Регистровый файл. Память. Программируемое устройство

Лабораторная работа #3

В рамках лабораторной работы необходимо: научиться синтезировать адресуемые регистровые структуры, задействовать блоки памяти и реализовывать устройство с примитивным программным управлением.

Основной ход выполнения работы выглядит так:

1. Синтез регистрового файла и памяти с побайтовой адресацией
2. Верификация синтезированных модулей памяти
3. Реализация программируемого устройства на основе RAM, RF и ALU
4. Проверка функционирования устройства на отладочном стенде

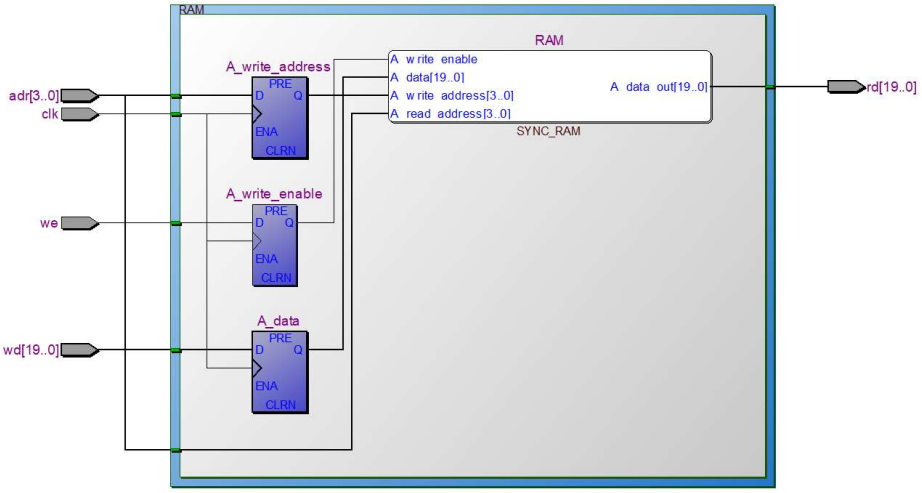
Ни один традиционный процессор не обходится без: арифметико-логического устройства (ALU), выполняющего арифметические и поразрядно логические операции, подключенного к нему регистрового файла (RF) и основной памяти (RAM), где хранятся все данные и исполняемые программы (если речь идет о принстонской архитектуре организации памяти). Предыдущая лабораторная работа была посвящена синтезу и верификации АЛУ. Эта работа посвящена созданию трехпортового регистровый файл и синтезу памяти в ПЛИС.

Ниже приводится пример Verilog HDL кода демонстрирующего описание модуля адресуемой памяти `RAM` (`Random Access Memory`, память с произвольным доступом). Синтезируемый модуль имеет 16 адресуемых слов (ячеек памяти) разрядностью `20` бит каждое. Чтобы адресовать 16 слов необходима 4-битная адресная шина `adr`, потому что $\log_2(16) = 4$.

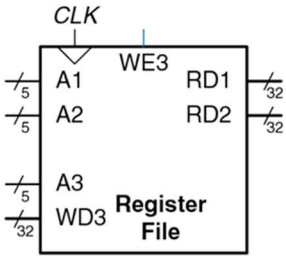
Демонстрируемая память имеет один порт чтения/записи, это значит, что в один момент времени можно обратиться только к одной из имеющихся 16 ячеек памяти, то есть память имеет единственный адресный вход. `rd` — 20-битный выходной сигнал отображающий содержимое ячейки памяти (слова) по адресу `adr`. Разрешение на

запись управляется сигналом we. wd — 20-битные входные данные для записи в ячейку по адресу adr, если we == 1. clk – сигнал синхронизации. Запись в память происходит синхронно, то есть только в момент переключения сигнала clk из 0 в 1.

```
1  |  module mem16_20 (
2  |        input                clk,
3  |        input      [3:0]     adr,  // address
4  |        input      [19:0]    wd,   // Write Data
5  |        input                we,   // Write Enable
6  |        output     [19:0]    rd    // Read Data
7  |        );
8  |  reg [19:0] RAM [0:15];  // создать память из 16-ти 20-битных ячеек
9  |
10 |  assign rd = RAM[adr];   // подключение выхода rd к
11 |                          // ячейке памяти с адресом adr
12 |  always @ (posedge clk)              // запись данных wd
13 |        if (we) RAM[adr] <= wd;       // в ячейку по адресу adr,
14 |                                      // если we == 1
15 |  endmodule
```



