計算機組織

實驗五

結果報告

第十二組

組員:

|  |  |
| --- | --- |
| 學號 | 姓名 |
| E24099059 | 陳旭祺 |
| F74044046 | 蕭佑永 |

日期:2020/11/03

一、實驗內容

(1)實作一

1.題目:Fill the Following Space

Useful Information:

Custom Instrution

RISC-V Addressing modes

一張含有 桌 的圖片

自動產生的描述

第一部分RISCVInstrFormats.td

def OPC\_CUSTOM : RISCVOpcode<0b0001011>;

class RVInstR\_CUSTOM<bits<7> funct7, bits<3> funct3, RISCVOpcode opcode, dag outs, dag ins, string opcodestr, string argstr>

: RVInst<outs, ins, opcodestr, argstr, [], InstFormatR> {

bits<5> rs2;

bits<5> rs1;

bits<5> rd;

let Inst{31-25} = funct7;

// 類似Risc-V R-type addressing modes

let Inst{24-20} = rs2;

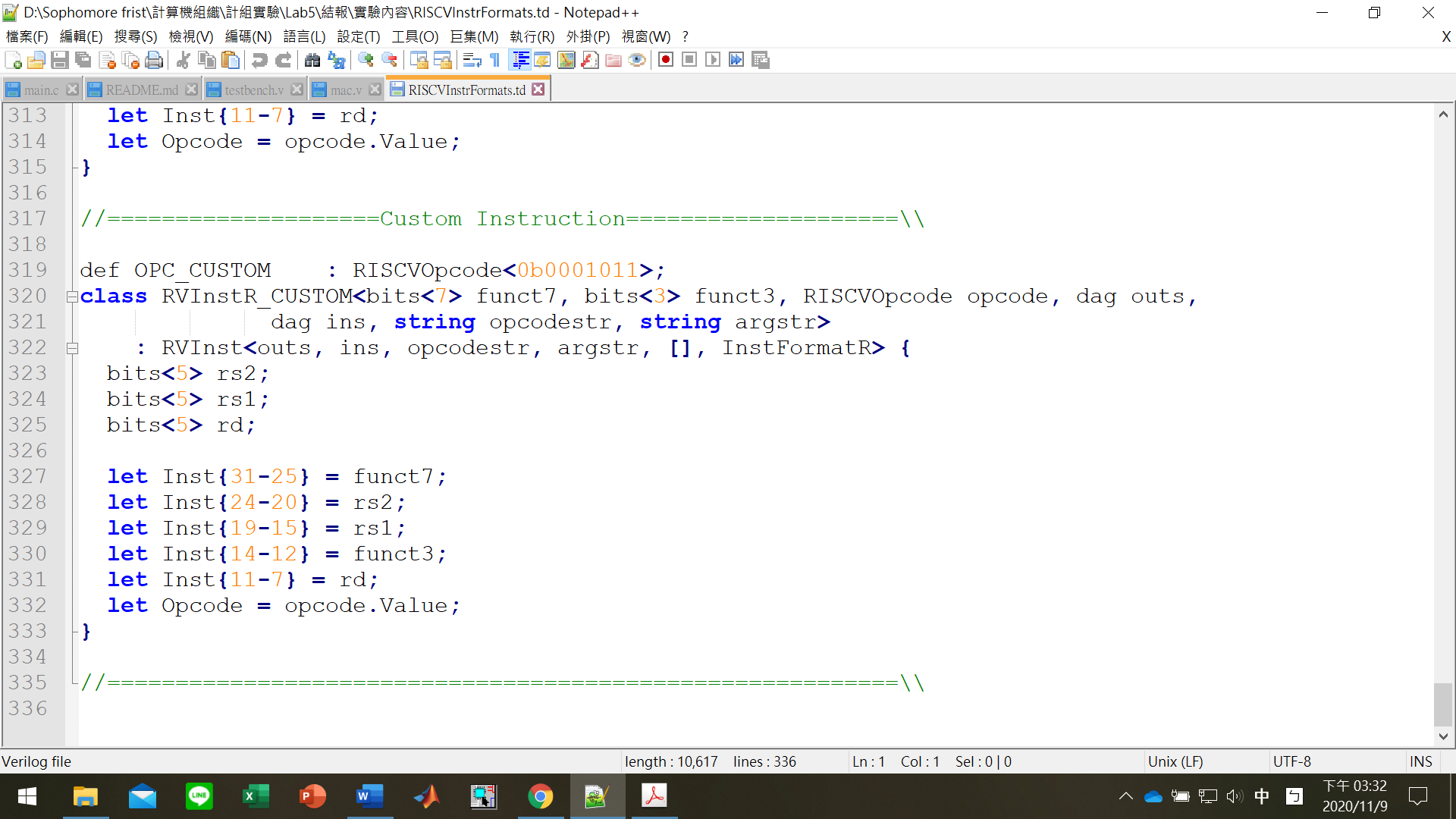
let Inst{19-15} = rs1;

