Языки описания схем

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

Блок 20

"Грамотная" разработка схем
Данные и управление
Операционный и управляющий автоматы

Лектор:
Подымов Владислав Васильевич

E-mail:
valdus@yandex.ru

ВМК МГУ, 2019, осенний семестр

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:

- Писать код языка одной парадигмы,
  придерживаясь понятий другой парадигмы, — плохо
- Писать модуль/функцию/метод/класс/... так,
  что смысл всего непосредственного содержимого
  невозможно удержать в уме — плохо
- Реализовывать много разнородных понятий
  в одном файле/модуле/классе/... — плохо,
  а выделять нетривиальные “логически целостные” понятия
  и реализовывать их отдельно — хорошо

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:

- Обобщать и инкапсулировать — хорошо,
  если упрощение восприятия кода
  оказывается весомее обилия технических деталей,
  а иначе плохо

- “Изобретать велосипед”:
  не использовать известный готовый функционал,
  не использовать паттерны проектирования в типовых местах, ... — плохо,
  а использовать готовые реализации и концепции — хорошо

- ... ...

Хороший код и плохой код

Проектирование схемы — это, по большому счёту,
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)

Как следствие,

- правила “грамотного” программирования, применяющиеся
  для всех парадигм, применяются и при проектировании схем
- правила, видоизменяющиеся от парадигмы к парадигме,
  применяются и при проектировании схем,
  изменяясь подходящим образом
- существуют и правила, специфичные для проектирования схем
  и редко встречающиеся в других парадигмах

Хороший код и плохой код

Примеры общих правил:

- Разрабатывать схему, мысля не в схемных терминах
  и не в терминах выбранного языка разработки — плохо
- Концентрировать столько деталей в одном месте,
  что все их невозможно удержать в уме — плохо
- “Изобретать велосипед” — плохо
  (*но хороших готовых схем в открытом доступе намного меньше,
  чем готовых программ, так что нередко приходится*)
- Обобщать и инкапсулировать — хорошо, если <...>, а иначе плохо

Хороший код и плохой код

Примеры видоизменяющихся общих правил:

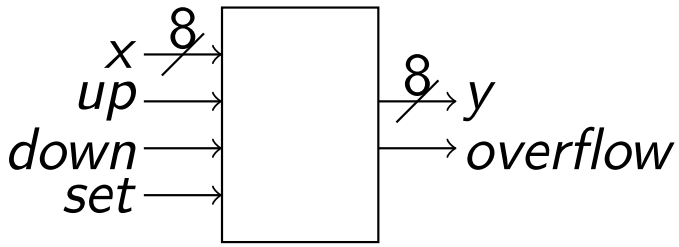
- Выделять “логически целостные” подсхемы и группы подсхем и реализовывать их отдельно (в отдельном модуле, файле, ...) — хорошо
- Идейно и визуально разделять группы подсхем, выполняющие разные задачи — хорошо
- Объединять провода, выполняющие единую задачу, в шины — хорошо, если это повышает наглядность и не добавляет слишком много технических деталей, а иначе плохо

А правила, специфичные для схемной парадигмы и редко встречающиеся в других парадигмах, можно обсудить подробнее

Хороший код и плохой код

Сквозной пример

Попробуем разработать такую синхронную схему Σ со сбросом:



$y(0) = 0$

$overflow(0) = 0$

Если $set(t) = 1$, то $y(t) = x(t)$

Если $set(t) = 0$ и $up(t) = 1$, то $y(t) = y(t-1) + 1$ (*по модулю* 2^8)

Если $set(t) = up(t) = 0$ и $down(t) = 1$, то $y(t) = y(t-1) - 1$

Иначе $y(t) = y(t-1)$

Если при переходе от $y(t-1)$ к $y(t)$ произошло арифметическое переполнение, то $overflow(t) = 1$, а иначе $overflow(t) = 0$

Данные и управление в схеме

Для “грамотной” разработки сложных схем повсеместно используется
принцип разделения данных и управления

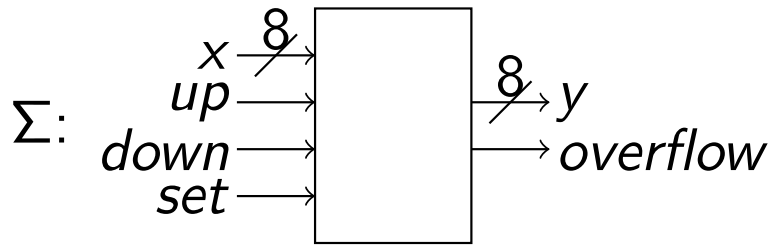
Если отбросить некоторые детали и исключительные случаи, то
начало применения этого принципа выглядит так

Соединения схемы делятся на информационные и управляющие

Значениями в управляющих соединениях (управляющими значениями)
задаются “настройки” выполнения подсхем и внешних схем:
режимы работы, способы пересылки и преобразования сигналов, ...

