CS3350 Computer Organization
Mazen Baioumy
250924925
Assignment 3

<u>Answer to Question 1:</u>

Answer to Question 1

1. add $v1 (3), $a2 (6), $t8 (24) (R-Type)

OP code	RS	RT	RD	shamt	Funct
0x00	$a2	$t8	$v1	0	0x20
000000	00110	11000	00011	00000	100000
6 bits	5bits	5bits	5bits	5 bits	6 bits

Answer: 0000 0000 1101 1000 0001 1000 0010 0000

Final answer: 0x00D81820

2. lb $s2, 37($t7) (I-type)

OP code	RS	RT	Immediate
lb → 32 32 → $\boxed{0x20}$ 100000	$t7 (15) 01111	$s2 (18) 10010	0x0025 (37)
6 bits	5 bits	5bits	16 bits

Answer: 1000 0001 1111 0010 0025

Final answer: 0x81F20025

For branch instructions Put $v0$ in Rs Field and $t4$ in RT Field

3. bne $v0, $t4, 732 (I-type)

OP Code	RS	RT	Immediate
bne → 5 5 → 0x5 000101	$v0 → 2 00010	$t4 → 12 01100	732 1011011100 = 02DC
6 bits	5 bits	5 bits	16 bits

Answer: 0001 0100 0100 1100

$\boxed{\text{Final answer: } 0x144C02DC}$

4. sra $s4, $s6, 12 → source register goes to (rt field)
→ nothing goes to RS at

OP code	RS	RT	RD	ShAmt	Funct
0x00 000000	0x00 00000	$s6 → 22 16 4 2 10110	$s4 → 20 16 4 10100 20	11 01011	0x03 000011
6 bits	5 bit	5 bit	5 bits	5 bits	6 bits

Answer: 0000 0000 0001 0110 1010 0010 1100 0011

$\boxed{\text{Final Answer: } 0x0016A2C3}$

Answer to Question 2:

a.) Since we need the value of L2 in the instruction: **beq $t4, $0, L2.** The PC at each instruction increments to point to the next instruction. Therefore, when we are at **beq** instruction, the program counter will aim to the memory address of the instruction **sub $t3, $t3, $t2.** Since **beq** instruction is saying that the value of $t4 and $0 registers are equal, we jump to the L2 instruction **(sub $t3, $t2, $t3).** The value of L2 which is the immediate will be 2. Due to the program counter at **beq** instruction contains the address for **sub $t3, $t3, $t2** instruction. For the sub instruction, to get to L2, we need to move by 2 instructions. Therefore, we add a logical shift left on the value 2. This will allow the jump. The format of the beq instruction is broken down to.

OP → 4 → 000100
$t4 → 12 → 01100
$0 → 0 → 00000
Immediate → 2 → 0000000000000010

So, the value of L2 is 2. And Hexadecimal value is 0x11800002.

b.) Assuming that the fetched instruction is a J-type jump instruction, which has an immediate field of 26 bits that specifies the jump target address, we can use the NPC formula as follows:

NPC = (PC & 0xF0000000) | (address << 2)

where NPC is the value of the program counter on the next clock cycle, PC is the current value of the program counter, and address is the 26-bit jump target address extracted from the immediate field.

Extracting the jump target address from the instruction word, we have:

Address = 0x089F01A7 & 0x09F01A7 = 0x022F806B

Substituting the given values, we have:

NPC = (0x1258AB91 & 0xF0000000) | (0x09F01A7<< 2)
= 0x12000000 | 0x07C069C
= 0x127C069C

Therefore, the value of the program counter on the next clock cycle, when the fetched instruction is a J-type jump instruction, is 0x127C069C.

