Лабораторная работа 2	Группа 05	2023
Построение схем	Ратьков Андрей Игоревич	

1 Инструментарий, описание, ссылки

1.1 Инструментарий

1. Программа Logisim-evolution v3.8.0
2. Язык System Verilog (все программы на нём я запускал только с помощью команды `iverilog -g2012` (то есть, видимо, пользовался стандартом языка 2012))
3. Typst для написания отчёта ♡

1.2 Описание

Нужно собрать схему “Синхронный стек” на 5 ячейках, каждая хранит 4 бита информации. Можно делать pop, push, get. Нужно сначала смоделировать в logisim, а потом описать на verilog двумя способами — на транзисторах и без.

1.3 Ссылка на репозиторий и краткое изложение того, что я сделал

https://github.com/skkv-mkn/mkn-comp-arch-2023-circuit-AndrewRatkov

Я выполнял только задачи типа normal: logisim normal, stack_structural_normal, stack_behaviour_normal.

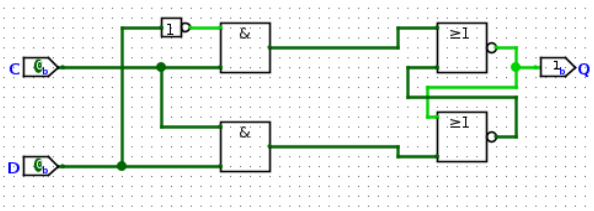
Далее, в главе 2 описано, как я делал схему в logisim, в главе 3 написано, как я писал структурную реализацию на System Verilog, в главе 4 чуть-чуть про поведенческую реализацию, в главе 5 — тестирование (всех реализаций).

2 Как вообще работает схема (сразу с описание в logisim evolution)

2.1 Довольно очевидные вещи

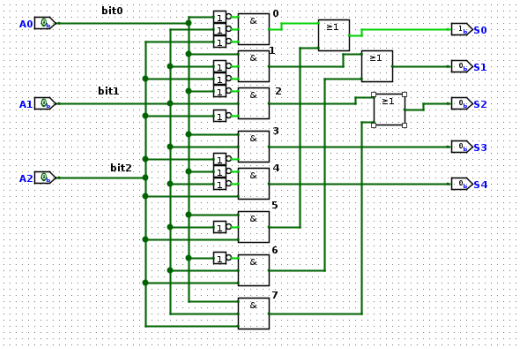
2.1.1 Подсхема D-trigger

Просто реализованный D-триггер. На входе он имеет 2 входа: C (синхронизация) и D (запиываемый бит). Когда $C = 0$, ничего не происходит, D-триггер просто продолжает хранить значение, которое в него когда-то записали. Когда $C = 1$, D-триггер сохраняет значение, которое ему подали на входе D. В общем, это обычная ячейка памяти для хранения 1 бита информации. Из них и будет состоять стек. Выход Q — сохранённый бит.



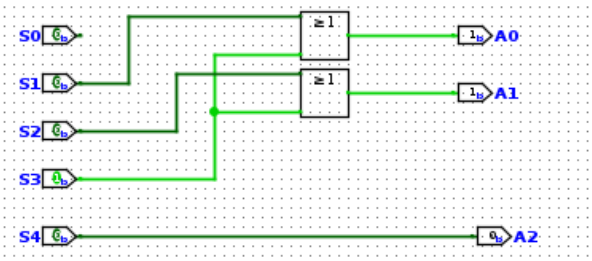
2.1.2 Подсхема ThreeToFive

Схема, которая на вход принимает три провода A_0, A_1, A_2 — закодированное число $\overline{A_2A_1A_0}_2$. Выводится число $\overline{A_2A_1A_0}_2$ по модулю 5 (то есть, если $\overline{A_2A_1A_0}_2 = j, j \in \{0,1,2,3,4\}$, то на выходе S_j будет 1, а на остальных 0)



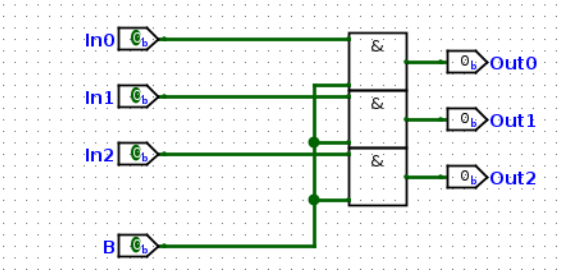
2.1.3 Подсхема FiveToThree

Получает на вход 5 проводов $S_0, S_1, ..., S_4$. Подразумевается, ровно на одном из них 1, на остальных 0. На выходе 3 провода A_0, A_1, A_2. Пусть $S_j = 1$. Тогда на A_0, A_1, A_2 значения будут такими, что $\overline{A_2A_1A_0}_2 = j$.



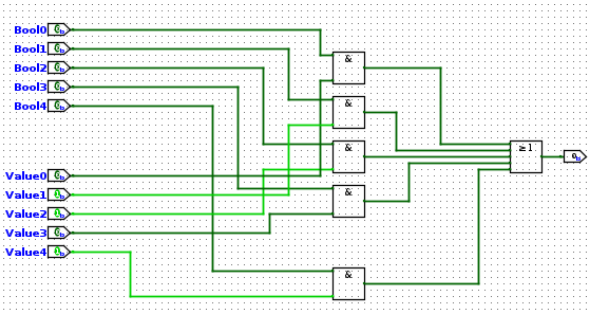
2.1.4 Подсхема ThreePusher

На входе имеет три входа — провода $\text{In}_0, \text{In}_1, \text{In}_2$ и Bool, На выходе — $\text{Out}_0, \text{Out}_1, \text{Out}_2$. Когда Bool = 1, схема пропускает провода A_0, A_1, A_2, то есть $\text{Out}_0 = \text{In}_0, \text{Out}_1 = \text{In}_1, \text{Out}_2 = \text{In}_2$. Иначе не пропускает — на каждом из выводов будет 0.



2.1.5 Подсхема Comp10to1

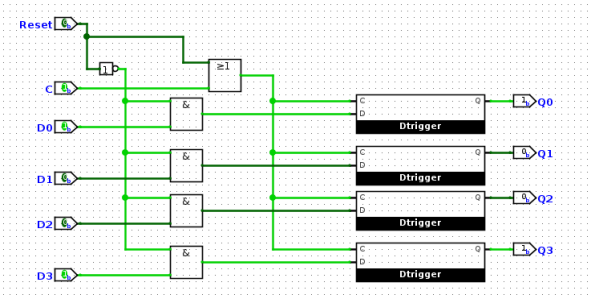
Получает на вход 5 проводов $\text{Bool}_0, \text{Bool}_1, ..., \text{Bool}_4$ и 5 провдов $\text{Value}_0, \text{Value}_1, ..., \text{Value}_4$. Подразумевается, ровно на одном из первых пяти проводов 1, на остальных из первых пяти 0. На выходе есть единственный провод. Пусть 1 на каком-то проводе Bool_j. Тогда значение на выходе будет равно Value_j.



2.2 Более важные вещи

2.2.1 Подсхема StackCell

Одна ячейка стека, хранящая 4 бита Q_0, Q_1, Q_2, Q_3 в четырёх D-триггерах. Имеет вход Reset — когда его включают (т.е. его значение равно 1), внутри всё зануляется. Также имеет входы C — синхронизацию и D_0, D_1, D_2, D_3 — биты, которые мы можем захотеть записать. Если $C = 1$, значения D_0, D_1, D_2, D_3 записываются соответственно в 0,1,2 и 3 D-триггеры, иначе ничего не происходит (что хранили, то и продолжаем хранить).



2.2.2 Подсхема MUX

Штука, которая читает биты из какой-то ячейки стека. На вход принимает следующие провода: A_0, A_1, A_2 — они кодируют номер ячекйи, из которой будем читать ($\overline{A_2 A_1 A_0}_2 \bmod 5$) и $D_0, D_1, ..., D_{19}$. D_0, D_1, D_2, D_3 — биты 0ой ячейки (от младшего к старшему), D_4, D_5, D_6, D_7 — биты 1ой ячейки (от младшего к старшему), и так далее, $D_{16}, D_{17}, D_{18}, D_{19}$ — биты 4ой ячейки (от младшего к старшему). С помощью ThreeToFive мы парсим A_0, A_1, A_2 в индекс ячейки, из которой будем читать (назовём его idx). Тогда в Q_0 мы запишем то, что лежит в $D_{4 \cdot \text{ idx } + 0}$, в Q_1 — $D_{4 \cdot \text{ idx } + 1}$, в Q_2 — $D_{4 \cdot \text{ idx } + 2}$, в Q_3 — $D_{4 \cdot \text{ idx } + 3}$.

