OSTIS-2012
(Open Semantic Technologies for Intelligent Systems)

УДК 004.822:514

РЕКОНФИГУРИРУЕМЫЕ ВЫЧИСЛЕНИЯ КАК СРЕДСТВО ОПТИМИЗАЦИИ РЕШЕНИЙ ПОДКЛАССА ЗАДАЧ

Абрамов Н.В.[*], Иванюк А.А[*]

[*]*Белорусский государственный университетинформатики и радиоэлектроники, г. Минск, Республика Беларусь*

nickolaib2004@gmail.com

ivaniuk@bsuir.by

Решение любой задачи можно представить в виде алгоритма. Алгоритм задает последовательность шагов. В некотором смысле алгоритм можно рассматривать как правила, которым нужно следовать для достижения конкретной цели. Однако правила сами по себе совершенно бесполезны при отсутствии некоторого объекта, который мог бы выполнять эти правила и тем самым получать некоторый результат.

Ключевые слова:реконфигурируемые вычисления, помехоустойчивое кодирование, обобщение подкласса задач.

Введение

Алгоритм можно представлять различным образом и с различной детализацией, при чем, уровень детализации должен быть достаточным для того чтобы исполнитель мог выполнить абсолютно все действия предписанные алгоритмом. Если в описании алгоритма будет иметь место хотя бы одно непонятное действие, алгоритм корректным являться не будет. Другими словами алгоритм и исполнитель алгоритма должны оперировать общими, понятными друг другу терминами. Применительно к вычислительной технике можно сказать, что например, невозможно выполнить некоторый алгоритм, представленный в виде блок-схем, или в виде текстового описания. Микропроцессоры не умеют оперировать такими сложными объектами. Поэтому в конечном итоги любое описание в любом виде, проецируется на набор инструкций, поддерживаемый конкретным микропроцессором. Однако не все задачи могут быть эффективно представлены в виде программы. В этом случае использование реконфигурируемых вычислений добавляет существенно больше возможностей для отображения алгоритмов на вычислительные устройства. Эффективное отображение прикладной области на реконфигурируемое устройство можно выполнить при помощи онтологий, другими словами необходимо провести формализацию предметной области и представить ее в виде семантической сети.

1. Микропроцессор общего назначения

В качестве примера микропроцессора общего назначения рассмотрим виртуальный MIPS процессор WinMIPS64. Имитатор данного процессора находиться в открытом доступе, более подробную информацию о нем можно найти в [Hennesy, 2006]. WinMIPS64 – 64 битный RISC процессор, с конвейерной обработкой команд. Структура конвейера данного микроконтроллера представлена на рисунке 1. Конвейер состоит из пяти основных шагов.

IF (Instruction Fetch) шаг на котором происходит извлечение инструкции и памяти программы в регистр инструкции. При этом счетчик команд увеличивается на единицу.

ID (Instruction Decode) шаг, на котором имеет место декодирование инструкции извлечение двух операндов, которые подаются на вход АЛУ. Кроме того, на данном шаге выполняются операции ветвления.

EX (Execution) шаг предназначен для выполнения некоторых действий над операндами, которые были поданы на шаге ID в АЛУ. В зависимости от кода инструкции на данном шаге может иметь место сложение, вычитание, умножение, сдвиг, работа с вещественными числами и т.д.

MEM (MEMoryAccess) шаг, на котором выполняются инструкции по работе с памятью, такие как ld, sd, lb, sb и т.д. С полным набором поддерживаемых инструкций можно ознакомиться в [Scott, 2003].

WB (Write Back) шаг, на котором данные полученные из памяти или как результат вычислений в АЛУ, записываются в регистр назначения. Запись данных на этой стадии происходит в течение первой половины такта, таким образом, на второй половине такта на стадии ID можно прочитать новое значение регистра.

