Computer Systems Organisation (CS2.201)

Summer 2021, IIIT Hyderabad

Assignment 2

Name: Abhinav S Menon
Roll No.: 2020114001

Instruction

The instruction assigned is Instruction 2, *i.e.*, `pOPq`.
This instruction will pop the top two values off the stack, execute the operation on them, and push the result back on the stack. It was assumed they would be popped as the values below the top of the stack are not supposed to be accessible, by nature of the stack data structure.

Task 1

The encoding for this intruction will be one byte long.

Byte 0

The icode part of the instruction will be `0xc`.
The ifun part is the same as that operation's – `0x0` for `addq`, `0x1` for `subq`, `0x2` for `andq`, and `0x3` for `xorq`.

Task 2

```
.pos 0x0
irmovq stack,  %rsp     # initialise stack pointer
irmovq array,  %rdi     # base address of array
call   main
halt

.align 8
array:                  # initialise with [0,1,2,3,4,5,6,7]
.quad 0x00
.quad 0x01
.quad 0x02
.quad 0x03
.quad 0x04
.quad 0x05
.quad 0x06
.quad 0x07

main:
```

```
irmovq $0,     %rcx     # counter: i = 0
irmovq $8,     %rdx     # increment

loop:                   # pushes all array entries on stack
irmovq $64,    %rbx     # used to check termination condition
addq   %rcx,   %rdi     # now %rdi has address of A[i]
mrmovq (%rdi), %rax     # move A[i] to %rax
pushq  %rax             # push A[i] onto stack
subq   %rcx,   %rdi     # now %rdi has base address again
addq   %rdx,   %rcx     # increment %rcx: i++
subq   %rcx,   %rbx     # check if %rcx == 64: i == 8
jne    loop             # if it is, exit

irmovq $0,     %rcx     # counter: i = 0
irmovq $1,     %rdx     # increment

sum:                    # sums up top 8 values on stack
irmovq $7,     %rbx     # used to check termination condition
paddq                   # pops top two off stack, adds and pushes back
addq   %rdx,   %rcx     # increment %rcx: i++
subq   %rcx,   %rbx     # check if %rcx == 7: i == 7
jne    sum              # if it is, exit

popq   %rax             # pop sum off stack into %rax for returning
ret

.pos 0x300              # initialise stack
stack:
```

Task 3

The memory dump of the above code is:

```
0x000:
0x000: 30f40000000000000300
0x00a: 30f70000000000000020
0x014: 800000000000000060
0x01d: 00

0x020:
0x020:
0x020: 0x0000000000000000
0x028: 0x0000000000000001
0x030: 0x0000000000000002
0x038: 0x0000000000000003
0x040: 0x0000000000000004
```

```
0x048: 0x0000000000000005
0x050: 0x0000000000000006
0x058: 0x0000000000000007

0x060:
0x060: 30f10000000000000000
0x06a: 30f20000000000000008

0x074:
0x074: 30f30000000000000040
0x07e: 6017
0x080: 50700000000000000000
0x08a: a00f
0x08c: 6117
0x08e: 6021
0x090: 6113
0x092: 740000000000000074

0x09b: 30f10000000000000000
0x0a5: 30f20000000000000001

0x0af:
0x0af: 30f30000000000000007
0x0b9: c0
0x0ba: 6021
0x0bc: 6113
0x0be: 7400000000000000af

0x0c8: b00f
0x0ca: 90

0x300:
0x300:
```

Task 4

This instruction will take three cycles to complete on the sequential architecture as it has to manipulate the memory three times.

The first instance of the instruction is at `0x0b9`. At this point, the stack top is `0x02b8` in `%rsp` (the stack top). The top two values on the stack are `0x07` (at `0x02b8`) and `0x06` (at `0x02c0`).

Stage	Cycle 1	Cycle 2	Cycle 3
Fetch	`icode:ifun` $\leftarrow$ M_1[`0x0b9`] = `c:0`		`valP` $\leftarrow$ `PC` + 1 = `0x0ba`
Decode	`valA` $\leftarrow$ `R[%rsp]` = `0x2b8`	`valA` $\leftarrow$ `R[%rsp]` = `0x2c0`	`valA` $\leftarrow$ `valM`$_1$ = `0x07` `valB` $\leftarrow$ `valM`$_2$ = `0x06`
Execute	`valE`$_1$ $\leftarrow$ `valA` + 8 = `0x2c0`	`valE`$_2$ $\leftarrow$ `valA` + 0 = `0x2c0`	`valE` $\leftarrow$ `valA` + `valB` = `0x0d`
Memory	`valM`$_1$ $\leftarrow$ `M[valA]` = `0x07`	`valM`$_2$ $\leftarrow$ `M[valA]` = `0x06`	`M[valE`$_2$`]` $\leftarrow$ `valE` = `0x0d`
Write-back	`R[%rsp]` $\leftarrow$ `valE`$_1$ = `0x2c0`		
PC Update			`PC` $\leftarrow$ `valP` = `0x0ba`

