# Design Project 4
# Simple Computer Assembly Language Programming

Richard Ngo

TABLE OF CONTENTS

Pseudo code:

PRE PROGRAM CONDITIONS:

Since BRZ can only move 32 addresses, jump locations will be calculated and stored in memory at the beginning of the code in order to keep the actual algorithms of the program as compact as possible

Also, since M and N only need to be checked once, this is done at the beginning, before the program loop as well

If M+N is greater than 0x20 or 32, assuming that N is at least 1, it means that OP A will eventually enter OP B property, and thus, N can only be equal to 0x20 – M in order to keep this from happening.

Code will jump to end if M by itself is greater than F1, less than 10 or N by itself is equal to 0

START OF PROGRAM:

R4 is used to load immediate values, useful for jumps and branches

R1 stores the mask 0000000000001111 in order to focus on single digit

R7 is loaded with 1 and used as counter

R2 is loaded with op A from address at memory 0

CHECK ALGORITHM:

R3 is Mask of R2 (R2 and with R1)

If 9 – R3 is negative, store FFFF in memory 32+ from OPA address

Shift R2 right 4 times to move on to next digit

If R2 is empty, it means that all digits have been checked, and either OP B is loaded into R2 and CHECK ALGORITHM is looped if R7 is 1, or program jumps to ADD ALGORITHM if R7 is 0. Decrement R7 before jump.

ADD ALGORITHM:

R2 and R7 are loaded with op A and op B

R6 loaded with 6 to add 6

R0 stores which digits that 6 must be added to, initially loaded with 0

R3 is R2(A) masked with R1

R4 is R7(B) masked with R1

If R3 + R4 is greater than 9, add R0 with R6

Shift R2 and R7 right 4 times to move to next digit

Shift R6 left 4 times in order to keep track of where 6 should be added

If R6 is 0, it means that all digits has been checked and the result is equal to R0 + op A + op B

