

# Языки описания схем

mk.cs.msu.ru → Лекционные курсы → Языки описания схем

## Блок 29

Хороший код и плохой код  
Данные и управление  
Операционный и управляющий автоматы

Лектор:  
**Подымов Владислав Васильевич**  
E-mail:  
**valdus@yandex.ru**

ВМК МГУ, 2023/2024, осенний семестр

## Вступление: хороший код и плохой код

В программировании (например, на том же C/C++) есть свод писанных и неписанных правил, оценивающих высокоуровневую организацию (архитектуру) кода как **хорошую** или **плохую**

Будем условно называть такие правила **правилами (требованиями) стиля**

Проектирование схемы на языке высокого уровня абстракции (например, V) — это **тоже программирование**, только в особой парадигме, использующей:

- ▶ особый набор примитивных понятий
  - ▶ точки, сигналы, фронты, такты, ...
- ▶ особый набор команд
  - ▶ вентили, комбинационные операции, триггеры, ...
- ▶ особый вид композиции команд
  - ▶ соединение проводами

Поэтому можно предъявлять требования стиля и к коду схемы

# Вступление: хороший код и плохой код

Правила стиля можно (огрублённо) поделить на несколько категорий:

- ▶ **Универсальные:** без изменений дословно применяющиеся ко всем парадигмам и языкам
- ▶ **Полууниверсальные:** имеющие для всех или многих парадигм одинаковую идею, но разные технические детали, адаптирующиеся под конкретные парадигмы
- ▶ **Специальные:** имеющие смысл только для одной или нескольких (немногих) парадигм

# Вступление: хороший код и плохой код

Примеры универсальных правил стиля (применительно к  $\mathcal{V}$ ):

- ▶ Разрабатывать схему, мысля не в терминах парадигмы языка (схемной), — это **плохо**
- ▶ Концентрировать в одном месте столько деталей, что их трудно удержать в уме, — это **плохо**
- ▶ Обобщать и инкапсулировать — это **хорошо**, если восприятие и использование кода упрощается, и **плохо**, если нет
- ▶ «Изобретать велосипед» — это **плохо**, а использовать готовое (готовые модули  $\mathcal{V}$ , архитектурные решения и т.п.) — это **хорошо**
  - ▶ Только, увы, для схем готовое нередко оказывается недоступным из коммерческих соображений, так что «велосипеды» нередки

# Вступление: хороший код и плохой код

## Примеры полууниверсальных правил

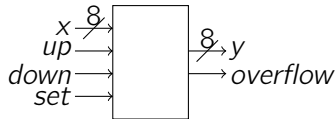
- ▶ Разбивать код на «обозримые» обособленные «логически целостные» подсхемы и группы подсхем, модули — это **хорошо**, а «смешивать» разнородные подсхемы в одном месте — это **плохо**
- ▶ Объединять точки, выполняющие общую задачу, в шины — **хорошо**, если восприятие кода упрощается и использование не очень усложняется, и **плохо** иначе
- ▶ Делать явное преобразования типов — **хорошо**, если эти преобразования неочевидны и могут привести к ошибке по невнимательности, и **плохо**, если очевидны

Можете вспомнить ещё много таких правил, опираясь на известные языки программирования (в том числе на **C/C++**), и применить их к  $\mathcal{U}$

А подробного разговора заслуживают **специальные правила стиля**

## Сквозной пример

Правила разработки **хорошего** кода будут обсуждаться на примере такой синхронной схемы с асинхронным сбросом (будем отсылать к нему как к **управляемому счётчику**):



$$y(1) = 0$$

$$overflow(1) = 0$$

$$y(t+1) = \begin{cases} x(t), & \text{если } set(t) = 1; \\ y(t) + 1, & \text{если } set(t) = 0 \text{ и } up(t) = 1; \\ y(t) - 1, & \text{если } set(t) = up(t) = 0 \text{ и } down(t) = 1 \\ y(t) & \text{иначе} \end{cases}$$

$$overflow(t+1) = 1 \Leftrightarrow$$

при переходе от  $y(t)$  к  $y(t+1)$  происходит арифметическое переполнение

# Данные и управление в схеме

Основное специальное правило стиля при разработке схем в целом и при использовании  $\mathcal{U}$  в частности, которое рекомендуется осознать и научиться умело использовать, — это

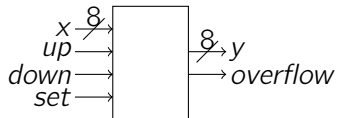
принцип разделения данных и управления

Согласно этому принципу, все точки схемы делятся на два класса: **информационные** и **управляющие**

Значениями в **управляющих** точках задаются «настройки» выполнения схемы: режимы работы, наступление исключительных случаев, выбор способов пересылки и преобразования сигналов, ...

Значения в **информационных** точках (**данные**) пассивно преобразуются схемой согласно текущим настройкам, задающимся управляющими значениями

# Данные и управление в схеме



**Например,** в управляемом счётчике:

- ▶ **данные** со входа *x* по необходимости сохраняются и преобразуются и в конечном итоге направляются в выход *y*
- ▶ значения на входах *up*, *down*, *set* **управляют** способом преобразования и перенаправления данных
- ▶ если значение на выходе *overflow* будет использоваться в другой схеме, то скорее всего оно будет **управлять** выполнением схемы, отмечая наступление исключительного случая



# Операционный и управляющий автоматы

В соответствии с делением точек на **управляющие** и **информационные** вся схема делится на две подсхемы:

**операционный автомат** и **управляющий автомат**<sup>1</sup>

**Операционный автомат** содержит все подсхемы, портами которых являются **информационные** точки, и «ничего лишнего»: реализацию

- ▶ всех способов преобразования **данных** схемой, но **не управления** выбором этих способов в конкретные моменты времени, и
- ▶ **управляющих** значений, задающихся непосредственно **данными** (**свойств данных**), для отправки в **управляющий автомат**

**Управляющий автомат** содержит всё остальное:

- ▶ подсхемы **управления** преобразованием **данных** и **управляющие** выходы схемы, нетривиально соотносящиеся с **данными**,
- ▶ и при этом **не содержит никаких данных**

---

<sup>1</sup> Термин «автомат» здесь расценивайте как историческое наследие, а не как обязательство к применению автоматов Мура

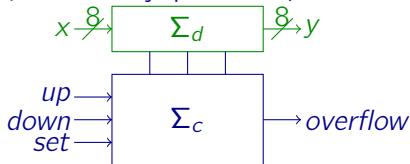
# Операционный и управляющий автоматы

В «западной» литературе и её переводе можно также встретить и другие названия этих подсистем:

- ▶ **Операционный автомат:** datapath, контур данных, data unit, блок данных
- ▶ **Управляющий автомат:** controlpath, контур управления, control unit, блок управления

**Например,** в управляемом счётчике

разделение на операционный и управляющий автоматы выглядит так:



# Операционный и управляющий автоматы

При использовании разделения **данных** и **управления** соответствующие автоматы нередко разрабатываются почти независимо:

- ▶ При разработке **операционного автомата** следует задумываться только о способах преобразования и свойствах **данных**
- ▶ При разработке **управляющего автомата** следует задумываться только о преобразовании **управляющих** значений

Такая независимая разработка позволяет:

- ▶ «Идеологически» упростить код
- ▶ Независимо применять разные подходы к разработке подсистем
- ▶ Избегать «глупых» ошибок, возникающих по недосмотру при смешении разнородных деталей реализации
- ▶ Улучшить поддерживаемость и упростить переиспользование кода