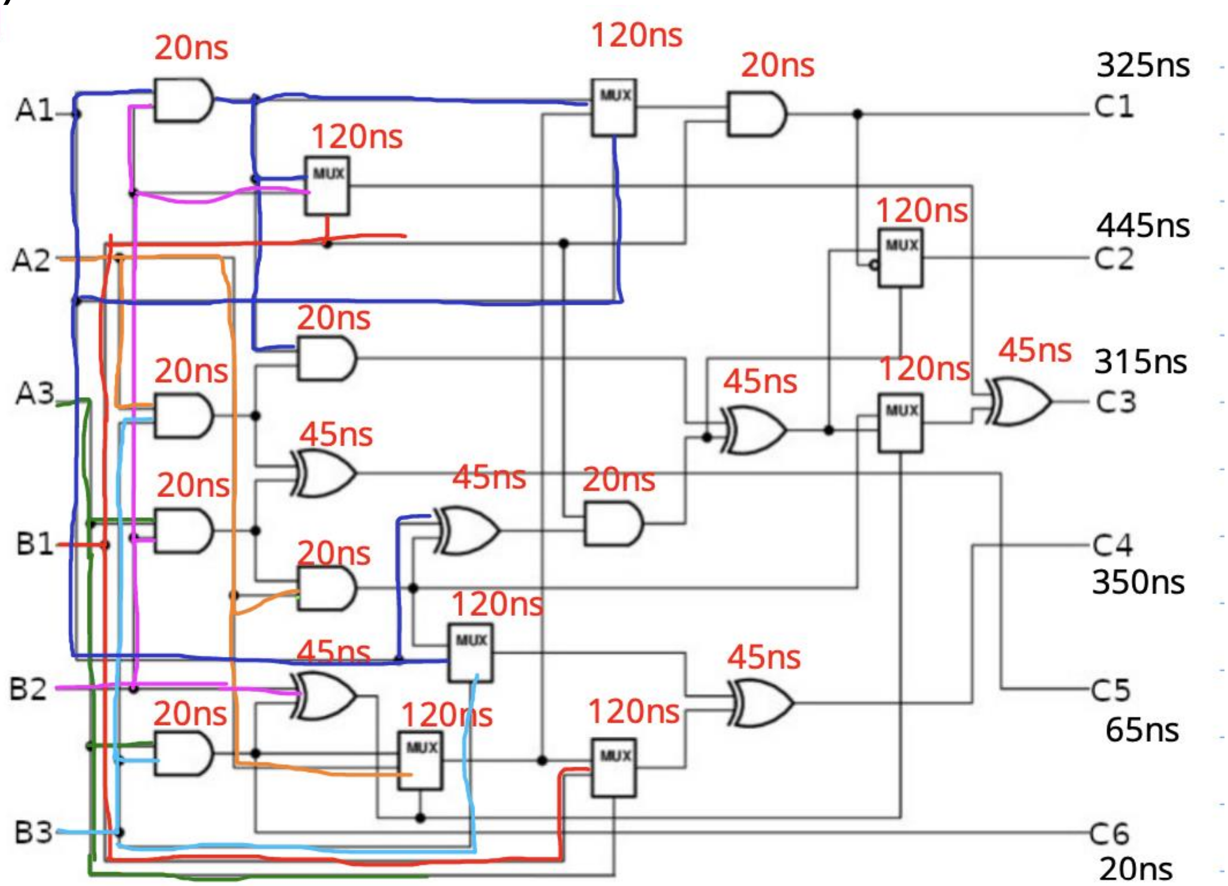
Answer to Question 3:

