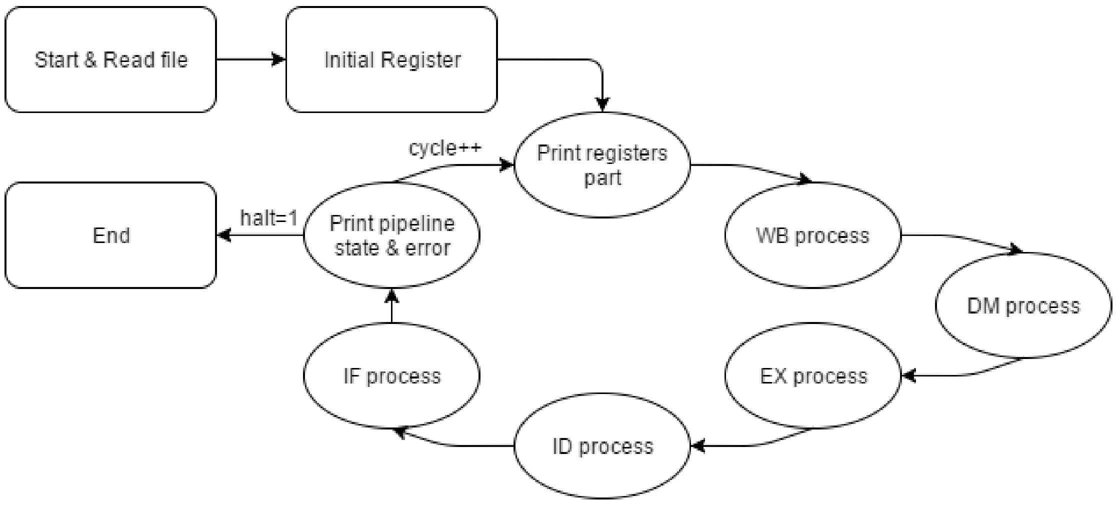
Computer Architecture Project 2 Report

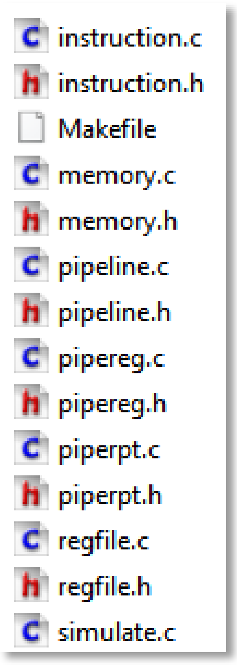
104062261 葉毓浩

1) Project Description

1-1) Program Flow chart:



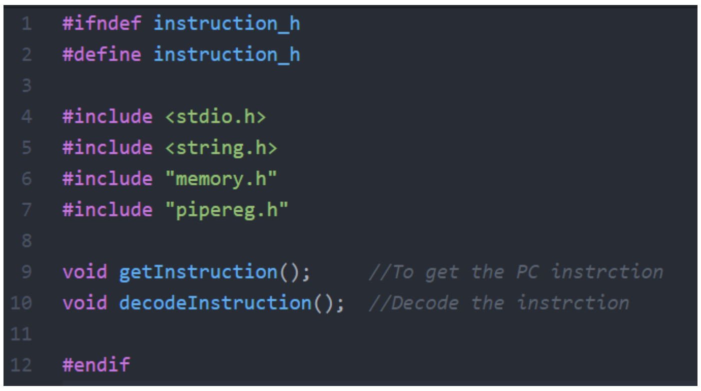
1-2) Detailed Description:


instruction.c
instruction.h
Makefile
memory.c
memory.h
pipeline.c
pipeline.h
pipereg.c
pipereg.h
piperpt.c
piperpt.h
regfile.c
regfile.h
simulate.c

The file I created in this project is shown in the left picture.