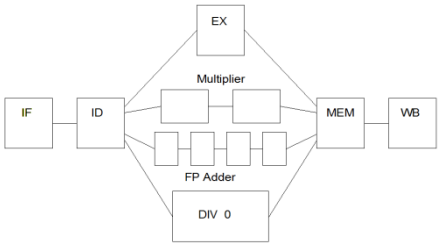

Рисунок 1 - Структура конвейера микропроцессора WinMIPS64

Данные пять ступеней конвейера, и заданный набор инструкций позволяют реализовать практически любую задачу.

2. Применение микропроцессора общего назначения для решения задач

Рассмотрим три алгоритма из области помехоустойчивого кодирования: использование бита четности, код Хэмминга, циклический код.

Данные алгоритмы основаны на внесении избыточных проверочных бит в сообщение, что позволяет в случае ошибок в системе связивосстанавливать потерянные данные или, по крайней мере, сообщать об имеющейся в них ошибке [Блейхут, 1986].

Первый рассматриваемый алгоритм – использование бита четности для определения наличия ошибки нечетной кратности.

Имея информационное сообщение $m = \{i_3 i_2 i_1 i_0\}$ можно вычислить бит четности по следующей формуле:

$$p = i_3 \oplus i_2 \oplus i_1 \oplus i_0$$

Совмещая информационное сообщение с битом четности, передаваемое сообщение можно представить в виде $c' = \{i_3 i_2 i_1 i_0 p\}$. Расширяя сообщение до одного байта, по средствам добавления нулей в старшие разряды получаем окончательный код $c = \{000 i_3 i_2 i_1 i_0 p\}$. Восьми битный код подходит для хранения в регистрах большинства микропроцессоров.

Программа для микропроцессора MIPS64, формирующая кодовое сообщение c,представлена в листинге 1.

```
1:   ;Листинг 1 – Бит четности
2:   .data
3:   Message:     .word 7
4:   Count:               .word 8
5:   Result:      .word 0
6:   Mask:                .word 1
7:   MaskBit1:    .word 1
8:   MaskBit2:    .word 2
9:   MaskBit3:    .word 4
10:  MaskBit4:    .word 8
11:  .text
12:  main:
13:  ;r4 - исходноесообщение
14:  Ld     r4,Message(r0)
15:  ;r5 - счетчик
16:  ld    r5,Count(r0)
17:  ;r6 - маска
18:  ld    r6,Mask(r0)
19:  ;r7 - результат
20:  ld    r7,Result(r0)
21:  ;определяеминформационныебиты
22:  ld    r6, MaskBit1(r0)
23:  and   r8, r4, r6
24:  beqz  r8, skip_i1
25:  ld r9, Mask(r0)
26:  skip_i1:
27:  ld    r6, MaskBit2(r0)
28:  and   r8, r4, r6
29:  beqz  r8, skip_i2
30:  ld    r10, Mask(r0)
31:  skip_i2:
32:  ld    r6, MaskBit3(r0)
33:  and   r8, r4, r6
34:  beqz  r8, skip_i3
35:  ld    r11, Mask(r0)
36:  skip_i3:
37:  ld    r6, MaskBit4(r0)
38:  and   r8, r4, r6
39:  beqz  r8, skip_i4
40:  ld    r12, Mask(r0)
41:  skip_i4:
42:  xor   r13, r9, r10
43:  xor   r13, r13, r11
44:  xor   r13, r13, r12
45:  dsll  r4, r4, 1
46:  xor   r6, r4, r13
47:  halt
```

Очевидно, можно привести множество других реализаций, однако был выбран именно такой способ, для демонстрации сходства реализаций алгоритмов одной предметной области.

Программа выполняет следующие действия. Исходное сообщение помещается в регистр r4. Затем в регистры r9, r10, r11, r12 помещаются биты данного сообщения. Имея отдельные биты сообщения нетрудно вычислить их сумму по модулю два используя инструкцию xor. Определив бит четности, формируется окончательное сообщение по средствам сдвига исходного и добавления бита четности в младший разряд при помощи инструкции xor.

