Onn Rengingad 304845951
Bar Ben Ari 204034284
Barak Levy 311431894

Lab 1 - Advanced Computer Architecture

1. **Warm-up questions:**

1. asm_cmd(ADD, 5, 0, 1, 10)

2. asm_cmd(ADD, 5, 0, 1, -512)

3. asm_cmd(SUB, 4, 3, 4, dm) - The immediate is ignored so 0 was randomly chosen (dm = doesn't matter).

4. asm_cmd(JLT, dm, 3, 1, x) - Destination register is ignored (7 can be randomly chosen), and x represents any immediate value.

5. By dividing the 32-bits constant into two 16-bits immediate strings (A 16-bits MSB bitstring - **immMSB**, and a 16-bits LSB bitstring - **immLSB**) it is possible to load the 32-bits constant. First apply an ADD operation to set the register's 16 LSB bits to be the 16-bits LSB string, and then apply a LHI operation to set the register's 16 MSB bits to the MSB string of the 32-bits constant. Overall, we get:
   asm_cmd(ADD, 5, 0, 1, immLSB)
   asm_cmd(LHI, 5, dm, dm, immMSB)

6. Subroutine calls define the situation where the instruction pointer jumps between different sections of the memory, and allows executing commands that are not adjacent to one another. The Flow Control architecture commands e.g JLT, JLE, JEQ etc. are meant to enable relocating the instruction pointer to different sections, while also saving the original address pointed by the instruction pointer to register 7 if the relocation occurs. Thus, when a jump occurs, the set of commands, in the subroutine section, is executed and then the instruction pointer returns to the following address after the one stored in register 7, and by so, subroutine calls are enabled.

2. **Example program:**

1. The program computes the partial sum (sum of all previous cells, include the current cell) of the sequence stored in 15-22 memory cells.
2. The inputs stored at 15-22 memory cells – mem [15] until mem [22].

Onn Rengingad 304845951
Bar Ben Ari 204034284
Barak Levy 311431894

3. The outputs stored at 16-22 memory cells – mem [16] until mem [22].
4. Commented version:

```
/*
 * Program starts here
 */
asm_cmd(ADD, 2, 1, 0, 15);// 0: R2 = 15
asm_cmd(ADD, 3, 1, 0, 1); // 1: R3 = 1
asm_cmd(ADD, 4, 1, 0, 8); // 2: R4 = 8
asm_cmd(JEQ, 0, 3, 4, 11); // 3: if R4 == R3 jump to pc = 11
asm_cmd(LD,  5, 0, 2, 0); // 4: R5 = mem[R2]
asm_cmd(ADD, 2, 2, 1, 1); // 5: R2 += 1
asm_cmd(LD,  6, 0, 2, 0); // 6: R6 = mem[R2]
asm_cmd(ADD, 6, 6, 5, 0); // 7: R6 = R6 + R5
asm_cmd(ST,  0, 6, 2, 0); // 8: mem[R2] = R6
asm_cmd(ADD, 3, 3, 1, 1); // 9: R3 += 1
asm_cmd(JEQ, 0, 0, 0, 3); // 10: jump to pc = 3
asm_cmd(HLT, 0, 0, 0, 0); // 11: halt
```

5. New program having fewer references to memory:

```
/*
 * Program starts here
 */
asm_cmd(ADD, 2, 1, 0, 15);// 0: R2 = 15
asm_cmd(ADD, 3, 1, 0, 1); // 1: R3 = 1
asm_cmd(ADD, 4, 1, 0, 8); // 2: R4 = 8
asm_cmd(LD,  6, 0, 2, 0); // 3: R6 = mem[R2]
asm_cmd(JEQ, 0, 3, 4, 11); // 4: if R4 == R3 jump to pc = 11
asm_cmd(ADD, 5, 0, 6, 0); // 5: R5 = R6
asm_cmd(LD,  6, 0, 2, 0); // 6: R6 = mem[R2]
asm_cmd(ADD, 6, 6, 5, 0); // 7: R6 = R6 + R5
asm_cmd(ADD, 2, 2, 1, 1); // 8: R2 += 1
asm_cmd(ST,  0, 6, 2, 0); // 9: mem[R2] = R6
asm_cmd(ADD, 3, 3, 1, 1); // 10: R3 += 1
asm_cmd(JEQ, 0, 0, 0, 3); // 11: jump to pc = 3
asm_cmd(HLT, 0, 0, 0, 0); // 12: halt
```

