1. Behavioural description of a Processor

How to model the behaviour of a 'complex' processor in VHDL? This section shows a way how this could be done. The functional behaviour is not a 'model' of a real processor; no timing aspects are modeled. The idea of this description is based on Peter Ashenden's "VHDL Cookbook" first edition from 1990. We have changed the original processor description with a number of addressing modes, and other instructions. The instruction set is a little bit strange but is choosen for educational purposes only. Furthermore the original bi-directional bus is split into two uni-directional busses only to reduce the complexity for students.
An informal description of the processor is given first. Afterwards a formal description, using VHDL, is given. If the informal and formal descriptions are not in compliance with each other the formal one is correct.

1.1. Informal description of the processor.

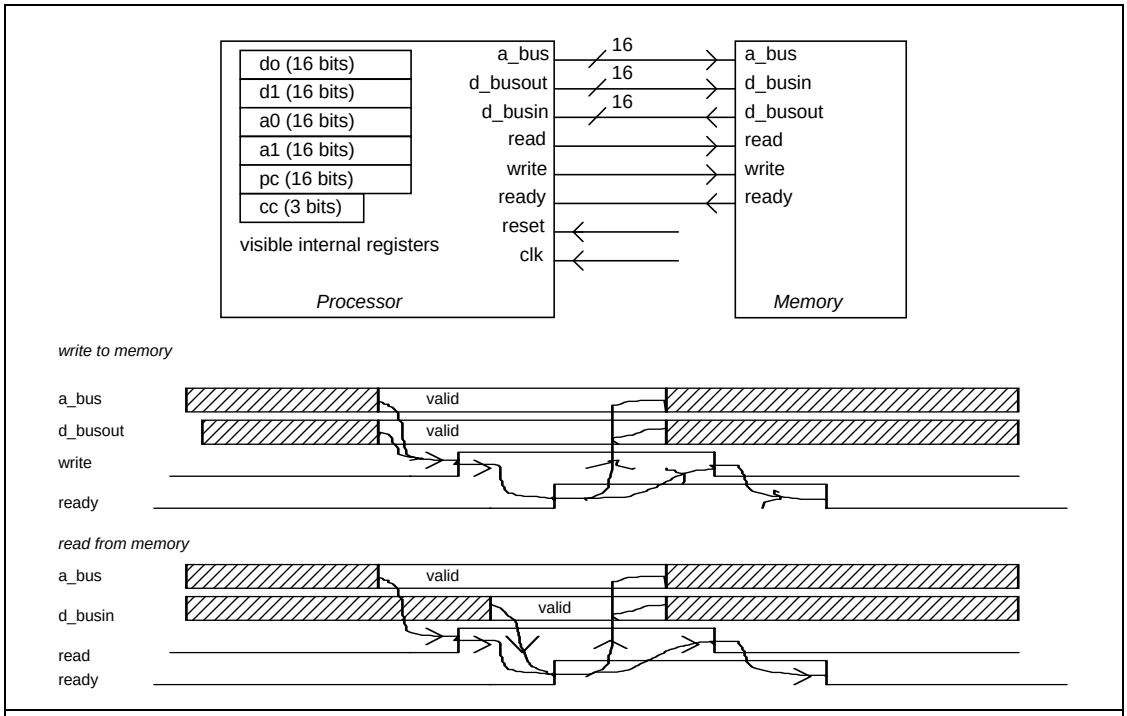

fig. 1.1 processor with memory.

The processor has the following visible internal registers:

- Two data registers, d0 and d1 of 16 bits wide.
- Two address registers, a0 and a1 of 16 bits wide.
- A program counter, pc of 16 bits wide
- A condition code register, cc of 3 bits wide, with:
  - N bit: is true if the result of an operation is negative
  - Z bit: is true if the result of an operation is zero
  - V bit: has two meanings:
    1. is true if an overflow occurs during an arithmetic operation, or
    2. contains the result of a comparison operation

The external address and data bus are 16 bits wide and the communication between processor and memory is based on a handshake protocol.

