Компьютерные инструменты в образовании, 2022
№ 3: 82–93
УДК: 004.31:004.051
http://cte.eltech.ru
doi:10.32603/2071-2340-2022-3-82-93

ИССЛЕДОВАНИЕ ОПЕРАЦИИ СОВМЕЩЁННОГО УМНОЖЕНИЯ-ВЫЧИТАНИЯ НА ПРОЦЕССОРЕ Baikal-T

Архипов И. С.[1], студент, ✉ arkhipov.iv99@mail.ru

[1]Санкт-Петербургский государственный университет,
Университетский пр., д. 28, Старый Петергоф, 198504, Санкт-Петербург, Россия

Аннотация

Данная статья является продолжением цикла исследований, посвящённых исследованию совмещённых операций на процессоре Baikal-T. В ней рассмотрены различные особенности работы команды совмещённого умножения-вычитания. Приведены разнообразные примеры использования команды, сделаны вычисления и сформулированы выводы. Также описаны ситуации, в которых использование команды совмещённого умножения-вычитания оправдано, и ситуации, в которых её использование не выгодно относительно времени работы программы.

Ключевые слова: *MIPS, процессор Байкал, умножение-вычитание, оптимизация, ассемблер.*

Цитирование: Архипов И. С. Исследование операции совмещённого умножения-вычитания на процессоре Baikal-T // Компьютерные инструменты в образовании. 2022. № 3. С. 82–93. doi: 10.32603/2071-2340-2022-3-82-93

1. ВВЕДЕНИЕ

С самого зарождения вычислительной техники высокопроизводительные вычисления являются одной из важнейших задач и теоретических и практических исследований и разработок в области информатики. Высокопроизводительные системы используются в самых разных областях для решения практических задач: метеорология, экономика, нефтегазовая промышленность и другие.

По мере развития сферы высокопроизводительных вычислений были разработаны самые разные технологии. Широкое применение получили многопроцессорные вычисления с распараллеливанием задач. Исследования велись и в рамках вычислений на одном процессоре. Так был разработан внутренний параллелизм и конвейеризация. Одновременно с увеличением производительности аппаратного обеспечения возрастало и качество компиляции программ. Были изучены и опубликованы самые разные методы анализа и оптимизации исходного программного кода.

В процессе развития высокопроизводительных вычислений в однопроцессорных системах были реализованы низкоуровневые команды совмещённых арифметических операций. Примером такой операции является совмещённое умножение-вычитание. Эта команда умножает два числа и вычитает его из регистра-аккумулятора. Совмещённое

умножение-вычитание позволяет ускорить вычисление множества распростанённых задач. Другим примером подобной операции является совмещённое умножение-сложение.

Операция совмещённого умножения-вычитания не настолько популярна, как схожая с ней операция совмещённого умножения-сложения. Например, вторая включена в стандарт IEEE 754-2019 [1] в отличие от первой. Однако команда совмещённого умножения-вычитания реализована в некоторых архитектурах. Например, она включена в архитектуры MIPS [2], на базе которой сделан российский процессор Baikal-T[1].

В первой работе [3] были исследованы особенности команды совмещённого умножения-сложения. В данной статье рассматривается операция совмещённого умножения-вычитания. По сравнению с предыдущей работой в этом исследовании были расширены имеющиеся эксперименты для исследования особенностей команды и добавлены новые. Дополнительно для каждого эксперимента по результатам измерений проведено сравнение с подобным экспериментом для операции совмещённого умножения-сложения. В конце статьи на основе проделанной работы сделаны выводы об эффективности команды совмещённого умножения-вычитания.

2. ОБЗОР

С момента выхода первой работы новых публикаций на данную тематику не было. Имеется несколько статей, посвящённых импортозамещению и российскому рынку электроники [4, 5], однако они мало имеют отношения к теме данной публикации.

В данном разделе стоит уделить внимание предыдущей работе [3]. В этой статье были исследованы особенности работы команды совмещённого умножения-сложения на процессоре Baikal-T. Был приведён пример умножения матриц, на котором проявлялась особенность работы команды, а также несколько экспериментов с измерениями для разных случаев использования операции. В конце был сделан вывод об эффективности использования инструкции. В настоящей статье автор придерживается той же структуры для исследования операции совмещённого умножения-вычитания, но содержательно был увеличен объём измерений для каждого эксперимента и количество экспериментов, что позволяет сделать более глубокие выводы об эффективности команды.

