ECE 2504 Design Project 4 (Spring 2015)
Cover Sheet

Student Name: Bowei Zhao____

Pledge: I have neither given nor received unauthorized assistance on this assignment.

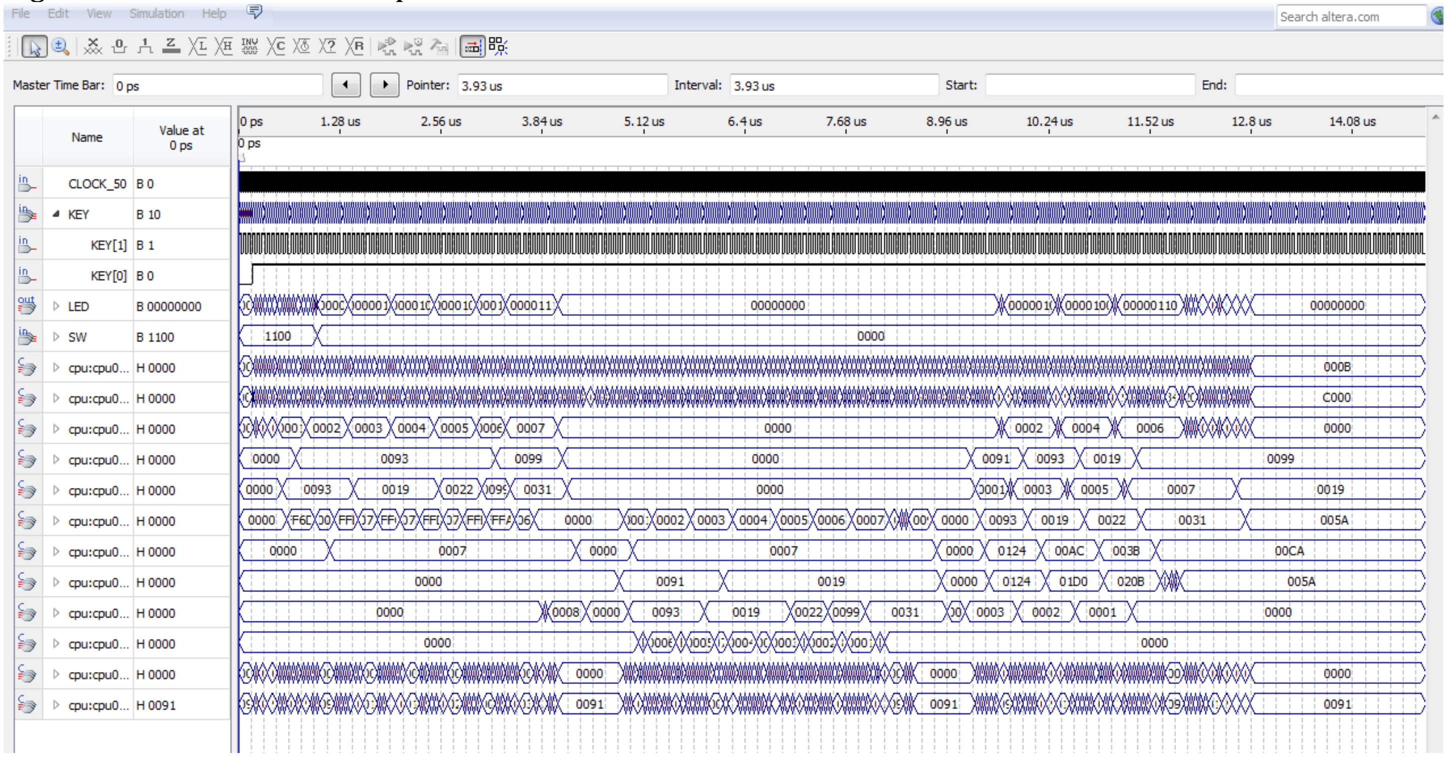
Signed: Bowei Zhao____

Project Grading (to be completed by GTA or instructor)
The design project will be graded on the basis of 100 points, allocated as shown below.

Project Item	**Value**	**Points**
Submitted Materials		
• Completed cover sheet with name, four ID digits, and signed pledge	5	
• Simulation waveforms demonstrating proper execution of the program	20	
• Commented, formatted assembly source code file	15	
• Instruction.txt file	5	
• Data.txt file	5	
Validation:		
• Functionality of Program	50	
TOTAL POINTS	100	

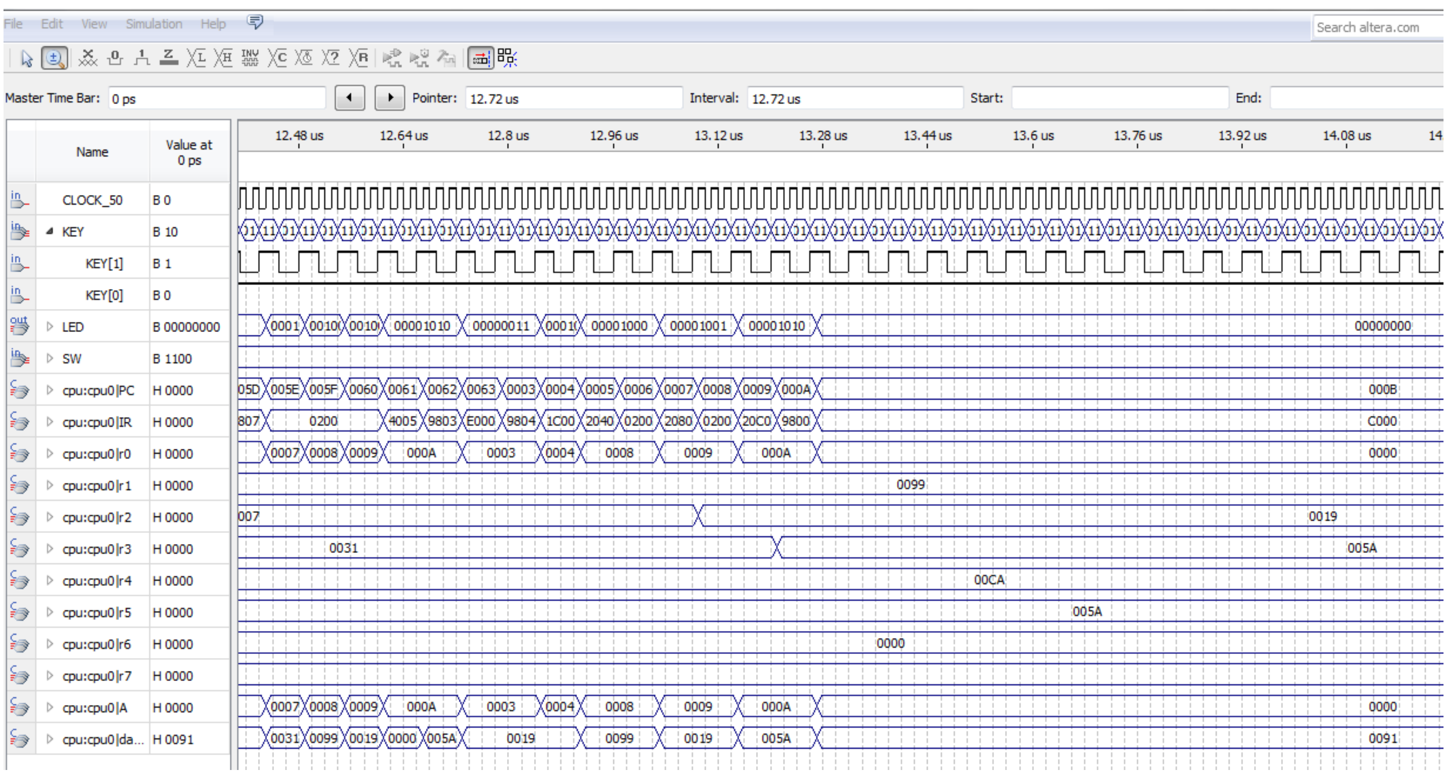
Grader Comments:

