Block I
Apollo Guidance Computer (AGC)

How to build one in your basement

Part 6: Assembler

John Pultorak
December, 2004

Abstract

This report describes my successful project to build a working reproduction of the 1964 prototype for the Block I Apollo Guidance Computer. The AGC is the flight computer for the Apollo moon landings, and is the world's first integrated circuit computer.

I built it in my basement. It took me 4 years.

If you like, you can build one too. It will take you less time, and yours will be better than mine.

I documented my project in 9 separate .pdf files:

Part 1	Overview: Introduces the project.
Part 2	CTL Module: Design and construction of the control module.
Part 3	PROC Module: Design and construction of the processing (CPU) module.
Part 4	MEM Module: Design and construction of the memory module.
Part 5	IO Module: Design and construction of the display/keyboard (DSKY) module.
Part 6	Assembler: A cross-assembler for AGC software development.
Part 7	C++ Simulator: A low-level simulator that runs assembled AGC code.
Part 8	Flight Software: My translation of portions of the COLOSSUS 249 flight software.
Part 9	Test & Checkout: A suite of test programs in AGC assembly language.

Overview

This AGC cross-assembler, coded in C++, produces AGC object code in Motorola S-format. The code can be burned into EPROM for use by the hardware AGC, or executed by the C++ AGC simulator.

Operation

The assembler reads an AGC source code text file (having a .asm extension). It generates an assembly listing text file (.lst extension) and an two object code text files (_H.hex and _L.hex suffixes). The object code files are readable by the AGC simulator or an EPROM programmer.

Syntax

Source code files are text files containing multiple lines of source code. Each line is terminated by a newline character. Source code files can be produced by any editor, as long as it doesn't insert any hidden characters or formatting information.

Each line of source code consists of one of the following:

a) A blank line.
b) A comment (comments must begin with a semicolon (;)).
c) A line of assembler code.

The assembler ignores blank lines and anything that occurs after a semicolon on any given line.

A line of assembler code consists of the following components:

1) A label (this is optional).
2) An op code or assembler directive.
3) An operand (also optional).
4) A comment (optional; comments must start with a semicolon).

The components, if present, must appear in the order given. Each component is separated from the next by one or more white spaces or tabs. The only constraint is that the label, if present, must start in the 1st column of the line. If no label is present, the op code or assembler directive must not start in the 1st column, but may appear in any subsequent column.

The operand may consist of one of the following:

a) An asterisk (the assembler substitutes in the current value of the location counter.
b) A symbolic name (the assemble substitutes in the value during the second pass of the assembly.
c) A numeric constant. Octal contants must be preceeded by a '%'; hexadecimal constants are preceeded by a '$'; anything else is treated as a decimal.
d) An expression consisting of one or more of the above, separated by the operators: +, -, @ (for multiplication), or / (for division). Unary plus or minus is also allowed.

Examples:

*+2	Location counter plus 2.
LABEL+%10/2	LABEL plus octal number 10 divided by decimal number 2.
-5	Negative decimal 5.

Assembler Directives

The following directives are supported:

ORG	Set the location counter to the operand following ORG.
EQU	Set the label to the operand value.
DS	Define a word of storage; set the value to the operand value.
CADR	Define a word of storage; set the value to the operand value (assumed to be a 14-bit address; this is an alias for CADR).
ADRES	Define a word of storage; set the value to the operand value (operand is treated as a 12-bit address).
INCL	Inline include a file; the operand contains the filename.

Addressing

The location counter and symbol table always contain the full 14-bit address. AGC instructions require a 12-bit address as an operand. Addressing is implemented in 1K banks (bits 10 through 1). Bits 12 and 11 select banks 0 (erasable memory), 1 (fixed-fixed), or 2 (fixed-fixed). These banks are directly addressable in 12-bits. Banks above 2 are called 'fixed-switchable' and are addressed using the BANK register: bits 12 and 11 of the operand are set to '1' and the bank register provides the high-order 4 bits (bits 14 through 11) of the address. The lower 10 bits of the operand provide the address within the bank.

Errata

a) The assembler ignores the last line of the source (.asm) file. If the last line of your source file contains code, you must add an additional blank line to the end of your source file to ensure that your last line of code is assembled.

b) The symbol table could be sorted before the second pass. The linear search through the symbol table could be replaced by a more efficient method. (But, assembly is so fast, it doesn't matter).

c) The assembler directives and syntax don't match those of the original block I AGC.

d) A macro definition capability might be handy.

