GeekBrains

Лекция 2. Операции с данными

Введение в информатику

Оглавление

[00:01:36]

Вступление

Уважаемые студенты, всем доброго времени суток! Я рад вас приветствовать на нашей второй лекции по курсу информатики. Предлагаю вспомнить, на чём мы остановились на прошлом уроке. Мы познакомились с элементной базой, на основе которой строится вся вычислительная техника. То есть с транзисторами, которые могут работать с булевой алгеброй и с 0 и 1, представляющими саму булеву алгебру. Но также перед нами стояла задача, как мы можем представить данные реального мира с помощью 0 и 1 в нашем компьютере и как мы можем ими оперировать. Именно об этом мы сегодня поговорим.

[00:01:36]

Сравнение десятичной и двоичной логики

Для начала посмотрим, как можно работать с обычными числами представляя их в двоичном виде. Вспомним, как мы считаем в обычной десятичной системе.

Допустим, число 356. Оно состоит из:

- 6 — едениц
- 5 — десятков (50)
- 3 — сотен (300)

Сложив эти числа мы получим 356.

Обратите внимание!

- Единицы — 10^0
- Десятки — 10^1
- Сотни — 10^2
- Тысячи — 10^3
- И так далее...

В двоичной системе всё тоже самое, только цифр у нас всего две — 0 и 1. Соответственно, у нас всё будет умножаться на 2 в какой-то степени.

- Один — 2^0

- Два — 2^1
- Четыре — 2^2
- Восемь — 2^3
- И так далее...

С помощью двоичной системы мы также можем представлять любое число, которое можем посчитать.

Вспомним, как мы считаем в десятичной системе. У нас есть девять цифр и каждый раз мы прибавляем единицу на конец. То есть 356+1=357. Когда мы доходим до 9 (например, 359) и прибавляем один, наше следующее число в конце становится 0, но при этом десятки увеличиваются на 1 (например, 359+1+360).

В двоичной системе происходит всё тоже самое. Единственное отличие у нас всего 2 цифры — 0 и 1. То есть 0 — это первая цифра, по аналогии с 0 в десятичных. А 1 — это последняя цифра, как 9 в десятичной системе.

Сравнение десятичной и двоичной логики

356 = 6 x 1 + 5 x 10 + 3 x 100
357 = 7 x 1 + 5 x 10 + 3 x 100
358 = 8 x 1 + 5 x 10 + 3 x 100
359 = 9 x 1 + 5 x 10 + 3 x 100
360 = 0 x 1 + 6 x 10 + 3 x 100

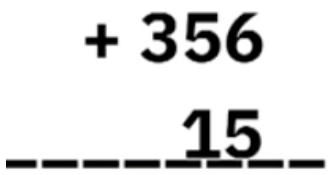
101100100 = 0x1+0x2+1x4+0x8+0x16+1x32+1x64+0x128+1x256 = 356
101100101 = 1x1+0x2+1x4+0x8+0x16+1x32+1x64+0x128+1x256 = 357
101100110 = 0x1+1x2+1x4+0x8+0x16+1x32+1x64+0x128+1x256 = 358
101100111 = 1x1+1x2+1x4+0x8+0x16+1x32+1x64+0x128+1x256 = 359
101101000 = 0x1+0x2+0x4+1x8+0x16+1x32+1x64+0x128+1x256 = 360

Важно понимать! Двоичная система — это полноценная система счисления, в ней можно выражать любые числа и работать с ними также, как в десятичной.

Также важно понимать! Что в двоичной системе нам довольно сложно представлять информацию и оперировать ей, поскольку мы привыкли к десятичной системе, но для компьютера двоичная система наиболее удобна, так как транзисторы оперируют именно 0 и 1. Немного позже мы более подробно изучим, как это работает.

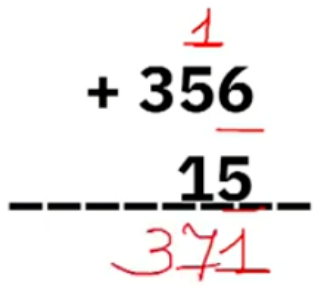
Сейчас разберёмся, как происходит сложение чисел в двоичной и десятичной системе.

В десятичной системе складывая числа 356 и 15 мы выполняем следующие действия:

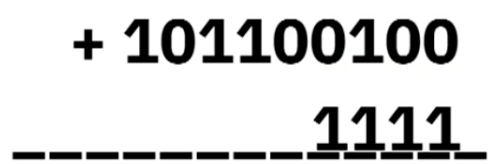


1. Складываем последние цифры числа 356 и числа 15. **6 + 5 = 11.** При сложении столбиком мы **записываем 1** в крайнем правом ряду, а 1 (вторую, она же 10) добавляем к десяткам, в нашем случае к 5 и 1.
2. Теперь складываем десятки 5 + 1 = 6, но у нас ещё есть 1 из первого пункта, поэтому **5 + 1 + 1 = 7.**
3. Поскольку число с сотнями у нас одно **300**, то просто переносим **3** вниз (под черту).

Таким образом получаем:

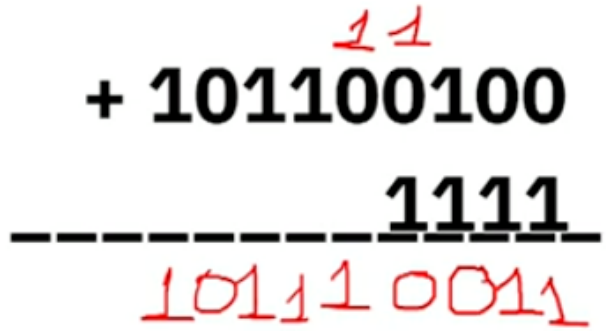


Что же происходит в двоичной системе счисления? Принцип тот же, что и при сложении десятичных.



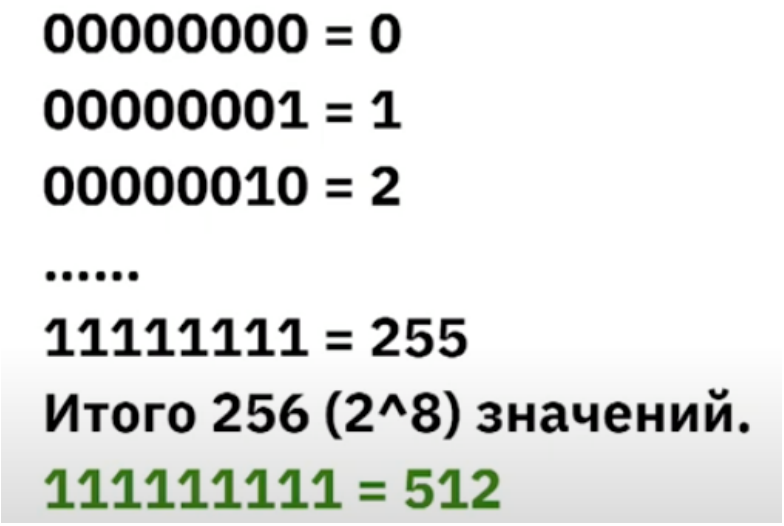
1. Начинаем с крайнего правого ряда. К **0 + 1 = 1** и записываем под чертой.
2. Продолжаем **0 + 1 = 1**.
3. Теперь складываем **1 + 1.** В этом мы получаем число 1 0 в двоичной системе. **0 — записываем, а 1 — переносим** в следующий столбец слева.
4. Тоже самое что и в пункте 3. **0 — пишем, 1 — переносим**.

5. Поскольку в пункте 4 мы перенесли сюда 1, то складываем **0 + 1 = 1**.
6. Остальные числа просто переносим вниз, так как их не с чем складывать.



Как видите принцип один и тот же. **Запомните этот принцип сложения столбиком, в дальнейшем он нам пригодится, когда мы будем его реализовывать на транзисторах!**

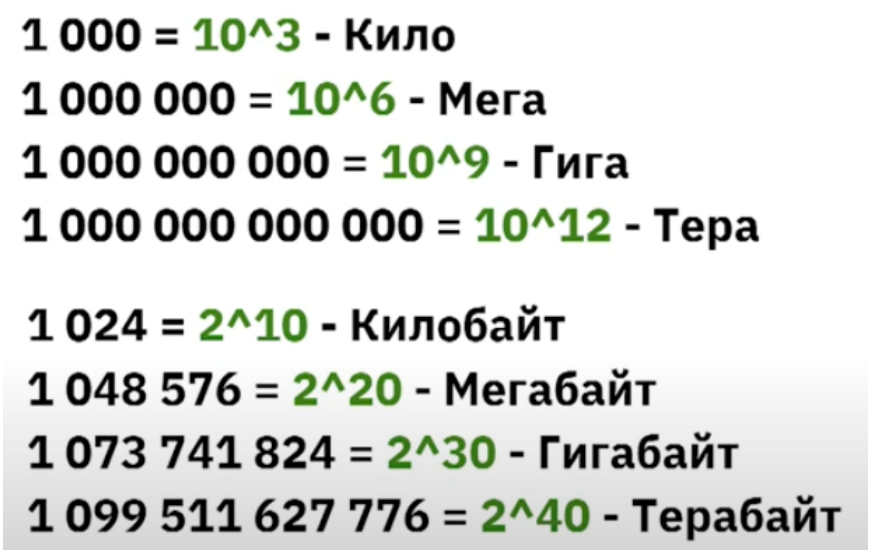
Важно понимать, что количество разрядов позволяет выразить определённые числа. Если у нас всего 8 разрядов, то максимальное число, которое мы можем выразить — 255. Получается с помощью восьми разрядов мы можем выразить числа от 0 до 255. Каждый разряд в представлении компьютера называется — **бит**. Когда нам нужно выразить больше чисел, мы просто увеличиваем количество бит, при увеличении бит на 1 число возможных значений возрастёт до 512, если увеличить ещё на 1 бит, то 1024 значения и так далее.



