```c
/*
   alu.c
   - 21.11.05/BHO1
   bho1 29.12.2006
   bho1 6.12.2007
   bho1 30.11.2007 - clean up
   bho1 24.11.2009 - assembler instruction
   bho1 3.12.2009 - replaced adder with full_adder
   bho1 20.7.2011 - rewrite: minimize global vars, ALU-operations are modeled with ftc taking in/out register as parameter
   bho1 6.11.2011 - rewrite flags: adding flags as functional parameter. Now alu is truly a function


   GPL applies

   -->> YOUR FULL NAME HERE  <<--
*/

#include <stdio.h>
#include <string.h>

#include "alu.h"
#include "alu-opcodes.h"
#include "register.h"
#include "flags.h"
int const max_mue_memory = 100;

char mue_memory[100]= "100 Byte - this memory is at your disposal"; /*mue-memory */
char* m = mue_memory;

unsigned int c = 0;     /* carry bit address    */
unsigned int s = 1;         /* sum bit address      */
unsigned int c_in = 2;  /* carry in bit address */

/*
  testet ob alle bits im akkumulator auf null gesetzt sind.
  Falls ja wird 1 returniert, ansonsten 0
*/
int zero_test(char accumulator[]){
  int i;
  for(i=0;accumulator[i]!='\0'; i++){
        if(accumulator[i]!='0')
          return 0;
  }
  return 1;
}

/*
  Halfadder: addiert zwei character p,q und schreibt in
  den Mue-memory das summen-bit und das carry-bit.
*/
void half_adder(char p, char q){
  //your code here
}

/*
   void adder(char pbit, char qbit, char cbit)
   Adder oder auch Fulladder:
   Nimmt zwei character bits und ein carry-character-bit
   und schreibt das Resultat (summe, carry) in den Mue-speicher
*/
void full_adder(char pbit, char qbit, char cbit){
  //your code here
}

```



```c
/*
  Invertieren der Character Bits im Register reg
*/
void one_complement(char reg[]){
  //your code here
}

/*
  Das zweier-Komplement des Registers reg wird in reg geschrieben
  reg := K2(reg)
*/
void two_complement(char reg[]){
  //your code here
}


/*
  Die Werte in Register rega und Register regb werden addiert, das
  Resultat wird in Register accumulator geschrieben. Die Flags cflag,
  oflag, zflag und sflag werden entsprechend gesetzt

  accumulator := rega + regb
*/
void op_add(char rega[], char regb[], char accumulator[], char flags[]){
  //your code here
}

/*

  ALU_OP_ADD_WITH_CARRY

  Die Werte des carry-Flags und der Register rega und
  Register regb werden addiert, das
  Resultat wird in Register accumulator geschrieben. Die Flags cflag,
  oflag, zflag und sflag werden entsprechend gesetzt

  accumulator := rega + regb + carry-flag

*/
void op_addc(char rega[], char regb[], char accumulator[], char flags[]){
  //your code here
}

/*
  Die Werte in Register rega und Register regb werden subtrahiert, das
  Resultat wird in Register accumulator geschrieben. Die Flags cflag,
  oflag, zflag und sflag werden entsprechend gesetzt

  accumulator := rega - regb
*/
void op_sub(char rega[], char regb[], char accumulator[], char flags[]){
  //your code here
}

/*
  Die Werte in Register rega und Register regb werden logisch geANDet, das
  Resultat wird in Register accumulator geschrieben. Die Flags cflag,
  oflag, zflag und sflag werden entsprechend gesetzt

  accumulator := rega AND regb
*/
void op_and(char rega[], char regb[], char accumulator[], char flags[]){
  //your code here
}
/*
  Die Werte in Register rega und Register regb werden logisch geORt, das
```

```
131   Resultat wird in Register accumulator geschrieben. Die Flags cflag,
132   oflag, zflag und sflag werden entsprechend gesetzt
133
134   accumulator := rega OR regb
135 */
136 void op_or(char rega[], char regb[], char accumulator[], char flags[]){
137   //your code here
138 }
139 /*
140   Die Werte in Register rega und Register regb werden logisch geXORt, das
141   Resultat wird in Register accumulator geschrieben. Die Flags cflag,
142   oflag, zflag und sflag werden entsprechend gesetzt
143
144   accumulator := rega XOR regb
145 */
146 void op_xor(char rega[], char regb[], char accumulator[], char flags[]){
147   //your code here
148 }
149
150
151 /*
152   Einer-Komplement von Register rega
153   rega := not(rega)
154 */
155 void op_not_a(char rega[], char regb[], char accumulator[], char flags[]){
156   //your code here
157 }
158
159
160 /* Einer Komplement von Register regb */
161 void op_not_b(char rega[], char regb[], char accumulator[], char flags[]){
162   //your code here
163 }
164
165
166 /*
167   Negation von Register rega
168   rega := -rega
169 */
170 void op_neg_a(char rega[], char regb[], char accumulator[], char flags[]){
171   //your code here
172 }
173
174 /*
175   Negation von Register regb
176   regb := -regb
177 */
178 void op_neg_b(char rega[], char regb[], char accumulator[], char flags[]){
179   //your code here
180 }
181
182
183 /*
184   clear mue_memory
185 */
186 void alu_reset(){
187   int i;
188
189   for(i=0;i<max_mue_memory;i++)
190     m[i] = '0';
191 }
192
193 /*
194   alu function
195   Needed register are already alocated and may be modified
196   mainly a switchboard
```



```
197
198   alu_fct(int opcode, char reg_in_a[], char reg_in_b[], char reg_out_accu[], cha
    r flags[])
199
200 */
201 void alu(unsigned int alu_opcode, char reg_in_a[], char reg_in_b[], char reg_out
    _accu[], char flags[]){
202
203   switch ( alu_opcode ){
204   case ALU_OP_ADD :
205     op_add(reg_in_a, reg_in_b, reg_out_accu, flags);
206     break;
207   case ALU_OP_ADD_WITH_CARRY :
208         op_addc(reg_in_a, reg_in_b, reg_out_accu, flags);
209         break;
210   case ALU_OP_SUB :
211     op_sub(reg_in_a, reg_in_b, reg_out_accu, flags);
212     break;
213   case ALU_OP_AND :
214     op_and(reg_in_a, reg_in_b, reg_out_accu, flags);
215     break;
216   case ALU_OP_OR:
217     op_or(reg_in_a, reg_in_b, reg_out_accu, flags);
218     break;
219   case ALU_OP_XOR :
220     op_xor(reg_in_a, reg_in_b, reg_out_accu, flags);
221     break;
222   case ALU_OP_NEG_A :
223     op_neg_a(reg_in_a, reg_in_b, reg_out_accu, flags);
224     break;
225   case ALU_OP_NEG_B :
226     op_neg_b(reg_in_a, reg_in_b, reg_out_accu, flags);
227     break;
228   case ALU_OP_NOT_A :
229     op_not_a(reg_in_a, reg_in_b, reg_out_accu, flags);
230     break;
231   case ALU_OP_NOT_B :
232     op_not_b(reg_in_a, reg_in_b, reg_out_accu, flags);
233     break;
234   case ALU_OP_RESET :
235         alu_reset();
236         break;
237   default:
238     printf("ALU(%i): Invalide operation %i selected", alu_opcode, alu_opcode);
239   }
240 }
```