ISTANBUL TECHNICAL UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

BLG 222E

COMPUTER ORGANIZATION PROJECT 2 REPORT

CRN	:	21334
LECTURER	:	Assoc. Prof. Dr. Gökhan İnce

GROUP MEMBERS:

150210928	:	Racha Badreddine
150220061	:	Melike Beşparmak

SPRING 2024

CONTENTS

Contents

1 INTRODUCTION [10 points]

In this project, we designed and implemented a CPU system based on the previous project, where an ALU system was implemented. The CPU system can execute 34 operations and uses a memory unit containing 256 words of 8 bits. We first analyzed the provided instructions according to the given formats, then wrote all the RTL (Register Transfer Level) and control signals needed to implement the project using the Verilog HDL (Hardware Description Language) in the Vivado environment.

1.1 Task Distribution

While working on this project, we made sure that we thoroughly understood and could implement the provided system. To achieve this, we held regular meetings whenever necessary to collaborate on development and overcome any obstacles we encountered. In addition to our meetings, each of us individually drafted the Register Transfer Language (RTL), compared our results, and identified any errors.

2 MATERIALS AND METHODS [40 points]

2.1 Preliminary

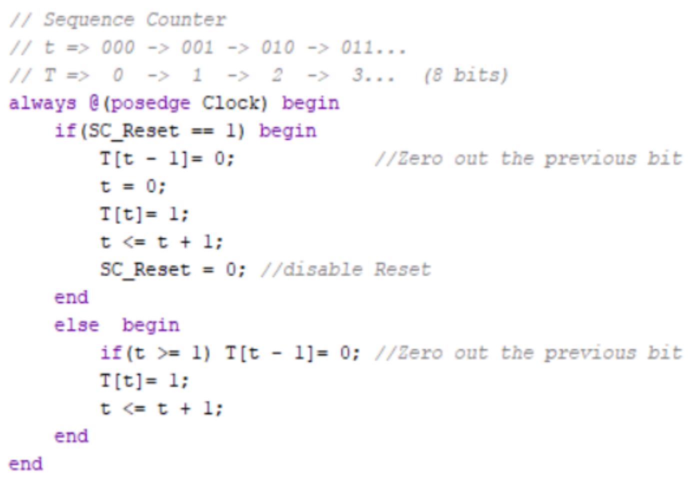
Our primary focus, given that every module from the previous project is ready, was to understand the instructions and create a layout accordingly. We designed the CPU module around clock signals, ensuring that operations dictated by the opcode are executed on every clock cycle, similar to control signals. To keep track of the clock cycles, we implemented a sequence counter. Additionally, we aimed for generalization wherever possible. In the basic computer model, the operation cycle typically includes three steps: fetch, decode, and execute. While the first two steps are common across all operations, the execution depends on the operation. We followed a similar approach in our design, as explained in the following sections.
The code structure consists of an always block for each Clock cycle and an initial begin block. In the initial begin block, all registers are cleared and initialized to 0, except for the SP register, which starts from the bottom of the memory and is initialized to 255.
The first always block encapsulates the first part of the fetch operation, including T_0, while the second block includes T_1. The other blocks handle the remaining timing signals. In each block we opened cases according to the opcode, inside each case we generated the necessary control signals to perform the operation. Every time the execution ends we reset our sequence counter so it starts fetching and executing the next instruction.

2.1.1 Instruction Format

To initiate an execution cycle, the instruction needs to be loaded into the instruction register (IR). Given that instructions are 16 bits while memory words are 8 bits, the fetch process requires two clock cycles. Following fetching, the instruction's format is determined based on the opcode. Explicit decoding is unnecessary, as Verilog's "cases" feature is enough for this purpose.
We categorized operations into two groups: format 1 and format 2. The first group encompasses instructions such as BRA, BNE, BEQ, POP, PSH, MOVH, LDR, STR, BX, BL, LDRIM, and STRIM, primarily involving branching, stack operations, and memory access. The second group involves instructions needing the arithmetic-logic unit (ALU) without memory references. These operations can be further subdivided based on the need for a second source register.
We considered the worst-case scenario for each group and recorded the steps for each clock cycle, ensuring consistency across instructions within each group except for the function selector (FunSel) of the ALU. Subsequently, we documented each step in the register transfer language (RTL), identified the necessary control signals, and implemented the designed system.

2.1.2 Sequence Counter

Since we are working with synchronized circuits, tracking the clock signal is crucial. We implemented a sequence counter as in Figure 1.



```
// Sequence Counter
// t => 000 -> 001 -> 010 -> 011...
// T =>  0  ->  1  ->  2  ->  3...   (8 bits)
always @(posedge Clock) begin
    if(SC_Reset == 1) begin
        T[t - 1]= 0;            //Zero out the previous bit
        t = 0;
        T[t]= 1;
        t <= t + 1;
        SC_Reset = 0; //disable Reset
    end
    else  begin
        if(t >= 1) T[t - 1]= 0; //Zero out the previous bit
        T[t]= 1;
        t <= t + 1;
    end
end
```

Figure 1: Sequence Counter

There are two registers, one is uppercase T which is 8 bits, and a lowercase t which is 3 bits. The first register T is used as an array and the second t is like a pointer indicating the indices of T. During operation, when the sequence counter (SC) is not reset, the bit

T[t] is set to 1, while the previous index is set to 0. So that the counter will work as:
00000001 $\rightarrow$ 00000010 $\rightarrow$ 00000100 in binary
$T[0] \rightarrow T[1] \rightarrow T[2]$ in array representation
1 $\rightarrow$ 2 $\rightarrow$ 4 in decimal

The reset block initializes the counter at T[0] and then disables the reset signal to prevent unintended resets during operation. As long as the reset signal remains disabled, the counter continues counting normally, preserving its current value without resetting.

2.1.3 Fetch

Fetch involves a two-clock cycle process. First, the least significant bits (LSB) are loaded into the Program Counter (PC), followed by the loading of the most significant bits (MSB).
$T_0 : IR[7:0] \leftarrow M[PC], PC \leftarrow PC + 1$
$T_1 : IR[15:8] \leftarrow M[PC]$

2.2 Operations

RTL:

a-Instructions with memory reference:

All the RTLs we used to implement the instructions are shown in the table below. We grouped some instructions to be explained in detail later:

OPCODE	Symbol	RTL	Group
0x00	BRA	D_0T_2: $S_1 \leftarrow$ Value(IR[7:0]) D_0T_3: $S_2 \leftarrow$ PC D_0T_4: PC $\leftarrow S_1 + S_2$, SC$\leftarrow$0	1
0x01	BNE	D_1T_2Z': $S_1 \leftarrow$ Value(IR[7:0]) D_1T_3Z': $S_2 \leftarrow$ PC D_1T_4Z': PC $\leftarrow S_1 + S_2$, SC$\leftarrow$0 else: D_1T_2Z: SC$\leftarrow$ 0	1
0x02	BEQ	D_2T_2Z: $S_1 \leftarrow$ Value(IR[7:0]) D_2T_3Z: $S_2 \leftarrow$ PC D_2T_4Z: PC $\leftarrow S_1 + S_2$, SC$\leftarrow$0 else: D_2T_2Z': SC$\leftarrow$ 0	1

