Санкт-Петербургский политехнический университет Петра Великого

Институт компьютерных наук и технологий

Кафедра компьютерных систем и программных технологий

Отчёт по курсовой работе

Дисциплина: Высокоуровневое моделирование средствами SystemC

Тема: Разработка потактовой модели процессора

Выполнил студент гр. 13541/2	____________ (подпись)	Чеботарев Г.М.
Руководитель	____________ (подпись)	Мамутова О.В.

“_” _________2017 г.

Санкт-Петербург

2017

Оглавление

1. Техническое задание:

Раздел 1. Спецификация компонентов процессорной системы.

Обязательные компоненты системы: АЛУ, контроллер памяти, регистровый файл.

Дополнительные компоненты системы по индивидуальному заданию:

- контроллер прямого доступа к памяти (DMA)
- поддержка виртуальной памяти (MMU)
- поддержка защиты памяти (MPU)
- кэш-память команд/данных ($I/D)
- интерфейс ввода/вывода (GPIO)

Варианты индивидуальных заданий

№ п/п	DMA	MMU	MPU	$I/D	GPIO
1	+	+			+
2	+		+		+
3	+			+	+
4		+	+		+
5		+		+	+
6			+	+	+

Предусмотреть отладочный режим работы модулей.

Раздел 2. Спецификация системного окружения для отладки и тестирования процессора.

Допущения:

Время доступа к внешней памяти — от 7 процессорных тактов.

Раздел 3. Спецификация системы команд процессора типа RISC, исходя из выбранного набора компонентов процессорной системы и целевых алгоритмов.

Раздел 4. Разработка описания ядра процессора, компонентов процессорной системы и элементов системного окружения на языке SystemC.

Раздел 5. Модульное тестирование разработанной системы.

Раздел 6. Создание программы в машинных кодах для реализации заданных алгоритмов. Демонстрация работоспособности процессора. Оценка эффективности выполнения алгоритмов.

Алгоритмы:

1. Сортировка (для массивов размером до 1024 слов)

```
a[n]
FOR j=0 TO n-2 STEP 1
        f=0
        FOR i=0 TO n-1-j STEP 1
                IF a[i] > a[i+1] THEN
                        SWAP A[i], A[i+1]
                        f=1
        IF f=0 THEN EXIT FOR
```

2. Умножение матриц (для матриц с каждой размерностью до 1024 слов)

```
a[n][m]
b[m][p]
q[n][p] = 0
FOR i = 0 TO n-1 STEP 1
        FOR j = 0 TO p-1 STEP 1
                FOR k = 0 TO m-1 STEP 1
                        q[i][j] = q[i][j] + a[i][k]*b[k][j]
```

3. Медианный фильтр для двумерного массива

```
  matrix[m_width][m_height]
  edgex = win_width / 2;
  edgey = win_height / 2;
  for (x = edgex; x < m_width - edgex; x++){
      for (y = edgey; y < m_height - edgey; y++){
         window[win_width][win_height]
         for (fx = 0; fx < win_width; fx++){
            for (fy = 0; fy < win_height; fy++){
               window[fx][fy] := matrix[x + fx - edgex][y + fy - edgey]
            }
         }
         sort window[][]
         matrix[x][y] := window[win_width / 2][win_height / 2]
      }
   }
```

2. Краткая документация по микпроцессору

2.1. Архитектура разработанного микропроцессора

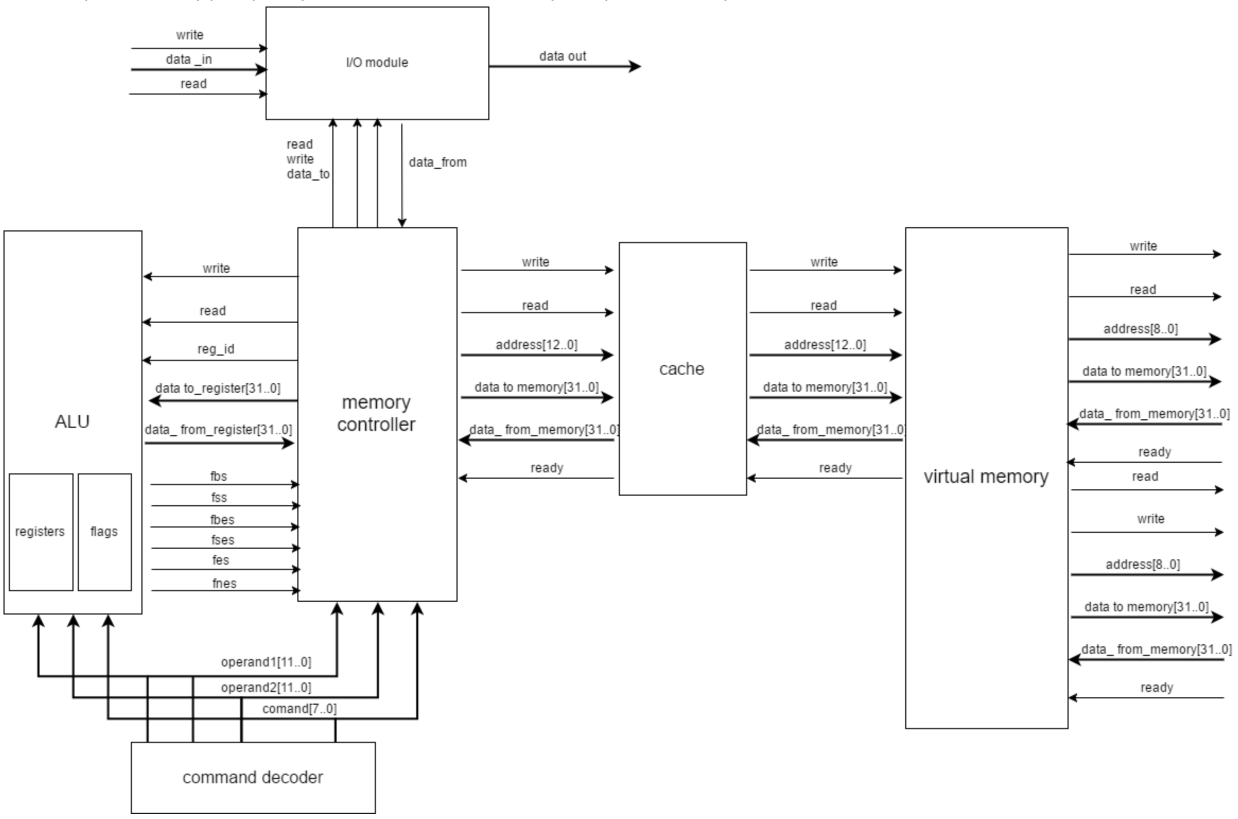


Рис. 1. Архитектура разработанного микропроцессора

Согласно варианту, была разработана следующая схема микропроцессора (рис. 1). Микропроцессор состоит из 6 модулей:

1. Арифметически-логическое устройство
2. Контроллер памяти
3. Кэш данных/команд
4. Модуль виртуальной памяти
5. Модуль ввода/вывода
6. Декодер команд

2.2. Описание работы

Контроллер памяти (КП) выступает так же в качестве управляющего устройства. УУ генерирует сигналы на считывание команд из памяти. Память представлена кэшем, блоком виртуальной памяти и непосредственно памятью в виде массива в тесте. Считав команду из памяти, КП передает команду на дешифрацию декодеру команд (ДК). ДК декодирует команду. Команда представлена 3 составляющими: КОП (8 бит), операнд 1 и операнд 2 (по 12 бит). Декодировав команду, ДК одновременно передает КОП и два операнда на входы АЛУ и КП. И АЛУ и КП анализируют КОП, и в случае, если КОП предназначен для них - выполняют операцию. В противном случае, команда ими игнорируется. Основная работа происходит в КП, ДК, АЛУ, КЭШе, ВП, однако так же есть возможность записать и считать данные в/из модуля I/O.

Блок виртуальной памяти работает сразу с двумя массивами памяти. В момент запуска МП, виртуальная память погружается 1 страницей. Размер страница 256 байт. Максимальное количество загруженных страниц в ОЗУ равно 4. В случае, необходимости, 1 из страниц выгружается. После загрузки ВП, кэш так же инициализируется (считывая первый блок из 1 страницы). Размер кэша 16 байт, 32 блока. В случае получения команды end, данные сохраняются из кэша в ОЗУ, из ОЗУ в основной массив памяти.

2.3. Реализованне иструкции

Мнемоника команд:	Код	Описание команд
АЛУ		
Группа 1 – математические команды		
add reg1, reg2	00000010	Reg1 = reg1 + reg2
sub reg1, reg2	00010010	Reg1 = reg1 - reg2
mult reg1, reg2	00100010	Reg1 = reg1 * reg2
div re1, reg2	00110010	Reg1 = reg1 / reg2
inc reg1	01000010	Reg1 = reg1 + 1
dec reg1	01010010	Reg1 = reg1 – 1
Группа 2 – команды булевой алгебры		
xor reg1, reg2	00000100	Reg1 = reg1 xor reg2
and reg1, reg2	00010100	Reg1 = reg1 and reg2
or reg1, reg2	00100100	Reg1 = reg1 or reg2
not re1, reg2	00110100	Reg1 = not reg1
Группа 3 – мат. сдвиги		
rs reg1	00000110	Reg1 = reg1 >> 1
ls reg1	00010110	Reg1 = reg1 << 1
rsn reg1, reg2	00100110	Reg1 = reg1 >> reg2
lsn reg1, reg2	00110110	Reg1 = reg1 << reg2
Группа 4 – мат. неравенства		
fbs reg1, reg2 (1)	00001000	Fbs = reg1 > reg2
fss reg1, reg2 (2)	00011000	Fss = reg1 < reg2
fbes reg1, reg2 (5)	00101000	Fbes = reg1 >= reg2
fses reg1, reg2 (6)	00111000	Fses = reg1 > reg2
fes reg1, reg2 (3)	01001000	Fes = reg1 == reg2
fnes reg1, reg2 (4)	01011000	Fnes = reg1 != reg2
**	***********	**************************
Контроллер памяти		
Группа 1 – Команды с прямой адресацией		
mov addr, #5		
mov addr, addr	00000011	Memory[addr] = number
mov addr, reg	00010011	Memory[addr1] = Memory[addr2]
mov reg1, reg2	00100011	Memory[addr] = reg

mov reg1, addr	00110011	Reg1 = reg2
mov reg, #5	01000011	Reg = addr
mov reg, cmd_counter	01010011	Reg = number
mov cmd_counter, reg	01100011	Reg = cmd_counter
	01110011	cmd_counter = reg
Группа 2 – Команды переходов		
jump #5	00000101	cmd_counter = number

cjmp idflag, addr	00010101	if(flag == true)
		cmd_counter = number

ret	00100101	
call addr	00110101	
Группа 3 – Команды с косвенной адресацией		
movx reg1, reg2	00000111	reg1 = memory[reg2]
movx reg1, addr	00010111	reg1 = memory[addr]
movm reg1, reg2	00100111	memory[reg2] = reg1
movm addr, reg2	00110111	memory[reg2] = memory[addr]
Группа 4 – команды i/o		
mov reg, ioreg	00001001	reg = ioreg
mov ioreg, reg	00011001	ioreg = reg

Для тестирования микропроцессора, был написан набор тестов.

3. Тестирование

3.1. Виртуальная память

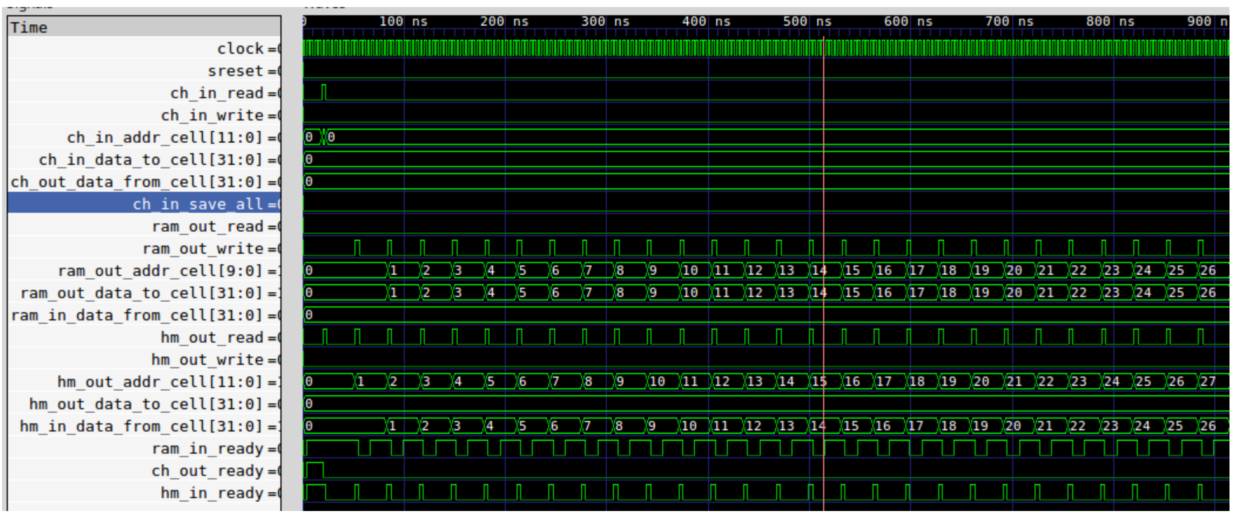


Рис. 2. Считывание страницы из основного блока памяти

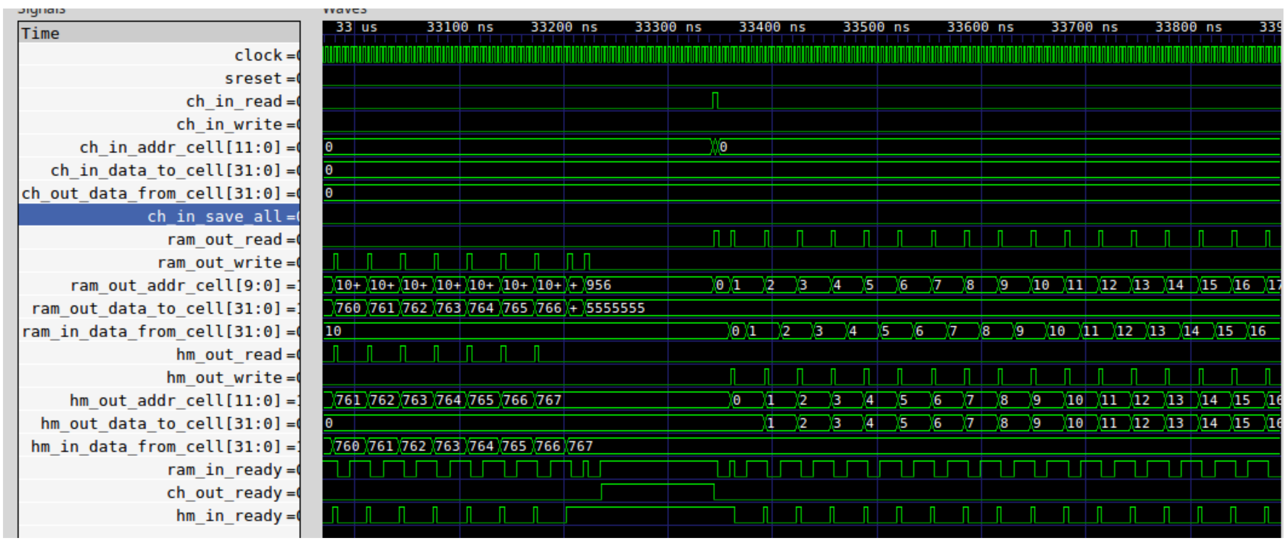


Рис. 3. Запись страницы в основном блок памяти

Во время теста, на входы виртуальной памяти поступают запросы на считывание данных и их запись в разные страницы памяти. Блок ВМ подкачивает необходимые страницы, по мере необходимости. В некоторый момент, количество свободной памяти в ОЗУ становится недостаточным. Тогда ВМ проверяет страницу, которая находиться в ОЗУ дольше всех (принцип fifo). Если страница была модифицирована, то она сначала выгружается, и затем на освобождённое место подгружается нужная страница. Если же страница не была модифицирована, то запись происходит поверх.

Код теста приведен в Приложении 1. Код модуля в Приложении 11.

3.2. Кэш

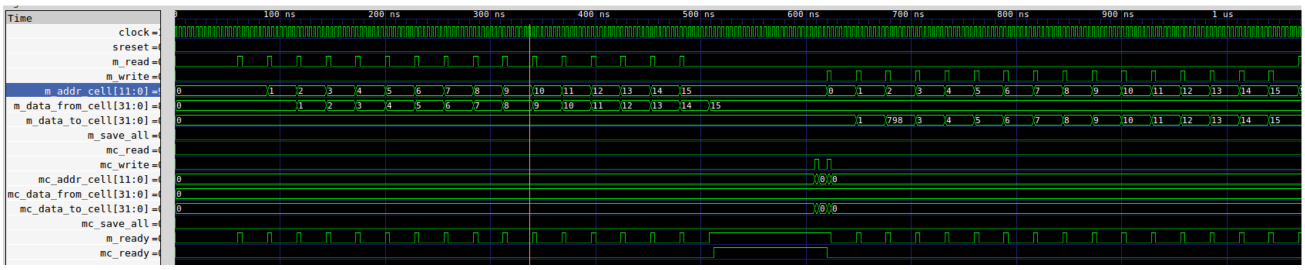


Рис. 4 . Загрузка и выгрузка блока памяти

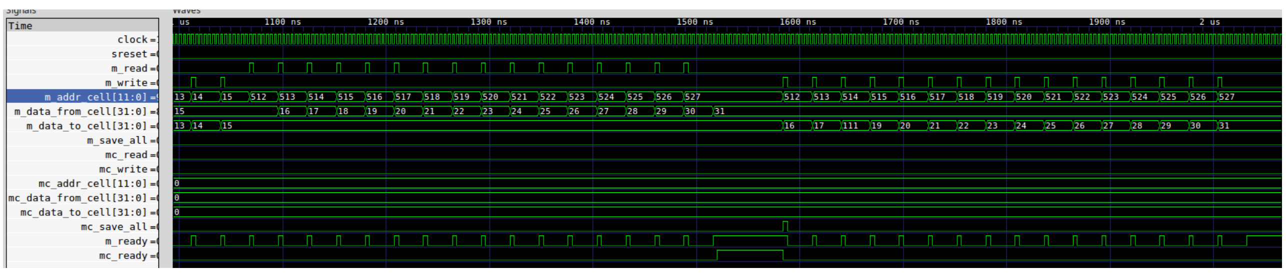


Рис. 5. Выгрузка содержимого по сигналу.

Работа кэша, схожа с работой блока ВП. Тест так же схож. Главным отличием можно считать размер блоков. Блок кэша имеет размер 16 байт. В Кэше таких блоков 32. Разработанный модуль кэша - прямого отображения. Test bench заключался в демонстрации возможностей разработанного модуля (загрузка различных блоков, перекрывающих друг друга и нет, с модификацией данных и без.) Код приведен в приложении 2. Код модуля в Приложении 12.

3.3. Декодер команд

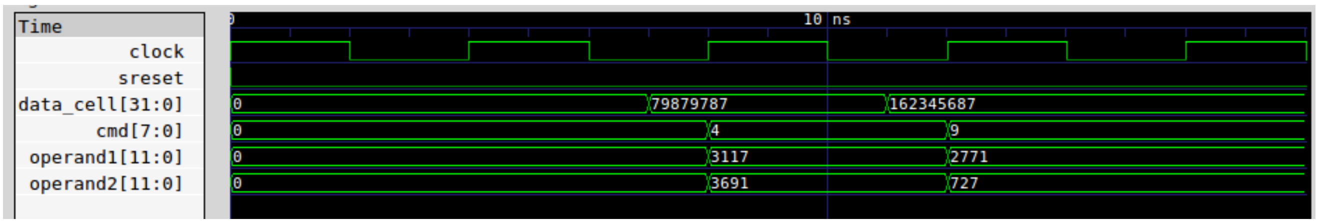


Рис. 6. Декодирование команды

Декодер команд самый простой из модулей. Он принимает на вход 32-битную ячейку из памяти, и декодирует ее, выдавая на свои выходы КОП, Операнд1 и Операнд2. Тест приведен в приложении 3. Код модуля в Приложении 14.

3.4. Арифметически-логическое устройство

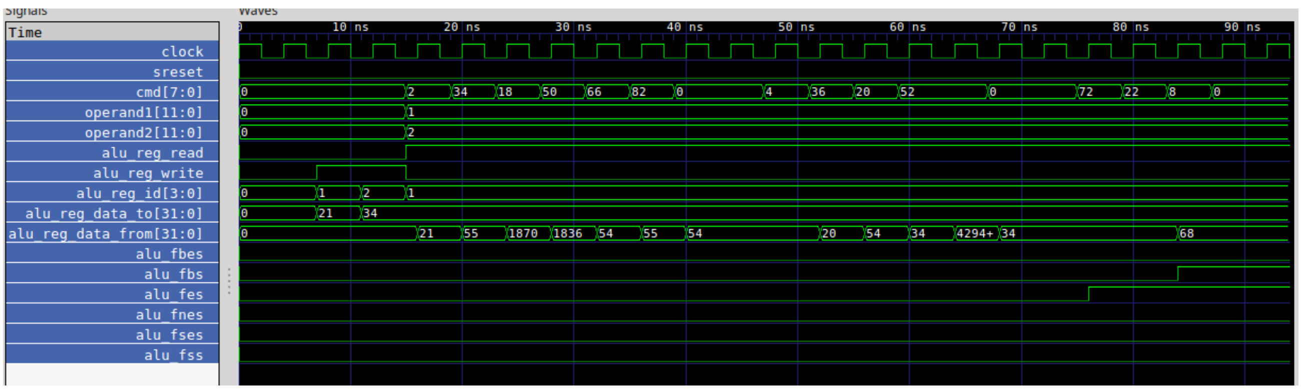


Рис. 7. Демонстрация работы

Арифметически-логический блок работает за один такт. На первом такте получает данные, на второй выдает результат. Команды АЛУ реализованы таким образом, что он работает только с РОН. Прямого взаимодействия с памятью не имеет. На рисунке 7 представлен приведен пример работы АЛУ, для ряда математических и логических операций. Первоначально значения записываются в регистры, после на каждом такте выполняется некоторая команда, пришедшая на вход (в данном случае это «+», «-», «*» и т.д.). Код теста приведен в Приложении 4. Код модуля в Приложении 15.

3.5. Устройство I/O

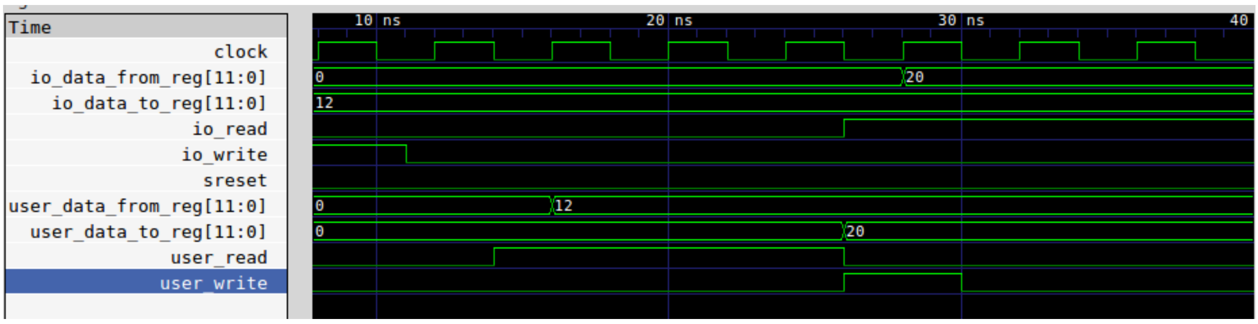


Рис. 8. Демонстрация работы модуля i/o

На первом такте МК подает сигнал записи данных в регистр i/o, на третьем пользователь читает данные. Затем пользователь подает свои данные, подкрепляя их сигналом записи. Код теста приведен в приложении 5. Код модуля в Приложении 16.

3.6. Контроллер памяти

Контроллер памяти так же является управляющим устройство, т.е. он генерируется сигналы для организации работы:

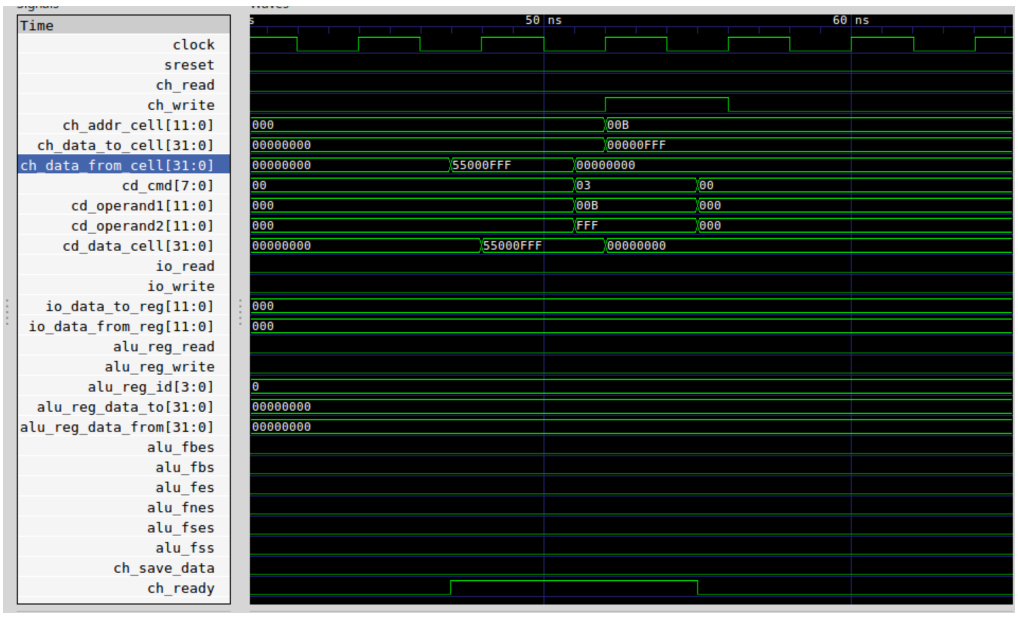


Рис. 9. Организация работы модулей.

