ISTANBUL TECHNICAL UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

BLG 222E
COMPUTER ORGANIZATION
PROJECT 2 REPORT

CRN	:	21334
LECTURER	:	Doç. Dr. Gökhan İnce

GROUP MEMBERS:

150210116	:	YASİN SAYMAZ
150220031	:	ZELİHA MELEK BEKDEMİR

SPRING 2024

Contents

1 INTRODUCTION

We started by designing a Sequence Counter for keeping track of the cycles of operations. We designed a decoder that takes the output of this sequence counter. Sequence Decoder uses the binary representation of 7-bit T to give T_0, T_1, T_2, T_3 T_4, T_5, T_6, T_7 a return value 0 or 1. Only one of the Ts can be at any time. We continued by determining how many clock cycles need, how we can group them and which signals they require to work. We coded fetch and decode operations. For the later parts, we needed to decode the IROut[15:10] to find out which operation we were going to implement, therefore we, also, implemented an Instruction Decoder. It also returns 1 For only one operation and 0 for the rest.

With T_2, we started our operation cycles. Our system has two types of instruction code.

OPCODE(6-BIT)	RSEL(2-BIT)	ADDRESS(8-BIT)

OPCODE(6-BIT)	S(1-BIT)	DSTREG(3-BIT)	SREG1(3-BIT)	SREG2(3-BIT)

Based on the value of OPCODE, our system executes different set of instructions. For some operations, register selection values are given in the instruction code and we needed to consider every type of register we need in that specific operation. We decided to implement another module for returning selection values for different IROut[x:y] values.

After considering every possible clock cycle and deciding on which signals to use, we have completed our control unit.

Finally, our CPUSystem module takes a CLock, Reset and T input and calls for Control Unit, ALU System, IR Decodor, Sequence Counter, Sequence Decoder by giving the right parameters. It represents the whole system.

2 MATERIALS AND METHODS [40 points]

2.1 SEQUENCE COUNTER

- Inputs
  - Clock
  - Reset
- Outputs

  7:0 T

With every positive-edge of clock, counter goes up until max cycle is obtained or reset is given as 1.

We implemented this module so that when the system has started, counter has an initial value of 7'b1000000, because this value is never used in our system. As the clock goes from low to high, this counter does a cyclic left transition by shifting the first six bits to left by one bit and writing the seventh bit as the first. This operation continues until Reset becomes 1. When Reset is 1, counter becomes 7'b0000001.

This module returns an output T that will later be decoded in Sequence Decoder.

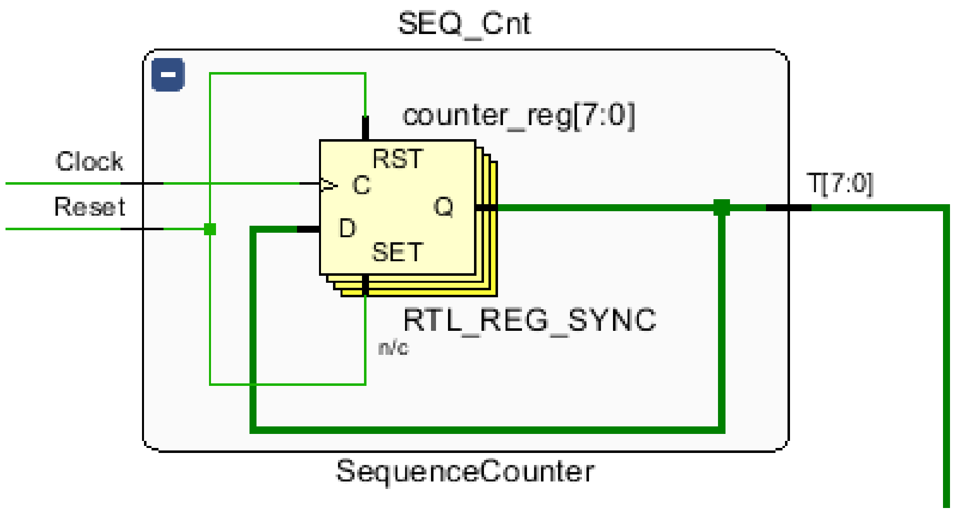


Figure 1: Sequence Counter

2.2 SEQUENCE DECODER

- Inputs

  7:0 T

- Outputs

  - T_0, T_1, T_2, T_3 T_4, T_5, T_6, T_7

For these Ts it performs a decoding so that

- T_0 = !T[7]&!T[6]&!T[5]&!T[4]&!T[3]&!T[2]&!T[1]&T[0]
- T_1 = !T[7]&!T[6]&!T[5]&!T[4]&!T[3]&!T[2]&T[1]&!T[0]
- T_2 = !T[7]&!T[6]&!T[5]&!T[4]&!T[3]&T[2]&!T[1]&!T[0]
- T_3 = !T[7]&!T[6]&!T[5]&!T[4]&T[3]&!T[2]&!T[1]&!T[0]
- T_4 = !T[7]&!T[6]&!T[5]&T[4]&!T[3]&!T[2]&!T[1]&!T[0]
- T_5 = !T[7]&!T[6]&T[5]&!T[4]&!T[3]&!T[2]&!T[1]&!T[0]
- T_6 = !T[7]&T[6]&!T[5]&!T[4]&!T[3]&!T[2]&!T[1]&!T[0]
- T_7 = T[7]&!T[6]&!T[5]&!T[4]&!T[3]&!T[2]&!T[1]&!T[0]

Because of the nature of Sequence Counter, this decoding gives 1 to only one of the above at any time.

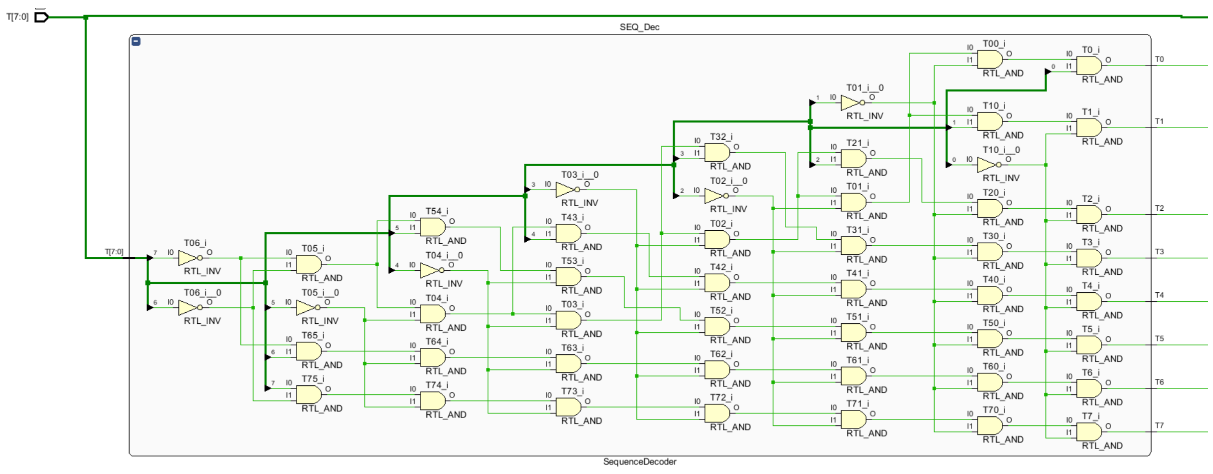


Figure 2: Sequence Decoder

2.3 INSTRUCTION DECODER

For the convenience of using only the names of operations, we decided to use a decoder that returns 0 or 1 based on the opcode that is given in IROut[15:10]. Based on the hexadecimal representations of the opcodes given in the homework paper, we used expressions that will give the corresponding values by using the gicen opcode signal.