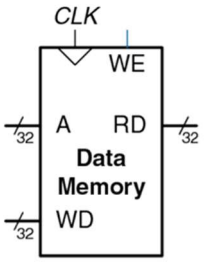
Первое задание



(1 часть задания) Необходимо описать на языке Verilog HDL модуль регистрового файла под названием RF, который содержит 32 адресуемых регистра и имеет 2 порта для чтения и один порт для записи. На картинке справа дано условное

изображение разрабатываемого устройства. A1[4:0] — это 5-битный адрес одного из 32-х адресуемых регистров, содержимое которого подается на 32-х битный выход RD1[31:0]. A1[4:0] и RD1[31:0] относятся к первому порту чтения регистрового файла. A2[4:0] и RD2[31:0] — выполняют аналогичные функции и являются вторым портом для чтения. Остальные входы и выходы относятся к третьему порту – записи. Если сигнал WE3 установлен в единицу, то по фронту CLK по адресу A3[4:0] будут записаны данные со входа WD3[31:0]. Также необходимо предусмотреть сигнал reset (на рисунке не обозначен) синхронно сбрасывающий содержимое всех регистров в ноль.

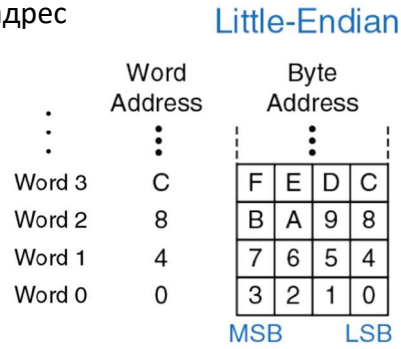
На практике оказывается очень удобно иметь простой доступ к такой часто используемой константе как 0. Это можно реализовать, например, заменив один из регистров на константное значение нуля. Записывать в этот регистр не имеет смысла — он всегда хранит ноль. Строго говоря, это даже не будет ячейка памяти. Разрабатываемый регистровый файл должен иметь константный 0 по адресу 0.



(2 часть задания) После разработки RF необходимо реализовать отдельный модуль памяти под названием DM (Data Memory — картинка справа) с одним портом чтения/записи хранящий 32 битные слова, но имеющий побайтовую адресацию.

Побайтовая адресация означает, что каждый байт (Byte) в памяти имеет свой уникальный адрес. Адрес подается на 32-х битный вход A[31:0]. При этом необходимо реализовать считывание не одного байта, а одного 32-х битного слова (Word) – 4-х байт, которые можно получить на выходе RD[31:0]. RD — это 32-битное содержимое четырех последовательных байтовых ячеек памяти (4 байта это 4 x 8 = 32 бита), по адресам A + 3 (подключается к выходам RD[31:24]), A + 2 (подключается к выходам RD[23:16]), A + 1 (подключается к выходам RD[15:8]) и A (подключается к выходам RD[7:0]). Доступ к памяти должен быть выровнен, это значит, что адрес слова должен быть кратен количеству входящих в него байт, то есть 4-м. Значит, при считывании слова два младших бита адреса A[1:0] должны быть равны нулю.

Например, второе слово (`Word 2` на картинке) имеет адрес 8 (`Word address`) и включает в себя байты по адресам 8, 9, A и B (`Byte address`). Если отбросить два младших бита адреса, получается порядковый номер слова.



$8_{10} = 1000_2 \Rightarrow 10_2 = 2_{10}$.

```
0x8 = 1000
0x9 = 1001
0xA = 1010
0xB = 1011
```

`WD[31:0]` — это 32-битная шина данных, которые по сигналу синхроимпульса `CLK` будут записаны по адресу `A[31:0]` в том случае, если сигнал разрешения записи `WE` установлен в 1. Адрес для записи слова так же должен быть выровнен.

