University of Applied Sciences
GEORG-SIMON-OHM FACHHOCHSCHULE NÜRNBERG
FACHBEREICH elektrotechnik feinwerktechnik informationstechnik

Praktikum Mikroelektronik

im WS 2006/07

Projekt: Entwicklung eines Calculators

Gruppenmitglieder:

Elbadi, Abdelilah	758972
Held, Matthias	728036
Uhl, Michael	712560

Betreuer:

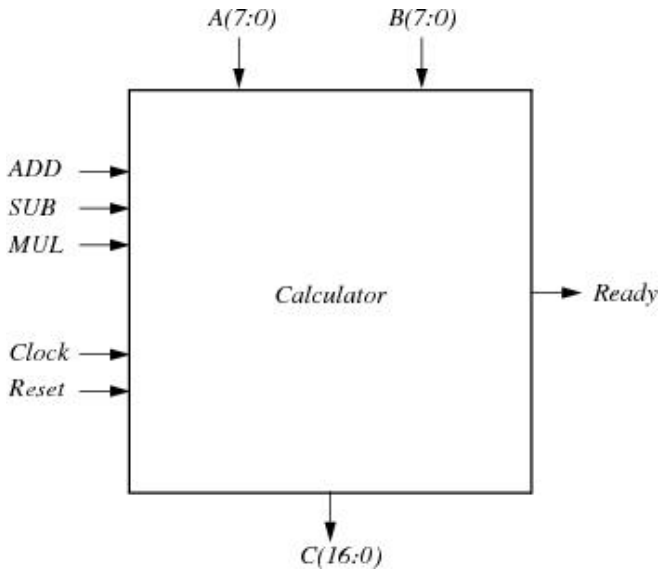
Prof. Dr. Siegl

Inhaltsverzeichnis

1 Aufgabenstellung

Mit Hilfe moderner Designwerkzeuge soll ein ASIC-Design (Applied Specific Integrated Circuit) am Beispiel eines Calculator-Bausteins entworfen und verifiziert werden. Zieltechnologie ist zum einen ein Standardzell-Design und zum anderen ein FPGA-Design (Field Programmable Gate Array) bzw. ein CPLD-Design (Complex Programmable Logic Device). An einem leicht verständlichen praktischen Beispiel gilt es, in den Designablauf und in die dafür erforderlichen Designwerkzeuge einzuführen.

Mit dem Calculator-Baustein sollen zwei "unsigned" Dualzahlen *A* und *B* mit jeweils 12-Bit Länge addiert, subtrahiert bzw. multipliziert werden. Mit den Steuersignalen *ADD, SUB, Mul* wird die auszuführende Operation ausgewählt. Ist eines dieser Steuersignale aktiv, so erfolgt die Übernahme der Operanden in den Calculator-Baustein, die ausgewählte Funktion wird dann ausgeführt und das Ergebnis als "signed" Dualzahl *C* der Länge 24 Bit ausgegeben. Die asynchronen Steuersignale ADD, SUB, MUL sind länger aktiv, als die auszuführende Operation.



- Abbildung 1: Calculator-Baustein mit den Schnittstellensignalen -

1.1. Addition zweier Dualzahlen

Abbildung 2 zeigt beispielhaft die Addition zweier unsigned Dualzahlen. Das Ergebnis kann einen Übertrag (Carry-Bit) enthalten, ist aber immer positiv. Zur Darstellung des Ergebnisses werden bei 12-Bit-Operanden 13-Bit benötigt.

Carry	Dualzahl (11:0)												Dezimal
2^{12}	2^{11}	2^{10}	2^{9}	2^{8}	2^{7}	2^{6}	2^{5}	2^{4}	2^{3}	2^{2}	2^{1}	2^{0}	
	0	0	0	0	1	1	0	0	0	0	1	1	195
	0	0	0	0	1	0	0	1	0	0	0	1	145
0	0	0	0	1	0	1	0	1	0	1	0	0	340

Abbildung 2: Addition zweier Dualzahlen -

1.2. Subtraktion zweier Dualzahlen

Die Subtraktion zweier Dualzahlen wird zweckmäßig ausgeführt durch Addition des Komplements des Subtrahenden. Bei einer 12-stelligen Dualzahl erfolgt die Komplementbildung des Subtrahenden und dessen Addition zum Minuenden gemäß Abbildung 3:

Carry	Dualzahl (11:0)												Dezimal
2^{12}	2^{11}	2^{10}	2^{9}	2^{8}	2^{7}	2^{6}	2^{5}	2^{4}	2^{3}	2^{2}	2^{1}	2^{0}	
	0	0	0	0	1	1	0	0	0	0	1	1	195
	0	0	0	0	1	0	0	1	0	0	0	1	145
1	0	0	0	0	0	1	1	0	1	1	1	1	855
0	0	0	0	0	0	0	1	1	0	0	1	0	50

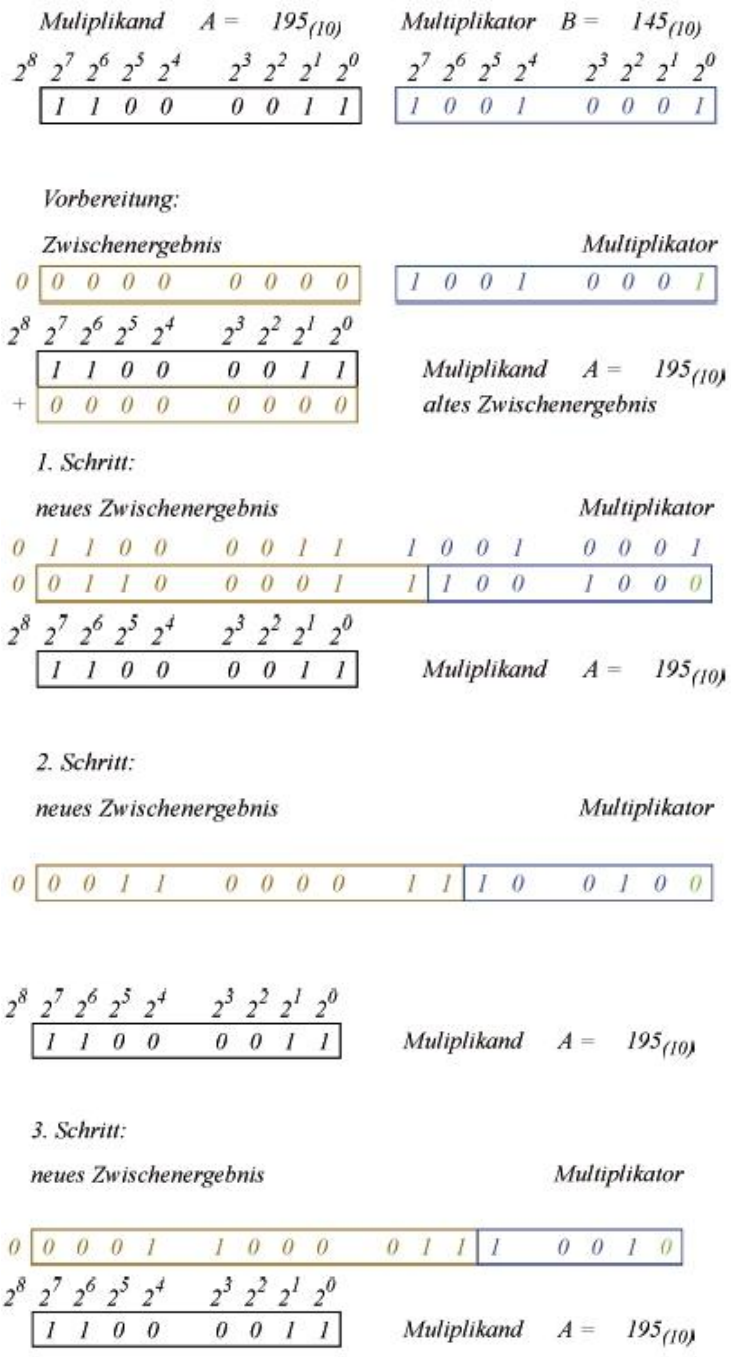
- Abbildung 3: Subraktion zweier Dualzahlen -

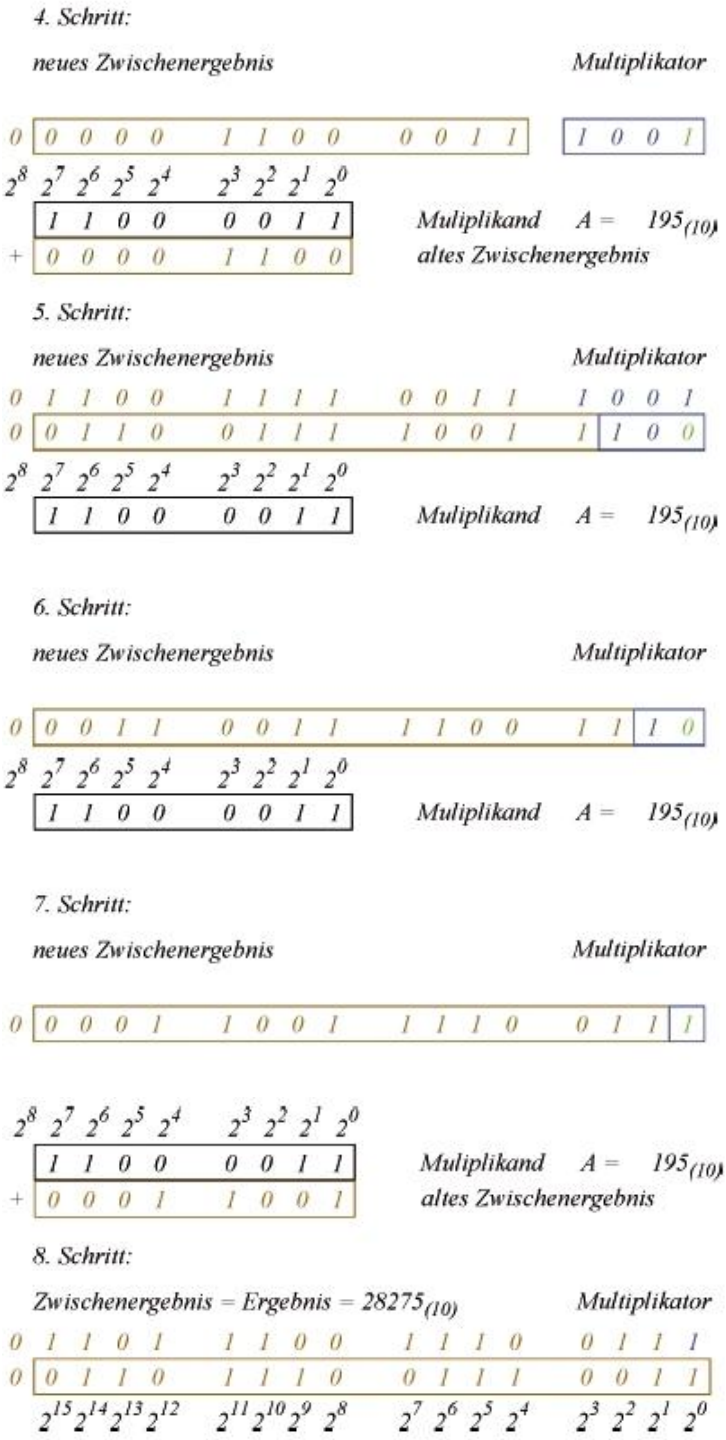
In der gestellten Aufgabe wird von unsigned Dualzahlen für A und B ausgegangen. Dann ist bei A>B das Ergebnis positiv. Der dabei auftretende Übertrag muss ausgeblendet werden. Bei A<B ist das Ergenis negativ. Das Carry-Bit muss gesetzt werden. Zur korrekten Darstellung ist für das Ergebnis eine signed" Dualzahl vorzusehen.

1.3. Multiplikation zweier Dualzahlen

Die Multiplikation zweier Dualzahlen kann auf verschiedene Weise durchgeführt werden. Sie lässt sich auch auf die mehrfache Addition zweier Dualzahlen zurückführen. Abbildung 4 veranschaulicht die Vorgehensweise anhand eines konkreten Beispiels.
Der Mutliplikand (im Beispiel 1100 0011) bleibt unverändert. Der Multiplikator (im Beispiel 1001 0001) wird rechts vom Zwischenergebnis angeordnet. Nach Durchführung des Vorbereitungsschrittes ist das Zwischenergebnis zunächst 0000 0000. Ist das LSB-Bit (niedrigstwertige Bit) des Multiplikators "1", so müssen der Multiplikand und das Zwischenergebnis addiert werden. Das neue Zwischenergebnis ist anschließend um eine Stelle nach rechts zu verschieben. Dabei erfolgt auch eine Verschiebung des Multiplikators um 1 Stelle nach rechts. Ist das LSB-Bit "0", so entfällt die Addition, es wird das Zwischenergebnis um 1 Stelle nach rechts verschoben. Insgesamt sind bei einer n-stelligen Dualzahl nach diesem Verfahren n Teilschritte zur Ausführung der Multiplikation erforderlich.

Aufgrund der Vorgabe der Aufgabenstellung ist für die Operanden A und B je eine unsigned Dualzahl gegeben. Das Vorzeichen spielt also keine Rolle. Das Ergebnis ist immer positiv. Die Länge des Ergebnisses kann bei 12-Bit-Operanden 24-Bit umfassen.

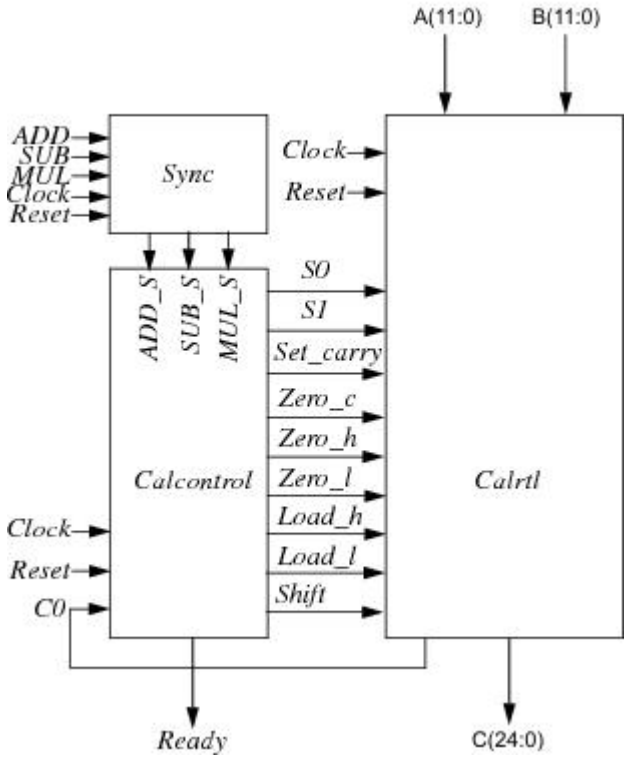




- Abbildung 4: Multiplikation zweier Dualzahlen -

1.4. Erforderliche Architektur

Die zur Lösung der Aufgabe erforderliche Architektur zeigt Abbildung 2. Im Allgemeinen besteht ein Logiksystem aus einem Datenpfad und einem Kontrollpfad. Der Datenpfad ist im Funktionsblock Calrtl realisiert, der Kontrollpfad in Calcontrol. Zur Synchronisation der asynchronen Steuersignale ADD, SUB und MUL wird der Synchronblock Sync benötigt. Dabei sollen die Ausgangssignale ADD_S, SUB_S und MUL_S dem Block Calcontrol unabhängig von der tatsächlichen Länge der Eingangssignale ADD, SUB und MUL drei Taktlängen zur Verfügung stehen. Der Datenpfad Calrtl übernimmt die Operanden und führt die Operationen gemäß den aktiven Steuersignalen aus. Der Funktionsblock Calcontrol erzeugt die für die Durchführung der Operationen erforderlichen Steuersignale. Mit Ready wird signalisiert, dass die Operation abgeschlossen ist.



