Computer Organization

Lab 4

Zhengyu Li

Zlr25

Caleb Lee

Calzy8

Nicholas Bouckaert

Nabyq6

Objective:

The objective of this lab was to extend the pipelined MIPS Simulator that we developed in Lab 3. For this portion of the simulator we want it to be able to handle data hazards and execute the program appropriately. To do this we implemented both pipeline stalls and data forwarding

Distribution:

For this portion of the lab most of the coding was done by Caleb Lee with the

assistance and communication of Zhengyu and Nicholas. We all would talk about the proper way we feel the lab should be implementing these next steps.The report then fell upon Nicholas and Caleb to write up, for the most part. As Zhengyu will be coding the next lab.

Milestones:

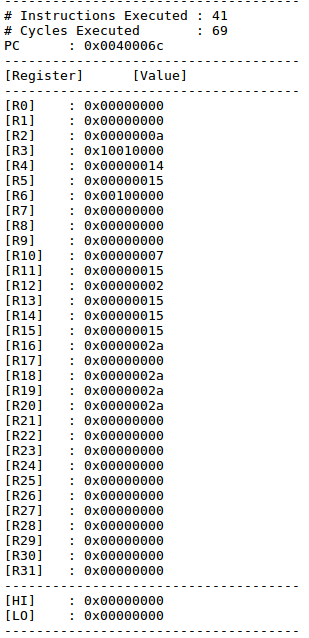
First major milestone was understanding exactly what the goal of the lab was to do. As the whole idea was clear at first, then we discussed different implementation methods. As one person would understand how to code it in one way while the other would see it a different way of coding. So as a group we had to come to a consensus on the approach. Other than that after coming to a consensus the implementation was rather straight forward.

Implementation:

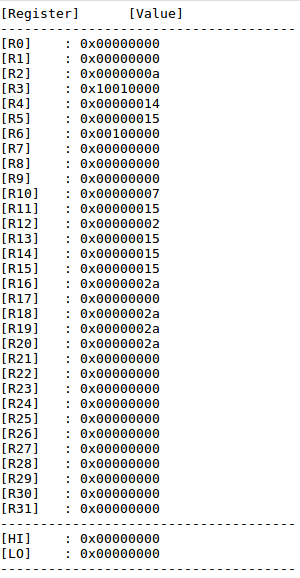
For stalling, we used a simple check of the registers and set a stall flag equal to 2 or 1 depending on if it was in the execution stage or the memory stage. If forwarding is enabled, then it sets flags the are dealt with in the EX stage. If stall is not equal to 0, ID\_EX.IR is set to 0. IF only runs if stall is 0. EX stage checks for forwarding and if so it gets the proper values. If stalling is implemented, EX only runs whenever the stalling is finished as well as ID and IF. Writeback decrements stall if stall is greater than 0. IF the MEM\_WB or EX\_MEM registers are 0, then EX, MEM, and WB just return and don’t run to avoid redundant code runs. EX uses a switch statement if forwarding is enabled to check if we need to grab from MEM\_WB.ALUOutput, EX\_MEM.ALUOutput, or MEM\_WB.LMD.

Conclusion:

In this lab we learned how to stall a pipelined MIPS instruction simulator with IF, ID, EX, MEM, and WB stages. This was done through a stall flag that represents how many cycles the pipeline needs to stall for. We also learned how to forward data from different stages based on what kind of instructions and what kind of hazard is generated.



Stalled version



Forwarded version

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <stdint.h>

#include <assert.h>

#include "mu-mips.h"

int ENABLE\_FORWARDING = 0;

int stall = 0;

uint32\_t ID\_EX\_rs = 0;

uint32\_t ID\_EX\_rt = 0;

uint32\_t EX\_MEM\_RegisterRd = 0;

uint32\_t EX\_MEM\_RegisterRt = 0;

uint32\_t MEM\_WB\_RegisterRt = 0;

uint32\_t MEM\_WB\_RegisterRd = 0;

int EX\_MEM\_RegWrite = 1;

int MEM\_WB\_RegWrite = 1;

int forwardA = 0;

int forwardB = 0;

uint32\_t prevInstr = 0;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Print out a list of commands available

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*/

void help() {

printf("------------------------------------------------------------------\n\n");

printf("\t\*\*\*\*\*\*\*\*\*\*MU-MIPS Help MENU\*\*\*\*\*\*\*\*\*\*\n\n");

printf("sim\t-- simulate program to completion \n");

printf("run <n>\t-- simulate program for <n> instructions\n");

printf("rdump\t-- dump register values\n");

printf("reset\t-- clears all registers/memory and re-loads the program\n");

printf("input <reg> <val>\t-- set GPR <reg> to <val>\n");

printf("mdump <start> <stop>\t-- dump memory from <start> to <stop> address\n");

printf("high <val>\t-- set the HI register to <val>\n");

printf("low <val>\t-- set the LO register to <val>\n");

printf("print\t-- print the program loaded into memory\n");

printf("show\t-- print the current content of the pipeline registers\n");

printf("?\t-- display help menu\n");

printf("forward\t Set/reset forwarding\n");

printf("quit\t-- exit the simulator\n\n");

printf("------------------------------------------------------------------\n\n");