Несмотря на 32-битный адрес, разрабатываемая память должна поддерживать только часть адресного пространства, то есть меньше, чем 2^{30} байт = 4 ГБ. При обращении к отсутствующим ячейкам памяти выход `RD` должен выдавать ноль.

Задание: необходимо синтезировать память размером 64 слова, располагающиеся по адресам `0xNN000000` - `0xNN0000FC`, где `NN` — это две последних цифры номера студенческого билета. Сброс по сигналу reset в данной памяти не нужен!

В ПЛИС предусмотрена возможность инициализации имеющейся памяти подготовленными данными. Это значит, что при конфигурации ПЛИС в памяти будут размещены заранее известные значения, например, программа для процессора, или какие-нибудь полезные константы, или что-либо еще. Ниже представлен синтаксис инициализации памяти значениями из файла на Verilog HDL.

```
initial $readmemb ("file_name", reg_name);
```

В предложенном примере память, под названием `reg_name`, инициализируется (то есть задаются ее начальные значения) содержимым текстового файла `file_name`. При этом текстовая организация файла должна соответствовать организации памяти. То есть, если

```
reg [7:0] reg_name [0:2]
```

(три восьмибитных регистра), то файл должен выглядеть подобно рисунку справа.

```
01001011
11011001
10110100
```

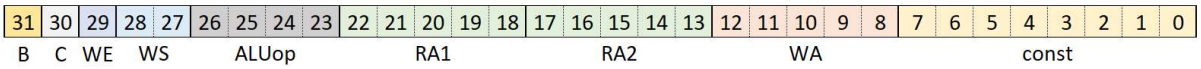
Второе задание

Хотя созданные модули относительно просты, тем не менее необходимо убедиться в правильности их функционирования. Для этого нужно написать `testbench` реализующий, на модели, последовательную автоматическую запись во все регистры RF некоторых случайных значений, а затем их считывание; убеждаясь в правильности записанного. В результате работы в терминал должны быть выведены сообщения: `(1)` об адресах регистров, в которые производится запись, `(2)` данные, которые были записаны, `(3)` считаны и `(4)` результат автоматического сравнения в конце в виде сообщения «`good`» или «`bad`», в случае успехов и ошибок, соответственно.

Третье задание

Теперь будем собирать примитивное программируемое устройство. Когда в нашем распоряжении есть АЛУ, регистровый файл и память это не составит труда.

В побайтово адресуемой памяти будет храниться программа, состоящая из инструкций. Каждая инструкция имеет разрядность 32 бита и кодируется следующим образом.



- `B` – выполнить безусловный переход;
- `C` – выполнить условный переход;
- `WE` – разрешение на запись в регистровый файл;
- `WS[1:0]` – источник данных для записи в регистровый файл (`0` – константа из инструкции, 1 – данные с переключателей, 2 – результат операции АЛУ;
- `ALUop[3:0]` – код операции, которую надо выполнить АЛУ;
- `RA1[4:0]` – адрес первого операнда для АЛУ;
- `RA2[4:0]` – адрес второго операнда для АЛУ;
- `WA[4:0]` – адрес регистра в регистровом файле, куда будет производиться запись;
- `const[7:0]` – 8-битное значение константы.

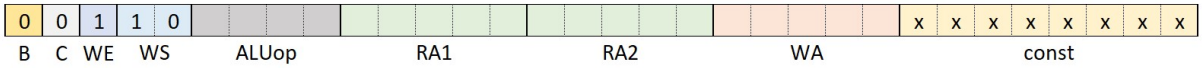
Предложенный способ кодирования инструкций позволяет реализовать пять типов операций: три – для работы с данными (определяется полем `WS`), и две – для

управления ходом выполнения программы (определяется полями B и C), которые могут повлиять на счетчик команд (PC — Program Counter, регистр хранящий адрес выполняемой инструкции). Рассмотрим особенности кодирования различных типов команд.