На первом такте КП, генерирует сигнал на считывание команды из памяти (память на момент считывания готова к работе). После, полученная команда передается на вход ДК. На следующем такте команда возвращается в преобразованном виде (коп + 2 операнда). В зависимости от пришедшего КОП, будет выполнена та или иная операция (см. рис. 10)

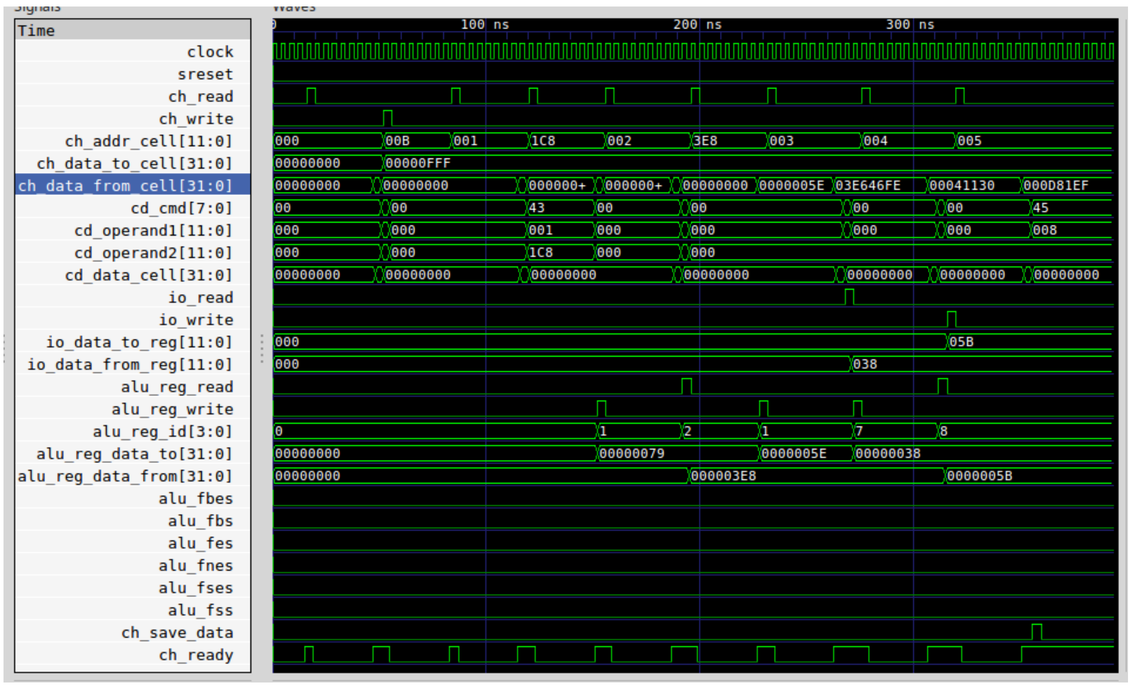
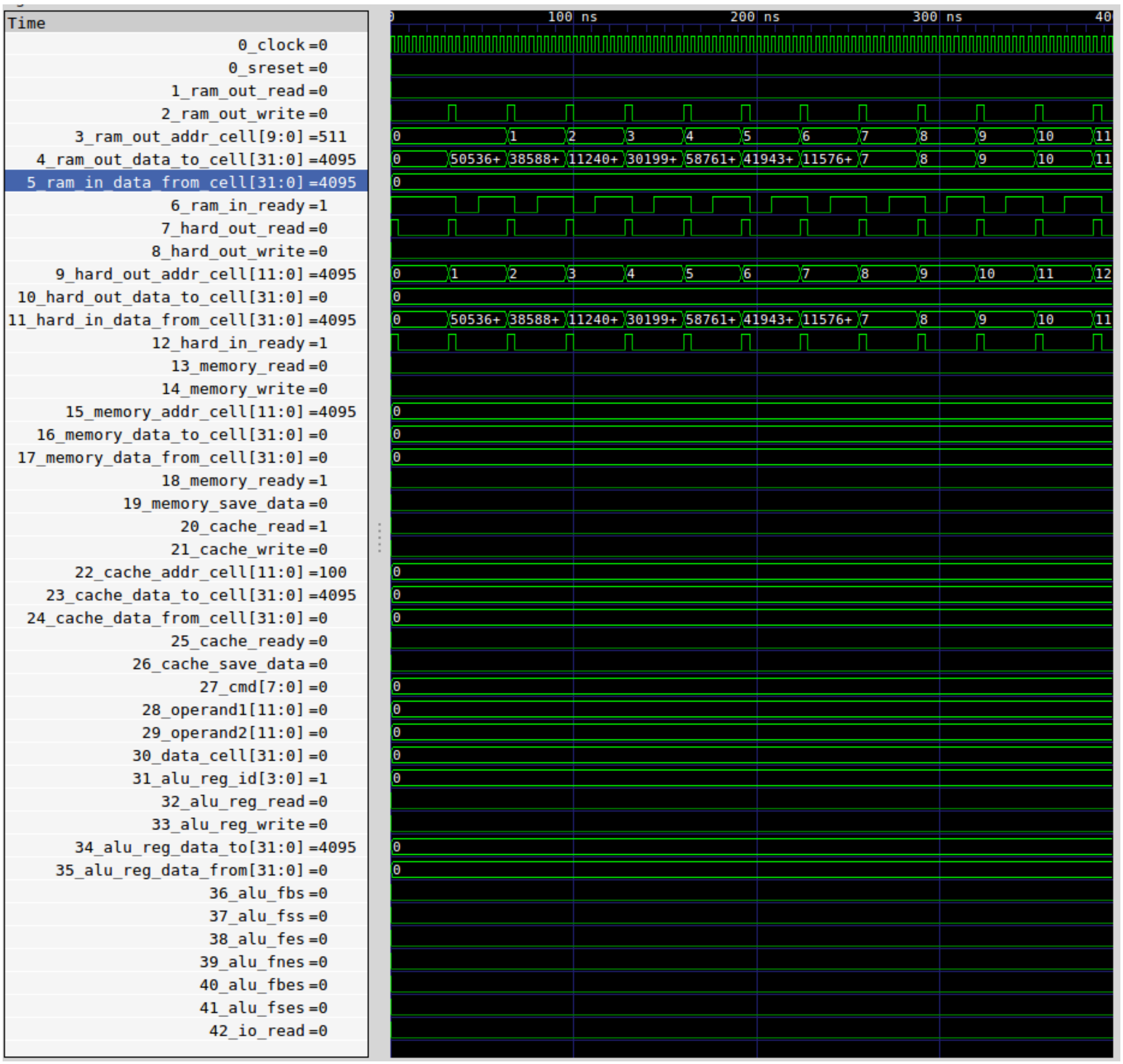


Рис. 10. Работа контроллера памяти

Тест приведен в приложении 6. Код модуля в Приложении 13.

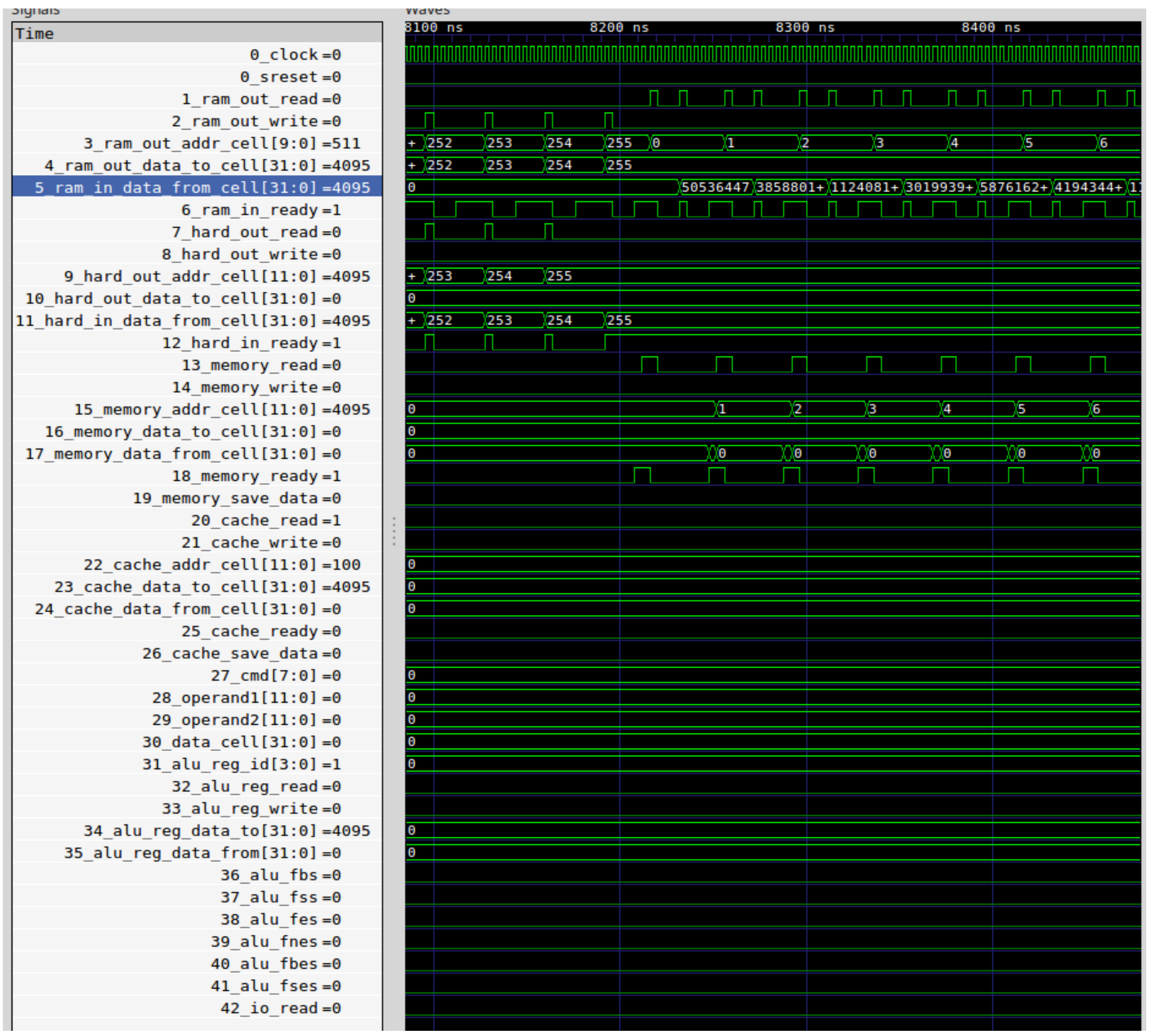
3.7. Тестирование всего микропроцессора

В завершении модули были собраны воедино, и был проведен небольшой тест на программе из 7 команд, по возможности демонстрирующий работу всех модулей МП.



```
Time
0_clock=0
0_sreset=0
1_ram_out_read=0
2_ram_out_write=0
3_ram_out_addr_cell[9:0]=511
4_ram_out_data_to_cell[31:0]=4095
5_ram_in_data_from_cell[31:0]=4095
6_ram_in_ready=1
7_hard_out_read=0
8_hard_out_write=0
9_hard_out_addr_cell[11:0]=4095
10_hard_out_data_to_cell[31:0]=0
11_hard_in_data_from_cell[31:0]=4095
12_hard_in_ready=1
13_memory_read=0
14_memory_write=0
15_memory_addr_cell[11:0]=4095
16_memory_data_to_cell[31:0]=0
17_memory_data_from_cell[31:0]=0
18_memory_ready=1
19_memory_save_data=0
20_cache_read=1
21_cache_write=0
22_cache_addr_cell[11:0]=100
23_cache_data_to_cell[31:0]=4095
24_cache_data_from_cell[31:0]=0
25_cache_ready=0
26_cache_save_data=0
27_cmd[7:0]=0
28_operand1[11:0]=0
29_operand2[11:0]=0
30_data_cell[31:0]=0
31_alu_reg_id[3:0]=1
32_alu_reg_read=0
33_alu_reg_write=0
34_alu_reg_data_to[31:0]=4095
35_alu_reg_data_from[31:0]=0
36_alu_fbs=0
37_alu_fss=0
38_alu_fes=0
39_alu_fnes=0
40_alu_fbes=0
41_alu_fses=0
42_io_read=0
```

Рис. 11. Загрузка первой страницы из «жесткого диска» в «ОЗУ»



```
Time
                0_clock=0
               0_sreset=0
           1_ram_out_read=0
          2_ram_out_write=0
    3_ram_out_addr_cell[9:0]=511
 4_ram_out_data_to_cell[31:0]=4095
5_ram_in_data_from_cell[31:0]=4095
           6_ram_in_ready=1
          7_hard_out_read=0
         8_hard_out_write=0
  9_hard_out_addr_cell[11:0]=4095
10_hard_out_data_to_cell[31:0]=0
11_hard_in_data_from_cell[31:0]=4095
          12_hard_in_ready=1
           13_memory_read=0
          14_memory_write=0
  15_memory_addr_cell[11:0]=4095
 16_memory_data_to_cell[31:0]=0
17_memory_data_from_cell[31:0]=0
          18_memory_ready=1
      19_memory_save_data=0
            20_cache_read=1
           21_cache_write=0
   22_cache_addr_cell[11:0]=100
 23_cache_data_to_cell[31:0]=4095
24_cache_data_from_cell[31:0]=0
           25_cache_ready=0
       26_cache_save_data=0
               27_cmd[7:0]=0
         28_operand1[11:0]=0
         29_operand2[11:0]=0
        30_data_cell[31:0]=0
        31_alu_reg_id[3:0]=1
           32_alu_reg_read=0
          33_alu_reg_write=0
    34_alu_reg_data_to[31:0]=4095
  35_alu_reg_data_from[31:0]=0
               36_alu_fbs=0
               37_alu_fss=0
               38_alu_fes=0
              39_alu_fnes=0
              40_alu_fbes=0
              41_alu_fses=0
               42_io_read=0
```

Рис. 12. Загрузка первого блока кэша, после считывания первой страницы в «ОЗУ»

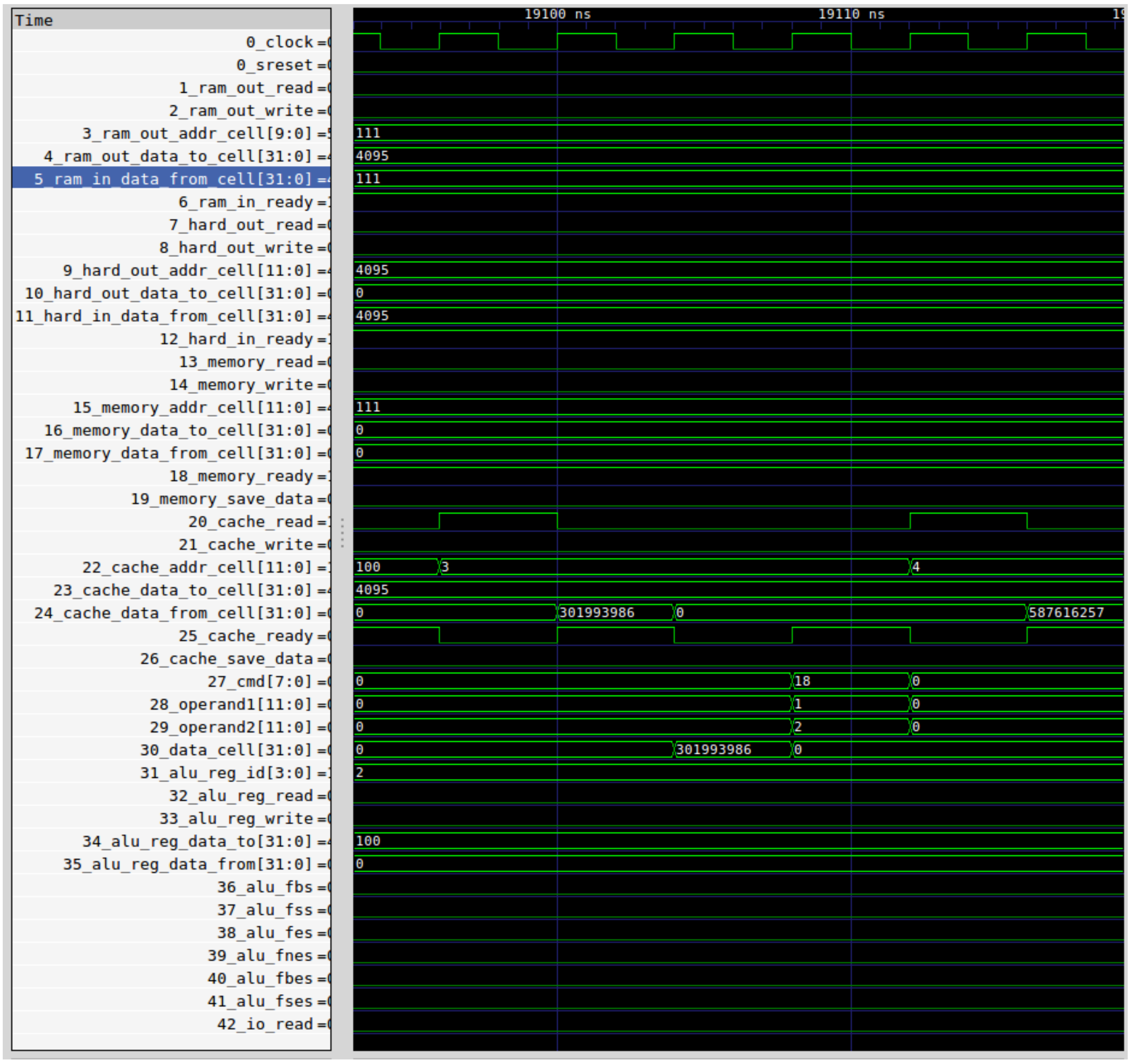


Рис. 13. Чтение команд из кэша, и исполнение

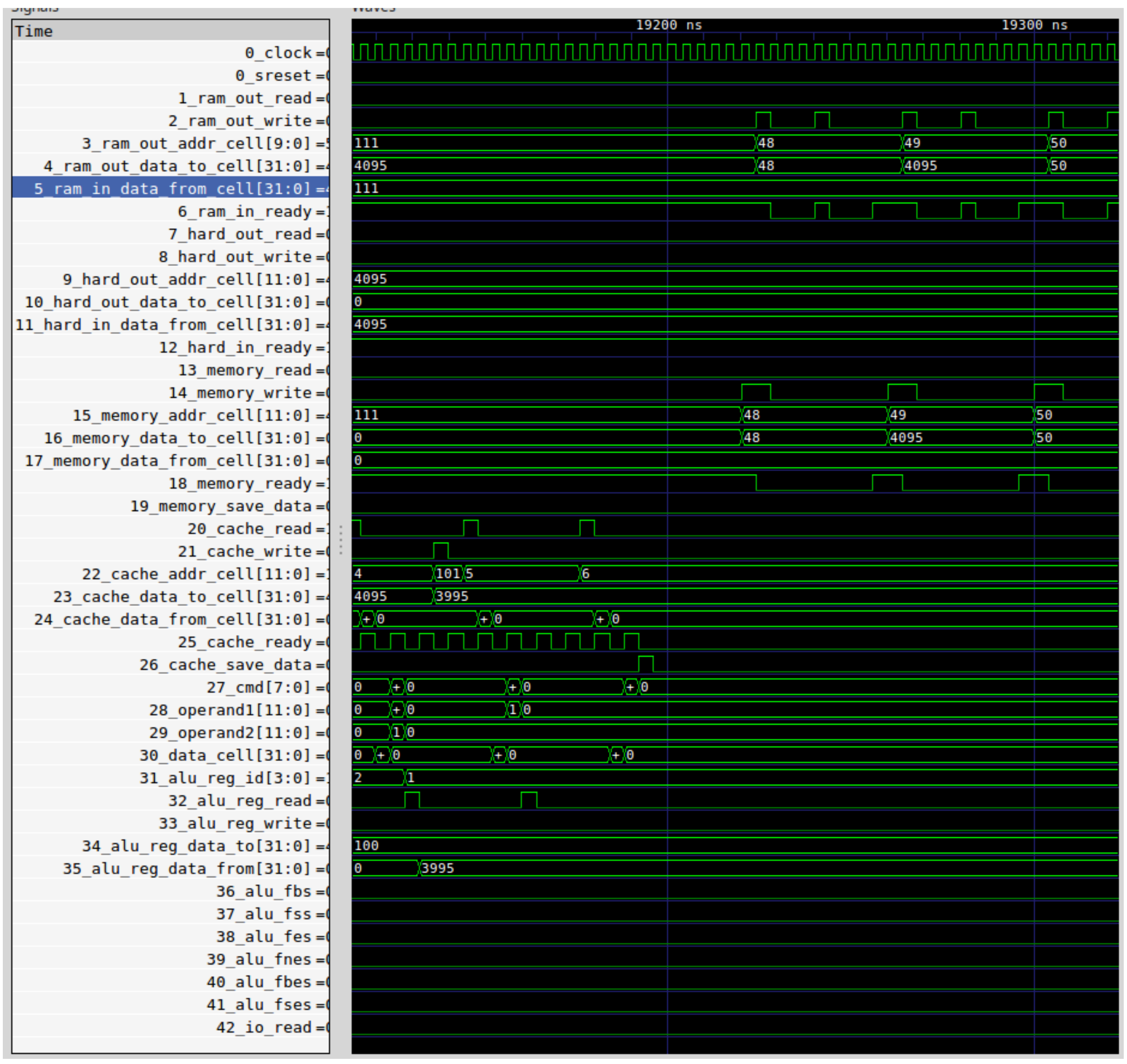


Рис. 14. Обработка сигнала завершения работы (сохранение данных в жесткий диск)

Код теста приведен в приложении 7. Код top-модуля в Приложении 17.

4. Программы и алгоритмы

4.1. Сортировка

Для реализации алгоритма сортировки, приведенного ранее, был написан следующий ассемблерный код:

```
    mov reg0, j;        # init
    mov reg1, i;
    mov reg2, f;
    mov reg3, n;        # count of the elements
    mov reg13, reg3;    # n - 2
    dec reg13;
    dec reg13;
    mov reg4, reg3;
    dec reg4;           # n-1
    mov reg5, a;        # start address
    mov reg15, #1;
    jmp X3;
X1:
    inc reg0;           # j++
    fbs reg0, reg13;    # do while j <= n-2
    jmp fbs, X6;

    mov reg2, #0;       # f = 0
    mov reg1, #0;       # i = 0
    dec reg4;           # n - 1 - j
    jmp X3;

X2:
    inc reg1            # i++
    fbs reg1, reg4      # do while i <= n - 1 - j
    jmp fbs, X1

X3:
    # A[i]
    mov reg6, reg1;     # i
    add reg6, reg5;     # i + a0
    movx reg7, reg6;    # memory[i + a0]

    # A[i + 1]
    mov reg8, reg6;     # i + a0
    inc reg8;           # i + a0 + 1
    movx reg9, reg8;    # memory[i + a0 + 1]

    fbs reg7, reg9;
    jmp fbs, X4;
    jmp X5;

X4:
    movm reg9, reg6;
    movm reg7, reg8;
    mov reg2, #1;

X5:
    fes reg2, reg15;
    jmp fes, X2;
    jmp X1;

X6: end
```

Затем, ассемблерный код был преобразован в машинный:

```
    // programm
    hardware_memory[0] = 1392508928;    // mov reg0, #0;      # init
    hardware_memory[1] = 1392513024;    // mov reg1, #0;
    hardware_memory[2] = 1392517120;    // mov reg2, #0;
    hardware_memory[3] = 1392521416;    // mov reg3, #200;     # count of the elements
    hardware_memory[4] = 855691267;     // mov reg13, reg3;    # n - 2
    hardware_memory[5] = 1375784960;    // dec reg13;
    hardware_memory[6] = 285212672;     // dec reg13;
    hardware_memory[7] = 855654403;     // mov reg4, reg3;
    hardware_memory[8] = 1375748096;    // dec reg4;           # n-1
    hardware_memory[9] = 1392529508;    // mov reg5, #100;     # start address
    hardware_memory[10] = 1392570369;   // mov reg15, #1;
    hardware_memory[11] = 83976192;     // jmp X3(22);
    hardware_memory[12] = 1107296256;   // inc reg0;           # j++             !X1
    hardware_memory[13] = 134217741;    // fbs reg0, reg13;    # do while j <= n-2
    hardware_memory[14] = 352325669;    // jmp fbs, X6(37);
    hardware_memory[15] = 1392517120;   // mov reg2, #0;       # f = 0
    hardware_memory[16] = 1392513024;   // mov reg1, #0;       # i = 0
    hardware_memory[17] = 1375748096;   // dec reg4;           # n - 1 - j
    hardware_memory[18] = 83976192;     // jmp X3(22);
    hardware_memory[19] = 1107300352;   // inc reg1            # i++             !X2
    hardware_memory[20] = 134221828;    // fbs reg1, reg4      # do while i <= n - 1 - j
    hardware_memory[21] = 352325644;    // jmp fbs, X1(12)
    hardware_memory[22] = 855662593;    // mov reg6, reg1;     # i      A[i]     !X3
    hardware_memory[23] = 33579013;     // add reg6, reg5;     # i + a0
    hardware_memory[24] = 117469190;    // movx reg7, reg6;    # memory[i + a0]
    hardware_memory[25] = 855670790;    // mov reg8, reg6;     # i + a0          A[i + 1]
    hardware_memory[26] = 1107329024;   // inc reg8;           # i + a0 + 1
    hardware_memory[27] = 117477384;    // movx reg9, reg8;    # memory[i + a0 + 1]
    hardware_memory[28] = 134246409;    // fbs reg7, reg9;
    hardware_memory[29] = 352325663;    // jmp fbs, X4(31);
    hardware_memory[30] = 84025344;     // jmp X5(34);
    hardware_memory[31] = 654348294;    // movm reg9, reg6;    # swap             !X4
    hardware_memory[32] = 654340104;    // movm reg7, reg8;
    hardware_memory[33] = 1392517121;   // mov reg2, #1;       # f = 1
    hardware_memory[34] = 1207967759;   // fes reg2, reg15;    #                  !X5
    hardware_memory[35] = 352333843;      // jmp fes, X2(19);     # if f == 1 the cycle is contin
    hardware_memory[36] = 83935232;       // jmp X1;
    hardware_memory[37] = 1157627904;     // end                  #                  !X6
```

Данный код, сортирует данные, расположенные с 100 по 300 адрес включительно. Для выполнения сортироки требуется 30 секунд. Результат будет приведен на демонстрации.Код был отлажен на размерности 5, после было увеличено количество сортируемых элементов до 201. Тест приведен в Приложении 8.

4.2. Умножение матриц

Алгоритм умножения матриц так же приведен в отчете выше. Для его реализации был написан следующий код, на языке ассемблер (на основе имеющихся команд в МП)

```
    mov reg0, i;
    mov reg1, j;
    mov reg2, k;
    mov reg3, n;
    mov reg4, m;
    mov reg5, p;
    mov reg6, a0;
    mov reg7, b0;
    mov reg8, c0;
    jump X3
X1:
    mov reg1, 0;        # j = 0
    fes reg0, reg3;     # set flag-fes, if i == n
    jump fes, X4;       # if flag-fes is set, jump to X4
    inc reg0;         # reg0++
X2:
    mov reg2, 0;        # k = 0
    #q[i][j]
    mov reg12, reg0;    # i
    mult reg12, reg3;   # i * n
    add reg12, reg1;    # i * n + j
    add reg12, reg8;    # c0 + i * n + j
    movm reg11, reg12;  # memory[c0 + i * n + j] =  reg11
    mov reg11, #0;      # reset register
    fes reg1, reg5;     # set flag-fes, if j == p
    jump fes, X1;       # if flag-fes is set, jump to X1
    inc reg1;         # reg1++
X3:
    #q[n*i+j] = q[n*i+j] + a[i*n+k] * b[k*m+j]
    #a[i][k]
    mov reg9, reg0;     # i
    mult reg9, reg3;    # i * n
    add reg9, reg2;     # i * n + k
```

```
   add reg9, reg6;      # a0 + i * n + k
   movx reg9, reg9;     # reg9 = memory[a0 + i * n + k]
   #b[k][j]
   mov reg10, reg2;     # k
   mult reg10, reg4;    # k * m
   add reg10, reg1;     # k * m + j
   add reg10, reg7;     # b0 + k * m + j
   movx reg10, reg10;   # reg10 = memory[b0 + k * m + j]
   #q[i][j]
   mult reg9, reg10;    # a[i][k]*b[k][j]
   add reg11, reg9;     # q[i][j] + a[i][k]*b[k][j]
   fes reg2, reg4;      # set flag-fes, if k == m
   jump fes, X2;        # if flag-fes is set, jump to X2
   inc reg2;            # reg2++
   jump X3;
X4:
   ret;
```

Затем, ассемблерный код был преобразован в машинный:

```
 hardware_memory[0] = 1392508928;        // mov reg0, i;
hardware_memory[1] = 1392513024;         // mov reg1, j;
hardware_memory[2] = 1392517120;         // mov reg2, k;
hardware_memory[3] = 1392521226;         // mov reg3, n; n=10
hardware_memory[4] = 855691267;          // mov reg13, reg3;
hardware_memory[5] = 1375784960;         // dec reg13;
hardware_memory[6] = 1392525322;         // mov reg4, m; m=10
hardware_memory[7] = 855695364;          // mov reg14, reg4;
hardware_memory[8] = 1375789056;         // dec reg14;
hardware_memory[9] = 1392529418;         // mov reg5, p; p=10
hardware_memory[10] = 855699461;         // mov reg15, reg5;
hardware_memory[11] = 1375793152;        // dec reg15;
hardware_memory[12] = 1392533604;        // mov reg6, #100;     # a0
hardware_memory[13] = 1392538700;        // mov reg7, #1100;    # b0
hardware_memory[14] = 1392543796;        // mov reg8, #2100;    # c0
hardware_memory[15] = 84013056;          // jump X3
hardware_memory[16] = 1392513024;        // mov reg1, 0;        # j = 0           X1!
hardware_memory[17] = 1207959565;        // fes reg0, reg13;    # set flag-fes, if i == n
hardware_memory[18] = 352333871;         // cjmp fes, X4;       # if flag-fes is set, jump to X4
hardware_memory[19] = 1107296256;        // inc reg0;           # reg0++
hardware_memory[20] = 84013056;          // jump X3
hardware_memory[21] = 1392517120;        // mov reg2, 0;        # k = 0           X2!
hardware_memory[22] = 855687168;         // mov reg12, reg0;    # i
hardware_memory[23] = 570474499;         // mult reg12, reg3;   # i * n
hardware_memory[24] = 33603585;          // add reg12, reg1;    # i * n + j
hardware_memory[25] = 33603592;          // add reg12, reg8;    # c0 + i * n + j
hardware_memory[26] = 654356492;         // movm reg11, reg12;  # m[c0 + i * n + j] =  reg11
hardware_memory[27] = 1392553984;        // mov reg11, #0;      # reset register
hardware_memory[28] = 1207963663;        // fes reg1, reg15;    # set flag-fes, if j == p
```

```
hardware_memory[29] = 352333840;       // jump fes, X1;       # if flag-fes is set, jump to X1
hardware_memory[30] = 1107300352;      // inc reg1;           # reg1++
hardware_memory[31] = 855674880;       // mov reg9, reg0;     # i              X3!
hardware_memory[32] = 570462211;       // mult reg9, reg3;    # i * n
hardware_memory[33] = 33591298;        // add reg9, reg2;     # i * n + k
hardware_memory[34] = 33591302;        // add reg9, reg6;     # a0 + i * n + k
hardware_memory[35] = 117477385;       // movx reg9, reg9;    # reg9 = m[a0 + i * n + k]
hardware_memory[36] = 855678978;       // mov reg10, reg2;    # k
hardware_memory[37] = 570466308;       // mult reg10, reg4;   # k * m
hardware_memory[38] = 33595393;        // add reg10, reg1;    # k * m + j
hardware_memory[39] = 33595399;        // add reg10, reg7;    # b0 + k * m + j
hardware_memory[40] = 117481482;       // movx reg10, reg10;  # reg10 = m[b0 + k * m + j]
hardware_memory[41] = 570462218;       // mult reg9, reg10;   # a[i][k]*b[k][j]
hardware_memory[42] = 33599497;        // add reg11, reg9;    # q[i][j] + a[i][k]*b[k][j]
hardware_memory[43] = 1207967758;      // fes reg2, reg14;    # set flag-fes, if k == m
hardware_memory[44] = 352333845;       // jump fes, X2;       # if flag-fes is set, jump to X2
hardware_memory[45] = 1107304448;      // inc reg2;           # reg2++
hardware_memory[46] = 84013056;        // jump X3
hardware_memory[47] = 1157627904;      // end                 #                X4!
```

Данная реализация умножает две матрицы 10 на 10, расположенные по адресам 100-199 и 1100-1199, и записывает результат с 2100-2199 адрес. Умножениематрицы 10 на 10 выполняется за 40 секунд. Алгоритм может быть использован как для умножения квадратных матриц, так и для прямоугольных. В связи с ограничением памяти 4кБайт, накладываются ограничения на размеры матриц. Код теста приведен в Приложении 9.