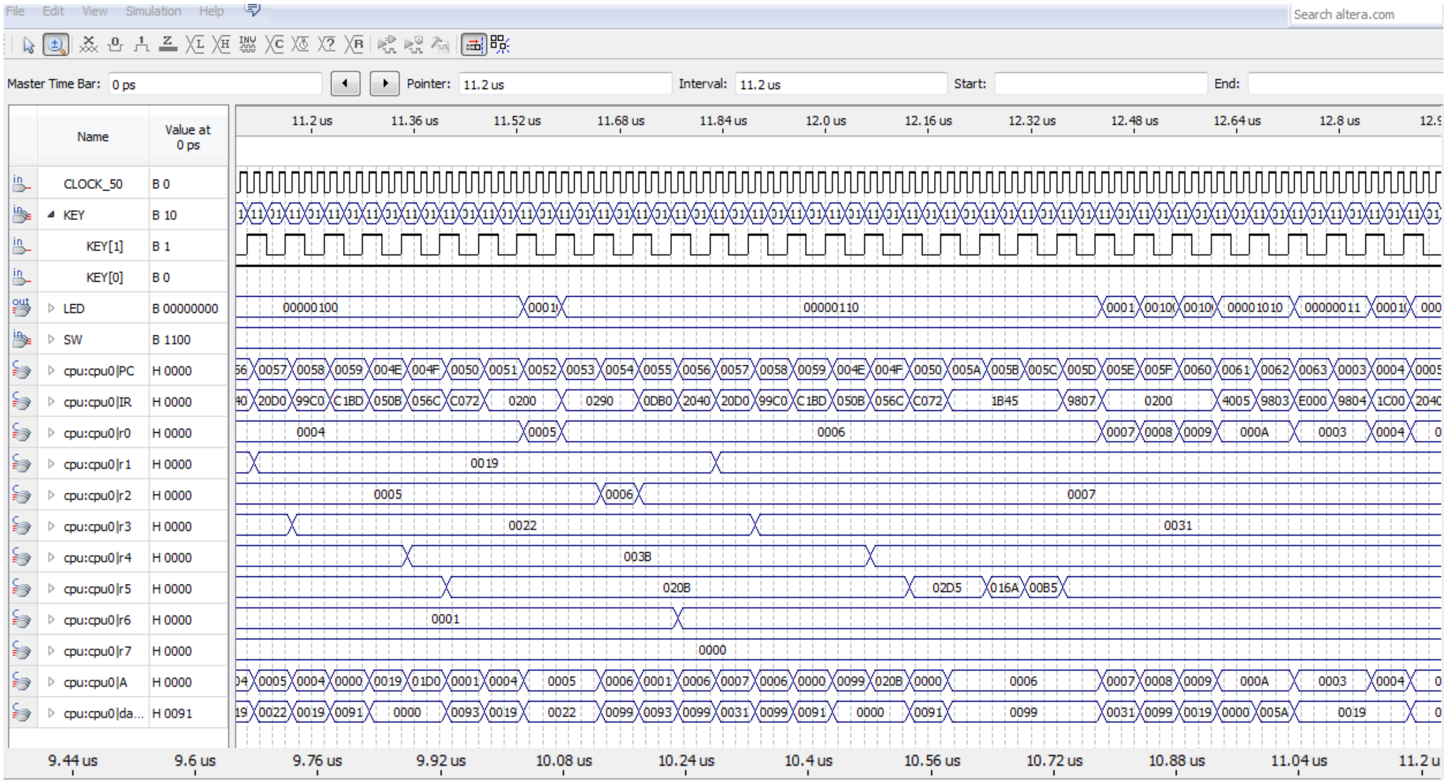
Figure 1: WaveForm for Sample Simulation Run

