Lab 5 - Single Cycle Processor

Gina DiCarlo

Eun Il Kim

Kenny Tan

CSC 34300

May 26th, 2017

1. Describe each of the processor components: include but not limited to the following:
   - Input/output signal.
   - Implementation details.
   - How it interacts with the rest of the system.

Processor: The input is the clock and the enable from the processor. The output of the processor is the ALU output. This test bench module runs the overhead of the program (schematic). It will initiate by assigning values within the FSM.

FSM: Takes the clock and enable input from the processor. The output is the following: Instruction Fetch → ROM, Instruction Decode → Ctl, Execution → Alu, Memory → RAM, and Write Back → PC and write data. Provides states to perform the 5 stages of instruction in a single cycle CPU.

ROM: Takes in a 32-bit address from the PC, then outputs the 32 bits instructions stored in the ROM arrays using the address as the index. It outputs the first 6-bits into the control component. Now depending on the instruction; R-type or I-type (J-type not implemented in this design), from a R-type the next 5-bits goes to the register as rs or read register 1, the 5 after that goes to rd or read register 2.

REG: Takes in the read addresses from the ROM then it accesses the data within the ROM using these addresses during the ID stage; both 5-bits named rs and rt respectively. However during the WB stage we input the values from the final mux32'b from either the RAM or ALU.

CTL: Takes the opcode from ROM (first 6 bits instruction). Depending on instruction; R-type or I-type (J-type is not implemented in this design) the control output will be different. For an R-type the RegDst and the RegWrite will both be equal to 1; and set everything else to 0. For an I-type (lw) the Alusrc, Memread, MemtoReg and Regwrite; gets set to 1, leaving all else 0. Additionally, for (sw) Alusrc, Memwrite, and RegDst gets initialized to 1, while the rest is set to 0. Lastly, for a branch instruction (BNE, BEQ) we set the control branch to 1 along with RegDst.

EXT: Takes a 16-bit input from the ROM instruction and extends it to 32-bits, the value which it outputs.

ALU_CTL: Takes in an aluop code of 2-bits based on the instruction; and, or, add, sub, etc... It then outputs a 4-bit value, which we called the ALU_CTR and is used by the ALU.

ALU: Takes a 32-bit number from the register "read data 1" and another 32-bit number from "read data 2" then the 4-bit input from ALU CTL tells the component weather it should perform an addition or subtraction via the instruction type. The output is 32-bit address.

RAM: Takes in the 32-bit ALU address and 32'b data from the register and can either read (lw) using the ALU address or write (sw) to address using the data from register depending on the control signals. Since we are only interested in 2'HEX (ie. 8'b) of possible saveable input

address we only take the last 8 bit, giving us 256 possible addresses. We use this data to either save or write back into the register.

ADD 1: The inputs are the current PC value of the program called “PC update” and outputs the next PC instruction address by adding 4.

ADD 2: Takes in the output of add 1 as the input and depending on the instruction set; R-type or I-type (J-type not implemented in this design) it will add the input with the shift address and output it through the multiplexer back to the next PC instruction.

