# ECE 610 project3 report

## Github

# Input Sample:

```
addi x4,x0,4                    1  // x4 = 4
addi x7,x0,7                    2  // x7 = 7
add x11,x4,x7                   3  // x11 = 4 + 7 = 11
sw x11,44(x0)                   4  // mem[44:47] = 11
lw x17,44(x0)                   5  // x17 = mem[44:47] = 11 this is not hazard.
mul x18,x11,x17                 6  // x18 = 11 * 11 = 121
sll x8,x7,x4                    7  // x8 = 7<<4 = 112
slti x6,x8,180                  8  // x6 = 1 (112<180)
beq x0,x0,L1                    9  // jump 0=0 change this line to access other instruction
and x3,x11,x7                   10 // x3 = 11 and 7 = 3
lui x30,74565                   11 // x30 = 305,397,760 (0x12345000)
addi x30,x30,1656               12 // x30 = 305,419,896 (0x12345678)
andi x2,x11,6                   13 // x2 = 11 and 6 = 2
or x15,x11,x7                   14 // x15 = 11 or 7 = 15
ori x31,x15,16                  15 // x31 = 15 or 16 = 31
srl x31,x31,x2                  16 // x31 = 31>>2 = 7
sub x1,x4,x3                    17 // x1 = 4 - 3 = 1
L1:add x5,x2,x3                 18 // L1 x5 = 0+0 = 0 not 3 or 5
```

To check the instructions between beq and L1, please change one of the x0 into x4

# Mode:

**Instruction mode:**

**This is a mode that like single processer, it will run instruction by instruction. It's not pipelining so that there is no hazard.**

**Cycle mode:**

**This is a mode that has pipelining method, with a hazard controller which just stall 3 cycle until the hazard instruction write back.**

# IF

**Fetching the instruction: <add x0,x0,x0>**

1. **Key-in mode (4) -> (6)**

   A mode that we can input our instruction via terminal.

```
--------------------------------------------------
How many instructions do you want to enter?
number:3

Instruction1 :addi x5,x0,8

Instruction2 :add x7,x5,x0

Instruction3 :mul x7,x5,x7
```

2. **File mode (8)**

   A mode that will read sample.txt.

```
What kind of mode you want to select
1 Instruction mode
2 Cycle mode
3 exit
4 key instruction mode
5 print instruction
6 decode
7 lucky seven
8 read instruction.txt
8
Instructions:
addi x4,x0,4
addi x7,x0,7
add x11,x4,x7
sw x11,44(x0)
lw x17,44(x0)
mul x18,x11,x17
sll x8,x7,x4
slti x6,x8,180
beq x0,x0,L1
and x3,x11,x7
lui x30,74565
addi x30,x30,1656
andi x2,x11,6
or x15,x11,x7
ori x31,x15,16
srl x31,x31,x2
sub x1,x4,x3
L1:add x5,x2,x3
```

# ID

**Decode the instruction: <add x0,x0,x0 -> 00000000000000000000000000110011>**

**There are 6 types: R , I, lw, sw, beq, lui**

```
switch (index)
{
case 0:
    // for R instruction
    tempV = decode_r(split_result);
    break;
case 1: // for I instruction
    tempV = decode_i(split_result);
    break;
case 2: // for lw
    tempV = decode_lw(split_result);
    break;
case 3: // for sw
    tempV = decode_sw(split_result);
    break;
case 4: // for beq
    tempV = decode_beq(split_result);
    break;
case 5: // for lui
    tempV = decode_lui(split_result);
    break;
}
```

After decoding, we store the instructions' information(e.g. rd rs1 rs2 imm…) into class "Instruction".

```
class Instruction
{
public:
    Instruction *nextInstruction = nullptr;
    string original_ins = "";      // add x0,x0,x0
    string type = "";              // R
    string instruction_name = ""; // add
    string opcode = "";            // 0010100
    string func7 = "";
    string func3 = "";
    string rd = "";
    string rs1 = "";
    string rs2 = "";
    string imm1 = "";
    string imm2 = "";
    string binary_ins = "";
    string jump = "";                // name of the branch destination e.g L1, jump , goto L
    string imm = "";                 // the immediate
    int ALUOutput = 0;               // the ALUOutput
    long long int mul_ALUOutput = 0; // this is the ALUOutput for multiplication
    int LMD = 0;                     // this is the data that load from mem to the register
    string kind = "";                // this shows the instruction is e.g. add = 1 , sub = 2.
    int state = 1;                   // this is the state
};
```