Данная микропрограмма состоит из 25 инструкций. Так как обработка команд происходит конвейерным образом, это приводит к тому, что одновременно в микропроцессоре выполняется до пяти команд, находящихся на различных этапах конвейера. Такая псевдопараллельная работа приводит к тому, что 25 инструкций, с учетом того, что на одну команду требуется 5 тактов, выполняются за 38 тактов, а не за 125, если бы имела места последовательная обработка команд. Таким образом, на одну команду в данной микропрограмме было затрачено в среднем 1.52 такта.

Второй алгоритм – код Хэмминга (7,4). Данный код позволяет исправлять ошибку единичной кратности. Общая длина кода составляет семь бит.

Среди этих бит четыре бит информационные и три бита проверочные.

Имея сообщение m, проверочные биты можно вычислить по следующей формуле:

$$r_0 = i_0 \oplus i_1 \oplus i_3;$$
$$r_1 = i_0 \oplus i_2 \oplus i_3;$$
$$r_2 = i_1 \oplus i_2 \oplus i_3.$$

Определив проверочные биты можно сформировать кодовое слово $c' = \{i_3 i_2 i_1 i_0 r_2 r_1 r_0\}$, добавляя 0 в старший разряд получаем представление кода размером в один байт $c = \{0 i_3 i_2 i_1 i_0 r_2 r_1 r_0\}$. Программа для микропроцессора MIPS64, формирующая кодовое сообщения c используя код Хэмминга (7,4), представлена в листинге 2.

```
1:  ;Листинг 2 – Код Хэмминга
2:  .data
3:  Message:   .word 13
4:  Count:             .word 8
5:  Result:    .word 0
6:  Mask:              .word 1
7:  MaskBit1:  .word 1
8:  MaskBit2:  .word 2
9:  MaskBit3:  .word 4
10: MaskBit4:  .word 8
11: .text
12: main:
13: ld  r4,  Message(r0)
14: ld  r5,  Count(r0)
15: ld  r6,  Mask(r0)
16: ld  r7,  Result(r0)
17: ld  r8,  Result(r0)
18: ld  r9,  Result(r0)
19: ld  r10, Result(r0)
20: ld  r11, Result(r0)
21: ld  r12, Result(r0)
22: ld  r6, MaskBit1(r0)
23: and        r8, r4, r6
24: beqz       r8, skip_i1
25: ld  r9,  Mask(r0)
26: skip_i1:
27: ld  r6,  MaskBit2(r0)
28: and        r8,  r4, r6
29: beqz       r8,  skip_i2
30: ld         r10, Mask(r0)
31: skip_i2:
32: ld         r6,  MaskBit3(r0)
33: and        r8,  r4, r6
34: beqz       r8,  skip_i3
35: ld         r11, Mask(r0)
36: skip_i3:
37: ld         r6, MaskBit4(r0)
38: and        r8, r4, r6
39: beqz       r8, skip_i4
40: ld         r12, Mask(r0)
41: skip_i4:
42: ;определяемпроверочные биты
43: xor r13, r9,  r10
44: xor r13, r13, r12
45: xor r14, r9, r11
46: xor r14, r14, r12
47: xor r15, r10, r11
48: xor r15, r15, r12
49: ;формируем закодированное сообщение
50: dsll       r6,  r4,  3
51: dsll       r14, r14, 1
52: dsll       r15, r15, 2
53: xor r8,  r13, r14
54: xor r8,  r8,  r15
55: xor r6,  r6,  r8
56: halt
```

Анализируя листинг программы 2, можно выявить некоторые ее сходства с листингом программы 1. Различие заключается в вычислении проверочных бит и формировании кодового слова. Участки кода по извлечению информационных бит из регистра идентичны.

Данная микропрограмма состоит из 37 инструкций. Время выполнения программы составило 50 тактов. Таким образом, на одну команду пришлось 1.351 такта.

