COMP2120 Computer Organisation
24/25 Semester 2
Assignment 2

1. 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).

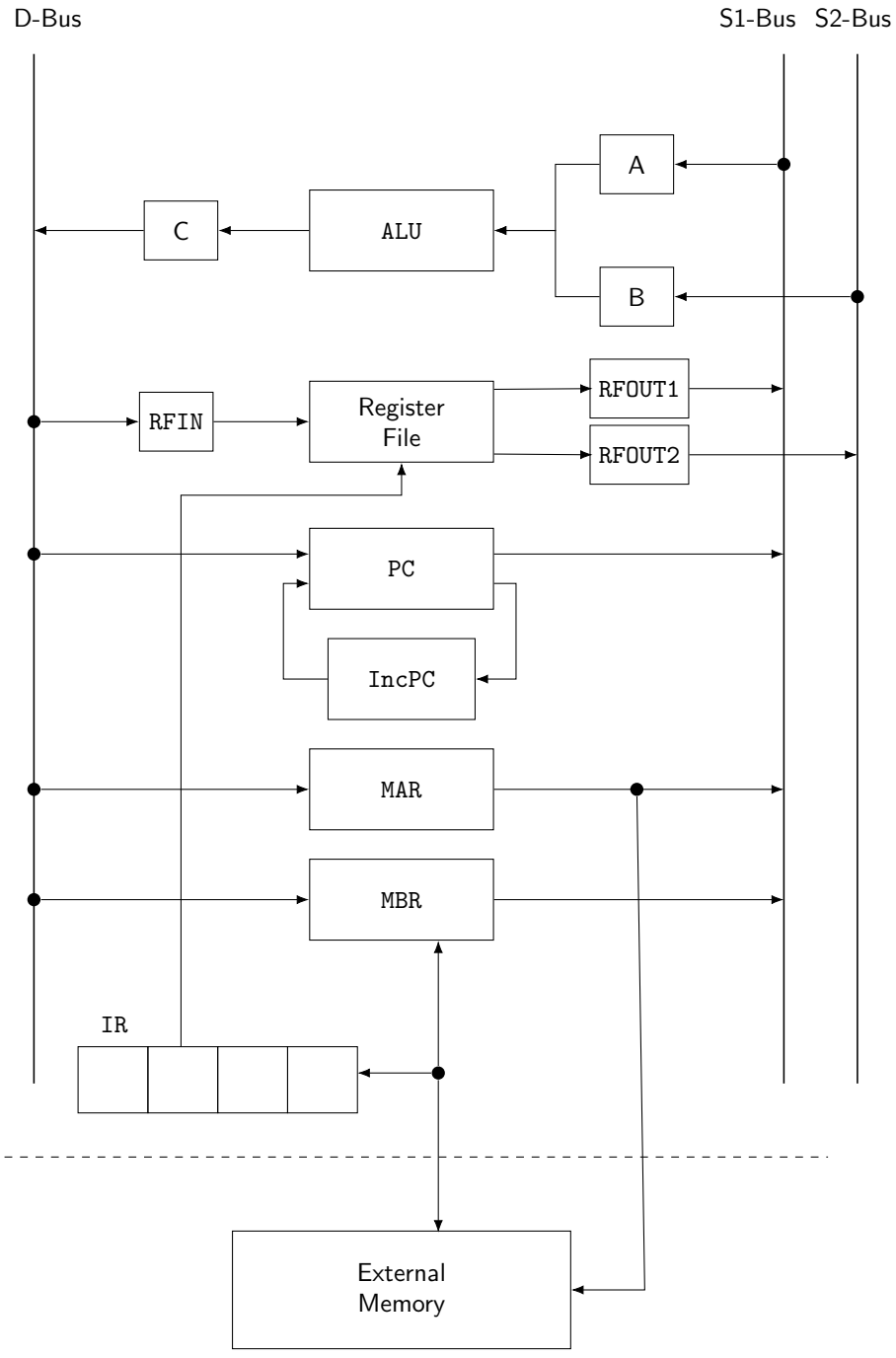

Figure 1: A simplified CPU

The processor has instructions which specifies 3 operands explicitly (namely, 2 source and 1 destination operands). The leftmost

byte of the instruction represents the oper- ation 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/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
```

- **Logic 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 operations, source operand 2 field is not used and will be set as 0000 0000.

Load/Store Instruction

Moving data from Memory to registers and vice versa.

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

Condition Code (cc)	Instruction	Description
`00000000`	`BR`	Unconditional 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.

> ***Author's Note:*** *The code files are also provided on the website. Click on the "Download Attachments" button to download.*

Solution: *(The missing parts in the example program:)*

Note that the comments are included for explanation only, and are not required in your submission.

