Samuel Youssef
May 12, 2018

CSC 34200 - Computer Organization

Instructor: Prof. Zheng Peng

Lab **7**

1. The MS PowerPoint file is uploaded on blackboard. The diagram for the complete design is shown below:

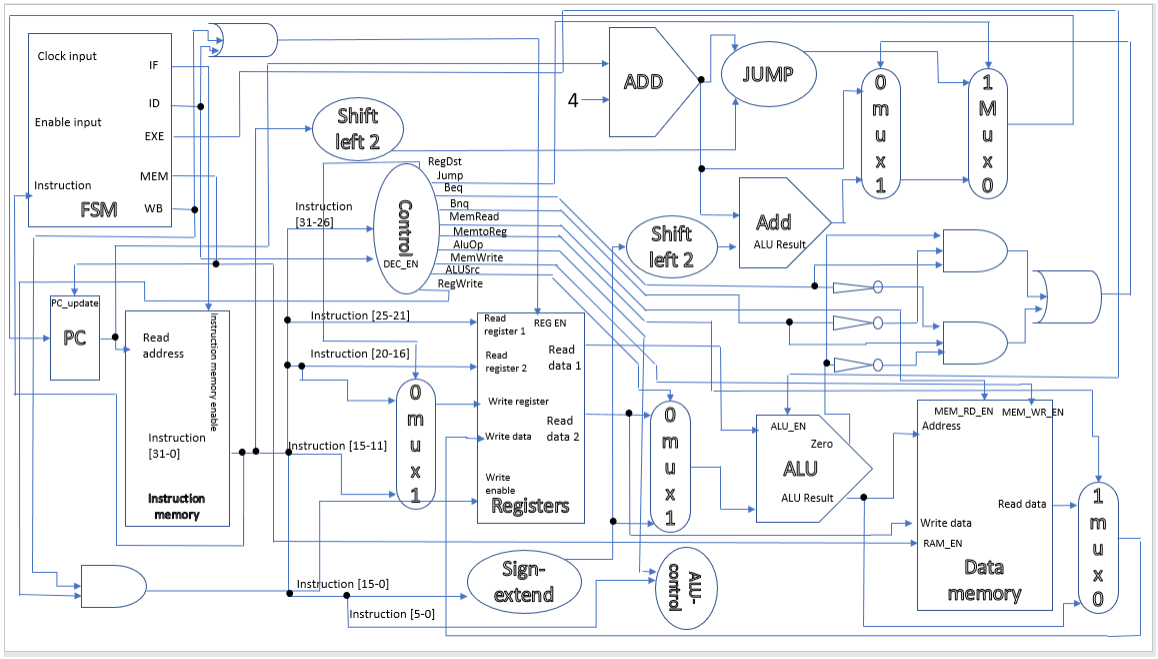
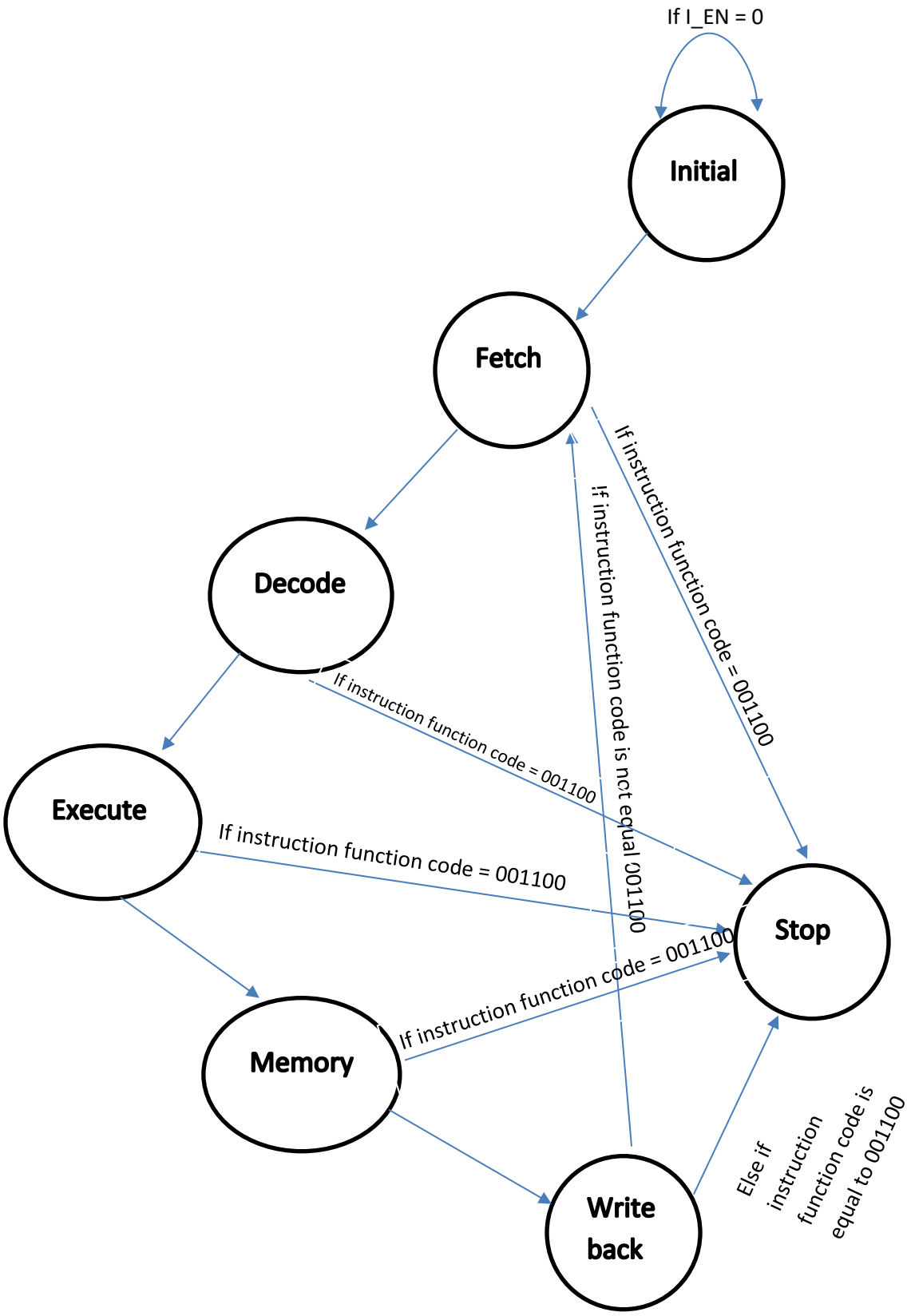