- Abbildung 5 :Architektur des Calculator-Bausteins -

Die einzelnen Signale haben dabei folgende Aufgabe:

Signal	Bedeutung
ADD	Eingangssignal Addition (asynchron)
SUB	Eingangssignal Subtraktion (asynchron)
MUL	Eingangssignal Multiplikation (asynchron)
A(11:0)	Eingangssignal Operant A (Dualzahl, 12 Bit, unsigned)
B(11:0)	Eingangssignal Operant B (Dualzahl, 12 Bit, unsigned)
Clock	Takt

Reset	Reset
C (24:0)	Ausgangssignal Ergebnis (Dualzahl, 25 Bit, signed)
ADD_S	Steuersignal Addition (synchron)
SUB_S	Steuersignal Subtraktion (synchron)
MUL_S	Steuersignal Multiplikation (synchron)
S0	Steuersignal für Mux
S1	Steuersignal für Mux
Set_carry	Carry setzen, C(24) setzen
Zero_c	Rücksetzen von C(24)
Zero_h	Rücksetzen von C(23:12)
Zero_l	Rücksetzen von C(11:0)
Load_h	Laden von C(23:12)
Load_l	Laden von C(11:0)
Shift	Rechts-Shift von C(24:0)
C0	Rückführung des Carry-Bits
Ready	Ausgangssignal Operation abgeschlossen

- Tabelle 1: Signale des Calculator-Bausteins -

Zur Lösung der Aufgabe steht das Simulationssystem ModelSim von Mentor Graphics zur Verfügung.

2 Realisierung des Claculators

2.1. Sync-Block

Die zentrale Aufgabe des Sync-Blocks ist es die externen asynchronen Steuersignale ADD für die Addition, SUB für die Subtraktion und MUL für die Multiplikation in synchrone Steuersignale für den nachfolgenden Block Calcontrol zu wandeln. Die synchronen Steuersignale ADD_S, SUB_S und MUL_S sollen unabhängig von der tatsächlichen Länge der externen Signale 3 Taktlängen betragen. Zur Lösung dieser Problematik wurden Gruppenintern 2 verschiedene Ansätze verfolgt.

Zum einen stand eine Variante mit 6 JKK-Flip-Flops (diese bilden einen 3 zu 1 Frequenzteiler) und 3 D-Flip-Flops (zur eigentlichen Erfüllung der Aufgabe).

```
library IEEE;
use IEEE.std_logic_1164.all;
use work.all;

entity sync is
    port(
        CLK   :   in std_logic;
        Reset :   in std_logic;
        ADD   :   in std_logic;
        SUB   :   in std_logic;
        MUL   :   in std_logic;
        ADD_S :   out std_logic;
        SUB_S :   out std_logic;
        MUL_S :   out std_logic);
end sync;

architecture sync_arch of sync is

    component JKFF
        port(
            CLK, Reset, J, K   :   in std_logic;
            Q, NQ              :   out std_logic);
        end component;

    component DFF
        port(
            CLK, Reset, D    :   in std_logic;
            T                :   out std_logic);
        end component;

    signal s1, s2, s3, s4, s5, s6, s7, s8, s9 : std_logic;

    begin
       D_ADD1: JKFF port map (CLK => CLK, Reset => Reset, K => '1', J => s2, NQ => s1);
       D_ADD2: JKFF port map (CLK => CLK, Reset => Reset, K => '1', J => s1, Q => s2, NQ =>
s3);
       D_ADD:  DFF  port map (CLK => s3, Reset => Reset, D => ADD, T => ADD_S);

       D_SUB1: JKFF port map (CLK => CLK, Reset => Reset, K => '1', J => s5, NQ => s4);
       D_SUB2: JKFF port map (CLK => CLK, Reset => Reset, K => '1', J => s4, Q => s5, NQ =>
s6);
       D_SUB:  DFF  port map (CLK => s6, Reset => Reset, D => SUB, T => SUB_S);

       D_MUL1: JKFF port map (CLK => CLK, Reset => Reset, K => '1', J => s8, NQ => s7);
       D_MUL2: JKFF port map (CLK => CLK, Reset => Reset, K => '1', J => s7, Q => s8, NQ =>
s9);
```

```
      D_MUL:  DFF  port map (CLK => s9, Reset => Reset, D => MUL, T => MUL_S);

end sync_arch;
```

- Code 1: Sync-Block Variante 1 -

Aufgrund der Komplexität der Lösung wurde die Variante jedoch für das weitere Projekt verworfen und nachfolgende Lösung verwendet:

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE work.ALL;

ENTITY sync IS
    PORT(  clk        : IN  std_logic;
           reset      : IN  std_logic;
           add        : IN  std_logic := '0';
           sub        : IN  std_logic := '0';
           mul        : IN  std_logic := '0';
           add_s      : OUT std_logic;
           sub_s      : OUT std_logic;
           mul_s      : OUT std_logic);
END sync;

ARCHITECTURE behave OF sync IS
signal int_add1, int_sub1, int_mul1 : std_logic := '0';        --interne Signale 1
signal int_add2, int_sub2, int_mul2 : std_logic := '0';        --interne Signale 2
signal add_v, sub_v, mul_v : std_logic := '0';                 --Signale werden nur 1-mal
durchgelassen
signal timer_1, timer_2, timer_3 : integer range 0 to 3 := 0;  --Zähler für 3 Taktperioden
BEGIN

dff: PROCESS(clk)     --drei Zusätzliche D-FF vor den Eingängen
    BEGIN
    IF (clk = '1' AND clk'event) THEN
        int_add1 <= add;
        int_sub1 <= sub;
        int_mul1 <= mul;
    END IF;
END PROCESS dff;

sync: PROCESS(clk)
    BEGIN
    IF (clk = '1' AND clk'event) THEN


        IF (int_add1 = '1' AND int_sub2 = '0' AND int_mul2 = '0' AND add_v = '0') THEN
            int_add2 <= '1';
            add_v <= '1';
        END IF;
        IF (int_add1 = '0') THEN
            add_v <= '0';
        END IF;
        IF (int_add2 = '1') THEN
            timer_1 <= timer_1 + 1;
            IF (timer_1 >= 2) THEN
                timer_1 <= 0;
                int_add2 <= '0';
            END IF;
        END IF;


        IF(int_sub1 = '1' AND int_add2 = '0' AND int_mul2 = '0' AND sub_v = '0') THEN
            int_sub2 <= '1';
            sub_v <= '1';
        END IF;
        IF (int_sub1 = '0') THEN
```

```
            sub_v <= '0';
        END IF;
        IF (int_sub2 = '1') THEN
            timer_2 <= timer_2 + 1;
            IF (timer_2 >= 2) THEN
                timer_2 <= 0;
                int_sub2 <= '0';
            END IF;
        END IF;


        IF(int_mul1 = '1' AND int_add2 = '0' AND int_sub2 = '0' AND mul_v = '0') THEN
            int_mul2 <= '1';
            mul_v <= '1';
        END IF;
        IF (int_mul1 = '0') THEN
            mul_v <= '0';
        END IF;
        IF (int_mul2 = '1') THEN
            timer_3 <= timer_3 + 1;
            IF (timer_3 >= 2) THEN
                timer_3 <= 0;
                int_mul2 <= '0';
            END IF;
        END IF;


    END IF;
END PROCESS sync;

res: PROCESS(clk)
    BEGIN
    IF (clk = '1' AND clk'event AND reset = '1') THEN
        add_s <= '0';
        sub_s <= '0';
        mul_s <= '0';
    ELSIF (clk = '1' AND clk'event AND reset = '0') THEN
        add_s <= int_add2;
        sub_s <= int_sub2;
        mul_s <= int_mul2;
    END IF;
END PROCESS res;

END behave;
```

- Code 2: Sync-Block Variante 2 -

Vor der eigentlichen Synchronisationsschaltung wurden drei zusätzliche D-Flip-Flops geschaltet um hiermit Setup- und Holdzeit der nachfolgenden Schaltung zu garantieren. Die Verzögerungszeit beträgt hierbei 1 Taktperiode des Systemtaktes.

Neben der bereits erwähnten Guard-Schaltung (3 DFF) besteht die Architecture des Sync-Blocks aus zwei Prozessen. Der Prozess „res" hat die Aufgabe das Reset zu realisieren. Er besteht daher aus zwei if-Anweisungen. Die eigentliche Synchronisationsschaltung ist im Prozess „sync" beschrieben. Dieser ist als Zähler realisiert und verwendet zahlreiche interne Signale.

Beispielhaft an den Signalen für die Addition soll die Funktionsweise erläutertet werden:

```
IF (clk = '1' AND clk'event) THEN
        IF (int_add1 = '1' AND int_sub2 = '0' AND int_mul2 = '0' AND add_v = '0') THEN
            int_add2 <= '1';
            add_v <= '1';
        END IF;
```