Task 5

The first 20 cycles of the above program on SEQ are as follows

Cycle	`PC`	`CC`	`%rax`	`%rcx`	`%rdx`	`%rbx`	`%rdi`	`%rsp`	Memory
1	0x00	000	0x0	0x0	0x0	0x0	0x0	0x300	-
2	0x0a	000	0x0	0x0	0x0	0x0	0x20	0x300	-
3	0x14	000	0x0	0x0	0x0	0x0	0x20	0x2f8	`M[0x2f8]` $\leftarrow$ `0x60`
4	0x60	000	0x0	0x0	0x0	0x0	0x20	0x2f8	-
5	0x6a	000	0x0	0x0	0x8	0x0	0x20	0x2f8	-
6	0x74	000	0x0	0x0	0x8	0x40	0x20	0x2f8	-
7	0x7e	000	0x0	0x0	0x8	0x40	0x20	0x2f8	-
8	0x80	000	0x0	0x0	0x8	0x40	0x20	0x2f8	-
9	0x8a	000	0x0	0x0	0x8	0x40	0x20	0x2f0	`M[0x2f0]` $\leftarrow$ `0x0`
10	0x8c	000	0x0	0x0	0x8	0x40	0x20	0x2f0	-
11	0x8e	000	0x0	0x8	0x8	0x40	0x20	0x2f0	-
12	0x90	000	0x0	0x8	0x8	0x38	0x20	0x2f0	-
13	0x92	000	0x0	0x8	0x8	0x38	0x20	0x2f0	-
14	0x74	000	0x0	0x8	0x8	0x40	0x20	0x2f0	-
15	0x7e	000	0x0	0x8	0x8	0x40	0x28	0x2f0	-
16	0x80	000	0x1	0x8	0x8	0x40	0x28	0x2f0	-
17	0x8a	000	0x1	0x8	0x8	0x40	0x28	0x2e8	`M[0x2e8]` $\leftarrow$ `0x1`
18	0x8c	000	0x1	0x8	0x8	0x40	0x20	0x2e8	-

Cycle	PC	CC	%rax	%rcx	%rdx	%rbx	%rdi	%rsp	Memory
19	0x8e	000	0x1	0x10	0x8	0x40	0x20	0x2e8	-
20	0x90	000	0x1	0x10	0x8	0x30	0x20	0x2e8	-

Task 6

Wherever data forwarding is made use of, the corresponding values being shifted are marked in the columns.

The processor is assumed to follow an "always-taken" branch prediction strategy. This strategy goes correctly in the below analysis.

Load-use hazards occur twice in the first 20 cycles.

The program as a whole takes 144 cycles to execute (assuming that `pOPq` takes 4 cycles). If `pOPq` takes n cycles, the program takes $116 + 7n$ cycles to complete.

Task 7

One such instruction can be `OPmovl rA, rB, rC`. This would calculate `R[rA] OP R[rB]` and put the result into `rC` without modifying the contents `rA` or `rB`.

A new control signal would be needed for this as the instruction has three fields, so three register IDs need to be read in the fetch stage. Two of the values have to be read from the register file in decode, while the third would contain the destination register to use in the write-back stage.

Pipelined Cycle Diagram

Instruction	Location	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
`irmovq stack, %rsp`	**0x0**	**F**	**D**	**E**	**M** M_valE = 0x300	**W**																		
`irmovq array, %rdi`	**0xa**		**F**	**D**	**E**	**M**	**W**																	
`call main`	**0x14**			**F**	**D** valB = M_valE = 0x300	**E**	**M**	**W**																
`irmovq $0, %rcx`	**0x60**				**F**	**D**	**E**	**M**	**W** W_valE = 0x0															
`irmovq $8, %rdx`	**0x6a**					**F**	**D**	**E**	**M**	**W**														
`irmovq $64, %rbx`	**0x74**						**F**	**D**	**E**	**M**	**W**													
`addq %rcx, %rdi`	**0x7e**							**F**	**D** valA = W_valE = 0x0	**E** e_valE = 0x20	**M**	**W**												
`mrmovq (%rdi), %rax`	**0x80**								**F**	**D** valA = e_valE = 0x20	**E**	**M** m_valM = 0x0	**W**											
`bubble`												**E**	**M**	**W**										
`pushq %rax`	**0x8a**									**F**	**D**	**D** valA = m_valM = 0x0	**E**	**M**	**W**									
`subq %rcx, %rdi`	**0x8c**										**F**	**F**	**D**	**E**	**M**	**W**								
`addq %rdx, %rcx`	**0x8e**												**F**	**D**	**E** e_valE = 0x28	**M**	**W**							
`subq %rcx, %rbx`	**0x90**													**F**	**D** valA = e_valE = 0x28	**E**	**M**	**W**						
`jne loop`	**0x92**														**F**	**D**	**E**	**M**	**W**					
`irmovq $64, %rbx`	**0x74**															**F**	**D**	**E**	**M**	**W**				
`addq %rcx, %rdi`	**0x7e**																**F**	**D**	**E** e_valE = 0x28	**M**	**W**			
`mrmovq (%rdi), %rax`	**0x80**																	**F**	**D** valA = e_valE = 0x28	**E**	**M** m_valM = 0x1	**W**		
`bubble`																					**E**	**M**	**W**	
`pushq %rax`	**0x8a**																		**F**	**D**	**D** valA = m_valM = 0x1	**E**	**M**	**W**
`subq %rcx, %rdi`	**0x8c**																			**F**	**F**	**D**	**E**	**M**
`addq %rdx, %rcx`	**0x8e**																					**F**	**D**	**E** e_valE = 0x30
`subq %rcx, %rbx`	**0x90**																						**F**	**D** valA = e_valE = 0x30