ECE 486 Instruction Set Architecture Simulator

By: Ryan Bornhorst, Travis Hermant, Abdullah Barghouti, Ammar Khan, Ali Boshehri

Table of Contents:

Introduction **3**

Internal Design Documentation **4**

Assumptions and Design Decisions **5**

Addressing Modes **6**

Addressing Modes and their description 7

Program Counter Addressing 7

Testing **9**

Conclusion **9**

Appendix **10**

main.c 10

Pdp.c 13

pdp.h 30

opcodes.h 33

Example Assembly File For Testing 35

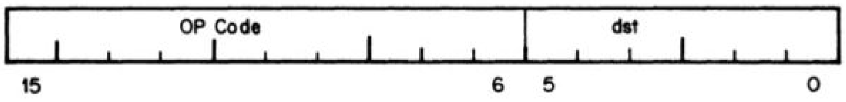
Output Trace 35

Introduction

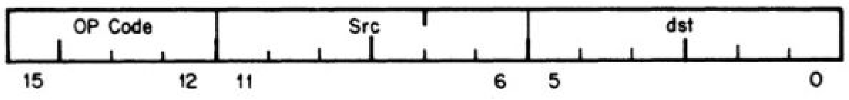
The purpose of our project is to simulate the PDP-11 Instruction Set Architecture by parsing op-codes of various instructions. Instructions are divided into groups consisting of single operands, double operands, register source/destination, and branch instructions. Each instruction is represented in 16-bits. The PDP-11 Simulator is written in the preferred language of the students, and in our case this was C. This simulation is done in order to improve our understanding of the PDP-11 architecture.

4.2 INSTRUCTION FORMATS
The major instruction formats are:

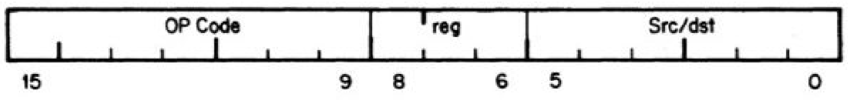
Single Operand Group



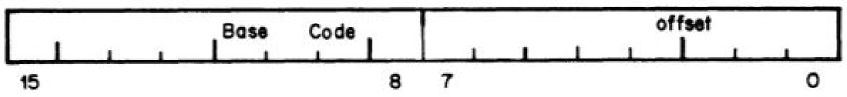
Double Operand Group



Register-Source or Destination



Branch



Internal Design Documentation

PDP-11 is a 16-bit design and uses octal for its memory addresses rather than the more conventional hexadecimal. Test conditions cover all PDP-11 instructions and addressing modes with the exception of instructions in the following groups: trap & interrupt, miscellaneous, and condition instructions. It must accommodate as much as 32K words.

Testing will consist of writing short chunks of PDP-11 code and running them through two provided pieces of software. The Macro-11 assembler and the obj2ascii translator will convert the code into its ascii representation, this is what we will use to analyze with our simulator. Each line generated by the obj2ascii translator represents a 16-bit octal value. Any line that begins with an @ indicates a change to the current load address.

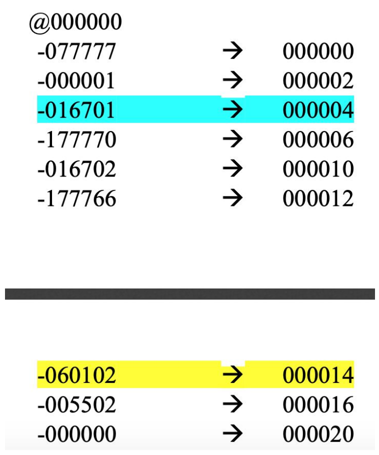
The simulator should produce the following outputs:

- A memory trace file like the one used in 485 that shows each instruction fetch and memory data read and write
- A summary that includes the total number of instructions executed displayed on the screen

Assumptions and Design Decisions

Every instruction has a distinct series of Op-Codes for each instruction, so to check which instruction we're looking at we created a mask for each instruction type. Our masks are split into two major groups; instructions with leading 1's such as byte versions of single operand instructions and some branch instruction, and the remaining instructions with leading 0's.

The trace file displays both the type and address of each instruction. The type is dependent on if the instruction is a data read, data write, or instruction fetch. The address is the relative PC location for that specific instruction. Depending on the specific mode and instruction, the PDP-11 simulator will handle it differently. For example, during the relative addressing mode the instruction such as MOV A, R1, will be performed in a particular way. In the diagram on the right, note that at PC = 000000, the address of the A is stored, and at PC = 000004,the MOV instruction is performed. The address location for the value of A is given by 177770. To calculate the actual address of A, 177770 must be added to PC+2 value which is in this case 000010. Once we add 177770 and 000010 gives us 200000, which is an 18 bit value. We need to discard the 17th and 18th bit, which yields 000000, and that is where A is stored.

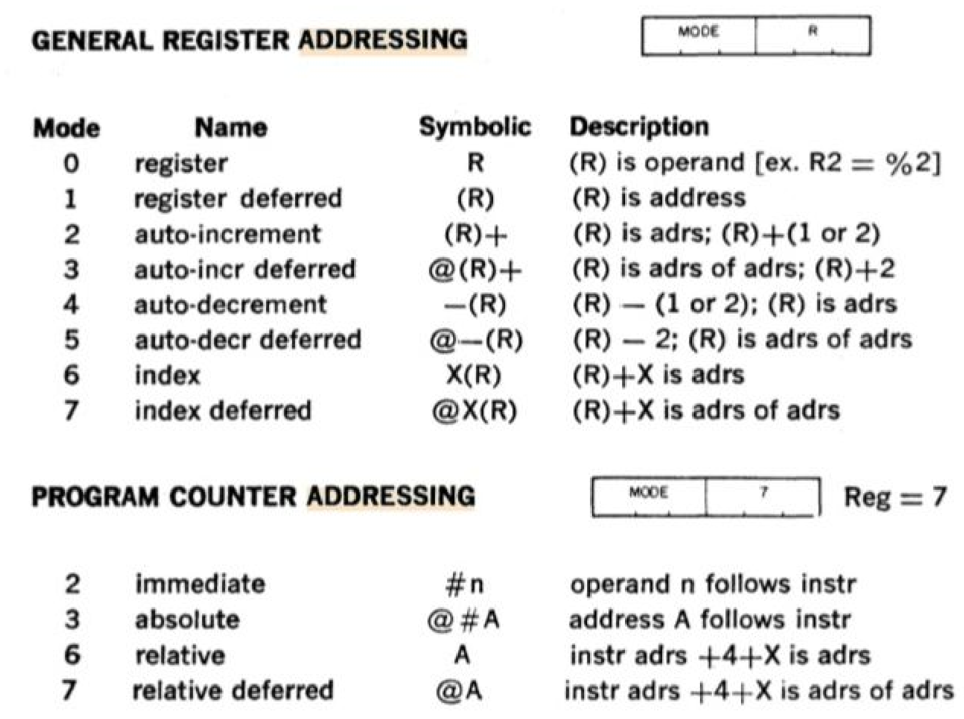


```
@000000
-077777      →    000000
-000001      →    000002
-016701      →    000004
-177770      →    000006
-016702      →    000010
-177766      →    000012

-060102      →    000014
-005502      →    000016
-000000      →    000020
```

The bulk of the addressing modes work in this fashion with a few addressing modes that vary. Such addressing modes include the Immediate addressing mode, Index deferred addressing mode, Auto-increment deferred addressing mode, auto-increment addressing mode, and the Relative deferred addressing mode.

The modes for both the destinations and sources were filled by using the following table.



GENERAL REGISTER ADDRESSING

MODE	R

Mode	Name	Symbolic	Description
0	register	R	(R) is operand [ex. R2 = %2]
1	register deferred	(R)	(R) is address
2	auto-increment	(R)+	(R) is adrs; (R)+(1 or 2)
3	auto-incr deferred	@(R)+	(R) is adrs of adrs; (R)+2
4	auto-decrement	−(R)	(R) − (1 or 2); (R) is adrs
5	auto-decr deferred	@−(R)	(R) − 2; (R) is adrs of adrs
6	index	X(R)	(R)+X is adrs
7	index deferred	@X(R)	(R)+X is adrs of adrs

PROGRAM COUNTER ADDRESSING

MODE	7

Reg = 7

2	immediate	#n	operand n follows instr
3	absolute	@#A	address A follows instr
6	relative	A	instr adrs +4+X is adrs
7	relative deferred	@A	instr adrs +4+X is adrs of adrs

Addressing Modes

Depending on the type of operand, the number of bits allocated for selecting the addressing mode and selecting a general register can vary.

For single operand: three bits are allocated for selecting the address modes and another three register are allocated for selecting one of the general registers. Totaling 6 bits that we identify as the destination field.

For double operand three bits are allocated for selecting the address modes and another three register are allocated for selecting one of the general registers for the destination field, and another set of three bits for address modes and three for general register for the source field.

Addressing Modes and their description

Register: Used for fast instruction execution when no memory is used. Ideal for register to register operations.