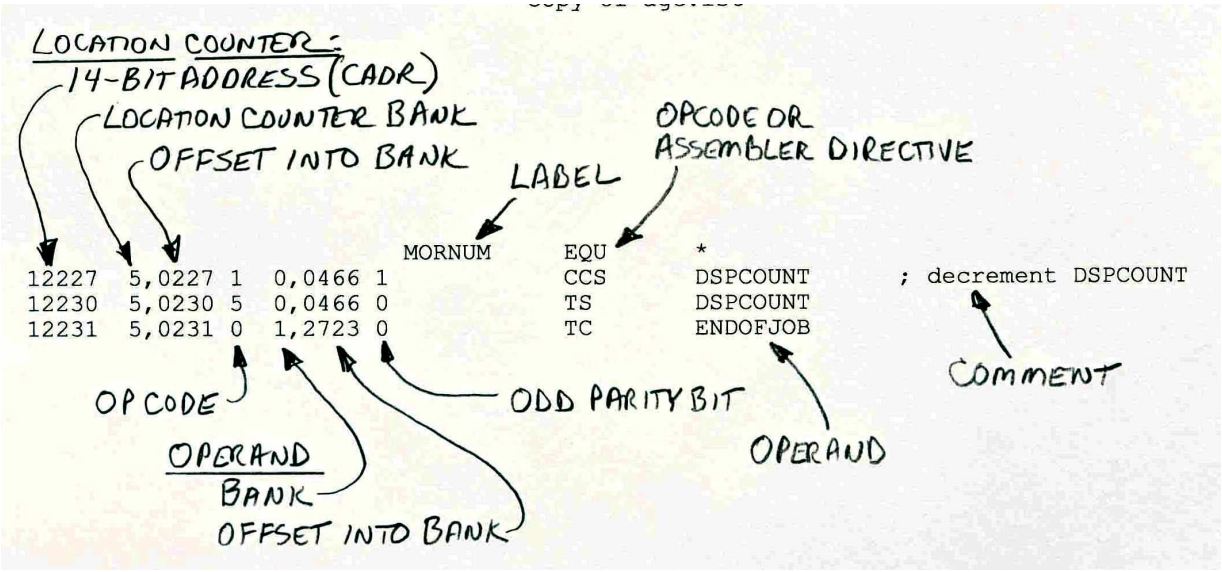
e) Generates empty .lst and .obj files if the source code file does not exist.

f) Incorrectly assigns a zero value to labels that equate to a forward referenced label. i.e.:

```
LABEL1      EQU   LABEL2        ; assembler incorrectly sets LABEL1 to zero
LABEL2      EQU   *
```

Assembler listing

The assembler produces a list file (.lst) that shows the assembled source code. The format of the file was designed to mimic (somewhat) the format of the original assembler. You will find a fragment of the original assembler listing reproduced in part 8.



List files from my assembler have this format:

Object code

The object code is output to text files in Motorola S-Record format (S2F) suitable for immediate use by EPROM programmers, and also readable by the C++ simulator.

There's lots of documentation on S-Record formats on the internet.

The files are pretty big. If you open one up, you'll discover they contain many short records that look like these:

```
S2140004001800000160117CF160117CF160117CFD8
S2140004101601 17CF160117CF0F272C48A4BC0026AD
S2140004207BBCCC4040250026400101310141 2A40DA
S2140004303223012B00232323410101 02FDFEFF008E
S2140004400102010000000001014 13B403C42422302
S2140004502353230042002323233D424223236023C9
```

This is a fragment of the source code for the lower 8 bits of the TECO1 test and checkout program (described in part 9).

The assembler contains C++ code for generating these files; the C++ simulator has code for reading them.

