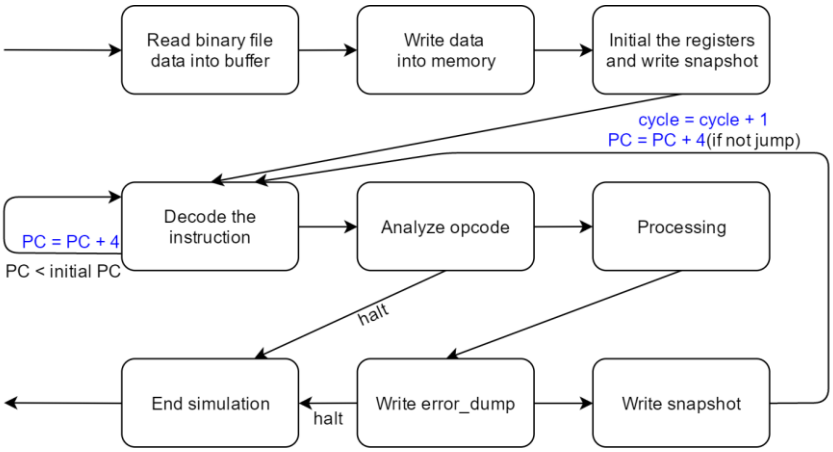
Computer Architecture Project 1 Report

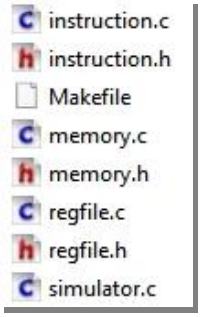
104062261 葉毓浩

1) Project Description

1-1) Program Flow chart:



1-2) Detailed Description:



The file I created in my project is modularize almost same as the suggestion.

I. Simulator part:

```
simulator.c
FILE *snapshot, *error_dump;
void writeSnapshot(unsigned int cycles);
void writeError(unsigned int cycles);

int main()
{
  //Initial the simulator
  unsigned int cycles, halt, instruction;
  snapshot = fopen("snapshot.rpt", "w");
  error_dump = fopen("error_dump.rpt", "w");
  readBin();
  writeMem();
  cycles = 0;
  halt = 0;
  initREG();
  writeSnapshot(cycles);
  //Start the simulation
  while(halt!=1)
  {
    cycles ++;
    writeToRegZero = 0;
    numberOverflow = 0;
    overwriteHILO = 0;
    memAddOverflow = 0;
    dataMisaligned = 0;
    halt = doInstruction();
    if(halt!=1) writeSnapshot(cycles);
    writeError(cycles);
  }
  fclose(snapshot);
  fclose(error_dump);
  return 0;
}
```

The main function and two file write function “snapshot.rpt” and “error_dump.rpt” was put in here. The main function is the big picture of the single-cycle processor. The main process is written in the main function.

II. Memory part:

memory.h

```
#ifndef memory_h
#define memory_h

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

#define MEM_SIZE 1024

FILE *iimage, *dimage;
unsigned char *iBuffer, *dBuffer;

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

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

#endif
```

The processor memory is written in here. And the read binary file and write memory function is also written in here. Using buffer to save the data read form the binary file, first initial the “PC” and “Stack Pointer” address, then according to the number of words that mentioned in the binary file and write the words into the instruction memory and data memory.

III. Register file part:

regfile.h

```
#ifndef regfile_h
#define regfile_h

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

#define REG_SIZE 32

unsigned int initPC;  //Save the initial PC address
unsigned int L_REG[REG_SIZE], L_PC, L_HI, L_LO;  //Save the last changed value

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

void initREG();  //Initial all the register

#endif
```

The register value, the register last changed value and the initial PC address is save in this part, an initial register function is written in here too. That initial function is copy the register value to the last changed register before the simulator is going to work.