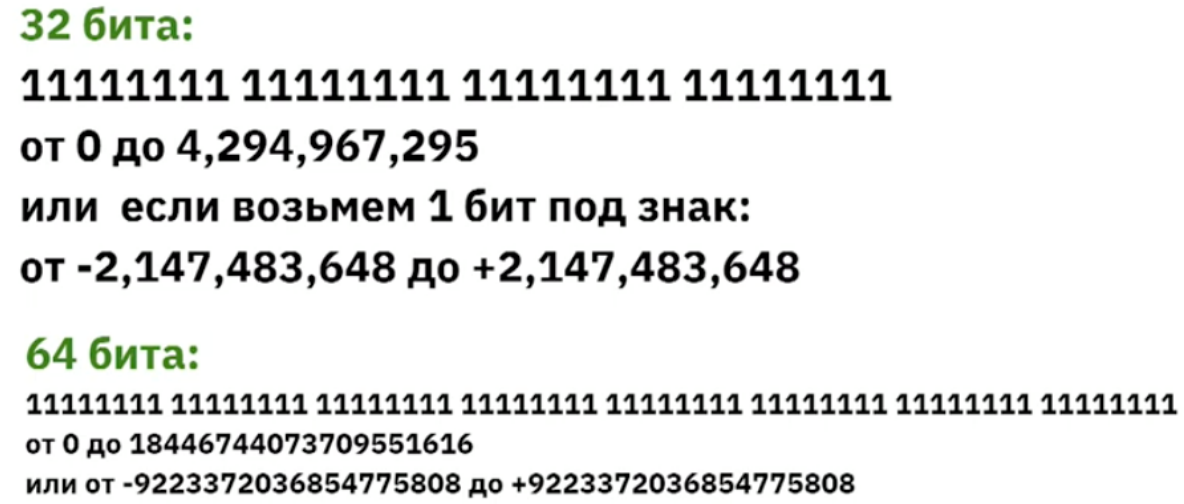
Ранние компьютеры оперировали 8 битными числами, поэтому для них есть специальное название **байт**.

8 бит = 1 байт

Также как в десятичной системе, у нас есть приставки для обозначения больших чисел в двоичной системе.



Почему 2^{10}, 2^{20}, 2^{30} и так далее? К этому надо относиться как к факту, потому что эти числа более близки к соответствующим числам в десятичной системе.



Потом у нас компьютеры стали оперировать числами состоящие из 32 бит и соответственно у нас максимальное число, которое может выразить 32 бита, в десятичном виде представляет 4 миллиарда с лишним. Это довольно большое число, но население планеты уже не выразишь с помощью 32 битного числа. Более того, хотелось бы выражать ещё отрицательное число, поэтому мы можем украсть 1 бит и если он будет, допустим, равен нулю, это будет обычно положительное число, а если он у нас равен единице, то отрицательное число. Поэтому мы можем уже выражать диапазон от -2 миллиардов с лишним до 2 миллиардов с лишним.

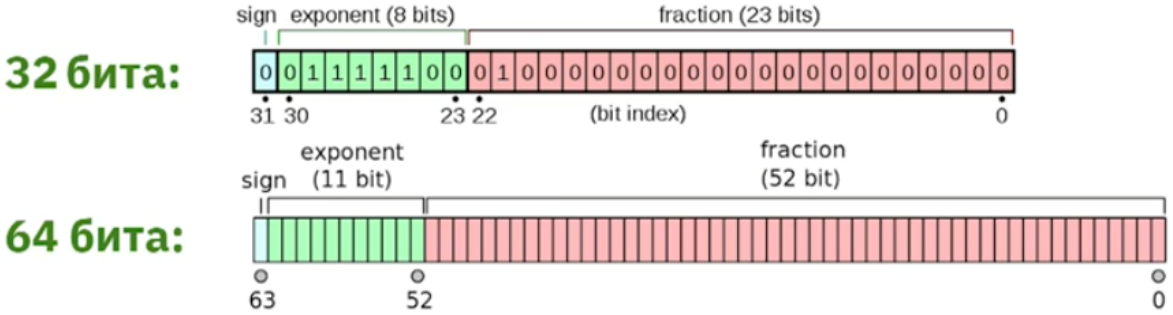
После этого компьютеры ещё более развивались у нас, появилась 64-битная архитектура, которая могла оперировать уже 64 битными числами. Обычное 64-битное число может себе содержать значение примерно от -9 квинтиллионов до 9 квинтиллионов.

[00:09:43]

Выражение дробей в двоичном коде

Также неплохо было бы выражать и десятичные числа в виде дроби, которые у нас записаны. Они также называются числа с плавающей точкой или float, потому что точка может ставиться в любом месте числа, выражая ту или иную дробь.

Экспоненциальная запись:
-298.564 = -0.298564*10^3
3.14 = 0.314*10^1



Как представляют такие числа? Здесь нам поможет стандарт IEEE 754, это стандарт инженеров из института электроники и электротехники. Зачем был придуман этот стандарт? Есть довольно много способов выражать такие числа, но надо было прийти к какому-то одному, чтобы производители представляли эти числа примерно одинаково.

Стандарт работает по следующему принципу. В десятичной системе мы любое число дробное можем выразить в виде основания или дробной части, которая будет находиться от 0 до 1. Допустим, в нашем примере -298,564 мы можем записать как минус -0.298,564 и умноженное на десятку в какой-то степени, в нашем случае — это 10^3. Десятичную дробь **3,14** мы также можем записать в виде числа, которое находится от 0 до 1. Это будет **0,314 *** 10^1.

Таким образом, мы можем отдельно хранить дробную часть и степень десятки. Стандарт говорит, что если у нас дано 32 битное число типа float, мы можем хранить дробную часть в виде 23 бит, а часть экспонент, мы будем хранить в степени, в которую возводим.

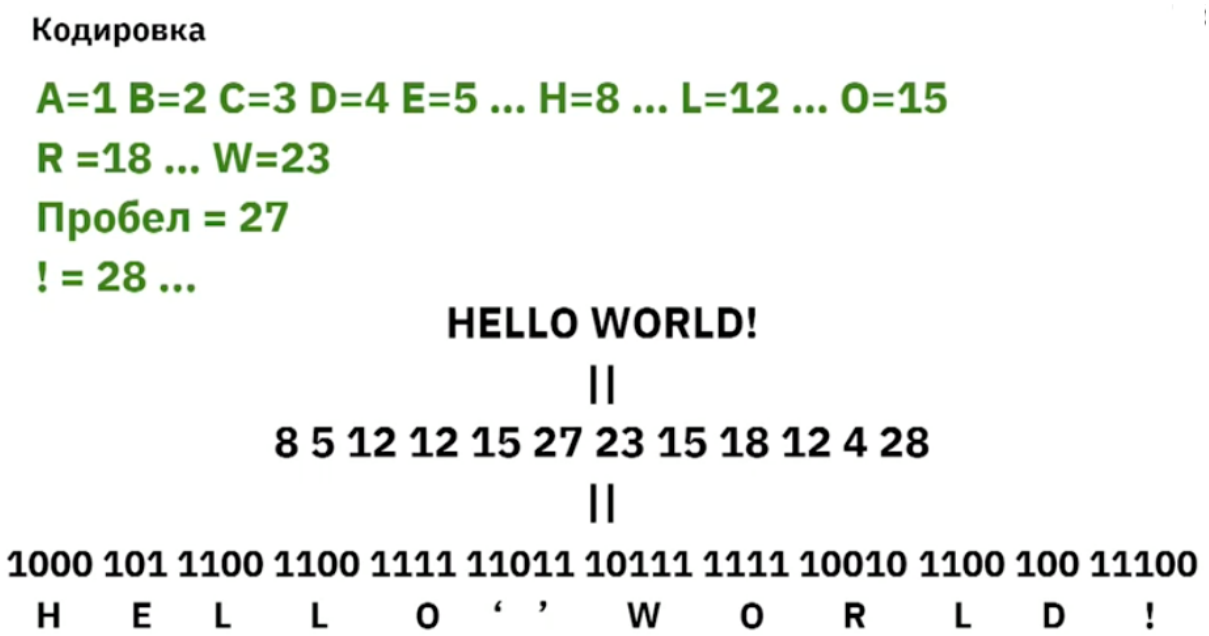
Важно помнить! Для битов снова работает степень двойки. То есть мы умножаем на 2 в какой-то степени, при этом дробная часть записывается в виде классического двоичного числа.

В 64 битном стандарте у нас под экспоненту выделено 11 бит (под степень двойки), а под дробную часть выделено 52 бита, что позволяет хранит более точные числа и оперировать ими.

[00:12:02]

Кодировка

Как выражать числа мы более-менее понимаем. Но шло время, компьютеры развивались и помимо чисел им приходилось работать с текстом. Чтобы выражать какой-либо текст и хранить его или оперировать этим текстом в компьютерах, нам необходима — **кодировка**. Она позволяет одну букву закодировать в определённое число. Мы можем придумать свою кодировку, например, у нас буква a — 1, буква b — 2 и так далее.



Если мы допустим запишем фразу hello world с восклицательным знаком, мы можем выразить в виде такого набора чисел 8, 5, 12, 12, 15, 27 и так далее. А если мы будем выражать в двоичном виде согласно кодировке, которую мы только что придумали, то будет такая последовательность 0 и 1.

Но нужно понимать, что если каждый производитель будет придумывать свою кодировку, то они будут различаться и последовательность 0 и 1 будет отличаться от компьютера к компьютеру. Поэтому был придуман стандарт ASCII, который расшифровывается как American Standard Code Information

Interchange. Я подчеркиваю именно слово interchange, то есть он создан был для обмена между разными компьютерами, чтобы кодировка была одинаковая, чтобы слова и буквы представлялись одинаково для различных компьютеров, которые использовались в те времена.

