How to design your own CPU on FPGAs with VHDL

A Lecture by Dr. Juergen Sauermann

Note: The figures look pretty bad. I am working on the scaling of the PNG files. In the meantime, plase use the HTML version.

Table of Contents

Table of Contents

Table of Contents

Table of Contents

1 INTRODUCTION AND OVERVIEW

This lecture describes in detail how you can design a CPU (actually an embedded system) in VHDL.

The CPU has an instruction set similar to the instruction set of the popular 8-bit CPUs made by **Atmel**. The instruction set is described in http://www.atmel.com/dyn/resources/prod_documents/doc0856.pdf. We use an existing CPU so that we can reuse a software tool chain (**avr-gcc**) for this kind of CPU and focus on the hardware aspects of the design. At the end of the lecture, however, you will be able to design your own CPU with a different instruction set.

We will not implement the full instruction set; only the fraction needed to explain the principles, and to run a simple "Hello world" program (and probably most C programs) will be described.

1.1 Prerequisites

Initially you will need two programs:

- **ghdl** from http://ghdl.free.fr (a free VHDL compiler and simulator) and
- **gtkwave** from http://gtkwave.sourceforge.net (a free visualization tool for the output of ghdl).

These two programs allow you to design the CPU and simulate its functions.

Later on, you will need a FPGA toolchain for creating FPGA design files and to perform timing simulations. For this lecture we assume that the free **Xilinx Webpack** tool chain is used (http://www.xilinx.com). The latest version of the Xilinx Webpack provides ISE 11 (as of Nov. 2009) but we used ISE 10.1 because we used an FPGA board providing a good old Spartan 2E FPGA (actually an xc2s300e device) which is no longer supported in ISE 11.

Once the CPU design is finished, you need a C compiler that generates code for the AVR CPU. For downloading **avr-gcc**, start here:

http://www.avrfreaks.net/wiki/index.php/Documentation:AVR_GCC#AVR-GCC_on_Unix_and_Linux

In order to try out the CPU, you will need some FPGA board and a programming cable. We used a Memec "Spartan-IIE LC Development Kit" board and an Avnet "Parallel Cable 3" for this purpose. These days you will want to use a more recent development environment. When the hardware runs, the final thing to get is a software toolchain, for example **avr-gcc** for the instruction set and memory layout used in this lecture. Optional, but rather helpful, is **eclipse** (http://www.eclipse.org) with the AVR plugin.

Another important prerequisite is that the reader is familiar with VHDL to some extent. You do not need to be a VHDL expert in order to follow this lecture, but you should not be a VHDL novice either.

1.2 Other useful links.

A good introduction into VHDL design with open source tools can be found here:

http://www.armadeus.com/wiki/index.php?title=How_to_make_a_VHDL_design_in_Ubuntu/Debian

1.3 Structure of this Lecture

This lecture is organized as a sequence of lessons. The first lessons will describe the VHDL files of the CPU design in a top-down fashion.

Then follows a lesson on how to compile, simulate and build the design.

Finally there is a listing of all design files with line numbers. Pieces of these design files will be spread over the different lessons and explained there in detail. In the end, all code lines in the appendix should have been explained, with the exception of comments, empty lines and the like. Repetitions such as the structure of VHDL files or different opcodes that are implemented in the same way, will only be described once.

All source files are provided (without line numbers) in a tar file that is stored next to this lecture.

1.4 Naming Conventions

In all lessons and in the VHDL source files, the following conventions are used:

VHDL entities and components and VHDL keywords are written in **lowercase**. Signal names and variables are written in **UPPERCASE**. Each signal has a prefix according to the following rules:

I_	for inputs of a VHDL entity.
Q_	for outputs of a VHDL entity.
L_	for local signals that are generated by a VHDL construct (e.g. by a signal assignment).
x_	with an uppercase x for signals **generated** by an instantiated entity.

For every instantiated component we choose an uppercase letter x (other than I, L, or Q). All signals driven by the component then get the prefix **Q_** (if the instantiated component drives an output of the entity being defined) or the prefix **x_** (if the component drives an internal signal).

Apart from the prefix, we try to keep the name of a signal the same across different VHDL files. Unless the prefix matters, we will use the signal name **without its prefix** in our descriptions.

Another convention is that we use one VHLD source file for every entity that we define and that the name of the file (less the .vhd extension) matches the entity name.

1.5 Directory Structure

Create a directory of your choice. In that directory, **mkdir** the following sub-directories:

app	for building the program that will run on the CPU (i.e. **hello.c**)
simu	for object files generated by **ghdl**
src	for VHDL source files of the CPU and **avr_fpga.ucf**
test	for a VHDL testbench
tools	for tools **end_conv** and **make_mem**
work	working directory for **ghdl**

Initially the directory should look like this:

```
# ls -R .
./app:
hello.c

./simu:

./src:
alu.vhd
avr_fpga.ucf
avr_fpga.vhd
baudgen.vhd
common.vhd
COPYING
cpu_core.vhd
data_mem.vhd
data_path.vhd
io.vhd
opc_deco.vhd
opc_fetch.vhd
prog_mem_content.vhd
prog_mem.vhd
reg_16.vhd
register_file.vhd
segment7.vhd
status_reg.vhd
uart_rx.vhd
uart_tx.vhd
uart.vhd

./test:
RAMB4_S4_S4.vhd
test_tb.vhd

./tools:
end_conv.cc
make_mem.cc

./work:
```

1.6 Other Useful Tools

This lecture was prepared with 3 excellent tools:

- **vim** 7.1.38 for preparing the text of the lecture,
- txt2html 2.46 for converting the text into html, and
- **dia** 0.5 for drawing the figures.

2 TOP LEVEL

This lesson defines what we want to create and the top level VHDL file for it.

2.1 Design Purpose

We assume that the purpose of the design is to build an FPGA with the following features:

- a CPU similar to the Atmel ATmega8,
- a serial port with a fixed baud rate, and
- an output for a single digit 7-segment display.

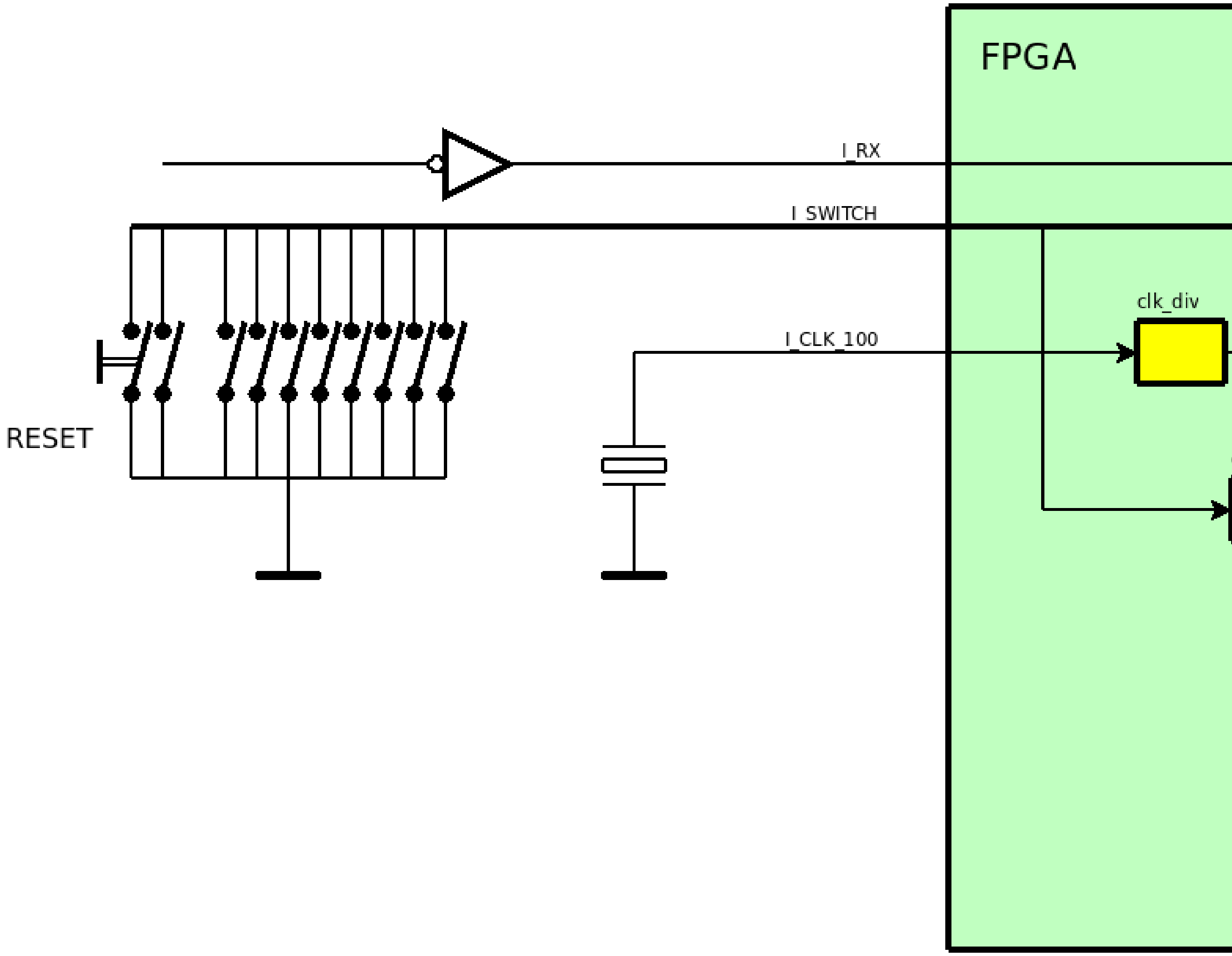
It is assumed that a suitable hardware exists.

2.2 Top Level Design

A CPU with I/O is a somewhat complicated beast. In order to tame it, we brake it down into smaller and smaller pieces until the pieces become trivial.

The trick is to perform the breakdown at points where the connection between the pieces is weak (meaning that it consists of only a few signals).

The top level of our FPGA design, and a few components around the FPGA, looks like this:



This design consists of 2 big sub-components **cpu** and **ino**, a small sub-component **seg**, and some local processes like **clk_div** and **deb** that are not broken down in other VHDL files, but rather written directly in VHDL.

cpu and **ino** are described in lessons 4 and 8.

seg is a debug component that has the current program counter (**PC**) of the CPU as input and displays it as 4 hex digits that show up one by one with a short break between every sequence of hex digits. Since **seg** is rather board specific, we will not describe it in detail in this lecture.

The local processes are described further down in this lesson. Before that we explain the structure that will be used for all VHDL source file.

2.3 General Structure of our VHDL Files

2.3.1 Header and Library Declarations

Each VHDL source file starts with a comment containing a copyright notice (all files are released under the GPL) and the purpose of the file. Then follows a declaration of the libraries being used.

2.3.2 Entity Declaration

After the header and libraries, the entity that is defined in the VHDL file is declared. We declare one entity per VHDL file. The declaration consists of the name of the entity, the inputs, and the outputs of the entity. In this declaration the order of input and outputs does not matter, but we try to stick to the convention to declare the inputs before the outputs.

There is one exception: the file **common.vhd** does not define an entity, but a VHDL package. This package contains the definitions of constants that are used in more than one VHDL source file. This ensures that changes to these constants happen in all files using them.

2.3.3 Entity Architecture

Finally, the architecture of the entity is specified. The architecture consists of a header and a body.

In the header we declare the components, functions, signals, and constants that are used in the body.

The body defines how the items declared in the header are being used (i.e. instantiated, interconnected etc.). This body contains, so to say, the "intelligence" of the design.

2.4 avr_fpga.vhd

The top level of our design is defined in **avr_fpga.vhd**. Since this is our first VHDL file we explain it completely, line by line. Later on, we will silently skip repetitions of parts that have been described in other files or that are very similar.

2.4.1 Header and Library Declarations

avr_fpga.vhd starts with a copyright header.

```
-------------------------------------------------------------------------------
--
-- Copyright (C) 2009, 2010 Dr. Juergen Sauermann
--
--  This code is free software: you can redistribute it and/or modify
--  it under the terms of the GNU General Public License as published by
--  the Free Software Foundation, either version 3 of the License, or
--  (at your option) any later version.
--
--  This code is distributed in the hope that it will be useful,
--  but WITHOUT ANY WARRANTY; without even the implied warranty of
```

```
--  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
--  GNU General Public License for more details.
--
--  You should have received a copy of the GNU General Public License
--  along with this code (see the file named COPYING).
--  If not, see http://www.gnu.org/licenses/.
--
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--
-- Module Name:     avr_fpga - Behavioral
-- Create Date:     13:51:24 11/07/2009
-- Description:     top level of a CPU
--
--------------------------------------------------------------------------------

```

src/avr_fpga.vhd

The libraries used are more or less the same in all VHDL files:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

```

src/avr_fpga.vhd

The only Xilinx specific components needed for this lecture are the block RAMs. For functional simulation we provide compatible components written in VHDL. For FPGAs from other vendors you can probably include their FPGA libraries, but we have not tested this.

2.4.2 Entity Declaration

For a top level entity, the inputs and outputs of the entities are the pins of the FPGA.

For a lower level entity, the inputs and outputs of the entity are associated ("connected") with the inputs and outputs of the instance of the entity in a higher level entity. For this to work, the inputs and outputs of the declaration should match the component declaration in the architecture of another entity that instantiates the entity being declared.

Since we discuss the top-level, our inputs and outputs are pins of the FPGA. The top level entity is declared like this:

```
entity avr_fpga is
    port (  I_CLK_100   : in  std_logic;
            I_SWITCH    : in  std_logic_vector(9 downto 0);
            I_RX        : in  std_logic;

            Q_7_SEGMENT : out std_logic_vector(6 downto 0);
            Q_LEDS      : out std_logic_vector(3 downto 0);
            Q_TX        : out std_logic);
```

src/avr_fpga.vhd

We therefore have the following FPGA pins:

Pin	Purpose
I_CLK_100	a 100 MHz Clock from the board.
I_SWITCH	a 8 bit DIP switch and two single push-buttons.
I_RX	the serial input of our UART.
Q_7_SEGMENT	7 lines to the LEDs of a 7-segment display.
Q_LEDS	4 lines to single LEDs
Q_TX	the serial output of our UART.

The lower 8 bits of **SWITCH** come from a DIP switch while the upper two bits come from two push-buttons. The two push-buttons are used as reset buttons, while the DIP switch goes to the I/O component from where the CPU can read the value set on the switch.

2.4.3 Architecture of the Top Level Entity

The architecture has a head and a body, The head starts with the **architecture** keyword. The body starts with **begin** and ends with **end**, like this:

```
architecture Behavioral of avr_fpga is
```

src/avr_fpga.vhd

```
begin
```

src/avr_fpga.vhd

```
end Behavioral;
```

src/avr_fpga.vhd

2.4.3.1 Architecture Header

As we have seen in the first figure in this lesson, the top level uses 3 components: **cpu**, **io**, and **seg**. These components have to be declared in the header of the architecture. The architecture contains component declarations, signal declarations and others. We normally declare components and signals in the following order:

- declaration of functions
- declaration of constants
- declaration of the first component (type)
- declaration of signals driven by (an instance of) the first component
- declaration of the second component (type)
- declaration of signals driven by (an instance of) the second component
- ...
- declaration of signals driven by local processes and the like.

The first component declaration **cpu_core** and the signals driven by its instances are:

```
    component cpu_core
        port (  I_CLK       : in  std_logic;
                I_CLR       : in  std_logic;
                I_INTVEC    : in  std_logic_vector( 5 downto 0);
                I_DIN       : in  std_logic_vector( 7 downto 0);

                Q_OPC       : out std_logic_vector(15 downto 0);
                Q_PC        : out std_logic_vector(15 downto 0);
                Q_DOUT      : out std_logic_vector( 7 downto 0);
                Q_ADR_IO    : out std_logic_vector( 7 downto 0);
                Q_RD_IO     : out std_logic;
                Q_WE_IO     : out std_logic);
    end component;

    signal  C_PC            : std_logic_vector(15 downto 0);
    signal  C_OPC           : std_logic_vector(15 downto 0);
    signal  C_ADR_IO        : std_logic_vector( 7 downto 0);
    signal  C_DOUT          : std_logic_vector( 7 downto 0);
    signal  C_RD_IO         : std_logic;
    signal  C_WE_IO         : std_logic;
```

src/avr_fpga.vhd

The second component declaration **io** and its signals are:

```
    component io
        port (  I_CLK       : in  std_logic;
                I_CLR       : in  std_logic;
                I_ADR_IO    : in  std_logic_vector( 7 downto 0);
                I_DIN       : in  std_logic_vector( 7 downto 0);
```

```
                I_RD_IO     : in  std_logic;
                I_WE_IO     : in  std_logic;
                I_SWITCH    : in  std_logic_vector( 7 downto 0);
                I_RX        : in  std_logic;

                Q_7_SEGMENT : out std_logic_vector( 6 downto 0);
                Q_DOUT      : out std_logic_vector( 7 downto 0);
                Q_INTVEC    : out std_logic_vector(5 downto 0);
                Q_LEDS      : out std_logic_vector( 1 downto 0);
                Q_TX        : out std_logic);
    end component;

    signal N_INTVEC         : std_logic_vector( 5 downto 0);
    signal N_DOUT           : std_logic_vector( 7 downto 0);
    signal N_TX             : std_logic;
    signal N_7_SEGMENT      : std_logic_vector( 6 downto 0);
```

src/avr_fpga.vhd

Note: Normally we would have used **I_** as a prefix for signals driven by instance **ino** of **io**. This conflicts, however, with the prefix reserved for inputs and we have used the next letter **n** of **ino** as prefix instead.

The last component is **seg**:

```
    component segment7
        port ( I_CLK        : in  std_logic;

               I_CLR        : in  std_logic;
               I_OPC        : in  std_logic_vector(15 downto 0);
               I_PC         : in  std_logic_vector(15 downto 0);

               Q_7_SEGMENT  : out std_logic_vector( 6 downto 0));
    end component;

    signal S_7_SEGMENT      : std_logic_vector( 6 downto 0);
```

src/avr_fpga.vhd

The local signals, which are not driven by any component, but by local processes and inputs of the entity, are:

```
    signal L_CLK            : std_logic := '0';
    signal L_CLK_CNT        : std_logic_vector( 2 downto 0) := "000";
    signal L_CLR            : std_logic;            -- reset,  active low
    signal L_CLR_N          : std_logic := '0';     -- reset,  active low
    signal L_C1_N           : std_logic := '0';     -- switch debounce, active low
    signal L_C2_N           : std_logic := '0';     -- switch debounce, active low

    begin
```

src/avr_fpga.vhd

The **begin** keyword in the last line marks the end of the header of the architecture and the start of its body.

2.4.3.2 Architecture Body

We normally use the following order in the architecture body:

- components instantiated
- processes
- local signal assignments

Thus the architecture body is more or less using the same order as the architecture header. The component instantiations instantiate one or more instances of a component type and connect the "ports" of the instantiated component to the signals in the architecture.

The first component declared was **cpu** so we also instantiate it first:

```
        cpu : cpu_core
        port map(   I_CLK       => L_CLK,
                    I_CLR       => L_CLR,
                    I_DIN       => N_DOUT,
                    I_INTVEC    => N_INTVEC,

                    Q_ADR_IO    => C_ADR_IO,
                    Q_DOUT      => C_DOUT,
                    Q_OPC       => C_OPC,
                    Q_PC        => C_PC,
                    Q_RD_IO     => C_RD_IO,
                    Q_WE_IO     => C_WE_IO);
```

src/avr_fpga.vhd

The first line instantiates a component of type **cpu_core** and calls the instance **cpu**. The following lines map the names of the ports in the component declaration (in the architecture header) to either inputs of the entity, outputs of the entity, or signals declared in the architecture header.

We take **cpu** as an opportunity to explain our naming convention for signals. Our rule for entity inputs and outputs has the consequence that all left sides of the port map have either an **I_** prefix or an **O_** prefix. This follows from the fact that a component instantiated in one architecture corresponds to an entity declared in some other VHDL file and there the **I_** or **O_** convention applies,

The next observation is that all component outputs either drive an entity output or a local signal that starts with the letter chosen for the instantiated component. This is the C_ in the **cpu** case. The **cpu** does not drive an entity output directly (so all outputs map to **C_** signals), but in the **io** outputs there is one driving an entity output (see below).

Finally the component inputs can be driven from more or less anywhere, but from the prefix (**L_** or not) we can see if the signal is directly driven by another component (when the prefix is not **L_**) or by some logic defined further down in the architecture (signal assignments, processes).

Thus for the **cpu** component we can already tell that this component drives a number of local signals (that are not entity outputs but inputs to other components or local processes)> These signals are those on the right side of the port map starting with **C_**. We can also tell that there are some local processes generating a clock signal **L_CLK** and a reset signal **L_CLR**, and the **ino** component (which uses prefix **N_**) drives an interrupt vector **N_INTVEC** and a data bus **N_DOUT**.

The next two components being instantiated are **ino** of type **io** and **seg** of type **segment7**:

```
        ino : io
        port map(   I_CLK       => L_CLK,
                    I_CLR       => L_CLR,
                    I_ADR_IO    => C_ADR_IO,
                    I_DIN       => C_DOUT,
                    I_RD_IO     => C_RD_IO,
                    I_RX        => I_RX,
                    I_SWITCH    => I_SWITCH(7 downto 0),
                    I_WE_IO     => C_WE_IO,

                    Q_7_SEGMENT => N_7_SEGMENT,
                    Q_DOUT      => N_DOUT,
                    Q_INTVEC    => N_INTVEC,
                    Q_LEDS      => Q_LEDS(1 downto 0),
                    Q_TX        => N_TX);

        seg : segment7
        port map(   I_CLK       => L_CLK,
                    I_CLR       => L_CLR,
                    I_OPC       => C_OPC,
                    I_PC        => C_PC,

                    Q_7_SEGMENT => S_7_SEGMENT);
```

src/avr_fpga.vhd

Then come the local processes. The first process divides the 100 MHz input clock from the board into a 25 MHz clock used by the CPU. The design can, with some tuning of the timing optimizations, run at 33 MHz, but for our purposes we are happy with 25 MHz.

```
        clk_div : process(I_CLK_100)
        begin
            if (rising_edge(I_CLK_100)) then
                L_CLK_CNT       <= L_CLK_CNT + "001";
                if (L_CLK_CNT = "001") then
                    L_CLK_CNT       <= "000";
                    L_CLK       <= not L_CLK;
```

```
                end if;
            end if;
        end process;
```

src/avr_fpga.vhd

The second process is **deb**. It debounces the push-buttons used to reset the **cpu** and **ino** components. When either button is pushed ('0' on **SWITCH(8)** or **SWITCH(9)**) then **CLR_N** goes low immediately and stays low for two more cycles after the button was released.

```
        deb : process(L_CLK)
        begin
            if (rising_edge(L_CLK)) then
                -- switch debounce
                if ((I_SWITCH(8) = '0') or (I_SWITCH(9) = '0')) then    -- pushed
                    L_CLR_N     <= '0';
                    L_C2_N      <= '0';
                    L_C1_N      <= '0';
                else                                                    -- released
                    L_CLR_N     <= L_C2_N;
                    L_C2_N      <= L_C1_N;
                    L_C1_N      <= '1';
                end if;
            end if;
        end process;
```

src/avr_fpga.vhd

Finally we have the signals driven directly from other signals. This kind of signal assignments are the same as processes, but use a simpler syntax for frequently used logic.

```
        L_CLR       <= not L_CLR_N;

        Q_LEDS(2)       <= I_RX;
        Q_LEDS(3)       <= N_TX;
        Q_7_SEGMENT        <= N_7_SEGMENT when (I_SWITCH(7) = '1') else S_7_SEGMENT;
        Q_TX       <= N_TX;
```

src/avr_fpga.vhd

CLR is generated by inverting **CLR_N**. The serial input and the serial output are visualized by means of two LEDs. The 7 segment output can be driven from two sources: from a software controlled I/O register in the I/O unit, or from the seven segment component (that shows the current PC and opcode) being executed. The latter component is quite useful for debugging purposes.

Note: We use a **_N** suffix to denote an active low signal. Active low signals normally come from some FPGA pin since inside the FPGA active low signals have no advantage over active high signals. In the good old TTL days active low signals were preferred for strobe signals since the HIGH to LOW transition in TTL was faster than the LOW to high transition. Some active low signals we see today are left-overs from those days. Another common case is switches and push-buttons that are held high with a pull-up resistor when open and short-cut to ground when the switch is closed.

2.5 Component Tree

The following figure shows the breakdown of almost all components in our design. Only the memory modules in **sram** and **pmem** and the individual registers in **regs** are hidden because they would blow up the structure without providing too much information.

We have already discussed the top level **fpga** and its 3 components **cpu**, **ino**, and **seg**.

The **cpu** will be further broken down into a data path **dpath**, an opcode decode **odec**, and an opcode fetch **opcf**. The data path consist of a 16-bit ALU **alui**, a register file **regs**, and a data memory **sram**. The opcode fetch contains a program counter **pcnt** and a program memory **pmem**.

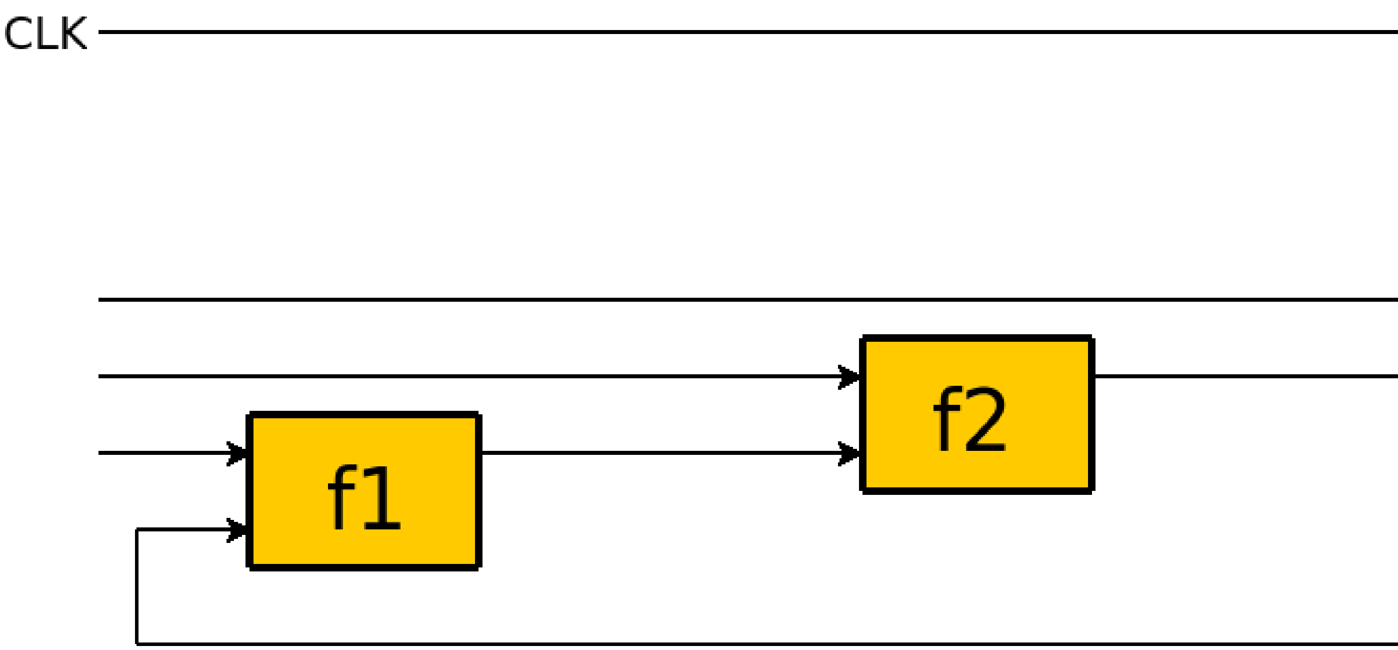
The **ino** contains a **uart** that is further broken down into a baud rate generator **baud**, a serial receiver **rx**, and a serial transmitter **tx**.

3 DIGRESSION: PIPELINING

In this short lesson we will give a brief overview of a design technique known as pipelining. Most readers will already be familiar with it; those readers should take a day off or proceed to the next lesson.

Assume we have a piece of combinational logic that happens to have a long propagation delay even in its fastest implementation. The long delay is then caused by the slowest path through the logic, which will run through either many fast elements (like gates) or a number of slower elements (likes adders or multipliers), or both.

That is the situation where you should use pipelining. We will explain it by an example. Consider the circuit shown in the following figure.

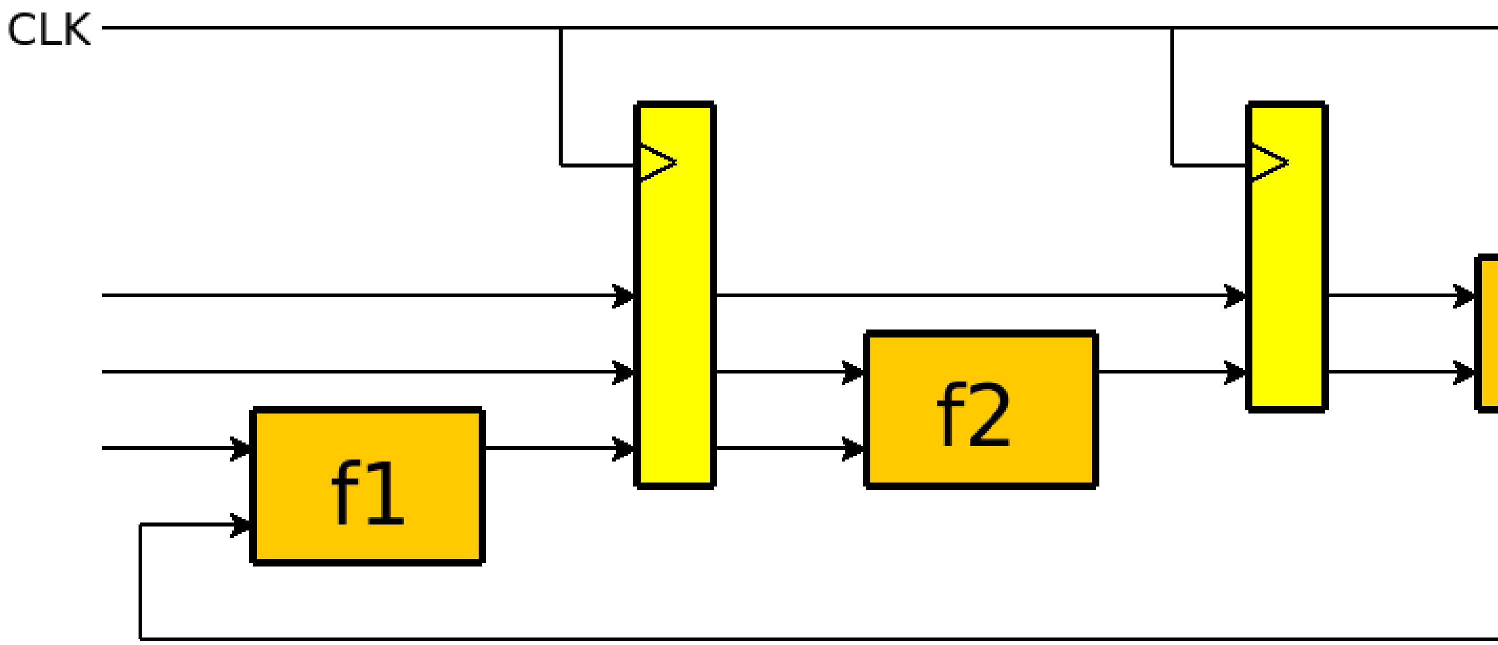


The circuit is a sequential logic which consists of 3 combinational functions f1, f2, and f3 and a flip-flop at the output of f3.

Let t1, t2, and t3 be the respective propagation delays of f1, f2, and f3. Assume that the slowest path of the combinational logic runs from the upper input of f1 towards the output of f3. Then the total delay of the combinational is t = t1 + t2 + t3. The entire circuit cannot be clocked faster than with frequency 1/t.

Now pipelining is a technique that slightly increases the delay of a combinational circuit, but thereby allows different parts of the logic at the same time. The slight increase in total propagation delay is more than compensated by a much higher throughput.

Pipelining divides a complex combinational logic with an accordingly long delay into a number of stages and places flip-flops between the stages as shown in the next figure.

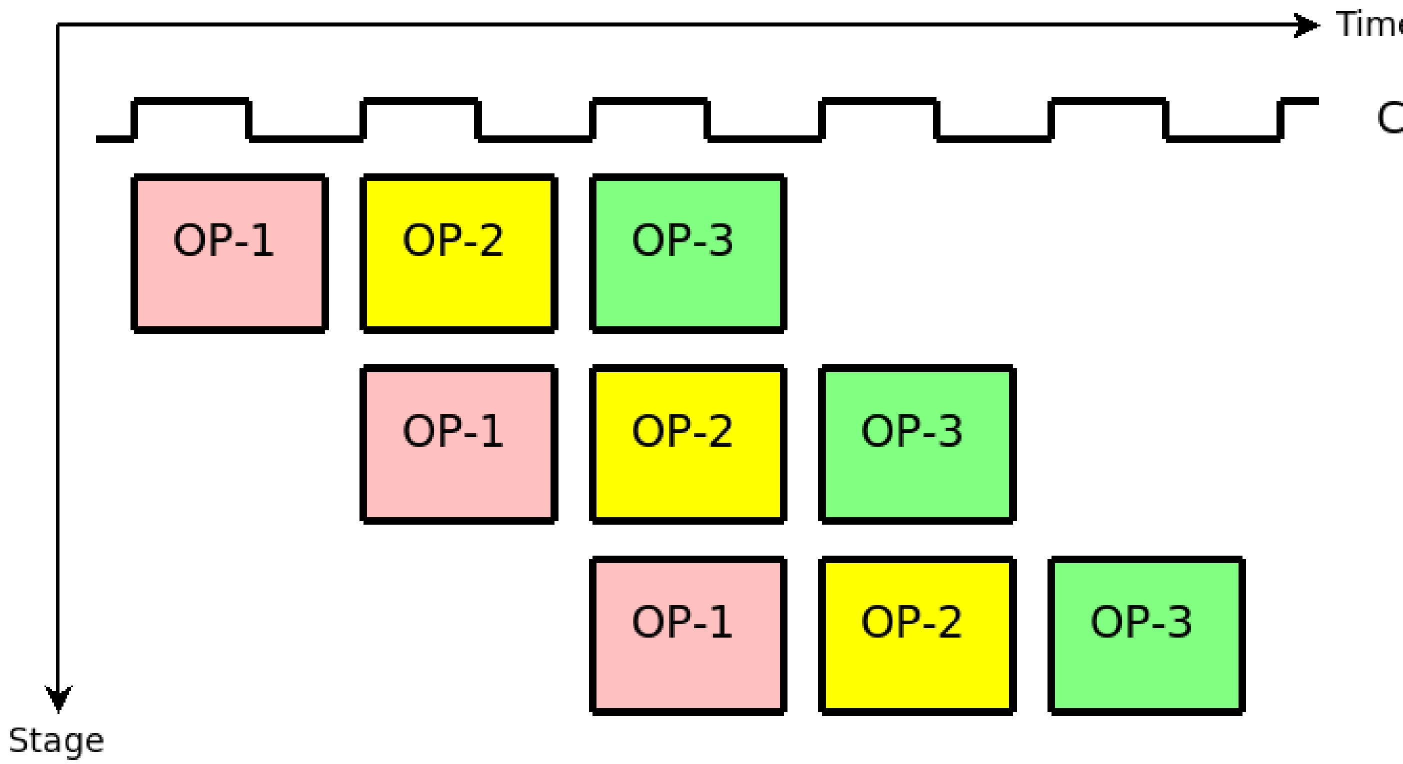


The slowest path is now max(t1, t2, t3) and the new circuit can be clocked with frequency 1/max(t1, t2, t3) instead of 1/(t1 + t2 + t3). If the functions f1, f2, and f3 had equal propagation delays, then the max. frequency of the new circuit would have tripled compared to the old circuit.

It is generally a good idea when using pipelining to divide the combinational logic that shall be pipelined into pieces with similar delay. Another aspect is to divide the combinational logic at places where the number of connections between the pieces is small since this reduces the number of flip-flops that are being inserted.

The first design of the CPU described in this lecture had the opcode decoding logic (which is combinational) and the data path logic combined. That design had a worst path delay of over 50 ns (and hence a max. frequency of less than 20 MHz). After splitting of the opcode decoder, the worst path delay was below 30 ns which allows for a frequency of 33 MHz. We could have divides the pipeline into even more stages (and thereby increasing the max. frequency even further). This would, however, have obscured the design so we did not do it.

The reason for the improved throughput is that the different stages of a pipeline work in parallel while without pipelining the entire logic would be occupied by a single operation. In a pipeline the single operation is often displayed like this (one color = one operation).



This kind of diagram shows how an operation is distributed over the different stages over time.

To summarize, pipelining typically results in:

- a slightly more complex design,
- a moderately longer total delay, and
- a considerable improvement in throughput.

4 THE CPU CORE

In this lesson we will discuss the core of the CPU. These days, the same kind of CPU can come in different flavors that differ in the clock frequency that that support, bus sizes, the size of internal caches and memories and the capabilities of the I/O ports they provide. We call the common part of these different CPUs the **CPU core**. The CPU core is primarily characterized by the instruction set that it provides. One could also say that the CPU core is the implementation of a given instruction set.

The details of the instruction set will only be visible at the next lower level of the design. At the current level different CPUs (with different instruction sets) will still look the same because they all use the same structure. Only some control signals will be different for different CPUs.

We will use the so-called **Harvard architecture** because it fits better to FPGAs with internal memory modules. Harvard architecture means that the program memory and the data memory of the CPU are different. This gives us more flexibility and some instructions (for example **CALL**, which involves storing the current program counter in memory while changing the program counter and fetching the next instruction) can be executed in parallel).

Different CPU cores differ in the in the instruction set that they support. The types of CPU instructions (like arithmetic instructions, move instructions, branch instructions, etc.) are essentially the same for all CPUs. The differences are in the details like the encoding of the instructions, operand sizes, number of registers addressable, and the like).

Since all CPUs are rather similar apart from details, within the same base architecture (Harvard vs. von Neumann), the same structure can be used even for different instruction sets. This is because the same cycle is repeated again and again for the different instructions of a program. This cycle consists of 3 phases:

- Opcode fetch
- Opcode decoding
- Execution

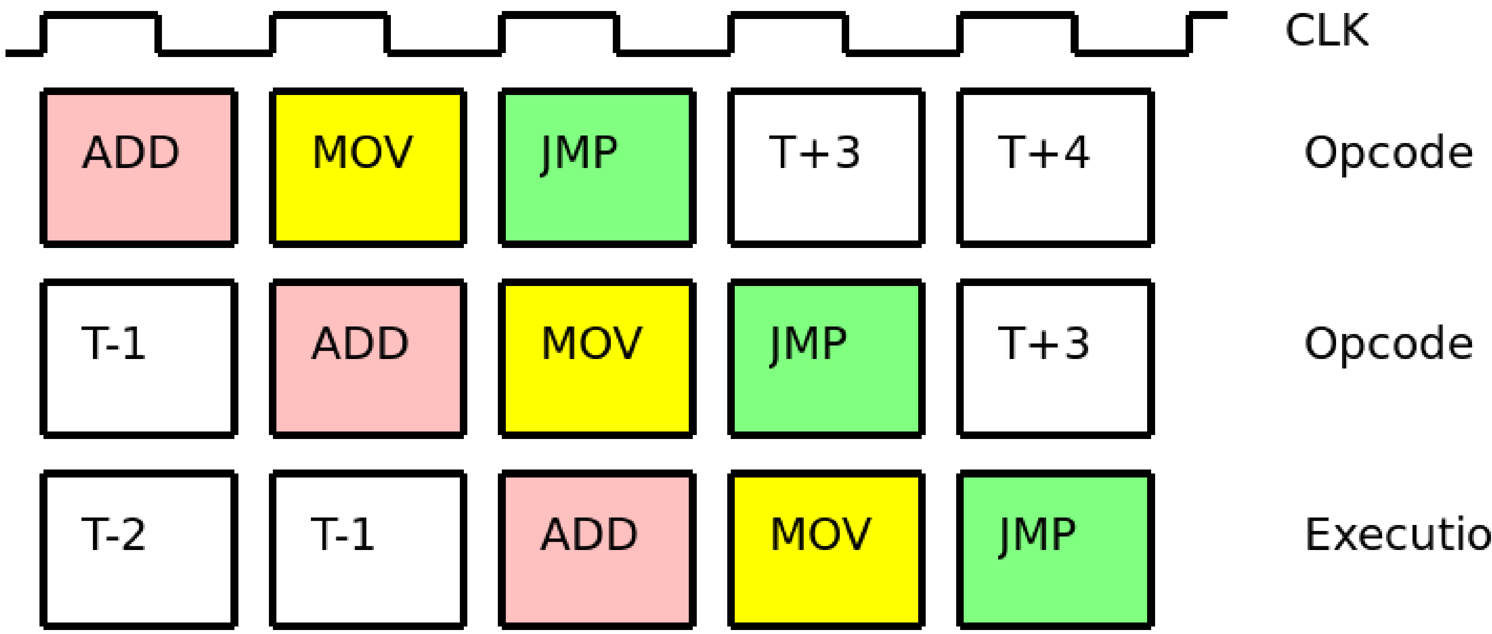
Opcode fetch means that for a given value of the program counter **PC**, the instruction (opcode) stored at location PC is read from the program memory and that the PC is advanced to the next instruction.

Opcode decoding computes a number of control signals that will be needed in the execution phase.

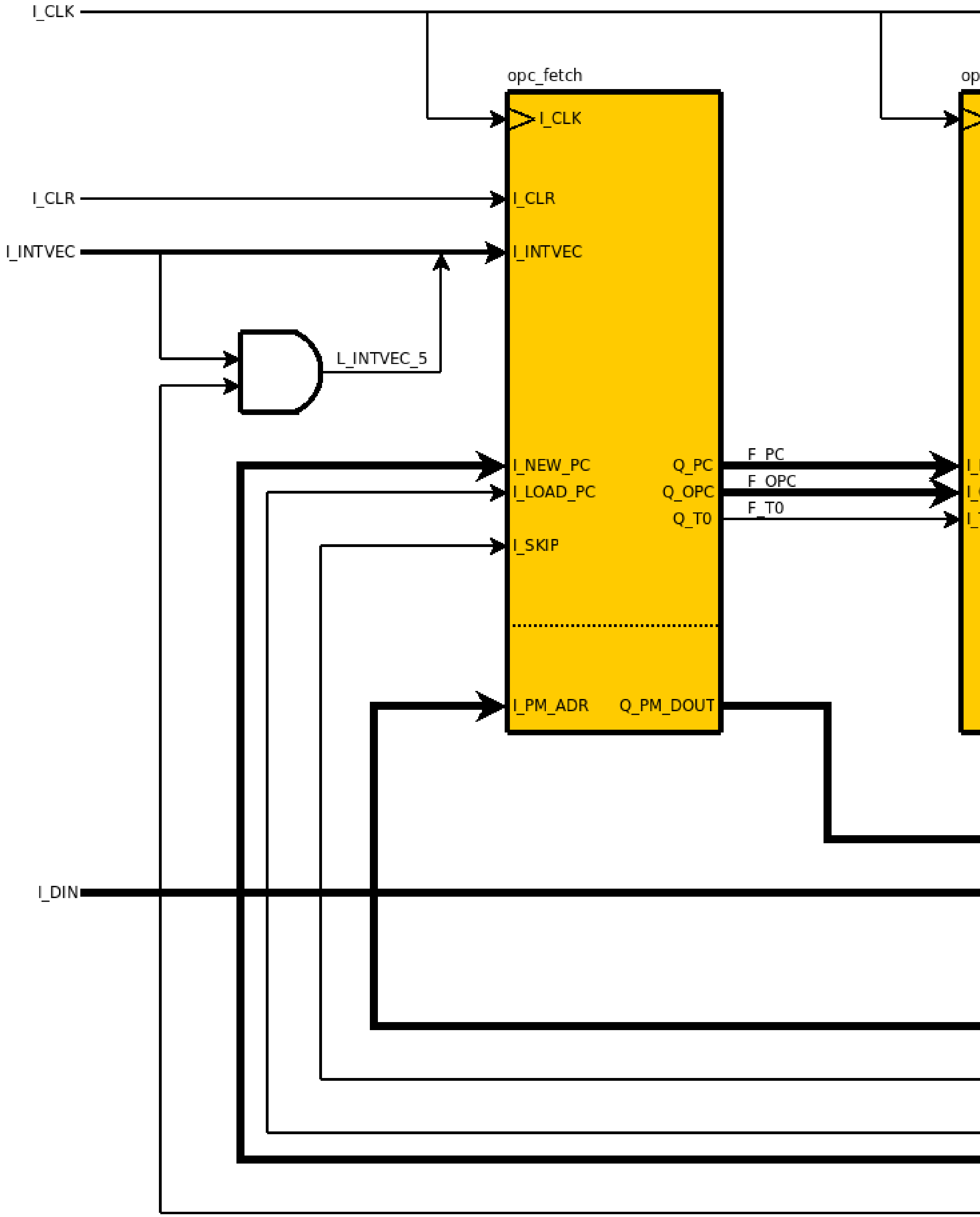
Execution then executes the opcode which means that a small number of registers or memory locations is read and/or written.

In theory these 3 phases could be implemented in a combinational way (a static program memory, an opcode decoder at the output of the program memory and an execution module at the output of the opcode decoder). We will see later, however, that each phase has a considerable complexity and we therefore use a 3 stage pipeline instead.

In the following figure we see how a sequence of three opcodes ADD, MOV, and JMP is executed in the pipeline.



From the discussion above we can already predict the big picture of the CPU core. It consists of a pipeline with 3 stages opcode fetch, opcode decoder, and execution (which is called data path in the design because the operations required by the execution more or less imply the structure of the data paths in the execution stage:


I_CLK
opc_fetch
op
I_CLK
I_CLR
I_CLR
I_INTVEC
I_INTVEC
L_INTVEC_5
I_NEW_PC
Q_PC
F_PC
I_LOAD_PC
Q_OPC
F_OPC
Q_T0
F_T0
I_SKIP
I_PM_ADR
Q_PM_DOUT
I_DIN

The pipeline consists of the **opc_fetch** stage that drives **PC**, **OPC**, and **T0** signals to the opcode decoder stage. The **opc_deco** stage decodes the **OPC** signal and generates a number of control signals towards the execution stage, The execution stage then executes the decoded instruction.

The control signals towards the execution stage can be divided into 3 groups:

1. Select signals (**ALU_OP**, **AMOD**, **BIT**, **DDDDD**, **IMM**, **OPC**, **PMS**, **RD_M**, **RRRRR**, and **RSEL**). These signals control details (like register numbers) of the instruction being executed.
2. Branch and timing signals (**PC**, **PC_OP**, **WAIT**, (and **SKIP** in the reverse direction)). These signals control changes in the normal execution flow.
3. Write enable signals (**WE_01**, **WE_D**, **WE_F**, **WE_M**, and **WE_XYZS**). These signals define if and when registers and memory locations are updated.

We come to the VHDL code for the CPU core. The entity declaration must match the instantiation in the top-level design. Therefore:

```
    entity cpu_core is
        port (  I_CLK       : in  std_logic;
                I_CLR       : in  std_logic;
                I_INTVEC    : in  std_logic_vector( 5 downto 0);
                I_DIN       : in  std_logic_vector( 7 downto 0);

                Q_OPC       : out std_logic_vector(15 downto 0);
                Q_PC        : out std_logic_vector(15 downto 0);
                Q_DOUT      : out std_logic_vector( 7 downto 0);
                Q_ADR_IO    : out std_logic_vector( 7 downto 0);
                Q_RD_IO     : out std_logic;
                Q_WE_IO     : out std_logic);
```

src/cpu_core.vhd

The declaration and instantiation of **opc_fetch**, **opc_deco**, and **dpath** simply reflects what is shown in the previous figure.

The multiplexer driving **DIN** selects between data from the I/O input and data from the program memory. This is controlled by signal **PMS** (**program memory select**):

```
        L_DIN       <= F_PM_DOUT when (D_PMS = '1') else I_DIN(7 downto 0);
```

src/cpu_core.vhd

The interrupt vector input **INTVEC** is **and**'ed with the global interrupt enable bit in the status register (which is contained in the data path):

```
        L_INTVEC_5      <= I_INTVEC(5) and R_INT_ENA;
```

src/cpu_core.vhd

This concludes the discussion of the CPU core and we will proceed with the different stages of the pipeline. Rather than following the natural order (opcode fetch, opcode decoder, execution), however, we will describe the opcode decoder last. The reason is that the opcode decoder is a consequence of the design of the execution stage. Once the execution stage is understood, the opcode decoder will become obvious (though still complex).

5 OPCODE FETCH

In this lesson we will design the opcode fetch stage of the CPU core. The opcode fetch stage is the simplest stage in the pipeline. It is the stage that put life into the CPU core by generating a sequence of opcodes that are then decoded and executed. The opcode fetch stage is sometimes called the **sequencer** of the CPU.

Since we use the Harvard architecture with separate program and data memories, we can simply instantiate the program memory in the opcode fetch stage. If you need more memory than your FPGA provides internally, then you can design address and data buses towards an external memory instead (or in addition). Most current FPGAs provide a lot of internal memory, so we can keep things simple.

The opcode fetch stage contains a sub-component **pmem**, which is is the program memory. The main purpose of the opcode fetch stage is to manipulate the program counter (**PC**) and to produce opcodes. The **PC** is a local signal:

```
    signal L_PC             : std_logic_vector(15 downto 0);
```

src/opc_fetch.vhd

The **PC** is updated on every clock with its next value. The **T0** output is cleared when the **WAIT** signal is raised. This causes the T0 output to be '1' on the first cycle of a 2 cycle instruction and '0' on the second cycle:

```
       lpc: process(I_CLK)
        begin
            if (rising_edge(I_CLK)) then
                L_PC      <= L_NEXT_PC;
                L_T0      <= not L_WAIT;
            end if;
        end process;
```

src/opc_fetch.vhd

The next value of the **PC** depends on the **CLR**, **WAIT**, **LOAD_PC**, and **LONG_OP** signals:

```
        L_NEXT_PC       <= X"0000"        when (I_CLR     = '1')
                else L_PC           when (L_WAIT    = '1')
                else I_NEW_PC       when (I_LOAD_PC = '1')
                else L_PC + X"0002" when (L_LONG_OP = '1')
                else L_PC + X"0001";
```

src/opc_fetch.vhd

The **CLR** signal, which overrides all others, resets the **PC** to 0. It is generated at power on and when the reset input of the CPU is asserted. The **WAIT** signal freezes the **PC** at its current value. It is used when an instruction needs two **CLK** cycles to complete. The **LOAD_PC** signal causes the **PC** to be loaded with the value on the **NEW_PC** input. The **LOAD_PC** signal is driven by the execution stage when a jump instruction is executed. If neither **CLR**, **WAIT**, or **LOAD_PC** is present then the **PC** is advanced to the next instruction. If the current instruction is one of **JMP**, **CALL**, **LDS** and **STS**, then it has a length of two 16-bit words and **LONG_OP** is set. This causes the PC to be incremented by 2 rather than by the normal instruction length of 1:

```
    -- Two word opcodes:
    --
    --        9       3210
    -- 1001 000d dddd 0000 kkkk kkkk kkkk kkkk - LDS
    -- 1001 001d dddd 0000 kkkk kkkk kkkk kkkk - SDS
    -- 1001 010k kkkk 110k kkkk kkkk kkkk kkkk - JMP
    -- 1001 010k kkkk 111k kkkk kkkk kkkk kkkk - CALL
    --
    L_LONG_OP      <= '1' when (((P_OPC(15 downto  9) = "1001010") and
                                 (P_OPC( 3 downto  2) = "11"))       -- JMP, CALL
                            or  ((P_OPC(15 downto 10) = "100100") and
                                 (P_OPC( 3 downto  0) = "0000")))    -- LDS, STS
            else '0';
```

src/opc_fetch.vhd

The **CLR**, **SKIP**, and **I_INTVEC** inputs are used to force a **NOP** (no operation) opcode or an "interrupt opcode" onto the output of the opcode fetch stage. An interrupt opcode is an opcode that does not belong to the normal instruction set of the CPU (and is therefore not generated by assemblers or compilers), but is used internally to trigger interrupt processing (pushing of the PC, clearing the interrupt enable flag, and jumping to specific locations) further down in the pipeline.