0x03	POP	D_3T_2: SP $\leftarrow$ SP + 1 D_3T_3: R_x[15:8] $\leftarrow$ M[SP],SP $\leftarrow$ SP + 1 D_3T_4: R_x[7:0] $\leftarrow$ M[SP], SC $\leftarrow$ 0	2
0x04	PSH	D_4T_2: M[SP]$\leftarrow$ R_x[7:0], SP $\leftarrow$ SP - 1 D_4T_3: M[SP]$\leftarrow$ R_x[15:8], SP $\leftarrow$ SP - 1, SC$\leftarrow$ 0	2
0x11 | 0x14	MOVH | MOVL	$D_{17}T_2$:R_x[15:8] $\leftarrow$ IR[7:0](IMMEDIATE),SC$\leftarrow$ 0 $D_{20}T_2$:R_x[7:0] $\leftarrow$ IR[7:0](IMMEDIATE), SC$\leftarrow$ 0	3
0x12	LDR	$D_{18}T_2$: $R_x[7:0]$ $\leftarrow$ M[AR], AR $\leftarrow$ AR + 1 $D_{18}T_3$: $R_x[15:8]$ $\leftarrow$ M[AR], SC $\leftarrow$ 0	4
0x13	STR	$D_{19}T_2$: M[AR] $\leftarrow$ $R_x[7:0]$, AR $\leftarrow$ AR + 1 $D_{19}T_3$: M[AR] $\leftarrow$ $R_x[15:8]$, SC $\leftarrow$ 0	4
0x1E	BX	$D_{30}T_2$: S_1 $\leftarrow$ PC, PC $\leftarrow$ R_x $D_{30}T_3$: M[SP]$\leftarrow$ S_1[7:0], SP $\leftarrow$ SP - 1 $D_{30}T_4$: M[SP]$\leftarrow$ S_1[15:8],SP $\leftarrow$ SP - 1, SC $\leftarrow$ 0	5
0x1F	BL	$D_{31}T_2$: SP $\leftarrow$ SP + 1 $D_{31}T_3$: PC[15:8]$\leftarrow$ M[SP] $D_{31}T_4$: SP $\leftarrow$ SP + 1 $D_{31}T_5$: PC[7:0]$\leftarrow$ M[SP], SC $\leftarrow$ 0	5
0x20	LDRIM	$D_{32}T_2$: AR $\leftarrow$ IR[7:0] (value) $D_{32}T_3$: R_x[7:0] $\leftarrow$ M[AR], AR $\leftarrow$ AR + 1 $D_{32}T_4$: R_x[15:8] $\leftarrow$ M[AR], SC $\leftarrow$ 0	6
0x21	STRIM	$D_{33}T_2$: S_1 $\leftarrow$ AR, AR $\leftarrow$ IR[7:0] (Address of OFFSET) $D_{33}T_3$: S_2 $\leftarrow$ M[AR] $D_{33}T_4$: AR $\leftarrow$ S_1 + S_2 $D_{33}T_5$: M[AR] $\leftarrow$ R_x[7:0], AR $\leftarrow$ AR + 1 $D_{33}T_6$: M[AR] $\leftarrow$ R_x[15:8], SC $\leftarrow$ 0	6

Table 1: First Format RTL

In all the RTLs shown in the table above, we have written all the necessary control signals to execute the specific micro-operations at each Clock Signal. Since all instructions in the table deal with 16-bit words, every write or read operation to memory requires two clock signals to transfer the word. In the upcoming explanation, when we refer to the first clock signal, we indicate the first clock signal in the execution process (T2).

Group 1:

This group includes the branch instructions BRA, BNE, and BEQ. All these instructions cause the Program Counter (PC) to branch. The value representing the address present in the least 8 bits of the Instruction Register (IR) is used. Both the PC and the values are transferred to the Register File (RF) since both Arithmetic Logic Unit (ALU) inputs are taken from the RF. After that, the addition operation is performed without updating the flags, and the new value is transferred back to the PC, causing the program to jump to this address to continue execution. The difference between these instructions is that BRA is performed directly. However, BNE requires the Z flag to be 0 to perform the branching, while BEQ requires Z to be 1. In these branch instructions, the return value is not saved. All these instructions are executed in 3 CLock Signals after fetching.

Group 2:

This group contains operations involving the Stack memory, encompassing both POP and PUSH instructions. In our design, the Stack occupies the last portion of the memory, hence our Stack Pointer (SP) is initially set to 255, representing the last address of the memory. Our stack design relies on the SP always pointing to an available (empty) address.

For the PUSH instruction, we first write the Least Significant Bits (LSB) of the word in register R_x to the empty address, simultaneously, the SP is decreased (as it starts from the bottom). Subsequently, we write the most significant bits (MSB) and decrement the SP again to keep it pointing to the empty address. The PUSH requires 2 Clock Signals to be executed.

Conversely, for the POP instruction, we begin by incrementing the SP to locate the MSB of the word. Then, in the next CLK signal, we write the MSB to the register as they are encountered first with another increment of the SP. Finally, we write the LSB to the register, and the SP will automatically point to the empty address. The POP is executed in 3 Clock Signals.

Group 3:

This group includes both MOVH and MOVL instructions. Both instructions are executed in 1 clock signal since they just require transferring the IMMEDIATE (8 LSB of the IR) either to MSB or LSB to the Register specified in the RSEL bits.

Group 4:

LDR and STR are part of this group, standing for Load Register and Store Register, respectively. Both instructions require 2 Clock Signals for execution. In both cases, the address stored in the Address Register (AR) is utilized. For LDR, we load the word from memory into the Register specified by the RSEL bits, while for STR, we store the value from the register in the memory location specified by AR.

Group 5:

This group consists of BX and BL instructions, which are similar to PUSH and POP operations. The BX instruction aims to branch the PC to the address specified in Rx

(RSEL) and store the return address in the stack. Initially, we transfer the PC to one of the Scratch registers, and simultaneously, the new address from Rx is sent to the PC. Subsequently, in the remaining clock signals, we push the return value present in the Scratch Register following the same logic explained earlier.

The BL operation essentially involves returning the address back to the PC by performing a POP operation from the stack to the PC. The same logic applies as explained earlier.

Group 6:

LDRIM and STRIM are the instructions within this group, somewhat similar to LDR and STR.

For LDRIM, we begin by reading the address to be read from (IR[7:0]) and storing it in the Address Register (AR). A comment that needs to be mentioned here is that since the value is only 8 bits, the new address stored in AR will have 0s in its MSB, which means accessing only a small part (the first one) of the memory. Subsequently, we load the 16-bit word into the register, following the same logic as explained before.

In the case of STRIM, we first transfer the AR value to one of the scratch registers since we need to perform an addition operation. Then, we read the OFFSET from memory using the address specified in the LSB of the IR, and transfer it to another scratch register. Next, the new address with the OFFSET is stored in the AR, followed by the STR operation, employing the same logic as explained previously.

b-Instructions without memory reference:

First, enabling the destination register (DSTREG) and choosing the output for the SREG from ARF and RF has been done based on the following flowchart:

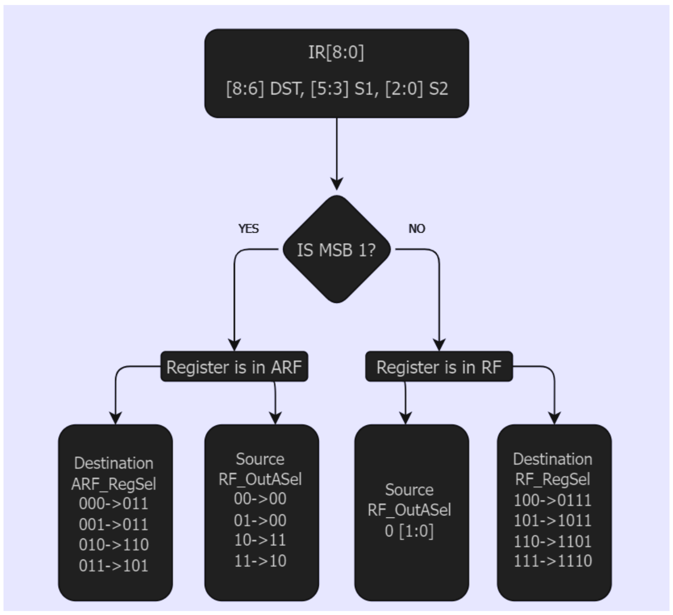

IR[8:0]
[8:6] DST, [5:3] S1, [2:0] S2
IS MSB 1?
YES
NO
Register is in ARF
Register is in RF
Destination ARF_RegSel 000->011 001->011 010->110 011->101
Source RF_OutASel 00->00 01->00 10->11 11->10
Source RF_OutASel 0 [1:0]
Destination RF_RegSel 100->0111 101->1011 110->1101 111->1110

Figure 2: Flowchart for DSTREG and SREG

The most significant bit of those 3 bits specifies whether the register is in ARF or RF. Then, according to the bits we either set the RegSel if it is DSTREG or output if it is SREG.

