# *Работа в аудитории.*

## Краткое руководство по использованию Keil ARM Simulator

1. Запустите Keil ARM Simulator. У нас версия µVision V5.22.0.0.
2. Кликните на меню **Project**, выберите пункт **New Microvision Project.** Обратите внимание, что ярко-синий шрифт указывает на ввод в и из компьютера.
3. Введите имя файла в поле **File name**. Скажем, **`MyFirstExample.`**
4. Нажмите на кнопку **Save**.
5. Это вызовет появление окна **Select Device for Target 'Target 1'**. Теперь вы должны выбрать семейство и версию процессора, который вы собираетесь использовать.
6. В списке устройств выберите ARM, а затем из нового списка выберите ARM7 (Big Endian).
7. Нажмите на кнопку OK. Окно исчезнет. Вы вернетесь к главному окну µVision.
8. Нам необходимо войти в исходную программу. Нажмите **File**. Выберите **New** и нажмите на нее. Появится окно редактирования Text1. Теперь мы можем ввести простую программу. Например:

```
        AREA MyFirstExample, CODE, READONLY
        ENTRY
        MOV   r0,#0x4      ; load 4 into r0
        MOV   r1,#0x5      ; load 5 into r1
        ADD   r2,r0,0xe1   ; add r0 to r1 and put the result in r2
S       B     S            ; force infinite loop by branching to this line
        END                ; end of program
```

9. После набора программы выберите **File**, а затем **Save** в меню. Система запросит ввести **File name**. Используйте имя **`MyFirstExample.s`**. Суффикс `.s` указывает на исходный код.
10. После этого система вернет вас к окну, которое теперь будет называться **MyFirstExample.**
11. Теперь вы должны настроить окружающую среду. Нажмите **Project** в главном меню. Из выпадающего списка выберите **Manage**. Это в свою очередь послужит появлению нового списка. **Select Components, Environment, Books..**
12. Теперь вы получите форму с тремя окнами. Внизу правого окна выберите **Add Files**.
13. Появляется обычное окно обзора файлов Windows. Нажмите на стрелку выбора типа файла (расширение файла) и выберите тип файла **Asm Source File (* .s *; * .src; * .a *).** Теперь вы должны увидеть ваш собственный файл **`MyFirstExample.s`** в окне. Выберите его и перейдите на вкладку Add. Это добавляет исходный файл в проект. Затем нажмите на Close. Теперь вы увидите файл `MyFirstExample.s` в крайнем правом окне. Нажмите кнопку OK, чтобы выйти.
14. Вот и все. Вы готовы ассемблировать ваш файл.
15. Нажмите **Project** в главном меню и нажмите на **Built target**.
16. В нижнем окне **Build Output** вы увидите результат процесса ассемблирования.
17. Вы должны увидеть что-то вроде:

```
*** Using Compiler 'V5.06 update 4 (build 422)', folder: 'G:\Keil_v5\ARM\ARMCC\Bin'
Build target 'Target 1'
assembling MyFirstExample.s...
linking...
Program Size: Code=16 RO-data=0 RW-data=0 ZI-data=0
".\Objects\MyFirstExample.axf" - 0 Error(s), 0 Warning(s).
Build Time Elapsed:  00:00:01
```

18. Магическая фраза: **“0 Error(s)”**. Если вы не получаете такой фразы, вы должны обнаружить ошибку в исходном файле и снова повторить действия по запуску ассемблирования: нажать на **Project** и **Build target**.

**Пример 1. Сложение**
Задача: Вычислить P = Q + R + S
Допустим Q = 2, R = 4, S = 5. Предположим, что r1 = Q, r2 = R, r3 = S. Результат P сохранить в регистре r0.

Фрагмент кода:

```
ADD r0,r1,r2 ; сложить Q и R, результат сохранить в P
ADD r0,r0,r3 ; сложить S и P, результат сохранить в P
```

Программа:

```
     AREA Example1, CODE, READONLY
     ADD r0,r1,r2
     ADD r0,r3
Stop B Stop
     END
```

**Примечание:**

1. Точка с запятой указывает на введенный пользователем комментарий. Все, что следует после точки с запятой в строке игнорируется ассемблером.
2. Первая строка `AREA Example1, CODE, READONLY` является директивой ассемблера и требуется для запуска программы. Это является особенностью системы разработки, а не языка ассемблер ARM. Ассемблеры различных компаний могут иметь различные директивы для определения начала программы. В нашем случае, `AREA` ссылается на сегмент кода, `Example1` – это имя, которое мы присвоили ему, `CODE` указывает скорее на начало исполнимого кода, чем на область данных и `READONLY` указывает, что это не модифицируемая часть кода.
3. Все, что находится в первом столбце кода (в нашем случае Stop) представляет собой метку, которая может использоваться для ссылки на эту строку.
4. Команда `Stop B Stop` означает “ветвление на строку с меткой `Stop`” и используется для создания бесконечного цикла. Это удобный способ прекращения программ в простых примерах, как этот.
5. Последняя строка `END` является директивой ассемблера, которая указывает ассемблеру, что нет больше кода, который следует выполнять. чтобы следовать. Этой директивой заканчивается программа.

На рис. Example 1.1 показано содержимое экрана после загрузки программы.

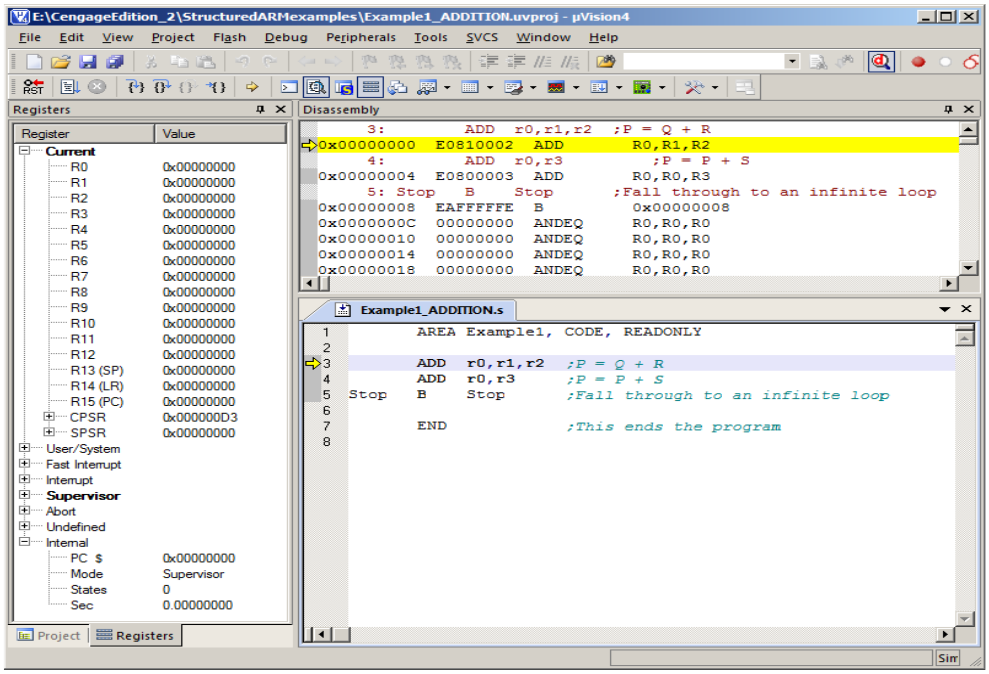


Рис. Example 1.1. Состояние системы после загрузки кода программы Example 1.

