Пакеты проектирования сверхбольших интегральных схем

Лектор:

Подымов Владислав Васильевич

e-mail:
valdus@yandex.ru

Осень 2016

Лекция 1

Часть 1

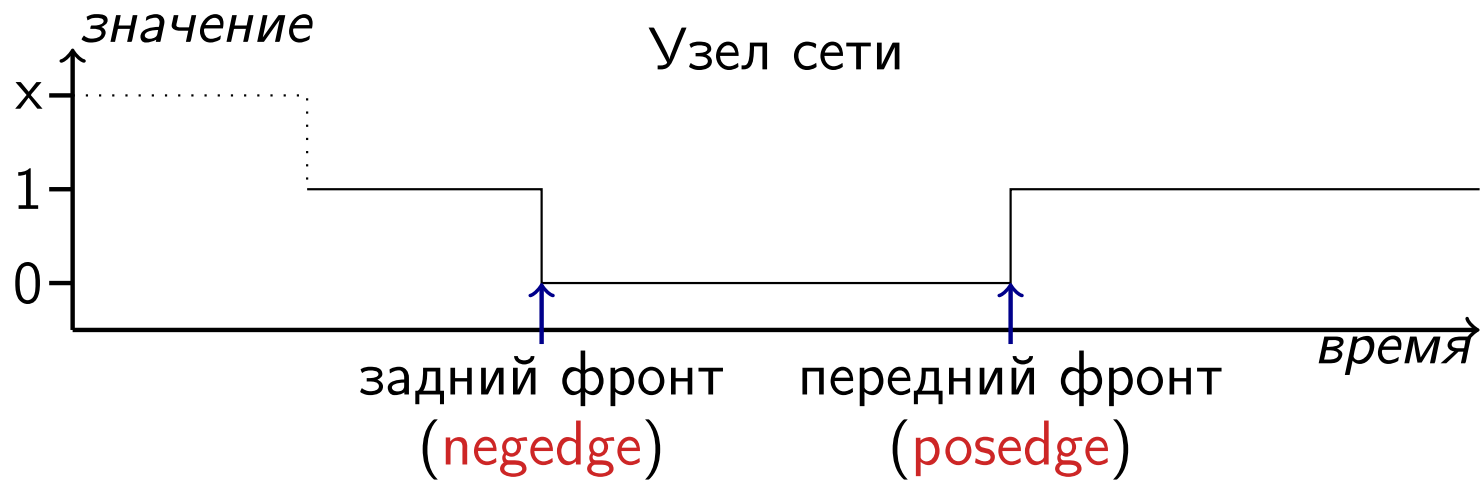
Напоминание основ

Логические значения

- 0 — логический ноль
- 1 — логическая единица
- x — неопределённое значение
  - *например, в неинициализированном регистре*
- z — состояние высокого импеданса
  - *оно нужно в основном для управления шиной, к которой подключено много независимых устройств*
  - *в ближайшее время это значение нам не понадобится*

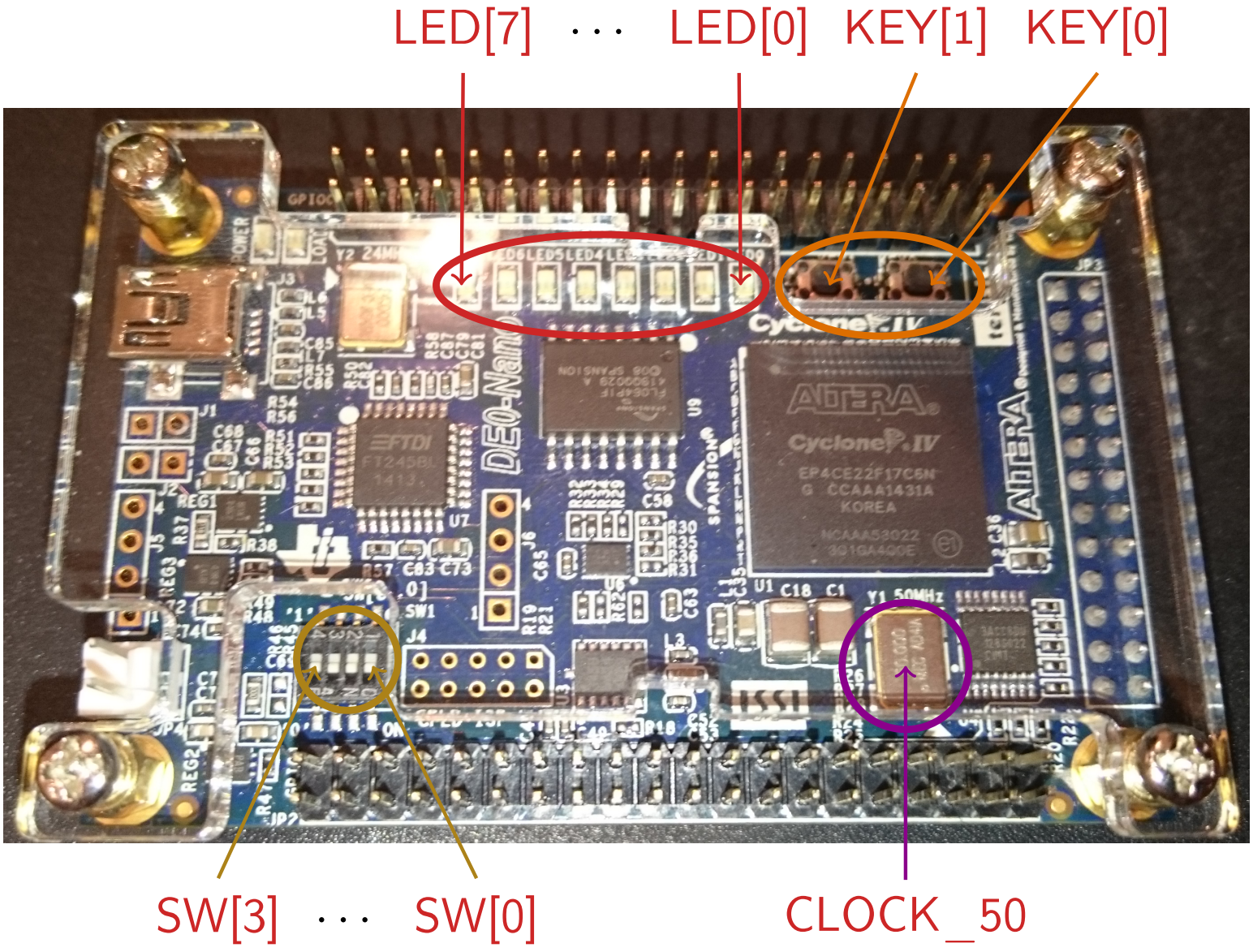
Логические значения

Что мы строим: схему, между узлами которой в реальном времени передаются логические значения



- **0** и **1** — это конкретные **уровни напряжения**
- **x** — это **абстракция**: уровень напряжения, соответствующий неизвестному логическому значению