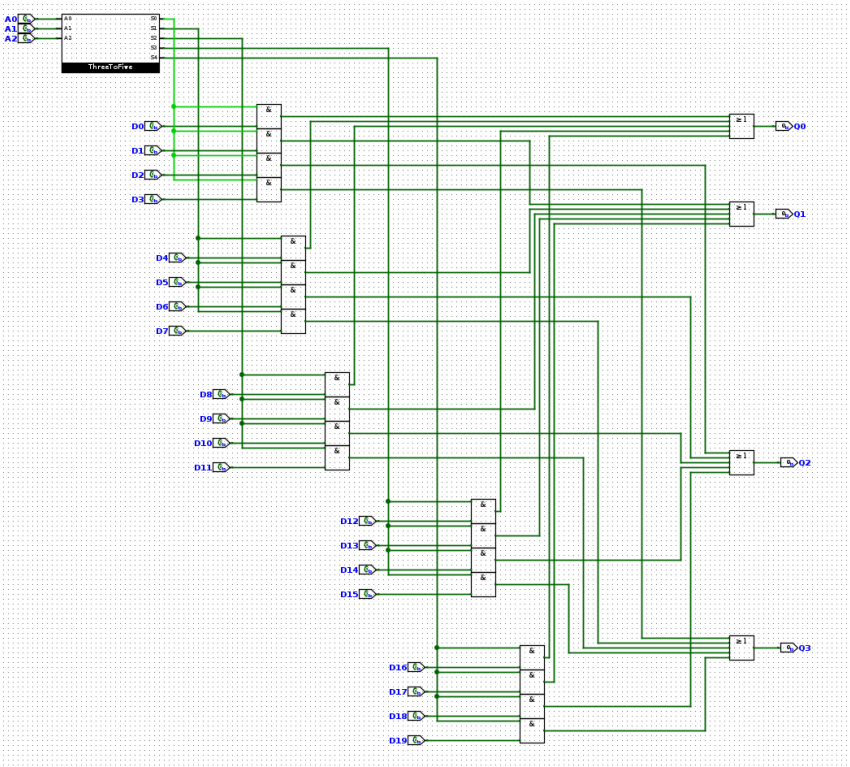
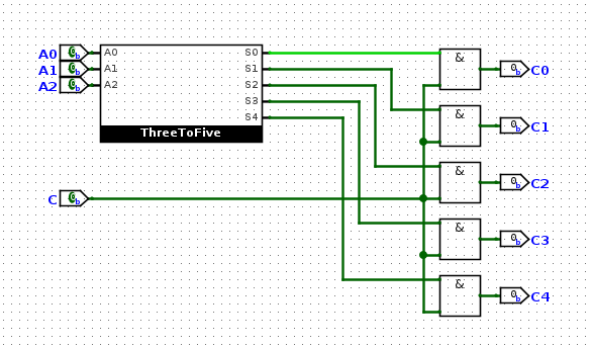


Figure 1: MUX

2.2.3 Подсхема DEMUX

Иногда (а именно при push) нужно писать что-то в какую-то ячейку, когда синхронизация равна 1. У нас ячеек много и в каждую входит свой провод для синхронизации (см. StackCell). А поскольку мы будем записывать ровно в одну ячейку, нам нужно, чтобы входящий в неё провод синхронизации был 1, а в остальные 0.

Так вот, на вход у нас есть провода A_0, A_1, A_2, кодирующие номер $\left(\overline{A_2A_1A_0}_2 \bmod 5\right) \in \{0, ..., 4\}$ ячейки, куда мы будем писать, и провод синхронизации C. Выходящие провода — $C_0, ..., C_4$ — провода синхронизации для ячеек $0, ..., 4$ соответственно. Они всегда все (кроме, ровно, одного, с номером $\overline{A_2A_1A_0}_2 \bmod 5$, когда $C = 1$) равны 0.



2.2.4 Подсхема MEM

Входы A_0, A_1, A_2 — код ячейки, с которой мы работаем ($\overline{A_2A_1A_0}_2 (\text{mod } 5) \in \{0, ..., 4\}$); R_W — читаем мы (0) или пишем(1); C — синхронизация, D_0, D_1, D_2, D_3 — биты (от младшего к старшему) которые мы можем захотеть записать в какую-нибудь ячейку, Reset — если это равно 1, то вся память очищается (заполняется нулями).

Выходы — Q_0, Q_1, Q_2, Q_3 — содержимое ячейки, с которой мы работаем (если мы писали что-то в ячейку, то это ровно D_0, D_1, D_2, D_3, которые мы в неё записали).

Используется 5 подсхем StackCell (для пяти ячеек), DEMUX (чтобы, если мы записываем, то есть если $\text{R\_M} = 1 \wedge C = 1$, значение 1 синхронизации попало только в нужную ячейку с номером $\overline{A_2A_1A_0}_2$), MUX (для чтения)

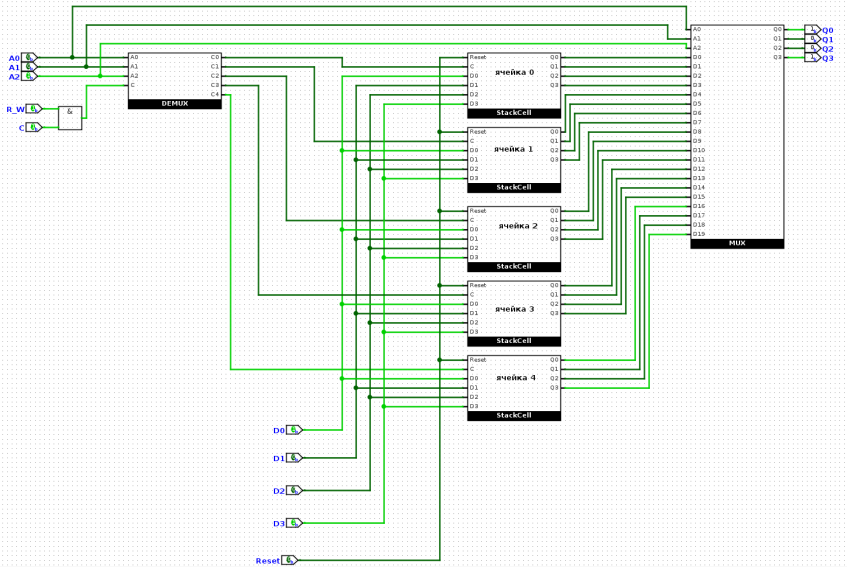


Figure 2: MEM

2.3 Арифметические вещи

Зачем вообще это нужно? Мы будем хранить указатель на ячейку **следующую** за верхней заполненной ячейкой стека. От нас могут попросить сделать pop или push — после этого надо соответственно уменьшить или увеличить значение указателя на 1 (по модулю 5). А ещё к нам может поступить запрос сделать get(x). Тогда нужно вывести значение, которое лежит на в ячейке, которая на $x + 1 (\text{mod } 5)$ левее ячейки, на которую смотрит указатель.

2.3.1 Подсхема Plus1

Нам на вход поступают три провода A_0, A_1, A_2, кодирующие число $\overline{A_2A_1A_0}_2 (\text{mod } 5)$. На выводе даём три провода B_0, B_1, B_2 такие что число $\overline{B_2B_1B_0}_2$ таково, что $\overline{B_2B_1B_0}_2 = \overline{A_2A_1A_0}_2 + 1 (\text{mod } 5)$. Это легко делается с помощью ThreeToFive и FiveToThree:

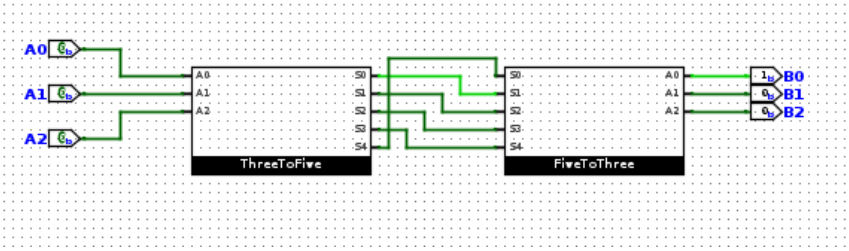


Figure 3: Plus1

2.3.2 Подсхема Minus1

Нам на вход поступают три провода A_0, A_1, A_2, кодирующие число $\overline{A_2A_1A_0}_2 (\mathrm{mod}\ 5)$. На выводе даём три провода B_0, B_1, B_2 такие что число $\overline{B_2B_1B_0}_2$ таково, что $\overline{B_2B_1B_0}_2 = \overline{A_2A_1A_0}_2 - 1 (\mathrm{mod}\ 5)$. Это тоже легко делается с помощью ThreeToFive и FiveToThree:

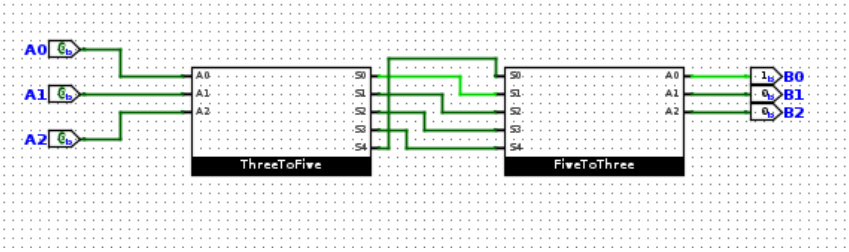


Figure 4: Minus1

2.3.3 Подсхема MinusMod5

На вход даются два числа $\overline{A_2A_1A_0}_2$ и $\overline{B_2B_1B_0}_2$, закодированные в шести проводах $A_0, A_1, A_2, B_0, B_1, B_2$. Выводы R_0, R_1, R_2 таковы, что $\overline{R_2R_1R_0}_2$ — число от 0 до 4 такое что $\overline{R_2R_1R_0}_2 = \overline{A_2A_1A_0}_2 - \overline{B_2B_1B_0}_2 (\mathrm{mod}\ 5)$. Работает оно так: для начала помощью последовательного применения ThreeToFive и FiveToThree на проводах A_0, A_1, A_2 делается то же значение по моделю 5, только от 0 до 4 (если на них значение хотя бы 5, то просто вычитается 5). Значения на B_0, B_1, B_2 перекодировываются в число от 0 до 4 с помощью ThreeToFive. Далее, с помощью четырёх подсхем Minus1, мы умеем получать значения этого числа, уменьшенного на 0, 1 ,2 ,3 и 4. И ровно одно из этих значений нам надо вывести. Для этого используются три элемента Comp10to1 (Потому что у нас 3 бита выодных данных).