Третий алгоритм – циклический код [Уильямс, 1993]. Для вычисления циклического кода необходимо задать порождающий простой полином. В данной работе, для того, чтобы можно было воспользоваться тем же самым исходным сообщением, как и в предыдущих двух алгоритмах, используется полином четвертой степени $p(x) = x^4 + x + 1$. Имея простой полином можно построить порождающую матрицу, при помощи которой можно рассчитать код для заданного сообщения. Порождающая матрица для данного полинома имеет следующий вид:

$$G = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \end{bmatrix}.$$

Используя порождающую матрицу можно вычислить код следующим образом:

$$\mathrm{c} = m \cdot G$$

Обратив внимание на то, что в левой части матрицы находиться единичная матрица, можно упростить задачу вычисления кода, рассчитывая только проверочные биты по следующей формуле:

$$r_0 = i_3 \oplus i_0;$$
$$r_1 = i_3 \oplus i_2 \oplus i_0;$$
$$r_2 = i_2 \oplus i_1;$$
$$r_3 = i_1 \oplus i_0.$$

Имея проверочные биты можно сформировать кодовое слово $c = \{i_3 i_2 i_1 i_0 r_3 r_2 r_1 r_0\}$.

Микропрограмма, формирующая кодовое сообщение c, используя алгоритм формирования циклического кода, представлена в листинге 3.

```
1:  ;Листинг 3 – Циклический код
2:  .data
3:  Message:       .word 13
4:  Count: .word 8
5:  Result:        .word 0
6:  MaskBit1:      .word 1
7:  MaskBit2:      .word 2
8:  MaskBit3:      .word 4
9:  MaskBit4:      .word 8
10: Mask:  .word 1
11: .text
12: main:
13: ld             r4,  Message(r0)
14: ld             r5,  Count(r0)
15: ld             r6,  Mask(r0)
16: ld             r7,  Result(r0)
17: ld             r6,  MaskBit1(r0)
18: and            r8,  r4, r6
```

```
19: beqz          r8,  skip_i1
20: ld            r9,  Mask(r0)
21: skip_i1:
22: ld            r6,  MaskBit2(r0)
23: and           r8,  r4, r6
24: beqz          r8,  skip_i2
25: ld            r10, Mask(r0)
26: skip_i2:
27: ld            r6,  MaskBit3(r0)
28: and           r8,  r4, r6
29: beqz          r8,  skip_i3
30: ld            r11, Mask(r0)
31: skip_i3:
32: ld            r6,  MaskBit4(r0)
33: and           r8,  r4, r6
34: beqz          r8,  skip_i4
35: ld            r12, Mask(r0)
36: skip_i4:
37: xor           r13, r9,  r12
38: xor           r14, r12, r10
39: xor           r14, r14, r9
40: xor           r15, r11, r10
41: xor           r16, r12, r11
42: dsll          r6,  r4,  4
43: dsll          r14, r14, 1
44: dsll          r15, r15, 2
45: dsll          r16, r16, 3
46: xor           r8,  r14, r15
47: xor           r8,  r8,  r16
48: xor           r8,  r8,  r13
49: xor           r6,  r8,  r6
50: halt
```

Микропрограмма, приведеннаявлистинге 3, состоитиз 33 инструкций, которые выполняются на микропроцессоре MIPS64 за 47 тактов. Таким образом, на выполнение одной инструкции в среднем требуется 1.424 такта.

Имея три реализации можно заметить явные схожие черты у них. Можно оценить среднее время выполнения команды по трем реализациям – 1.432 такта. Кроме того можно легко заметить недостаток микропроцессора MIPS64 для выполнения некоторых действий. В частности извлечение конкретного бита из регистра общего назначения требует четыре инструкции, одна из которых условный переход, который приводит к сбросу конвейера:

```
1:  ld            r6, MaskBit1(r0)
2:  and           r8, r4, r6
3:  beqz          r8, skip_i1
4:  ld            r9, Mask(r0)
5:  skip_i1:
```

Реализовав данную операцию в виде одной команды можно как минимум увеличить скорость работы данного участка кода на 4.296 такта.

Однако возникает закономерный вопрос, каким образом можно добавить желаемую функциональность, наибольшим образом подходящую для реализуемого алгоритма?