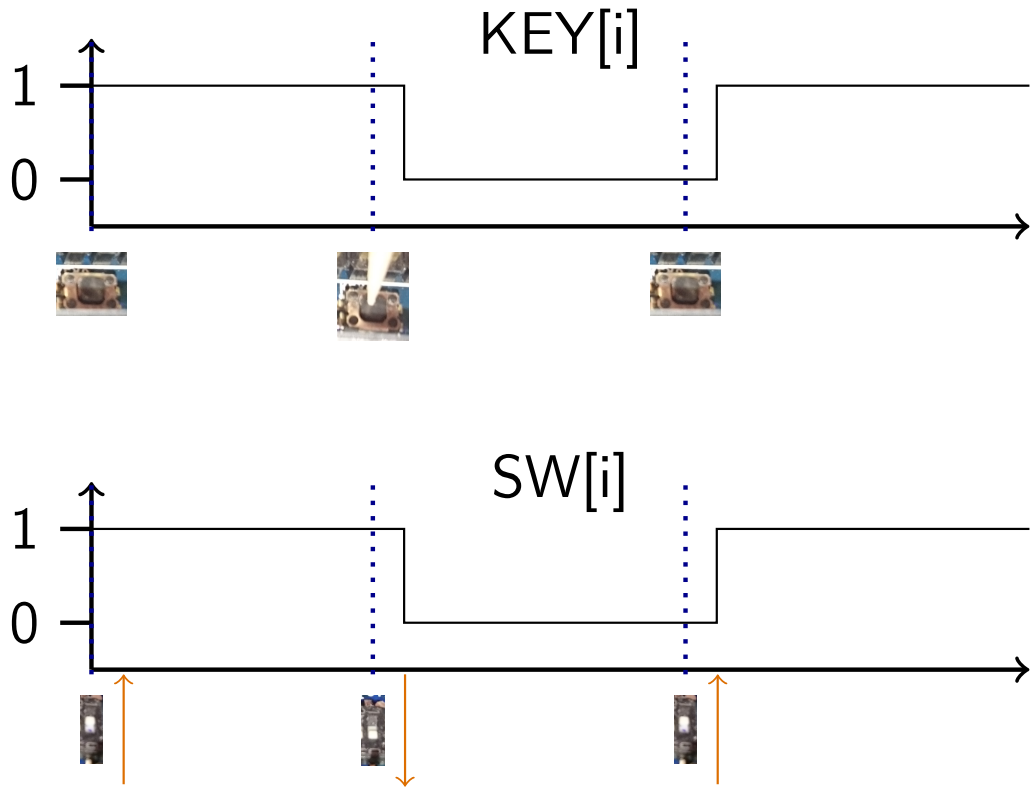
Ввод-вывод DE0-Nano



(*и ещё GPIO, но они нам пока не нужны*)

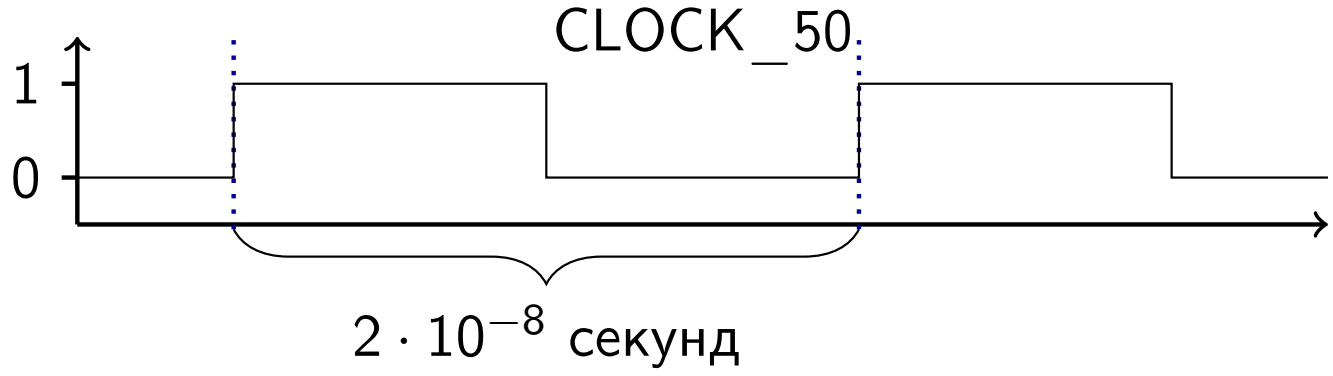
Ввод-вывод DE0-Nano

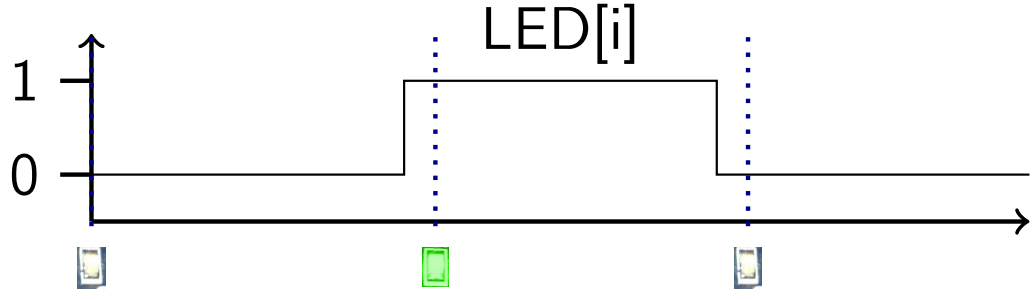
Как ведёт себя ввод-вывод DE0-Nano



Ввод-вывод DE0-Nano

Как ведёт себя ввод-вывод DE0-Nano



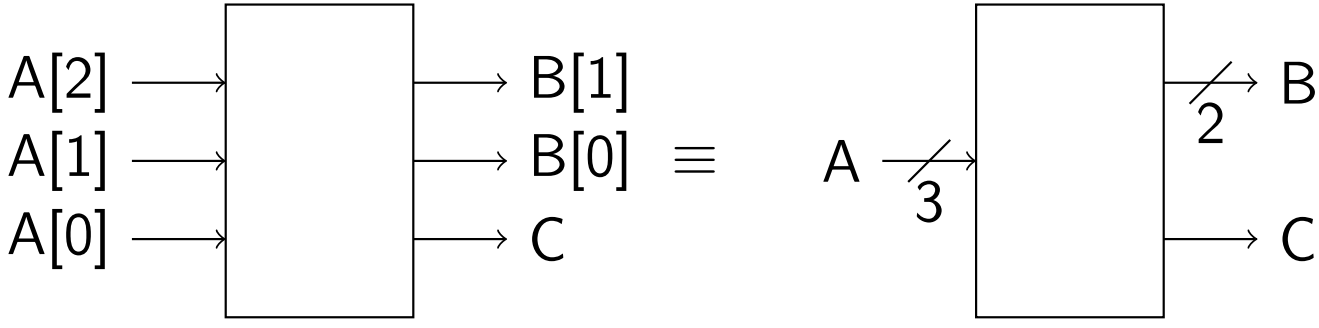


Verilog: модули и шины

"Строительный блок" дизайна Verilog — модуль:

