THE UNIVERSITY OF HONG KONG
Department of Computer Science
COMP2120 Computer Organization
Assignment 2

Due Date: March 11, 11pm

Consider a simple 32-bit processor with the data path as shown in fig. 1. The processor has 32 general purpose registers. There are 3 buses, S1-bus, S2-bus and D-bus connecting the registers for data movement. The register files has 2 read ports and 1 write port (i.e. it can perform 2 read and 1 write at the same time).

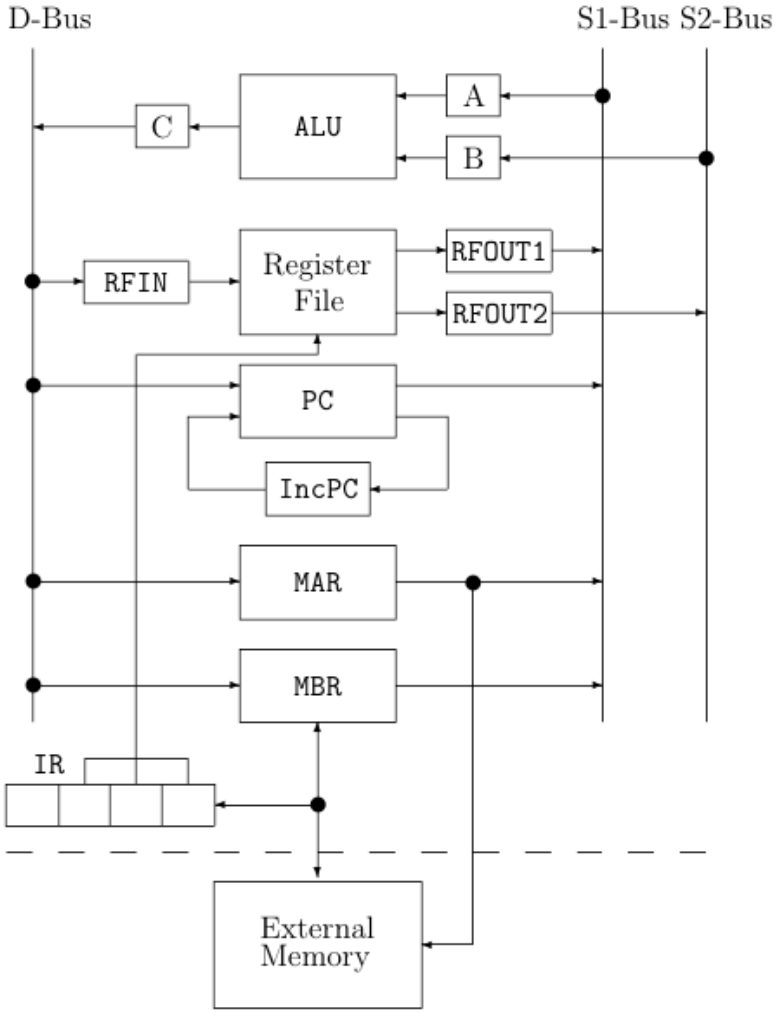

Figure 1: A simplified CPU

The processor has instructions which specifies 3 operands explicitly (namely, 2 source and 1 destination operands). The leftmost byte of the instruction represents the operation to be performed, such as `ADD`, `SUB` etc. For arithmetic and logic operations, the

operands must be in registers. Hence the 3 bytes will give the addresses of operands in the register file. There will be a direct path connecting these 3 bytes in the `IR` (*Instruction Register*) to the address of the register file, so that when you perform read/write on register file, the register specified in these bytes will be accessed.

If the instruction is `LOAD` or `STORE` to load a word from memory to register, and vice versa, the source operand (`LOAD`) or destination operand (`STORE`) refer to a memory address. How to find this address is specified by *Addressing Mode*. In this machine, for simiplicity, the memory operand byte (source/destination) will always be `1111 1111` (or in hex `0xff`), which means that the actual 32-bit memory address will be given in the word following the instruction (see example program below).

The ALU has the following operations: `ADD`, `SUB`, bitwise `AND`, `OR`, and `NOT`. For operations with only one operand (e.g. `NOT`), source operand 1 is used, and source operand 2 is empty.

Finally, there is a branch instruction, which performs conditional or unconditional branch as specified in the `cc` field of the instruction. The branch address is specified in the word following the instruction, the same as in `LOAD`/`STORE` instruction.