На представленных ниже примерах, если внутри поля находится «0» или «1», это значит, что для выполнения данной инструкции бит должен быть обязательно выставлен в соответствующее значение. Если внутри поля стоит «x», то его содержимое не имеет значения и никак не влияет на получаемый результат. Если поле пустое, значит оно используется в этой инструкции по прямому назначению.

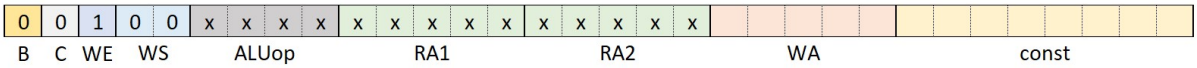
1. Инструкция **обработки данных на АЛУ**. В результате операции считывается два операнда из регистрового файла по адресам, указанным в полях RA1 и RA2. Между операндами выполняется операция, закодированная в поле ALUop, а результат помещается в регистровый файл по адресу WA.

   `reg[WA] ← reg[RA1] (`*ALUop*`) reg[RA2]`



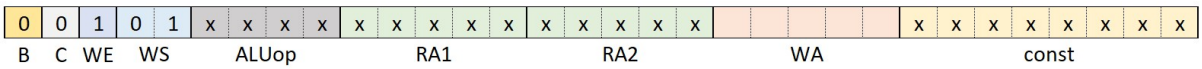
2. **Загрузка константы** из инструкции в регистровый файл по адресу WA.

   `reg[WA] ← const`



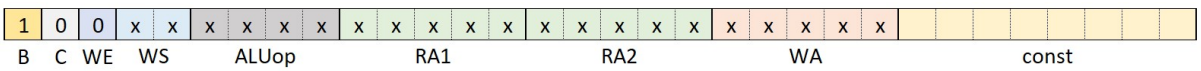
3. **Загрузка константы**, выставленной на переключателях (switches) в регистровый файл по адресу WA.

   `reg[WA] ← switches`



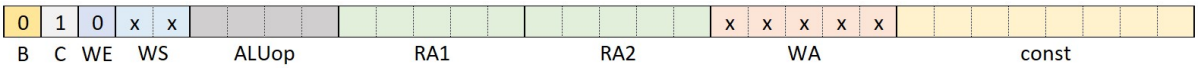
4. **Безусловный переход**. При выполнении этой инструкции к адресу выполняемой команды необходимо прибавить значение поля const, умноженное на 4 (потому, что побайтовая адресация).

   `PC ← PC + (const << 2)`

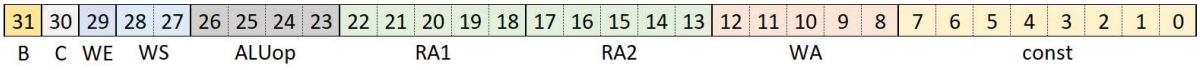


5. Инструкция **условного перехода**. В АЛУ выполняется сравнение двух чисел, хранящихся в регистровом файле по адресам RA1 и RA2. Способ сравнения кодируется в поле ALUop. Если в результате сравнения формируется флаг сравнения АЛУ, то к счетчику команд прибавляется число из поля const умноженное на 4. Если же флаг равен нулю, то PC увеличивается на 4, в таком случае говорят, что условный переход не произошел.

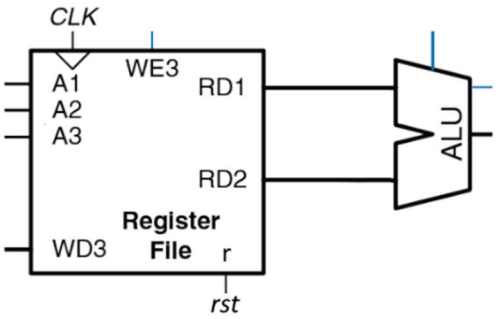
```
if (reg[RA1] (ALUop) reg[RA2]) then
      PC ← PC + (const << 2)
else
      PC ← PC + 4
```



Давайте теперь посмотрим, как нужно объединить имеющиеся функциональные блоки, чтобы поддерживать выполнение предложенных инструкций, другими словами – построим микроархитектуру устройства.