Store result in address in memory 32+ from OPA address

If N is 0, jump to end program

Decrement N and restore in memory

Increment M to change added operations and restore in memory

Jump back to start of program

Assembly Source Program:

```
ldi r0, 3
shl r0, r0
shl r0, r0
inc r0, r0
shl r0, r0
jmp r0
ldi r0, 5
shl r0,r0
shl r0,r0
ld r1, r0  //r1<-M[0x14]
adi r0, r0, 4
shl r0, r0
adi r0, r0, 3
ld r2, r0  //r2<-M[0x33]
inc r0, r0
ld r3, r0  //r3<-M[0x34]
inc r0, r0
ld r4, r0  //r4<-M[0x35]
inc r0, r0
ld r5, r0  //r5<-M[0x36]
```

```
inc r0, r0
ld r6, r0  //r6<-M[0x37]
inc r0, r0
ld r7, r0  //r7<-M[0x38]
ldi r0, 0
brz r0, 0
ldi r0, 5
shl r0, r0
shl r0, r0
shl r0, r0
shl r0, r0
adi r0, r0, 7 //1010111 = 87
ldi r1, 2
st r1, r0// start of loop in memory 2
ldi r0, 6
shl r0, r0
shl r0, r0
shl r0, r0
shl r0, r0
shl r0, r0//11000000 = 192
ldi r1, 3
st r1, r0// error jump in memory 3
ldi r0, 4
shl r0, r0
shl r0, r0
shl r0, r0
adi r0, r0, 5
shl r0, r0
shl r0, r0
```

```
shl r0, r0
shl r0, r0
shl r0, r0
adi r0, r0, 5
shl r0, r0
shl r0, r0
adi r0, r0, 2//PID  = 1296 = 1001010010110
ldi r1, 5
shl r1, r1
shl r1, r1 // 0x14 == 10100
st r1, r0 // overwrite 0x14 with PID
ldi r1, 4
shl r1, r1
mova r3, r1//for later, r3 = 8
shl r1, r1//r1 = 16
shl r2, r1//r2 = 32
dec r1, r1//r1 = 15
ldi r0, 0
ld r0, r0 // r0 = M
sub r1, r1, r0
sub r2, r0, r2
brn r1, 4 //If 15 - M < 0, M is valid and skip jump to end
brn r2, 3 //If M - 32 < 0, M is valid and skip jump to end
ldi r0, 6
jmp r0 //Invalid M or N = 0, jump to end
ldi r1, 1
ld r1, r1//r1 = N
brz r1, -6 //If N = 0, No additions are needed and jump to end
add r2, r1, r0//r2 = M + N
```

```
shl r3, r3

shl r3, r3

adi r3, r3, 1//r3 = 33

sub r2, r2, r3

brn r2, 5//if (M+N) - 33 < 0, OP A doesn't go out of bounds and is valid, goes to main loop

dec r3, r3 // r3 = 32

sub r1, r3, r0

ldi r0, 1

st r0, r1//if (M+N) - 33 >= 0, OP A does go out of bounds and N is adjusted to (20-M) keep this from happening

ldi r2, 0//start of program loop

ld r2, r2

ld r2, r2//r2 = opA

ldi r7, 0//r7 = count = 1

ldi r1, 7

shl r1, r1

adi r1, r1, 1//r1 = ...01111, mask

and r3, r1, r2//CHECK ALGORITHM

ldi r4, 7

adi r4, r4, 2//r4 = 9

sub r4, r4, r3

brn r4, 8//if 9 - operand is negative, operand is not valid bcd and branch to jump to error algorithm

shr r2, r2

shr r2, r2

shr r2, r2

shr r2, r2//change digit

brz r2, 6//switch jump//if r2 is empty from shifting, branch to op switch algorithm

ldi r4, 0

brz r4, -11//start CHECK ALGORITHM again
```

```
ldi r0, 3 //jump to error
ld r0, r0
jmp r0
brz r7, 11//op switch algorithm, if count == zero, proceed to ADD ALGORITHM
dec r7, r7 //count is decremented
ldi r0, 4
shl r0, r0
shl r0, r0
ldi r4, 0
ld r4, r4
add r4, r4, r0 // r4 = opA memory address + 16 = opB memory address
ld r2, r4 // switch r2 from opA to opB
ldi r4, 0
brz r4, -14 //branch to start CHECK ALGORITHM again
ldi r4, 0 //ADD ALGORITHM
ld r4, r4
ldi r0, 4
shl r0, r0
shl r0, r0
add r4, r4, r0//r4 = opA memory address + 16 = opB memory address
ld r2, r4//r4 = r2 = opB
ldi r7, 0
ld r7, r7
ld r7, r7//r7 = r7 = opA
ldi r0, 0//r0 holds r6s to be added
ldi r6, 6//r6 holds what hex digit to add 6 to
and r3, r1, r2//add loop, r3 is masked op B
and r4, r1, r7//r4 is masked op A
add r3, r3, r4 // digits are added
```

```
ldi r4, 7
adi r4, r4, 2//r4 = 9
sub r4, r4, r3
brn r4, 16//if resultant digit is greater or equal to 10, branch to add 6 algorithm
shr r2, r2
shr r2, r2
shr r2, r2
shr r2, r2//change op b digit
shr r7, r7
shr r7, r7
shr r7, r7
shr r7, r7//change op a digit
shl r6, r6
shl r6, r6
shl r6, r6
shl r6, r6//change digit to add 6 to
brz r6, 10//if r6 == 0, r6 has been shifted empty and all BCD has been checked, branch to end of program loop
ldi r4, 0
brz r4, -21//reset add loop
add r0, r0, r6 //add 6 algorithm
ldi r4, 4
shl r4, r4
shl r4, r4
add r2, r4, r2//carry over 1 (16) to next digit
ldi r4, 0
brz r4, -21//branch back to register shifts in loop
ldi r1, 0//end of program loop
ld r1, r1
```

```
ld r1, r1//r1 = op A
ldi r3, 4
shl r3, r3
shl r3, r3//r3 = 16
ldi r4, 0
ld r4, r4
add r4, r4, r3
ld r4, r4//r4 = op b
add r1, r1, r4//r1 = opA + opB = hex
add r0, r0, r1//hex has 6s added necessary digits to change to BCD
ldi r1, 0
ld r1, r1
shl r3, r3//r3 = 32
add r3, r3, r1//r3 = opA memory address + 32 = result memory address
st r3, r0//store BCD result into memory
ldi r0, 0
ld r0, r0
inc r0, r0
ldi r1, 0
st r1, r0//increment M and store
ldi r0, 1
ld r0, r0
dec r0, r0
ldi r1, 1
st r1, r0//decrement N and store
brz r0, 28//if N == 0, branch to end
ldi r0, 2
ld r0, r0//r0 = instruction address of beginning of program loop
jmp r0//repeat loop
```

```
ldi r3, 4//error algorithm
shl r3, r3
shl r3, r3
shl r3, r3
ldi r1, 0
ld r1, r1
add r3, r3, r1//r3 = op A memory address + 32 = result memory address
ldi r0, 0
not r0, r0//r0 = FFFF
st r3, r0//store FFFF in result memory address
ldi r0, 0
ld r0, r0
inc r0, r0
ldi r1, 0
st r1, r0//increment M and store
ldi r0, 1
ld r0, r0
dec r0, r0
ldi r1, 1
st r1, r0//decrement N and store
brz r0, 4//if N == 0, branch to end
ldi r0, 2 //r0 = instruction address of beginning of program loop
ld r0, r0//repeat loop
jmp r0
ldi r0, 6 //end program
jmp r0
```

