RU EN ПРОГРАММНЫЕ СИСТЕМЫ: ТЕОРИЯ И ПРИЛОЖЕНИЯ ISSN 2079-3316
научная статья ПРОГРАММНОЕ И АППАРАТНОЕ ОБЕСПЕЧЕНИЕ РАСПРЕДЕЛЕННЫХ И СУПЕРКОМПЬЮТЕРНЫХ СИСТЕМ

УДК 004.312.4
10.25209/2079-3316-2023-14-4-67-89

Метод построения циклических конвейеров

Игорь Алексеевич **Адамович**[1]✉, Юрий Андреевич **Климов**[2]

[1] Институт программных систем им. А. К. Айламазяна РАН, Веськово, Россия
[2] Институт прикладной математики им. М. В. Келдыша РАН, Москва, Россия
[1]✉*i.a.adamovich@gmail.com*

Аннотация. Одним из наиболее эффективных способов организации вычислений на ASIC или FPGA является построение неостанавливаемых конвейеров. Однако для некоторых вычислительных схем получаемый конвейер может оказаться слишком большим для имеющихся ресурсов ASIC или FPGA. Авторами предлагается метод построения циклических конвейеров, управление потоками данных в которых основано на счетчиках и не зависит от данных, передаваемых по конвейеру. Предложенный метод позволяет строить более компактные неостанавливаемые конвейеры со скважностью, равной количеству проходов по циклу, которые должны пройти данные, чтобы конвейер преобразовал их в искомый результат.

Ключевые слова и фразы: конвейер, ПЛИС, микросхема, скважность, очередь, кредит

Для цитирования: Адамович И. А., Климов Ю. А. *Метод построения циклических конвейеров* // Программные системы: теория и приложения. 2023. **Т. 14**. № 4(59). С. 67–89. https://psta.psiras.ru/read/psta2023_4_67-89.pdf

Введение

При разработке эффективных программно-аппаратных комплексов встречаются ситуации, в которых процессор общего назначения не способен обеспечить требуемую производительность при заданных рамках потребляемой мощности. Частым решением в таких условиях является перенос вычислительно-емкого ядра программы на одну или несколько ASIC- или FPGA-микросхем.

ASIC [1] (Application-Specific Integrated Circuit, интегральная схема специального назначения) — это интегральная схема, которая спроектирована для решения какой-то одной отдельной задачи и, в отличие от процессора общего назначения, не предназначена для решения любых задач. Благодаря специализации ASIC имеет оптимизированные на схемотехническом уровне соединения элементов, а также их расположение, что во многих случаях позволяет значительно увеличить производительность и эффективность комплекса. Возможность адаптировать ASIC на таком глубоком уровне к решению конкретной проблемы является основой для тонкой настройки и оптимизации всей системы. Кроме того, при равной вычислительной мощности ASIC зачастую потребляет значительно меньше энергии по сравнению с процессором общего назначения, когда измерение производится на одной и той же задаче. Следует отметить, что разработка и выпуск ASIC занимает много времени и обходится существенно дороже готовых процессоров общего назначения, поэтому круг применения ASIC достаточно узок.

Промежуточным вариантом между процессором общего назначения и ASIC является FPGA. FPGA [2] (Field Programmable Gate Array, программируемая пользователем вентильная матрица) — это программируемая микросхема, в которую загружается не программа для процессора общего назначения, а описание схемы. Она не такая быстрая как ASIC: работает на меньшей частоте, потребляет больше энергии, но обладает схожими с ASIC возможностями для оптимизации схемы соединений и расположения элементов, что в некоторых задачах дает значительные преимущества перед процессорами общего назначения.

Основное преимущество FPGA перед ASIC — программируемость. Схему соединений и расположение элементов в любой момент можно изменить. Благодаря этому свойству FPGA часто используют как прототип для будущей специализированной микросхемы: если при разработке допущена ошибка, то в FPGA ее легко исправить, в отличие от ASIC.

Простой перенос алгоритма на ASIC или FPGA может не дать ожидаемого прироста в производительности (см. например раздел 2 в [3]). Одним из наиболее используемых и эффективных методов адаптации

алгоритма к архитектуре ASIC/FPGA является метод конвейеризации [2]. Суть этого метода заключается в разбиении алгоритма на несколько последовательных этапов, позволяя системе обрабатывать несколько задач одновременно (по задаче на каждый этап), как на производственной линии.

При разработке конвейерной реализации алгоритма необходимо, с одной стороны, соблюсти баланс между необходимой производительностью и занимаемой конвейером площадью микросхемы и, с другой стороны, обеспечить поток данных через конвейер, когда блоки, выдающие данные в конвейер, и блоки, потребляющие данные из конвейера, могут по различным причинам быть не готовыми к выдаче или потреблению данных.

Как общие традиционные способы построения конвейеров [4], так и частные, например, применяемые для построения процессорных микроархитектур [5], часто могут оказываться непригодны из-за больших длин конвейеров.

В исследованиях мы столкнулись с необходимостью сконструировать алгоритм, потенциально реализуемый на нескольких конвейерах, но ресурсы FPGA не позволяли построить полную конвейерную схему. При этом в FPGA не было достаточно ресурсов для реализации традиционной промежуточной буферизации данных, и схема такой реализации оказалась слишком сложна. Поэтому мы поставили перед собой цель разработать новое решение обозначенной выше проблемы.

В настоящей статье предлагается новый, разработанный авторами способ построения конвейеров, который позволяет в некоторых случаях упростить разработку и существенно повысить тактовую частоту схемы по сравнению с традиционными вариантами. Новый способ эффективен при невозможности разместить полный конвейер в микросхеме. В этом случае мы предлагаем организовать в конвейере цикл и предлагаем локальный способ разрешения конфликтов в месте объединения потоков данных. В предлагаемом циклическом конвейере отсутствует возможность остановки (stall) конвейера, что и упрощает разработку, и повышает тактовую частоту разработанной схемы. Конвейеры без возможности остановки мы будем называть *неостанавливаемыми*.

Следует сказать, что широко используется [6, 7] способ построения систем с помощью кредитного механизма, который используется и в нашей работе. Схема [8] основывается на неуправляемом подходе к построению конвейеров и ее можно включить в нашу систему конвейеров в качестве вспомогательной. В [9] используются циклические конвейеры, но задействуют классические способы управления и остановки конвейеров.