An example can be given as:

BRA = !T[6]&!T[5]&!T[4]&!T[3]&!T[2]&!T[1]&!T[0]

Taking the complement of every zero digit and anding with the digits that are one is derived from the logic that the operation must return if and only if the opcode is 000000. Since BRA is only 1 when all the digits are 0, we took every digit' complement and combine them with and. We applied the same logic for every operation and returned them as output. For 34 operations, based on the given opcode only one of them is 1 at any given time.

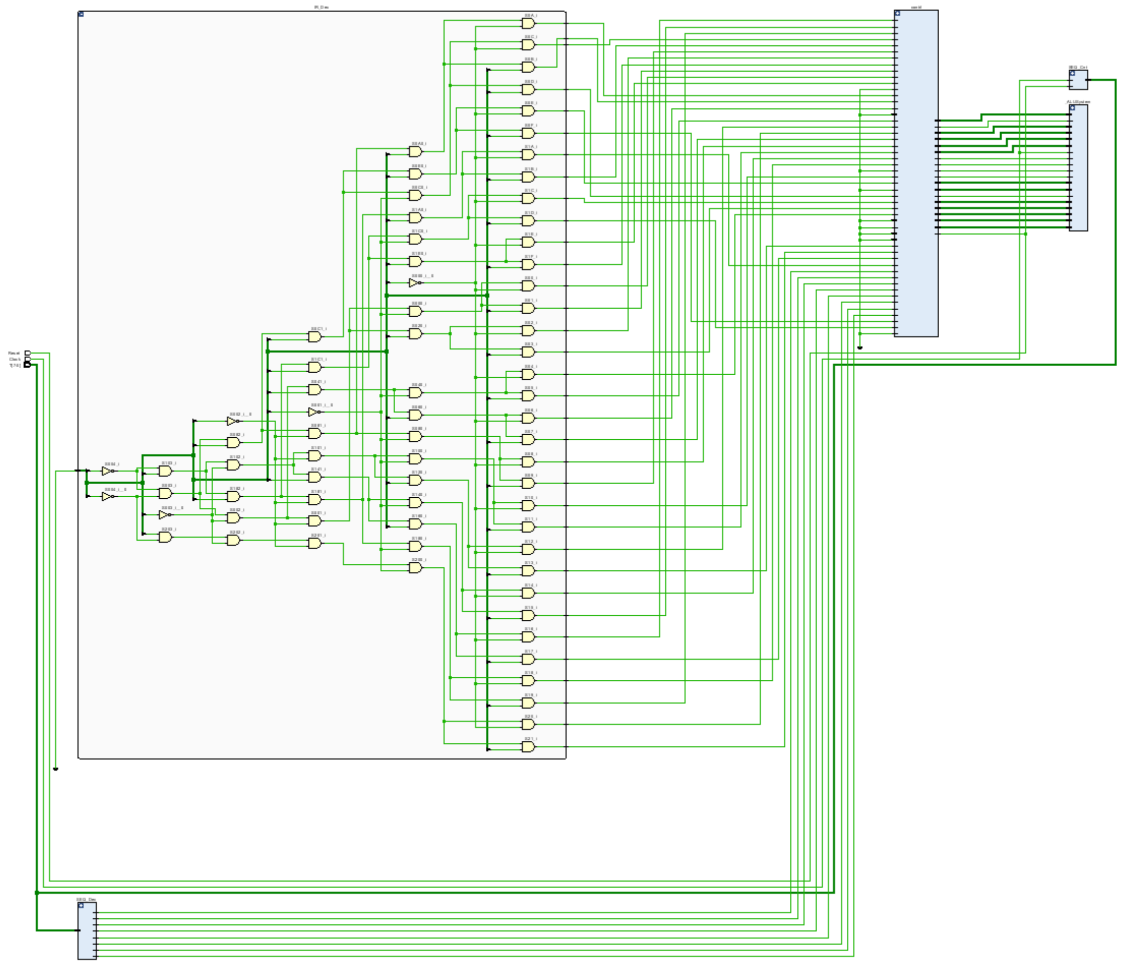


Figure 3: Instruction Decoder

2.4 DSTREG/SREG1/SREG2/RSEL

For the first type of instruction, the register we use in the operations depends on the selection codes on the instruction. Again, rather than checking every possible case in our control unit module, we implemented a function that takes IROut[9:8]as input and returns a RegSel value for register selection.

It uses a case(X) structure similar to to one we implemented in the module for DSTREG/SREG1/SREG2 options. By looking at the RegSel values that were given in the first homework, we assigned the correct values and returned them.

Algorithm 1: Pseudocode for RSEL module

```
1: switch RSEL do
2:     case 2'b00 do RegSel <= 4'0111;
3:     ...
4:     case 2'b11 do RegSel <= 4'1110;
5: end
6: return RegSel
```

For the second type of instruction, source and destination registers depend on the selection codes on the instruction. Rather than checking every possible case in our control unit module, we decided to use functions that takes the necessary parts of IROut and uses it to return a RegSel value for register selection.

It uses a case(X) structure to decide for different DSTREG/SREG1/SREG2 options. Since given DSTREG,SREG1 and SREG2 values are same, their implementation is same except the inputs that are given in the main CPUSystem are different.

For DSTREG we give IROut[8:6], for SREG1 we give IROut[5:3] and for SREG2 we give IROut[2:0].

By looking at the RegSel values that were given in the first homework, we assigned the correct values and returned them.

Algorithm 2: Pseudocode for DSTREG/SREG1/SREG2 modules

```
1: switch DSTREG/SREG1/SREG2 do
2:     case 3'b000 do RegSel <= 3'011;
3:     ...
4:     case 3'b111 do RegSel <= 4'1110;
5: end
6: return RegSel
```

2.5 CONTROL UNIT

We started by separating all the instructions into parts so that they have similar operations to each other. Instructions in the same group have the same number of clocks and perform similar operations on the same clocks.

Operation	**Clock**
BRA, BNE, BEQ	T5
POP	T3
PSH	T4
INC, DEC	T5
LSL, LSR, ASR, CSL, CSR, NOT, MOVS	T4
AND, OR, XOR, NAND, ADD, ADC, SUB, ADDS, SUBS, ANDS, ORRS, XORS	T5
MOVH, MOVL	T2
LDR	T3
STR	T3
BL	T4
BX	T5
LDRIM	T3
STRIM	T6

