NSU-2023-T06L2e02

Вы замечали, что итоговая сумма счета или чека часто используется для проверки вычислений. Да, в денежных документах она нужна, главным образом, для того, чтобы вы могли узнать, сколько всего денег потрачено, но часто она также используется для проверки целостности данных кассы или бухгалтерии. Если какая-то из строчек над итогом была изменена после оплаты, то в сумме они не смогли бы произвести тот же итог. Да, теоретически возможно убрать из одной строчки ровно столько, сколько было добавлено к другой, и это не приведет к изменению итога. Но если была изменена только одна строка, это может быть немедленно обнаружено пересчетом итога. Для случайных изменений, одно изменение более вероятно, чем два согласованных.

Интересно, что очень похожая техника используется при хранении данных и в сетях. Для проверки целостности битовой строки, все биты объединяются операцией XOR, и результирующая 1-битовая контрольная сумма добавляется к данным. По очевидным причинам, контрольная сумма не может сойтись если любой одиночный бит, включая и саму сумму, по каким-то причинам изменится впоследствии. Но если изменятся два бита, контрольная сумма может по-прежнему сойтись.

Есть хитрый математический трюк, известный как циклический избыточный код (Cyclic Redundancy Check, CRC) который вычисляет более сложную многобитовую контрольную сумму. Математическое обоснование этого кода мы не будем приводить здесь из-за недостатка места, но его реализация так проста, что она не требует знания сколько-нибудь продвинутой математики. Вместо представления “правильного” математического обоснования, мы проиллюстрируем эту распространенную технику по аналогии.

Длинное деление и полиномиальный остаток

CRC вычисляет так называемый *полиномиальный остаток*, но он вычисляется таки образом, что это почти не отличается от длинного

деления двоичных чисел. Давайте вспомним, как это происходит, на примере деления 00110101/101 (53/5 в десятичной записи).

0	0	1	1	0	1	0	1
1	0	1					
0	0	1	1	0	1	0	1
	1	0	1				
0	0	1	1	0	1	0	1
		1	0	1			
0	0	0	0	1	1	0	1
			1	0	1		
0	0	0	0	1	1	0	1
				1	0	1	
0	0	0	0	0	0	1	1
					1	0	1
0	0	0	0	0	0	1	1

Алгоритм деления определяет, сколько раз делитель “входит” в красную часть делимого. В случае двоичных чисел, это может быть только 0 или 1. Если ответ 0, мы ничего не делаем а если 1, мы вычитает делитель из соответствующей части делимого. Потом мы сдвигаемся к следующей цифре, и т.д., пока делитель не наложится на правую (младшую) часть делимого и больше не может сдвинуться. В этот момент, красная часть и оказывается остатком от деления.

В приведенном примере, на первом шаге $001 < 101$, поэтому ответ 0 и мы продолжаем, затем мы получаем $011 < 101$ и мы продолжаем дальше затем мы получаем $110 > 101$ и мы вычитаем, $011 < 110$, продолжаем, $110 > 101$, вычитаем, $011 < 101$, пытаемся продолжить, но не можем потому что мы уже на правом краю. Мы получаем остаток 011. И, в самом деле, 53/5 дает остаток 3.

Отличие между этим и полиномиальным остатком, который нам нужен, состоит в том, что для вычисления последнего мы не вычитаем, но вычисляем побитовый XOR между красной частью делимого и делителем, если старший бит первого равен 1, иначе делаем только сдвиг. Поэтому мы проверяем только один бит, и если он поднят, мы вычисляем побитовый XOR, иначе ничего не делаем. В этом и есть вся разница.

Вот пример. Поскольку битовые строки теперь не являются числами, мы не можем сконвертировать их в десятичную запись для проверки. Давайте посчитаем полиномиальный остаток для 11100011 and 101:

1	0	0	1	0	1	0	0
1	0	1					
0	0	1	1	0	1	0	0
	1	0	1				
0	0	1	1	0	1	0	0
		1	0	1			
0	0	0	1	1	1	0	0
			1	0	1		
0	0	0	0	1	0	0	0
				1	0	1	
0	0	0	0	0	0	1	0
					1	0	1
0	0	0	0	0	0	1	0

Здесь, окончательное значение остатка, которое и есть нужная нам контрольная сумма CRC, равен 010. Заметьте, что на втором и последнем шагах старший красный бит был нулевым, поэтому XOR не был выполнен.

Дизайн

“Делитель”, который в случае вычисления полиномиального остатка называется “генерирующим полиномом” или GP, обычно приводится с удаленной обязательной старшей единицей (в предыдущем примере его надо было бы записать как 01). Это значение выбирается так, чтобы максимизировать зависимость контрольной суммы от каждого бита “делимого”. При некоторых выборах GP размером n бит, невозможно исказить два или менее бита в последовательности размера 2^n без нарушения контрольной суммы (действительно, очень полезно для обнаружения двойных ошибок). Но не будем погружаться в математические детали.

Ваша работа состоит в том, чтобы построить микросхему CRC для 16-битного GP 0x8005, или 1 1000 0000 0000 0101 (включая старший бит). Этот GP – практически используемое значение, широко применяемое в реальных устройствах.

Несколько наблюдений, которые помогут вам построить это несложное устройство.

1. Хотя в рассмотренных примерах делитель двигался, а делимое было неподвижным, было бы проще работать с неподвижным делителем и движущимся делимым. Делимое двигалось бы справа налево мимо делителя, выровненного со старшими $n+1$ битами, в данном случае n=16.

2. Чтобы сдвинуть делимое на одну позицию влево, вы можете использовать цепочку из шестнадцати 1-битовых регистров, один за другим. Каждый регистр копирует из предыдущего и последний берет следующий бит делимого из контакта `bits-in` на каждом такте. Это называется *сдвиговый регистр*, и он доступен в библиотеке *Memory*, но его использование запрещено правилами; постройте свой собственный! Выход всех 16 ступеней должен выдаваться на 16-битный контакт `CRC`, предоставленный на макетной плате.

3. Преимущество изготовления вашего собственного сдвигового регистра в том, что вы можете добавить логику *между* ступенями, обеспечив, чтобы биты в позициях 0, 2 и 15 (посмотрите на GP) были инвертированы перед сохранением. Это предложение почти выдает секрет предлагаемого нами дизайна, нам не следует больше ничего говорить!

Контакт `clk` предоставляет тактовую частоту для вашего дизайна. Управляйте им вручную для отладки вашей схемы, и убедитесь, что на одном шаге оно работает как надо. Потом отправьте ваше решение для тестирования роботом.

Как отправлять вашу работу на проверку

Не перемещайте входные и выходные контакты, потому что Logisim присоединяет к ним тестовую схему, основываясь на их положении, а не по имени (это неудобно, но мы ничего не можем с этим сделать). Если вы хотите, вы можете присоединить туннели к контактам и разместить другие концы туннелей в удобные для вас места на макетной плате. Таким образом, вы можете размещать вашу схему удобным для вас образом и, в то же время, быть уверенными, что тестирующий робот будет правильно подсоединен к схеме.

Проверьте устройство, нажимая входные контакты при помощи ручных контролов и записывая ваши наблюдения. Ответьте на это сообщение, присоединив файл схемы с вашим решением.