Figure 1: The complete design of MIPS processor.

2. All components of the processor have been implemented successfully. Simulations give targeted results. All Xilinx project files have been uploaded to blackboard.

3. There are seven states in the design of the processor (initial state, IF state, ID state, EXE state, MEM state, WE state, Stop state). The state diagram is shown below:

Note → the function code for SYSCALL instruction is 001100.


If I_EN = 0
Initial
Fetch
Decode
Execute
Memory
Write back
Stop
If instruction function code = 001100
If instruction function code is not equal 001100
If instruction function code = 001100
If instruction function code = 001100
If instruction function code = 001100
Else if instruction function code is equal to 001100

- Note: In this design, all five stages were supposed to be done in a single cycle (ID stage only activated when state = ID). However as instructed, each stage of execution of one instruction was done in a single cycle (total of five cycles for five states). There were multiple fixations of code in multiple modules, and those fixations are shown below:
    1. The decoder module (from lab 5 submission) have been fixed to lock the output signals. Picture of the locked outputs are shown below.

```
architecture Behavioral of DEC is
signal O_DEC_RegDst_temp: std_logic;
signal O_DEC_Jump_temp : std_logic;
signal O_DEC_Beq_temp : std_logic;
signal O_DEC_Bne_temp : std_logic;
signal O_DEC_MemRead_temp: std_logic;
signal O_DEC_MemtoReg_temp: std_logic;
signal O_DEC_ALUOp_temp: std_logic_vector(1 downto 0);
signal O_DEC_MemWrite_temp : std_logic;
signal O_DEC_ALUSrc_temp : std_logic;
signal O_DEC_RegWrite_temp : std_logic;

begin
O_DEC_RegDst <= O_DEC_RegDst_temp;
O_DEC_Jump <= O_DEC_Jump_temp;
O_DEC_Beq <= O_DEC_Beq_temp;
O_DEC_Bne <= O_DEC_Bne_temp;
O_DEC_MemRead <= O_DEC_MemRead_temp;
O_DEC_MemtoReg <= O_DEC_MemtoReg_temp;
O_DEC_ALUOp <= O_DEC_ALUOp_temp;
O_DEC_MemWrite <= O_DEC_MemWrite_temp;
O_DEC_ALUSrc <= O_DEC_ALUSrc_temp;
O_DEC_RegWrite  <= O_DEC_RegWrite_temp;
process (I_DEC_EN , I_DEC_Opcode)
begin
```

    ------Fixed code of decoder module------

    2. The register module (submission of lab 6) have been changed to fix a bug in the code (the code initially allowed writing into the register with I_REG_EN = 0). The bug has been removed and the register in the new module can only write data if I_REG_WE and I_REG_EN are both set to ‘1’. The fixed code is shown below.

```
if I_REG_EN = '1' then
   temp1:= to_integer(unsigned(I_REG_SEL_RS));
   temp2:= to_integer(unsigned(I_REG_SEL_RT));
   temp_data_a <= REG(temp1);
   temp_data_b <= REG(temp2);

   if I_REG_WE = '1' and I_REG_SEL_RD /= "00000" then
   temp3 := to_integer(unsigned(I_REG_SEL_RD));
   REG(temp3) <= I_REG_DATA_RD;
   end if;
end if;
end process;
```

----------fixed Register module----------

- Routing different signals:
  - O_FSM_IF → I_ROM_EN.
  - O_FSM_DEC → I_DEC_EN and I_REG_EN.
  - O_FSM_EX → I_ALU_EN.
  - O_FSM_MEM → I_RAM_EN and I_PC_Update. Branch if equal instruction is the longest instruction that might affect the process of updating the PC. The decision of branching is made right after the execution stage. So, we can update the PC at the memory stage which is the stage right after the execution stage.
  - O_FSM_WB → I_REG_EN.
- The design that has been implemented in this project is almost the same as the design that was given during the lecture with some additional components described below:
  - An OR gate was added; its output goes to I_REG_EN and it takes two inputs: O_FSM_DEC, and O_FSM_WB. The addition of this OR gate to the design is self-explanatory as we want to enable the register during both the decoding stage and the write back stage.
  - Three inverters, two three-input AND gates, and one two-input OR gate to support branch instructions (BEQ, BNE).
  - And gate with two inputs and one output. The inputs are O_FSM_WB, and O_DEC_RegWrite. The output goes to I_REG_WE. If this And gate was not added, the Register array would have shown conflicted values.

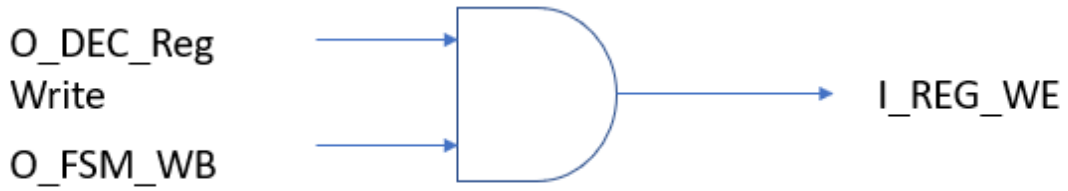


figure 2: added circuit to fix problem of overwriting registers without necessity to do so.

