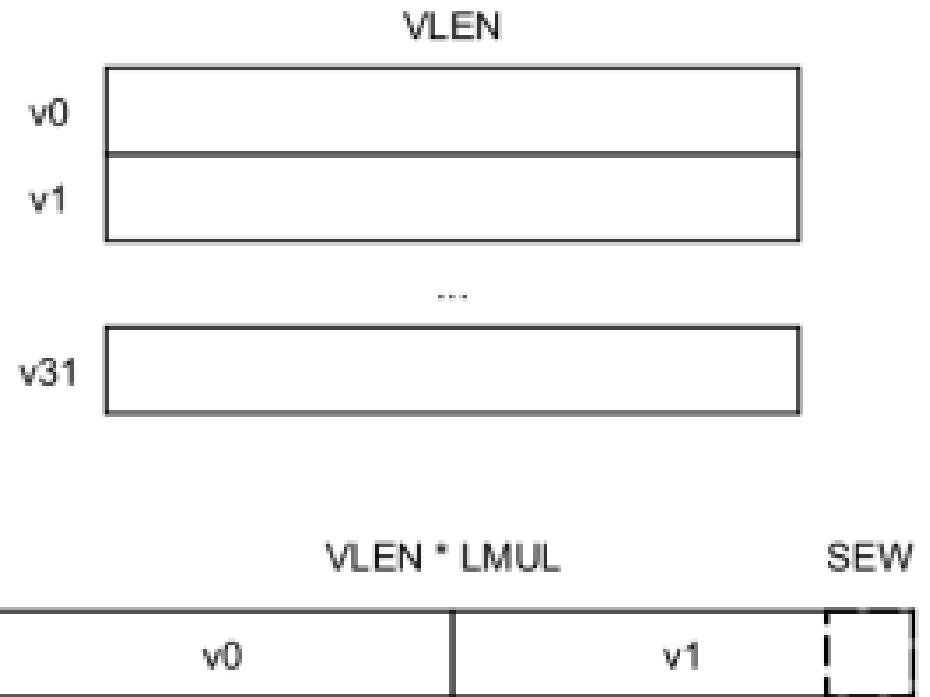
Векторизация как возможность

```
void memcpy(char *dst, const char *src, size_t n) {  
    for (int i = 0; i < n; ++i)  
        dst[i] = src[i];  
}
```

- Мотивация: мы стремимся к повышению производительности «горячих» циклов.
- Возможность: наличие векторных регистров и векторного юнита.
- Идея: обрабатывать за одну итерацию цикла более одного элемента данных, используя векторные операции.

Основная идея для RVV

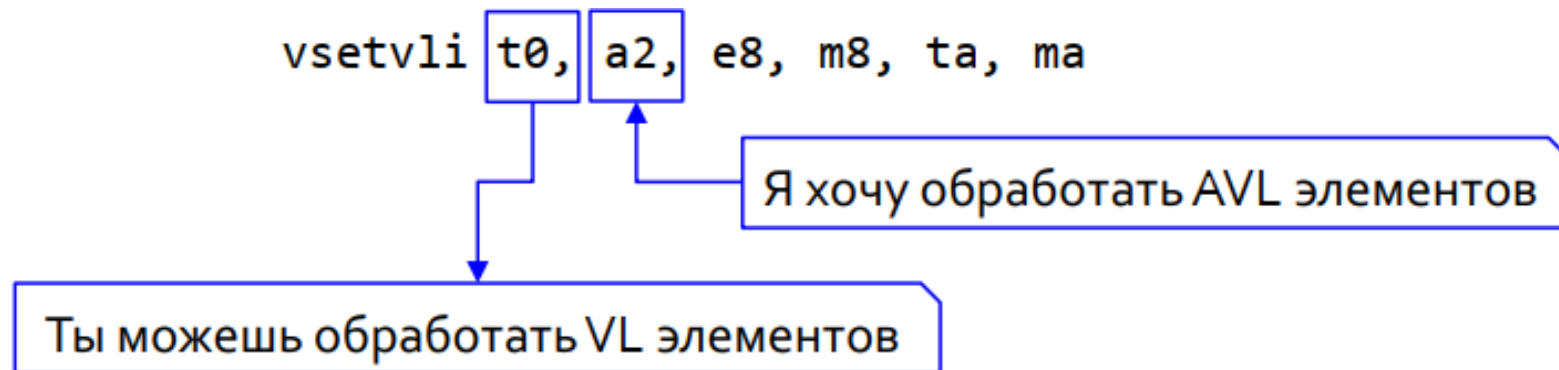
- 32 векторных регистра **v0 ... v31**
- Каждый какой-то длины **VLEN**
- Эти регистры можно комбинировать группами, динамически изменяя размер группы **LMUL**.
- Группа регистров это вектор у которого также динамически можно задавать **SEW** (single element width).
- Для установки режима используются специальные инструкции (см. далее)



