УЧЕБНО-МЕТОДИЧЕСКОЕ ПОСОБИЕ

«Разработка программного обеспечения для отечественных архитектур микропроцессоров»

Москва 2025

УДК 004.415.2.041(075.8)
ББК 32.973.4

Учебно-методическое пособие
«Разработка программного обеспечения
для отечественных архитектур микропроцессоров»

Пособие применяется при обучении по следующим дисциплинам:
«Инфраструктура создания ПО для отечественных архитектур микропроцессоров»
«Сборка программного обеспечения в отечественных репозиториях»
«Разработка программного обеспечения для отечественных архитектур микропроцессоров»

Автор:
В. С. Малиночкин.

Автор выражает благодарность за поддержку разработки настоящего учебно-методического пособия Автономной некоммерческой организации «Координационный центр национального домена сети Интернет».

Рецензент: В. А. Кутуков, к.т.н.

ISBN 978-5-6054409-4-9

Оглавление

Введение 4

1 Структура команд 6

2 Команды микропроцессора 10
- 2.1 Операции сложения, вычитания и умножения 10
- 2.2 Операции деления и вычисления остатка 14
- 2.3 Логические операции 18
- 2.4 Сдвиги 19
- 2.5 Операции сравнения 20
- 2.6 Операции подготовки передачи управления и передачи управления 25
- 2.7 Операции считывания из памяти 28
- 2.8 Операции записи в память 34
- 2.9 Соответствие операций и АЛК 48

Заключение 50

Введение

Учебно-методическое пособие «Разработка программного обеспечения для отечественных архитектур микропроцессоров» предназначено для очного и дистанционного обучения студентов по программам бакалавриата, специалитета, магистратуры и дополнительного профессионального образования базовой кафедры общественно-государственного объединения «Ассоциация документальной электросвязи» (АДЭ) «Технологии электронного обмена данными» (ТЭОД) в Московском техническом университете связи и информатики (МТУСИ).

Пособие состоит из введения, заключения и двух глав:

- Структура команд.
- Команды микропроцессора.

Во второй главе содержатся практические задания по следующим направлениям:

- Операции сложения, вычитания и умножения.
- Операции деления и вычисления остатка.
- Операции сравнения.
- Операции считывания из памяти.
- Операции записи в память.

В пособии представлено описание некоторых из наиболее часто применяемых команд микропроцессора «Эльбрус», приведены примеры их использования, а также задания для самостоятельного выполнения по каждому из рассматриваемых наборов команд.

Задания для самостоятельного выполнения предполагают написание программного кода на языке С, а также функции на ассемблере, реализующей аналогичные вычисления, что и код на языке С (за исключением вызова функции и вывода результата). Предполагается, что читатель уже

знаком с языком С, поэтому комментарии к выполнению кода на этом языке не приводятся.

Перед началом каждого практического занятия преподаватель выдаёт заготовку программного кода на языке С и ассемблере, в которых написана необходимая вспомогательная структура для вызова из кода на языке С функций, реализованных на ассемблере.

Глава 1
Структура команд

В данной главе приведена общая структура команд, а также подробно разобран пример заготовки программного кода на языке C и ассемблере, который выдаёт преподаватель в рамках практических занятий.

Рассмотрим общую структуру команд:

```
ADD(s/d) (.s) ops (? predX)
```

В указанной команде:

- `ADD` — название операции.
- `(s/d)` — формат операндов и результата:
  - `s` — операнды и/или результат являются 32-битным значением (одинарное слово);
  - `d` — операнды и/или результат являются 64-битным значением (двойное слово);
  - `x` — операнды и/или результат являются значением расширенного формата;
  - `b` — результат является предикатом.
- `(.s)` — спекулятивное исполнение (необязательный аргумент).
- `ops` — операнды операции.
- `(? predX)` — исполнение операции под предикатом X (необязательный аргумент).

Рассмотрим на примере команды сложения 32-битных операндов — `ADDs`:

```
ADDs,0 %r1, %r2, %r3
```

Операция сложения выполняется над 32-битными операндами, расположенными в регистрах `%r1` и `%r2`, в арифметико-логическом канале с номером 0. Результат операции помещается в регистр `%r3`.

Разбор заготовки кода

Ниже приведён полный пример кода, который предоставляется вместе с первым заданием. Заготовки для остальных заданий имеют аналогичную структуру.

В первом задании необходимо написать программный код, реализующий математическое выражение из четырёх переменных.

Код на C:

```
#include <stdio.h>

int calc_asm(int, int, int, int);

int main() {
    int a = ;
    int b = ;
    int c = ;
    int d = ;

    int res = ;
    printf("%d\n", res);

    int res1 = calc_asm(a, b, c, d);
    printf("%d\n", res1);

    return 0;
}
```

Как видно из кода — объявлен прототип функции `calc_asm`, но тела функции в коде нет. Тело функции будет реализовано в коде на ассемблере.

Код на ассемблере:

```
  .ignore ld_st_style
  .ignore strict_delay
  .text
```

```
  .global calc_asm
  .type   calc_asm, #function
  .align  8

calc_asm:
  {
    setwd wsz = 0x6, nfx = 0x1, dbl = 0x0
  }

  /* v Put your asm code here v */

  /* ^     End of asm code    ^ */

  {
    nop 5
    RETURN %ctpr3
  }
  {
    CT    %ctpr3
  }
```

Рассмотрим программный код на ассемблере до начала функции:

1. `ignore ld_st_style` — отключает предупреждения о стиле применения операций чтения и записи в память;
2. `ignore strict_delay` — отключает строгие проверки задержек конвейера;
3. `.text` — начало секции кода (исполняемых инструкций);
4. `.global calc_asm` — объявляет метку `calc_asm` глобальной, делая её видимым из других файлов;
5. `.type calc_asm, #function` — указывает, что `calc_asm` является функцией;

6. `.align 8` — выравнивание инструкций по границе 8 байт (64 бита);

7. `calc_asm:` — метка `calc_asm`, после которой начинается тело её функции.

После объявленной метки следует два комментария, между которыми требуется написать программный код. На ассемблере Эльбруса комментариями являются все символы, находящиеся между началом комментария `/*` и окончанием комментария `*/`. Комментарий может быть многострочным.

В конце подготавливается возврат из процедуры `RETURN` и непосредственно возврат `CT`. Подробнее об этих командах написано в главе 2.

Программный код на ассемблере разбит на широкие команды. Каждая широкая команда обрамляется в фигурные скобки `{ }`. Внутри фигурных скобок есть определённые требования по написанию кода:

- Нельзя использовать одно и то же исполнительное устройство одного арифметико-логического канала дважды.

- Устройство передачи управления можно использовать только один раз.

Глава 2
Команды микропроцессора

2.1 Операции сложения, вычитания и умножения

Синтаксис операций

- `ADDs/d <оп1> <оп2> <результат>` — операция сложения двух 32/64-битных целых операндов. Результат занимает 32 бита. Операция доступна во всех АЛК.
- `SUBs/d <оп1> <оп2> <результат>` — операция вычитания двух 32/64-битных целых операндов. Результат занимает 32 бита. Операция доступна во всех АЛК.
- `MULs/d <оп1> <оп2> <результат>` — операция умножения двух 32/64-битных целых операндов. Результат занимает 32 бита. Операция доступна во всех АЛК, кроме 2 и 5.
- `uMULx <оп1> <оп2> <результат>` — операция беззнакового умножения двух 32-битных целых операндов. Результат занимает 64 бита. Операция доступна во всех АЛК, кроме 2 и 5.
- `sMULx <оп1> <оп2> <результат>` — операция умножения с учётом знака двух 32-битных целых операндов. Результат занимает 64 бита. Операция доступна во всех АЛК, кроме 2 и 5.