Register Deferred: The address of the desired operand is stored in one of the general registers. That address then points the CPU to the actual operand (the operand is not located on the CPU)

Auto-increment: A register contains the address of the operand, once the operand returns, the address is incremented so that the register will contain next location/address. (depending on the type of instruction, the increment value will be different. For byte we increment by 1, for word we increment by 2)

Auto- increment Deferred: A register contains a pointer to the address of the operand. Once the address is located, the pointer increments by two regardless of the type of instruction (byte or word). This is only useful when we access operands in consecutive locations.

Auto-decrement: Similar to the functionality of auto-increment but it moves in reverse order.

Auto decrement Deferred: A register consists of a pointer to the address of the operand. The pointer is decremented by two regardless of the type of instruction (byte or word), now the newly altered pointer is used to store an address (out of the CPU, in memory)

Index: The index address is added to the base address to give the address of the operand.
Index Deferred: The index address is added to the base address to give the a pointer that points to the address of the operand.

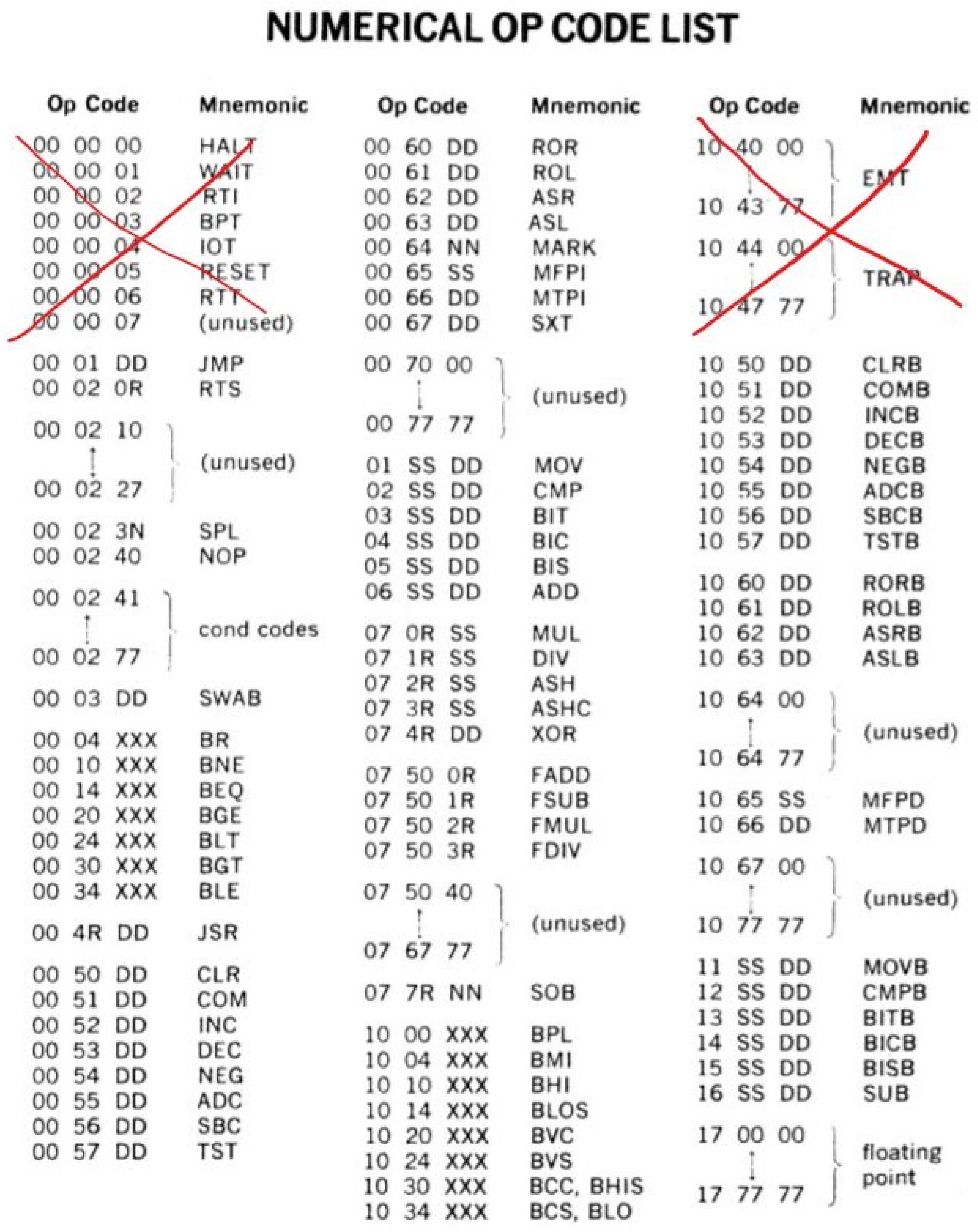
Program Counter Addressing

Immediate: This mode is equivalent to using the auto-increment mode, and this mode provides time improvements for accessing constant operands by including the constant in the memory location immediately following the instruction word.

Absolute: This mode is similar to the auto-increment mode. For this mode the contents of the location following the instruction are taken as the address of the operand. Immediate data are interpreted as absolute address.

Relative: This mode is index mode 6 using the PC. The operand's address is calculated by adding the offset that follows the instruction to the updated contents of the PC. The offset that follows the instruction is directed by PC+2 and this is summed with PC+4, which yields the effective address.

Relative Deferred: The offset that follows the instruction holds the address that points to the address that needs to be accessed by this addressing mode. Address offset + (PC + 4) holds the address that is being accessed.



NUMERICAL OP CODE LIST

Op Code	Mnemonic
~~00 00 00~~	~~HALT~~
~~00 00 01~~	~~WAIT~~
~~00 00 02~~	~~RTI~~
~~00 00 03~~	~~BPT~~
~~00 00 04~~	~~IOT~~
~~00 00 05~~	~~RESET~~
~~00 00 06~~	~~RTT~~
~~00 00 07~~	~~(unused)~~
00 01 DD	JMP
00 02 0R	RTS
00 02 10 ↕ 00 02 27	(unused)
00 02 3N	SPL
00 02 40	NOP
00 02 41 ↕ 00 02 77	cond codes
00 03 DD	SWAB
00 04 XXX	BR
00 10 XXX	BNE
00 14 XXX	BEQ
00 20 XXX	BGE
00 24 XXX	BLT
00 30 XXX	BGT
00 34 XXX	BLE
00 4R DD	JSR
00 50 DD	CLR
00 51 DD	COM
00 52 DD	INC
00 53 DD	DEC
00 54 DD	NEG
00 55 DD	ADC
00 56 DD	SBC
00 57 DD	TST
00 60 DD	ROR
00 61 DD	ROL
00 62 DD	ASR
00 63 DD	ASL
00 64 NN	MARK
00 65 SS	MFPI
00 66 DD	MTPI
00 67 DD	SXT
00 70 00 ↕ 00 77 77	(unused)
01 SS DD	MOV
02 SS DD	CMP
03 SS DD	BIT
04 SS DD	BIC
05 SS DD	BIS
06 SS DD	ADD
07 0R SS	MUL
07 1R SS	DIV
07 2R SS	ASH
07 3R SS	ASHC
07 4R DD	XOR
07 50 0R	FADD
07 50 1R	FSUB
07 50 2R	FMUL
07 50 3R	FDIV
07 50 40 ↕ 07 67 77	(unused)
07 7R NN	SOB
10 00 XXX	BPL
10 04 XXX	BMI
10 10 XXX	BHI
10 14 XXX	BLOS
10 20 XXX	BVC
10 24 XXX	BVS
10 30 XXX	BCC, BHIS
10 34 XXX	BCS, BLO
~~10 40 00 ↕ 10 43 77~~	~~EMT~~
~~10 44 00 ↕ 10 47 77~~	~~TRAP~~
10 50 DD	CLRB
10 51 DD	COMB
10 52 DD	INCB
10 53 DD	DECB
10 54 DD	NEGB
10 55 DD	ADCB
10 56 DD	SBCB
10 57 DD	TSTB
10 60 DD	RORB
10 61 DD	ROLB
10 62 DD	ASRB
10 63 DD	ASLB
10 64 00 ↕ 10 64 77	(unused)
10 65 SS	MFPD
10 66 DD	MTPD
10 67 00 ↕ 10 77 77	(unused)
11 SS DD	MOVB
12 SS DD	CMPB
13 SS DD	BITB
14 SS DD	BICB
15 SS DD	BISB
16 SS DD	SUB
17 00 00 ↕ 17 77 77	floating point

Testing

Our testing consisted of writing multiple files and doing an exhaustive run through of every instruction type. Following that we checked for each of the different register addressing modes to make sure they were handled correctly. In addition to the trace file output we also have an option for printing to the console in a debug fashion, separating each instruction by being either single operand, double operand, or program data.