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Read a 32-bit word from memory \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

uint32\_t mem\_read\_32(uint32\_t address)

{

int i;

for (i = 0; i < NUM\_MEM\_REGION; i++) {

if ( (address >= MEM\_REGIONS[i].begin) && ( address <= MEM\_REGIONS[i].end) ) {

uint32\_t offset = address - MEM\_REGIONS[i].begin;

return (MEM\_REGIONS[i].mem[offset+3] << 24) |

(MEM\_REGIONS[i].mem[offset+2] << 16) |

(MEM\_REGIONS[i].mem[offset+1] << 8) |

(MEM\_REGIONS[i].mem[offset+0] << 0);

}

}

return 0;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Write a 32-bit word to memory \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void mem\_write\_32(uint32\_t address, uint32\_t value)

{

int i;

uint32\_t offset;

for (i = 0; i < NUM\_MEM\_REGION; i++) {

if ( (address >= MEM\_REGIONS[i].begin) && (address <= MEM\_REGIONS[i].end) ) {

offset = address - MEM\_REGIONS[i].begin;

MEM\_REGIONS[i].mem[offset+3] = (value >> 24) & 0xFF;

MEM\_REGIONS[i].mem[offset+2] = (value >> 16) & 0xFF;

MEM\_REGIONS[i].mem[offset+1] = (value >> 8) & 0xFF;

MEM\_REGIONS[i].mem[offset+0] = (value >> 0) & 0xFF;

}

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Execute one cycle \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void cycle() {

handle\_pipeline();

CURRENT\_STATE = NEXT\_STATE;

CYCLE\_COUNT++;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Simulate MIPS for n cycles \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void run(int num\_cycles) {

if (RUN\_FLAG == FALSE) {

printf("Simulation Stopped\n\n");

return;

}

printf("Running simulator for %d cycles...\n\n", num\_cycles);

int i;

for (i = 0; i < num\_cycles; i++) {

if (RUN\_FLAG == FALSE) {

printf("Simulation Stopped.\n\n");

break;

}

cycle();

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* simulate to completion \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void runAll() {

if (RUN\_FLAG == FALSE) {

printf("Simulation Stopped.\n\n");

return;

}

printf("Simulation Started...\n\n");

while (RUN\_FLAG){

cycle();

}

printf("Simulation Finished.\n\n");

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Dump a word-aligned region of memory to the terminal \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void mdump(uint32\_t start, uint32\_t stop) {

uint32\_t address;

printf("-------------------------------------------------------------\n");

printf("Memory content [0x%08x..0x%08x] :\n", start, stop);

printf("-------------------------------------------------------------\n");

printf("\t[Address in Hex (Dec) ]\t[Value]\n");

for (address = start; address <= stop; address += 4){

printf("\t0x%08x (%d) :\t0x%08x\n", address, address, mem\_read\_32(address));

}

printf("\n");

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Dump current values of registers to the teminal \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void rdump() {

int i;

printf("-------------------------------------\n");

printf("Dumping Register Content\n");

printf("-------------------------------------\n");

printf("# Instructions Executed\t: %u\n", INSTRUCTION\_COUNT);

printf("# Cycles Executed\t: %u\n", CYCLE\_COUNT);

printf("PC\t: 0x%08x\n", CURRENT\_STATE.PC);

printf("-------------------------------------\n");

printf("[Register]\t[Value]\n");

printf("-------------------------------------\n");

for (i = 0; i < MIPS\_REGS; i++){

printf("[R%d]\t: 0x%08x\n", i, CURRENT\_STATE.REGS[i]);

}

printf("-------------------------------------\n");

printf("[HI]\t: 0x%08x\n", CURRENT\_STATE.HI);

printf("[LO]\t: 0x%08x\n", CURRENT\_STATE.LO);

printf("-------------------------------------\n");

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Read a command from standard input. \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void handle\_command() {

char buffer[20];

uint32\_t start, stop, cycles;

uint32\_t register\_no;

int register\_value;

int hi\_reg\_value, lo\_reg\_value;

printf("MU-MIPS SIM:> ");

if (scanf("%s", buffer) == EOF){

exit(0);

}

switch(buffer[0]) {

case 'S':

case 's':

if (buffer[1] == 'h' || buffer[1] == 'H'){

show\_pipeline();

} else {

runAll();

}

break;

case 'M':

case 'm':

if (scanf("%x %x", &start, &stop) != 2){

break;

}

mdump(start, stop);

break;

case '?':

help();

break;

case 'Q':

case 'q':

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

printf("Exiting MU-MIPS! Good Bye...\n");

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

exit(0);

case 'R':

case 'r':

if (buffer[1] == 'd' || buffer[1] == 'D'){

rdump();

}else if(buffer[1] == 'e' || buffer[1] == 'E'){

reset();