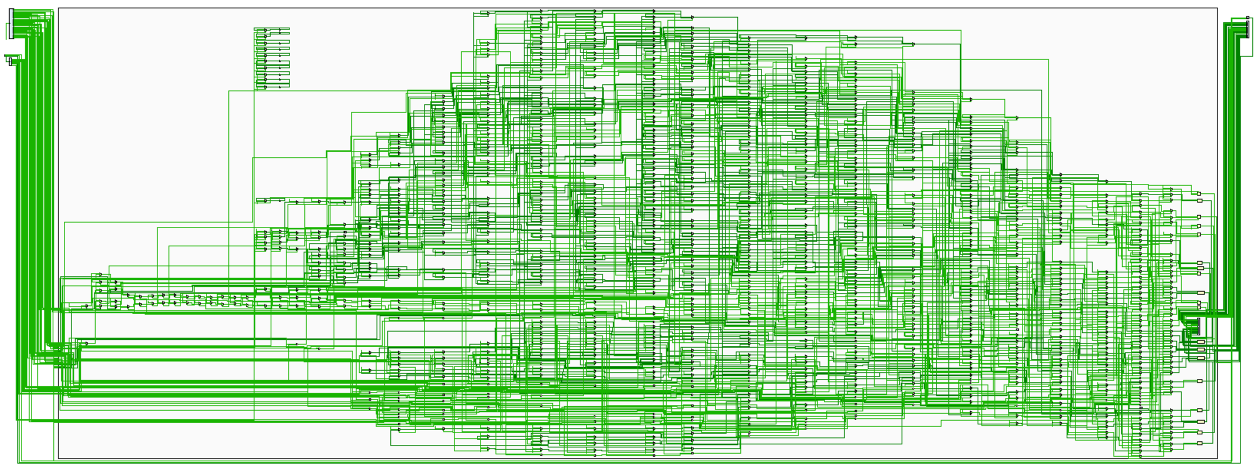


Figure 4: Control Unit

2.5.1 BRA,BNE,BEQ

We can complete these 3 types of operations in 5 clock cycle, including fetch and decode operations. This operations assigns PC+VALUE to PC.

OPCODE & SYMBOL	**OPERATION**
0x00 & BRA	PC ← PC + VALUE
0x01 & BNE	IF Z=0 THEN PC ← PC + VALUE
0x02 & BEQ	IF Z=1 THEN PC ← PC + VALUE

Regardless of which of these 3 operations, we jointly perform the PC ← PC + VALUE operation. But all of these have different conditions. According to this information, we wrote our code as shown below.

```
(BRA & Tx)|(BNE & Tx & !Z)|(BEQ & Tx &Z)
```

Since x indicates the clock value after the fetch operation, it will take all values from 2 to 5. Now let's examine what we do in each clock cycle.

In *T2* we do:

→ Select PC in ARF
→ Send it RF via MUXA
→ Select S1 and Load function in RF
→ Disable all registers in ARF

In *T3*:

→ Select PC in ARF
→ Send it RF via MUXA
→ Select S1 and Load bus to the RF
→ Disable all registers in ARF

In *T4*:

→ Select and send S1 to ALU via A wire
→ Select and send S2 to ALU via B wire
→ Select A+B function in ALU and operate it

In *T5*:

→ Load ALUOut to bus
→ Send it ARF via MUXB
→ Select PC and Load function in RF
→ Set RESET to 1

2.5.2 POP

We can complete POP operation in 3 clock cycles. POP operations increment SP by 1 and load Value stored in memory at the address pointed by the Stack Pointer to Rx.

OPCODE & SYMBOL	**OPERATION**
0x03 & POP	SP $\leftarrow$ SP + 1, Rx $\leftarrow$ M[SP]

We set our condition line for the POP Instruction like this:

```
(POP & Tx)
```

Since we can complete POP operation in 3 clock , x can take only 2 and 3.
Now let's examine what we do in each clock cycle.

In *T2* we do:

$\rightarrow$ Enable Read in Memory and Load MemOut to the BUS
$\rightarrow$ Send it to the RF via MUXA
$\rightarrow$ Select Rx and Load LOW part of the BUS
$\rightarrow$ Select SP in ARF and Load it to the OutD wire
$\rightarrow$ Enable SP in ARF and Increment it.

In *T3*:

$\rightarrow$ Enable Read in Memory and Load MemOut to the BUS
$\rightarrow$ Send it to the RF via MUXA
$\rightarrow$ Select Rx and Load HIGH part of the BUS
$\rightarrow$ Send it RF via MUXA
$\rightarrow$ Select S1 and Load bus to the RF
$\rightarrow$ Disable all registers in ARF

2.5.3 PSH

We can complete PSH operation in 4 clock cycles. PSH operation decrement SP by 1 and stores the value of the register Rx into the memory location pointed to by the Stack Pointer.

OPCODE & SYMBOL	OPERATION
0x04 & PSH	M[SP] ← Rx, SP ← SP – 1

We set our condition line for the POP Instruction like this:

```
(PSH & Tx)
```

Since we can complete PSH operation in 4 clock , x can take values between 2 and 4. Before analysing clocks, we know RSEL[1:0] and OutASel[1:0] points exactly same registers R1,R2,R3 and R4 in order.
Now let's examine what we do in each clock cycle.

In *T2* we do:

→ Select Rx register in RF according to RSEL[1:0] signal.
→ Assign A value to ALU by using OutA wire.

In *T3*:

→ Enable Write in Memory
→ Select SP in ARF and load to the OutD wire.
→ Point the address of Memory
→ Load Memory with the HIGH part of BUS using MUXC
→ Select SP in ARF and decrement it

In *T4*:

→ Enable Write in Memory
→ Select SP in ARF and load to the OutD wire.
→ Point the address of Memory
→ Load Memory with the LOW part of BUS using MUXC

2.5.4 INC,DEC

We can complete these 2 types of operations in 6 clock cycle, including fetch and decode operations. This operations assigns ıncrements or decrements the value in the given SREG1 registers and write it into DSTREG.

OPCODE & SYMBOL	OPERATION
0x05 & INC	DSTREG ← SREG1 + 1
0x06 & DEC	DSTREG ← SREG1 – 1

Other than the operations they performed, the structure of the code is same. The difference is in between the ARF_Funsel, RF_Funsel or ALU_Funsel.

```
((INC & Tx)|(DEC & Tx))
```

Since x indicates the clock value after the fetch operation, it will take all values from 2 to 5. Now let's examine what we do in each clock cycle.

In *T2* we do:

→ Check if DSTREG and SREG1 are same.
If they are same,

- Check if DSTREG is in ARF or RF
- Select source register and assign it to RF_RegSel or ARF_RegSel depending on DSTREG's location.
- Check the operation and update the ARF_FunSel or RF_FunSel. In other words, increment or decrement.
- Set RESET to 1

If they are not same,

- Check DSTEG[2] to find out where our destination is
- Check SREG1[2] to find where our source register is
  If DSTREG[2] == 1'b1,
  * If SREG1[2] == 1'b1, send the source register into ALU, select A function of ALU and give 0 to ALU_WF.
  * If SREG1[2] == 1'b0, send the source registers value to RF via MUXA, select destination register and load.

If DSTREG[2] == 1'b0,

  * If SREG1[2] == 1'b1, send the source register into ALU, select A function of ALU and give 0 to ALU_WF.
  * If SREG1[2] == 1'b0, send the source registers value to RF via MUXA, select S1 and load.

In *T3*:

→ Check DSTREG[2]
If it is 1,

- Check SREG1[2]
  If it is 1,
  * Connect ALUOut to RF via MUXA
  * Select load function
  * Select destination register

  If it is 0,
  * Select the RF_FunSel as increment or decrement depending on the operation.
  * Set RESET 1.

If it is 0,

- Check SREG1[2]
  If it is 1,
  * Connect ALUOut to ARF via MUXB
  * Select load function
  * Select destination register

  If it is 0,
  * Send S1 as an input to ALU
  * Select A function
  * Select WF as 0

In *T4*:

→ Check DSTREG[2]
If it is 1,

  - Check SREG1[2]
    If it is 1,
    * Choose RF_FunSel depending on the operations as increment or decrement
    * Select destination register
    * Set RESET as 1.