The main instructions we focused on testing included the single operand, double operand, and branch instructions. These instructions were written in PDP11 assembly. Each of these instructions were tested first in register mode and then the syntax of the PDP11 assembly file was changed in order to be tested in the other addressing modes such as auto-decrement mode and index mode.

One of the issues we faced, is that we did not test for being able to read and obj2ascii file that starts with a *, and as a result the simulator did not interpret certain files during the demo correctly. However, we were not aware that this was a requirement based on the project description.

Conclusion

After completing this project we now have a much better understanding of how the PDP-11 set architecture works. Going into the project, we were vaguely familiar with the functionality of the different addressing modes in the architecture. Furthermore, our understanding of the various instructions was also very limited. However, after we started writing the code for the project, our understanding of the addressing modes and instructions both drastically improved.

Even though PDP-11 is an older and simplistic architecture compared to more modern ISAs, we can appreciate the difficulty that went into the design. This project really helped us learn about all of the details that the microarchitecture has to deal with to even process what appears to be a simple instruction set. Developing a simulator for a modern 64 bit architecture like x86 would be incredibly difficult to implement.

Lastly, we learned that a lot of our simulation design could have been more efficient. We were writing the code as we were learning how PDP-11 functions so some of the functionality of the simulator was implemented inefficiently. If we could restart the project again, we think that it would be much easier and much less code to create an even better simulator.

Appendix

main.c

```
#include "pdp.h"

/**********

PDP-11 SIMULATOR

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

/***** Vars *****/
int                 n_lines;            // # lines read from file
int             starting_instr;         // first program instruction
char            *cmd;                   // command line instruction
uint16_t      start_addr;               // starting address of instruction
unsigned long   addr_temp;              // temp addr used for char/long conversion
char             fn     [BUFF_SIZE];    // filename
char            *line   [LINE_SIZE];    // lines as string
unsigned long       oct_num   [LINE_SIZE];      // lines as u_long
uint16_t   oct16    [LINE_SIZE];        // lines as u_16
var_data   data     [LINE_SIZE];        // data that gets stored before instructions
uint16_t   memory   [LINE_SIZE];        // memory addresses
instr_single   s_instr [LINE_SIZE];     // single operand structs
instr_double   d_instr [LINE_SIZE];     // double operand structs
int                 n_data;                     // # of stored data values
int                 n_single, n_double; // # of single/double operand instrs
uint16_t      PC    [LINE_SIZE];        // program counter
sim_output          sim_o     [LINE_SIZE];      // simulator output
int                 n_sim;                      // # sim outputs
int                 ret;                        // return value
FILE                *fp;                        // file pointer for writing out simulator results

/***** Initialize Variables *****/
n_lines   = 0;
starting_instr      = 0;
cmd                 = NULL;
start_addr          = 0177777;
addr_temp           = 0x0;
n_single  = 0;
n_double  = 0;
n_data              = 0;
n_sim               = 0;
ret                 = ERROR_NONE;

/***** Command Line Args *****/
if(argc == 2)
        cmd = argv[1];
else if(argc == 3) {
        cmd = argv[1];
        if(argv[2][0] == '*'){
```

```
		addr_temp		= strtoul((&argv[2][1]), NULL, 8);
				start_addr= addr_temp;
		}
	} else{
		printf("Generate Ascii: ./pdp obj2ascii\n"
		      "Run Simulator:  ./pdp <filename> "
		      "*<address of first instruction>\n");
		ret = ERROR;
		return ret;
	}

	/***** Run Simulator *****/
	if(!strcmp(cmd, "obj2ascii")){
		ret = obj2ascii(argv[2]);
		return ret;
	} else if(start_addr != 0177777){

		/***** File I/O *****/
		snprintf(fn, BUFF_SIZE, "ascii/%s.ascii", cmd);
		ret = rd_ascii_file(fn, line, &n_lines);
		if(ret == ERROR){
			printf("File I/O Error. Check filename.\n");
			return ret;
		}

		/***** String -> Octal *****/
		ret = str_to_oct(line, oct_num, oct16, n_lines, PC, start_addr,
					&starting_instr, data, &n_data);
		if(ret == ERROR){
			printf("Error Converting File String to Octal.\n");
			return ret;
		}

		/***** Find Valid Instructions *****/
		ret = get_instruction(oct16, s_instr, d_instr, n_lines, &n_single,
					  &n_double, starting_instr, PC);
		if(ret == ERROR){
			printf("Error Obtaining Valid Instructions.\n");
			return ret;
		}

		/***** Get Data PC and Memory Address *****/
		ret = data_mem_addr(oct16, data, PC, n_lines, n_data);

		/***** Store Operand Data *****/
		ret = store_reg_vals(oct16, s_instr, d_instr, PC, n_lines, n_single, n_double);
		if(ret == ERROR){
			printf("Error Storing Operand Data.\n");
			return ret;
		}

		/***** Determine Simulator Values *****/
		ret = fetch_instructions(oct16, s_instr, d_instr, PC, data, sim_o,
						n_lines, n_single, n_double, n_data, &n_sim);
```

```
}else
		printf("Invalid Command Line Arguments.\n");

/***** Print Statements for Debugging *****/
#if DEBUG || AMA
printf("\nSTRING-----\n");
for(int i = 0; i < n_lines+1; i++){
		printf("%s\n", line[i]);
}
printf("\nOCTAL-----PC\n");
for(int i = 0; i < n_lines; i++){
		printf("%06o    %03o\n", oct16[i], PC[i]);
}
#endif
#if DEBUG && !AMA
printf("\nPROGRAM_DATA\n");
for(int i = 0; i < n_data; i++) {
		printf("%06o %o ", data[i].data, data[i].PC);
		for(int j = 0; j < data[i].n_memory; j++) {
				printf("%06o ", data[i].memory[j]);
		}
		printf("\n");
}
printf("\nSINGLE_OPERAND_INSTRUCTIONS\n");
for(int i = 0; i < n_single; i++) {
		printf("%-5s %03o %01o %01o %06o   %o\n", s_instr[i].instr, s_instr[i].opcode,
				s_instr[i].mode_dd, s_instr[i].dd, s_instr[i].dd_reg, s_instr[i].PC);
}
printf("\nDOUBLE_OPERAND_INSTRUCTIONS\n");
for(int i = 0; i < n_double; i++) {
		printf("%-5s %01o %01o %01o %01o %01o %06o %06o   %o\n", d_instr[i].instr,
				d_instr[i].opcode, d_instr[i].mode_ss, d_instr[i].ss,
				d_instr[i].mode_dd, d_instr[i].dd, d_instr[i].ss_reg,
				d_instr[i].dd_reg, d_instr[i].PC);
}
#endif
#if !AMA
fclose(fopen("trace.txt", "w"));
fp = fopen("trace.txt", "a");
if(!fp)
		printf("File not working for writing to\n");

printf("\nSIMULATOR RESULTS\n");
printf("Total Instructions:       %d\n", n_double + n_single);
for(int i = 0; i < n_sim; i++) {
		printf("%d %06o\n", sim_o[i].type, sim_o[i].addr);
		fprintf(fp, "%d %06o\n", sim_o[i].type, sim_o[i].addr);
}
fclose(fp);
printf("\n");
#endif

/***** Free Memory *****/
for(int i = 0; i < n_lines; i++){
		free(line[i]); }return ret;
}
```