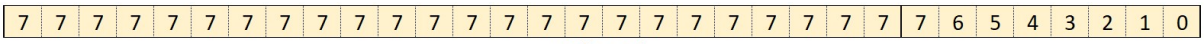
Биты [26:23] инструкции кодируют операцию ALUop для ранее разработанного АЛУ (придется незначительно поправить декодер). Адреса операндов, над которыми будет выполняться операция располагаются в полях RA1 и RA2 – биты [22:18] и [17:13] инструкции, соответственно. Чтобы операнды из регистрового файла поступали на входы АЛУ – соединим выходы портов чтения и входы АЛУ.



Адрес, по которому будет произведена запись в регистровый файл указывается в поле WA закодированного в битах [12:8] инструкции. Источник данных, которые будут записаны по адресу WA, определяется [28:27] битами инструкции – полем WS:

- WS == 00 – на вход WD3 регистрового файла, подается константа, закодированная в поле const [7:0] со знаковым расширением до 32-х бит (знакорасширение реализуется в модуле SE (Sign Extend), который

просто клонирует старший бит константы `const[7]` во все старшие биты `[31:8]` 32-х битного выхода, с целью сохранения знака)

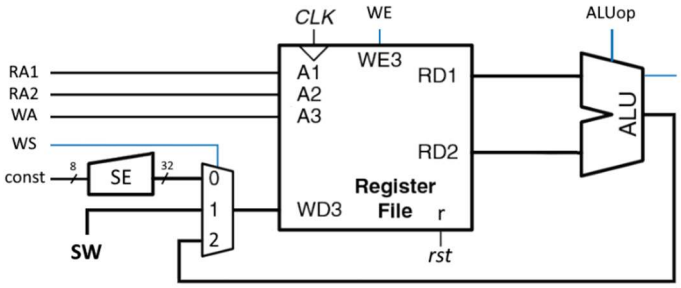


SE(const)

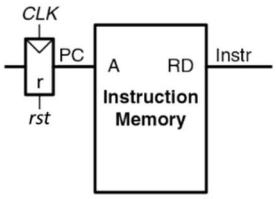
- `WS == 01` – значение с переключателей (`switches`)
- `WS == 10` – результат операции из АЛУ

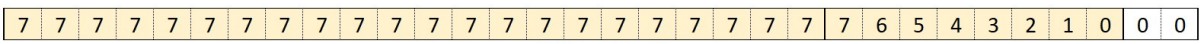
Для выбора источника данных, для записи в регистровый файл, потребуется разместить на входе `WD3` мультиплексор, управляемый сигналом `WS` из инструкции, и мультиплексирующий: знакорасширеное значение константы, сигналы с переключателей (`SW – switches`) и результат операции АЛУ.

Так как значения адресов операндов (`RA1`, `RA2` и `WA`), сигнал разрешения записи в регистровый файл (`WE`) и код операции для АЛУ (`ALUop`) передаются в инструкции в непосредственном виде, то их можно сразу подключать к перечисленным входам.



Любое программируемое устройство обладает специальным регистром для хранения адреса выполняемой инструкции – PC (`Program Counter`), который указывает на адрес текущей выполняемой инструкции в памяти (в нашем случае с побайтовой адресацией). Так же в нем должна быть предусмотрена логика изменения значения PC после выполнения очередной инструкции. В разрабатываемом устройстве это реализуется с помощью управляющих кодов B и C, которые определяют, что запишется в PC: `PC + 4` или `PC + (const << 2)`.



