Lab5

Are we still supposed to be storing a number at 4000

At cycle 32222 it start putting random values in the register need to check that part

/\*

Name 1: Siddhant Pandit

UTEID 1: shp695

\*/

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

/\* \*/

/\* LC-3b Simulator \*/

/\* \*/

/\* EE 460N - Lab 5 \*/

/\* The University of Texas at Austin \*/

/\* \*/

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

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

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

/\* \*/

/\* Files: ucode Microprogram file \*/

/\* pagetable page table in LC-3b machine language \*/

/\* isaprogram LC-3b machine language program file \*/

/\* \*/

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

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

/\* These are the functions you'll have to write. \*/

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

void eval\_micro\_sequencer();

void cycle\_memory();

void eval\_bus\_drivers();

void drive\_bus();

void latch\_datapath\_values();

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

/\* A couple of useful definitions. \*/

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

#define FALSE 0

#define TRUE 1

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

/\* Use this to avoid overflowing 16 bits on the bus. \*/

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

#define Low16bits(x) ((x) & 0xFFFF)

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

/\* Definition of the control store layout. \*/

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

#define CONTROL\_STORE\_ROWS 64

#define INITIAL\_STATE\_NUMBER 18

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

/\* Definition of bit order in control store word. \*/

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

enum CS\_BITS {

IRD1, IRD0,

COND2,COND1, COND0,

J5, J4, J3, J2, J1, J0,

LD\_PTE,

LD\_VA,

LD\_PTBR,

LD\_SSP,

LD\_USP,

LD\_PSR,

LD\_VECTOR,

LD\_EXEC,

LD\_MAR,

LD\_MDR,

LD\_IR,

LD\_BEN,

LD\_REG,

LD\_CC,

LD\_PC,

GATE\_PTBR,

GATE\_ADDER\_VA,

GATE\_VA,

GATE\_ADDER\_PTE,

GATE\_PTE,

GATE\_SP,

GATE\_PSR,

GATE\_TABLE,

GATE\_PC,

GATE\_MDR,

GATE\_ALU,

GATE\_MARMUX,

GATE\_SHF,

PCMUX1, PCMUX0,

SPMUX1,SPMUX0,

PSRMUX,

DRMUX1,DRMUX0,

SR1MUX1,SR1MUX0,

PTEMUX,

TRAMUX,

ADDR1MUX,

ADDR2MUX1, ADDR2MUX0,

MARMUX,

ALUK1, ALUK0,

RS,

MIO\_EN,

R\_W,

DATA\_SIZE,

SAVEMDR,

LOADMDR,

LSHF1,

/\* MODIFY: you have to add all your new control signals \*/

CONTROL\_STORE\_BITS

} CS\_BITS;

enum excep {

filler,

interruptLC,

pageFault,

unaligned,

protect,

opcode

}excep;

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

/\* Functions to get at the control bits. \*/

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

int GetIRD(int \*x) { return(x[IRD1] <<1)+ (x[IRD0]); }

int GetCOND(int \*x) { return(x[COND2] << 2)+(x[COND1] << 1) +

(x[COND0]); }

int GetJ(int \*x) { return((x[J5] << 5) + (x[J4] << 4) +

(x[J3] << 3) + (x[J2] << 2) +

(x[J1] << 1) + x[J0]); }

int GetLD\_PTE(int \*x) { return(x[LD\_PTE]); }

int GetLD\_VA(int \*x) { return(x[LD\_VA]); }

int GetLD\_PTBR(int \*x) { return(x[LD\_PTBR]); }

int GetLD\_SSP(int \*x) { return(x[LD\_SSP]); }

int GetLD\_USP(int \*x) { return(x[LD\_USP]); }

int GetLD\_PSR(int \*x) { return(x[LD\_PSR]); }

int GetLD\_VECTOR(int \*x) { return(x[LD\_VECTOR]); }

int GetLD\_EXEC(int \*x) { return(x[LD\_EXEC]); }

int GetLD\_MAR(int \*x) { return(x[LD\_MAR]); }

int GetLD\_MDR(int \*x) { return(x[LD\_MDR]); }

int GetLD\_IR(int \*x) { return(x[LD\_IR]); }

int GetLD\_BEN(int \*x) { return(x[LD\_BEN]); }

int GetLD\_REG(int \*x) { return(x[LD\_REG]); }

int GetLD\_CC(int \*x) { return(x[LD\_CC]); }

int GetLD\_PC(int \*x) { return(x[LD\_PC]); }

int GetGATE\_PTBR(int \*x) { return(x[GATE\_PTBR]); }

int GetGATE\_ADDER\_VA(int \*x) { return(x[GATE\_ADDER\_VA]); }

int GetGATE\_VA(int \*x) { return(x[GATE\_VA]); }

int GetGATE\_ADDER\_PTE(int \*x){ return(x[GATE\_ADDER\_PTE]); }

int GetGATE\_PTE(int \*x) { return(x[GATE\_PTE]); }

int GetGATE\_SP(int \*x) { return(x[GATE\_SP]); }

int GetGATE\_PSR(int \*x) { return(x[GATE\_PSR]); }

int GetGATE\_TABLE(int \*x) { return(x[GATE\_TABLE]); }

int GetGATE\_PC(int \*x) { return(x[GATE\_PC]); }

int GetGATE\_MDR(int \*x) { return(x[GATE\_MDR]); }

int GetGATE\_ALU(int \*x) { return(x[GATE\_ALU]); }

int GetGATE\_MARMUX(int \*x) { return(x[GATE\_MARMUX]); }

int GetGATE\_SHF(int \*x) { return(x[GATE\_SHF]); }

int GetPCMUX(int \*x) { return((x[PCMUX1] << 1) + x[PCMUX0]); }

int GetSPMUX(int \*x) { return((x[SPMUX1] << 1) + x[SPMUX0]); }

int GetPSRMUX(int \*x) { return(x[PSRMUX]); }

int GetDRMUX(int \*x) { return((x[DRMUX1] << 1) + x[DRMUX0]); }

int GetSR1MUX(int \*x) { return((x[SR1MUX1] << 1) + x[SR1MUX0]); }

int GetPTEMUX(int \*x) { return(x[PTEMUX]); }

int GetTRAMUX(int \*x) { return(x[TRAMUX]); }

int GetADDR1MUX(int \*x) { return(x[ADDR1MUX]); }

int GetADDR2MUX(int \*x) { return((x[ADDR2MUX1] << 1) + x[ADDR2MUX0]); }

int GetMARMUX(int \*x) { return(x[MARMUX]); }

int GetALUK(int \*x) { return((x[ALUK1] << 1) + x[ALUK0]); }

int GetRS(int \*x) { return(x[RS]); }

int GetMIO\_EN(int \*x) { return(x[MIO\_EN]); }

int GetR\_W(int \*x) { return(x[R\_W]); }

int GetDATA\_SIZE(int \*x) { return(x[DATA\_SIZE]); }

int GetSAVEMDR(int \*x) { return(x[SAVEMDR]); }

int GetLOADMDR(int \*x) { return(x[LOADMDR]); }

int GetLSHF1(int \*x) { return(x[LSHF1]); }

/\* MODIFY: you can add more Get functions for your new control signals \*/

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

/\* The control store rom. \*/

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

int CONTROL\_STORE[CONTROL\_STORE\_ROWS][CONTROL\_STORE\_BITS];