If it is 0,

  - Check SREG1[2]
    If it is 1,
    * Select ARF_FunSel as increment or decrement
    * Select destination register
    * Set RESET as 1

    If it is 0,
    * Connect ALUOut to ARF via MUXB
    * Select load function
    * Select destination register

In *T5*:

→ Select destination register
→ Select ARF_FunSel
→ Set RESET as 1

2.5.5 LSL, LSR, ASR, CSL, CSR, NOT, MOVS

We can complete these 7 types of operations in 4 clock cycle, including fetch and decode operations. All of the operations perform similar things. They sent SREG1 to ALU and after any process in ALU, sent ALUOut to the DSTREG.

LSL operation does Logical Left Shift, LSR operation does Logical Right Shift, ASR operation does Arithmetic Shift Right, CSL operation does Circular Shift Left, CSR operation does Circular Shift Right, NOT operation does Logical NOT, MOVS operation does Move and Set Flags.

OPCODE & SYMBOL	**OPERATION**
0x07 & LSL	DSTREG ← LSL SREG1
0x08 & LSR	DSTREG ← LSR SREG1
0x09 & ASR	DSTREG ← ASR SREG1
0x0A & CSL	DSTREG ← CSL SREG1
0x0B & CSR	DSTREG ← CSR SREG1
0x0E & NOT	DSTREG ← NOT SREG1
0x18 & MOVS	DSTREG ← SREG1 (Flags changes)

We set our condition line for the POP Instruction like this:

```
((LSL| LSR  | ASR | CSL | CSR | NOT | MOVS) & Tx)
```

Since we can complete all these operations in 4 clock, x can take values between 2 and 4. Now let's examine what we do in each clock cycle.

In *T2* we do:

→ Check whether SREG1 is in RF or NOT.
→ If SREG1 is in RF

- Select Rx in RF according to SREG[1:0] signal
- Set S to ALU_WF to check whether flags will change or not.
- Perform one of the 7 Operations according to the signal from the FunSel.

→ If SREG1 is not in RF.

- Select Rx in ARF according to SREG[1:0] signal
- Load it to the BUS and sent it to RF via MUXA
- Enable S1 in RF and

In *T3*:

→ Check whether SREG1 is in RF or NOT.
→ If SREG1 is in RF

Check whether DSTREG is in RF or NOT. (In the same place)

- If DSTREG is in RF
  - Send the value loaded on BUS to RF via MUXA
  - Select Register in RF using the DestinationDR module.
  - Select Load function in RF
- IF DSTREG is not in RF
  - Send the value loaded on BUS to ARF via MUXB
  - Select Register in ARF using the DestinationDR module.
  - Select Load function in ARF

→ If SREG1 is not in RF.

- Select R1 in RF and send it to ALU via OutA
- Set S to ALU_WF to check whether flags will change or not.
- Perform one of the 7 Operations according to the signal from the FunSel.

In *T4*:

→ Check whether DSREG is in RF or NOT.
→ If DSREG is in RF

- Send the value loaded on BUS to RF via MUXA
- Select Register in RF using the DestinationDR module.
- Select Load function in RF

→ If DSREG is not in RF.

- Send the value loaded on BUS to ARF via MUXB
- Select Register in ARF using the DestinationDR module.
- Select Load function in ARF

2.5.6 AND, OR, XOR, NAND, ADD, ADC, SUB, ADDS, SUBS, ANDS, ORRS, XORS

We can complete these 2 types of operations in 6 clock cycle, including fetch and decode operations. These operations do arithmetic or logical operations for the values in the given SREG1 and SREG2 registers and write it into DSTREG.

OPCODE & SYMBOL	OPERATION
0x0C & AND	DSTREG ← SREG1 AND SREG2
0x0D & ORR	DSTREG ← SREG1 OR SREG2
0x0F & XOR	DSTREG ← SREG1 XOR SREG2
0x10 & NAND	DSTREG ← SREG1 NAND SREG2
0x15 & ADD	DSTREG ← SREG1 + SREG2
0x16 & ADC	DSTREG ← SREG1 + SREG2 + CARRY
0x17 & SUB	DSTREG ← SREG1 - SREG2
0x18 & ADDS	DSTREG ← SREG1 + SREG2, Flags will change
0x1A & SUBS	DSTREG ← SREG1 - SREG2, Flags will change
0x1B & ANDS	DSTREG ← SREG1 AND SREG2, Flags will change
0x1C & ORRS	DSTREG ← SREG1 OR SREG2, Flags will change
0x1D & XORS	DSTREG ← SREG1 XOR SREG2, Flags will change

Other than the operations they performed, the structure of the code is nearly same. The difference is usually in between the ALU_Funsel.

```
((AND & Tx)|(ORR & Tx)|(XOR & Tx)|(NAND & Tx)|(ADD & Tx)|(ADC & Tx)|
(SUB & Tx)|(ADDS & Tx)|(SUBS & Tx)|(ANDS & Tx)|(ORRS & Tx)|(XORS & Tx))
```

Since x indicates the clock value after the fetch operation, it will take all values from 2 to 5. Now let’s examine what we do in each clock cycle.

In *T2* we do:

→ Set ALU_WF as S

→ Check if SREG2[2] and SREG1[2] are same.
If they are same,

- Check SREG1[2]
  If they are in RF,
  - Send SREG1 and SREG2 's values to ALU.
  - Set AlU_FunSel depending on the operation.

  If they are in ARF,
  - Send SREG1's value to RF via MUXA
  - Select load signal
  - Write to S1

If they are not same,

- Check SREG1[2] to find out where our destination is If SREG1[2] == 1'b1,
  - Send SREG2 to RF via MUXA
  - Select load signal
  - Write to S1

  If it is 0,
  - Send SREG1 to RF via MUXA
  - Select load signal
  - Write to S1

In *T3*:

→ Check if SREG2[2] and SREG1[2] are same.
If they are same,

- Check SREG1[2]
  If they are in RF,
  - Check DSTREG[2]. If it is RF, we connect ALUOut to RF via MUXA, select load signal and select the destination register. If it is ARF, we connect ALUOut to ARF via MUXB, select load signal and select the destination register.

        * Set RESET as 1.

      If they are in ARF,

        * Send SREG2's value to RF via MUXA
        * Select load signal
        * Write to S2

  If they are not same,

    - Check SREG1[2]
      If SREG1[2] == 1'b1,

        * Send SREG1 and S1 as input to ALU
        * Set ALU_WF as S
        * Assign the operation function to ALu_FunSel

      If it is 0,

        * Send SREG2 and S1 as input to ALU
        * Set ALU_WF as S
        * Assign the operation function to ALu_FunSel

In *T4*:

→ Check if SREG2[2] and SREG1[2] are same.
If they are same,

- Send S1 and S2 as input to ALU
- Set ALU_WF as S
- Assign the operation function to ALu_FunSel

If they are not same,

- Check DSTREG[2]
  If DSTREG[2] == 1'b1,

    * Connect ALUOut to RF via MUXA
    * Select load function
    * Select the destination register

  If it is 0,

    * Connect ALUOut to ARF via MUXB
    * Select load function

  * Select the destination register
  * Set RESET as 1

In *T5*:

→ Check DSTREG[2]
→ If it is RF, we connect ALUOut to RF via MUXA, if it is ARF, we connect it via MUXB. We select load function and destination register.
→ Set RESET as 1

2.5.7 MOVH, MOVL

MOVL and MOVH operations assign 8-bit IMMEDIATE to DSRTREG. While MOVH assigns to the HIGH part of DSTREG, MOVL assigns to the LOW part of DSTREG.

