A minimal 8Bit CPU in a 32 Macrocell CLPD.

Tim Böscke, t.boescke@tuhh.de

February 17, 2002

This documents describes a successful attempt to fit a simple VHDL - CPU into a 32 macrocell CPLD. The CPU has been simulated and has so far been synthesized for the Lattice M4A 32/32 (ispDesignExpert Starter) and the Xilinx 9536 (WebPack). However, all macrocell counts in this document refer to the M4A 32/32.

The CPU entity description (basically an interface to asynchronous sram):

```
entity CPU8BIT2 is
  port (
    data:   inout std_logic_vector(7 downto 0);
    adress: out   std_logic_vector(5 downto 0);
    oe:     out   std_logic;
    we:     out   std_logic;
    rst:    in    std_logic;
    clk:    in    std_logic);
end;
```

1 Programming model

1.1 Registers and memory

The CPU is accumulator based and supports a bare minimum of registers. The Accu has a width of eight Bit and is complemented by a carry flag. The PC has a width of six Bit which allows to adress 64 eight Bit words of memory. The memory is shared between program code and data.

1.2 Instruction set

Each instruction is one word wide. A single instruction format is used. It is encoded with a two bit opcode and a six bix adress/immediate field.

Mnemonic	Opcode	Description
NOR	00AAAAAA	Accu = Accu NOR mem[AAAAAA]
ADD	01AAAAAA	Accu = Accu + mem[AAAAAA], update carry
STA	10AAAAAA	mem[AAAAAA] = Accu
JCC	11DDDDDD	Set PC to DDDDDD when carry = 0, clear carry

Table 1: Instruction set listing.

The four encodable instructions are listed in table 1. The choice of instructions was inspired by another minimal CPU design, the MPROZ[1]. However instead of being used in a memory-memory architecture, like the MPROZ, the instructions are used in the context of an accu based architecture. This made the

[1] ftp://mistress.informatik.unibw-muenchen.de/pub/mproz/

additional *STA* instruction mandatory. The benefits are a bigger code density (Instructions are just one word instead of two.) and an even simpler cpu architecture.

One interesting aspect is the branch instruction *JCC*. Branches are always conditional. However the *JCC* instruction clears the carry, so that succeeding branches are always taken. This allows efficient unconditional, or two way branches.

Below is one of the programs tested on the CPU. It calculates the greatest common divisor of two numbers using Dijkstras algorithm.

Listing 1: GCD example

```
    start :
          NOR   allone   ;Akku = 0
10        NOR   b
          ADD   one      ;Akku = - b

          ADD   a        ;Akku = a - b
                         ;Carry set when akku >= 0
15        JCC   neg

          STA   a

          ADD   allone
20        JCC   end      ;A=0 ? -> end, result in b

          JCC   start
    neg:
          NOR   zero
25        ADD   one      ;Akku = -Akku

          STA   b
          JCC   start    ;Carry was not altered
    end:
30        JCC   end
```

2 Architecture

2.1 Datapath

One design goal was to minimize the amount of macrocells used purely for combinational logic, to maximize the amount of usable registers. Due to this, structures like multiplexers between registers and the adress/data output had to be avoided at all costs. One consequence was to divide the datapath into one path for the adress and one for the data.

In contrast to other small cpus the adress generation is not done with the main ALU, therefore a distinct incrementer was required for the PC. Fortunately the PC incrementer does still fit into the macrocells holding the PC register, allowing the full 'adress - datapath' to fit into 12 macrocells.

The 'data - datapath' occupies 14 Macrocells. (eight for the akku, one for the carry, five combinational macrocells for carry propagation).

