```
/***************************************************************
 *
 * ssim.c - Sequential Y86 simulator
 *
 * Copyright (c) 2002, R. Bryant and D. O'Hallaron, All rights reserved.
 * May not be used, modified, or copied without permission.
 ***************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#include <unistd.h>
#include <string.h>
#include "isa.h"
#include "sim.h"

#define MAXBUF 1024

#ifdef HAS_GUI
#include <tk.h>
#endif /* HAS_GUI */

#define MAXARGS 128
#define MAXBUF 1024
#define TKARGS 3

/***************
 * Begin Globals
 ***************/

/* Simulator name defined and initialized by the compiled HCL file */
/* according to the -n argument supplied to hcl2c */
extern char simname[];

/* SEQ=0, SEQ+=1. Modified by HCL main() */
int plusmode = 0;

/* Parameters modifed by the command line */
int gui_mode = FALSE;    /* Run in GUI mode instead of TTY mode? (-g) */
char *object_filename;   /* The input object file name. */
FILE *object_file;       /* Input file handle */
bool_t verbosity = 2;    /* Verbosity level [TTY only] (-v) */
int instr_limit = 10000; /* Instruction limit [TTY only] (-l) */
bool_t do_check = FALSE; /* Test with YIS? [TTY only] (-t) */

/*************
 * End Globals
 *************/


/*************************
 * Begin function prototypes
 *************************/

static void usage(char *name);           /* Print helpful usage message */
static void run_tty_sim();               /* Run simulator in TTY mode */

#ifdef HAS_GUI
void addAppCommands(Tcl_Interp *interp); /* Add application-dependent commands */
#endif /* HAS_GUI */

/************************
 * End function prototypes
 ************************/


/***************************************************************
 * Part 1: This part is the initial entry point that handles general
 * initialization. It parses the command line and does any necessary
 * setup to run in either TTY or GUI mode, and then starts the
 * simulation.
 ***************************************************************/

/*
 * sim_main - main simulator routine. This function is called from the
 * main() routine in the HCL file.
 */
int sim_main(int argc, char **argv)
{
    int i;
    int c;
    char *myargv[MAXARGS];


    /* Parse the command line arguments */
    while ((c = getopt(argc, argv, "htgl:v:")) != -1) {
        switch(c) {
        case 'h':
            usage(argv[0]);
            break;
        case 'l':
            instr_limit = atoi(optarg);
            break;
        case 'v':
            verbosity = atoi(optarg);
            if (verbosity < 0 || verbosity > 2) {
                printf("Invalid verbosity %d\n", verbosity);
                usage(argv[0]);
            }
            break;
        case 't':
            do_check = TRUE;
            break;
        case 'g':
            gui_mode = TRUE;
            break;
        default:
            printf("Invalid option '%c'\n", c);
            usage(argv[0]);
            break;
        }
    }


    /* Do we have too many arguments? */
    if (optind < argc - 1) {
        printf("Too many command line arguments:");
        for (i = optind; i < argc; i++)
            printf(" %s", argv[i]);
        printf("\n");
        usage(argv[0]);
    }
```

```
    /* The single unflagged argument should be the object file name */
    object_filename = NULL;
    object_file = NULL;
    if (optind < argc) {
        object_filename = argv[optind];
        object_file = fopen(object_filename, "r");
        if (!object_file) {
            fprintf(stderr, "Couldn't open object file %s\n", object_filename);
            exit(1);
        }
    }


    /* Run the simulator in GUI mode (-g flag) */
    if (gui_mode) {

#ifndef HAS_GUI
        printf("To run in GUI mode, you must recompile with the HAS_GUI constant defin
ed.\n");
        exit(1);
#endif /* HAS_GUI */

        /* In GUI mode, we must specify the object file on command line */
        if (!object_file) {
            printf("Missing object file argument in GUI mode\n");
            usage(argv[0]);
        }

        /* Build the command line for the GUI simulator */
        for (i = 0; i < TKARGS; i++) {
            if ((myargv[i] = malloc(MAXBUF*sizeof(char))) == NULL) {
                perror("malloc error");
                exit(1);
            }
        }
        strcpy(myargv[0], argv[0]);

#if 0
        printf("argv[0]=%s\n", argv[0]);
        {
            char buf[1000];
            getcwd(buf, 1000);
            printf("cwd=%s\n", buf);
        }
#endif

        if (plusmode == 0) /* SEQ */
            strcpy(myargv[1], "seq.tcl");
        else
            strcpy(myargv[1], "seq+.tcl");
        strcpy(myargv[2], object_filename);
        myargv[3] = NULL;

        /* Start the GUI simulator */
#ifdef HAS_GUI
        Tk_Main(TKARGS, myargv, Tcl_AppInit);
#endif /* HAS_GUI */
        exit(0);
    }

    /* Otherwise, run the simulator in TTY mode (no -g flag) */
    run_tty_sim();

    exit(0);
}

/*
 * run_tty_sim - Run the simulator in TTY mode
 */
static void run_tty_sim()
{
    int icount = 0;
    status = STAT_AOK;
    cc_t result_cc = 0;
    int byte_cnt = 0;
    mem_t mem0, reg0;
    state_ptr isa_state = NULL;


    /* In TTY mode, the default object file comes from stdin */