After the fixations of the code of multiple modules, and the added circuitry mentioned above, I was able to set each control signal to its corresponding stage only. Picture of the FSM is shown below.

```
        if I_FSM_EN = '1' then
           if I_FSM_INST (5 downto 0) /= "001100" then
                    if state = INITIAL then state <= FETCH; identifier <= 1;
                    elsif state = FETCH  then state <= DECODE;
                    elsif state = DECODE  then state <= EXECUTE;
                    elsif state = EXECUTE  then state <= MEMORY;
                    elsif state = MEMORY  then state <= WRITE_BACK;
                    elsif state = WRITE_BACK and identifier <= 1 then state <= FETCH;
                    --else state <= STOP;
                    end if;
           else
              state <= STOP;
           end if;
        else
           if state = INITIAL then state <= INITIAL;
           else state <= STOP;
           end if;
        end if;
     end if;
  end process;

  stateFSM: process(state)
  begin
     if state = FETCH then
        O_FSM_IF <= '1'; O_FSM_ID <= '0'; O_FSM_EX <= '0'; O_FSM_ME <= '0'; O_FSM_WB <= '0';
     elsif state = DECODE then
        O_FSM_IF <= '0'; O_FSM_ID <= '1'; O_FSM_EX <= '0'; O_FSM_ME <= '0'; O_FSM_WB <= '0';
     elsif state = EXECUTE then
        O_FSM_IF <= '0'; O_FSM_ID <= '0'; O_FSM_EX <= '1'; O_FSM_ME <= '0'; O_FSM_WB <= '0';
     elsif state = MEMORY then
        O_FSM_IF <= '0'; O_FSM_ID <= '0'; O_FSM_EX <= '0'; O_FSM_ME <= '1'; O_FSM_WB <= '0';
     elsif state = WRITE_BACK then
        O_FSM_IF <= '0'; O_FSM_ID <= '0'; O_FSM_EX <= '0'; O_FSM_ME <= '0'; O_FSM_WB <= '1';
     else
        O_FSM_IF <= '0'; O_FSM_ID <= '0'; O_FSM_EX <= '0'; O_FSM_ME <= '0'; O_FSM_WB <= '0';
     end if;
```

--------picture of FSM module after fixations has been done on all components of process--------

- At fetch stage: O_FSM_IF = ‘1’, O_FSM_ID = ‘0’, O_FSM_EX = ‘0’, O_FSM_MEM = ‘0’, O_FSM_WB = ‘0’.
- At decode stage: O_FSM_IF = ‘’0, O_FSM_ID = ‘1’, O_FSM_EX = ‘0’, O_FSM_MEM = ‘0’, O_FSM_WB = ‘0’.

- At execution stage: O_FSM_IF = ‘0’, O_FSM_ID = ‘0’, O_FSM_EX = ‘1’, O_FSM_MEM = ‘0’, O_FSM_WB = ‘0’.
- At memory stage: O_FSM_IF = ‘0’, O_FSM_ID = ‘0’, O_FSM_EX = ‘0’, O_FSM_MEM = ‘1’, O_FSM_WB = ‘0’.
- At write back stage: O_FSM_IF = ‘0’, O_FSM_ID = ‘0’, O_FSM_EX = ‘0’, O_FSM_MEM = ‘0’, O_FSM_WB = ‘1’.

❖ The program is started if I_EN = ‘1’ → (processor is enabled). The program will move to stop stage if I_EN = ‘0’ at any point during the simulation. If a SYSCALL instruction was encountered, the program also moves to stop stage and it terminates.

❖ The FSM module checks (in every instruction during the execution of the program) for the function code of SYSCALL instruction which is 001100. If this function code is encountered, the processor moves to stop stage and the program is terminated.

