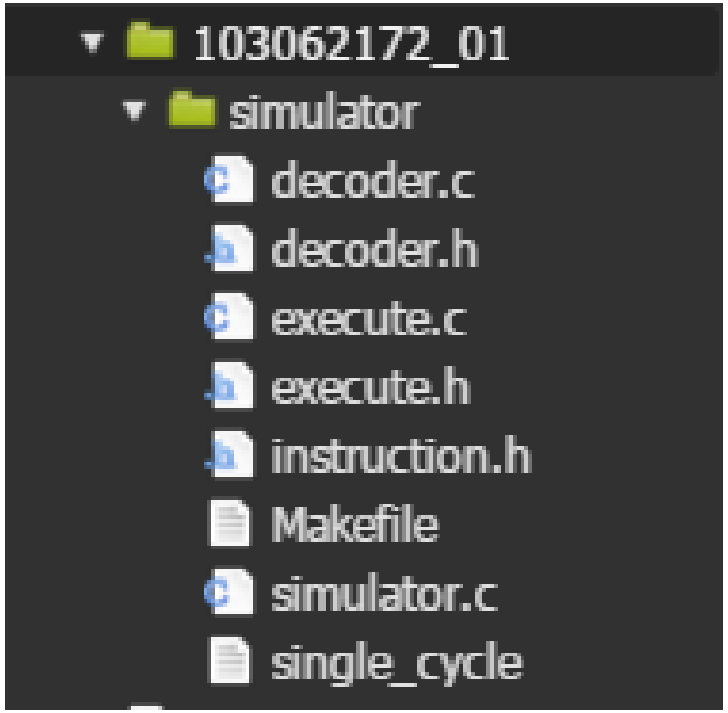
Architecture Project 1

103062172 黃心佑

程式結構



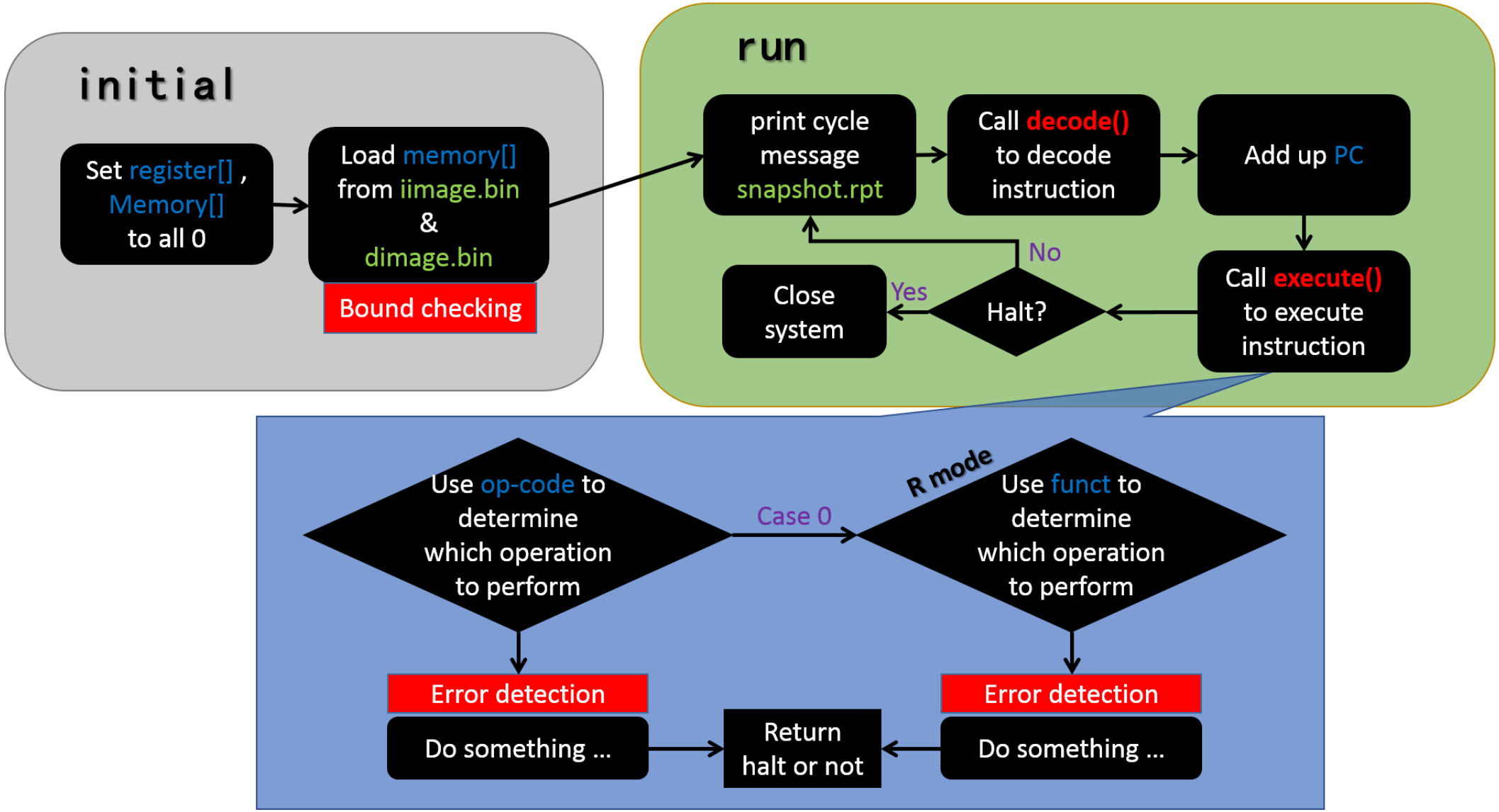
Decoder.c : 將raw instruction存進自己定義的資料結構(instruction)內。

Execute.c : 將一個instruction解讀並執行及錯誤偵測、處理。

Instruction.h : 一個資料結構，用來儲存指令。

Simulator.c : 程式進入點(main)，將memory(I&D)讀進程式，
操作PC讀取指令(IF)，呼叫decoder(ID)，
再把指令送進execute(MEM,EX,WB)執行。

Flowchart


initial
Set register[] , Memory[] to all 0
Load memory[] from iimage.bin & dimage.bin
Bound checking
run
print cycle message snapshot.rpt
Call decode() to decode instruction
Add up PC
Call execute() to execute instruction
Halt?
No
Yes
Close system
Use op-code to determine which operation to perform
Case 0
R mode
Use funct to determine which operation to perform
Error detection
Do something ...
Return halt or not
Error detection
Do something ...

In simulator.c

```
37 //--translate from little endian to big endian--//
38 int read_4_bytes(FILE *fptr){
39     unsigned char buf[4];// 4 * 8 bytes
40     int i=0;
41     int x = 0;
42     fread(buf,1,4,fptr);
43     for(i=0;i<4;i++){
44         x = x<<8;
45         x += buf[i];
46     }
47     return x;
48 }
49
```

- **Int read_4_bytes(FILE*)** :

  因為存在iimage.bin及dimage.bin內的格式為little endian ，固需要讀入一個word(4 bytes)時，就必須做一些處理，這個function會從傳入的file pointer讀入4 bytes並存在一個integer內，再回傳。(主要用於讀取PC、I mem size、SP 、D mem size)。

- **get/set functions :**

  我將一些常用的陣列及變數如register[]、D_memory[]、pc、sp宣告在simulator.c中，因為C本身不支援物件導向，要**跨檔案**來存取這些變數就只能靠get/set function。

In instruction.h

```
1  #ifndef Instruction_h
2  #define Instruction_h
3
4  struct instructions {
5      unsigned int opcode;
6      unsigned int rs,rt,rd;
7      unsigned int shamt;
8      unsigned int func;
9
10     unsigned int constant_16;
11     unsigned int constant_26;
12 };
13 typedef struct instructions instruction;
14
15 #endif
```

先不管要執行什麼指令，
把全部變數按照它在**instruction**
中對應的位置讀進來就對了。

In **decoder.c**

- **Int decode(unsigned char[] raw_inst ,int pc , instruction *inst) :**

  將raw instruction從raw_inst[pc] (raw_inst其實就是I_memory)讀出，並依照bit的位置把raw instruction填入inst內的變數中(op-code、rs、rt...)。

```c
#include<stdio.h>
#include"instruction.h"
int decode(unsigned char raw_inst[] , int pc , instruction *inst){
    inst->opcode = (raw_inst[pc])>>2;
    inst->rs = ((raw_inst[pc]&0x03)<<3) + (raw_inst[pc+1]>>5);
    inst->rt = (raw_inst[pc+1]&0x1F);
    inst->rd = (raw_inst[pc+2]>>3);
    inst->shamt = ((raw_inst[pc+2]&0x07)<<2) + (raw_inst[pc+3]>>6);
    inst->func = raw_inst[pc+3]&0x3F;

    inst->constant_16 = ((char)raw_inst[pc+2]<<8) | raw_inst[pc+3]; //<----- negative handling...

    inst->constant_26 = ((char)(raw_inst[pc]&0x03)<<24) | ((raw_inst[pc+1])<<16) |
                                    ((raw_inst[pc+2])<<8) | raw_inst[pc+3]; //<-- here too
    return 0;
}

```

In **execute. c （1/3)**

- **Int execute(instruction *inst , FILE *fError) :**

  根據inst執行指令，並將錯誤訊息印到fError所指的file(error_dump.rpt)。

  首先根據inst->opcode辨識operation，如果是0，呼叫R_mode()，否則執行該指令。如需停止系統(halt)，回傳值設為1。

- **Int R_mode(instruction *inst , FILE *fError) :**

  延續execute()的功能，因R mode的op-code均為零，須對此instruction的funct辨識才可以知道operation。如需停止系統(halt)，回傳值設為1。

Note : 在所有指令(除sll)執行前，都會先呼叫errorDetect() (next page)進行錯誤偵測。

In **execute. c （2/3)**

- Int errorDetect(int W0E_flag,int NO_flag ,int AO_flag ,int ME_flag ,
  int leftReg ,int right_val_1 ,int right_val_2 ,FILE *fError) :
  錯誤偵測。如需halt回傳值設為1。

  W0E_flag : 要偵測Write $0 Error，設此flag為1。
  NO_flag : 要偵測Number Overflow，設此flag為1。
  AO_flag : 要偵測Address Overflow，設此flag為使用寬度(byte)。
  ME_flag : 要偵測Misalignment Error ，設此flag為使用寬度(byte)。
  leftReg : 等號左邊的register 的number。
  right_val_1、 right_val_2 : 等號右邊的兩個參數。
  fError : 指向error_dump.rpt的file pointer。

In **execute.c** (3/3)

- How to detect error?
  - Write $0 Error : 如果左邊register number =0，就report。

    Note : 在setReg()中有進行處理，所以這裡只做report。
  - Number Overflow : 等號右邊兩數的MSB如果相同，相加後應該還是相同。
  - Address Overflow : 檢查等號右兩數相加後再加上寬度是否超過bound。以及等號右兩數相加後是否小於零。
  - Misalignment Error : 檢查等號右兩數相加後是否被該寬度整除。

Testcase design

- 針對每個operation進行功能測試，測試功能是否正確。
- 對所有operation做Write $0 error測試。
- 針對number overflow進行測試，主要分成R mode跟I mode:
  - R mode: add 2147483647 -1(no overflow)
    sub -2147483648 -2147483648(overflow) <-最容易犯錯
    sub 0 -2147483648(no overflow) <-次容易犯錯
  - I mode: addi -1 -32768(16 bits) (no overflow) <-測試16bits正負號轉換
    addi -2147483648 32768(16bits) (overflow) <-測試16bits正負號轉換
- NOP : 只有inst全零的狀況才是NOP，可能會有人多處理。
- Error priority: 一方面測試是否有考慮會有多種error同時發生，一方面測試是否有注意error report的順序。