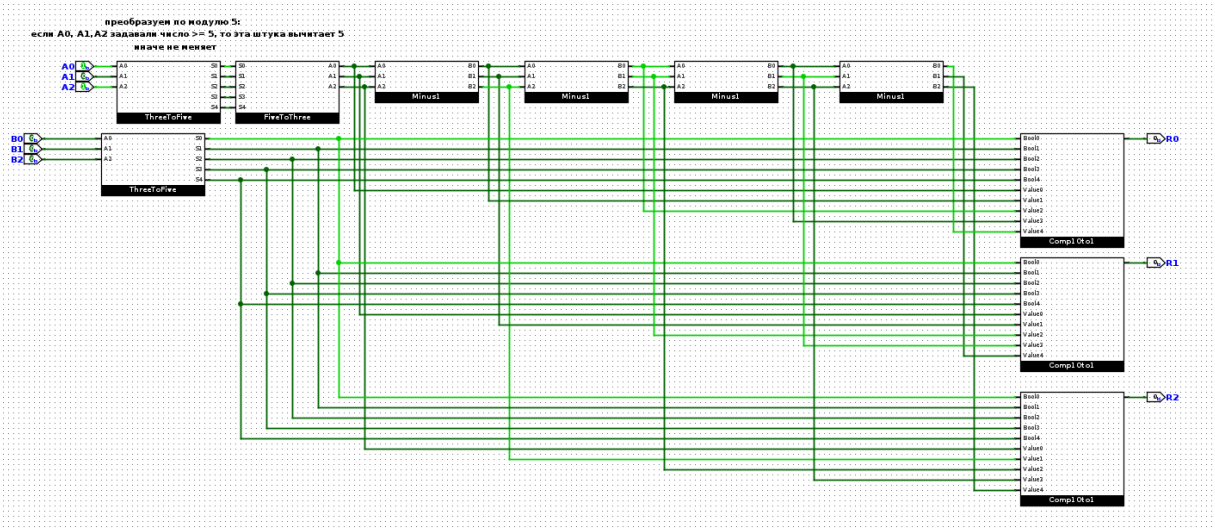


Figure 5: MinusMod5

2.3.4 Подсхема PointerModifier

Итак, у нас есть указатель. После того, как мы делаем pop, его значение должно уменьшаться по модулю 5, после того, как мы делем push — увеличиваться по модулю 5, когда делаем nop или get — не меняться, а когда происходит reset, его значение будет становиться 0. Для того, чтобы менять значение указателя, нам нужна эта схема.

Тип изменения состояния задаётся через 2 входных провода inf1 и inf2:

inf1	0	0	1	1
inf2	0	1	0	1
тип изменения состояния	зануление	+	–	сохранение

Также на входе подаются три провода, A_0, A_1, A_2 кодирующие указатель. Выходы — B_0, B_1, B_2 — кодируют новое значение указателя ($\overline{B_2 B_1 B_0}_2$). Выглядит эта схема так:

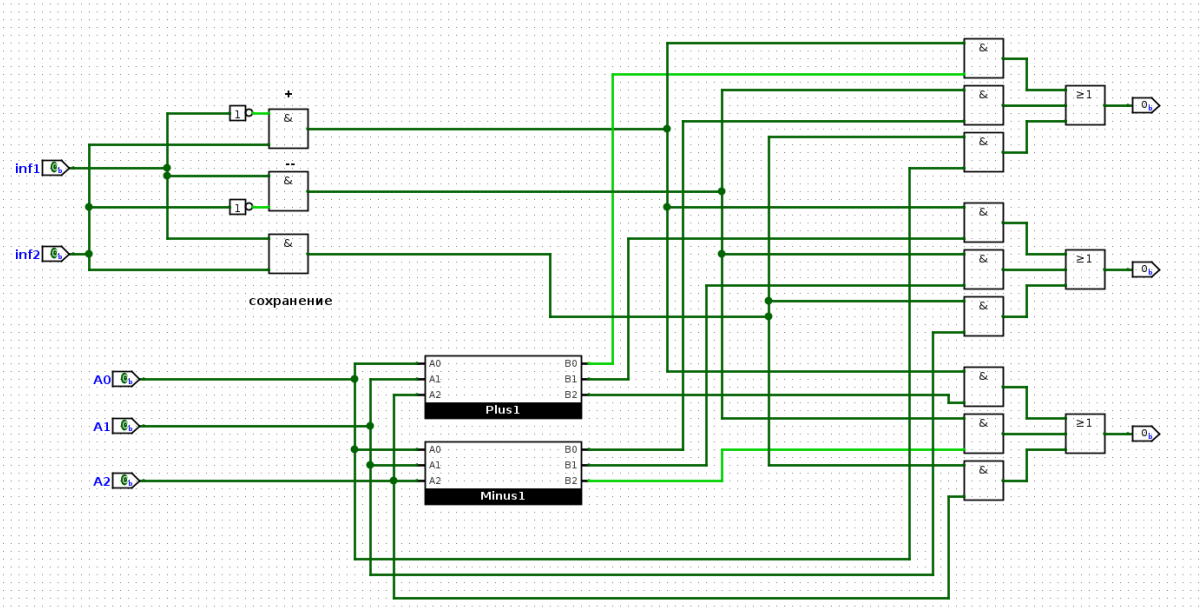


Figure 6: PointerModifier

2.3.5 Подсхема PointerManager

Подсхема, работающая с указателем. Она хранит его последнее сохранённое значение указателя (в правых трёх D-триггерах), исходя из него и поданных на вход параметров inf1 и inf2 генерирует новое значение (и когда синхронизация $C = 1$ записывает его в три левых D-триггера), когда синхронизация отключается ($C = 0$), новое значение перезаписывается в старое.

Итак, входы: inf1 и inf2 — кодируют, как будет меняться указатель (как в предыдущем пункте), C — синхронизация, reset. Когда reset = 1, состояние обнуляется (во все D-триггеры записываем 0).

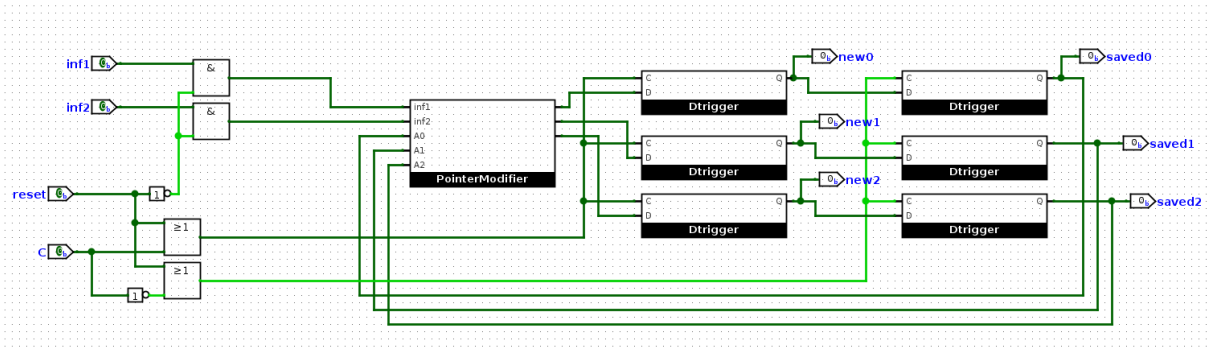


Figure 7: PointerManager

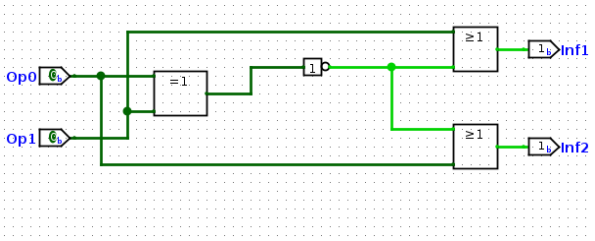
2.3.6 Подсхема OpToInf

На входе нам будут задавать какую-то операцию, закодированную двумя битами (назовём их Op0 и Op1).

Op0	0	1	0	1
Op1	0	0	1	1
Операция	nop (0)	push (1)	pop (2)	get (3)

Когда у нас nop или get, значение указателя надо сохранить (не менять) — то есть на вход в PointerManager нужно подать inf1 = 1, inf2 = 1. Когда у нас pop, значение указателя нужно будет уменьшить на 1 — то есть PointerManager получит на вход inf1 = 1, inf2 = 0. Когда у нас push значение указателя нужно будет увеличить на 1 — то есть PointerManager получит на вход inf1 = 0, inf2 = 1.

