Эволюция наших взглядов на CoDesign за последние 15 лет

Мы познакомились с понятием CoDesign (совместное проектирование аппаратных и программных средств) в процессе работы с сотрудниками департамента R&D фирмы ITALTEL (г.Милан). Надо сказать, что это сотрудничество было многолетним и успешным, например, мы реализовали первый в Европе ATM-коммутатор (аппаратура и ПО, а также средства управления глобальными ATM сетями). Позже мы сосредоточили свои усилия на реализации ПО мобильных телефонов. В те годы мобильная связь только начинала развиваться и, пожалуй, основной проблемой была battery supply, т.е. эффективность использования электроэнергии, обеспечиваемой громоздкими аккумуляторами. Итальянцы сначала заказали нам разработку ПО мобильных телефонов (именно трубок, а не базовых станций или MSC) всех известных на тот момент стандартов NMT, GSM и CDMA, а потом попросили оценить, какой процент возможностей микропроцессора M6800 используется для выполнения этого ПО. Результат нас ошеломил – не более 20%! На уровне вентилей нет возможности управлять потреблением электроэнергии, вентилю не скажешь: «в этот такт помолчи и ничего не делай».

Так возникла идея разработки микропроцессора, специальным образом оптимизированного на заданную задачу. Разработка микропроцессора – это очень сложная и дорогая задача, но и мобильные телефонные трубки выпускаются миллионами. Чуть позже обнаружились и другие массовые приложения, где выгодно использовать специализированные микропроцессоры, поэтому задача была обобщена следующим образом. Пусть имеется программа на каком-либо алгоритмическом языке высокого уровня. Заданы ограничения на время исполнения этой программы, например, каждая нить должна исполняться не более чем за 1 мс. Требуется спроектировать микропроцессор, который выполняет заданную программу с учетом временных ограничений и имеет при этом минимальное число вентилей (минимальное потребление электроэнергии, минимальная стоимость – все эти факторы практически линейно зависят друг от друга). В качестве дополнительного важного требования выдвигалась автоматическая генерация компилятора в систему команд создаваемого микропроцессора, отладчика, потактового симулятора и других инструментальных средств – ведь оптимальный микропроцессор рождался «голым».

Мы приступили к решению этой задачи и примерно за год нащупали подход, как нам казалось, вполне перспективный. Мы предложили разработку средств, позволяющих найти в программе повторяющиеся «времяпожирающие» части, точнее объекты, чтобы можно было инкапсулировать и данные, которые эти объекты обрабатывают. Затем эти объекты нужно реализовать в виде специализированных команд, расширяющих какую-то выбранную заранее архитектуру. Процесс предполагался итеративным: выбор объектов-кандидатов, расширение системы команд, генерация компилятора и симулятора, профилирование, оценка того, что получилось, если результаты не удовлетворяют архитектора, процесс повторяется, начиная с новых кандидатов. Такой подход к CoDesign был опубликован в трудах международной конференции в Берлине в 1996 г. [1].

Однако не всё оказалось так просто. Для реализации автоматической генерации компилятора в только что полученную систему команд мы сильно рассчитывали на технику retargetable compiler generation, мы прочитали десятки статей на эту тему, выполнили целый ряд масштабных экспериментов [2], но оказалось, что трудности такого описания архитектур ЭВМ, по

которому можно автоматически сгенерировать кодогенератор, слишком велики для задачи CoDesign, в которой по нашему плану архитектор мог бы менять архитектуру за минуты.

Выход из этого тупика нашел Д.Булычев. Вместо того, чтобы итеративно подбирать оптимальную для данной задачи архитектуру, он предложил схему, при которой система команд строится за одну итерацию на основе анализа дерева разбора программы. Разумеется, предельным решением является единственная команда, реализующая всю программу. Очевидно, что это приводит к чрезмерному расходу аппаратуры, например, если в программе есть 100 сложений и 10 умножений, в аппаратной реализации будет ровно столько сумматоров и умножителей. Таким образом, нужно вводить какие-то разумные ограничения на сложность используемых команд, например, в одной команде не может более двух обращений к памяти или не более двух умножений и т.д. Разумность ограничений – это задача архитектора, который лучше всех представляет специфику решаемой задачи.