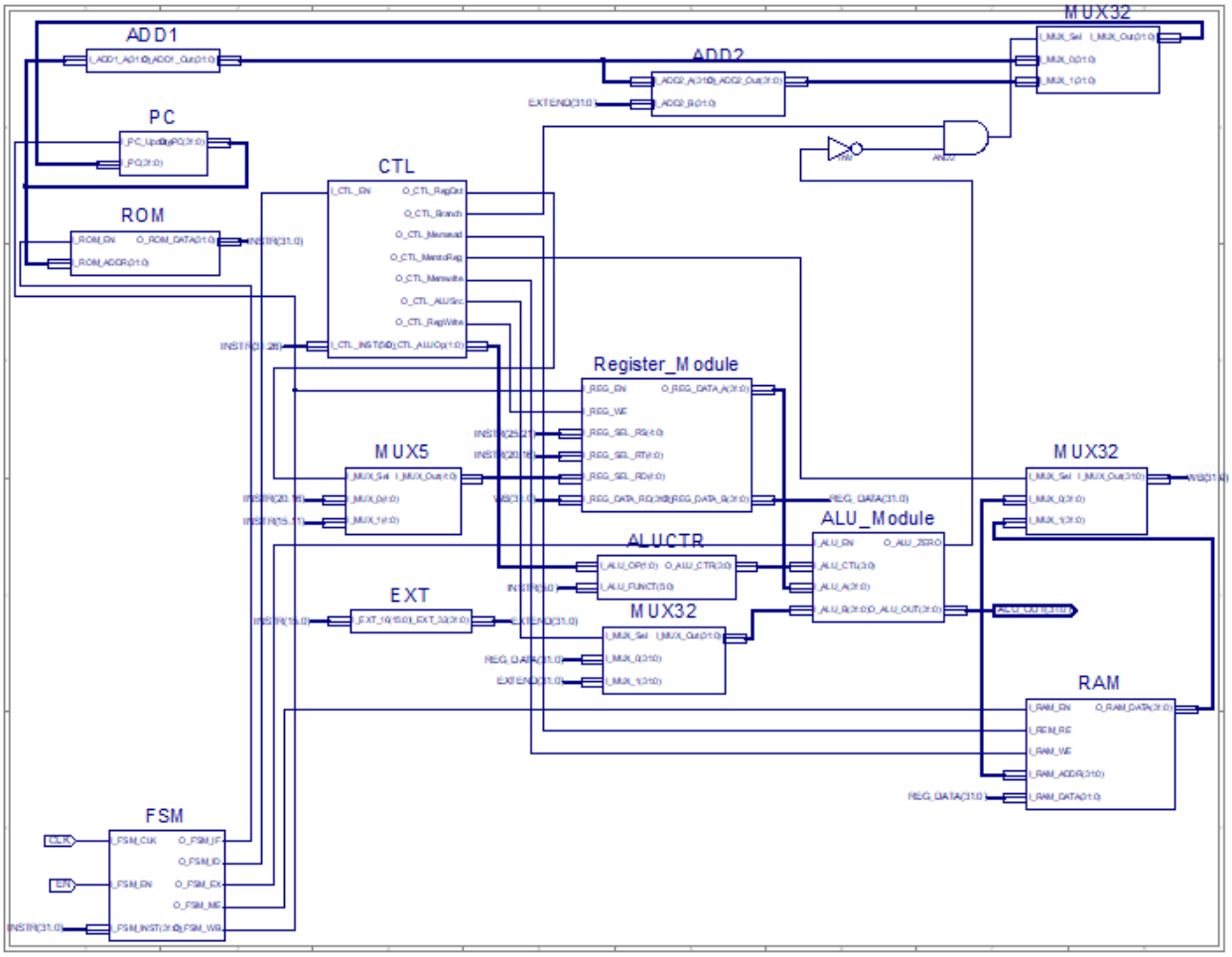
PC: Takes the initialized value of zero, it outputs to ROM and increments itself by four through add1, will use add2 depending or not it branches and outputs a new PC address into itself.

MUX 5-bit: Takes the output from any other processor component. Outputs either a 0 or 1. Acts as a Selector because different instruction address require different types of outputs. This will be used to decide which slice of instructruction to use in the register, depends on whether it’s an r or i instruction.

MUX 32-bit: Takes the output from any other processor component. Outputs either a 0 or 1. Acts as a Selector because different instruction address require different types of outputs. It will be used to decide to use the extended value or the value of the read_rt from register in the input of ALU. Then another one will be used to determine whether to pass the ALU value or the RAM value for the WB.

2. Include the top level design of your processor (if different from the provided example).

[It’s the same, but we’ll include it anyway]



3. Demonstrate that each of the five instructions can be correctly executed on the processor. Include the relevant signals. Properly zoom in/out the waveform so that the values of the signals could be clearly seen in your screenshots.

