Работу выполняем в программе логического моделирования Logicly. Это сугубо учебная программа уровня детского сада.

Все модели заготовлены. Поэтому панель компонентов Вам не понадобится. Можете свернуть ее влево, нажав на вертикальный синий квадрат с треугольником. Файлы моделей открываем через меню File/Open.

Курсор графической оболочки находится в одном из двух режимов – Pan (клавиша P) и Select (клавиша S). Можно переключать режимы и через меню Tools, но лучше запомнить две кнопки. В режиме P «гоняем» возможно не лезущую в экран картинку туда-сюда. Масштаб схемы регулируется колесом мыши или ползунком справа внизу. В режиме S кликаем по переключателя и кнопкам, управляя состояниями входных сигналов. Выходные сигналы изображаются лампочками. Из удобств – можно выделить в курсором любой фрагмент схемы и заказать выдачу его таблицы истинности – самая правая кнопка на панели инструментов.

Моделирование запускается и останавливаетя через меню Simulate. После запуска моделирования иногда требуется привести схему в исходное состояние, подав на нее сигнал сброса. Особенно это касается схем с обратной связью типа JK и T триггеров.

Внутренность блока на схеме открывается в новом окне по двойному щелчку на нем. В этом окне можно изучать работу блока автономно, не думая об окружении. Важно, что закрывается это окно только по кнопке Close слева вверху.

Управление цветами и прочим упрятано в меню Edit/Application Settings.

1. Изучаем элементы комбинаторной логики

1.1. Элементарные вентили – gates.logicly

Уяснить логику функционирования вентилей NOT(И) ($y =!x$), AND(И) ($y = x_1x_2$) и OR(ИЛИ) ($y = x_1 + x_2$). Изучить их таблицы истинности. Обратить внимание, что наблюдение единицы на выходе AND позволяет однозначно предсказать состояния на всех его входах, в то время как наблюдение нуля не говорит о входах почти ничего. С вентилем OR все наоборот. Таким образом, состояние 1 на выходе AND и 0 на выходе OR является выделенными (особыми).

Проанализировать варианты реализации 4-входовых вентилей AND/OR «лестницей»

$$y = x_1x_2x_3x_4 = (((x_1x_2)x_3)x_4)$$
$$y = x_1 + x_2 + x_3 + x_4 = (((x_1 + x_2) + x_3) + x_4)$$

и «деревом»

$$y = x_1x_2x_3x_4 = ((x_1x_2)(x_3x_4))$$
$$y = x_1 + x_2 + x_3 + x_4 = (x_1 + x_2) + (x_3 + x_4)$$

Возможность альтернативных реализаций опирается на свойства коммутативности а ассоциативности функций ADN/OR. Какой из вариантов предпочтительнее по задержке ?

1.2. Тождества де Моргана – morgan.logicly

Между операциями AND/OR имеется двойственность, которая выражается тождествами де Моргана

$$!(x_1x_2) =!x_1+!x_2; \quad !(x_1 + x_2) =!x_1!x_2.$$

Поэтому

$$(x_1x_2) =!(!x_1+!x_2); \quad x_1 + x_2 =!(!x_1!x_2).$$

Иными словами, AND – это OR в инвертированной логике и наоборот. Убедиться в этом на примерах. Понять, что эти факты имеют место и при произвольном числе входов.

1.3. Комбинированные вентили – nandnor.logicly

Знакомимся с комбинированными вентилями NAND (NOT-AND/И-НЕ) – $y =!(x_1x2)$ и NOR (NOT-OR) – $y =!(x_1 + x_2)$. Важно усвоить существование двойственных вариантов их реализации (де Морган)

$$!(x_1x_2) =!x_1+!x_2; \quad !(x_1 + x_2) =!x_1!x_2;$$

Иными словами, NAND – это AND в инверсной логике на выходе или OR в инверсной логике на входе. Обратите внимание на варианты реализации инверторов на NAND и NOR.