    if (!object_file) {
        object_file = stdin;
    }

    /* Initializations */
    if (verbosity >= 2)
        sim_set_dumpfile(stdout);
    sim_init();

    /* Emit simulator name */
    printf("%s\n", simname);

    byte_cnt = load_mem(mem, object_file, 1);
    if (byte_cnt == 0) {
        fprintf(stderr, "No lines of code found\n");
        exit(1);
    } else if (verbosity >= 2) {
        printf("%d bytes of code read\n", byte_cnt);
    }
    fclose(object_file);
    if (do_check) {
        isa_state = new_state(0);
        free_mem(isa_state->r);
        free_mem(isa_state->m);
        isa_state->m = copy_mem(mem);
        isa_state->r = copy_mem(reg);
        isa_state->cc = cc;
    }

    mem0 = copy_mem(mem);
    reg0 = copy_mem(reg);


    icount = sim_run(instr_limit, &status, &result_cc);
    if (verbosity > 0) {
        printf("%d instructions executed\n", icount);
        printf("Status = %s\n", stat_name(status));
        printf("Condition Codes: %s\n", cc_name(result_cc));
        printf("Changed Register State:\n");
        diff_reg(reg0, reg, stdout);
```

```
        printf("Changed Memory State:\n");
        diff_mem(mem0, mem, stdout);
    }
    if (do_check) {
        byte_t e = STAT_AOK;
        int step;
        bool_t match = TRUE;

        for (step = 0; step < instr_limit && e == STAT_AOK; step++) {
            e = step_state(isa_state, stdout);
        }

        if (diff_reg(isa_state->r, reg, NULL)) {
            match = FALSE;
            if (verbosity > 0) {
                printf("ISA Register != Pipeline Register File\n");
                diff_reg(isa_state->r, reg, stdout);
            }
        }
        if (diff_mem(isa_state->m, mem, NULL)) {
            match = FALSE;
            if (verbosity > 0) {
                printf("ISA Memory != Pipeline Memory\n");
                diff_mem(isa_state->m, mem, stdout);
            }
        }
        if (isa_state->cc != result_cc) {
            match = FALSE;
            if (verbosity > 0) {
                printf("ISA Cond. Codes (%s) != Pipeline Cond. Codes (%s)\n",
                       cc_name(isa_state->cc), cc_name(result_cc));
            }
        }
        if (match) {
            printf("ISA Check Succeeds\n");
        } else {
            printf("ISA Check Fails\n");
        }
    }
}



/*
 * usage - print helpful diagnostic information
 */
static void usage(char *name)
{
    printf("Usage: %s [-htg] [-l m] [-v n] file.yo\n", name);
    printf("file.yo required in GUI mode, optional in TTY mode (default stdin)\n");
    printf("   -h     Print this message\n");
    printf("   -g     Run in GUI mode instead of TTY mode (default TTY)\n");
    printf("   -l m   Set instruction limit to m [TTY mode only] (default %d)\n", inst
r_limit);
    printf("   -v n   Set verbosity level to 0 <= n <= 2 [TTY mode only] (default %d)\
n", verbosity);
    printf("   -t     Test result against ISA simulator (yis) [TTY mode only]\n");
    exit(0);
}
```

```
/***************************************************
 * Part 2: This part contains the core simulator routines.
 ***************************************************/

/*********************
 * Begin Part 2 Globals
 *********************/

/*
 * Variables related to hardware units in the processor
 */
mem_t mem;  /* Instruction and data memory */
int minAddr = 0;
int memCnt = 0;

/* Other processor state */
mem_t reg;               /* Register file */
cc_t cc = DEFAULT_CC;    /* Condition code register */
cc_t cc_in = DEFAULT_CC; /* Input to condition code register */

/*
 * SEQ+: Results computed by previous instruction.
 * Used to compute PC in current instruction
 */
byte_t prev_icode = I_NOP;
byte_t prev_ifun = 0;
word_t prev_valc = 0;
word_t prev_valm = 0;
word_t prev_valp = 0;
bool_t prev_bcond = FALSE;

byte_t prev_icode_in = I_NOP;
byte_t prev_ifun_in = 0;
word_t prev_valc_in = 0;
word_t prev_valm_in = 0;
word_t prev_valp_in = 0;
bool_t prev_bcond_in = FALSE;


/* Program Counter */
word_t pc = 0; /* Program counter value */
word_t pc_in = 0;/* Input to program counter */

/* Intermediate values */
byte_t imem_icode = I_NOP;
byte_t imem_ifun = F_NONE;
byte_t icode = I_NOP;
word_t ifun = 0;
byte_t instr = HPACK(I_NOP, F_NONE);
word_t ra = REG_NONE;
word_t rb = REG_NONE;
word_t valc = 0;
word_t valp = 0;
bool_t imem_error;
bool_t instr_valid;

word_t srcA = REG_NONE;
word_t srcB = REG_NONE;
word_t destE = REG_NONE;
word_t destM = REG_NONE;
```

```
word_t vala = 0;
word_t valb = 0;
word_t vale = 0;

bool_t bcond = FALSE;
bool_t cond = FALSE;
word_t valm = 0;
bool_t dmem_error;

bool_t mem_write = FALSE;
word_t mem_addr = 0;
word_t mem_data = 0;
byte_t status = STAT_AOK;


/* Values computed by control logic */
int gen_pc();  /* SEQ+ */
int gen_icode();
int gen_ifun();
int gen_need_regids();
int gen_need_valC();
int gen_instr_valid();
int gen_srcA();
int gen_srcB();
int gen_dstE();
int gen_dstM();
int gen_aluA();
int gen_aluB();
int gen_alufun();
int gen_set_cc();
int gen_mem_addr();
int gen_mem_data();
int gen_mem_read();
int gen_mem_write();
int gen_Stat();
int gen_new_pc();

/* Log file */
FILE *dumpfile = NULL;

#ifdef HAS_GUI
/* Representations of digits */
static char digits[16] =
    {'0', '1', '2', '3', '4', '5', '6', '7',