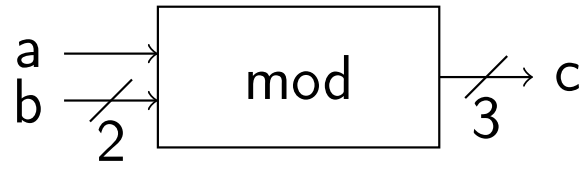
Входы и выходы (*и многое другое*) могут быть объединены в массивы (индексация с ноля; при определении задаются смещения сначала последнего, потом первого элемента: [7:0]; в остальном — как в C++)

Массив проводов (входов, выходов и других) — это шина:



Verilog: определение модуля

Лучше всего описывать модуль в отдельном файле с расширением .v и названием, совпадающим с названием модуля



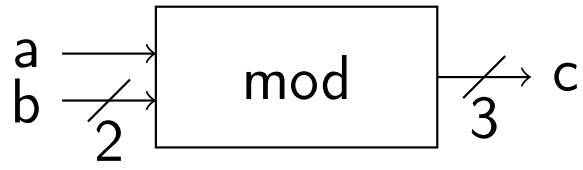
Файл `mod.v`: (*первый вариант*)

```
module mod(a, b, c);
  input a;
  input [1:0] b;
  output [2:0] c;

  // description
endmodule
// EMPTY LINE!
```

Verilog: определение модуля

Лучше всего описывать модуль в отдельном файле с расширением .v и названием, совпадающим с названием модуля



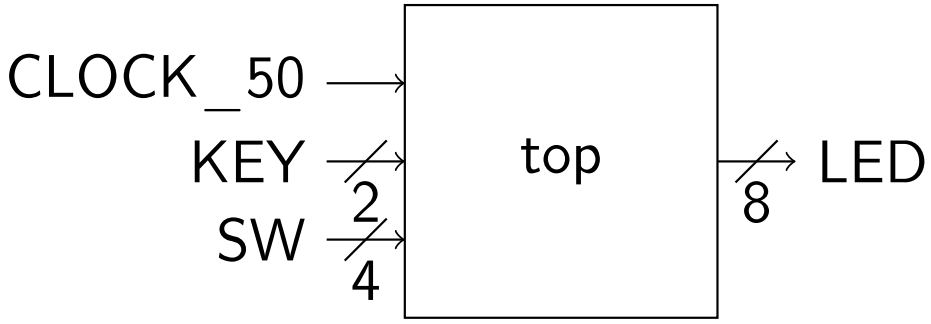
Файл `mod.v`: (*второй вариант*)

```
module mod
   ( input a,
     input [1:0] b,
     output [2:0] c
   );
   // description
endmodule
// EMPTY LINE!
```

Verilog: главный модуль

Среди модулей обязательно есть главный (top module): это и есть разрабатываемая схема, которая будет заливаться в FPGA

При работе с DE0-Nano главный модуль будет выглядеть так:



Файл `top.v`:

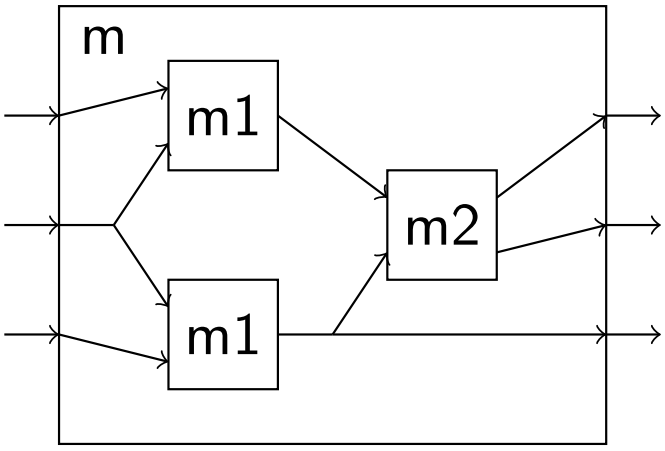
```
module top
  ( input [1:0] KEY,
    input [3:0] SW,
    input CLOCK_50,
    output [7:0] LED
  );
  // description
endmodule
```

Verilog: способы описания модуля

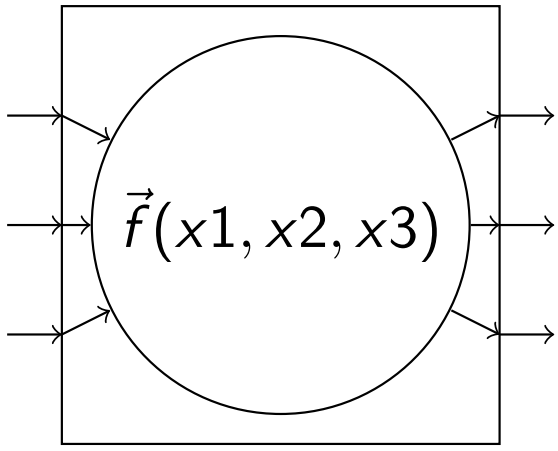
Обычно различают два подхода к описанию модуля:

- структурный: явно описать инстанциации (*страшная калька с английского*, будем называть их экземплярами) других модулей и связи между ними
- функциональный: без явного описания структуры задать взаимосвязь входов и выходов

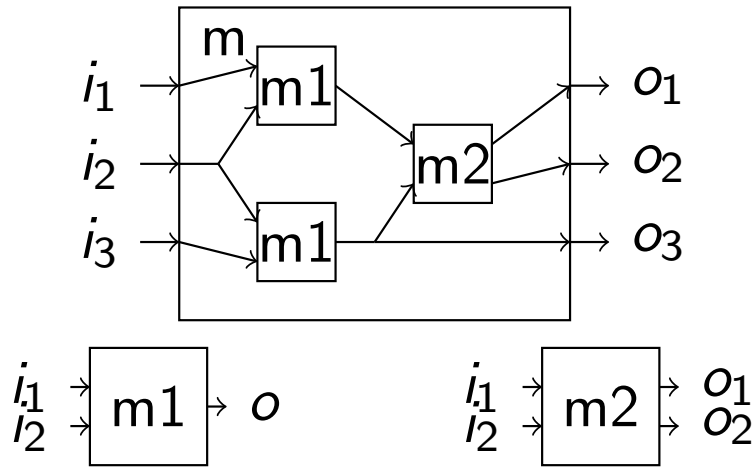
Структурное описание:



Функциональное описание:



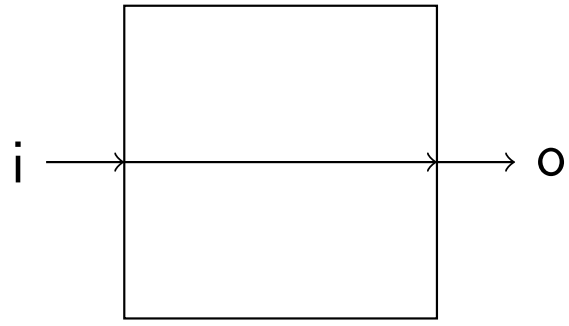
Verilog: структурное описание
экземпляры модулей




```
module m(input i1, i2, i3, output o1, o2, o3);
  wire w;
  m1 upleft(.i1(i1), .i2(i2), .o(w));
  m1 downleft(.i1(i2), .i2(i3), .o(o3));
  m2 right(.i1(w), .i2(o3), .o1(o1), .o2(o2));
endmodule
```