Из-за того, что отсутствуют средства ввода исходных данных в регистры, вы должны сделать это вручную. Просто дважды щелкните по нужному регистру, а затем изменить его значение.

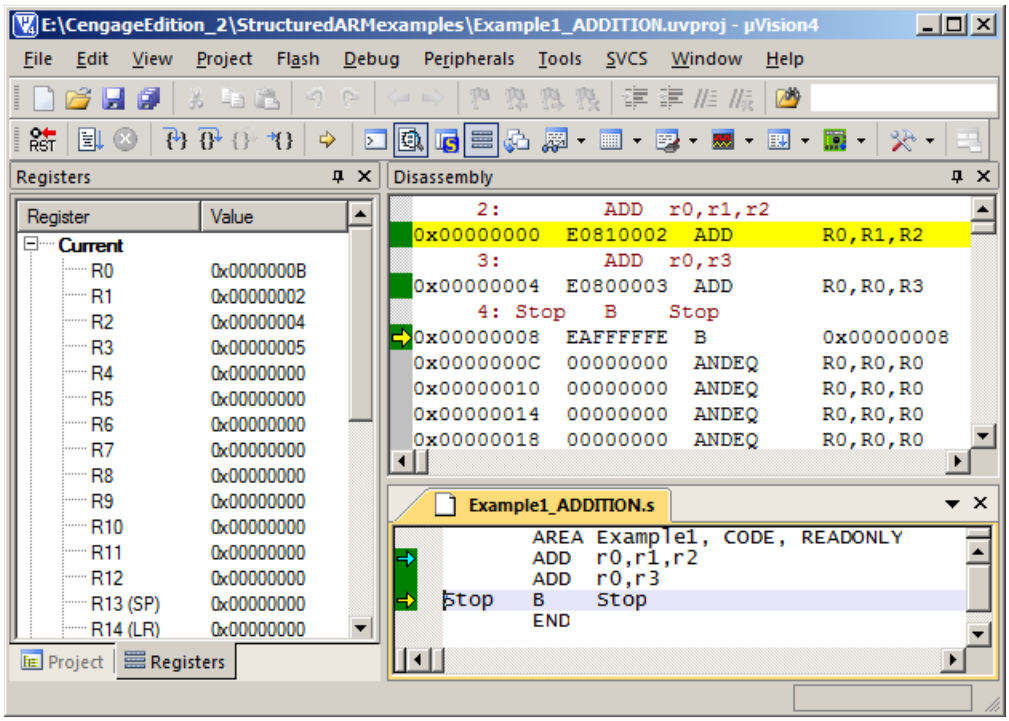


Рис. Example 1.2. Состояние системы после запуска кода.

Обратите внимание, что содержимое r0 равно 2 + 4 + 5 = 11 = 0x0B. Это результат, который мы ожидали.

**Пример 2. Сложение**

Задача такая же, что и в примере 1: Вычислить P = Q + R + S

Снова предположим, что Q = 2, R = 4, S = 5 и пусть r1 = Q, r2 = R, r3 = S. В этом случае, перед запуском программы мы разместим данные в памяти в виде констант.

Фрагмент кода:

```
MOV r1,#Q          ; load Q into r1
MOV r2,#R          ; load R into r2
MOV r3,#S          ; load S into r3
ADD r0,r1,r2       ; Add Q to R
ADD r0,r0,r3       ; Add S to (Q + R)
```

Здесь мы используем инструкцию `MOV`, которая копирует значение в регистр. Значение может быть содержимым другого регистра или литералом. Литерал обозначается символом #. Мы можем написать, например, `MOV r7, r0; MOV r1, # 25` или `MOV r5, #Second.`
Мы использовали символьные имена Q, R и S. Мы должны связать эти имена с фактическими значениями. Это можно сделать с помощью директивы ассемблера `EQU` (приравнять); например,

```
Q      EQU   2
```

Эта директива соотносит символьное имя `Q` к значению 2. Если программист использует `Q` в выражении, это точно так же, как написание 2. Цель использования `Q`, а не 2, чтобы сделать программу более читаемой.

Программа:

```
      AREA Example2, CODE, READONLY
      MOV r1,#Q          ;load r1 with the constant Q
      MOV r2,#R
      MOV r3,#S
      ADD r0,r1,r2
      ADD r0,r0,r3
Stop  B Stop

Q     EQU 2              ; Equate the symbolic name Q to the value 2
R     EQU 4              ;
S     EQU 5              ;
      END
```

На рис. Example 2.1 состояние системы после загрузки кода. Если посмотреть на окно дисассемблирования, то можно увидеть, что константы замещены их действительными значениями.

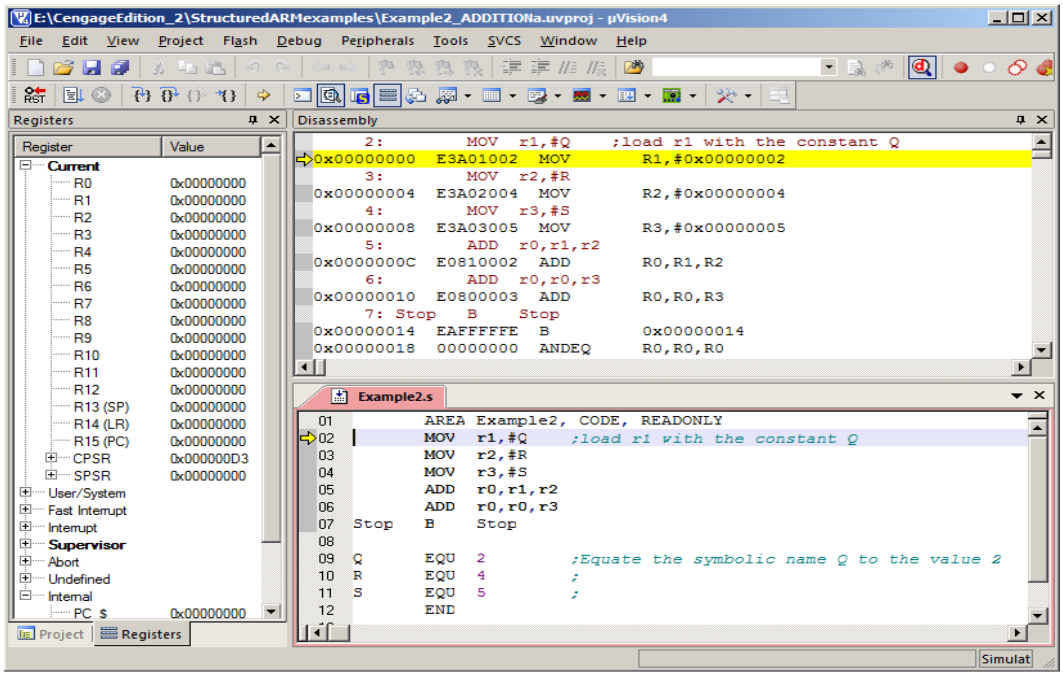

Рис. Example 2.1. Состояние системы после загрузки кода.