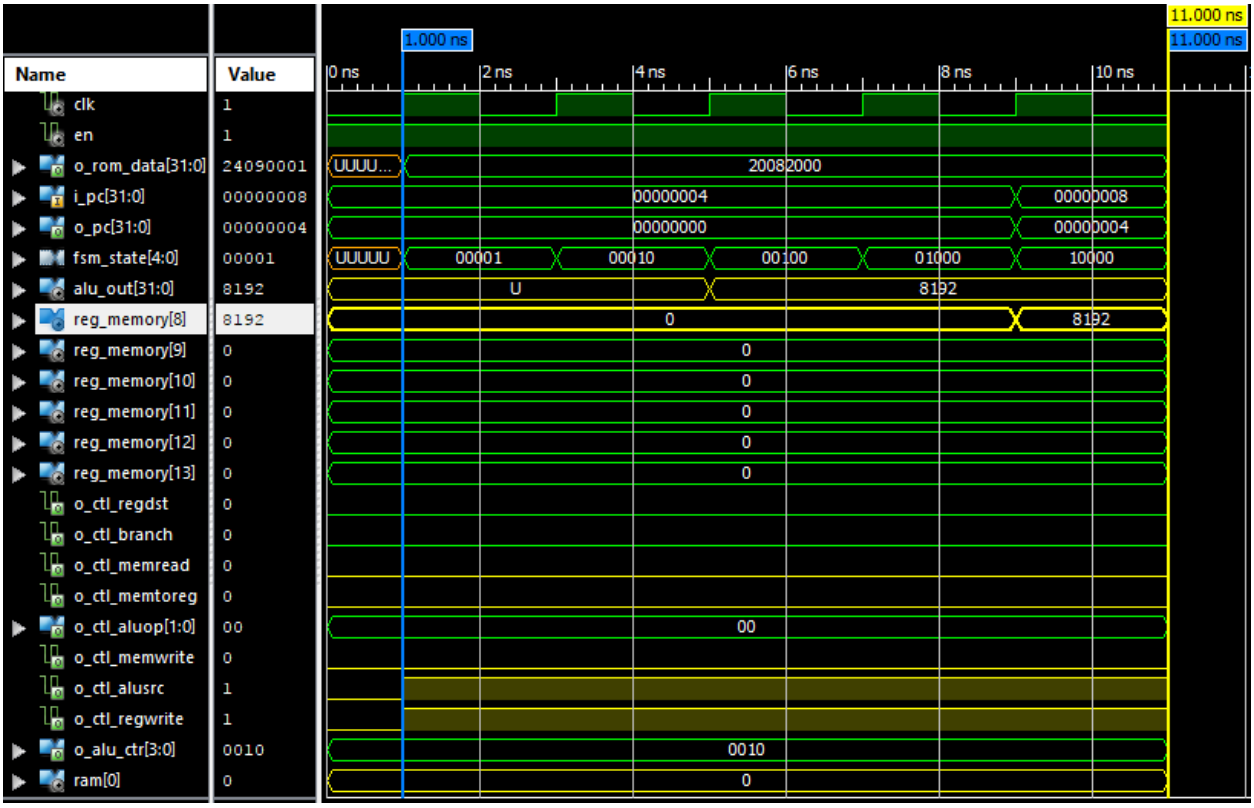
----------SIDE NOTE----------

The memory address is indeed 0x0000 2000, but we are using the integer (decimal) representation which will give us 8192. Similarly with the reg_memories where the output is in integer format.

Source: la $t0, memory; Basic: $8, $8, 0x00000004

The following waveform shows the execution of an I-type instruction. This instruction is encoded 0x24090001, as shown in the waveform named under “o_rom_data[31:0].” It performs an add immediately from the address from reg_memory[8] to the variable $t0 which is 0.

Signal	Value	Note
regdst	0	Not writing to any register
branch	0	Does not branch because this is an I-type
memtoread	0	Not reading to any register
memtoreg	0	Not loading from memory and writing to a register
aluop	0b00	And for ALU
memwrite	0	Not writing to the memory
alusrc	1	Second input is the sign extension
regwrite	1	Writes to register
alu_ctr	0b0010	Addition on ALU
reg_file[8]	8192	Value in first register
reg_file[]	0	Value in second register
alu_out	8192	Result of addition
o_pc	0x00000004	Current instruction address
i_pc	0x	Next instruction address
i_pc* (for branch)	0x00000008	No branch so this is the next instruction