В работах [3] и [10] рассматриваются классы задач, реализуемых на FPGA и ASIC, а также способы решения таких задач. Работы [11] и [12] близки нашему исследованию и посвящены реализации алгоритма вычисления хеш-функции SHA-3. Более подробный обзор упомянутых работ и их сравнение с нашим исследованием будут проведены в четвертой главе «Близкие работы».

Дальнейшее изложение будет построено следующем образом. В первой главе проведен обзор способов построения конвейера и методов управления конвейерами. Во второй главе будет описан предлагаемый подход к построению неостанавливаемых циклических конвейеров. В третьей главе будет представлена практическая ценность подхода на основе неостанавливаемых циклических конвейеров. Четвертая глава посвящена обзору работ близких к нашей. И наконец заключение завершает статью.

1. Построение конвейеров

1.1. Пример программы

Для сравнения различных подходов к архитектуре конвейера рассмотрим простую задачу, программа которой на псевдокоде изображена на листинге 1.

```
для всех a ∈ A:
    b := a
    выполнить N раз:
        b := f(b)
    положить b в B
```

Листинг 1. Простой алгоритм-пример

Программа представляет собой двойной цикл, который обрабатывает элементы из множества A. Для каждого элемента a из множества A выполняются следующие действия:

(1) Переменная b инициализируется значением a.

(2) Вложенный цикл выполняется N раз, на каждой итерации вычисляется функция $f(b)$ и результат помещается снова в переменную b.

(3) После завершения вложенного цикла финальное значение из переменной b помещается в множество B.

В результате программа применяет N раз функцию f к каждому элементу множества A, а результаты сохраняет во множестве B.

Алгоритмы близкого вида встречаются в различных областях: численная математика, криптографические вычисления и другие.

Обычно реализация такого алгоритма на процессорах общего назначения предполагает, что множества A и B хранятся во внешней по отношению к процессору памяти (например в ОЗУ).

1.2. Реализации в ASIC/FPGA

1.2.1. *«Наивная» реализация*

Выполним перенос описанной выше программы на FPGA или ASIC. Сначала используем «наивный» подход, который заключается в том, что структура схемы будет максимально похожа на структуру программы. На верхнем архитектурном уровне «наивный» вариант схемы можно представить так:

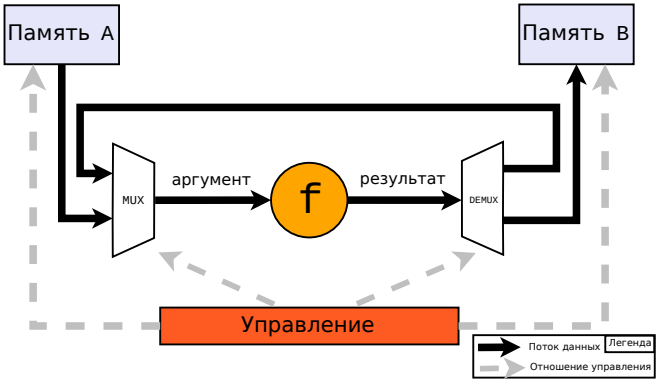

Рисунок 1. Архитектура «наивной» схемы

На рисунке 1 толстыми линиями показаны шины данных. Пунктирными серыми линиями условно показано управление блоками, реальные сигналы управления для простоты опущены.

Отметим, что шины данных содержат сигналы, для передачи как самих данных, так и дополнительного управляющего сигнала `valid`. Это обычная практика, позволяющая отличить валидные данные от невалидных без понимания природы самих данных. Данный сигнал широко используется в интерфейсах различных блоков и протоколах (например, AXI [13] или Avalon [14]). В данном кратком описании используемых подходов мы не будем заострять внимание на использовании сигнала `valid`, так как оно естественно и широко известно.

Данные из памяти `A`, в которой хранятся элементы множества A, подаются на вход блока `f`, который реализует функцию f. Данные с выхода блока `f` снова подаются ему на вход, чтобы выполнить N вызовов функции f. Финальный результат с выхода блока `f` записывается в память `B`.

Чтобы решить конфликты, возникшие на входе и выходе блока `f`, используются мультиплексор `MUX` и демультиплексор `DEMUX`. По сигналу с блока управления мультиплексор `MUX` передает в блок `f` либо данные из памяти `A`, либо выход блока `f`. А демультиплексор `DEMUX` по сигналу от блока управления направляет результат вычисления блока `f` либо снова на вход `f` (через мультиплексор `MUX`), либо в память `B`.

В рамках данной статьи будем считать, что реализация блока `f` вычисляет результат за 1 такт и только на основе своих входов (без зависимости от состояния блока `f` на прошлом такте). Возможны различные варианты реализации блока `f`, которые не будут рассматриваться в данной статье. Мы акцентируем внимание на конвейерных реализациях внутреннего цикла.

Такт — это логическая единица времени, реальное значение которой определяется частотой тактового сигнала схемы. Чем выше тактовая частота — тем меньше времени занимает 1 такт и тем быстрее (в реальных единицах времени) схема выдает результат. Однако это значит, что меньше вычислений можно выполнить за 1 такт, именно поэтому так важно сократить накладные расходы в рамках такта.

Отметим, что в качестве блоков памяти `A` и `B` можно рассматривать не только внутреннюю блочную память ASIC или FPGA, но и любые блоки, выдающие исходные данные и потребляющие результаты: внешнюю память, каналы связи и другие.

Представленная на рисунке 1 «наивная» схема обладает существенным недостатком: она обычно работает медленней процессора общего назначения. Основная причина медленной работы «наивной» схемы заключается в том, что обработка элементов множества A выполняется последовательно: одновременно схемой может обрабатываться только один элемент множества A. Частота процессора общего назначения существенно выше частоты FPGA, поэтому без параллельной обработки данных затруднительно спроектировать схему, которая будет работать быстрее процессора.

1.2.2. *Конвейерная реализация*

Для повышения производительности схемы рассмотрим другой вариант реализации программы, обеспечивающий высокое быстродействие. В отличие от процессора общего назначения, в котором параллельная обработка реализуется либо за счет независимой работы нескольких ядер, либо за счет использования векторных инструкций, в ASIC или FPGA основным методом распараллеливания является конвейер. Это, однако, не исключает возможности реализации независимых блоков для параллельной обработки, подобных ядрам процессора.