Instruction: Get instruction from memory and decode it
Memory: Read memory from binary file and save here
Pipeline: Handle each pipeline process
Pipereg : The pipeline data structure
Piperpt: Write snapshot.rpt and error_dump.rpt
Regfile: Save the registers in here
Simulate: The main simulation process

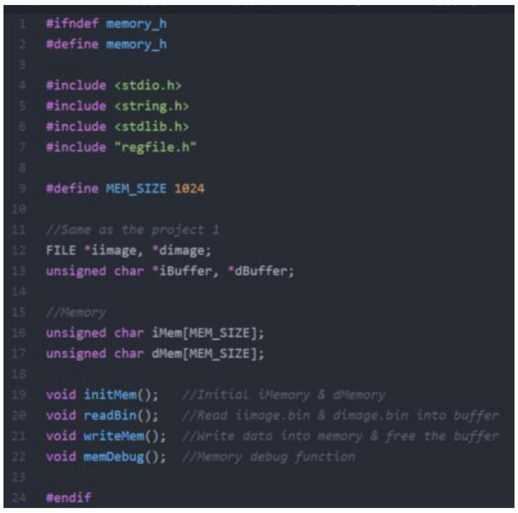
I. Instruction part:



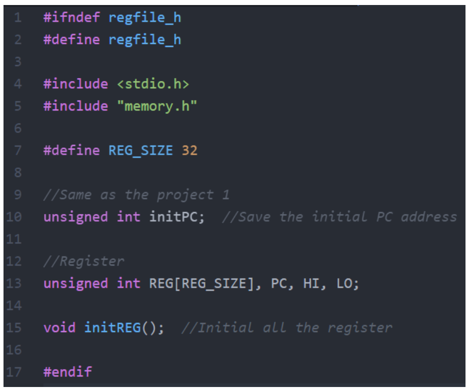
```c
#ifndef instruction_h
#define instruction_h

#include <stdio.h>
#include <string.h>
#include "memory.h"
#include "pipereg.h"

void getInstruction();     //To get the PC instrction
void decodeInstruction();  //Decode the instrction

#endif
```

In this part, I have written two function about the instruction in here. The "getInstruction" function will be call in the IF process and save the instruction data in the IF/ID pipeline register. And the "decodeInstruction" function is called by the "getInstruction" to decode the instruction.

II. Memory & Regfile part:



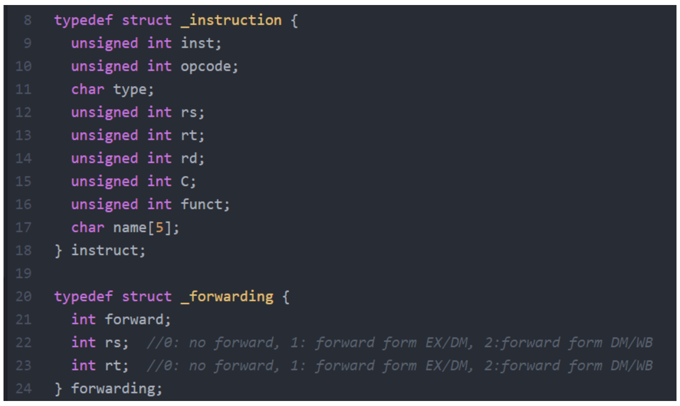
```c
#ifndef memory_h
#define memory_h

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "regfile.h"

#define MEM_SIZE 1024

//Same as the project 1
FILE *iimage, *dimage;
unsigned char *iBuffer, *dBuffer;

//Memory
unsigned char iMem[MEM_SIZE];
unsigned char dMem[MEM_SIZE];

void initMem();    //Initial iMemory & dMemory
void readBin();    //Read iimage.bin & dimage.bin into buffer
void writeMem();   //Write data into memory & free the buffer
void memDebug();   //Memory debug function

#endif
```



```c
#ifndef regfile_h
#define regfile_h

#include <stdio.h>
#include "memory.h"

#define REG_SIZE 32

//Same as the project 1
unsigned int initPC;  //Save the initial PC address

//Register
unsigned int REG[REG_SIZE], PC, HI, LO;

void initREG();  //Initial all the register

#endif
```