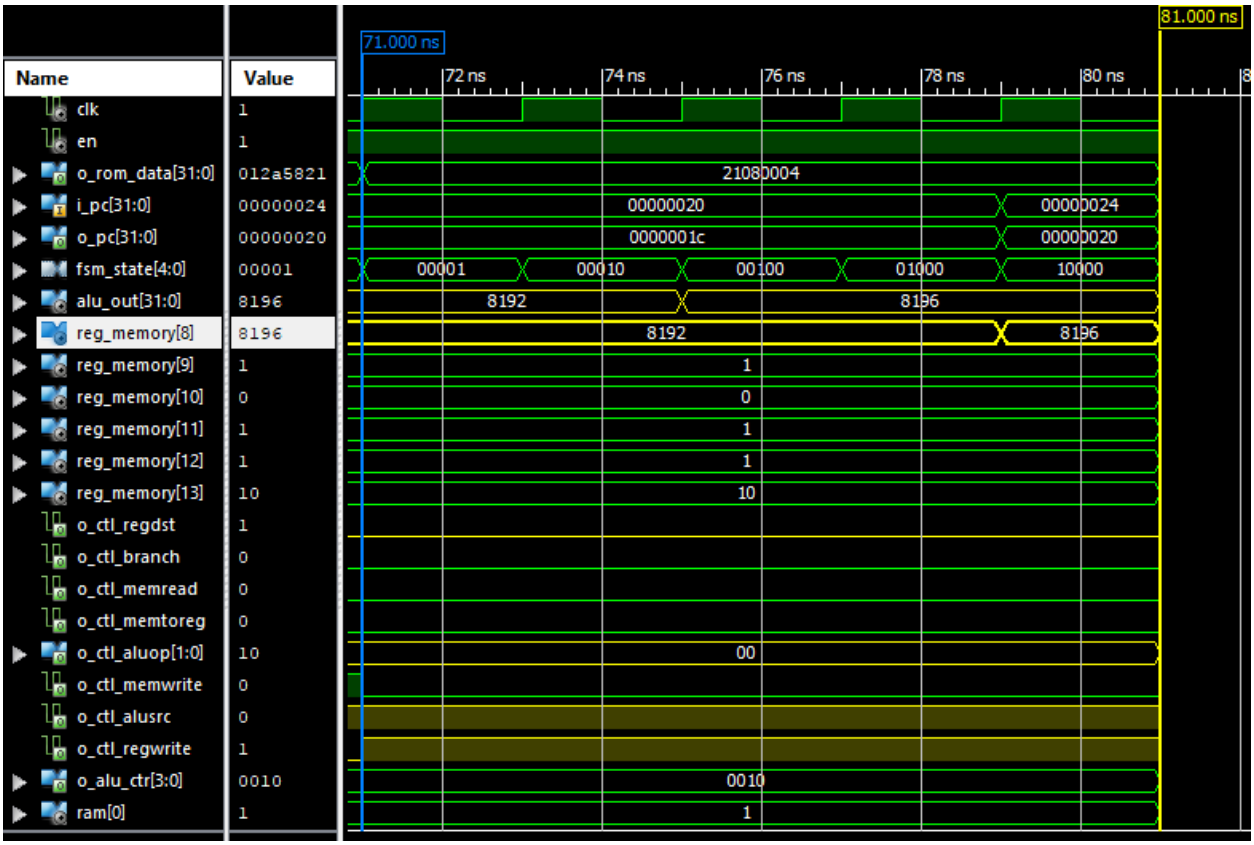


Source: li $t1, 1; Basic: addiu $9, $0, 0x00000001

The encoded instruction for below is 0x21080004 (o_rom_data). Addi takes its “immediate” part of its address, 0x00000001 (in decimal value 1) and adds it implicitly to 0, then stores this value in register 9 (reg_memory[9]). As shown below the value of register_memory[9] is indeed equal to 1.

Signal	Value	Note
regdst	1	Write to first register listed (register 9 for this example)
branch	0	Is not a branch instruction.
memtoread	0	Not reading from memory
memtoreg	0	Not loading from memory and writing to a register
aluop	0b10	Add on ALU.
memwrite	0	Not writing to the memory.
alusrc	1	Second input is the sign extension of instruction[15:0]
regwrite	1	Writing to register 9.

alu_ctr	0b0010	Add for ALU
reg_file[12]	1	Value in the first register.
reg_file[13]	10	Value in the second register.
alu_out	0x00002004	Result of the addition (converted from integer to hex).
o_pc	0x0000001C	Current instruction address.
i_pc	0x00000020	Next instruction address.