}

else {

if (scanf("%d", &cycles) != 1) {

break;

}

run(cycles);

}

break;

case 'I':

case 'i':

if (scanf("%u %i", &register\_no, &register\_value) != 2){

break;

}

CURRENT\_STATE.REGS[register\_no] = register\_value;

NEXT\_STATE.REGS[register\_no] = register\_value;

break;

case 'H':

case 'h':

if (scanf("%i", &hi\_reg\_value) != 1){

break;

}

CURRENT\_STATE.HI = hi\_reg\_value;

NEXT\_STATE.HI = hi\_reg\_value;

break;

case 'L':

case 'l':

if (scanf("%i", &lo\_reg\_value) != 1){

break;

}

CURRENT\_STATE.LO = lo\_reg\_value;

NEXT\_STATE.LO = lo\_reg\_value;

break;

case 'P':

case 'p':

print\_program();

break;

case 'f':

if (scanf("%d", &ENABLE\_FORWARDING) != 1) {

break;

}

ENABLE\_FORWARDING == 0 ? printf("Forwarding OFF\n") : printf("Forwarding ON\n");

break;

default:

printf("Invalid Command.\n");

break;

}

}

// COPIED OVER FROM LAB 1 - calebs code

void print\_instruction(uint32\_t addr){

/\*IMPLEMENT THIS\*/

char string[64];

uint32\_t instruction = mem\_read\_32(addr);

uint32\_t op = instruction & 0xFC000000; //opcode

uint32\_t bk = instruction & 0x0000003F; //back of 32 bits to id the op code

uint8\_t rs = (instruction & 0x03E00000) >> 21; //source

uint8\_t rt = (instruction & 0x001F0000) >> 16; //destination

uint16\_t im = instruction & 0x0000FFFF; //immediate / offset

uint8\_t rd = (instruction & 0x0000F800) >> 11; //reg destination (register)

uint8\_t h = (instruction & 0x0000007C0) >> 6; //for shift instructions)

switch(op) {

case 0x00000000: //ADD, ADDU, AND, NOR, OR, SLT, SUB, SUBU, XOR, SLL, SRL, DIV, DIVU, MULT, MULTU, SRA, MFHI, MFLO, MTHI, MTLO, JR, JALR

switch(bk) {

case 0x00000020: //ADD

sprintf(string, "ADD $%d, $%d, $%d\n", rd, rs, rt);

break;

case 0x00000021: //ADDU

sprintf(string, "ADDU $%d, $%d, $%d\n", rd, rs, rt);

break;

case 0x00000024: //AND

sprintf(string, "AND $%d, $%d, $%d\n", rd, rs, rt);

break;

case 0x00000027: //NOR

sprintf(string, "NOR $%d, $%d, $%d\n", rd, rs, rt);

break;

case 0x00000025: //OR

sprintf(string, "OR $%d, $%d, $%d\n", rd, rs, rt);

break;

case 0x0000002A: //SLT

sprintf(string, "SLT $%d, $%d, $%d\n", rd, rs, rt);

break;

case 0x00000022: //SUB

sprintf(string, "SUB $%d, $%d, $%d\n", rd, rs, rt);

break;

case 0x00000023: //SUBU

sprintf(string, "SUBU $%d, $%d, $%d\n", rd, rs, rt);

break;

case 0x00000026: //XOR

sprintf(string, "XOR $%d, $%d, $%d\n", rd, rs, rt);

break;

case 0x00000000: //SLL

sprintf(string, "SLL $%d, $%d, 0x%X\n", rd, rt, h);

break;

case 0x00000002: //SRL

sprintf(string, "SRL $%d, $%d, %d\n", rd, rt, h);

break;

case 0x0000001A: //DIV

sprintf(string, "DIV $%d, $%d\n", rs, rt);

break;

case 0x0000001B: //DIVU

sprintf(string, "DIVU $%d, $%d\n", rs, rt);

break;

case 0x00000018: //MULT

sprintf(string, "MULT $%d, $%d\n", rs, rt);

break;

case 0x00000019: //MULTU

sprintf(string, "MULTU $%d, $%d\n", rs, rt);

break;

case 0x00000003: //SRA

sprintf(string, "SRA $%d, $%d, %d\n", rd, rt, h);

break;

case 0x00000010: //MFHI

sprintf(string, "MFHI $%d\n", rd);

break;

case 0x00000012: //MFLO

sprintf(string, "MFLO $%d\n", rd);

break;

case 0x00000011: //MTHI

sprintf(string, "MTHI $%d, $%d, $%d\n", rd, rs, rt);

break;

case 0x00000013: //MTLO

sprintf(string, "MTLO $%d, $%d, $%d\n", rd, rs, rt);

break;

case 0x00000008: //JR

sprintf(string, "JR $%d\n", rs);

break;

case 0x00000009: //JALR

sprintf(string, "JALR $%d, $%d, $%d\n", rd, rs, rt);

break;

case 0x0000000C: //SYSTEM CALL

sprintf(string, "SYSTEMCALL\n");

break;