4.3. Медианный фильтр для двумерного массива

В качестве алгоритма для фильтра был использован следующий код:

```
matrix[m_width][m_height]
edgex = win_width / 2;
edgey = win_height / 2;
for (x = edgex; x < m_width - edgex; x++){
   for (y = edgey; y < m_height - edgey; y++){
      window[win_width][win_height]
      for (fx = 0; fx < win_width; fx++){
         for (fy = 0; fy < win_height; fy++){
            window[fx][fy] := matrix[x + fx - edgex][y + fy - edgey]
         }
      }
      sort window[][]
      matrix[x][y] := window[win_width / 2][win_height / 2]
   }
}
```

При инициализации задается размер массива и размер окна. Элементы, попавшие в окно, копируются и сортируются. В отсортированном массиве окна выбирается средний элемент. Этот элемент непосредствменно записывается в массив (в соответствующий адрес).

Ассемблерный код:

```
mov reg0, m_width   # m_width
mov reg1, m_height  # m_height

mov reg2, w_width   # w_width
mov reg3, w_height  # w_height

mov reg14, reg2
mult reg14, reg3    # count of elements in window

mov reg15, #2
mov reg4, reg2      # w_width
div reg4, reg15     # edgex = w_width/2

mov reg5, reg3      # w_height
div reg5, reg15     # edgey = w_height/2

mov reg6, reg4      # x = edgex                # X1:
mov reg7, reg5      # y = edgey                # X2:
mov reg8, #0        # fx = 0                   # X3:
mov reg9, #0        # fy = 0                   # X4:

#window[fx][fy] := matrix[x + fx - edgex][y + fy - edgey]
mov reg10, reg6     # x
add reg10, reg8     # x + fx
sub reg10, reg4     # x + fx - edgex
mult reg10,reg0     # (x + fx - edgex) * m_width
add reg10, reg7     # (x + fx - edgex) * m_width + y
add reg10, reg9     # (x + fx - edgex) * m_width + y + fy
sub reg10, reg5     # (x + fx - edgex) * m_width + y + fy - edgey
add reg10, #200     # 200 + (x + fx - edgex) * m_width + y + fy - edgey = bias in memory to
element
movx reg10, reg10   # reg10 = memory[reg10]
mov reg11, reg8     # fx
mult reg11, reg2    # fx * w_width
add reg11, reg9     # fx * w_width + fy
add reg11, #128     # 128 + fx * w_width + fy = bias in the window memory
movm reg10, reg11   # memory[reg11] = reg10

# цикл 4
inc reg9
fss reg9, reg3      # if reg9 < w_height
jmp fss, X4
```

```
# цикл 3
inc reg8
fss reg8, reg2      # if reg8 < w_width
jmp fss, X3

call save_reg
mov reg3, reg14;    # count of the elements in window
call sort
call read_reg

#matrix[x][y] := window[w_width / 2][w_height / 2]
mov reg10, reg2     # w_width
div reg10, reg15    # w_width / 2
mult reg10, reg2    # w_width / 2 * w_width
mov reg11, reg3     # w_height
div reg11, reg15    # w_height / 2
add reg10, reg11    # w_width / 2 * w_width + w_height / 2
add reg10, #128     # 128 + w_width / 2 * w_width + w_height / 2
movx reg10, reg10   # reg10 = memory[reg10]
mov reg11, reg6     # x
mult reg11, reg0    # x * m_width
add reg11, reg7     # x * m_width + y
add reg11, #200
movm reg10, reg11   # memory[reg11] = reg10

# цикл 2
inc reg7
mov reg15, reg1     # m_height
sub reg15, reg5     # m_height - edgey
fss reg7, reg15     # if reg7 < reg15
jmp fss, X2

# цикл 1
inc reg6
mov reg15, reg0     # m_width
sub reg15, reg4     # m_width - edgex
fss reg6, reg15     # if reg6 < reg15
jmp fss, X1
end

save_reg:
   mov #100, reg0
   mov #101, reg1
   mov #102, reg2
   mov #104, reg4
   mov #105, reg5
   mov #106, reg6
   mov #107, reg7
   mov #108, reg8
   mov #109, reg9
   mov #113, reg13
   mov #115, reg15
   ret

read_reg:
   mov reg0, #100
   mov reg1, #101
   mov reg2, #102
   mov reg4, #104
   mov reg5, #105
   mov reg6, #106
   mov reg7, #107
   mov reg8, #108
   mov reg9, #109
   mov reg13, #113
   mov reg15, #115
   ret


sort:
   mov reg0, #0       # row or column
   mov reg1, i;       # init
   mov reg2, f;
   //mov reg3, n;       # count of the elements
   mov reg13, reg3;    # n - 2
   dec reg13;
   //dec reg13;
   mov reg4, reg3;
   dec reg4;          # n-1
   mov reg5, #128;     # start addres
   mov reg15, #1;
   jmp X3;
X1:
   inc reg0;          # j++
   fbs reg0, reg13;    # do while j <= n-2
   jmp fbs, X6;
   mov reg2, #0;      # f = 0
   mov reg1, #0;      # i = 0
   dec reg4;          # n - 1 - j
   jmp X3;
X2:
   inc reg1           # i++
   fbs reg1, reg4      # do while i <= n - 1 - j
   jmp fbs, X1
X3:
   # A[i]
   mov reg6, reg1;     # i
   add reg6, reg5;     # i + a0
   movx reg7, reg6;    # memory[i + a0]
   # A[i + 1]
   mov reg8, reg6;     # i + a0
   inc reg8;           # i + a0 + 1
   movx reg9, reg8;    # memory[i + a0 + 1]
   fbs reg7, reg9;
   jmp fbs, X4;
   jmp X5;
X4:
   movm reg9, reg6;
   movm reg7, reg8;
   mov reg2, #1;
X5:
   fes reg2, reg15;
   jmp fes, X2;
   jmp X1;
X6: ret              #
```

Машинный код:

```
hardware_memory[0] =  1392508938;        // mov reg0, m_width   # m_width
   hardware_memory[1] =  1392513034;     // mov reg1, m_height  # m_height
   hardware_memory[2] =  1392517123;     // mov reg2, w_width   # w_width
   hardware_memory[3] =  1392521219;     // mov reg3, w_height  # w_height
   hardware_memory[4] =  855695362;      // nope cmd (mov reg14, reg2)       FIX
   hardware_memory[5] =  1392566280;  //mov reg14, #8    # count of elemes in window    FIX
   hardware_memory[6] =  1392570370;     // mov reg15, #2
   hardware_memory[7] =  855654402;      // mov reg4, reg2      # w_width
   hardware_memory[8] =  1392558208;     // # bias for window
   hardware_memory[9] =  1392562376;     // # bias for memory
   hardware_memory[10] =  838877199;     // div reg4, reg15     # edgex = w_width/2
   hardware_memory[11] =  855658499;     // mov reg5, reg3      # w_height
   hardware_memory[12] =  838881295;     // div reg5, reg15     # edgey = w_height/2
   hardware_memory[13] =  855662596;     // mov reg6, reg4      # x = edgex      //!X1
   hardware_memory[14] =  855666693;     // mov reg7, reg5      # y = edgey      //!X2
   hardware_memory[15] =  1392541696;   // mov reg8, #0       # fx = 0           //!X3
   hardware_memory[16] =  1392545792;   // mov reg9, #0       # fy = 0           //!X4
   // window[fx][fy] := matrix[x + fx - edgex][y + fy - edgey]
   hardware_memory[17] =  855678982;     // mov reg10, reg6    # x
   hardware_memory[18] =  33595400;      // add reg10, reg8    # x + fx
   hardware_memory[19] =  302030852;     // sub reg10, reg4    # x + fx - edgex
   hardware_memory[20] =  570466304;     // mult reg10,reg0    # (x + fx - edgex) * m_width
   hardware_memory[21] =  33595399;      // add reg10, reg7  # (x + fx - edgex) * m_width + y
   hardware_memory[22] =  33595401;      // add reg10, reg9  # (x + fx - edgex) * m_width + y + fy
   hardware_memory[23] =  302030853;     // sub reg10, reg5    # (x + fx - edgex) * m_width + y + fy - edgey
   hardware_memory[24] =  33595405;      // add reg10, #200      # 200 + (x + fx - edgex) *
m_width + y + fy - edgey = bias in memory to element
   hardware_memory[25] =  117481482;     // movx reg10, reg10   # reg10 = memory[reg10]
   hardware_memory[26] =  855683080;     // mov reg11, reg8     # fx
   hardware_memory[27] =  570470402;     // mult reg11, reg2    # fx * w_width
   hardware_memory[28] =  33599497;      // add reg11, reg9     # fx * w_width + fy
   hardware_memory[29] =  33599500;      // add reg11, #128     # 128 + fx * w_width + fy =
                                         // bias in the window memory
   hardware_memory[30] =  654352395;     // movm reg10, reg11   # memory[reg11] = reg10
   // цикл 4
   hardware_memory[31] =  1107333120;    // inc reg9
   hardware_memory[32] =  402690051;     // fss reg9, reg3     # if reg9 < w_height
   hardware_memory[33] =  352329745;     // jmp fss, X4(17)
   // цикл 3
   hardware_memory[34] =  1107329024;    // inc reg8
   hardware_memory[35] =  402685954;     // fss reg8, reg2     # if reg8 < w_width
   hardware_memory[36] =  352329744;     // jmp fss, X3(16)
   hardware_memory[37] =  889458688;     // call save_reg(65)
   hardware_memory[38] =  855650318;     // mov reg3, reg14;    # count of the elements in
window
   hardware_memory[39] =  889556992;     // call sort (89)
   hardware_memory[40] =  889507840;     // call read_reg(77)
   // matrix[x][y] := window[w_width / 2][w_height / 2]
   hardware_memory[41] =  855678978;     // mov reg10, reg2     # w_width
   hardware_memory[42] =  838901775;     // div reg10, reg15    # w_width / 2
   hardware_memory[43] =  570466306;     // mult reg10, reg2    # w_width / 2 * w_width
   hardware_memory[44] =  1392553987;    // mov reg11, #3       # w_height    FIX
   hardware_memory[45] =  838905871;     // div reg11, reg15    # w_height / 2
   hardware_memory[46] =  33595403;      // add reg10, reg11    # w_width / 2 * w_width + w_height / 2
   hardware_memory[47] =  33595404;      // add reg10, #128    # 128 + w_width / 2 * w_width + w_height / 2
   hardware_memory[48] =  117481482;     // movx reg10, reg10   # reg10 = memory[reg10]
   hardware_memory[49] =  855683078;     // mov reg11, reg6     # x
   hardware_memory[50] =  570470400;     // mult reg11, reg0    # x * m_width
   hardware_memory[51] =  33599495;      // add reg11, reg7     # x * m_width + y
   hardware_memory[52] =  33599501;      // add reg11, #200
   hardware_memory[53] =  654352395;     // movm reg10, reg11   # memory[reg11] = reg10
   // цикл 2
   hardware_memory[54] =  1107324928;    // inc reg7
   hardware_memory[55] =  855699457;     // mov reg15, reg1     # m_height
   hardware_memory[56] =  302051333;     // sub reg15, reg5    # m_height - edgey
   hardware_memory[57] =  402681871;     // fss reg7, reg15    # if reg7 < reg15
   hardware_memory[58] =  352329743;     // jmp fss, X2(15)
   // цикл 1
   hardware_memory[59] =  1107320832;    // inc reg6
   hardware_memory[60] =  855699456;     // mov reg15, reg0     # m_width
   hardware_memory[61] =  302051332;     // sub reg15, reg4    # m_width - edgex
   hardware_memory[62] =  402677775;     // fss reg6, reg15    # if reg6 < reg15
   hardware_memory[63] =  352329742;     // jmp fss, X1(14)
   hardware_memory[64] =  1157627904;    // end
   //!save_reg
   hardware_memory[65] =  587816960;     // mov #150, reg0
   hardware_memory[66] =  587821057;     // mov #151, reg1
   hardware_memory[67] =  587825154;     // mov #152, reg2
```

```
hardware_memory[68] =  587829252;     // mov #153, reg4
hardware_memory[69] =  587833349;     // mov #154, reg5
hardware_memory[70] =  587837446;     // mov #155, reg6
hardware_memory[71] =  587841543;     // mov #156, reg7
hardware_memory[72] =  587845640;     // mov #157, reg8
hardware_memory[73] =  587849737;     // mov #158, reg9
hardware_memory[74] =  587853837;     // mov #159, reg13
hardware_memory[75] =  587857935;     // mov #160, reg15
hardware_memory[76] =  620756992;     // ret
//!read_reg
hardware_memory[77] =  1124073622;   // mov reg0, #150
hardware_memory[78] =  1124077719;   // mov reg1, #151
hardware_memory[79] =  1124081816;   // mov reg2, #152
hardware_memory[80] =  1124090009;   // mov reg4, #153
hardware_memory[81] =  1124094106;   // mov reg5, #154
hardware_memory[82] =  1124098203;   // mov reg6, #155
hardware_memory[83] =  1124102300;   // mov reg7, #156
hardware_memory[84] =  1124106397;   // mov reg8, #157
hardware_memory[85] =  1124110494;   // mov reg9, #158
hardware_memory[86] =  1124126879;   // mov reg13, #159
hardware_memory[87] =  1124135072;   // mov reg15, #160
hardware_memory[88] =  620756992;     // ret
// Sort
hardware_memory[89] = 1392508928;    // mov reg0, #0;       # init
hardware_memory[90] = 1392513024;    // mov reg1, #0;
hardware_memory[91] = 1392517120;    // mov reg2, #0;
hardware_memory[92] = 855691267;      // mov reg13, reg3;    # n - 2
hardware_memory[93] = 1375784960;    // dec reg13;
hardware_memory[94] = 855654403;      // mov reg4, reg3;
hardware_memory[95] = 1375748096;    // dec reg4;              # n-1
hardware_memory[96] = 1392529536;    // mov reg5, #128;      # start address
hardware_memory[97] = 1392570369;    // mov reg15, #1;
hardware_memory[98] = 84332544;       // jmp X3(109);
hardware_memory[99] = 1107296256;    // inc reg0;              # j++                !X1
hardware_memory[100] = 134217741;    // fbs reg0, reg13;      # do while j <= n-2
hardware_memory[101] = 352325756;    // jmp fbs, X6(124);
hardware_memory[102] = 1392517120;  // mov reg2, #0;         # f = 0
hardware_memory[103] = 1392513024;  // mov reg1, #0;         # i = 0
hardware_memory[104] = 1375748096;  // dec reg4;              # n - 1 - j
hardware_memory[105] = 84332544;     // jmp X3(109);
hardware_memory[106] = 1107300352;  // inc reg1               # i++                !X2
hardware_memory[107] = 134221828;    // fbs reg1, reg4         # do while i <= n - 1 - j
hardware_memory[108] = 352325731;    // jmp fbs, X1(99)
hardware_memory[109] = 855662593;    // mov reg6, reg1;       # i        A[i]      !X3
hardware_memory[110] = 33579013;      // add reg6, reg5;       # i + a0
hardware_memory[111] = 117469190;    // movx reg7, reg6;     # memory[i + a0]
hardware_memory[112] = 855670790;    // mov reg8, reg6;       # i + a0           A[i + 1]
hardware_memory[113] = 1107329024;  // inc reg8;              # i + a0 + 1
hardware_memory[114] = 117477384;    // movx reg9, reg8;     # memory[i + a0 + 1]
```

```
    hardware_memory[115] = 134246409;    // fbs reg7, reg9;
    hardware_memory[116] = 352325750;    // jmp fbs, X4(118);
    hardware_memory[117] = 84381696;     // jmp X5(121);
    hardware_memory[118] = 654348294;    // movm reg9, reg6;   # swap            !X4
    hardware_memory[119] = 654340104;    // movm reg7, reg8;
    hardware_memory[120] = 1392517121;   // mov reg2, #1;      # f = 1
    hardware_memory[121] = 1207967759;   // fes reg2, reg15;   #                 !X5
    hardware_memory[122] = 352325738;    // jmp fes, X2(106);  # if f == 1 the cycle is cont
    hardware_memory[123] = 84291584;     // jmp X1(99);
    hardware_memory[124] = 620756992;    // ret                #                 !X6
```

Алгоритм фильтрует заданным двумерный массив (в данном конкретном случае это массив 10 на 10, записанный по адрес 200). Код теста приведен в Приложении 10.

5. Заключение

Результатом курсовой работы стал микропроцессор, состоящий из 6 блоков, и набор программ, реализующих заданные алгоритмы (сортировку, умножение матриц, медианную фильтрацию двумерного массива). Микропроцессор имеет порядка 40 команд, в том числе вызов подфункции (стек глубиной 4), набор математических, логических операций, и большой набор команд пересылок данных.

Для каждого модуля был написан тест, демонстрирующий его работу. Кроме того, был написан тест всего микропроцессора, демонстрирующий работы всех модулей в всязке.

Разработанные программы, на основе заданных алгоритмов были проверены на 2 наборах: набор небольшой и средне размерности. Сортировка 4 элементов и 201.Умножение матриц 2x2 и 10x10. И Медианная фильтрация массива размерностью 10x10. На основе корректной работы написанных программ, можно сделать вывод, что микропропроцессор работает правильно, без ошибок.

Курсовая работа значительно повысила навыки работы с библиотекой systemC.