Операция умножения может увеличить размер числа в битах примерно вдвое, потому что каждое умножение на 2 добавляет один бит к представлению числа, и если попытаться умножить число, которое уже близко к максимальному для своего размера, на число, которое само требует примерно столько же бит, результат может удвоить требуемое количество бит. Поэтому результат умножения двух операндов по 32 бита может занимать 64 бита.

Рассмотрим несколько примеров выполнения операций:

1. `ADDs,0 %r0, %r0, %r1` — операция сложения с двумя 32-битными операндами `%r0`, результат записывается в регистр `%r1`. Для простоты понимания эту операцию можно расписать как: `%r1 = %r0 + %r0`.

2. `SUBs,1 %r4, %r2, %r0` — операция вычитания с двумя 32-битными операндами: `%r4` и `%r2`, результат записывается в регистр `%r0`. Эту операцию можно расписать как: `%r0 = %r4 - %r2`.

3. `uMULx,2 %r0, %r1, %r2` — операция беззнакового умножения двух 32-битных операндов: `%r0` и `%r1`, 64-битный результат записывается в регистр `%r2`. Эту операцию можно расписать как: `%r2 = %r0 * %r1`.

4. `ADDs,0 %r0, 5, %r1` — операция сложения 32-битного операнда `%r0` и числа 5, результат записывается в регистр `%r1`. Эту операцию можно расписать как: `%r1 = %r0 + 5`.

5. `ADDs,1 0x3, %r0` — операция перемещения 32-битного числа 3 в регистр `%r0`. Эту операцию можно расписать как: `%r0 = 3`.

Пример

Реализуем программный код на языке C, который высчитывает выражение `a * c + b - d`. В качестве значений a, b, c и d выберем 2, 5, 4, 3, соответственно.

Реализация на C:

```
int main()  {
  int a = 2;
  int b = 5;
  int c = 4;
  int d = 3;
  int res = a * c + b - d;
  return 0;
```

```
}
```

Реализуем аналогичный код на ассемблере Эльбруса:

```
{ MULs,0 %r0, %r2, %r4 }
{ ADDs,0 %r4, %r1, %r5 }
{ SUBs,0 %r5, %r3, %r0 }
```



Аргументы, переданные в функцию, реализованную на ассемблерном коде, при её вызове из языка C, попадают в регистры, начиная с `%r0`. Таким образом в примере выше `a` находилось бы в `%r0`, `b` в `%r1`, `c` в `%r2`, `d` в `%r3`. А сам вызов никак не отличался от вызова любой другой функции.



При возврате из функции, реализованной на ассемблере, всегда возвращается значение, которое находится в регистре `%r0`.

Это наивная реализация, в которой каждая операция реализуется в своей широкой команде. Попробуем объединить операции, чтобы они выполнялись параллельно. Для этого обратимся к исходному выражению `a*c+b-d`, из которого видно, что операции умножения `a * c` и вычитания `b - d` можно выполнить параллельно. Операцию сложения параллельно выполнить не получится, так как её операндами являются результаты умножения и вычитания.

Получим следующий код:

```
{
  MULs,0 %r0, %r2, %r4
  SUBs,1 %r1, %r3, %r5
}
{ ADDs,0 %r4, %r5, %r0 }
```

Задания

Получите у преподавателя заготовки для написания программного кода на языке C и ассемблере Эльбруса и реализуйте выражение, указанное в задании. В результате код, написанный на C и код, написанный

на ассемблере, должны выводить в консоль один и тот же результат. Значения переменных выбираются случайным образом.

Вариант задания определяется по порядковому номеру в списке группы.

Список заданий:

1. `a + b * c - d`
2. `(a + b) * (c - d)`
3. `a + c * (b - d)`
4. `a + b - c * d`
5. `(a + b) * c - d`
6. `a * b + c - d`
7. `a + b * (c - d)`
8. `(a - b) * (c + d)`
9. `a * (b + c * d)`
10. `a + (b - c) * d`
11. `a - b + c * d`
12. `(a + b) * (c + d)`
13. `a * b * c - d`
14. `a - (b + c * d)`
15. `(a * b) - (c * d)`
16. `a + b - c - d`
17. `a * (b - c) + d`
18. `(a + b - c) * d`

19. `a + (b + c + d)`
20. `a - (b - c + d)`
21. `a * (b * c - d)`
22. `(a * d + b) - c`
23. `a * (b + c) - d`
24. `a + b * c + d`

2.2 Операции деления и вычисления остатка

Синтаксис операций

- `uDIVs/d <оп1> <оп2> <результат>` — операция беззнакового целочисленного деления 32/64-битного операнда на 32/64-битный операнд. Результат занимает 32/64 бита. Операция доступна только в АЛК 5.
- `sDIVs/d <оп1> <оп2> <результат>` — операция целочисленного деления с учётом знака 32/64-битного операнда на 32/64-битный операнд. Результат занимает 32/64 бита. Операция доступна только в АЛК 5.
- `uDIVx <оп1> <оп2> <результат>` — операция беззнакового целочисленного деления 64-битного операнда на 32-битный операнд. Результат занимает 32 бита. Операция доступна только в АЛК 5.
- `sDIVx <оп1> <оп2> <результат>` — операция целочисленного деления с учётом знака 64-битного операнда на 32-битный операнд. Результат занимает 32 бита. Операция доступна только в АЛК 5.
- `uMODx <оп1> <оп2> <результат>` — операция получения остатка от деления без учёта знака 64-битного операнда на 32-битный операнд. Результат занимает 32 бита. Операция доступна только в АЛК 5.

- `sMODx <оп1> <оп2> <результат>` — операция получения остатка от деления с учётом знака 64-битного операнда на 32-битный операнд. Результат занимает 32 бита. Операция доступна только в АЛК 5.



При попытке деления на ноль или получения остатка от деления на ноль возникнет ошибка:

.

Рассмотрим несколько примеров выполнения операций:

1. `uMODx,5 %r3, 0x2, %r4` — операция получения остатка от деления без учёта знака 64-битного операнда `%r3` и числа 2, 32-битный результат записывается в регистр `%r4`. Для простоты понимания эту операцию можно расписать как: `%r4 = %r3 mod 2`.

2. `sDIVs,5 %r0, 10, %r0` — операция деления с учётом знака 32-битного операнда `%r0` на число 10, 32-битный результат записывается в регистр `%r0`. Для простоты понимания эту операцию можно расписать как: `%r0 = %r0 / 10`.

Пример

Реализуем программный код на языке C, который высчитывает выражение `a * b + c / d - a mod d`. В качестве значений a, b, c и d выберем 5, 10, 30, 3, соответственно. `mod` — операция получения остатка от деления (в коде на C это оператор `%`). В этот раз объявление переменных опустим.

Реализация на C:

```
int main()  {
  int res = a * b + c / d - a % d;
  return 0;
}
```

Реализация на ассемблере:

```
{
  sMULx,0 %r0, %r1, %r4
  sDIVx,5 %r2, %r3, %r5
}
{
  ADDs,0  %r4, %r5, %r6
  sMODx,5 %r0, %r3, %r7
}
{
  SUBs,0 %r6, %r7, %r0
}
```

Получения остатка от деления можно выполнить в несколько команд, последовательно реализовав операции в соответствии с математическим алгоритмом. Остаток от деления `a` на `b` можно вычислить за три шага:

1. Целочисленное деление `a` на `b`: `c = a / b`;
2. Целочисленное умножение `c` на `b`: `d = c * b`;
3. Вычитание из `a` результата умножения `d`: `e = a - d`.