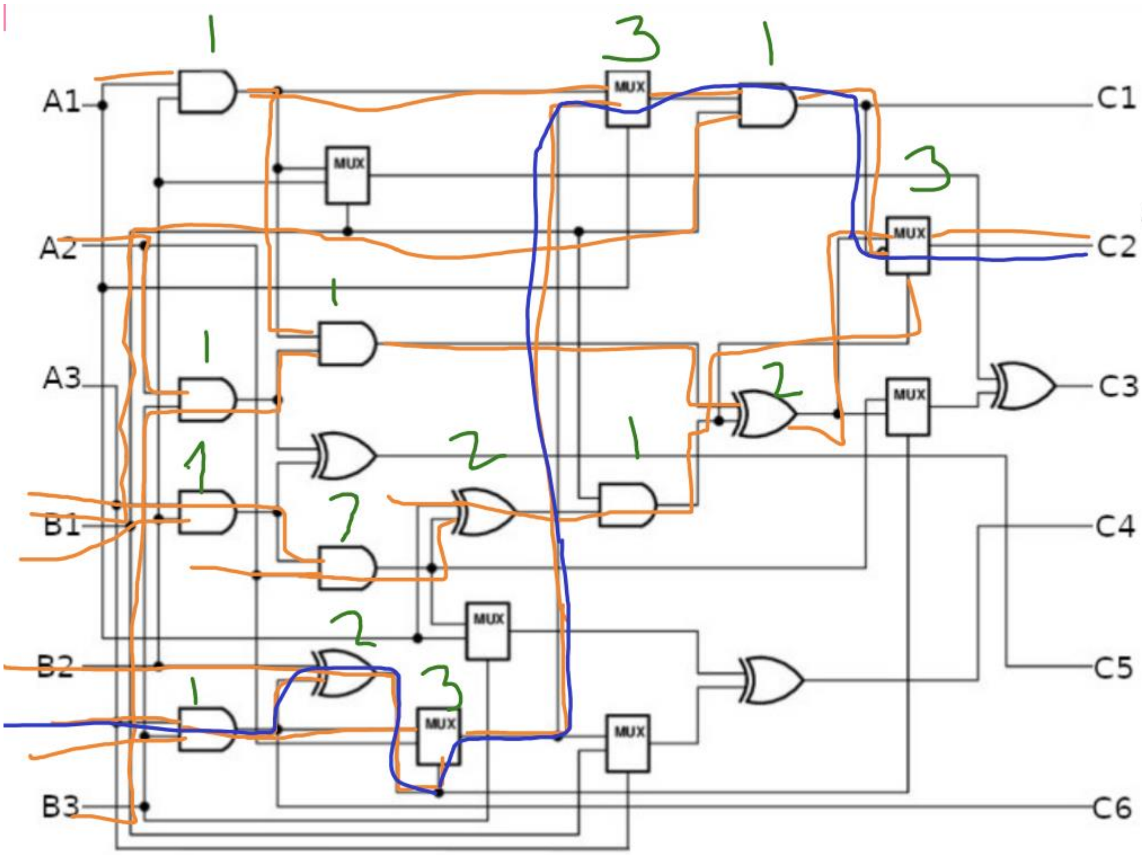
1. In line 3 The code needs to **be changed** from (add $t0, $v1, $zero) **to** → (sll $t1, $a1, 2)
2. In line 4 the code needs to **be changed** from (add $t0, $v0, $t0) **to** → (add $t1, $a0, $t1)
3. In line 5 code needs to be **changed** from (lw $t1, 0($t0)) **to** → (lw $t0, 0($t1))
4. In line 6 code needs to be **changed** from (lw $s1, 4($t0)) **to** → (lw $t2, 4($t1))
5. In line 7 code needs to be **changed** from (sw $s1, 0($t0)) **to** → (sw $t2, 0($t1))

6. In line 8 code needs to be **changed** from (sw $t1, 4($t0)) **to** → (sw $t0, 4($t1))
7. **Before** line 14 (sw $s3, 12($sp) **add the** following line → (sw $ra, 16($sp))
8. In line 35 code needs to **be changed** from (add $v1, $s1 , $zero) to → (add $a1, $s1, $zero)
9. In line 36 code needs to **be changed** from (j swap) **to** → (jal swap)
10. **After** line 46 (lw $s3, 12($sp)) **add the** following line → (lw $ra, 16($sp))

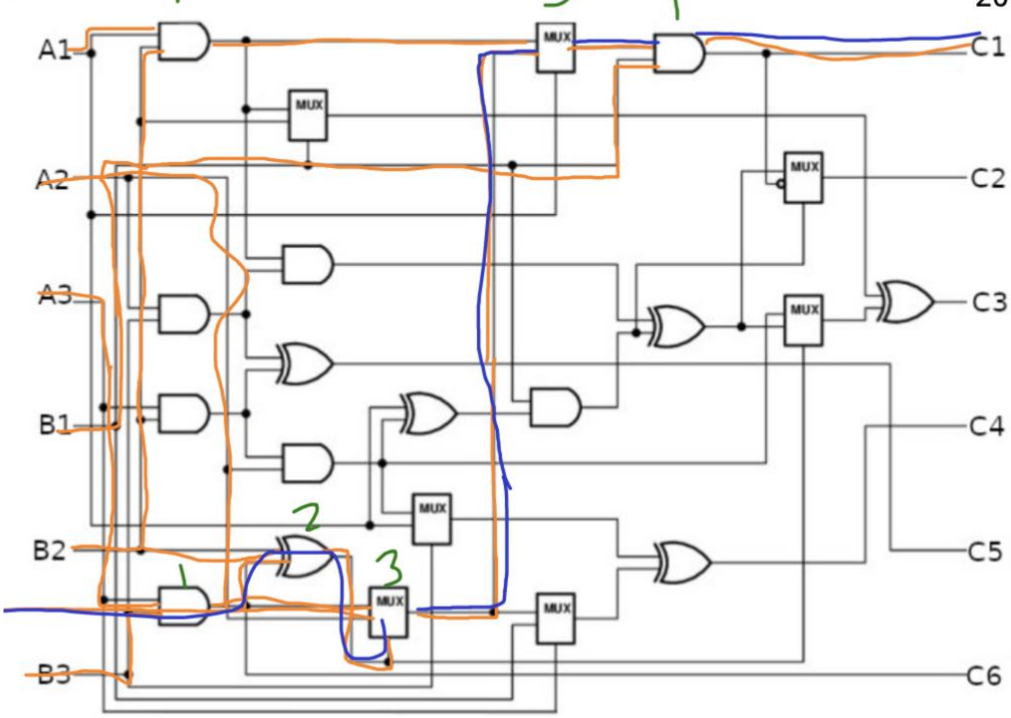
The following 10 changes allow the code to run and produces the correct output using bubble sort to sort the following numbers as seen from Qtspim compiler. Screenshot of code is located below.