У вентиля NAND выделенным состоянием на выходе является нуль – ему отвечает наличие единиц на всех входах, у вентиля NOR выделена единица – нули на всех входах. Ясно что вентили NADN/NOR могут иметь любое число входов. Однако реализовывать их «лестницей» или «деревом» не получается. Требуется лестничная или древовидная структура из ADN/OR с инвертором на выходе или слоем инверторов на входах.

1.4. Нормальные формы – normalforms.logicly

Всякая булева функция от n переменных $y = f(x_1, \ldots, x_n)$ может быть реализована в одной из двух нормальных форм – конъюнктивной (AND) или дизъюнктивной (OR).

В основе всего лежат понятия обобщенного конъюнктора и дизъюнктора. Элементарный конъюнктор $y = x_1x_2 \ldots x_n$ опознает комбинацию $(1, 1, \ldots 1)$ (все единицы) на входах и подтверждает факт ее наличия выделенным состоянием 1 на выходе. Введя слой инверторов на входах, можно заставить конъюнктор опознавать любую другую комбинацию. Пример конъюнктора, опознающего комбинацию $(0, 1, 0, 1)$ дан в модели. Имея на руках таблицу истинности произвольной функции, реализуем конъюнкторы, опознающие все комбинации входов, на которых эта функция принимает значение 1. Затем объединим их выходы по логике OR. Придем к нормальной конъюнктивной форме.

Обощенный дизъюнктор (см. модель) также опознает комбинацию входов, но подтверждает факт ее наличия выдачей выделенного нуля. Нужно опознать дизъюнкторами все комбинации входов, отвечающие нулям в таблице истинности функции, а затем перемножить их выходы по AND. Получится нормальная дизъюнктивная форма.

Для примера в упражнении даны реализации функции XOR (Исключительно ИЛИ/Сумматор по модуля два) в одной и другой формах. Обратите внимание, как переход к инверсной логике на внутренних шинах, позволяет обойтись комбинированными вентелями одного типа – NAND или NOR.

1.5. Минимизация нормальной формы – мажоритарный элемент – majoritary.logicly

Реализацию в нормальной форме часто удается упростить. В модели приведена реализация в нормальной конъюнктивной форме мажоританого элемента от трех переменных – функции, принимающей значение 1, если число единиц на входах больше

числа нулей:

$$maj(x_1, x_2, x_3) = !x_1x_2x_3 + x_1!x_2x_3 + x_1x_2!x_3 + x_1x_2x_3 =$$

$$= !x_1x_2x_3 + x_1!x_2x_3 + x_1x_2(x_3 + !x_3) =$$

$$= !x_1x_2x_3 + x_1!x_2x_3 + x_1x_2 = (!x_1x_2 + x_1!x_2)x_3 + x_1x_2 = XOR(x_1, x_2)x_3 + x_1x_2$$

Убедитесь в корректности работы исходной реализации и ее упрощенных форм.

1.6. Дешифратор – DX.logicly

Дешифратор – это система, которая преобразует заданный на n входах двоичный позиционный код адреса объекта в сигнал выбора этого объекта. Сигнал выбора появляется на одном из $N = 2^n$ выходов. Схемотехнически дешифратор – это просто набор всех возможных обобщенных конъюнкторов или дизъюнкторов от n переменных. Каждый опознает ровно одну комбинацию входов и выдает на свой выход сигнал выбора.

В модели реализованы дешифраторы 2x4 с двумя входами (a_0, a_a) и четырьмя выходами (y_0, y_1, y_2, y_3). Реализация на конъюнкторах подтверждает выбор активным высоким уровнем, реализация на дизъюнкторах – активным низким.

Разберитесь в реализации и работе дешифратора 2x4 со входом выбора en (внутренность блока открывается по двойному щелчку на нем), и дешифратора 3x8, построенного каскадированием двух дешифраторов 2x4.

1.7. Логические ключи – logickey.logicly

Распространено использование вентилей в качестве логических ключей. В этой интерпретации входы вентиля рассматриваются как функционально неэквивалентые: один – как вход данных (сигнала), а второй – как вход разрешения (en). В зависимости от разрешающего сигнала данные проходят на выход - ключ открыт (контакт замкнут), либо не проходят – ключ закрыт (контакт разомкнут).