Pdp.c

```
#include "pdp.h"

/**********

Get Register Values

**********/
int get_reg(uint16_t *reg_val, uint16_t reg_get){

        if(reg_get == 00)
                *reg_val = R0;
        else if(reg_get == 01)
                *reg_val = R1;
        else if(reg_get == 02)
                *reg_val = R2;
        else if(reg_get == 03)
                *reg_val = R3;
        else if(reg_get == 04)
                *reg_val = R4;
        else if(reg_get == 05)
                *reg_val = R5;
        else if(reg_get == 06)
                *reg_val = R6;
        else if(reg_get == 07)
                *reg_val = R7;

        return 0;
}

/**********

Set Register Values

**********/
int set_reg(uint16_t reg_val, uint16_t reg_set){

        if(reg_set == 00)
                R0 = reg_val;
        else if(reg_set == 01)
                R1 = reg_val;
        else if(reg_set == 02)
                R2 = reg_val;
        else if(reg_set == 03)
                R3 = reg_val;
        else if(reg_set == 04)
                R4 = reg_val;
        else if(reg_set == 05)
                R5 = reg_val;
        else if(reg_set == 06)
                R6 = reg_val;
        else if(reg_set == 07)
                R7 = reg_val;
```

```
return 0;

}

/**********

Store Simulation Results

**********/
int fetch_instructions(uint16_t *oct, instr_single *s, instr_double *d,
                        uint16_t *PC, var_data *data, sim_output *sim,
                        int n_lines, int n_single, int n_double, int n_data,
                        int *n_sim) {

uint16_t temp_reg_val = 00;

int ret = ERROR_NONE;

int j = 0;
int k = 0;

/***** Look through all ascii values *****/
for(int i = n_data; i < n_lines; i++) {
        /***** Check against single/double instructions *****/
        for(j = 0, k = 0; (j < n_single) || (k < n_double); j++,k++) {
                /***** If single op PC is the same as ascii PC, instruction found *****/
                if(PC[i] == s[j].PC) {
                        sim[*n_sim].type = 2;
                        sim[*n_sim].addr = s[j].PC;
                        ++(*n_sim);
                        /***** Check to see if fetch accesses memory *****/
                        if(s[j].dd == 07) {
                                if(s[j].mode_dd == 02){
                                        get_reg(&temp_reg_val, s[j].dd);
                                        set_reg(temp_reg_val + oct[s[j].PC + 02], s[j].dd);
                                        sim[*n_sim].type = 0;
                                        sim[*n_sim].addr = s[j].PC + 02;
                                        ++(*n_sim);
                                        set_reg(s[j].PC + 04, s[j].dd);
                                }else if(s[j].mode_dd == 03){
                                        sim[*n_sim].type = 0;
                                        sim[*n_sim].addr = s[j].PC + 02;
                                        ++(*n_sim);
                                        sim[*n_sim].type = 1;
                                        sim[*n_sim].addr = s[j].dd_reg;
                                        ++(*n_sim);
                                        set_reg(s[j].PC + 04, s[j].dd);
                                }else if(s[j].mode_dd == 06){
                                        sim[*n_sim].type = 0;
                                        sim[*n_sim].addr = s[j].PC + 02;
                                        ++(*n_sim);
                                        sim[*n_sim].type = 1;
                                        sim[*n_sim].addr = s[j].dd_reg + (s[j].PC + 04);
                                        ++(*n_sim);
                                        set_reg(s[j].PC + 04, s[j].dd);
                                }else if(s[j].mode_dd == 07){
                                        sim[*n_sim].type = 0;
                                        sim[*n_sim].addr = s[j].PC + 02;
                                        ++(*n_sim);
                                        sim[*n_sim].type = 0;
                                        sim[*n_sim].addr = s[j].dd_reg + (s[j].PC + 04);
                                        ++(*n_sim);
                                        sim[*n_sim].type = 1;
                                        sim[*n_sim].addr = oct[s[j].dd_reg + (s[j].PC + 04)];
                                        ++(*n_sim);
                                        set_reg(s[j].PC + 04, s[j].dd);
                                }
                        }else if(s[j].mode_dd == 02){
                                get_reg(&temp_reg_val, s[j].dd);
                                set_reg(temp_reg_val + 02, s[j].dd);
                                sim[*n_sim].type = 0;
                                sim[*n_sim].addr = temp_reg_val + 02;
                                ++(*n_sim);
                        }else if(s[j].mode_dd == 03){
                                get_reg(&temp_reg_val, s[j].dd);
                                set_reg(temp_reg_val + 02, s[j].dd);
                                sim[*n_sim].type = 0;
                                sim[*n_sim].addr = oct[temp_reg_val + 02];
                                ++(*n_sim);
                        }else if(s[j].mode_dd == 04){
                                get_reg(&temp_reg_val, s[j].dd);
                                set_reg(temp_reg_val - 02, s[j].dd);
                                sim[*n_sim].type = 0;
                                sim[*n_sim].addr = temp_reg_val - 02;
                                ++(*n_sim);
                        }else if(s[j].mode_dd == 05){
                                get_reg(&temp_reg_val, s[j].dd);
                                set_reg(temp_reg_val - 02, s[j].dd);
                                sim[*n_sim].type = 0;
                                sim[*n_sim].addr = oct[temp_reg_val - 02];
                                ++(*n_sim);
                        }else if(s[j].mode_dd == 06){
                                get_reg(&temp_reg_val, s[j].dd);
                                set_reg(temp_reg_val + s[j].PC + 04, s[j].dd);
                                sim[*n_sim].type = 0;
                                sim[*n_sim].addr = temp_reg_val + s[j].PC + 04;
                                ++(*n_sim);
                                sim[*n_sim].type = 1;
                                sim[*n_sim].addr = temp_reg_val + s[j].PC + 04;
                                ++(*n_sim);
                        }else if(s[j].mode_dd == 07){
                                get_reg(&temp_reg_val, s[j].dd);
                                set_reg(temp_reg_val + s[j].PC + 04, s[j].dd);
                                sim[*n_sim].type = 0;
                                sim[*n_sim].addr = temp_reg_val + s[j].PC + 04;
                                ++(*n_sim);
                                sim[*n_sim].type = 1;
                                sim[*n_sim].addr = oct[temp_reg_val + s[j].PC + 04];
                                ++(*n_sim);
                        }
                        break;
                }
                /***** If double op PC is the same as ascii PC, instruction found *****/
                if(PC[i] == d[k].PC) {
```

```
sim[*n_sim].type = 2;
sim[*n_sim].addr = d[k].PC;
++(*n_sim);
/***** Check Address Mode for Memory Access *****/
if(d[k].ss == 07) {
        if(d[k].mode_ss == 02){
                get_reg(&temp_reg_val, d[k].ss);
                set_reg(temp_reg_val + oct[d[k].PC + 02], d[k].ss);
                sim[*n_sim].type = 0;
                sim[*n_sim].addr = d[k].PC + 02;
                ++(*n_sim);
                set_reg(d[k].PC + 02, d[k].ss);
        }else if(d[k].mode_ss == 03){
                sim[*n_sim].type = 0;
                sim[*n_sim].addr = d[k].PC + 02;
                ++(*n_sim);
                sim[*n_sim].type = 0;
                sim[*n_sim].addr = d[k].ss_reg;
                ++(*n_sim);
                set_reg(d[k].PC + 02, d[k].ss);
        }else if(d[k].mode_ss == 06){
                sim[*n_sim].type = 0;
                sim[*n_sim].addr = d[k].PC + 02;
                ++(*n_sim);
                sim[*n_sim].type = 0;
                sim[*n_sim].addr = d[k].ss_reg + (d[k].PC + 04);
                ++(*n_sim);
                set_reg(d[k].ss_reg + d[k].PC + 04, d[k].ss);
        }else if(d[k].mode_ss == 07){
                sim[*n_sim].type = 0;
                sim[*n_sim].addr = d[k].PC + 02;
                ++(*n_sim);
                sim[*n_sim].type = 0;
                sim[*n_sim].addr = oct[d[k].PC + 02];
                set_reg(oct[d[k].PC + 02], d[k].ss);
        }
}else if(d[k].mode_ss == 01){
        get_reg(&temp_reg_val, d[k].ss);
        set_reg(temp_reg_val, d[k].ss);
        sim[*n_sim].type = 0;
        sim[*n_sim].addr = temp_reg_val;
        ++(*n_sim);
}else if(d[k].mode_ss == 02){
        get_reg(&temp_reg_val, d[k].ss);
        set_reg(temp_reg_val + 02, d[k].ss);
        sim[*n_sim].type = 0;
        sim[*n_sim].addr = temp_reg_val + 02;
        ++(*n_sim);
}else if(d[k].mode_ss == 03){