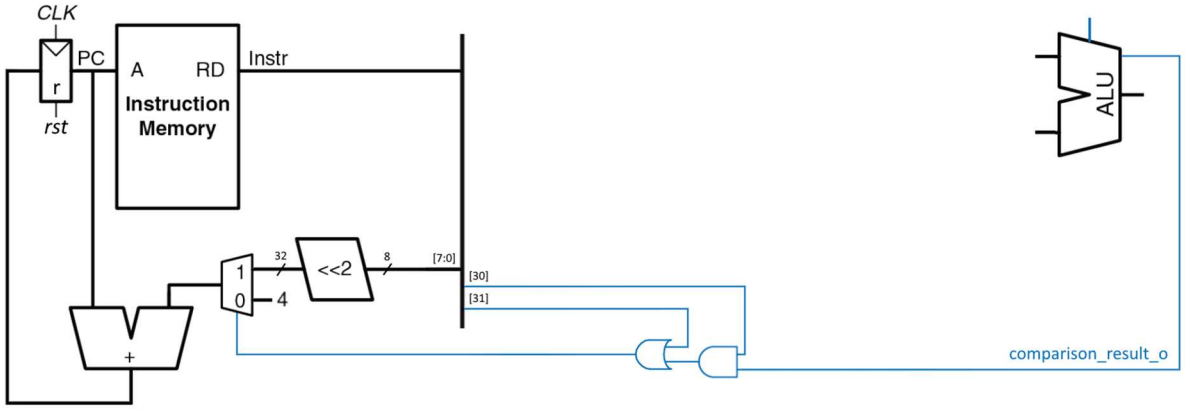


SE(const << 2)

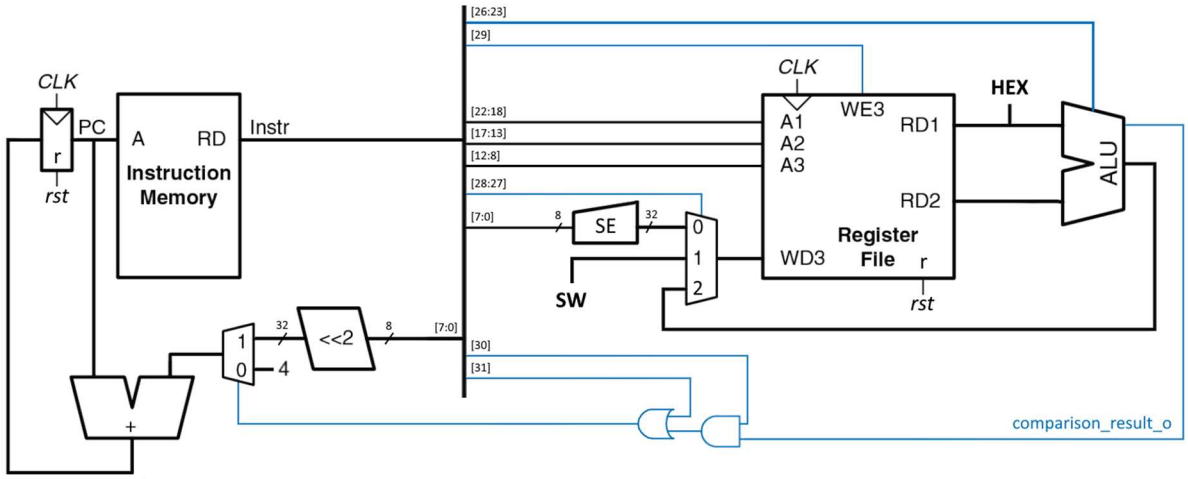
- Если поля B и C равны нулю, то значение PC по фронту синхроимпульса CLK должно увеличиваться на 4, то есть `PC = PC + 4`.
- Если поле `B == 1`, то `PC = PC + (const << 2)` – безусловный переход.

- Если поле `C == 1` (при этом `B == 0`), то, если флаг сравнения, формируемый АЛУ равен нулю (`comparison_result_o == 0`), то `PC = PC + 4`, в противном случае `PC = PC + (const << 2)`.

Чтобы реализовать поддержку описанного механизма, необходимо добавить к устройству сумматор, на один вход которого подается значение из PC, а на втором мультиплексируются: константа из инструкции и константное 4. Мультиплексор должен пропускать SE(`const << 2`) или когда `B == 1`, или когда `C == 1` и `comparison_result_o == 1`. Очевидно, выход сумматора должен сохраняться в PC, поэтому подключается к его входу.