```
    L_INVALIDATE      <= I_CLR or I_SKIP;

    Q_OPC      <= X"00000000" when (L_INVALIDATE = '1')
        else P_OPC       when (I_INTVEC(5) = '0')
        else (X"000000" & "00" & I_INTVEC);     -- "interrupt opcode"
```

src/opc_fetch.vhd

CLR is derived from the reset input and also resets the program counter. **SKIP** comes from the execution stage and is used to invalidate parts of the pipeline, for example when a decision was made to take a conditional branch. This will be explained in more detail in the lesson about branching.

5.1 Program Memory

The program memory is declared as follows:

```
entity prog_mem is
    port (  I_CLK       : in  std_logic;

            I_WAIT      : in  std_logic;
            I_PC        : in  std_logic_vector(15 downto 0); -- word address
            I_PM_ADR    : in  std_logic_vector(11 downto 0); -- byte address

            Q_OPC       : out std_logic_vector(31 downto 0);
            Q_PC        : out std_logic_vector(15 downto 0);
            Q_PM_DOUT   : out std_logic_vector( 7 downto 0));
end prog_mem;
```

src/prog_mem.vhd

5.2.1 Dual Port Memory

The program memory is a dual port memory. This means that two different memory locations can be read or written at the same time. We don't write to the program memory, be we would like to read two addresses at the same time. The reason are the **LPM** (load program memory) instructions. These instructions read from the program memory while the program memory is fetching the next instructions. In a way these instructions violate the Harvard architecture, but on the other hand they are extremely useful for string constants in C. Rather than initializing the (typically smaller) data memory with these constants, one can leave them in program memory and access them using **LPM** instructions,

Without a dual port memory, we would have needed to stop the pipeline during the execution of **LPM** instructions. Use of dual port memory avoids this additional complexity.

The second port used for **LPM** instructions consists of the address input **PM_ADR** and the data output **PM_DOUT**. **PM_ADR** is a 12-bit byte address (and consequently **PM_DOUT** is an 8-bit output. In contrast, the other port uses an 11-bit word address.

The other signals of the program memory belong to the first port which is used for opcode fetches.

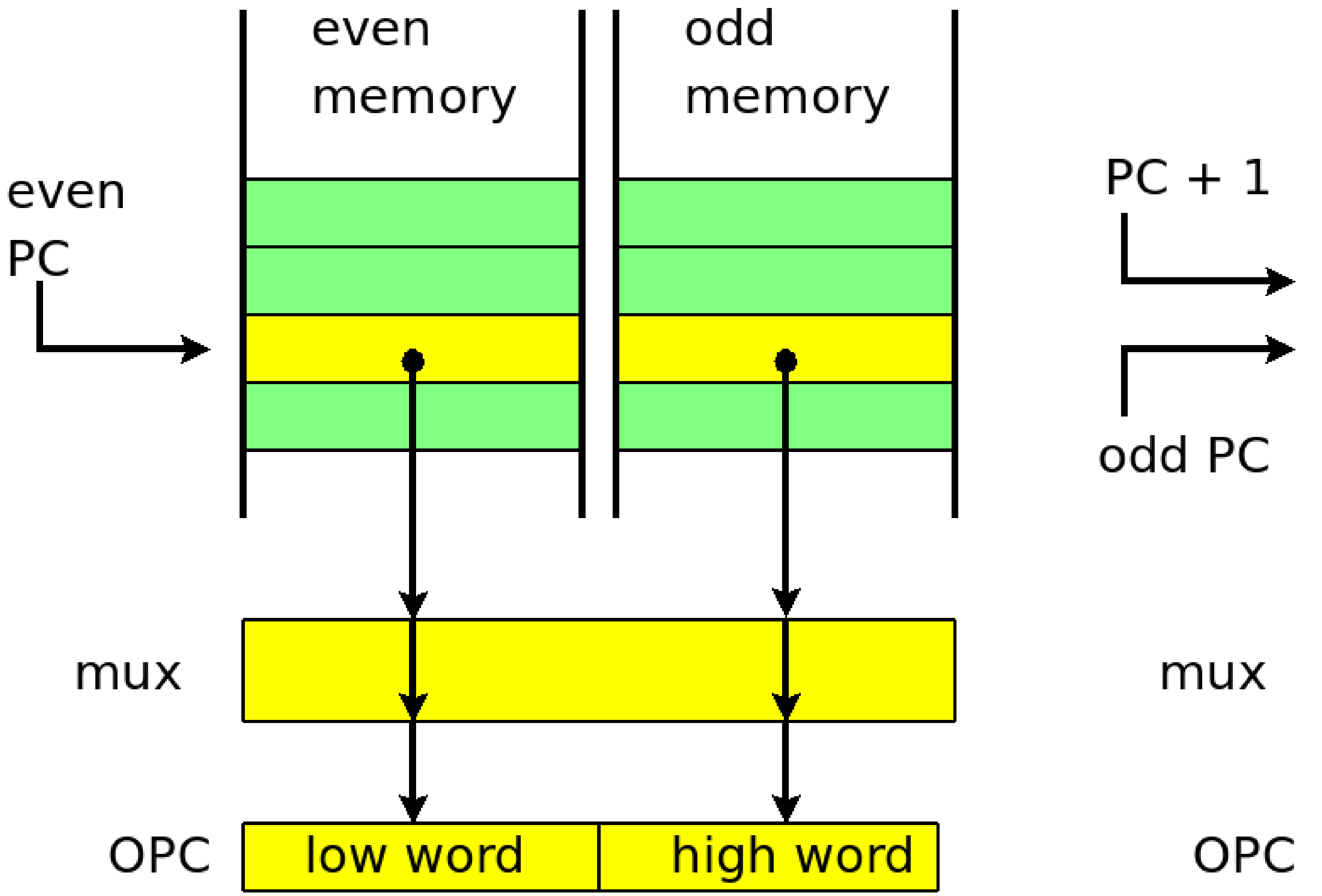
5.2.2 Look-ahead for two word instructions

The vast majority of AVR instructions are single-word (16-bit) instructions. There are 4 exceptions, which are **CALL**, **JMP**, **LDS**, and **STS**. These instructions have addresses (the target address for **CALL** and **JMP** and data memory address for $LDS# and **STS**) in the word following the opcode.

There are two ways to handle such opcodes. One way is to look back in the pipeline when the second word is needed. When one of these instructions reaches the execution stage, then the next word is clocked into the decoding stage (so we could fetch it from there). It might lead to complications, however, when it comes to invalidating the pipeline, insertion of interrupts and the like.

The other way, and the one we choose, is to divide the program memory into an even memory and an odd memory. The internal memory modules in an FPGA are anyhow small and therefore using two memories is almost as simple as using one (both would consist of a number of smaller modules).

There are two cases to consider: (1) an even **PC** (shown on the left of the following figure) and (2) an odd **PC** shown on the right. In both cases do we want the (combined) memory at address **PC** to be stored in the lower word of the **OPC** output and the next word (at **PC**+1) in the upper word of **OPC**.



We observe the following:

- the odd memory address is **PC[10:1]** in both cases.
- the even memory address is **PC[10:1]** + **PC[0]** in both cases.
- the data outputs of the two memories are either straight or crossed, depending (only) on **PC[0]**.

In VHDL, we express this like:

```
        L_PC_O      <= I_PC(10 downto 1);
        L_PC_E      <= I_PC(10 downto 1) + ("000000000" & I_PC(0));
        Q_OPC(15 downto  0)      <= M_OPC_E when L_PC_0 = '0' else M_OPC_O;
        Q_OPC(31 downto 16)      <= M_OPC_E when L_PC_0 = '1' else M_OPC_O;
```

src/prog_mem.vhd

The output multiplexer uses the **PC** and **PM_ADR** of the previous cycle, so we need to remember the lower bit(s) in signals **PC_0** and **PM_ADR_1_0**:

```
        pc0: process(I_CLK)
        begin
            if (rising_edge(I_CLK)) then
                Q_PC      <= I_PC;
                L_PM_ADR_1_0      <= I_PM_ADR(1 downto 0);
                if ((I_WAIT = '0')) then
                    L_PC_0      <= I_PC(0);
                end if;
            end if;
        end process;
```

src/prog_mem.vhd

The split into two memories makes the entire program memory 32-bit wide. Note that the PC is a word address, while PM_ADR is a byte address.

5.2.3 Memory block instantiation and initialization.

The entire program memory consists of 8 memory modules, four for the even half (components **pe_0**, **pe_1**, **pe_2**, and **pe_3**) and four for the odd part (**po_0**, **po_1**, **po_2**, and **po_3**).

We explain the first module in detail:

```
        pe_0 : RAMB4_S4_S4 ---------------------------------------------------------
        generic map(INIT_00 => pe_0_00, INIT_01 => pe_0_01, INIT_02 => pe_0_02,
                    INIT_03 => pe_0_03, INIT_04 => pe_0_04, INIT_05 => pe_0_05,
                    INIT_06 => pe_0_06, INIT_07 => pe_0_07, INIT_08 => pe_0_08,
                    INIT_09 => pe_0_09, INIT_0A => pe_0_0A, INIT_0B => pe_0_0B,
                    INIT_0C => pe_0_0C, INIT_0D => pe_0_0D, INIT_0E => pe_0_0E,
                    INIT_0F => pe_0_0F)
```

```
        port map(ADDRA => L_PC_E,                  ADDRB => I_PM_ADR(11 downto 2),
                 CLKA  => I_CLK,                   CLKB  => I_CLK,
                 DIA   => "0000",                  DIB   => "0000",
                 ENA   => L_WAIT_N,                ENB   => '1',
                 RSTA  => '0',                     RSTB  => '0',
                 WEA   => '0',                     WEB   => '0',
                 DOA   => M_OPC_E(3 downto 0),     DOB   => M_PMD_E(3 downto 0));
```

src/prog_mem.vhd

The first line instantiates a module of type **RAMB4_S4_S4**, which is a dual-port memory module with two 4-bit ports. For a Xilinx FPGA you can used these modules directly by uncommenting the use of the UNISIM library. For functional simulation we have provided a **RAMB4_S4_S4.vhd** component in the test directory. This component emulates the real **RAMB4_S4_S4** as good as needed.

The next lines define the content of each memory module by means of a generic map. The elements of the generic map (like **pe_0_00**, **pe_0_01**, and so forth) define the initial memory content of the instantiated module. **pe_0_00**, **pe_0_01**, .. are themselves defined in **prog_mem_content.vhd** which is included in the library section:

```
    use work.prog_mem_content.all;
```

src/prog_mem.vhd

The process from a C (or C++) source file **hello.c** to the final FPGA is then:

- write, compile, and link **hello.c** (produces **hello.hex**).
- generate **prog_mem_content.vhd** from **hello.hex** (by means of tool **make_mem**, which is provided with this lecture).
- simulate, synthesize and implement the design.
- create a bitmap file.
- flash the FPGA (or serial PROM).

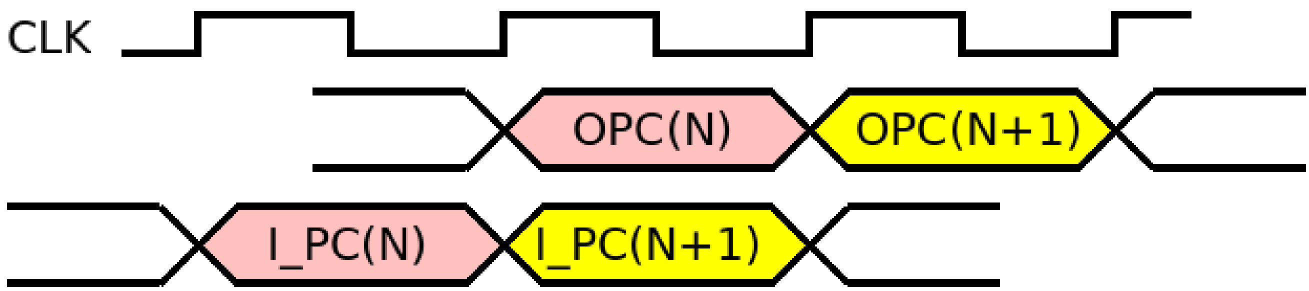
There are other ways of initializing the memory modules, such as updating sections of the bitmap file, but we found the above sequence easier to use.

After the generic map, follows the port map of the memory module. The two addresses **ADDRA** and **ADDRB** of the two ports come from the **PC** and **PM_ADR** inputs as already described.

Both ports are clocked from **CLK**. Since the program memory is read-only, the **DIA** and **DIB** inputs are not used (set to 0000) and **WEA** and **WEB** are 0. **RSTA** and **RSTB** are not used either and are set to 0. **ENA** is used for keeping the **OPC** when the pipeline is stopped, while **ENB** is not used. The memory outputs **DOA** and **DOB** go to the output multiplexers of the two ports.

5.2.3 Delayed PC

Q_PC is **I_PC** delayed by one clock. The program memory is a synchronous memory, which has the consequence that the program memory output **OPC** for a given **I_PC** is always one clock cycle behind as shown in the figure below on the left.



By clocking **I_PC** once, we re-align **Q_PC** and **OPC** as shown on the right:

```
                Q_PC       <= I_PC;
```

src/prog_mem.vhd

5.3 Two Cycle Opcodes

The vast majority of instructions executes in one cycle. Some need two cycles because they involve reading of a synchronous memory. For these signals **WAIT** signal is generated on the first cycle:

```
        -- Two cycle opcodes:
        --
        -- 1001 000d dddd .... - LDS etc.
        -- 1001 0101 0000 1000 - RET
        -- 1001 0101 0001 1000 - RETI
        -- 1001 1001 AAAA Abbb - SBIC
        -- 1001 1011 AAAA Abbb - SBIS
        -- 1111 110r rrrr 0bbb - SBRC
        -- 1111 111r rrrr 0bbb - SBRS
        --
        L_WAIT      <= '0'  when (L_INVALIDATE = '1')
             else '0'  when (I_INTVEC(5)  = '1')
             else L_T0 when ((P_OPC(15 downto   9) = "1001000" )    -- LDS etc.
                         or  (P_OPC(15 downto   8) = "10010101")    -- RET etc.
                         or  ((P_OPC(15 downto 10) = "100110")      -- SBIC, SBIS
                           and P_OPC(8) = '1')
```

```
                          or  (P_OPC(15 downto  10) = "111111"))     -- SBRC, SBRS
             else  '0';
```

src/opc_fetch.vhd

5.4 Interrupts

The opcode fetch stage is also responsible for part of the interrupt handling. Interrupts are generated in the I/O block by setting **INTVEC** to a value with the highest bit set:

```
                                L_INTVEC      <= "101011";       -- _VECTOR(11)
                            end if;
                        elsif (L_TX_INT_ENABLED and not U_TX_BUSY) = '1' then
                            if (L_INTVEC(5) = '0') then     -- no interrupt pending
                                L_INTVEC      <= "101100";       -- _VECTOR(12)
                            end if;
                        else                                -- no interrupt
                            L_INTVEC      <= "000000";
```

src/io.vhd

The highest bit of **INTVEC** indicates that the lower bits contain a valid interrupt number. **INTVEC** proceeds to the cpu core where the upper bit is **and**'ed with the global interrupt enable bit (in the status register):

```
            L_INTVEC_5      <= I_INTVEC(5) and R_INT_ENA;
```

src/cpu_core.vhd

The (possibly modified) **INTVEC** then proceeds to the opcode fetch stage. If the the global interrupt enable bit was set, then the next valid opcode is replaced by an "interrupt opcode":

```
            Q_OPC      <= X"00000000" when (L_INVALIDATE = '1')
                else P_OPC       when (I_INTVEC(5) = '0')
                else (X"000000" & "00" & I_INTVEC);     -- "interrupt opcode"
```

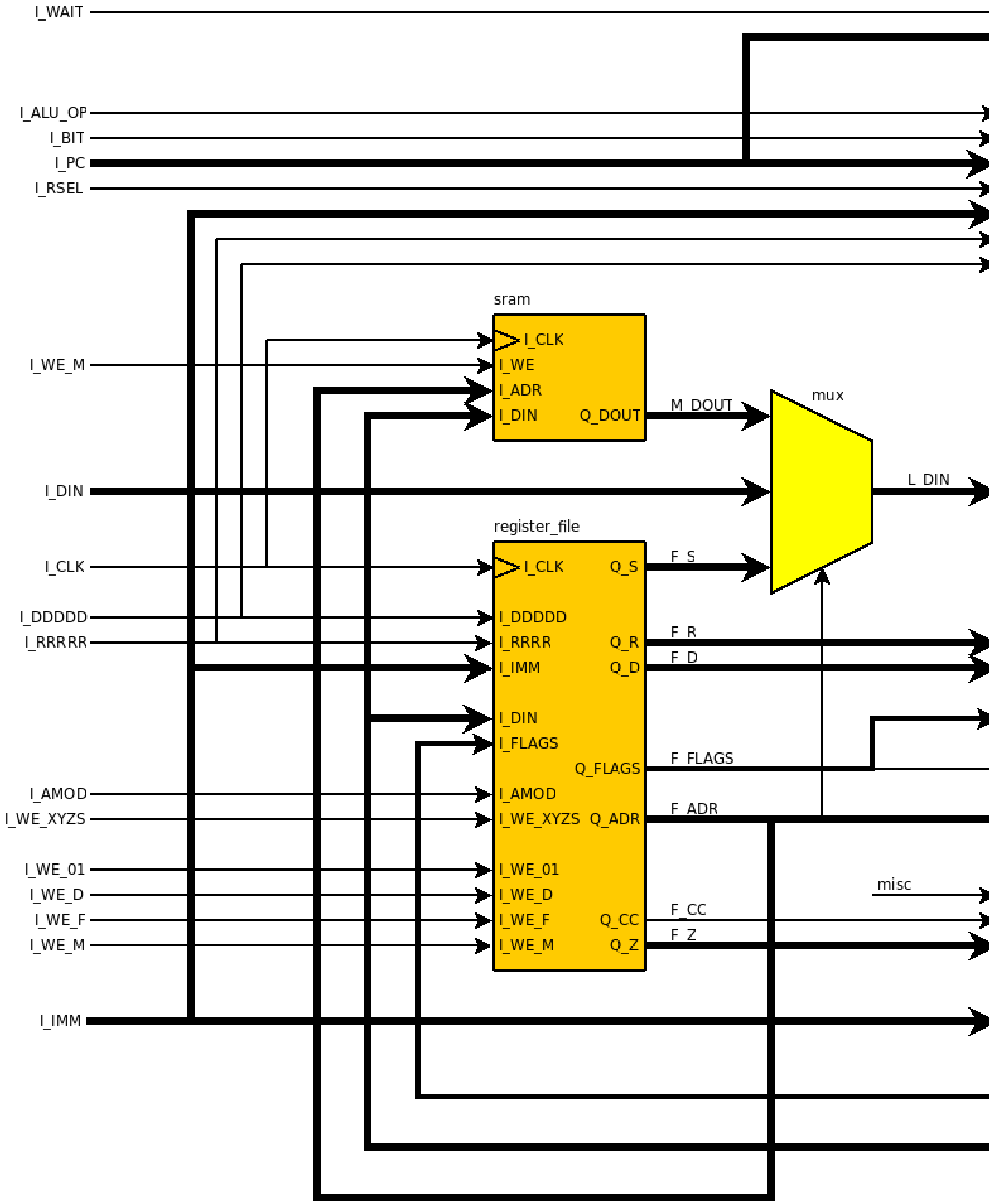
src/opc_fetch.vhd

The interrupt opcode uses a gap after the **NOP** instruction in the opcode set of the AVR CPU. When the interrupt opcode reaches the execution stage then is causes a branch to the location determined by the lower bits of **INTVEC**, pushes the program counter, and clears the interrupt enable bit. This happens a few clock cycles later. In the meantime the opcode fetch stage keeps inserting interrupt instructions into the pipeline. These additional interrupt instructions are being invalidated by the execution stage when the first interrupt instruction reaches the execution stage.

6 DATA PATH

In this lesson we will describe the data path of the CPU. We discuss the basic elements of the data path, but without reference to particular instructions. The implementation of instructions will be discussed in the next lesson. In this lesson we are more interested in the capabilities of the data path.

The data path consists of 3 major components: a register file, an ALU (arithmetic/logic unit), and the data memory:


I_WAIT
I_ALU_OP
I_BIT
I_PC
I_RSEL
sram
I_CLK
I_WE_M
I_WE
I_ADR
I_DIN
Q_DOUT
M_DOUT
mux
I_DIN
L_DIN
register_file
I_CLK
I_CLK
Q_S
F_S
I_DDDDD
I_DDDDD
I_RRRRR
I_RRRR
Q_R
F_R
I_IMM
Q_D
F_D
I_DIN
I_FLAGS
Q_FLAGS
F_FLAGS
I_AMOD
I_AMOD
I_WE_XYZS
I_WE_XYZS
Q_ADR
F_ADR
I_WE_01
I_WE_01
misc
I_WE_D
I_WE_D
I_WE_F
I_WE_F
Q_CC
F_CC
I_WE_M
I_WE_M
Q_Z
F_Z
I_IMM

6.1 Register File

The AVR CPU has 32 general purpose 8-bit registers. Most opcodes use individual 8-bit registers, but some that use a pair of registers. The first register of a register pair is always an even register, while the other register of a pair is the next higher odd register. Instead of using 32 8-bit registers, we use 16 16-bit register pairs. Each register pair consists of two 8-bit registers.

6.1.1 Register Pair

A single register pair is defined as:

```
entity reg_16 is
    port (  I_CLK       : in  std_logic;
            I_D         : in  std_logic_vector (15 downto 0);
            I_WE        : in  std_logic_vector ( 1 downto 0);
            Q           : out std_logic_vector (15 downto 0));
end reg_16;
```

reg_16.vhd

The **Q** output provides the current value of the register pair. There is no need for a read strobe, because (unlike I/O devices) reading the current value of a register pair has no side effects.

The register pair can be written by setting one or both bits of the **WE** input. If both bits are set then the all 16 bits of **D** are written; the low byte to the even register and the higher byte to the odd register of the pair. If only one bit is set then the register corresponding then the bit set in **WE** defines the register to be written (even bit = even register, odd bit = odd register) and the value to be written is in the lower byte of **DIN**:

```
    process(I_CLK)
    begin
        if (rising_edge(I_CLK)) then
            if (I_WE(1) = '1') then
                L(15 downto 8)      <= I_D(15 downto 8);
            end if;
            if (I_WE(0) = '1') then
                L( 7 downto 0)      <= I_D( 7 downto 0);
            end if;
        end if;
    end process;
```

src/reg_16.vhd

6.1.2 The Status Register

The status register is an 8-bit register. This register can be updated by writing to address 0x5F. Primarily it is updated, however, as a side effect of the execution of ALU operations. If, for example, an arithmetic/logic instruction produces a result of 0, then the zero flag (the second bit in the status register) is set. An arithmetic overflow in an ADD instruction causes the carry bit to be set, and so on. The status register is declared as:

```
entity status_reg is
    port (  I_CLK       : in  std_logic;

            I_COND      : in  std_logic_vector ( 3 downto 0);
            I_DIN       : in  std_logic_vector ( 7 downto 0);
            I_FLAGS     : in  std_logic_vector ( 7 downto 0);
            I_WE_F      : in  std_logic;
            I_WE_SR     : in  std_logic;

            Q           : out std_logic_vector ( 7 downto 0);
            Q_CC        : out std_logic);
end status_reg;
```

src/status_reg.vhd

If **WE_FLAGS** is '1' then the status register is updated as a result of an ALU operation; the new value of the status register is provided on the **FLAGS** input which comes from the ALU.

If **WE_SR** is '1' then the status register is updated as a result of an I/O write operation (like **OUT** or **STS**); the new value of the status register is provided on the **DIN** input.

The output **Q** of the status register holds the current value of the register. In addition there is a **CC** output that is '1' when the condition indicated by the **COND** input is fulfilled. This is used for conditional branch instructions. **COND** comes directly from the opcode for a branch instruction (bit 10 of the opcode for the "polarity" and bits 2-0 of the opcode for the bit of the status register that is being tested).

6.1.3 Register File Components

The register file consists of 16 general purpose register pairs **r00** to **r30**, a stack pointer **sp**, and an 8-bit status register **sr**:

```
r00: reg_16 port map(I_CLK => I_CLK, I_WE => L_WE( 1 downto  0), I_D => I_DIN, Q => R_R00);
r02: reg_16 port map(I_CLK => I_CLK, I_WE => L_WE( 3 downto  2), I_D => I_DIN, Q => R_R02);
r04: reg_16 port map(I_CLK => I_CLK, I_WE => L_WE( 5 downto  4), I_D => I_DIN, Q => R_R04);
r06: reg_16 port map(I_CLK => I_CLK, I_WE => L_WE( 7 downto  6), I_D => I_DIN, Q => R_R06);
r08: reg_16 port map(I_CLK => I_CLK, I_WE => L_WE( 9 downto  8), I_D => I_DIN, Q => R_R08);
r10: reg_16 port map(I_CLK => I_CLK, I_WE => L_WE(11 downto 10), I_D => I_DIN, Q => R_R10);
```

```
        r12: reg_16 port map(I_CLK => I_CLK, I_WE => L_WE(13 downto 12), I_D => I_DIN, Q => R
        r14: reg_16 port map(I_CLK => I_CLK, I_WE => L_WE(15 downto 14), I_D => I_DIN, Q => R
        r16: reg_16 port map(I_CLK => I_CLK, I_WE => L_WE(17 downto 16), I_D => I_DIN, Q => R
        r18: reg_16 port map(I_CLK => I_CLK, I_WE => L_WE(19 downto 18), I_D => I_DIN, Q => R
        r20: reg_16 port map(I_CLK => I_CLK, I_WE => L_WE(21 downto 20), I_D => I_DIN, Q => R
        r22: reg_16 port map(I_CLK => I_CLK, I_WE => L_WE(23 downto 22), I_D => I_DIN, Q => R
        r24: reg_16 port map(I_CLK => I_CLK, I_WE => L_WE(25 downto 24), I_D => I_DIN, Q => R
        r26: reg_16 port map(I_CLK => I_CLK, I_WE => L_WE(27 downto 26), I_D => L_DX,  Q => R
        r28: reg_16 port map(I_CLK => I_CLK, I_WE => L_WE(29 downto 28), I_D => L_DY,  Q => R
        r30: reg_16 port map(I_CLK => I_CLK, I_WE => L_WE(31 downto 30), I_D => L_DZ,  Q => R
```

src/register_file.vhd

```
        sp:  reg_16 port map(I_CLK => I_CLK, I_WE => L_WE_SP,            I_D => L_DSP, Q => R
```

src/register_file.vhd

```
        sr: status_reg
        port map(   I_CLK       => I_CLK,
                    I_COND      => I_COND,
                    I_DIN       => I_DIN(7 downto 0),
                    I_FLAGS     => I_FLAGS,
                    I_WE_F      => I_WE_F,
                    I_WE_SR     => L_WE_SR,
                    Q           => S_FLAGS,
                    Q_CC        => Q_CC);
```

src/register_file.vhd

Each register pair drives a 16-bit signal according to the (even) number of the register pair in the register file:

```
    signal R_R00            : std_logic_vector(15 downto 0);
    signal R_R02            : std_logic_vector(15 downto 0);
    signal R_R04            : std_logic_vector(15 downto 0);
    signal R_R06            : std_logic_vector(15 downto 0);
    signal R_R08            : std_logic_vector(15 downto 0);
    signal R_R10            : std_logic_vector(15 downto 0);
    signal R_R12            : std_logic_vector(15 downto 0);
    signal R_R14            : std_logic_vector(15 downto 0);
    signal R_R16            : std_logic_vector(15 downto 0);
    signal R_R18            : std_logic_vector(15 downto 0);
    signal R_R20            : std_logic_vector(15 downto 0);
    signal R_R22            : std_logic_vector(15 downto 0);
    signal R_R24            : std_logic_vector(15 downto 0);
    signal R_R26            : std_logic_vector(15 downto 0);
    signal R_R28            : std_logic_vector(15 downto 0);
    signal R_R30            : std_logic_vector(15 downto 0);
    signal R_SP             : std_logic_vector(15 downto 0);    -- stack pointer
```

src/register_file.vhd

6.1.4 Addressing of General Purpose Registers

We address individual general purpose registers by a 5-bit value. Normally an opcode using an individual general purpose 8-bit register has a 5 bit field which is the address of the register. The opcode decoder transfers this field to its **DDDDD** or **RRRRR** output. For some opcodes not all 32 registers can be used, but only 16 (e.g. **ANDI**) or 8 (e.g. **MUL**). In these cases the register field in the opcode is smaller and the opcode decoder fills in the missing bits. Some opcodes imply particular registers (e.g. some **LPM** variant), and again the opcode decoder fills in the implied register number.

An opcode may address no, one, two, or three registers or pairs. If one register is addressed, then the number of that register is encoded in the **DDDDD** signal.
If two (or more) registers are used, then one (normally the destination register) is encoded in the **DDDDD** signal and the other (source) is encoded in the **RRRRR** signal. Opcodes with 3 registers (e.g. MUL) use an implied destination register pair (register pair 0) and two source registers encoded in the **DDDDD** and **RRRRR** signals.

6.1.5 Addressing of General Purpose Register Pairs

We address register pairs by addressing the even register of the pair. The address of a register pair is therefore a 5-bit value with the lowest bit cleared. The opcode normally only has a 4-bit field for a register pair and the lowest (cleared) bit is filled in by the opcode decoder. Like for individual registers it can happen that not all 16 register pairs can be addresses (e.g. **ADIW**). This is handles in the same way as for individual registers.

In the AVR context, the register pairs **R26**, **R28**, and **R30** are also called (pointer registers) **X**, **Y**, and **Z** respectively.

6.1.6 Requirements on the Register File

If we go through the opcodes of the AVR CPU, then we see the capabilities that the register file must provide for general purpose registers (or register pairs):

Capability	Opcode (example)
Read one register, read/write another register	ADD Rd, Rr
Write one register, read/write another register	LD Rd, (X+)
Write one register, read another register	LD Rd, (X)
Read/write one register	ASR Rd
Read one register, read another register	CMP Rd, Rr
Read one register, read another register	LD Rd, Rr
Read one register	IN Rd, A
Write one register	OUT A, Rr

6.1.7 Reading Registers or Register Pairs

There are 4 cases:

- Read register or register pair addresses by **DDDDD**.
- Read register or register pair addresses by **RRRRR**.
- Read register addressed by the current I/O address.
- Read **X**, **Y**, or **Z** pointer implied by the addressing mode **AMOD**.

- Read the **Z** pointer implied by the instruction (**IJMP** or **ICALL**).

Some of these cases can happen simultaneously. For example the **ICALL** instruction reads the **Z** register (the target address of the call) while it pushed the current PC onto the stack. Likewise, **ST** may need the **X**, **Y**, or **Z** for address calculations and a general purpose register that is to be stored in memory. For this reason we provide 5 different outputs in the register file. These outputs are addressing the general purpose registers differently (and they can be used in parallel):

- **Q_D** is the content of the register addressed by **DDDDD**.
- **Q_R** is the content of the register pair addressed by **RRRR**.
- **Q_S** is the content of the register addressed by **ADR**.
- **Q_ADR** is an address defined by **AMOD** (and may use **X**, **Y**, or **Z**).
- **Q_X** is the content of the Z register.

Q_D is one of the register pair signals as defined by **DDDDD**. We read the entire pair; the selection of the even/odd register within the pair is done later in the ALU based on **DDDDD(0)**:

```
    process(R_R00, R_R02, R_R04, R_R06, R_R08, R_R10, R_R12, R_R14,
            R_R16, R_R18, R_R20, R_R22, R_R24, R_R26, R_R28, R_R30,
            I_DDDDD(4 downto 1))
    begin
        case I_DDDDD(4 downto 1) is
            when "0000" => Q_D      <= R_R00;
            when "0001" => Q_D      <= R_R02;
            when "0010" => Q_D      <= R_R04;
            when "0011" => Q_D      <= R_R06;
            when "0100" => Q_D      <= R_R08;
            when "0101" => Q_D      <= R_R10;
            when "0110" => Q_D      <= R_R12;
            when "0111" => Q_D      <= R_R14;
            when "1000" => Q_D      <= R_R16;
            when "1001" => Q_D      <= R_R18;
            when "1010" => Q_D      <= R_R20;
            when "1011" => Q_D      <= R_R22;
            when "1100" => Q_D      <= R_R24;
            when "1101" => Q_D      <= R_R26;
            when "1110" => Q_D      <= R_R28;
            when others => Q_D      <= R_R30;
        end case;
    end process;
```

src/register_file.vhd

Q_R is one of the register pair signals as defined by **RRRR**:

```
    process(R_R00, R_R02, R_R04,  R_R06, R_R08, R_R10, R_R12, R_R14,
            R_R16, R_R18, R_R20, R_R22, R_R24, R_R26, R_R28, R_R30, I_RRRR)
    begin
```

```
            case I_RRRR is
                when "0000" => Q_R      <= R_R00;
                when "0001" => Q_R      <= R_R02;
                when "0010" => Q_R      <= R_R04;
                when "0011" => Q_R      <= R_R06;
                when "0100" => Q_R      <= R_R08;
                when "0101" => Q_R      <= R_R10;
                when "0110" => Q_R      <= R_R12;
                when "0111" => Q_R      <= R_R14;
                when "1000" => Q_R      <= R_R16;
                when "1001" => Q_R      <= R_R18;
                when "1010" => Q_R      <= R_R20;
                when "1011" => Q_R      <= R_R22;
                when "1100" => Q_R      <= R_R24;
                when "1101" => Q_R      <= R_R26;
                when "1110" => Q_R      <= R_R28;
                when others => Q_R      <= R_R30;
            end case;
        end process;
```

src/register_file.vhd

The general purpose registers, but also the stack pointer and the status register, are mapped into the data memory space:

Address	Purpose
0x00 - 0x1F	general purpose CPU registers.
0x20 - 0x5C	miscellaneous I/O registers.
0x5D	stack pointer low
0x5E	stack pointer high
0x5F	status register
0x60 - 0xFFFF	data memory

If an address corresponding to a register in the register file (i.e. a general purpose register, the stack pointer, or the status register is read, then the register shall be returned.
For example, LD Rd, R22 shall give the same result as LDS Rd, 22.

The 8-bit **Q_S** output contains the register addresses by **ADR**:

```
        process(R_R00, R_R02, R_R04, R_R06, R_R08, R_R10, R_R12, R_R14,
                R_R16, R_R18, R_R20, R_R22, R_R24, R_R26, R_R28, R_R30,
                R_SP, S_FLAGS, L_ADR(6 downto 1))
        begin
            case L_ADR(6 downto 1) is
                when "000000" => L_S      <= R_R00;
                when "000001" => L_S      <= R_R02;
                when "000010" => L_S      <= R_R04;
                when "000011" => L_S      <= R_R06;
                when "000100" => L_S      <= R_R08;
                when "000101" => L_S      <= R_R10;
                when "000110" => L_S      <= R_R12;
                when "000111" => L_S      <= R_R14;
```

```
                when "001000" => L_S      <= R_R16;
                when "001001" => L_S      <= R_R18;
                when "001010" => L_S      <= R_R20;
                when "001011" => L_S      <= R_R22;
                when "001100" => L_S      <= R_R24;
                when "001101" => L_S      <= R_R26;
                when "001110" => L_S      <= R_R28;
                when "001111" => L_S      <= R_R30;
                when "101111" => L_S      <= R_SP ( 7 downto 0) & X"00";     -- SPL
                when others   => L_S      <= S_FLAGS & R_SP (15 downto 8);   -- SR/SPH
            end case;
        end process;
```

src/register_file.vhd

6.1.8 Writing Registers or Register Pairs

In order to write a register, we need to select the proper input (data source) and the proper **WE** signal. For most registers, the only possible data source is **DIN** which comes straight from the ALU. The pointer register pairs **X**, **Y**, and **Z**, however, can also be changed as a side effect of the post-increment (**X+**, **Y+**, **Z+**) and pre-decrement (**-X**, **-Y**, **-Z**) addressing modes of the **LDS** and **STS** instructions. The addressing modes are discussed in more detail in the next chapter; here it suffices to note that the **X**, **Y**, and #Z #registers get there data from **DX**, **DY**, and **DZ**, respectively rather than from **DIN**.

There is a total of 4 cases where general purpose registers are written. Three of these cases that are applicable to all general purpose registers and one case collects special cases for particular registers (the register numbers are then implied).

We compute a 32 bit write enable signal for each of the four cases and **OR** them together.

The first case is a write to an 8-bit register addressed by **DDDDD**. For this case we create the signal **WE_D**:

```
        L_WE_D( 0)      <= I_WE_D(0) when (I_DDDDD = "00000") else '0';
        L_WE_D( 1)      <= I_WE_D(0) when (I_DDDDD = "00001") else '0';
        L_WE_D( 2)      <= I_WE_D(0) when (I_DDDDD = "00010") else '0';
        L_WE_D( 3)      <= I_WE_D(0) when (I_DDDDD = "00011") else '0';
        L_WE_D( 4)      <= I_WE_D(0) when (I_DDDDD = "00100") else '0';
        L_WE_D( 5)      <= I_WE_D(0) when (I_DDDDD = "00101") else '0';
        L_WE_D( 6)      <= I_WE_D(0) when (I_DDDDD = "00110") else '0';
        L_WE_D( 7)      <= I_WE_D(0) when (I_DDDDD = "00111") else '0';
        L_WE_D( 8)      <= I_WE_D(0) when (I_DDDDD = "01000") else '0';
        L_WE_D( 9)      <= I_WE_D(0) when (I_DDDDD = "01001") else '0';
        L_WE_D(10)      <= I_WE_D(0) when (I_DDDDD = "01010") else '0';
        L_WE_D(11)      <= I_WE_D(0) when (I_DDDDD = "01011") else '0';
        L_WE_D(12)      <= I_WE_D(0) when (I_DDDDD = "01100") else '0';
        L_WE_D(13)      <= I_WE_D(0) when (I_DDDDD = "01101") else '0';
        L_WE_D(14)      <= I_WE_D(0) when (I_DDDDD = "01110") else '0';
        L_WE_D(15)      <= I_WE_D(0) when (I_DDDDD = "01111") else '0';
        L_WE_D(16)      <= I_WE_D(0) when (I_DDDDD = "10000") else '0';
        L_WE_D(17)      <= I_WE_D(0) when (I_DDDDD = "10001") else '0';
        L_WE_D(18)      <= I_WE_D(0) when (I_DDDDD = "10010") else '0';
```

```
        L_WE_D(19)      <= I_WE_D(0) when (I_DDDDD = "10011") else '0';
        L_WE_D(20)      <= I_WE_D(0) when (I_DDDDD = "10100") else '0';
        L_WE_D(21)      <= I_WE_D(0) when (I_DDDDD = "10101") else '0';
        L_WE_D(22)      <= I_WE_D(0) when (I_DDDDD = "10110") else '0';
        L_WE_D(23)      <= I_WE_D(0) when (I_DDDDD = "10111") else '0';
        L_WE_D(24)      <= I_WE_D(0) when (I_DDDDD = "11000") else '0';
        L_WE_D(25)      <= I_WE_D(0) when (I_DDDDD = "11001") else '0';
        L_WE_D(26)      <= I_WE_D(0) when (I_DDDDD = "11010") else '0';
        L_WE_D(27)      <= I_WE_D(0) when (I_DDDDD = "11011") else '0';
        L_WE_D(28)      <= I_WE_D(0) when (I_DDDDD = "11100") else '0';
        L_WE_D(29)      <= I_WE_D(0) when (I_DDDDD = "11101") else '0';
        L_WE_D(30)      <= I_WE_D(0) when (I_DDDDD = "11110") else '0';
        L_WE_D(31)      <= I_WE_D(0) when (I_DDDDD = "11111") else '0';
```

src/register_file.vhd

The second case is a write to a 16-bit register pair addressed by **DDDD** (**DDDD** is the four upper bits of **DDDDD**). For this case we create signal **WE_DD**:

```
        L_DDDD      <= I_DDDDD(4 downto 1);
        L_WE_D2      <= I_WE_D(1) & I_WE_D(1);
        L_WE_DD( 1 downto  0)      <= L_WE_D2 when (L_DDDD = "0000") else "00";
        L_WE_DD( 3 downto  2)      <= L_WE_D2 when (L_DDDD = "0001") else "00";
        L_WE_DD( 5 downto  4)      <= L_WE_D2 when (L_DDDD = "0010") else "00";
        L_WE_DD( 7 downto  6)      <= L_WE_D2 when (L_DDDD = "0011") else "00";
        L_WE_DD( 9 downto  8)      <= L_WE_D2 when (L_DDDD = "0100") else "00";
        L_WE_DD(11 downto 10)      <= L_WE_D2 when (L_DDDD = "0101") else "00";
        L_WE_DD(13 downto 12)      <= L_WE_D2 when (L_DDDD = "0110") else "00";
        L_WE_DD(15 downto 14)      <= L_WE_D2 when (L_DDDD = "0111") else "00";
        L_WE_DD(17 downto 16)      <= L_WE_D2 when (L_DDDD = "1000") else "00";
        L_WE_DD(19 downto 18)      <= L_WE_D2 when (L_DDDD = "1001") else "00";
        L_WE_DD(21 downto 20)      <= L_WE_D2 when (L_DDDD = "1010") else "00";
        L_WE_DD(23 downto 22)      <= L_WE_D2 when (L_DDDD = "1011") else "00";
        L_WE_DD(25 downto 24)      <= L_WE_D2 when (L_DDDD = "1100") else "00";
        L_WE_DD(27 downto 26)      <= L_WE_D2 when (L_DDDD = "1101") else "00";
        L_WE_DD(29 downto 28)      <= L_WE_D2 when (L_DDDD = "1110") else "00";
        L_WE_DD(31 downto 30)      <= L_WE_D2 when (L_DDDD = "1111") else "00";
```

src/register_file.vhd

The third case is writing to the memory mapped I/O space of the general purpose registers. It is similar to the first case, but now we select the register by **ADR** instead of **DDDDD**. When reading from the I/O mapped register above we did not check if **ADR** was completely correct (and different addresses could read the same register. This was OK, since some multiplexer somewhere else would discard the value read for addresses outside the range from 0x00 to 0x1F. When writing we have to be more careful and check the range by means of **WE_A**. For the third case we use signal **WE_IO**:

```
        L_WE_IO( 0)      <= L_WE_A when (L_ADR(4 downto 0) = "00000") else '0';
        L_WE_IO( 1)      <= L_WE_A when (L_ADR(4 downto 0) = "00001") else '0';
        L_WE_IO( 2)      <= L_WE_A when (L_ADR(4 downto 0) = "00010") else '0';
        L_WE_IO( 3)      <= L_WE_A when (L_ADR(4 downto 0) = "00011") else '0';
        L_WE_IO( 4)      <= L_WE_A when (L_ADR(4 downto 0) = "00100") else '0';
        L_WE_IO( 5)      <= L_WE_A when (L_ADR(4 downto 0) = "00101") else '0';
        L_WE_IO( 6)      <= L_WE_A when (L_ADR(4 downto 0) = "00110") else '0';
        L_WE_IO( 7)      <= L_WE_A when (L_ADR(4 downto 0) = "00111") else '0';
        L_WE_IO( 8)      <= L_WE_A when (L_ADR(4 downto 0) = "01000") else '0';
        L_WE_IO( 9)      <= L_WE_A when (L_ADR(4 downto 0) = "01001") else '0';
        L_WE_IO(10)      <= L_WE_A when (L_ADR(4 downto 0) = "01010") else '0';
        L_WE_IO(11)      <= L_WE_A when (L_ADR(4 downto 0) = "01011") else '0';
        L_WE_IO(12)      <= L_WE_A when (L_ADR(4 downto 0) = "01100") else '0';
        L_WE_IO(13)      <= L_WE_A when (L_ADR(4 downto 0) = "01101") else '0';
        L_WE_IO(14)      <= L_WE_A when (L_ADR(4 downto 0) = "01110") else '0';
        L_WE_IO(15)      <= L_WE_A when (L_ADR(4 downto 0) = "01111") else '0';
        L_WE_IO(16)      <= L_WE_A when (L_ADR(4 downto 0) = "10000") else '0';
        L_WE_IO(17)      <= L_WE_A when (L_ADR(4 downto 0) = "10001") else '0';
        L_WE_IO(18)      <= L_WE_A when (L_ADR(4 downto 0) = "10010") else '0';
        L_WE_IO(19)      <= L_WE_A when (L_ADR(4 downto 0) = "10011") else '0';
        L_WE_IO(20)      <= L_WE_A when (L_ADR(4 downto 0) = "10100") else '0';
        L_WE_IO(21)      <= L_WE_A when (L_ADR(4 downto 0) = "10101") else '0';
        L_WE_IO(22)      <= L_WE_A when (L_ADR(4 downto 0) = "10110") else '0';
        L_WE_IO(23)      <= L_WE_A when (L_ADR(4 downto 0) = "10111") else '0';
        L_WE_IO(24)      <= L_WE_A when (L_ADR(4 downto 0) = "11000") else '0';
        L_WE_IO(25)      <= L_WE_A when (L_ADR(4 downto 0) = "11001") else '0';
        L_WE_IO(26)      <= L_WE_A when (L_ADR(4 downto 0) = "11010") else '0';
        L_WE_IO(27)      <= L_WE_A when (L_ADR(4 downto 0) = "11011") else '0';
        L_WE_IO(28)      <= L_WE_A when (L_ADR(4 downto 0) = "11100") else '0';
        L_WE_IO(29)      <= L_WE_A when (L_ADR(4 downto 0) = "11101") else '0';
        L_WE_IO(30)      <= L_WE_A when (L_ADR(4 downto 0) = "11110") else '0';
        L_WE_IO(31)      <= L_WE_A when (L_ADR(4 downto 0) = "11111") else '0';
```

src/register_file.vhd

The last case for writing is handled by **WE_MISC**. The various multiplication opcodes write their result to register pair 0; this case is indicated the the **WE_01** input. Then we have the pre-decrement and post-increment addressing modes that update the **X**, **Y**, or **Z** register:

```
        L_WE_X     <= I_WE_XYZS when (I_AMOD(3 downto 0) = AM_WX) else '0';
        L_WE_Y     <= I_WE_XYZS when (I_AMOD(3 downto 0) = AM_WY) else '0';
        L_WE_Z     <= I_WE_XYZS when (I_AMOD(3 downto 0) = AM_WZ) else '0';
        L_WE_MISC     <= L_WE_Z & L_WE_Z &      -- -Z and Z+ address modes  r30
                    L_WE_Y & L_WE_Y &      -- -Y and Y+ address modes  r28
                    L_WE_X & L_WE_X &      -- -X and X+ address modes  r26
                    X"000000" &            -- never                    r24 - r02
                    I_WE_01 & I_WE_01;     -- multiplication result    r00
```

src/register_file.vhd

The final **WE** signal is then computed by **or**'ing the four cases above:

```
        L_WE      <= L_WE_D or L_WE_DD or L_WE_IO or L_WE_MISC;
```

src/register_file.vhd

The stack pointer can be updated from two sources: from **DIN** as a memory mapped I/O or implicitly from **XYZS** by addressing modes (e.g. for **CALL**, **RET**, **PUSH**, and **POP** instructions) that write to the **SP** (**AM_WS**).