4. The first instruction is addi $8, $0, 0x00002000 (0x20082000):

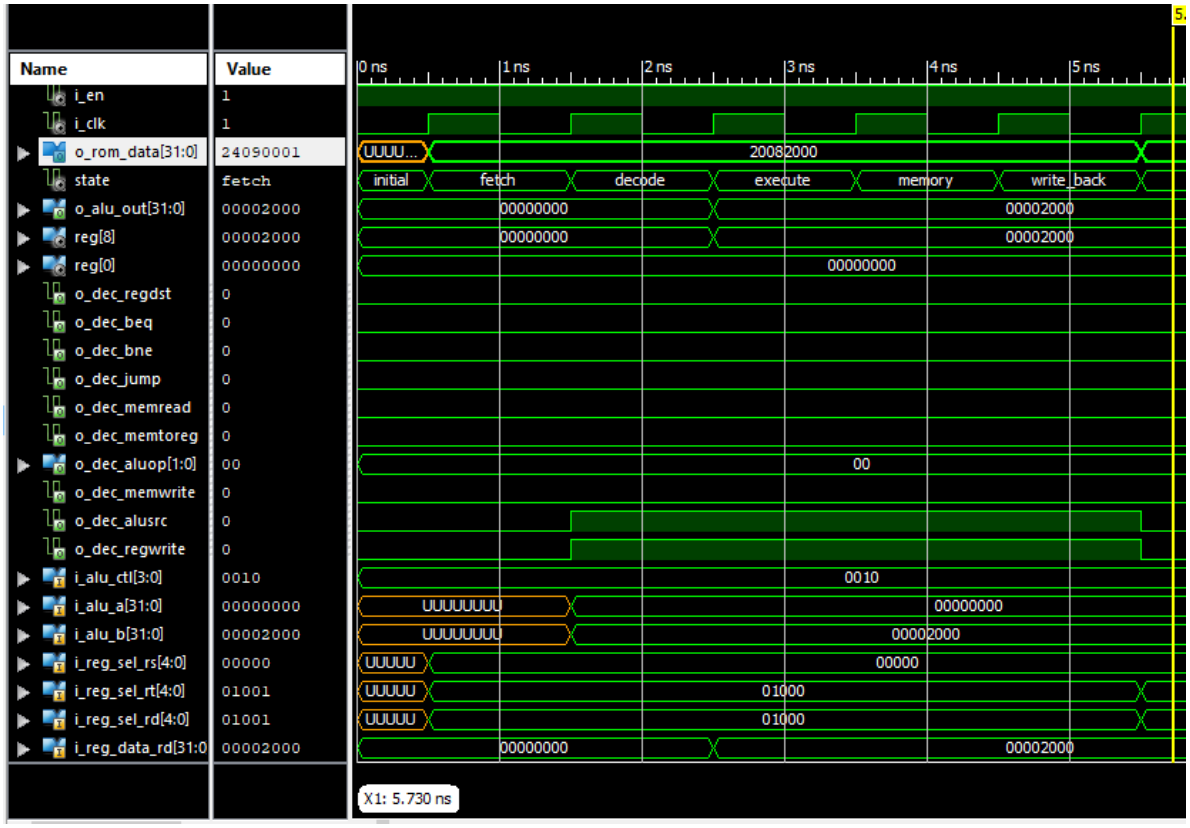


figure 3: addi $8, $0, 0x00002000 (0x20082000) instruction.

Signal	Value	Note
Regdst	0	Not R-type instruction
Bne	0	Not branch instruction
Beq	0	Not branch instruction
Memread	0	Not reading from the mem.
Memtoreg	0	Not reading from memory and writing to a register
Aluop	**0b** 00	Addition on ALU
Memwrite	0	Not writing to a memory location
Alusrc	1	Second ALU input is an immediate value
Regwrite	1	Writing back register $8
Alu_ctr	**0b** 0010	Addition on inputs to ALU
I_alu_a	**0x** 00000000	First input to ALU
I_alu_b	**0x** 00002000	Second input to ALU
I_reg_sel_rs	**0b** 00000	Source register in the instruction ($0)
I_reg_sel_rt	**0b** 01000	Target register in the instruction ($8)
I_reg_sel_rd	**0b** 01000	Destination register in the instruction ($8)
I_reg_data_rd	**0x** 00002000	Result of ALU operation to be written to destination register
Reg_file [8]	**0x** 00002000	Value of register $8 after writing back stage
Reg_file [0]	**0x** 00000000	Value of register ($0) (source register)

The second instruction is addiu $9, $0, 0x00000001 (0x24090001):

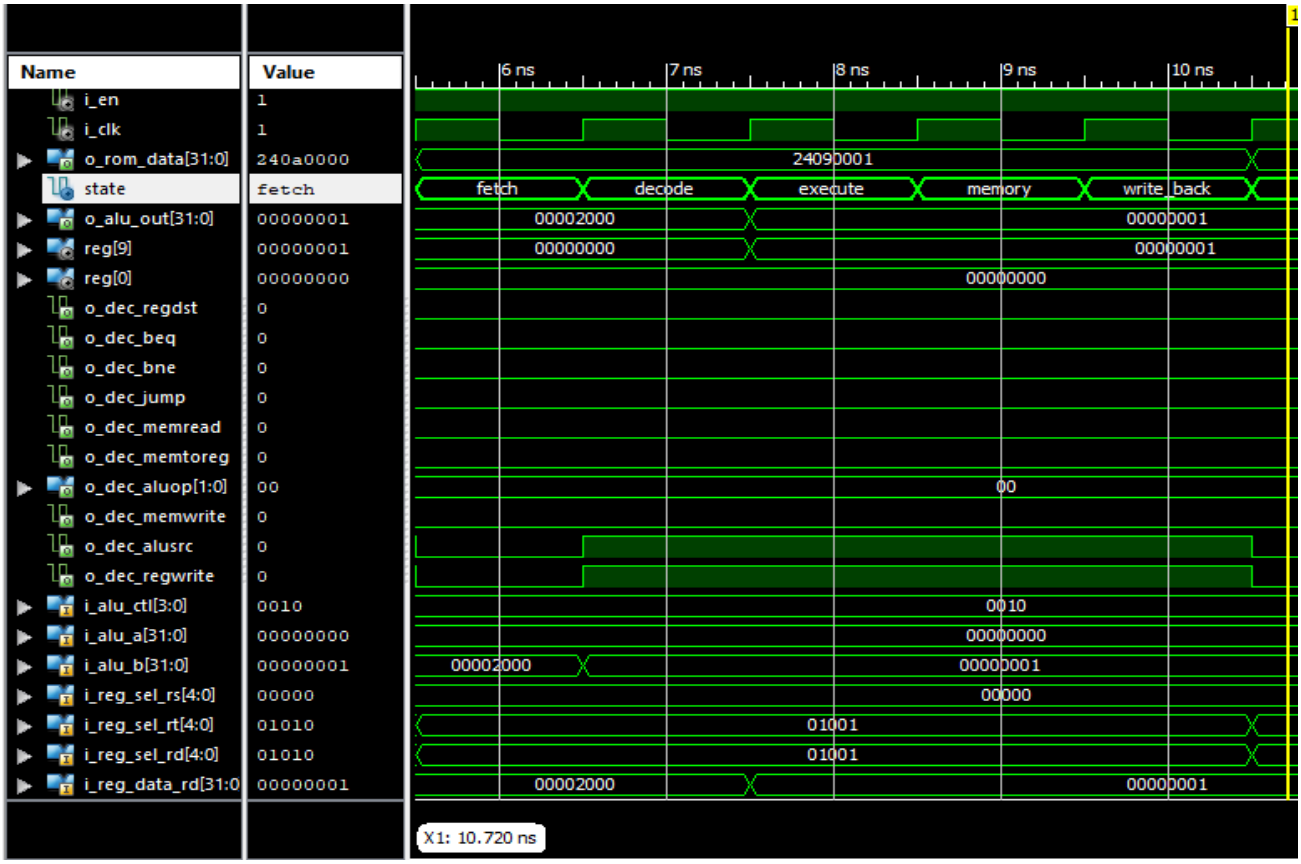


Figure 4: addiu $9, $0, 0x00000001 (0x24090001) instruction.

