Department of Computer Science

Assignment #4

Shaheer Ziya
Roll: *3035946760*
Class: *2B*
Session: *2021-22*
Email: *shaheer@connect.hku.hk*

Course: *COMP 2120* – Teacher: *Dr. K P Chan, Wang Tianqi & and Xie Zerong*
Submission date: *17 Apr, 2022*

Question 1

Hand assemble the provided code and describe what the function SQ does.

Solution

The hand assembled code is as follows:

```
      SUB    R4,R4,R4          0000H: 01040404
      LD     P1,R1             0004H: 060FF01   00000078
      MOV    R1,R2             000CH: 0501002
      LD     P2,R3             0010H: 0600FF03  0000007C
L:    MOV    R1,R10            0018H: 0501000A
      CALL   SQ                001CH: 0C00FF00  00000044
      ADD    R4,R11,R4         0024H: 00040B04
      ADD    R1,R2,R1          0028H: 00010201
      SUB    R3,R1,R5          002CH: 01030105
      BNZ    L                 0040H: 0802FF00  00000018
      ST     R4,P              0038H: 0704FF00  00000080
      HLT                      0040H: 09000000
SQ:   PUSH   R12               0044H: 0A0C0000
      PUSH   R13               0048H: 0A0D0000
      LD     P1,R13            004CH: 0600FF0D  00000078
      SUB    R11,R11,R11       0054H: 010B0B0B
      MOV    R10,R12           0058H: 050A000C
L2:   ADD    R11,R10,R11       005CH: 000B0A0B
      SUB    R12,R13,R12       0060H: 010C0D0C
      BNZ    L2                0064H: 0802FF00  0000005c
      POP    R13               006CH: 0B00000D
      POP    R12               0070H: 0B00000C
      RET                      0074H: 0D000000
P1:   .WORD  1                 0078H: 00000001
P2:   .WORD  A                 007CH: 0000000A
P:    .WORD                    0080H: 00000000
```

The function SQ in the program squares whatever the value is stored in Register 10. The provided program gives the sum of the squares of the first 9 natural numbers.

$$\sum_{i=1}^{9} i^2 = 285$$

The assembly code in a high-level language might look like what follows. This is far from efficient but helps me visualize what is happening in the program.

```python
# prog2.py
# High-Level Code for prog1 (Assembly)
# Created by Shaheer Ziya on Apr 9, 2022 UTC+08 19:39

# Implement the stack using a python list
stack = []
# Initialize Registers 11, 12 & 13
R11, R12, R13 = 0, "Some Important Value", "Maybe even Garbage"
```

```python
def SQ(R10: int) -> int:
  '''Square the passed value and return it'''
  global R11, R12, R13
  stack.append(R12)
  stack.append(R13)

  R13 = 1      # P1 = 1
  R11 -= R11   # Initialize R11 to 0
  R12 = R10    # R10 is the number of loop in the main function we are in

  while (R12 != 0):
    R11 += R10  # Add R10 to R11
    R12 -= R13  # Decrement R12

  R13 = stack.pop()
  R12 = stack.pop()

  return R11


def main():
  # Initialize Registers with neccesary values
  R4 = 0     # P1 = 0
  R1 = 1     # P2 = 1
  R2 = R1    # R2 = 1
  R3 = 0xA

  R11 = 0

  # R5 determines how many iterations left
  R5 = R3 - R1
  # Condition in while loop acts as BNZ instruction
  while (R5 != 0):
    R10 = R1
    R11 = SQ(R10)
    R4 += R11 # Accumulate the results of each iteration (R11) into R4
    R1 += R2       # Increment R1
    R5 = R3 - R1 # Update R5

  # Store the final result in the last word of the program
  P = R4

  # Output the result for convenience
  print(f"The final value is {P}")

main()
```

Question 2

Describe the data transfer and/or transformation sequences involved in the execution of the **CALL** and **RET** instructions

Solution

Call

The format of the CALL operation is such that after reading it, the program counter is made to point towards the address of where the function begins. In our example, absolute addressing is employed, such that the word following the CALL operation is the address of where the called function begins.

So, currently in the execution process, the Program Counter (PC) points the word in the program which is the address to the first line of the function.

$$\begin{gathered} \text{A} \leftarrow \text{PC} \\ \text{C} \leftarrow \text{ALU} \leftarrow \text{A} \\ \text{MAR} \leftarrow \text{C} \\ \text{MBR} \leftarrow \text{mem[MAR]} \end{gathered}$$

After this sequence, the memory buffer register (MBR) holds the address to the first line of the function.

$$\begin{gathered} \text{A} \leftarrow \text{MBR} \\ \text{C} \leftarrow \text{ALU} \leftarrow \text{A} \\ \text{MAR} \leftarrow \text{C} \\ \text{TEMP} \leftarrow \text{MAR} \\ \text{PC++} \end{gathered}$$

At this moment in the program execution, the address to the first line of the function has been moved from the MBR to the TEMP register by moving through the ALU to the MAR and finally in its destination TEMP.