Кодировка ASCII

dec	hex	oct	char	dec	hex	oct	char	dec	hex	oct	char	dec	hex	oct	char
0	0	000	NULL	32	20	040	space	64	40	100	@	96	60	140	`
1	1	001	SOH	33	21	041	!	65	41	101	A	97	61	141	a
2	2	002	STX	34	22	042	"	66	42	102	B	98	62	142	b
3	3	003	ETX	35	23	043	#	67	43	103	C	99	63	143	c
4	4	004	EOT	36	24	044	$	68	44	104	D	100	64	144	d
5	5	005	ENQ	37	25	045	%	69	45	105	E	101	65	145	e
6	6	006	ACK	38	26	046	&	70	46	106	F	102	66	146	f
7	7	007	BEL	39	27	047	'	71	47	107	G	103	67	147	g
8	8	010	BS	40	28	050	(	72	48	110	H	104	68	150	h
9	9	011	TAB	41	29	051	)	73	49	111	I	105	69	151	i
10	a	012	LF	42	2a	052	*	74	4a	112	J	106	6a	152	j
11	b	013	VT	43	2b	053	+	75	4b	113	K	107	6b	153	k
12	c	014	FF	44	2c	054	,	76	4c	114	L	108	6c	154	l
13	d	015	CR	45	2d	055	-	77	4d	115	M	109	6d	155	m
14	e	016	SO	46	2e	056	.	78	4e	116	N	110	6e	156	n
15	f	017	SI	47	2f	057	/	79	4f	117	O	111	6f	157	o
16	10	020	DLE	48	30	060	0	80	50	120	P	112	70	160	p
17	11	021	DC1	49	31	061	1	81	51	121	Q	113	71	161	q
18	12	022	DC2	50	32	062	2	82	52	122	R	114	72	162	r
19	13	023	DC3	51	33	063	3	83	53	123	S	115	73	163	s
20	14	024	DC4	52	34	064	4	84	54	124	T	116	74	164	t
21	15	025	NAK	53	35	065	5	85	55	125	U	117	75	165	u
22	16	026	SYN	54	36	066	6	86	56	126	V	118	76	166	v
23	17	027	ETB	55	37	067	7	87	57	127	W	119	77	167	w
24	18	030	CAN	56	38	070	8	88	58	130	X	120	78	170	x
25	19	031	EM	57	39	071	9	89	59	131	Y	121	79	171	y
26	1a	032	SUB	58	3a	072	:	90	5a	132	Z	122	7a	172	z
27	1b	033	ESC	59	3b	073	;	91	5b	133	[	123	7b	173	{
28	1c	034	FS	60	3c	074	<	92	5c	134	\	124	7c	174	|
29	1d	035	GS	61	3d	075	=	93	5d	135	]	125	7d	175	}
30	1e	036	RS	62	3e	076	>	94	5e	136	^	126	7e	176	~
31	1f	037	US	63	3f	077	?	95	5f	137	_	127	7f	177	DEL

И здесь у нас кодировали символы с помощью 7 бит. 7 бит — 128 различных значений. Учитывая, что в английском алфавите 28 букв, на большие и маленькие уже уходит 56, у нас ещё остаётся много места для разных символов. **Таблица ASCII довольно не сложная и не громоздкая, но у неё есть один минус — она для английского алфавита.** Если надо использовать другие алфавиты, русский или китайский, в котором около 3000 символов, то нам нужен более мощный стандарт, который использует большее количество бит для хранения того или иного символа. И такой стандарт был придуман в 1992 году. Он называется стандарт Unicode и позволяет выражать любой символ любого алфавита, любого языка на нашей планете.

[00:14:30]

Представление данных в двоичном коде

Так же, как мы представляем символы в виде цифр, в виде 0 и 1 с помощью кодировок, то же самое мы делаем на самом деле и для наших изображений, для видео, для звука.

Давайте разберёмся с изображениями. Допустим, у нас есть чёрно-белое изображение, состоящее из пикселей. Тогда мы можем каждый пиксель обозначить через бит. Если 1 бит = 1, это чёрный пиксель, а если 1 бит = 0, это будет белый пиксель. Картинка может быть выражена всего лишь с помощью 2 бит. Более сложные картинки могут быть выражены с помощью набора бит и такой матрицы. Для неё потребуется всего лишь 25 бит. Картина немного усложняется, когда мы используем цветные картинки.

Есть такая модель представления цвета, как RGB (от слов Red, Green, Blue, Красный, Зеленый и Голубой). И она опирается на тот факт, что любой цвет в мире мы можем выразить благодаря трём компонентам (цветам). Представим, что у нас есть жёлтый цвет (пиксель). Для его записи используем красный и зелёный пиксели, но не синий. Получается, что на пиксель у нас уходит 3 бита. И каждый бит показывает наличие или отсутствует того или иного цвета в нашей смеси. 3 бита это довольно мало. 2^3 — это всего 8 цветов, которые мы можем выразить, если будем записывать в таком формате. Поэтому современные картинки хранят по целому байту на присутствии того или иного базового цвета (красного, зелёного или синего), получается 24 бита на пиксель, что даёт нам возможность выражать картинки примерно 16 миллионами цветов. Что в принципе достаточно много.

Важно понимать, что любую картинку мы представляем с помощью 0 и 1. Любое видео это просто набор чередующихся картинок из тех же 0 и 1. Тоже самое работает относительно звука, где мы каждый звук можем закодировать в 0 и 1 и проигрывать с определённой частотой. То есть мы понимаем, что

данные реального мира не так сложно представлять в виде 0 и 1, чтобы потом уже работать с ними на компьютерах. Главное заранее договориться о стандартах.

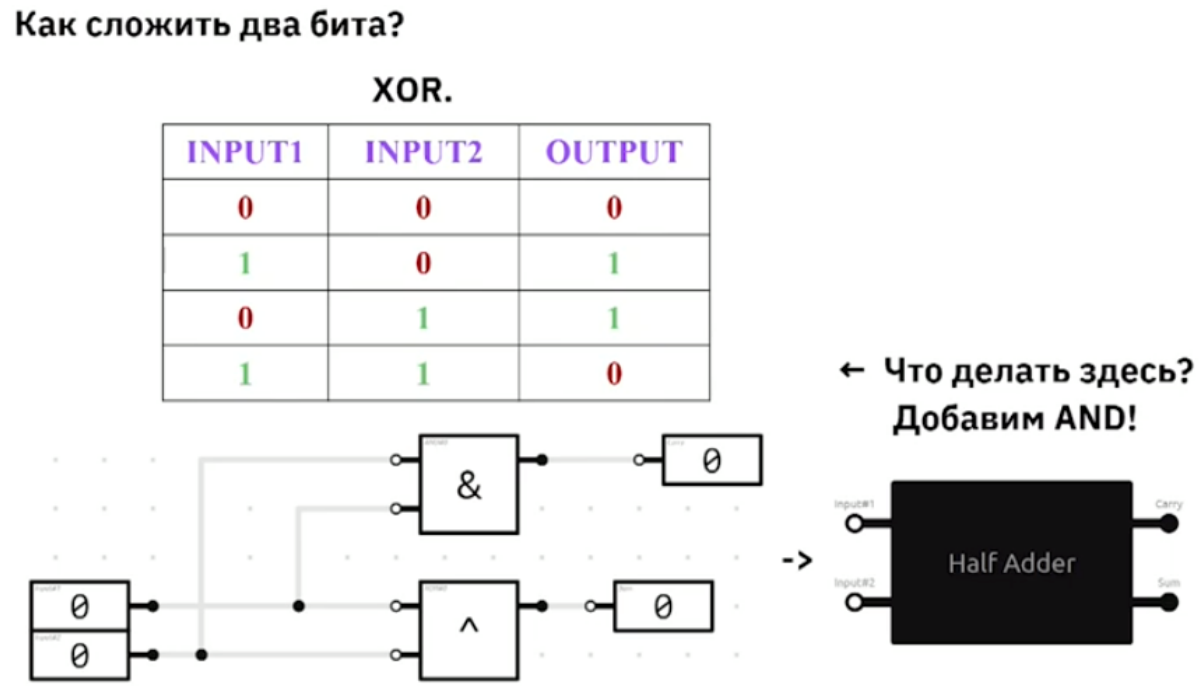
[00:17:12]

Arithmetic Logic Unit

Итак, а как мы можем оперировать этими 0 и 1 в компьютерах? Основной элемент, который позволяет делать различные операции над ними, называется Arithmetic Logic Unit (ALU). Внутри он состоит из двух блоков — арифметического и логического блока. Арифметически работает над математическими операциями, а логически над логическими.

[00:17:31]

Арифметический блок



Посмотрим на самую базовую операцию арифметического блока — операцию сложения. Попробуем понять, как эта операция производится внутри компьютера с помощью транзисторов. И для этого сначала сложим 2 бита. Если мы будем складывать 2 обычных бита, то нам поможет логический гейт XOR, который мы изучили на прошлом уроке.

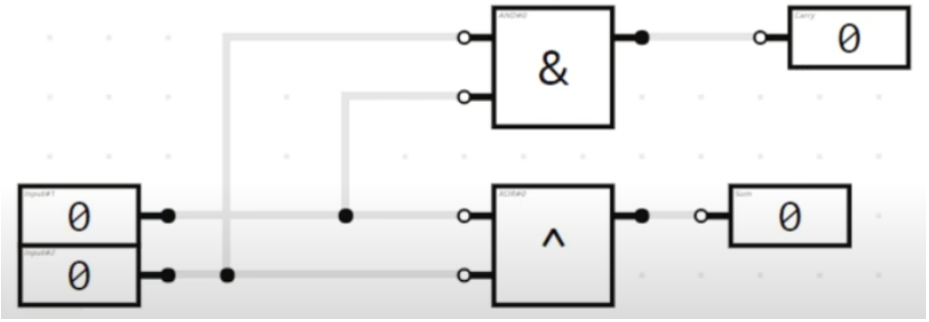
Итак, какой логический гейт возвратить значения input? Это логический гейт И. Поэтому соединив его параллельно, мы можем получить блок, который будет работать следующим образом. Если у нас будет подаваться 2 бита на

вход, то мы будем получать сумму благодаря нашему XOR, на выходе после него мы получим 1 только тогда, когда на входе будут две 1. Мы можем всё упаковать в устройство half adder или полусумматор. Он позволяет просто складывать 2 бита и выдавать перенос в единичку, если на вход подаётся две единички.

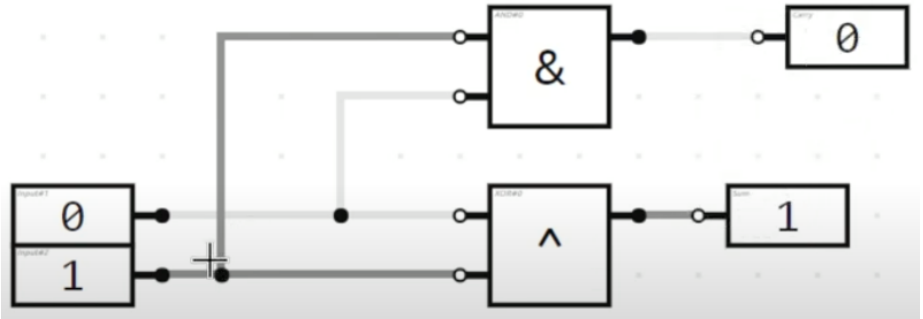
Как мы можем складывать теперь 2 бита с учётом переноса? Представим, что мы передвигаемся в следующий разряд и там перенос уже может быть. Он опять может быть либо 0, либо 1. Для этого мы можем переместиться в лабораторную среду Boolr, в котором мы можем строить любые схемы с помощью наших гейтов.