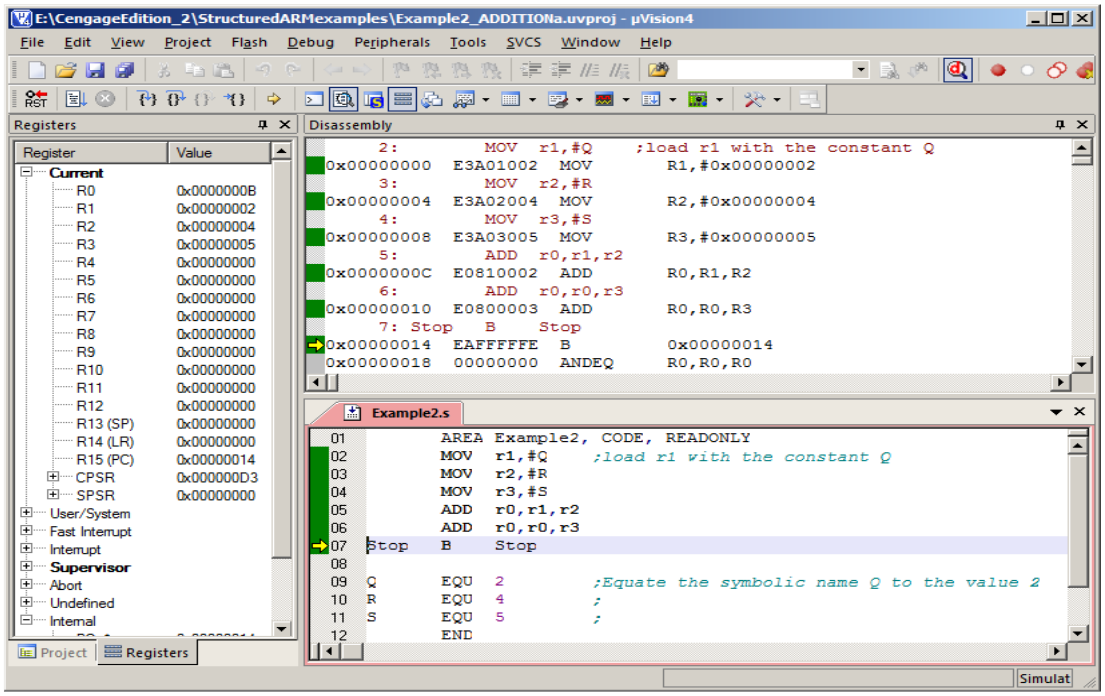


Рис. Example 2.2. Состояние системы после запуска кода.

**Пример 3. Сложение**

Задача остается прежней: Вычислить P = Q + R + S. Как и прежде, Q = 2, R = 4, S = 5 и пусть r1 = Q, r2 = R, r3 = S.

В этом случае, перед запуском программы мы разместим данные в памяти в виде констант. Сначала мы используем команду загрузки в регистр, `LDR r1,Q`, чтобы загрузить регистр r1 содержимым ячейки памяти, на которую ссылается Q. Такой команды вообще-то не существует, и она отсутствует в наборе команд ARM. Но ассемблер ARM автоматически изменяет его на реальную инструкцию.

Команда `LDR r1,Q` называется *псевдоинструкцией*. Но это облегчает жизнь программиста.

Фрагмент кода:

```
        LDR r1,Q          ; load r1 with Q
        LDR r2,R          ; load r2 with R
        LDR r3,S          ;load r3 with S
        ADD r0,r1,r2      ;add Q to R
        ADD r0,r0,r3      ;add in S
        STR r0,Q          ;store result in Q
```

Программа:

```
        AREA Example3, CODE, READONLY
        LDR r1,Q          ;load r1 with Q
        LDR r2,R          ;load r2 with R
        LDR r3,S          ;load r3 with S
        ADD r0,r1,r2      ;add Q to R
        ADD r0,r3         ;add in S
        STR r0,Q          ;store result in Q
Stop    B Stop

        AREA Example3, CODE, READWRITE
P       SPACE 4           ; save one word of storage
Q       DCD 2             ; create variable Q with initial value 2
R       DCD 4             ; create variable R with initial value 4
S       DCD 5             ; create variable S with initial value 5
        END
```

Обратите внимание, как мы должны создать область данных в конце программы. Мы зарезервировали места для P, Q, R и S. Использовали директиву `SPACE`, чтобы зарезервировать 4 байта памяти для переменной S. После этого мы резервируем область памяти для Q, R и S. В каждом случае мы используем директиву ассемблера `DCD`, чтобы зарезервировать слово (4 байта) и инициализировать его. Например,

```
Q       DCD    2              ; create variable Q with initial value 2
```

означает "вызвать текущую строку Q и сохранить слово 0x00000002 в этом месте.
На рисунке Example 3.1 показано состояние программы после того, как она была загружена. В данном случае мы использовали команду обзора памяти (view memory), чтобы показать пространство памяти. Мы выделили три константы, которые были предварительно загружены в память.
Если взглянуть на дисассемблированный код, то можно увидеть, что псевдоинструкция `LDR r1,Q` фактически переводится в реальную инструкцию ARM `LDR r1, [PC, # 0x0018]`. Это по-прежнему команда загрузки, но режим адресации в этом случае косвенно-регистровый (register indirect). Адрес вычисляется сложением содержимого счетчика команд программы (program counter PC) и шестнадцатеричного смещения 0x18. Отметим также, что счетчик команд всегда на 8 байт больше адреса текущей инструкции. Это особенность конвейера ARM процессора. Следовательно, адрес операнда равен: [PC] + 0x18 + 8 = 0 + 18 + 8 = 0x20.
Если вы посмотрите на область отображения памяти, вы увидите, что содержимое ячейки 0x20 действительно 0x00000002.