     '8', '9', 'A', 'B', 'C', 'D', 'E', 'F'};
#endif /* HAS_GUI */

/********************
 * End Part 2 Globals
 ********************/

#ifdef HAS_GUI

/* Create string in hex/oct/binary format with leading zeros */
/* bpd denotes bits per digit  Should be in range 1-4,
   bpw denotes bits per word.*/
void wstring(unsigned x, int bpd, int bpw, char *str)
{
    int digit;
    unsigned mask = (1 << bpd) - 1;
    for (digit = (bpw-1)/bpd; digit >= 0; digit--) {
        unsigned val = (x >> (digit * bpd)) & mask;
        *str++ = digits[val];
    }
    *str = '\0';
}

/* used for formatting instructions */
static char status_msg[128];

/* SEQ+ */
static char *format_prev()
{
    char istring[9];
    char mstring[9];
    char pstring[9];
    wstring(prev_valc, 4, 32, istring);
    wstring(prev_valm, 4, 32, mstring);
    wstring(prev_valp, 4, 32, pstring);
    sprintf(status_msg, "%c %s %s %s %s",
            prev_bcond ? 'Y' : 'N',
            iname(HPACK(prev_icode, prev_ifun)),
            istring, mstring, pstring);

    return status_msg;
}

static char *format_pc()
{
    char pstring[9];
    wstring(pc, 4, 32, pstring);
    sprintf(status_msg, "%s", pstring);
    return status_msg;
}

static char *format_f()
{
    char valcstring[9];
    char valpstring[9];
    wstring(valc, 4, 32, valcstring);
    wstring(valp, 4, 32, valpstring);
    sprintf(status_msg, "%s %s %s %s %s",
            iname(HPACK(icode, ifun)),
            reg_name(ra),
            reg_name(rb),
            valcstring,
            valpstring);
    return status_msg;
}

static char *format_d()
{
    char valastring[9];
    char valbstring[9];
    wstring(vala, 4, 32, valastring);
    wstring(valb, 4, 32, valbstring);
    sprintf(status_msg, "%s %s %s %s %s %s",
            valastring,
            valbstring,
            reg_name(destE),
            reg_name(destM),
            reg_name(srcA),
```

```
            reg_name(srcB));

    return status_msg;
}

static char *format_e()
{
    char valestring[9];
    wstring(vale, 4, 32, valestring);
    sprintf(status_msg, "%c %s",
            bcond ? 'Y' : 'N',
            valestring);
    return status_msg;
}

static char *format_m()
{
    char valmstring[9];
    wstring(valm, 4, 32, valmstring);
    sprintf(status_msg, "%s", valmstring);
    return status_msg;
}

static char *format_npc()
{
    char npcstring[9];
    wstring(pc_in, 4, 32, npcstring);
    sprintf(status_msg, "%s", npcstring);
    return status_msg;
}
#endif /* HAS_GUI */

/* Report system state */
static void sim_report() {

#ifdef HAS_GUI
    if (gui_mode) {
        report_pc(pc);
        if (plusmode) {
            report_state("PREV", format_prev());
            report_state("PC", format_pc());
        } else {
            report_state("OPC", format_pc());
        }
        report_state("F", format_f());
        report_state("D", format_d());
        report_state("E", format_e());
        report_state("M", format_m());
        if (!plusmode) {
            report_state("NPC", format_npc());
        }
        show_cc(cc);
    }
#endif /* HAS_GUI */

}

static int initialized = 0;
void sim_init()
{
    /* Create memory and register files */
    initialized = 1;
    mem = init_mem(MEM_SIZE);
    reg = init_reg();
    sim_reset();
    clear_mem(mem);
}

void sim_reset()
{
    if (!initialized)
        sim_init();
    clear_mem(reg);
    minAddr = 0;
    memCnt = 0;

#ifdef HAS_GUI
    if (gui_mode) {
        signal_register_clear();
        create_memory_display(minAddr, memCnt);
        sim_report();
    }
#endif

    if (plusmode) {
        prev_icode = prev_icode_in = I_NOP;
        prev_ifun = prev_ifun_in = 0;
        prev_valc = prev_valc_in = 0;
        prev_valm = prev_valm_in = 0;
        prev_valp = prev_valp_in = 0;
        prev_bcond = prev_bcond_in = FALSE;
        pc = 0;
    } else {
        pc_in = 0;
    }
    cc = DEFAULT_CC;
    cc_in = DEFAULT_CC;
    destE = REG_NONE;
    destM = REG_NONE;
    mem_write = FALSE;
    mem_addr = 0;
    mem_data = 0;

    /* Reset intermediate values to clear display */
    icode = I_NOP;
    ifun = 0;
    instr = HPACK(I_NOP, F_NONE);
    ra = REG_NONE;
    rb = REG_NONE;
    valc = 0;
    valp = 0;

    srcA = REG_NONE;
    srcB = REG_NONE;
    destE = REG_NONE;
    destM = REG_NONE;
    vala = 0;
    valb = 0;
    vale = 0;

    cond = FALSE;
```

```
    bcond = FALSE;
    valm = 0;

    sim_report();
}

/* Update the processor state */
static void update_state()
{
    if (plusmode) {
        prev_icode = prev_icode_in;
        prev_ifun  = prev_ifun_in;
        prev_valc  = prev_valc_in;
        prev_valm  = prev_valm_in;
        prev_valp  = prev_valp_in;
        prev_bcond = prev_bcond_in;
    } else {
        pc = pc_in;
    }
    cc = cc_in;
    /* Writeback */
    if (destE != REG_NONE)
        set_reg_val(reg, destE, vale);
    if (destM != REG_NONE)
        set_reg_val(reg, destM, valm);

    if (mem_write) {
      /* Should have already tested this address */
      set_word_val(mem, mem_addr, mem_data);
        sim_log("Wrote 0x%x to address 0x%x\n", mem_data, mem_addr);
#ifdef HAS_GUI
            if (gui_mode) {
                if (mem_addr % 4 != 0) {
                    /* Just did a misaligned write.