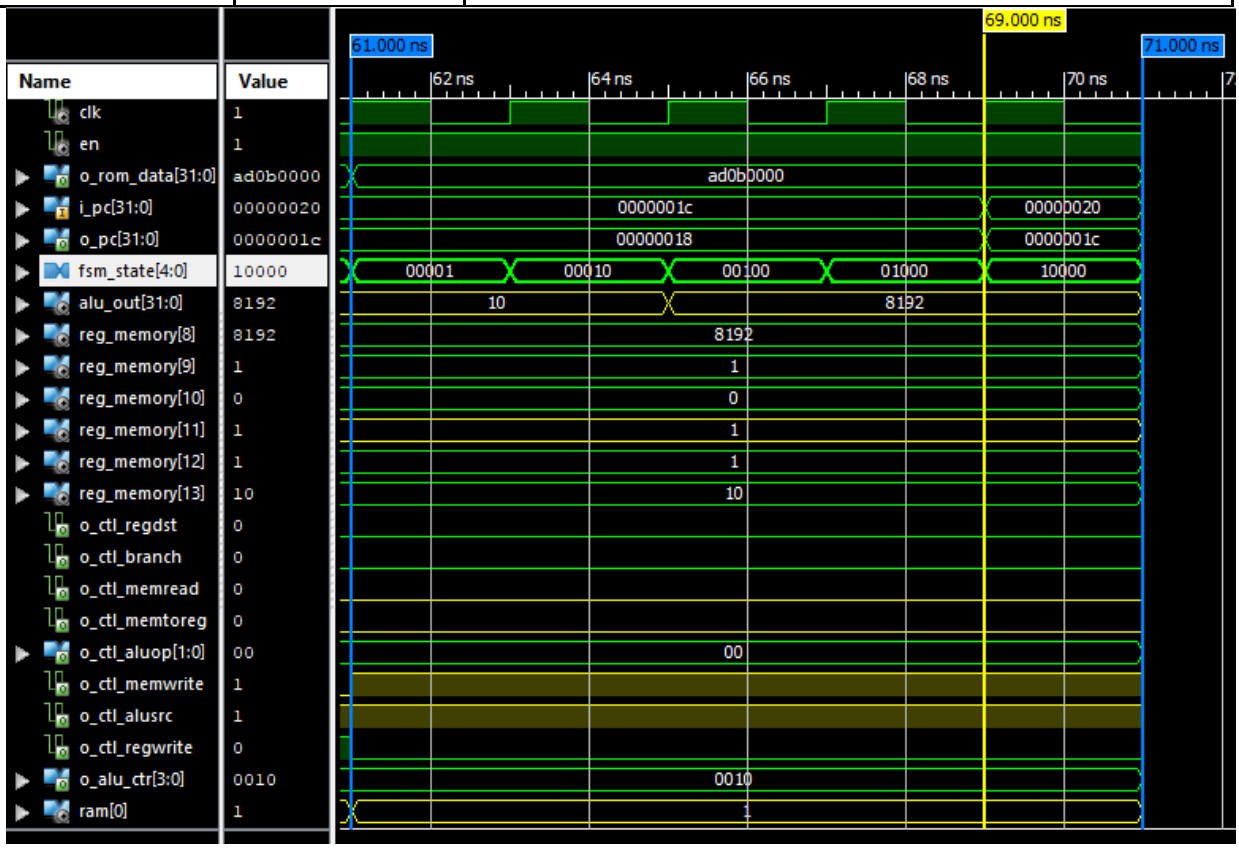


sw $11, 0($8)

The encoded instruction for below is 0xad0b0000 (o_rom_data), and it is for the I-type instruction store word. The value of F(1), equaling to 1 at this point, is being stored to the address that is in register 8, (reg_memory[8]).

Signal	**Value**	**Note**
regdst	0	Not writing to any register.
branch	0	Not a branch instruction.

memtoread	0	Not reading from memory.
memtoreg	0	Not loading from memory and writing to a register.
aluop	0b00	Add instruction for ALU.
memwrite	1	Writing to memory.
alusrc	1	Gets second input from sign extension.
regwrite	0	Does not write to a register.
alu_ctr	0b0010	Add instruction for ALU.
reg_file[12]	1	First input value for the register.
reg_file[13]	10	Second input value for the register.
alu_out	0x00002000	Output of the ALU (converted from integer to hex).
o_pc	0x00000018	Current instruction address.
i_pc	0x0000001C	Next instruction address.

