MCPU - A Minimal 8Bit CPU in a 32 Macrocell CPLD.

Tim Böscke, cpldcpu@opencores.org

02/2001 - Revised 10/2004

This document describes a successful attempt to fit a simple VHDL - CPU into a 32 macrocell CPLD. The CPU has been simulated and synthesized for the Lattice ispMach M4A-32 (ispLever) and the Xilinx 9536 (WebPack). Interestingly, Quartus II was not able to fit the design for the 32 macrocell variants of the Altera MAX3000/MAX7000 series.

All macrocell counts in this document refer to the M4A-32.

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

```vhdl
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 accumulator has a width of eight bits and is complemented by a carry flag. The program counter (PC) has a width of six bits which allows addressing of 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 bit address/immediate field.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Opcode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOR</td>
<td>00AAAAAA</td>
<td>Accu = Accu NOR mem[AAAAAA]</td>
</tr>
<tr>
<td>ADD</td>
<td>01AAAAAA</td>
<td>Accu = Accu + mem[AAAAAA], update carry</td>
</tr>
<tr>
<td>STA</td>
<td>10AAAAAA</td>
<td>mem[AAAAAA] = Accu</td>
</tr>
<tr>
<td>JCC</td>
<td>11DDDDDD</td>
<td>Set PC to DDDDD when carry = 0, clear carry</td>
</tr>
</tbody>
</table>

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,

\(^1\)ftp://mistress.informatik.unibw-muenchen.de/pub/mproz/
like the MPROZ, the instructions are used in the context of an accu based architecture. This made the additional STA instruction mandatory. The benefits are a better 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.

<table>
<thead>
<tr>
<th>Macro</th>
<th>Assembler Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLR</td>
<td>NOR allone</td>
<td>Clear Accu (allone contains 0xFF)</td>
</tr>
<tr>
<td>LDA mem</td>
<td>NOR allone,ADD mem</td>
<td>Load mem into Accu</td>
</tr>
<tr>
<td>NOT</td>
<td>NOR zero</td>
<td>Invert content of Accu (zero contains 0x00)</td>
</tr>
<tr>
<td>JMP dst</td>
<td>JCC dst, JCC dst</td>
<td>Unconditional jump to dst</td>
</tr>
<tr>
<td>JCS dst</td>
<td>JCC *+2, JCC dst</td>
<td>Jump if carry set</td>
</tr>
<tr>
<td>SUB mem</td>
<td>NOR zero, ADD mem, ADD one</td>
<td>Subtract mem from Accu (one contains 0x01)</td>
</tr>
</tbody>
</table>

Table 2: Examples for macros to implement common instructions.

Some examples of macros to implement instructions known from other CPUs are given in table 2. The listing below shows one of the programs tested on the CPU. It uses Dijkstra’s algorithm to calculate the greatest common divisor of two numbers.

```assembly
Listing 1: GCD example

start:
    NOR allone ; Akku = 0
    NOR b      ; Akku = - b
    ADD a      ; Akku = a - b
    JCC neg    ; Carry set when akku >= 0
7
    STA a
    ADD allone
    JCC end    ; A=0 ? --> end, result in b
15
    JCC start