```
IF (int_add1 = '0') THEN
    add_v <= '0';
END IF;
IF (int_add2 = '1') THEN
    timer_1 <= timer_1 + 1;
    IF (timer_1 >= 2) THEN
        timer_1 <= 0;
        int_add2 <= '0';
    END IF;
END IF;
```

Das interne Signal int_add1 ist das Signal am Q-Ausgang des D-FF nach der Guard-Schaltung. Falls dieses Signal 1 ist, die internen Signale int_sub2, int_mul2 und add_v 0 sind, so werden die Signale int_add2 und add_v auf 1 gesetzt. add_v hat dabei eine Überwachungsfunktion, dass die Schleife nur einmal ausgeführt wird. Die nächste if-Anweisung im Code bewirkt, dass das Hilfssignal add_v wieder auf 0 gesetzt wird, falls int_add1 0 ist. Der nächste if-Block realisiert den Zähler. Dabei bleibt int_add2 solange 1, bis das Signal timer_1 den Wert 2 erreicht hat (dies entspricht den geforderten 3 Taktperioden). Anschließend wird timer_1 und int_add2 wieder auf 0 gesetzt.

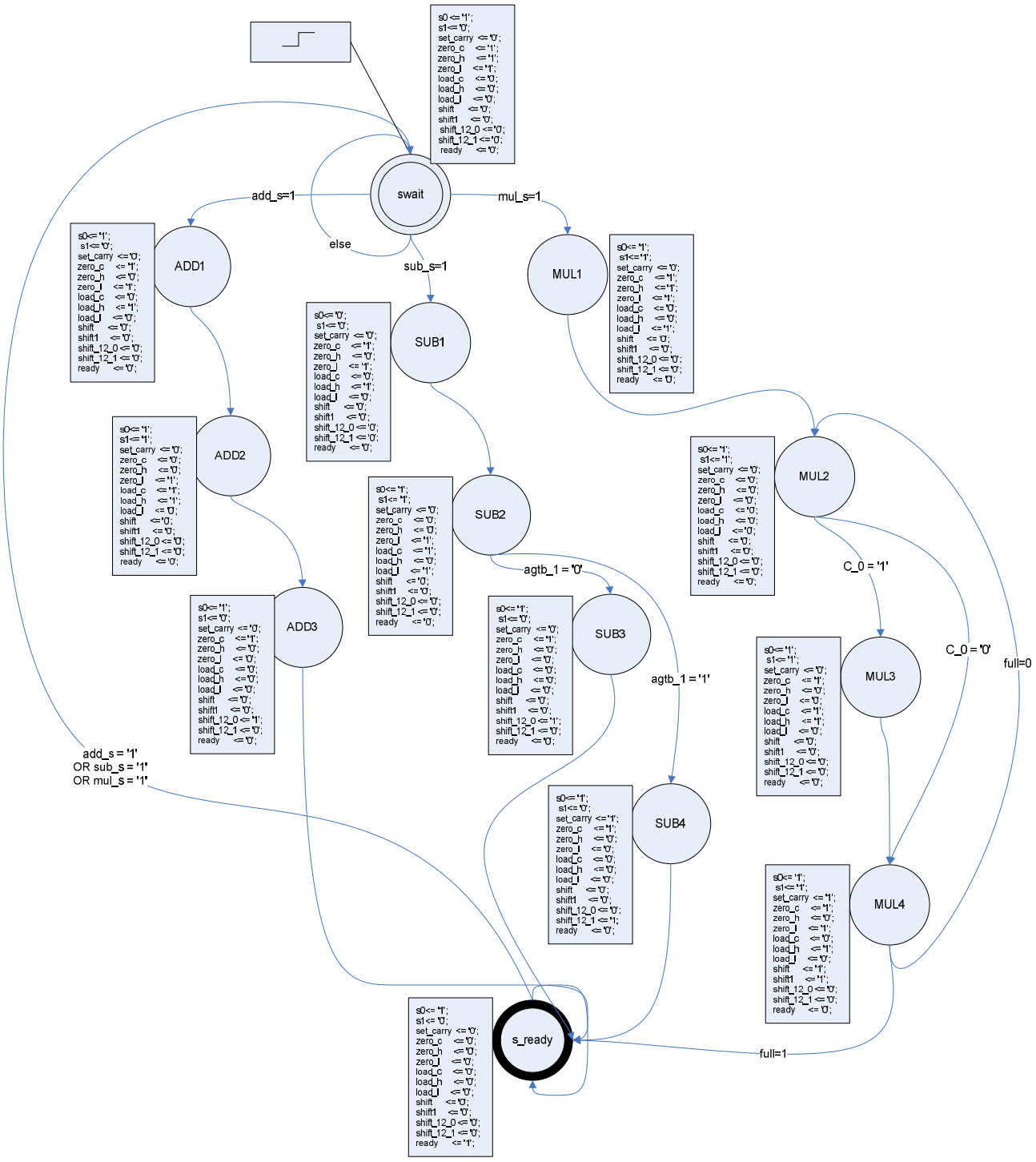
Für die Subtraktion und Multiplikation geschieht dies in entsprechender Weise.

2.2. Calcontrol

Der Calcontrol-Block nimmt den Kontrollpfad im Calculator-Baustein ein. Das heißt, er erzeugt die für die Durchführung der Rechenoperationen die erforderlichen Steuersignale. Die externen Signale werden in der Entity beschrieben durch:

Signal	Bedeutung
clk	Systemtakt (IN std_logic)
reset	Reset (IN std_logic)
c_0	Steuersignal vom Akkumulator (IN std_logic)
add_s	Synchrones ADD (IN std_logic)
sub_s	Synchrones SUB (IN std_logic)
mul_s	Synchrones MUL (IN std_logic)
agtb_1	Steuersignal für Subtraktion (IN std_logic)
s0	Steuersignal für MUX (OUT std_logic)
s1	Steuersignal für MUX (OUT std_logic)
set_carry	Steuersignal zum Carrybit setzten (OUT std_logic)
zero_c	C-Register auf 0 setzten (OUT std_logic)
zero_h	High Teil des Akkumulatorregisters auf 0 setzten (OUT std_logic)
zero_l	Low Teil des Akkumulatorregisters auf 0 setzten (OUT std_logic)
load_c	Daten von ALU ins C-Register laden (OUT std_logic)
load_h	Daten in den High Teil des Akkumulatorregisters laden (OUT std_logic)
load_l	Daten in den Low Teil des Akkumulatorregisters laden (OUT std_logic)
shift_12_0	Ergebnis um 12 nach rechts schieben und mit 0 auffüllen(OUT std_logic)
shift_12_1	Ergebnis um 12 nach rechts schieben und mit 1 auffüllen(OUT std_logic)

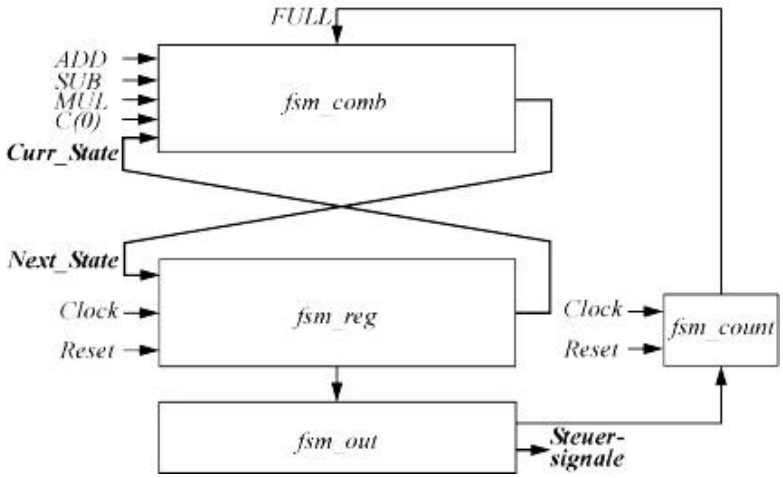
- Tabelle 2: Externe Signale des Calcontrol-Blocks -



- Abbildung 6 : Zustandsübergangsdiagram Calcontrol -

In Abbildung 6 ist das Zustandsübergangsdiagram Calcontrol-Blockes zu sehen. Sämtliche in Tabelle 2 beschriebenen Signale sind den Zuständen zugeordnet.

Da die Funktionalität des Calcontrol-Blockes als Moore-Automat ausgeführt wurde, wird er in der Architecture folglich durch 3 Prozesse beschrieben: `fsm_comb: PROCESS (clk)` (der Kombinatorikteil, Beschreibung der Zustandsübergänge), `fsm_out: PROCESS (curr_state)` (die eigentliche Beschreibung der Zustände und der Ausgang der Steuersignale) und `fsm_reg: PROCESS(clk)` (die Realisierung des Resetsignals). Im vorliegenden Fall wird zusätzlich noch der Prozess `fsm_count: PROCESS(clk)` benötigt. Dieser ist als Zähler realisiert und wird bei der Multiplikation benötigt, da wie oben beschrieben die Multiplikation aus n-1 Schritten (hier 11) besteht.



- Abbildung 7 : Blockschaltbild Calcontrol -

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE work.ALL;

ENTITY calcontrol IS
    PORT(  clk        : IN  std_logic;
           reset      : IN  std_logic;
           c_0        : IN  std_logic;
           add_s      : IN  std_logic;
           sub_s      : IN  std_logic;
           mul_s      : IN  std_logic;
           agtb_1     : IN  STD_LOGIC;
           s0         : OUT std_logic;
           s1         : OUT std_logic;
           set_carry  : OUT std_logic;
           zero_c     : OUT std_logic;
           zero_h     : OUT std_logic;
           zero_l     : OUT std_logic;
           load_c     : OUT std_logic;
           load_h     : OUT std_logic;
           load_l     : OUT std_logic;
           shift      : OUT std_logic;
           shift_12_0 : OUT std_logic;
           shift_12_1 : OUT std_logic;
           ready      : OUT std_logic);
END calcontrol;

ARCHITECTURE behave OF calcontrol IS

SIGNAL counter : integer range 0 to 11 := 0;
SIGNAL full    : STD_LOGIC;
SIGNAL shift1  : STD_LOGIC;
```

```
SIGNAL m       : STD_LOGIC := '0';

TYPE state IS (swait, s_ready,
               add1, add2, add3,
               sub1, sub2, sub3, sub4,
               mul1, mul2, mul3, mul4 );
SIGNAL curr_state: state;
SIGNAL next_state: state;


BEGIN

fsm_comb: PROCESS (clk)
BEGIN
    CASE curr_state IS
        WHEN swait =>
            IF add_s = '1' THEN
                next_state <= add1;
            ELSIF sub_s = '1' THEN
                next_state <= sub1;
            ELSIF mul_s = '1' THEN
                next_state <= mul1;
            ELSE
                next_state <= swait;
            END IF;

        WHEN add1 =>
            next_state <= add2;
        WHEN add2 =>
            next_state <= add3;
        WHEN add3 =>
            next_state <= s_ready;

        WHEN sub1 =>
            next_state <= sub2;
        WHEN sub2 =>
            IF (agtb_1 = '0') THEN
                next_state <= sub3;
            END IF;
            IF (agtb_1 = '1') THEN
                next_state <= sub4;
            END IF;
        WHEN sub3 =>
            next_state <= s_ready;
        WHEN sub4 =>
            next_state <= s_ready;

        WHEN mul1 =>
            next_state <= mul2;
        WHEN mul2 =>
            IF (C_0 = '1') THEN
                next_state <= mul3;
            ELSE
                next_state <= mul4;
            END IF;
        WHEN mul3 =>
            next_state <= mul4;
        WHEN mul4 =>
            IF (full = '1') THEN
                next_state <= s_ready;
            ELSE
                next_state <= mul2;
            END IF;


        WHEN s_ready =>
            IF (add_s = '1' OR sub_s = '1' OR mul_s = '1') THEN
                next_state <= swait;
            ELSE
                next_state <= s_ready;
            END IF;
```

```
        WHEN OTHERS =>
            next_state <= swait;
    END CASE;
END PROCESS fsm_comb;


fsm_reg: PROCESS(clk)
BEGIN
    IF (clk'event AND clk = '1') THEN
        IF (reset = '1') THEN
            curr_state <= swait;
        ELSE
            curr_state <= next_state;
        END IF;
    END IF;
END PROCESS fsm_reg;


fsm_count: PROCESS(clk)
BEGIN
    IF (mul_s = '1') THEN
        m <= '1';
    END IF;
    IF (clk'event AND clk = '1') THEN
        IF (m = '1' AND counter < 11) THEN
            IF (shift1 = '1') THEN
                counter <= counter + 1;
            END IF;
            full <= '0';
        ELSE
            full <= '1';
            m <= '0';
            counter <= 0;
        END IF;
    END IF;
END PROCESS fsm_count;


fsm_out: PROCESS (curr_state)
BEGIN
    CASE curr_state IS
        WHEN swait =>
            s0         <= '1'; --b durchschalten
            s1         <= '0';
            set_carry  <= '0';
            zero_c     <= '1';
            zero_h     <= '1';
            zero_l     <= '1';
            load_c     <= '0';
            load_h     <= '0';
            load_l     <= '0';
            shift      <= '0';
            shift1     <= '0';
            shift_12_0 <= '0';
            shift_12_1 <= '0';
            ready      <= '0';
-------------------------------------------------------------------------------
        WHEN add1 =>
            s0         <= '1'; --b durchschalten
            s1         <= '0';
            set_carry  <= '0';
            zero_c     <= '1';
            zero_h     <= '0';
            zero_l     <= '1';
            load_c     <= '0';
            load_h     <= '1'; --h laden
            load_l     <= '0';
            shift      <= '0';
            shift1     <= '0';
            shift_12_0 <= '0';
```

```
            shift_12_1 <= '0';
            ready      <= '0';

        WHEN add2 =>
            s0         <= '1'; --ALU durchschalten
            s1         <= '1';
            set_carry  <= '0';
            zero_c     <= '0';
            zero_h     <= '0';
            zero_l     <= '1';
            load_c     <= '1';
            load_h     <= '1'; --h laden
            load_l     <= '0';
            shift      <= '0';
            shift1     <= '0';
            shift_12_0 <= '0';
            shift_12_1 <= '0';
            ready      <= '0';

        WHEN add3 =>
            s0         <= '1';
            s1         <= '0';
            set_carry  <= '0';
            zero_c     <= '1';
            zero_h     <= '0';
            zero_l     <= '0';
            load_c     <= '0';
            load_h     <= '0';
            load_l     <= '0';
            shift      <= '0';
            shift1     <= '0';
            shift_12_0 <= '1'; --shiften um 12
            shift_12_1 <= '0';
            ready      <= '0';
-------------------------------------------------------------------------------------
        WHEN sub1 =>
            s0         <= '0'; --ZK durchschalten
            s1         <= '0';
            set_carry  <= '0';
            zero_c     <= '1';
            zero_h     <= '0';
            zero_l     <= '1';
            load_c     <= '0';
            load_h     <= '1'; --h laden
            load_l     <= '0';
            shift      <= '0';
            shift1     <= '0';
            shift_12_0 <= '0';
            shift_12_1 <= '0';
            ready      <= '0';

        WHEN sub2 =>
            s0         <= '1'; --ALU durchschalten
            s1         <= '1';
            set_carry  <= '0';
            zero_c     <= '0';
            zero_h     <= '0';
            zero_l     <= '1';
            load_c     <= '0';
            load_h     <= '1'; --h laden
            load_l     <= '0';
            shift      <= '0';
            shift1     <= '0';
            shift_12_0 <= '0';
            shift_12_1 <= '0';
            ready      <= '0';

        WHEN sub3 =>
            s0         <= '1';
            s1         <= '0';
            set_carry  <= '0';
```

```
            zero_c     <= '1';
            zero_h     <= '0';
            zero_l     <= '0';
            load_c     <= '0';
            load_h     <= '0';
            load_l     <= '0';
            shift      <= '0';
            shift1     <= '0';
            shift_12_0 <= '1'; --shiften um 12 und 0 setzen
            shift_12_1 <= '0';
            ready      <= '0';

        WHEN sub4 =>
            s0         <= '1';
            s1         <= '0';
            set_carry  <= '1';
            zero_c     <= '0';
            zero_h     <= '0';
            zero_l     <= '0';
            load_c     <= '0';
            load_h     <= '0';
            load_l     <= '0';
            shift      <= '0';
            shift1     <= '0';
            shift_12_0 <= '0';
            shift_12_1 <= '1'; --shiften um 12 und 1 setzen
            ready      <= '0';
-------------------------------------------------------------------------------------
        WHEN mul1 =>
            s0         <= '1'; --ALU durchschalten
            s1         <= '1';
            set_carry  <= '0';
            zero_c     <= '1';
            zero_h     <= '1';
            zero_l     <= '0';
            load_c     <= '0';
            load_h     <= '0';
            load_l     <= '1'; --l laden
            shift      <= '0';
            shift1     <= '0';
            shift_12_0 <= '0';
            shift_12_1 <= '0';
            ready      <= '0';

        WHEN mul2 =>
            s0         <= '1'; --ALU durchschalten
            s1         <= '1';
            set_carry  <= '0';
            zero_c     <= '0';
            zero_h     <= '0';
            zero_l     <= '0';
            load_c     <= '0';
            load_h     <= '0';
            load_l     <= '0';
            shift      <= '0';
            shift1     <= '0';
            shift_12_0 <= '0';
            shift_12_1 <= '0';
            ready      <= '0';

        WHEN mul3 =>
            s0         <= '1'; --ALU durchschalten
            s1         <= '1';
            set_carry  <= '0';
            zero_c     <= '0';
            zero_h     <= '0';
            zero_l     <= '0';
            load_c     <= '1';
            load_h     <= '1'; --h laden
            load_l     <= '0';
            shift      <= '0';
```

```
            shift1     <= '0';
            shift_12_0 <= '0';
            shift_12_1 <= '0';
            ready      <= '0';

        WHEN mul4 =>
            s0         <= '1'; --ALU durchschalten
            s1         <= '1';
            set_carry  <= '1';
            zero_c     <= '0';
            zero_h     <= '0';
            zero_l     <= '0';
            load_c     <= '0';
            load_h     <= '0';
            load_l     <= '0';
            shift      <= '1';
            shift1     <= '1'; --shift
            shift_12_0 <= '0';
            shift_12_1 <= '0';
            ready      <= '0';
-------------------------------------------------------------------------------
        WHEN s_ready =>
            s0         <= '1';
            s1         <= '0';
            set_carry  <= '0';
            zero_c     <= '0';
            zero_h     <= '0';
            zero_l     <= '0';
            load_c     <= '0';
            load_h     <= '0';
            load_l     <= '0';
            shift      <= '0';
            shift1     <= '0';
            shift_12_0 <= '0';
            shift_12_1 <= '0';
            ready      <= '1';

        WHEN OTHERS =>
            NULL;
    END CASE;
END PROCESS fsm_out;

END behave;
```

- Code 3: Calcontrol-Block -

In der VHDL-Beschreibung des Calcontrol-Blockes (Code 3) lassen sich gut die einzelnen Zustände erkennen:

2.2.1. Zustand swait

Der Calculator befindet sich solange im Zustand swait, bis nicht eines der Signale ADD_S, SUB_S oder MUL_S 1 sind. In diesem Zustand schaltet der Multiplexer bereits das Signal B durch (s0=1, s1=0) und setzt das C- und das Akkumulatorregister (High- und Low-Teil) auf 0. Damit lässt sich in den nachfolgenden Zuständen der Rechenoperationen jeweils ein Zustand sparen.