В модели показано, что в качестве логического ключа можно применить любой вентиль. Разница в том, что ключ может оказаться инвертирующим или неинвертирующим, а в закрытом состоянии на выходе может оказаться 0 или 1. Проанализируйте все варианты, и уясните когда и что наблюдается. Хорошее упражнение на понимание работы вентилей.

Ключи лежат в основе техники коммутации цифровых сигналов. Для примера в модели приведен мультиплексор 2x1, который подключает к выходу один из двух сигналов d_0, d_1 в зависимости от уровня на адресном входе ad. Даны два варианта реализации и пример реализации со входом разрешения.

1.8. Мультиплексоры/демультиплексоры – muxdemux.logicly

Изучите схему и работу 4-канального мультиплектора, который коммутирует вход Data на один из четырех выходов y_0, y_1, y_2, y_3 под управление адреса a_0, a_1. При пассивном низком уровне на входе en все каналы коммутации выключены. В случае демультиплектора все наоборот: данные с одного из четырех входов подаются на выход. Прочие игнорируются.

Обратите внимание на иллюстрацию того факта, что на демультиплексоре с достаточным числом входов можно реализовать любую булеву функцию, выложив таблицу истинности на входы каналов.

1.9. Инкрементор/декрементор – incdec.logicly

Инкрементор – это блок, который интерпретирует вход $(a_0, \ldots, a_{n-1})$ как двоичный позиционный код целого числа от 0 до $2^n - 1$, прибавляет к нему единицу и показывает результат на выходе $(b_0, \ldots, b_{n-1})$. Это типичная комбинационная схема, которая реализует сразу n функций $b_k(a_0, \ldots, a_{n-1})$ от n переменных. Нужные таблицы истинности каждый может построить самостоятельно. Декрементор отличается только тем, что он единицу вычитает.

Простой подход к реализации дает представление об одноразрядном инкременторе, который принимает на входы бит a, перенос их предыдущего разряда C_{in} и вычисляет выход b и перенос в следующий разряд C_{out}. Логика работы одноразрядного инкрементора оказывается крайне простой:

$$b = a \hat{} C_{in}; \quad C_{out} = aC_{in}.$$

Для декрементора все отличается только логикой формирования переноса:

$$b = a \hat{} C_{in}; \quad C_{out} = !aC_{in}.$$

В модели реализованы одноразрядный инкрементор, декрементор, и двухрежимный блок, который умеет работать инкрементором или декрементором в зависимости от сигнала Up. Вникнете в их структуру и работу. Поизучайте работу 4-разрядных инкрементора, декрементора и двухрежимного блока.

1.10. Быстрые инкрементор/декрементор – fastinc.logicly

Каждый одноразрядный инкрементор формирует результат b как сумму по модулю два входного бита a и переноса из предыдущего разряда. В реализациях этой модели сумматоры реализованы как таковые, а формирование переносов поручено отдельному логическому блоку, который анализирует сразу все биты входа (a_0, a_1, a_2, a_3) и раздает переносы всем сумматорам параллельно. Параллелизм вычисления переносов может повысить быстродействие.

В модели представлено два варианта реализации блока формирования переносов. Первый представляет собой цепь сквозного переноса – его распространения через последовательность ключей. Первый же нуль входного кода (a_0, a_1, a_2, a_3) закрывает ключ, обрывая цепь. Перенос достается только тем, у кого все соседи со стороны младших разрядов находятся в единице. Легко понять, что это эквивалентно схеме с последовательным включением одноразрядных инкременторов.

Вторая реализация, известная как схема параллельного переноса, побыстрее. В ней факты наличия пачек единиц в младших разрядов проверяются многовходовыми вентилями AND. По существу, речь идет о различии быстродействий реализации многовходового AND лестницей и деревом.

Подумайте, как изменить схемы формирования переноса, чтобы реализовать декремент.