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

/\* Main memory. \*/

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

/\* MEMORY[A][0] stores the least significant byte of word at word address A

MEMORY[A][1] stores the most significant byte of word at word address A

There are two write enable signals, one for each byte. WE0 is used for

the least significant byte of a word. WE1 is used for the most significant

byte of a word. \*/

#define WORDS\_IN\_MEM 0x2000 /\* 32 frames \*/

#define MEM\_CYCLES 5

int MEMORY[WORDS\_IN\_MEM][2];

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

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

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

/\* LC-3b State info. \*/

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

#define LC\_3b\_REGS 8

int RUN\_BIT; /\* run bit \*/

int BUS; /\* value of the bus \*/

typedef struct System\_Latches\_Struct{

int PC, /\* program counter \*/

MDR, /\* memory data register \*/

MAR, /\* memory address register \*/

IR, /\* instruction register \*/

N, /\* n condition bit \*/

Z, /\* z condition bit \*/

P, /\* p condition bit \*/

BEN; /\* ben register \*/

int READY; /\* ready bit \*/

/\* The ready bit is also latched as you dont want the memory system to assert it

at a bad point in the cycle\*/

int REGS[LC\_3b\_REGS]; /\* register file. \*/

int MICROINSTRUCTION[CONTROL\_STORE\_BITS]; /\* The microintruction \*/

int STATE\_NUMBER; /\* Current State Number - Provided for debugging \*/

/\* For lab 4 \*/

int VECT;

int USP;

int SSP; /\* Initial value of system stack pointer \*/

/\* MODIFY: you should add here any other registers you need to implement interrupts and exceptions \*/

int PSR;

/\* For lab 5 \*/

int PTBR; /\* This is initialized when we load the page table \*/

int VA; /\* Temporary VA register \*/

/\* MODIFY: you should add here any other registers you need to implement virtual memory \*/

int PTE; // for holding PTE

} System\_Latches;

/\* Data Structure for Latch \*/

System\_Latches CURRENT\_LATCHES, NEXT\_LATCHES;

/\* For lab 5 \*/

#define PAGE\_NUM\_BITS 9

#define PTE\_PFN\_MASK 0x3E00

#define PTE\_VALID\_MASK 0x0004

#define PAGE\_OFFSET\_MASK 0x1FF

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

/\* A cycle counter. \*/

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

int CYCLE\_COUNT;

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

/\* \*/

/\* Procedure : help \*/

/\* \*/

/\* Purpose : Print out a list of commands. \*/

/\* \*/

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

void help() {

printf("----------------LC-3bSIM Help-------------------------\n");

printf("go - run program to completion \n");

printf("run n - execute program for n cycles \n");

printf("mdump low high - dump memory from low to high \n");

printf("rdump - dump the register & bus values \n");

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

printf("quit - exit the program \n\n");

}

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

/\* \*/

/\* Procedure : cycle \*/

/\* \*/

/\* Purpose : Execute a cycle \*/

/\* \*/

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

void cycle() {

eval\_micro\_sequencer();

cycle\_memory();

eval\_bus\_drivers();

drive\_bus();

latch\_datapath\_values();

CURRENT\_LATCHES = NEXT\_LATCHES;

CYCLE\_COUNT++;

}

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

/\* \*/

/\* Procedure : run n \*/

/\* \*/

/\* Purpose : Simulate the LC-3b for n cycles. \*/

/\* \*/

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

void run(int num\_cycles) {

int i;

if (RUN\_BIT == FALSE) {

printf("Can't simulate, Simulator is halted\n\n");

return;

}

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

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

if (CURRENT\_LATCHES.PC == 0x0000) {

RUN\_BIT = FALSE;

printf("Simulator halted\n\n");

break;

}

cycle();

}

}

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

/\* \*/

/\* Procedure : go \*/

/\* \*/

/\* Purpose : Simulate the LC-3b until HALTed. \*/

/\* \*/

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

void go() {

if (RUN\_BIT == FALSE) {

printf("Can't simulate, Simulator is halted\n\n");

return;

}

printf("Simulating...\n\n");

while (CURRENT\_LATCHES.PC != 0x0000)

cycle();

RUN\_BIT = FALSE;

printf("Simulator halted\n\n");

}

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

/\* \*/

/\* Procedure : mdump \*/

/\* \*/

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

/\* output file. \*/

/\* \*/

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

void mdump(FILE \* dumpsim\_file, int start, int stop) {

int address; /\* this is a byte address \*/

printf("\nMemory content [0x%0.4x..0x%0.4x] :\n", start, stop);

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

for (address = (start >> 1); address <= (stop >> 1); address++)

printf(" 0x%0.4x (%d) : 0x%0.2x%0.2x\n", address << 1, address << 1, MEMORY[address][1], MEMORY[address][0]);

printf("\n");

/\* dump the memory contents into the dumpsim file \*/

fprintf(dumpsim\_file, "\nMemory content [0x%0.4x..0x%0.4x] :\n", start, stop);

fprintf(dumpsim\_file, "-------------------------------------\n");

for (address = (start >> 1); address <= (stop >> 1); address++)

fprintf(dumpsim\_file, " 0x%0.4x (%d) : 0x%0.2x%0.2x\n", address << 1, address << 1, MEMORY[address][1], MEMORY[address][0]);

fprintf(dumpsim\_file, "\n");

fflush(dumpsim\_file);

}

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

/\* \*/

/\* Procedure : rdump \*/

/\* \*/

/\* Purpose : Dump current register and bus values to the \*/

/\* output file. \*/

/\* \*/

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

void rdump(FILE \* dumpsim\_file) {

int k;

printf("\nCurrent register/bus values :\n");

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

printf("Cycle Count : %d\n", CYCLE\_COUNT);

printf("PC : 0x%0.4x\n", CURRENT\_LATCHES.PC);

printf("IR : 0x%0.4x\n", CURRENT\_LATCHES.IR);

printf("STATE\_NUMBER : 0x%0.4x\n\n", CURRENT\_LATCHES.STATE\_NUMBER);

printf("BUS : 0x%0.4x\n", BUS);

printf("MDR : 0x%0.4x\n", CURRENT\_LATCHES.MDR);

printf("MAR : 0x%0.4x\n", CURRENT\_LATCHES.MAR);

printf("CCs: N = %d Z = %d P = %d\n", CURRENT\_LATCHES.N, CURRENT\_LATCHES.Z, CURRENT\_LATCHES.P);

printf("PSR : 0x%0.4x\n", CURRENT\_LATCHES.PSR);

printf("Registers:\n");

for (k = 0; k < LC\_3b\_REGS; k++)

printf("%d: 0x%0.4x\n", k, CURRENT\_LATCHES.REGS[k]);

printf("\n");