Assembler v1.6

```
/*
 ************************************************************
 *
 * Cross Assembler for Block I Apollo Guidance Computer (AGC4)
 * THIS VERSION IS MODIFIED TO OUTPUT IN S-RECORD FORMAT SUITABLE FOR
 * LOADING IN EPROM (s2f format).
 * Author: John Pultorak
 * 6/1/2002
 *
 ************************************************************

  Versions:
       1.0 -  First version of the AGC assembler.
       1.1 -  Added ability to handle simple arithmetic expressions for the operand.
       1.2 -  Changed label fields to 14 char. Printed symbol table in 3 columns.
                 Corrected wrong implementation of OVSK.
       1.3 -  Added support for bank addressing.
       1.4 -  Added capability to inline code with nested includes.
       1.5 -  Added CADR, ADRES assembler directives. Swapped addresses for
                 TIME1 and TIME2 for compatibility with Block II PINBALL routines.
       1.6 -  Fixed the parser so it doesn't mangle comments anymore. Added
                 FCADR and ECADR to the assembler directives. Added a check for
                 assembled code that overwrites already used locations.
       1.6 -  EPROM Assembler outputs to low and high EPROM files.

Operation:
       The assembler reads an AGC source code file (having a .asm extension).
       It generates an assembly listing text file (.lst extension) and an
       object code text file (.obj extension). The object code file is readable
       by the AGC simulator.

Syntax:
       Source code files are text files containing multiple lines of source code.
       Each line is terminated by a newline character. Source code files can be
       produced by any editor, as long as it doesn't insert any hidden characters
       or formatting information.

       Each line of assembly code consists of one of the following:
              a) a blank line;
              b) a comment (comments must begin with a semicolon (;));
              c) or a line of assembler code.

       The assembler ignores blank lines and anything that occurs after a semicolon
       on any given line.

       A line of assembler code consists of the following components:
              1) a label (this is optional);
              2) an op code or assembler directive;
              3) an operand (also optional);
              4) a comment (optional; comments must start with a semicolon)

       The components, if present, must appear in the order given. Each component is
       separated from the next by one or more white spaces or tabs. The only constraint
       is that the label, if present, must start in the 1st column of the line. If no
       label is present, the op code or assembler directive must not start in the 1st
       column, but may appear in any subsequent column.

       The operand may consist of one of the following:
              a) an asterisk (the assembler substitutes in the current value of the
                    location counter;
              b) a symbolic name (the assemble substitutes in the value during the
                    second pass of the assembly;
              c) a numeric constant. Octal contants must be preceeded by a '%';
                    hexadecimal constants are preceeded by a '$'; anything else
                    is treated as a decimal.
              d) an expression consisting of one or more of the above, separated by
                  the operators: +, -, @ (for multiplication), or / (for division).
                    Unary plus or minus is also allowed.
                    examples:
                           *+2                   means location counter plus 2
                           LABEL+%10/2
                           -5
```

```
Assembler Directives:
        The following directives are supported:
        ORG - set the location counter to the operand following ORG.
        EQU     - set the label to the operand value.
        DS - define a word of storage; set the value to the operand value.
        CADR - define a word of storage; set the value to the operand value
               (assumed to be a 14-bit address; this is an alias for CADR).
        ADRES - define a word of storage; set the value to the operand value
               (operand is treated as a 12-bit address).
        INCL - inline include a file; the operand contains the filename.

Addressing:
        The location counter and symbol table always contain the full 14-bit address.
        AGC instructions require a 12-bit address as an operand. Addressing is implemented
        in 1K banks (bits 10 through 1). Bits 12 and 11 select banks 0 (erasable memory),
        1 (fixed-fixed), or 2 (fixed-fixed). These banks are directly addressable in
        12-bits. Banks above 2 are called 'fixed-switchable' and are addressed using
        the BANK register: bits 12 and 11 of the operand are set to '1' and the bank
        register provides the high-order 4 bits (bits 14 through 11) of the address.
        The lower 10 bits of the operand provide the address within the bank.

Errata:
        The assembler ignores the last line of the source (.asm) file. If the last
        line of your source file contains code, you must add an additional blank
        line to the end of your source file to ensure that your last line of code
        is assembled.

        The symbol table should be sorted before the second pass. The linear
        search through the symbol table should be replaced by a more efficient
        method.

