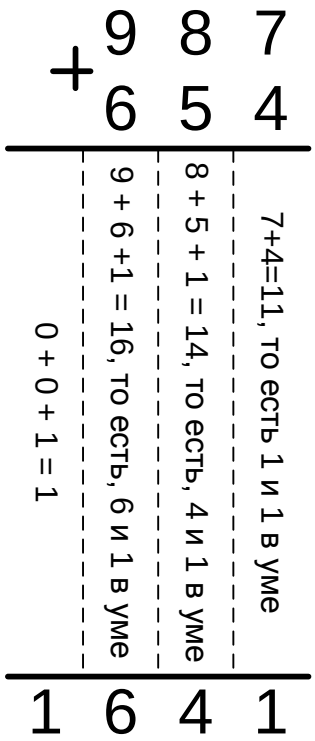
Сумматор

Ну что ж. Давайте кратенько познакомимся с вычисляющими функциями. Для этого вспомним, как происходит сложение в столбик:



Итак, мы пользуемся таблицей сложения, причём в сложении участвуют три числа:

1) Из первого слагаемого
2) Из второго слагаемого
3) Перенос из предыдущего числа (при счёте на бумажке, мы называем это «в уме», в электронике - перенос)

Таблица будет содержать 10^3 строк, мне все их писать лень, давайте запишем какие-нибудь характерные…

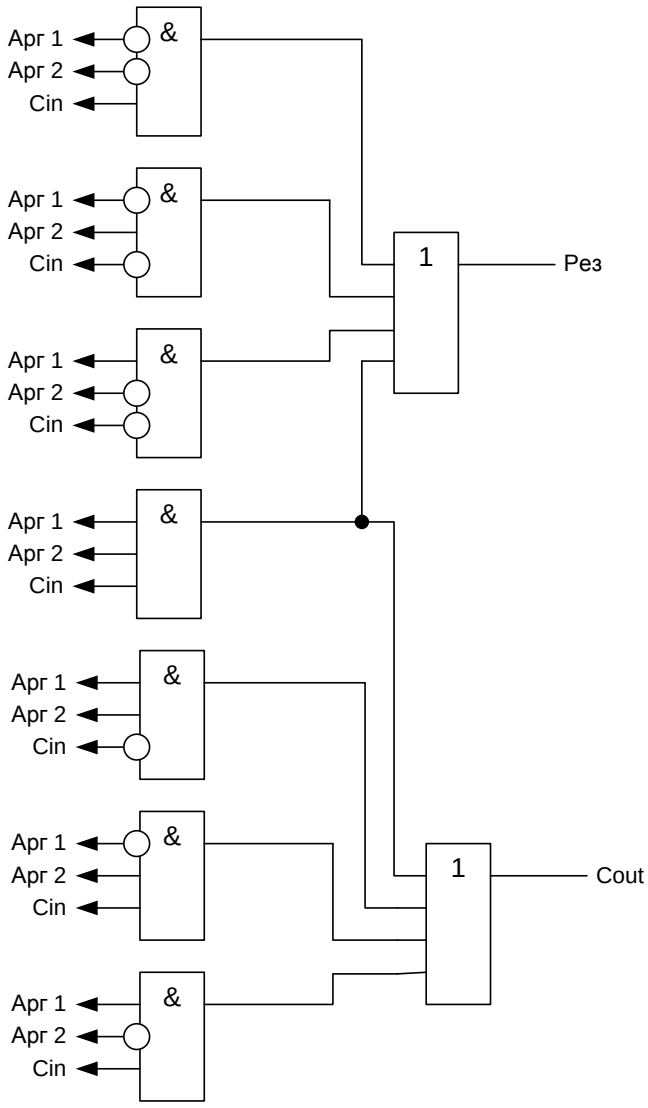
1 слагаемое	2 слагаемое	Входной перенос	Результат	Выходной перенос
0	0	0	0	0
0	0	1	1	0
0	0	2	2	0
5	3	1	9	0
5	3	2	0	1
5	3	3	1	1
9	9	7	5	2
9	9	8	6	2
9	9	9	7	2

Если что-то не понятно – попрактикуйтесь в сложении в столбик до полного понимания. Главный тезис – все числа у нас однозначные. Лишний знак уходит в перенос (остаётся в уме).