/\* dump the state information into the dumpsim file \*/

fprintf(dumpsim\_file, "\nCurrent register/bus values :\n");

fprintf(dumpsim\_file, "-------------------------------------\n");

fprintf(dumpsim\_file, "Cycle Count : %d\n", CYCLE\_COUNT);

fprintf(dumpsim\_file, "PC : 0x%0.4x\n", CURRENT\_LATCHES.PC);

fprintf(dumpsim\_file, "IR : 0x%0.4x\n", CURRENT\_LATCHES.IR);

fprintf(dumpsim\_file, "STATE\_NUMBER : 0x%0.4x\n\n", CURRENT\_LATCHES.STATE\_NUMBER);

fprintf(dumpsim\_file, "BUS : 0x%0.4x\n", BUS);

fprintf(dumpsim\_file, "MDR : 0x%0.4x\n", CURRENT\_LATCHES.MDR);

fprintf(dumpsim\_file, "MAR : 0x%0.4x\n", CURRENT\_LATCHES.MAR);

fprintf(dumpsim\_file, "CCs: N = %d Z = %d P = %d\n", CURRENT\_LATCHES.N, CURRENT\_LATCHES.Z, CURRENT\_LATCHES.P);

fprintf(dumpsim\_file, "Registers:\n");

for (k = 0; k < LC\_3b\_REGS; k++)

fprintf(dumpsim\_file, "%d: 0x%0.4x\n", k, CURRENT\_LATCHES.REGS[k]);

fprintf(dumpsim\_file, "\n");

fflush(dumpsim\_file);

}

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

/\* \*/

/\* Procedure : get\_command \*/

/\* \*/

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

/\* \*/

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

void get\_command(FILE \* dumpsim\_file) {

char buffer[20];

int start, stop, cycles;

printf("LC-3b-SIM> ");

scanf("%s", buffer);

printf("\n");

switch(buffer[0]) {

case 'G':

case 'g':

go();

break;

case 'M':

case 'm':

scanf("%i %i", &start, &stop);

mdump(dumpsim\_file, start, stop);

break;

case '?':

help();

break;

case 'Q':

case 'q':

printf("Bye.\n");

exit(0);

case 'R':

case 'r':

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

rdump(dumpsim\_file);

else {

scanf("%d", &cycles);

run(cycles);

}

break;

default:

printf("Invalid Command\n");

break;

}

}

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

/\* \*/

/\* Procedure : init\_control\_store \*/

/\* \*/

/\* Purpose : Load microprogram into control store ROM \*/

/\* \*/

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

void init\_control\_store(char \*ucode\_filename) {

FILE \*ucode;

int i, j, index;

char line[200];

printf("Loading Control Store from file: %s\n", ucode\_filename);

/\* Open the micro-code file. \*/

if ((ucode = fopen(ucode\_filename, "r")) == NULL) {

printf("Error: Can't open micro-code file %s\n", ucode\_filename);

exit(-1);

}

/\* Read a line for each row in the control store. \*/

for(i = 0; i < CONTROL\_STORE\_ROWS; i++) {

if (fscanf(ucode, "%[^\n]\n", line) == EOF) {

printf("Error: Too few lines (%d) in micro-code file: %s\n",

i, ucode\_filename);

exit(-1);

}

/\* Put in bits one at a time. \*/

index = 0;

for (j = 0; j < CONTROL\_STORE\_BITS; j++) {

/\* Needs to find enough bits in line. \*/

if (line[index] == '\0') {

printf("Error: Too few control bits in micro-code file: %s\nLine: %d\n",

ucode\_filename, i);

exit(-1);

}

if (line[index] != '0' && line[index] != '1') {

printf("Error: Unknown value in micro-code file: %s\nLine: %d, Bit: %d\n",

ucode\_filename, i, j);

exit(-1);

}

/\* Set the bit in the Control Store. \*/

CONTROL\_STORE[i][j] = (line[index] == '0') ? 0:1;

index++;

}

/\* Warn about extra bits in line. \*/

if (line[index] != '\0')

printf("Warning: Extra bit(s) in control store file %s. Line: %d\n",

ucode\_filename, i);

}

printf("\n");

}

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

/\* \*/

/\* Procedure : init\_memory \*/

/\* \*/

/\* Purpose : Zero out the memory array \*/

/\* \*/

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

void init\_memory() {

int i;

for (i=0; i < WORDS\_IN\_MEM; i++) {

MEMORY[i][0] = 0;

MEMORY[i][1] = 0;

}

}

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

/\* \*/

/\* Procedure : load\_program \*/

/\* \*/

/\* Purpose : Load program and service routines into mem. \*/

/\* \*/

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

void load\_program(char \*program\_filename, int is\_virtual\_base) {

FILE \* prog;

int ii, word, program\_base, pte, virtual\_pc;

/\* Open program file. \*/

prog = fopen(program\_filename, "r");

if (prog == NULL) {

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

exit(-1);

}

/\* Read in the program. \*/

if (fscanf(prog, "%x\n", &word) != EOF)

program\_base = word >> 1;

else {

printf("Error: Program file is empty\n");

exit(-1);

}

if (is\_virtual\_base) {

if (CURRENT\_LATCHES.PTBR == 0) {

printf("Error: Page table base not loaded %s\n", program\_filename);

exit(-1);

}

/\* convert virtual\_base to physical\_base \*/

virtual\_pc = program\_base << 1;

pte = (MEMORY[(CURRENT\_LATCHES.PTBR + (((program\_base << 1) >> PAGE\_NUM\_BITS) << 1)) >> 1][1] << 8) |

MEMORY[(CURRENT\_LATCHES.PTBR + (((program\_base << 1) >> PAGE\_NUM\_BITS) << 1)) >> 1][0];

printf("virtual base of program: %04x\npte: %04x\n", program\_base << 1, pte);

if ((pte & PTE\_VALID\_MASK) == PTE\_VALID\_MASK) {

program\_base = (pte & PTE\_PFN\_MASK) | ((program\_base << 1) & PAGE\_OFFSET\_MASK);

printf("physical base of program: %x\n\n", program\_base);

program\_base = program\_base >> 1;

} else {

printf("attempting to load a program into an invalid (non-resident) page\n\n");

exit(-1);

}

}

else {

/\* is page table \*/

CURRENT\_LATCHES.PTBR = program\_base << 1;

}

ii = 0;

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

/\* Make sure it fits. \*/

if (program\_base + ii >= WORDS\_IN\_MEM) {

printf("Error: Program file %s is too long to fit in memory. %x\n",

program\_filename, ii);

exit(-1);

}

/\* Write the word to memory array. \*/

MEMORY[program\_base + ii][0] = word & 0x00FF;

MEMORY[program\_base + ii][1] = (word >> 8) & 0x00FF;;

ii++;

}

if (CURRENT\_LATCHES.PC == 0 && is\_virtual\_base)

CURRENT\_LATCHES.PC = virtual\_pc;

printf("Read %d words from program into memory.\n\n", ii);