Program Tests:

If op A and B are valid:

0333 + 0666 = 0999

If op A and B are valid, but their result is invalid:

1296(PID) + 9999 = AC2F-> 11295 (1295 since 4th digit will be excluded)

If op A is an invalid BCD:

11A1 + 1111 = FFFF

If op B is an invalid BCD:

1111 + 1B11 = FFFF

If the first op A address is outside of 10 to 1F range:

0: 0009

1: 0003

and

0: 0020

1: 0001

If the first op A address is inside of 10 to 1F range, but N value causes it to go out of range:

0: 001D

1: 0007

Simulation and Results:

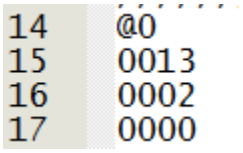


Figure 1: Testing for if op A and B are valid and if op A and B are valid, but their result is invalid

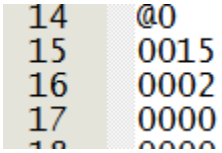


Figure 2: Testing for when op A or op B has an invalid BCD

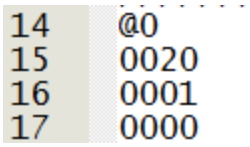


Figure 3: Testing for if the first op A address is greater than 1F

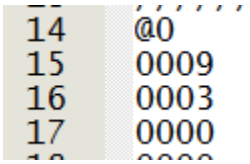


Figure 4: Testing for if the first op A address is less than 0x10

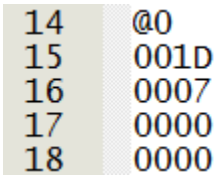


Figure 5: Testing for if the first op A address is inside of 0x10 to 0x1F range, but N value causes it to go out of range

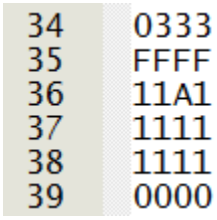


Figure 6: Test values loaded in for addresses 13-18

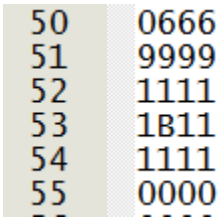


Figure 7: Test values loaded in for addresses 23-28

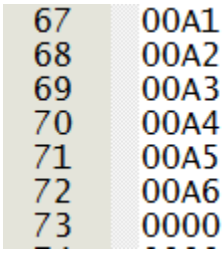


Figure 8: Result values edited to see which have been overwritten (addresses 33-38)

```
10  ld r1, r0  //r1<-M[0x14]
11  adi r0, r0, 4
12  shl r0, r0
13  adi r0, r0, 3
14  ld r2, r0  //r2<-M[0x33]
15  inc r0, r0
16  ld r3, r0  //r3<-M[0x34]
17  inc r0, r0
18  ld r4, r0  //r4<-M[0x35]
19  inc r0, r0
20  ld r5, r0  //r5<-M[0x36]
21  inc r0, r0
22  ld r6, r0  //r6<-M[0x37]
23  inc r0, r0
24  ld r7, r0  //r7<-M[0x38]
```