}

break;

case 0x04000000:

switch(rt) {

case 0x00000000:

sprintf(string, "BLTZ $%d, %d\n", rs, im);

break;

case 0x00000008:

sprintf(string, "BGEZ $%d, %d\n", rs, im);

break;

}

break;

case 0x20000000:

sprintf(string, "ADDI $%d, $%d, 0x%04X\n", rt, rs, im);

break;

case 0x24000000:

sprintf(string, "ADDIU $%d, $%d, 0x%04X\n", rt, rs, im);

break;

case 0x30000000:

sprintf(string, "ANDI $%d, $%d, 0x%04X\n", rt, rs, im);

break;

case 0x34000000:

sprintf(string, "ORI $%d, $%d, 0x%04X\n", rt, rs, im);

break;

case 0x28000000:

sprintf(string, "SLTI $%d, $%d, 0x%04X\n", rt, rs, im);

break;

case 0x38000000:

sprintf(string, "XORI $%d, $%d, 0x%04X\n", rt, rs, im);

break;

case 0x10000000:

sprintf(string, "BEQ $%d, $%d, %d\n", rs, rt, im);

break;

case 0x14000000:

sprintf(string, "BNE $%d, $%d, %d\n", rs, rt, im);

break;

case 0x18000000:

sprintf(string, "BLEZ $%d, %d\n", rs, im);

break;

case 0x1C000000:

sprintf(string, "BGTZ $%d, %d\n", rs, im);

break;

case 0x08000000:

sprintf(string, "J 0x%X\n", instruction & 0x03FFFFFF);

break;

case 0x0C000000:

sprintf(string, "JAL %u\n", instruction & 0x03FFFFFF);

break;

case 0x80000000:

sprintf(string, "LB $%d, %d ($%d) \n", rt, im, rs);

break;

case 0x84000000:

sprintf(string, "LH $%d, $%d, %d\n", rs, rt, im);

break;

case 0x8C000000:

sprintf(string, "LW $%d, %d ($%d)\n", rt, im, rs);

break;

case 0xA0000000:

sprintf(string, "SB $%d, $%d($%d)\n", rt, im, rs);

break;

case 0xA4000000:

sprintf(string, "SH $%d, 0x%04X, %d\n", rs, rt, im);

break;

case 0xAC000000:

sprintf(string, "SW $%d, 0x%04X ($%d)\n", rt, im, rs);

break;

case 0x3C000000:

sprintf(string, "LUI $%d, 0x%04X\n", rt, im);

break;

}

printf("%s", string);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* reset registers/memory and reload program \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void reset() {

int i;

/\*reset registers\*/

for (i = 0; i < MIPS\_REGS; i++){

CURRENT\_STATE.REGS[i] = 0;

}

CURRENT\_STATE.HI = 0;

CURRENT\_STATE.LO = 0;

for (i = 0; i < NUM\_MEM\_REGION; i++) {

uint32\_t region\_size = MEM\_REGIONS[i].end - MEM\_REGIONS[i].begin + 1;

memset(MEM\_REGIONS[i].mem, 0, region\_size);

}

IF\_ID.PC = 0;

IF\_ID.IR = 0;

IF\_ID.A = 0;

IF\_ID.B = 0;

IF\_ID.imm = 0;

IF\_ID.ALUOutput = 0;

IF\_ID.LMD = 0;

ID\_EX.PC = 0;

ID\_EX.IR = 0;

ID\_EX.A = 0;

ID\_EX.B = 0;

ID\_EX.imm = 0;

ID\_EX.ALUOutput = 0;

ID\_EX.LMD = 0;

EX\_MEM.PC = 0;

EX\_MEM.IR = 0;

EX\_MEM.A = 0;

EX\_MEM.B = 0;

EX\_MEM.imm = 0;

EX\_MEM.ALUOutput = 0;

EX\_MEM.LMD = 0;

MEM\_WB.PC = 0;

MEM\_WB.IR = 0;

MEM\_WB.A = 0;

MEM\_WB.B = 0;

MEM\_WB.imm = 0;

MEM\_WB.ALUOutput = 0;

MEM\_WB.LMD = 0;

/\*load program\*/

load\_program();

/\*reset PC\*/

INSTRUCTION\_COUNT = 0;

CURRENT\_STATE.PC = MEM\_TEXT\_BEGIN;

NEXT\_STATE = CURRENT\_STATE;

RUN\_FLAG = TRUE;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Allocate and set memory to zero \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void init\_memory() {

int i;

for (i = 0; i < NUM\_MEM\_REGION; i++) {

uint32\_t region\_size = MEM\_REGIONS[i].end - MEM\_REGIONS[i].begin + 1;

MEM\_REGIONS[i].mem = malloc(region\_size);

memset(MEM\_REGIONS[i].mem, 0, region\_size);

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* load program into memory \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void load\_program() {

FILE \* fp;

int i, word;

uint32\_t address;

