# CSO Assignment 2

| Column | |
|---|---|
| Course | CSO |

## CSO Assignment 2

### Instruction 3

> **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].

### Tasks

#### Task 1: Instruction Encoding

Our instruction requires the following parameters to be supplied to it.

1. Target registers
2. The swap condition [ne, ge, le, g, l, e]

In essence, it is pretty similar to the encoding of the conditional move instructions.

We pick the following encoding.

1. `0xC` (12 in decimal) for the instruction code. [4 bits]
2. Integers in the range `[0, 5]` for the function code (swap condition specifier). [4 bits]
3. The second byte contains the register specifiers for `rA` and `rB` [4 bits each. Total 1 byte]

We adopt the following mapping for function code.

**Function codes**

| Function code | Swap | Condition |
|---|---|---|

| Function code | Swap | Condition |
|---|---|---|
| 0 | cxchngle | Less than or equal to |
| 1 | cxchngl | Less than |
| 2 | cxchnge | Equal to |
| 3 | cxchnglne | Not equal to |
| 4 | cxchngge | Greater than or equal to |
| 5 | cxchngg | Greater than |

Our final instruction encoding:


The total instruction size is 2 bytes.

### Task 2: Array swaps

> 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. For example, element 0 will be swapped with element 7 if e[7] >
> e[0], element 1 with element 6 if e[6] > e[1] and so on. Along with other instructions you
> have to use **cxchng[XX] rA , rB** instruction in this problem.

**Code:**

```
# Simple setup routine
# Execution starts here. We setup stack and call main.
.pos 0x0
irmovq  stack, %rsp
call main
halt

# Stores the contents of our array
.align 8
arr:
    .quad 0x7
    .quad 0x1
    .quad 0x2
    .quad 0x3
    .quad 0x4
```

```
    .quad 0x5
    .quad 0x6
    .quad 0x0

# Main calls our swap function
swapfn:
    # Loop does arithmetic using constants 8 & 1. Setup registers for it
    irmovq $8, %r8
    irmovq $1, %r9
loop:
    # End loop if %rdx iterations are done (3rd argument)
    je endloop
    # Pretending it's a for loop where we're iterating over i to n-1
    # move index i and n-i to registers r10 and r11
    mrmovq (%rdi), %r10
    mrmovq (%rsi), %r11
    # Call our swap instruction
    cxchngg %r10, %r11
    # Store the data back to memory
    rmmovq %r10, (%rdi)
    rmmovq %r11, (%rsi)
    # Perform loop update operations
    addq %r8, %rdi
    subq %r8, %rsi
    subq %r9, %rdx
    jmp loop
endloop:
    ret

main:
    # Setup argument registers similar to x86 conventions
    # Assume function is of the form:
    # swapfn(long long *firstelem, long long *lastelem, long long numswaps)
    irmovq arr, %rdi
    rrmovq %rdi, %rsi
    irmovq $4, %rdx
    # Setup 56 in r8 for constant addition to rsi
    irmovq $56, %r8
    addq %r8, %rsi
    # Call the swap function
    call swapfn
    ret

# Setup stack
.pos 0x200
stack:
```