Instruction Format

Arithmetic/Logic Instruction

The instruction format of the machine (except `LOAD`, `STORE` and `BRANCH`):

Opcode	Source Operand 1	Source Operand 2	Destination Operand

The instructions can be categorized into the following types:

- **Arithmetic Operations**

```
ADD    R1, R2, R3 ; R3 <- R1 + R2
SUB    R1, R2, R3 ; R3 <- R1 - R2
```

- **Logical Operations**

```
AND    R1, R2, R3 ; R3 <- R1 and R2
OR     R1, R2, R3 ; R3 <- R1 or R2
NOT    R1, R3     ; R3 <- not R1
```

- **Data Movement Instruction**

```
MOV    R1, R3     ; R3 <- R1
```

Note that in the `NOT` and `MOV` operation, source operand 2 field is not used and will be set as `00000000`.

Load/Store Instruction

Moving data from Memory to registers and vice versa.

```
LD      A, R3       ; R3 <- A, A is in memory
ST      R1, A       ; A <- R1, A is in memory
```

Load instruction:

Opcode (Load)	00000000	Addressing Mode	Destination Operand

Store instruction:

Opcode (Store)	Source Operand	Addressing Mode	00000000

where the addressing mode (how to find the target address) is specified in byte 2 of the instruction. In this machine, only one addressing mode is used, where the target address is given by the word following the `LOAD` or `STORE` instruction (Absolute Addressing). This is specified as `11111111` in that byte.

Control Instruction

Control flow is by using `BRANCH` instruction. There are two types of branch instruction — *conditional* and *unconditional* Branch. Branch Instruction Format:

Opcode (Branch)	Condition Code (cc)	Addressing Mode	00000000

Conditional branch is based on the result of previous ALU operation, which is store in a flag register. In this machine, we only use a `ZERO` flag, which will be set to 1 if the ALU operation results in 0, and set to 0 otherwise. The target address is specified in the same way as in memory operation. Similarly, the byte of Addressing Mode is set to `11111111`.
The condition code is specified as

Condiation Code (cc)	Instruction	Description
00000000	BR	Unconditaion Branch, always goto
00000001	BZ	Branch if Zero flag is set
00000010	BNZ	Branch if Zero flag is NOT set

Halt Instruction

The `HLT` instruction is used to stop the program. The other 3 bytes are all 0.

Opcodes

Instruction	Opcode	Instruction	Opcode	Instruction	Opcode
ADD	00000000	OR	00000100	Bcc	00001000
SUB	00000001	MOV	00000101	HLT	00001001
NOT	00000010	LD	00000110		
AND	00000011	ST	00000111		

Part I: Example Program

The simulator program is given in `sim.py`. The code for the `SUB` and `ST` instruction is missing. Study the simulator code carefully, and complete the missing part.

Running the simulator program:

```
[python3] sim.py [-d] prog
```

If `-d` option is specified, the program will print out debug information.
The simulator obtains input program from the file `prog`.
Test you simulator with the following simple program:

```
        LD      P0,R4                   0000:  0600ff04  0000003c
        LD      P1,R1                   0008:  0600ff01  00000040
        MOV     R1,R2                   0010:  05010002
        LD      P2,R3                   0014:  0600ff03  00000044
L:      ADD     R4,R1,R4                001C:  00040104
        ADD     R1,R2,R1                0020:  00010201
        SUB     R3,R1,R5                0024:  01030105
        BNZ     L                       0028:  0802ff00  0000001c
        ST      R4,P                    0030:  0704ff00  00000048
        HLT                             0038:  09000000
P0:     .WORD   0                       003C:  00000000
P1:     .WORD   1                       0040:  00000001
P2:     .WORD   A                       0044:  0000000a
P:      .WORD                           0048:  00000000
```

What does this program do?

Part II: Hand Assemble

Translate the following program into hexadecimal form, and put it in a file named `prog2` with the same format as the file `prog`. Run the simulator by

```
[python3] sim.py [-d] prog2
```

Write down the final result stored in `P`. What does the program do?

```
        LD      P0, R4
        LD      P1, R1
        LD      P2, R2
        LD      P3, R3
L:      ADD     R4, R2, R4
        SUB     R3, R1, R3
        BNZ     L
        ST      R4, P
        HLT
P0:     .WORD 0
P1:     .WORD 1
P2:     .WORD 5
P3:     .WORD 4
P:      .WORD
```

A working simulator (executable pyc file only, without source code) is given to you, so that you can complete this part using this program, if your simulator in Part I is not working.