}

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

/\* \*/

/\* Procedure : initialize \*/

/\* \*/

/\* Purpose : Load microprogram and machine language program \*/

/\* and set up initial state of the machine \*/

/\* \*/

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

void initialize(char \*argv[], int num\_prog\_files) {

int i;

init\_control\_store(argv[1]);

init\_memory();

load\_program(argv[2],0);

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

load\_program(argv[i + 3],1);

}

CURRENT\_LATCHES.Z = 1;

CURRENT\_LATCHES.STATE\_NUMBER = INITIAL\_STATE\_NUMBER;

memcpy(CURRENT\_LATCHES.MICROINSTRUCTION, CONTROL\_STORE[INITIAL\_STATE\_NUMBER], sizeof(int)\*CONTROL\_STORE\_BITS);

CURRENT\_LATCHES.SSP = 0x3000; /\* Initial value of system stack pointer \*/

/\* MODIFY: you can add more initialization code HERE \*/

NEXT\_LATCHES = CURRENT\_LATCHES;

RUN\_BIT = TRUE;

}

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

/\* \*/

/\* Procedure : main \*/

/\* \*/

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

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

FILE \* dumpsim\_file;

/\* Error Checking \*/

if (argc < 4) {

printf("Error: usage: %s <micro\_code\_file> <page table file> <program\_file\_1> <program\_file\_2> ...\n",

argv[0]);

exit(1);

}

printf("LC-3b Simulator\n\n");

initialize(argv, argc - 3);

if ( (dumpsim\_file = fopen( "dumpsim", "w" )) == NULL ) {

printf("Error: Can't open dumpsim file\n");

exit(-1);

}

while (1)

get\_command(dumpsim\_file);

}

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

/\* Do not modify the above code, except for the places indicated

with a "MODIFY:" comment.

You are allowed to use the following global variables in your

code. These are defined above.

CONTROL\_STORE

MEMORY

BUS

CURRENT\_LATCHES

NEXT\_LATCHES

You may define your own local/global variables and functions.

You may use the functions to get at the control bits defined

above.

Begin your code here \*/

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

#define INSTR(pos, wid,instr) ((instr) >> (pos) & ~(~0<<(wid)))

#define Low8bits(x) ((x) & 0x00FF)

#define High8bits(x) ((x) & 0xFF00)

#define SIGN\_EXT(num,wid) ((num) >> (wid) - 1? ~0 <<(wid) | (num) :(num))

#define ZERO\_EXTEND(num,pos,wid) ((num) >> (pos) & ~(~0 << (wid)))

#define Table 0x0200

int inter\_request = 0;

int value\_gateSP=0;

int value\_gatePSR=0;

int value\_gateTable=0;

int first\_cycle = 0;

int mem\_cycle=0;

int value\_gatePC=0;

int value\_gateMDR=0;

int value\_gateALU=0;

int value\_gateMARMUX=0;

int value\_gateSHF=0;

int main\_addr\_output=0;

int value\_gatePCFinal=0;

int return\_state=0;

int exception\_occur=0;

int value\_gatePTBR=0;

int value\_gateADDER\_VA=0;

int value\_gateVA=0;

int value\_gateADDER\_PTE=0;

int value\_gatePTE=0;

int value\_muxPTE=0;

int protect\_mdr=0;

void eval\_micro\_sequencer() {

/\*

\* Evaluate the address of the next state according to the

\* micro sequencer logic. Latch the next microinstruction.

\*/

if(first\_cycle==0){

CURRENT\_LATCHES.PSR = 0x8000;

first\_cycle=1;

}

if(CYCLE\_COUNT==300){

inter\_request=1;

}

if(GetRS(CURRENT\_LATCHES.MICROINSTRUCTION)==1){

return\_state = GetJ(CURRENT\_LATCHES.MICROINSTRUCTION);

}

int state = CURRENT\_LATCHES.STATE\_NUMBER;

int IRD = GetIRD(CURRENT\_LATCHES.MICROINSTRUCTION);

int j = GetJ(CURRENT\_LATCHES.MICROINSTRUCTION);

int cond = GetCOND(CURRENT\_LATCHES.MICROINSTRUCTION);

int add\_mode = INSTR(11,1,CURRENT\_LATCHES.IR);

int next\_instr[CONTROL\_STORE\_BITS];

// printf("\nBEGINNING OF NEW STATE\n\nSTATE NUMBER RIGHT NOW: %d\n\n",CURRENT\_LATCHES.STATE\_NUMBER);

// printf("\nCYCLE COUNT: %d\n\n",CYCLE\_COUNT);

if(IRD==1){

int next\_state = INSTR(12,4,CURRENT\_LATCHES.IR);

for(int i=0;i<CONTROL\_STORE\_BITS;i++){

next\_instr[i] = CONTROL\_STORE[next\_state][i];

}

memcpy(NEXT\_LATCHES.MICROINSTRUCTION,next\_instr,sizeof(int)\*CONTROL\_STORE\_BITS);

NEXT\_LATCHES.STATE\_NUMBER = next\_state;

return;

}

else if(IRD==0){

if(cond==0){

for(int i=0;i<CONTROL\_STORE\_BITS;i++){

next\_instr[i] = CONTROL\_STORE[j][i];

}

memcpy(NEXT\_LATCHES.MICROINSTRUCTION,next\_instr,sizeof(int)\*CONTROL\_STORE\_BITS);

NEXT\_LATCHES.STATE\_NUMBER = j;

return;

}

if(cond==1){

if(CURRENT\_LATCHES.READY==1){

j+=2;

NEXT\_LATCHES.READY=0;

}

for(int i=0;i<CONTROL\_STORE\_BITS;i++){

next\_instr[i] = CONTROL\_STORE[j][i];

}

memcpy(NEXT\_LATCHES.MICROINSTRUCTION,next\_instr,sizeof(int)\*CONTROL\_STORE\_BITS);

NEXT\_LATCHES.STATE\_NUMBER = j;

return;

}

if(cond==2){

if(CURRENT\_LATCHES.BEN==1){

j+=4;

}

for(int i=0;i<CONTROL\_STORE\_BITS;i++){

next\_instr[i] = CONTROL\_STORE[j][i];

}

memcpy(NEXT\_LATCHES.MICROINSTRUCTION,next\_instr,sizeof(int)\*CONTROL\_STORE\_BITS);

NEXT\_LATCHES.STATE\_NUMBER = j;

return;

}

if(cond==3){

if(add\_mode==1){

j+=1;

}

for(int i=0;i<CONTROL\_STORE\_BITS;i++){

next\_instr[i] = CONTROL\_STORE[j][i];

}

memcpy(NEXT\_LATCHES.MICROINSTRUCTION,next\_instr,sizeof(int)\*CONTROL\_STORE\_BITS);

NEXT\_LATCHES.STATE\_NUMBER = j;

return;

}

if(cond==4){

int jsr15 = INSTR(15,1,CURRENT\_LATCHES.PSR);

if(jsr15==1){

j+=8;

}

for(int i=0;i<CONTROL\_STORE\_BITS;i++){

next\_instr[i] = CONTROL\_STORE[j][i];

}

memcpy(NEXT\_LATCHES.MICROINSTRUCTION,next\_instr,sizeof(int)\*CONTROL\_STORE\_BITS);

NEXT\_LATCHES.STATE\_NUMBER = j;

return;

}

}

}

void cycle\_memory() {

/\*

\* This function emulates memory and the WE logic.