Все используемые имена (в том числе провода: wire) должны быть определены перед использованием

Verilog: структурное описание непрерывное присваивание




```
module trivial(input i, output o);
  assign o = i;
endmodule
```

assign провод = выражение; :

- в любой момент времени (*с некоторой задержкой при изменении значения*) на проводе должно быть значение выражения

Verilog: выражения

Что можно использовать при написании выражений:

- логические операции
  - например, `a && b` — логическое И
- арифметические операции
  - например, `a + b` — это сложение двух чисел одинаковой битности с переполнением
- побитовые операции
  - например, `a & b` — это побитовое И двух битовых массивов одинаковой длины
- отношения
  - например, `a < b` возвращает логическую 1, если число, двоичная запись которого есть `a`, меньше такового для `b`, и логический 0 иначе

Verilog: выражения

Что можно использовать при написании выражений:

- конкатенации
  - например, `{a, b}` — битовый массив, составленный из `a` и `b`
- редукции
  - например, `&a` — логическая 1, если все биты `a` — единицы, и логический 0 иначе
- условия
  - например, `cond ? a : b` работает как в C++; `cond` должно иметь логическое значение, а `a` и `b` должны иметь одинаковое число бит
- константы
  - например, `0` — это логический ноль, а `5'b00110` — пятибитная двоичная запись числа 6

(*полный список операций спрашивайте у интернета*)

Verilog: функциональное описание
блок always

Он выглядит так:

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

В аргументе перечисляются места (*например, провода*), при изменении сигнала в которых должно производиться какое-то действие

В данном случае:

- при изменении логического значения в a,
- а также когда в b возникает передний фронт,
- а также когда в c возникает задний фронт

Действие перезаписывает значения сигналов модуля

После выполнения действия получившиеся значения **сохраняются** в проводах **до следующего выполнения** блока

Verilog: функциональное описание
блок always

Он выглядит так:

```
always @(a or posedge b or negedge c)
begin
  // sequence of statements
end
```

Действий можно задавать много, и тогда их обычным программистским образом нужно соединить в составное действие

Verilog: функциональное описание
блок always

Он выглядит так:

```
always @(a, posedge b, negedge c)
begin
  // sequence of statements
end
```

В какой-то момент разработчики стандарта Verilog поняли, что “or” писать неудобно, так что разрешили вместо него ставить запятую

Какие же действия можно писать в always-блоке?

Verilog: функциональное описание блокирующее присваивание

```
always @(b, c)
begin
  b = c;
  a = b;
  c = a;
end
```

- **Последовательно** делается следующее:
  - в b выставляется *начальное* значение из c
  - в a выставляется *изменённое* значение из b
  - в c выставляется *изменённое* значение из a

Блокирующее присваивание моделирует последовательное выполнение команд: пока присваивание не выполнено, следующие команды не выполняются
(*но в конечном итоге строится схема, просто она имеет хитрую структуру с блоками памяти*)

Verilog: функциональное описание неблокирующее присваивание

```
always @(b, c)
begin
   b <= c;
   a <= b;
   c <= a;
end
```

- **Одновременно** делается следующее:
  - в b выставляется *начальное* значение из c
  - в a выставляется *начальное* значение из b
  - в c выставляется *начальное* значение из a

Вообще говоря, *одновременности* не бывает, но в реальной схеме эти действия будут выполнены близко по времени, и блоки памяти будут организованы так, чтобы выставлялись именно *начальные* значения

Verilog: ~~регистры~~ переменные

При выставлении сигналов в схеме могут понадобиться дополнительные (*неявные*) **ячейки памяти**

Чтобы компилятор имел возможность распознать такие места и по необходимости синтезировать дополнительную память, в Verilog вводится понятие ~~регистра~~ переменной (*терминология менялась в стандарте*)

Всё, что появляется в присваиваниях (=, <=) слева, **должно** быть обявлено как переменная:

```
reg a, b, c;
always @(b, c)
begin
  b = c; a = b; c = a;
end
```

Всё остальное *может* быть объявлено переменной

Verilog: ~~регистры~~ переменные

- имя **не может** одновременно быть переменной и проводом
- в некоторых случаях (например, при встрече в левой части непрерывного присваивания) имя **не может** быть переменной
- все входы и выходы являются проводами по умолчанию
- все входы **обязаны** быть проводами
- выходы *можно* определять как переменные: достаточно
  - дописать в начале модуля `reg <имя выхода>;` или
  - при определении выхода написать

    `output reg <имя выхода>`

    вместо

    `output <имя выхода>`

Verilog: функциональное описание
условные переходы

```
if(cond) stmt;
else stmt;
```

```
case(a)
  3'b000: stmt;
  3'b010: stmt;
  3'b011: stmt;
  default: stmt;
endcase
```

Условные инструкции тоже можно писать

Как и инструкцию switch-case

Они интерпретируются обычным образом (*примерно как в C++*)

Verilog: параметры

Иногда бывает нужно написать несколько невероятно похожих, но всё же разных модулей

Например:

```
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

module register5(input load, reset, clock,
  input [4:0] in, output reg [4:0] out);
  always @(posedge clock, negedge reset)
    if(~reset) out <= 0;
    else if(~load) out <= in;
endmodule

...
```

Verilog: параметры

Чтобы описать сразу всё разнообразие модулей, отличающихся только какими-то константными значениями (например, регистры — размером шины), достаточно описать один модуль с соответствующими параметрами:

```
module register
  #(parameter Width = 5)
  ( input load, reset, clock,
    input [Width-1:0] in,
    output reg [Width-1:0] out
  );
  always @(posedge clock, negedge reset)
    if(~reset) out <= 0;
    else if(~load) out <= in;
endmodule
```

Verilog: параметры

Чтобы описать сразу всё разнообразие модулей, отличающихся только какими-то константными значениями (например, регистры — размером шины), достаточно описать один модуль с соответствующими параметрами:

Или так:

```
module register(load, reset, clock, in, out);
  parameter Width = 5;
  input load, reset, clock;
  input [Width-1:0] in;
  output reg [Width-1:0] out;

  always @(posedge clock, negedge reset)
    if(~reset) out <= 0;
    else if(~load) out <= in;
endmodule
```

Verilog: параметры

```
parameter Width = 5;
```

Параметр можно писать вместо числа *почти везде* в модуле (**нельзя — в константах на месте размера**)

Значение параметра по умолчанию указывается при его определении (здесь — 5)

Экземпляр параметризованного модуля может быть вызван двумя способами:

- с явным указанием параметров (указание параметров — такое же, как и входов-выходов)

  ```
  register r #(.Width(3)) (<arguments>)
  ```