3. ИЗМЕРЕНИЯ ВРЕМЕНИ

В предыдущей работе в качестве примера для измерения времени было взято умножение матриц. Здесь же взят похожий пример, только в самом вложенном цикле сложение заменено на вычитание. Код приведён в Приложении 1. При работе программы бОльшая часть времени приходится на строки 27–29. Рассмотрен ассемблерный код этих строк, порождённый различными компиляторами. Файлы с ассемблерным кодом полностью всей программы из Приложения 1 выложены в отдельном репозитории [6]. Код цикла в строках 27–29, полученный в результате трансляции Clang [7] версии 6.0.0, представлен в Algorithm 1, а код, полученный после трансляции GGC [8] версии 7.5.0, — в Algorithm 2.

В коде, транслированном GCC, машинных инструкций меньше, чем в коде, транслированном Clang. Однако, как и в случае с примером умножения матриц из предшествующей работы [3], измерения времени на плате BFK 3.1 с процессором Baikal-T с помощью утилиты *time* [9] показали, что код, порождённый компилятором Clang, работает быстрее

[1] В предыдущей статье было указано, что процессор имеет разные названия: Baikal-T1, Baikal-T и BE-T1000. В данной статье используется то же название, что и в предыдущей публикации.

Algorithm 1: Внутренний цикл после компиляции Clang

```
$BB0_12:
  addu $1, $9, $24      # вычисление адреса a[i][k]
  lw $25, 0($14)        # вырезка b[k][j]
  addiu $14, $14, 800   # инкремент индуцированной переменной
  addiu $24, $24, 4     # инкремент индуцированной переменной
  lw $1, 0($1)          # вырезка a[i][k]
  mul $1, $25, $1       # умножение
  bne $24, $6, $BB0_12  # переход
  subu $15, $15, $1     # вычитание
```

Algorithm 2: Внутренний цикл после компиляции GCC

```
$L8:
  addiu $2,$2,800  # инкремент индуцированной переменной
  lw $5,0($3)      # вырезка a[i][k]
  lw $4,-800($2)   # вырезка b[k][j]
  addiu $3,$3,4    # инкремент индуцированной переменной
  bne $6,$2,$L8    # переход
  msub $5,$4       # умножение и вычитание
```

кода, порождённого компилятором GCC. Причём время работы обеих программ примерно равно времени работы соответствующих программ умножения матриц. Результат измерений представлен в таблице 1.

Таблица 1. Сравнение Clang и GCC

Транслятор	Время, с
Clang	61,53
GCC	72,16

Оказалось, что разница во времени обусловлена использованием команды *msub*. Это и есть команда совмещённого умножения-вычитания, которая используется для сокращения числа инструкций. Влияние *msub* на время работы программы можно продемонстрировать следующим экспериментом. В коде, порождённом Clang, заменим команды умножения и вычитания на *msub*, а в коде, порождённом GCC, заменим *msub* на команды умножения и вычитания и замерим время работы программы. Результаты представлены в таблице 2 и таблице 3 соответственно для Clang и GCC. Код переписанных программ представлен в Algorithm 3 и Algorithm 4 соотвественно для Clang и GCC.

Таблица 2. Время исполнения после трансляции Clang с и без использования *msub*

	Время, с
С	73,43
Без	61,53

Таблица 3. Время исполнения после трансляции GCC с и без использования *msub*

	Время, с
С	72,16
Без	54,52

Algorithm 3: Код после трансляции Clang с использованием *msub*

```
$BB0_12:
  addu $1, $9, $24      # вычисление адреса a[i][k]
  lw $25, 0($14)        # вырезка b[k][j]
  addiu $14, $14, 800   # инкремент индуцированной переменной
  addiu $24, $24, 4     # инкремент индуцированной переменной
  lw $1, 0($1)          # вырезка a[i][k]
  bne $24, $6, $BB0_12  # переход
  msub $1, $25          # умножение и вычитание
```

Algorithm 4: Внутренний цикл после компиляции GCC

```
$L8:
  addiu $2,$2,800  # инкремент индуцированной переменной
  lw $5,0($3)      # вырезка a[i][k]
  lw $4,-800($2)   # вырезка b[k][j]
  addiu $3,$3,4    # инкремент индуцированной переменной
  mul $4, $4, $5   # умножение
  bne $6,$2,$L8    # переход
  sub $16, $16, $4 # вычитание
```

Чтобы убедиться, что программы работают медленнее из-за самой команды *msub*, а не из-за какой-либо последовательности команд, включающей *msub*, из Algorithm 1, Algorithm 2, Algorithm 3 и Algorithm 4 последовательно убирались команды и делались замеры.