```
        L_DSP      <= L_XYZS when (I_AMOD(3 downto 0) = AM_WS) else I_DIN;
```

src/register_file.vhd

The status register can be written as memory mapped I/O from the **DIN** input or from the **FLAGS** input (from the ALU). The **WE_SR** input (for memory mapped I/O) and the **WE_FLAGS** input (for flags set as side effect of ALU operations) control from where the new value comes:

```
        L_WE_SR         <= I_WE_M when (L_ADR = X"005F") else '0';
```

src/register_file.vhd

```
                    I_DIN       => I_DIN(7 downto 0),
                    I_FLAGS     => I_FLAGS,
                    I_WE_F      => I_WE_F,
```

src/register_file.vhd

6.1.9 Addressing Modes

The CPU provides a number of addressing modes. An addressing mode is a way to compute an address. The address specifies a location in the program memory, the data memory, the I/O memory, or some general purpose register. Computing an address can have side effects such as incrementing or decrementing a pointer register.

The addressing mode to be used (if any) is encoded in the **AMOD** signal. The **AMOD** signal consists of two sub-fields: the address source and the address offset.

There are 5 possible address sources:

```
        constant AS_SP  : std_logic_vector(2 downto 0) := "000";     -- SP
        constant AS_Z   : std_logic_vector(2 downto 0) := "001";     -- Z
        constant AS_Y   : std_logic_vector(2 downto 0) := "010";     -- Y
        constant AS_X   : std_logic_vector(2 downto 0) := "011";     -- X
        constant AS_IMM : std_logic_vector(2 downto 0) := "100";     -- IMM
```

src/common.vhd

The address sources **AS_SP**, **AS_X**, **AS_Y**, and **AS_Z** are the stack pointer, the **X** register pair, the **Y** register pair, or the **Z** register pair. The **AS_IMM** source is the **IMM** input (which was computed from the opcode in the opcode decoder).

There are 6 different address offsets. An address offset can imply a side effect like incrementing or decrementing the address source. The lowest bit of the address offset indicates whether a side effect is intended or not:

```
        constant AO_0   : std_logic_vector(5 downto 3) := "000";     -- as is
        constant AO_Q   : std_logic_vector(5 downto 3) := "010";     -- +q
        constant AO_i   : std_logic_vector(5 downto 3) := "001";     -- +1
        constant AO_ii  : std_logic_vector(5 downto 3) := "011";     -- +2
        constant AO_d   : std_logic_vector(5 downto 3) := "101";     -- -1
        constant AO_dd  : std_logic_vector(5 downto 3) := "111";     -- -2
```

src/common.vhd

The address offset **AO_0** does nothing; the address source is not modified. Address offset **AO_Q** adds some constant **q** to the address source; the constant **q** is provided on the **IMM** input (thus derived from the opcode). Address offsets **AO_i** resp. **AO_ii** increment the address source after the operation by 1 resp. 2 bytes. The address computed is the address source. Address offsets **AO_d** resp. **AO_dd** decrement the address source before the operation by 1 resp. 2 bytes. The address computed is the address source minus 1 or 2.

The constants **AM_WX**, **AM_WY**, **AM_WZ**, and **AM_WS** respectively indicate if the **X**, **Y**, **Z**, or **SP** registers will be updated and are used to decode the **WE_XYZS** signal to the register concerned and to select the proper inputs:

```
        L_WE_X      <= I_WE_XYZS when (I_AMOD(3 downto 0) = AM_WX) else '0';
        L_WE_Y      <= I_WE_XYZS when (I_AMOD(3 downto 0) = AM_WY) else '0';
        L_WE_Z      <= I_WE_XYZS when (I_AMOD(3 downto 0) = AM_WZ) else '0';
        L_WE_MISC      <= L_WE_Z & L_WE_Z &      -- -Z and Z+ address modes  r30
                      L_WE_Y & L_WE_Y &      -- -Y and Y+ address modes  r28
```

```
                       L_WE_X & L_WE_X &      -- -X and X+ address modes  r26
```

src/register_file.vhd

```
    L_DX       <= L_XYZS when (L_WE_MISC(26) = '1')        else I_DIN;
    L_DY       <= L_XYZS when (L_WE_MISC(28) = '1')        else I_DIN;
    L_DZ       <= L_XYZS when (L_WE_MISC(30) = '1')        else I_DIN;
    L_DSP      <= L_XYZS when (I_AMOD(3 downto 0) = AM_WS) else I_DIN;
```

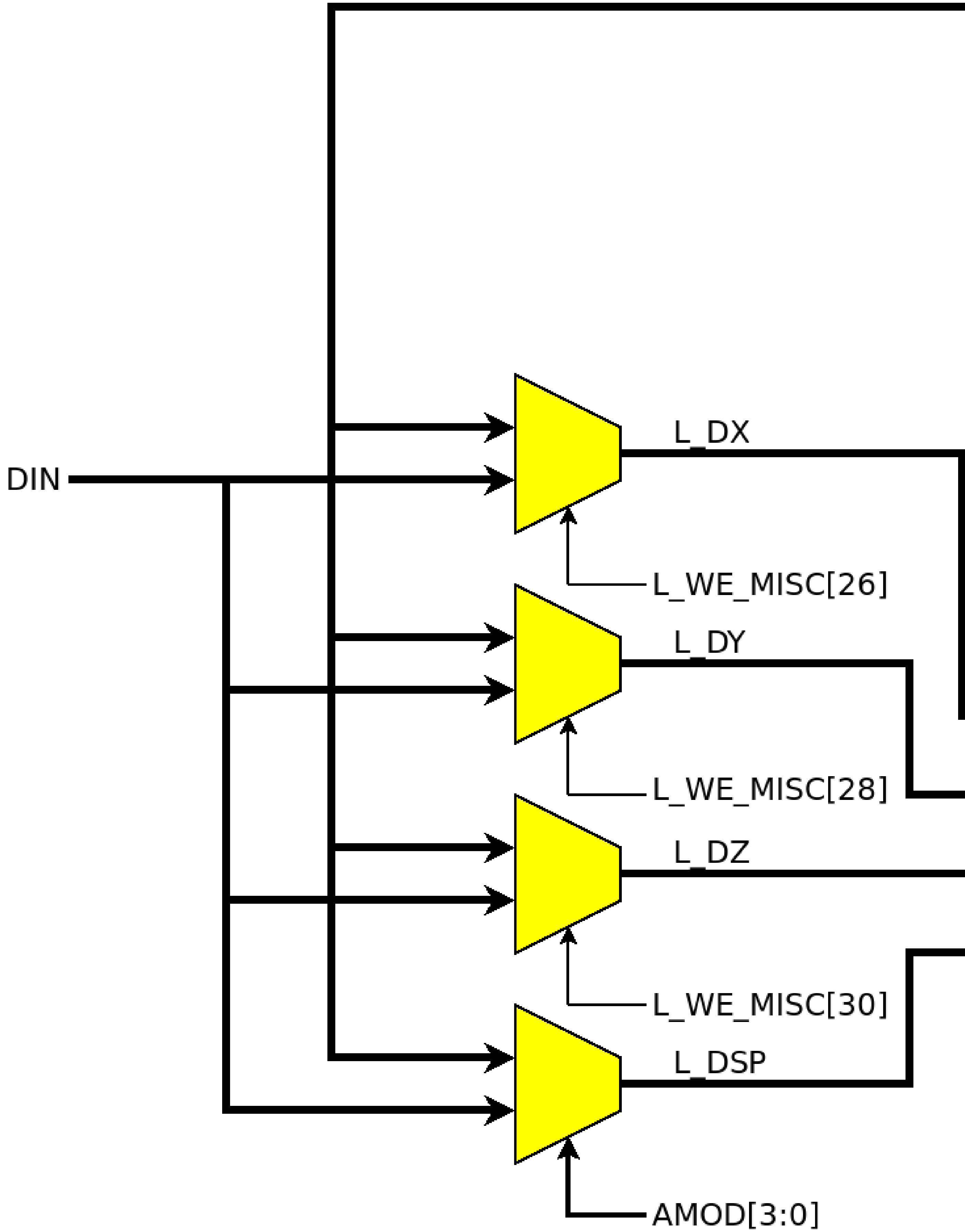
src/register_file.vhd

Not all combinations of address source and address offset occur; only the following combinations are needed:

```
    constant AMOD_ABS : std_logic_vector(5 downto 0) := AO_0  & AS_IMM; -- IMM
    constant AMOD_X   : std_logic_vector(5 downto 0) := AO_0  & AS_X;   -- (X)
    constant AMOD_Xq  : std_logic_vector(5 downto 0) := AO_Q  & AS_X;   -- (X+q)
    constant AMOD_Xi  : std_logic_vector(5 downto 0) := AO_i  & AS_X;   -- (X++)
    constant AMOD_dX  : std_logic_vector(5 downto 0) := AO_d  & AS_X;   -- (--X)
    constant AMOD_Y   : std_logic_vector(5 downto 0) := AO_0  & AS_Y;   -- (Y)
    constant AMOD_Yq  : std_logic_vector(5 downto 0) := AO_Q  & AS_Y;   -- (Y+q)
    constant AMOD_Yi  : std_logic_vector(5 downto 0) := AO_i  & AS_Y;   -- (Y++)
    constant AMOD_dY  : std_logic_vector(5 downto 0) := AO_d  & AS_Y;   -- (--Y)
    constant AMOD_Z   : std_logic_vector(5 downto 0) := AO_0  & AS_Z;   -- (Z)
    constant AMOD_Zq  : std_logic_vector(5 downto 0) := AO_Q  & AS_Z;   -- (Z+q)
    constant AMOD_Zi  : std_logic_vector(5 downto 0) := AO_i  & AS_Z;   -- (Z++)
    constant AMOD_dZ  : std_logic_vector(5 downto 0) := AO_d  & AS_Z;   -- (--Z)
    constant AMOD_SPi : std_logic_vector(5 downto 0) := AO_i  & AS_SP;  -- (SP++)
    constant AMOD_SPii: std_logic_vector(5 downto 0) := AO_ii & AS_SP;  -- (SP++)
    constant AMOD_dSP : std_logic_vector(5 downto 0) := AO_d  & AS_SP;  -- (--SP)
    constant AMOD_ddSP: std_logic_vector(5 downto 0) := AO_dd & AS_SP;  -- (--SP)
```

src/common.vhd

The following figure shows the computation of addresses:


DIN
L_DX
L_WE_MISC[26]
L_DY
L_WE_MISC[28]
L_DZ
L_WE_MISC[30]
L_DSP
AMOD[3:0]

6.2 Data memory

The data memory is conceptually an 8-bit memory. However, some instructions (e.g. CALL, RET) write two bytes to consecutive memory locations. We do the same trick as for the program memory and divide the data memory into an even half and an odd half. The only new thing is a multiplexer at the input:

```
        L_DIN_E      <= I_DIN( 7 downto 0) when (I_ADR(0) = '0') else I_DIN(15 downto 8);
        L_DIN_O      <= I_DIN( 7 downto 0) when (I_ADR(0) = '1') else I_DIN(15 downto 8);
```

src/data_mem.vhd

The multiplexer is needed because the data memory is a read/write memory while the program memory was read-only. The multiplexer swaps the upper and lower bytes of **DIN** when writing to odd addresses.

6.3 Arithmetic/Logic Unit (ALU)

The most obvious component of a CPU is the ALU where all arithmetic and logic operations are computed. We do a little trick here and implement the data move instructions (**MOV**, **LD**, **ST**, etc.) as ALU operations that simply moves the data source to the output of the ALU. The data move instructions can use the same data paths as the arithmetic and logic instructions.

If we look at the instructions set of the CPU then we see that a number of instructions are quite similar. We use these similarities to reduce the number of different instructions that need to be implemented in the ALU.

- Some instructions have 8-bit and 16-bit variants (e.g. **ADD** and **ADIW**).
- Some instructions have immediate variants (e.g. **CMP** and **CMPI**).
- Some instructions differ only in whether they update the destination register or not (e.g. **CMP** and **SUB**).

The ALU is a completely combinational circuit and therefore it has no clock input. We can divide the ALU into a number of blocks that are explained in the following.

6.3.1 **D** Input Multiplexing.

We have seen earlier that the **D** input of the ALU is the output of the register pair addressed by **DDDDD[4:1]** and that the **D0** input of the ALU is **DDDDD[0]**:

```
                        Q_D         => F_D,
```

src/data_path.vhd

```
                     I_D          => F_D,
                     I_D0         => I_DDDDD(0),
```

src/data_path.vhd

If **D0** is zero, then the lower byte of the ALU operation comes from the even register regardless of the size (8-bit or 16-bit) of the operation. If **D0** is odd, then the lower byte of the ALU operation comes from the odd register of the pair (and must be an 8-bit operation since register pairs always have the lowest bit of **DDDDD** cleared.

The upper byte of the operation (if any) is always the odd register of the pair.

We can therefore compute the lower byte, called **D8**, from **D** and **D0**:

```
        L_D8        <= I_D(15 downto 8) when (I_D0 = '1') else I_D(7 downto 0);
```

src/alu.vhd

6.3.2 **R** and **IMM** Input Multiplexing.

Multiplexing of the **R** input works like multiplexing of the **D** input. Some opcodes can have immediate operand instead of a register addressed by **RRRRR**. We compute the signal **R8** for opcodes that cannot have an immediate operand, and **RI8** for opcodes that can have an immediate operand.

This is some fine tuning of the design: the **MULT** opcodes can take a while to compute but cannot have an immediate operand. It makes therefore sense to have a path from the register addressed by **RRRRR** to the multiplier and to put the register/immediate multiplexer outside that critical path through the ALU.

```
        L_R8        <= I_R(15 downto 8) when (I_R0 = '1') else I_R(7 downto 0);
        L_RI8        <= I_IMM           when (I_RSEL = RS_IMM) else L_R8;
```

src/alu.vhd

6.3.3 Arithmetic and Logic Functions

The first step in the computation of the arithmetic and logic functions is to compute a number of helper values. The reason for computing them beforehand is that we need these values several times, either for different but similar opcodes (e.g. **CMP** and **SUB**) but also for the result and for the flags of the same opcode.

```
        L_ADIW_D        <= I_D + ("0000000000" & I_IMM(5 downto 0));
        L_SBIW_D        <= I_D - ("0000000000" & I_IMM(5 downto 0));
        L_ADD_DR        <= L_D8 + L_RI8;
        L_ADC_DR        <= L_ADD_DR + ("0000000" & I_FLAGS(0));
        L_ASR_D         <= L_D8(7) & L_D8(7 downto 1);
        L_AND_DR        <= L_D8 and L_RI8;
        L_DEC_D         <= L_D8 - X"01";
        L_INC_D         <= L_D8 + X"01";
        L_LSR_D         <= '0' & L_D8(7 downto 1);
        L_NEG_D         <= X"00" - L_D8;
        L_NOT_D         <= not L_D8;
        L_OR_DR         <= L_D8 or L_RI8;
        L_PROD          <= (L_SIGN_D & L_D8) * (L_SIGN_R & L_R8);
        L_ROR_D         <= I_FLAGS(0) &  L_D8(7 downto 1);
        L_SUB_DR        <= L_D8 - L_RI8;
        L_SBC_DR        <= L_SUB_DR - ("0000000" & I_FLAGS(0));
        L_SIGN_D        <= L_D8(7) and I_IMM(6);
        L_SIGN_R        <= L_R8(7) and I_IMM(5);
        L_SWAP_D        <= L_D8(3 downto 0) & L_D8(7 downto 4);
        L_XOR_DR        <= L_D8 xor L_R8;
```

src/alu.vhd

Most values should be obvious, but a few deserve an explanation: There is a considerable number of multiplication functions that only differ in the signedness of their operands. Instead of implementing a different 8-bit multiplier for each opcode, we use a common signed 9-bit multiplier for all opcodes. The opcode decoder sets bits 6 and/or 5 of the **IMM** input if the **D** operand and/or the **R** operand is signed. The signs of the operands are then **SIGN_D** and **SIGN_R**; they are 0 for unsigned operations. Next the signs are prepended to the operands so that each operand is 9-bit signed. If the operand was unsigned (and the sign was 0) then the new signed 9-bit operand is positive. If the operand was signed and positive (and the sign was 0 again) then the new operand is positive again. If the operand was signed and negative, then the sign was 1 and the new operand is also negative.

6.3.4 Output and Flag Multiplexing

The necessary computations in the ALU have already been made in the previous section. What remains is to select the proper result and setting the flags. The output **DOUT** and the flags are selected by **ALU_OP**. We take the first two values of **ALU_OP** as an example and leave the remaining ones as an exercise for the reader.

```
        process(L_ADC_DR, L_ADD_DR, L_ADIW_D, I_ALU_OP, L_AND_DR, L_ASR_D,
                I_BIT, I_D, L_D8, L_DEC_D, I_DIN, I_FLAGS, I_IMM, L_MASK_I,
                L_INC_D, L_LSR_D, L_NEG_D, L_NOT_D, L_OR_DR, I_PC, L_PROD,
                I_R, L_RI8, L_RBIT, L_ROR_D, L_SBIW_D, L_SUB_DR, L_SBC_DR,
                L_SIGN_D, L_SIGN_R, L_SWAP_D, L_XOR_DR)
        begin
```

```
            Q_FLAGS(9)      <= L_RBIT xor not I_BIT(3);      -- DIN[BIT] = BIT[3]
            Q_FLAGS(8)      <= ze(L_SUB_DR);                 -- D == R for CPSE
            Q_FLAGS(7 downto 0)      <= I_FLAGS;
            L_DOUT      <= X"0000";

            case I_ALU_OP is
                when ALU_ADC =>
                    L_DOUT      <= L_ADC_DR & L_ADC_DR;
                    Q_FLAGS(0)      <= cy(L_D8(7), L_RI8(7), L_ADC_DR(7));   -- Carry
                    Q_FLAGS(1)      <= ze(L_ADC_DR);                         -- Zero
                    Q_FLAGS(2)      <= L_ADC_DR(7);                          -- Negative
                    Q_FLAGS(3)      <= ov(L_D8(7), L_RI8(7), L_ADC_DR(7));   -- Overflow
                    Q_FLAGS(4)      <= si(L_D8(7), L_RI8(7), L_ADC_DR(7));   -- Signed
                    Q_FLAGS(5)      <= cy(L_D8(3), L_RI8(3), L_ADC_DR(3));   -- Halfcarry

                when ALU_ADD =>
                    L_DOUT      <= L_ADD_DR & L_ADD_DR;
                    Q_FLAGS(0)      <= cy(L_D8(7), L_RI8(7), L_ADD_DR(7));   -- Carry
                    Q_FLAGS(1)      <= ze(L_ADD_DR);                         -- Zero
                    Q_FLAGS(2)      <= L_ADD_DR(7);                          -- Negative
                    Q_FLAGS(3)      <= ov(L_D8(7), L_RI8(7), L_ADD_DR(7));   -- Overflow
                    Q_FLAGS(4)      <= si(L_D8(7), L_RI8(7), L_ADD_DR(7));   -- Signed
                    Q_FLAGS(5)      <= cy(L_D8(3), L_RI8(3), L_ADD_DR(3));   -- Halfcarry
```

src/alu.vhd

First of all, the default values for the flags and the ALU output are chosen. The default of **L_OUT** is 0, while the default for **O_FLAGS** is **I_FLAGS**. This means that all flags that are not explicitly changed remain the same. The upper two flag bits are set according to specific needs of certain skip instructions (CPSE, SBIC, SBIS, SBRC, and SBRS).

Then comes a big case statement for which we explain only the first two cases, **ALU_ADC** and **ALU_ADD**.

The expected value of **DOUT** was already computed as **L_ADC_DR** in the previous section and this value is assigned to **DOUT**.

After that the flags that can change in the execution of the **ADC** opcode are computed. The computation of flags is very similar for a number of different opcodes. We have therefore defined functions **cy()**, **ze()**, **ov()**, and **si()** for the usual way of computing these flags:

The half-carry flags is computed like the carry flag but on bits 3 rather than bits 7 of the operands and result.

The next example is **ADD**. It is similar to **ADC**, but now **L_ADD_DR** is used instead of **L_ADC_DR**.

6.3.5 Individual ALU Operations

The following table briefly describes how the **DOUT** output of the ALU is computed for the different **ALU_OP** values.

ALU_OP	DOUT	Size
ALU_ADC	D + R + Carry	8-bit
ALU_ADD	D + R	8-bit

ALU_ADIW	D + IMM	16-bit
ALU_AND	D and R	8-bit
ALU_ASR	D >> 1	8-bit
ALU_BLD	T-flag << IMM	8-bit
ALU_BST	(set T-flag)	8-bit
ALU_COM	not D	8-bit
ALU_DEC	D - 1	8-bit
ALU_EOR	D xor R	8-bit
ALU_IN	DIN	8-bit
ALU_INC	D + 1	8-bit
ALU_LSR	D >> 1	8-bit
ALU_D_MOV_Q	D	16-bit
ALU_R_MOV_Q	R	16-bit
ALU_MULT	D * R	16-bit
ALU_NEG	0 - D	8-bit
ALU_OR	A or R	8-bit
ALU_PC	PC	16-bit
ALU_PC_1	PC + 1	16-bit
ALU_PC_2	PC + 2	16-bit
ALU_ROR	D rotated right	8-bit
ALU_SBC	D - R - Carry	8-bit
ALU_SBIW	D - IMM	16-bit
ALU_SREG	(set a flag)	8-bit
ALU_SUB	D - R	8-bit
ALU_SWAP	D[3:0] & D[7:4]	8-bit

For all 8-bit computations, the result is placed onto the upper and onto the lower byte of **L_DOUT**. This saves a multiplexer at the inputs of the registers.

The final result of the ALU is obtained by multiplexing the local result **L_DOUT** and **DIN** based on **I_RSEL**.

```
381         Q_DOUT      <= (I_DIN & I_DIN) when (I_RSEL = RS_DIN) else L_DOUT;
```

src/alu.vhd

We could have placed this multiplexer at the **R** input (combined with the multiplexer for the **DIN** input) or at the **DOUT** output. Placing it at the output gives a better timing, since the opcodes using the DIN input do not perform ALU operations.

6.3.5 Temporary Z and T Flags

There are two opcodes that use the value of the **Z** flag (**CPSE**) or the #T flag (SBIC, SBIS) without setting them. For timing reasons, they are executed in two cycles - one cycle for performing a comparison or a bit access and a second cycle for actually making the decision to skip the next instruction or not.

For this reason we have introduced copies of the **Z** and **T** flags and called them **FLAGS_98**. They store the values of these flags within an instruction, but without updating the status register. The two flags are computed in the ALU:

```
            Q_FLAGS(9)       <= L_RBIT xor not I_BIT(3);       -- DIN[BIT] = BIT[3]
            Q_FLAGS(8)       <= ze(L_SUB_DR);                  -- D == R for CPSE
```

src/alu.vhd

The result is stored in the data path:

```
        flg98: process(I_CLK)
        begin
            if (rising_edge(I_CLK)) then
                L_FLAGS_98      <= A_FLAGS(9 downto 8);
            end if;
        end process;
```

src/data_path.vhd

6.4 Other Functions

Most of the data path is contained in its components **alu**, **register_file**, and **data_mem**. A few things are written directly in VHDL and shall be explained here.

Some output signals are driven directly from inputs or from instantiated components:

```
        Q_ADR           <= F_ADR;
        Q_DOUT          <= A_DOUT(7 downto 0);
        Q_INT_ENA       <= A_FLAGS(7);
        Q_OPC           <= I_OPC;
        Q_PC            <= I_PC;
```

src/data_path.vhd

The address space of the data memory is spread over the register file (general purpose registers, stack pointer, and status register), the data RAM, and the external I/O registers outside of the data path. The external I/O registers reach from 0x20 to 0x5C (including) and the data RAM starts at 0x60. We generate write enable signals for these address ranges, and a read strobe for external I/O registers. We also control the multiplexer at the input of the ALU by the address output of the register file:

```
        Q_RD_IO        <= '0'                    when (F_ADR <X"20")
                else (I_RD_M and not I_PMS) when (F_ADR       <X"5D")
                else '0';
        Q_WE_IO         <= '0'                    when (F_ADR <X"20")
                else I_WE_M(0)              when (F_ADR       <X"5D")
                else '0';
        L_WE_SRAM       <= "00"   when  (F_ADR <X"0060") else I_WE_M;
        L_DIN           <= I_DIN  when (I_PMS = '1')
                else F_S    when  (F_ADR      <X"0020")
                else I_DIN  when  (F_ADR       <X"005D")
                else F_S    when  (F_ADR       <X"0060")
                else M_DOUT(7 downto 0);
```

src/data_path.vhd

Most instructions that modify the program counter (other than incrementing it) use addresses that are being provided on the **IMM** input (from the opcode decoder).
The two exceptions are the **IJMP** instruction where the new **PC** value is the value of the **Z** register pair, and the **RET** and **RETI** instructions where the new **PC** value is popped from the stack. The new value of the **PC** (if any) is therefore:

```
        Q_NEW_PC      <= F_Z    when I_PC_OP = PC_LD_Z       -- IJMP, ICALL
               else M_DOUT when I_PC_OP = PC_LD_S       -- RET, RETI
               else I_JADR;                             -- JMP adr
```

src/data_path.vhd

Conditional branches use the **CC** output of the register file in order to decide whether the branch shall be taken or not. The opcode decoder drives the **COND** input according to the relevant bit in the status register (**I_COND[2:0]**) and according to the expected value (**COND[3]**) of that bit.

The **LOAD_PC** output is therefore '1' for unconditional branches and **CC** for conditional branches:

```
        process(I_PC_OP, F_CC)
```

```
    begin
        case I_PC_OP is
            when PC_BCC  => Q_LOAD_PC      <= F_CC;      -- maybe (PC on I_JADR)
            when PC_LD_I => Q_LOAD_PC      <= '1';       -- yes: new PC on I_JADR
            when PC_LD_Z => Q_LOAD_PC      <= '1';       -- yes: new PC in Z
            when PC_LD_S => Q_LOAD_PC      <= '1';       -- yes: new PC on stack
            when others  => Q_LOAD_PC      <= '0';       -- no.
        end case;
    end process;
```

src/data_path.vhd

When a branch is taken (in the execution stage of the pipeline), then the next instruction after the branch is about to be decoded in the opcode decoder stage. This instruction must not be executed, however, and we therefore invalidate it by asserting the **SKIP** output. Another case where instructions need to be invalidated are skip instructions (**CPSE**, **SBIC**, **SBIS**, **SBRC**, and **SBRS**). These instructions do not modify the **PC**, but they nevertheless cause the next instruction to be invalidated:

```
    process(I_PC_OP, L_FLAGS_98, F_CC)
    begin
        case I_PC_OP is
            when PC_BCC    => Q_SKIP      <= F_CC;            -- if cond met
            when PC_LD_I   => Q_SKIP      <= '1';             -- yes
            when PC_LD_Z   => Q_SKIP      <= '1';             -- yes
            when PC_LD_S   => Q_SKIP      <= '1';             -- yes
            when PC_SKIP_Z => Q_SKIP      <= L_FLAGS_98(8);   -- if Z set
            when PC_SKIP_T => Q_SKIP      <= L_FLAGS_98(9);   -- if T set
            when others    => Q_SKIP      <= '0';             -- no.
        end case;
    end process;
```

src/data_path.vhd

This concludes the discussion of the data path. We have now installed the environment that is needed to execute opcodes.

7 OPCODE DECODER

In this lesson we will describe the opcode decoder. We will also learn how the different instructions provided by the CPU will be implemented. We will not describe every opcode, but rather groups of instructions whose individual instructions are rather similar.

The opcode decoder is the middle state of our CPU pipeline. Therefore its inputs are defined by the outputs of the previous stage and its outputs are defined by the inputs of the next stage.

7.1 Inputs of the Opcode Decoder

- **CLK** is the clock signal. The opcode decoder is a pure pipeline stage so that no internal state is kept between clock cycles. The output of the opcode decoder is a pure function of its inputs.
- **OPC** is the opcode being decoded.
- **PC** is the program counter (the address in the program memory from which OPC was fetched).
- **T0** is '1' in the first cycle of the execution of the opcode. This allows for output signals of two-cycle instructions that are different in the first and the second cycle.

7.2 Outputs of the Opcode Decoder

Most data buses of the CPU are contained in the data path. In contrast, most control signals are generated in the opcode decoder. We start with a complete list of these control signals and their purpose. There are two groups of signals: select signals and write enable signals. Select signals are used earlier in the execution of the opcode for controlling multiplexers. The write enable signals are used at the end of the execution to determine where results shall be stored. Select signals are generally more time-critical that write enable signals.

The select signals are:

- **ALU_OP** defines which particular ALU operation (like **ADD**, **ADC**, **AND**, ...) the ALU shall perform.
- **AMOD** defines which addressing mode (like **absolute**, **Z+**, **-SP**, etc.)shall be used for data memory accesses.
- **BIT** is a bit value (0 or 1) and a bit number used in bit instructions.
- **DDDDD** defines the destination register or register pair (if any) for storing the result of an operation. It also defines the first source register or register pair of a dyadic instructions.
- **IMM** defines an immediate value or branch address that is computed from the opcode.
- **JADR** is a branch address.
- **OPC** is the opcode being decoded, or 0 if the opcode was invalidated by means of **SKIP**.
- **PC** is the **PC** from which **OPC** was fetched.
- **PC_OP** defines an operation to be performed on the **PC** (such as branching).
- **PMS** is set when the address defined by **AMOD** is a program memory address rather than a data memory address.
- **RD_M** is set for reads from the data memory.
- **RRRRR** defines the second register or register pair of a dyadic instruction.
- **RSEL** selects the source of the second operand in the ALU. This can be a register (on the **R** input), an immediate value (on the **IMM** input), or data from memory or I/O (on the **DIN** input).

The write enable signals are:

- **WE_01** is set when register pair 0 shall be written. This is used for multiplication instructions that store the multiplication product in register pair 0.
- **WE_D** is set when the register or register pair **DDDDD** shall be written. If both bits are set then the entire pair shall be written and **DDDDD[0]** is 0. Otherwise **WE_D[1]** is 0, and one of the registers (as defined by **DDDDD[0]**) shall be written,
- **WE_F** is set when the status register (flags) shall be written.
- **WE_M** is set when the memory (including memory mapped general purpose registers and I/O registers) shall be written. If set, then the **AMOD** output defines how to compute the memory address.
- **WE_XYZS** is set when the stack pointer or one of the pointer register pairs **X**, **Y**, or **Z** shall be written. Which of these register is meant is encoded in **AMOD**.

7.3 Structure of the Opcode Decoder

The VHDL code of the opcode decoder consists essentially of a huge case statement. At the beginning of the case statement there is a section assigning a default value to each output. Then follows a case statement that decodes the upper 6 bits of the opcode:

```
        process(I_CLK)
        begin
        if (rising_edge(I_CLK)) then
            --
            -- set the most common settings as default.
            --
            Q_ALU_OP       <= ALU_D_MV_Q;
            Q_AMOD         <= AMOD_ABS;
            Q_BIT          <= I_OPC(10) & I_OPC(2 downto 0);
            Q_DDDDD        <= I_OPC(8 downto 4);
            Q_IMM          <= X"0000";
            Q_JADR         <= I_OPC(31 downto 16);
            Q_OPC          <= I_OPC(15 downto  0);
            Q_PC           <= I_PC;
            Q_PC_OP        <= PC_NEXT;
            Q_PMS          <= '0';
            Q_RD_M         <= '0';
            Q_RRRRR        <= I_OPC(9) & I_OPC(3 downto 0);
            Q_RSEL         <= RS_REG;
            Q_WE_D         <= "00";
            Q_WE_01        <= '0';
            Q_WE_F         <= '0';
            Q_WE_M         <= "00";
            Q_WE_XYZS      <= '0';

            case I_OPC(15 downto 10) is
                when "000000" =>
```

src/opc_deco.vhd

...

```
                when others =>
            end case;
        end if;
```

```
656         end process;
```

src/opc_deco.vhd

7.4 Default Values for the Outputs

The opcode decoder generates quite a few outputs. A typical instruction, however, only sets a small fraction of them. For this reason we provide a default value for all outputs before the top level case statement, as shown above.
For each instruction we then only need to specify those outputs that differ from the default value.

Every default value is either constant or a function of an input. Therefore the opcode decoder is a typical "stateless" pipeline stage. The default values are chosen so that they do not change anything in the other stages (except incrementing the PC, of course). In particular, the default values for all write enable signals are '0'.

7.5 Checklist for the Design of an Opcode.

Designing an opcode starts with asking a number of questions. The answers are found in the specification of the opcode. The answers identify the outputs that need to be set other than their default values. While the instructions are quite different, the questions are always the same:

1. What operation shall the ALU perform? Set **ALU_OP** and **Q_WE_F** accordingly.
2. Is a destination register or destination register pair used? If so, set **DDDDD** (and **WE_D** if written).
3. Is a second register or register pair involved? If so, set **RRRRR**.
4. Does the opcode access the memory? If so, set **AMOD**, **PMS**, **RSEL**, **RD_M**, **WE_M**, and **WE_XYZS** accordingly.
5. Is an immediate or implied operand used? If so, set **IMM** and **RSEL**.
6. Is the program counter modified (other than incrementing it)? If so, set **PC_OP** and **SKIP**.
7. Is a bit number specified in the opcode ? If so, set **BIT**.
8. Are instructions skipped? If so, set **SKIP**.

Equipped with this checklist we can implement all instructions. We start with the simplest instructions and proceed to the more complex instructions.

7.6 Opcode Implementations

7.6.1 The NOP instruction

The simplest instruction is the NOP instruction which does - nothing. The default values set for all outputs do nothing either so there is no extra VHDL code needed for this instruction.

7.6.2 8-bit Monadic Instructions

We call an instruction **monadic** if its opcode contains one register number and if the instructions reads the register before computing a new value for it.

Only items 1. and 2. in our checklist apply. The default value for **DDDDD** is already correct. Thus only **ALU_OP**, **WE_D**, and **WE_F** need to be set. We take the **DEC Rd** instruction as an example:

```
                    --
                    --  1001 010d dddd 1010 - DEC
                    --
                    Q_ALU_OP       <= ALU_DEC;
                    Q_WE_D       <= "01";
                    Q_WE_F       <= '1';
```

src/opc_deco.vhd

All monadic arithmetic/logic instructions are implemented in the same way; they differ by their **ALU_OP**.

7.6.3 8-bit Dyadic Instructions, Register/Register

We call an instruction **dyadic** if its opcode contains two data sources (a data source being a register number or an immediate operand). As a consequence of the two data sources, dyadic instructions occupy a larger fraction of the opcode space than monadic functions.

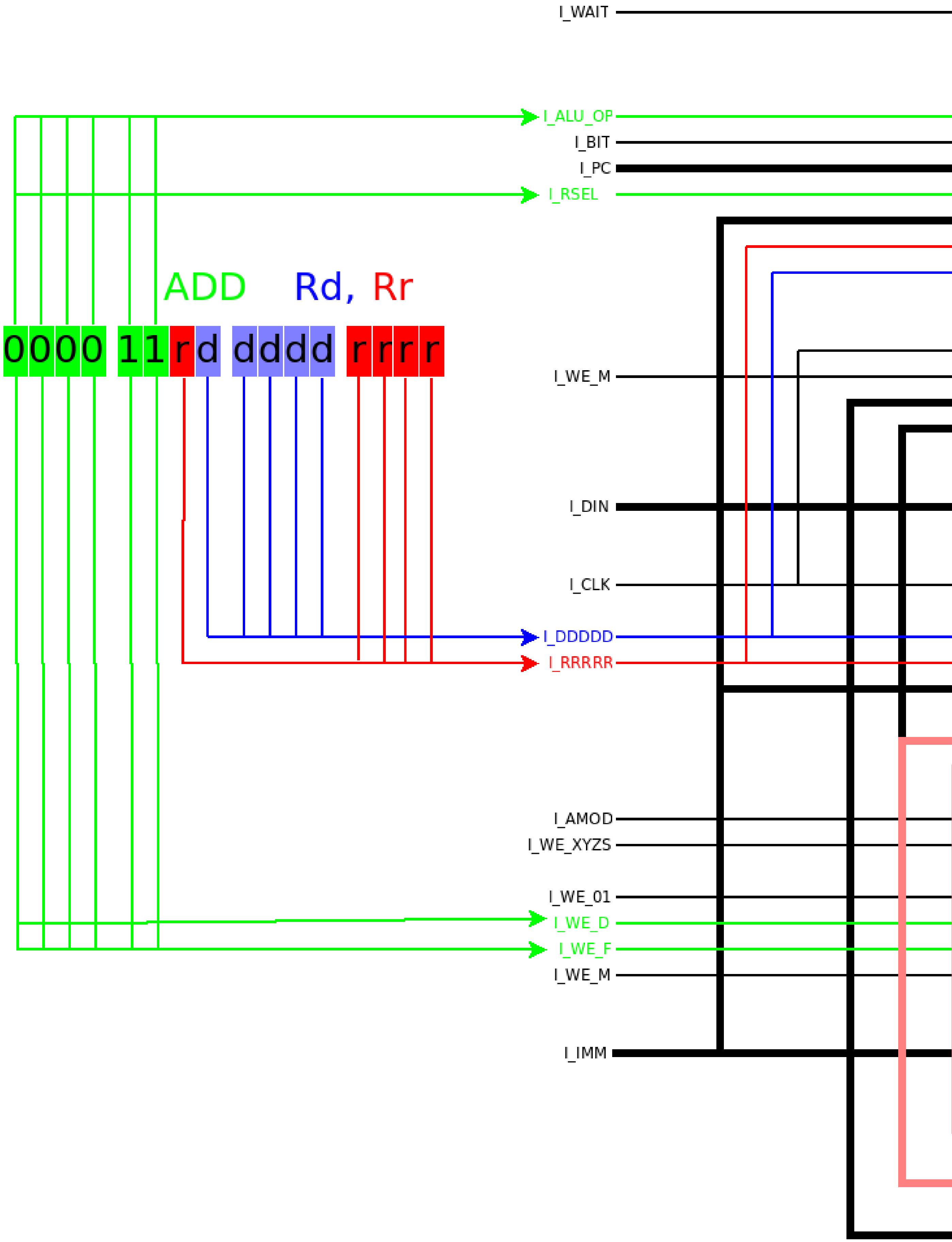
We take the **ADD Rd, Rr** opcode as an example.

Compared to the monadic functions now item 3. in the checklist applies as well. This would mean we have to set **RRRRR** but by chance the default value is already correct. Therefore:

```
            --
            -- 0000 11rd dddd rrrr - ADD
            --
            Q_ALU_OP       <= ALU_ADD;
            Q_WE_D       <= "01";
            Q_WE_F       <= '1';
```

src/opc_deco.vhd

The dyadic instructions do not use the I/O address space and therefore they completely execute inside the data path. The following figure shows the signals in the data path that are used by the **ADD Rd, Rr** instruction:


I_WAIT
I_ALU_OP
I_BIT
I_PC
I_RSEL
ADD Rd, Rr
0000 11rd dddd rrrr
I_WE_M
I_DIN
I_CLK
I_DDDDD
I_RRRRR
I_AMOD
I_WE_XYZS
I_WE_01
I_WE_D
I_WE_F
I_WE_M
I_IMM

The opcode for **ADD Rd, Rr** is **0000 11rd dddd rrrr**. The opcode decoder extracts the 'd' bits into the **DDDDD** signal (blue), the 'r' bits into the **RRRRR** signal (red), and computes **ALU_OP**, **WE_D**, and **WE_F** from the remaining bits (green) as above.

The register file converts the register numbers **Rd** and **Rr** that are encoded in the **DDDDD** and **RRRRR** signals to the contents of the register pairs at its **D** and **R** outputs. The lowest bit of the **DDDDD** and **RRRRR** signals also go to the ALU (inputs **D0** and **R0**) where the odd/even register selection from the two register pairs is performed.

The decoder also selects the proper **ALU_OP** from the opcode, which is **ALU_ADD** in this example. With this input, the ALU computes the sum of the its **D** and **R** inputs and drives its **DOUT** (pink) with the sum. It also computes the flags as defined for the **ADD** opcode.

The decoder sets the **WE_D** and **WE_F** inputs of the register file so that the **DOUT** and **FLAGS** outputs of the ALU are written back to the register file.

All this happens within a single clock cycle, so that the next instruction can be performed in the next clock cycle.

The other dyadic instructions are implemented similarly. Two instructions, **CMP** and **CPC**, deviate a little since they do not set **WE_D**. Only the flags are set as a result of the comparison. Apart from that, **CMP** and **CPC** are identical to the **SUB** and **SBC**; they don't have their own **ALU_OP** but use those of the **SUB** and **SBC** instructions.

The **MOV Rd, Rr** instruction is implemented as a dyadic function. It ignores it first argument and does not set any flags.

7.6.4 8-bit Dyadic Instructions, Register/Immediate

Some of the dyadic instructions have an immediate operand (i.e. the operand is contained in the opcode) rather than using a second register. For such instructions, for example **ANDI**, we extract the immediate operand from the opcode and set **RSEL**. Since the immediate operand takes quite some space in the opcode, the register range was restricted a little and hence the default **DDDDD** value needs a modification.

```
                        --
                        -- 0111 KKKK dddd KKKK - ANDI
                        --
                        Q_ALU_OP        <= ALU_AND;
                        Q_IMM(7 downto 0)       <= I_OPC(11 downto 8) & I_OPC(3 downto 0);
                        Q_RSEL       <= RS_IMM;
                        Q_DDDDD(4)       <= '1';    -- Rd = 16...31
                        Q_WE_D       <= "01";
                        Q_WE_F       <= '1';
```

src/opc_deco.vhd

7.6.5 16-bit Dyadic Instructions

Some of the dyadic 8-bit instructions have 16-bit variants, for example **ADIW**. The second operand of these 16-bit variants can be another register pair or an immediate operand.

```
                            --
                            --  1001 0110 KKdd KKKK - ADIW
                            --  1001 0111 KKdd KKKK - SBIW
                            --
                            if (I_OPC(8) = '0') then    Q_ALU_OP       <= ALU_ADIW;
                            else                        Q_ALU_OP       <= ALU_SBIW;
                            end if;
                            Q_IMM(5 downto 4)      <= I_OPC(7 downto 6);
                            Q_IMM(3 downto 0)      <= I_OPC(3 downto 0);
                            Q_RSEL      <= RS_IMM;
                            Q_DDDDD      <= "11" & I_OPC(5 downto 4) & "0";

                            Q_WE_D      <= "11";
                            Q_WE_F      <= '1';
```

src/opc_deco.vhd

These instructions are implemented similar to their 8-bit relatives, but in contrast to them both **WE_D** bits are set. This causes the entire register pair to be updated. **LDI** and **MOVW** are also implemented as 16-bit dyadic instruction.

7.6.6 Bit Instructions

There are some instructions that are very similar to monadic functions (in that they refer to only one register) but have a small immediate operand that addresses a bit in that register. Unlike dyadic functions with immediate operands, these bit instructions do not use the register/immediate multiplexer in the ALU (they don't have a register counterpart for the immediate operand). Instead, the bit number from the instruction is provided on the **BIT** output of the opcode decoder. The **BIT** output has 4 bits; in addition to the (lower) 3 bits needed to address the bit concerned, the fourth (upper) bit indicates the value (bit set or bit cleared) of the bit for those instructions that need it.

The ALU operations related to these bit instructions are **ALU_BLD** and **ALU_BIT_CS.**

ALU_BLD stores the T bit of the status register into a bit in a general purpose register; this is used to implement the **BLD** instruction.

ALU_BIT_CS is a dual-purpose function.

The first purpose is to copy a bit in a general purpose register into the **T** flag of the status register. This use of **ALU_BIT_CS** is selected by setting (only) the **WE_F** signal so that the status register is updated with the new **T** flag. The **BST** instruction is implemented this way. The the bit value in **BIT[3]** is ignored.

The second purpose is to set or clear a bit in an I/O register. The ALU first computes a bitmask where only the bit indicated by **BIT[2:0]** is set. Depending on BIT[3] the register is then **or**'ed with the mask or **and**'ed with the complement of the mask. This sets or clears the bit in the current value of the register. This use of **ALU_BIT_CS** is selected by **WE_M** so the I/O register is updated with the new value. The **CBI** and **SBI** instructions are implemented this way.

ALU_BIT_CS is also used by the skip instructions **SBRC** and **SBRC** that are described in the section about branching.

7.6.7 Multiplication Instructions

There is a zoo of multiplication instructions that differ in the signedness of their operands (**MUL**, **MULS**, **MULSU**) and in whether the final result is shifted (**FMUL**, **FMULS**, and **FMULSU**) or not. The opcode decoder sets certain bits in the IMM signal to indicate the type of multiplication:

IMM(7) = 1	shift (FMULxx)
IMM(6) = 1	Rd is signed
IMM(5) = 1	Rr is signed

We also set the **WE_01** instead of the **WE_D** signal because the multiplication result is stored in register pair 0 rather than in the Rd register of the opcode.

```
                                --
                                -- 0000 0011 0ddd 0rrr - _MULSU  SU "010"
                                -- 0000 0011 0ddd 1rrr - FMUL    UU "100"
                                -- 0000 0011 1ddd 0rrr - FMULS   SS "111"
                                -- 0000 0011 1ddd 1rrr - FMULSU  SU "110"
                                --
                                Q_DDDDD(4 downto 3)      <= "10";    -- regs 16 to 23
                                Q_RRRRR(4 downto 3)      <= "10";    -- regs 16 to 23
                                Q_ALU_OP       <= ALU_MULT;
                                if I_OPC(7) = '0' then
                                    if I_OPC(3) = '0' then
                                        Q_IMM(7 downto 5)      <= MULT_SU;
                                    else
                                        Q_IMM(7 downto 5)      <= MULT_FUU;
                                    end if;
                                else
                                    if I_OPC(3) = '0' then
                                        Q_IMM(7 downto 5)      <= MULT_FSS;
                                    else
                                        Q_IMM(7 downto 5)      <= MULT_FSU;
                                    end if;
                                end if;
                                Q_WE_01       <= '1';
                                Q_WE_F       <= '1';
```

src/opc_deco.vhd

7.6.8 Instructions Writing To Memory or I/O

Instructions that write to memory or I/O registers need to set **AMOD**. **AMOD** selects the pointer register involved (**X**, **Y**, **Z**, **SP**, or none). If the addressing mode involves a pointer register and updates it, then **WE_XYZS** needs to be set as well.

The following code fragment shows a number of store functions and how **AMOD** is computed:

```
                    --
                    -- 1001 00-1r rrrr 0000 - STS
                    -- 1001 00-1r rrrr 0001 - ST Z+. Rr
                    -- 1001 00-1r rrrr 0010 - ST -Z. Rr
                    -- 1001 00-1r rrrr 1000 - ST Y. Rr
                    -- 1001 00-1r rrrr 1001 - ST Y+. Rr
                    -- 1001 00-1r rrrr 1010 - ST -Y. Rr
                    -- 1001 00-1r rrrr 1100 - ST X. Rr
                    -- 1001 00-1r rrrr 1101 - ST X+. Rr
                    -- 1001 00-1r rrrr 1110 - ST -X. Rr
                    -- 1001 00-1r rrrr 1111 - PUSH Rr
                    --
                    Q_ALU_OP      <= ALU_D_MV_Q;
                    Q_WE_M      <= "01";
                    Q_WE_XYZS      <= '1';
                    case I_OPC(3 downto 0) is
                        when "0000" => Q_AMOD      <= AMOD_ABS;  Q_WE_XYZS <= '0';
                        when "0001" => Q_AMOD      <= AMOD_Zi;
                        when "0010" => Q_AMOD      <= AMOD_dZ;
                        when "1001" => Q_AMOD      <= AMOD_Yi;
                        when "1010" => Q_AMOD      <= AMOD_dY;
                        when "1100" => Q_AMOD      <= AMOD_X;    Q_WE_XYZS <= '0';
                        when "1101" => Q_AMOD      <= AMOD_Xi;
                        when "1110" => Q_AMOD      <= AMOD_dX;
                        when "1111" => Q_AMOD      <= AMOD_dSP;
                        when others =>
                    end case;
```

src/opc_deco.vhd

ALU_OP is set to **ALU_D_MOV_Q**. This causes the source register indicated by **DDDDD** to be switched through the ALU unchanged so that is shows up at the input of the data memory and of the I/O block. We set **WE_M** so that the value of the source register will be written.

Write instructions to memory execute in a single cycle.

7.6.9 Instructions Reading From Memory or I/O

Instructions that read from memory set **AMOD** and possibly **WE_XYZS** in the same way as instructions writing to memory.

The following code fragment shows a number of load functions:

```
                Q_IMM       <= I_OPC(31 downto 16);   -- absolute address for LDS/STS
                if (I_OPC(9) = '0') then         -- LDD / POP
                    --
                    -- 1001 00-0d dddd 0000 - LDS
                    -- 1001 00-0d dddd 0001 - LD Rd, Z+
                    -- 1001 00-0d dddd 0010 - LD Rd, -Z
                    -- 1001 00-0d dddd 0100 - (ii)  LPM Rd, (Z)
                    -- 1001 00-0d dddd 0101 - (iii) LPM Rd, (Z+)
                    -- 1001 00-0d dddd 0110 - ELPM Z        --- not mega8
                    -- 1001 00-0d dddd 0111 - ELPM Z+       --- not mega8
                    -- 1001 00-0d dddd 1001 - LD Rd, Y+
                    -- 1001 00-0d dddd 1010 - LD Rd, -Y
                    -- 1001 00-0d dddd 1100 - LD Rd, X
                    -- 1001 00-0d dddd 1101 - LD Rd, X+
                    -- 1001 00-0d dddd 1110 - LD Rd, -X
                    -- 1001 00-0d dddd 1111 - POP Rd
                    --
                    Q_RSEL      <= RS_DIN;
                    Q_RD_M      <= I_T0;
                    Q_WE_D      <= '0' & not I_T0;
                    Q_WE_XYZS      <= not I_T0;
                    Q_PMS      <= (not I_OPC(3)) and I_OPC(2) and (not I_OPC(1));
                    case I_OPC(3 downto 0) is
                        when "0000" => Q_AMOD      <= AMOD_ABS;  Q_WE_XYZS <= '0';
                        when "0001" => Q_AMOD      <= AMOD_Zi;
                        when "0100" => Q_AMOD      <= AMOD_Z;    Q_WE_XYZS <= '0';
                        when "0101" => Q_AMOD      <= AMOD_Zi;
                        when "1001" => Q_AMOD      <= AMOD_Yi;
                        when "1010" => Q_AMOD      <= AMOD_dY;
                        when "1100" => Q_AMOD      <= AMOD_X;    Q_WE_XYZS <= '0';
                        when "1101" => Q_AMOD      <= AMOD_Xi;
                        when "1110" => Q_AMOD      <= AMOD_dX;
                        when "1111" => Q_AMOD      <= AMOD_SPi;
                        when others =>                       Q_WE_XYZS      <= '0';
                    end case;
```

src/opc_deco.vhd

The data read from memory now comes from the **DIN** input. We therefore set **RSEL** to **RS_DIN**. The data read from the memory is again switched through the ALU unchanged, but we use **ALU_R_MOV_Q** instead of **ALU_D_MOV_Q** because the data from memory is now routed via the multiplexer for **R8** rather than via the multiplexer for **D8**. We generate **RD_M** instead of **WE_M** since we are now reading and not writing. The result is stored in the register indicated by **DDDDD**, so we set **WE_D**.

One of the load instructions is **LPM** which reads from program store rather then from the data memory. For this instruction we set **PMS**.

Unlike store instructions, load instructions execute in two cycles. The reason is the internal memory modules which need one clock cycle to produce a result. We therefore generate the **WE_D** and **WE_XYZS** only on the second of the two cycles.

7.6.10 Jump and Call Instructions

7.6.10.1 Unconditional Jump to Absolute Address

The simplest case of a jump instruction is **JMP**, an unconditional jump to an absolute address:

The target address of the jump follows after the instruction. Due to our odd/even trick with the program memory, the target address is provided on the upper 16 bits of the opcode and we need not wait for it. We copy the target address from the upper 16 bits of the opcode to the **IMM** output. Then we set **PC_OP** to **PC_LD_I**:

```
                                        --
                                        --  1001 010k kkkk 110k - JMP (k = 0 for 16 bit)
                                        --  kkkk kkkk kkkk kkkk
                                        --
                                        Q_PC_OP      <= PC_LD_I;
```

src/opc_deco.vhd

The execution stage will then cause the **PC** to be loaded from its **JADR** input:

```
                    when PC_LD_I => Q_LOAD_PC      <= '1';        -- yes: new PC on I_JADR
```

src/data_path.vhd

The next opcode after the **JMP** is already in the pipeline and would be executed next. We invalidate the next opcode so that it will not be executed:

```
                    when PC_LD_I   => Q_SKIP      <= '1';             -- yes
```

src/data_path.vhd

An instruction similar to **JMP** is **IJMP**. The difference is that the target address of the jump is not provided as an immediate address following the opcode, but is the content of the Z register. This case is handled by a different **PC_OP**:

```
                                --
                                --  1001 0100 0000 1001 IJMP
                                --  1001 0100 0001 1001 EIJMP   -- not mega8
                                --  1001 0101 0000 1001 ICALL
                                --  1001 0101 0001 1001 EICALL   -- not mega8
                                --
                                Q_PC_OP      <= PC_LD_Z;
```

src/opc_deco.vhd

The execution stage, which contains the **Z** register, performs the selection of the target address, as we have already seen in the discussion of the data path.

7.6.10.2 Unconditional Jump to Relative Address

The **RJMP** instruction is similar to the **JMP** instruction. The target address of the jump is, however, an address relative to the current **PC** (plus 1). We sign-extend the relative address (by replicating **OPC(11)** until a 16-bit value is reached) and add the current **PC**.