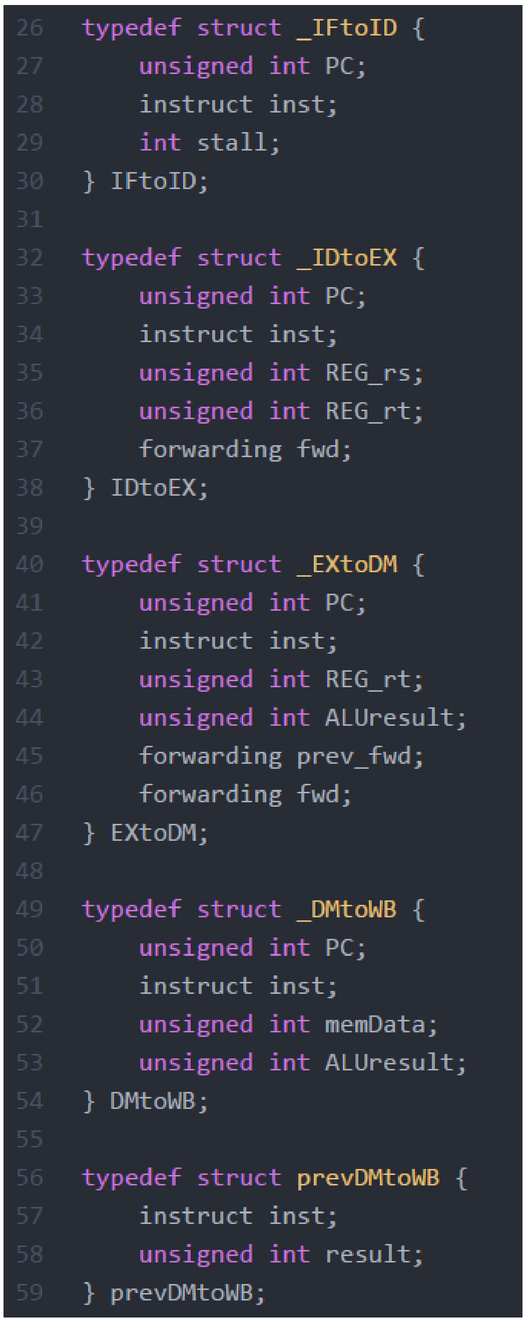
This two file is the same as the project 1 I have done, I just copy it from the project 1 and reuse in this project.

III. Pipeline & Pipereg part:



```c
typedef struct _instruction {
  unsigned int inst;
  unsigned int opcode;
  char type;
  unsigned int rs;
  unsigned int rt;
  unsigned int rd;
  unsigned int C;
  unsigned int funct;
  char name[5];
} instruct;

typedef struct _forwarding {
  int forward;
  int rs;  //0: no forward, 1: forward form EX/DM, 2:forward form DM/WB
  int rt;  //0: no forward, 1: forward form EX/DM, 2:forward form DM/WB
} forwarding;
```

First, I created two structure to save the instruction and the forwarding command as this picture.



```c
typedef struct _IFtoID {
    unsigned int PC;
    instruct inst;
    int stall;
} IFtoID;

typedef struct _IDtoEX {
    unsigned int PC;
    instruct inst;
    unsigned int REG_rs;
    unsigned int REG_rt;
    forwarding fwd;
} IDtoEX;

typedef struct _EXtoDM {
    unsigned int PC;
    instruct inst;
    unsigned int REG_rt;
    unsigned int ALUresult;
    forwarding prev_fwd;
    forwarding fwd;
} EXtoDM;

typedef struct _DMtoWB {
    unsigned int PC;
    instruct inst;
    unsigned int memData;
    unsigned int ALUresult;
} DMtoWB;

typedef struct prevDMtoWB {
    instruct inst;
    unsigned int result;
} prevDMtoWB;
```

IF/ID data:

Simply save the instruction data load in the IF process and the stall state of IF.

ID/EX data:

Save the instruction data and the register value of rs and rt. The fwd will save the forwarding state of ID.