Для первого замера были взяты программы в неизменённом варианте. Во втором замере из всех программ была убрана вторая команда. В третьем замере из Algorithm 1 и Algorithm 3 были убраны первая и пятая команды, а из Algorithm 2 и Algorithm 4 третья команда. В последнем, четвертом замере из Algorithm 1 и Algorithm 3 была убрана третья команда, а из Algorithm 2 и Algorithm 4 четвёртая команда.

Результаты измерений представлены на графике на рисунке 1. Отсюда можно сделать вывод, что дело именно в самой команде *msub*, а не в какой-либо последовательности команд, частью которой является *msub*.

Все описанные выше эксперименты аналогичны проведённым экспериментам для команды *madd* в предыдущей публикации по теме [3]. И результаты для этих двух команд получились схожие. Однако дальнейшие эксперименты с замером тактов процессора выявят некоторые различия между *madd* и *msub*.

4. ИЗМЕРЕНИЯ ТАКТОВ ПРОЦЕССОРА

Для дальнейшего изучения особенностей инструкции *msub* необходим инструмент для измерения тактов, что даст более подробную информацию о работе команды. Для этого в архитектуре MIPS существует команда rdhwr [2], которая кладёт в указанный регистр

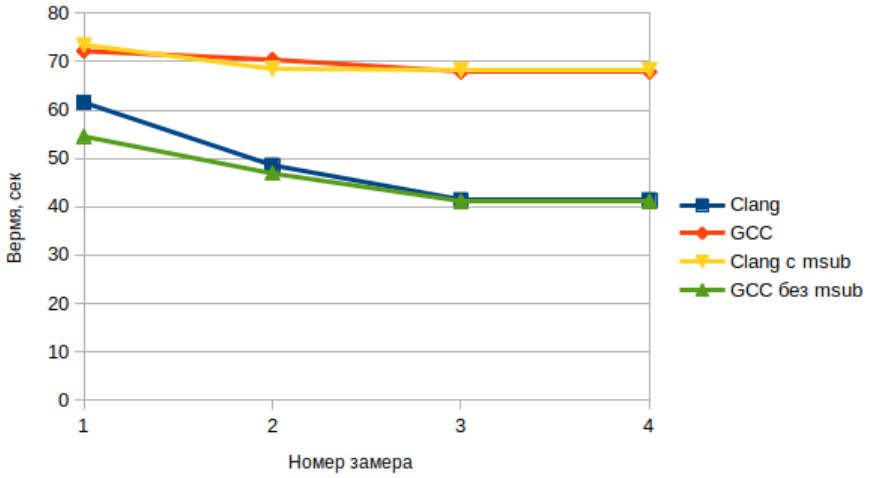

Рис. 1. Время работы программ

значение счётчика тактов. В Algorithm 5 представлен фрагмент программы для измерения количества тактов, необходимых для выполнения некоторого набора команд. Полная версия программы для измерения каждого набора команд из этой статьи выложена в отдельном репозитории [6]. Программа в цикле 100 раз исполняет команды с измерением счётчика тактов, а далее программа на Python подсчитывает среднее значение тактов, необходимое для выполнения команд.

Algorithm 5: Подсчёт числа тактов

```
$L3:
  rdhwr $2, $2   # сохранение счётчика в начале
  # команды, для которых необходимо измерить такты
  ...
  rdhwr $3, $2   # сохранение счётчика второй раз
  subu $2,$3,$2  # вычитание
  # вывод результат в консоль
  ...
  # переход на метку $L3 в цикле
  ...
```

4.1. Эффективность изолированной команды *msub*

Прежде всего нужно изучить поведение команды *msub* в изолированном состоянии, то есть вне связок с другими командами. Для этого было произведено измерение количества тактов для исполнения от 1 до 30 инструкций *msub* подряд. Для сравнения был произведён такой же эксперимент для пары команд *mul* + *sub*. Результат измерений представлен на рисунке 2.

Из графика видно, что до 24 инструкций (пар инструкций) для выполнения *msub* и *mul* + *sub* требуется примерно одинаковое количество тактов. А далее *msub* начинает требовать всё больше тактов по сравнению с *mul* + *sub*. Стоит обратить внимание на резкий скачок в количестве тактов при увеличении количества блоков инструкций с двух до трёх. После него количество тактов увеличивается постепенно.

Аналогичный эксперимент для *madd* из предыдущей работы [3] показал иные результаты. Инструкция *madd* в большинстве случаев работала быстрее связки *mul* + *addu*, и только при количестве блоков больше 18 стала работать заметно хуже. Увеличение количества тактов происходило без резких скачков на всём промежутке измерения.