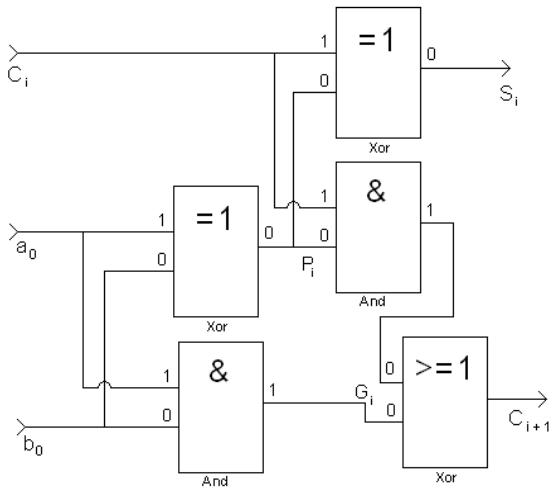
А для двоичных чисел всё проще. Там таблица будет содержать всего 2^3=8 строк и выглядит так:

1 слагаемое	**2 слагаемое**	**Входной перенос**	**Результат**	**Выходной перенос**
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Ну, строить логические функции по таблице истинности, мы уже умеем. Получаем:

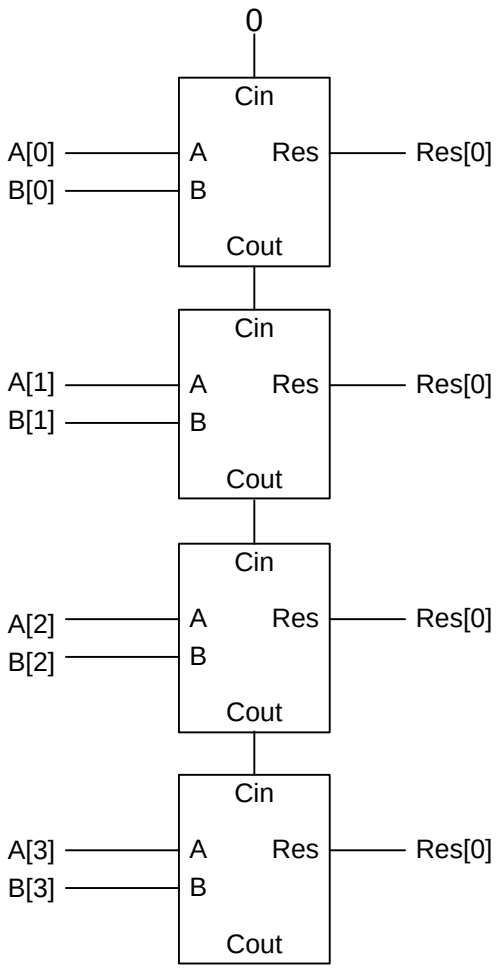


Если порыться в Интернете, поискав слово «Сумматор», то схема будет чуть иной:



Таблицы истинности у этих вариантов совпадают. Просто раньше, когда всё делалось на мелкой логике, каноническая схема содержала меньше элементов, посему была удобнее (если, конечно, не использовать микросхему сумматора, а паять его на рассыпухе). В наше время, если мы и будем делать сумматор, то на ПЛИС. Как мы уже (надеюсь) помним, CPLD содержат только базис И-ИЛИ-НЕ, так что компилятор всё равно преобразует каноническую схему к тому варианту, который нарисовал я. Что же до FPGA, то там любая логика реализуется в виде таблицы истинности, а значит, каноническая схема преобразуется к таблице. Посему, как бы мы ни нарисовали, всё равно результат (после обработки компилятором) будет один и тот же. А таблицу истинности превращать в функцию, на мой взгляд, проще именно моим путём. Тем более, что он позволит оценить сложность реализации функции в реальной CPLD…

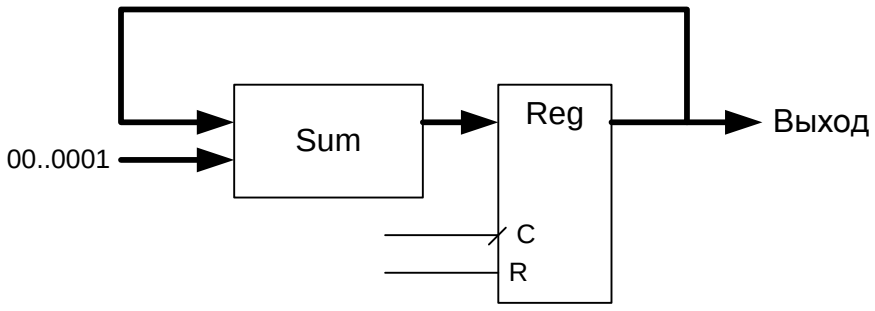
Ну, а теперь – сколько разрядов нам надо сложить, столько сумматоров и объединяем. Допустим, мы делаем 4-разрядный процессор. Прекрасно…