OPCODE & SYMBOL	**OPERATION**
0x11 & MOVH	DSTREG[15:8] ← IMMEDIATE (8-bit)
0x14 & MOVL	DSTREG[7:0] ← IMMEDIATE (8-bit)

We can operate this instruction in only one cycle except of fetch and decode cylces.

```
((MOVH|MOVL) &T2)
```

Let's look at the examination of the code.

T2:

→ Select ADDRESS from IR(7-0)

→ Send it RF via MUXA

→ Select register in RF using RxSel module.

  – For MOVH: Select Only Write High function
  – For MOVL: Select Only Write Low function

2.5.8 LDR

In LDR operation, the value stored in memory at the address specified by the AR register is loaded into the register Rx. Rx will be selected by RxSel module.

OPCODE & SYMBOL	OPERATION
0x12 & LDR	Rx ← M[AR] (AR is 16-bit register)

We can operate LDR instruction in 2 cycle except of fetch and decode cycles. So x takes values 2 and 3. The reason why it happens in 2 Clock is because MemOut carries 8 bits and the value we will send is 16 bits. We move 8 bits in each clock cycle and write first the LOW and then the High value to the desired target register.

```
((LDR) & Tx)
```

Let's examine what happens to each clock cycle to perform the LDR operation. .

In *T2*:

→ Select AR in ARF and sent it to Memory via OutD
→ Increase AR in ARF by one.
→ Enable Read in Memory
→ Sent Memory to RF via MUXA using MemOut wire
→ Select Rx register in RF using RxSel module
→ Select Only Load LOW function and load to selected register Rx.

In *T3*:

→Select increased value of AR in ARF and sent it to Memory via OutD
→ Enable Read in Memory
→ Sent Memory to RF via MUXA using MemOut wire
→ Select Rx register in RF using RxSel module
→ Select Only Load HIGH function and load to selected register Rx.

2.5.9 STR

In STR operation, the value stored in Rx is loaded into memory. Rx will be selected by RxSel module.

OPCODE & SYMBOL	OPERATION
0x13 & STR	M[AR] $\leftarrow$ Rx (AR is 16-bit register)

We can operate LDR instruction in 2 cycle except of fetch and decode cycles. So x takes values 2 and 3.

```
((STR) & Tx)
```

Let's examine what happens to each clock cycle to perform the STR operation.

In *T2*:

→ Send Rx to ALU
→ Select A function for ALU and 0 for ALU_WF

In *T3*:

→ Use MUXC to load the LSB of ALUOut to M[AR]
→ Select AR as the address of the place in the memory that will be written
→ Select PC and Load function in RF
→ Select MEM_CS as 0 and MEM_WR as 1.
→ Set RESET 1

2.5.10 BL

In BL operation, the value stored in the memory that has an address of SP value, is loaded into PC.

OPCODE & SYMBOL	OPERATION
0x1F & BL	PC $\leftarrow$ M[SP]

We can operate BL instruction in 4 cycle except of fetch and decode cycles. So x takes values 2 and 5.

`((BL) & Tx)`

Let's examine what happens to each clock cycle to perform the BL operation.

In *T2*:

→ Send SP to Memory in addres line
→ Select MEM_CS as 0 and MEM_WR as 0 to perform read operation.
→ Send MEMOut to ARF via MUXB
→ Select PC and load LSB

In *T3*:

→ Select SP
→ Increment

In *T4*:

→ Send SP to Memory in addres line
→ Select MEM_CS as 0 and MEM_WR as 0 to perform read operation.
→Send MEMOut to ARF via MUXB
→Select PC and load MSB

In *T5*:

→ Select SP
→ Increment

2.5.11 BX

We can complete BX operation in 5 clock cycles. BX assigns the current value of the Program Counter into the memory location pointed to by the Stack Pointer and assigns Rx to PC for branching or jumping to a new instruction address. In summary, BX operation saves the current PC onto the stack and then assigns the value of register Rx to the PC.

OPCODE & SYMBOL	**OPERATION**
0x1E & BX	M[SP] ← PC, PC ← Rx

We set our condition line for the BX Instruction like this:

```
(BX & Tx)
```

Since we can complete BX operation in 5 clocks, x can take values ranging from 2 to 5 inclusive.
Now let's examine what we do in each clock cycle.

In *T2*:

→ Select PC and load it to the BUS by using OutC wire
→ Sent it loaded wire to RF using MUXA
→ Select S3 register in RF
→ Select Load function and load BUS to the S3

In *T3*:

→ Select S3 and sent it to ALU via OutA
→ Pass the value A directly from the ALU without any process.
→ Select SP in ARF and send to Memory via OutD
→ Enable Read in Memory
→ Load the LOW part of the BUS to Memory via MUXC
→ Select SP in ARF and increment the value by one

In *T4*:

→ Select S3 and sent it to ALU via OutA
→ Pass the value A directly from the ALU without any process.
→ Select SP in ARF and send to Memory via OutD
→ Enable Read in Memory
→ Load the HIGH part of the BUS to Memory via MUXC

In *T5*:

→ Select Rx by using RSEL[1:0] signal and send it to ALU via OutA
→ Pass the value A directly from the ALU without any process.
→ Send BUS value to ARF using MUXB
→ Select PC in ARF
→ Select load function for ARF and load to the PC

2.5.12 LDRIM

In LDRIM operation, the value stored in the address that is given in the instruction code is loaded into Rx. Rx will be selected by RxSel module.

OPCODE & SYMBOL	OPERATION
0x20 & LDRIM	Rx ← VALUE (VALUE defined in ADDRESS bits)

We can operate LDRIM instruction in 3 cycle except of fetch and decode cycles. So x takes values 2 and 4.

```
((LDRIM) & Tx)
```

Let's examine what happens to each clock cycle to perform the LDRIM operation.

In *T2*:

→ Connect the IROut[7:0] To AR inside of ARF via MUXB.
→ Select the function that clears the MSB and loads the LSB
→ Select the AR and load.

In *T3*:

→ Select AR to display in address lines
→ Select MEM_CS as 0, MEM_WR as 0.

In *T4*:

→ Connect MEMOut using MUXA to RF
→ Select load function
→ Select destination register.
→ Set RESET as 1

2.5.13 STRIM

In STRIM operation, the value stored in the Rx is loaded into M[AR+OFFSET]. Rx will be selected by RxSel module.

OPCODE & SYMBOL	OPERATION
0x21 & STRIM	M[AR+OFFSET] ← Rx (AR is 16-bit register)

We can operate STRIM instruction in 5 cycle except of fetch and decode cycles. So x takes values 2 and 6.

```
((LDRIM) & Tx)
```

Let's examine what happens to each clock cycle to perform the STRIM operation.

In *T2*:

→ Write IROut[7:0] to S1
→ Use the function that clears the MSB and loads the LSB

In *T3*:

→ Send AR to RF via MUXA
→ Select S2 and load

In *T4*:

→ Send S1 and S2 to ALU
→Select A + B function and set ALu_WF as 0.
→Send ALUOut to AR via MUXB
→Load to AR

In *T5*:

→ Send AR as address to memory
→ Set MEM_CS as 0 and MEM_WR as 1.
→Connect ALUOut to Memory input via MUXC
→ Write the LSB of ALUOut to memory
→Increment AR

In *T6*:

→ Send AR as address to memory
→Set MEM_CS as 0 and MEM_WR as 1.
→ Connect ALUOut to Memory input via MUXC
→Write the MSB of ALUOut to memory
→Set RESET as 1

2.6 CPU SYSTEM

It is the ultimate system that calls for Sequence Counter, Sequence Decoder, Instruction Decoder, ALU System and Control Unit with the necessary parameters.