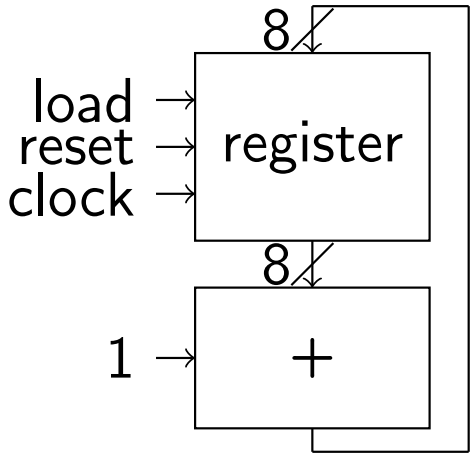
- без указания параметров — тогда подставляется значение по умолчанию

  ```
  register r (<arguments>)
  ```

Часть 2

Управляющие автоматы

Введение: зачем нужен управляющий автомат

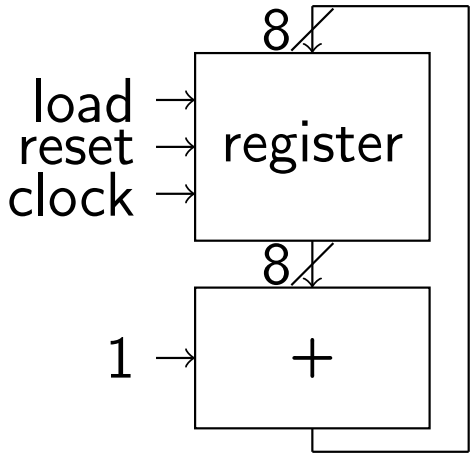


Эта схема определяет то, как преобразуются данные, то есть операционный автомат

В зависимости от того, что подаётся на входные сигналы `load`, `reset`, `clock`, данные, записанные в регистр, могут

- оставаться такими же, как и были
- увеличиваться на единицу
- сбрасываться в ноль

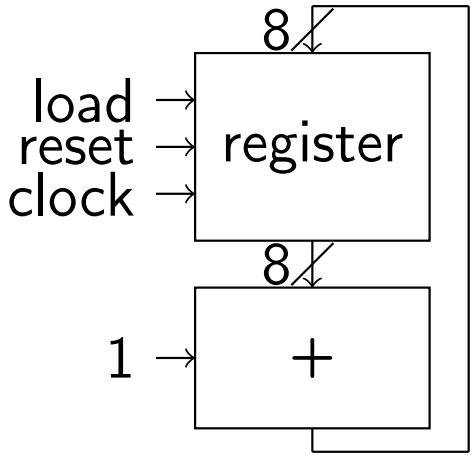
Введение: зачем нужен управляющий автомат



А как заставить регистр делать то, что мы хотим?

1. Подвести ко входам регистра имеющиеся элементы управления (в DE0-Nano — KEY[i], SW[i], CLOCK_50) и управлять регистром, нажимая на кнопки и щёлкая выключателями
2. **Заставить схему делать эту работу за нас**
   - что особенно полезно, если управляющих входов больше, чем элементов управления, а часто без этого в принципе не обойтись

Введение: зачем нужен управляющий автомат



Схема, которая выставляет за нас управляющие сигналы нужным образом в нужные моменты времени — это управляющий автомат

Простенькая задачка

А как разработать подходящий управляющий автомат?

Те, кто выпускались бакалаврами с нашей кафедры: вспоминаем далёкий первый курс, а заодно чуть менее далёкий четвёртый — знания об автоматах-преобразователях с этих курсов очень даже могут пригодиться

Начнём с такой задачки: **регистр должен посчитать числа от ноля до двух и остановиться**

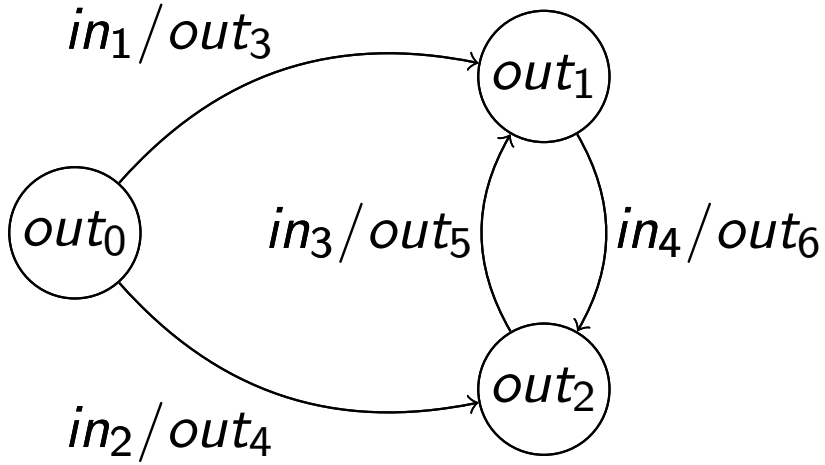
Можно пошагово расписать последовательность действий, которые должен сделать регистр, чтобы эту задачку решить:

1. записать в себя ноль
2. прибавить единицу
3. прибавить единицу
4. остановиться

Автоматы

Что такое автомат?

- У него есть конечное множество состояний
- Общаясь с внешней средой, он переходит из одного состояния в другое *в дискретном времени* (т.е. пошагово)
- В зависимости от текущего состояния, он выдаёт нечто на выход, то есть во внешнюю среду (**автомат Мура**)
- Совершая переход, он также способен выдавать нечто на выход (**автомат Мили**)



Автоматы

1. записать в себя ноль
2. прибавить единицу
3. прибавить единицу
4. остановиться

Попробуем записать этот алгоритм в автоматном виде

Откуда взять дискретное время?

Есть входной провод CLOCK_50, и можно дискретно отсчитывать моменты времени по передним фронтам приходящих от него сигналов

Автоматы

1. записать в себя ноль
2. прибавить единицу
3. прибавить единицу
4. остановиться

Попробуем записать этот алгоритм в автоматном виде

Откуда взять состояния?

Четыре пункта алгоритма — это, по большому счёту, четыре состояния:

- каждый пункт точно описывает, что автомат должен послать во внешнюю среду (*то есть в операционный автомат*)
- каждый пункт может быть сделан за один такт времени

Автоматы

1. записать в себя ноль
2. прибавить единицу
3. прибавить единицу
4. остановиться

Попробуем записать этот алгоритм в автоматном виде

Как соединить между собой эти состояния?

По цепочке от предыдущего к следующему, не обращая внимания на то, что происходит во внешней среде

Автоматы

1. записать в себя ноль
2. прибавить единицу
3. прибавить единицу
4. остановиться

Попробуем записать этот алгоритм в автоматном виде

Что когда выдавать на выход?

