CSO Assignment 2

Roll no: 2020101093

Task 1:

Instruction encoding for the new instruction is 1 byte long which is as follows.

paddq :

icode : ifun

A	1

psubq:

icode : ifun

A	2

pandq:

icode : ifun

A	3

pxorq:

icode : ifun

A	4

The new instruction is 1 byte long.

Task 2:

```
# Execution begins at address 0
        .pos 0
        irmovq stack, %rsp        # Set up stack pointer
        call main                 # Execute main program
        halt                      # Terminate program

 # Array of 8 elements
         .align 8
 array:
          .quad 0x000000000001
          .quad 0x000000000002
          .quad 0x000000000003
          .quad 0x000000000004
```

```
               .quad 0x000000000005
               .quad 0x000000000006
               .quad 0x000000000007
               .quad 0x000000000008

 main:
          irmovq array,%rdi                  # address of array→ %rdi
          irmovq $8,%rsi
          call sum                           # sum(array, 8)
          ret
# long sum(long *start, long count)
# start has array address
# start in %rdi, count in %rsi
sum:
          irmovq $8,%r8           # Constant 8
          irmovq $1,%r9           # Constant 1
          xorq %rax,%rax          # sum = 0
          andq %rsi,%rsi          # Set CC
          pushq  %rax             # top of stack has value 0
          jmp test                # Goto test

loop:
          mrmovq (%rdi),%rbx                 # Get *start
          pushq %rbx              # pushes the value in the array to top of the stack
          paddq                # adds top two elements of the stack and pushes the result to the stack
                               # and also update %rcx,%rdx
         addq %r8,%rdi            # start++
          subq %r9,%rsi            # count-- ,  Set CC
test:
          jne loop          # Stop when ZF=1,i.e,when value of %rsi becomes 0
          popq  %rax        # pops the final sum from the top of  stack to %rax

# we need to do 16 pops to remove the intermediate values we pushed to stack during
# calculation of sum
         irmovq $16,%r10
pops:
          popq  %rbx
          subq %r9,%r10
          jne pops
          ret               # Return
# Stack starts here and grows to lower addresses
          .pos 0x200
stack:
```

Task 3

```
              0x0000:                    |       .pos 0
0x0000: 30f4000200000000000000  |  irmovq stack, %rsp
0x000a: 805800000000000000     |       call main                    # Execute main program
0x0013: 00                     |       halt                       # Terminate program
                               |     # Array of 8 elements
0x0014:                        |            .align 8
0x0018:                        |  array:
0x0018: 0100000000000000                |              .quad 0x000000000001
0x0020: 0200000000000000                |              .quad 0x000000000002
0x0028: 0300000000000000                |              .quad 0x000000000003
0x0030: 0400000000000000                |              .quad 0x000000000004
0x0038: 0500000000000000                |              .quad 0x000000000005
0x0040: 0600000000000000               |             .quad 0x000000000006
0x0048: 0700000000000000                |              .quad 0x000000000007
0x0050: 0800000000000000                |              .quad 0x000000000008

0x0058:                                  |  main:
0x0058: 30f71800000000000000         | irmovq array,%rdi                # 0x0018→ %rdi
0x0062: 30f60800000000000000        | irmovq $8,%rsi
0x006c: 807600000000000000          | call sum                         # sum(array, 8)
0x0075: 90                          | ret
                     | # long sum(long *start, long count)
                      | # start has array address
                      | # start in %rdi, count in %rsi
0x0076:                              | sum:
0x0076: 30f80800000000000000         | irmovq $8,%r8          # Constant 8
0x0080: 30f90100000000000000         | irmovq $1,%r9          # Constant 1
0x008a: 6300                         | xorq %rax,%rax         # sum = 0
0x008c: 6266                        | andq %rsi,%rsi         # Set CC
0x008e: a00f                        | pushq  %rax             # top of stack has value 0
0x0090: 70aa00000000000000          | jmp test               # Goto test

0x0099:                              | loop:
0x0099: 50370000000000000000         | mrmovq (%rdi),%rbx            # Get *start
0x00a3: a03f                         | pushq %rbx
0x00a5: a1                           | paddq
0x00a6: 6087                        | addq %r8,%rdi            # start++
0x00a8: 6196                         | subq %r9,%rsi                    # count-- ,  Set CC
0x00aa:                             | test:
0x00aa: 749900000000000000           | jne loop        # Stop when ZF=1
0x00b3: b00f                        | popq  %rax                            |

 |# we need to do 16 pops to remove the intermediate values we
|   pushed to stack during     calculation of sum
```

```
0x00b5: 30fa1000000000000000        | irmovq $16,%r10
0x00bf:                              | pops:
0x00bf: b03f                        | popq  %rbx
0x00c1: 619a                         | subq %r9,%r10
0x00c3: 74bf00000000000000           | jne pops
0x00cc: 90                          | ret                  # Return
                                    | # Stack starts here and grows to lower addresses
0x00cd:                             | .pos 0x200
0x0200:                             | stack:
```

Task 4

Assuming there are two address lines in memory stage one connecting **valA**, other with **valE.** And also two values **valM1,valM2** can be read simultaneously from the data memory, i.e,valM1=M[valE],valM2=M[valA].
In the decode stage we added valS which is used in the cycle2 of the new instruction.
Similarly,we assume there are two ALU units in execute stage which calculates values **valE,valE1**
Address of the new instruction used is 0x00a5 .It requires two cycles to complete.

Stage	**cycle1**	**0x00a5: a1 | paddq**
Fetch	icode : ifun ← M_1[PC] valP ← PC + 1	icode : ifun ← M_1[0x00a5] = a:1 valP ← 0x0a5 + 1 = 0x0a6

Decode	valA ← R[%rsp]
	valB ← R[%rsp]
		valA ← R[%rsp] = 0x1E0
		valB ← R[%rsp] = 0x1E0

Execute	valE ← valB + 8	valE ←0x1E0 + 8 = 0x1E8

Memory	valM1 ← M_8[valE]
	valM2 ← M_8[valA]
		valM1 ← M_8[0x1E8] = 0
		valM2 ← M_8[0x1E0] = 0x001

Write back	R[%rcx] ← valM1 R[%rdx] ← valM2	R[%rcx] ← 0 R[%rdx] ← 0x001

PC update	PC ← PC(no update)	PC ← 0x00a5

Stage	**cycle2**	**0x00a5: a1 | paddq**
Fetch	icode : ifun ← M_1[PC] valP ← PC + 1	icode : ifun ← M_1[0x00a5] = a:1 valP ← 0x0a5 + 1 = 0x0a6

Decode	valA ← R[%rcx]
	valB ← R[%rdx]
	valS← R[%rsp]
		valA ← R[%rcx] = 0
		valB ← R[%rdx] = 0x001
		valS ← R[%rsp] = 0x1E0

Execute	valE ← valB OP valA
	Set CC
	valE1 ← valS - 8
		valE ←0x001+0 = 0x001
		ZF ← 0 , SF ← 0 , OF ← 0
		valE1 ←0x1E0- 8 = 0x1D8

Memory	M_8[valE1] ←valE	M_8[0x1D8] ←0x001
Write back	R[%rsp] ← valE1	R[%rsp] ← 0x1D8
PC update	PC ← valP	PC ← 0x00a6

Thus,the new instruction takes 2 cycles to complete .

Task 5

The above Y86-64 code uses only %rdi,%rsi,%r8,%r9,%r10,%rax,%rbx,%rcx,%rdx among general purpose registers. The left -over general purpose registers have random values. Initially %rdi,%rsi,%r8,%r9,%r10,%rax,%rbx,%rcx,%rdx have some random values. Registers which are not mentioned in the column of general purpose registers have some random values which do not affect the program.

cycle	PC	GENERAL PURPOSE REGISTERS	CC	%rsp	Modified addresses
1	0x0000		000	0x200→%rsp	-
2	0x000a		000	0x1F8→%rsp	0x0013→M[0x1F8]
3	0x0058	0x0018→ %rdi	000	-	-
4	0x0062	8→ %rsi, %rdi=0x0018	000	-	-
5	0x006c	%rsi=8 , %rdi=0x0018	000	0x1F0→ %rsp	0x0075→M[0x1F0]
6	0x0076	8→ %r8, %rsi=8 , %rdi=0x0018	000	-	-
7	0x0080	1→ %r9, %r8=8, %rsi=8 , %rdi=0x0018	000	-	-
8	0x008a	0→%rax, %r9=1 ,%r8=8, %rsi=8 , %rdi=0x0018	100→ CC	-	-
9	0x008c	8→ %rsi, %rax=0,%r9=1 ,%r8=8, %rdi=0x0018	000→ CC	-	-
10	0x008e	%rsi=8, %rax=0,%r9=1 ,%r8=8, %rdi=0x0018	000	0x1E8→%rsp	0→ M[0x1E8]
11	0x0090	%rsi=8, %rax=0,%r9=1 ,%r8=8, %rdi=0x0018	000	-	-
12	0x00aa	%rsi=8, %rax=0,%r9=1 ,%r8=8, %rdi=0x0018	000	-	-
13	0x0099	0x0001→%rbx, %rsi=8, %rax=0,%r9=1 ,%r8=8, %rdi=0x0018	000	-	-
14	0x00a3	%rbx=1, %rsi=8, %rax=0,%r9=1 ,%r8=8, %rdi=0x0018	000	0x1E0→ %rsp	0x0001→M[0x1E0]