Figure 9: Contents of registers

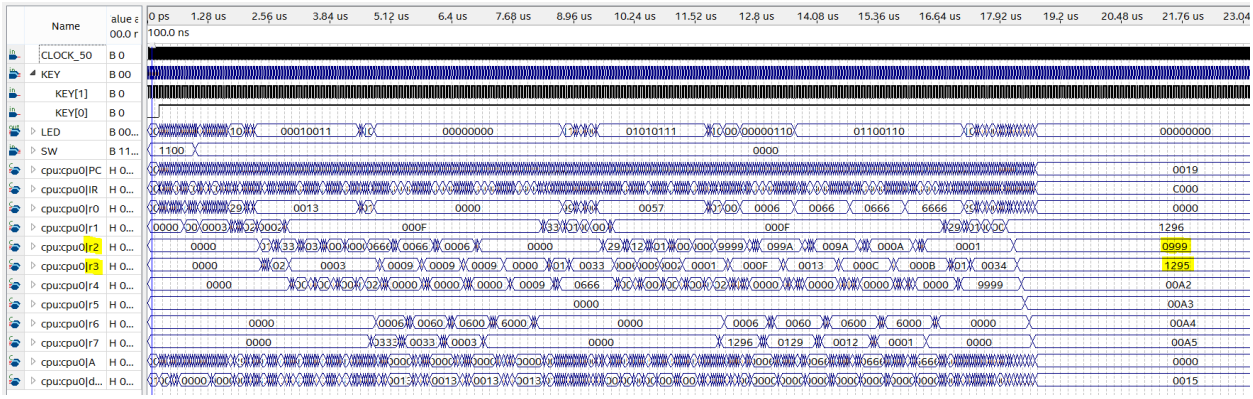


Figure 10: Simulation for valid BCD additions

First in order to test for when op A and B are valid and when op A and B are valid, but their result is invalid, M = 0013 and N = 2 is written into the instruction (see Figure 1) in order to correspond to the necessary operation's locations (see Figures 6 and 7). Since memory location 0x33 is displayed by R2, and 0x34 is displayed by R3 (see Figure 9), R2 should end up with the result 0999, and R3 should end up with the result 1295. The simulated results were accurate with the predictions. (see Figure 10)

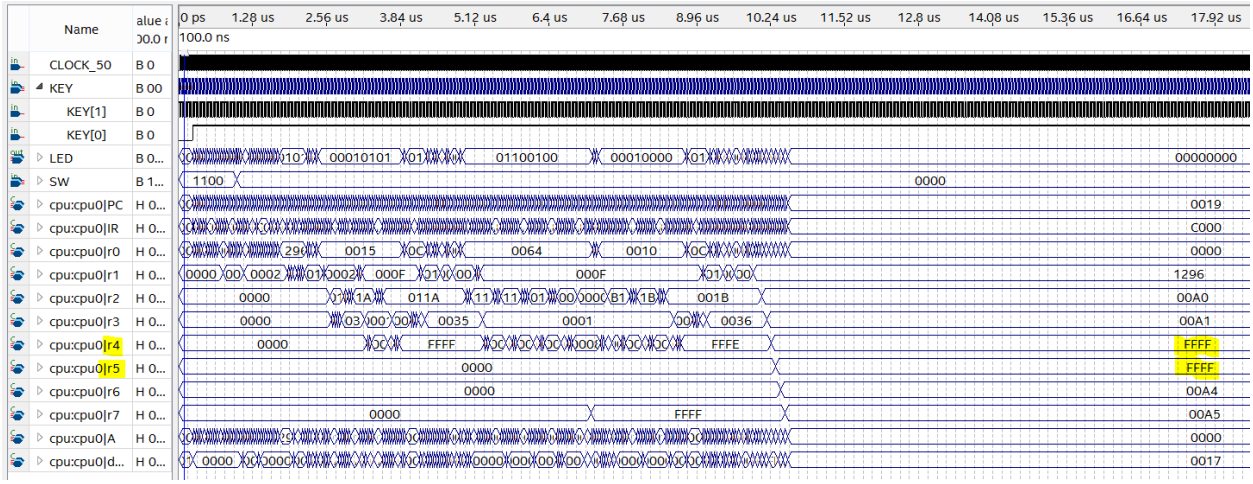


Figure 11: Simulation for invalid BCD additions

In order to test for when op A and B are invalid BCD's and are unable to be added, M = 0015 and N = 2 is written into the instruction (see Figure 2) in order to correspond to the necessary operation's locations (see Figures 6 and 7). Since memory location 0x35 is displayed by R4, and 0x36 is displayed by R5 (see Figure 9), R4 should end up with the result FFFF, and R5 should end up with the result FFFF, since FFFF represents an invalid result. The simulated results were accurate with the predictions. (see Figure 11)