Итак, задача сводится к «раскраске» дерева разбора программы маленькими поддеревьями, т.е. командами с заданными ограничениями. Как известно, задача поиска поддерева в дереве имеет экспоненциальную сложность, поэтому для реальных программ такое «лобовое» решение будет работать очень долго. Д.Булычеву удалось найти эффективный алгоритм, в котором экспоненциальная часть сведена к минимуму. Мы провели целую серию экспериментов, в которых выбирали оптимальную систему команд для известных библиотек, ориентированных на область встроенных приложений. Даже для больших программ время выбора оптимальной системы команд не превышало одной минуты, при этом сокращение длины кода по сравнению со стандартными архитектурами было существенным. Заметим, что при таком подходе задача генерации компилятора просто исчезла – объектный код получается сразу во время выбора системы команд. Были разработаны прототип генератора VHDL, т.е. аппаратной реализации выбранной архитектуры (правда, только для однотактных команд без конвейера), генераторы ассемблера, дизассемблера и потактного симулятора. В 2004 г. Д.Булычев защитил кандидатскую диссертацию [3], на основе которой в ЗАО «Ланит-Терком» был открыт инвестиционный проект по доведению этой многообещающей работы до промышленной технологии.

Чтобы добиться максимальной эффективности, необходимо максимально распараллелить исходную программу. Эта задача давно известна и популярна, но её решение имеет теоретические пределы, кроме того, часто пользователь может предложить исполнять два фрагмента программы параллельно, даже если с формальной точки зрения это невозможно (например, он знает, что какие-то ситуации в его программе не встретятся по каким-то внешним, не отраженным в тексте программы причинам). Таким образом, входным языком системы CoDesign должен быть не стандартный C (или какой-то другой традиционный язык высокого уровня), а специально разработанный язык, который, с одной стороны, достаточно выразителен и удобен для пользователя, а, с другой стороны, дает возможность пользователю определять параллельные конструкции на любом уровне и допускать эффективную реализацию в VHDL или Verilog.

Обзор аналогичных работ

Понятно, что не только нам пришла в голову эта идея. Нам удалось познакомиться со многими языками, предназначенными для эффективного синтеза аппаратуры, но все они обладают крупными, на наш взгляд, недостатками.

В проекте ROCCC 2.0 [4] используется C-подобный язык для распараллеливания тел циклов. Параллельные версии циклов затем отображаются в FPGA для обеспечения большей производительности. Собственно, весь этот проект ориентирован на высокопроизводительные вычисления, а не на разработку аппаратуры. Например, в этом языке нет понятия цикла, поэтому невозможно описать произвольную аппаратуру.

Spec C [5] – расширение C для разработки аппаратуры. Взаимодействие между параллельными сущностями базируется на событиях. Семантика временных интервалов для доставки событий практически повторяет семантику VHDL. Предусмотрены операторы для описания произвольного потока управления и для явного указания параллелизма и конвейера. С другой стороны, нет явного деления на аппаратные такты (clocks), что, в свою очередь, не позволяет определить надежную доставку сообщений, приоритеты для каналов, через которые несколько сообщений могут быть посланы одновременно (т.е. в одном и том же такте).

Handel-C [6] (построен на основе языка Occam [7], который, в свою очередь, базируется на модели взаимодействующих последовательных процессов CSP[8]) представляет из себя еще одно расширение C. Handel-C предназначен для проектирования синхронных решений с удачной семантикой каналов и надежной доставки сообщений.

