

Dec 04, 14 21:01

alu.c

Page 1/10

```

/*
  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

  -->> 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 - 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 */

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)

```

Dec 04, 14 21:01

alu.c

Page 2/10

```

{
    //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 = '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';
    }
}

```

Dec 04, 14 21:01

alu.c

Page 3/10

```

    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)
{
    half_adder(pbit, qbit);
    char carry1 = m[c];
    half_adder(m[s], cbit);
    if (carry1 == '1')
    {
        m[c] = '1';
    }
}

/*
Invertieren der Character Bits im Register reg
one_complement(char reg[]) --> NOT(reg)
*/
void one_complement(char reg[])
{
    int i = 0;
    for (i = 7; i >= 0; i--)
    {
        if (reg[i] == '1')
            reg[i] = '0';
        else
            reg[i] = '1';
    }
}

/*
Das zweier-Komplement des Registers reg wird in reg geschrieben
reg := K2(reg)
*/
void two_complement(char reg[])
{
    int i = 0;
    one_complement(reg);
    m[c] = '1';
    for (i = 7; i >= 0; i--)
    {
        if (reg[i] == '0')
        {
            reg[i] = '1';
            m[c] = '0';
            break;
        }
        else
        {
            reg[i] = '0';
        }
    }
}
}

/*

```

Dec 04, 14 21:01

alu.c

Page 4/10

```

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[])
{
    alu_reset();
    clearCarryflag(flags);
    clearOverflowflag(flags);
    int i = 0;

    for (i = 7; i >= 0; i--)
    {
        full_adder(rega[i], regb[i], m[c]);
        accumulator[i] = m[s];
    }

    if ((rega[0] == '1' ^ regb[0] == '1' ^ accumulator[0] == '0') || (rega[0] == '0'
^ regb[0] == '0' ^ accumulator[0] == '1'))
    {
        setOverflowflag(flags);
    }
    if (m[c] == '1')
        setCarryflag(flags);

    zsflagging(flags, accumulator);
}

/*
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[])
{
    char carry;
    carry = m[c];
    op_add(rega, regb, accumulator, flags);
    if (carry == '1')
    {
        char temp[8];
        char one[8] = {'0', '0', '0', '0', '0', '0', '0', '1'};
        strcpy(temp, accumulator);
        op_add(temp, one, 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 = rega + NOT(regb) + 1

```

Dec 04, 14 21:01

alu.c

Page 5/10

```

*/
void op_sub(char rega[], char regb[], char accumulator[], char flags[])
{
    char temp[8];
    int i = 0;
    for (i = 0; i < 8; i++)
    {
        temp[i] = regb[i];
    }
    two_complement(regb);
    char carry = m[c];

    op_add(rega, regb, accumulator, flags);

    for (i = 0; i < 8; i++)
    {
        regb[i] = temp[i];
    }
    // Overflow for subtraction : 0 && 1 && 1 or 1 && 0 && 0
    if ((rega[0] == '0' ^ regb[0] == '1' ^ accumulator[0] == '1') ||
        (rega[0] == '1' ^ regb[0] == '0' ^ accumulator[0] == '0'))
    {
        setOverflowflag(flags);
    }
    else
    {
        clearOverflowflag(flags);
    }

    if (carry == '1')
        setCarryflag(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[])
{
    char carry = m[c];
    op_sub(rega, regb, accumulator, flags);
    if (carry == '1')
    {
        char temp[8];
        char one[8] = {'0', '0', '0', '0', '0', '0', '0', '1'};
        strcpy(temp, accumulator);
        op_add(temp, one, accumulator, 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

```

Dec 04, 14 21:01

alu.c

Page 6/10

```

accumulator := rega AND regb
*/
void op_and(char rega[], char regb[], char accumulator[], char flags[])
{
    int i = 0;
    for (i = 0; i < 8; 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[])
{
    int i = 0;
    for (i = 0; i < 8; 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[])
{
    int i = 0;
    for (i = 0; i < 8; i++)
    {
        if ((rega[i] == '1' ^ regb[i] == '0') || (rega[i] == '0' ^ regb[i] == '1'))
            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[])
{

```

Dec 04, 14 21:01

alu.c

Page 7/10

```

    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
*/
void op_alu_asl(char regina[], char reginb[], char regouta[], char flags[])
{
    int i = 0;
    if (regina[0] == '1')
        setCarryflag(flags);
    else
        clearCarryflag(flags);

    for (i = 7; i >= 0; i--)
    {
        int dest = i-1;
        if (dest >= 0)
            regouta[dest] = regina[i];
    }
    regouta[7] = '0';
}

/*
    logical shift right
    lsr
*/
void op_alu_lsr(char regina[], char reginb[], char regouta[], char flags[])
{
    int i = 0;
    for (i = 0; i < 8; i++)
    {

```

Dec 04, 14 21:01

alu.c

Page 8/10

```

        int dest = i+1;
        if (dest < 8)
            regouta[dest] = regina[i];
    }

    if (getCarryflag(flags) == '1')
        regouta[0] = '1';
    else
        regouta[0] = '0';
}

/*
    rotate
    rotate left
*/
void op_alu_rol(char regina[], char reginb[], char regouta[], char flags[])
{
    char temp = getCarryflag(flags);
    op_alu_asl(regina, reginb, regouta, flags);
    regouta[7] = temp;
}

/*
    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[])
{
    char temp = regina[7];
    op_alu_asl(regina, reginb, regouta, flags);
    regouta[7] = temp;
    if (temp == '1')
        setCarryflag(flags);
    else
        clearCarryflag(flags);
}

/*
    Procedural approach to ALU with side-effect:
    Needed register are already allocated 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);

```

Dec 04, 14 21:01	alu.c	Page 9/10
gs);	<pre> 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 : op_alu_reset(reg_in_a, reg_in_b, reg_out_accu, (flags=NULL)?dummyflags:flags); break; default: printf("ALU(%i): Invalid operation %i selected", alu_opcode, alu_opcode); } } </pre>	

Dec 04, 14 21:01	alu.c	Page 10/10