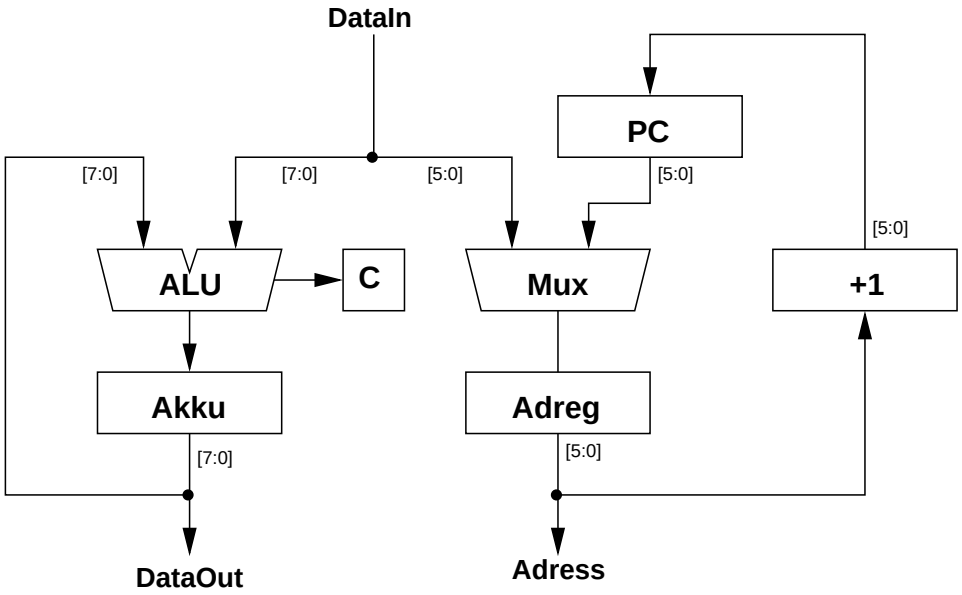

Figure 1: Datapath of the CPU.

2.2 Control

The datapath is controlled by a simple state machine with 5 states. The state encoding was carefully chosen, to minimize the required amount of macrocells to store and decode the states. Two additional macrocells are used to generate the OE and WE signals. The total count of macrocells used for the control amounts to 5.

The state encoding for the state machine is listed in table 2.

Almost all instructions are executed in two clock cycles. The only exception is a taken branch, which is being executed in a single cycle.

State	Function	Operations	Next
000 S0	Fetch instruction /Operand adress	pc ⇐ adreg + 1, adreg = data oe ⇐ 0, data ⇐ Z	S0 w. opcode = 11, c = 0 S1 w. opcode = 10 S2 w. opcode = 01 S3 w. opcode = 00 S5 w. opcode = 11, c = 1
001 S1	Write akku to memory	we ⇐ 0, data ⇐ akku adreg ⇐ pc	S0
010 S2	Read operand, ADD	oe ⇐ 0, data ⇐ z, adreg ⇐ pc akku ⇐ akku + data , update carry	S0
011 S3	Read operand, NOR	oe ⇐ 0, data ⇐ z, adreg ⇐ pc akku ⇐ akku NOR data	S0
101 S5	Clear carry, Read PC	carry ⇐ 0, adreg ⇐ pc	S0

Table 2: The state machine.

3 Sources

A ZIP-Archive containing the VHDL-Sources of the CPU and the testbench can be downloaded here: http://www.tuhh.de/ setb0209/cpu/.

Listing 2: CPU source

```
--
-- Minimal 8 Bit CPU
--
-- rev 15102001
--
-- 01-02/2001 Tim Boescke
-- 10  /2001 slight changes for proper simulation.
--
-- t.boescke@tuhh.de
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity CPU8BIT2 is
        port (  data:   inout   std_logic_vector (7 downto 0);
                adress: out     std_logic_vector (5 downto 0);
                oe:     out     std_logic;
                we:     out     std_logic;
                rst:    in      std_logic;
                clk:    in      std_logic );
end;

architecture CPU_ARCH of CPU8BIT2 is
        signal  akku:   std_logic_vector (8 downto 0);  -- akku(8) is carry !
        signal  adreg:  std_logic_vector (5 downto 0);
        signal  pc:     std_logic_vector (5 downto 0);
        signal  states: std_logic_vector (2 downto 0);
begin
        process(clk,rst)
        begin
           if (rst = '0') then
                adreg   <= (others => '0');    -- start execution at memory location 0
                states  <= "000";
                akku <= (others => '0');
                pc   <= (others => '0');
           elsif rising_edge(clk) then

                -- PC / Adress path
                if (states = "000") then
                        pc      <= adreg + 1;
                        adreg   <= data(5 downto 0);
                else
                        adreg   <= pc;
                end if;

                -- ALU / Data Path
                case states is
                        when "010" => akku <= ("0" & akku(7 downto 0)) + ("0" & data); -- add
                        when "011" => akku(7 downto 0) <= akku(7 downto 0) nor data; -- nor
                        when "101" => akku(8) <= '0'; -- branch not taken, clear carry
                        when others => null; -- instr. fetch, jcc taken (000), sta (001)
                end case;

                -- State machine
                if (states /= "000") then states <= "000";                      -- fetch next opcode
                elsif (data(7 downto 6) = "11" and akku(8)='1') then states <= "101";-- branch n. taken
                        else  states <= "0" & not data(7 downto 6);             -- execute instruction
                end if;
           end if;
        end process;

        -- output
        adress  <= adreg;
        data    <= "ZZZZZZZZ" when states /= "001" else akku(7 downto 0);
        oe <= '1' when (clk='1' or states = "001" or rst='0' or states = "101") else '0';
                                        -- no memory access during reset and
        we <= '1' when (clk='1' or states /= "001" or rst='0') else '0';
                                        -- state "101" (branch not taken)

end CPU_ARCH;
```