Signal	Value	Note
Regdst	0	Not R-type instruction
Bne	0	Not branch instruction
Beq	0	Not branch instruction
Memread	0	Not reading from the mem.
Memtoreg	0	Not reading from memory and writing to a register
Aluop	**0b** 00	Addition on ALU
Memwrite	0	Not writing to a memory location
Alusrc	1	Second ALU input is an immediate value
Regwrite	1	Writing back register $9
Alu_ctr	**0b** 0010	Addition on inputs to ALU
I_alu_a	**0x** 00000000	First input to ALU
I_alu_b	**0x** 00000001	Second input to ALU
I_reg_sel_rs	**0b** 00000	Source register in the instruction ($0)
I_reg_sel_rt	**0b** 01001	Target register in the instruction ($9)

I_reg_sel_rd	**0b** 01001	Destination register in the instruction ($9)
I_reg_data_rd	**0x** 00000001	Result of ALU operation to be written to destination register
Reg_file [9]	**0x** 00000001	Value of register $9 after writing back stage
Reg_file [0]	**0x** 00000000	Value of register ($0) (source register)

The third instruction sw $11, 0x00000000($8) (0xad0b0000):

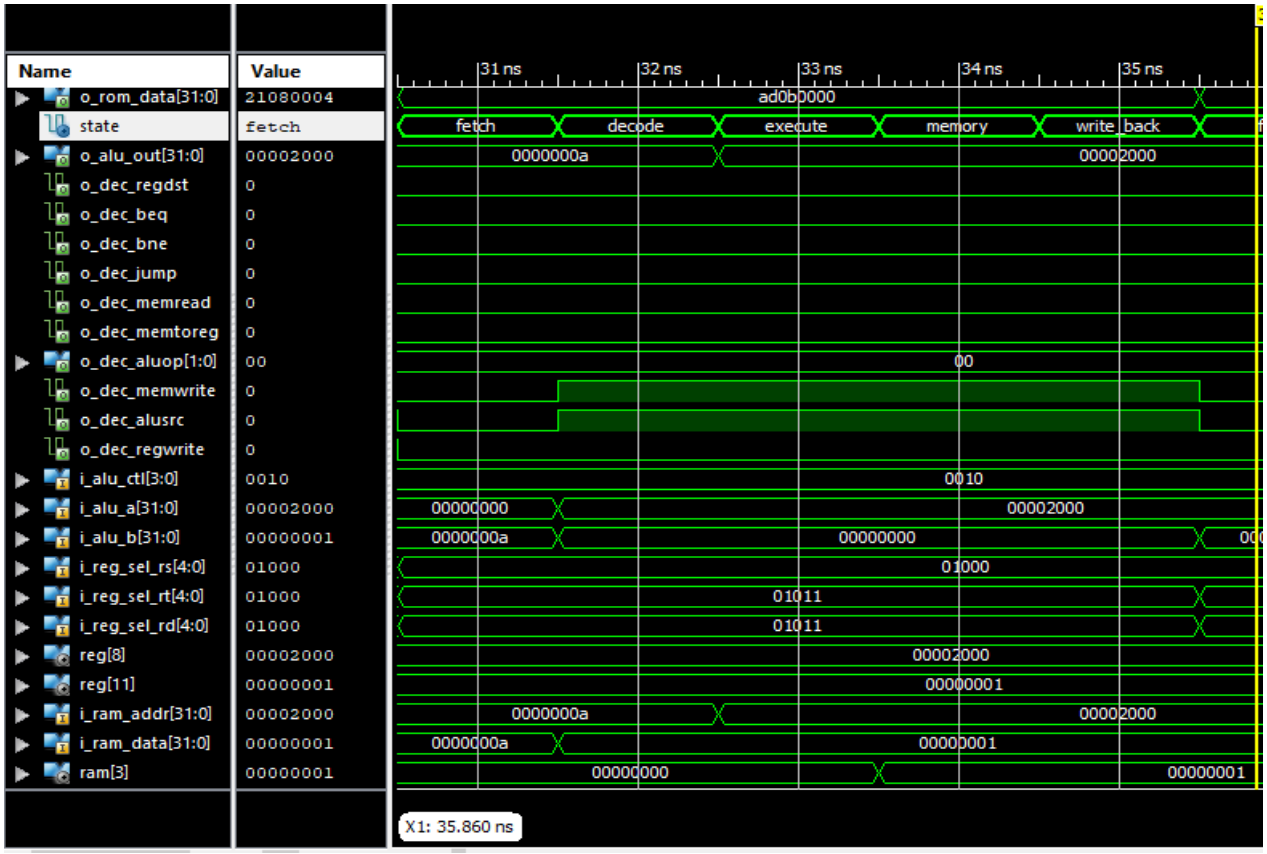


Figure 5: sw $11, 0x00000000($8) (0xad0b0000) instruction.