/\* Open program file. \*/

fp = fopen(prog\_file, "r");

if (fp == NULL) {

printf("Error: Can't open program file %s\n", prog\_file);

exit(-1);

}

/\* Read in the program. \*/

i = 0;

while( fscanf(fp, "%x\n", &word) != EOF ) {

address = MEM\_TEXT\_BEGIN + i;

mem\_write\_32(address, word);

printf("writing 0x%08x into address 0x%08x (%d)\n", word, address, address);

i += 4;

}

PROGRAM\_SIZE = i/4;

printf("Program loaded into memory.\n%d words written into memory.\n\n", PROGRAM\_SIZE);

fclose(fp);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* maintain the pipeline \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void handle\_pipeline() {

/\*INSTRUCTION\_COUNT should be incremented when instruction is done\*/

/\*Since we do not have branch/jump instructions, INSTRUCTION\_COUNT should be incremented in WB stage \*/

WB();

MEM();

EX();

ID();

IF();

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* writeback (WB) pipeline stage: \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void WB() {

if(MEM\_WB.IR == 0) {

return;

}

uint32\_t instruction = MEM\_WB.IR;

prevInstr = instruction;

uint32\_t op = (instruction & 0xFC000000)>>26; //opcode

uint32\_t bk = instruction & 0x0000003F; //back of 32 bits to id the op code

uint32\_t rt = (instruction & 0x001F0000) >> 16; //destination

uint32\_t rd = (instruction & 0x0000F800) >> 11; //reg destination (register)

switch( op ) {

case 0x00: //Reg-Reg

if(bk == 0xC && MEM\_WB.ALUOutput == 0xA){

RUN\_FLAG = FALSE;

MEM\_WB.ALUOutput = 0x0;

break;

}

NEXT\_STATE.REGS[rd] = MEM\_WB.ALUOutput;

break;

case 0x23: //LW

NEXT\_STATE.REGS[rt] = MEM\_WB.LMD;

break;

case 0x2B: //SW

break;

default: //Reg-Imm

NEXT\_STATE.REGS[rt] = MEM\_WB.ALUOutput;

break;

}

INSTRUCTION\_COUNT++;

if(stall != 0 ) {

stall--;

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* memory access (MEM) pipeline stage: \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void MEM() {

MEM\_WB.IR = EX\_MEM.IR;

uint32\_t instruction, data, opcode;

instruction = MEM\_WB.IR;

MEM\_WB\_RegisterRt = (instruction & 0x001F0000) >> 16; //reg destination (register)

MEM\_WB\_RegisterRd = (instruction & 0x0000F800) >> 11;

EX\_MEM.A = ID\_EX.A;

opcode = (MEM\_WB.IR & 0xFC000000) >> 26;

if(MEM\_WB.IR == 0) {

return;

}

switch(opcode){

case 0x20: //LB

data = mem\_read\_32(EX\_MEM.ALUOutput);

MEM\_WB.LMD = ((data & 0x000000FF) & 0x8000) > 0 ? (data | 0xFFFFFF00) : ( data & 0x000000FF);

break;

case 0x21: //LH

data = mem\_read\_32(EX\_MEM.ALUOutput);

MEM\_WB.LMD = ((data & 0x000000FF) & 0x8000) > 0 ? (data | 0xFFFF0000) : ( data & 0x0000FFFF);

break;

case 0x23: //LW

data = mem\_read\_32(EX\_MEM.ALUOutput);

MEM\_WB.LMD = data;

break;

case 0x28: //SB

data = mem\_read\_32(EX\_MEM.ALUOutput);

data = (data & 0xFFFFFF00) | (EX\_MEM.B & 0x000000FF);

mem\_write\_32(EX\_MEM.ALUOutput, EX\_MEM.B);

break;

case 0x29: //SH

data = mem\_read\_32(EX\_MEM.ALUOutput);

data = (data & 0xFFFF0000) | (EX\_MEM.B & 0x0000FFFF);

mem\_write\_32(EX\_MEM.ALUOutput, EX\_MEM.B);

break;

case 0x2B: //SW

mem\_write\_32(EX\_MEM.ALUOutput, EX\_MEM.B);

break;

default:

MEM\_WB.ALUOutput = EX\_MEM.ALUOutput;

break;

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* execution (EX) pipeline stage: \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void EX() {

if(stall == 0) {

EX\_MEM.IR = ID\_EX.IR;

uint32\_t instruction = EX\_MEM.IR;

uint32\_t op = (instruction & 0xFC000000) >> 26; //opcode

EX\_MEM\_RegisterRd = (instruction & 0x0000F800) >> 11; //reg destination (register)

uint32\_t bk = instruction & 0x0000003F; //back of 32 bits to id the op code

uint32\_t rt = (instruction & 0x001F0000) >> 16; //destination

uint32\_t im = instruction & 0x0000FFFF;

EX\_MEM\_RegisterRt = rt;

if(EX\_MEM.IR == 0) {

return;

}

if(ENABLE\_FORWARDING){

ID\_EX.A = CURRENT\_STATE.REGS[ID\_EX\_rs];