При переходе ничего не нужно делать (*в более сложных случаях может понадобиться, но не тут*)

В каждом состоянии достаточно выставить нужные сигналы на входах `load` и `reset` регистра:

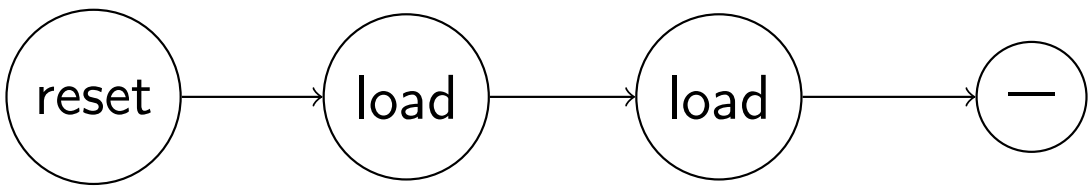
- выставляем `reset = 0` — регистр сбрасывается (*немедленно*)
- выставляем `load = 0` — значение в регистре увеличивается (*по переднему фронту CLOCK_50*)

Автоматы

1. записать в себя ноль
2. прибавить единицу
3. прибавить единицу
4. остановиться

Попробуем записать этот алгоритм в автоматном виде

Диаграмма автомата (диаграмма Мура?) автомата, описывающего алгоритм:

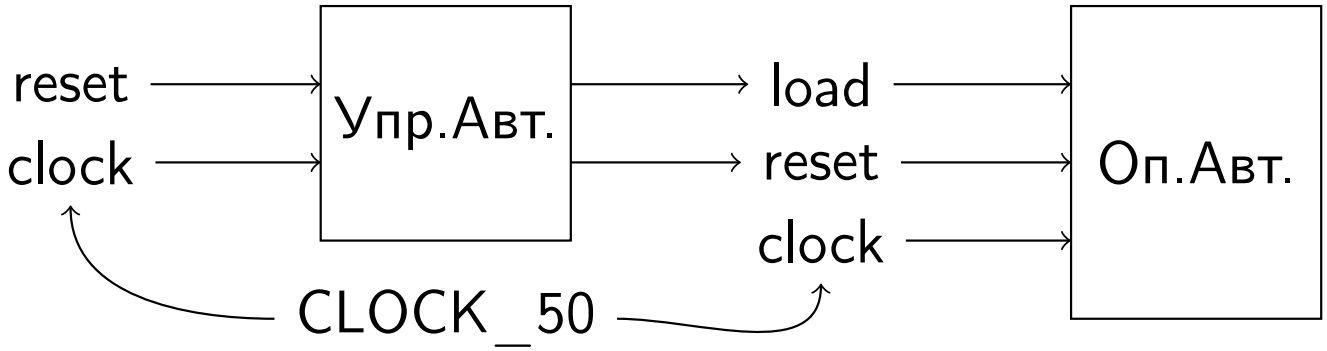


Взаимодействие операционного и управляющего автоматов

И как это всё будет выглядеть “в железе”?

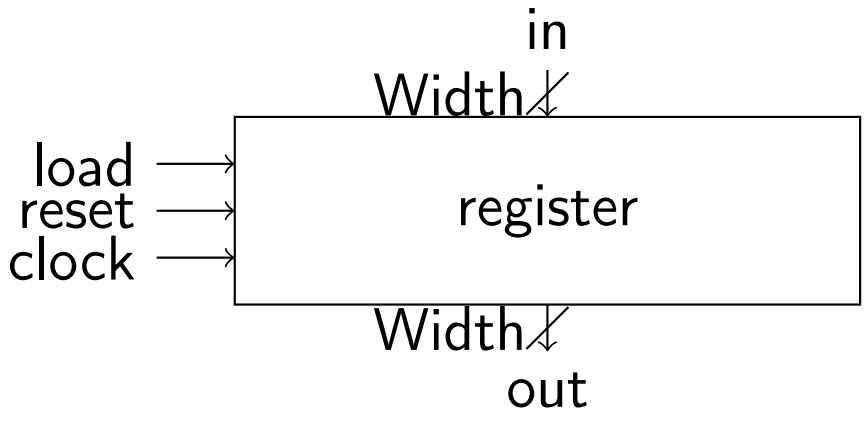
Чтобы всё заработало, достаточно заиметь и верно соединить:

- операционный автомат: коробочку, в которой всё работает однозначно, кроме сигналов `load`, `reset`, `clock`
- управляющий автомат:
  - для нормальной работы требуется *дискретное время* (clock) и инициализация (`reset`)
  - основное назначение — в нужное время в нужном порядке выставлять сигналы `load`, `reset`
- тактовый генератор CLOCK_50



Решение задачки

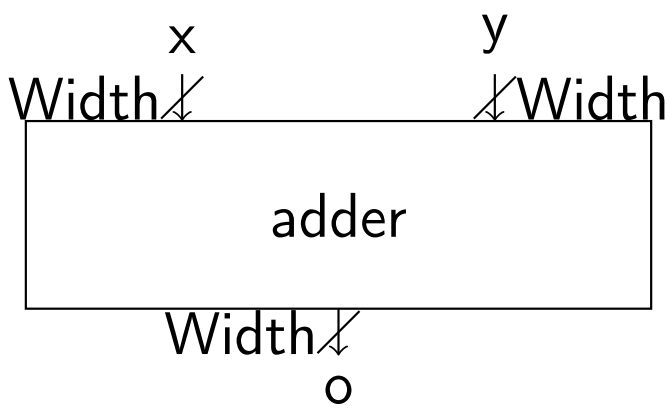
Регистр:




```
module register
  #(parameter Width = 8)
  ( input [Width-1:0] in,
    output reg [Width-1:0] out,
   input load, reset, clock
  );
  always @(posedge clock, negedge reset)
    if(~reset) out <= 0;
   else if(~load) out <= in;
endmodule
```

Решение задачки

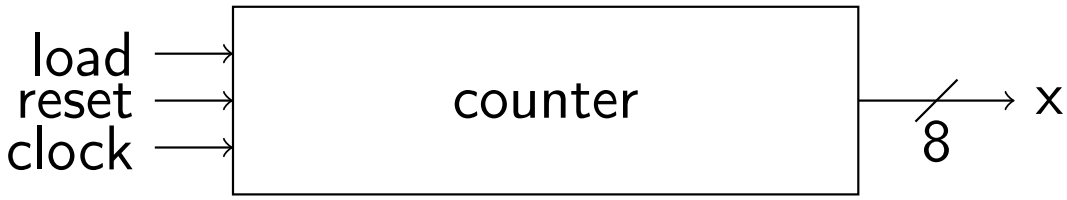
Сумматор:




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

Решение задачки

Операционный автомат (с выводом значения регистра в LED):




```
module counter(input load, reset, clock,
               output [7:0] x);
  parameter Width = 8;
  wire [Width-1:0] in, out;

  register r(.in(in), .out(out), .load(load),
             .reset(reset), .clock(clock));
  adder a(.x(out), .y(8'b00000001), .o(in));

  assign x = out;
endmodule
```

Решение задачки

Управляющий автомат:


```
module fsm(input clock, reset,
           output reg load, o_reset);
  reg [1:0] c_state, n_state;

  always @(c_state)
    case(c_state)
     2'b00:
    begin
      load = 1;
      o_reset = 0;
      n_state = 2'b01;
    end
```

Решение задачки

Управляющий автомат:


```
 2'b01:
begin
   load = 0;
   o_reset = 1;
   n_state = 2'b10;
end
 2'b10:
begin
   load = 0;
   o_reset = 1;
   n_state = 2'b11;
end
```