Всё, теперь всё будет само вычисляться… Правда, у этой схемы есть один недостаток. Перенос в ней распространяется примерно так же, как и в асинхронном счётчике. То есть, сначала сформируется перенос для бита 0, затем он учтётся для бита 1, только тогда сформируется перенос для бита 1 и т.д. Существуют сумматоры с ускоренным переносом, познакомиться с которыми любой желающий сможет при помощи поисковых систем в Интернете.

Пример использования сумматора

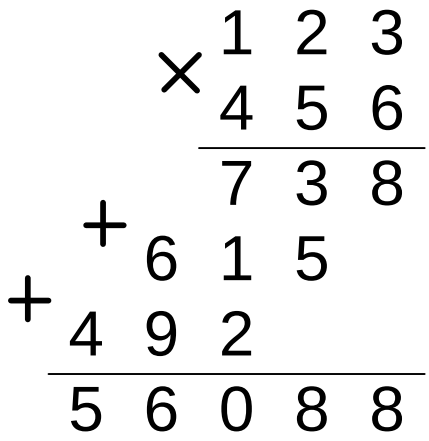
В своё время, я обещал показать один из вариантов реализации синхронного счётчика, в котором используется сумматор. Пришла пора сделать это.



Сначала регистр сбрасывается в ноль (для этого у него есть вход R). Сумматор сложит этот ноль с единицей, так что на входе регистра образуется единица, которая будет защёлкнута по ближайшему тактовому импульсу. При этом, сумматор сформирует двойку, которая защёлкнется по следующему импульсу, породив тройку на выходе сумматора. Если сумматор построен по схеме с ускоренным переносом, то очередное значение на входе регистра будет образовываться достаточно быстро, что повысит быстродействие счётчика по сравнению с асинхронным вариантом. Именно такие счётчики строят компиляторы языка VHDL при создании схем для ПЛИС. Хотя, данная схема является не единственно возможной реализацией для синхронных счётчиков. Кому интересна альтернативная реализация – смотрите, например, http://www.transwarp.ch/datasheets/MC14161.pdf.

Основная идея умножителей

Очень кратко рассмотрим идею построения умножителей, не вдаваясь в детали реализации. Давайте вспомним, как умножаются трёхзначные десятичные числа в столбик.



Мы видим следующие операции:

1) Умножение (согласно таблице умножения)
2) Столько сложений, сколько знаков в числе
3) Сдвиг перед каждым сложением

Собственно, в двоичной арифметике всё так же. Таблица умножения до безумия проста:

1 множитель	2 множитель	Результат
0	0	0
0	1	0
1	0	0
1	1	1

Если вспомнить таблицу истинности для операции «Логическое И», то можно заметить, что это и есть умножение.