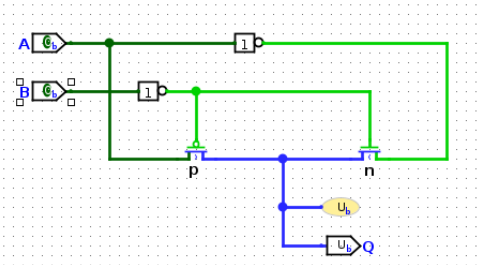
Когда у нас pop, значение указателя нужно будет уменьшить на 1 — то есть PointerManager получит на вход inf1 = 0, inf2 = 1. Когда у нас push, значение указателя нужно будет увеличить на 1 — то есть PointerManager получит на вход inf1 = 1, inf2 = 0.



2.4 Собираем всё, что сделали, в схему

2.4.1 Подсхема InAndOutHelper

Полезная штука, которая нам понадобится для входо-выходов. Она имеет два входа A и B и один выход Q. Когда A = 0, на выходе неизвестное (высокоимпедансное) значение. Когда A = 1, значение на выходе равно B. Состоит из двух отрицаний и транзисторов p-типа и n-типа.



2.4.2 Подсхема stack

Имеет следующие входы:

1. Op_0, Op_1 — они кодируют информацию об операции над стеков (см. выше)
2. Reset — для сигнала сброса: когда он равен 1, всё сбрасывается
3. C — синхронизация

4. G_0, G_1, G_2 — параметры для get: они задают номер ячейки $\overline{G_2 G_1 G_0}_2$ считая сверху от стека, которую мы и будем читать
5. D_0, D_1, D_2, D_3 — биты, которые мы можем захотеть положить на вершину стека

И выходы Q_0, Q_1, Q_2, Q_3 — прочитанные биты, если он что-то читал.
Как оно работает?
Во-первых, Op_0 и Op_1 переводятся с помощью элемента OpToInf в два провода, которые передаются в элемент PointerManager, чтобы он понимал, какие преобразования с указателем делать. Также в PointerManager идут Reset и C.

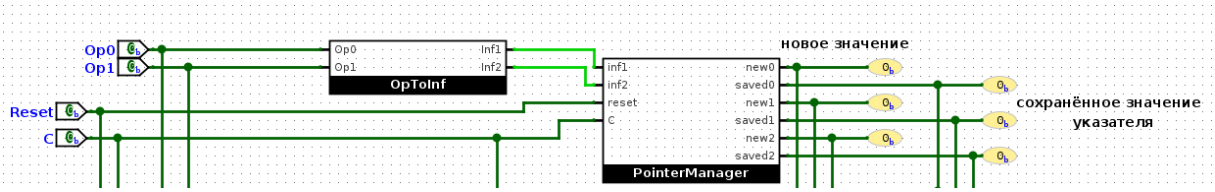
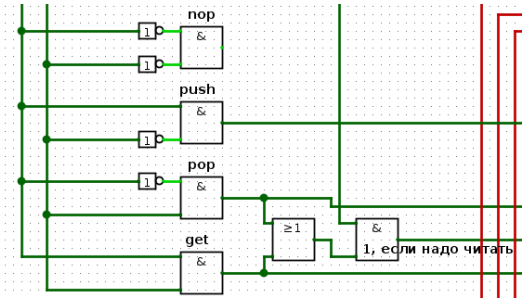


Figure 8: stack, верхняя часть

Из элемента PointerManager выходят шесть проводов: 3 из них — сохранённое (текущее значение указателя), другие 3 — новое (которое отличается от сохранённого при $C = 1$, см. выше описание PointerManager).
Эта часть схемы определяет, какой тип операции задал нам пользователь.
Эта часть схемы определяет, какой тип операции задал нам пользователь (nop/push/pop/get). Если у нас pop/get, то нам нужно будет что-то прочитать при $C = 1$, поэтому у нас появляется соответствующий провод, идущий в правую часть схемы



Для get предусмотрены 3 входа G_0, G_1, G_2. Когда пользователь запрашивает get (пускай x = $\overline{G_2 G_1 G_0}$), ему нужно получить, что лежит в x-ой ячейке с верху стека, то есть в ячейке P − x − 1 (mod 5), где P — ячейка, на которую указывает указатель. Чтобы понять номер ячейки, из которой нам надо читать, сделана вот эта часть схемы (тут красные провода, появляющиеся сверху, кодируют ячейку, на которую смотрит указатель):

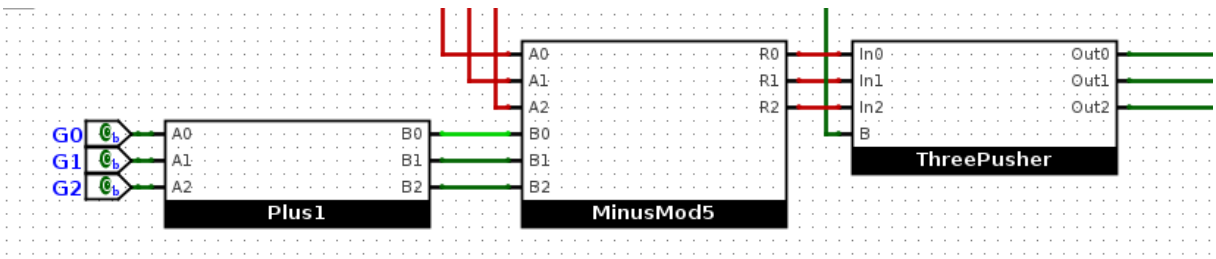


Figure 9: stack, нижняя часть, обработка адреса в get

Какой адрес ячейки нужно отправить в MEM? Ну, если у нас get, то тот, что описан только что выше. Когда nop — вообще ничего можно не отправлять, раз у нас нет операции. Когда push — сохранённое значение указателя (потому что оно, по определению, и указывает на ячейку, куда будем пушить). А вот когда у нас pop, в MEM нужно отправить **новое** значение указателя (то есть сохранённое минус 1) — это в точности адрес ячейки, из которой мы pop-аем.

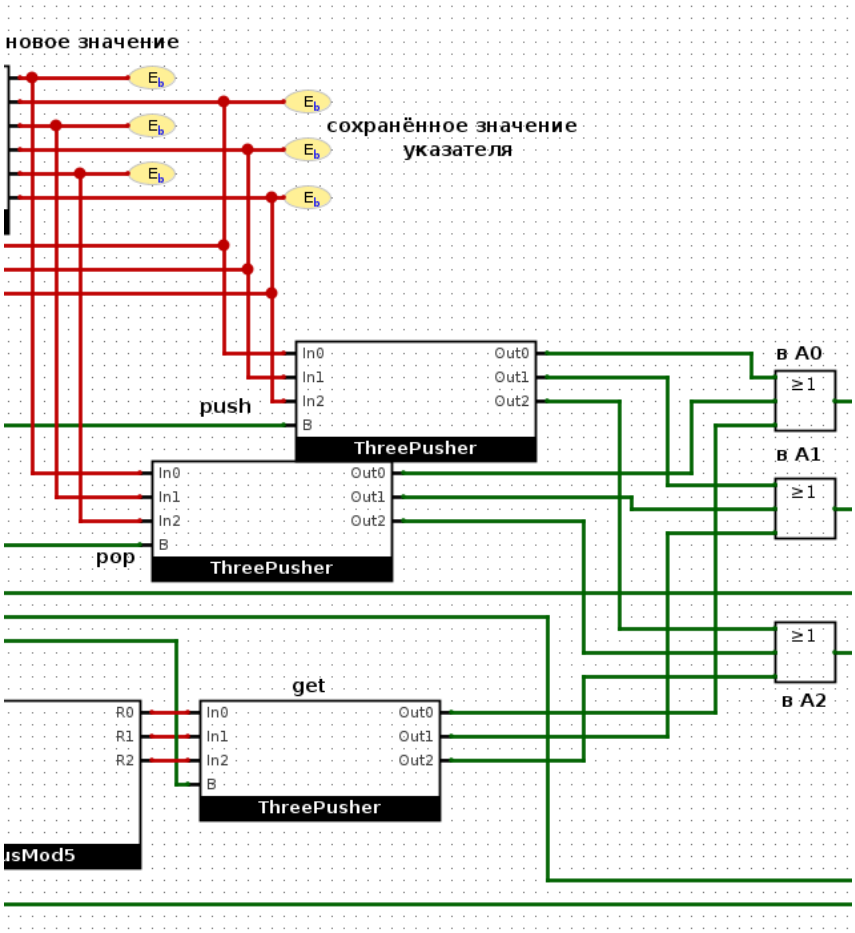


Figure 10: stack, получаем адрес ячейки, с которой работаем

Ну, и самая главная часть схемы — MEM и выходы.

В MEM входят адрес ячейки, из которой надо читать или писать, закодированный в проводах: A_0, A_1, A_2. Так же входит булево значение R_W, пишем мы или читаем (описано ранее). Также есть вход для синхронизации C и Reset. И D_0, D_1, D_2, D_3 — то, что мы кладём в ячейку (если пушаем). Выходы из MEM нам не всегда нужно выводить пользователю, а только при pop и get. Поэтому в выходы схемы Q_0, Q_1, Q_2, Q_3 что-то пойдёт, только когда мы что-то читаем. Для этого нажны 4 подсхемы InAndOutHelper.