```
                --
                -- 1100 kkkk kkkk kkkk - RJMP
                --
                Q_JADR      <= I_PC + (I_OPC(11) & I_OPC(11) & I_OPC(11) & I_OPC(11)
                                  & I_OPC(11 downto 0)) + X"0001";
                Q_PC_OP      <= PC_LD_I;
```

src/opc_deco.vhd

The rest of **RJMP** is the same as for **JMP**.

7.6.10.3 Conditional Jump to Relative Address

There is a number of conditional jump instructions that differ by the bit in the status register that controls whether the branch is taken or not. **BRCS** and **BRCC** branch if bit 0 (the carry flag) is set resp. cleared. **BREQ** and **BRNE** branch if bit 1 (the zero flag) is set resp. cleared, and so on.

There is also a generic form where the bit number is an operand of the opcode. **BRBS** branches if a status register flag is set while **BRBC** branches if a bit is cleared. This means that **BRCS**, **BREQ**, ... are just different names for the **BRBS** instruction, while **BRCC**, **BRNE**, ... are different name for the **BRBC** instruction.

The relative address (i.e. the offset from the PC) for **BRBC/BRBS** is shorter (7 bit) than for **RJMP** (12 bit). Therefore the sign bit of the offset is replicated more often in order to get a 16-bit signed offset that can be added to the **PC**.

```
--
-- 1111 00kk kkkk kbbb - BRBS
-- 1111 01kk kkkk kbbb - BRBC
--       v
-- bbb: status register bit
-- v: value (set/cleared) of status register bit
--
Q_JADR      <= I_PC + (I_OPC(9) & I_OPC(9) & I_OPC(9) & I_OPC(9)
                   & I_OPC(9) & I_OPC(9) & I_OPC(9) & I_OPC(9)
                   & I_OPC(9) & I_OPC(9 downto 3)) + X"0001";
Q_PC_OP      <= PC_BCC;
```

src/opc_deco.vhd

The decision to branch or not is taken in the execution stage, because at the time where the conditional branch is decoded, the relevant bit in the status register is not yet valid.

7.6.10.4 Call Instructions

Many unconditional jump instructions have "call" variant. The "call" variant are executed like the corresponding jump instruction. In addition (and at the same time), the **PC** after the instruction is pushed onto the stack. We take **CALL**, the brother of **JMP** as an example:

```
--
--  1001 010k kkkk 111k - CALL (k = 0)
--  kkkk kkkk kkkk kkkk
--
Q_ALU_OP       <= ALU_PC_2;
Q_AMOD       <= AMOD_ddSP;
Q_PC_OP       <= PC_LD_I;
Q_WE_M       <= "11";     -- both PC bytes
Q_WE_XYZS       <= '1';
```

src/opc_deco.vhd

The new things are an **ALU_OP** of **ALU_PC_2**. The ALU adds 2 to the **PC**, since the **CALL** instructions is 2 words long. The **RCALL** instruction, which is only 1 word long would use **ALU_PC_1** instead. **AMOD** is pre-decrement of the **SP** by 2 (since the return address is 2 bytes long). Both bits of **WE_M** are set since we write 2 bytes.

7.6.10.5 Skip Instructions

Skip instructions do not modify the PC, but they invalidate the next instruction. Like for conditional branch instructions, the condition is checked in the execution stage.

We take **SBIC** as an example:

```
                    --
                    --  1001 1000 AAAA Abbb - CBI
                    --  1001 1001 AAAA Abbb - SBIC
                    --  1001 1010 AAAA Abbb - SBI
                    --  1001 1011 AAAA Abbb - SBIS
                    --
                    Q_ALU_OP      <= ALU_BIT_CS;
                    Q_AMOD      <= AMOD_ABS;
                    Q_BIT(3)      <= I_OPC(9);   -- set/clear

                    -- IMM = AAAAAA + 0x20
                    --
                    Q_IMM(4 downto 0)      <= I_OPC(7 downto 3);
                    Q_IMM(6 downto 5)      <= "01";

                    Q_RD_M      <= I_T0;
                    if ((I_OPC(8) = '0') ) then     -- CBI or SBI
                        Q_WE_M(0)      <= '1';
                    else                            -- SBIC or SBIS
                        if (I_T0 = '0') then        -- second cycle.
                            Q_PC_OP      <= PC_SKIP_T;
                        end if;
                    end if;
```

src/opc_deco.vhd

First of all, **AMOD**, **IMM**, and **RSEL** are set such that the value from the I/O register indicated by **IMM** reaches the ALU. **ALU_OP** and **BIT** are set such that the relevant bit reaches **FLAGS_98(9)** in the data path. The access of the bit followed by a skip decision would have taken too long for a single cycle. We therefore extract the bit in the first cycle and store it in the **FLAGS_98(9)** signal in the data path. In the next cycle, the decision to skip or not is taken.

The **PC_OP** of **PC_SKIP_T** causes the **SKIP** output of the execution stage to be raised if **FLAGS_98(9)** is set:

```
                when PC_SKIP_T => Q_SKIP      <= L_FLAGS_98(9);  -- if T set
```

src/data_path.vhd

A similar instruction is CPSE, which skips the next instruction when a comparison (rather than a bit in an I/O register) indicates equality. It works like a CP instruction, but raises **SKIP** in the execution stage rather than updating the status register.

7.6.10.6 Interrupts

We have seen earlier, that the opcode fetch stage inserts "interrupt instructions" into the pipeline when an interrupt occurs. These interrupt instructions are similar to **CALL** instructions. In contrast to **CALL** instructions, however, we use **ALU_INTR** instead of **ALU_PC_2**. This copies the **PC** (rather than **PC** + 2) to the output of the ALU (due to the fact that we have overridden a valid instruction and want to continue with exactly that instruction after returning from the interrupt, Another thing that **ALU_INTR** does is to clear the **I** flag in the status register.

The interrupt opcodes are implemented as follows:

```
                            --
                            -- 0000 0000 0000 0000 - NOP
                            -- 0000 0000 001v vvvv - INTERRUPT
                            --
                            if (I_OPC(5)) = '1' then   -- interrupt
                                Q_ALU_OP       <= ALU_INTR;
                                Q_AMOD       <= AMOD_ddSP;
                                Q_JADR       <= "0000000000" & I_OPC(4 downto 0) & "0";
                                Q_PC_OP       <= PC_LD_I;
                                Q_WE_F       <= '1';
                                Q_WE_M       <= "11";
                            end if;
```

src/opc_deco.vhd

7.6.11 Instructions Not Implemented

A handful of instructions was not implemented. The reasons for not implementing them is one of the following:

1. The instruction is only available in particular devices, typically due to extended capabilities of these devices (**EICALL**, **EIJMP**, **ELPM**).
2. The instruction uses capabilities that are somewhat unusual in general (**BREAK**, **DES**, **SLEEP**, **WDR**).

These instructions are normally not generated by C/C++ compilers, but need to be generated by means of **#asm** directives. At this point the reader should have learned enough to implement these functions when needed.

7.7 Index of all Instructions

The following table lists all CPU instructions and a reference to the chapter where they are (supposed to be) described.

ADC	7.6.3	8-bit Dyadic Instructions, Register/Register
ADD	7.6.3	8-bit Dyadic Instructions, Register/Register

ADIW	7.6.5	16-bit Dyadic Instructions
AND	7.6.3	8-bit Dyadic Instructions, Register/Register
ANDI	7.6.4	8-bit Dyadic Instructions, Register/Immediate
ASR	7.6.2	8-bit Monadic Instructions
BCLR	7.6.2	8-bit Monadic Instructions
BLD	7.6.2	8-bit Monadic Instructions
BRcc	7.6.10	Jump and Call Instructions
BREAK	7.6.11	Instructions Not Implemented
BSET	7.6.2	8-bit Monadic Instructions
BST	7.6.2	8-bit Monadic Instructions
CALL	7.6.10	Jump and Call Instructions
CBI	7.6.6	Bit Instructions
CBR	-	see ANDI
CL<flag>	-	see BCLR
CLR	-	see LDI
COM	7.6.2	8-bit Monadic Instructions
CP	7.6.3	8-bit Dyadic Instructions, Register/Register
CPC	7.6.3	8-bit Dyadic Instructions, Register/Register
CPI	7.6.4	8-bit Dyadic Instructions, Register/Immediate
CPSE	7.6.10	Jump and Call Instructions
DEC	7.6.2	8-bit Monadic Instructions
DES	7.6.11	Instructions Not Implemented
EICALL	7.6.11	Instructions Not Implemented
EIJMP	7.6.11	Instructions Not Implemented
ELPM	7.6.11	Instructions Not Implemented
EOR	7.6.3	8-bit Dyadic Instructions, Register/Register
FMUL[SU]	7.6.7	Multiplication Instructions
ICALL	7.6.10	Jump and Call Instructions
IN	7.6.9	Instructions Reading From Memory or I/O
INC	7.6.2	8-bit Monadic Instructions
IJMP	7.6.10	Jump and Call Instructions
JMP	7.6.10	Jump and Call Instructions
LDD	7.6.9	Instructions Reading From Memory or I/O
LDI	7.6.5	16-bit Dyadic Instructions
LDS	7.6.9	Instructions Reading From Memory or I/O
LSL	7.6.2	8-bit Monadic Instructions
LSR	7.6.2	8-bit Monadic Instructions
MOV	7.6.3	8-bit Dyadic Instructions, Register/Register
MOVW	7.6.5	16-bit Dyadic Instructions
MUL[SU]	7.6.7	Multiplication Instructions
NEG	7.6.2	8-bit Monadic Instructions
NOP	7.6.1	The NOP instruction

NOT	7.6.2	8-bit Monadic Instructions
OR	7.6.3	8-bit Dyadic Instructions, Register/Register
ORI	7.6.4	8-bit Dyadic Instructions, Register/Immediate
OUT	7.6.8	Instructions Writing To Memory or I/O
POP	7.6.9	Instructions Reading From Memory or I/O
PUSH	7.6.8	Instructions Writing To Memory or I/O
RCALL	7.6.10	Jump and Call Instructions
RET	7.6.10	Jump and Call Instructions
RETI	7.6.10	Jump and Call Instructions
RJMP	7.6.10	Jump and Call Instructions
ROL	7.6.2	8-bit Monadic Instructions
SBC	7.6.3	8-bit Dyadic Instructions, Register/Register
SBCI	7.6.4	8-bit Dyadic Instructions, Register/Immediate
SBI	7.6.6	Bit Instructions
SBIC	7.6.10	Jump and Call Instructions
SBIS	7.6.10	Jump and Call Instructions
SBIW	7.6.5	16-bit Dyadic Instructions
SBR	-	see ORI
SBRC	7.6.10	Jump and Call Instructions
SBRS	7.6.10	Jump and Call Instructions
SE<flag>	-	see BSET
SER	-	see LDI
SLEEP	7.6.11	Instructions Not Implemented
SPM	7.6.8	Instructions Writing To Memory or I/O
STD	7.6.8	Instructions Writing To Memory or I/O
STS	7.6.8	Instructions Writing To Memory or I/O
SUB	7.6.3	8-bit Dyadic Instructions, Register/Register
SUBI	7.6.4	8-bit Dyadic Instructions, Register/Immediate
SWAP	7.6.2	8-bit Monadic Instructions
WDR	7.6.11	Instructions Not Implemented

This concludes the discussion of the CPU. In the next lesson we will proceed with the input/output unit.

8 INPUT/OUTPUT

The last piece in the design is the input/output unit. Strictly speaking it does not belong to the CPU as such, but we discuss it briefly to see how it connects to the CPU.

8.1 Interface to the CPU

As we have already seen in the top level design, the I/O unit uses the same clock as the CPU (which greatly simplifies its design).

The interface towards the CPU consist of the following signals:

ADR_IO	The number of an individual I/O register
DIN	Data to an I/O register (I/O write)
RD_IO	Read Strobe
WR_IO	Write Strobe
DOUT	Data from an I/O register (I/O read cycle.

These signals are well known from other I/O devices like UARTs, Ethernet Controllers, and the like.

The CPU supports two kinds of accesses to I/O registers: I/O reads (with the IN or LDS instructions, but also for the skip instructions SBIC and SBIS), and I/O writes (with the OUT or STS instructions, but also with the bit instructions CBI and SBI).

The skip instructions SBIC and SBIS execute in 2 cycles; in the first cycle an I/O read is performed while the skip (or not) decision is made in the second cycle. The reason for this is that the combinational delay for a single cycle would have been too long.

From the I/O unit's perspective, I/O reads and writes are performed in a single cycle (even if the CPU needs another cycle to complete an instruction.

The I/O unit generates an interrupt vector on its **INTVEC** output. The upper bit of the **INTVEC** output is set if an interrupt is pending.

8.2 CLR Signal

Some I/O components need a **CLR** signal to bring them into a defined state. The **CLR** signal of the CPU is used for this purpose.

8.3 Connection the FPGA Pins

The remaining signals into and out of the I/O unit are more or less directly connected to FPGA pins.

The **RX** input comes from an RS232 receiver/driver chip and is the serial input for an UART (active low). The TX output (also active low) is the serial output from that UART and goes back to the RS232 receiver/driver chip:

```
                    I_RX        => I_RX,

                    Q_TX        => Q_TX,
```

src/io.vhd

The **SWITCH** input comes from a DIP switch on the board. The values of the switch can be read from I/O register **PINB** (0x36).

```
                when X"36"  => Q_DOUT       <= I_SWITCH;  -- PINB
```

src/io.vhd

The 7_**SEGMENT** output drives the 7 segments of a 7-segment display. This output can be set from software by writing to the **PORTB** (0x38) I/O register. The segments can also be driven by a debug function which shows the current **PC** and the current opcode of the CPU.

```
                        when X"38"  => Q_7_SEGMENT       <= I_DIN(6 downto 0);    -- PORTB
```

src/io.vhd

The choice between the debug display and the software controlled display function is made by the DIP switch setting:

src/avr_fpga.vhd

8.4 I/O Read

I/O read cycles are indicated by the **RD_IO** signal. If **RD_IO** is applied, then the address of the I/O register to be read is provided on the **ADR_IO** input and the value of that register is expected on **DOUT** at the next **CLK** edge.

This is accomplished by the I/O read process:

```
    iord: process(I_ADR_IO, I_SWITCH,
                  U_RX_DATA, U_RX_READY, L_RX_INT_ENABLED,
                  U_TX_BUSY, L_TX_INT_ENABLED)
    begin
        -- addresses for mega8 device (use iom8.h or #define __AVR_ATmega8__).
        --
        case I_ADR_IO is
            when X"2A"  => Q_DOUT      <=            -- UCSRB:
                                   L_RX_INT_ENABLED  -- Rx complete int enabled.
                                 & L_TX_INT_ENABLED  -- Tx complete int enabled.
                                 & L_TX_INT_ENABLED  -- Tx empty int enabled.
                                 & '1'               -- Rx enabled
                                 & '1'               -- Tx enabled
                                 & '0'               -- 8 bits/char
                                 & '0'               -- Rx bit 8
                                 & '0';              -- Tx bit 8
            when X"2B"  => Q_DOUT      <=            -- UCSRA:
                                   U_RX_READY       -- Rx complete
                                 & not U_TX_BUSY    -- Tx complete
                                 & not U_TX_BUSY    -- Tx ready
                                 & '0'              -- frame error
                                 & '0'              -- data overrun
                                 & '0'              -- parity error
                                 & '0'              -- double dpeed
                                 & '0';             -- multiproc mode
            when X"2C"  => Q_DOUT      <= U_RX_DATA; -- UDR
            when X"40"  => Q_DOUT      <=           -- UCSRC
                                   '1'              -- URSEL
                                 & '0'              -- asynchronous
                                 & "00"             -- no parity
                                 & '1'              -- two stop bits
                                 & "11"             -- 8 bits/char
                                 & '0';             -- rising clock edge

            when X"36"  => Q_DOUT      <= I_SWITCH;  -- PINB
            when others => Q_DOUT      <= X"AA";
        end case;
    end process;
```

src/io.vhd

I/O registers that are not implemented (i.e almost all) set **DOUT** to 0xAA as a debugging aid.

The outputs of sub-components (like the UART) are selected in the I/O read process.

8.5 I/O Write

I/O write cycles are indicated by the **WR_IO** signal. If **WR_IO** is applied, then the address of the I/O register to be written is provided on the **ADR_IO** input and the value to be written is supplied on the DIN input:

```
    iowr: process(I_CLK)
    begin
```

```
            if (rising_edge(I_CLK)) then
                if (I_CLR = '1') then
                    L_RX_INT_ENABLED       <= '0';
                    L_TX_INT_ENABLED       <= '0';
                elsif (I_WE_IO = '1') then
                    case I_ADR_IO is
                        when X"38"  => Q_7_SEGMENT       <= I_DIN(6 downto 0);    -- PORTB
                                       L_LEDS      <= not L_LEDS;
                        when X"40"  =>  -- handled by uart
                        when X"41"  =>  -- handled by uart
                        when X"43"  => L_RX_INT_ENABLED      <= I_DIN(0);
                                       L_TX_INT_ENABLED      <= I_DIN(1);
                        when others =>
                    end case;
                end if;
            end if;
        end process;
```

src/io.vhd

In the I/O read process the outputs of sub-component were multiplexed into the final output **DOUT** and hence their register numbers (like 0x2C for the **UDR** read register) were visible, In the I/O write process, however, the inputs of sub-components (again like 0x2C for the **UDR** write register) are not visible in the write process and decoding of the **WR** (and **RD** where needed) strobes for sub components is done outside of these processes:

```
        L_WE_UART      <= I_WE_IO when (I_ADR_IO = X"2C") else '0'; -- write UART UDR
        L_RD_UART      <= I_RD_IO when (I_ADR_IO = X"2C") else '0'; -- read  UART UDR
```

src/io.vhd

8.6 Interrupts

Some I/O components raise interrupts, which are coordinated in the I/O interrupt process:

```
        ioint: process(I_CLK)
        begin
            if (rising_edge(I_CLK)) then
                if (I_CLR = '1') then
                    L_INTVEC      <= "000000";
                else
                    if (L_RX_INT_ENABLED and U_RX_READY) = '1' then
                        if (L_INTVEC(5) = '0') then     -- no interrupt pending
                            L_INTVEC      <= "101011";       -- _VECTOR(11)
                        end if;
                    elsif (L_TX_INT_ENABLED and not U_TX_BUSY) = '1' then
```

```
                    if (L_INTVEC(5) = '0') then     -- no interrupt pending
                        L_INTVEC      <= "101100";       -- _VECTOR(12)
                    end if;
                else                                -- no interrupt
                    L_INTVEC      <= "000000";
                end if;
            end if;
        end if;
    end process;
```

src/io.vhd

8.7 The UART

The UART is an important facility for debugging programs that are more complex than out **hello.c**. We use a fixed baud rate of 38400 Baud and a fixed data format of 8 data bits and 2 stop bits. Therefore the corresponding bits in the UART control registers of the original AVR CPU are not implemented. The fixed values are properly reported, however.

The UART consists of 3 independent sub-components: a baud rate generator, a receiver, and a transmitter.

8.7.1 The UART Baud Rate Generator

The baud rate generator is clocked with a frequency of **clock_freq** and is supposed to generate a x1 clock of **baud_rate** for the transmitter and a x16 clock of 16***baud_rate** for the receiver.

The x16 clock is generated like this:

```

    baud16: process(I_CLK)
    begin
        if (rising_edge(I_CLK)) then
            if (I_CLR = '1') then
                L_COUNTER      <= X"00000000";
            elsif (L_COUNTER >= LIMIT) then
                L_COUNTER      <= L_COUNTER - LIMIT;
            else
                L_COUNTER      <= L_COUNTER + BAUD_16;
            end if;
        end if;
    end process;

    baud1: process(I_CLK)
```

src/baudgen.vhd

We have done a little trick here. Most baud rate generators divide the input clock by a fixed integer number (like the one shown below for the x1 clock). That is fine if the input clock is a multiple of the output clock.

More often than not is the CPU clock not a multiple of the the baud rate. Therefore, if an integer divider is used (like in the original AVR CPU, where the integer divisor was written into the UBRR I/O register) then the error in the baud rate cumulates over all bits transmitted. This can cause transmission errors when many characters are sent back to back. An integer divider would have set **L_COUNTER** to 0 after reaching **LIMIT**, which would have cause an absolute error of **COUNTER** - **LIMIT**. What we do instead is to subtract **LIMIT**, which does no discard the error but makes the next cycle a little shorter instead.

Instead of using a fixed baud rate interval of N times the clock interval (as fixed integer dividers would), we have used a variable baud rate interval; the length of the interval varies slightly over time, but the total error remains bounded. The error does not cumulate as for fixed integer dividers.

If you want to make the baud rate programmable, then you can replace the generic **baud_rate** by a signal (and the trick would still work).

The x1 clock is generated by dividing the x16 clock by 16:

```
            if (rising_edge(I_CLK)) then
                if (I_CLR = '1') then
                    L_CNT_16      <= "0000";
                elsif (L_CE_16 = '1') then
                    L_CNT_16      <= L_CNT_16 + "0001";
                end if;
            end if;
        end process;

        L_CE_16      <= '1' when (L_COUNTER >= LIMIT) else '0';
        Q_CE_16      <= L_CE_16;
        Q_CE_1      <= L_CE_16 when L_CNT_16 = "1111" else '0';
```

src/baudgen.vhd

8.7.2 The UART Transmitter

The UART transmitter is a shift register that is loaded with the character to be transmitted (prepended with a start bit):

```
                    elsif (L_FLAG /= I_FLAG) then       -- new byte
                        Q_TX      <= '0';                    -- start bit
                        L_BUF      <= I_DATA;                -- data bits
                        L_TODO      <= "1001";
                    end if;
```

src/uart_tx.vhd

The **TODO** signal holds the number of bits that remain to be shifted out. The transmitter is clocked with the x1 baud rate:

```
                elsif (I_CE_1 = '1') then
                    if (L_TODO /= "0000") then          -- transmitting
                        Q_TX      <= L_BUF(0);                -- next bit
                        L_BUF          <= '1' & L_BUF(7 downto 1);
                        if (L_TODO = "0001") then
                            L_FLAG      <= I_FLAG;
                        end if;
                        L_TODO      <= L_TODO - "0001";
                    elsif (L_FLAG /= I_FLAG) then       -- new byte
                        Q_TX      <= '0';                     -- start bit
                        L_BUF      <= I_DATA;                 -- data bits
                        L_TODO      <= "1001";
                    end if;
                end if;
```

src/uart_tx.vhd

8.7.3 The UART Receiver

The UART transmitter runs synchronously with the CPU clock; but the UART receiver does not. We therefore clock the receiver input twice in order to synchronize it with the CPU clock:

```
        process(I_CLK)
        begin
            if (rising_edge(I_CLK)) then
                if (I_CLR = '1') then
                    L_SERIN      <= '1';
                    L_SER_HOT      <= '1';
                else
                    L_SERIN        <= I_RX;
                    L_SER_HOT      <= L_SERIN;
                end if;
            end if;
        end process;
```

src/uart_rx.vhd

The key signal in the UART receiver is **POSITION** which is the current position within the received character in units if 1/16 bit time. When the receiver is idle and a start bit is received, then **POSITION** is reset to 1:

```
                    if (L_POSITION = X"00") then            -- uart idle
                        L_BUF        <= "1111111111";
                        if (L_SER_HOT = '0')  then          -- start bit received
                            L_POSITION      <= X"01";
                        end if;
```

src/uart_rx.vhd

At every subsequent edge of the 16x baud rate, **POSITION** is incremented and the receiver input (**SER_HOT**) input is checked at the middle of each bit (i.e. when **POSITION[3:0]** = "0111"). If the start bit has disappeared at the middle of the bit, then this is considered noise on the line rather than a valid start bit:

```
                        if (L_POSITION(3 downto 0) = "0111") then       -- 1/2 bit
                            L_BUF      <= L_SER_HOT & L_BUF(9 downto 1);     -- sample data
                            --
                            -- validate start bit
                            --
                            if (START_BIT and L_SER_HOT = '1') then     -- 1/2 start bit
                                L_POSITION      <= X"00";
                            end if;

                            if (STOP_BIT) then                          -- 1/2 stop bit
                                Q_DATA      <= L_BUF(9 downto 2);
                            end if;
```

src/uart_rx.vhd

Reception of a byte already finishes at 3/4 of the stop bit. This is to allow for cumulated baud rate errors of 1/4 bit time (or about 2.5 % baud rate error for 10 bit (1 start, 8 data, and 1 stop bit) transmissions). The received data is stored in **DATA**:

```
                        elsif (STOP_POS) then                       -- 3/4 stop bit
                            L_FLAG      <= L_FLAG xor (L_BUF(9) and not L_BUF(0));
                            L_POSITION      <= X"00";
```

src/uart_rx.vhd

If a greater tolerance against baud rate errors is needed, then one can decrease **STOP_POS** a little, but generally it would be safer to use 2 stop bits on the sender side.

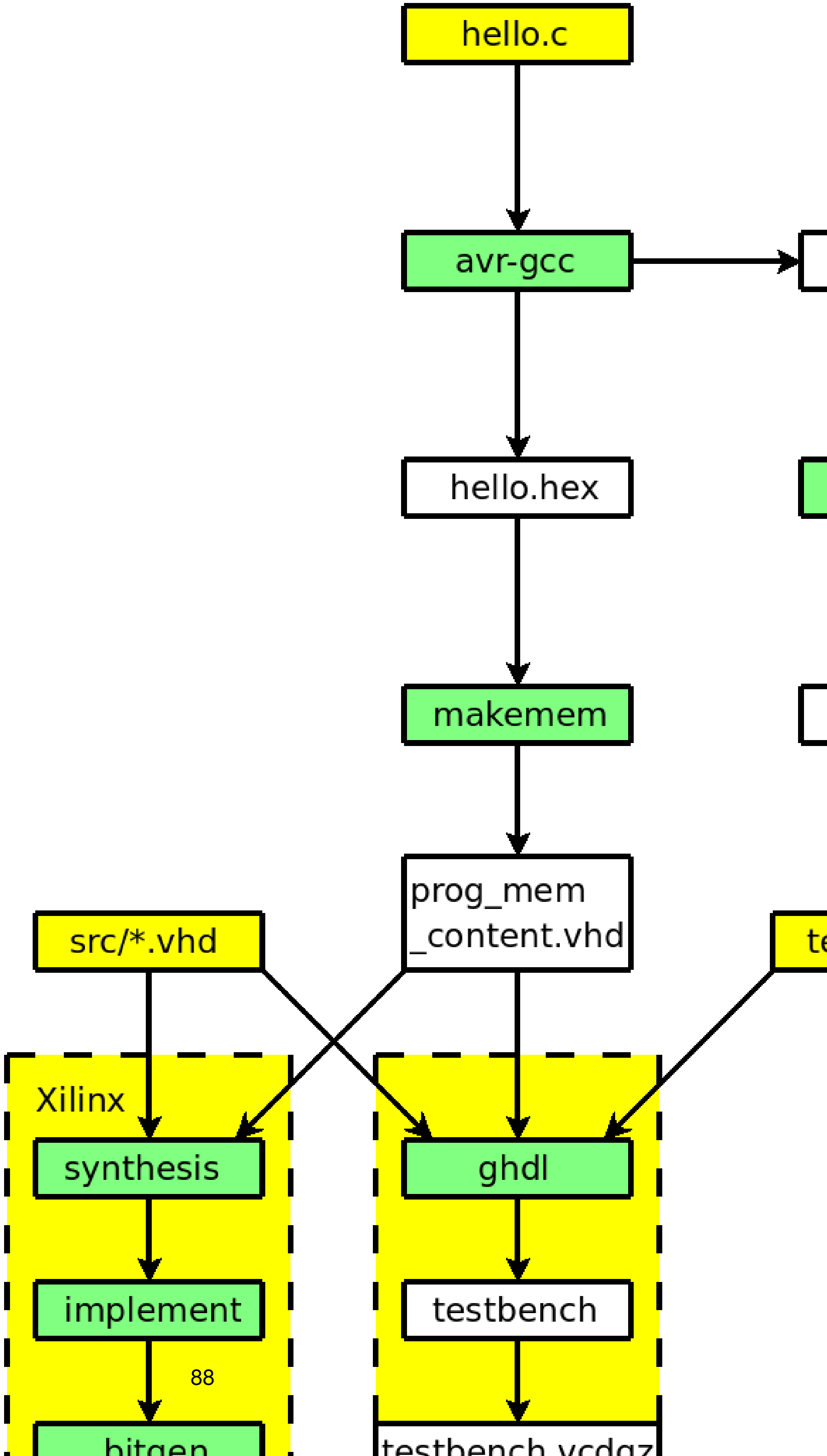
This finalizes the description of the FPGA. We will proceed with the design flow in the next lesson.

9 TOOLCHAIN SETUP

In this lesson we will learn how to set up a toolchain on a Linux box. We will not describe, however, how the tools are downloaded and installed. The installation of a tools is normally described in the documentation that comes with the tool.

Places from where tools can be downloaded were already presented in the first lecture.

The following figure gives an overview of the entire flow. We show source files in yellow, temporary files in white and tools in green.


hello.c
avr-gcc
hello.hex
makemem
prog_mem
_content.vhd
src/*.vhd
Xilinx
synthesis
implement
ghdl
testbench

We start with a C source file **hello.c**. This file is compiled with **avr-gcc**, a **gcc** variant that generates opcodes for the AVR CPU. The compilation produces 2 output files: **hello.lss** and **hello.hex**.

hello.lss is a listing file and may optionally be post-processed by the tool **end_conv** which converts the little-endian format of **hello.lss** into a slightly different format that is more in line with the way gtkwave shows the hex values of signals.

The main purpose of the compilation is to produce **hello.hex**. **hello.hex** contains the opcodes produced from **hello.c** in Intel-Hex format.

hello.hex is then fed into **make_mem**. **make_mem** is a tool that converts the Intel-Hex format into VHDL constants. These constants are used to initialize the block RAM modules of the program memory. The output of **make_mem** is **memory_content.vhd** (which, as you certainly remember, was included by **prog_mem.vhd**).

At this point, there are two possible ways to proceed. You could do a functional simulation or a timing simulation.

8.1 Functional Simulation

Initially you will be concerned mostly with the functional simulation. On this branch you debug the VHDL code until it looks functionally OK. In order to perform the functional simulation, you need 3 sorts of VHDL files:

1. The VHLD source files that were discussed in the previous lessons,
2. the **memory_content.vhd** just described, and
3. a testbench that mimics the board containing the FPGA to be (**test_tb.vhd**, and a VHDL implementation of device specific components used (in our case this is only **RAMB4_S4_S4.vhd**). Both files are provided in the directory called **test**.

All these VHDL files are then processed by **ghdl**. **ghdl** produces a single output file **testbench** in directory **simu**. **testbench** is an executable file. **testbench** is then run in order to produces a gzip'ed **vcd** (value change dump) file called **testbench.vcdgz**.

The last step is visualize **testbench.vcdgz** by means of the tool **gtkwave**. **gtkwave** is similar to the **ModelSim** provided by Xilinx, but it has two advantages: it does not bother the user with licence installations (even in the "free" versions provided by Xilinx) and it runs under Linux. There are actually more advantages of the **ghdl/gtkwave** combination; after having used both tools in the past the author definitely prefers **ghdl/gtkwave**.

An example output of the functional simulation that shows the operation our CPU:

We can compare the CPU signals shown with the assembler code being executed. The CPU is executing inside the C function **uart_puts()**:

```
      00000095: (uart_puts):
        95:   01AC            movw    r20, r24
        96:   C003            rjmp     0x9A           ; 0x134
        97:   9B5D            sbis    0x0b, 5 ; 11
        98:   CFFE            rjmp     0x97           ; 0x12e
        99:   B92C            out     0x0c, r18       ; 12
        9A:   01FC            movw    r30, r24
        9B:   9601            adiw    r24, 0x01       ; 1
        9C:   9124            lpm     r18, Z
        9D:   2322            and     r18, r18
        9E:   F7C1            brne     0x97           ; 0x12e
        9F:   1B84            sub     r24, r20
        A0:   0B95            sbc     r25, r21
        A1:   9701            sbiw    r24, 0x01       ; 1
        A2:   9508            ret
```

`app/hello.lss1`

8.2 Timing Simulation and FPGA Configuration

After the CPU functions correctly, the design can be fed into the Xilinx toolchain. This toolchain is better described in the documentation that comes with it, so we don't go to too much detail here.

We used Webpack 10.1, which can be downloaded from Xilinx.

The first step is to set up a project in the ISE project navigator with the proper target device. Then the VHDL files in the **src** directory are added to the project. Next the **Synthesize** and **Implementation** steps of the design flow are run.

If this is successful, then we can generate a programming file. There are a number of ways to configure Xilinx FPGAs, and the type of programming file needed depends on the particular way of configuring the device. The board we used for testing the CPU had a serial PROM and therefore we generated a programming file for the serial PROM on the board. The FPGA would then load from the PROM on start-up. Other ways of configuring the device are via JTAG, which is also quite handy during debugging.

The entire build process is a little lengthy (and the devil is known to hide in the details). We therefore go through the entire design flow in a step-by-step fashion.

8.3 Downloading and Building the Tools

- Download and install **ghdl**.
- Download and install **gtkwave**.
- Download and install the Xilinx toolchain.
- Build the **make_mem** tool. The source is this:

```
#include "assert.h"
#include "stdio.h"
#include "stdint.h"
#include "string.h"

const char * hex_file = 0;
const char * vhdl_file = 0;

uint8_t buffer[0x10000];

//-----------------------------------------------------------------------------
uint32_t
get_byte(const char *  cp)
{
uint32_t value;
const char cc[3] = { cp[0], cp[1], 0 };
const int cnt = sscanf(cc, "%X", &value);
   assert(cnt == 1);
   return value;
}
//-----------------------------------------------------------------------------
void
read_file(FILE * in)
{
   memset(buffer, 0xFF, sizeof(buffer));
char line[200];
   for (;;)
       {
         const char * s = fgets(line, sizeof(line) - 2, in);
         if (s == 0)   return;
         assert(*s++ == ':');
         const uint32_t len     = get_byte(s);
         const uint32_t ah      = get_byte(s + 2);
         const uint32_t al      = get_byte(s + 4);
         const uint32_t rectype = get_byte(s + 6);
         const char * d = s + 8;
         const uint32_t addr = ah      <<8 | al;

         uint32_t csum = len + ah + al + rectype;
         assert((addr + len)       <= 0x10000);
         for (uint32_t l = 0; l      <len; ++l)
             {
               const uint32_t byte = get_byte(d);
               d += 2;
               buffer[addr + l] = byte;
               csum += byte;
             }

         csum = 0xFF & -csum;
         const uint32_t sum = get_byte(d);
         assert(sum == csum);
       }
}
//-----------------------------------------------------------------------------
void
write_vector(FILE * out, bool odd, uint32_t mem, uint32_t v)
{
const uint8_t * base = buffer;

```

```
      // total memory is 2 even bytes, 2 odd bytes, 2 even bytes, ...
      //
      if (odd)   base += 2;

      // total memory is 4 kByte organized into 8 memories.
      // thus each of the 16 vectors covers 256 bytes.
      //
      base += v*256;

      // memories 0 and 1 are the low byte of the opcode while
      // memories 2 and 3 are the high byte.
      //
      if (mem >= 2)   ++base;

   const char * px = odd ? "po" : "pe";
      fprintf(out, "constant %s_%u_%2.2X : BIT_VECTOR := X\"", px, mem, v);
      for (int32_t d = 63; d >= 0; --d)
          {
            uint32_t q = base[4*d];
            if (mem & 1)   q >>= 4;     // high nibble
            else           q &= 0x0F;   // low nibble
            fprintf(out, "%X", q);
          }

      fprintf(out, "\";\r\n");
   }
   //-----------------------------------------------------------------------------
   void
   write_mem(FILE * out, bool odd, uint32_t mem)
   {
   const char * px = odd ? "po" : "pe";

      fprintf(out, "-- content of %s_%u --------------------------------------"
                   "--------------------------------------------\r\n", px, mem);

      for (uint32_t v = 0; v       <16; ++v)
          write_vector(out, odd, mem, v);

      fprintf(out, "\r\n");
   }
   //-----------------------------------------------------------------------------
   void
   write_file(FILE * out)
   {
      fprintf(out,
   "\r\n"
   "library IEEE;\r\n"
   "use IEEE.STD_LOGIC_1164.all;\r\n"
   "\r\n"
   "package prog_mem_content is\r\n"
   "\r\n");

      for (uint32_t m = 0; m       <4; ++m)
          write_mem(out, false, m);

      for (uint32_t m = 0; m       <4; ++m)
          write_mem(out, true,  m);

      fprintf(out,
   "end prog_mem_content;\r\n"
   "\r\n");
   }
```

```
//-----------------------------------------------------------------------------
int
main(int argc, char * argv[])
{
   if (argc > 1)   hex_file = argv[1];
   if (argc > 2)   vhdl_file = argv[2];

FILE * in = stdin;
   if (hex_file)   in = fopen(hex_file, "r");
   assert(in);
   read_file(in);
   fclose(in);

FILE * out = stdout;
   if (vhdl_file)   out = fopen(vhdl_file, "w");
   write_file(out);
   assert(out);
}
//-----------------------------------------------------------------------------
```

tools/make_mem.cc

The command to build the tool is:

```
# Build makemem.
g++ -o make_mem make_mem.cc
```

- Build the **end_conv** tool. The source is this:

```
#include "assert.h"
#include "ctype.h"
#include "stdio.h"
#include "string.h"

//-----------------------------------------------------------------------------
int
main(int argc, const char * argv)
{
char buffer[2000];
int pc, val, val2;

   for (;;)
       {
         char * s = fgets(buffer, sizeof(buffer) - 2, stdin);
         if (s == 0)   return 0;

         // map lines '  xx:' and 'xxxxxxxx; to 2* the hex value.
         //
```

```
          if (
              (isxdigit(s[0]) || s[0] == ' ') &&
              (isxdigit(s[1]) || s[1] == ' ') &&
              (isxdigit(s[2]) || s[2] == ' ') &&
               isxdigit(s[3]) && s[4] == ':')   // '  xx:'
             {
               assert(1 == sscanf(s, " %x:", &pc));
               if (pc & 1)        printf("%4X+:", pc/2);
               else               printf("%4X:", pc/2);
               s += 5;
             }
          else if (isxdigit(s[0]) && isxdigit(s[1]) && isxdigit(s[2]) &&
                   isxdigit(s[3]) && isxdigit(s[4]) && isxdigit(s[5]) &&
                   isxdigit(s[6]) && isxdigit(s[7]))             // 'xxxxxxxx'
             {
               assert(1 == sscanf(s, "%x", &pc));
               if (pc & 1)   printf("%8.8X+:", pc/2);
               else          printf("%8.8X:", pc/2);
               s += 8;
             }
          else                                // other: copy verbatim
             {
               printf("%s", s);
               continue;
             }

           while (isblank(*s))   printf("%c", *s++);

           // endian swap.
           //
           while (isxdigit(s[0]) &&
                  isxdigit(s[1]) &&
                           s[2] == ' ' &&
                  isxdigit(s[3]) &&
                  isxdigit(s[4]) &&
                           s[5] == ' ')
              {
               assert(2 == sscanf(s, "%x %x ", &val, &val2));
               printf("%2.2X%2.2X  ", val2, val);
               s += 6;
              }

          char * s1 = strstr(s, ".+");
          char * s2 = strstr(s, ".-");
          if (s1)
             {
               assert(1 == sscanf(s1 + 2, "%d", &val));
               assert((val & 1) == 0);
               sprintf(s1, " 0x%X", (pc + val)/2 + 1);
               printf(s);
               s = s1 + strlen(s1) + 1;
             }
          else if (s2)
             {
               assert(1 == sscanf(s2 + 2, "%d", &val));
               assert((val & 1) == 0);
               sprintf(s2, " 0x%X", (pc - val)/2 + 1);
               printf(s);
               s = s2 + strlen(s2) + 1;
             }

          printf("%s", s);
```

```
            }
    }
    //-----------------------------------------------------------------------------
```

tools/end_conv.cc

The command to build the tool is:

```
# Build end_conv.
g++ -o end_conv end_conv.cc
```

8.4 Preparing the Memory Content

We write a program **hello.c** that prints "Hello World" to the serial line.

The source is this:

```
#include "stdint.h"
#include "avr/io.h"
#include "avr/pgmspace.h"

#undef F_CPU
#define F_CPU 25000000UL
#include "util/delay.h"


     //----------------------------------------------------------------------//
    //                                                                      //
   //   print char cc on UART.                                             //
  //    return number of chars printed (i.e. 1).                          //
 //                                                                      //
//----------------------------------------------------------------------//
uint8_t
uart_putc(uint8_t cc)
{
    while ((UCSRA & (1      <<UDRE)) == 0)      ;
    UDR = cc;
    return 1;
}

     //----------------------------------------------------------------------//
    //                                                                      //
   //   print char cc on 7 segment display.                                //
  //    return number of chars printed (i.e. 1).                          //
 //                                                                      //
//----------------------------------------------------------------------//
```

```
// The segments of the display are encoded like this:
//
//
//      segment     PORT B
//      name        Bit number
//      ----A----   ----0----
//      |       |   |       |
//      F       B   5       1
//      |       |   |       |
//      ----G----   ----6----
//      |       |   |       |
//      E       C   4       2
//      |       |   |       |
//      ----D----   ----3----
//
//-----------------------------------------------------------------------------

#define SEG7(G, F, E, D, C, B, A)   (~(G      <
```

app/hello.c

The commands to create **hello.hex** and **hello.css** are:

```
# Compile and link hello.c.
avr-gcc -Wall -Os -fpack-struct -fshort-enums -funsigned-char -funsigned-bitfields -mmcu=atmega8 \
    -DF_CPU=33333333UL -MMD -MP -MF"main.d" -MT"main.d" -c -o"main.o" "main.c"
avr-gcc -Wl,-Map,AVR_FPGA.map -mmcu=atmega8 -o"AVR_FPGA.elf"  ./main.o

# Create an opcode listing.
avr-objdump -h -S AVR_FPGA.elf  >"AVR_FPGA.lss"

# Create intel hex file.
avr-objcopy -R .eeprom -O ihex AVR_FPGA.elf  "AVR_FPGA.hex"
```

Create **hello.css1**, a better readable from of **hello.css**:

```
# Create hello.css1.
./end_conv <hello.css > hello.css1
```

Create **prog_mem_content.vhd**.