let Inst{14-12} = funct3;

let Inst{11-7} = rd;

let Opcode = opcode.Value;

}



第二部分RISCVInstrInfo.td

let hasSideEffects = 0, mayLoad = 0, mayStore = 0 in

class CUSTOM\_INSTR\_1<bits<7> funct7, bits<3> funct3, string opcodestr>

: RVInstR\_CUSTOM<funct7, funct3, OPC\_CUSTOM, (outs), (ins GPR:$rs1), opcodestr, "$rs1">;

let hasSideEffects = 0, mayLoad = 0, mayStore = 0 in

class CUSTOM\_INSTR\_2<bits<7> funct7, bits<3> funct3, string opcodestr>

: RVInstR\_CUSTOM<funct7, funct3, OPC\_CUSTOM, (outs GPR:$rd), (ins), opcodestr, "$rd">;

def MLD : CUSTOM\_INSTR\_1<0b0000001,0b011,"custom1">, Sched<[]>{

let rs2 = 0; //參照助教表中自訂指令

let rd = 0;

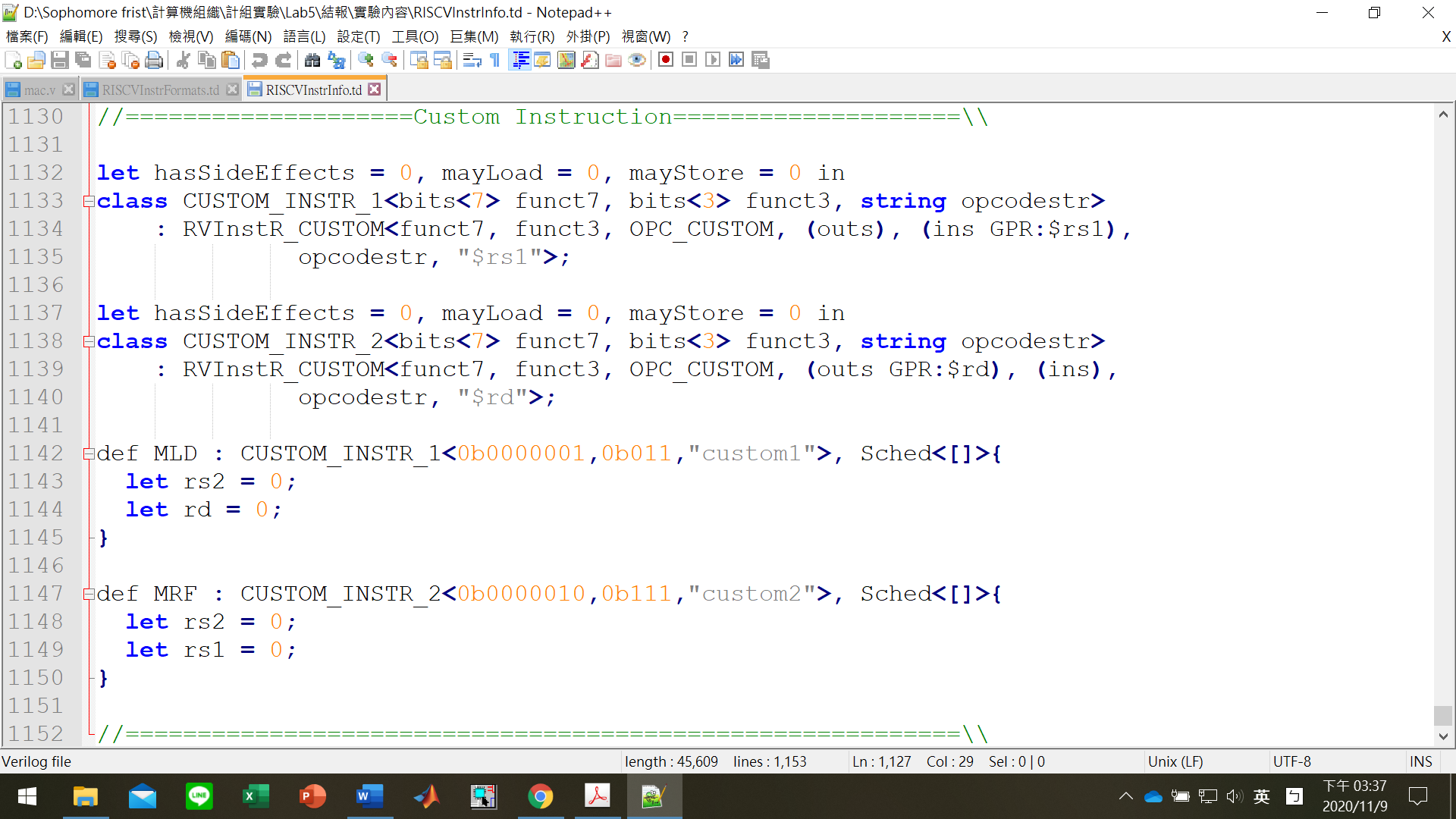
}

def MRF : CUSTOM\_INSTR\_2<0b0000010,0b111,"custom2">, Sched<[]>{

let rs2 = 0;

let rs1 = 0;