THE UNIVERSITY OF HONG KONG
Department of Computer Science
COMP2120 Computer Organization
Assignment 4

Due Date: Apr 30, 11pm, 2024.

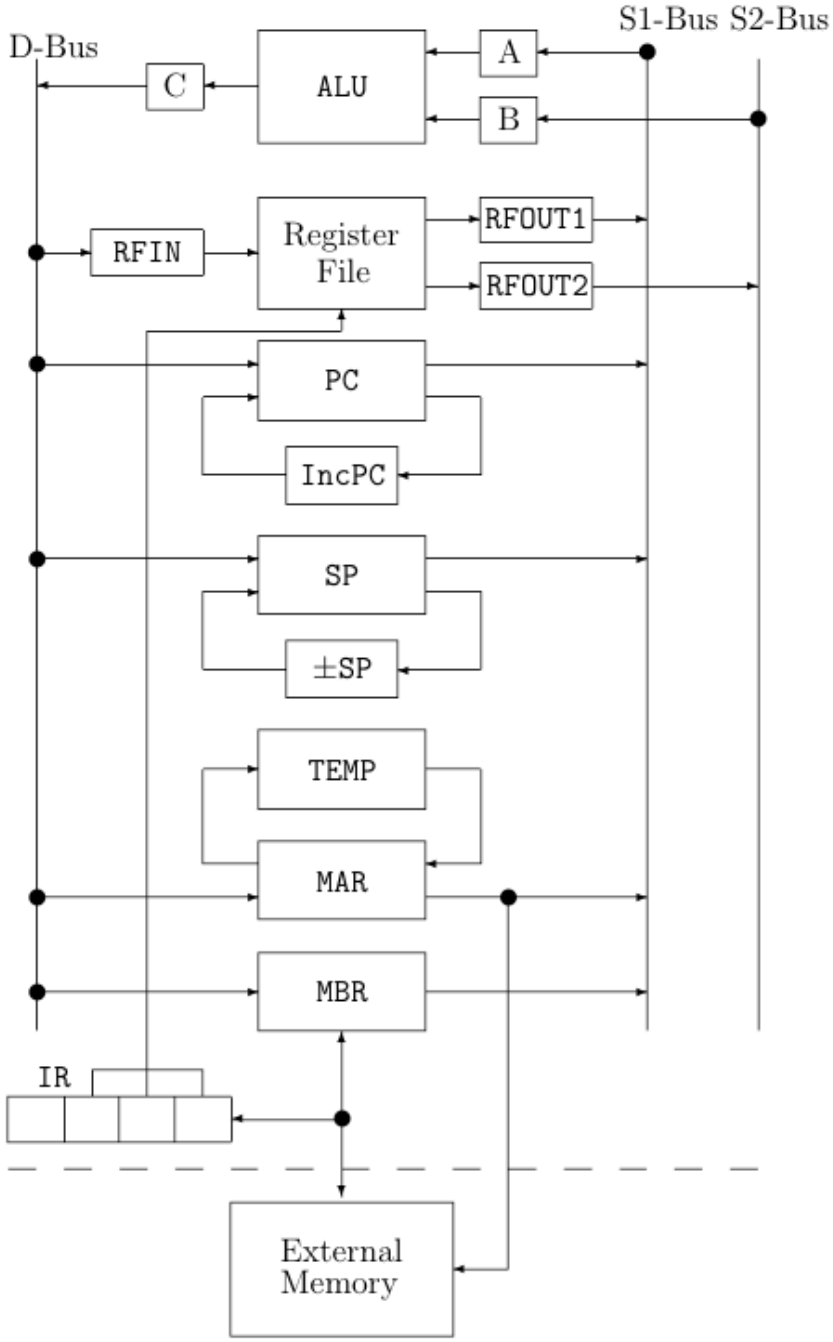

Figure 1: A simplified CPU

This assignment is based on the CPU and simulator in Assignment 2.