We implemented there function in decode.h and decode.cpp

**Decode example:**

**-R type:**

```
[2]
> nextInstruction: 0x170e6bd85b8
> original_ins: "add x11,x4,x7"
> type: "R"
> instruction_name: "add"
> opcode: "0110011"
> func7: "0000000"
> func3: "000"
> rd: "01011"
> rs1: "00100"
> rs2: "00111"
> imm1: ""
> imm2: ""
> binary_ins: "00000000011100100000010110110011"
```

# EX

**The ALU execution happens in here. <add rd,rs1,rs2 -> rd = rs1 + rs2 >**

**There are 16 functions:**

add, sub, addi, mul, lw, sw, beq, lui, and, andi, or, ori, sll, srl, slti, sltiu.

We implement all these instructions in execution.h

Example:

```
int Add_ex(Instruction adding)
{
    // int rd = std::bitset<32>(adding.rd).to_ulong();
    int rs1 = std::bitset<32>(adding.rs1).to_ulong();
    int rs2 = std::bitset<32>(adding.rs2).to_ulong();
    // registers[rd] = registers[rs1] + registers[rs2]; write back is not this part
    int ALUOutput = registers[rs1] + registers[rs2];
    return ALUOutput;
}
```

# MEM

**There are 3 functions:**

lw, sw, beq.

```
int Lw_mem(Instruction loading)
{
    int rs1 = std::bitset<32>(loading.rs1).to_ulong();
    int LMD;
    for (int i = 0; i < 4; i++)
    {
        LMD |= memory[loading.ALUOutput + i] << (8 * i); // little endian 0A0B0C0D => memory[0] = 0D
    }
    return LMD;
}

void Sw_mem(Instruction storing)
{
    int rs1 = std::bitset<32>(storing.rs1).to_ulong();
    memory[storing.ALUOutput + 3] = (registers[rs1] >> 24) & 0xFF;
    memory[storing.ALUOutput + 2] = (registers[rs1] >> 16) & 0xFF;
    memory[storing.ALUOutput + 1] = (registers[rs1] >> 8) & 0xFF;
    memory[storing.ALUOutput] = registers[rs1] & 0xFF;
}

int Beq_mem(Instruction branching)
{
    if (branching.ALUOutput == 1)
    {
        int LMD = std::bitset<32>(branching.imm).to_ulong();
        return LMD;
    }
    else
        return 0;
}
```

Lw_mem will return the LMD, which is the value that load from memory[n:n+3]

Sw_mem will store the value into memory[n:n+3]

Beq_mem will return the value that where the instruction is.

We implement all these instructions in Memory.h

# WB

This state we will write back the value into register.

For example:

```
void Add_wb(Instruction adding)
{
    int rd = std::bitset<32>(adding.rd).to_ulong();
    registers[rd] = adding.ALUOutput;
}
```

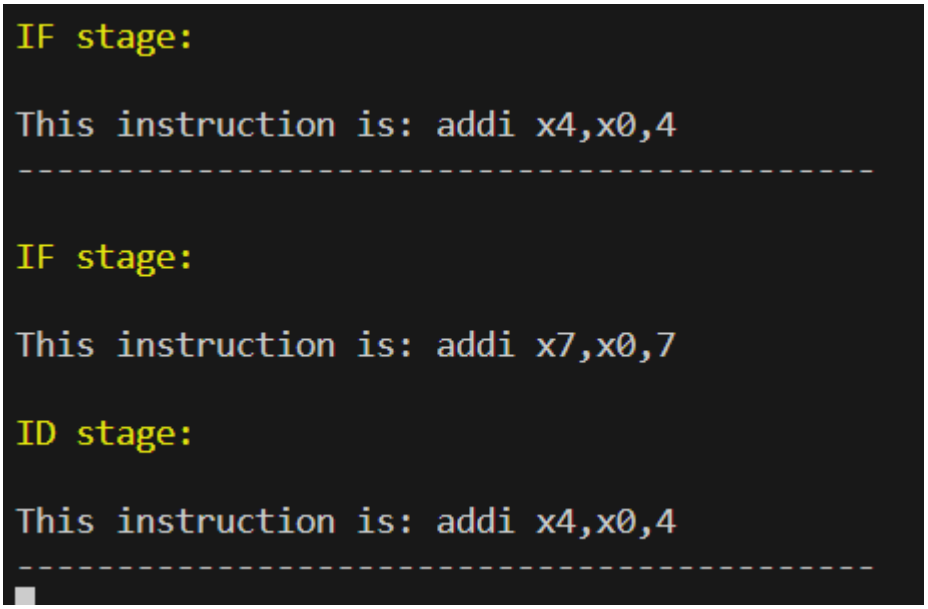
# Pipelining

The work will be Three part:

1. **Pushing new instruction into pipeline**

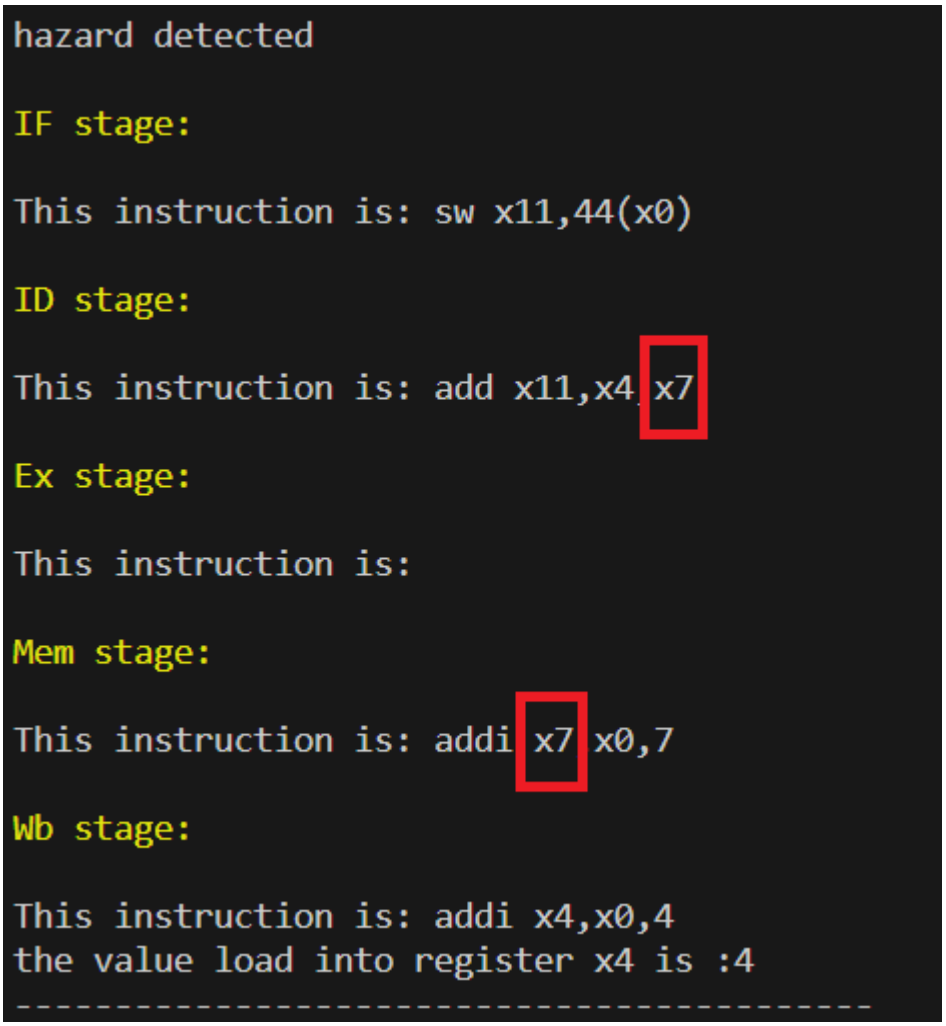
   If there is no hazard, we just keep fetching new instruction.

   But if we have a hazard, we will insert a bubble into Ex state and until there is no hazard.



2. **Hazard detection.**

We check from the Wb state to ID state, we don't need to stall if the hazard's location is at IF state because we still can fetch next instruction and not effect to the system.



3. **Branch**

We will stall the pipelining and flush all the instructions in pipelining if we jump.

```
IF stage:

This instruction is: lui x30,74565

ID stage:

This instruction is: and x3,x11,x7

Ex stage:

This instruction is:

Mem stage:

This instruction is:

Wb stage:

This instruction is: beq x0,x0,L1
Jump to the 18th instruction
-----------------------------------------
```

```
Wb stage:

This instruction is: beq x0,x0,L1
Jump to the 18th instruction
-----------------------------------------

IF stage:

This instruction is: L1:add x5,x2,x3

ID stage:

This instruction is:

Ex stage:

This instruction is:

Mem stage:

This instruction is:

Wb stage:

This instruction is:
-----------------------------------------
```