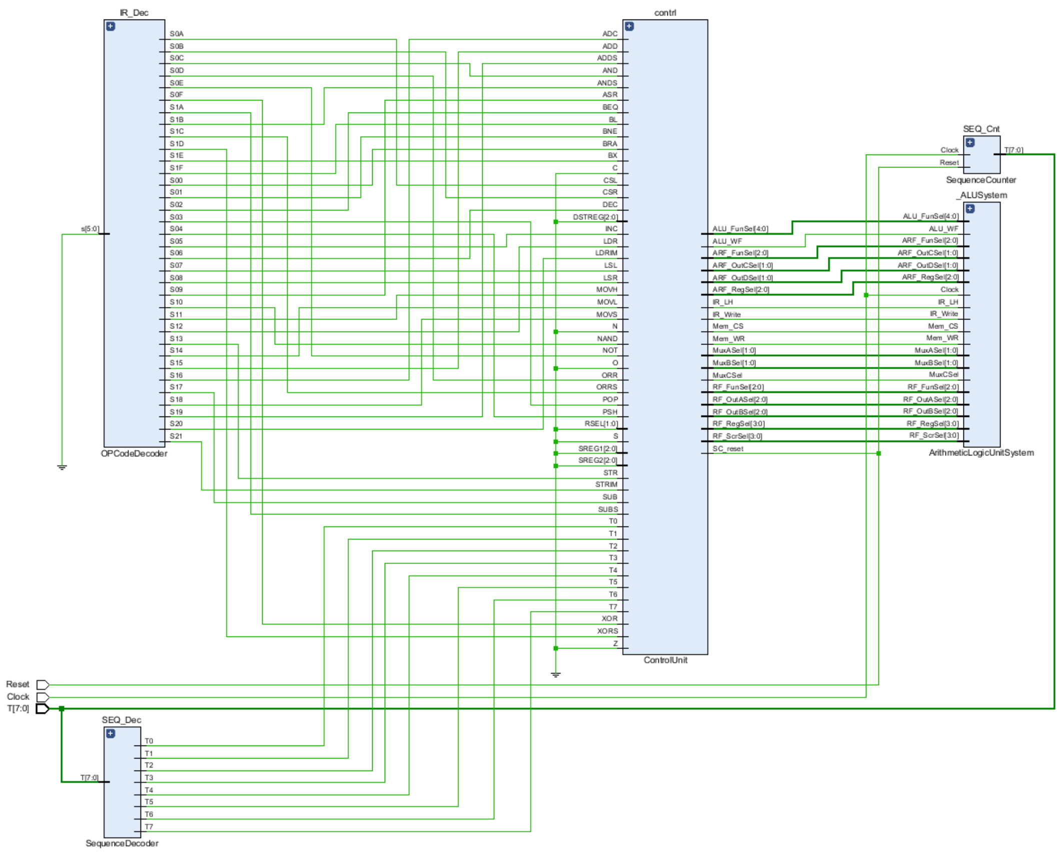


Figure 5: CPU System

2.7 MEMORY

In the last page of homework, there were consecutive lines of instructions that was asked to implement from us. We determined the necessary binary codes of each instruction considering which type is the instruction, how many parts it has, which register is source or destination. After turning them into binary, considering the nature of this program that was asked to us, we determined the memory lines we were going to use and also we did not forget to divide the instructions into half since memory can only hold 8 bits of words. Finally, we wrote our instructions.

3 RESULTS [15 points]

After finishing our codes, by using the Vivado environment and running the simulation, we obtained some outputs in the format of in time Tx what are the registers value.

By writing some basic instruction codes to memory we tried to see if our code was working. Also, we used the signal screen in Vivado to understand when things are changing, which signals has what value and if they are performing according to what they need to do.

Here our results for the project separately:

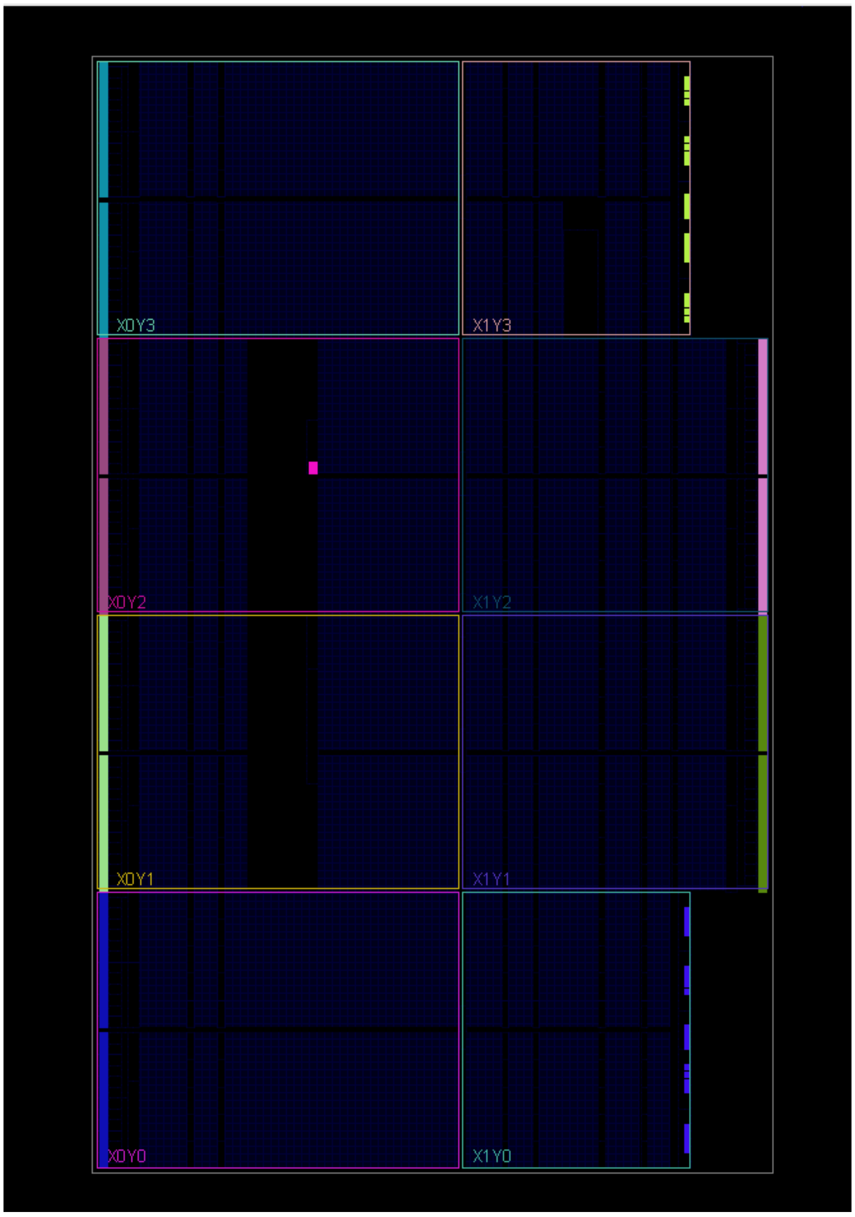


Figure 6: RTL Analysis

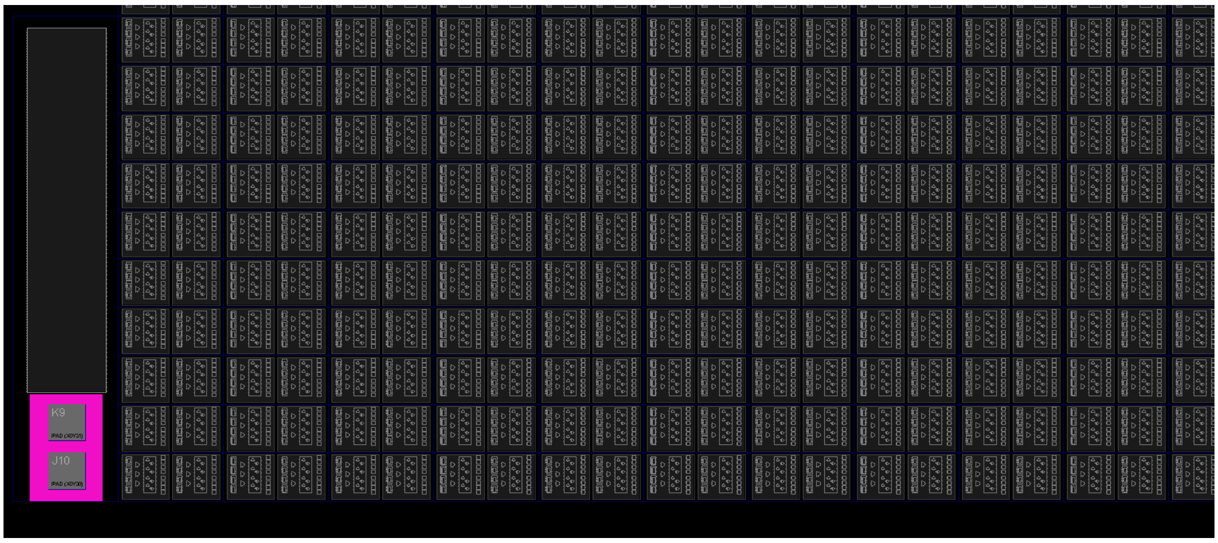


Figure 7: RTL Analysis 2

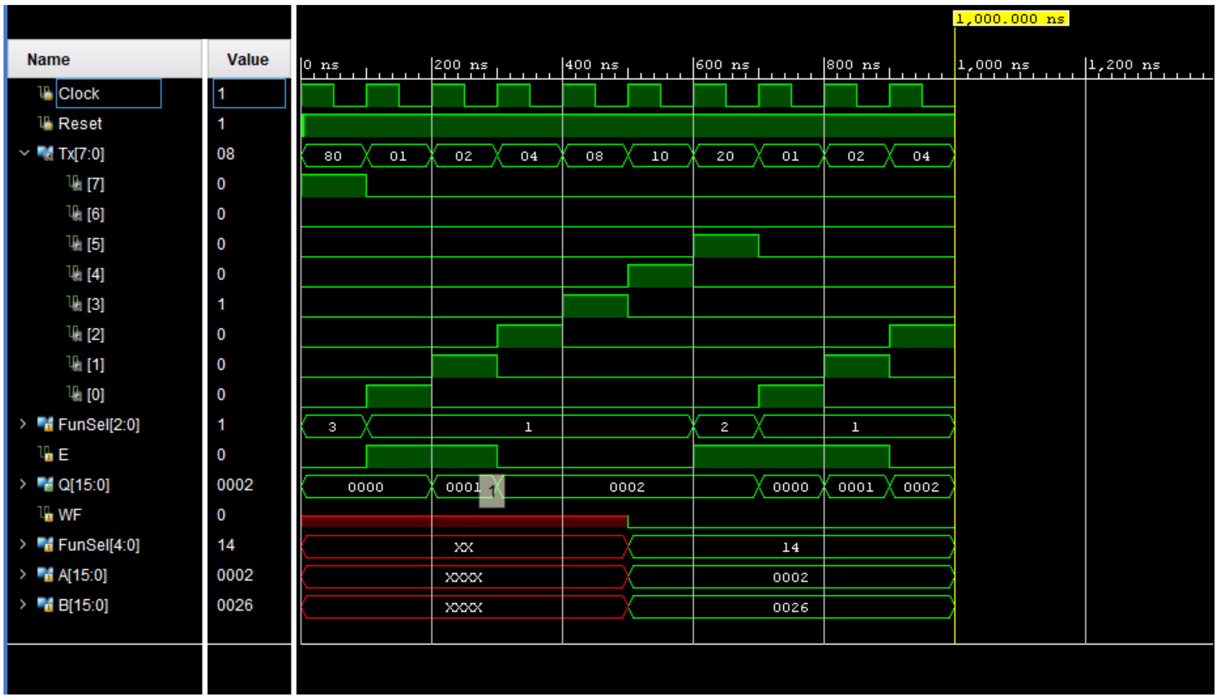
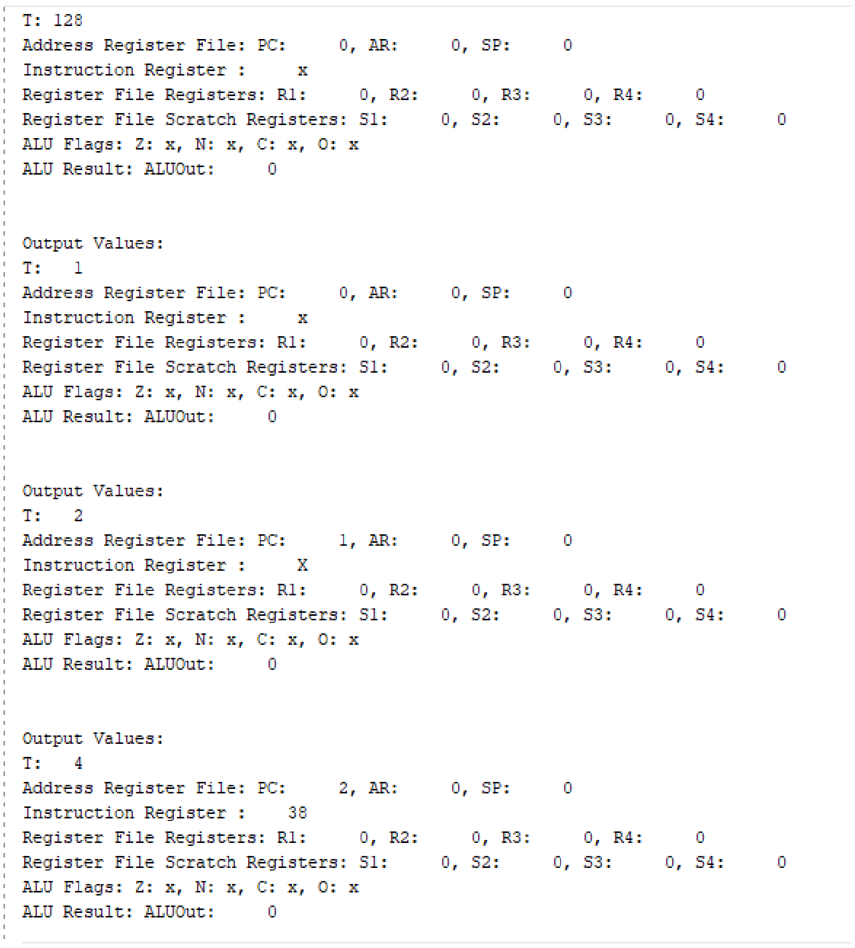