        The assembler directives and syntax don't match those of the original block
        I AGC.

        A macro definition capability would be handy.

    Generates empty .lst and .obj files if the source code file does not exist.

    Incorrectly assigns a zero value to labels that equate to a forward referenced
        label. i.e.:
                LABEL1          EQU             LABEL2          ; assembler incorrectly sets LABEL1
to zero
                LABEL2          EQU             *

        The assembler generates object code for eraseable memory addresses.
*/

#include <string.h>
#include <stdlib.h>
#include <ctype.h>
#include <iostream.h>
#include <stdio.h>


        // Create a flag for each memory location in a 14-bit address space
        // These are initialized to false, and then set as the code is assembled.
        // An attempt to assemble code on top of a previously used location is
        // flagged as an error.
const unsigned agcMemSize = 037777+1; // # of cells in a 14-bit address range
bool memoryUsed [agcMemSize];

//*********************************************************************
// MODIFIED FROM THE ORIGINAL ASSEMBLER HERE.
// This array represents the fixed memory PROM (14-bit address)
int EPROM_H [agcMemSize];
int EPROM_L [agcMemSize];
//*********************************************************************

unsigned pass = 0;
unsigned locCntr = 0; // address in 14-bit format
unsigned errorCount = 0;

FILE* fpList = 0;      // output (assembly listing)
```

```
//*********************************************************************
// MODIFIED FROM THE ORIGINAL ASSEMBLER HERE.
FILE* fpObj_H = 0;     // output MSB's (object code in S-RECORD format)
FILE* fpObj_L = 0;     // output LSB's (object code in S-RECORD format)
//*********************************************************************


struct ocode
{
        char* name;
        unsigned code;
        bool isOpCode; // true=it is an op code
        bool addrOpnd; // true=convert operand to 12-bit address format
        unsigned len; // words
};

ocode allcodes[] =
{
                // Block I op codes.
        { "TC",         0000000, true, true, 1 },
        { "CCS",        0010000, true, true, 1 },
        { "INDEX",      0020000, true, true, 1 },
        { "XCH",        0030000, true, true, 1 },
        { "CS",         0040000, true, true, 1 },
        { "TS",         0050000, true, true, 1 },
        { "AD",         0060000, true, true, 1 },
        { "MASK",       0070000, true, true, 1 },
        { "MP",         0040000, true, true, 1 },
        { "DV",         0050000, true, true, 1 },
        { "SU",         0060000, true, true, 1 },

                // Implied address codes (R393: 3-12)
        { "RESUME",     0020025, true, false, 1 },     // INDEX 25
        { "EXTEND",     0025777, true, false, 1 },     // INDEX 5777
        { "INHINT",     0020017, true, false, 1 },     // INDEX 17
        { "RELINT",     0020016, true, false, 1 },     // INDEX 16
        { "XAQ",        0000000, true, false, 1 },     // TC A
        { "RETURN",     0000001, true, false, 1 },     // TC Q
        { "NOOP",       0030000, true, false, 1 },     // XCH A
        { "COM",        0040000, true, false, 1 },     // CS A
        { "TCAA",       0050002, true, false, 1 },     // TS Z
        { "OVSK",       0050000, true, false, 1 },     // TS A
        { "DOUBLE",     0060000, true, false, 1 },     // AD A
        { "SQUARE",     0040000, true, false, 1 },     // MP A

                // For "clarity" (R393: 3-12)
        { "TCR",        0000000, true, true, 1 },      // same as TC; subroutine call with return
        { "CAF",        0030000, true, true, 1 },      // same as XCH; address to fixed acts like
Clear and Add
        { "OVIND",      0050000, true, true, 1 },      // same as TS

                // Assembler directives
        { "DS",         0, false, false, 1 },  // define storage; reserves 1 word of memory
        { "CADR",       0, false, false, 1 },  // define 14-bit addr; reserves 1 word of memory
        { "FCADR",      0, false, false, 1 },  // define 14-bit addr; reserves 1 word of memory
        { "ECADR",      0, false, false, 1 },  // define 14-bit addr; reserves 1 word of memory
        { "ADRES",      0, false, true,  1 },  // define 12-bit addr; reserves 1 word of memory
        { "ORG",        0, false, false, 0 },  // origin; sets location counter to operand value
        { "EQU",        0, false, false, 0 },  // equate; assigns a value to a label
        { "INCL",       0, false, false, 0 },  // include; inline include source code
        { "",           0, false, false, 99 }  // end-of-list flag
};

void parse(char* buf, char* labl, char* opcd, char* opnd, char* cmnt)
{
                // strip off newline char.
        buf[strlen(buf) - 1] = '\0';

                // replace any horizontal tabs with spaces
        for(unsigned i=0; i<strlen(buf); i++)
        {
                if(buf[i] == ';') break; // don't edit comments
                if(buf[i] == '\t') buf[i] = ' ';
        }

        strcpy(labl,"");
```

```
        strcpy(opcd,"");
        strcpy(opnd,"");
        strcpy(cmnt,"");

        char* sp = buf;
        char* s = 0;

        enum {_labl=0, _opcd, _opnd, _cmnt } mode = _labl;


        if(buf[0] == ' ') mode = _opcd;
        do
        {
                s = strtok(sp, " "); sp = 0;
                if(s)
                {
                        if(s[0] == ';')
                        {
                                // Copy the remainder of the comment verbatim.