        get_reg(&temp_reg_val, d[k].ss);
        set_reg(temp_reg_val + 02, d[k].ss);
        sim[*n_sim].type = 0;
        sim[*n_sim].addr = oct[temp_reg_val + 02];
        ++(*n_sim);
}else if(d[k].mode_ss == 04){
        get_reg(&temp_reg_val, d[k].ss);
        set_reg(temp_reg_val - 02, d[k].ss);
```

```
		sim[*n_sim].type = 0;
		sim[*n_sim].addr = temp_reg_val - 02;
		++(*n_sim);
}else if(d[k].mode_ss == 05){
		get_reg(&temp_reg_val, d[k].ss);
		set_reg(temp_reg_val - 02, d[k].ss);
		sim[*n_sim].type = 0;
		sim[*n_sim].addr = oct[temp_reg_val - 02];
		++(*n_sim);
}else if(d[k].mode_ss == 06){
		get_reg(&temp_reg_val, d[k].ss);
		set_reg(temp_reg_val + oct[d[k].PC + 02], d[k].ss);
		sim[*n_sim].type = 0;
		sim[*n_sim].addr = temp_reg_val + oct[d[k].PC + 02];
		++(*n_sim);
}else if(d[k].mode_ss == 07){
		get_reg(&temp_reg_val, d[k].ss);
		set_reg(oct[temp_reg_val + oct[d[k].PC + 02]], d[k].ss);
		sim[*n_sim].type = 0;
		sim[*n_sim].addr = oct[temp_reg_val + oct[d[k].PC + 02]];
		++(*n_sim);
}
/***** Check Address Mode for Memory Access *****/
if(d[k].dd == 07){
		if(d[k].mode_dd == 02){
				get_reg(&temp_reg_val, d[k].dd);
				set_reg(temp_reg_val + oct[d[k].PC + 04], d[k].dd);
				sim[*n_sim].type = 0;
				sim[*n_sim].addr = d[k].PC + 04;
				++(*n_sim);
				set_reg(d[k].PC + 04, d[k].dd);
		}else if(d[k].mode_dd == 03){
				sim[*n_sim].type = 0;
				sim[*n_sim].addr = d[k].PC + 04;
				++(*n_sim);
				sim[*n_sim].type = 1;
				sim[*n_sim].addr = d[k].dd_reg;
				++(*n_sim);
				set_reg(d[k].PC + 04, d[k].dd);
		}else if(d[k].mode_dd == 06){
				sim[*n_sim].type = 0;
				sim[*n_sim].addr = d[k].PC + 04;
				++(*n_sim);
				sim[*n_sim].type = 1;
				sim[*n_sim].addr = d[k].dd_reg + (d[k].PC + 06);
				++(*n_sim);
				set_reg(d[k].dd_reg + d[k].PC + 06, d[k].dd);
		}else if(d[k].mode_dd == 07){
				sim[*n_sim].type = 0;
				sim[*n_sim].addr = d[k].PC + 04;
				++(*n_sim);
				sim[*n_sim].type = 1;
				sim[*n_sim].addr = oct[d[k].PC + 04];
				++(*n_sim);
				set_reg(oct[d[k].PC + 04], d[k].dd);
		}
}else if(d[k].mode_dd == 00){
```

```
                        get_reg(&temp_reg_val, d[k].ss);
                        set_reg(temp_reg_val, d[k].dd);
                }else if(d[k].mode_dd == 01){
                        get_reg(&temp_reg_val, d[k].dd);
                        sim[*n_sim].type = 0;
                        sim[*n_sim].addr = temp_reg_val;
                        ++(*n_sim);
                }else if(d[k].mode_dd == 02){
                        get_reg(&temp_reg_val, d[k].dd);
                        set_reg(temp_reg_val + 02, d[k].dd);
                        sim[*n_sim].type = 0;
                        sim[*n_sim].addr = temp_reg_val + 02;
                        ++(*n_sim);
                }else if(d[k].mode_dd == 03){
                        get_reg(&temp_reg_val, d[k].dd);
                        set_reg(temp_reg_val + 02, d[k].dd);
                        sim[*n_sim].type = 0;
                        sim[*n_sim].addr = oct[temp_reg_val + 02];
                        ++(*n_sim);
                }else if(d[k].mode_dd == 04){
                        get_reg(&temp_reg_val, d[k].dd);
                        set_reg(temp_reg_val - 02, d[k].dd);
                        sim[*n_sim].type = 0;
                        sim[*n_sim].addr = temp_reg_val - 02;
                        ++(*n_sim);
                }else if(d[k].mode_dd == 05){
                        get_reg(&temp_reg_val, d[k].dd);
                        set_reg(temp_reg_val - 02, d[k].dd);
                        sim[*n_sim].type = 0;
                        sim[*n_sim].addr = oct[temp_reg_val - 02];
                        ++(*n_sim);
                }else if(d[k].mode_dd == 06){
                        get_reg(&temp_reg_val, d[k].dd);
                        set_reg(temp_reg_val + d[k].PC + 04, d[k].dd);
                        sim[*n_sim].type = 0;
                        sim[*n_sim].addr = temp_reg_val + d[k].PC + 04;
                        ++(*n_sim);
                        sim[*n_sim].type = 1;
                        sim[*n_sim].addr = temp_reg_val + d[k].PC + 04;
                        ++(*n_sim);
                }else if(d[k].mode_dd == 07){
                        get_reg(&temp_reg_val, d[k].dd);
                        set_reg(temp_reg_val + d[k].PC + 04, d[k].dd);
                        sim[*n_sim].type = 0;
                        sim[*n_sim].addr = temp_reg_val + d[k].PC + 04;
                        ++(*n_sim);
                        sim[*n_sim].type = 1;
                        sim[*n_sim].addr = oct[temp_reg_val + d[k].PC + 04];
                        ++(*n_sim);
                }
                break;
            }
        }
}

return ret;
```

```
}

/**********

Assign both source and destination values

**********/
int store_reg_vals(uint16_t *oct, instr_single *s, instr_double *d,
                        uint16_t *PC, int n_lines, int n_single, int n_double) {

int ret       = ERROR_NONE;

int index   = 0;
int SINGLE          = 0;
int DOUBLE          = 1;

/***** Use Address Modes to Store Operand Register Values *****/
for(int i = 0; i < n_lines; i++) {
        /***** Single Operand Address Modes *****/
        for(int j = 0; j < n_single; j++) {
                if(PC[i] == s[j].PC) {
                        ret = dd_addr_mode(oct[i], i, &index, SINGLE);
                        if(index != 9999)
                                s[j].dd_reg = oct[index];
                        else
                                s[j].dd_reg = 000000;
                }
        }
        /***** Double Operand Address Modes *****/
        for(int j = 0; j < n_double; j++) {
                if(PC[i] == d[j].PC) {
                        ret = dd_addr_mode(oct[i], i, &index, DOUBLE);
                        if(index != 9999)
                                d[j].dd_reg = oct[index];
                        else
                                d[j].dd_reg = 000000;
                        ret = ss_addr_mode(oct[i], i, &index);
                        if(index != 9999)
                                d[j].ss_reg = oct[index];
                        else
                                d[j].ss_reg = 000000;
                }
        }
}

return ret;

}

/**********

Assign source value

**********/
int ss_addr_mode(uint16_t mode, int index, int *index_out) {

/***** Masks for Address Modes *****/
```

```
uint16_t ss_mode_mask         = 0007000;
uint16_t ss_pc_mask           = 0007700;

/***** Check SS Mode *****/
if((mode & ss_pc_mask) == ss_IMM){
        *index_out = index + 1;
}else if((mode & ss_pc_mask) == ss_ABS){
        *index_out = index + 1;
}else if((mode & ss_pc_mask) == ss_REL){
        *index_out = index + 1;
}else if((mode & ss_pc_mask) == ss_REL_DEF){
        *index_out = index + 1;
}else
        *index_out = 9999;

return 0;

}

/**********

Assign destination value

**********/
int dd_addr_mode(uint16_t mode, int index, int *index_out, int type) {

int DOUBLE = 1;

/***** Masks for Address Modes *****/
uint16_t dd_mode_mask         = 0000070;
uint16_t dd_pc_mask           = 0000077;
uint16_t ss_pc_mask           = 0007700;

/***** Check DD Mode *****/
if(type == DOUBLE) {
        if((mode & dd_pc_mask) == dd_IMM){
                if(((mode & ss_pc_mask) == ss_IMM) || ((mode & ss_pc_mask) == ss_ABS) ||
                   ((mode & ss_pc_mask) == ss_REL) || ((mode & ss_pc_mask) == ss_REL_DEF))
                        *index_out = index + 2;
                else
                        *index_out = index + 1;
        }else if((mode & dd_pc_mask) == dd_ABS){
                if(((mode & ss_pc_mask) == ss_IMM) || ((mode & ss_pc_mask) == ss_ABS) ||
                   ((mode & ss_pc_mask) == ss_REL) || ((mode & ss_pc_mask) == ss_REL_DEF))
                        *index_out = index + 2;
                else
                        *index_out = index + 1;
        }else if((mode & dd_pc_mask) == dd_REL){
                if(((mode & ss_pc_mask) == ss_IMM) || ((mode & ss_pc_mask) == ss_ABS) ||
                   ((mode & ss_pc_mask) == ss_REL) || ((mode & ss_pc_mask) == ss_REL_DEF))
                        *index_out = index + 2;
                else
                        *index_out = index + 1;
        }else if((mode & dd_pc_mask) == dd_REL_DEF){
                if(((mode & ss_pc_mask) == ss_IMM) || ((mode & ss_pc_mask) == ss_ABS) ||
                   ((mode & ss_pc_mask) == ss_REL) || ((mode & ss_pc_mask) == ss_REL_DEF))
                        *index_out = index + 2;
                else
                        *index_out = index + 1;
        }else
                *index_out = 9999;
} else {
        if((mode & dd_pc_mask) == dd_IMM){
                *index_out = index + 1;
        }else if((mode & dd_pc_mask) == dd_REL){
                *index_out = index + 1;
        }else
                *index_out = 9999;
}

return 0;

}

/**********

Decode memory access for data variables

***********/
int data_mem_addr(uint16_t *oct, var_data *data, uint16_t *PC, int n_lines,
                  int n_data) {

int ret = ERROR_NONE;