Figure 8: Signals



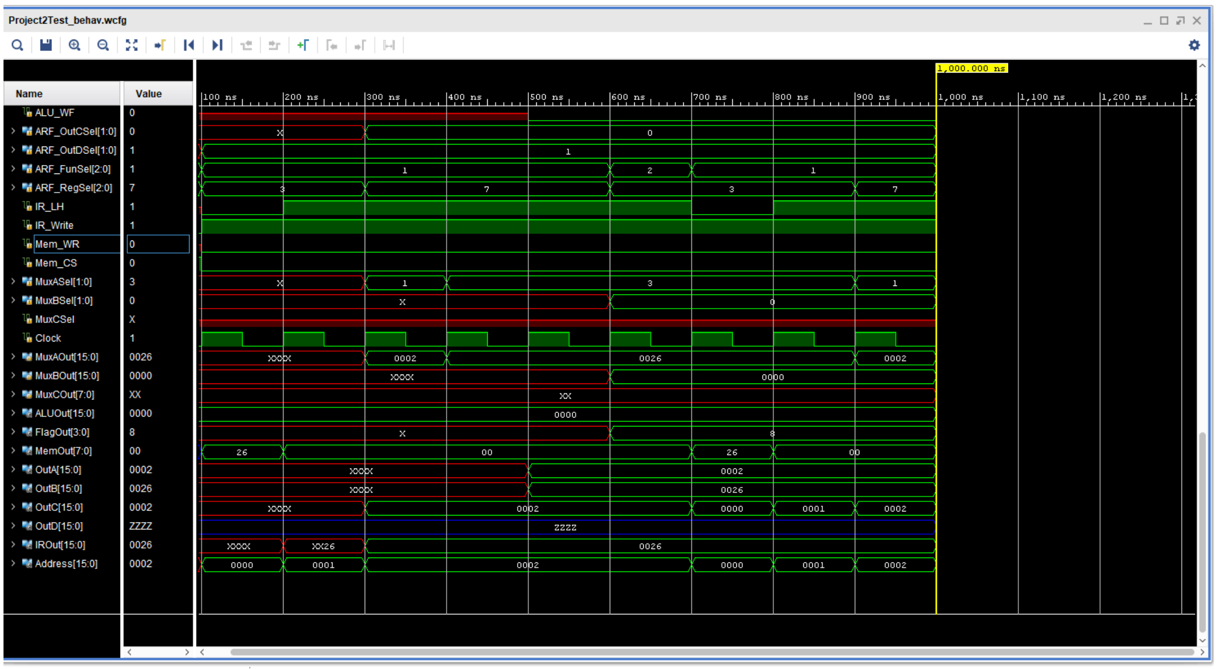
```
T: 128
Address Register File: PC:      0, AR:      0, SP:      0
Instruction Register :      x
Register File Registers: R1:      0, R2:      0, R3:      0, R4:      0
Register File Scratch Registers: S1:      0, S2:      0, S3:      0, S4:      0
ALU Flags: Z: x, N: x, C: x, O: x
ALU Result: ALUOut:      0


Output Values:
T:    1
Address Register File: PC:      0, AR:      0, SP:      0
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:      0
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:      0
Instruction Register :    38
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 9: Results for first 4 clock



Project2Test_behav.wcfg

Name	Value
ALU_WF	0
ARF_OutCSel[1:0]	0
ARF_OutDSel[1:0]	1
ARF_FunSel[2:0]	1
ARF_RegSel[2:0]	7
IR_LH	1
IR_Write	1
Mem_WR	0
Mem_CS	0
MuxASel[1:0]	3
MuxBSel[1:0]	0
MuxCSel	X
Clock	1
MuxAOut[15:0]	0026
MuxBOut[15:0]	0000
MuxCOut[7:0]	XX
ALUOut[15:0]	0000
FlagOut[3:0]	8
MemOut[7:0]	00
OutA[15:0]	0002
OutB[15:0]	0026
OutC[15:0]	0002
OutD[15:0]	ZZZZ
IROut[15:0]	0026
Address[15:0]	0002

Figure 10: Signals for MUX and OUT wires

4 DISCUSSION [25 points]

In our project, we used modules to define the circuit elements. We determined the necessary parameters that these elements take and defined them respectfully. The control unit does not need a Clock to operate but Sequence Counter needs a Clock. So we used the positive edge of clock as **always(posedge Clock)** to keep track of our cycles. We decoded Sequence Counter's output and used different variables that have 0 or 1 values according to the output.

To perform operations, We also needed to have some operation variables, therefore, We designed a specific purpose IR Decoder similar to the Sequence Decoder we designed earlier. We used binary logic expressions for handling the case of returning 1 for only one variable for one instruction. To execute the operations, we needed a control unit. We used T_0 and T_1 for fetch and decode. During T_0 LSB of instruction is written to IR[7:0] from memory and after writing this from M[AR+1] MSB is written to IR[15:7] in T_1.

The Control Unit uses these decodings the decide which operation and which clock cycle to perform. We used multiple if cases to be sure that an operation is performed in the right time and when the operation is finished, we set the RESET as 1 to assign 1 to the clock. Assigning 1 to clock starts the system from beginning and since PC shows the next instruction to perform, it continues by uploading the memory contents into the instruction register for different instructions and executes it.

For different operations, different clock cycles are needed. Based on which register system is involved, if there is an operation performed on ALU or the way of information transformation, clocks can be extended or shortened. Considering the path of information transformation, we tried to optimize our code in the best way that there is no extra clock cycle. In every clock cycle, we determined which system is used and which signals are needed and assigned them to the proper variables.

General CPU System has a module calling inside, so the necessary parameters such that opcode, operation variables, and Tx are always updated whenever there is a change in the given parameters. Even though it is possible to change inputs anytime, since registers work with a clock, operations are not affected. The modules are connected in a way that every input and output is identified and wires are used to connect them. We were careful about input and output parameters, since the order of parameters matter.

5 CONCLUSION [10 points]

Throughout this assignment, we learned the general working logic and algorithms of Instructions and the operations they can perform according to clock circles. In this project, we considered the Register and ALU parts that we implemented in Project 1 as a whole and coded the working logic of various Instructions in Verilog to make a functional CPU.

There were a few challenges we encountered and a few things we learned while doing the project. For example, we learned the capacity of the operations we can perform in a clock cycle. At the beginning of the project, we encountered various logical errors because we performed operations that needed to be done on more than one clock, especially those that required ALU processing, on a single clock.

In addition, being so intertwined with the BUS gave us information about the communication of hardware parts with each other. We learned about the signals we need to give to the MUX to send the value loaded on the BUS to another Register or Memory, or the function selections we need to use to prevent unwanted signals.

Since Project 1, each building we built was dependent on the previous one and built upon it. We started from the most basic IR and continued up to the ALU design. In this project, we took the necessary steps to turn the ALU system we installed into a workable CPU. We established the sequence counter, sequence decoder, ir decoder, destination and source register and control unit structure so that the CPU we installed can perform each instruction we want. We established a separate logic system for each functional instruction and built the CPU as a whole.