}

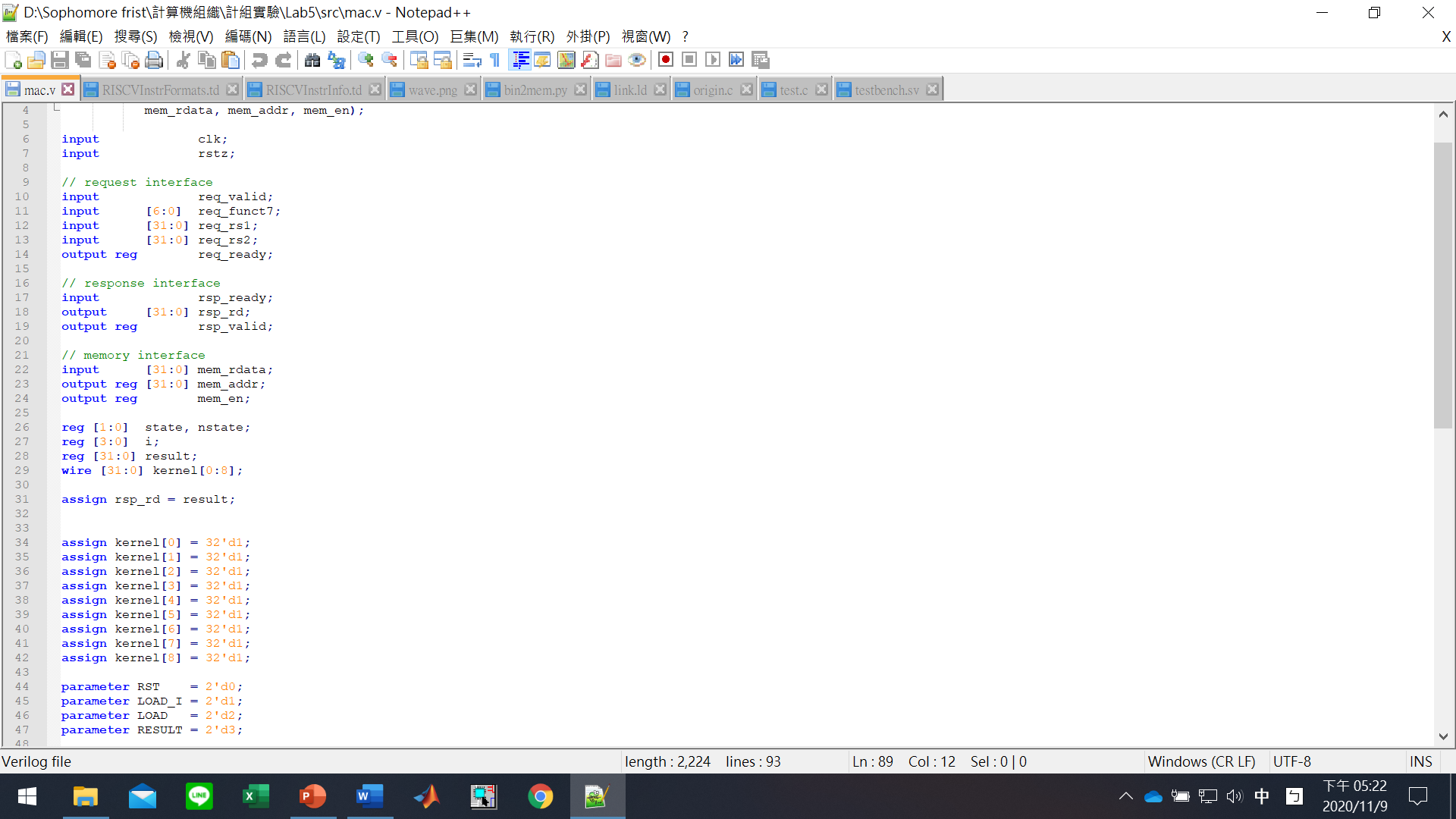