Задания

Получите у преподавателя заготовки для написания программного кода на языке C и ассемблере Эльбруса и реализуйте выражение, указанное в задании. В результате код, написанный на C и код, написанный на ассемблере, должны выводить в консоль один и тот же результат. Значения переменных выбираются случайным образом.

Вариант задания определяется по порядковому номеру в списке группы. Задание под номером `0` является общим для всех.

Список заданий:

0. Реализуйте вычисление остатка от деления `a` на `b` в соответствии с математическим алгоритмом (без использования встроенной операции `mod`).

1. a / b + c * d - c mod d
2. a * b / c + d - a mod b
3. b + c / d - a * b mod c
4. c / d * a + b - c mod d
5. d - a * b / c + d mod a
6. a mod b + c * d - a / b
7. b - c mod d * a / b + c
8. c * d mod a + b - c / d
9. d / a - b mod c * d + a
10. a + b * c mod d - a / b
11. b / c + d * a mod b - c
12. c * d / a + b mod c - d
13. d mod a - b + c * d / a
14. a - b mod c * d / a + b
15. b * c - d mod a + b / c
16. c mod d + a * b - c / d
17. d / a * b mod c - d + a
18. a * c / d - b + c mod a
19. b + d * a mod c - b / d
20. c - a / d * b + d mod c
21. d mod b + c - a * b / d

2.3 Логические операции

Синтаксис операций

- `AND(s/d) <оп1> <оп2> <результат>` — операция логического умножения (И) между двумя 32/64-битными операндами. Операция доступна во всех АЛК.
- `ANDn(s/d) <оп1> <оп2> <результат>` — операция логического умножения (И) между первым 32/64-битным операндом и инверсией второго 32/64-битного операнда. Операция доступна во всех АЛК.
- `OR(s/d) <оп1> <оп2> <результат>` — операция логического сложения (ИЛИ) между двумя 32/64-битными операндами. Операция доступна во всех АЛК.
- `ORn(s/d) <оп1> <оп2> <результат>` — операция логического сложения (ИЛИ) между первым 32/64-битным операндом и инверсией второго 32/64-битного операнда. Операция доступна во всех АЛК.
- `XOR(s/d) <оп1> <оп2> <результат>` — операция логического исключающего ИЛИ между двумя 32/64-битными операндами. Операция доступна во всех АЛК.
- `XORn(s/d) <оп1> <оп2> <результат>` — операция логического исключающего ИЛИ между первым 32/64-битным операндом и инверсией второго 32/64-битного операнда. Операция доступна во всех АЛК.

Рассмотрим несколько примеров выполнения операций:

- `XORs,0 %r0, %r0, %r0` — операция исключающего ИЛИ между `%r0` и `%r0`. В результате в `%r0` будет записан ноль;
- `ANDs,0 %r0, 0xffff0000, %r1` — операция логического И между значением в регистре `%r0` и константы `0xffff0000`. В результате в регистре `%r1` будет записаны старшие 16 бит из регистра `%r0`.

2.4 Сдвиги

Синтаксис операций

- `SHL(s/d) <оп1> <оп2> <результат>` — операция логического сдвига влево 32/64-битного операнда 1 на количество разрядов, указанных в операнде 2.
- `SHR(s/d) <оп1> <оп2> <результат>` — операция логического сдвига вправо 32/64-битного операнда 1 на количество разрядов, указанных в операнде 2.
- `SCL(s/d) <оп1> <оп2> <результат>` — операция циклического сдвига влево 32/64-битного операнда 1 на количество разрядов, указанных в операнде 2.
- `SCR(s/d) <оп1> <оп2> <результат>` — операция циклического сдвига вправо 32/64-битного операнда 1 на количество разрядов, указанных в операнде 2.
- `SAR(s/d) <оп1> <оп2> <результат>` — операция арифметического сдвига вправо 32/64-битного операнда 1 на количество разрядов, указанных в операнде 2.

Различия между сдвигами:

- При использовании логического сдвига влево освободившиеся младшие разряды заполняются нулями.
- При использовании логического сдвига вправо освободившиеся старшие разряды заполняются нулями.
- При использовании циклического сдвига влево освободившиеся младшие разряды заполняются выдвинутыми старшими разрядами.
- При использовании циклического сдвига вправо освободившиеся старшие разряды заполняются выдвинутыми младшими разрядами.

- При использовании арифметического сдвига вправо освободившиеся старшие разряды заполняются значением старшего разряда.



При применении циклического сдвига регистр можно считать кольцевым буфером, в котором после самого старшего разряда следует самый младший разряд.



При использовании арифметического сдвига сохраняется знак числа, а при использовании логического знак не сохраняется.

2.5 Операции сравнения

Синтаксис операций

- `CMPb(s/d)b <оп1> <оп2> <пр. регистр>` — «меньше без учёта знака» для двух 32/64-битных операндов. Операция доступна во всех АЛК, кроме 2 и 5.
- `CMPe(s/d)b <оп1> <оп2> <пр. регистр>` — «равно» для двух 32/64-битных операндов. Операция доступна во всех АЛК, кроме 2 и 5.
- `CMPbe(s/d)b <оп1> <оп2> <пр. регистр>` — «меньше или равно без учёта знака» для двух 32/64-битных операндов. Операция доступна во всех АЛК, кроме 2 и 5.
- `CMPl(s/d)b <оп1> <оп2> <пр. регистр>` — «меньше с учётом знака» для двух 32/64-битных операндов. Операция доступна во всех АЛК, кроме 2 и 5.
- `CMPle(s/d)b <оп1> <оп2> <пр. регистр>` — «меньше или равно с учётом знака» для двух 32/64-битных операндов. Операция доступна во всех АЛК, кроме 2 и 5.
- `CMPp(s/d)b <оп1> <оп2> <пр. регистр>` — проверка на «нечётный результат» для двух 32/64-битных операндов. Операция доступна во всех АЛК, кроме 2 и 5.

- `CMPs(s/d)b <оп1> <оп2> <пр. регистр>` — проверка на «отрицательный результат» для двух 32/64-битных операндов. Операция доступна во всех АЛК, кроме 2 и 5.

Операция сравнения происходит посредством вычитания операнда два из операнда один. В результате вычитания определяются флаги, по состоянию флагов определяется результат сравнения.



Результат выполнения операция CMP можно сохранять только в предикатные регистры `%pred`.

Рассмотрим несколько примеров выполнения операций:

1. `CMPesb,0 %r0, %r1, %pred0` — операция сравнения двух 32-битных операндов `%r0` и `%r1` на их равенство, результат записывается в предикатный регистр `%pred0`.

2. `CMPlesb,1 %r0, %r2, %pred1` — операция сравнения двух 32-битных операндов `%r0` и `%r2` на то, что `%r0` <= `%r2` с учётом знака, результат записывается в предикатный регистр `%pred1`.

3. `CMPpsb,3 %r0, %r3, %pred2` — операция определения флага «нечётного результата» для операции `%r0 - %r3`, результат записывается в предикатный регистр `%pred2`.



Операции «больше» в ассемблере нет. Для её реализации нужно отрицать результат операции «меньше или равно».