На рисунке 2 изображена схема, которую принято называть *конвейерной*. В ней отсутствует цикл (а также мультиплексор и демультиплексор), в котором данные с выхода блока `f` передаются на вход этому же блоку. Вместо циклической структуры, выбран другой вариант — данные с одного

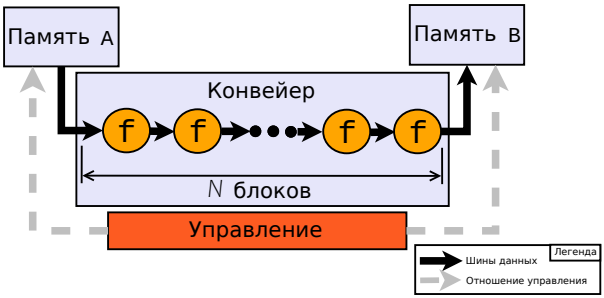

Рисунок 2. Архитектура конвейерной схемы

блока `f` передаются на вход другому идентичному блоку `f`. Каждый отдельный блок `f` принято называть *стадией* конвейера.

Конвейерная структура позволяет обрабатывать несколько вычислений внутреннего цикла параллельно. Каждый элемент множества A из памяти `A` в порядке очереди подается на вход конвейера. Когда первый элемент a_1 входит в конвейер, он попадает на первую стадию. После этого, в следующем такте результат вычислений первой стадии конвейера $f(a_1)$ будет передан на вторую стадию, а на первую стадию поступит следующий элемент a_2. Затем, в следующем такте результат второй стадии конвейера $f(f(a_1))$ будет передан на третью стадию, а на первую стадию будет подан следующий элемент a_3 и так далее. В результате на выходе с последней стадии конвейера для каждого элемента a будет получено $f^N(a)$, что и требуется вычислить.

Отметим, что после заполнения конвейера, при его полной загрузке, обрабатываются N элементов одновременно: по одному элементу на каждой стадии.

Конвейерный вариант устройства схемы имеет ряд преимуществ над «наивным»:

- Если число элементов в множестве A гораздо больше длины конвейера N, то конвейерный вариант в N раз производительней «наивного» за счет возможности параллельной обработки элементов из A.
- Отсутствуют блоки `MUX` и `DEMUX`, что уменьшает вычисления в рамках одного такта и повышает тактовую частоту схемы.
- Возможна взаимная оптимизация соседних блоков `f`, так как они связаны напрямую.

Нужно отметить, что рассматриваемая конвейерная архитектура в некоторых случаях обладает существенным недостатком: если число N велико, то размеры конвейера получаются слишком большими и конвейер может не поместиться в заданной площади микросхемы ASIC или FPGA. Или иногда бывает не нужна такая высокая производительность микросхемы, а важнее использовать меньше ресурсов микросхемы. В таких случаях обычно строят меньший конвейер, прогоняя данные через него в несколько заходов, сохраняя промежуточные результаты в памяти.

Применительно к нашей задаче размер конвейера получался больше, чем вся имеющаяся FPGA, поэтому необходимо было сократить конвейер. При этом и использование промежуточной памяти было невозможно.

1.3. Управление конвейером

При использовании конвейеров разработчику необходимо решить проблему, что делать, если память `B` не может принять данные из конвейера. Рассмотрим наиболее часто используемые подходы.

1.3.1. *Управление сигналом `enable`*

Одним из очевидных способов является использование сигнала `enable`, включающего или выключающего весь конвейер целиком (см. рисунок 3).

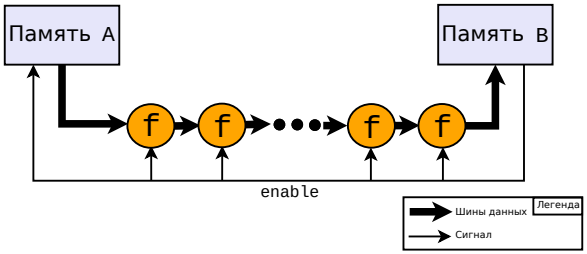

Рисунок 3. Управление сигналом enable

Несмотря на его простоту, у этого метода есть несколько недостатков:

- Длинный сигнал `enable` и зависимость всех блоков от него может значительно снижать тактовую частоту такой схемы.
- Если память `B` не готова принимать данные, то останавливается весь конвейер, хотя какие-то стадии конвейера могли бы вычислять данные, если следующие за ними стадии пусты.

1.3.2. *Управление сигналами `ready`*

Другим часто используемым способом [4] является использование между всеми стадиями сигнала `ready`, направленного против потока данных, для индикации, что следующая стадия или память `B` готовы принимать данные (см. рисунок 4).

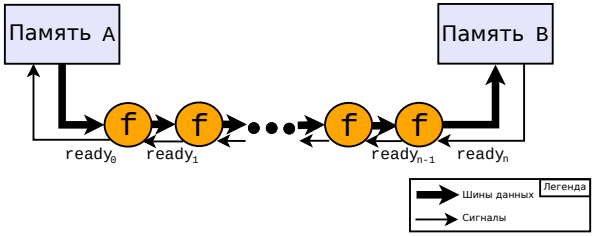

Рисунок 4. Управление сигналами ready

Такого рода сигналы широко используются в различных интерфейсах, например AXI [13], Avalon [14] и другие.

В зависимости от способа [4] реализации сигнала `ready` схема может иметь разные недостатки:

- Если сигнал $\texttt{ready}_i$ комбинационно зависит от сигнала $\texttt{ready}_{i+1}$, то фактически получается длинный сигнал `enable`, который может снижать тактовую частоту схемы.
- Если реализовывать промежуточную буферизацию на каждой стадии, чтобы разорвать комбинационную связь между соседними сигналами `ready`, то потребуются дополнительные ресурсы для реализации такой буферизации.

Тем не менее данный способ широко используется, поскольку позволяет без усилий соединять стадии конвейера и другие блоки, написанные разными разработчиками.

1.3.3. *Управление на основе кредитов*

Альтернативным вариантом, применяемым в промышленности, является кредитный механизм или просто кредиты [6, с. 57; 7]. Кредиты широко используются в различных сетевых протоколах (например, TCP/IP или PCIe), и могут быть использованы для управление конвейером.