3. Реконфигурируемые вычисления

Внедрение реконфигурируемых вычислений позволяет добиться желаемого решения. Приведем листинг программы расчета бита четности в том виде, в котором нам бы хотелось его видеть.

```
1:  ;Листинг 4 – Бит четности
      (реконфигурация)
2:  .data
3:  Message:       .word 7
4:  Count:         .word 8
5:  Result:        .word 0
6:  Mask:   .word 1
7:  .text
8:  main:
9:  ld             r4, Message(r0)
10: ld             r5, Count(r0)
11: ld             r6, Mask(r0)
12: ld             r7, Result(r0)

13: exbt    r9,  r4, 0
14: exbt           r10, r4, 1
15: exbt           r11, r4, 2
16: exbt           r12, r4, 3

17: xor            r13, r9, r10
18: xor            r13, r13, r11
19: xor            r13, r13, r12
20: dsll    r4, r4, 1
21: xor            r6, r4, r13
22: halt
```

Влистинге 4 содержитсяинструкция`exbt` (**ex**tract**b**i**t**), котораяпоформату полностью соответствует формату команд принятому в данном микропроцессоре, то есть первый операнд задает приемник, второй и третий операнды задают операнды, подаваемые на входы АЛУ. Результатом работы данной инструкции, является занесение указанного в третьем операнде бита, второго операнда в младший разряд первого. Другими словами имеет место извлечение указанного бита из указанного регистра.

Реконфигурируемые вычисления позволяют изменять архитектуру устройства в процессе выполнения за счет перепрограммирования внутренних конфигурационных ячеек. Конфигурационные ячейки управляют связями между базовыми элементами чипа, такими как вентили «и-не», «или», «и» и т.д., или такими как LUT-таблицы, которые могут функционировать как память или логическая функция. Другими словами реконфигурируемые устройства поддерживают определенные средства для модификации своей архитектуры в процессе работы [Compton, 2002], [Bobda, 2007].

Одним из достоинств реконфигурируемых устройств является их способность реализовать практически любую требуемую архитектуру. Можно без изменений взять описание микропроцессора MIPS64 и отобразить его на реконфигурируемое устройство, и оно станет функционировать согласно той логике, которая предусмотрена данной архитектурой. Очевидно, что изначальное описание не предусматривает наличие таких инструкций как exbt и какую-либо реконфигурацию.

Однако, имея описание микропроцессора на высокоуровневом языке описания аппаратуры, таком как VHDL, можно модифицировать некоторые элементы микропроцессора, таким образом, чтобы он мог воспользоваться возможностями, предоставляемыми реконфигурируемым устройством.

При использовании динамической реконфигурации кроме некоторого набора команд возникает гораздо более широкое понятие – конфигурация. При чем, различные конфигурации могут предоставлять различные наборы команд, которые в свою очередь могут быть использованы в одной и той же программе. Принцип работы схож с принципом работы с памятью, когда при работе с памятью используются одни и те же адреса, но в различных банках памяти.

Для достижения данного эффекта команды, которыми оперирует программист можно разбить на два класса: команды реконфигурации, обычные команды. При выполнении такой программы, вычислительное устройство при обработке команды реконфигурации выполняет перепрограммирование реконфигурируемой части устройства, а при обработке обычных команд – передает управление текущей конфигурации.

Очевидно, что переключение конфигурации достаточно длительный процесс. При реконфигурации необходимо дождаться завершения инструкций находящихся в конвейере, так как реконфигурация фактически изменяет элементы конвейера. Кроме того необходимо время на собственное изменение внутренних конфигурационных ячеек устройства. Таким образом, в целом процесс может затянуться. На сегодняшний день существует несколько реконфигурируемых архитектур, поддерживающие достаточно быструю реконфигурацию, например адаптивная архитектура описанная в [Watson, 2002].

