```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)
   bho1 20.9.2014 cleaned

   GPL applies

   -->>  Marco Schmid  <<--
   */

#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"; /*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 */

void alu_reset()
{
    int i;
    for(i=0;i<max_mue_memory;i++)
        m[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);
```



```c

    //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 = '0';
    char carry = '0';
    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();

    /* clear rega, regb, accumulator, flags */
    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
   */
```

```
130 void full_adder(char pbit, char qbit, char cbit)
131 {
132     half_adder(pbit, qbit);
133     char carry1 = m[c];
134     half_adder(m[s], cbit);
135     if (carry1 == '1')
136     {
137         m[c] = '1';
138     }
139 }
140
141 /*
142    Invertieren der Character Bits im Register reg
143    one_complement(char reg[]) --> NOT(reg)
144    */
145 void one_complement(char reg[])
146 {
147     int i = 0;
148     for (i = 7; i >= 0 ; i--)
149     {
150         if (reg[i] == '1')
151             reg[i] = '0';
152         else
153             reg[i] = '1';
154     }
155 }
156
157 /*
158    Das zweier-Komplement des Registers reg wird in reg geschrieben
159 reg := K2(reg)
160 */
161 void two_complement(char reg[])
162 {
163     int i = 0;
164     one_complement(reg);
165     m[c] = '1';
166     for (i = 7; i >= 0; i--)
167     {
168         if (reg[i] == '0')
169         {
170             reg[i] = '1';
171             m[c] = '0';
172             break;
173         }
174         else
175         {
176             reg[i] = '0';
177         }
178     }
179 }
180
181 /*
182    Die Werte in Register rega und Register regb werden addiert, das
183    Resultat wird in Register accumulator geschrieben. Die Flags cflag,
184    oflag, zflag und sflag werden entsprechend gesetzt
185
186 accumulator := rega + regb
187 */
188 void op_add(char rega[], char regb[], char accumulator[], char flags[])
189 {
190     alu_reset();
191     clearCarryflag(flags);
192     clearOverflowflag(flags);
193     int i = 0;
194
195     for (i = 7; i >= 0; i--)
```



```
196     {
197         full_adder(rega[i], regb[i], m[c]);
198         accumulator[i] = m[s];
199     }
200
201     if ((rega[0] == '1' && regb[0] == '1' && accumulator[0] == '0') ||
202             (rega[0] == '0' && regb[0] == '0' && accumulator[0] == '1'))
203     {
204         setOverflowflag(flags);
205     }
206     if (m[c] == '1')
207         setCarryflag(flags);
208
209     zsflagging(flags, accumulator);
210 }
211
212 /*
213
214    ALU_OP_ADD_WITH_CARRY
215
216    Die Werte des carry-Flags und der Register rega und
217    Register regb werden addiert, das
218    Resultat wird in Register accumulator geschrieben. Die Flags cflag,
219    oflag, zflag und sflag werden entsprechend gesetzt
220
221 accumulator := rega + regb + carry-flag
222
223 */
224 void op_adc(char rega[], char regb[], char accumulator[], char flags[])
225 {
226     char carry;
227     carry = m[c];
228     op_add(rega, regb, accumulator, flags);
229     if (carry == '1')
230     {
231         char temp[8];
232         char one[8] = {'0', '0', '0', '0', '0', '0', '0', '1'};
233         strcpy(temp, accumulator);
234         op_add(temp, one, accumulator, flags);
235     }
236 }
237
238 /*
239    Die Werte in Register rega und Register regb werden subtrahiert, das
240    Resultat wird in Register accumulator geschrieben. Die Flags cflag,
241    oflag, zflag und sflag werden entsprechend gesetzt
242
243 accumulator := rega - regb = rega + NOT(regb) + 1
244 */
245 void op_sub(char rega[], char regb[], char accumulator[], char flags[])
246 {
247     char temp[8];
248     int i = 0;
249     for (i = 0; i < 8; i++)
250     {
251         temp[i] = regb[i];
252     }
253     two_complement(regb);
254     char carry = m[c];
255
256     op_add(rega, regb, accumulator, flags);
257
258     for (i = 0; i < 8; i++)
259     {
260         regb[i] = temp[i];
261     }
```

```
262     // Overflow for subtraction : 0 && 1 && 1 or 1 && 0 && 0
263     if ((rega[0] == '0' && regb[0] == '1' && accumulator[0] == '1') ||
264             (rega[0] == '1' && regb[0] == '0' && accumulator[0] == '0'))
265     {
266         setOverflowflag(flags);
267     }
268     else
269     {
270         clearOverflowflag(flags);
271     }
272
273     if (carry == '1')
274         setCarryflag(flags);
275 }
276
277 /*
278    subtract with carry
279    SBC
280    accumulator =
281    a - b - !c  =
282    a - b - !c + 256 =
283    a - b - (1-c) + 256 =
284    a + (255 - b) + c =
285    a + !b + c
286 accumulator := rega - regb = rega + NOT(regb) +carryflag
287
288 */
289 void op_alu_sbc(char rega[], char regb[], char accumulator[], char flags[])
290 {
291     char carry = m[c];
292     op_sub(rega, regb, accumulator, flags);
293     if (carry == '1')
294     {
295         char temp[8];
296         char one[8] = {'0', '0', '0', '0', '0', '0', '0', '1'};
297         strcpy(temp, accumulator);
298         op_add(temp, one, accumulator, flags);
299     }
300 }
301
302 /*
303    Die Werte in Register rega und Register regb werden logisch geANDet,
304    das Resultat wird in Register accumulator geschrieben.