На рисунке 5 показана схема конвейера, с управлением на основе кредитного счетчика. По сравнению с предыдущими вариантами, все

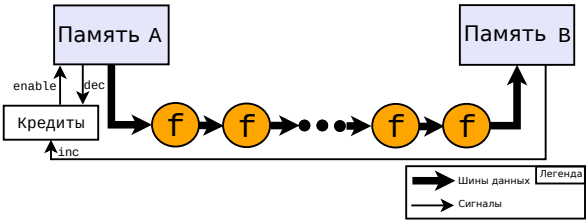

Рисунок 5. Управление кредитами

управление вынесено в отдельный блок, а конвейер реализуется без возможности остановки (неостанавливаемый конвейер). Наличие кредита у памяти `A` гарантирует, что отправленные в конвейер данные будут записаны в память `B`, не случится переполнение памяти. Если кредита на отправку данных нет, то память `A` не должна посылать новые данные в конвейер.

В результате разделяются задачи построения эффективного конвейера и недопущения переполнения памяти `B`. Отметим, что если в конкретной схеме в качестве получателя результатов конвейера используется не память, а какой-то блок с непредсказуемым заранее временем работы, то в таких случаях после конвейера устанавливают очередь (FIFO), наличие места в которой определяет значение кредитного счетчика, и такая очередь играет роль буфера между блоками, готовыми отправлять и получать данные на разных тактах, но работающих с одинаковой средней скоростью.

2. Циклический конвейер

2.1. Общая архитектура

В данном разделе будет описана предлагаемая нами схема реализации конвейера. Для того, чтобы уменьшить используемые конвейером ресурсы, надо зациклить конвейер (см. рисунок 6).

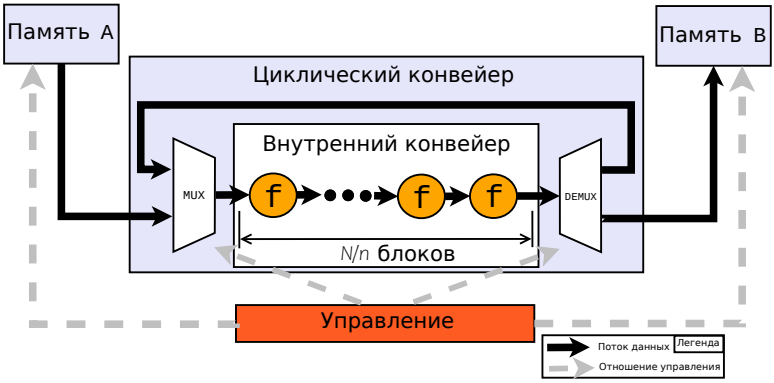

Рисунок 6. Архитектура циклического конвейера

Суть схемы циклического конвейера заключается в том, что длинный конвейер укорачивается в n раз, то есть длина внутреннего линейного конвейера становится N/n. При этом в схеме появляется цикличность: после

того как элемент данных обработан последней из N/n стадий внутреннего конвейера он снова подается на вход первой стадии внутреннего конвейера. Такой цикл выполняется n раз, в результате каждый элемент проходит $N/n \cdot n = N$ стадий внутреннего конвейера и представляет из себя искомый результат вычислений.

Циклический вариант обрабатывает элементы параллельно: в нем одновременно может обрабатываться N/n элементов множества A. И он занимает ресурсов в n раз меньше конвейерной схемы, поэтому подобрав n, его можно вместить в требуемые размеры ASIC или FPGA.

Циклический конвейер является комбинацией двух описанных выше — «наивной» и конвейерной. Как и у «наивной» реализации в нем имеются два места возможных конфликтов: блоки мультиплексор и демультиплексор, а также запись из конвейера в память `B`, которая может быть заполнена.

Эти проблемы можно попытаться решить, управляя стадиями конвейера одним из описанных ранее способов или их комбинацией, однако это оказалось проблематично. Во-первых, это управление достаточно трудоемко реализовать. И во-вторых, сигналы остановки конвейера значительно снижают рабочую тактовую частоту.

Другим важным требованием было то, что нам надо было реализовать несколько связанных между собой конвейеров: результат одного должен был передаваться на вход другому. Поэтому мы не могли останавливать поток данных. Мы хотели сделать решение, которое позволяло бы комбинировать такие конвейеры так же, как и обычные конвейеры без циклов.

Поэтому мы решили реализовать неостанавливаемые циклические конвейеры с использованием кредитного счетчика, который разрешает конфликты при доступе в память `B`. Теперь необходимо разработать решение по управлению мультиплексором `MUX` и демультиплексором `DEMUX`.

2.2. Управление мультиплексором и демультиплексором

Мы предлагаем решение, в котором мультиплексор на входе циклического конвейера управляется без необходимости понимать, какие данные сейчас обрабатывает конвейер, и других нюансов. Это существенно повышает максимально возможную тактовую частоты схемы, а значит и производительность в целом.

Снова рассмотрим циклический неостанавливаемый конвейер (рисунок 7). Мы хотим, чтобы данные, пришедшие из памяти `A` прошли n

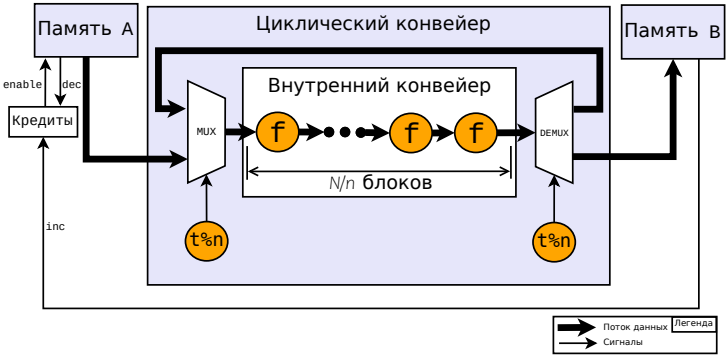

Рисунок 7. Циклический конвейер со скважностью n

кругов по конвейеру из N/n стадий. Обозначим $L := N/n$ количество стадий во внутреннем линейном конвейере. Для простоты будем считать, что каждая стадия выполняется 1 такт. И предположим, что L будет взаимопростым с n.

Отметим, что данные на шине данных снабжены дополнительным сигналом `valid`. Для полного описания предлагаемой схемы нам потребуется явно зафиксировать правила работы с этим сигналом, которые были опущены ранее для простоты изложения.