EX/DM data:

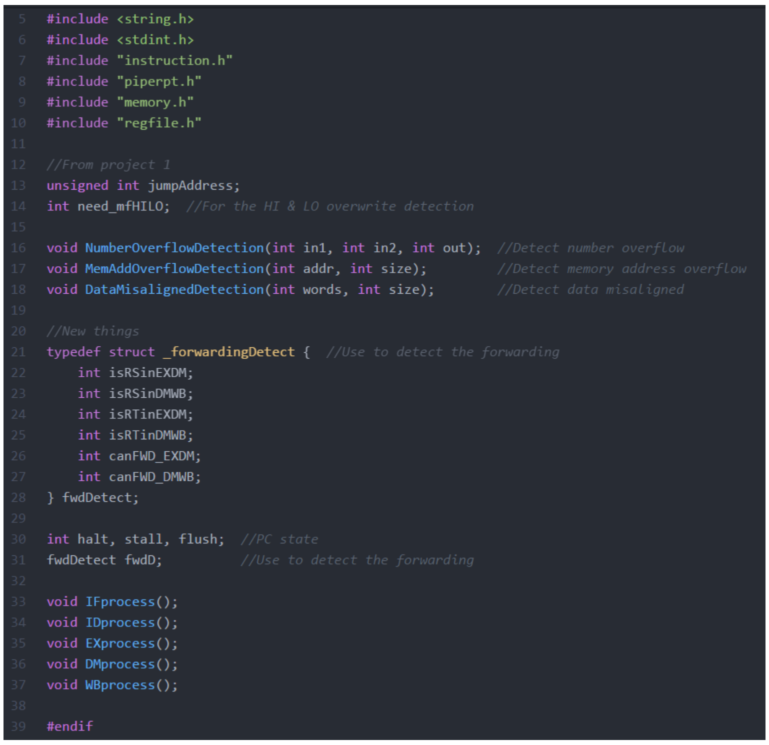
Save the instruction data and the ALU result. The forwarding data, previous one is to save the state of this cycle, and the other one is to save the state of next cycle.

DM/WB data:

Save the instruction data, the ALU result and memory data.

Previous DM/WB data:

Save the instruction data and the result of ALU or memory data.



```c
#include <string.h>
#include <stdint.h>
#include "instruction.h"
#include "piperpt.h"
#include "memory.h"
#include "regfile.h"

//From project 1
unsigned int jumpAddress;
int need_mfHILO;  //For the HI & LO overwrite detection

void NumberOverflowDetection(int in1, int in2, int out);  //Detect number overflow
void MemAddOverflowDetection(int addr, int size);         //Detect memory address overflow
void DataMisalignedDetection(int words, int size);        //Detect data misaligned

//New things
typedef struct _forwardingDetect {  //Use to detect the forwarding
    int isRSinEXDM;
    int isRSinDMWB;
    int isRTinEXDM;
    int isRTinDMWB;
    int canFWD_EXDM;
    int canFWD_DMWB;
} fwdDetect;

int halt, stall, flush;  //PC state
fwdDetect fwdD;          //Use to detect the forwarding

void IFprocess();
void IDprocess();
void EXprocess();
void DMprocess();
void WBprocess();

#endif
```

<- pipeline.h

I copy the error detection from the project 1 and reuse it in this project. And I created a structure to save the state of forwarding detection, it will be use in the ID processing part. The rough detail of each process is:

IF: Get the instruction from the instruction memory, decode and save in the IF/ID data, check the stall and update the stall state of the next cycle.

ID: Check the stall state first, if yes, update the register only, if not, get the new instruction. Then, check the instruction and do the forwarding detection, if it is branch instruction, do the forwarding, others will update the forwarding information in the EX/DM pipeline data and wait the next cycle EX process to do the forwarding. Final, do the flush detection.