                                strcat(cmnt, s); strcat(cmnt, " ");
                                s += strlen(s) + 1; strcat(cmnt, s);
                                return;

                        }
                        switch(mode)
                        {
                        case _labl: strcat(labl, s); mode = _opcd; break;
                        case _opcd: strcat(opcd, s); mode = _opnd; break;
                        case _opnd: strcat(opnd, s); mode = _cmnt; break;
                        }
                }
        } while(s);
}

struct symbl
{
        char name[20];
        unsigned val;
};


symbl symTab[5000];
unsigned nSym = 0;

        // Pre-defined symbols corresponding to architectural
        // conventions in the block I AGC4.
symbl constSymTab[] =
{
        { "A",                  00      },
        { "Q",                  01      },
        { "Z",                  02      },
        { "LP",                 03      },
        { "IN0",                04      },
        { "IN1",                05      },
        { "IN2",                06      },
        { "IN3",                07      },
        { "OUT0",               010     },
        { "OUT1",               011     },
        { "OUT2",               012     },
        { "OUT3",               013     },
        { "OUT4",               014     },
        { "BANK",               015     },
        { "CYR",                020     },
        { "SR",                 021     },
        { "CYL",                022     },
        { "SL",                 023     },
        { "ZRUPT",              024     },
        { "BRUPT",              025     },
        { "ARUPT",              026     },
        { "QRUPT",              027     },
        { "OVCTR",              034     },
        { "TIME2",              035     },
        { "TIME1",              036     },
        { "TIME3",              037     },
        { "TIME4",              040     },
        { "GOPROG",             02000   },
```

```
        { "T3RUPT",           02004  },
        { "ERRUPT",           02010  },
        { "DSRUPT",           02014  },
        { "KEYRUPT",          02020  },
        { "UPRUPT",           02024  },
        { "EXTENDER",         05777  },
        { "BANK0",            000057 }, // erasable memory, just above the counter assignments
        { "BANK1",            002000 }, // fixed-fixed
        { "BANK2",            004000 }, // fixed-fixed
        { "BANK3",            006000 }, // start of fixed-switchable
        { "BANK4",            010000 },
        { "BANK5",            012000 },
        { "BANK6",            014000 },
        { "BANK7",            016000 },
        { "BANK10",           020000 },
        { "BANK11",           022000 },
        { "BANK12",           024000 },
        { "BANK13",           026000 },
        { "BANK14",           030000 },
        { "",                 0      }
};

void add(char* labl, unsigned value)
{
                // Check whether symbol is already defined.
        unsigned i;
        for(i=0; i<nSym; i++)
        {
                if(strcmp(symTab[i].name, labl)==0)
                {
                        fprintf(fpList,"*** ERROR: %s redefined.\n", symTab[i].name);
                        errorCount++;
                        return;
                }
        }

                // Add new symbol to symbol table
        strcpy(symTab[nSym].name, labl);
        symTab[nSym].val = value;
        nSym++;
}


// Return the int value of the operand string. The string is
// assumed to be a simple value (not an expression)
int _getopnd(char* opnd)
{
        if(strlen(opnd)==0)
                return 0;
        else if(opnd[0] == '$') // hex number
                return strtol(opnd+1, 0, 16);
        else if(opnd[0] == '%') // octal number
                return strtol(opnd+1, 0, 8);
        else if(isdigit(opnd[0])) // decimal number
                return strtol(opnd, 0, 10);
        else if(opnd[0] == '*')
                return locCntr;
        else // must be label; look up value
        {
                unsigned i;
                for(i=0; i<nSym; i++)
                {
                        if(strcmp(symTab[i].name, opnd)==0)
                                return symTab[i].val;
                }

                // Not there, so check whether symbol is an
                // assembler-defined constant. If so, copy it to
                // the user-defined symbols.
                for(i=0; strcmp(constSymTab[i].name,"") != 0; i++)
                {
                        if(strcmp(constSymTab[i].name, opnd)==0)
                        {
                                strcpy(symTab[nSym].name, opnd);
                                symTab[nSym].val = constSymTab[i].val;
                                nSym++;
```

```
                                return constSymTab[i].val;
                        }
                }

                if(pass == 1)
                {
                        fprintf(fpList,"*** ERROR: %s undefined.\n", opnd);
                        errorCount++;
                }
        }
        return 0;
}

// returns pointer to new position in istr
char* getToken(char* istr, char* ostr)
{
        *ostr = '\0';

                // bump past any whitespace
        while(*istr == ' ') istr++;

        if(*istr == '\0') return istr;

        bool keepGoing = true;
        do
        {
                *ostr = *istr;
                if(*ostr == '+' || *ostr == '-' || *ostr == '@' || *ostr == '/')