ID\_EX.B = CURRENT\_STATE.REGS[ID\_EX\_rt];

ID\_EX.imm = im;

if(forwardA == 0x10){

switch((MEM\_WB.IR & 0xFC000000) >> 26){

case 0x23: //LW

ID\_EX.A = MEM\_WB.LMD;

break;

default:

ID\_EX.A = EX\_MEM.ALUOutput;

break;

}

} else if (forwardB == 0x10){

switch((MEM\_WB.IR & 0xFC000000) >> 26){

case 0x23: //LW

ID\_EX.B = MEM\_WB.LMD;

break;

default:

ID\_EX.B = EX\_MEM.ALUOutput;

break;

}

}

if(forwardA == 0x01){

switch((MEM\_WB.IR & 0xFC000000) >> 26){

case 0x23: //LW

ID\_EX.A = MEM\_WB.LMD;

break;

default:

ID\_EX.A = MEM\_WB.ALUOutput;

break;

}

} else if (forwardB == 0x01){

switch((MEM\_WB.IR & 0xFC000000) >> 26){

case 0x23: //LW

ID\_EX.B = MEM\_WB.LMD;

break;

default:

ID\_EX.B = MEM\_WB.ALUOutput;

break;

}

switch((prevInstr & 0xFC000000) >> 26) {

case 0x23: //LW

ID\_EX.B = MEM\_WB.LMD;

break;

default:

ID\_EX.B = MEM\_WB.ALUOutput;

break;

}

}

}

forwardA = 0x00;

forwardB = 0x00;

switch(op) {

case 0x00://backend case or Bk

switch( bk ) {

case 0x00://SLL

EX\_MEM.ALUOutput = ID\_EX.A << ID\_EX.B;

break;

case 0x02://SRL

EX\_MEM.ALUOutput = ID\_EX.A >> ID\_EX.B;

break;

case 0x03://SRA

if(( ID\_EX.A & 0x80000000) == 1) {

EX\_MEM.ALUOutput = ~(~ID\_EX.A >> ID\_EX.B);

} else {

EX\_MEM.ALUOutput = ID\_EX.A >> ID\_EX.B;

}

break;

case 0x0C: //SYSCALL

EX\_MEM.ALUOutput = 0xA;

break;

case 0x10:// MFHI

EX\_MEM.ALUOutput = CURRENT\_STATE.HI;

break;

case 0x11:// MTHI

CURRENT\_STATE.HI = ID\_EX.A;

break;

case 0x12: // MFLO

EX\_MEM.ALUOutput = CURRENT\_STATE.LO;

break;

case 0x13://MTLO

CURRENT\_STATE.LO = ID\_EX.A;

break;

case 0x18: { //MULT

uint64\_t productone, producttwo, multi;

if((ID\_EX.A & 80000000) == 0x80000000) {

productone = 0xFFFFFFFF00000000 | ID\_EX.A;

} else {

productone = 0x00000000FFFFFFFF & ID\_EX. A;

}

if((CURRENT\_STATE.REGS[rt] & 0x80000000) == 0x80000000 ) {

producttwo = 0xFFFFFFFF00000000 | ID\_EX.B;

} else {

producttwo = 0x00000000FFFFFFFF & ID\_EX.B;

}

multi = productone \* producttwo;

NEXT\_STATE.HI = ( multi & 0xFFFFFFFF00000000);

NEXT\_STATE.LO = ( multi \* 0x00000000FFFFFFFF) >> 32;

break;

}

case 0x19: { //MULTU

uint64\_t product;

product = (uint64\_t)ID\_EX.A \* (uint64\_t)ID\_EX.B;

NEXT\_STATE.LO = (product & 0x00000000FFFFFFFF);

NEXT\_STATE.HI = (product & 0xFFFFFFFF00000000) >> 32;

break;

}

case 0x1A: //DIV

if(ID\_EX.B != 0) {

NEXT\_STATE.LO = (int32\_t)ID\_EX.A / (int32\_t)ID\_EX.B;

NEXT\_STATE.HI = (int32\_t)ID\_EX.A % (int32\_t)ID\_EX.B;

}

break;

case 0x1B: //DIVU

if(ID\_EX.B != 0) {

NEXT\_STATE.LO = ID\_EX.A / ID\_EX.B;

NEXT\_STATE.HI = ID\_EX.A % ID\_EX.B;

}

break;

case 0x20: //ADD

EX\_MEM.ALUOutput = ID\_EX.A + ID\_EX.B;

break;

case 0x21: //ADDU

EX\_MEM.ALUOutput = ID\_EX.A + ID\_EX.B;

break;

case 0x22: //SUB

EX\_MEM.ALUOutput = ID\_EX.A - ID\_EX.B;

break;

case 0x23: //SUBU

EX\_MEM.ALUOutput = ID\_EX.A - ID\_EX.B;

break;

case 0x24: //AND

EX\_MEM.ALUOutput = ID\_EX.A & ID\_EX.B;

break;

case 0x25: //OR

EX\_MEM.ALUOutput = ID\_EX.A | ID\_EX.B;

break;

case 0x26: //XOR

EX\_MEM.ALUOutput = ID\_EX.A ^ ID\_EX.B;

break;

case 0x27: //NOR

EX\_MEM.ALUOutput = ~(ID\_EX.A | ID\_EX.B);

break;

case 0x2A: //SLT

if(ID\_EX.A < ID\_EX.B){

EX\_MEM.ALUOutput = 0x1;