Значения в информационных соединениях (данные)
“пассивно” преобразуются подсхемами согласно текущим настройкам

Данные и управление в схеме



Например, в Σ:

- 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 режимами работы этой схемы

Операционный и управляющий автоматы

После разделения всех соединений на информационные и управляющие вся схема делится на две соответствующие части (подсхемы):

- Σ_d: все информационные соединения
  и все подсхемы, к портам которых подключены эти соединения
- Σ_c: все управляющие соединения и все подсхемы, не попавшие в Σ_d

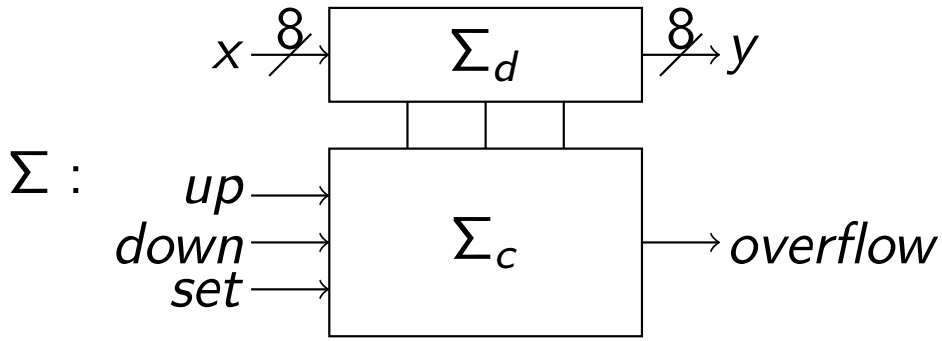
У каждой из этих частей есть несколько названий,
возникших в разных научно-технических школах:

- Советская школа
  - Σ_d: операционный автомат[1]
  - Σ_c: управляющий автомат[1]
- Американская школа + современный перевод
  - Σ_d: контур данных (datapath), или блок данных (data unit)
  - Σ_c: контур управления (controlpath),
    или блок управления (control unit)

[1] Слово "автомат" здесь не совсем адекватно: *не каждой схеме соответствует автомат* — но это *традиционные* названия, и в таких названиях часто используются *устаревшие* слова

Операционный и управляющий автоматы

Например, для Σ разделение
на операционный и управляющий автоматы выглядит так:



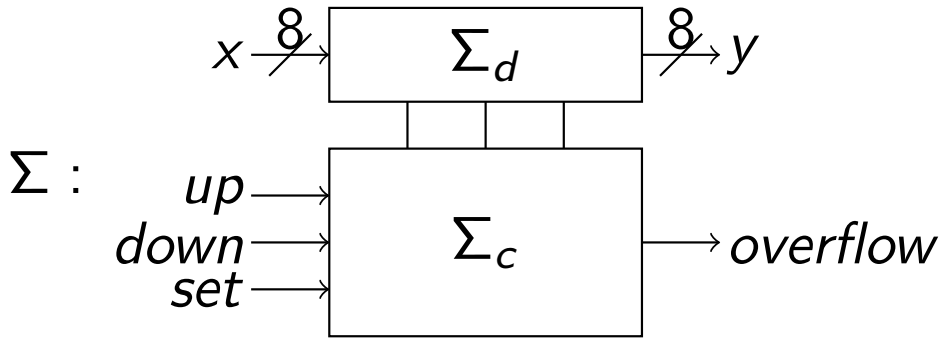
Согласно “грамотному” подходу к разработке больших схем,

- операционный и управляющий автоматы
  проектируются *почти* независимо друг от друга, и
- проектирование этих автоматов
  основано на принципиально разных подходах

Операционный и управляющий автоматы

Например, для Σ разделение

на операционный и управляющий автоматы выглядит так:



Такое *почти* независимое проектирование позволяет

- в реализации схемы отдельно и независимо ответить на вопросы "**что** в принципе схема должна уметь делать с данными?" и "**как** фактически схема будет работать с данными?"
- избежать "глупых" ошибок, возникающих из-за обилия деталей и случаев при смешивании "что" и "как" в реализации
- избежать лишних трудозатрат при дальнейшем использовании схемы (оптимизация, правка функционала, переиспользование подсхем), изменяя/используя "что" независимо от "как" и наоборот