```
# Create prog_mem_content.vhd.
```

```
./make_mem <hello.hex > src/prog_mem_content.vhd
```

8.5 Performing the Functional Simulation

8.5.1 Preparing a Testbench

We prepare a testbench in which we instantiate the top-level FPGA design of the CPU. The test bench provides a clock signal and a reset signal for the CPU:

```
-------------------------------------------------------------------------------
--
-- Copyright (C) 2009, 2010 Dr. Juergen Sauermann
--
--  This code is free software: you can redistribute it and/or modify
--  it under the terms of the GNU General Public License as published by
--  the Free Software Foundation, either version 3 of the License, or
--  (at your option) any later version.
--
--  This code is distributed in the hope that it will be useful,
--  but WITHOUT ANY WARRANTY; without even the implied warranty of
--  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
--  GNU General Public License for more details.
--
--  You should have received a copy of the GNU General Public License
--  along with this code (see the file named COPYING).
--  If not, see http://www.gnu.org/licenses/.
--
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
--
-- Module Name:    alu - Behavioral
-- Create Date:    16:47:24 12/29/2009
-- Description:    arithmetic logic unit of a CPU
--
-------------------------------------------------------------------------------
--
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity testbench is
end testbench;

architecture Behavioral of testbench is

component avr_fpga
    port (  I_CLK_100   : in  std_logic;
            I_SWITCH    : in  std_logic_vector(9 downto 0);
            I_RX        : in  std_logic;

            Q_7_SEGMENT : out std_logic_vector(6 downto 0);
            Q_LEDS      : out std_logic_vector(3 downto 0);
```

```
            Q_TX        : out std_logic);
end component;

signal L_CLK_100            : std_logic;
signal L_LEDS               : std_logic_vector(3 downto 0);
signal L_7_SEGMENT          : std_logic_vector(6 downto 0);
signal L_RX                 : std_logic;
signal L_SWITCH             : std_logic_vector(9 downto 0);
signal L_TX                 : std_logic;

signal  L_CLK_COUNT         : integer := 0;

begin

    fpga: avr_fpga
    port map(   I_CLK_100   => L_CLK_100,
                I_SWITCH    => L_SWITCH,
                I_RX        => L_RX,

                Q_LEDS      => L_LEDS,
                Q_7_SEGMENT => L_7_SEGMENT,
                Q_TX        => L_TX);

    process -- clock process for CLK_100,
    begin
        clock_loop : loop
            L_CLK_100      <= transport '0';
            wait for 5 ns;

            L_CLK_100      <= transport '1';
            wait for 5 ns;
        end loop clock_loop;
    end process;

    process(L_CLK_100)
    begin
        if (rising_edge(L_CLK_100)) then
            case L_CLK_COUNT is
                when 0 => L_SWITCH      <= "0011100000";   L_RX <= '0';
                when 2 => L_SWITCH(9 downto 8)      <= "11";
                when others =>
            end case;
            L_CLK_COUNT      <= L_CLK_COUNT + 1;
        end if;
    end process;
end Behavioral;

```

test/test_tb.vhd

8.5.2 Defining Memory Modules

We also need a VHDL file that implements the Xilinx primitives that we use. This is only one: the memory module RAMB4_S4_S4:

```
-------------------------------------------------------------------------------
--
-- Copyright (C) 2009, 2010 Dr. Juergen Sauermann
--
--  This code is free software: you can redistribute it and/or modify
--  it under the terms of the GNU General Public License as published by
--  the Free Software Foundation, either version 3 of the License, or
--  (at your option) any later version.
--
--  This code is distributed in the hope that it will be useful,
--  but WITHOUT ANY WARRANTY; without even the implied warranty of
--  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
--  GNU General Public License for more details.
--
--  You should have received a copy of the GNU General Public License
--  along with this code (see the file named COPYING).
--  If not, see http://www.gnu.org/licenses/.
--
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
--
-- Module Name:    prog_mem - Behavioral
-- Create Date:    14:09:04 10/30/2009
-- Description:    a block memory module
--
-------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity RAMB4_S4_S4 is
    generic(INIT_00 : bit_vector := X"00000000000000000000000000000000"
                                 &  "00000000000000000000000000000000";
            INIT_01 : bit_vector := X"00000000000000000000000000000000"
                                 & X"00000000000000000000000000000000";
            INIT_02 : bit_vector := X"00000000000000000000000000000000"
                                 & X"00000000000000000000000000000000";
            INIT_03 : bit_vector := X"00000000000000000000000000000000"
                                 & X"00000000000000000000000000000000";
            INIT_04 : bit_vector := X"00000000000000000000000000000000"
                                 & X"00000000000000000000000000000000";
            INIT_05 : bit_vector := X"00000000000000000000000000000000"
                                 & X"00000000000000000000000000000000";
            INIT_06 : bit_vector := X"00000000000000000000000000000000"
                                 & X"00000000000000000000000000000000";
            INIT_07 : bit_vector := X"00000000000000000000000000000000"
                                 & X"00000000000000000000000000000000";
            INIT_08 : bit_vector := X"00000000000000000000000000000000"
                                 & X"00000000000000000000000000000000";
            INIT_09 : bit_vector := X"00000000000000000000000000000000"
                                 & X"00000000000000000000000000000000";
            INIT_0A : bit_vector := X"00000000000000000000000000000000"
                                 & X"00000000000000000000000000000000";
            INIT_0B : bit_vector := X"00000000000000000000000000000000"
                                 & X"00000000000000000000000000000000";
            INIT_0C : bit_vector := X"00000000000000000000000000000000"
                                 & X"00000000000000000000000000000000";
            INIT_0D : bit_vector := X"00000000000000000000000000000000"
                                 & X"00000000000000000000000000000000";
```

```
            INIT_0E : bit_vector := X"00000000000000000000000000000000"
                                  & X"00000000000000000000000000000000";
            INIT_0F : bit_vector := X"00000000000000000000000000000000"
                                  & X"00000000000000000000000000000000");

    port(   ADDRA   : in  std_logic_vector(9 downto 0);
            ADDRB   : in  std_logic_vector(9 downto 0);
            CLKA    : in  std_ulogic;
            CLKB    : in  std_ulogic;
            DIA     : in  std_logic_vector(3 downto 0);
            DIB     : in  std_logic_vector(3 downto 0);
            ENA     : in  std_ulogic;
            ENB     : in  std_ulogic;
            RSTA    : in  std_ulogic;
            RSTB    : in  std_ulogic;
            WEA     : in  std_ulogic;
            WEB     : in  std_ulogic;

            DOA     : out std_logic_vector(3 downto 0);
            DOB     : out std_logic_vector(3 downto 0));
end RAMB4_S4_S4;

architecture Behavioral of RAMB4_S4_S4 is

function cv(A : bit) return std_logic is
begin
   if (A = '1') then return '1';
   else              return '0';
   end if;
end;

function cv1(A : std_logic) return bit is
begin
   if (A = '1') then return '1';
   else              return '0';
   end if;
end;

signal DATA : bit_vector(4095 downto 0) :=
    INIT_0F & INIT_0E & INIT_0D & INIT_0C & INIT_0B & INIT_0A & INIT_09 & INIT_08 &
    INIT_07 & INIT_06 & INIT_05 & INIT_04 & INIT_03 & INIT_02 & INIT_01 & INIT_00;

begin

    process(CLKA, CLKB)
    begin
        if (rising_edge(CLKA)) then
            if (ENA = '1') then
                DOA(3)      <= cv(DATA(conv_integer(ADDRA & "11")));
                DOA(2)      <= cv(DATA(conv_integer(ADDRA & "10")));
                DOA(1)      <= cv(DATA(conv_integer(ADDRA & "01")));
                DOA(0)      <= cv(DATA(conv_integer(ADDRA & "00")));
                if (WEA = '1') then
                    DATA(conv_integer(ADDRA & "11"))      <= cv1(DIA(3));
                    DATA(conv_integer(ADDRA & "10"))      <= cv1(DIA(2));
                    DATA(conv_integer(ADDRA & "01"))      <= cv1(DIA(1));
                    DATA(conv_integer(ADDRA & "00"))      <= cv1(DIA(0));
                end if;
           end if;
        end if;

        if (rising_edge(CLKB)) then
```

```
                if (ENB = '1') then
                    DOB(3)      <= cv(DATA(conv_integer(ADDRB & "11")));
                    DOB(2)      <= cv(DATA(conv_integer(ADDRB & "10")));
                    DOB(1)      <= cv(DATA(conv_integer(ADDRB & "01")));
                    DOB(0)      <= cv(DATA(conv_integer(ADDRB & "00")));
                    if (WEB = '1') then
                        DATA(conv_integer(ADDRB & "11"))      <= cv1(DIB(3));
                        DATA(conv_integer(ADDRB & "10"))      <= cv1(DIB(2));
                        DATA(conv_integer(ADDRB & "01"))      <= cv1(DIB(1));
                        DATA(conv_integer(ADDRB & "00"))      <= cv1(DIB(0));
                    end if;
                end if;
            end if;
        end process;

    end Behavioral;

```

test/RAMB4_S4_S4.vhd

8.5.3 Creating the testbench executable

We assume the following file structure:

- a **test** directory that contains the testbench (**test_tb.vhd**) and the memory module (**RAMB4_S4_S4.vhd**).
- a **src** directory that contains all other VHDL files.
- a **simu** directory (empty).
- A **Makefile** like this:

```
PROJECT=avr_core

# the vhdl source files (except testbench)
#
FILES           += src/*.vhd

# the testbench sources and binary.
#
SIMFILES        = test/test_tb.vhd test/RAMB4_S4_S4.vhd
SIMTOP          = testbench

# When to stop the simulation
#
# GHDL_SIM_OPT  = --assert-level=error
GHDL_SIM_OPT    = --stop-time=40us

SIMDIR          = simu

FLAGS           = --ieee=synopsys --warn-no-vital-generic -fexplicit --std=93c

all:
        make compile
        make run 2>& 1 | grep -v std_logic_arith
```

```
		make view

compile:
		@mkdir -p simu
		@echo -----------------------------------------------------------------
		ghdl -i $(FLAGS) --workdir=simu --work=work $(SIMFILES) $(FILES)
		@echo
		@echo -----------------------------------------------------------------
		ghdl -m $(FLAGS) --workdir=simu --work=work $(SIMTOP)
		@echo
		@mv $(SIMTOP) simu/$(SIMTOP)

run:
		@$(SIMDIR)/$(SIMTOP) $(GHDL_SIM_OPT) --vcdgz=$(SIMDIR)/$(SIMTOP).vcdgz

view:
		gunzip --stdout $(SIMDIR)/$(SIMTOP).vcdgz | gtkwave --vcd gtkwave.save

clean:
		ghdl --clean --workdir=simu

```

Makefile

Then

```
# Run the functional simulation.
make
```

It will take a moment, but then a **gtkwave** window like the one shown earlier in this lesson will appear. It my look a little different due due to different default settings (like background color). In that window you can add new signals from the design that you would like to investigate, remove signals you are not interested in, and so on. At the first time, no signals will be shown; you can add some by selecting a component instance at the right, selecting a signal in that component, and then pushing the **append** button on the right.

The **make** command has actually made 3 things:

- make compile (compile the VHLD files)
- make run (run the simulation), and
- make view

The first two steps (which took most of the total time) need only be run after changes to the VHDL files.

8.6 Building the Design

When the functional simulation looks OK, it is time to implement the design and check the timing. We describe this only briefly, since the Xilinx documentation of the Xilinx toolchain is a much better source of information.

8.6.1 Creating an UCF file

Before implementing the design, we need an **UCF** file. That file describes timing requirements, pin properties (like pull-ups for our DIP switch), and pin-to-signal mappings:

```
NET     I_CLK_100       PERIOD = 10 ns;
NET     L_CLK           PERIOD = 35 ns;

NET     I_CLK_100       TNM_NET = I_CLK_100;
NET     L_CLK           TNM_NET = L_CLK;

NET     I_CLK_100       LOC = AA12;
NET     I_RX            LOC = M3;
NET     Q_TX            LOC = M4;

# 7 segment LED display
#
NET     Q_7_SEGMENT        LOC = V3;
NET     Q_7_SEGMENT        LOC = V4;
NET     Q_7_SEGMENT        LOC = W3;
NET     Q_7_SEGMENT        LOC = T4;
NET     Q_7_SEGMENT        LOC = T3;
NET     Q_7_SEGMENT        LOC = U3;
NET     Q_7_SEGMENT        LOC = U4;

# single LEDs
#
NET     Q_LEDS             LOC = N1;
NET     Q_LEDS             LOC = N2;
NET     Q_LEDS             LOC = P1;
NET     Q_LEDS             LOC = P2;

# DIP switch(0 ... 7) and two pushbuttons (8, 9)
#
NET     I_SWITCH           LOC = H2;
NET     I_SWITCH           LOC = H1;
NET     I_SWITCH           LOC = J2;
NET     I_SWITCH           LOC = J1;
NET     I_SWITCH           LOC = K2;
NET     I_SWITCH           LOC = K1;
NET     I_SWITCH           LOC = L2;
NET     I_SWITCH           LOC = L1;
NET     I_SWITCH           LOC = R1;
NET     I_SWITCH           LOC = R2;

NET     I_SWITCH           PULLUP;

```

src/avr_fpga.ucf

8.6.2 Synthesis and Implementation

- Start the ISE project manager and open a new project with the desired FPGA device.
- Add the VHDL files and the **UCF** file in the **src** directory to the project (Project->Add Source).
- Synthesize and implement the design (Process->Implement top Module).

This generates a number of reports, netlists, and other files. There should be no errors. There will be warnings though, including timing constraints that are not met.

It is important to understand the reason for each warning. Warnings often point to faults in the design.

The next thing to check is the timing reports. We were lucky:

```
#Timing report fragment:
================================================================================
Timing constraint: NET "L_CLK" PERIOD = 35 ns HIGH 50%;

 676756190 paths analyzed, 2342 endpoints analyzed, 0 failing endpoints
 0 timing errors detected. (0 setup errors, 0 hold errors)
 Minimum period is  34.981ns.
--------------------------------------------------------------------------------

================================================================================
Timing constraint: NET "I_CLK_100_BUFGP/IBUFG" PERIOD = 10 ns HIGH 50%;

 19 paths analyzed, 11 endpoints analyzed, 0 failing endpoints
 0 timing errors detected. (0 setup errors, 0 hold errors)
 Minimum period is   3.751ns.
--------------------------------------------------------------------------------


All constraints were met.
```

This tells us that we have enough slack on the crystal CLK_100 signal (8.048ns would allow for up to 124 MHz). We had specified a period of 35 ns irequirement for the CPU clock:

```
  2     NET     L_CLK           PERIOD = 35 ns;
```

```
src/avr_fpga.ucf
```

The CPU runs at 25 MHz, or 40 ns. The 35 ns come from the 40 ms minus a slack of 5 ns. With some tweaking of optimization options, we could have reached 33 MHz, but then the slack would have been pretty small.

However, we rather stay on th safe side.

8.7 Creating a Programming File

Next we double-click "Generate Programming file" in the ISE project navigator. This generates a file **avr_fpga.bit** in the project directory. This can also be run from a Makefile or from the command line (the command is **bitgen**).

8.8 Configuring the FPGA

At this point, we have the choice between configuring the FPGA directly via JTAG, or flashing an EEPROM and then loading the FPGA from the EEPROM.

8.8.1 Configuring the FPGA via JTAG Boundary Scan

Configuring the FPGA can be done with the Xilinx tool called **impact**. The file needed by **impact** is **avr_fpga.bit** from above. The configuration loaded via JTAG will be lost when the FPGA looses power.

Choose "Boundary Scan" in **impact**, select the FPGA and follow the instructions.

8.8.2 Flashing PROMs

In theory this can also be done from ISE. In practice it could (and actually did) happen that the programming cable (I use an old parallel 3 cable) is not detected by impact.

Before flashing the PROM, the **avr_fpga.bit** from the previous step needs to translated into a format suitable for the PROM. My PROM is of the serial variety, so I start **impact**, choose "PROM File Formatter" and follow the instructions.

After converting **avr_fpga.bit** into, for example, **avr_fpga.mcs**, the PROM can be flashed. Like before choose "Boundary Scan" in #impact. This time, however, you select the PROM and not the FPGA, and follow the instructions.

This concludes the description of the design flow and also of the CPU. The remaining lessons contain the complete listings of all sources files discussed in this lectures.

Thank you very much for your attention.

10 LISTING OF alu.vhd

```
--------------------------------------------------------------------------------
--
-- Copyright (C) 2009, 2010 Dr. Juergen Sauermann
--
--  This code is free software: you can redistribute it and/or modify
--  it under the terms of the GNU General Public License as published by
--  the Free Software Foundation, either version 3 of the License, or
--  (at your option) any later version.
--
--  This code is distributed in the hope that it will be useful,
--  but WITHOUT ANY WARRANTY; without even the implied warranty of
--  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
--  GNU General Public License for more details.
--
--  You should have received a copy of the GNU General Public License
--  along with this code (see the file named COPYING).
--  If not, see http://www.gnu.org/licenses/.
--
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--
-- Module Name:    alu - Behavioral
-- Create Date:    13:51:24 11/07/2009
-- Description:    arithmetic logic unit of a CPU
--
--------------------------------------------------------------------------------
--
library IEEE;
use IEEE.std_logic_1164.ALL;
use IEEE.std_logic_ARITH.ALL;
use IEEE.std_logic_UNSIGNED.ALL;

use work.common.ALL;

entity alu is
    port (  I_ALU_OP    : in  std_logic_vector( 4 downto 0);
            I_BIT       : in  std_logic_vector( 3 downto 0);
            I_D         : in  std_logic_vector(15 downto 0);
            I_D0        : in  std_logic;
            I_DIN       : in  std_logic_vector( 7 downto 0);
            I_FLAGS     : in  std_logic_vector( 7 downto 0);
            I_IMM       : in  std_logic_vector( 7 downto 0);
            I_PC        : in  std_logic_vector(15 downto 0);
            I_R         : in  std_logic_vector(15 downto 0);
            I_R0        : in  std_logic;
            I_RSEL      : in  std_logic_vector( 1 downto 0);

            Q_FLAGS     : out std_logic_vector( 9 downto 0);
            Q_DOUT      : out std_logic_vector(15 downto 0));
end alu;

architecture Behavioral of alu is

function ze(A: std_logic_vector(7 downto 0)) return std_logic is
begin
    return not (A(0) or A(1) or A(2) or A(3) or
                A(4) or A(5) or A(6) or A(7));
end;
```

```

    function cy(D, R, S: std_logic) return std_logic is
    begin
        return (D and R) or (D and not S) or (R and not S);
    end;

    function ov(D, R, S: std_logic) return std_logic is
    begin
        return (D and R and (not S)) or ((not D) and (not R) and S);
    end;

    function si(D, R, S: std_logic) return std_logic is
    begin
        return S xor ov(D, R, S);
    end;

    signal L_ADC_DR     : std_logic_vector( 7 downto 0);    -- D + R + Carry
    signal L_ADD_DR     : std_logic_vector( 7 downto 0);    -- D + R
    signal L_ADIW_D     : std_logic_vector(15 downto 0);    -- D + IMM
    signal L_AND_DR     : std_logic_vector( 7 downto 0);    -- D and R
    signal L_ASR_D      : std_logic_vector( 7 downto 0);    -- (signed D) >> 1
    signal L_D8         : std_logic_vector( 7 downto 0);    -- D(7 downto 0)
    signal L_DEC_D      : std_logic_vector( 7 downto 0);    -- D - 1
    signal L_DOUT       : std_logic_vector(15 downto 0);
    signal L_INC_D      : std_logic_vector( 7 downto 0);    -- D + 1
    signal L_LSR_D      : std_logic_vector( 7 downto 0);    -- (unsigned) D >> 1
    signal L_MASK_I     : std_logic_vector( 7 downto 0);    -- 1      <<IMM
    signal L_NEG_D      : std_logic_vector( 7 downto 0);    -- 0 - D
    signal L_NOT_D      : std_logic_vector( 7 downto 0);    -- 0 not D
    signal L_OR_DR      : std_logic_vector( 7 downto 0);    -- D or R
    signal L_PROD       : std_logic_vector(17 downto 0);    -- D * R
    signal L_R8         : std_logic_vector( 7 downto 0);    -- odd or even R
    signal L_RI8        : std_logic_vector( 7 downto 0);    -- R8 or IMM
    signal L_RBIT       : std_logic;
    signal L_SBIW_D     : std_logic_vector(15 downto 0);    -- D - IMM
    signal L_ROR_D      : std_logic_vector( 7 downto 0);    -- D rotated right
    signal L_SBC_DR     : std_logic_vector( 7 downto 0);    -- D - R - Carry
    signal L_SIGN_D     : std_logic;
    signal L_SIGN_R     : std_logic;
    signal L_SUB_DR     : std_logic_vector( 7 downto 0);    -- D - R
    signal L_SWAP_D     : std_logic_vector( 7 downto 0);    -- D swapped
    signal L_XOR_DR     : std_logic_vector( 7 downto 0);    -- D xor R

    begin

        dinbit: process(I_DIN, I_BIT(2 downto 0))
        begin
            case I_BIT(2 downto 0) is
                when "000"  => L_RBIT       <= I_DIN(0);   L_MASK_I <= "00000001";
                when "001"  => L_RBIT      <= I_DIN(1);   L_MASK_I <= "00000010";
                when "010"  => L_RBIT      <= I_DIN(2);   L_MASK_I <= "00000100";
                when "011"  => L_RBIT      <= I_DIN(3);   L_MASK_I <= "00001000";
                when "100"  => L_RBIT      <= I_DIN(4);   L_MASK_I <= "00010000";
                when "101"  => L_RBIT      <= I_DIN(5);   L_MASK_I <= "00100000";
                when "110"  => L_RBIT      <= I_DIN(6);   L_MASK_I <= "01000000";
                when others => L_RBIT      <= I_DIN(7);   L_MASK_I <= "10000000";
            end case;
        end process;

        process(L_ADC_DR, L_ADD_DR, L_ADIW_D, I_ALU_OP, L_AND_DR, L_ASR_D,
                I_BIT, I_D, L_D8, L_DEC_D, I_DIN, I_FLAGS, I_IMM, L_MASK_I,
                L_INC_D, L_LSR_D, L_NEG_D, L_NOT_D, L_OR_DR, I_PC, L_PROD,
```

```
                I_R, L_RI8, L_RBIT, L_ROR_D, L_SBIW_D, L_SUB_DR, L_SBC_DR,
                L_SIGN_D, L_SIGN_R, L_SWAP_D, L_XOR_DR)
        begin
            Q_FLAGS(9)      <= L_RBIT xor not I_BIT(3);     -- DIN[BIT] = BIT[3]
            Q_FLAGS(8)      <= ze(L_SUB_DR);                -- D == R for CPSE
            Q_FLAGS(7 downto 0)      <= I_FLAGS;
            L_DOUT      <= X"0000";

            case I_ALU_OP is
                when ALU_ADC =>
                    L_DOUT      <= L_ADC_DR & L_ADC_DR;
                    Q_FLAGS(0)      <= cy(L_D8(7), L_RI8(7), L_ADC_DR(7));   -- Carry
                    Q_FLAGS(1)      <= ze(L_ADC_DR);                         -- Zero
                    Q_FLAGS(2)      <= L_ADC_DR(7);                          -- Negative
                    Q_FLAGS(3)      <= ov(L_D8(7), L_RI8(7), L_ADC_DR(7));   -- Overflow
                    Q_FLAGS(4)      <= si(L_D8(7), L_RI8(7), L_ADC_DR(7));   -- Signed
                    Q_FLAGS(5)      <= cy(L_D8(3), L_RI8(3), L_ADC_DR(3));   -- Halfcarry

                when ALU_ADD =>
                    L_DOUT      <= L_ADD_DR & L_ADD_DR;
                    Q_FLAGS(0)      <= cy(L_D8(7), L_RI8(7), L_ADD_DR(7));   -- Carry
                    Q_FLAGS(1)      <= ze(L_ADD_DR);                         -- Zero
                    Q_FLAGS(2)      <= L_ADD_DR(7);                          -- Negative
                    Q_FLAGS(3)      <= ov(L_D8(7), L_RI8(7), L_ADD_DR(7));   -- Overflow
                    Q_FLAGS(4)      <= si(L_D8(7), L_RI8(7), L_ADD_DR(7));   -- Signed
                    Q_FLAGS(5)      <= cy(L_D8(3), L_RI8(3), L_ADD_DR(3));   -- Halfcarry

                when ALU_ADIW =>
                    L_DOUT      <= L_ADIW_D;
                    Q_FLAGS(0)      <= L_ADIW_D(15) and not I_D(15);         -- Carry
                    Q_FLAGS(1)      <= ze(L_ADIW_D(15 downto 8)) and
                                   ze(L_ADIW_D(7 downto 0));             -- Zero
                    Q_FLAGS(2)      <= L_ADIW_D(15);                         -- Negative
                    Q_FLAGS(3)      <= I_D(15) and not L_ADIW_D(15);         -- Overflow
                    Q_FLAGS(4)      <= (L_ADIW_D(15) and not I_D(15))
                             xor (I_D(15) and not L_ADIW_D(15));       -- Signed

                when ALU_AND =>
                    L_DOUT      <= L_AND_DR & L_AND_DR;
                    Q_FLAGS(1)      <= ze(L_AND_DR);                         -- Zero
                    Q_FLAGS(2)      <= L_AND_DR(7);                          -- Negative
                    Q_FLAGS(3)      <= '0';                                  -- Overflow
                    Q_FLAGS(4)      <= L_AND_DR(7);                          -- Signed

                when ALU_ASR =>
                    L_DOUT      <= L_ASR_D & L_ASR_D;
                    Q_FLAGS(0)      <= L_D8(0);                              -- Carry
                    Q_FLAGS(1)      <= ze(L_ASR_D);                          -- Zero
                    Q_FLAGS(2)      <= L_D8(7);                              -- Negative
                    Q_FLAGS(3)      <= L_D8(0) xor L_D8(7);                  -- Overflow
                    Q_FLAGS(4)      <= L_D8(0);                              -- Signed

                when ALU_BLD =>     -- copy T flag to DOUT
                    case I_BIT(2 downto 0) is
                        when "000"  => L_DOUT( 0)      <= I_FLAGS(6);
                                       L_DOUT( 8)      <= I_FLAGS(6);
                        when "001"  => L_DOUT( 1)      <= I_FLAGS(6);
                                       L_DOUT( 9)      <= I_FLAGS(6);
                        when "010"  => L_DOUT( 2)      <= I_FLAGS(6);
                                       L_DOUT(10)      <= I_FLAGS(6);
                        when "011"  => L_DOUT( 3)      <= I_FLAGS(6);
                                       L_DOUT(11)      <= I_FLAGS(6);
```

```
                when "100"  => L_DOUT( 4)       <= I_FLAGS(6);
                               L_DOUT(12)       <= I_FLAGS(6);
                when "101"  => L_DOUT( 5)       <= I_FLAGS(6);
                               L_DOUT(13)       <= I_FLAGS(6);
                when "110"  => L_DOUT( 6)       <= I_FLAGS(6);
                               L_DOUT(14)       <= I_FLAGS(6);
                when others => L_DOUT( 7)       <= I_FLAGS(6);
                               L_DOUT(15)       <= I_FLAGS(6);
            end case;

        when ALU_BIT_CS =>  -- copy I_DIN to T flag
            Q_FLAGS(6)       <= L_RBIT xor not I_BIT(3);
            if (I_BIT(3) = '0') then    -- clear
                L_DOUT(15 downto 8)       <= I_DIN and not L_MASK_I;
                L_DOUT( 7 downto 0)       <= I_DIN and not L_MASK_I;
            else                        -- set
                L_DOUT(15 downto 8)       <= I_DIN or L_MASK_I;
                L_DOUT( 7 downto 0)       <= I_DIN or L_MASK_I;
            end if;

        when ALU_COM =>
            L_DOUT      <= L_NOT_D & L_NOT_D;
            Q_FLAGS(0)       <= '1';                                    -- Carry
            Q_FLAGS(1)       <= ze(not L_D8);                           -- Zero
            Q_FLAGS(2)       <= not L_D8(7);                            -- Negative
            Q_FLAGS(3)       <= '0';                                    -- Overflow
            Q_FLAGS(4)       <= not L_D8(7);                            -- Signed

        when ALU_DEC =>
            L_DOUT      <= L_DEC_D & L_DEC_D;
            Q_FLAGS(1)       <= ze(L_DEC_D);                            -- Zero
            Q_FLAGS(2)       <= L_DEC_D(7);                             -- Negative
            if (L_D8 = X"80") then
                Q_FLAGS(3)       <= '1';                                -- Overflow
                Q_FLAGS(4)       <= not L_DEC_D(7);                     -- Signed
            else
                Q_FLAGS(3)       <= '0';                                -- Overflow
                Q_FLAGS(4)       <= L_DEC_D(7);                         -- Signed
            end if;

        when ALU_EOR =>
            L_DOUT      <= L_XOR_DR & L_XOR_DR;
            Q_FLAGS(1)       <= ze(L_XOR_DR);                           -- Zero
            Q_FLAGS(2)       <= L_XOR_DR(7);                            -- Negative
            Q_FLAGS(3)       <= '0';                                    -- Overflow
            Q_FLAGS(4)       <= L_XOR_DR(7);                            -- Signed

        when ALU_INC =>
            L_DOUT      <= L_INC_D & L_INC_D;
            Q_FLAGS(1)       <= ze(L_INC_D);                            -- Zero
            Q_FLAGS(2)       <= L_INC_D(7);                             -- Negative
            if (L_D8 = X"7F") then
                Q_FLAGS(3)       <= '1';                                -- Overflow
                Q_FLAGS(4)       <= not L_INC_D(7);                     -- Signed
            else
                Q_FLAGS(3)       <= '0';                                -- Overflow
                Q_FLAGS(4)       <= L_INC_D(7);                         -- Signed
            end if;

        when ALU_INTR =>
            L_DOUT      <= I_PC;
            Q_FLAGS(7)       <= I_IMM(6);    -- ena/disable interrupts
```

```
                when ALU_LSR  =>
                    L_DOUT      <= L_LSR_D & L_LSR_D;
                    Q_FLAGS(0)      <= L_D8(0);                          -- Carry
                    Q_FLAGS(1)      <= ze(L_LSR_D);                      -- Zero
                    Q_FLAGS(2)      <= '0';                              -- Negative
                    Q_FLAGS(3)      <= L_D8(0);                          -- Overflow
                    Q_FLAGS(4)      <= L_D8(0);                          -- Signed

                when ALU_D_MV_Q =>
                    L_DOUT      <= L_D8 & L_D8;

                when ALU_R_MV_Q =>
                    L_DOUT      <= L_RI8 & L_RI8;

                when ALU_MV_16 =>
                    L_DOUT      <= I_R(15 downto 8) & L_RI8;

                when ALU_MULT =>
                    Q_FLAGS(0)      <= L_PROD(15);                       -- Carry
                    if I_IMM(7) = '0' then              -- MUL
                        L_DOUT      <= L_PROD(15 downto 0);
                        Q_FLAGS(1)      <= ze(L_PROD(15 downto 8))       -- Zero
                                and ze(L_PROD( 7 downto 0));
                    else                                -- FMUL
                        L_DOUT      <= L_PROD(14 downto 0) & "0";
                        Q_FLAGS(1)      <= ze(L_PROD(14 downto 7))       -- Zero
                                and ze(L_PROD( 6 downto 0) & "0");
                    end if;

                when ALU_NEG =>
                    L_DOUT      <= L_NEG_D & L_NEG_D;
                    Q_FLAGS(0)      <= not ze(L_D8);                     -- Carry
                    Q_FLAGS(1)      <= ze(L_NEG_D);                      -- Zero
                    Q_FLAGS(2)      <= L_NEG_D(7);                       -- Negative
                    if (L_D8 = X"80") then
                        Q_FLAGS(3)      <= '1';                          -- Overflow
                        Q_FLAGS(4)      <= not L_NEG_D(7);               -- Signed
                    else
                        Q_FLAGS(3)      <= '0';                          -- Overflow
                        Q_FLAGS(4)      <= L_NEG_D(7);                   -- Signed
                    end if;
                    Q_FLAGS(5)      <= L_D8(3) or L_NEG_D(3);            -- Halfcarry

                when ALU_OR =>
                    L_DOUT      <= L_OR_DR & L_OR_DR;
                    Q_FLAGS(1)      <= ze(L_OR_DR);                      -- Zero
                    Q_FLAGS(2)      <= L_OR_DR(7);                       -- Negative
                    Q_FLAGS(3)      <= '0';                              -- Overflow
                    Q_FLAGS(4)      <= L_OR_DR(7);                       -- Signed

                when ALU_PC_1 =>    -- ICALL, RCALL
                    L_DOUT      <= I_PC + X"0001";

                when ALU_PC_2 =>    -- CALL
                    L_DOUT      <= I_PC + X"0002";

                when ALU_ROR =>
                    L_DOUT      <= L_ROR_D & L_ROR_D;
                    Q_FLAGS(1)      <= ze(L_ROR_D);                      -- Zero
                    Q_FLAGS(2)      <= I_FLAGS(0);                       -- Negative
                    Q_FLAGS(3)      <= I_FLAGS(0) xor L_D8(0);           -- Overflow
```

```
                Q_FLAGS(4)      <= I_FLAGS(0);                         -- Signed

            when ALU_SBC =>
                L_DOUT      <= L_SBC_DR & L_SBC_DR;
                Q_FLAGS(0)      <= cy(L_SBC_DR(7), L_RI8(7), L_D8(7));   -- Carry
                Q_FLAGS(1)      <= ze(L_SBC_DR) and I_FLAGS(1);          -- Zero
                Q_FLAGS(2)      <= L_SBC_DR(7);                          -- Negative
                Q_FLAGS(3)      <= ov(L_SBC_DR(7), L_RI8(7), L_D8(7));   -- Overflow
                Q_FLAGS(4)      <= si(L_SBC_DR(7), L_RI8(7), L_D8(7));   -- Signed
                Q_FLAGS(5)      <= cy(L_SBC_DR(3), L_RI8(3), L_D8(3));   -- Halfcarry

            when ALU_SBIW =>
                L_DOUT      <= L_SBIW_D;
                Q_FLAGS(0)      <= L_SBIW_D(15) and not I_D(15);         -- Carry
                Q_FLAGS(1)      <= ze(L_SBIW_D(15 downto 8)) and
                              ze(L_SBIW_D(7 downto 0));             -- Zero
                Q_FLAGS(2)      <= L_SBIW_D(15);                         -- Negative
                Q_FLAGS(3)      <= I_D(15) and not L_SBIW_D(15);         -- Overflow
                Q_FLAGS(4)      <=  (L_SBIW_D(15) and not I_D(15))
                           xor (I_D(15) and not L_SBIW_D(15));      -- Signed

            when ALU_SREG =>
                case I_BIT(2 downto 0) is
                    when "000"  => Q_FLAGS(0)      <= I_BIT(3);
                    when "001"  => Q_FLAGS(1)      <= I_BIT(3);
                    when "010"  => Q_FLAGS(2)      <= I_BIT(3);
                    when "011"  => Q_FLAGS(3)      <= I_BIT(3);
                    when "100"  => Q_FLAGS(4)      <= I_BIT(3);
                    when "101"  => Q_FLAGS(5)      <= I_BIT(3);
                    when "110"  => Q_FLAGS(6)      <= I_BIT(3);
                    when others => Q_FLAGS(7)      <= I_BIT(3);
                end case;

            when ALU_SUB =>
                L_DOUT      <= L_SUB_DR & L_SUB_DR;
                Q_FLAGS(0)      <= cy(L_SUB_DR(7), L_RI8(7), L_D8(7));   -- Carry
                Q_FLAGS(1)      <= ze(L_SUB_DR);                         -- Zero
                Q_FLAGS(2)      <= L_SUB_DR(7);                          -- Negative
                Q_FLAGS(3)      <= ov(L_SUB_DR(7), L_RI8(7), L_D8(7));   -- Overflow
                Q_FLAGS(4)      <= si(L_SUB_DR(7), L_RI8(7), L_D8(7));   -- Signed
                Q_FLAGS(5)      <= cy(L_SUB_DR(3), L_RI8(3), L_D8(3));   -- Halfcarry

            when ALU_SWAP =>
                L_DOUT      <= L_SWAP_D & L_SWAP_D;

            when others =>
        end case;
    end process;

    L_D8      <= I_D(15 downto 8) when (I_D0 = '1') else I_D(7 downto 0);
    L_R8      <= I_R(15 downto 8) when (I_R0 = '1') else I_R(7 downto 0);
    L_RI8      <= I_IMM           when (I_RSEL = RS_IMM) else L_R8;

    L_ADIW_D       <= I_D + ("0000000000" & I_IMM(5 downto 0));
    L_SBIW_D       <= I_D - ("0000000000" & I_IMM(5 downto 0));
    L_ADD_DR       <= L_D8 + L_RI8;
    L_ADC_DR       <= L_ADD_DR + ("0000000" & I_FLAGS(0));
    L_ASR_D        <= L_D8(7) & L_D8(7 downto 1);
    L_AND_DR       <= L_D8 and L_RI8;
    L_DEC_D        <= L_D8 - X"01";
    L_INC_D        <= L_D8 + X"01";
    L_LSR_D        <= '0' & L_D8(7 downto 1);
```

```
        L_NEG_D         <= X"00" - L_D8;
        L_NOT_D         <= not L_D8;
        L_OR_DR         <= L_D8 or L_RI8;
        L_PROD          <= (L_SIGN_D & L_D8) * (L_SIGN_R & L_R8);
        L_ROR_D         <= I_FLAGS(0) &  L_D8(7 downto 1);
        L_SUB_DR        <= L_D8 - L_RI8;
        L_SBC_DR        <= L_SUB_DR - ("0000000" & I_FLAGS(0));
        L_SIGN_D        <= L_D8(7) and I_IMM(6);
        L_SIGN_R        <= L_R8(7) and I_IMM(5);
        L_SWAP_D        <= L_D8(3 downto 0) & L_D8(7 downto 4);
        L_XOR_DR        <= L_D8 xor L_R8;

        Q_DOUT      <= (I_DIN & I_DIN) when (I_RSEL = RS_DIN) else L_DOUT;

    end Behavioral;

```

src/alu.vhd

11 LISTING OF avr_fpga.vhd

```
-------------------------------------------------------------------------------
--
-- Copyright (C) 2009, 2010 Dr. Juergen Sauermann
--
--  This code is free software: you can redistribute it and/or modify
--  it under the terms of the GNU General Public License as published by
--  the Free Software Foundation, either version 3 of the License, or
--  (at your option) any later version.
--
--  This code is distributed in the hope that it will be useful,
--  but WITHOUT ANY WARRANTY; without even the implied warranty of
--  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
--  GNU General Public License for more details.
--
--  You should have received a copy of the GNU General Public License
--  along with this code (see the file named COPYING).
--  If not, see http://www.gnu.org/licenses/.
--
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
--
-- Module Name:     avr_fpga - Behavioral
-- Create Date:     13:51:24 11/07/2009
-- Description:     top level of a CPU
--
-------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity avr_fpga is
    port (  I_CLK_100   : in  std_logic;
            I_SWITCH    : in  std_logic_vector(9 downto 0);
            I_RX        : in  std_logic;

            Q_7_SEGMENT : out std_logic_vector(6 downto 0);
            Q_LEDS      : out std_logic_vector(3 downto 0);
            Q_TX        : out std_logic);
end avr_fpga;

architecture Behavioral of avr_fpga is

component cpu_core
    port (  I_CLK       : in  std_logic;
            I_CLR       : in  std_logic;
            I_INTVEC    : in  std_logic_vector( 5 downto 0);
            I_DIN       : in  std_logic_vector( 7 downto 0);

            Q_OPC       : out std_logic_vector(15 downto 0);
            Q_PC        : out std_logic_vector(15 downto 0);
            Q_DOUT      : out std_logic_vector( 7 downto 0);
            Q_ADR_IO    : out std_logic_vector( 7 downto 0);
            Q_RD_IO     : out std_logic;
            Q_WE_IO     : out std_logic);
end component;

```

```
    signal  C_PC            : std_logic_vector(15 downto 0);
    signal  C_OPC           : std_logic_vector(15 downto 0);
    signal  C_ADR_IO        : std_logic_vector( 7 downto 0);
    signal  C_DOUT          : std_logic_vector( 7 downto 0);
    signal  C_RD_IO         : std_logic;
    signal  C_WE_IO         : std_logic;

    component io
        port (  I_CLK       : in  std_logic;
                I_CLR       : in  std_logic;
                I_ADR_IO    : in  std_logic_vector( 7 downto 0);
                I_DIN       : in  std_logic_vector( 7 downto 0);
                I_RD_IO     : in  std_logic;
                I_WE_IO     : in  std_logic;
                I_SWITCH    : in  std_logic_vector( 7 downto 0);
                I_RX        : in  std_logic;

                Q_7_SEGMENT : out std_logic_vector( 6 downto 0);
                Q_DOUT      : out std_logic_vector( 7 downto 0);
                Q_INTVEC    : out std_logic_vector(5 downto 0);
                Q_LEDS      : out std_logic_vector( 1 downto 0);
                Q_TX        : out std_logic);
    end component;

    signal N_INTVEC         : std_logic_vector( 5 downto 0);
    signal N_DOUT           : std_logic_vector( 7 downto 0);
    signal N_TX             : std_logic;
    signal N_7_SEGMENT      : std_logic_vector( 6 downto 0);

    component segment7
        port ( I_CLK        : in  std_logic;

               I_CLR        : in  std_logic;
               I_OPC        : in  std_logic_vector(15 downto 0);
               I_PC         : in  std_logic_vector(15 downto 0);

               Q_7_SEGMENT  : out std_logic_vector( 6 downto 0));
    end component;

    signal S_7_SEGMENT      : std_logic_vector( 6 downto 0);

    signal L_CLK            : std_logic := '0';
    signal L_CLK_CNT        : std_logic_vector( 2 downto 0) := "000";
    signal L_CLR            : std_logic;            -- reset,  active low
    signal L_CLR_N          : std_logic := '0';     -- reset,  active low
    signal L_C1_N           : std_logic := '0';     -- switch debounce, active low
    signal L_C2_N           : std_logic := '0';     -- switch debounce, active low

    begin

        cpu : cpu_core
        port map(   I_CLK       => L_CLK,
                    I_CLR       => L_CLR,
                    I_DIN       => N_DOUT,
                    I_INTVEC    => N_INTVEC,

                    Q_ADR_IO    => C_ADR_IO,
                    Q_DOUT      => C_DOUT,
                    Q_OPC       => C_OPC,
                    Q_PC        => C_PC,
                    Q_RD_IO     => C_RD_IO,
                    Q_WE_IO     => C_WE_IO);
```

```

    ino : io
    port map(   I_CLK       => L_CLK,
                I_CLR       => L_CLR,
                I_ADR_IO    => C_ADR_IO,
                I_DIN       => C_DOUT,
                I_RD_IO     => C_RD_IO,
                I_RX        => I_RX,
                I_SWITCH    => I_SWITCH(7 downto 0),
                I_WE_IO     => C_WE_IO,

                Q_7_SEGMENT => N_7_SEGMENT,
                Q_DOUT      => N_DOUT,
                Q_INTVEC    => N_INTVEC,
                Q_LEDS      => Q_LEDS(1 downto 0),
                Q_TX        => N_TX);

    seg : segment7
    port map(   I_CLK       => L_CLK,
                I_CLR       => L_CLR,
                I_OPC       => C_OPC,
                I_PC        => C_PC,

                Q_7_SEGMENT => S_7_SEGMENT);

    -- input clock scaler
    --
    clk_div : process(I_CLK_100)
    begin
        if (rising_edge(I_CLK_100)) then
            L_CLK_CNT       <= L_CLK_CNT + "001";
            if (L_CLK_CNT = "001") then
                L_CLK_CNT       <= "000";
                L_CLK       <= not L_CLK;
            end if;
        end if;
    end process;

    -- reset button debounce process
    --
    deb : process(L_CLK)
    begin
        if (rising_edge(L_CLK)) then
            -- switch debounce
            if ((I_SWITCH(8) = '0') or (I_SWITCH(9) = '0')) then    -- pushed
                L_CLR_N       <= '0';
                L_C2_N        <= '0';
                L_C1_N        <= '0';
            else                                                    -- released
                L_CLR_N       <= L_C2_N;
                L_C2_N        <= L_C1_N;
                L_C1_N        <= '1';
            end if;
        end if;
    end process;

    L_CLR       <= not L_CLR_N;

    Q_LEDS(2)       <= I_RX;
    Q_LEDS(3)       <= N_TX;
    Q_7_SEGMENT        <= N_7_SEGMENT when (I_SWITCH(7) = '1') else S_7_SEGMENT;
    Q_TX       <= N_TX;
```

```

end Behavioral;

```

src/avr_fpga.vhd

12 LISTING OF baudgen.vhd

```
-------------------------------------------------------------------------------
--
-- Copyright (C) 2009, 2010 Dr. Juergen Sauermann
--
--  This code is free software: you can redistribute it and/or modify
--  it under the terms of the GNU General Public License as published by
--  the Free Software Foundation, either version 3 of the License, or
--  (at your option) any later version.
--
--  This code is distributed in the hope that it will be useful,
--  but WITHOUT ANY WARRANTY; without even the implied warranty of
--  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
--  GNU General Public License for more details.
--
--  You should have received a copy of the GNU General Public License
--  along with this code (see the file named COPYING).
--  If not, see http://www.gnu.org/licenses/.
--
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
--
-- Module Name:    baudgen - Behavioral
-- Create Date:    13:51:24 11/07/2009
-- Description:    fixed baud rate generator
--
-------------------------------------------------------------------------------
--
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity baudgen is
    generic(clock_freq  : std_logic_vector(31 downto 0);
                baud_rate   : std_logic_vector(27 downto 0));
    port(   I_CLK       : in  std_logic;

            I_CLR       : in  std_logic;
            Q_CE_1      : out std_logic;    -- baud x  1 clock enable
            Q_CE_16     : out std_logic);   -- baud x 16 clock enable
end baudgen;


architecture Behavioral of baudgen is

constant BAUD_16        : std_logic_vector(31 downto 0) := baud_rate & "0000";
constant LIMIT          : std_logic_vector(31 downto 0) := clock_freq - BAUD_16;

signal L_CE_16          : std_logic;
signal L_CNT_16         : std_logic_vector( 3 downto 0);
signal L_COUNTER        : std_logic_vector(31 downto 0);

begin

    baud16: process(I_CLK)
    begin
        if (rising_edge(I_CLK)) then
            if (I_CLR = '1') then
```

```
                L_COUNTER      <= X"00000000";
            elsif (L_COUNTER >= LIMIT) then
                L_COUNTER      <= L_COUNTER - LIMIT;
            else
                L_COUNTER      <= L_COUNTER + BAUD_16;
            end if;
        end if;
    end process;

    baud1: process(I_CLK)
    begin
        if (rising_edge(I_CLK)) then
            if (I_CLR = '1') then
                L_CNT_16      <= "0000";
            elsif (L_CE_16 = '1') then
                L_CNT_16      <= L_CNT_16 + "0001";
            end if;
        end if;
    end process;

    L_CE_16      <= '1' when (L_COUNTER >= LIMIT) else '0';
    Q_CE_16      <= L_CE_16;
    Q_CE_1      <= L_CE_16 when L_CNT_16 = "1111" else '0';

end behavioral;

```

src/baudgen.vhd

13 LISTING OF common.vhd

```
--------------------------------------------------------------------------------
--
-- Copyright (C) 2009, 2010 Dr. Juergen Sauermann
--
--  This code is free software: you can redistribute it and/or modify
--  it under the terms of the GNU General Public License as published by
--  the Free Software Foundation, either version 3 of the License, or
--  (at your option) any later version.
--
--  This code is distributed in the hope that it will be useful,
--  but WITHOUT ANY WARRANTY; without even the implied warranty of
--  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
--  GNU General Public License for more details.
--
--  You should have received a copy of the GNU General Public License
--  along with this code (see the file named COPYING).
--  If not, see http://www.gnu.org/licenses/.
--
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--
-- Module Name:    common
-- Create Date:    13:51:24 11/07/2009
-- Description:    constants shared by different modules.
--
--------------------------------------------------------------------------------
--
library IEEE;
use IEEE.STD_LOGIC_1164.all;

package common is

    ------------------------------------------------------------------------

    -- ALU operations
    --
    constant ALU_ADC    : std_logic_vector(4 downto 0) := "00000";
    constant ALU_ADD    : std_logic_vector(4 downto 0) := "00001";
    constant ALU_ADIW   : std_logic_vector(4 downto 0) := "00010";
    constant ALU_AND    : std_logic_vector(4 downto 0) := "00011";
    constant ALU_ASR    : std_logic_vector(4 downto 0) := "00100";
    constant ALU_BLD    : std_logic_vector(4 downto 0) := "00101";
    constant ALU_BIT_CS : std_logic_vector(4 downto 0) := "00110";
    constant ALU_COM    : std_logic_vector(4 downto 0) := "00111";
    constant ALU_DEC    : std_logic_vector(4 downto 0) := "01000";
    constant ALU_EOR    : std_logic_vector(4 downto 0) := "01001";
    constant ALU_MV_16  : std_logic_vector(4 downto 0) := "01010";
    constant ALU_INC    : std_logic_vector(4 downto 0) := "01011";
    constant ALU_INTR   : std_logic_vector(4 downto 0) := "01100";
    constant ALU_LSR    : std_logic_vector(4 downto 0) := "01101";
    constant ALU_D_MV_Q : std_logic_vector(4 downto 0) := "01110";
    constant ALU_R_MV_Q : std_logic_vector(4 downto 0) := "01111";
    constant ALU_MULT   : std_logic_vector(4 downto 0) := "10000";
    constant ALU_NEG    : std_logic_vector(4 downto 0) := "10001";
    constant ALU_OR     : std_logic_vector(4 downto 0) := "10010";
    constant ALU_PC_1   : std_logic_vector(4 downto 0) := "10011";
    constant ALU_PC_2   : std_logic_vector(4 downto 0) := "10100";
    constant ALU_ROR    : std_logic_vector(4 downto 0) := "10101";
```

```
    constant ALU_SBC    : std_logic_vector(4 downto 0) := "10110";
    constant ALU_SBIW   : std_logic_vector(4 downto 0) := "10111";
    constant ALU_SREG   : std_logic_vector(4 downto 0) := "11000";
    constant ALU_SUB    : std_logic_vector(4 downto 0) := "11001";
    constant ALU_SWAP   : std_logic_vector(4 downto 0) := "11010";

    ----------------------------------------------------------------------
    --
    -- PC manipulations
    --
    constant PC_NEXT    : std_logic_vector(2 downto 0) := "000";    -- PC += 1
    constant PC_BCC     : std_logic_vector(2 downto 0) := "001";    -- PC ?= IMM
    constant PC_LD_I    : std_logic_vector(2 downto 0) := "010";    -- PC = IMM
    constant PC_LD_Z    : std_logic_vector(2 downto 0) := "011";    -- PC = Z
    constant PC_LD_S    : std_logic_vector(2 downto 0) := "100";    -- PC = (SP)
    constant PC_SKIP_Z  : std_logic_vector(2 downto 0) := "101";    -- SKIP if Z
    constant PC_SKIP_T  : std_logic_vector(2 downto 0) := "110";    -- SKIP if T

    ----------------------------------------------------------------------
    --
    -- Addressing modes. An address mode consists of two sub-fields,
    -- which are the source of the address and an offset from the source.
    -- Bit 3 indicates if the address will be modified.