/***** Store memory accesses for each data variable *****/
for(int i = 0; i < n_lines; i++) {
        for(int j = 0; j < n_data; ++j) {
                if(((data[j].PC + MAX_ADDR) - (PC[i] + 02)) == oct[i]) {
                        data[j].memory[data[j].n_memory] = oct[i];
                        ++data[j].n_memory;
                }
        }
}

return ret;

}

/**********

Check if Opcode is Valid

**********/
int valid_opcode(uint16_t opcode, uint16_t mask, int type, char *msg) {

int valid = 0;

/***** Word Instruction Opcodes *****/
if(type) {
        valid = 1;
        if((opcode & mask) == JMP)                                      snprintf(msg, BUFF_SIZE,
"JMP");
        else if((opcode & mask) == SWAB)                        snprintf(msg, BUFF_SIZE, "SWAB");
        else if((opcode & mask) == BR)                                  snprintf(msg, BUFF_SIZE,
"BR");
        else if((opcode & mask) == BNE)                              snprintf(msg, BUFF_SIZE,
"BNE");
        else if((opcode & mask) == BEQ)                              snprintf(msg, BUFF_SIZE,
"BEQ");
        else if((opcode & mask) == BGE)                              snprintf(msg, BUFF_SIZE,
"BGE");
        else if((opcode & mask) == BLT)                              snprintf(msg, BUFF_SIZE,
"BLT");
        else if((opcode & mask) == BGT)                              snprintf(msg, BUFF_SIZE,
"BGT");
        else if((opcode & mask) == BLE)                              snprintf(msg, BUFF_SIZE,
"BLE");
        else if((opcode & mask) == CLR)                              snprintf(msg, BUFF_SIZE,
"CLR");
        else if((opcode & mask) == COM)                              snprintf(msg, BUFF_SIZE,
"COM");
        else if((opcode & mask) == INC)                              snprintf(msg, BUFF_SIZE,
"INC");
        else if((opcode & mask) == DEC)                              snprintf(msg, BUFF_SIZE,
"DEC");
        else if((opcode & mask) == NEG)                              snprintf(msg, BUFF_SIZE,
"NEG");
        else if((opcode & mask) == ADC)                              snprintf(msg, BUFF_SIZE,
"ADC");
        else if((opcode & mask) == SBC)                              snprintf(msg, BUFF_SIZE,
"SBC");
        else if((opcode & mask) == ROR)                              snprintf(msg, BUFF_SIZE,
"ROR");
        else if((opcode & mask) == ROL)                              snprintf(msg, BUFF_SIZE,
"ROL");
        else if((opcode & mask) == ASR)                              snprintf(msg, BUFF_SIZE,
"ASR");
        else if((opcode & mask) == ASL)                              snprintf(msg, BUFF_SIZE,
"ASL");
        else if((opcode & mask) == MOV)                              snprintf(msg, BUFF_SIZE,
"MOV");
        else if((opcode & mask) == CMP)                              snprintf(msg, BUFF_SIZE,
"CMP");
        else if((opcode & mask) == BIT)                              snprintf(msg, BUFF_SIZE,
"BIT");
        else if((opcode & mask) == BIC)                              snprintf(msg, BUFF_SIZE,
"BIC");
        else if((opcode & mask) == BIS)                              snprintf(msg, BUFF_SIZE,
"BIS");
        else if((opcode & mask) == ADD)                              snprintf(msg, BUFF_SIZE,
"ADD");
        else if((mask == 070000) && ((opcode >> 9) == (JSR >> 9)))        snprintf(msg, BUFF_SIZE, "JSR");
        else if((mask == 070000) && ((opcode >> 6) == (RTS >> 6)))        snprintf(msg, BUFF_SIZE, "RTS");
        else if((opcode & (mask | 077000)) == MUL)                  snprintf(msg, BUFF_SIZE, "MUL");
        else if((opcode & (mask | 077000)) == DIV)                  snprintf(msg, BUFF_SIZE, "DIV");
        else if((opcode & (mask | 077000)) == ASH)                  snprintf(msg, BUFF_SIZE, "ASH");
        else if((opcode & (mask | 077000)) == ASHC)                 snprintf(msg, BUFF_SIZE, "ASHC");
        else if((opcode & (mask | 077000)) == XOR)                  snprintf(msg, BUFF_SIZE, "XOR");
        else if((opcode | mask) == HALT)                            snprintf(msg, BUFF_SIZE, "HALT");
        else
                valid = 0;
```

```
}else {
        valid = 1;
        printf("instr: %06o\n", opcode & mask);
        /***** Byte Instruction Opcodes *****/
        uint32_t BPLmask = 0177777;
        if(opcode == 0100000)                       valid = 0;
        else if((opcode & BPLmask) == BPL)      snprintf(msg, BUFF_SIZE, "BPL");
        else if((opcode & mask) == BMI)         snprintf(msg, BUFF_SIZE, "BMI");
        else if((opcode & mask) == BHI)                 snprintf(msg, BUFF_SIZE, "BHI");
        else if((opcode & mask) == BLOS)        snprintf(msg, BUFF_SIZE, "BLOS");
        else if((opcode & mask) == BVC)                 snprintf(msg, BUFF_SIZE, "BVC");
        else if((opcode & mask) == BVS)                 snprintf(msg, BUFF_SIZE, "BVS");
        else if((opcode & mask) == BCC)                 snprintf(msg, BUFF_SIZE, "BCC");
        else if((opcode & mask) == BCS)                 snprintf(msg, BUFF_SIZE, "BCS");
        else if((opcode & mask) == CLRB)        snprintf(msg, BUFF_SIZE, "CLRB");
        else if((opcode & mask) == COMB)        snprintf(msg, BUFF_SIZE, "COMB");
        else if((opcode & mask) == INCB)        snprintf(msg, BUFF_SIZE, "INCB");
        else if((opcode & mask) == DECB)        snprintf(msg, BUFF_SIZE, "DECB");
        else if((opcode & mask) == NEGB)        snprintf(msg, BUFF_SIZE, "NEGB");
        else if((opcode & mask) == ADCB)        snprintf(msg, BUFF_SIZE, "ADCB");
        else if((opcode & mask) == SBCB)        snprintf(msg, BUFF_SIZE, "SBCB");
        else if((opcode & mask) == RORB)        snprintf(msg, BUFF_SIZE, "RORB");
        else if((opcode & mask) == ROLB)        snprintf(msg, BUFF_SIZE, "ROLB");
        else if((opcode & mask) == ASRB)        snprintf(msg, BUFF_SIZE, "ASRB");
        else if((opcode & mask) == ASLB)        snprintf(msg, BUFF_SIZE, "ASLB");
        else if((opcode & mask) == MOVB)        snprintf(msg, BUFF_SIZE, "MOVB");
        else if((opcode & mask) == CMPB)        snprintf(msg, BUFF_SIZE, "CMPB");
        else if((opcode & mask) == BITB)        snprintf(msg, BUFF_SIZE, "BITB");
        else if((opcode & mask) == BICB)        snprintf(msg, BUFF_SIZE, "BICB");
        else if((opcode & mask) == BISB)        snprintf(msg, BUFF_SIZE, "BISB");
        else if((opcode & mask) == SUB)                 snprintf(msg, BUFF_SIZE, "SUB");
        else
                valid = 0;
}

return valid;

}

/**********

Get all Instructions

Store them as either Single
or Double Operand Instructions

**********/
int get_instruction(uint16_t *oct, instr_single *s,
                        instr_double *d, int n_lines,
                        int *n_single, int *n_double, int instr_start,
                        uint16_t *PC) {

int ret = ERROR_NONE;

/***** Word or Byte Instruction *****/
int BYTE = 0;
int WORD = 1;
int SINGLE = 0;
int DOUBLE = 1;

int flag = 0;

/***** Masks for Single Operand Values *****/
uint32_t sbm                        = 0100000;
uint16_t s_op_mask                  = 0177700;
uint16_t s_mode_dd_mask     = 0100070;
uint16_t s_dd_mask                  = 0100007;

/***** Masks for Double Operand Values *****/
uint16_t d_op_mask          = 0170000;
uint16_t d_mode_ss_mask             = 0107000;
uint16_t d_ss_mask          = 0100700;
uint16_t d_mode_dd_mask             = 0100070;
uint16_t d_dd_mask          = 0100007;

/***** Masks for Address Modes *****/
uint16_t ss_mode_mask       = 0007000;
uint16_t dd_mode_mask       = 0000070;
uint16_t ss_dummy;

/***** Branch Offset Mask *****/
uint16_t br_offset_mask = 0000377;
uint16_t temp_branch        = 0000000;

/***** Assign Byte Instructions to Struct *****/
for(int i = instr_start; i < n_lines; i++) {
        if((oct[i] & sbm) == sbm) {
                s[*n_single].opcode = (oct[i] & s_op_mask)              >> 6;
                s[*n_single].mode_dd        = (oct[i] & s_mode_dd_mask) >> 3;
                s[*n_single].dd                     = (oct[i] & s_dd_mask);
                s[*n_single].PC                     = PC[i];

                d[*n_double].opcode                 = (oct[i] & d_op_mask)              >> 12;
                d[*n_double].mode_ss        = (oct[i] & d_mode_ss_mask) >> 9;
                d[*n_double].ss                     = (oct[i] & d_ss_mask)              >> 6;
                d[*n_double].mode_dd        = (oct[i] & d_mode_dd_mask) >> 3;
                d[*n_double].dd                     = (oct[i] & d_dd_mask);
                d[*n_double].PC                     = PC[i];

                temp_branch = oct[i] - (oct[i] & br_offset_mask);

                /***** Only Assign if Opcode Valid *****/
                if(valid_opcode(oct[i], s_op_mask, BYTE, s[*n_single].instr)) {
                        printf("single- dd_m: %o dd: %o\n", s[*n_single].mode_dd, s[*n_single].dd);
                        if(s[*n_single].dd == 07){
                                if(s[*n_single].mode_dd == 02)
                                        ++i;
                                else if(s[*n_single].mode_dd == 03)
                                        ++i;
