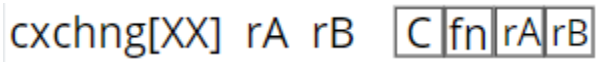
Project
Architectural Exploration

- **Parth Maradia**

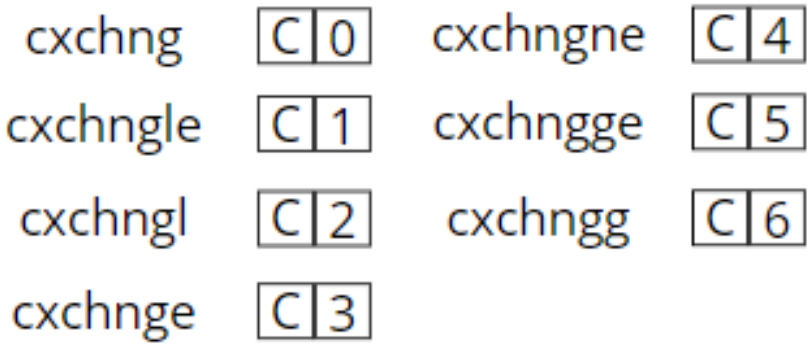
Instruction: cxchng[XX] rA , rB :- Swap the values in rA and rB. The swap only happens if the condition [XX] is satisfied. Here [XX] can be any one of the jump conditions given in Figure 4.3 of chapter 4. The registers can be any of the registers from Figure 4.4 of chapter 4. Example - cxchngge %eax, %ebx [Explanation - Exchange %eax with %ebx only of %ebx >= %eax].

Task-1

Instruction Encoding for Instruction - 3 is:-



Here [XX] denotes the conditional branch as follows:-



Ifun : 0 is the default function which exchanges the two values without any conditions.
Ifun : 1 , 2 … 6 call the function less than or equal to, less than , equal to , not equal to , greater than or equal to , and greater than respectively.

Task-2

Problem 3: Given an array of 8 unsigned integers. Swap the leftmost 4 values with the rightmost 4 values in the array. The swap will only happen if the element on the right is greater than the element on the left.

```
# 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            # 64 bits = 8 bytes each
array:
    .quad 0x01234abc
    .quad 0x00c000c000c0
    .quad 0x0b000b000b00
    .quad 0xa000a000a000
    .quad 0x01234567
    .quad 0x01234598
    .quad 0x0abcde
    .quad 0x000d000d000d

main:
    irmovq array,%rdi
    irmovq $4,%rsi
    call swapArr        # swapArr(array, 8)
    ret

 # void swapArr(long *arr, long count)

 # &arr[0] in %rdi, count in %rsi

swapArr:
    irmovq $8,%r8           # Constant 8
    irmovq $1,%r9           # Constant 1
```

```
    irmovq $56, %r10         # Constant 56
    irmovq $0 , %r13         # Constant 0
    xorq %rax , %rax
    andq %rsi,%rsi           # Set CC
    jmp test                 # Goto test

loop:
    rrmovq %rdi, %r11        # Get &arr[0] for lhs of compare
    rrmovq %rdi, %r12        # Get &arr[0] for rhs of compare

    addq %r13 , %r11         # get &arr[l] for lhs of compare
    addq %r10 , %r12         # get &arr[r] for rhs of compare

    mrmovq (%r11), %rcx      # contains arr[l] for lhs of compare
    mrmovq (%r12), %rdx      # contains arr[r] for rhs of compare

    cxchngg %rcx , %rdx     # exchange only occurs if arr[l] < arr[r]

    rmmovq %rcx , (%r11)     # updating value in the arr[l]
    rmmovq %rdx , (%r12)     # updating value in the arr[r]

    addq %r8 , %r13         # l++
    subq %r8 , %r10         # r--

    subq %r9,%rsi            # count--. Set CC

test:
    jne loop                 # Stop when 0
    ret                      # Return

 # Stack starts here and grows to lower addresses
.pos 0x200
stack:
```

Some part of this code was taken from the Textbook Figure 4.7

I assumed the array to be in hexadecimal so that it is easier to write it in little endian format.