The table of RTL is shown below:

OPCODE	Symbol	RTL	Group
0x05|0x06	INC|DEC	$(D_5 \| D_6)T_2$: DSTREG $\leftarrow$ SREG1 D_5T_3: DSTREG $\leftarrow$ DSTREG + 1, SC$\leftarrow$ 0 D_6T_3: DSTREG $\leftarrow$ DSTREG - 1, SC$\leftarrow$ 0	7
0x07|0x08 0x09|0x0A 0x0B|0x0E 0x18	LSL|LSR ASR|CSL CSR|NOT MOVS	$(D_7 \| D_8 \| D_9 \| D_{10} \| D_{11} \| D_{14} \| D_{24})T_2$ IR[5]$'$: $S_1 \leftarrow$ SREG1 T_3: ALU.WF = IR[9] D_7T_3: DSTREG $\leftarrow$ LSL SREG1(OR S1), SC$\leftarrow$ 0 D_8T_3: DSTREG $\leftarrow$ LSR SREG1(OR S1), SC$\leftarrow$ 0 D_9T_3: DSTREG $\leftarrow$ ASR SREG1(OR S1), SC$\leftarrow$ 0 $D_{10}T_3$: DSTREG $\leftarrow$ CSL SREG1(OR S1), SC$\leftarrow$ 0 $D_{11}T_3$: DSTREG $\leftarrow$ CSR SREG1(OR S1), SC$\leftarrow$ 0 $D_{14}T_3$: DSTREG $\leftarrow$ NOT SREG1(OR S1), SC$\leftarrow$ 0 $D_{24}T_3$: DSTREG $\leftarrow$ SREG1(OR S1), SC$\leftarrow$ 0	8
0x0C|0x0D 0x0F|0x10 0x15|0x16 0x17|0x19 0x1A|0x1B 0x1C|0x1D	AND|ORR XOR|NAND ADD|ADC SUB|ADDS SUBS|ANDS ORRS|XORS	$(D_{12} \| D_{13} \| D_{15} \| D_{16} \| D_{21} \| D_{22} \| D_{23} \| D_{25} \| D_{26} \| D_{27} \| D_{28} \| D_{29})T_2$IR[5]$'$: $S_1 \leftarrow$ SREG1 $(D_{12} \| D_{13} \| D_{15} \| D_{16} \| D_{21} \| D_{22} \| D_{23} \| D_{25} \| D_{26} \| D_{27} \| D_{28} \| D_{29})T_3$IR[3]$'$: $S_2 \leftarrow$ SREG2 T_4: ALU.WF = IR[9] $(D_{12} \| D_{27})T_4$: DSTREG $\leftarrow$ (R_x or S1) AND (R_x or S2) $(D_{13} \| D_{28})T_4$: DSTREG $\leftarrow$ (R_x or S1) OR (R_x or S2) $(D_{15} \| D_{29})T_4$: DSTREG $\leftarrow$ (R_x or S1) XOR (R_x or S2) $D_{16}T_4$: DSTREG $\leftarrow$ (R_x or S1) NAND (R_x or S2) $(D_{21} \| D_{25})T_4$: DSTREG $\leftarrow$ (R_x or S1) + R_x or S2) $D_{22}T_4$: DSTREG $\leftarrow$ (R_x or S1) + (R_x or S2) + C $(D_{23} \| D_{26})T_4$: DSTREG $\leftarrow$ (R_x or S1) - R_x or S2)	9

Table 2: Second Format RTL

As previously explained, for this format of instructions, we aimed to generalize the microoperations to be executed. This led to the categorization into three main groups:

The first group requires the function to be executed in the register.

The second group involves operations performed on a single source register (SREG), executing various operations on it.

Lastly, the third group conducts operations on two source registers.

In the upcoming explanation, when we refer to the first clock signal, we are indicating the first clock signal in the execution process (T2).

Group 7:

This group includes increment and decrement instructions, which require 2 Clock Signals to be executed in a general way. The first Clock signal is used to transfer the word from SREG to DSTREG. According to the flow chart (*figure 2*) we provided the necessary control signals. Subsequently, during the second Clock signal, the increment or decrement operation is performed in the DSTREG. According to our understanding of the format, we assigned the S bit to the ALU.WF that enables the flags to be changed. However, if the S bit is 1, the flags may not always be changed because if the source register is in ARF, then transferring it to the destination register will not pass through the ALU.

Group 8:

This group consists of operations executed on a single source register via the ALU. It includes LSL, LSR, ASR, CSL, CSR, NOT, and MOVS instructions. Given that the only distinction among these instructions is the operation performed by the ALU (determined by the FunSel), we grouped them together even in the code.

During the first clock signal, the source register is transferred to the RF if it resides in the ARF since the ALU only accepts inputs from it. Subsequently, during the second clock signal, the operation is executed in the ALU based on the opcode, and the result is transferred to the DSTREG. The S bit is assigned to the ALU.WF to change the flags if necessary.

Group 9:

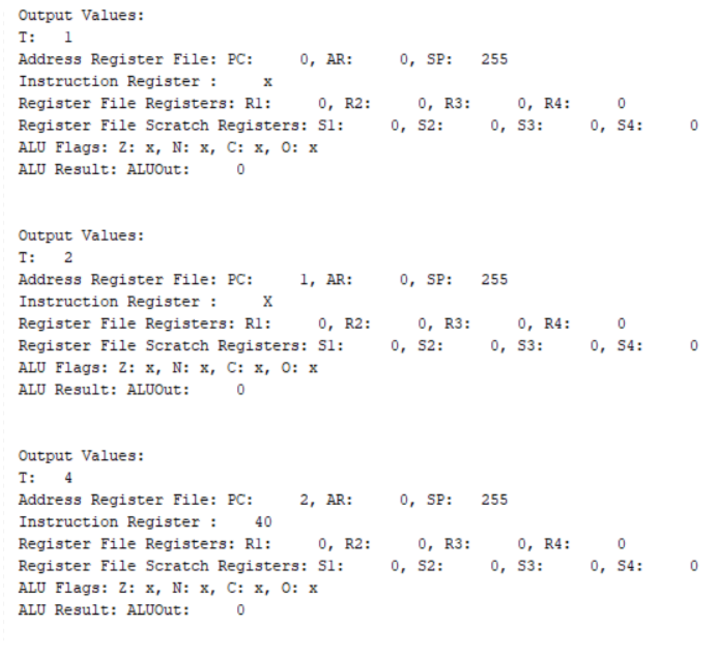
This group consists of operations executed on two source registers through the ALU. It includes AND, ORR, XOR, NAND, ADD, ADC, SUB, ADDS, SUBS, ANDS, ORRS and XORS.

For all of these instructions, the first Clock signal is used to transfer SREG1 to one of the scratch registers if it resides in ARF, and similarly, the second Clock signal is used to transfer SREG2 to another scratch register if it also resides in ARF.

During T_4, the operation in the ALU is selected based on the opcode, and the result is transferred to the DSTREG.

The S bit is transmitted to the ALU.WF to enable the flags to be updated when necessary. According to our understanding, in the ADD operation for instance, the S bit will always be 0, while in the ADDS operation, it will be 1, allowing the flags to change. This is why we grouped these two instructions and executed them together, similarly for other pairs such as ORR and ORRS.