```
220 def set_SUB():
221     # Fill in the code for SUB instruction here
222     Signal["calc_addr"] = 0          # Disable address calculation (not needed)
223     Signal["branch"] = 0             # This is not a branch instruction
224     Signal["read_RF_port_1"] = 1     # Enable read from RF1 (source operand 1)
225     Signal["read_RF_port_2"] = 1     # Enable read from RF2 (source operand 2)
226     Signal["write_RF"] = 1           # Enable writing to register file (destination operand)
227     Signal["src_of_S1"] = "RFOUT1"   # Connect S1-bus source to register file output port 1
228     Signal["dst_of_S1"] = "A"        # Route S1-bus destination to ALU input A
229     Signal["src_of_S2"] = "RFOUT2"   # Connect S2-bus source to register file output port 2
230     Signal["dst_of_S2"] = "B"        # Route S2-bus destination to ALU input B
231     Signal["src_of_D"] = "C"         # Connect D-bus source to ALU output C
232     Signal["dst_of_D"] = "RFIN"      # Route D-bus destination to register file input
233     Signal["doalu"] = 1              # Enable ALU operation
234     Signal["ALU_func"] = "OP_SUB"    # Set ALU function to subtraction operation
```

```
    Signal["move_via_S1"] = 1        # Enable data movement through S1-bus
    Signal["move_via_S2"] = 1        # Enable data movement through S2-bus
    Signal["move_via_D"] = 1         # Enable data movement through D-bus
    Signal["read_memory"] = 0        # Disable memory read (not a load instruction)
    Signal["write_memory"] = 0       # Disable memory write (not a store instruction)
    Signal["dohalt"] = 0             # Do not halt the processor
```

```
def set_ST():
    # Fill in the code for ST instruction here
    Signal["calc_addr"] = 1          # Enable address calculation (needed for 0xFF mode)
    Signal["branch"] = 0             # This is not a branch instruction
    Signal["read_RF_port_1"] = 1     # Enable read from RF1 (source operand - register)
    Signal["read_RF_port_2"] = 0     # Disable read from RF2 (not needed for store)
    Signal["write_RF"] = 0           # Disable writing to register file (storing to memory)
    Signal["src_of_S1"] = "RFOUT1"   # Connect S1-bus source to register file output port 1
    Signal["dst_of_S1"] = "A"        # Route S1-bus destination to ALU input A
    Signal["src_of_S2"] = ""         # S2-bus source not needed (no second operand)
    Signal["dst_of_S2"] = ""         # S2-bus destination not needed (no second operand)
    Signal["src_of_D"] = "C"         # Connect D-bus source to ALU output C
    Signal["dst_of_D"] = "MBR"       # Route D-bus destination to Memory Buffer Register
    Signal["doalu"] = 1              # Enable ALU operation
    Signal["ALU_func"] = "OP_COPY"   # Set ALU function to copy operation (pass data through)
    Signal["move_via_S1"] = 1        # Enable data movement through S1-bus
    Signal["move_via_S2"] = 0        # Disable data movement through S2-bus (not used)
    Signal["move_via_D"] = 1         # Enable data movement through D-bus
    Signal["read_memory"] = 0        # Disable memory read (this is a store, not load)
    Signal["write_memory"] = 1       # Enable memory write (store data to memory)
    Signal["dohalt"] = 0             # Do not halt the processor
```

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:   01010305
        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?

Solution: The program computes the sum of integers from 1 to 9.

Explanation: After the first four lines, the registers are initialized as follows:

- `R4` = 0 (loaded from memory address `P0 - 0x3C`)

- R1 = 1 (loaded from memory address P1 - 0x40)
- R2 = 1 (copied from R1, this is a constant as no operations use it as destination)
- R3 = 10 (loaded from memory address P2 - 0x44, also a constant)

In every iteration of the loop labeled L:, the following operations occur:

- R4 is incremented by the value in R1
- R1 is incremented by the constant 1 (from R2)
- R5 is set to be R3 (10) - R1
- If the previous operation results in non-zero, i.e., R1 $\neq$ 10, branch back to L:

When R1 reaches 10, R4 will have accumulated the sum of integers from 1 to 9. Finally, the value in R4 is stored back to memory address P - 0x48, and the program halts.

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.

Solution: The program is hand assembled as follows:

```
0600ff04
0000003c
0600ff01
00000040
0600ff02
00000044
0600ff03
00000048
00040204
```

```
01030103
0802ff00
00000020
0704ff00
0000004C
09000000
00000000
00000001
00000005
00000004
00000000
```

The purpose of this program is to compute the sum of $5+5+5+5=20$. The final result stored in memory address `P` is `0x00000014`.