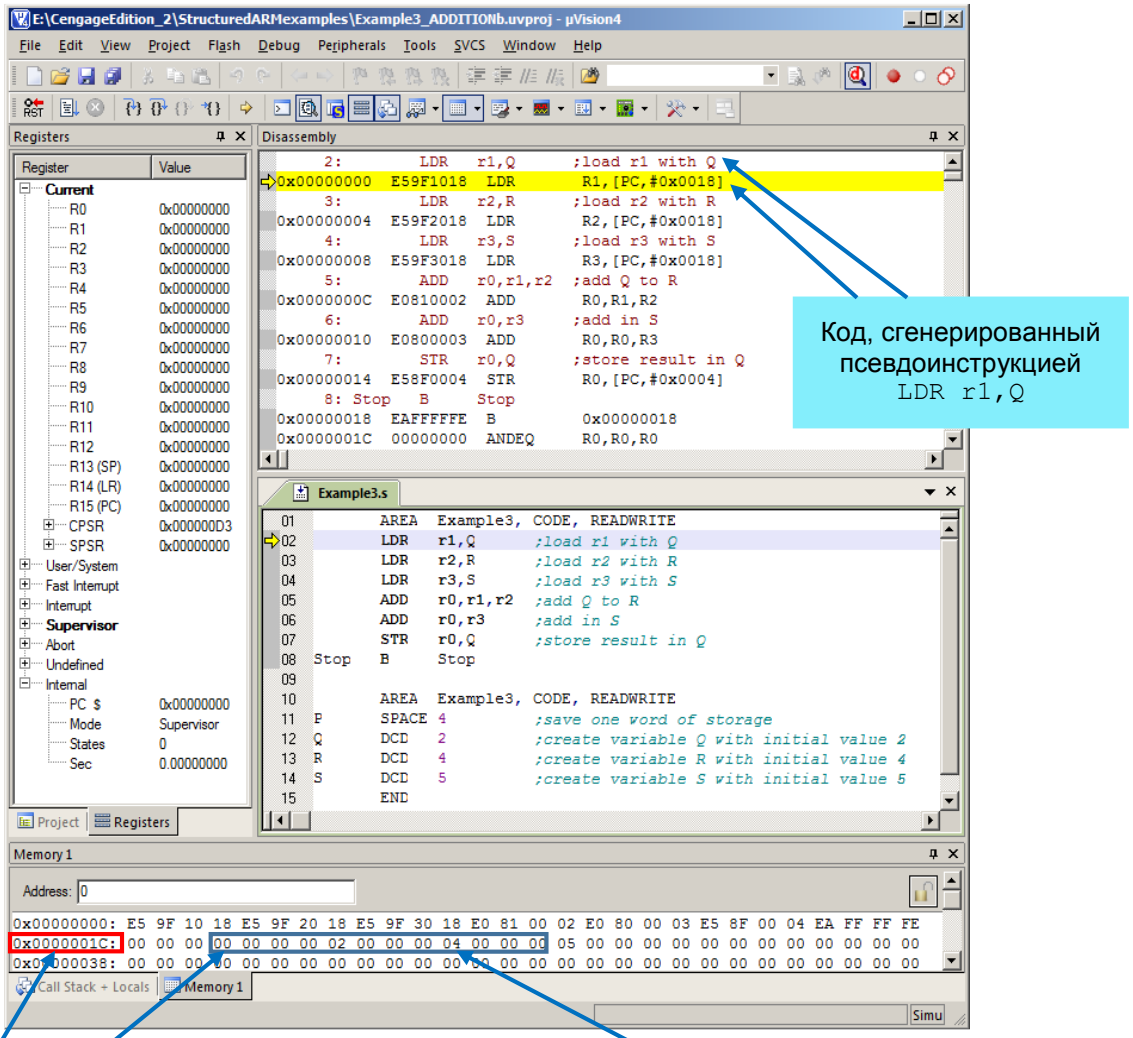


**Рис. Example 3.1** Состояние системы после загрузки программы

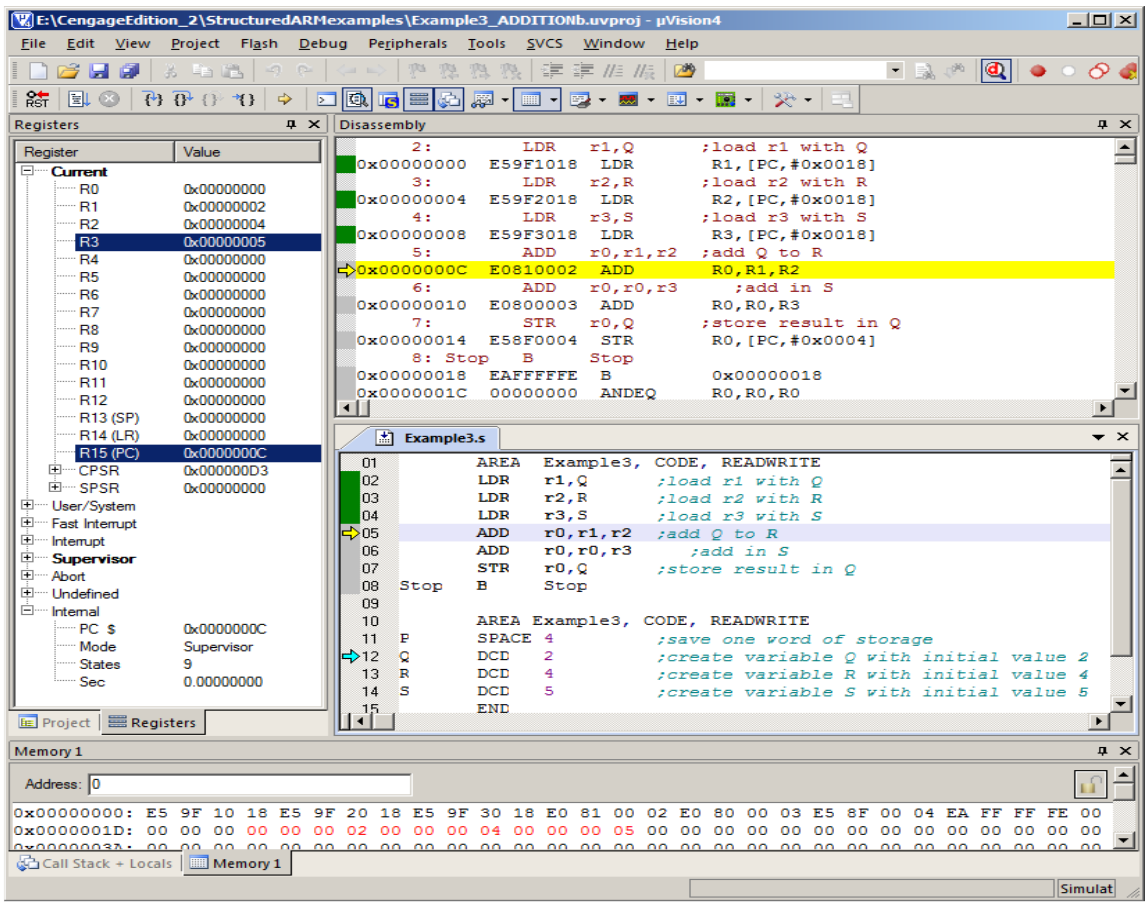


**Рис. Example 3.2**

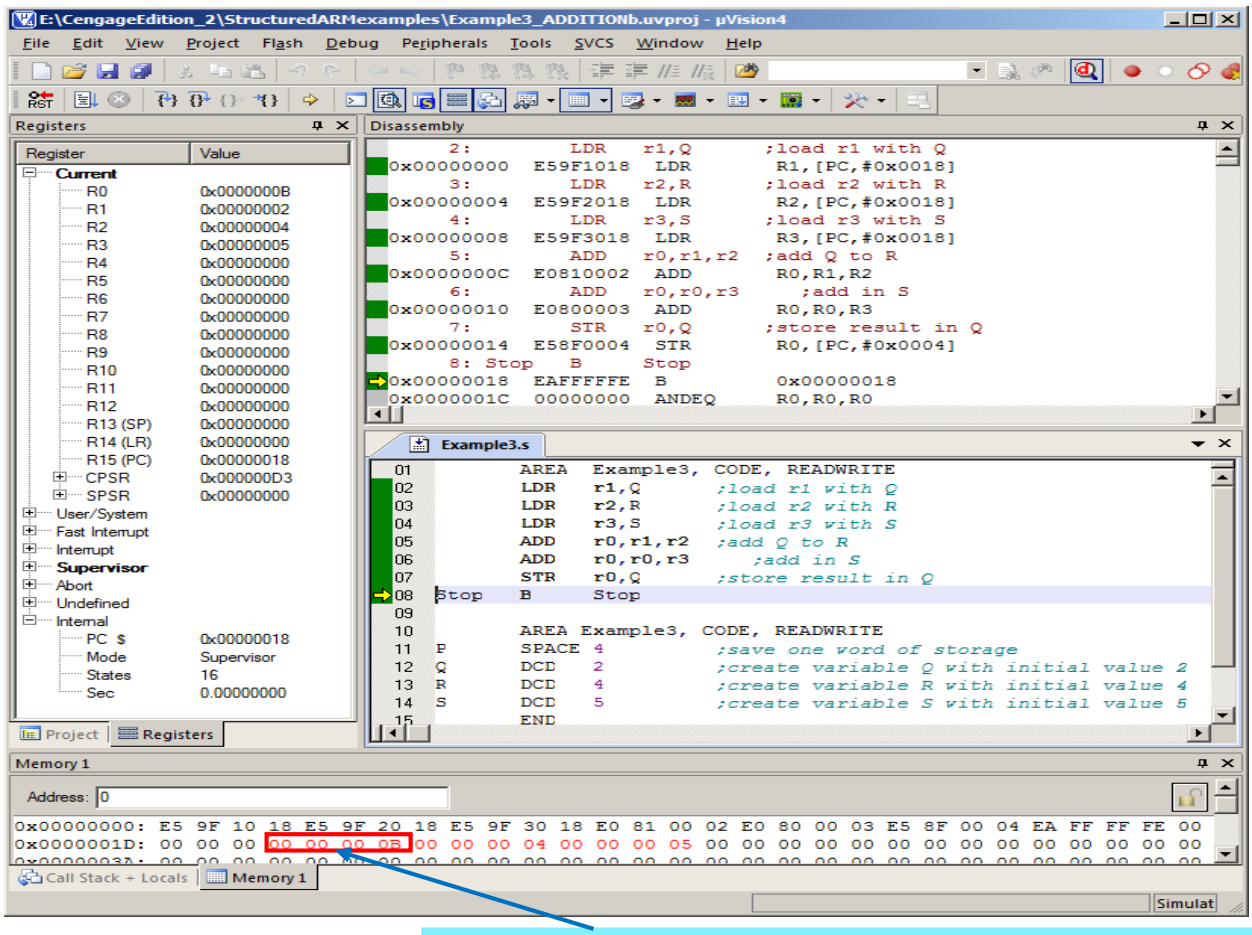


**Рис. Example 3.3**

**Пример 4. Сложение**

Задача:

Вычислить P = Q + R + S, при этом Q = 2, R = 4, S = 5. В этом случае мы используем ***регистровую косвенную адресацию (register indirect addressing)*** для доступа к переменным. Иначе говоря, нам необходимо установить указатель на переменные и иметь доступ к ним через этот указатель.

Фрагмент кода:

```
            ADR   r4,TheData  ; r4 points to the data area
            LDR   r1,[r4,#Q]  ; load Q into r1
            LDR   r2,[r4,#R]  ; load R into r2
            LDR   r3,[r4,#S]  ; load S into r3
            ADD   r0,r1,r2    ; add Q and R
            ADD   r0,r0,r3    ; add S to the total
            STR   r0,[r4,#P]  ; save the result in memory
```

Программа:

```
            AREA Example4, CODE, READWRITE
            ENTRY
            ADR   r4,TheData  ; r4 points to the data area
            LDR   r1,[r4,#Q]  ; load Q into r1
            LDR   r2,[r4,#R]  ; load R into r2
            LDR   r3,[r4,#S]  ; load S into r3
            ADD   r0,r1,r2    ; add Q and R
            ADD   r0,r0,r3    ; add S to the total
            STR   r0,[r4,#P]  ; save the result in memory
Stop        B     Stop

P           EQU   0           ; offset for P
Q           EQU   4           ; offset for Q
R           EQU   8           ; offset for R