Signal	Value	Note
Regdst	0	Not R-type instruction
Bne	0	Not branch instruction
Beq	0	Not branch instruction
Memread	0	Not reading from the mem.
Memtoreg	0	Not reading from memory and writing to a register
Aluop	**0b** 00	Addition on ALU
Memwrite	1	Writing to a memory

		location (offset = 0x00000000) and (base address = 0x 00002000)
Alusrc	1	Second ALU input is an immediate value
Regwrite	0	Not Writing back to a register
Alu_ctr	**0b** 0010	Addition on inputs to ALU
I_alu_a	**0x** 00002000	First input to ALU
I_alu_b	**0x** 00000000	Second input to ALU
I_reg_sel_rs	**0b** 01000	Source register in the instruction ($8)
I_reg_sel_rt	**0b** 01011	Target register in the instruction ($11)
I_reg_sel_rd	**0b** 01011	Destination register in the instruction ($11)
Reg_file [8]	**0x** 00002000	The content of register $8
Reg_file [11]	**0x** 00000001	The content of register $11
O_alu_out	**0x** 00002000	Result of ALU operation (address of memory location to write data)
I_ram_addr	**0x** 00002000	The address to which the data will be written
I_ram_data	**0x** 00000001	The data to be written to specified memory location
Ram [3]	**0b** 00000001	Data has been successfully moved to specified memory location

The fourth instruction addu $11, $9, $10 (0x012a5821):

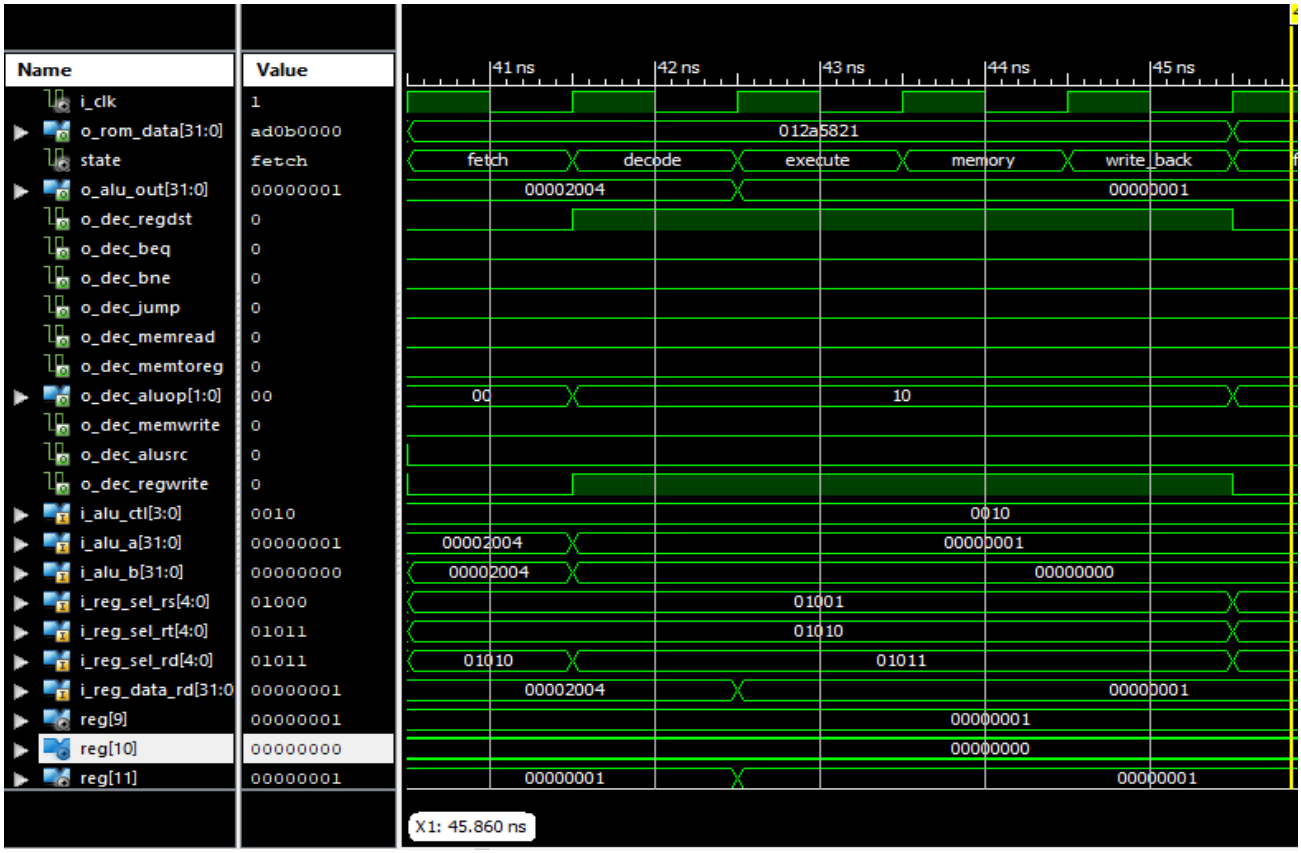


Figure 6: addu $11, $9, $10 (0x012a5821) instruction.