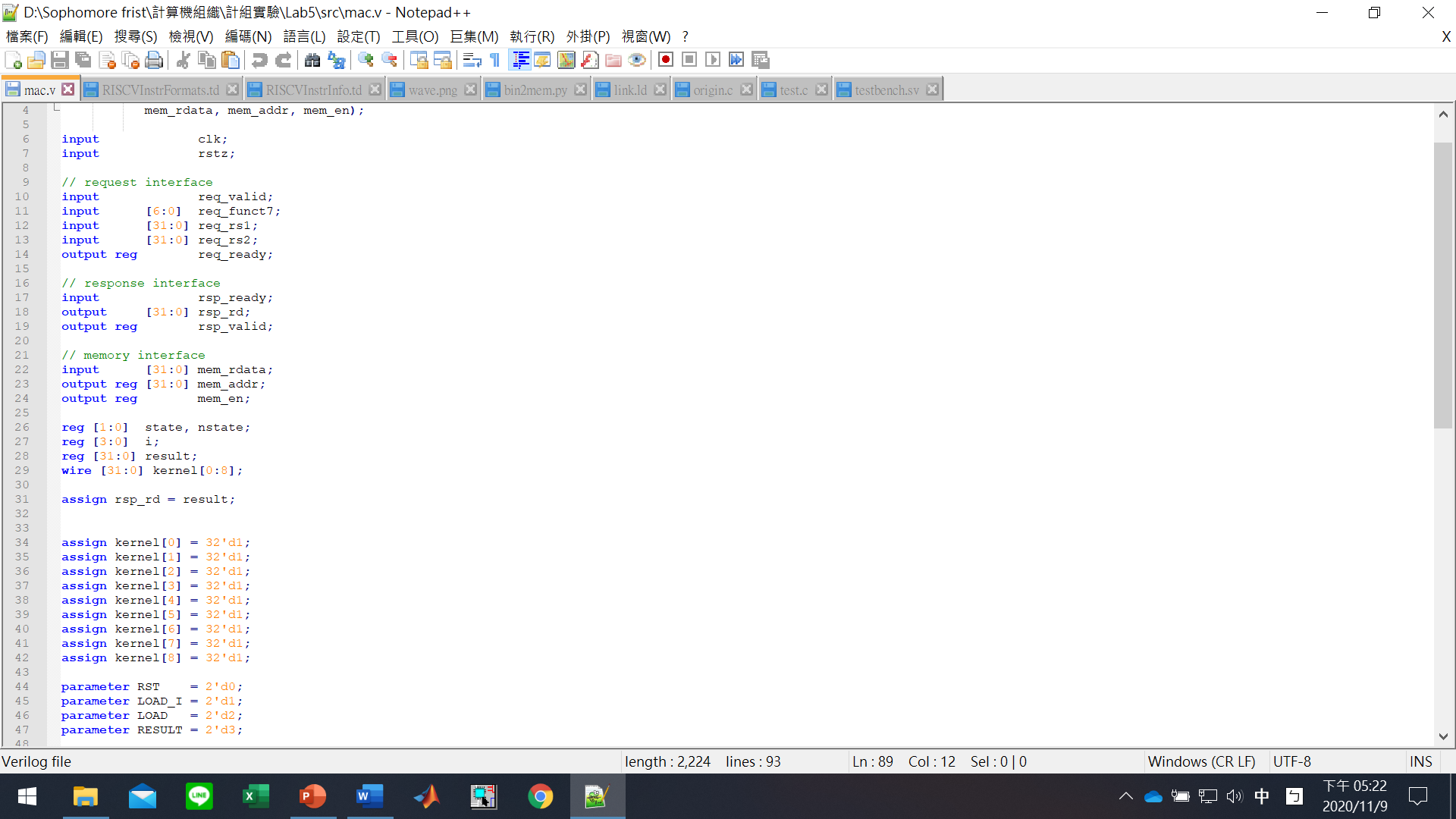
2.結果分析(modelsim):