3 RESULTS [15 points]

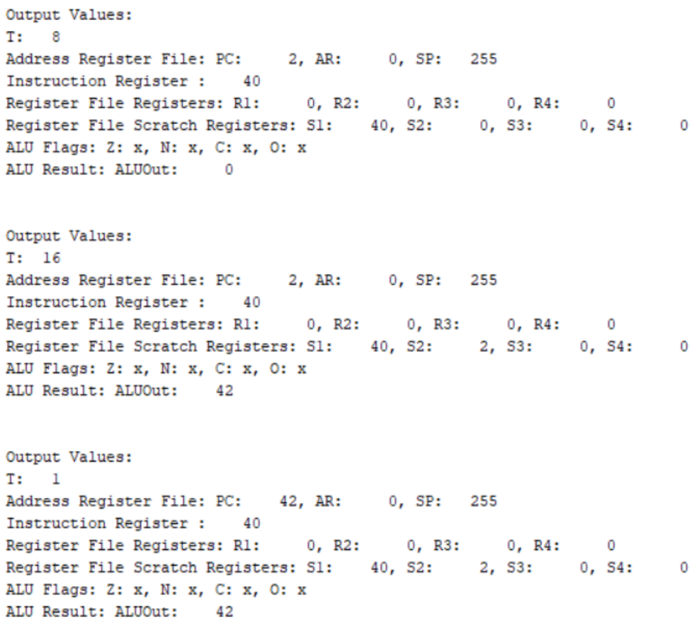


```
Output Values:
T:   1
Address Register File: PC:     0, AR:     0, SP:   255
Instruction Register :     x
Register File Registers: R1:     0, R2:     0, R3:     0, R4:     0
Register File Scratch Registers: S1:     0, S2:     0, S3:     0, S4:     0
ALU Flags: Z: x, N: x, C: x, O: x
ALU Result: ALUOut:     0

Output Values:
T:   2
Address Register File: PC:     1, AR:     0, SP:   255
Instruction Register :     X
Register File Registers: R1:     0, R2:     0, R3:     0, R4:     0
Register File Scratch Registers: S1:     0, S2:     0, S3:     0, S4:     0
ALU Flags: Z: x, N: x, C: x, O: x
ALU Result: ALUOut:     0

Output Values:
T:   4
Address Register File: PC:     2, AR:     0, SP:   255
Instruction Register :    40
Register File Registers: R1:     0, R2:     0, R3:     0, R4:     0
Register File Scratch Registers: S1:     0, S2:     0, S3:     0, S4:     0
ALU Flags: Z: x, N: x, C: x, O: x
ALU Result: ALUOut:     0
```

Figure 3: Execution of BRA instruction

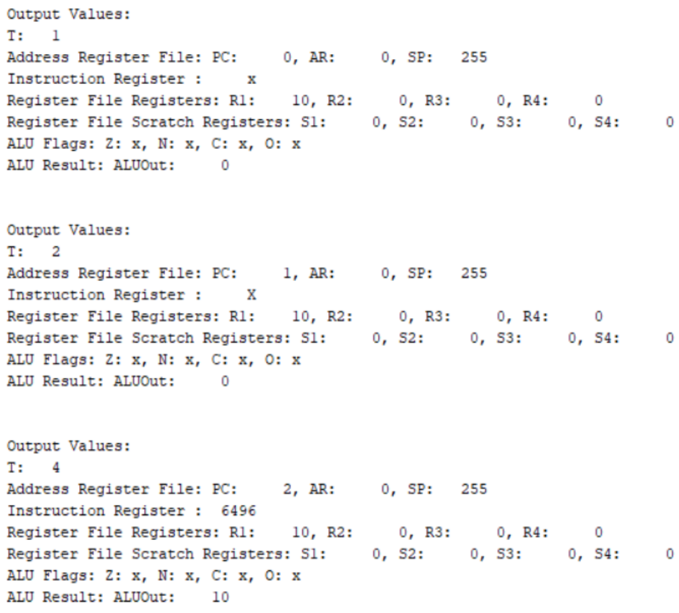


```
Output Values:
T:   8
Address Register File: PC:     2, AR:     0, SP:   255
Instruction Register :    40
Register File Registers: R1:     0, R2:     0, R3:     0, R4:     0
Register File Scratch Registers: S1:    40, S2:     0, S3:     0, S4:     0
ALU Flags: Z: x, N: x, C: x, O: x
ALU Result: ALUOut:     0

Output Values:
T:  16
Address Register File: PC:     2, AR:     0, SP:   255
Instruction Register :    40
Register File Registers: R1:     0, R2:     0, R3:     0, R4:     0
Register File Scratch Registers: S1:    40, S2:     2, S3:     0, S4:     0
ALU Flags: Z: x, N: x, C: x, O: x
ALU Result: ALUOut:    42

Output Values:
T:   1
Address Register File: PC:    42, AR:     0, SP:   255
Instruction Register :    40
Register File Registers: R1:     0, R2:     0, R3:     0, R4:     0
Register File Scratch Registers: S1:    40, S2:     2, S3:     0, S4:     0
ALU Flags: Z: x, N: x, C: x, O: x
ALU Result: ALUOut:    42
```

Figure 4: Execution of BRA instruction

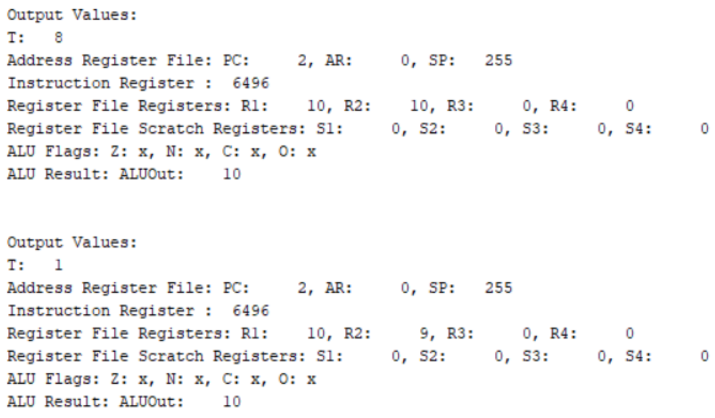


```
Output Values:
T:   1
Address Register File: PC:     0, AR:     0, SP:   255
Instruction Register :     x
Register File Registers: R1:    10, R2:     0, R3:     0, R4:     0
Register File Scratch Registers: S1:     0, S2:     0, S3:     0, S4:     0
ALU Flags: Z: x, N: x, C: x, O: x
ALU Result: ALUOut:     0


Output Values:
T:   2
Address Register File: PC:     1, AR:     0, SP:   255
Instruction Register :     X
Register File Registers: R1:    10, R2:     0, R3:     0, R4:     0
Register File Scratch Registers: S1:     0, S2:     0, S3:     0, S4:     0
ALU Flags: Z: x, N: x, C: x, O: x
ALU Result: ALUOut:     0


Output Values:
T:   4
Address Register File: PC:     2, AR:     0, SP:   255
Instruction Register :  6496
Register File Registers: R1:    10, R2:     0, R3:     0, R4:     0
Register File Scratch Registers: S1:     0, S2:     0, S3:     0, S4:     0
ALU Flags: Z: x, N: x, C: x, O: x
ALU Result: ALUOut:    10
```

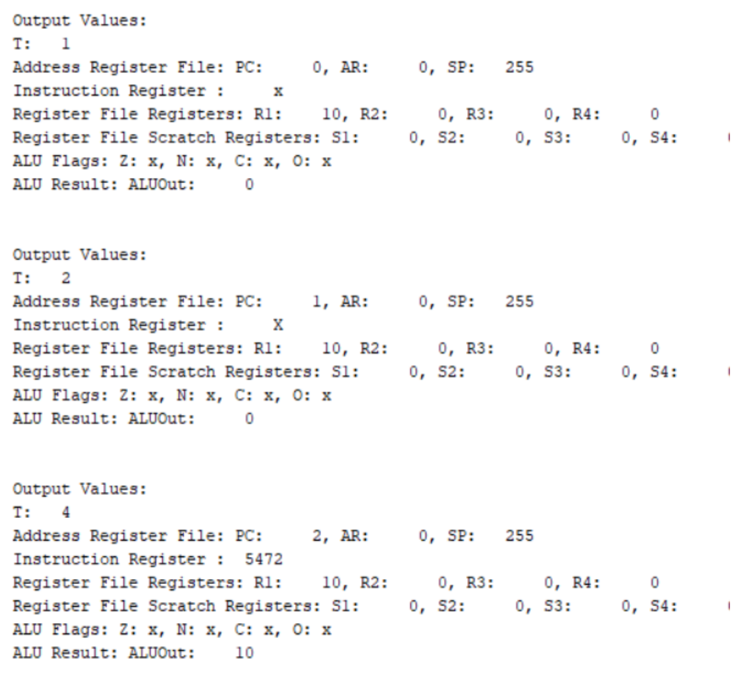
Figure 5: Execution of DEC instruction



```
Output Values:
T:   8
Address Register File: PC:     2, AR:     0, SP:   255
Instruction Register :  6496
Register File Registers: R1:    10, R2:    10, R3:     0, R4:     0
Register File Scratch Registers: S1:     0, S2:     0, S3:     0, S4:     0
ALU Flags: Z: x, N: x, C: x, O: x
ALU Result: ALUOut:    10


Output Values:
T:   1
Address Register File: PC:     2, AR:     0, SP:   255
Instruction Register :  6496
Register File Registers: R1:    10, R2:     9, R3:     0, R4:     0
Register File Scratch Registers: S1:     0, S2:     0, S3:     0, S4:     0
ALU Flags: Z: x, N: x, C: x, O: x
ALU Result: ALUOut:    10
```