Идея инструкций vset*, AVL и VL

vsetvli rd, rs1, vtype1 # AVL = x[rs1], x[rd] = vl
vsetivli rd, uimm, vtype1 # AVL = uimm, x[rd] = vl
vsetvl rd, rs1, rs2 # AVL = x[rs1], VT = x[rs2], x[rd] = vl

vsetvl rd, x0, ... # AVL = VLMAX



memcpy

```
.global memcpy

memcpy:
    mv a3, a0                                # Copy destination
loop:
    vsetvli t0, a2, e8, m8, ta, ma          # Vectors of 8 * VLEN
    vle8.v v0, (a1)                          # Load bytes
    add a1, a1, t0                           # Bump pointer
    sub a2, a2, t0                           # Decrement count
    vse8.v v0, (a3)                          # Store bytes
    add a3, a3, t0                           # Bump pointer
    bnez a2, loop                            # Any more?
    ret                                      # Return
```

Работа с платой

```
$ cat hello.c
#include <stdio.h>
int main() { printf("Hello, world!\n"); }
```

```
$ riscv64-linux-gnu-gcc hello.c -static -o hello.x
```

```
$ scp hello.x user@board-ip:~
```

```
$ ssh user@board-ip
```

```
$ uname -m
riscv
```

```
$ ~/hello.x
Hello, world!
```

Собираем с расширениями

```
$ riscv64-linux-gnu-gcc file.c -march=rv64i  
-mabi=lp64 -static -O2 -o app
```

```
$ riscv64-linux-gnu-gcc file.c -march=rv64im  
-mabi=lp64 -static -O2 -o app
```

```
$ riscv64-linux-gnu-gcc file.c -march=rv64imf  
-mabi=lp64f -static -O2 -o app
```

```
$ riscv64-linux-gnu-gcc file.c -march=rv64imfd  
-mabi=lp64d -static -O2 -o app
```

```
$ riscv64-linux-gnu-gcc file.c -march=rv64gc  
-mabi=lp64d -static -O2 -o app
```

Пришло время отладки

```
$ riscv64-linux-gnu-gcc buggy-sort.c -static -o  
buggy-sort
```

```
$ ./buggy-sort
```



Кросс-отладка

По аналогии с кросс-компиляцией:

- **build** – машина, где происходит сборка отладчика
- **host** – машина, где происходит запуск фронтенда отладчика
- **target** – машина, где происходит запуск отлаживаемого приложения

Host:

```
$ riscv64-linux-gnu-gdb buggy-sort  
> remote <board-ip>:<port>
```

Target:

```
$ gdbserver <port> <program> [ args ... ]
```

To be continued ...

На следующем занятии как сделать свой RISC-V процессор и

- Как ведется разработка программного обеспечения для процессора, которого нет
- Когда решают, что пора оформлять заказ на производство
- Почему первый блин всегда комом
- Как верифицируют процессоры