The instruction format of the processor is one or two words wide. The format of the first word of an instruction is:

```
<      opcode     >
<inst_type> <instr> <source> <destination>
     3        5        4          4          bits wide
```

with:

coding 'inst_type':

000	data movement
001	arithmetic and logical operations
010	shift operations
100	program control
111	miscellaneous

coding of 'source' and 'destination':

0000	none
0001	immediate(#)
0010	D0
0011	D1
0100	A0
0101	A1
0110	(A0) Register a0 contains the address of the data
0111	(A1) Register a1 contains the address of the data

coding of 'instr'

see figure 1.2

Some examples:

sub D0 D1	001_00000_0010_0011
beq (A0)	100_00001_0000_0100
beq #4355	100_00001_0000_0001
	0100_0011_0101_0101
vset	111_00100_0000_0000
inca A0	111_10000_0000_0100
ror D1	010_00101_0000_0011
mov #4231 , (A0)	000_00000_0001_0110
	0100_0010_0011_0001
add D0 , D1	001_00100_0010_0011
add #3124 , D0	001_00100_0001_0010
	0011_0001_0010_0100

Arithmetic and logical operations
restriction: d = D0 of D1, s = D0, D1, (A0), (A1) or immediate.
In case of an overflow the result is a don't care.

instruct	mnemonic		
00000	subt s,d	d := d - s	subtract
00001	abssub s,d	d := | (d-s)	subtract, absolute result
00010	absmsub s,d	d := -|(d-s)	subtract, negative result
00100	add s,d	d := d + s	add
00101	absadd s,d	d := | (d+s)	add, absolute result
00110	absmadd s,d	d := -| (d+s)	add, negative result
01000	maxi s,d	d := Maxi(d,s)	maximum in d
01001	maxa s,d	d := Maxi(|d,|s)	maximum absolute values in d
01010	mini s,d	d := Mini(d,s)	minimum in d
01011	mina s,d	d := Mina(|d,|s)	minimum absolute values in d
01100	absl s,d	d := |s	absolute value of s in d
01101	absmin s,d	d := -|s	negative abs. value of s in d
01110	mul s,d	d := d(7:0)*s(7:0)	two bytes are multiplied
01111	absmul s,d	d := |(d(7:0)*s(7:0))	absolute result of product
10000	kl s,d	d < s	V bit of condition code reg. is affected
10001	klg s,d	d <= s	V bit of condition code reg. is affected
10010	kla s,d	(|d) < (|s)	V bit of condition code reg. is affected
10011	klga s,d	(|d) <= (|s)	V bit of condition code reg. is affected
10100	comp s,d	d = s	V bit of condition code reg. is affected

shift operations
restriction: destination = D0, D1, source = none

instruct	mnemonic		
00000	asl d	d <- d(14:0) & 0	arithmetic shift left
00001	asr d	d <- d(15) & d(15:1)	arithmetic shift right
00010	lsl d	d <- d(14:0) & 0	logical shift left
00011	lsr d	d <- 0 & d(15:1)	logical shift right
00100	rol d	d <- d(14:0) & d(15)	rotate left
00101	ror d	d <- d(0) & d(15:1	rotate right

data movement
restriction: d = D0, D1, A0, A1, (A0), (A1) or Immediate s = D0, D1, A0, A1, (A0), (A1) or Immediate, AND s and d not both immediate.

instruct	mnemonic	
00000	mov s,d	move data from s to d

condition code register is not affected

program control
restriction: destination = immediate, (A0), (A1), source = none
If the condition is true the new program counter is:
pc <- pc + displacement,
condition code register is not affected

instruct	mnemonic	
00000	bra d	branch always
00001	beq d	branch if Z=1

00010	bne d		branch if Z=0
00011	bvs d		branch if V=1
00100	bvc d		branch if V=0
00101	bpl d		branch if N=0
00110	bmi d		branch if N=1

<u>miscellaneous</u>
setting and resetting of condition code register bits:
restriction: destination = none, source = none
not used condition code bits are not affected

instruct	mnemonic		
00000	nset		set N bit
00001	nclr		clear N bit
00010	zset		set Z bit
00011	zclr		clear Z bit
00100	vset		set V bit
00101	vclr		clear V bit

increment/decrement the registers A0 en A1.
restriction: destination = A0, A1, source = none

instruct	mnemonic		
10000	inca d	d:=d+1	increment d
10001	deca d	d:=d-1	decrement d

fig. 1.2 instruction set of the processor.

1.2. Memory.

```
-- The entity memory contains conversion functions used in the automatically
-- generated architecture for the memeory.
-- Declaring it locally prevents from using it elsewehere in a design unit.
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;
USE work.processor_types.ALL;
ENTITY memory IS
  GENERIC (tpd  : time := 1 ns);
  PORT(d_busout : OUT bit16;
       d_busin  : IN  bit16;
       a_bus    : IN  bit16;
       write    : IN  std_ulogic;
       read     : IN  std_ulogic;
       ready    : OUT std_ulogic);

  PROCEDURE int2bitv(int : IN integer; bitv: OUT std_ulogic_vector) IS
  BEGIN
    bitv:=std_ulogic_vector(to_signed(int,bitv'LENGTH));
  END int2bitv;

  FUNCTION bitv2int(bitv : IN std_ulogic_vector) RETURN integer IS
  BEGIN
    RETURN to_integer(signed(bitv));
  END bitv2int;

  FUNCTION bitv2nat (bitv : IN std_ulogic_vector) RETURN natural IS
  BEGIN
    RETURN to_integer(unsigned(bitv));
  END bitv2nat;
END memory;
-------------------------------------------------------
ARCHITECTURE behaviour OF memory IS
BEGIN
  PROCESS
    CONSTANT low_address:natural:=0;
    CONSTANT high_address:natural:=300;  -- upper limit of the memory
                                         -- INCREASE this number IF the program
                                         -- needs more memory. Don't FORget
                                         -- that the addresses used to write
                                         -- to and read from should be available.
    TYPE memory_array IS
      ARRAY (natural RANGE low_address TO high_address) OF integer;
    VARIABLE mem:memory_array:=
           (18,                    --        mov #6,d0        0000 0000 0001 0010
            6,                     --                         0000 0000 0000 0110
            20,                    --        mov #62,a0       0000 0000 0001 0100
            62,                    --                         0000 0000 0011 1110
            21,                    --        mov #63,a1       0000 0000 0001 0101
            63,                    --                         0000 0000 0011 1111
            19,                    --        mov #1,d1        0000 0000 0001 0011
            1,                     --                         0000 0000 0000 0001
            54,                    --        mov d1,(a0)      0000 0000 0011 0110
            55,                    --        mov d1,(a1)      0000 0000 0011 0111
            13347,                 -- lbl:   comp d0,d1       0011 0100 0010 0011
            -31999,                --        bvs einde:       1000 0011 0000 0001
            9,                     --                         0000 0000 0000 1001
            9235,                  --        add #1,d1        0010 0100 0001 0011
            1,                     --                         0000 0000 0000 0001
            55,                    --        mov d1,(a1)      0000 0000 0011 0111
            11875,                 --        mul (a0),d1      0010 1110 0110 0011
            -3836,                 --        deca a0          1111 0001 0000 0100
            54,                    --        mov d1,(a0)      0000 0000 0011 0110
            115,                   --        mov (a1),d1      0000 0000 0111 0011
            -32767,                --        bra lbl:         1000 0000 0000 0001
            -12,                   --                         1111 1111 1111 0100
            -32767,                -- einde: bra einde:       1000 0000 0000 0001
            -2,                    --                         1111 1111 1111 1110
          OTHERS => 0
         );
    VARIABLE address:natural;
    VARIABLE data_out:bit16;
    CONSTANT unknown : bit16 := (OTHERS=>'-');
  BEGIN
    ready <= '0' AFTER tpd;
    --
    -- WAIT FOR a command
```

```
    --
    WAIT UNTIL (read='1') OR (write='1');
    address:=bitv2nat(a_bus);
    ASSERT (address>=low_address) and (address<=high_address)
      REPORT "out of memory range" SEVERITY warning;
    IF write='1'
      THEN
        mem(address):=bitv2int(d_busin);
        ready<='1' AFTER tpd;
        WAIT UNTIL write='0';                -- WAIT UNTIL END of write cycle
      ELSE -- read ='1';
        int2bitv(mem(address),data_out);
        d_busout <= data_out;
        ready<='1' AFTER tpd;
        WAIT UNTIL read='0';
        d_busout <= unknown;
    END IF;
  END PROCESS;
END behaviour;
```

fig. 1.3 behavioural description of a memory.

Figure 1.3 shows a behavioural description of the memory. The program in the memory determines the factorial of the numbers 1 to 6 and the results are placed in the memory adresses 62 (1 !) downto 57 (6 !).
An assembler is developed for this processor. At the time the assembler was developed the package numeric_std was not yet supported. This is also the reason why in the entity declaration ‘conversion’ functions are declared. (Funny place isn’t it!)

1.3. Formal description of the processor.

A package processor_types contains a number of often used subtypes, all supported instructions, the coding of source and destination, and shift operators.

1.3.1. Processor types

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE work.processor_types.ALL;
ENTITY processor IS
  PORT (d_busout: OUT bit16;
        d_busin : IN  bit16;
        a_bus   : OUT bit16;
        write   : OUT std_ulogic;
        read    : OUT std_ulogic;
        ready   : IN  std_ulogic;
        reset   : IN  std_ulogic;
        clk     : IN  std_ulogic);
END processor;

LIBRARY ieee;
USE ieee.numeric_std.ALL;
ARCHITECTURE behaviour OF processor IS
BEGIN
  PROCESS
    VARIABLE pc : natural;
    VARIABLE a0 : bit16;
    VARIABLE a1 : bit16;
    VARIABLE d0 : bit16;
    VARIABLE d1 : bit16;
    VARIABLE cc : bit3;
      ALIAS cc_n  : std_ulogic IS cc(2);
      ALIAS cc_z  : std_ulogic IS cc(1);
      ALIAS cc_v  : std_ulogic IS cc(0);
    VARIABLE data : bit16;
    VARIABLE current_instr:bit16;
      ALIAS op  : bit8 IS current_instr(15 DOWNTO 8);
      ALIAS src : bit4 IS current_instr( 7 DOWNTO 4);
      ALIAS dst : bit4 IS current_instr( 3 DOWNTO 0);
    VARIABLE error_src_dst  : boolean;    -- error in src or dst in insruction
    VARIABLE rs,rd          : bit16;      -- temporary variables
    VARIABLE rs_int, rd_int : integer;    -- integer representation of rs, rd.
    VARIABLE rs_low, rd_low : integer;    --    "     "  positions 7..0 of rs and rd.
    VARIABLE rc             : std_ulogic;  --    "
    VARIABLE displacement  : bit16;
    VARIABLE jump          : boolean;     -- used in branch instructions
    VARIABLE tmp           : bit16;
    CONSTANT one           : bit16 := (0 => '1', OTHERS => '0');
    CONSTANT dontcare      : bit16 := (OTHERS => '-');

    PROCEDURE memory_read (addr   : IN natural;
                           result : OUT bit16) IS
    -- Used 'global' signals are:
    --   clk, reset, ready, read, a_bus, d_busin
    -- read data from addr in memory
    BEGIN
      -- put address on output
      a_bus <= std_ulogic_vector(to_unsigned(addr,16));
      WAIT UNTIL clk='1';
      IF reset='1' THEN
        RETURN;
      END IF;

      LOOP -- ready must be low (handshake)
        IF reset='1' THEN
          RETURN;
        END IF;
        EXIT WHEN ready='0';
        WAIT UNTIL clk='1';
      END LOOP;

      read <= '1';
      WAIT UNTIL clk='1';
      IF reset='1' THEN
        RETURN;
      END IF;

      LOOP
        WAIT UNTIL clk='1';
        IF reset='1' THEN
          RETURN;
        END IF;
```

```
      IF ready='1' THEN
        result:=d_busin;
        EXIT;
      END IF;
    END LOOP;
    WAIT UNTIL clk='1';
    IF reset='1' THEN
      RETURN;
    END IF;

    read <= '0';
    a_bus <= dontcare;
  END memory_read;

  PROCEDURE memory_write(addr : IN natural;
                         data : IN bit16) IS
  -- Used 'global' signals are:
  --   clk, reset, ready, write, a_bus, d_busout
  -- write data to addr in memory
    VARIABLE add : bit16;
  BEGIN
    -- put address on output
    a_bus <= std_ulogic_vector(to_unsigned(addr,16));
    WAIT UNTIL clk='1';
    IF reset='1' THEN
      RETURN;
    END IF;

    LOOP -- ready must be low (handshake)
      IF reset='1' THEN
        RETURN;
      END IF;
      EXIT WHEN ready='0';
      WAIT UNTIL clk='1';
    END LOOP;

    d_busout <= data;
    WAIT UNTIL clk='1';
    IF reset='1' THEN
      RETURN;
    END IF;
    write <= '1';

    LOOP
      WAIT UNTIL clk='1';
      IF reset='1' THEN
        RETURN;
      END IF;
       EXIT WHEN ready='1';
    END LOOP;
    WAIT UNTIL clk='1';
    IF reset='1' THEN
      RETURN;
    END IF;
    --
    write <= '0';
    d_busout <= dontcare;
    a_bus <= dontcare;
  END memory_write;

  PROCEDURE read_data(s_d    : IN bit4;
                      d0, d1 : IN bit16;
                      a0, a1 : IN bit16;
                      pc     : inout natural;
                      data   : OUT bit16) IS
  -- read data from d0,d1,a0,a1,(a0),(a1),imm
    VARIABLE tmp : bit16;
  BEGIN
    CASE s_d IS
      WHEN rd0    => data := d0;
      WHEN rd1    => data := d1;
      WHEN ra0    => data := a0;
      WHEN ra1    => data := a1;
      WHEN a0_ind => memory_read(to_integer(unsigned(a0)),data);
      WHEN a1_ind => memory_read(to_integer(unsigned(a1)),data);
      WHEN imm    => memory_read(pc,data);
                     pc := pc + 1;
      WHEN OTHERS => ASSERT false REPORT "illegal src/dst while reading data"
                     SEVERITY warning;
    END CASE;
  END read_data;
```

```
  PROCEDURE write_data(s_d    : IN bit4;
                       d0, d1 : INOUT bit16;
                       a0, a1 : INOUT bit16;
                       pc     : INOUT natural;
                       data   : IN bit16) IS
  -- write data to d0,d1,a0,a1,(a0),(a1),imm
    VARIABLE tmp:bit16;
    VARIABLE addr: bit16;
  BEGIN
    CASE s_d IS
      WHEN rd0    => d0:=data;
      WHEN rd1    => d1:=data;
      WHEN ra0    => a0 := data;
      WHEN ra1    => a1 := data;
      WHEN a0_ind => memory_write(to_integer(unsigned(a0)),data);
      WHEN a1_ind => memory_write(to_integer(unsigned(a1)),data);
      WHEN imm    => memory_read(pc,addr);
                     pc := pc + 1;
                     memory_write(to_integer(unsigned(addr)),data);
      WHEN OTHERS => ASSERT false REPORT "illegal src or dst while writing data"
                     SEVERITY warning;
    END CASE;
  END write_data;

BEGIN
  --
  -- check FOR reset active
  --
  IF reset='1' THEN
    read <= '0';
    write <= '0';
    pc := 0;
    cc := "000"; -- clear condition code register
    LOOP         -- synchrone reset
      WAIT UNTIL clk='1';
      EXIT WHEN reset='0';
    END LOOP;
  END IF;
  --
  -- fetch next instruction
  --
  memory_read(pc,current_instr);
  IF reset /= '1' THEN
    pc:=pc+1;
    --
    -- decode & execute
    --
    CASE op IS

     WHEN mov =>
       error_src_dst:= NOT member(src,rd0&rd1&ra0&ra1&a0_ind&a1_ind&imm) or
                       NOT member(dst,rd0&rd1&ra0&ra1&a0_ind&a1_ind&imm) or
                       ((src=imm) and (dst=imm));
       ASSERT NOT error_src_dst REPORT "illegal inst. mov"
         SEVERITY warning;
       read_data(src,d0,d1,a0,a1,pc,rs);
       write_data(dst,d0,d1,a0,a1,pc,rs);
       cc := cc;  -- condition code register is unchanged.

     WHEN subt|abssub|absmsub|add|absadd|absmadd|maxi|maxa|mini|mina|
          absl|absmin|mul|absmul =>
       error_src_dst:= NOT member(src,rd0&rd1&a0_ind&a1_ind&imm) or
                       NOT member(dst,rd0&rd1);
       ASSERT NOT error_src_dst REPORT "illegal inst. ARITHMETIC" SEVERITY warning;
       read_data(src,d0,d1,a0,a1,pc,rs); rs_int:=to_integer(signed(rs));
                                         rs_low:=to_integer(signed(rs(7 DOWNTO 0)));
       read_data(dst,d0,d1,a0,a1,pc,rd); rd_int:=to_integer(signed(rd));
                                         rd_low:=to_integer(signed(rd(7 DOWNTO 0)));
       CASE op IS
         WHEN subt    => rd_int :=       rd_int - rs_int;
         WHEN abssub  => rd_int :=  abs( rd_int - rs_int );
         WHEN absmsub => rd_int := -abs( rd_int - rs_int );
         WHEN add     => rd_int :=       rd_int + rs_int;
         WHEN absadd  => rd_int :=  abs( rd_int + rs_int );
         WHEN absmadd => rd_int := -abs( rd_int + rs_int );

         WHEN maxi    => IF      rs_int > rd_int
                           THEN rd_int:=rs_int;
                         END IF;
         WHEN maxa    => IF abs(rs_int) > abs(rd_int)
```

```
                        THEN rd_int:=abs(rs_int);
                        ELSE rd_int:=abs(rd_int);
                      END IF;
      WHEN mini    => IF      rs_int < rd_int
                        THEN rd_int:=rs_int;
                      END IF;
      WHEN mina    => IF abs(rs_int) < abs(rd_int)
                        THEN rd_int:=abs(rs_int);
                        ELSE rd_int:=abs(rd_int);
                      END IF;
      WHEN absl    => rd_int := abs(rs_int);
      WHEN absmin  => rd_int := -abs(rs_int);
      WHEN mul     => rd_int :=      rd_low * rs_low;
      WHEN absmul  => rd_int := abs (rd_low * rs_low);
      WHEN OTHERS  => NULL;
    END CASE;
    set_cc_rd(rd_int,cc,rd);
    write_data(dst,d0,d1,a0,a1,pc,rd);

  WHEN kl|klg|kla|klga|comp =>
    error_src_dst:= NOT member(src,rd0&rd1&a0_ind&a1_ind&imm) or
                    NOT member(dst,rd0&rd1);
    ASSERT NOT error_src_dst REPORT "illegal inst. COMPARE" SEVERITY warning;
    read_data(src,d0,d1,a0,a1,pc,rs); rs_int:=to_integer(signed(rs));
    read_data(dst,d0,d1,a0,a1,pc,rd); rd_int:=to_integer(signed(rd));
    CASE op IS
      WHEN kl      => cc_v := bool2std(     rd_int <  rs_int);
      WHEN klg     => cc_v := bool2std(     rd_int <= rs_int );
      WHEN kla     => cc_v := bool2std(abs(rd_int) <  abs(rs_int));
      WHEN klga    => cc_v := bool2std(abs(rd_int) <= abs(rs_int));
      WHEN comp    => cc_v := bool2std( rd = rs);
      WHEN OTHERS  => NULL;
    END CASE;
    cc_n := '-'; cc_z := '-';

  WHEN asl|asr|lsl|lsr|rol_87|ror_87 =>
    error_src_dst:= NOT member(src,none) or
                    NOT member(dst,rd0&rd1);
    ASSERT NOT error_src_dst REPORT "illegal inst. SHIFT" SEVERITY warning;
    read_data(dst,d0,d1,a0,a1,pc,rd);
    CASE op IS
      WHEN asl => rd:=shift(rd,left,arithmetic);
      WHEN asr => rd:=shift(rd,right,arithmetic);
      WHEN lsl => rd:=shift(rd,left,logical);
      WHEN lsr => rd:=shift(rd,right,logical);
      WHEN rol_87 => rd:=rotate(rd,left);
      WHEN ror_87 => rd:=rotate(rd,right);
      WHEN OTHERS  => NULL;
    END CASE;
    cc := "---";
    write_data(dst,d0,d1,a0,a1,pc,rd);

  WHEN bra|beq|bne|bvs|bvc|bpl|bmi =>
    error_src_dst:= NOT member(src,none) or
                    NOT member(dst,a0_ind&a1_ind&imm);
    ASSERT NOT error_src_dst REPORT "illegal inst. BRANCH" SEVERITY warning;
    CASE op IS
      WHEN bra => jump := TRUE;
      WHEN beq => jump := cc_z='1';
      WHEN bne => jump := cc_z='0';
      WHEN bvs => jump := cc_v='1';
      WHEN bvc => jump := cc_v='0';
      WHEN bpl => jump := cc_n='0';
      WHEN bmi => jump := cc_n='1';
      WHEN OTHERS  => NULL;
    END CASE;
    -- condition code register has NOT changed
    IF jump
      THEN
        CASE dst IS
          WHEN imm    => memory_read(pc,displacement);
                         pc := pc +1;
          WHEN a0_ind => memory_read(to_integer(unsigned(a0)),displacement);
          WHEN a1_ind => memory_read(to_integer(unsigned(a1)),displacement);
          WHEN OTHERS => ASSERT false
                         REPORT "illegal destination in BRANCH instruction"
                         SEVERITY warning;
        END CASE;
        pc := pc + to_integer(signed(displacement));
      ELSE IF dst=imm THEN pc := pc + 1; END IF;  -- skip contents next address
    END IF;
```

```
      WHEN nset|nclr|zset|zclr|vset|vclr =>
        error_src_dst:= NOT member(src,none) or
                        NOT member(dst,none);
        ASSERT NOT error_src_dst
        REPORT "illegal instruction SET or CLR of CC" SEVERITY warning;
        CASE op IS
          WHEN nset    => cc_n:='1';
          WHEN nclr    => cc_n:='0';
          WHEN zset    => cc_z:='1';
          WHEN zclr    => cc_z:='0';
          WHEN vset    => cc_v:='1';
          WHEN vclr    => cc_v:='0';
          WHEN OTHERS  => NULL;
        END CASE;
        -- other condition code bits will be NOT changed

      WHEN inca|deca =>
        error_src_dst:= NOT member(src,none) or
                        NOT member(dst,ra0&ra1);
        ASSERT NOT error_src_dst REPORT "illegal inst. INCA, DECA" SEVERITY warning;
        CASE op IS
          WHEN inca =>
            CASE dst IS
              WHEN ra0 => IF a0 = (a0'RANGE => '1') -- upper bound?
                            THEN a0 := (OTHERS => '-');
                            ELSE a0 := std_ulogic_vector(unsigned(a0)+1);
                          END IF;
              WHEN ra1 => IF a1 = (a1'RANGE => '1') -- upper bound?
                            THEN a1 := (OTHERS => '-');
                            ELSE a1 := std_ulogic_vector(unsigned(a1)+1);
                          END IF;
              WHEN OTHERS  => NULL;
            END CASE;
          WHEN deca =>
            CASE dst IS
              WHEN ra0 => IF a0 = (a0'RANGE => '0') -- lower bound?
                            THEN a0 := (OTHERS => '-');
                            ELSE a0 := std_ulogic_vector(unsigned(a0)-1);
                          END IF;
              WHEN ra1 => IF a1 = (a1'RANGE => '0') -- lower bound?
                            THEN a1 := (OTHERS => '-');
                            ELSE a1 := std_ulogic_vector(unsigned(a1)-1);
                          END IF;
              WHEN OTHERS  => NULL;
            END CASE;
          WHEN OTHERS => NULL;
        END CASE;
        cc := "---";

      WHEN OTHERS => ASSERT false REPORT "illegal instruction" SEVERITY warning;

     END CASE;
    END IF;
  END PROCESS;
END behaviour;
```

fig. 1.4 formal description of the behaviour of the processor.

1.3.2. Processor description

********** PROCESSOR **********

**move processor types

1.4. The processor and the memory

In figure 1.1 a scheme is given of the connection between the processor and the memory. In figure 1.5 the VHDL description is shown. Figure 1.6 shows the waveform of a simulation. In this waveform the 16 bits values of the data and address busses are given in an 'integer' representation. Notice that the values are what may be expected from the program in the memory (figure 1.3).

```
ENTITY dut IS
END dut;

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE work.processor_types.ALL;
ARCHITECTURE memory_processor OF dut IS
  component memory
    GENERIC (tpd  : time := 1 ns);
    PORT(d_busout : OUT bit16;
         d_busin  : IN  bit16;
         a_bus    : IN  bit16;
         write    : IN  std_ulogic;
         read     : IN  std_ulogic;
         ready    : OUT std_ulogic);
  END component;
  component processor
    PORT (d_busout: OUT bit16;
          d_busin : IN  bit16;
          a_bus   : OUT bit16;
          write   : OUT std_ulogic;
          read    : OUT std_ulogic;
          ready   : IN  std_ulogic;
          reset   : IN  std_ulogic;
          clk     : IN  std_ulogic);
  END component;
  SIGNAL data_from_cpu,data_to_cpu,addr : bit16;
  SIGNAL read,write,ready               : std_ulogic;
  SIGNAL reset                          : std_ulogic := '1';
  SIGNAL clk                            : std_ulogic := '0';
BEGIN
  cpu:processor
      PORT MAP(data_from_cpu,data_to_cpu,addr,write,read,ready,reset,clk);
  mem:memory
      GENERIC MAP (1 ns)
      PORT MAP (data_to_cpu,data_from_cpu,addr,write,read,ready);
  reset <= '1', '0' AFTER 100 ns;
  clk   <= NOT clk AFTER 10 ns;
END memory_processor;
-------------------------------------------------------
CONFIGURATION test_of_mem_proc OF dut IS
  FOR memory_processor
    FOR cpu:processor USE ENTITY work.processor (behaviour); END FOR;
    FOR mem:memory USE ENTITY work.memory (behaviour); END FOR;
  END FOR;
END test_of_mem_proc;
```

fig. 1.5 structural description of processsor and memory.

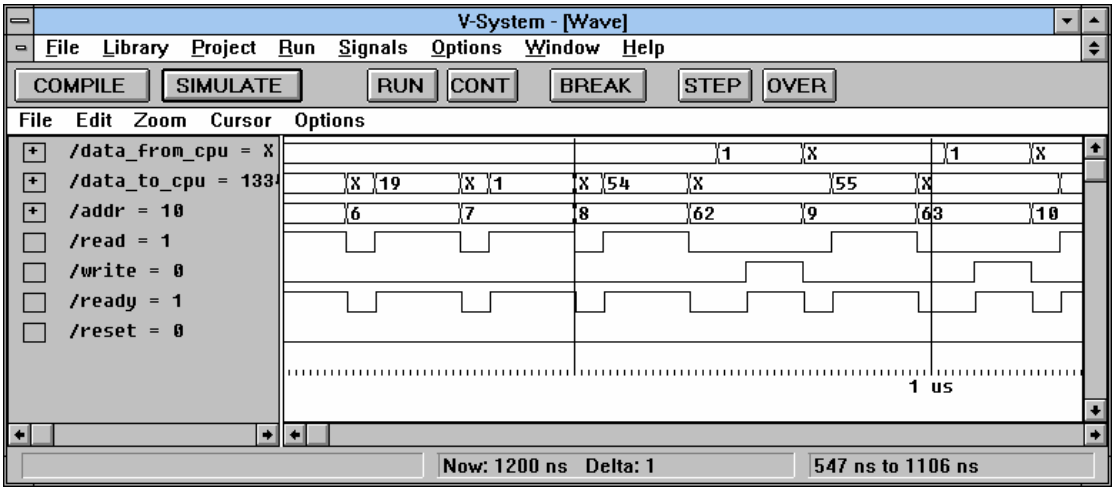

fig. 1.6 part of the simulation result.