В то же время в этом языке не хватает некоторых возможностей для достижения максимальной эффективности:

1. Сообщение не может быть принято и отправлено дальше внутри одного такта.
2. Несколько процессов не могут получить сообщение из одного и того же канала в одном такте.
3. Процесс не может отказаться от приема сообщений.
4. Структурная декомпозиция в Handel-C не такая удобная, как в VHDL.

Bluespec[9,10] позволяет описывать системы в виде множества модулей, взаимодействующих через команды, которые они посылают друг другу, или через доступ к портам, из которых разрешено только чтение. Внутреннее описание модуля основывается на концепции охраняемых атомарных действий, которые оперируют с локальными данными модуля. Каждое действие предваряется предикатом, разрешающим исполнение действия в этом такте. Чтобы выполнить какие-то действия параллельно, нужно запустить специальный динамический планировщик, для которого пользователь должен задать дополнительные данные.

Главным недостатком Bluespec является невозможность точной потактовой симуляции, без чего невозможно промышленное применение.

SystemC [11] позволяет описывать аппаратуру на разных уровнях абстракции и предназначен, в основном, для моделирования на системном уровне, хотя и имеет возможности описания аппаратуры на RTL уровне. Строго говоря, SystemC – это не язык, а скорее набор идиом (образцов) для языка C++. Мы думаем, что «программирование в терминах идиом» делает невозможным многие желаемые расширения языка, например, нужный поток управления.

Краткий обзор предлагаемого нами подхода

После нескольких лет напряженной работы над распараллеливанием и конвейеризацией программы пользователя, предназначенной для аппаратной реализации, было решено начать с другого конца. Мы определили минимальное ядро языка, в котором конвейеризация заложена с самого начала и почти не видна пользователю, для параллелизма предусмотрены наглядные средства, эффективно реализуемые в аппаратуре, а традиционные конструкции высокого уровня (циклы, процедуры, сравнение с образцами и т.д.) задаются как расширения ядра, причем каждое расширение может быть точно описано средствами ядра. Основные действия ядра – посылка и прием сообщений, нет даже понятия состояния. Чтобы в следующем такте процесс «помнил» состояние, он в конце каждого такта посылает информацию о состоянии себе на вход, а присваивание – это посылка нового значения на более приоритетный вход этого процесса. Язык, получающийся объединением ядра и расширений, получил название HaSCoL (Hardware and Software Codesign Language, т.е. язык для совместной разработки аппаратных и программных средств).

Программа на языке HaSCoL представляет из себя множество процессов, общающихся между собой сообщениями через свои разъемы (соединители, plugs), которые бывают входными и выходными. Например:

```
process X =
      begin
            in input;
            out output (int);
      end;
```

Здесь описан процесс `X`, в котором есть входной разъем `input` без параметров и выходной разъем `output` с одним целым параметром. Поскольку в начале такта в разъем может прийти несколько сообщений из разных источников, нужно уметь как-то управлять их приоритетами. Для этой цели служит понятие `порт`, например,

```
in input (int, int) [A,B];
```

здесь входной разъем `input` имеет два порта `A` и `B` (каждый с двумя целыми параметрами), причем первый порт `A` имеет более высокий приоритет, чем второй порт `B`. Если процесс затребует сообщение из разъема `input`, а там будет 2 сообщения (в обоих портах), то прочитается сообщение из порта A. Если в определении разъема порт не указан, то считается, что разъем имеет один порт, имя которого совпадает с именем разъема.

Внутреннее устройство процесса никак не видно извне, процесс также ничего не знает о своем окружении, он знает только свои разъемы, поэтому два процесса с одинаковыми разъемами абсолютно взаимозаменяемы. Внутри одного процесса могут быть определены другие процессы, т.е. процесс является «кирпичиком» структурной декомпозиции сложных систем.

