Dear All,

These remarks act as an extension to the project description, explaining some points in further detail. **Make sure you read it carefully as it will be considered in the evaluation.**

Part 1: Quick Remarks

1) The PC starts from 0.

The first instruction will be stored at address 0 in all packages.

2) In Packages 1 and 3, R0 (Zero Register) can be the destination in an instruction.

The instruction must continue normally in the pipeline.

However, it won't overwrite the value of R0. The value will remain 0 in all cases.

You must **not** throw an error if R0 is the destination register.vs

Let the instruction continue normally in the pipeline, but don't overwrite the value of R0.

3) For each clock cycle, you need to print which instruction is in each stage, as well as, the values that entered the stage, and the output of this stage.

Moreover, if you changed the value of a location in the memory or the register file, you need to print that this location or register (including R0) value has changed alongside the new value (and in which stage did the value change).

At the end of your program, you need to print the values of all registers (general and special purpose including the PC and SREG), and the full instruction and data memory locations.

4) Immediate values are signed (2's complement), which means they can be positive or negative.

The only exception is the "shift" instructions, where the immediate will always be positive.

5) You are required to make sure that the data types used match the project description.

If an instruction is 32 bits, you need to use "Integer" or if you use a String of 0s and 1s, then you must ensure that the string contains 32 characters.

If an instruction is 16 bits, you need to use "short int", or a string of 16 characters.

If a value is 8 bits, you need to use "char" or “int_8” from the <stdint> library.

6) For Packages 2 and 4, check for an example on the Carry flag calculation at the end of this document..

7) When parsing the text file containing the assembly instructions, you are not allowed to keep track of the fields.

You should create a decimal/binary/String of 0s and 1s concatenated instructions to be stored in the instruction memory.

Then, during the decode stage, you will decode the instruction into ALL POSSIBLE FORMATS.

Any help inserted from the parsing stage will not be correct.

8) In the conditional branch instructions in all packages, "PC + 1" is the PC that was already incremented during the fetch stage.

You are not required to increment it again. It just indicates that the PC of the branch instruction will be incremented by 1, which was done during the fetch stage already.

Check Part 5 in this post to understand more.

9) You should NOT pre-calculate the total cycles that an assembly program will take and use it as a stopping condition for the simulation. Only use it as a point of reference when you are debugging. The correct way of stopping would be that there are no more instructions to fetch.

Part 2: Branches and Jumps 1.0

Clarification regarding the conditional branches/jumps for **ALL PACKAGES**:

```
PC = PC + 1 + IMM = Address of branch instruction + 1 (to point to the next
instruction) + immediate
```

Example (using the instructions from Package 1):

```
0: BNE R1 R2 2

1: ADD R1 R2 R3

2: ADD R4 R5 R6

3: ADD R7 R8 R9

if(R1 != R2) {
```

```
PC = address of branch instruction + 1 "to point to next instruction" + 2
"offset"

}
```

This applies to:

Package 1: Branch if Not Equal (BNE)

Package 2: Branch if Equal Zero (BEQZ)

Package 3: Jump if Equal (JEQ)

Package 4: Branch if Equal Zero (BEQZ)

Also, the unconditional jump instructions (J and JMP) in Packages 1 and 3 will have the following format:

```
0: J 2 // JMP 2 // Address = PC[31:28] + 2 = 2 since the most significant 4
bits of the current PC are 0s.

1: ADD R1 R2 R3

2: ADD R4 R5 R6

3: ADD R7 R8 R9
```

While the Jump Register (JR) and Branch Register (BR) instructions in Packages 2 and 4 will have the following format:

```
0: JR R0 R1 // BR R0 R1

1: ADD R1 R2 R3

2: ADD R4 R5 R6

3: ADD R7 R8 R9

// Address = Concatenation between the value of R0 and R1 where R0 is the most
significant byte and R1 is the least significant byte

// Address = 00000000 00000000 (since R0 and R1 initially contain the value 0)
```

Part 3: Braches and Jumps 2.0

Normally, the PC is updated in the **Fetch** stage to point to the next instruction.

```
PC = PC + 1
```

However, during the Execute stage of a conditional branch/jump instruction, the PC can be updated with the branch address if the condition is true.

Regarding the unconditional jump instructions, you will also update the PC with the jump address during the **Execute** stage in all packages.

In all cases, if the condition is true, or we are unconditionally jumping, all instructions that entered the datapath after the branch/jump instruction must be ignored (not executed and dropped) and we should start fetching the instruction we branched/jumped to.

In Packages 2 and 4, you will fetch the instruction (that we branched/jumped to) directly in the following clock cycle after the Execute stage of the branch/jump instruction.