Решение задачки

Управляющий автомат:


```
      2'b11:
     begin
       load = 1;
       o_reset = 1;
       n_state = 2'b11;
     end
   endcase

  always @(posedge clock, negedge reset)
    if(~reset) c_state <= 0;
    else c_state <= n_state;
endmodule
```

Решение задачки

Главный модуль (reset выведен на KEY[1], и мы, нажимая на KEY[0], генерируем тактовые импульсы):

```
module top( SW, KEY, LED, CLOCK_50);
  input wire [3:0] SW;
  input wire [1:0] KEY;
  output [7:0] LED;
  input wire CLOCK_50;

  wire load, reset;
  counter op_aut(.load(load), .reset(reset),
                 .clock(KEY[0]), .x(LED));
  fsm c_aut(.clock(KEY[0]), .reset(KEY[1]),
            .load(load), .o_reset(reset));
endmodule
```

А теперь задачка посложнее

Хочу, чтобы счётчик работал так:

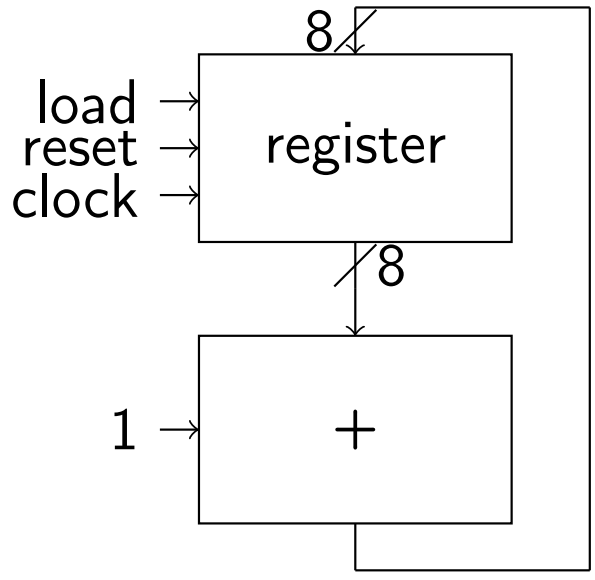
- выключателями SW составляю двоичную четырёхбитную запись числа
- кнопкой KEY[1] запускаю алгоритм
- счётчик отсчитывает с ноля до составленного числа, прибавляет единицу и останавливается

В чём здесь сложности?

1. Диаграмма Мура нелинейна (*есть циклы*)
2. Управляющий автомат, чтобы знать, что делать, должен анализировать информацию из внешнего мира
3. Передавать **данные** в управляющий автомат — плохо (*управляющий автомат должен* ***управлять***, *а не вычислять*)
4. Значит, нужно добавить в операционный автомат схему, работающую с данными (*проверяющую, досчитал ли регистр до конца*) и передающую результат работы в управляющий автомат

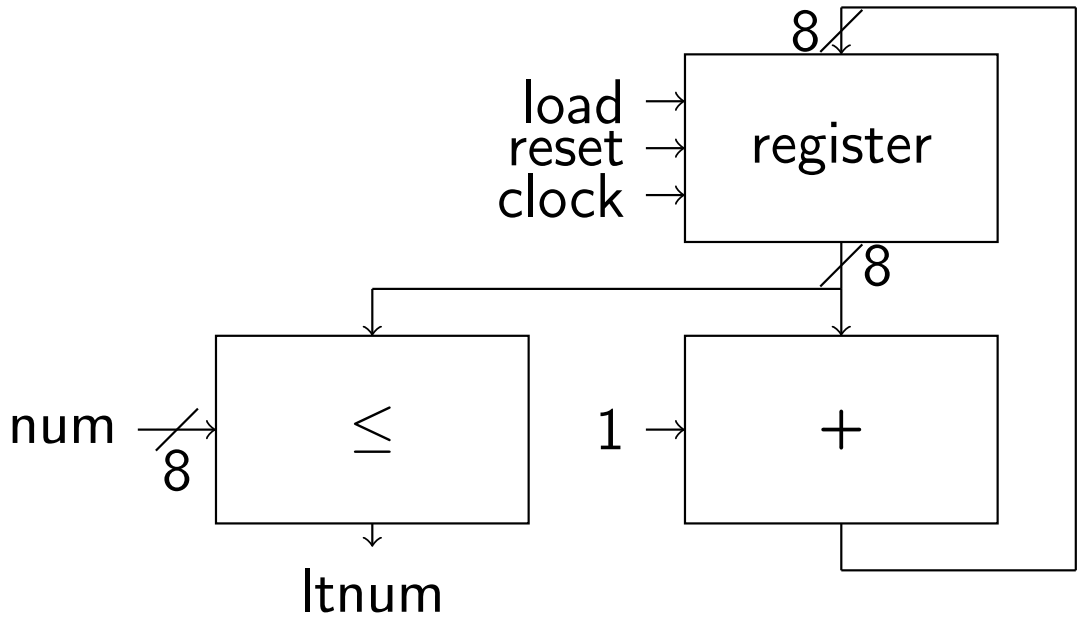
Как изменится операционный автомат

Было:



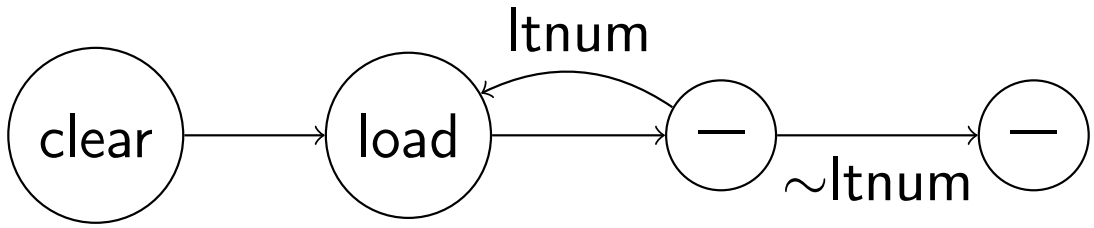
Как изменится операционный автомат

Стало:



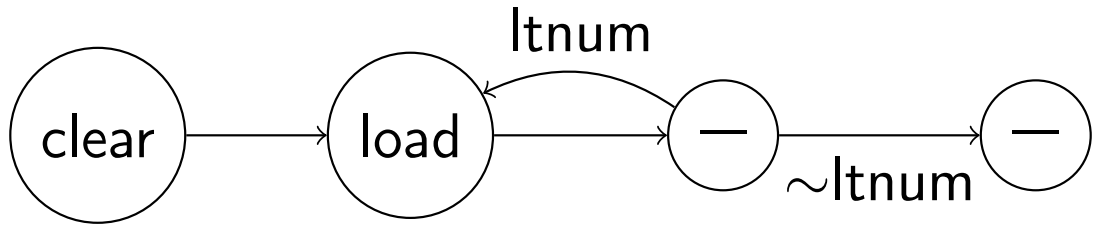
- Добавился блок сравнения
- Добавилась входная шина `num`
- Добавился выходной сигнал `ltnum` — он будет пересылаться управляющему автомату