Процесс, внутри которого определены подпроцессы (вложенные процессы), может задать определенную топологию соединений между разъемами своих подпроцессов (но только непосредственно вложенных в него!):

– входные разъёмы родительского процесса могут быть связаны с входными разъёмами вложенных процессов;

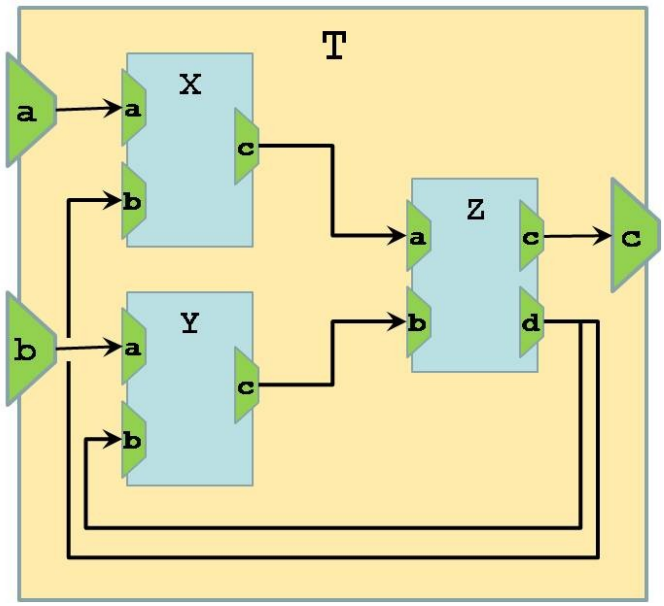
– выходные разъёмы вложенных процессов могут быть связаны с выходными разъёмами родительского процесса;

– выходные разъемы вложенных процессов могут быть связаны с входными разъёмами этого же или других вложенных процессов (того же уровня вложенности!).

На самом деле, все эти соединения – не более чем «синтаксический сахар», всё это можно описать в текстовом виде на языке HaSCoL обработчиками, перепосылающими сообщения, но, разумеется, графическое представление намного нагляднее.

Приведем пример

```
process T = begin
  in a, b;
  out c;
 process X = begin
    in a, b;
    out c;
 end with a = a, b = Z.d, c = Z.a;
 process Y = begin
    in a, b;
    out c;
 end with a = b, b = Z.d, c = Z.b;
 process Z = begin
    in a, b;
    out c, d;
 end with c = c;
end
```



Обратите внимание, что, например, в описании процесса X в записи **with** a=a первое вхождение a обозначает собственный разъем процесса X, а второе вхождение – разъем родительского процесса.

Процесс может читать сообщения из своих входных разъемов, а также из выходных разъемов своих непосредственно вложенных подпроцессов.

Процесс может посылать сообщения в свои входные и выходные разъемы, а также во входные разъемы своих непосредственно вложенных подпроцессов. На самом деле, сообщения посылаются не в разъем, а в порты этого разъема, читаются же сообщения из разъема в целом, а не из каких-то его портов.

Описание процесса имеет следующую структуру:

```
process p = begin
```

in входные разъемы;

out выходные разъемы;

data локальные переменные;

let локальные обозначения;

один или несколько обработчиков сообщений

end

Как мы уже говорили, понятия переменной или состояния в языке HaSCoL нет, они моделируются перепосылками сообщений, но для простоты вполне можно считать, что есть регистры, которые играют роль переменных. Тот факт, что реализация регистров осуществляется перепосылками сообщений, автору программы не виден. Регистры играют роль локальных данных.

Обозначение – это способ сокращения записи и экономии аппаратуры.

Рассмотрим пример

```
let ab = a+b
```

Далее `ab` в рамках текущего процесса можно рассматривать как полный эквивалент формулы `a+b`, которая будет заново вычисляться в каждом вхождении обозначения `ab`. Больше всего это похоже на макроподстановку, но есть и важное отличие – по записи **let** `ab = a+b` будет сгенерирован один сумматор, а все вхождения `ab` перейдут в «проводок», запускающий этот сумматор. В случае макрогенерации каждое вхождение `ab` порождало бы отдельный сумматор, что, разумеется, ухудшает эффективность.