                                else if(s[*n_single].mode_dd == 06)
                                        ++i;
                                else if(s[*n_single].mode_dd == 07)
                                        ++i;
                        } else if(s[*n_single].mode_dd == 06)
                                        ++i;
                        else if(s[*n_single].mode_dd == 07)
                                        ++i;
                ++(*n_single);
                }else if(valid_opcode(oct[i], d_op_mask, BYTE, d[*n_double].instr)) {
                        if(d[*n_double].ss == 07){
                                if(d[*n_double].mode_ss == 02)
                                        ++i;
                                else if(d[*n_double].mode_ss == 03)
                                        ++i;
                                else if(d[*n_double].mode_ss == 06)
                                        ++i;
                                else if(d[*n_double].mode_ss == 07)
                                        ++i;
                        } else if(d[*n_double].mode_ss == 06)
                                        ++i;
                        else if(d[*n_double].mode_ss == 07)
                                        ++i;
                        if(d[*n_double].dd == 07){
                                if(d[*n_double].mode_dd == 02)
                                        ++i;
                                else if(d[*n_double].mode_dd == 03)
                                        ++i;
                                else if(d[*n_double].mode_dd == 06)
                                        ++i;
                                else if(d[*n_double].mode_dd == 07)
                                        ++i;
                        } else if(d[*n_double].mode_dd == 06)
                                        ++i;
                        else if(d[*n_double].mode_dd == 07)
                                        ++i;
                        ++(*n_double);
                }else if(valid_opcode(temp_branch, s_op_mask, BYTE, s[*n_single].instr)) {
                        s[*n_single].opcode = (temp_branch & s_op_mask)         >> 6;
                        s[*n_single].mode_dd        = (temp_branch & s_mode_dd_mask) >> 3;
                        s[*n_single].dd             = (temp_branch & s_dd_mask);
                        ++(*n_single);
                }else
                        continue;
        }
}

/***** Assign Word Instructions to Struct *****/
for(int i = instr_start; i < n_lines; i++) {
        if((oct[i] & sbm) != sbm) {
                d[*n_double].opcode         = (oct[i] & d_op_mask)          >> 12;
                d[*n_double].mode_ss    = (oct[i] & d_mode_ss_mask)  >> 9;
                d[*n_double].ss             = (oct[i] & d_ss_mask)          >> 6;
                d[*n_double].mode_dd    = (oct[i] & d_mode_dd_mask)  >> 3;
                d[*n_double].dd             = (oct[i] & d_dd_mask);
                d[*n_double].PC             = PC[i];

                s[*n_single].opcode = (oct[i] & s_op_mask)              >> 6;
                s[*n_single].mode_dd    = (oct[i] & s_mode_dd_mask)  >> 3;
                s[*n_single].dd             = (oct[i] & s_dd_mask);
                s[*n_single].PC             = PC[i];
```

```
temp_branch = oct[i] - (oct[i] & br_offset_mask);

/***** Only Assign if Opcode Valid *****/
if(valid_opcode(oct[i], d_op_mask, WORD, d[*n_double].instr)) {
        printf("ss_m:%o ss:%o\n", d[*n_double].mode_ss, d[*n_double].ss);
        printf("dd_m:%o dd:%o\n", d[*n_double].mode_dd, d[*n_double].dd);
        if(d[*n_double].ss == 07){
                if(d[*n_double].mode_ss == 02)
                        ++i;
                else if(d[*n_double].mode_ss == 03)
                        ++i;
                else if(d[*n_double].mode_ss == 06)
                        ++i;
                else if(d[*n_double].mode_ss == 07)
                        ++i;
        } else if(d[*n_double].mode_ss == 06)
                        ++i;
        else if(d[*n_double].mode_ss == 07)
                        ++i;
        if(d[*n_double].dd == 07){
                if(d[*n_double].mode_dd == 02)
                        ++i;
                else if(d[*n_double].mode_dd == 03)
                        ++i;
                else if(d[*n_double].mode_dd == 06)
                        ++i;
                else if(d[*n_double].mode_dd == 07)
                        ++i;
        } else if(d[*n_double].mode_dd == 06)
                        ++i;
        else if(d[*n_double].mode_dd == 07)
                        ++i;

        ++(*n_double);
} else if(valid_opcode(oct[i], s_op_mask, WORD, s[*n_single].instr)) {
        if(s[*n_single].dd == 07){
                if(s[*n_single].mode_dd == 02)
                        ++i;
                else if(s[*n_single].mode_dd == 03)
                        ++i;
                else if(s[*n_single].mode_dd == 06)
                        ++i;
                else if(s[*n_single].mode_dd == 07)
                        ++i;
        } else if(s[*n_single].mode_dd == 06)
                        ++i;
        else if(s[*n_single].mode_dd == 07)
                        ++i;
        ++(*n_single);
} else if(valid_opcode(temp_branch, s_op_mask, WORD, s[*n_single].instr)) {
        s[*n_single].opcode = (temp_branch & s_op_mask)          >> 6;
        s[*n_single].mode_dd        = (temp_branch & s_mode_dd_mask) >> 3;
        s[*n_single].dd             = (temp_branch & s_dd_mask);
        ++(*n_single);
} else if((i == n_lines-1) && valid_opcode(oct[i], HALT, WORD,
                                        s[*n_single].instr)) {
```

```
					s[*n_single].PC = PC[i];
					++(*n_single);
			} else
					continue;
		}
}

/***** No Valid Instructions *****/
if(((*n_single == 0) && (*n_double == 0)) || (ret == ERROR))
		ret = ERROR;
else
		ret = ERROR_NONE;

return ret;

}

/**********

Read Data from ASCII File

**********/
int rd_ascii_file(char *filename, char **line, int *n_lines){

char		buf[BUFF_SIZE];
int		ret = ERROR_NONE;

FILE *fp;

fp = fopen(filename, "r");

if(fp == NULL){
		printf("Can't open file!\n");
		printf("Format: ./pdp filepath/filename\n");
		ret = ERROR;
		return ret;
}

/***** Read Ascii Line by Line *****/
while(fscanf(fp, "%s", buf) != EOF){
		line[*n_lines] = (char*)malloc(BUFF_SIZE*sizeof(char));
		snprintf(line[*n_lines], sizeof line[*n_lines], "%s", buf);
		++(*n_lines);
}

/***** @000000 Needs to be Removed *****/
--(*n_lines);

fclose(fp);

return ret;

}

/**********

Convert String to Octal Value
```

```
**********/
int str_to_oct(char ** line, unsigned long *oct, uint16_t *oct16, int n_lines,
                uint16_t *PC, uint16_t start_addr, int *start_instr,
                var_data *data, int *n_data){

char        new_line[BUFF_SIZE];
int         ret = ERROR_NONE;

/***** Remove Unwanted Chars and Convert String -> Octal *****/
for(int i = 0; i < n_lines; i++){
            memmove(new_line, line[i]+1, 7);
            oct[i] = strtoul(new_line, NULL, 8);
}

for(int i = 0; i < n_lines; i++) {
            oct16[i] = oct[i+1];

            /***** Increment Program Counter *****/
            if(i > 0)
                        PC[i]                   = PC[i-1] + 2;
            else
                        PC[i]                   = 0;

            /***** Store the PC for the HALT Instruction *****/
            if(i == (n_lines - 1))
                        PC[n_lines]             = PC[i-1] + 2;

            /***** Find the First Program Instruction *****/
            if(start_addr == PC[i])
                        *start_instr = i;
}

/***** Store Program Data before Instructions *****/
for(int i = 0; i < *start_instr; i++) {
            data[i].data            = oct16[i];
            data[i].PC= PC[i];
            ++(*n_data);
}

return ret;

}

/**********

Run macro11/obj2ascii Converter

**********/
int obj2ascii(char *file){

char                    buff[100];
char                    buff2[200];

char            *s1                 = "cd ascii; ./macro11 ";
char            *s2                 = ".mac -o ";
char            *s3                 = ".obj -l ";
char        *s4                 = ".lst";
char            *s5             = " -e AMA";

char            *s6             = "cd ascii; ./obj2ascii ";
char            *s7             = ".obj ";
char            *s8             = ".ascii";

int                     ret                     = ERROR_NONE;

sprintf(buff, s1);
sprintf(buff + strlen(buff), file);
sprintf(buff + strlen(buff), s2);
sprintf(buff + strlen(buff), file);
sprintf(buff + strlen(buff), s3);
sprintf(buff + strlen(buff), file);
sprintf(buff + strlen(buff), s4);
printf("%s\n", buff);

sprintf(buff2, s6);
sprintf(buff2 + strlen(buff2), file);
sprintf(buff2 + strlen(buff2), s7);
sprintf(buff2 + strlen(buff2), file);
sprintf(buff2 + strlen(buff2), s8);
printf("%s\n", buff2);

/***** Run on Command Line *****/
#if AMA
sprintf(buff + strlen(buff), s5);
ret = system(buff);
#else
ret = system(buff);
#endif

ret = system(buff2);

return ret;

}
```