\* Keep track of which cycle of MEMEN we are dealing with.

\* If fourth, we need to latch Ready bit at the end of

\* cycle to prepare microsequencer for the fifth cycle.

\*/

int mem\_use = GetMIO\_EN(CURRENT\_LATCHES.MICROINSTRUCTION);

if(mem\_use==1){

mem\_cycle++;

}

else{

return;

}

}

void eval\_bus\_drivers() {

/\*

\* Datapath routine emulating operations before driving the bus.

\* Evaluate the input of tristate drivers

\* Gate\_MARMUX,

\* Gate\_PC,

\* Gate\_ALU,

\* Gate\_SHF,

\* Gate\_MDR.

\*/

int spMux = GetSPMUX(CURRENT\_LATCHES.MICROINSTRUCTION);

int psrMux = GetPSRMUX(CURRENT\_LATCHES.MICROINSTRUCTION);

int psrMux\_output = 0;

int data\_size = GetDATA\_SIZE(CURRENT\_LATCHES.MICROINSTRUCTION);

int updated\_mar = CURRENT\_LATCHES.MAR >> 1;

int gateMDRStatus = GetGATE\_MDR(CURRENT\_LATCHES.MICROINSTRUCTION);

if(gateMDRStatus==1){

if(data\_size==0){

int mar\_zero = INSTR(0,1,CURRENT\_LATCHES.MAR);

if(mar\_zero==0){

value\_gateMDR=Low16bits(SIGN\_EXT((MEMORY[updated\_mar][0]),8));

}

if(mar\_zero==1){

value\_gateMDR = Low16bits(SIGN\_EXT((MEMORY[updated\_mar][1]),8));

}

}

if(data\_size==1){

value\_gateMDR = CURRENT\_LATCHES.MDR;

}

}

int dr\_mux = GetDRMUX(CURRENT\_LATCHES.MICROINSTRUCTION);

int immm5 = INSTR(0,5,CURRENT\_LATCHES.IR);

int am4 = INSTR(0,4,CURRENT\_LATCHES.IR);

int off6 = INSTR(0,6,CURRENT\_LATCHES.IR);

int off9 = INSTR(0,9,CURRENT\_LATCHES.IR);

int off11 = INSTR(0,11,CURRENT\_LATCHES.IR);

int sr2mux\_output;

int bit5= INSTR(5,1,CURRENT\_LATCHES.IR);

if(bit5==0){

sr2mux\_output = CURRENT\_LATCHES.REGS[INSTR(0,3,CURRENT\_LATCHES.IR)];

}

else if(bit5==1){

sr2mux\_output = SIGN\_EXT(immm5,5);

}

int sr1mux = GetSR1MUX(CURRENT\_LATCHES.MICROINSTRUCTION);

int addr1mux = GetADDR1MUX(CURRENT\_LATCHES.MICROINSTRUCTION);

int addr2mux = GetADDR2MUX(CURRENT\_LATCHES.MICROINSTRUCTION);

int marmux = GetMARMUX(CURRENT\_LATCHES.MICROINSTRUCTION);

int aluk = GetALUK(CURRENT\_LATCHES.MICROINSTRUCTION);

int pcmux = GetPCMUX(CURRENT\_LATCHES.MICROINSTRUCTION);

int lshf = GetLSHF1(CURRENT\_LATCHES.MICROINSTRUCTION);

int sr1mux\_output;

int dr1mux\_output;

int addr1mux\_output;

int addr2mux\_output;

if(sr1mux==0){

int reg\_num = INSTR(9,3,CURRENT\_LATCHES.IR);

sr1mux\_output = CURRENT\_LATCHES.REGS[reg\_num];

}

else if(sr1mux==1){

int reg\_num = INSTR(6,3,CURRENT\_LATCHES.IR);

sr1mux\_output = CURRENT\_LATCHES.REGS[reg\_num];

}

else if(sr1mux==2){

sr1mux\_output = CURRENT\_LATCHES.REGS[6];

}

if(dr\_mux==0){

int reg\_num = INSTR(9,3,CURRENT\_LATCHES.IR);

dr1mux\_output = CURRENT\_LATCHES.REGS[reg\_num];

}

else if(dr\_mux==1){

dr1mux\_output = CURRENT\_LATCHES.REGS[7];

}

else if(dr\_mux==2){

dr1mux\_output = CURRENT\_LATCHES.REGS[6];

}

if(addr1mux==0){

addr1mux\_output= CURRENT\_LATCHES.PC;

}

else if(addr1mux==1){

addr1mux\_output = sr1mux\_output;

}

//sp mux is after this

if(spMux==0){

value\_gateSP= CURRENT\_LATCHES.SSP;

}

if(spMux==1){

value\_gateSP= sr1mux\_output-2;

}

if(spMux==2){

value\_gateSP= sr1mux\_output+2;

}

if(spMux==3){

value\_gateSP= CURRENT\_LATCHES.USP;

}

if(addr2mux==0){

addr2mux\_output=0;

}

if(addr2mux==1){

addr2mux\_output= SIGN\_EXT(off6,6);

}

if(addr2mux==2){

addr2mux\_output= SIGN\_EXT(off9,9);

}

if(addr2mux==3){

addr2mux\_output= SIGN\_EXT(off11,11);

}

if(lshf==1){

addr2mux\_output = addr2mux\_output<<1;

}

main\_addr\_output= addr2mux\_output+addr1mux\_output;

if(aluk==0){

value\_gateALU = sr2mux\_output+sr1mux\_output;

}

else if(aluk==1){

value\_gateALU = sr2mux\_output & sr1mux\_output;

}

else if(aluk==2){

value\_gateALU = sr2mux\_output ^ sr1mux\_output;

}

else if(aluk==3){

if(data\_size==1){

value\_gateALU = sr1mux\_output;

}

else if(data\_size==0){

value\_gateALU = (Low8bits(sr1mux\_output)<<8)+(Low8bits(sr1mux\_output));

}

}

int shift= INSTR(4, 2,CURRENT\_LATCHES.IR);

if(shift==0){

value\_gateSHF= sr1mux\_output<<am4;

}

else if(shift==1){

value\_gateSHF = sr1mux\_output>>am4;

}

else if(shift==3){

value\_gateSHF = (SIGN\_EXT(sr1mux\_output, 16) >> am4);

}

if(marmux==0){

int ir7 = INSTR(0,8,CURRENT\_LATCHES.IR);

value\_gateMARMUX= (ZERO\_EXTEND(ir7,0,8))<<1;

}

else if(marmux==1){

value\_gateMARMUX = main\_addr\_output;

}

value\_gatePCFinal=CURRENT\_LATCHES.PC;

if(pcmux==0){

value\_gatePC = CURRENT\_LATCHES.PC;