6. Приложения

Приложение 1. Тестирование виртуальной памяти

```
#include "systemc.h"
#include "Virtual_memory.h"

#define soft_assert(signal, expected) \
    if (signal.read() != expected) { \
        cerr << "@" << sc_time_stamp() << " Check failed. Expected: " << expected << ". Actual: 
" << signal.read() << ".\n" << endl; \
    }

sc_clock clk("clock", 4, SC_NS);
sc_signal<bool> sreset;                // restart

sc_signal<bool> ch_in_read;                  // read from cell
sc_signal<bool> ch_in_write;                 // write in cell
sc_signal<sc_uint<12> > ch_in_addr_cell;
sc_signal<sc_uint<32> > ch_in_data_to_cell;
sc_signal<sc_uint<32> > ch_out_data_from_cell;
sc_signal<bool> ch_out_ready;
sc_signal<bool> ch_in_save_all;

// to ram memory
sc_signal<bool> ram_out_read;               // read from cell
sc_signal<bool> ram_out_write;              // write in cell
sc_signal<sc_uint<10> > ram_out_addr_cell;
sc_signal<sc_uint<32> > ram_out_data_to_cell;
sc_signal<sc_uint<32> > ram_in_data_from_cell;
sc_signal<bool> ram_in_ready;

// to hard memory
sc_signal<bool> hm_out_read;               // read from cell
sc_signal<bool> hm_out_write;              // write in cell
sc_signal<sc_uint<12> > hm_out_addr_cell;
sc_signal<sc_uint<32> > hm_out_data_to_cell;
sc_signal<sc_uint<32> > hm_in_data_from_cell;
sc_signal<bool> hm_in_ready;

// memory
int ram_memory[1024];
int hardware_memory[4096];



int TIMEOUT_RAM = 4;
int TIMEOUT_HARD_MEMORY = 7;
int CYCLE_INDEX = TIMEOUT_HARD_MEMORY + 1;
int ram_counter = 0;
int hm_counter = 0;
```

```
bool ram_run = false, hm_run = false;
bool buffer_ram_read = false, buffer_ram_write = false, buffer_hm_read = false,
buffer_hm_write = false;

void setSignal(){

   if(ram_out_read || ram_out_write){
      ram_run = true;
      ram_in_ready = false;
      buffer_ram_read = ram_out_read.read();
      buffer_ram_write = ram_out_write.read();
   }

   if(ram_run){
      ram_counter++;
   }

   if(ram_counter == TIMEOUT_RAM && buffer_ram_read)
      ram_in_data_from_cell = ram_memory[ram_out_addr_cell.read()];


   if(ram_counter == TIMEOUT_RAM && buffer_ram_write)
      ram_memory[ram_out_addr_cell.read()] = ram_out_data_to_cell.read();


   if(ram_counter == TIMEOUT_RAM){
      ram_counter = 0;
      ram_run = false;
      buffer_ram_read = false;
      buffer_ram_write = false;
      ram_in_ready = true;
   }

   if(hm_run){
      hm_counter++;
   }

   if(hm_out_read || hm_out_write){
      hm_run = true;
      hm_in_ready = false;
      buffer_hm_read = hm_out_read.read();
      buffer_hm_write = hm_out_write.read();
   }

   if(hm_counter == TIMEOUT_HARD_MEMORY && buffer_hm_read){
      hm_in_data_from_cell = hardware_memory[hm_out_addr_cell.read()];
   }
   if(hm_counter == TIMEOUT_HARD_MEMORY && buffer_hm_write)
      hardware_memory[hm_out_addr_cell.read()] = hm_out_data_to_cell.read();

   if(hm_counter == TIMEOUT_HARD_MEMORY){
      hm_counter = 0;
      hm_run = false;
      buffer_hm_read = false;
      buffer_hm_write = false;
      hm_in_ready = true;
   }

}

void printRAM(){
   cout
<<"\n\n\n\n******************************printRAM************************"
<<endl;
   int i,j;
   for(j = 0; j < 32; j++){
      for(i = 0; i < 32; i++){
         cout<< ram_memory[j*32 + i] <<" ";
      }
      if((j*32 + i) % 256 == 0)
         cout<<"_"<< endl;
      else
         cout<<endl;
   }
}

void printHARDWARE(){
   cout
<<"\n\n\n\n******************************printHARDWARE*******************"
<<endl;
   int i,j;
   for(j = 0; j < 128; j++){
      for(i = 0; i < 32; i++){
         cout<< hardware_memory[j*32 + i] <<" ";
      }
      if((j*32 + i) % 256 == 0)
         cout<<"_"<< endl;
      else
         cout<<endl;
   }
}
int sc_main(int argc, char* argv[]) {



   Virtual_memory vm("Virtual_memory");
```

```
vm.clk(clk);
vm.sreset(sreset);
vm.ch_in_read(ch_in_read);
vm.ch_in_write(ch_in_write);
vm.ch_in_addr_cell(ch_in_addr_cell);
vm.ch_in_data_to_cell(ch_in_data_to_cell);
vm.ch_out_data_from_cell(ch_out_data_from_cell);
vm.ch_out_ready(ch_out_ready);
vm.ch_in_save_all(ch_in_save_all);
vm.ram_out_read(ram_out_read);
vm.ram_out_write(ram_out_write);
vm.ram_out_addr_cell(ram_out_addr_cell);
vm.ram_out_data_to_cell(ram_out_data_to_cell);
vm.ram_in_data_from_cell(ram_in_data_from_cell);
vm.ram_in_ready(ram_in_ready);
vm.hm_out_read(hm_out_read);
vm.hm_out_write(hm_out_write);
vm.hm_out_addr_cell(hm_out_addr_cell);
vm.hm_out_data_to_cell(hm_out_data_to_cell);
vm.hm_in_data_from_cell(hm_in_data_from_cell);
vm.hm_in_ready(hm_in_ready);

sc_trace_file *wf = sc_create_vcd_trace_file("MPC_waveform");
// Dump the desired signals
sc_trace(wf, clk, "clock");
sc_trace(wf, sreset, "sreset");
sc_trace(wf, ch_in_read, "ch_in_read");
sc_trace(wf, ch_in_write, "ch_in_write");
sc_trace(wf, ch_in_addr_cell, "ch_in_addr_cell");
sc_trace(wf, ch_in_data_to_cell, "ch_in_data_to_cell");
sc_trace(wf, ch_out_data_from_cell, "ch_out_data_from_cell");
sc_trace(wf, ch_out_ready, "ch_out_ready");
sc_trace(wf, ch_in_save_all, "ch_in_save_all");
sc_trace(wf, ram_out_read, "ram_out_read");
sc_trace(wf, ram_out_write, "ram_out_write");
sc_trace(wf, ram_out_addr_cell, "ram_out_addr_cell");
sc_trace(wf, ram_out_data_to_cell, "ram_out_data_to_cell");
sc_trace(wf, ram_in_data_from_cell, "ram_in_data_from_cell");
sc_trace(wf, ram_in_ready, "ram_in_ready");
sc_trace(wf, hm_out_read, "hm_out_read");
sc_trace(wf, hm_out_write, "hm_out_write");
sc_trace(wf, hm_out_addr_cell, "hm_out_addr_cell");
sc_trace(wf, hm_out_data_to_cell, "hm_out_data_to_cell");
sc_trace(wf, hm_in_data_from_cell, "hm_in_data_from_cell");
sc_trace(wf, hm_in_ready, "hm_in_ready");
sreset = false;

//init hardware memory
for(int i = 0; i < 4096; i ++)
```

```
    hardware_memory[i] = i;

  // nothing
  sc_start(4, SC_NS);
  hm_in_ready = true;             // real memory is ready to work
  ram_in_ready = true;
  sc_start(11, SC_NS);

  //1 READ
*****************************************************************
  sc_start(4, SC_NS);

  ch_in_read = true;              // set signal by reading
  ch_in_addr_cell = 10;             // virtual addr a cell, which should be reading

  sc_start(4, SC_NS);

  // reset input signals (show, that signals is not up all time)
  ch_in_read = false;
  ch_in_addr_cell = 0;

  // read page from hardware memory
  for(int i = 0; i < 260 * CYCLE_INDEX; i ++){
    setSignal();
    sc_start(4, SC_NS);
  }

  // 2 WRITE WITHOU LOADING NEW PAGE
****************************************
  ch_in_write = true;             // set signal by reading
  ch_in_addr_cell = 50;             // virtual addr a cell, which should be reading
  ch_in_data_to_cell = 2222222;


  sc_start(4, SC_NS);
  setSignal();

  //3 WRITE WITH LOADING NEW PAGE
*****************************************************************
  ch_in_write = true;              // set signal by reading
  ch_in_addr_cell = 2010;             // virtual addr a cell, which should be reading
  ch_in_data_to_cell = 33333333;
  sc_start(4, SC_NS);

  // reset input signals (show, that signals is not up all time)
  ch_in_write = false;
  ch_in_addr_cell = 0;
  ch_in_data_to_cell = 0;
  // read page from hardware memory
  for(int i = 0; i < 260 * CYCLE_INDEX; i ++){
     setSignal();
     sc_start(4, SC_NS);
  }

  sc_start(8, SC_NS);

  //4 WRITE WITH LOADING NEW PAGE
*****************************************************************
  ch_in_write = true;                 // set signal by reading
  ch_in_addr_cell = 4095;             // virtual addr a cell, which should be reading
  ch_in_data_to_cell = 4444444;
  sc_start(4, SC_NS);
  // reset input signals (show, that signals is not up all time)
  ch_in_write = false;
  ch_in_addr_cell = 0;
  ch_in_data_to_cell = 0;
  // read page from hardware memory
  for(int i = 0; i < 260 * CYCLE_INDEX; i ++){
     setSignal();
     sc_start(4, SC_NS);
  }

  sc_start(8, SC_NS);

  //5 WRITE WITH LOADING NEW PAGE
*****************************************************************
  ch_in_write = true;                 // set signal by reading
  ch_in_addr_cell = 700;              // virtual addr a cell, which should be reading
  ch_in_data_to_cell = 5555555;
  sc_start(4, SC_NS);

  // reset input signals (show, that signals is not up all time)
  ch_in_write = false;
  ch_in_addr_cell = 0;
  ch_in_data_to_cell = 0;

  // read page from hardware memory
  for(int i = 0; i < 260 * CYCLE_INDEX; i ++){
     setSignal();
     sc_start(4, SC_NS);
  }

  sc_start(8, SC_NS);

 // printHARDWARE();
 // printRAM();
```

```
  //6 READ
*****************************************************************
  ch_in_read = true;               // set signal by reading
  ch_in_addr_cell = 1400;           // virtual addr a cell, which should be reading
  sc_start(4, SC_NS);

  // reset input signals (show, that signals is not up all time)
  ch_in_read = false;
  ch_in_addr_cell = 0;
  // read page from hardware memory
  for(int i = 0; i < 520 * CYCLE_INDEX; i ++){
    setSignal();
    sc_start(4, SC_NS);
  }
  sc_start(20, SC_NS);

  //6 READ
*****************************************************************
  ch_in_read = true;               // set signal by reading
  ch_in_addr_cell = 813;           // virtual addr a cell, which should be reading
  sc_start(4, SC_NS);
  // reset input signals (show, that signals is not up all time)
  ch_in_read = false;
  ch_in_addr_cell = 0;
  // read page from hardware memory
  for(int i = 0; i < 520 * CYCLE_INDEX; i ++){
    setSignal();
    sc_start(4, SC_NS);
  }
  sc_start(20, SC_NS);

  //6 SAVE
*****************************************************************
  ch_in_save_all = true;               // set signal by reading
  sc_start(4, SC_NS);

  // reset input signals (show, that signals is not up all time)
  ch_in_save_all = false;

  // save pages to hardware memory
  for(int i = 0; i < 780 * CYCLE_INDEX; i ++){
    setSignal();
    sc_start(4, SC_NS);
  }
  printHARDWARE();
  printRAM();
  return 0;
}
```

```
#include "systemc.h"
#include "Cache.h"

#define soft_assert(signal, expected) \
    if (signal.read() != expected) { \
        cerr << "@" << sc_time_stamp() << " Check failed. Expected: " << expected << ". Actual:
" << signal.read() << ".\n" << endl; \
    }

sc_clock clk("clock", 4, SC_NS);
sc_signal<bool> sreset;                          // restart

// to memory controller
sc_signal<bool> mc_read;                         // read from cache cell
sc_signal<bool> mc_write;                        // write to cache in cell
sc_signal<sc_uint<12> > mc_addr_cell;            // addr needing memory cell
sc_signal<sc_uint<32> > mc_data_to_cell;         // data wrote in memory cell
sc_signal<sc_uint<32> > mc_data_from_cell;       // data from memory cell
sc_signal<bool> mc_ready;
sc_signal<bool> mc_save_all;

// to virtual memory
sc_signal<bool> m_read;                          // if data in cache not exit, get it from memory
sc_signal<bool> m_write;
sc_signal<sc_uint<12> > m_addr_cell;             // addr needing memory cell
sc_signal<sc_uint<32> > m_data_to_cell;          // data wrote in memory cell
sc_signal<sc_uint<32> > m_data_from_cell;        // data from memory cell
sc_signal<bool> m_ready;
sc_signal<bool> m_save_all;

Cache ch("Cache");

int MEMORY_SPEED = 7;
int CYCLE_INDEX = MEMORY_SPEED + 1;
int clock_counter = 0;
bool run = false;
bool buffer_read = false;
int mumber = 0;
void setSignal(){

  if(m_read || m_write){
    run = true;
    m_ready = false;
    buffer_read = m_read.read();
  }

  if(run)
    clock_counter++;
  if(clock_counter == MEMORY_SPEED && buffer_read)
    m_data_from_cell = mumber++;

  if(clock_counter == MEMORY_SPEED){
    clock_counter = 0;
    run = false;
    buffer_read = false;
    m_ready = true;
  }

}

void printCache(){
  for(int i = 0; i < 32; i++){
    for(int j = 0; j < 18; j++)
      cout << ch.cache_memory[i][j]<< " ";
    cout<<endl;
  }
}
int sc_main(int argc, char* argv[]) {


  ch.clk(clk);
  ch.sreset(sreset);

  ch.mc_read(mc_read);
  ch.mc_write(mc_write);
  ch.mc_addr_cell(mc_addr_cell);
  ch.mc_data_to_cell(mc_data_to_cell);
  ch.mc_data_from_cell(mc_data_from_cell);
  ch.mc_ready(mc_ready);
  ch.mc_save_all(mc_save_all);

  ch.m_read(m_read);
  ch.m_write(m_write);
  ch.m_addr_cell(m_addr_cell);
  ch.m_data_to_cell(m_data_to_cell);
  ch.m_data_from_cell(m_data_from_cell);
  ch.m_ready(m_ready);
  ch.m_save_all(m_save_all);

  sc_trace_file *wf = sc_create_vcd_trace_file("MPC_waveform");
  // Dump the desired signals
  sc_trace(wf, clk, "clock");
  sc_trace(wf, sreset, "sreset");
  sc_trace(wf, mc_read, "mc_read");
  sc_trace(wf, mc_write, "mc_write");
  sc_trace(wf, mc_addr_cell, "mc_addr_cell");
```

```
   sc_trace(wf, mc_data_to_cell, "mc_data_to_cell");
   sc_trace(wf, mc_data_from_cell, "mc_data_from_cell");
   sc_trace(wf, mc_ready, "mc_ready");
   sc_trace(wf, m_read, "m_read");
   sc_trace(wf, m_write, "m_write");
   sc_trace(wf, m_ready, "m_ready");
   sc_trace(wf, m_addr_cell, "m_addr_cell");
   sc_trace(wf, m_data_to_cell, "m_data_to_cell");
   sc_trace(wf, m_data_from_cell, "m_data_from_cell");
   sc_trace(wf, mc_save_all, "mc_save_all");
   sc_trace(wf, m_save_all, "m_save_all");

   sreset = false;
   m_ready = false;
   // nothing
   sc_start(7*CYCLE_INDEX, SC_NS);

   //1 READ FIRST BLOCK IN CACHE, BECAUSE IT EMPTY ****************************
   // show, that vm don`t work if physical memory is not ready
   sc_start(4, SC_NS);
   // now is work
   m_ready = true;          // real memory is ready to work

   // load data in cache-block
   sc_start(4, SC_NS);
   mc_addr_cell = 0;
   for(int i = 0; i < 17 * CYCLE_INDEX; i ++){
     setSignal();
     sc_start(4, SC_NS);
   }



   //2
WRITE*****************************************************************
   // show, that vm don`t work if physical memory is not ready
   // set the input singals
   mc_write = true;        // set signal by writing
   mc_addr_cell = 2;       // virtual addr a cell, which should be reading
   mc_data_to_cell = 798;
   //reset input singals
   sc_start(4, SC_NS);
   mc_write = false;
   mc_addr_cell = 0;
   mc_data_to_cell = 0;
   sc_start(8, SC_NS);

   //3. WRITE DATA WITH RELOAD CACHE
*******************************************
   mc_write = true;
```

```
    mc_addr_cell = 514;
    mc_data_to_cell = 111;
    //reset input singals
    sc_start(4, SC_NS);
    mc_write = false;
    mc_addr_cell = 0;
    mc_data_to_cell = 0;
    // read new data from memory
    for(int i = 0; i < 30 * CYCLE_INDEX; i ++){
       setSignal();
       sc_start(4, SC_NS);
    }

    //4. SAVE DATA TO MEMORY
    mc_save_all = true;
    sc_start(4, SC_NS);
    mc_save_all = false;
    // write  data to memory
    for(int i = 0; i < 15 * CYCLE_INDEX; i ++){
       setSignal();
       sc_start(4, SC_NS);
    }
    printCache();
    return 0;
}
```

```
#include "systemc.h"
#include "Command_decoder.h"

#define soft_assert(signal, expected) \
     if (signal.read() != expected) { \
        cerr << "@" << sc_time_stamp() << " Check failed. Expected: " << expected << ". Actual:
" << signal.read() << ".\n" << endl; \
     }

int sc_main(int argc, char* argv[]) {
   sc_clock clk("clock", 4, SC_NS);
   sc_signal<bool> sreset;                    // restart

   // to memory controller and alu
   sc_signal<sc_uint<8>> cmd;
   sc_signal<sc_uint<12>> operand1;
   sc_signal<sc_uint<12>> operand2;
   sc_signal<sc_uint<32>> data_cell;

   Command_decoder cd("Command_decoder");
   cd.clk(clk);
   cd.sreset(sreset);
   cd.cmd(cmd);
   cd.operand1(operand1);
   cd.operand2(operand2);
   cd.data_cell(data_cell);

   sc_trace_file *wf = sc_create_vcd_trace_file("MPC_waveform");
   // Dump the desired signals
   sc_trace(wf, clk, "clock");
   sc_trace(wf, sreset, "sreset");
   sc_trace(wf, cmd, "cmd");
   sc_trace(wf, operand1, "operand1");
   sc_trace(wf, operand2, "operand2");
   sc_trace(wf, data_cell, "data_cell");
   sreset = false;
   data_cell = 0;   // nothing
   sc_start(7, SC_NS);
   // data
   data_cell = 79879787;
   sc_start(4, SC_NS);
   // data 2
   data_cell = 162345687;
   sc_start(4, SC_NS);
   sc_start(4, SC_NS);
   return 0;
}
```

```
#include "systemc.h"
#include "ALU.h"

#define soft_assert(signal, expected) \
      if (signal.read() != expected) { \
          cerr << "@" << sc_time_stamp() << " Check failed. Expected: " << expected << ". Actual: 
" << signal.read() << ".\n" << endl; \
      }

int sc_main(int argc, char* argv[]) {
    sc_clock clk("clock", 4, SC_NS);
    sc_signal<bool> sreset;                    // restart

    // to command decoder
    sc_signal<sc_uint<8>> cmd;
    sc_signal<sc_uint<12>> operand1;
    sc_signal<sc_uint<12>> operand2;

    // memory controller
    sc_signal<sc_uint<4>> alu_reg_id;
    sc_signal<bool> alu_reg_read;              // read from cache cell
    sc_signal<bool> alu_reg_write;             // write to cache in cell
    sc_signal<sc_uint<32>> alu_reg_data_to;
    sc_signal<sc_uint<32>> alu_reg_data_from;
    sc_signal<bool> alu_fbs;
    sc_signal<bool> alu_fss;
    sc_signal<bool> alu_fes;
    sc_signal<bool> alu_fnes;
    sc_signal<bool> alu_fbes;
    sc_signal<bool> alu_fses;

    Alu al("ALu");
    al.clk(clk);
    al.sreset(sreset);
    al.cmd(cmd);
    al.operand1(operand1);
    al.operand2(operand2);
    al.alu_reg_id(alu_reg_id);
    al.alu_reg_read(alu_reg_read);
    al.alu_reg_write(alu_reg_write);
    al.alu_reg_data_to(alu_reg_data_to);
    al.alu_reg_data_from(alu_reg_data_from);
    al.alu_fbs(alu_fbs);
    al.alu_fss(alu_fss);
    al.alu_fes(alu_fes);
    al.alu_fnes(alu_fnes);
    al.alu_fbes(alu_fbes);
    al.alu_fses(alu_fses);
```

```
sc_trace_file *wf = sc_create_vcd_trace_file("MPC_waveform");
// Dump the desired signals
sc_trace(wf, clk, "clock");
sc_trace(wf, sreset, "sreset");
sc_trace(wf, cmd, "cmd");
sc_trace(wf, operand1, "operand1");
sc_trace(wf, operand2, "operand2");
sc_trace(wf, alu_reg_id, "alu_reg_id");
sc_trace(wf, alu_reg_read, "alu_reg_read");
sc_trace(wf, alu_reg_write, "alu_reg_write");
sc_trace(wf, alu_reg_data_to, "alu_reg_data_to");
sc_trace(wf, alu_reg_data_from, "alu_reg_data_from");
sc_trace(wf, alu_fbs, "alu_fbs");
sc_trace(wf, alu_fss, "alu_fss");
sc_trace(wf, alu_fes, "alu_fes");
sc_trace(wf, alu_fnes, "alu_fnes");
sc_trace(wf, alu_fbes, "alu_fbes");
sc_trace(wf, alu_fses, "alu_fses");

sreset = false;
cmd = 0;
operand1 = 0;
operand2 = 0;

// nothing
sc_start(4, SC_NS);

// write data in reg1
alu_reg_id = 1;
alu_reg_write = true;
alu_reg_data_to = 21;
sc_start(4, SC_NS);

// write data in reg2
alu_reg_id = 2;
alu_reg_data_to = 34;
sc_start(4, SC_NS);

// read data from reg0
alu_reg_id = 1;
alu_reg_read = true;
alu_reg_write = false;        //reset the write signal

//input cmd and operands
operand1 = 1;                 // id regs
operand2 = 2;
```

```
   // Mathematical command
   cmd = 2;                    // add
   sc_start(4, SC_NS);
   cmd = 34;                   // mult
   sc_start(4, SC_NS);
   cmd = 18;                   // sub
   sc_start(4, SC_NS);
   cmd = 50;                   // div
   sc_start(4, SC_NS);
   cmd = 66;                   // inc
   sc_start(4, SC_NS);
   cmd = 82;                   // dec
   sc_start(4, SC_NS);
   cmd = 0;
   sc_start(8, SC_NS);

   // boolean cmd
   cmd = 4;                    // xor
   sc_start(4, SC_NS);
   cmd = 36;                   // or
   sc_start(4, SC_NS);
   cmd = 20;                   // and
   sc_start(4, SC_NS);
   cmd = 52;                   // not
   sc_start(4, SC_NS);
   cmd = 52;                   // not
   sc_start(4, SC_NS);
   cmd = 0;
   sc_start(8, SC_NS);

   // Inequality cmd
   cmd = 72;                   // first equal second ?
   sc_start(4, SC_NS);
   cmd = 22;                   // reg1 << 1
   sc_start(4, SC_NS);
   cmd = 8;                    // reg1 > reg2 ?
   sc_start(4, SC_NS);
   cmd = 0;
   sc_start(8, SC_NS);

   return 0;
}
```

```
#include "systemc.h"
#include "io_module.h"

#define soft_assert(signal, expected) \
      if (signal.read() != expected) { \
         cerr << "@" << sc_time_stamp() << " Check failed. Expected: " << expected << ". Actual:
" << signal.read() << ".\n" << endl; \
      }

int sc_main(int argc, char* argv[]) {
   sc_clock clk("clock", 4, SC_NS);
   sc_signal<bool> sreset;                    // restart

   sc_signal<bool> io_read;
   sc_signal<bool> io_write;
   sc_signal<sc_uint<12> > io_data_to_reg;
   sc_signal<sc_uint<12> > io_data_from_reg;

   // from user
   sc_signal<bool> user_read;
   sc_signal<bool> user_write;
   sc_signal<sc_uint<12> > user_data_to_reg;
   sc_signal<sc_uint<12> > user_data_from_reg;

   IO io("IO");
   io.clk(clk);
   io.sreset(sreset);
   io.io_read(io_read);
   io.io_write(io_write);
   io.io_data_to_reg(io_data_to_reg);
   io.io_data_from_reg(io_data_from_reg);
   io.user_read(user_read);
   io.user_write(user_write);
   io.user_data_to_reg(user_data_to_reg);
   io.user_data_from_reg(user_data_from_reg);

   sc_trace_file *wf = sc_create_vcd_trace_file("MPC_waveform");
   // Dump the desired signals
   sc_trace(wf, clk, "clock");
   sc_trace(wf, sreset, "sreset");
   sc_trace(wf, io_read, "io_read");
   sc_trace(wf, io_write, "io_write");
   sc_trace(wf, io_data_to_reg, "io_data_to_reg");
   sc_trace(wf, io_data_from_reg, "io_data_from_reg");
   sc_trace(wf, user_read, "user_read");
   sc_trace(wf, user_write, "user_write");
   sc_trace(wf, user_data_to_reg, "user_data_to_reg");
   sc_trace(wf, user_data_from_reg, "user_data_from_reg");
```

```
    sreset = false;
    // nothing
    sc_start(7, SC_NS);

    // MP is write, user is read,
    io_write = true;
    io_data_to_reg = 12;

    sc_start(4, SC_NS);
    io_write = false;

    sc_start(3, SC_NS);
    user_read = true;

    sc_start(12, SC_NS);

    // user is write, MP is read
    user_read = false;
    user_write = true;
    user_data_to_reg = 20;
    io_read = true;
    sc_start(4, SC_NS);
    user_write = false;
    sc_start(12, SC_NS);

    return 0;
}
```

```
#include "systemc.h"
#include "Memory_controller.h"

#define soft_assert(signal, expected) \
    if (signal.read() != expected) { \
        cerr << "@" << sc_time_stamp() << " Check failed. Expected: " << expected << ". Actual:
" << signal.read() << ".\n" << endl; \
    }

int sc_main(int argc, char* argv[]) {
  sc_clock clk("clock", 4, SC_NS);
  sc_signal<bool> sreset;                    // restart

  // to cache
  sc_signal<bool> ch_read;                   // read from cache cell
  sc_signal<bool> ch_write;                  // write to cache in cell
  sc_signal<sc_uint<12> > ch_addr_cell;      // addr needing memory cell
  sc_signal<sc_uint<32> > ch_data_to_cell;   // data wrote in memory cell
  sc_signal<sc_uint<32> > ch_data_from_cell; // data from memory cell
  sc_signal<bool> ch_ready;
  sc_signal<bool> ch_save_data;

  // to command decoder
  sc_signal<sc_uint<8>> cd_cmd;
  sc_signal<sc_uint<12>> cd_operand1;
  sc_signal<sc_uint<12>> cd_operand2;
  sc_signal<sc_uint<32>> cd_data_cell;

  // alu
  //--registers
  sc_signal<sc_uint<4>> alu_reg_id;
  sc_signal<bool> alu_reg_read;              // read from cache cell
  sc_signal<bool> alu_reg_write;             // write to cache in cell
  sc_signal<sc_uint<32>> alu_reg_data_to;
  sc_signal<sc_uint<32>> alu_reg_data_from;
  //--flags
  sc_signal<bool> alu_fbs;
  sc_signal<bool> alu_fss;
  sc_signal<bool> alu_fes;
  sc_signal<bool> alu_fnes;
  sc_signal<bool> alu_fbes;
  sc_signal<bool> alu_fses;


  //io_module
  sc_signal<bool> io_read;
  sc_signal<bool> io_write;
  sc_signal<sc_uint<12> > io_data_to_reg;
```

```
sc_signal<sc_uint<12> > io_data_from_reg;

Memory_controller mc("Memory_controller");
mc.clk(clk);
mc.sreset(sreset);
// to cache
mc.ch_read(ch_read);
mc.ch_write(ch_write);
mc.ch_addr_cell(ch_addr_cell);
mc.ch_data_to_cell(ch_data_to_cell);
mc.ch_data_from_cell(ch_data_from_cell);
mc.ch_ready(ch_ready);
mc.ch_save_data(ch_save_data);

// to command decoder
mc.cd_cmd(cd_cmd);
mc.cd_operand1(cd_operand1);
mc.cd_operand2(cd_operand2);
mc.cd_data_cell(cd_data_cell);

// to alu
mc.alu_reg_id(alu_reg_id);
mc.alu_reg_read(alu_reg_read);
mc.alu_reg_write(alu_reg_write);
mc.alu_reg_data_to(alu_reg_data_to);
mc.alu_reg_data_from(alu_reg_data_from);
mc.alu_fbs(alu_fbs);
mc.alu_fss(alu_fss);
mc.alu_fbes(alu_fbes);
mc.alu_fses(alu_fses);
mc.alu_fes(alu_fes);
mc.alu_fnes(alu_fnes);

// to io
mc.io_read(io_read);
mc.io_write(io_write);
mc.io_data_to_reg(io_data_to_reg);
mc.io_data_from_reg(io_data_from_reg);

sc_trace_file *wf = sc_create_vcd_trace_file("MPC_waveform");

// Dump the desired signals
sc_trace(wf, clk, "clock");
sc_trace(wf, sreset, "sreset");
// to cache
sc_trace(wf, ch_read, "ch_read");
sc_trace(wf, ch_write, "ch_write");
sc_trace(wf, ch_addr_cell, "ch_addr_cell");
sc_trace(wf, ch_data_to_cell, "ch_data_to_cell");
```

```
   sc_trace(wf, ch_data_from_cell, "ch_data_from_cell");
   sc_trace(wf, ch_ready, "ch_ready");
   sc_trace(wf, ch_save_data, "ch_save_data");

   // to command decoder
   sc_trace(wf, cd_cmd, "cd_cmd");
   sc_trace(wf, cd_operand1, "cd_operand1");
   sc_trace(wf, cd_operand2, "cd_operand2");
   sc_trace(wf, cd_data_cell, "cd_data_cell");

   // to alu
   sc_trace(wf, alu_reg_id, "alu_reg_id");
   sc_trace(wf, alu_reg_read, "alu_reg_read");
   sc_trace(wf, alu_reg_write, "alu_reg_write");
   sc_trace(wf, alu_reg_data_to, "alu_reg_data_to");
   sc_trace(wf, alu_reg_data_from, "alu_reg_data_from");
   sc_trace(wf, alu_fbs, "alu_fbs");
   sc_trace(wf, alu_fss, "alu_fss");
   sc_trace(wf, alu_fbes, "alu_fbes");
   sc_trace(wf, alu_fses, "alu_fses");
   sc_trace(wf, alu_fes, "alu_fes");
   sc_trace(wf, alu_fnes, "alu_fnes");

   // to io
   sc_trace(wf, io_read, "io_read");
   sc_trace(wf, io_write, "io_write");
   sc_trace(wf, io_data_to_reg, "io_data_to_reg");
   sc_trace(wf, io_data_from_reg, "io_data_from_reg");

   ch_ready = false;              // memory is not ready
   sc_start(15, SC_NS);
   // set signals to get cmd-cell
   ch_ready = true;               // at some point the memory becomes ready
   sc_start(4, SC_NS);
   ch_ready = false;              // memory is busy. it read cmd
   sc_start(28, SC_NS);             // delay


   // 1. (mov addr, #5) READ CMD and execute it
   // set signals "from data"
   ch_ready = true;               // at some point the memory becomes ready again
   ch_data_from_cell = 1426067455;    // and return data from cell
   sc_start(4, SC_NS);             // in the next clock, data-cell will send to command-decoder

   ch_data_from_cell = 0;           // reset signal

   cd_cmd = 3;                    // command-decoder return cmd and 2 operands in the next
moment
   cd_operand1 = 11;
```

```
cd_operand2 = 4095;
sc_start(4, SC_NS);
// reset
cd_cmd = 0;
cd_operand1 = 0;
cd_operand2 = 0;

ch_ready = false;               // memory is busy. it write data (4095) to addr (11)
sc_start(28, SC_NS);              // delay
ch_ready = true;                // at some point the memory becomes ready
sc_start(4, SC_NS);

// 2. (mov reg, addr) READ CMD AND EXECUTE IT
ch_ready = false;               // memory is busy. it read cmd
sc_start(28, SC_NS);              // delay
ch_ready = true;                // at some point the memory becomes ready

// cmd-cell from memory
ch_data_from_cell = 2326067455;
sc_start(4, SC_NS);

//command-decoder send operands and cmd
ch_data_from_cell = 0; // reset
cd_cmd = 67;
cd_operand1 = 1;
cd_operand2 = 456;

sc_start(4, SC_NS);
ch_ready = false;               // memory is busy. it read data from addt(456)
sc_start(28, SC_NS);              // delay
ch_ready = true;                // at some point the memory becomes ready

// data-cell from memory, which will be write in register
cd_cmd = 0;
cd_operand1 = 0;
cd_operand2 = 0;
ch_data_from_cell = 121;
sc_start(4, SC_NS);              // write data from addr to register

// reset
ch_data_from_cell = 0;


//3. movx reg1,reg2 (reg1 = memory[reg2])
//read new command-data-cell
sc_start(4, SC_NS);
ch_ready = false;               // memory is busy. it read data from addt(456)
sc_start(28, SC_NS);              // delay
ch_ready = true;                // at some point the memory becomes ready
```

```
// memory return data-cell
ch_data_from_cell = 141567455;
// reset signals
alu_reg_data_from = 0;
sc_start(4, SC_NS);
//command-decoder send operands and cmd
ch_data_from_cell = 0; // reset
cd_cmd = 7;
cd_operand1 = 1;
cd_operand2 = 2;
sc_start(4, SC_NS);
// reset signal and return data from reg2
cd_cmd = 0;
cd_operand1 = 0;
cd_operand2 = 0;
alu_reg_data_from = 1000;
// read data from memory
sc_start(4, SC_NS);
ch_ready = false;               // memory is busy. it read data from addt(1000)
sc_start(28, SC_NS);              // delay
ch_ready = true;                // at some point the memory becomes ready
// return data from memory
ch_data_from_cell = 94;
sc_start(4, SC_NS);


// 4. Mov reg, ioreg
// read new cmd
sc_start(4, SC_NS);
ch_ready = false;               // memory is busy. it read new cmd
sc_start(28, SC_NS);              // delay
// memory return data-cell
ch_data_from_cell = 65423102;
ch_ready = true;                // at some point the memory becomes ready
sc_start(4, SC_NS);
// comand decoder return result
cd_cmd = 9;
cd_operand1 = 7;
cd_operand2 = 0;
sc_start(4, SC_NS);
// reset signals from deccoder
cd_cmd = 0;
cd_operand1 = 0;
cd_operand2 = 0;
// io return data from io_reg
io_data_from_reg = 56;            // io module set data on output signals
sc_start(4, SC_NS);
```

```
    // 5. Mov ioreg, reg
    // read new cmd
    sc_start(4, SC_NS);
    ch_ready = false;              // memory is busy. it read new cmd
    sc_start(28, SC_NS);             // delay
    ch_ready = true;               // at some point the memory becomes ready
    // data-cell from memory
    ch_data_from_cell = 266544;
    sc_start(4, SC_NS);
    // command decoder result return
    cd_cmd = 25;
    cd_operand1 = 8;
    cd_operand2 = 0;
    sc_start(4, SC_NS);
    // reset signals from deccoder
    cd_cmd = 0;
    cd_operand1 = 0;
    cd_operand2 = 0;
    // alu return data from register 8
    alu_reg_data_from = 91;          // io module set data on output signals
    sc_start(4, SC_NS);

    //6. save all
    sc_start(4, SC_NS);
    ch_ready = false;              // memory is busy. it read new cmd
    sc_start(28, SC_NS);             // delay
    ch_ready = true;               // at some point the memory becomes ready
    // data-cell from memory
    ch_data_from_cell = 885231;
    sc_start(4, SC_NS);
    // command decoder result return
    cd_cmd = 69;
    cd_operand1 = 8;
    cd_operand2 = 0;
    sc_start(40, SC_NS);
    return 0;
}
```

```
#include "systemc.h"
#include "Top.h"

#define soft_assert(signal, expected) \
     if (signal.read() != expected) { \
        cerr << "@" << sc_time_stamp() << " Check failed. Expected: " << expected << ". Actual:
" << signal.read() << ".\n" << endl; \
     }
sc_clock clk("clock", 4, SC_NS);
   sc_signal<bool> sreset;                    // restart

// virtual memory <=> real memory
// to ram memory
sc_signal<bool> ram_out_read;                // read from cell
sc_signal<bool> ram_out_write;               // write in cell
sc_signal<sc_uint<10> > ram_out_addr_cell;
sc_signal<sc_uint<32> > ram_out_data_to_cell;
sc_signal<sc_uint<32> > ram_in_data_from_cell;
sc_signal<bool> ram_in_ready;
// to hard memory
sc_signal<bool> hm_out_read;                 // read from cell
sc_signal<bool> hm_out_write;                // write in cell
sc_signal<sc_uint<12> > hm_out_addr_cell;
sc_signal<sc_uint<32> > hm_out_data_to_cell;
sc_signal<sc_uint<32> > hm_in_data_from_cell;
sc_signal<bool> hm_in_ready;
// User <-> io_module
sc_signal<bool> user_read;
sc_signal<bool> user_write;
sc_signal<sc_uint<12> > user_data_to_reg;
sc_signal<sc_uint<12> > user_data_from_reg;

Top top("Top");

// memory
int ram_memory[1024];
int hardware_memory[4096];


int TIMEOUT_RAM = 4;
int TIMEOUT_HARD_MEMORY = 7;
int CYCLE_INDEX = TIMEOUT_HARD_MEMORY + 1;
int ram_counter = 0;
int hm_counter = 0;
bool ram_run = false, hm_run = false;
bool buffer_ram_read = false, buffer_ram_write = false, buffer_hm_read = false,
buffer_hm_write = false;
```

```
void setSignal(){

   if(ram_out_read || ram_out_write){
      ram_run = true;
      ram_in_ready = false;
      buffer_ram_read = ram_out_read.read();
      buffer_ram_write = ram_out_write.read();
   }

   if(ram_run){
      ram_counter++;
   }

   if(ram_counter == TIMEOUT_RAM && buffer_ram_read)
      ram_in_data_from_cell = ram_memory[ram_out_addr_cell.read()];


   if(ram_counter == TIMEOUT_RAM && buffer_ram_write)
      ram_memory[ram_out_addr_cell.read()] = ram_out_data_to_cell.read();


   if(ram_counter == TIMEOUT_RAM){
      ram_counter = 0;
      ram_run = false;
      buffer_ram_read = false;
      buffer_ram_write = false;
      ram_in_ready = true;
   }

   if(hm_run){
      hm_counter++;
   }

   if(hm_out_read || hm_out_write){
      hm_run = true;
      hm_in_ready = false;
      buffer_hm_read = hm_out_read.read();
      buffer_hm_write = hm_out_write.read();
   }

   if(hm_counter == TIMEOUT_HARD_MEMORY && buffer_hm_read){
      hm_in_data_from_cell = hardware_memory[hm_out_addr_cell.read()];
   }



   if(hm_counter == TIMEOUT_HARD_MEMORY && buffer_hm_write)
      hardware_memory[hm_out_addr_cell.read()] = hm_out_data_to_cell.read();

   if(hm_counter == TIMEOUT_HARD_MEMORY){
        hm_counter = 0;
        hm_run = false;
        buffer_hm_read = false;
        buffer_hm_write = false;
        hm_in_ready = true;
    }

}

void printRAM(){
    cout
<<"\n\n\n\n******************************printRAM************************"
<<endl;
    int i,j;
    for(j = 0; j < 32; j++){
        for(i = 0; i < 32; i++){
            cout<< ram_memory[j*32 + i] <<" ";
        }
        if((j*32 + i) % 256 == 0)
            cout<<"_"<< endl;
        else
            cout<<endl;
    }
}

void printHARDWARE(){
    cout
<<"\n\n\n\n******************************printHARDWARE*******************"
<<endl;
    int i,j;
    for(j = 0; j < 128; j++){
        for(i = 0; i < 32; i++){
            cout<< hardware_memory[j*32 + i] <<" ";
        }
        if((j*32 + i) % 256 == 0)
            cout<<"_"<< endl;
        else
            cout<<endl;
    }
}

int sc_main(int argc, char* argv[]) {

    top.clk(clk);
    top.sreset(sreset);
    // to ram memory
    top.ram_out_read(ram_out_read);
    top.ram_out_write(ram_out_write);
    top.ram_out_addr_cell(ram_out_addr_cell);
    top.ram_out_data_to_cell(ram_out_data_to_cell);
    top.ram_in_data_from_cell(ram_in_data_from_cell);
    top.ram_in_ready(ram_in_ready);
    // to hard memory
    top.hm_out_read(hm_out_read);
    top.hm_out_write(hm_out_write);
    top.hm_out_addr_cell(hm_out_addr_cell);
    top.hm_out_data_to_cell(hm_out_data_to_cell);
    top.hm_in_data_from_cell(hm_in_data_from_cell);
    top.hm_in_ready(hm_in_ready);
    // io_module
    top.user_read(user_read);
    top.user_write(user_write);
    top.user_data_to_reg(user_data_to_reg);
    top.user_data_from_reg(user_data_from_reg);

    sc_trace_file *wf = sc_create_vcd_trace_file("MPC_waveform");
    sc_trace(wf, clk, "0_clock");
    sc_trace(wf, sreset, "0_sreset");
    // virtual memory <=> real memory
    // ram
    sc_trace(wf, ram_out_read, "1_ram_out_read");
    sc_trace(wf, ram_out_write, "2_ram_out_write");
    sc_trace(wf, ram_out_addr_cell, "3_ram_out_addr_cell");
    sc_trace(wf, ram_out_data_to_cell, "4_ram_out_data_to_cell");
    sc_trace(wf, ram_in_data_from_cell, "5_ram_in_data_from_cell");
    sc_trace(wf, ram_in_ready, "6_ram_in_ready");
    // to hard disk
    sc_trace(wf, hm_out_read, "7_hard_out_read");
    sc_trace(wf, hm_out_write, "8_hard_out_write");
    sc_trace(wf, hm_out_addr_cell, "9_hard_out_addr_cell");
    sc_trace(wf, hm_out_data_to_cell, "10_hard_out_data_to_cell");
    sc_trace(wf, hm_in_data_from_cell, "11_hard_in_data_from_cell");
    sc_trace(wf, hm_in_ready, "12_hard_in_ready");
    // virtual memory <-> cache
    sc_trace(wf, top.m_read, "13_memory_read");
    sc_trace(wf, top.m_write, "14_memory_write");
    sc_trace(wf, top.m_addr_cell, "15_memory_addr_cell");
    sc_trace(wf, top.m_data_to_cell, "16_memory_data_to_cell");
    sc_trace(wf, top.m_data_from_cell, "17_memory_data_from_cell");
    sc_trace(wf, top.m_ready, "18_memory_ready");
    sc_trace(wf, top.m_save_data, "19_memory_save_data");
    // cache <-> memory controller
    sc_trace(wf, top.mc_read, "20_cache_read");
    sc_trace(wf, top.mc_write, "21_cache_write");
    sc_trace(wf, top.mc_addr_cell, "22_cache_addr_cell");
    sc_trace(wf, top.mc_data_to_cell, "23_cache_data_to_cell");
    sc_trace(wf, top.mc_data_from_cell, "24_cache_data_from_cell");
    sc_trace(wf, top.mc_ready, "25_cache_ready");
    sc_trace(wf, top.mc_save_data, "26_cache_save_data");
    // command decoder <-> alu and memory controller
    sc_trace(wf, top.cmd, "27_cmd");
    sc_trace(wf, top.operand1, "28_operand1");
    sc_trace(wf, top.operand2, "29_operand2");
    sc_trace(wf, top.data_cell, "30_data_cell");
    // alu <-> memory controller
    sc_trace(wf, top.reg_id, "31_alu_reg_id");
    sc_trace(wf, top.reg_read, "32_alu_reg_read");
    sc_trace(wf, top.reg_write, "33_alu_reg_write");
    sc_trace(wf, top.reg_data_to, "34_alu_reg_data_to");
    sc_trace(wf, top.reg_data_from, "35_alu_reg_data_from");
    sc_trace(wf, top.fbs, "36_alu_fbs");
    sc_trace(wf, top.fss, "37_alu_fss");
    sc_trace(wf, top.fes, "38_alu_fes");
    sc_trace(wf, top.fnes, "39_alu_fnes");
    sc_trace(wf, top.fbes, "40_alu_fbes");
    sc_trace(wf, top.fses, "41_alu_fses");
    // memory controller <-> io_module
    sc_trace(wf, top.io_read, "42_io_read");
    sc_trace(wf, top.io_write, "43_io_write");
    sc_trace(wf, top.io_data_to_reg, "44_io_data_to_reg");
    sc_trace(wf, top.io_data_from_reg, "45_io_data_from_reg");
    // User <-> io_module
    sc_trace(wf, user_read, "46_user_read");
    sc_trace(wf, user_write, "47_user_write");
    sc_trace(wf, user_data_to_reg, "48_user_data_to_reg");
    sc_trace(wf, user_data_from_reg, "49_user_data_from_reg");

    for(int i = 0; i< 4096; i++)
       hardware_memory[i] = i;

    // Пишем в 49 адрес число 4095. После записываем в регистр 1, число 4095,
    // используя косвенную адресацию (49 -> 4095 = 4095). Записываем в регистр 2
    // число, которое храниться в 100 адресе. Затем производиться вычитание
    // регистр1 - регистр2, результат пишеться в регистр1. Значение из регистра 1
    // пишется в адрес 101. Значение из регистра 1 пишется в io-register.
    hardware_memory[0] = 50536447;  // mov addr, #5 (addr = 49, #5 = 4095)
    hardware_memory[1] = 385880113; // movx reg, addr (reg1, addr = 49)
    hardware_memory[2] = 1124081764;// mov reg, addr (reg2, addr = 100)
    hardware_memory[3] = 301993986; // sub reg1, reg2 (reg1 = 4095, reg2 = 100)
    hardware_memory[4] = 587616257; // mov addr, reg (addr = 100, reg = 1)
    hardware_memory[5] = 419434496; // mov ioreg, reg2
    hardware_memory[6] = 1157627904; // end
    // it he beggining
    hm_in_ready = true;
    ram_in_ready = true;

    // read page from hardware memory
```

```
    for(int i = 0; i < 1000 * CYCLE_INDEX; i ++){
        setSignal();
        sc_start(4, SC_NS);
    }

    printHARDWARE();
    return 0;
}
```

```
#include "systemc.h"
#include "Top.h"

#define soft_assert(signal, expected) \
     if (signal.read() != expected) { \
        cerr << "@" << sc_time_stamp() << " Check failed. Expected: " << expected << ". Actual:
" << signal.read() << ".\n" << endl; \
     }
sc_clock clk("clock", 4, SC_NS);
   sc_signal<bool> sreset;                     // restart

// virtual memory <=> real memory
// to ram memory
sc_signal<bool> ram_out_read;                // read from cell