neg:
    NOR zero   ; Akku = -Akku
    ADD one    ; Akku = -Akku
20
    STA b      ; Carry was not altered
    JCC start  ; Carry was not altered
end:
    JCC end
```

2
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 address/data output had to be avoided at all costs. One consequence was to divide the datapath into one path for the address and one for the data.

In contrast to other small cpus the address generation is not done by 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 'address - datapath' to fit into 12 macrocells.

The 'data - datapath' occupies 14 macrocells. (Eight for the accumulator, one for the carry flag and five combinational macrocells for carry propagation).

![Datapath Diagram]

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 3.

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

<table>
<thead>
<tr>
<th>State</th>
<th>Function</th>
<th>Operations</th>
<th>Next</th>
</tr>
</thead>
<tbody>
<tr>
<td>000 S0</td>
<td>Fetch instruction</td>
<td>( \text{pc} \leftarrow \text{adreg} + 1, \text{adreg} \leftarrow \text{data} ) ( \text{oe} \leftarrow 0, \text{data} \leftarrow Z )</td>
<td>S0 w. opcode = 11, c = 0</td>
</tr>
<tr>
<td></td>
<td>/Operand adress</td>
<td></td>
<td>S1 w. opcode = 10</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>S2 w. opcode = 01</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>S3 w. opcode = 00</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>S5 w. opcode = 11, c = 1</td>
</tr>
<tr>
<td>001 S1</td>
<td>Write akku to memory</td>
<td>( \text{we} \leftarrow 0, \text{data} \leftarrow \text{akku} ) ( \text{adreg} \leftarrow \text{pc} )</td>
<td>S0</td>
</tr>
<tr>
<td>010 S2</td>
<td>Read operand, ADD</td>
<td>( \text{oe} \leftarrow 0, \text{data} \leftarrow z, \text{adreg} \leftarrow \text{pc} )</td>
<td>S0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \text{akku} \leftarrow \text{akku} + \text{data} ), update carry</td>
<td></td>
</tr>
<tr>
<td>011 S3</td>
<td>Read operand, NOR</td>
<td>( \text{oe} \leftarrow 0, \text{data} \leftarrow z, \text{adreg} \leftarrow \text{pc} )</td>
<td>S0</td>
</tr>
<tr>
<td>101 S5</td>
<td>Clear carry, Read PC</td>
<td>( \text{akku} \leftarrow \text{akku NOR data} ) ( \text{carry} \leftarrow 0, \text{adreg} \leftarrow \text{pc} )</td>
<td>S0</td>
</tr>
</tbody>
</table>

Table 3: The state machine.

3 Sources

A ZIP-Archive containing the VHDL-Sources of the CPU and the testbench can be downloaded at: .
-- 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);
  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');
      states <= "000";
      pc <= (others => '0');
    elsif rising_edge(clk) then
      -- PC / Address 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 "001" => akku <= ("0" & akku(7 downto 0)) + ("0" & data);
      when "011" => akku(7 downto 0) <= akku(7 downto 0) nor data;
      when "101" => akku(8) <= '0';
      when others => null;
      end case;

      -- State machine
      if (states /= "000") then states <= "000";
      elsif (data(7 downto 6) = "11" and akku(8)=1') then states <= "101";
      else states <= "0" & not data(7 downto 6);
    end if;

    -- output
    adress <= adreg;
    oe <= "ZZZZZZZ" when states /= "001" else akku(7 downto 0);
    we <= '1' when (clk=1') or states = "001" or rst='0' or states = "101" else '0';
  end process;
end CPU_ARCH;
module vCpu3(data, adress, oe, we, rst, clk);

inout [7:0] data;
output [5:0] adress;
output oe;
output we;
input rst;
input clk;

reg [8:0] accumulator; // accumulator(8) is carry!
reg [5:0] adreg;
reg [5:0] pc;
reg [2:0] states;

always @(posedge clk)
if (!rst)
begin
  adreg <= 0;
  states <= 0;
  accumulator <= 0;
end
else begin
  // PC / Address path
  if (!|states) begin
    pc <= adreg + 1;
    adreg <= pc;
  end
  else adreg <= pc;

  // ALU / Data Path
  case(states)
  3'b010: accumulator <= {1'b0, accumulator[7:0]} + {1'b0, data}; // add
  3'b011: accumulator[7:0] <= ~((accumulator[7:0][data]); // nor
  3'b101: accumulator[8] <= 1'b0; // branch not taken, clear carry
  default: instruction fetch, jcc taken
  endcase

  // State machine
  if (!|states) states <= 0;
  else begin
    if ( |data[7:6] & & accumulator[8]) states <= 3'b101;
    else states <= {1'b0, |data[7:6]};
  end

  // output
  assign adress = adreg;
  assign data = states!=3'b001 ? accumulator[7:0] : 8'bZZZZZZZZ;
  assign oe = clk | !rst | (states==3'b001);
  assign we = clk | !rst | (states!=3'b001);
endmodule

Listing 3: Verilog version of the CPU, unverified.