Lab2 Report

Liron Cohen 207481268

Yuval Mor 209011543

Question 1

1. After execution of 2 cycles, the values will be:

$$r[2] = 12$$
$$r[3] = 45$$
$$r[4] = 57$$
$$r[5] = -12$$

2. The code tries to write a value to an old register (by using $spro{-}{>}\ r[1]\ =$) and also tries to read from a new register (by using $+\ sprn{-}{>}\ r[3]$).
   Both are wrong, we can only read from old registers and write to new registers.

Question 2

1. The first instruction takes 7 clock cycles:

$$IDLE \rightarrow FETCH0 \rightarrow FETCH1 \rightarrow DEC0 \rightarrow DEC1 \rightarrow EXEC0 \rightarrow EXEC1 \rightarrow FETCH0$$

   The rest of the instructions take 6 clock cycles each:

$$FETCH0 \rightarrow FETCH1 \rightarrow DEC0 \rightarrow DEC1 \rightarrow EXEC0 \rightarrow EXEC1 \rightarrow FETCH0$$

   So in total, the execution of n instructions will take $6n + 1$ clock cycles.

2. Yes, there are microarchitectures that can support memory access every clock cycle. For example, a pipelined microarchitecture.
3. Advantages of the presented microarchitecture:
   a. Simple implementation.
   b. No hazards at all.

   Disadvantages of the presented microarchitecture:
   a. We can't instructions can't run in parallel.
   b. Lower throughput.

Question 3

Implementation of the low-level simulator is attached, along with the required txt files.

Question 4

The required files are attached.

Question 5

The required files are attached.

Question 6

We were asked to implement a DMA state machine capable of copying a block of memory in the background, in parallel to continuing execution of the main program assembly code. In order to enable that, we chose to implement the DMA using threads. This way, one thread is executing the main program and the other thread is executing the DMA copy.

Our DMA state machine includes three states:

1. DMA_STATE_IDLE
    a. before any copy was called, and after HALT. In this state, the DMA does not copy anything.
2. DMA_STATE_WAIT
    a. after the first copy function was called, but before the thread was prioritized by the OS. We will also be at this state after finishing copying and before getting a new call for copy.
    Notice that if a copy operation is called while another is in the process, the second one is waiting until the first one is finished.
3. DMA_STATE_ACTIVE
    a. once the thread was prioritized by the OS and starts copying until the copy is done.

We also added the two required instructions to support DMA:

1. CPY
    a. inputs are source address, destination address, and length.
    This instruction starts the copy operation.
2. POL
    a. Input is a destination address. The operation puts in the desired address the number of bytes left to copy.

Question 7

Our DMA testing program checks the execution of copying 50 bytes from source address 50 to destination address 60. The program contains the following parts:

1. Initialization of the parameters - Initiating the source to be 50, the destination to be 60, and the length to be 50.
2. Calling the copy operation with the parameters.
3. Main program code to run in parallel - the program calculates the accumulative sum of values in memory from address 50 to 54.
4. Polling and checking for copy completion.
5. Copy validation -
    a. Putting in registers the values for source, destination, and final destination address.
    b. Loading the first byte in the source address and the destination address.
    c. If they are not the same values, write the result to failure (0) and jump to halt.
    d. If they are the same, incrementing the addresses for checking and continuing.
    e. This loop is running until we got to the final destination address.

```
// Liron Cohen 207481268
// Yuval Mor 209011543

// Initialization of the parameters
asm_cmd(ADD, 3, 1, 0, 50); // 0: R3 = 50 SOURCE
asm_cmd(ADD, 4, 1, 0, 60); // 1: R4 = 60 DESTINATION
asm_cmd(ADD, 5, 1, 0, 50); // 2: R5 = 50 LENGTH

// Calling the copy operation
asm_cmd(CPY, 4, 3, 5, 0); // 3: START COPY

// Main program code to run in parallel
asm_cmd(ADD, 2, 1, 0, 50); // 4: R2 = 50
asm_cmd(ADD, 5, 1, 0, 55); // 5: R5 = 55
asm_cmd(ADD, 3, 0, 0, 0); // 6: R3 = 0
asm_cmd(LD,  4, 0, 2, 0); // 7: R4 = MEM[R2]
asm_cmd(ADD, 3, 3, 4, 0); // 8: R3+= R4
asm_cmd(ADD, 2, 2, 1, 1); // 9: R2++
asm_cmd(JLT, 0, 2, 5, 7); // 10: if R2 < R5 jump to line 7

// Polling and checking for copy completion
asm_cmd(POL, 2, 0, 0, 0); // 11: R[2] = DMA_S->REMAIN (number of bytes left to copy)
asm_cmd(JNE, 0, 2, 0, 11); // 12: if R[2] != 0 jump to 11

// Copy validation
asm_cmd(ADD, 2, 1, 0, 1); // 13: R2 = 1 COPY CORRECT
asm_cmd(ADD, 3, 1, 0, 50); // 14: R3 = 50 SOURCE
asm_cmd(ADD, 4, 1, 0, 60); // 15: R4 = 60 DESTINATION
asm_cmd(ADD, 5, 1, 0, 100); // 16: R5 = 100  SOURCE + LENGTH
asm_cmd(LD, 6, 0, 3, 0); // 17: R6 = MEM[R3]
asm_cmd(LD, 7, 0, 4, 0); // 18: R7 = MEM[R4]
asm_cmd(JEQ, 0, 6, 7, 22); // 19: if (R[6] == R[7]) jump to 22
asm_cmd(ADD, 2, 0, 0, 0); // 20: R2 = 0 COPY WRONG
asm_cmd(JEQ, 0, 0, 0, 25); // 21: jump to 25 HALT
asm_cmd(ADD, 3, 3, 1, 1); // 22: R3++
asm_cmd(ADD, 4, 4, 1, 1); // 23: R4++
asm_cmd(JLT, 0, 3, 5, 17); // 24: if R3 < R5 jump to line 17
asm_cmd(HLT, 0, 0, 0, 0); // 25: HALT
```

The required files are attached.