1.11. Сумматор с последовательным переносом – summ.logicly

Сумматор складывает двоичные позиционные коды (a_0, a_1, a_2, a_3), (b_0, b_1, b_2, b_3) на входах и передает результат на выход (s_0, s_1, s_2, s_3).

В примитивном виде все начинается с реализации одноразрядного полусумматора hADD, который принимает биты b, a и выдает результат их сложения s и перенос в старший разряд. Логика полусумматора легко угадывается:

$$s = a \hat{} b; \quad c = ab.$$

Посмотрите на его реализацию в модели. Полный одноразрядный сумматор учитывает также и входной перенос C_{in}. Его реализация на двух полусумматорах ясна из модели.

Изучите представленную в модели реализацию 4-разрядного сумматора на четырех одноразрядных полных сумматорах. Ясно, что эту реализацию можно транслировать в сторону старших разрядов неограниченно.

1.12. Сумматор с параллельным переносом – fastsumm.logicly

Идея ускоренного переноса та же, что и для инкрементора: нужно построить блок, который, анализируя входы (a_0, a_1, a_2, a_3), (b_0, b_1, b_2, b_3), раздавал бы переносы всем поразрядным сумматорам параллельно, см. модель. К сожалению, логика формирования переносов при сложении не так проста, как при инкременте.

Представленный в модели одноразрядный сумматор FADD принимает биты a, b перенос c и, наряду с результатом сложения s, формирует два вспомогательные признака p, g, используемые для вычисления переносов:

$$s = a\hat{}b\hat{}c; \quad p = a\hat{}b; \quad g = ab;$$

Признак g (Generation) устанавливается тогда, когда в данном разряде генерируется перенос – происходит сложение двух единиц. Признак p (Propagation) – когда через данный разряд происходит распространение переноса из предыдущего. Одновременная установка признаков p, g исключена. (Почему ?). Перенос c_n в разряд n возникает если 1) сгенерирован перенос в предыдущем разряде (признак g_{n-1}) или 2) был перенос c_{n-1} на вход предыдущего разряда и он распространился через него (условие $c_{n-1}p_{n-1}$). Таким образом:

$$c_1 = g_0 + c_0p_0;$$

$$c_2 = g_1 + c_1p_1;$$

$$c_3 = g_2 + c_2p_2;$$

$$c_4 = g_3 + c_3p_3.$$

или, последовательно исключая c справа,

$$c_1 = g_0 + c_0p_0;$$

$$c_2 = g_1 + g_0p_1 + c_0p_0p_1;$$

$$c_3 = g_2 + g_1p_2 + g_0p_1p_2 + c_0p_0p_1p_2;$$

$$c_4 = g_3 + g_2p_3 + g_1p_2p_3 + g_0p_1p_2p_3 + c_0p_0p_1p_2p_3.$$

Именно эта логика и реализована в боке $FAST\_CARRY$ нашей модели. Изучите ее работу. Проверьте выполнение формул для переносов. Проконтролируйте правильность работы сумматора на нескольких примерах. Выявившему дефект – награда.

Наличие входа C_{in} и выхода C_{out} позволяет соединять 4-разрядные секции сумматора с параллельным переносом каскадно. Можно реализовать вычисление признаков p, g и для 4-разрядной секции в целом. Тогда 4 такие секции удастся соединить в один 16-разрядный сумматор по той же логике формирования ускоренного переноса.

2. Изучаем комбинаторику с обратными связями – триггеры

2.1. RS триггер и его производные – RSTriggers.logicly

Взгляните на схему RS-триггера на элементах NAND в модели. При высоких пассивных уровнях на входах $!s, !r$ это просто два замкнутые в кольцо инвертора. Такая система может пребывать в одном из двух устойчивых состояний – $(q, !q) = (1, 0)$ или $(q, !q) = (0, 1)$. В первом случае триггер пребывает в состоянии 1, во втором – в состоянии 0 (хранит бит информации). Это режим хранения.