<u>Answer to Question 4:</u>
a.)

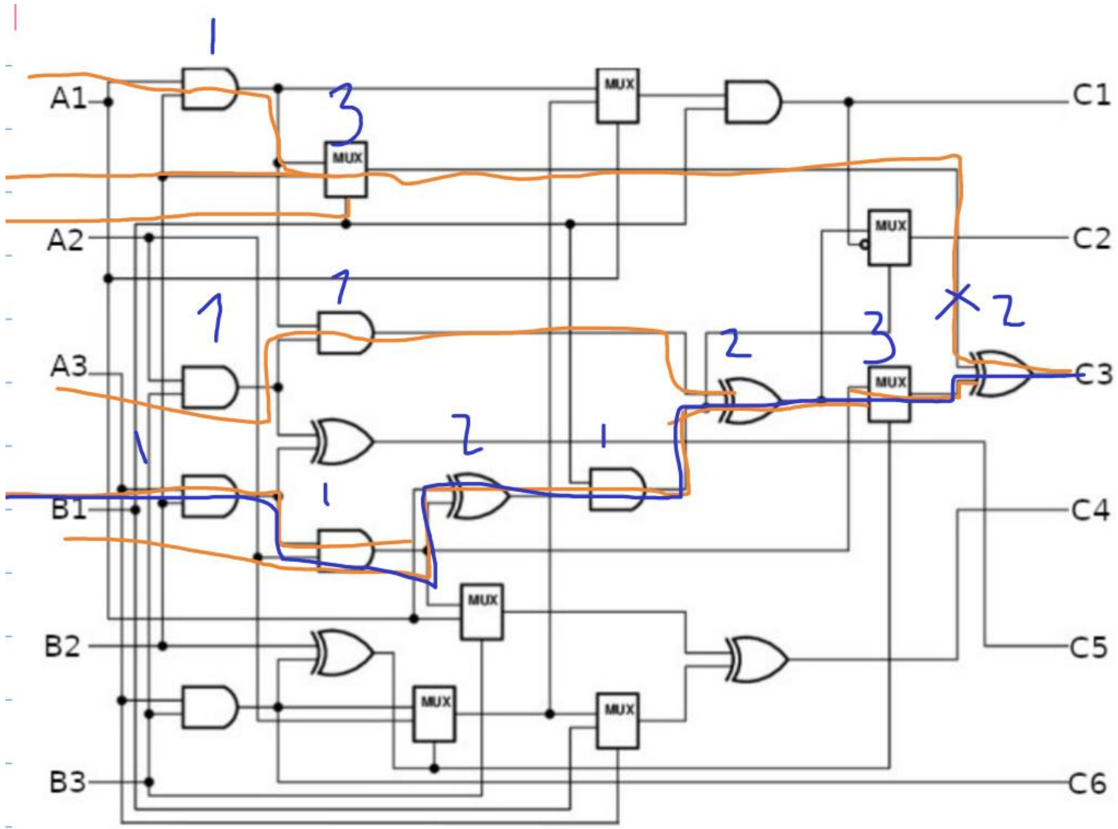




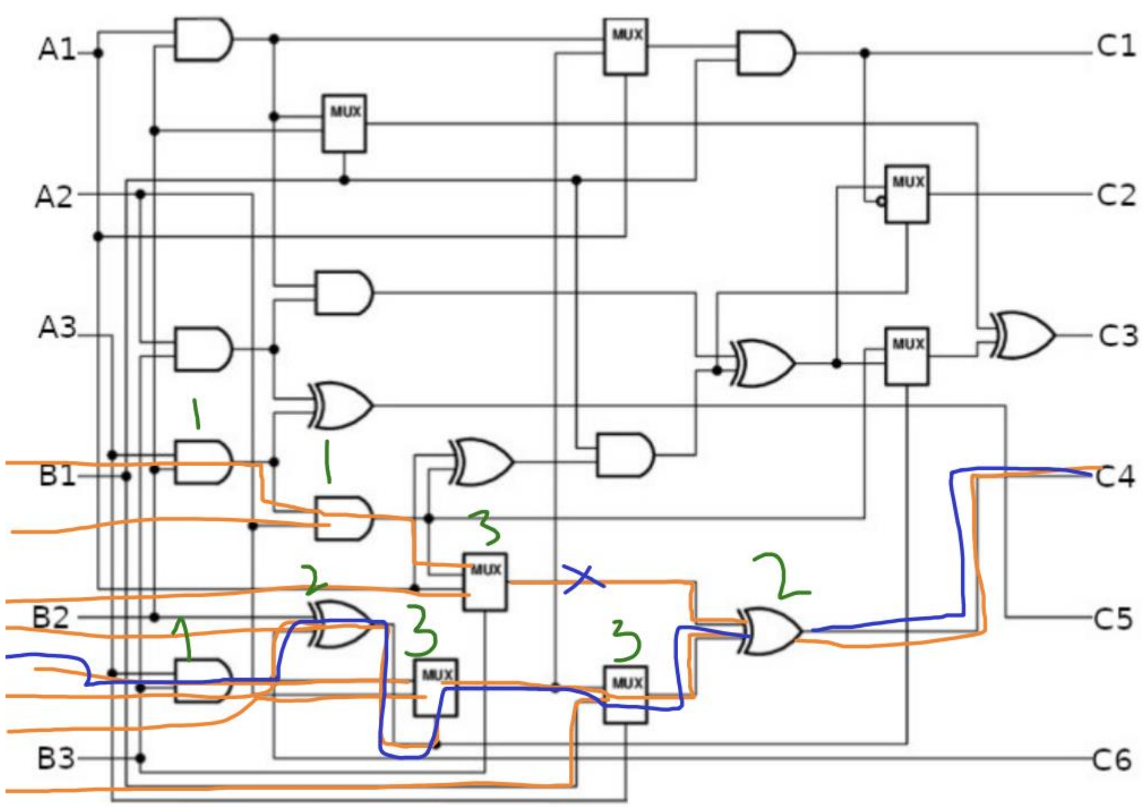
445ns



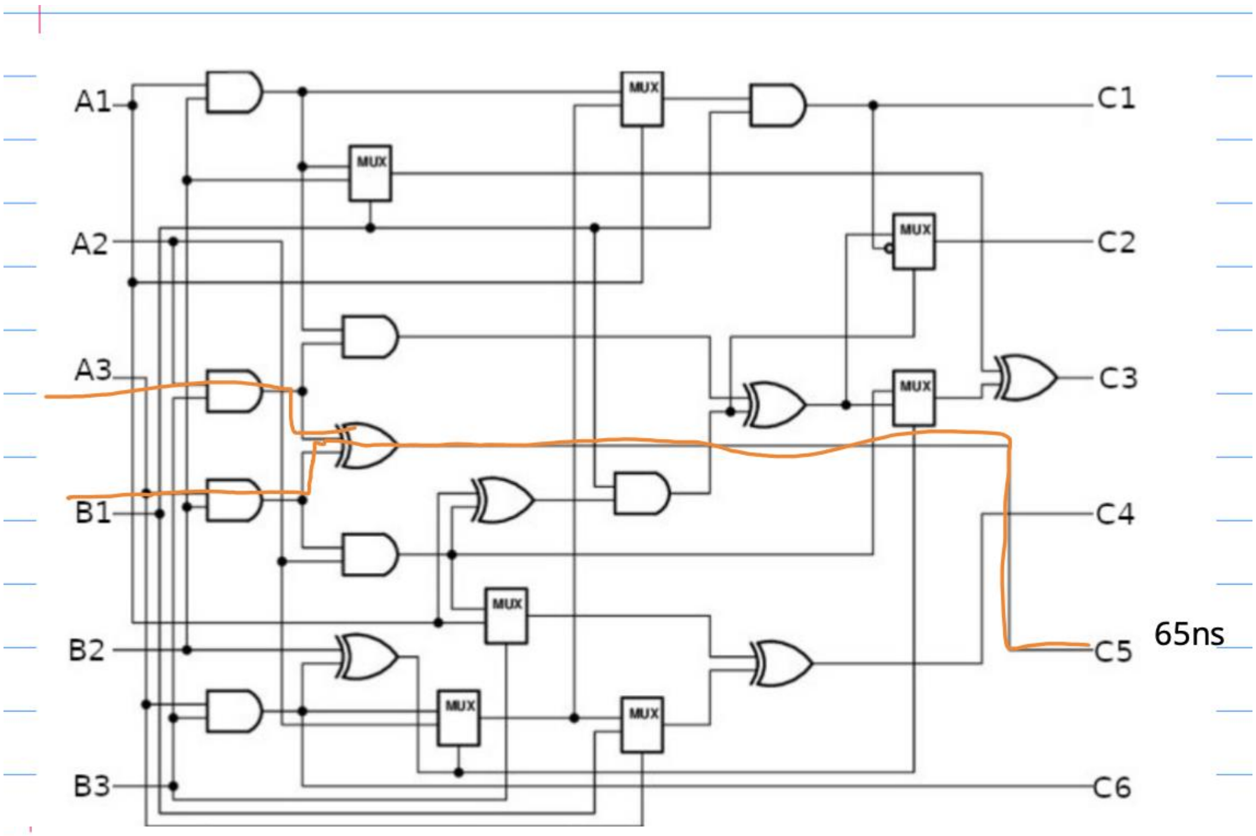
20+120+120+45+20
= 325ns

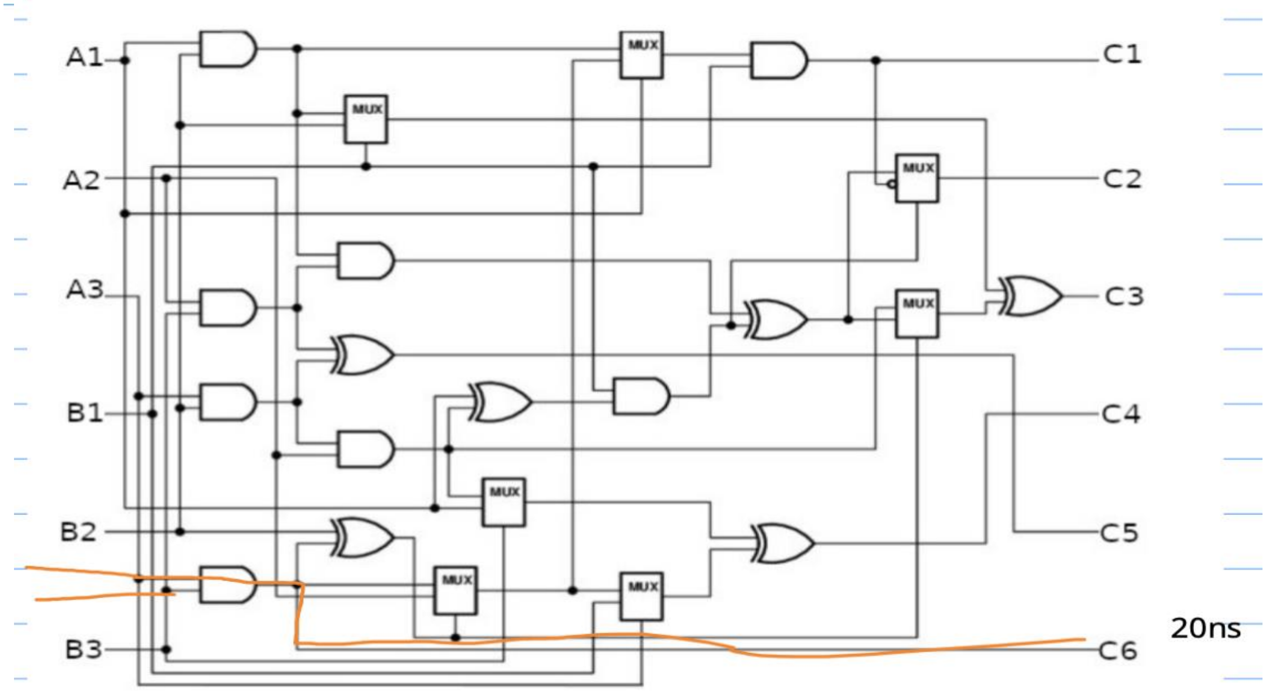

A1
A2
A3
B1
B2
B3
C1
C2
C3
C4
C5
C6
MUX
1
3
2
x

315 ns


A1
A2
A3
B1
B2
B3
C1
C2
C3
C4
C5
C6
MUX
1
2
3
x

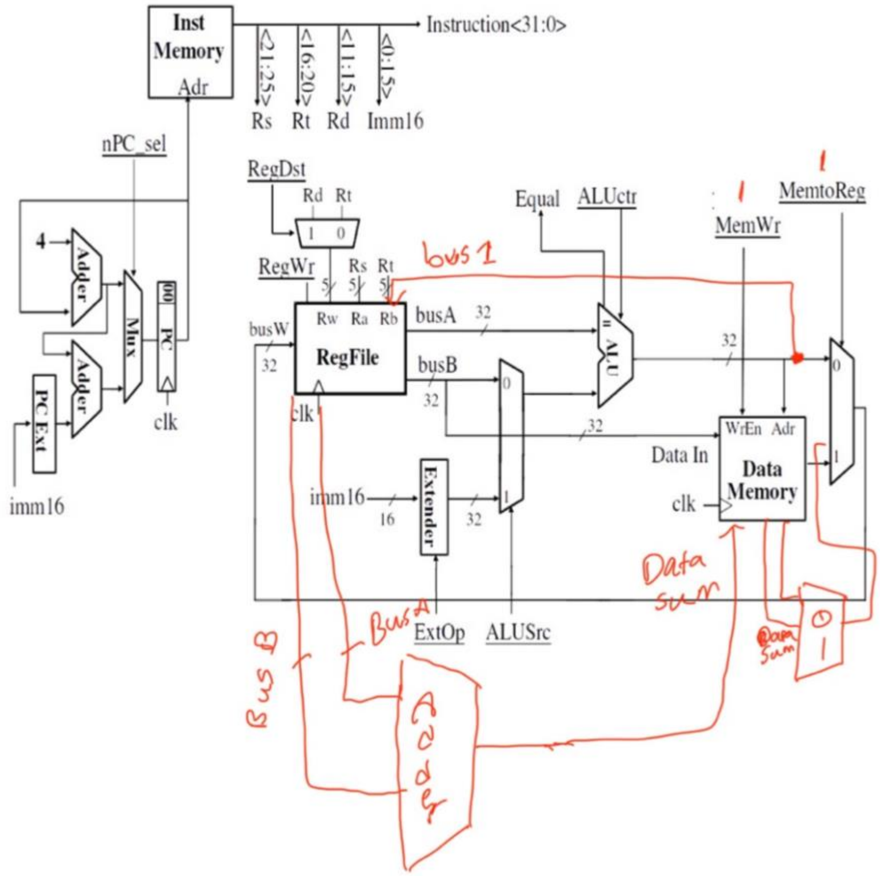
350ns





b.) The propagation delay of the circuit is the time it takes for the output with the longest critical path to become available. In this case, that output is C2, and its value at **<u>445ns.</u>**

Answer to Question 5:



b.) I placed an ADDER, control signals and a few wires to add the function to the already existing circuit. When the foo instruction is read and decoded, the PC is incremented by 4. As we leave the RegFile, both the busA and busB are are sent to the ALU (therefore imm is not picked in the multiplexer). The operation done in there is an addition, so Reg[$rs] + Reg[$rt]. As it goes out of the ALU, it passes to Data Memory and picks up the memory at index Reg[$rs] + Reg[$rt]. It picks it up and its value is saved in $rt when it goes back to the RegFile. As for the other instructions in the RTL, it is done using the additions I brought in. Two wires leaving busA and imm32 are fed into an adder, resulting in Reg[$rs] + IMM. The result is saved in $rs as it goes back to the RegFile, and the control signal Foo2 helps us with that. To save the value in $rt into the memory location at $rs, we add another wire going to Data memory alongside a control signal Foo1. When it is 1, it takes that new wire as the memory index and the content from DATA IN as its content. When any instruction other than foo is executed, Foo2 is set to 0 so that the value from the adder does not change anything in the RegFile. The 01 adder on the right will check wether data has been stored in REG[$rs], then based on it if its 1 or 0 it will store data in r2 or not. The left adder will determine what output comes from the Mem[adder]. Moreover, Foo1 is set to 0, so no additional memory access is allowed. Everything runs as usual. Data flows through modified datapath as foo is being executed.

c.) RegDst = rt
MemtoReg = Mem
ALUSrc = 1
nPC_sel = +4
RegWrite = 1 (write)
MemWrite = 1 (write)
ExtOp = 1
Jump = 0
ALUctr = add
FooLoad = 1
FooAdd = 1

d.) Foo1 = 0, Foo2 = 0

Answer to Question 6:

Answers to question 6:

a. $115 + 225 + 175 + 200 + 150 + 425 + 200 + 225$

$\boxed{= 1715\ PS}$

b. In pipeline dataPath clock cycle time:

$\Rightarrow$ Maximum$(115, 225, 175, 200, 150, 425, 200, 225) = \boxed{425PS}$

c. Ideal speedup $= 1715 / 425 = \boxed{4.0352}$

Execution time of 300 instructions in single cycle dataPath (300 x 1715) PS. In Pipeline machine, first instruction will take 8 cycles. Next 299 instructions will take 1 clock cycle each. Total number of clock needed = $(8+299) = \boxed{307}$
Execution time of 300 instructions in Pipeline datapath = $(307) \times (425)$

Actual speedup= $(300 \times 1715) / (300 \times 425 + 2975)$

$= 3.943$

D. Add → 5 stages

S11 → 4 stages

SW → 5 stages

lW → 7 stages

beq → 3 stages

Total clocks needed = (1500)(5) + (4)(250) + (5)(500) + (400)(7) + (350)(3)

$\boxed{= 14850}$

Total instruction = (1500 + 250 + 500 + 400 + 350) = $\boxed{3000}$

i) $\boxed{CPi = (14850 / 3000) = 4.95}$ 3

ii) Execution time = (3000)(4.95)(425) = $\boxed{6.31\ \mu s}$

↳ (Total clocks)(Clock time)

ANOTHER METHOD FOR CALCULATING EXECUTION TIME IS:

For ii we can calculate execution time using:

add = (5*425) = 2125

sll = (4*425) = 1700

sw = (5*425) = 2125

lw = (7*425) = 2975

beq = (3*425) = 1275

Total execution time = (1500 x add) + (250 x sll) + (500 x sw) + (400 x lw) + (350 x beq) = 6.311

FOR QUESTION 3

```
swap:
        sll $t1, $a1, 2
        add $t1, $a0, $t1   #
        lw $t0, 0($t1)      #
        lw $t2, 4($t1)      #
        sw $t2, 0($t1)      #
        sw $t0, 4($t1)
        jr $ra

#####
# Bubble sort an array where
#####
sort:
        addi $sp, $sp, 16
        sw $ra, 16($sp)
        sw $s3, 12($sp)
        sw $s2,  8($sp)
        sw $s1,  4($sp)
        sw $s0,  0($sp)

        add $s2, $a0, $zero
        add $s3, $a1, $zero
        add $s0, $zero, $zero

for1tst:
        slt $t0, $s0, $s3
        beq $t0, $zero, exit1
        addi $s1, $s0, -1
for2tst:
        slti $t0, $s1, 0
        bne $t0, $zero, exit2
        sll $t1, $s1, 2
        add $t2, $s2, $t1
        lw $t3, 0($t2)
        lw $t4, 4($t2)
        slt $t0, $t4, $t3
        beq $t0, $zero, exit2
        add $v0, $s2, $zero
        add $a1, $s1, $zero
        jal swap
        addi $s1, $s1, -1
        j for2tst

exit2:
        addi $s0, $s0, 1
        j for1tst


exit1:
        lw $s0,  0($sp)
        lw $s1,  4($sp)
        lw $s2,  8($sp)
        lw $s3, 12($sp)
        lw $ra, 16($sp)
        addi $sp, $sp, -16
        jr $ra
```