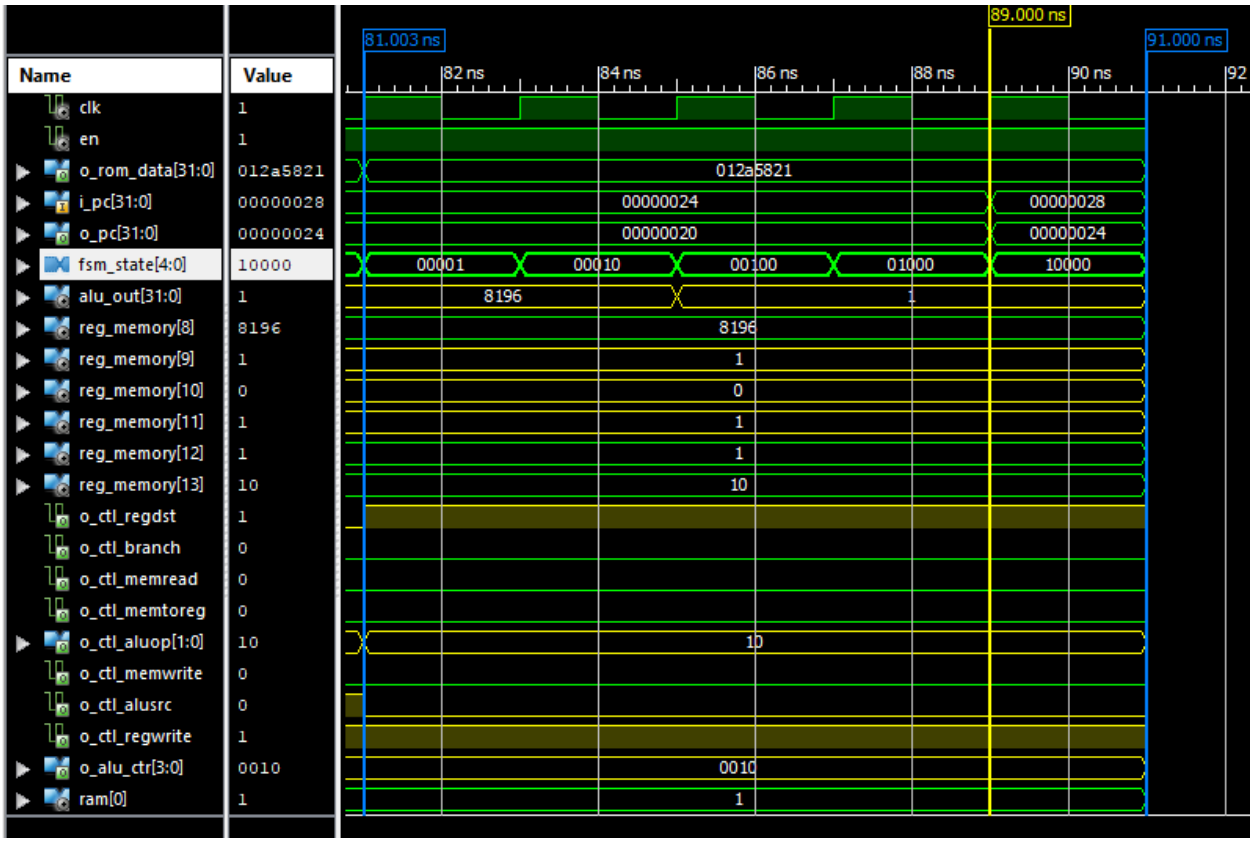


addu $t3, $t1, $t2

Addu is an R-format instruction, the encoded address for the specific instruction is 0x012a5821(o_rom_data).Below it takes the values from $t1 (register_memory[9]) and $t2

register_memory[10]), adds them as unsigned integers and then stores the value into register $t3, (register_memory11]). As you can see below the value of register 9 is 1, register 10 is 0, and register 11 is 1 (1+0 = 1).

Signal	**Value**	**Note**
regdst	1	Writing to a register.
branch	0	It is not a branch instruction.
memtoread	0	Not reading from the memory.
memtoreg	0	Not loading from memory and writing to a register.
aluop	0b10	Addition on ALU.
memwrite	0	Not writing to the memory.
alusrc	0	Second ALU input is from the register.
regwrite	1	Writing to a register.
alu_ctr	0b0010	Addition on ALU.
reg_file[11]	1	The value of the register to be saved.
reg_file[9]	1	The value in the first register.
reg_file[10]	0	The value in the second register.
alu_out	1	Result of the Addition.
o_pc	0x00000024	Current instruction address.
i_pc	0x00000020	Next instruction address.