Пусть блоки конвейера работают по следующим правилам:

- Данные из памяти `A` разрешается подавать на вход циклического конвейера только в такты, номер которых делится на n. Если память `A` подает данные, то она обязана выставить сигнал `valid` в 1, иначе — в 0.
- Мультиплексор `MUX` в такты, номер которых делится на n, пропускает данные от памяти `A` на вход внутреннего конвейера. В остальные такты он пропускает данные с выхода демультиплексора `DEMUX`.
- Демультиплексор `DEMUX` в такты, номер которых делится на n, пропускает данные от внутреннего конвейера в память `B`. В остальные такты он пропускает данные на вход мультиплексора `MUX`.
- Данные записываются в память `B` только на тактах, номера которых делятся на n, и сигнал `valid` равен 1.

Отметим, что мультиплексор `MUX` и демультиплексор `DEMUX` работают синхронно: либо они оба направляют данные по циклу, либо мультиплексор направляет новое значение из памяти `A` на вход конвейера, а демультиплексор направляет вычисленное значение в память `B`.

Докажем следующую теорему:

Теорема. *Пусть длина L внутреннего конвейера взаимно проста с количеством кругов n и все блоки работают согласно описанному выше алгоритму. Тогда в память* `B` *будут записаны результаты вычислений $f^N(a)$ для всех a из памяти* `A`*.*

Доказательство. Данные a_i из памяти `A` передаются на вход циклическому конвейеру в такты, чей номер делится на n, т.е. в такты вида $i \cdot n$ $(i \geqslant 0)$.

Данные перемещаются с входа внутреннего линейного конвейера до его выхода за L тактов, т.е. они окажутся на выходе в такты вида $i \cdot n + j \cdot L$ $(j > 0)$.

Если $i \cdot n + j \cdot L$ не будет делиться на n, то данные будут поданы на вход внутреннего линейного конвейера. А если будет делиться, то записаны в память `B`.

Так как L и n взаимопростые, то $i \cdot n + j \cdot L$ будет делиться на n только тогда, когда j будет делиться на n, то есть первый раз это произойдет при $j = n$. Значит данное сделает n проходов по внутреннему линейному циклу длины L и на такте $i \cdot n + n \cdot L$ будет записано в память `B`. Что и требовалось доказать.

□

Рассмотрим построенный циклический конвейер. В отличие от обычного конвейера, который готов принимать данные каждый такт, наш циклический конвейер готов принимать данные только каждые n тактов, n называется *скважностью* (интервалом инициализации или периодичностью подачи данных) конвейера [3].

Реализовать скважность в схеме достаточно просто: поставим счетчик от 0 до $n - 1$, который увеличивается на 1 каждый такт и вместо достижении значения n сбрасывается на 0. Такой счетчик не зависит от каких-либо внешних сигналов, кроме тактового и сигнала сброса всей схемы, поэтому на реализацию скважности почти не тратится ресурсов, как и площади кристалла. При необходимости счетчики скважности можно дублировать, что приводит к меньшим задержкам, а значит повышает тактовую частоту, на которой может работать схема.

Итак, если дополнить предлагаемое решение кредитным счетчиком и FIFO на выходе, то важным преимуществом является отсутствие механизма остановки конвейера: новые данные всегда можно подавать раз в n тактов независимо от состояния системы. И результат всегда будет на выходе раз в n тактов.

Это, с одной стороны, уменьшает сложность конвейера: в результате упрощается реализация стадий конвейера, а отсутствие сигналов управления или остановки приводит к более высокой частоте схемы. С другой стороны, такие конвейеры требуют внешнего механизма защиты от переполнения памяти B или другого потребителя данных. Впрочем, это аналогично обычным неостанавливаемым конвейерам.

3. Практическая ценность

Неуправляемые конвейеры возникают в реальных задачах. Описанный в настоящей статье подход достаточно легко применить, если дан конвейер длины N без циклов, который на каждой отдельной стадии совершает одинаковые преобразования над потоком данных. Такой конвейер можно уменьшить приблизительно в n раз зациклив.

Единственным условием зацикливания является взаимная простота результирующей длины конвейера $L = N/n + k$ и количества итераций n. Число k означает количество пустых стадий — при необходимости, конвейер, полученный в результате зацикливания, можно дополнить несколькими пустыми стадиями для соблюдения условия взаимной простоты n и L.

При зацикливании конвейера количество стадий уменьшается в n раз (в идеальном случае) и, как результат, его производительность, определяемая количеством обработанных данных в единицу времени, также уменьшается в n раз. При этом одновременно с уменьшением производительности площадь на кристалле, занимаемая конвейером, также уменьшается (приблизительно в n раз).

Требуется отметить, что из-за потенциального наличия пустых стадий, конвейер может уменьшиться не ровно в n раз, а чуть меньше. Однако, если длины конвейеров достаточно большие, число пустых стадий практически не влияет на размеры.

Таким образом, предлагаемый метод зацикливания разумно использовать, когда уменьшение производительности допустимо и существует необходимость сэкономить занимаемое конвейером место в микросхеме.

Дополнительно отметим, что такого рода неостанавливаемые циклические конвейеры со скважностью больше 1 могут комбинироваться ровно так же, как и привычные конвейеры со скважностью, равной 1. Единственное, что требуется, – следить за согласованием конвейеров как по скважности, так и по фазе, и при необходимости вставлять дополнительные стадии для согласования.

В качестве примера класса задач, при решении которых возникают циклические конвейеры, можно привести реализацию различных численных, криптографических и других алгоритмов.

Для подтверждения практической ценности было проведено сравнение нового подхода управления циклическими конвейерами и традиционного на основе явного контроля потока данных с возможностью остановки различных частей конвейера. Для сравнения была выбрана задача вычисления криптографического хэша. Чтобы схема поместилась в доступную нам FPGA нам пришлось реализовать несколько циклов. Но в результате при полной нагрузке все стадии конвейера работают, что обеспечивает максимальную производительность при использовании данного количества ресурсов.

На рисунке 8 изображена схема, выбранная для эксперимента. Центральную часть занимает основной конвейер, который является цикли-

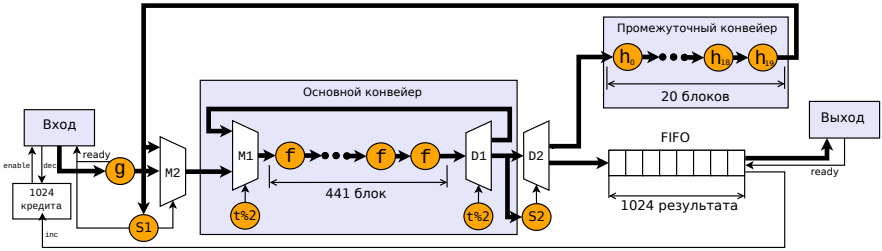

Рисунок 8. Архитектура схемы инициализации хеша

ческим. В нем 441 стадия, и он сжат в 2 раза по отношению к потенциально полному конвейеру. Таким сжатием мы смогли уложиться в отведенные для данной схемы ресурсы, сохранив приемлемую производительность. Итак, сжатие n равно 2, также как и скважность.