    -- address source
    constant AS_SP  : std_logic_vector(2 downto 0) := "000";     -- SP
    constant AS_Z   : std_logic_vector(2 downto 0) := "001";     -- Z
    constant AS_Y   : std_logic_vector(2 downto 0) := "010";     -- Y
    constant AS_X   : std_logic_vector(2 downto 0) := "011";     -- X
    constant AS_IMM : std_logic_vector(2 downto 0) := "100";     -- IMM

    -- address offset
    constant AO_0   : std_logic_vector(5 downto 3) := "000";     -- as is
    constant AO_Q   : std_logic_vector(5 downto 3) := "010";     -- +q
    constant AO_i   : std_logic_vector(5 downto 3) := "001";     -- +1
    constant AO_ii  : std_logic_vector(5 downto 3) := "011";     -- +2
    constant AO_d   : std_logic_vector(5 downto 3) := "101";     -- -1
    constant AO_dd  : std_logic_vector(5 downto 3) := "111";     -- -2
    --                                                   |
    --                                                +--+
    -- address updated ?                              |
    --                                                v
    constant AM_WX : std_logic_vector(3 downto 0) := '1' & AS_X;  -- X ++ or --
    constant AM_WY : std_logic_vector(3 downto 0) := '1' & AS_Y;  -- Y ++ or --
    constant AM_WZ : std_logic_vector(3 downto 0) := '1' & AS_Z;  -- Z ++ or --
    constant AM_WS : std_logic_vector(3 downto 0) := '1' & AS_SP; -- SP ++/--

    -- address modes used
    --
    constant AMOD_ABS : std_logic_vector(5 downto 0) := AO_0  & AS_IMM; -- IMM
    constant AMOD_X   : std_logic_vector(5 downto 0) := AO_0  & AS_X;   -- (X)
    constant AMOD_Xq  : std_logic_vector(5 downto 0) := AO_Q  & AS_X;   -- (X+q)
    constant AMOD_Xi  : std_logic_vector(5 downto 0) := AO_i  & AS_X;   -- (X++)
    constant AMOD_dX  : std_logic_vector(5 downto 0) := AO_d  & AS_X;   -- (--X)
    constant AMOD_Y   : std_logic_vector(5 downto 0) := AO_0  & AS_Y;   -- (Y)
    constant AMOD_Yq  : std_logic_vector(5 downto 0) := AO_Q  & AS_Y;   -- (Y+q)
    constant AMOD_Yi  : std_logic_vector(5 downto 0) := AO_i  & AS_Y;   -- (Y++)
    constant AMOD_dY  : std_logic_vector(5 downto 0) := AO_d  & AS_Y;   -- (--Y)
    constant AMOD_Z   : std_logic_vector(5 downto 0) := AO_0  & AS_Z;   -- (Z)
    constant AMOD_Zq  : std_logic_vector(5 downto 0) := AO_Q  & AS_Z;   -- (Z+q)
    constant AMOD_Zi  : std_logic_vector(5 downto 0) := AO_i  & AS_Z;   -- (Z++)
    constant AMOD_dZ  : std_logic_vector(5 downto 0) := AO_d  & AS_Z;   -- (--Z)
```

```
        constant AMOD_SPi : std_logic_vector(5 downto 0) := AO_i  & AS_SP;  -- (SP++)
        constant AMOD_SPii: std_logic_vector(5 downto 0) := AO_ii & AS_SP;  -- (SP++)
        constant AMOD_dSP : std_logic_vector(5 downto 0) := AO_d  & AS_SP;  -- (--SP)
        constant AMOD_ddSP: std_logic_vector(5 downto 0) := AO_dd & AS_SP;  -- (--SP)

        -----------------------------------------------------------------------

        -- Stack pointer manipulations.
        --
        constant SP_NOP : std_logic_vector(2 downto 0) := "000";
        constant SP_ADD1: std_logic_vector(2 downto 0) := "001";
        constant SP_ADD2: std_logic_vector(2 downto 0) := "010";
        constant SP_SUB1: std_logic_vector(2 downto 0) := "011";
        constant SP_SUB2: std_logic_vector(2 downto 0) := "100";

        -----------------------------------------------------------------------
        --
        -- ALU multiplexers.
        --
        constant RS_REG : std_logic_vector(1 downto 0) := "00";
        constant RS_IMM : std_logic_vector(1 downto 0) := "01";
        constant RS_DIN : std_logic_vector(1 downto 0) := "10";

        -----------------------------------------------------------------------
        --
        -- Multiplier variants. F means FMULT (as opposed to MULT).
        -- S and U means signed vs. unsigned operands.
        --
        constant MULT_UU  : std_logic_vector(2 downto 0) := "000";
        constant MULT_SU  : std_logic_vector(2 downto 0) := "010";
        constant MULT_SS  : std_logic_vector(2 downto 0) := "011";
        constant MULT_FUU : std_logic_vector(2 downto 0) := "100";
        constant MULT_FSU : std_logic_vector(2 downto 0) := "110";
        constant MULT_FSS : std_logic_vector(2 downto 0) := "111";

        -----------------------------------------------------------------------

    end common;

```

src/common.vhd

14 LISTING OF cpu_core.vhd

```
-------------------------------------------------------------------------------
--
-- Copyright (C) 2009, 2010 Dr. Juergen Sauermann
--
--  This code is free software: you can redistribute it and/or modify
--  it under the terms of the GNU General Public License as published by
--  the Free Software Foundation, either version 3 of the License, or
--  (at your option) any later version.
--
--  This code is distributed in the hope that it will be useful,
--  but WITHOUT ANY WARRANTY; without even the implied warranty of
--  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
--  GNU General Public License for more details.
--
--  You should have received a copy of the GNU General Public License
--  along with this code (see the file named COPYING).
--  If not, see http://www.gnu.org/licenses/.
--
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
--
-- Module Name:    cpu_core - Behavioral
-- Create Date:    13:51:24 11/07/2009
-- Description:    the instruction set implementation of a CPU.
--
-------------------------------------------------------------------------------
--
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity cpu_core is
    port (  I_CLK       : in  std_logic;
            I_CLR       : in  std_logic;
            I_INTVEC    : in  std_logic_vector( 5 downto 0);
            I_DIN       : in  std_logic_vector( 7 downto 0);

            Q_OPC       : out std_logic_vector(15 downto 0);
            Q_PC        : out std_logic_vector(15 downto 0);
            Q_DOUT      : out std_logic_vector( 7 downto 0);
            Q_ADR_IO    : out std_logic_vector( 7 downto 0);
            Q_RD_IO     : out std_logic;
            Q_WE_IO     : out std_logic);
end cpu_core;

architecture Behavioral of cpu_core is

component opc_fetch
    port(   I_CLK       : in  std_logic;

            I_CLR       : in  std_logic;
            I_INTVEC    : in  std_logic_vector( 5 downto 0);
            I_NEW_PC    : in  std_logic_vector(15 downto 0);
            I_LOAD_PC   : in  std_logic;
            I_PM_ADR    : in  std_logic_vector(11 downto 0);
            I_SKIP      : in  std_logic;

```

```
                Q_OPC       : out std_logic_vector(31 downto 0);
                Q_PC        : out std_logic_vector(15 downto 0);
                Q_PM_DOUT   : out std_logic_vector( 7 downto 0);
                Q_T0        : out std_logic);
    end component;

    signal F_PC             : std_logic_vector(15 downto 0);
    signal F_OPC            : std_logic_vector(31 downto 0);
    signal F_PM_DOUT        : std_logic_vector( 7 downto 0);
    signal F_T0             : std_logic;

    component opc_deco is
        port (  I_CLK       : in  std_logic;

                I_OPC       : in  std_logic_vector(31 downto 0);
                I_PC        : in  std_logic_vector(15 downto 0);
                I_T0        : in  std_logic;

                Q_ALU_OP    : out std_logic_vector( 4 downto 0);
                Q_AMOD      : out std_logic_vector( 5 downto 0);
                Q_BIT       : out std_logic_vector( 3 downto 0);
                Q_DDDDD     : out std_logic_vector( 4 downto 0);
                Q_IMM       : out std_logic_vector(15 downto 0);
                Q_JADR      : out std_logic_vector(15 downto 0);
                Q_OPC       : out std_logic_vector(15 downto 0);
                Q_PC        : out std_logic_vector(15 downto 0);
                Q_PC_OP     : out std_logic_vector( 2 downto 0);
                Q_PMS       : out std_logic;  -- program memory select
                Q_RD_M      : out std_logic;
                Q_RRRRR     : out std_logic_vector( 4 downto 0);
                Q_RSEL      : out std_logic_vector( 1 downto 0);
                Q_WE_01     : out std_logic;
                Q_WE_D      : out std_logic_vector( 1 downto 0);
                Q_WE_F      : out std_logic;
                Q_WE_M      : out std_logic_vector( 1 downto 0);
                Q_WE_XYZS   : out std_logic);
    end component;

    signal D_ALU_OP         : std_logic_vector( 4 downto 0);
    signal D_AMOD           : std_logic_vector( 5 downto 0);
    signal D_BIT            : std_logic_vector( 3 downto 0);
    signal D_DDDDD          : std_logic_vector( 4 downto 0);
    signal D_IMM            : std_logic_vector(15 downto 0);
    signal D_JADR           : std_logic_vector(15 downto 0);
    signal D_OPC            : std_logic_vector(15 downto 0);
    signal D_PC             : std_logic_vector(15 downto 0);
    signal D_PC_OP          : std_logic_vector(2 downto 0);
    signal D_PMS            : std_logic;
    signal D_RD_M           : std_logic;
    signal D_RRRRR          : std_logic_vector( 4 downto 0);
    signal D_RSEL           : std_logic_vector( 1 downto 0);
    signal D_WE_01          : std_logic;
    signal D_WE_D           : std_logic_vector( 1 downto 0);
    signal D_WE_F           : std_logic;
    signal D_WE_M           : std_logic_vector( 1 downto 0);
    signal D_WE_XYZS        : std_logic;

    component data_path
        port(   I_CLK       : in    std_logic;

                I_ALU_OP    : in  std_logic_vector( 4 downto 0);
                I_AMOD      : in  std_logic_vector( 5 downto 0);
```

```
                I_BIT       : in  std_logic_vector( 3 downto 0);
                I_DDDDD     : in  std_logic_vector( 4 downto 0);
                I_DIN       : in  std_logic_vector( 7 downto 0);
                I_IMM       : in  std_logic_vector(15 downto 0);
                I_JADR      : in  std_logic_vector(15 downto 0);
                I_PC_OP     : in  std_logic_vector( 2 downto 0);
                I_OPC       : in  std_logic_vector(15 downto 0);
                I_PC        : in  std_logic_vector(15 downto 0);
                I_PMS       : in  std_logic;  -- program memory select
                I_RD_M      : in  std_logic;
                I_RRRRR     : in  std_logic_vector( 4 downto 0);
                I_RSEL      : in  std_logic_vector( 1 downto 0);
                I_WE_01     : in  std_logic;
                I_WE_D      : in  std_logic_vector( 1 downto 0);
                I_WE_F      : in  std_logic;
                I_WE_M      : in  std_logic_vector( 1 downto 0);
                I_WE_XYZS   : in  std_logic;

                Q_ADR       : out std_logic_vector(15 downto 0);
                Q_DOUT      : out std_logic_vector( 7 downto 0);
                Q_INT_ENA   : out std_logic;
                Q_LOAD_PC   : out std_logic;
                Q_NEW_PC    : out std_logic_vector(15 downto 0);
                Q_OPC       : out std_logic_vector(15 downto 0);
                Q_PC        : out std_logic_vector(15 downto 0);
                Q_RD_IO     : out std_logic;
                Q_SKIP      : out std_logic;
                Q_WE_IO     : out std_logic);
    end component;

    signal R_INT_ENA        : std_logic;
    signal R_NEW_PC         : std_logic_vector(15 downto 0);
    signal R_LOAD_PC        : std_logic;
    signal R_SKIP           : std_logic;
    signal R_ADR            : std_logic_vector(15 downto 0);

    -- local signals
    --
    signal L_DIN            : std_logic_vector( 7 downto 0);
    signal L_INTVEC_5       : std_logic;

    begin

        opcf : opc_fetch
        port map(   I_CLK       => I_CLK,

                    I_CLR       => I_CLR,
                    I_INTVEC(5) => L_INTVEC_5,
                    I_INTVEC(4 downto 0) => I_INTVEC(4 downto 0),
                    I_LOAD_PC   => R_LOAD_PC,
                    I_NEW_PC    => R_NEW_PC,
                    I_PM_ADR    => R_ADR(11 downto 0),
                    I_SKIP      => R_SKIP,

                    Q_PC        => F_PC,
                    Q_OPC       => F_OPC,
                    Q_T0        => F_T0,
                    Q_PM_DOUT   => F_PM_DOUT);

        odec : opc_deco
        port map(   I_CLK       => I_CLK,

```

```
                        I_OPC       => F_OPC,
                        I_PC        => F_PC,
                        I_T0        => F_T0,

                        Q_ALU_OP    => D_ALU_OP,
                        Q_AMOD      => D_AMOD,
                        Q_BIT       => D_BIT,
                        Q_DDDDD     => D_DDDDD,
                        Q_IMM       => D_IMM,
                        Q_JADR      => D_JADR,
                        Q_OPC       => D_OPC,
                        Q_PC        => D_PC,
                        Q_PC_OP     => D_PC_OP,
                        Q_PMS       => D_PMS,
                        Q_RD_M      => D_RD_M,
                        Q_RRRRR     => D_RRRRR,
                        Q_RSEL      => D_RSEL,
                        Q_WE_01     => D_WE_01,
                        Q_WE_D      => D_WE_D,
                        Q_WE_F      => D_WE_F,
                        Q_WE_M      => D_WE_M,
                        Q_WE_XYZS   => D_WE_XYZS);

    dpath : data_path
    port map(           I_CLK       => I_CLK,

                        I_ALU_OP    => D_ALU_OP,
                        I_AMOD      => D_AMOD,
                        I_BIT       => D_BIT,
                        I_DDDDD     => D_DDDDD,
                        I_DIN       => L_DIN,
                        I_IMM       => D_IMM,
                        I_JADR      => D_JADR,
                        I_OPC       => D_OPC,
                        I_PC        => D_PC,
                        I_PC_OP     => D_PC_OP,
                        I_PMS       => D_PMS,
                        I_RD_M      => D_RD_M,
                        I_RRRRR     => D_RRRRR,
                        I_RSEL      => D_RSEL,
                        I_WE_01     => D_WE_01,
                        I_WE_D      => D_WE_D,
                        I_WE_F      => D_WE_F,
                        I_WE_M      => D_WE_M,
                        I_WE_XYZS   => D_WE_XYZS,

                        Q_ADR       => R_ADR,
                        Q_DOUT      => Q_DOUT,
                        Q_INT_ENA   => R_INT_ENA,
                        Q_NEW_PC    => R_NEW_PC,
                        Q_OPC       => Q_OPC,
                        Q_PC        => Q_PC,
                        Q_LOAD_PC   => R_LOAD_PC,
                        Q_RD_IO     => Q_RD_IO,
                        Q_SKIP      => R_SKIP,
                        Q_WE_IO     => Q_WE_IO);

    L_DIN       <= F_PM_DOUT when (D_PMS = '1') else I_DIN(7 downto 0);
    L_INTVEC_5       <= I_INTVEC(5) and R_INT_ENA;
    Q_ADR_IO       <= R_ADR(7 downto 0);

end Behavioral;
```

src/cpu_core.vhd

15 LISTING OF data_mem.vhd

```
-------------------------------------------------------------------------------
--
-- Copyright (C) 2009, 2010 Dr. Juergen Sauermann
--
--  This code is free software: you can redistribute it and/or modify
--  it under the terms of the GNU General Public License as published by
--  the Free Software Foundation, either version 3 of the License, or
--  (at your option) any later version.
--
--  This code is distributed in the hope that it will be useful,
--  but WITHOUT ANY WARRANTY; without even the implied warranty of
--  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
--  GNU General Public License for more details.
--
--  You should have received a copy of the GNU General Public License
--  along with this code (see the file named COPYING).
--  If not, see http://www.gnu.org/licenses/.
--
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
--
-- Module Name:    data_mem - Behavioral
-- Create Date:    14:09:04 10/30/2009
-- Description:    the data mempry of a CPU.
--
-------------------------------------------------------------------------------
--
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity data_mem is
    port (  I_CLK       : in  std_logic;

            I_ADR       : in  std_logic_vector(10 downto 0);
            I_DIN       : in  std_logic_vector(15 downto 0);
            I_WE        : in  std_logic_vector( 1 downto 0);

            Q_DOUT      : out std_logic_vector(15 downto 0));
end data_mem;

architecture Behavioral of data_mem is

constant zero_256 : bit_vector := X"00000000000000000000000000000000"
                                & X"00000000000000000000000000000000";
constant nine_256 : bit_vector := X"99999999999999999999999999999999"
                                & X"99999999999999999999999999999999";

component RAMB4_S4_S4
    generic(INIT_00 : bit_vector := zero_256;
            INIT_01 : bit_vector := zero_256;
            INIT_02 : bit_vector := zero_256;
            INIT_03 : bit_vector := zero_256;
            INIT_04 : bit_vector := zero_256;
            INIT_05 : bit_vector := zero_256;
            INIT_06 : bit_vector := zero_256;
            INIT_07 : bit_vector := zero_256;
```

```
                INIT_08 : bit_vector := zero_256;
                INIT_09 : bit_vector := zero_256;
                INIT_0A : bit_vector := zero_256;
                INIT_0B : bit_vector := zero_256;
                INIT_0C : bit_vector := zero_256;
                INIT_0D : bit_vector := zero_256;
                INIT_0E : bit_vector := zero_256;
                INIT_0F : bit_vector := zero_256);

        port(   DOA     : out std_logic_vector(3 downto 0);
                DOB     : out std_logic_vector(3 downto 0);
                ADDRA   : in  std_logic_vector(9 downto 0);
                ADDRB   : in  std_logic_vector(9 downto 0);
                CLKA    : in  std_ulogic;
                CLKB    : in  std_ulogic;
                DIA     : in  std_logic_vector(3 downto 0);
                DIB     : in  std_logic_vector(3 downto 0);
                ENA     : in  std_ulogic;
                ENB     : in  std_ulogic;
                RSTA    : in  std_ulogic;
                RSTB    : in  std_ulogic;
                WEA     : in  std_ulogic;
                WEB     : in  std_ulogic);
    end component;

    signal L_ADR_0      : std_logic;
    signal L_ADR_E      : std_logic_vector(10 downto 1);
    signal L_ADR_O      : std_logic_vector(10 downto 1);
    signal L_DIN_E      : std_logic_vector( 7 downto 0);
    signal L_DIN_O      : std_logic_vector( 7 downto 0);
    signal L_DOUT_E     : std_logic_vector( 7 downto 0);
    signal L_DOUT_O     : std_logic_vector( 7 downto 0);
    signal L_WE_E       : std_logic;
    signal L_WE_O       : std_logic;

    begin

        sr_0 : RAMB4_S4_S4 ---------------------------------------------------------
        generic map(INIT_00 => nine_256, INIT_01 => nine_256, INIT_02 => nine_256,
                    INIT_03 => nine_256, INIT_04 => nine_256, INIT_05 => nine_256,
                    INIT_06 => nine_256, INIT_07 => nine_256, INIT_08 => nine_256,
                    INIT_09 => nine_256, INIT_0A => nine_256, INIT_0B => nine_256,
                    INIT_0C => nine_256, INIT_0D => nine_256, INIT_0E => nine_256,
                    INIT_0F => nine_256)

        port map(   ADDRA => L_ADR_E,               ADDRB => "0000000000",
                    CLKA  => I_CLK,                 CLKB  => I_CLK,
                    DIA   => L_DIN_E(3 downto 0),   DIB   => "0000",
                    ENA   => '1',                   ENB   => '0',
                    RSTA  => '0',                   RSTB  => '0',
                    WEA   => L_WE_E,                WEB   => '0',
                    DOA   => L_DOUT_E(3 downto 0),  DOB   => open);

        sr_1 : RAMB4_S4_S4 ---------------------------------------------------------
        generic map(INIT_00 => nine_256, INIT_01 => nine_256, INIT_02 => nine_256,
                    INIT_03 => nine_256, INIT_04 => nine_256, INIT_05 => nine_256,
                    INIT_06 => nine_256, INIT_07 => nine_256, INIT_08 => nine_256,
                    INIT_09 => nine_256, INIT_0A => nine_256, INIT_0B => nine_256,
                    INIT_0C => nine_256, INIT_0D => nine_256, INIT_0E => nine_256,
                    INIT_0F => nine_256)

        port map(   ADDRA => L_ADR_E,               ADDRB => "0000000000",
```

```
                CLKA  => I_CLK,                 CLKB  => I_CLK,
                DIA   => L_DIN_E(7 downto 4),   DIB   => "0000",
                ENA   => '1',                   ENB   => '0',
                RSTA  => '0',                   RSTB  => '0',
                WEA   => L_WE_E,                WEB   => '0',
                DOA   => L_DOUT_E(7 downto 4),  DOB   => open);

    sr_2 : RAMB4_S4_S4 -----------------------------------------------------
    generic map(INIT_00 => nine_256, INIT_01 => nine_256, INIT_02 => nine_256,
                INIT_03 => nine_256, INIT_04 => nine_256, INIT_05 => nine_256,
                INIT_06 => nine_256, INIT_07 => nine_256, INIT_08 => nine_256,
                INIT_09 => nine_256, INIT_0A => nine_256, INIT_0B => nine_256,
                INIT_0C => nine_256, INIT_0D => nine_256, INIT_0E => nine_256,
                INIT_0F => nine_256)

    port map(   ADDRA => L_ADR_O,               ADDRB => "0000000000",
                CLKA  => I_CLK,                 CLKB  => I_CLK,
                DIA   => L_DIN_O(3 downto 0),   DIB   => "0000",
                ENA   => '1',                   ENB   => '0',
                RSTA  => '0',                   RSTB  => '0',
                WEA   => L_WE_O,                WEB   => '0',
                DOA   => L_DOUT_O(3 downto 0),  DOB   => open);

    sr_3 : RAMB4_S4_S4 -----------------------------------------------------
    generic map(INIT_00 => nine_256, INIT_01 => nine_256, INIT_02 => nine_256,
                INIT_03 => nine_256, INIT_04 => nine_256, INIT_05 => nine_256,
                INIT_06 => nine_256, INIT_07 => nine_256, INIT_08 => nine_256,
                INIT_09 => nine_256, INIT_0A => nine_256, INIT_0B => nine_256,
                INIT_0C => nine_256, INIT_0D => nine_256, INIT_0E => nine_256,
                INIT_0F => nine_256)

    port map(   ADDRA => L_ADR_O,               ADDRB => "0000000000",
                CLKA  => I_CLK,                 CLKB  => I_CLK,
                DIA   => L_DIN_O(7 downto  4),  DIB   => "0000",
                ENA   => '1',                   ENB   => '0',
                RSTA  => '0',                   RSTB  => '0',
                WEA   => L_WE_O,                WEB   => '0',
                DOA   => L_DOUT_O(7 downto  4), DOB   => open);


    -- remember ADR(0)
    --
    adr0: process(I_CLK)
    begin
        if (rising_edge(I_CLK)) then
            L_ADR_0      <= I_ADR(0);
        end if;
    end process;

    -- we use two memory blocks _E and _O (even and odd).
    -- This gives us a memory with ADR and ADR + 1 at th same time.
    -- The second port is currently unused, but may be used later,
    -- e.g. for DMA.
    --

    L_ADR_O      <= I_ADR(10 downto 1);
    L_ADR_E      <= I_ADR(10 downto 1) + ("000000000" & I_ADR(0));

    L_DIN_E      <= I_DIN( 7 downto 0) when (I_ADR(0) = '0') else I_DIN(15 downto 8);
    L_DIN_O      <= I_DIN( 7 downto 0) when (I_ADR(0) = '1') else I_DIN(15 downto 8);

    L_WE_E      <= I_WE(1) or (I_WE(0) and not I_ADR(0));
```

```
        L_WE_O      <= I_WE(1) or (I_WE(0) and     I_ADR(0));

        Q_DOUT( 7 downto 0)      <= L_DOUT_E when (L_ADR_0 = '0') else L_DOUT_O;
        Q_DOUT(15 downto 8)      <= L_DOUT_E when (L_ADR_0 = '1') else L_DOUT_O;

    end Behavioral;

```

src/data_mem.vhd

16 LISTING OF data_path.vhd

```
-------------------------------------------------------------------------------
--
-- Copyright (C) 2009, 2010 Dr. Juergen Sauermann
--
--  This code is free software: you can redistribute it and/or modify
--  it under the terms of the GNU General Public License as published by
--  the Free Software Foundation, either version 3 of the License, or
--  (at your option) any later version.
--
--  This code is distributed in the hope that it will be useful,
--  but WITHOUT ANY WARRANTY; without even the implied warranty of
--  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
--  GNU General Public License for more details.
--
--  You should have received a copy of the GNU General Public License
--  along with this code (see the file named COPYING).
--  If not, see http://www.gnu.org/licenses/.
--
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
--
-- Module Name:    data_path - Behavioral
-- Create Date:    13:24:10 10/29/2009
-- Description:    the data path of a CPU.
--
-------------------------------------------------------------------------------
--
library IEEE;
use IEEE.std_logic_1164.ALL;
use IEEE.std_logic_ARITH.ALL;
use IEEE.std_logic_UNSIGNED.ALL;

use work.common.ALL;

entity data_path is
    port(   I_CLK           : in  std_logic;

            I_ALU_OP    : in  std_logic_vector( 4 downto 0);
            I_AMOD      : in  std_logic_vector( 5 downto 0);
            I_BIT       : in  std_logic_vector( 3 downto 0);
            I_DDDDD     : in  std_logic_vector( 4 downto 0);
            I_DIN       : in  std_logic_vector( 7 downto 0);
            I_IMM       : in  std_logic_vector(15 downto 0);
            I_JADR      : in  std_logic_vector(15 downto 0);
            I_OPC       : in  std_logic_vector(15 downto 0);
            I_PC        : in  std_logic_vector(15 downto 0);
            I_PC_OP     : in  std_logic_vector( 2 downto 0);
            I_PMS       : in  std_logic;  -- program memory select
            I_RD_M      : in  std_logic;
            I_RRRRR     : in  std_logic_vector( 4 downto 0);
            I_RSEL      : in  std_logic_vector( 1 downto 0);
            I_WE_01     : in  std_logic;
            I_WE_D      : in  std_logic_vector( 1 downto 0);
            I_WE_F      : in  std_logic;
            I_WE_M      : in  std_logic_vector( 1 downto 0);
            I_WE_XYZS   : in  std_logic;

            Q_ADR       : out std_logic_vector(15 downto 0);
```

```
                Q_DOUT      : out std_logic_vector( 7 downto 0);
                Q_INT_ENA   : out std_logic;
                Q_LOAD_PC   : out std_logic;
                Q_NEW_PC    : out std_logic_vector(15 downto 0);
                Q_OPC       : out std_logic_vector(15 downto 0);
                Q_PC        : out std_logic_vector(15 downto 0);
                Q_RD_IO     : out std_logic;
                Q_SKIP      : out std_logic;
                Q_WE_IO     : out std_logic);
end data_path;

architecture Behavioral of data_path is

component alu
    port (  I_ALU_OP    : in  std_logic_vector( 4 downto 0);
            I_BIT       : in  std_logic_vector( 3 downto 0);
            I_D         : in  std_logic_vector(15 downto 0);
            I_D0        : in  std_logic;
            I_DIN       : in  std_logic_vector( 7 downto 0);
            I_FLAGS     : in  std_logic_vector( 7 downto 0);
            I_IMM       : in  std_logic_vector( 7 downto 0);
            I_PC        : in  std_logic_vector(15 downto 0);
            I_R         : in  std_logic_vector(15 downto 0);
            I_R0        : in  std_logic;
            I_RSEL      : in  std_logic_vector( 1 downto 0);

            Q_FLAGS     : out std_logic_vector( 9 downto 0);
            Q_DOUT      : out std_logic_vector(15 downto 0));
end component;

signal A_DOUT           : std_logic_vector(15 downto 0);
signal A_FLAGS          : std_logic_vector( 9 downto 0);

component register_file
    port (  I_CLK       : in  std_logic;

            I_AMOD      : in  std_logic_vector( 5 downto 0);
            I_COND      : in  std_logic_vector( 3 downto 0);
            I_DDDDD     : in  std_logic_vector( 4 downto 0);
            I_DIN       : in  std_logic_vector(15 downto 0);
            I_FLAGS     : in  std_logic_vector( 7 downto 0);
            I_IMM       : in  std_logic_vector(15 downto 0);
            I_RRRR      : in  std_logic_vector( 4 downto 1);
            I_WE_01     : in  std_logic;
            I_WE_D      : in  std_logic_vector( 1 downto 0);
            I_WE_F      : in  std_logic;
            I_WE_M      : in  std_logic;
            I_WE_XYZS   : in  std_logic;

            Q_ADR       : out std_logic_vector(15 downto 0);
            Q_CC        : out std_logic;
            Q_D         : out std_logic_vector(15 downto 0);
            Q_FLAGS     : out std_logic_vector( 7 downto 0);
            Q_R         : out std_logic_vector(15 downto 0);
            Q_S         : out std_logic_vector( 7 downto 0);
            Q_Z         : out std_logic_vector(15 downto 0));
end component;

signal F_ADR            : std_logic_vector(15 downto 0);
signal F_CC             : std_logic;
signal F_D              : std_logic_vector(15 downto 0);
signal F_FLAGS          : std_logic_vector( 7 downto 0);
```

```
    signal F_R              : std_logic_vector(15 downto 0);
    signal F_S              : std_logic_vector( 7 downto 0);
    signal F_Z              : std_logic_vector(15 downto 0);

    component data_mem
        port (  I_CLK       : in  std_logic;

                I_ADR       : in  std_logic_vector(10 downto 0);
                I_DIN       : in  std_logic_vector(15 downto 0);
                I_WE        : in  std_logic_vector( 1 downto 0);

                Q_DOUT      : out std_logic_vector(15 downto 0));
    end component;

    signal M_DOUT           : std_logic_vector(15 downto 0);

    signal L_DIN            : std_logic_vector( 7 downto 0);
    signal L_WE_SRAM        : std_logic_vector( 1 downto 0);
    signal L_FLAGS_98       : std_logic_vector( 9 downto 8);

    begin

        alui : alu
        port map(   I_ALU_OP    => I_ALU_OP,
                    I_BIT       => I_BIT,
                    I_D         => F_D,
                    I_D0        => I_DDDDD(0),
                    I_DIN       => L_DIN,
                    I_FLAGS     => F_FLAGS,
                    I_IMM       => I_IMM(7 downto 0),
                    I_PC        => I_PC,
                    I_R         => F_R,
                    I_R0        => I_RRRRR(0),
                    I_RSEL      => I_RSEL,

                    Q_FLAGS     => A_FLAGS,
                    Q_DOUT      => A_DOUT);

        regs : register_file
        port map(   I_CLK       => I_CLK,

                    I_AMOD      => I_AMOD,
                    I_COND(3)   => I_OPC(10),
                    I_COND(2 downto 0)=> I_OPC(2 downto 0),
                    I_DDDDD     => I_DDDDD,
                    I_DIN       => A_DOUT,
                    I_FLAGS     => A_FLAGS(7 downto 0),
                    I_IMM       => I_IMM,
                    I_RRRR      => I_RRRRR(4 downto 1),
                    I_WE_01     => I_WE_01,
                    I_WE_D      => I_WE_D,
                    I_WE_F      => I_WE_F,
                    I_WE_M      => I_WE_M(0),
                    I_WE_XYZS   => I_WE_XYZS,

                    Q_ADR       => F_ADR,
                    Q_CC        => F_CC,
                    Q_D         => F_D,
                    Q_FLAGS     => F_FLAGS,
                    Q_R         => F_R,
                    Q_S         => F_S,   -- Q_Rxx(F_ADR)
                    Q_Z         => F_Z);
```

```

        sram : data_mem
        port map(   I_CLK    => I_CLK,

                    I_ADR    => F_ADR(10 downto 0),
                    I_DIN    => A_DOUT,
                    I_WE     => L_WE_SRAM,

                    Q_DOUT   => M_DOUT);

        -- remember A_FLAGS(9 downto 8) (within the current instruction).
        --
        flg98: process(I_CLK)
        begin
            if (rising_edge(I_CLK)) then
                L_FLAGS_98      <= A_FLAGS(9 downto 8);
            end if;
        end process;

        -- whether PC shall be loaded with NEW_PC or not.
        -- I.e. if a branch shall be taken or not.
        --
        process(I_PC_OP, F_CC)
        begin
            case I_PC_OP is
                when PC_BCC  => Q_LOAD_PC      <= F_CC;      -- maybe (PC on I_JADR)
                when PC_LD_I => Q_LOAD_PC      <= '1';       -- yes: new PC on I_JADR
                when PC_LD_Z => Q_LOAD_PC      <= '1';       -- yes: new PC in Z
                when PC_LD_S => Q_LOAD_PC      <= '1';       -- yes: new PC on stack
                when others  => Q_LOAD_PC      <= '0';       -- no.
            end case;
        end process;

        -- whether the next instruction shall be skipped or not.
        --
        process(I_PC_OP, L_FLAGS_98, F_CC)
        begin
            case I_PC_OP is
                when PC_BCC    => Q_SKIP      <= F_CC;            -- if cond met
                when PC_LD_I   => Q_SKIP      <= '1';             -- yes
                when PC_LD_Z   => Q_SKIP      <= '1';             -- yes
                when PC_LD_S   => Q_SKIP      <= '1';             -- yes
                when PC_SKIP_Z => Q_SKIP      <= L_FLAGS_98(8);   -- if Z set
                when PC_SKIP_T => Q_SKIP      <= L_FLAGS_98(9);   -- if T set
                when others    => Q_SKIP      <= '0';             -- no.
            end case;
        end process;

        Q_ADR           <= F_ADR;
        Q_DOUT          <= A_DOUT(7 downto 0);
        Q_INT_ENA       <= A_FLAGS(7);
        Q_OPC           <= I_OPC;
        Q_PC            <= I_PC;

        Q_RD_IO         <= '0'                    when (F_ADR <X"20")
                else (I_RD_M and not I_PMS) when (F_ADR       <X"5D")
                else '0';
        Q_WE_IO          <= '0'                    when (F_ADR <X"20")
                else I_WE_M(0)              when (F_ADR       <X"5D")
                else '0';
        L_WE_SRAM        <= "00"   when  (F_ADR <X"0060") else I_WE_M;
        L_DIN            <= I_DIN  when (I_PMS = '1')
```

```
                else F_S    when  (F_ADR      <X"0020")
                else I_DIN  when  (F_ADR       <X"005D")
                else F_S    when  (F_ADR       <X"0060")
                else M_DOUT(7 downto 0);

        -- compute potential new PC value from Z, (SP), or IMM.
        --
        Q_NEW_PC       <= F_Z    when I_PC_OP = PC_LD_Z       -- IJMP, ICALL
               else M_DOUT when I_PC_OP = PC_LD_S       -- RET, RETI
               else I_JADR;                             -- JMP adr

    end Behavioral;

```

src/data_path.vhd

17 LISTING OF io.vhd

```
--------------------------------------------------------------------------------
--
-- Copyright (C) 2009, 2010 Dr. Juergen Sauermann
--
--  This code is free software: you can redistribute it and/or modify
--  it under the terms of the GNU General Public License as published by
--  the Free Software Foundation, either version 3 of the License, or
--  (at your option) any later version.
--
--  This code is distributed in the hope that it will be useful,
--  but WITHOUT ANY WARRANTY; without even the implied warranty of
--  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
--  GNU General Public License for more details.
--
--  You should have received a copy of the GNU General Public License
--  along with this code (see the file named COPYING).
--  If not, see http://www.gnu.org/licenses/.
--
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--
-- Module Name:    io - Behavioral
-- Create Date:    13:59:36 11/07/2009
-- Description:    the I/O of a CPU (uart and general purpose I/O lines).
--
--------------------------------------------------------------------------------
--
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity io is
    port (  I_CLK       : in  std_logic;

            I_CLR       : in  std_logic;
            I_ADR_IO    : in  std_logic_vector( 7 downto 0);
            I_DIN       : in  std_logic_vector( 7 downto 0);
            I_SWITCH    : in  std_logic_vector( 7 downto 0);
            I_RD_IO     : in  std_logic;
            I_RX        : in  std_logic;
            I_WE_IO     : in  std_logic;

            Q_7_SEGMENT : out std_logic_vector( 6 downto 0);
            Q_DOUT      : out std_logic_vector( 7 downto 0);
            Q_INTVEC    : out std_logic_vector( 5 downto 0);
            Q_LEDS      : out std_logic_vector( 1 downto 0);
            Q_TX        : out std_logic);
end io;

architecture Behavioral of io is

component uart
    generic(CLOCK_FREQ  : std_logic_vector(31 downto 0);
            BAUD_RATE   : std_logic_vector(27 downto 0));
    port(   I_CLK       : in  std_logic;
            I_CLR       : in  std_logic;
            I_RD        : in  std_logic;
```

```
            I_WE        : in  std_logic;
            I_RX        : in  std_logic;
            I_TX_DATA   : in  std_logic_vector(7 downto 0);

            Q_RX_DATA   : out std_logic_vector(7 downto 0);
            Q_RX_READY  : out std_logic;
            Q_TX        : out std_logic;
            Q_TX_BUSY   : out std_logic);
end component;

signal U_RX_READY       : std_logic;
signal U_TX_BUSY        : std_logic;
signal U_RX_DATA        : std_logic_vector( 7 downto 0);

signal L_INTVEC         : std_logic_vector( 5 downto 0);
signal L_LEDS           : std_logic;
signal L_RD_UART        : std_logic;
signal L_RX_INT_ENABLED : std_logic;
signal L_TX_INT_ENABLED : std_logic;
signal L_WE_UART        : std_logic;

begin
    urt: uart
    generic map(CLOCK_FREQ  => std_logic_vector(conv_unsigned(25000000, 32)),
                BAUD_RATE   => std_logic_vector(conv_unsigned(   38400, 28)))
    port map(   I_CLK      => I_CLK,
                I_CLR      => I_CLR,
                I_RD       => L_RD_UART,
                I_WE       => L_WE_UART,
                I_TX_DATA  => I_DIN(7 downto 0),
                I_RX       => I_RX,

                Q_TX       => Q_TX,
                Q_RX_DATA  => U_RX_DATA,
                Q_RX_READY => U_RX_READY,
                Q_TX_BUSY  => U_TX_BUSY);

    -- IO read process
    --
    iord: process(I_ADR_IO, I_SWITCH,
                  U_RX_DATA, U_RX_READY, L_RX_INT_ENABLED,
                  U_TX_BUSY, L_TX_INT_ENABLED)
    begin
        -- addresses for mega8 device (use iom8.h or #define __AVR_ATmega8__).
        --
        case I_ADR_IO is
            when X"2A"  => Q_DOUT       <=             -- UCSRB:
                                   L_RX_INT_ENABLED  -- Rx complete int enabled.
                                 & L_TX_INT_ENABLED  -- Tx complete int enabled.
                                 & L_TX_INT_ENABLED  -- Tx empty int enabled.
                                 & '1'               -- Rx enabled
                                 & '1'               -- Tx enabled
                                 & '0'               -- 8 bits/char
                                 & '0'               -- Rx bit 8
                                 & '0';              -- Tx bit 8
            when X"2B"  => Q_DOUT       <=             -- UCSRA:
                                   U_RX_READY       -- Rx complete
                                 & not U_TX_BUSY    -- Tx complete
                                 & not U_TX_BUSY    -- Tx ready
                                 & '0'              -- frame error
                                 & '0'              -- data overrun
                                 & '0'              -- parity error
```

```
                                 & '0'              -- double dpeed
                                 & '0';             -- multiproc mode
                when X"2C"  => Q_DOUT       <= U_RX_DATA; -- UDR
                when X"40"  => Q_DOUT       <=             -- UCSRC
                                   '1'              -- URSEL
                                 & '0'              -- asynchronous
                                 & "00"             -- no parity
                                 & '1'              -- two stop bits
                                 & "11"             -- 8 bits/char
                                 & '0';             -- rising clock edge

                when X"36"  => Q_DOUT       <= I_SWITCH;  -- PINB
                when others => Q_DOUT       <= X"AA";
            end case;
        end process;

        -- IO write process
        --
        iowr: process(I_CLK)
        begin
            if (rising_edge(I_CLK)) then
                if (I_CLR = '1') then
                    L_RX_INT_ENABLED       <= '0';
                    L_TX_INT_ENABLED       <= '0';
                elsif (I_WE_IO = '1') then
                    case I_ADR_IO is
                        when X"38"  => Q_7_SEGMENT       <= I_DIN(6 downto 0);    -- PORTB
                                       L_LEDS       <= not L_LEDS;
                        when X"40"  =>  -- handled by uart
                        when X"41"  =>  -- handled by uart
                        when X"43"  => L_RX_INT_ENABLED      <= I_DIN(0);
                                       L_TX_INT_ENABLED      <= I_DIN(1);
                        when others =>
                    end case;
                end if;
            end if;
        end process;

        -- interrupt process
        --
        ioint: process(I_CLK)
        begin
            if (rising_edge(I_CLK)) then
                if (I_CLR = '1') then
                    L_INTVEC      <= "000000";
                else
                    if (L_RX_INT_ENABLED and U_RX_READY) = '1' then
                        if (L_INTVEC(5) = '0') then     -- no interrupt pending
                            L_INTVEC      <= "101011";       -- _VECTOR(11)
                        end if;
                    elsif (L_TX_INT_ENABLED and not U_TX_BUSY) = '1' then
                        if (L_INTVEC(5) = '0') then     -- no interrupt pending
                            L_INTVEC      <= "101100";       -- _VECTOR(12)
                        end if;
                    else                                -- no interrupt
                        L_INTVEC      <= "000000";
                    end if;
                end if;
            end if;
        end process;

        L_WE_UART      <= I_WE_IO when (I_ADR_IO = X"2C") else '0'; -- write UART UDR
```

```
        L_RD_UART       <= I_RD_IO when (I_ADR_IO = X"2C") else '0'; -- read  UART UDR

        Q_LEDS(1)       <= L_LEDS;
        Q_LEDS(0)       <= not L_LEDS;
        Q_INTVEC        <= L_INTVEC;

    end Behavioral;

```

src/io.vhd

18 Listing of opc_deco.vhd

```
--------------------------------------------------------------------------------
--
-- Copyright (C) 2009, 2010 Dr. Juergen Sauermann
--
--  This code is free software: you can redistribute it and/or modify
--  it under the terms of the GNU General Public License as published by
--  the Free Software Foundation, either version 3 of the License, or
--  (at your option) any later version.
--
--  This code is distributed in the hope that it will be useful,
--  but WITHOUT ANY WARRANTY; without even the implied warranty of
--  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
--  GNU General Public License for more details.
--
--  You should have received a copy of the GNU General Public License
--  along with this code (see the file named COPYING).
--  If not, see http://www.gnu.org/licenses/.
--
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--
-- Module Name:    opc_deco - Behavioral
-- Create Date:    16:05:16 10/29/2009
-- Description:    the opcode decoder of a CPU.
--
--------------------------------------------------------------------------------
--
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

use work.common.ALL;

entity opc_deco is
    port (  I_CLK       : in  std_logic;

            I_OPC       : in  std_logic_vector(31 downto 0);
            I_PC        : in  std_logic_vector(15 downto 0);
            I_T0        : in  std_logic;

            Q_ALU_OP    : out std_logic_vector( 4 downto 0);
            Q_AMOD      : out std_logic_vector( 5 downto 0);
            Q_BIT       : out std_logic_vector( 3 downto 0);
            Q_DDDDD     : out std_logic_vector( 4 downto 0);
            Q_IMM       : out std_logic_vector(15 downto 0);
            Q_JADR      : out std_logic_vector(15 downto 0);
            Q_OPC       : out std_logic_vector(15 downto 0);
            Q_PC        : out std_logic_vector(15 downto 0);
            Q_PC_OP     : out std_logic_vector( 2 downto 0);
            Q_PMS       : out std_logic;  -- program memory select
            Q_RD_M      : out std_logic;
            Q_RRRRR     : out std_logic_vector( 4 downto 0);
            Q_RSEL      : out std_logic_vector( 1 downto 0);
            Q_WE_01     : out std_logic;
            Q_WE_D      : out std_logic_vector( 1 downto 0);
            Q_WE_F      : out std_logic;
            Q_WE_M      : out std_logic_vector( 1 downto 0);
```

```
            Q_WE_XYZS   : out std_logic);
end opc_deco;

architecture Behavioral of opc_deco is

begin

    process(I_CLK)
    begin
    if (rising_edge(I_CLK)) then
        --
        -- set the most common settings as default.
        --
        Q_ALU_OP        <= ALU_D_MV_Q;
        Q_AMOD          <= AMOD_ABS;
        Q_BIT           <= I_OPC(10) & I_OPC(2 downto 0);
        Q_DDDDD         <= I_OPC(8 downto 4);
        Q_IMM           <= X"0000";
        Q_JADR          <= I_OPC(31 downto 16);
        Q_OPC           <= I_OPC(15 downto  0);
        Q_PC            <= I_PC;
        Q_PC_OP         <= PC_NEXT;
        Q_PMS           <= '0';
        Q_RD_M          <= '0';
        Q_RRRRR         <= I_OPC(9) & I_OPC(3 downto 0);
        Q_RSEL          <= RS_REG;
        Q_WE_D          <= "00";
        Q_WE_01         <= '0';
        Q_WE_F          <= '0';
        Q_WE_M          <= "00";
        Q_WE_XYZS       <= '0';

        case I_OPC(15 downto 10) is
            when "000000" =>
                case I_OPC(9 downto 8) is
                    when "00" =>
                        --
                        -- 0000 0000 0000 0000 - NOP
                        -- 0000 0000 001v vvvv - INTERRUPT
                        --
                        if (I_OPC(5)) = '1' then   -- interrupt
                            Q_ALU_OP       <= ALU_INTR;
                            Q_AMOD       <= AMOD_ddSP;
                            Q_JADR       <= "0000000000" & I_OPC(4 downto 0) & "0";
                            Q_PC_OP       <= PC_LD_I;
                            Q_WE_F       <= '1';
                            Q_WE_M       <= "11";
                        end if;

                    when "01" =>
                        --
                        -- 0000 0001 dddd rrrr - MOVW
                        --
                        Q_DDDDD       <= I_OPC(7 downto 4) & "0";
                        Q_RRRRR       <= I_OPC(3 downto 0) & "0";
                        Q_ALU_OP       <= ALU_MV_16;
                        Q_WE_D       <= "11";

                    when "10" =>
                        --
                        -- 0000 0010 dddd rrrr - MULS
                        --
```

```
            Q_DDDDD      <= "1" & I_OPC(7 downto 4);
            Q_RRRRR      <= "1" & I_OPC(3 downto 0);
            Q_ALU_OP      <= ALU_MULT;
            Q_IMM(7 downto 5)      <= MULT_SS;
            Q_WE_01      <= '1';
            Q_WE_F      <= '1';

        when others =>
            --
            -- 0000 0011 0ddd 0rrr - _MULSU  SU "010"
            -- 0000 0011 0ddd 1rrr - FMUL    UU "100"
            -- 0000 0011 1ddd 0rrr - FMULS   SS "111"
            -- 0000 0011 1ddd 1rrr - FMULSU  SU "110"
            --
            Q_DDDDD(4 downto 3)      <= "10";    -- regs 16 to 23
            Q_RRRRR(4 downto 3)      <= "10";    -- regs 16 to 23
            Q_ALU_OP      <= ALU_MULT;
            if I_OPC(7) = '0' then
                if I_OPC(3) = '0' then
                    Q_IMM(7 downto 5)      <= MULT_SU;
                else
                    Q_IMM(7 downto 5)      <= MULT_FUU;
                end if;
            else
                if I_OPC(3) = '0' then
                    Q_IMM(7 downto 5)      <= MULT_FSS;
                else
                    Q_IMM(7 downto 5)      <= MULT_FSU;
                end if;
            end if;
            Q_WE_01      <= '1';
            Q_WE_F      <= '1';
    end case;

when "000001" | "000010" =>
    --
    -- 0000 01rd dddd rrrr - CPC = SBC without Q_WE_D
    -- 0000 10rd dddd rrrr - SBC
    --
    Q_ALU_OP      <= ALU_SBC;
    Q_WE_D      <= '0' & I_OPC(11);  -- write Rd if SBC.
    Q_WE_F      <= '1';

when "000011" =>
    --
    -- 0000 11rd dddd rrrr - ADD
    --
    Q_ALU_OP      <= ALU_ADD;
    Q_WE_D      <= "01";
    Q_WE_F      <= '1';

when "000100" => -- CPSE
    Q_ALU_OP      <= ALU_SUB;
    Q_RD_M      <= I_T0;
    if (I_T0 = '0') then        -- second cycle.
        Q_PC_OP      <= PC_SKIP_Z;
    end if;

when "000101" | "000110" =>
    --
    -- 0001 01rd dddd rrrr - CP = SUB without Q_WE_D
    -- 0000 10rd dddd rrrr - SUB
```

```
                --
                Q_ALU_OP      <= ALU_SUB;
                Q_WE_D        <= '0' & I_OPC(11);  -- write Rd if SUB.
                Q_WE_F        <= '1';

            when "000111" =>
                --
                -- 0001 11rd dddd rrrr - ADC
                --
                Q_ALU_OP      <= ALU_ADC;
                Q_WE_D        <= "01";
                Q_WE_F        <= '1';

            when "001000" =>
                --
                -- 0010 00rd dddd rrrr - AND
                --
                Q_ALU_OP      <= ALU_AND;
                Q_WE_D        <= "01";
                Q_WE_F        <= '1';

            when "001001" =>
                --
                -- 0010 01rd dddd rrrr - EOR
                --
                Q_ALU_OP      <= ALU_EOR;
                Q_WE_D        <= "01";
                Q_WE_F        <= '1';

            when "001010" => -- OR
                --
                -- 0010 10rd dddd rrrr - OR
                --
                Q_ALU_OP      <= ALU_OR;
                Q_WE_D        <= "01";
                Q_WE_F        <= '1';

            when "001011" =>
                --
                -- 0010 11rd dddd rrrr - MOV
                --
                Q_ALU_OP      <= ALU_R_MV_Q;
                Q_WE_D        <= "01";

            when "001100" | "001101" | "001110" | "001111"
               | "010100" | "010101" | "010110" | "010111" =>
                --
                -- 0011 KKKK dddd KKKK - CPI
                -- 0101 KKKK dddd KKKK - SUBI
                --
                Q_ALU_OP      <= ALU_SUB;
                Q_IMM(7 downto 0)      <= I_OPC(11 downto 8) & I_OPC(3 downto 0);
                Q_RSEL        <= RS_IMM;
                Q_DDDDD(4)      <= '1';    -- Rd = 16...31
                Q_WE_D        <= '0' & I_OPC(14);
                Q_WE_F        <= '1';

            when "010000" | "010001" | "010010" | "010011" =>
                --
                -- 0100 KKKK dddd KKKK - SBCI
                --
                Q_ALU_OP      <= ALU_SBC;
```

```
                Q_IMM(7 downto 0)      <= I_OPC(11 downto 8) & I_OPC(3 downto 0);
                Q_RSEL      <= RS_IMM;
                Q_DDDDD(4)      <= '1';    -- Rd = 16...31
                Q_WE_D      <= "01";
                Q_WE_F      <= '1';

            when "011000" | "011001" | "011010" | "011011" =>
                --
                -- 0110 KKKK dddd KKKK - ORI
                --
                Q_ALU_OP      <= ALU_OR;
                Q_IMM(7 downto 0)      <= I_OPC(11 downto 8) & I_OPC(3 downto 0);
                Q_RSEL      <= RS_IMM;
                Q_DDDDD(4)      <= '1';    -- Rd = 16...31
                Q_WE_D      <= "01";
                Q_WE_F      <= '1';

            when "011100" | "011101" | "011110" | "011111" =>
                --
                -- 0111 KKKK dddd KKKK - ANDI
                --
                Q_ALU_OP      <= ALU_AND;
                Q_IMM(7 downto 0)      <= I_OPC(11 downto 8) & I_OPC(3 downto 0);
                Q_RSEL      <= RS_IMM;
                Q_DDDDD(4)      <= '1';    -- Rd = 16...31
                Q_WE_D      <= "01";
                Q_WE_F      <= '1';

            when "100000" | "100001" | "100010" | "100011"
               | "101000" | "101001" | "101010" | "101011" =>
                --
                -- LDD (Y + q) == LD (y) if q == 0
                --
                -- 10q0 qq0d dddd 1qqq  LDD (Y + q)
                -- 10q0 qq0d dddd 0qqq  LDD (Z + q)
                -- 10q0 qq1d dddd 1qqq  SDD (Y + q)
                -- 10q0 qq1d dddd 0qqq  SDD (Z + q)
                --        L/      Z/
                --        S       Y
                --
                Q_IMM(5)      <= I_OPC(13);
                Q_IMM(4 downto 3)      <= I_OPC(11 downto 10);
                Q_IMM(2 downto 0)      <= I_OPC( 2 downto  0);

                if (I_OPC(3) = '0') then    Q_AMOD      <= AMOD_Zq;
                else                        Q_AMOD      <= AMOD_Yq;
                end if;

                Q_RD_M      <= not I_OPC(9);             -- '1'  if LDD
                Q_WE_M      <= '0' & I_OPC(9);           -- "01" if STD

            when "100100" =>
                Q_IMM      <= I_OPC(31 downto 16);   -- absolute address for LDS/STS
                if (I_OPC(9) = '0') then        -- LDD / POP
                    --
                    -- 1001 00-0d dddd 0000 - LDS
                    -- 1001 00-0d dddd 0001 - LD Rd, Z+
                    -- 1001 00-0d dddd 0010 - LD Rd, -Z
                    -- 1001 00-0d dddd 0100 - (ii)  LPM Rd, (Z)
                    -- 1001 00-0d dddd 0101 - (iii) LPM Rd, (Z+)
                    -- 1001 00-0d dddd 0110 - ELPM Z        --- not mega8
                    -- 1001 00-0d dddd 0111 - ELPM Z+       --- not mega8
```