                       Need to display both words */
                    word_t align_addr = mem_addr & ~0x3;
                    word_t val;
                    get_word_val(mem, align_addr, &val);
                    set_memory(align_addr, val);
                    align_addr+=4;
                    get_word_val(mem, align_addr, &val);
                    set_memory(align_addr, val);
                } else {
                    set_memory(mem_addr, mem_data);
                }
            }
#endif /* HAS_GUI */
    }
}

/* Execute one instruction */
/* Return resulting status */
static byte_t sim_step()
{
    word_t aluA;
    word_t aluB;
    word_t alufun;

    status = STAT_AOK;
    imem_error = dmem_error = FALSE;

update_state(); /* Update state from last cycle */

if (plusmode) {
    pc = gen_pc();
}
valp = pc;
instr = HPACK(I_NOP, F_NONE);
imem_error = !get_byte_val(mem, valp, &instr);
if (imem_error) {
    sim_log("Couldn't fetch at address 0x%x\n", valp);
}
imem_icode = HI4(instr);
imem_ifun = LO4(instr);
icode = gen_icode();
ifun  = gen_ifun();
instr_valid = gen_instr_valid();
valp++;
if (gen_need_regids()) {
    byte_t regids;
    if (get_byte_val(mem, valp, &regids)) {
        ra = GET_RA(regids);
        rb = GET_RB(regids);
    } else {
        ra = REG_NONE;
        rb = REG_NONE;
        status = STAT_ADR;
        sim_log("Couldn't fetch at address 0x%x\n", valp);
    }
    valp++;
} else {
    ra = REG_NONE;
    rb = REG_NONE;
}

if (gen_need_valC()) {
    if (get_word_val(mem, valp, &valc)) {
    } else {
        valc = 0;
        status = STAT_ADR;
        sim_log("Couldn't fetch at address 0x%x\n", valp);
    }
    valp+=4;
} else {
    valc = 0;
}
sim_log("IF: Fetched %s at 0x%x.  ra=%s, rb=%s, valC = 0x%x\n",
        iname(HPACK(icode,ifun)), pc, reg_name(ra), reg_name(rb), valc);

if (status == STAT_AOK && icode == I_HALT) {
    status = STAT_HLT;
}

srcA = gen_srcA();
if (srcA != REG_NONE) {
    vala = get_reg_val(reg, srcA);
} else {
    vala = 0;
}

srcB = gen_srcB();
if (srcB != REG_NONE) {
```

```
        valb = get_reg_val(reg, srcB);
    } else {
        valb = 0;
    }

    cond = cond_holds(cc, ifun);

    destE = gen_dstE();
    destM = gen_dstM();

    aluA = gen_aluA();
    aluB = gen_aluB();
    alufun = gen_alufun();
    vale = compute_alu(alufun, aluA, aluB);
    cc_in = cc;
    if (gen_set_cc())
        cc_in = compute_cc(alufun, aluA, aluB);

    bcond =  cond && (icode == I_JMP);

    mem_addr = gen_mem_addr();
    mem_data = gen_mem_data();


    if (gen_mem_read()) {
      dmem_error = dmem_error || !get_word_val(mem, mem_addr, &valm);
      if (dmem_error) {
        sim_log("Couldn't read at address 0x%x\n", mem_addr);
      }
    } else
      valm = 0;

    mem_write = gen_mem_write();
    if (mem_write) {
      /* Do a test read of the data memory to make sure address is OK */
      word_t junk;
      dmem_error = dmem_error || !get_word_val(mem, mem_addr, &junk);
    }

    status = gen_Stat();

    if (plusmode) {
        prev_icode_in = icode;
        prev_ifun_in = ifun;
        prev_valc_in = valc;
        prev_valm_in = valm;
        prev_valp_in = valp;
        prev_bcond_in = bcond;
    } else {
        /* Update PC */
        pc_in = gen_new_pc();
    }
    sim_report();
    return status;
}

/*
  Run processor until one of following occurs:
  - An error status is encountered in WB.
  - max_instr instructions have completed through WB
  Return number of instructions executed.
  if statusp nonnull, then will be set to status of final instruction
  if ccp nonnull, then will be set to condition codes of final instruction
*/
int sim_run(int max_instr, byte_t *statusp, cc_t *ccp)
{
    int icount = 0;
    byte_t run_status = STAT_AOK;
    while (icount < max_instr) {
        run_status = sim_step();
        icount++;
        if (run_status != STAT_AOK)
            break;
    }
    if (statusp)
        *statusp = run_status;
    if (ccp)
        *ccp = cc;
    return icount;
}

/* If dumpfile set nonNULL, lots of status info printed out */
void sim_set_dumpfile(FILE *df)
{
    dumpfile = df;
}

/*
 * sim_log dumps a formatted string to the dumpfile, if it exists
 * accepts variable argument list
 */
void sim_log( const char *format, ... ) {
    if (dumpfile) {
        va_list arg;
        va_start( arg, format );
        vfprintf( dumpfile, format, arg );
        va_end( arg );
    }
}


/*************************************************************
 * Part 3: This part contains simulation control for the TK
 * simulator.
 *************************************************************/

#ifdef HAS_GUI

/**********************
 * Begin Part 3 globals
 **********************/

/* Hack for SunOS */
extern int matherr();
int *tclDummyMathPtr = (int *) matherr;

static char tcl_msg[256];

/* Keep track of the TCL Interpreter */
static Tcl_Interp *sim_interp = NULL;
```

```
static mem_t post_load_mem;

/*********************
 * End Part 3 globals
 *********************/


/* function prototypes */
int simResetCmd(ClientData clientData, Tcl_Interp *interp,
                int argc, char *argv[]);
int simLoadCodeCmd(ClientData clientData, Tcl_Interp *interp,
                   int argc, char *argv[]);
int simLoadDataCmd(ClientData clientData, Tcl_Interp *interp,
                   int argc, char *argv[]);
int simRunCmd(ClientData clientData, Tcl_Interp *interp,
              int argc, char *argv[]);
void addAppCommands(Tcl_Interp *interp);

/*****************************************************************************
 *      tcl command definitions
 *****************************************************************************/

/* Implement command versions of the simulation functions */
int simResetCmd(ClientData clientData, Tcl_Interp *interp,
                int argc, char *argv[])
{
    sim_interp = interp;
    if (argc != 1) {
        interp->result = "No arguments allowed";
        return TCL_ERROR;
    }
    sim_reset();
    if (post_load_mem) {
        free_mem(mem);
        mem = copy_mem(post_load_mem);
    }
    interp->result = stat_name(STAT_AOK);
    return TCL_OK;
}

int simLoadCodeCmd(ClientData clientData, Tcl_Interp *interp,
                   int argc, char *argv[])
{
    FILE *object_file;
    int code_count;
    sim_interp = interp;
    if (argc != 2) {
        interp->result = "One argument required";
        return TCL_ERROR;
    }
    object_file = fopen(argv[1], "r");
    if (!object_file) {
        sprintf(tcl_msg, "Couldn't open code file '%s'", argv[1]);
        interp->result = tcl_msg;
        return TCL_ERROR;
    }
    sim_reset();
    code_count = load_mem(mem, object_file, 0);
    post_load_mem = copy_mem(mem);
    sprintf(tcl_msg, "%d", code_count);
    interp->result = tcl_msg;
    fclose(object_file);
    return TCL_OK;
}

int simLoadDataCmd(ClientData clientData, Tcl_Interp *interp,
                   int argc, char *argv[])
{
    FILE *data_file;
    int word_count = 0;
    interp->result = "Not implemented";
    return TCL_ERROR;


    sim_interp = interp;
    if (argc != 2) {