Figure 6: Execution of DEC instruction

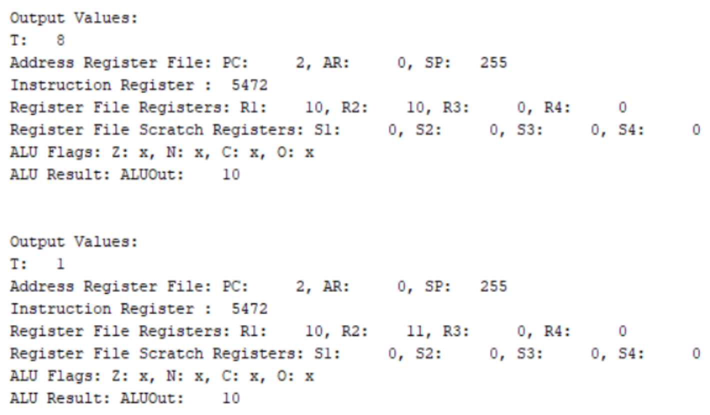


```
Output Values:
T:   1
Address Register File: PC:     0, AR:     0, SP:   255
Instruction Register :     x
Register File Registers: R1:    10, R2:     0, R3:     0, R4:     0
Register File Scratch Registers: S1:     0, S2:     0, S3:     0, S4:
ALU Flags: Z: x, N: x, C: x, O: x
ALU Result: ALUOut:     0


Output Values:
T:   2
Address Register File: PC:     1, AR:     0, SP:   255
Instruction Register :     X
Register File Registers: R1:    10, R2:     0, R3:     0, R4:     0
Register File Scratch Registers: S1:     0, S2:     0, S3:     0, S4:
ALU Flags: Z: x, N: x, C: x, O: x
ALU Result: ALUOut:     0


Output Values:
T:   4
Address Register File: PC:     2, AR:     0, SP:   255
Instruction Register :  5472
Register File Registers: R1:    10, R2:     0, R3:     0, R4:     0
Register File Scratch Registers: S1:     0, S2:     0, S3:     0, S4:
ALU Flags: Z: x, N: x, C: x, O: x
ALU Result: ALUOut:    10
```

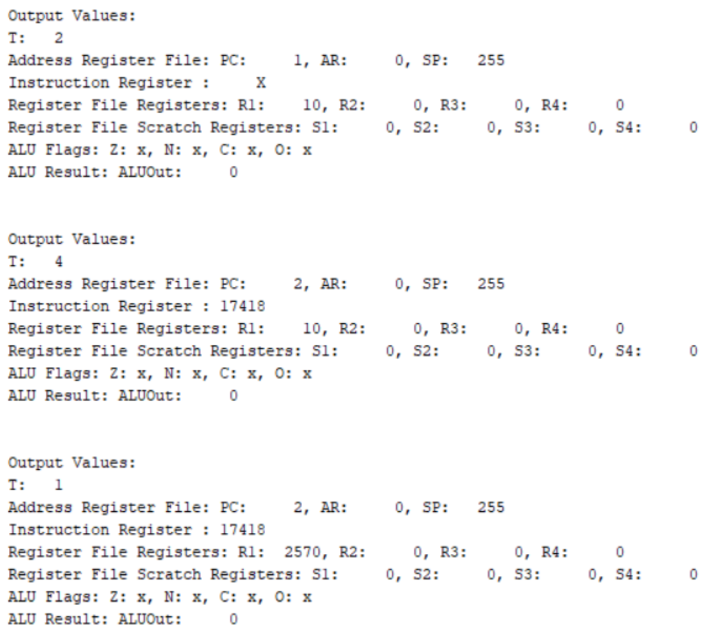
Figure 7: Execution of INC instruction



```
Output Values:
T:   8
Address Register File: PC:     2, AR:     0, SP:   255
Instruction Register :  5472
Register File Registers: R1:    10, R2:    10, R3:     0, R4:     0
Register File Scratch Registers: S1:     0, S2:     0, S3:     0, S4:     0
ALU Flags: Z: x, N: x, C: x, O: x
ALU Result: ALUOut:    10


Output Values:
T:   1
Address Register File: PC:     2, AR:     0, SP:   255
Instruction Register :  5472
Register File Registers: R1:    10, R2:    11, R3:     0, R4:     0
Register File Scratch Registers: S1:     0, S2:     0, S3:     0, S4:     0
ALU Flags: Z: x, N: x, C: x, O: x
ALU Result: ALUOut:    10
```

Figure 8: Execution of INC instruction

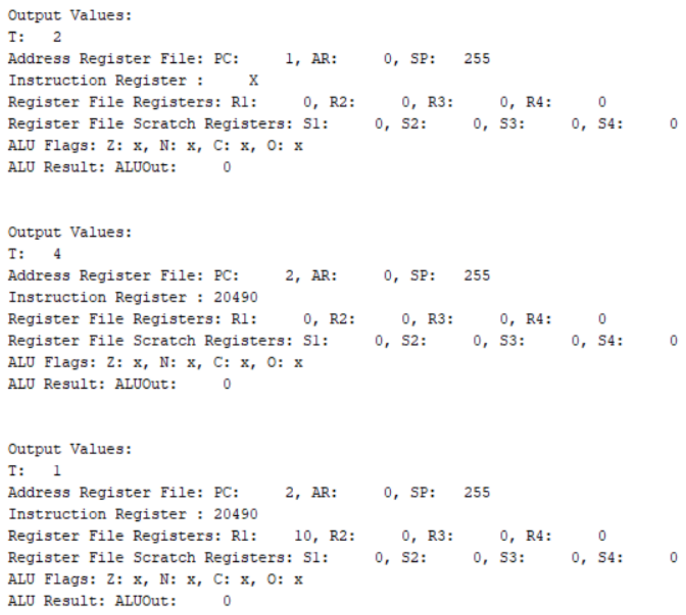


```
Output Values:
T:   2
Address Register File: PC:     1, AR:     0, SP:   255
Instruction Register :     X
Register File Registers: R1:    10, R2:     0, R3:     0, R4:     0
Register File Scratch Registers: S1:     0, S2:     0, S3:     0, S4:     0
ALU Flags: Z: x, N: x, C: x, O: x
ALU Result: ALUOut:     0


Output Values:
T:   4
Address Register File: PC:     2, AR:     0, SP:   255
Instruction Register : 17418
Register File Registers: R1:    10, R2:     0, R3:     0, R4:     0
Register File Scratch Registers: S1:     0, S2:     0, S3:     0, S4:     0
ALU Flags: Z: x, N: x, C: x, O: x
ALU Result: ALUOut:     0


Output Values:
T:   1
Address Register File: PC:     2, AR:     0, SP:   255
Instruction Register : 17418
Register File Registers: R1:  2570, R2:     0, R3:     0, R4:     0
Register File Scratch Registers: S1:     0, S2:     0, S3:     0, S4:     0
ALU Flags: Z: x, N: x, C: x, O: x
ALU Result: ALUOut:     0
```