In Packages 1 and 3, you must wait for the branch/jump instruction to finish the 2 clock cycles it will spend in the Execute stage, then the branch/jump instruction will move to the Memory stage in the following cycle while nothing will be done in the Fetch stage.

Thus, we will fetch the instruction (that we branched/jumped to) after 2 cycles after finishing the Execute stage of branch/jump instruction.

```
X: Branch/Jump in Execute (1st time) + an instruction in the decode stage

X+1: Branch/Jump in Execute (2nd time) + an instruction in the decode stage +
an instruction in the fetch stage

X+2: Branch/Jump in Memory (nothing will be fetched since we are using the
memory in the memory stage) + an instruction in the execute stage + an
instruction in the decode stage

X+3 Fetch new instruction (branched/jumped) + Branch/Jump in Write Back + drop
the other 2 instructions in the datapath
```

Any instruction fetched while we are calculating the branch address and condition or the jump address must be dropped from the datapath when we branch or jump.

Part 4: Hazards

You are not required to handle Data Hazards.

Regarding Control Hazards, you only need to do the following:

As I mentioned before, the PC is updated with the branch/jump address during the execute stage for all packages.

You need to make sure that during the execute stage if you are updating the PC with the branch or jump address, you need to remove the instructions in the decode and fetch stages (flush the instructions that entered the pipeline after the branch/jump instruction), and in the upcoming clock cycle, you will fetch the instruction pointed to by the new PC.

However, the branch/jump instruction which was at the execute stage will move to the memory stage (Packages 1 and 3).

Imagine if we have a branch instruction followed by "add" and "sub" instructions.

If the branch condition is "taken", which means that we will branch, we will update the PC in the execute stage with the address of the new instruction that we should jump to.

However, the "add" and "sub" instructions were already in the pipeline, so we need to remove them.

Example:

(It does not represent any of the 4 packages)

```
Cycles 1 2 3 4 5 6 7 8


Branch F D E M W

ADD    F D

SUB          F

NEW          F D E M W


Note: NEW refers to the new instruction that we branched/jumped to.

Also, in Packages 2 and 4, the execute stage is the last stage since we don't
have memory and writeback stages.

However, in Packages 1 and 3, the decode and execute stages take 2 clock cycles
each, and we only fetch a new instruction during an odd clock cycle, so you'll
```

update the PC in the second execute cycle, and then wait for 2 cycles to fetch the new instruction.

The above example applies to all of the unconditional jump instructions (J, JMP, BR, JR).

Also, it applies to all of the conditional jump instructions (BEQZ, BNE, JEQ) if the condition is true.

If the condition is false, we will continue executing the "add" and "sub" instructions normally.

How to remove/flush an instruction from the pipeline?

- You can remove it by clearing all the variables associated with it, or by using flags.

- You must make sure that the removed instructions won't affect the registers' values and the data memory values. Other than that, use any technique to remove it from the pipeline.

Also, it's normal to have empty stages in the middle not operating on any instruction after removing the instructions.

In Packages 1 and 3, the execute stage takes 2 clock cycles.
It is better (and I personally suggest it) to update the PC with the branch address (if the branch is taken) or the jump address after the second execute cycle and not after the first one.
Meaning, let the branch/jump instruction finish its 2 clock cycles first in the execute stage, then if you are going to branch or jump, remove/flush the instructions in the decode and fetch stage from the previous cycle, and fetch the new instruction pointed by the updated PC.

Part 5: Conditional Branches PC

There is 1 approach for handling the PC value when dealing with conditional branches/jumps:

It is more accurate if you used the PC of the conditional branch instruction (storing it with the data of the instruction for later use).

Example:

Packages 2 and 4:

```
Cycles      1 2 3 4 5 6 7 8

0: BEQZ R1 5   F D E --> We stored the PC of BEQZ, thus: PC + 1 + IMM = 0 + 1 + 5 = 6 if (R1 == 0)
```

```
1: ADD R2 R3       F D --> Removed

2: ADD R4 R5       F --> Removed

3: SUB R6 R7

4: SUB R8 R9

5: ADD R10 R11

6: ADD R12 R13     F D E

7: SUB R14 R15          F D E

8: SUB R16 R17            F D E
```