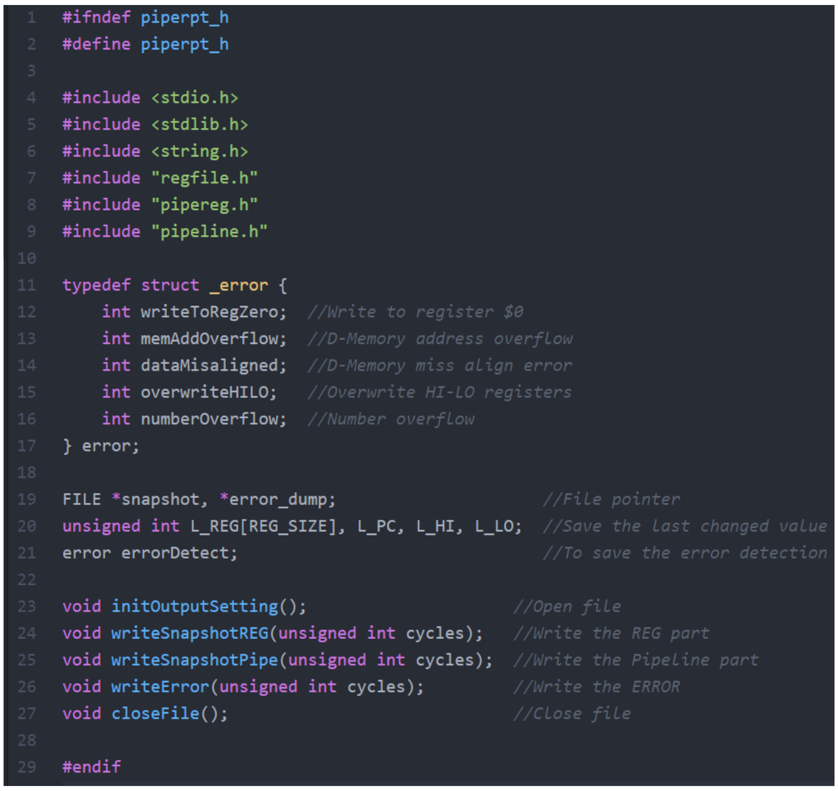
EX: Check the stall state, if yes, add a NOP. If not, get the new instruction and check if it need to do forwarding and calculate the ALU result.

DM: Get the new instruction and do the load word and store word in this process if needed.

WB: Get the new instruction do the write back register in this process if needed.

IV. Simulate & Piperpt part:

The simulate file is to call the initial part and run every processing in a while loop, it is the big picture of the simulation. And the pipeline processing is following “WB->DM->EX->ID->IF” order.

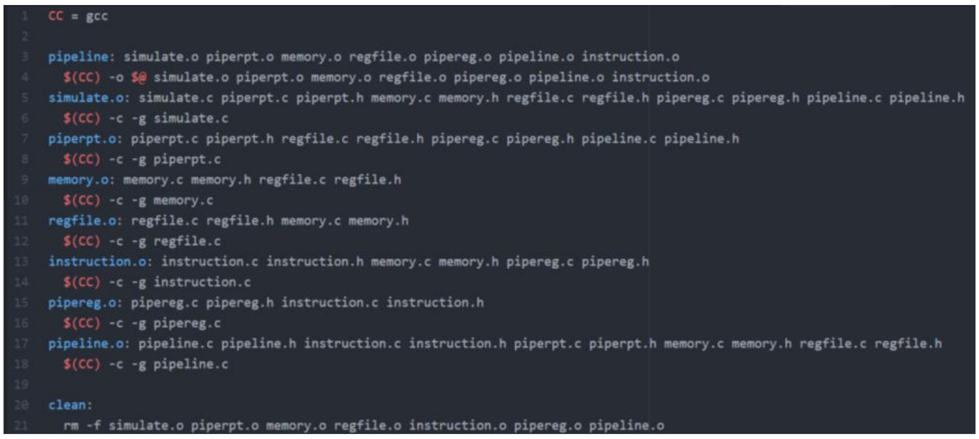


```c
#ifndef piperpt_h
#define piperpt_h

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "regfile.h"
#include "pipereg.h"
#include "pipeline.h"

typedef struct _error {
    int writeToRegZero;  //Write to register $0
    int memAddOverflow;  //D-Memory address overflow
    int dataMisaligned;  //D-Memory miss align error
    int overwriteHILO;   //Overwrite HI-LO registers
    int numberOverflow;  //Number overflow
} error;

FILE *snapshot, *error_dump;                        //File pointer
unsigned int L_REG[REG_SIZE], L_PC, L_HI, L_LO;     //Save the last changed value
error errorDetect;                                  //To save the error detection

void initOutputSetting();                       //Open file
void writeSnapshotREG(unsigned int cycles);     //Write the REG part
void writeSnapshotPipe(unsigned int cycles);    //Write the Pipeline part
void writeError(unsigned int cycles);           //Write the ERROR
void closeFile();                               //Close file

#endif
```

<- piperpt.h

The piperpt part is copy the output file function from the project 1 and reuse it in this project, but I had spin off the “writeSnapshot” function in two function, one is write the register change and the other one is write the pipeline state.