Исполнение под условием предикатного регистра

Многие операции можно исполнять под условием предикатного регистра, тем самым реализуя выполнение команд при условии (если условие, то команда). При выполнении операции под предикатным регистром его значения можно инвертировать. Для этого применяется символ . Рассмотрим пример на языке C:

```
int main()  {
  int a = 5;
  int b = 6;
  int res;
  if (a > b)
    res = 0;
  else
    res = 2;
  return 0;
}
```

Эквивалентным кодом на ассемблере будет (считаем, что в `%r0` и `%r1` уже находятся значения `a` и `b`):

```
{ CMPlesb,0 %r0, %r1, %pred0 }
{
  ADDs,0 0x0, %r0 ? ~%pred0
  ADDs,1 0x2, %r0 ? %pred0
}
```

В приведённом коде операция `ADDs,1 0x2, %r0 ? %pred0` будет выполнена, если результат сравнения `CMPlesb,0 %r0, %r1, %pred0` равен единице (то есть в `%r0 <= %r1`), в ином случае будет выполнена строчка кода `ADDs,0 0x0, %r0 ? %pred0`.



Всего для применения доступно 32 предикатных регистра `%pred0 - %pred31`.

Логические предикаты

При возникновении необходимости объединения условий, например `a > b` и `c < d` существует механизм логических предикатов. Всего доступно семь логических предикатов (`@p0 - @p6`).

- `pass <пр. регистр> <лог. предикат>` — перенос из предикатного регистра `@predX` в логический предикат `@p0 - @p3`. Пе-

реносить в предикаты `@p4 - @p6` запрещено.

- `pass <лог. предикат> <пр. регистр>` — перенос из логического предиката `@p4 - @p6` в предикатный регистр `@predX`. Перенос из логических предикатов `@p0 - @p3` не допускается.
- `landp <лог. предикат1> <лог. предикат2> <результат>` — логическое И между двумя логическими предикатами (можно применять инверсию предикатов). Результат записывается в один из логических предикатов `@p4 - @p6`.



Команда для передачи из предикатного регистра в логический предикат и из логического предиката в предикатный регистр называется идентично — `pass`.

Реализуем код на ассемблере, который идентичен условию `if (a > b && c < d)` на языке C. Считаем, что аргументы `a, b, c, d` были переданы в функцию на ассемблере при её вызове из кода на языке C:

```
{
  CMPlesb,0 %r0, %r1, %pred0
  CMPlsb,1 %r2, %r3, %pred1
}
{
  pass %pred0, @p0
  pass %pred1, @p1
  landp ~@p0, @p1, @p4
  pass @p4, %pred2
}
```

В результате в предикатном регистре %pred2 содержится результат подсчёта условия `if (a > b && c < d)`.



Логические предикаты живут только в одной широкой команде. Поэтому все операции над логическими предикатами: передача предикатных регистров в логические предикаты, логическое И между логическими предикатами и возврат значений в предикатный регистр — должны быть в рамках одной пары фигурных скобок.

Задания

Получите у преподавателя заготовки для написания программного кода на языке С и ассемблере Эльбруса и реализуйте выражение, указанное в задании. В результате код, написанный на С и код, написанный на ассемблере, должны выводить в консоль один и тот же результат. Значения переменных выбираются случайным образом.

Вариант задания определяется по порядковому номеру в списке группы.

Список заданий:

1. Если `a > b` и `c` чётно, то вернуть `a-c`, иначе `a+b`
2. Если `a < b` и `a > -3`, то вернуть `a*b`, иначе `a+c`
3. Если `a <= c` и `b` кратно трём, то вернуть `b-c`, иначе `a+c`
4. Если `a > c` и `c > 5`, то вернуть `a+b`, иначе `a*b`
5. Если `a >= b` и `c` кратно пяти, то вернуть `a+c`, иначе `b*c`
6. Если `a <= b` и `c < 20`, то вернуть `a*c`, иначе `a*b`
7. Если `a < c` и `b` положительное, то вернуть `a*b`, иначе `b-a`
8. Если `a <= b` и `c` отрицательное, то вернуть `a-b`, иначе `a*c`
9. Если `b > c` и `a > c`, то вернуть `c-a`, иначе `b-c`
10. Если `b < c` и `a > b`, то вернуть `a+b`, иначе `b*c`

11. Если `b <= c` и `a` чётно, то вернуть `b-a`, иначе `a+c`
12. Если `b > a` и `b` нечётно, то вернуть `c-b`, иначе `b-a`
13. Если `b < a` и `a > c`, то вернуть `b-a`, иначе `c+a`
14. Если `b <= a` и `c <= b`, то вернуть `a*b`, иначе `b*c`
15. Если `b >= c` и `c == 4`, то вернуть `a-b`, иначе `b-a`
16. Если `c <= b` и `a > 25`, то вернуть `b+c`, иначе `c-b`
17. Если `c > b` и `c < a`, то вернуть `a+c`, иначе `b*c`
18. Если `c > a` и `c` не кратно 3, то вернуть `a*c`, иначе `b-c`
19. Если `c <= a` и `a > b`, то вернуть `a-c`, иначе `c-a`
20. Если `c >= b` и `b > a`, то вернуть `a+b`, иначе `c-b`

2.6 Операции подготовки передачи управления и передачи управления

В данном разделе рассматриваются только подготовленная передача управления.

Синтаксис операций

`RETURN %ctpr3` — операция подготовки возврата из процедуры, она включает:

- восстановление указателя команды IP. Указатель команды (IP) задаётся операцией RETURN (считывается из стека связующей информации);
- восстановление регистрового окна и фрейма стека пользователя; окно и фрейм считываются из стека связующей информации операцией CT;

- восстановление контекста; новый контекст включает глобальные переменные, литералы и классы. Контекст автоматически считывается из памяти операцией CT.

Передача управления осуществляется только операцией `CT`.

`DISP <регистр управления> <метка>` — подготовка перехода по литеральному смещению. Позволяет подготовить передачу управления на определённую именованную область кода (метку), связав регистр управления с указанной областью памяти.

Операции `RETURN` и `DISP` работают с регистрами подготовки перехода `%ctpr`. Всего доступно 3 регистра `%ctpr1 - %ctpr3`. Все типы перехода за исключением `RETURN` могут использовать любой из этих регистров управления. Операция `RETURN` реализуется только на `%ctpr3`.

`CT <регистр управления>` — операция передачи управления из одной программной ветви в другую.

Пример

Напишем программный код на ассемблере, реализуеющий следующее условие с помощью меток: если `a > b`, то вернуть `a * b`, иначе `a - b`. В конце кода должен быть возврат из функции.

```
  {
    DISP %ctpr1, lelse
    CMPlesb,0 %r0, %r1, %pred0
  }
  { CT %ctpr1 ? %pred0 }
  {
    DISP %ctpr2, end
    MULs,0 %r0, %r1, %r0
  }
  { CT %ctpr2 }
lelse:
  { SUBs,0 %r0, %r1, %r0 }
end:
```

```
{
  RETURN %ctpr3
}
{ CT %ctpr3 }
```

В приведённом коде:

1. `DISP %ctpr1, lelse` — связывает регистр подготовки перехода `%ctpr1` и метку `lelse`;
2. `CMPlesb,0 %r0, %r1, %pred0` — сохраняет в `%pred0` результат сравнения `%r0 <= %r1`, то есть `a <= b`;
3. `CT %ctpr1 ? %pred0` — переход по адресу в регистре `%ctpr1` (метка `lelse`) при условии, что `%pred0` равен единице. Если переход состоялся, то последующим будет выполняться код, начиная со строчки с названием метки `lelse` (пункт №7). Иначе код, который идёт дальше после этой строки.
4. `DISP %ctpr2, end` — связывает регистр подготовки перехода `%ctpr2` и метку `end`;
5. `MULs,0 %r0, %r1, %r0` — умножение `%r0` на `%r1` и запись результата в `%r0`, то есть `%r0 = a * b`;
6. `CT %ctpr2` — переход по адресу в регистре `%ctpr2` (метка `end`);
7. `SUBs,0 %r0, %r1, %r0` — вычитание `%r1` из `%r0` и запись результата в `%r0`, то есть `%r0 = a - b`;
8. `RETURN %ctpr3` — подготовка к возврату из процедуры `%ctpr3`;
9. `CT %ctpr3` — возврат из процедуры.

