**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:

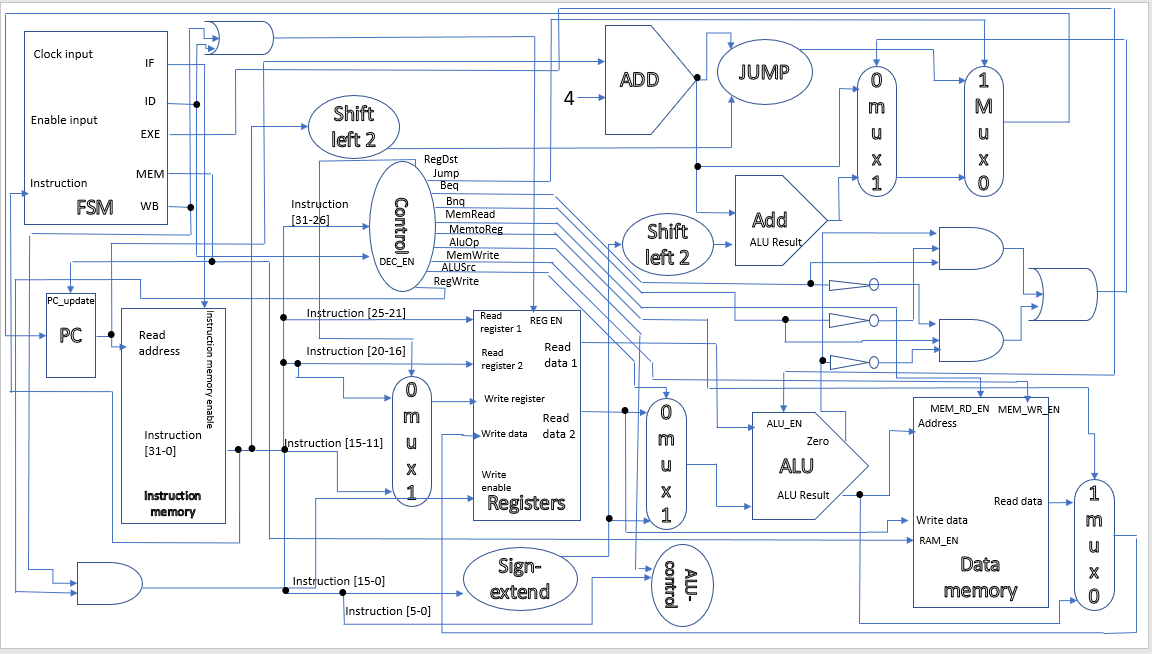


Figure 1: The complete design of MIPS processor.

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

If instruction function code = 001100

Else if instruction function code is equal to 001100

If instruction function code is not equal 001100

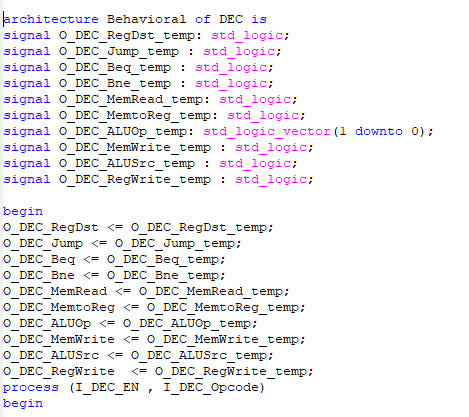
If instruction function code = 001100

If instruction function code = 001100

If instruction function code = 001100

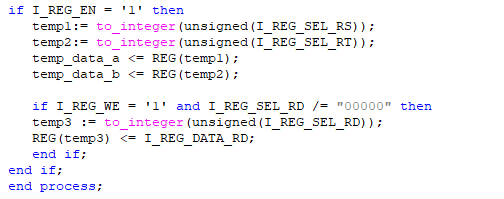
* Note: In this design, all five stages were supposed to be done in a single cycle (ID stage only activated when state = ID). However as instructed, each stage of execution of one instruction was done in a single cycle (total of five cycles for five states). There were multiple fixations of code in multiple 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.



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

1. 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.



----------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.

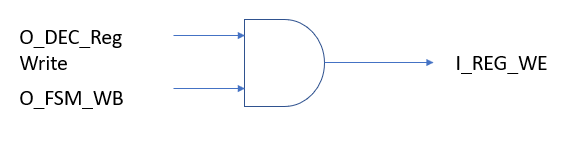
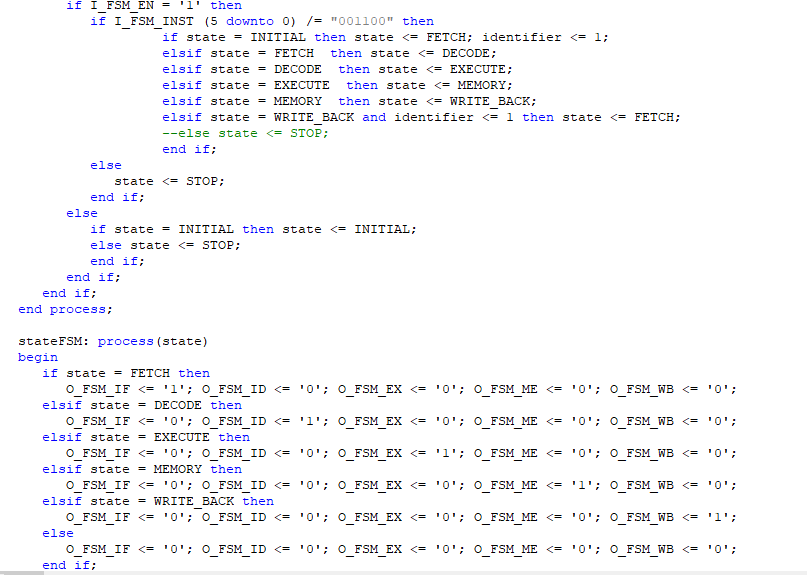