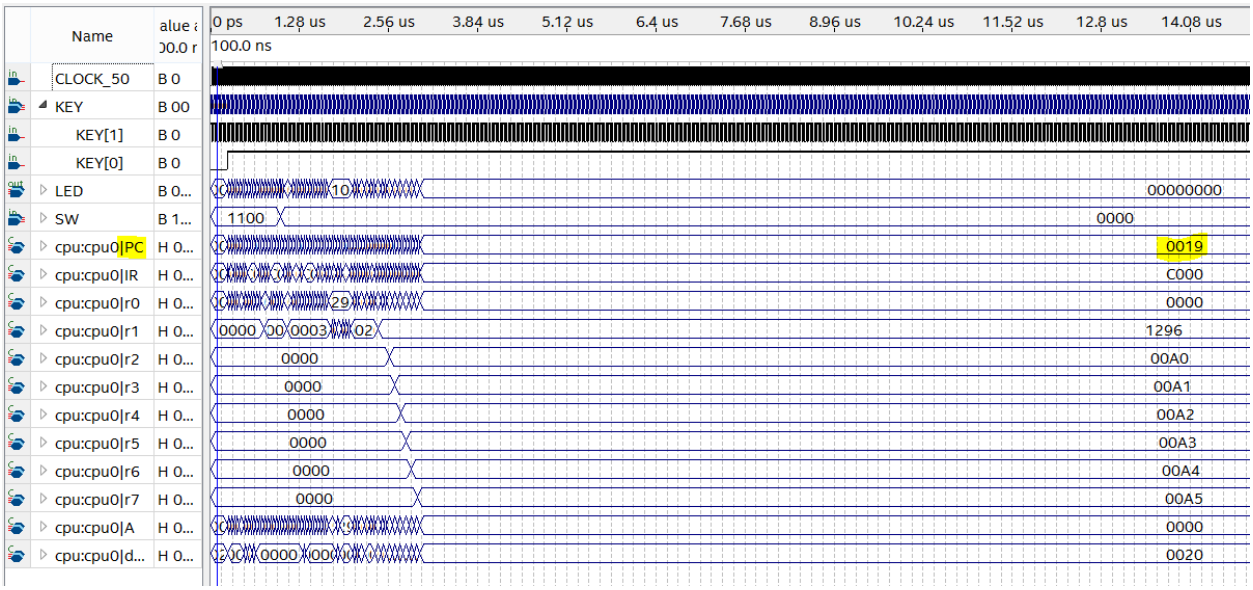


Figure 12: Simulation for invalid M

In order to test for when M points to an invalid A that is outside of the 10 to 1F range, M = 0020 and N = 1, and M = 0009 and N = 3 is written into the instructions (see Figures 3 and 4). Both of these results should cause the simulation to end short without overwriting any of the memory (see Figure 8), and cause PC to remain at address 0x19 (25), since that is where the instruction points to at the end of the program. The simulated results were accurate with the predictions. (see Figure 12)

```
83 brn r2, 5//if (M+N) - 33 < 0, OP A doesn't go out of bounds and is valid, goes to main loop
84 dec r3, r3 // r3 = 32
85 sub r1, r3, r0
86 ldi r0, 1
87 st r0, r1//if (M+N) - 33 >= 0, OP A does go out of bounds and N is adjusted to (20-M) keep this from happening
```

Figure 13: Instruction address where N is updated and stored in R1

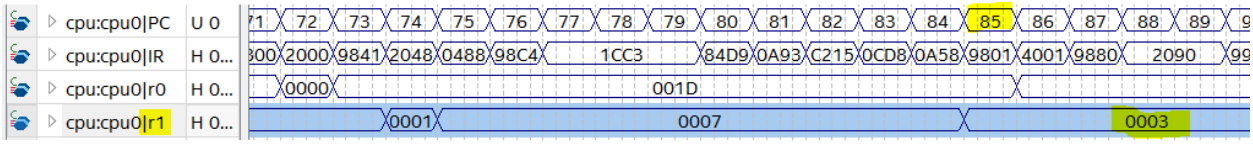


Figure 14: Simulation for when N eventually causes M to be invalid

In order to test for when N causes M to go out of op A range, M = 001D and N = 7, is written into the instructions (see Figure 5). Since the result of this operation is to be stored at 003D and beyond, which isn't covered by our observed registers (see Figure 9), this can be tested instead by viewing what N is being updated. In the code (see Figure 13), R1 is equal to N after it has been changed to 0x20-M, which in this case should be 3, starting when PC is equal to 85. The simulated results were accurate with the predictions. (see Figure 14)

All simulations ran as expected, meaning that the program ran correctly.