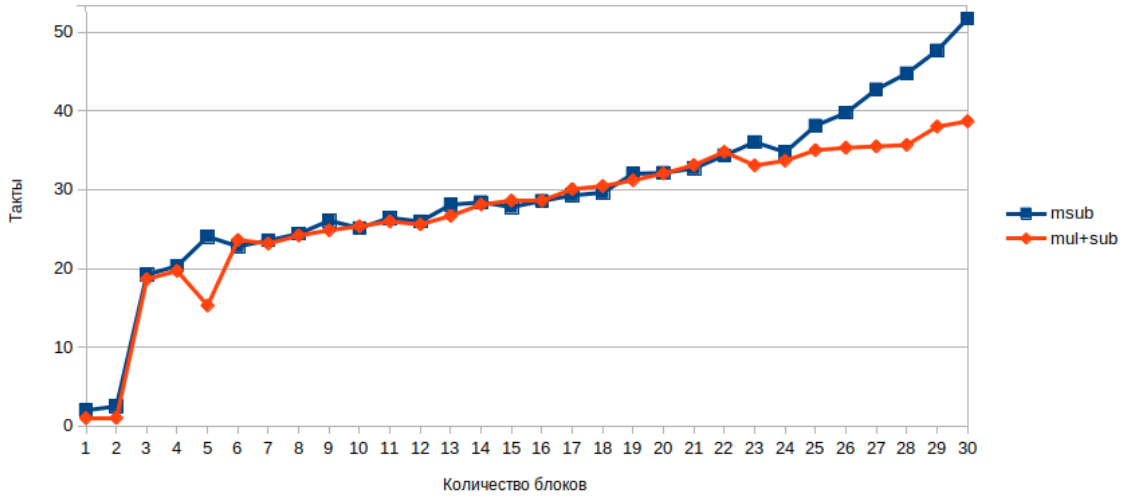

Рис. 2. Изолированные команды: количество тактов

4.2. Эффективность команды *msub* в последовательных переходах

В данном подразделе изучим особенности работы команды *msub* в связке с командой условного перехода. Обратимся к коду, представленному в Algorithm 6 и Algorithm 7. Вырезка ассемблерного кода в Algorithm 6 является набором блоков кода, состоящих из команды условного перехода и *msub* в слоте задержки. Аналогично устроен код и в Algorithm 7, только вместо *msub* используется *mul* + *sub*, а в слоте задержки стоит *sub*.

Algorithm 6: Последовательные переходы: *msub* + *bne*

```
bne $12, $13, $A1
msub $10, $11
$A1:
bne $12, $13, $A2
msub $10, $11
$A2:
...
```

Algorithm 7: Последовательные переходы: *mul* + *bne* + *sub*

```
mul $11, $10, $11
bne $12, $13, $A1
sub $14, $11, $14
$A1:
mul $11, $10, $11
bne $12, $13, $A2
sub $14, $11, $14
$A2:
...
```

Приведённые выше вырезки кода можно вставить в программу из Algorithm 5 для измерения количества тактов. Это даст представление о работе инструкции *msub* в связке с условным переходом. Именно эта связка присутствует в циклах, содержащих совмещённое умножение-вычитание. Полные тексты программ, для которых были произведены замеры, выложены в отдельном репозитории [6]. Результат измерений представлен на рисунке 3.