## Task 3: Memory dump

```
                            | # Simple setup routine
                            | # Execution starts here. We setup stack and call main.
0x0000:                     | .pos 0x0
0x0000: 30f40002000000000000 | irmovq  stack, %rsp
0x000a: 80af00000000000000  | call main
0x0013: 00                  | halt
                            |
                            | # Stores the contents of our array
0x0014:                     | .align 8
0x0018:                     | arr:
0x0018: 07                  |     .quad 0x7
0x0020: 01                  |     .quad 0x1
0x0028: 02                  |     .quad 0x2
0x0030: 03                  |     .quad 0x3
0x0038: 04                  |     .quad 0x4
0x0040: 05                  |     .quad 0x5
0x0048: 06                  |     .quad 0x6
```

```
0x0050: 00                      |     .quad 0x0
                                |
                                | # Main calls our swap function
0x0058:                         | swapfn:
                                |     # Loop does arithmetic using constants 8 & 1. Setup registers for it
0x0058: 30f80800000000000000    |     irmovq $8, %r8
0x0062: 30f90100000000000000    |     irmovq $1, %r9
0x006c:                         | loop:
                                |     # End loop if %rdx iterations are done (3rd argument)
0x006c: 73ae00000000000000      |     je endloop
                                |     # Pretending it's a for loop where we're iterating over i to n-1
                                |     # move index i and n-i to registers r10 and r11
0x0075: 50a700000000000000000   |     mrmovq (%rdi), %r10
0x007f: 50b600000000000000000   |     mrmovq (%rsi), %r11
                                |     # Call our swap instruction
0x0089: c5ab                    |     cxchngg %r10, %r11
                                |     # Store the data back to memory
0x008b: 40a700000000000000000   |     rmmovq %r10, (%rdi)
0x0095: 40b600000000000000000   |     rmmovq %r11, (%rsi)
                                |     # Perform loop update operations
0x009f: 6087                    |     addq %r8, %rdi
0x00a1: 6186                    |     subq %r8, %rsi
0x00a3: 6192                    |     subq %r9, %rdx
0x00a5: 706c00000000000000      |     jmp loop
0x00ae:                         | endloop:
0x00ae: 90                      |     ret
                                |
0x00af:                         | main:
                                |     # Setup argument registers similar to x86 conventions
                                |     # Assume function is of the form:
                                |     # swapfn(long long *firstelem, long long *lastelem, long long numswaps)
0x00af: 30f718000000000000000   |     irmovq arr, %rdi
0x00b9: 2076                    |     rrmovq %rdi, %rsi
0x00bb: 30f204000000000000000   |     irmovq $4, %rdx
                                |     # Setup 56 in r8 for constant addition to rsi
0x00c5: 30f838000000000000000   |     irmovq $56, %r8
0x00cf: 6086                    |     addq %r8, %rsi
                                |     # Call the swap function
0x00d1: 805800000000000000      |     call swapfn
0x00da: 90                      |     ret
                                |
                                | # Setup stack
0x00db:                         | .pos 0x200
0x0200:                         | stack:
                                |
```

### Task 4: Fetch, Decode, Execute, Memory, Write Back, PC Update Table

> **Assumptions**:
> We are required to set condition codes and perform a conditional swap based on the conditional codes set in the same operation. This operation must also take 1 clock cycle and not read back on the previously saved state. Violating this principle will make pipelining impossible.
> Addressing these requirements will require some (minor) hardware / HCL additions.
> 1. At the decode stage we set the values of `dstE` and `dstM` to `rA` and `rB`.
> 2. We must perform the comparison at the execute stage to get the condition codes. So our execute stage will compute `subq`. `valE` gets set to

> `valB - valA`. This also has the corresponding effect of computing the CC's.
> 3. Next, we use the CC generated and the fn code to verify if `cxchnXX`'s condition is satisfied. If true then the 1-bit value of `cnd` is set to 1, else 0. If `cnd` is 1, we propagate it up. If it is 0, we add additional logic blocks which set the values of `dstE` and `dstM` to `0xF` to indicate no write back.
> 4. Now, our biggest hurdle. Writes to the register file are always via values `valE` (computed by the ALU at execution stage) and `valM` (data fetched from memory at the memory stage). We have also used up `valE` for computing B-A to set CC's. Hence we will need additional hardware to assist here.
> 4.1 Setting `valE` to the desired value is relatively easy. We can have another control block in the execution stage which sets the value of `valE` to `valB` if the instruction fetched is a `cxchng` instruction. This is as simple as adding a simple check in the HCL for `valE` to always be set to `valB` if the instruction is `cxchng`. Note that it does not depend on `cnd` as we are anyways cancelling any writes to the register file if `cnd` is 0.
> 4.2 Setting `valM` to `valA` is slightly more complicated. `valM` is generated in the memory stage, a stage after execution. However, we already have hardware that propagates the value of `valA` to the memory stage (for instructions such as `rmmovq` where we need to write data (`valA`) to memory. We can make use of this existing hardware to propagate `valA` to the memory stage. Here, we again add a control block that checks if the instruction is a `cxchng` instruction. If yes it sets the value of `valM` to `valA` and this is sent to the write-back stage.

> No further modifications or assumptions need to be made. This entire process can execute in one clock cycle with minor modifications. We also never require one stage to depend on the state of any value updated by a previous stage. Write-back depending on the values of `dstE` and `dstM` (set in execute depending on the CC) will either write-back the exchanged values to the register file or will leave them untouched.