Посмотрим, как half adder работает внутри:

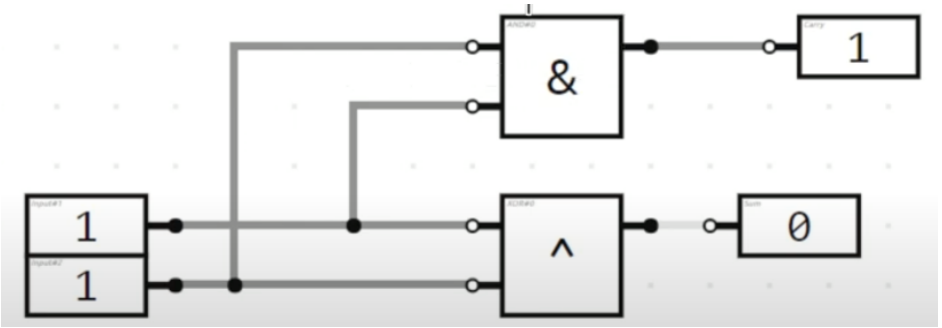
- **Если подаём на два входа 0**



- **Если подаём на один вход 1**

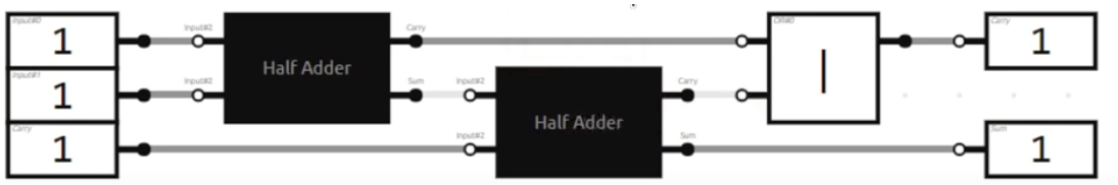


- **Если подаём на два входа 1**



А как работает устройство full adder? Это уже полный сумматор. Он должен принимать на вход carry с предыдущей позиции. А также два бита, которые нам необходимо сложить.

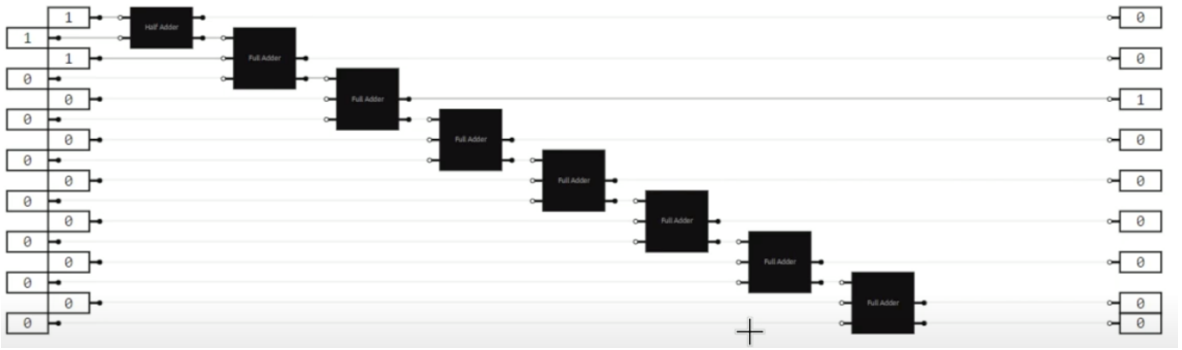
Если мы таким образом соберём нашу схему, то можем в результате сложения получить сумму, которая у нас выходит на последнем полусумматора. А carry, то есть перенос, получить в том случае, когда на входах две 1. Когда у нас все три 1, мы можем увидеть, что у нас сумма 1 и перенос возникает.



Когда у нас carry и переноса нет, то мы можем сложить 1 с 1, получить 0 в результате и перенос, который может пойти в следующий разряд.

[00:22:45]

Как сложить два 8 битных числа?



Посмотрим на схему, которая показывает, как можно сложить два 8 битных числа. **Обратите внимание**, что первые разряды у нас складываются с помощью half adder. Если в начале 1 и 1, мы получаем в результате 0. Он ушёл, а carry переносим на full adder. Он у нас умеет складывать с carry (переносом). Ниже мы подаём 0 и 1. Это у нас второй разряд. Опять full adder это складывает, и у него на выходе получается сумма, которая записывается в результирующий бит выходного числа. И снова возникает перенос, который может быть, а который может не быть. Но в нашем случае перенос есть, и он

отправляется на следующий full adder. Следующий full adder складывает у нас биты из третьего разряда, где у нас четвёрки (умножение на четвёрки идёт). И при этом он берёт перенос с предыдущего full adder. Так цепляясь друг за друга, выполняется сложение столбиков. Мы можем сложить два любых числа.Итак, мы поняли, как складывать два произвольных 8 битных числа, с помощью устройств half adder и full adder.

Для вычитания есть соответствующие компоненты, состоящие из простых логических гейтов, которые также состоят внутри из транзисторов, которые позволяют делать такие операции.

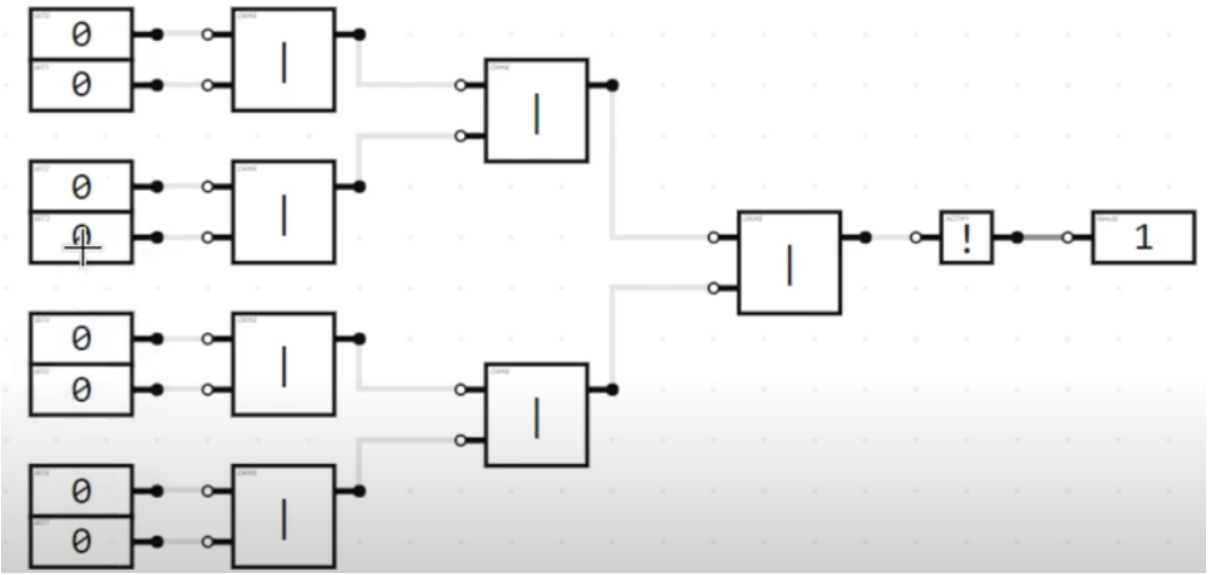
Что такое умножение? Умножение — это просто повторение сложения определённого количества раз. Если мы 5 * 10, то нужно 10 раз сложить 5 + 5. Поэтому если мы сюда поставим счётчик, мы можем выполнять это сложение несколько раз и получать операцию умножения.

Таким образом, напичков такими компонентами, как adder 8-бит или subtract 8-бит, мы можем сделать свой арифметический блок.

[00:25:38]

Логический блок

Логический блок позволяет нам проверять числа на больше или меньше и отвечать истинно или нет. Например, самое простейшее проверять число равно ли оно 0. Посмотрим, как работает эта схема.

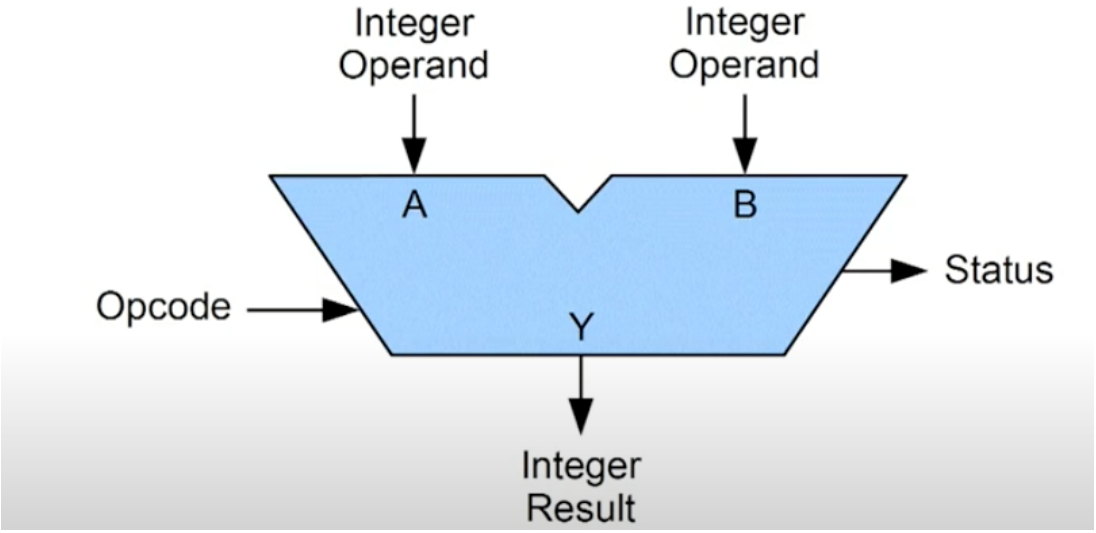


У нас есть 8-битное число. Если мы каждые 2 бита попарно свяжем через гейт ИЛИ, а потом эти два гейта ИЛИ тоже между собой свяжем, а в конце инвертируем их через операцию НЕ, то на выходе получим 1 только тогда, когда на входе все 0. Потому что как только в любой бит мы подадим

напряжение (1), у нас через все ИЛИ эта 1 пройдёт и инвертируется в 0, где бы эта 1 не была и сколько бы этих 1 не было. Такой довольно простой компонент, позволяет быстро проверить равно ли число 0 или нет. Вернуть в зависимости от этого результат false или true.