                        keepGoing = false;
                ostr++; istr++;
        }
        while(keepGoing && *istr != '\0' && *istr != '+' && *istr != '-'
                && *istr != '@' && *istr != '/');

        *ostr = '\0';
        return istr;
}

int _eval(char* sp, int tot)
{
        if(*sp == '\0') return tot;

        char op[20];
        sp = getToken(sp, op);

        char vstr[20];
        int val = 0;
        sp = getToken(sp,vstr);
        if(*vstr =='-') // unary minus
        {
                sp = getToken(sp, vstr);
                val = -(_getopnd(vstr));
        }
        else
                val = _getopnd(vstr);

        switch(*op)
        {
        case '+': tot += val; break;
        case '-': tot -= val; break;
        case '@': tot *= val; break;
        case '/': tot /= val; break;
        }

        return _eval(sp,tot);
}

int eval(char* sp)
{
        char op[20];
        getToken(sp, op);
        char sp1[80];
        if(*op != '+' && *op != '-')
                strcpy(sp1,"+");
        else
                strcpy(sp1,"");
```

```
        strcat(sp1, sp);
        return _eval(sp1, 0);
}

// Return the value of the operand string. The string may
// be a simple token or an expression consisting of multiple
// tokens and operators. Evaluation occurs from left to right;
// no parenthesis allowed. Unary minus is allowed and correctly
// evaluated. Valid operators are +, -, @, and /. The @ operator
// is multiplication (the traditional * operator already is used
// to refer to the location counter.
unsigned getopnd(char* opnd)
{
        //return _getopnd(opnd); // the old call did not allow for expressions

        unsigned retval = 0;
        if(strcmp(opnd,"-0") == 0 || strcmp(opnd,"-%0") == 0 || strcmp(opnd,"-$0") == 0)
                retval = 077777;       // -0
        else
        {               // return the int value of the operand
                int opndVal = eval(opnd);

                        // now, convert the number into 16-bit signed AGC format
                if(opndVal < 0)
                {
                                // convert negative values into AGC 16-bit 1's C form.
                        opndVal = 077777 + opndVal;
                        if(opndVal < 0)
                        {
                                fprintf(fpList,"*** ERROR: %s underflowed.\n", opnd);
                                errorCount++;
                                opndVal = 0;
                        }
                }
                else if(opndVal > 077777)
                {
                        fprintf(fpList,"*** ERROR: %s overflowed.\n", opnd);
                        errorCount++;
                        opndVal = 0;
                }
                retval = (unsigned) opndVal;
        }
        return retval;
}

bool isDefined(char* opcd)
{
        for(int j=0; allcodes[j].len != 99; j++)
        {
                if(strcmp(allcodes[j].name, opcd) == 0)
                {
                        return true;
                }
        }
        return false;
}

unsigned getopcode(char* opcd)
{
        for(int j=0; allcodes[j].len != 99; j++)
        {
                if(strcmp(allcodes[j].name, opcd) == 0)
                {
                        return allcodes[j].code;
                }
        }
        fprintf(fpList,"*** ERROR: %s undefined.\n", opcd);
        errorCount++;

        return 0;
}

bool has12bitAddr(char* opcd)
{
        for(int j=0; allcodes[j].len != 99; j++)
```

```
	{
		if(strcmp(allcodes[j].name, opcd) == 0)
		{
			return allcodes[j].addrOpnd;
		}
	}
	return false;
}

bool isOpCode(char* opcd)
{
	for(int j=0; allcodes[j].len != 99; j++)
	{
		if(strcmp(allcodes[j].name, opcd) == 0)
		{
			return allcodes[j].isOpCode;
		}
	}
	return false;
}

unsigned getoplen(char* opcd)
{
	for(int j=0; allcodes[j].len != 99; j++)
	{
		if(strcmp(allcodes[j].name, opcd) == 0)
		{
			return allcodes[j].len;
		}
	}
	return 0;
}

void updateLocCntr(char* opcd, char* opnd)
{
	unsigned size = 0;
	for(int i=0; allcodes[i].len != 99; i++)
	{
		if(strcmp(allcodes[i].name, opcd) == 0)
		{
			size = allcodes[i].len; break;
		}
	}
	locCntr += size;

	if(strcmp(opcd,"ORG") == 0)
	{
		locCntr = getopnd(opnd);
	}
}

unsigned genOddParity(unsigned r)
{
	//check the lower 15 bits of 'r' and return the odd parity
	unsigned evenParity =
		(1&(r>>0))  ^ (1&(r>>1))  ^ (1&(r>>2))  ^ (1&(r>>3))  ^
		(1&(r>>4))  ^ (1&(r>>5))  ^ (1&(r>>6))  ^ (1&(r>>7))  ^
		(1&(r>>8))  ^ (1&(r>>9))  ^ (1&(r>>10)) ^ (1&(r>>11)) ^
		(1&(r>>12)) ^ (1&(r>>13)) ^ (1&(r>>14));
	return ~evenParity & 1; // odd parity
}


	// Read the source file and build the symbol table.
void readSourceForPass1(char* fn)
{
	char buf[256];
	char labl[100];		// label
	char opcd[100];		// op code
	char opnd[100];		// operand
	char cmnt[100];		// comment

			// Open the source code file.
	FILE* fp = fopen(fn, "r");
	if(!fp)
	{
```

```
                perror("fopen failed for source file");
                return;
        }

        while(fgets(buf, 256, fp))
        {
                parse(buf, labl, opcd, opnd, cmnt);

                if(strcmp(opcd,"INCL")==0)
                        readSourceForPass1(opnd);

                if(strlen(labl)>0)
                {
                        if(strcmp(opcd,"EQU")==0)
                                add(labl, getopnd(opnd));
                        else
                                add(labl, locCntr);
                }
                updateLocCntr(opcd, opnd);
        }
