МФК по ПЛИСам

justanothercatgirl

2025

Содержание

Лекция 1: Мфк начинается (введение) (2025-02-19)

О чём говорим: что такое интегральная схема, как их проектировать, уровни абстракции (как перейти от задумки к реализации), маршрут проектирования, меры качества и оптимизации

Процесс создания интегральных систем: Фотолитография (на кремний переносим фотошаблоны и всякое такое)

Правило Мура: в 1965 году Гордон Мур сказал, что каждые 2 года число транзисторов удваивается, потом поправил на каждые полтора года, но сейчас у нас физические ограничения:

- Транзисторы имеют конечные размеры
- Транзисторы только в 3-хмерном пространстве
- Скорость света ограничена
- статься: Limits on fundamental limits to computation (nature)

Подходы к проектированию:

- Заказное проектирование: человек думает вручную целиком
- Полузаказное: всё ещё человек думает, но он использует заранее спроектированные элементы
- Программируемые Логические Интегральные Схемы (ПЛИС): Они заранее спроектированы, но архитекрурно они устроены так, что логику можно конфигурировать.

Мы каждый логический вентиль можем представить как схему из набора транзисторов.

А как сделать программируемую матрицу? У нас есть сетка коммутаторов и сетка логических элементов, и коммутаторы ”запоминают”, какие соединения использоввать, чтобы получить разные функции у ЛЭ.

Второй вариант: Хранить в ячейках памяти разные логические схемы (реазируем функции 4 переменных через мультиплексеры, которые выбирают адреса памяти, строим из них древо и получаем 2^n возможных значений)

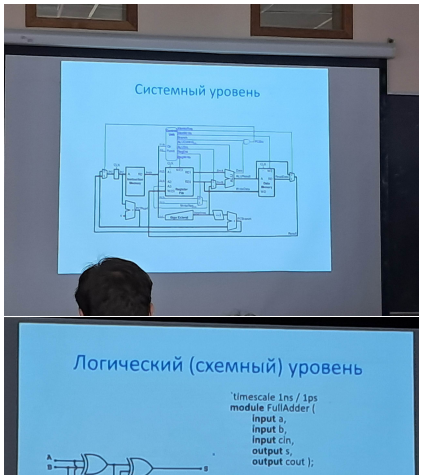
На кристалле системы состоят из макроблоков, например: Память, стандартные ячейки, Аналоговая схема, АЦП и ПЛИС с контактами (они все взаимосвязаны)

Мы будем рассматривать Сихнронные схемы! Большинство современных схем устроены по синхронному принципу: есть комбинационный логический блок, и под тактовый генератор у нас одно вычисление может занимать несколько тактов (логический блок сохраняет, какую долю вычислений мы произвели); например, деление занимает много тактов.

Лекция 2: Собственно, проектирование (2025-02-26)

Уровни Абстракции

У нас есть таки уровни:



1. Системный - набор концептуальных элементов, ”блоков”

   например, устройство микропроцессора:
   Есть блок с памятью, хранящий в себе инструкции - 1 адресный вход, один выход на чтение
   Есть регистровый файл - N универсальных регистров, 3 адресных входа, 2 читательных выхода
   ALU - Арифметико-логическое устройство, к нему подходит регистр и мультиплексор {регистр|число}
   ALU может отправить результат в память, при этом результатом

ALU может быть адрес
Так же есть счётчик команд - `%rip`
за всем этим следит определённый блок управления

То есть, другими словам - это ”концептуальный” уровень, мы созаём систему команд, микроархитектуру, взаимодействие компонент системы, использование HDL

2. Автоматный - или поведенческий (каждый компонент нашей схемы - это автомата)

   На этом уровне мы описываем поведение блока как набор команд, которые он исполняет, например, сумматор: у него 3 входа, 2 выхода, он складывает 3 однобитовых числа, и как результат даёт два бита: результат и переход на новый разряд

   Мы используем HDL, автоматные модели, чтобы описать логику.
   Сам уровень ещё называется RTL - Register transfer level

3. Логический - представляем автомату из логических схем

   Тоже можно использоваать HDL - через wire, and, or и т.п.