}

else if(pcmux==1){

value\_gatePC = BUS;

}

else if(pcmux==2){

value\_gatePC = main\_addr\_output;

}

else if(pcmux==3){

value\_gatePCFinal=CURRENT\_LATCHES.PC-2;

}

value\_gatePSR = CURRENT\_LATCHES.PSR;

value\_gateTable = CURRENT\_LATCHES.VECT + Table;

//for lab 5

int ptemux = GetPTEMUX(CURRENT\_LATCHES.MICROINSTRUCTION);

int tramux = GetTRAMUX(CURRENT\_LATCHES.MICROINSTRUCTION);

value\_gatePTBR = CURRENT\_LATCHES.PTBR;

value\_gateVA = CURRENT\_LATCHES.VA;

int temp[CONTROL\_STORE\_BITS];

for(int i=0;i<CONTROL\_STORE\_BITS;i++){

temp[i] = CONTROL\_STORE[return\_state][i];

}

int rw = GetR\_W(temp);

rw=rw<<1;

rw+=1;

if(ptemux==1){

value\_muxPTE = CURRENT\_LATCHES.PTE | rw;

}

value\_gatePTE = value\_muxPTE;

value\_gateADDER\_PTE = (((INSTR(9,5,CURRENT\_LATCHES.PTE))<<9) +(INSTR(0,9,CURRENT\_LATCHES.VA))) & 0x3FFF;

// printf("\nCHECKING VALUE OF VA: %X\n",((INSTR(9,7,CURRENT\_LATCHES.VA))<<1));

//printf("\nCHECKING VALUE OF ptbr: %X\n",(INSTR(8,8,CURRENT\_LATCHES.PTBR))<<8);

value\_gateADDER\_VA = (((INSTR(9,7,CURRENT\_LATCHES.VA))<<1)+ (INSTR(8,8,CURRENT\_LATCHES.PTBR)<<8));

}

void drive\_bus() {

/\*

\* Datapath routine for driving the bus from one of the 5 possible

\* tristate drivers.

\*/

int GateTable = GetGATE\_TABLE(CURRENT\_LATCHES.MICROINSTRUCTION);

int GatePSR = GetGATE\_PSR(CURRENT\_LATCHES.MICROINSTRUCTION);

int GateSP = GetGATE\_SP(CURRENT\_LATCHES.MICROINSTRUCTION);

int Gatepc = GetGATE\_PC(CURRENT\_LATCHES.MICROINSTRUCTION);

int Gatemdr = GetGATE\_MDR(CURRENT\_LATCHES.MICROINSTRUCTION);

int Gatealu = GetGATE\_ALU(CURRENT\_LATCHES.MICROINSTRUCTION);

int GateMarmux = GetGATE\_MARMUX(CURRENT\_LATCHES.MICROINSTRUCTION);

int Gateshf = GetGATE\_SHF(CURRENT\_LATCHES.MICROINSTRUCTION);

int Gateptbr = GetGATE\_PTBR(CURRENT\_LATCHES.MICROINSTRUCTION);

int GateAdderva = GetGATE\_ADDER\_VA(CURRENT\_LATCHES.MICROINSTRUCTION);

int Gateva = GetGATE\_VA(CURRENT\_LATCHES.MICROINSTRUCTION);

int GateAdderpte = GetGATE\_ADDER\_PTE(CURRENT\_LATCHES.MICROINSTRUCTION);

int Gatepte = GetGATE\_PTE(CURRENT\_LATCHES.MICROINSTRUCTION);

if(Gatepc || Gatemdr || Gatealu || GateMarmux || Gateshf || GateSP || GatePSR || GateTable || Gateptbr || GateAdderva || Gateva || GateAdderpte || Gatepte) {

if(Gateptbr){

BUS= Low16bits(value\_gatePTBR);

}

if(GateAdderva){

BUS= Low16bits(value\_gateADDER\_VA);

}

if(Gateva){

BUS= Low16bits(value\_gateVA);

}

if(GateAdderpte){

BUS= Low16bits(value\_gateADDER\_PTE);

}

if(Gatepte){

BUS= Low16bits(value\_gatePTE);

}

if(GateTable){

BUS = Low16bits(value\_gateTable);

}

if(GateSP) {

BUS = Low16bits(value\_gateSP);

}

if(GatePSR) {

BUS = Low16bits(value\_gatePSR);

}

if (Gatepc) {

BUS = Low16bits(value\_gatePCFinal);

}

if (Gatemdr) {

BUS = Low16bits(value\_gateMDR);

}

if (Gatealu) {

BUS = Low16bits(value\_gateALU);

}

if (GateMarmux) {

BUS = Low16bits(value\_gateMARMUX);

}

if (Gateshf) {

BUS = Low16bits(value\_gateSHF);

}

}

else{

BUS= Low16bits(0);

}

// printf("Gate Table: %X\n Gate ptbr: %x\n Gate adderva: %x\n Gate va: %x\n Gate adderpte: %x\n Gate pte: %x\n",value\_gatePTBR,value\_gateADDER\_VA,value\_gateVA,value\_gateADDER\_PTE,value\_gatePTE);

}

void latch\_datapath\_values() {

/\*

\* Datapath routine for computing all functions that need to latch

\* values in the data path at the end of this cycle. Some values

\* require sourcing the bus; therefore, this routine has to come

\* after drive\_bus.

\*/

int PSR\_changed = 0;

int ld\_pte = GetLD\_PTE(CURRENT\_LATCHES.MICROINSTRUCTION);

int ld\_va = GetLD\_VA(CURRENT\_LATCHES.MICROINSTRUCTION);

int mux\_pte = GetPTEMUX(CURRENT\_LATCHES.MICROINSTRUCTION);

int ld\_ptbr = GetLD\_PTBR(CURRENT\_LATCHES.MICROINSTRUCTION);

int ld\_psr = GetLD\_PSR(CURRENT\_LATCHES.MICROINSTRUCTION);

int ld\_exc = GetLD\_EXEC(CURRENT\_LATCHES.MICROINSTRUCTION);

int ld\_ssp = GetLD\_SSP(CURRENT\_LATCHES.MICROINSTRUCTION);

int ld\_usp = GetLD\_USP(CURRENT\_LATCHES.MICROINSTRUCTION);

int ld\_vector = GetLD\_VECTOR(CURRENT\_LATCHES.MICROINSTRUCTION);

//printf("\n\nLD values: pte: %d\n va:%d\n ptbr: %d\n",ld\_pte,ld\_va,ld\_ptbr);

//printf("\n\nLD values: PSR: %d\n exc:%d\nSSP: %d\n USP:%d\nvector: %d\n\n",ld\_psr,ld\_exc,ld\_ssp,ld\_usp,ld\_vector);

if(mem\_cycle==MEM\_CYCLES-1){

NEXT\_LATCHES.READY=1;