Task-3

Write the memory dump of the instructions written by you for the problem in the previous task (i.e Task 2). Refer to Figure 4.8 (left part) of chapter 4 from the book. Assume the starting address is 0x0000.

```
                            | # Execution begins at address 0
0x0000:                     | .pos 0            # We assume stack begins at 0
                            |
0x0000: 30f40002000000000000| irmovq stack, %rsp  # Set up stack pointer
                            |
0x000a: 8058000000000000    | call main           # Execute main program
                            |
0x0013: 00                  | halt                # Terminate program
                            |
                            | # Array of 8 elements
0x0014:                     | .align 8            # objdump in little endian
0x0018:                     | array:
0x0018: 0d000d000d          |     .quad 0x000d000d000d
0x0020: c000c000c0          |     .quad 0x00c000c000c0
0x0028: 000b000b000b        |     .quad 0x0b000b000b00
0x0030: 00a000a000a0        |     .quad 0xa000a000a000
0x0038: 67452301            |     .quad 0x01234567
0x0040: 98452301            |     .quad 0x01234598
0x0048: debc0a              |     .quad 0x0abcde
0x0050: bc4a2301            |     .quad 0x01234abc
                            |
0x0058:                     | main:
0x0058: 30f71800000000000000|     irmovq array,%rdi
0x0062: 30f60400000000000000|     irmovq $4,%rsi
0x006c: 8076000000000000    |     call swapArr        # swapArr(array, 8)
0x0075: 90                  |     ret
                            |
                            |  # void swapArr(long *arr, long count)
                            |  # &arr[0] in %rdi, count in %rsi
                            |
0x0076:                     | swapArr:
0x0076: 30f80800000000000000|     irmovq $8,%r8           # Constant 8
0x0080: 30f90100000000000000|     irmovq $1,%r9           # Constant 1
0x008a: 30fa3800000000000000|     irmovq $56, %r10        # Constant 56
```

```
0x0094: 30fd0000000000000000    |     irmovq $0 , %r13        # Constant 0
0x009e: 6300                    |     xorq %rax , %rax
0x00a0: 6266                    |     andq %rsi,%rsi          # Set CC
0x00a2: 70e300000000000000      |     jmp test                # Goto test
                                |
                                |
                                |
0x00ab:                         | loop:
0x00ab: 207b                    |     rrmovq %rdi, %r11       # Get &arr[0] for
lhs of compare
0x00ad: 207c                    |     rrmovq %rdi, %r12       # Get &arr[0] for
rhs of compare
                                |
0x00af: 60db                    |     addq %r13 , %r11        # get &arr[l] for
lhs of compare
0x00b1: 60ac                    |     addq %r10 , %r12        # get &arr[r] for
rhs of compare
                                |
0x00b3: 501b0000000000000000    |     mrmovq (%r11), %rcx     # contains arr[l]
for lhs of compare
0x00bd: 502c0000000000000000    |     mrmovq (%r12), %rdx     # contains arr[r]
for rhs of compare
                                |
0x00c7: C6bc                    |     cxchngg %rcx , %rdx     # exchange only
occurs if arr[l] < arr[r]
                                |
0x00c9: 401b0000000000000000    |     rmmovq %rcx , (%r11)    # updating value in
the arr[l]
0x00d3: 402c0000000000000000    |     rmmovq %rdx , (%r12)    # updating value in
the arr[r]
                                |
0x00dd: 608d                    |     addq %r8 , %r13        # l++
0x00df: 618a                    |     subq %r8 , %r10        # r--
                                |
0x00e1: 6196                    |     subq %r9,%rsi           # count--. Set CC
                                |
0x00e3:                         | test:
0x00e3: 74ab00000000000000      |     jne loop                # Stop when 0
0x00ec: 90                      |     ret                     # Return
                                |
```

```
                            |   # Stack starts here and grows to lower
addresses
                            |
0x00ed:                     | .pos 0x200
0x0200:                     | stack:
```