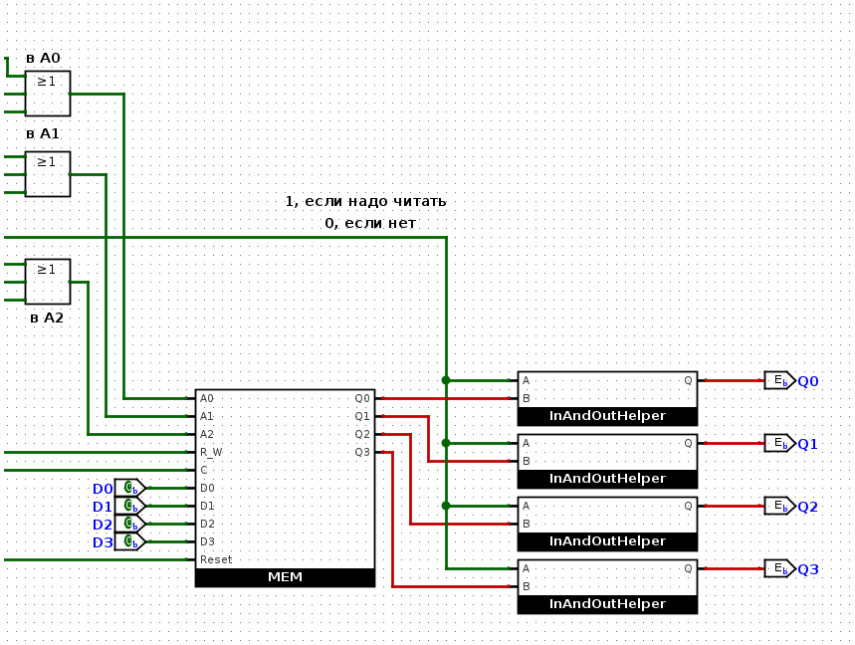


Figure 11: stack, MEM и выход

Всё, схему stack описали!

2.4.3 Схема main

Просто к схеме stack присобачиваем входы и входо-выходы:

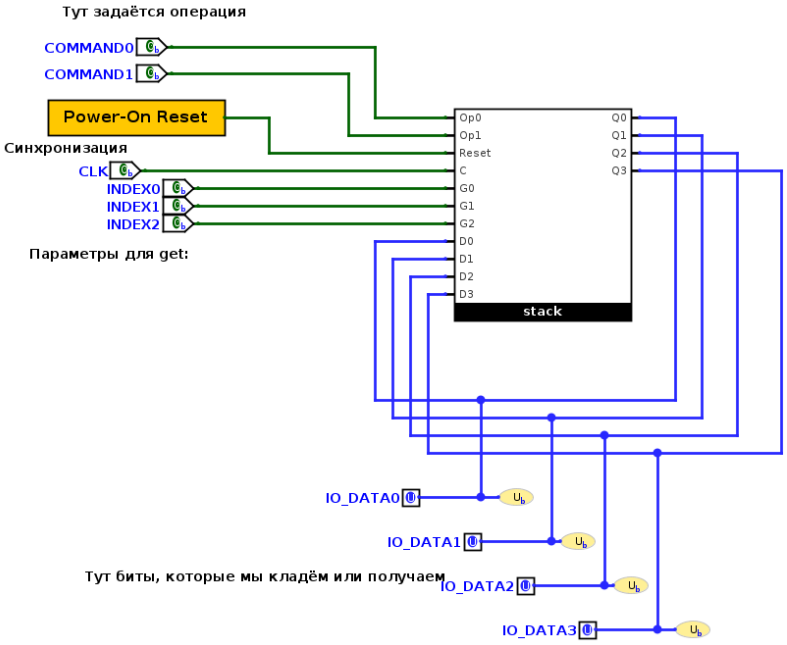
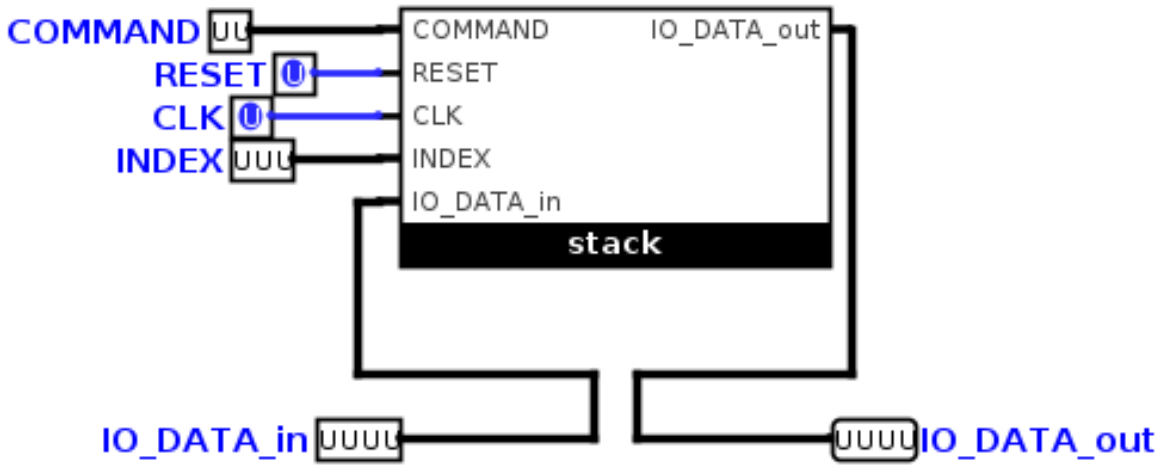


Figure 12: main

2.5 Важное замечание

Я поменял входы и выходы в схеме, чтобы они выглядели как в шаблоне. Тестирование— в главе 5. В итоге main выглядит так:



3 Пишем на языке SystemVerilog структурную модель стека (stack_structural)

Как и в предыдущей главе, будем сначала реализовывать какие-то базовые штуки, из которых постепенно будем строить всю схему. Каждую штуку я проверял на каких-то тестах, результаты работы модуля на соответствующих ему тестах я писал в табличку, когда мне было не лень (если, конечно, модуль не совсем очевиный и для него мало тестов). А вообще к каждой подсхеме (модулю) я писал свои тесты, иногда перебирая всевозможные случаи (если их немного).

Где-то, где мне хотелось, я вставлял код. Но это скучно, монотонно, и, кажется, бесполезно, поэтому этого немного (это было просто тупое перефигачивание схемы из logisim в verilog).

3.1 Довольно очевидные вещи

3.1.1 D-trigger

```
module DTrigger(input wire C, D, output wire Q);
    wire w1, w2, notQ, notD;
    not ND(notD, D);

    and IN1(w1, C, notD);
    and IN2(w2, C, D);
    nor NOR1(notQ, Q, w2);
    nor NOR2(Q, notQ, w1);
endmodule
```

Команды в тесте идут слева направо:

C	0	1	1	0	0	1	1
D	0	0	1	1	0	0	1
Q	x	0	1	1	1	0	1

Модуль с тестами для этой подсхемы называется `test_DTrigger`.

3.1.2 ThreeToFive

Модуль с тестами для этой подсхемы называется `test_ThreeToFive`. Там перебраны всевоможные входные данные.

3.1.3 FiveToThree

Модуль с тестами для этой подсхемы называется `test_FiveToThree`. Там перебраны всевоможные входные данные.

3.1.4 ThreePusher

Модуль с тестами для этой подсхемы называется `test_Three_pusher`. Там перебраны всевоможные входные данные.

3.1.5 Comp10to1

```
module Comp10to1(input wire B0, B1, B2, B3, B4, V0, V1, V2, V3, V4, output wire OUT);
    wire[4:0] PUSH;
    and PV0(PUSH[0], B0, V0);
    and PV1(PUSH[1], B1, V1);
    and PV2(PUSH[2], B2, V2);
    and PV3(PUSH[3], B3, V3);
    and PV4(PUSH[4], B4, V4);
    or out(OUT, PUSH[0], PUSH[1], PUSH[2], PUSH[3], PUSH[4]);
endmodule
```

Вот пример работы на каких-то входных данных:

B0	B1	B2	B3	B4	V0	V1	V2	V3	V4	OUT
0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	1	0	0	0	0	1
0	1	0	0	0	1	1	0	0	0	1
0	0	1	0	0	1	1	0	0	0	0
0	0	0	1	0	1	1	0	1	0	1
0	0	0	0	0	1	1	0	1	1	0
0	0	0	0	0	1	1	0	1	0	0

Модуль с тестами для этой подсхемы называется `test_Comp10to1`.