Мультиплексор `M1` и демультиплексор `D1` входят в основной циклический конвейер и их управление не зависит от данных. Мультиплексор `M1` каждый четный такт подает во внутренний конвейер данные от внешнего мультиплексора `M2`, а каждый нечетный – данные с выхода основного конвейера, замыкая цикл. Дополняя `M1`, демультиплексор `D1` каждый нечетный такт подает данные на вход циклическому конвейеру, а каждый четный такт – далее по схеме для последующей обработки другими блоками.

Мультиплексор `M2` является приоритетным. Он управляется блоком `S1`. Блок `S1` смотрит на данные из промежуточного конвейера и если они валидны, то мультиплексор `M2` передает их на вход циклическому конвейеру, иначе передаются данные из блока `g`.

Демультиплексор `D2` управляется блоком `S2`. `S2` вынужденно заглядывает в поступающие в него данные и на основе этого решает, куда их передать – в промежуточный конвейер или в выходное FIFO. Это свойство алгоритма формирования хэша.

Данные, прошедшие два круга в циклическом конвейере, по решению демультиплексора D2 могут поступать в промежуточный конвейер. Последний состоит из двадцати различных стадий, после преобразования в которых данные передаются опять на вход циклическому конвейеру. При этом нет необходимости задерживать промежуточный конвейер, поскольку его длина подобрана так, что данные с выхода промежуточного конвейера поступают на вход основному конвейеру только в четные такты.

Также схема содержит FIFO и кредитный счетчик, которые защищают конвейер от переполнения.

Как показано в таблице 1, предлагаемый новый способ оказался более простым в реализации (требует меньше строк кода), требует меньше ресурсов и получаемая схема работает на большей частоте.

Таблица 1. Сравнение параметров циклических конвейеров

	Частота	Логич. ячейки	Регистры	Строки кода
Традиционный подход	195 МГц	28446	19499	1031
Неостанавливаемый конвейер	233 МГц	26559	14874	912
Отношение	0.837	1.07	1.31	1.13

Для проведения экспериментов использовалась FPGA фирмы Xilinx семейства xc7a200tfbg484-2.

В эксперименте сравнивались схемы:

(*1*) с рисунка 8;

(*2*) схема, в вычислительной части такая же, как на рисунке 8, но с традиционным управлением конвейерами с помощью enable.

4. Близкие работы

Сравним наше исследование с другими близкими работами.

В классическом труде [4] в главе 23 рассматривается конвейерная организация вычислений. Показано, что при достаточно большом объеме данных (существенно больше глубины конвейера) ускорение от использования конвейера равно его длине.

Также в [4] рассматривается проблема переполнения принимающей памяти и необходимость остановки конвейера. Рассматривается два подхода к построению управляемых конвейеров: без дополнительной буферизации и с дополнительной буферизацией. В первом случае от сигнала о готовности принимающей памяти ready комбинационно зависят все стадии конвейера. Во втором случае используются очереди на регистрах глубины два (также называемые skid-buffer), позволяющий разрывать

длинную комбинационную связь ready ценой дополнительных регистров на каждом этапе конвейера.

В отличие от нашей работы, в [4] рассматриваются традиционные, базовые варианты конвейеров. Такие конвейеры являются полными, поскольку в них отсутствуют циклы. Схемы, основанные на принципах из [4], могут получаться очень большими. Наш метод позволяет сжимать рассматриваемые полные конвейеры в циклические меньшего размера, позволяя вписаться в площадь микросхемы за счет допустимого понижения производительности. Также в [4] обсуждаются механизмы остановки конвейеров, а мы предлагаем делать их неостанавливаемыми (и излагаем способ), что повышает производительность и позволяет экономить ресурсы FPGA или ASIC в случае циклических систем. Суммируя, можно утверждать, что работа [4] является источником информации по традиционным методам создания конвейеров, с которыми мы сравниваем свои варианты архитектур.

Конвейеры активно используются в промышленности для организации вычислений как в FPGA, так и в заказных микросхемах ASIC. Указанные проблемы с управлением конвейера во многих случаях решаются другим путем: используется неуправляемый конвейер, но вокруг которого строится схема на основе кредитов, не допускающая переполнения выходной памяти [6, 7].

Предлагаемая нами архитектура требует защиты от переполнения, для который изложенный, например, в [6, с. 57; 7] кредитный механизм подходит очень хорошо.

Конвейеры активно используются при построении различных процессоров. В книге [5] в главе 7 рассматриваются различные микроархитектуры процессора RISC-V, в том числе и традиционная конвейерная микроахритектура из 5 шагов: выборка инструкции (Fetch), декодирование инструкции (Decode), выполнение инструкции (Execute), операции с памятью (Memory), запись результата (WriteBack).

Процессоры могут использовать результаты вычисления одной из прошлых команд для вычисления новых, тем самым имея некоторый аналог цикла. Это роднит процессорные конвейеры с нашими. Однако стадии в процессоре обычно уникальны и не повторяются, поэтому их не сжать в циклический конвейер нашим способом. Также процессор часто требует остановки конвейера в силу наличия многотактных стадий: это препятствует построению неостанавливаемой архитектуры, на которой основывается наша работа. Тем не менее процессорные микроархитектуры являются классическими примерами конвейеризации.

Для выполнения арифметических операций с плавающей точкой также используются конвейеры. Библиотека программного IP-блока[1] [8] и другие аналогичные реализуют неуправляемую конвейерную схему для вычислений умножения. При этом блоки имеют простой интерфейс ввода и вывода нужных данных.

Так же как и у нас, в умножителе [8] используется неостанавливаемый подход, но он не имеет циклов. Все это позволяет, например, использовать конвейеризованное умножение в качестве нескольких стадий в циклических конвейерах нашей архитектуры (или даже в качестве отдельного промежуточного конвейера).