In this assignment, extra instructions are added. They are the PUSH, POP, CALL and RET instruction. In order to implement these instructions, the CPU is modified as follows:

1. A new register (SP, the stack pointer) is included. SP provides output to S1-bus, and receives input from D-bus. Also, the SP has special hardware to increase and decrease its value by 4 (similar to PC). This is provided by the special function do_incSP(), and do_decSP(), which is in turn controlled by the flag incSP and decSP.

2. A new register (TEMP) is included, which is directly connected to the MAR only, via a dedicated data path. Again you can move data between MAR and TEMP and special function do_MAR_to_TEMP() and do_TEMP_to_MAR() are provided, which are controlled by the MAR_to_TEMP and TEMP_to_MAR flag.

3. A new flag push_pop is included, which will move the SP to MAR. Otherwise, the CPU remains the same.

New instructions provided include:

PUSH Rn : SP ← SP-4; mem[SP] ← Rn

00001010	n	00000000	00000000

POP Rn : Rn ← mem[SP]; SP ← SP+4

00001011	00000000	00000000	n

CALL proc :

00001100	00000000	11111111	00000000

RET :

00001101	00000000	00000000	00000000

Summary Opcode:

Instruction	Opcode	Instruction	Opcode	Instruction	Opcode
ADD	00000000	MOV	00000101	PUSH	00001010
SUB	00000001	LD	00000110	POP	00001011
NOT	00000010	ST	00000111	CALL	00001100
AND	00000011	Bcc	00001000	RET	00001101
OR	00000100	HLT	00001001		

The program

The revised simulator program is given in `sim2.py`. Study the simulator code carefully.

1. Hand assemble the following assembly code and put it in a program file. Run the simulator on this program. Explain what the function `SQ` does?

```
     SUB    R4,R4,R4                0000H:  01040404
     LD     P1,R1                   0004H:  0600ff01  00000078
     MOV    R1,R2                   000CH:  05010002
     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                       0030H:  0802ff00  00000018
     ST     R4,P                    0038H:  0704ff00  00000080
     HLT                            0040H:  09000000

/* Procedure to calculate _____, input is R10, output is R11 */
/* The proc uses R12 and R13, need to save them on entry */
/* and restore them when exit*/

SQ:  PUSH   R12                     0044H:  ....
     PUSH   R13                     0048H:  ....
     LD     P1, R13                 004CH:  ....
     SUB    R11,R11,R11             0054H:
     MOV    R10,R12                 0058H:
L2:  ADD    R11,R10,R11             005CH:
     SUB    R12,R13,R12             0060H:
     BNZ    L2                      0064H:
     POP    R13                     006CH:
     POP    R12                     0070H:
     RET                            0074H:
P1:  .WORD  1                       0078H:  00000001
P2:  .WORD  A                       007CH:  0000000a
P:   .WORD                          0080H:  00000000
```

2. Run the simulator in debug mode. Write down the data transfer/transformation sequences involved in the execution of the instructions `CALL` and `RET`.

   You may skip intermediate step provided by the simulator, for example the instruction fetchs step should look like:

   ```
   MAR <- PC
   IR <- mem[MAR]
   ```

   or in English, move the value of `PC` to `MAR`. Then read memory and the result (`mem[MAR]`) is moved to `IR`, i.e. just write down the source and destination of the data movement, without the paths etc.

3. Modify the program so that it will calculate the value of $1 - 2 + 3 - 4 \cdots - 8 + 9$. That is,

   ```
   sum = 0;
   for i=1 to 9 do sum += sq(i)
   ```

   Where `sq(i)` return `i` when `i` is odd, otherwise return `-i`. Note that the original program is already a loop from 1 to 9. Just replace the function `SQ` by

   ```
   if (R10 is odd) R11 = R10;
   else R11 = 0 - R10;
   ```

   Since we don't have a `NEG` instruction, to find $-x$, we use $0 - x$.

   To check if a number x is odd, just check if the rightmost bit is 1. We can find x `AND` $00000000 \cdots 0001$. (i.e. 1) After `AND` operation, all bits `AND`ed with 0 will be 0. If the rightmost bit is 0, then the result is 0. Otherwise the result is non-zero.

   Note that the address of `P1`, `P2` and `P` may got changed when the length of the function `SQ` is changed. You may need to change the address of them in the program, e.g. in line 2

   ```
   LD P1,R1
   ```

   you may need to find the new address of `P1`, and also in line $4 \cdots$.