        fclose(fp);
}

        // Read the source file and symbol table and build
        // the object code
void readSourceForPass2(char* fn)
{
        char buf[256];
        char labl[100];       // label
        char opcd[100];       // op code
        char opnd[100];       // operand
        char cmnt[100];       // comment

                // Open the source code file.
        FILE* fp = fopen(fn, "r");
        if(!fp)
        {
                perror("fopen failed for source file");
                return;
        }

        while(fgets(buf,256,fp))
        {
                parse(buf, labl, opcd, opnd, cmnt);

                if(strcmp(opcd,"INCL")==0)
                {
                                // Include directive (INCL).
                        fprintf(fpList, "                          %-14s %-8s %-14s %s\n",
                                labl, opcd, opnd, cmnt);
                        readSourceForPass2(opnd);
                }
                else if(strcmp(opcd,"")==0)
                {
                                // Comment.
                        fprintf(fpList, "                          %s\n", cmnt);
                }
                else if(getoplen(opcd) == 0)
                {
                                // Must be ORG or EQU assembler directive.
                        fprintf(fpList, "                          %-14s %-8s %-14s %s\n",
                                labl, opcd, opnd, cmnt);

                        if(!isDefined(opcd))
                        {
                                fprintf(fpList,"*** ERROR: %s undefined.\n", opcd);
                                errorCount++;
                        }
                }
                else
                {
                                // Since we got this far, we know the assembly line contains a
                                // valid 'opcd' that reserves some storage space. It must be an
                                // instruction or a DS.

                                // Location counter (locCntr) contains 14-bit address; symbol table
```

```
        // also stores 14-bit addresses. If the operand is an address above
        // bank 3 (05777), it is not directly addressable and needs to be
        // converted into a 12-bit bank address. In bank addressing, bits
        // 12,11 are set and bits 10-1 contain the address inside the bank.
        // (The programmer must set the bank register to select the correct
        // bank.)

        // Generate a string containing the address (for the list file).
        // If the address is erasable, or fixed-fixed, show the 12-bit
        // address.
        // If the address is fixed-switchable, show the bank number,
        // followed by the 10-bit bank address.
unsigned locCntrBank = (036000 & locCntr) >> 10;
char locCntrString[20];
if(locCntrBank <= 3)
    sprintf(locCntrString, "   %04o", locCntr);
else
        sprintf(locCntrString, "%2o,%04o", locCntrBank, 01777 & locCntr);


        // Generate the data to be stored at that address. Convert to
        // 12-bit address format, if necessary.
unsigned operValue = getopnd(opnd);
unsigned operBank = (036000 & operValue) >> 10;
if(has12bitAddr(opcd))
{
                // Convert operand from a 14-bit address to a 12-bit
                // address.
                // First, find bank (bits 14-11). If the bank is <= 03, no
                // conversion is necessary.
        if(operBank > 03)
        {
                        // Bank is not directly addressable, so get 10 bit
                        // bank address and set fixed-switchable flag bits
                        // 12 and 11.
                operValue = (01777 & operValue) | 06000;
        }
}
unsigned data = getopcode(opcd) + operValue;
data |= genOddParity(data) << 15;

        // Generate a string containing the data info for the list file.
char dataString[20];
if(isOpCode(opcd))
        sprintf(dataString, "%01o %2o,%04o %1o",
                (getopcode(opcd) & 070000) >> 12, operBank, operValue,
                        genOddParity(data));
else
        sprintf(dataString, "    %05o %1o", operValue, genOddParity(data));


if(memoryUsed[locCntr])
{
        fprintf(fpList,"*** ERROR: %06o address already in use.\n",
                locCntr);
        errorCount++;
}
memoryUsed[locCntr] = true;


fprintf(fpList, "%05o %7s %11s %-14s %-8s %-14s %s\n",
                locCntr, locCntrString, dataString, labl, opcd, opnd, cmnt);

//*****************************************************************
// MODIFIED FROM THE ORIGINAL ASSEMBLER HERE.
// Insert the assembled code into an array, indexed by the address.
// This has the effect of sorting the data by address, so we can walk
// through the addresses and output the code to EPROM later.
unsigned dataLow  = 0x00ff & data;
unsigned dataHigh = (0xff00 & data) >> 8;

if(locCntr >=1024) // FIXED MEMORY only; not ERASEABLE
{
        // fixed memory
        EPROM_H [locCntr] = dataHigh;
        EPROM_L [locCntr] = dataLow;
```

```
                        }
                        //*********************************************************************

                }

                updateLocCntr(opcd, opnd);
        }
        fclose(fp);
}

//*****************************************************************
// MODIFIED FROM THE ORIGINAL ASSEMBLER HERE.

void writeEPROM(FILE* fpObj, int EPROM[])
{
                // Write an EPROM file using Motorola's S-Record format (s2f).