Принимая во внимание вышенаписанное, и, имея реконфигурируемый процессор с двумя конфигурациями, листинг 4, может быть переписан следующим образом:

```
23: ;Листинг 5 – Бит четности
     (реконфигурация)
24: .data
25: Message:        .word 7
26: Count:  .word 8
27: Result:         .word 0
28: Mask:   .word 1
29: .text
30: main:
31: ld              r4, Message(r0)
32: ld              r5, Count(r0)
33: ld              r6, Mask(r0)
34: ld              r7, Result(r0)
35: conf    1
36: exbt    r9,  r4, 0
37: exbt            r10, r4, 1
38: exbt            r11, r4, 2
39: exbt            r12, r4, 3
40: conf    0
41: xor             r13, r9, r10
42: xor             r13, r13, r11
43: xor             r13, r13, r12
44: dsll    r4, r4, 1
45: xor             r6, r4, r13
46: halt
```

В листинге 4, инструкция `conf`, относится к категории инструкций конфигурации, и приводит к переключению текущей конфигурации на указанную в единственном операнде. При этом инструкция exbt может иметь такой же код как, например, и инструкций xor, что не приведет ни к каким конфликтам, так как мнемоника инструкции интересна только программисту, а функциональность при таком подходе определяется как конфигурацией, так и инструкцией.

Упрощенная логическая структура такого конвейера представлена на рисунке 2.

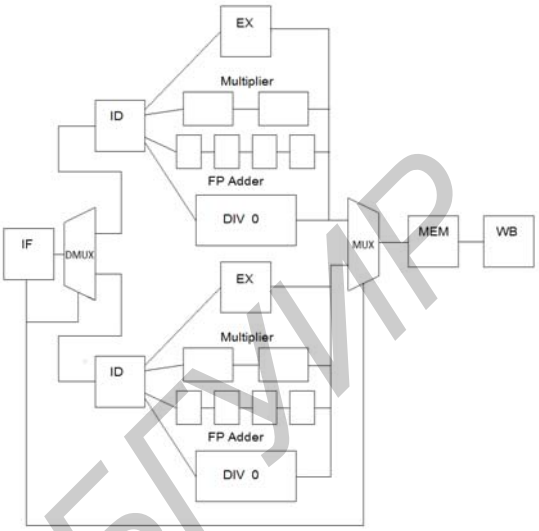

Рисунок 2 - Реконфигурируемый конвейер

В данной архитектуре на этапе извлечения команды определяется необходимая конфигурация, которая определяет декодер и АЛУ используемые для дальнейшей обработки инструкций.

На рисунке 3 приведены размеры программ с использованием и без использования реконфигурации.

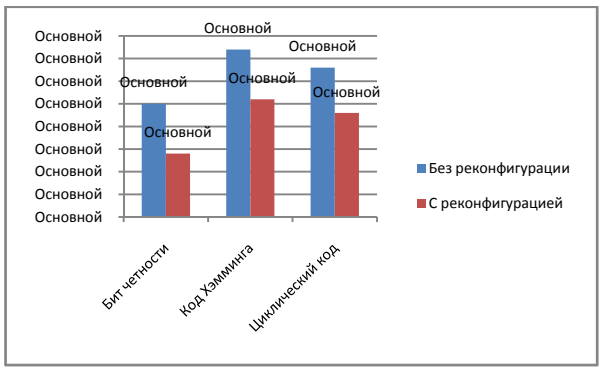

Рисунок 3 - Сравнение размеров программ без и с использованием реконфигурации

На рисунке 4 приведено сравнение времени выполнения в тактах приведенных выше программ с и без использования реконфигурации.

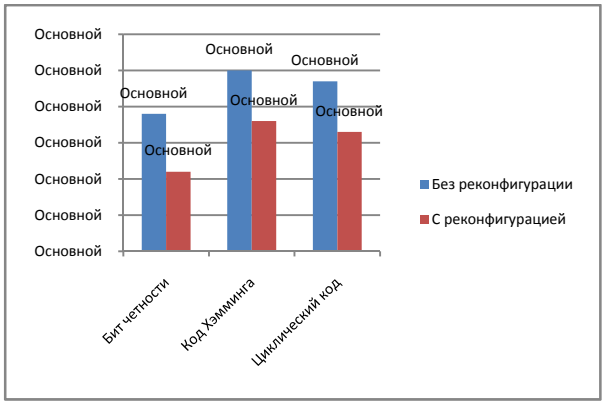

Рисунок 4 - Время выполнения программы без и с использованием реконфигурации

Заключение

Внедрение реконфигурируемых вычислений вносит существенную гибкость в систему, позволяя для различных подзадач приложения выделять специализированные аппаратные элементы, которые, за счет динамической реконфигурации могут использовать одни и те же ресурсы.