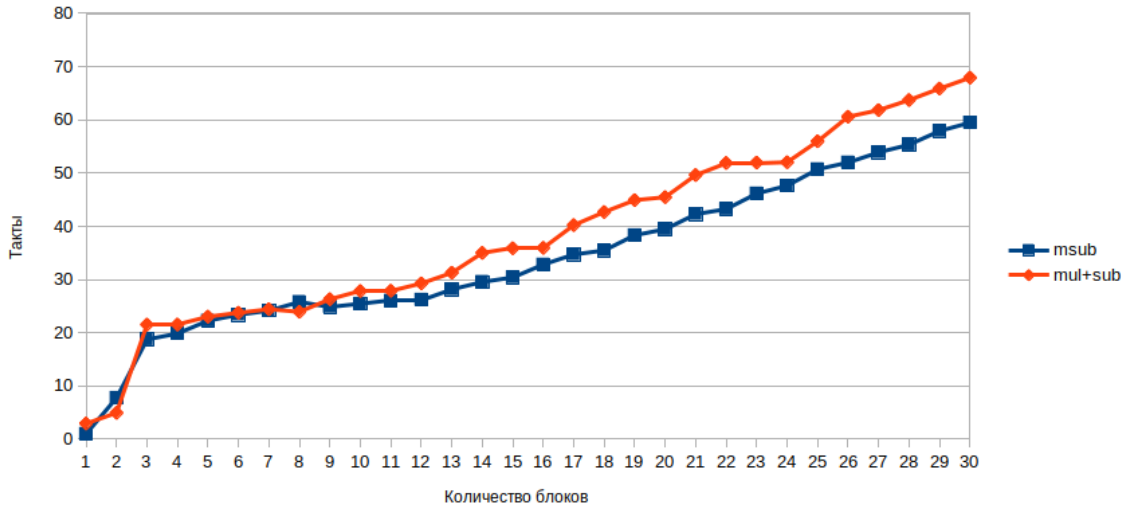

Рис. 3. Последовательные переходы: количество тактов

График показывает, что при небольшом количестве блоков *msub* и *mul* + *sub* требуют примерно одинаковое количество тактов. Однако по мере увеличения последовательных блоков с командой условного перехода *msub* начинает выигрывать в количестве тактов у связки *mul* + *sub*. Довольно интересный результат на фоне того, что изолированная команда *msub* работает хуже связки *mul* + *sub*, а также цикл с *msub*, где присутствует связка с командой условного перехода, работает гораздо хуже аналогичного цикла без *msub*.

Так же, как и на графике на рисунке 2, присутствует резкий скачок в количестве тактов при увеличении количества блоков с двух до трёх.

Проведём краткое сравнение с аналогичным экспериментом для инструкции *madd* из предыдущей работы. В отличие от *msub* она и при небольших количествах блоков работает лучше связки *mul* + *addu*. О работе *madd* при увеличении количества блоков сложно делать выводы, так как количество измерений в предыдущей работе для этого недостаточно. Как и в предыдущем опыте, для *madd* никаких скачков при переходе от двух блоков к трём не наблюдается.

4.3. Эффективность команды *msub* в цикле

Далее изучим поведение команды *msub* в цикле, ведь именно в этой ситуации наблюдается большой проигрыш во времени для примера из Приложения 1. Для этого будем изучать количество тактов, наобходимых для циклов из Algorithm 8 и Algorithm 9. Измерения были произведены для количества итераций цикла от 1 до 30. Результат представлен на рисунке 4. Дополнительно были сделаны замеры каждые 10 итераций с количеством итераций от 10 до 300. Результат представлен на рисунке 5.

Algorithm 8: Цикл *msub* + *bne*

```
$A1:
  addiu $12, $12, 1
  bne $12, $13, $A1
  msub $10, $11
```

Algorithm 9: Цикл *mul* + *bne* + *sub*

```
$A1:
  addiu $12, $12, 1
  mul $11, $10, $11
  bne $12, $13, $A1
  sub $14, $11, $14
```