Подавая активные нули на входы $!s$ (Set) и $!r$ (Reset) можно переключать триггер из одного состояния в другое. Тут же и попробуйте это поделать ! Триггер подчиняется воле человека без слов.

Подачу на $!s, !r$ двух нулей сразу он воспринимает как одновременную выдачу двух противоположных приказов – встать в 1 и встать в 0 – и «сходит с ума». На выходах устанавливаются две единицы. Их уже нельзя интерпретировать как прямой и инверсный выводы внутреннего состояния триггера.

Комбинацию $(!s!r) = (00)$ рассматривают как запрещенную. Неприятность в том, что при переходе на $(!s!r)$ из (00) в режим хранения (11) в схеме триггера возникает логическая гонка.

Это явление воспроизводит отдельная схема в модели. Обратите внимание, что при $sr = 0$ комбинации на входах обоих вентилей тождественны (1,0). Такими же они окажутся и сразу после перехода $sr = 1$ – (1,1). Всякий вентиль NAND, увидев у себя на входах (1,1), попытается поставить на выходе нуль. Проблема в том, что обоим им это не удастся. Кто поставит на выходе нуля первым, тот и не даст это сделать другому. Получается, что после перехода из запрещенного режима $sr = 0$ в режим хранения $sr = 1$ состояние триггера не предсказуемо. Конкретный триггер будет предпочитать одно из двух состояний, поскольку в нем один из двух вентилей «пошустрее» другого. Это же наблюдается и в модели.

Тот же по духу триггер можно реализовать и на двух вентилях NOR (см. модель). Только активные уровни на входах установки оказываются единичными, при $(s, r) = (0, 0)$ имеет место режим хранения, а состояние $(s, r) = (1, 1)$ запрещено – два нуля на выходах.

Добавление на установочные входы триггеров логических ключей вводит вход разрешения en (Enable). При $en = 1$ ключи открыты и, пользуясь входами s, r, с триггером можно делать что вздумается. Если же $en = 0$, ключи закрыты, триггер пребывает в режиме хранения независимо от сигналов на входах.

При $en = 1$ триггер «прозрачен» в том смысле, что изменения на s, r-входах тут же проявляются на выходах. При переходе сигнала en из 1 в 0 триггер сохраняет (защелкивает) последнее достигнутое состояние. Призрачные триггеры этого типа называют «защелками» (Latch). Их основное назначение – прием информации с шины по стробу записи. Особенно популярна защелка типа d с единственным информационным входом. Изучите ее работу по модели.

2.2. Двухтактные триггеры – TTTriggers.logicly

Принципиальный недостаток защелок – их прозрачность. При активном уровне на входе разрешения они ведут себя как обычная комбинационная логика – изменение на входе влечет немедленное изменение на выходе. Это не позволяет использовать обратные связи. Подумайте, к примеру, что произойдет, если инверсный выход $!q$

d-защелки подключить к ее d-входу. (Смоделировать это в данной программе не получится).

Исторически первый способ преодоления прозрачности – это использование противофазного тактирования: пока в тандеме из двух таких триггеров – ведущего (Master) и ведомого (Slave) один устанавливается, второй – блокирован.

Загляните внутрь двухтактного RS-триггера в модели. Вы увидите там два соединенные тандемом триггера с противофазным (через инвертор) тактированием. При единичном уровне на входе clk можно что угодно делать с состоянием ведущего триггера. Но это никак никак не скажется на состоянии ведомого, установка которого запрещена. При переходе clk из 1 в 0 блокированным окажется ведущий, а защелкнутое в нем состояние перепишется в ведомый. Таким образом, все изменения сигналов на выходах двухтактного триггера оказываются жестко привязанными к отрицательным перепадам clk.

Изучите работу двухтактного триггера и двух его несложных модификаций – D-триггер и JK-триггер. С D-триггером мы уже сталкивались. В JK-триггере введены перекрестные связи с выходов ведомого на входы ведущего. Это позволяет ввести на место запрещенного режима $j = k = 1$ так называемый счетный режим, при котором триггер переключается по каждому отрицательному перепаду на clk, принимая в себя инверсию собственного состояния.