Additionally, the PC has been incremented such that it points to the next instruction after the most recent function call.

$$\begin{gathered} \text{SP - -} \\ \text{A} \leftarrow \text{SP} \\ \text{C} \leftarrow \text{ALU} \leftarrow \text{A} \\ \text{MAR} \leftarrow \text{C} \end{gathered}$$

The stack pointer (SP) points to the object at the top of the system stack. First we decrement the stack pointer so that it points to the empty space above the top of the stack.

We then move the stack pointer through the ALU to the memory address register such that it contains the stack pointer now.

$$
\begin{gathered}
\text{A} \leftarrow \text{PC} \\
\text{C} \leftarrow \text{ALU} \leftarrow \text{A} \\
\text{MBR} \leftarrow \text{C} \\
\text{mem[MAR]} \leftarrow \text{MBR}
\end{gathered}
$$

Here we write the current value of PC (i.e. where to start executing next after we're done with the function and return from it) to the top of the stack. Recall that MAR holds the stack pointer, pointing to an empty space at the top of the stack. So we simply move the content of PC to MBR and push it to the stack.

$$
\begin{gathered}
\text{MAR} \leftarrow \text{TEMP} \\
\text{A} \leftarrow \text{MAR} \\
\text{C} \leftarrow \text{ALU} \leftarrow \text{A} \\
\text{PC} \leftarrow \text{C}
\end{gathered}
$$

Finally, recall that the TEMP register holds the address to the first line of the function. We move this address to the program counter.

At this stage, the top of the stack holds the address to the instruction after we exit or return from the function. Furthermore, the PC holds the address to the first line in the function so until a return statement is encountered we will continue executing the function body.

Return

Notice that when we encounter the return statement, the top of the stack pointer (SP) will be pointing towards the instruction following the function call.

$$
\begin{gathered}
\text{A} \leftarrow \text{SP} \\
\text{C} \leftarrow \text{ALU} \leftarrow \text{A} \\
\text{MAR} \leftarrow \text{C} \\
\text{MBR} \leftarrow \text{mem[MAR]} \\
\text{SP++}
\end{gathered}
$$

Therefore, we move the stack pointer into the memory address register so that we can read the value at the top of the stack (in memory).

We read the top of the stack and store its value into the MBR. We then increment the stack pointer so that it points to the value now on top of the stack.

$$
\begin{gathered}
\text{A} \leftarrow \text{MBR} \\
\text{C} \leftarrow \text{ALU} \leftarrow \text{A} \\
\text{PC} \leftarrow \text{C}
\end{gathered}
$$

Finally we move the contents of MBR (the address of instruction to execute just after the latest function call) to the Program Counter. Hence we have exited/returned from the function and continue executing in the scope from which the function was called.

Question 3

Modify the program in Q1 so that the following sum is calculated instead:

$$\sum_{i=1}^{9}(-1)^{i+1}\cdot i$$

Solution

```
        SUB     R4,R4,R4          0000H: 01040404
        LD      P1,R1             0004H: 060FF01   00000084
        MOV     R1,R2             000CH: 0501002
        LD      P2,R3             0010H: 0600FF03  00000088
   L:   MOV     R1,R10            0018H: 0501000A
        CALL    SQ                001CH: 0C00FF00  00000044
        ADD     R4,R11,R4         0024H: 00040B04
        ADD     R1,R2,R1          0028H: 00010201
        SUB     R3,R1,R5          002CH: 01030105
        BNZ     L                 0040H: 0802FF00  00000018
        ST      R4,P              0038H: 0704FF00  0000008C
        HLT                       0040H: 09000000
  SQ:   PUSH    R12               0044H: 0A0C0000
        PUSH    R13               0048H: 0A0D0000
        SUB     R11,R11,R11       004CH: 010B0B0B
        MV      R11,R12           0050H: 050B000C
        LD      P1,R13            0054H: 0600FF0D  00000084
        AND     R10,R13,R13       005CH: 030A0D0D
        BNZ     F1                0060H: 0802FF00  00000074
        SUB     R12,R10,R11       0068H: 010C0A0B
        BR      F2                006CH: 0800FF00  00000078
  F1:   MV      R10,R11           0074H: 050A000B
  F2:   POP     R13               0078H: 0B00000D
        POP     R12               007CH: 0B00000C
        RET                       0080H: 0D000000
  P1:   .WORD   1                 0084H: 00000001
  P2:   .WORD   A                 0088H: 0000000A
  P:    .WORD                     008CH: 00000000
```

Only the SQ function (and the addresses of the words P1,P2 & P) have been modified.
The SQ function initializes R11 to zero, sets R12 to zero & R13 to `00...01`

We then perform use bitwise AND between the given number (stored in R10) and `00...01` (stored in R13) to check if the last digit is 1 or not (which is equivalent to checking if the number is odd or not). If the number is odd, we branch to F1, where we return the number as is (after we've popped R12 & R13 back from the stack). Otherwise we return the inverse of that number by calculating 0 - x, where x is that number. The output is accumulated in R4 and reported to us at the end just like in the first program.