   Отсюда мы уже можум получить схемы из функциональных элементов, моделируем структуры

   Так же это называется gatelist

4. Транзисторный - каждый логический элемент - это электрическая транзисторная схема

   Каждый элемент представлен транзисторной схемой, мы можем оценить и понять, как работает какой логический элемент. Как правило, это предоставляется какой-нибо библиотекой

   Тут мы используем иммитационное моделирование, проверяем утечки энергии, понимает физику системы

5. Топологический - схема имеет геометрическую форму

   Тут мы рисуем, как будет выглядеть элемент в кремние, имеем геометрию

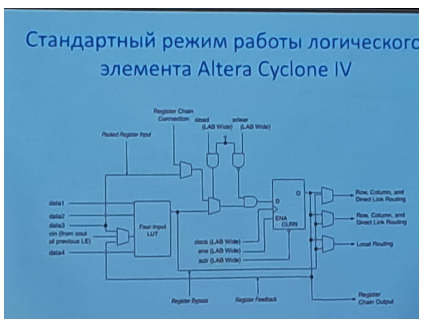
   На заводе происходит много задач: DRC (Design rulecheck) [], LVS (Layout Vs Schematics)

   Сейчас у лазеров длина волны 162 нм; при этом транзисторы 5-10 нм.

Рассмотрим это на примере ПЛИС `DE0-Nano`:
У нас есть Периферия и Altera Cyclone IV: Сам ПЛИС
Altera устроена из блоков матрицы, блоки состоят из векторов по 32 ячейки, каждая ячейка - та самая ”настраиваемая” логическая функция

Внутри каждой ячейки 4 входа, они могут выводиться на соседние ячейки, область трассировки или в регистры. Так же блок может работать в режиме однобитового сумматора (сделано для скорости).

Каждый блок имеет шину, соединяющую ячейки, а между блоками есть общие шины, чтобы передавать сигналы в соседние блоки



Классический маршрут проектирования

Спецификация системы (мысли) $\rightarrow$ проектирование архитектуры (HDL) $\rightarrow$ Функциональное проектирование $\rightarrow$ Логическое проектирование (соединям друг с другом схемы) $\rightarrow$ Физическое проектирование (DRC, LVS, ERC) $\rightarrow$ Проверка топологии $\rightarrow$ Корпусирование и тестирование

Пример:

Счётчик чётности - 3 входа: IN (4 бита), CLK, RST, выход булевый.

Нам нужно привязать "абстрактный элемент, проверяющий чётность" ячейкой из библиотеки. Физическое проектирование заключается в расположении этой ячейке на матрице.

Метрики качества: цена, Надёжность, скорость, энергопотребление.

Лекция 3. HDL, введение в Verilog (2025-03-05)

Популярные языки описания аппаратуры: Verilog, VHDL, SystemVerilog, SystemC

Есть 2 стиля разработки: сверху-вниз и снизу-вверх (соответственно, по уравням)

Так же есть 3 уровня абстракции:

1. Поведенческий (behavioral) - арифметические операторы, алгоритмы, автоматы и т.п.
2. Уровень регистровых передач (Register-Transfer) - описание алгоритмов на уровне операций с регистрами
3. Уровень схемы (Gate) - функциональные элементы в библиотеках

про Verilog

Изначально он разрабатывался для симмуляции работы реальной схемы; в итоге он стал использоваться для их синтеза.

Логические значения:

- 0: логический 0, значение напряжения
- 1: логическая 1, другое значение напряжения
- x: неопределённое значение
- z - высокий импеданс - по сути это значит, что значение в ущле неуправляемо (нам оно не понадобится)

Мы будем строить схему, между узлами которой в реальное время передаются логические сигналы:

negedge: 1 -> 0

posedge: 0 -> 1

Типы данных в языке: net data types (wire), variable data types (reg)

wire: Как правило, лоигческий вентиль reg: регистр, триггер или т.п.

Пример объявления: `wire [MSB:LSB] x;` - объявляем (беззнаковую) шину шириной `MSB-LSB+1`, MSB = Most significant bit, LSB = Least significant bit

"Строительные блоки" - это **Модули**

У каждого модуля есть входы и выходы и тело, IO могут быть reg и wire, так же могут быть шины [bus]

Концептуально: модуль - это описание какой-то схемы, по аналогии это что-то вроде функции, функтора, класс и т.п.

Naming convention: для модуля создаём `.v` файл с названием, совпадающим с названием модуля:

```
module mod(a, b, c);
    input wire a;
    input wire [1:0] b;
```

```
    output reg [2:0] c;
    // implementation here
endmodule;
// DONT FORGET NEWLINE (cause it's from 1900s, yeeeah)!!!
```

Если что, можно писать и внутри функции типы переменных

Способы описания модуля

Структурный: Пишем модули и связи между ними
Функциональный: Задаём поведения модулей и связи между ними, но при этом не паримся о структуре этих модулей

```
//we have modules m1 (i_1, i_2 -> 0), m2 (i_1, i_2 -> o_1, o_2)
module m(input i1, i2, i3, output o1, o2, o3) // default: wire
    wire w;
    m1 upleFFFt(.i_1(i1), .i_2(i2), .o(w));
    m1 dOwnright(.i_1(i2), .i_2(i3), .o(o3));
    m2 lASSt(.i_i(w), .i_2(o3), .o_1(o1), .o_2(o2));
endmodule
```

Синтаксис подключений: `<module_name> <var_name>(.module_input(from_where), …)`
Так же в Verilog есть предзаданные модули: `or, and, xor, nor, nand, xnor`
У них первая первый аргумент всегда выход, остальные - всегда входы


```
module and_from_nand (output o, input i1, i2);
    wire mid;
    nand U1(mid, i1, i2);
    nand(o, mid, mid);
endmodule;
```

Про защёлки и триггеры: Триггеры реагируют на изменение сигнала, защёлки - на уровень сигнала
Чтобы создать ячейку памяти, нужно создать цикл из инверторов: система, которая сама себя поддерживает
На этой лекции всё, а на следующей будет непрерывное присванивание

Лекция 4: продолжаем Verilog (2025-03-12)

`always`-блок: он будет срабатывать всякий раз, когда аргумент становится правдой (как event/inperrupt handler)

```
always @(a or posedge b or negedge c)
// statement
```

Все проверяемые полюса должны быть **регистрами**
Вместо `or` можно ставит запятую. Разница есть, но небольшая

Блокирующие/непрерывные операторы присваивания

Пример блокирующего:

```
begin
```

```
    b = c;
    a = b;
    c = a;
end
```
Эти действия будут выполняться ”последовательно”. Результат: поменяются местами `a, c`

Пример неблокирующего: просто заменить = на <=

```
begin
    a <= c;
    c <= a;
end
```
Эти действия будут выполняться ”одновременно”: изменения в переменные вносятся в конце, справа от знака присваивания всегда одно значение

управляющие конструкции

`if (cond) stmt; else stmt;` - **условие**

`case (a) 3'b000: smtm; ...; default: stmt; endcase` - **переключение**, работает как в C

`initial-block`: запускается при инициализации платы

Пример: модуль регистра:

```
module register3(input load, reset, clock, input [2:0] in, output reg [2:0] out);
    always @(posedge clock, negedge reset)
        if (~reset) out <= 0;
        else if (~load) out <= in;
endmodule
```

Структура программы на Verilog:

1. набор модулей, которые описывают различные функциональные элементы
2. модуль определяет ввод/вывод
3. может содержать эементы
4. в теле может быть initial, reg, присваивания, always, другие модули (их экземпляры)

Замечание: модуль - это не функция!! это именно функциональный элемент. его нельзя вызвать!
Главный модуль: точка входа (top-level-entity)

Пример: счётчик

Входы: `clock, reset` (сброс значения при 1), `enable`
Выход: `counter_out`

Ссылки:
Полезные ресурсы по верилогу: https://www.asic-world.com/
Онлайн редактор: https://edaplayground.com
Прикольный онлайн проект: https://github.com/MIPSfpga/schoolMIPS

```
// ?????
```

Параметры и generate-блоки

Типа **шаблонные параметры** - шаблоны модулей (как `template<>` в C++)

```
module register #(parameter width = 5) (args...)
    // ...
endmodule
```

generate-блоки

```
generate
    genvar i;
    for (i = 0; i < WIDTH; i = i + 1)
        begin: stage // names are mandatory!
            // generated expressions...
        end
endgenerate
```

се директивы начинаются с символа ` (на букве ё)
В этих блоках **можно**: использовать `generate, always, <=,`
в этих блоках **нельзя**: аааaа слайд переключилиииии
Внутри generate-блока меняется смысл `case, for, if`, если у нас итерация идёт по `genvar`

Препроцессор

Позволяет заменить код пред компиляцией. директива `` `define ALU_AND 2*b00 ``, намимер, сделает так, чтобы в коде `` `ALU_AND `` будет заменяться на `2*b00`

Лекция 5. А как аппаратный ускоритель собрать? (2025-03-19)

У нас есть проблемы с... числами. Например, с 1 байтом мы имеем всего 2^8 чисел, а с N байтами 2^{8N}.
С вещественными числами ещё хуже. Мы их представляем в формате *знак-мантисса-экспонента*.
Поэтому плотность точек у начала координат очень высокая, а вдали от начала координат очень маленькая.
`fixed-point arithmetic`: Степень фиксируют, мантисса остаётся.

Сумматоры (Full Adder, FA)

Имеют 2 входа известной размерности (+1 на перенос разряда), и 1 выход (+1 переход разряда)

```
module adder#(parameter WIDTH = 8)(
    input [WIDTH - 1 : 0] x, y,
    input carry_in,
    output [WIDTH - 1 : 0] z,
    output carry_out
);
    assign [carry_out, z] = x + y + carry_in;
endmodule
```

компараторы

Имеют 2 входа для чисел и 1 (или 6)))) выходов для результата сравнения

```
module comparator#(parameter WIDTH=8)(
        input [WIDTH-1:0] x, y, output eq, neq, ge, le, geq, leq
);
    assign eq = (x == y);
    assign neq = (x != y);
    ...
endmodule
```

устройство сдвига

Есть арифметический сдвиг, логический сдвиг и циклический сдвиг. арифметический: зависит от знака, логический: просто двигает биты, а пустые места заполняет нулями, циклический: "перебрасывается".

Логический есть в стандарте: `<<, >>`.

Чтобы сделать циклический, можно сделать дополнительную шину размером `2 * WIDTH`, в обе её части заложить число и сдвинуть её: тогда в одной из копий получатся нули, а во второй - то, что нам нужно. (в правой или левой копии - зависит от направления сдвига)

ALU (арифметико-логическое устройство)

Имеет 2 входа для чисел, один для `opcode`, одно число на выходе и, условно, вспомогательные выходы по типу ошибок (`underflow, overflow e.t.c.`)

Часто (самые простые) ALU устроены так, что результаты вычисляются для КАЖДОЙ арифметической операции, а потом с помощью мультиплексоров выбирается то, что нужно.

Параметры оценивания блока:

1. Сложность
2. Задержка
3. Энергопотребление

Проблема каскада сумматоров: Если мы подключим N сумматоров в ряд (через переносы через разряд), то задержка будет составлять $\underline{O}(N)$, что очень плохо. Можно ли сделать по-другому? Можно, с помощью *carry-lookahead*. Делается это через generate & propagate проводов.

Сегодня лекция была супер которенькая, т.к. у препода жёйско болело горло.

Лекция 6. Автоматы. (2025-03-26)

Конечный автомат: совокупность 3 алфавитов (A, B, Q) - входной, выходной и состояний, 2 функции $(\varphi : A \times Q \to Q, \psi : A \times Q \to B)$ - состояний и переходов, и начальное состояние q_0
$Aut = \{A, B, C, \varphi, \psi, q_0\}$

Можно считать, что автомат работает во времени (время дискретно), получая данные на входе и выдавая их на выходе (по алфавиту).