```
                -- 1001 00-0d dddd 1001 - LD Rd, Y+
                -- 1001 00-0d dddd 1010 - LD Rd, -Y
                -- 1001 00-0d dddd 1100 - LD Rd, X
                -- 1001 00-0d dddd 1101 - LD Rd, X+
                -- 1001 00-0d dddd 1110 - LD Rd, -X
                -- 1001 00-0d dddd 1111 - POP Rd
                --
                Q_RSEL      <= RS_DIN;
                Q_RD_M      <= I_T0;
                Q_WE_D      <= '0' & not I_T0;
                Q_WE_XYZS      <= not I_T0;
                Q_PMS      <= (not I_OPC(3)) and I_OPC(2) and (not I_OPC(1));
                case I_OPC(3 downto 0) is
                    when "0000" => Q_AMOD      <= AMOD_ABS;  Q_WE_XYZS <= '0';
                    when "0001" => Q_AMOD      <= AMOD_Zi;
                    when "0100" => Q_AMOD      <= AMOD_Z;    Q_WE_XYZS <= '0';
                    when "0101" => Q_AMOD      <= AMOD_Zi;
                    when "1001" => Q_AMOD      <= AMOD_Yi;
                    when "1010" => Q_AMOD      <= AMOD_dY;
                    when "1100" => Q_AMOD      <= AMOD_X;    Q_WE_XYZS <= '0';
                    when "1101" => Q_AMOD      <= AMOD_Xi;
                    when "1110" => Q_AMOD      <= AMOD_dX;
                    when "1111" => Q_AMOD      <= AMOD_SPi;
                    when others =>                      Q_WE_XYZS      <= '0';
                end case;
            else                        -- STD / PUSH
                --
                -- 1001 00-1r rrrr 0000 - STS
                -- 1001 00-1r rrrr 0001 - ST Z+. Rr
                -- 1001 00-1r rrrr 0010 - ST -Z. Rr
                -- 1001 00-1r rrrr 1000 - ST Y. Rr
                -- 1001 00-1r rrrr 1001 - ST Y+. Rr
                -- 1001 00-1r rrrr 1010 - ST -Y. Rr
                -- 1001 00-1r rrrr 1100 - ST X. Rr
                -- 1001 00-1r rrrr 1101 - ST X+. Rr
                -- 1001 00-1r rrrr 1110 - ST -X. Rr
                -- 1001 00-1r rrrr 1111 - PUSH Rr
                --
                Q_ALU_OP      <= ALU_D_MV_Q;
                Q_WE_M      <= "01";
                Q_WE_XYZS      <= '1';
                case I_OPC(3 downto 0) is
                    when "0000" => Q_AMOD      <= AMOD_ABS;  Q_WE_XYZS <= '0';
                    when "0001" => Q_AMOD      <= AMOD_Zi;
                    when "0010" => Q_AMOD      <= AMOD_dZ;
                    when "1001" => Q_AMOD      <= AMOD_Yi;
                    when "1010" => Q_AMOD      <= AMOD_dY;
                    when "1100" => Q_AMOD      <= AMOD_X;    Q_WE_XYZS <= '0';
                    when "1101" => Q_AMOD      <= AMOD_Xi;
                    when "1110" => Q_AMOD      <= AMOD_dX;
                    when "1111" => Q_AMOD      <= AMOD_dSP;
                    when others =>
                end case;
            end if;

        when "100101" =>
            if (I_OPC(9) = '0') then
                if (I_OPC(3) = '0') then
                    --
                    --  1001 010d dddd 0000 - COM
                    --  1001 010d dddd 0001 - NEG
                    --  1001 010d dddd 0010 - SWAP
```

```
                    --  1001 010d dddd 0011 - INC
                    --  1001 010d dddd 0101 - ASR
                    --  1001 010d dddd 0110 - LSR
                    --  1001 010d dddd 0111 - ROR
                    --
                    case I_OPC(2 downto 0) is
                        when "000"  => Q_ALU_OP       <= ALU_COM;
                        when "001"  => Q_ALU_OP       <= ALU_NEG;
                        when "010"  => Q_ALU_OP       <= ALU_SWAP;
                        when "011"  => Q_ALU_OP       <= ALU_INC;
                        when "101"  => Q_ALU_OP       <= ALU_ASR;
                        when "110"  => Q_ALU_OP       <= ALU_LSR;
                        when "111"  => Q_ALU_OP       <= ALU_ROR;
                        when others =>
                    end case;
                    Q_WE_D       <= "01";
                    Q_WE_F       <= '1';
                else
                    case I_OPC(2 downto 0) is
                        when "000"  =>
                            if (I_OPC(8)) = '0' then
                                --
                                --  1001 0100 0sss 1000 - BSET
                                --  1001 0100 1sss 1000 - BCLR
                                --
                                Q_BIT(3 downto 0)       <= I_OPC(7 downto 4);
                                Q_ALU_OP       <= ALU_SREG;
                                Q_WE_F       <= '1';
                            else
                                --
                                --  1001 0101 0000 1000 - RET
                                --  1001 0101 0001 1000 - RETI
                                --  1001 0101 1000 1000 - SLEEP
                                --  1001 0101 1001 1000 - BREAK
                                --  1001 0101 1100 1000 - LPM     [ R0,(Z) ]
                                --  1001 0101 1101 1000 - ELPM   not mega8
                                --  1001 0101 1110 1000 - SPM
                                --  1001 0101 1111 1000 - SPM #2
                                --  1001 0101 1010 1000 - WDR
                                --
                                case I_OPC(7 downto 4) is
                                    when "0000" =>  -- RET
                                        Q_AMOD       <= AMOD_SPii;
                                        if (I_T0 = '1') then
                                            Q_RD_M       <= '1';
                                        else
                                            Q_PC_OP       <= PC_LD_S;
                                            Q_WE_XYZS       <= not I_T0;
                                        end if;

                                    when "0001" =>  -- RETI
                                        Q_ALU_OP       <= ALU_INTR;
                                        Q_IMM(6)       <= '1';
                                        Q_AMOD       <= AMOD_SPii;
                                        if (I_T0 = '1') then
                                            Q_RD_M       <= '1';
                                        else
                                            Q_PC_OP       <= PC_LD_S;
                                            Q_WE_XYZS       <= not I_T0;
                                        end if;

                                    when "1000" =>  -- (i) LPM R0, (Z)
```

```
                            Q_DDDDD      <= "00000";
                            Q_AMOD      <= AMOD_Z;
                            Q_PMS      <= '1';
                            Q_WE_D      <= '0' & not I_T0;

                        when "1110" =>  -- SPM
                            Q_DDDDD      <= "00000";
                            Q_AMOD      <= AMOD_Z;
                            Q_PMS      <= '1';
                            Q_WE_M      <= "01";

                        when "1111" =>  -- SPM #2
                            -- page write: not su[pported

                        when others =>
                    end case;
                end if;

            when "001"  =>
                --
                --  1001 0100 0000 1001 IJMP
                --  1001 0100 0001 1001 EIJMP   -- not mega8
                --  1001 0101 0000 1001 ICALL
                --  1001 0101 0001 1001 EICALL   -- not mega8
                --
                Q_PC_OP      <= PC_LD_Z;
                if (I_OPC(8) = '1') then        -- ICALL
                    Q_ALU_OP      <= ALU_PC_1;
                    Q_AMOD      <= AMOD_ddSP;
                    Q_WE_M      <= "11";
                    Q_WE_XYZS      <= '1';
                end if;

            when "010"  =>
                --
                --  1001 010d dddd 1010 - DEC
                --
                Q_ALU_OP      <= ALU_DEC;
                Q_WE_D      <= "01";
                Q_WE_F      <= '1';

            when "011"  =>
                --
                --  1001 0100 KKKK 1011 - DES   -- not mega8
                --

            when "100" | "101"  =>
                --
                --  1001 010k kkkk 110k - JMP (k = 0 for 16 bit)
                --  kkkk kkkk kkkk kkkk
                --
                Q_PC_OP      <= PC_LD_I;

            when "110" | "111"  =>
                --
                --  1001 010k kkkk 111k - CALL (k = 0)
                --  kkkk kkkk kkkk kkkk
                --
                Q_ALU_OP      <= ALU_PC_2;
                Q_AMOD      <= AMOD_ddSP;
                Q_PC_OP      <= PC_LD_I;
                Q_WE_M      <= "11";     -- both PC bytes
```

```
                                Q_WE_XYZS       <= '1';

                            when others =>
                        end case;
                    end if;
                else
                    --
                    --  1001 0110 KKdd KKKK - ADIW
                    --  1001 0111 KKdd KKKK - SBIW
                    --
                    if (I_OPC(8) = '0') then    Q_ALU_OP       <= ALU_ADIW;
                    else                        Q_ALU_OP       <= ALU_SBIW;
                    end if;
                    Q_IMM(5 downto 4)       <= I_OPC(7 downto 6);
                    Q_IMM(3 downto 0)       <= I_OPC(3 downto 0);
                    Q_RSEL      <= RS_IMM;
                    Q_DDDDD      <= "11" & I_OPC(5 downto 4) & "0";

                    Q_WE_D      <= "11";
                    Q_WE_F      <= '1';
                end if; -- I_OPC(9) = 0/1

            when "100110" =>
                --
                --  1001 1000 AAAA Abbb - CBI
                --  1001 1001 AAAA Abbb - SBIC
                --  1001 1010 AAAA Abbb - SBI
                --  1001 1011 AAAA Abbb - SBIS
                --
                Q_ALU_OP       <= ALU_BIT_CS;
                Q_AMOD      <= AMOD_ABS;
                Q_BIT(3)       <= I_OPC(9);   -- set/clear

                -- IMM = AAAAAA + 0x20
                --
                Q_IMM(4 downto 0)       <= I_OPC(7 downto 3);
                Q_IMM(6 downto 5)       <= "01";

                Q_RD_M      <= I_T0;
                if ((I_OPC(8) = '0') ) then     -- CBI or SBI
                    Q_WE_M(0)       <= '1';
                else                            -- SBIC or SBIS
                    if (I_T0 = '0') then        -- second cycle.
                        Q_PC_OP       <= PC_SKIP_T;
                    end if;
                end if;

            when "100111" => -- MUL
                --
                --  1001 11rd dddd rrrr - MUL
                --
                 Q_ALU_OP       <= ALU_MULT;
                 Q_IMM(7 downto 5)       <= "000"; --  -MUL UU;
                 Q_WE_01       <= '1';
                 Q_WE_F       <= '1';

            when "101100" | "101101" =>
                --
                -- 1011 0AAd dddd AAAA - IN
                --
                Q_RSEL      <= RS_DIN;
                Q_AMOD      <= AMOD_ABS;
```

```

                    -- IMM = AAAAAA
                    --     + 010000 (0x20)
                    Q_IMM(3 downto 0)      <= I_OPC(3 downto 0);
                    Q_IMM(4)      <= I_OPC(9);
                    Q_IMM(6 downto 5)      <= "01" + ('0' & I_OPC(10 downto 10));

                    Q_WE_D      <= "01";

                when "101110" | "101111" =>
                    --
                    -- 1011 1AAr rrrr AAAA - OUT
                    --
                    Q_ALU_OP      <= ALU_D_MV_Q;
                    Q_AMOD      <= AMOD_ABS;

                    -- IMM = AAAAAA
                    --     + 010000 (0x20)
                    --
                    Q_IMM(3 downto 0)      <= I_OPC(3 downto 0);
                    Q_IMM(4)      <= I_OPC(9);
                    Q_IMM(6 downto 5)      <= "01" + ('0' & I_OPC(10 downto 10));
                    Q_WE_M      <= "01";

                when "110000" | "110001" | "110010" | "110011" =>
                    --
                    -- 1100 kkkk kkkk kkkk - RJMP
                    --
                    Q_JADR      <= I_PC + (I_OPC(11) & I_OPC(11) & I_OPC(11) & I_OPC(11)
                                    & I_OPC(11 downto 0)) + X"0001";
                    Q_PC_OP      <= PC_LD_I;

                when "110100" | "110101" | "110110" | "110111" =>
                    --
                    -- 1101 kkkk kkkk kkkk - RCALL
                    --
                    Q_JADR      <= I_PC + (I_OPC(11) & I_OPC(11) & I_OPC(11) & I_OPC(11)
                                    & I_OPC(11 downto 0)) + X"0001";
                    Q_ALU_OP      <= ALU_PC_1;
                    Q_AMOD      <= AMOD_ddSP;
                    Q_PC_OP      <= PC_LD_I;
                    Q_WE_M      <= "11";     -- both PC bytes
                    Q_WE_XYZS      <= '1';

                when "111000" | "111001" | "111010" | "111011" => -- LDI
                    --
                    -- 1110 KKKK dddd KKKK - LDI Rd, K
                    --
                    Q_ALU_OP      <= ALU_R_MV_Q;
                    Q_RSEL      <= RS_IMM;
                    Q_DDDDD      <= '1' & I_OPC(7 downto 4);     -- 16..31
                    Q_IMM(7 downto 0)      <= I_OPC(11 downto 8) & I_OPC(3 downto 0);
                    Q_WE_D      <= "01";

                when "111100" | "111101" =>
                    --
                    -- 1111 00kk kkkk kbbb - BRBS
                    -- 1111 01kk kkkk kbbb - BRBC
                    --       v
                    -- bbb: status register bit
                    -- v: value (set/cleared) of status register bit
                    --
```

```
                    Q_JADR      <= I_PC + (I_OPC(9) & I_OPC(9) & I_OPC(9) & I_OPC(9)
                                    & I_OPC(9) & I_OPC(9) & I_OPC(9) & I_OPC(9)
                                    & I_OPC(9) & I_OPC(9 downto 3)) + X"0001";
                    Q_PC_OP      <= PC_BCC;

                when "111110" =>
                    --
                    -- 1111 100d dddd 0bbb - BLD
                    -- 1111 101d dddd 0bbb - BST
                    --
                    if I_OPC(9) = '0' then  -- BLD: T flag to register
                        Q_ALU_OP      <= ALU_BLD;
                        Q_WE_D      <= "01";
                    else                    -- BST: register to T flag
                        Q_AMOD      <= AMOD_ABS;
                        Q_BIT(3)      <= I_OPC(10);
                        Q_IMM(4 downto 0)      <= I_OPC(8 downto 4);
                        Q_ALU_OP      <= ALU_BIT_CS;
                        Q_WE_F      <= '1';
                    end if;

                when "111111" =>
                    --
                    -- 1111 110r rrrr 0bbb - SBRC
                    -- 1111 111r rrrr 0bbb - SBRS
                    --
                    -- like SBIC, but and general purpose regs instead of I/O regs.
                    --
                    Q_ALU_OP      <= ALU_BIT_CS;
                    Q_AMOD      <= AMOD_ABS;
                    Q_BIT(3)      <= I_OPC(9);   -- set/clear bit
                    Q_IMM(4 downto 0)      <= I_OPC(8 downto 4);
                    if (I_T0 = '0') then
                        Q_PC_OP      <= PC_SKIP_T;
                    end if;

                when others =>
            end case;
        end if;
        end process;

    end Behavioral;

```

src/opc_deco.vhd

19 LISTING OF opc_fetch.vhd

```
-------------------------------------------------------------------------------
--
-- Copyright (C) 2009, 2010 Dr. Juergen Sauermann
--
--  This code is free software: you can redistribute it and/or modify
--  it under the terms of the GNU General Public License as published by
--  the Free Software Foundation, either version 3 of the License, or
--  (at your option) any later version.
--
--  This code is distributed in the hope that it will be useful,
--  but WITHOUT ANY WARRANTY; without even the implied warranty of
--  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
--  GNU General Public License for more details.
--
--  You should have received a copy of the GNU General Public License
--  along with this code (see the file named COPYING).
--  If not, see http://www.gnu.org/licenses/.
--
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
--
-- Module Name:    opc_fetch - Behavioral
-- Create Date:    13:00:44 10/30/2009
-- Description:    the opcode fetch stage of a CPU.
--
-------------------------------------------------------------------------------
--
library IEEE;
use IEEE.std_logic_1164.ALL;
use IEEE.std_logic_ARITH.ALL;
use IEEE.std_logic_UNSIGNED.ALL;

entity opc_fetch is
    port (  I_CLK       : in  std_logic;

            I_CLR       : in  std_logic;
            I_INTVEC    : in  std_logic_vector( 5 downto 0);
            I_LOAD_PC   : in  std_logic;
            I_NEW_PC    : in  std_logic_vector(15 downto 0);
            I_PM_ADR    : in  std_logic_vector(11 downto 0);
            I_SKIP      : in  std_logic;

            Q_OPC       : out std_logic_vector(31 downto 0);
            Q_PC        : out std_logic_vector(15 downto 0);
            Q_PM_DOUT   : out std_logic_vector( 7 downto 0);
            Q_T0        : out std_logic);
end opc_fetch;

architecture Behavioral of opc_fetch is

component prog_mem
    port (  I_CLK       : in  std_logic;

            I_WAIT      : in  std_logic;
            I_PC        : in  std_logic_vector (15 downto 0);
            I_PM_ADR    : in  std_logic_vector (11 downto 0);

            Q_OPC       : out std_logic_vector (31 downto 0);
```

```
                Q_PC        : out std_logic_vector (15 downto 0);
                Q_PM_DOUT   : out std_logic_vector ( 7 downto 0));
    end component;

    signal P_OPC            : std_logic_vector(31 downto 0);
    signal P_PC             : std_logic_vector(15 downto 0);

    signal L_INVALIDATE     : std_logic;
    signal L_LONG_OP        : std_logic;
    signal L_NEXT_PC        : std_logic_vector(15 downto 0);
    signal L_PC             : std_logic_vector(15 downto 0);
    signal L_T0             : std_logic;
    signal L_WAIT           : std_logic;

    begin

        pmem : prog_mem
        port map(   I_CLK       => I_CLK,

                    I_WAIT      => L_WAIT,
                    I_PC        => L_NEXT_PC,
                    I_PM_ADR    => I_PM_ADR,

                    Q_OPC       => P_OPC,
                    Q_PC        => P_PC,
                    Q_PM_DOUT   => Q_PM_DOUT);

       lpc: process(I_CLK)
        begin
            if (rising_edge(I_CLK)) then
                L_PC      <= L_NEXT_PC;
                L_T0      <= not L_WAIT;
            end if;
        end process;

        L_NEXT_PC       <= X"0000"        when (I_CLR     = '1')
                else L_PC           when (L_WAIT    = '1')
                else I_NEW_PC       when (I_LOAD_PC = '1')
                else L_PC + X"0002" when (L_LONG_OP = '1')
                else L_PC + X"0001";

        -- Two word opcodes:
        --
        --        9       3210
        -- 1001 000d dddd 0000 kkkk kkkk kkkk kkkk - LDS
        -- 1001 001d dddd 0000 kkkk kkkk kkkk kkkk - SDS
        -- 1001 010k kkkk 110k kkkk kkkk kkkk kkkk - JMP
        -- 1001 010k kkkk 111k kkkk kkkk kkkk kkkk - CALL
        --
        L_LONG_OP       <= '1' when (((P_OPC(15 downto  9) = "1001010") and
                                  (P_OPC( 3 downto  2) = "11"))       -- JMP, CALL
                             or  ((P_OPC(15 downto 10) = "100100") and
                                  (P_OPC( 3 downto  0) = "0000")))    -- LDS, STS
                else '0';

        -- Two cycle opcodes:
        --
        -- 1001 000d dddd .... - LDS etc.
        -- 1001 0101 0000 1000 - RET
        -- 1001 0101 0001 1000 - RETI
        -- 1001 1001 AAAA Abbb - SBIC
        -- 1001 1011 AAAA Abbb - SBIS
```

```
        -- 1111 110r rrrr 0bbb - SBRC
        -- 1111 111r rrrr 0bbb - SBRS
        --
        L_WAIT      <= '0'  when (L_INVALIDATE = '1')
             else '0'  when (I_INTVEC(5)  = '1')
             else L_T0 when ((P_OPC(15 downto   9) = "1001000" )    -- LDS etc.
                         or  (P_OPC(15 downto   8) = "10010101")    -- RET etc.
                         or  ((P_OPC(15 downto 10) = "100110")      -- SBIC, SBIS
                           and P_OPC(8) = '1')
                         or  (P_OPC(15 downto  10) = "111111"))     -- SBRC, SBRS
            else  '0';

        L_INVALIDATE      <= I_CLR or I_SKIP;

        Q_OPC      <= X"00000000" when (L_INVALIDATE = '1')
            else P_OPC       when (I_INTVEC(5) = '0')
            else (X"000000" & "00" & I_INTVEC);     -- "interrupt opcode"

        Q_PC      <= P_PC;
        Q_T0      <= L_T0;

    end Behavioral;

```

src/opc_fetch.vhd

19 LISTING OF opc_fetch.vhd

20 LISTING OF prog_mem_content.vhd

```

library IEEE;
use IEEE.STD_LOGIC_1164.all;

package prog_mem_content is

-- content of pe_0 ----------------------------------------------------------------------
constant pe_0_00 : BIT_VECTOR := X"F180C8135798FC118181E0CA1010905100EF1D2C5F8CCCCCCCCCCC
constant pe_0_01 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFF3E5864BEFFF1194EECFF8514CE38
constant pe_0_02 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
constant pe_0_03 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
constant pe_0_04 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
constant pe_0_05 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
constant pe_0_06 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
constant pe_0_07 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
constant pe_0_08 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
constant pe_0_09 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
constant pe_0_0A : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
constant pe_0_0B : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
constant pe_0_0C : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
constant pe_0_0D : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
constant pe_0_0E : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
constant pe_0_0F : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF

-- content of pe_1 ----------------------------------------------------------------------
constant pe_1_00 : BIT_VECTOR := X"3F930800000088842808F0CBEA0ADA0FB1DC1072644000000000000
constant pe_1_01 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFAC9CBC0DC0C0DC8F0ED109C2FF00
constant pe_1_02 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
constant pe_1_03 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
constant pe_1_04 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
constant pe_1_05 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
constant pe_1_06 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
constant pe_1_07 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
constant pe_1_08 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
constant pe_1_09 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
constant pe_1_0A : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
constant pe_1_0B : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
constant pe_1_0C : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
constant pe_1_0D : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
constant pe_1_0E : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
constant pe_1_0F : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF

-- content of pe_2 ----------------------------------------------------------------------
constant pe_2_00 : BIT_VECTOR := X"F7100B00000044404050F0007606760000F54AC0C05444444444444
constant pe_2_01 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFF0101B55F5117B71141335B711F05
constant pe_2_02 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
constant pe_2_03 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
constant pe_2_04 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
constant pe_2_05 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
constant pe_2_06 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
constant pe_2_07 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
constant pe_2_08 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
constant pe_2_09 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
constant pe_2_0A : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
constant pe_2_0B : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
constant pe_2_0C : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
constant pe_2_0D : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
constant pe_2_0E : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
```

```
constant pe_2_0F : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF

-- content of pe_3 ---------------------------------------------------------------
constant pe_3_00 : BIT_VECTOR := X"4FEECBCCCCCC33FFFF9EC000F3CEF39EEEBE2062624999999999999999999
constant pe_3_01 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFC00009EEBE9990F90909990F90CC9E04FEEC
constant pe_3_02 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
constant pe_3_03 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
constant pe_3_04 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
constant pe_3_05 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
constant pe_3_06 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
constant pe_3_07 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
constant pe_3_08 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
constant pe_3_09 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
constant pe_3_0A : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
constant pe_3_0B : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
constant pe_3_0C : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
constant pe_3_0D : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
constant pe_3_0E : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
constant pe_3_0F : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF

-- content of po_0 ---------------------------------------------------------------
constant po_0_00 : BIT_VECTOR := X"3F1A0560796F99750980CDCCE1D001D2A0D4F047C0C333333333333333333
constant po_0_01 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFF6EAE400D48FFE081B2CFF1421CDC098F1A0
constant po_0_02 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
constant po_0_03 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
constant po_0_04 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
constant po_0_05 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
constant po_0_06 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
constant po_0_07 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
constant po_0_08 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
constant po_0_09 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
constant po_0_0A : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
constant po_0_0B : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
constant po_0_0C : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
constant po_0_0D : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
constant po_0_0E : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
constant po_0_0F : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF

-- content of po_1 ---------------------------------------------------------------
constant po_1_00 : BIT_VECTOR := X"82082B8888885408888985000B1B1B00EACD1065604555555555555555555
constant po_1_01 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFF0F00D1CD01DCC82508C0082025A9B22082
constant po_1_02 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
constant po_1_03 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
constant po_1_04 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
constant po_1_05 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
constant po_1_06 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
constant po_1_07 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
constant po_1_08 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
constant po_1_09 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
constant po_1_0A : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
constant po_1_0B : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
constant po_1_0C : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
constant po_1_0D : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
constant po_1_0E : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
constant po_1_0F : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF

-- content of po_2 ---------------------------------------------------------------
constant po_2_00 : BIT_VECTOR := X"0F7609BCC88F400424409B444720072096F0E01FF8C000000000000000000
constant po_2_01 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFF4F4000F05111B36001337B369B107CF760
constant po_2_02 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
constant po_2_03 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
constant po_2_04 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
```

```
    constant po_2_05 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
    constant po_2_06 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
    constant po_2_07 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
    constant po_2_08 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
    constant po_2_09 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
    constant po_2_0A : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
    constant po_2_0B : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
    constant po_2_0C : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
    constant po_2_0D : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
    constant po_2_0E : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
    constant po_2_0F : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF

    -- content of po_3 --------------------------------------------------------------
    constant po_3_00 : BIT_VECTOR := X"E59EE9EEEEEEFFC3333EB999909EE09CEEBEB02664400000000000
    constant po_3_01 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFC9C9CEEBE99901290C0999129B90
    constant po_3_02 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
    constant po_3_03 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
    constant po_3_04 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
    constant po_3_05 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
    constant po_3_06 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
    constant po_3_07 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
    constant po_3_08 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
    constant po_3_09 : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
    constant po_3_0A : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
    constant po_3_0B : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
    constant po_3_0C : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
    constant po_3_0D : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
    constant po_3_0E : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
    constant po_3_0F : BIT_VECTOR := X"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF

    end prog_mem_content;
```

src/prog_mem_content.vhd

20 LISTING OF prog_mem_content.vhd

21 LISTING OF prog_mem.vhd

```
--------------------------------------------------------------------------------
--
-- Copyright (C) 2009, 2010 Dr. Juergen Sauermann
--
--  This code is free software: you can redistribute it and/or modify
--  it under the terms of the GNU General Public License as published by
--  the Free Software Foundation, either version 3 of the License, or
--  (at your option) any later version.
--
--  This code is distributed in the hope that it will be useful,
--  but WITHOUT ANY WARRANTY; without even the implied warranty of
--  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
--  GNU General Public License for more details.
--
--  You should have received a copy of the GNU General Public License
--  along with this code (see the file named COPYING).
--  If not, see http://www.gnu.org/licenses/.
--
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--
-- Module Name:    prog_mem - Behavioral
-- Create Date:    14:09:04 10/30/2009
-- Description:    the program memory of a CPU.
--
-----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

-- the content of the program memory.
--
use work.prog_mem_content.all;

entity prog_mem is
    port (  I_CLK       : in  std_logic;

            I_WAIT      : in  std_logic;
            I_PC        : in  std_logic_vector(15 downto 0); -- word address
            I_PM_ADR    : in  std_logic_vector(11 downto 0); -- byte address

            Q_OPC       : out std_logic_vector(31 downto 0);
            Q_PC        : out std_logic_vector(15 downto 0);
            Q_PM_DOUT   : out std_logic_vector( 7 downto 0));
end prog_mem;

architecture Behavioral of prog_mem is

constant zero_256 : bit_vector := X"00000000000000000000000000000000"
                                & X"00000000000000000000000000000000";

component RAMB4_S4_S4
    generic(INIT_00 : bit_vector := zero_256;
            INIT_01 : bit_vector := zero_256;
            INIT_02 : bit_vector := zero_256;
            INIT_03 : bit_vector := zero_256;
            INIT_04 : bit_vector := zero_256;
```

```
            INIT_05 : bit_vector := zero_256;
            INIT_06 : bit_vector := zero_256;
            INIT_07 : bit_vector := zero_256;
            INIT_08 : bit_vector := zero_256;
            INIT_09 : bit_vector := zero_256;
            INIT_0A : bit_vector := zero_256;
            INIT_0B : bit_vector := zero_256;
            INIT_0C : bit_vector := zero_256;
            INIT_0D : bit_vector := zero_256;
            INIT_0E : bit_vector := zero_256;
            INIT_0F : bit_vector := zero_256);

        port(   ADDRA   : in  std_logic_vector(9 downto 0);
                ADDRB   : in  std_logic_vector(9 downto 0);
                CLKA    : in  std_ulogic;
                CLKB    : in  std_ulogic;
                DIA     : in  std_logic_vector(3 downto 0);
                DIB     : in  std_logic_vector(3 downto 0);
                ENA     : in  std_ulogic;
                ENB     : in  std_ulogic;
                RSTA    : in  std_ulogic;
                RSTB    : in  std_ulogic;
                WEA     : in  std_ulogic;
                WEB     : in  std_ulogic;

                DOA     : out std_logic_vector(3 downto 0);
                DOB     : out std_logic_vector(3 downto 0));
    end component;

    signal M_OPC_E      : std_logic_vector(15 downto 0);
    signal M_OPC_O      : std_logic_vector(15 downto 0);
    signal M_PMD_E      : std_logic_vector(15 downto 0);
    signal M_PMD_O      : std_logic_vector(15 downto 0);

    signal L_WAIT_N     : std_logic;
    signal L_PC_0       : std_logic;
    signal L_PC_E       : std_logic_vector(10 downto 1);
    signal L_PC_O       : std_logic_vector(10 downto 1);
    signal L_PMD        : std_logic_vector(15 downto 0);
    signal L_PM_ADR_1_0 : std_logic_vector( 1 downto 0);

    begin

        pe_0 : RAMB4_S4_S4 ---------------------------------------------------------
        generic map(INIT_00 => pe_0_00, INIT_01 => pe_0_01, INIT_02 => pe_0_02,
                    INIT_03 => pe_0_03, INIT_04 => pe_0_04, INIT_05 => pe_0_05,
                    INIT_06 => pe_0_06, INIT_07 => pe_0_07, INIT_08 => pe_0_08,
                    INIT_09 => pe_0_09, INIT_0A => pe_0_0A, INIT_0B => pe_0_0B,
                    INIT_0C => pe_0_0C, INIT_0D => pe_0_0D, INIT_0E => pe_0_0E,
                    INIT_0F => pe_0_0F)
        port map(ADDRA => L_PC_E,                  ADDRB => I_PM_ADR(11 downto 2),
                 CLKA  => I_CLK,                   CLKB  => I_CLK,
                 DIA   => "0000",                  DIB   => "0000",
                 ENA   => L_WAIT_N,                ENB   => '1',
                 RSTA  => '0',                     RSTB  => '0',
                 WEA   => '0',                     WEB   => '0',
                 DOA   => M_OPC_E(3 downto 0),     DOB   => M_PMD_E(3 downto 0));

        pe_1 : RAMB4_S4_S4 ---------------------------------------------------------
        generic map(INIT_00 => pe_1_00, INIT_01 => pe_1_01, INIT_02 => pe_1_02,
                    INIT_03 => pe_1_03, INIT_04 => pe_1_04, INIT_05 => pe_1_05,
                    INIT_06 => pe_1_06, INIT_07 => pe_1_07, INIT_08 => pe_1_08,
```

```
                INIT_09 => pe_1_09, INIT_0A => pe_1_0A, INIT_0B => pe_1_0B,
                INIT_0C => pe_1_0C, INIT_0D => pe_1_0D, INIT_0E => pe_1_0E,
                INIT_0F => pe_1_0F)
    port map(ADDRA => L_PC_E,                   ADDRB => I_PM_ADR(11 downto 2),
             CLKA  => I_CLK,                    CLKB  => I_CLK,
             DIA   => "0000",                   DIB   => "0000",
             ENA   => L_WAIT_N,                 ENB   => '1',
             RSTA  => '0',                      RSTB  => '0',
             WEA   => '0',                      WEB   => '0',
             DOA   => M_OPC_E(7 downto 4),      DOB   => M_PMD_E(7 downto 4));

    pe_2 : RAMB4_S4_S4 ---------------------------------------------------------
    generic map(INIT_00 => pe_2_00, INIT_01 => pe_2_01, INIT_02 => pe_2_02,
                INIT_03 => pe_2_03, INIT_04 => pe_2_04, INIT_05 => pe_2_05,
                INIT_06 => pe_2_06, INIT_07 => pe_2_07, INIT_08 => pe_2_08,
                INIT_09 => pe_2_09, INIT_0A => pe_2_0A, INIT_0B => pe_2_0B,
                INIT_0C => pe_2_0C, INIT_0D => pe_2_0D, INIT_0E => pe_2_0E,
                INIT_0F => pe_2_0F)
    port map(ADDRA => L_PC_E,                   ADDRB => I_PM_ADR(11 downto 2),
             CLKA  => I_CLK,                    CLKB  => I_CLK,
             DIA   => "0000",                   DIB   => "0000",
             ENA   => L_WAIT_N,                 ENB   => '1',
             RSTA  => '0',                      RSTB  => '0',
             WEA   => '0',                      WEB   => '0',
             DOA   => M_OPC_E(11 downto 8),     DOB   => M_PMD_E(11 downto 8));

    pe_3 : RAMB4_S4_S4 ---------------------------------------------------------
    generic map(INIT_00 => pe_3_00, INIT_01 => pe_3_01, INIT_02 => pe_3_02,
                INIT_03 => pe_3_03, INIT_04 => pe_3_04, INIT_05 => pe_3_05,
                INIT_06 => pe_3_06, INIT_07 => pe_3_07, INIT_08 => pe_3_08,
                INIT_09 => pe_3_09, INIT_0A => pe_3_0A, INIT_0B => pe_3_0B,
                INIT_0C => pe_3_0C, INIT_0D => pe_3_0D, INIT_0E => pe_3_0E,
                INIT_0F => pe_3_0F)
    port map(ADDRA => L_PC_E,                   ADDRB => I_PM_ADR(11 downto 2),
             CLKA  => I_CLK,                    CLKB  => I_CLK,
             DIA   => "0000",                   DIB   => "0000",
             ENA   => L_WAIT_N,                 ENB   => '1',
             RSTA  => '0',                      RSTB  => '0',
             WEA   => '0',                      WEB   => '0',
             DOA   => M_OPC_E(15 downto 12),    DOB   => M_PMD_E(15 downto 12));

    po_0 : RAMB4_S4_S4 ---------------------------------------------------------
    generic map(INIT_00 => po_0_00, INIT_01 => po_0_01, INIT_02 => po_0_02,
                INIT_03 => po_0_03, INIT_04 => po_0_04, INIT_05 => po_0_05,
                INIT_06 => po_0_06, INIT_07 => po_0_07, INIT_08 => po_0_08,
                INIT_09 => po_0_09, INIT_0A => po_0_0A, INIT_0B => po_0_0B,
                INIT_0C => po_0_0C, INIT_0D => po_0_0D, INIT_0E => po_0_0E,
                INIT_0F => po_0_0F)
    port map(ADDRA => L_PC_O,                   ADDRB => I_PM_ADR(11 downto 2),
             CLKA  => I_CLK,                    CLKB  => I_CLK,
             DIA   => "0000",                   DIB   => "0000",
             ENA   => L_WAIT_N,                 ENB   => '1',
             RSTA  => '0',                      RSTB  => '0',
             WEA   => '0',                      WEB   => '0',
             DOA   => M_OPC_O(3 downto 0),      DOB   => M_PMD_O(3 downto 0));

    po_1 : RAMB4_S4_S4 ---------------------------------------------------------
    generic map(INIT_00 => po_1_00, INIT_01 => po_1_01, INIT_02 => po_1_02,
                INIT_03 => po_1_03, INIT_04 => po_1_04, INIT_05 => po_1_05,
                INIT_06 => po_1_06, INIT_07 => po_1_07, INIT_08 => po_1_08,
                INIT_09 => po_1_09, INIT_0A => po_1_0A, INIT_0B => po_1_0B,
                INIT_0C => po_1_0C, INIT_0D => po_1_0D, INIT_0E => po_1_0E,
```

```
                    INIT_0F => po_1_0F)
        port map(ADDRA => L_PC_O,                   ADDRB => I_PM_ADR(11 downto 2),
                 CLKA  => I_CLK,                    CLKB  => I_CLK,
                 DIA   => "0000",                   DIB   => "0000",
                 ENA   => L_WAIT_N,                 ENB   => '1',
                 RSTA  => '0',                      RSTB  => '0',
                 WEA   => '0',                      WEB   => '0',
                 DOA   => M_OPC_O(7 downto 4),      DOB   => M_PMD_O(7 downto 4));

        po_2 : RAMB4_S4_S4 ---------------------------------------------------------
        generic map(INIT_00 => po_2_00, INIT_01 => po_2_01, INIT_02 => po_2_02,
                    INIT_03 => po_2_03, INIT_04 => po_2_04, INIT_05 => po_2_05,
                    INIT_06 => po_2_06, INIT_07 => po_2_07, INIT_08 => po_2_08,
                    INIT_09 => po_2_09, INIT_0A => po_2_0A, INIT_0B => po_2_0B,
                    INIT_0C => po_2_0C, INIT_0D => po_2_0D, INIT_0E => po_2_0E,
                    INIT_0F => po_2_0F)
        port map(ADDRA => L_PC_O,                   ADDRB => I_PM_ADR(11 downto 2),
                 CLKA  => I_CLK,                    CLKB  => I_CLK,
                 DIA   => "0000",                   DIB   => "0000",
                 ENA   => L_WAIT_N,                 ENB   => '1',
                 RSTA  => '0',                      RSTB  => '0',
                 WEA   => '0',                      WEB   => '0',
                 DOA   => M_OPC_O(11 downto 8),     DOB   => M_PMD_O(11 downto 8));

        po_3 : RAMB4_S4_S4 ---------------------------------------------------------
        generic map(INIT_00 => po_3_00, INIT_01 => po_3_01, INIT_02 => po_3_02,
                    INIT_03 => po_3_03, INIT_04 => po_3_04, INIT_05 => po_3_05,
                    INIT_06 => po_3_06, INIT_07 => po_3_07, INIT_08 => po_3_08,
                    INIT_09 => po_3_09, INIT_0A => po_3_0A, INIT_0B => po_3_0B,
                    INIT_0C => po_3_0C, INIT_0D => po_3_0D, INIT_0E => po_3_0E,
                    INIT_0F => po_3_0F)
        port map(ADDRA => L_PC_O,                   ADDRB => I_PM_ADR(11 downto 2),
                 CLKA  => I_CLK,                    CLKB  => I_CLK,
                 DIA   => "0000",                   DIB   => "0000",
                 ENA   => L_WAIT_N,                 ENB   => '1',
                 RSTA  => '0',                      RSTB  => '0',
                 WEA   => '0',                      WEB   => '0',
                 DOA   => M_OPC_O(15 downto 12),    DOB   => M_PMD_O(15 downto 12));

        -- remember I_PC0 and I_PM_ADR for the output mux.
        --
        pc0: process(I_CLK)
        begin
            if (rising_edge(I_CLK)) then
                Q_PC      <= I_PC;
                L_PM_ADR_1_0      <= I_PM_ADR(1 downto 0);
                if ((I_WAIT = '0')) then
                    L_PC_0      <= I_PC(0);
                end if;
            end if;
        end process;

        L_WAIT_N      <= not I_WAIT;

        -- we use two memory blocks _E and _O (even and odd).
        -- This gives us a quad-port memory so that we can access
        -- I_PC, I_PC + 1, and PM simultaneously.
        --
        -- I_PC and I_PC + 1 are handled by port A of the memory while PM
        -- is handled by port B.
        --
        -- Q_OPC(15 ... 0) shall contain the word addressed by I_PC, while
```

```
        -- Q_OPC(31 ... 16) shall contain the word addressed by I_PC + 1.
        --
        -- There are two cases:
        --
        -- case A: I_PC     is even, thus I_PC + 1 is odd
        -- case B: I_PC + 1 is odd , thus I_PC is even
        --
        L_PC_O      <= I_PC(10 downto 1);
        L_PC_E      <= I_PC(10 downto 1) + ("000000000" & I_PC(0));
        Q_OPC(15 downto  0)      <= M_OPC_E when L_PC_0 = '0' else M_OPC_O;
        Q_OPC(31 downto 16)      <= M_OPC_E when L_PC_0 = '1' else M_OPC_O;

        L_PMD      <= M_PMD_E               when (L_PM_ADR_1_0(1) = '0') else M_PMD_O;
        Q_PM_DOUT      <= L_PMD(7 downto 0) when (L_PM_ADR_1_0(0) = '0')
                else L_PMD(15 downto 8);

    end Behavioral;

```

src/prog_mem.vhd

21 LISTING OF prog_mem.vhd

22 LISTING OF reg_16.vhd

```
--------------------------------------------------------------------------------
--
-- Copyright (C) 2009, 2010 Dr. Juergen Sauermann
--
--  This code is free software: you can redistribute it and/or modify
--  it under the terms of the GNU General Public License as published by
--  the Free Software Foundation, either version 3 of the License, or
--  (at your option) any later version.
--
--  This code is distributed in the hope that it will be useful,
--  but WITHOUT ANY WARRANTY; without even the implied warranty of
--  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
--  GNU General Public License for more details.
--
--  You should have received a copy of the GNU General Public License
--  along with this code (see the file named COPYING).
--  If not, see http://www.gnu.org/licenses/.
--
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--
-- Module Name:    Register - Behavioral
-- Create Date:    12:37:55 10/28/2009
-- Description:    a register pair of a CPU.
--
-----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity reg_16 is
    port (  I_CLK       : in  std_logic;

            I_D         : in  std_logic_vector (15 downto 0);
            I_WE        : in  std_logic_vector ( 1 downto 0);

            Q           : out std_logic_vector (15 downto 0));
end reg_16;

architecture Behavioral of reg_16 is

signal L                : std_logic_vector (15 downto 0) := X"7777";
begin

    process(I_CLK)
    begin
        if (rising_edge(I_CLK)) then
            if (I_WE(1) = '1') then
                L(15 downto 8)      <= I_D(15 downto 8);
            end if;
            if (I_WE(0) = '1') then
                L( 7 downto 0)      <= I_D( 7 downto 0);
            end if;
        end if;
    end process;

    Q      <= L;
```

```
end Behavioral;
```

src/reg_16.vhd

23 LISTING OF register_file.vhd

```
-------------------------------------------------------------------------------
--
-- Copyright (C) 2009, 2010 Dr. Juergen Sauermann
--
--  This code is free software: you can redistribute it and/or modify
--  it under the terms of the GNU General Public License as published by
--  the Free Software Foundation, either version 3 of the License, or
--  (at your option) any later version.
--
--  This code is distributed in the hope that it will be useful,
--  but WITHOUT ANY WARRANTY; without even the implied warranty of
--  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
--  GNU General Public License for more details.
--
--  You should have received a copy of the GNU General Public License
--  along with this code (see the file named COPYING).
--  If not, see http://www.gnu.org/licenses/.
--
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
--
-- Module Name:    RegisterFile - Behavioral
-- Create Date:    12:43:34 10/28/2009
-- Description:    a register file (16 register pairs) of a CPU.
--
-------------------------------------------------------------------------------
--
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

use work.common.ALL;

entity register_file is
    port (  I_CLK       : in  std_logic;

            I_AMOD      : in  std_logic_vector( 5 downto 0);
            I_COND      : in  std_logic_vector( 3 downto 0);
            I_DDDDD     : in  std_logic_vector( 4 downto 0);
            I_DIN       : in  std_logic_vector(15 downto 0);
            I_FLAGS     : in  std_logic_vector( 7 downto 0);
            I_IMM       : in  std_logic_vector(15 downto 0);
            I_RRRR      : in  std_logic_vector( 4 downto 1);
            I_WE_01     : in  std_logic;
            I_WE_D      : in  std_logic_vector( 1 downto 0);
            I_WE_F      : in  std_logic;
            I_WE_M      : in  std_logic;
            I_WE_XYZS   : in  std_logic;

            Q_ADR       : out std_logic_vector(15 downto 0);
            Q_CC        : out std_logic;
            Q_D         : out std_logic_vector(15 downto 0);
            Q_FLAGS     : out std_logic_vector( 7 downto 0);
            Q_R         : out std_logic_vector(15 downto 0);
            Q_S         : out std_logic_vector( 7 downto 0);
            Q_Z         : out std_logic_vector(15 downto 0));
end register_file;
```

```
architecture Behavioral of register_file is

component reg_16
    port (  I_CLK       : in    std_logic;

            I_D         : in    std_logic_vector(15 downto 0);
            I_WE        : in    std_logic_vector( 1 downto 0);

            Q           : out   std_logic_vector(15 downto 0));
end component;

signal R_R00            : std_logic_vector(15 downto 0);
signal R_R02            : std_logic_vector(15 downto 0);
signal R_R04            : std_logic_vector(15 downto 0);
signal R_R06            : std_logic_vector(15 downto 0);
signal R_R08            : std_logic_vector(15 downto 0);
signal R_R10            : std_logic_vector(15 downto 0);
signal R_R12            : std_logic_vector(15 downto 0);
signal R_R14            : std_logic_vector(15 downto 0);
signal R_R16            : std_logic_vector(15 downto 0);
signal R_R18            : std_logic_vector(15 downto 0);
signal R_R20            : std_logic_vector(15 downto 0);
signal R_R22            : std_logic_vector(15 downto 0);
signal R_R24            : std_logic_vector(15 downto 0);
signal R_R26            : std_logic_vector(15 downto 0);
signal R_R28            : std_logic_vector(15 downto 0);
signal R_R30            : std_logic_vector(15 downto 0);
signal R_SP             : std_logic_vector(15 downto 0);    -- stack pointer

component status_reg is
    port (  I_CLK       : in  std_logic;

            I_COND      : in  std_logic_vector ( 3 downto 0);
            I_DIN       : in  std_logic_vector ( 7 downto 0);
            I_FLAGS     : in  std_logic_vector ( 7 downto 0);
            I_WE_F      : in  std_logic;
            I_WE_SR     : in  std_logic;

            Q           : out std_logic_vector ( 7 downto 0);
            Q_CC        : out std_logic);
end component;

signal S_FLAGS          : std_logic_vector( 7 downto 0);

signal L_ADR            : std_logic_vector(15 downto 0);
signal L_BASE           : std_logic_vector(15 downto 0);
signal L_DDDD           : std_logic_vector( 4 downto 1);
signal L_DSP            : std_logic_vector(15 downto 0);
signal L_DX             : std_logic_vector(15 downto 0);
signal L_DY             : std_logic_vector(15 downto 0);
signal L_DZ             : std_logic_vector(15 downto 0);
signal L_PRE            : std_logic_vector(15 downto 0);
signal L_POST           : std_logic_vector(15 downto 0);
signal L_S              : std_logic_vector(15 downto 0);
signal L_WE_SP_AMOD     : std_logic;
signal L_WE             : std_logic_vector(31 downto 0);
signal L_WE_A           : std_logic;
signal L_WE_D           : std_logic_vector(31 downto 0);
signal L_WE_D2          : std_logic_vector( 1 downto 0);
signal L_WE_DD          : std_logic_vector(31 downto 0);
signal L_WE_IO          : std_logic_vector(31 downto 0);
```

```
    signal L_WE_MISC        : std_logic_vector(31 downto 0);
    signal L_WE_X           : std_logic;
    signal L_WE_Y           : std_logic;
    signal L_WE_Z           : std_logic;
    signal L_WE_SP          : std_logic_vector( 1 downto 0);
    signal L_WE_SR          : std_logic;
    signal L_XYZS           : std_logic_vector(15 downto 0);

    begin

        r00: reg_16 port map(I_CLK => I_CLK, I_WE => L_WE( 1 downto  0), I_D => I_DIN, Q => R
        r02: reg_16 port map(I_CLK => I_CLK, I_WE => L_WE( 3 downto  2), I_D => I_DIN, Q => R
        r04: reg_16 port map(I_CLK => I_CLK, I_WE => L_WE( 5 downto  4), I_D => I_DIN, Q => R
        r06: reg_16 port map(I_CLK => I_CLK, I_WE => L_WE( 7 downto  6), I_D => I_DIN, Q => R
        r08: reg_16 port map(I_CLK => I_CLK, I_WE => L_WE( 9 downto  8), I_D => I_DIN, Q => R
        r10: reg_16 port map(I_CLK => I_CLK, I_WE => L_WE(11 downto 10), I_D => I_DIN, Q => R
        r12: reg_16 port map(I_CLK => I_CLK, I_WE => L_WE(13 downto 12), I_D => I_DIN, Q => R
        r14: reg_16 port map(I_CLK => I_CLK, I_WE => L_WE(15 downto 14), I_D => I_DIN, Q => R
        r16: reg_16 port map(I_CLK => I_CLK, I_WE => L_WE(17 downto 16), I_D => I_DIN, Q => R
        r18: reg_16 port map(I_CLK => I_CLK, I_WE => L_WE(19 downto 18), I_D => I_DIN, Q => R
        r20: reg_16 port map(I_CLK => I_CLK, I_WE => L_WE(21 downto 20), I_D => I_DIN, Q => R
        r22: reg_16 port map(I_CLK => I_CLK, I_WE => L_WE(23 downto 22), I_D => I_DIN, Q => R
        r24: reg_16 port map(I_CLK => I_CLK, I_WE => L_WE(25 downto 24), I_D => I_DIN, Q => R
        r26: reg_16 port map(I_CLK => I_CLK, I_WE => L_WE(27 downto 26), I_D => L_DX,  Q => R
        r28: reg_16 port map(I_CLK => I_CLK, I_WE => L_WE(29 downto 28), I_D => L_DY,  Q => R
        r30: reg_16 port map(I_CLK => I_CLK, I_WE => L_WE(31 downto 30), I_D => L_DZ,  Q => R
        sp:  reg_16 port map(I_CLK => I_CLK, I_WE => L_WE_SP,            I_D => L_DSP, Q => R

        sr: status_reg
        port map(   I_CLK       => I_CLK,
                    I_COND      => I_COND,
                    I_DIN       => I_DIN(7 downto 0),
                    I_FLAGS     => I_FLAGS,
                    I_WE_F      => I_WE_F,
                    I_WE_SR     => L_WE_SR,
                    Q           => S_FLAGS,
                    Q_CC        => Q_CC);