sc_signal<bool> ram_out_write;               // write in cell
sc_signal<sc_uint<10> > ram_out_addr_cell;
sc_signal<sc_uint<32> > ram_out_data_to_cell;
sc_signal<sc_uint<32> > ram_in_data_from_cell;
sc_signal<bool> ram_in_ready;
// to hard memory
sc_signal<bool> hm_out_read;                 // read from cell
sc_signal<bool> hm_out_write;                // write in cell
sc_signal<sc_uint<12> > hm_out_addr_cell;
sc_signal<sc_uint<32> > hm_out_data_to_cell;
sc_signal<sc_uint<32> > hm_in_data_from_cell;
sc_signal<bool> hm_in_ready;
// User <-> io_module
sc_signal<bool> user_read;
sc_signal<bool> user_write;
sc_signal<sc_uint<12> > user_data_to_reg;
sc_signal<sc_uint<12> > user_data_from_reg;

Top top("Top");

// memory
int ram_memory[1024];
int hardware_memory[4096];


int TIMEOUT_RAM = 4;
int TIMEOUT_HARD_MEMORY = 7;
int CYCLE_INDEX = TIMEOUT_HARD_MEMORY + 1;
int ram_counter = 0;
int hm_counter = 0;
bool ram_run = false, hm_run = false;
bool buffer_ram_read = false, buffer_ram_write = false, buffer_hm_read = false,
buffer_hm_write = false;
```

```
void setSignal(){

   if(ram_out_read || ram_out_write){
      ram_run = true;
      ram_in_ready = false;
      buffer_ram_read = ram_out_read.read();
      buffer_ram_write = ram_out_write.read();
   }

   if(ram_run){
      ram_counter++;
   }

   if(ram_counter == TIMEOUT_RAM && buffer_ram_read)
      ram_in_data_from_cell = ram_memory[ram_out_addr_cell.read()];


   if(ram_counter == TIMEOUT_RAM && buffer_ram_write)
      ram_memory[ram_out_addr_cell.read()] = ram_out_data_to_cell.read();


   if(ram_counter == TIMEOUT_RAM){
      ram_counter = 0;
      ram_run = false;
      buffer_ram_read = false;
      buffer_ram_write = false;
      ram_in_ready = true;
   }

   if(hm_run){
      hm_counter++;
   }

   if(hm_out_read || hm_out_write){
      hm_run = true;
      hm_in_ready = false;
      buffer_hm_read = hm_out_read.read();
      buffer_hm_write = hm_out_write.read();
   }

   if(hm_counter == TIMEOUT_HARD_MEMORY && buffer_hm_read){
      hm_in_data_from_cell = hardware_memory[hm_out_addr_cell.read()];
   }



   if(hm_counter == TIMEOUT_HARD_MEMORY && buffer_hm_write)
      hardware_memory[hm_out_addr_cell.read()] = hm_out_data_to_cell.read();

   if(hm_counter == TIMEOUT_HARD_MEMORY){
        hm_counter = 0;
        hm_run = false;
        buffer_hm_read = false;
        buffer_hm_write = false;
        hm_in_ready = true;
    }

}

void printRAM(){
    cout
<<"\n\n\n\n******************************printRAM*************************"
<<endl;
    int i,j;
    for(j = 0; j < 32; j++){
        for(i = 0; i < 32; i++){
            cout<< ram_memory[j*32 + i] <<" ";
        }
        if((j*32 + i) % 256 == 0)
            cout<<"_"<< endl;
        else
            cout<<endl;
    }
}

void printHARDWARE(){
    cout
<<"\n\n\n\n******************************printHARDWARE********************"
<<endl;
    int i,j;
    for(j = 0; j < 128; j++){
        for(i = 0; i < 32; i++){
            cout<< hardware_memory[j*32 + i] <<" ";
        }
        if((j*32 + i) % 256 == 0)
            cout<<"_"<< endl;
        else
            cout<<endl;
    }
}

int sc_main(int argc, char* argv[]) {

    top.clk(clk);
    top.sreset(sreset);
    // to ram memory
    top.ram_out_read(ram_out_read);
    top.ram_out_write(ram_out_write);
    top.ram_out_addr_cell(ram_out_addr_cell);
    top.ram_out_data_to_cell(ram_out_data_to_cell);
    top.ram_in_data_from_cell(ram_in_data_from_cell);
    top.ram_in_ready(ram_in_ready);
    // to hard memory
    top.hm_out_read(hm_out_read);
    top.hm_out_write(hm_out_write);
    top.hm_out_addr_cell(hm_out_addr_cell);
    top.hm_out_data_to_cell(hm_out_data_to_cell);
    top.hm_in_data_from_cell(hm_in_data_from_cell);
    top.hm_in_ready(hm_in_ready);
    // io_module
    top.user_read(user_read);
    top.user_write(user_write);
    top.user_data_to_reg(user_data_to_reg);
    top.user_data_from_reg(user_data_from_reg);

    sc_trace_file *wf = sc_create_vcd_trace_file("MPC_waveform");
    sc_trace(wf, clk, "0_clock");
    sc_trace(wf, sreset, "0_sreset");
    // virtual memory <=> real memory
    // ram
    sc_trace(wf, ram_out_read, "1_ram_out_read");
    sc_trace(wf, ram_out_write, "2_ram_out_write");
    sc_trace(wf, ram_out_addr_cell, "3_ram_out_addr_cell");
    sc_trace(wf, ram_out_data_to_cell, "4_ram_out_data_to_cell");
    sc_trace(wf, ram_in_data_from_cell, "5_ram_in_data_from_cell");
    sc_trace(wf, ram_in_ready, "6_ram_in_ready");
    // to hard disk
    sc_trace(wf, hm_out_read, "7_hard_out_read");
    sc_trace(wf, hm_out_write, "8_hard_out_write");
    sc_trace(wf, hm_out_addr_cell, "9_hard_out_addr_cell");
    sc_trace(wf, hm_out_data_to_cell, "10_hard_out_data_to_cell");
    sc_trace(wf, hm_in_data_from_cell, "11_hard_in_data_from_cell");
    sc_trace(wf, hm_in_ready, "12_hard_in_ready");
    // virtual memory <-> cache
    sc_trace(wf, top.m_read, "13_memory_read");
    sc_trace(wf, top.m_write, "14_memory_write");
    sc_trace(wf, top.m_addr_cell, "15_memory_addr_cell");
    sc_trace(wf, top.m_data_to_cell, "16_memory_data_to_cell");
    sc_trace(wf, top.m_data_from_cell, "17_memory_data_from_cell");
    sc_trace(wf, top.m_ready, "18_memory_ready");
    sc_trace(wf, top.m_save_data, "19_memory_save_data");
    // cache <-> memory controller
    sc_trace(wf, top.mc_read, "20_cache_read");
    sc_trace(wf, top.mc_write, "21_cache_write");
    sc_trace(wf, top.mc_addr_cell, "22_cache_addr_cell");
    sc_trace(wf, top.mc_data_to_cell, "23_cache_data_to_cell");
    sc_trace(wf, top.mc_data_from_cell, "24_cache_data_from_cell");
    sc_trace(wf, top.mc_ready, "25_cache_ready");
    sc_trace(wf, top.mc_save_data, "26_cache_save_data");
    // command decoder <-> alu and memory controller
    sc_trace(wf, top.cmd, "27_cmd");
    sc_trace(wf, top.operand1, "28_operand1");
    sc_trace(wf, top.operand2, "29_operand2");
    sc_trace(wf, top.data_cell, "30_data_cell");
    // alu <-> memory controller
    sc_trace(wf, top.reg_id, "31_alu_reg_id");
    sc_trace(wf, top.reg_read, "32_alu_reg_read");
    sc_trace(wf, top.reg_write, "33_alu_reg_write");
    sc_trace(wf, top.reg_data_to, "34_alu_reg_data_to");
    sc_trace(wf, top.reg_data_from, "35_alu_reg_data_from");
    sc_trace(wf, top.fbs, "36_alu_fbs");
    sc_trace(wf, top.fss, "37_alu_fss");
    sc_trace(wf, top.fes, "38_alu_fes");
    sc_trace(wf, top.fnes, "39_alu_fnes");
    sc_trace(wf, top.fbes, "40_alu_fbes");
    sc_trace(wf, top.fses, "41_alu_fses");
    // memory controller <-> io_module
    sc_trace(wf, top.io_read, "42_io_read");
    sc_trace(wf, top.io_write, "43_io_write");
    sc_trace(wf, top.io_data_to_reg, "44_io_data_to_reg");
    sc_trace(wf, top.io_data_from_reg, "45_io_data_from_reg");
    // User <-> io_module
    sc_trace(wf, user_read, "46_user_read");
    sc_trace(wf, user_write, "47_user_write");
    sc_trace(wf, user_data_to_reg, "48_user_data_to_reg");
    sc_trace(wf, user_data_from_reg, "49_user_data_from_reg");

    int n = 200;
    for(int i = 100; i< 100 + n + 1; i++)
      hardware_memory[i] = n + 101 - i;

    // programm
    hardware_memory[0] = 1392508928;    // mov reg0, #0;      # init
    hardware_memory[1] = 1392513024;    // mov reg1, #0;
    hardware_memory[2] = 1392517120;    // mov reg2, #0;
    hardware_memory[3] = 1392521416;    // mov reg3, #200;    # count of the elements
    hardware_memory[4] = 855691267;     // mov reg13, reg3;   # n - 2
    hardware_memory[5] = 1375784960;    // dec reg13;
    hardware_memory[6] = 285212672;   // dec reg13;
    hardware_memory[7] = 855654403;     // mov reg4, reg3;
    hardware_memory[8] = 1375748096;    // dec reg4;          # n-1
    hardware_memory[9] = 1392529508;    // mov reg5, #100;    # start address
    hardware_memory[10] = 1392570369;   // mov reg15, #1;
    hardware_memory[11] = 83976192;               // jmp X3(22);
    hardware_memory[12] = 1107296256;   // inc reg0;          # j++
                                !X1
    hardware_memory[13] = 134217741;    // fbs reg0, reg13;   # do while j <= n-2
```

```
   hardware_memory[14] = 352325669;       // jmp fbs, X6(37);
   hardware_memory[15] = 1392517120;    // mov reg2, #0;         # f = 0
   hardware_memory[16] = 1392513024;    // mov reg1, #0;         # i = 0
   hardware_memory[17] = 1375748096;    // dec reg4;             # n - 1 - j
   hardware_memory[18] = 83976192;                 // jmp X3(22);
   hardware_memory[19] = 1107300352;    // inc reg1              # i++
                                       !X2
   hardware_memory[20] = 134221828;       // fbs reg1, reg4        # do while i <= n - 1 - j
   hardware_memory[21] = 352325644;       // jmp fbs, X1(12)
   hardware_memory[22] = 855662593;       // mov reg6, reg1;       # i
                    A[i]                !X3
   hardware_memory[23] = 33579013;                 // add reg6, reg5;       # i + a0
   hardware_memory[24] = 117469190;       // movx reg7, reg6;    # memory[i + a0]
   hardware_memory[25] = 855670790;       // mov reg8, reg6;     # i + a0
         A[i + 1]
   hardware_memory[26] = 1107329024;    // inc reg8;             # i + a0 + 1
   hardware_memory[27] = 117477384;       // movx reg9, reg8;    # memory[i + a0 + 1]
   hardware_memory[28] = 134246409;       // fbs reg7, reg9;
   hardware_memory[29] = 352325663;       // jmp fbs, X4(31);
   hardware_memory[30] = 84025344;                 // jmp X5(34);
   hardware_memory[31] = 654348294;       // movm reg9, reg6;  # swap
!X4
   hardware_memory[32] = 654340104;       // movm reg7, reg8;
   hardware_memory[33] = 1392517121;    // mov reg2, #1;         # f = 1
   hardware_memory[34] = 1207967759;    // fes reg2, reg15;      #
                    !X5
   hardware_memory[35] = 352333843;       // jmp fes, X2(19);      # if f == 1 the cycle is
continues
   hardware_memory[36] = 83935232;                 // jmp X1;
   hardware_memory[37] = 1157627904;    // end                   #
                    !X6

   // it he beggining
   hm_in_ready = true;
   ram_in_ready = true;

   // read page from hardware memory
   for(int i = 0; i < 250000 * CYCLE_INDEX; i ++){
      setSignal();
      sc_start(4, SC_NS);
   }

   printHARDWARE();
   cout<<endl;
   cout<<endl;
   cout<<endl;
   printRAM();
   return 0;
}
```

```
#include "systemc.h"
#include "Top.h"

#define soft_assert(signal, expected) \
     if (signal.read() != expected) { \
         cerr << "@" << sc_time_stamp() << " Check failed. Expected: " << expected << ". Actual:
" << signal.read() << ".\n" << endl; \
     }
sc_clock clk("clock", 4, SC_NS);
   sc_signal<bool> sreset;                    // restart

// virtual memory <=> real memory
// to ram memory
sc_signal<bool> ram_out_read;                 // read from cell
sc_signal<bool> ram_out_write;                // write in cell
sc_signal<sc_uint<10> > ram_out_addr_cell;
sc_signal<sc_uint<32> > ram_out_data_to_cell;
sc_signal<sc_uint<32> > ram_in_data_from_cell;
sc_signal<bool> ram_in_ready;
// to hard memory
sc_signal<bool> hm_out_read;                  // read from cell
sc_signal<bool> hm_out_write;                 // write in cell
sc_signal<sc_uint<12> > hm_out_addr_cell;
sc_signal<sc_uint<32> > hm_out_data_to_cell;
sc_signal<sc_uint<32> > hm_in_data_from_cell;
sc_signal<bool> hm_in_ready;
// User <-> io_module
sc_signal<bool> user_read;
sc_signal<bool> user_write;
sc_signal<sc_uint<12> > user_data_to_reg;
sc_signal<sc_uint<12> > user_data_from_reg;

Top top("Top");

// memory
int ram_memory[1024];
int hardware_memory[4096];


int TIMEOUT_RAM = 4;
int TIMEOUT_HARD_MEMORY = 7;
int CYCLE_INDEX = TIMEOUT_HARD_MEMORY + 1;
int ram_counter = 0;
int hm_counter = 0;
bool ram_run = false, hm_run = false;
bool buffer_ram_read = false, buffer_ram_write = false, buffer_hm_read = false,
buffer_hm_write = false;
```

```
void setSignal(){

   if(ram_out_read || ram_out_write){
      ram_run = true;
      ram_in_ready = false;
      buffer_ram_read = ram_out_read.read();
      buffer_ram_write = ram_out_write.read();
   }

   if(ram_run){
      ram_counter++;
   }

   if(ram_counter == TIMEOUT_RAM && buffer_ram_read)
      ram_in_data_from_cell = ram_memory[ram_out_addr_cell.read()];


   if(ram_counter == TIMEOUT_RAM && buffer_ram_write)
      ram_memory[ram_out_addr_cell.read()] = ram_out_data_to_cell.read();


   if(ram_counter == TIMEOUT_RAM){
      ram_counter = 0;
      ram_run = false;
      buffer_ram_read = false;
      buffer_ram_write = false;
      ram_in_ready = true;
   }

   if(hm_run){
      hm_counter++;
   }

   if(hm_out_read || hm_out_write){
      hm_run = true;
      hm_in_ready = false;
      buffer_hm_read = hm_out_read.read();
      buffer_hm_write = hm_out_write.read();
   }

   if(hm_counter == TIMEOUT_HARD_MEMORY && buffer_hm_read){
      hm_in_data_from_cell = hardware_memory[hm_out_addr_cell.read()];
   }


   if(hm_counter == TIMEOUT_HARD_MEMORY && buffer_hm_write)
      hardware_memory[hm_out_addr_cell.read()] = hm_out_data_to_cell.read();

   if(hm_counter == TIMEOUT_HARD_MEMORY){
        hm_counter = 0;
        hm_run = false;
        buffer_hm_read = false;
        buffer_hm_write = false;
        hm_in_ready = true;
    }

}

void printRAM(){
    cout
<<"\n\n\n\n******************************printRAM************************"
<<endl;
    int i,j;
    for(j = 0; j < 32; j++){
        for(i = 0; i < 32; i++){
            cout<< ram_memory[j*32 + i] <<" ";
        }
        if((j*32 + i) % 256 == 0)
            cout<<"_"<< endl;
        else
            cout<<endl;
    }
}

void printHARDWARE(){
    cout
<<"\n\n\n\n******************************printHARDWARE*******************"
<<endl;
    int i,j;
    for(j = 0; j < 128; j++){
        for(i = 0; i < 32; i++){
            cout<< hardware_memory[j*32 + i] <<" ";
        }
        if((j*32 + i) % 256 == 0)
            cout<<"_"<< endl;
        else
            cout<<endl;
    }
}

int sc_main(int argc, char* argv[]) {

    top.clk(clk);
    top.sreset(sreset);
    // to ram memory
    top.ram_out_read(ram_out_read);
    top.ram_out_write(ram_out_write);
    top.ram_out_addr_cell(ram_out_addr_cell);
    top.ram_out_data_to_cell(ram_out_data_to_cell);
    top.ram_in_data_from_cell(ram_in_data_from_cell);
    top.ram_in_ready(ram_in_ready);
    // to hard memory
    top.hm_out_read(hm_out_read);
    top.hm_out_write(hm_out_write);
    top.hm_out_addr_cell(hm_out_addr_cell);
    top.hm_out_data_to_cell(hm_out_data_to_cell);
    top.hm_in_data_from_cell(hm_in_data_from_cell);
    top.hm_in_ready(hm_in_ready);
    // io_module
    top.user_read(user_read);
    top.user_write(user_write);
    top.user_data_to_reg(user_data_to_reg);
    top.user_data_from_reg(user_data_from_reg);

    sc_trace_file *wf = sc_create_vcd_trace_file("MPC_waveform");
    sc_trace(wf, clk, "0_clock");
    sc_trace(wf, sreset, "0_sreset");
    // virtual memory <=> real memory
    // ram
    sc_trace(wf, ram_out_read, "1_ram_out_read");
    sc_trace(wf, ram_out_write, "2_ram_out_write");
    sc_trace(wf, ram_out_addr_cell, "3_ram_out_addr_cell");
    sc_trace(wf, ram_out_data_to_cell, "4_ram_out_data_to_cell");
    sc_trace(wf, ram_in_data_from_cell, "5_ram_in_data_from_cell");
    sc_trace(wf, ram_in_ready, "6_ram_in_ready");
    // to hard disk
    sc_trace(wf, hm_out_read, "7_hard_out_read");
    sc_trace(wf, hm_out_write, "8_hard_out_write");
    sc_trace(wf, hm_out_addr_cell, "9_hard_out_addr_cell");
    sc_trace(wf, hm_out_data_to_cell, "10_hard_out_data_to_cell");
    sc_trace(wf, hm_in_data_from_cell, "11_hard_in_data_from_cell");
    sc_trace(wf, hm_in_ready, "12_hard_in_ready");
    // virtual memory <-> cache
    sc_trace(wf, top.m_read, "13_memory_read");
    sc_trace(wf, top.m_write, "14_memory_write");
    sc_trace(wf, top.m_addr_cell, "15_memory_addr_cell");
    sc_trace(wf, top.m_data_to_cell, "16_memory_data_to_cell");
    sc_trace(wf, top.m_data_from_cell, "17_memory_data_from_cell");
    sc_trace(wf, top.m_ready, "18_memory_ready");
    sc_trace(wf, top.m_save_data, "19_memory_save_data");
    // cache <-> memory controller
    sc_trace(wf, top.mc_read, "20_cache_read");
    sc_trace(wf, top.mc_write, "21_cache_write");
    sc_trace(wf, top.mc_addr_cell, "22_cache_addr_cell");
    sc_trace(wf, top.mc_data_to_cell, "23_cache_data_to_cell");
    sc_trace(wf, top.mc_data_from_cell, "24_cache_data_from_cell");
    sc_trace(wf, top.mc_ready, "25_cache_ready");
```

```
   sc_trace(wf, top.mc_save_data, "26_cache_save_data");
   // command decoder <-> alu and memory controller
   sc_trace(wf, top.cmd, "27_cmd");
   sc_trace(wf, top.operand1, "28_operand1");
   sc_trace(wf, top.operand2, "29_operand2");
   sc_trace(wf, top.data_cell, "30_data_cell");
   // alu <-> memory controller
   sc_trace(wf, top.reg_id, "31_alu_reg_id");
   sc_trace(wf, top.reg_read, "32_alu_reg_read");
   sc_trace(wf, top.reg_write, "33_alu_reg_write");
   sc_trace(wf, top.reg_data_to, "34_alu_reg_data_to");
   sc_trace(wf, top.reg_data_from, "35_alu_reg_data_from");
   sc_trace(wf, top.fbs, "36_alu_fbs");
   sc_trace(wf, top.fss, "37_alu_fss");
   sc_trace(wf, top.fes, "38_alu_fes");
   sc_trace(wf, top.fnes, "39_alu_fnes");
   sc_trace(wf, top.fbes, "40_alu_fbes");
   sc_trace(wf, top.fses, "41_alu_fses");
   // memory controller <-> io_module
   sc_trace(wf, top.io_read, "42_io_read");
   sc_trace(wf, top.io_write, "43_io_write");
   sc_trace(wf, top.io_data_to_reg, "44_io_data_to_reg");
   sc_trace(wf, top.io_data_from_reg, "45_io_data_from_reg");
   // User <-> io_module
   sc_trace(wf, user_read, "46_user_read");
   sc_trace(wf, user_write, "47_user_write");
   sc_trace(wf, user_data_to_reg, "48_user_data_to_reg");
   sc_trace(wf, user_data_from_reg, "49_user_data_from_reg");

   for(int i = 0; i< 4096; i++)
      hardware_memory[i] = i;

   // programm
hardware_memory[0] = 1392508928;        // mov reg0, i;
hardware_memory[1] = 1392513024;        // mov reg1, j;
hardware_memory[2] = 1392517120;        // mov reg2, k;
hardware_memory[3] = 1392521226;        // mov reg3, n; n=10
hardware_memory[4] = 855691267;         // mov reg13, reg3;
hardware_memory[5] = 1375784960;        // dec reg13;
hardware_memory[6] = 1392525322;        // mov reg4, m; m=10
hardware_memory[7] = 855695364;         // mov reg14, reg4;
hardware_memory[8] = 1375789056;        // dec reg14;
hardware_memory[9] = 1392529418;        // mov reg5, p; p=10
hardware_memory[10] = 855699461;        // mov reg15, reg5;
hardware_memory[11] = 1375793152;       // dec reg15;
hardware_memory[12] = 1392533604;       // mov reg6, #100;     # a0
hardware_memory[13] = 1392538700;       // mov reg7, #1100;    # b0
hardware_memory[14] = 1392543796;       // mov reg8, #2100;    # c0
hardware_memory[15] = 84013056;         // jump X3
```

```
hardware_memory[16] = 1392513024;       // mov reg1, 0;        # j = 0
                                   X1!
hardware_memory[17] = 1207959565;       // fes reg0, reg13;    # set flag-fes, if i == n
hardware_memory[18] = 352333871;        // cjmp fes, X4;       # if flag-fes is set, jump to X4
hardware_memory[19] = 1107296256;       // inc reg0;           # reg0++
hardware_memory[20] = 84013056;         // jump X3
hardware_memory[21] = 1392517120;       // mov reg2, 0;        # k = 0
                                   X2!
hardware_memory[22] = 855687168;        // mov reg12, reg0;    # i
hardware_memory[23] = 570474499;        // mult reg12, reg3;   # i * n
hardware_memory[24] = 33603585;         // add reg12, reg1;    # i * n + j
hardware_memory[25] = 33603592;         // add reg12, reg8;    # c0 + i * n + j
hardware_memory[26] = 654356492;        // movm reg11, reg12;  # memory[c0 + i * n + j] =
reg11
hardware_memory[27] = 1392553984;       // mov reg11, #0;      # reset register
hardware_memory[28] = 1207963663;       // fes reg1, reg15;    # set flag-fes, if j == p
hardware_memory[29] = 352333840;        // jump fes, X1;       # if flag-fes is set, jump to X1
hardware_memory[30] = 1107300352;       // inc reg1;           # reg1++
hardware_memory[31] = 855674880;        // mov reg9, reg0;     # i
                                   X3!
hardware_memory[32] = 570462211;        // mult reg9, reg3;    # i * n
hardware_memory[33] = 33591298;         // add reg9, reg2;     # i * n + k
hardware_memory[34] = 33591302;         // add reg9, reg6;     # a0 + i * n + k
hardware_memory[35] = 117477385;        // movx reg9, reg9;    # reg9 = memory[a0 + i * n
+ k]
hardware_memory[36] = 855678978;        // mov reg10, reg2;    # k
hardware_memory[37] = 570466308;        // mult reg10, reg4;   # k * m
hardware_memory[38] = 33595393;         // add reg10, reg1;    # k * m + j
hardware_memory[39] = 33595399;         // add reg10, reg7;    # b0 + k * m + j
hardware_memory[40] = 117481482;        // movx reg10, reg10;  # reg10 = memory[b0 + k
* m + j]
hardware_memory[41] = 570462218;        // mult reg9, reg10;   # a[i][k]*b[k][j]
hardware_memory[42] = 33599497;         // add reg11, reg9;    # q[i][j] + a[i][k]*b[k][j]
hardware_memory[43] = 1207967758;       // fes reg2, reg14;    # set flag-fes, if k == m
hardware_memory[44] = 352333845;        // jump fes, X2;       # if flag-fes is set, jump to X2
hardware_memory[45] = 1107304448;       // inc reg2;           # reg2++
hardware_memory[46] = 84013056;         // jump X3
hardware_memory[47] = 1157627904;       // end                 #
                                   X4!

int A[33][33];
int B[33][33];
int C[33][33];

  // matrix
  for(int i = 100; i < 200; i++)
    hardware_memory[i] =  1;
  for(int i = 1100; i < 1200; i++)
     hardware_memory[i] =  2;


  // it he beggining
  hm_in_ready = true;
  ram_in_ready = true;

  // read page from hardware memory
  for(int i = 0; i < 100000 * CYCLE_INDEX; i ++){
     setSignal();
     sc_start(4, SC_NS);
  }
  cout<<endl;
  cout<<endl;
  cout<<endl;
  printHARDWARE();
  for(int i = 0; i < 10; i++){
    for(int j = 0; j < 10; j++){
       A[i][j] = 1;
       B[i][j] = 2;
    }
  }

  cout<<endl;
  cout<<endl;
  cout<<endl;
  for(int i = 0; i < 10; i++)
    for(int j = 0; j < 10; j++)
       for(int k = 0; k < 10; k++)
          C[i][j] = C[i][j] + A[i][k]*B[k][j];

  for(int i = 0; i < 10; i++){
    for(int j = 0; j < 10; j++){
       cout<< C[i][j] << " ";
    }
    cout<<endl;
  }
  return 0;
}
```

```
#include "systemc.h"
#include "Top.h"

#define soft_assert(signal, expected) \
     if (signal.read() != expected) { \
         cerr << "@" << sc_time_stamp() << " Check failed. Expected: " << expected << ". Actual:
" << signal.read() << ".\n" << endl; \
     }
sc_clock clk("clock", 4, SC_NS);
   sc_signal<bool> sreset;                    // restart

// virtual memory <=> real memory
// to ram memory
sc_signal<bool> ram_out_read;                // read from cell
sc_signal<bool> ram_out_write;               // write in cell
sc_signal<sc_uint<10> > ram_out_addr_cell;
sc_signal<sc_uint<32> > ram_out_data_to_cell;
sc_signal<sc_uint<32> > ram_in_data_from_cell;
sc_signal<bool> ram_in_ready;
// to hard memory
sc_signal<bool> hm_out_read;                 // read from cell
sc_signal<bool> hm_out_write;                // write in cell
sc_signal<sc_uint<12> > hm_out_addr_cell;
sc_signal<sc_uint<32> > hm_out_data_to_cell;
sc_signal<sc_uint<32> > hm_in_data_from_cell;
sc_signal<bool> hm_in_ready;
// User <-> io_module
sc_signal<bool> user_read;
sc_signal<bool> user_write;
sc_signal<sc_uint<12> > user_data_to_reg;
sc_signal<sc_uint<12> > user_data_from_reg;

Top top("Top");

// memory
int ram_memory[1024];
int hardware_memory[4096];


int TIMEOUT_RAM = 4;
int TIMEOUT_HARD_MEMORY = 7;
int CYCLE_INDEX = TIMEOUT_HARD_MEMORY + 1;
int ram_counter = 0;
int hm_counter = 0;
bool ram_run = false, hm_run = false;
bool buffer_ram_read = false, buffer_ram_write = false, buffer_hm_read = false,
buffer_hm_write = false;
```

```
void setSignal(){

  if(ram_out_read || ram_out_write){
    ram_run = true;
    ram_in_ready = false;
    buffer_ram_read = ram_out_read.read();
    buffer_ram_write = ram_out_write.read();
  }

  if(ram_run){
    ram_counter++;
  }

  if(ram_counter == TIMEOUT_RAM && buffer_ram_read)
    ram_in_data_from_cell = ram_memory[ram_out_addr_cell.read()];


  if(ram_counter == TIMEOUT_RAM && buffer_ram_write)
    ram_memory[ram_out_addr_cell.read()] = ram_out_data_to_cell.read();


  if(ram_counter == TIMEOUT_RAM){
    ram_counter = 0;
    ram_run = false;
    buffer_ram_read = false;
    buffer_ram_write = false;
    ram_in_ready = true;
  }

  if(hm_run){
    hm_counter++;
  }

  if(hm_out_read || hm_out_write){
    hm_run = true;
    hm_in_ready = false;
    buffer_hm_read = hm_out_read.read();
    buffer_hm_write = hm_out_write.read();
  }

  if(hm_counter == TIMEOUT_HARD_MEMORY && buffer_hm_read){
    hm_in_data_from_cell = hardware_memory[hm_out_addr_cell.read()];
  }


  if(hm_counter == TIMEOUT_HARD_MEMORY && buffer_hm_write)
    hardware_memory[hm_out_addr_cell.read()] = hm_out_data_to_cell.read();

  if(hm_counter == TIMEOUT_HARD_MEMORY){
        hm_counter = 0;
        hm_run = false;
        buffer_hm_read = false;
        buffer_hm_write = false;
        hm_in_ready = true;
    }

}

void printRAM(){
    cout
<<"\n\n\n\n******************************printRAM*************************"
<<endl;
    int i,j;
    for(j = 0; j < 32; j++){
        for(i = 0; i < 32; i++){
            cout<< ram_memory[j*32 + i] <<" ";
        }
        if((j*32 + i) % 256 == 0)
            cout<<"_"<< endl;
        else
            cout<<endl;
    }
}

void printHARDWARE(){
    cout
<<"\n\n\n\n******************************printHARDWARE*******************"
<<endl;
    int i,j;
    for(j = 0; j < 128; j++){
        for(i = 0; i < 32; i++){
            cout<< hardware_memory[j*32 + i] <<" ";
        }
        if((j*32 + i) % 256 == 0)
            cout<<"_"<< endl;
        else
            cout<<endl;
    }
}

int sc_main(int argc, char* argv[]) {

    top.clk(clk);
    top.sreset(sreset);
    // to ram memory
    top.ram_out_read(ram_out_read);
    top.ram_out_write(ram_out_write);
    top.ram_out_addr_cell(ram_out_addr_cell);
    top.ram_out_data_to_cell(ram_out_data_to_cell);
    top.ram_in_data_from_cell(ram_in_data_from_cell);
    top.ram_in_ready(ram_in_ready);
    // to hard memory
    top.hm_out_read(hm_out_read);
    top.hm_out_write(hm_out_write);
    top.hm_out_addr_cell(hm_out_addr_cell);
    top.hm_out_data_to_cell(hm_out_data_to_cell);
    top.hm_in_data_from_cell(hm_in_data_from_cell);
    top.hm_in_ready(hm_in_ready);
    // io_module
    top.user_read(user_read);
    top.user_write(user_write);
    top.user_data_to_reg(user_data_to_reg);
    top.user_data_from_reg(user_data_from_reg);

    sc_trace_file *wf = sc_create_vcd_trace_file("MPC_waveform");
    sc_trace(wf, clk, "0_clock");
    sc_trace(wf, sreset, "0_sreset");
    // virtual memory <=> real memory
    // ram
    sc_trace(wf, ram_out_read, "1_ram_out_read");
    sc_trace(wf, ram_out_write, "2_ram_out_write");
    sc_trace(wf, ram_out_addr_cell, "3_ram_out_addr_cell");
    sc_trace(wf, ram_out_data_to_cell, "4_ram_out_data_to_cell");
    sc_trace(wf, ram_in_data_from_cell, "5_ram_in_data_from_cell");
    sc_trace(wf, ram_in_ready, "6_ram_in_ready");
    // to hard disk
    sc_trace(wf, hm_out_read, "7_hard_out_read");
    sc_trace(wf, hm_out_write, "8_hard_out_write");
    sc_trace(wf, hm_out_addr_cell, "9_hard_out_addr_cell");
    sc_trace(wf, hm_out_data_to_cell, "10_hard_out_data_to_cell");
    sc_trace(wf, hm_in_data_from_cell, "11_hard_in_data_from_cell");
    sc_trace(wf, hm_in_ready, "12_hard_in_ready");
    // virtual memory <-> cache
    sc_trace(wf, top.m_read, "13_memory_read");
    sc_trace(wf, top.m_write, "14_memory_write");
    sc_trace(wf, top.m_addr_cell, "15_memory_addr_cell");
    sc_trace(wf, top.m_data_to_cell, "16_memory_data_to_cell");
    sc_trace(wf, top.m_data_from_cell, "17_memory_data_from_cell");
    sc_trace(wf, top.m_ready, "18_memory_ready");
    sc_trace(wf, top.m_save_data, "19_memory_save_data");
    // cache <-> memory controller
    sc_trace(wf, top.mc_read, "20_cache_read");
    sc_trace(wf, top.mc_write, "21_cache_write");
    sc_trace(wf, top.mc_addr_cell, "22_cache_addr_cell");
    sc_trace(wf, top.mc_data_to_cell, "23_cache_data_to_cell");
    sc_trace(wf, top.mc_data_from_cell, "24_cache_data_from_cell");
    sc_trace(wf, top.mc_ready, "25_cache_ready");
```

```
   sc_trace(wf, top.mc_save_data, "26_cache_save_data");
   // command decoder <-> alu and memory controller
   sc_trace(wf, top.cmd, "27_cmd");
   sc_trace(wf, top.operand1, "28_operand1");
   sc_trace(wf, top.operand2, "29_operand2");
   sc_trace(wf, top.data_cell, "30_data_cell");
   // alu <-> memory controller
   sc_trace(wf, top.reg_id, "31_alu_reg_id");
   sc_trace(wf, top.reg_read, "32_alu_reg_read");
   sc_trace(wf, top.reg_write, "33_alu_reg_write");
   sc_trace(wf, top.reg_data_to, "34_alu_reg_data_to");
   sc_trace(wf, top.reg_data_from, "35_alu_reg_data_from");
   sc_trace(wf, top.fbs, "36_alu_fbs");
   sc_trace(wf, top.fss, "37_alu_fss");
   sc_trace(wf, top.fes, "38_alu_fes");
   sc_trace(wf, top.fnes, "39_alu_fnes");
   sc_trace(wf, top.fbes, "40_alu_fbes");
   sc_trace(wf, top.fses, "41_alu_fses");
   // memory controller <-> io_module
   sc_trace(wf, top.io_read, "42_io_read");
   sc_trace(wf, top.io_write, "43_io_write");
   sc_trace(wf, top.io_data_to_reg, "44_io_data_to_reg");
   sc_trace(wf, top.io_data_from_reg, "45_io_data_from_reg");
   // User <-> io_module
   sc_trace(wf, user_read, "46_user_read");
   sc_trace(wf, user_write, "47_user_write");
   sc_trace(wf, user_data_to_reg, "48_user_data_to_reg");
   sc_trace(wf, user_data_from_reg, "49_user_data_from_reg");

   for(int i = 0; i < 4095; i++)
     hardware_memory[i] = i;

   int n = 100;
   for(int i = 200; i< 200 + n + 1; i++)
     hardware_memory[i] = 1;
   for(int i = 200; i< 200 + n + 1; i = i + 1)
     hardware_memory[i] = 2;
   for(int i = 200; i< 200 + n + 1; i = i + 4)
     hardware_memory[i] = 3;
     for(int i = 200; i< 200 + n + 1; i = i + 10)
     hardware_memory[i] = 10;
   // programm
   hardware_memory[0] =  1392508938;     // mov reg0, m_width   # m_width
   hardware_memory[1] =  1392513034;     // mov reg1, m_height  # m_height
   hardware_memory[2] =  1392517123;     // mov reg2, w_width   # w_width
   hardware_memory[3] =  1392521219;     // mov reg3, w_height  # w_height
   hardware_memory[4] =  855695362;      // nope cmd (mov reg14, reg2)      FIX
   hardware_memory[5] =  1392566280;   //mov reg14, #8    # count of elements in window
FIX
```

```
   hardware_memory[6] =  1392570370;     // mov reg15, #2
   hardware_memory[7] =  855654402;       // mov reg4, reg2      # w_width
   hardware_memory[8] =  1392558208;     // # bias for window
   hardware_memory[9] =  1392562376;     // # bias for memory   01010011 000000001101
000011001000
   hardware_memory[10] =  838877199;     // div reg4, reg15     # edgex = w_width/2
   hardware_memory[11] =  855658499;     // mov reg5, reg3      # w_height
   hardware_memory[12] =  838881295;     // div reg5, reg15     # edgey = w_height/2
   hardware_memory[13] =  855662596;     // mov reg6, reg4      # x = edgex
  //!X1
   hardware_memory[14] =  855666693;     // mov reg7, reg5      # y = edgey
  //!X2
   hardware_memory[15] =  1392541696;   // mov reg8, #0       # fx = 0
   //!X3
   hardware_memory[16] =  1392545792;   // mov reg9, #0       # fy = 0
   //!X4
   // window[fx][fy] := matrix[x + fx - edgex][y + fy - edgey]
   hardware_memory[17] =  855678982;     // mov reg10, reg6     # x
   hardware_memory[18] =  33595400;       // add reg10, reg8     # x + fx
   hardware_memory[19] =  302030852;     // sub reg10, reg4     # x + fx - edgex
   hardware_memory[20] =  570466304;     // mult reg10,reg0     # (x + fx - edgex) * m_width
   hardware_memory[21] =  33595399;       // add reg10, reg7     # (x + fx - edgex) * m_width
+ y
   hardware_memory[22] =  33595401;       // add reg10, reg9     # (x + fx - edgex) * m_width
+ y + fy
   hardware_memory[23] =  302030853;     // sub reg10, reg5     # (x + fx - edgex) * m_width
+ y + fy - edgey
   hardware_memory[24] =  33595405;       // add reg10, #200     # 200 + (x + fx - edgex) *
m_width + y + fy - edgey = bias in memory to element
   hardware_memory[25] =  117481482;     // movx reg10, reg10   # reg10 = memory[reg10]
   hardware_memory[26] =  855683080;     // mov reg11, reg8     # fx
   hardware_memory[27] =  570470402;     // mult reg11, reg2    # fx * w_width
   hardware_memory[28] =  33599497;       // add reg11, reg9     # fx * w_width + fy
   hardware_memory[29] =  33599500;       // add reg11, #128     # 128 + fx * w_width + fy =
bias in the window memory
   hardware_memory[30] =  654352395;     // movm reg10, reg11   # memory[reg11] = reg10
   // цикл 4
   hardware_memory[31] =  1107333120;   // inc reg9
   hardware_memory[32] =  402690051;     // fss reg9, reg3      # if reg9 < w_height
   hardware_memory[33] =  352329745;     // jmp fss, X4(17)       //00010101 000000000010
000000010001
   // цикл 3
   hardware_memory[34] =  1107329024;   // inc reg8
   hardware_memory[35] =  402685954;     // fss reg8, reg2      # if reg8 < w_width
   hardware_memory[36] =  352329744;     // jmp fss, X3(16)
   hardware_memory[37] =  889458688;     // call save_reg(65) 00110101 000001000001
000000000000
   hardware_memory[38] =  855650318;     // mov reg3, reg14;    # count of the elements in
window
```

```
   hardware_memory[39] =  889556992;     // call sort (89) 00110101 000001011001
000000000000
   hardware_memory[40] =  889507840;     // call read_reg(77) 00110101 000001001101
000000000000
   // matrix[x][y] := window[w_width / 2][w_height / 2]
   hardware_memory[41] =  855678978;     // mov reg10, reg2     # w_width
   hardware_memory[42] =  838901775;     // div reg10, reg15    # w_width / 2
   hardware_memory[43] =  570466306;     // mult reg10, reg2    # w_width / 2 * w_width
   hardware_memory[44] =  1392553987;  // mov reg11, #3       # w_height    FIX
   hardware_memory[45] =  838905871;     // div reg11, reg15    # w_height / 2
   hardware_memory[46] =  33595403;       // add reg10, reg11    # w_width / 2 * w_width +
w_height / 2
   hardware_memory[47] =  33595404;       // add reg10, #128     # 128 + w_width / 2 *
w_width + w_height / 2
   hardware_memory[48] =  117481482;     // movx reg10, reg10   # reg10 = memory[reg10]
   hardware_memory[49] =  855683078;     // mov reg11, reg6     # x
   hardware_memory[50] =  570470400;     // mult reg11, reg0    # x * m_width
   hardware_memory[51] =  33599495;       // add reg11, reg7     # x * m_width + y
   hardware_memory[52] =  33599501;       // add reg11, #200
   hardware_memory[53] =  654352395;     // movm reg10, reg11   # memory[reg11] = reg10
   // цикл 2
   hardware_memory[54] =  1107324928;  // inc reg7
   hardware_memory[55] =  855699457;     // mov reg15, reg1     # m_height
   hardware_memory[56] =  302051333;     // sub reg15, reg5     # m_height - edgey
   hardware_memory[57] =  402681871;     // fss reg7, reg15     # if reg7 < reg15
   hardware_memory[58] =  352329743;     // jmp fss, X2(15)
   // цикл 1
   hardware_memory[59] =  1107320832;  // inc reg6
   hardware_memory[60] =  855699456;     // mov reg15, reg0     # m_width
   hardware_memory[61] =  302051332;     // sub reg15, reg4     # m_width - edgex
   hardware_memory[62] =  402677775;     // fss reg6, reg15     # if reg6 < reg15
   hardware_memory[63] =  352329742;     // jmp fss, X1(14)
   hardware_memory[64] =  1157627904;  // end
   //!save_reg
   hardware_memory[65] =  587816960;     // mov #150, reg0
   hardware_memory[66] =  587821057;     // mov #151, reg1
   hardware_memory[67] =  587825154;     // mov #152, reg2
   hardware_memory[68] =  587829252;     // mov #153, reg4
   hardware_memory[69] =  587833349;     // mov #154, reg5
   hardware_memory[70] =  587837446;     // mov #155, reg6
   hardware_memory[71] =  587841543;     // mov #156, reg7
   hardware_memory[72] =  587845640;     // mov #157, reg8
   hardware_memory[73] =  587849737;     // mov #158, reg9
   hardware_memory[74] =  587853837;     // mov #159, reg13
   hardware_memory[75] =  587857935;     // mov #160, reg15
   hardware_memory[76] =  620756992;     // ret
   //!read_reg
   hardware_memory[77] =  1124073622;  // mov reg0, #150
   hardware_memory[78] =  1124077719;  // mov reg1, #151
```

```
hardware_memory[79] =  1124081816;   // mov reg2, #152
hardware_memory[80] =  1124090009;   // mov reg4, #153
hardware_memory[81] =  1124094106;   // mov reg5, #154
hardware_memory[82] =  1124098203;   // mov reg6, #155
hardware_memory[83] =  1124102300;   // mov reg7, #156
hardware_memory[84] =  1124106397;   // mov reg8, #157
hardware_memory[85] =  1124110494;   // mov reg9, #158
hardware_memory[86] =  1124126879;   // mov reg13, #159
hardware_memory[87] =  1124135072;   // mov reg15, #160
hardware_memory[88] =  620756992;    // ret
// Sort
hardware_memory[89] = 1392508928;    // mov reg0, #0;      # init
hardware_memory[90] = 1392513024;    // mov reg1, #0;
hardware_memory[91] = 1392517120;    // mov reg2, #0;
hardware_memory[92] = 855691267;     // mov reg13, reg3;   # n - 2
hardware_memory[93] = 1375784960;    // dec reg13;
hardware_memory[94] = 855654403;     // mov reg4, reg3;
hardware_memory[95] = 1375748096;    // dec reg4;          # n-1
hardware_memory[96] = 1392529536;    // mov reg5, #128;    # start address
hardware_memory[97] = 1392570369;    // mov reg15, #1;
hardware_memory[98] = 84332544;                 // jmp X3(109);
hardware_memory[99] = 1107296256;    // inc reg0;          # j++
                                !X1
hardware_memory[100] = 134217741;    // fbs reg0, reg13;   # do while j <= n-2
hardware_memory[101] = 352325756;    // jmp fbs, X6(124);
hardware_memory[102] = 1392517120;   // mov reg2, #0;      # f = 0
hardware_memory[103] = 1392513024;   // mov reg1, #0;      # i = 0
hardware_memory[104] = 1375748096;   // dec reg4;          # n - 1 - j
hardware_memory[105] = 84332544;     // jmp X3(109);
hardware_memory[106] = 1107300352;   // inc reg1           # i++
                                !X2
hardware_memory[107] = 134221828;    // fbs reg1, reg4     # do while i <= n - 1 - j
hardware_memory[108] = 352325731;    // jmp fbs, X1(99)
hardware_memory[109] = 855662593;    // mov reg6, reg1;    # i
                A[i]            !X3
hardware_memory[110] = 33579013;     // add reg6, reg5;    # i + a0
hardware_memory[111] = 117469190;    // movx reg7, reg6;   # memory[i + a0]
hardware_memory[112] = 855670790;    // mov reg8, reg6;    # i + a0
       A[i + 1]
hardware_memory[113] = 1107329024;   // inc reg8;          # i + a0 + 1
hardware_memory[114] = 117477384;    // movx reg9, reg8;   # memory[i + a0 + 1]
hardware_memory[115] = 134246409;    // fbs reg7, reg9;
hardware_memory[116] = 352325750;    // jmp fbs, X4(118);
hardware_memory[117] = 84381696;     // jmp X5(121);
hardware_memory[118] = 654348294;    // movm reg9, reg6;   # swap                  !X4

hardware_memory[119] = 654340104;    // movm reg7, reg8;
hardware_memory[120] = 1392517121;   // mov reg2, #1;      # f = 1
hardware_memory[121] = 1207967759;   // fes reg2, reg15;   #                      !X5
```

```
   hardware_memory[122] = 352325738;    // jmp fes, X2(106);    # if f == 1 the cycle is
continues
   hardware_memory[123] = 84291584;      // jmp X1(99);
   hardware_memory[124] = 620756992;    // ret                          #
                  !X6

   // it he beggining
   hm_in_ready = true;
   ram_in_ready = true;

   // read page from hardware memory
   for(int i = 0; i < 50000 * CYCLE_INDEX; i ++){
      setSignal();
      sc_start(4, SC_NS);
   }

   printHARDWARE();
   cout<<endl;
   cout<<endl;
   cout<<endl;
   printRAM();
   return 0;
}
```