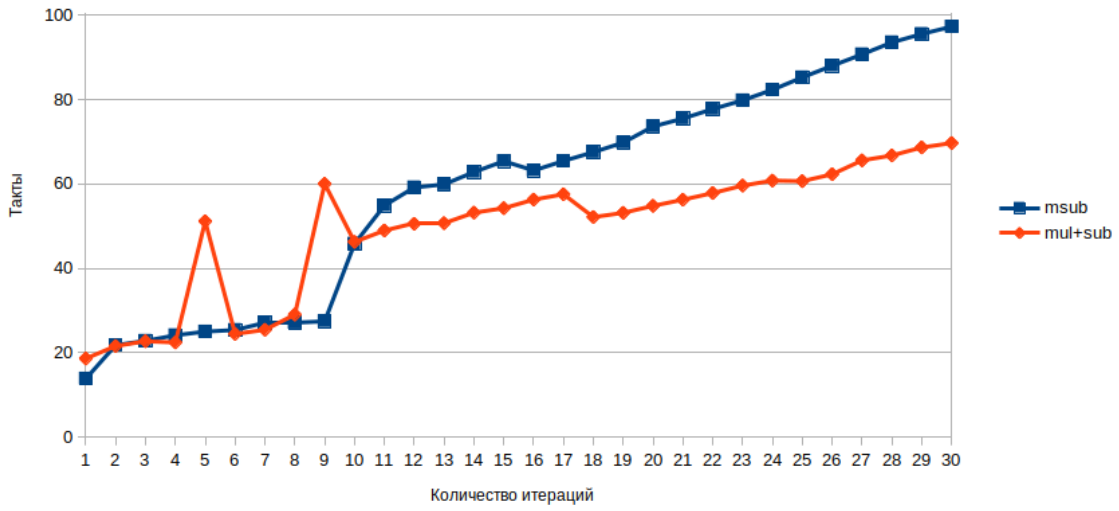

Рис. 4. Цикл: небольшое количество итераций

На небольшом количестве итераций (до 10) *msub* и *mul* + *sub* примерно одинаковы с точки зрения эффективности (не считая странных скачков, о которых речь пойдет далее). После десяти итераций *msub* становится всё менее эффективной в сравнении с *mul* + *sub*. Стоит отметить, что, в отличие от предыдущих экспериментов, здесь не происходит резкого скачка в количестве тактов при увеличении количества итераций с двух до трёх.

Особое внимание обращают на себя странные скачки количества тактов при пяти и девяти итерациях для программы с *mul* + *sub*. Подобное странное увеличение количества тактов можно наблюдать на графике на рисунке 5 для 20 итераций, хотя на рисунке 4 такого резкого изменения не зафиксировано. Подобных аномалий для *mul* + *addu* из результатов измерений в предыдущей работе не было обнаружено. Возникает предположение, что цикл с *mul* + *sub* может работать нестабильно относительно количества затрачиваемых тактов, хотя данное предположение требует отдельного исследования.

На рисунке 5 продемонстрировано, что количество тактов, необходимых для цикла и с *msub*, и с *mul* + *sub*, увеличивается линейно. Причём количество тактов в случае использования *msub* растёт быстрее.

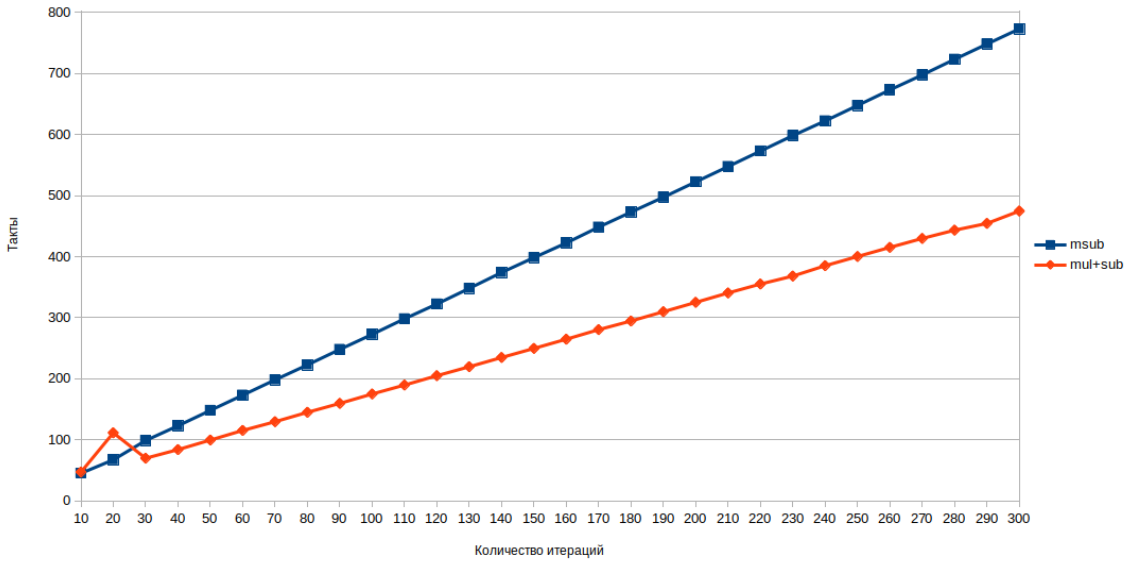

Рис. 5. Цикл: большое количество итераций

4.4. Эффективность команды *msub* в обратных переходах

В циклах управление передаётся не на следующую команду, а на одну из предыдущих. Необходимо проверить, влияет ли это обстоятельство на работу команды *msub*. Для этого были сделаны измерения для развёрнутого цикла, где переход на другой блок происходит не в последовательном, как в разделе 4.2, а в обратном порядке. Из-за размера программы для измерения нецелесообразно размещать её в статье, как в предыдущих случаях. Код можно посмотреть в отдельном репозитории [6]. Результат измерений представлен на рисунке 6.