                // Some parameters that control file format. You can change maxBytes
                // without affecting anything else. 'addressBytes' is determined by
                // the choosen S-Record format.
        const int maxBytes = 20;  // set limit on record length
        const int addressBytes = 3; // 16-bit address range
        const int sumCheckBytes = 1;

        const int maxdata = maxBytes - addressBytes - sumCheckBytes;

        int i=0; // current EPROM address
        int sumCheck = 0;
        while (i < agcMemSize)
        {
                        // get dataByteCount; the number of bytes of EPROM data per record.
                int dataByteCount = maxdata;
                if(i + dataByteCount >= agcMemSize)
                {
                        dataByteCount = agcMemSize - i;
                }
                        // write record header (*** 2 byte address assumed ***)
                int totalByteCount = dataByteCount + addressBytes + sumCheckBytes;
                fprintf(fpObj, "S2%02X%06X", totalByteCount, i);
                sumCheck = totalByteCount & 0xff;
                sumCheck = (sumCheck + ((i & 0xff0000) >> 16)) % 256;
                sumCheck = (sumCheck + ((i & 0x00ff00) >>  8)) % 256;
                sumCheck = (sumCheck + ((i & 0x0000ff)      )) % 256;

                        // write data bytes into record
                for(int j=0; j<dataByteCount; j++)
                {
                        fprintf(fpObj, "%02X", EPROM [i+j]);
                        sumCheck = (sumCheck + EPROM [i+j]) % 256;

                }
                        // terminate record by adding the checksum and a newline.
                fprintf(fpObj, "%02X\n", (~sumCheck) & 0xff);

                i += dataByteCount;
        }
                // write an end-of-file record here
        i=0; // set address zero for last record
        sumCheck  = 0x04; // byte count
        sumCheck = (sumCheck + ((i & 0xff0000) >> 16)) % 256;
        sumCheck = (sumCheck + ((i & 0x00ff00) >>  8)) % 256;
        sumCheck = (sumCheck + ((i & 0x0000ff)      )) % 256;
        fprintf(fpObj, "S804%06X%02X", i, (~sumCheck) & 0xff);

}

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

void main(int argc, char* argv[])
{
        cout << "AGC Block I assembler" << endl;

                // The assembler reads an assembly source code file
                // with a .asm extension; i.e.: myProg.asm
                // It writes an assembly listing text file with
                // a .lst extension (myProg.lst) and an object code
```

```
		// text file with a .obj extension (myProg.obj)
#ifdef NOTDEF
	// use this to enter the source file using command line
	if(argc != 2)
	{
		cout << "*** ERROR: source file name not specified." << endl;
		exit(-1);
	}

	fp = fopen(argv[1], "r");
#endif

	char sourcefile[80];
	cout << "Enter source file: ";
	cin >> sourcefile;

		// Valid source files have a .asm extension; strip the
		// extension off so we can use the prefix for the list
		// and object files.
	char prefix[80];
	strcpy(prefix, sourcefile);

	char* p = prefix;
	while(*p != '\0') { p++; if(*p == '.') break; }
	if(strcmp(p,".asm") != 0)
	{
		cerr << "*** ERROR: Source file not *.asm" << endl;
		exit(-1);
	}
	*p = '\0';

		// Open a text file for the assembly listing. The filename
		// will have a .lst extension.
	char listfile[80];
	sprintf(listfile, "%s.lst", prefix);
	fpList = fopen(listfile, "w");
	if(!fpList)
	{
		perror("fopen failed for assembly list file");
		exit(-1);
	}

	//*****************************************************************
	// MODIFIED FROM THE ORIGINAL ASSEMBLER HERE.
		// Open a two text files for the object code. The filenames
		// will have a .hex extension
	char objfile[80];

	sprintf(objfile, "%s_H.hex", prefix);
	fpObj_H = fopen(objfile, "w");
	if(!fpObj_H)
	{
		perror("fopen failed for object file");
		exit(-1);
	}

	sprintf(objfile, "%s_L.hex", prefix);
	fpObj_L = fopen(objfile, "w");
	if(!fpObj_L)
	{
		perror("fopen failed for object file");
		exit(-1);
	}

	fprintf(fpList,"Block I Apollo Guidance Computer (AGC4) assembler version 1.6 for
EPROM\n\n");

		// INITIALIZE EPROM
	for(int k=0; k< agcMemSize; k++)
	{
		EPROM_H [k] = 0;
		EPROM_L [k] = 0;
	}
	//*****************************************************************

	fprintf(fpList,"First pass: generate symbol table.\n");
```

```
        readSourceForPass1(sourcefile);

        locCntr = 0;
        pass++;

                // Clear the memory use flags; these are used to catch
                // any overwriting of already assembled code.
        memset(memoryUsed, false, sizeof(bool) * agcMemSize);

        fprintf(fpList,"Second pass: generate object code.\n\n");
        readSourceForPass2(sourcefile);

        //*********************************************************************
        // MODIFIED FROM THE ORIGINAL ASSEMBLER HERE.
                // Write the EPROM data to file
        writeEPROM(fpObj_H, EPROM_H);
        writeEPROM(fpObj_L, EPROM_L);

        fclose(fpObj_H);
        fclose(fpObj_L);
        //*********************************************************************

        fprintf(fpList,"\nAssembly complete. Errors = %d\n", errorCount);

        fprintf(fpList,"\nSymbol table:\n");

        unsigned j=0;
        for(unsigned i=0; i<nSym; i++)
        {
                fprintf(fpList,"%-14s %06o   ", symTab[i].name, symTab[i].val);
                j = (j+1) % 3;
                if(j==0) fprintf(fpList,"\n");
        }
        fclose(fpList);
}
```