Onn Rengingad 304845951
Bar Ben Ari 204034284
Barak Levy 311431894

3. **ISS simulator Testing #1:**

1. Assembly code of the program:

```
/* Initialize variables */
asm_cmd(LD, 2, 0, 1, 1000); // 0: R2 = Memory[1000] Loading Multiplicand
asm_cmd(LD, 3, 0, 1, 1001); // 1: R3 = Memory[1001] Loading Multiplier
asm_cmd(ADD, 5, 0, 0, 0);   // 2: R5 = 0 => R5 is defined to be the RESULT of the multiplication

/* Main loop that computes the multiplication of the two numbers */
asm_cmd(AND, 4, 3, 1, 1);   // 3: R4 = R3&1 => The LSB of R3 describes the contribution of the Multiplier   (If it is 1, than sum it, otherwise, do
asm_cmd(JEQ, 0, 4, 0, 6);   // 4: JUMP - skipping the Multiplier's contribution, because its' CURRENT LSB is zero
asm_cmd(ADD, 5, 5, 2, 0);   // 5: R5 = R5+R2 => Result update   (If the current Multiplier LSB is 1, than sum it, otherwise, don't)
asm_cmd(LSF, 2, 2, 1, 1);   // 6: R2 = R2<<1 => Left shift for the becoming-greater Multiplicand
asm_cmd(RSF, 3, 3, 1, 1);   // 7: R3 = R3>>1 => Right shift of the Multiplier   (for further scanning)
asm_cmd(JNE, 0, 3, 0, 3);   // 8: JUMP to another addition if the Multiplier hasn't been scanned completely   (meaning - different than zero)

/* Last section that saves the multiplication result in the needed memory slot and exits */
asm_cmd(ST, 0, 5, 1, 1002); // 9: Memory[1002] = R5 => Storing the multiplication result at 1002
asm_cmd(HLT, 0, 0, 0, 0);   // 10: HALT!
```

2. The files attached within the folder.

4. **ISS simulator Testing #2:**

1. Assembly code of the program:

```
/* initialize variables */
asm_cmd(ADD, 2, 1, 0, 1); // 0: R2 = 1
asm_cmd(ST, 0, 2, 1, 1000); // 1: mem[1000] = R2
asm_cmd(ADD, 3, 0, 2, 0); // 2: R3 = R2
asm_cmd(ST, 0, 3, 1, 1001); // 3: mem[1001] = R3
asm_cmd(ADD, 4, 0, 1, 1038); // 4: R4 = 1038
asm_cmd(ADD, 5, 0, 1, 1002); // 5: R5 = 1002

/* loop that calculate the 3-40 Fibonacci numbers */
asm_cmd(JEQ, 0, 4, 5, 13); // 6: if R4 == R5 jump to pc = 13
asm_cmd(ADD, 6, 2, 3, 0); // 7: R6 = R2 + R3
asm_cmd(ST, 0, 5, 6, 0); // 8: mem[R5] = R6
asm_cmd(ADD, 2, 3, 0, 0); // 9: R2 = R3
asm_cmd(ADD, 3, 6, 0, 0); // 10: R3 = R6
asm_cmd(ADD, 5, 5, 1, 1); // 11: R5 += 1
asm_cmd(JEQ, 0, 0, 0, 6); // 12: jump to pc = 6
asm_cmd(HLT, 0, 0, 0, 0); // 13: halt

```

2. The files attached within the folder.
3. If we run this program for the first 100 elements of the Fibonacci sequence we can get two problems:

a. We can get a negative – if the MSB bit is '1' bit the iss will give as a negative number (two's complement).
b. Big Fibonacci number (like 99 - 1333DB76A7C594BFC3) can't be represented in the sram (only 9 hexadecimal smbols).

5. **ISS simulator implementation:**

1. The files attached within the folder.