Итоговая микроархитектура разрабатываемого устройства, которую требуется реализовать, представлена ниже. На рисунке также отражено подключение первого порта чтения к семисегментным индикаторам (HEX). Входы синхронного сброса `rst` необходимо подключить к одной из кнопок на отладочной плате, с целью перезапуска устройства.



Четвертое задание

Необходимо написать программу, реализующую выполнение алгоритма, предложенного преподавателем. В качестве входных данных будут использоваться значения с переключателей (`switches`) и константы, передаваемые внутри инструкций. Дойдя до конца, программа должна останавливаться, а адресуемое значение первого порта на чтение должно указывать на итоговый результат, в таком случае он высветится на семисегментных индикаторах. Чтобы остановить программу, достаточно выполнить инструкцию безусловного перехода со значением `const = 0`. В таком случае на каждом такте PC будет присваивать `PC + 0`.

Сама программа представляется файлом, состоящим из `0` и 1, записанных строками по 32 знака. Каждая строка — инструкция, а весь файл – программа. Файл нужен для инициализации памяти инструкций (механизм инициализации памяти упоминался выше).

Пример

Допустим необходимо определять произведение числа 13 на заданное (на переключателях). Результат выводить на семисегментные дисплеи.

Известно, что произведение – это многократное сложение. Будем складывать число 13 несколько раз по кругу. Количество кругов задается переключателями, сохраним это число в регистр и на каждой итерации будем добавлять 13 к результату, и вычитать по одному из количества кругов, пока количество кругов не станет равным нулю. Напишем следующую программу.

```
reg[1] ← 13          //размещаем в регистре эталонное число 13
reg[2] ← switches    //запоминаем сколько сделать повторений
reg[3] ← 1           //чтобы уменьшать количество оставшихся кругов
if (reg[2] == reg[0]) PC ← PC + (4 * 4)//если осталось 0 кругов, то в конец
reg[4] ← reg[4] + reg[1]  //прибавить 13 к итоговому произведению
reg[2] ← reg[2] – reg[3]  //уменьшить на 1 количество оставшихся кругов
PC ← PC + (–3 * 4)  //вернуться на 3 инструкции назад
PC ← PC + (0 * 4)   //остановить программу и выводить ответ
```

Ниже представлена эта же программа, но записанная в виде последовательности машинных инструкций. Некоторые биты помечены как «x», это значит, что не имеет значения, что именно в них записано. Это сделано для удобства визуального восприятия, в файле и памяти эти биты должны быть чему-то равны, 0 или 1.

	B	C	WE	WS		ALUop				RA1					RA2					WA					const							
0x00	0	0	1	0	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	0	0	0	0	1	0	0	0	0	1	1	0	1
0x04	0	0	1	0	1	x	x	x	x	x	x	x	x	x	x	x	x	x	x	0	0	0	1	0	x	x	x	x	x	x	x	x
0x08	0	0	1	0	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	0	0	0	1	1	0	0	0	0	0	0	0	1
0x0C	0	1	0	x	x	1	1	0	0	0	0	0	1	0	0	0	0	0	0	x	x	x	x	x	0	0	0	0	0	1	0	0
0x10	0	0	1	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	1	0	0	x	x	x	x	x	x	x	x
0x14	0	0	1	1	0	0	0	0	1	0	0	0	1	0	0	0	0	1	1	0	0	0	1	0	x	x	x	x	x	x	x	x
0x18	1	0	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	1	1	1	1	1	1	0	1
0x1C	1	0	0	x	x	x	x	x	x	0	0	1	0	0	x	x	x	x	x	x	x	x	x	x	0	0	0	0	0	0	0	0

В последней инструкции обязательно необходимо выставить адрес регистра RA1, в котором хранится результат расчета.

Проект с программой необходимо запустить либо на отладочном стенде, либо смоделировать его работу на компьютере, убедиться в правильной работоспособности, после чего сдать преподавателю.