Обработчик устроен следующим образом:

```
условие {тело}
```

В условии принимаются сообщения. На основании успешности принятия сообщений принимается решение о запуске операторов тела.

Тело

Тело – это набор из последовательно исполняющихся конструкций:

$$\{S_1; S_2; \cdots; S_m\}$$

причем конструкции $\{S_1; S_2; \cdots; S_m\}$ исполняются в конвейерном режиме: если тело начало исполняться в каком-то такте (т.е. начала исполняться конструкция S_1), а для этого необходимо, чтобы условие в этом такте было истинным, то в следующем такте будет исполняться S_2. При этом и S_1 снова будет исполняться, если и в следующем такте условие будет истинным, и так далее. Если в каком-то такте условие ложно, исполнение S_1 не начинается, но исполнение остальных конструкций продолжается (если они не приостанавливаются по каким-то своим внутренним причинам).

Таким образом, возникает следующая картина. Одному телу $\{S_1; S_2; \cdots; S_m\}$ могут соответствовать несколько параллельно протекающих процессов, каждый на своем уровне конвейера, например,

1) $S_1; S_2; S_3; S_4$

2) $S_1; S_2; S_3$
3) $S_1; S_2$
4) S_1

Здесь первый процесс дошел до исполнения оператора S_4, а четвертый только начал S_1.

А что же будет, когда исполнится последняя в теле конструкция S_m? Это означает, что закончена обработка данных, когда-то полученных в сообщении, запустившем S_1. Но за это время могло прийти много других сообщений, конвейер работает, пока не будет выключено электропитание.

Каждое S_i состоит из параллельно исполняемых простейших операторов, например, присваиваний и посылок сообщений

$$\{P_1|P_2|\cdots|P_k\}$$

Все P_i исполняются одновременно за один такт, причем они исполняются действительно параллельно, например, в $\{x := y|y := x\}$ переменные x и y обменяются значениями (в начале такта будут прочитаны y и x, а в конце такта эти значения будут записаны в x и y).

В качестве простейших операторов P_i могут использоваться следующие операторы:

`skip` – пустой оператор.

`x:=E` – присваивание, `x` – переменная, `E` – выражение (вполне традиционное без побочных эффектов). Присваиваниями нужно пользоваться аккуратно, т.к. разные экземпляры процессов конвейера будут присваивать значения в одну и ту же переменную.

`x=E` – локальное обозначение, `x` – локальное имя, `E` – выражение. Локальные обозначения «доживут» до использования в своем экземпляре конвейера, не вызывая никаких конфликтов, т. е., в каждом процессе конвейера будет действовать свое локальное обозначение. Именно поэтому использование локальных обозначений предпочтительнее использования присваиваний.

В отличие от конструкции **`let`** здесь нет никаких хитростей. Выражение `E` один раз вычисляется и запоминается в локальном обозначении `x`. Никто не запрещает использовать то же самое локальное обозначение `x` в новых описаниях даже других типов, но, разумеется, это вредит понимаемости («читабельности») программы. В этом случае старое локальное обозначение просто исчезает. Для тех, кто помнит Алгол 68, можно сказать, что `x=E` – это полный аналог описания тождества.

Условный оператор

$$\boldsymbol{if}\ C_1\ \boldsymbol{then}\ S_1\ \boldsymbol{elif}\ C_2\ \boldsymbol{then}\ S_2\ \cdots\ [\boldsymbol{else}\ S_m]\ \boldsymbol{fi}$$

Здесь $C_1,\ C_2, \cdots$ – условные выражения.

$S_1, S_2, \cdots\ S_m$ – конструкции из параллельно исполняемых простейших операторов $\{P_1|P_2|\cdots|P_k\}$.

