```
/*
   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 wi
th 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 th
ey are unused (this may change later)
   bho1 20.9.2014 cleaned


   GPL applies

   -->> Jan Scheidegger <<--
   */

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

#include "alu.h"
#include "alu-opcodes.h"
#include "register.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 */

/*
   clear mue_memory
   */
void alu_reset(){
    int i;
    for(i=0;i<max_mue_memory;i++)
        m[i] = '0';
}
void clearArray(char accumulator[]) {
    int i;
    for(i =0;i<REG_WIDTH;i++) accumulator[i] = '0';
}

/*
   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;
```



```
}

void zsflagging(char* flags,char *acc){
    //Zeroflag
    if(zero_test(acc))
        setZeroflag(flags);
    else
        clearZeroflag(flags);

    //Signflag
    if(acc[0] == '1')
        setSignflag(flags);
    else
        clearSignflag(flags);
}

/*
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){
    char result, carry;
    if (p == '0' && q == '0') {
        result = '0';
        carry = '0';
    }else if(p=='0' && q=='1') {
        result = '1';
        carry = '0';
    }else if(p=='1' && q=='0') {
        result = '1';
        carry = '0';
    } else if(p=='1' && q=='1') {
        result = '0';
        carry = '1';
    }

    m[c] = carry;
    m[s] = result;
}



/*
   Reset ALU
   resets registers and calls alu_op_reset
   */
void op_alu_reset(char rega[], char regb[], char accumulator[], char flags[]){
    int i;
    alu_reset();
    for(i=0; i<REG_WIDTH; i++){
        rega[i] = '0';
        regb[i] = '0';
        accumulator[i] = '0';
        flags[i] = '0';
    }
}

/*
   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){
    char carry1, carry2, result;
    half_adder(pbit, qbit);
    carry1 = m[c];
    half_adder(m[s], cbit);
    carry2 = m[c];
    result = m[s];
    if(carry1 == '1' || carry2 == '1') {

        m[c] = '1';
    }
    else {
        m[c] = '0';
    }
}

/*
   Invertieren der Character Bits im Register reg
   one_complement(char reg[]) --> NOT(reg)
   */

void one_complement(char reg[]){
    int i;
    for(i=0; i<REG_WIDTH; i++) {
        reg[i] = (reg[i] == '1' ? '0' : '1');
    }
}

/*
   Das zweier-Komplement des Registers reg wird in reg geschrieben
reg := K2(reg)
*/
void two_complement(char reg[]){
    int i;
    one_complement(reg);
    m[c] = '1';
    for(i = REG_WIDTH -1; i>=0; i--) {
        if(reg[i] == '0') {
            reg[i] = '1';
            m[c] = '0';
            break;
        }
        else {
            reg[i] = '0';
        }
    }
}


/*
   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[]){
    clearCarryflag(flags);
    op_adc(rega, regb, accumulator, flags);
}
```



```
/*
   ALU_OP_ADD_WITH_CARRY

   Die Werte des carry-Flags und der Register rega und
   Register regb werden addiert, das
   printf("%c %c %c", carry1, carry2, result);
   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[]){
    m[c] = getCarryflag(flags);
    int i;
    for(i=REG_WIDTH -1; i>=0;i--) {
        full_adder(rega[i], regb[i], m[c]);
        accumulator[i] = m[s];
    }
    m[c] == '1' ? setCarryflag(flags) : clearCarryflag(flags);
    if ((rega[0] == '1' && regb[0] == '1' && accumulator[0] == '0') || (rega[0]
== '0' && regb[0] == '0' && accumulator[0] == '1'))
    {
        setOverflowflag(flags);
    }
    else {
        clearOverflowflag(flags);
    }
    zsflagging(flags, accumulator);

}

/*
   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 = rega + NOT(regb) + 1
*/
void op_sub(char rega[], char regb[], char accumulator[], char flags[]){
    clearArray(accumulator);
    clearOverflowflag(flags);
    char temp[REG_WIDTH];
    int i;
    for(i = 0; i<REG_WIDTH;i++) temp[i] = regb[i];
    two_complement(temp);
    char carry = m[c];
    op_add(rega, temp, accumulator, flags);
    zsflagging(flags, accumulator);
    if(carry == '1')
        setCarryflag(flags);
    if ((rega[0] == '1' && temp[0] == '1' && accumulator[0] == '0') || (rega[0]
== '0' && temp[0] == '0' && accumulator[0] == '1'))
    {
        setOverflowflag(flags);
    }
    else {
        clearOverflowflag(flags);
    }
}

/*
```

```
   subtract with carry
   SBC
   accumulator =
   a − b − !c  =
   a − b − !c + 256 =
   a − b − (1−c) + 256 =
   a + (255 − b) + c =
   a + !b + c
accumulator := rega − regb = rega + NOT(regb) +carryflag

*/
void op_alu_sbc(char rega[], char regb[], char accumulator[], char flags[]){
}


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

accumulator := rega AND regb
*/
void op_and(char rega[], char regb[], char accumulator[], char flags[]){
    clearArray(accumulator);
    int i;
    for(i=REG_WIDTH −1; i>=0;i−−) {
        if(rega[i] == '1' && regb[i] == '1')
            accumulator[i] = '1';
        else
            accumulator[i] = '0';
    }
    zsflagging(flags, accumulator);
}
/*
   Die Werte in Register rega und Register regb werden logisch geORt,
   das Resultat wird in Register accumulator geschrieben.