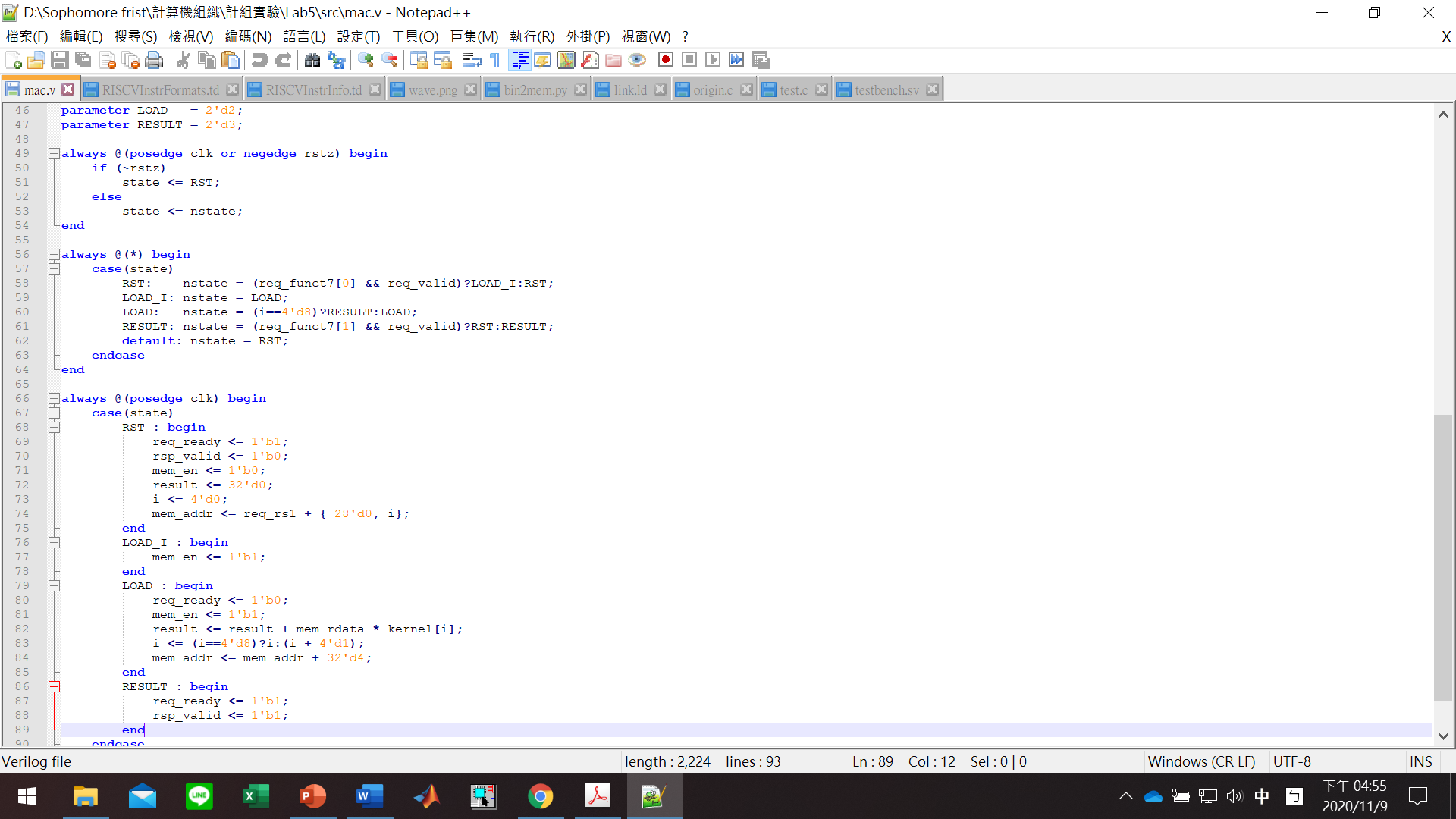
一張含有 建築物, 坐, 大 的圖片

自動產生的描述

X10最後值變成84的來源，可由mac.v檔推測其功能為將我們自訂的暫存器的值藉由如下圖Load與Load Immediately兩種定址抓取值到使用者的暫存器x10，如上圖顯示，才能讓我們看到數值並驗證結果。







(2)挑戰題

1.題目: Compile source with/without optimization

origin.ll

一張含有 文字 的圖片

自動產生的描述

opt.ll

一張含有 文字 的圖片

自動產生的描述

2.結果分析

兩者原C Code應為

|  |
| --- |
| #include <stdio.h>  int call\_add(int a, int b){  return a + b;  }  int main(){  int a = 1;  int b = 2;  int a\_plus\_b = call\_add(a, b);  return 0;  } |

上述即是做1+2=3的簡單運算，但差別是未優化時是compiler是照著C code依序編譯而沒考慮到之後會執行什麼步驟，所以是先分配空間，讀取1、2值並存入記憶體，再讀出到暫存器去做add運算，十分冗長也浪費記憶體空間；而優化後的組合語言十分簡潔，直接在暫存器做運算。由此差別也讓我們了解編譯器優化前後的比較與優化的重要性。

LLVM 簡介

由 Swift 之父 Chris Lattner 創造，並由 Apple 大力贊助，LLVM 最初是 Low Level Virtual Machine 四個字的縮寫，不過隨著專案的發展，LLVM 已經成為了一整套編譯器工具組合，官方也已經放棄 LLVM 四個字的縮寫意義，將專案的簡述訂為「The LLVM Compiler Infrastructure」。

就像官方給的副標一樣，我們可以將 LLVM 視為一個非常模組化的編譯器，對打造語言、研究編譯技術的開發者來說，其強大的地方在於可以將任一個部分抽換掉或插入新的模組，就可以做出一個新的語言、或是將現有程式移植到新硬體平台、或是為特定目的做最佳化後的編譯器。