        interp->result = "One argument required";
        return TCL_ERROR;
    }
    data_file = fopen(argv[1], "r");
    if (!data_file) {
        sprintf(tcl_msg, "Couldn't open data file '%s'", argv[1]);
        interp->result = tcl_msg;
        return TCL_ERROR;
    }
    sprintf(tcl_msg, "%d", word_count);
    interp->result = tcl_msg;
    fclose(data_file);
    return TCL_OK;
}


int simRunCmd(ClientData clientData, Tcl_Interp *interp,
              int argc, char *argv[])
{
    int step_limit = 1;
    byte_t run_status;
    cc_t cc;
    sim_interp = interp;
    if (argc > 2) {
        interp->result = "At most one argument allowed";
        return TCL_ERROR;
    }
    if (argc >= 2 &&
        (sscanf(argv[1], "%d", &step_limit) != 1 ||
         step_limit < 0)) {
        sprintf(tcl_msg, "Cannot run for '%s' cycles!", argv[1]);
        interp->result = tcl_msg;
        return TCL_ERROR;
    }
    sim_run(step_limit, &run_status, &cc);
    interp->result = stat_name(run_status);
    return TCL_OK;
}

/*****************************************************************************
 *      registering the commands with tcl
 *****************************************************************************/

void addAppCommands(Tcl_Interp *interp)
{
    sim_interp = interp;
```

```
    Tcl_CreateCommand(interp, "simReset", (Tcl_CmdProc *) simResetCmd,
                      (ClientData) NULL, (Tcl_CmdDeleteProc *) NULL);
    Tcl_CreateCommand(interp, "simCode", (Tcl_CmdProc *) simLoadCodeCmd,
                      (ClientData) NULL, (Tcl_CmdDeleteProc *) NULL);
    Tcl_CreateCommand(interp, "simData", (Tcl_CmdProc *) simLoadDataCmd,
                      (ClientData) NULL, (Tcl_CmdDeleteProc *) NULL);
    Tcl_CreateCommand(interp, "simRun", (Tcl_CmdProc *) simRunCmd,
                      (ClientData) NULL, (Tcl_CmdDeleteProc *) NULL);
}

/*****************************************************************************
 *      tcl functionality called from within C
 *****************************************************************************/

/* Provide mechanism for simulator to update register display */
void signal_register_update(reg_id_t r, word_t val) {
    int code;
    sprintf(tcl_msg, "setReg %d %d 1", (int) r, (int) val);
    code = Tcl_Eval(sim_interp, tcl_msg);
    if (code != TCL_OK) {
        fprintf(stderr, "Failed to signal register set\n");
        fprintf(stderr, "Error Message was '%s'\n", sim_interp->result);
    }
}

/* Provide mechanism for simulator to generate memory display */
void create_memory_display() {
    int code;
    sprintf(tcl_msg, "createMem %d %d", minAddr, memCnt);
    code = Tcl_Eval(sim_interp, tcl_msg);
    if (code != TCL_OK) {
        fprintf(stderr, "Command '%s' failed\n", tcl_msg);
        fprintf(stderr, "Error Message was '%s'\n", sim_interp->result);
    } else {
        int i;
        for (i = 0; i < memCnt && code == TCL_OK; i+=4) {
            int addr = minAddr+i;
            int val;
            if (!get_word_val(mem, addr, &val)) {
                fprintf(stderr, "Out of bounds memory display\n");
                return;
            }
            sprintf(tcl_msg, "setMem %d %d", addr, val);
            code = Tcl_Eval(sim_interp, tcl_msg);
        }
        if (code != TCL_OK) {
            fprintf(stderr, "Couldn't set memory value\n");
            fprintf(stderr, "Error Message was '%s'\n", sim_interp->result);
        }
    }
}

/* Provide mechanism for simulator to update memory value */
void set_memory(int addr, int val) {
    int code;
    int nminAddr = minAddr;
    int nmemCnt = memCnt;

    /* First see if we need to expand memory range */
    if (memCnt == 0) {
        nminAddr = addr;
        nmemCnt = 4;
    } else if (addr < minAddr) {
        nminAddr = addr;
        nmemCnt = minAddr + memCnt - addr;
    } else if (addr >= minAddr+memCnt) {
        nmemCnt = addr-minAddr+4;
    }
    /* Now make sure nminAddr & nmemCnt are multiples of 16 */
    nmemCnt = ((nminAddr & 0xF) + nmemCnt + 0xF) & ~0xF;
    nminAddr = nminAddr & ~0xF;

    if (nminAddr != minAddr || nmemCnt != memCnt) {
        minAddr = nminAddr;
        memCnt = nmemCnt;
        create_memory_display();
    } else {
        sprintf(tcl_msg, "setMem %d %d", addr, val);
        code = Tcl_Eval(sim_interp, tcl_msg);
        if (code != TCL_OK) {
            fprintf(stderr, "Couldn't set memory value 0x%x to 0x%x\n",
                    addr, val);
            fprintf(stderr, "Error Message was '%s'\n", sim_interp->result);
        }
    }
}

/* Provide mechanism for simulator to update condition code display */
void show_cc(cc_t cc)
{
    int code;
    sprintf(tcl_msg, "setCC %d %d %d",
            GET_ZF(cc), GET_SF(cc), GET_OF(cc));
    code = Tcl_Eval(sim_interp, tcl_msg);
    if (code != TCL_OK) {
        fprintf(stderr, "Failed to display condition codes\n");
        fprintf(stderr, "Error Message was '%s'\n", sim_interp->result);
    }
}

/* Provide mechanism for simulator to clear register display */
void signal_register_clear() {
    int code;
    code = Tcl_Eval(sim_interp, "clearReg");
    if (code != TCL_OK) {
        fprintf(stderr, "Failed to signal register clear\n");
        fprintf(stderr, "Error Message was '%s'\n", sim_interp->result);
    }
}

/* Provide mechanism for simulator to report instructions as they are
   read in
*/

void report_line(int line_no, int addr, char *hex, char *text) {
    int code;
    sprintf(tcl_msg, "addCodeLine %d %d {%s} {%s}", line_no, addr, hex, text);
    code = Tcl_Eval(sim_interp, tcl_msg);
    if (code != TCL_OK) {
        fprintf(stderr, "Failed to report code line 0x%x\n", addr);
        fprintf(stderr, "Error Message was '%s'\n", sim_interp->result);
    }
```

```
}


/* Provide mechanism for simulator to report which instruction
   is being executed */
void report_pc(unsigned pc)
{
    int t_status;
    char addr[10];
    char code[12];
    Tcl_DString cmd;
    Tcl_DStringInit(&cmd);
    Tcl_DStringAppend(&cmd, "simLabel ", -1);
    Tcl_DStringStartSublist(&cmd);
    sprintf(addr, "%u", pc);
    Tcl_DStringAppendElement(&cmd, addr);

    Tcl_DStringEndSublist(&cmd);
    Tcl_DStringStartSublist(&cmd);
    sprintf(code, "%s","*");
    Tcl_DStringAppend(&cmd, code, -1);
    Tcl_DStringEndSublist(&cmd);
    t_status = Tcl_Eval(sim_interp, Tcl_DStringValue(&cmd));
    if (t_status != TCL_OK) {
        fprintf(stderr, "Failed to report code '%s'\n", code);
        fprintf(stderr, "Error Message was '%s'\n", sim_interp->result);
    }
}

/* Report single line of stage state */
void report_state(char *id, char *txt)
{
    int t_status;
    sprintf(tcl_msg, "updateStage %s {%s}", id, txt);
    t_status = Tcl_Eval(sim_interp, tcl_msg);
    if (t_status != TCL_OK) {
        fprintf(stderr, "Failed to report processor status\n");
        fprintf(stderr, "\tStage %s, status '%s'\n",
                id, txt);
        fprintf(stderr, "\tError Message was '%s'\n", sim_interp->result);
    }
}

/*
 * Tcl_AppInit - Called by TCL to perform application-specific initialization.
 */
int Tcl_AppInit(Tcl_Interp *interp)
{
    /* Tell TCL about the name of the simulator so it can  */
    /* use it as the title of the main window */
    Tcl_SetVar(interp, "simname", simname, TCL_GLOBAL_ONLY);

    if (Tcl_Init(interp) == TCL_ERROR)
        return TCL_ERROR;
    if (Tk_Init(interp) == TCL_ERROR)
        return TCL_ERROR;
    Tcl_StaticPackage(interp, "Tk", Tk_Init, Tk_SafeInit);

    /* Call procedure to add new commands */
    addAppCommands(interp);
    /*
     * Specify a user-specific startup file to invoke if the application
     * is run interactively.  Typically the startup file is "~/.apprc"
     * where "app" is the name of the application.  If this line is deleted
     * then no user-specific startup file will be run under any conditions.
     */
    Tcl_SetVar(interp, "tcl_rcFileName", "~/.wishrc", TCL_GLOBAL_ONLY);
    return TCL_OK;

}


#endif /* HAS_GUI */
```

```
################################################################################
# Parsing of command line flags                                                #
################################################################################

proc flagVal {flag default} {
    global argv
    foreach t $argv {
        if {[string match "-$flag*" $t]} {return [string range $t 2 end]}
    }
    return $default
}

proc findFlag {flag} {
    global argv
    foreach t $argv {
        if {[string match "-$flag" $t]} {return 1}
    }
    return 0
}

################################################################################
# Register File Implementation.  Shown as array of 8 columns                   #
################################################################################


# Font used to display register contents
set fontSize [expr 10 * [flagVal "f" 12]]
set codeFontSize [expr 10 * [flagVal "c" 10]]
set labFontSize [expr 10 * [flagVal "l" 10]]
set bigFontSize [expr 10 * [flagVal "b" 16]]
set dpyFont "*-courier-medium-r-normal--*-$fontSize-*-*-*-*-*-*"
set labFont "*-helvetica-medium-r-normal--*-$labFontSize-*-*-*-*-*-*"
set bigLabFont "*-helvetica-bold-r-normal--*-$bigFontSize-*-*-*-*-*-*"
set codeFont "*-courier-medium-r-normal--*-$codeFontSize-*-*-*-*-*-*"
# Background Color of normal register
set normalBg white
# Background Color of highlighted register
set specialBg LightSkyBlue

# Height of titles separating major sections of control panel
set sectionHeight 2

# How many rows of code do I display
set codeRowCount [flagVal "r" 50]

# Keep track of previous highlighted register
set lastId -1
proc setReg {id val highlight} {
    global lastId normalBg specialBg
    if {$lastId >= 0} {
        .r.reg$lastId config -bg $normalBg
        set lastId -1
    }
    if {$id < 0 || $id >= 8} {
        error "Invalid Register ($id)"
    }
    .r.reg$id config -text [format %8x $val]
    if {$highlight} {
        uplevel .r.reg$id config -bg $specialBg
        set lastId $id
    }
}

# Clear all registers
proc clearReg {} {
    global lastId normalBg
    if {$lastId >= 0} {
        .r.reg$lastId config -bg $normalBg
        set lastId -1
    }
    for {set i 0} {$i < 8} {incr i 1} {
        .r.reg$i config -text ""
    }
}

# Set all 3 condition codes
proc setCC {zv cv ov} {
    .cc.cc0 config -text [format %d $zv]
    .cc.cc1 config -text [format %d $cv]
    .cc.cc2 config -text [format %d $ov]
}


### Create display for misc. state
frame .flags
pack .flags -in . -side bottom

################################################################################
# Status Display                                                               #
################################################################################

set simStat "AOK"
# Line to display simulation status
frame .stat
pack .stat -in .flags -side left
label .stat.statlab -width 7 -text "Stat" -font $bigLabFont -height $sectionHeight
label .stat.statdpy -width 3 -font $dpyFont -relief ridge -bg white -textvariable simS
tat
label .stat.fill -width 6 -text ""
pack .stat.statlab .stat.statdpy .stat.fill  -in .stat -side left
################################################################################
# Condition Code Display                                                       #
################################################################################
# Create Window for condition codes
frame .cc
pack .cc -in .flags -side right

label .cc.lab -text "Condition Codes" -font $bigLabFont -height $sectionHeight
pack .cc.lab -in .cc -side left


set ccnames [list "Z" "S" "O"]

# Create Row of CC Labels
for {set i 0} {$i < 3} {incr i 1} {
    label .cc.lab$i -width 1 -font $dpyFont -text [lindex $ccnames $i]
    pack .cc.lab$i -in .cc -side left
    label .cc.cc$i -width 1 -font $dpyFont -relief ridge -bg $normalBg
    pack .cc.cc$i -in .cc -side left
}
```

```
################################################################################
# Register Display                                                             #
################################################################################


# Create Window for registers
frame .r
pack .r -in . -side bottom
# Following give separate window for register file
# toplevel .r
# wm title .r "Register File"
label .r.lab -text "Register File" -font $bigLabFont -height $sectionHeight
pack .r.lab -in .r -side top
# Set up top row control panel (disabled)
# frame .r.cntl
# pack .r.cntl -fill x -in .r
# label .r.labreg -text "Register" -width 10
# entry .r.regid -width 3 -relief sunken -textvariable regId -font $dpyFont
# label .r.labval -text "Value" -width 10
# entry .r.regval -width 8 -relief sunken -textvariable regVal -font $dpyFont
# button .r.doset -text "Set" -command {setReg $regId $regVal 1} -width 6
# button .r.c -text "Clear" -command clearReg -width 6
# pack .r.labreg .r.regid .r.labval .r.regval .r.doset .r.c  -in .r.cntl -side left

set regnames [list "%eax" "%ecx" "%edx" "%ebx" "%esp" "%ebp" "%esi" "%edi"]

# Create Row of Register Labels
frame .r.labels
pack .r.labels -side top -in .r

for {set i 0} {$i < 8} {incr i 1} {
    label .r.lab$i -width 8 -font $dpyFont -text [lindex $regnames $i]
    pack .r.lab$i -in .r.labels -side left
}

# Create Row of Register Entries
frame .r.row
pack .r.row -side top -in .r


# Create 8 registers
for {set i 0} {$i < 8} {incr i 1} {
    label .r.reg$i -width 8 -font $dpyFont -relief ridge \
            -bg $normalBg
    pack .r.reg$i -in .r.row -side left
}


################################################################################
#  Main Control Panel                                                          #
################################################################################

#
# Set the simulator name (defined in simname in ssim.c)
# as the title of the main window
#
wm title . $simname

# Control Panel for simulator
set cntlBW 9
frame .cntl
pack .cntl
button .cntl.quit -width $cntlBW -text Quit -command exit
button .cntl.run -width $cntlBW -text Go -command simGo
button .cntl.stop -width $cntlBW -text Stop -command simStop
button .cntl.step -width $cntlBW -text Step -command simStep
button .cntl.reset -width $cntlBW -text Reset -command simResetAll
pack .cntl.quit .cntl.run .cntl.stop .cntl.step .cntl.reset -in .cntl -side left
# Simulation speed control
scale .spd -label {Simulator Speed (10*log Hz)} -from -10 -to 30 -length 10c \
  -orient horizontal -command setSpeed
pack .spd

# Simulation mode
set simMode forward

# frame .md
# pack .md
# radiobutton .md.wedged -text Wedged -variable simMode \
#       -value wedged -width 10 -command {setSimMode wedged}
# radiobutton .md.stall -text Stall -variable simMode \
#       -value stall -width 10 -command {setSimMode stall}
# radiobutton .md.forward -text Forward -variable simMode \
#       -value forward -width 10 -command {setSimMode forward}
# pack .md.wedged .md.stall .md.forward -in .md -side left

# simDelay defines #milliseconds for each cycle of simulator
# Initial value is 1000ms
set simDelay 1000
# Set delay based on rate expressed in log(Hz)
proc setSpeed {rate} {
  global simDelay
  set simDelay [expr round(1000 / pow(10,$rate/10.0))]
}

# Global variables controlling simulator execution
# Should simulator be running now?
set simGoOK 0

proc simStop  {} {
  global simGoOK
  set simGoOK 0
}

proc simStep {} {
    global simStat
    set simStat [simRun 1]
}

proc simGo {} {
    global simGoOK simDelay simStat
    set simGoOK 1
    # Disable the Go and Step buttons
    # Enable the Stop button
    while {$simGoOK} {
        # run the simulator 1 cycle
        after $simDelay
        set simStat [simRun 1]
        if {$simStat != "AOK" && $simStat != "BUB"} {set simGoOK 0}
        update
    }
    # Disable the Stop button
```

```
    # Enable the Go and Step buttons
}

################################################################################
#  Processor State display                                                     #
################################################################################

# Overall width of pipe register display
set procWidth 40
set procHeight 1
set labWidth 8

# Add labeled display to window
proc addDisp {win width name} {
    global dpyFont labFont
    set lname [string tolower $name]
    frame $win.$lname
    pack $win.$lname -in $win -side left
    label $win.$lname.t -text $name -font $labFont
    label $win.$lname.c -width $width -font $dpyFont -bg white -relief ridge
    pack $win.$lname.t $win.$lname.c -in $win.$lname -side top
    return [list $win.$lname.c]
}

# Set text in display row
proc setDisp {wins txts} {
    for {set i 0} {$i < [llength $wins] && $i < [llength $txts]} {incr i} {
        set win [lindex $wins $i]
        set txt [lindex $txts $i]
        $win config -text $txt
    }
}

frame .p -width $procWidth
pack .p -in . -side bottom
label .p.lab -text "Processor State" -font $bigLabFont -height $sectionHeight
pack .p.lab -in .p -side top
label .p.pc -text "PC Update Stage" -height $procHeight -font $bigLabFont -width $proc
Width -bg NavyBlue -fg White
#label .p.wb -text "Writeback Stage" -height $procHeight -font $bigLabFont -width $pro
cWidth -bg NavyBlue -fg White
label .p.mem -text "Memory Stage" -height $procHeight -font $bigLabFont -width $procWi
dth -bg NavyBlue -fg White
label .p.ex -text "Execute Stage" -height $procHeight -font $bigLabFont -width $procWi
dth -bg NavyBlue -fg White
label .p.id -text "Decode Stage" -height $procHeight -font $bigLabFont -width $procWid
th -bg NavyBlue -fg White
label .p.if -text "Fetch Stage" -height $procHeight -font $bigLabFont -width $procWidt
h -bg NavyBlue -fg White
# New PC
frame .p.npc
# Mem
frame .p.m
# Execute
frame .p.e
# Decode
frame .p.d
# Fetch
frame .p.f
# Old PC
frame .p.opc
pack .p.npc .p.pc .p.m .p.mem .p.e .p.ex .p.d .p.id .p.f .p.if .p.opc -in .p -side top
 -anchor w -expand 1

# Take list of lists, and transpose nesting
# Assumes all lists are of same length
proc ltranspose {inlist} {
    set result {}
    for {set i 0} {$i < [llength [lindex $inlist 0]]} {incr i} {
        set nlist {}
        for {set j 0} {$j < [llength $inlist]} {incr j} {
            set ele [lindex [lindex $inlist $j] $i]
            set nlist [concat $nlist [list $ele]]
        }
        set result [concat $result [list $nlist]]
    }
    return $result
}

# Fields in PC displayed
# Total size = 8
set pwins(OPC) [ltranspose [list [addDisp .p.opc 8 PC]]]

# Fetch display
# Total size = 6+8+4+4+8 = 30
set pwins(F) [ltranspose \
           [list [addDisp .p.f 6 Instr] \
                 [addDisp .p.f 4 rA]\
                 [addDisp .p.f 4 rB] \
                 [addDisp .p.f 8 valC] \
                 [addDisp .p.f 8 valP]]]

# Decode Display
# Total size = 4+8+4+8+4+4 = 32
set pwins(D) [ltranspose \
           [list \
                 [addDisp .p.d 8 valA] \
                 [addDisp .p.d 8 valB] \
                 [addDisp .p.d 4 dstE] \
                 [addDisp .p.d 4 dstM] \
                 [addDisp .p.d 4 srcA] \
                 [addDisp .p.d 4 srcB]]]



# Execute Display
# Total size = 3+8 = 11
set pwins(E) [ltranspose \
           [list [addDisp .p.e 3 Cnd] \
                 [addDisp .p.e 8 valE]]]

# Memory Display
# Total size = 8
set pwins(M) [ltranspose \
           [list [addDisp .p.m 8 valM]]]

# New PC Display
# Total Size = 8
set pwins(NPC) [ltranspose \
           [list [addDisp .p.npc 8 newPC]]]
```

```
# update status line for specified proc register
proc updateStage {name txts} {
    set Name [string toupper $name]
    global pwins
    set wins [lindex $pwins($Name) 0]
    setDisp $wins $txts
}

################################################################################
#                    Instruction Display                                       #
################################################################################

toplevel .c
wm title .c "Program Code"
frame .c.cntl
pack .c.cntl -in .c -side top -anchor w
label .c.filelab -width 10 -text "File"
entry .c.filename -width 20 -relief sunken -textvariable codeFile \
        -font $dpyFont -bg white
button .c.loadbutton -width $cntlBW -command {loadCode $codeFile} -text Load
pack .c.filelab .c.filename .c.loadbutton -in .c.cntl -side left

proc clearCode {} {
    simLabel {} {}
    destroy .c.t
    destroy .c.tr
}

proc createCode {} {
    # Create Code Structure
    frame .c.t
    pack .c.t -in .c -side top -anchor w
    frame .c.tr
    pack .c.tr -in .c.t -side top -anchor nw
}

proc loadCode {file} {
    # Kill old code window
    clearCode
    # Create new one
    createCode
    simCode $file
    simResetAll
}

# Start with initial code window, even though it will be destroyed.
createCode

# Add a line of code to the display
proc addCodeLine {line addr op text} {
    global codeRowCount
    # Create new line in display
    global codeFont
    frame .c.tr.$addr
    pack .c.tr.$addr -in .c.tr -side top -anchor w
    label .c.tr.$addr.a -width 5 -text [format "0x%x" $addr] -font $codeFont
    label .c.tr.$addr.i -width 12 -text $op -font $codeFont
    label .c.tr.$addr.s -width 2 -text "" -font $codeFont -bg white
    label .c.tr.$addr.t -text $text -font $codeFont
    pack .c.tr.$addr.a .c.tr.$addr.i .c.tr.$addr.s \
            .c.tr.$addr.t -in .c.tr.$addr -side left
}

# Keep track of which instructions have stage labels

set oldAddr {}

proc simLabel {addrs labs} {
    global oldAddr
    set newAddr {}
    # Clear away any old labels
    foreach a $oldAddr {
        .c.tr.$a.s config -text ""
    }
    for {set i 0} {$i < [llength $addrs]} {incr i} {
        set a [lindex $addrs $i]
        set t [lindex $labs $i]
        if {[winfo exists .c.tr.$a]} {
            .c.tr.$a.s config -text $t
            set newAddr [concat $newAddr $a]
        }
    }
    set oldAddr $newAddr
}

proc simResetAll {} {
    global simStat
    set simStat "AOK"
    simReset
    simLabel {} {}
    clearMem
}

################################################################################
#    Memory Display                                                            #
################################################################################
toplevel .m
wm title .m "Memory Contents"
frame .m.t
pack .m.t -in .m -side top -anchor w

label .m.t.lab -width 6 -font $dpyFont -text "      "
pack .m.t.lab -in .m.t -side left
for {set i 0} {$i < 16} {incr i 4} {
    label .m.t.a$i -width 8 -font $dpyFont -text [format "  0x---%x" [expr $i % 16]]
    pack .m.t.a$i -in .m.t -side left
}


# Keep track of range of addresses currently displayed
set minAddr 0
set memCnt  0
set haveMem 0

proc createMem {nminAddr nmemCnt} {
    global minAddr memCnt haveMem codeFont dpyFont normalBg
    set minAddr $nminAddr
    set memCnt $nmemCnt

    if { $haveMem } { destroy .m.e }

    # Create Memory Structure
```

```
    frame .m.e
    set haveMem 1
    pack .m.e -in .m -side top -anchor w
    # Now fill it with values
    for {set i 0} {$i < $memCnt} {incr i 16} {
        set addr [expr $minAddr + $i]

        frame .m.e.r$i
        pack .m.e.r$i -side bottom -in .m.e
        label .m.e.r$i.lab -width 6 -font $dpyFont -text [format "0x%.3x-"  [expr $add
r / 16]]
        pack .m.e.r$i.lab -in .m.e.r$i -side left

        for {set j 0} {$j < 16} {incr j 4} {
            set a [expr $addr + $j]
            label .m.e.v$a -width 8 -font $dpyFont -relief ridge \
                -bg $normalBg
            pack .m.e.v$a -in .m.e.r$i -side left
        }
    }
}

proc setMem {Addr Val} {
    global minAddr memCnt
    if {$Addr < $minAddr || $Addr > [expr $minAddr + $memCnt]} {
        error "Memory address $Addr out of range"
    }
    .m.e.v$Addr config -text [format %8x $Val]
}

proc clearMem {} {
    destroy .m.e
    createMem 0 0
}



################################################################################
#    Command Line Initialization                                               #
################################################################################

# Get code file name from input

# Find file with specified extension
proc findFile {tlist ext} {
    foreach t $tlist {
        if {[string match "*.$ext" $t]} {return $t}
    }
    return ""
}


set codeFile [findFile $argv yo]
if {$codeFile != ""} { loadCode $codeFile}
```