Задания

Реализуйте задание из прошлого раздела **Операции сравнения**, но с применением меток.

2.7 Операции считывания из памяти

Операция считывания читает несколько байтов из пространства памяти и помещает их в регистр назначения. Операция считывания доступна в `0, 2, 3, 5` АЛК.

Синтаксис операции

- `LDb <адрес> <[маска]> <результат>` — считывание байта без знака;
- `LDh <адрес> <[маска]> <результат>` — считывание полуслова (16 бит) без знака;
- `LDw <адрес> <[маска]> <результат>` — считывание одинарного слова (32 бита);
- `LDd <адрес> <[маска]> <результат>` — считывание двойного слова (64 бита).



Маска является необязательным аргументом и её можно не указывать.

Пример

С помощью операций считывания из памяти можно получать значения элементов массива. Реализуем объявление массива на языке C, а обработку его значений на ассемблере.

Код на C:

```
#include <stdio.h>

int calc_asm(int*);

int ar[2] = {1, 2};
```

```
int main() {
  int res = calc_asm(ar);
}
```

Код на ассемблере:

```
calc_asm:
  {
    LDw,0 %r0, %r1
    ADDs,1 %r0, 0x4, %r0
  }
  { LDw,0 %r0, %r2 }
  { ADDs,0 %r1, %r2, %r0 }
```

В ассемблере нет возможности обращаться к элементам массива по индексам, поэтому чтение элементов массива происходит посредством указателя (адреса на ячейку памяти). В регистр `%r0` передан адрес на первый элемент массива. Применив операцию `LDw,0 %r0, %r1`, в регистре `%r1` будет находиться первый элемент массива. Чтобы перейти к следующему элементу массива, требуется сдвинуть адрес на то количество байт, которое занимают данные в массиве. Для целочисленного типа данных `int` это значение равно четырём. Для `float`, `long`, `double` — восьми. После операции `ADDs,1 %r0, 0x4, %r0` адрес будет указывать на второй элемент массива. Таким образом, чтобы считывать массив, потребуется два действия: считывание элемента, сдвиг адреса до следующего элемента.



Если попытаться считать элемент за границами массива, то никакой ошибки не последует.

Рассмотрим организацию цикла для обработки элементов массива. В цикле будет подсчитывать сумму всех чётных элементов.

Код на C:

```
#include <stdio.h>
```

```
int calc_asm(int*, int);
int ar[100];

int main() {
  for (int i=0;i<100;i++) {
    ar[i]=i;
  }
  int res = calc_asm(ar, 100);
  printf("%d\n", res);
}
```

Код на ассемблере:

```
calc_asm:
  {
    ADDs,0 0x0, %r3
    ADDs,1 0x0, %r4
    DISP %ctpr1, LOOP
  }
LOOP:
  { LDw,0 %r0, %r2 }
  { ADDs,1 %r0, 0x4, %r0 }
  { uMODx,5 %r2, 2, %r5 }
  { CMPesb,0 %r5, 0, %pred0 }
  { ADDs,0 %r4, %r2, %r4 ? %pred0 }
  { ADDs,0 %r3, 1, %r3 }
  { CMPlsb,0 %r3, %r1, %pred1 }
  { CT %ctpr1 ? %pred1 }
  { ADDs,0 %r4, %r0 }
```

Построчно рассмотрим, что делает код.

1. `ADDs,0 0x0, %r3` — обнуление регистра `%r3`. Регистр `%r3` будем использовать как номер итерации цикла;

2. `ADDs,0 0x0, %r4` — обнуление регистра `%r4`. Регистр `%r4` будем использовать для накопления суммы чётных элементов;

3. `DISP %ctpr1, LOOP` — связывает регистр подготовки перехода `%ctpr1` и метку `LOOP`;

4. `LDw,0 %r0, %r2` — считывание значения, которое располагается по адресу, указанному в регистре `%r0`, и запись этого значения в регистр `%r2`;

5. `ADDs,1 %r0, 0x4, %r0` — перемещение указателя на следующий элемент массива;

6. `uMODx,5 %r2, 2, %r5` — получение остатка от деления значения регистра `%r2` на число 2;

7. `CMPesb,0 %r5, 0, %pred0` — проверка равенства значения регистра `%r5` на ноль. То есть является ли число чётным;

8. `ADDs,0 %r4, %r2, %r4 ? %pred0` — если число чётное, то накапливать сумму чётных элементов;

9. `ADDs,0 %r3, 1, %r3` — увеличение номера итерации цикла на единицу;

10. `CMPlsb,0 %r3, %r1, %pred1` — проверка номера итерации цикла (в `%r1` находится число 100);

11. `CT %ctpr1 ? %pred1` — если номер цикла меньше количества элементов, то переход в начало цикла (на метку `LOOP`); Если номер цикла больше или равен количеству элементов, то строка пропускается и выполняется следующая;

12. `ADDs,0 %r4, %r0` — перенос значения регистра `%r4` в регистр `%r0`.

Код на ассемблере можно более компактно распределить по широким командам для параллельного исполнения:

```
calc_asm:
  {
    ADDs,0 0x0, %r3
    ADDs,1 0x0, %r4
    DISP %ctpr1, LOOP
  }
LOOP:
  {
    LDw,0 %r0, %r2
    ADDs,1 %r0, 0x4, %r0
    ADDs,2 %r3, 1, %r3
  }
  {
    uMODx,5 %r2, 2, %r5
    CMPlsb,0 %r3, %r1, %pred1
  }
  {
    CMPesb,0 %r5, 0, %pred0
  }
  {
    ADDs,0 %r4, %r2, %r4 ? %pred0
    CT %ctpr1 ? %pred1
  }
  { ADDs,0 %r4, %r0 }
```

Логика исполнения кода осталась неизменной, только операции были перетасованы в широкие команды для параллельного исполнения.

Задания

Получите у преподавателя заготовки для написания программного кода на языке C и ассемблере Эльбруса и реализуйте выражение, указанное в задании. В результате код, написанный на C и код, написанный на ассемблере, должны выводить в консоль один и тот же результат. Значения переменных выбираются случайным образом.

Вариант задания определяется по порядковому номеру в списке группы.



Так как при накоплении произведения элементов может случиться ситуация с выходом за границы разрядной сетки, то после каждого вычисления результата произведения вычисляйте остаток от деления этого результата на 10001 и записывайте обратно в регистр с результатом.

Задание:

1. Сумма элементов с 5 по 63 индексы (включительно).
2. Произведение элементов, значение которых больше или равно 500 и меньше 1500.
3. Сумма элементов с индексами до 20 и после 60 (включительно).
4. Сумма элементов с чётными индексами и чётными значениями.
5. Произведение элементов с нечётными индексами и нечётными значениями.
6. Сумма элементов, значение которых больше 666, а индексы больше 33.
7. Сумма элементов с чётными индексами и значениями кратными 5.
8. Произведение элементов, значение которых меньше 3500, а индексы меньше 60.
9. Сумма элементов со значениями меньше 666 или больше 6666 (не включительно).
10. Сумма элементов с чётными индексами и нечётными значениями.
11. Сумма элементов, значение которых больше 5000 и кратно 3.
12. Сумма элементов с нечётными индексами и чётными значениями.

13. Произведение элементов с 13 по 77 индексы.
14. Произведение элементов с чётными индексами и чётными значениями.
15. Сумма элементов с нечётными индексами и нечётными значениями.
16. Сумма элементов со значениями меньше 1234 или больше 5678 (не включительно).
17. Произведение элементов, значение которых меньше 3222 и кратно 4.
18. Сумма элементов, значение которых больше 6666, а индексы больше 66.
19. Сумма элементов, значения которых находятся в диапазоне от 1000 до 2000 (включительно).
20. Произведение элементов с индексами до 25 и после 66 (включительно).

2.8 Операции записи в память

Операции записи сохраняют несколько байтов из регистра источника в пространство памяти. Операции записи доступны в `2` и `5` АЛК.

Синтаксис операции

- `STb <источник> <адрес> <[маска]>` — запись байта;
- `STh <источник> <адрес> <[маска]>` — запись полуслова (16 бит);
- `STw <источник> <адрес> <[маска]>` — запись одинарного слова (32 бита);
- `STd <источник> <адрес> <[маска]>` — запись двойного слова (64 бита).

Пример

С помощью операции записи реализуем на ассемблере возврат двух чисел — сумма чётных элементов и произведение нечётных элементов массива.

Код на C:

```
#include <stdio.h>

void calc_asm(int*, int, int*);
int ar[20];
int out[2];

int main() {
  for (int i=0;i<20;i++) {
    ar[i]=i;
  }
  calc_asm(ar, 20, out);
  printf("%d # %d\n", out[0], out[1]);
}
```

Код на ассемблере:

```
  calc_asm:
    {
      ADDs,0 0x0, %r4
      ADDs,1 0x0, %r5
      ADDs,2 0x1, %r6
      DISP %ctpr1, LOOP
    }
  LOOP:
    {
      LDw,0 %r0, %r3
      ADDs,1 %r0, 0x4, %r0
      ADDs,2 %r4, 1, %r4
    }
```

```
{
  uMODx,5 %r3, 2, %r7
  CMPlsb,0 %r4, %r1, %pred1
}
{
  CMPesb,0 %r7, 0, %pred0
}
{
  ADDs,0 %r5, %r3, %r5 ? %pred0
  MULs,1 %r6, %r3, %r6 ? ~%pred0
  CT %ctpr1 ? %pred1
}
{
  STw,2 %r2, %r5
  ADDs,0 %r2, 0x4, %r2
}
{ STw,2 %r2, %r6 }
```

Код работает аналогично тому, что описан в разделе **Операции считывания из памяти**, кроме пары моментов:

1. Добавлено вычисление произведения нечётных элементов массива — `MULs,1 %r6, %r3, %r6 ?  %pred0`;
2. Добавлена запись элемента массива в память — `STw,2 %r2, %r5`. Принцип записи в память аналогичен принципу чтения: необходимо каждый раз увеличивать значение регистра, где хранится адрес элемента массива, на то количество байт, сколько места занимает один его элемент.

Реализуем обработку двумерного массива. Посчитаем отдельно сумму элементов в чётных и нечётных строках массива.

Код на C:

```
#include <stdio.h>
```

```
void calc_asm(int[][10], int, int, int*);

int ar[20][10];
int out[2];

int main() {
  for (int i=0;i<20;i++) {
    for (int j=0;j<10;j++)
      ar[i][j]=i*10+j;
  }
  int a = 0;
  int b = 0;

  for (int i=0;i<20;i++)
    for (int j=0;j<10;j++)
      if (i % 2 == 0)
        a += ar[i][j];
      else
        b += ar[i][j];

  printf("%d # %d\n", a, b);
  calc_asm(ar, 20, 10, out);
  printf("%d # %d\n", out[0], out[1]);
}
```

Код на ассемблере:

```
calc_asm:
{
  ADDs,0 0x0, %r11
  ADDs,1 0x0, %r10
  ADDs,2 0x0, %r9
  ADDs,3 0x0, %r8
  DISP %ctpr1, OLOOP
}
```

```
{
  DISP %ctpr2, ILOOP
}

OLOOP:
ILOOP:
{
  LDw,0 %r0, %r4
  ADDs,1 %r0, 0x4, %r0
  ADDs,2 %r10, 0x1, %r10
  uMODx,5 %r11, 0x2, %r7
}
{
  CMPesb,0 %r7, 0x0, %pred1
  CMPlsb,1 %r10, %r2, %pred2
}
{
  ADDs,0 %r9, %r4, %r9 ? %pred1
  ADDs,1 %r8, %r4, %r8 ? ~%pred1
  CT %ctpr2 ? %pred2
}
{
  ADDs,0 %r11, 0x1, %r11
  ADDs,1 0x0, %r10
}
  CMPlsb,0 %r11, %r1, %pred2
  CT %ctpr1 ? %pred2
{
  STw,2 %r3, %r9
  ADDs,1 %r3, 0x4, %r3
}
{
  STw,2 %r3, %r8
}
```

В коде на ассемблере организуется два цикла — внешний (аналог внешнего цикла `for` с индексом `i`) и внутренний (аналог цикла с индексом `j`). Подробно разберём программный код:

0. При вызове функции из языка C передаются следующие параметры, которые попадают в соответствующие им регистры:

    - `ar` — целочисленный массив, который необходимо обработать. В регистре `%r0` расположен адрес первого элемента массива;
    - `20` — количество строк в массиве. Расположено в регистре `%r1`;
    - `10` — количество элементов в каждой строке. Расположено в регистре `%r2`;
    - `out` — массив для возврата значений. В регистре `%r3` расположен адрес первого элемента массива.

1. Первая широкая команда (до начала внешнего цикла):

    - `ADDs,0 0x0, %r11` — обнуление регистра `%r11`. В регистре `%r11` расположен номер строки (аналог переменной `i`);
    - `ADDs,1 0x0, %r10` — обнуление регистра `%r10`. В регистре `%r10` расположен номер столбца (аналог переменной `j`);
    - `ADDs,2 0x0, %r9` — обнуление регистра `%r9`. В регистре `%r9` накапливается сумма элементов чётных строк;
    - `ADDs,3 0x0, %r8` — обнуление регистра `%r8`. В регистре `%r8` накапливается сумма элементов нечётных строк;
    - `DISP %ctpr1, OLOOP` — связывает регистр подготовки перехода `%ctpr1` и метку `OLOOP` (начало внешнего цикла).

2. Вторая широкая команда (до начала внешнего цикла):

    - `DISP %ctpr2, ILOOP` — связывает регистр подготовки перехода `%ctpr2` и метку `ILOOP` (начало внутреннего цикла).

3. Третья широкая команда (внутренний цикл):

    - `LDw,0 %r0, %r4` — считывание значения, которое располагается по адресу, указанному в регистре `%r0`, и запись этого значения в регистр `%r4`;
    - `ADDs,1 %r0, 0x4, %r0` — перемещение указателя на следующий элемент массива;
    - `ADDs,2 %r10, 0x1, %r10` — увеличение номера столбца на 1;
    - `uMODx,5 %r11, 0x2, %r7` — получение остатка от деления номера строки на `2`.