Task-4

Create the Fetch, Decode, Execute, Memory, Write Back, PC Update Table. This table will capture how your instruction gets implemented across these stages on the Y86 architecture. My code executes the cxchngg instruction at 0x00b6: C6bc

PC = 0x00c7

STAGE	Cxchngg rA, rB
Fetch	Icode:ifun ← $\mathbf{M_1}$[PC] rA : rB ← $\mathbf{M_1}$[PC+1] valP ← PC+2
Decode	valA ← R[rA] valB ← R[rB]
Execute	Cnd ← Cond(CC , ifun)
Memory	
Write back	R[rA] ← Cnd? valB : valA R[rB] ← Cnd? valA : valB
PC update	PC ← valP

The Fetch stage, first byte of cxchngg has icode: ifun part and the next byte has the rA rB values respectively. Since no other inputs are required the valP is set to PC + 2.
In Decode the values from rA and rB are read and stored in valA and valB resp.
In Execute stage valA and valB are compared and Cnd is set based on CC , ifun and the comparison of valA and valB.
In the Write Back stage the values of register rA and rB are updated based on the Cnd.
In PC update stage, PC is set to valP , i.e. incremented by 2.

Task-5

Starting from the first instruction in your Y86 code of your program in Task 2, trace 20 cycles of execution on the sequential Y86 architecture. For each cycle, report the value of the program counter, condition code registers, the general purpose registers, stack pointer (if it is used), and any modified addresses in memory.

Cycle	Modifications	SF	ZF	OF
1:	0x0000: irmovq stack, %rsp # rsp ← 0x0200	0	0	0
2:	0x000a: call main	0	0	0
3:	0x0058: irmovq array,%rdi # rdi ← 0x0018	0	0	0
4:	0x0062: irmovq $4,%rsi # rsi ← 0x0004	0	0	0
5:	0x006c: call swapArr	0	0	0
6:	0x0076: irmovq $8,%r8 # r8 ← 0x0008	0	0	0
7:	0x0080: irmovq $1,%r9 # r9 ← 0x0008	0	0	0
8:	0x008a: irmovq $56,%r10 # r10 ← 0x0008	0	0	0
9:	0x0094: irmovq $0,%r13 # r13 ← 0x0008	0	0	0
10:	0x009e: xorq %rax , %rax # rax ← 0x0000	0	1	0
11:	0x00a0: andq %rsi,%rsi # Set CC	0	0	0
12:	0x00a2: jmp test	0	0	0
13:	0x00e3: jne loop	0	0	0
14:	0x00ab: rrmovq %rdi, %r11 # r11 ← 0x0018	0	0	0
15:	0x00ad: rrmovq %rdi, %r12 # r12 ← 0x0018	0	0	0
16:	0x00af: addq %r13 , %r11 # r11 ← 0x0018	0	0	0
17:	0x00b1: addq %r10 , %r12 # r12 ← 0x0050	0	0	0
18:	0x00b3: mrmovq (%r11), %rcx # rcx ← 0x0d000d000d	0	0	0

19:	0x00bd: mrmovq (%r12), %rdx # rdx ← 0x01234abc	0	0	0
20:	0x00c7: cxchngg %rcx , % rdx # swaps if conditions met	0	0	0

The updated PC is written next to the subsequent instruction and thus the PC after 20th cycle is executed will be 0x00c9 , since cxchngg takes 2 bytes worth of instructions.
In the 10th instruction Zero Flag is set to 1 since we perform XOR operation on rax with itself. Hence to reset the flag to 0 , in 11th instruction we perform AND operation on rsi with itself, (no changes in value of rsi , hence all three flag set to 0 0 0).

Task-6

Find the number of cycles required by your program to execute. Draw the pipelined cycle diagram for the first 20 instructions in your program.

For calculating the total number of cycles,
We need to find the total number of instructions and bubbles/stalls.

In the initialization part, main and, swapArr, we use a total of 12 instructions with no bubbles/stalls, since no data hazard takes place. Hence A = 12.

There is no bubble or stall required in jmp and jne loop because no hazard takes place. Hence B = 2.