} else {

EX\_MEM.ALUOutput = 0x0;

}

break;

}

break; //end of Reg-Reg functions

case 0x08: //ADDI

EX\_MEM.ALUOutput = ID\_EX.A + ID\_EX.imm;

break;

case 0x09: //ADDIU

EX\_MEM.ALUOutput = ID\_EX.A + ((ID\_EX.imm & 0x8000) > 0 ? (ID\_EX.imm |0xFFFF0000) : (ID\_EX.imm & 0x0000FFFF));

break;

case 0x0A: //SLTI

if((int32\_t)ID\_EX.A - (int32\_t)((ID\_EX.imm & 0x8000) > 0 ? (ID\_EX.imm |0xFFFF0000) : (ID\_EX.imm & 0x0000FFFF)) < 0){

EX\_MEM.ALUOutput = 0x1;

} else {

EX\_MEM.ALUOutput = 0x0;

}

break;

case 0x0C: //ANDI

EX\_MEM.ALUOutput = ID\_EX.A ^ (ID\_EX.imm & 0x0000FFFF);

break;

case 0x0D: //ORI

EX\_MEM.ALUOutput = ID\_EX.A | (ID\_EX.imm & 0x0000FFFF);

break;

case 0x0E: //XORI

EX\_MEM.ALUOutput = ID\_EX.A ^ ID\_EX.imm;

break;

case 0x0F: //LUI

EX\_MEM.ALUOutput = ID\_EX.imm << 16;

break;

case 0x20: //LB

EX\_MEM.ALUOutput = ID\_EX.A + ID\_EX.imm;

break;

case 0x21: //LH

EX\_MEM.ALUOutput = ID\_EX.A + ID\_EX.imm;

break;

case 0x23: //LW

EX\_MEM.ALUOutput = ID\_EX.A + ((ID\_EX.imm & 0x8000) > 0 ? (ID\_EX.imm |0xFFFF0000) : (ID\_EX.imm & 0x0000FFFF));

EX\_MEM.B = ID\_EX.B;

break;

case 0x28: //SB

EX\_MEM.ALUOutput = ID\_EX.A + ID\_EX.imm;

break;

case 0x29: //SH

EX\_MEM.ALUOutput = ID\_EX.A + ID\_EX.imm;

break;

case 0x2B: //SW

EX\_MEM.ALUOutput = ID\_EX.A + ((ID\_EX.imm & 0x8000) > 0 ? (ID\_EX.imm |0xFFFF0000) : (ID\_EX.imm & 0x0000FFFF));

EX\_MEM.B = ID\_EX.B;

break;

default:

printf("Instruction is not implemented at 0x%x\n", EX\_MEM.IR);

break;

}

}

} //function end

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* instruction decode (ID) pipeline stage: \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void ID() {

if(stall == 0){

ID\_EX.IR = IF\_ID.IR;

uint32\_t instruction = ID\_EX.IR;

uint32\_t op = (instruction & 0xFC000000) >>26; //opcode

uint32\_t bk = instruction & 0x0000003F; //back of 32 bits to id the op code

ID\_EX\_rs = (instruction & 0x03E00000) >> 21; //source

ID\_EX\_rt = (instruction & 0x001F0000) >> 16; //destination

uint32\_t im = instruction & 0x0000FFFF; //immediate / offset change to 32bits for the sign-extend

uint32\_t h = (instruction & 0x0000007C0) >> 6; //for shift instructions) \*\*\*\* know as SA in the BOOK

if((EX\_MEM\_RegWrite && (EX\_MEM\_RegisterRd != 0)) && (EX\_MEM\_RegisterRd == ID\_EX\_rs)) {

if(ENABLE\_FORWARDING == 1) {

forwardA = 0x10;

} else {

stall = 2;

}

}

if((EX\_MEM\_RegWrite && (EX\_MEM\_RegisterRd != 0)) && (EX\_MEM\_RegisterRd == ID\_EX\_rt)) {

if(ENABLE\_FORWARDING == 1) {

forwardB = 0x10;

} else {

stall = 2;

}

}

if((EX\_MEM\_RegWrite && (EX\_MEM\_RegisterRt != 0)) && (EX\_MEM\_RegisterRt == ID\_EX\_rs)) {

if(ENABLE\_FORWARDING == 1) {

forwardA = 0x10;

} else {

stall = 2;

}

}

if((EX\_MEM\_RegWrite && (EX\_MEM\_RegisterRt != 0)) && (EX\_MEM\_RegisterRt == ID\_EX\_rt)) {

if(ENABLE\_FORWARDING == 1) {

forwardB = 0x10;