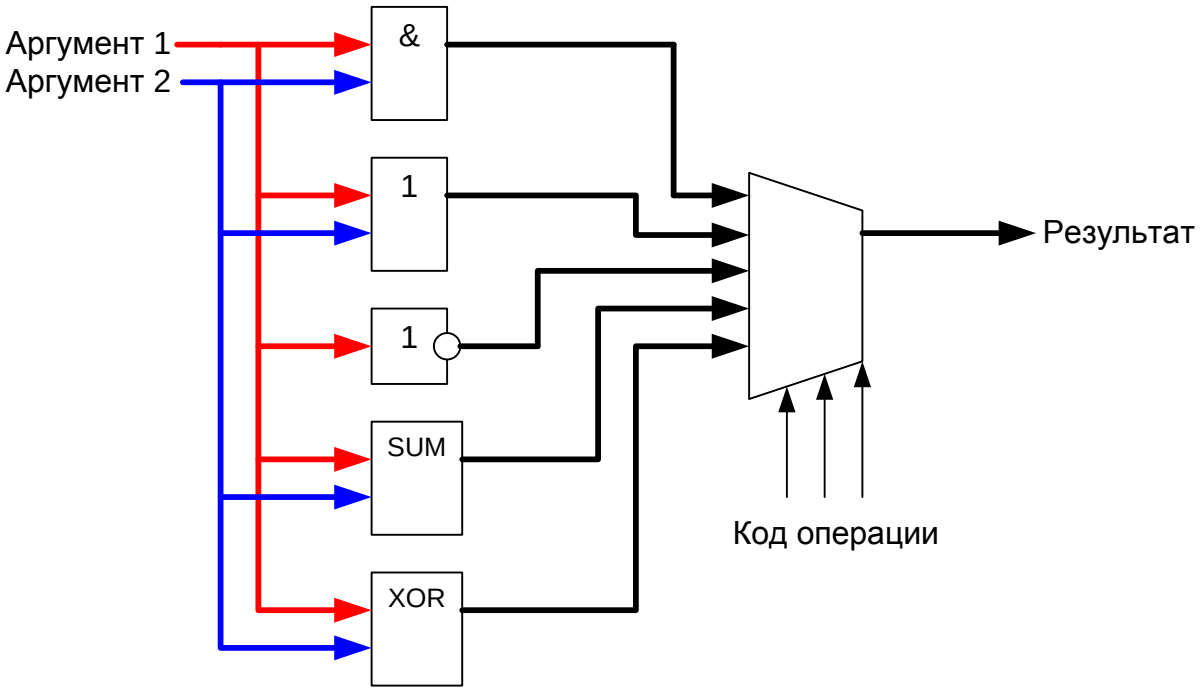
Вот со сложениями сложнее. Для 8 разрядных чисел надо выполнить 8 операций сложения, для 16 разрядных – 16, для 32 разрядных – 32. А сумматор соответствующего

размера, как мы видели выше – не самая простая функция. А надо 32 сумматора… Уууух! Поэтому в старинных процессорах операции умножения нет вообще. В более современных, операция умножения есть, но для её выполнения используется один сумматор и регистр сдвига. Посему, она выполняется за 8, 16, 32 такта (смотря какая разрядность у числа). Вспомните, в архитектуре x86 операция умножения очень длительная. Теперь вы знаете почему.

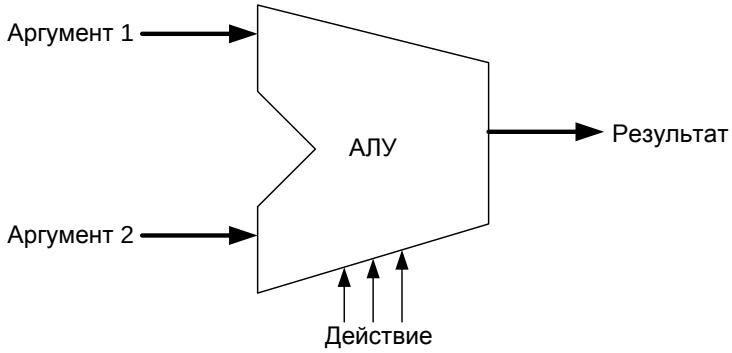
Первые микросхемы, где были вынуждены использовать умножение, выполняемое за один такт (за счёт большого числа сумматоров на кристалле) – это сигнальные процессоры, там без этого никак. Затем однотактные умножители появились в дорогих FPGA. В последнее время, даже дешёвые FPGA содержат один-два умножителя, выполняющих действие за один такт. Но логическую сложность операции следует всегда помнить. Она очень-очень высока. Ну, и теперь понятно, почему Pentium так тормозит на простейшей, казалось бы, операции.

Арифметико-логическое устройство

Ну вот. И напоследок, давайте рассмотрим сердце любого процессора – арифметическо-логическое устройство. Я нарисую его не в самом оптимальном, но зато в самом понятном виде.