Signal	**Value**	**Note**
Regdst	1	R-type instruction
Bne	0	Not branch instruction
Beq	0	Not branch instruction
Memread	0	Not reading from the mem.
Memtoreg	0	Not reading from memory and writing to a register
Aluop	**0b** 10	Addition on ALU according to the function code of the instruction
Memwrite	0	Not writing to a memory location
Alusrc	0	Second ALU input is coming from a register
Regwrite	1	Writing back to register $11
Alu_ctr	**0b** 0010	Addition on inputs to ALU
I_alu_a	**0x** 00000001	First input to ALU (register $9)

I_alu_b	**0x** 00000000	Second input to ALU (register $10)
I_reg_sel_rs	**0b** 01001	Source register in the instruction ($9)
I_reg_sel_rt	**0b** 01010	Target register in the instruction ($10)
I_reg_sel_rd	**0b** 01011	Destination register in the instruction ($11)
I_reg_data_rd	**0x** 00000001	Result of ALU operation to be written to destination register
Reg_file [9]	**0x** 00000001	Value of register $9
Reg_file [10]	**0x** 00000000	Value of register ($10)
Reg_file [11]	**0x** 00000001	Value of register ($11) after write-back stage

Fifth instruction bne $13, $12, 0xfffffff9 (0x15acfff9):

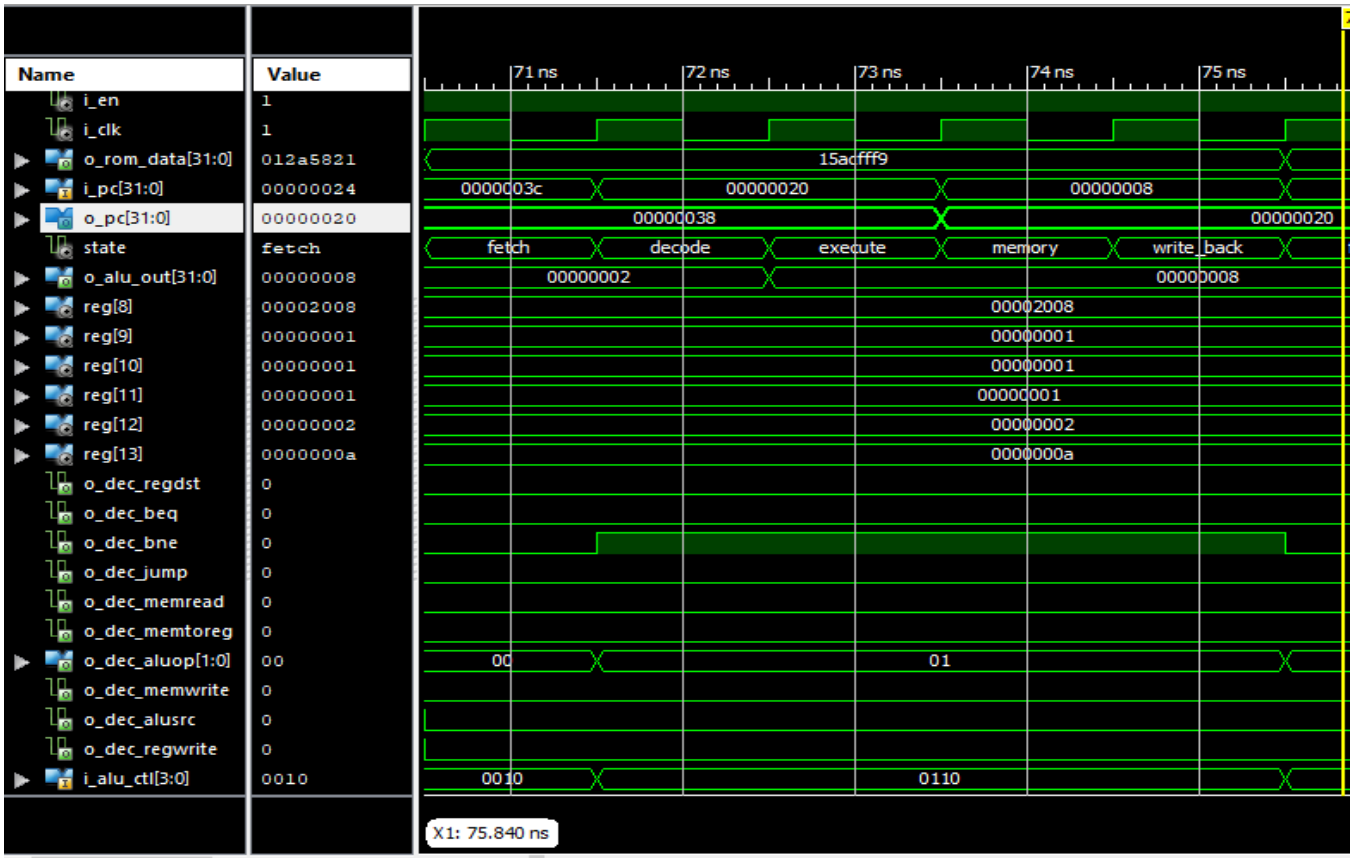


figure 7: bne $13, $12, 0xfffffff9 (0x15acfff9) instruction.

Signal	Value	Note
Regdst	0	Not R-type instruction
Bne	1	Branch if not equal instruction

Beq	0	Not branch if equal instruction
Memread	0	Not reading from memory
Memtoreg	0	Not reading from memory and writing back to register
Aluop	**0b** 01	Doing subtraction on ALU
Memwrite	0	Not writing to a memory location
Alusrc	0	Second input to ALU is from a register
Regwrite	0	Not writing back to a register
Alu_ctr	**0b** 0110	Subtraction on ALU
Reg_file [12]	**0x** 00000002	The value in the first register
Reg_file [13]	**0x** 0000000a	The value in the second register
Alu_out	**0x** 00000008	Result of subtraction on ALU
O_pc	**0x** 00000038	Current instruction address
I_pc	**0x** 0000003c	Next instruction address
I_pc*	**0x** 00000020	After the ALU subtraction, it turns out that a branch operation is needed. The branch target address is 0x0000003c + 4×0xfffffff9=0x00000020. Therefore, the I_pc value is updated to 0x00000020.