Все $C_1,\ C_2, \cdots$ и $S_1, S_2, \cdots$ исполняются параллельно за 1 такт, но семантика HaSCoL устроена таким образом, что видимый эффект (присваивание новых значений переменным, посылка

сообщений) будет иметь только та конструкция S_k, условное выражение которой C_k выдало значение `истина`.

Условный оператор заимствован из языка Алгол 68.

inform `m` – ненадежная посылка сообщения `m`.

В операторе **inform** `m` доставка сообщения не гарантируется. После того, как сообщение послано, исполнение посылающего процесса продолжается без всяких условий (т.е. даже если сообщение никто не получил).

send `E` – надежная посылка всех сообщений, содержащихся в `E`.

send – один из самых сложных операторов языка HaSCoL. Разберем его на последовательно усложняющихся примерах.

Пусть в каком-то процессе есть разъем

```
in input(int, int)[A,B];
```

Тогда в обработчиках этого процесса может встретиться оператор

```
send input'B(1,2)
```

`input` – имя разъема, `B`– имя порта этого разъема, `(1,2)` – фактические параметры посылаемого сообщения.

Если в разъеме, куда посылается сообщение, порты не определены, то подразумевается, что есть только один порт, имя которого совпадает с именем разъема, тогда имя порта в операторе **send** можно не указывать.

Если посылаемое сообщение может быть принято получателем (причины, почему сообщение не может быть принятым, мы разберем при описании условия обработчика), результатом **send** является логическое значение `истина`, в противном случае результатом является `ложь`, тогда оператор, в котором **send** исполняется, приостанавливается до тех пор, пока сообщение не будет принято.

В одном операторе **send** может посылаться несколько сообщений, каждая посылка вырабатывает логическое значение `истина` или `ложь` и выполняется независимо от других посылок этого **send**. Значения посылок собираются в традиционную логическую формулу («и», «или», «не» и т.д.), если в итоге получается `ложь`, оператор, в котором исполняется **send**, приостанавливается, хотя некоторые сообщения благополучно доставлены.

Рассмотрим примеры

```
send n(1) and m(2)
```

Здесь посылается сообщение с фактическим параметром 1 в разъем `n` и сообщение с фактическим параметром 2 в разъем `m`. Исполнение конструкции S, в которой встретился этот оператор **send**, будет продолжено, только если оба сообщения будут успешно доставлены. Если одно из сообщений будет доставлено, а второе – нет, исполнение S будет приостановлено, пока задержанное сообщение не будет доставлено тоже.

```
send n(1) or m(2)
```

Для продолжения работы нужно, чтобы хотя бы одно из этих сообщений было доставлено.

Заметьте, что оператор **send** m **or true** является полным эквивалентом оператора **inform** m.

Посылка сообщения оператором **send** называется надежной, потому что сообщение никогда не пропадает, когда приемник будет готов к приему, сообщение будет доставлено. Это вызывает определенные трудности в реализации, например, для сообщений, которые не могут быть сразу же доставлены по назначению, приходится заводить специальные аппаратные буферы. Сейчас авторы HaSCoL работают над тем, чтобы сделать основным механизмом ненадежные сообщения.

Поскольку сообщения доставляются в том же такте, когда посылаются, по ошибке автора программы может возникнуть следующая ситуация, называемая комбинационным циклом:

$$p_1 \to p_2 \to p_3 \to p_1$$

Здесь процесс p_1 посылает сообщение процессу p_2, который перепосылает его в p_3, а тот – снова в p_1! Такие ситуации ловятся статически транслятором с языка HaSCoL.

Условие

В языке HaSCoL есть оператор посылки сообщения (даже два – **send** и **inform**), а оператора **receive** (получить) или ему подобного нет. Считается, что практически в каждом такте процесс может принять сообщения от других процессов (или от себя), поэтому роль оператора получения сообщения играет условие, с которого начинается каждый обработчик процесса. Пример условия:

```
m(a) when a>∅
```

Здесь из разъема m принимается сообщение (если оно там есть), если параметр a этого сообщения больше ∅, то условие возвращает значение истина, и начинается исполнение тела. Если сообщения не было или параметр был меньше или равен ∅, то возвращается значение ложь, а исполнение тела приостанавливается.