IV. Instruction part:

instruction.h

```
#ifndef instruction_h
#define instruction_h

#include <stdio.h>
#include <string.h>
#include <stdint.h>
#include "memory.h"
#include "regfile.h"

unsigned int instruction, opcode, rs, rt, rd, shamt, funct, immediate, address;
int writeToRegZero, numberOverflow, overwriteHILO, memAddOverflow, dataMisaligned;
int need_mfHILO;  //For the HI & LO overwrite detection

int doInstruction();  //If detected halt, return 1
int decode();         //Decode the instruction, if detected halt, return 1
int rType(char *command);  //Deal with R-Type instruction, if detected halt, return 1
int iType(char *command);  //Deal with I-Type instruction, if detected halt, return 1
int jType(char *command);  //Deal with J-type instruction, if detected halt, return 1
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

#endif
```

It is the heart of my simulator, many important function is written in here. The “doInstruction” function is to get the instruction form the instruction memory and call the “decode” function to handle what to do. Then depending on the instruction and call “type” function to do the next step. In the “type” function, it will do the right process and detect is there any error command. The error will be recorded and print it out in the “error_dump.rpt”.

V. Some detail in the decoding part:

```
}else if(strcmp(command, "sub")==0)
{
  //printf("0x%08X: sub $%u, $%u, $%u\n", PC, rd, rs, rt);
  if(rd==0) writeToRegZero = 1;
  unsigned int result = (int32_t)REG[rs] - (int32_t)REG[rt];
  NumberOverflowDetection(REG[rs], REG[rt]*(-1), result);  //a - b = a + (-b)
  if(writeToRegZero!=1) REG[rd] = result;
```

The number overflow detection in subtract is special. Turn the formula “A - B” into “A + (-B)” so that we can use the number overflow detection with the addition function together.

```
}else if(strcmp(command, "mult")==0)
{
  if(need_mfHILO==1) overwriteHILO = 1;
  //printf("0x%08X: mult $%u, $%u\n", PC, rs, rt);
  int64_t R_rs = (int32_t)REG[rs];
  int64_t R_rt = (int32_t)REG[rt];
  uint64_t result = R_rs * R_rt;
  //printf("%lld * %lld = %lld\n", R_rs, R_rt, result);
  HI = result >> 32;
  LO = result << 32 >> 32;
  need_mfHILO = 1;
```

In multiple function we can use a 64-bits signed number to save the result. But don’t forget to transform 32-bits signed number into 64-bits number first, otherwise it may trigger some bugs. Because if the number is a 32-bits negative number, we have to do sign extension before we do the multiple.

```
immediate = (short int)instruction << 16 >> 16;  //Immediate is not unsigned int!!!!
```

Immediate is a signed number. After saving it as an unsigned number, we should remember every time we use it as a signed number we should transform it back to a signed number.

VI. Makefile part:

Makefile

```
1  CC = gcc
2
3  single_cycle: simulator.o memory.o regfile.o instruction.o
4    $(CC) -o $@ simulator.o memory.o regfile.o instruction.o
5  simulator.o: simulator.c memory.c memory.h regfile.c regfile.h instruction.c instruction.h
6    $(CC) -c -g simulator.c
7  memory.o: memory.c memory.h regfile.c regfile.h
8    $(CC) -c -g memory.c
9  regfile.o: regfile.c regfile.h memory.c memory.h
10   $(CC) -c -g regfile.c
11 instrucion.o: instruction.c instruction.h regfile.c regfile.h memory.c memory.h
12   $(CC) -c -g instruction.c
13 clean:
14   rm -f simulator.o memory.o regfile.o instruction.o
```

Makefile can help us easier to compile the new program after we do some changes. And we can write a clean command to help us clean up the “*.o” file after we compile the new program.

2) Test case Design

2-1) Detail Description of Test case:

testcase.S

```
1  lw   $a0, 0($zero)
2  lw   $a1, 0($zero)
3  addi $t0, $a0, 8
4  sub  $t1, $a1, $a0
5  lw   $a2, 8($zero)
6  lw   $a3, 12($zero)
7  and  $v0, $a2, $a3
8  or   $zero, $t0, $t1
9  addi $sp, $sp, -4
10 sw  $ra, 0($sp)
```

dimage.bin

Offset(h)	00	01	02	03	
00000000	00	00	08	00	□...
00000004	00	00	00	04	....
00000008	FF	FF	FF	FC	ÿÿÿü
0000000C	00	0C	87	63	..‡c
00000010	00	A4	87	65	.¤‡e
00000014	00	02	04	80	...€

A simple test case to test some instruction processing, write $0 error and address overflow. Because I set the initial $sp in 0x00000800(2048), so in line 10 it will be address overflow and halt the simulation.