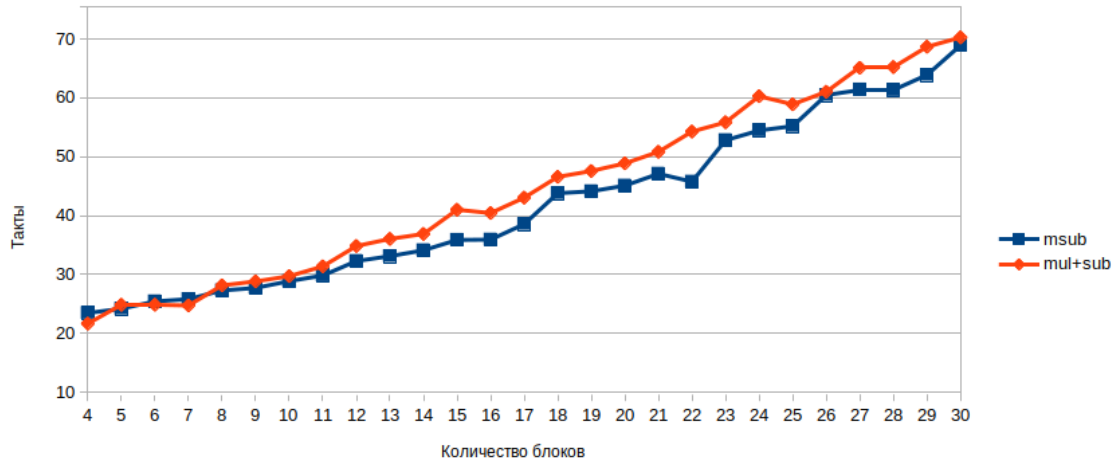

Рис. 6. Обратные переходы: количество тактов

В данном случае результаты схожи с результатами из раздела 4.2. Также при небольших количествах блоков разница в количестве тактов между программами с *msub* и *mul* + *sub* невелика, но с увеличением блоков программа, использующая *mul* + *sub*, начинает ра-

ботать хуже своего аналога с *msub*. Из этого можно сделать вывод, что порядок переходов в программе не влияет на работу команды *msub*.

5. ЗАКЛЮЧЕНИЕ

В статье были исследованы особенности работы инструкции совмещённого умножения-вычитания и сделано сравнение с инструкцией совмещённого умножения-сложения. Был рассмотрен пример для изучения времени работы программ с использованием операции совмещённого умножения-вычитания, а также изучены различные ситуации использования рассматриваемой команды.

В отличие от выводов статьи про совмещённое умножение-сложение [3], результаты в данной публикации не столь однозначные. Точно можно сказать, что использование команды *msub* не выгодно в циклах с точки зрения времени работы программы. В остальных случаях, рассматриваемых в работе (изолированное использование команды, использование команды в последовательных и обратных переходах) в некоторых случаях незначительно выигрывало использование *msub*, а в некоторых связка команд *mul* + *sub*. Поэтому насчёт этих случаев нельзя сделать однозначный вывод об эффективности того или иного подхода.

Приложение 1

```
void main()
{
  int a[200][200], b[200][200], c[200][200], i, j, k, v;
  for (i = 0; i < 200; i++)
  {
    for (j = 0; j < 200; j++)
    {
      a[i][j] = i * j;
    }
  }

  for (i = 0; i < 200; i++)
  {
    for (j = 0; j < 200; j++)
    {
      b[i][j] = i + j;
    }
  }

  for (v = 0; v < 2000; v++)
  {
    for(i = 0; i < 200; i++)
  {
      for(j = 0; j < 200; j++)
      {
        c[i][j] = 0;
          for(k = 0; k < 200; k++)
          c[i][j] —= a[i][k] * b[k][j];
      }
    }
  }
  printf("%i\n", c[0][0]);
}
```

Список литературы

1. IEEE Standard for Floating-Point Arithmetic // IEEE Std 754-2019 (Revision of IEEE 754-2008). 22.06.2019. 2019. P. 1–84, doi: 10.1109/IEEESTD.2019.8766229
2. MIPS Architecture for Programmers. Volume II-A: The MIPS32. Instruction Set Manual. Document Number: MD00086, Revision 6.06. 2016. https://s3-eu-west-1.amazonaws.com/downloads-mips/documents/MD00086-2B-MIPS32BIS-AFP-6.06.pdf (дата обращения: 31.10.2022).
3. *Архипов И. С.* Исследование операции совмещённого умножения-сложения на процессоре Baikal-T // Компьютерные инструменты в образовании. 2022. № 1. С. 46—56. doi: 10.32603/2071-2340-2022-1-46-56
4. *Ильина С. А.* Рынок полупроводников: глобальная цепочка создания стоимости и динамика в условиях кризиса // Вестник Института экономики Российской академии наук. 2022. № 3. С. 112–135.
5. *Смирнов В. М., Коновалова С. С.* Необходимость перехода ОВД на российское программное обеспечение // Международный журнал гуманитарных и естественных наук, 2022. Т. 4, № 2. С. 74–76. doi: 10.24412/2500-1000-2022-4-2-74-76
6. Репозиторий с дополнительными материалами к статье. URL: https://github.com/IvanArkhipov1999/Multiply-accumulate-operations-research.
7. Официальный сайт LLVM. URL: https://llvm.org/ (дата обращения: 31.10.2022).
8. Официальный сайт GCC. URL: https://gcc.gnu.org/ (дата обращения: 31.10.2022).
9. time(1) Linux manual page. URL: https://man7.org/linux/man-pages/man1/time.1.html (дата обращения: 31.10.2022).