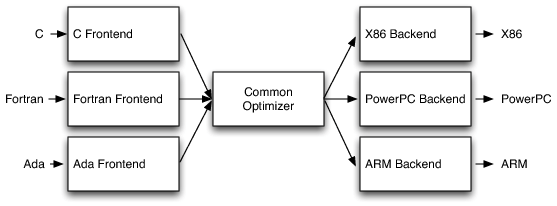
一張含有 文字, 白板 的圖片

自動產生的描述

|  |
| --- |
| 典型的編譯器架構，分成 Frontend (Parser)、Optimizer、Backend (Code Generator) 三部分。Frontend 負責將原始碼解析成語法樹 (或其他適合的資料結構)，Optimizer 負責尋找程式邏輯中有沒有可以簡化來提升執行速度的可能，並改良邏輯 (例如刪除 a = a + 0 這種無意義的指令)，Backend 負責把低階的機器碼寫出來。 |

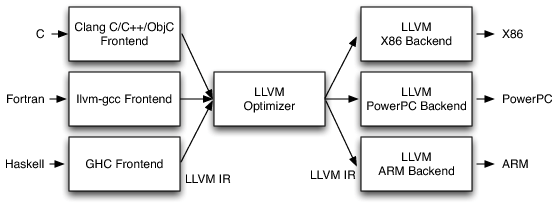
典型的編譯器可以分成 Frontend、Optimizer、Backend 三個部分。理論上我們可以拿 A、B 兩個語言的編譯器，把 B 的頭（Frontend）接到 A 的身體（Optimizer、Backend）上，就能獲的一個將 B 語言編譯到 A 所支援的平台上的新編譯器。

但一般編譯器即使架構上遵循 Frontend、Optimizer、Backend 分工，三者之間通常會做得密不可分，在互相交換資料時都走內部私有 API，這使得要修改現有編譯器來做出自己的編譯器相當困難，不同編譯器之間也無法共用已經實作完成的程式碼。



|  |
| --- |
| 理論上我們只要把典型的編譯器架構的 Frontend 或 Backend 抽換掉，就可以變成不同程式語言到不同平台架構的編譯器。 |

而 LLVM 就是實現了這個理想狀況的編譯器工具包。



|  |
| --- |
| LLVM 編譯器架構。 |

LLVM 定義了一個通用的程式中介表示法，LLVM IR。LLVM IR 是一種類似機器語言，但為了通用性以及給編譯器設計者方便而簡化的版本。在 LLVM 的世界裡，大家都講 LLVM IR：Frontend 把原始語言的邏輯翻譯成 LLVM IR、Optimizer 把 LLVM IR 整理成效率更好的 LLVM IR、Backend 拿到 LLVM IR 來生成機器目標平台的機器語言。如此一來，無論是語言設計者想要創造一個新語言、演算法設計師想改進程式的效能、或是硬體或虛擬機製造者要做一個新的平台，都能得益於 LLVM 的世界，新語言只要設計好 frontend parser 就能使用現有的 optimizer 技術並編譯到各種不同平台、新的 optimizer 可以套用在各種語言和平台的編譯器上、新硬體或虛擬機只要支援 LLVM IR 就可以在上面跑各種主流語言。(擷取自: <http://www.aosabook.org/en/llvm.html>)

二、實驗心得(心得可隨意發揮，但是內容會做工程認證保留檔案)

陳旭祺:

這次實驗讓我們了解到LLVM(Low Level Virtual Machine)模組化的編譯器強大的功能，讓我們能修改RISC-V的backend，藉由類似RISC-V R-type定址格式，決定此架構下的機器語言指令對應的運算數。而挑戰題藉由比較兩個LLVM組合語言的檔案讓我們了解到程式優化前後的差別。這次實驗一開始我不太了解其架構，實驗後到LLVM官網查了不少資料才比較清楚。

蕭佑永: