Aufgabe 1: Programme

a) ram1a.hex

```
; @80 * 4 = @81
;Code 00 ... 0D
@00:2F00;         LOAD 0x00
@01:AF81;         WRITE @ERG
@02:2F04;         LOAD 0x04
@03:AF32;         WRITE @B
@04:8F80;         LOAD @A
@05:9081;         ADD akku @ERG
@06:AF81;         WRITE @ERG
@07:8F32;         LOAD @B
@08:1401;         B--
@09:AF32;         WRITE @B
@0a:6001;         JZ 1 if B == 0
@0b:4F04;         JMP @04
@0c:2FFF;         LOAD 0xFF
@0d:AFFF;         WRITE @ENDSIM
;Vars 32 ... 81
@32:0000;   @B
@80:0004;   @A
@81:0000;   @ERG
;ENDSIM
@ff:0000;   @ENDSIM
```

b) ram1b.hex

```
;@F0 * @F1 = @F2
; 00 ... 15 Programm
@00:2F10;         LOAD 0x10         -|
@01:AFE0;         STORE @LOOPCNT    LOOPCNT wird auf 16 gesetzt
@02:8FE0; LOOPS:  LOAD @LOOPCNT     --
@03:6F14;         JZ @ENDE          Abbruch wenn LOOPCNT = 0 ist.
@04:1401;         SUB 0x01            |
@05:AFE0;         STORE @LOOPCNT    LOOTCNT--;
@06:8FF1;         LOAD @OP2         --
@07:1101;         AND 0x01            |
@08:6F0c;         JZ @SHIFT         Sprung wenn NICHT addiert wird (OPS2 && 0x01 = 0)
@09:8FF2;         LOAD @RES         --
@0a:90F0;         ADD @OP1            |
@0b:AFF2;         STORE @RES        RES += OPS1S
@0c:8FF1; SHIFT:  LOAD @OP2          --
@0d:1800;         SHR                 |
@0e:AFF1;         STORE @OP2        OP2 wird durch einen Shift nach rechts halbiert
@0f:6F14;         JZ @ENDE          Abbruch wenn OPS == 0 ist
@10:8FF0;         LOAD @OP1         --
@11:1900;         SHL                |
@12:AFF0;         STORE @OP1        OP1 wird durch einen Shift nach links verdoppelt
@13:4F02;         JMP @LOOPS        Sprung zum Schleifenstart
@14:2FFF; ENDE:   LOAD 0xFF         Schreiben von 0xFF @0xFF gestopt
@15:AFFF;         STORE @ENDSIM     -|
; e0 Temp Variablen
@e0:0000; @LOOPCNT
; f0 ... f2 Parameter und Ergebnis
@f0:FFFB; @OP1 -5
@f1:FFFC; @OP2 -4
@f2:0000; @RES -5 * -4 = 20:10 = 14 :16
;ENDSIM
@ff:0000; @ENDSIM
```

c) Ram1c.hex

```
;@F0 * @F1 = @F2
; 00 ... 19 Programm
@00:2F04; LOAD 0x04
@01:3500; NEG
@02:FF01; PUSH ACC
@03:2F05; LOAD 0x05
@04:3500; NEG
@05:FF01; PUSH ACC
@06:FF00; PUSH PC
@07:4F40; JMP @ MULTS
@08:EF01; <--Return
@09:AFFE; STORE @PROD
@0a:2FFF; LOAD 0xFF
@0b:AFFF; STORE @ENDSIM
@40:EF01; MULTS:  POP @ACC          -|
@41:AF61;         STORE @RETADR     Retrunadresse vom Stack lesen und speichern
@42:EF01;         POP @ACC          -|
@43:AF62;         STORE @OP1        Multiplikand 1 vom Stack lesen und speichern
@44:EF01;         POP @ACC          -|
@45:AF63;         STORE @OP2        Multiplikand 1 vom Stack lesen und speichern
@46:2F10;         LOAD 0x10         -|
@47:AF60;         STORE @LOOPCNT    LOOPCNT wird auf 16 gesetzt
@48:8F60; LOOPS:  LOAD @LOOPCNT       -|
@49:6F5A;         JZ @ENDE          Abbruch wenn LOOPCNT = 0 ist.
@4a:1401;         SUB 0x01            |
@4b:AF60;         STORE @LOOPCNT    LOOTCNT--;
@4c:8F63;         LOAD @OP2         --
@4d:1101;         AND 0x01            |
@4e:6F52;         JZ @SHIFT         Sprung wenn NICHT addiert wird (OP2 && 0x01 = 0)
@4f:8F64;         LOAD @RES                --
@50:9062;         ADD @OP1            |
@51:AF64;         STORE @RES        RES += OPS1S
@52:8F63; SHIFT:  LOAD @OP2         --
@53:1800;         SHR                 |
@54:AF63;         STORE @OP2        OP2 wird durch einen Shift nach rechts halbiert
@55:6F5A;         JZ @ENDE          Abbruch wenn OPS2 == 0 ist
@56:8F62;         LOAD @OP1         --
@57:1900;         SHL                 |
@58:AF62;         STORE @OP1        OP1 wird durch einen Shift nach links verdoppelt
@59:4F48;         JMP @LOOPS        Sprung zum Schleifenstart
@5a:8F64; ENDE:   LOAD @RES         -|
@5b:FF01;         PUSH ACC          Produkt vom Speicher lesen und im Stack speichern
@5c:8F61;         LOAD @RETADR      --
@5d:FF01;         PUSH ACC            |
@5e:EF00;         POP @PC           Returnaddresse vom Speicher --> Stack --> PC
; 60 ... 64 Variablen
@60:0000; @LOOPCNT
@61:0000; @RETADR
@62:FFFB; @OP1 vom Stack
@63:FFFC; @OP2 vom Stack
@64:0000; @RES
;ENDSIM
@fe:0000; @PROD
@ff:0000; @ENDSIM
```