Virtual_memory.h

```
#include "systemc.h"

#ifndef Memory_H
#define Memory_H
   SC_MODULE(Virtual_memory){

         sc_in_clk clk;                    // main clock
         sc_in<bool> sreset;                  // restart

         // to cache
         sc_in<bool> ch_in_read;                    // read from cell
         sc_in<bool> ch_in_write;                   // write in cell
         sc_in<sc_uint<12> > ch_in_addr_cell;
         sc_in<sc_uint<32> > ch_in_data_to_cell;
         sc_out<sc_uint<32> > ch_out_data_from_cell;
         sc_out<bool> ch_out_ready;
         sc_in<bool> ch_in_save_all;

         // to ram memory
         sc_out<bool> ram_out_read;               // read from cell
         sc_out<bool> ram_out_write;              // write in cell
         sc_out<sc_uint<10> > ram_out_addr_cell;
         sc_out<sc_uint<32> > ram_out_data_to_cell;
         sc_in<sc_uint<32> > ram_in_data_from_cell;
         sc_in<bool> ram_in_ready;

         // to hard memory
         sc_out<bool> hm_out_read;               // read from cell
         sc_out<bool> hm_out_write;              // write in cell
         sc_out<sc_uint<12> > hm_out_addr_cell;
         sc_out<sc_uint<32> > hm_out_data_to_cell;
         sc_in<sc_uint<32> > hm_in_data_from_cell;
         sc_in<bool> hm_in_ready;

         unsigned int pages[4][2];            // page-table   (addr + flag_changed)



         int page, addr;                   // local variables
         bool first_step = true;              // first step in the executing command

         // buffer for input signals
         bool inner_read = false;
         bool inner_write = false;
         int inner_cell;

         // flags
         bool changing_page = false;
         bool page_hit = false;
         bool ram_is_empty = true;
```

```
        bool memory_is_lock = false;
        bool wait_data = false;
        bool stop_and_save = false;
        bool page_is_saving = false;

        // local registers
        int addr_page = 0;      // addr of the page
        int page_index = 0;     // index in array
        int page_size = 256;
        int bias = 0;
        int pointer = 0;
        int pointer_for_end = 0;

        void workWithMemory();

        SC_CTOR(Virtual_memory) :
            clk("clk"),
            sreset("sreset"),
            ch_in_read("ch_in_read"),
            ch_in_write("ch_in_write"),
            ch_in_addr_cell("ch_in_addr_cell"),
            ch_in_data_to_cell("ch_in_data_to_cell"),
            ch_out_data_from_cell("ch_out_data_from_cell"),
            ch_out_ready("ch_out_ready"),
            ch_in_save_all("ch_in_save_all"),
            ram_out_read("ram_out_read"),
            ram_out_write("ram_out_write"),
            ram_out_addr_cell("ram_out_addr_cell"),
            ram_out_data_to_cell("ram_out_data_to_cell"),
            ram_in_data_from_cell("ram_in_data_from_cell"),
            ram_in_ready("ram_in_ready"),
            hm_out_read("hm_out_read"),
            hm_out_write("hm_out_write"),
            hm_out_addr_cell("hm_out_addr_cell"),
            hm_out_data_to_cell("hm_out_data_to_cell"),
            hm_in_data_from_cell("hm_in_data_from_cell"),
            hm_in_ready("hm_in_ready")
        {
          SC_CTHREAD(workWithMemory, clk.pos());
          async_reset_signal_is(sreset,false);

          for(int i = 0; i < 4; i++){
            pages[i][0] = 16;
            pages[i][1] = 0;
          }
          cout<<"reset"<<endl;
        }
    };
#endif
```