Есть и более универсальные библиотеки с большим набором функций. Например, библиотека IP-блока [9] реализует блоки различных операций с числами с плавающей точкой. Используется AXI [13] протокол (с сигналом ready для остановки потока данных при неготовности) для ввода данных и получения результата. Важной особенностью этой библиотеки является наличие параметра «Cycles per Operation», который задает скважность. Например, при значении 1 получается полностью конвейерная схема, готовая на каждом такте принимать новые данные. При значении 2 получается схема, готовая принимать данные только каждые два такта, снижает пропускную способность блока, но и примерно в два раза снижает используемые ресурсы. Именно для того, чтобы приостанавливать входные данные, когда блок не готов принимать входные данные или получатель результата не готов принимать результат, используется AXI протокол.

Так же как и у нас, в библиотеке программного IP-блока [9] используется скважность и зацикливание. В итоге [9] является примером циклической схемы, которая использует управляемый конвейер, в то время как мы предлагаем более производительный вариант – неостанавливаемую циклическую архитектуру на основе принципа взаимной простоты чисел. В общем случае наш вариант позволяет добиться большей тактовой частоты и использует меньше ресурсов микросхемы.

Конвейеры широко применяются для организации вычислений. В работе [3] исследуется какие задачи (и каким образом) могут решаться на суперкомпьютере с FPGA. Анализируется конвейерная организация вычислений и рассматриваются классы задач, для которых такая организация вычислений возможна. В работе [10] рассматриваются языки и подходы к организации вычислений в виде конвейера и встающие на этом пути проблемы.

[1] Intellectual Property block – сложный функциональный блок, который лицензируется для использования другим компаниям

Работы [3] и [10] описывает общие принципы создания конвейерных ускорителей на FPGA, которые могут использовать и нашу циклическую архитектуру.

Циклические конвейерные структуры возникают при реализации криптографических алгоритмов. Так в работе [11] рассматривается архитектура системы, состоящей из нескольких ядер, выстроенных в конвейер. При этом каждое ядро имеет циклическую «наивную» реализацию, подобную описанной нами в разделе 1. Фактически ядро в [11] является циклическим конвейером длины 1. Наш метод построения конвейеров более общий, поскольку ограничивает длину циклического конвейера только взаимной простотой чисел.

В работе [12] рассматривается конвейерная реализация алгоритма вычисления хэша SHA-3. Архитектура системы в [12] основывается на циклическом конвейере. Однако возможное переполнение конвейера не рассматривается, считается, что система снаружи сама заботится о заполнении конвейера. Также работа [12] рассматривает не общий способ построения циклических конвейеров, а только для алгоритма SHA-3. Мы в своей работе исследовали более общий случай, не зависящий от конкретного алгоритма. Также мы допускаем наличие нескольких взаимосвязанных конвейеров, тогда как в [12] конвейер единственный.

Заключение

Авторы настоящей статьи в своих исследованиях столкнулись с необходимостью реализации производительной конвейерной схемы в ограниченных ресурсами рамках. В качестве решения был разработан способ построения системы неуправляемых циклических конвейеров на основе принципа взаимной простоты чисел. Предложенное решение является компромиссом и сочетает в себе компактность с производительностью.

Преимущество циклического конвейера в том, что после введения цикла полный конвейер сжимается в n раз, в результате чего занимаемая конвейером площадь также уменьшается приблизительно в n раз. Такое сжатие позволяет уместить схему в фиксированные рамки FPGA или ASIC. Нужно упомянуть, что создание циклического конвейера также сокращает производительность в n раз по отношению к полному конвейеру. Однако часто производительность полного конвейера является избыточной, а вот ресурсов FPGA или ASIC не хватает.

Предложенный способ построения неостанавливаемых циклических конвейеров зачастую позволяет упростить разработку, сэкономить ресурсы и существенно повысить тактовую частоту схемы по сравнению с традиционными вариантами.

Список литературы

[1] Taraate V. *Logic Synthesis and SOC Prototyping.*– Singapore: Springer.– 2020.– ISBN 978-981-15-1313-8.– xix+251 pp. ↑68

[2] Kilts S. *Advanced FPGA Design: Architecture, Implementation, and Optimization.*– Wiley-IEEE Press.– 2007.– ISBN 9780470127896.– 352 pp. ↑68, 69

[3] Андреев С. С., Дбар С. А., Лацис А. О, Плоткина Е. А. *Как и почему могут быть использованы на практике суперкомпьютеры на базе FPGA.*– М.: РАН.– 2017.– ISBN 978-5-906906-61-8.– 40 с. ↑68, 70, 79, 84, 85

[4] Dally W. J., Harting R. C. *Digital Design: A Systems Approach.*– Cambridge University Press.– 2012.– ISBN 978-0-521-19950-6.– 636 pp. ↑69, 74, 75, 82, 83

[5] Harris S. L., Harris D. *Digital Design and Computer Architecture, RISC-V Edition.*– Elseiver Inc.– 2022.– ISBN 978-0-12-820064-3.– 592 pp. ↑69, 83

[6] *Intel® Hyperflex™ Architecture High-Performance Design Handbook*, ID: 683353, Version: 2021.10.04.– Intel Corporation.– 2021.– 147 pp. ↑69, 75, 83

[7] Emas M. N., Baylis A., Stitt G. *High-frequency absorption-FIFO pipelining for Stratix 10 HyperFlex*, 2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM) (Boulder, CO, USA, 2018).– 2018.– Pp. 97–100. ↑69, 75, 83

[8] *LogiCORE IP Multiplier v11.2*, DS255 March 1, 2011.– Xilinx, Inc.– 2011.– 13 pp. ↑69, 84

[9] *LogiCORE IP Floating-Point Operator v6.0*, DS816 January 18, 2012.– Xilinx, Inc.– 2012.– 41 pp. ↑69, 84

[10] Андреев С. С., Дбар С. А., Лацис А. О., Плоткина Е. А. *О применении технологий высокоуровневого синтеза к схемной реализации вычислений* // Препринты ИПМ им. М.В. Келдыша.– 2021.– ид. 34.– 19 с. ↑70, 84, 85

[11] Ioannou L., Michail H. E., Voyiatzis A. G. *High performance pipelined FPGA implementation of the SHA-3 hash algorithm*, 2015 4th Mediterranean Conference on Embedded Computing (MECO) (Budva, Montenegro, 2015).– 2015.– Pp. 68-71. ↑70, 85

[12] Wong M. M., Haj-Yahya J., Sau S. Chattopadhyay A. *A new high throughput and area efficient SHA-3 implementation*, 2018 IEEE International Symposium on Circuits and Systems (ISCAS) (Florence, Italy, 2018).– 2018.– Pp. 1-5. ↑70, 85

[13] *Vivado Design Suite: AXI Reference Guide*, UG1037 (v4.0) July 15, 2017.– Xilinx, Inc.– 2017.– 175 pp. ↑71, 75, 84

[14] *Avalon® Interface Specifications*, ID: 683091, Version: 2022.01.24.– Intel Corporation.– 2022.– 71 pp. ↑71, 75

Поступила в редакцию 08.11.2023;
одобрена после рецензирования 29.11.2023;
принята к публикации 29.11.2023;
опубликована онлайн 13.12.2023.