S           EQU   12          ; offset for S

            AREA Example4, CODE, READWRITE
TheData     SPACE 4           ; save one word of storage for P
            DCD   2           ; create variable Q with initial value 2
            DCD   4           ; create variable R with initial value 4
            DCD   5           ; create variable S with initial value 5
            END
```

На рис. Example 4.1 показано состояние системы после того, как загружена программа. Следует признать, что такое написание кода не очень эффективно, слишком растянут код, но тем не менее, эта программа иллюстрирует несколько концепций.

Во-первых, инструкция `ADR r4,TheData` загружает адрес начала области данных (метка TheData) в регистр `r4`. То есть, `r4` указывает на начало области данных. Если посмотреть на код, то видно, что для P зарезервировано четыре байта, а затем были загружены значения для Q, R и S в последовательные ячейки памяти пословно. Обратите внимание, что ни одна из этих ячеек не имеет метку.

Инструкция `ADR` (load an address into a register - загрузка адреса в регистр) является псевдоинструкцией. Если посмотреть на дисассемблированный код на рис. Example 4.1, то можно увидеть, что эта инструкция транслирована в команду ADD r4, PC, # 0x18. Вместо загрузки фактического адреса метки TheData в `r4` загружается содержимое счетчика команд PC плюс смещение. Программисты могут не беспокоиться о том, как ARM собирается перевести команду ADR в реальный код. В этом и заключается прелесть псевдоинструкции, ассемблер сам это сделает правильным образом.