}

int ld\_mar = GetLD\_MAR(CURRENT\_LATCHES.MICROINSTRUCTION);

int ld\_mdr = GetLD\_MDR(CURRENT\_LATCHES.MICROINSTRUCTION);

int ld\_ir = GetLD\_IR(CURRENT\_LATCHES.MICROINSTRUCTION);

int ld\_ben = GetLD\_BEN(CURRENT\_LATCHES.MICROINSTRUCTION);

int ld\_reg = GetLD\_REG(CURRENT\_LATCHES.MICROINSTRUCTION);

int ld\_cc = GetLD\_CC(CURRENT\_LATCHES.MICROINSTRUCTION);

int ld\_pc = GetLD\_PC(CURRENT\_LATCHES.MICROINSTRUCTION);

int dr\_mux = GetDRMUX(CURRENT\_LATCHES.MICROINSTRUCTION);

if(ld\_pte){

if(mux\_pte==0){

NEXT\_LATCHES.PTE = BUS;

}

else if(mux\_pte==1){

NEXT\_LATCHES.PTE = value\_muxPTE;

}

}

if(ld\_va){

NEXT\_LATCHES.VA = BUS;

}

if(ld\_psr){

int psrMux = GetPSRMUX(CURRENT\_LATCHES.MICROINSTRUCTION);

if(psrMux==0){

NEXT\_LATCHES.PSR = CURRENT\_LATCHES.PSR & 0x7FFF;

PSR\_changed=1;

}

if(psrMux==1){

NEXT\_LATCHES.PSR = BUS;

int n = INSTR(2,1,NEXT\_LATCHES.PSR);

int z = INSTR(1,1,NEXT\_LATCHES.PSR);

int p = INSTR(0,1,NEXT\_LATCHES.PSR);

PSR\_changed=1;

if(z){

NEXT\_LATCHES.Z=1;

NEXT\_LATCHES.P=0;

NEXT\_LATCHES.N=0;

}

else if(n){

NEXT\_LATCHES.Z=0;

NEXT\_LATCHES.P=0;

NEXT\_LATCHES.N=1;

}

else if(p){

NEXT\_LATCHES.Z=0;

NEXT\_LATCHES.P=1;

NEXT\_LATCHES.N=0;

}

}

}

if(ld\_ssp){

int sr1Mux = GetSR1MUX(CURRENT\_LATCHES.MICROINSTRUCTION);

int sr = INSTR(9,3,CURRENT\_LATCHES.IR);

if(sr1Mux==1){

sr = INSTR(6,3,CURRENT\_LATCHES.IR);

}

if(sr1Mux==2){

sr=6;

}

NEXT\_LATCHES.SSP = CURRENT\_LATCHES.REGS[sr];

}

if(ld\_usp){

int sr1Mux = GetSR1MUX(CURRENT\_LATCHES.MICROINSTRUCTION);

int sr = INSTR(9,3,CURRENT\_LATCHES.IR);

if(sr1Mux==1){

sr = 7;

}

if(sr1Mux==2){

sr=6;

}

NEXT\_LATCHES.USP = CURRENT\_LATCHES.REGS[sr];

}

if(ld\_vector){

NEXT\_LATCHES.VECT = CURRENT\_LATCHES.VECT<<1;

}

if(ld\_mar){

NEXT\_LATCHES.MAR = BUS;

}

if(ld\_mdr){

NEXT\_LATCHES.MDR = BUS;

}

if(ld\_ir){

NEXT\_LATCHES.IR = BUS;

}

if(ld\_ben){

int n = CURRENT\_LATCHES.N;

int z = CURRENT\_LATCHES.Z;

int p = CURRENT\_LATCHES.P;

int ir\_n = INSTR(11,1,CURRENT\_LATCHES.IR);

int ir\_z = INSTR(10,1,CURRENT\_LATCHES.IR);

int ir\_p = INSTR(9,1,CURRENT\_LATCHES.IR);

NEXT\_LATCHES.BEN = (n & ir\_n) + (z & ir\_z) + (p & ir\_p);

}

if(ld\_reg){

int dr = INSTR(9,3,CURRENT\_LATCHES.IR);

if(dr\_mux==1){

dr=7;

}

if(dr\_mux==2){

dr=6;

}

NEXT\_LATCHES.REGS[dr]= BUS;

}

if(ld\_cc){

int sign\_bit= INSTR(15,1,BUS);

if(BUS==0){

NEXT\_LATCHES.Z=1;

NEXT\_LATCHES.P=0;

NEXT\_LATCHES.N=0;

}

else if(sign\_bit){

NEXT\_LATCHES.Z=0;

NEXT\_LATCHES.P=0;

NEXT\_LATCHES.N=1;

}

else if(sign\_bit==0){

NEXT\_LATCHES.Z=0;

NEXT\_LATCHES.P=1;

NEXT\_LATCHES.N=0;

}

}

if(ld\_pc){

int pc\_mux = GetPCMUX(CURRENT\_LATCHES.MICROINSTRUCTION);

if(pc\_mux==0){

NEXT\_LATCHES.PC = CURRENT\_LATCHES.PC+2;

}

else if(pc\_mux==1){

NEXT\_LATCHES.PC = BUS;

}

else if(pc\_mux==2){

NEXT\_LATCHES.PC = main\_addr\_output;

}

}

if(mem\_cycle==MEM\_CYCLES) {

mem\_cycle=0;

int state = GetJ(CURRENT\_LATCHES.MICROINSTRUCTION);

int read\_write = GetR\_W(CURRENT\_LATCHES.MICROINSTRUCTION);

int data\_size = GetDATA\_SIZE(CURRENT\_LATCHES.MICROINSTRUCTION);

int updated\_mar = CURRENT\_LATCHES.MAR >> 1;

int current\_mdr = CURRENT\_LATCHES.MDR;

if (read\_write) {

if (data\_size) {

//state 16

int msb\_md = (current\_mdr & 0xFF00)>>8;

int lsb\_md = current\_mdr & 0x00FF;

MEMORY[updated\_mar][0] = lsb\_md;

MEMORY[updated\_mar][1] = msb\_md;

} else {

//state 17

int mar\_zero = INSTR(0, 1, CURRENT\_LATCHES.MAR);

MEMORY[updated\_mar][mar\_zero] = Low8bits(current\_mdr);

}

} else {

if (ld\_mdr==1) {

//state 25,28,29,33

NEXT\_LATCHES.MDR = Low16bits(

(Low8bits(MEMORY[updated\_mar][0])) + High8bits((MEMORY[updated\_mar][1]) << 8));

}

}

}

if(CURRENT\_LATCHES.STATE\_NUMBER==8 && exception\_occur==1){

exception\_occur=0;

}

int interr = inter\_request && CURRENT\_LATCHES.STATE\_NUMBER==18 && !exception\_occur;

int exception\_caused = 0;

int next\_instr[CONTROL\_STORE\_BITS];

int J = GetJ(CURRENT\_LATCHES.MICROINSTRUCTION);

for(int i=0;i<CONTROL\_STORE\_BITS;i++){

next\_instr[i] = CONTROL\_STORE[J][i];