Conclusion:

The program was a success, as it was able to pass all tests and simulate correctly. There were a few problems that were run into during the process of creating this program, like making the code compact enough to be branched through conveniently, though this was able to be overcome by keeping as many unnecessary calculations out of the algorithm as possible by doing them at the start. Through this project, more familiarity with the assembly language as well as how to add BCD binary numbers was acquired. Simulated data sets are included below. (see Figures 15 and 16)

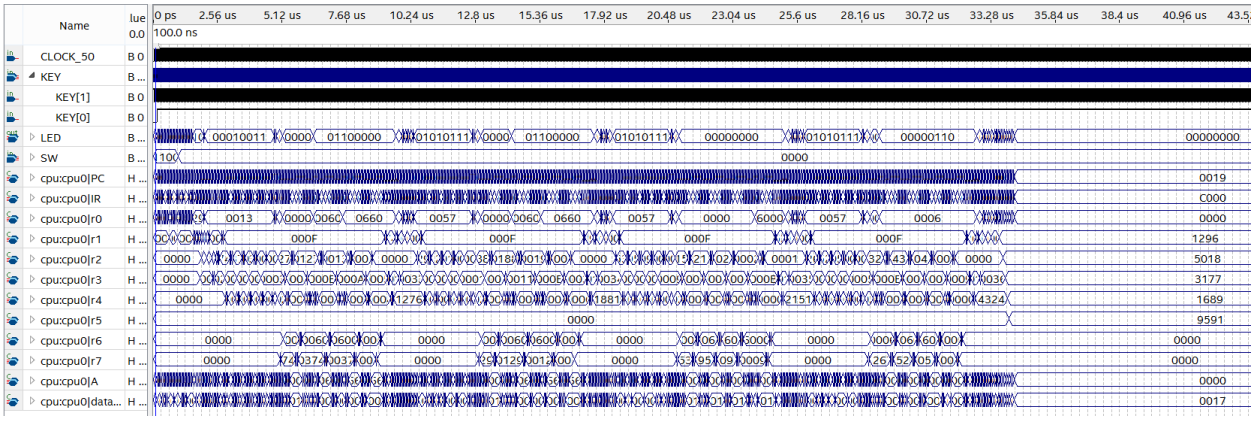


Figure 15: Test 1 Results

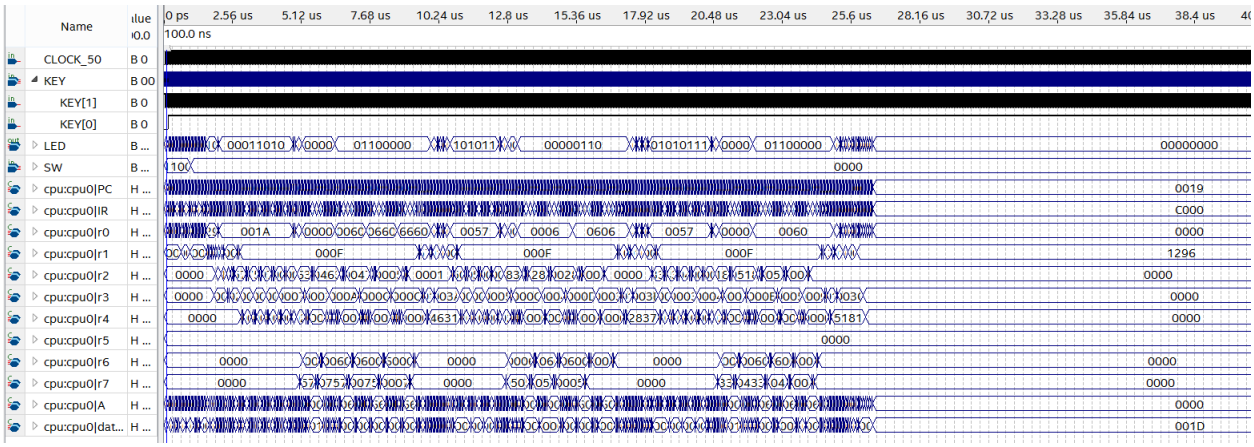


Figure 16: Test 2 Results