Такие же компоненты, правда, более сложные, есть для сравнения чисел на больше или меньше и так далее. Кстати, на больше или меньше можно вообще сравнить, вычитая более большее число из меньшего, мы получим отрицательный результат и первый бит будет 1.

Собрав всё вместе, мы получаем такое устройство, как Arithmetic Logic Unit. ALU обозначается большой буквы V:



- На вход **(Integer Operand)** подаётся число A, рядом подаётся число B.
- Слева на вход подаётся **Opcode (operation code**) — код операции, который надо производить. Например, это сложение или вычитание или сравнение на ноль, которое мы рассмотрели.
- Снизу у нас выдаётся результат **(Integer Result)**.
- Справа на выход у нас подаётся статус **(Status)** или флаг, который обозначает ошибку или какое-то определённое состояние. Например, если мы сложим два 8 битных числа, где все единички, мы получим 9 битное число, которое не влезет 8 бит. Поэтому у нас в конце перенос будет равен 1. И этот последний перенос мы можем вынести сюда в статус, тем самым обозначив ошибку переполнения памяти. Довольно распространённая ошибка в программировании, когда мы думаем, что оперируем бесконечной памятью, она всегда может закончиться и могут закончиться биты для хранения какого-то значения. В ALU статус

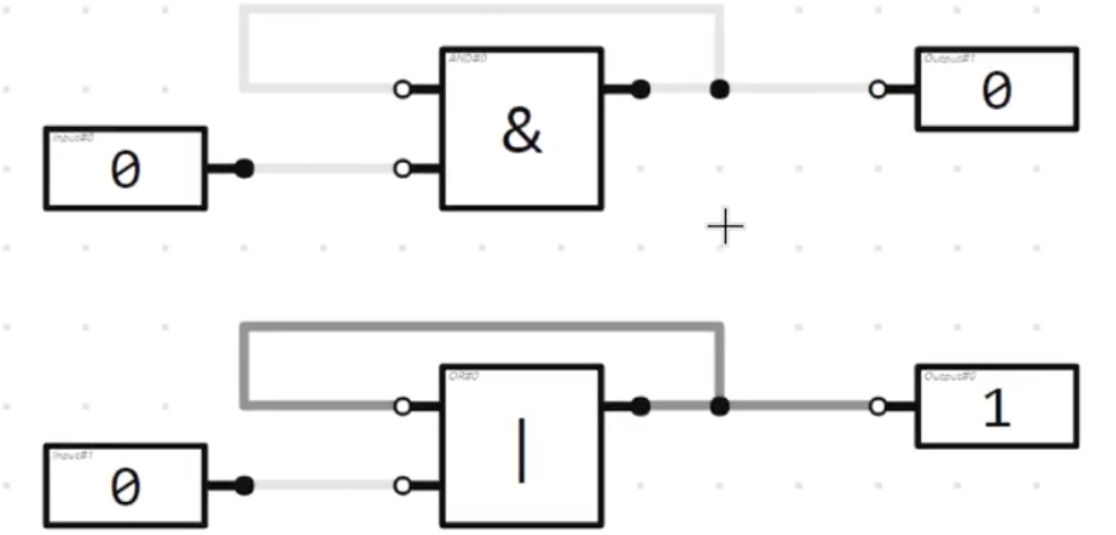
выполняет роль флага, который поднимается, когда у нас происходит ошибка.

Теперь мы знаем, как работают математические операции внутри. Они работают благодаря транзисторам. По сути, это может сравнить с выполнением математических операций при помощи счёт, в которых базовым элементами являются бусины. В компьютерах роль бусин играет транзистор.

[00:28:56]

Откуда ALU берёт данные?

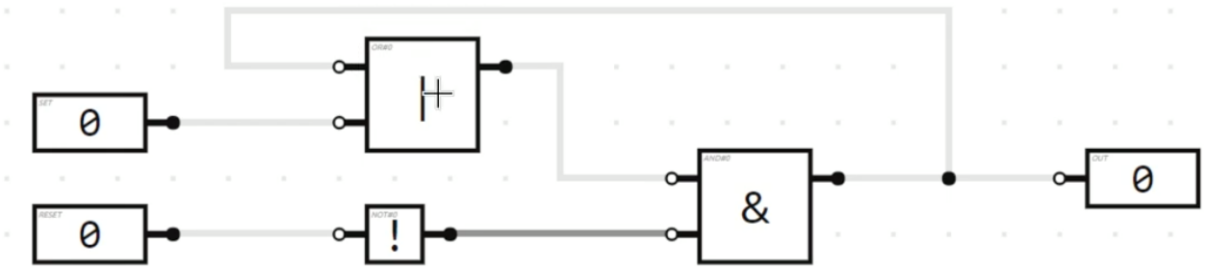
Для этого нужна подача энергии откуда-то. То есть нам нужна память, которая хранит определённые биты. Придумаем такую память. Если мы посмотрим внимательно на эти две схемы:



Если мы выход с операции ИЛИ замкнём себе же на вход, то как только подадим 1 на input, на выходе у нас тоже будет 1. Таким образом операция ИЛИ даст нам 1 на выходе и при этом поступит сама себе на вход. Теперь, как только мы будем на input включать 0 или 1 у нас на выходе будет замкнута 1. *Чем не память одного бита? Вполне память.*

Тоже самое мы можем сделать с гейтом И. Если мы подадим на вход 0, то на выходе всегда будет 0. Сама себе на вход операция И также будет подавать 0. Поэтому если мы будем включать и выключать ток, то ничего не произойдет. У нас будет вечно храниться 0. *Чем не память для хранения нуля?*

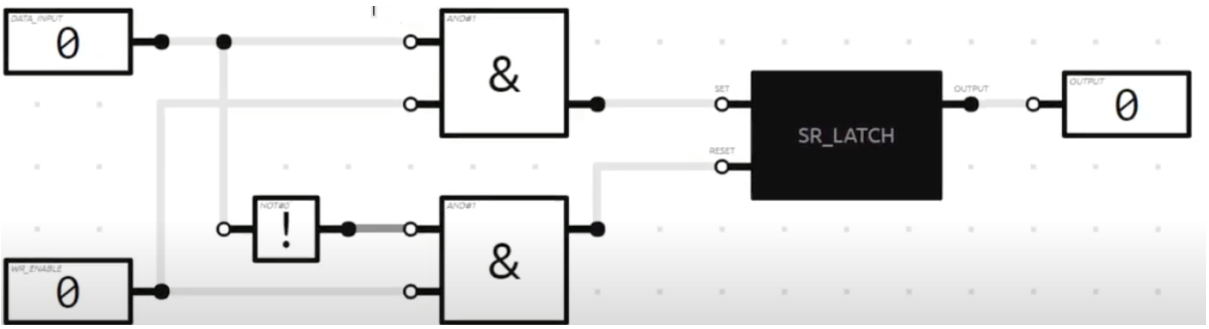
Но хотелось бы с этими значениями как-то играться, как-то скидывать, перезаписывать их и так далее. Инженеры придумали довольно хитрую схему, которая называется **RS_latch**.



Где эти два компонента практически последовательно соединяются друг с другом. Сначала идёт ИЛИ, потом И. И с выхода И ток подаётся на вход к ИЛИ.

Что происходит в этом случае? Как только нижний input, который называется reset, установлен в 0 (тока нет). Операция НЕ возвращает 1. Подавая 1 на верхний input, мы получаем 1 на выходе, где она записывается. Если мы подаём на верхний input 0, то уже ничего не происходит. У нас на выходе будет постоянно находится 1. Но последний input, нижний input недаром называется reset (сброс). Как только мы пододим на него 1, на выходе всё сбросится в 0. Снова подаём 0 на reset и можем записывать 1, которая опять ничем не сбросится, пока не используем снова reset и так по кругу. Это называется reset set latch. *Latch в переводе с английского означает защёлка*. Она защелкивает состояние. В некоторой литературе это также называется **RS_trigger**.

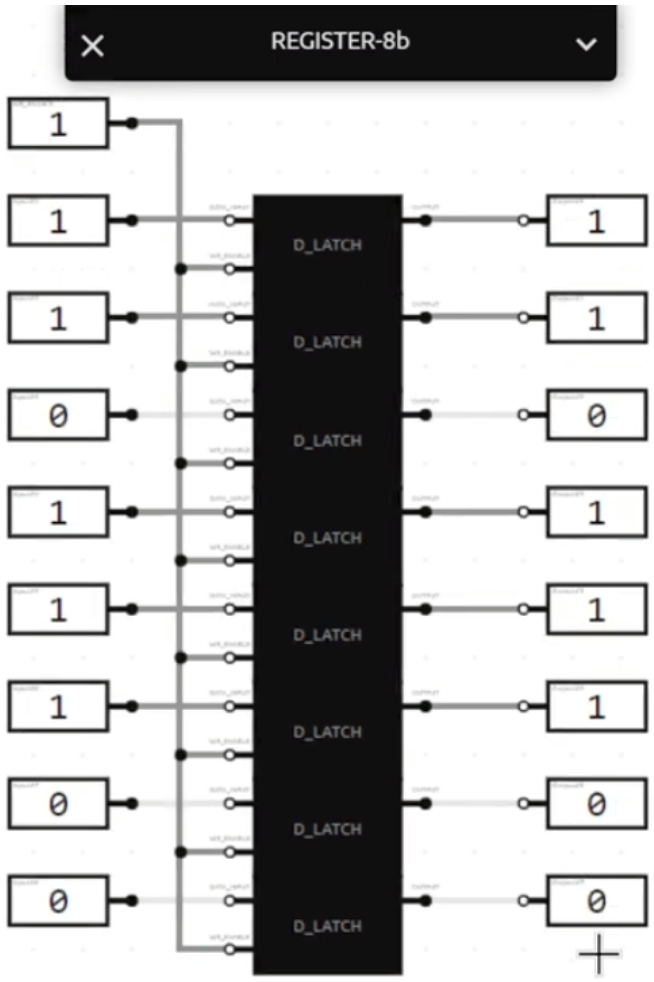
Если мы эту конструкцию ещё усложним, потому что reset set latch интуитивно не очень понятно. Нам хочется подать какие-то данные на вход и либо записывать их, либо нет. Для этого придумали **D_latch (D_trigger)**:



Он состоит из **SR_latch**, перед которым находятся два параллельно подключенных гейта И. Верхний input теперь называется data input, на него мы подаём некоторую информацию (данные). А нижний input — write enable, который позволяет или не позволяет записывать на output наше значение. Как только у нас в write enable стоит 0, нам запрещено что-либо записывать и хранить на output. Поэтому если мы подаём 1 или 0 на data input, то ничего не происходит. Но как только мы подаём напряжение на write enable, оно поступает на два гейта И. После этого, как только мы устанавливаем на data

input 1, у нас на выходе записывается 1. Или 0, если мы записываем 0 на data input. То есть мы можем записать 1, снова убрать write enable в 0 и на выходе будет хранить 1 независимо от того, что подаётся на data.