}

int next\_MAR\_value = NEXT\_LATCHES.MAR;

int IRD = GetIRD(CURRENT\_LATCHES.MICROINSTRUCTION);

int psr15 = INSTR(15,1,CURRENT\_LATCHES.PSR);

if(NEXT\_LATCHES.STATE\_NUMBER==10 || NEXT\_LATCHES.STATE\_NUMBER==11){

printf("\nEXCEPTION FOR OPCODE \n");

exception\_caused=1;

NEXT\_LATCHES.VECT = opcode;

for(int i=0;i<CONTROL\_STORE\_BITS;i++){

next\_instr[i] = CONTROL\_STORE[54][i];

}

memcpy(NEXT\_LATCHES.MICROINSTRUCTION,next\_instr,sizeof(int)\*CONTROL\_STORE\_BITS);

NEXT\_LATCHES.STATE\_NUMBER = 54;

}

if(IRD==2){

int rs = GetRS(CURRENT\_LATCHES.MICROINSTRUCTION);

if(rs==1){

//check for unaligned access here

int unaligned\_check = INSTR(0,1,NEXT\_LATCHES.MAR);

if(GetDATA\_SIZE(next\_instr) && unaligned\_check && exception\_caused==0){

printf("\nEXCEPTION FOR UNALIGNED \n");

exception\_caused=1;

NEXT\_LATCHES.VECT=unaligned;

}

}

else if(rs==0){

int protectBit = INSTR(3,1,NEXT\_LATCHES.PTE);

int valid = INSTR(2,1,NEXT\_LATCHES.PTE);

int value = (protectBit==0 && psr15==1 && exception\_caused==0) || (protectBit==0 && return\_state!=28 && exception\_caused==0);

//printf("\n\nvalue1: %X values: protectBit: %X valid: %X exceptioncaused: %X psr15: %X return state: %X\n",value,protectBit,valid,exception\_caused,psr15,return\_state);

if(protectBit==0 && !(psr15==0 || return\_state==28) && exception\_caused==0){

printf("\nEXCEPTION FOR PROTECT \n");

//printf("Cycle: %d\n",CYCLE\_COUNT);

printf("\n\n NEXT latches: GLOBAL VALUES:\n return\_state: %d\n PTBR: %X\n VA: %X\n PTE: %X\n ",return\_state,NEXT\_LATCHES.PTBR,NEXT\_LATCHES.VA,NEXT\_LATCHES.PTE);

printf("\nNEXT STATE NUMBER NOW: %d\n\n",NEXT\_LATCHES.STATE\_NUMBER);

exception\_caused=1;

NEXT\_LATCHES.VECT=protect;

}

if(exception\_caused==0 && valid==0){

//printf("\nEXCEPTION FOR PAGEFAULT \n");

exception\_caused=1;

NEXT\_LATCHES.VECT=pageFault;

}

}

// if(0<=next\_MAR\_value && next\_MAR\_value<=0x2FFF && psr15==1){

// exception\_caused=1;

// NEXT\_LATCHES.VECT = protect;

// }

// int unaligned\_check = INSTR(0,1,NEXT\_LATCHES.MAR);

// int next\_j = GetJ(CURRENT\_LATCHES.MICROINSTRUCTION);

// for(int i=0;i<CONTROL\_STORE\_BITS;i++){

// next\_instr[i] = CONTROL\_STORE[next\_j][i];

// }

// int testing\_off = 0;

// if(GetDATA\_SIZE(next\_instr) && unaligned\_check && exception\_caused==0){

// exception\_caused=1;

// NEXT\_LATCHES.VECT = unaligned;

// }

if(interr==1 && exception\_caused==0){

printf("\ninterrupt happens\n");

for(int i=0;i<CONTROL\_STORE\_BITS;i++){

next\_instr[i] = CONTROL\_STORE[38][i];

}

memcpy(NEXT\_LATCHES.MICROINSTRUCTION,next\_instr,sizeof(int)\*CONTROL\_STORE\_BITS);

NEXT\_LATCHES.STATE\_NUMBER = 38;

NEXT\_LATCHES.VECT = interruptLC<<1;

inter\_request=0;

}

if(exception\_caused==1){

exception\_occur=1;

for(int i=0;i<CONTROL\_STORE\_BITS;i++){

next\_instr[i] = CONTROL\_STORE[54][i];

}

memcpy(NEXT\_LATCHES.MICROINSTRUCTION,next\_instr,sizeof(int)\*CONTROL\_STORE\_BITS);

NEXT\_LATCHES.STATE\_NUMBER = 54;

}

if(exception\_caused==0 && interr==0 && rs==1){

for(int i=0;i<CONTROL\_STORE\_BITS;i++){

next\_instr[i] = CONTROL\_STORE[34][i];

}

memcpy(NEXT\_LATCHES.MICROINSTRUCTION,next\_instr,sizeof(int)\*CONTROL\_STORE\_BITS);

NEXT\_LATCHES.STATE\_NUMBER = 34;

}

if(exception\_caused==0 && interr==0 && rs==0){

for(int i=0;i<CONTROL\_STORE\_BITS;i++){

next\_instr[i] = CONTROL\_STORE[J][i];

}

memcpy(NEXT\_LATCHES.MICROINSTRUCTION,next\_instr,sizeof(int)\*CONTROL\_STORE\_BITS);

NEXT\_LATCHES.STATE\_NUMBER = J;

}

}

if(IRD==3){

for(int i=0;i<CONTROL\_STORE\_BITS;i++){

next\_instr[i] = CONTROL\_STORE[return\_state][i];

}

memcpy(NEXT\_LATCHES.MICROINSTRUCTION,next\_instr,sizeof(int)\*CONTROL\_STORE\_BITS);

NEXT\_LATCHES.STATE\_NUMBER = return\_state;

}

if(PSR\_changed==0){

if(NEXT\_LATCHES.Z==1){

NEXT\_LATCHES.PSR = (CURRENT\_LATCHES.PSR | 0x0002) & 0xFFF2;

}

if(NEXT\_LATCHES.P==1){

NEXT\_LATCHES.PSR = (CURRENT\_LATCHES.PSR | 0x0001) & 0xFFF1;

}

if(NEXT\_LATCHES.N==1){

NEXT\_LATCHES.PSR = (CURRENT\_LATCHES.PSR | 0x0004) & 0xFFF4;

}

}

int save\_MDR = GetSAVEMDR(CURRENT\_LATCHES.MICROINSTRUCTION);

if(save\_MDR==1){

protect\_mdr= Low16bits(CURRENT\_LATCHES.MDR);

}

int load\_MDR = GetLOADMDR(CURRENT\_LATCHES.MICROINSTRUCTION);

if(load\_MDR==1){

NEXT\_LATCHES.MDR = Low16bits(protect\_mdr);

}

printf("\n\n NEXT latches: GLOBAL VALUES:\n return\_state: %d\n PTBR: %X\n VA: %X\n PTE: %X\n ",return\_state,NEXT\_LATCHES.PTBR,NEXT\_LATCHES.VA,NEXT\_LATCHES.PTE);

printf("\nNEXT STATE NUMBER NOW: %d\n\n",NEXT\_LATCHES.STATE\_NUMBER);

}