Когда необходимо Q загрузить в r1, используется команда `LDR r1,[r4,#Q]`. Это ARM команда загрузки регистра литеральным смещением (literal offset), т.е. значением Q. Если вы посмотрите на EQU области, то Q приравнивается к 4 и, следовательно, регистр r1 загружается значением данных, которое находится в 4 байтах от ячейки, куда указывает r4. Эта адрес ячейки памяти, где хранятся данные, соответствующие Q.

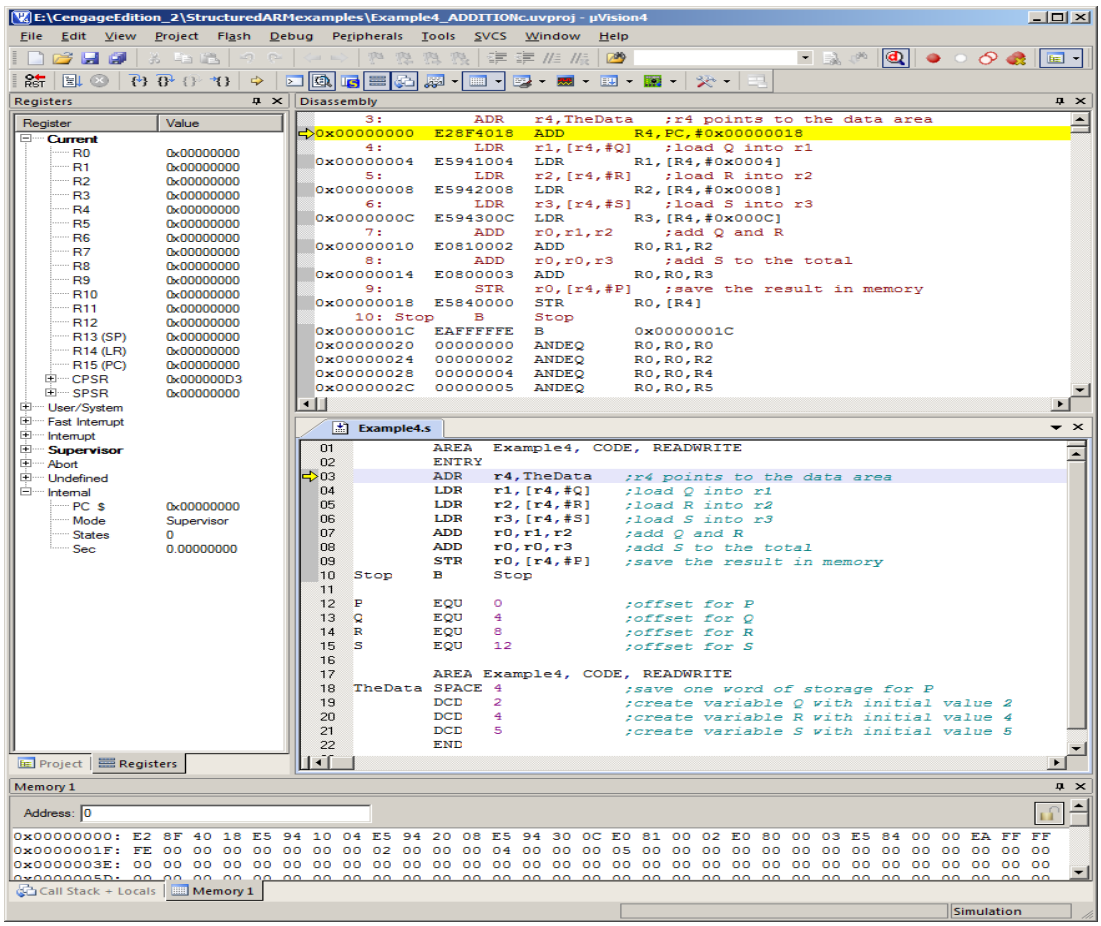


**Рис. Example 4.1** Состояние системы после загрузки программы

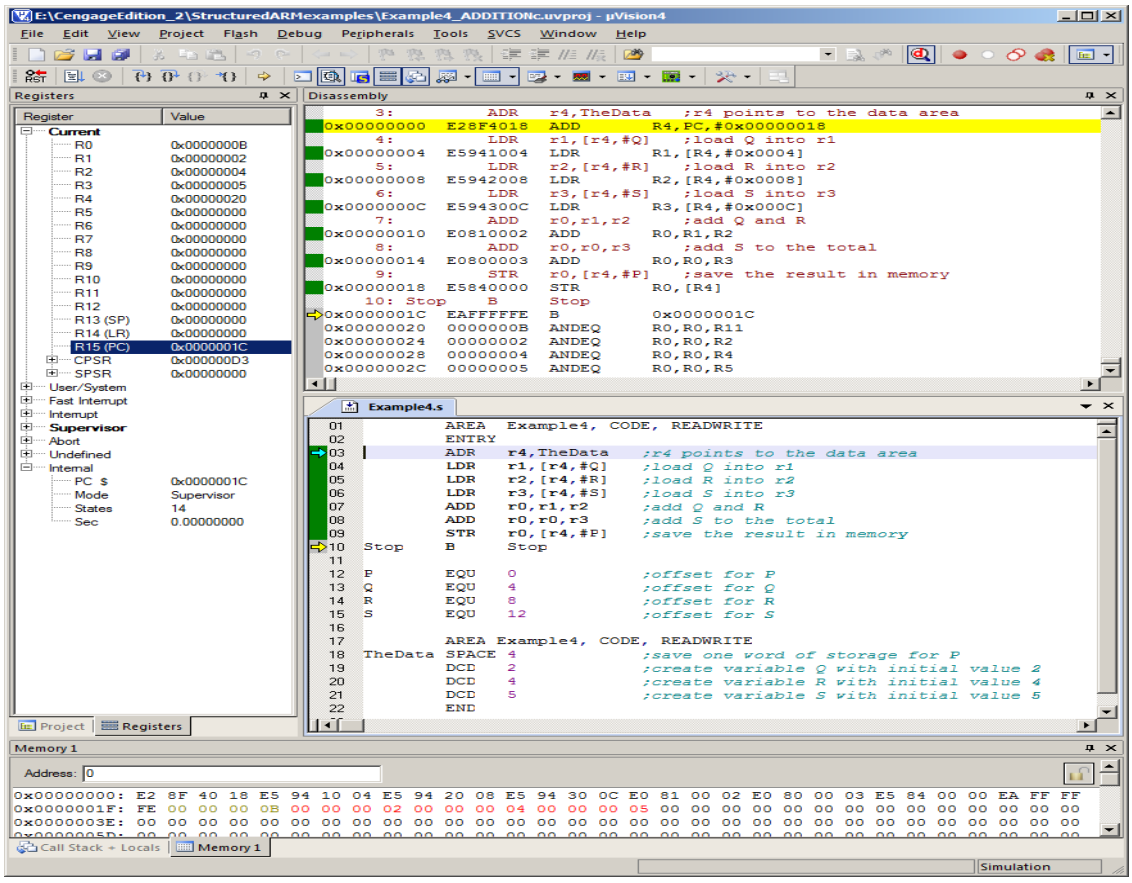


**Рис. Example 4.2** Состояние системы после выполнения программы

**Пример 5. Сложение**

Повторим тот же самый пример еще раз. Но на этот раз напишем программу в более компактном виде.