3.2 Более важные вещи

3.2.1 StackCell

```
module StackCell(input wire R, C, D0, D1, D2, D3, output wire Q0, Q1, Q2, Q3);
    wire notR; not nr(notR, R);
    wire RorC; or r_or_c(RorC, R, C);
    wire D0andNR; and AND0(D0andNR, D0, notR);
    wire D1andNR; and AND1(D1andNR, D1, notR);
    wire D2andNR; and AND2(D2andNR, D2, notR);
    wire D3andNR; and AND3(D3andNR, D3, notR);
    DTrigger d_tr0(.C(RorC), .D(D0andNR), .Q(Q0));
    DTrigger d_tr1(.C(RorC), .D(D1andNR), .Q(Q1));
    DTrigger d_tr2(.C(RorC), .D(D2andNR), .Q(Q2));
    DTrigger d_tr3(.C(RorC), .D(D3andNR), .Q(Q3));
endmodule
```

Команды в тесте идут сверху вниз

R	C	D0	D1	D2	D3	Q0	Q1	Q2	Q3
0	0	0	0	0	0	x	x	x	x
0	1	1	0	0	1	1	0	0	1
1	1	1	0	0	1	0	0	0	0
0	1	1	1	1	1	1	1	1	1
0	0	1	0	1	1	1	1	1	1

Модуль с тестами для этой подсхемы называется `test_StackCell`.

3.2.2 MUX

Да, эта штука, написанная не самым удобным способом (😬) работает. Она получает на входе 20 ячееек $D_0, ..., D_{19}$, соответствующих 5 4-битным ячейкам памяти стека и A_2, A_1, A_0, кодирующие адрес ячейки стека, из которой мы читаем.
Вот тест:
Пускай у нас

```
D0 = 0; D1 = 1; D2 = 1; D3 = 0;
D4 = 1; D5 = 1; D6 = 0; D7 = 0;
D8 = 0; D9 = 0; D10 = 1; D11 = 1;
D12 = 1; D13 = 0; D14 = 0; D15 = 1;
D16 = 1; D17 = 0; D18 = 1; D19 = 1;
```

Тогда при таких вводах A_2, A_1, A_0 у нас такие выходы Q_0, Q_1, Q_2, Q_3:

A2, A1, A0	Q0, Q1, Q2, Q3	A2, A1, A0	Q0, Q1, Q2, Q3
0 0 0	0 1 1 0	1 0 0	1 0 1 1
0 0 1	1 1 0 0	1 0 1	0 1 1 0
0 1 0	0 0 1 1	1 1 0	1 1 0 0
0 1 1	1 0 0 1	1 1 1	0 0 1 1

Модуль с тестами для этой подсхемы называется `test_MUX`.

3.2.3 DEMUX

Полностью тестируется на всех входных данных в модуле `test_DEMUX`

3.2.4 MEM

Собирается из DEMUX, 5 ячеек StackCell и MUX.

```
module MEM(input wire A0, A1, A2, R_W, C, RESET,
                      D0, D1, D2, D3,
           output wire Q0, Q1, Q2, Q3);
    wire C0, C1, C2, C3, C4;
    wire W0, W1, W2, W3, W4, W5, W6, W7, W8, W9, W10,
         W11, W12, W13, W14, W15, W16, W17, W18, W19;
    wire RWandC; and make_RWandC(RWandC, R_W, C);
    Demux dmx(.A0(A0), .A1(A1), .A2(A2), .C(RWandC),
              .C0(C0), .C1(C1), .C2(C2), .C3(C3), .C4(C4));
    StackCell cell0(.R(RESET), .C(C0), .D0(D0), .D1(D1), .D2(D2), .D3(D3),
                    .Q0(W0), .Q1(W1), .Q2(W2), .Q3(W3));
    StackCell cell1(.R(RESET), .C(C1), .D0(D0), .D1(D1), .D2(D2), .D3(D3),
                    .Q0(W4), .Q1(W5), .Q2(W6), .Q3(W7));
    StackCell cell2(.R(RESET), .C(C2), .D0(D0), .D1(D1), .D2(D2), .D3(D3),
```

```
                .Q0(W8), .Q1(W9), .Q2(W10), .Q3(W11));
    StackCell cell3(.R(RESET), .C(C3), .D0(D0), .D1(D1), .D2(D2), .D3(D3),
                .Q0(W12), .Q1(W13), .Q2(W14), .Q3(W15));
    StackCell cell4(.R(RESET), .C(C4), .D0(D0), .D1(D1), .D2(D2), .D3(D3),
                .Q0(W16), .Q1(W17), .Q2(W18), .Q3(W19));
    MUX mx(.A0(A0), .A1(A1), .A2(A2),
           .D0(W0), .D1(W1), .D2(W2), .D3(W3),
           .D4(W4), .D5(W5), .D6(W6), .D7(W7),
           .D8(W8), .D9(W9), .D10(W10), .D11(W11),
           .D12(W12), .D13(W13), .D14(W14), .D15(W15),
           .D16(W16), .D17(W17), .D18(W18), .D19(W19),
           .Q0(Q0), .Q1(Q1), .Q2(Q2), .Q3(Q3));
endmodule
```

Модуль с тестами для этой подсхемы называется `test_MEM`.

3.3 Арифметические вещи

3.3.1 Plus1

```
module Plus1(input wire A0, A1, A2, output wire B0, B1, B2);
    wire S0, S1, S2, S3, S4;
    ThreeToFive ttf(.A0(A0), .A1(A1), .A2(A2),
                    .S0(S0), .S1(S1), .S2(S2), .S3(S3), .S4(S4));
    FiveToThree fft(.S0(S4), .S1(S0), .S2(S1), .S3(S2), .S4(S3),
                    .A0(B0), .A1(B1), .A2(B2));
endmodule : Plus1
```

Напомню, что число $\overline{B_2B_1B_{0_2}}$ таково, что $\overline{B_2B_1B_{0_2}} = \left(\overline{A_2A_1A_{0_2}} + 1\right) \bmod 5$, $\overline{B_2B_1B_{0_2}} \in \{0, ..., 4\}$.
Вот пример работы на **всех** тестовых входных данных для этого модуля:

A0	A1	A2	B0	B1	B2
0	0	0	0	0	1
0	0	1	0	1	0
0	1	0	0	1	1
0	1	1	1	0	0

A2	A1	A0	B2	B1	B0
1	0	0	0	0	0
1	0	1	0	0	1
1	1	0	0	1	0
1	1	1	0	1	1

Этот модуль тестируется на всех входных данных в модуле `test_Plus1`

3.3.2 Minus1

```
module Minus1(input wire A0, A1, A2, output wire B0, B1, B2);
    wire S0, S1, S2, S3, S4;
    ThreeToFive ttf(.A0(A0), .A1(A1), .A2(A2),
                    .S0(S0), .S1(S1), .S2(S2), .S3(S3), .S4(S4));
    FiveToThree fft(.S0(S1), .S1(S2), .S2(S3), .S3(S4), .S4(S0),
                    .A0(B0), .A1(B1), .A2(B2));
endmodule : Minus1
```

Напомню, что число $\overline{B_2B_1B_{0_2}}$ таково, что $\overline{B_2B_1B_{0_2}} = \left(\overline{A_2A_1A_{0_2}} - 1\right) \bmod 5$, $\overline{B_2B_1B_{0_2}} \in \{0, ..., 4\}$.
Вот пример работы на **всех** тестовых входных данных для этого модуля:

A0	A1	A2	B0	B1	B2
0	0	0	1	0	0
0	0	1	0	0	0
0	1	0	0	0	1
0	1	1	0	1	0

A2	A1	A0	B2	B1	B0
1	0	0	0	1	1
1	0	1	1	0	0
1	1	0	0	0	0
1	1	1	0	0	1

Этот модуль тестируется на всех входных данных в модуле `test_Minus1`

3.3.3 MinusMod5

Как и обычно, перефигачиваем схему из logisim в verilog.

```
module MinusMod5(input wire A0, A1, A2, B0, B1, B2, output wire R0, R1, R2);
    wire X00, X01, X02; // A2A1A0 mod 5
    wire T0, T1, T2, T3, T4; // промежуточные провода для этого
    ThreeToFive ttf1(.A0(A0), .A1(A1), .A2(A2),
                     .S0(T0), .S1(T1), .S2(T2), .S3(T3), .S4(T4));
    FiveToThree fft1(.A0(X00), .A1(X01), .A2(X02),
                     .S0(T0), .S1(T1), .S2(T2), .S3(T3), .S4(T4));
    wire X10, X11, X12; // A2A1A0 - 1 mod 5
    wire X20, X21, X22; // A2A1A0 - 2 mod 5
    wire X30, X31, X32; // A2A1A0 - 3 mod 5
    wire X40, X41, X42; // A2A1A0 - 4 mod 5
    Minus1 m0(.A0(X00), .A1(X01), .A2(X02), .B0(X10), .B1(X11), .B2(X12));
    Minus1 m1(.A0(X10), .A1(X11), .A2(X12), .B0(X20), .B1(X21), .B2(X22));
    Minus1 m2(.A0(X20), .A1(X21), .A2(X22), .B0(X30), .B1(X31), .B2(X32));
    Minus1 m3(.A0(X30), .A1(X31), .A2(X32), .B0(X40), .B1(X41), .B2(X42));

    wire S0, S1, S2, S3, S4;
    ThreeToFive ttf2(.A0(B0), .A1(B1), .A2(B2),
                     .S0(S0), .S1(S1), .S2(S2), .S3(S3), .S4(S4));

    Comp10to1 c0(.B0(S0), .B1(S1), .B2(S2), .B3(S3), .B4(S4),
                 .V0(X00), .V1(X10), .V2(X20), .V3(X30), .V4(X40), .OUT(R0));
    Comp10to1 c1(.B0(S0), .B1(S1), .B2(S2), .B3(S3), .B4(S4),
                 .V0(X01), .V1(X11), .V2(X21), .V3(X31), .V4(X41), .OUT(R1));
    Comp10to1 c2(.B0(S0), .B1(S1), .B2(S2), .B3(S3), .B4(S4),
                 .V0(X02), .V1(X12), .V2(X22), .V3(X32), .V4(X42), .OUT(R2));
endmodule : MinusMod5
```

В тестах есть модуль `test_MinusMod5`, в нём перебираются **всевозможные** значения уменьшаемого и вычитаемого и проверяется на правильность результат работы `MinusMod5`.