Поэтому D_trigger, выступает в роли хранителя 1 бита благодаря разрешению записи или его отсутствию. Как только разрешение записи есть, мы можем хранить любой бит. Но одного бита довольно мало для нашей задачи. Как нам хранить 8 битное число? Для этого можно собрать схему Register-8b.



Как видите, мы можем самый верхний write enable одновременно подать сразу на все 8 D_latch. Если мы пододим 1 на все 8 triggers, у нас это число запишется и будет храниться в этом триггере. Убираем разрешение записи, оно подключено параллельно и убирается сразу со всех триггеров. Теперь какое бы число мы снова не подавали на входы, оно никак влиять не будет. Мы его не записываем, а хранится только заранее записанное число.

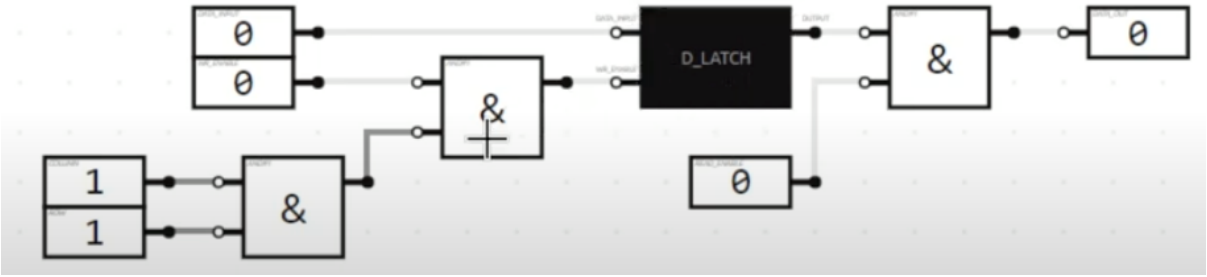
Такая память получается довольно громоздкой. Для 8 бит нам нужно 16 жил, как минимум, чтобы подвести 8 проводов на input и 8 проводов на output. Но если у нас, допустим, стоит задача хранить 128 битное число или 256 битное

число, это получается нам надо уже 512 проводов 256 на input и 256 на output. Плюс один провод для write enable всегда можно подавать сразу на все D-триггеры. Это довольно дорогое решение. Поэтому такая память (регистровая) стоит очень дорого. Её очень дорого делать. Но зато к ней очень быстро можно обращаться, практически мгновенно записывать и доставать оттуда числа.

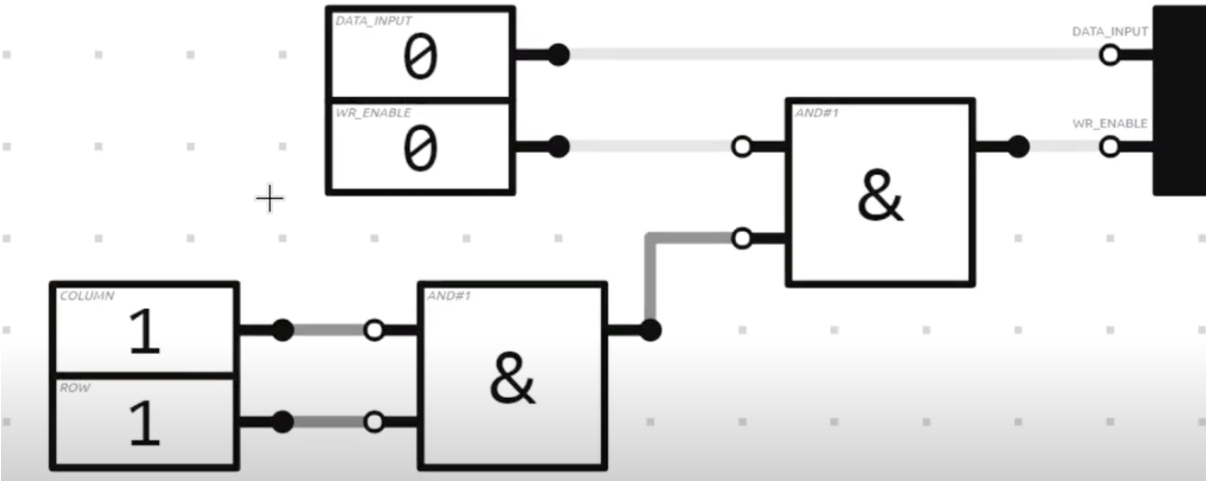
Чтобы удешевить этот момент, инженеры пошли дальше. И придумали хранить память в виде таблицы, в нашем случае MEM_16-bit. То есть мы можем каждый отдельный бит записать в ряды и колонки. Допустим, у нас будет 4 бита для хранения в рядах и 4 колонки. Итого 16 бит. При этом к каждому биту мы можем обращаться по координате (номеру ряда и колонки, которую мы подаём). Мы можем туда подать write enable, сложить его через гейт И и получить 1, только в том случае, когда у нас совпадает номер колонки и номер ряда, который мы хотим выбрать для записи.

[00:36:23]

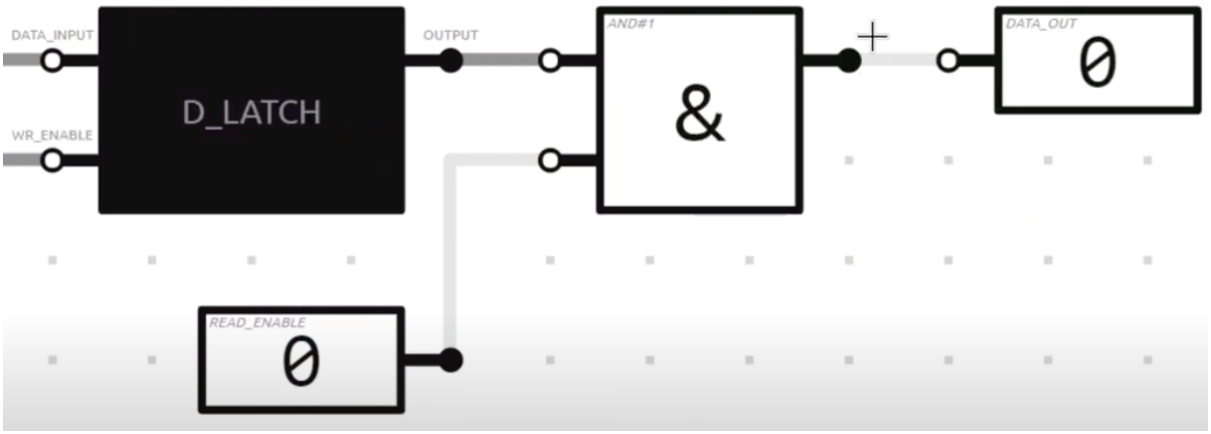
Из чего у нас здесь состоит отдельный бит?



Если мы рассмотрим всё это покрупнее, то у нас на input подаётся номер колонки и на row подаётся номер строки.



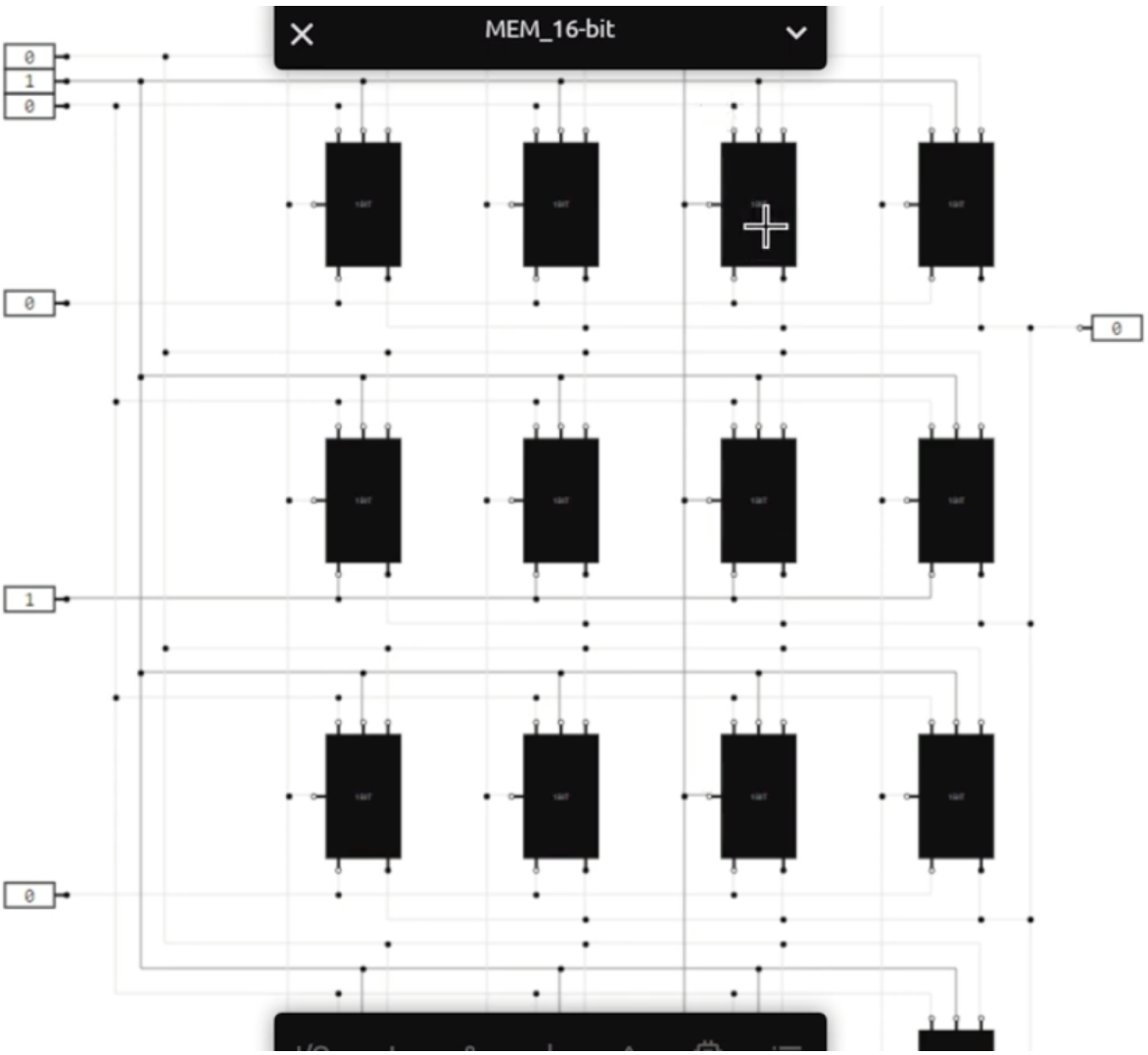
Как только подаём на них две единички, мы можем их сложить дальше с ещё одним гейтом И, который подаёт write enable. Получается на D триггер подаётся write только тогда, когда у нас и на row, и на column подаётся 1 (и на строчку, и на столбец). Таким образом, мы можем выбрать только 1 бит, подать на него write enable и записывать какую-то информацию.