Для упрощения кода, мы использовали простые числовые смещения (поскольку имелось относительно малое количество данных и комментарии подсказывали нам, какие действия выполняются что на самом деле). Обратите внимание, мы использовали метки Q, R и S для данных. Эти метки являются избыточными и нет необходимости в них, так как они не упоминаются нигде в программе. Но в этом нет ошибки. Эти метки просто служат напоминанием для программиста.

```
            AREA Example5, CODE, READWRITE
            ENTRY
            ADR r0,P            ; r4 points to the data area
            LDR r1,[r0,#4]      ; load Q into r1
            LDR r2,[r0,#8]      ; load R into r2
            ADD r2,r1,r2        ; add Q and R
            LDR r1,[r0,#12]     ; load S into r3
            ADD r2,r2,r1        ; add S to the total
            STR r1,[r2]         ; save the result in memory
Stop        B Stop

            AREA Example5, CODE, READWRITE
P           SPACE 4             ; save one word of storage for P
Q           DCD 2               ; create variable Q with initial value 2
R           DCD 4               ; create variable R with initial value 4
S           DCD 5               ; create variable S with initial value 5
            END
```

Отметим также, что мы повторно использовали регистры, чтобы избежать использования большего количества регистров. В этом примере используется только r0, r1 и r2. Если после использования регистра, его значение не больше не участвует в программе, то он может быть использован повторно. Однако, это может сделать отладку сложнее. В этом примере в одном месте программы регистр r1 содержит Q и в другой месте - S. И, наконец, он содержит результат S.

На рис. Example 5.1 приводится состояние системы после загрузки программы, а на рис. Example 5.2 показано состояние системы после выполнения программы.

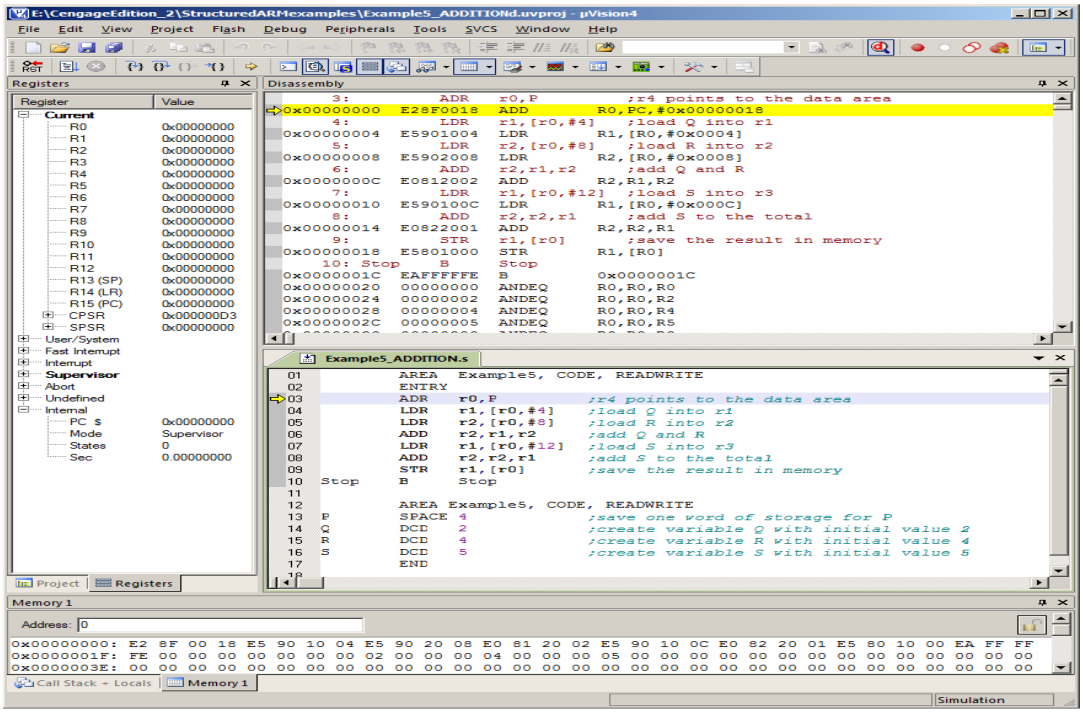


Рис. Example 5.1. Состояние системы перед выполнением программы.