In the loop: there is 6 bubble and 13 instructions in the loop which repeats itself 4 times. Hence C = 19 * 4 = 76.

The code ends after 1 ret instruction. Hence D = 1.

Therefore Total number of Cycles required = 12 + 2 + 76 + 1 = **91**

My program takes exactly 91 cycles to execute. 11 cycles in the initial stage and the rest while looping and swapping the array using cxchngg instruction.

	Instructions	Comments , Clock Cycle ->	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
2	irmovq stack, %rsp	init stack to 0x200	F	D	E	M	W																						
3	call main			F	D	E	M	W																					
4	irmovq array,%rdi				F	D	E	M	W																				
5	irmovq $4,%rsi					F	D	E	M	W																			
6	call swapArr						F	D	E	M	W																		
7	irmovq $8,%r8							F	D	E	M	W																	
8	irmovq $1,%r9								F	D	E	M	W																
9	irmovq $56,%r10									F	D	E	M	W															
10	irmovq $0,%r13										F	D	E	M	W														
11	xorq %rax , %rax											F	D	E	M	W													
12	andq %rsi,%rsi	Sets CC -> 000											F	D	E	M	W												
13	jmp test													F	D	E	M	W											
14	jne loop														F	D	E	M	W										
15	rrmovq %rdi, %r11															F	D	E	M	W									
16	rrmovq %rdi, %r12																F	D	E	M	W								
17	bubble	added bubble in fetch stage																F											
18	addq %r13 , %r11																		F	D	E	M	W						
19	addq %r10 , %r12																			F	D	E	M	W					
20	mrmovq (%r11), %rcx																				F	D	E	M	W				
21	mrmovq (%r12), %rdx																					F	D	E	M	W			
22	bubble	added bubble in fetch stage																					F						
23	bubble	added bubble in fetch stage																						F					
24	cxchngg %rcx , % rdx																								F	D	M	E	W

If not visible clearly, you can check. Project Pipeline

There forwarding is used in few cycles in order to compute the values in decode or Execute Stage.
One bubble is added after rmmovq %rdi , %r12 , to prevent data hazard in subsequent instructions. We pair this bubble with forwarding to increasing its efficiency.
Two bubbles are added after the mrmovq (%r12) , %rdx , to prevent data hazard in cxchngg (which requires the values of rA and rB for operation) , we pair the bubbles with forwarding to increase its efficiency.

Task-7

Come up with any new Y86 Instruction, that will require you to add new hardware block/control signals to the Y86 architecture. Briefly describe the instruction and how it requires a new hardware block.

- The new Y86 Instruction that i would like to add to Y86 architecture is bitwise OR instruction.
- The bitwise OR instruction is a very useful and versatile instruction and can be used in combinations with already existing bit operators like And(&) and XOR(^) to make new operations.
- Bitwise OR instruction will be given instruction encoding as follows:
    - Icode : D ; Ifun : 0
    - rA : register 1 ; rB : register 2
    - No other memory / destination required.

- Hence , Bitwise OR will take 2 Byte memory (i.e.) increase the PC counter by 2 since it only takes icode , ifun , rA and rB as input to operate.
- The Bitwise OR operator will conduct a bitwise OR operations between rA and rB and store the calculated output in rB , i.e. rB is the destination register.
- The hardware part of bitwise OR instruction is easy to implement. We can built its ALU using 2 bit OR GATE. We will use 64 or 32 (based on the device) such 2 bit OR GATES to compute bitwise OR operation on two numbers stored in the input registers.
- After computing the value it will be stored in rB .
- Since overflow is not possible in bitwise OR operation the overflow flag will remain 0. The sign flag and zero flag are checked and raised according to output.
- The bitwise OR instruction will require exactly 5 clock cycles to operate and necessary errors would be handled within the pre-existing Y86 architecture. We are just adding additional ALU hardware. The rest of the hardware will remain same .

The final bitshift OR instruction in action will look something like this:
Eg.

```
0x0020  orq  %rdx , %rcx                # rdx ← 10 (1010) , rcx ← 13 (1101)
0x0022
```