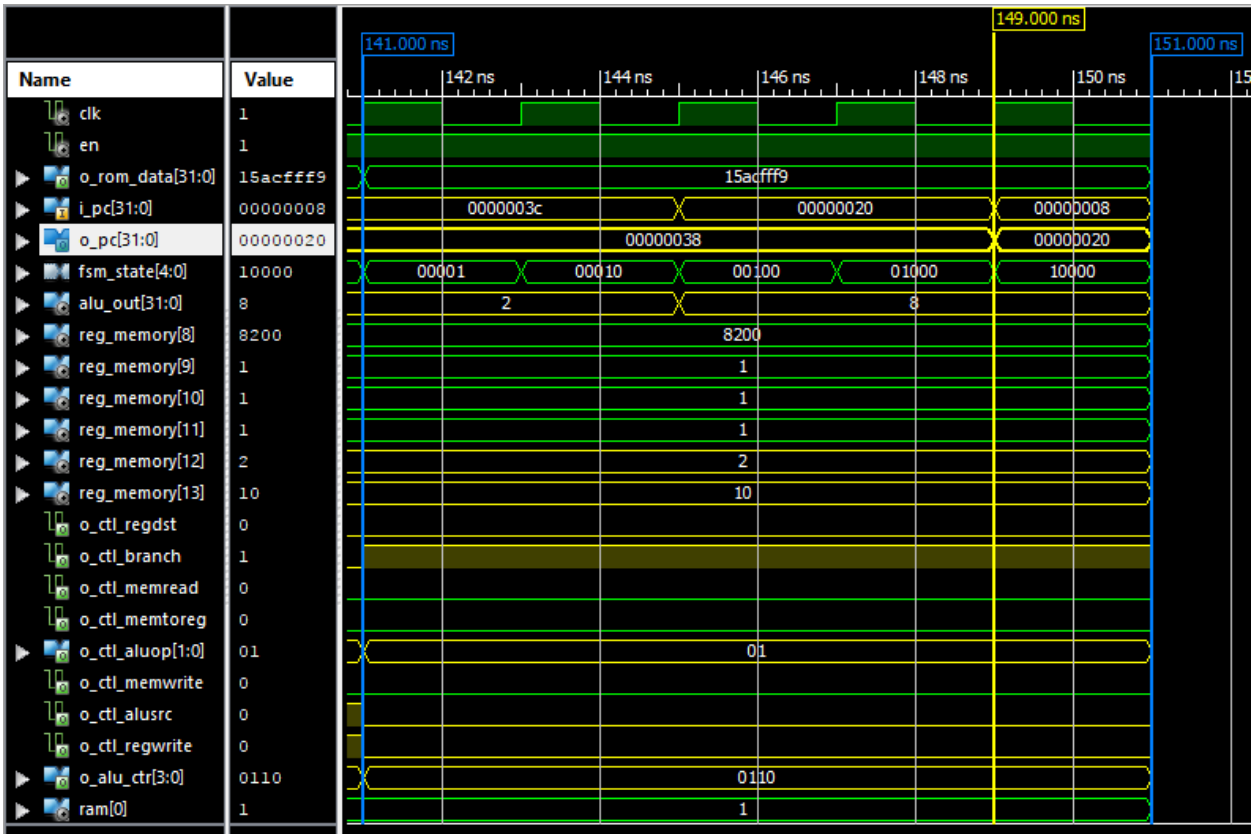


Bne $13, $12, loop

The branch not equal is an I-type instruction. Here it compares the value in register 12 (reg_memory[12]), which is a 2 and the value in register 13(reg_memory[13]), which is a 10, and branches to loop. The address for the BNE at this point is 0x00000038 (o_pc[31:0]), the following instruction is 0x00000020 (o_pc[31:0]), showing that it branched back to the address of loop.

Signal	Value	Note
regdst	0	Not writing to any register.
branch	1	It is a branch instruction.
memtoread	0	Not reading from the memory.
memtoreg	0	Not loading from memory and writing to a register.
aluop	0b01	Subtraction on ALU.
memwrite	0	Not writing to the memory.

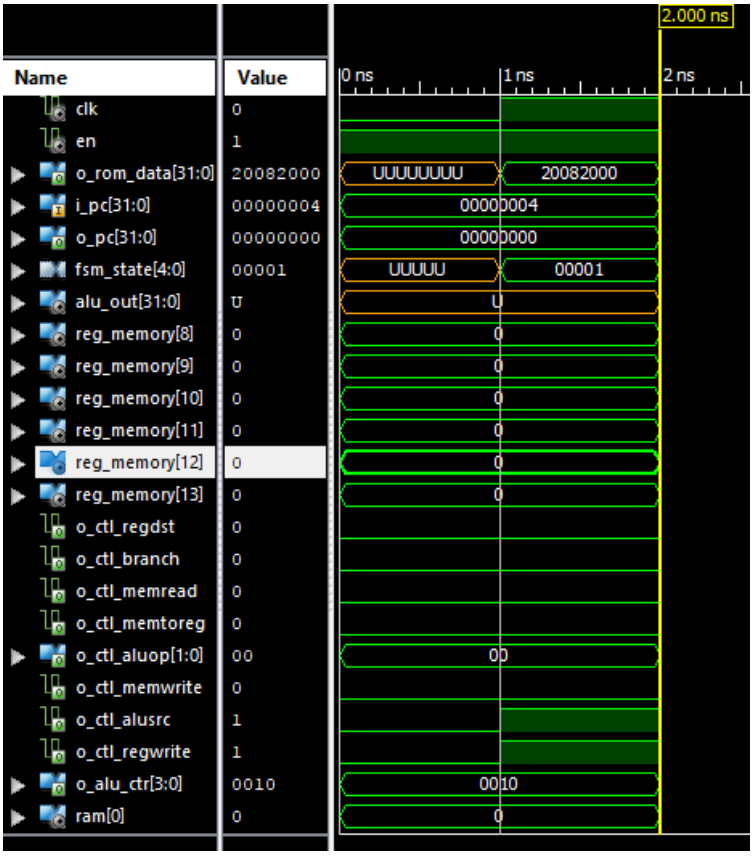
alusrc	0	Second ALU input is from the register.
regwrite	0	Not writing to a register.
alu_ctr	0b0110	Subtraction on ALU.
reg_file[13]	10	The value in the first register.
reg_file[12]	2	The value in the second register.
alu_out	3	Result of the subtraction.
o_pc	0x00000038	Current instruction address.
i_pc	0x0000003C	Next instruction address.
i_pc* (for branch)	0x00000020	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.



