```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 fct taking
  in/out register as parameter
  bho1 6.11.2011 - rewrite flags: adding flags as functional parameter. Now alu is truly a
  function
  bho1 26.11.2012 - remove bit declaration from op_alu_asl and op_alu_ror as they are unused
  (this may change later)


  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;
}

/*
```

```c
  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){
  if(p=='0' && q=='0'){
    m[c] = '0';
    m[s] = '0';
  }
  else if(p=='1' && q=='1'){
    m[c] = '1';
    m[s] = '0';
  }
  else {
    m[c] = '0';
    m[s] = '1';
  }
}

/*
  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){
  half_adder(pbit, qbit);
  if(m[c] == '0' ){
    half_adder(m[s], cbit);
  } else {
    m[s] = cbit;
  }
}

/*
  Invertieren der Character Bits im Register reg
*/
void one_complement(char reg[]){
  int i;

  for(i=0; reg[i] != 0; i++){
    reg[i] = (reg[i]=='0') ? '1' : '0';
  }
}

/*
  Das zweier-Komplement des Registers reg wird in reg geschrieben
  reg := K2(reg)
*/
void two_complement(char reg[]){
  int i;

  one_complement(reg);
  for(i=REG_WIDTH-1; i >= 0; i--){
    if(reg[i] == '1'){
      reg[i] = '0';
    }
    else if(reg[i] == '0'){
      reg[i] = '1';
      break;
    }
  }
}

void set_flags(char rega[], char regb[], char accumulator[], char flags[]){
  // carry-flag
  if(m[c] == '1'){
    setCarryflag(flags);
  }
  else {
    clearCarryflag(flags);
  }

  //overflow-flag
  if((rega[0] == '0' && regb[0] == '0' && accumulator[0] == '1') ||
     (rega[0] == '1' && regb[0] == '1' && accumulator[0] == '0')) {

    setOverflowflag(flags);
  }
  else {
    clearOverflowflag(flags);
  }

  //zero-flag
  if(zero_test(accumulator) == 1){
    setZeroflag(flags);
  }
  else {
    clearZeroflag(flags);
  }

  //signed-flag
  if(accumulator[0] == '1'){
    setSignflag(flags);
  }
  else {
    clearSignflag(flags);
  }

}

/*
  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[]){
  int i;
  m[c] = '0';

  for(i=REG_WIDTH-1; i >= 0; i--){
    full_adder(rega[i], regb[i], m[c]);
    accumulator[i] = m[s];
  }

  set_flags(rega, regb, accumulator, flags);
}

/*

  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_adc(char rega[], char regb[], char accumulator[], char flags[]){
  int i;
  m[c] = getCarryflag(flags);

  for(i=REG_WIDTH-1; i >= 0; i--){
    full_adder(rega[i], regb[i], m[c]);
    accumulator[i] = m[s];
  }

  set_flags(rega, regb, accumulator, flags);

}

/*
  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[]){
  int i;
  m[c] = '0';

  two_complement(regb);

  for(i=REG_WIDTH-1; i >= 0; i--){
    full_adder(rega[i], regb[i], m[c]);
    accumulator[i] = m[s];
  }

  set_flags(rega, regb, accumulator, flags);

}

/*
  subtract with carry
  SBC
  laubr: muss man nicht machen
*/
void op_alu_sbc(char rega[], char regb[], char accumulator[], char flags[]){
  int i;
  m[c] = getCarryflag(flags);

  two_complement(regb);

  for(i=REG_WIDTH-1; i >= 0; i--){
    full_adder(rega[i], regb[i], m[c]);
    accumulator[i] = m[s];
  }

  set_flags(rega, regb, accumulator, flags);

}


/*
  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[]){
  int i;

  for(i=REG_WIDTH-1; i >= 0; i--){
    accumulator[i] = (rega[i] == '1' && regb[i] == '1') ? '1' : '0';
  }

  set_flags(rega, regb, accumulator, flags);
}
/*
  Die Werte in Register rega und Register regb werden logisch geORt, das
  Resultat wird in Register accumulator geschrieben. Die Flags cflag,
  oflag, zflag und sflag werden entsprechend gesetzt

  accumulator := rega OR regb
*/
void op_or(char rega[], char regb[], char accumulator[], char flags[]){
```

```
  int i;

  for(i=REG_WIDTH-1; i >= 0; i--){
    accumulator[i] = (rega[i] == '0' && regb[i] == '0') ? '0' : '1';
  }

  set_flags(rega, regb, accumulator, flags);

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

  accumulator := rega XOR regb
*/
void op_xor(char rega[], char regb[], char accumulator[], char flags[]){