pdp.h

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "opcodes.h"

/*********

DEFINES

**********/

/***** State *****/
#define ERROR_NONE          0
#define ERROR               -1

/***** Buffer Size *****/
#define BUFF_SIZE               100

/***** File I/O Buffer *****/
#define LINE_SIZE  500

/***** Memory Size *****/
#define MAX_ADDR                0200000

/**********

VARIABLES

**********/

uint16_t R0;
uint16_t R1;
uint16_t R2;
uint16_t R3;
uint16_t R4;
uint16_t R5;
uint16_t R6;
uint16_t R7;

uint8_t N[LINE_SIZE];
uint8_t Z[LINE_SIZE];
uint8_t V[LINE_SIZE];
uint8_t C[LINE_SIZE];

/**********

STRUCTS

**********/
typedef struct {
        int             type;
        uint16_t    addr;
        char            descr[BUFF_SIZE];
} sim_output;

typedef struct {
        uint16_t    data;
        uint16_t    PC;
        uint16_t    memory[LINE_SIZE];
        int             n_memory;
} var_data;

typedef struct {
        uint16_t    opcode;
        uint8_t         mode_dd;
        uint8_t         dd;
        uint16_t    PC;
        uint16_t    dd_reg;
        char        instr[BUFF_SIZE];
} instr_single;

typedef struct {
        uint8_t     opcode;
        uint8_t         mode_ss;
        uint8_t         ss;
        uint8_t         mode_dd;
        uint8_t         dd;
        uint16_t    PC;
        uint16_t    ss_reg;
        uint16_t    dd_reg;
        char        instr[BUFF_SIZE];
} instr_double;

/**********

FUNCTIONS

***********/

/***** File I/O *****/
int rd_ascii_file(char *, char **, int *);

/***** String to Octal *****/
int str_to_oct(char **, unsigned long *, uint16_t *, int, uint16_t *,
            uint16_t, int *, var_data *, int *);

/***** Obj2Ascii *****/
int obj2ascii(char *);

/***** Word Instruction *****/
int get_instruction(uint16_t *, instr_single *, instr_double *,
                    int, int *, int *, int, uint16_t *);

/***** Determin Valid Opcode *****/
int valid_opcode(uint16_t, uint16_t, int, char *);

/***** Find Address Mode Source *****/
```

```
int ss_addr_mode(uint16_t, int, int *);

/***** Find Address Mode Destination *****/
int dd_addr_mode(uint16_t, int, int *, int);

/***** Store Source/Destination Values *****/
int store_reg_vals(uint16_t *, instr_single *, instr_double *, uint16_t*, int, int, int);

/***** Fetch Simulator Instructions *****/
int fetch_instructions(uint16_t *, instr_single *, instr_double *, uint16_t *, var_data *,
                       sim_output *, int, int, int, int, int *);

/***** Get Memory Addresses for Data Accesses *****/
int data_mem_addr(uint16_t *, var_data *, uint16_t *, int, int);

/***** Set Register Values *****/
int set_reg(uint16_t, uint16_t);

/***** Get Register Values *****/
int get_reg(uint16_t *, uint16_t);
```

opcodes.h

```
#include <stdint.h>


/***** Addressing Modes *****/
static const uint16_t dd_REG            = 0000000;
static const uint16_t dd_REG_DEF        = 0000010;
static const uint16_t dd_AUT_INC        = 0000020;
static const uint16_t dd_AUT_INC_DEF    = 0000030;
static const uint16_t dd_AUT_DEC        = 0000040;
static const uint16_t dd_AUT_DEC_DEF= 0000050;
static const uint16_t dd_IND            = 0000060;
static const uint16_t dd_IND_DEF        = 0000070;

static const uint16_t ss_REG            = 0000000;
static const uint16_t ss_REG_DEF        = 0001000;
static const uint16_t ss_AUT_INC        = 0002000;
static const uint16_t ss_AUT_INC_DEF    = 0003000;
static const uint16_t ss_AUT_DEC        = 0004000;
static const uint16_t ss_AUT_DEC_DEF= 0005000;
static const uint16_t ss_IND            = 0006000;
static const uint16_t ss_IND_DEF        = 0007000;

/***** Program Counter Addressing *****/
static const uint16_t dd_IMM            = 0000027;
static const uint16_t dd_ABS            = 0000037;
static const uint16_t dd_REL            = 0000067;
static const uint16_t dd_REL_DEF        = 0000077;

static const uint16_t ss_IMM            = 0002700;
static const uint16_t ss_ABS            = 0003700;
static const uint16_t ss_REL            = 0006700;
static const uint16_t ss_REL_DEF        = 0007700;

/***** Halt Command - Stops Program *****/
static const uint16_t HALT      = 0000000;

/***** Single Operand - Word *****/
static const uint16_t JMP       = 0000100;
static const uint16_t SWAB      = 0000300;
static const uint16_t BR        = 0000400;
static const uint16_t BNE       = 0001000;
static const uint16_t BEQ       = 0001400;
static const uint16_t BGE       = 0002000;
static const uint16_t BLT       = 0002400;
static const uint16_t BGT       = 0003000;
static const uint16_t BLE       = 0003400;
static const uint16_t CLR       = 0005000;
static const uint16_t COM       = 0005100;
static const uint16_t INC       = 0005200;
static const uint16_t DEC       = 0005300;
static const uint16_t NEG       = 0005400;
static const uint16_t ADC       = 0005500;
static const uint16_t SBC       = 0005600;
static const uint16_t ROR       = 0006000;
static const uint16_t ROL       = 0006100;
static const uint16_t ASR       = 0006200;
static const uint16_t ASL       = 0006300;

/***** Double Operand - Word *****/
static const uint16_t MOV       = 0010000;
static const uint16_t CMP       = 0020000;
static const uint16_t BIT       = 0030000;
static const uint16_t BIC       = 0040000;
static const uint16_t BIS       = 0050000;
static const uint16_t ADD       = 0060000;
static const uint16_t MUL       = 0070000;
static const uint16_t DIV       = 0071000;
static const uint16_t ASH       = 0072000;
static const uint16_t ASHC      = 0073000;
static const uint16_t XOR       = 0074000;
static const uint16_t JSR       = 0004000;
static const uint16_t RTS       = 0000200;

/**** Single Operand - Byte *****/
static const uint16_t BPL       = 0100000;
static const uint16_t BMI       = 0100400;
static const uint16_t BHI       = 0101000;
static const uint16_t BLOS      = 0101400;
static const uint16_t BVC       = 0102000;
static const uint16_t BVS       = 0102400;
static const uint16_t BCC       = 0103000;
static const uint16_t BCS       = 0103400;
static const uint16_t CLRB      = 0105000;
static const uint16_t COMB      = 0105100;
static const uint16_t INCB      = 0105200;
static const uint16_t DECB      = 0105300;
static const uint16_t NEGB      = 0105400;
static const uint16_t ADCB      = 0105500;
static const uint16_t SBCB      = 0105600;
static const uint16_t RORB      = 0106000;
static const uint16_t ROLB      = 0106100;
static const uint16_t ASRB      = 0106200;
static const uint16_t ASLB      = 0106300;

/***** Double Operand - Byte *****/
static const uint16_t MOVB      = 0110000;
static const uint16_t CMPB      = 0120000;
static const uint16_t BITB      = 0130000;
static const uint16_t BICB      = 0140000;
static const uint16_t BISB      = 0150000;
static const uint16_t SUB       = 0160000;
```

Example Assembly File For Testing

```
;PDP11 Assembly Instructions
;used to generate ascii read in by simulator

OLD:    .WORD   0
NEW:    .WORD   0
FIBO:   .WORD   0
N:      .WORD   6

START:
        CLR R0
        CLR R1
        MOV #1, R0
        MOV #2, R1
        MOV R1, R0
        MOV R0, (R1)
        MOV R0, (R1)+
        MOV R0, @(R1)+
        MOV R0, -(R1)
        MOV R0, @-(R1)
        MOV R0, 200(R1)
        MOV R0, @200(R1)
        MOV R0, @#200(R1)
        MOV R0, OLD
        MOV R0, @OLD

END:    HALT
        .END    START
```

Output Trace

```
2 000010
2 000012
2 000014
0 000016
2 000020
0 000022
2 000024
2 000026
0 000022
```

```
2 000030
0 000024
2 000032
0 010067
2 000034
0 000024
2 000036
0 010071
2 000040
0 000066
1 000066
2 000044
0 000136
1 000000
2 000050
0 000054
1 000200
2 000054
0 000060
1 000002
2 000060
0 000064
1 000000
2 000064
```