**Table**

| Aa Stage | Generic cxchng | Specific cxchng |
|---|---|---|
| Fetch | icode :ifun ← M1[PC] rA :rB ← M1[PC + 1] valP ← PC + 2 | icode :ifun ← M1[0×0089] = c:5 rA :rB ← M1[0×008A] = a:b valP ← 0×0089 + 2 = 0×008B |
| Decode | valA ← R[rA] valB ← R[rB] | valA ← R[rA] = 7 valB ← R[rB] = 0 |

| Stage | Generic cxchng | Specific cxchng |
|---|---|---|
| Execute | valE ← valB - valA Cnd ← Cond(CC, ifun) | valE ← 0-7 = -7 Cnd ← Cond(010, greater) = false |
| Memory | - | - |
| Write back | If(Cnd) R[rB] ← valA If(Cnd) R[rA] ← valB | (Don't do anything as condition is false) |
| PC Update | PC ← valP | PC = 0×008B |

## Task 5: Trace of first 20 cycles of execution

**Trace**

| No | Instruction | Update(s) |
|---|---|---|
| 1 | `irmovq stack, %rsp` | # %rsp = 0×200 |
| 2 | `call main` | # M8[0×1f8] = 0×0013 # %rsp = 0×1f8 # PC = 0×00af |
| 3 | `irmovq arr, %rdi` | # %rdi = 0×0018 |
| 4 | `rrmovq %rdi, %rsi` | # %rsi = 0×0018 |
| 5 | `irmovq $4, %rdx` | # %rdx = 0×0004 |
| 6 | `irmovq $56, %r8` | # %r8 = 0×0038 |
| 7 | `addq %r8, %rsi` | # %rsi = 0×0050 # CC = 000 |
| 8 | `call swapfn` | # M8[0×1f0] = 0×00da # %rsp = 0×1f0 # PC = 0×0058 |
| 9 | `irmovq $8, %r8` | # r8 = 0×8 |
| 10 | `irmovq $1, %r9` | # %r9 = 0×1 |
| 11 | `je endloop` | # Not taken |
| 12 | `mrmovq (%rdi), %r10` | # %r10 = 0×7 |
| 13 | `mrmovq (%rsi), %r11` | # %r11 = 0×0 |
| 14 | `cxchngg %r10, %r11` | # Do nothing |
| 15 | `rmmovq %r10, (%rdi)` | # M8[0×0018] = 0×7 |
| 16 | `rmmovq %r11, (%rsi)` | # M8[0×0050] = 0×0 |
| 17 | `addq %r8, %rdi` | # %rdi = 0×20 # CC = 000 |
| 18 | `subq %r8, %rsi` | # %rsi = 0×48 # CC = 000 |
| 19 | `subq %r9, %rdx` | # %rdx = 3 # CC = 000 |
| 20 | `jmp loop` | # PC = 0×006c |

## Task 6: Pipelined cycle diagram & minimum cycles

4 + (Number of bubbles) + 3*(Number of Ret's) + (Number of Instructions)

**Minimum number of instructions:**

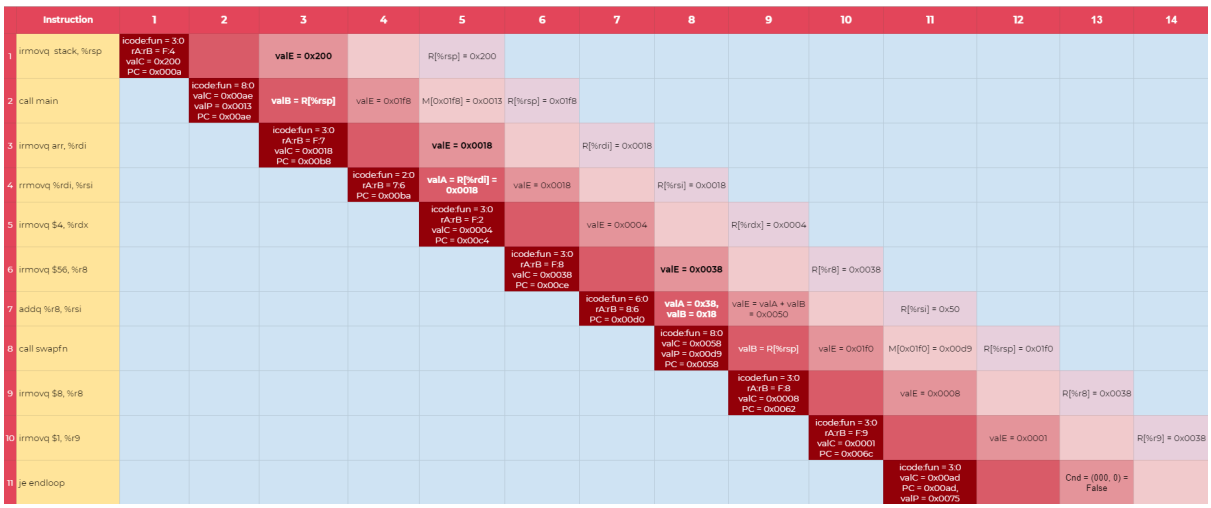
Assuming the CPU always predicts that a conditional jump is always taken and that my pipelined CPU implementation has proper data forwarding.