  int i;

  for(i=REG_WIDTH-1; i >= 0; i--){
    accumulator[i] = (rega[i] != regb[i]) ? '1' : '0';
  }

  set_flags(rega, regb, accumulator, flags);

}


/*
  Einer-Komplement von Register rega
  rega := not(rega)
*/
void op_not_a(char rega[], char regb[], char accumulator[], char flags[]){
  one_complement(rega);
  set_flags(rega, regb, accumulator, flags);
}


/* Einer Komplement von Register regb */
void op_not_b(char rega[], char regb[], char accumulator[], char flags[]){
  one_complement(regb);
  set_flags(rega, regb, accumulator, flags);
}


/*
  Negation von Register rega
  rega := -rega
*/
void op_neg_a(char rega[], char regb[], char accumulator[], char flags[]){
  two_complement(rega);
  set_flags(rega, regb, accumulator, flags);
}
```

```c
/*
  Negation von Register regb
  regb := -regb
*/
void op_neg_b(char rega[], char regb[], char accumulator[], char flags[]){
  two_complement(regb);
  set_flags(rega, regb, accumulator, flags);
}

/*
  arithmetic shift left
  asl
*/
void op_alu_asl(char regina[], char reginb[], char regouta[], char flags[]){

}


/*
  logical shift right
  lsr
*/
void op_alu_lsr(char regina[], char reginb[], char regouta[], char flags[]){

}
/*
  rotate
  rotate left
*/
void op_alu_rol(char regina[], char reginb[], char regouta[], char flags[]){

}
/*
  rotate
  rotate left
  Move each of the bits in  A one place to the right. Bit 7 is filled with
  the current value of the carry flag whilst the old bit 0 becomes the new carry flag value.
*/
void op_alu_ror(char regina[], char reginb[], char regouta[], char flags[]){

}


/*
  clear mue_memory
*/
void alu_reset(){
  int i;

  for(i=0;i<max_mue_memory;i++)
    m[i] = '0';
}
```

```c
/*

  Procedural approach to ALU with side-effect:
  Needed register are already alocated and may be modified
  mainly a switchboard

  alu_fct(int opcode, char reg_in_a[], char reg_in_b[], char reg_out_accu[], char flags[])

*/
void alu(unsigned int alu_opcode, char reg_in_a[], char reg_in_b[], char reg_out_accu[], char
flags[]){
  char dummyflags[9] = "00000000";
  switch ( alu_opcode ){
  case ALU_OP_ADD :
    op_add(reg_in_a, reg_in_b, reg_out_accu, (flags==NULL)?dummyflags:flags);
    break;
  case ALU_OP_ADD_WITH_CARRY :
    op_adc(reg_in_a, reg_in_b, reg_out_accu, (flags==NULL)?dummyflags:flags);
    break;
  case ALU_OP_SUB :
    op_sub(reg_in_a, reg_in_b, reg_out_accu, (flags==NULL)?dummyflags:flags);
    break;
  case ALU_OP_SUB_WITH_CARRY :
    op_alu_sbc(reg_in_a, reg_in_b, reg_out_accu, (flags==NULL)?dummyflags:flags);
    break;
  case ALU_OP_AND :
    op_and(reg_in_a, reg_in_b, reg_out_accu, (flags==NULL)?dummyflags:flags);
    break;
  case ALU_OP_OR:
    op_or(reg_in_a, reg_in_b, reg_out_accu, (flags==NULL)?dummyflags:flags);
    break;
  case ALU_OP_XOR :
    op_xor(reg_in_a, reg_in_b, reg_out_accu, (flags==NULL)?dummyflags:flags);
    break;
  case ALU_OP_NEG_A :
    op_neg_a(reg_in_a, reg_in_b, reg_out_accu, (flags==NULL)?dummyflags:flags);
    break;
  case ALU_OP_NEG_B :
    op_neg_b(reg_in_a, reg_in_b, reg_out_accu, (flags==NULL)?dummyflags:flags);
    break;
  case ALU_OP_NOT_A :
    op_not_a(reg_in_a, reg_in_b, reg_out_accu, (flags==NULL)?dummyflags:flags);
    break;
  case ALU_OP_NOT_B :
    op_not_b(reg_in_a, reg_in_b, reg_out_accu, (flags==NULL)?dummyflags:flags);
    break;
  case ALU_OP_ASL :
    op_alu_asl(reg_in_a, reg_in_b, reg_out_accu, (flags==NULL)?dummyflags:flags);
    break;
  case ALU_OP_LSR :
    op_alu_lsr(reg_in_a, reg_in_b, reg_out_accu, (flags==NULL)?dummyflags:flags);
    break;
  case ALU_OP_ROL:
    op_alu_rol(reg_in_a, reg_in_b, reg_out_accu, (flags==NULL)?dummyflags:flags);
    break;
  case ALU_OP_ROR:
    op_alu_ror(reg_in_a, reg_in_b, reg_out_accu, (flags==NULL)?dummyflags:flags);
    break;
  case ALU_OP_RESET :
    alu_reset();
    break;
  default:
    printf("ALU(%i): Invalide operation %i selected", alu_opcode, alu_opcode);
  }
}
```