Aufgabe 2: Erweiterung der CPU

a) ALU-Betriebsart MUL(`cpu_alu.vhd`)
Im ALU wir MUL zusammen mit AND, über `inc_sel = 0` und `path_inc = 0/1` gemultiplxed. Dazu müssen die Betriebsarten AND, OR, NOR angepasst werden.

Betriebsart			Decodierte Steuersignale (binär)					
Bin.	Hex.	Symbol	inc_sel	pre_inv	path_inc	path_crtl	path_sel	post_inv
„0001“	1	AND	0	(0,0)	0	0	0	0
„0010“	2	MUL	0	(0,0)	1	0	0	0
„0110“	6	OR	0	(1,1)	0	0	0	1
„0111“	7	NOR	0	(1,1)	0	0	0	0

```
when ALU_MODE_AND => dec <= ('0',('0','0'),'0','0','0','0');
when ALU_MODE_MUL => dec <= ('0',('0','0'),'1','0','0','0');
when ALU_MODE_OR  => dec <= ('0',('1','1'),'0','0','0','1');
when ALU_MODE_NOR => dec <= ('0',('1','1'),'0','0','0','0');
```

Für die Multiplikation wird eine Methode in VHDL benutzt:

```
function mult(op1, op2 : alu_word_t) return alu_word_t is
            variable o1, o2   : std_logic_vector(15 downto 0);
            variable erg      : std_logic_vector(31 downto 0);
        begin
            o1 := op1(15 downto 0);
            o2 := op2(15 downto 0);
            erg := ZERO16 & ZERO16;
            for i in 0 to 15 loop
                if (o2(0) = '1') then
                    erg := std_logic_vector(signed(erg) + signed(o1));
                end if;
                o2 := "0" & o2(15 downto 1);
                o1 := o1(14 downto 0) & "0";
            end loop;

            if (op1(15) = '1' xor op2(15) = '1') then
                if (erg(15) = erg(16)) then
                    erg(16) := '0';
                else
                    erg(16) := '1';
                end if;
            else
                if (erg(15) = '1') then
                    erg(16) := '1';
                else
                    erg(16) := '0';
                end if;
            end if;
            return erg(16 downto 0);
        end;
```

Welche wie folgt eingebunden wird dazu kommt noch die zusätzliche Variable `tmpm` auf die der Prozess `alu_dec` reagieren muss:

```
signal tmpm, tmp_r                  : alu_word_t;
…
if dec.path_inc = '0' then
      tmpm <= tmp1a and tmp2a;                                  -- "&"
else
      tmpm <= mult(tmp1a, tmp2a);                               -- "*"
end if;

if dec.path_ctrl = '0' then
      tmp2c <= tmpm;
else
      tmp2c <= tmp2b;
end if;
```

b) Erweiterung des Steuerwerks (cpu_ctrl.vhd)
Es soll ein Modus implementiert werden der 16-Bit Direktoperanden einlesen kann, dazu wird eine neue Adressierungsart eingeführt: PMEM. Bei dieser wird der PC als Adresse für die zu ladenden Daten ausgewertet und am Ende des Zyklus 2 weiter gesetzt um die Datenadresse zu überspringen.

```
type operand_result_enum_t is (ACC,PC,INST,AMEM,RMEM,PMEM,SMEM,IREG,NONE);
```

der neue Modus ist wie folgt definiert.

```
when CTRL_MODE16=> dec <= (ACC, PMEM, ACC, ALWAYS, true, DC16);
```

Der Inhalt des Akkumulators wird dem der PC-Direktoperand verknüpft und wieder im Akkumulator gespeichert. Dazu muss zuerst im Prozess ctrl_dec eine Unterscheidung zwischen MODE16 und den anderen Modi:

```
if reg_q.inst(15 downto 12) = CTRL_MODE16 then
      dec.operand <= std_logic_vector(to_unsigned(reg_q.pc,16));
else
      dec.operand <= ZERO16(15 downto 8) & reg_q.inst(7 downto 0);
end if;
```

Dann müssen nur noch die einzelnen Phasen angepasst werden, damit vom PMEM korrekt geladen werden kann:

```
READ_MEM

if dec.src1 = AMEM or dec.src2 = AMEM or dec.src2 = PMEM then

OPERATE

elsif dec.src2 = AMEM or dec.src2 = RMEM or dec.src2 = SMEM or dec.src2 = PMEM
then
```

am Schluss muss noch der PC erhöht werden:

```
OPERATE (ganz am Ende)

if dec.src2 = PMEM then
      reg_d.pc <= reg_q.pc + 1;
end if;
```

In cpu_pack.vhd müssen die beiden neuen Modi noch als Konstanten eingefügt warden:

```
constant ALU_MODE_MUL         : alu_mode_t      := "0010";
…
constant CTRL_MODE16          : ctrl_mode_t     := "0111";
```