        -- The output of the register selected by L_ADR.
        --
        process(R_R00, R_R02, R_R04, R_R06, R_R08, R_R10, R_R12, R_R14,
                R_R16, R_R18, R_R20, R_R22, R_R24, R_R26, R_R28, R_R30,
                R_SP, S_FLAGS, L_ADR(6 downto 1))
        begin
            case L_ADR(6 downto 1) is
                when "000000" => L_S      <= R_R00;
                when "000001" => L_S      <= R_R02;
                when "000010" => L_S      <= R_R04;
                when "000011" => L_S      <= R_R06;
                when "000100" => L_S      <= R_R08;
                when "000101" => L_S      <= R_R10;
                when "000110" => L_S      <= R_R12;
                when "000111" => L_S      <= R_R14;
                when "001000" => L_S      <= R_R16;
                when "001001" => L_S      <= R_R18;
                when "001010" => L_S      <= R_R20;
                when "001011" => L_S      <= R_R22;
                when "001100" => L_S      <= R_R24;
                when "001101" => L_S      <= R_R26;
                when "001110" => L_S      <= R_R28;
                when "001111" => L_S      <= R_R30;
                when "101111" => L_S      <= R_SP ( 7 downto 0) & X"00";     -- SPL
```

```
                when others   => L_S      <= S_FLAGS & R_SP (15 downto 8);   -- SR/SPH
            end case;
        end process;

        -- The output of the register pair selected by I_DDDDD.
        --
        process(R_R00, R_R02, R_R04, R_R06, R_R08, R_R10, R_R12, R_R14,
                R_R16, R_R18, R_R20, R_R22, R_R24, R_R26, R_R28, R_R30,
                I_DDDDD(4 downto 1))
        begin
            case I_DDDDD(4 downto 1) is
                when "0000" => Q_D      <= R_R00;
                when "0001" => Q_D      <= R_R02;
                when "0010" => Q_D      <= R_R04;
                when "0011" => Q_D      <= R_R06;
                when "0100" => Q_D      <= R_R08;
                when "0101" => Q_D      <= R_R10;
                when "0110" => Q_D      <= R_R12;
                when "0111" => Q_D      <= R_R14;
                when "1000" => Q_D      <= R_R16;
                when "1001" => Q_D      <= R_R18;
                when "1010" => Q_D      <= R_R20;
                when "1011" => Q_D      <= R_R22;
                when "1100" => Q_D      <= R_R24;
                when "1101" => Q_D      <= R_R26;
                when "1110" => Q_D      <= R_R28;
                when others => Q_D      <= R_R30;
            end case;
        end process;

        -- The output of the register pair selected by I_RRRR.
        --
        process(R_R00, R_R02, R_R04,  R_R06, R_R08, R_R10, R_R12, R_R14,
                R_R16, R_R18, R_R20, R_R22, R_R24, R_R26, R_R28, R_R30, I_RRRR)
        begin
            case I_RRRR is
                when "0000" => Q_R      <= R_R00;
                when "0001" => Q_R      <= R_R02;
                when "0010" => Q_R      <= R_R04;
                when "0011" => Q_R      <= R_R06;
                when "0100" => Q_R      <= R_R08;
                when "0101" => Q_R      <= R_R10;
                when "0110" => Q_R      <= R_R12;
                when "0111" => Q_R      <= R_R14;
                when "1000" => Q_R      <= R_R16;
                when "1001" => Q_R      <= R_R18;
                when "1010" => Q_R      <= R_R20;
                when "1011" => Q_R      <= R_R22;
                when "1100" => Q_R      <= R_R24;
                when "1101" => Q_R      <= R_R26;
                when "1110" => Q_R      <= R_R28;
                when others => Q_R      <= R_R30;
            end case;
        end process;

        -- the base value of the X/Y/Z/SP register as per I_AMOD.
        --
        process(I_AMOD(2 downto 0), I_IMM, R_SP, R_R26, R_R28, R_R30)
        begin
            case I_AMOD(2 downto 0) is
                when AS_SP  => L_BASE      <= R_SP;
                when AS_Z   => L_BASE      <= R_R30;
```

```
            when AS_Y   => L_BASE      <= R_R28;
            when AS_X   => L_BASE      <= R_R26;
            when AS_IMM => L_BASE      <= I_IMM;
            when others => L_BASE      <= X"0000";
        end case;
    end process;

    -- the value of the X/Y/Z/SP register after a potential PRE-decrement
    -- (by 1 or 2) and POST-increment (by 1 or 2).
    --
    process(I_AMOD(5 downto 3), I_IMM)
    begin
        case I_AMOD(5 downto 3) is
            when AO_0   => L_PRE      <= X"0000";    L_POST <= X"0000";
            when AO_i   => L_PRE      <= X"0000";    L_POST <= X"0001";
            when AO_ii  => L_PRE      <= X"0000";    L_POST <= X"0002";
            when AO_q   => L_PRE      <= I_IMM;      L_POST <= X"0000";
            when AO_d   => L_PRE      <= X"FFFF";    L_POST <= X"FFFF";
            when AO_dd  => L_PRE      <= X"FFFE";    L_POST <= X"FFFE";
            when others => L_PRE      <= X"0000";    L_POST <= X"0000";
        end case;
    end process;

    L_XYZS      <= L_BASE + L_POST;
    L_ADR       <= L_BASE + L_PRE;

    L_WE_A      <= I_WE_M when (L_ADR(15 downto 5) = "00000000000") else '0';
    L_WE_SR         <= I_WE_M when (L_ADR = X"005F") else '0';
    L_WE_SP_AMOD      <= I_WE_XYZS when (I_AMOD(2 downto 0) = AS_SP) else '0';
    L_WE_SP(1)      <= I_WE_M when (L_ADR = X"005E") else L_WE_SP_AMOD;
    L_WE_SP(0)      <= I_WE_M when (L_ADR = X"005D") else L_WE_SP_AMOD;

    L_DX       <= L_XYZS when (L_WE_MISC(26) = '1')        else I_DIN;
    L_DY       <= L_XYZS when (L_WE_MISC(28) = '1')        else I_DIN;
    L_DZ       <= L_XYZS when (L_WE_MISC(30) = '1')        else I_DIN;
    L_DSP      <= L_XYZS when (I_AMOD(3 downto 0) = AM_WS) else I_DIN;

    -- the WE signals for the differen registers.
    --
    -- case 1: write to an 8-bit register addressed by DDDDD.
    --
    -- I_WE_D(0) = '1' and I_DDDDD matches,
    --
    L_WE_D( 0)      <= I_WE_D(0) when (I_DDDDD = "00000") else '0';
    L_WE_D( 1)      <= I_WE_D(0) when (I_DDDDD = "00001") else '0';
    L_WE_D( 2)      <= I_WE_D(0) when (I_DDDDD = "00010") else '0';
    L_WE_D( 3)      <= I_WE_D(0) when (I_DDDDD = "00011") else '0';
    L_WE_D( 4)      <= I_WE_D(0) when (I_DDDDD = "00100") else '0';
    L_WE_D( 5)      <= I_WE_D(0) when (I_DDDDD = "00101") else '0';
    L_WE_D( 6)      <= I_WE_D(0) when (I_DDDDD = "00110") else '0';
    L_WE_D( 7)      <= I_WE_D(0) when (I_DDDDD = "00111") else '0';
    L_WE_D( 8)      <= I_WE_D(0) when (I_DDDDD = "01000") else '0';
    L_WE_D( 9)      <= I_WE_D(0) when (I_DDDDD = "01001") else '0';
    L_WE_D(10)      <= I_WE_D(0) when (I_DDDDD = "01010") else '0';
    L_WE_D(11)      <= I_WE_D(0) when (I_DDDDD = "01011") else '0';
    L_WE_D(12)      <= I_WE_D(0) when (I_DDDDD = "01100") else '0';
    L_WE_D(13)      <= I_WE_D(0) when (I_DDDDD = "01101") else '0';
    L_WE_D(14)      <= I_WE_D(0) when (I_DDDDD = "01110") else '0';
    L_WE_D(15)      <= I_WE_D(0) when (I_DDDDD = "01111") else '0';
    L_WE_D(16)      <= I_WE_D(0) when (I_DDDDD = "10000") else '0';
    L_WE_D(17)      <= I_WE_D(0) when (I_DDDDD = "10001") else '0';
    L_WE_D(18)      <= I_WE_D(0) when (I_DDDDD = "10010") else '0';
```

```
        L_WE_D(19)       <= I_WE_D(0) when (I_DDDDD = "10011") else '0';
        L_WE_D(20)       <= I_WE_D(0) when (I_DDDDD = "10100") else '0';
        L_WE_D(21)       <= I_WE_D(0) when (I_DDDDD = "10101") else '0';
        L_WE_D(22)       <= I_WE_D(0) when (I_DDDDD = "10110") else '0';
        L_WE_D(23)       <= I_WE_D(0) when (I_DDDDD = "10111") else '0';
        L_WE_D(24)       <= I_WE_D(0) when (I_DDDDD = "11000") else '0';
        L_WE_D(25)       <= I_WE_D(0) when (I_DDDDD = "11001") else '0';
        L_WE_D(26)       <= I_WE_D(0) when (I_DDDDD = "11010") else '0';
        L_WE_D(27)       <= I_WE_D(0) when (I_DDDDD = "11011") else '0';
        L_WE_D(28)       <= I_WE_D(0) when (I_DDDDD = "11100") else '0';
        L_WE_D(29)       <= I_WE_D(0) when (I_DDDDD = "11101") else '0';
        L_WE_D(30)       <= I_WE_D(0) when (I_DDDDD = "11110") else '0';
        L_WE_D(31)       <= I_WE_D(0) when (I_DDDDD = "11111") else '0';

        --
        -- case 2: write to a 16-bit register pair addressed by DDDD.
        --
        -- I_WE_DD(1) = '1' and L_DDDD matches,
        --
        L_DDDD       <= I_DDDDD(4 downto 1);
        L_WE_D2       <= I_WE_D(1) & I_WE_D(1);
        L_WE_DD( 1 downto  0)       <= L_WE_D2 when (L_DDDD = "0000") else "00";
        L_WE_DD( 3 downto  2)       <= L_WE_D2 when (L_DDDD = "0001") else "00";
        L_WE_DD( 5 downto  4)       <= L_WE_D2 when (L_DDDD = "0010") else "00";
        L_WE_DD( 7 downto  6)       <= L_WE_D2 when (L_DDDD = "0011") else "00";
        L_WE_DD( 9 downto  8)       <= L_WE_D2 when (L_DDDD = "0100") else "00";
        L_WE_DD(11 downto 10)       <= L_WE_D2 when (L_DDDD = "0101") else "00";
        L_WE_DD(13 downto 12)       <= L_WE_D2 when (L_DDDD = "0110") else "00";
        L_WE_DD(15 downto 14)       <= L_WE_D2 when (L_DDDD = "0111") else "00";
        L_WE_DD(17 downto 16)       <= L_WE_D2 when (L_DDDD = "1000") else "00";
        L_WE_DD(19 downto 18)       <= L_WE_D2 when (L_DDDD = "1001") else "00";
        L_WE_DD(21 downto 20)       <= L_WE_D2 when (L_DDDD = "1010") else "00";
        L_WE_DD(23 downto 22)       <= L_WE_D2 when (L_DDDD = "1011") else "00";
        L_WE_DD(25 downto 24)       <= L_WE_D2 when (L_DDDD = "1100") else "00";
        L_WE_DD(27 downto 26)       <= L_WE_D2 when (L_DDDD = "1101") else "00";
        L_WE_DD(29 downto 28)       <= L_WE_D2 when (L_DDDD = "1110") else "00";
        L_WE_DD(31 downto 30)       <= L_WE_D2 when (L_DDDD = "1111") else "00";

        --
        -- case 3: write to an 8-bit register pair addressed by an I/O address.
        --
        -- L_WE_A = '1' and L_ADR(4 downto 0) matches
        --
        L_WE_IO( 0)       <= L_WE_A when (L_ADR(4 downto 0) = "00000") else '0';
        L_WE_IO( 1)       <= L_WE_A when (L_ADR(4 downto 0) = "00001") else '0';
        L_WE_IO( 2)       <= L_WE_A when (L_ADR(4 downto 0) = "00010") else '0';
        L_WE_IO( 3)       <= L_WE_A when (L_ADR(4 downto 0) = "00011") else '0';
        L_WE_IO( 4)       <= L_WE_A when (L_ADR(4 downto 0) = "00100") else '0';
        L_WE_IO( 5)       <= L_WE_A when (L_ADR(4 downto 0) = "00101") else '0';
        L_WE_IO( 6)       <= L_WE_A when (L_ADR(4 downto 0) = "00110") else '0';
        L_WE_IO( 7)       <= L_WE_A when (L_ADR(4 downto 0) = "00111") else '0';
        L_WE_IO( 8)       <= L_WE_A when (L_ADR(4 downto 0) = "01000") else '0';
        L_WE_IO( 9)       <= L_WE_A when (L_ADR(4 downto 0) = "01001") else '0';
        L_WE_IO(10)       <= L_WE_A when (L_ADR(4 downto 0) = "01010") else '0';
        L_WE_IO(11)       <= L_WE_A when (L_ADR(4 downto 0) = "01011") else '0';
        L_WE_IO(12)       <= L_WE_A when (L_ADR(4 downto 0) = "01100") else '0';
        L_WE_IO(13)       <= L_WE_A when (L_ADR(4 downto 0) = "01101") else '0';
        L_WE_IO(14)       <= L_WE_A when (L_ADR(4 downto 0) = "01110") else '0';
        L_WE_IO(15)       <= L_WE_A when (L_ADR(4 downto 0) = "01111") else '0';
        L_WE_IO(16)       <= L_WE_A when (L_ADR(4 downto 0) = "10000") else '0';
        L_WE_IO(17)       <= L_WE_A when (L_ADR(4 downto 0) = "10001") else '0';
        L_WE_IO(18)       <= L_WE_A when (L_ADR(4 downto 0) = "10010") else '0';
```

```
        L_WE_IO(19)       <= L_WE_A when (L_ADR(4 downto 0) = "10011") else '0';
        L_WE_IO(20)       <= L_WE_A when (L_ADR(4 downto 0) = "10100") else '0';
        L_WE_IO(21)       <= L_WE_A when (L_ADR(4 downto 0) = "10101") else '0';
        L_WE_IO(22)       <= L_WE_A when (L_ADR(4 downto 0) = "10110") else '0';
        L_WE_IO(23)       <= L_WE_A when (L_ADR(4 downto 0) = "10111") else '0';
        L_WE_IO(24)       <= L_WE_A when (L_ADR(4 downto 0) = "11000") else '0';
        L_WE_IO(25)       <= L_WE_A when (L_ADR(4 downto 0) = "11001") else '0';
        L_WE_IO(26)       <= L_WE_A when (L_ADR(4 downto 0) = "11010") else '0';
        L_WE_IO(27)       <= L_WE_A when (L_ADR(4 downto 0) = "11011") else '0';
        L_WE_IO(28)       <= L_WE_A when (L_ADR(4 downto 0) = "11100") else '0';
        L_WE_IO(29)       <= L_WE_A when (L_ADR(4 downto 0) = "11101") else '0';
        L_WE_IO(30)       <= L_WE_A when (L_ADR(4 downto 0) = "11110") else '0';
        L_WE_IO(31)       <= L_WE_A when (L_ADR(4 downto 0) = "11111") else '0';

        -- case 4 special cases.
        -- 4a. WE_01 for register pair 0/1 (multiplication opcode).
        -- 4b. I_WE_XYZS for X (register pairs 26/27) and I_AMOD matches
        -- 4c. I_WE_XYZS for Y (register pairs 28/29) and I_AMOD matches
        -- 4d. I_WE_XYZS for Z (register pairs 30/31) and I_AMOD matches
        --
        L_WE_X       <= I_WE_XYZS when (I_AMOD(3 downto 0) = AM_WX) else '0';
        L_WE_Y       <= I_WE_XYZS when (I_AMOD(3 downto 0) = AM_WY) else '0';
        L_WE_Z       <= I_WE_XYZS when (I_AMOD(3 downto 0) = AM_WZ) else '0';
        L_WE_MISC       <= L_WE_Z & L_WE_Z &      -- -Z and Z+ address modes  r30
                     L_WE_Y & L_WE_Y &      -- -Y and Y+ address modes  r28
                     L_WE_X & L_WE_X &      -- -X and X+ address modes  r26
                     X"000000" &            -- never                    r24 - r02
                     I_WE_01 & I_WE_01;     -- multiplication result    r00

        L_WE       <= L_WE_D or L_WE_DD or L_WE_IO or L_WE_MISC;

        Q_S       <= L_S( 7 downto 0) when (L_ADR(0) = '0') else L_S(15 downto 8);
        Q_FLAGS       <= S_FLAGS;
        Q_Z       <= R_R30;
        Q_ADR       <= L_ADR;

    end Behavioral;

```

src/register_file.vhd

24 LISTING OF segment7.vhd

```
--------------------------------------------------------------------------------
--
-- Copyright (C) 2009, 2010 Dr. Juergen Sauermann
--
--  This code is free software: you can redistribute it and/or modify
--  it under the terms of the GNU General Public License as published by
--  the Free Software Foundation, either version 3 of the License, or
--  (at your option) any later version.
--
--  This code is distributed in the hope that it will be useful,
--  but WITHOUT ANY WARRANTY; without even the implied warranty of
--  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
--  GNU General Public License for more details.
--
--  You should have received a copy of the GNU General Public License
--  along with this code (see the file named COPYING).
--  If not, see http://www.gnu.org/licenses/.
--
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--
-- Module Name:    segment7 - Behavioral
-- Create Date:    12:52:16 11/11/2009
-- Description:    a 7 segment LED display interface.
--
--------------------------------------------------------------------------------
--
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity segment7 is
    port ( I_CLK        : in  std_logic;

           I_CLR        : in  std_logic;
           I_OPC        : in  std_logic_vector(15 downto 0);
           I_PC         : in  std_logic_vector(15 downto 0);

           Q_7_SEGMENT : out std_logic_vector( 6 downto 0));
end segment7;

--      Signal      Loc Alt
---------------------------
--      SEG_LED(0)  V3  A
--      SEG_LED(1)  V4  B
--      SEG_LED(2)  W3  C
--      SEG_LED(3)  T4  D
--      SEG_LED(4)  T3  E
--      SEG_LED(5)  U3  F
--      SEG_LED(6)  U4  G
--
architecture Behavioral of segment7 is

function lmap(VAL: std_logic_vector( 3 downto 0))
         return std_logic_vector is
begin
    case VAL is         --      6543210
```

```
            when "0000" =>  return "0111111";   -- 0
            when "0001" =>  return "0000110";   -- 1
            when "0010" =>  return "1011011";   -- 2
            when "0011" =>  return "1001111";   -- 3
            when "0100" =>  return "1100110";   -- 4    ----A----       ----0----
            when "0101" =>  return "1101101";   -- 5    |       |       |       |
            when "0110" =>  return "1111101";   -- 6    F       B       5       1
            when "0111" =>  return "0000111";   -- 7    |       |       |       |
            when "1000" =>  return "1111111";   -- 8    +---G---+       +---6---+
            when "1001" =>  return "1101111";   -- 9    |       |       |       |
            when "1010" =>  return "1110111";   -- A    E       C       4       2
            when "1011" =>  return "1111100";   -- b    |       |       |       |
            when "1100" =>  return "0111001";   -- C    ----D----       ----3----
            when "1101" =>  return "1011110";   -- d
            when "1110" =>  return "1111001";   -- E
            when others =>  return "1110001";   -- F
        end case;
    end;

    signal L_CNT            : std_logic_vector(27 downto 0);
    signal L_OPC            : std_logic_vector(15 downto 0);
    signal L_PC             : std_logic_vector(15 downto 0);
    signal L_POS            : std_logic_vector( 3 downto 0);

    begin

        process(I_CLK)    -- 20 MHz
        begin
            if (rising_edge(I_CLK)) then
                if (I_CLR = '1') then
                    L_POS      <= "0000";
                    L_CNT      <= X"0000000";
                    Q_7_SEGMENT      <= "1111111";
                else
                    L_CNT      <= L_CNT + X"0000001";
                    if (L_CNT =  X"0C00000") then
                        Q_7_SEGMENT      <= "1111111";      -- blank
                    elsif (L_CNT =  X"1000000") then
                        L_CNT      <= X"0000000";
                        L_POS      <= L_POS + "0001";
                        case L_POS is
                            when "0000" =>  -- blank
                                Q_7_SEGMENT      <= "1111111";
                            when "0001" =>
                                L_PC      <= I_PC;       -- sample PC
                                L_OPC      <= I_OPC;     -- sample OPC
                                Q_7_SEGMENT      <= not lmap(L_PC(15 downto 12));
                            when "0010" =>
                                Q_7_SEGMENT      <= not lmap(L_PC(11 downto  8));
                            when "0011" =>
                                Q_7_SEGMENT      <= not lmap(L_PC( 7 downto  4));
                            when "0100" =>
                                Q_7_SEGMENT      <= not lmap(L_PC( 3 downto  0));
                            when "0101" =>  -- minus
                                Q_7_SEGMENT      <= "0111111";
                            when "0110" =>
                                Q_7_SEGMENT      <= not lmap(L_OPC(15 downto 12));
                            when "0111" =>
                                Q_7_SEGMENT      <= not lmap(L_OPC(11 downto  8));
                            when "1000" =>
                                Q_7_SEGMENT      <= not lmap(L_OPC( 7 downto  4));
                            when "1001" =>
```

```
                                Q_7_SEGMENT      <= not lmap(L_OPC( 3 downto  0));
                                L_POS      <= "0000";
                            when others =>
                                L_POS      <= "0000";
                        end case;
                    end if;
                end if;
            end if;
        end process;

    end Behavioral;

```

src/segment7.vhd

25 LISTING OF status_reg.vhd

```
-------------------------------------------------------------------------------
--
-- Copyright (C) 2009, 2010 Dr. Juergen Sauermann
--
--  This code is free software: you can redistribute it and/or modify
--  it under the terms of the GNU General Public License as published by
--  the Free Software Foundation, either version 3 of the License, or
--  (at your option) any later version.
--
--  This code is distributed in the hope that it will be useful,
--  but WITHOUT ANY WARRANTY; without even the implied warranty of
--  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
--  GNU General Public License for more details.
--
--  You should have received a copy of the GNU General Public License
--  along with this code (see the file named COPYING).
--  If not, see http://www.gnu.org/licenses/.
--
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
--
-- Module Name:    Register - Behavioral
-- Create Date:    16:15:54 12/26/2009
-- Description:    the status register of a CPU.
--
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity status_reg is
    port (  I_CLK       : in  std_logic;

            I_COND      : in  std_logic_vector ( 3 downto 0);
            I_DIN       : in  std_logic_vector ( 7 downto 0);
            I_FLAGS     : in  std_logic_vector ( 7 downto 0);
            I_WE_F      : in  std_logic;
            I_WE_SR     : in  std_logic;

            Q           : out std_logic_vector ( 7 downto 0);
            Q_CC        : out std_logic);
end status_reg;

architecture Behavioral of status_reg is

signal L                : std_logic_vector ( 7 downto 0);
begin

    process(I_CLK)
    begin
        if (rising_edge(I_CLK)) then
            if (I_WE_F = '1') then          -- write flags (from ALU)
                L       <= I_FLAGS;
            elsif (I_WE_SR = '1') then      -- write I/O
                L       <= I_DIN;
            end if;
        end if;
```

```
        end process;

        cond: process(I_COND, L)
        begin
            case I_COND(2 downto 0) is
                when "000"  => Q_CC      <= L(0) xor I_COND(3);
                when "001"  => Q_CC      <= L(1) xor I_COND(3);
                when "010"  => Q_CC      <= L(2) xor I_COND(3);
                when "011"  => Q_CC      <= L(3) xor I_COND(3);
                when "100"  => Q_CC      <= L(4) xor I_COND(3);
                when "101"  => Q_CC      <= L(5) xor I_COND(3);
                when "110"  => Q_CC      <= L(6) xor I_COND(3);
                when others => Q_CC      <= L(7) xor I_COND(3);
            end case;
        end process;

        Q       <= L;

    end Behavioral;

```

src/status_reg.vhd

26 LISTING OF uart_rx.vhd

```
--------------------------------------------------------------------------------
--
-- Copyright (C) 2009, 2010 Dr. Juergen Sauermann
--
--  This code is free software: you can redistribute it and/or modify
--  it under the terms of the GNU General Public License as published by
--  the Free Software Foundation, either version 3 of the License, or
--  (at your option) any later version.
--
--  This code is distributed in the hope that it will be useful,
--  but WITHOUT ANY WARRANTY; without even the implied warranty of
--  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
--  GNU General Public License for more details.
--
--  You should have received a copy of the GNU General Public License
--  along with this code (see the file named COPYING).
--  If not, see http://www.gnu.org/licenses/.
--
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--
-- Create Date:    14:22:28 11/07/2009
-- Design Name:
-- Module Name:    uart_rx - Behavioral
-- Description:    a UART receiver.
--
--------------------------------------------------------------------------------
--
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity uart_rx is
    PORT(   I_CLK       : in  std_logic;
            I_CLR       : in  std_logic;
            I_CE_16     : in  std_logic;            -- 16 times baud rate
            I_RX        : in  std_logic;            -- Serial input line

            Q_DATA      : out std_logic_vector(7 downto 0);
            Q_FLAG      : out std_logic);       -- toggle on every byte received
end uart_rx;

architecture Behavioral of uart_rx is

signal L_POSITION       : std_logic_vector(7 downto 0);     --  sample position
signal L_BUF            : std_logic_vector(9 downto 0);
signal L_FLAG           : std_logic;
signal L_SERIN          : std_logic;                -- double clock the input
signal L_SER_HOT        : std_logic;                -- double clock the input

begin

    -- double clock the input data...
    --
    process(I_CLK)
    begin
        if (rising_edge(I_CLK)) then
```

```
                if (I_CLR = '1') then
                    L_SERIN      <= '1';
                    L_SER_HOT      <= '1';
                else
                    L_SERIN        <= I_RX;
                    L_SER_HOT      <= L_SERIN;
                end if;
            end if;
        end process;

        process(I_CLK, L_POSITION)
            variable START_BIT : boolean;
            variable STOP_BIT  : boolean;
            variable STOP_POS  : boolean;

        begin
        START_BIT := L_POSITION(7 downto 4) = X"0";
        STOP_BIT  := L_POSITION(7 downto 4) = X"9";
        STOP_POS  := STOP_BIT and L_POSITION(3 downto 2) = "11"; -- 3/4 of stop bit

            if (rising_edge(I_CLK)) then
                if (I_CLR = '1') then
                    L_FLAG      <= '0';
                    L_POSITION      <= X"00";    -- idle
                    L_BUF           <= "1111111111";
                    Q_DATA          <= "00000000";
                elsif (I_CE_16 = '1') then
                    if (L_POSITION = X"00") then            -- uart idle
                        L_BUF       <= "1111111111";
                        if (L_SER_HOT = '0')  then          -- start bit received
                            L_POSITION      <= X"01";
                        end if;
                    else
                        L_POSITION      <= L_POSITION + X"01";
                        if (L_POSITION(3 downto 0) = "0111") then       -- 1/2 bit
                            L_BUF      <= L_SER_HOT & L_BUF(9 downto 1);     -- sample data
                            --
                            -- validate start bit
                            --
                            if (START_BIT and L_SER_HOT = '1') then     -- 1/2 start bit
                                L_POSITION      <= X"00";
                            end if;

                            if (STOP_BIT) then                          -- 1/2 stop bit
                                Q_DATA      <= L_BUF(9 downto 2);
                            end if;
                        elsif (STOP_POS) then                       -- 3/4 stop bit
                            L_FLAG      <= L_FLAG xor (L_BUF(9) and not L_BUF(0));
                            L_POSITION      <= X"00";
                        end if;
                    end if;
                end if;
            end if;
        end process;

        Q_FLAG      <= L_FLAG;

    end Behavioral;
```

src/uart_rx.vhd

27 LISTING OF uart_tx.vhd

```
-------------------------------------------------------------------------------
--
-- Copyright (C) 2009, 2010 Dr. Juergen Sauermann
--
--  This code is free software: you can redistribute it and/or modify
--  it under the terms of the GNU General Public License as published by
--  the Free Software Foundation, either version 3 of the License, or
--  (at your option) any later version.
--
--  This code is distributed in the hope that it will be useful,
--  but WITHOUT ANY WARRANTY; without even the implied warranty of
--  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
--  GNU General Public License for more details.
--
--  You should have received a copy of the GNU General Public License
--  along with this code (see the file named COPYING).
--  If not, see http://www.gnu.org/licenses/.
--
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
--
-- Module Name:    uart_tx - Behavioral
-- Create Date:    14:21:59 11/07/2009
-- Description:    a UART receiver.
--
-------------------------------------------------------------------------------
--
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity uart_tx is
    port(   I_CLK       : in  std_logic;
            I_CLR       : in  std_logic;            -- RESET
            I_CE_1      : in  std_logic;            -- BAUD rate clock enable
            I_DATA      : in  std_logic_vector(7 downto 0);   -- DATA to be sent
            I_FLAG      : in  std_logic;            -- toggle to send data
            Q_TX        : out std_logic;            -- Serial output line
            Q_FLAG      : out std_logic);           -- Transmitting Flag
end uart_tx;

architecture Behavioral of uart_tx is

signal L_BUF            : std_logic_vector(7 downto 0);
signal L_TODO           : std_logic_vector(3 downto 0);     -- bits to send
signal L_FLAG           : std_logic;

begin

    process(I_CLK)
    begin
        if (rising_edge(I_CLK)) then
            if (I_CLR = '1') then
                Q_TX        <= '1';
                L_BUF       <= "11111111";
                L_TODO      <= "0000";
                L_FLAG      <= I_FLAG;                  -- idle
```

```
                elsif (I_CE_1 = '1') then
                    if (L_TODO /= "0000") then          -- transmitting
                        Q_TX      <= L_BUF(0);                -- next bit
                        L_BUF          <= '1' & L_BUF(7 downto 1);
                        if (L_TODO = "0001") then
                            L_FLAG      <= I_FLAG;
                        end if;
                        L_TODO      <= L_TODO - "0001";
                    elsif (L_FLAG /= I_FLAG) then       -- new byte
                        Q_TX      <= '0';                     -- start bit
                        L_BUF      <= I_DATA;                 -- data bits
                        L_TODO      <= "1001";
                    end if;
                end if;
            end if;
        end process;

        Q_FLAG      <= L_FLAG;

    end Behavioral;
```

src/uart_tx.vhd

28 LISTING OF RAMB4_S4_S4.vhd

```
-------------------------------------------------------------------------------
--
-- Copyright (C) 2009, 2010 Dr. Juergen Sauermann
--
--  This code is free software: you can redistribute it and/or modify
--  it under the terms of the GNU General Public License as published by
--  the Free Software Foundation, either version 3 of the License, or
--  (at your option) any later version.
--
--  This code is distributed in the hope that it will be useful,
--  but WITHOUT ANY WARRANTY; without even the implied warranty of
--  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
--  GNU General Public License for more details.
--
--  You should have received a copy of the GNU General Public License
--  along with this code (see the file named COPYING).
--  If not, see http://www.gnu.org/licenses/.
--
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
--
-- Module Name:    prog_mem - Behavioral
-- Create Date:    14:09:04 10/30/2009
-- Description:    a block memory module
--
-------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity RAMB4_S4_S4 is
    generic(INIT_00 : bit_vector := X"00000000000000000000000000000000"
                                  &  "00000000000000000000000000000000";
            INIT_01 : bit_vector := X"00000000000000000000000000000000"
                                  & X"00000000000000000000000000000000";
            INIT_02 : bit_vector := X"00000000000000000000000000000000"
                                  & X"00000000000000000000000000000000";
            INIT_03 : bit_vector := X"00000000000000000000000000000000"
                                  & X"00000000000000000000000000000000";
            INIT_04 : bit_vector := X"00000000000000000000000000000000"
                                  & X"00000000000000000000000000000000";
            INIT_05 : bit_vector := X"00000000000000000000000000000000"
                                  & X"00000000000000000000000000000000";
            INIT_06 : bit_vector := X"00000000000000000000000000000000"
                                  & X"00000000000000000000000000000000";
            INIT_07 : bit_vector := X"00000000000000000000000000000000"
                                  & X"00000000000000000000000000000000";
            INIT_08 : bit_vector := X"00000000000000000000000000000000"
                                  & X"00000000000000000000000000000000";
            INIT_09 : bit_vector := X"00000000000000000000000000000000"
                                  & X"00000000000000000000000000000000";
            INIT_0A : bit_vector := X"00000000000000000000000000000000"
                                  & X"00000000000000000000000000000000";
            INIT_0B : bit_vector := X"00000000000000000000000000000000"
                                  & X"00000000000000000000000000000000";
            INIT_0C : bit_vector := X"00000000000000000000000000000000"
```

```
                                 & X"00000000000000000000000000000000";
                INIT_0D : bit_vector := X"00000000000000000000000000000000"
                                 & X"00000000000000000000000000000000";
                INIT_0E : bit_vector := X"00000000000000000000000000000000"
                                 & X"00000000000000000000000000000000";
                INIT_0F : bit_vector := X"00000000000000000000000000000000"
                                 & X"00000000000000000000000000000000");

        port(   ADDRA   : in  std_logic_vector(9 downto 0);
                ADDRB   : in  std_logic_vector(9 downto 0);
                CLKA    : in  std_ulogic;
                CLKB    : in  std_ulogic;
                DIA     : in  std_logic_vector(3 downto 0);
                DIB     : in  std_logic_vector(3 downto 0);
                ENA     : in  std_ulogic;
                ENB     : in  std_ulogic;
                RSTA    : in  std_ulogic;
                RSTB    : in  std_ulogic;
                WEA     : in  std_ulogic;
                WEB     : in  std_ulogic;

                DOA     : out std_logic_vector(3 downto 0);
                DOB     : out std_logic_vector(3 downto 0));
    end RAMB4_S4_S4;

    architecture Behavioral of RAMB4_S4_S4 is

    function cv(A : bit) return std_logic is
    begin
       if (A = '1') then return '1';
       else              return '0';
       end if;
    end;

    function cv1(A : std_logic) return bit is
    begin
       if (A = '1') then return '1';
       else              return '0';
       end if;
    end;

    signal DATA : bit_vector(4095 downto 0) :=
        INIT_0F & INIT_0E & INIT_0D & INIT_0C & INIT_0B & INIT_0A & INIT_09 & INIT_08 &
        INIT_07 & INIT_06 & INIT_05 & INIT_04 & INIT_03 & INIT_02 & INIT_01 & INIT_00;

    begin

        process(CLKA, CLKB)
        begin
            if (rising_edge(CLKA)) then
                if (ENA = '1') then
                    DOA(3)      <= cv(DATA(conv_integer(ADDRA & "11")));
                    DOA(2)      <= cv(DATA(conv_integer(ADDRA & "10")));
                    DOA(1)      <= cv(DATA(conv_integer(ADDRA & "01")));
                    DOA(0)      <= cv(DATA(conv_integer(ADDRA & "00")));
                    if (WEA = '1') then
                        DATA(conv_integer(ADDRA & "11"))      <= cv1(DIA(3));
                        DATA(conv_integer(ADDRA & "10"))      <= cv1(DIA(2));
                        DATA(conv_integer(ADDRA & "01"))      <= cv1(DIA(1));
                        DATA(conv_integer(ADDRA & "00"))      <= cv1(DIA(0));
                    end if;
               end if;
```

```
            end if;

            if (rising_edge(CLKB)) then
                if (ENB = '1') then
                    DOB(3)      <= cv(DATA(conv_integer(ADDRB & "11")));
                    DOB(2)      <= cv(DATA(conv_integer(ADDRB & "10")));
                    DOB(1)      <= cv(DATA(conv_integer(ADDRB & "01")));
                    DOB(0)      <= cv(DATA(conv_integer(ADDRB & "00")));
                    if (WEB = '1') then
                        DATA(conv_integer(ADDRB & "11"))      <= cv1(DIB(3));
                        DATA(conv_integer(ADDRB & "10"))      <= cv1(DIB(2));
                        DATA(conv_integer(ADDRB & "01"))      <= cv1(DIB(1));
                        DATA(conv_integer(ADDRB & "00"))      <= cv1(DIB(0));
                    end if;
                end if;
            end if;
        end process;

    end Behavioral;

```

test/RAMB4_S4_S4.vhd

28 LISTING OF RAMB4_S4_S4.vhd

29 LISTING OF test_tb.vhd

```
-------------------------------------------------------------------------------
--
-- Copyright (C) 2009, 2010 Dr. Juergen Sauermann
--
--  This code is free software: you can redistribute it and/or modify
--  it under the terms of the GNU General Public License as published by
--  the Free Software Foundation, either version 3 of the License, or
--  (at your option) any later version.
--
--  This code is distributed in the hope that it will be useful,
--  but WITHOUT ANY WARRANTY; without even the implied warranty of
--  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
--  GNU General Public License for more details.
--
--  You should have received a copy of the GNU General Public License
--  along with this code (see the file named COPYING).
--  If not, see http://www.gnu.org/licenses/.
--
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
--
-- Module Name:    alu - Behavioral
-- Create Date:    16:47:24 12/29/2009
-- Description:    arithmetic logic unit of a CPU
--
-------------------------------------------------------------------------------
--
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity testbench is
end testbench;

architecture Behavioral of testbench is

component avr_fpga
    port (  I_CLK_100   : in  std_logic;
            I_SWITCH    : in  std_logic_vector(9 downto 0);
            I_RX        : in  std_logic;

            Q_7_SEGMENT : out std_logic_vector(6 downto 0);
            Q_LEDS      : out std_logic_vector(3 downto 0);
            Q_TX        : out std_logic);
end component;

signal L_CLK_100            : std_logic;
signal L_LEDS               : std_logic_vector(3 downto 0);
signal L_7_SEGMENT          : std_logic_vector(6 downto 0);
signal L_RX                 : std_logic;
signal L_SWITCH             : std_logic_vector(9 downto 0);
signal L_TX                 : std_logic;

signal  L_CLK_COUNT         : integer := 0;

begin

```

```
        fpga: avr_fpga
        port map(   I_CLK_100   => L_CLK_100,
                    I_SWITCH    => L_SWITCH,
                    I_RX        => L_RX,

                    Q_LEDS      => L_LEDS,
                    Q_7_SEGMENT => L_7_SEGMENT,
                    Q_TX        => L_TX);

        process -- clock process for CLK_100,
        begin
            clock_loop : loop
                L_CLK_100      <= transport '0';
                wait for 5 ns;

                L_CLK_100      <= transport '1';
                wait for 5 ns;
            end loop clock_loop;
        end process;

        process(L_CLK_100)
        begin
            if (rising_edge(L_CLK_100)) then
                case L_CLK_COUNT is
                    when 0 => L_SWITCH      <= "0011100000";   L_RX <= '0';
                    when 2 => L_SWITCH(9 downto 8)      <= "11";
                    when others =>
                end case;
                L_CLK_COUNT      <= L_CLK_COUNT + 1;
            end if;
        end process;
    end Behavioral;

```

test/test_tb.vhd

30 LISTING OF avr_fpga.ucf

```
NET     I_CLK_100       PERIOD = 10 ns;
NET     L_CLK           PERIOD = 35 ns;

NET     I_CLK_100       TNM_NET = I_CLK_100;
NET     L_CLK           TNM_NET = L_CLK;

NET     I_CLK_100       LOC = AA12;
NET     I_RX            LOC = M3;
NET     Q_TX            LOC = M4;

# 7 segment LED display
#
NET     Q_7_SEGMENT       LOC = V3;
NET     Q_7_SEGMENT       LOC = V4;
NET     Q_7_SEGMENT       LOC = W3;
NET     Q_7_SEGMENT       LOC = T4;
NET     Q_7_SEGMENT       LOC = T3;
NET     Q_7_SEGMENT       LOC = U3;
NET     Q_7_SEGMENT       LOC = U4;

# single LEDs
#
NET     Q_LEDS            LOC = N1;
NET     Q_LEDS            LOC = N2;
NET     Q_LEDS            LOC = P1;
NET     Q_LEDS            LOC = P2;

# DIP switch(0 ... 7) and two pushbuttons (8, 9)
#
NET     I_SWITCH          LOC = H2;
NET     I_SWITCH          LOC = H1;
NET     I_SWITCH          LOC = J2;
NET     I_SWITCH          LOC = J1;
NET     I_SWITCH          LOC = K2;
NET     I_SWITCH          LOC = K1;
NET     I_SWITCH          LOC = L2;
NET     I_SWITCH          LOC = L1;
NET     I_SWITCH          LOC = R1;
NET     I_SWITCH          LOC = R2;

NET     I_SWITCH          PULLUP;
```

src/avr_fpga.ucf

30 LISTING OF avr_fpga.ucf

31 LISTING OF Makefile

```
PROJECT=avr_core

# the vhdl source files (except testbench)
#
FILES           += src/*.vhd

# the testbench sources and binary.
#
SIMFILES        = test/test_tb.vhd test/RAMB4_S4_S4.vhd
SIMTOP          = testbench

# When to stop the simulation
#
# GHDL_SIM_OPT  = --assert-level=error
GHDL_SIM_OPT    = --stop-time=40us

SIMDIR          = simu

FLAGS           = --ieee=synopsys --warn-no-vital-generic -fexplicit --std=93c

all:
	make compile
	make run 2>& 1 | grep -v std_logic_arith
	make view

compile:
	@mkdir -p simu
	@echo -----------------------------------------------------------------
	ghdl -i $(FLAGS) --workdir=simu --work=work $(SIMFILES) $(FILES)
	@echo
	@echo -----------------------------------------------------------------
	ghdl -m $(FLAGS) --workdir=simu --work=work $(SIMTOP)
	@echo
	@mv $(SIMTOP) simu/$(SIMTOP)

run:
	@$(SIMDIR)/$(SIMTOP) $(GHDL_SIM_OPT) --vcdgz=$(SIMDIR)/$(SIMTOP).vcdgz

view:
	gunzip --stdout $(SIMDIR)/$(SIMTOP).vcdgz | gtkwave --vcd gtkwave.save

clean:
	ghdl --clean --workdir=simu
```

Makefile

32 LISTING OF hello.c

```
#include "stdint.h"
#include "avr/io.h"
#include "avr/pgmspace.h"

#undef F_CPU
#define F_CPU 25000000UL
#include "util/delay.h"


     //-----------------------------------------------------------------------//
    //                                                                       //
   //   print char cc on UART.                                              //
  //    return number of chars printed (i.e. 1).                           //
 //                                                                       //
//-----------------------------------------------------------------------//
uint8_t
uart_putc(uint8_t cc)
{
    while ((UCSRA & (1      <<UDRE)) == 0)      ;
    UDR = cc;
    return 1;
}

     //-----------------------------------------------------------------------//
    //                                                                       //
   //   print char cc on 7 segment display.                                 //
  //    return number of chars printed (i.e. 1).                           //
 //                                                                       //
//-----------------------------------------------------------------------//
// The segments of the display are encoded like this:
//
//
//      segment     PORT B
//      name        Bit number
//      ----A----   ----0----
//      |       |   |       |
//      F       B   5       1
//      |       |   |       |
//      ----G----   ----6----
//      |       |   |       |
//      E       C   4       2
//      |       |   |       |
//      ----D----   ----3----
//
//-----------------------------------------------------------------------------

#define SEG7(G, F, E, D, C, B, A)   (~(G      <
```
app/hello.c

32 LISTING OF hello.c

33 LISTING OF make_mem.cc

```
#include "assert.h"
#include "stdio.h"
#include "stdint.h"
#include "string.h"

const char * hex_file = 0;
const char * vhdl_file = 0;

uint8_t buffer[0x10000];

//-----------------------------------------------------------------------------
uint32_t
get_byte(const char *  cp)
{
uint32_t value;
const char cc[3] = { cp[0], cp[1], 0 };
const int cnt = sscanf(cc, "%X", &value);
   assert(cnt == 1);
   return value;
}
//-----------------------------------------------------------------------------
void
read_file(FILE * in)
{
   memset(buffer, 0xFF, sizeof(buffer));
char line[200];
   for (;;)
       {
         const char * s = fgets(line, sizeof(line) - 2, in);
         if (s == 0)   return;
         assert(*s++ == ':');
         const uint32_t len     = get_byte(s);
         const uint32_t ah      = get_byte(s + 2);
         const uint32_t al      = get_byte(s + 4);
         const uint32_t rectype = get_byte(s + 6);
         const char * d = s + 8;
         const uint32_t addr = ah      <<8 | al;

         uint32_t csum = len + ah + al + rectype;
         assert((addr + len)       <= 0x10000);
         for (uint32_t l = 0; l      <len; ++l)
             {
               const uint32_t byte = get_byte(d);
               d += 2;
               buffer[addr + l] = byte;
               csum += byte;
             }

         csum = 0xFF & -csum;
         const uint32_t sum = get_byte(d);
         assert(sum == csum);
       }
}
//-----------------------------------------------------------------------------
void
write_vector(FILE * out, bool odd, uint32_t mem, uint32_t v)
{
const uint8_t * base = buffer;
```

```

   // total memory is 2 even bytes, 2 odd bytes, 2 even bytes, ...
   //
   if (odd)   base += 2;

   // total memory is 4 kByte organized into 8 memories.
   // thus each of the 16 vectors covers 256 bytes.
   //
   base += v*256;

   // memories 0 and 1 are the low byte of the opcode while
   // memories 2 and 3 are the high byte.
   //
   if (mem >= 2)   ++base;

const char * px = odd ? "po" : "pe";
   fprintf(out, "constant %s_%u_%2.2X : BIT_VECTOR := X\"", px, mem, v);
   for (int32_t d = 63; d >= 0; --d)
       {
         uint32_t q = base[4*d];
         if (mem & 1)   q >>= 4;     // high nibble
         else           q &= 0x0F;   // low nibble
         fprintf(out, "%X", q);
       }

   fprintf(out, "\";\r\n");
}
//-------------------------------------------------------------------------------
void
write_mem(FILE * out, bool odd, uint32_t mem)
{
const char * px = odd ? "po" : "pe";

   fprintf(out, "-- content of %s_%u --------------------------------------"
                "--------------------------------------------\r\n", px, mem);

   for (uint32_t v = 0; v        <16; ++v)
       write_vector(out, odd, mem, v);

   fprintf(out, "\r\n");
}
//-------------------------------------------------------------------------------
void
write_file(FILE * out)
{
   fprintf(out,
"\r\n"
"library IEEE;\r\n"
"use IEEE.STD_LOGIC_1164.all;\r\n"
"\r\n"
"package prog_mem_content is\r\n"
"\r\n");

   for (uint32_t m = 0; m        <4; ++m)
       write_mem(out, false, m);

   for (uint32_t m = 0; m        <4; ++m)
       write_mem(out, true,  m);

   fprintf(out,
"end prog_mem_content;\r\n"
"\r\n");
```

```
}
//-----------------------------------------------------------------------------
int
main(int argc, char * argv[])
{
   if (argc > 1)   hex_file = argv[1];
   if (argc > 2)   vhdl_file = argv[2];

FILE * in = stdin;
   if (hex_file)   in = fopen(hex_file, "r");
   assert(in);
   read_file(in);
   fclose(in);

FILE * out = stdout;
   if (vhdl_file)   out = fopen(vhdl_file, "w");
   write_file(out);
   assert(out);
}
//-----------------------------------------------------------------------------
```

tools/make_mem.cc

34 LISTING OF end_conv.cc

```
#include "assert.h"
#include "ctype.h"
#include "stdio.h"
#include "string.h"

//-----------------------------------------------------------------------------
int
main(int argc, const char * argv)
{
char buffer[2000];
int pc, val, val2;

   for (;;)
       {
         char * s = fgets(buffer, sizeof(buffer) - 2, stdin);
         if (s == 0)   return 0;

         // map lines '  xx:' and 'xxxxxxxx; to 2* the hex value.
         //
         if (
             (isxdigit(s[0]) || s[0] == ' ') &&
             (isxdigit(s[1]) || s[1] == ' ') &&
             (isxdigit(s[2]) || s[2] == ' ') &&
              isxdigit(s[3]) && s[4] == ':')   // '  xx:'
            {
              assert(1 == sscanf(s, " %x:", &pc));
              if (pc & 1)       printf("%4X+:", pc/2);
              else              printf("%4X:", pc/2);
              s += 5;
            }
         else if (isxdigit(s[0]) && isxdigit(s[1]) && isxdigit(s[2]) &&
                  isxdigit(s[3]) && isxdigit(s[4]) && isxdigit(s[5]) &&
                  isxdigit(s[6]) && isxdigit(s[7]))             // 'xxxxxxxx'
            {
              assert(1 == sscanf(s, "%x", &pc));
              if (pc & 1)   printf("%8.8X+:", pc/2);
              else          printf("%8.8X:", pc/2);
              s += 8;
            }
         else                             // other: copy verbatim
            {
              printf("%s", s);
              continue;
            }

          while (isblank(*s))   printf("%c", *s++);

          // endian swap.
          //
          while (isxdigit(s[0]) &&
                 isxdigit(s[1]) &&
                          s[2] == ' ' &&
                 isxdigit(s[3]) &&
                 isxdigit(s[4]) &&
                          s[5] == ' ')
             {
              assert(2 == sscanf(s, "%x %x ", &val, &val2));
              printf("%2.2X%2.2X  ", val2, val);
```

```
                  s += 6;
                 }

            char * s1 = strstr(s, ".+");
            char * s2 = strstr(s, ".-");
            if (s1)
               {
                 assert(1 == sscanf(s1 + 2, "%d", &val));
                 assert((val & 1) == 0);
                 sprintf(s1, " 0x%X", (pc + val)/2 + 1);
                 printf(s);
                 s = s1 + strlen(s1) + 1;
               }
            else if (s2)
               {
                 assert(1 == sscanf(s2 + 2, "%d", &val));
                 assert((val & 1) == 0);
                 sprintf(s2, " 0x%X", (pc - val)/2 + 1);
                 printf(s);
                 s = s2 + strlen(s2) + 1;
               }

            printf("%s", s);
          }
}
//-----------------------------------------------------------------------------
```

tools/end_conv.cc