Virtual_memory.cpp

```
#include "Virtual_memory.h"

void Virtual_memory::workWithMemory()
{
  while (true) {
    // reset read/write signals for memory
    ram_out_read = false;
    ram_out_write = false;
    hm_out_read = false;
    hm_out_write = false;

    // free memory flag
    memory_is_lock = false;

    // reset output signal
    ch_out_data_from_cell = 0;

    // wait input signal "save ram in harddisk"
    if(!stop_and_save)
      stop_and_save = ch_in_save_all.read();


    // if ram memory and hard disk is ready, virtual memory is ready too
    if(ram_in_ready && hm_in_ready)
      ch_out_ready = true;
    else
      ch_out_ready = false;

    // read input signal, if:
    // 1. page is not reloading now
    // 2. have got input read/write signal
    // 3. ram is not saving pages to hard disk now
    if(!changing_page && (ch_in_read.read() || ch_in_write.read()) && !stop_and_save){

      // save addres cell and address page to inner register
      addr = ch_in_addr_cell.read();
      addr_page = (addr >> 8) & 15;                    // 12-8 bits

      // save input signals to buffers
      inner_read = ch_in_read.read();
      inner_write = ch_in_write.read();
      inner_cell = ch_in_data_to_cell.read();

      // reset hit glag
      page_hit = false;

      // set hit flag true/false by situation
      if(ram_is_empty)
        page_hit = false;
      else{
        // set index of pages array in correct location
        page_index = pointer;
        // checking a page hit
        for(int i = 0; i < 4; i++){
            if(addr_page == pages[i][0]){
                page_index = i; // reload index if hit
                page_hit = true;
            }
        }
    }

    // set output signal, vm is not ready
    ch_out_ready = false;
}


// this part need for correct saving pages, when input signal "save all" is true
if(stop_and_save && !page_is_saving){

    // reset inner registers
    addr_page = 0;
    wait_data = false;

    // set index of pages array in correct location
    page_index = pointer_for_end;
    if(pointer_for_end == 4){
        pointer_for_end = 0;
        stop_and_save = false;
    }else{
        pointer_for_end++;
    }
}

// part for hit-situation
if(page_hit && ram_in_ready && !stop_and_save){

    //wait data from ram
    if(wait_data){
        // set output singals
        ch_out_ready = true;
        ch_out_data_from_cell = ram_in_data_from_cell;

        // reset an inner flag
        wait_data = false;
    }

    // writing
    if(inner_write){
        // set writing signals to ram memory
        ram_out_write = true;
        ram_out_data_to_cell = inner_cell;
        ram_out_addr_cell = (addr & 255) | (page_index << 8);         // 10-bit addr in ram

        // set signals to cache
        ch_out_data_from_cell = 0;
        ch_out_ready = false;                          // set to cache "vm is not ready"

        // set the inner registers
        pages[page_index][1] = 1;                         // page is changed
        inner_write = false;                          // reset the inner flag of the write

    }

    // reading
    if(inner_read){
        // signals to read data from ram
        ram_out_read = true;
        ram_out_addr_cell = (addr & 255) | (page_index << 8);         // 10-bit addr in ram

        // set signals to cache
        ch_out_data_from_cell = 0;
        ch_out_ready = false;                          // set to cache "vm is not ready"

        // set the inner registers
        inner_read = false;                          // reset the inner flag of the read
        wait_data = true;                              // flag to wait data from ram
    }

}

// this part2 need for correct saving pages, when input signal "save all" is true
if(stop_and_save && pages[page_index][1] == 1){
    page_is_saving = true;
}

// if page is missing, then reload page
if((!page_hit || stop_and_save) && ram_in_ready && hm_in_ready){

    //set the updating page flag
    changing_page = true;

    // save a old page to hard disk
    if(pages[page_index][1] == 1){

        if(wait_data){
            // write data from ram in hardware memory
            hm_out_write = true;
            hm_out_data_to_cell = ram_in_data_from_cell;
            hm_out_addr_cell = (pages[page_index][0] << 8) + bias;         // 12-bit addr in hm
            memory_is_lock = true;                         // hardware is locked
            bias++;
            wait_data = false;
        }

        if(bias < page_size){

            // signals to read data from ram
            ram_out_read = true;
            ram_out_addr_cell = (page_index << 8) + bias;                // 10-bit addr in ram
            wait_data = true;                                  // flag to wait data from ram

          }else {

            // if page is saved, reset flag "row is changed"
            pages[page_index][0] = addr_page;
            pages[page_index][1] = 0;
            bias = 0;

            page_is_saving = false;
          }
          ch_out_ready = false;
        }

        // read a new row
        if(pages[page_index][1] == 0 && !memory_is_lock && !stop_and_save){

          if(wait_data){
            // write data from ram in hardware memory
            ram_out_write = true;
            ram_out_data_to_cell = hm_in_data_from_cell;
            ram_out_addr_cell = (page_index << 8) + bias;                 // 10-bit addr in hm
            bias++;
            wait_data = false;
          }
          if(bias < page_size){
            // signals to read data from ram
            hm_out_read = true;
            hm_out_addr_cell = (addr & 3840) + bias;                  // 12-bit addr in ram
            wait_data = true;                                  // flag to wait data from ram
          }else{
            ram_is_empty = false;
            bias = 0;
            changing_page = false;                               // page is reload
            pages[page_index][1] = 0;
            pages[page_index][0] = addr_page;
            page_hit = true;

            pointer++;                                       // incriment page-pointer

            if(pointer == 4)
              pointer = 0;

          }
          ch_out_ready = false;
        }
      }
      wait();
    }
  }
```

Cache.h

```
#include "systemc.h"

#ifndef CACHE_H
#define CACHE_H

   SC_MODULE (Cache)
   {
      // [size of cache][ 16 data cell + 1 tag + 1 state]
      unsigned int cache_memory[32][18];         // cache

      sc_in_clk clk;                             // main clock
      sc_in<bool> sreset;                           // restart

      // to mp
      sc_in<bool> mc_read;                     // read from cache cell
      sc_in<bool> mc_write;                    // write to cache in cell
      sc_in<sc_uint<12> > mc_addr_cell;         // addr needing memory cell
      sc_in<sc_uint<32> > mc_data_to_cell;       // data wrote in memory cell
      sc_out<sc_uint<32> > mc_data_from_cell;    // data from memory cell
      sc_out<bool> mc_ready;
      sc_in<bool> mc_save_all;

      // to virtual memory
      sc_out<bool> m_read;                     // if data in cache not exit, get it from memory
      sc_out<bool> m_write;
      sc_out<sc_uint<12> > m_addr_cell;         // addr needing memory cell
      sc_out<sc_uint<32> > m_data_to_cell;       // data wrote in memory cell
      sc_in<sc_uint<32> > m_data_from_cell;      // data from memory cell
      sc_in<bool> m_ready;
      sc_out<bool> m_save_all;


      bool cache_hit = false;
      bool updating_cache = false;
      bool memory_lock = false;
      bool wait_data = false;       // wait data from memory
      bool isEmpty = true;          // cashe is empty

      //buffer
      bool inner_read, inner_write;
      int inner_cell;
      bool stop_and_save = false;
      bool row_is_saving = false;

      // the local variables
      int tag = 0, addr = 0, bias = 0, id_cache_row = 0;
      int id_cell = 0;
      int pointer_for_end = 0;
      int delay = 0;

      // trigger
      bool save_in_next_clk = false;
      void clear_cache_memory();
      void workWithCache();

      SC_CTOR(Cache) :
            clk("clk"),
            sreset("sreset"),

            mc_read("mc_read"),
            mc_write("mc_write"),
            mc_addr_cell("mc_addr_cell"),
            mc_data_to_cell("mc_data_to_cell"),
            mc_data_from_cell("mc_data_from_cell"),
            mc_ready("mc_ready"),
            mc_save_all("mc_save_all"),

            m_read("m_read"),
            m_write("m_write"),
            m_addr_cell("m_addr_cell"),
            m_data_to_cell("m_data_to_cell"),
            m_data_from_cell("m_data_from_cell"),
            m_ready("m_ready"),
            m_save_all("m_save_all")
      {
         SC_CTHREAD(workWithCache, clk.pos());
         async_reset_signal_is(sreset,false);
         clear_cache_memory();
      }
   };

#endif
```

Cache.cpp

```
#include "Cache.h"

void Cache::clear_cache_memory(){
   // reset cashe
   for(int i = 0; i < (sizeof(cache_memory)/sizeof(*cache_memory)); i++){
      cache_memory[i][16] = 0;
      cache_memory[i][17] = 0;
   }
}
```

```
void Cache::workWithCache()
{
   while (true) {
      mc_data_from_cell = 0;
      if(delay == 1){
         mc_ready = true;
         memory_lock = false;
         m_save_all = false;


         // wait input signal "save ram in harddisk"
         if(!stop_and_save)
            stop_and_save = mc_save_all.read();


         if(!updating_cache && (mc_read.read() || mc_write.read()) && !stop_and_save){
            addr = mc_addr_cell.read();
            tag = (addr >> 4) & 255;       // 12-4 bits
            bias = addr & 15;              // 3-0 bits

            id_cache_row = tag & 31;       // index in table (8-4 bit)

            inner_read = mc_read.read();
            inner_write = mc_write.read();
            inner_cell = mc_data_to_cell.read();
         }

         // checking row in table
         if(cache_memory[id_cache_row][16] == tag && !isEmpty)
            cache_hit = true;
         else
            cache_hit = false;


         // this part need for correct saving pages, when input signal "save all" is true
         if(stop_and_save && !row_is_saving){

            // reset inner registers
            id_cache_row = pointer_for_end;
            wait_data = false;

            // set index of pages array in correct location
            if(pointer_for_end == 32){
               pointer_for_end = 0;
               stop_and_save = false;
               save_in_next_clk = true;
            }else{
               pointer_for_end++;
            }
         /// cout<<"pointer_for_end " <<pointer_for_end<<endl;
        }

        if(save_in_next_clk && m_ready){
           m_save_all = true;
           save_in_next_clk = false;
        }


        if(cache_hit && !stop_and_save){
           //cout<<"=================== " << addr<<endl;
           // writing
           if(inner_write){
              cache_memory[id_cache_row][bias] = inner_cell;
              cache_memory[id_cache_row][17] = 1;                  // cell is updated
              inner_write = false;
             // cout<<"write bro " << inner_cell<<endl;
           }

           // reading
           if(inner_read){
              mc_data_from_cell = cache_memory[id_cache_row][bias];
              //cout<<"read bro " << cache_memory[id_cache_row][bias] <<endl;
              inner_read = false;
           }

        }

        // this part2 need for correct saving pages, when input signal "save all" is true
        if(stop_and_save && cache_memory[id_cache_row][17] == 1){
           row_is_saving = true;
        }

        // if cache is missing, we should updating cache block
        if(!cache_hit && (inner_read || inner_write || isEmpty) || stop_and_save){

           //set the updating flag
           updating_cache = true;
           mc_ready = false;
           memory_lock = false;
           m_write = false;
           m_read = false;

           // save a old row in the memory
           if(cache_memory[id_cache_row][17] == 1 && m_ready){

              // set signals to writing in memory
              m_write = true;
              m_addr_cell = (cache_memory[id_cache_row][16] << 4) | id_cell;   // set addr
```

```
                m_data_to_cell = cache_memory[id_cache_row][id_cell];          // set data

                id_cell++;                                        // incrementing counter id of the cells
                memory_lock = true;

                // if row is save, reset flag "row is changed"
                if(id_cell == 16){
                   cache_memory[id_cache_row][17] = 0;
                   id_cell = 0;
                   row_is_saving = false;
                }
             }
             // read a new row
             if(cache_memory[id_cache_row][17] == 0 && !memory_lock && m_ready &&
!stop_and_save){
                // wait data from real memory
                if(wait_data){
                   cache_memory[id_cache_row][id_cell] = m_data_from_cell.read();
                   id_cell++;
                }
                // if row is read, the change a tag
                if(id_cell == 16){
                   cache_memory[id_cache_row][16] = tag;
                   id_cell = 0;
                   wait_data = false;
                   isEmpty = false;
                   updating_cache = false;
                }else{

                   // set signals to reading from memory
                   m_read = true;
                   m_addr_cell = (tag << 4) | id_cell;                    // set addr

                   wait_data = true;
                }
             }
          }
          delay = 0;
       }else{
          delay++;

          if(true)
             mc_ready = false;
       }
       wait();
    }
}
```

Memory_controller.h

```
#include "systemc.h"


#ifndef MEMORY_CONTROLLER_H
#define MEMORY_CONTROLLER_H

  SC_MODULE (Memory_controller)
  {
    sc_in_clk clk;                    // main clock
    sc_in<bool> sreset;                  // restart

    // to cache
    sc_out<bool> ch_read;                 // read from cache cell
    sc_out<bool> ch_write;                // write to cache in cell
    sc_out<sc_uint<12> > ch_addr_cell;        // addr needing memory cell
    sc_out<sc_uint<32> > ch_data_to_cell;      // data wrote in memory cell
    sc_in<sc_uint<32> > ch_data_from_cell;    // data from memory cell
    sc_in<bool> ch_ready;
    sc_out<bool> ch_save_data;

    // to command decoder
    sc_in<sc_uint<8>> cd_cmd;
    sc_in<sc_uint<12>> cd_operand1;
    sc_in<sc_uint<12>> cd_operand2;
    sc_out<sc_uint<32>> cd_data_cell;

    // alu
    //--registers
    sc_out<sc_uint<4>> alu_reg_id;
    sc_out<bool> alu_reg_read;                 // read from cache cell
    sc_out<bool> alu_reg_write;                // write to cache in cell
    sc_out<sc_uint<32>> alu_reg_data_to;
    sc_in<sc_uint<32>> alu_reg_data_from;
    //--flags
    sc_in<bool> alu_fbs;
    sc_in<bool> alu_fss;
    sc_in<bool> alu_fes;
    sc_in<bool> alu_fnes;
    sc_in<bool> alu_fbes;
    sc_in<bool> alu_fses;
    // to io
    sc_out<bool> io_read;
    sc_out<bool> io_write;
    sc_out<sc_uint<12> > io_data_to_reg;
    sc_in<sc_uint<12> > io_data_from_reg;
```

```
	int cmd_counter = 0;
	bool memory_lock;		// one of the memory controller block blocked access to
memory
	bool first_step = true;	// cmd in first step of the execute
	bool second_step = true;
	bool wait_cmd;			// memory controller send signal "getNewCmd", and is awaiting
result
	bool end = false;
	// the inner registers
	int comand, type_slave, cmd_group, id_cmd, inner_reg;
	int op1, op2;
	int cd_delay = 0;
	int mc_delay = 0;
	int alu_delay = 0;
	// sub-programs
	int level1, level2, level3, level4;

	void controlling();

	SC_CTOR(Memory_controller) :
		clk("clk"),
		sreset("sreset"),
		ch_read("ch_read"),
		ch_write("ch_write"),
		ch_addr_cell("ch_addr_cell"),
		ch_data_to_cell("ch_data_to_cell"),
		ch_data_from_cell("ch_data_from_cell"),
		ch_save_data("ch_save_data"),
		cd_cmd("cd_cmdcmd"),
		cd_operand1("cd_operand1"),
		cd_operand2("cd_operand2"),
		cd_data_cell("cd_data_cell"),
		alu_reg_id("alu_reg_id"),
		alu_reg_read("alu_reg_read"),
		alu_reg_write("alu_reg_write"),
		alu_reg_data_to("alu_reg_data_to"),
		alu_reg_data_from("alu_reg_data_from"),
		alu_fbs("alu_fbs"),
		alu_fss("alu_fss"),
		alu_fes("alu_fes"),
		alu_fnes("alu_fnes"),
		alu_fbes("alu_fbes"),
		alu_fses("alu_fses")
	{
	   SC_CTHREAD(controlling, clk.pos());
	   async_reset_signal_is(sreset,false);
	}
   };
#endif
```