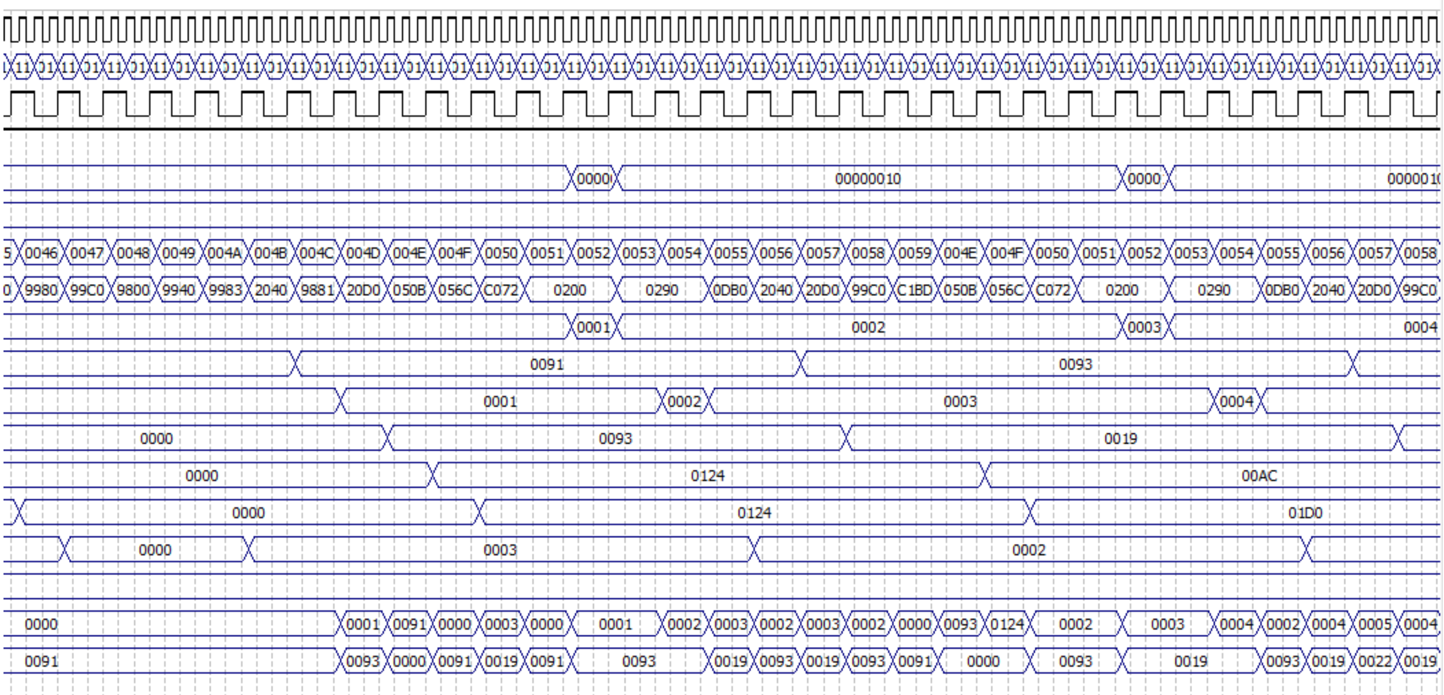


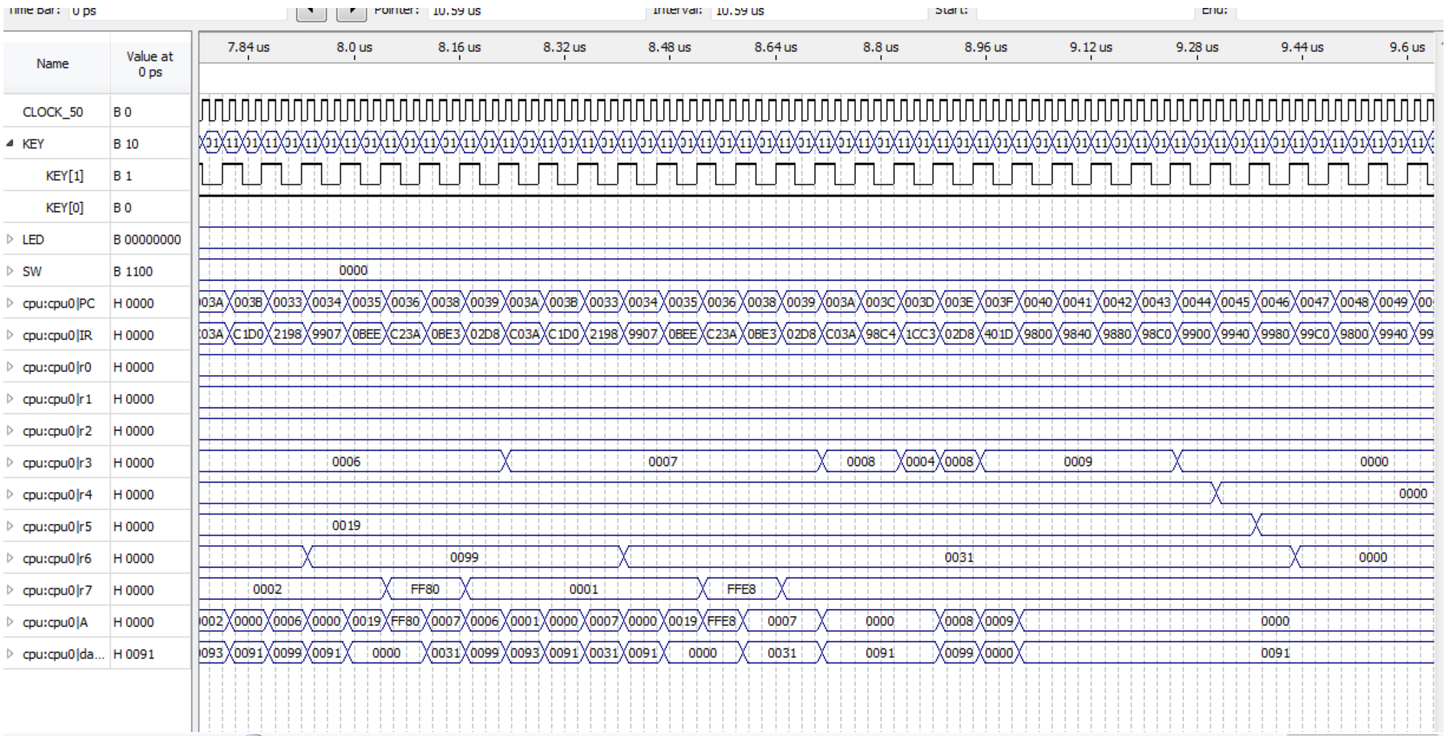
Featured above is my simulation run with my PID and storing the max in R1, min in R2, R3 in average.