Sixth instruction Syscall (0x0000000c):

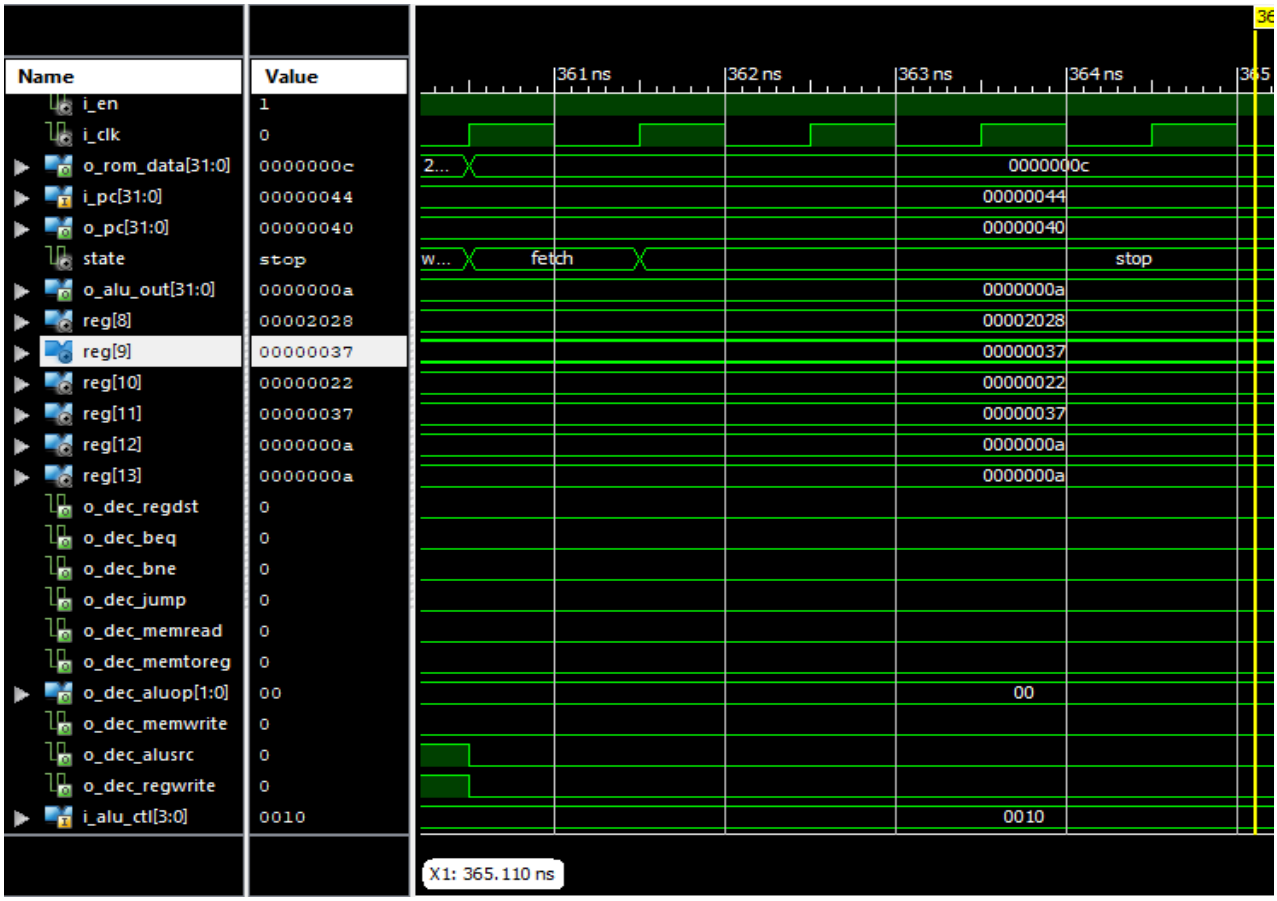


Figure 8: Syscall (0x0000000c) instruction.

Signal	**Value**	**Note**
Regdst	0	Not R-type instruction
Bne	0	Not branch instruction
Beq	0	Not branch instruction
Memread	0	Not reading from memory
Memtoreg	0	Not reading from memory and writing back to a register
Aluop	**0b** 00	Doing addition on ALU. Essentially the result of addition operation will not be used anywhere as this is a call to terminate the program
Memwrite	0	Not writing to any memory location
Alusrc	0	Second input of ALU is

		coming from a register
Regwrite	0	Not writing to a register
Alu_ctr	**0b** 0010	On addition ALU
Reg_file [12]	**0x** 0000000a	The value in the first register
Reg_file [13]	**0x** 0000000a	The value in the second register
Alu_out	**0x** 0000000a	Result of addition on ALU
O_pc	**0x** 00000040	Current instruction address
I_pc	**0x** 00000044	Next instruction address
I_pc*	**0x** 00000044	Next instruction address. Not branches. The program terminates at O_pc so this address is not used

5. Show below are the content of the register array and the data memory array after the program had finished execution:

	0	1
0	00000000	00000000
2	0000000A	00000000
4	00000000	00000000
6	00000000	00000000
8	00002028	00000037
10	00000022	00000037
12	0000000A	0000000A
14	00000000	00000000
16	00000000	00000000
18	00000000	00000000
20	00000000	00000000
22	00000000	00000000
24	00000000	00000000
26	00000000	00000000
28	00000000	00000000
30	00000000	00000000

figure 9: The content of register file (in hexadecimal format) after finishing execution of the program.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	1	0	0	0	1	0	0	0	2	0	0	0	3
16	0	0	0	5	0	0	0	8	0	0	0	13	0	0	0	21
32	0	0	0	34	0	0	0	55	0	0	0	0	0	0	0	0
48	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
64	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
80	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
96	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
112	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
128	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
144	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
160	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
176	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
192	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
208	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
224	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
240	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 10: the content of the RAM array (in unsigned decimal format) after finishing execution of the program.

- The design generates the targeted results→ first 10 Fibonacci numbers.