Figure 9: Execution of MOVH instruction

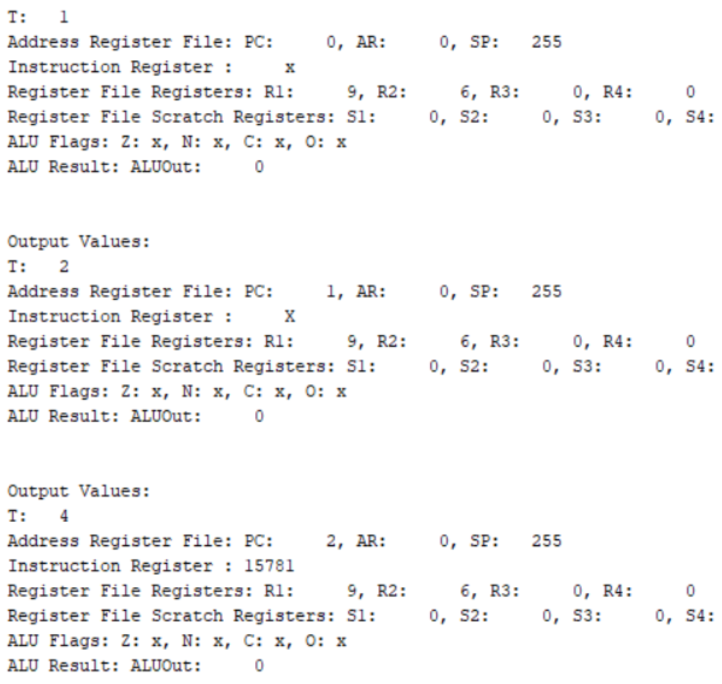


```
Output Values:
T:   2
Address Register File: PC:     1, AR:     0, SP:   255
Instruction Register :     X
Register File Registers: R1:     0, R2:     0, R3:     0, R4:     0
Register File Scratch Registers: S1:     0, S2:     0, S3:     0, S4:     0
ALU Flags: Z: x, N: x, C: x, O: x
ALU Result: ALUOut:     0


Output Values:
T:   4
Address Register File: PC:     2, AR:     0, SP:   255
Instruction Register : 20490
Register File Registers: R1:     0, R2:     0, R3:     0, R4:     0
Register File Scratch Registers: S1:     0, S2:     0, S3:     0, S4:     0
ALU Flags: Z: x, N: x, C: x, O: x
ALU Result: ALUOut:     0


Output Values:
T:   1
Address Register File: PC:     2, AR:     0, SP:   255
Instruction Register : 20490
Register File Registers: R1:    10, R2:     0, R3:     0, R4:     0
Register File Scratch Registers: S1:     0, S2:     0, S3:     0, S4:     0
ALU Flags: Z: x, N: x, C: x, O: x
ALU Result: ALUOut:     0
```

Figure 10: Execution of MOVL instruction

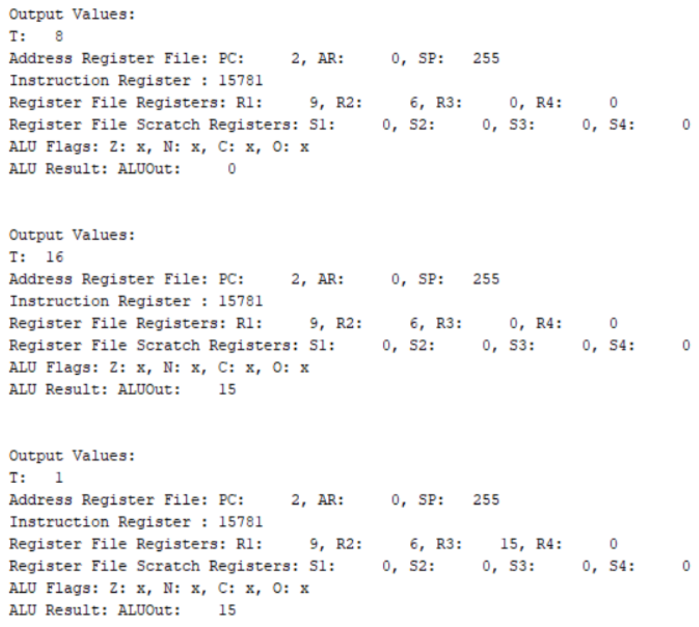


```
T:   1
Address Register File: PC:     0, AR:     0, SP:   255
Instruction Register :     x
Register File Registers: R1:     9, R2:     6, R3:     0, R4:     0
Register File Scratch Registers: S1:     0, S2:     0, S3:     0, S4:
ALU Flags: Z: x, N: x, C: x, O: x
ALU Result: ALUOut:     0


Output Values:
T:   2
Address Register File: PC:     1, AR:     0, SP:   255
Instruction Register :     X
Register File Registers: R1:     9, R2:     6, R3:     0, R4:     0
Register File Scratch Registers: S1:     0, S2:     0, S3:     0, S4:
ALU Flags: Z: x, N: x, C: x, O: x
ALU Result: ALUOut:     0


Output Values:
T:   4
Address Register File: PC:     2, AR:     0, SP:   255
Instruction Register : 15781
Register File Registers: R1:     9, R2:     6, R3:     0, R4:     0
Register File Scratch Registers: S1:     0, S2:     0, S3:     0, S4:
ALU Flags: Z: x, N: x, C: x, O: x
ALU Result: ALUOut:     0
```

Figure 11: Execution of XOR instruction

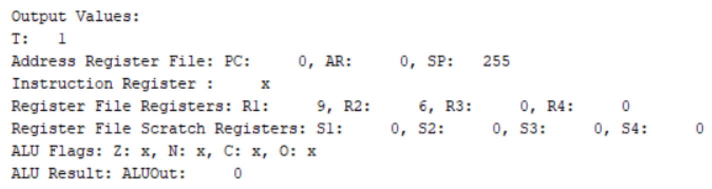


```
Output Values:
T:   8
Address Register File: PC:     2, AR:     0, SP:   255
Instruction Register : 15781
Register File Registers: R1:     9, R2:     6, R3:     0, R4:     0
Register File Scratch Registers: S1:     0, S2:     0, S3:     0, S4:     0
ALU Flags: Z: x, N: x, C: x, O: x
ALU Result: ALUOut:     0


Output Values:
T:  16
Address Register File: PC:     2, AR:     0, SP:   255
Instruction Register : 15781
Register File Registers: R1:     9, R2:     6, R3:     0, R4:     0
Register File Scratch Registers: S1:     0, S2:     0, S3:     0, S4:     0
ALU Flags: Z: x, N: x, C: x, O: x
ALU Result: ALUOut:    15


Output Values:
T:   1
Address Register File: PC:     2, AR:     0, SP:   255
Instruction Register : 15781
Register File Registers: R1:     9, R2:     6, R3:    15, R4:     0
Register File Scratch Registers: S1:     0, S2:     0, S3:     0, S4:     0
ALU Flags: Z: x, N: x, C: x, O: x
ALU Result: ALUOut:    15
```