Более того, если мы сюда положим 1 в AND, то мы можем подавать Read Enable и считывать то, что храним в D триггере. Если Read Enable нет, то мы и считать не можем. А если Read Enable есть, то мы считаем то, что здесь хранится. В нашем случае хранится 1 и у нас на data out 1. А если будет хранится 0, то на data out будет 0.

Но это хранение 1 бита с помощью колонки и строчки. Мы можем упаковать это в компонент, где на вход будет подаваться номер колонки (column), номер строки (ROW). Read Enable и Write Enable, если мы захотим что-то считать или записать в этот бит. И data input — это данные, которые мы туда записываем.

Таким образом, разместив 16 бит в ряды и колонки, мы можем устанавливать 1 на определённые колонки и строки, к которые мы хотим обратится.



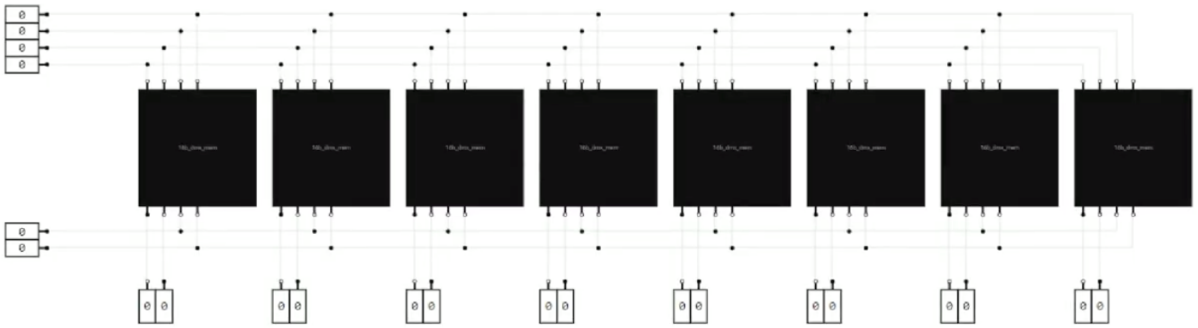
Важно понимать, что Date Input падает на все 16 бит, но только в бите, который находится во второй строке и в третьей колонке, Date Input запишется в наш Dead Trigger, который находится внутри. Потому то именно эту строку и колонку мы выбрали 1.

Если мы захотим все это прочитать, то можем поставить Read Enable. Он упадёт на все 16 бит, но достанется только оттуда, где row и column равны 1.

Запомним!

- Write Enable позволяет или запрещает нам записывать.
- Column и ROW позволяют обратиться к определённому биту в таблице.
- Read Enable позволяет или запрещает считывать. Но только из того бита, к которому мы обратились.

Итак, если мы разместим 16 битные кусочки памяти последовательно друг за другом, то сможем хранить 1 байт, но как бы размазывая его по этим 16 битным ячейкам.

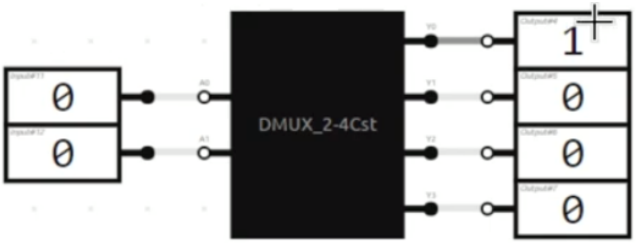


Эти биты мы можем хранить по одинаковому адресу. То есть, если мы обращаемся ко второй строке и к третьей колонке, то можем обратиться одновременно ко второй строчке и третьей колонке к каждому из этих компонентов и записать внутрь по одному биту. Потом мы можем их прочесть, включив Read Enable сразу на все D-триггеры, мы можем последовательно достать биты, которые хранятся в каждом элементе. Единственное, чтобы подать какой-то адрес, строчки или колонки, мы теперь можем воспользоваться мультиплексом или демультиплексор.

[00:41:32]

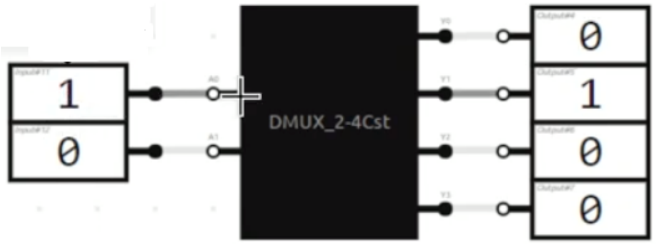
Что такое мультиплекс или демультиплексор?

На самом деле, это просто устройство, которое возвращает всегда одну 1 в каком-то определённом бите, которое обозначает номер этого бита в десятичном значении.



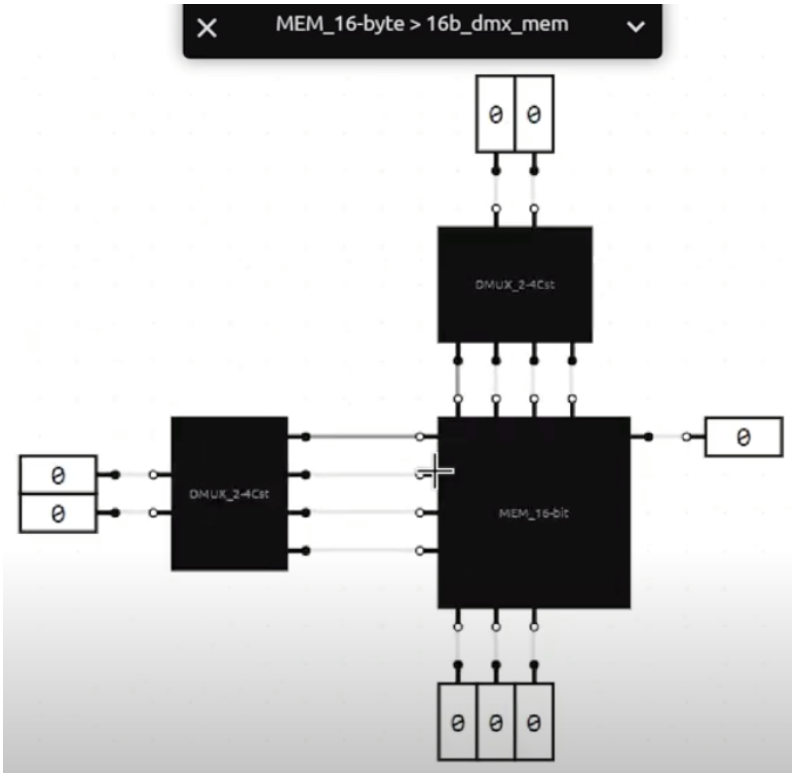
То есть, если у нас на вход подаётся два 0, в десятичном значении — это 0. Соответственно, у нас в нулевом элементе будет 1.

Как только мы подаём 0 1 (в десятичном значении — 2), у нас второй бит будет в единичке с правой стороны.



И так далее. Такая схема называется демультиплексор. То есть, подавая в двоичном значении какое-то число, с другой стороны будет гореть только одна строчка, которая будет соответствовать номеру этого числа в десятичном представлении. Таким образом, мы можем подать напряжение благодаря всего лишь двум битам на одну из четырех наших строчек. И записать адрес строки с помощью 2 бит. У нас четыре строки, с помощью 2 бит записываем 4 различных значения. Делаем тоже самое для номера наших столбцов. Мы можем записать адрес памяти в виде 4 бит, где первые два бита — это адрес строки, а вторые два бита — это адрес столбца. Ставим два мультиплексора и подаём напряжение только по определённой строчке и по определённому столбцу.

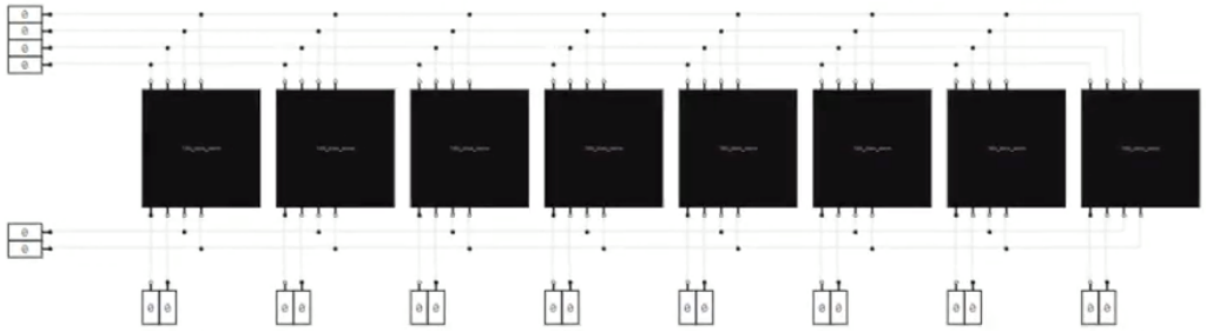
Итак, благодаря demultiplexer мы можем в каждом элементе нашей памяти, который хранит 1 бит, обращаться таким образом.



У нас есть demultiplexer для строчек, demultiplexer для колонок. Подавая сбоку на вход определённые значения, мы можем подавать напряжение на одну любую строчку. То же самое мы можем сделать и сверху.

Таким образом, с помощью demultiplexer мы можем вместо 16 проводов всего 4 проводам выражать адрес в виде координаты отдельного бита, в котором мы можем опять-таки либо писать, либо читать определённую дату (бит, информацию).