} else {

stall = 2;

}

}

if((MEM\_WB\_RegWrite && (MEM\_WB\_RegisterRt != 0)) && (MEM\_WB\_RegisterRt == ID\_EX\_rs)) {

if(ENABLE\_FORWARDING == 1) {

forwardA = 0x01;

} else {

stall = 1;

}

}

if((MEM\_WB\_RegWrite && (MEM\_WB\_RegisterRt != 0)) && (MEM\_WB\_RegisterRt == ID\_EX\_rt)) {

if(ENABLE\_FORWARDING == 1) {

forwardB = 0x01;

} else {

stall = 1;

}

}

if((MEM\_WB\_RegWrite && (MEM\_WB\_RegisterRd != 0)) && (MEM\_WB\_RegisterRd == ID\_EX\_rs)) {

if(ENABLE\_FORWARDING == 1) {

forwardA = 0x01;

} else {

stall = 1;

}

}

if((MEM\_WB\_RegWrite && (MEM\_WB\_RegisterRd != 0)) && (MEM\_WB\_RegisterRd == ID\_EX\_rt)) {

if(ENABLE\_FORWARDING == 1) {

forwardB = 0x01;

} else {

stall = 1;

}

}

if(op == 0X00) {

switch(bk) {

case 0x00://SLL last of the bk shouldSIM be 000000

ID\_EX.A = CURRENT\_STATE.REGS[ID\_EX\_rs];

ID\_EX.B = CURRENT\_STATE.REGS[h];

ID\_EX.imm = 0;

break;

case 0x02://SRL last of the bk should be 000010

ID\_EX.A = CURRENT\_STATE.REGS[ID\_EX\_rs];

ID\_EX.B = CURRENT\_STATE.REGS[h];

ID\_EX.imm = 0;

break;

case 0x03://SRA last of the bk should be 000011

ID\_EX.A = CURRENT\_STATE.REGS[ID\_EX\_rs];

ID\_EX.B = CURRENT\_STATE.REGS[h];

ID\_EX.imm = 0;

break;

default: //Reg-Reg

ID\_EX.A = CURRENT\_STATE.REGS[ID\_EX\_rs];

ID\_EX.B = CURRENT\_STATE.REGS[ID\_EX\_rt];

ID\_EX.imm = 0;

break;

}

} else { //immediate

ID\_EX.A = CURRENT\_STATE.REGS[ID\_EX\_rs];

ID\_EX.B = CURRENT\_STATE.REGS[ID\_EX\_rt];

ID\_EX.imm = im;

}

if(stall != 0) {

ID\_EX.IR = 0x00;

}

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* instruction fetch (IF) pipeline stage: \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void IF() {

/\*IMPLEMENT THIS\*/

if(stall == 0) {

IF\_ID.IR = mem\_read\_32(CURRENT\_STATE.PC);

IF\_ID.PC = CURRENT\_STATE.PC + 4; //incrementing to the next address

NEXT\_STATE.PC = IF\_ID.PC; // new instruction

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Initialize Memory \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void initialize() {

init\_memory();

CURRENT\_STATE.PC = MEM\_TEXT\_BEGIN;

NEXT\_STATE = CURRENT\_STATE;

RUN\_FLAG = TRUE;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Print the program loaded into memory (in MIPS assembly format) \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void show\_pipeline(){

printf("\nCurrent PC: %x\n", CURRENT\_STATE.PC);

printf("IF/ID.IR %x\n", IF\_ID.IR);

printf("IF/ID.PC %x\n", IF\_ID.PC);

printf("ID/EX.IR %x\n", ID\_EX.IR);

printf("ID/EX.A %x\n", ID\_EX.A);

printf("ID/EX.B %x\n", ID\_EX.B);

printf("ID/EX.imm %x\n", ID\_EX.imm);

printf("EX/MEM.IR %x\n", EX\_MEM.IR);

printf("EX/MEM.ALUOutput %x\n", EX\_MEM.ALUOutput);

printf("MEM/WB.IR %x\n", MEM\_WB.IR);

printf("MEM/WB.ALUOutput %x\n", MEM\_WB.ALUOutput);

printf("MEM/WB.LMD %x\n\n", MEM\_WB.LMD);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Print the current pipeline \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void print\_program(){

int i;

uint32\_t addr;

for(i=0; i<PROGRAM\_SIZE; i++)

{

addr = MEM\_TEXT\_BEGIN + (i\*4);

printf("[0x%x]\t", addr);

print\_instruction(addr);

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* main \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int main(int argc, char \*argv[]) {

printf("\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

printf("Welcome to MU-MIPS SIM...\n");

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n\n");

if (argc < 2) {

printf("Error: You should provide input file.\nUsage: %s <input program> \n\n", argv[0]);

exit(1);

}

strcpy(prog\_file, argv[1]);

initialize();

load\_program();

help();

while (1){

handle\_command();

}

return 0;

}