Figure 12: Execution of XOR instruction

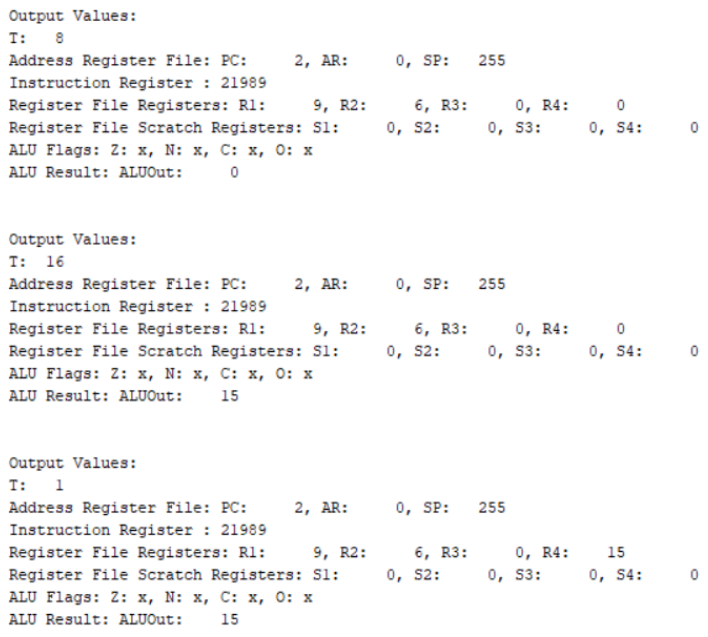


```
Output Values:
T:   1
Address Register File: PC:     0, AR:     0, SP:   255
Instruction Register :     x
Register File Registers: R1:     9, R2:     6, R3:     0, R4:     0
Register File Scratch Registers: S1:     0, S2:     0, S3:     0, S4:     0
ALU Flags: Z: x, N: x, C: x, O: x
ALU Result: ALUOut:     0


Output Values:
T:   2
Address Register File: PC:     1, AR:     0, SP:   255
Instruction Register :     X
Register File Registers: R1:     9, R2:     6, R3:     0, R4:     0
Register File Scratch Registers: S1:     0, S2:     0, S3:     0, S4:     0
ALU Flags: Z: x, N: x, C: x, O: x
ALU Result: ALUOut:     0


Output Values:
T:   4
Address Register File: PC:     2, AR:     0, SP:   255
Instruction Register : 21989
Register File Registers: R1:     9, R2:     6, R3:     0, R4:     0
Register File Scratch Registers: S1:     0, S2:     0, S3:     0, S4:     0
ALU Flags: Z: x, N: x, C: x, O: x
ALU Result: ALUOut:     0
```

Figure 13: Execution of ADD instruction

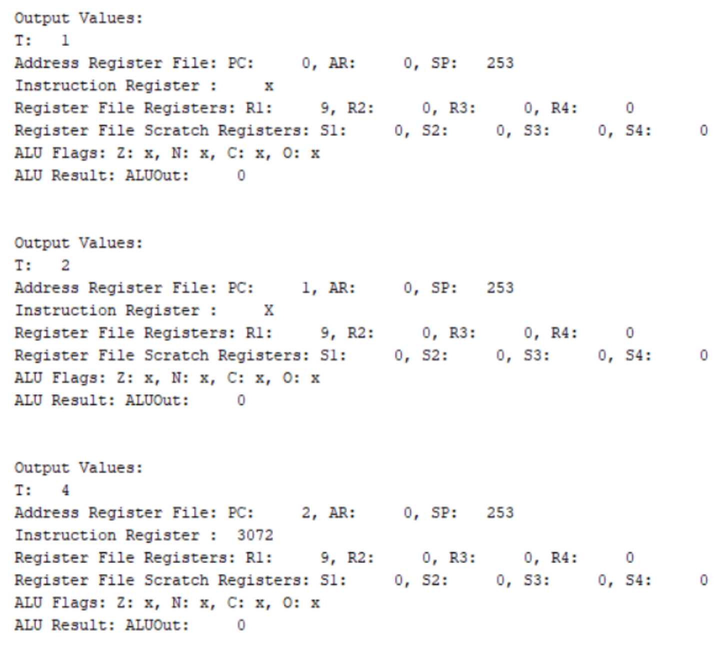


```
Output Values:
T:   8
Address Register File: PC:     2, AR:     0, SP:   255
Instruction Register : 21989
Register File Registers: R1:     9, R2:     6, R3:     0, R4:     0
Register File Scratch Registers: S1:     0, S2:     0, S3:     0, S4:     0
ALU Flags: Z: x, N: x, C: x, O: x
ALU Result: ALUOut:     0


Output Values:
T:  16
Address Register File: PC:     2, AR:     0, SP:   255
Instruction Register : 21989
Register File Registers: R1:     9, R2:     6, R3:     0, R4:     0
Register File Scratch Registers: S1:     0, S2:     0, S3:     0, S4:     0
ALU Flags: Z: x, N: x, C: x, O: x
ALU Result: ALUOut:    15


Output Values:
T:   1
Address Register File: PC:     2, AR:     0, SP:   255
Instruction Register : 21989
Register File Registers: R1:     9, R2:     6, R3:     0, R4:    15
Register File Scratch Registers: S1:     0, S2:     0, S3:     0, S4:     0
ALU Flags: Z: x, N: x, C: x, O: x
ALU Result: ALUOut:    15
```

Figure 14: Execution of ADD instruction

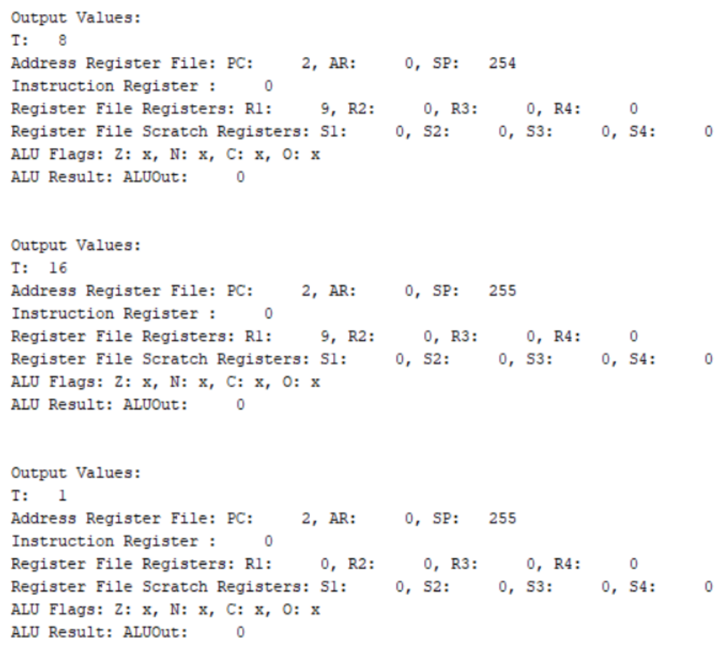


```
Output Values:
T:   1
Address Register File: PC:     0, AR:     0, SP:   253
Instruction Register :     x
Register File Registers: R1:     9, R2:     0, R3:     0, R4:     0
Register File Scratch Registers: S1:     0, S2:     0, S3:     0, S4:     0
ALU Flags: Z: x, N: x, C: x, O: x
ALU Result: ALUOut:     0


Output Values:
T:   2
Address Register File: PC:     1, AR:     0, SP:   253
Instruction Register :     X
Register File Registers: R1:     9, R2:     0, R3:     0, R4:     0
Register File Scratch Registers: S1:     0, S2:     0, S3:     0, S4:     0
ALU Flags: Z: x, N: x, C: x, O: x
ALU Result: ALUOut:     0


Output Values:
T:   4
Address Register File: PC:     2, AR:     0, SP:   253
Instruction Register :  3072
Register File Registers: R1:     9, R2:     0, R3:     0, R4:     0
Register File Scratch Registers: S1:     0, S2:     0, S3:     0, S4:     0
ALU Flags: Z: x, N: x, C: x, O: x
ALU Result: ALUOut:     0
```

Figure 15: Execution of POP instruction



```
Output Values:
T:   8
Address Register File: PC:     2, AR:     0, SP:   254
Instruction Register :     0
Register File Registers: R1:     9, R2:     0, R3:     0, R4:     0
Register File Scratch Registers: S1:     0, S2:     0, S3:     0, S4:     0
ALU Flags: Z: x, N: x, C: x, O: x
ALU Result: ALUOut:     0


Output Values:
T:  16
Address Register File: PC:     2, AR:     0, SP:   255
Instruction Register :     0
Register File Registers: R1:     9, R2:     0, R3:     0, R4:     0
Register File Scratch Registers: S1:     0, S2:     0, S3:     0, S4:     0
ALU Flags: Z: x, N: x, C: x, O: x
ALU Result: ALUOut:     0


Output Values:
T:   1
Address Register File: PC:     2, AR:     0, SP:   255
Instruction Register :     0
Register File Registers: R1:     0, R2:     0, R3:     0, R4:     0
Register File Scratch Registers: S1:     0, S2:     0, S3:     0, S4:     0
ALU Flags: Z: x, N: x, C: x, O: x
ALU Result: ALUOut:     0
```