Например, разместив 8 таких элементов друг за другом, мы можем записать 1 байт, как бы размазав его на все эти 8 элементов. Значит, мы можем каждый отдельный бит хранить в ячейке.



Обратите внимание! Мы можем произвольно обращаться по любому адресу и доставать любое значение по любому адресу. Это называется random access memory или **память с произвольным доступом (она же оперативная память)**. Теперь вы знаете, как она работает. Все компоненты работают благодаря простым логическим гейтам, которые внутри устроены благодаря транзисторам. То есть память работает благодаря тем же транзисторам, что и в ALU, благодаря которым выполняются математические и логические операции.

[00:49:43]

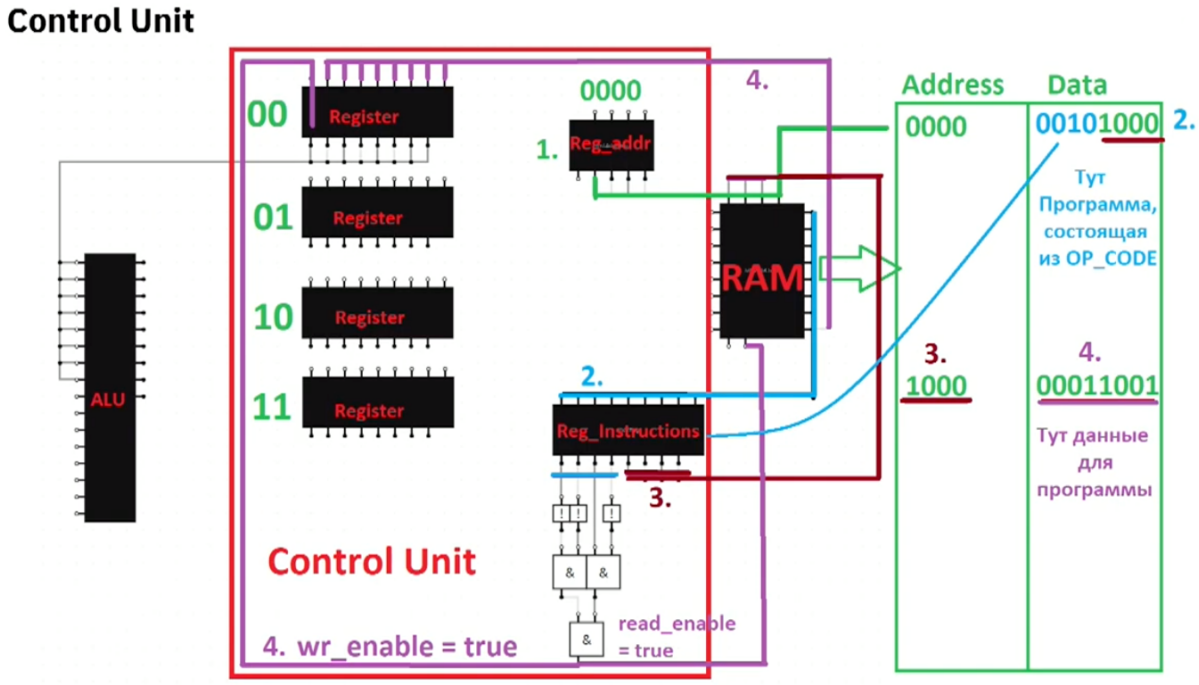
Control Unit

Теперь попробуем собрать всё воедино.

- Есть ALU, который выполняет математические и логические операции.
- Есть регистры — это память, которая хранит биты последовательно и может очень быстро выдавать их значение.
- Есть RAM — оперативная память. К ней мы обращаемся по определённому адресу. Достаём значение какой-то ячейки по определенному адресу.

Как все это взаимосвязано и как увязать с тем, что мы хотим всем этим управлять, подавая какие-то инструкции.

Допустим, у нас может быть 4 регистра, из которых мы можем доставать данные и подавать на ALU. А так как их четыре, мы можем дать им адреса: регистр 00, регистр 01, регистр 10 и регистр 11. Более того, у нас есть ещё вспомогательный регистр. Первый регистр (00), это регистр адресов, в котором хранится адрес текущей исполняемой инструкции. Также есть регистр самих инструкций, который проверяет, что должна делать инструкция, и выполняет ее. Есть оперативная память.



Как только мы включаем компьютер, у нас значение регистра адресов устанавливается в 0. Эти нули подаются на чтение из RAM. То есть идёт обращение к адресу 0000 в RAM. Мы обращаемся и получаем инструкцию, допустим, она есть в RAM.

Теперь подробнее рассмотрим инструкцию. Инструкция состоит из двух частей, в нашем случае 00101000. Первые 4 бита могут означать действие, что необходимо сделать. Например, наша инструкция 0010 говорит, что надо обратиться к регистру 00 и 10 означает, что надо записать в регистр 00 значение, которое лежит в RAM по адресу 1000. То есть вторая половина это адрес (operation code), к чему мы применяем инструкцию. Получается у нас инструкция разбивается на две части. Первая часть — operation code, а вторая часть — адрес в RAM, к которому мы применяем operation code.

Итак, у нас 0000 падают на RAM. Из RAM достаётся значение 00101000 и считывается в наш регистр инструкций. Регистр инструкций устроен хитрым образом. Первые 4 бита он сразу передаёт на чтение и отдаёт на специально

хитро устроенные гейты, которые возвращают 1 только в том случае, если они совпадают с определённым операционным кодом. В нашем случае под operation code надо построить следующий гейт: 2 гейта НЕ, третий гейт пропускаем и четвёртый гейт НЕ. Тогда в нашем случае, если подаётся значение 0010, которое берётся из нашей первой половины. Мы первые 2 бита инвертируем, получаем 11. Третий бит остается какой есть (тоже 1). И в четвёртом бите тоже 1, у нас 0 инвертируется на 1. Применяя логическое И ко всем этим битам мы получаем 1. И заметьте ещё раз, единичку мы получаем только тогда, когда у нас opcode равен 0010. Таким образом, 1 можно подать дальше:

- в качестве read enable на RAM, чтобы оттуда можно было читать;
- в write enable на наш регистр с адресом 00.

Напоминаю последние 4 бита — это адрес из RAM. Если мы их пододадим опять на RAM, то из RAM достанется значение по адресу 1000. И оно отправится на запись в регистр 00. Таким образом, мы сделали простую вещь, обратились к ячейке в RAM, положили в регистр 00 и в регистре 00 у нас лежат биты, которые лежали до этого в RAM.

Важно понять сам принцип, что благодаря хитро устроенным гейтам мы можем подавать 1 (ток), куда нам необходимо и управлять токами, битами, записывать туда, куда нам необходимо.

Например. Мы можем сделать настолько хитрый Opcode, который дальше подаст 1 на наш ALU в качестве Opcode сложения. Потом подать 1 туда, чтобы записать результат в регистр 10. Более того, можем придумать Opcode, который считывает регистр 10 в наш RAM. Таким образом, мы в сейчас придумали устройство, которое просто складывает два числа. Более того, мы не только придумали устройство, но и написали программу, хранящуюся в RAM в виде 0 и 1, которая берёт два числа по определённому адресу и складывает их между собой.

Все это мы можем упаковать и назвать Control Unit, а именно: регистры памяти, вспомогательные регистры адресов, регистры инструкции и так далее. Control Unit будет заниматься тем, чтобы контролировать работу, считывать определённые инструкции по определённому адресу, доставать из них Opcode, применять их к определённым адресам в RAM. И всё это с помощью тока и обычных транзисторов.

Более того, соединив Control Unit с нашим ALU, мы получим — **CPU (Central Process Unit)**, который будет связан с RAM и будет выполнять инструкции, то есть программу, которая в RAM загружена.

[00:57:23]

Устройство Clock

Стоит отметить! Есть устройство, как clock, которое позволяет включать те или иные схемы в какой-то определённый момент времени, чтобы всё это дело выполнялось последовательно. До этого я сам управлял всем и рассказывал вам в какой момент времени берутся ячейки из определённых адресов, берутся те самые Opcode и так далее. Здесь управляющим выступает clock. Clock просто подаёт напряжение на определённой части схемы, включая или выключая их, тем самым заставляя работать определённые места в CPU. Итак, включение и выключение определённых частей схемы в нашем Central Process Unit можно назвать итерацией.

Включение и выключение таких итераций можно делать с определённой частотой. Например, если мы будем делать это с частотой 1 раз в секунду, это будет 1 Гц, довольно медленно. За одну секунду мы сочетаем одну ячейку из памяти RAM, потом за вторую секунду мы отправим её в регистр 00. Но это было бы очень долго, если бы так процессоры работали. Современные процессоры работают на частотах в несколько Ггц и могут выполнять несколько миллиардов операций в секунду.

Если представить, что одна итерация выполняется за 1 секунду, то открытие просто странички Хабра заняло бы 500 лет. Кстати, на Хабре есть очень интересная статья «Насколько быстры компьютеры», которая предлагает поразмышлять на эту тему. Советую её найти и почитать.

Итак, мы собрали практически с нуля из транзисторов целый CPU с оперативной памятью, который может выполнять простую программку по сложению двух чисел. На самом деле, оперейшн кодов в процессорах очень много, несколько тысяч запросто может быть в процессорах. Более того, регистры там более большие, они далеко не 8 битные. Оперативная память у нас тоже не 16 байт, а гораздо больше. Есть и 32 гигабайта, и 64 гигабайта в серверах используется.

[00:59:45]

Заключение

Теперь мы знаем, как с помощью ноликов и единичек хранятся данные в нашем компьютере. Как эти данные реального мира вообще преобразовываются в нолики и единички. Более того, с помощью, допустим, тех же самых ALU мы можем выполнять операции над этими данными.

Складывать, вычитать, сравнивать, всё что угодно. ALU гораздо более сложный внутри, чем мы рассмотрели. И все это работает и контролируется с помощью CPU, который является сердцем компьютера.

Теперь вы знаете, как устроены все эти процессы внутри компьютеров, как они протекают и как помогают исполнению ваших программ. Надеюсь, эти знания помогут вам в будущем. Вы сможете более глубоко понимать и осознавать, как ваш код работает внутри, компьютер исполняется на CPU. Ну, как минимум, закроется темное пятно на карте ваших знаний.

На этом всё. Был рад с вами знакомству и до новых встреч. С вами был Игорь Негода.