Пример автомата: потоковый двоичный сумматор: $A = \{00, 01, 10, 00\}, B = Q = \{0, 1\}$. Состояние тут - это либо перенос есть, либо переноса нет. Функция перехода - это логическая таблица (в зависимости от первого, второго битов A, бита Q).

Отличие от машины тьюринга в том, что автоматы не могут ”двигаться назад”, а машина тьюринга может, и ещё может стоять на месте.

Любой конечный автомат можно представить в виде диаграммы Мура - графа состояний, а рёбра графа есть входные и выходные данные.

Отношение автоматов к проектировке

Сначала мы задаём каждое сосояние в виде двоичной последовательноси, потом мы реализуем блок с функциями φ, ψ, и подключаем его к элементу синхронизации

Для автоматов используются always-блоки:

```
always @(state, in)
    out = state ^ in[0] ^ in[1];
    case state
^^I1'b0: if (in == 2'b11)
            next_state = 1'b1;
        else
            next_state = 1'b0;
        1'n1: if (in == 2'b00)
            next_state = 1'b0;
        else
            next_state = 1'b1;
    endcase

always @(posedge clock, negedge reset)
    if (~reset)
        state <= 1'b0;
    else
        state <= next;
```

Тут логика такая: всегда, когда меняется state или вход, запускается автомат сумматора. При этом меняется `next_state`. Потом часы записывают `next_state` в `state`

У нас есть парадигма ”основного” (вычислительного) и управляющего автомата.

Пример

Допастим мы хотим сделать такую систему: `for (int i = 0; i < 10; ++i) x+= y; if (x < 0) y = 0; else x = 0;`

Сначала мы реализуем регистр: у него опрерации LOAD и CLEAR; потом у нас есть сумматор, у которого выход идёт на регистр, а регистр замкнут на складыватель (т.е. один вход сумматора - y, а замкнутый вход - x. Потом у нас есть компараторы (2), один считает циклы, второй сравнивает x и 0. То есть в итоге у нас 6 входов (load, clear на 3 переменные), 3 выхода (x, y, i) и 2 выхода (с компараторов)

Потому у нас есть управляющий автомат:

состояние A: начальное; B - загрузка x, i, C - ожидание, D , E, F не успел списать с доски, всего 6 (т.е. 3 бита)

Лекция 7. Static timing analysis. (2025-04-02)

Статический временной анализ - мы пытаемся расчитать время переключения ”в худшем случае”, и распространяем эту оценку на всё остальное.

Проблемы синхронизации

Есть 2 типа ограничения:

- время установки сигнала (время, требущееся на вычисление всех элементов)
- ограничение до переключения синхронизации (long-path и short-path constraints)

То есть long-path: это сколько сигнал должен быть стабилен **до переключения**, а short-path: сколько он должен быть стабилен **после переключения**

То есть у нас коспромис: либо считаем быстро, но неаккуратно, либо аккуратно, но медленно))))

Примр: у нас в ряд подключено несколько блоков комбинационной логики (без задержки), и между ними стоят системы синхронизации (задержки), подключенные к часам, и задача - сколько времени делать задержку...

Основные причина задержен:

- gateway (транзисторы)
- wire (распространение сигнала)
- clock skew (перекос во времени прихода синхросигнала. Например, даже если мы везде выдержим длину проводов, могут появиться различия из-за температуры платы и т.п.)

Алгоритмы синхронизации

Комбинационный блок можно представить как направленный ациклический граф (DAG), где каждое ребро графа является задержкой, и мы рассчитываем самое позднее возможное время: потом рассчитываем то, что слэнгово называется *slack = допустимое время прихода сигнала - максимальное время прихода*: если он положительный, то всё ок, если отрицательный, то могут быть проблемы (а могут и не быть).

Используя теорию графов и динамическое программирование, можно довольно оптимально вычислить максимальное время прихода сигнала: т.к. граф ацикличный, поэтому можем устроить что-то вроде поиска вширину.