figure 2: added circuit to fix problem of overwriting registers without necessity to do so.

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.



--------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.

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

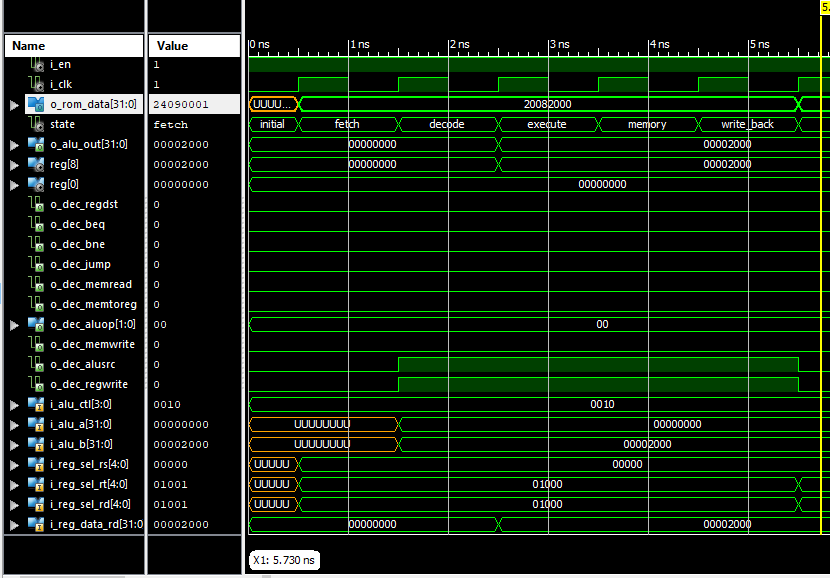


figure 3: addi $8, $0, 0x00002000 (0x20082000) instruction.

|  |  |  |
| --- | --- | --- |
| **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):

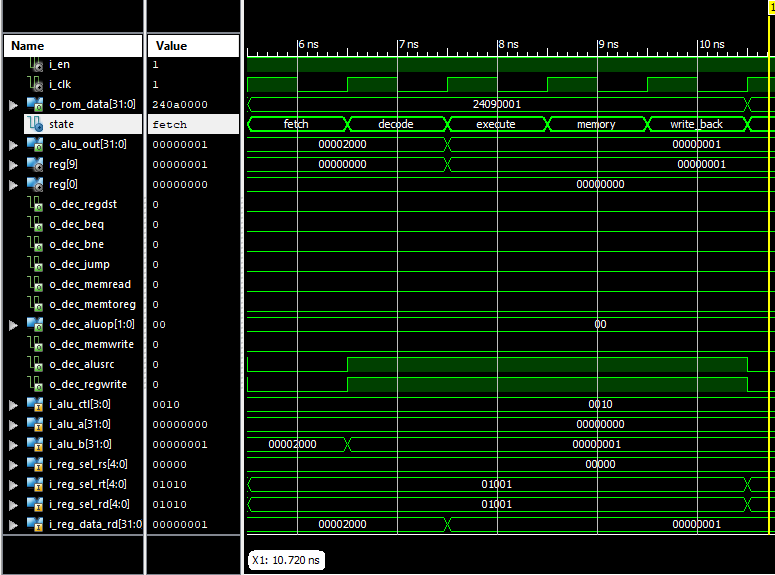


Figure 4: addiu $9, $0, 0x00000001 (0x24090001) instruction.

|  |  |  |
| --- | --- | --- |
| **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):