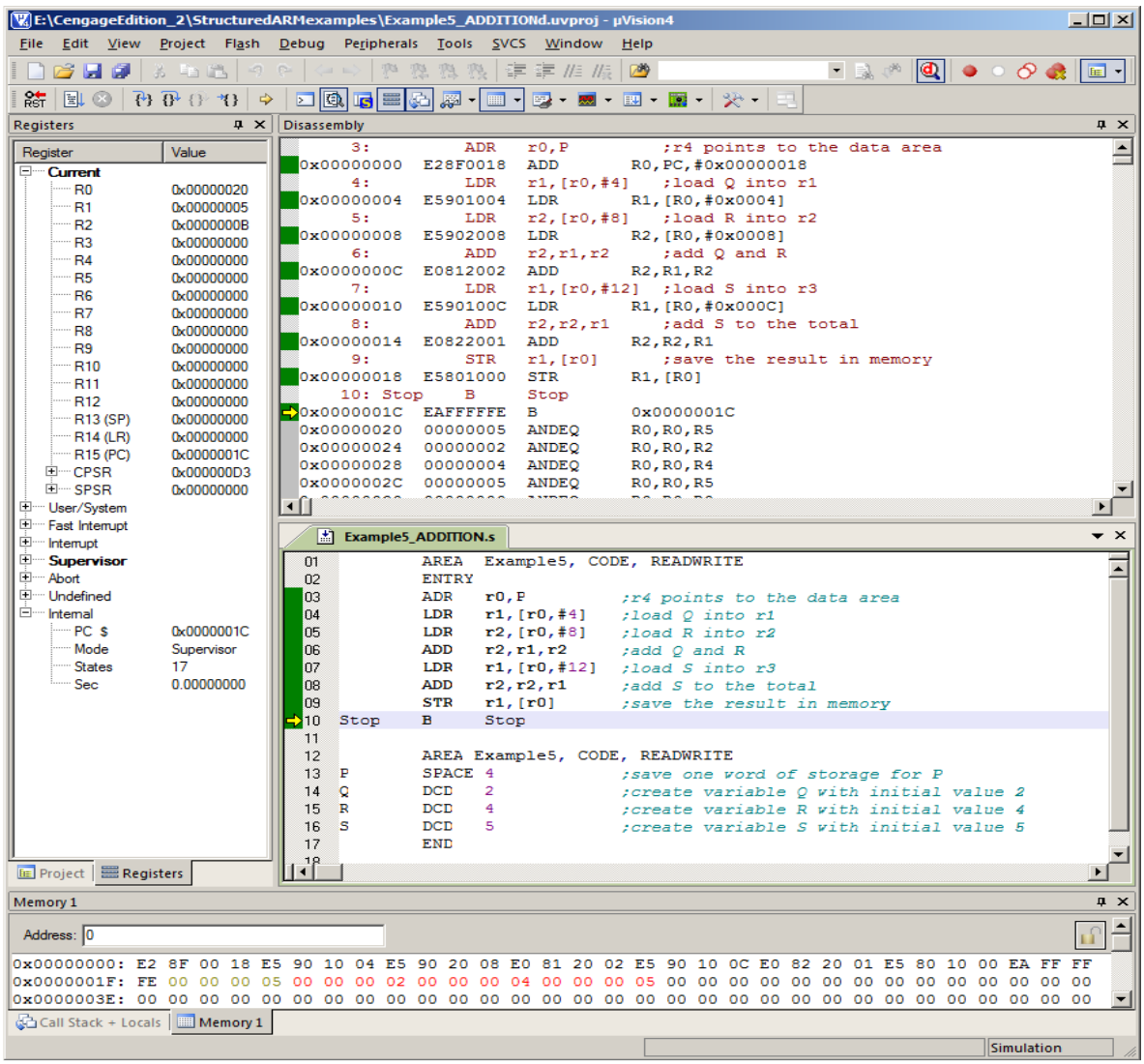


Рис. Example 5.2. Состояние системы после выполнения программы.

**Примеры 1 – 5. Выводы.**
Программы, использующие Keil ARM IDE, начинаются со строки

```
AREA nameOfProg, CODE, READONLY
```

и заканчиваются

```
END.
```

- Данные можно хранить в памяти до запуска программы, используя директиву DCD (определить константу).
- Можно написать ADD r0, r1, # 4 или ADD r0, r1, K1. Однако, если вы используете символическое имя (например, K1), то вам необходимо использовать оператор EQU, чтобы приравнять его (связать его) к действительному значению.
- Некоторые инструкции являются псевдоинструкциями. Они не являются инструкциями ARM, но они являются краткой формой записи команды, которая автоматически транслируется в одну или более инструкций ARM.
- Инструкция MOV r1, r2 или MOV r1, #literal имеет два операнда и перемещает содержимое регистра или литерала в регистр.

## Пример 6. Арифметические выражения

Задача:

Вычислить значение арифметического выражения **(A + 8B + 7C – 27) / 4,** где A=25, B=19 и C=99.

В этом примере используются литералы.

Фрагмент кода:

```
MOV r0,#25              ; Load register r0 with A which is 25
MOV r1,#19              ; Load register r1 with B which is 19
ADD r0,r0,r1,LSL #3     ; Add 8 x B to A in r0
MOV r1,#99              ; Load register r1 with C which is 99 (reuse of r1)
MOV r2,#7               ; Load register r2 with 7
MLA r0,r1,r2,r0         ; Add 7 x C to total in r0
SUB r0,r0,#27           ; Subtract 27 from the total
MOV r0,ASR #2           ; Divide the total by 4
```

Необходимо отметить несколько вещей.

Во-первых, Умножение или деление на число в степени 2 можно выполнить путем сдвига влево или вправо, соответственно. Для этого в системе команд ARM существуют соответствующие команды. Например, инструкция `ADD r0,r0,r1,LSL #3` означает: сдвинуть содержимое регистра r1 влево три раза (умножить на 8) и затем сложить его с содержимым регистра r0 и результат поместить в r0.

Во-вторых, мы можем использовать инструкции сложения и умножения (MLA), чтобы выполнить операцию P=P+Q*R. В нашем случае мы можем выполнить умножение 7 x C и сложить результат с текущим значением total в r0. Обратите внимание на формат этой инструкции.

Наконец, мы можем выполнить деление на 4, применив два сдвига вправо к делимому. Рис. Example 6.1 демонстрирует состояние системы после того, как программа выполнена.

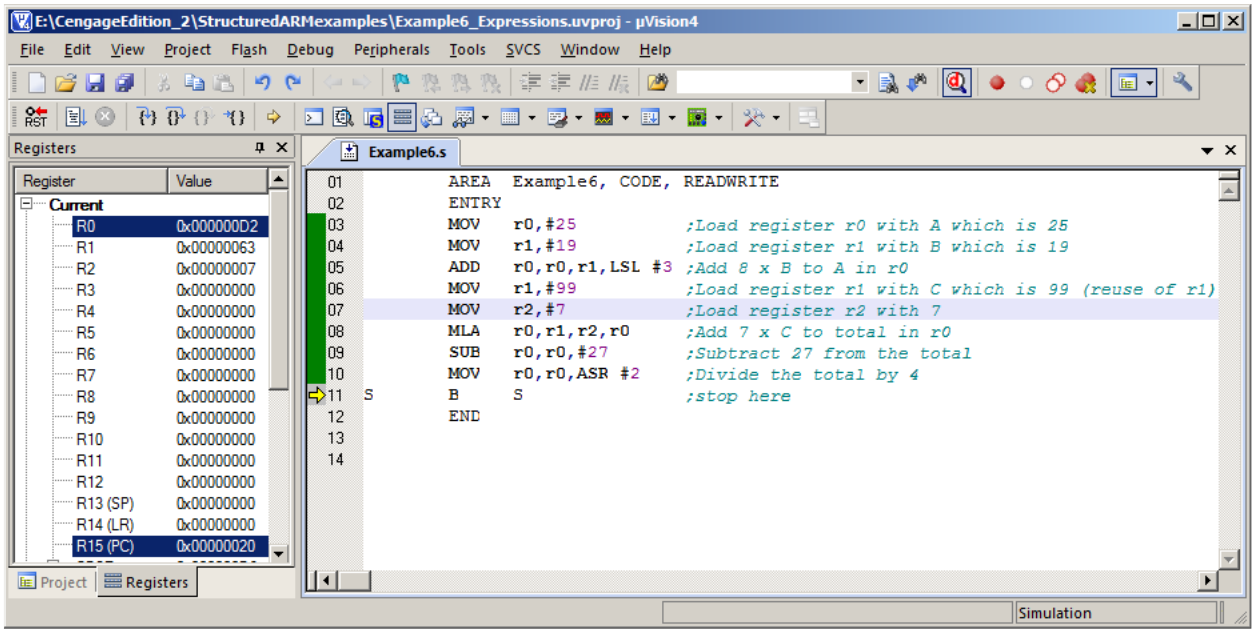


**Рис. Example 6.1 С**остояние системы после выполнения программы