4. Четвёртая широкая команда (внутренний цикл):

    - `CMPesb,0 %r7, 0x0, %pred1` — проверка номера строки на чётность;
    - `CMPlsb,1 %r10, %r2, %pred2` — проверка окончания внутреннего цикла. В `%pred2` находится 1, если номер столбца меньше количества элементов в строке.

5. Пятая широкая команда (внутренний цикл):

    - `ADDs,0 %r9, %r4, %r9 ? %pred1` — накапливание суммы, если номер строки чётный;
    - `ADDs,1 %r8, %r4, %r8 ? %pred1` — накапливание суммы, есил номер строки нечётный;
    - `CT %ctpr2 ? %pred2` — возврат к началу внутреннего цикла, не все столбцы были обработаны.

6. Шестая широкая команда (внешний цикл):

    - `ADDs,0 %r11, 0x1, %r11` — увеличение номера строки на `1`;
    - `ADDs,1 0x0, %r10` — обнуление регистра `%r10` с номером столбца.

7. Седьмая широкая команда (внешний цикл):

    - `CMPlsb,0 %r11, %r1, %pred2` — проверка окончания внешнего цикла;
    - `CT %ctpr1 ? %pred2` — возврат к началу внешнего цикла, если не все строки были обработаны.

8. Восьмая широкая команда (после циклов):

    - `STw,2 %r3, %r9` — запись суммы элементов с чётных строк в массив вывода;
    - `ADDs,1 %r3, 0x4, %r3` — перемещение указателя на следующий элемент массива.

9. Девятая широкая команда (после циклов):

    - `STw,2 %r3, %r8` — запись суммы элементов с нечётных строк в массив вывода.

Задания

Получите у преподавателя заготовки для написания программного кода на языке C и ассемблере Эльбруса и реализуйте выражение, указанное в задании. В результате код, написанный на C и код, написанный на ассемблере, должны выводить в консоль один и тот же результат. Значения переменных выбираются случайным образом.

Вариант задания определяется по порядковому номеру в списке группы.

Первое задание: После каждого вычисления произведения от результата получать остаток по модулю 100001 (сто тысяч один) и записывать обратно в результат, чтобы не происходило переполнение.

Вернуть из функции на ассемблере два числа.

1. Первое - Произведение элементов с нечётными индексами и нечётными значениями. Второе - сумма оставшихся элементов.

2. Первое - Произведение элементов, значение которых больше или равно 500 и меньше 1500. Второе - сумма оставшихся элементов.

3. Первое - Сумма элементов со значениями от 1234 до 5678 (не включительно). Второе - произведение оставшихся элементов.

4. Первое - Произведение элементов, значение которых меньше 3222 и кратно 4. Второе - сумма оставшихся элементов.

5. Первое - Сумма элементов со значениями от 666 до 6666 (не включительно). Второе - произведение оставшихся элементов.

6. Первое - Сумма элементов с чётными индексами и нечётными значениями. Второе - произведение оставшихся элементов.

7. Первое - Сумма элементов, значение которых больше 5000 и кратно 3. Второе - произведение оставшихся элементов.

8. Первое - Сумма элементов с нечётными индексами и чётными значениями. Второе - произведение оставшихся элементов.

9. Первое - Сумма элементов с нечётными индексами и нечётными значениями. Второе - произведение оставшихся элементов.

10. Первое - Сумма элементов, значение которых больше 6666, а индексы больше 66. Второе - произведение оставшихся элементов.

11. Первое - Сумма элементов с чётными индексами и значениями кратными трём. Второе - произведение оставшихся элементов.

12. Первое - Произведение элементов с индексами до 25 и после 66 (включительно). Второе - сумма оставшихся элементов.

13. Первое - Сумма элементов, значение которых больше 666, а индексы больше 33. Второе - произведение оставшихся элементов.

14. Первое - Сумма элементов с чётными индексами и значениями кратными 5. Второе - произведение оставшихся элементов.

15. Первое - Произведение элементов, значение которых меньше 3500, а индексы меньше 60. Второе - сумма оставшихся элементов.

16. Первое - Сумма элементов с 5 по 63 индексы (включительно). Второе - произведение оставшихся элементов.

17. Первое - Произведение элементов с индексами от 20 до 60 (включительно). Второе - сумма оставшихся элементов.

18. Первое - Сумма элементов с чётными индексами и чётными значениями. Второе - произведение оставшихся элементов.

19. Первое - Произведение элементов с 13 по 77 индексы. Второе - сумма оставшихся элементов.

20. Первое - Произведение элементов с чётными индексами и чётными значениями. Второе - сумма оставшихся элементов.

Второе задание:

1. Для чисел в диапазоне от 500 до 2500 найти минимум. Для чисел вне диапазона найти максимум среди них.

2. Для чисел с чётными индексами и значениями меньше 5000 найти максимум. Для чисел вне диапазона найти минимум среди них.

3. Для элементов с индексами с 10 до 59 найти максимум. Для оставшихся элементов найти количество чётных чисел.

4. Для чисел с чётными значениями и чётными индексами найти минимум. Для оставшихся чисел найти среднее арифметическое.

5. Для чисел со значениями больше 5000 и кратными трём найти максимум. Для оставшихся чисел найти количество нечётных.

6. Для элементов с нечётными индексами и нечётными значениями найти среднее арифметическое. Для оставшихся найти минимимум среди них.

7. Для чисел в диапазоне от 3333 до 6666 найти количество чётных чисел. Для чисел вне диапазона найти минимум среди них.

8. Для элементов с индексами меньше 60 и чётными значениями найти максимум. Для чисел вне диапазона найти количество нечётных.

9. Для элементов с индексами больше 40 и значениями меньше 2000 найти максимум. Для оставшихся элементов найти минимум.

10. Для чисел с нечётными значениями и чётными индексами найти минимум. Для оставшихся чисел найти среднее арифметическое.

11. Для элементов с индексами с 60 до 70 найти количество нечётных чисел. Для оставшихся элементов найти максимум.

12. Для чисел в диапазоне от 3333 до 7777 найти максимум. Для чисел вне диапазона найти минимум.

13. Для чисел со значениями больше 5000 и кратными четырём найти минимум. Для оставшихся чисел найти максимум.

14. Для чисел в диапазоне от 333 до 3333 найти минимум. Для чисел вне диапазона найти среднее арифметическое.

15. Для элементов с индексами меньше 60 и нечётными значениями найти максимум. Для чисел вне диапазона найти количество нечётных чисел.

16. Для элементов с индексами с 5 до 75 найти максимум. Для оставшихся элементов найти минимум среди них.

17. Для элементов с индексами больше 60 и чётными значениями найти минимум. Для чисел вне диапазона найти количество чётных чисел.

18. Для чисел с чётными значениями и чётными индексами найти минимум. Для оставшихся чисел найти среднее арифметическое.

19. Для чисел с чётными значениями и нечётными индексами найти минимум. Для оставшихся чисел найти максимум.

20. Для элементов с индексами с 43 до 88 найти минимум. Для оставшихся элементов найти количество чётных чисел.

Третье задание:

1. Если индекс чётный и значение чётное, то найти минимум; Иначе если индекс нечётный и значение нечётное, найти среднее арифметическое; Иначе найти максимум.

2. Если значения в диапазоне от 1000 до 4000, то найти максимум среди них; Иначе если значения в диапазоне от 4001 до 7001, то найти минимум среди них; Иначе найти среднее арифметическое оставшихся значений.