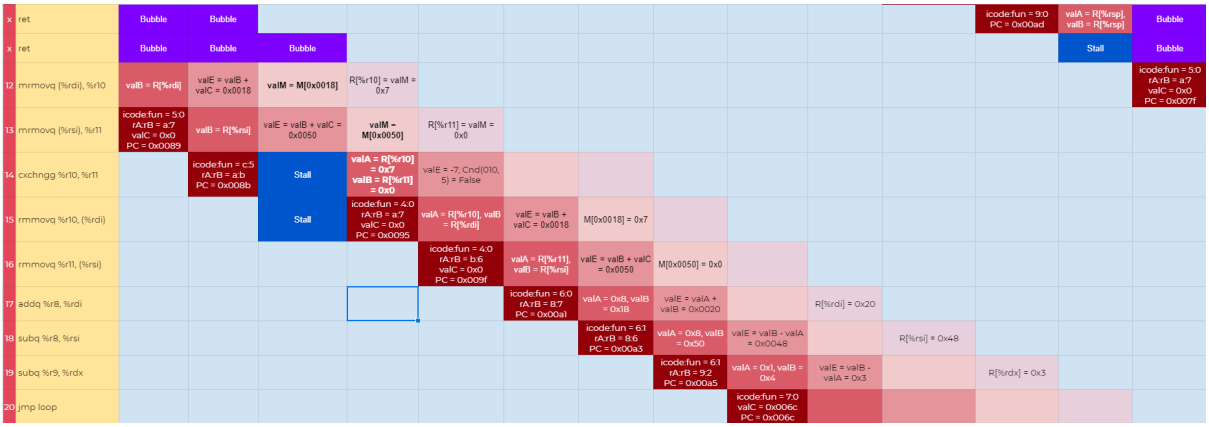
> In the initial setup, we have 3 instructions. No stalls/bubbles.
> Total used: 3

> In the main function, we have 7 instructions. No stalls/bubbles. One return which costs 3 cycles.
> Total used: 10

> In the swapfn function, we have 2 instructions before the loop.
> Assuming the predictor always predicts that the jump is taken.
> The loop runs 4 times. 3 times it will predict incorrectly. And hence I will lose 2 cycles per misprediction to bubbling as it was trying to execute incorrect instructions.
> I further lose 1 cycle to a stall per loop iteration as my `cxchngg` instruction ends up in a load hazard not solvable through data forwarding.
> Therefore in the loop, I have 4*10 instructions + 3*2 bubbles + 4*1 stalls.
> Finally, I have a return which costs 3 cycles.
> Total used: 56

Therefore, the minimum number of cycles I would require to execute my program assuming perfect data forwarding and a PC predictor which always predicts taking the jump is `56 + 10 + 3` =
**69 cycles.**






In case the attached table isn't clearly visible I've attached a link to the spreadsheet here:

https://docs.google.com/spreadsheets/d/1usPYJXljJBNytdGJD-fAysznbvwoyYWZHim9fh403TA/edit?usp=sharing

## Task 7: New instruction with the requirement of additional hardware

A new instruction that I would like to propose that requires the use of additional hardware is the `RSEED` instruction. Modern computers have an entropy pool collector. It is essentially a sensor that accumulates information about various measurements such as mouse movements, temperature changes, etc. in an effort to gather entropy. This instruction would require the use of such hardware which is capable of collecting this entropy. This process is, however, as is to be expected, fairly slow and will eat up multiple clock cycles. Hence it is important to pipeline it well.

The following is a possible implementation.

It works as a unit of its own regard. When it receives a query, which can be obtained at the fetch phase, the instruction should query the unit and leave the rest of the pipeline untouched. The unit can work parallelly on its own until it has gathered the required entropy to generate a random number. Once obtained it can write back to a register say (A). The rest of the instructions can continue to use the main pipeline as required unless there is a read to register A in which case the pipeline is stalled until the unit is able to respond back.