305    Die Flags zflag und sflag werden entsprechend gesetzt
306
307 accumulator := rega AND regb
308 */
309 void op_and(char rega[], char regb[], char accumulator[], char flags[])
310 {
311     int i = 0;
312     for (i = 0; i < 8; i++)
313     {
314         if (rega[i] == '1' && regb[i] == '1')
315             accumulator[i] = '1';
316         else
317             accumulator[i] = '0';
318     }
319     zsflagging(flags, accumulator);
320 }
321
322 /*
323    Die Werte in Register rega und Register regb werden logisch geORt,
324    das Resultat wird in Register accumulator geschrieben.
325    Die Flags zflag und sflag werden entsprechend gesetzt
326
327 accumulator := rega OR regb
```



```
328 */
329 void op_or(char rega[], char regb[], char accumulator[], char flags[])
330 {
331     int i = 0;
332     for (i = 0; i < 8; i++)
333     {
334         if (rega[i] == '1' || regb[i] == '1')
335             accumulator[i] = '1';
336         else
337             accumulator[i] = '0';
338     }
339     zsflagging(flags, accumulator);
340 }
341
342 /*
343    Die Werte in Register rega und Register regb werden logisch geXORt,
344    das Resultat wird in Register accumulator geschrieben.
345    Die Flags zflag und sflag werden entsprechend gesetzt
346
347 accumulator := rega XOR regb
348 */
349 void op_xor(char rega[], char regb[], char accumulator[], char flags[])
350 {
351     int i = 0;
352     for (i = 0; i < 8; i++)
353     {
354         if ((rega[i] == '1' && regb[i] == '0') ||
355                 (rega[i] == '0' && regb[i] == '1'))
356             accumulator[i] = '1';
357         else
358             accumulator[i] = '0';
359     }
360     zsflagging(flags, accumulator);
361 }
362
363 /*
364    Einer-Komplement von Register rega
365 rega := not(rega)
366 */
367 void op_not_a(char rega[], char regb[], char accumulator[], char flags[])
368 {
369     one_complement(rega);
370 }
371
372 /* Einer Komplement von Register regb */
373 void op_not_b(char rega[], char regb[], char accumulator[], char flags[])
374 {
375     one_complement(regb);
376 }
377
378 /*
379    Negation von Register rega
380 rega := -rega
381 */
382 void op_neg_a(char rega[], char regb[], char accumulator[], char flags[])
383 {
384     two_complement(rega);
385 }
386
387 /*
388    Negation von Register regb
389 regb := -regb
390 */
391 void op_neg_b(char rega[], char regb[], char accumulator[], char flags[])
392 {
393     two_complement(regb);
```

```
394 }
395
396 /*
397    bit ->   7                         0
398    +---+---+---+---+---+---+---+---+
399    carryflag <-- |   |   |   |   |   |   |   |   |  <-- 0
400    +---+---+---+---+---+---+---+---+
401
402    arithmetic shift left
403    asl
404    */
405 void op_alu_asl(char regina[], char reginb[], char regouta[], char flags[])
406 {
407     int i = 0;
408     if (regina[0] == '1')
409         setCarryflag(flags);
410     else
411         clearCarryflag(flags);
412
413     for (i = 7; i >= 0; i--)
414     {
415         int dest = i-1;
416         if (dest >= 0)
417             regouta[dest] = regina[i];
418     }
419     regouta[7] = '0';
420 }
421
422 /*
423    logical shift right
424    lsr
425    */
426 void op_alu_lsr(char regina[], char reginb[], char regouta[], char flags[])
427 {
428     int i = 0;
429     for (i = 0; i < 8; i++)
430     {
431         int dest = i+1;
432         if (dest < 8)
433             regouta[dest] = regina[i];
434     }
435
436     if (getCarryflag(flags) == '1')
437         regouta[0] = '1';
438     else
439         regouta[0] = '0';
440 }
441
442 /*
443    rotate
444    rotate left
445    */
446 void op_alu_rol(char regina[], char reginb[], char regouta[], char flags[])
447 {
448     char temp = getCarryflag(flags);
449     op_alu_asl(regina, reginb, regouta, flags);
450     regouta[7] = temp;
451 }
452
453 /*
454    rotate
455    rotate left
456    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.