3. Если значение меньше 4000 и индекс меньше 40, то найти количество чётных; Иначе если значение меньше 8000 и индекс меньше 80, то найти количество нечётных; Иначе найти минимум.

4. Если значение кратно 2 и индекс меньше 20, то найти максимум; Иначе если значение кратно 3 и индекс больше 60, то найти минимум; Иначе найти среднее арифметическое.

5. Если индекс нечётный и значение чётное, то найти максимум; Иначе если индекс чётный и значение нечётное, то найти минимум; Иначе найти среднее арифметическое.

6. Если индекс больше 10 и меньше 40, то найти среднее арифметическое; Иначе если индекс от 41 до 88, то найти количество чётных; Иначе найти минимум.

7. Если значение больше 5000 и индекс больше 60, то найти количество чётных; Иначе если индекс меньше 40 и значение больше 5001, то найти среднее арифметическое; Иначе найти максимум.

8. Если значение чётное и индекс больше 22, то найти среднее арифметическое; Иначе если значение нечётное и индекс меньше 22, то найти минимум; Иначе найти количество чётных.

9. Если значение меньше 5555 и индекс меньше 55, то найти минимум; Иначе если значение больше 5555 и индекс больше 55, то найти максимум; Иначе найти среднее арифметическое.

10. Если индексы больше 10 и меньше 60, то найти максимум; Иначе если значение больше 1000 и меньше 6000, то найти минимум; Иначе найти количество нечётных.

11. Если значение больше 333 и индекс кратен трём, то найти среднее арифметическое; Иначе если индекс кратен двум и значение кратно двум, то найти максимум; Иначе найти минимум.

12. Если индексы между 10 и 77, то найти количество чётных; Иначе если индексы чётные и значения нечётные, то найти количество нечётных; Иначе найти среднее арифметическое.

13. Если индексы нечётные и значения кратны пяти, то найти минимум; Иначе если индексы чётные и значения больше 5555, то найти максимум; Иначе найти среднее арифметическое.

14. Если значения больше 5000 и меньше 9000, то найти максимум; Иначе если индексы больше 50 и меньше 90, то найти минимум; Иначе найти количество чётных.

15. Если значения в диапазоне от 4001 до 7001, то найти минимум; Иначе если индексы чётные и значения меньше 4000, то найти максимум; Иначе найти среднее арифметическое.

16. Если индексы чётные и значения чётные, то найти количество чётных; Иначе если индексы нечётные и значения нечётные, то найти количество нечётных; Иначе найти максимум.

17. Если значения кратны пяти и меньше 5000, то найти минимум; Иначе если значения больше 5001 и кратны четырём, то найти максимум; Иначе найти среднее арифметическое.

18. Если индекс меньше 40 и значение больше 6666, то найти среднее арифметическое; Иначе если индекс от 41 до 88, то найти количество чётных; Иначе найти минимум.

19. Если индексы от 55 до 77, то найти максимум; Иначе если индекс чётный и значение чётное, то найти минимум; Иначе найти среднее арифметическое.

20. Если индексы кратны четырём и значения кратны четырём, то найти количество чётных; Иначе если индексы кратны трём и индексы больше 44, то найти минимум; Иначе найти максимум.

Четвёртое задание:

1. Для каждой строки массива посчитать сумму её элементов;

2. Для каждого столбца посчитать сумму его элементов;

3. Посчитать общую сумму элементов, которые находятся только в чётных строках;

4. Найти среднее арифметическое элементов, находящихся в нечетных столбцах;

5. Посчитать сумму элементов, у которых сумма индексов строки и столбца четная;

6. В каждой строке найти максимальный элемент и вывести его;

7. Найти сумму элементов, у которых индекс строки больше индекса столбца;

8. Найти минимальный элемент среди тех, у которых совпадает номер строки и номер столбца;

9. В каждом столбце найти минимальный элемент и вывести его;
10. Подсчитать количество чётных чисел в каждом столбце;
11. Подсчитать количество нечётных чисел в каждой строке;
12. Посчитать сумму всех элементов, которые стоят по периметру массива (первая строка, последняя строка, первый столбец, последний столбец);
13. Посчитать сумму элементов, у которых совпадает номер строки и номер столбца;
14. Найти среднее арифметическое элементов, находящихся в чётных столбцах;
15. В каждой строке найти минимальный элемент и вывести его;
16. В каждом столбце найти максимальный элемент и вывести его;
17. Найти сумму элементов, которые находятся одновременно в нечётных строках и чётных столбцах;
18. Найти сумму элементов, у которых индекс строки меньше индекса столбца;
19. Найти среднее арифметическое элементов, находящихся в чётных строках;
20. Найти максимальный элемент среди тех, у которых совпадает номер строки и номер столбца.

2.9 Соответствие операций и АЛК

Ниже представлена таблица наименований операций, их синтаксиса и арифметико-логических каналов, в которых их можно исполнять.

Таблица 2.1 – Соответствие операция и АЛК

Операция	Операнды	Доступные АЛК
ADD(s/d)	оп1 оп2 результат	0-5
SUB(s/d)	оп1 оп2 результат	0-5
MUL(s/d)	оп1 оп2 результат	0, 1, 3, 4
(u/s)MULx	оп1 оп2 результат	0, 1, 3, 4
(u/s)DIVs/d	оп1 оп2 результат	5
(u/s)DIVx	оп1 оп2 результат	5
(u/s)MODx	оп1 оп2 результат	5
AND(s/d)	оп1 оп2 результат	0-5
ANDn(s/d)	оп1 оп2 результат	0-5
OR(s/d)	оп1 оп2 результат	0-5
ORn(s/d)	оп1 оп2 результат	0-5
XOR(s/d)	оп1 оп2 результат	0-5
XORn(s/d)	оп1 оп2 результат	0-5
SHL(s/d)	оп1 оп2 результат	0-5
SHR(s/d)	оп1 оп2 результат	0-5
SCL(s/d)	оп1 оп2 результат	0-5
SCR(s/d)	оп1 оп2 результат	0-5
SAR(s/d)	оп1 оп2 результат	0-5
CMPb(s/d)b	оп1 оп2 предикат	0, 1, 3, 4
CMPe(s/d)b	оп1 оп2 предикат	0, 1, 3, 4
CMPbe(s/d)b	оп1 оп2 предикат	0, 1, 3, 4
CMPl(s/d)b	оп1 оп2 предикат	0, 1, 3, 4
CMPle(s/d)b	оп1 оп2 предикат	0, 1, 3, 4
CMPp(s/d)b	оп1 оп2 предикат	0, 1, 3, 4
CMPs(s/d)b	оп1 оп2 предикат	0, 1, 3, 4
pass	предикат лпред	АЛК не указывается
pass	лпред предикат	АЛК не указывается
landp	лпред1 лпред2 рез.	АЛК не указывается
RETURN	%ctpr3	АЛК не указывается
DISP	рег. управления метка	АЛК не указывается
CT	регистр управления	АЛК не указывается
LD(b/h/w/d)	адрес маска результат	0, 2, 3, 5
ST(b/h/w/d)	источник адрес маска	2, 5

Заключение

Уважаемый читатель!

В данном пособии вы познакомились с часто используемыми командами ассемблера Эльбрус. Вы научились применять их для решения базовых задач: вычисления математических выражений, организации ветвлений и циклов, чтения из памяти и записи в память.

В пособии намеренно отсутствует информация о спекулятивном исполнении команд. В качестве направления для дальнейшего развития рекомендуем самостоятельно ознакомиться с этим важным механизмом, а также обратить внимание на более плотное заполнение широких команд — на методы оптимизации кода.