Каждый обработчик может получить и сразу несколько сообщений из разных разъемов, таким образом, в общем виде условие выглядит так:

$$m_1[\boldsymbol{and}\ C_1][\boldsymbol{when}\ G_1],\ m_2[\boldsymbol{and}\ C_2][\boldsymbol{when}\ G_2],\ \cdots,\ m_k[\boldsymbol{and}\ C_k][\boldsymbol{when}\ G_k]$$

Как обычно, запись $[\boldsymbol{and}\ C]$ означает, что $\boldsymbol{and}\ C$ может отсутствовать, аналогично $[\boldsymbol{when}\ G]$. Все m имеют форму $p\ (n_1, \cdots, n_m)$, где p – имя входного разъема, $n_1, \cdots, n_m$ – формальные параметры, соответствующие по количеству и типам параметрам данного разъема p. Если сообщение принимается, то формальные параметры получают в качестве значений фактические параметры сообщения.

Условие обработчика выдает значение истина, если на все разъемы, упомянутые в условии, пришли сообщения и все логические выражения G_i истинны. В этом случае тело обработчика начинает исполняться.

Логические выражения C_i играют другую роль – если сообщение может быть принято и C_i истинно, то сообщение из этого разъема удаляется, иначе сообщение остается и может быть использовано еще раз. Если фрагмент ***and*** C_i опущен, по умолчанию принимается `истина`.

Аналогичное умолчание принимается и для фрагмента ***when*** G_i.

Примеры:

```
m(a), n(b) when a>b
```

Это условие истинно, если через входные разъемы `m` и `n` приходят сообщения и фактический параметр первого сообщения больше фактического параметра второго. Если все это выполнено, оба сообщения удаляются из своих разъемов.

`m(a) and a>0, n(b) when a>b` – так же, как и в предыдущем примере, но из разъема `m` сообщение удаляется только, если фактический параметр больше Ø.

Есть еще две формы условия обработчика, не связанные с получением сообщений.

`init` – истинно только однажды при запуске системы (по сигналу `reset`);

`default` – истинно в начале каждого такта.

Пример

Продемонстрируем все вышесказанное на реальном примере – реализация очереди FIFO (первый пришел – первый ушел).

```
process Queue = begin
  in  inp(int);
  out outp(int);

data buffer : [0..7 : uint (3)]int,
        first : uint(3) = 0,
        last : uint(3) = 0;                          } data

 let notEmpty  =  first ! =  last;
let isFull  =  first  ==  last  +  1;                 } local bindings
    let notFull  =  not isFull;

            default when notEmpty {
              send outp (buffer[first]) | first :=  first  +  1
             }
            inp (x) when (notFull or (isFull and outp)) {
                if not notEmpty then inform outp(x)
                fi |

if notEmpty or not outp then buffer[last] :=  x | last :=  last  +  1
fi
             }                                       } handlers

end
```

Мы пока вообще не упоминали типов данных, обрабатываемых языком HaSCoL. Но в этом примере ничего сложного нет:

`uint` – это беззнаковое целое,

`uint(3)` – беззнаковое целое длиной 3 бита, т.е. числа от 0 до 7, заметьте, что для таких данных 7+1=0 (перенос пропадает).

`buffer` – это массив из 8 целых чисел, его индексами служат целые числа типа `uint(3)`.

Таким образом, в этом примере описывается аппаратная реализация очереди FIFO с буфером из 8 целых элементов и двумя указателями `first` – индекс в очереди первого кандидата на отправку и `last` – индекс в очереди последнего неотправленного сообщения.