Total instructions that entered the pipeline: 6

Total cycles: 3 + (5 * 1) = 8 clock cycles

Packages 1 and 3:

```
Cycles             1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21

0: BNE/JEQ R1 R2 3  F D D E E M W --> We stored the PC of BNE/JEQ, thus PC + 1
+ IMM = 0 + 1 + 3 = 4 if condition is true

1: ADD R2 R3 R4          F D D --> Removed

2: ADD R5 R6 R7              F --> Removed

3: SUB R8 R9 R10

4: SUB R11 R12 R13             F D D E  E  M  W

5: ADD R14 R15 R16                   F D  D  E  E  M  W

6: ADD R17 R18 R19                      F  D  D  E  E  M  W

7: SUB R20 R21 R22                      F  D  D  E  E  M  W

8: SUB R23 R24 R25                            F  D  D  E  E  M  W
```

Total instructions that entered the pipeline = 8

Total cycles = 7 + (7 * 2) = 21 clock cycle

Overflow vs Carry

Overflow: Occurs when the sign of the result can't be represented (needs 1 more bit).

Example: The result is 32 bits (max size of our data), but we need 1 more bit to represent the sign (33 bits).

Note 1: There is a difference between the carry bit (useful in unsigned operations), and the overflow bit (useful in signed operations).

Note 2: An overflow can occur, even if the carry is 0. They are different!

Overflow Examples:

Adding 2 positive numbers and the result is negative.

Adding 2 negative numbers and the result is positive.

Quick tip to determine whether we have an overflow or not:

XOR the last 2 carries (from bit 6 to bit 7, and from bit 7)

XOR == 1 --> Overflow

XOR == 0 --> No Overflow

XOR Table:

0 XOR 0 = 0

0 XOR 1 = 1

1 XOR 0 = 1

1 XOR 1 = 0

Assume our values are represented in 8 bits only:

Example 1: Carry with Overflow

-128 in 8 bits = 1000 0000

-128 + -128 = -256 (result is negative)

```
   7654 3210   --> Bit Numbers
 1 0000 000    --> Carrys
   1000 0000   --> -128
 +             --> +
   1000 0000   --> -128
 =             --> =
   0000 0000  --> 0 INCORRECT (sign bit not represented in 8 bits, need 1 more bit)
```

In the above example, we have a carry of 1, and the result is 00000000 = 0.
So, the result sign is positive while we were expecting a negative value.
Thus, we have an overflow.
Also, using our XOR method, the last 2 carries are 0 (from bit 6 to bit 7) and 1 (from bit 7)
Therefore, 1 X0R 0 = 1 --> Overflow!

Example 2: Carry with No Overflow

64 in 8 bits = 0100 0000
-64 in 8 bits = 1100 0000

64 + -64 = 0

```
   7654 3210  --> Bit Numbers
 1  1000 000   --> Carrys
   0100 0000 --> 64
 +             --> +
   1100 0000  --> -64
 =             --> =
   0000 0000 --> 0 CORRECT (although we have a carry bit, we don't have an overflow)
```

The result is 0 which contains the expected sign.
Using the XOR trick, the last 2 carries are 1 (from bit 6 to 7) and 1 (from bit 7).
1 XOR 1 = 0 (no overflow, although we have a carry bit!)

Example 3: Overflow with No Carry

64 + 64 = 128

```
   7654 3210 --> Bit Numbers
 0 1000 000   --> Carrys
   0100 0000 --> 64
 +             --> +
   0100 0000 --> 64
 =             --> =
```

1000 0000 --> -128 INCORRECT (although we don't have a carry bit, but we have an overflow)

We added 2 positive numbers and the result is negative.
We don't have a carry, however using the XOR trick, we have an overflow
Carry from bit 6 to 7 is 1, while Carry from bit 7 is 0.
0 XOR 1 = 1 --> Overflow!

Also, using our eyes, we can see the result having a negative sign although we are adding 2 positive numbers.

Carry Flag Check

Dear All,

Since there are some special cases where UNSIGNED(R1) OP UNSIGNED(R2) > Byte.MAX_VALUE does not handle.

The best solution to handle all cases is instead of checking whether the result is > Byte.MAX_VALUE or not, we can check on bit 8 (9th bit) of the result whether it is 1 or 0.

We will still get the unsigned value for both operands, and check on the 9th bit (bit 8) of the result.

9th bit (bit 8) = 1 --> Carry = 1

9th bit (bit 8) = 0 --> Carry = 0

Example:

R5 = -5 = 0b11111011 in 8 bits
R6 = 5 = 0b00000101 in 8 bits

int temp1 = R5 & 0x000000FF = 0b00000000000000000000000011111011
int temp2 = R6 & 0x000000FF = 0b00000000000000000000000000000101

```
if( ((temp1 OP temp2) & MASK) == MASK) {
        Carry = 1;
} else {
        Carry = 0;
}
```