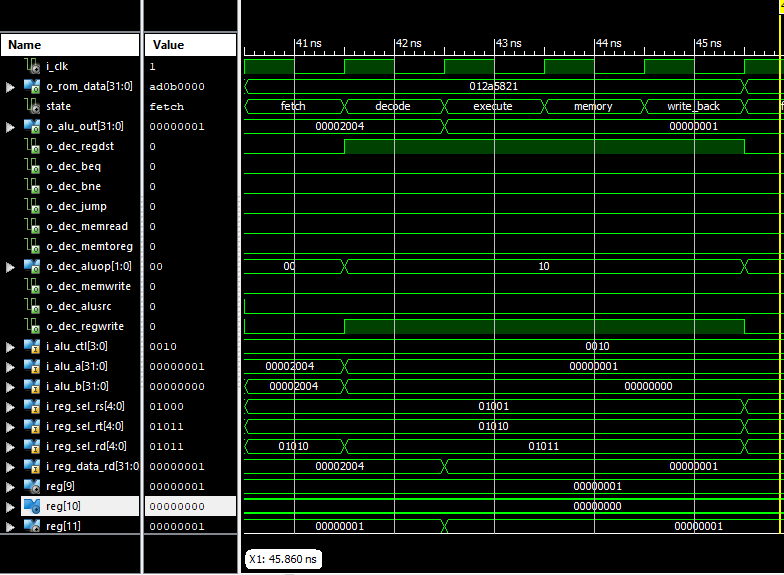
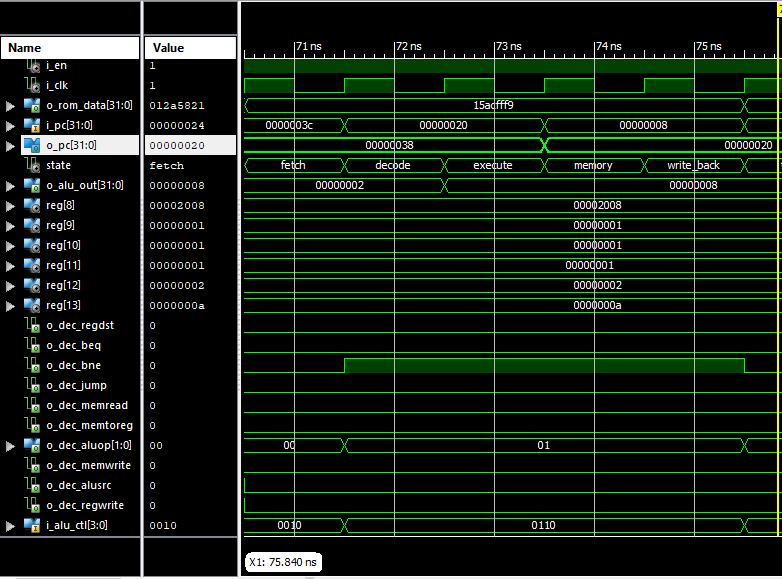


Figure 6: addu $11, $9, $10 (0x012a5821) instruction.

|  |  |  |
| --- | --- | --- |
| **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):

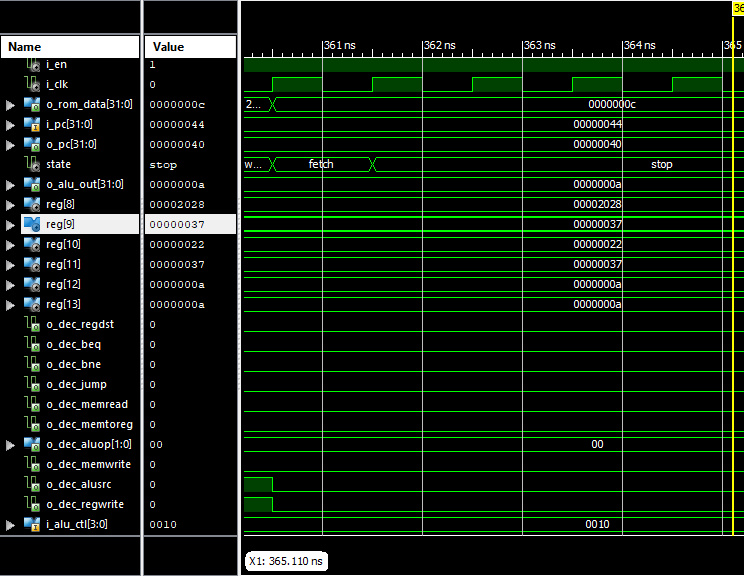


Figure 8: Syscall (0x0000000c) instruction.

|  |  |  |
| --- | --- | --- |
| **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 |

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

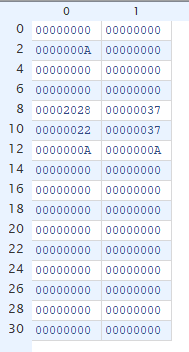


figure 9: The content of register file (in hexadecimal format) after finishing execution of the program.

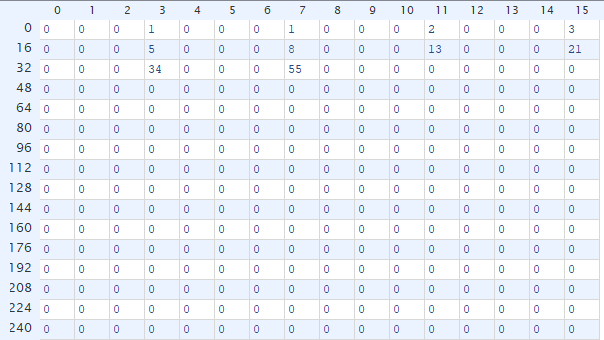


Figure 10: the content of the RAM array (in unsigned decimal format) after finishing execution of the program.

* The design generates the targeted results🡪 first 10 Fibonacci numbers.