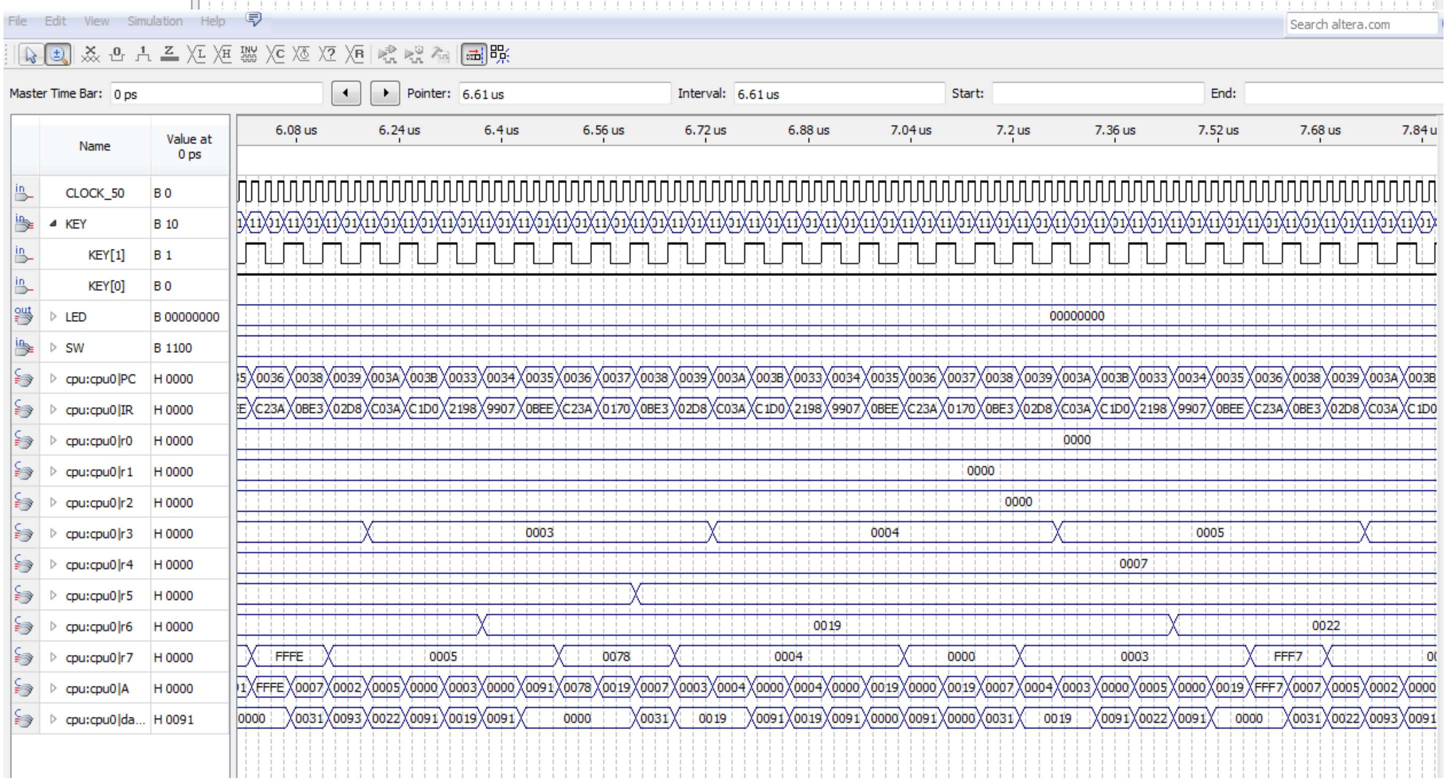
Below are the entire Quartus Simulation showing the IR and PC values in more detail. They start from the very end of the 15us to 0ns from larger to smaller. They are in order, but feature slight overlap.

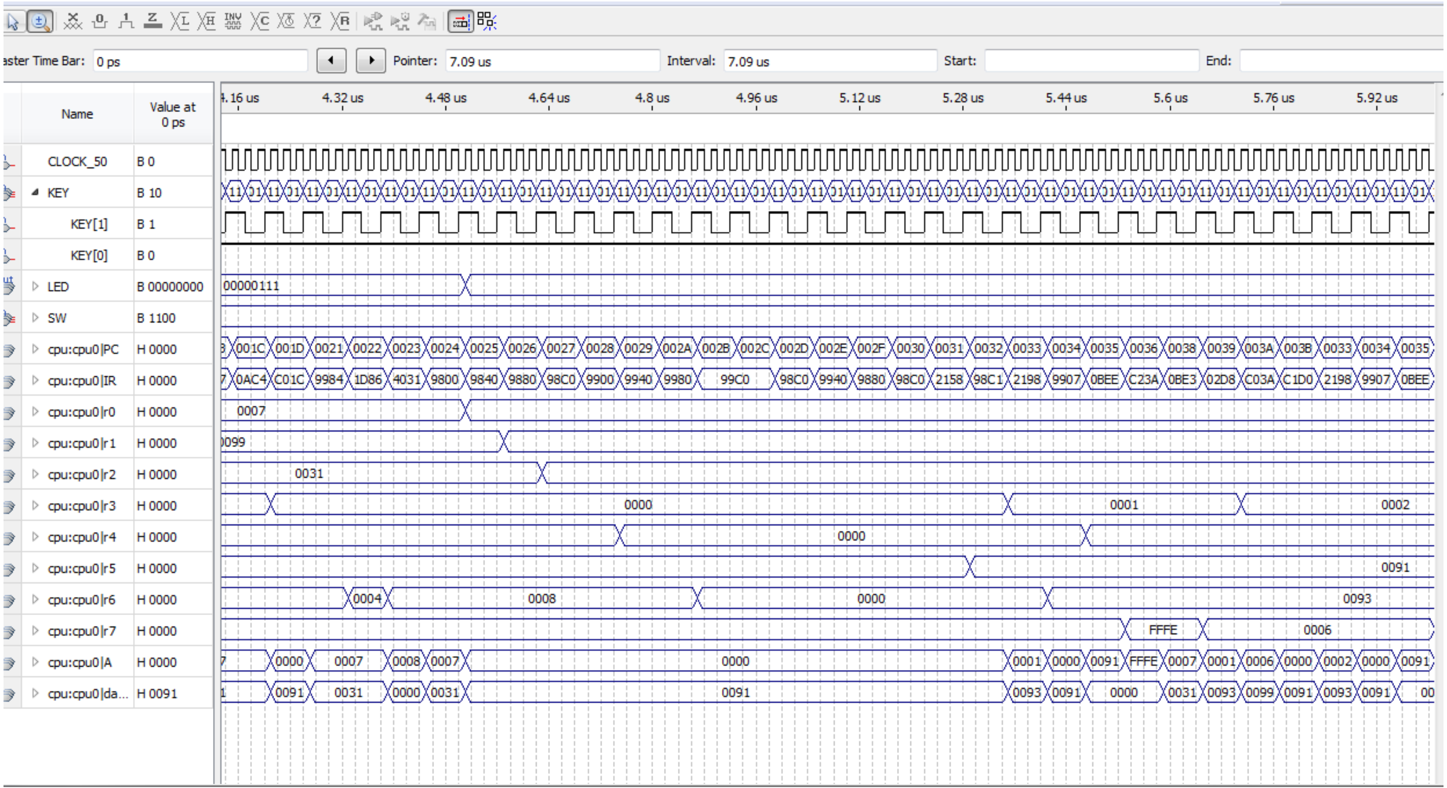


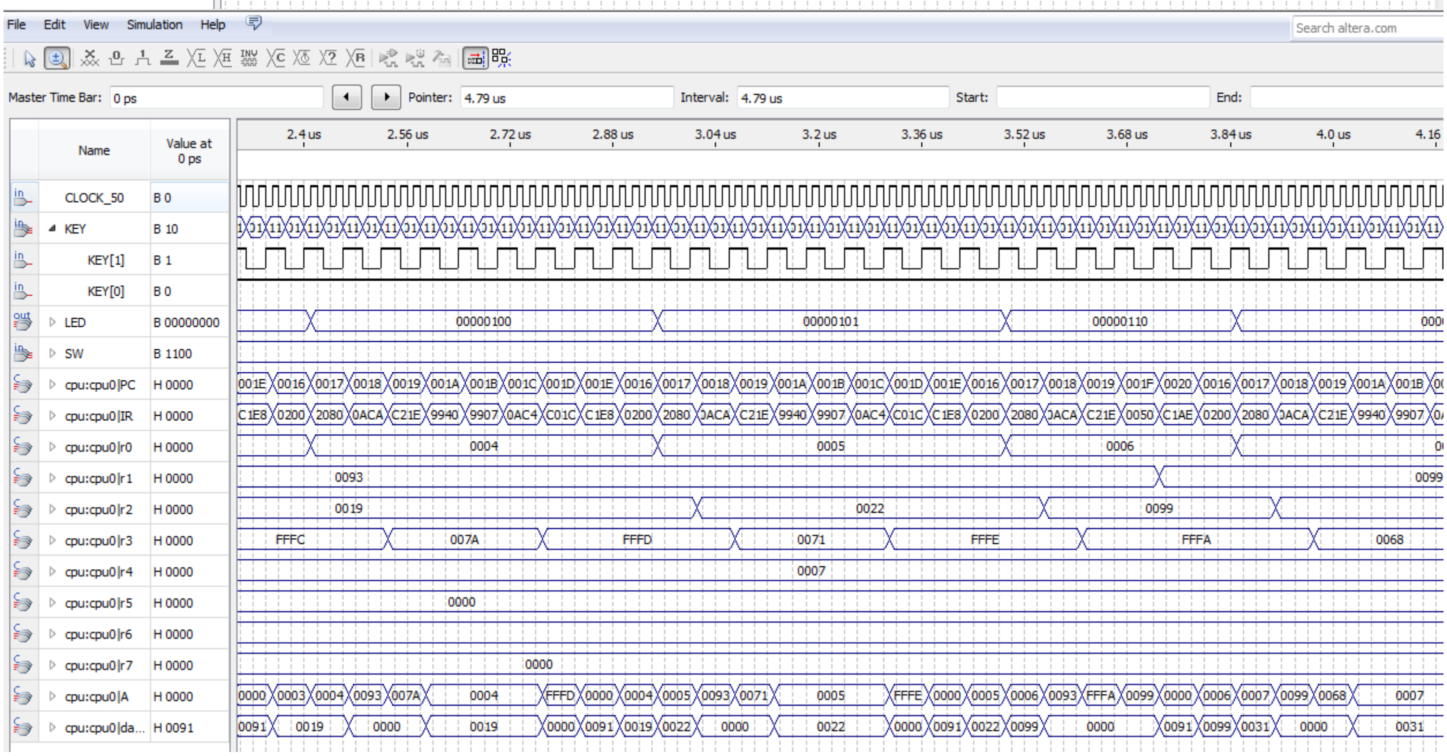

File Edit View Simulation Help
Search altera.com
Master Time Bar: 0 ps
Pointer: 11.2 us
Interval: 11.2 us
Start:
End:
Name
Value at 0 ps
11.2 us
11.36 us
11.52 us
11.68 us
11.84 us
12.0 us
12.16 us
12.32 us
12.48 us
12.64 us
12.8 us
CLOCK_50 B 0
KEY B 10
KEY[1] B 1
KEY[0] B 0
LED B 00000000
SW B 1100
cpu:cpu0|PC H 0000
cpu:cpu0|IR H 0000
cpu:cpu0|r0 H 0000
cpu:cpu0|r1 H 0000
cpu:cpu0|r2 H 0000
cpu:cpu0|r3 H 0000
cpu:cpu0|r4 H 0000
cpu:cpu0|r5 H 0000
cpu:cpu0|r6 H 0000
cpu:cpu0|r7 H 0000
cpu:cpu0|A H 0000
cpu:cpu0|da... H 0091
9.44 us
9.6 us
9.76 us
9.92 us
10.08 us
10.24 us
10.4 us
10.56 us
10.72 us
10.88 us
11.04 us

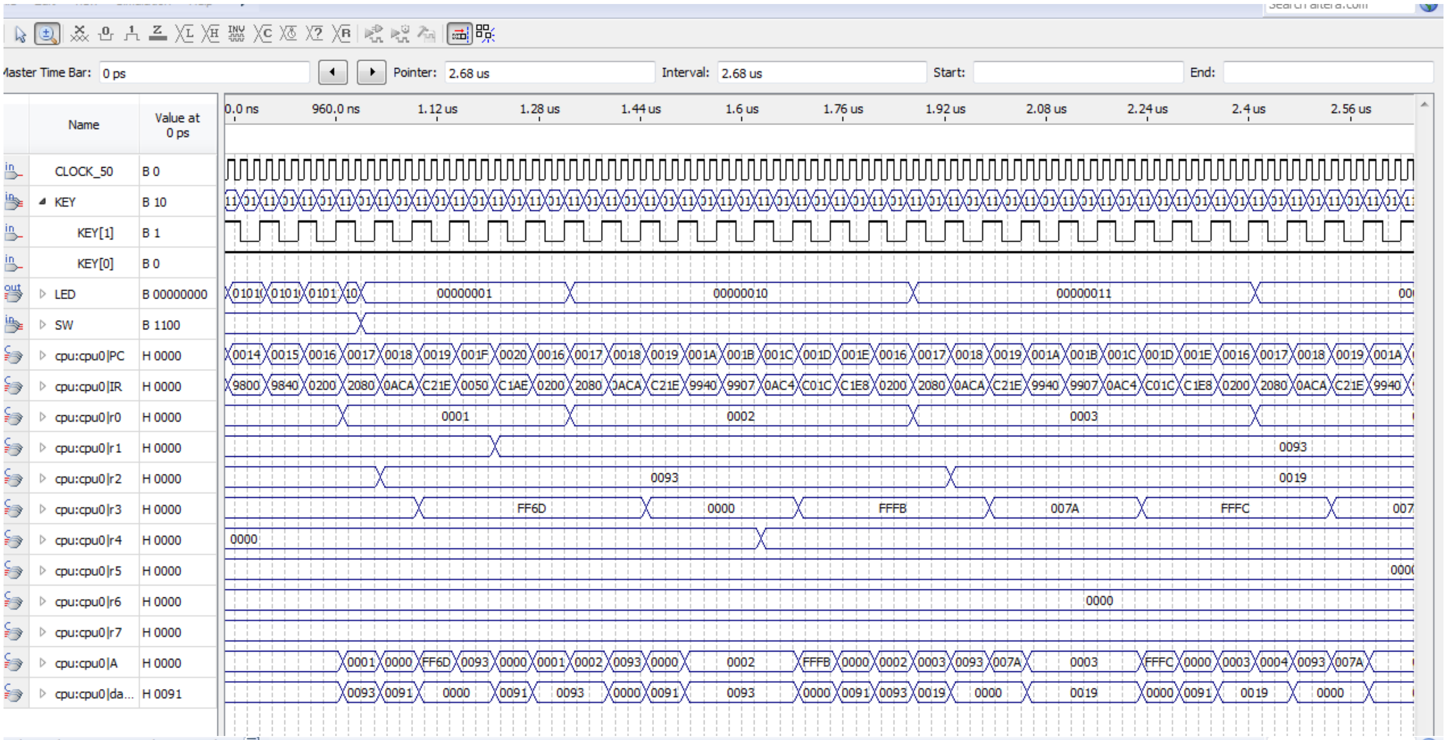


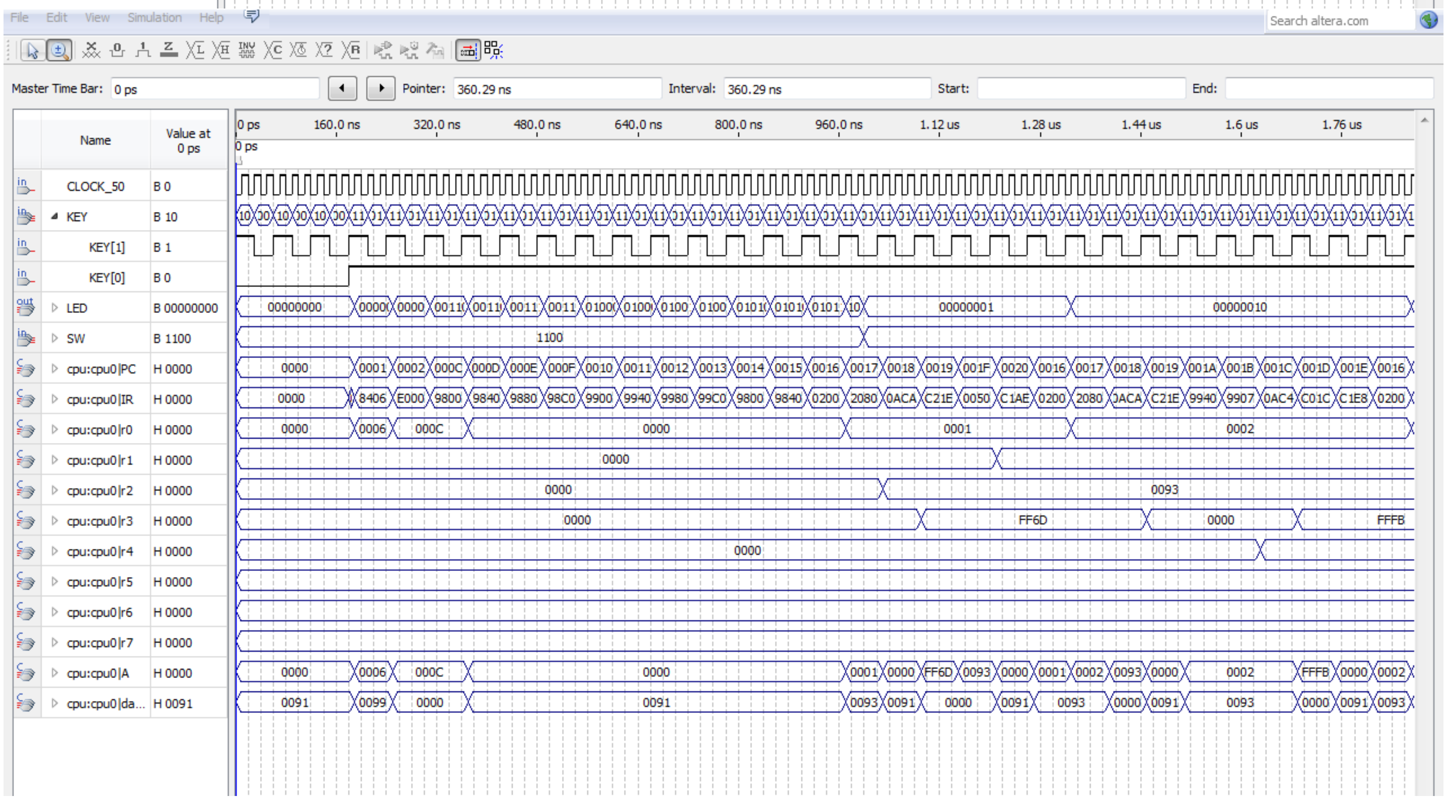

Time Bar: 0 ps
Pointer: 10.59 us
Interval: 10.59 us
Start:
End:
Name
Value at 0 ps
7.84 us
8.0 us
8.16 us
8.32 us
8.48 us
8.64 us
8.8 us
8.96 us
9.12 us
9.28 us
9.44 us
9.6 us
CLOCK_50 B 0
KEY B 10
KEY[1] B 1
KEY[0] B 0
LED B 00000000
SW B 1100
cpu:cpu0|PC H 0000
cpu:cpu0|IR H 0000
cpu:cpu0|r0 H 0000
cpu:cpu0|r1 H 0000
cpu:cpu0|r2 H 0000
cpu:cpu0|r3 H 0000
cpu:cpu0|r4 H 0000
cpu:cpu0|r5 H 0000
cpu:cpu0|r6 H 0000
cpu:cpu0|r7 H 0000
cpu:cpu0|A H 0000
cpu:cpu0|da... H 0091


File Edit View Simulation Help
Search altera.com
Master Time Bar: 0 ps
Pointer: 6.61 us
Interval: 6.61 us
Start:
End:
Name
Value at 0 ps
6.08 us
6.24 us
6.4 us
6.56 us
6.72 us
6.88 us
7.04 us
7.2 us
7.36 us
7.52 us
7.68 us
CLOCK_50 B 0
KEY B 10
KEY[1] B 1
KEY[0] B 0
LED B 00000000
SW B 1100
cpu:cpu0|PC H 0000
cpu:cpu0|IR H 0000
cpu:cpu0|r0 H 0000
cpu:cpu0|r1 H 0000
cpu:cpu0|r2 H 0000
cpu:cpu0|r3 H 0000
cpu:cpu0|r4 H 0000
cpu:cpu0|r5 H 0000
cpu:cpu0|r6 H 0000
cpu:cpu0|r7 H 0000
cpu:cpu0|A H 0000
cpu:cpu0|da... H 0091


Master Time Bar: 0 ps
Pointer: 7.09 us
Interval: 7.09 us
Start:
End:
Name
Value at 0 ps
4.16 us 4.32 us 4.48 us 4.64 us 4.8 us 4.96 us 5.12 us 5.28 us 5.44 us 5.6 us 5.76 us 5.92 us
CLOCK_50 B 0
KEY B 10
KEY[1] B 1
KEY[0] B 0
LED B 00000000
SW B 1100
cpu:cpu0|PC H 0000
cpu:cpu0|IR H 0000
cpu:cpu0|r0 H 0000
cpu:cpu0|r1 H 0000
cpu:cpu0|r2 H 0000
cpu:cpu0|r3 H 0000
cpu:cpu0|r4 H 0000
cpu:cpu0|r5 H 0000
cpu:cpu0|r6 H 0000
cpu:cpu0|r7 H 0000
cpu:cpu0|A H 0000
cpu:cpu0|da... H 0091
00000111
001C 001D 0021 0022 0023 0024 0025 0026 0027 0028 0029 002A 002B 002C 002D 002E 002F 0030 0031 0032 0033 0034 0035 0036 0038 0039 003A 003B 0033 0034 0035
0AC4 C01C 9984 1D86 4031 9800 9840 9880 98C0 9900 9940 9980 99C0 98C0 9940 9880 98C0 2158 98C1 2198 9907 0BEE C23A 0BE3 02D8 C03A C1D0 2198 9907 0BEE
0007
0031
0000 0001 0002
0000
0091
0004 0008 0000 0093
FFFE 0006
0000 0007 0008 0007 0000 0001 0000 0091 FFFE 0007 0001 0006 0000 0002 0000 0091
0091 0031 0000 0031 0091 0093 0091 0000 0031 0093 0099 0091 0093 0091


File Edit View Simulation Help
Search altera.com
Master Time Bar: 0 ps
Pointer: 4.79 us
Interval: 4.79 us
Start:
End:
Name
Value at 0 ps
2.4 us 2.56 us 2.72 us 2.88 us 3.04 us 3.2 us 3.36 us 3.52 us 3.68 us 3.84 us 4.0 us
CLOCK_50 B 0
KEY B 10
KEY[1] B 1
KEY[0] B 0
LED B 00000000
SW B 1100
cpu:cpu0|PC H 0000
cpu:cpu0|IR H 0000
cpu:cpu0|r0 H 0000
cpu:cpu0|r1 H 0000
cpu:cpu0|r2 H 0000
cpu:cpu0|r3 H 0000
cpu:cpu0|r4 H 0000
cpu:cpu0|r5 H 0000
cpu:cpu0|r6 H 0000
cpu:cpu0|r7 H 0000
cpu:cpu0|A H 0000
cpu:cpu0|da... H 0091
00000100 00000101 00000110
001E 0016 0017 0018 0019 001A 001B 001C 001D 001E 0016 0017 0018 0019 001A 001B 001C 001D 001E 0016 0017 0018 0019 001F 0020 0016 0017 0018 0019 001A 001B
C1E8 0200 2080 0ACA C21E 9940 9907 0AC4 C01C C1E8 0200 2080 0ACA C21E 9940 9907 0AC4 C01C C1E8 0200 2080 0ACA C21E 0050 C1AE 0200 2080 0ACA C21E 9940 9907
0004 0005 0006
0093 0099
0019 0022 0099
FFFC 007A FFFD 0071 FFFE FFFA 0068
0007
0000
0000
0000 0003 0004 0093 007A 0004 FFFD 0000 0004 0005 0093 0071 0005 FFFE 0000 0005 0006 0093 FFFA 0099 0000 0006 0007 0099 0068 0007
0091 0019 0000 0019 0000 0091 0019 0022 0000 0022 0000 0091 0022 0099 0000 0091 0099 0031 0000 0031





Start Program User Code Snippet:

// MAXIMUM VALUE FINDING

ldi r0, 0

ldi r1, 0

ldi r2, 0

ldi r3, 0

ldi r4, 0

ldi r5, 0

ldi r6, 0

ldi r7, 0

// The instructions above are meant to be used to clear all the register values to 0 so that in later uses, it will not mess it up. These are set to 0 because subtract and others that use values without initialization may mess itself up if there is a present variable already used.

ldi r0, 0

ldi r1, r0

inc r0, r0

ld r2, r0

// The code above is utilized to store values into r1 and r2 based on the first and second values in the data.txt. This is used later to find the biggest number by doing a constant comparison of the numbers with each other.

sub r3, r1, r2

// We use subtraction here as a way to decide which value is bigger. If R2 is bigger, then it means r3 - which is the final stored value for destination register - is negative.

brn r3, 6

// if r3 is negative, meaning r2 is bigger, we want to go down to value 6 and do some loops (1)

ldi r5, 0

ldi r4, 7

sub r3, r0, r4

brz r3, 4

brz r5, 56

mova r1, r2

// (1)So now we are down here to mova from the loop up above. We go to here because since we know r2 is bigger, we need to keep track of it. Logic right? So we store it from the temp variable of R2 into a more permanent variable of R1.

// ELSE it runs the code right above. IT ONLY runs this when/if it finds that R2 is not bigger, meaning its a failure. So then it loads the previous value of 0 and then the last number in data. Then it does some subtration to the two. If the value is 0, it will continue the code down below, else, it will still go down as R5 is always 0, to continue the other else loop

brz r5, 54

// Well you see, R5 is always = 0 so this is really only here to make it so that it loops back up. How nice!

ldi r6, 4

shl r6, r6

st r6, r1

// Then we set r6 to 4, and shift it left once to multiply it to 8. And then we do a store the value of r1 into the location of r6 which is 8 in memory.

// MAXIMUM VALUE FINDING END

// MINIMUM VALUE FINDING BEGIN

ldi r0, 0

ldi r1, 0

ldi r2, 0

ldi r3, 0

ldi r4, 0

ldi r5, 0

ldi r6, 0

ldi r7, 0

// We reset all the registers yet again to preserve contents in case of future error.

ldi r7, 0

ldi r3, 0

ldi r5, 0

ldi r2, 0

ldi r3, 0

ld r5, r3

ldi r3, 1

ld r6, r3

ldi r4, 7

// The values in location 0 in memory are stored in r5. While the value of r6 will be location 1 in memory thanks to r3 being assigned to integer 1

sub r7, r5, r6

// we then subtract r5 from r6. This is done to see which one will end up being the bigger or smaller number in the end.

// if r7 is negative it means r6 is bigger...which we dont really want, so it will redo the loop, we want r7 to be positive showing that r5 is indeed the bigger numer so that r6 can be the smaller number

brn r7, 2

mova r5, r6

// If the value of r6 is smaller, which we want, then we move it into the register r5 which we are reseting now to use this new value as we don't really have a use for r5 in the loop after the subtraction, so it can be a different temp file this time around.

sub r7, r4, r3

// we then do another subtraction with r4 and r3 which subtract the other two lines/iterations of the code

// after this, we increment r3 so we can load a different value into it next time.

inc r3, r3

brz r7, 2

// if r7 is zero, it means the loop is pretty much over so we go down to the shifting and do some storing in there!

brz r2, -8

// if r2 is zero which it will almost always be if the brz above doesn't get a true statement, it will constantly continue the loop

ldi r3, 4

shl r3, r3

inc r3, r3

st r3, r5

// we load the value of 4 into r3, and do a shift left to make it 8 and then an inc to make it 9 so that this is the code for minimum and then we store r5 into the location of r3 which is that 9

// MINIMUM CODE END!

// AVERAGE CODE BEGIN

ldi r0, 0

ldi r1, 0

ldi r2, 0

ldi r3, 0

ldi r4, 0

ldi r5, 0

ldi r6, 0

ldi r7, 0

// we reset all the Memory values and the code yet agains

ldi r0, 0

ldi r5, 0

ldi r6, 3

// we set r6 to 3 as a counter that will decriment. The loop only runs three times. This is because the memory locations 0 and 1 that get added up run outside the loop. Memory location pairs 2 3, 4 5, 6 7 are the ones that get added up while the loop runs, this happens three times thus. So to get out of the loop, we have a single decrimenter in there that gets reduced every time. In the end, the decrimenter will be zero, and the loop will end and it will branch to the shifting and storing functions.

ld r1, r0

ldi r2, 1

ld r3, r2

// we now store values into r0 and r2, and then we use load functions to load the actual memory values into r1 and r3 respectively. This gives us our first two values

add r4, r1, r3

// we then use an add function to add up the values of the r1 and r3

add r5, r5, r4

// we use the r5, r5, r4 to continously keep a running total of the max.

brz r6, 10

// if r6 is zero, we want the loop to break. This will jump down 10 lines of code to the shifting. This only hapens when the loop has run 3 times successfully

inc r0, r0

inc r0, r0

inc r2, r2

inc r2, r2

dec r6, r6

// we do a double increment of both r0 and r2 which are counters because we want the valued pairs to be the +2 past what it was before so that the program wont add the same two numbers continuously

// we do a decrement of 6 so that we can break the loop when r6 = 0 with the BRZ

ld r1, r0

ld r3, r2

// we restore values and re run the loop after incrementing

ldi r7, 0

// r7 is always = 0 so this will always run until the loop breaks, at which point we wont be able to.

brz r7, 53

// since r7 is always 0, this will always run granting the break loop above doesn't initialize.

shr r5, r5

// this code only runs once the break loop gets initialized with r6 = 0 after being decrimented 3 times from the loop that it runs through

shr r5, r5

shr r5, r5

// you perform a shift right on the value of r5, which holds the runing total 3 times. This is 3 times because 2^3 is 8 which is the value we need to divide the numbers by to get our values. Of course in hex, not just regular decimal division.

ldi r0, 7

// the r0 is now re-initialized/reset to be a diffferent value, it will be here to be used for store.

inc r0, r0

inc r0, r0

inc r0, r0

// we increment r0 3 times so that r0 is = 10. This is 0x10 which is the location we need our average to be stored in, which corresponds with register 3.

st r0, r5

// so we store the new total into the mem location of 40

// and do a jump to location in memory 3

ldi r0, 3
jmp r0

DATA.TXT CODE:

@0

0091

0093

0093

0019

0019

0022

0099

0031

Instructions.TXT Code:

9800

9840

9880

98c0

9900

9940

9980

99c0

9800

9840

0200

2080

0aca

c21e

9940

9907

0ac4

c01c

c1e8

0050

c1ae

9984
1d86
4031
9800
9840
9880
98c0
9900
9940
9980
99c0
99c0
98c0
9940
9880
98c0
2158
98c1
2198
9907
0bee
c23a
0170
0be3
02d8
c03a
c1d0
98c4
1cc3
02d8
401d
9800
9840
9880
98c0
9900

9940

9980

99c0

9800

9940

9983

2040

9881

20d0

050b

056c

c072

0200

0200

0290

0290

0db0

2040

20d0

99c0

c1bd

1b45

1b45

1b45

9807

0200

0200

0200

4005

9803
e000