   Die Flags zflag und sflag werden entsprechend gesetzt

accumulator := rega OR regb
*/
void op_or(char rega[], char regb[], char accumulator[], char flags[]){
    clearArray(accumulator);
    int i;
    for(i=REG_WIDTH −1; i>=0;i−−) {
        if(rega[i] == '1' || regb[i] == '1')
            accumulator[i] = '1';
        else
            accumulator[i] = '0';
    }
    zsflagging(flags, accumulator);
}
/*
   Die Werte in Register rega und Register regb werden logisch geXORt,
   das Resultat wird in Register accumulator geschrieben.
   Die Flags zflag und sflag werden entsprechend gesetzt

accumulator := rega XOR regb
*/
void op_xor(char rega[], char regb[], char accumulator[], char flags[]){
    clearArray(accumulator);
    int i;
    for(i=REG_WIDTH −1; i>=0;i−−) {
```



```
        if(rega[i] ^ regb[i])
            accumulator[i] = '1';
        else
            accumulator[i] = '0';
    }
    zsflagging(flags, accumulator);
}


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


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


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

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

/*
   bit −>   7                           0
   +−−−+−−−+−−−+−−−+−−−+−−−+−−−+−−−+
   carryflag <−− |   |   |   |   |   |   |   |   |  <−− 0
   +−−−+−−−+−−−+−−−+−−−+−−−+−−−+−−−+

   arithmetic shift left
   asl

    ASL Arithmetic Shift Left; Motorola 680x0, Motorola 68300; shifts the conten
ts of a data register (8, 16, or 32 bits) or memory location (16 bits) to the le
ft (towards most significant bit) by a specified amount (by 1 to 8 bits for an i
mmediate operation on a data register, by the contents of a data register modulo
 64 for a data register, or by 1 bit only for a memory location), with the high−
order bit being shifted into the carry and extend flags, zeros shifted into the
low−order bit, overflow flag indicating a change of sign; sets or clear flags
   */
void op_alu_asl(char regina[], char reginb[], char regouta[], char flags[]){
    int i;
    regouta[0] == '1' ? setCarryflag(flags) : clearCarryflag(flags);
    for(i=1;i<REG_WIDTH;i++) {
        regouta[i−1] = regouta[i];
```

```
    }
    regouta[REG_WIDTH−1] = ’0’;
    zsflagging(flags, regouta);
}


/*
   logical shift right
   lsr
   */
void op_alu_lsr(char regina[], char reginb[], char regouta[], char flags[]){
    int i;
    for(i=REG_WIDTH−1;i>=1;i−−) {
        regouta[i] = regouta[i−1];
    }
    regouta[0] = ’0’;
    zsflagging(flags, regouta);
}
/*
   rotate
   rotate left
   */
void op_alu_rol(char regina[], char reginb[], char regouta[], char flags[]){
    char old_carry = getCarryflag(flags);
    regouta[0] == ’1’ ? setCarryflag(flags) : clearCarryflag(flags);
    int i;
    for(i = 0; i<REG_WIDTH−1;i++) regouta[i] = regouta[i+1];
    regouta[REG_WIDTH−1] = old_carry;

}

/*
   rotate
   rotate right
   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[]){
    char old_carry = getCarryflag(flags);
    regouta[REG_WIDTH−1] == ’1’ ? setCarryflag(flags) : clearCarryflag(flags);
    int i;
    for(i = REG_WIDTH−1; i>0;i−−) regouta[i] = regouta[i−1];
    regouta[0] = old_carry;
}


/*


   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[], ch
ar 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:fl
ags);
            break;
        case ALU_OP_ADD_WITH_CARRY :
            op_adc(reg_in_a, reg_in_b, reg_out_accu, (flags==NULL)?dummyflags:fl
ags);
            break;
        case ALU_OP_SUB :
            op_sub(reg_in_a, reg_in_b, reg_out_accu, (flags==NULL)?dummyflags:fl
ags);
            break;
        case ALU_OP_SUB_WITH_CARRY :
            op_alu_sbc(reg_in_a, reg_in_b, reg_out_accu, (flags==NULL)?dummyflag
s:flags);
            break;
        case ALU_OP_AND :
            op_and(reg_in_a, reg_in_b, reg_out_accu, (flags==NULL)?dummyflags:fl
ags);
            break;
        case ALU_OP_OR:
            op_or(reg_in_a, reg_in_b, reg_out_accu, (flags==NULL)?dummyflags:fla
gs);
            break;
        case ALU_OP_XOR :
            op_xor(reg_in_a, reg_in_b, reg_out_accu, (flags==NULL)?dummyflags:fl
ags);
            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)?dummyflag
s:flags);
            break;
        case ALU_OP_LSR :
            op_alu_lsr(reg_in_a, reg_in_b, reg_out_accu, (flags==NULL)?dummyflag
s:flags);
            break;
        case ALU_OP_ROL:
            op_alu_rol(reg_in_a, reg_in_b, reg_out_accu, (flags==NULL)?dummyflag
s:flags);
            break;
        case ALU_OP_ROR:
            op_alu_ror(reg_in_a, reg_in_b, reg_out_accu, (flags==NULL)?dummyflag
s:flags);
            break;
        case ALU_OP_RESET :
            op_alu_reset(reg_in_a, reg_in_b, reg_out_accu, (flags==NULL)?dummyfl
ags:flags);
```

```
            break;
        default:
            printf("ALU(%i): Invalide operation %i selected", alu_opcode, alu_opcode);
    }
}
```