2.3. Динамический D-триггер – DTrigger.logicly

Динамический D-триггер крайне прост концептульно (он принимает и запоминает в себе информацию с d-входа по нарастающему фронту сигнала clk), но довольно сложен схемно.

Его понимание разумно начать с упрощенной схемы (см. «первое приближение» в модели). Видно, что это также тандем из одного выходного – ведомого и двух входных – ведущих RS-триггеров.

Проверьте, что при низком уровне на входе тактирования clk, ведомый заведомо находится в режиме хранения (уровни 1 на обоих входах установки). Бросается в глаза, что при изменении сигнала на входе d триггера в нем просто срабатывает цепочка из двух инверторов $d{-}!d{-}d$. При этом один из ведущих триггеров неизменно оказывается в запрещенном состоянии – две единицы на выходах. Какой именно, зависит от уровня на d-входе.

Тот из пары ведущих триггеров, который находится в «правильном» состоянии, просто не узнает о переходе сигнала clk с нуля на единицу, поскольку на другом входе соответствующего вентиля уже присутствует 0. У того же, который находится в запрещенном состоянии при переходе clk в единицу на входе вентиля $NAND$ образуется две единицы. На выходе появляется нуль, который и обеспечивает установку ведомого триггера в состояние согласно действующему уровню на d.

В принципе, это все. Есть небольшая проблема с реакцией триггера на изменение d-входа при высоком clk. Последовательность действий, которая позволяет ее обнаружить, указана в модели. Там же приведен и способ ее устранения добавлением всего одной дополнительной связи. Сравнивая работу двух схем легко понять, почему добавление этой связи снимает проблему.

Более серьезные проблемы возникают при попытке ввести в схему триггера входы асинхронной установки clr (Clear) и pre (Preset). Изучение поведения предварительной (левой) схемы в модели показывает, что асинхронная установка только ведомого триггера не вполне адекватна. Все хорошо, при $clk = 0$, когда ведомый триггер пребывает в режиме хранения. А вот при $clk = 1$ состояние ведомого триггера активно

устанавливается со стороны одного из ведущих. Поэтому изменить его состояние автономно не удается. Изменяя состояние ведомого, нужно менять также и состояния ведущих. Все это выливается в необходимость введения целых трех дополнительных связей, маркированных в модели цифрами 1,2,3.

Чтобы понять роль этих связей, полезно удалить их, а затем добавлять по одной, наблюдая за тем какие именно дефекты поведения исходной схемы устраняются и за счет чего. Простота задачи в том, что все эти дефекты проявляются в ходе анализа асинхронных установок при высоком уровне clk.

2.4. Счетные триггеры – TTrigger.logicly

Концепция счетного триггера проста. Как видно из анализа его внутренности в модели – это просто динамический D-триггер, выход q которого подключен ко входу d через сумматор по модуля два (XOR). Второй вход сумматора выведен наружу в качестве входа T.

Вспомним, что сумматор по модулю два – это важная деталь инкрментора. Там он суммирует содержимое текущего разряда со входом переноса. В данном случае этот сумматор просто упрятан внутрь триггера. Если сигнал переноса из предыдущего разряда на T-вход приходит, триггер переключается на каждом такте, если нет – сохраняет свое состояние. Чтобы организовать на T-триггерах двоичный счетчик (инкрементор) достаточно включить их тандемом и организовать логику формирования переноса. С принципы формирования переносов в инкременторах мы уже знакомы.

2.5. Счетчики – Counters.logicly

Реализованы 4-разрядные инкрементирующие счетчики со входом сброса на счетных T-триггерах по схемам со сквозным и параллельным переносом.

2.6. Вычитающие счетчики – Counters_inv.logicly

Те же счетчики преобразованы в декрементирующие (вычитающие).

2.7. Реверсивный счетчик и реверсивный счетчик со входом синхронной установки – Counters_rev.logicly

Счетчики со сквозным переносом дополнены цепями переключения направления счета (шина UP) и синхронной начальной установки (шина L).