Рекомендовал к публикации *к.ф.-м.н. С. А. Романенко*

Информация об авторах:

Игорь Алексеевич Адамович

Научный сотрудник Института программных систем имени А. К. Айламазяна РАН. Научные интересы: метавычисления, суперкомпиляция, частичные вычисления, верификация программ, разработка на FPGA и ASIC. Принимал активное участие в разработке интерконнектов для коммуникационных сетей «Паутина» и «3D-тор» суперкомпьютера «СКИФ-Аврора»

0000-0001-9728-3024
e-mail: *i.a.adamovich@gmail.com*

Юрий Андреевич Климов

Старший научный сотрудник ИПМ им. М.В. Келдыша РАН, к.ф.-м.н. Разработчик метода специализации на основе частичных вычислений, принимал активное участие в разработке коммуникационного программного обеспечения для сетей SCI, 3D-тор суперкомпьютера «СКИФ-Аврора» и «МВС-Экспресс» суперкомпьютера К-100.

0000-0001-5081-1547
e-mail: *yuklimov@keldysh.ru*

Вклад авторов: *И. А. Адамович* – 50% (идея, методология, программное обеспечение, расследование, сбор материала, написание черновой версии, доработка и редактирование, визуализация); *Ю. А. Климов* – 50% (идея, методология, расследование, сбор материала, курирование данных, написание черновой версии, доработка и редактирование, визуализация, администрирование).

Авторы заявляют об отсутствии конфликта интересов.

EN RU PROGRAM SYSTEMS: THEORY AND APPLICATIONS ISSN 2079-3316
Research Article HARDWARE AND SOFTWARE FOR DISTRIBUTED AND SUPERCOMPUTER SYSTEMS

UDC 004.312.4
10.25209/2079-3316-2023-14-4-67-89

Cyclic pipeline systems

Igor Alekseevich **Adamovich**[1], Yuri Andreevich **Klimov**[2]

[1] Ailamazyan Program Systems Institute of RAS, Ves'kovo, Russia
[2] Keldysh Institute of Applied Mathematics of RAS, Moscow, Russia
[1]✉*i.a.adamovich@gmail.com*

Abstract. One of the most efficient ways to organize calculations on ASIC or FPGA is the creation of non-stallable pipelines. However, for some computing circuits, the resulting pipeline may be too large for available ASIC or FPGA resources. The authors propose a method for constructing cyclic pipelines, in which data flow control is based on counters and does not depend on the data being transmitting along the pipeline. We proposed the method makes it possible to build more compact non-stallable pipelines. One of the main details of method is to use cycle ratio equal to the number of times the data must go through the loop, after which the pipeline converts the data into the desired result. (*In Russian*).

Key words and phrases: pipeline, ASIC, FPGA, integrated circuit, periodicity, queue, credit

2020 *Mathematics Subject Classification*: 94-05; 94C30

For citation: Igor A. Adamovich, Yuri A. Klimov. *Cyclic pipeline systems*. Program Systems: Theory and Applications, 2023, **14**:4(59), pp. 67–89. (*In Russ.*). https://psta.psiras.ru/read/psta2023_4_67-89.pdf

References

[1] V. Taraate. *Logic Synthesis and SOC Prototyping*, Springer, Singapore, 2020, ISBN 978-981-15-1313-8, xix+251 pp.

[2] S. Kilts. *Advanced FPGA Design: Architecture, Implementation, and Optimization*, Wiley-IEEE Press, 2007, ISBN 9780470127896, 352 pp.

[3] S. S. Andreev, S. A. Dbar, A. O Lacis, E. A. Plotkina. *How and why FPGA-based Supercomputers can be used in Practice*, RAN, M., 2017, ISBN 978-5-906906-61-8 (in Russian), 40 pp.

[4] W. J. Dally, R. C. Harting. *Digital Design: A Systems Approach*, Cambridge University Press, 2012, ISBN 978-0-521-19950-6, 636 pp.

[5] S. L. Harris, D. Harris. *Digital Design and Computer Architecture, RISC-V Edition*, Elseiver Inc, 2022, ISBN 978-0-12-820064-3, 592 pp.

[6] *Intel® Hyperflex™ Architecture High-Performance Design Handbook*, ID: 683353, Version: 2021.10.04, Intel Corporation, 2021, 147 pp.

[7] M. N. Emas, A. Baylis, G. Stitt. "High-frequency absorption-FIFO pipelining for Stratix 10 HyperFlex", 2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM) (Boulder, CO, USA, 2018), 2018, pp. 97–100.

[8] *LogiCORE IP Multiplier v11.2*, DS255 March 1, 2011, Xilinx, Inc, 2011, 13 pp.

[9] *LogiCORE IP Floating-Point Operator v6.0*, DS816 January 18, 2012, Xilinx, Inc, 2012, 41 pp.

[10] S. S. Andreev, S. A. Dbar, A. O. Lacis, E. A. Plotkina, *Preprinty IPM im. M.V. Keldysha*, 2021, id. 34 (in Russian), 19 pp.

[11] L. Ioannou, H. E. Michail, A. G. Voyiatzis. "High performance pipelined FPGA implementation of the SHA-3 hash algorithm", 2015 4th Mediterranean Conference on Embedded Computing (MECO) (Budva, Montenegro, 2015), 2015, pp. 68-71.

[12] M. M. Wong, J. Haj-Yahya, S. Chattopadhyay A. Sau. "A new high throughput and area efficient SHA-3 implementation", 2018 IEEE International Symposium on Circuits and Systems (ISCAS) (Florence, Italy, 2018), 2018, pp. 1-5.

[13] *Vivado Design Suite: AXI Reference Guide*, UG1037 (v4.0) July 15, 2017, Xilinx, Inc, 2017, 175 pp.

[14] *Avalon® Interface Specifications*, ID: 683091, Version: 2022.01.24, Intel Corporation, 2022, 71 pp.