Итак, описан процесс Queue с входным разъемом `inp` и выходным `outp`. Оба разъёма имеют по одному целому параметру. В процессе описаны переменные `buffer`, `first` и `last`, а также введены локальные обозначения `not_Empty` (очередь не пуста), `is_Full` (очередь заполнена полностью) и `not_Full` (в очереди есть свободные места).

Описаны два обработчика. Первый довольно простой и запускается каждый такт, если очередь не пуста. Он посылает в выходной разъем первый элемент очереди и продвигает указатель `first`.

Второй обработчик запускается, если во входном разъеме есть сообщение и в очереди есть место, куда его записать. На самом деле, условие запуска обработчика сложнее – если очередь полна, но выходной порт готов принять сообщение, это означает, что первый обработчик точно пошлет первое сообщение из очереди, освободив тем самым одну позицию, которую второй обработчик займет в конце такта.

Использование идентификатора выходного разъёма в качестве аргумента логического выражения без намерения послать сообщение не описывалось ранее, это просто признак готовности разъёма к приёму сообщения.

Тело второго обработчика состоит из двух параллельно исполняемых операторов – в первом принятое сообщение сразу же передается на выход, если очередь пуста (в этом случае последнее принятое сообщение является и первым), а во втором – записывается в конец очереди.

В примере есть одна хитрость, которую без объяснений трудно понять. Речь идет об использовании во втором обработчике оператора **`inform`** вместо **`send`**. Дело вот в чем. Если в первом обработчике оператор **`send`** не сможет передать сообщение (выходной разъём не готов принять сообщение), обработчик приостановится и будет ждать момента, когда сообщение все-таки уйдет.

От приостановки первого обработчика никто не пострадает, очередь будет принимать сообщения во втором обработчике, пока не переполнится. Но если бы мы во втором обработчике использовали оператора **`send`**, то любая задержка в нем заблокировала бы дальнейшее заполнение очереди. Поэтому вместо **`send`** использован неблокирующий оператор **`inform`**, а в конце добавлено условие **`or not`** `outp`. Если очередь была пуста, но входной разъем не был готов принять сообщение, то принятое из входного разъема сообщение могло бы потеряться, а с добавлением этого условия оно запишется в конец очереди по обычным правилам.

Список литературы

[1] A. Barabanov, M. Bombana, N. Fominykh, G. Gorla, A. Terekhov. Reusable objects for optimized DSP design. In Embedded Microprocessor Systems, IOS Press, 1996

[2] D. Boulytchev, D. Lomov. An Empirical Study of Retargetable Compilers. In Perspectives of System Informatics, Springer Berlin / Heidelberg, 2001

[3] Д.Булычев, «Прототипирование встроенных систем на основе описания макроархитектуры», диссертация на соискание учёной степени кандидата физ-мат наук, СПбГУ, 2004 г.

[4] W. Najjar and J. Villareal. Reconfigurable Computing in the New Age of Parallelism. SAMOS Workshop, 2009

[5] R. D. Jianwen Zhu and D. D. Gajski. Syntax and Semantics of the SpecC Language. In Proceedings of the Synthesis and System Integration of Mixed Technologies, 1997.

[6] Celoxica. Handel-C Language Reference Manual, 2005.

[7] I. Page and W. Luk. Compiling Occam into FPGAs. “FPGAs”, Abingdon EE&CS books, pages 271-283, 1991.

[8] C. A. R. Hoare. Communicating Sequential Processes. Prentice-Hall International, 1985.

[9] Arvind, R. S. Nikhil, D.Rosenband and N. Dave. High-Level Synthesis: An Essential Ingredient for Designing Complex ASICs. In Proceedings of ICCAD, 2004.

[10] J. Hoe. Operation-Centric Hardware Description and Synthesis. PhD thesis, Dept. of EE&CS, MIT, 2000.

[11] G. M. Thorsten Grötker, Stan Liao and S. Swan. System Design with SystemC. Kluwer Academic Publishers, 2002.