3.3.4 PointerModifier

Написано в модуле `PointerModifier` Полностью тестируется на тесте `test_PointerModifier`

3.3.5 PointerManager

Написано в модуле `PointerManager` Полностью тестируется на тесте `test_PointerManager`

3.4 Собираем всё, что сделали в схему

3.4.1 InAndOutHelper

Написано в модуле `InAndOutHelper` Полностью тестируется на тесте `test_InAndOutHelper`

3.4.2 Stack

Написано в модуле `Stack`. Он принимает на входе значение операции в проводах Op0 и Op1, RESET, C (синхронизацию), индекс ячейки для get в [2:0]G, биты, которые нужно класть в [3:0]D, и имеет выходы ANS0, ANS1, ANS2, ANS3.

3.4.3 Главный модуль — stack_structural

Просто stack, к которому подключили входы и выходы с правильными названиями. Тестируется в модуле stack_structural_tb.

4 Пишем на языке SystemVerilog поведенческую модель стека (stack_behaviour)

Я реализовал всё в одном модуле Stack (в нём входы и выходы идейно такие же, как и в stack_behaviour_normal), и потом этот модуль правильно подключил в модуле stack_behaviour_normal. Тесты для этого никак не поменялись, почти.

5 Тестирование

Для начала я покажу, как устроено тестирование в stack_structural_tb.sv (stack_behaviour_tb.sv – аналогично). Для начала, тестирование главного модуля (стека в целом) Там происходит следующее.
Сначала делается RESET: в стеке станет [0, 0, 0, 0, 0], указатель на 0
Далее пушаются последовательно 4, 15, 2, в стеке станет [4, 15, 2, 0, 0], указатель на 3
Далее делают 2 раза pop, в стеке станет [4, 15, 2, 0, 0], указатель на 1
Далее делается push 10, 3, 9, 8, в стеке станет [4, 10, 3, 9, 8], указатель на 0
Далее делается push 1, 0, 5, в стеке станет [1, 0, 5, 9, 8], указатель на 3.
Далее делается RESET, после этого в стеке станет [0, 0, 0, 0, 0]
Далее делают pop, в стеке по-прежнему [0, 0, 0, 0, 0].
Далее делают push 4, в стеке станет [4, 0, 0, 0, 0]
Более потробнее в комментариях к тесту. При запуске теста он пишет, что собственно, происходит:

```
DEBUG MAIN
1)Делаем RESET
Проверим, что везде лежат нули:
get 0: 0000 ( 0)
get 1: 0000 ( 0)
get 2: 0000 ( 0)
get 3: 0000 ( 0)
get 4: 0000 ( 0)
get 5: 0000 ( 0)
get 6: 0000 ( 0)
get 7: 0000 ( 0)
2)Давайте теперь запушаем что-то: (4; 15; 2)
PUSHED 0100 ( 4)
PUSHED 1111 (15)
PUSHED 0010 ( 2)
И снова проверим, что у нас там лежит в стеке:
get 0: 0010 ( 2)
get 1: 1111 (15)
get 2: 0100 ( 4)
get 3: 0000 ( 0)
get 4: 0000 ( 0)
get 5: 0010 ( 2)
```

```
get 6: 1111 (15)
get 7: 0100 ( 4)
3)Сделаем два раза pop
POPPED 0010 ( 2)
POPPED 1111 (15)
И снова проверим, что у нас там лежит в стеке:
get 0: 0100 ( 4)
get 1: 0000 ( 0)
get 2: 0000 ( 0)
get 3: 0010 ( 2)
get 4: 1111 (15)
get 5: 0100 ( 4)
get 6: 0000 ( 0)
get 7: 0000 ( 0)
4)Давайте снова запушаем что-то: (10, 3, 9, 8)
PUSHED 1010 (10)
PUSHED 0011 ( 3)
PUSHED 1001 ( 9)
PUSHED 1000 ( 8)
И снова проверим, что у нас там лежит в стеке:
get 0: 1000 ( 8)
get 1: 1001 ( 9)
get 2: 0011 ( 3)
get 3: 1010 (10)
get 4: 0100 ( 4)
get 5: 1000 ( 8)
get 6: 1001 ( 9)
get 7: 0011 ( 3)
5)И ещё раз запушаем что-то, чтобы случился переход по циклу и перезапись: (1, 0, 5)
PUSHED 0001 ( 1)
PUSHED 0000 ( 0)
PUSHED 0101 ( 5)
И снова проверим, что у нас там лежит в стеке:
get 0: 0101 ( 5)
get 1: 0000 ( 0)
get 2: 0001 ( 1)
get 3: 1000 ( 8)
get 4: 1001 ( 9)
get 5: 0101 ( 5)
get 6: 0000 ( 0)
get 7: 0001 ( 1)
6)Снова делаем RESET
Проверим, что везде лежат нули:
get 0: 0000 ( 0)
get 1: 0000 ( 0)
get 2: 0000 ( 0)
get 3: 0000 ( 0)
get 4: 0000 ( 0)
get 5: 0000 ( 0)
get 6: 0000 ( 0)
get 7: 0000 ( 0)
7)Попробуем сделать pop
POPPED 0000 ( 0)
И снова проверим, что у нас там лежит в стеке:
get 0: 0000 ( 0)
get 1: 0000 ( 0)
```

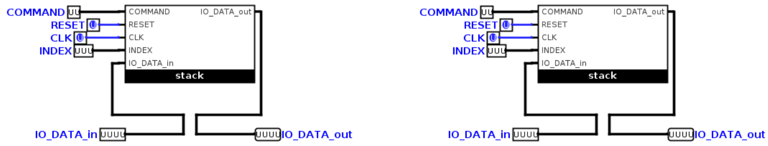
```
get 2: 0000 ( 0)
get 3: 0000 ( 0)
get 4: 0000 ( 0)
get 5: 0000 ( 0)
get 6: 0000 ( 0)
get 7: 0000 ( 0)
8)Давайте снова запушаем что-то: (4)
PUSHED 0100 ( 4)
И снова проверим, что у нас там лежит в стеке:
get 0: 0100 ( 4)
get 1: 0000 ( 0)
get 2: 0000 ( 0)
get 3: 0000 ( 0)
get 4: 0000 ( 0)
get 5: 0100 ( 4)
get 6: 0000 ( 0)
get 7: 0000 ( 0)
```

Теперь, как правильно тестировать программу. Вверху файла stack_structural.sv (7 строка), написано: `bit test_main = 1;`. Когда это значение 1, то у нас тестируется только главный модуль и выводится то, что написано нескоьлко строк выше. Если это зачение поменять на 0, то будут тестироваться все остальные модули. Если модуль протестирован успешно, программа пишет “No errors in [название модуля]!” (еще бывает в скобочках FULL_CHECK, если перебирались всевозможные входные данные к тесту).
То есть, у меня оно выводит (когда нет ошибок) такое:

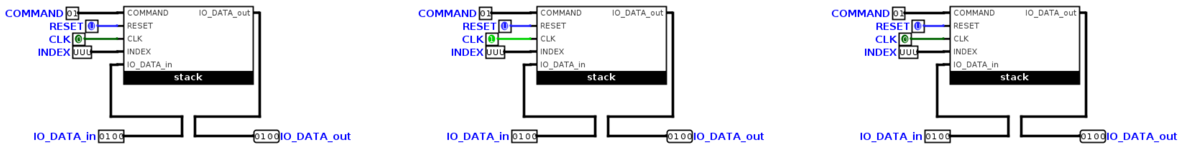
```
No errors in OpToInf! [FULL CHECK]
No errors in InAndOutHelper! [FULL CHECK]
No errors in FiveToThree! [FULL CHECK]
No errors in Comp10to1!
No errors in D-trigger!
No errors in ThreeToFive! [FULL CHECK]
No errors in MUX!
No errors in Minus1! [FULL CHECK]
No errors in Plus1! [FULL CHECK]
No errors in StackCell!
No errors in DEMUX! [FULL CHECK]
No errors in ThreePusher! [FULL CHECK]
No errors in MEM!
No errors in Pointer Modifier! [FULL CHECK]
No errors in minusMod5! [FULL CHECK]
No errors in PointerManager!
No errors in Stack!
```

Также есть флаги для дебага подсхем main и stack (need_to_debug_MEMб need_to_debug_stack). Если сделать их равными 1, то будет выводиться подробный дебаг при тестировании этих модулей (использовалось для локального дебага).
Теперь тестирование в logisim. (Те же самые действия в упрощённом (в смысле их меньше) формате).

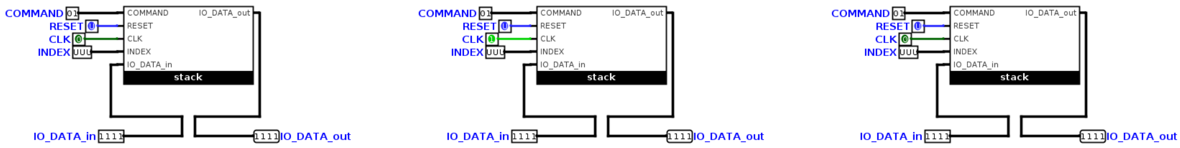
В начале делаем RESET (первая картинка — то, что было в начале, вторая — то, что после RESET (с виду то же самое).)



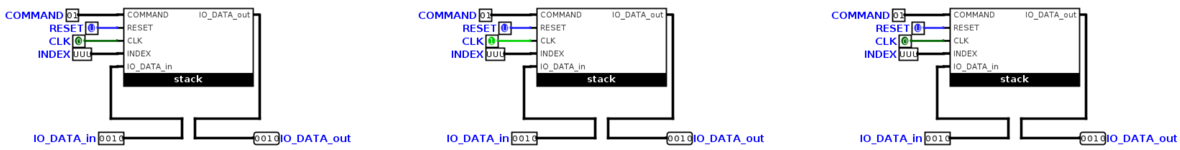
Пушаем 4 (скрины с переключением синхронизации):



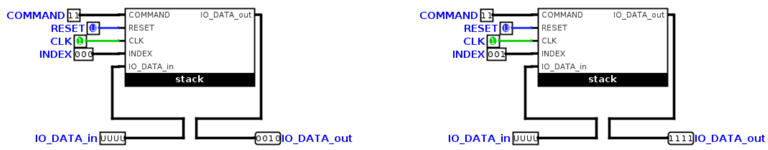
Пушаем 15 (скрины с переключением синхронизации):

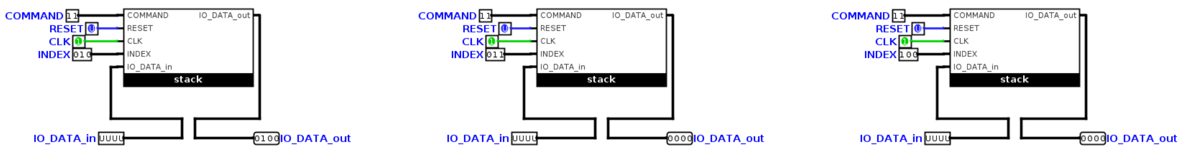


Пушаем 2 (скрины с переключением синхронизации):

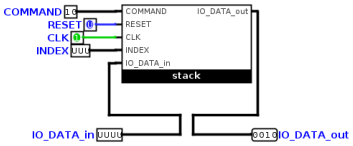


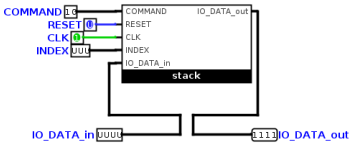
Посмотрим, что нам выведет get(0), get(1), ..., get(4):



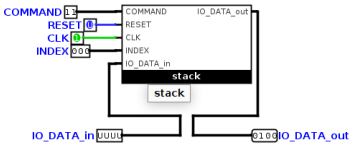


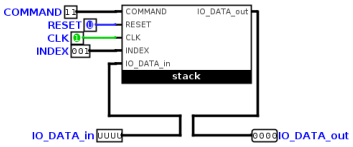
Дважды делаем pop:

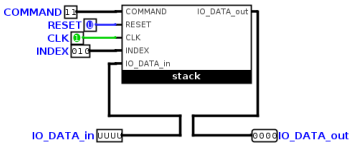


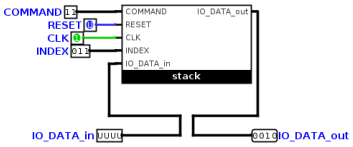


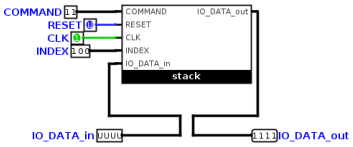
Посмотрим, что нам выведет get(0), get(1), ..., get(4):



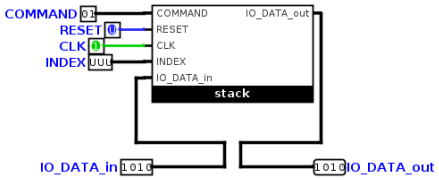


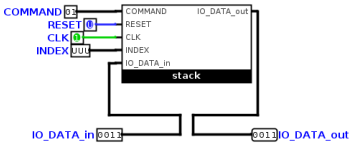


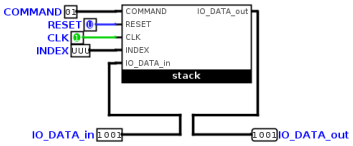


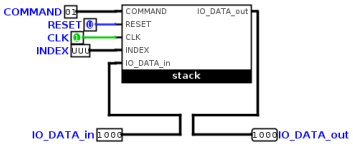


Делаем push 10,3,9,8.

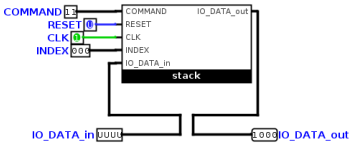


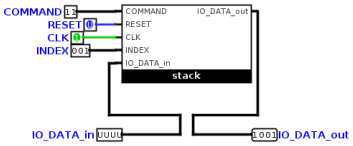


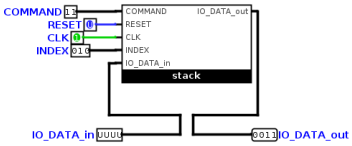


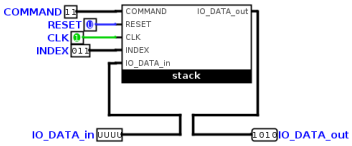


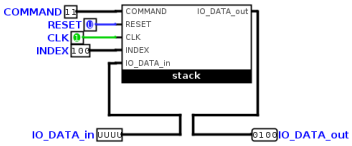
Посмотрим, что нам выведет get(0), get(1), ..., get(4):



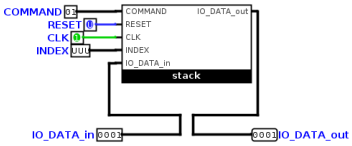


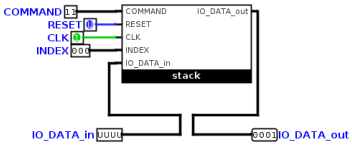


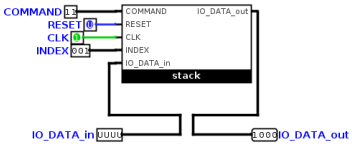


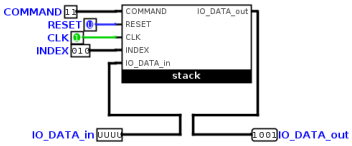


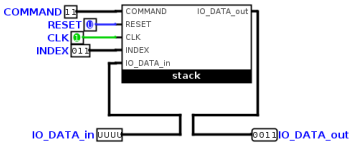
Запушаем 1 (случится переход по циклу) и сделаем get(0), ..., get(4).

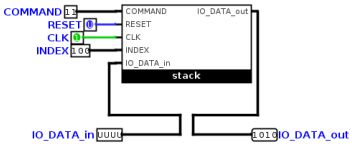




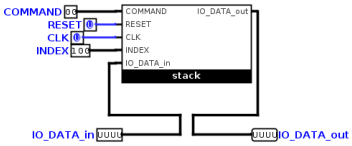


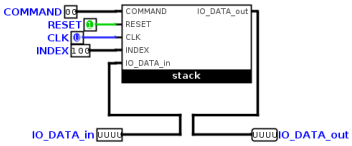




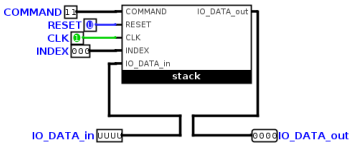


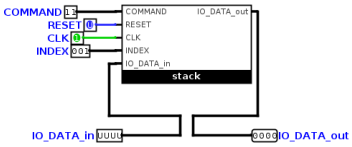
Сделаем NOP и RESET:

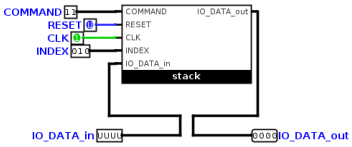


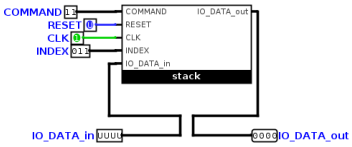


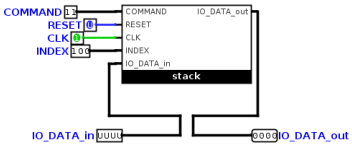
Посмотрим, что нам выведет get(0), get(1), ..., get(4):











Кажется, на этом всё. Я благодарен читателю, за то, что он добрался до этого места и извиняюсь за качество этой работы (а особенно некоторых модулей в stack_structural).