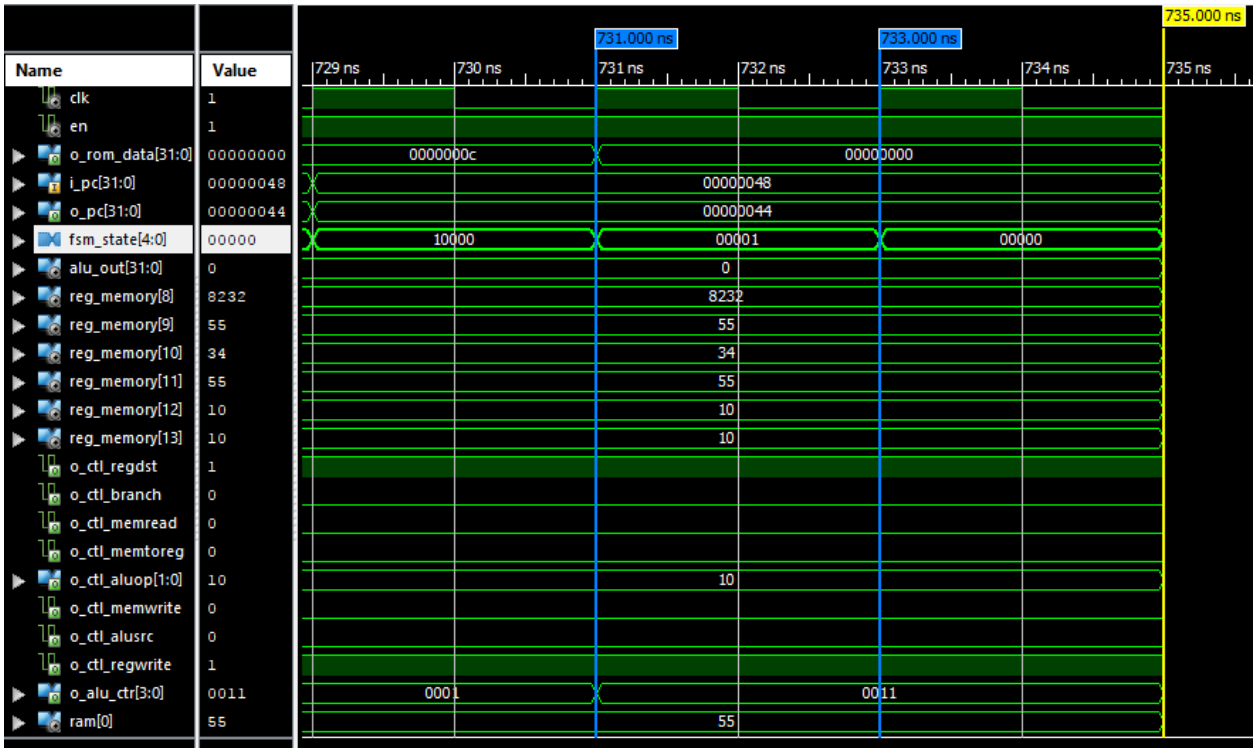
4 - 5. Use screenshots of the simulation waveform to show that the program can be started and stopped properly. Describe how you start and stop the processor. Prove the Fibonacci numbers

are correctly generated by showing the data memory contents when the processor finishes the MIPS program.

We initiated the program by force starting the pc value to 0x0000 0000. Inititally the design started with 0xUUUUUUUU. The problem with that former design was that the pc was getting the value of i_pc which had the value of 0xUUUUUUUU so it was always passing no value to the pc and gave us incorrect values for the rest of the sequence..



Although the program supposedly ends at 731 ns where the ROM outputs the instruction of 0x0000 0000. As the instruction (of 0x0000 0000) is only updated when the fsm is in IF stage at 731-733 ns, we need the instruction to be set at the IF stage. At the ID stage from 733-734 ns,



is where we would end the process, as there is no instruction to be read.

In addition to knowing that the end of the program is officially at 731 ns, we also know the sequence that the code is being iterated.

Values of interest:

Update $t3: reg_memory[11] = 55

Save to ram[$t0]: ram[0] = 55

Update for the next call:

Update $t0: reg_memory[8] = 8232

Update $t2: reg_memory[10] = 34

Update $t1: reg_memory[9] = 55

Update $t4: reg_memory[12] = 10

No changes:

$t5: reg_memory[13] = 10