Поступила в редакцию 28.09.2022, окончательный вариант — 31.10.2022.

Архипов Иван Сергеевич, студент образовательной программы магистратуры «Математическое обеспечение и администрирование информационных систем» СПБГУ 2 года обучения, ✉ **arkhipov.iv99@mail.ru**

Computer tools in education, 2022
№ 3: 82–93
http://cte.eltech.ru
doi:10.32603/2071-2340-2022-3-82-93

Investigation of the Multiply-sub Operation on the Baikal-T Processor

Arkhipov I. S.[1], Graduate, ✉ arkhipov.iv99@mail.ru

[1]Saint Petersburg State University, 28, Universitetskiy pr., 198504, Saint Petersburg, Russia

Abstract

This article is a continuation of a cycle of research devoted to the study of combined operations on the Baikal-T processor. It discusses various features of the combined multiplication-subtraction command. Various examples of using the command are given, calculations are made and conclusions are formulated. It also describes situations in which the use of the combined multiplication-subtraction command is justified, and situations in which its use is not profitable relative to the operating time of the program.

Keywords: *MIPS, Baikal processor, multiply-subtraction, optimization, assembler.*

Citation: I. S. Arkhipov, "Investigation of the Multiply-sub Operation on the Baikal-T Processor," *Computer tools in education*, no. 3, pp. 82–93, 2022 (in Russian); doi: 10.32603/2071-2340-2022-3-82-93

References

1. IEEE, "IEEE Standard for Floating-Point Arithmetic," in *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pp. 1–84, 2019; doi: 10.1109/IEEESTD.2019.8766229
2. Wave Computing Inc., "MIPS Architecture for Programmers. Volume II-A: The MIPS32. Instruction Set Manual. Document Number: MD00086, Revision 6.06," in *www.wavecomp.ai*, 2016. [Online]. Available: https://s3-eu-west-1.amazonaws.com/downloads-mips/documents/MD00086-2B-MIPS32BIS-AFP-6.06.pdf
3. I. S. Arkhipov, "Investigation of the Multiply-add Operation on the BaikalT Processor," *Computer tools in education*, no. 1, pp. 46–56, 2022 (in Russian); doi: 10.32603/2071-2340-2022-1-46-56
4. S. A. Ilyina, "Semiconductor market: Global value chain and dynamics in a crisis," *The Bulletin of the Institute of Economics of the Russian Academy of Sciences*, no. 3, pp. 112–135, 2022 (in Russian).
5. V. M. Smirnov and S. S. Konovalova, "The need to switch the ATS to russian software," *Mezhdunarodnyi zhurnal gumanitarnykh i estestvennykh nauk*, vol. 4, no. 2, pp. 74–76, 2022 (in Russian); doi: 10.24412/2500-1000-2022-4-2-74-76
6. I. S. Arkhipov, "Multiply-accumulate-operations-research," in *GitHUB*, 13 Jul. 2022. [Online]. Available: https://github.com/IvanArkhipov1999/Multiply-accumulate-operations-research
7. LLVM-admin team, "The LLVM Compiler Infrastructure LLVM," in *llvm.org*. [Online]. Available: https://llvm.org/
8. GCC team, "GCC, the GNU Compiler Collection," in *gcc.gnu.org*. [Online]. Available: https://gcc.gnu.org/
9. Jambit GmbH, "time(1) Linux manual page," in *man7.org*. [Online]. Available: https://man7.org/linux/man-pages/man1/time.1.html

Received 28-09-2022, the final version — 31-10-2022.

Ivan Arkhipov, Graduate of the educational program "Mathematical support and administration of information systems", 2 year of study, ✉ arkhipov.iv99@mail.ru