2.2.2. Beschreibung der Addition

Für die Addition werden 3 Zustände benötigt (ADD1, ADD2 und ADD3). Im Zustand ADD1 wird dabei mittels des Multiplexers B durchgeschaltet (s0=1, s1=0), in den High-Teil des Akkumulatorregister geladen und das C- und der Low-Teil des Akkumulatorregisters auf 0 gesetzt. Im Zustand ADD2 wird das Ausgangssignal der ALU mit dem Multiplexer durchgeschaltet und in den High-Teil des Akkumulatorregisters geladen (load_h) und das C- bzw. der Low-Teil des Akkumulatorregisters auf 0 gesetzt (load_c, load_h). Im letzten Zustand der Addition (ADD3) wird wiederum B durchgeschaltet und das C-Register auf 0 gesetzt und gleichzeitig das Ergebnis der Addition um 12 Bit nach rechts geschiftet (in den Low-Teil des Akkumulatorregisters).
Die Eigentliche Addition wird demnach im Zustand ADD1 beschrieben, anschießend wird das Ergebnis in den High-Teil des Akkumulatorregisters (ADD2) geladen und im Zustand ADD3 in den Low-Teil des Akkumulatorregisters geschoben (schieben um 12 Bit).

2.2.3. Beschreibund der Subtraktion

Für die Subtraktion sind 4 Zustände vorgesehen, da unterschieden werden muss, ob das Ergebnis positiv oder negativ ist. Allerdings werden davon nur jeweils 3 benötigt. Je nachdem wird Zustand 3 oder Zustand 4 ausgeführt.
Im Zustand SUB1 werden das C-Register und der Low-Teil des Akkumulatorregisters auf 0 gesetzt und das Ausgangssignal des Multiplexers (s0=0, s1=0 - entspricht dem Zweierkomplement von B) in den High-Teil des Akkumulatorregisters geladen. Anschließend wird das Ergebnis der ALU vom Multiplexer durchgeschaltet und in den High-Teil des Akkumulatorregisters geladen und der Low-Teil des Akkumulatorregisters auf 0 gesetzt (SUB2). Nun findet innerhalb der Subtraktion eine Fallunterscheidung statt. Ist das Steuersignal agtb_1 gleich 1 (aus dem a>b-Vergleich, dass Ergebnis wird also negativ), so wird in den Zustand SUB4 gesprungen. Dort wird B mittels des Multiplexers durchgeschaltet, das Carrybit wird gesetzt, und das zuvor im Zustand 2 in den High-Teil des Akkumulatorregisters geladene Ergebnis um 12 Bit nach rechts in den Low-Teil des Akkumulatorregisters geladen und die restlichen Stellen mit 1 aufgefüllt. Ist das Steuersignal agtb_1 gleich 0, dass Ergebnis wird also positiv, so wird vom Zustand SUB2 in den Zustand SUB3 gesprungen. Dort wird wie in SUB4 B mittels des Multiplexers durchgeschaltet, das Carrybit gesetzt und das Ergebnis um 12 Bit nach rechts in den Low-Teil des Akkumulatorregisters geschoben – nun werden jedoch die Stellen links davon mit 0 aufgefüllt.

2.2.4. Beschreibung der Multiplikation

Für die Multiplikation sind ebenfalls 4 Zustände vorgesehen, da die Multiplikation aus einer Kombination von Additionen (der Zwischenergebnisse) und Schieben um 1 Bit nach rechts (der Zwischenergebnisse) besteht. Als Kriterium wird hierbei das Signal C_0 verwendet (das LSB des Akkumulatorregisters). Ist C_0 gleich 1, so wird der Multiplikand (Signal A) und das Zwischenergebnis addiert (MUL3). Ist C_O gleich 0, so wird das Zwischenergebnis um 1 Bit nach rechts verschoben. Die Fallunterscheidung findet in Zustand MUL2 statt. Bei der Multiplikation findet das im Calcontrol-Prozess fsm_count generierte Hilfssignal full Anwendung. Dieses dient als Zähler, ob n-1 Schritte für die

Multiplikation ausgeführt wurden (siehe 1.3, 2.1). Diese Fallunterscheidung findet im Zustand MUL4 statt.
Im Zustand MUL1 wird vom Multiplexer das Ausgangssignal der ALU durchgeschaltet (s0=1, s1=1), das C-Register und der High-Teil des Akkumulatorregisters auf 0 gesetzt und der Multiplikator (B) in den Low-Teil des Akkumulatorregisters geladen. Im Zustand MUL2 wird wiederum das Ergebnis der ALU vom Multiplexer durchgeschaltet. Zusätzlich findet die oben erwähnte Fallunterscheidung statt. Ist das LSB vom Low-Teil des Multiplexer 0, so wird das gerade durchgeschaltete Ergebnis der ALU um 1 Bit nach rechts geschoben. Dies findet im Zustand MUL4 statt. Dort wird ebenfalls das Ergebnis der ALU (dieses entspricht dem erwähnten Zwischenergebnis) durchgeschaltet, in den High-Teil Akkumulatorregisters geladen, das Carrybit wird auf 1 gesetzt und das C-Register bzw. der Low-Teil des Akkumulatorregisters wird auf 0 gesetzt. Das Signal shift wird auf 1 gesetzt (das Ergebnis wird um 1 Bit nach rechts geschoben) – ebenso das Hilfssignal schift1 (dieses wird für den Prozess fsm_count benötigt, da auf `shift : OUT std_logic` nicht schreibend zugegriffen werden kann) Dabei findet die zweite Fallunterscheidung bei der Multiplikation statt. Nämlich ob das oben erwähnte Hilfssignal full 0 oder 1 ist (n-1 Schritte durchgeführt). Ist full gleich 0, so wird zum Zustand MUL2 zurückgegangen und damit eine Addition vom Multiplikanden (A) und dem Zwischenergebnis durchgeführt. Die Zustände MUL2, MUL3 und MUL4 wiederholen sich nun, bis das Hilfssignal full gleich 1 ist. Erst dann wird in den finalen Zustand s_ready des Zustandsautomaten gegangen und das Ergebnis steht für die Ausgabe zur Verfügung.

2.2.5. Zustand s_ready

Im Zustand s_ready wird das Eingangssignal B (s0=1, s1=0) durchgeschaltet und das ready-Signal erhält den Wert 1 als Indikator, dass das Ergebnis der Rechenoperation zur Verfügung steht.

2.3. Calrtl

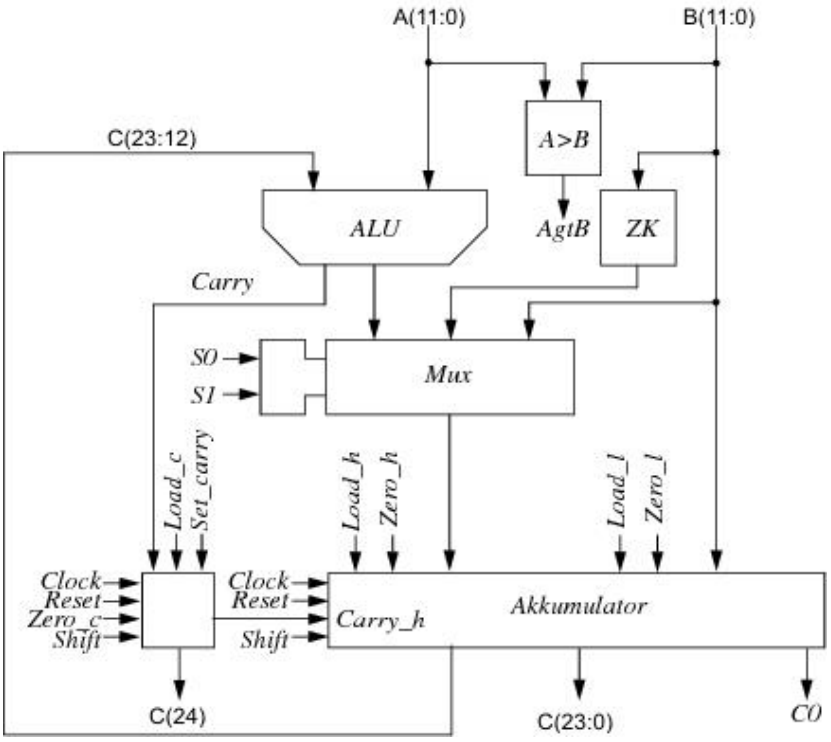
Der Calrtl-Block nimmt den Datenpfad im Calculator-Baustein ein. Das heißt, er übernimmt die Operanten (A und B) und führt die Operationen gemäß den aktiven Steuersignalen aus. Der Calrtl ist innerhalb in Modulen aufgeteilt, in welchen die Funktionen des Calrt-Blockes implementiert sind (Blockschaltbild, siehe Abbildung 8).

Die externen Signale werden in der Entity beschrieben durch:

Signal	Bedeutung
clk	Systemtakt (IN std_logic)
reset	Reset (IN std_logic)
s0	Steuersignal für MUX (IN std_logic)
s1	Steuersignal für MUX (IN std_logic)
set_carry	Steuersignal zum Carrybit setzten (IN std_logic)
zero_c	C-Register auf 0 setzten (IN std_logic)
zero_h	High Teil des Akkumulatorregisters auf 0 setzten (IN std_logic)
zero_l	Low Teil des Akkumulatorregisters auf 0 setzten (IN std_logic)
load_h	Daten in den High Teil des Akkumulatorregisters laden (IN std_logic)
load_l	Daten in den Low Teil des Akkumulatorregisters laden (IN std_logic)

load_c	Daten von ALU ins C-Register laden (IN std_logic)
shift	Ergebnis um 1 Bit nach rechts schieben (IN std_logic)
shift_12_0	Ergebnis um 12 nach rechts schieben und mit 0 auffüllen(IN std_logic)
shift_12_1	Ergebnis um 12 nach rechts schieben und mit 1 auffüllen(IN std_logic)
a	Operant a (IN std_logic_vector(11 DOWNTO 0))
b	Operant b (IN std_logic_vector(11 DOWNTO 0))
agtb_1	Steuersignal für Subtraktion (OUT std_logic)
c_0	Steuersignal vom Akkumulator (OUT std_logic)
c	Ergebnis der Rechenoperation (OUT std_logic_vector(24 DOWNTO 0))

- Tabelle 3: Externe Signale des Calrtl-Blocks -



- Abbildung 8 : Blockschaltbild Calrtl -

Die in Abbildung 8 zu sehenden Module sind in der Architecture des Calrtl-Blockes mittels Component Instanstiation implementiert und über Portmaps miteinander verdrahtet. Am Anfang der Architecture werden noch einige interne Signale deklariert, die nur innerhalb der Calrtl vorkommen.

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE work.ALL;

ENTITY calrtl IS
    PORT(  clk        : IN  std_logic;
           reset      : IN  std_logic;
           s0         : IN  std_logic;
```

```
          s1         : IN  std_logic;
          set_carry  : IN  std_logic;
          zero_c     : IN  std_logic;
          zero_h     : IN  std_logic;
          zero_l     : IN  std_logic;
          load_h     : IN  std_logic;
          load_l     : IN  std_logic;
          load_c     : IN  std_logic;
          shift      : IN  std_logic;
          shift_12_0 : IN  std_logic;
          shift_12_1 : IN  std_logic;
          a          : IN  std_logic_vector(11 DOWNTO 0);
          b          : IN  std_logic_vector(11 DOWNTO 0);
          agtb_1     : OUT STD_LOGIC;
          c_0        : OUT STD_LOGIC;
          c          : OUT std_logic_vector(24 DOWNTO 0) := "0000000000000000000000000");
END calrtl;

ARCHITECTURE behave OF calrtl IS
    SIGNAL carry      : STD_LOGIC;
    SIGNAL agtb       : STD_LOGIC;
    SIGNAL c_24       : STD_LOGIC;
    SIGNAL alu_out    : STD_LOGIC_VECTOR (11 DOWNTO 0);
    SIGNAL c_high     : STD_LOGIC_VECTOR (11 DOWNTO 0);
    SIGNAL zk_out     : STD_LOGIC_VECTOR (11 DOWNTO 0);
    SIGNAL mux_out    : STD_LOGIC_VECTOR (11 DOWNTO 0);
    SIGNAL c_23       : STD_LOGIC_VECTOR (23 DOWNTO 0);

COMPONENT alu
    PORT(  a          : IN  STD_LOGIC_VECTOR (11 DOWNTO 0);
           c_high     : IN  STD_LOGIC_VECTOR (11 DOWNTO 0);
           alu_out    : OUT STD_LOGIC_VECTOR (11 DOWNTO 0);
           carry      : OUT STD_LOGIC);
END COMPONENT;

COMPONENT zk
    PORT(  b          : IN  STD_LOGIC_VECTOR (11 DOWNTO 0);
           zk_out     : OUT STD_LOGIC_VECTOR (11 DOWNTO 0));
END COMPONENT;

COMPONENT mux
    PORT(  s0         : IN  STD_LOGIC;
           s1         : IN  STD_LOGIC;
           b          : IN  STD_LOGIC_VECTOR (11 DOWNTO 0);
           zk_out     : IN  STD_LOGIC_VECTOR (11 DOWNTO 0);
           alu_out    : IN  STD_LOGIC_VECTOR (11 DOWNTO 0);
           mux_out    : OUT STD_LOGIC_VECTOR (11 DOWNTO 0));
END COMPONENT;

COMPONENT akkumulator
    PORT(  clk        : IN  STD_LOGIC;
           reset      : IN  STD_LOGIC;
           zero_l     : IN  STD_LOGIC;
           zero_h     : IN  STD_LOGIC;
           load_l     : IN  STD_LOGIC;
           load_h     : IN  STD_LOGIC;
           shift      : IN  STD_LOGIC;
           shift_12_0 : IN  STD_LOGIC;
           shift_12_1 : IN  STD_LOGIC;
           c_24       : IN  STD_LOGIC;
           mux_out    : IN  STD_LOGIC_VECTOR (11 DOWNTO 0);
           b          : IN  STD_LOGIC_VECTOR (11 DOWNTO 0);
           c_0        : OUT STD_LOGIC;
           c_23       : OUT STD_LOGIC_VECTOR (23 DOWNTO 0);
           c_high     : OUT STD_LOGIC_VECTOR (11 DOWNTO 0));
END COMPONENT;

COMPONENT c_register
    PORT(  clk        : IN  STD_LOGIC;
           reset      : IN  STD_LOGIC;
           shift      : IN  STD_LOGIC;
```

```
           load_c     : IN  STD_LOGIC;
           zero_c     : IN  STD_LOGIC;
           carry      : IN  STD_LOGIC;
           set_carry  : IN  STD_LOGIC;
           c_24       : OUT STD_LOGIC);
END COMPONENT;

COMPONENT a_b
    PORT(  a          : IN  STD_LOGIC_VECTOR (11 DOWNTO 0);
           b          : IN  STD_LOGIC_VECTOR (11 DOWNTO 0);
           agtb       : OUT STD_LOGIC);
END COMPONENT a_b;

BEGIN
    alu1: alu port map (a => a, c_high => c_high, alu_out => alu_out, carry => carry);

    zwk1: zk  port map (b => b, zk_out => zk_out);

    mux1: mux port map (s0 => s0, s1 => s1, b => b, zk_out => zk_out, alu_out => alu_out,
mux_out => mux_out);

    akk1: akkumulator port map (clk => clk, reset => reset,
                                zero_l => zero_l, zero_h => zero_h, load_l => load_l, load_h
=> load_h, shift => shift, shift_12_0 => shift_12_0, shift_12_1 => shift_12_1,
                                c_24 => c_24, mux_out => mux_out, b => b, c_0 => c_0, c_23 =>
c_23, c_high => c_high);

    reg1: c_register  port map (clk => clk, reset => reset, zero_c => zero_c, shift => shift,
carry => carry,
                                load_c => load_c, set_carry => set_carry, c_24 => c_24);

    agb1: a_b  port map (a => a, b => b, agtb => agtb);

    agtb_1 <= agtb;
    c <= (c_24 & c_23(23 DOWNTO 0));

END behave;
```

- Code 4: Calrtl-Block -

Auf die einzelnen Module wird im Folgenden eingegangen.

2.3.1. ALU

Die arithmetic logical unit (ALU) ist der Teil des Calrtl-Blockes, in dem die Addition ausgeführt wird. Da, wie bereits im Kapitel 2.2 beschrieben, sämtliche Rechenoperationen auf einer Addition zurückzuführen sind, findet hier also auch für die Subtraktion (Addition von A und dem Zweierkomplement von B) und die Multiplikation die Berechnung der (Zwischen-) Ergebnisse statt. Die ALU wurde mit folgendem VHDL-Code beschrieben:

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE work.ALL;

ENTITY alu IS
    PORT(  a          : IN  STD_LOGIC_VECTOR (11 DOWNTO 0);
           c_high     : IN  STD_LOGIC_VECTOR (11 DOWNTO 0);
           alu_out    : OUT STD_LOGIC_VECTOR (11 DOWNTO 0);
           carry      : OUT STD_LOGIC);
END alu;
```

```
ARCHITECTURE behave OF alu IS
SIGNAL and_s : std_logic_vector(11 DOWNTO 0);
SIGNAL xor_s : std_logic_vector(11 DOWNTO 0);
SIGNAL ueb_s : std_logic_vector(11 DOWNTO 0);

BEGIN
    and_s       <= a AND c_high;
    xor_s       <= a XOR c_high;
    alu_out(0)  <= xor_s(0);
    ueb_s(0)    <= and_s(0);
    alu_out(1)  <= xor_s(1)  XOR ueb_s(0);
    ueb_s(1)    <= and_s(1)  OR  (xor_s(1)  AND ueb_s(0));
    alu_out(2)  <= xor_s(2)  XOR ueb_s(1);
    ueb_s(2)    <= and_s(2)  OR  (xor_s(2)  AND ueb_s(1));
    alu_out(3)  <= xor_s(3)  XOR ueb_s(2);
    ueb_s(3)    <= and_s(3)  OR  (xor_s(3)  AND ueb_s(2));
    alu_out(4)  <= xor_s(4)  XOR ueb_s(3);
    ueb_s(4)    <= and_s(4)  OR  (xor_s(4)  AND ueb_s(3));
    alu_out(5)  <= xor_s(5)  XOR ueb_s(4);
    ueb_s(5)    <= and_s(5)  OR  (xor_s(5)  AND ueb_s(4));
    alu_out(6)  <= xor_s(6)  XOR ueb_s(5);
    ueb_s(6)    <= and_s(6)  OR  (xor_s(6)  AND ueb_s(5));
    alu_out(7)  <= xor_s(7)  XOR ueb_s(6);
    ueb_s(7)    <= and_s(7)  OR  (xor_s(7)  AND ueb_s(6));
    alu_out(8)  <= xor_s(8)  XOR ueb_s(7);
    ueb_s(8)    <= and_s(8)  OR  (xor_s(8)  AND ueb_s(7));
    alu_out(9)  <= xor_s(9)  XOR ueb_s(8);
    ueb_s(9)    <= and_s(9)  OR  (xor_s(9)  AND ueb_s(8));
    alu_out(10) <= xor_s(10) XOR ueb_s(9);
    ueb_s(10)   <= and_s(10) OR  (xor_s(10) AND ueb_s(9));
    alu_out(11) <= xor_s(11) XOR ueb_s(10);
    ueb_s(11)   <= and_s(11) OR  (xor_s(11) AND ueb_s(10));
    carry       <= ueb_s(11);
END behave;
```

- Code 5: ALU -

Die ALU besteht aus den externen Signalen a (`IN STD_LOGIC_VECTOR (11 DOWNTO 0)`, Eingang für den Operanten A), c_high (`IN STD_LOGIC_VECTOR (11 DOWNTO 0)`, der zweite Operand für die ALU – bestehend aus dem High-Teil des Akkumulatorregisters (C(23:12)), alu_out (`OUT STD_LOGIC_VECTOR (11 DOWNTO 0)`, dem Ausgang der ALU an dem das Ergebnis der Addition anliegt) und dem Ausgangssignal carry (`OUT STD_LOGIC`, dem Übertrag).
Neben den in der Portlist der Entity beschriebenen Signalen werden für die Funktionserfüllung noch die internen Signale and_s, xor_s und ueb_s (jeweils `std_logic_vector(11 DOWNTO 0)`) benötigt.
Das Signal and_s entsteht aus der and-Verknüpfung von a und c_high. Das Signal xor_s entsteht aus der XOR-Verknüpfung der beiden Eingangssignale.

Die Addition geschieht in der ALU bitweise:

```
alu_out(0)      <= xor_s(0);
    ueb_s(0)    <= and_s(0);
    alu_out(1)  <= xor_s(1)  XOR ueb_s(0);
    ueb_s(1)    <= and_s(1)  OR  (xor_s(1)  AND ueb_s(0));
        .
        .
        .
    alu_out(11) <= xor_s(11) XOR ueb_s(10);
    ueb_s(11)   <= and_s(11) OR  (xor_s(11) AND ueb_s(10));
    carry       <= ueb_s(11);
```

Zum Schluss bleibt das 12-te Bit von ueb_s übrig, dieses wird auf den carry-Ausgang der ALU durchgeschaltet.

2.3.2. ZK-Block

Im ZK-Block wird das Zweierkomplement des Operanten B gebildet. Wie bereits beschrieben, wird dieses zur Subtraktion benötigt. Demnach besitzt dieser Block nur jeweils 1 Eingangs- (b) und 1 Ausgangssignal (zk_out). Das Zweierkomplement entsteht aus der Negation des Eingangssignals und anschließender Addition von 1:

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.std_logic_unsigned.ALL;
USE work.ALL;

ENTITY zk IS
    PORT(  b          : IN  STD_LOGIC_VECTOR (11 DOWNTO 0);
           zk_out     : OUT STD_LOGIC_VECTOR (11 DOWNTO 0));
END zk;

ARCHITECTURE behave OF zk IS

BEGIN
    zk_out <= ((NOT b) + 1);
END behave;
```

- Code 6: ZK-Block -

2.3.3. Mux

Der Multiplexer schaltet wahlweise anhand der Steuersignale s0 und s1 (erzeugt von Calcontrol) die Signale b (Operand B), zk_out (Ausgangssignal des Blockes ZK) oder alu_out (Ausgangssignal des Blockes ALU) auf seinen Ausgang mux_out durch.

Durchgeschaltetes Signal	s0	s1
zk_out	0	0
b	1	0
alu_out	X	1

- Tabelle 4: MUX -

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.std_logic_unsigned.ALL;
USE work.ALL;

ENTITY mux IS
    PORT(  s0         : IN  STD_LOGIC;
           s1         : IN  STD_LOGIC;
           b          : IN  STD_LOGIC_VECTOR (11 DOWNTO 0);
           zk_out     : IN  STD_LOGIC_VECTOR (11 DOWNTO 0);
           alu_out    : IN  STD_LOGIC_VECTOR (11 DOWNTO 0);
           mux_out    : OUT STD_LOGIC_VECTOR (11 DOWNTO 0));
END mux;
```

```
ARCHITECTURE behave OF mux IS

BEGIN
    mux_out <= zk_out WHEN s0 = '0' AND s1 = '0'
    ELSE b WHEN s0 = '1' AND s1 = '0'
    ELSE alu_out WHEN s1 = '1'
    ELSE "000000000000";
END behave;
```

- Code 7: MUX -

2.3.4. Akkumulator

Der Akkumulator ist ein 24 Bit breites Register zum speichern und ausgeben der Daten. Intern ist er unterteilt in den High- (C(23:12)) und dem Low-Teil (C(11:0).

In der Entity werden die Schnittstellen beschrieben:

Signal	Bedeutung
clk	Systemtakt (IN std_logic)
reset	Reset (IN std_logic)
zero_l	Low Teil des Akkumulatorregisters auf 0 setzten (IN std_logic)
zero_h	High Teil des Akkumulatorregisters auf 0 setzten (IN std_logic)
load_l	Daten in den Low Teil des Akkumulatorregisters laden (IN std_logic)
load_h	Daten in den High Teil des Akkumulatorregisters laden (IN std_logic)
shift	Daten um 1 Bit nach rechts schieben (IN std_logic)
shift_12_0	Daten um 12 nach rechts schieben und mit 0 auffüllen(IN std_logic)
shift_12_1	Daten um 12 nach rechts schieben und mit 1 auffüllen(IN std_logic)
c_24	Das Carrybit vom C-Register (IN std_logic)
mux_out	Der Ausgang vom Multiplexer (IN std_logic))
b	Operant b (IN std_logic_vector(11 DOWNTO 0))
c_0	Steuersignal vom Akkumulator (OUT std_logic)
c_23	Ausgang des Akkumulators (OUT std_logic_vector(23 DOWNTO 0))
c_high	Ausgang des High-Teils des Akkumulators OUT STD_LOGIC_VECTOR (11 DOWNTO 0)

- Tabelle 5: Schnittstellen des Akkumulators -

Aufgrund der Unterteilung des Akkumulators in eine High- und einem Low-Teil, ist es notwendig diese auch separat ansprechen zu können. Daher sind die „Anweisungen" zero bzw. load für jede dieser Teile vorhanden. Zero (_l/_h) besagt, dass jeweils der komplette Registerteil auf Null gesetzt werden soll. Mit dem Befehl load_l wird der Operand B in den Low-Teil geladen. Mit dem Befehl load_h wird das Ausgangssignal vom Multiplexer (mux_out) in den High-Teil des Akkumulators geladen.
Das Steuersignal shift besagt, dass die Daten des gesamten Registers (C(23:0)) bitweise um 1 Stelle nach rechts verschoben werden sollen. Mit shift_12_0 werden die Daten aus dem Low-Teil in den High-Teil übernommen, der High-Teil wird auf 0 gesetzt und für das LSB des Low-Teils wird das Carrybit gesetzt (C24). Der Befehl shift_12_1 besagt, dass die Daten aus dem High-Teil in den Low-Teil übernommen werden sollen und der High-

Teil auf 1 gesetzt werden soll (diese Operation ist bei einem negativen Ergebnis notwendig).
Als Ausgangssignale stehen dem Akkumulator c_0, c_23 und c_high zur Verfügung. Das Ausgangssignal c_23 repräsentiert das Ergebnis der Rechenoperationen. Mit c_high gelangen die Daten des High-Teils (C(23:12)) vom Akkumulator in die ALU. Das LSB des Low-Teiles (C0) wird einzeln herausgeführt, da dieses als Steuersignal für den Calcontrol-Block bei der Multiplikation benötigt wird (vgl. Abschnitt 2.2.4).

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.std_logic_unsigned.ALL;
USE work.ALL;

ENTITY akkumulator IS
    PORT(  clk        : IN  STD_LOGIC;
           reset      : IN  STD_LOGIC;
           zero_l     : IN  STD_LOGIC;
           zero_h     : IN  STD_LOGIC;
           load_l     : IN  STD_LOGIC;
           load_h     : IN  STD_LOGIC;
           shift      : IN  STD_LOGIC;
           shift_12_0 : IN  STD_LOGIC;
           shift_12_1 : IN  STD_LOGIC;
           c_24       : IN  STD_LOGIC;
           mux_out    : IN  STD_LOGIC_VECTOR (11 DOWNTO 0);
           b          : IN  STD_LOGIC_VECTOR (11 DOWNTO 0);
           c_0        : OUT STD_LOGIC;
           c_23       : OUT STD_LOGIC_VECTOR (23 DOWNTO 0);
           c_high     : OUT STD_LOGIC_VECTOR (11 DOWNTO 0));
END akkumulator;

ARCHITECTURE behave OF akkumulator IS
SIGNAL c_hx : STD_LOGIC_VECTOR (11 DOWNTO 0);
SIGNAL c_lx : STD_LOGIC_VECTOR (11 DOWNTO 0);

BEGIN
process (clk)
    BEGIN
        IF (clk'event AND clk = '1') THEN
            IF (reset = '1') THEN
                c_lx <= (OTHERS => '0');
                c_hx <= (OTHERS => '0');
            ELSE
                IF (zero_l = '1') THEN
                    c_lx <= "000000000000";
                END IF;
                IF (zero_h = '1') THEN
                    c_hx <= "000000000000";
                END IF;
                IF (load_l = '1') THEN
                    c_lx <= b;
                END IF;
                IF (load_h = '1') THEN
                    c_hx <= mux_out;
                END IF;
                IF (SHIFT = '1') THEN
                    c_lx(0)  <= c_lx(1);
                    c_lx(1)  <= c_lx(2);
                    c_lx(2)  <= c_lx(3);
                    c_lx(3)  <= c_lx(4);
                    c_lx(4)  <= c_lx(5);
                    c_lx(5)  <= c_lx(6);
                    c_lx(6)  <= c_lx(7);
                    c_lx(7)  <= c_lx(8);
                    c_lx(8)  <= c_lx(9);
                    c_lx(9)  <= c_lx(10);
                    c_lx(10) <= c_lx(11);
                    c_lx(11) <= c_hx(0);
```

```
                    c_hx(0)  <= c_hx(1);
                    c_hx(1)  <= c_hx(2);
                    c_hx(2)  <= c_hx(3);
                    c_hx(3)  <= c_hx(4);
                    c_hx(4)  <= c_hx(5);
                    c_hx(5)  <= c_hx(6);
                    c_hx(6)  <= c_hx(7);
                    c_hx(7)  <= c_hx(8);
                    c_hx(8)  <= c_hx(9);
                    c_hx(9)  <= c_hx(10);
                    c_hx(10) <= c_hx(11);
                    c_hx(11) <= c_24;
                END IF;
                IF (shift_12_0 = '1') THEN
                    c_lx     <= c_hx;
                    c_hx     <= "000000000000";
                    c_hx(0)  <= c_24;
                END IF;
                IF (shift_12_1 = '1') THEN
                    c_lx     <= c_hx;
                    c_hx     <= "111111111111";
                END IF;
            END IF;
        END IF;
END PROCESS;

c_0    <= c_lx(0);
c_high <= c_hx;
c_23   <= (c_hx(11 DOWNTO 0) & c_lx(11 DOWNTO 0) );

END behave;
```

- Code 8: Akkumulator -

2.3.5. C-Register

Das Carry-Register ist verantwortlich für das richtige Setzten des Carrybits. Für die Entscheidung werden Steuersignale shift, load_c, zero_c und set_carry vom Calcontrol-Block und das Signa carry von der ALU ausgewertet und das Carrybit mit c_24 ausgegeben.

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.std_logic_unsigned.ALL;
USE work.ALL;

ENTITY c_register IS
    PORT(  clk        : IN  STD_LOGIC;
           reset      : IN  STD_LOGIC;
           shift      : IN  STD_LOGIC;
           load_c     : IN  STD_LOGIC;
           zero_c     : IN  STD_LOGIC;
           carry      : IN  STD_LOGIC;
           set_carry  : IN  STD_LOGIC;
           c_24       : OUT STD_LOGIC);
END c_register;

ARCHITECTURE behave OF c_register IS

BEGIN

PROCESS (clk)
    BEGIN
```

```
        IF (clk'event AND clk = '1') THEN
            IF (reset = '1') THEN
                c_24 <= '0';
            ELSE
                IF (set_carry = '1') THEN
                    c_24 <= '1';
                END IF;
                IF (zero_c = '1' OR shift = '1') THEN
                    c_24 <= '0';
                END IF;
                IF (load_c = '1') THEN
                    c_24 <= carry;
                END IF;
            END IF;
        END IF;
END PROCESS;

END behave;
```

- Code 9: C-Register -

Das Carrybit ist 1, falls das Steuersignal set_carry 1 ist (im Zustand SUB4, d.h. das Ergebnis ist negativ, und im Zustand MUL4 von Calcontrol). Das Carrybit ist 0, falls das C-Register auf 0 gesetzt wird und wenn die Daten um 1 Bit nach rechts geschoben werden (im Zustand MUL4 von Calcontrol, wenn das Zwischenergebnis um 1 Bit nach rechts geschoben wird). Falls das Carrybit geladen werden soll (load_c), so hängt das das Carrybit (c_24) natürlich vom Carrybit der ALU (carry) ab und bekommt dessen Wert.

2.3.6. A>B-Block

Der A>B-Block stellt einen Größenvergleich zwischen den Operanten A und B an und erzeugt daraus das Steuersignal agtb. Dieses Signal wird bei der Subtraktion im Zustand SUB2 ausgewertet und kennzeichnet, ob das Ergebnis positiv oder negativ wird.

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.std_logic_unsigned.ALL;
USE work.ALL;

ENTITY a_b IS
    PORT(  a          : IN  STD_LOGIC_VECTOR (11 DOWNTO 0);
           b          : IN  STD_LOGIC_VECTOR (11 DOWNTO 0);
           agtb       : OUT STD_LOGIC);
END a_b;

ARCHITECTURE behave OF a_b IS

BEGIN
    agtb <= '1' WHEN (a < b)
    ELSE '0';
END behave;
```

- Code 10: A>B-Block -

2.4. Calculator

Um die Funktionalität des Calculators herzustellen ist es notwendig die in den Abschnitten 2.1, 2.2 und 2.3 beschriebenen Blöcke zusammenzuführen. Getestet wird der Calculator über eine Testbench.

2.4.1. Calculator

In der Portmap der Entity werden die externen Schnittstellen wie sie in der Aufgabenstellung spezifiziert wurden (Abschnitt 1) deklariert. In der Architecture werden die einzelnen Blöcke mittels Component Instantiation hinzugefügt und mit der Portmap verschalten. Die für die Verschaltung notwendigen internen Signale müssen ebenfalls in der Architecture deklariert werden (vergleiche Abschnitt 1.4).

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE work.ALL;

ENTITY calculator IS
    PORT(  clk        : IN  std_logic;
           reset      : IN  std_logic;
           add        : IN  std_logic;
           sub        : IN  std_logic;
           mul        : IN  std_logic;
           ready      : OUT std_logic;
           a          : IN  std_logic_vector(11 DOWNTO 0);
           b          : IN  std_logic_vector(11 DOWNTO 0);
           c          : OUT std_logic_vector(24 DOWNTO 0) );
END calculator;

ARCHITECTURE behave OF calculator IS
SIGNAL add_s, sub_s, mul_s : std_logic;
SIGNAL c_0, s0, s1, set_carry, agtb_1 : std_logic;
SIGNAL zero_c, zero_h, zero_l, load_c, load_h, load_l, shift, shift_12_0, shift_12_1 :
std_logic;

COMPONENT sync
    PORT(  clk        : IN  std_logic;
           reset      : IN  std_logic;
           add        : IN  std_logic := '0';
           sub        : IN  std_logic := '0';
           mul        : IN  std_logic := '0';
           add_s      : OUT std_logic;
           sub_s      : OUT std_logic;
           mul_s      : OUT std_logic);
END COMPONENT;

COMPONENT calcontrol
    PORT(  clk        : IN  std_logic;
           reset      : IN  std_logic;
           c_0        : IN  std_logic;
           add_s      : IN  std_logic;
           sub_s      : IN  std_logic;
           mul_s      : IN  std_logic;
           agtb_1     : IN  STD_LOGIC;
           s0         : OUT std_logic;
           s1         : OUT std_logic;
           set_carry  : OUT std_logic;
           zero_c     : OUT std_logic;
           zero_h     : OUT std_logic;
           zero_l     : OUT std_logic;
           load_c     : OUT std_logic;
```

```
           load_h     : OUT std_logic;
           load_l     : OUT std_logic;
           shift      : OUT std_logic;
           shift_12_0 : OUT std_logic;
           shift_12_1 : OUT std_logic;
           ready      : OUT std_logic);
END COMPONENT;

COMPONENT calrtl
    PORT(  clk        : IN  std_logic;
           reset      : IN  std_logic;
           s0         : IN  std_logic;
           s1         : IN  std_logic;
           set_carry  : IN  std_logic;
           zero_c     : IN  std_logic;
           zero_h     : IN  std_logic;
           zero_l     : IN  std_logic;
           load_h     : IN  std_logic;
           load_l     : IN  std_logic;
           load_c     : IN  std_logic;
           shift      : IN  std_logic;
           shift_12_0 : IN  std_logic;
           shift_12_1 : IN  std_logic;
           a          : IN  std_logic_vector(11 DOWNTO 0);
           b          : IN  std_logic_vector(11 DOWNTO 0);
           agtb_1     : OUT STD_LOGIC;
           c_0        : OUT STD_LOGIC;
           c          : OUT std_logic_vector(24 DOWNTO 0));
END COMPONENT;

BEGIN

sync1: sync       port map (clk => clk, reset => reset,
                            add => add, sub => sub, mul => mul,
                            add_s => add_s, sub_s => sub_s, mul_s => mul_s);
calc1: calcontrol port map (clk => clk, reset => reset,
                            c_0 => c_0, add_s => add_s, sub_s => sub_s, mul_s => mul_s,
                            s0 => s0, s1 => s1, set_carry => set_carry, agtb_1 => agtb_1,
                            zero_c => zero_c, zero_h => zero_h, zero_l => zero_l,
                            load_c => load_c, load_h => load_h, load_l => load_l,
                            shift => shift, shift_12_0 => shift_12_0, shift_12_1 =>
shift_12_1, ready => ready);
calr1: calrtl     port map (clk => clk, reset => reset,
                            c_0 => c_0, s0 => s0, s1 => s1, set_carry => set_carry, agtb_1 =>
agtb_1,
                            zero_c => zero_c, zero_h => zero_h, zero_l => zero_l,
                            load_h => load_h, load_l => load_l, load_c => load_c,
                            shift => shift, shift_12_0 => shift_12_0, shift_12_1 =>
shift_12_1, a => a, b => b, c => c);

END behave;
```

- Code 11: Calculator -

2.4.2. Testbench

Die Testbench dient zur Simulation des Calculators. Mit ihr werden sämtliche externe Signale erzeugt.

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;

ENTITY calculator_tb IS
END calculator_tb;
```

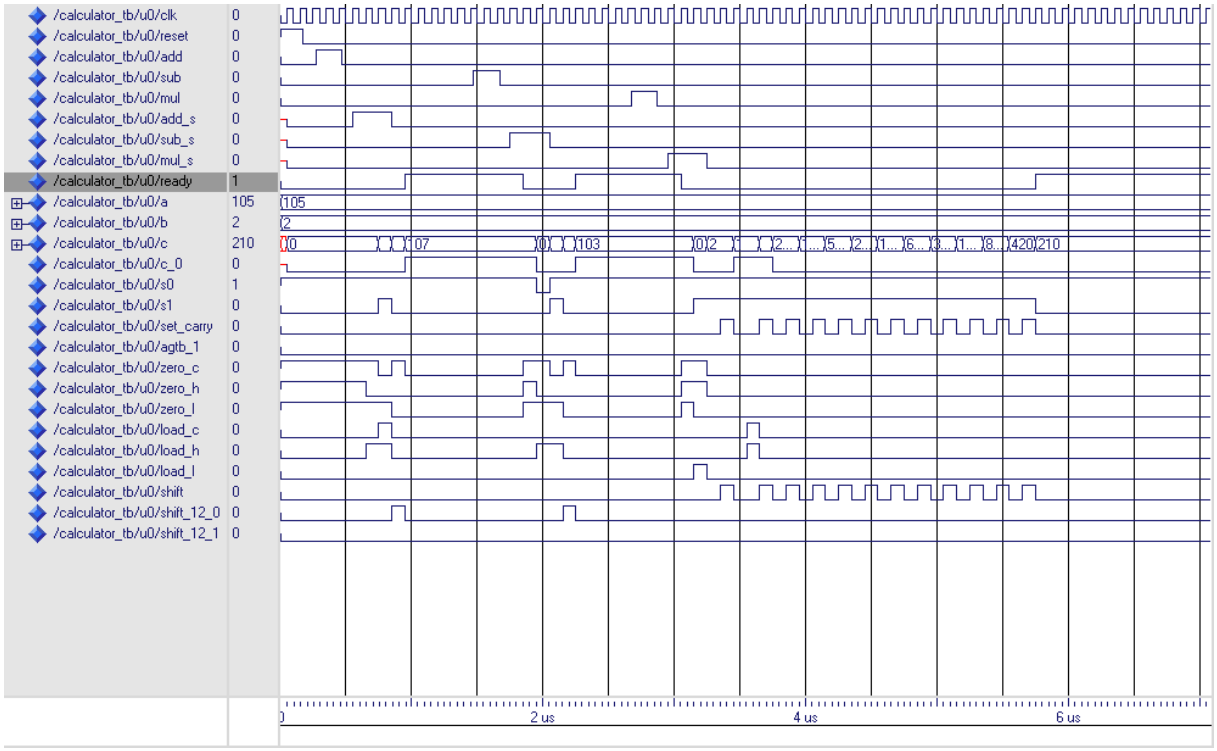
```
ARCHITECTURE structural OF calculator_tb IS
SIGNAL clk, reset, add, sub, mul, ready : std_logic;
SIGNAL a, b : std_logic_vector (11 DOWNTO 0);
SIGNAL c : std_logic_vector (24 DOWNTO 0);

COMPONENT calculator
    PORT(  clk        : IN  std_logic;
           reset      : IN  std_logic;
           add        : IN  std_logic;
           sub        : IN  std_logic;
           mul        : IN  std_logic;
           ready      : OUT std_logic;
           a          : IN  std_logic_vector(11 DOWNTO 0);
           b          : IN  std_logic_vector(11 DOWNTO 0);
           c          : OUT std_logic_vector(24 DOWNTO 0) );
END COMPONENT;

BEGIN
    U0: calculator port map (clk => clk, reset => reset,
                             add => add, sub => sub, mul => mul,
                             ready => ready, a => a, b => b, c => c);

p_clk: PROCESS
    BEGIN
        clk <= '0', '1' AFTER 50 ns;
        WAIT FOR 100 ns;
END PROCESS p_clk;

p_res: PROCESS
    BEGIN
        reset <= '1', '0' AFTER 170 ns;
        WAIT FOR 50000 ns;
END PROCESS p_res;

p_add: PROCESS
    BEGIN
        add <= '0';
        sub <= '0';
        mul <= '0';
        WAIT FOR 270 ns;  add <= '1';
        WAIT FOR 200 ns;  add <= '0';
        WAIT FOR 1000 ns; sub <= '1';
        WAIT FOR 200 ns;  sub <= '0';
        WAIT FOR 1000 ns; mul <= '1';
        WAIT FOR 200 ns;  mul <= '0';
        WAIT;
END PROCESS p_add;

p_a_b: PROCESS
    BEGIN
        --a <= "000000000011"; b <= "000000000100"; --3,4 (7,-1,12)
        --a <= "000001101001"; b <= "000000000010"; --105,2(107,103,210)
        --a <= "000011001111"; b <= "000011110000"; --207,240(447,-33,49680)
        --a <= "000011111111"; b <= "000011111111"; --255,255(510,0,65025)
        --a <= "000000101111"; b <= "000000011011"; --47,27(74,20,1269)
        --a <= "111111111111"; b <= "111111111111"; --4095,4095(8190,0,16769025)
        --a <= "111111111110"; b <= "111111111111"; --4094,4095(8189,-1,16764930)
        --a <= "000000000010"; b <= "111111111111"; --2,4095(4097,-4093,8190)
        --a <= "111111111111"; b <= "000000000010"; --4095,2(4097,4093,8190)




        WAIT;
END PROCESS p_a_b;

END structural;
```

- Code 12: Calculator Testbench -

Für die Erzeugung der externen Signale (Stimuli) wurden vier Prozesse eingeführt. Mit dem Prozess p_clk wird der Systemtakt erzeugt. Die Impulsbreite beträgt 50 ns – hieraus resultiert eine Frequenz von 10 MHz. Der Prozess p_res erzeugt den Reset. Anfangs ist er 1, nach 170 ns 0. Nach weiteren 50 µs wiedererholt er sich. Mit dem Prozess p_add werden die asynchronen Steuersignale ADD, SUB und MUL erzeugt Diese treten pro Simulation nur jeweils 1 Mal auf (ADD nach 270 ns für 200 ns, SUB nach 1470 ns für 200 ns und MUL nach 2670 ns für 200 ns). Der Prozess p_a_b generiert die Operanten A und B. Diese liegen während der gesamten Simulation an. Für Testzwecke sind im Code 12 mehrere Varianten der Operanten aufgeführt – dabei ist vor der Simulation jeweils das Kommentarsymbol zu entfernen. Im Rahmen des Projektes wurden alle Varianten simuliert. Die Ergebnisse sind im Abschnitt 3 zu finden.

3 Simulation

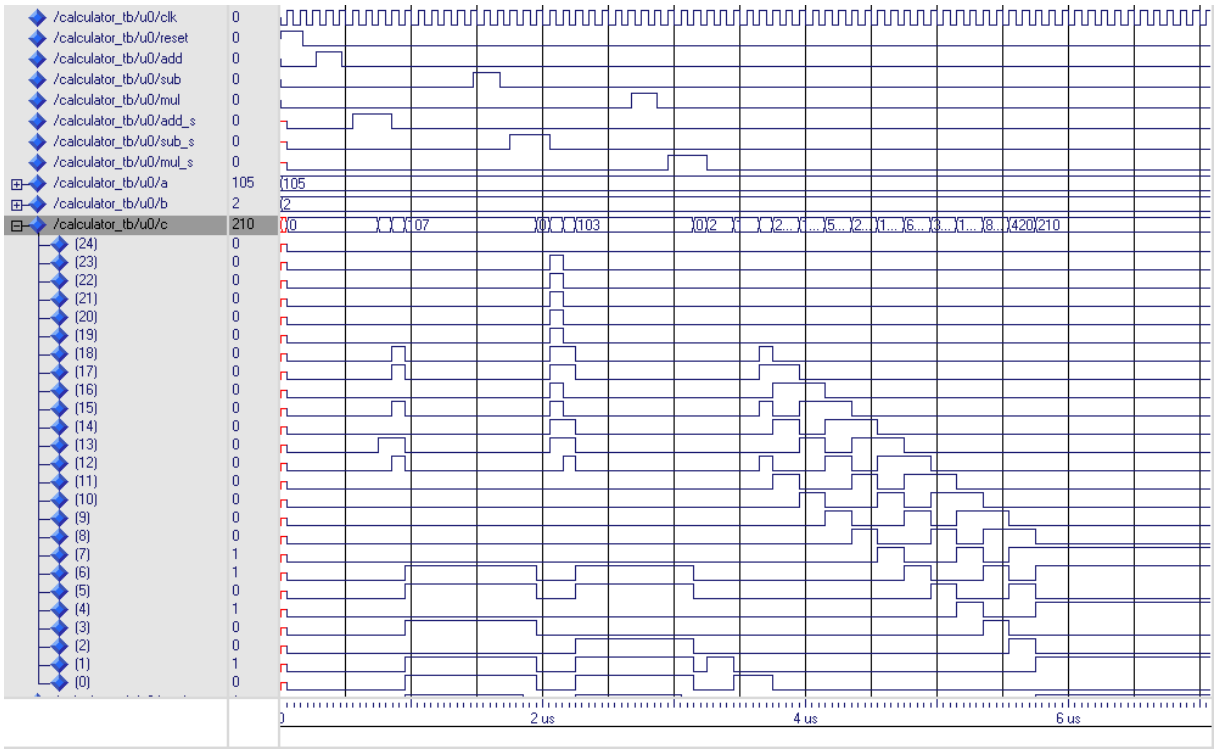
3.1. a=105, b=2



- Abbildung 8: a=105, b=2 -

Die erwarteten Ergebnisse der Rechenoperationen (Addition: 107, Subtraktion: 103, Multiplikation: 210) wurden bestätig.

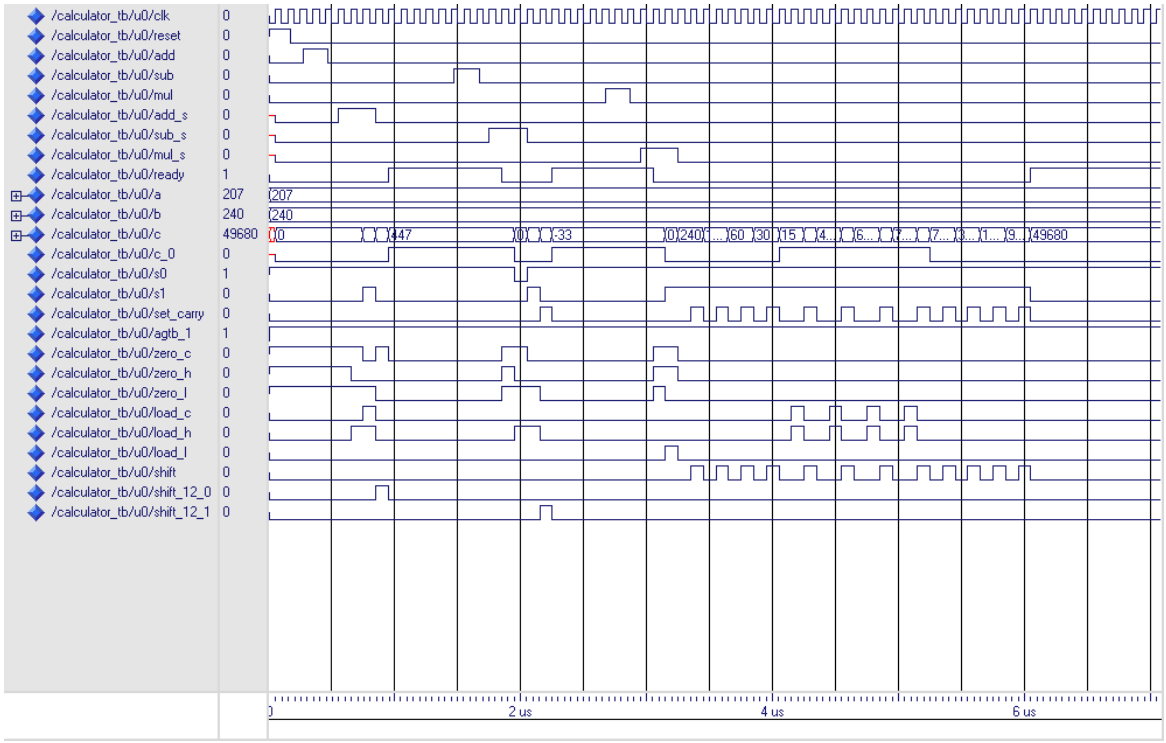
Für die Addition und Subtraktion werden 3 Taktperioden benötigt (3 Zustände im Calcontrol-Block), für die Multiplikation 14.



- Abbildung 9: a=105, b=2 (Ausschnitt) -

Hier wurde das gesamte Akkumulatorregister c sichtbar gemacht. Dabei ist gut zu erkennen, dass bei der Addition und Subtraktion das Ergebnis bereits beim Ausführungstakt im High-Teil des Akkumulatorregisters anliegt und zum 3. Ausführungstakt in den Low-Teil geschoben wird. Bei der Multiplikation ist das System der Schiebung / Addition des Zwischenergebnisses ebenfalls gut zu erkennen.

3.2. a=207, b=240

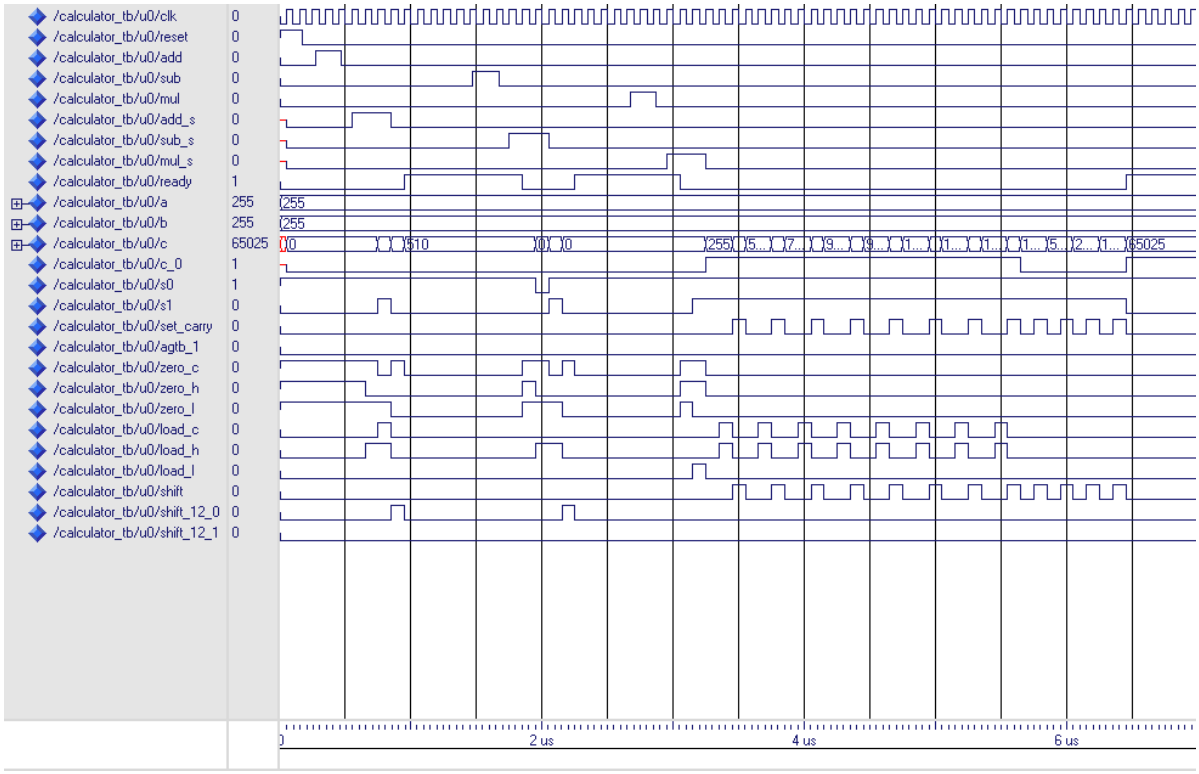


- Abbildung 10: a=207, b=240 -

Die erwarteten Ergebnisse der Rechenoperationen (Addition: 447, Subtraktion: -33, Multiplikation: 49680) wurden bestätig.

Das set_carry Signal ist zum 3. Ausführungstakt der Subtraktion aktiv (Zustand SUB4, da Ergebnis negativ, vgl. Signal agtb_1 (a>b-Vergleich)) und bei jedem Zustand MUL4 (das Zwischenergebnis wird um 1 Bit nach rechts geschoben).

3.3. a=255, b=255

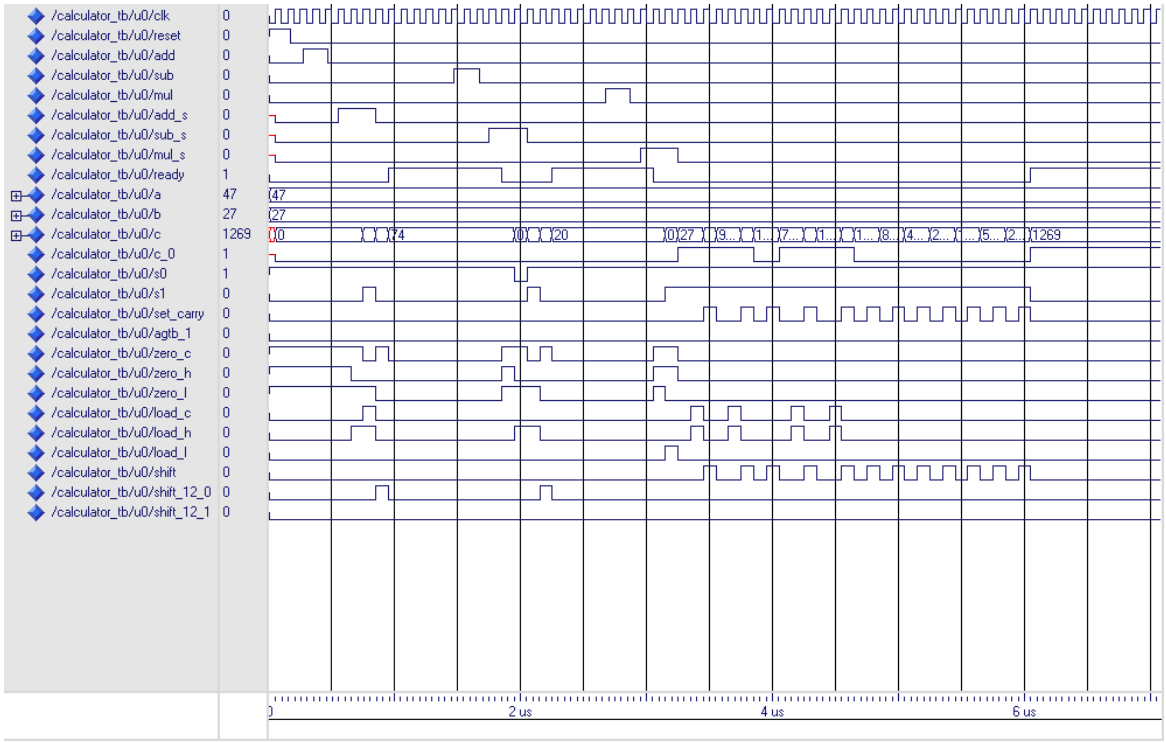


- Abbildung 11: a=255, b=255 -

Die erwarteten Ergebnisse der Rechenoperationen (Addition: 510, Subtraktion: 0, Multiplikation: 65025) wurden bestätig.

Das Ergebnis der Subtraktion ist 0 (also positiv), demnach ist jeweils im 3. Ausführungstakt der Addition und Subtraktion das Signal shift_12_0 aktiv.

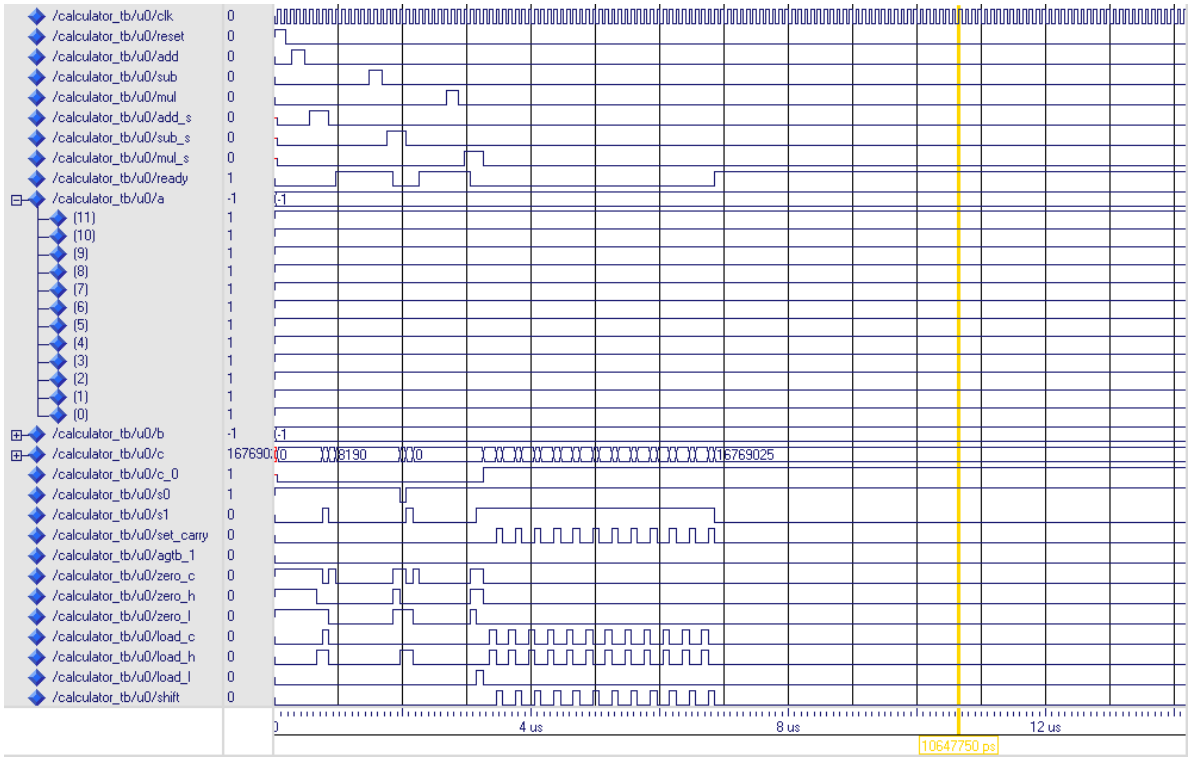
3.4. a=47, b=27



- Abbildung 12: a=47, b=27 -

Die erwarteten Ergebnisse der Rechenoperationen (Addition: 74, Subtraktion: 20, Multiplikation: 1269) wurden bestätig.

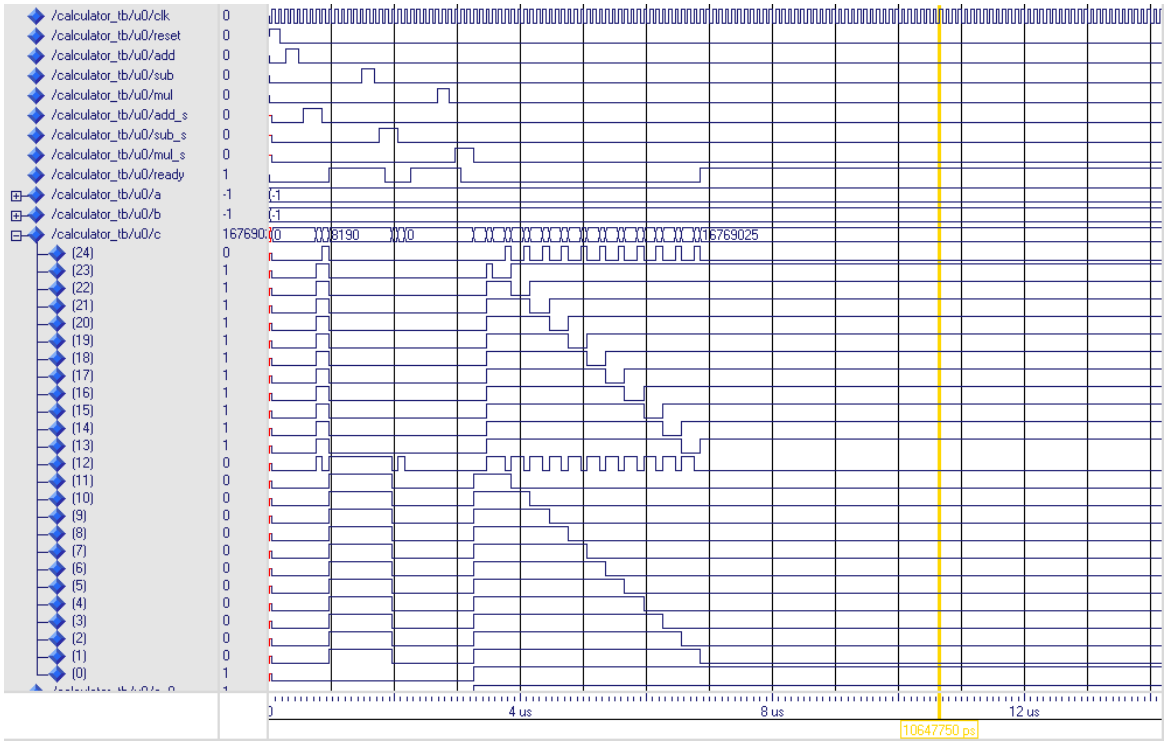
3.5. a=4095, b=4095



- Abbildung 13: a=4095, b=4095 -

Die erwarteten Ergebnisse der Rechenoperationen (Addition: 8190, Subtraktion: 0, Multiplikation: 16769025) wurden bestätig.

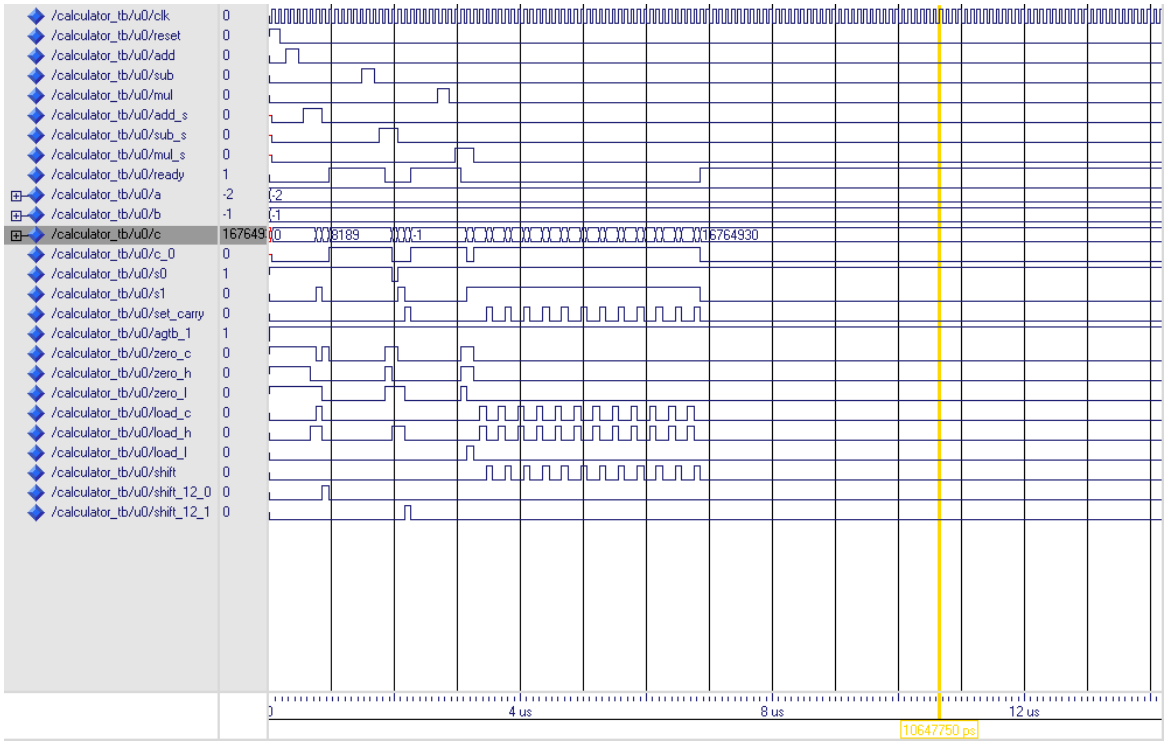
Die Operanden A und B (4095 = 2^{12}-1 entspricht der oberen Grenze des Wertebereichs für ein 12-Bit breites Binärsignal) werden bei der Simulation dezimal als -1 dargestellt, da das binär zu 111111111111 codierte Signal vom Simulator vorzeichenbehaftet interpretiert wird. Abhilfe würde an dieser Stelle die Einstellung des Radixs auf unsigned zu ändern schaffen. Für die durchgeführten Rechenoperationen spielt dies allerdings keine Rolle, da die Eingangssignale unsigned Integerwerten entsprechen.



- Abbildung 14: a=4095, b=4095 (Ausschnitt)-

Hier ist nochmals gut die Funktionsweise der Multiplikation zu erkennen. Die Multiplikation benötigt nun wesentlich mehr Operationen als noch unter 3.1. Entsprechend länger dauert es, bis die Berechnung abgeschlossen ist.

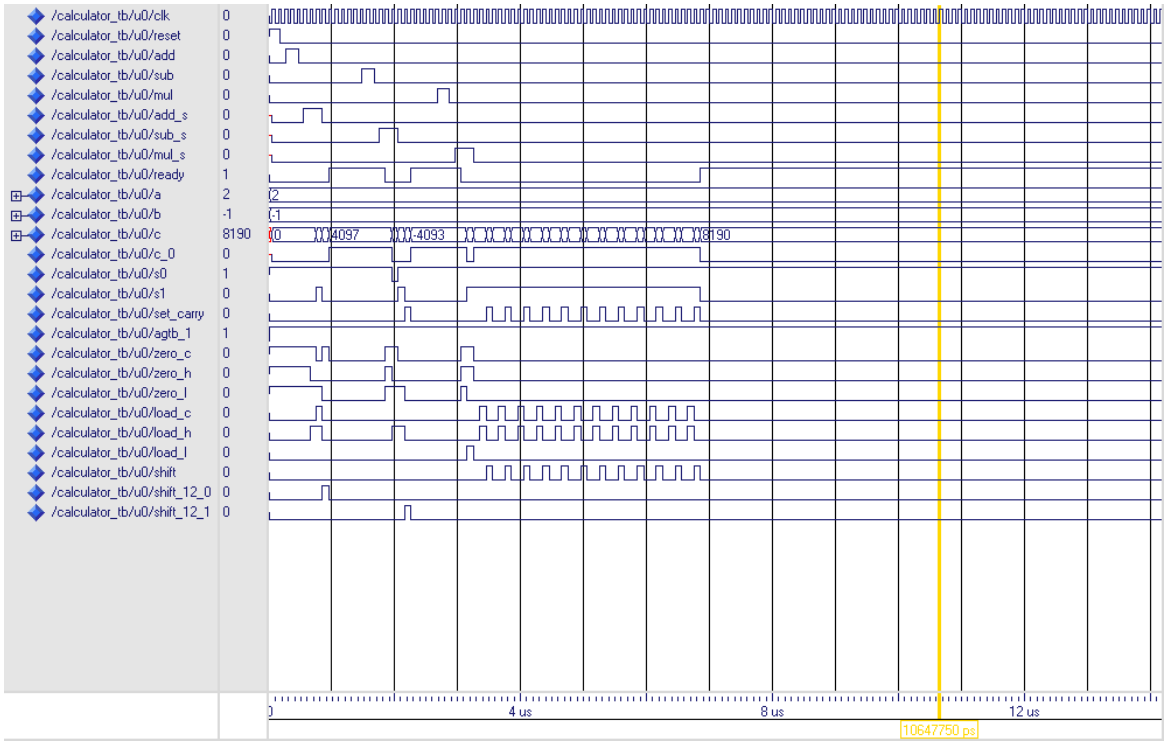
3.6. a=4094, b=4095



- Abbildung 15: a=4094, b=4095 -

Die erwarteten Ergebnisse der Rechenoperationen (Addition: 8189, Subtraktion: -1, Multiplikation: 16764930) wurden bestätig.

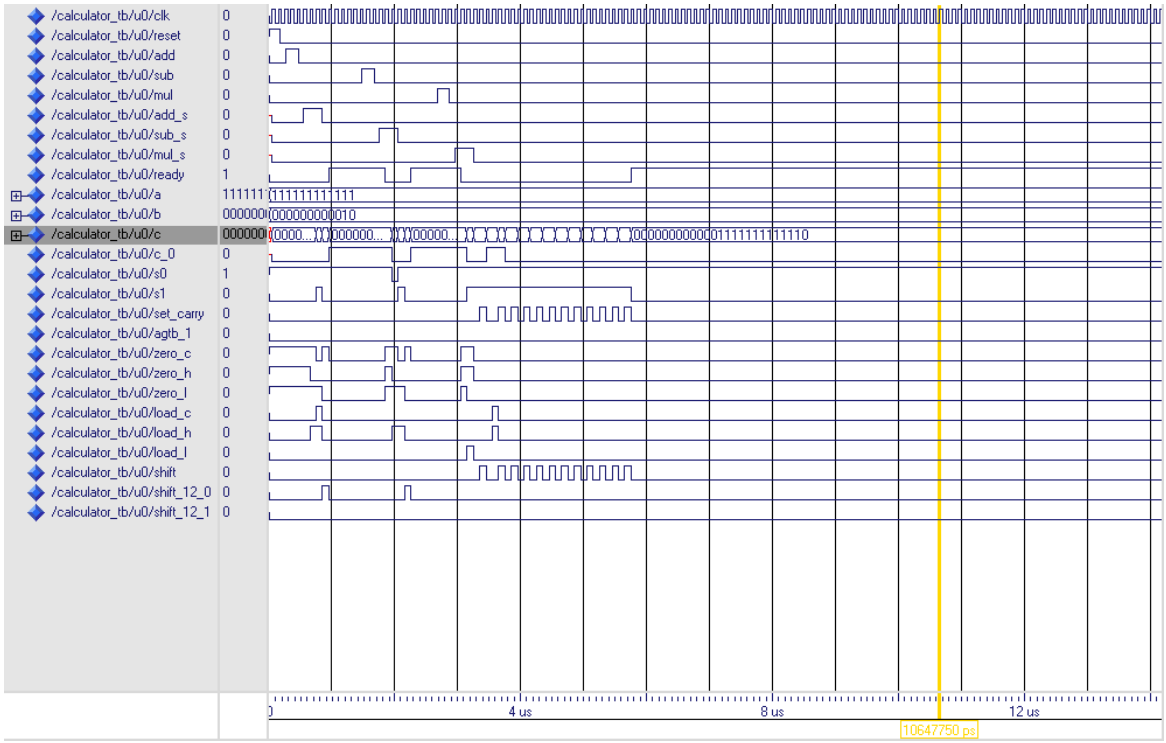
3.7. a=2, b=4095



- Abbildung 16: a=2, b=4095 -

Die erwarteten Ergebnisse der Rechenoperationen (Addition: 4097, Subtraktion: -4093, Multiplikation: 8190) wurden bestätig.

3.8. a=4095, b=2



- Abbildung 17: a=4095, b=2 -

Die erwarteten Ergebnisse der Rechenoperationen mit den Operanten A = 111111111111 und B= 000000000010 (Addition: 000000000001000000000001 (dez 4097), Subtraktion: 000000000000111111111101 (dez 4093), Multiplikation: 000000000001111111111110 (dez 8190)) wurden bestätig.