Для конечного пользователя такого устройства, добавляется необходимость определять требуемую конфигурацию и следить, за тем, какие инструкции используются в выбранной конфигурации. Однако та гибкость, и возможность оптимизировать специфические участки предметной области, делают вполне приемлемыми новые требования к написанию программ.

Библиографическийсписок

[Henessy, 2006]Computer Architecture: A Quantitative Approach, 4th Edition / Henessy – Morgan Cauffman, 2006 – 704 p.

[Scott, 2003]Using WinMIPS64 Simulator – Электронныйресурс: Режимдоступа: ftp://ftp.computing.dcu.ie/pu b/resources/crypto/winmipstut.doc

[Блейхут, 1986] Теория и практика кодов, контролирующих ошибки / Р. Блейхут. – М.: Мир, 1986. 576 с.

[Уильямс, 1993] Элементарное руководство по CRC-алгоритмам обнаружения ошибок / Р. Уильямс. – 1993. 36 с.

[Compton, 2002]ReconfigurableComputing: ASurveyofSystemsandSoftware, / ACMComputingSurveys, June 2002, pp. 171-210.

[Bobda, 2007]Introduction to Reconfigurable Computing. Architectures, Al-gorithms and Applications / C. Bobda – Dordrecht: Springer, – 2007 – 375 p.

[Watson, 2002]Adaptive Computing IC Technology Enables SDR and Multi-functionality in next-generation wireless devices / J. Watson // SDR 02 Technical Conference and Product Exposition, – 2002.

RECONFIGURABLE COMPUTING AS MEANS OF SOLUTIONS' OPTIMIZATION OF TASKS OFCOMMON FIELD

Abramov N. V.*, Ivaniuk A.A.*

**Belarusian State University of Informatics and Radioelectronics, Minsk, Republic of Belarus*

nickolaib2004@gmail.com, ivaniuk@bsuir.by

The solution to any problem can be represented in the form of the algorithm. The algorithm defines a sequence of steps. The algorithm can be regarded as rules to follow to achieve a particular goal. However, the rules themselves are quite useless in the absence of an object that could meet these rules and thereby obtain a result.

Introduction

The algorithm can be represented in different ways and with varying detail,the level of detail is determined by executor, because it might be able to do absolutely all actions prescribed by the algorithm. If there is at least one incomprehensible actin the description of the algorithm, it will not be considered correct. In other words, the algorithm andit's executor must operate on common, understandable to each otherterms. With the respect to computer technology we can say that for example, it is impossible to perform a certain algorithm, presented in the form of block diagrams, or as a text description. Microprocessors are not able to handle such complex objects.Therefore, the final outcome of any description in any form is projected onto the set of instructionssupported by a specific microprocessor. However, not all problems can be effectively represented in the form of the program. In this case, the use of reconfigurable computing adds significantly more opportunities to map algorithms on computing devices.Efficient mapping of applied problems on reconfigurable device can be performed using ontology, in other words it is necessary to formalize the domain and present it as a semantic network.

Main Part

This paper discusses the use of reconfigurable computing to optimize the solutions belonging to the same domain. Examples are taken from the field of antinoise coding: parity bit check, Hamming code (7,4), CRC code.

For each selected algorithm source code for microprocessor simulator MIPS64 is listed.

Using statistics provided by algorithm realization common parts of each realization is determined, and using this information an optimization for each of them is suggested. The solution is based on the use of reconfigurable computing. Reconfiguration is introduced in the source pipeline of the processor to fit better the algorithm.

Conclusion

Reconfigurable computing introduces significant flexibility into the system, providing specialized hardware elements for specific subtasks of the program that, due to the dynamic reconfiguration can even share same resources.

The use of the reconfigurable device for the end user is added up to the selection of the appropriate configuration.