Memory_controller.cpp

```
#include "Memory_controller.h"

void Memory_controller::controlling()
{
   alu_reg_read = false;                    // don`t read from alu
   alu_reg_write = false;                   // don`t write in alu reg
   ch_write = false;                    // don`t write in memory/cache
   ch_read = false;                     // only read new cmd from memory
   io_read = false;
   io_write = false;
   cd_data_cell = 0;                        // send nope comand to comd decoder
   ch_save_data = false;

   //if(ch_ready)
      mc_delay++;

   // memory controller is work only if cache ready
   if(ch_ready && !end && mc_delay >1){

      // don`t process if cmd nope or controller is busy
      if(cd_cmd.read() != 0 && first_step){
         // read operands and comand in buffer
         op1 = cd_operand1.read();
         op2 = cd_operand2.read();
         comand = cd_cmd.read();

         type_slave = comand & 1;
         cmd_group = (comand >> 1) & 7;
         id_cmd = (comand >> 4) & 15;

         wait_cmd = false;

         cmd_counter++;
      }

      // if cmd for memory controoler, then work
      if(type_slave == 1)
      {
         if(first_step && second_step){

            // first step
            switch(cmd_group){
               // moving cmd
               case(1):

                  // mov addr, #5 (1 time)
                  if(id_cmd == 0) {
```

```
        // set signals for write
        ch_write = true;
        ch_addr_cell = op1;     // addr of cell in memory
        ch_data_to_cell = op2;  // data, which will be written

        // set flags for this cmd
        memory_lock = true;
        first_step = true;
        mc_delay=0;

        //cout<<"mov addr, #5"<<endl;
        break;
    }

    // mov X, addr (2 time) -- read data-cell from addr
    if(id_cmd == 1 || id_cmd == 4) {

        // set signals for read
        ch_read = true;
        ch_addr_cell = op2;

        // set flags for this cmd
        memory_lock = true;
        first_step = false;

        mc_delay=0;
        //cout<< " mov addr, addr    or    mov reg, addr"<<endl;
        break;
    }

    // mov X, reg (2 time)  -- read data from register
    if(id_cmd == 2 || id_cmd == 3) {

        // set signals for read
        alu_reg_read = true;
        alu_reg_id = op2;

        // set flags for this cmd
        memory_lock = true;
        first_step = false;
        //cout<<" mov addr, reg   or    mov reg, reg"<<endl;
        break;
    }

    // mov reg, #5 (1 time)
    if(id_cmd == 5) {
        // set signals for write
        alu_reg_write = true;
        alu_reg_id = op1;       // id of reg
```

```
                alu_reg_data_to = op2;   // data, which will be written

                // set flags for this cmd
                memory_lock = true;
                first_step = true;
                //cout<<"mov reg, #5"<<endl;
                break;
            }

            // mov reg, cmd_counter (1 time)
            if(id_cmd == 6) {
                // set signals for write
                alu_reg_write = true;
                alu_reg_id = op1;               // id of reg
                alu_reg_data_to = cmd_counter;   // data, which will be written

                // set flags for this cmd
                memory_lock = true;
                first_step = true;
                //cout<<"mov reg, cmd_counter"<<endl;
                break;
            }

            // mov cmd_counter, reg1 (2 time)
            if(id_cmd == 7) {
                // set signals for write
                alu_reg_read = true;
                alu_reg_id = op2;               // id of reg
                alu_reg_data_to = cmd_counter;   // data, which will be written

                // set flags for this cmd
                memory_lock = true;
                first_step = false;
                //cout<<"mov cmd_counter, reg1"<<endl;
                break;
            }

            break;

        // jumping
        case(2):

            // jmp #5 (jump to addr)
            if(id_cmd == 0) {
                cmd_counter = op1;
                first_step = true;
                //cout<<"jmp #5 "<<endl;
                break;
            }
```

```
// jump if condition is true
if(id_cmd == 1) {
   //cout<<"jump if condition"<<endl;
   // fbs-1, fss-2, fes-3, fnes-4, fbes-5, fses-6
   if(op1 == 1 && alu_fbs){ cmd_counter = op2; /*cout<<"alu_fbs"<<endl;*/}
   if(op1 == 2 && alu_fss){ cmd_counter = op2; /*cout<<"alu_fss"<<endl;*/}
   if(op1 == 3 && alu_fes){ cmd_counter = op2;/* cout<<"alu_fes"<<endl;*/}
   if(op1 == 4 && alu_fnes){ cmd_counter = op2; /*cout<<"alu_fnes"<<endl;*/}
   if(op1 == 5 && alu_fbes){ cmd_counter = op2; /*cout<<"alu_fbes"<<endl;*/}
   if(op1 == 6 && alu_fses){ cmd_counter = op2; /*cout<<"alu_fses"<<endl;*/}

   first_step = true;

   break;
}

// ret
if(id_cmd == 2) {
   //cout<<"ret"<<endl;
   if(level4 != 0){
      cmd_counter = level4;
      level4 = 0;

   }else if(level3 != 0){
      cmd_counter = level3;
      level3 = 0;

   }else if(level2 != 0){
      cmd_counter = level2;
      level2 = 0;

   }else if(level1 != 0){
      cmd_counter = level1;
      level1 = 0;
   }

   first_step = true;

   break;
}

// call adr
if(id_cmd == 3) {
   //cout<<"call adr"<<endl;
   if(level1 == 0){
      level1 = cmd_counter;
      cmd_counter = op1;
```

```
            }else if(level2 == 0){
               level2 = cmd_counter;
               cmd_counter = op1;

            }else if(level3 == 0){
               level3 = cmd_counter;
               cmd_counter = op1;

            }else if(level4 == 0){
               level4 = cmd_counter;
               cmd_counter = op1;
            }

            first_step = true;
            break;
         }

         if(id_cmd == 4) {
            //cout<<"end"<<endl;
            end = true;
            memory_lock = true;
            first_step = true;
            ch_save_data = true;
            break;
         }
         break;

      // movxing cmd
      case(3):

         // movx reg1, reg2 (reg1 = memory[*reg2]) STEP1 - read addr from reg2
         if(id_cmd == 0) {
            //cout<<"movx reg1, reg2"<<endl;
            // set signals for read
            alu_reg_read = true;
            alu_reg_id = op2;

            // set flags for this cmd
            memory_lock = true;
            first_step = false;

            break;
         }

         // movx reg1, addr (reg1 = memory[*addr]) STEP1 - read addr from addr
         if(id_cmd == 1) {
            // cout<<"movx reg1, addr"<<endl;
            // set signals for read
            ch_read = true;
```

```
                ch_addr_cell = op2;

                // set flags for this cmd
                memory_lock = true;
                first_step = false;

                mc_delay=0;
                break;
            }

            // movm reg1, reg2 (memory[*reg2] = reg1) STEP1 - read addr from reg2
            if(id_cmd == 2) {
                //cout<<"movm reg1, reg2"<<endl;
                // set signals for read
                alu_reg_read = true;
                alu_reg_id = op2;

                // set flags for this cmd
                memory_lock = true;
                first_step = false;

                break;
            }

            // movm addr, reg2 (memory[*reg2] = memory[addr]) STEP1 - read addr from
reg2
            if(id_cmd == 3) {
                //cout<<"movm addr, reg2"<<endl;
                // set signals for read
                alu_reg_read = true;
                alu_reg_id = op2;

                // set flags for this cmd
                memory_lock = true;
                first_step = false;

                break;
            }

            break;

        // reading
        case(4):

            //mov reg, -io_reg-
            if(id_cmd == 0){
                //cout<<"mov reg, -io_reg-"<<endl;
                // set signals for read
                io_read = true;
                // set flags for this cmd
                memory_lock = true;
                first_step = false;

                break;
            }

            // mov -io_reg-, reg
            if(id_cmd == 1){
                //cout<<"mov -io_reg-, reg"<<endl;
                // set signals for read
                alu_reg_read = true;
                alu_reg_id = op1;

                // set flags for this cmd
                memory_lock = true;
                first_step = false;
                break;
            }
            break;
    }
    if(first_step){
        op1 = 0;
        op2 = 0;
        comand = 0;
        type_slave = 0;
        cmd_group = 0;
        id_cmd = 0;
        first_step = true;
    }
}
else if(second_step)
{
    switch(cmd_group){
        // moving cmd
        case(1):
            // mov addr, addr (2 time)
            if(id_cmd == 1) {

                // set signals for write
                ch_write = true;
                ch_addr_cell = op1;
                ch_data_to_cell = ch_data_from_cell.read();

                // set flags for this cmd
                memory_lock = true;
                second_step = true;
```

```
                    mc_delay=0;
                    break;
                }

                // mov addr, reg (2 time)
                if(id_cmd == 2) {

                    // set signals for write
                    ch_write = true;
                    ch_addr_cell = op1;
                    ch_data_to_cell = alu_reg_data_from.read();

                    // set flags for this cmd
                    memory_lock = true;
                    second_step = true;

                    mc_delay=0;
                    break;
                }

                // mov reg, reg (2 time)
                if(id_cmd == 3) {

                    // set signals for write
                    alu_reg_write = true;
                    alu_reg_id = op1;
                    alu_reg_data_to = alu_reg_data_from.read();

                    // set flags for this cmd
                    memory_lock = true;
                    second_step = true;
                    break;
                }

                // mov reg, addr (2 time)
                if(id_cmd == 4) {

                    // set signals for write
                    alu_reg_write = true;
                    alu_reg_id = op1;
                    alu_reg_data_to = ch_data_from_cell.read();

                    // set flags for this cmd
                    memory_lock = true;
                    second_step = true;
                    break;
                }

                // mov cmd_counter, reg (2 time)
```

```
        if(id_cmd == 7) {

            // set signals for write
            cmd_counter = alu_reg_data_from.read();

            // set flags for this cmd
            memory_lock = false;
            second_step = true;
            break;
        }
        break;

    // Indirect addressing
    case(3):
        // movx reg1, reg2 (reg1 = memory[*reg2]) STEP2 - read data from memory
        if(id_cmd == 0){
            // set signals for read
            ch_read = true;
            ch_addr_cell = alu_reg_data_from.read() & 4095;

            // set flags for this cmd
            memory_lock = true;
            second_step = false;

            mc_delay=0;
            break;
        }

        // movx reg1, addr (reg1 = memory[*addr]) STEP2 - read data from addr
        if(id_cmd == 1) {
            // set signals for read
            ch_read = true;
            ch_addr_cell = ch_data_from_cell.read() & 4095;

            // set flags for this cmd
            memory_lock = true;
            second_step = false;

            mc_delay=0;
            break;
        }

        // movm reg1, reg2 (memory[*reg2] = reg1) STEP2 - read data from reg1
        if(id_cmd == 2) {
            // save addr in inner register
            inner_reg = alu_reg_data_from.read();

            // set signals for read
            alu_reg_read = true;
```

```
                    alu_reg_id = op1;

                    // set flags for this cmd
                    memory_lock = true;
                    second_step = false;

                    break;
                }

                // movm addr, reg2 (memory[*reg2] = memory[addr]) STEP2 - read data from
memory
                if(id_cmd == 3) {

                    // save addr in inner register
                    inner_reg = alu_reg_data_from.read();

                    // set signals for read
                    ch_read = true;
                    ch_addr_cell = op1;

                    // set flags for this cmd
                    memory_lock = true;
                    second_step = false;

                    mc_delay=0;
                    break;
                }
              case(4):
                //mov reg, io_reg
                if(id_cmd == 0){

                    // set signals for read
                    alu_reg_write = true;
                    alu_reg_id = op1;
                    alu_reg_data_to = io_data_from_reg.read();

                    // set flags for this cmd
                    memory_lock = true;
                    second_step = true;

                    break;
                }
                // mov io_reg, reg
                if(id_cmd == 1){

                    // set signals for read
                    io_write = true;
                    io_data_to_reg = alu_reg_data_from.read() & 4095;
                    // set flags for this cmd
                    memory_lock = true;
                    second_step = true;
                    break;
                }
            }

            if(second_step){
                op1 = 0;
                op2 = 0;
                comand = 0;
                type_slave = 0;
                cmd_group = 0;
                id_cmd = 0;
                first_step = true;
            }
        }else{

            switch(cmd_group){
                case(3):

                // movx reg1, reg2 (reg1 = memory[*reg2]) STEP3 - write data from memory in
register
                if(id_cmd == 0 || id_cmd == 1){

                    // set signals for read
                    alu_reg_write = true;
                    alu_reg_id = op1;
                    alu_reg_data_to = ch_data_from_cell;

                    // set flags for this cmd
                    memory_lock = true;
                    break;
                }
                                // movm reg1, reg2 (memory[*reg2] = reg1) STEP2 - read data
from reg1
                if(id_cmd == 2) {
                    // set signals for write
                    ch_write = true;
                    ch_addr_cell = inner_reg;
                    ch_data_to_cell = alu_reg_data_from.read();

                    // set flags for this cmd
                    memory_lock = true;

                    mc_delay=0;
                    break;
                }
```

```
                                    // movm addr, reg2 (memory[*reg2] = memory[addr]) STEP2 -
read data from memory
                  if(id_cmd == 3) {

                     // set signals for write
                     ch_write = true;
                     ch_addr_cell = inner_reg;
                     ch_data_to_cell = ch_data_from_cell.read();

                     // set flags for this cmd
                     memory_lock = true;

                     mc_delay=0;
                     break;
                  }
                  break;
            }

            op1 = 0;
            op2 = 0;
            comand = 0;
            type_slave = 0;
            cmd_group = 0;
            id_cmd = 0;
            first_step = true;
            second_step = true;

         }
      }

      // set signal for read cmd
      if(wait_cmd){
         cd_data_cell = ch_data_from_cell;

         if(cd_delay >= 2){
            wait_cmd = false;
            cd_delay = 0;
         }else{
            cd_delay++;
         }
         memory_lock = true;

      }

      if(!memory_lock){
         // set signal for read cmd
         ch_read = true;              // read new cmd from memory
         ch_addr_cell = cmd_counter;    // set addr, witch will be read from memory
         wait_cmd = true;
```

```
            // reset inner register
            type_slave = 0;
            cmd_group = 0;
            id_cmd = 0;
            mc_delay=0;
        }

        // free lock
        memory_lock = false;

    }else{
        ch_save_data = false;
    }

    wait();
}
```

Command_decoder.h

```
#include "systemc.h"

#ifndef Command_decoder_h
#define         Command_decoder_h

  SC_MODULE(Command_decoder){
    sc_in_clk clk;                      // main clock
    sc_in<bool> sreset;                   // restart

    // to memory controller and alu
    sc_out<sc_uint<8>> cmd;
    sc_out<sc_uint<12>> operand1;
    sc_out<sc_uint<12>> operand2;
    sc_in<sc_uint<32>> data_cell;

    void decoding_command();

    SC_CTOR(Command_decoder) :
        cmd("cmd"),
        operand1("operand1"),
        operand2("operand2"),
        data_cell("data_cell")
    {
      cout << "new Command_decoder" << endl;
      SC_CTHREAD(decoding_command, clk.pos());
      async_reset_signal_is(sreset,false);
    }
  };
#endif
```

Command_decoder.cpp

```
#include "Command_decoder.h"

void Command_decoder::decoding_command()
{
  int input_cell = data_cell.read();    // read from memory
  while(true){
    cmd =(input_cell >> 24) &  255;       // 8 2
    operand1 = (input_cell >> 12) & 4095;  // 12
    operand2 = input_cell & 4095;         // 12
    wait();
  }
}
```

ALU.h

```
#include "systemc.h"

#ifndef ALU_H
#define ALU_H

SC_MODULE(Alu){
		sc_in_clk clk;
		sc_in<bool> sreset;

	// to command decoder
	sc_in<sc_uint<8>> cmd;
	sc_in<sc_uint<12>> operand1;
	sc_in<sc_uint<12>> operand2;

	// memory controller
	sc_in<sc_uint<4>> alu_reg_id;
	sc_in<bool> alu_reg_read;			// read from cache cell
	sc_in<bool> alu_reg_write;			// write to cache in cell
	sc_in<sc_uint<32>> alu_reg_data_to;
	sc_out<sc_uint<32>> alu_reg_data_from;
	sc_out<bool> alu_fbs;
	sc_out<bool> alu_fss;
	sc_out<bool> alu_fes;
	sc_out<bool> alu_fnes;
	sc_out<bool> alu_fbes;
	sc_out<bool> alu_fses;

	//registers
	unsigned int registers[16];

	// fbs-1, fss-2, fes-3, fnes-4, fbes-5, fses-6
	bool fbs, fss, fes, fnes, fbes, fses;

	// Mathematical actions (result write in a)
	void _add(int a, int b);
	void _sub(int a, int b);
	void _mult(int a, int b);
	void _div(int a, int b);
	void _inc(int a);
	void _dec(int a);

	// Boolean actions
	void _xor(int a, int b);
	void _and(int a, int b);
	void _or(int a, int b);
	void _not(int a);

	// shifts
	void _rs(int a);
	void _ls(int a);
```

```
        void _rsn(int a, int n);
        void _lsn(int a, int n);

        // comparisons
        void _fbs(int first, int two);
        void _fss(int first, int two);
        void _fbes(int first, int two);
        void _fses(int first, int two);
        void _fes(int first, int two);
        void _fnes(int first, int two);

        void calculate();
        void clear_regs_and_flags();

        int comand, type_slave, cmd_group, id_cmd;
        int id_reg, op1, op2;

            SC_CTOR(Alu):
                        clk("clk"),
                        sreset("reset"),
              cmd("cmd"),
              operand1("operand1"),
              operand2("operand2"),
              alu_reg_id("alu_reg_id"),
              alu_reg_read("alu_reg_read"),
              alu_reg_write("alu_reg_write"),
              alu_reg_data_to("alu_reg_data_to"),
              alu_reg_data_from("alu_reg_data_from"),
              fbs("fbs"),
              fss("fss"),
              fes("fes"),
              fnes("fnes"),
              fbes("fbes"),
              fses("fses")
            {
           SC_CTHREAD(calculate, clk.pos());
           cout << "Executing new" << endl;
           async_reset_signal_is(sreset,false);
           clear_regs_and_flags();
            }

};
#endif
```

ALU.cpp

```
#include "ALU.h"
void Alu::clear_regs_and_flags(){
   // reset cashe
   for(int i = 0; i < (sizeof(registers)/sizeof(*registers)); i++)
      registers[i] = 0;

   // clear registers
```

```
  fbs = 0;
  fss = 0;
  fes = 0;
  fnes = 0;
  fbes = 0;
  fses = 0;
}

void Alu::calculate(){

  // Set uniq number of register
  id_reg = alu_reg_id.read();

  // write to registers
  if(alu_reg_write.read())
    registers[id_reg] = alu_reg_data_to.read();

  // read from registers
  if(alu_reg_read.read()){
    alu_reg_data_from = registers[id_reg];
  }

  // alu work
  op1 = operand1.read();
  op2 = operand2.read();
  comand = cmd.read();
  type_slave = comand & 1;

  if(type_slave == 0)// alu is ɸ slave with id equal zero
  {
    cmd_group = (comand >> 1) & 7;
    id_cmd = (comand >> 4) & 15;

    switch(cmd_group){
      case (0):
        // nope comand
        break;

      // Mathematic
      case(1):
        if(id_cmd == 0) { _add(op1, op2); break; }
        if(id_cmd == 1) { _sub(op1, op2); break; }
        if(id_cmd == 2) { _mult(op1, op2); break; }
        if(id_cmd == 3) { _div(op1, op2); break; }
        if(id_cmd == 4) { _inc(op1); break; }
        if(id_cmd == 5) { _dec(op1); break; }
        break;

      // Boolean
      case(2):
        if(id_cmd == 0) { _xor(op1, op2); break; }
        if(id_cmd == 1) { _and(op1, op2); break; }
        if(id_cmd == 2) { _or(op1, op2); break; }
```

```
            if(id_cmd == 3) { _not(op1); break; }
            break;

        // Shifts
        case(3):
            if(id_cmd == 0) { _rs(op1); break; }
            if(id_cmd == 1) { _ls(op1); break; }
            if(id_cmd == 2) { _rsn(op1, op2); break; }
            if(id_cmd == 3) { _lsn(op1, op2); break; }
            break;

        // Inequality
        case(4):
            if(id_cmd == 0) { _fbs(op1, op2); break; }
            if(id_cmd == 1) { _fss(op1, op2); break; }
            if(id_cmd == 2) { _fbes(op1, op2); break; }
            if(id_cmd == 3) { _fses(op1, op2); break; }
            if(id_cmd == 4) { _fes(op1, op2); break; }
            if(id_cmd == 5) { _fnes(op1, op2); break; }
            break;
      }
   }

   alu_fbs = fbs;
   alu_fss = fss;
   alu_fes = fes;
   alu_fnes = fnes;
   alu_fbes = fbes;
   alu_fses = fses;

   wait();
}

// Mathematical actions (result write in a)
void Alu::_add(int id_1, int id_2){ registers[id_1] += registers[id_2]; /*cout<<"add"<<endl;*/}
void Alu::_sub(int id_1, int id_2){ registers[id_1] -= registers[id_2]; /*cout<<"sub"<<endl;*/}
void Alu::_mult(int id_1, int id_2){registers[id_1] *= registers[id_2]; /*cout<<"mult"<<endl;*/}
void Alu::_div(int id_1, int id_2){ registers[id_1] /= registers[id_2];/* cout<<"div"<<endl;*/}
void Alu::_inc(int id_1){registers[id_1] += 1; /*cout<<""<<endl;*/}
void Alu::_dec(int id_1){registers[id_1] -= 1; /*cout<<""<<endl;*/}

// Boolean actions
void Alu::_xor(int id_1, int id_2){registers[id_1] = registers[id_1] xor registers[id_2];
/*cout<<"xor"<<endl;*/}
void Alu::_and(int id_1, int id_2){registers[id_1] &= registers[id_2];/* cout<<"and"<<endl;*/}
void Alu::_or(int id_1, int id_2){registers[id_1] |= registers[id_2]; /*cout<<"or"<<endl;*/}
void Alu::_not(int id_1){registers[id_1] = ~registers[id_1]; /*cout<<"not"<<endl;*/}

// shifts
void Alu::_rs(int id_1){ registers[id_1] = registers[id_1] >> 1; /*cout<<"rs"<<endl;*/}
void Alu::_ls(int id_1){ registers[id_1] = registers[id_1] << 1; /*cout<<"ls"<<endl;*/}
void Alu::_rsn(int id_1, int id_2){ registers[id_1] = registers[id_1] >> registers[id_2];
/*cout<<"rsn"<<endl;*/}
```

```
void Alu::_lsn(int id_1, int id_2){ registers[id_1] = registers[id_1] << registers[id_2];
/*cout<<"lsn"<<endl;*/}

// comparison
// fbs-1, fss-2, fes-3, fnes-4, fbes-5, fses-6
void Alu::_fbs(int id_1, int id_2){ fbs = registers[id_1] > registers[id_2]; /*cout<<"bfs"<<endl;*/}
void Alu::_fss(int id_1, int id_2){ fss = registers[id_1] < registers[id_2]; /*cout<<"fss"<<endl;*/}
void Alu::_fbes(int id_1, int id_2){ fbes = registers[id_1] >= registers[id_2]; /*cout<<"fbes"<<endl;*/}
void Alu::_fses(int id_1, int id_2){ fses = registers[id_1] <= registers[id_2]; /*cout<<"fses"<<endl;*/}
void Alu::_fes(int id_1, int id_2){ fes = registers[id_1] == registers[id_2]; /*cout<<"fes"<<endl;*/}
void Alu::_fnes(int id_1, int id_2){fnes = registers[id_1] != registers[id_2]; /*cout<<"fnes"<<endl;*/}
```

io_module.h

```
#include "systemc.h"

#ifndef IO_H
#define IO_H

   SC_MODULE (IO)
   {
      int io_register;

      sc_in_clk clk;                          // main clock
      sc_in<bool> sreset;                       // restart

      // to mc
      sc_in<bool> io_read;
      sc_in<bool> io_write;
      sc_in<sc_uint<12> > io_data_to_reg;
      sc_out<sc_uint<12> > io_data_from_reg;

      // from user
      sc_in<bool> user_read;
      sc_in<bool> user_write;
      sc_in<sc_uint<12> > user_data_to_reg;
      sc_out<sc_uint<12> > user_data_from_reg;

      void io_cycle();

      SC_CTOR(IO) :
              clk("clk"),
              sreset("sreset"),

              io_read("io_read"),
              io_write("io_write"),
              io_data_to_reg("io_data_to_reg"),
              io_data_from_reg("io_data_from_reg"),
              user_read("user_read"),
              user_write("user_write"),
              user_data_to_reg("user_data_to_reg"),
              user_data_from_reg("user_data_from_reg")

      {
         SC_CTHREAD(io_cycle, clk.pos());
         async_reset_signal_is(sreset,false);
      }
   };

#endif
```

io_module.cpp

```
#include "io_module.h"

void IO::io_cycle()
{
   while (true) {

      if(user_read.read())
         user_data_from_reg = io_register & 4095;

      if(user_write.read())
         io_register = user_data_to_reg.read();

      if(io_read.read())
         io_data_from_reg = io_register & 4095;

      if(io_write.read())
         io_register = io_data_to_reg.read();

      wait();
   }
}
```

Top.h

```
#include "systemc.h"

#include "Virtual_memory.h"
#include "ALU.h"
#include "Cache.h"
#include "Memory_controller.h"
#include "Command_decoder.h"
#include "io_module.h"

#ifndef Main_bus
#define Main_bus
  SC_MODULE(Top){
    sc_in_clk clk;                  // main clock
    sc_in<bool> sreset;                // restart


    Virtual_memory vm;
    Alu alu;
    Cache cache;
    Memory_controller mc;
    Command_decoder cd;
    IO io;

    // virtual memory <=> real memory
    // to ram memory
    sc_out<bool> ram_out_read;              // read from cell
    sc_out<bool> ram_out_write;             // write in cell
    sc_out<sc_uint<10> > ram_out_addr_cell;
    sc_out<sc_uint<32> > ram_out_data_to_cell;
    sc_in<sc_uint<32> > ram_in_data_from_cell;
    sc_in<bool> ram_in_ready;
    // to hard memory
    sc_out<bool> hm_out_read;               // read from cell
    sc_out<bool> hm_out_write;              // write in cell
    sc_out<sc_uint<12> > hm_out_addr_cell;
    sc_out<sc_uint<32> > hm_out_data_to_cell;
    sc_in<sc_uint<32> > hm_in_data_from_cell;
    sc_in<bool> hm_in_ready;

    // memory <-> cache
    sc_signal<bool> m_read;                 // if data in cache not exit, get it from memory
    sc_signal<bool> m_write;
    sc_signal<sc_uint<12> > m_addr_cell;        // addr needing memory cell
    sc_signal<sc_uint<32> > m_data_to_cell;      // data wrote in memory cell
    sc_signal<sc_uint<32> > m_data_from_cell;     // data from memory cell
    sc_signal<bool> m_ready;                // cache is ready to work
    sc_signal<bool> m_save_data;

    // cache <-> memory controller
    sc_signal<bool> mc_read;                // read from cache cell
    sc_signal<bool> mc_write;                    // write to cache in cell
    sc_signal<sc_uint<12> > mc_addr_cell;          // addr needing memory cell
    sc_signal<sc_uint<32> > mc_data_to_cell;       // data wrote in memory cell
    sc_signal<sc_uint<32> > mc_data_from_cell;     // data from memory cell
    sc_signal<bool> mc_ready;                    // cache is ready to work
    sc_signal<bool> mc_save_data;

    // command decoder <-> alu and memory controller
    sc_signal<sc_uint<8>> cmd;
    sc_signal<sc_uint<12>> operand1;
    sc_signal<sc_uint<12>> operand2;
    sc_signal<sc_uint<32>> data_cell;

    // alu <-> memory controller
    sc_signal<sc_uint<4>> reg_id;
    sc_signal<bool> reg_read;                    // read from cache cell
    sc_signal<bool> reg_write;                   // write to cache in cell
    sc_signal<sc_uint<32>> reg_data_to;
    sc_signal<sc_uint<32>> reg_data_from;
    //--flags
    sc_signal<bool> fbs;
    sc_signal<bool> fss;
    sc_signal<bool> fes;
    sc_signal<bool> fnes;
    sc_signal<bool> fbes;
    sc_signal<bool> fses;

    // memory controller <-> io_module
    sc_signal<bool> io_read;
    sc_signal<bool> io_write;
    sc_signal<sc_uint<12> > io_data_to_reg;
    sc_signal<sc_uint<12> > io_data_from_reg;

    // User <-> io_module
    sc_in<bool> user_read;
    sc_in<bool> user_write;
    sc_in<sc_uint<12> > user_data_to_reg;
    sc_out<sc_uint<12> > user_data_from_reg;

    void main_cycle();

    SC_CTOR(Top) : vm("memory"), cache("cache"), mc("mc"),
        alu("alu"), cd("cd"), io("io"){

      // 1. MEMORY
      vm.clk(clk);
      vm.sreset(sreset);
      // to cache
      vm.ch_out_data_from_cell(m_data_from_cell);
      vm.ch_in_data_to_cell(m_data_to_cell);
      vm.ch_in_addr_cell(m_addr_cell);
      vm.ch_in_read(m_read);
      vm.ch_in_write(m_write);
```

```
vm.ch_out_ready(m_ready);
vm.ch_in_save_all(m_save_data);
// to real memory
vm.ram_out_read(ram_out_read);
vm.ram_out_write(ram_out_write);
vm.ram_out_addr_cell(ram_out_addr_cell);
vm.ram_out_data_to_cell(ram_out_data_to_cell);
vm.ram_in_data_from_cell(ram_in_data_from_cell);
vm.ram_in_ready(ram_in_ready);
vm.hm_out_read(hm_out_read);
vm.hm_out_write(hm_out_write);
vm.hm_out_addr_cell(hm_out_addr_cell);
vm.hm_out_data_to_cell(hm_out_data_to_cell);
vm.hm_in_data_from_cell(hm_in_data_from_cell);
vm.hm_in_ready(hm_in_ready);

// 2. CACHE
cache.clk(clk);
cache.sreset(sreset);
// to memory controller
cache.mc_data_from_cell(mc_data_from_cell);
cache.mc_data_to_cell(mc_data_to_cell);
cache.mc_addr_cell(mc_addr_cell);
cache.mc_read(mc_read);
cache.mc_write(mc_write);
cache.mc_ready(mc_ready);
cache.mc_save_all(mc_save_data);
// to memory
cache.m_read(m_read);                    // if data in cache not exit, get it from memory
cache.m_write(m_write);
cache.m_addr_cell(m_addr_cell);          // addr needing memory cell
cache.m_data_to_cell(m_data_to_cell);    // data wrote in memory cell
cache.m_data_from_cell(m_data_from_cell);   // data from memory cell
cache.m_ready(m_ready);
cache.m_save_all(m_save_data);

// 3. COMANDS DECODER
cd.clk(clk);
cd.sreset(sreset);
cd.cmd(cmd);
cd.operand1(operand1);
cd.operand2(operand2);
cd.data_cell(data_cell);

// 5. ALU
alu.clk(clk);
alu.sreset(sreset);
// to command decoder
alu.cmd(cmd);
alu.operand1(operand1);
alu.operand2(operand2);
// to memory controller
// --flag
```

```
        alu.alu_fbs(fbs);
        alu.alu_fss(fss);
        alu.alu_fes(fes);
        alu.alu_fnes(fnes);
        alu.alu_fbes(fbes);
        alu.alu_fses(fses);
        // --registers and acums
        alu.alu_reg_id(reg_id);
        alu.alu_reg_read(reg_read);
        alu.alu_reg_write(reg_write);
        alu.alu_reg_data_to(reg_data_to);
        alu.alu_reg_data_from(reg_data_from);


        // 6. MEMORY CONTROLLER
        mc.clk(clk);
        mc.sreset(sreset);
        // to command decoder
        mc.cd_cmd(cmd);
        mc.cd_operand1(operand1);
        mc.cd_operand2(operand2);
        mc.cd_data_cell(data_cell);
        // to alu
        // --flags
        mc.alu_fbs(fbs);
        mc.alu_fss(fss);
        mc.alu_fes(fes);
        mc.alu_fnes(fnes);
        mc.alu_fbes(fbes);
        mc.alu_fses(fses);
        // -- registers and acums
        mc.alu_reg_id(reg_id);
        mc.alu_reg_read(reg_read);
        mc.alu_reg_write(reg_write);
        mc.alu_reg_data_to(reg_data_to);
        mc.alu_reg_data_from(reg_data_from);
        // to cache
        mc.ch_read(mc_read);
        mc.ch_write(mc_write);
        mc.ch_addr_cell(mc_addr_cell);
        mc.ch_data_to_cell(mc_data_to_cell);
        mc.ch_data_from_cell(mc_data_from_cell);
        mc.ch_ready(mc_ready);
        mc.ch_save_data(mc_save_data);
        // to io_module
        mc.io_read(io_read);
        mc.io_write(io_write);
        mc.io_data_to_reg(io_data_to_reg);
        mc.io_data_from_reg(io_data_from_reg);

        // 7.IO_MODULE
        io.clk(clk);
        io.sreset(sreset);
```

```
          // to mc
          io.io_read(io_read);
          io.io_write(io_write);
          io.io_data_to_reg(io_data_to_reg);
          io.io_data_from_reg(io_data_from_reg);

          io.user_read(user_read);
          io.user_write(user_write);
          io.user_data_to_reg(user_data_to_reg);
          io.user_data_from_reg(user_data_from_reg);

          cout<<"Signals initializing is finished"<<endl;
       }

   };
#endif
```