15	0x00a5	0→ %rcx,0x0001→%rdx, %rbx=1, %rsi=8, %rax=0,%r9=1 ,%r8=8, %rdi=0x0018	000	-	-
16	0x00a5	%rbx=1, %rsi=8, %rax=0, %r9=1 ,%r8=8, %rdi=0x0018,%rcx=0, %rdx=0x0001	000→ CC	0x1D8→%rsp	0x0001→M[0x1D8]
17	0x00a6	0x0020→%rdi,%rbx=1, %rsi=8, %rax=0,%r9=1 ,%r8=8,%rcx=0, %rdx=0x0001	000→ CC	-	-
18	0x00a8	7→%rsi, %rdi=0x0020, %rbx=1, %rax=0,%r9=1 ,%r8=8,%rcx=0, %rdx=0x0001	000→ CC	-	-
19	0x00aa	%rsi=7, %rdi=0x0020, %rbx=1, %rax=0,%r9=1 ,%r8=8,%rcx=0, %rdx=0x0001	000	-	-
20	0x0099	0x0002→%rbx,%rsi=7, %rdi=0x0020, %rax=0, %r9=1 ,%r8=8,%rcx=0, %rdx=0x0001	000	-	-

Task 6

The whole code takes 152 cycles to get executed.It takes 9 cycles for each iteration of loop.
There are a total of 8 iterations.
It takes 15 cycles from the start of the program to enter into the loop.
It takes a total 83 cycles to get the final sum in the function sum.
After popping the final sum to %rax, it clears all the array elements pushed into the stack by popping 16 times which takes a total of 48 cycles .
After summing up all the other cycles as well, we get a total of 152 cycles.

irmovq stack, %rsp	F	D	E	M	W														
call main		F	D	E	M	W													
irmovq array,%rdi			F	D	E	M	W												
irmovq $8,%rsi				F	D	E	M	W											
call sum					F	D	E	M	W										
irmovq $8,%r8						F	D	E	M	W									
irmovq $1,%r9							F	D	E	M	W								
xorq %rax,%rax								F	D	E	M	W							
andq %rsi,%rsi									F	D	E	M	W						
pushq %rax (uses forwarding)										F	D	E	M	W					
jmp test											F	D	E	M	W				
jne loop												F	D	E	M	W			
mrmovq (%rdi),%rbx													F	D	E	M	W		
nop														F	D	E	M	W	
pushq %rbx (uses forwarding)															F	D	E	M	W

paddq									F	D	E	M	W						
nop										F	D	E	M	W					
paddq (uses forwarding)											F	D	E	M	W				
addq %r8,%rdi												F	D	E	M	W			
subq %r9,%rsi													F	D	E	M	W		

Task 7

The new instruction i’m adding is I[opq] , where opq can be replaced by addq, subq, andq, xorq.
Instruction encoding for the above instructions are given as follows:

Iaddq :   icode: ifun

5	1

Isubq: icode: ifun

5	2

Iandq: icode: ifun

5	3

Ixorq: icode: ifun

5	4

The whole instruction is 10 bytes long which is as follows:

Iopq D(rA), rB :

5	fn	rA	rB	D

Eg: The instruction Iaddq D(%rax), %rbx adds the value present in %rbx with the value present in the memory address (%rax+D) and moves the final sum to %rbx.

Since the instruction requires accessing one of the operands from the data memory to perform the operation , it needs to read value from the memory prior to performing arithmetic/logical operation.
In the existing Y86 architecture ,we have the ALU unit to perform arithmetic/logical operations in the Execute stage which comes before the Memory stage.
Inorder to perform the above defined new instruction, we first need to read value from the memory and later perform the given arithmetic/logical operation .
So,we need to add another ALU unit in the Memory stage in addition to the data memory block which performs the operation on the value read from the data memory.
The newly added ALU unit is enabled only when it detects the instruction encoding of the new instruction.Otherwise, it stays in disable state for all other instructions.
The resulting output from this ALU unit is indicated as ValE1 .It also sets the condition codes.
Both ALU units present in the Execute stage and Memory stage have a common CC block in the Execute unit(assume there's a direct wire connection from the ALU unit in the Memory stage with the CC block in the Execute stage).
In case if both ALU units performed arithmetic/logical operations in the same cycle, then ValE will be considered in updating condition codes because the instruction in the Execute stage is the most recent arithmetic/logical instruction then the one in the Memory stage.

let's describe how the new instruction executes in various stages.

Stage	**lopq D(rA), rB**

Fetch	icode : ifun ← M_1[PC]
	rA : rB ← M_1 [PC + 1]
	valC ← M_8 [PC + 2]
	valP ← PC + 10

Decode	valA ← R[rA]
	valB ← R[rB]

Execute	valE ← valA +valC

Memory	valM ← M_8[valE]
	valE1 ← valM OP valB
	Set CC

Write back	R[rB] ← valE1
PC update	PC ← valP