V. Makefile part:

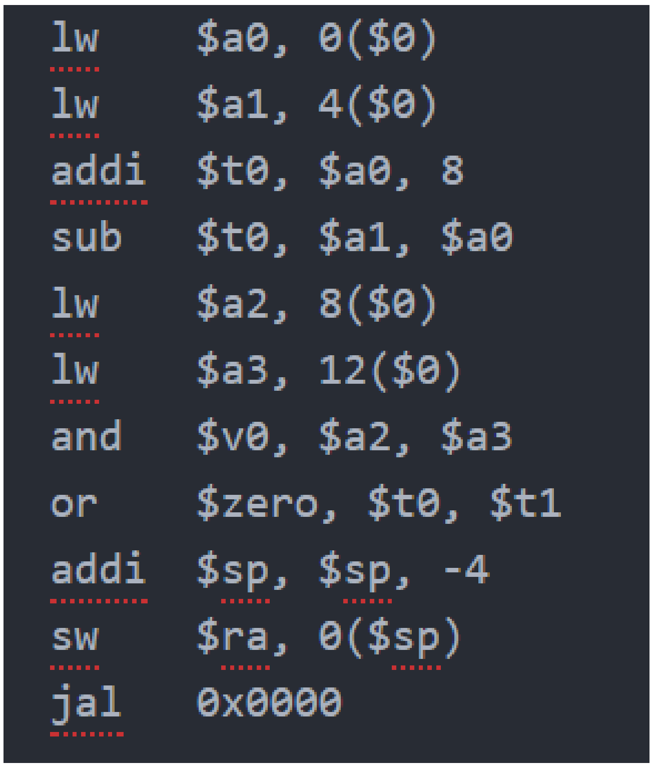


```makefile
CC = gcc

pipeline: simulate.o piperpt.o memory.o regfile.o pipereg.o pipeline.o instruction.o
  $(CC) -o $@ simulate.o piperpt.o memory.o regfile.o pipereg.o pipeline.o instruction.o
simulate.o: simulate.c piperpt.c piperpt.h memory.c memory.h regfile.c regfile.h pipereg.c pipereg.h pipeline.c pipeline.h
  $(CC) -c -g simulate.c
piperpt.o: piperpt.c piperpt.h regfile.c regfile.h pipereg.c pipereg.h pipeline.c pipeline.h
  $(CC) -c -g piperpt.c
memory.o: memory.c memory.h regfile.c regfile.h
  $(CC) -c -g memory.c
regfile.o: regfile.c regfile.h memory.c memory.h
  $(CC) -c -g regfile.c
instruction.o: instruction.c instruction.h memory.c memory.h pipereg.c pipereg.h
  $(CC) -c -g instruction.c
pipereg.o: pipereg.c pipereg.h instruction.c instruction.h
  $(CC) -c -g pipereg.c
pipeline.o: pipeline.c pipeline.h instruction.c instruction.h piperpt.c piperpt.h memory.c memory.h regfile.c regfile.h
  $(CC) -c -g pipeline.c

clean:
  rm -f simulate.o piperpt.o memory.o regfile.o instruction.o pipereg.o pipeline.o
```

Just using a simple way to include all the file that I need to compile the target “*.o”. And also the clean up function.

2) Test case Design

2-1) Detail Description of Test case:



```
lw      $a0, 0($0)
lw      $a1, 4($0)
addi    $t0, $a0, 8
sub     $t0, $a1, $a0
lw      $a2, 8($0)
lw      $a3, 12($0)
and     $v0, $a2, $a3
or      $zero, $t0, $t1
addi    $sp, $sp, -4
sw      $ra, 0($sp)
jal     0x0000
```

A simple test case to test some instruction processing, write $0 error and address overflow. And because I set the stack pointer in the 0x00000800, so if the simulation is correct, there will have an address overflow.