Figure 16: Execution of POP instruction

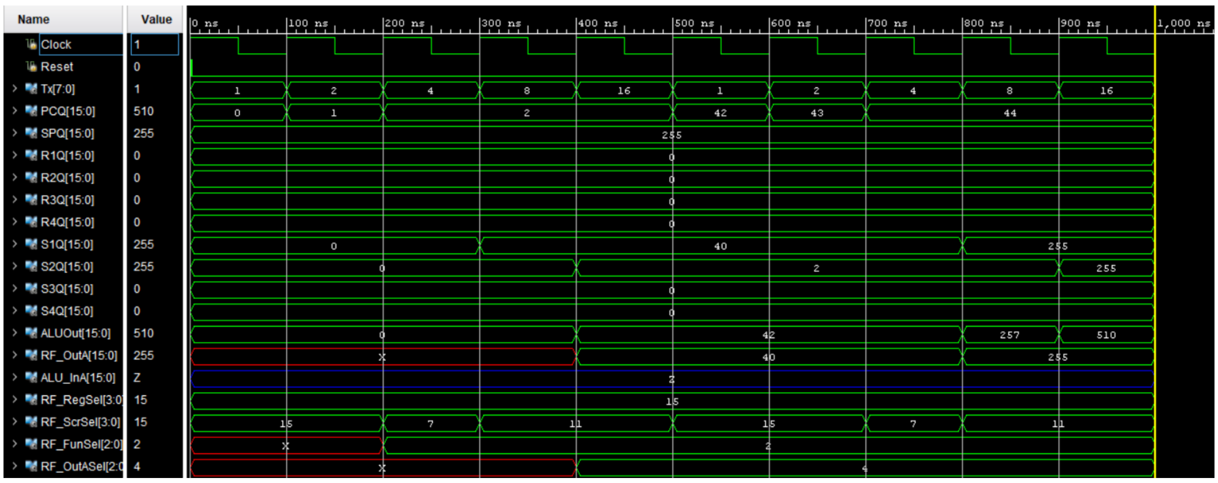


Figure 17: signals' behavior in BRA execution

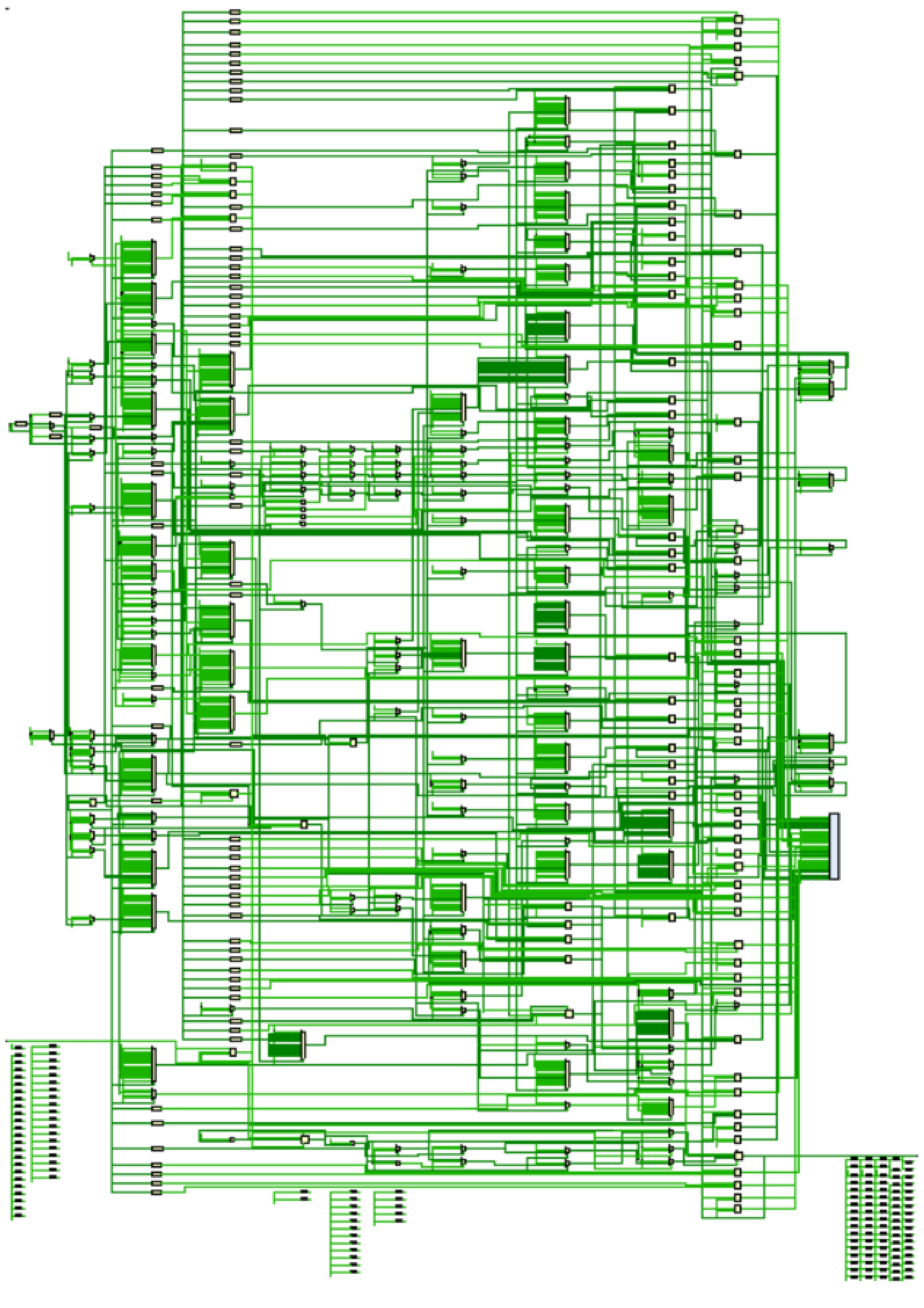


Figure 18: Schenatic of CPU System

4 DISCUSSION [25 points]

Testing our implementation relied on output examples. In our initial setup, we set the PC to start from the first address, while the SP began from the bottom of the memory. This configuration facilitated testing, particularly for instructions like POP (Figure 15-16) and PUSH, where we adjusted the memory file's first addresses according to the chosen instruction. As detailed in the Results section, all outputs aligned with our implementation and matched the RTL and explanations provided earlier in the Methods subsection. Although not all results were included in the Results section, they were consistent with our expectations.

Simulation played a pivotal role in validating our implementation. Initially, encountering incorrect results prompted us to check our code details and fix any mistakes. This process significantly enhanced our understanding of the relationship between all the parts of our system and how the connections work between them.

Each instruction presented above corresponds to a specific RTL and detailed explanation in the Methods section. To execute each micro-operation, we followed the system designed in the first project, generating the necessary control signals. This task posed a challenge, as enabling a register and applying an operation required careful consideration of subsequent clock signals to disable it when necessary, preventing unexpected outputs.

Using a modified simulation file allowed us to track all registers, wires, and signals within our system, facilitating error identification and understanding the reasons behind the misbehaving we used to get. An example of such simulation, specifically for the BRA instruction, has been included in the Results section. This simulation not only confirms the correct functioning of our sequence counter and Program Counter (PC) but also provides insights into our system's behavior.

The Schematic provided in the results part is so confusing. Since we are implementing a hardwired system a lot of gates and connections were present in the system.

5 CONCLUSION [10 points]

We improved our hardwired design skills and learned the key points of designing a control unit. Also, we created new simulation sources and used debugging features of the Vivado environment. While working on this project, we encountered challenges related to understanding Verilog's different blocks and features. However, these obstacles motivated us to learn more. In the end, our perseverance paid off, and we successfully implemented the CPU system as required. To conclude, we finished building a basic computer and tested its features, confirming that the material covered was fully understood.