Из плюсов такого подхода: сложность у него линейная, поэтому он очень быстрый. Из минусов, он очень пессимистичный - часто оценивает задержку намного больше, чем нужно.

Глюки (трепетания) ((glitches))

Про энергопотребление: энергия тратится в основном на ”переключение” элементов. Оптимизировать ”переключения” крайне сложно.

трепетания - это дополнительные ”переключения” выходного значения элемента из-за разницы в приходе сингала, и, хоть на результат они не влияют, но оочень сильно влияют на энергопотребление (схема работает вхолостую).

Пример: допустим, элемент имеет на вход 2 параметра. Один из них пришёл намного раньше первого - поэтому на выходе у нас какое-то время может быть неправильное значение. Когда придёт второй сигнал, на выходе значение становится правильным, при этом оно переключается. В более сложных схемах это крайне быстро выходит из-под контроля (потому что каждое переключение одного элемента может вызывать переключение следующих, а потом последующих и т.п...)

Критические пути

Формально это тот путь, который требует наибольшее время на вычисление. При этом в жизни этот путь может не совпадать с расчётами - например, возьмём конъюнкцию: если один из входов 0, уже можно не ждать второго значения, и это сильно мешает расчётам. Поэтому появляются ”ложно критические” пути. Так же это может быть из-за того, что схема спроектированна так, что никогда в теории не может подаваться какой-то набор значений, и критический путь становится уже другим.

Входы и выходы (огранизация систем)

У нас есть абсолютно независимые системы (коробка, которая всё время что-то делает), и зависимые (например, можем хотеть заставить говорить смеху и компьютер, наприме). Поэтому нам нужно решить вопрос **коммуникаций** и **протоколов взаимодействия**

Протоколы коммуникации обязательно должны быть стандартизированы (чтобы не произошёл k+r C момент). Характеризуются следующими параметрами: Тип или предназначение (что передавать, много и мало или редко и много), расстояние (по воздуху или по проводам), производительность (с какой скоростью им можно пользоваться), безопасность (можно ли перерезать провод и подслушать, что там происходит) и надёжность (TCP vs UDP).

типы:

1. GPIO (general-purpose input-output), просто поключаем провода, может быть для простого обмена данных
2. Serial IO: есть два волка - параллельная и последовательная передача данных.
3. протокол SPI (серийный переферийный интерфейс): основан на синхронности. 1 сигнал: часы мастера, 2, 3 - двунаправленная побитовая коммуникация плат, 4 сигнал - ”выбор”, откуда читать данные (каждому устройству даётся свой индекс). Поддерживает: 1 мастер + несколько подчинённых, full-duplex (полная двунаправленность) скорость передачи 0-50 мегабит в секунду, но нет стандартизаци На самом деле чаще всего данные передаются блоками 4-16 бит

   Сам процесс передачи данных, как правило, устроен по принципу ”сдвиговых регистров”. При этом мастер более чем вправе менять свою тактовую частоту)

   Плюсы: быстрый, простой, нет адресации, широко поддерживается. Минусы: выбор подчинённого, нет подтверждений целостности, нет управления потоком, завязан на часах.
4. протокол UART (Универсальный Асинхронный Приниматель или Получатель, universal asynchronous reciever transmitter). У него нет завязки на часы, поэтому у него есть свой ”формат” передачи данных: коммуникация идёт ”пакетами”, сначала стартовый бит (LOW), потом данные, потом флаги, потом стоповый бит (STOP), когда нет передачи, сигнал всегда HIGH

   Сам UART обновляется в 16 раз чаще, чем обновляются данные (передача). Приниматель может посчитать ”центр” передачи (т.к. каждое значение передаётся 16 раз), но при этом они всё ещё должны договориться о частоте передачи (а то получится ситуация как с пересекающимися волнами синуса)

   Коррекция ошибок: при передаче считается сумма всех бит и отправляется в конце, при этом читатель проверяет чексумму. Так же может быть такое, что стоповый бит не пришёл, обычно при этом передача повторяется.