457    */
```



```
458 void op_alu_ror(char regina[], char reginb[], char regouta[], char flags[])
459 {
460     char temp = regina[7];
461     op_alu_asl(regina, reginb, regouta, flags);
462     regouta[7] = temp;
463     if (temp == '1')
464         setCarryflag(flags);
465     else
466         clearCarryflag(flags);
467 }
468
469 /*
470
471    Procedural approach to ALU with side-effect:
472    Needed register are already alocated and may be modified
473    mainly a switchboard
474
475    alu_fct(int opcode, char reg_in_a[], char reg_in_b[], char reg_out_accu[], ch
    ar flags[])
476
477 */
478 void alu(unsigned int alu_opcode, char reg_in_a[], char reg_in_b[], char reg_out
    _accu[], char flags[])
479 {
480     char dummyflags[9] = "00000000";
481     switch ( alu_opcode ){
482         case ALU_OP_ADD :
483             op_add(reg_in_a, reg_in_b, reg_out_accu, (flags==NULL)?dummyflags:fl
    ags);
484             break;
485         case ALU_OP_ADD_WITH_CARRY :
486             op_adc(reg_in_a, reg_in_b, reg_out_accu, (flags==NULL)?dummyflags:fl
    ags);
487             break;
488         case ALU_OP_SUB :
489             op_sub(reg_in_a, reg_in_b, reg_out_accu, (flags==NULL)?dummyflags:fl
    ags);
490             break;
491         case ALU_OP_SUB_WITH_CARRY :
492             op_alu_sbc(reg_in_a, reg_in_b, reg_out_accu, (flags==NULL)?dummyflag
    s:flags);
493             break;
494         case ALU_OP_AND :
495             op_and(reg_in_a, reg_in_b, reg_out_accu, (flags==NULL)?dummyflags:fl
    ags);
496             break;
497         case ALU_OP_OR:
498             op_or(reg_in_a, reg_in_b, reg_out_accu, (flags==NULL)?dummyflags:fla
    gs);
499             break;
500         case ALU_OP_XOR :
501             op_xor(reg_in_a, reg_in_b, reg_out_accu, (flags==NULL)?dummyflags:fl
    ags);
502             break;
503         case ALU_OP_NEG_A :
504             op_neg_a(reg_in_a, reg_in_b, reg_out_accu, (flags==NULL)?dummyflags:
    flags);
505             break;
506         case ALU_OP_NEG_B :
507             op_neg_b(reg_in_a, reg_in_b, reg_out_accu, (flags==NULL)?dummyflags:
    flags);
508             break;
509         case ALU_OP_NOT_A :
510             op_not_a(reg_in_a, reg_in_b, reg_out_accu, (flags==NULL)?dummyflags:
    flags);
511             break;
```

```
512         case ALU_OP_NOT_B :
513             op_not_b(reg_in_a, reg_in_b, reg_out_accu, (flags==NULL)?dummyflags:
    flags);
514             break;
515         case ALU_OP_ASL :
516             op_alu_asl(reg_in_a, reg_in_b, reg_out_accu, (flags==NULL)?dummyflag
    s:flags);
517             break;
518         case ALU_OP_LSR :
519             op_alu_lsr(reg_in_a, reg_in_b, reg_out_accu, (flags==NULL)?dummyflag
    s:flags);
520             break;
521         case ALU_OP_ROL:
522             op_alu_rol(reg_in_a, reg_in_b, reg_out_accu, (flags==NULL)?dummyflag
    s:flags);
523             break;
524         case ALU_OP_ROR:
525             op_alu_ror(reg_in_a, reg_in_b, reg_out_accu, (flags==NULL)?dummyflag
    s:flags);
526             break;
527         case ALU_OP_RESET :
528             op_alu_reset(reg_in_a, reg_in_b, reg_out_accu, (flags==NULL)?dummyfl
    ags:flags);
529             break;
530         default:
531             printf("ALU(%i): Invalide operation %i selected", alu_opcode, alu_op
    code);
532     }
533 }
534
```