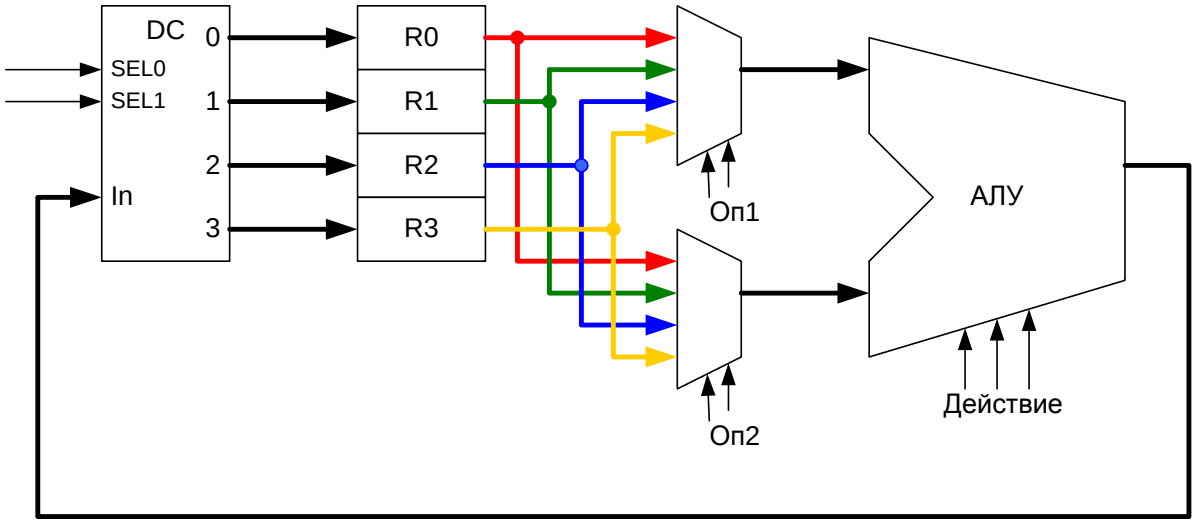


Оптимизированное АЛУ будет отличаться тем, что там часть функций выполняется одними и теми же логическими элементами (примерно, как у меня в сумматоре один элемент И использовался и для результата и для переноса), но тогда всё будет не так наглядно. А в целом – задача АЛУ состоит в том, чтобы обрабатывать два аргумента в соответствии с двоичным кодом операции. Если код равен 000, то у нас на выходе будет результат операции «Логическое И». Если код 001 – «Логическое ИЛИ», если 010, то «Логическое НЕ», если 011, то сумма двух аргументов, если 100 – «Исключающее ИЛИ». Список операций, выполняемых АЛУ можно расширять, суть от этого не изменится. Имеются блоки, которые выполняют какие-то действия, имеется выход, на который выдаётся результат запрошенной операции. На схемах АЛУ обычно обозначают так:



Использование АЛУ в процессоре

Давайте построим гипотетический процессор. Будем считать, что читатель знаком с основами его работы, то есть, слышал про ассемблер. Если не знаком – потом исправим ☺. Для простоты рисунка, у него будет всего 4 регистра – R0, R1, R2 и R3. Любой регистр может быть использован или как первый операнд или как второй. На АЛУ содержимое попадёт через мультиплексор. Результат из АЛУ через демультиплексор будет помещён в регистр-результат.



Собственно, такая схема является излюбленной у разработчиков RISC процессоров. Там сначала при помощи отдельных команд аргументы загружаются в регистры, а затем – обрабатываются в АЛУ, а результат – снова попадает в регистр.

Теперь становится понятно, как устроена машинная команда. Если посмотреть документацию на машинный код, то там видны чёткие группы битов. Какие-то задают регистр-источник, какие-то – регистр приёмник, какие-то – регистр-результат, какие-то – код операции. На самом деле, эти биты ничего не задают. Они просто тупо и цинично проходят на управляющие входы мультиплексоров, АЛУ и демультиплексора. А те уже – пропускают сигнал, куда следует…

Двоичный код машинной команды

«Операция с АЛУ»	Оп1	Оп2	Рез	Действ	Зарезервировано

Заключение

Ну вот. Примерно так, если быть очень кратким, можно наметить общие черты ответа на вопрос, который, напомню, звучал так: «**С давних лет никак не могу понять, каким образом из кучи логических элементов, которые в зависимости от того что им дают на вход, выдают разные значения на выход, можно собрать что-то, что выполняет более сложную функцию, чем транзистор**».

Теперь вы в общих чертах вы можете читать чужие схемы, если Заказчик чего-нибудь пришлёт, для вас это уже будет не китайская грамота. Ну, и что-то своё уже можно начинать творить. Не обязательно с паяльником, можно – в модели, например, при помощи среды Proteus или, скажем, пары Quartus/ModelSim.

Творить своё можно на основе знакомства с чужим. Могу посоветовать полезные сайты http://www.rlocman.ru/ и http://radiokot.ru/. В общем, было бы желание, а возможности найдутся. А пока, цикл статей считаем законченным. До новых встреч…