Как будет выглядеть управляющий автомат



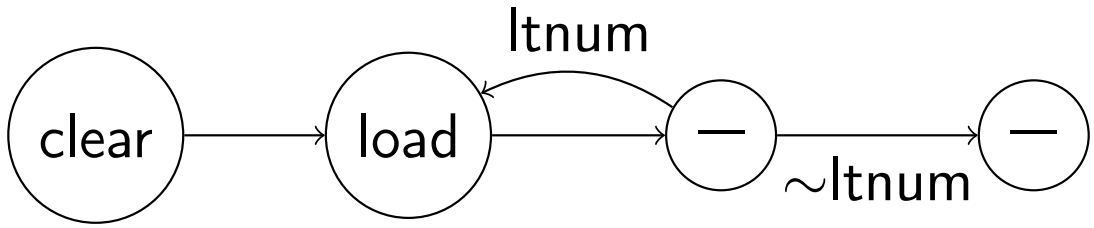

```
module fsm(input clock, reset, ltnum,
           output reg load, o_reset);
...
```

Как будет выглядеть управляющий автомат



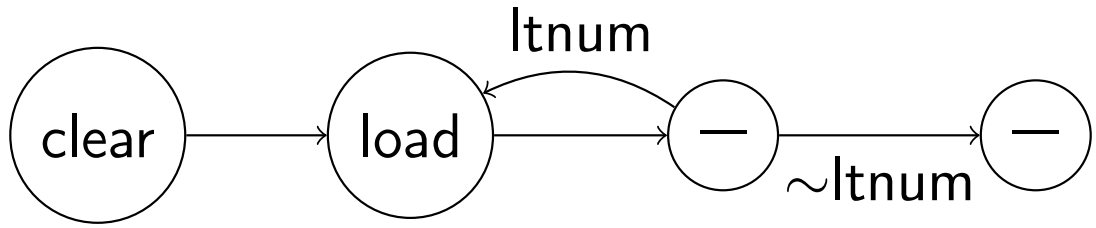

```
...
    2'b00:
    begin
      o_reset = 0;
      load = 1;
      n_state = 2'b01;
    end
...
```

Как будет выглядеть управляющий автомат



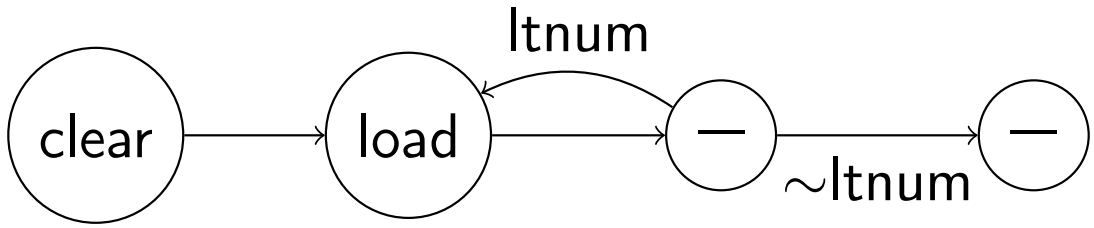

```
...
    2'b01:
    begin
      o_reset = 1;
      load = 0;
      n_state = 2'b10;
    end
...
```

Как будет выглядеть управляющий автомат



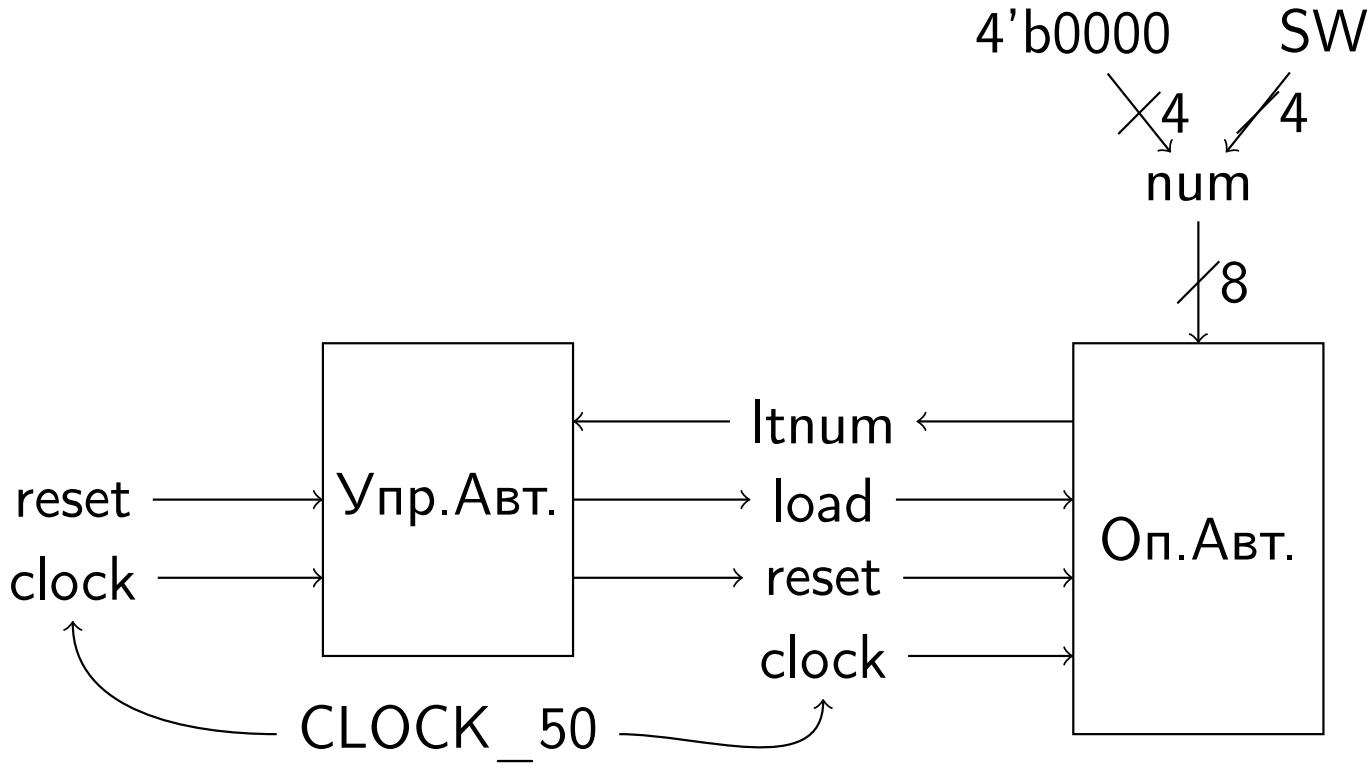

```
...
    2'b10:
    begin
      o_reset = 1;
      load = 1;
      if(ltnum) n_state = 2'b01;
      else n_state = 2'b11;
    end
...
```

Как будет выглядеть управляющий автомат




```
...
    2'b11:
    begin
      o_reset = 1;
      load = 1;
      n_state = 2'b11;
    end
...
```

Как будет выглядеть взаимодействие управляющего и операционного автоматов



А остальную часть решения додумайте сами

Конец лекции 1